

Qt 5.10 GUI完全参考手册

作者：黄勇

2018-8-18(第一版)

出版社：??????

前言

本书作者：黄邦勇帅(原名：黄勇)，QQ：42444472（读者意见可发至 QQ）

本书需具有 C++语言基础，若对 C++语言不熟习可参阅《C++语法详解》一书(电子工业出版社，黄勇，2017-6-30)

本书适合希望使用 Qt C++开发跨平台应用程序的读者阅读。

本书对 QWidget 的各个子类及其相关的框架结构作了详细的介绍，主要包括，元对象系统，信号和槽，Qt 事件，Qt 主窗口，布局管理及焦点系统，对话框，模型/视图框架，拖放和剪贴板，Qt 文本系统，Qt 界面外观，Qt 2D 绘图和 Qt 的输入输出。

本书是一本讲解 Qt 原理性的书籍，对 Qt 的各种原理讲解透彻、深入、细致。书中的示例都是完整的实例程序。

本书章节划分合理，思路清楚，不显得杂乱无章，对每个章节的知识点讲解全面细致，各章节内容不相互重合累赘，方便复习查阅。相对于其他书而言对 Qt 的细节原理讲解更透彻简单易懂。

本人能力有限，其中难免有误解之处，望指出更正。

声明：禁止抄袭，复印，转载本书内容，本书作者拥有完全版权。

第 1 章 Qt 快速入门目录

[1.1 Qt 简介](#)

[1.2 Qt Creator 的使用](#)

[1.2.1 qt 的下载及安装注意事项](#)

[1.2.2 Qt Create 的界面介绍](#)

[1.2.3 手动添加套件](#)

[1.3 使用 Qt Creator 编写一个 C++程序](#)

[1.3.1 创建一个空项目](#)

[1.3.2 添加 C++代码](#)

[1.3.3 运行程序](#)

[1.3.4 使用 QtCreator 编辑器的一些技巧](#)

[1.4 使用 Qt Creator 编写 Qt 程序](#)

[1.4.1 方法 1：使用代码编写 Qt 程序](#)

[1.4.2 方法 2：使用界面编辑器（即设计模式）编写 Qt 程序](#)

[1.4.3 方法 3：使用 Qt 设计师界面类编写 Qt 程序](#)

[1.4.4 方法 4：使用“Qt Widgets Application”编写 Qt 程序](#)

[1.4.5 方法 5：使用记事本及 Qt 命令程序编写 Qt 程序](#)

[1.5 发布程序](#)

[1.5.1 编译后生成的各种文件简介](#)

[1.5.2 手动发布程序](#)

[1.5.3 使用 windeployqt 工具部署文件（仅限 windows）](#)

[1.5.4 静态编译和动态编译](#)

[1.6 Qt 的重要文件简介](#)

[1.6.1 项目文件（pro 文件）及其语法](#)

[1.6.2 moc 简介](#)

[1.6.3 pro、pri、prf 文件简介](#)

[1.7 Qt 框架结构简介](#)

[1.7.1 Qt 基本框架](#)

[1.7.2 Qt5 模块架构](#)

[1.8 帮助文档的使用](#)

第 2 章 Qt 元对象系统、信号和槽及事件目录

[2.1 元对象系统](#)

[2.1.1 元对象系统基本概念](#)

[2.1.2 Q_OBJECT 宏](#)

[2.1.3 使用 Qt Creator 启动元对象系统](#)

[2.2.4 在命令行启动元对象系统](#)

[2.2 元对象](#)

[2.2.1 QByteArray 类简介](#)

[2.2.2 元对象系统与反射机制](#)

- [2.2.3 使用反射机制获取类对象的成员函数的信息](#)
- [2.2.4 使用反射机制获取与类相关的信息](#)
- [2.3 属性系统](#)
 - [2.3.1 属性基础](#)
 - [2.3.2 QVariant 类](#)
 - [2.3.3 使用 QObject 类中的成员函数存取属性值与动态属性](#)
 - [2.3.4 使用反射机制获取属性的信息](#)
- [2.4 信号与槽](#)
 - [2.4.1 信号和槽原理](#)
 - [2.4.2 创建信号和槽](#)
 - [2.4.3 信号和槽的关联（连接）](#)
 - [2.4.4 断开信号和槽的关联](#)
 - [2.4.5 signals、slots、emit 关键字原型](#)
- [2.5 对象树与生命期](#)
 - [2.5.1 组合模式与对象树](#)
 - [2.5.2 QObject 类、对象树、生命期](#)
- [2.6 事件](#)
 - [2.6.1 QApplication、QGuiApplication、QCoreApplication](#)
 - [2.6.2 Qt 对事件的描述及分类](#)
 - [2.6.3 事件的传递（或分发）及处理](#)
 - [2.6.4 事件的接受和忽略](#)
 - [2.6.5 事件过滤器](#)
 - [2.6.6 自定义事件与事件的发送](#)
 - [2.6.7 事件的传递顺序总结](#)
 - [2.6.8 鼠标和键盘事件共同使用的类及函数](#)
 - [2.6.9 鼠标事件](#)
 - [2.6.10 键盘事件](#)

第 3 章 Qt 窗口及 QWidget 类目录

- [3.1 QtWidgets 模块及窗口基本概念](#)
 - [3.1.1 QtWidgets 模块中的类的继承图及帮助文档的使用](#)
 - [3.1.2 Qt 中窗口的基本概念](#)
 - [3.1.3 Qt 实现窗口及其部件的原理](#)
 - [3.1.4 部件构造函数参数 f 的取值](#)
 - [3.1.5 部件的删除](#)
 - [3.1.6 QFlags 模板类详解](#)
- [3.2 QWidget 类](#)
 - [3.2.1 基础](#)
 - [3.2.2 与部件大小和位置有关的成员](#)
 - [3.2.3 窗口大小的限制与默认大小](#)
 - [3.2.4 窗口的状态（最大化最小化）](#)
 - [3.2.5 窗口的显示及可见性](#)

- [3.2.6 标题、透明度、启用、禁用](#)
- [3.2.7 窗口标志、设置其他属性](#)
- [3.2.8 获取窗口部件、设置父部件](#)
- [3.2.9 鼠标光标](#)
- [3.2.10 其他](#)

第 4 章 Qt 常用部件目录

[4.1 按钮部件](#)

- [4.1.1 共同特性](#)
- [4.1.2 QAbstractButton 抽象类](#)
- [4.1.3 QPushButton 类\(标准按钮\)](#)
- [4.1.4 QCheckBox 类\(复选按钮\)](#)
- [4.1.5 QRadioButton 类\(单选按钮\)](#)
- [4.1.6 QToolButton 类\(工具按钮\)](#)

[4.2 容器部件](#)

- [4.2.1 QDialogButtonBox 按钮框](#)
- [4.2.2 QButtonGroup 按钮组](#)
- [4.2.3 QGroupBox 组框](#)

[4.3 带边框的部件](#)

- [4.3.1 QFrame 类](#)
- [4.3.2 QLabel 标签](#)
- [4.3.3 QLCDNumberLCD 数字](#)

[4.4 输入部件](#)

- [4.4.1 QComboBox 下拉列表、组合框](#)
- [4.4.2 QLineEdit 行编辑器](#)
- [4.4.3 QValidator 抽象类、验证器及其子类](#)

[4.5 旋转框、微调按钮](#)

- [4.5.1 QAbstractSpinBox 旋转框或微调框](#)
- [4.5.2 QSpinBox 类](#)
- [4.5.3 QDoubleSpinBox 类](#)

[4.6 时间系统](#)

- [4.6.1 时间系统基础](#)
- [4.6.2 QDate 类](#)
- [4.6.3 QTime 类](#)
- [4.6.4 QDateTime 类](#)
- [4.6.5 QDateTimeEdit 类](#)
- [4.6.6 QDateEdit 类和 QTimeEdit 类](#)
- [4.6.7 QTimer 计时器](#)
- [4.6.8 QCalendarWidget 日历](#)

[部件公用枚举](#)

第 5 章 Qt 布局管理及焦点系统目录

[5.1 布局原理](#)

[5.1.1 布局基础](#)

[5.1.2 部件拉伸 \(Stretch\) 原理及大小策略](#)

[5.1.3 大小约束 \(主窗口最大最小大小的设置\)](#)

[5.1.4 内容边距 \(ContentsMargins\)、间距 \(spacing\)、QSpace](#)

[5.1.5 嵌套布局及布局位于容器中](#)

[5.2 各布局管理器类](#)

[5.2.1 QBoxLayout 及其子类 \(盒式布局\)](#)

[5.2.2 QGridLayout 类 \(网格布局\)](#)

[5.2.3 QFormLayout 类 \(表单布局\)](#)

[5.3 实现多页面切换](#)

[5.3.1 QStackedLayout 类 \(分组布局或栈布局\)](#)

[5.3.2 QStackedWidget 类](#)

[5.3.3 QTabBar 类 \(选项卡栏\)](#)

[5.3.4 QTabWidget 类 \(选项卡部件\)](#)

[5.4 QSplitter 分离器\(或分隔符\)](#)

[5.4.1 QSplitter 类 \(分离器\)](#)

[5.4.2 QSplitterHandle 类 \(分界线\)](#)

[5.5 自定义布局管理器](#)

[5.5.1 QLayout 抽象类中的公有成员函数](#)

[5.5.2 QLayoutItem、QSpacerItem、QWidgetItem 类](#)

[5.5.3 自定义布局的实现](#)

[5.6 Qt 焦点系统](#)

[5.6.1 焦点链 \(焦点循环\)](#)

[5.6.2 获取焦点信息](#)

[5.6.3 焦点代理 \(FocusProxy\)](#)

[5.6.4 设置焦点及焦点策略](#)

[5.6.5 焦点事件](#)

[5.6.6 自定义焦点循环](#)

[5.6.7 QFocusFrame 类 \(焦点框, 自定义焦点框的外形\)](#)

[本章公用枚举](#)

第 6 章 Qt 对话框目录

[6.1 QDialog 类\(对话框\)](#)

[6.1.1 对话框与窗口](#)

[6.1.2 模态与非模态对话框](#)

[6.1.3 对话框返回的信息](#)

[6.1.4 对话框与窗口的关闭和隐藏](#)

- [6.2 QMessageBox 类\(消息对话框\)](#)
- [6.3 QErrorMessage 类\(错误消息对话框\)](#)
- [6.4 QColorDialog 类\(颜色对话框\)](#)
- [6.5 QFontDialog 类\(字体对话框\)](#)
- [6.6 QFileDialog 类\(文件对话框\)](#)
 - [6.6.1 文件对话框基础](#)
 - [6.6.2 QFileDialog 类中的属性](#)
 - [6.6.3 文件过滤器](#)
 - [6.6.4 QFileDialog 类中的函数](#)
 - [6.6.5 QFileDialog 类中的信号](#)
- [6.7 QInputDialog 类\(输入对话框\)](#)
 - [6.7.1 输入对话框基础](#)
 - [6.7.2 QInputDialog 类中的属性](#)
 - [6.7.3 QInputDialog 类中的函数](#)
 - [6.7.4 QInputDialog 类中的信号](#)
- [6.8 QProgressDialog 类\(进度对话框\)和 QProgressBar\(进度条\)](#)
 - [6.8.1 进度条原理](#)
 - [6.8.2 QProgressDialog 类\(进度对话框\)](#)
 - [6.8.3 QProgressBar 类\(进度条\)](#)
- [6.9 QWizard 类\(向导\)和 QWizardPage 类\(向导页\)](#)
 - [6.9.1 向导基础](#)
 - [6.9.2 向导外观](#)
 - [6.9.3 向导中的按钮](#)
 - [6.9.4 向导中的页面](#)
 - [6.9.5 验证页面中的内容](#)
 - [6.9.6 各页面间的通信\(字段\)](#)
 - [6.9.7 实现非线性向导](#)
 - [6.9.8 QWizard 类中的属性](#)
 - [6.9.9 QWizard 类中的函数](#)
 - [6.9.10 QWizardPage 类中的属性和函数](#)

第 7 章 Qt 主窗口目录

- [7.1 QMainWindow 类主窗口基础](#)
- [7.2 QMenu 类、QMenuBar 类、QAction 类基础](#)
 - [7.2.1 基本概念](#)
 - [7.2.2 创建菜单的方法](#)
 - [7.2.3 部件的所有权](#)
 - [7.2.4 QAction 动作基础](#)
- [7.3 QShortcut 类、快捷键](#)
 - [7.3.1 快捷键基础](#)
 - [7.3.1 QShortcut 类中的属性](#)
 - [7.3.2 QShortcut 类中的函数](#)
- [7.4 QKeySequence 类、键序列](#)
 - [7.4.1 键序列基础](#)
 - [7.4.2 QKeySequence 类中的枚举](#)
 - [7.4.3 QKeySequence 类中的函数](#)
- [7.5 QAction 类、QActionGroup 类](#)

- [7.5.1 动作基本规则](#)
- [7.5.2 QAction 类中的属性](#)
- [7.5.3 QAction 类中的函数](#)
- [7.5.4 QAction 类中的槽和信号](#)
- [7.5.5 QWidget 类中与 QAction 有关的函数](#)
- [7.5.6 QActionGroup 类动作组](#)
- [7.6 QMenu 类、菜单](#)
 - [7.6.1 菜单基本规则](#)
 - [7.6.2 QMenu 类中的属性](#)
 - [7.6.3 QMenu 类中的函数](#)
- [7.7 QMenuBar 类、菜单栏](#)
 - [7.7.1 菜单栏基本规则](#)
 - [7.7.2 QMenuBar 类中的属性](#)
 - [7.7.3 QMenuBar 类中的函数](#)
- [7.8 QToolBar 类、工具栏](#)
 - [7.8.1 工具栏基本规则](#)
 - [7.8.2 QToolBar 类中的属性](#)
 - [7.8.3 QToolBar 类中的函数](#)
 - [7.8.4 QToolBar 类中的信号](#)
- [7.9 QStatusBar 类、状态栏](#)
 - [7.9.1 状态栏基本规则](#)
 - [7.9.2 QStatusBar 类中的属性](#)
 - [7.9.3 QStatusBar 类中的函数](#)
- [7.10 QDockWidget 类、可停靠窗口、悬浮窗口](#)
 - [7.10.1 可停靠窗口基本规则](#)
 - [7.10.2 QDockWidget 类中的属性](#)
 - [7.10.3 QDockWidget 类中的函数](#)
 - [7.10.4 QDockWidget 类中的信号](#)
- [7.11 QMainWindow 类、主窗口](#)
 - [7.11.1 QMainWindow 类中的属性](#)
 - [7.11.2 QMainWindow 类中的函数](#)
 - [7.11.3 QMainWindow 类中的信号](#)

第 8 章 Qt 模型、视图框架目录

第 1 篇 自定义模型/视图框架

- [8.1 模型、视图原理](#)
- [8.2 模型：QAbstractItemModel 类](#)
- [8.3 视图：QAbstractItemView 类（视图基类）](#)
- [8.4 选择：QItemSelectionModel 类与 QItemSelection 类](#)
- [8.5 委托：QAbstractItemDelegate 与 QStyleOptionViewItem](#)
- [8.6 索引：QModelIndex 类](#)
- [8.7 自定义视图示例](#)

第 2 篇 Qt 实现的标准模型/视图框架相关类

- [8.8 标准模型：QStandardItemModel 类及 QStandardItem 类](#)
- [8.9 列表模型：QAbstractListModel 类、QAbstractTableModel 类、QStringListModel 类](#)
- [8.10 文件系统模型：QFileSystemModel 类](#)
- [8.11 表格视图：QTableView 类](#)
- [8.12 列表视图：QListView 类](#)
- [8.13 树视图：QTreeView 类](#)
- [8.14 标头视图：QHeaderView 类](#)
- [8.15 列视图：QColumnView 类](#)
- [8.16 项目委托：QStyleItemDelegate 类](#)

[第3篇 使用现成的模型/视图部件](#)

[8.17 表格部件: QTableWidget 类](#)

[8.18 列表部件: QListWidget 类](#)

[8.19 树部件: QTreeWidget 类](#)

第9章 Qt 拖放、剪贴板

[9.1 拖放原理](#)

[9.1.1 拖放基本原理](#)

[9.1.2 拖放动作或称为放置动作](#)

[9.1.3 使用拖放打开文件](#)

[9.2 与拖放事件有关的类及函数](#)

[9.2.1 QDropEvent 类](#)

[9.2.2 QDragMoveEvent 类](#)

[9.2.3 QDragEnterEvent 类和 QDragLeaveEvent 类](#)

[9.2.4 QWidget 类中与拖放有关的函数](#)

[9.3 QDrag 类](#)

[9.4 QMimeData 类与拖放自定义类型数据](#)

[9.4.1 基本规则](#)

[9.4.2 QMimeData 类中的函数](#)

[9.4.3 子类化 QMimeData](#)

[9.4.4 重新实现 QMimeData 类中的虚函数](#)

[9.5 QClipboard 类\(剪贴板\)](#)

第10章 Qt 滚动目录

[10.1 滚动条、滑块\(QAbstractSlider 类、QScrollBar 类、QSlider 类\)](#)

[10.1.1 基本原理](#)

[10.1.2 最大、最小值和步长](#)

[10.1.3 跟踪 Tracking 与当前值 Value、当前位置 Position](#)

[10.1.4 QAbstractSlider 类中的属性和函数](#)

[10.1.5 QAbstractSlider 类中的信号](#)

[10.1.6 QScrollBar 类](#)

[10.1.7 QSlider 类](#)

[10.1.8 QDial 类](#)

[10.2 QScrollArea 类、\(滚动区域\)](#)

[10.3 QAbstractScrollArea 类\(抽象滚动区域\)](#)

[10.3.1 QAbstractScrollArea 类中的属性](#)

[10.3.2 QAbstractScrollArea 类中的函数](#)

[10.3.3 自定义滚动区域](#)

第11章 Qt 文本系统目录

[11.1 重要基本概念及原理](#)

- [11.2 QPlainTextEdit 类](#)
- [11.3 QTextEdit 类](#)
- [11.4 表格: QTextTable 和 QTextTableFormat 类](#)
- [11.5 框架: QTextFrame 和 QTextFrameFormat 类](#)
- [11.6 文本块: QTextBlock、QTextBlockFormat 类](#)
- [11.7 列表: QTextList、QTextListFormat 类](#)
- [11.8 图像: QTextImageFormat 类和文本片段: QTextFragment 类](#)
- [11.9 插入自定义文档对象（文档元素）与总结](#)
- [11.10 QTextCharFormat 类及 QTextFormat 和 QTextObject 类简介](#)
- [11.11 QTextCursor 类](#)
- [11.12 QTextDocument 类](#)
- [11.13 其他类: QTextOption、QTextDocumentFragment 等](#)
- [11.14 语法高亮: QSyntaxHighlighter 类](#)

第 12 章 Qt 2d 绘图目录

- [12.1 二 D 绘图基础](#)
- [12.2 绘制直线与 QLineF 类](#)
- [12.3 绘制矩形与 QRectF 类](#)
- [12.4 绘制椭圆、弧、弦、扇形、圆角矩形](#)
- [12.5 绘制点、折线、多边形（QPolygonF 类）](#)
- [12.6 QPainterPath 类（路径）](#)
- [12.7 绘制文本](#)
- [12.8 QPen 类（画笔）](#)
- [12.9 QBrush 类（画刷）与渐变（QGradient 类及其子类）](#)
- [12.10 填充](#)
- [12.11 裁剪区域（QRegion 类）](#)
- [12.12 坐标变换（QTransform 类）](#)
- [12.13 绘制图像（QImage、QPixmap、QBitmap）](#)
- [12.14 抗锯齿和图像合成](#)

第 13 章 Qt 界面外观

- [13.1 简单的使用 QStyle 类](#)
 - [13.1.1 样式基础](#)
 - [13.1.2 QStyleFactory 类及其函数](#)
- [13.2 QPalette 类（调色版）](#)
- [13.3 自定义部件的外观](#)
- [13.4 子类化 QStyle](#)
 - [13.4.1 样式元素](#)
 - [13.4.2 样式绘制函数](#)
 - [13.4.3 子类化 QStyle 类的方法](#)
- [13.5 QStyle 类的其他枚举及成员函数](#)
 - [13.5.1 QStyle::PixMetric 枚举及相关成员函数](#)
 - [13.5.2 QStyle::StandardPixmap 枚举及相关成员函数](#)
 - [13.5.3 QStyle::StyleHint 枚举及相关成员函数](#)

- [13.5.4 其他枚举及相关成员函数](#)
- [13.5.5 QStyle 类中的其他成员函数](#)
- [13.6 QStyle 类中枚举的总结](#)
- [13.7 QStyleOption（样式选项）及其子类](#)
- [13.8 样式表](#)
 - [13.8.1 样式表基础](#)
 - [13.8.2 样式表语法基础](#)
 - [13.8.3 选择器](#)
 - [13.8.4 子控件](#)
 - [13.8.5 伪状态](#)
- [13.9 样式表的属性](#)
 - [13.9.1 背景色、前景色、所选文本的颜色](#)
 - [13.9.2 盒子模型及相关属性](#)
 - [13.9.3 与位置和大小有关的属性](#)
 - [13.9.4 字体、文本、图标、图像、不透明度属性](#)
 - [13.9.5 其他属性](#)
 - [13.9.6 属性类型](#)
- [13.10 设置各部件样式表的方法（综合示例）](#)
 - [13.10.1 基本规则](#)
 - [13.10.2 设置各部件样式表的方法](#)
- [13.11 样式表的其他规则](#)
 - [13.11.1 层叠和继承](#)
 - [13.11.2 名称空间及使用 QObject 属性](#)
 - [13.11.3 冲突解决](#)

第 14 章 Qt 输入/输出(暂定)

- 14.1 QDataStream 类(数据流)
- 14.2 QTextStream 类(文本流)
- 14.3 QFile 类
- 14.4 QDir 类

本文作者：黄邦勇帅(原名：黄勇)

本文要求读者已经非常熟悉 C++ 的语法，若读者不熟悉 C++ 语法，推荐参阅《C++ 语法详解》(作者：黄勇)一书，电子工业出版社出版。

本文是 QT 的入门文章，主要讲解了怎样使用 Qt Creator、编写 QT 程序的几种方法、QT 程序的生成过程、QT 程序产生的各种文件、QT 程序的发布、QT 项目文件(pro 文件)的语法、以及 QT 的框架结构。学完本文就能够熟练使用 Qt Creator，以及了解 QT 的整个组织结构，并能使用 QT 编写简单的 QT 程序。

本文内容由浅入深，内容较为全面，本文使用的 qt 版本为 qt5.10.1。

本文内容完全属于个人见解与参考文献的作者无关，限于水平有限，其中难免有误解之处，望指出更正。

声明：禁止抄袭、复印、转载本文，本文作者拥有完全版权。

主要参考文献：

- 1、Qt Creator 快速入门(第 3 版) 霍亚飞 编著 北京航空航天大学出版社 2017 年 1 月
- 2、C++ GUI Qt4 编程(第 2 版) [加拿大]Jasmin Blanchette [英] Mark Summerfield 著 闫锋欣 曾泉人 张志强 译 电子工业出版社 2008 年 8 月
- 3、Qt 5 开发及实例 陆文周 主编 电子工业出版社 2014 年 1 月

第 1 章 Qt 快速入门目录

[1.1 Qt 简介](#)

[1.2 Qt Creator 的使用](#)

[1.2.1 qt 的下载及安装注意事项](#)

[1.2.2 Qt Create 的界面介绍](#)

[1.2.3 手动添加套件](#)

[1.3 使用 Qt Creator 编写一个 C++程序](#)

[1.3.1 创建一个空项目](#)

[1.3.2 添加 C++代码](#)

[1.3.3 运行程序](#)

[1.3.4 使用 QtCreator 编辑器的一些技巧](#)

[1.4 使用 Qt Creator 编写 Qt 程序](#)

[1.4.1 方法 1：使用代码编写 Qt 程序](#)

[1.4.2 方法 2：使用界面编辑器（即设计模式）编写 Qt 程序](#)

[1.4.3 方法 3：使用 Qt 设计师界面类编写 Qt 程序](#)

[1.4.4 方法 4：使用“Qt Widgets Application”编写 Qt 程序](#)

[1.4.5 方法 5：使用记事本及 Qt 命令程序编写 Qt 程序](#)

[1.5 发布程序](#)

[1.5.1 编译后生成的各种文件简介](#)

[1.5.2 手动发布程序](#)

[1.5.3 使用 windeployqt 工具部署文件（仅限 windows）](#)

[1.5.4 静态编译和动态编译](#)

[1.6 Qt 的重要文件简介](#)

[1.6.1 项目文件（pro 文件）及其语法](#)

[1.6.2 moc 简介](#)

[1.6.3 pro、pri、prf 文件简介](#)

[1.7 Qt 框架结构简介](#)

[1.7.1 Qt 基本框架](#)

[1.7.2 Qt5 模块架构](#)

[1.8 帮助文档的使用](#)

第 1 部分 Qt 快速入门

- 1、本书使用的是 windows 10 操作系统，主要讲解 windows 系统下的 Qt 程序
- 2、本书使用的 qt 版本为 qt5.10.1，Qt Creator 的版本号为 Qt Creator 4.5.1
- 3、QT5 的源代码文件必须使用 UTF-8 编码。

1.1 Qt 简介

- 1、Qt 是一个跨平台的 C++图形用户界面库，说简单点，Qt 的本质就是一个 C++类库，使用 Qt 就是怎样使用 Qt 类库中的类及其类中的成员函数的问题。在 QT5 中 QML(这是一种声明性语言)和 Qt Quick 成为 Qt 的核心之一，但 C++仍是 QT 的核心。
- 2、Qt 是跨平台的，也就是说，使用一个 Qt 开发框架就可以开发出能在桌面、嵌入式、移动等多个平台运行的应用程序，因此一套代码可以运行在各个不同的平台上。从 QT5.6 开始实现了对 Andriod、iOS、winRt 等移动平台的完整支持，目前 Qt 支持 Windows、Linux、macOS、Android、iOS、WinRT 等平台，将现有的 QT 程序移植到这些平台，只需重新编译一次源代码即可。
- 3、Qt 虽然是使用的 C++语言，但不是使用的标准 C++，Qt 进行了一定程度的“扩展”。虽然如此，但 C++仍然是基础。

1.2 Qt Creator 的使用

一、qt 的下载及安装注意事项

- 1、Qt Creator 是 Qt 的集成开发环境(IDE)，Qt 5.10.1 是 Qt 类库的版本，Qt Creator 4.5.1 是 Qt Creator 的版本，类似于 C++11 是 C++语言的版本，Visual C++2015 是 IDE 的版本。
- 2、Qt 的下载地址为：http://download.qt.io/official_releases/qt
- 3、进入下载页面后选择需要下载的 Qt 版本，本书使用的是 Qt5.10.1，然后会发现 Qt5.10.1 版本还有多个类型的 Qt 安装程序下载，他们的名称具有如下类型的形式
`qt-opensource-windows-x86-5.10.1.exe`，//本书使用的版本
`qt-opensource-mac-x64-5.10.1.dmg`，
其中 `opensource` 表示此安装程序是开源版本的，除开源版本外，Qt 还有商业版本的(商业版本需要购买)，第一安装程序是 Windows 版本的 Qt。
- 4、安装注意事项：遇到登录或注册 Qt 账号的窗口时，选择“skip”按钮跳过即可，这一步并不影响 Qt 的安装和使用，其余步骤保持默认，直接点击“next”即可。遇到选择组件的步骤时建议全选

二、Qt Create 的界面介绍

1、欢迎模式的窗口界面(首先打开 Qt Creator 时的界面)如下图所示



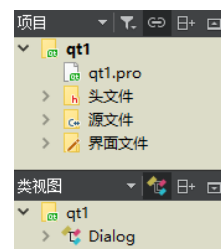
2、编辑模式的窗口界面如下图所示



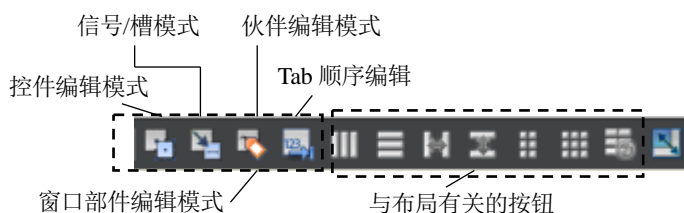


3、工具栏各按钮作用如下

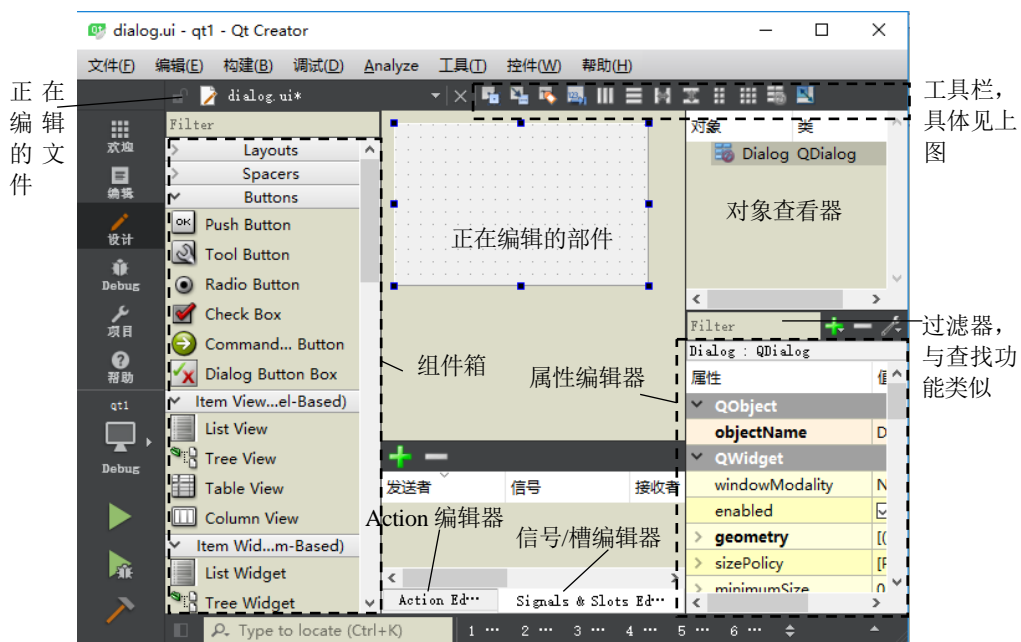
- 下拉列表①：选择在视图区显示的内容
- 按钮②：可控制视图区的显示形式
- 与编辑器同步按钮③：通常选中此按钮，否则在不同文件间切换时，视图区中的文件选择加亮条不会跟随移动
- 按钮 4：用于在视图区分栏显示不同的内容，在展开的菜单中选择“类视图”后的效果如右图
- 按钮⑤：可关闭当前显示的内容，但不能重新打开，若整个视图区都被关闭了，则要重新打开视图区，需点击编辑器左下角的类似按钮(即编辑模式中的按钮 6)
- 下拉列表⑦：显示当前正在编辑的文件，可使用 **Ctrl+Tab** 在打开的文件间进行切换，注意，下拉列表中的文件不一定会出现在左侧的“视图区”内。在此下拉列表上右击或单击左侧的按钮⑥，会出现一个对该下拉列表中的文件进行操作的弹出菜单(比如删除当前文件等)
- 按钮⑧：可关闭左侧下拉列表⑦当前正在编辑的文件
- 下拉列表⑨：显示当前正在编辑代码的位置位于哪个类及成员函数中



4、设计模式的窗口界面如下两图所示(需要添加相应的 ui 文件才可显示)

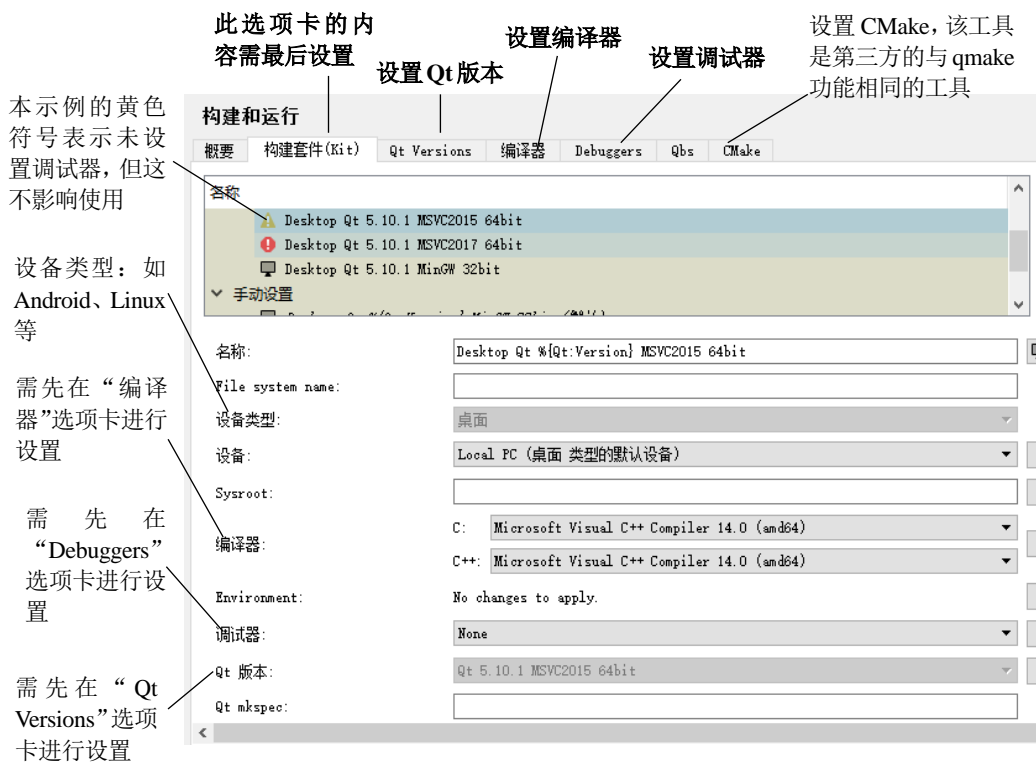


设计模式工具栏的各按钮

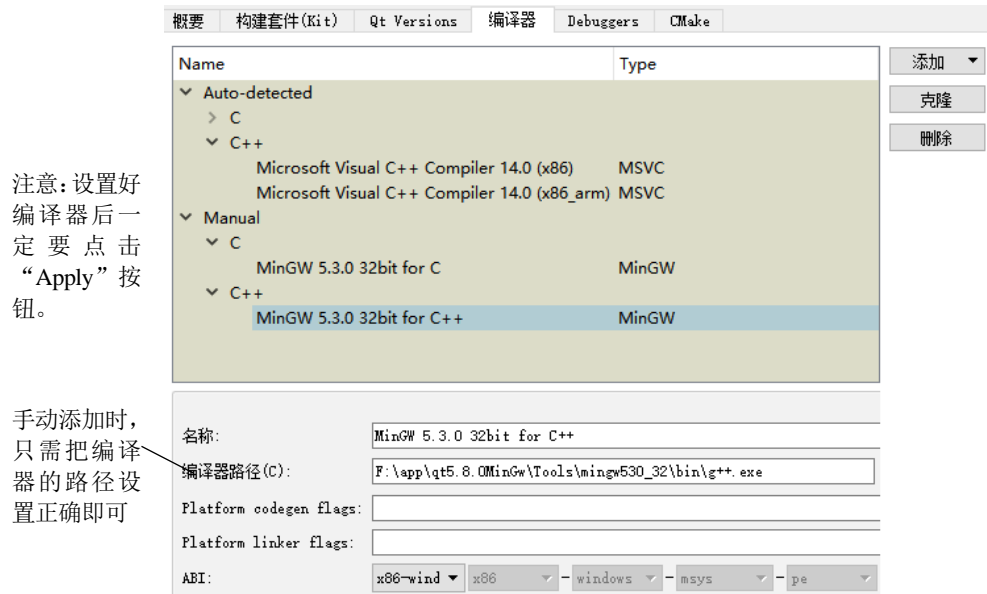


三、手动添加套件

- 1、套件(kit): 是指成套的元件，因此套件是多个元件的集合。在 Qt 中，qmake 工具、Qt 类库、编译器、调试器都是套件中的元件。
- 2、为什么需要手动添加套件：
 - 通常 Qt 安装程序会自动检测已安装的编译器和 Qt 版本，如果成功，相关的套件(kit)将会自动添加到 Qt Creator 中，否则就必须自己添加套件。
 - 如果安装了多个 Qt 版本，或者使用不同编译器、调试器，则需要自己添加套件来告诉 Qt Creator 在哪里查找 Qt 版本、编译器、调试器。
- 3、Qt Creator 的主要工具是 qmake 和 Qt 界面设计器，这两个工具都已经集成在 Qt Creator 之中。
 - qmake 工具可以根据工程文件(.pro)产生出不同平台下的 makefile 文件。
 - 前文的“设计模式”就是使用的 Qt 的界面设计器，在 Qt 界面设计器中可以使用拖放快速设计用户界面，而不需要编写代码。
- 4、Qt Creator 是一个集成开发环境 (IDE)，但独立的 Qt Creator 安装程序并未安装 Qt (这里指 Qt 类库)或任何 Qt 工具，比如 qmake 工具。注意：在安装 Qt Creator 时，可以不安装任何的 Qt 类库。因此要使用 Qt Creator 进行 Qt 开发，还需要向 Qt Creator 中添加 Qt 版本和编译器。
- 5、手动添加套件的方法
 - ①、点击“工具”>“选项”在弹出的对话框中选择“构建和运行”，界面如下图所示。



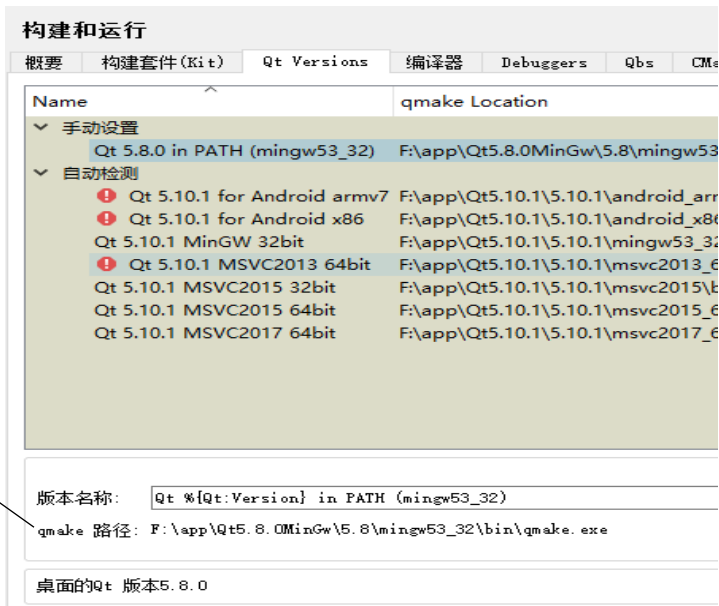
②、添加编译器，其选项卡的界面如下图



③、设置 Qt 版本，其选项卡的界面如下图

注意：设置好后一定要点击“Apply”按钮。

手动添加时，只需把 qmake 的路径设置正确即可



④、Debuggers 和 CMake 暂时不需要设置，因为它们不影响使用。

1.3 使用 Qt Creator 编写一个 C++ 程序

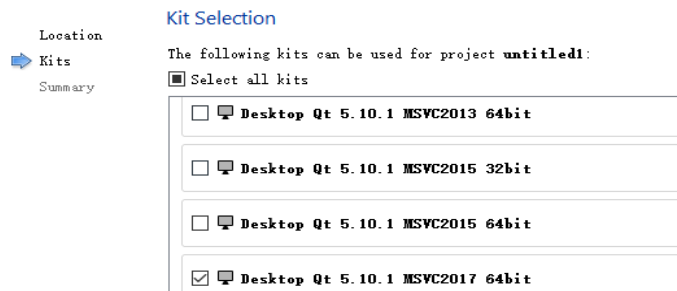
一、创建一个空项目

- 1、选择“文件”>“新建文件或项目...”，然后在对话框左侧选择“其他项目”，在右侧选择“Empty qmake Project” (即空项目)，然后单击“Choose”按钮，进入如下图所示内容的对话框



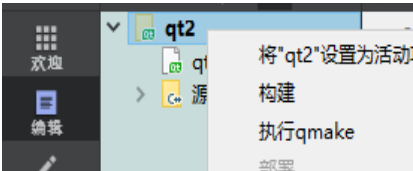
- 2、点击下一步，出现如下图所示对话框，以选择使用哪个“套件”运行创建的项目。注：若添加了多个套件，则会出现下图所示的对套件的多个选择，此处选择如图所示套件，点击

下一步，然后再出现一个对话框，对于该步骤可不用设置，直接单击“完成”按钮即可，至此项目 qt2 创建完成。



二、添加 C++ 代码

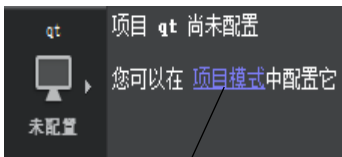
- 1、点击左侧的“编辑”(或按下 Ctrl+2)进入编辑模式，在“视图区”中的 qt2 文件夹上右击(见右图)，弹出菜单，然后选择“添加新文件”，弹出一个对话框，在对话框左侧选择“C++”，右侧选择“C++ Source File”(即 C++源文件)，然后点击“Choose”按钮，再次弹出对话框，在对话框的名称处输入新建的 C++源文件的名称，在路径处选择新建的 C++源文件的保存路径，建议使用默认路径，以使源文件保存在当前的项目文件夹 q2 中，然后点击下一步，此时保持默认值，直接点击完成即可，至此 C++源文件创建完成。
- 2、在“视图区”展开“qt2”文件夹，再展开“源文件”文件夹，双击新创建的 C++源文件，然后在编辑区中输入如下 C++代码。



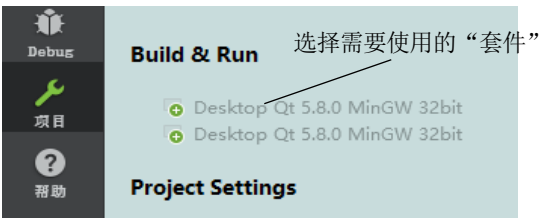
```
#include<iostream>
using namespace std;
int main() { cout<<"Qt"<<endl; return 0;}
```

三、运行程序

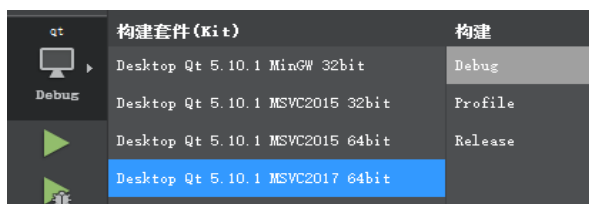
- 1、选择套件：点击左侧的“构造/套件选择器”，此时会根据项目的不同情况，出现如下图所示的几种情形



点击此处会自动跳转到右图的项目模式，以选择需要使用哪个套件来运行该项目。



情形 1



情形 2



情形 3

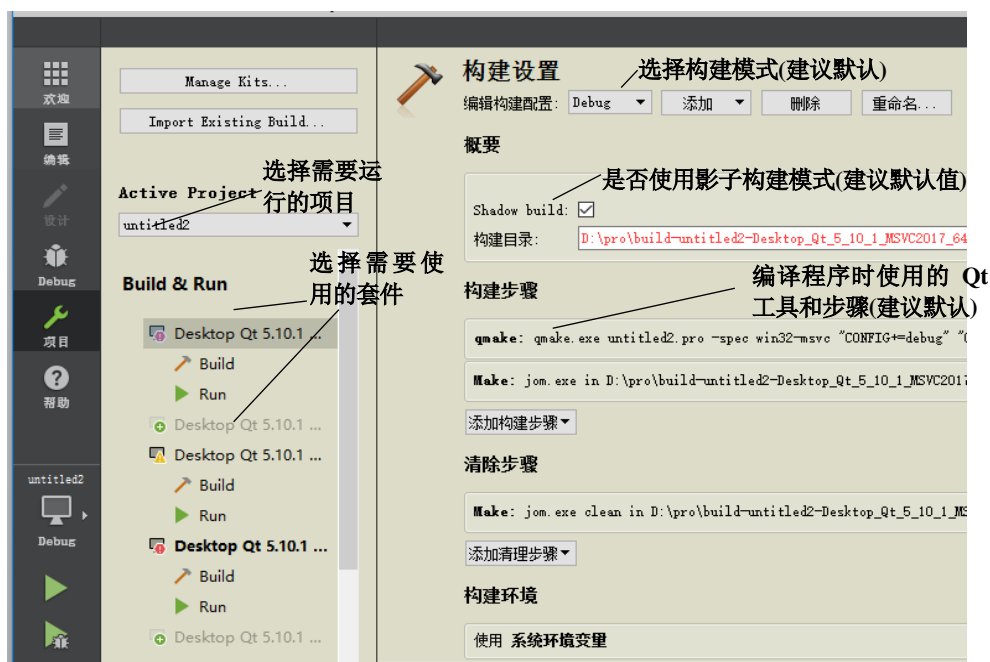
选择需要运行的项目

选择需要运行该程序的套件

选择构建模式

以上两种情形，也可在“项目模式”进行设置，项目模式对话框见下图

2、项目模式对话框(此后若不特别说明，文中均使用如下图所示的设置)



3、点击左侧的“运行”按钮(或使用 Ctrl+R)运行程序，点击 Qt Creator 下方的“应用程序输出”窗口(或使用 Alt+3)可看到程序输出的内容，在“编译输出”(Ctrl+4)窗口可看到程序的编译执行过程。

4、影子构建：使用该模式会使编译生成的文件和源代码文件分别存放，通常使用默认值。

三、使用 Qt Creator 编辑器的一些技巧

- 1、字体大小的缩放：使用 Ctrl+"+"放大字体，Ctrl+"-"缩小字体，或使用 Ctrl+“鼠标滚轮”，使用 Ctrl+0 还原为默认大小。
- 2、当输入一个字符时，按下 Ctrl+“空格”，会出现与该字符有关的名称选项。
- 3、若调用一个函数，当输入函数名时，会出现一个函数参数的小提示框，可使用上下方向键查看该函数重载版本的原型。

1.4 使用 Qt Creator 编写 Qt 程序

若不特别说明，以后的程序都使用“MinGw”版本的套件，构建模式使用“Debug”版本。

一、方法 1：使用代码编写 Qt 程序

- 1、新建空项目，双击项目文件(.pro 文件)，然后在项目文件中添加如下代码，

```
greaterThan(QT_MAJOR_VERSION,4):QT += widgets
```

上述代码表示，若 Qt 版本大于 4，则添加 widgets 模块。因后面的程序需要使用 widgets 模块中的类，所以需要在 pro 文件中添加该模块。使用上述代码主要是为了与 Qt4 兼容，因此上述代码也可简写为：QT += widgets。

- 2、注意：若修改了 pro 文件，建议重新构建该项目文件，否则对项目文件的更改不会反应到程序中，方法为选择“构建”>“重新构建项目 xxx”。
- 3、添加 C++源文件，然后在源文件中添加如下代码：

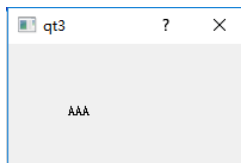
```
#include<QApplication> /*Qt 中的每一个类都有一个与其同名的头文件，因此要使用哪个类，就需要包含相应的头文件*/

#include<QDialog>
#include<QLabel>
int main(int argc, char *argv[]) {
    QApplication a(argc, argv); /*任何一个 widgets 程序都需要包含一个 QApplication 对象，该对象用于管理程序的资源、事件等*/

    QDialog d; //创建一个对话框，Qt 使用 QDialog 类对象创建对话框。
    d.resize(200, 100); //设置对话框的大小。
    QLabel s(&d); /*创建一个标签，并把该标签放在对话框 d 之中。同理，Qt 使用 QLabel 类对象创建标签*/
    s.setText("AAA"); //设置标签显示的文本。
    s.move(50, 50); //设置标签位于对话框中的位置。
    d.show(); //显示对话框，默认情况下部件是不可见的，因此需显示。
    a.exec(); /*程序进入消息(或事件)循环，等待用户的输入并进行响应，这时程序将控制权交给 Qt 用于完成事件处理。*/

    return 0; }
```

运行结果如下图所示

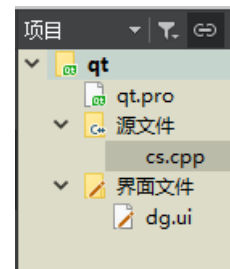


- 4、验证项目文件的更改：把上例在项目文件中添加的 `QT += widgets` 代码删除掉，然后保存该项目文件(可使用 `Ctrl+S`)，然后运行程序(`Ctrl+R`)，发现程序仍能正常执行，这说明对项目文件的更改未起作用。此时，选择“构建”>“重新构建项目 xxx”，此时在“问题窗口”(`Alt+1`)，便显示出了产生的错误信息，这说明此时对项目文件的更改才产生作用。

二、方法 2：使用界面编辑器(即设计模式)编写 Qt 程序

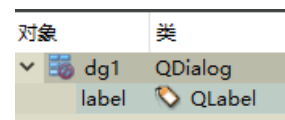
1、新建各种文件：

新建一个名称为 qt 的空项目，然后添加一个名为 cs 的 C++源文件，再添加一个 ui 文件，其步骤为：右击项目文件夹 qt，选择“添加新文件”，在对话框的左侧选择“Qt”，右侧选择“Qt Designer Form”，在随后弹出的对话框中选择“Dialog without Buttons” (即不含按钮的对话框)，在下一步的名称中输入“dg.ui” (名称可任意选择)，路径保持默认。最终创建的文件如右图所示



2、编辑 ui 文件：双击 ui 文件，进入设计模式界面，

- 添加标签：在设计模式界面的左侧找到名称为“Label”的部件，然后按住左键不放，将其拖放到窗口的合适位置。
- 修改属性：对话框和标签的大小可直接拖放进行修改，也可展开右下侧的 geometry 属性修改。标签名称的修改可以双击该标签然后直接输入名称，也可在标签的 text 属性处修改。
- 修改对话框对象的名称：为避免使用相同名称产生的混乱，对此将对话框对象的名称修改为 dg1，其方法为，在右上侧双击对象下的名称(如右图)，然后直接输入名称即可，也可在此名称上右击，然后选择“改变对象名称”，也可在 `objectName` 属性处进行修改。



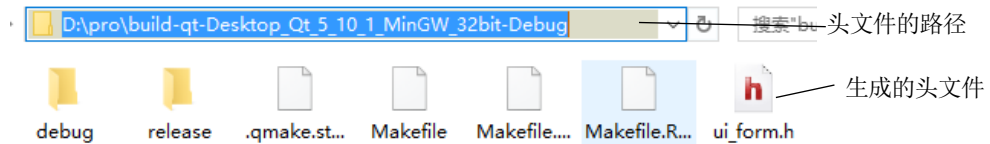
3、查看 ui 文件：在编辑模式(`Ctrl+2`)下，右击 ui 文件，选择“用...打开”，此时可看到有如下选项：

- 界面编辑器和 System Editor：就是使用前面的方法编辑 ui 文件。
- 普通文本编辑器：此时可看到 ui 文件的内容，该文件是一个 XML 文件，里面是一些关于界面部件的信息。
- 二进制编辑器：此时可使用二进制的形式查看 ui 文件的内容。
- Qt Designer：即 Qt 设计师，这是一个用于设计界面的软件，此时会打开一个“Qt 设计师”软件来编辑 ui 文件。

3、根据 ui 文件生成相应的头文件

- 此时 C++源文件的内容可以为空。

- 选择“构建”>“构建所有项目”(或 Ctrl+Shift+B), 因为没有 main 函数(源文件为空), 因此会产生错误, 但这并不会阻止使用 ui 文件产生出相应的头文件。
- 头文件的位置: 如下图所示



若“项目模式”(Ctrl+5)中的选项都是默认选项, 则头文件的位置与“项目模式”中“构建设置”对话框中的“构建目录”和“运行设置”对话框中的“Working directory”(工作目录)相同。

4、由 ui 产生的头文件 ui_dg.h 的内容及意义如下所示

```
#ifndef UI_DG_H //ifndef、#define、#endif 预处理器用于防止头文件被多次包含
#define UI_DG_H
#include <QtCore/QVariant> //包含了一系列的头文件
#include <QtWidgets/QAction>
#include <QtWidgets/QApplication>
#include <QtWidgets/QButtonGroup>
#include <QtWidgets/QDialog>
#include <QtWidgets/QHeaderView>
#include <QtWidgets/QLabel>
QT_BEGIN_NAMESPACE //Qt 的命名空间开始宏

class Ui_dgl { public: //定义一个类
    QLabel *label; //这就是添加到对话框中的标签
    void setupUi(QDialog *dgl) { /*此函数用于生成界面, 程序必须调用此函数, 才能产生界面, 也就是说, 虽然设计模式能拖放出界面的外观, 但还是要编写代码来调用此函数以生成界面*/.

        if (dgl->objectName().isEmpty())
            dgl->setObjectName(QStringLiteral("dgl")); //设置对话框的对象名称
        dgl->resize(200, 100); //设置对话框的大小
        label = new QLabel(dgl); //将标签添加到对话框之上。
        label->setObjectName(QStringLiteral("label")); //设置标签的对象名称
        label->setGeometry(QRect(60, 50, 54, 12)); //设置标签的位置和大小。
        retranslateUi(dgl); //该函数在下面进行了定义。
        QMetaObject::connectSlotsByName(dgl); } //主要用于实现信号和槽相关联。
    void retranslateUi(QDialog *dgl) { //主要用于对窗口中的字符串进行编码转换。
        dgl->setWindowTitle(QApplication::translate("dgl", "Dialog", Q_NULLPTR));
        label->setText(QApplication::translate("dgl", "AAA", Q_NULLPTR));
    };
};

namespace Ui { class dgl; public Ui_dgl {}; } //定义一个名称空间。
QT_END_NAMESPACE //Qt 的命名空间结束宏
#endif
```

- 5、在 C++源文件中输入如下内容, 然后程序即可正确运行, 也就是说使用此方法, 只需在界面设计器之中设计界面, 然后在 C++源文件中编写如下代码就可以了。

```
#include "ui_dg.h" /*包含由 ui 文件生成的头文件, 此处应使用双引号, 否则不会在当前目录搜索头文件。*/

int main(int argc, char *argv[]) {
```

```

    QApplication a(argc, argv);
    QDialog d;
    Ui::dg1 ui;    //使用名称空间 Ui 中的类 dg1 创建一个对象(Ui 名称空间定义于 ui_dg.h 中)。
    ui.setupUi(&d);    //调用 dg1 的成员函数 setupUi 生成界面(即 ui_dg.h 文件中的函数)。
    d.show();        a.exec();
    return 0;        }

```

6、以上程序可简化为如下形式的代码

```

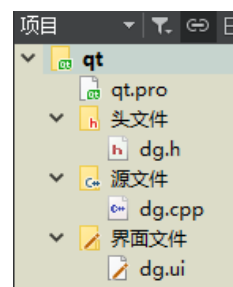
#include <QtWidgets/QApplication>
#include <QtWidgets/QDialog>
#include <QtWidgets/QLabel>
class Ui_dg1{ public:
    QLabel *label;
    void setupUi(QDialog *dlg){
        dlg->resize(200, 100);
        label->setGeometry(QRect(60, 50, 54, 12));
        label = new QLabel(dlg);
        label->setText("AAA");
    };
int main(int argc, char *argv[]){    QApplication a(argc, argv);
    QDialog d;        Ui_dg1 ui;        ui.setupUi(&d);
    d.show();        a.exec();        return 0;    }

```

三、方法 3：使用 Qt 设计师界面类编写 Qt 程序

1、新建各种文件：

新建一个名称为 qt 的空项目，然后再添加一个 Qt 设计师界面类，步骤为：右击项目文件夹 qt，选择“添加新文件”，在对话框的左侧选择“Qt”，右侧选择“Qt 设计师界面类”，在随后弹出的对话框中选择“Dialog without Buttons”（即不含按钮的对话框），在下一步的“类名”中输入“dg”（名称可任意选择），其余的头文件、源文件、ui 文件的名称会自动生存，保持默认即可。确定后系统会自动生成头文件、项目文件、源文件及 ui 文件，最终的文件如右图所示



2、在设计模式下编辑 ui 文件，只需在对话框中添加一个标签(Label)即可，注意 pro 文件中仍需添加 QT+=widgets 的代码。

3、在编辑模式下添加一个 C++ 源文件，该源文件主要用于编写 main 函数，其代码如下

```

#include<QApplication>
#include"dg.h"    //该头文件为创建“Qt 设计师界面类”时创建的。
int main(int argc, char *argv[]){
    QApplication a(argc, argv);    dg w;        w.show();    a.exec();    return 0; }

```

然后程序即可运行。可见，使用 Qt 设计师界面类编写 Qt 程序更简洁。

4、Qt 设计师界面类下的源码追踪，打开由系统生成的文件，其源码分别如下

//dg.h 文件的内容如下

```

#ifndef DG_H    //防止头文件被包含多次
#define DG_H
#include <QDialog>
namespace Ui {    class dg; }    /*名称空间 Ui 中的 dg 与下面定义的 dg 是两个不同的类，Ui::dg
                                定义于头文件 ui_dg.h 内，这里仅是对 Ui::dg 的一个前置声明。*/

```

```
class dg : public QDialog    { //该类仅含一个构造函数、析构函数和 ui 指针成员
    Q_OBJECT    //此宏必须在类的开始定义，主要作用是扩展传统 C++ 类的功能，如信号、槽等
public:    explicit dg(QWidget *parent = 0); //explicit 是 C++ 关键字
    ~dg();
private:    Ui::dg *ui; };
#endif // DG_H
```

//dg.cpp 文件的内容如下

```
#include "dg.h"
#include "ui_dg.h"    /*该头文件在构建(或运行)时会由系统自动生存，其内容与方法 2 生成的类
                        似头文件相同，位于目录
                        D:\pro\build-qt-Desktop_Qt_5_8_0_MinGW_32bit-Debug*/

//对构造函数的定义
dg::dg(QWidget *parent): QDialog(parent), ui(new Ui::dg) { //C++ 语法，成员初始化表的使用
    ui->setupUi(this); }    /*使用 Ui::dg::setupUi 生成界面，可见，方法 3，只是把方法 2 在 main
                            函数中对 Ui::dg::setupUi 函数的调用封装在了另一个类之中，以此
                            简化了 main 函数中的代码。*/

dg::~dg() {    delete ui; }    //对析构函数的定义
```

//ui_dg.h 文件中的关键内容如下(详细内容见方法 2)

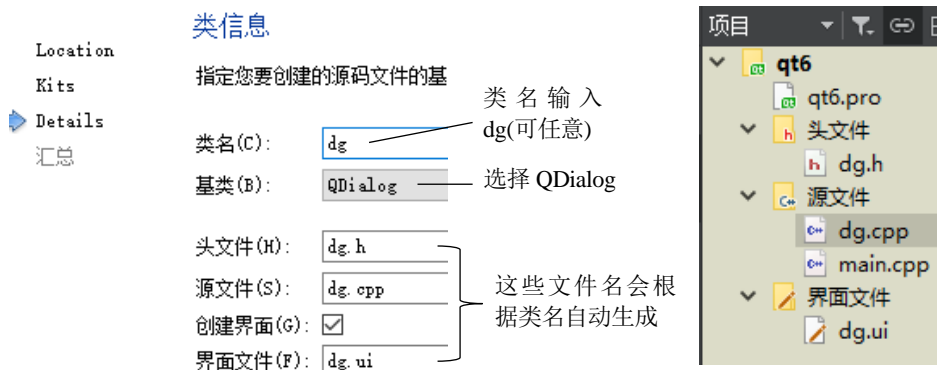
```
#ifndef UI_DG_H
#define UI_DG_H
#include <QtCore/QVariant>
.....
namespace Ui {    class dg: public Ui_dg {};} //重点：对 Ui::dg 的定义
```

四、方法 4：使用“Qt Widgets Application”编写 Qt 程序

1、使用此方法可以不需编写代码，直接在设计模式拖放出窗口之后运行即可。

2、新建各种文件：

新建项目时，在左侧选择“Application”，右侧选择“Qt Widgets Application”，中间步骤同以上方法，然后再在下图所示窗口中进行如下设置，最终的文件如下图右侧所示

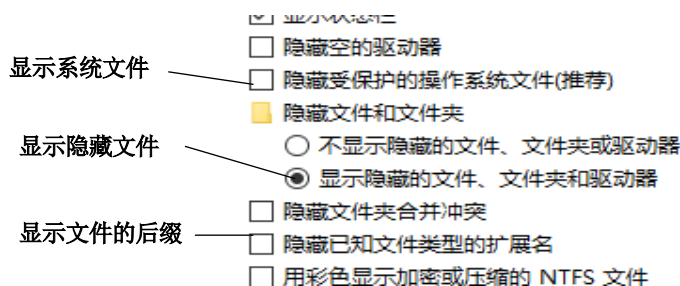


3、在设计模式下编辑好 ui 文件之后，直接运行即可，此方法不需要修改 pro 文件，系统已经为我们自动生成了相应的代码。

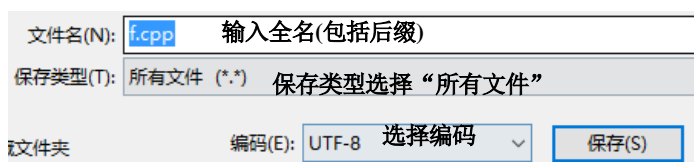
4、使用此方法生成的各文件的代码，与方法 3 是类似的。

五、方法 5：使用记事本及 Qt 命令程序编写 Qt 程序(以 minGW 编译器为例)

- 1、本文使用命令程序主要是为了让读者了解 Qt 的执行步骤，因此不会重点讲解。
- 2、为避免不必要的麻烦，请对操作系统的“文件夹选项”作如下设置

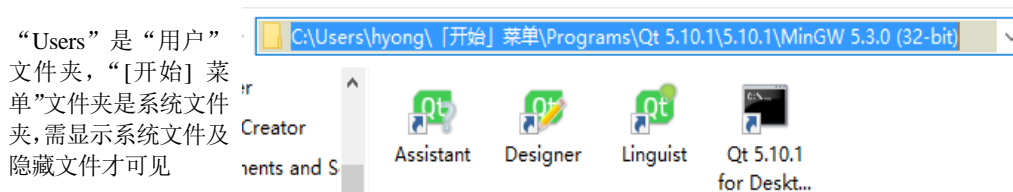


- 3、创建一个记事本，并把方法 1 中的程序代码复制到记事本之中，然后把记事本文件另存为.cpp 文件(假设名称为 f.cpp)，存储时编码选择 UTF-8，以避免产生乱码，如下图

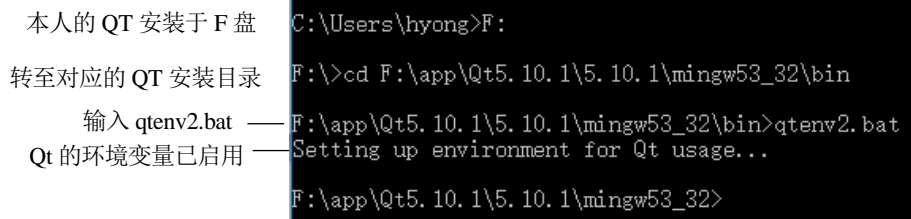


- 4、因为要使用 Qt 内置的命令工具，因此需要启用 Qt 环境变量，方法如下

- 方法 1: 在开始菜单的 Qt 安装目录中找到 Qt 5.10.1 for Desktop (MinGW 5.3.0 32 bit)。
注：win10 操作系统可能在开始菜单中找不到上述程序，需在下图所示路径查找



- 方法 2: 使用 windows 的 cmd 命令启用 qt 的环境变量，方法为：按下 win+R，输入 cmd，然后在命令行转至如下图所示路径(也可配置环境变量一劳永逸解决此问题)



- 5、编译并运行程序，步骤如下

- ①、在命令行下，转至 f.cpp 文件所在目录，并输入 qmake -project 命令以生成 pro 文件
qmake 工具简介：qmake 可由与平台无关的 pro 文件产生出与平台相关的 makefile 文

件,也可使用-project 参数在当前目录下生成 pro 文件,此时 qmake 会搜索当前目录下扩展名为.h、.cpp、.ui 等的文件,生成一个列举这些文件的 pro 文件。

本人的 f.cpp 文
件所在路径

输入的命令

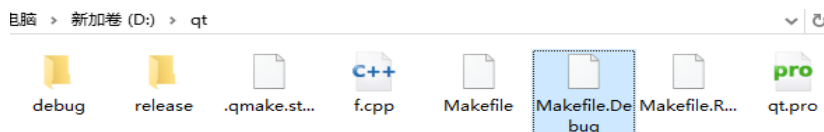
```
D:\>cd qt
D:\qt>qmake -project
D:\qt>_
```

- ②、此时在 D:\qt 文件夹下会生成一个 pro 文件,使用记事本打开该文件,并在其中添加语句“QT+=widgets”,然后输入 qmake 命令,以生成编译时使用的 Makefile 文件。

也可输入
qmake qt.pro

```
D:\qt>qmake
Info: creating stash file D:\qt\.qmake.stash
```

- ③、进入 D:\qt 文件夹,此时该文件夹中生成的文件如下图所示



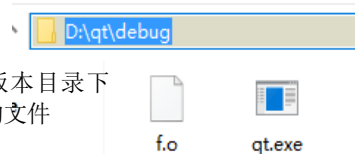
- ④、编译程序:

- ◆ 输入 mingw32-make(注意,之间没有空格)命令,可以产生 Qt 程序的 release 版本。
- ◆ 输入 mingw32-make -f Makefile.Debug,可产生 Qt 程序的 debug 版本,注意 mingw32-make 之间没有空格,-f 的前后都有一空格。
- ◆ 此时会在上图所示的release 目录或debug 目录下产生f.o (这是中间文件)和qt.exe 两个文件(如下图所示)

```
D:\qt>mingw32-make -f Makefile.Debug
```

输入的命令

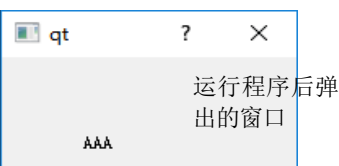
对应版本目录下
生成的文件



- ⑤、运行程序: 程序必须在命令行下运行,直接在文件夹下双击 qt.exe 运行程序会出错(因为程序还未发布),若生成的程序是 debug 版本的,则在命令行下进入该文件夹,然后输入 qt.exe 以运行程序,如下图

输入的命令

```
D:\qt>cd debug
D:\qt\debug>qt.exe
D:\qt\debug>
```



6、以上使用的 Qt 程序分别位于以下目录,注意 ui 文件需要使用 qt 的 uic 程序编译。

F:\app\Qt5.10.1\5.10.1\mingw53_32\bin

//qmake.exe 所在目录

F:\app\Qt5.10.1\Tools\mingw530_32\bin

//mingw32-make.exe 所在目录

7、从以上步骤可以看出，Qt 程序的执行步骤如下

- 生成与平台无关的 pro 文件
- 然后根据 pro 文件产生出与平台有关的 makefile 文件，注意：makefile 是与平台有关的，也就是说，不同的平台下产生出的 makefile 文件是不同的，比如 linux 下的 makefile 文件就与 windows 下的 makefile 文件不同。
- 使用编译工具编译项目，最后运行程序，注意：编译项目还可使用 make 或 nmake 工具。

8、构建、重新构建、运行的区别

- ①、qmake 用于生成项目文件(pro 文件)
- ②、构建：就是编译，但是只编译有变化的部分
- ③、重新构建：是把所有部分都重新编译。当 pro 文件有改动时，建议对其项目进行重新构建。
- ④、运行：就是调用构建之后生成的可执行文件(比如 exe 文件)
- ⑤、所以 Qt 程序运行的顺序是：qmake – 构建 – 运行

9、若使用 Qt Creator 运行以上程序，可在 Qt Creator 的“编译窗口”(Alt+4)看到如下的与以上步骤相对应的命令，如下图所示

```
22:15:05: 为项目qt执行步骤 ...
22:15:05: 配置没有改变, 跳过 qmake 步骤。
22:15:05: 正在启动 "F:\app\Qt5.10.1\Tools\mingw530_32\bin\mingw32-make.exe"

F:/app/Qt5.10.1/Tools/mingw530_32/bin/mingw32-make -f Makefile.Debug
mingw32-make[1]: Entering directory 'D:/pro/build-qt-Desktop_Qt_5_10_1_MinGW_32bit-Debug'
mingw32-make[1]: Nothing to be done for 'first'.
mingw32-make[1]: Leaving directory 'D:/pro/build-qt-Desktop_Qt_5_10_1_MinGW_32bit-Debug'
22:15:06: 进程"F:\app\Qt5.10.1\Tools\mingw530_32\bin\mingw32-make.exe"正常退出。
22:15:06: Elapsed time: 00:01.
```

六、总结

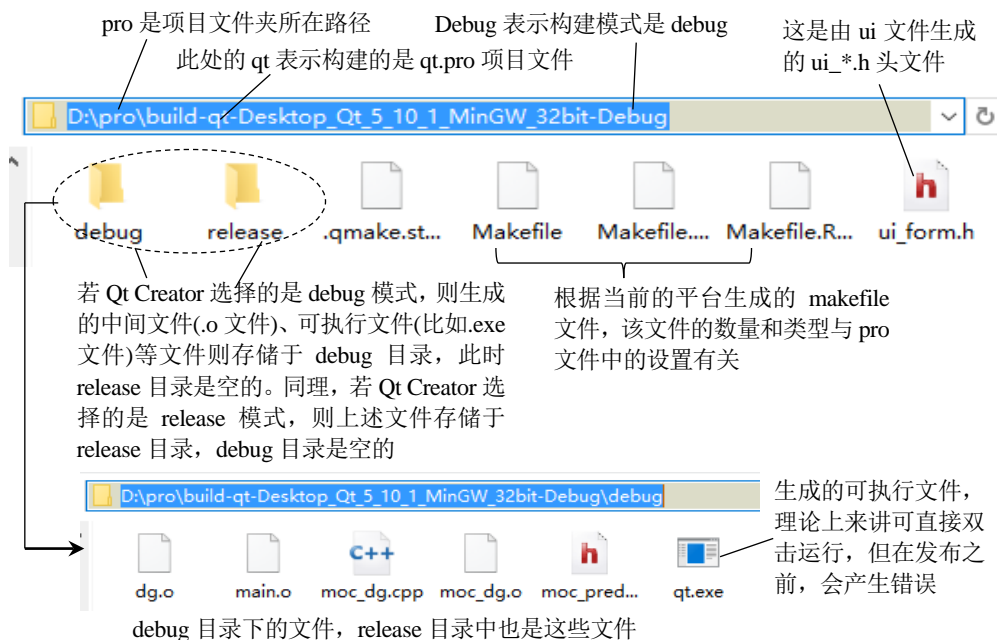
由以上 5 种编写 Qt 程序的方法可以看到，由界面设计器生成的 ui 文件，其代码是使用的子类化的方式生成的(即继承 Qt 类库的源码)，因此编写 Qt 程序最好使用子类化的方式。

1.5 发布程序(以 minGW 编译器为例)

- 1、注意：修改 pro 文件后，最好执行“构建”>“重新构建项目”，否则 pro 文件的更改将不会反应到程序上。
- 2、发布程序的目的：就是让编译后生成的可执行文件(如 exe 文件)，能在其他计算机上运行。

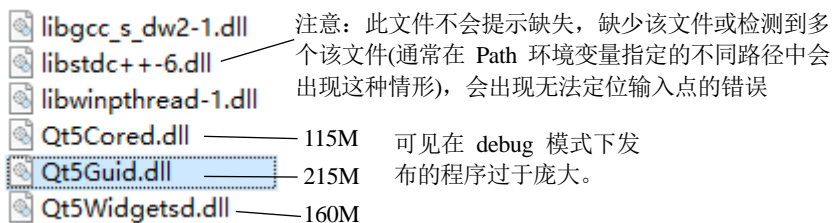
一、编译后生成的各种文件简介

Qt Creator 构建项目后产生的文件及目录见下图



二、手动发布程序

- 1、直接双击以运行可执行文件(以 exe 文件为例), 会弹出提示缺少 Qt5Core.dll 等一系列文件的错误, 这说明要运行该 exe 文件, 需要有提示的文件的支持, 所以应把这些缺少的文件复制到 exe 文件所在的目录下。
- 2、注意: 发布程序所需的文件会依使用的 Qt Creator 版本、构建时的模式、编译器、编写的程序的复杂度等的不同而有所不同。本例使用如下版本: Qt Creator 4.5.1 (基于 Qt 5.10.1 版本), 编译器为 MinGw5.3.0 32bit, exe 文件使用 debug 模式构建, 且 exe 文件仅仅是显示一个对话框(程序很简单), 所以需要的文件很少。
- 3、运行 exe 文件所必需的文件及目录如下(debug 模式),



目录为: F:\app\Qt5.10.1\5.10.1\mingw53_32\bin //依 qt 安装路径而有所不同

- 4、发布程序: 把以上文件和 “F:\app\Qt5.10.1\5.10.1\mingw53_32\plugins” 目录中的 platforms 文件夹(其中只需保留 qwindowsd.dll 文件)、连同 exe 文件一起打包复制到别的计算机上, 双击其中的 exe 主程序, 就可以运行了。当然, 本例发布的是 windows 程序, 因此要求别

的计算机也运行 windows 系统。其中的 qwindowsd.dll 是发布 windows 平台程序必须的文件，对于发布 release 版本的程序，需要保留的是 qwindows.dll(少一个字母 d)

5、手动发布程序及 debug 版本的缺点：

- 很明显 debug 版本发布的程序过于庞大(因为含有调试信息)，解决此问题的办法是发布 release 版本(发布版本)的程序(对于本例的 release 版本只有十多 M)
- 手动发布程序时容易漏掉需要的库文件(dll 文件)。因为本例比较简单，所以需要的库文件比较少，若编写的程序比较复杂，则会需要很多库文件，此时使用手动发布程序就无法胜任了。

6、本地计算机上运行 exe 文件还可使用以下两种方法

- 把 exe 文件复制到 F:\app\Qt5.10.1\5.10.1\mingw53_32\bin 目录下，直接运行即可，
- 把以上路径设置为 windows 的环境变量，使用此方法后，不管生成的 exe 文件位于 windows 中的什么位置，都可直接双击运行。环境变量的设置方法为，在“我的电脑上”右击，选择“属性”，然后选择“高级系统设置”，找到“环境变量”，然后在“系统变量”栏目下找到“Path”变量，然后把以上路径添加到此变量中即可。

三、使用 windeployqt 工具部署文件(仅限 windows)

1、windeployqt 工具所在路径：F:\app\Qt5.10.1\5.10.1\mingw53_32\bin

2、在命令行下使用 windeployqt 工具部署文件

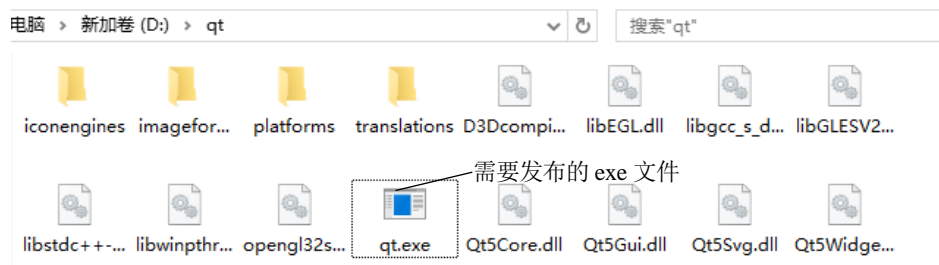
- 若要使用 windows 的 cmd 运行 windeployqt 程序，则需要转至该程序所在目录，也可配置环境变量，否则请使用开始菜单中 Qt 自带的命令行工具 Qt 5.10.1 for Desktop
- 把使用 Qt Creator 生成的 exe 文件复制到一个单独的目录中，比如 D:\qt。建议使用 release 版本以减小发布的程序的大小。
- 打开 Qt 命令行工具，并输入如下命令

```
windeployqt D:\qt\xxx.exe
```

若发布的是 Qt Quick 程序，需要使用 qmlDir 参数指定 qml 的安装目录，比如

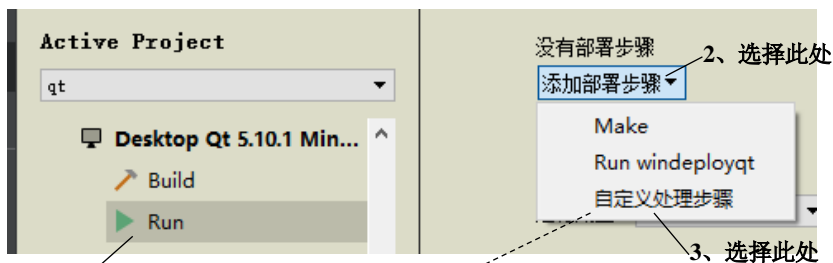
```
windeployqt D:\qt\xxx.exe --qmlDir F:\Qt\qml
```

- 打开 D:\qt 所在目录，可看到生成了一大堆文件(总计约 40M，对于本例而言，很多文件是不需要的)，如下图所示，把以下文件整体打包，复制到其他计算机上，便可直接运行了。

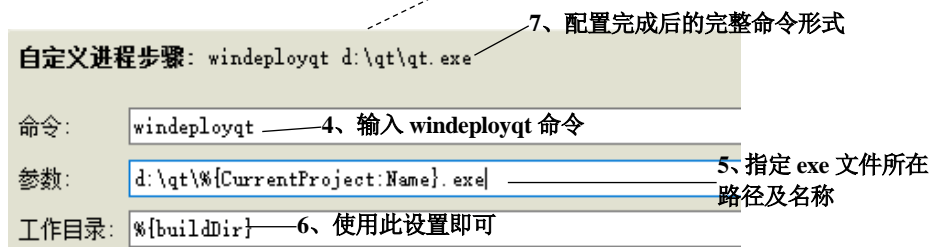


3、在 Qt Creator 中使用 windeployqt 工具部署文件，

- ①、在“项目模式”(Ctrl+5)下进行如下图所示设置

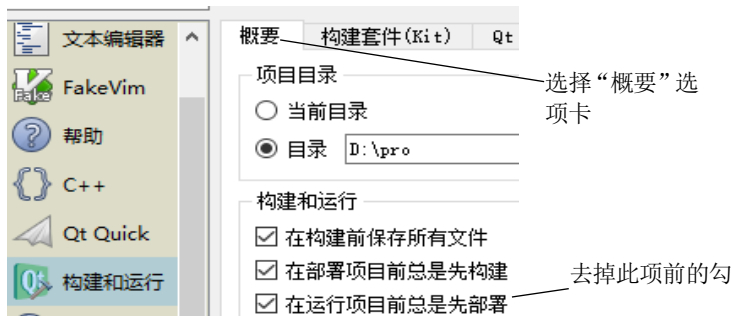


1、选择 Run



说明：1、生成的 exe 文件的路径和名称可在 pro 文件中进行设置
 2、步骤 5 使用的是一种指定目录和文件名的方式。
 3、若步骤 5 要使用默认值，则参数栏应使用如下命令
 %{buildDir}/%{CurrentBuild:Type}/%{CurrentProject:Name}.exe
 最终结果为
 windeployqt D:\pro\build-qt-Desktop_Qt_5_10_1_MinGW_32bit-Release/release/qt.exe
 其中 D:\pro 为项目文件所在目录。

- ②、重新构建项目并运行程序(注意：必须运行程序，否则 Qt Creator 不会执行设置好的 windeployqt 命令)，然后找到生成的 exe 文件所在目录，可以发现，此时 exe 文件已经部署好了，只需打包复制到别的计算机上便可运行了。
- ③、注意：使用此方法时，Qt Creator 每次运行程序都会进行部署，这会严重影响 Qt Creator 的运行速度，可在“工具”>“选项”，“构建和运行”中的“概要”选项卡下把“在运行前总是先部署”前的勾去掉，如下图所示



- 4、Mac OS 平台下，使用 Qt 的自带工具 macdeployqt 即可，其使用方法与 windows 下的 windeployqt 类似。
- 5、其他说明

- 可以用 Enigma Virtual Box 软件把多个文件封装到应用程序主文件，从而制作成为一个单独的可执行的绿色软件。
- 若项目还用了其他 SDK，比如 OpenCV 等，此时仍需要手动拷贝所需的 dll，若不知道缺少哪些 dll，则可用 Dependency Walker 软件来查看缺少哪些 dll 文件。

四、静态编译和动态编译

- 1、动态编译：Qt Creator 默认的编译方式就是动态编译，使用该方式在发布程序时，需要为发布的程序包含相应的 dll 文件。此种方式的缺点是，有时有些 dll 文件非常庞大，优点就是当发布多个程序时，可以共享 dll 文件。
- 2、静态编译：使用静态编译的程序可以直接运行，因此发布简单、文件单一、不需要太多的 dll，但缺少灵活性，不能部署插件，生成的 exe 文件较大，而且还会牵扯到 QT 的授权问题(这时需要花钱购买)。
- 3、要生成静态版本的程序，则 Qt 的库也必须是静态编译的才行，但 Qt 并不提供静态库，因此要使用静态编译需要把整个 Qt 的库重新编译为静态库，然后使用编译好的静态库来生成静态程序。所以静态编译的重点就是静态库，生成静态库的过程需要花费好几小时时间，当然也可直接从网上下载别人编译好了的静态库。具体方法从略。

1.6 Qt 的重要文件简介

一、项目文件(pro 文件)及其语法

- 1、项目文件(pro 文件)的作用是列举项目中的源文件，
- 2、pro 文件的语法形式为：“变量 操作符 值”，比如 QT += widgets，多个值之间使用空格分开。
- 3、pro 文件的注释：从“#”开始，直至本行结束。
- 4、pro 文件的操作符见下表

pro 文件的操作符	
操作符	说明
=	把值赋给变量，比如 CONFIG = qt release，表示将值 qt 和 release 赋给变量 CONFIG，这些新值会清除掉该变量以前的值。
+=	把值追加给变量，比如 QT+=widgets，表示把值 widgets 追加到变量 QT 中，以前的值不会被清除。
-=	把值从变量中移除，比如 QT -=widgets，表示把值 widgets 从变量 QT 中移除。
=	把值添加到变量中，但在这之前变量不能拥有该值，否则什么也不做。比如 QT=widgets，表示若 QT 没有值 widgets，则把值赋给 QT

- 5、pro 文件中自定义变量及使用\$\$符号引用变量，比如
AA=qt release

CONFIG=\$\$AA # 与 CONFIG=qt release 等同

6、pro 文件中的条件判断

- win 32{QT+=widgets} # 若 win32 为真，则执行大括号中的语句
 else {CONFIG = qt release} # 否则执行此处的语句，else 是可选的。
- win32: QT+=widgets # 若 win32 为真则执行冒号后的语句。
- greaterThan(QT_MAJOR_VERSION,4): QT+=widgets #若 Qt 版本大于 4，则执行冒号后的语句。

7、pro 文件常用的变量如下表

TEMPLATE	生成指定模板类型的 makefile 文件，可取以下值 <ul style="list-style-type: none">● app(默认): 生成应用程序● lib: 库文件● subdirs: 子目录，此时还需要指定 SUBDIRS 变量以指定子目录，qmake 会自动搜索以目录名命名的 pro 文件且会编译该项目● vcapp: 生成 visual studio 应用程序，仅限 windows● vclib: 生成 visual studio 库，仅限 windows
LANGUAGE	指定使用的语言，默认为 C++
HEADERS	指定项目中的所有头文件，比如 HEADERS+=xx.h
SOURCES	指定项目中的所有源文件，比如 SOURCES+= 1.cpp 2.cpp
QT	指定项目中使用的 QT 模块，默认情况下 QT 包含 core 和 gui，即默认包含 QtCore 和 QtGui 模块。QT 变量前文已经使用。
FROMS	指定项目中的 ui 文件。
DEFINES	相当于在 C++中使用#define 定义了一个符号常量，比如 DEFINES = FFF=11 则类似于在 C++中有语句: #define FFF 11
INCLUDEPATH	指定#include 语句搜索头文件时的目录，比如在 D:\a\b 下有一 c.h 的头文件，若不使用 INCLUDEPATH 变量，则应这样包含头文件#include "D:\a\b\c.h"，若使用该变量，比如 INCLUDEPATH = D:/a/b #正反斜杠符号都可以 则只需这样包含 c.h 即可: #include "c.h"
TARGET	指定生成的可执行文件的名称，若不指定则与 pro 文件同名，比如 TARGET=xx，则在 windows 下生成的应用程序的名称就是 xx.exe
DESTDIR	指定可执行文件(比如.exe 文件)放置的目录
DLLDESTDIR	指定目标库文件放置的目录
VERSION	指定目标库的版本号
LIBS	指定项目要链接的库。比如 LIBS=D:/a/b.lib
OBJECTS_DIR	指定中间文件(.o 或.obj)放置的目录
RESOURCES	指定 qrc 文件的名称
RCC_DIR	指定 qrc 文件转换为 qrc_*.h 文件的存放目录
MOC_DIR	指定来自 moc 的所有中间文件放置的目录
UI_DIR	指定 ui 文件转换为 ui_*.h 文件的存放目录
RC_FILE	指定资源文件的名称，该值很少需要修改
RC_ICONS	仅适用于 windows，指定应用程序的图标
CODECFORSRC	指定源文件的编码方式，比如 CODECFORSRC=GBK

CONFIG 变量	
作用及用法：指定配置信息，使用该变量不应使用=，否则会清除掉 CONFIG 的默认值，因此应使用+=或-=，比如 CONFIG +=debug	
可取值	说明
release	无论 Qt Creator 选择的哪个模式，都以 release 模式构建项目，即可执行文件不含有调试信息
debug	无论 Qt Creator 选择的哪个模式，都以 debug 模式构建项目，即可执行文件含有调试信息，若同时指定了 release 和 debug，则最后指定的那个模式有效。
debug_and_release	Qt Creator 选择的哪个模式，就以哪个模式构建项目。该模式会把 debug 和 release 构建在不同的目录，此时最终生成的可执行文件(如 windows 下的 exe 文件)不会放置于对应的 debug 或 release 目录下。
debug_and_release_target	若与 debug_and_release 同时指定，则会把最终生成的可执行文件(如 windows 下的 exe 文件)放置于相应的 debug 或 release 目录下
build_all	若与 debug_and_release 同时指定，则不管 Qt Creator 选择的哪个模式，都会同时以 debug 和 release 模式构建项目
构建模式默认值	debug_and_release 和 debug_and_release_target 同时指定
warn_on	编译器输出尽可能多的警告。
warn_off	编译器输出尽可能少的警告。
qt	指定应用程序或库使用 Qt，该项被默认包含。
exceptions	启用异常(默认)
exceptions_off	禁用异常
thread	启用多线程。
dll	动态编译库，仅适用于 lib 模板
staticlib	静态编译库，仅适用于 lib 模板
plugin	编译一个插件，仅适用于 lib 模板，该项会使 dll 值生效。
windows	程序是一个 windows 窗口应用程序，仅适用于 app 模板
console	程序是一个控制台应用程序，仅适用于 app 模板
app_bundle	默认，仅适用于 Mac，指可执行文件被放到束中。
lib_bundle	仅适用于 Mac，指库被放到框架中
rtti	启用 RTTI(默认情况下，同编译器默认值)。
rtti_off	禁用 RTTI
stl	启用 STL
stl_off	禁用 STL
c++11	启用 C++11 支持，默认为禁用。若编译器不支持，则此设置无影响。
c++14	启用 C++14 支持，默认为禁用。若编译器不支持，则此设置无影响。

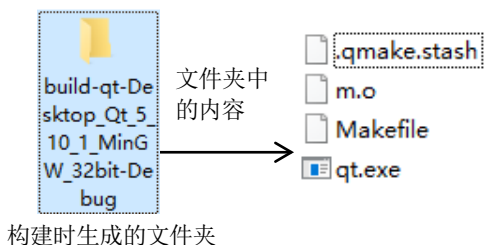
8、示例

- 由构建时生成的文件夹的名称可以判断出 Qt Creator 是以什么模式构建的项目，若名称的最后是 Debug 则是以 debug 模式构建的，若是 Release 则是以 release 模式构建的
- 根据所生成的可执行文件(比如 exe 文件)的大小，可以判断出可执行文件最终是以什么模式生成的，使用 debug 模式生成的可执行文件会比 release 模式生成的可执行文件更大，因为 debug 模式会包含一些调试信息。

示例 1:

在 pro 文件中增加如下内容, 结果如右图

```
CONFIG-=debug_and_release
CONFIG-=debug_and_release_target
CONFIG-=debug
CONFIG-=release
CONFIG+=release #以此模式构建项目
```

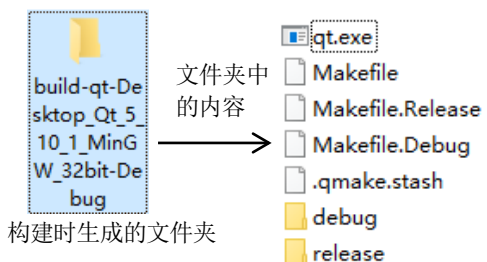


由文件夹名中的 Debug 可知, Qt Creator 选择的是以 debug 模式构建项目, 但由生成的 qt.exe 大小可以判断出, qt.exe 是 release 版本的。由此可见构建模式最终是由 pro 文件中设置的 CONFIG+=release 决定的。

示例 2:

在 pro 文件中增加如下内容, 结果如右图

```
CONFIG-=debug_and_release
CONFIG-=debug_and_release_target
CONFIG-=debug
CONFIG-=release
CONFIG+=debug_and_release
```

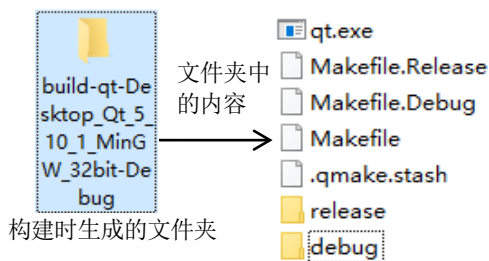


由图可见, 构建时生成的文件夹中多了两个文件夹, 分别对应 debug 和 release 两种构建模式, 而且 makefile 文件也相应的多了对应于两种模式的版本。注意 m.o 文件不在该文件夹中因为 Qt Creator 选择的是以 debug 模式构建项目, 且 pro 文件中设置的构建模式为 debug_and_release。因此最终的构建模式是 debug, 此时 m.o 文件位于 debug 文件夹中。release 文件夹此时是空的。同理若 Qt Creator 选择 release 模式构建项目, 则 debug 文件夹是空的。

示例 3:

在 pro 文件中增加如下内容, 结果如右图

```
CONFIG-=debug_and_release
CONFIG-=debug_and_release_target
CONFIG-=debug
CONFIG-=release
CONFIG+=debug_and_release
CONFIG+=build_all
```



由 pro 文件的设置可知, 无论 Qt Creator 选择的是什么模式构建的项目, 此时都会同时以两种模式构建, 因此 release 文件夹和 debug 文件夹中都各自含有一个对应版本的 m.o 文件, 但 qt.exe 文件是 release 版本的, 且位于构建时生成的文件夹之中。

示例 4:

在 pro 文件中增加如下内容, 结果如右图

```
CONFIG-=debug_and_release
CONFIG-=debug_and_release_target
CONFIG-=debug
CONFIG-=release
CONFIG+=debug_and_release
CONFIG+=build_all
CONFIG+=debug_and_release_target
```



由 pro 文件的设置可知, 无论 Qt Creator 选择的是什么模式构建的项目, 此时都会同时以两种模式构建

本例需注意的是 qt.exe 并不在构建时生成的文件夹中, 此时在 release 文件夹和 debug 文件夹中各自含有一个对应版本的 m.o 文件和 qt.ext 文件。

二、moc 简介

- 1、moc 全称是 Meta-Object Compiler, 即“元对象编译器”。
- 2、因为 Qt 不是使用的标准 C++ 语言, 因此 Qt 在将源码交给标准 C++ 编译器(如 gcc)之前, 需要先把扩展的语法去除掉。完成这一操作的就是 moc。
- 3、moc 执行步骤

Qt 程序编译之前, 先使用 moc 分析 C++ 源文件, 若在头文件中包含了宏 Q_OBJECT, 则会生成另外一个包含了 Q_OBJECT 宏实现代码的 C++ 源文件, 这个新文件的名字是在原文件名之前加上 moc_ 构成。这个新文件同样会进入编译系统, 最终被链接到二进制代码中去, 因此我们在 Qt 构建后的文件夹中, 见到 moc_*.o 和 moc_*.cpp 的文件, 就是由 moc 生成的。注意: 新文件不会“替换”掉旧的文件, 而是与原文件一起编译, 另外, moc 的执行是在预处理器之前。

三、pro、pri、prf 文件简介

- 1、pro 文件前文已详细介绍了。
- 2、pri 文件: 该类型文件类似于 C++ 中的头文件, 可以在 pro 文件中使用 include 将其包含进来, 因此 pri 文件中的语法与 pro 文件是相同的。示例如下

假设 xx.pri 文件内容如下

```
QT+=widgets
HEADERS+=widget.h
```

假设 xx.pro 文件内容如下

```
include(xx.pri)    #把 pri 文件的内容包含进来
CONFIG+=QT
```

- 3、prf 文件: 该文件类似于 pri 文件, 文件中的语法与 pro 文件相同, 也需要被包含到 pro 文件内, 但包含的方法不是使用 include, 而是用 CONFIG 变量指定或 load 函数加载, 而且 prf 文件还应放置在 qmake 能搜索到的目录中去。qmake 搜索的具体目录请参阅帮助文档, 以下为 qmake 搜索的目录之一

F:\app\Qt5.10.1\5.10.1\mingw53_32\mkspecs\features

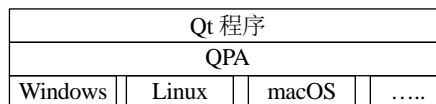
示例如下

- 假设 xx.prf 文件的内容如下
QT+=widgets
- 然后把 xx.prf 文件存储到上面所指的 features 目录下
- 在 pro 文件中包含 prf 文件的方法如下
CONFIG+=xx #也可使用 load(xx)
此时就相当于 pro 文件中有一条语句 QT+=widgets

1.7 Qt 框架结构简介

一、Qt 基本框架

- 1、Qt 5 引入了模块化的概念，Qt 5 将实现众多功能的 Qt 库细分为各个模块，也就是说一个模块中包含了实现某种功能的众多 C++类库，比如 Qt GUI 模块用于图形用户界面绘制，该模块中包含了实现 GUI 组件的类库，比如 QFont、QImage、QOpenGL、QWindow 等类都位于 QtGUI 模块中。同理 QtMultimedia 模块提供了对多媒体的支持，其中包含有对音频、视频等功能的类。Qt 4 也有模块的概念，但没有 Qt 5 划分得细。
- 2、由以上可见，开发 QT 程序，首先需要指定使用 Qt 来开发哪方面的程序，即需要指定模块，其方法就是在 pro 文件中使用 QT 变量，比如前面经常用到的 QT+=widgets 就表示使用 QtWidgets 模块。
- 3、平台抽象层：在 Qt 5 中，所有平台都是使用 Qt 平台抽象(QPA)创建的，Qt 平台抽象层是一个插件架构，它允许动态加载一个窗口系统。也就是说 QPA 是 Qt 程序和平台之间的一个中间接口，具体原理如下图。



二、Qt5 模块架构

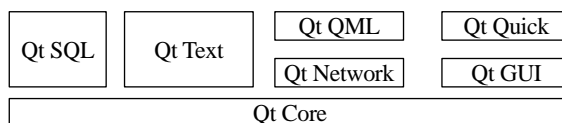
- 1、Qt 将模块分为 3 个部分：Qt Essentials(Qt 基本模块)、Qt Add-Ons(Qt 扩展模块)、Qt Tools(Qt 开发工具)
- 2、基本模块：定义了适用于所有平台的基础功能，在 Qt 5 中，它们将保持源代码和二进制兼容，因此所有程序都需要使用基本模块提供的功能(不一定需要使用所有的功能)。其中，基本模块中的 QtCore 模块是基础，所有其他的 Qt 模块都要依赖于这个模块。Qt 的基本模块见下面的表格
- 3、扩展模块：是针对某种特定目的的模块，扩展模块可在某个特定平台上使用，或者所有平台上都可使用(但不一定通用)。扩展模块请参阅帮助文档。

Qt 的基本模块	
注意：模块的名称是以 Qt 开始的，类是以 Q 开始的 以下的 pro 文件语法，指的是怎样在 pro 文件中添加该模块。	
Qt Core	1、所有其他的 Qt 模块都依赖于这个模块，这是 Qt 的核心类库，提供了非 GUI 的核心功能。 2、主要有五大功能，即元对象系统、属性系统、对象模型、对象树、信号和槽。另外还提供了一些额外的框架(比如状态机框架、动画框架等)、还有线程、输入输出等功能。 3、相对于 Qt4，在 Qt5 中增加了对 JSON 的支持，并把 XML 模块移到了该模块。 4、pro 文件语法：QT+=core，若使用 qmake 构建项目，则默认包含该模块
Qt GUI	1、这是图形用户界面(GUI)开发的最基础的模块，该模块提供了对图形用户界面的基本处理，包括窗口系统集成、事件处理、2D 图形、基本图像、字体、文本、OpenGL、OpenGL ES 集成等内容。 2、相对于 Qt4，Qt5 把图形部件类(包括重要的 QApplication)从该模块中移到了 Qt Widgets 模块、把对打印相关的类移到了 Qt Print Support 模块，同时把 OpenGL 相关类移到了该模块，同时废除了以前的 Qt OpenGL 模块。 3、pro 文件语法：QT+=gui，若使用 qmake 构建项目，则默认包含该模块
Qt Multimedia	提供音频、视频、广播、相机等功能。pro 文件语法：QT+=multimedia
Qt Multimedia Widgets	提供了额外的多媒体部件和控件的支持。pro 文件语法：QT+=multimediawidgets
Qt Network	提供了基于网络的功能。pro 文件语法：QT+=network
Qt QML	1、Qt QML 模块提供了使用 QML 语言开发程序和库的一个框架，它提供了实现 QML 语言的基础结构
Qt Quick	2、Qt Quick 模块为 QML 提供了可视化组件、动画框架等用于构件用户界面的功能。 3、Qt QML 和 Qt Quick 两个模块是从 Qt4 的 Qt Declarative 模块分离出来的。 4、注：QML 是一种脚本化编程语言。QML 和 Qt Quick 的关系类似于 C++和 Qt 的关系。 5、pro 文件语法：QT+=qml QT+=quick
Qt Quick Controls	提供用于 Qt Quick 中构建经典桌面风格的用户界面。
Qt Quick Dialogs	提供用于 Qt Quick 的系统对话框
Qt Quick Layouts	提供用于 Qt Quick 的布局
Qt SQL	提供对 SQL 的支持。pro 文件语法：QT+=sql
Qt Test	提供对 Qt 程序的单元测试工具。pro 文件语法：QT+=testlib
Qt Widgets	提供了 C++用户界面部件。pro 文件语法：QT+=widgets

4、由以上讲解可知，Qt 把功能划分为模块，模块中包含众多实现该功能的类，而有些模块又是在某些模块的基础上实现的(即需要使用该模块提供的一部分类)，由此根据模块之间的相互依赖关系可以绘出 Qt 模块的框架图。

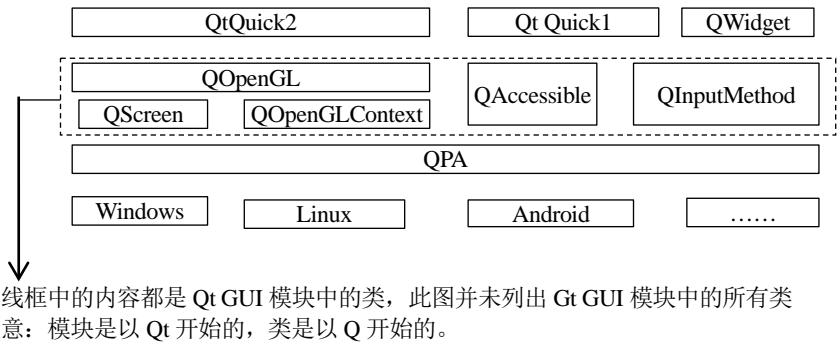
5、各模块间依赖关系简介

- 所有模块都依赖于 Qt Core
- Qt Test、Qt SQL、Qt Network、Qt GUI 直接依赖于 Qt Core
- Qt QML 依赖于 Qt Network，而 Qt Quick 又依赖于 Qt GUI 模块中的 OpenGL 功能
- 根据以上依赖关系，可画出如下图所示的 Qt 模块框架图，以下图只是一个简图，主要是为了说明怎样理解模块之间的依赖关系。



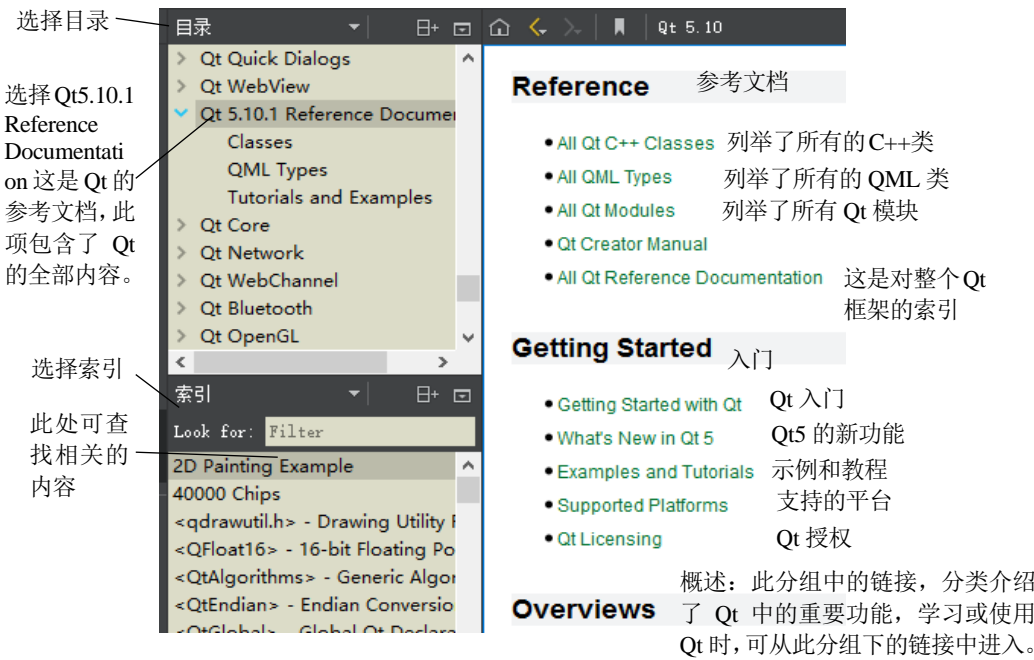
6、Qt 图形界面间的依赖关系

Qt GUI 模块中的众多类被分为两大类别，一类以 OpenGL 为核心，这是 Qt Quick2 的基础；另一类是以辅助访问和输入方式为基础的一般图形显示类，这是 QWidget 类和 Qt Quick1 模块的基础，因此可作出如下的框架图，以下图只是一个简图，主要是为了说明怎样理解模块和类之间的依赖关系。



三、帮助文档的使用

帮助文档是学习 QT 的有力助手，在 Qt Creator 中进入“帮助模式”(Ctrl+6)，其界面如下图，读者也可在 <http://do.qt.io> 中找到类似右侧图形的界面。还可在本地磁盘的 F:\app\Qt5.10.1\Docs\Qt-5.10.1\qtdoc\index.html 中使用浏览器打开帮助文档



作者：黄勇

本文作者：黄邦勇帅(原名：黄勇)

本文要求读者已经非常熟悉 C++的语法

本文是 Qt 的核心部分，主要讲解了元对象、属性系统、对象树与生命期、信号与槽、事件系统，本文全面剖析了 Qt 的元对象系统，本文由浅入深，讲解全面细致，易读易懂，学完本文能让读者真正全面的明白 Qt 元对象系统，明白 Qt 的核心内容，为后续学习 Qt 铺下坚实的基础。

本文对 C++虚函数的语法要求比较多，下面对其中的语法作一简介

```
class A{void f(){};    ma.f()
```

以上语句会被编译器内部转换为

```
class A{void void f(A* const this){}; 和 f(&ma);
```

以上语句表明 this 指针指向的是 ma，转换之后更能明白 this 指针指向的何处。

本文使用的是 windows 10 操作系统，Qt 版本为 Qt5.8.0，Qt Creator 的版本为 Qt Creator 4.2.1
本文内容完全属于个人见解与参考文献的作者无关，限于水平有限，其中难免有误解之处，望指出更正。

声明：禁止抄袭、复印、转载本文，本文作者拥有完全版权。

主要参考文献：

- 1、C++语法详解 黄勇 编著 电子工业出版社 2017 年 7 月
- 2、Qt Creator 快速入门(第 3 版) 霍亚飞 编著 北京航空航天大学出版社 2017 年 1 月
- 3、C++ GUI Qt4 编程(第 2 版) [加拿大]Jasmin Blanchette [英] Mark Summerfield 著 闫锋欣 曾泉人 张志强 译 电子工业出版社 2008 年 8 月
- 4、Qt 5 开发及实例 陆文周 主编 电子工业出版社 2014 年 1 月
- 5、精通 Qt4 编程(第 2 版) 蔡志明 卢传富 李立夏 等编著 电子工业出版社 2011 年 2 月
- 6、Qt on Android 核心编程 安晓辉 著 电子工业出版社 2015 年 1 月
- 7、C++ Qt 设计模式(第 2 版) [美] Alan Ezust、Paul Ezust 著 闫锋欣 张学敏 张君施 等译 张德保 审 电子工业出版社 2012 年 7 月
- 8、Qt 中的 C++技术 张波编著 电子工业出版社 2012 年 7 月
- 9、Head First 设计模式(中文版) Eric FreeMan , Elisabeth Freeman , Kathy Sierra , Bert Bates 著 O'Reilly Taiwan 公司译 UMLChina 改编 中国电力出版社

第 2 章 Qt 元对象系统、信号和槽及事件目录

[2.1 元对象系统](#)

[2.1.1 元对象系统基本概念](#)

[2.1.2 Q_OBJECT 宏](#)

[2.1.3 使用 Qt Creator 启动元对象系统](#)

[2.2.4 在命令行启动元对象系统](#)

[2.2 元对象](#)

[2.2.1 QByteArray 类简介](#)

[2.2.2 元对象系统与反射机制](#)

[2.2.3 使用反射机制获取类对象的成员函数的信息](#)

[2.2.4 使用反射机制获取与类相关的信息](#)

[2.3 属性系统](#)

[2.3.1 属性基础](#)

[2.3.2 QVariant 类](#)

[2.3.3 使用 QObject 类中的成员函数存取属性值与动态属性](#)

[2.3.4 使用反射机制获取属性的信息](#)

[2.4 信号与槽](#)

[2.4.1 信号和槽原理](#)

[2.4.2 创建信号和槽](#)

[2.4.3 信号和槽的关联（连接）](#)

[2.4.4 断开信号和槽的关联](#)

[2.4.5 signals、slots、emit 关键字原型](#)

[2.5 对象树与生命期](#)

[2.5.1 组合模式与对象树](#)

[2.5.2 QObject 类、对象树、生命期](#)

[2.6 事件](#)

[2.6.1 QApplication、QGuiApplication、QCoreApplication](#)

[2.6.2 Qt 对事件的描述及分类](#)

[2.6.3 事件的传递（或分发）及处理](#)

[2.6.4 事件的接受和忽略](#)

[2.6.5 事件过滤器](#)

[2.6.6 自定义事件与事件的发送](#)

[2.6.7 事件的传递顺序总结](#)

[2.6.8 鼠标和键盘事件共同使用的类及函数](#)

[2.6.9 鼠标事件](#)

[2.6.10 键盘事件](#)

第 2 部分 Qt 元对象系统、信号和槽及事件

2.1 元对象系统

一、元对象系统基本概念

- 1、Qt 的元对象系统提供的功能有：对象间通信的信号和槽机制、运行时类型信息和动态属性系统等。
- 2、元对象系统是 Qt 对原有的 C++ 进行的一些扩展，主要是为实现信号和槽机制而引入的，信号和槽机制是 Qt 的核心特征。
- 3、要使用元对象系统的功能，需要满足以下三个条件
 - ①、该类必须继承自 QObject 类。
 - ②、必须在类声明的私有区域添加 Q_OBJECT 宏，该宏用于启动元对象特性，然后便可使用动态特性、信号和槽等功能了。
 - ③、元对象编译器(moc)为每个 QObject 的子类，提供实现了元对象特性所必须的代码。
- 4、元对象系统具体运行原则
 - ①、因为元对象系统是对 C++ 的扩展，因此使用传统的编译器是不能直接编译启用了元对象系统的 Qt 程序的，对此在编译 Qt 程序之前，需要把扩展的语法去掉，该功能就是 moc 要做的事。
 - ②、moc 全称是 Meta-Object Compiler(元对象编译器)，它是一个工具(类似于 qmake)，该工具读取并分析 C++ 源文件，若发现一个或多个包含了 Q_OBJECT 宏的类的声明，则会生成另外一个包含了 Q_OBJECT 宏实现代码的 C++ 源文件(该源文件通常名称为 moc_*.cpp)，这个新的源文件要么被#include 包含到类的源文件中，要么被编译键接到类的实现中(通常是使用的此种方法)。注意：新文件不会“替换”掉旧的文件，而是与原文件一起编译。
- 5、其他概念
 - ①、元对象代码：指的是 moc 工具生成的源文件的代码，其中包含有 Q_OBJECT 宏的实现代码
 - ②、moc 工具的路径为：F:\app\Qt5.8.0MinGw\5.8\mingw53_32\bin

二、Q_OBJECT 宏

列出该宏代码的目的主要是为了讲解使用 Qt Creator 时易犯的错误。

```
#define Q_OBJECT \  
public: \  
.....  
static const QMetaObject staticMetaObject; \  
virtual const QMetaObject *metaObject() const; \  
virtual void *qt_metacast(const char *); \  
①  
②  
③
```

```

    virtual int qt_metacall(QMetaObject::Call, int, void **); \ ④
    QT_TR_FUNCTIONS \ ⑤
private: \
    static void qt_static_metacall(QObject *, QMetaObject::Call, int, void **);
.....

```

- 1、选中 Q_OBJECT 然后按下 F2 可追踪到该宏的如上定义(仅列出关键代码),
- 2、可见, Q_OBJECT 宏为声明的类之中增加了一些成员, 而且②、③、④是虚函数成员(注意, 这些虚函数没有定义), 按 C++语法, 虚函数必须定义或者被声明为纯虚函数, moc 工具的工作之一就是生成以上成员的定义, 并且还会生成一些其他必要的代码。此处讲解 Q_OBJECT 宏代码的目的主要是为了讲解使用 Qt Creator 时易犯的错误及怎样正确使用 Q_OBJECT 宏, 读者可查看由 moc 生成的 moc_*.cpp 的内容以了解更详细的信息。

三、使用 Qt Creator 启动元对象系统

- 1、此时 moc 工具是通过 Qt Creator 来使用的, 因此必须保证 moc 能发现并处理项目中包含有 Q_OBJECT 宏的类, 为此, 需要遵守以下规则
 - 从 QObject 派生的含有 Q_OBJECT 宏的类的定义必须在头文件中。
 - 确保 pro 文件中, 是否列举了项目中的所有源文件(SOURCES 变量)和头文件(HEADERS 变量)
 - 应在头文件中使用逻辑指令(比如#ifdef)防止头文件被包含多次。
 - QObject 类应是基类列表中的第一个类。
 - 由以上规则可见, 使用 Qt Creator 编写代码时, 类应定义在头文件中, 成员函数的定义应位于源文件中(这样可避免头文件被包含多次产生的重定义错误), 虽然这样编写程序比较麻烦, 但这是一种良好的代码组织方式。
- 2、不按规则 1 的方法编写程序, 则 moc 工具就不能正确生成代码, 这时的错误原因通常是未定义由 Q_OBJECT 展开后在类中声明的虚函数引起的, 其错误信息如下:
 - 若使用 MinGw 编译, 产生的错误信息类似如下:


```
undefined reference to `vtable for A'
```

 表示类 A 的虚函数表(vtable)不能正常生成, 通常是有虚函数未定义。
 - 若使用 VC++2015 编译, 产生的错误信息类似如下:


```
LNK2001: 无法解析的外部符号 "public: virtual struct QMetaObject const * thiscall A::metaObject(void)const " (.....)
```

 表示虚函数 A::metaObject 未定义。
- 3、以下错误不易被发现

若定义了 QObject 类的派生类, 并进行了构建, 在这之后再添加 Q_OBJECT 宏, 则此时必须执行一次 qmake 命令(“构建” > “执行 qmake”), 否则 moc 不能生成代码。

示例: 使用 Qt Creator 启动元对象系统

新建空项目、添加一个 m.h 和 m.cpp 文件。

/*m.h 文件内容。要在 Qt Creator 中启动元对象系统, 包含 Q_OBJECT 宏的类的定义必须位于头文件中, 否则 moc 工具不能生成元对象代码*/。

```

#ifdef M_H //用于防止头文件被多次包含的逻辑指令
#define M_H

```



```

#include<QObject>    //因为要使用 QObject 类，为此需要包含此头文件
class B{};
//错误，moc 不能启动。因为多重继承时 QObject 必须位于基类列表中的第一位。
//class A:public B,public QObject{ Q_OBJECT    };

class A:public QObject,public B{ /*要使用元对象系统必须继承自 QObject 类，且 QObject 应位于基类
                                继承列表中的第一位。*/
    Q_OBJECT    //启动元对象系统。Q_OBJECT 必须位于私有区域。
    public:     A() {qDebug("DDDD\n");}    }; //QDebug 函数用于在控制台输出一些信息

#endif // M_H

//m.cpp 文件内容。
#include "m.h"
int main(int argc, char *argv[]){ A ma;    return 0; }

```

运行以上程序，可在 debug 目录下找到一个 moc_m.cpp 的源文件，该源文件就是使用 moc 工具生成的，该源文件中的代码就是元对象代码，读者可查看其代码。若在该目录没有 moc_m.cpp 文件，说明 moc 工具未能正常启动，这时需在 Qt Creator 中执行 qmake 命令，再构建程序。

四、在命令行启动元对象系统

- 1、在命令行需使用 moc 工具，并在源文件中包含 moc 工具生成的 cpp 文件。
- 2、此时包含 Q_OBJECT 的类不需要位于头文件中，假设位于 m.cpp 文件内，内容为：

```

#include<QObject>
class A:public QObject{ Q_OBJECT
    public:     A(){}    };
int main(int argc, char *argv[]){ A ma; return 0; }

```

- 3、打开 Qt 5.8 for Desktop (MinGW 5.3.0 32 bit) 命令行工具，输入如下命令

```

moc d:\qt\m.cpp -o d:\qt\mm.cpp

```

以上命令会根据 m.cpp 生成 mm.cpp 文件，mm.cpp 文件中的代码就是元对象代码，此处 m.cpp 和 mm.cpp 都位于 d:\qt 文件夹下。

- 4、然后再次打开 m.cpp，在其中使用#include 把 mm.cpp 包含进去，如下

```

#include<QObject>
//#include "mm.cpp"    //不能把 mm.cpp 包含在类 A 的定义之前，原因见下面的注释
class A:public QObject{ Q_OBJECT
    public:     A(){}    };
#include "mm.cpp"    /*必须把 mm.cpp 包含在类 A 的定义之后，因为 mm.cpp 源文件中有对类 A 的
                    成员的定义，此时必须见到类 A 的完整定义。*/
int main(int argc, char *argv[]){ A ma; ;return 0; }

```

- 5、然后再使用 qmake 生成项目文件和 makefile 文件，再使用 mingw32-make 命令即可。

2.2 元对象

一、QByteArray 类简介

1、QByteArray 类简介

- 该类是一个用于处理字符串的类似于 C++ 的 string 类型的类，在 Qt 中，对字符串的处理，经常使用的是 QString 类，
- 该类保证字符串以 '\0' 结尾，并使用隐式共享(copy-on-write)来减少内存用量和不必要的数据复制。
- QByteArray 适合用于存储纯二进制数据和内存资源比较短缺的情况下。
- 下面是对 QByteArray 类的简单使用方法

```
#include<QByteArray>
#include<iostream>
using namespace std;
int main(int argc, char *argv[]) {
    QByteArray by("AAA");    //创建 QByteArray 的方法之一。
    const char * pc="ABC";
    QByteArray by1(pc);    //创建 QByteArray 的方法之一。
    const char *pc1=by.data(); //返回指向该字符串的 char*类型的指针
    cout<<pc1<<endl;    //输出 AAA
    by1.append("DDD");    //在末尾追加字符串
    cout<<by1.data()<<endl; //输出 by1 中的字符串 ABCDDD

    cout<<by1.size()<<endl; //输出 6，返回字符串的字符数(不含'\0')
    cout<<by1[2]<<endl; //使用下标访问单个字符，输出 C
    cout<<by1.at(1)<<endl; //at 函数类似于下标算符。输出 B
    return 0;}
```

二、元对象系统与反射机制

- 1、reflection 模式(反射模式或反射机制)：是指在运行时，能获取任意一个类对象的所有类型信息、属性、成员函数等信息的一种机制。
- 2、元对象系统与反射机制
 - ①、元对象系统提供的功能之一是为 QObject 派生类对象提供运行时的类型信息及数据成员的当前值等信息，也就是说，在运行阶段，程序可以获取 QObject 派生类对象所属类的名称、父类名称、该对象的成员函数、枚举类型、数据成员等信息，其实这就是反射机制。
 - ②、因为 Qt 的元对象系统必须从 QObject 继承，又从反射机制的主要作用可看到，Qt 的元对象系统主要是为程序提供了 QObject 类对象及其派生类对象的信息，也就是说不是从 QObject 派生的类对象，则无法使用 Qt 的元对象系统来获取这些信息。本文仅讨论各个类提供了哪方面的信息，至于这些类是怎样实现的，本文不作讨论，因为实现这些功能的代码非常多。
- 3、元对象：是指用于描述另一个对象结构的对象。使用编程语言具体实现时，其实就是一个类的对象，只不过这个对象专门用于描述另一个对象而已，比如 class B{...}; class A{...B mb;...};假设 mb 是用来描述类 A 创建的对象，则 mb 就是元对象。
- 4、Qt 具体实现反射机制的方法

- ①、Qt 使用了一系列的类来实现反射机制，这些类对对象的各个方面进行了描述，其中 `QMetaObject` 类描述了 `QObject` 及其派生类对象的所有元信息，该类是 Qt 元对象系统的核心类，通过该类的成员函数可以获取 `QObject` 及其派生类对象的所有元信息，因此可以说 `QMetaObject` 类的对象是 Qt 中的元对象。注意：要调用 `QMetaObject` 类中的成员函数需要使用 `QMetaObject` 类型的对象。
- ②、对对象的成员进行描述：一个对象包含数据成员、函数成员、构造函数、枚举成员等成员，在 Qt 中，这些成员分别使用了不同的类对其进行描述，比如函数成员使用类 `QMetaMethod` 进行描述，属性使用 `QMetaProperty` 类进行描述等，然后使用 `QMetaObject` 类对整个类对象进行描述，比如要获取成员函数的函数名，其代码如下：

```
QMetaMethod qm = metaObject->method(1);    //获取索引为 1 的成员函数
QDebug()<<qm.name()<<"\n";    //输出该成员函数的名称。
```

5、使用 Qt 反射机制的条件

- ①、需要继承自 `QObject` 类，并需要在类之中加入 `Q_OBJECT` 宏。
- ②、注册成员函数：若希望普通成员函数能够被反射，需要在函数声明之前加入 `QObject::Q_INVOKABLE` 宏。
- ③、注册成员变量：若希望成员变量能被反射，需要使用 `Q_PROPERTY` 宏。

6、Qt 反射机制实现原理简述

- ①、`Q_OBJECT` 宏展开之后有一个虚拟成员函数 `metaObject()`，该函数会返回一个指向 `QMetaObject` 类型的指针，其原型为

```
virtual const QMetaObject *metaObject() const;
```

因为启动了元对象系统的类都包含 `Q_OBJECT` 宏，所以这些类都有含有 `metaObject()` 虚拟成员函数，通过该函数返回的指针调用 `QMetaObject` 类中的成员函数，便可查询到 `QObject` 及其派生类对象的各种信息。
- ②、Qt 的 moc 会完成以下工作
 - 为 `Q_OBJECT` 宏展开后所声明的成员函数的生成实现代码
 - 识别 Qt 中特殊的关键字及宏，比如识别出 `Q_PROPERTY` 宏、`Q_INVOKABLE` 宏、`slot`、`signals` 等

三、使用反射机制获取类对象的成员函数的信息

1、`QMetaMethod` 类

- ①、作用：用于描述对象的成员函数，可使用该类的成员函数获取对象成员函数的信息。
- ②、该类拥有如下成员：

```
enum MethodType{Method, Signal, Slot, Constructor}
```

此枚举用于描述函数的类型，即：普通成员函数(Method)、信号(Signal)、槽(Slot)、构造函数(Constructor)。

```
enum Access{Private, Protected, Public}
```

此枚举主要用于描述函数的访问级别(私有的、受保护的、公有的)

- `QByteArray methodSignature() const;` //返回函数的签名(qt5.0)
- `MethodType methodType() const;` //返回函数的类型(信号、槽、成员函数、构造函数)
- `QByteArray name() const;` //返回函数的名称(qt5.0)

- `int parameterCount() const;` //返回函数的参数数量(qt5.0)
- `QList<QByteArray> parameterNames() const;` 返回函数参数名称的列表。
- `int parameterType(int index) const;`
返回指定索引处的参数类型。返回值是使用 `QMetaType` 注册的类型，若类型未注册，则返回值为 `QMetaType::UnknownType`。
- `QList<QByteArray> parameterTypes() const;` 返回函数参数类型的列表。
- `int returnType() const;`
返回函数的返回类型。返回值是使用 `QMetaType` 注册的类型，若类型未注册，则返回值为 `QMetaType::UnknownType`。
- `const char * typeName() const;` 返回函数的返回类型的名称。
- `Access access() const;` 返回函数的访问级别(私有的、受保护的、公有的)

2、`QMetaObject` 类中有关获取类对象成员函数的函数有：

①、`int indexOfMethod(const char* f) const;`

返回名为 `f` 的函数的索引号，否则返回-1。此处应输入正确的函数签名，比如函数形式为 `void f(int i,int j);`则正确的形式为 `xx.indexOfMethod("f(int,int)");` 以下形式都不是正确的形式，"`f(int i, int j)`"、"`void f(int, int)`"、"`f`"、"`void f`"等。

②、`int indexOfSignal(const char * s) const;`

返回信号 `s` 的索引号，否则返回-1，若指定的函数存在，但不是信号，仍返回-1。

③、`int indexOfConstructor(const char *c) const;` //返回构造函数 `c` 的索引号，否则返回-1

④、`int constructorCount() const;` //返回构造函数的数量。

⑤、`QMetaMethod constructor(int i) const;` 返回指定索引 `i` 处的构造函数的元数据。

⑥、`int methodCount() const;`

返回函数的数量，包括基类中的函数、信号、槽和普通成员函数。

⑦、`int methodOffset() const;`

返回父类中的所有函数的总和，也就是说返回的值是该类中的第一个成员函数的索引位置。

⑧、`QMetaMethod method(int i) const;` 返回指定索引 `i` 处的函数的元数据。

//头文件 `m.h` 的内容(文件读者自行创建)

```
#ifndef M_H    //要使用元对象系统，需在头文件中定义类。
#define M_H
#include<QObject>    //因为要使用 QObject 类，为此需要包含此头文件
```

```
class A:public QObject{
    Q_OBJECT        //启动元对象系统，必须声明此宏
public:
    //定义 2 个构造函数、1 个信号、3 个函数。
    Q_INVOKABLE A() {}    //要想函数被反射，需要指定 Q_INVOKABLE 宏。
    Q_INVOKABLE A(int) {}
    Q_INVOKABLE void f() {}
    Q_INVOKABLE void g(int i,float j) {}
    void g1() {}    //注意：此函数不会被反射。
    signals: void gb3();    };
```

```

class B:public A{
    Q_OBJECT    //要使用元对象系统,应在每个类之中都声明此宏
    public:
    //定义1个函数、2个信号
    Q_INVOKABLE void f() {}
    signals: void gb4();    void gb5();    };
#endif // M_H

```

//源文件内容(文件读者自行创建)

```

#include "m.h"
#include <QMetaMethod>
#include <QByteArray>
#include <iostream>
using namespace std;

int main() {    A ma;    B mb;    //创建两个对象
    const QMetaObject *pa=ma.metaObject();
    const QMetaObject *pb=mb.metaObject();
//以下为通过 QMetaObject 的成员函数获取的信息。
    int j=pa->methodCount(); /*返回对象 ma 中的成员函数数量,包括从父类 QObject 继承而来的5个
                                成员函数及本对象中的2个成员函数(注意,不包括 g1)、1个信号,所以
                                总数为8。*/
    cout<<j<<endl; //输出 8
    int i=pa->indexOfMethod("g(int,float)"); //获取对象 ma 中的成员函数 g 的索引号。
    cout<<i<<endl; //输出 7
    i=pa->constructorCount(); //获取对象 ma 所属类中的构造函数的数量。
    cout<<i<<endl; //输出 2
    i=pb->constructorCount(); /*获取对象 mb 所属类 B 中的构造函数的数量,因类 B 无构造函数,所以
                                返回值为0,此处也可看到,构造函数数量不包含父类的构造函数*/
    cout<<i<<endl; //输出 0。
    i=pa->indexOfConstructor("A(int)"); //获取对象 ma 所属类中的构造函数 A(int)的索引号。
    cout<<i<<endl; //输出 1。
    i=pa->indexOfSignal("gb3()"); //获取对象 ma 的信号 gb3()的索引号。
    cout<<i<<endl; //输出 5。
    i=pa->indexOfSignal("f()"); /*获取对象 ma 的信号 f()的索引号。因为成员函数 f 存在,但不是信
                                号,所以返回值为-1。*/
    cout<<i<<endl; //输出 -1。
    i=pb->methodOffset(); /*获取父类的成员函数数量,包括父类 A 及 QObject 中的成员函数,总共为8。
                                */
    cout<<i<<endl; //输出 8,此处相当于是对象 mb 自身成员函数开始处的索引号。
//以下为通过 QMetaMethod 的成员函数获取的信息。
    //获取对象 ma 的成员函数 g 的元数据。
    QMetaMethod m=pa->method(pa->indexOfMethod("g(int,float)"));
    QByteArray s= m.name(); //获取成员函数 g 的函数名。
    cout<<s.data()<<endl; //输出 g
    s=m.methodSignature(); //获取函数 g 的签名
    cout<<s.data()<<endl; //输出 g(int,float)
    i=m.methodType(); /*获取函数 g 的类型,此处返回的是 QMetaMethod::MethodType 中定义的枚举值,
                                其中 Method=0,表示类型为成员函数*/
    cout<<i<<endl; //输出 0(表示成员函数)。
//以下信息与函数的返回类型有关

```

```

s=m.typeName(); //获取函数 g 的返回值的类型名
cout<<s.data()<<endl; //输出 void
i=m.returnType(); /*获取函数 g 返回值的类型，此处的类型是 QMetaType 中定义的枚举值，其中枚举值 QMetaType::void=43*/
cout<<i<<endl; //输出 43
//以下信息与函数的参数有关
i=m.parameterType(1); /*获取函数 g 中索引号为 1 的参数类型，此处的类型是 QMetaType 中定义的枚举值，其中枚举值 QMetaType::float=38*/
cout<<i<<endl; //输出 38
QList<QByteArray> q=m.parameterNames(); //获取函数 g 的参数名列表
cout<<q[0].data()<<q[1].data()<<endl; //输出 ij
q=m.parameterTypes(); //获取函数 g 的参数类型列表。
cout<<q[0].data()<<q[1].data()<<endl; //输出 intfloat
return 0; }

```

四、使用反射机制获取与类相关的信息

1、QMetaObject 类中获取与类相关的信息的成员函数有

- `const char* className() const;`
获取类的名称，注意，若某个 `QObject` 的子类未启动元对象系统(即未使用 `Q_OBJECT` 宏)，则该函数将获取与该类最接近的启动了元对象系统的父类的名称，而不再返回该类的名称，因此建议所有的 `QObject` 子类都使用 `Q_OBJECT` 宏。
- `const QMetaObject* superClass() const;`
//返回父类的元对象，若没有这样的对象则返回 0。
- `bool inherits(const QMetaObject* mo) const; (Qt5.7)`
//若该类继承自 `mo` 描述的类型，则返回 `true`，否则返回 `false`。类被认为继承自身。

2、QObject 类中获取与类相关的信息的成员函数有

- `bool inherits(const char* className) const;`
//若该类是 `className` 指定的类的子类则返回 `true`，否则返回 `false`。类被认为继承自身。

示例：继承关系的判断

//头文件 `m.h` 的内容

```

#ifndef M_H //要使用元对象系统，需在头文件中定义类。
#define M_H
#include<QObject>
class A:public QObject{ Q_OBJECT};
class B:public A{ Q_OBJECT};
class C:public QObject{Q_OBJECT};
class D:public C{};
#endif // M_H

```

//源文件 `m.cpp` 的内容

```

#include "m.h"
#include <QMetaMethod>
#include <iostream>
using namespace std;
int main() { A ma; B mb; C mc; D md;

```

```

    const QMetaObject *pa=ma.metaObject();
    const QMetaObject *pb=mb.metaObject();
    cout<<pa->className()<<endl; //输出类名 A
//使用 QMetaObject::inherits() 函数判断继承关系。
    cout<<pa->inherits(pa)<<endl; //输出 1, 类被认为是自身的子类
    cout<<pa->inherits(pb)<<endl; //输出 0, 由 pb 所描述的类 B 不是类 A 的子类。
    cout<<pb->inherits(pa)<<endl; //输出 1, 由 pa 所描述的类 A 是类 B 的子类。
//使用 QObject::inherits() 函数判断继承关系。
    cout<<ma.inherits("B")<<endl; //输出 0, 类 A 不是类 B 的子类。
    cout<<ma.inherits("A")<<endl; //输出 1, 类被认为是自身的子类
    cout<<md.inherits("D")<<endl; //输出 0, 因为类 D 未启动元对象系统。
    cout<<md.inherits("C")<<endl; /*输出 1, 虽然类 D 未启动元对象系统, 但类 C 已启动, 此种情形下
                                   能正确判断继承关系。*/
    cout<<md.metaObject()->className()<<endl; /*输出 C, 此处未输出 D, 因为类 D 未启动元对象系统,
                                                与类 D 最接近的启动了元对象系统的父类是 C, 因此返回 C。*/

    return 0;    }

```

3、qobject_cast 函数，使用语法如下

DestType* qobject_cast<DestType*>(QObject *p);

- 该函数类似于 C++ 中的 dynamic_cast，但执行速度比 dynamic_cast 更快，且不需要 C++ 的 RTTI 的支持，但 qobject_cast 仅适用于 QObject 及其派生类。
- 主要作用是把源类型 QObject 转换为尖括号中的目标类型 DestType(或者子类型)，并返回指向目标类型的指针，若转换失败，则返回 0。或者说源类型 QObject 属于目标类型 DestType(或其子类型)，则返回指向目标类型的指针，否则返回 0。
- 使用 qobject_cast 的条件：目标类型 DestType 必须继承(直接或间接)自 QObject，并使用 Q_OBJECT 宏。

示例：qobject_cast 及其应用

//头文件 m.h 的内容。

```

#ifndef M_H //要使用元对象系统，需在头文件中定义类。
#define M_H
#include <QObject>
#include <iostream>
using namespace std;

class A:public QObject{ Q_OBJECT
    public: void fa() {cout<<"FA"<<endl;}    };
class B:public A{ Q_OBJECT
    public: void fb() {cout<<"FB"<<endl;}    };
class C:public QObject{Q_OBJECT
    public: void fc() {cout<<"FC"<<endl;}    };
class D:public C{ public: void fd() {cout<<"FD"<<endl;}    };
#endif // M_H

```

//源文件 m.h 的内容。

```

#include "m.h"
#include <QMetaMethod>
#include <iostream>
using namespace std;

```

//qobject_cast 的简单应用(类型判断)


```

void g(QObject *p) {
    if(qobject_cast<A*>(p)) //若 p 是类 A 及其派生类类型
        {cout<<"GA"<<endl;}
    if(qobject_cast<B*>(p))//若 p 是类 B 及其派生类类型
        {cout<<"GB"<<endl;}
    else //若 p 不是类 B 及其派生类类型
        cout<<"XX"<<endl;    }
}

int main() {    A *pa=new A;    B *pb=new B;    C *pc=new C;    D *pd=new D;
    qobject_cast<B*>(pa)->fb(); //输出 FB, 转换成功后可调用于子类中的函数。
    //qobject_cast<D*>(pc); //错误, 因为类 D 未使用 Q_OBJECT 宏。
    g(pa); //输出 GA、XX。因为 pa 不是 B 及其派生类类型所以会输出 XX。
    g(pb); //输出 GA、GB。因为 pb 是 A 的派生类类型, 所以首先输出 GA, 然后输出 GB。
    g(pc); //输出 XX, 因为 pc 即不是 A 也不是 B 及其派生类的类型, 所以输出 XX。
    g(pd); //输出 XX, 原因同上。
    return 0;    }

```

2.3 属性系统

一、属性基础

- 1、属性与数据成员相似，但是属性可使用 Qt 元对象系统的功能。他们的主要差别在于存取方式不相同，比如属性值通常使用读取函数(即函数名通常以 `get` 开始的函数)和设置函数(即函数名通常以 `set` 开始的函数)来存取其值，除此种方法外，Qt 还有其他方式存取属性值。
- 2、在 Qt 中属性和数据成员是两个不同的概念，他们可以相关联也可以没有联系，比如名为 `a` 的属性，与数据成员 `a`，虽然他们名称相同，若他们之间没有产生关联，则数据成员 `a` 与属性 `a` 是完全不相关的，通常，一个属性都有与之相关联的数据成员，而采用的命名规则通常是加上 `m_` 前缀，比如属性名为 `a`，则与之相关联的数据成员名称通常为 `m_a`。
- 3、属性值可使用以下方式进行存取
 - 可使用 `QObject::property` 和 `QObject::setProperty` 函数进行存取
 - 若属性有相关联的存取函数，则可使用存取函数进行存取
 - 属性还可通过元对象系统的 `QMetaObject` 类进行存取。
 - 若属性与某个数据成员相关联，则可通过存取普通数据成员的值来间接存取属性的值。
 - 注意：Qt 中的类，只有属性没有数据成员，因此只能通过前面三种方式对属性值进行修改。
- 4、要在类之中声明属性，该类必须继承自 `QObject` 类，还应使用 `Q_OBJECT` 宏，然后使用 `QObject::Q_PROPERTY` 宏声明属性，该宏语法如下：

```
Q_PROPERTY(type name
```

```
(READ getFunction [WRITE setFunction])
```



```

    MEMBER memberName [(READ getFunction | WRITE setFunction)]
    [RESET resetFunction]
    [NOTIFY notifySignal]
    [REVISION int]
    [DESIGNABLE bool]
    [SCRIPTABLE bool]
    [STORED bool]
    [USER bool]
    [CONSTANT]
    [FINAL]

```

- 方括号中的是可选项，各选项之间使用空格隔开。
- **type:** 指定属性的类型，可以是 `QVariant` 支持的类型或用户自定义类型，若是枚举类型，还需使用 `Q_ENUMS` 宏对枚举进行注册，若是自定义类型，需要使用 `Q_DECLARE_METATYPE(Type)` 宏进行注册(详见后文)。
- **name:** 指定属性的名称。
- **READ getFunction:**
 - 用于指定读取函数(读取属性的值)，其中 **READ** 表示读取，不可更改，`getFunction` 用于指定函数名称。
 - 若没有指定 **MEMBER** 变量，则必须指定 **READ** 函数。
 - 通常，**READ** 函数是 `const` 的，**READ** 函数必须返回属性的类型或对该类型的引用。
- **WRITE setFunction:**
 - 用于指定设置函数(设置属性的值)，其中 **WRITE** 表示写入，不可更改，`setFunction` 用于指定函数名称。
 - 此函数必须只有一个参数，且返回值类型必须为 `void`。
 - 若为只读属性，则不需要指定 **WRITE** 函数。
- **MEMBER memberName**
 - 用于把指定的成员变量 `memberName` 设置为具有可读和可写性质，而无需创建 **READ** 和 **WRITE** 函数。其中 **MEMBER** 表示成员，不可更改，`memberName` 表示类中的成员变量。
 - 若没有指定 **READ** 函数，则必须指定 **MEMBER** 变量。
- **RESET resetFunction:** 用于把属性重置为默认值，该函数不能有参数，且返回值必须为 `void`。其中 **RESET** 表示重置，不可更改，`resetFunction` 用于指定函数名称。
- **NOTIFY notifySignal:** 表示给属性关联一个信号。如果设置该项，则应指定该类中已经存在的信号，每当属性值发生变化时就会发出该信号。若使用 **MEMBER** 变量，则 **NOTIFY** 信号必须为零或一个参数，且参数必须与属性的类型相同，该参数将接受该属性的新值。
- **REVISION:** 设置版本号，默认为 0。
- **DESIGNABLE:** 用于设置属性在 GUI 设计工具的属性编辑器(例如 Qt 设计师)中是否可见，多数属性是可见的，默认值为 `true`，即可见。该变量也可以指定一个返回

布尔值的成员函数替代 true 或 false。

- **SCRIPTABLE:** 设置属性是否可被脚本引擎访问，默认为 true
- **STORED:** 设置在保存对象的状态时是否必须保存属性的值。大多数属性此值为 true
- **USER:** 设置属性是否为可编辑的属性，每一个类只有一个 USER 属性(默认为 false)。比如 QAbstractButton::checked 是用户可编辑的属性
- **CONSTANT:** 表明属性的值是常量，常量属性不能有 WRITE 函数和 NOTIFY 信号。对于同一个对象实例，每一次使用常量属性的 READ 函数都必须得到相同的值，但对于类的不同实例，这个值可以不同。
- **FINAL:** 表示属性不能被派生类重写。

示例：声明及使用属性

头文件 m.h 内容

```
#ifndef M_H    //要使用元对象系统，需在头文件中定义类。
#define M_H
#include<QObject>

class A:public QObject{    Q_OBJECT
public:
/*通常，若属性名为 a，则相应的读取函数通常命名为 geta，设置函数命名为 seta，本例并未使用这种命名规则*/

    //声明一个类型为 int，名称为 a 的属性，并使用函数 f 读取属性值，使用函数 g 设置属性值。
    Q_PROPERTY(int a READ f WRITE g)

    //声明一个类型为 int，名称为 b 的属性，并把该属性与成员变量 m_b 相关联，该属性未设置存取函数。
    Q_PROPERTY(int b MEMBER m_b)

    //声明一个只读属性 c，本例没有设置该属性值的函数，因此该属性是只读的。
    Q_PROPERTY(int c READ getc)

/*在存取函数中可把属性与成员变量相关联，方法如下所示。对存取函数的返回类型和参数类型及数量在本例影响不大，在后文会讲解其影响。*/
    int f() {return m_a;}    //属性 a 的读取函数。
    void g(int i) {m_a=i;}  //属性 a 的设置函数。
    int getc() {m_c=3; return m_c;} /*成员变量也可以不与属性相关联，本函数也可直接返回数值 3。
                                    从此处可看到，属性可以不与数据成员相关联。*/

    int m_a,m_b; //属性若命名为 a，则与其相对应的成员变量习惯上应命名为 m_a。
    private:int m_c; };//成员变量通常都应声明为私有的，这样可提高程序的封装性。
#endif // M_H
```

//源文件内容

```
#include "m.h"
#include <iostream>
using namespace std;
int main(int argc, char *argv[]){    A ma;
    //像普通成员变量一样存取属性值
    ma.m_a=1;    cout<<ma.m_a<<endl; //输出 1
    //因为属性 b 没有存取函数，本例暂时只使用普通成员变量的方式存取该属性值。
```

```

ma.m_b=2;      cout<<ma.m_b<<endl;  //输出 2
ma.g(4);        //使用属性 a 的设置函数修改属性值。
cout<<ma.f()<<endl;  // 输出 4，使用属性 a 的读取函数读取属性值。
cout<<ma.getc()<<endl; /*输出 3，属性 c 是只读的，只能通过他的读取函数访问其值，因为没有设置
                        函数，因此无法改变属性 c 的值。*/

return 0; }

```

二、QVariant 类

- 1、使用 `QObject::property` 函数可读取属性的值，使用 `QObject::setProperty` 函数可以设置属性的值，但是属性有很多种类型，怎样使用 `property` 函数返回的属性值具有正确的类型呢？为解决这个问题，使用了一个 `QVariant` 来描述类型。
- 2、`QVariant` 类用于封装数据成员的类型及取值等信息，该类类似于 C++ 共用体 `union`，一个 `QVariant` 对象，一次只能保存一个单一类型的值。该类封装了 Qt 中常用的类型，对于 `QVariant` 不支持的类型（比如用户自定义类型），则需要使用 `Q_DECLARE_METATYPE(Type)` 宏进行注册(详见后文)。
- 3、`QVariant` 拥有常用类型的单形参构造函数，因此可把这些常用类型转换为 `QVariant` 类型，同时 `QVariant` 还重载了赋值运算符，因此可把常用类型的值直接赋给 `QVariant` 对象。注：由 C++ 语法可知，单形参构造函数可进行类型转换。

- 4、使用 `QVariant` 构造函数和赋值运算符，见下面示例

注意：`QVariant` 没有 `char` 类型的构造函数，若使用 `char` 值会被转换为对应的 `int` 型。

```
QVariant v(1); //调用 QVariant(int) 构造函数创建一个 QVariant 类型的对象，并把数值 1 保存到 v 中。
```

```
v=2.2;        //调用 QVariant 的赋值运算符，把值 2 保存在 v 之中，因为 QVariant 是类似于共用体的类，
                因此同一时间只会保存一个值。
```

- 5、获取 `QVariant` 对象存储的值的类型，可使用如下函数

- `Type type() const`

获取 `QVariant` 对象当前存储值的类型，类型以枚举 `QMetaType::Type` 的形式表示

- `const char * typeName() const;`

以字符串的形式返回 `QVariant` 对象存储的值的类型。若是无效类型则返回 0。

- `const char* typeName(int t);`

把以枚举类型 `QMetaType::Type` 表示的类型以字符串的形式返回。若枚举值为 `QMetaType::UnknownType` 或不存在，则返回一个空指针。

- 示例

```

QVariant v(1);
cout<<v.typeName()<<endl;      //输出 int
cout<<v.typeName(v.type())<<endl; //输出 int

```

- 6、获取和设置 `QVariant` 对象存储的值，可使用如下函数

- `void setValue(const T& v);`

把一个值的副本存储到 `QVariant` 对象中，若类型 `T` 是 `QVariant` 不支持的类型，则使用 `QMetaType` 来存储该值，若 `QMetaType` 也不能处理，则发生编译错误。注：若是用户自定义类型则需要使用宏 `Q_DECLARE_METATYPE(...)` 进行注册(详见后文)

- `T value() const;`
把存储的值转换为类型 `T` 并返回转换后的值，存储值本身不会被改变。若 `T` 是 `QVariant` 支持的类型，则该函数与 `toInt`、`toString` 等函数功能完全相同。注：使用该函数时需要使用尖括号指定 `T` 的类型，比如 `xx.value<int>()`;
- `T toT()`
 - 其中 `T` 是某一类型，比如若 `T` 是 `int`，则该函数形式就为 `int toInt()`。
 - 该函数用于把存储的值转换为类型 `T` 并返回转换后的值，存储值本身不会被改变。其中比较常用的是 `toString` 函数，该函数可把存储的值转换为 `QString` 形式，这样便可以字符串的形式输出存储的值。
 - 注意：没有与自定义类型相对应的 `toT` 函数，比如 `class C{}`；则没有 `toC` 函数，要把存储的值转换为自定义类型，需要使用 `value` 函数，且还需对自定义类型注册。

7、注意： `QVariant` 中的枚举类型 `Type` 已被废弃。

8、使用 `QVariant` 的默认构造函数，将创建一个无效的 `QVariant` 对象(或空的 `QVariant` 对象)，可通过 `isNull()`成员函数进行判断。

示例：QVariant 类的使用

```
#include<QVariant>
#include <iostream>
using namespace std;

class C{};          //自定义类型
int main(int argc, char *argv[]) {
    QVariant v('a'); /*QVariant 没有专门的 char 构造函数，此处的字符 a 会被转换为 int 型，因此 v
                      中存储的是数值 97，而不是字符 a 。*/
    cout<<v.value<int>()<<endl;          //输出 97
    cout<<v.value<char>()<<endl;          //输出 a，将 97 转换为 char 型，并输出转换后的值。
    cout<<v.toChar().toLatin1()<<endl;    /*输出 a，原因同上，注意 toChar 返回的类型是 QChar 而不
                                           是 char。*/
    cout<<v.toString().toString()<<endl; /*输出 97，把存储在 v 中的值转换为 QString，然后以字
                                           符串形式输出。*/
    cout<<v.typeName()<<endl;              //输出 int，可见存储在 v 中的值的类型为 int
    cout<<v.typeToName(v.type())<<endl; /*输出 int，其中 type 返回存储值的枚举形式表示的类型，而
                                           typeToName 则以字符串形式显示该枚举值所表示的类型。*/

    char c='b';
    v.setValue(c);
    cout<<v.toString().toString()<<endl; //输出 b
    cout<<v.typeName()<<endl; /*输出 char，若是使用 QVariant 构造函数和直接赋值 char 型字符，此
                              处会输出 int，这是 setValue 与他们的区别。*/

    C mc;                //自定义类型 C 的对象 mc
    //QVariant v1(mc);   //错误，没有相应的构造函数。
    QVariant v2;
    //v2=mc;             //错误，没有与类型 C 匹配的赋值运算符函数。
    //v2.setValue(mc);   //错误，自定义类型 C 未使用宏 Q_DECLARE_METATYPE 声明。
    return 0;
}
```

三、使用 QObject 类中的成员函数存取属性值与动态属性

1、注册自定义类型与 QMetaType 类

- ①、QMetaType 类用于管理元对象系统中命名的类型，该类用于帮助 QVariant 中的类型以及队列中信号和槽的连接。它将类型名称与类型关联，以便在运行时动态创建和销毁该名称。
- ②、QMetaType::Type 枚举类型定义了 QMetaType 支持的类型。其原型为
`enum Type{void, Bool,Int,.....UnknowType}` //全部类型详见帮助文档。
- ③、对于 QVariant 类和属性中使用的自定义类型，都需要进行注册，然后才能使用。使用宏 `Q_DECLARE_METATYPE()` 声明新类型，使它们可供 QVariant 和其他基于模板的函数使用。调用 `qRegisterMetaType()` 将类型提供给非模板函数。

④、使用 `Q_DECLARE_METATYPE(Type)` 宏声明类型

- 使用该宏声明类型之后，会使所有基于模板的函数都知道该类型。
- 使用该宏的类需要具有 `public` 默认构造函数、`public` 析构函数和 `public` 复制构造函数。
- 使用该宏时，被声明的类型 `Type` 需要是完全定义的类型，因此，该宏通常位于类或结构的声明之后。
- 对于指针类型，需要使用 `Q_DECLARE_OPAQUE_POINTER(T)` 宏进行声明。
- 对于 QVariant 类，只需使用该宏声明类型之后便可使用该类型了。
- 若需要在队列中的信号和槽连接中，或 QObject 的属性系统中使用该类型，则还必须调用 `qRegisterMetaType` 函数注册该类型，因为这些情况是动态运行的。
- 以下类型会自动注册，不需使用此宏(全部内容详见帮助文档)
 - 指向从 QObject 派生的类的指针类型。
 - 使用 `Q_ENUM` 或 `Q_FLAG` 注册的枚举
 - 具有 `Q_GADGET` 宏的类。
- 示例：

```
class A{}; Q_DECLARE_METATYPE(A) //声明位于类定义之后
namespace N{ class B{}; } Q_DECLARE_METATYPE(N::B) //类型位于名称空间中的情形
A ma; QVariant v; v.setVaule(ma); ma.value<A>(); //声明后 QVariant 类便可直接使用
```

④、使用 `int qRegisterMetaType<T>()` 函数注册类型

- 使用该函数时需要使用尖括号指定 `T` 的类型，比如 `qRegisterMetaType<int>()`
- 该函数返回 QMetaType 使用的内部 ID。
- 类型 `T` 必须使用 `Q_DECLARE_METATYPE(Type)` 宏声明
- 类型注册后，就可以在运行时动态创建和销毁该类型的对象了。
- 被注册的类或结构需要具有 `public` 默认构造函数、`public` 析构函数和 `public` 复制构造函数。

示例：声明与注册类型

```
#include<QVariant>
#include <iostream>
using namespace std;

class A{public: int i;};
```

```

class B{public:int i;};
class D{public:D(int) {};};//该类无 public 默认构造函数
class E{ };

//声明类型
Q_DECLARE_METATYPE(A)
Q_DECLARE_METATYPE(B)
//Q_DECLARE_METATYPE(D) //错误, 类 D 没有公有的默认构造函数
//Q_DECLARE_METATYPE(F) //错误, 因为父类 QObject 的复制构造函数、赋值运算符等是私有的。

int main(int argc, char *argv[]) {
//注册类型
    qRegisterMetaType<B>();
    //qRegisterMetaType<E>(); //错误, 类型 E 未使用宏 Q_DECLARE_METATYPE(T) 声明
    // qRegisterMetaType<F>();
    A ma;   ma.i=1;
    B mb;   mb.i=2;

    //QVariant v1(ma); //错误, 没有相应的构造函数。
    QVariant v;
    v.setValue(ma); //将对象 ma 存储在 v 之中
    cout<<v.value<A>().i<<endl; //输出 1。
    cout<<v.typeName()<<endl; //输出 A
    cout<<v.toString().toString()<<endl; //输出一个空字符, 因为 ma 是一个对象, 不是一个值。

//自定义类型需要使用 userType 才能返回正确的类型 ID。
    cout<<v.typeToName(v.userType())<<endl; //输出 A
    cout<<v.typeToName(v.type())<<endl; //不一定输出 A。

    A mal;
    mal=v.value<A>(); //把存储在 v 之中的对象 ma 赋值给 mal
    cout<<mal.i<<endl; //输出 1, 可见赋值成功。
    B mbl;
    //mbl=v.value<A>(); //错误, 类型不相同。
    mbl=v.value<B>(); //正确, 由类型 A 转换到类型 B 失败, 此时 value 会返回一个默认构造的值。
    cout<<mbl.i<<endl; //输出 0。
    return 0; }

```

2、QVariant QObject::property(const char* name) const;

作用：获取属性名称为 name 的值，该值以 QVariant 对象的形式返回。若属性 name 不存在，则返回的 QVariant 对象是无效的。

3、setProperty 函数及动态属性

```
bool QObject::setProperty(const char* name, const QVariant &v);
```

- 作用：把属性 name 设置为值 v。
- 若属性使用 Q_PROPERTY 进行了声明，且值 v 与属性 name 的类型兼容，则把值 v 存储在属性 name 中，并返回 true，
- 若值与属性的类型不兼容则属性不会更改，并返回 false。

动态属性

- 若属性 `name` 未使用 `Q_PROPERTY` 进行声明，则把该属性和值作为新属性添加到对象中，并返回 `false`，这就是动态属性。
- 动态属性仍可使用 `property` 进行查询，还可设置一个无效的 `QVariant` 对象来删除动态属性。
- 动态属性是基于某个类的实例的，也就是说动态属性是添加到某个类的对象中的，而不是添加到 `QMetaObject` 中的，这意味着，无法使用 `QMetaObject` 的成员函数获取动态属性的信息。
- 更改动态属性的值，会发送 `QDynamicPropertyChangeEvent` 到该对象。

示例：动态属性及使用 `property` 和 `setProperty` 存取属性值。

头文件 `m.h` 的内容

`#ifndef M_H` //要使用元对象系统，需在头文件中定义类。

```
#define M_H
#include<QObject>
class B{public:int i;};
class C{public:int i;};
class D{public:int i;};
Q_DECLARE_METATYPE(B)
Q_DECLARE_METATYPE(C)

class Z:public QObject{    Q_OBJECT
public:    Z() {}
    Q_PROPERTY(B b READ fb WRITE gb)
    Q_PROPERTY(C c READ fc WRITE gc)
    Q_PROPERTY(D d READ fd WRITE gd)
    B fb() {return m_mb;}    void gb(B x) {m_mb=x;}
    C fc() {return m_mc;}    void gc(C x) {m_mc=x;}
    D fd() {return m_md;}    void gd(D x) {m_md=x;}
    B m_mb;    C m_mc;    D m_md;    };
#endif // M_H
```

//源文件的内容

```
#include "m.h"
#include<QVariant>
#include <iostream>
using namespace std;

int main(int argc, char *argv[]) {
//注册类型
    qRegisterMetaType<B>();
    qRegisterMetaType<C>();
    B mb;    C mc;    D md;    Z mz;
    mb.i=2; mc.i=3; md.i=4;
    mz.gb(mb);    mz.gc(mc);    mz.gd(md);
//使用 property 和 setProperty 存取属性值。
    //mz.property("d"); //错误，不能使用 property 函数访问属性 d，因为属性 d 的类型 D 未注册。
    mz.property("MMM"); /*这是正确的，因为属性 MMM 不存在，所以，返回的是一个空的 QVariant 对象，
        可见，属性不存在与属性的类型未注册是不同的。*/
}
```



```

cout<<mz.f d().i<<endl;    /*输出 4。虽然不能使用 property 函数访问属性 d，但仍可使用存取函数
                             访问该属性的值。*/

QVariant v;    B mb1;
mb1.i=6;    v.setValue(mb1);
//mz.setProperty("b",mb1); //错误，第二个参数的类型不匹配。
mz.setProperty("b",v);    //正确设置属性 b 的值的方法，把属性 b 的值设置为 v 中存储的值 mb1。
mz.setProperty("c",v);    /*正确，但是属性 c 的类型与 v 中存储的值的类型不兼容，因此属性 c 不
                             会被更改*/
mz.setProperty("c",7);    //原因同上。
cout<<mz.property("b").typeName()<<endl;    //输出 B，输出属性 b 的类型
cout<<mz.property("c").typeName()<<endl;    //输出 C，输出属性 c 的类型
cout<<mz.property("b").value<B>().i<<endl;    //输出 6。输出的是 mb1.i 的值。
cout<<mz.property("c").value<C>().i<<endl;    //输出 3，属性 c 的值并未被更改。
//动态属性
mc.i=7;    v.setValue(mc);
// mz.setProperty("w",mc); //错误，第二个参数的类型不匹配。
mz.setProperty("x",v);    //动态属性，新增加属性 x，并设置其值为 v 中存储的值(即 mc)
cout<<mz.property("x").typeName()<<endl;    //输出 C，即动态属性 x 的类型。
cout<<mz.property("x").value<C>().i<<endl;    //输出 7。
Z mz1;
//cout<<mz1.property("x").typeName()<<endl;    //错误，动态属性 x 是基于对象 mz 的。
cout<<mz1.property("b").typeName()<<endl;    //输出 B，属性 b 不是动态属性。
return 0; }

```

四、使用反射机制获取属性的信息

1、QMetaProperty 类

- ①、作用：用于描述对象的属性，可使用该类的成员函数获取对象属性的信息。
- ②、该类拥有一系列的返回 bool 值的成员函数，用于判断属性的行为(见下表)

以下函数的返回类型都为 bool			
isReadable()	可读返回 true	isEnumType()	若属性的类型是枚举，则返回 true
isWritable()	可写返回 true	isFinal()	声明属性时 FINAL 是否为 true
isValid()	属性有效则返回 true。	isFlagType()	若属性的类型是标志枚举，则返回 true
isConstant()	声明属性时 CONSTANT 是否为 true	isResettable()	若属性可被重置为默认值同返回 true， 即声明属性时指定了 RESET 函数
bool isUser(const QObject* object=Q_NULLPTR) const		声明属性时 USER 是否为 true	
bool isStored(const QObject* object=Q_NULLPTR) const		声明属性时 STORED 是否为 true	
bool isScriptable(const QObject* object=Q_NULLPTR)		声明属性时 SCRIPTABLE 是否为 true	
bool isDesignable(const QObject* object=Q_NULLPTR)		声明属性时 DESIGNABLE 是否为 true	

③、其他成员函数如下

- const char* name() const; 获取属性的名称
- const char* typeName() const; 返回此属性类型的名称。
- QVariant::Type type() const; 返回属性的类型，其值为 QVariant::Type 的枚举值之一。
- int userType() const;

返回此属性的用户类型，返回值是使用 `QMetaType` 注册的值之一(是 `QMetaType` 类中的一个枚举值)，若类型未注册，则返回 `QMetaType::UnknownType`

- `int propertyIndex() const;` 返回此属性的索引。
- `QMetaEnum enumerator() const`
若属性的类型是枚举类型则返回该枚举，否则返回的值是未定义的。
- `QVariant read(const QObject* object) const;`
从给定的对象 `object` 读取属性的值，若能读取值则返回该值，否则返回一个无效的 `QVariant` 对象。
- `bool write(QObject* object, const QVariant & v) const;`
把值 `v` 作为属性值写入给定的对象 `object` 中，若写入成功，则返回 `true`，否则返回 `false`。若值的类型与属性的类型不相同，则尝试进行转换，若属性是可重置的，则空的 `QVariant` 对象(即无效的 `QVariant` 对象)等价于调用 `reset()`函数，或者以其他方式设置一个默认构造的对象。
- `bool reset(QObject* object) const;`
使用 `RESET` 函数重置给定的对象 `object` 的属性，若重置成功，则返回 `true`，否则返回 `false`
- `bool hasNotifySignal() const;`
若属性有通知信号，则返回 `true`，否则返回 `false`。
- `QMetaMethod notifySignal() const;`
若属性指定了通知信号，则返回该信号的 `QMetaMethod` 实例，否则返回无效的 `QMetaMethod`
- `int notifySignalIndex() const;` 返回属性通知信号的索引，否则返回-1。

2、QMetaObject 类中与属性有关的成员函数有

- `int indexOfProperty(const char* name) const;` 返回属性 `name` 的索引，否则返回-1。
- `int propertyCount() const;` 返回属性的数量(包括从父类继承的属性)
- `int propertyOffset() const;`
返回父类中的属性数量，也就是说此返回值是此类第一个属性的索引位置。
- `QMetaProperty property(int index) const;`
返回索引号为 `index` 的属性的元数据，若不存在这样的属性，则返回空的 `QMetaProperty`
- `QMetaProperty userProperty() const;` 返回 `USER` 标志设置为 `true` 的属性的元数据。

示例：使用 QMetaObject 成员函数存取属性值

//源文件 m.h 的内容

`#ifndef M_H` //要使用元对象系统，需在头文件中定义类。

`#define M_H`

`#include <QObject>`

`#include <iostream>`

`using namespace std;`

`class Z:public QObject{ Q_OBJECT`

`public:`

`Q_PROPERTY(int b READ gb WRITE sb)`

```

        int gb() {return m_mb;}    void sb(int x) {m_mb=x;}
        int m_mb;                };
#endif // M_H

//源文件内容
#include "m.h"
#include<QMetaProperty>

int main(int argc, char *argv[]) {
    Z mz;
    const QMetaObject *p=mz.metaObject();
    QMetaProperty pe=p->property(p->indexOfProperty("b")); //获取属性 b 的元数据。
    cout<<pe.name()<<endl;    //输出属性的名称 b
    cout<<pe.typeName()<<endl; //输出属性 b 的类型 int
    pe.write(&mz,47); //把值 47 写入属性 b
    cout<<pe.read(&mz).value<int>()<<endl; //输出属性 b 的值 47。
    return 0;    }
}

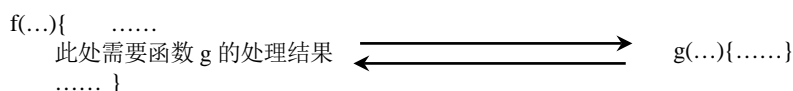
```

2.4 信号与槽

- 1、信号和槽是用于对象之间的通信的，这是 Qt 的核心。为此 Qt 引入了一些关键字，他们是 slots、signals、emit，这些都不是 C++ 关键字，是 Qt 特有的，这些关键字会被 Qt 的 moc 转换为标准的 C++ 语句。
- 2、Qt 的部件类中有一些已经定义好了的信号和槽，通常的作法是子类化部件类，然后添加自己的信号和槽。
- 3、因为信号和槽与函数相似，所以通常把信号称为信号函数，槽称为槽函数。

一、信号和槽原理

- 1、C++ 虽然是面向对象的语言，但程序的具体实现代码仍然是由函数来实现的，因此所谓的对象之间的通信，从程序设计语言语法角度来看就是函数调用的问题，只不过是某个对象的成员函数调用另一个对象的成员函数而已，本文从语法角度讲解信号和槽的原理，这样更容易理解信号和槽的实现原理。注：信号和槽其实是观察者模式的一种实现，有兴趣的读者可以参阅《设计模式》课程。
- 2、函数调用的几种形式



见上图，假设函数 f 需要 g 的处理结果，有以下几种处理方式

- ①、最简单的方式就是直接调用函数 g，但这种方式有一个明显的缺点，必须知道函数 g 的名称“g”以及函数 g 的参数类型。但是若 f 只需要 g 的处理结果就可以了，而 g

②、另一种方式就是回调函数，即在函数 f 中使用一个指向函数的指针去调用需要的函数，这样就可以调用任意名称的函数(只要函数类型与指针相同即可)，此时只要是完成了函数 g 功能的函数都可以作为函数 f 的结果被调用，这样就不会被函数名称所限制。

```
void (*p)(int i,int j);      //假设这是系统内部代码
f(...){.....; p(2,3); .....} //假设 f 也是系统内部源代码函数
void g(int,int){.....}      //假设 g 是由程序员实现的代码
void h(int,int){.....}      //原理同 g
p=h;                        //只需对指向函数的指针 p 赋予不同的函数地址,便能实现由系统回调不同的由程序员实现的代码,此处在函数 f 内部将调用函数 h。
```

其基本思想如下:

- 创建一个信号，其中创建信号需要一些规则。
- 当需要调用外部函数时，发送一个信号，
- 此时与该信号相关联的槽便会被调用，槽其实就是一个函数，当然要使函数成为槽是有一定规则的。槽与信号的关联需要由程序员来完成。
- 在 Qt 中，信号和槽都需要位于一个类之中。
- 示例(具体实现时 Qt 引入了一些新的关键字):

f(...){.....; 发送: x;} //信号既可由程序员发送,也可由系统发送,发送信号就相当于是一个函数调用(此处相当于调用函数 g),当接收信号的槽(即 g)执行完其自身的代码后,会继续沿着 x 之后的代码执行。

2、信号需符合以下规则

- 信号使用 `signals` 关键字声明,在其后面有一个冒号“:”,在其前面不能有 `public`、`private`、`protected` 访问控制符,信号默认是 `public` 的。
- 信号只需像函数那样声明即可,其中可以有参数,参数的主要作用是用于和槽的通信,这就像普通函数的参数传递规则一样。信号虽然像函数,但是对他的调用方式不一样,信号需要使用 `emit` 关键字发射。
- 信号只需声明,不能对其进行定义,信号是由 `moc` 自动生成的。
- 信号的返回值只能是 `void` 类型的。

3、声明槽需符合以下规则

- 声明槽需要使用 slots 关键字，在其后面有一个冒号“:”，且槽需使用 public、private、protected 访问控制符之一。
- 槽就是一个普通的函数，可以像使用普通函数一样进行使用，槽与普通函数的主要区别是，槽可以与信号关联。

4、发射信号需符合以下规则：

- 发射信号需要使用 emit 关键字，注意，在 emit 后面不需要冒号。
- emit 发射的信号使用的语法与调用普通函数相同，比如有一个信号为 void f(int)，则发送的语法为：emit f(3);
- 当信号被发射时，与其相关联的槽函数会被调用(注意：信号和槽需要使用 QObject::connect 函数进行关联之后，发射信号后才会调用相关联的槽函数)。
- 注意：因为信号位于类之中，因此发射信号的位置需要位于该类的成员函数中或该类能见到信号的标识符的位置。

5、信号和槽的关系

- 槽的参数类型需要与信号参数的类型相对应，
- 槽的参数不能多余信号的参数，因为若槽的参数更多，则多余的参数不能接收到信号传递过来的值，若在槽中使用了这些多余的无值的参数，就会产生错误。
- 若信号的参数多余槽的参数，则多余的参数将被忽略。
- 一个信号可以与多个槽关联，多个信号也可以与同一个槽关联，信号也可以关联到另一个信号上。
- 若一个信号关联到多个槽时，则发射信号时，槽函数按照关联的顺序依次执行。
- 若信号连接到另一个信号，则当第一个信号发射时，会立即发射第二个信号。

6、因 Qt 在其类库中预定义了很多信号和槽，因此在 Qt 中可以 仅使用 Qt 类库中预定义的信号和槽，也可以只使 Qt 类库中预定义的信号而使用自己的槽，也可以使用 Qt 类库中预定义的槽来响应自己定义的信号，当然，槽和信号也都可以使用自定义的。

示例：创建信号和槽

//头文件 m.h 的内容

```
#ifndef M_H
#define M_H
#include<QObject>
#include <iostream>
using namespace std;

class A:public QObject{ //信号和槽必须继承自 QObject 类
    Q_OBJECT           //必须添加该宏
    //public signals:void s1(int); //错误 signals 前不能有访问控制符。
    signals:void s();          //使用 signals 关键字声明信号，信号的语法与声明函数相同。
    signals:void s(int,int);   //正确，信号可以有参数，也可以重载。
        //void s2() {}        //错误，信号只需声明，不能定义。
        void s3();            //注意：这仍是声明的一个信号
    public:                  //信号声明结束后，重新使用访问控制符，表示以下声明的是成员函数。
        void g(){
```

```

        emit s3();    /*发射信号，其语法与调用普通函数相同，在信号与槽关联之前，发射的信号不会调用相应的槽函数。*/
    // emit: s3();    //错误，emit 后不能有冒号。
};

class B:public QObject{    Q_OBJECT
    public slots:           //使用 slots 关键字声明槽
        void x() {cout<<"X"<<endl;}    /*正确，槽就是一个普通函数，只是需要使用 slots 关键字，且能和信号相关联。*/
        //slots: void x() {}    //错误，声明槽时需要指定访问控制符。
    public:
        void g() { // emit s3();    //错误，在类 B 中对于标识符 s3 是不可见的
            }    };
#endif // M_H

//源文件的内容
#include "m.h"
int main(int argc, char *argv[]){    A ma;    B mb;
    QObject::connect(&ma, &A::s3, &mb, &B::x);    //关联信号和槽，详见后文
    ma.g();    //调用对象 mb 的成员函数 x 输出 X，可见对象 ma 和 mb 之间实现了通信。
    return 0; }

```

三、信号和槽的关联(连接)

1、信号和槽使用 QObject 类中的成员函数 connect 进行关联，该函数有多个重载版本，如下所示。

2、形式 1: static QMetaObject::Connection connect(

```

const QObject *sender,    const char *signal,
const QObject *receiver,    const char *method,
Qt::ConnectionType type = Qt::AutoConnection)

```

- 示例:

```

class A:public QObject {Q_OBJECT    singals: void s(int i);};
class B:public QObject{Q_OBJECT    public slots: void x(int i){};
A ma;    B mb;
QObject::connect (&ma, SIGNAL( s(int) ), &mb, SLOT(x(int) ));

```

信号的指定必须使用宏 SIGNAL()和槽必须使用宏 SLOT()，这两个宏能把括号中的内容转换为与形参相对应的 const char*形式。在指定函数时，只能指定函数参数的类型，不能有参数名，也不能指定函数的返回类型。比如 SLOT(x(int i))，是错误的，因为指定了参数名 i，正确形式为 SLOT(x(int))

各参数意义如下

- sender: 表示需要发射信号的对象。
- signal: 表示需要发射的信号，该参数必须使用 SIGNAL()宏。
- receiver: 表示接收信号的对象。
- method: 表示与信号相关联的槽函数，这个参数也可以是信号，从而实现信号与信号的关联。该参数若是槽，需使用 SLOT()宏，若是信号需使用 SIGNAL 宏。

- 返回值的类型为 `QMetaObject::Connection`，如果成功将信号连接到槽，则返回连接的句柄，否则，连接句柄无效，可通过将句柄转换为 `bool` 来检查该句柄是否有效。该返回值可用于 `QObject::disconnect()` 函数的参数，以断开该信号和槽的关联。至于该类型不必深入研究。
- **type:** 用于指明信号和槽的关联方式，它决定了信号是立即传送到一个槽还是在稍后时间排队等待传送。关联方式使用枚举 `Qt::ConnectionType` 进行描述，下表为其取值及意义

Qt::ConnectionType 的取值(type 参数的取值)		
枚举	值	说明
<code>Qt::AutoConnection</code>	0	(自动关联, 默认值)。若接收者驻留在发射信号的线程中(即信号和槽在同一线程中), 则使用 <code>Qt::DirectConnection</code> , 否则, 使用 <code>Qt::QueuedConnection</code> 。当信号发射时确定使用哪种关联类型。
<code>Qt::DirectConnection</code>	1	直接关联。当信号发射后, 立即调用槽。在槽执行完之后, 才会执行发射信号之后的代码(即 <code>emit</code> 关键字之后的代码)。该槽在信号线程中执行。
<code>Qt::QueuedConnection</code>	2	队列关联。当控制权返回到接收者线程的事件循环后, 槽才会被调用, 也就是说 <code>emit</code> 关键字后面的代码将立即执行, 槽将在稍后执行, 该槽在接收者的线程中执行。
<code>Qt::BlockingQueuedConnection</code>	3	阻塞队列关联。和 <code>Qt::QueuedConnection</code> 一样, 只是信号线程会一直阻塞, 直到槽返回。如果接收者驻留在信号线程中, 则不能使用此连接, 否则应用程序将会死锁。
<code>Qt::UniqueConnection</code>	0x80	唯一关联。这是一个标志, 可使用按位或与上述任何连接类型组合。当设置 <code>Qt::UniqueConnection</code> 时, 则只有在重复的情况下才会进行连接, 如果已经存在重复连接(即, 相同的信号指向同一对象上的完全相同的槽), 则连接将失败, 此时将返回无效的 <code>QMetaObject::Connection</code>

3、形式 2: `QMetaObject::Connection connect(const QObject *sender, const char *signal, const char *method, Qt::ConnectionType type = Qt::AutoConnection) const`

- 各参数的意义见形式 1
- **注意:** 此函数是非静态的, 它是 `QObject` 的成员函数。
- 此函数是形式 1 的简化版本, 相当于 `connect(sender, signal, this, method, type)`
- 示例: 假设 `void s(int i)` 是类 `A` 中定义的信号, `void x(int i)` 是类 `B` 中定义的槽, 则 `A ma; B mb; mb.connect(&ma, SIGNAL(s(int)), SLOT(x(int)));` 以上注意调用 `connect` 的方式。

4、形式 3: `static QMetaObject::Connection connect(const QObject *sender, PointerToMemberFunction signal, const QObject *receiver, PointerToMemberFunction method, Qt::ConnectionType type = Qt::AutoConnection)`

- 各参数的意义见形式 1
- 这是 Qt5 中加入的新函数。

- 示例：假设 `void s(int i)` 是类 A 中定义的信号，`void x(int i)` 是类 B 中定义的槽，则
`A ma; B mb; QObject::connect(&ma, &A::s, &mb, &B::x);`

注意：该函数对信号和槽函数的指定方式不是使用的宏。

- 该形式的函数其实是一个模板函数，其完整原型类似如下：

```
template<typename PointerToMemberFunction>
static QMetaObject::Connection connect(.....)
```

5、形式 4: `static QMetaObject::Connection connect(const QObject *sender, PointerToMemberFunction signal, Functor functor)`

- 各参数的意义见形式 1
- 该函数的第三个参数支持仿函数、全局函数、静态函数、Lambda 表达式，但是不能是类的非静态成员函数
- 示例：假设 `void s(int i)` 是类 A 中定义的信号，`void x(int i)` 是类 B 中定义的静态槽，则
`A ma; QObject::connect(&ma, &A::s, &B::x);`
- 该形式的函数其实是一个模板函数，其完整原型类似如下：

```
template<typename PointerToMemberFunction, typename Functor>
static QMetaObject::Connection connect(.....)
```

6、形式 5: `static QMetaObject::Connection QObject::connect (const QObject * sender, const QMetaMethod& signal, const QObject * receiver, const QMetaMethod& method, Qt::ConnectionType type = Qt::AutoConnection)`

- 此函数的工作方式与形式 1 相同，只是它使用 `QMetaMethod` 指定信号和槽。

7、形式 3 与形式 1 的区别

- ①、形式 1 的 `SIGNAL` 和 `SLOT` 宏实际是把该宏的参数转换为字符串，当信号和槽相关联时，使用的是字符串进行匹配，因此，信号和槽的参数类型的名字必须在字符串意义上相同，所以信号和槽无法使用兼容类型的参数，也因此也不能使用 `typedef` 或 `namespace` 的类型，虽然他们的实际类型相同，但由于字符串名字不同，从而无法使用形式 1。
- ②、形式 3 的信号和槽函数的参数类型不需完全一致，可以进行隐式转换。形式 3 还支持 `typedef` 和命名空间。
- ③、形式 3 以指针的形式指定信号和槽函数，不需再使用 `SIGNAL()` 和 `SLOT` 宏。
- ④、形式 3 的槽函数可以不使用 `slots` 关键字声明，任意的成员函数都可以是槽函数。形式 1 的槽函数必须使用 `slots` 修饰。
- ⑤、形式 1 的槽函数不受 `private` 的限制，也就是说即使槽是 `private` 的，仍可通过信号调用该槽函数，而形式 3 则在使用 `connect` 时就会发生错误。
- ⑥、当信号或槽函数有重载的形式时，使用形式 3 可能会产生二义性错误，此时可使用函数指针的形式指定信号或槽函数，或者使用形式 1，比如

```
class A:public QObject {Q_OBJECT    singals: void s(int i);    };
class B:public QObject{Q_OBJECT    public slots: void x(){}    void x(int i){}};
A ma; B mb;
```

```
QObject::connect(&ma, &A::s, &mb, &B::x); //二义性错误。
```

可使用如下方式解决(对于信号类似)

```
QObject::connect(&ma, &A::s, &mb, static_cast<void (B::*)(int)> (&B::x));
```

示例：信号和槽的关联

//头文件 m.h 的内容

```
#ifndef M_H
#define M_H
#include<QObject>
#include <iostream>
using namespace std;

class A:public QObject{      Q_OBJECT
    signals:void s();    void s(int,int);    void s1();    void s2(int); };
class B:public QObject{      Q_OBJECT
    public slots:
        void x() {cout<<"x"<<endl;}          void y(int i,int j) {cout<<"y"<<i<<j<<endl;}
        void z(int) {cout<<"zi"<<endl;}        void z1(float) {cout<<"z1f"<<endl;}
    public:
        void z2() { //注意，该函数未使用 slots 声明。
                    cout<<"z2"<<endl; }
    private slots:    //私有槽
        void z3() {    cout<<"z3"<<endl;    }    };
#endif // M_H
```

//源文件的内容

```
#include "m.h"
int main(int argc, char *argv[]){    A ma;    B mb;
    QObject::connect(&ma, SIGNAL(s()), &mb, SLOT(x())); //形式 1
    emit ma.s();    //输出 x
    QObject::connect(&ma, &A::s1, &mb, &B::x); //形式 3
    emit ma.s1();    //输出 x
    // QObject::connect(&ma, &A::s1, &mb, &B::y); //错误，槽参数的数量多余信号参数的数量
//类型转换
    //关联失败，形式 1 不支持类型转换
    //QObject::connect(&ma, SIGNAL(s2(int)), &mb, SLOT(z1(float)));
    typedef int T;
    //关联失败，对于形式 1，类型 T 和 int 在字符串形式上并不相同。
    //QObject::connect(&ma, SIGNAL(s2(T)), &mb, SLOT(z(int)));
    QObject::connect(&ma, &A::s2, &mb, &B::z1); //正确，形式 3 支持隐式类型转换
    emit ma.s2(2); //输出 z1f
//槽函数与 slots
    //关联失败，形式 1 的槽必须使用 slots 声明。
    //QObject::connect(&ma, SIGNAL(s1()), &mb, SLOT(z2()));
    QObject::connect(&ma, &A::s1, &mb, &B::z2); //正确，形式 3 的槽不需使用 slots 声明。
    emit ma.s1();    /*输出 x, z2, 注意：在之前 s1 和 x 的关联并未断开，此时信号 s1 同时与槽 x 和
                        z2 关联。*/
//访问控制符
    QObject::connect(&ma, SIGNAL(s1()), &mb, SLOT(z3())); //正确，形式 1 的槽不受访问控制符限制。
    emit ma.s1();    //输出 x, z2, z3, 因为此时 s1 与多个槽相关联。
    //QObject::connect(&ma, &A::s1, &mb, &B::z3); //错误，z3 是私有的，形式 3 会受访问控制符的限制。
```



```
//函数重载，形式2和形式4的使用见正文
return 0;    }
```

四、断开信号和槽的关联

1、信号和槽使用 `QObject` 类中的成员函数 `disconnect` 函数断开其关联，该函数有多个重载版本，如下所示。

2、形式 1: `static bool QObject::disconnect(`

```
const QObject *sender,      const char *signal,
const QObject *receiver,    const char *method)
```

- 断开 `sender` 对象中的信号 `signal` 与 `receiver` 对象中槽函数 `method` 的关联。
- 注意：此函数指定 `signal` 和 `method` 时需要使用 `SIGNAL` 和 `SLOT` 宏。
- 如果连接成功断开，则返回 `true`；否则返回 `false`。
- 当所涉及的任何对象被销毁时，信号槽连接被移除。
- 0 可以用作通配符，分别表示“任意信号”、“任何接收对象”或“接收对象中的任何插槽”。
- `sender` 永远不会是 0。
- 如果 `signal` 为 0，则将 `receiver` 对象中的槽函数 `method` 与所有信号断开。否则，则只与指定的信号断开，此方法可断开槽与所有信号的关联，比如

类 A 中有信号 `void s()` 和 `void s1()`；类 B 中有槽 `void x()`；

A ma; B mb;

然后把 `ma` 中的信号 `s` 和 `s1` 与 `mb` 中的槽 `x` 相关联，

若 `signal` 为 0，则 `mb` 中的 `x` 会断开与 `s` 和 `s1` 的关联。

- 如果 `receiver` 为 0，此时 `method` 也必须为 0
- 如果 `method` 为 0，则断开与连接到 `receiver` 的任何连接。否则，只有命名 `method` 的槽将被断开连接，而所有其他槽都将被单独保留。如果没有 `receiver`，则 `method` 必须为 0，因此不能断开所有对象上指定的槽。比如

类 A 中有信号 `void s()`；类 B 中有槽 `void x()`；类 C 中有槽 `void y()`；

A ma; B mb, mb1; C mc;

然后把信号 `s` 与对象 `mb` 中的槽 `x`、对象 `mb1` 中的槽 `x`、对象 `mc` 中的槽 `y` 相关联，

若 `receiver` 被指定为 `mb`，`method` 为 0，则 `mb` 中的 `x`、`mb1` 中的 `x` 会与信号 `s` 断开，但 `mc` 中的 `y` 不会与信号 `s` 断开，

若 `receiver` 被指定为 `mb`，`method` 为 `x`，则 `mb` 中的 `x` 会与信号 `s` 断开，`mb1` 中的 `x`、`mc` 中的 `y` 不会与 `s` 断开。

- 除此之外，还有以下几种常用的用法

`disconnect(&ma, 0, 0, 0);` 断开与对象 `ma` 中的所有信号相关联的所有槽。

`disconnect(&ma, SIGNAL(s()), 0, 0);` 断开与对象 `ma` 中的信号 `s` 相关联的所有槽。

`disconnect (&ma, 0, &mb, 0);` 断开 `ma` 中的所有信号与 `mb` 中的所有槽的关联

3、形式 2: `static bool QObject::disconnect (const QMetaObject::Connection &connection)`

该函数断开使用 connect 函数返回的信号和槽的关联，若操作失败则返回 false。

4、形式 3: static bool QObject::disconnect(

```
const QObject *sender,          PointerToMemberFunction signal,
const QObject *receiver,        PointerToMemberFunction method)
```

- 此方法与形式 1 相同，只是指定函数的方式是使用函数指针。
- 注意：该函数不能断开信号连接到一般函数或 Lambda 表达式之间的关联，此时需要使用形式 2 来断开这种关联。

5、形式 4: static bool QObject::disconnect(

```
const QObject *sender,          const QMetaMethod &signal,
const QObject *receiver,        const QMetaMethod &method)
```

该函数与形式 1 相同，只是指定函数的方式是使用 QMetaMethod 类。

6、形式 5: bool QObject::disconnect(const char *signal = Q_NULLPTR,

```
const QObject *receiver = Q_NULLPTR,
const char *method = Q_NULLPTR) const
```

注意：该函数是非静态的，该函数是形式 1 的重载形式。

7、形式 6: bool QObject::disconnect(const QObject *receiver,

```
const char *method = Q_NULLPTR) const
```

注意：该函数是非静态的，该函数是形式 1 的重载形式。

- 8、注意：若关联信号和槽时使用的是函数指针形式，则在断开信号和槽时，最好使用相对应的函数指针形式的 disconnect 版本，以避免产生无法断开关联的情形。

五、signals、slots、emit 关键字原型

- 1、在以上关键字上右击然后按下 F2，在 qobjectdefs.h 文件中可以看到这些关键字的原型

- 2、signals 关键字：最终被#define 替换为一个访问控制符，其简化后的语法为

```
#define signals public
```

- 3、slots 关键字：最终被#define 替换为一个空宏，即简化后的语法为：#define slots

- 4、emit 关键字：同样被#define 替换为一个空宏，即简化后为：#define emit

- 5、由以上各关键字原型可见，使用 emit 发射信号，其实就是一个简单的函数调用。

2.5 对象树与生命期

- 1、为什么要使用对象树：GUI 程序通常是存在父子关系的，比如一个对话框之中含有按钮、列表等部件，按钮、列表、对话框等部件其实就是一个类的对象(注意是类的对象，而非类)，很明显这些对象之间是存在父子关系的，因此一个 GUI 程序通常会由一个父对象维护着一系列的子对象列表，这样更方便对部件的管理，比如当按下 tab 键时，父对象会依据子对象列表令各子对象依次获得焦点。当关闭对话框时，父对象依据子对象列表，找到

每个子对象，然后删除它们。在 Qt 中，对对象的管理，使用的是树形结构，也就是对象树。

2、子对象和父对象：本小节的父/子对象是相对于由对象组成的树形结构而言了，父节点对象被称为父对象，子节点对象被称为子对象。注意：子对象并不是指类中的对象成员。

一、组合模式与对象树

- 1、组合模式指的是把类的对象组织成树形结构，这种树形结构也称为对象树，Qt 使用对象树来管理 QObject 及其子类的对象。注意：这里是指的类的对象而不是类。把类组织成树形结构只需使用简单的继承机制便可实现。
- 2、使用组合模式的主要作用是可以通过根节点对象间接调用子节点中的虚函数，从而可以间接的对子节点对象进行操作。
- 3、组合模式的基本思想是使用父类类型的指针指向子类对象，并把这个指针存储在一个数组中(使用容器更方便)，然后每创建一个类对象就向这个容器中添加一个指向该对象的指针。下面的示例为核心代码

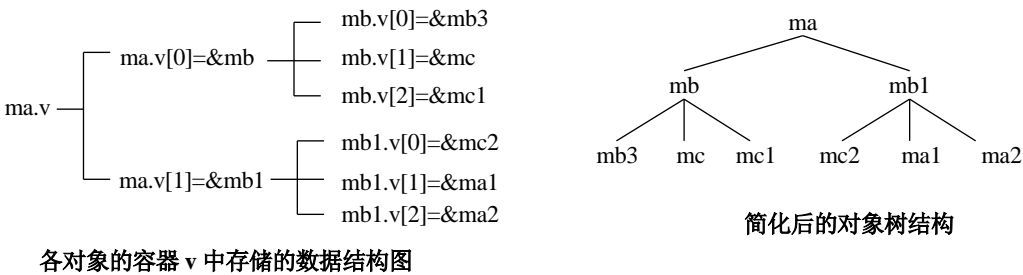
示例：组合模式核心代码

```
#include <iostream>
#include<vector>      //使用容器。
using namespace std;

class A {public: //顶级父类。
    vector<A*> v; /*创建一个存储父类类型指针的容器，此步也可使用形如A* m[11]的数组代替，但数组不能自动扩容。*/
    void add(A* ma) { v.push_back(ma);} }; //add函数的主要作用是将ma添加到容器v中。
class B :public A {public: //需要继承自类A
    void add(A* ma) { v.push_back(ma); } };
class C :public A {public: };//需要继承自类A，该类无add函数，也就是说该类的对象不能有子对象。

int main(){    A ma, ma1, ma2, ma3;    B mb, mb1, mb2, mb3;    C mc, mc1, mc2, mc3;
    //创建对象树。
    ma.add(&mb);        ma.add(&mb1);
    mb.add(&mb3);        mb.add(&mc);        mb.add(&mc1);
    mb1.add(&mc2);        mb1.add(&ma1);        mb1.add(&ma2); }
```

程序的结构如下图



示例：打印出组合模式中的各对象的名称

```
#include <iostream>
#include <string>
#include<vector>
using namespace std;

class A {public:
    string name;//用于存储创建的对象名称
    vector<A*> v; //创建一个存储父类类型指针的容器。
    A() {}
    A(string n){name=n;}
    void add(A* ma) { v.push_back(ma);} //将ma添加到容器v中。
    virtual string g(){return name;} //虚函数，用于获取对象名。
    void f() { //用于显示当前对象的容器v中存储的内容。
        if(!v.empty()) //若v不为空，则执行以下语句。
            {cout<<name<<"="; //输出当前对象的名称。
            for (vector<int>::size_type i = 0; i!=v.size(); i++) //遍历容器v。
                {cout<<v[i]->g()<<" ";} //输出容器v中存储的对象的名称，注意g是虚函数。
            cout<<endl;} } //f结束
    virtual void pt() { //该函数会被递归调用，用以输出整个对象树中对象的名称。
        f();
        for (vector<int>::size_type i = 0; i!=v.size(); i++)
            v[i]->pt(); } //注意pt是虚函数，假如v[i]类型为其子类型B时，则会调用B::pt()
    }; //类A结束
class B :public A {public: //需要继承自类A，代码与类A类似
    string name;
    B(string n){name=n;}
    void add(A* ma) { v.push_back(ma); }
    string g(){return name;}
    void f() {
        if(!v.empty()) {cout<<name<<"=";
            for (vector<int>::size_type i = 0; i!=v.size(); i++) {cout<<v[i]->g()<<" ";}
            cout<<endl;} } //f结束
    void pt() { f();
        for (vector<int>::size_type i = 0; i!=v.size(); i++) v[i]->pt(); }
    }; //类B结束
class C :public A {public: //需要继承自类A，该类无add函数，也就是说该类的对象不能有子对象。
    string name;
    C(string n){name=n;}
    void f() {
        if(!v.empty()) {cout<<name<<"=";
            for (vector<int>::size_type i = 0; i!=v.size(); i++) {cout<<v[i]->g()<<" ";}
            cout<<endl;} } //f结束
    string g(){return name;}
    void pt() { f();
        for (vector<int>::size_type i = 0; i!=v.size(); i++) v[i]->pt();} }; //类C结束

int main()
{ //创建对象时传递该对象的名称以便存储。
    A ma("ma"), ma1("ma1"), ma2("ma2"), ma3("ma3"), ma4("ma4");
    B mb("mb"), mb1("mb1"), mb2("mb2"), mb3("mb3"), mb4("mb4");
    C mc("mc"), mc1("mc1"), mc2("mc2"), mc3("mc3"), mc4("mc4");
```

```

ma.add(&mb);    //ma.v[0]=&mb;
mb.add(&mb1);   //mb.v[0]=&mb1;
mb.add(&mb2);   //mb.v[1]=&mb2;
ma.add(&mb1);   //ma.v[1]=&mb1;
mb1.add(&mb3);  //mb1.v[0]=&mb3;
mb1.add(&mb4);  //mb1.v[1]=&mb4;
mb2.add(&mc1);  //mb2.v[0]=&mc1;
mb2.add(&mc2);  //mb2.v[1]=&mc2;
mb2.add(&ma1);  //mb2.v[2]=&ma1;
cout<<"各对象拥有的子对象"<<endl;
ma.f();        mb.f();        mb1.f();        mb2.f();
cout<<endl<<"整个对象中结构"<<endl;
ma.pt();       }

```

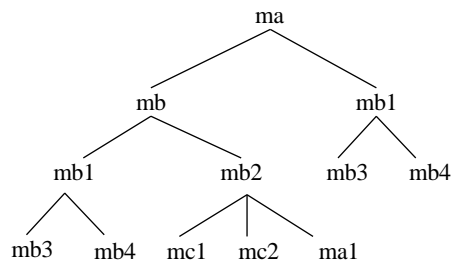
运行结果及对象的组织结构如下图

```

各对象拥有的子对象
ma=mb, mb1,
mb=mb1, mb2,
mb1=mb3, mb4,
mb2=mc1, mc2, ma1,

整个对象中结构
ma=mb, mb1,
mb=mb1, mb2,
mb1=mb3, mb4,
mb2=mc1, mc2, ma1,
mb1=mb3, mb4,
请按任意键继续. . .

```



示例：使用父节点对象删除子结构对象

```

#include <iostream>
#include <string>
#include<vector>
using namespace std;

class A {public:
    vector<A*> v; //容器
    string name; //存储对象名
    virtual string g() {return name;} //虚函数，用于获取子对象的名称。
    A() {}
    A(string n) {name=n;}
    A(A* ma, string n) {name=n; ma->v.push_back(this); }
    virtual ~A() { //虚析构函数，原理请参阅相关C++语法
        cout << "~A" << endl; //❶，输出的字符用于测试
        if(!v.empty()) { //❷
            cout << "XXXXX" << endl; //用于测试
            for (vector<int>::size_type i = 0; i != v.size(); i++) //❸
            { cout << "object = " << v[i]->g() << endl; //❹输出的字符用于测试
              delete v[i]; //❺
              //v[i]->~A();
            }
        }
    }
};

```

```

        cout<<"YYYY"<<endl;           //⑥输出的字符用于测试
    } }
    cout << "#####" << endl;    //⑦输出的字符用于测试
};

class B :public A {public:    string name;        string g() {return name;}
    B() {}        B(string n) {name=n;}        B(A* ma, string n):A(ma,n) {    name=n;    }
    ~B() { cout << "~B" << ", "; }           //⑧输出的字符用于测试
};

class C :public A {public:    string name;        string g() {return name;}
    C(A* ma, string n):A(ma,n) {    name=n;}
    ~C() { cout << "~C" << ", "; }           //⑨输出的字符用于测试
};

class D :public B {public:    string name;        string g() {return name;}
    D(A* ma, string n):B(ma,n) {    name=n;}
    ~D() { cout << "~D" << ", "; }           //⑩输出的字符用于测试
};

int main() {
    A ma("ma");    //B mb("mb");
    B *pb=new B(&ma, "pb");    B *pb1=new B(&ma, "pb1");
    B *pb2=new B(pb, "pb2");    C *pc1=new C(pb, "pc1");
    A* pa1=new A(pb1, "pa1");    B* pb3=new B(pb1, "pb3");    C* pc2=new C(pb1, "pc2");
    D* pd1=new D(pc1, "pd1");    D* pd2=new D(pc1, "pd2");    }
}

```

运行结果及对象树结构见下图

```

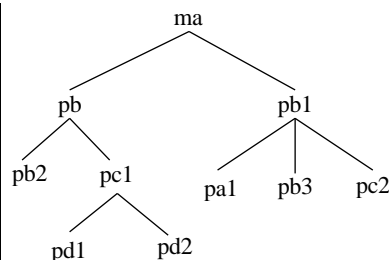
A
XXXXX
object = pb
B, A
XXXXX
object = pb2
B, A
#####
YYYY
object = pc1
C, A
XXXXX
object = pd1
D, B, A
#####
YYYY
object = pd2
D, B, A
#####
YYYY
#####
YYYY
#####
YYYY
#####
YYYY

```

```

object = pb1
B, A
XXXXX
object = pa1
A
#####
YYYY
object = pb3
B, A
#####
YYYY
object = pc2
C, A
#####
YYYY
#####
YYYY
#####
YYYY
#####
请按任意键继续.

```



程序运行分析:

1、程序结束时，局部对象 ma 生命期的结束，执行类 A 的析构函数，输出~A，

- 2、此时 `ma.v` 不为空，因此输出 `XXXXXX`，第 ③ 步为真 `i=0`，执行第 ④ 步，此时 `ma.v[0]=pb`，因此调用类 B 的虚函数 `g()`，输出 “object=pb”，接着执行第 ⑤ 步。
- 3、第 ⑤ 步，`delete v[i] = delete ma.v[0] = delete pb`，由虚析构函数性质可知，此时会先调用类 B 的析构函数，然后再次调用类 A 的析构函数，因此首先输出 “~B,”，接着再次进入类 A 的析构函数，输出 “~A”，接着执行第 ② 步。
- 4、此时 `pb->v[0]=pb2` 不为空(注意，此时是 `pb`)，重复执行以上步骤，输出 “XXXXXX,object=pb2,~B, ~A”，然后再次执行第 ② 步。
- 5、此时 `pb2->v` 为空，因此跳过整个 `if` 结构，输出后面的 “#####”，此时整个树形结构 `pb2` 分支执行结束，返回到第 ⑤ 步执行其后的语句，输出 “YYYY”。
- 6、然后再次执行第 ③ 步判断循环语句，此时 `i=1` 仍为真，同理，此时 `pb->v[1]=pc1`(注意，此时仍是判断的 `pb.v`)，重复上述步骤，删除该树形结构下的 `pd1` 和 `pd2`，至此 `ma` 的 `pb` 分支执行结束。
- 7、`ma` 的 `pb1` 分支，执行过程与 `pb` 分支相同。

二、QObject 类、对象树、生命期

- 1、为方便讲解，本文把由 `QObject` 及其子类创建的对象称为 `QObject`、`QObject` 对象或 `Qt` 对象。在 `Qt` 中，`QObject` 对象通常就是指的 `Qt` 部件。
- 2、`QObject` 类是所有 `Qt` 对象的基类，是 `Qt` 对象模型的核心，所有 `Qt` 部件都继承自 `QObject`。
- 3、`QObject` 及其派生类的单形构造函数应声明为 `explicit`，以避免发生隐式类型转换。
- 4、`QObject` 类既没有复制构造函数也没有赋值操作符函数(实际上它们被声明为私有的)，因此无法通过值传递的方式向函数传递一个 `QObject` 对象。
- 5、`Qt` 库中的 `QObject` 对象是以树状结构组织自己的，当创建一个 `QObject` 对象时，可以为其设置父对象，新创建的对象会被加入到父对象的子对象列表中(可通过 `QObject::children()` 函数查询)，因为 `Qt` 的部件类，都是以 `QObject` 为基类，因此，`Qt` 的所有部件类都具有对象树的特性。
- 6、对象树的组织规则：
 - ①、每一个 `QObject` 对象只能有一个父 `QObject` 对象，但可以有任意数量的子 `QObject` 对象。比如

```
A ma; B mb; C mc;
ma.setParent(&mb);    //将对象 ma 添加到 mb 的子对象列表中，
ma.setParent(&mc);    //该语句会把 ma 从 mb 的子对象列表中移出，并将其添加到
                        mc 的子对象列表中。setParent 函数见后文。
```
 - ②、`QObject` 对象会把指向各个子对象地址的指针放在 `QObjectList` 之中。`QObjectList` 是 `QList<QObject*>` 的别名，`QList` 是 `Qt` 的一个列表容器。
- 7、对象删除规则(注意：以下规则并非 C++语法规则，而是 `Qt` 的对象树规则):
 - ①、基本规则：父对象会自动删除子对象。父对象拥有对象的所有权，在父对象被删除时会在析构函数中自动删除其子对象。
 - ②、手动删除子对象：当手动删除子对象时，会把该子对象从父对象的列表中移除，以避免父对象被删除时该子对象被再次删除。总之 `QObject` 对象不会被删除两次。
 - ③、当一个对象被删除时，会发送一个 `destroyed()` 信号，可以捕捉该信号以避免对 `QObject`

对象的悬垂引用。

8、对象创建规则

- ①、子对象通常应创建在堆中(使用 `new` 创建), 此时就不再需要使用 `delete` 将其删除了, 当父对象被删除时, 会自动删除该子对象。
- ②、对于 Qt 程序, 父对象通常创建在栈上, 不应创建在堆上(使用 `new` 创建)
- ③、子对象不应创建在栈中, 因为若父对象比子对象更早的结束生命期(即父对象创建于子对象之后), 则子对象会被删除两次, 第一次发生在父对象生命期结束时, 由 Qt 对象树的规则, 使用父对象删除子对象, 第二次发生在子对象生命期结束时, 由 C++ 规则删除子对象。这种错误可使用先创建父对象后创建子对象的方法解决, 依据 C++ 规则, 子对象会先被删除, 由 Qt 对象树规则知, 此时子对象会从父对象的列表中移除, 当父对象结束生命期时, 就不会再次删除子对象了。

9、其他规则: 应确保每一个 `QObject` 对象在 `QApplication` 之后创建, 在 `QApplication` 销毁之前销毁, 因此 `QObject` 对象不能是 `static` 存储类型的, 因为 `static` 对象将在 `main()` 返回之后才被销毁, 其销毁时间太迟了。

10、名象的名称: 可以为每个对象设置一个对象名称, 其主要作用是方便对对象树进行查询和管理。对象名称和对象是不同的, 比如 `A ma`; 其中 `ma` 是对象, 若为 `ma` 设置一个名称为 “SSS”, 则对象 `ma` 的对象名称为 “SSS”

11、设置父对象的方法

- ①、创建对象时, 在构造函数中指定父对象, `QObject` 类及其子类都有一个形如 `QObject *parent=0` 的形参的构造函数, 因此我们可以在创建对象时在构造函数中直接指定父对象。
- ②、使用 `void QObject::setParent(QObject *parent)` 函数为该对象设置父对象。

12、设置对象名称:

对象名称由 `QObject` 的 `objectName` 属性指定(默认值为空字符串), 该属性的读取函数分别如下所示 (注: 对象的类名可通过 `QMetaObject::className()` 查询。)

```
QString objectName() const           //读取该对象名称
void setObjectName(const QString &name); //设置该对象的名称为 name
```

示例: Qt 的对象树与对象的删除

```
#include<QObject>
#include <iostream>
using namespace std;

class A:public QObject{public:           //子类化 QObject
    A() {}           A(QObject *p):QObject(p) {}
    ~A() {cout<<objectName().toStdString()<<"=~A"<<endl;} };

class B:public QObject{public:           //子类化 QObject
    B() {}           B(QObject *p):QObject(p) {}
    ~B() {cout<<objectName().toStdString()<<"=~B"<<endl;} };

int main(int argc, char *argv[]) {
    A ma;           //父对象通常创建在栈上
```



```

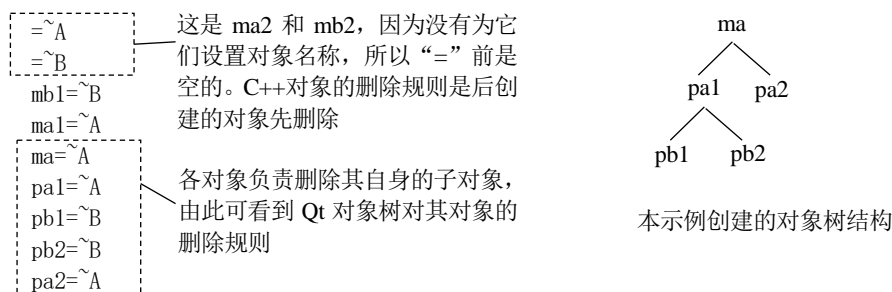
A *pa1=new A(&ma);    //在堆上指定父对象
A *pa2=new A(&ma);
B *pb1=new B(pa1);
B *pb2=new B(pa1);
ma.setObjectName("ma");    //为各对象设置对象名
pa1->setObjectName("pa1");    pa2->setObjectName("pa2");
pb1->setObjectName("pb1");    pb2->setObjectName("pb2");
A ma1;        B mb1;
mb1.setParent(&ma1);    //在栈上把 ma1 指定为 mb1 的父对象，此处父对象创建于子对象之前。
ma1.setObjectName("ma1");    mb1.setObjectName("mb1");

B mb2;        A ma2;
//mb2.setParent(&ma2);    /*错误，在栈上指定父对象时，父对象应创建于子对象之前，此处会导致
                           子对象 mb2 被删除两次。*/

return 0; }

```

运行结果如下图所示



13、查询对象树的信息，可使用以下 QObject 类中的成员函数

①、QObject* parent() const 返回一个指向父对象的指针。

②、const QObjectList& children() const

- 作用：返回指向父对象中全部子对象的指针列表，新添加的子对象位于列表的最后(某些特定操作可改变该顺序，例如提升或降低 QWidget 子对象)。其中 QObjectList 类型如下

```
typedef QList<QObject*> QObjectList;    //QList 是 Qt 的列表容器。
```

③、QList<T> findChildren (const QString& name=QString(),

```
Qt::FindChildOptions options=Qt::FindChildrenRecursively) const
```

- 作用：返回能转换为类型 T，且名称为 name 的所有子对象，若没有此对象，则返回空列表，若 name 为默认值，则会匹配所有对象的名称。该函数是按递归方式执行的。
- 该函数与下面介绍的 findChild 的主要用途是可以通过父对象(或父部件)获取指向子对象(或子部件)的指针。
- name 参数：该参数是由 QObject::setObjectName 函数设置的名称。
- options 参数：该参数用于指定查询子对象的方式，该参数需要指定一个

FindChildOption 类型的枚举值，FindChildOptions(注意，后面多了一个 s)是由 QFlags<FindChildOption>使用 typedef 重命名后的结果。该参数可取以下两个 FindChildOption 类型的枚举值

- Qt::FindDirectChildrenOnly: 表示查找该对象的直接子对象。

- Qt::FindChildrenRecursively: 表示按递归方式查询该对象的所有子对象。

- 注：经测试，当该对象没有名称为 name 的子对象时，该函数才返回空。

④、T findChild(const QString& name=QString(),

Qt::FindChildOptions options=Qt::FindChildrenRecursively) const

该函数的作用与 findChildren 相同，但是该函数只能返回单个的子对象，

⑤、void dumpObjectTree()

该函数可在调试模式(debug)下输出某个对象的整个对象树结构。该函数在发布模式(release)下不会执行任何操作。

示例：查询对象树

```
#include<QDebug>    //需使用 qDebug() 函数，该函数的用法类似于 C++ 的 cout
#include<QObject>
```

```
class A:public QObject{ public: A() {} A(QObject *p):QObject(p) {} void f() {qDebug()<<"AF";}};
class B:public QObject{ public:B() {} B(QObject *p):QObject(p) {} void f() {qDebug()<<"BF";}};
class C:public QObject{public: int c;
                        C() {} C(QObject *p):QObject(p) {} void f() {qDebug()<<"CF";}};
```

```
int main(int argc, char *argv[]) {
    A ma;                A *pa1=new A(&ma);                A *pa2=new A(&ma);
    B *pb1=new B(pa1);    B *pb2=new B(pa1);
    C *pc1=new C(pb1);    C *pc2=new C(pb1);
    ma.setObjectName("ma");    pa1->setName("pa1");    pa2->setName("pa2");
    pb1->setName("pb1");    pb2->setName("pb2");
    pc1->setName("pc1");    pc2->setName("pc2");
    pc2->c=2;                pc1->c=1;
```

```
    QObjectList st= ma.children();
    qDebug()<<"ma="<<ma.children();
    //以上输出: ma= (QObject(0x82d638, name = "pa1"), QObject(0x82d478, name = "pa2"))
    qDebug()<<"pb1="<<pb1->children();
    //以上输出: pb1= (QObject(0x840f38, name = "pc1"), QObject(0x840bf0, name = "pc2"))
```

```
    qDebug()<<"\n##### dumpObjectTree #####";
    ma.dumpObjectTree();                //输出见下图
    qDebug()<<"#####";
    pb1->dumpObjectTree();                //输出见下图
```

```
    qDebug()<<"\n##### findChild #####";
    C* p1=ma.findChild<C*>("pc1"); //通过父对象 ma 获取指向子对象 pc1 的指针。
    p1->f(); //输出 CF
    qDebug()<<p1->c; //输出 1
    C* p2=ma.findChild<C*>("pc2",Qt::FindDirectChildrenOnly); /*获取 ma 的直接子对象中名称为 pc2
    的对象的指针，因为 pc2 不是 ma 的直接子类，所以该处返回 NULL。*/
    //qDebug()<<p2->c; //错误，此时 p2 指向的是 NULL
```

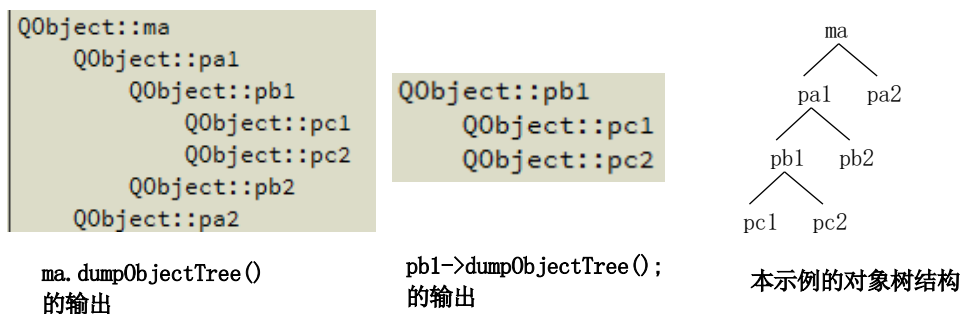
```

QDebug() << "\n##### findChildren #####";
QList<C*> s=ma.findChildren<C*>("pc1"); //Qt::FindDirectChildrenOnly
QDebug() << s; //输出: (QObject(0x840f38, name = "pc1"))
QDebug() << s[0]->c; //输出 1
QList<C*> s1=ma.findChildren<C*>("pc2", Qt::FindDirectChildrenOnly);
QDebug() << s1; //输出一个空圆括号, 因为 pc2 不是 ma 的直接子对象。
//查找 ma 的直接子对象
QList<C*> s2=ma.findChildren<C*>(QString(), Qt::FindDirectChildrenOnly);
QDebug() << s2; //输出: (QObject(0x82d638, name = "pa1"), QObject(0x82d478, name = "pa2"))
QList<C*> s3=ma.findChildren<C*>(); //获取 ma 的整个对象树结构的列表
QDebug() << s3;
//以上输出: (QObject(0x82d638, name = "pa1"), QObject(0x840f00, name = "pb1"),
//QObject(0x840f38, name = "pc1"), QObject(0x840bf0, name = "pc2"),
//QObject(0x840b48, name = "pb2"), QObject(0x82d478, name = "pa2"))

return 0; }

```

运行结果及说明见下图



2.6 事件

一、QApplication、QGuiApplication、QCoreApplication 简介

1、继承关系见下图, 其中左侧为顶级父类

QObject ← QCoreApplication ← QGuiApplication ← QApplication

- 2、一个程 序中只能有一个 QCoreApplication 及其子类的对象。
- 3、QCoreApplication: 主要提供无 GUI 程序的事件循环。
- 4、QGuiApplication: 用于管理 GUI 程序的控制流和主要设置。
- 5、QApplication: 该类专门为 QGuiApplication 提供基于 QWidget 的程序所需的一些功能, 主要用于处理部件的初始化、最终化。主要职责如下:

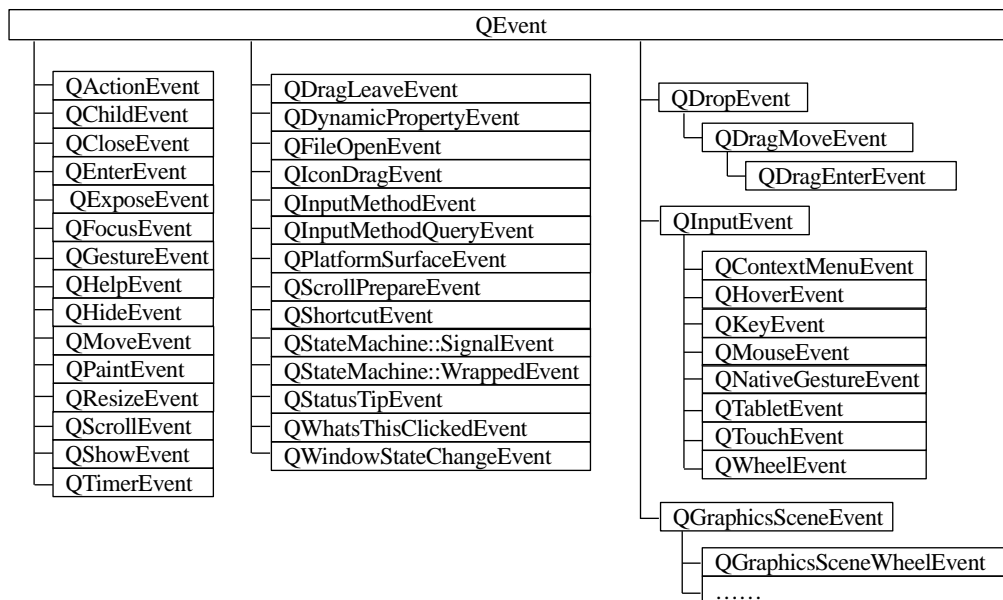
- 使用用户的桌面设置初始化应用程序。
- 执行事件处理，也就是说该类能从底层系统接收并分发事件。比如，使用 `QCoreApplication::sendEvent()` 或 `QCoreApplication::postEvent()` 函数分发自定义事件。
- 解析常用命令行参数并设置其内部状态。
- 定义了应用程序的界面外观，可使用 `QApplication::setStyle()` 进行更改。
- 指定应用程序如何分配颜色。
- 使用 `QCoreApplication::translate()` 函数对字符串进行转换。
- 通过 `QApplication::desktop()` 函数处理桌面，通过 `QCoreApplication::clipboard()` 函数处理剪贴板。
- 管理应用程序的鼠标光标。比如使用 `QGuiApplication::setOverrideCursor()` 函数设置光标等。

二、Qt 对事件的描述及分类

- 1、事件：是由程序内部或外部产生的事情或某种动作的通称。比如用户按下键盘或鼠标，就会产生一个键盘事件或鼠标事件(这是由程序外部产生的事件)；再如，当窗口第一次显示时，会产生一个绘制事件，以通知窗口需要重新绘制其自身，从而使该窗口可见(这是由程序内部产生的事件)。
- 2、事件和信号：
 - 他们两个是不同的概念，不要弄混淆。信号是由对象产生的，而事件则不一定是由对象产生的(比如由鼠标产生的事件)，事件通常来自底层的窗口系统，但也可以手动发送自定义的事件，可见信号和事件的来源是不同的。
 - 事件既可以同步使用，也可以异步使用(取决于调用 `sendEvent()` 还是 `postEvents()`)，而使用信号和槽总是同步的。事件的另一个好处是可以被过滤。
- 3、Qt 中使用 `QEvent` 及其子类来描述事件(其继承关系见下图)，比如 `QMouseEvent` 类用于描述与鼠标相关的事件，该类保存了与鼠标相关的大量信息，比如是哪一个键激发了该事件、产生该事件时鼠标的位置等。
- 4、事件的分类：
 - ①、方式一：根据事件的来源和传递方式，事件可分为以下三大类
 - 自发事件：这是由窗口系统生成的，这些事件置于系统队列中，并由事件循环一个接一个地处理。
 - 发布的事件(Posted events)：该类事件由 Qt 或应用程序生成，这些事件由 Qt 排队，并由事件循环处理。
 - 发送的事件(Sent events)：该类事件由 Qt 或应用程序生成，这些事件直接发送到目标对象，不经过事件循环处理。
 - ②、方式二：事件被细分为很多种类型(有一百多种)，每一种类型使用 `QEvent` 类中的枚举常量进行表示，比如 `QMouseEvent` 管理的鼠标事件有鼠标双击、移动、按下等类型，这些类型分别使用 `QEvent::Type` 枚举类型中的枚举常量 `MouseButtonDoubleClick`、`MouseMove`、`MouseButtonPress` 表示。所有的类型分类请查阅帮助文档。可使用函数 `Type QEvent::type() const` 获取事件的类型
- 5、使用 Qt 编程，几乎不需考虑事件，因为当产生某种事件时，Qt 窗口部件都会发射一个相

应的信号(即 Qt 会把事件转换为一个对应的信号), 比如按钮被按下时, 会产生一个 `MouseButtonPress` 事件, Qt 会处理这一事件, 并且会发射一个 `clicked()` 单击信号, 程序员可以直接处理 `clicked()` 信号, 而不必处理底层的事件。

- 对于事件, 我们不需要知道 Qt 是怎样把事件转换为 `QEvent` 或其子类类型的对象的, 程序员只需要知道怎样传递和处理这些事件即可。比如对于按下鼠标事件, 不需要知道 Qt 是怎样把该事件转换为 `QMouseEvent` 类型的对象的, 只需要知道怎样传递和处理该事件即可。

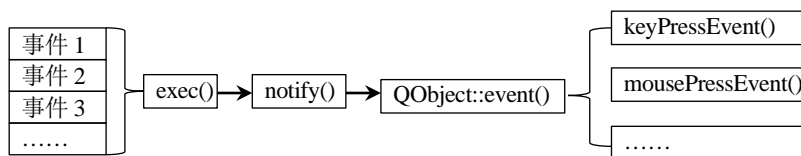


三、事件的传递(或分发)及处理

1、事件传递步骤(见下图):

- 基本规则: 若事件未被目标对象处理, 则把事件传递给它父对象处理, 若父对象仍未处理, 则再传递给父对象的父对象处理, 重复这个过程, 直至这个事件被处理或到达顶级对象为止。注意: 事件是在对象间传递的, 这里是指对象的父子关系, 而不是指类的父子关系。
- 在 Qt 中有一个事件循环, 该循环负责从可能产生事件的地方捕获各种事件, 并把这些事件转换为带有事件信息的对象, 然后由 Qt 的事件处理流程分发给需要处理事件的对象来处理事件。
- 通过调用 `QCoreApplication::exec()` 函数启动事件主循环, 主循环从事件队列中获取事件, 然后创建一个合适的 `QEvent` 对象或其子类类型的对象来表示该事件, 在此步骤中, 事件循环首先处理所有发布的事件, 直到队列为空, 然后处理自发的事件, 最后处理在自发事件期间产生的已发布事件。注意: 发送的事件不由事件循环处理, 该类事件会被直接传递给对象。
- 然后, Qt 会调用 `QCoreApplication::notify()` 函数对事件进行传递(或分发)
- 最后, `QObject` 对象调用 `QObject::event()` 函数接收事件,

- event()函数是一个虚函数，是 QObject 对象处理事件的入口，
- 在 QObject 的子类中通常会重写 event()函数，比如 QWidget 类就重写了 event()函数。
- event()函数与事件处理函数的关系：event()函数负责把事件传递给目标对象并调用对应的事件处理函数处理事件，比如调用 QWidget::keyPressEvent()函数处理键盘按下事件等，注意，QObject 中的 event()函数并不能实现该功能，这是通过 QObject 的子类中重写的 event()函数实现的，比如调用 QWidget::keyPressEvent()函数，就是由在 QWidget 类中重写的 event()函数来完成的。
- event()函数对大多数事件都调用了默认的处理函数，但并不能包括全部的事件，因此，有时我们需要重写该函数，这也是我们处理 Qt 事件的方式之一。
- 注意：event()函数不会处理事件，他只是根据事件的类型进行事件的传递(或分发)，若返回 true 则表示这个事件被接受并进行了处理，否则事件未被处理，需进行进一步传递或丢弃。



2、事件的处理

- ①、任何 QObject 的子类都可以处理事件，QEvent 及其子类虽然可以描述一个事件，但并不对事件进行处理。
- ②、事件处理函数

- 事件是由类的成员函数(通常为虚函数)处理的，通常把处理事件的成员函数称为事件处理函数。由此可见，处理事件需要两个要求：处理该事件的对象和该对象中用于处理该事件的事件处理函数。
- Qt 中对绝大多数常用类型的事件提供了默认的事件处理函数，并作了默认处理，这些事件处理函数位于 QObject 的子类中，比如 QWidget::mousePressEvent()函数用于处理鼠标按下事件，QWidget::keyPressEvent()用于处理键盘按下事件等。对这些默认事件处理函数的调用是通过 QObject::event()虚函数进行的，因此若默认的事件处理函数不能满足用户要求(比如对 Tab 键的处理)，则可以通过重新实现 event()函数来达到目的。下面为 QWidget 类的部分源代码(qwidget.cpp)

```

bool QWidget::event(QEvent *event) {
    .....
    switch (event->type()) {
        .....
        //调用 QWidget 类的默认事件处理函数
        case QEvent::MouseButtonPress:
            mousePressEvent((QMouseEvent*)event); break;
        case QEvent::MouseButtonRelease:
            mouseReleaseEvent((QMouseEvent*)event); break;
        .....
    }
    .....
}
  
```

3、由事件传递过程可见，Qt 可以使用以下 5 种方式来处理事件

- ①、重新实现各部件内部默认的事件处理函数，比如重新实现 `paintEvent()`、`mousePressEvent()`等事件处理函数，这是最常用、最简单的方式
- ②、重新实现 `QObject::event()`函数(需继承 `QObject` 类)，使用该方式，可以在事件到达默认的事件处理函数之前捕获到该事件。该方式常被用来处理 Tab 键的默认意义。在重新实现 `event()`函数时，必须对未明确处理的事件调用基类的 `event()`函数，比如，若子类化 `QWidget`，并重写了 `event()`函数，但未调用父类的 `event()`函数，则程序可能会不能正常显示界面。
- ③、在 `QObject` 对象上安装(或称为注册)事件过滤器(见后文)。对象一旦使用 `installEventFilter()`注册，传递给目标对象的所有事件都会先传递给这个监视对象的 `eventFilter()`函数。或同一对象上安装了多个事件处理器，则按照安装的逆序依次激活这些事件处理器。
- ④、在 `QApplication` 上安装(或称为注册)事件过滤器。该方式与重新实现 `notify` 函数一样强大。一旦在 `QApplication` 对象(程序中只有一个这种类型的对象)上注册了事件过滤器，则程序中每个对象的每个事件都会在发送到其他事件过滤器之前，发送到这个 `eventFilter()`函数。该方式对于调试非常有用。
- ⑤、子类化 `QApplication`，并重新实现 `QCoreApplication::notify()`函数，由事件传递过程可知，该函数提供了对事件的完全控制，因此功能非常强大，所以重写该函数会很复杂，也因此此方法很少被使用。这是唯一能在事件过滤器之前捕获到事件的方式。
- ⑥、注意：`exec()`、`notify()`和 `event()`函数都不会处理事件，他们只是根据事件的类型进行事件的传递(或分发)。

示例：事件处理的方式

```
#include <QApplication>
#include <QWidget>
#include <QObject>
#include <iostream>
using namespace std;

class A:public QWidget{public:
    bool event(QEvent* e); //事件处理方式 1: 重写虚函数 QObject::event()
    void mousePressEvent(QMouseEvent* e); }; //事件处理方式 2: 重写 QWidget 类中的此虚函数

bool A::event(QEvent* e){ static int i=0; //i 用于计数
    cout<<"E"<<i++<<endl; //验证该函数能在事件到达目标对象之前被捕获
    if(e->type()==QEvent::KeyPress) //判断是否是键盘按下事件。
        cout<<"keyDwon"<<endl;
    return QWidget::event(e); /*此处应调用父类的 event 函数以对未处理的事件进行处理，若此处不
                                调用父类的 event，则本例 mousePressEvent 处理函数中的内容将不会
                                被执行。读者可使此处返回 1 或者 0 进行验证。*/
}

void A::mousePressEvent(QMouseEvent* e){ //处理鼠标按下事件，这是 Qt 中最简单的事件处理方式。
    cout<<"mouseDwon"<<endl;
}

int main(int argc, char *argv[]){
    QApplication a(argc, argv); //在 Qt 中 QApplication 类型的对象只能有一个
```

```

A ma;    //创建一个部件
ma.resize(333,222); //设置部件的大小
ma.show(); //显示创建的部件

a.exec(); //在此处进入事件主循环。
return 0;    }

```

程序运行结果及说明(见右图)

只要窗口 ma 接收到事件就会输出 Exx，可见 event 函数捕获事件的范围及时机，当按下键盘上的任一键时会输出"keyDown"，当按下任一鼠标键时，会输出 mouseDown

```

E24
mouseDwon
E25
E26
E27
E28
keyDwon
E29
E30

```

示例：重写 notify 函数

```

#include <QApplication>
#include<QWidget>
#include<QObject>
#include <iostream>
using namespace std;

class A:public QWidget{public: bool event(QEvent* e); }; //重写虚函数 QObject::event()
class AA:public QApplication{public:
    AA(int i,char *p[]):QApplication(i,p) {}
    bool notify(QObject *o,QEvent *e);}; //重写虚函数 notify

bool A::event(QEvent* e){    cout<<"E"<<endl; //用于验证该函数是否被执行
    return QWidget::event(e);    } //应调用父类的 event 函数处理未处理的事件
bool AA::notify(QObject *o,QEvent *e){
    static int i=0;
    cout<<"N"<<i++<<endl;
    if(o->objectName()=="ma"&&o->type()==QEvent::KeyPress) //若对象为 ma 且事件为键盘按下事件
        cout<<"keyDwon"<<endl;
    return QApplication::notify(o,e); } /*应调用父类的 notify 函数，否则本示例的 event 函数不会被执行，同时无法关闭窗口*/

int main(int argc, char *argv[]){
    AA aa(argc,argv);//在 Qt 中 QApplication 或其子类型的对象只能有一个
    A ma;    //创建一个部件
    ma.setObjectName("ma"); //设置对象名，方便在 notify 函数中调用
    ma.resize(333,222); //设置部件的大小
    ma.show(); //显示创建的部件

    aa.exec();//在此处进入事件主循环。
    return 0;    }

```

程序运行结果及说明(见右图)

本示例可以看到 notify()函数和 event()函数的执行顺序，及 notify 函数和 event 函数捕获的事件的范围和时机，当用户按下键盘上的键时会输出 keyDown

```

N0
N1
N2
E
N3
E
N4
N5
E

```


示例：事件传递

```
#include <QApplication>
#include<QWidget>
#include<QPushButton>
#include<QObject>
#include <iostream>
using namespace std;

class A:public QWidget{public:    bool event(QEvent* e); };//子类化 QWidget
class C:public QPushButton{public:    bool event(QEvent* e);}; //子类化标准按钮 QPushButton

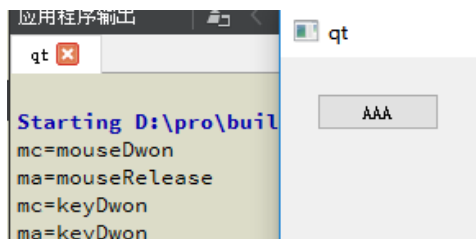
bool A::event(QEvent* e){
    if(e->type()==QEvent::KeyPress) //判断是否是键盘按下事件。
        {cout<<objectName().toString()<<"=keyDwon"<<endl; }
    if(e->type()==QEvent::MouseButtonPress) //判断是否是鼠标按下事件。
        {cout<<objectName().toString()<<"=mouseDwon"<<endl; }
    if(e->type()==QEvent::MouseButtonRelease) //判断是否是鼠标释放事件。
        {cout<<objectName().toString()<<"=mouseRelease"<<endl; }
    return QWidget::event(e);    }

bool C::event(QEvent* e){
    if(e->type()==QEvent::KeyPress) //判断是否是键盘按下事件。
        {cout<<objectName().toString()<<"=keyDwon"<<endl;
        return 0; } //将事件传递给父对象处理
    if(e->type()==QEvent::MouseButtonPress) //判断是否是鼠标按下事件。
        {cout<<objectName().toString()<<"=mouseDwon"<<endl;
        return 1; } //事件不传递
    return QWidget::event(e);}

int main(int argc, char *argv[]){
    QApplication a(argc, argv); //在 Qt 中 QApplication 类型的对象只能有一个
    A ma;    C *mc=new C();
    mc->setParent(&ma); /*设置父对象为 ma，若 mc 未处理的对象会传递给此处设置的父对象 ma 处理，
        注意事件传递是在对象之间的父子关系，而不是类之间的父子关系。*/
    mc->setText("AAA");
    mc->move(22, 22); //设置 mc 相对于 ma 的位置
    ma.setObjectName("ma"); //设置对象名
    mc->setObjectName("mc");
    ma.resize(333, 222); //设置部件的大小
    ma.show();
    a.exec(); //在此处进入事件主循环。
    return 0;    }
```

程序运行结果及说明(见图)

- 1、当在按钮上按下鼠标键时，调用 C::event() 函数输出 mc=mouseDown，
- 2、此时该函数返回 1，表示事件不再传递
- 3、但是该函数并未处理鼠标释放事件，因此调用 QWidget::event()对该事件作默认处理，因默认未处理该事件，因此调用 mc 的父



输出结果

本示例创建的窗口

对象 ma 处理该事件，此时调用 A::event()

函数输出 ma=mouseRelease，至此鼠标事件处理结束。

- 4、当按钮获得焦点，按下键盘上的按键时的处理方式与鼠标事件类似，只是 C::event()函数直接把键按下事件传递给了父对象 ma 处理。

四、事件的接受和忽略

- 1、事件可以被接受或忽略，被接受的事件不会再传递给其他对象，被忽略的事件会被传递给其他对象处理，或者该事件被丢弃(即没有对象处理该事件)
- 2、使用 QEvent::accept()函数表示接受一个事件，使用 QEvent::ignore()函数表示忽略一个事件。也就是说若调用 accept()，则事件不会传递给父对象，若调用 ignore()则事件会向父对象传递。
- 3、Qt 默认值是 accept(接受事件)，但在 QWidget 的默认事件处理函数(比如 keyPressEvent())中，默认值是 ignore()，因为这样可实现事件的传递(即子对象未处理就传递给父对象处理)。对事件的接受和忽略，最好是明确的调用 accept()和 ignore 函数。
- 4、在 event()函数中调用 accept()或 ignore()是没有意义的，event()函数通过返回一个 bool 值来告诉调用者是否接受了事件(true 表示接受事件)。是否调用 accept()是用于事件处理函数与 event()函数之间通信的，而 event()函数返回的 bool 值是用于与 QApplication::notify()函数之间通信的。
- 5、注意：QCloseEvent(关闭事件)有一些不同，QCloseEvent::ignore()表示取消关闭操作，而 QCloseEvent::accept()则表示让 Qt 继续关闭操作。为避免产生混淆，最好在 closeEvent()处理函数的重新实现中显示地调用 accept()和 ignore()。

示例：事件的接受和忽略

```
#include <QApplication>
#include <QWidget>
#include <QKeyEvent>
#include <QMouseEvent>
#include <QPushButton>
#include <QObject>
#include <iostream>
using namespace std;

class A:public QWidget{public:
    //重写以下事件处理函数
    void mousePressEvent(QMouseEvent *e){ cout<<"AmouseDwon"<<endl;} //鼠标按下
    void mouseReleaseEvent(QMouseEvent *e){ cout<<"AmouseRelease"<<endl;} //鼠标释放
    void keyPressEvent(QKeyEvent* e){ cout<<"AkeyDwon"<<endl;} //键盘按下
    void keyReleaseEvent(QKeyEvent* e){ cout<<"AkeyRelease"<<endl;} }; //键盘释放

class C:public QPushButton{public:
    void mousePressEvent(QMouseEvent *e){ cout<<"CmouseDwon"<<endl;
        e->accept(); } //验证事件被接受后不会再被传递给父对象
    void mouseReleaseEvent(QMouseEvent *e){ cout<<"CmouseRelease"<<endl;
        QWidget::mouseReleaseEvent(e); } //验证 QWidget 默认为忽略事件
    void keyPressEvent(QKeyEvent* e){ cout<<"CkeyDwon"<<endl;
        e->ignore(); } //验证事件被忽略后会被传递给父对象
    void keyReleaseEvent(QKeyEvent* e){ cout<<"CkeyRelease"<<endl;
```

```

        //验证 Qt 的默认处理方式接受事件(此处未明确调用 accept() 或 ignore() 函数)
    }
};

int main(int argc, char *argv[]) {
    QApplication a(argc, argv); //在 Qt 中 QApplication 类型的对象只能有一个
    A ma;    C *mc=new C();
    mc->setParent(&ma); //设置父对象为 ma
    mc->setText("AAA");
    mc->move(22, 22); //设置 mc 相对于 ma 的位置
    ma.resize(333, 222); //设置部件的大小
    ma.show();
    a.exec(); //在此处进入事件主循环。
    return 0;
}

```

程序运行结果及说明(见图)

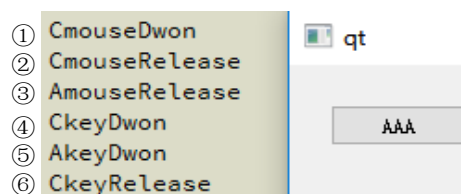
1、当在键钮上按下鼠标时，调用

C::mousePressEvent()函数，该函数接受该事件，因此事件不再传递。因此仅输出 CmouseDown

2、当释放鼠标时调用 C::mouseReleaseEvent() 函数输出 CmouseRelease，该函数调用 QWidget 的默认事件处理函数忽略该事件，因此事件传递给 mc 的父对象 ma 处理，

此时调用 A::mouseReleaseEvent()函数输出 AmouseRelease。

3、④~⑥的键盘事件与鼠标事件类似，只是⑥表示的是 Qt 的默认处理方式接受该事件



输出结果

本示例创建的窗口

示例：event() 函数与事件处理函数的关系。

```

#include <QApplication>
#include<QWidget>
#include<QMouseEvent>
#include<QObject>
#include <iostream>
using namespace std;

class A:public QWidget{public:
    bool event(QEvent* e) {
        if(e->type()==QEvent::MouseButtonPress) //处理鼠标事件
            {cout<<"AE"<<endl;
             mousePressEvent((QMouseEvent*)e); /*明确调用鼠标事件处理函数处理该事件，需要注意的是 Qt 对事件处理函数的默认处理方式是接受事件。*/
             //return QWidget::event(e); /*也可通过父类的 event 函数间接的调用鼠标事件处理函数。
             return 0; } //event() 函数的返回值与事件处理函数没有关系，返回 0 只表示该事件可以传递给父对象处理，返回 1 则事件不再传递。*/
        return QWidget::event(e); //其他事件使用父类的 event 函数处理
    }

    void mousePressEvent(QMouseEvent *e){cout<<"AK"<<endl; } };

int main(int argc, char *argv[]) {
    QApplication aa(argc, argv);
    A ma;    ma.resize(333, 222);    ma.show();    aa.exec();    return 0;
}

```

运行结果说明：在窗口中点击鼠标会先后输出 AE 和 AK

五、事件过滤器

- 1、事件过滤器用于拦截传递到目标对象的事件，这样可以实现监视目标对象事件的作用
- 2、Qt 实现事件过滤器的步骤如下：

- ①、Qt 调用

```
void QObject::installEventFilter (QObject* filterObj)
```

把 filterObj 对象设置安装(或注册)为事件过滤器，filterObj 也称为过滤器对象。事件过滤器通常在构造函数中进行注册。

- ②、在上一步注册的 filterObj 对象，通过调用

```
bool QObject::eventFilter(QObject* obj, QEvent* e);
```

来接收拦截到的事件。也就是说拦截到的事件在 filterObj 对象中的 eventFilter 函数中处理。eventFilter 的第一个参数 obj 指向的是事件本应传递到的目标对象。

- ③、使用 QObject::removeEventFilter(QObject *obj)函数可以删除事件过滤器。

- 3、事件过滤器处理事件的规则

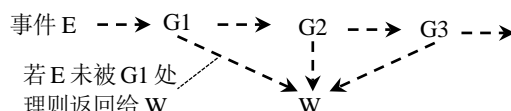
- ①、过滤器对象的 eventFilter()函数可以接受或拒绝拦截到的事件，若该函数返回 false，则表示事件需要作进一步处理，此时事件会被发送到目标对象本身进行处理(注意：这里并未向父对象进行传递)，若 eventFilter()返回 true，则表示停止处理该事件，此时目标对象和后面安装的事件过滤器就无法获得该事件。

- ②、若同一对象安装了多个事件过滤器，则最后安装的过滤器首先被激活。

- 4、为什么使用事件过滤器：使用事件过滤器可以简化程序代码。比如按钮 1 和标签 1，对按下 A 键的事件响应相同的操作，若不使用事件过滤器，则需要分别子类化按钮和标签部件，并重新实现各自的事件处理函数。再如使用同一个子类化按钮的类 C 创建的按钮 1 和按钮 2，对按下键 A，按钮 1 和按钮 2 需要作不同的响应，若不使用事件过滤器，则他们的响应是相同的，若使用事件过滤器，则可以拦截按钮 1 或按钮 2 的事件并作特殊处理。

- 5、理解事件过滤器

观察者模式：其原理为，设有一目标对象 S，它有多多个观察该对象的对象 G1，G2，G3，当 S 发生变化时，S 的观察者会依情形改变自身。应用于 Qt 事件过滤器，则是，首先使用 S 的成员函数 installEventFilter 函数把 G1，G2，G3 设置为 S 的观察者，所有本应传递给 S 的事件 E，则先传递给观察者 G1，G2，G3，然后观察者调用其成员函数 eventFilter 对传递进来的事件进行处理，若 eventFilter 返回 true 表示事件处理完毕，返回 false 则返回给被观察者 S 进行处理。见下图。



示例：事件过滤器的使用

```
#include <QApplication>
#include <QWidget>
```

```

#include<QMouseEvent>
#include<QPushButton>
#include<QObject>
#include <iostream>
using namespace std;

class A:public QObject{public: //该类的对象用作过滤器对象，使用事件过滤器需继承 QObject
    bool eventFilter(QObject *w, QEvent *e){
        if(e->type()==QEvent::MouseButtonPress)
            {cout<<w->objectName().toString(); //验证 w 为事件本应到达的目标对象
              cout<<"=Ak"<<endl;
              return 1; //返回 1 表示该事件不再进一步处理
            }
        return 0;} //返回 0 表示其余事件交还给目标对象处理，本例应返回 0，否则添加了该
                    //过滤器的按钮将无法显示。*/
};

class B:public A{public: //继承自类 A
    bool eventFilter(QObject *w, QEvent *e){
        if(e->type()==QEvent::MouseButtonPress){
            cout<<w->objectName().toString()<<"=Bk"<<endl;
            return 0;}
        return 0;}
};

class C:public QWidget{public: void mousePressEvent(QMouseEvent *e){cout<<"Ck"<<endl;}};
class D:public QPushButton{public: void mousePressEvent(QMouseEvent *e){cout<<"Dk"<<endl;}};

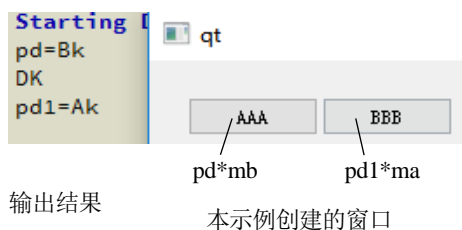
int main(int argc, char *argv[]){
    QApplication a(argc, argv);
    //创建对象，注意：本例父对象应先创建，以避免生命期过早结束
    A ma;      B mb;      C mc;      D *pd=new D;      D *pd1=new D;
    pd->setText("AAA");    pd->move(22, 22);    pd1->setText("BBB");    pd1->move(99, 22);
    //设置对象名
    ma.setObjectName("ma");    mb.setObjectName("mb");    mc.setObjectName("mc");
    pd->.setObjectName("pd");    pd1->.setObjectName("pd1");
    //设置父对象
    pd->setParent(&mc);    pd1->setParent(&mc);
    mb.setParent(&ma);    //①
    //注册过滤器对象
    pd->installEventFilter(&mb); //②
    pd1->installEventFilter(&ma); //③

    mc.resize(333, 222);    mc.show();    a.exec();
    return 0;    }

```

程序运行结果及说明(见图)

- 1、当用鼠标按下按钮 AAA 时，输出 pd=Bk 和 Dk。因为按钮 AAA 安装的过滤器对象为 mb，因此由 mb 的 eventFilter 函数处理该事件，输出 pd=BK，
- 2、此时 mb::eventFilter()返回 0，表示此事件需作进一步处理，



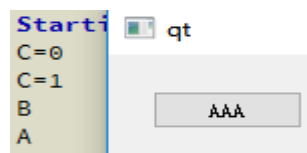
- 3、于是把该事件传递给目标对象处理(即 `pd` 所指向的对象), 注意: 本例虽然为 `mb` 设置了父对象 `ma`, 但事件并不会传递给父对象处理, 而是返回给目标对象。
- 4、此时调用 `D::mousePressEvent` 函数, 输出 `Dk`, 至此事件处理结束。
- 5、用鼠标按下按钮 `BBB` 输出 `pd1=Ak` 的原理略(较简单)

示例：添加多个事件过滤器

```
#include <QApplication>
#include<QWidget>
#include<QMouseEvent>
#include<QPushButton>
#include<QObject>
#include <iostream>
using namespace std;
class A:public QObject{public:
    bool eventFilter(QObject *w, QEvent *e){
        if(e->type()==QEvent::MouseButtonPress)
            { cout<<"A"<<endl;
              return 0; } //此处应返回 0, 注意, 返回 1 事件将停止传递。
        return 0;} };
class B:public QObject{public:
    bool eventFilter(QObject *w, QEvent *e){
        if(e->type()==QEvent::MouseButtonPress){ cout<<"B"<<endl;return 0; }
        return 0;} };
class C:public QObject{public:
    bool eventFilter(QObject *w, QEvent *e){
        static int i=0;
        if(e->type()==QEvent::MouseButtonPress){ cout<<"C="<<i++<<endl;return 0; }
        return 0;} };
int main(int argc, char *argv[]){
    QApplication a(argc,argv); //在 Qt 中 QApplication 类型的对象只能有一个
    //创建对象
    A ma; B mb; C mc,mcl; //事件过滤器对象
    QWidget w; QPushButton *pb=new QPushButton();
    pb->setText("AAA"); pb->move(22,22);
    pb->setParent(&w); //设置父对象
    //注册过滤器对象
    pb->installEventFilter(&ma); pb->installEventFilter(&mb);
    pb->installEventFilter(&mc); pb->installEventFilter(&mcl);
    w.resize(333,222); w.show(); a.exec(); return 0; }
```

程序运行结果及说明(见图)

按钮 `AAA` 安装的过滤器对象依次为 `ma`, `mb`, `mc`, `mcl`, 因此按下鼠标时, 依次调用对象 `mcl`, `mc`, `mb`, `ma`(即逆序)的 `eventFilter` 函数, 需要注意的是: 当安装了多个事件过滤器之后, `eventFilter` 函数返回 0 并不会使事件返回给目标对象, 而是传递给下一个过滤器对象, 当所有过滤器对象都不处理该事件时才会传递给目标对象。



本示例创建的窗口

六、自定义事件与事件的发送

1、发送事件由以下两个函数完成

```
static void QCoreApplication::postEvent (QObject* receiver, QEvent* event,  
int priority=Qt::NormalEventPriority);  
static bool QCoreApplication::sendEvent(QObject* receiver, QEvent* event)
```

- receiver: 指向接收事件的对象
- event: 表示需要发送的事件
- priority: 表示事件的优先级, 事件会按优先级排序, 高优先级的事件排在队列的前面。其取值为枚举类型 Qt::EventPriority 中的枚举值, 如下

Qt::HighEventPriority: 值为 1。

Qt::NormalEventPriority: 值为 0。

Qt::LowEventPriority: 值为 -1。

优先级只是一个相对值, 其值可取介于规定的最大值和最小值之间的任何值, 比如可使 priority 参数的值为 Qt::HighEventPriority + 10。

2、发送事件(sendEvent)与发布事件(postEvent)

①、发布(post)事件:

- 把事件添加到事件队列中, 并立即返回。
- 发布事件必须在堆(比如使用 new)上创建事件, 因为事件被发布后, 事件队列将获得事件的所有权并自动将其删除。发布事件后再访问该事件是不安全的。
- 发布事件还可以对事件进行合并(或称为压缩), 比如在返回事件循环之前连续发布了多个相同的事件, 则这多个相同的事件会自动合并为一个单一的事件。可合并的事件有鼠标移动事件、调整大小事件等。

②、发送(send)事件: 把事件直接发送给接收事件的目标对象。事件发送之后不会被删除, 发送的事件通常创建在堆栈上。

3、自定义事件原理

- ①、基本原理: 事件其实就是使用的一个整数值表示的, 因此在创建自定义事件时, 只须给事件指定一个整数值即可, 在 Qt 中, 这个整数值是通过枚举类型 QEvent::Type 定义的, 事件的其他信息可以封装在一个自定义的类之中。
- ②、自定义的事件即可以是预定义类型, 也可以是自定义类型的。
- ③、自定义类型的事件, 需要定义一个事件编号, 该编号必须大于 QEvent::User(其值为 1000), 小于 QEvent::MaxUser(其值为 65535)。
- ④、各种事件不能重叠(即 QEvent::Type 类型的值不能相同), 小于 QEvent::User 的事件是 Qt 内部定义的事件, 他们不会重叠, 对于自定义的事件可以使用 registerEventType() 函数来保证事件不重叠, 该函数原型如下:

```
static int QEvent::registerEventType ( int hint = -1 );
```

如果 hint 的值不会产生重叠, 则会返回这个值; 如果 hint 不合法, 系统会自动分配一个合法值并返回。因此, 可使用该函数的返回值创建 Type 类型的值。

4、创建自定义事件的方法和步骤

- ①、可以使用以下方式创建自定义事件

- 使用 QEvent 的构造函数创建事件，其原型为：

QEvent(Type type);

示例：QEvent::Type t1=(QEvent::Type)1333; //定义事件编号

QEvent e(t); //创建事件 e

- 使用 Qt 已创建好的事件类型创建事件，比如使用 QKeyEvent 类创建键盘事件。
- 继承 QEvent 类，创建自定义事件。

②、使用 QCoreApplication::postEvent()或 QCoreApplication::sendEvent()函数发送事件。

③、可使用以下方法处理自定义事件

- 重写 QObject::event()函数，在该函数内直接处理自定义事件或调用自定义的事件处理函数处理事件。
- 安装事件过滤器，然后在过滤器对象的 eventFilter()函数中处理自定义事件。
- 当然，还可以重写 QCoreApplication::notify()函数。

示例：自定义事件的使用

```
#include <QApplication>
#include<QWidget>
#include<QObject>
#include <iostream>
using namespace std;
```

```
QEvent::Type t1=(QEvent::Type)1333;
QEvent e(t1);            //使用 QEvent 的构造函数在堆栈上创建自定义事件
```

```
class E:public QEvent{public:      //子类化 QEvent 以创建自定义事件
```

```
//方式 1：使用静态成员。
```

```
    //使用静态成员主要是为了正确初始化父类部分 QEvent，比如
```

```
    //E():t2((QEvent::Type)1324),QEvent(t2) {}，若 t2 不是静态的，则则初始化之后 t2 为 1324，但传递
```

```
    //给 QEvent 的 t2 是一个不确定的值，因为按照 C++规则，对父类部分的初始化先于数据成员的初始化。
```

```
    static QEvent::Type t2;    //注意: 不要使用名称 t, 因为 QEvent 类之中有一个名称为 t 的成员变量。
```

```
    E():QEvent(t2) {}
```

```
//方式 2：使用带一个参数的构造函数
```

```
    QEvent::Type t3;
```

```
    explicit E(QEvent::Type t4):t3(t4),QEvent(t4) {}
```

```
};
```

```
QEvent::Type E::t2=(QEvent::Type)1334;
```

```
class A:public QWidget{public:
```

```
    bool event(QEvent* e) {    //重写 event 函数以处理自定义事件
```

```
        if(e->type()==t1)    //判断事件类型是否为 t1
```

```
            {cout<<"AE"<<e->type()<<"，";
```

```
                f1((E*)e);    //调用自定义的处理函数处理该事件
```

```
                return 1; } }
```

```
        if(e->type()==E::t2) {cout<<"BE"<<e->type()<<"，";    f2((QEvent*)e); return 1; }
```

```
        if(e->type()==((E*)e)->t3) {cout<<"CE"<<e->type()<<"，";    f3((E*)e); return 1; }
```

```
        return QWidget::event(e);
```

```
    }    //event 结束
```

```
//以下为处理自定义事件的事件处理函数
```

```
void f1(E *e) {cout<<"F1"<<endl;}
```



```

    void f2(QEvent *e) {cout<<"F2"<<endl;}
    void f3(E *e) {cout<<"F3"<<endl;}
}; //类A 结束。
int main(int argc, char *argv[]) {    QApplication aa(argc, argv);
    A ma;    E me;    E *pe=new E((QEvent::Type)1335);
    //发布或发送事件
    aa.sendEvent(&ma,&e);    aa.sendEvent(&ma,&me);    aa.postEvent(&ma,pe);
    //aa.postEvent(&ma,&me); //错误，发布的事件 me 必须是在堆上创建的。
    ma.resize(333,222);
    ma.show();    aa.exec();    return 0;    }

```

运行程序后，输出结果依次为 AE1333,F1 BE1334,F2 CE1335,F3

示例：使用事件过滤器处理自定义事件

```

#include <QApplication>
#include<QWidget>
#include<QObject>
#include <iostream>
using namespace std;

QEvent::Type t1=(QEvent::Type)QEvent::registerEventType(1333);
QEvent e1(t1); //使用 QEvent 的构造函数创建自定义事件
//t2 的值与 t1 重复，使用 registerEventType 会自动产生一个合法的值
QEvent::Type t2=(QEvent::Type)QEvent::registerEventType(1333);
QEvent e2(t2);

class A:public QWidget{public:
    bool event(QEvent* e) {
        if(e->type()==t1) {cout<<"AE"<<e->type()<<",";    f1((QEvent*)e);    return 1; }
        if(e->type()==t2) {cout<<"BE"<<e->type()<<",";    f2((QEvent*)e);    return 1; }
        return QWidget::event(e);    } //event 结束
    void f1(QEvent *e) {cout<<"F1"<<endl;}
    void f2(QEvent *e) {cout<<"F2"<<endl;}
};

class B:public QObject{public:
    bool eventFilter(QObject *w, QEvent *e) {
        if(e->type()==t1) {    cout<<"A"<<endl;return 1; }
        if(e->type()==t2) {    cout<<"B"<<endl;return 0; }
        return 0;}
};

int main(int argc, char *argv[]) {    QApplication aa(argc, argv);
    A ma;    B mb;
    ma.installEventFilter(&mb); //安装事件过滤器
    aa.sendEvent(&ma,&e1);    aa.sendEvent(&ma,&e2);
    ma.resize(333,222);    ma.show();    aa.exec();    return 0;    }

```

本例依次输出 A B BE65535,F2， 其中 65535 为使用 registerEventType 函数为 t2 分配的合法的数值。可见，安装事件过滤器之后，发送的事件 e1 和 e2 会使用过滤器对象的 eventFilter 函数进行处理。

七、事件的传递顺序总结

- 1、event()函数、事件过滤器、和事件处理函数的调用顺序如下：
首先按逆序调用事件过滤器，然后调用 event()函数，最后调用事件处理函数(注意：事件处理函数需在 event()函数中明确或间接调用，否则不会调用事件处理函数)。

八、鼠标和键盘事件共同使用的类及函数

- 1、通过调用从 QInputEvent::modifiers()函数可以查询键盘修饰键的状态。
- 2、使用 QWidget::setEnabled()函数可以启用或禁用部件的鼠标和键盘事件。
- 3、键盘修饰键(即 Ctrl、Alt、Shift 等键)使用枚举类型 Qt::KeyboardModifier 来描述，其取值如下表

Qt::KeyboardModifier 枚举的成员		
标志类型为: typedef QFlags<KeyboardModifier> KeyboardModifiers		
枚举值	值	说明
Qt::NoModifier	0x0000 0000	无修饰键被按下
Qt::ShiftModifier	0x0200 0000	按下 Shift 键
Qt::ControlModifier	0x0400 0000	按下 Ctrl 键
Qt::AltModifier	0x0800 0000	按下 Alt 键
Qt::MetaModifier	0x1000 0000	按下 Meta 键(windows 系统为 windows 键)
Qt::KeypadModifier	0x2000 0000	按下小键盘上的键
Qt::GroupSwitchModifier	0x4000 0000	按下 Mode_switch 键(仅限 X11)

九、鼠标事件

- 1、Qt 使用 QMouseEvent 类来描述与鼠标有关的信息，比如鼠标的位置坐标，键是否被按下等，使用枚举类型 QEvent::Type 描述与鼠标有关的事件，比如 QEvent::MouseButtonPress 表示鼠标按下事件，QEvent::MouseMove 表示鼠标移动事件等。
- 2、是否接收鼠标事件，最好明确的调用 QEvent::ignore()和 QEvent::accept()函数。
- 3、下面是 Qt 在默认情况下对鼠标事件的处理，
 - ①、注意：只有在按下鼠标按钮(左、右、中键都可)时才会发生鼠标移动事件(即 QEvent::MouseMove)，使用 QWidget :: setMouseTracking () 函数启用鼠标跟踪后，就即使不按下鼠标按钮也会产生鼠标移动事件。
 - ②、Qt 在部件内按下鼠标按钮时会自动抓取鼠标，也就是说，当按下鼠标后，将鼠标移除该小部件将继续接收鼠标事件，直到最后一个鼠标按钮被释放。可使用 QWidget::grabMouse()函数抓取鼠标，使用 QWidget::releaseMouse()函数释放鼠标抓取。
- 4、Qt 使用枚举类型 Qt::MouseButton 来描述鼠标的信息，下表为该枚举定义的部分成员
注意：QMouseButtons(最后多一个 s)的类型为

Qt::MouseButton 枚举的成员		
标志类型为: typedef QFlags<MouseButton> QMouseButtons		
枚举值	值	说明
Qt::NoButton	0x0000 0000	按钮状态不指向任何按钮。

Qt::LeftButton	0x0000 0001	按下左键
Qt::RightButton	0x0000 0002	按下右键
Qt::MidButton 或 Qt::MiddleButton	0x0000 0004	按下中键

5、下面为 QMouseEvent 类中的成员函数

①、QMouseEvent(Type type, const QPointF &localPos, Qt::MouseButton button, Qt::MouseButtons buttons, Qt::KeyboardModifiers modifiers);

- type: 表示鼠标事件的类型，必须是 QEvent::MouseButtonPress、QEvent::MouseButtonRelease、QEvent::MouseButtonDblClick、QEvent::MouseMove 之一。
- localPose: 鼠标光标的位置。其中 QPointF 是一个用于描述点的类，该类类似于 QPoint，只是 QPointF 能接受浮点类型的坐标值，比如 QPointF pf(33.3, 22.4)，表示创建一个位于(33.3, 22.4)的点。
- button: 表示产生鼠标事件的按钮，若事件类型是 MouseEvent，则此值是 Qt::NoButton
- buttons: 产生鼠标事件时哪些鼠标按钮处于按下状态。
- modifiers: 键盘修饰键
- 说明: 这是 QMouseEvent 类的构造函数之一(共有 4 个)，本文仅列举这一个，构造函数可以用于创建 QMouseEvent 类型的自定义事件，构造函数的参数虽然较多，但是，这些参数只是为创建的事件提供了一些必要的信息而已。

示例:

```
QMouseEvent e(QEvent::MouseButtonPress, QPointF(33,33),
              Qt::LeftButton, Qt::LeftButton, Qt::NoModifier);
```

表示创建一个类型为 QEvent::MouseButtonPress 的鼠标按下事件，产生该事件时鼠标的光标位于(33,33)处，该事件是由鼠标左键产生的，且产生该事件时鼠标左键处于按下状态，且没有键盘修饰键被按下。创建该自定义事件后，可以使用 sendEvent 进行发送，并使用部件类的 event()函数接收并处理该事件。

②、Qt::MouseEventSource source() const; //qt5.3

返回鼠标事件的来源信息。鼠标事件除了可以来自于物理鼠标之外，还可以来自于其他来源，比如触摸屏的仿真鼠标事件。鼠标事件的来源使用枚举 Qt::MouseEventSource 来描述，该枚举可取以下值。

Qt::MouseEventSource 枚举的成员		
枚举值	值	说明
Qt::MouseEventNotSynthesized	0	最常见的值。表示鼠标事件来源于真正的鼠标
Qt::MouseEventSynthesizedBySystem	1	由触摸事件合成的鼠标事件
Qt::MouseEventSynthesizedByQt	2	由 Qt 未处理的触摸事件合成的鼠标事件
Qt::MouseEventSynthesizedByApplication	3	由应用程序合成的鼠标事件。(qt5.6)

③、Qt::MouseButton button() const; //返回产生鼠标事件的按钮

Qt::MouseButtons buttons() const; //返回产生鼠标事件时处于按下状态的按钮。

以上两函数的区别在于 `buttons` 返回的值是 `Qt::MouseButton` 类型，这是一个 `QFlags` 模板类型，而 `Qt::MouseButton` 仅仅是一个枚举类型，因此 `buttons` 函数的返回值可以是 `Qt::LeftButton`, `Qt::RightButton`, `Qt::MidButton` 使用 OR 运算符的组合。具体见下表，读者可自行写出其示例程序，并验证。

button()和 buttons()的区别			
注：为缩减文字长度，产生的事件使用中文描述，而不用 QEvent 中的 Type 枚举值			
	产生的事件	button()	buttons()
按下左键	键按下	Qt::LeftButton	Qt::LeftButton
接着按下右键	键按下	Qt::RightButton	Qt::LeftButton Qt::RightButton
释放左键	键释放	Qt::LeftButton	Qt::RightButton
接着移动鼠标	鼠标移动	Qt::NoButton	Qt::RightButton
最后释放右键	键释放	Qt::RightButton	Qt::NoButton

以下函数为与鼠标光标位置有关的函数，有关这些位置之间的坐标转换位于 `QWidget` 类中

- ④、`int x() const; int y() const;` //返回鼠标光标的 x 和 y 位置。相对于产生事件的部件
- ⑤、`QPoint pos() const;` //返回鼠标光标的位置。相对于产生事件的部件
- ⑥、`const QPointF& localPos() const;` //返回鼠标光标的位置。相对于产生事件的部件，qt5.0
- ⑦、`const QPointF& screenPos() const;` //返回鼠标光标的位置。相对于显示屏，qt5.0
- ⑧、`const QPointF& windowPos() const;` //qt5.0
返回鼠标光标的位置(相对于产生事件的窗口)，第 6~8 的函数需要注意，他们返回的是 `const` 类型的对象，按 C++语法，`const` 类型的对象只能调用 `const` 类型的成员函数。
- ⑨、`int globalX()const; int globalY()const;` //返回鼠标光标的全局 x 和 y 位置。
- ⑩、`QPoint globalPos() const;` //返回鼠标光标的全局位置。

示例：鼠标光标的位置

```
#include<QWidget>
#include<QMouseEvent>
#include<QPushButton>
#include<QObject>
#include <iostream>
using namespace std;

class B:public QObject{public:
    bool eventFilter(QObject *watched, QEvent *event){
        QMouseEvent *e=(QMouseEvent*)event;
        if(e->type()==QEvent::MouseButtonPress){ //是否是鼠标按下事件
            cout<<"x="<<e->x()<<endl; //返回值是相对于本例的按钮 AAA 左上角的。
            cout<<"pos="<<e->pos().x()<<endl; //同上
            /*注意：localPos()、windowPos()、screenPos() 三个函数返回的是 const 类型的对象，只能调用
            const 类型的成员函数*/
            cout<<"local="<<e->localPos().x()<<endl; //同上
            cout<<"window="<<e->windowPos().x()<<endl; //返回值是相对于 w 左上角的
            cout<<"screen="<<e->screenPos().x()<<endl; //返回值是相对于屏幕左上角的
            cout<<"global="<<e->globalPos().x()<<endl; } //同上，if 结束
```

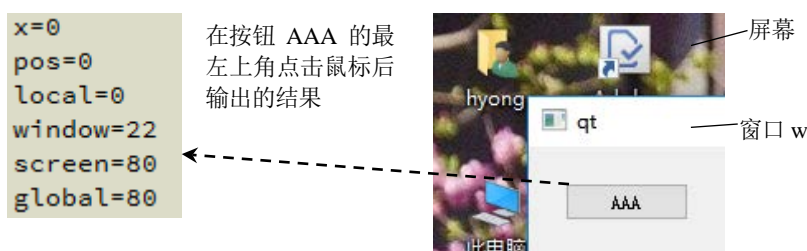
```

return 0;    //返回 0，以把其余事件交还给目标对象处理。
}    //eventFilter 结束
};

int main(int argc, char *argv[]) {
    QApplication aa(argc, argv); //在 Qt 中 QApplication 或其子类型的对象只能有一个
    QWidget w;    B mb;    QPushButton *pb=new QPushButton(&w);
    pb->move(22,22);    //使按钮位于相对于 w 左上角(22,22)处。
    pb->setText("AAA");    pb->installEventFilter(&mb);    w.resize(333,222);
    w.move(50,50);    //使窗口部件在相对屏幕左上角(50,50)处显示
    w.show();    aa.exec();    return 0;    }

```

运行结果及说明



6、QWidget 类中与鼠标事件有关的函数

①、void grabMouse()

捕获鼠标输入，鼠标输入被捕获后，直到调用 releaseMouse()函数之前，所有的鼠标事件都由该部件接收，其他部件则无法接收到鼠标事件。只有可见的部件才能捕获鼠标输入，若 isVisible()返回 false，则该部件不能调用 grabMouse()函数。使用 Qt 时，通常不需要捕获鼠标，因为 Qt 会抓住并释放鼠标，特别地，当鼠标按钮被按下时，Qt 会捕获鼠标，并将其保持到最后一个按钮被释放为止。

②、void grabMouse(const QCursor &cursor) //捕获鼠标输入，并改变光标的形状。

③、void releaseMouse() //释放捕获的鼠标

④、static QWidget* mouseGrabber(); 返回正在捕获鼠标输入的部件，若没有则返回 0。

⑤、bool hasMouseTracking() const;

返回 true 表示启用了鼠标跟踪，返回 false 表示禁用鼠标跟踪(默认值)，若禁用了鼠标跟踪，则至少需要按下一个鼠标按钮才会产生鼠标移动事件。若启用了鼠标跟踪，则即使未按下鼠标按钮，也会产生鼠标移动事件。

⑥、void setMouseTracking(bool enable) 设置鼠标跟踪状态

⑦、bool underMouse() const;

若部件位于鼠标光标之下，则返回 true，否则返回 false。此值在拖放操作期间未正确更新。

示例：捕获鼠标输入

```
#include <QApplication>
#include<QWidget>
#include<QMouseEvent>
#include<QPushButton>
#include<QObject>
#include <iostream>
using namespace std;
class B:public QObject{public:
    bool eventFilter(QObject *watched, QEvent *event){
        QMouseEvent *e=(QMouseEvent*)event;
        QPushButton* w=(QPushButton*)watched;
        if(e->type()==QEvent::MouseButtonPress){
            cout<<"B"<<endl;
            /*判断光标是否位于部件之下，注：要使以下语句结果为 true，需注释掉 main 函数中的鼠标
            捕获代码 pb->grabMouse();*/
            //if(w->underMouse())    cout<<"under"<<endl;
            return 1; } // 返回 1，结束对此事件的处理。if 结束
    }
};
class C:public QObject{public:
    bool eventFilter(QObject *watched, QEvent *event){
        QMouseEvent *e=(QMouseEvent*)event;
        if(e->type()==QEvent::MouseButtonPress){ cout<<"C"<<endl;    return 1; }
        return 0;    }    };
class D:public QObject{public:
    bool eventFilter(QObject *watched, QEvent *event){
        QMouseEvent *e=(QMouseEvent*)event;
        if(e->type()==QEvent::MouseButtonPress){cout<<"D"<<endl; return 1;    }
        return 0;}    };
int main(int argc, char *argv[]){
    QApplication aa(argc,argv);
    QWidget w;    B mb; C mc; D md;
    QPushButton *pb=new QPushButton(&w);    QPushButton *pb1=new QPushButton(&w);
    pb->move(22,22);    pb->setText("AAA");    pb1->move(99,22);    pb1->setText("BBB");
    pb->installEventFilter(&mb);    pb1->installEventFilter(&mc);
    w.installEventFilter(&md);
    pb->grabMouse();    /*使按钮 AAA 捕获鼠标，此时产生的鼠标事件都只会发送给按钮 AAA，也就是说
                        其他部件无法获得鼠标事件。*/
    //pb->setMouseTracking(true); //鼠标跟踪与鼠标移动事件的关系，请读者自行增加代码进行验证。
    w.resize(333,222);    w.show();    aa.exec();    return 0;    }
```

运行结果及说明：该程序无论在程序的什么位置点击鼠标，鼠标事件都只会传送给按钮 AAA，因此只会输出 B

十、键盘事件

- 1、Qt 使用 QKeyEvent 类来描述与键盘有关的信息，比如按下或释放键的代码，使用枚举类型 QEvent::Type 描述与键盘有关的事件，比如 QEvent::KeyPress 表示键盘按下事件，QEvent::KeyRelease 表示键盘释放事件等。
- 2、是否接收键盘事件，最好明确的调用 QEvent::ignore()和 QEvent::accept()函数。

- 3、按键与字符：对于键盘，通常按下一个键会产生一个字符，比如按下 A 键，会产生字符 "a"，但有些键不会产生字符，比如 delete、insert 等键，而且按下 A 键不一定必须产生字符 "a"(比如在中文、德文等其他非英文国家键盘布局的情形下，按键 A 不一定会产生字符 "a")。
- 4、捕获键盘输入的原理与捕获鼠标输入是相同的，详见鼠标事件。
- 5、按键与自动重复：自动重复是指按下键盘上的键(修饰键除外)不放时，会不断重复的发送键按下事件，Qt 默认是启用自动重复的，若要实现类似按键 A+D 之类的快捷键，就需要关闭自动重复。可使用如下方法来关闭自动重复

```
if( QKeyEvent::isAutoRepeat() ) return; // 若自动重复则什么也不做。
```

- 6、压缩按键事件(默认为关闭):
 - ①、按下一个键不放时，会不断的发送键按下事件，若程序的处理跟不上键的输入，则 Qt 可能会尝试压缩该事件，以便在每个事件中处理多个字符。
 - ②、Qt 只对可打印字符启用键事件压缩，对于像 Enter、Backspace 等键不会启用键事件压缩。
 - ③、Mac 和 X11 以外的平台不支持此功能。
 - ④、可使用以下函数来设置键压缩属性

```
void QWidget::setAttribute ( Qt::WidgetAttribute attribute, bool on=true);
```

- 若 on 为 true，则设置部件的属性 attribute，若为 false，则清除该属性。
- 示例：setAttribute (Qt::WA_KeyCompression , true) //启用键事件压缩。
- 其中 Qt::WidgetAttribute 是一个枚举，该枚举定义了部件的一些常用属性，其中对键事件压缩的枚举值为 Qt::WA_KeyCompression

```
bool QWidget::testAttribute(Qt::WidgetAttribute attribute) const
```

- 若设置了属性 attribute 则返回 true，否则返回 false。

- 7、Qt 使用枚举类型 Qt::Key 来描述键盘的键代码，下表为该枚举定义的部分成员(详见帮助文档)

Qt::Key 枚举的成员(无标志类型)				
枚举值	值	说明	枚举值	值
Qt::Key_A ~ Qt::Key_Z	0x41 ~ 0x5a	键盘上的 A~Z 键	Qt::Key_Enter	0x0100 0005
Qt::Key_0 ~ QtKey_9	0x30 ~ 0x39	小键盘和主键盘上的数字键	Qt::Key_Delete	0x0100 0007
Qt::Key_F1 ~ Qt::Key_F35	0x0100 0030 ~ 0x0100 0052		Qt::Key_Space 或 Qt::Key_Any	0x20
Qt::Key_Escape	0x0100 0000	Esc 键	Qt::Key_Print	0x0100 0009
Qt::Key_Tab	0x0100 0001	Tab 键	Qt::Key_Home	0x0100 0010
Qt::Key_Backspace	0x0100 0003		Qt::Key_End	0x0100 0011
Qt::Key_Control	0x0100 0021		Qt::Key_Left	0x0100 0012
Qt::Key_Meta	0x0100 0022		Qt::Key_Up	0x0100 0013
Qt::Key_Alt	0x0100 0023		Qt::Key_Right	0x0100 0014
Qt::Key_CapsLock	0x0100 0024		Qt::Key_Down	0x0100 0015
Qt::Key_NumLock	0x0100 0025		Qt::Key_PageUp	0x0100 0016
Qt::Key_ScrollLock	0x0100 0026		Qt::Key_PageDown	0x0100 0017
Qt::Key_Pause	0x0100 0008		Qt::Key_Shift	0x0100 0020
Qt::Key_unknown	0x01ff ffff			

8、下面为 QKeyEvent 类中的成员函数

①、QKeyEvent (Type type, int key, Qt::KeyboardModifiers modifiers,

const QString &text = QString(), bool autorep = false, ushort count = 1)

- type: 表示键盘事件的类型, 必须是 QEvent::KeyPress、QEvent::KeyRelease、QEvent::ShortcutOverride 之一
- key: 用于描述键代码的 Qt::Key 枚举值。
- modifiers: 表示键盘的修饰键。
- text: 表示由按键生成的 Unicode 字符。
- autorep: 是否启用自动重复
- count: 产生该事件时的按键数量。
- 说明: 这是 QKeyEvent 类的构造函数之一(共有 2 个), 本文仅列举这一个, 构造函数可以用于创建 QKeyEvent 类型的自定义事件, 构造函数的参数虽然较多, 但是, 这些参数只是为创建的事件提供了一些必要的信息而已。

示例:

```
QKeyEvent e(QEvent::KeyPress, Qt::Key_A, Qt::NoModifier ,  
            "b", false, 1 );
```

表示创建一个类型为 QEvent::KeyPress 的键盘按下事件, 该事件由按键 A 产生, 没有键盘修饰键被按下, 但是产生的字符是 "b" 而不是 "a", 该事件的重复计数被关闭, 且按键数只有 1 个。创建该自定义事件后, 可以使用 sendEvent 进行发送, 并使用部件类的 event() 函数接收并处理该事件。

②、bool isAutoRepeat() const

若事件来自自动重复键, 则返回 true, 否则表示来自初始键按下, 返回 false, 注意, 若事件是一个多键压缩事件, 则此函数可能会不确定地返回 true 或 false。

③、int key() const

返回按下或释放时来自 Qt::Key 中定义的键代码 (不区分大小写)

④、QString text() const

返回按键所对应的 Unicode 字符。注: 按下 Shift, Control, Alt 和 Meta 等修饰键时, 返回值在不同平台间有所不同, 并可能返回空字符串。

⑤、Qt::KeyboardModifiers modifiers() const /

返回按下的修饰键, 注意: 此函数并不总是可靠。

⑥、bool matches (QKeySequence::StandardKey key) const

- 若按键与在枚举中 QKeySequence::StandardKey 定义的标准 key 匹配, 则返回 true, 否则返回 false。
- 枚举 QKeySequence::StandardKey 定义了一些常用的快捷键, 比如 StandardKey::Copy 表示的“复制”的标准快捷键为 Ctrl+C, 若按下 Ctrl+C 则这时与 StandardKey::Copy 是匹配的, 返回 true。其他的还有“剪功”的快捷键为 Ctrl+X, 其枚举值为 StandardKey::Cut 等。

9、QWidget 类中与键盘事件有关的函数

①、void grabKeyboard() //捕获键盘输入。

- ②、void **releaseKeyboard()** //释放捕获的键盘输入。
- ③、static QWidget* **keyboardGrabber()**;返回正在捕获键盘输入的部件，若没有则返回 0。

作者：黄邦勇帅(原名：黄勇)

2018-3-4

十一、本小节学习了以下类及其成员函数

1、QEvent 类

- ①、**QEvent**(Type type) //构造函数，用于创建自定义事件
- ②、virtual ~**QEvent**() //析构函数
- ③、void **accept()** //接受事件
- ④、void **ignore()** //忽略事件
- ⑤、void **setAccepted**(bool a) //设置是否接受事件
- ⑥、Type **type()** const //返回事件的类型
- ⑦、static int **registerEventType**(int hint=-1);
注册并返回自定义事件类型，使用该函数的返回值可保证自定义的事件不会有重复的 Type 枚举值。
- ⑧、enum **Type**{None, ActionAdded, ... Keypress,...} //定义事件的类型，详见帮助文档

2、QObject 类

- ①、virtual bool **event**() //接收并处理事件，先于默认的事件处理函数
- ②、void **installEventFilter**(QObject *finterObj); //安装(或注册)事件过滤器
- ③、virtual bool **eventFilter**(QObject* w, QEvent *e) //用于接收并处理事件过滤器对象接收的事件。
- ④、void **removeEventFilter**(QObject* obj); //移除事件过滤器。

3、QCoreApplication 类

- ①、virtual bool **notify**(QObject* r, QEvent* e) //该函数用于分发事件
- ②、int **exec**(); //使用该函数进入事件主循环。
- ③、void **postEvent**(QObject* r, QEvent* e, int priority=Qt::NormalEventPriority); //用于发布自定义事件
- ④、bool **sendEvent**(QObject* r, QEvent *e); //用于发送自定义事件

4、QWidget 类

- ①、void **grabKeyboard**() //捕获键盘输入
- ②、void **grabMouse**() //捕获鼠标输入
- ③、void **grabMouse**(const QCursor& c); //捕获鼠标输入
- ④、QWidget* **keyboardGrabber**(); //返回捕获键盘输入的部件
- ⑤、QWidget* **mouseGrabber**(); //返回捕获鼠标输入的部件
- ⑥、void **releaseMouse**() //释放捕获的鼠标输入
- ⑦、void **releaseKeyboard**() //释放捕获的键盘输入。
- ⑧、bool **hasMouseTracking**() const; //是否启用鼠标跟踪

⑨、void **setMouseTracking**(bool enable) //设置鼠标跟踪状态

⑩、bool **underMouse**() const; //部件是否位于鼠标光标之下

11、void **setAttribute** (Qt::WidgetAttribute attribute, bool on=true); //设置是否启用键压缩。

12、bool **testAttribute**(Qt::WidgetAttribute attribute) const //返回是否启用了键压缩。

5、本小节学习到的枚举类型

①、Qt::WidgetAttribute 枚举的枚举值 WA_KeyCompression(键压缩)

②、Qt::KeyboardModifier //描述键盘修饰键(即 Alt、Ctrl、Shift 等键)

③、Qt::MouseButton //描述鼠标信息(比如按下的是鼠标左键、右键或中键等)

④、Qt::MouseEventSource //描述鼠标事件的来源(比如鼠标事件来源于真正的鼠标还是由触摸屏仿真的)

⑤、Qt::Key //描述键盘的键代码。

6、以及 QMouseEvent 和 QKeyEvent 类中

本文作者：黄邦勇帅(原名：黄勇)

本文要求读者已经非常熟悉 C++ 的语法，若读者不熟悉 C++ 语法，推荐参阅《C++ 语法详解》(作者：黄勇)一书，电子工业出版社出版。

本文讲解了 Qt 的窗口原理，以及 QWidget 类中常使用的函数和属性，本文主要讲解 Qt 的原理性问题，因此未使用 Qt 设计师，而是使用手动编写的代码

本文是 QT 的入门文章，但是学习本文之前最好还是对 Qt 的元对象系统要有所了解，因为本文未介绍元对象系统的任何内容(元对象系统详见本人所作相关文章)。

学完本文读者会对 Qt 的窗口框架知识有一个初步的认识，为后续课程打下坚实的基础。

本文使用的是 windows 10 操作系统，Qt 版本为 Qt5.8.0，Qt Creator 的版本为 Qt Creator 4.2.1
本文内容完全属于个人见解与参考文献的作者无关，限于水平有限，其中难免有误解之处，望指出更正。

声明：禁止抄袭、复印、转载本文，本文作者拥有完全版权。

主要参考文献：

- 1、Qt Creator 快速入门(第 3 版) 霍亚飞 编著 北京航空航天大学出版社 2017 年 1 月
- 2、C++ GUI Qt4 编程(第 2 版) [加拿大]Jasmin Blanchette [英] Mark Summerfield 著 闫锋欣 曾泉人 张志强 译 电子工业出版社 2008 年 8 月
- 3、Qt 5 开发及实例 陆文周 主编 电子工业出版社 2014 年 1 月
- 4、精通 Qt4 编程(第 2 版) 蔡志明 卢传富 李立夏 等编著 电子工业出版社 2011 年 2 月
- 5、Qt on Android 核心编程 安晓辉 著 电子工业出版社 2015 年 1 月
- 6、C++ Qt 设计模式(第 2 版) [美] Alan Ezust、Paul Ezust 著 闫锋欣 张学敏 张君施 等译 张德保 审 电子工业出版社 2012 年 7 月
- 7、Qt 中的 C++ 技术 张波编著 电子工业出版社 2012 年 7 月
- 8、Head First 设计模式(中文版) Eric FreeMan , Elisabeth Freeman , Kathy Sierra , Bert Bates 著 O'Reilly Taiwan 公司译 UMLChina 改编 中国电力出版社

第3章 Qt 窗口及 QWidget 类目录

[3.1 QtWidgets 模块及窗口基本概念](#)

[3.1.1 QtWidgets 模块中的类的继承图及帮助文档的使用](#)

[3.1.2 Qt 中窗口的基本概念](#)

[3.1.3 Qt 实现窗口及其部件的原理](#)

[3.1.4 部件构造函数参数 f 的取值](#)

[3.1.5 部件的删除](#)

[3.1.6 QFlags 模板类详解](#)

[3.2 QWidget 类](#)

[3.2.1 基础](#)

[3.2.2 与部件大小和位置有关的成员](#)

[3.2.3 窗口大小的限制与默认大小](#)

[3.2.4 窗口的状态（最大化最小化）](#)

[3.2.5 窗口的显示及可见性](#)

[3.2.6 标题、透明度、启用、禁用](#)

[3.2.7 窗口标志、设置其他属性](#)

[3.2.8 获取窗口部件、设置父部件](#)

[3.2.9 鼠标光标](#)

[3.2.10 其他](#)

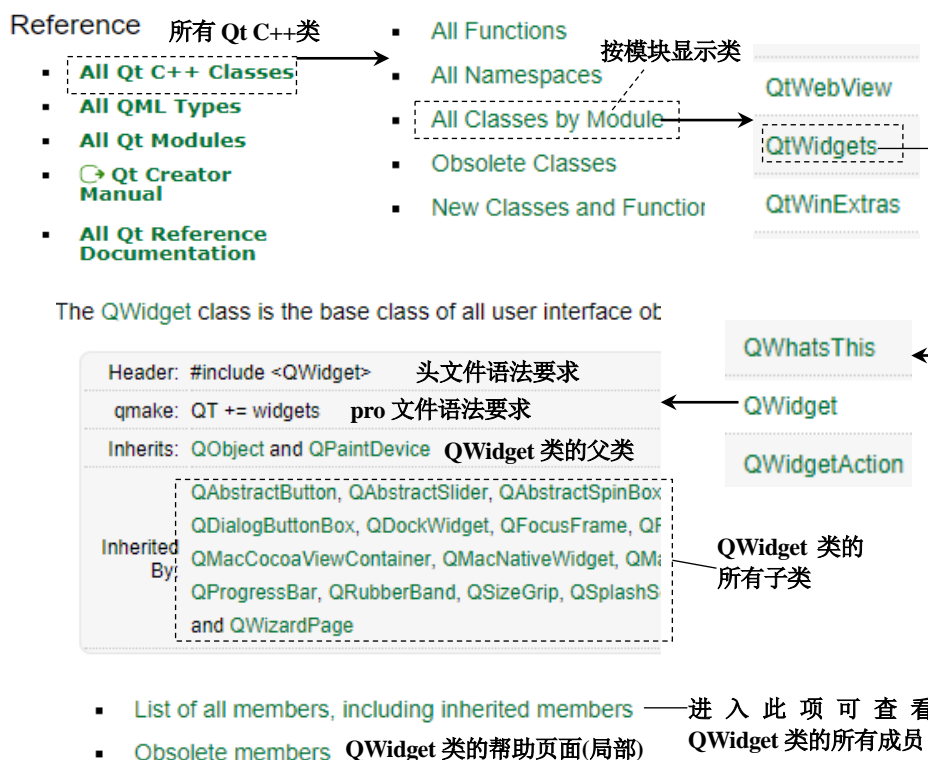
第 3 部分 Qt 窗口及 QWidget 类

注意：本教程都假设读者在 pro 文件中已添加了正确的 QT+=widgets 语句，文中不再重复累述添加此语句。

3.1 QtWidgets 模块及窗口基本概念

一、QtWidgets 模块中的类的继承图及帮助文档的使用

- 1、帮助文档的使用：找到 F:\app\Qt5.8.0\VS2015\Docs\Qt-5.8\qtdoc\目录下的 index.html 文档，使用浏览器打开，或者访问 <http://doc.qt.io> 主页(进入主页后选择相应 qt 版本的链接)，然后找到下图虚线框中的链接，便可进入 QWidget 类所在的帮助页面(注：也可在 All Qt C++ Classes 链接中通过查找到需要的类)。



- 2、若追踪帮助文档中类的继承关系，可绘制图 QtWidgets 模块中的所有类的继承关系图，图 2.1 为与 QWidget 类有关的部分类的继承图(完整的继承图，读者可自行追踪)。

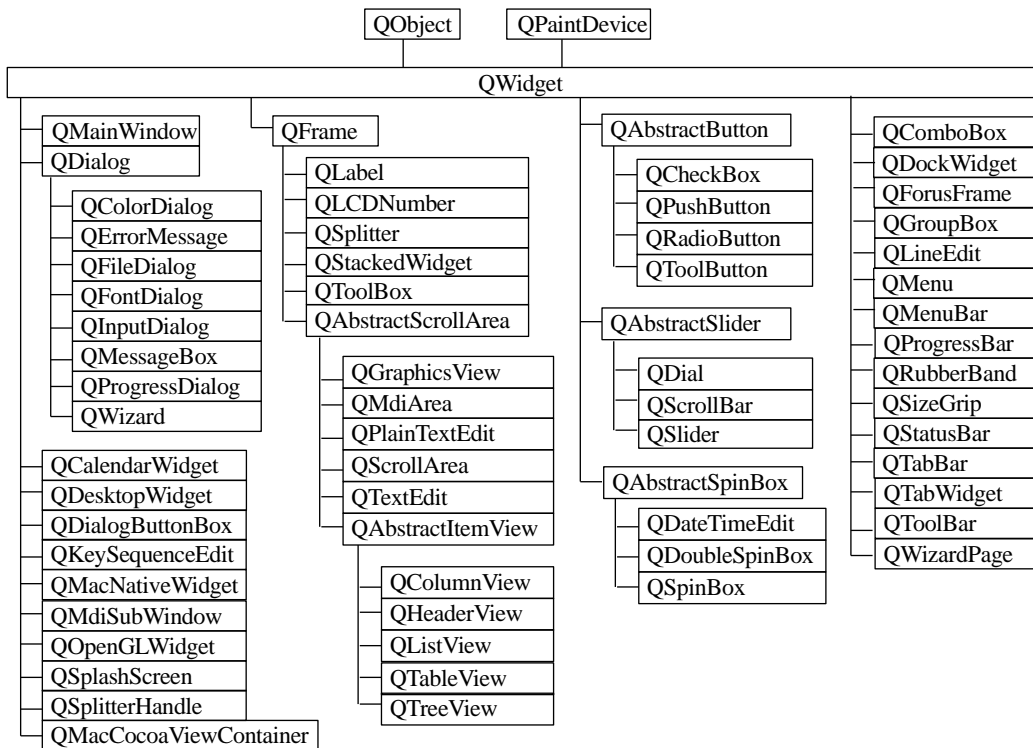


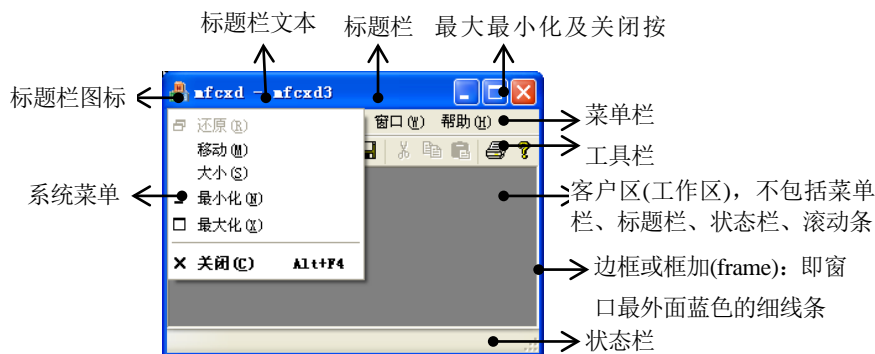
图 2.1: 与 QWidget 类的继承关系图(部分类使用字母排序)

- 3、QObject 是所有 Qt 对象的基础类，QPaintDevie 是所有可绘制对象的基础类。
- 4、QWidget 类是所有用户界面对象的基础类，QWidget 及其子类是开发桌面应用的核心，这些类都位于 QtWidgets 模块内，注意：QtWidgets 是模块，QWidget 是类(少一个字母 t 和最后的 s)，在 Qt 中很多名称都只相差一个字母，因此要注意观看。

二、Qt 中窗口的基本概念

- 1、部件或窗口部件：Qt 把建立用户界面的元素称为窗口部件(widget)，简称部件，比如：主窗口、对话框、按钮、标签等在 Qt 中都称为部件。
- 2、窗口：没有嵌入到其他部件中的部件被称为窗口，通常来说窗口是没有父部件的部件(也可以有父部件)，因此窗口又被称为顶级部件，与其相对的非窗口部件，称为子部件。窗口通常含有边框和标题栏(当然也可以没有这些)。窗口若有父部件，则在父级被删除时删除。QDialog 和 QMainWindow 部件，默认为窗口，即使在构造函数中为其指定父部件，仍是窗口。窗口通常会显示在任务栏上。
- 3、容器：放置其他部件的部件被称为容器，注意，容器可以嵌入到其他部件(即容器可以有父部件)，而窗口则不能。
- 4、窗口和窗口部件在 Qt 中是两个不同的概念，为避免引起混乱，以后把窗口部件统称为部件。

- 5、若一个部件是另一个部件的父对象，则子部件的边界会完全的位于父部件的边界内部。
- 6、Qt 中大部分部件都是用作子部件的，比如标签、按钮等。经常被用作窗口的部件是 QMainWindow 和 QDialog 的各个子类。
- 7、窗口的组成见下图



窗口的组成

三、Qt 实现窗口及其部件的原理

- 1、部件就是由 QWidget 类及其子类创建的对象，QWidget 类及其子类的功能是在 Qt 类库内部自动实现的，我们不需要关心具体是怎样实现的，只需知道如何使用这些类就行了。
- 2、因为窗口不能嵌入到其他部件，因此不是窗口的部件必有父部件，注意：具体实现时会有一些例外。
- 3、Qt 使用枚举 Qt::WindowType 对部件的类型进行描述，比如该部件是否是窗口，是否是子窗口，是否是对话框，是否拥有菜单栏等。
- 4、基于以上规则，Qt 每个部件的构造函数都会接受一个或两个标准的如下形式的参数，也就是说 QWidget 类及其子类的构造函数都会有如下形式的一个或两个形参。

①、QWidget *parent = 0

作用：该参数用于指定此部件的父部件，若为 0(默认值)，则表示此部件没有父部件，因此是一个窗口。

②、Qt::WindowFlags f=0

- 语法解析：Qt 是一个 C++ 名称空间，关于 WindowFlags 类型，后文会做讲解，此处只需知道他需要一个 Qt::WindowType 类型(注意，最后少一个 s)的枚举值，且可以使用“|”运算符进行组合取值即可。
- 作用：设置部件的标志，用于设置部件的类型及其外观。默认情况下，其值为 Qt::Widget，表示若部件没有父部件，则就是子部件，否则就是窗口。若标志取值为 Qt::Window，则此时无论该部件是否有父部件，都表示该部件是一个窗口，也就是说在这种情况下，该部件可以是一个有父部件的窗口(这是个例外)。
- 注意：Qt 的 X11 版本可能无法执行所有的标志组合，但在 Windows 上，可设置任何组合标志。

- 3、注意：若把子部件添加到已经可见的小部件中，则必须明确的显示该子部件以使其可见。

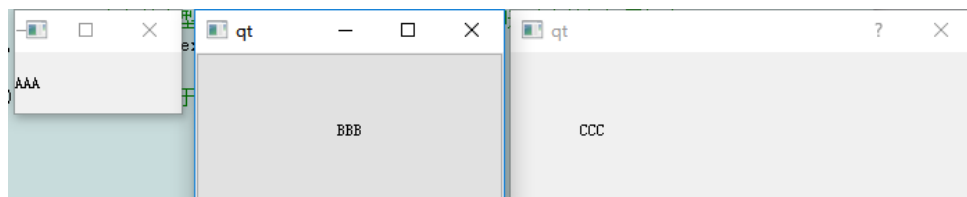
示例 1: 验证 QWidget 及其子类的构造函数参数及 Qt 窗口的原理

注：以下示例只需在项目中添加一个 C++ 源文件，并在该文件中输入以下内容即可。

```
#include<QtWidgets> /*添加此头文件会包含整个 QtWidgets 模块中的所有类。实际编写程序时像这样大型的
头文件应尽量包含可能少的数量，以加快程序的运行，本文作为讲解目的，
为了减少程序长度而使用了这种简便的方式。*/

int main(int argc, char *argv[]) {
    QApplication a(argc, argv);
    QDialog d; //窗口 1，创建一个对话框部件 d。
    d.resize(333, 222);
    QLabel t(&d, Qt::Window); /*窗口 2，验证，QWidget 类及其子类的构造函数都有 QWidget *parent=0，
    或者 Qt::WindowFlags f=0 形参。此处标志取值为 Qt::Window，则此时无论部件
    t 是否有父部件，都表示该部件是一个窗口*/
    //QLabel t1(); //注意：此语法不是使用默认构造函数创建 t1，而是在声明一个函数 t1
    t.setText("AAA"); t.resize(111, 44); t.move(50, 50);
    t.show(); //因为标签 t 现在是窗口，所以需要明确的显示。
    d.show();
    QPushButton b; /*窗口 3，默认情况下，按钮 b 没有指定父部件，因此按钮 b 是一个窗口，注意：
    QPushButton 没有参数类型为 Qt::WindowFlags 的构造函数，即无法为按钮设置标志*/
    b.resize(222, 111); b.setText("BBB"); b.show();
    QLabel t2(&d); //部件 t2 添加于在对话框 d 显示之后。
    t2.setText("CCC"); t2.move(50, 50);
    t2.show(); //因为 t2 在 d 显示之后添加的，所以必须明确的显示 t2，否则 t2 会不可见。
    return a.exec();}
```

运行结果产生如下 3 个窗口



标签 t 是窗口

按钮 b 是窗口

标签 t2 位于对话框 d 中

四、部件构造函数参数 f 的取值

1、部件构造函数的参数 f (即标志)的类型为 Qt::WindowFlags，该类型是使用 C++ 语法的 typedef 重命名后的类型，其最终结果的语法为

```
typedef QFlags<WindowType> WindowFlags
```

注意：WindowType 比 WindowFlags 少一个字符“s”，这是 Qt 的命名习惯，所以以后需要注意区分。

①、以上语句更具体的实现原理详见后文，此处先有必要了解一下其中的功能。

②、WindowType：是一个枚举类型，参数 f 的取值必须是 WindowType 的成员，该枚举类型拥有众多的成员，这些不同的成员决定了部件的不同外观和类型(具体的意义见表 2.1)，其语句原型大约如下

```
enum WindowType {Widget, Window, Dialog, Sheet, ..., WindowType_Mask}
```

③、WindowFlags：是类型 QFlags<WindowType>，这种类型是枚举类型所对应的标志，

该类型之所以能接收枚举类型 `WindowType` 的值，是因为在 `QFlags` 中有一个单形参构造函数，这里列出其简化后的形式为 `QFlags<WindowType> f{...}`，注：单形参构造函数可用于类型转换(这是 C++ 语法)

- ④、**QFlags**：是 Qt 中内置的模板类，其主要作用是枚举值及其组合运算提供类型安全的算法，这意味着，在给参数 `f` 指定值时，必须是类型正确的枚举值，否则就会发生错误。组合运算是指在枚举值之间进行按位与、或等运算。
- ⑤、标志的名称通常与对应的枚举类型仅相差一个字母 `s`，这一点要注意
- ⑥、使用标志和枚举的不同：若函数的参数是标志，则意味着实参可以是多个按位“或”的枚举类型值，若函数的参数是枚举，则只能指定该枚举类型的单个值，不能使用按位“或”运算符。

表 2.1 Qt::WindowType 枚举的取值及意义

对应标志：Qt::WindowFlags

说明：

- 1、Qt::WindowType 枚举用于描述窗口的类型。
- 2、以下枚举值虽然都有对应的整数值，但不能向部件的构造函数直接传递整数值(值 0 除外)，因为整数值类型和此处要求的枚举类型 `WindowFlags` 是不同的类型(类型安全的保证是由 `QFlags` 模板类实现的)。
- 3、在 Qt Creator 的“欢迎模式”(Ctrl+1)下直接搜索“Window Flags”可找到 Qt 内置的示例程序，点击该程序可直接在 Qt Creator 中运行，示例演示了以下所有各种不同取值下窗口的外观及特性。

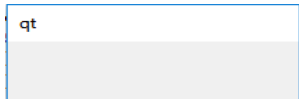
以下取值用于指定窗口类型

大部分类型都包含取值 `Qt::Window`，说明该部件是一个窗口。其中的一些值取决于底层窗口管理器是否支持

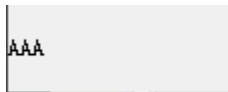
Qt::Widget(0x0000 0000)	QWidget 构造函数的默认值，表示若部件没有父部件，则就是子部件，否则就是窗口。
Qt::Window(0x0000 0001)	此时无论该部件是否有父部件，都表示该部件是一个窗口，也就是说在这种情况下，该部件可以是一个有父部件的窗口(例外情形)，通常该窗口会有标题栏、边框等
Qt::Dialog(0x0000 0002 Window)3	该部件是一个对话框，这是 QDialog 构造函数的默认值。对话框通常没有最大化和最小化按钮，这是与窗口的区别。
Qt::Sheet(0x0000 0004 Window)5	该部件是一个 Macintosh(苹果电脑)表单
Qt::Drawer(Sheet Dialog)7	该部件是一个 Macintosh 抽屉
Qt::Popup(0x0000 0008 Window)9	该部件是一个弹出式顶层窗口
Qt::Tool (Popup Dialog)11	该部件是一个工具窗口，通常是一个用于显示工具按钮的小窗口，若工具窗口有父部件，则将始终保持在父部件的最上面，否则与使用了 Qt::WindowStaysOnTopHint 提示相同。
Qt::ToolTip(Popup Sheet)13	该部件是一个提示窗口，该窗口没有标题栏和边框
Qt::SplashScreen(ToolTip Dialog)15	该部件是一个欢迎窗口，这是 QSplashScreen 构造函数的默认值
Qt::Desktop(0x0000 0010 Window)17	该部件是个桌面窗口，这是 QDesktopWidget 构造函数的默认值。
Qt::SubWindow(0x0000 0012)18	该部件是一个子窗口，比如 QMdiSubWindow 部件
Qt::CoverWindow(0x0000 0040 Window)	该部件是一个封面窗口，通常用于在程序最小化时显示该窗口。
下面的值是对窗口的外观进行的一些设置	
Qt::MSWindowsOwnDC(0x0000 0200)	为 windows 系统上的窗口添加自身的显示上下文菜单。
Qt::MSWindowsFixedSizeDialogHint (0x0000 0100)	为 windows 系统的窗口加上一个窄的边框，该值通常用于固定大小的对话框。

Qt::X11BypassWindowManagerHint(0x0000 0400)	完全忽略窗口管理器，其作用是产生一个不被管理的无窗口边框的窗口，此时用户无法使用键盘进行输入，除非手动调用函数 <code>QWidget::activeWindow()</code>
Qt::FramelessWindowHint(0x0000 0800)	产生一个无边框的窗口，此时用户无法移动窗口和改变窗口的大小。
Qt::CustomizeWindowHint(0x0200 0000)	关闭默认的窗口标题提示
Qt::WindowTitleHint(0x0000 1000)	为窗口添加一个标题栏
Qt::WindowSystemMenuHint(0x0000 2000)	为窗口添加一个系统菜单，并尽可能添加一个关闭按钮(比如，在 Mac 上)
Qt::WindowMinimizeButtonHint(0x0000 4000)	为窗口添加一个最小化按钮
Qt::WindowMaximizeButtonHint(0x0000 8000)	为窗口添加一个最大化按钮
Qt::WindowMinMaxButtonsHint(0x0000 4000 0x0000 8000)	为窗口添加一个最大化和最小化按钮
Qt::WindowCloseButtonHint(0x0800 0000)	为窗口添加一个关闭按钮
Qt::WindowContextHelpButtonHint(0x0001 0000)	为窗口添加一个帮助按钮。
Qt::WindowStaysOnTopHint(0x0004 0000)	使窗口停留在所有其他窗口之上。
Qt::WindowStaysOnBottomHint(0x0400 0000)	使窗口停留在所有其他窗口之下。
Qt::WindowTransparentForInput(0x0008 0000)	通知窗口系统，该窗口仅用于输出(即仅显示内容)不需要输入，因此当产生输入事件时，该窗口就像不存在一样，这意味着，无法用鼠标选中此窗口，当用鼠标点击该窗口时会直接选中该窗口之下的窗口。
Qt::WindowDoesNotAcceptFocus(0x0020 0000)	通知窗口系统，该窗口不接收输入焦点。
Qt::WindowType_Mask(0x0000 00ff)	用于提取窗口标志中的窗口类型部分的掩码。

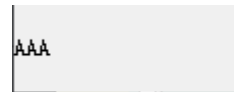
2、下图展示几个比较典型的窗口样式



Qt::CoverWidow
可移动，不能改变大小，不能关闭，也没有系统菜单



Qt::SplashScreen
不能移动，不能改变大小，不能关闭，也没有系统菜单



Qt::FramelessWindowHint
不能移动，不能改变大小，不能关闭，也没有系统菜单

五、部件的删除

部件的删除原理：Qt 会在删除父对象时自动删除其所属的所有子对象(详见元对象系统)。

六、QFlags 模板类详解

- 1、QFlags 是 Qt 中内置的模板类，其主要作用是为枚举值及其组合运算提供类型安全的算法，组合运算是指在枚举值之间进行按位与、或等运算。
- 2、QFlags 模板类的源代码位于 `qflags.h` 头文件内，路径为：
F:\app\Qt5.8.0MinGw\5.8\mingw53_32\include\QtCore
- 3、在 Qt Creator 中，要查看某个标识符的定义，可以先选中该标识符，然后右击，再选择“Follow Symbol Under Cursor”，或选中标识符后再按 F2，则就会自动跳到该标识符的定义处，比如要查看 `qflags.h` 头文件的内容，先在 Qt Creator 编辑窗口中输入 `#include <qflags.h>` 然后

选中“qflags.h”，按下 F2，此时会自动跳到该头文件内，此时在下图所示位置即可看到该头文件的具体路径



4、下面对 QFlags 的源代码作一解释，以以下代码为例

`Q_DECL_CONSTEXPR inline QFlag(int ai) Q_DECL_NOTHROW : i(ai) {}`

- `Q_DECL_CONSTEXPR`：按 F2 之后可看到，这个宏什么也不表示，因此可省略。
- `Q_DECL_NOTHROW`：按下 F2 之后可看到，这是一个与是否抛出异常有关的宏，该宏对于理解 QFlags 模板类的工作原理不是关键，因此也可忽略。
- 忽略以上两个宏和 `inline` 之后，以上代码其实就是 `QFlag(int ai):i(ai){}`
- 注意：在 Qt 中，很多标识符就只相差一个字符 s，因此要注意观看，比如在 `qflags.h` 头文件中的源码就有两个类 `QFlag` 和 `QFlags`，这是两个不同的类。

5、下面的示例程序模仿了 QFlags 模板类的实现原理，明白该例代码后，就能理解源代码的工作原理了，为简化程序，以下示例仅对 QFlags 的其中一种操作符进行了模仿，其他操作符的原理是相同的。

示例：QFlags类实现原理

```
1 #include<iostream>
2 using namespace std;
3 class B{public: /*此类专门用于类型转换，即用于把类型B转换为int和把int转换为类型B。此类相当于源代码中的QFlag，以下两个函数都是转换函数，详见(<C++语法详解>12.3节)*/
4     int i;
5     B(int f):i(f){cout<<"B"<<endl;} //6
6     operator int(){cout<<"Bi"<<endl; return i;} //8
7 template<class T> class A{ public: //模板类A，类似于源代码中的QFlags
8     int i;
9     typedef T T1;
10    A(T t):i(int(t)){cout<<"AA"<<endl;} //3
11    A(B f):i(f){cout<<"AB"<<endl;} /*类A的单形参构造函数不能为int类型，以把int类型转换为类A 类型，这样可避免直接使用整型实参调用f函数 7*/
12    A operator|(T t){cout<<"AC"<<endl; //4
13        return A(B(i|(int)t)); } //5
14    A operator|(A t) //C++语法，形参中的A t相当于A<T> t;详见《C++语法详解》15.6节
15        {cout<<"AD"<<endl;return A(B(i|t.i));}
16 };
17 #define D(t,E) typedef A<E> t; //宏，相当于源码中的宏Q_DECLARE_FLAGS(Flags,Enum)
18 /*宏P相当于源代码中的Q_DECLARE_OPERATORS_FOR_FLAGS(Flags)，该宏展开后的两个全局函数用于两枚半类型进行相加。*/
19 #define P(t) \
20 A<t::T1> operator|(t::T1 f1,t::T1 f2){cout<<"E"<<endl;return A<t::T1>(f1)|f2;}\
21 A<t::T1> operator|(t::T1 f1,A<t::T1> f2){cout<<"F"<<endl;return f2|f1;}
22 enum EE {a=1, b, c, d};
23 D(ss,EE); //展开后为: typedef A<EE> ss;
```

```

22 P(ss); /*展开宏P后，相当于在此处定义了两个对“|”重载的全局函数，展开后的最终代码如下：
        A<EE> operator|(EE f1,EE f2){cout<<"E"<<endl;return A<EE>(f1)|f2;} ②
        A<EE> operator|(EE f1,A<EE> f2){cout<<"F"<<endl;return f2|f1;}*/
23 void f(ss j){cout<<"G="<<j.i<<endl;} //⑨，类型ss保证传递给f函数的参数的类型安全性。
24 enum XX{x,y,z};
25 int main(){
26     EE me=c;
        //验证，f函数只能接受枚举类型EE的成员
27     f(a); //正确，输出G=1，a的类型为EE
28     f(a|b); //正确，输出G=3，a|b的结果为EE类型 ①
29     f(me|d); //正确，输出G=7，原因同上
30     //f(1|b); //错误。因为无法把1|b的结果转换为类型EE
31     //f(b|2); //错误。同上。
32     //f(1|2); //错误，因为1|2的结果类型不是EE
33     //f(x|y); //错误，f只能接受枚举类型EE的成员
        return 0;}

```

程序运行过程分析(C++语法本文不会讲解，语法请参阅《C++语法详解》):

1、第 21 行，宏 D 的展开比较简单，直接替换即可，略

2、第 22 行的宏 P 展开步骤为(以第 18 行代码的函数头为例进行说明):

- 首先使用 ss 替换 18 行中的 t，结果为 `A<ss::T1> operator|(ss::T1 f1,ss::T1 f2)`
- 把在 21 行使用宏 D 展开后的 ss 的原类型 `A<EE>` 替换上一步中的 ss，结果为
`A<A<EE>::T1> operator|(A<EE>::T1 f1,A<EE>::T1 f2)`
- 由其中的 `A<EE>::T1` 从第 9 行，可推得 T1 就是枚举类型 EE 的别名，因此 `A<EE>::T1` 是与类型 EE 等价的，于上把上一步中的 `A<EE>::T1` 使用 EE 替换后为

`A<EE> operator|(EE f1,EE f2)`

可见，重载的“|”操作符函数需要接收的是两个 EE 类型的形参，即对枚举类型 EE 重载了“|”操作符。

3、第 28 行 `f(a|b)` 执行步骤的分析:

首先执行 `a|b`

因为 a 和 b 的类型都为 EE，所以执行 22 行宏 P 展形后的全局重载函数，输出 E

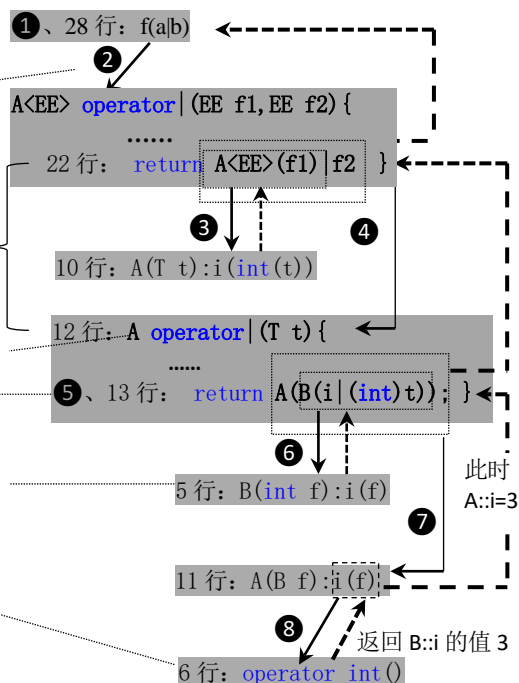
步骤③和④：因为 f1 的类型与 T 的类型相同，因此调用第 10 行的构造函数，把 f1 转换为 A<EE>类型，并返回该构造函数创建的临时对象，该对象的类型为 A<EE>，且 f2 的类型为 EE，与第 12 行函数的参数匹配，所以第 3 步返回后，接着调用第 12 行的重载函数。其间会输出 AA，而 A::i=1

输出 AC

因为 t 的值此时与 f2 相同，为 2，因此，执行 `i(int)t` 的结果为 3，

调用该函数进行类型转换，并返回创建的临时对象(类型为 B)，之后接着调用第 11 行的函数进行类型转换。此时 B::i=3，输出 B

此时 f 的类型为 B，而 i 的类型为 int，因此调用类 B 的转换函数，将其转换为 int，其间会输出 Bi，接着输出 AB



3.2 QWidget 类

注：本小节对窗口和部件不会作太详细的区分，会混用这两个概念，因为部件可以作为窗口，窗口其实也就是部件，因此窗口和部件都有相同的特性。

一、基础

- 1、QWidget 类描述了部件的基本信息，由该类创建的部件在外观上是一个最简单的部件，它看起来是一个空的窗口，QWidget 类通常用于创建一个窗口。QWidget 类非常复杂，该类包含一百多个函数，子类化 QWidget 类或其子类就可以创建自己的窗口部件。Qt 的所有 UI 元素都是 QWidget 类的子类或与 QWidget 子类一起使用。
- 2、任何没有父母的 QWidget 都将成为一个窗口，在大多数平台上都会列在桌面的任务栏中。通常应用程序只需要一个窗口即主窗口。
- 3、具有父级的 QWidget 可以通过设置 Qt:: Window 标志成为一个窗口。根据窗口管理系统的不同，这些辅助窗口通常堆叠在其各自的父窗口之上，并且没有自己的任务栏条目。
- 4、QWidget 类拥有很多的成员，该类的成员函数与几乎所有 Qt 部件都有关系，因此有些成员函数会在后续相关章节介绍，本文不作详细介绍，读者可查阅 Qt 的帮助文档。。

二、与部件大小和位置有关的成员

- 1、窗口其实就是一个矩形，因此窗口的大小(即宽和高)和位置可使用如下几种方式对其进行描述。
 - ①、使用两个点来描述窗口的大小和位置。
 - ②、使用窗口左上角的点和窗口的宽、高来描述窗口的大小和位置。
- 2、Qt 中的下列类是与描述矩形有关的(此处仅作一简介)。
 - ①、QPoint 类：该类描述一个点(精度为 int 型)，比如 QPoint p(44,55)表示创建一个坐标位于(44,55)的点 p。
 - ②、QSize 类：该类用于描述二维对象的大小，可使用该类来描述矩形的高度和宽度。
 - ③、QRect 类：该类用于描述一个矩形(精度为 int 型)，可使用该类从多个方面描述一个矩形，示例如下：
 - QRect r1(22,33, 44, 55); 表示构造一个左上角位于坐标(22,33)，宽度为 44，高度为 55 的矩形 r1，注意，QRect 直接使用 4 个数值构造的是坐标&高/宽形式的矩形，而不是使用两个点构造的矩形。
 - QRect r2(QPoint(22,33), QPoint(44,55));表示构造一个左上角位于坐标(22,33)，右下角位于坐标(44,55)的矩形 r2。
 - QRect r3(QPoint(22,33), QSize(44,55));表示构造一个左上角位于坐标(22,33)，宽度为 44，高度为 55 的矩形 r3。
- 3、Qt 依据部件的位置和大小是否包括窗口的框架(或边框)，使用不同的属性对窗口的大小和位置进行描述
- 4、QWidget 类中包括框架的属性
 - ①、x: const int 访问函数: int x() const

y: const int 访问函数: int y() const

说明: 以上两属性保存部件左上角的 x 和 y 坐标, x 和 y 默认值都为 0。

②、**frameSize:** const QSize 访问函数: QSize frameSize() const

说明: 保存部件的大小(即宽和高)

③、**frameGeometry:** const QRect 访问函数: QRect frameGeometry() const

说明: 以上属性保存部件的几何形状(即部件的左上角坐标和宽/高)。

④、**pos:** QPoint

访问函数: QPoint pos() const; void move(int x, int y); void move(const QPoint&)

说明: 可通过以上属性读取和设置部件的位置(即左上角的坐标)。

⑤、总结:

- 以上属性保存部件相对于父窗口的位臵和大小, 若部件是窗口, 则坐标值相对于桌面窗口
- 以上属性无法设置部件的大小, 仅可以使用 move()设置部件的位臵坐标, 也就是说在创建部件时, 无法为部件指定包括边框在内的大小。

5、QWidget 类中不包括框架的属性

①、**width:** const int 访问函数: int width() const

height: const int 访问函数: int height() const

说明: 以上两属性保存部件的宽(width)和高(height), 它们不支持多屏幕。

②、**rect:** const QRect 访问函数: QRect rect() const

说明: 保存部件的几何形状(仅能获取部件的宽/高), 该属性等于 QRect(0,0,width(),height());也就是说使用该属性获取的部件的位臵始终位于坐标(0,0)处。

③、**geometry:** QRect

访问函数: const QRect& geometry() const; void setGeometry(int x, int y, int w, int h);

void setGeometry(const QRect&)

说明: 该属性保存部件相对于父部件的几何形状(即部件的左上角坐标和宽/高)。

若部件的大小位于 minimumSize()和 maximumSize()之外, 则会对其进行调整。

④、**size:** QSize

访问函数: QSize size() const; void resize(int w, int h); void resize(const QSize&);

说明: 该属性保存部件的大小,

若大小超出 minimumSize()和 maximumSize()定义的范围, 则会调整其大小。

若把部件大小设置为 QSize(0,0), 则部件将不会显示在屏幕上。

⑤、总结: 以上属性仅能使用 setGeometry()和 resize()两个函数设置部件的属性, 其余函数都只能获取部件的信息, 而不能对部件的位臵或大小进行设置。

6、总结: 可使用以下两种方法设置部件的位臵和大小

①、通常使用 move()设置部件的位臵, 使用 resize()设置部件的大小。

②、使用 setGeometry()函数同时设置部件的位臵和大小。

③、无法为部件指定包含边框在内的大小, 因为无论是使用 move()还是 setGeometry()设置部件的大小, 都是不包护壁部件的边框的。

7、产生的事件

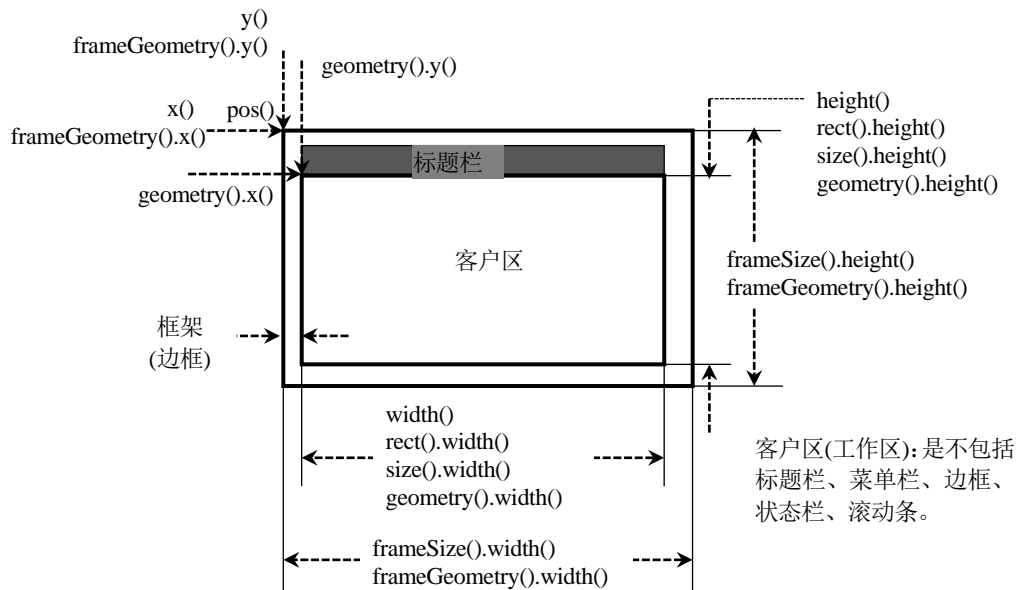
①、当位臵改变时, 若部件可见, 则会产生 moveEvent()事件, 若部件调整大小时可见, 则会产生 resizeEvent()事件, 当更改部件的几何形状, 若部件可见时, 会产生 moveEvent()或 resizeEvent()事件, 若部件不可见, 则保证在显示之前产生这些事件。

②、注意: 在 moveEvent()中调用 move()或 setGeometry 可能会产生无限递归。

在 `resizeEvent()` 或 `moveEvent()` 中调用 `setGeometry()` 函数, 可能会导致无限递归。

若在 `resizeEvent()` 中调用 `setGeometry()` 或 `resize()` 函数, 可能会导致无限递归。

8、Qt 对窗口位置和大小进行描述的属性见下图。



示例：与窗口有关的位置和大小属性

```
#include<QtWidgets>
#include <iostream>
using namespace std;
int main(int argc, char *argv[]) {
    QApplication a(argc, argv);
    QWidget w;
    w.resize(300,200); //设置部件大小(不含边框)，无法设置含有边框的部件的大小。
    w.move(40,40); //部件左上角的位置(含边框)
    //w.setGeometry(40,40,300,200); /*使用此方法设置的坐标位置是不包含边框的，读者可注释掉之后
                                   自行验证。*/

    w.show();

    cout<<"size"<<endl;
    cout<<"width"<<w.width()<<endl; //输出 300。
    cout<<"rect"<<w.rect().width()<<endl; //输出 300
    cout<<"size"<<w.size().width()<<endl; //输出 300
    cout<<"geometry"<<w.geometry().width()<<endl; //输出 300

    cout<<"*****"<<endl;
    cout<<"fgeometry"<<w.frameGeometry().width()<<endl; //输出 316，包含部件两边边框的宽度
    cout<<"fsize"<<w.frameSize().width()<<endl; //同上。

    cout<<"local"<<endl;
```



```
cout<<"x="<<w.x()<<endl; //输出 40
cout<<"pos. x="<<w.pos().x()<<endl; //输出 40
cout<<"frameGeometry. x="<<w.frameGeometry().x()<<endl; //输出 40
cout<<"rect. x="<<w.rect().x()<<endl; //输出 0, rect 属性不能获取部件的坐标位置。
cout<<"rect. y="<<w.rect().y()<<endl; //输出 0, 原因同上。
cout<<"geometry. x="<<w.geometry().x()<<endl; //输出 48, 此处的值是排除边框之后的坐标值。
return a.exec();}
```

三、窗口大小的限制与默认大小

1、QWidget 类中对部件大小限制的属性(见下表)

QWidget 类中对部件尺寸的限制(属性)		
属性名	访问函数	说明
maximumSize: QSize	QSize maximumSize() const; void setMaximumSize(const QSize&); void setMaximumSize(int, int);	以像素为单位保存部件的最大/最小大小, 不能把部件调整为比最大大小更大或比最小大小更小。若部件小于设置的最小大小, 则会自动调整为最小大小; 若部件大于最大大小, 则会自动调整为最大大小。最小大小属性的高度和宽度为 0, 表示取消该属性的限制。最大大小高度/宽度的默认值为 166777215, 最小大小高度/宽度的默认值为 0。
minimumSize: QSize	QSize minimumSize() const; void setMinimumSize(const QSize&); void setMinimumSize(int , int);	
以下属性为上面属性对应的版本		
maximumHeight : int	int maximumHeight() const;	void setMaximumHeight(int maxh);
maximumWidth: int	int maximumWidth() const;	void setMaximumWidth(int maxw);
minimumHeight: int	int minimumHeight() const;	void setMinimumHeight(int minh);
minimumWidth: int	int minimumWidth() const;	void setMinimumWidth(int minw);

2、QWidget 类中对部件大小限制的函数(见下表)

QWidget 类中对部件尺寸的限制(函数)	
函数原型	说明
void setFixedHeight (int h);	把窗口的高度设置为固定大小, 此时窗口的高度无法(比如通过鼠标)调整其大小, 仅能调整窗口的宽度, 并且窗口无法最大化。
void setFixedWidth (int w);	把窗口的宽度设置为固定大小, 此时窗口的高度无法(比如通过鼠标)调整其大小, 仅能调整窗口的高度, 并且窗口无法最大化。
void setFixedSize (const QSize &); void setFixedSize(int , int);	把窗口的高度和宽度设置为固定的大小, 此时窗口的大小无法改变(比如通过鼠标调整其大小), 且窗口不能最大化。若把高度和宽度大小固定为 QWIDGETSIZE_MAX(即 16777215)可解除这一限制。

示例：部件大小的限制

```
#include<QtWidgets>
int main(int argc, char *argv[]) {
    QApplication a(argc, argv);
```

```

QWidget w;
w.setMaximumSize(300,300);    /*设置 w 的最大大小为(300,300)，设置该值后，不能把窗口调整为比
                                (300,300)更大。*/
w.resize(500,500);    //w 的大小大于最大大小(300,200)，w 会被自动调整为最大大小(300,200)
w.move(40,40);
// w.setFixedSize(200,200);    /*固定 w 的大小为(200,200)，调用该函数后，窗口将不能调整大小，
                                读者可去掉本行注释进行验证。*/

w.show();
return a.exec();}

```

3、QWidget 类中使用以下属性保存部件的默认大小

sizeHint: const QSize 访问函数: virtual QSize sizeHint() const;

此属性保存部件的默认大小，或称为部件的大小提示、建议大小、首选大小。默认大小是指在创建部件时未设置其大小时，而使用的大小。这是个虚函数，可以重新实现该函数以指定部件的默认大小。

示例：部件的默认大小

```

#include<QtWidgets>
#include <iostream>
using namespace std;

class A:public QWidget{public:
    //重新实现 sizeHint() 函数以指定部件的默认大小
    QSize sizeHint()const{    QSize s(700,600);    return s;} };

class B:public QPushButton{public:
    B(QString s="",QWidget *p=0):QPushButton(s,p){}
    QSize sizeHint()const{    QSize s(100,50);    return s;} };

int main(int argc, char *argv[]){
    QApplication a(argc,argv);
    //以下未明确为部件指定其大小，这些部件将使用相应类中 sizeHint() 函数设置的默认大小。
    A w;    B *pb=new B("AAA",&w);    pb->move(22,22);    w.show();    return a.exec();}

```

四、窗口的状态 (最大化/最小化)

1、QWidget 类中的属性

①、maximized: const bool 访问函数: bool isMaximized() const;

部件是否最大化，该属性仅限 windows，默认值为 false

注意：该属性并不一定能获得正确的结果(比如在 X11 上)

②、minimized: const bool 访问函数: bool isMinimized() const;

部件是否最小化，该属性仅限 windows，默认值为 false

③、fullScreen: const bool: 访问函数: bool isFullScreen() const;

部件是否以全屏模式显示。全屏显示时，不会显示任何的窗口装饰(如标题栏)，这是与最大化的区别。默认值为 false。

2、QWidget 类中的槽函数(注意，以下函数是槽)

- ①、void showFullScreen() //slot
以全屏模式显示部件，此函数只对窗口有效，要返回全屏模式，需调用 showNormal()。
全屏模式在 windows 下会正常工作，但在 X 下有些问题。
- ②、void showMaximized() //slot
以最大化的形式显示部件，此函数只对窗口有效。在 X11 上可能不能正常工作。
- ③、void showMinimized() //slot
以最小化的形式显示部件，此函数只对窗口有效。
- ④、void showNormal() //slot
在最大化或最小化之后还原部件，此函数只对窗口有效。

3、QWidget 类中的函数

- ①、Qt::WindowStates windowState() const;
返回当前窗口的状态，其值为 Qt::WindowState(见后文)的枚举值的组合。
- ②、void setWindowState(Qt::WindowStates windowState);
 - 设置窗口状态，参数 windowState 可取值为 Qt::WindowState(见后文)的枚举值的组合。
 - 若窗口不可见(即 isVisible()返回 false)，则窗口状态将在调用 show()时生效，若窗口可见，更改会立即生效；
 - 调用此函数会隐藏部件，必须调用 show()使部件再次可见。
 - 注意：在某些系统中 Qt::WindowsActive 不会立即生效。
 - 当窗口状态发生变化时，会产生 QEvent::WindowStateChange 事件(处理函数为 chnageEvent())，注意：调整窗口的大小不会产生该事件

4、Qt::WindowState 枚举及其取值见下表

Qt::WindowState 枚举的成员		
标志为：Qt::WindowStates		
作用：描述顶级窗口的当前状态		
枚举成员	值	说明
Qt::WindowNoState	0x0000 0000	窗口没有设置状态(即处于正常状态)
Qt::WindowMinimized	0x0000 0001	窗口被最小化
Qt::WindowMaximized	0x0000 0002	窗口被最大化
Qt::WindowFullScreen	0x0000 0004	窗口充满整个屏幕，其周围没有任何框架(比如无标题栏)。
Qt::WindowActive	0x0000 0008	窗口是活动窗口，即该窗口具有键盘焦点

示例：窗口的状态(最大化/最小化)

```
#include<QtWidgets>
#include <iostream>
using namespace std;

class A:public QWidget{public:
    //双击窗口时使窗口最大化，再次双击则还原。注意其中的异或运算
    void mouseDoubleClickEvent(QMouseEvent *e)
        {setWindowState(windowState() ^Qt::WindowMaximized);}

    //用于验证窗口的状态发生改变时会产生该事件(注意，调整窗口大小不会产生该事件)。
```

```

        void changeEvent(QEvent* e) {          cout<<"CCC"<<endl;    }
    };
int main(int argc, char *argv[]) {
    QApplication a(argc, argv);
    A w;
    QPushButton *pb1=new QPushButton("Max", &w);
    QPushButton *pb2=new QPushButton("Min", &w);
    QPushButton *pb3=new QPushButton("Nor", &w);
    pb1->move(22, 22);    pb2->move(99, 22);    pb3->move(22, 55);

    //为按钮 pb1、pb2、pb3 的 clicked(单击) 信号分别安装槽，即当单击并释放鼠标时会调用后面的槽函数。
    QObject::connect(pb1, &QPushButton::clicked, &w, &QWidget::showMaximized);
    QObject::connect(pb2, &QPushButton::clicked, &w, &QWidget::showMinimized);
    QObject::connect(pb3, &QPushButton::clicked, &w, &QWidget::showNormal);

    w.resize(300, 200);
    w.showMaximized(); //首次显示以最大化形式显示窗口。
    return a.exec();}

```

程序结果及说明: 首次运行该程序时窗口以最大化形式显示, 点击按钮 Max 会使窗口最大化, 点点 Min 按钮会使窗口最小化, 点击按钮 Nor 窗口会正常显示, 在主窗口的空白处双击鼠标会使窗口最大化, 再次双击会使窗口正常显示。

五、窗口的显示及可见性

1、在 Qt 中部件的隐藏与可见性规则如下:

- ①、隐藏的部件意味着不可见, 部件不能同时可见和隐藏, 但不可见的部件不一定是隐藏的
- ②、隐藏的部件只有在调用 show()函数时, 才会变得可见, 显示父部件, 不会使隐藏的部件显示, 通常显示父部件时会使其子部件可见。
- ③、若祖先不可见, 则子部件不可见, 具体分为如下情形:
 - 若直到窗口的所有父部件都是可见的, 则可调用 setVisible(true)或 show()使该部件设置为可见
 - 只要祖先是不可见的, 即使使用 setVisible(true)或 show()也不能使该部件可见
- ④、使用 hide()或 setVisible(false)明确隐藏的部件永远不会可见, 即使该部件的祖先变得可见, 除非明确的显示该部件(即调用 show());
- ⑤、被其他窗口遮住的部件被认为是可见的
- ⑥、以下情形会使部件被隐藏:
 - 被创建为独立的窗口, 也就是说独立创建的窗口默认是不可见的, 需调用 show()函数进行显示。
 - 被创建为可见部件的子部件, 也就是说在创建的窗口调用 show()之后, 添加到该窗口中的子部件是不可见的, 要使此子部件可见, 需调用 show()显示。
 - 调用 hide()或 setVisible(false), 这是明确的隐藏部件

2、部件的可见性状态发生改变时会产生显示事件和隐藏事件。当用户最小化窗口时会产生隐

藏事件，再次恢复时会产生显示事件。

显示事件：QEvent::show，处理函数为 showEvent(QShowEvent*)

隐藏事件：QEvent::hide，处理函数为 hideEvent(QHideEvent*)

3、QWidget 类中与可见性有关的属性

visible: bool 访问函数: bool isVisible() const; virtual void setVisible(bool visible);

4、QWidget 类中与可见性有关的槽和函数

①、void hide() //这是槽。表示隐藏部件，此函数等效于 setVisible(false);

②、bool isHidden() const

若部件被隐藏，则返回 true，否则返回 false。isHidden()意味着!isVisible(),

③、bool isVisibleTo (const QWidget * ancestor) const

若显示祖先部件 ancestor 时，此部件变得可见，则返回 true。isVisibleTo(0)与 isVisible()相同。

示例如下：

```
QWidget w;    w.resize(300,200);
QPushButton *pb1=new QPushButton("AAA",&w);
//pb1->hide();    //显示隐藏按钮 pb1。
cout<<pb1->isVisibleTo(&w)<<endl;
w.show();    /*若 pb1->hide()被注释掉，则显示父部件 w 时，会使子部件 pb1 变为可见。上一行将输出 1，若 pb1->hide()未被注释掉，则上一行输出 0，因为此时显示父部件 w 时，子部件 pb1 仍不可见。*/
```

④、void show() //显示部件及其子部件。这是槽函数

示例：部件的可见性

```
#include<QtWidgets>
#include <iostream>
using namespace std;

int main(int argc, char *argv[]) {
    QApplication a(argc, argv);
    QWidget w;      w.resize(300, 200);
    QPushButton *pb1=new QPushButton("AAA", &w); pb1->move(22, 22);
    QPushButton *pb2=new QPushButton("BBB", &w); pb2->move(99, 22);
    cout<<w.isVisible()<<endl;    //0, 不可见
    cout<<w.isHidden()<<endl;    //1, 隐藏

    //在显示父部件之前按钮 pb1 的可见性与隐藏性;
    cout<<pb1->isVisible()<<endl;    //0, 不可见
    cout<<pb1->isHidden()<<endl;    //0, 非隐藏
    pb2->hide();      //明确隐藏按钮 pb2，此时在父部件 w 中，不会显示该按钮
    cout<<"BBB="<<pb2->isVisible()<<endl;    //0, 不可见
    cout<<"BBB="<<pb2->isHidden()<<endl;    //1, 隐藏
    w.show();    /*独立创建的窗口默认是不可见的，因此需调用 show() 显示窗口 w，调用此函数后会使得 w 的子部件(即按钮 pb1)可见，但 pb2 不可见(因为 pb2 明确调用了 hide())*/
    //显示父部件之后按钮 pb1 的可见性与隐藏性
    cout<<pb1->isVisible()<<endl;    //1, 可见
```

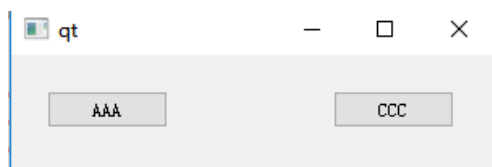
```

cout<<pb1->isHidden()<<endl; //0, 非隐藏

//在窗口 w 可见之后创建的子部件是不可见的, 要使其可见需调用 show() 函数。
QPushButton *pb3=new QPushButton("CCC",&w); pb3->move(199,22);
cout<<"CCC="<<pb3->isVisible()<<endl; //0, 不可见
cout<<"CCC="<<pb3->isHidden()<<endl; //1, 隐藏
pb3->show(); //使按钮 pb3 可见。
cout<<"lCCC="<<pb3->isVisible()<<endl; //1, 可见
cout<<"lCCC="<<pb3->isHidden()<<endl; //0, 非隐藏
return a.exec();}

```

运行结果如下图



按钮 AAA 和 CCC 都被显示出来了, 但 BBB 未被显示, 因为 BBB 明确的调用 hide()隐藏了, 且未调用 show()显示

六、标题、透明度、启用/禁用

1、QWidget 类中与标题有关的属性

①、windowTitle: QString

访问函数: QString windowTitle() const; void setWindowTitle(const QString &);

- 该属性保存窗口的标题, 仅适用于顶级窗口部件(比如窗口和对话框)。
- 若未设置标题, 则标题基于 windowFilePath。若这两个都没有设置, 则标题是一个空字符串。

②、windowFilePath: QString

访问函数: QString windowFilePath() const;

void setWindowFilePath(const QString& filePath);

- 该属性保存与部件相关联的路径,
- 该属性仅对 windows 有意义。
- 若设置了文件路径, 但未设置标题, 则将窗口标题设置为 QFileInfo::fileName()函数获取的指定路径的文件名。
- 若设置了窗口标题, 则标题优先, 且显示标题而不显示文件路径字符串。
- 若未设置路径, 则该属性是一个空字符串(默认值)

2、QWidget 类中与透明度有关的属性

windowOpacity: double

访问函数: qreal windowOpacity() const; void setWindowOpacity(qreal level);

- 该属性保存窗口的不透明度级别, 其范围是 1.0(不透明)到 0.0(透明), 默认值为 1.0。
- 半透明的窗口更新和调整 大小会明显的比不透明窗口慢。
- 在 X11 上需运行 composite manager

- qreal 是 Qt 使用 typedef 定义的 double 类型，除非被配置为 float

3、QWidget 类中对部件的启用/禁用

- ①、enabled: bool 访问函数: bool isEnabled() const; void setEnabled(bool)
 - 该属性保存是否启用部件(默认为 true)，禁用的部件不会处理鼠标和键盘事件，QAbstractButton 除外。
 - 一些部件在禁用时会以不同的方式被显示(比如变灰)。
 - 禁用部件会隐式禁用其所有子部件。
 - 启用窗口小部件将启用除顶层窗口小部件或已明确禁用的所有子窗口小部件。
 - 无法在父窗口是禁用状态时，明确的启用不是窗口的子部件。
 - 禁用/启用状态发生变化时会产生 QEvent::EnabledChange 事件(处理函数为 changeEvent());
- ②、void QWidget::setDisabled (bool disable) //槽函数
若 disable 为 true 则禁用部件输入事件，否则启用输入事件。

示例：部件的标题、透明性、启用/禁用状态

//m.h 头文件内容。

```
#include<QtWidgets>
#include <iostream>
using namespace std;
```

```
class A:public QPushButton{Q_OBJECT
public:  A(QString s="",QWidget *p=0):QPushButton(s,p) {}
void mousePressEvent(QMouseEvent *e) {
    emit clicked(); //该信号是 QAbstractButton 类中内置的信号
    cout<<"XXX"<<endl; }
public slots:
    void f() {setEnabled(isEnabled()^true);} //使部件在禁用和启用间转换。
};
```

//m.cpp 源文件内容。

```
#include "m.h"
int main(int argc, char *argv[]) {
    QApplication a(argc,argv);
    QWidget w;          w.resize(300,200);
    A *p=new A("PPP",&w);      p->resize(200,100);      p->move(44,44);
    A *pb=new A("AAA",p);      A *pb1=new A("BBB",p);      pb1->move(77,0);
    A *pb2=new A("CCC",&w);      A *pb3=new A("DDD",&w);      pb3->move(77,0);
    w.setWindowFilePath("D:/aaa/fff"); //标题被设置为路径 D:/aaa 中的文件名 fff
    //w.setWindowOpacity(0.5); //使窗口成为半透明状态。去掉注释后可自行验证。
    p->setEnabled(false); //禁用按钮 p，该设置会使 p 的子部件 pb1 和 pb2 被禁用。
    pb1->setEnabled(true); //无法启用按钮 pb2，因为其父部件 p 已被禁用。
    QObject::connect(pb2,&QPushButton::clicked,pb3,&A::f);
    w.show(); return a.exec();}
```

运行结果及说明

第一次点击按钮 CCC 会使按钮 DDD 被禁用，再次点击 CCC，会启用 DDD。每次点击按钮

设置的标题为 fff

父部件 PPP 被禁用，导致其子部件 AAA 和 BBB 也被禁用，父部件被禁用后，即使明确的启用了子部件，子

七、窗口标志、设置其他属性

1、设置窗口标志(见前文)的属性及函数如下：

①、windowFlags: Qt::WindowFlags

访问函数：Qt::WindowFlags windowFlags() const;

void setWindowFlags(Qt::windowFlags type);

该属性保存窗口的标志。更改窗口标志时会调用 setParent(), 这会导致该部件被隐藏, 需调用 show()才能使该部件再次可见

②、void setWindowFlag (Qt::WindowType flag , bool on = true); //qt5.9

该函数用于设置窗口标志, 若 on 为 true 则设置标志, 否则清除标志

③、Qt::WindowType windowType () const

获取窗口的类型, 该函数与 windowFlags()相同。

2、枚举 Qt::WidgetAttribute 描述了部件的一些属性, 比如 Qt::WA_MouseTracking 表示部件是否启用了鼠标跟踪, Qt::WA_UnderMouse 表示部件位于鼠标光标之下, 该枚举中的一些其他成员在后续章节会有介绍, 更详细的内容请查询帮助文档。该属性值可使用 QWidget 类中的如下函数进行设置和查询。

①、void setAttribute(Qt::WidgetAttribute attribute, bool on=true);

若 on 为 true 则设置属性, 否则清除该属性。

②、bool testAttribute(Qt::WidgetAttribute attribute) const

若设置了属性 attribute 则返回 true, 否则返回 false。

八、获取窗口部件、设置父部件

1、QWidget * window() const

返回此部件的的下一个祖先窗口部件(即该部件所属的窗口)。若该部件是一个窗口, 则返回该部件本身。典型的用法是改变窗口标题。

2、bool isWindow () const

若部件是独立窗口则返回 true, 否则返回 false。

3、QWidget * parentWidget () const

返回此部件的父部件, 若没有父部件, 则返回 0。

4、QWidget 类中可使用以下函数设置父部件

①、void QWidget::setParent (QWidget * parent)

②、void QWidget::setParent (QWidget * parent, Qt::WindowFlags f)

- 设置该部件的父部件为 parent，并将该部件移至新的父部件的位置(0,0)处(经测试，并不会移至(0,0)处)。
- 若新的父部件是旧的父部件，则此函数不执行任何操作。
- 若新的父部件与旧的父部件位于同一窗口中，则不会更改 Tab 键的顺序或键盘焦点。
- 若新的父部件位于一个不同的窗口中，则新创建的部件及其子部件将按照以前相同的内部顺序附加到新的父部件的 tab chain (tab 链)的末尾。
- 若移动的部件中有一个具有键盘焦点，则 setParent 会调用该小部件的 clearFocus()
- 作为更改其父部件的一部分，即使该部件以前是可见的，也会变为不可见，因此需调用 show()重新使该部件再次可见。

示例：获取窗口部件、设置父部件

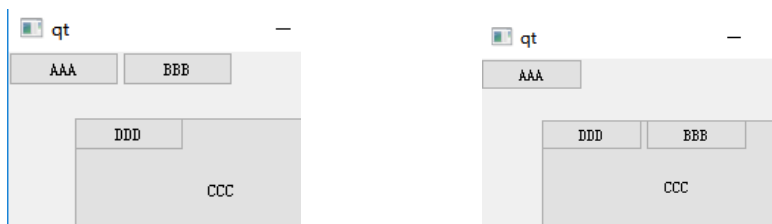
```
#include<QtWidgets>
#include <iostream>
using namespace std;

class B:public QPushButton{ public: B(QString s="",QWidget *p=0):QPushButton(s,p) {}
    void mousePressEvent(QMouseEvent *e) {
        QWidget *pw=window(); //获取部件所属的窗口
        B* s1=pw->findChild<B*>("pb1"); //获取名称为 pb1 的按钮，此按钮会被设置为 pb2 的子部件
        B* s2=pw->findChild<B*>("pb2"); //此按钮在本例会作为 pb1 的父按钮部件
        //QObject::findChild() 函数，详见元对象系统章节。
        if(objectName()=="pb") { //若按下的是按钮 pb
            s1->setParent(s2); /*把 pb2 设置为 pb1 的父部件，即 pb1 会被移至 pb2，因此 pb1 此时会被隐藏*/
            cout<<s1->objectName().toString()<<endl; //输出 pb1
            cout<<s2->objectName().toString()<<endl; //输出 pb2
        }
        s1->show(); //让按钮 pb1 再次显示出来
    }
};

int main(int argc, char *argv[]) {
    QApplication a(argc, argv);
    QWidget w; w.resize(300,200);
    B *pb=new B("AAA",&w);
    B *pb1=new B("BBB",&w); pb1->move(77,0);
    B *pb2=new B("CCC",&w); pb2->move(44,44); pb2->resize(200,100);
    B *pb3=new B("DDD",pb2);
    w.setObjectName("w"); pb->setObjectName("pb"); pb1->setObjectName("pb1");
    pb2->setObjectName("pb2"); pb3->setObjectName("pb3");
    w.show();
    QWidget* pw=pb2->window();
    QWidget* sw=pb3->parentWidget();
    cout<<pw->objectName().toString()<<endl; //输出 w，pb2 所属的窗口为 w
```

```
cout<<pb2->isWindow()<<endl; //输出 0, pb2 虽是父部件, 但不是窗口
cout<<sw->objectName().toString()<<endl; //输出 pb2
return a.exec();}
```

运行结果及说明



点击左图中的按钮 AAA 之后, 按钮 BBB 会移至右图所示位置。

九、鼠标光标

1、QWidget 类中与光标有关的属性

cursor: QCursor

访问函数: QCursor cursor() const; void setCursor(const QCursor&); void unsetCursor();

- 此属性保存部件的光标形状, 当鼠标指针位于该部件上时, 会使用此形状。
- 若未设置光标, 或调用 unsetCursor()之后, 则使用父部件的光标。
- 此属性的默认值为 Qt::ArrowCursor (即标准箭头光标)
- 即使捕获了鼠标, 某些低层窗口实现也会在光标离开部件后重置光标。若希望在窗口外光标不被重置, 可使用 QApplication::setOverrideCursor()函数。



















2、static void QApplication::setOverrideCursor(const QCursor & c);

- 设置部件的光标, 使用此方法设置的光标形状, 在鼠标离开该部件后, 只要仍在程序的所有部件中, 其形状就不会改变, 这通常用于一些长时间的操作过程中(比如沙漏形光标)。
- 使用该函数设置的光标, 会一直显示在所有程序的部件中, 直到调用 restoreOverrideCursor()或另一个 setOverrideCursor()函数。
- 使用 setOverrideCursor()函数之后, 应调用 restoreOverrideCursor()函数以取消之前的设置, 否则使用 setOverrideCursor()设置的光标不会被销毁。

3、QCursor 类及 Qt::CursorShape 枚举简介

- QCursor 类用于描述鼠标的光标及形状, 可使用该类的构造函数创建一个自定义的由位图创建的光标。
- Qt::CursorShape 是 Qt 内部自带的预定义光标形状。
- QCursor 的构造函数之一就是使用的该枚举类型的形参, 其形式为 QCursor(Qt::CursorShape shape); 注意: C++ 语法, 单形参构造函数可以进行类型转换, 因此 QWidget::setCursor(const QCursor&)函数, 可直接使用 Qt::CursorShape 枚举类型的形参值作为其参数。

- 下表为 Qt::CursorShape 枚举类中定义的预定义光标形状

Qt::CursorShape 枚举(无标志)					
枚举成员	值	形状	枚举成员	值	形状
Qt::ArrowCursor	0		Qt::PointingHandCursor	13	
Qt::UpArrowCursor	1		Qt::ForbiddenCursor	14	
Qt::CrossCursor	2		Qt::OpenHandCursor	17	
Qt::WaitCursor	3		Qt::CloseHandCursor	18	
Qt::IBeamCursor	4		Qt::WhatsThisCursor	15	
Qt::SizeVerCursor	5		Qt::BusyCursor	16	
Qt::SizeHorCursor	6		Qt::DragMoveCursor	20	拖动时常用的光标
Qt::SizeBDiagCursor	7		Qt::DragCopyCursor	19	拖动以复制时常用的光标
Qt::SizeFDiagCursor	8		Qt::DragLinkCursor	21	报动以链接时常用的光标
Qt::SizeAllCursor	9		Qt::BitmapCursor	24	
Qt::SplitVCursor	11		Qt::BlankCursor	10	空白/不可见
Qt::SplitHCursor	12				

4、bool QWidget::underMouse() const

若部件位于鼠标光标之下，则返回 true，否则返回 false，在拖放操作过程中，此值未正确更新。

示例：光标的使用

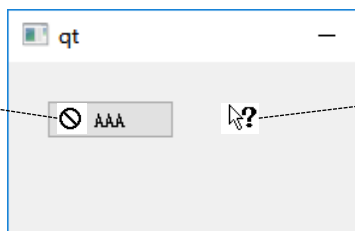
```
#include<QtWidgets>
#include <iostream>
using namespace std;

class B:public QPushButton{public:
    B(QString s="",QWidget *p=0):QPushButton(s,p){}
    void mousePressEvent(QMouseEvent *e){
        if(e->button()==Qt::LeftButton) QApplication::setOverrideCursor(Qt::WhatsThisCursor);
        if(e->button()==Qt::RightButton){
            QApplication::restoreOverrideCursor(); //取消之前使用 setOverrideCursor() 设置的光标
            setCursor(Qt::ForbiddenCursor);}}
};

int main(int argc, char *argv[]){ QApplication a(argc,argv);
    QWidget w;      B *pb=new B("AAA",&w);    pb->move(22,22);
    //pb->setCursor(Qt::WhatsThisCursor); //在此处使用此函数设置的光标会起作用。
    //QApplication::setOverrideCursor(Qt::WhatsThisCursor); //在此处使用该函数设置光标不起作用。
    w.resize(300,200);    w.show();          return a.exec();}
```

运行结果及说明

按下鼠标右键后，此形状只在按钮 AAA 中会有保持，只要鼠标移至按钮 AAA 外，就会恢复父部件的默认光标形状。



按下鼠标左键后，只要鼠标未移除窗口，则光标会一直保持此形状，移除窗口后会恢复父部件的默认光标形状。

十、其他

QWidget 类还包含与部件有关的默认事件处理函数, 比如 `keyPressEvent()`、`mousePressEvent()`、`moveEvent()`等, 这些事件处理函数都是虚拟的, 程序员可重新实现这些函数以处理对应的事件。

作者: 黄邦勇帅(原名: 黄勇)

2018-3-12

本文作者：黄邦勇帅(原名：黄勇)

本文要求读者已经非常熟悉 C++ 的语法，若读者不熟悉 C++ 语法，推荐参阅《C++ 语法详解》(作者：黄勇)一书，电子工业出版社出版。

本文是 QT 的入门文章，主要讲解了 Qt 中各种常用的部件(控件)的基本使用方法，包括，按钮、组合框(QComboBox)，行编辑器(QLineEdit)、组框(QGroupBox)、旋转框(QAbstractSpinBox)、日历(QCalendarWidget)、时间系统(QDate、QTime、QTimer)等，本文内容全面，易学易懂。

本文使用的是 windows 10 操作系统，Qt 版本为 Qt5.10.1，Qt Creator 的版本为 Qt Creator 4.5.1 本文内容完全属于个人见解与参考文献的作者无关，限于水平有限，其中难免有误解之处，望指出更正。

声明：禁止抄袭、复印、转载本文，本文作者拥有完全版权。

主要参考文献：

- 1、Qt Creator 快速入门(第 3 版) 霍亚飞 编著 北京航空航天大学出版社 2017 年 1 月
- 2、C++ GUI Qt4 编程(第 2 版) [加拿大]Jasmin Blanchette [英] Mark Summerfield 著 闫锋欣 曾泉人 张志强 译 电子工业出版社 2008 年 8 月
- 3、Qt 5 开发及实例 陆文周 主编 电子工业出版社 2014 年 1 月
- 4、精通 Qt4 编程(第 2 版) 蔡志明 卢传富 李立夏 等编著 电子工业出版社 2011 年 2 月
- 5、Qt on Android 核心编程 安晓辉 著 电子工业出版社 2015 年 1 月
- 6、C++ Qt 设计模式(第 2 版) [美] Alan Ezust、Paul Ezust 著 闫锋欣 张学敏 张君施 等译 张德保 审 电子工业出版社 2012 年 7 月
- 7、Qt 中的 C++ 技术 张波编著 电子工业出版社 2012 年 7 月
- 8、Head First 设计模式(中文版) Eric FreeMan , Elisabeth Freeman , Kathy Sierra , Bert Bates 著 O'Reilly Taiwan 公司译 UMLChina 改编 中国电力出版社

第4章 Qt 常用部件目录

[4.1 按钮部件](#)

[4.1.1 共同特性](#)

[4.1.2 QAbstractButton 抽象类](#)

[4.1.3 QPushButton 类\(标准按钮\)](#)

[4.1.4 QCheckBox 类\(复选按钮\)](#)

[4.1.5 QRadioButton 类\(单选按钮\)](#)

[4.1.6 QToolButton 类\(工具按钮\)](#)

[4.2 容器部件](#)

[4.2.1 QDialogButtonBox 按钮框](#)

[4.2.2 QButtonGroup 按钮组](#)

[4.2.3 QGroupBox 组框](#)

[4.3 带边框的部件](#)

[4.3.1 QFrame 类](#)

[4.3.2 QLabel 标签](#)

[4.3.3 QLCDNumberLCD 数字](#)

[4.4 输入部件](#)

[4.4.1 QComboBox 下拉列表、组合框](#)

[4.4.2 QLineEdit 行编辑器](#)

[4.4.3 QValidator 抽象类、验证器及其子类](#)

[4.5 旋转框、微调按钮](#)

[4.5.1 QAbstractSpinBox 旋转框或微调框](#)

[4.5.2 QSpinBox 类](#)

[4.5.3 QDoubleSpinBox 类](#)

[4.6 时间系统](#)

[4.6.1 时间系统基础](#)

[4.6.2 QDate 类](#)

[4.6.3 QTime 类](#)

[4.6.4 QDateTime 类](#)

[4.6.5 QDateTimeEdit 类](#)

[4.6.6 QDateEdit 类和 QTimeEdit 类](#)

[4.6.7 QTimer 计时器](#)

[4.6.8 QCalendarWidget 日历](#)

[部件公用枚举](#)

第 4 部分 Qt 常用部件

注意：本程序都假设读者在 pro 文件中已添加了正确的 `QT+=widgets` 语句，文中不再重复累述添加此语句。

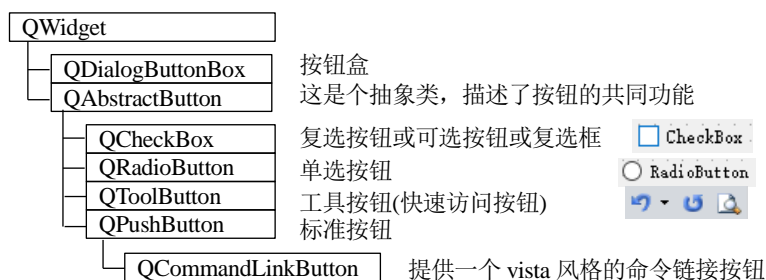
本文注重讲解原理，因此使用的是手写的 Qt 程序，对于使用 Qt 设计师快速设计 Qt 程序会在专门章节讲解。

本文所讲的部件，就是常说的“控件”，比如按钮、标签等部件。

4.1 按钮部件

一、与按钮有关的类及按钮的共同特性

1、Qt 用于描述按钮部件的类、继承关系、各按钮的名称和样式，如下图：



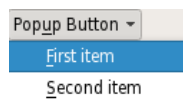
2、助记符：使用字符"&"可在为按钮指定文本标签时设置快捷键，在&之后的字符将作为快捷键。比如 "A&BC" 则 Alt+B 将成为该按钮的快捷键，使用"&"字符设置的快捷键常称为助记符，一般会在其对应字符下加上下划线，对于 windows 当按下 Alt 键时，才会显示下划线。默认情况下 Qt 不会显示加下划线的快捷键。

3、按钮的排他性(或称为独占性)和排他性组(独占性组)：是指位于同一组(被称为排他性组)中的可选按钮，任何时候只能有一个按钮被选中，选中一个按钮会自动取消之前选中的按钮。Qt 使用 `QButtonGroup` 类描述对按钮的分组，通常单选按钮具有排他性，当然复选按钮也可以具有排他性。

4 可选中、被选中、选中状态：可选中是指像复选按钮或单选按钮之类的按钮，他们是可以被选中的，被选中的按钮呈现出的状态就是选中状态。对于复选按钮，选中后在前面会有一个勾形的符号，而标准按钮 `QPushButton` 是不可被选中的。

5、按钮的按下和弹起状态：当在按钮上按下鼠标不放时，按钮通常看起来像是被按下的样子，这时的状态就是按下状态。

6、菜单按钮：是指单击该按钮会弹出一个下拉菜单的按钮，见右图，在 Qt 中，标准按钮 `QPushButton`、工具按钮 `QToolButton` 可设置为菜单按钮。



7、按钮的启用与禁用：按钮被禁用时通常会变灰，按钮的启用/禁用是由 QWidget 类中的 enabled 属性描述的。

二、QAbstractButton 抽象类

- 1、QAbstractButton 是个抽象类，该类是标准按钮 QPushButton、复选按钮 QCheckBox、单选按钮 QRadioButton、工具按钮 QToolButton 的父类，该类描述了按钮的一些共同功能，比如单击按钮发出的信号，按钮的状态等。注意：C++语法规定，不能创建抽象类的对象。
- 2、若要子类化 QAbstractButton 类，必须重新实现 paintEvent()函数(这是个纯虚函数)，以绘制按钮的轮廓或像素图，并且建议重新实现 sizeHint()，若有两个以上状态的按钮(如三态按钮)，还必须重新实现 checkStateSet()和 nextCheckState()函数。
- 3、注意：若重新实现了按钮类中的事件处理函数，有可能会阻止 Qt 对按钮的默认处理行为，比如若重写了 QCheckBox 类的 mousePressEvent()，则当在该按钮上单击鼠标时，不会使按钮被选中。因此，程序员重写这些事件处理函数时，通常需要调用父类的该函数，以便使用 Qt 的默认处理行为。
- 4、按下状态与选中状态
 - ①、Qt 在内置的默认事件处理函数 mousePressEvent()中(以鼠标点击为例)设置了按钮的按下状态，在 mouseReleaseEvent()中清除了按钮的按下状态。也就是说，若程序员仅仅重写了按钮的 mouseReleaseEvent() 函数，且在其中未清除按钮的按下状态，则当使用鼠标点击按钮后，按钮后一直处于按下状态。若重写了 mousePressEvent()函数，且未对按钮设置按下状态，则当用鼠标点击按钮时，则按钮不会出现按下状态。读者可自行编写程序进行验证。
 - ②、按下状态与选中状态的默认执行顺序是：当用鼠标点击按钮时(使用其他方式点击按钮类似)，首先设置按下状态，然后设置选中状态，若再次用鼠标点击按钮，也是首先设置按下状态，然后再清除按钮的选中状态。在每次鼠标释放时会清除按钮的按下状态。

5、QAbstractButton 类中的属性

QAbstractButton 属性速查表

属性名	说明	属性名	说明
autoExclusive	自动排他性	checked	是否被选中
autoRepeat	是否启用自动重复	down	是否处于按下状态
autoRepeatDelay	初始延迟(毫秒)	icon	按钮上显示的图标
autoRepeatInterval	时间间隔(毫秒)	iconSize	显示的图标的大小
checkable	是否可选中	shortcut	获取和设置快捷键
text	获取和设置显示的文本		

- ①、autoExclusive: bool 访问函数: bool autoExclusive() const; void setAutoExclusive(bool);
描述了按钮的自动排他性,若启用了该属性,则属于同一父部件的可选中按钮的行为,就好像是在同一排他性组中的按钮一样。除了单选按钮，默认为关闭。
- ②、autoRepeat: bool 访问函数: bool autoRepeat () const; void setAutoRepeat(bool);
描述了按钮是否启用自动重复。当按钮处于按下状态(比如按下按钮不放)时，会以固定间隔发送 pressed(), released(), clicked()信号。默认为关闭。经测试，自动重复对

默认按钮无效，也就是按下 **enter** 键时，即使关闭自动重复，默认按钮仍会重复发送上述信号。

- ③、**autoRepeatDelay**: int 访问函数: int autoRepeatDelay() const; void setAutoRepeatDelay(int);
自动重复的初始延迟(毫秒)
- ④、**autoRepeatInterval**: int 访问函数: int autoRepeatInterval() const; void setAutoRepeatInterval(int);
自动重复的时间间隔(毫秒)
- ⑤、**checkable**: bool 访问函数: bool isCheckedable() const; void setCheckable(bool);
按钮是否可选中，默认为可选中
- ⑥、**checked**: bool 访问函数: bool isChecked()const; void setChecked(bool)
按钮是否被选中(即是否处于选中状态)，只有可选中按钮才能被选中。默认未被选中。
- ⑦、**down**: bool 访问函数: bool isDown()const;void setDown(bool);
按钮是否被按下(即是否处于按下状态)。若此属性为 **true**，则按钮被按下。若把此属性设置为 **true**，则不会发送 **pressed()**和 **clicked()**信号(经测试，仍会发送这些信号)。默认为 **false**。
- ⑧、**icon**: QIcon 访问函数: QIcon icon() const; void setIcon(const QIcon &);
按钮上显示的图标，
- ⑨、**iconSize**: QSize 访问函数: QSize iconSize() const; void setIconSize(const QSize);
按钮上显示的图标的大小。默认大小由 GUI 样式定义。这是图标的最大大小，较小的图标不会被放大。
- ⑩、**shortcut** : QKeySequence
访问函数: QKeySequence shortcut() const; void setShortcut(const QKeySequence&);
保存与按钮关联的助记符，QKeySequence 类型见后文
- ⑪、**text**: QString 访问函数: QString text()const; void setText(const QString&);
按钮上显示的文本。若按钮没有文本，则 **text()**返回一个空字符串。

示例 1: 按钮的排他性与自动重复

//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>
#include <iostream>
using namespace std;
class B:public QPushButton{ Q_OBJECT
public:    B(QString s="",QWidget *p=0):QPushButton(s,p) {}
//若重写以下事件处理函数，则使用该类创建的按钮，将不会发送 Qt 内置的信号(比如 clicked()等)
//void mousePressEvent(QMouseEvent *e){cout<<"D"<<endl;}
public slots:    //注: qt5.0 之后，可使用普通函数作为槽函数。
void f(){    cout<<"F"<<endl; }
};
#endif
```

//m.cpp 文件的内容

```
#include "m.h"
int main(int argc, char *argv[]){
    QApplication a(argc,argv);
```

```

//创建部件
QWidget w;          B *pb1=new B("AAA",&w);          B *pb11=new B("AAA1",&w);
QCheckBox *pb2=new QCheckBox("BBB",&w);    QCheckBox *pb3=new QCheckBox("CCC",&w);
QCheckBox *pb4=new QCheckBox("DDD",&w);    QRadioButton *pb5=new QRadioButton("EEE",&w);
QRadioButton *pb6=new QRadioButton("FFF",&w);          //单选按钮默认具有排他性。
QCheckBox *pb7=new QCheckBox("GGG",&w);
//布局各部件
pb1->move(22,22);    pb11->move(99,22);
pb2->move(22,77);    pb3->move(99,77);    pb4->move(155,77);
pb5->move(22,122);    pb6->move(99,122);    pb7->move(155,122);
pb1->setAutoRepeat(true);          //pb1 开启自动重复
pb1->setAutoRepeatDelay(1000);    //设置初始延迟为 1 秒
pb1->setAutoRepeatInterval(2000); //设置时间间隔为 2 秒

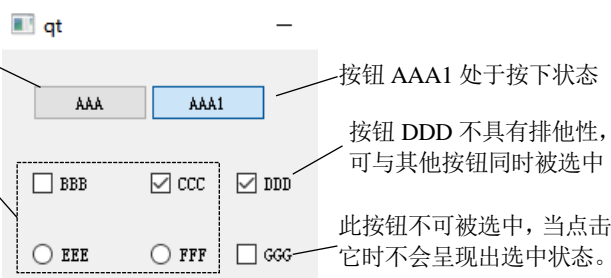
//把按钮 pb1 的 clicked 信号与槽 f 关联。
QObject::connect(pb1,&QPushButton::released,pb1,&B::f);
pb2->setAutoExclusive(true);    pb3->setAutoExclusive(true);    //开启按钮的排他性
pb11->setDown(true);          //设置为按下状态
pb7->setCheckable(false);    //设置为不可选中。
w.resize(300,200);    w.show();          return a.exec();}

```

运行结果及说明

按住按钮 AAA 不放，程序会每隔一定时间输出字符 F。

这 4 个按钮具有排他性，同时只能有一个按钮被选中。注：虚线框实际程序是没有的



5、QAbstractButton 类中的成员函数

- ①、`QAbstractButton(QWidget *parent=Q_NULLPTR);` //构造函数
- ②、`QButtonGroup* group() const` //该函数见后文。

返回此按钮所属的组，若按钮不是任何 `QButtonGroup` 的成员，则返回 0。

6、QAbstractButton 类中的信号

- ①、`void clicked(bool checked=false);`

● 以下情形会发送此信号

- 鼠标点击按钮然后释放时，注意：按钮释放时才会发送。
- 调用 `click()`或 `animateClick()`函数时。
- 按下对应的快捷键或空格键时。

- 当调用 `setDown()`、`setChecked()`或 `toggle()`函数时，不会发送该信号。
 - 若按钮是可选中的，当按钮被选中时，参数 `checked` 为 `true`，若按钮未被选中，则为 `false`。
 - 需要注意的是 `QWidget` 类并不发送此信号及 `pressed` 和 `released` 信号。
- ②、`void pressed();` 按下按钮时发送此信号
 - ③、`void released();` 释放按钮时发送此信号。
 - ④、`void toggled(bool checked);`
 - 每当可选中按钮切换状态时，都会发送此信号。若按钮被选中，则参数 `checked` 为 `true`，若按钮被取消选中，则为 `false`。
 - 按钮状态的改变可能是由于用户操作，`click()`槽函数或 `setChecked()`函数被调用的结果。
 - 在发出信号前，将更新排他性按钮组中按钮的状态，

7、QAbstractButton 类中的槽

- ①、`void animateClick(int m=100);`
执行动画单击：即，立即按下按钮，然后在 `m` 毫秒之后释放。在释放按钮之前再次调用此函数，会重新设置计时器。所有与单击有关的信号都会根据情况发出。若该按钮被禁用，则此功能不起作用。
- ②、`void click()`
此槽接收来自与点击相关的常见信号，若按钮是可选中的，则切换该按钮的状态。若该按钮被禁用，则此槽函数不起作用。
- ③、`void toggle();` 切换可选中按钮的状态。

8、QAbstractButton 类中的虚函数(受保护的)

- ①、`void checkStateSet()`
当使用 `setChecked()`时，会调用此虚函数，除非它是在 `nextCheckState()`中调用的。它允许子类重置其中间按钮状态。
- ②、`bool hitButton(const QPoint &pos) const;`
若 `pos` 位于可单击按钮的矩形内，则返回 `true`，否则返回 `false`。可单击区域默认是整个部件，子类可重新实现此函数，以提供对不同形状和大小的可单击区域的支持。
- ③、`void nextCheckState()`
当按钮被单击时，调用此虚函数，该函数允许子类实现中间按钮状态。
- ④、`void paintEvent(QPaintEvent* e)=0;` 这是个纯虚函数，子类必须重新实现该函数。

9、另外该类还重新实现了从 `QWidget` 类继承而来的一些函数，比如 `keyPressEvent()`、`mouseMoveEvent()`函数等。

示例 2：动画点击与状态切换

//m.h 文件的内容。

```
#ifndef M_H
#define M_H
#include<QtWidgets>
```

```

#include <iostream>
using namespace std;
class B:public QPushButton{ Q_OBJECT
public:    B(QString s="",QWidget *p=0):QPushButton(s,p) {}
public slots:    //注: qt5.0 之后, 可使用普通函数作为槽函数。
void f() {    cout<<"F"<<endl; }
void g() {    animateClick(4000); } //执行动画点击, 即按下按钮 4 秒之后, 按钮才会被弹起。
};
#endif

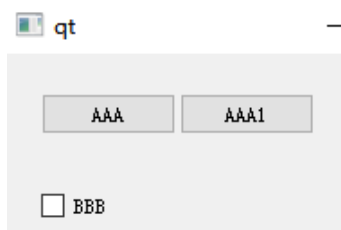
//m.cpp 文件的内容
#include "m.h"
int main(int argc, char *argv[]) {
    QApplication a(argc, argv);
    //创建部件
    QWidget w;          B *pb1=new B("AAA",&w);    B *pb11=new B("AAA1",&w);
    QCheckBox *pb2=new QCheckBox("BBB",&w);

    //布局各部件
    pb1->move(22,22);    pb11->move(99,22);    pb2->move(22,77);

    //把各按钮与相应的槽关联。
    QObject::connect(pb1,&QPushButton::clicked,pb2,&QCheckBox::toggle);
    QObject::connect(pb2,&QCheckBox::toggled,pb1,&B::f);
    QObject::connect(pb11,&QPushButton::pressed,pb1,&B::g);
    w.resize(300,200);    w.show();    return a.exec();}

```

运行结果及说明



- 当按下按钮 AAA1 时, 按钮 AAA 执行动画点击, 此时按钮被按下, 4 秒之后按钮被弹起, 此时按钮 AAA 发送 clicked 信号(注意 clicked 信号发送时机), 调用 toggle()函数, 使按钮 BBB 的状态改变, 此时按钮 BBB 发送 toggled 信号, 调用 B::f()函数输出 F。重复执行以上步骤, 按钮 BBB 的状态会在选中和未选中之间切换, 每次状态被改变都会输出 F。
直接点击按钮 AAA 也会改变按钮 BBB 的状态。

三、QPushButton 类(标准按钮)

- 1、默认按钮: 是指当用户按下 enter 时, 默认按钮会被自动按下。
- 2、自动默认按钮: 是指在满足一些条件时会自动成为默认按钮的按钮, 比如当自动默认按钮获得焦点时, 会自动成为默认按钮, 普通按钮即使获得焦点也不会成为默认按钮。
- 3、默认按钮和自动默认按钮, 在其周围通常会绘制一个额外的边框, 最多可达 3 个像素或更多, Qt 会保留这个空间在默认按钮的周围, 因此, 自动默认按钮和默认按钮可能会有更大的默认尺寸。在 Qt 中, 自动默认按钮不会显示这个边框, 但默认按钮会显示, 因此, 从外观上看, 有这个边框的按钮就是默认按钮。

4、默认按钮基本规则：

- ①、一次只能有一个按钮是默认按钮，若同时设置多个默认按钮，则最后设置的按钮是默认按钮。
 - ②、按下空格键始终单击的是具有焦点的按钮(这是按下空格键与 **enter** 键的不同)。
 - ③、应使用信号和槽，而不是事件来响应默认按钮的操作。比如对于默认按钮，应把 **clicked()**、**pressed()**或 **released()**信号与某个槽关联，以响应默认按钮按下时的操作，若使用鼠标或键盘事件，将不能达到预期的效果，因为按下 **enter** 键不是鼠标事件，而键盘事件会接收来自键盘的所有按键的事件。
 - ④、经测试，**QPushButton** 的 **autoRepeat** (自动重复)属性对默认按钮无效，也就是说即使关闭自动重复，当按下 **enter** 键时，仍会重复发送信号，注意，仅针对按下 **enter** 键，若按下的是空格键，则自动重复是有作用的。
- 5、默认按钮与自动默认按钮的行为：默认按钮的行为仅在对话框中有作用，也就是说若不是在对话框中，则不一定按以下规则执行。
- 按下 **enter** (回车)键时，若自动默认按钮当前具有焦点，则按下自动默认按钮。
 - 按下 **enter** (回车)键时，若对话框中有自动默认按钮但没有默认按钮，则按下当前具有焦点的自动默认按钮，若此时没有按钮具有焦点，则按下焦点链中的下一个自动默认按钮。

6、QPushButton 类中的属性

- ①、**autoDefault**: **bool** 访问函数: **bool autoDefault() const;** **void setAutoDefault(bool);**
若此属性为 **true**，则此按钮是一个自动默认按钮。若按钮的父部件是 **QDialog**，则此属性的默认值为 **true**，否则为 **false**。
- ②、**default**: **bool** 访问函数: **bool isDefault() const;** **void setDefault(bool);**
此属性描述了是否为默认按钮。默认为 **false**。
- ③、**flat**: **bool** 访问函数: **bool isFlat() const;** **void setFlat(bool);**
此属性描述了是否提高按钮的边框。若此属性为 **true**，大多数样式不会绘制按钮的背景，除非按下按钮。也就是说，若该属性为 **true**，在通常情况下，若不按下按钮，则该按钮看起来就像一个标签一样，是平的。**setAutoFillBackground()** 函数可确保使用 **QPalette::Button** 画刷填充背景。默认为 **false**。

7、QPushButton 类中的函数、槽

- ①、**QPushButton** (**QWidget*** parent = **Q_NULLPTR**);
QPushButton (**const QString &text** , **QWidget*** parent = **Q_NULLPTR**)
QPushButton (**const QIcon &icon** , **const QString &text** , **QWidget*** parent = **Q_NULLPTR**)
- ②、**QMenu*** **menu()** **const;**
返回与此按钮相关的弹出菜单，若未设置弹出菜单，则返回 0。菜单详见相关章节
- ③、**void setMenu** (**QMenu *** menu);
将弹出菜单 **menu** 与按钮关联，这会使按钮成为菜单按钮。
- ④、**void showMenu();** //槽
显示(弹出)与此按钮相关联的弹出菜单，若没有菜单，则什么也不做。在用户关闭弹

出菜单之前，该函数不会返回。

示例 3：默认按钮与自动默认按钮

//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>
#include <iostream>
using namespace std;

class B:public QPushButton{ Q_OBJECT
public:    B(QString s="",QWidget *p=0):QPushButton(s,p) {}
public slots:    //注：qt5.0 之后，可使用普通函数作为槽函数。
void f() {    cout<<"F"<<endl; }    void g() {    cout<<"G"<<endl; }
void h() {    cout<<"H"<<endl; }    void j() {    cout<<"J"<<endl; }
void k() {    cout<<"K"<<endl; }
};
#endif
```

//m.cpp 文件的内容

```
#include "m.h"
int main(int argc, char *argv[]) {
    QApplication a(argc, argv);
    //创建部件
    //QWidget w;
    QDialog w;    //默认按钮只在对话框中有效。读者可使用 QWidget 进行验证。
    B *pb1=new B("AAA",&w);    B *pb2=new B("BBB",&w);    B *pb3=new B("CCC",&w);
    B *pb4=new B("DDD",&w);    B *pb5=new B("EEE",&w);
    QCheckBox *pb6=new QCheckBox("XXX",&w);

    pb2->setDefault(true);    //设置 pb2 为默认按钮。
    pb3->setAutoDefault(0);    //取消 pb3 的自动默认按钮属性
    pb4->setFlat(true);    //设置 pb4 的 flat 属性为 true。
    // pb5->setAutoDefault(0);    //错误 pb5 不是 QPushButton，不能设置自动默认按钮

    cout<<pb1->autoDefault()<<endl;    //输出 1。对话框中的按钮默认具有自动默认按钮属性
    cout<<pb1->isDefault()<<endl;    //输出 0。

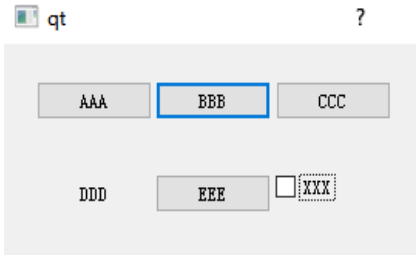
    //布局各部件
    pb1->move(22, 22);    pb2->move(99, 22);    pb3->move(177, 22);
    pb4->move(22, 77);    pb5->move(99, 77); pb6->move(177, 77);

    //验证默认按钮和自动默认按钮，需要使用信号和槽机制。
    QObject::connect(pb1,&QPushButton::clicked,pb1,&B::f);
    QObject::connect(pb2,&QPushButton::clicked,pb2,&B::g);
    QObject::connect(pb3,&QPushButton::clicked,pb3,&B::h);
    QObject::connect(pb4,&QPushButton::clicked,pb4,&B::j);
    QObject::connect(pb5,&QPushButton::clicked,pb5,&B::k);
    w.resize(300,200);    w.show();    return a.exec();}
```

运行结果及说明

BBB 是默认按钮，可明显的看到默认按钮周围有一个加宽的边框。

DDD 的 flat 属性被设置为 true，在按下该按钮之前，看起来就像标签一样，是平的。



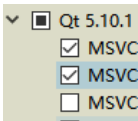
CCC 和 XXX 不具有自动默认按钮的属性。

虚线方框为表示该按钮当前具有输入焦点

- 按以下操作测试
- 1、使用键盘上的方向键，将焦点移至 XXX，因 XXX 不是自动默认按钮，因此此时 BBB 是默认按钮(会显示额外边框)，按下回车，此时输出 G，表示执行的是按下按钮 BBB 的操作。
 - 2、将焦点移至 CCC，因 CCC 不是自动默认按钮，因此此时 BBB 是默认按钮(会显示额外边框)，按下回车，此时输出 G，表示执行的是按下按钮 BBB 的操作。按下空格键，输出 H，表示执行的是按下按钮 CCC 的操作。这是按下空格与回车键的区别。
 - 3、将焦点移至 AAA，因 AAA 是自动默认按钮，因此此时 AAA 是默认按钮(此时 AAA 会显示额外边框)，按下回车，此时输出 F，表示执行的是按下按钮 AAA 的操作。

四、QCheckBox 类(复选按钮)

- 1、复选按钮的第三状态(见右图 Qt5.10.1 的选中状态)：是指除了选中 and 未选中状态之外的第三种状态，这种状态用来指示“不变”，表示用户既不选中也不取消选中该复选按钮。可使用 setTristate()函数启用这种状态，使用 checkState()检查按钮的当前状态。



- 2、QCheckBox 类的成员比较简单，如下所示

- ①、`tristate: bool` 访问函数: `bool isTristate() const; void setTristate(bool y=true);`
该属性保存复选按钮是否为三态按钮，默认为 false。注意：属性的设置函数 setTristate 只能设置此按钮具有三态按钮的形式，但不能使该按钮呈现出第三种状态，即按钮前面的小方框内不会有小黑方框的填充，但点击该按钮，会在三种状态间变换。设置按钮的状态应使用下面介绍的成员函数 setCheckState()。
- ②、`QCheckBox(QWidget* parent = Q_NULLPTR);` //构造函数
`QCheckBox(const QString &text , QWidget* parent = Q_NULLPTR);`
- ③、`Qt::CheckState checkState() const;` //返回复选框的选中状态。
- ④、`void setCheckState(Qt::CheckState state);`

设置复选按钮的状态为 state，该函数可设置复选按钮的三种状态，QAbstractButton::setChecked()只能设置两种状态。Qt::CheckState 枚举如下表

Qt::CheckState 枚举(无标志)

该枚举描述了部件的选中状态。

枚举成员	值	说明
Qt::Unchecked	0	未选中

Qt::PartiallyChecked	1	部分被选中，一个项目中的子项目被选中(但不是全部)，则该项目就是部分选中状态。
Qt::Checked	2	选中

⑤、`void stateChanged(int state);` //信号

当复选按钮的状态发生变化时发送该信号。若复选按钮具有三态形式，则按钮会在“选中、未选中、部分选中”三种状态间变化，只要这三种状态变化都会发送该信号，但 `QAbstractButton::toggled()` 信号在三种状态间变化时，则不一定会发送。

五、QRadioButton 类(单选按钮)

- 1、单选按钮默认是具有排他性的，还要注意的是单选按钮没有第三种状态。
- 2、单选按钮比较简单，其成员仅有构造函数，其余成员都是继承自父类或重新实现的父类中的函数，其构造函数如下所示

```
QRadioButton(QWidget* parent = Q_NULLPTR);
QRadioButton(const QString &text, QWidget* parent = Q_NULLPTR);
```

示例 4：复选按钮的第三状态

//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>
#include <iostream>
using namespace std;

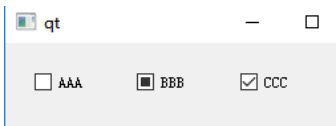
class B:public QCheckBox{ Q_OBJECT
public:    B(QString s="", QWidget *p=0):QCheckBox(s,p) {}
public slots:    //注: qt5.0 之后, 可使用普通函数作为槽函数。
    void f() {    cout<<"F"<<endl; }
    void g() {    cout<<"G"<<endl; }
};
#endif
```

//m.cpp 文件的内容

```
#include "m.h"
int main(int argc, char *argv[]) {
    QApplication a(argc, argv);
    QWidget w;    B *pb1=new B("AAA",&w);    B *pb2=new B("BBB",&w);    B *pb3=new B("CCC",&w);
    pb1->move(22,22);    pb2->move(99,22);    pb3->move(177,22);
    //注意以下两函数的区别
    pb1->setTristate(true);    //开启按钮 pb1 的第三状态
    pb2->setCheckState(Qt::PartiallyChecked);    //设置按钮 pb2 的状态为第三状态。

    //把按钮的信号与槽关联。
    QObject::connect(pb1, &QCheckBox::toggled, pb1, &B::f);
    QObject::connect(pb1, &QCheckBox::stateChanged, pb1, &B::g);
    w.resize(300,200);    w.show();    return a.exec();}
```

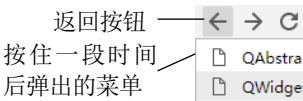

运行结果及说明



按钮 AAA 已开启第三状态，但未呈现出第三状态(即前面无黑色方块)，但点击按钮 AAA 时，会在三种状态间切换。
当点击按钮 AAA 时，其状态从部分选中状态(黑色方块)转换为选中状态(勾形符号)时，不会发送 toggled()信号，但会发送 stateChanged()信号。

六、QToolButton 类(工具按钮)

- 1、工具按钮通常不显示文本标签，仅仅是一个图标。
- 2、工具按钮通常还具有自动提升的样式，即当鼠标指向该按钮时，按钮会呈现 3D 效果。当工具按钮位于 QToolBar(工具栏)中时，会自动打开该功能，可通过使用 QToolButton::setAutoRaise()进行设置。
- 3、工具按钮还可以具有弹出式菜单，通常该菜单会延迟一段时间才会显示，最常见的是“返回”按钮，当按住“返回”按钮一段时间后，会弹出一个菜单。
- 4、工具按钮通常需要添加一个 QAction 实例，这个实例主要用于响应按下工具按钮应该响应的动作，使用该实例可使菜单和工具按钮的动作保持同步(一致)。有关该实例的具体内容请参阅相关章节。



5、QToolButton 类中的属性

- ①、**arrowType:** Qt::ArrowType
访问函数: Qt::ArrowType arrowType() const; void setArrowType(Qt::ArrowType type);
此属性描述了是否显示一个箭头作为工具按钮的图标，默认为 Qt::NoArrow

Qt::ArrowType 枚举的成员(无标志)

成员	值	说明	成员	值	说明
Qt::NoArrow	0	无箭头	Qt::LeftArrow	3	向左箭头
Qt::UpArrow	1	向上箭头	Qt::RightArrow	4	向右箭头
Qt::DownArrow	2	向下箭头			

- ②、**autoRaise:** bool
访问函数: bool autoRaise() const; void setAutoRaise(bool enable);
此属性描述了是否启用自动提升，默认为禁用(即为 false)。当使用 QMacStyle 时，此属性在 macOS 上被忽略。
- ③、**popupMode:** ToolButtonPopupMode
访问函数: ToolButtonPopupMode popupMode() const; void setPopupMode(ToolButtonPopupMode mode);
此属性描述了弹出菜单的使用方式，默认为 QToolButton::DelayedPopup

QToolButton::ToolButtonPopupMode 枚举

该枚举描述了工具按钮应如何弹出菜单

成员	值	说明
QToolButton::DelayedPopup	0	按住工具按钮一段时间后显示菜单(超时时间取决于样式, 详见 QStyle::SH_ToolButton_PopupDelay)
QToolButton::MenuButtonPopup	1	显示一个特殊的箭头, 表示该工具按钮存在菜单, 按下箭头时, 会显示菜单
QToolButton::InstantPopup	2	按下工具按钮时, 立即显示菜单。在此模式下, 按钮本身的 action 不会被触发。

④、toolButtonStyle: Qt::ToolButtonStyle

访问函数: Qt::ToolButtonStyle toolButtonStyle() const;

void setToolButtonStyle(Qt::ToolButtonStyle style); //槽

此属性描述了工具按钮图标和文本的显示方式, 即仅显示文本、图标、还是图标和文本一起显示。默认为 Qt::ToolButtonIconOnly (即仅显示图标)

QToolButton 会自动把 setToolButtonStyle()槽函数连接到 QMainWindow 中的相关信号。

Qt::ToolButtonStyle 枚举的成员(无标志)

作用: 描述按钮的文本和图标的显示方式

成员	值	说明	成员	值	说明
Qt::ToolButtonIconOnly	0	仅显示图标			
Qt::ToolButtonTextOnly	1	仅显示文本			
Qt::ToolButtonTextBesideIcon	2	文字位于图标旁			
Qt::ToolButtonTextUnderIcon	3	文字位于图标下			
Qt::ToolButtonFollowStyle	4	遵循系统设置, 在 Unix 上, 这将使用桌面环境中的用户设置, 在其他平台上, 仅表示图标。			

6、QToolButton 类中的函数

①、QToolButton(QWidget* parent = Q_NULLPTR);

构造函数, 需要注意的是, 不能通过构造函数为工具按钮设置文本和图标。

②、QAction* defaultAction() const; //返回默认 QAction

③、QMenu* menu() const; //返回与此按钮关联的菜单, 若没有菜单则返回 0。

④、void setDefaultAction(QAction* action) //槽

设置默认动作(action)为 action, 若该按钮具有默认动作, 则 action 定义按钮的属性, 比如文本、图标等。

⑤、void setMenu(QMenu* menu); 设置与该按钮相关联的菜单, 菜单的所有权不会传递到该按钮。

⑥、void showMenu() //槽

显示(弹出)与此按钮相关联的弹出式菜单, 若没有这样的菜单, 则什么也不做, 在用户关闭该菜单之前, 此函数不会返回。

⑦、void triggered(QAction * action) //信号

当触发给定的动作 action 时, 发送此信号。该操作还可以与用户界面的其他部分相关联, 比如菜单项、键盘快捷键等, 以这种方式共享动作, 可使用户界面操作更一致。

示例 5: QToolButton 工具按钮

```

#include<QtWidgets>
#include <iostream>
using namespace std;

int main(int argc, char *argv[]) {
    QApplication a(argc, argv);
    //创建部件
    QWidget w;
    QPushButton *pb1=new QPushButton(&w);    QPushButton *pb2=new QPushButton(&w);

    pb1->setText("AAA");           //设置显示的文本
    pb1->setIcon(QIcon("F:\\1.png")); //为工具按钮设置图标，图标来自于F盘的1.png文件。
    pb1->setIconSize(QSize(44,44)); //图标大小需使用 QAbstractButton::setIconSize 函数设置。
    pb1->setToolButtonStyle(Qt::ToolButtonTextBesideIcon); //工具按钮将文字显示于图标旁边。
    pb1->setAutoRaise(true);       //为该工具按钮设置自动提升属性

    pb2->setArrowType(Qt::UpArrow); //此处设置的箭头会替换掉以下设置的图标
    pb2->setIcon(QIcon("F:\\1.png"));
    pb2->setIconSize(QSize(44,44));

    pb1->move(22,22);    pb2->move(111,22);    // pb3->move(177,22);
    w.resize(300,200);    w.show();           return a.exec();}

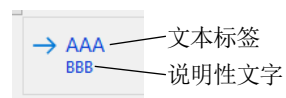
```

运行结果及说明



七、QCommandLinkButton 类(vista 风格命令链接按钮)

该按钮与标准按钮的区别是左侧有一个箭头，并且多了一行说明性的文字，其余与普通按钮类似，因此本文不在此按钮进行介绍。



4.2 容器部件

QDialogButtonBox(按钮框)、QButtonGroup(按钮组)、QGroupBox(组框)、

一、QDialogButtonBox 类(按钮框)

- 1、QDialogButtonBox 直接继承自 QWidget 类。很多程序都需要把按钮组织在一起，以呈现给用户作出一个选择，比如当关闭文件时，会弹出一个询问用户是否保存文件的对话框(通常该对话框包含 yes、no 和 cancel 按钮)等。
- 2、QDialogButtonBox 类的主要作用如下
 - ①、快速的把按钮(通常是标准按钮)组织在一起，当把按钮添加到由该类创建的按钮框之中时，按钮会被自动布局，也就是说，不需要手动设置按钮的大小和位置，QDialogButtonBox 类已经为我们做好了。
 - ②、QDialogButtonBox 类还提供了一些信号，在满足条件时，按下 QDialogButtonBox 类中的按钮会发送这些信号，但该类并未提供与这些信号相对应的槽处理函数。
- 3、向按钮框中添加按钮的方法如下：
 - ①、方法 1：在创建 QDialogButtonBox 类对象时的构造函数中直接指定由 Qt 设计好的内置的标准按钮。
 - ②、方法 2：创建自定义的按钮，然后使用 QDialogButtonBox::addButton()函数把该按钮添加到按钮框中，在添加按钮时，需要为自定义的按钮指定该按钮所要具有的 Role (作用或角色)，指定作用的主要目的是为了发送信号，当按钮具有指定的作用时，按下该按钮就会发送相对应的信号。按钮的作用是由 QDialogButtonBox 类中的 ButtonRole 枚举定义的。

4、QDialogButtonBox 类中的枚举

QDialogButtonBox::ButtonLayout 枚举(无标志)

用于描述布局按钮时需使用的布局策略，按钮布局由当前样式指定，在 X11 上，可能会受到桌面环境的影响。

成员	值	说明
QDialogButtonBox::WinLayout	0	windows 上使用的策略
QDialogButtonBox::MacLayout	1	macOs 上使用的策略
QDialogButtonBox::KdeLayout	2	KDE 上使用的策略
QDialogButtonBox::GnomeLayout	3	GNOME 上使用的策略
QDialogButtonBox::AndroidLayout	GnomeLayout+2	Android 上使用的策略

QDialogButtonBox::ButtonRole 枚举(无标志)

用于描述按钮框中按钮的角色或作用(Role)。

成员	值	说明
QDialogButtonBox::InvalidRole	-1	该按钮无效
QDialogButtonBox::AcceptRole	0	对话框被接受(如 OK 按钮)

QDialogButtonBox::RejectRole	1	对话框被拒绝(如 cancel 按钮)
QDialogButtonBox::DestructiveRole	2	点击该按钮会产生破坏性更改(如放弃当前的更改) 并关闭对话框
QDialogButtonBox::ActionRole	3	单击该按钮会导致对话框中的元素进行更改。
QDialogButtonBox::HelpRole	4	用于请求帮助(如帮助按钮)
QDialogButtonBox::YesRole	5	“是” 按钮
QDialogButtonBox::NoRole	6	“否” 按钮
QDialogButtonBox::ApplyRole	8	该按钮应用当前的更改
QDialogButtonBox::ResetRole	7	重置为默认值。

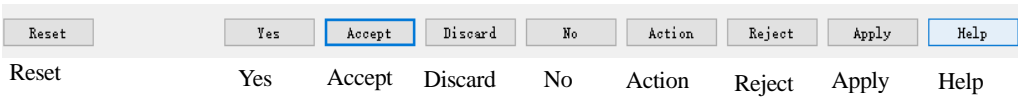
StandardButton 枚举

标志：StandardButtons

描述了 Qt 内预定义的标准按钮，这些按钮都由 ButtonRole 枚举指定了作用(Role)

成员	值	说明
QDialogButtonBox::Ok	0x0000 0400	“OK 按钮”，由 AcceptRole 定义
QDialogButtonBox::Open	0x0000 2000	“Open 按钮”，由 AcceptRole 定义
QDialogButtonBox::Save	0x0000 0800	“Save 按钮”，由 AcceptRole 定义
QDialogButtonBox::Cancel	0x0040 0000	“Cancel 按钮”，由 RejectRole 定义
QDialogButtonBox::Close	0x0020 0000	“Close 按钮”，由 RejectRole 定义
QDialogButtonBox::Discard	0x0080 0000	“Discar(丢弃)按钮”，使用 DestructiveRole 定义，取决于平台
QDialogButtonBox::Apply	0x0200 0000	“Apply 按钮”，由 ApplyRole 定义
QDialogButtonBox::Reset	0x0400 0000	“Reset 按钮”，由 ResetRole 定义
QDialogButtonBox::RestoreDefaults	0x0800 0000	“Restore Defaults 按钮”，由 ResetRole 定义
QDialogButtonBox::Help	0x0100 0000	“Help 按钮”，由 HelpRole 定义
QDialogButtonBox::SaveAll	0x0000 1000	“Save All 按钮”，由 AcceptRole 定义
QDialogButtonBox::Yes	0x0000 4000	“Yes 按钮”，由 YesRole 定义
QDialogButtonBox::YesToAll	0x0000 8000	“Yes To All 按钮”，由 YesRole 定义
QDialogButtonBox::No	0x0001 0000	“No 按钮”，由 NoRole 定义
QDialogButtonBox::NoToAll	0x0002 0000	“No To All 按钮”，由 NoRole 定义
QDialogButtonBox::Abort	0x0004 0000	“Abort 按钮”，由 RejectRole 定义
QDialogButtonBox::Retry	0x0008 0000	“Retry 按钮”，由 AcceptRole 定义
QDialogButtonBox::Ignore	0x0010 0000	“Ignore 按钮”，由 AcceptRole 定义
QDialogButtonBox::NoButton	0x0000 0000	无效按钮

5、按钮框中按钮的布局策略，按钮框中的按钮是根据按钮的 Role(作用或角色)而进行布局的，而且还会因系统的不同而不同，下图是在 windows 下水平方向的布局策略



5、QDialogButtonBox 类中的属性

- ①、`centerButtons`: bool 访问函数: bool centerButtons () const void setCenterButtons(bool)
描述按钮框中的按钮是否居中，默认为 false(即不居中)

- ②、**orientation**: Qt::Orientation
访问函数: Qt::Orientation **orientation**() const void **setOrientation**(Qt::Orientation orientation)
描述按钮框中按钮的方向，默认为水平。

Qt::Orientation 枚举

标志: Qt::Orientations

用于描述对象的方向

成员	值	说明	成员	值	说明
Qt::Horizontal	0x1	水平方向	Qt::Vertical	0x2	垂直方向

- ③、**standardButtons**: StandardButtons
访问函数: StandardButtons **standardButtons**() const
void **setstandardButtons**(StandardButtons buttons)
此属性控制按钮框使用哪些 Qt 内置的标准按钮。StandardButtons 是 QDialogButtonBox 类定义的枚举标志，可使用该属性的设置函数，以按位或的方式设置 Qt 内置的标准按钮。

6、QDialogButtonBox 类的构造函数

- ①、**QDialogButtonBox** (QWidget* parent = Q_NULLPTR);
②、**QDialogButtonBox** (Qt::Orientation orientation , QWidget* parent = Q_NULLPTR);
③、**QDialogButtonBox** (StandardButtons buttons , QWidget* parent = Q_NULLPTR);
④、**QDialogButtonBox** (StandardButtons buttons , Qt::Orientation orientation ,
QWidget* parent = Q_NULLPTR);

7、QDialogButtonBox 类中的函数

- ①、void **addButton**(QAbstractButton *button , ButtonRole role);
QPushButton* **addButton**(const QString &text , ButtonRole role);
QPushButton* **addButton**(StandardButton button);
以上函数以不同的方式向由 QDialogButtonBox 创建的按钮框中添加按钮，后面两个函数会返回相应的按钮，若指定的 role 无效，则不会把该按钮添加到按钮框中，后面两个函数会返回 0。
②、void **removeButton**(QAbstractButton* button);
从按钮框中删除 button，而不删除按钮框，并将该按钮的父部件设置为零。
③、void **clear**();
清除按钮框，即删除按钮框中的所有按钮。
④、QPushButton* **button**(StandardButton which) const;
返回与标准按钮 which 相对应的 QPushButton，若 which 不存在按钮框中，则返回 0。
⑤、QList<QAbstractButton*> **buttons**() const;
返回按钮框中所有按钮的列表。
⑥、StandardButton **standardButton**(QAbstractButton* button) const;
返回与 button 对应的 StandardButton 枚举值，若 button 不是标准按钮，则返回 NoButton。
⑦、ButtonRole **buttonRole**(QAbstractButton* button) const;
返回 button 的按钮作用(ButtonRole)，若 button 为 0 或尚未添加到按钮框中，则返回

InvalidRole。

8、QDialogButtonBox 类中的信号

①、void **clicked**(QAbstractButton *button); //信号

当点击按钮框中的按钮时发送此信号,注意:该信号与 QAbstractButton 类中的 clicked 信号的参数类型是不一样的。

②、void **rejected**(); //信号

在按钮框内用 RejectRole 或 NoRole 定义的按钮会发送此信号。

③、void **accepted**() //信号

在按钮框内用 AcceptRole 或 YesRole 定义的按钮会发送此信号。

④、void **helpRequested**(); //信号

在按钮框内用 HelpRole 定义的按钮会发送此信号。

示例 6: QDialogButtonBox 按钮框的使用

//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>
#include <iostream>
using namespace std;
class B:public QObject { Q_OBJECT
    public slots:          //注: qt5.0 之后, 可使用普通函数作为槽函数。
    void f() { cout<<"No"<<endl;
               exit(1);      } //结束程序
    void g() { cout<<"Yes"<<endl;}
    void h() { cout<<"Help"<<endl;}
    void j(QAbstractButton* b) { cout<<"clicked"<<endl;}    };
#endif // M_H
```

//m.cpp 文件的内容

```
#include "m.h"
int main(int argc, char *argv[]) {    QApplication a(argc, argv);
    //QWidget w;
    B w;
    //QDialogButtonBox 继承自 QWidget, 因此其本身可以是一个窗口。
    //向按钮框中添加 Qt 内置的 Ok 和 Cancel 按钮
    QDialogButtonBox *d=new QDialogButtonBox(QDialogButtonBox::Ok| QDialogButtonBox::Cancel);
    //创建一些自定义按钮, 不需要指定父部件
    QPushButton *pb1=new QPushButton("Reset1");
    QPushButton *pb2=new QPushButton("Yes2");
    QPushButton *pb3=new QPushButton("Accept3");
    QPushButton *pb4=new QPushButton("Discard4");
    QPushButton *pb5=new QPushButton("No5");
    QPushButton *pb6=new QPushButton("Action6");
    QPushButton *pb7=new QPushButton("Reject7");
    QPushButton *pb8=new QPushButton("Apply8");
    //向按钮框中添加按钮, 按钮并不会按以下顺序添加, 而是有固定的顺序的。
    d->addButton(pb3, QDialogButtonBox::AcceptRole);
```



```

d->addButton(pb1, QDialogButtonBox::ResetRole);
d->addButton(pb2, QDialogButtonBox::YesRole);
d->addButton(pb4, QDialogButtonBox::DestructiveRole);
d->addButton(pb6, QDialogButtonBox::ActionRole);
d->addButton(pb7, QDialogButtonBox::RejectRole);
d->addButton(pb5, QDialogButtonBox::NoRole);
d->addButton(pb8, QDialogButtonBox::ApplyRole);

// 以下为 addButton 函数的另两个重载版本
d->addButton("Help9", QDialogButtonBox::HelpRole);
//d->addButton(QDialogButtonBox::Reset); //只能添加一个标准按钮, 不能使用或运算符添加多个。

//也可使用以下函数添加 Qt 内置的标准按钮, 但以下函数会清除掉使用构造函数时指定的标准按钮。
//d->setStandardButtons(QDialogButtonBox::Reset|QDialogButtonBox::YesToAll);
//为按钮框关联信号和槽
QObject::connect(d, &QDialogButtonBox::rejected, &w, &B::f);
QObject::connect(d, &QDialogButtonBox::accepted, &w, &B::g);
QObject::connect(d, &QDialogButtonBox::helpRequested, &w, &B::h);
//按钮框中的所有按钮都会发送 clicked 信号
// QObject::connect(d, SIGNAL(clicked(QAbstractButton*)), &w, SLOT(j(QAbstractButton*)));
d->resize(300,200);    d->show();    return a.exec();    }

```

运行结果及说明



点击按钮 No5、Reject7、Cancel 会发送 rejected 信号，程序会结束
 点击按钮 Yes2、Accept3、Ok 会发送 accepted 信号，程序输出 Yes
 点击按钮 Help9，会发送 helpRequested 信号，程序输出 Help
 点击其他按钮，本程序没有输出。

二、QButtonGroup 类(按钮组)

- 1、为方便讲解，本文把由 QButtonGroup 类创建的对象称为按钮组。
- 2、QButtonGroup 继承自 QObject 类，因此该类创建的不是一个窗口，使用该类创建的对象是不可见的，即从外观上无法分辨哪些按钮是一组的。该类提供了一个容器，用于组织按钮部件(即可以把按钮放于该容器内)，这样可以方便管理组中的每个按钮，通常见到的就是把单选按钮放于一组按钮组中。
- 3、默认情况下，按钮组是具有排他性的或称为是独占的，即在任何时候只能选中按钮组中的一个按钮，选中一个按钮会取消其他已选中的按钮。
- 4、按钮组中的按钮通常是可被选中的按钮，QCheckBox(复选按钮)通常用于非排他性组。
- 5、排他性按钮组的特点：
 - 在排他性按钮组中，用户不能通过点击来取消当前已经被选中的按钮，
 - 在排他性按钮组中，当前选中的按钮可使用 checkedButton() 获取。

- 创建了一个排他性按钮组，则在初始状态时应选中组中的一个按钮，否则组中的按钮在初始状态时将没有按钮被选中。
- 6、单击一个按钮，会发送 `buttonClicked()` 信号，若该按钮是独占组中的可选中按钮，则意味着，该按钮已被选中。
 - 7、`QButtonGroup` 还可在整数和按钮间进行映射，可使用 `setId()` 函数，把一个整数 `id` 分配给一个按钮，并使用 `id()` 检索它。当前被选中的按钮的 `id`，可使用 `checkedId()` 函数获取，并且有一个重载的信号 `buttonClicked()` 发出按钮的 `id`。`id-1` 是 `QButtonGroup` 保留的，意思是“没有这样的按钮”。

8、QButtonGroup 类中的属性和信号

- ①、`exclusive: bool` 访问函数: `bool exclusive() const; void setExclusive(bool);`
该属性描述按钮组是否具有排他性，默认为 `true` (即具有排他性)
- ②、`void buttonClicked(QAbstractButton* button);` //信号
`void buttonClicked(int id);` //信号
以下情形会发送以上信号：按下按钮然后释放时，键入快捷键时，调用 `QAbstractButton::Click()` 或 `QAbstractButton::animateClick()` 时。
- ③、`void buttonPressed(QAbstractButton* button);` //信号
`void buttonPressed(int id);` //信号
当按下按钮时发送以上信号
- ④、`void buttonReleased(QAbstractButton* button);` //信号
`void buttonReleased(int id);` //信号
当释放按钮时发送以上信号
- ⑤、`void buttonToggled(QAbstractButton* button, bool checked);` //信号, qt5.2
`void buttonToggled(int id, bool checked);` //信号, qt5.2
当按钮被切换时发送以上信号，若按钮被选中，则 `checked` 为 `true`，若未选中，则为 `false`。

9、QButtonGroup 类中的函数

- ①、`QButtonGroup(QObject* parent = Q_NULLPTR);` //构造函数:
- ②、`void addButton(QAbstractButton* button, int id = -1);`
将给定的按钮 `button` 添加到按钮组中。若 `id` 为 `-1`，则会自动分配 `id` 给按钮，自动分配的 `id` 从 `-2` 开始，保证为负值。若要分配自己的 `id`，请使用正值以避免冲突。
- ③、`void setId(QAbstractButton* button, int id);` 设置按钮 `button` 的 `id`，注意 `id` 不能为 `-1`。
- ④、`void removeButton(QAbstractButton* button);` 从按钮组中删除按钮 `button`
- ⑤、`int id(QAbstractButton* button) const;`
返回按钮 `button` 的 `id`，若不存在这样的按钮，则返回 `-1`。
- ⑥、`QAbstractButton* button(int id) const;` 返回指定 `id` 的按钮，若该按钮不存在，则返回 `0`。
- ⑦、`QList<QAbstractButton*> buttons() const;` 返回按钮组中按钮的列表，该列表有可能为空。
- ⑧、`QAbstractButton* checkedButton() const;`
返回按钮组中被选中的按钮，若没有按钮被选中，则为 `0`。

⑨、`int checkedId() const;`

返回被选中的按钮的 id，若没有按钮被选中，则为-1。

⑩、注意：要获取按钮所属的按钮组，需使用 `QAbstractButton::group()` 函数。

示例 7：QButtonGroup(按钮组)的使用

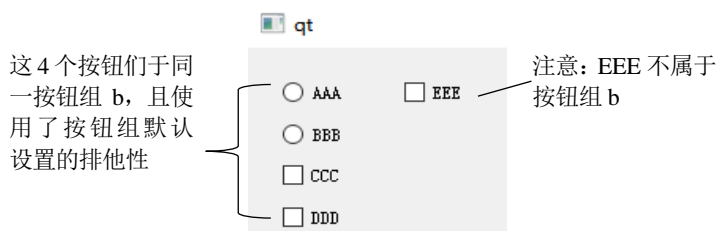
//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>
#include <iostream>
using namespace std;
class B:public QButtonGroup{    Q_OBJECT
public slots:
void f(QAbstractButton* b){
    int i=id(b); //获取按钮 b 的 id
    QAbstractButton* c=checkedButton(); //获取当前被选中的按钮
    cout<<"id="<<i<<endl;
    cout<<"button="<<c->objectName().toStdString()<<endl;
    };
};
#endif // M_H
```

//m.cpp 文件的内容

```
#include "m.h"
int main(int argc, char *argv[]){    QApplication a(argc, argv);
    QWidget w;    B *b=new B;    //创建按钮组
    QRadioButton *rb1=new QRadioButton("AAA",&w);
    QRadioButton *rb2=new QRadioButton("BBB",&w);
    QCheckBox *cb1=new QCheckBox("CCC",&w);    QCheckBox *cb2=new QCheckBox("DDD",&w);
    QCheckBox *cb3=new QCheckBox("EEE",&w);
    //设置对象名。
    rb1->setObjectName("AAA");    rb2->setObjectName("BBB");
    cb1->setObjectName("CCC");    cb2->setObjectName("DDD");    cb3->setObjectName("EEE");
    //把按钮添加到按钮组 b 之中
    b->addButton(rb1);    //未指定 id，默认为-2
    b->addButton(rb2);    //未指定 id，默认为-3
    b->addButton(cb1, 2);    //指定 id 为 2
    b->addButton(cb2);    //未指定 id，此时默认为-4
    b->setId(cb2, 3);    //把 cb2 的 id 设置为 3。
    //布局按钮
    rb1->move(22, 22);    rb2->move(22, 50);    cb1->move(22, 77);
    cb2->move(22, 105);    cb3->move(99, 22);
    //关联信号和槽
    QObject::connect(b, SIGNAL(buttonClicked(QAbstractButton*)), b, SLOT(f(QAbstractButton*)));
    w.resize(300, 200);    w.show();    return a.exec(); }
```

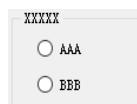
运行结果及说明



- 1、因为未对按钮组中的按钮设置初始选中按钮，所以初始状态下没有按钮被选中。
- 2、左侧的 4 个按钮任何时候只能有一个按钮被选中。
- 3、按钮 EEE 不属于按钮组 b，该按钮可与左侧的 4 个按钮同时选中。
- 4、当选中按钮组中的按钮时，不能使用点击来取消当前选中的按钮。
- 5、当点击按钮组中的按钮时，会输出该按钮的 id 和按钮名称。

三、QGroupBox 类(组框)

- 1、QGroupBox(组框)，直接继承自 QWidget 类，因此使用该对象，可作为窗口使用，组框在外观上是可见的。



注意：按钮周围的带标题 XXXXX 的细线方框才是组框

- 2、QGroupBox 类(组框)，提供了一个顶部带有标题的箱形框架，然后在该框架中可以显示其他部件，组框的主要作用是把各部件组织在一起，以方便管理。
- 3、QGroupBox 不会自动布置组框内的子部件。
- 4、组框中的子部件可以是任何部件，通常是单选按钮和复选按钮。

5、QGroupBox 类中的属性

- ①、**title**: QString 访问函数: QString title() const; void setTitle(const QString &);
此属性描述组框的标题文本。

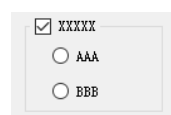
- ②、**alignment**: Qt::Alignment

访问函数: Qt::Alignment alignment()const; void setAlignment(int);

- 此属性描述组框标题文本的对齐方式，大多数样式都将标题放于组框的顶部。
- Qt::Alignment 枚举是用于描述对齐方式的，对于组框标题的水平对齐方式可取值为: Qt::AlignLeft(左对齐, 默认值)、Qt::AlignRight(右对齐)、Qt::AlignHCenter(居中对齐)，
- 注意：该枚举的设置函数 setAlignment(int)的参数是 int 型，也就是说可以直接使用 Qt::Alignment 枚举所代表的整数值来指定对齐方式。比如 setAlignment (1) ; 表示左对齐。

- ③、**checkable**: bool 访问函数: bool isCheckedable() const; void setCheckable (bool);

此属性用于描述组框的标题是否具有复选框(见右图)，即组框是否可被选中，若该属性为 true，当取消选中标题的复选框时，组框中的子部件都会被禁用。默认为 false(即不可被选中)

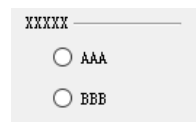


- ④、**checked**: bool 访问函数: bool isChecked() const; void setChecked (bool);
该属性描述组框是否被选中，使用此属性需要启用 checkable 属性。默认为 true。

- ⑤、**flat**: bool 访问函数: bool isFlat() const; void setFlat(bool);

此属性描述组框是否具有边框, 若此属性为 true, 则只绘制组框顶部的边框(见右图), 也就是说组框左、右和下侧的边框不会被绘制。默认为 false(禁用)。

注意: 在某些样式中, 有边框和无边框可能具有相似的形式。



6、QGroupBox 类中的构造函数和信号

- ①、**QGroupBox** (QWidget* parent = Q_NULLPTR); //构造函数
QGroupBox (const QString &title , QWidget* parent = Q_NULLPTR); //构造函数
- ②、void **clicked** (bool checked = false); //信号
发送时机同其他 clicked 信号, 但要注意: 若调用 setChecked()发不会发送该信号。
- ③、void **toggled** (bool on); //信号
若组框是可被选中的, 则在组框的状态被切换时发送该信号, 若组选被选中, 则 on 为 true, 否则为 false。

7、该类没有其他特别的成员函数。

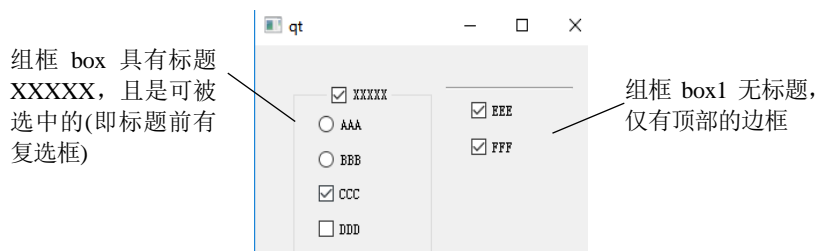
示例 8: 组框的使用

```
#include<QtWidgets>
#include <iostream>
using namespace std;
int main(int argc, char *argv[]) {                      QApplication a(argc, argv);
    QWidget w;
    QGroupBox *box=new QGroupBox(&w);                      box->move(33, 33);                      box->resize(120, 150);
    QGroupBox *box1=new QGroupBox(&w);                      box1->move(165, 33);                      box1->resize(110, 75);
    QButtonGroup *b=new QButtonGroup;                      //按钮组
    //把按钮加入到组框
    QRadioButton *rb1=new QRadioButton("AAA", box);
    QRadioButton *rb2=new QRadioButton("BBB", box);
    QCheckBox *cb1=new QCheckBox("CCC", box);                      QCheckBox *cb2=new QCheckBox("DDD", box);
    QCheckBox *cb3=new QCheckBox("EEE", box1);                      QCheckBox *cb4=new QCheckBox("FFF", box1);
    //将属于同一组的按钮加入到按钮组 b
    b->addButton(rb1);                      b->addButton(rb2);                      b->addButton(cb1, 2);                      b->addButton(cb2);
    //布局组框 box 中的子部件
    rb1->move(22, 22);                      rb2->move(22, 50);                      cb1->move(22, 77);                      cb2->move(22, 105);
    //布局组框 box1 中的子部件
    cb3->move(22, 11);                      cb4->move(22, 40);

    box->setTitle("XXXXX");                      //设置标题
    box->setAlignment(Qt::AlignHCenter);                      //居中显示标题
    box->setCheckable(true);                      //组框 box 可被选中

    box1->setFlat(true);                      //组框 box1 无边框
    w.resize(300, 200);                      w.show();                      return a.exec();                      }
```

运行结果及说明



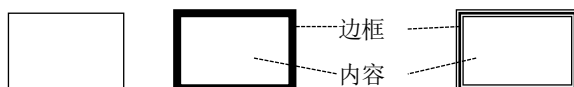
- 1、组框 XXXXX 中的子部件加入到了按钮组中，因此其中的 4 个按钮，任何时候只能有一个被选中，而组框 box1 中的两个按钮可被同时选中。
- 2、若取消选中组框 XXXXX，则该组框中的 4 个按钮都会被禁用。

4.3 带边框的部件

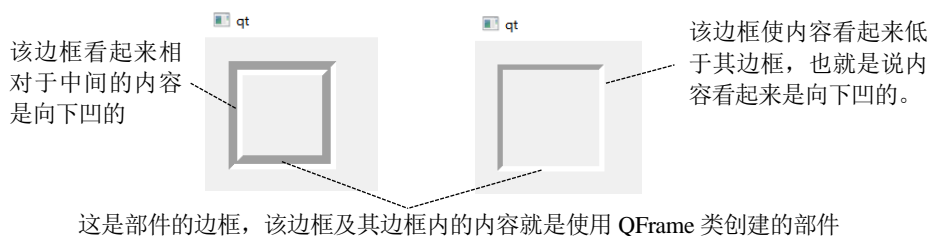
QFrame、QLabel(标签)、QLCDNumber(LCD 数字)

一、QFrame 类

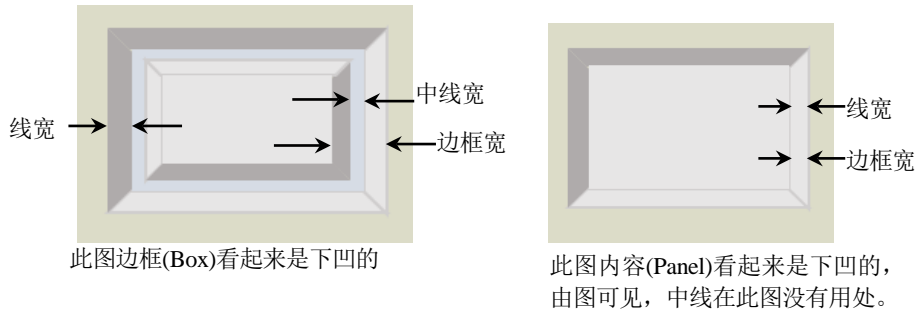
- 1、QFrame 类是带有边框的部件的基类，带边框部件的特点是有有一个明显的边框，QFrame 类就是用来实现边框的不同效果的(把这种效果称为边框样式)，所有继承自 QFrame 的子类都可以使用 QFrame 类实现的效果。
- 2、部件通常是矩形的(其他形状的原理类似)，由下图可见，矩形的边界线有粗有细，而且还可以有其他形式，而 QFrame 类，主要就是描述的类似于下图矩形的边界线的形式(或样式)，只不过在 Qt 中把这种矩形的边界线称为部件的边框。



- 3、对边框线条使用不同的颜色、粗细度、明暗度等可以使整个部件在视觉上呈现出不同的效果(见下图)，比如使部件的边框看起来相对于内容是向上凸或向下凹的，或者使内容看起来高于或底于边框。



- 4、从以上讲解可知，边框可以使部件的边框或内容看起来是向上凸或向下凹的，Qt 从以下几方面对其进行描述：
 - 使用“边框形状(frame shape)”来描述向上凸或向下凹的是边框还是内容，Qt 把此处的边框称为 Box，把此处的内容称为 Panel
 - 使用“阴影样式(shadow style)”来描述是向上凸、向下凹或是平面。Qt 使用 Raised(提升)表示向上凸，使用 Sunken 表示向下凹，使用 Plain 表示平面。
 - 线宽(lineWidth)：即边框线的宽度。(见下图)
 - 中线宽(midLineWidth)：是指边框中间的额外线的宽度(见下图)，它使用第三种颜色以获得特殊的 3D 效果，中线仅用于 Raised(提升)或 Sunken(下沉)的 Box、HLine 和 VLine 边框。
 - 边框宽(frameWidth)：指的边框的总宽度(见下图)，具体取决于使用的样式。



5、QFrame 类中的枚举

QFrame::Shadow 枚举

此枚举描述了边框 3D 效果的阴影类型(即上凸、下凹或平面)

成员	值	说明
QFrame::Plain	0x0010	边框和内容显示为平面
QFrame::Raised	0x0010	边框和内容看起来是凸起的, 使用当前颜色组的明暗色绘制 3D 凸起线。
QFrame::Sunken	0x0030	边框和内容看起来是下凹的, 使用当前颜色组的明暗色绘制 3D 下凹线。

enum QFrame::Shape

此枚举描述了边框的形状

成员	值	说明
QFrame::NoFrame	0	什么也没画(即无边框)
QFrame::Box	0x0001	QFrame 在内容周围绘制一个方框
QFrame::Panel	0x0002	QFrame 绘制一个面板, 使其内容看起来是下凹和上凸。
QFrame::StyledPanel	0x0006	绘制一个矩形面板, 其外观取决于当前的 GUI 样式, 可以下凹和上凸
QFrame::HLine	0x0004	QFrame 绘制一条无任何边框的水平线(可用作分隔符)
QFrame::VLine	0x0005	QFrame 绘制一条无任何边框的垂直线(可用作分隔符)
QFrame::WinPanel	0x0003	绘制一个 windows2000 风格的矩形面板, WinPanel 是为了兼容性而提供的, 建议使用 StyledPanel 代替

enum QFrame::StyleMask

此枚举是为 frameStyle()函数设定的, 其作用是提取该函数的返回值, 通常不需要使用该函数, 因为可使用 frameShadow()和 frameShape()函数的返回值代替 frameStyle()的返回值。

成员	值	说明
QFrame::Shadow_Mask	0x00f0	提取 frameStyle()的 Shadow 部分
QFrame::Shape_Mask	0x000f	提取 frameStyle()的 Shape 部分

6、QFrame 类中的属性

①、frameShadow: Shadow

访问函数: Shadow frameShadow() const; void setFrameShadow(Shadow);

此属性描述了边框的阴影效果(Shadow), 即向上凸、向下凹还是平面

- ②、**frameShape**: Shape 访问函数: Shape frameShape() const; void setFrameShape(Shape);

此属性描述了边框的形状(Shape), 即 Box 还是 Panel(面板)或线

- ③、**frameWidth**: const int 访问函数: int frameWidth() const;

此属性描述了边框的总宽度, 具体取决于使用的样式。

- ④、**lineWidth**: int 访问函数: int lineWidth() const; void setLineWidth(int);

此属性描述线宽, 默认值为 1。

- ⑤、**midLineWidth**: int 访问函数: int midLineWidth() const; void setMidLineWidth(int);

此属性描述中线宽度, 默认值为 0。

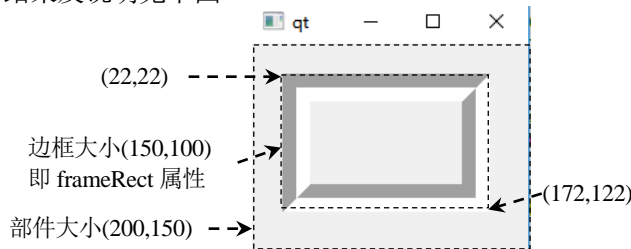
- ⑥、**frameRect**: QRect 访问函数: QRect frameRect() const; void setFrameRect(const QRect&);

该属性描述了绘制的边框矩形的大小, 默认情况下是整个部件的大小, 若将矩形设置为空矩形(比如 QRect(0,0,0,0), 则生成的矩形大小等效于该部件使用 rect()的返回值。此属性应区分部件和边框矩形是两个不同的概念, 部件是指使用 QFrame 类创建的对象, 该部件可使用 resize()函数设置其大小, 而边框矩形是位于该部件之内的, 当 QFrame 部件作为窗口时比较明显, 下面举例说明:

示例 9: QFrame 部件作为窗口时 frameRect 属性

```
#include<QtWidgets>
int main(int argc, char *argv[]) {    QApplication a(argc, argv);
    QFrame pf;    pf.resize(200, 150);
    pf.setFrameShadow(QFrame::Sunken);    pf.setFrameShape(QFrame::Box);
    pf.setLineWidth(10); //设置线宽, 若该值太小, 则 3D 效果不明显
    pf.setFrameRect(QRect(22, 22, 150, 100));
    pf.show();    return a.exec();}
```

运行结果及说明见下图

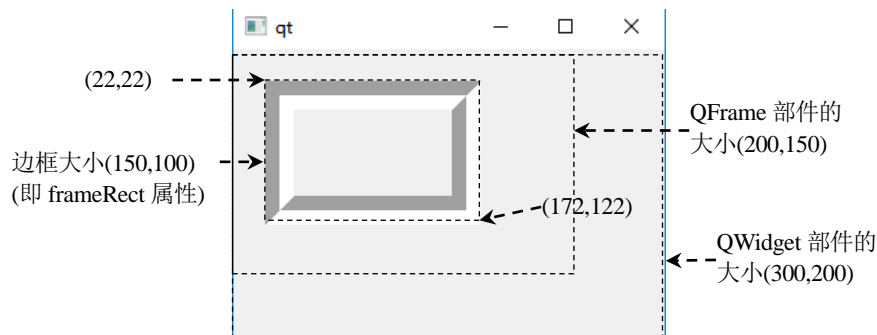


注: 图中虚线方框是为便于说明问题而添加的, 实际界面没有虚线框

示例 10: QFrame 部件作为子部件时 frameRect 属性

```
#include<QtWidgets>
int main(int argc, char *argv[]) {    QApplication a(argc, argv);
    QWidget w;    w.resize(300, 200);
    QFrame *pf=new QFrame(&w);    pf->resize(200, 150);
    pf->setFrameShadow(QFrame::Sunken);    pf->setFrameShape(QFrame::Box);
    pf->setLineWidth(10); //设置线宽, 若该值太小, 则 3D 效果不明显
    pf->setFrameRect(QRect(22, 22, 150, 100)); //QRect(0,0,0,0)是指的 pf->rect()而不是 w.rect()
    w.show();    return a.exec();}
```


运行结果及说明见下图



- 说明：
- 1、图中虚线方框是为便于说明问题而添加的，实际界面没有虚线框
 - 2、QFrame 部件占据的超出边框大小的位置在视觉上是看不见的，这对本例没有影响，但若在该区域添加了其他部件(比如按钮等)，则 QFrame 部件可能会把该按钮部件遮挡住。

7、从以上讲解可知，QFrame 的边框样式主要由该类的枚举和属性决定，下图为 QFrame 的边框样式的效果。

0				1				2				3				lineWidth()
0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	midLineWidth()
																Box, Plain
																Box, Raised
																Box, Sunken
																Panel, Plain
																Panel, Raised
																Panel, Sunken
																WinPanel, Plain
																WinPanel, Raised
																WinPanel, Sunken
																HLine, Plain
																HLine, Raised
																HLine, Sunken
																VLine, Plain
																VLine, Raised
																VLine, Sunken
																StyledPanel, Plain
																StyledPanel, Raised
																StyledPanel, Sunken

8、QFrame 类中的函数

- ①、`QFrame(QWidget* parent = Q_NULLPTR, Qt::WindowFlags f = Qt::WindowFlags());`
构造函数，默认创建的 QFrame 框架样式为 NoFrame，边框宽度为 1 像素。参数 parent 和 f 被传递给 QWidget 的构造函数。
- ②、`void setFrameStyle(int style);`
设置边框样式为 style， 参数 style 的值是枚举 QFrame::Shadow 和 QFrame::Shape 的按位或组合
- ③、`int frameStyle() const;`
返回边框的样式，默认值为 QFrame::Plain。注意，该函数返回的值是枚举 QFrame::Shadow 和 QFrame::Shape 的按位或组合,因此要获得具体的 QFrame::Shadow 或 QFrame::Shape 值，需提取该函数的返回值(见枚举 QFrame::StyleMask)。通常，该函数被 frameShadow()和 frameShape()函数代替

二、QLabel 类(标签)

- 1、QLabel 类是 QFrame 类的直接子类，因此 QLabel 可以使用从 QFrame 类继承而来的边框效果
- 2、Qt 的标签可用于显示纯文本、富文本、电影、图像等，但没有提供与用户交互的功能。
富文本就是指的符合 HTML 语言规范的文本，
- 3、QLabel 会试图猜测输入的文本是以纯文本还是以富文本的形式显示。
- 4、默认情况下，QLabel 的对齐方式是左对齐和垂直居中对齐显示，其中要显示的文本中的制表符会自动展开。
- 5、伙伴机制：是指在按下标签上的快捷键时，键盘焦点会被转移到标签的另一个部件上，这个部件就是标签的伙伴。使用"&"在标签文本中设置的助记符快捷键，只有在设置伙伴之后才会起作用，否则"&"字符会被直接显示在标签文本中。

6、QLabel 类中的属性

QLabel 类(标签)属性速查表			
属性名	说明	属性名	说明
text	标签显示的文本	scaledContents	是否缩放内容
margin	页边距	textFormat	显示富文本还是纯文本
wordWrap	是否自动换行	textInteractionFlags	设置用户是否可选择或修改文本
indent	文本缩进量	openExternalLinks	点击链接是发送 linkActivated 信号还是打开网页
alignment	文本对齐方式	hasSelectedText	是否有文本被选择
pixmap	标签的像素图	selectedText	保存所选择的文本

- ①、`text: QString` 访问函数: `QString text() const;` `void setText(const QString &);`
 - 此属性描述标签显示的文本。若未设置文本，则返回空字符串。设置文本会清除之前的内容。
 - 文本会根据 textFormat 属性的设置(默认为自动检测)，解释为纯文本或富文本。
 - 若已经设置了伙伴，则伙伴助记符会被新设置的文本更新。

②、**textFormat**: Qt::TextFormat

访问函数: Qt::TextFormat textFormat() const; void setTextFormat(Qt::TextFormat);

此属性描述标签的文本格式(即富文本还是纯文本), Qt::TextFormat 枚举成员如下:

- Qt::PlainText: 纯文本,
- Qt::RichText: 富文本。
- Qt::AutoText: 自动检测(默认值)

③、**wordWrap**: bool **访问函数**: bool wordWrap() const; void setWordWrap(bool);

此属性描述标签文本的自动换行策略, 若此属性为 true, 则标签文本在分词处会自动换行, 否则不会自动换行。默认为 false(即不自动换行)

④、**scaledContents**: bool

访问函数: bool hasScaledContents() const; void setScaledContents(bool);

此属性描述, 标签是否会缩放其内容, 以填充其可用的空间, 当启用标签显示一个像素图时, 会缩放该像素图以填充可用的空间, 也就是说此属性可缩放图像, 但不能缩放文本。默认值为 false。

⑤、**alignment**: Qt::Alignment

访问函数: Qt::Alignment alignment()const; void setAlignment(Qt::Alignment);

此属性描述标签内容的对齐方式, 默认为左对齐和垂直居中。Qt::Alignment 枚举用于描述对齐方式, 参见部件的公用枚举章节。

⑥、**indent**: int **访问函数**: int indent() const; void setIndent(int);

- 此属性描述标签的文本缩进(以像素为单位), 文本缩进与对齐方式有关, 若是左侧对齐的, 则缩进量相对于左侧边缘, 若是顶部对齐, 则缩进量相对于顶部边缘, 右对齐和底部对齐原理相同。
- 若缩进量为负, 或未设置缩进, 则按以下方式计算:
若 frameWidth()为 0, 则缩进量为 0, 若 frameWidth()大于 0, 则缩进量为部件当前 font()返回的字体的 x 字符宽度的一半。
- 默认值为-1。

⑦、**margin**: int **访问函数**: int margin() const; void setMargin(int)

此属性描述页边距的宽度, 页边距是指边框最内部像素与内容最外层像素之间的距离。默认值为 0。

⑧、**textInteractionFlags**: Qt::TextInteractionFlags

访问函数: Qt::TextInteractionFlags textInteractionFlags() const;

void setTextInteractionFlags(Qt::TextInteractionFlags);

- 此属性描述标签如何与用户输入交互,
- 若此属性被设置为 Qt::LinksAccessibleByKeyboard, 则焦点策略被自动设置为 Qt::StrongFocus(即部件通过 Tab 和点击获得焦点),
- 若此属性被设置为 Qt::TextSelectableByKeyboard, 则焦点策略被自动设置为 Qt::ClickFocus(即部件通点击获得焦点)。
- 默认值为 Qt::LinksAccessibleByMouse
- Qt::TextInteractionFlag 枚举描述文本显示部件与用户的交互方式, 可取值如下表

Qt::TextInteractionFlag 枚举

标志: Qt::TextInteractionFlags

作用: 描述文本显示部件与用户的交互方式

成员	值	说明
Qt::NoTextInteraction	0	不可与文本交互
Qt::TextSelectableByMouse	1	可使用鼠标选择文本, 并可使用鼠标的上下文菜单或键盘快捷键把文本复制到剪贴板。
Qt::TextSelectableByKeyboard	2	可使用键盘上的光标键选择文本, 并显示一个文本光标
Qt::LinksAccessibleByMouse	4	链接可使用鼠标激活
Qt::LinksAccessibleBykeyboard	8	链接可使用 tab 键获得焦点, 并使用 enter 键激活。
Qt::TextEditable	16	文本是可编辑的。
Qt::TextEditorInteraction	TextSelectableByMouse TextSelectableByKeyboard TextEditable 文本编辑器的默认值	
Qt::TextBrowserInteraction	TextSelectableByMouse LinksAccessibleByMouse LinksAccessibleBykeyboard 这是 QTextBrowser 的默认值。	

⑨、openExternalLinks: bool

访问函数: bool openExternalLinks() const; void setOpenExternalLinks(bool);

- 此属性描述, 是否应使用 QDesktopServices::openUrl() 自动打开链接, 而不是发出 linkActivated() 信号, 也就是说标签若以富文本的形式显示一个网页链接, 若此属性为 false, 则点击这个链接不会打开相应的网页。
- 要使用此属性需设置 textInteractionFlags 属性
- 默认为 false。

⑩、hasSelectedText: const bool

访问函数: bool hasSelectedText() const;

- 此属性描述是否有文本被选择, 若用户选择了部分或分部文本则返回 true, 否则返回 false。默认为 false。
- 注意: 标签上的文本默认是不可被选择的, 要使文本能被选择, 需设置 textInteractionFlags 属性。

⑪、selectedText: const QString

访问函数: QString selectedText() const;

此属性用于保存所选择的文本, 若未选择文本, 则为空字符串, 默认为空字符串。要使用此属性需设置 textInteractionFlags 属性以使标签能被选择。

⑫、pixmap: QPixmap

访问函数: const QPixmap* pixmap() const;

void setPixmap(const QPixmap&); //槽

此属性描述标签的像素图, 若没有设置此属性, 则返回 0, 设置此属性会清除以前的内容, 且伙伴快捷键会被禁用。

7、QLabel 类中的函数

①、QLabel(QWidget* parent = Q_NULLPTR, Qt::WindowFlags f = Qt::WindowFlags()); //构造函数

QLabel(const QString &text, QWidget* parent = Q_NULLPTR, Qt::WindowFlags f = Qt::WindowFlags());

- ②、`void clear();` //槽，清除标签的内容。
- ③、`void setBuddy(QWidget* buddy);`
把标签的伙伴设置为 `buddy`。若 `buddy` 为 0，则表示取消伙伴的设计。
- ④、`QWidget buddy() const;` 返回该标签的伙伴，若没有伙伴则返回 0。
- ⑤、`void setSelection(int start, int length);`
从 `start` 开始选择 `length` 个字符。注意：此函数需要设置 `textInteractionFlags` 属性。
- ⑥、`int selectionStart() const;`
返回标签中第一个选定的字符的索引，若未选择文本，则返回-1。注意：此函数需要设置 `textInteractionFlags` 属性。
- ⑦、`void setNum(int num);` //槽
`void setNum(double num);` //槽
将标签内容设置为纯文本，并把 `num` 以字符串的形式显示为标签的文本，并清除之前的内容，伙伴快捷键会被禁用。若 `num` 的字符串表示形式与标签的当前内容相同，则不执行任何操作。要注意的是数字和以文本形式显示的字符串并不是相同的，比如可把字符串直接赋值给 `QString` 对象，但不能把数字直接赋值给 `QString` 对象。
- ⑧、`void setPicture(const QPicture& picture);` //槽
将标签内容设置为图片 `picture`，并清除之前的内容，伙伴快捷键会被禁用。
- ⑨、`const QPicture* picture() const;` 返回标签的图片，若没有图片则返回 0。
- ⑩、`void setMovie(QMovie* movie);` //槽
将标签内容设置为影片 `movie`，并清除之前的内容，伙伴快捷键会被禁用。
- ⑪、`QMovie* movie() const;` 返回指向标签影片的指针，若没设置影片，则返回 0。
- ⑫、`void linkActivated(const QString &link);` //信号
当用户单击链接时发送此信号，引用的 URL 在 `link` 中传递。
- ⑬、`void linkHovered(const QString &link);` //信号，当用户悬停在链接上时，发送此信号。

示例 11：标签的边距/对齐、富文本、自动换行

```
#include<QtWidgets>
#include <iostream>
using namespace std;

int main(int argc, char *argv[]){    QApplication a(argc, argv);
    QWidget w;
    /*Qt 会自动检测到以下标签为富文本，该文本会被显示为具有网页链接的蓝色，以下内容以 HTML 语言
    规范解释为：把字符串“AAAA... kkkk”链接到网址 http://aaa */
    QLabel *pb=new QLabel("<a href=http://aaa>AAAA bbbb cccc dddd eeee ffff gggg kkkk</a>",&w);
    //设置标签边框样式，以使其看起来比较明显
    pb->setFrameStyle(QFrame::Box|QFrame::Raised);    pb->setLineWidth(5);
    pb->move(22, 22);    pb->resize(155, 86);    //设置标签位置及大小
    pb->setMargin(15);    //设置页边距为 15
    pb->setIndent(14);    //设置缩进量为 14(本例相对于左侧边缘，因为是左对齐的)
    pb->setWordWrap(true);    //设置文本在合适位置自动换行

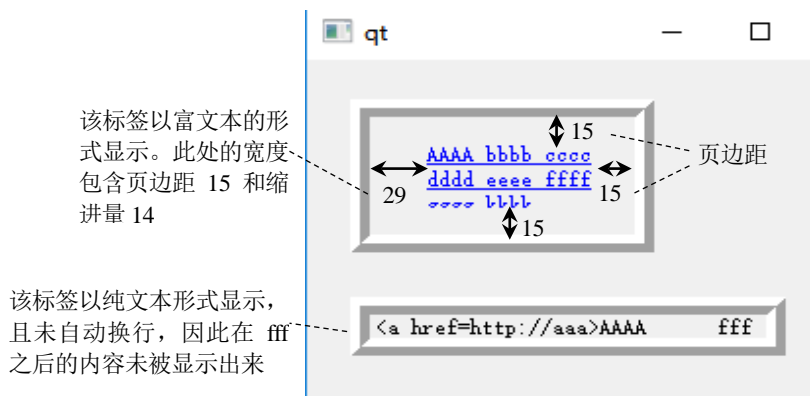
    //以下标签不能自动换行，也没缩进量和页边距，且以纯文本形式显示
    QLabel *pbl=new QLabel("<a href=http://aaa>AAAA    fff ggg</a>",&w);
```

```

pb1->setFrameStyle(QFrame::Box|QFrame::Raised);    pb1->setLineWidth(5);
pb1->move(22, 133);                                pb1->resize(222, 33);
pb1->setTextFormat(Qt::PlainText); //使标签文本以纯文本形式显示
w.resize(300, 200); w.show();    return a.exec();    }

```

运行结果及说明



示例 12：标签文本的选择、交互、超链接及其信号

//m.h 文件的内容

```

#ifndef M_H
#define M_H
#include<QtWidgets>
#include <iostream>
using namespace std;
class B:public QLabel{    Q_OBJECT
public:    B(QString s,QWidget* p):QLabel(s,p) {}
public slots:
void f(const QString &k){
    cout<<"link="<<k.toString()<<endl; //输出标签链接所关联的网址。
    cout<<selectedText().toString()<<endl;    }
};
#endif // M_H

```

//m.cpp 文件的内容

```

#include "m.h"
int main(int argc, char *argv[]) {    QApplication a(argc, argv);
    QWidget w;
    B *pb=new B("<a href=http://aaa>bbb</a>", &w);
    pb->move(22, 22);    pb->resize(222, 33);
    pb->setFrameStyle(QFrame::Box|QFrame::Raised);    pb->setLineWidth(5);
    pb->setOpenExternalLinks(true); /*设置标签可用浏览器打开链接，但该标签不再发送
                                    linkActivated 信号*/
    pb->setTextInteractionFlags(Qt::LinksAccessibleByMouse); //这是默认值
    QObject::connect(pb, &QLabel::linkActivated, pb, &B::f); //标签 pb 不会发送该信号。

    B *pb1=new B("<a href=http://aaa>ccc</a>", &w);
    pb1->setFrameStyle(QFrame::Box|QFrame::Raised);    pb1->setLineWidth(5);

```

```

pb1->resize(222, 33);          pb1->move(22, 63);
pb1->setOpenExternalLinks(false); /*这是默认值，此步可省略。此处表明标签会发送 linkActivated
                                信号，但不能打开链接。*/

//以下设置表示 pb1 的标签文本可编辑，可使用鼠标和键盘进行交互，也可使用鼠标和键盘激活链接。
pb1->setTextInteractionFlags(Qt::LinksAccessibleByMouse|Qt::LinksAccessibleByKeyboard|
                             Qt::TextSelectableByMouse|Qt::TextSelectableByKeyboard|
                             Qt::TextEditable);
QObject::connect(pb1, &QLabel::linkActivated, pb1, &B::f); //标签 pb1 会发送此信号

B *pb2=new B("<a href=http://aaa>ddd</a>", &w);
pb2->setFrameStyle(QFrame::Box|QFrame::Raised);          pb2->setLineWidth(5);
pb2->resize(222, 33);          pb2->move(22, 133);

/*以下设置表示，pb2 的标签文本只可使用鼠标和键盘交互，不能使用鼠标或键盘激活链接，因此点击
pb2 不会打开链接，也不会发送 linkActivated 信号*/
pb2->setTextInteractionFlags(Qt::TextSelectableByMouse|Qt::TextSelectableByKeyboard);
QObject::connect(pb2, &QLabel::linkActivated, pb2, &B::f); //标签 pb2 不会发送该信号
w.resize(300, 200);          w.show();          return a.exec();  }

```

运行结果及说明

用鼠标点击此链接会用浏览器打开网页，但不会发送 linkActivated 信号

点击此链接不会打开网页，但会发送 linkActivated 信号。同时此标签中的文本是可编辑的

点击此链接即不会打开网页，也不会发送 linkActivated 信号



说明：

- 1、选择标签 ccc 中文本的方法，可按下键盘 Shift 键不放，然后移动键盘方向键来选择文本，或用鼠标选择文本，当选择好之后，按下 enter 键激活该链接，此时会发送 linkActivated 信号，调用 f 函数，输出用户选择的文本。注意，若用鼠标点击激活链接，则只会选中该标签中的所有文本
- 2、B::f()槽的参数，保存的是标签的链接所关联的网址，也就是说激活链接 ccc 后，该参数保存的内容是"http://aaa"，而不是"ccc"
- 3、同时标签 ccc 的文本是可编辑的，也就是说用户可以把文本"ccc"删掉，再重新输入其他字符。

示例 13：标签图片的显示与缩放、伙伴、数字

//m.h 文件的内容

```

#ifndef M_H
#define M_H
#include<QtWidgets>
#include <iostream>
using namespace std;
class B:public QLabel{    Q_OBJECT

```



```

public: B(QString s, QWidget* p):QLabel(s, p) {}
public slots:
void f() {
    setScaledContents(hasScaledContents() ^ true); //使标签的内容在缩放与不缩放之间转换
} };
#endif // M_H

//m.cpp 文件的内容
#include "m.h"
int main(int argc, char *argv[]) {    QApplication a(argc, argv);
    QWidget w;        QPushButton *b=new QPushButton("AAAA", &w);
    B *pb=new B("<a href=http://aaa>bbb</a>", &w);
    pb->move(22, 22);    pb->resize(222, 88);

    //设置标签边框样式，以使其看起来比较明显
    pb->setFrameStyle(QFrame::Box|QFrame::Raised);    pb->setLineWidth(5);

    //把 f 盘下的 1.png 设置为标签的图片，此设置会清除标签之前的内容。
    pb->setPixmap(QPixmap("F:\\1.png"));
    QObject::connect(b, &QPushButton::clicked, pb, &B::f); //关联信号与槽函数

//以下内容验证数字的显示与伙伴的设置
    QLabel *pb1=new QLabel("ab&cde", &w); //该标签无伙伴，"&"字符会被显示在标签文本中
    pb1->move(22, 128);

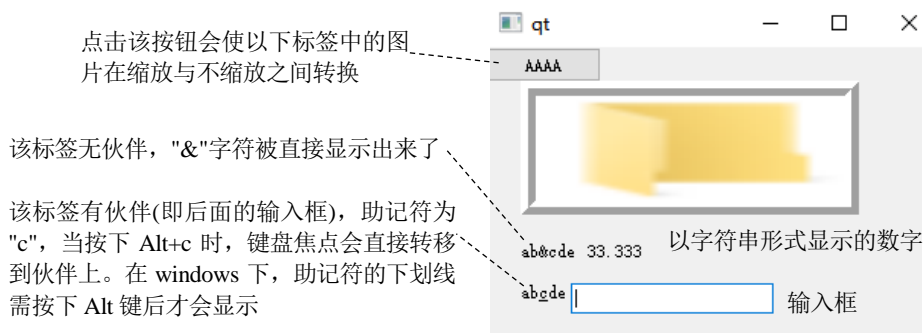
    QLabel *pb2=new QLabel("ab&cde", &w);    pb2->move(66, 128);
    pb2->setNum(33.333); //将数字 33.333 以字符串的形式显示为标签的文本，并清除之前的内容。

    QLabel *pb3=new QLabel("ab&cde", &w);    pb3->move(22, 155);
    QLineEdit *pe=new QLineEdit(&w);    pe->move(55, 155);
    pb3->setBuddy(pe); //设置伙伴后，标签的"&"字符将不会被显示。

    w.resize(300, 200);    w.show();    return a.exec(); }

```

运行结果及说明



8、显示省略号，字体大小简介

①、省略号、字体大小的功能不是在 QLabel 类中实现的，因此此处仅作一简介。

- ②、当标签中的文本太长无法显示时，可以使用省略号的形式来表示标签的内容未被完全显示，要使用该功能，需要借助于 `QFontMetrics` 类。要设置标签文本的大小、粗细、是否倾斜等需借助于 `QFont` 类(当然也可使用富文本的形式实现这些功能)。
- ③、省略号的实现，需使用 `QFontMetrics::elidedText()`函数处理需要显示的字符串，然后使用该函数返回的字符串设置为标签的文本，可使用 `QWidget::fontMetrics()`函数获取 `QFontMetrics` 类型的对象，然后通过该对象调用 `elidedText()`函数，具体见下面的示例。
- ④、设置字体大小：需使用 `QFont` 对象，然后调用该类的成员函数设置字体的大小等样式，再调用 `QWidget::setFont()`函数，把标签的字体设置为该 `QFont` 对象。具体见下面的示例

示例 14：标签文本的省略号、字体大小

```
#include<QtWidgets>
#include <iostream>
using namespace std;

int main(int argc, char *argv[]){    QApplication a(argc,argv);
    QWidget w;
    QLabel *pb=new QLabel(&w);  pb->move(22,22);
    QString s="AAAAAADDDDDDDDDDEEEEEEEEE";    //想要显示的字符。

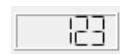
    /*以下 elidedText 函数表示，处理字符串 s，使其在中间显示省略号，最终字符串的显示长度为 55 个
    像素，并返回处理后的字符串。*/
    QString s1=pb->fontMetrics().elidedText(s,Qt::ElideMiddle,55);
    pb->setText(s1);    //将处理后的字符串设置为标签的文本

    QFont f("宋体");    //创建字体，该字体为“宋体”
    f.setPointSize(33);    //设置字体大小。
    f.setBold(true);    //加粗字体
    f.setItalic(true);    //使字体倾斜
    pb->setFont(f);    //为标签 pb 安装以上设置的字体
    w.resize(300,200);        w.show();    return a.exec();}
```



运行结果见右图

三、QLCDNumber 类(LCD 数字)，LCD 表示液晶显示屏



- 1、QLCDNumber 类用于显示类似于 LCD 显示屏上的字符(见右图)
- 2、QLCDNumber 类是 `QFrame` 类的直接子类，因此 `QLCDNumber` 以使用从 `QFrame` 类继承而来的边框效果
- 3、QLCDNumber 可显示的符号有：0，1，2，3，4，5，6，7，8，9，A，B，C，D，E，F，g，h，H，L，o，O，P，r，S，u，U，Y，小数点、减号、冒号、空格、度(使用键盘上的单引号指定)。不能显示的字符使用空格代替。
- 4、无法获取 `QLCDNumber` 对象的内容，仅能使用其成员函数 `value()`获取其对应的数值，若需要获取 `QLCDNumber` 对象中的文本，可把连接到 `display()`槽的信号，连接到另一个槽，并用该槽来存储 `QLCDNumber` 对象的文本。`display()`槽用于把内容显示到 `QLCDNumber`

对象上。

5、QLCDNumber 类可以显示 2 进制、8 进制、10 进制、16 进制，十进制可以显示浮点数，其他进制只会显示等效的整数值。

6、QLCDNumber 类中的属性

注意：设置以下属性时，需在 display()槽函数之前进行设置，否则有可能不会按设置的属性显示 LCD 数字。比如，若 setDigit()位于 display()之后，则显示的 LCD 数字位数仍是按之前设置的位数显示。

①、digitCount: int 访问函数: int digitCount() const; void setDigitCount (int);

此属性描述显示的位数，小数点会占 1 位，位数必须在 0~99 之间。默认值为 5。

②、smallDecimalPoint: bool

访问函数: bool smallDecimalPoint() const; void setSmallDecimalPoint(bool)

此属性描述小数点的样式，若为 true，则小数点位于两数字之间，此时数字间的间隙会变得稍宽一点，若为 false，则小数点会占据一个数字位置(见下图)。默认为 false。



③、mode: Mode 访问函数: Mode mode() const; void setMode (Mode);

此属性描述当前的显示模式(即显示的基数)，即 2 进制、8 进制、10 进制(默认)、16 进制。Mode 是 QLCDNumber 类中的枚举，该枚举描述了数字的基数，其成员如下表

Mode 枚举(无标志)

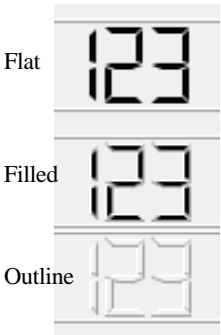
成员	值	说明	成员	值	说明
QLCDNumber::Hex	0	16 进制	QLCDNumber::Oct	2	8 进制
QLCDNumber::Dec	1	10 进制	QLCDNumber::Bin	3	2 进制

④、segmentStyle: SegmentStyle

访问函数: SegmentStyle segmentStyle() const; void setSegmentStyle(SegmentStyle)

此属性描述了 LCD 数字的外观，其中 SegmentStyle 是 QLCDNumber 类中的枚举，用于描述其外观(见右图)，其成员如下表

SegmentStyle 枚举(无标志)		
成员	值	说明
QLCDNumber::Outline	0	生成填充了背景色的凸起效果
QLCDNumber::Filled	1	生成填充前景色的凸起效果(默认值)
QLCDNumber::Flat	2	生成填充了前景色的平面效果
说明：若显示的数字过小，Filled 看起来与 Outline 相同，若显示的数字过大，则 Filled 看起来与 Flat 相同。		



⑤、`value: double` 访问函数: `double value() const;`

`intValue: int` 访问函数: `int intValue() const;`

- 以上两个属性的设置函数都是 `display()` 槽函数，详见后面的函数部分。
- 以上两个属性都是对应于 `QLCDNumber` 上显示的当前值，其中 `intValue` 属性描述的是当前值的最近整数值(采用四舍五入方式)，该属性用于 16 进制、8 进制和 2 进制模式。
- 若显示的值不是数字，则以上两属性的值都为 0，他们的默认也都为 0。

7、QLCDNumber 类中的函数

①、`QLCDNumber(QWidget* parent = Q_NULLPTR);`

`QLCDNumber(uint numDigits, QWidget* parent = Q_NULLPTR);`

构造函数，构造一个 LCD 数字，其位数设置为 5(或 `numDigits`)，基数为 10 进制，`smallDecimalPoint` 属性为 `false`，边框样式为凸起框，`segmentStyle` 属性为 `Filled`。

②、`void display(const QString &s);` //槽

`void display(int num);` //槽

`void display(double num);` //槽

以上槽函数表示，在 `QLCDNumber` 上显示由 `s` 或 `num` 表示的字符或数字。以上槽函数是 `intValue` 和 `value` 属性的设置函数。

③、`void setBinMode();` //槽，设置为 2 进制模式。

`void setDecMode();` //槽

`void setHexMode();` //槽

`void setOctMode();` //槽

以上函数是一种更便捷的设置进制模式的函数，可使用以上槽函数方便的与其他部件(比如按钮)的信号进行连接。

④、`bool checkOverflow(double num) const;`

`bool checkOverflow(int num) const;`

以上函数用于检测数字 `num` 是否太大而无法显示，若无法显示则返回 `true`，否则返回 `false`。

⑤、`void overflow();` //信号，若显示的字符或数字太大时，发送此信号。

示例 15: QLCDNumber 类的使用

```
#include<QtWidgets>
#include <iostream>
using namespace std;
```

```
int main(int argc, char *argv[]) {      QApplication a(argc, argv);
    QWidget w;
    QLCDNumber *pn=new QLCDNumber(&w);
    pn->move(22, 22); pn->resize(222, 44);
    pn->setDigitCount(7);    /*设置显示的位数，该函数应位于 display 函数之前，否则该函数的设置会
                           不起作用。*/
    pn->display(1245555345.2275687);    //太大的浮点数会以科学计数的形式显示
```

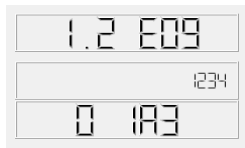
```

QLCDNumber *pn1=new QLCDNumber(&w);
pn1->move(22,77);    pn1->resize(222,44);
pn1->setSegmentStyle(QLCDNumber::Flat);
pn1->setDigitCount(20);
pn1->display("1234");
cout<<pn1->value()<<endl;    //输出 1234

QLCDNumber *pn2=new QLCDNumber(&w);
pn2->move(22,122);    pn2->resize(222,44);
pn2->display("0x1a3"); /*试图以 16 进制形式显示后面的数值 1a3，但不会成功，因为 0x1a3 会被当
                        作字符串显示，其中 x 无法显示，会被空格代替。*/
cout<<pn2->value()<<endl;    //输出 0，因为 pn2 显示的"0 1A3"不是有效数字。
    w.resize(300,200);    w.show();    return a.exec(); }

```

运行结果及说明



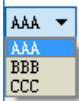
Qt 会根据部件的大小和显示的位数自动调整需要显示的字符的大小，设置的显示位数越多，则显示时其数字就越小

4.4 输入部件

QComboBox(组合框)、QLineEdit(行编辑器)、QValidator(验证器)

一、QComboBox 类(下拉列表、组合框)

1、QComboBox 类是 QWidget 类的直接子类，该类实现了一个组合框



2、QComboBox 类中的属性

QComboBox 类(组合框)属性速查表

属性名	说明	属性名	说明
count	获取项目数量	minimumContentsLength	组合框中最少字符数
maxCount	允许的最大项数	maxVisibleItems	向用户显示的最大项目数
editable	是否可被编辑	sizeAdjustPolicy	组合框大小变更策略
currentIndex	当前项目的索引	insertPolicy	插入新项目时在组合框中的位置策略
currentText	当前项目的文本	duplicatesEnabled	内容是否可重复
iconSize	组合框中图标的大小	modelColumn	模型中可见的列
frame	是否绘制默认边框	currentData	当前项目的数据

①、count: const int

访问函数: int count() const;

获取组合框中的项目数量，默认情况下，对于空组合框或未设置当前项目的组合框，其值为 0。

②、maxCount: int

访问函数: int maxCount() const; void setMaxCount(int);

此属性描述组合框允许的最大项数，若设置的最大数小于组合框中当前的项目数量，则额外的项目会被截断。默认值为可使用的最高带符号整数(通常为 2147483647)。

③、maxVisibleItems: int

访问函数: int maxVisibleItems() const; void setMaxVisibleItems(int);

此属性描述组合框在屏幕上向用户显示的项目数量(即可见项目数)(见右图)。默认为 10。注意：该属性在某些样式上可能会被忽略。



可见数为 2

④、minimumContentsLength: int

访问函数: int minimumContentsLength() const; void setMinimumContentsLength(int);

此属性描述，组合框项目的最少字符数量(见下图)，若此属性为正值，则 minimumSizeHint()和 sizeHint()会被考虑在内，默认为 0。



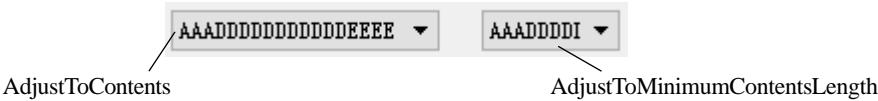
⑤、sizeAdjustPolicy: SizeAdjustPolicy

访问函数: SizeAdjustPolicy sizeAdjustPolicy() const; void setSizeAdjustPolicy(SizeAdjustPolicy);

- 此属性描述，当组合框的内容更改时，其组合框的大小如何更改。默认值为 AdjustToContentsOnFirstShow。注意，当 editable 被启用时，此属性需位于 editable 属性之前，否则该属性可能不起作用。
- SizeAdjustPolicy 是 QComboBox 类中的枚举，用于描述组合框的大小更改策略，其成员如下

QComboBox::SizeAdjustPolicy 枚举(无标志)

成员	值	说明
QComboBox::AdjustToContents	0	根据内容调整(见下图)
QComboBox::AdjustToContentsOnFirstShow(默认值)	1	第一次显示时，根据内容调整
QComboBox::AdjustToMinimumContentsLength	2	使用 AdjustToContents 或 AdjustToContentsOnFirstShow 代替
QComboBox::AdjustToMinimumContentsLengthWithIcon	3	调整为 minimumContentsLength 属性的大小加上图标空间。



⑥、insertPolicy: InsertPolicy

访问函数: InsertPolicy insertPolicy() const; void setInsertPolicy(InsertPolicy);

此属性描述插入新项目时应该出现在组合框中的位置，默认是 InsertAtBottom(新项目插入到底部)，其中 InsertPolicy 是 QComboBox 类中的枚举，该枚举用于描述插入项目的位置，其成员见下表。

QComboBox::InsertPolicy 枚举(无标志)

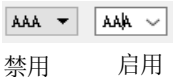
成员	值	说明
QComboBox::NoInsert	0	字符串不会插入到组合框中
QComboBox::InsertAtTop	1	字符串插入组合框中的第一项
QComboBox::InsertAtCurrent	2	使用字符串替换掉当前项目
QComboBox::InsertAtBottom	3	字符串插入到组合框的最后一项之后(默认值)
QComboBox::InsertAfterCurrent	4	字符串插入到组合框的当前项目之后
QComboBox::InsertBeforeCurrent	5	字符串插入到组合框的当前项目之前
QComboBox::InsertAlphabetically	6	字符串按字母顺序插入到组合框中。

⑦、editable: bool

访问函数: bool isEditable() const; void setEditable(bool);

此属性描述，组合框是否可由用户编辑，默认为 fasle。

注意：当禁用该属性时，将删除 validator 和 completer。



⑧、duplicatesEnabled: bool

访问函数: bool duplicatesEnabled() const; void setDuplicatesEnabled(bool);

此属性描述，用户是否可把重复项目输入到组合框中，注意：以编程的方式总是可以

插入重复项目到组合框中。默认为 false(不允许重复)

⑨、**currentData**: const QVariant //qt5.2

访问函数: QVariant currentData(int role = Qt::UserRole) const;

保存当前项目的数据, 对于空组合框或未设置当前项目的组合框, 默认情况下, 此属性为无效的 QVariant。

⑩、**currentIndex**: int

访问函数: int currentIndex() const; void setCurrentIndex(int);

信号: currentIndexChanged(int); void currentIndexChanged(const QString&);

此属性描述组合框当前项目的索引(从 0 开始), 插入或删除时, 索引可能会改变, 对于空组合框或未设置当前项目的组合框, 默认情况下, 此属性的值为-1。

⑪、**currentText**: QString

访问函数: QString currentText() const; void setCurrentText(const QString&);

信号: void currentTextChanged(const QString&);

- 此属性描述当前的文本, 注意: 设置函数 setCurrentText()并不能把新文本添加到组合框中, 该函数仅能使组合框显示该文本。
- 此属性的 setTextCurrentText()仅在组合框可编辑时才会起作用。
- 若组合框是可编辑的, 则 currentText 是编辑时显示的文本,
- 若组合框为空或未设置当前项目的组合框, 则为当前项目的值或空字符串。
- 若组合框是可编辑的, 则设置函数 setCurrentText()只需调用 setEditText()函数。

⑫、**iconSize**: QSize

访问函数: QSize iconSize() const; void setIconSize(const QSize&);

此属性描述组合框中显示的图标的大小。默认值是图标可以拥有的最大大小, 较小尺寸的图标不会被放大。

⑬、**frame**: bool

访问函数: bool hasFrame() const; void setFrame(bool);

此属性描述组合框是否绘制默认的边框, 默认为 true (启用)

⑭、**modelColumn**: int

访问函数: int modelColumn() const; void setModelColumn(int);

此属性描述, 模型中可见的列, 若此属性设置于填充组合框的内容之前, 则弹出的视图不会受到影响, 并且会显示第 1 列(默认值)。默认为 0。

示例 16: QComboBox (组合框) 的属性

//m.h 文件内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>
#include <iostream>
using namespace std;

class B:public QComboBox{    Q_OBJECT
public:    B(QWidget* p=0):QComboBox(p) {}
public slots:
```

```

void f() {      cout<<currentIndex()<<endl; //获取当前项目的索引
               cout<<currentText().toString()<<endl; //获取当前项目的文本
            }

void f1() {setCurrentText("EEE"); }//将 EEE 显示在组合框中，但不会添加到组合框中
};

#endif // M_H

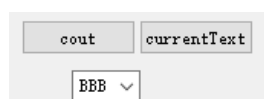
//m.cpp 文件的内容
#include "m.h"
int main(int argc, char *argv[]) {      QApplication a(argc, argv);
    QWidget w;
    QPushButton *b=new QPushButton("cout",&w);          b->move(22,22);
    QPushButton *b1=new QPushButton("currentText",&w);    b1->move(99,22);

    B *pcl=new B(&w);          pcl->move(55,55);
    //向组合框中添加内容
    pcl->insertItem(1,"AAA");    pcl->insertItem(2,"BBB");    pcl->insertItem(4,"CCC");

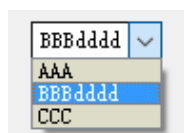
    QObject::connect(b, &QPushButton::clicked, pcl, &B::f);    //连接信号与槽
    QObject::connect(b1, &QPushButton::clicked, pcl, &B::f1);
    pcl->setSizeAdjustPolicy(QComboBox::AdjustToContents); //根据内容自动调整组合框大小
    pcl->setCurrentIndex(1); //设置当前项目的索引为 1，初始显示时会显示索引为 1 的项目。
    pcl->setDuplicatesEnabled(true); //可向组合框中添加重复的内容。
    pcl->setEditable(1);          //使组合框可编辑
    pcl->setInsertPolicy(QComboBox::InsertAtCurrent); //插入的内容替换当前项目
    w.resize(300,200);          w.show();    return a.exec();    }

```

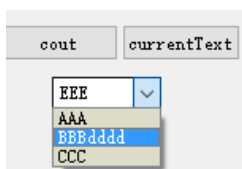
运行结果及说明



初始显示时，显示的是索引为 1 的项目，当点击 cout 按钮时会输出字符 BBB 及其索引 1。



输入图示内容，并按下回车后的情形，可见，组合框根据内容自动扩大了 size，而且输入的内容替换掉了当前的项目的内容



接着按下 currentText 按钮，此时组合框会显示 EEE，但该文本并未添加到组合框之中。

3、QComboBox 类中的函数

- ①、`QComboBox(QWidget* parent = Q_NULLPTR);` //构造函数
- ②、`void addItem(const QString &text, const QVariant& userData = QVariant());`
`void addItem(const QIcon &icon, const QString &text, const QVariant& userData = QVariant());`
`void addItems(const QStringList &texts);` //注意，该函数后面的字母 s

以上函数用于向组合框中添加内容，其中 `userData`(暂时不需要理解此参数)存储于 `Qt::UserRole` 中。

- ③、`void setItemText(int index, const QString &text);`
`QString itemText(int index) const;`

以上函数分别表示，把索引为 `index` 处的项目设置为 `text` (可用于修改项目)，和获取索引为 `index` 处的文本。索引是从 0 开始的。

- ④、`void setItemIcon(int index, const QIcon &icon);`
`QIcon itemIcon(int index) const;`

以上函数分别表示，把索引为 `index` 处的项目的图标设置为 `icon`，和获取索引为 `index` 处的图标。索引是从 0 开始的。

- ⑤、`void insertItem(int index, const QString &text, const QVariant &userData = QVariant());`
`void insertItem(int index, const QIcon &icon, const QString &text,`
`const QVariant &userData = QVariant());`

`void insertItems(int index, const QStringList &list);` //注意最后的字母 s

以上函数用于向组合框中插入内容，若指定的索引大于或等于项目总数，则新项目被追加到项目的末尾，若索引为 0 或负数，则新项目被添加到现有项目的前面。

- ⑥、`void removeItem(int index);`

删除指定索引处的项目，若索引被删除，将更新当前索引，若索引超出范围，则此函数不执行任何操作。

- ⑦、`void clear()` //槽，清除组合框中的所有项目

- ⑧、`void insertSeparator(int index);`

在索引 `index` 处插入分隔器，若指定的索引大于或等于项目总数，则新项目被追加到项目的末尾，若索引为 0 或负数，则新项目被添加到现有项目的前面。注：插入的分隔器在视觉上可能不会很明显的看得出来。

- ⑨、`void setLineEdit(QLineEdit* edit);`

设置组合框的文本编辑部件为 `edit`，设置该项之后，组合框会成为可编辑的。

- ⑩、`QLineEdit* lineEdit() const;`

返回组合框中的行编辑器，若没有，则返回 0。只有可编辑的组合框才会行编辑器。

- ⑪、`void setEditText(const QString &text);` //槽

将组合框的文本设置为 `text`，此函数不能把新文本添加到组合框中，仅能使组合框显示该文本。组合框是可编辑的状态时，该函数才会起作用。该函数与 `currentText` 属性的设置函数 `setCurrentText()`类似。

`void clearEditText()` //槽

此函数用于清除 `setEditText()`函数设置的文本，也可用于清除正在编辑的文本。

- ⑫、`virtual void showPopup();` //虚函数

`virtual void hidePopup()` //虚函数

以上函数用于显示或隐藏项目列表。

- ⑬、`int findText(const QString &text, Qt::MatchFlags flags = static_cast<Qt::MatchFlags>`
`(Qt::MatchExactly | Qt::MatchCaseSensitive)) const;`

查找 `text` 所在项目的索引。其中 `Qt::MatchFlag` 枚举是用于描述查找时的匹配方式的，

Qt::MatchFlags 是该枚举的标志，此处默认的区配方式是执行基于 QVariant 的匹配，且搜索时区分大小写。详细内容请参阅“部件公用枚举”章节。

⑭、`void setValidator(const QValidator* validator);`
`const QValidator* validator() const;`

以上函数用于设置和获取 QValidator(验证器)的，验证器用于对输入的文本进行验证，也就是说可以使用该类来限制用户的输入(比如只能输入数字等)。QValidator 类会在后文讲解。验证器需组合框在可编辑状态下。

示例 17: QComboBox (组合框) 内容的添加、设置、插入、移除、清除、查找

//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>
#include <iostream>
using namespace std;
//注：使用以下方式组织程序可以方便信号和槽的关联
class B:public QWidget{    Q_OBJECT
public:                    //创建部件
    QPushButton *b;       QPushButton *b1;   QPushButton *b2;   QPushButton *b3;
    QPushButton *b4;       QPushButton *b5;   QPushButton *b6;   QComboBox *pc1;  QLineEdit *pe;
    B(QWidget* p=0):QWidget(p) { //构造函数开始
        //创建和布局部件
        b=new QPushButton("add", this);        b1=new QPushButton("set", this);
        b2=new QPushButton("insert", this);    b3=new QPushButton("remove", this);
        b4=new QPushButton("clear", this);     b5=new QPushButton("show", this);
        b6=new QPushButton("find", this);
        b->move(22, 22);        b1->move(22, 44);    b2->move(22, 66);    b3->move(22, 88);
        b4->move(22, 111);    b5->move(22, 133);    b6->move(22, 155);
        pc1=new QComboBox(this);                pc1->move(111, 77);
        pe=new QLineEdit("XXXX", this);        pe->move(111, 44);
        pc1->setSizeAdjustPolicy(QComboBox::AdjustToContents); //组合框根据内容自动调整大小
        //向组合框 pc1 中添加文本
        QStringList s;
        s.append("AAA");        s.append("BBB");        s.append("CCC");        s.append("DDD");
        pc1->addItems(s);
        //连接信号与槽
        QObject::connect(b, &QPushButton::clicked, this, &B::addf);
        QObject::connect(b1, &QPushButton::clicked, this, &B::setf);
        QObject::connect(b2, &QPushButton::clicked, this, &B::insertf);
        QObject::connect(b3, &QPushButton::clicked, this, &B::removef);
        QObject::connect(b4, &QPushButton::clicked, pc1, &QComboBox::clear);
        QObject::connect(b5, &QPushButton::clicked, this, &B::showf);
        QObject::connect(b6, &QPushButton::clicked, this, &B::findf);
    } //构造函数结束

public slots:
    void addf() {        QString s=pe->text();    pc1->addItem(s); } //在末尾添加文本 s
    void setf() {        QString s=pe->text();    int i=pc1->currentIndex();
        pc1->setItemText(i, s);    } //把当前索引号处的文本设置为 s
    void insertf() {        QString s=pe->text();    int i=pc1->currentIndex();
        pc1->insertItem(i, s); } //在当前索引号处插入文本 s
```

```

void removef() {int i=pcl->currentIndex(); pcl->removeItem(i);} //移除当前索引号处的文本
void showf() {    pcl->showPopup();}
void findf() {    QString s=pe->text();
    //执行模糊搜索且区分大小写
    cout<<pcl->findText(s,Qt::MatchContains|Qt::MatchCaseSensitive)<<endl;    }
};
#endif // M_H

//m.cpp 文件的内容
#include "m.h"
int main(int argc, char *argv[]) {    QApplication a(argc, argv);
    B mb;        mb.resize(300,200);        mb.show();    return a.exec(); }

```

运行结果及说明



点击 add 按钮会把输入框的内容添加到组合框的末尾
 点击 insert 按钮, 会把输入框的内容插入到组合框的当前索引处
 点击 set 按钮, 会修改组合框当前索引处的文本,
 点击 remove 按钮, 会删除组合框当前索引处的文本
 点击 clear 按钮, 会清除组合框的所有内容
 点击 show 按钮, 会显示组合框的下拉列表
 点击 find 按钮, 会根据输入框的内容在组合框中进行查找, 若找到匹配的文本, 则输出该索引, 否则输出-1

4、以下函数暂时先不需了解, 因为需要模型/视图结构的知识。

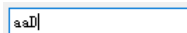
- ①、`QCompleter* completer() const;`
`void setCompleter(QCompleter* completer);`
 QCompleter 类主要用于实现自动完成功能(即输入内容时的自动补全功能)
- ②、`int findData(const QVariant &data, int role = Qt::UserRole, Qt::MatchFlags flags =`
`static_cast<Qt::MatchFlags>(Qt::MatchExactly | Qt::MatchCaseSensitive)) const;`
- ③、`QVariant itemData(int index, int role = Qt::UserRole) const;`
`void setItemData(int index, const QVariant &value, int role = Qt::UserRole);`
- ④、`QAbstractItemModel* model() const;`
`void setModel(QAbstractItemModel* model);`
- ⑤、`QModelIndex rootModelIndex() const;`
`void setRootModelIndex(const QModelIndex& index);`
- ⑥、`void setView(QAbstractItemView* itemView);`
`QAbstractItemView* view() const;`
- ⑦、`QAbstractItemDelegate* itemDelegate() const;`
`void setItemDelegate(QAbstractItemDelegate* delegate);`

5、QComboBox 类中的信号

- ①、`void activated(int index);` //信号
`void activated(const QString &text);` //信号
- 当用户在组合框中选中一个项目时，发送以上信号，其中 `index` 是被选中项目的索引，`text` 是被选中项目的文本。
 - 注意：即使选择未改变(即两次都选择相同的项目)，也会发送以上信号。
 - 注意：在组合中展开的下接列表中仅仅移动加亮条并不会使项目被选中，选择项目后需点击鼠标或按下 `enter` 键才会使项目选中。也可在组合框获得焦点时，下拉列表处于隐藏状态下，使用键盘上的上/下方向键选中组合框中的项目。
- ②、`void currentIndexChanged(int index);` //currentIndex 属性改变时发送
`void currentIndexChanged(const QString &text);` //currentIndex 属性改变时发送
`void currentTextChanged(const QString &text);` //currentText 属性改变时发送
- 当组合框中的 `currentIndex` 或 `currentText` 属性通过用户或编程的方式被改变时，就会发送以上信号。其中 `index` 是被改变后的项目的索引，`text` 是被改变后的项目的文本。
 - 注意：以上信号发送的条件是 `currentIndex` 或 `currentText` 属性改变，下面是其改变的时机
 - 在选中不同的项目后这两个属性都会产生改变，也就是说在组合框展开的下接列表中仅仅移动加亮条并不会使这两个属性改变，需要使项目被选中(使用鼠标点击或按下 `enter` 键)且不能选择与之前相同的项目时，`currentIndex` 或 `currentText` 属性才会改变。
 - 当组合框在可编辑状态下时，在组合框中改变文本的内容时，`currentText` 属性会改变，但 `currentIndex` 属性不会改变。
- ④、`void editTextChanged(const QString &text);` //信号
- 当组合框的行编辑器部件中的文本被更改时，发送此信号。`text` 是改变后的新文本。文本更改的时机，与 `currentText` 属性改变的时机相同，详见 `currentTextChanged` 信号。
- ⑤、`void highlighted(int index);` //信号
`void highlighted(const QString &text);` //信号
- 当在组合框展开的下接列表中改变加亮条时，发送以上信号。
- ⑥、以上信号发送时机的测试，读者可自行编写程序验证。

二、QLineEdit 类(行编辑器)

1、QLineEdit 类是 QWidget 类的直接子类，该类实现了一个单行的输入部件，即行编辑器，见右图



- 2、验证器(QValidator 类)和输入掩码简介：主要作用是验证用户输入的字符是否符合验证器的要求，即限制对用户的输入，比如仅能输入数字而不能输入字母等。
- 3、行编辑器默认支持复制、剪切、粘贴等常用操作，还支持一些常用的快捷键(比如 `Ctrl+C` 等)和鼠标右键的上下文菜单。下表为默认支持的键盘快捷键


默认支持的快捷键

按键	说明	按键	说明
左箭头	使光标向左移动一个字符	Ctrl+A	全选
Shift+左箭头	向左移动并选择一个字符	Ctrl+C	复制
右箭头	使光标向右移动一个字符	Ctrl+Insert	复制
Shift+右箭头	向右移动并选择一个字符	Ctrl+K	删除到行尾
Home	将光标移至行的开头	Ctrl+V	粘贴
End	将光标移至行尾	Shift+Insert	粘贴
Backspace	删除光标左侧的字符	Ctrl+X	剪切
Ctrl+Backspace	删除光标左侧的一个单词	Shift+Delete	剪切(即删除并复制)
Delete	删除光标右侧的字符	Ctrl+Z	撤消最后的操作
Ctrl+Delete	删除光标右侧的一个单词	Ctrl+Y	重做最后的操作

4、QLineEdit 类中的属性

QLineEdit 类(行编辑器)属性速查表

属性名	说明	属性名	说明
inputMask	设置输入掩码	acceptableInput	是否符合掩码和验证的要求
alignment	对齐方式	maxLength	文本最大长度
clearButtonEnabled	是否显示清除按钮	modified	内容是否被修改
cursorMoveStyle	文字光标移动的方式	placeholderText	是否显示占位符文本
cursorPosition	文字光标的位置	readOnly	是否启用只读模式
displayText	获取回显的文本	redoAvailable	描述重做是否已启用
dragEnabled	是否启用拖放	text	获取和设置文本
echoMode	回显模式	undoAvailable	描述撤消是否已启用。
frame	是否绘制边框	selectedText	获取选中的文本，设置需使用函数 setSelection()
hasSelectedText	是否有文本被选中		

- ①、**acceptableInput**: const bool 访问函数: bool hasAcceptableInput() const;
此属性描述，输入是否符合 inputMask 属性和 QValidator 类的要求。默认为 true。
- ②、**alignment**: Qt::Alignment
访问函数: Qt::Alignment alignment() const; void setAlignment(Qt::Alignment);
此属性描述行编辑器文本的对齐方式，默认为 Qt::AlignLeft(左对齐)和 AlignVenter(垂直居中)，Qt::Alignment 枚举详见“部件的公共枚举”章节。
- ③、**clearButtonEnabled**: bool //Qt5.2  清除按钮
访问函数: bool isClearButtonEnabled() const; void setClearButtonEnabled(bool);
此属性描述，行编辑器是否在不为空时显示“清除”按钮(见上图)。
- ④、**frame**: bool 访问函数: bool hasFrame() const; void setFrame(bool);
此属性描述是否绘制 QLineEdit 的默认边框(见下图)，默认为 true。

未绘制边框   绘制边框

- ⑤、**placeholderText**: QString
访问函数: QString placeholderText() const; void setPlaceholderText(const QString&);

此属性描述是否启用行编辑器的占位符文本。若行编辑器是空的,则当设置此属性后,即使行编辑器具有焦点,也会使行编辑器显示一个灰色的占位符文本(见下图),若在行编辑器中输入文本,则会清除占位符文本。默认为空字符串

当行编辑器为空时,显示的灰色占位符文本



- ⑥、**maxLength**: int 访问函数: int maxLength() const; void setMaxLength(int);
此属性描述文本的最大允许长度,若文本太长,会被截断。当发生截断时,任何选定的文本会被取消,文字光标位置设置为 0,且显示字符串的第一部分。默认为 32767
- ⑦、**hasSelectedText**: const bool 访问函数: bool hasSelectedText() const;
此属性描述是否有文本被选中,若用户选中部分或全部文本,则为 true,否则为 false,默认为 false。
- ⑧、**selectedText**: const QString 访问函数: QString selectedText() const;
获取选定的文本(要设置选中的文本需使用函数 setSelection()),若没有文本被选中,则此属性为空字符串,默认为空字符串。
- ⑨、**text**: QString 访问函数: QString text() const; void setText(const QString&);
信号: void textChanged(const QString &);
获取和设置行编辑器的文本。若设置此属性,则会清除所选的内容、撤消/重做历史记录,并将文字光标移至行尾,并把 modified 属性设为 false。使用 setText()插入文本内容时,不会被验证。若文本过长,则截断为 maxLength()的长度。默认值为空字符串。
- ⑩、**cursorPosition**: int 访问函数: int cursorPosition() const; void setCursorPosition(int);
此属性描述当前文字光标的位置,默认为 0。
- ⑪、**cursorMoveStyle**: Qt::CursorMoveStyle
访问函数: Qt::CursorMoveStyle cursorMoveStyle() const;
void setCursorMoveStyle(Qt::CursorMoveStyle);
● 此属性描述文字光标(插入符)的移动样式,Qt::CursorMoveStyle 枚举可取两个值,其意义如下
■ Qt::LogicalMoveStyle: 在从左向右显示的文本中,当按下左箭头键时光标向左移动,但在从右向右显示的文本中,按下左箭头键时光标向右移动。按下右箭头键的行为类似。
■ Qt::VisualMoveStyle: 无论是从左向右还是从右向左显示的文本,按下左箭头键时方向都向左移动,按下右箭头键时都向右移动。
- ⑫、**dragEnabled**: bool 访问函数: bool dragEnabled() const; void setDragEnabled(bool);
此属性描述是否启用拖动功能(即在选定的文本上按下鼠标并移动时可以移动或复制选定的文本),默认是禁用的。
- ⑬、**readOnly**: bool 访问函数: bool isReadOnly() const; void setReadOnly(bool);
此属性描述行编辑器是否是只读模式,在只读模式下,仍然可以复制文本,但不能编辑文本,而且不会显示文字光标。默认为 false。
- ⑭、**undoAvailable**: const bool 访问函数: bool isUndoAvailable() const;
此属性描述撤消(undo)是否可用,只要用户修改了行编辑器中的文本,则撤消就变成

可用的了。

⑮、**redoAvailable**: const bool

访问函数: bool isRedoAvailable() const;

此属性保存重做(redo)是否可用, 当在行编辑器中执行了一次或多次撤消(undo)操作, 则重做就会变得可用, 默认为 false。

⑯、**echoMode**: EchoMode

访问函数: EchoMode echoMode() const; void setEchoMode(EchoMode);

此属性描述行编辑器的回显模式, 回显模式是指行编辑器如何将用户输入的文本显示(或回显)给用户, QLineEdit::EchoMode 枚举对回显模式进行了描述, 见下表

QLineEdit::EchoMode 枚举(无标志)

成员	值	说明
QLineEdit::Normal	0	逐字显示输入的字符(默认值)
QLineEdit::NoEcho	1	不显示任何内容(可用于密码长度也需保密的密码)
QLineEdit::Password	2	显示与平台相关的密码掩码字符(比如将输入的内容显示为"*")。
QLineEdit::PasswordEchoOnEdit	3	在编辑时, 显示输入的字符, 否则以 Password 的形式显示字符。

⑰、**displayText**: const QString

访问函数: QString displayText() const;

- 此属性用于获取回显(显示)的文本,
- 若 echoMode 属性为 Normal 则返回的内容与 text()函数相同,
- 若 echoMode 属性为 Password 或 PasswordEchoOnEdit, 则返回一串与平台相关的长度为 text().length()的密码掩码字符, 比如返回"*****"。
- 若 echoMode 属性是 NoEcho, 则返回一个空字符串。默认为空字符串。

⑱、**inputMask**: QString

访问函数: QString inputMask() const; void setInputMask(const QString &);

- 此属性描述输入掩码,
- 此处掩码的作用是限制用户输入的文本(比如只能输入数字而不能输入字母等)。
- 若未设置掩码, 则返回空字符串, QValidator 类可以代替掩码的功能。
- 设置空字符串可以取消掩码的设置。空格字符是空白的默认字符。
- 设置掩码的方法如下:
 - 行编辑器的掩码需要对每一位输入的字符使用掩码字符(见下表)进行逐位设置。比如 setInputMask("AAAAA");表示只能在行编辑器中的前 5 位输入 ASCII 字母, 不能输入数字或其他字符。再如常见的 MAC 地址的输入就可以使用 setInputMask("HH:HH:HH:HH:HH:HH");的掩码设置。
 - 在掩码字符的分号之后的字符表示在行编辑器中输入空白时替换的字符。比如 setInputMask("AAAAA;_");表示输入的空白使用 "_" 替换。
 - 注: 空格需要选中要输入的那一位, 然后才能输入空格。
- 下表为设置掩码时使用的掩码字符及其限制

掩码字符	意义
注: “不是必须的”: 典型的用法是若该位不是必须的, 则可以使用空格设置该位的值, hh:HH 则在 h 处可以使用空格, 而在 H 处则不能是空格字符(必须是 16 进制字符)	
A	限制为 ASCII 字母字符 a-z, A-Z
a	允许 ASCII 字母字符, 但不是必须的

N	限制为 ASCII 字母字符和数字 a-z, A-Z, 0-9。
n	允许 ASCII 字母字符和数字, 但不是必须的
X	任何字符
x	任何字符, 但不是必须的
9	限制为 ASCII 数字 0-9。
0	允许 ASCII 数字, 但不是必须的
D	限制为 ASCII 数字 1-9。
d	允许 ASCII 数字(1-9), 但不是必须的。
#	允许 ASCII 数字和正负号, 但不是必须的。
H	限制为 16 进制字符, 即 0-9, A-F, a-f
h	允许 16 进制字符, 但不是必须的。
B	限制为 2 进制字符, 即只能是 0 或 1。
b	允许 2 进制字符, 但不是必须的。
>	此符号之后使用掩码字符限制的输入字符都被自动转换为大写字母
<	此符号之后使用掩码字符限制的输入字符都被自动转换为小写字母
!	关闭大小写转换
[] {}	保留
\	转义字符

⑲、**modified: bool** 访问函数: bool isModified() const; void setModified(bool);

此属性描述, 行编辑器的内容是否被用户修改了。调用 setText()函数会将该属性重置为 false。默认为 false。

示例 18: QLineEdit 类(行编辑器)的属性

//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>
#include<QDebug>
class B:public QLineEdit{    Q_OBJECT
public:    B(QWidget *p=0):QLineEdit(p) {}    B(QString s, QWidget *p=0):QLineEdit(s, p) {}
public slots:
void f() {    qDebug()<<displayText(); }    //qDebug() 函数的用法类似于 cout
void g() {    qDebug()<<text(); }    //f 和 g 槽, 用于验证 text 和 displayText 的区别
void h(QString s) {    //用于设置回显模式。
if(s=="Normal") setEchoMode(Normal);
if(s=="NoEcho") setEchoMode(NoEcho);
if(s=="Password") setEchoMode>Password);
if(s=="PasswordEchoOnEdit") setEchoMode>PasswordEchoOnEdit);    }
void j() {    QString s=text();    setInputMask(s);    }    //设置掩码
void k() {    setInputMask("");    }    //清除设置的掩码
};
#endif // M_H
```

//m.cpp 文件的内容。

```
#include "m.h"
int main(int argc, char *argv[]) {    QApplication a(argc, argv);
    QWidget w;
    QPushButton *pb=new QPushButton("display",&w);
```



```

QPushButton *pb1=new QPushButton("text",&w);
QPushButton *pb2=new QPushButton("mask",&w);
QPushButton *pb3=new QPushButton("clearmask",&w);
pb->move(22,22);    pb1->move(22,44);    pb2->move(22,66);    pb3->move(22,88);

QComboBox *pc=new QComboBox(&w);    pc->move(105,22); //使用组合框设置回显模式
pc->addItem("Normal");    pc->addItem("NoEcho");
pc->addItem("Password");    pc->addItem("PasswordEchoOnEdit");
B *pel=new B("124444",&w);    pel->move(105,88);    pel->resize(77,22);
//关联信号与槽
QObject::connect(pb, &QPushButton::clicked, pel, &B::f);
QObject::connect(pb1, &QPushButton::clicked, pel, &B::g);
QObject::connect(pb2, &QPushButton::clicked, pel, &B::j);
QObject::connect(pb3, &QPushButton::clicked, pel, &B::k);
QObject::connect(pc, SIGNAL(activated(const QString)), pel, SLOT(h(QString)));
w.resize(300,200);    w.show();    return a.exec();    }

```

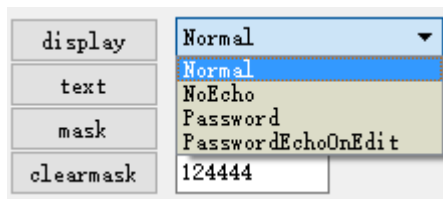
运行结果及说明

显示回显文本

显示文本


设置掩码

清除掩码



组合框的下拉列表
可以选择行编辑器
的回显模式

- 1、在组合框中选中"Normal"设置行编辑器为 Normal 回显模式
点击 display 和 text 按钮，此时都会输出图中的 124444
- 2、在组合框中选中"Password"设置行编辑器为 Password 回显模式
点击 display 会输出●●●●●●，点击 text 按钮，仍输出 124444
- 3、在组合框中选中"Normal"，然后在行编辑器中输入 HH:HH:HH，然后点击 mask 按钮，将该字符串

作为掩码字符设置行编辑器的输入掩码，设置之后的行编辑器样式为 ，此时只能在行编辑器中输入 16 进制的数字，其他字符不能被输入，点击 clearmask 按钮，可清除设置的掩码。

6、QLineEdit 类中的函数

1)、构造函数

```
QLineEdit(QWidget* parent = Q_NULLPTR);
```

```
QLineEdit(const QString &contents, QWidget* parent = Q_NULLPTR);
```


2)、用于编辑的函数

- ①、void copy() const; //槽，复制
- ②、void cut(); //槽，剪切，若当前验证器不允许删除，则 cut()将复制而不删除
- ③、void paste(); //槽，粘贴，若粘贴的文本不被验证器接受，则什么也不做。
- ④、void insert(const QString &newText); //插入
- ⑤、void backspace(); //删除左侧或所选择的文本。
- ⑥、void del(); //删除右侧或所选择的文本。

- ⑦、`void clear();` //清除行编辑器的内容。
- ⑧、`void redo();` //槽，若重做可用，则重做最后的操作
- ⑨、`void undo();` //槽，若撤消可用，则取消最后一次的操作。

3)、与选择有关的函数

- ①、`void setSelection(int start, int length);` 从位置 `start` 开始选择长度为 `length` 的文本，长度允许是负的
- ②、`void deselect();` 取消选定的文本
- ③、`void selectAll();` //槽，选择所有文本。
- ④、`int selectionLength() const;` //返回选择的文本长度，qt5.10
- ⑤、`int selectionStart() const;`
`int selectionEnd() const;` //qt5.10

返回行编辑器中“第一个选定的字符”或“最后一个选定的字符之后”的索引，若未选择文本，则返回-1。，如图，`selectionStart()`返回 2，而 `selectionEnd()`返回 6。

4)、与文字光标有关的函数

- ①、`void cursorPositionAt (const QPoint &pos);` //返回位于 `pos` 处的文字光标的位置
- ②、`void end(bool mark);`
`void home(bool mark);`
 将文字光标移到开头/行尾。若 `mark` 为 `true`，则从当前位置选择直到第一/最后一个位置的文本，否则取消已选定的文本。
- ③、`void cursorBackward (bool mark, int steps = 1);`
`void cursorForward (bool mark, int steps = 1);`
 将文字光标向右(Forward)/向左(Backward)移动 `steps` 个字符，若 `mark` 为真，则选择每个移过的字符，否则清除已选定的文本。
- ④、`void cursorWordBackward (bool mark);`
`void cursorWordForward (bool mark);`
 向右(Forward)/向左(Backward)将文字光标移动一个单词，若 `mark` 为 `true`，则选中该单词，否则取消被选中的内容。

5)、以下函数用于设置和获取页边距

```
void setTextMargins(int left, int top, int right, int bottom);
void setTextMargins(const QMargins &margins);
void getTextMargins(int *left, int *top, int *right, int *bottom) const;
QMargins textMargins() const;
```

6)、以下函数用于设置和获取验证器及自动完成器。

```
void setValidator( const QValidator* v);
const QValidator* validator() const;
void setCompleter(QCompleter* c);
```

```
QCompleter* completer() const;
```

7)、与上下文菜单有关的函数(即点击鼠标右键时弹出的菜单)

①、QMenu* createStandardContextMenu();

该函数创建了一个标准的上下文菜单，这个函数默认是由 contextMenuEvent() 事件处理函数调用的。

②、virtual void contextMenuEvent(QContextMenuEvent* event) //受保护的

- 该函数是 QWidget::contextMenuEvent() 函数的重新实现。
- 该函数的作用是显示由 createStandardContextMenu() 函数创建的标准上下文菜单，
- 若要禁用行编辑器的上下文菜单，可以设置 QWidget::contextMenuPolicy 属性的值为 Qt::NoContextMenu。
- 若要使用自定义的上下文菜单，则可以重新实现此函数。
- 若要扩展标准的上下文菜单，则重新实现该函数，并向 createStandardContextMenu() 函数返回的菜单中添加内容，然后返回修改后的菜单。代码大致如下

```
void contextMenuEvent(QContextMenuEvent* event) {  
    QMenu *menu = createStandardContextMenu();  
    menu->addAction("XXX");  
    .....  
    menu->exec(event->globalPos());  
    delete menu; }  

```

7、QLineEdit 类中的信号

①、void cursorPositionChanged(int old, int new); //信号

当文字光标移动时就会发送此信号，old 是移动前的位置，new 是移动后的位置。

②、void editingFinished(); //信号

当按下 enter 键或行编辑器失去焦点时，发送此信号。若在行编辑器中设置了验证器或掩码，则只有当输入的内容符合掩码的要求或验证器返回 QValidator::Acceptable 时，才会发送该信号。比如 setInputMask("AAAA"); 则只有在输入 4 位正确的字符之后才会发送此信号，若只正确输入了 3 位或 2 位则不会发送此信号。

③、void returnPressed() //信号

当按下 enter 键时(注意：失去焦点时不会发送)，发送此信号。若在行编辑器中设置了验证器或掩码，则只有当输入的内容符合掩码的要求或验证器返回 QValidator::Acceptable 时，才会发送该信号(具体见 editingFinished 信号)。

④、void selectionChanged(); //信号

只要选择发生了改变，就会发送此信号。注意：仅仅移动文字光标，而未使选择的内容发生改变，则不会发送此信号。

⑤、void textChanged(const QString &text); //信号

当文本改变时发送此信号，text 参数保存的是新的文本。当以编程的方式更改文本时(比如调用 setText())，也会发送此信号。

⑥、void textEdited(const QString &text); //信号

当文本被编辑时发送此信号，text 参数保存的是新的文本。当编程的方式更改文本时

(比如调用 `setText()`), 不会发送此信号。
以上函数及信号都比较简单, 示例略。

三、QValidator 抽象类(验证器)及其子类

- 1、QValidator 类直接继承自 QObject 类, 且是一个抽象类, 因此具体功能主要由其子类来实现, 或者子类化该类实现自定义的验证器。
- 2、验证器的作用是验证用户输入的字符是否符合规定的要求, 比如若要求用户只能输入数字而不能输入字母就可使用验证器来实现, QLineEdit 类中的输入掩码也可实现类似的功能。
- 3、验证器通常配合 QLineEdit、QSpinBox、QComboBox 一起使用,
- 4、验证器把用户输入的数据分为三个状态, 如下:

无效的(Invalid): 是指输入的字符是无效的。

中间状态(Intermediate): 是指输入的字符既不是明显的无效, 也不是最终可接受的结果。

可接受的(Acceptable): 是指输入的字符串作为最终结果是可接受的。

比如对于接受从 10 到 100 的整数行编辑器, 40 和 22 是可接受的, "aa"、1111 是无效的, 而 5.6 是中间状态, 因为用户可还可能在其后输入新的数字使其变为可接受的或无效的。

5、QValidator 类提供的函数如下:

①、`QValidator(QObject* parent = Q_NULLPTR);` //构造函数

②、`virtual State validate(QString &input, int &pos) const = 0;` //纯虚函数

子类必须实现该函数, State 是 QValidator 类中定义的枚举, 其成员用于描述验证字符串的三种状态, 取值如下:

QValidator::State 枚举

成员	值	说明
QValidator::Invalid	0	无效的
QValidator::Intermediate	1	中间状态
QValidator::Acceptable	2	可接受的

③、`void fixup(QString &input) const;`

该函数的主要功能是, 根据验证器的规则更改用户的输入, 更改后的字符串不一定是有效的字符串, 比如在只接受整数的行编辑器中, 可能会删除除数字之外的其他所有字符, 即使删除后的字符可能仍是无效的。默认情况下, 该函数什么也不做(未实现)

④、`QLocale locale() const;`

`void setLocale(const QLocale& locale);`

以上函数用于设置和返回用于验证器的区域设置(语言环境)。默认为使用 QLocale::setDefault()设置的默认区域设置, 若未设置默认区域设置, 则是操作系统的区域设置。不同的区域对字符的表示形式是不同的, 比如在德国 11.11 是指的 11.11。

6、QDoubleValidator 类:

该类是 QValidator 抽象类在 Qt 中的一个内置实现，用于提供对浮点数范围的检查，该类未提供 fixup() 函数。该类具有如下的属性和函数

- ①、**bottom**: double 访问函数: double bottom() const; void setBottom(double);
验证器的最小可接受值，默认为负无穷大。
- ②、**decimals**: int 访问函数: int decimals() const; void setDecimals(int);
验证器可接受的小数点后的最大位数，默认为 1000。
- ③、**notation**: Notation 访问函数: Notation notation() const; void setNotation(Notation);
该属性描述如何表示浮点数(使用科学计数法还是小数点)，默认为 ScientificNotation(科学计数法)。其中 QDoubleValidator::Notation 枚举可取值为
 - QDoubleValidator::StandarNotation: 以标准形式表示浮点数(即小数点形式)
 - QDoubleValidator::ScientficNotation: 以科学计数法表示浮点数，比如 1.2E-3。
- ④、**top**: double 访问函数: double top() const; void setTop(double);
验证器的最大可接受值，默认为无穷大。

⑤、构造函数

QDoubleValidator(QObject* parent = Q_NULLPTR);

QDoubleValidator(double bottom, double top, int decimals, QObject* parent = Q_NULLPTR);

其中参数 bottom 和 top 用于指定能够接受的浮点数的最大和最小值，decimals 表示能够接受的浮点数的小数位数。

- ⑥、virtual void **setRange**(double bottom, double top, int decimals = 0);
用于设置可接受的浮点数的范围。其参数意义见构造函数或相应的属性。
- ⑦、virtual QValidator::State **validate**(QString &input, int &pos) const;
这是对 QValidator::validate() 的重新实现。该函数根据 input 的值是否在验证器设置的浮点数的范围之内，返回 Invalid(无效的)、Intermediate(中间状态)、Acceptable(可接受的)。注意，若使用的是科学计数法表示浮点数，则输入的数值虽然明显超出了范围，但是该值可通过更改指数而变为有效的。默认情况下，参数 pos 未使用。

7、QIntValidator 类:

该类是 QValidator 抽象类在 Qt 中的一个内置实现，用于提供对整数范围的检查。该类具有如下的属性和函数

- ①、**bottom**: int 访问函数: int bottom() const; void setBottom(int);
验证器的最小可接受值，默认为-2147483647。
- ②、**top**: int 访问函数: int top() const; void setTop(int);
验证器的最大可接受值，默认为可用的最高有符号整数，通常为 2147483647。
- ③、构造函数
QIntValidator(QObject* parent = Q_NULLPTR);
QIntValidator(int minimum, int maximum, QObject* parent = Q_NULLPTR);
其中参数 minimum 和 tmaximum 用于指定能够接受的整数的最小和最大值。
- ④、virtual void **setRange**(int bottom, int top); 设置可接受的整数位于 bottom 和 top 之间。
- ⑤、virtual void **fixup**(QString &input) const; QValidator::fixup() 的重新实现。

⑥、`virtual QValidator::State validate(QString &input, int &pos) const;`

这是对 `QValidator::validate()` 的重新实现。该函数根据 `input` 的值是否在验证器设置的整数的范围之内，返回 `Invalid`(无效的)、`Intermediate`(中间状态)、`Acceptable`(可接受的)。默认情况下，参数 `pos` 未使用。

8、`QValidator` 类还有两个子类，分别是 `QRegExpValidator` 和 `QRegularExpressionValidator`，这两个类用于验证正则表达式。

示例 19：为 `QLineEdit` 添加验证器

```
#include<QtWidgets>
#include <iostream>
using namespace std;
int main(int argc, char *argv[]) {    QApplication a(argc, argv);
    QWidget w;
    QLineEdit *pe1=new QLineEdit(&w);    pe1->move(22,22);    pe1->resize(77,22);
    QIntValidator *vi=new QIntValidator(10,900,&w); //使整数范围位于 10~900 之间。
    pe1->setValidator(vi);    /*安装验证器 vi 后，pe1 将只能输出比 900 更小的整数，且不能输入字母
                                等非数字。*/
    QString s="1111";int i=0;
    cout<<vi->validate(s,i)<<endl; //输出 0，s 表示的字符串是无效整数。

    QLineEdit *pe2=new QLineEdit(&w);    pe2->move(111,22);    pe2->resize(77,22);
    QDoubleValidator *vd=new QDoubleValidator(1,11,3,&w);
    vd->setNotation(QDoubleValidator::StandardNotation); /*使用标准方式表示浮点数，否则即使输入
                                很大的浮点数也不会超出范围(因为可更改指数使其变为有效)*/
    pe2->setValidator(vd);    //安装验证器
    w.resize(300,200);    w.show();    return a.exec();    }
```

运行结果及说明

该输入框只能输入比 900 更小的数，且不能输入非数字的字符



该输入框只能两位整数位数的值，小数的位数不能超过 3 位，且不能输入非数字的字符(小数点除外)

4.5 旋转框(微调按钮)

QAbstractSpinBox、QSpinBox、QDoubleSpinBox

一、QAbstractSpinBox 类(旋转框或微调框)

- 1、QAbstractSpinBox 类是 QWidget 类的直接子类，虽然该类不是抽象类，但该类并未提供实际的功能，仅为旋转框提供了一些外观的形式以及需要子类实现了成员，也就是说点击微调按钮的上/下按钮，不会使其中的数值有变化。实际的功能是由该类的子类提供的，用户也可继承该类实现自定义的功能。
- 2、旋转框是由微调按钮(用于调整值)和行编辑器(用于显示值)组成的(见右图)
- 3、步长：是指当使用微调按钮的箭头增加/减少值时，该值将会增加/减少步长的数量，如若当前值为 10，步长为 2，则使用向上箭头调整其值时，该值将增长为 12。



3、QAbstractSpinBox 类中的属性

QAbstractSpinBox 类(属性速查表)

属性名	说明	属性名	说明
accelerated	是否加快调整速度	keyboardTracking	是否启用键盘跟踪
acceptableInput	是否符合验证的要求	readOnly	是否为只读
alignment	对齐方式	showGroupSeparator	是否显示千位分隔符
buttonSymbols	微调按钮的形式	specialValueText	设置和返回特殊值文本
correctionMode	中间值的更正模式	text	获取旋转框中的文本(包括前/后缀)
frame	是否绘制默认边框	wrapping	调整值时是否可以循环

- ①、**accelerated**: bool 访问函数: bool isAccelerated() const; void setAccelerated(bool);
此属性用于描述当按下旋转框的“向上/向下”按钮一段时间不放时，是否会加快调整旋转框中数值增加/减少的速度。默认为 false。
- ②、**acceptableInput**: bool 访问函数: bool hasAcceptableInput() const;
此属性用于获取输入的值是否满足当前验证器的要求。
- ③、**alignment**: Qt::Alignment
访问函数: Qt::Alignment alignment() const; void setAlignment(Qt::Alignment);
对齐方式，默认值为 Qt::AlignLeft。Qt::Alignment 枚举见“部件公共枚举”章节。
- ④、**buttonSymbols**: ButtonSymbols
访问函数: ButtonSymbols buttonSymbols() const; void setButtonSymbols(ButtonSymbols);
此属性用于设置当前旋转框右侧微调按钮的外观样式，默认值为 UpDownArrows。注意：有可能会以相同的方式显示 PlusMinus 和 UpDownArrows。其中 ButtonSymbols 枚举见下表

QAbstractSpinBox::ButtonSymbols 枚举(无标志)

用于描述旋转框右侧微调按钮的外观样式

成员	值	说明
QAbstractSpinBox::UnDownArrows	0	箭头形式
QAbstractSpinBox::PlusMinus	1	+或-符号的形式
QAbstractSpinBox::NoButtons	2	不显示按钮



⑤、correctionMode: CorrectionMode

访问函数: CorrectionMode correctionMode() const; void setCorrectionMode(CorrectionMode);

当编辑完成后,更正中间值的模式。默认为 QAbstractSpinBox::CorrectToPreviousValue。

枚举 CorrectionMode 见下表

QAbstractSpinBox::CorrectionMode 枚举(无标志)

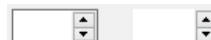
用于描述当编辑完成后,更正中间值的模式

成员	值	说明
QAbstractSpinBox::CorrectToPreviousValue	0	恢复到最后一个有效值
QAbstractSpinBox::CorrectToNearestValue	1	恢复到最近的有效值

⑥、frame: bool 访问函数: bool hasFrame() const; void setFrame(bool);

此属性描述是否绘制旋转框默认的边框

绘制



未绘制

⑦、keyboardTracking: bool

访问函数: bool keyboardTracking() const; void setKeyboardTracking(bool);

- 是否启用键盘跟踪(默认为启用)。
- 若启用了键盘跟踪,则当用户从键盘输入新值时,每键入一个值旋转框就会发送一个 valueChanged()信号。比如当用户通过键盘 6, 0, 0 而输入值 600 时,旋转框发出 3 个信号,其值分别为 6, 60, 600。
- 若禁用键盘跟踪,则旋转框不会在键入时立即发送 valueChanged()信号,而是在按下 enter 键、失去键盘焦点等时候发送该信号。

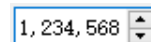
⑧、readOnly: bool 访问函数: bool isReadOnly() const; void setReadOnly(bool);

是否启用只读模式(默认为 false)。只读模式不能更改值(但仍可选择和复制),也不会显示文字光标。

⑨、showGroupSeparator: bool //qt5.3

访问函数: bool isGroupSeparatorShow() const; void setGroupSeparatorShow(bool);

此属性描述,是否启用千位分隔符(默认为 false),即右图中数字下的逗号。



注意: Qt 在输入数值后需要点击一下微调按钮才会显示出千位分隔符。

⑩、specialValueText: QString

访问函数: QString specialValueText() const; void setSpecialValueText(const QString&);

- 此属性描述旋转框的特殊值文本(默认为没有特殊值文本)。
- 特殊值文本是指,当值等于最小值时旋转框显示这个特殊值文本而不显示数值。
- 特殊值文本会被 QSpinBox::valueChanged()信号传递。

- 使用空字符串设置特殊值文本，可关闭特殊值文本的显示。
- 若未设置特殊值文本，则该属性为空字符串

- ⑪、**text**: const QString **访问函数**: QString text() const;
获取旋转框中的文本，包括前缀和后缀。
- ⑫、**wrapping**: bool **访问函数**: bool wrapping() const; void setWrapping(bool);
此属性描述旋转框在调整值时是否可以循环(默认为 false)，若该属性为 true，则当使用微调按钮将值增加到最大值时，会循环至最小值，反之亦然。只有在设置了最大值和最小值时才有意义。

4、QAbstractSpinBox 类中的函数

- ①、**QAbstractSpinBox**(QWidget* parent = Q_NULLPTR); //构造函数
- ②、**void selectAll**(); //槽，选中除了前缀和后缀以外的所有文本
- ③、**virtual void clear**() //槽，虚拟的，清除行编辑器中的所有文本，但前缀和后缀除外。
- ④、**virtual void stepBy**(int steps); //虚拟的
当激活微调按钮的向上/向下箭头时会调用此虚函数，该函数会把当前值增加/减少 steps，其实 steps 就是步长。比如当前值为 10，则调用 stepBy(3);之后的值为 13。子类化 QAbstractSpinBox 必须实现此函数，注意：即使结果值超出了最小值和最大值的范围，此函数仍会被调用。
- ⑤、**void stepDown**(); //槽
降低一个步长，类似于调用 stepBy(-1);准确的说是 stepBy(steps);因为 steps 的值不一定是-1，比如，对于 QSpinBox 若把 singleStep 属性设置为 2，则调用该函数将使用减少 2。
- ⑥、**void stepUp**(); //槽
提高一个步长，类似于调用 stepBy(1);准确的说是 stepBy(steps);见 stepDown()。
- ⑦、**virtual QValidator::State validate**(QString &input, int &pos) const; //虚拟的
此虚函数用于验证 input 是否有效。该虚函数在各子类中被重新实现。
- ⑧、**virtual void fixup**(QString &input) const; //虚拟的
若按下 return 或调用 interpretText()时，对 input 的验证不为 QValidator::Acceptable(可接受)时，则调用此虚函数，该函数会尝试更正文本以使其有效，该函数在各子类中被重新实现。
- ⑨、**virtual StepEnabled stepEnabled**() const; //受保护的
此函数决定了微调按钮向上/向下是否合法。若子类化 QAbstractSpinBox，需要实现此函数。StepEnabled 标志见下表

QAbstractSpinBox::StepEnabledFlag 枚举
标志为: QAbstractSpinBox::StepEnabled

成员	值
QAbstractSpinBox::StepNone	0x00
QAbstractSpinBox::StepUnEnabled	0x01
QAbstractSpinBox::StepDownEnabled	0x02

⑩、`QLineEdit* lineEdit() const;` //受保护的

`void setLineEdit(QLineEdit* lineEdit);` //受保护的

以上函数用于获取和设置旋转框的行编辑器。其中，参数 `lineEdit` 不能为 0。若设置的新的行编辑器 `lineEdit` 的 `QLineEdit::validator()` 函数返回 0(即未安装验证器)，则将在新的行编辑器中使用旋转框内部的验证器。

⑪、`virtual void keyPressEvent(QKeyEvent* event);` //受保护的

这是对 `QWidget::keyPressEvent()` 的重新实现，该函数重新实现了以下功能

按键	说明
向上箭头	这会调用 <code>stepBy(1)</code> ，表示调整值增加 1
向下箭头	这会调用 <code>stepBy(-1)</code> ，表示调整值减少 1
Page up	这会调用 <code>stepBy(10)</code> ，表示调整值增加 10
Page down	这会调用 <code>stepBy(-10)</code> ，表示调整值减少 10

⑫、`void editingFinished();` //信号

当编辑完成时发送该信号，即当旋转框失去焦点或按下 `enter` 键时，会发送该信号。

二、QSpinBox 类

- 1、QSpinBox 类是 QAbstractSpinBox 类的直接子类 and 具体实现，
- 2、QSpinBox 类被设计用于处理整数和离散值集合，对于浮点值使用 QDoubleSpinBox 类实现。
- 3、QSpinBox 默认只支持整数值，但可通过其内部的成员函数进行扩展，以支持使用不同的字符串。

3、QSpinBox 类中的属性

QSpinBox 类(属性速查表)			
属性名	说明	属性名	说明
<code>cleanText</code>	获取文本(不包括前/后缀)	<code>prefix</code>	设置和获取前缀
<code>displayIntegerBase</code>	设置和获取数值的基数	<code>suffix</code>	设置和获取后缀
<code>maximum</code>	设置和获取最大值	<code>singleStep</code>	设置和获取步长
<code>minimum</code>	设置和获取最小值	<code>value</code>	设置和获取值

①、`cleanText: const QString` 访问函数: `QString cleanText() const;`


获取旋转框中的文本，不包括前缀、后缀及前后的空格。使用 `QAbstractSpinBox::text` 属性可以获取包括前/后缀的文本。

②、`displayIntegerBase: int` //qt5.2

访问函数: `int displayIntegerBase() const; void setDisplayIntegerBase(int);`

设置和获取旋转框中的值的基数(支持 2~36 进制)，默认为 10(即 10 进制)

③、`maximum: int` 访问函数: `int maximum() const; void setMaximum(int);`

- ④、**minimum**: int 访问函数: int minimum() const; void setMinimum(int);
以上两属性用于设置和获取旋转框中的最大值(默认为 99)和最小值(默认为 0)
- ⑤、**prefix**: QString 访问函数: QString prefix() const; void setPrefix(const QString &);
- ⑥、**suffix**: QString 访问函数: QString suffix() const; void setSuffix(const QString &);
- 以上属性用于设置和获取旋转框的前缀(默认无前缀)和后缀(默认无后缀), 见右图
 - 前/后缀位于显示的值的前/后面, 常见用途是显示度量单位或货币的符号。
 - 设置空字符串可以关闭前/后缀,
 - 若未设置前/后缀则该属性是一个空字符串,
 - 若设置了 **specialValueText** 属性(特殊值文本), 当显示的值为最小值时, 不显示前/后缀, 而显示特殊值文本。
- 
- ⑦、**singleStep**: int 访问函数: int singleStep() const; void setSingleStep(int);
设置和获取旋转框的步长值, 默认为 1, 若该值设置为负数, 则该函数不起作用。可以重载虚函数 **stepBy()** 实现自己的策略。这里需说明 **QSpinBox** 实现的 **stepBy(int steps)** 虚函数的步长是 **steps*singleStep**, 比如, 若 **singleStep** 属性设置为 3, 则调用 **stepBy(4)**; 将使值增加 12, 可以这样理解, **singleStep** 才是步长, 而 **stepBy(4)** 表示将值增长 4 个步长的大小, 同理 **singleStep** 属性还会影响到 **stepUp()** 和 **stepDown()** 函数, 详见 **QAbstractSpinBox::StepUp()** 函数的讲解。
- ⑧、**value**: int
访问函数: int value() const; void setValue(int);
信号: **valueChanged**(int); void valueChanged(const QString &);
获取和设置旋转框的值, 若新值与旧值不同, 将发送 **valueChanged** 信号, 该信号的 **QString** 类型的形参, 保存的值包括前缀和后缀。

4、QSpinBox 类中的函数

- ①、**QSpinBox**(QWidget* parent = Q_NULLPTR); //构造函数
构造一个旋转框, 其值的范围是 0~99, 步长为 1, 初值为 0。
- ②、void **setRange**(int min, int max);
设置最大值和最小值, 这是属性 **maximum** 和 **minimum** 的便捷设置函数。
- ③、virtual QString **textFromValue**(int value) const; //虚拟的, 受保护的
当需要显示给定的值 **value** 时, 旋转框就会调用该虚函数, 默认实现是返回一个字符串, 且不会为 **specialValueText** 属性(特殊文本值)调用此函数, 且返回值中没有前/后缀。若重新实现此函数, 通常还需要重新实现 **valueFromText()** 和 **validate()** 函数。
- ④、virtual int **valueFromText**(const QString &text) const; //虚拟的, 受保护的
当需要将输入的文本 **text** 解释为值时, 旋转框就会调用此虚函数, 若以非数字形式显示旋转框中的值时, 就需要重新实现此函数。注意: **specialValueText** 属性(特殊文本值)不在此函数内处理。
- ⑤、void **valueChanged**(int i); //信号
void **valueChanged**(const QString &text); //信号

只要旋转框的值发生变化就会发送以上信号，新值在 i 中传递。其中 text 是包含前/后缀的新值。

示例 20: QSpinBox 类(旋转框)的使用

//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>
#include <iostream>
using namespace std;

class B:public QSpinBox{    Q_OBJECT
public:    B(QWidget *p=0):QSpinBox(p) {}
        int j;    //用于保存自定义的步长

public slots:
void setlong(int i) {j=i;} //使用该函数设置自定义的步长，可以为负值。
void stepBy(int s) {    //当激活微调按钮的上/下箭头时会自动调用该函数。
    //Qt 会自动获取 s 的值，当点击旋转框的上箭头时 s=1，点击下箭头时 s=-1
    //若按下的是 Page up 键，则 s=10，同理 Page down 时 s=-10。
    QSpinBox::stepBy(s*j); //设置自定义的步长。
}

QString textFromValue( int value) const{//当需要显示给定的值 value 时，会调用该函数
    QVariant v=value;    /*Qt 会自动获取旋转框中输入的值。使用 QVariant 类，可方便的在各种类型之间进行转换。*/
    QString s="YY"+v.toString()+"XX"; //连接 YY 与 v 转换之后的字符串
    //cout<<s.toString()<<endl;    //用于测试。
    return s; //字符串 s 会在旋转框中显示。
}

void f(QString t){ cout<<t.toString()<<endl; }//t 含有旋转框内的文本(包括前/后缀)
};

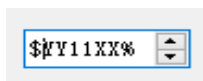
#endif // M_H
```

//m.cpp 文件的内容

```
#include "m.h"
int main(int argc, char *argv[]){    QApplication a(argc,argv);
    QWidget w;    B *psl=new B(&w);    psl->resize(77,22);    psl->move(22,55);
    psl->setRange(0,10000000);    //设置旋转框中的值的范围
    psl->setAccelerated(1);    //开启加速调整。
    psl->setValue(11);    //设置初始值为 11。
    psl->setSpecialValueText("XXX");    //设置特殊文本值(即最小值时显示的文本)
    psl->setWrapping(1);    //开启调整值循环，即继续增长最大值时，循环至最小值。
    psl->setPrefix("$");    //设置前缀
    psl->setSuffix("%");    //设置后缀
    //psl->setKeyboardTracking(false); //禁用键盘跟踪，默认为 true，该设置主要影响
    //valueChanged 信号的发送时机，读者可自行验证。
    /* psl->setDisplayIntegerBase(16);    设置为 16 进制。设置该函数后，旋转框对值的显示会出问题，
    因为重新实现的 textFromValue() 函数，未对进制的显示进行处理。*/
    psl->setlong(-2);    /*使用自定义的函数设置负值步长，设置后点击上箭头会使值减少，下箭头使值
    增长。*/
```

```
cout<<psl->cleanText().toStdString()<<endl; //输出旋转框内的文本, 不含前/后缀
cout<<psl->text().toStdString()<<endl; //输出旋转框内的文本, 含前/后缀
QObject::connect(psl, SIGNAL(valueChanged(const QString)), psl, SLOT(f(QString)));
w.resize(300,200); w.show(); return a.exec(); }
```

运行结果及说明



- 1、首次运行时会输出"YY11XX"和"\$YY11XX%"
- 2、前缀为\$, 后缀为%, 前后缀无法改变。
- 3、当点击上箭头时中间的数值 11 会以 2 的步长减少, 下箭头会以 2 的步长增加。
- 4、输入值时需要把除了前/后缀的字符全都清除, 才能输入正确的值。输入值之后按下回车键, 会在输入的字符前面自动加上 YY, 在后面加上 XX。
- 5、重新实现的 textFromValue 函数不会影响到特殊值文本的显示, 也就是说, 当旋转框的值减小到最小值时, 仍然显示的是本例设置的特殊文本"XXX"
- 6、每当值发生改变时都会产生 valueChanged 信号, 对应的槽函数, 都会以字符串的形式输出改变后的值(包括前/后缀)

三、QDoubleSpinBox 类

- 1、QDoubleSpinBox 类是 QAbstractSpinBox 类的直接子类和具体实现,
- 2、QDoubleSpinBox 类被设计用于处理双精度浮点数(即 double)。
- 3、QDoubleSpinBox 类会对数字进行四舍五入, 比如, 若小数位数设置为 2, 则调用 setValue(5.5756)将得到值 5.58。
- 4、QDoubleSpinBox 类的属性与 QSpinBox 类的属性及意义是相同的, 只是 QDoubleSpinBox 类的属性的类型有些为 double 类型, 而不是 int 型, 详见 QSpinBox 类的讲解。

①、**cleanText**: const QString 获取文本(不包括前/后缀)

②、**prefix**: QString 设置和获取前缀

③、**suffix**: QString 设置和获取后缀

④、**singleStep**: double 设置和获取步长

⑤、**value**: double 设置和获取值

⑥、**maximum**: double 设置和获取最大值(默认 99.99)

⑦、**minimum**: double 设置和获取最小值(默认为 0.0)

⑧、**decimals**: int 访问函数: int decimals() const; void setDecimals(int);

设置和获取浮点数的小数位数, 最大值为 DBL_MAX_10+DBL_DIG(即 323), 小数位数是由 double 类型(IEEE754 标准)本身限制的, 这是 QDoubleSpinBox 特有的属性。默认为 2。

- 5、QDoubleSpinBox 类的函数与 QSpinBox 类是相同的, 下面仅列出其原型及必要的说明

①、**QDoubleSpinBox**(QWidget* parent = Q_NULLPTR); //构造函数

构造 QDoubleSpinBox 旋转框, 其值为 0.0~99.99, 步长为 1.0, 小数位数为 2 位, 初始值为 0.00

- ②、`void setRange(double min, double max);`
- ③、`virtual QString textFromValue(double value) const;` //虚拟的
- ④、`virtual double valueFromText(const QString &text) const;` //虚拟的
- ⑤、`void valueChanged(double i);` //信号
 `void valueChanged(const QString &text);` //信号

4.6 时间系统

QDate、QTime、QDateTime、QTimer、QDateTimeEdit、QDateEdit、QTimeEdit 等

说明：文中出现的 qint64, qint32 之类的类型，都是由 qt 使用 typedef 重新命名后的相应类型，主要是为了保证该类型在所有平台上都具有相同的长度，比如 qint32 就是 signed int 类型，但 Qt 可以保证在所有平台上 qint32 都是 32 位的长度。

一、本小节会讲解如下类

- 1、QDate、QTime、QDateTime：这三个类都是 QtCore 模块中的独立类，分别用于描述日期(即年月日)、时间(即时分秒)、日期和时间，但不负责对日期和时间的显示。
- 2、QTimer(计时器)：该类是 QObject 类的直接子类。
- 3、QDateTimeEdit、QDateEdit、QTimeEdit：这几个类是 QAbstractSpinBox 类的直接子类，主要用于显示(即可把时间显示在窗口中让用户可见)、调整和编辑日期和时间。
- 4、QCalendarWidget：该类直接继承自 QWidget 类，用于描述日历。

二、时间系统基础

- 1、测量地球的自转可以用来确定时间，为此，人们选取了两个基本参考点，即春分点和平太阳点，由此确定的时间分别称为恒星时和平太阳时。由于地球自转速率的不规则以及地球在其椭圆轨道里运动速度的不均匀，所以使用该方法测量的时间准确度不高，被后来使用的历书时所代替了。
- 2、**历书时 ET (Ephemeris Time)**：是由天体力学定律确定的时间，又称牛顿时，原则上对于太阳系中任一天体，只要掌握了其精确的运动规律，都可以用来制定历书时，比如纽康(美国天文学家)根据地球绕太阳公转运动编制的太阳历表，历书时的起始时间为 1900 年 1 月 0 日 12 时正(历书时)，即 1900 年初太阳几何平黄经为 279 度 41 分 48.04 秒的瞬间。历书时的精度也不高，现在使用的是协调世界时 UTC
- 3、**世界时 UT(Universal Time)**：即**格林尼治标准时间 GMT (Greenwich Mean Time)**，格林尼治是英国的一个天文台所在地，世界时是以地球自转为基础的以本初子午线的平子夜作为 0 时开始的平太阳时。
- 4、**协调世界时 UTC**：是以原子时秒长为基础的时间计量系统，是利用原子的能级跃迁原子秒来计时的。因此其精度比较高，原子时的起点定于世界时(UT)的 1958 年 1 月 1 日 0 时 0 分 0 秒。
- 5、**闰秒**：协调世界时与协调世界时之间的时间是有误差的，为协调世界时与协调世界时之间的时间误差，采用的措施就是加入闰秒的方法。
- 6、**Epoch**：指的是 UTC 时间的 1970 年 1 月 1 日 0 时 0 分 0 秒。这是 Unix 下使用的时间的起始点。
- 7、夏令时 DST(Daylight Savint Time)：又称为日光节约时间，是指人为的把时间向前拔快一小时，以达到节约能源的目的。这个制度需要由国家来执行，在我国夏令时目前并未实行。
- 8、**儒略日 JD(Julian Day)**：
 - 儒略日：是使用一个数字来表示当前日期的方式，这种方式方便计算，还适合在其

他日历系统之间进行转换，但是表示儒略日的数字位数比较多。比如 2000 年 1 月 1 日的 UT12:00，其儒略日为 2451545(其计算公式比较复杂，略)

- 儒略日的起点为儒略历的公元前 4713 年 1 月 1 日中午 12 点(即公历的公元前 4714 年 11 月 24 日)，要计算当年的儒略历只需简单的加上 4713 即可，比如 2013 年的儒略历为 6716 年。
- 儒略日的儒略周期为 7980 年，该数字由 $28 \times 19 \times 15$ 得来的，其中 28 表示一太阳周期(此时星期的日序与月的日序会重复)，19 表示一太阴周期(此时阴历月年的日序和月相会重复)，15 为罗马皇帝所颁布的课税周期。

8、注意：没有第 0 年，0 年被认为是无效的年份。

三、QDate 类

1、QDate 类用于描述日期(即年月日)，该类可存储的儒略日范围为 -784350574879~784354017364，约为公元前 20 亿到公元后 20 亿年。

2、QDate 按原样解释两位数的年份，比如 77，就是指的公元 77 年，而不是 1977 或 2077 年。

3、构造 QDate 类对象的函数

- ①、`QDate();` //构造一个空的日期(空日期是无效日期)
- ②、`QDate(int y, int m, int n);`
构造一个 y 年 m 月 d 日的日期。若指定的日期无效，则不设置日期，且 `isValid()` 为 `false`。注意 0 年是无效的。
- ③、`static QDate currentDate();` //静态的
使用系统时钟报道的当前日期创建一个 QDate 对象。这是常用方法。
- ④、`static QDate fromString(const QString &string, Qt::DateFormat format = Qt::TextDate);` //静态
- ⑤、`static QDate fromString(const QString &string, const QString &format);` //静态
 - 以上两函数表示以给定的格式 format 返回由字符串 string 表示的 QDate 对象，或者说把字符串以指定的格式转换为 QDate 对象。若参数 string 表示的日期不符合格式参数 format 指定的格式，则以上函数会返回一个无效的 QDate。
 - 其中字符串 string 的格式由两种方式来表示，一种是由 Qt 内部使用枚举 `Qt::DateFormat` 已定义好的格式，一种是使用字符串的形式指定。
 - 使用字符串指定日期格式
 - 其方法是在字符串中使用一些特殊的字符代表日期或时间，Qt 使用字符 d 表示“日”，“M”表示月，“y”表示年，其中一个字符表示一位数字，具体见下表

字符	说明
d	指定“日”(1~31)，不能有前导 0，比如 01 是无效的。
dd	指定“日”(01~31)，必须有前导 0，比如直接指定 1，是无效的。
ddd	缩写的本地化日期名称(例如 Mon~Sun)，可用于表示星期数
dddd	较长的本地化日期名称(例如 Monday~Sunday)，用于表示星期数
M(注意：大写)	指定“月”(1~12)，不能有前导 0，比如 01 是无效的。

MM	指定“月”(01~12)，必须有前导 0，比如直接指定 1，是无效的。
MMM	缩写的本地化月份名称(例如 Jan~Dec)
MMMM	较长的本地化月分名称(例如 January~December)
yy	使用两位数指定“年”
yyyy	使用四位数指定“年”，位数不足时需要前导 0。

- 除以上表格外的其他字符和使用单引号括起来的字符都被视为文本，使用参数 `string` 表示日期时，视为文本的字符必须原样指定，否则指定的日期就是无效的。比如

```
QDate::fromString("2002yy2XX13", "yyyy'yy'MXXdd");
//表示 2002 年 2 月 13 日，其中 yy 和 XX 必须在第一个参数原样指定
```

- 不指定前导 0 的格式，使用的是贪心算法，比如

```
QDate::fromString("320", "Md"); //无效
//试图使日期表示为 3 月 20 号，但实际是 32 月 0 日
```

- 若格式中未指定“年”，则默认值为 1900，若未指定“月”和“日”，其默认值都为 1。比如

```
QDate::fromString("2-12", "M-d"); //1900 年 2 月 12 日;
QDate::fromString("2002-22", "yyyy-dd"); //2002 年 1 月 22 日
QDate::fromString("2002-3", "yyyy-M"); //2002 年 3 月 1 日
```

- 下表为 `Qt::DateFormat` 枚举定义的日期格式

Qt::DateFormat 枚举(无标志)

作用：描述日期的格式

字符	说明
<code>Qt::TextDate</code>	Qt 的默认格式，相当于格式“ddd MMM d yyyy”，其中 ddd 和 MMM 使用的是系统区域设置的本地化名称，即 <code>QLocal::system()</code> ，建议使用缩写的英文名，比如，ddd 表示星期，应使用 Mon(周 1), Fri(周 5), MMM 表示月份，应使用 Jan(1 月), Dec(12 月)等缩写。
<code>Qt::ISODate</code>	ISO 8601 标准的格式，其格式为 yyyy-MM-dd 或 yyyy-MM-ddTHH:mm:ss (比如 2022-02-22T12:33:44);
<code>Qt::ISODateWithMs</code>	ISO 8601 标准的格式，包括毫秒
<code>Qt::SystemLocaleShortDate</code>	操作系统使用的简短格式，取决于当前系统的区域设置。
<code>Qt::SystemLocaleLongDate</code>	操作系统使用的长格式，取决于当前系统的区域设置。
<code>Qt::DefaultLocaleShortDate</code> <code>Qt::DefaultLocaleLongDate</code>	应用程序的区域设置指定的简短/长格式。取决于默认的应用程序区域设置，若未设置默认区域设置，则使用系统的区域设置。
<code>Qt::RFC2822Date</code>	RFC2282、RFC850、RFC1036 标准的格式，即 [ddd.] dd MMM yyyy hh:mm[:ss] +/-TZ 或 ddd MMM dd yyyy hh:mm[:ss] +/-TZ

4、修改 QDate 的函数

- ①、`QDate addDays(qint64 ndays) const;`
`QDate addMonths(int nmonths) const;`

`QDate addYears(int nyears) const;`

以上函数表示向当前日期增加 `ndays` 天或 `nmonths` 月或 `nyears` 年之后，并返回更改后的日期。若最后计算出来的月/日组合不存在(比如 2 月 31 日)，则以上函数会返回最近的有效日期。示例：

`QDate pd(2001, 10, 31);` //pd 为 2001 年 10 月 31 日

`QDate pd1 = pd.addMonths(4);` //pd1 理论为 2002 年 2 月 31 日，修正后实际为 2002 年 2 月 28 日

②、`bool setDate(int y, int m, int d);`

设置日期的年、月、日。若日期有效，则返回 `true`，否则返回 `false`。

5、判断日期状态的函数

①、`static bool isLeapYear(int year)` //静态

如果指定的年份 `year` 是闰年，则返回 `true`，否则返回 `false`。

②、`bool isNull() const;`

若日期为空，则返回 `true`，否则返回 `false`。空日期也是无效的。

③、`bool isValid() const;`

④、`static bool isValid(int y, int m, int d);` //静态

当日期(或指定的日期)有效，则返回 `true`，否则返回 `false`。比如

`QDate::isValid(2002, 2, 31);` //false，因为 2 月没有 31 号。

6、获取日期信息的函数

注：下文的当前日期是指调用该函数的 `QDate` 对象所表示的日期

以下函数，若当前日期无效，都会返回 0。

①、`QString toString(const QString &format) const;`

②、`QString toString(Qt::DateFormat format = Qt::TextDate) const;`

以指定的格式 `format`，返回日期的字符串表示形式或者说把日期按格式 `format` 转换为字符串，`format` 参数见 `fromString()`。

③、`int day() const;` 返回“日”

④、`int month() const;` 返回“月”。

⑤、`int year() const;` 返回“年”，若是负数，则表示公元前。

⑥、`int dayOfWeek() const;` 返回当日是星期几(1~7)。

⑦、`int dayOfYear() const;` 返回当日所在当年的第几天(1~365 或 366)

⑧、`int daysInMonth() const;` 返回当前日期当月总共有多少天(28~31)。

⑨、`int daysInYear() const;` 返回当前日期当年总共有多少天(365 或 366)。

⑩、`int daysTo(const QDate& d) const;`

返回当前日期与日期 `d` 相差的天数，负数表示 `d` 位于当前日期之前，

⑪、`void getDate(int *y, int *m, int *d) const;`

把当前日期的“年”，“月”，“日”提取出来，并分别存储在 `y`，`m`，`d` 之中。

⑫、`int weekNumber(int *yearNumber = Q_NULLPTR) const;`

返回当前日期所在当年的第几周(1~53)，并将年份存储在 `yearNumber` 中。返回值是依 ISO8601 的规定执行的，即周是从星期 1 开始的，一年的星期 4，始终是该年的第

1 周，因此 yearNumber 的值可能会与当前日期的年份不相同，如下所示：

- 若 1 月 1 日是星期五、星期六或星期天，则当年的 1 月 1 日并不是当年的第 1 周，而是上一年的最后一周，比如 2000 年 1 月 1 日，是星期 6，因此 2000 年 1 月 1 号是 1999 年的最后一周，同理 2000 年 1 月 2 号也是 1999 年的最后一周。
- 若 12 月 31 号是星期 1、2、3，则当年的 12 月 31 日是第二年的第一周，比如 2002 年 12 月 31 日是星期 2，因此 2002 年 12 月 31 日都是 2003 年的第一周。同理 200 年 12 月 30 日也是 2003 年的第一周。

7、对日期进行转换

- ①、`static QDate fromJulianDay(qint64 jd);` //静态，把儒略日(Julian Day);转换为 QDate 对象。
- ②、`qint64 toJulianDay() const;` //把日期转换为儒略日(Julian Day);

8、其余重载的操作符函数有：!=、<、<=、==、>、>=

示例 21: QDate 类的使用

```
#include<QtWidgets>
#include<QDebug>;
int main(int argc, char *argv[]) {
    QDate *pd1=new QDate(2003,2,15); //创建日期为 2003 年 2 月 15 日
    QDate pd2=QDate::currentDate(); //以当前系统时间创日期。
    QDate pd3=QDate::fromString("2000.1.2","yyyy.M.d"); //以指定的格式创建日期
    //pd4 创建的日期无效，因为日期与指定的格式不符
    QDate pd4=QDate::fromString("2000.1.2","yyyy.VVVM.d");
    //以格式 Qt::TextDate(这是默认值)创建日期，注意：星期和月分通常是缩写的英文。
    QDate pd5=QDate::fromString("Fri Jan 22 2004",Qt::TextDate);

    qDebug()<<pd3.toString(); //输出"周日 1 月 2 2000"
    qDebug()<<pd3.toString("yyyy-MM-dd Week:ddd"); //输出"2000-01-02 Wee:周日"
    qDebug()<<pd3.dayOfWeek(); //输出 7(即星期天)。
    qDebug()<<pd3.daysInMonth(); //输出 31(即 2000 年 1 月分共有 31 天)
    int i,j,k;
    pd3.getDate(&i,&j,&k); //将当前日期分别保存在 i, j, k 之中
    qDebug()<<"i="<<i<<"j="<<j<<"k="<<k; //输出 i=2000, j=1, k=2。
    qDebug()<<pd3.weekNumber(&i)<<i; /*输出 52(即 2000 年 1 月 2 日是 1999 年的第 52 周)，这里要注意，连续输出的 i 的值可能仍是之前的 2000(即未改变)，这与 C++的<<运算符的语法规则有关。*/

    qDebug()<<i; //输出 1999。
    QDate pd6 = pd3.addMonths(3); //将日期增加 3 个月，并返回。因此 pd6 为 2000-4-2
    qDebug()<<pd6.dayOfYear(); //输出 93(即 4 月 2 日是 2000 年的第 93 天)。
    qDebug()<<pd3.daysTo(pd6); //输出 91(即 pd6 与 pd3 相差 91 天)
    return 0;
}
```

三、QTime 类

1、QTime 类用于描述时间(即时分秒)

2、QTime 类描述的时间采用的是 24 小时制，因此没有 AM/PM(上午/下午)的概念。QTime

也不知道时区和夏令时(DST)，QDateTime 类对以上概念会进行描述。

3、时间的精确性取决于底层的操作系统，并不是所有的操作系统都能提供 1 毫秒的精度。

3、构造 QTime 类对象的函数

①、**QTime()**;

构造一个空 QTime 对象，即 QTime(0,0,0,0)(即 0 时 0 分 0 秒 0 毫秒)，此时 isNull 返回 true，isValid()返回 false。

②、**QTime(int h, int m, int s = 0, int ms = 0);**

构造一个 h 时 m 分 s 秒 ms 毫秒的 QTime 对象，其中 h 必须为 0~23，m 和 s 必须为 0~59，ms 必须为 0~999。

③、**static QTime currentTime();** //静态

使用系统时钟报道的当前时间创建一个 QTime 对象。这是常用方法。

④、**static QTime fromMSecsSinceStartOfDay(int msec);** //静态

使用相对于 00:00:00 的毫秒数 msec 来创建时间，1 小时=3600*1000=360 0000 毫秒。

比如创建一个时间 09:20:30:400 的时间，则

msec = 9*3600000+20*60*1000+30*1000+400=33630400，因此其创建方法为

QTime qt=QTime::fromMSecsSinceStartOfDay(33630400); //09:20:30:400;

⑤、**static QTime fromString(const QString &string,**

Qt::DateFormat format = Qt::TextDate); //静态

⑥、**static QTime fromString(const QString &string, const QString &format);** //静态

- 以上两函数表示以给定的格式 format 返回由字符串 string 表示的 QTime 对象，或者说把字符串以指定的格式转换为 QTime 对象。若参数 string 表示的时间不符合格式参数 format 指定的格式，则以上函数会返回一个无效的 QTime。
- 其中字符串 string 的格式由两种方式来表示，一种是由 Qt 内部使用枚举 Qt::DateFormat (见 QDate::fromString())已定义好的格式，一种是使用字符串的形式指定。
- 使用字符串指定时间格式
 - 其方法是在字符串中使用一些特殊的字符代表日期或时间，Qt 使用字符 d 表示“日”，“M”表示月，“y”表示年，其中一个字符表示一位数字，具体见下表

字符	说明
h	指定小时(0~23，AM/PM 模式为 1~12)，不能有前导 0，比如 01 是无效的。
hh	指定小时(00~23，AM/PM 模式为 01~12)，有前导 0，比如直接指定 1，是无效的。
m	指定“分”(0~59)，不能有前导 0。
mm	指定“分”(00~59)，必须有前导 0。
s	指定“秒”(0~59)，不能有前导 0。
ss	指定“秒”(00~59)，必须有前导 0。
z	秒的小数部分(0~999)
zzz	秒的小数部分(000~999)
AP	解释为 AM/PM 时间，AP 必须为"AM"或"PM"
ap	解释为 AM/PM 时间，AP 必须为"am"或"pm"

- 除以上表格外的其他字符和使用单引号括起来的字符都被视为文本，使用参数 `string` 表示时间时，视为文本的字符必须原样指定，否则指定的时间无效(其示例见 `QDate::fromString()`，原理类似)
- 不指定前导 0 的格式，使用的是贪心算法(其示例见 `QDate::fromString()`)
- 若格式中未指定的部分，都默认为 0。

4、修改 QTime 的函数

①、`QTime addMSecs(int ms) const;` //注意，中间有个 M

②、`QTime addSecs(int s) const;` //注意，中间少一个 M

以上函数表示向当前时间增加 `ms` 毫秒或 `s` 秒之后，并返回更改后的时间。若最后计算出来的时间经过 00:00:00:000，则会自动循环。示例：

```
QTime qt(22,22,22); //22:22:22
```

```
QTime qt1;
```

```
qt1=qt.addSecs(50); //qt1 = 22:23:12
```

```
qt1=qt.addSecs(3*60*60); //qt1 = 01:22:22
```

③、`bool setHMS(int h, int m, int s, int ms=0);`

设置时间的时、分、秒、毫秒。若时间有效，则返回 `true`，否则返回 `false`。

5、判断 QTime 状态的函数

①、`bool isNull() const;` 若时间为空，则返回 `true`，否则返回 `false`。空时间也是无效的。

②、`bool isValid() const;`

```
static bool isValid(int h, int m, int s, int ms=0); //静态
```

当时间(或指定的时间)有效，则返回 `true`，否则返回 `false`。比如

```
QDate::isValid(22, 2, 88); //false, 秒大于 60。
```

6、获取 QTime 信息的函数

注：下文的当前时间是指调用该函数的 `QTime` 对象所表示的时间

①、`QString toString(const QString &format) const;`

②、`QString toString(Qt::DateFormat format = Qt::TextDate) const;`

以指定的格式 `format`，返回 `QTime` 的字符串表示形式，或者说把此 `QTime` 以指定的格式 `format` 转换为字符串，并返回该字符串。若时间无效，则返回空字符串。其中 `format` 的指定比 `QTime::fromString` 多了下表几种类型

字符	说明
H	指定小时(0~23，即使模式为 AM/PM)，不能有前导 0。
HH	指定小时(00~23，即使模式为 AM/PM)，有前导 0。
AP 或 A	显示 AM/PM
ap 或 a	显示 AM/PM
t	时区(比如"中国标准时间")。

- ③、`int hour() const;` 返回时(0~23)，若时间无效则返回-1。
- ④、`int minute() const;` 返回分(0~59)，若时间无效则返回-1。
- ⑤、`int second() const;` 返回秒(0~59)，若时间无效则返回-1。
- ⑥、`int msec() const;` 返回毫秒(0~999)，若时间无效则返回-1。
- ⑦、`int msecsSinceStartOfDay() const;` 返回当前时间距离 00:00:00 有多少毫秒的数量。
- ⑧、`int msecsTo(const QTime &t) const;`
 返回当前时间与时间 t 相差的毫秒数，负数表示 t 位于当前时间之前，QTime 是在一天内测量的，而一天有 86400000 毫秒，因此该值的范围在-86400000~86400000，若时间无效，则返回 0。
- ⑨、`int secsTo(const QTime &t) const;`
 该函数同 msecsTo()，只不过是每秒为单位，该函数不会考虑任何的毫秒。若时间无效，则返回 0。

7、其他

- ①、`int restart();`
 把时间重置为当前时间(这里是指系统显示的当前的时间)，并返回自上次调用 start() 或 restart()以来经过的毫秒数。注意：在调用 start()或 restart()函数 24 小时后，计数器会被归零。若调用 start()或 restart()之后，修改了系统的时钟，则结果是未定义的。
- ②、`void start();` //把时间设置为系统显示的当前时间。
- ③、`int elapsed() const;` //同 restart()，但不会重置时间为当前时间。

8、其余重载的操作符函数有：!=、<、<=、==、>、>=

9、QTime 的函数的原理同 QDate 类是相类似的，因此示例就省略了。

四、QDateTime 类

- 1、QDateTime 类用于描述日期(年月日)和时间(时分秒)，该类是 QDate 和 QTime 类的组合。
- 2、QDateTime 类不考虑闰秒，QDateTime 类支持时区(比如夏令时，UTC 等)
- 3、QDateTime 使用一个 qint64 位的毫秒值来存储日期，其范围大约在+/- 2.92 亿年，比 QDate 存储的范围要小。
- 4、时区：根据地球经度的不同，把全球划分为 24 个时区。每个时区之间都会相差一定的小时数，比如中国北京时间使用的是 UTC+8 表示在 UTC 时间的基础上偏移了 8 个小时。由此可见，同一时间在不同的时区，显示的时间是不一样的，比如对于相同的 UTC 时间 12:00:00，在 UTC+8 的时区则为 20:00:00，而在 UTC+4 的时区则为 16:00:00。注意：目前的偏移量的范围为±14 小时。
- 5、在 Qt 中，时区可使用两种方式来指定，一种是使用 Qt 内置的使用 Qt::TimeSpec 枚举已设置好的时区，一种是使用 QTimeZone 类创建自定义的时区。下表为需要用到的枚举

Qt::TimeSpec 枚举(无标志)

作用：用于描述时间规范

成员	值	说明
----	---	----

Qt::LocalTime	0	取决于当前系统区域设置的时区
Qt::UTC	1	协调世界时(UTC)
Qt::OffsetFromUTC	2	与 UTC 有一个偏移量的差距(以秒为单位)
Qt::TimeZone	3	使用一组特定的夏令时规则命名的时区。

6、构造 QDateTime 类对象的函数

①、QDateTime():

QDateTime(const QDate& date);

QDateTime(const QDate& date, const QTime &time, Qt::TimeSpec spec = Qt::LocalTime);

QDateTime(const QDate& date, const QTime &time, Qt::TimeSpec spec, int offsetSeconds);

QDateTime(const QDate& date, const QTime &time, const QTimeZone &timeZone);

以上函数表示以指定的时区(默认为 Qt::LocalTime)构造 QDateTime 对象, 参数如下:

- **date**: 表示日期(年月日)
- **time**: 表示时间(时分秒), 若 date 有效, 而 time 无效, 则 time 被设置为 00:00:00.000
- **spec**: 指定时间规范(即时区), 若 spec 不是 Qt::OffsetFromUTC, 则参数 offsetSeconds 会被忽略。若 spec 是 Qt::OffsetFromUTC, 但 offsetSeconds 为 0, 则 spec 将是 Qt::UTC, 即 0 秒的偏移量。若 spec 为 Qt::TimeZone, 则 spec 针是 Qt::LocalTime(即当前的系统时区)。
- **offsetSeconds**: 表示与 UTC 时间偏移的秒数。
- **timeZone**: 表示时区。

②、static QDateTime **fromString**(const QString &string, Qt::DateFormat format = Qt::TextDate) //静态

③、static QDateTime **fromString**(const QString &string, const QString &format) //静态

以上两函数与 QDate::fromString()和 QTime::fromString 是类似的, 详见这两个函数

④、static QDateTime **fromMSecsSinceEpoch**(qint64 msecs) //静态, qt4.7

static QDateTime fromMSecsSinceEpoch(qint64 msecs, Qt::TimeSpec spec, int offsetSeconds = 0)) //静态, qt5.2

static QDateTime fromMSecsSinceEpoch(qint64 msecs, const QTimeZone& timeZone) //静态, qt5.2

static QDateTime fromSecsSinceEpoch(qint64 secs, Qt::TimeSpec spec = Qt::LocalTime, int offsetSeconds = 0)) //静态, qt5.8

static QDateTime fromSecsSinceEpoch(qint64 secs, const QTimeZone& timeZone) //静态, qt5.8

- 以上 5 个函数表示, 返回一个 QDateTime 对象, 该对象使用相对于 1970-01-01 T 00:00:00.000 (UTC) 以来所经过的秒(secs)或毫秒(msecs)数创建, 并把该对象转换为 Qt::LocalTime 或由 spec 指定的规范或转换为由 timeZone 指定的时区时间。
- 若系统不支持时区, 则使用当地时间的 Qt::UTC 规范。
- 若 spec 不是 Qt::OffsetFromUTC, 则参数 offsetSeconds 会被忽略。
- 若 spec 是 Qt::OffsetFromUTC, 但 offsetSeconds 为 0, 则 spec 将是 Qt::UTC, 即 0 秒的偏移量。
- 若 spec 为 Qt::TimeZone, 则 spec 针是 Qt::LocalTime(即当前的系统时区)。

⑤、static QDateTime **currentDateTime**(); //静态, 获取当前系统所设置的时区的当前日期和时间。

⑥、static QDateTime **currentDateTimeUtc**(); //静态, 获取由当前系统所设置的 UTC 时间。

- ⑦、`static qint64 currentMSecsSinceEpoch();` //静态
`static qint64 currentSecsSinceEpoch();` //静态, qt5.8

以上两函数表示, 返回自 1970-01-01T00:00:00(UTC)以来, 所经过的毫秒或秒数。

示例 22: 理解时区

//假设系统设置的本地时区(LocalTime)为 UTC+8(即中国标准时间)

`QDateTime dt=QDateTime::currentDateTime();` //获取当前时前(时区由系统设置)

`QDateTime dt1=QDateTime::currentDateTimeUtc();` //获取当前时间(时区为 UTC)

//输出: `QDateTime(2018-03-27 20:03:53.404 中国标准时间 Qt::TimeSpec(LocalTime))`

`qDebug()<<dt;`

//输出: `QDateTime(2018-03-27 12:03:53.408 UTC Qt::TimeSpec(UTC))`

`qDebug()<<dt1;`

//注意: 以上两个时间相差 8 小时

7、修改 QDateTime 的函数

- ①、`QDateTime addDays(qint64 ndays) const;`
`QDateTime addMonths(int nmonths) const;`
`QDateTime addYears(int years) const;`
`QDateTime addSecs(qint64 s) const;`
`QDateTime addMSecs(qint64 msec) const;`

- 以上函数表示向当前日期增加 ndays 天或 nmonths 月或 nyears 年或 s 秒或 msec 毫秒之后, 并返回更改后的日期(示例见 QDateTime 类中的相应函数)。
- 如果 timeSpec() 是 QT::LocalTime 或 QT::TimeZone, 则会检查日期和时间, 以确定它们是否落入了“标准时间到夏令时的过渡时间”中, 即, 若转换时间是凌晨 2 点, 并且时钟提前到凌晨 3 点, 则从 02:00:00 到 02:59:59.999 的时间被认为是无效的。

- ②、`void setDate(const QDateTime &date);` //把日期设置为 date

- ③、`void setTime(const QDateTime &time);` //把时间设置为 time

- ④、`void setTimeSpec(Qt::TimeSpec spec);` //设置时间规范

- ⑤、`void setTimeZone(const QDateTime &toZone);` //设置时区, Qt5.2

- ⑥、`void setOffsetFromUtc(int offsetSeconds);` //Qt5.2

设置与 UTC 的偏移秒数, 该设置会使时间规范设置为 Qt::OffsetFromUTC

- ⑦、`void setMSecsSinceEpoch(qint64 msec);`

- ⑧、`void setSecsSinceEpoch(qint64 sec);` //qt5.8

以上两函数表示, 使用相对于 1970-01-01 T 00:00:00.000 (UTC) 以来所经过的秒(secs)或毫秒(msec)数设置日期和时间

- ⑨、`void swap(QDateTime &other);` //交换两个 QDateTime 对象, qt5.0

8、判断 QDateTime 状态的函数

- ①、`bool isDaylightTime() const;` //判断是否是夏令时。

- ②、`bool isNull() const;` //判断是否为空。
- ③、`bool isValid() const;` //判断是否有效。

9、获取 QDateTime 信息的函数

- ①、`QDate date() const;` //获取 QDateTime 的日期部分
- ②、`QTime time() const;` //获取 QDateTime 的时间部分
- ③、`Qt::TimeSpec timeSpec() const;` //获取 QDateTime 的时间规范
- ④、`QTimeZone timeZone() const;` //获取 QDateTime 的时区, qt5.2
- ⑤、`QString timeZoneAbbreviation() const;` //qt5.2

获取 QDateTime 的时区缩写。比如 `Qt::OffsetFromUTC` 的格式为 `UTC ± 00:00`;注意:缩写并不保证是唯一的值,即不同时区可能会有相同的缩写。

- ⑥、`int offsetFromUtc() const;` //获取 UTC 的当前偏移量(以秒为单位)。qt5.2
- ⑦、`qint64 daysTo(const QDateTime &other) const;`

返回该 QDateTime 到 other 之间相差的天数,若 other 早于该 QDateTime 则为负值。天数是以从一天到达另一天的 00:00:00 的次数计算的,比如第一天的 23:00 到第二天的 01:00 之间的 2 小时相差为一天。

- ⑧、`qint64 secsTo(const QDateTime &other) const;`
`qint64 msecsTo(const QDateTime &other) const;`

以上两函数表示,返回该 QDateTime 到 other 之间相差的秒数或毫秒数,若 other 早于该 QDateTime 则为负值。在比较之前会把两个 QDateTime 转换为 `Qt::UTC`,以保证转换的正确性。

10、对 QDateTime 的转换

- ①、`QString toString(const QString &format) const;`
`QString toString(Qt::DateFormat format = Qt::TextDate) const;`

把此 QDateTime 使用格式 format 转换为字符串,并返回该字符串,该函数 format 的取值详见 `QDate::fromString()` 和 `QTime::fromString()`。

- ②、`qint64 toMsecsSinceEpoch() const;`
`qint64 toSecsSinceEpoch() const;` //qt5.8

把此 QDateTime 转换为相对于 1970-01-01 T 00:00:00.000 (UTC) 以来所经过的秒或毫秒数,并返回该数值。

- ③、`QDateTime toUtc() const;`
把此 QDateTime 转换为标准的 UTC 时间,并返回转换后的 QDateTime。

- ④、`QDateTime toTimeZone(const QTimeZone &timeZone) const;`
把此 QDateTime 以指定的时区 timeZone 进行转换,并返回转换后的 QDateTime。

- ⑤、`QDateTime toLocalTime() const;`
把此 QDateTime 以指定的时间规范 `Qt::LocalTime` 进行转换,并返回转换后的 QDateTime。

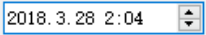
- ⑥、`QDateTime toTimeSpec(Qt::TimeSpec spec) const;`
把此 QDateTime 以指定的时间规范 spec 进行转换,并返回转换后的 QDateTime。

⑦、`QDateTime toOffsetFromUtc(int offsetSeconds) const;` //qt5.2

把此 QDateTime 以时间规范 OffsetFromUTC、偏移量 offsetSeconds 进行转换，并返回转换后的 QDateTime。

- 11、其余重载的操作符函数有：!=、<、<=、=、==、>、>=
- 12、QDateTime 类，主要是要弄清楚时区的概念，其他与 QDate 和 QTime 是类同的，示例可参见 QDate 类的示例。

四、QDateTimeEdit 类(见右图)



- 1、QDateTimeEdit 类是 QAbstractSpinBox 类的直接子类 and 具体实现，
- 2、QDateTimeEdit 类被设计用于显示和编辑日期、时间。
- 3、QDateTimeEdit 把日期和和时间分为几个部分，比如 2002/04/05T20:33:44.222 被分为 YearSection(年部分)、HourSection(小时部分)等。

4、QDateTimeEdit 类中的枚举

QDateTimeEdit::Section 枚举

标志：QDateTimeEdit::Sections

作用：用于描述日期时间的部分(以下各成员的意义比较明显，就不重述了)

成员	值	成员	值
QDateTimeEdit::NoSection	0x0000	QDateTimeEdit::HourSection	0x0010
QDateTimeEdit::AmPmSection	0x0001	QDateTimeEdit::DaySection	0x0100
QDateTimeEdit::MSecSection	0x0002	QDateTimeEdit::MonthSection	0x0200
QDateTimeEdit::SecondSection	0x0004	QDateTimeEdit::YearSection	0x0400
QDateTimeEdit::MinuteSection	0x0008		

4、QDateTimeEdit 类中的属性

QDateTimeEdit 类属性速查表

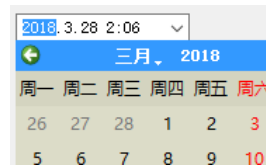
属性名	说明	属性名	说明
calendarPopup	是否弹出日历	time	旋转框的时间
currentSection	旋转框的当前部分	timeSpec	旋转框中时间的时间规范
currentSectionIndex	当前部分的索引	maximumDate	日期最大值
displayFormat	日期时间的格式	maximumDateTime	日期和时间的最大值
displayedSections	获取旋转框显示的部分	maximumTime	时间的最大值
sectionCount	旋转框显示的部分个数	minimumDate	日期的最小值
date	旋转框的日期	minimumDateTime	日期和时间的最小值
dateTime	旋转框的日期和时间	minimumTime	时间的最小值

- ①、`calendarPopup: bool` 访问函数: `bool calendarPopup() const; void setCalendarPopup(bool);`
此属性描述，是否弹出日历(下图是弹出日历的效果)。
- ②、`currentSection: Section`

访问函数: Section currentSection() const;

void setCurrentSection(Section);

获取和设置旋转框的当前部分, 注意, 把某部分设置为当前部分不会使该部分被选中, 要选中某一部分的内容请使用 setSelectedSection() 函数。



③、**currentSectionIndex:** int

访问函数: int currentSectionIndex() const; void setCurrentSectionIndex(int);

获取和设置旋转框的当前部分的索引(从 0 开始)。比如 2002/03/04, 若文字光标位于 4 之前, 则索引为 2(即第 2 部分)

④、**displayFormat:** QString

访问函数: QString displayFormat() const; void setDisplayFormat(const QString&);

获取和设置日期时间的格式, 详见 QDateTime::fromString 和 QTime::fromString, 注意: 若指定的年份只有两位数, 将被解释为 QDateTimeEdit 被初始化时的世纪, 默认为 21 世纪。

⑤、**displayedSections:** const Sections

访问函数: Sections displayedSections() const;

获取当前旋转框显示了时间的哪些部分, 比如 "12-31 "(假设显示的是 12 月 31 日), 则该属性的结果为 MonthSection | DaySection

⑥、**sectionCount:** const int

访问函数: int sectionCount() const;

获取旋转框显示了多少个部分。

⑦、**date:** QDate

访问函数: QDate date() const; void setDate(const QDate&);

信号: void dateChanged(const QDate& date);

获取和设置旋转框的日期, 默认为 2000/1/1

⑧、**dateTime:** QDateTime

访问函数: QDateTime dateTime() const; void setDateTime(const QDateTime&);

信号: void dateTimeChanged(const QDateTime &);

获取和设置旋转框的日期时间, 默认为 2000/1/1 T 00:00:00.000

⑨、**time:** QTime

访问函数: QTime time() const; void setTime(const QTime&);

信号: timeChanged(const QTime&);

获取和设置旋转框的时间, 默认为 00:00:00.000

⑩、**timeSpec:** Qt::TimeSpec

访问函数: Qt::TimeSpec timeSpec() const; void setTimeSpec(Qt::TimeSpec)

获取和设置旋转框中时间的的时间规范。Qt::TimeSpec 见 QDateTime 类

⑪、**maximumDate:** QDate

访问函数: QDate maximumDate() const; void setMaximumDate(const QDate&);

void clearMaximumDate(); //该函数用于恢复默认值

获取和设置日期的最大值, 默认为 7999 年 12 月 31 日。

⑫、**maximumDateTime:** QDateTime

访问函数: QDateTime maximumDateTime() const;

void setMaximumDateTime(const QDateTime&);

void clearMaximumDateTime(); //用于恢复默认值

获取和设置日期和时间的最大值，默认为 7999-12-31 23:59:59.999。

⑬、**maximumTime**: QTime

访问函数: QTime maximumTime() const; void setMaximumTime(const QTime&);
void clearMaximumTime();

获取和设置时间的最大值，默认为 23:59:59.999。

⑭、**minimumDate**: QDate

访问函数: QDate minimumDate() const; void setMinimumDate(const QDate&);
void clearMinimumDate(); //用于恢复默认值

获取和设置日期的最小值，默认为 1752-9-14

⑮、**minimumDateTime**: QDateTime

访问函数: QDateTime minimumDateTime() const;
void setMinimumDateTime(const QDateTime&);
void clearMinimumDateTime(); //用于恢复默认值
获取和设置日期和时间的最小值，默认为 1752-9-14 00:00:00.000

⑯、**minimumTime**: Time

访问函数: QTime minimumTime() const; void setMinimumTime(const QTime&);
void clearMinimumTime(); //用于恢复默认值
获取和设置时间的最小值，默认为 00:00:00.000

5、QDateTimeEdit 类的构造函数

- ①、**QDateTimeEdit**(QWidget *parent = Q_NULLPTR);
- ②、**QDateTimeEdit**(const QDateTime &datetime, QWidget *parent = Q_NULLPTR);
- ③、**QDateTimeEdit**(const QDate &date, QWidget *parent = Q_NULLPTR);
- ④、**QDateTimeEdit**(const QTime &time, QWidget *parent = Q_NULLPTR);

6、QDateTimeEdit 类中的函数

- ①、Section **sectionAt**(int index) const;

返回索引 i 表示的是日期时间的哪个部分，比如 2-13-2018(2018 年 2 月 13 日)，则 sectionAt(1)返回 DaySection

- ②、QString **sectionText**(Section section) const; 返回指定部分 section 处的文本。
void **setSelectedSection**(Section section);

选中指定部分 section 的文本，若 section 为 NoSection 则取消选中的所有文本。

- ③、void **setDateRange**(const QDate &min, const QDate &max);
void **setDateTimeRange**(const QDateTime &min, const QDateTime &max);
void **setTimeRange**(const QTime &min, const QTime &max);

以上 3 个函数是设置日期、时间和日期时间大小的便捷函数。

- ④、void **setCalendarWidget**(QCalendarWidget* calendarWidget);

把 calendarWidget 设置为该旋转框的日历。

- ⑤、QCalendarWidget* **calendarWidget**() const;

返回该旋转框的日历，若未设置日历，则创建并返回该日历。

- ⑥、`void dateChanged(const QDateTime &date);` //信号，日期改变时发送
- ⑦、`void timeChanged(const QTime &time);` //信号，时间改变时发送
- ⑧、`void dateTimeChanged(const QDateTime &datetime);` //信号，日期或时间改变时发送。

7、以下虚函数在子类化时可能需要被重新实现，以定义自己的功能。具体实现原理和方法可参阅 QSpinBox 类的讲解。

- ①、`QString textFromDateTime(const QDateTime &dateTime) const;` //虚拟的，受保护的。
- ②、`virtual QDateTime dateFromText(const QString &text) const;` //虚拟的，受保护的
- ③、`QValidator::State validate(QString &text, int &pos) const;` //虚拟的，受保护的。
- ④、`void fixup(QString &input) const;` //虚拟的，受保护的。
- ⑤、`virtual void stepBy(int steps);` //虚拟的，表示步长。

五、QDateEdit 类和 QTimeEdit 类

- 1、QDateEdit 和 QTimeEdit 类是 QDateTimeEdit 类的直接子类，
- 2、QDateEdit 类被设计用于显示和编辑日期，QTimeEdit 类用于显示和编辑时间。这两个类的功能都位于其父类 QDateTimeEdit 之中，因此没有其自身的成员函数，下面为这两个类的构造函数。

```
QTimeEdit(QWidget* parent = Q_NULLPTR);
QTimeEdit(const QTime& time, QWidget* parent = Q_NULLPTR);
QDateEdit(QWidget* parent = Q_NULLPTR);
QDateEdit(const QDateTime& date, QWidget* parent = Q_NULLPTR);
```

六、QTimer 类(计时器)

- 1、QTimer 类是 QObject 的直接子类，该类用于实现计时器，QTimer 类未继承自 QWidget 因此该类的对象没有像 QWidget 一样，有一个可见的窗口。
- 2、实现计时器的方法：

- ①、方法 1：使用 start() 函数。调用 start() 函数之后，计时器会以固定的时间间隔发送 timeout() 信号(也通常说成，计时器超时(即到达设置的时间间隔)时会发送 timeout() 信号)，因此只需把槽联接该信号即可。示例：

```
QTimer* t=new QTimer(this); //创建计时器。
connect(t, SIGNAL(timeout()), this, SLOT(f)); //把 timeout 信号关联到槽 f
t->start(1000); //每隔 1000 毫秒(1 秒)发送一次 timeout() 信号
```

- ②、方法 2：使用 QTimer::singleShot() 静态函数在指定的时间间隔后调用一次槽。代码为 `QTimer::singleShot(1000, this, SLOT(f));` //1 秒之后调用槽 f(注意：只会调用一次)。
- ③、使用 QObject::startTimer() 函数，并重新实现 QObject::timerEvent() 事件处理函数。调用 QObject::startTimer() 函数后，每隔一固定时间间隔就会产生 TimerEvent 事件，然后使用 QObject::timerEvent() 事件处理函数处理该事件

3、QTimer 类的属性

- ①、**active**: const bool 访问函数: bool isActive() const;
若计时器正在运行, 则返回 true, 否则返回 false
- ②、**interval**: int
访问函数: int interval () const; void setInterval(int msec);
void setInterval(std::chrono::milliseconds value); //chrono 是 C++11 的标准库
获取和设置计时器的时间间隔(或超时时间), 默认为 0。当事件队列中的所有事件都被处理后, 时间间隔为 0 的计时器将超时。设置处于活动状态的计时器的时间间隔, 会更改其 id, 即 timeId()返回的值会改变
- ③、**remainingTime**: const int 访问函数: int remainingTime() const; //qt5.0
获取以毫秒为单位的计时器的剩余时间。若计时器过期, 则返回 0, 若计时器处于非活动状态, 则返回-1。
- ④、**singleShot**: bool 访问函数: bool isSingleShot() const; void setSingleShot(bool);
此属性描述计时器是否为单次计时器, 单次计时器只会触发一次, 非单次计时器每隔一定时间触发一次。默认为 false(非单次计时器)
- ⑤、**timerType**: Qt::TimerType
访问函数: Qt::TimerType timerType() const; void setTimerType(Qt::TimerType);
获取和设置计时器的精度。默认为 Qt::CoarseTimer。Qt::TimerType 枚举见下表

Qt::TimerType 枚举(无标志)

描述计时器的精度

成员	值	说明
Qt::PreciseTimer	0	精确的计时器, 试图保持毫秒的精度。
Qt::CoarseTimer	1	粗略的计时器, 试图保持精度在期望间隔的 5% 以内。
Qt::VeryCoarseTimer	2	非常粗略的计时器, 只保持完整的秒的精度

4、QTimer 类中的函数

- ①、**QTimer**(QWidget* parent = Q_NULLPTR); //构造函数
 - ②、int **timeId**() const; //若计时器正在运行, 则返回其 ID, 否则返回-1。
 - ③、static void **singleShot**(int msec, const QObject* receiver, const char* member); //静态
static void **singleShot**(int msec, Qt::TimerType timerType, const QObject* receiver, const char* member); //静态
static void **singleShot**(int msec, const QObject* receiver, PointerToMemberFunction method); //静态
static void **singleShot**(int msec, Qt::TimerType timerType, const QObject* receiver, PointerToMemberFunction method); //静态
- 以上函数表示, 以时间间隔 msec(毫秒)调用一次槽函数 method 或 member(注意: 只会调用一次), 其中 timerType 参数表示计时器的精度, 见 timerType 属性。以上函数的使用方法与 QObject::connect()函数类似, 其中 method 表示指向成员函数的指针。
- ④、static void **singleShot**(int msec, Functor functor); //静态

```
static void singleShot(int msec, Qt::TimerType timerType, Functor functor); //静态
static void singleShot(int msec, const QObject* context, Functor functor); //静态
static void singleShot(int msec, Qt::TimerType timerType,
                        const QObject* context, Functor functor); //静态
```

以上函数中的参数 `functor` 表示仿函数。

5、以下函数需要 C++11 的支持

- ①、void **start**(std::chrono::milliseconds msec);
- ②、std::chrono::milliseconds **remainingTimeAsDuration**() const;
- ③、std::chrono::milliseconds **intervalAsDuration**() const; //需要 C++11 的支持。qt5.8
- ④、static void **singleShot**(std::chrono::milliseconds msec, Qt::TimerType timerType,
 const QObject* receiver, const char* member); //静态
- ⑤、static void **singleShot**(std::chrono::milliseconds msec,
 const QObject* receiver, const char* member); //静态

6、QTimer 类中的信号和槽

- ①、void **start**(int msec); //槽
以 msec(毫秒)的超时时间间隔启动或重新启动计时器。若计时器正在运行，则它将被停止并重新启动。若 `singleShot` 属性为 `true`，则计时器只会激活一次。
- ②、void **start**() //槽，start(int)的重载版本，时间间隔使用 `interval` 属性的设置。
- ③、void **stop**(); //槽，停止计时器。
- ④、void **timeout**(); //信号。计时器超时(即到达指定的时间间隔)时发出此信号

7、QObject 类中与计时器有关的函数

- ①、int **startTimer**(int interval, Qt::TimerType timetype = Qt::CoarseTimer);
启动计时器并返回其 id，并以 interval 毫秒的时间间隔产生一次 `QTimeEvent` 事件，直到调用 `killTimer`()停止该计时器，若无法启动计时器则返回 0。
- ②、void **timerEvent**(QTimeEvent* event); //虚拟的，受保护的。
该函数通常需要重新事件，以处理 `QTimeEvent` 事件。
- ③、void **killTimer**(int id); //停止 ID 为 id 的计时器，ID 由 `startTimer`()函数返回。

示例 23：时间的显示和修改(QTime、QTimer、QTimeEdit 类的使用)

//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>
#include <iostream>
using namespace std;
class B:public QTimeEdit{    Q_OBJECT
public:
    QPushButton *pb;    QPushButton *pbl;    QTimer t;
    B(QWidget *p=0):QTimeEdit(p){ } //默认构造函数
```



```

B(QTime d, QWidget *p=0):QTimeEdit(d, p) {
    pb=new QPushButton("edit", p);pb->move(22, 66);
    pbl=new QPushButton("ok", p); pbl->move(99, 66);
    setReadOnly(true);    //将旋转框设置为只读模式，这样用户就不能修改时间了
    setWrapping(true);    //开启微调按钮的循环调节模式
    connect(pb, &QPushButton::clicked, this, &B::g);
    connect(this, &QTimeEdit::editingFinished, this, &B::s);
    connect(pbl, &QPushButton::clicked, this, &B::s);
    connect(&t, &QTimer::timeout, this, &B::f);
    t.start(1000);        }//启动计时器，构造函数结束

public slots:
    void f() {    setTime(time().addSecs(1));    } /*使时间在当前时间的基础上增加 1 秒。该函数由计时器超时时调用，这样就可使 QTimeEdit 内显示的时间每隔一定时间自动增加 1 秒。*/

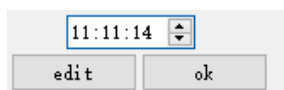
    void g() {    //该函数主要用于使 QTimeEdit 可编辑，这样用户才能编辑时间。
        t.stop();    //在编辑时间前，停止计时器
        setReadOnly(false);    }    //使 QTimeEdit 可编辑。

    void s() {    //此函数表示，当用户编辑完成后，使 QTimeEdit 中的时间自动运行起来。
        setReadOnly(true);    //编辑完成后，关闭对 QTimeEdit 的编辑
        t.start(1000);    }    //重新启动计时器，以使时间能走起来
};
#endif // M_H

//m.cpp 文件的内容
#include "m.h"
int main(int argc, char *argv[]) {    QApplication a(argc, argv);
    QWidget w;
    QTime t(11, 11, 11);    //也可使用 QTime::currentTime()来获取当前系统的时间。
    B *pd=new B(t, &w);    pd->move(55, 44);
    pd->setDisplayFormat("hh:mm:ss");    //设置时间的显示格式
    w.resize(300, 200);    w.show();    return a.exec();}

```

运行结果及说明



时间每隔 1 秒会自动增长，初始运行时，时间是不可修改的。当点击 edit 按钮时，时间会停止运行，此时可修改时间，当修改完成后按下 enter 键或按下 ok 按钮，时间会在新修改后的时间处开始自动增长。

示例 24: QObject::startTimer() 函数的使用

```

//m.h 文件的内容
#ifndef M_H
#define M_H
#include<QtWidgets>
#include <iostream>
using namespace std;

class B:public QPushButton{    Q_OBJECT
public:

```



```

int x;    //用于保存计时器的 ID
B(QString s="", QWidget *p=0):QPushButton(s,p) {}
void timerEvent(QTimerEvent* e) { cout<<"F"<<endl; } //重新实现该函数
public slots:
void f() {
    x=startTime(1000); } //启动计时器，启动后，每隔 1 秒会产生一次 QTimerEvent 事件
void g() { killTimer(x); } //停止计时器。
void gl() { cout<<"X"<<endl; } };
#endif // M_H

//m.cpp 文件的内容
#include "m.h"
int main(int argc, char *argv[]) { QApplication a(argc, argv);
    QWidget w;
    B *pb=new B("start",&w);    pb->move(22,22);
    B *pbl=new B("end",&w);    pbl->move(99,22);
    QTimer::singleShot(1000, pb,&B::gl); //1 秒之后调用一次函数 gl(注意，只会调用一次)
    QObject::connect(pb, &QPushButton::clicked, pb, &B::f);
    //注意：一定要关联 pb 的槽函数 g，否则 g 不能停止由 pb 启动的计时器
    QObject::connect(pbl, &QPushButton::clicked, pb, &B::g);
    w.resize(300,200);    w.show();    return a.exec(); }

```

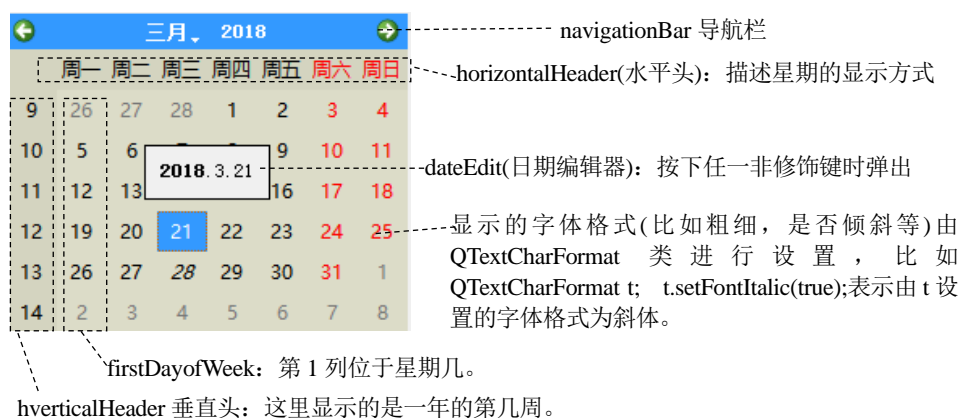
运行结果及说明



按下 start 按钮，会调用 B::f()函数，从而启动计时器，然后每隔 1 秒会产生一次 QTimerEvent 事件，该事件的事件处理函数，会输出字符 F(也就是每隔 1 秒会输出字符 F)，点击 end 按钮，可停止输出

七、QCalendarWidget 类(日历)

1、QCalendarWidget 类是 QWidget 的直接子类，该类用于日历，见下图



3、QCalendarWidget 类的属性

QCalendarWidget 类(日历)属性速查表

属性名	说明	属性名	说明
gridVisible	是否开启网格	horizontalHeaderFormat	设置水平头的格式
maximumDate	日期的最大值	verticalHeaderFormat	设置垂直头的格式
minimumDate	日期的最小值	dateEditAcceptDelay	日期编辑器的延迟时间
selectedDate	设置选择的日期	dateEditEnabled	是否启用日期编辑器
navigationBarVisible	是否显示导航栏	selectionMode	是否可在日历中选择日期
firstDayOfWeek	第一列显示的日期是星期几		

①、**dateEditAcceptDelay**: int

访问函数: int dateEditAcceptDelay() const; void setDateEditAcceptDelay(int)

获取和设置日期编辑器的延迟时间(以毫秒为单位), 即当用户在弹出的日期编辑器中输入内容之后, 该编辑器保持打开的时间, 过了这段时间后, 日期编辑器中被修改的内容将被日历部件接受, 弹出的日期编辑器被关闭, 默认为 1500 毫秒。

②、**dateEditEnabled**: bool

访问函数: bool isDateEditEnabled() const; void setDateEditEnabled(bool);

描述是否启用日期编辑器弹出窗口, 若启用该属性, 则按下任一非修饰键就会弹出日期编辑器, 在该编辑器中可直接输入数字, 也可使用左右箭头进行移动, 使用上下箭进行递增和递减。默认为 true(开启)

③、**firstDayOfWeek**: Qt::DayOfWeek

访问函数: Qt::DayOfWeek firstDayOfWeek() const; void setFirstDayOfWeek(Qt::DayOfWeek);

获取和设置第一列显示的日期是星期几。Qt::DayOfWeek 枚举如下表

Qt::DayOfWeek 枚举

成员	值	成员	值	成员	值
Qt::Monday	1(默认值)	Qt::Thursday	4	Qt::Saturday	6
Qt::Tuesday	2	Qt::Friday	5	Qt::Sunday	7
Qt::Wednesday	3				

④、**gridVisible**: bool

访问函数: bool isGridVisible() const; void setGridVisible(bool);

描述是否开启网格, 默认为 false

⑤、**horizontalHeaderFormat**: HorizontalHeaderFormat

访问函数: HorizontalHeaderFormat horizontalHeaderFormat() const;

void setHorizontalHeaderFormat(HorizontalHeaderFormat);

获取和设置水平头的格式, HorizontalHeaderFormat 枚举如下表

QCalendarWidget::HorizontalHeaderFormat 枚举

成员	值	说明
QCalendarWidget::SingleLetterDayNames	1	使用单字符缩写(比如使用 M 表示周一)
QCalendarWidget::ShortDayNames	2	使用简短的缩写(默认值), 比如"周一", "Mon"
QCalendarWidget::LongDayNames	3	使用完整的名称, 比如"星期一", "Monday"

QCalendarWidget::NoHorizontalHeader	0	无水平头。
-------------------------------------	---	-------

⑥、**maximumDate**: QDate

访问函数: QDate maximumDate()const; void setMaximumDate(const QDate&);

minimumDate: QDate

访问函数: QDate minimumDate()const; void setMinimumDate(const QDate&);

获取和设置日期的最大/最小值, 用户将无法选择在设置的最大/最小值范围之外的日期。默认的最大/最小值为 QDate 的最晚和最早日期。

⑦、**navigationBarVisible**: bool

访问函数: bool isNavigationBarVisible() const; void setNavigationBarVisible(bool);

描述是否显示导航栏, 默认为 true(显示)

⑧、**selectedDate**: QDate

访问函数: QDate selectedDate() const; void setSelectedDate(const QDate&);

获取和设置选择的日期, 默认为当前日期。

⑨、**selectionMode**: SelectionMode

访问函数: SelectionMode selectionMode() const; void setSelectionMode(SelectionMode);

描述用户是否可在日历中选择日期, SelectionMode 枚举见下表

QCalendarWidget::SelectionMode 枚举

成员	值	说明
QCalendarWidget::NoSelection	0	无法选择日期
QCalendarWidget::SingleSelection	1	可选择单个日期(默认值)

⑩、**verticalHeaderFormat**: VerticalHeaderFormat

访问函数: VerticalHeaderFormat verticalHeaderFormat() const;

void setVerticalHeaderFormat(VerticalHeaderFormat);

获取和设置垂直头的格式, VerticalHeaderFormat 枚举如下表

QCalendarWidget::VerticalHeaderFormat 枚举

成员	值	说明
QCalendarWidget::ISOWeekNumbers	0	显示 ISO 的周号(即一年的第几周)
QCalendarWidget::NoVerticalHeader	1	隐藏垂直头。

4、QCalendarWidget 类的函数

①、**QCalendarWidget** (QWidget* parent = Q_NULLPTR); //使用当前日期创建一个日历

②、int **monthShown**() const; //返回当前显示的月份(1~12)

③、int **yearShown**() const; //返回当前显示的年份

④、void **setDateRange**(const QDate& min, const QDate& max); //槽,
设置日期的范围, 用户只能在指定的范围内选择日期。

⑤、void **setCurrentPage**(int year, int month); //槽

显示指定年 `year` 和月 `month`，当前选择的日期不变。比如，当前选择的是 2002 年 2 月 13 日，则 `setCurrentPage(2018, 9);` 将显示在 2018 年 9 月，其中被选中的日期仍是 2002 年 2 月 13 日，未改变。

- ⑥、`void showSelectedDate();` //槽，显示当前选择日期的月分

`void showToday();` //槽，显示今天的月分

以上两函数可以编程的方式控制日历在当前选择日期的月分和当前日期月分之间转换。比如系统当前日期为 2018 年 3 月 3 日，当前选择的日期为 2013 年 4 月 8 日，则调用 `showSelectedDate()` 将使日历显示在 2013 年 4 月，调用 `showToday()` 将使日历显示在 2018 年 3 月，反复调用以上函数，日历会在以上两日期间反复显示。

- ⑦、`void showNextMonth();` //槽。

`void showNextYear();` //槽。

`void showPreviousYear();` //槽。

`void showPreviousMonth();` //槽。

以上函数表示显示当前日期的前或后一个月或一年，当前选择的日期不变。比如当前日期为 2019 年 2 月 13 日，则 `showNextMonth()` 将显示在 2019 年 3 月，延后一月，其中被选中的日期仍是 2019 年 2 月 13 日，未改变。而 `showPreviousYear()` 将显示在 2020 年 2 月(提前一年)，被选中的日期仍未变。

- ⑧、`void setDateTextFormat(const QDate& date, const QTextCharFormat& format);`

把指定的日期设置为指定的文本格式，比如 `date` 为 2019 年 4 月 3 日，假设 `format` 的格式为斜体，则 2019 年 4 月 3 日的数字 3 会以斜体显示。若 `date` 为空，则清除所有日期的格式。

- ⑨、`void setHeaderTextFormat(const QTextCharFormat& format);`

设置水平头和垂直头的文本格式，

- ⑩、`void setWeekdayTextFormat(Qt::DayOfWeek dayOfWeek, const QTextCharFormat& format);`

设置位于 `dayOfWeek` 那一列的文本的格式。

若 `setHeaderTextFormat` 和 `setWeekdayTextFormat` 同时设置了前景色或背景色，则 `setWeekdayTextFormat` 具有优先权。

- ⑪、`QMap<QDate, QTextCharFormat> dateTextFormat() const;`

`QTextCharFormat dateTextFormat(const QDate& date) const;`

`QTextCharFormat headerTextFormat() const;`

`QTextCharFormat weekdayTextFormat(Qt::DayOfWeek dayOfWeek) const;`

以上函数表示获取相应部分的 `QTextCharFormat` 对象。

- ⑫、`void updateCell(const QDate& date);` //受保护的，更新日期 `date` 的单元格。

- ⑬、`void updateCells();` //受保护的，更新所有可见的单元格。

- ⑭、`void paintCell(QPainter* painter, const QRect& rect, const QDate& date) const;` //虚拟的，受保护的
使用 `painter` 和 `rect` 绘制 `date` 的单元格。

5、QCalendarWidget 类中的信号

- ①、`void activated(const QDate& date);` //信号

当按下 `return`、`enter` 或双击日历中的日期时，发送此信号，注意单击不会发送。

- ②、`void clicked(const QDate& date);` //信号
单击鼠标时发出此信号，注意，按下 `enter` 不会发送。
- ③、`void currentPageChanged(int year, int month);` //信号
当前显示的月分发生变化时，发送此信号。`year` 和 `month` 为新的年份和月分。
- ④、`void selectionChanged();` //信号
当选择的日期发生改变时，发送此信号。注意：调用的 `showXXX` 函数不会改变被选中的日期，因此不会发送该信号，同理，在日历的导航栏上改变显示的年份或月分也不会改变被选中的日期，因此也不会发送该信号。

示例 25: QCalendarWidget(日历)的显示

//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>
#include <iostream>
using namespace std;
class B:public QCalendarWidget{    Q_OBJECT
public:
    B(QWidget *p=0):QCalendarWidget(p) {pe=new QLineEdit;} //在构造函数中初始化 pe
    QLineEdit    *pe; /*pe 创建于此主要是为了使 pe 关联到槽 B::f8(), 这样在 f8 之中才能得到用户
                        在 pe 中输入的文本内容，创建于此也可以看到信号和槽关联方式的灵活性*/

public slots:
    void f1() {showPreviousYear();    }    //显示到上一年
    void f2() {showNextYear();    }    //显示到下一年
    void f3() {showToday();    }    //显示到系统上的当前日期
    void f4() {showSelectedDate();    }    //显示到用户选中的日期的那个月
    void f5() {QTextCharFormat qt1;
                qt1.setBackground(QBrush(QColor(111,0,0)));    //设置背景色为红色
                setHeaderTextFormat(qt1);    } //把垂直头和水平头的背景色设置为红色
    void f6() {QTextCharFormat qt2;
                qt2.setBackground(QBrush(QColor(0,111,0)));
                setWeekdayTextFormat(QTextCharFormat::Friday,qt2);    } //把周五的那一列的背景色设置为绿色。
    void f7() {QTextCharFormat qt;    //qt 未设置任何格式
                setWeekdayTextFormat(QTextCharFormat::Friday,qt);    //清除垂直头和水平头的格式设置
                setHeaderTextFormat(qt);    } //清除格式设置
    void f8() {    setCurrentPage(pe->text().toInt(), 1);    } //显示到用户输入年份的第 1 个月
};
#endif // M_H
```

//m.cpp 文件的内容

```
#include "m.h"
int main(int argc, char *argv[]) {    QApplication a(argc, argv);
    QWidget w;
    QPushButton *pb1=new QPushButton("pre year",&w);    pb1->move(22,66);
    QPushButton *pb2=new QPushButton("next year",&w);    pb2->move(22,88);
    QPushButton *pb3=new QPushButton("show today",&w);    pb3->move(22,111);
    QPushButton *pb4=new QPushButton("show select",&w);    pb4->move(22,133);
    QPushButton *pb5=new QPushButton("header",&w);    pb5->move(22,155);
    QPushButton *pb6=new QPushButton("weekday",&w);    pb6->move(22,177);
```

```

QPushButton *pb7=new QPushButton("clear",&w);    pb7->move(22,199);
QLabel *pb8=new QLabel("enter year:",&w);    pb8->move(52,22);

B *pc=new B(&w);
pc->setHorizontalHeaderFormat(QCalendarWidget::ShortDayNames); //水平头使用简短的名称
pc->setFirstDayOfWeek(Qt::Sunday); //第一列日期为周日
pc->pe->setParent(&w);    pc->pe->move(122,22);    pc->move(122,44);

QObject::connect(pb1, &QPushButton::clicked, pc, &B::f1);
QObject::connect(pb2, &QPushButton::clicked, pc, &B::f2);
QObject::connect(pb3, &QPushButton::clicked, pc, &B::f3);
QObject::connect(pb4, &QPushButton::clicked, pc, &B::f4);
QObject::connect(pb5, &QPushButton::clicked, pc, &B::f5);
QObject::connect(pb6, &QPushButton::clicked, pc, &B::f6);
QObject::connect(pb7, &QPushButton::clicked, pc, &B::f7);
QObject::connect(pc->pe, &QLineEdit::editingFinished, pc, &B::f8); //注意 pe 的用法
w.resize(400,300);    w.show();    return a.exec();    }

```

运行结果及说明

在此输入框中可输入想要查看的年份

点击该按钮会使日期显示在 2011 年 1 月

点击该按钮会使日期显示在 2012 年 1 月

无论何时点击此按钮，日期都会显示到系统设置的当前日期那一月

无论怎样点击以上的按钮使日期显示到其他年份，被选中的日期都是 2011 年 1 月 12 日(除非选择了其他日期)，因此点击该按钮，都会使日历显示到这一页。

点击这两个按钮后的效果见右图，可见设置的周五这一列的背景优先于设置垂直头和水平头的背景



作者：黄邦勇帅(原名：黄勇)

2018-3-29

部件公用枚举

Qt::AlignmentFlag 枚举

标志: Qt::Alignment

此枚举用于描述对齐方式,一次只能指定一个水平标志和一个垂直标志,冲突的标志组合其行为是未定义的。

成员	值	说明
水平标志		
Qt::AlignLeft	0x0001	左对齐
Qt::AlignRight	0x0002	右对齐
Qt::AlignHCenter	0x0004	水平居中
Qt::AlignJustify	0x0008	在可用空间中对文本进行调整
垂直标志		
Qt::AlignTop	0x0020	顶部对齐
Qt::AlignBottom	0x0040	底部对齐
Qt::AlignVCenter	0x0080	垂直居中
Qt::AlignBaseline	0x0100	与基线一致
其他		
Qt::AlignAbsolute	0x0010	控件通常是从左到右布局的(默认值),但也能从右到左布局,此时 Qt::AlignLeft 将是向右侧对齐,而 Qt::AlignRight 将是向左侧对齐,将 Qt::AlignAbsolute 与上述对齐方式一起使用,可保证 Qt::AlignLeft 总是表示向左侧对齐,Qt::AlignLeft 总是表示向右侧对齐
Qt::AlignCenter		居中对齐 AlignVCenter AlignHCenter
Qt::AlignLeading		AlignLeft 的同义词
Qt::AlignTrailing		AlignRight 的同义词
以下值用于提取结果值中的水平和垂直标志		
Qt::AlignHorizontal_Mask		AlignLeft AlignRight AlignHCenter AlignJustify AlignAbsolute
Qt::AlignVertical_Mask		AlignTop AlignBottom AlignVCenter AlignBaseline

Qt::LayoutDirection 枚举(无标志)

用于描述布局和文本处理的方向(即从左到右布局还是从右到左布局),

成员	值	说明
Qt::LeftToRight	0	从左到右布局
Qt::RightToLeft	1	从右到左布局
Qt::LayoutDirectionAuto	2	自动布局

Qt::TextElideMode 枚举(无标志)

用于描述文本过长时,省略号出现的位置

成员	值	说明
Qt::ElideLeft	0	省略号出现在文本开头
Qt::ElideRight	1	省略号出现在文本末尾

Qt::ElideMiddle	2	省略号出现在文本中间
Qt::ElideNone	3	文本中不出现省略号

Qt::MatchFlag 枚举 标志: Qt::MatchFlags 用于描述搜索项目时使用的匹配方式		
成员	值	说明
Qt::MatchExactly	0	执行基于 QVariant 的匹配
Qt::MatchFixedString	8	执行基于字符串的匹配，此方式不区分大小写，除非指定了 MatchCaseSensitive 标志
Qt::MatchContains	1	搜索词与项目的内容匹配(类似于模糊搜索)，比如在“absde”中搜索“bs”表示是匹配的。
Qt::MatchStartsWith	2	搜索词需要与项目的开头匹配。比如在“abcdefg”中搜索“ab”表示是匹配的，但搜索“bc”则不是匹配的。
Qt::MatchEndsWith	3	搜索词需要与项目的末尾匹配。比如在“abcdefg”中搜索“fg”表示是匹配的，但搜索“bc”则不是匹配的。
Qt::MatchCaseSensitive	16	搜索区分大小写。
Qt::MatchRegExp	4	使用正则表达式作为搜索词执行基于字符串的匹配。
Qt::MatchWildcard	5	使用通配符(即符号“*”)，作为搜索词的字符串，执行基于字符串的匹配，比如在“abcdefg”中搜索“*b*”是匹配的，搜索“abc*”也是匹配的
Qt::MatchWrap	32	执行环绕搜索，即当搜索到达最后一个项目时，会再次从第一项开始搜索，直到搜索完所有项目
Qt::MatchRecursive	64	搜索整个层次结构。

Qt::TextFormat 枚举(无标志) 用于描述文本是否使用富文本。		
成员	值	说明
Qt::PlainText	0	纯文本
Qt::RichText	1	富文本
Qt::AutoText	2	自动检测(默认值)

作者：黄邦勇帅(原名：黄勇)

2018-3-29

本文作者：黄邦勇帅(原名：黄勇)

本文要求读者已经非常熟悉 C++ 语法。

本文是 QT 的基础，主要讲解了 Qt 的布局管理系统和焦点系统，本文对 Qt 的布局系统和焦点系统作了比较全面详细的深入讲解，本文从布局的原理出发，详细讲解了布局和焦点的每一个性质，并列举了详细的示例进行说明，学完本文就能完全弄明白 Qt 的布局原理及其熟练使用。本文内容由浅入深，易学易懂。

本文使用的是 windows 10 操作系统，Qt 版本为 Qt5.10.1，Qt Creator 的版本为 Qt Creator 4.5.1 本文内容完全属于个人见解与参考文献的作者无关，限于水平有限，其中难免有误解之处，望指出更正。

声明：禁止抄袭、复印、转载本文，本文作者拥有完全版权。

主要参考文献：

- 1、C++语法详解 黄勇 编著 电子工业出版社 2017 年 7 月
- 2、Qt Creator 快速入门(第 3 版) 霍亚飞 编著 北京航空航天大学出版社 2017 年 1 月
- 3、C++ GUI Qt4 编程(第 2 版) [加拿大]Jasmin Blanchette [英] Mark Summerfield 著 闫锋欣 曾泉人 张志强 译 电子工业出版社 2008 年 8 月
- 4、Qt 5 开发及实例 陆文周 主编 电子工业出版社 2014 年 1 月

第5章 Qt 布局管理及焦点系统目录

[5.1 布局原理](#)

[5.1.1 布局基础](#)

[5.1.2 部件拉伸 \(Stretch\) 原理及大小策略](#)

[5.1.3 大小约束 \(主窗口最大最小大小的设置\)](#)

[5.1.4 内容边距 \(ContentsMargins\)、间距 \(spacing\)、QSpace](#)

[5.1.5 嵌套布局及布局位于容器中](#)

[5.2 各布局管理器类](#)

[5.2.1 QBoxLayout 及其子类 \(盒式布局\)](#)

[5.2.2 QGridLayout 类 \(网格布局\)](#)

[5.2.3 QFormLayout 类 \(表单布局\)](#)

[5.3 实现多页面切换](#)

[5.3.1 QStackedLayout 类 \(分组布局或栈布局\)](#)

[5.3.2 QStackedWidget 类](#)

[5.3.3 QTabBar 类 \(选项卡栏\)](#)

[5.3.4 QTabWidget 类 \(选项卡部件\)](#)

[5.4 QSplitter 分离器\(或分隔符\)](#)

[5.4.1 QSplitter 类 \(分离器\)](#)

[5.4.2 QSplitterHandle 类 \(分界线\)](#)

[5.5 自定义布局管理器](#)

[5.5.1 QLayout 抽象类中的公有成员函数](#)

[5.5.2 QLayoutItem、QSpacerItem、QWidgetItem 类](#)

[5.5.3 自定义布局的实现](#)

[5.6 Qt 焦点系统](#)

[5.6.1 焦点链 \(焦点循环\)](#)

[5.6.2 获取焦点信息](#)

[5.6.3 焦点代理 \(FocusProxy\)](#)

[5.6.4 设置焦点及焦点策略](#)

[5.6.5 焦点事件](#)

[5.6.6 自定义焦点循环](#)

[5.6.7 QFocusFrame 类 \(焦点框, 自定义焦点框的外形\)](#)

[本章公用枚举](#)

第 5 部分 Qt 布局管理及焦点系统

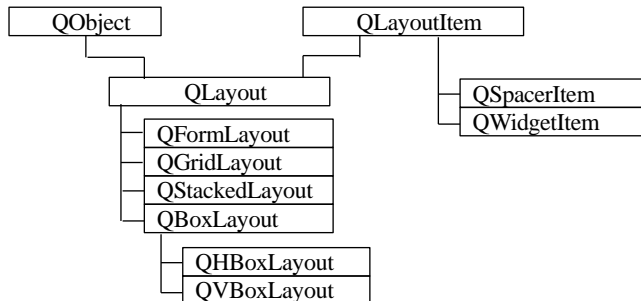
注意：本教程都假设读者在 pro 文件中已添加了正确的 `QT+=widgets` 语句，文中不再重复累述添加此语句。

本文注重讲解原理，因此使用的是手写的 Qt 程序，对于使用 Qt 设计师快速设计 Qt 程序会在专门章节讲解。

5.1 布局原理

一、布局基础

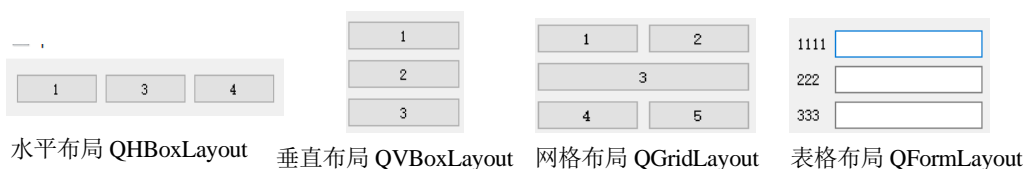
- 1、为讲解方便，把 Qt 的布局管理类或由这些类创建的对象称为布局管理器简称布局。
- 2、使用布局管理器的好处是可以不用为窗口中的每个部件设置其大小和位置，布局管理器会自动对这些部件进行排列，当窗口大小发生变化时，布局还会自动定位和调整部件的大小，当向布局中添加或移除一个部件时，布局会自动适应这些情况，总之使用布局能自动适应很多情形，为我们减少了大量负担。
- 3、Qt 布局管理系统使用的类的继承关系如下图：



- 4、QLayout 和 QLayoutItem 这两个类是抽象类，当设计自定义的布局管理器时才会使用到，通常使用的是由 Qt 实现的 QLayout 的几个子类。
- 5、布局项目(重要概念)：是指由布局管理的元素，包括 QWidget 部件、QLayout 布局，还有间距 QSpacerItem、QLayoutItem 等，其中 QLayoutItem 是用于描述由布局管理的项目的抽象类。
- 6、所有 QWidget 的子类都可以使用布局管理器来管理它们之中的子部件。
- 7、Qt 布局管理器的布局方式，有如下几种
 - ①、由 QHBoxLayout 类实现的水平布局，效果如下图
 - ②、由 QVBoxLayout 类实现的垂直布局，效果如下图

③、由 `QGridLayout` 类实现的二维网格布局，效果如下图

④、由 `QFormLayout` 实现的两列表格布局，效果如下图



8、Qt 使用布局管理器的步骤为，

①、首先创建一个布局管理器类的对象

②、然后使用该布局管理器类中的 `addWidget()` 函数，把需要由布局管理器管理的部件添加进来。还可使用 `addLayout()` 函数把其他布局管理器添加进来。

③、最后使用 `QWidget::setLayout()` 函数为窗口设置布局管理器。

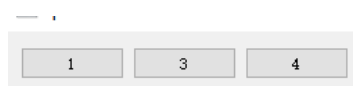
④、说明：

- 若为布局指定了父部件，则可以不使用 `QWidget::setLayout()` 函数，反之，调用 `QWidget::setLayout()` 函数安装布局，则可以不为该布局指定父部件。
- 不需要为添加到布局中的部件指定父部件，布局中的部件会自动成为安装布局的部件的子部件(使用 `QWidget::setParent()`)。注意：子部件的父部件不是布局，而是安装布局的部件。

⑤、下面为示例代码

```
QWidget w;  
//以下按钮会添加到布局中，所以都未设置父部件。  
QPushButton *pb1=new QPushButton("1");  
QPushButton *pb2=new QPushButton("3");  
QPushButton *pb3=new QPushButton("4");  
QHBoxLayout *ph= new QHBoxLayout; /*未指定父部件，因为使用了 setLayout 函数，该布局  
                                   是按水平方向排列的部件*/  
ph->addWidget(pb1); //把按钮添加到布局中  
ph->addWidget(pb2);  
ph->addWidget(pb3);  
w.setLayout(ph); //为窗口 w 安装布局  
w.show();
```

运行结果见右图



二、部件拉伸(Stretch)原理及大小策略

1、部件的大小策略 `sizePolicy`、大小限制、拉伸因子(Stretch Factors)的含义

①、部件的大小策略、大小提示、拉伸因子从三个方面对布局内的部件怎样进行拉伸以填满布局进行了说明。

②、拉伸因子：描述了各个部件在进行拉伸时，多个部件之间应以怎样的比例进行拉伸，比如把按钮 1、按钮 2、按钮 3 的拉伸因子分别为设置为 1, 2, 3，则按钮将按 1: 2: 3 的大小进行拉伸以填满整个布局空间(见下图)。注意：当主窗口的大小不能按计算出来的比例容纳下所有子部件时，子部件不一定会按设计好的比例进行排列。

1

2

3

③、大小策略：大小策略规定了部件以何种方式进行拉伸及压缩，比如部件不能被拉伸或压缩，部件不能被压缩得比大小提示更小等。

④、部件的大小限制：限制了部件可以被拉伸或压缩的范围，比如比如不能把部件压缩得比最小大小更小，或不能拉伸得比最大大小更大等。

2、QWidget 类中对部件大小进行限制的属性

①、**sizeHint**: const QSize //QWidget 类中的属性

访问函数: virtual QSize QWidget::sizeHint() const; //虚函数

部件的大小提示，就是指的部件的默认大小，提示的意思是 Qt 的建议或推荐。若部件没有设置布局，则默认实现会返回一个无效大小，否则返回布局的合适大小。注意：该属性只可读取不可设置，Qt 没有可以直接设置大小提示的函数或类，但可以重载 QWidget::sizeHint()虚函数来设置大小提示。

②、**minimumSizeHint**: const QSize //QWidget 类中的属性

访问函数: virtual QSize minimumSizeHint() const; //虚函数

- 部件的最小大小提示(只读属性)，若部件没有设置布局，则默认实现会返回一个无效大小，否则返回布局的最小大小，
- 大多数内置的部件都会重新实现 minimumSizeHint()虚函数。
- 若设置了 minimumSize 属性，则最小大小提示会被忽略。
- 除非设置了 minimumSize 属性或大小策略为 QSizePolicy::Ignore，否则布局管理器不会把部件的大小调整为比最小大小提示还要小的大小。
- 可重新实现该虚函数或设置 minimumSize 属性来改变最小大小提示的值。

③、**minimumSize**: QSize //QWidget 类中的属性

访问函数: QSize minimumSize() const;

void setMinimumSize(const QSize&);

void setMinimumSize(int m, int h);

- 部件的最小大小，部件不能被压缩得比最小大小更小的大小，若部件尺寸过小，则强制部件为最小大小。该属性的设置会覆盖由 QLayout 定义的最小大小。
- 默认值为 0。
- 要取消最小大小使用 QSize(0,0);

④、**minimumWidth** 和 **minimumHeight** 属性，这两个属性是 minimumSize 属性对应的宽度和高度属性。

⑤、**maximumSize**: QSize

访问函数: QSize maximumSize() const;

void setMaximumSize(const QSize&);

void setMaximumSize(int m, int h);

部件的最大大小，部件不能调整得比最大大小更大，默认值的宽和高都为 16777215。

3、设置拉伸因子的函数

在不同的布局管理器类中使用了不同的函数，下面列出 QBoxLayout 类中设置拉伸因子的函数，另外，对拉伸因子的设置还可通过 QSizePolicy 类进行设置。

```
bool QBoxLayout::setStretch(int index, int stretch); //为索引为 index(从 0 开始)的部件设置拉伸因子
bool QBoxLayout::setStretchFactor(QWidget* widget, int stretch); //为部件 widget 设置拉伸因子。
int QBoxLayout::stretch(int index) const; //返回索引 index 处的拉伸因子。
```

注：拉伸因子的设置应位于添加部件之后，否则将不起作用。

5、设置大小策略

①、sizePolicy: QSizePolicy //QWidget 类中的属性

访问函数: QSizePolicy sizePolicy() const;

void setSizePolicy(QSizePolicy);

void setSizePolicy(QSizePolicy::Policy horizontal, QSizePolicy::Policy vert);

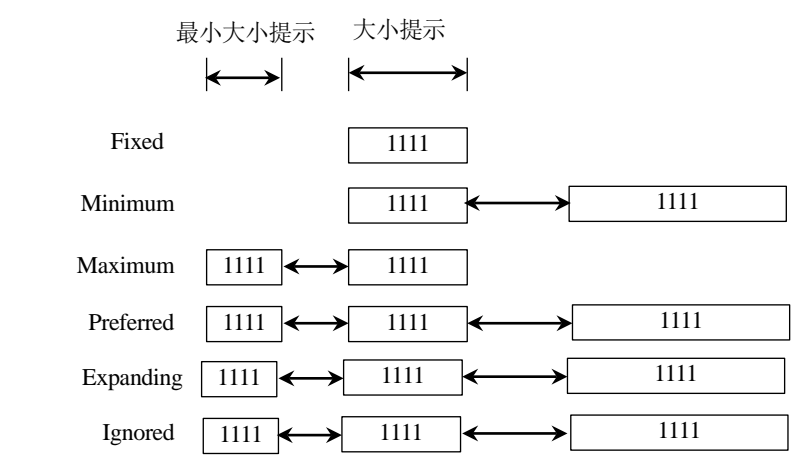
- QSizePolicy::Policy 枚举见下面的讲解
- 该属性描述了布局的大小策略，该属性需结合布局管理器使用，
- 若部件有一个布局进行管理，则使用该布局的大小策略，否则使用该属性的结果。
- 默认情况下部件是可以拉伸的，对于按钮、QLineEdit、QSpinBox 这种水平方向的部件，默认策略是可以水平拉伸，垂直固定，类似 QToolButton 这样的方形部件，默认策略是可以双向拉伸，支持不同方向的部件(比如 QSlider、QScrollBar 等)，只能在指定在相应的方向上拉伸。

②、QSizePolicy 类

- 该类用于描述布局的大小调整策略。
- QSizePolicy::Policy 枚举值描述的大小策略规则如下
 - 大小策略以大小提示作为拉伸或压缩的基本参考数据
 - 优先扩展权：是指部件会尽可能多地占用空间，有优先扩展权的策略若与其他策略同时存在时，拥有优先扩展权的部件将获得空间，比如 Preferred 与 Expanding 同时存在时，空间会分配给 Expanding。
 - 再次提醒：除 Ignore 外，若未设置最小大小提示或最小大小，则部件不能压缩得比大小提示更小的大小
 - 显示的设置最小大小、最大大小(注意，没有最大大小提示)会影响大小策略，比如若显示设置了最小大小和最大大小，则 Fixed 策略也能对部件进行拉伸或缩放。

QSizePolicy::Policy 枚举(无标志)	
描述部件的大小策略(下面的说明未考虑最大大小)	
成员	说明
QSizePolicy::Fixed	部件不可拉伸或压缩，其大小始终为 sizeHint 的大小。
QSizePolicy::Minimum	部件的大小提示是其最小大小，部件可以拉伸，但不能压缩得比大小提示更小。部件拉伸范围为 sizeHint~无限
QSizePolicy::Maximum	部件的大小提示是其最大大小，部件可以压缩，但不能拉伸得比大小提示更大。部件拉伸范围为 minimumSizeHint~sizeHint
QSizePolicy::Preferred	部件可任意拉伸和压缩，但大小提示仍是合适的大小。部件拉伸范围是 minimumSizeHint~无限
QSizePolicy::Expanding	部件可任意拉伸和压缩，但更希望是拉伸，设置该策略的部件拥有优先扩展权。部件拉伸范围是 minimumSizeHint~无限。

QSizePolicy::MinmumExpanding	部件的大小提示是其最小大小，部件可以拉伸，但不能压缩得比大小提示更小，设置该策略的 部件拥有优先扩展权 。部件拉伸范围为 sizeHint~无限。注：这种策略可能会被淘汰。
QSizePolicy::Ignored	忽略大小提示，部件可任意拉伸和压缩。部件拉伸范围为 minimumSizeHint~无限，注意：若 minimumSizeHint 的值为 0，则该策略可把部件压缩为 0 的大小(即部件不可见)，除该策略外，其余策略是不能把部件压缩为 0 的大小的。



说明：以 Expanding 为例，表示部件 1111 可以被压缩，但不会压缩得比最小大小提示更小，也可以被拉伸。

- QSizePolicy 类还包含一些对大小策略进行访问和设置的函数，这些函数都比较简单，从函数名就能知道其函数的用法，比如 QSizePolicy::setHorizontalPolicy(Policy) 表示设置水平方向的大小策略，QSize::setHorizontalStretch(int)表示设置水平方向的拉伸因子等等，因此就不一一列举出来了。下面仅列出其构造函数

```
QSizePolicy()
QSizePolicy(Policy hor, Policy ver, ControlType t = DefaultType);
```

其中参数 t 是 QSizePolicy::ControlType 枚举类型，该枚举描述的是需要设置大小策略的部件的类型，比如是 QSizePolicy::Label 表示这是一个 QLabel 实例，QSizePolicy::PushButton 表示这是一个 QPushButton 实例等。

6、大小策略与拉伸因子之间的关系

- ①、若部件的拉伸因子大于 0，则按照拉伸因子的比例分配空间，若拉伸因子为 0，则只有在其他部件不需要空间时才会获得空间，也就是说若一些部件拉伸因子大于 0，而一些部件拉伸因子为 0，则只有拉伸因子大于 0 的部件会被拉伸，而拉伸因子为 0 的部件不会被拉伸。若所有部件的拉伸因子都为 0，则按照大小策略的规则对部件进行拉伸。注意：若部件的大小策略为 Fixed，则即使设置了拉伸因子，该部件也不会被拉伸。以上规则可总结为，拉伸因子会使大小策略不起作用或失效(除了 Fixed 策略外)

- ②、任何部件，都不能压缩得比最小大小更小(若未设置最小大小，则为最小大小提示)，任何部件都不能拉伸得比最大大小更大。

示例 5.1：大小限制对大小策略的影响

```
#include<QtWidgets>
#include <iostream>
using namespace std;
int main(int argc, char *argv[]){    QApplication a(argc, argv);
    QWidget w;
    QPushButton *pb=new QPushButton("Fixed");
    QPushButton *pb1=new QPushButton("MaxSetMin");
    QPushButton *pb2=new QPushButton("MaxNoMin");
    //每部件设置大小策略
    pb->setSizePolicy(QSizePolicy::Fixed, QSizePolicy::Fixed);
    pb1->setSizePolicy(QSizePolicy::Maximum, QSizePolicy::Fixed);
    pb2->setSizePolicy(QSizePolicy::Maximum, QSizePolicy::Fixed);
    QHBoxLayout *pg=new QHBoxLayout;
    pb->resize(222, 222);    //使用布局后，resize 函数将不再起作用
    pb->setMinimumWidth(11);    pb->setMaximumWidth(188);    //为 pb 设置最大/最小大小
    pb1->setMinimumWidth(1);    //为 pb1 设置最小大小
    pg->addWidget(pb);    pg->addWidget(pb1);    pg->addWidget(pb2);    w.setLayout(pg);
    w.resize(300, 200);    w.show();    return a.exec();    }
```

运行结果及说明如下



由图可见，策略为 Fixed 的按钮在显示设置了最大/小大小后，该按钮仍可进行拉伸或压缩
策略为 Maximum 设置了最小大小的按钮 MaxSetMin 可进行压缩，但相同策略的未设置最小大小
的按钮 MaxNoMin 未压缩，仍然保持其大小提示的大小。

示例 5.2：部件的优先扩展权

```
#include<QtWidgets>
#include <iostream>
using namespace std;
int main(int argc, char *argv[]){    QApplication a(argc, argv);
    QWidget w;
    QPushButton *pb=new QPushButton("Preferred");
    QPushButton *pb1=new QPushButton("Expanding");    //该部件具有优先扩展权
    pb->setSizePolicy(QSizePolicy::Preferred, QSizePolicy::Fixed);
    pb1->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Fixed);
    QHBoxLayout *pg=new QHBoxLayout;    pg->addWidget(pb);    pg->addWidget(pb1);
    w.setLayout(pg);    w.resize(300, 200);    w.show();    return a.exec();    }
```

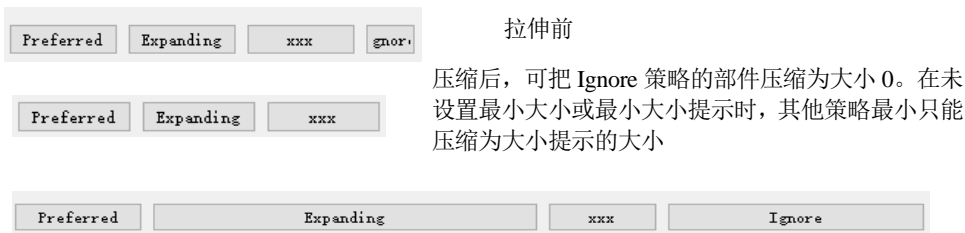

运行结果及说明



示例 5.3：拉伸因子与大小策略的关系

```
#include<QtWidgets>
#include <iostream>
using namespace std;
int main(int argc, char *argv[]) {    QApplication a(argc, argv);
    QWidget w;
    QPushButton *pb=new QPushButton("Preferred");
    QPushButton *pb1=new QPushButton("Expanding");
    QPushButton *pb2=new QPushButton("xxx");
    QPushButton *pb3=new QPushButton("Ignore");
    pb->setSizePolicy(QSizePolicy::Preferred,QSizePolicy::Fixed);
    pb1->setSizePolicy(QSizePolicy::Expanding,QSizePolicy::Fixed);
    pb2->setSizePolicy(QSizePolicy::Expanding,QSizePolicy::Fixed);
    pb3->setSizePolicy(QSizePolicy::Ignored,QSizePolicy::Fixed);
    QHBoxLayout *pg=new QHBoxLayout;
    pg->addWidget(pb);        pg->addWidget(pb1);
    pg->addWidget(pb2);        pg->addWidget(pb3);
    //拉伸因子应位于 addWidget() 之后, 否则拉伸因子将不起作用, 设置拉伸因子后扩展优先权将不起作用。
    pg->setStretch(0,1);        pg->setStretchFactor(pb1,3);    pg->setStretch(3,2);
    w.setLayout(pg);        w.resize(300,100);        w.show();    return a.exec();    }
```

运行结果及说明



除了 xxx 按钮, 其他按钮都拉伸了。设置拉伸因子后部件将按拉伸因子的比例进行拉伸, 未设置拉伸因子(即拉伸因子为 0)的按钮将不会被拉伸(比如本例的 xxx), 此时按钮 xxx 设置的大小策略将不起作用(除了 Fixed 策略外)

三、大小约束(主窗口最大最小大小的设置)

- 1、主窗口是指安装布局管理器的部件。
- 2、当显示设置了主窗口的最小大小后, 即使子部件大小策略为 Fixed 也能对部件进行压缩,

若不设置主窗口的最小大小，若子部件大小策略为 Fixed，则部件不会拉伸(示例略，读者可自行验证)。

3、大小约束(QLayout::sizeConstraint 属性)

大小约束是由属性 QLayout::sizeConstraint 进行描述的，大小约束其实就是用来设置主窗口的最大和最小值的，该属性原型如下：

sizeConstraint: SizeConstraint //注意大小写字母 S

访问函数: sizeConstraint sizeConstraint() const; void setSizeConstraint(SizeConstraint);

其中 SizeConstraint 是 QLayout 类中的枚举，如下表

QLayout::SizeConstraint 枚举

注: QLayout::minimumSize()~QLayout::maximumSize()的大小值，Qt 会根据子部件的大小策略自动进行计算

成员	值	说明
QLayout::SetDefaultConstraint	0	把主窗口的最小大小设置为 QLayout::minimumSize(), 除非主窗口已经具有最小大小，也就是说若主窗口已显示设置了最小大小，则此约束不会再使用 QLayout::minimumSize()设置主窗口的最小大小。这是默认值
QLayout::SetFixedSize	3	主窗口的大小设置为 sizeHint();此时主窗口大小无法被调整，通常使用此约束使主窗口自动适应所布置的部件的大小。
QLayout::SetMinimumSize	2	把主窗口的最小大小设置为 QLayout::minimumSize(), 主窗口不能调整为比该大小更小。
QLayout::SetMaximumSize	4	把主窗口的最大大小设置为 QLayout::maximumSize(), 主窗口不能调整为比该大小更大。
QLayout::SetMinAndMaxSize	5	把主窗口的大小设置为 QLayout::minimumSize()~QLayout::maximumSize()
QLayout::SetNoConstraint	1	不对主窗口的大小进行设置(即没有约束), 也就是说, 保持主窗口之前的最大和最小大小, 不对其进行更改。

示例 5.4: 大小约束(用于设置主窗口最大最小大小)

```
//m.h 文件的内容
#ifndef M_H
#define M_H
#include<QtWidgets>
#include<QDebug>
#include <iostream>
using namespace std;
class B:public QPushButton{    Q_OBJECT
public:    QWidget* pw;    QLayout *pl;    //创建两个数据成员用于传递数据
        B(QString s="",QWidget* p=0):QPushButton(s,p) {}
public slots:
    void f() {
        pw=parentWidget();    //获取当前按钮的父部件
        pl=pw->layout();    //获取父部件的布局管理器，layout()是QWidget类中的函数
        QString t=text();    //获取当前按钮显示的文本
        if(t=="Default")    //如果按下的是Default按钮
            { pl->setSizeConstraint(QLayout::SetDefaultConstraint);} //设置大小约束
    }
};
#endif
```

```

        if(t=="Fixed"){ p1->setSizeConstraint(QLayout::SetFixedSize); }
        if(t=="Maxi"){ p1->setSizeConstraint(QLayout::SetMaximumSize); }
        if(t=="MinAndMax"){ p1->setSizeConstraint(QLayout::SetMinAndMaxSize); }
        if(t=="NoCon"){ p1->setSizeConstraint(QLayout::SetNoConstraint); }
        if(t=="CusMin"){ pw->setMinimumSize(22,22); } //手动设置主窗口的最小值
        pw->activate(); //重新设置主窗口的布局, 此函数必须调用, 否则主窗口的数据不会立即更新
        g(); } //调用函数 g 显示当前父部件的最大/最小值

void g(){ //用于输出当前父部件的最大最小值。
    cout<<"minW="<<pw->minimumSize().width()<<" ";
    cout<<"minH="<<pw->minimumSize().height()<<" ";
    cout<<"maxW="<<pw->maximumSize().width()<<" ";
    cout<<"maxH="<<pw->maximumSize().height()<<endl; }
};
#endif // M_H

//m.cpp 文件的内容
#include "m.h"
int main(int argc, char *argv[]){ QApplication a(argc, argv);
    QWidget w; B *pb, *pb1, *pb2, *pb3, *pb4, *pb5; QHBoxLayout *pg;
    pb=new B("Default"); pb1=new B("Fixed"); pb2=new B("Maxi");
    pb3=new B("MinAndMax"); pb4=new B("NoCon"); pb5=new B("CusMin");
    //设置各子部件的大小策略
    pb->setSizePolicy(QSizePolicy::Fixed, QSizePolicy::Fixed);
    pb1->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Fixed);
    pb2->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Fixed);
    pb3->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Fixed);
    pb4->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Fixed);
    pb5->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Fixed);
    //把子部件添加到布局中
    pg=new QHBoxLayout;
    pg->addWidget(pb); pg->addWidget(pb1); pg->addWidget(pb2);
    pg->addWidget(pb3); pg->addWidget(pb4); pg->addWidget(pb5);
    w.setLayout(pg);
    //关联信号和槽
    QObject::connect(pb, &QPushButton::clicked, pb, &B::f);
    QObject::connect(pb1, &QPushButton::clicked, pb1, &B::f);
    QObject::connect(pb2, &QPushButton::clicked, pb2, &B::f);
    QObject::connect(pb3, &QPushButton::clicked, pb3, &B::f);
    QObject::connect(pb4, &QPushButton::clicked, pb4, &B::f);
    QObject::connect(pb5, &QPushButton::clicked, pb5, &B::f);
    w.resize(300,100); w.show(); return a.exec(); }

```

运行结果及说明

初次运行效果

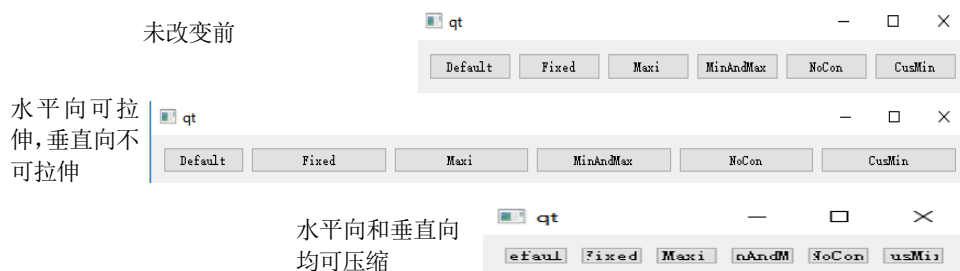


按以下顺序点击按钮：

- ①、点击 Default 按钮后，输出为：minW=502, minH=45, maxW=16777215, maxH=16777215，这是 SetDefaultConstraint 约束设置的主窗口的最大/最小值，其中最大值未设置(16777215 是最初的默认值未被改变)，最小值被调置为(502,45)，此时主窗口可拉伸，但不可压缩得比最小值小
- ②、点击 Fixed 按钮后，输出为：minW=502,minH=45,maxW=502,maxH=45，可见 SetFixedSize 约束把主窗口的最大/最小值都设置为了(502,45)，此时主窗口大小不可更改(因为最大最小值相等)，其效果如下图



- ③、点击 Maxi 按钮后，输出：minW=502,minH=45,maxW=524287,maxH=45，可见 SetMaximumSize 约束把最大值设置为了(524284,45)，但是最小值未被重新设置。此时主窗口可水平拉伸，但不能垂直拉伸，因为垂直方向最大最小值相等
- ④、点击 NoCon 输出的是上次设置的大小，也就是说 SetNoConstraint 约束未对主窗口的最大最小值进行设置，主窗口保持上次设置的值。主窗口仍可进行水平拉伸不可进行垂直拉伸
- ⑤、点击 CusMin 手动改变主窗口的最小值，输出 minW=22, minH=22, maxW=524287, maxH=45，此时主窗口可水平拉伸和压缩，但不能进行垂直拉伸和压缩。其效果如下图



显示设置最小大小后，即使大小策略为 Fixed 的子部件 Default 也能被压缩，但仍不可拉伸

- ⑥、再次点击 NoCon 按钮，验证 SetNoConstraint 约束不会对最大最小值进行设置，此时输出的是上次设置的大小 minW=22,minH=22,maxW=524287,maxH=45。
- ⑦、再次点击 Default 按钮，验证 SetDefaultConstraint 约束不会再次对最大最小值进行设置，此时输出的是上次设置的大小 minW=22,minH=22,maxW=524287,maxH=45。
- ⑧、验证 SetMaximumSize 约束只会设置最大值而不会设置最小值，步骤如下
点击 Fixed 按钮，此时主窗口大小更改为(502,45,502,45)，再点击 CusMin 按钮，主窗口大小被更改为(22,22,502,45)，接着点击 Maxi 按钮，输出 minW=22, minH=22, maxW=524287, maxH=45
- ⑨、其余规则读者可自行验证

4、总结：大小约束与大小策略的关系

- ①、大小约束是用来改变主窗口最大最小大小的，主窗口的最大最小大小决定了主窗口是否可进行拉伸或压缩，而子部件(比如本例的按钮)是否可被拉伸和压缩是由各子部件的大小策略决定的，因此大小约束和大小策略一个决定主窗口是否可拉伸压缩，一个决定了子部件是否可拉伸压缩。
- ②、系统计算主窗口的最大最小大小时会根据子部件的大小策略进行计算，比如本例的子部件在垂直方向的大小策略为 Fixed(即固定不变)，则主窗口依此计算出来的在垂直方向的最大和最小高度始终为 45，而水平方向各子部件的大小策略是 Expanding(可拉伸)，因此主窗口在水平方向计算出来的最大大小可以为 524287，读者可把本例子部件垂直方向的大小策略更改为 Expanding，再点击以上按钮来观察主窗口在垂直方向的最大最小值有何变化。

四、内容边距(ContentsMargins)、间距(spacing)和 QSpacerItem 类

- 1、内容边距：就是页边距，指的是布局中的各子部件(内容)与周围四个边的距离，内容边距比较简单，下面列出需要使用到的函数

```
QMargins QLayout::contentsMargins() const;
void QLayout::getContentsMargins(int *left, int *top, int *right, int *bottom) const;
void QLayout::setContentsMargins(int left, int top, int right, int bottom);
void QLayout::setContentsMargins(const QMargins& margin);
```

设置和返回内容边距，默认情况下使用的是样式提供的值，在大多数平台上，默认为 11 像素。

- 2、间距 spacing：指的是各部件之间的距离，
- 3、QSpacerItem 类

使用该类可以创建自定义的间距，使用该类创建的间距相当于是一个空白部件，它是布局中的一个项目，会在布局中占据一个位置，布局会为其分配一个索引号，也就是说由 QSpacerItem 类创建的对象是可以由布局管理器进行管理的。通常不需要使用这个类，因为在各布局管理器中有相应的函数代替了该类的功能。该类包括析构函数在内，总共只有 4 个公有的成员函数，其余函数都是从其父类继承而来的受保护的函数，下面列出这些成员函数

- ①、

```
QSpacerItem(int w, int h, QSizePolicy::Policy hPolicy = QSizePolicy::Minimum,
            QSizePolicy::Policy vPolicy = QSizePolicy::Minimum) //构造函数
void changeSize(int w, int h, QSizePolicy::Policy hPolicy = QSizePolicy::Minimum,
                QSizePolicy::Policy vPolicy = QSizePolicy::Minimum)
```

以上函数表示构造或修改一个具有宽度为 w，高度为 h，水平方向的大小策略为 hPolicy，垂直方向大小策略为 vPolicy 的间距(即 QSpacerItem 实例)。

- ②、

```
QSizePolicy sizePolicy() const; //返回此间距使用的大小策略。
```

- 4、因不同布局管理器使用的设置间距的函数不相同，下面先讲解 QBoxLayout 布局中的间距函数。注意：向布局中添加或插入的间距是 QSpacerItem，因此布局会为 QSpacerItem 分配一个索引号，这会改变各部件之间的原有索引号，比如在索引为 0 的按钮 1 和索引为 1 的按钮 2 之间插入一个 QSpacerItem 之后，则按钮 1 的索引仍为 0，QSpacerItem 的索引为 1，

按钮 2 的索引为 2,

①、`void QBoxLayout::addSpacing(int size);`

把大小为 size 的不可拉伸间距(QSpacerItem)添加到布局的末尾。

②、`void QBoxLayout::insertSpacing(int index, int size);`

在索引 index 处, 插入大小为 size 的不可拉伸间距(QSpacerItem)。若索引为负, 则在最后添加空格。

③、`void QBoxLayout::addStretch(int stretch = 0);`

把具有零最小大小和拉伸因子为 stretch 的可拉伸间距(QSpacerItem)添加到布局的末尾。

④、`void QBoxLayout::insertStretch(int index, int stretch = 0);`

在索引 index 处, 插入最小大小为 0, 拉伸因子为 stretch 的可拉伸间距(QSpacerItem)。若索引为负, 则在最后添加空格。

⑤、`void QBoxLayout::insertSpacerItem(int index, QSpacerItem* spacerItem);`

在索引 index 处, 插入最小大小为 0, 拉伸因子为 0 的 spacerItem。若索引为负, 则在最后添加空格。

⑥、`void QBoxLayout::addSpacerItem(QSpacerItem* spacerItem);`

把 spacerItem 添加到布局的末尾。

5、下面为 QLayout 类中与间距有关的属性

spacing: int //QLayout 类中的属性

访问函数: void setSpacing(int); int spacing() const;

- 设置和获取布局内部部件之间的间距, 若未设置值, 则从父布局或父窗口部件的样式继承。
- 若在 QGridLayout 和 QFormLayout 布局中使用 setHorizontalSpacing() 和 setVerticalSpacing 在水平和垂直方向设置了不同的间距, 则 Spacing()函数返回-1。
- 设置该属性会使所有部件之间都有所设置的间距。
- 注意: spacing 属性设置的间距不是一个 QSpacerItem, 它不是布局中的一个项目, spcing 属性只是设置了两部件之间的距离, 在这之间未插入任何东西, 因此布局不会为其分配索引号。

示例 5.5: 间距简单应用

```
#include<QtWidgets>
#include <iostream>
using namespace std;
int main(int argc, char *argv[]){    QApplication a(argc, argv);
    QWidget w;
    QPushButton *pb=new QPushButton("AAA");
    QPushButton *pb1=new QPushButton("BBB");
    QPushButton *pb2=new QPushButton("CCC");
    QPushButton *pb3=new QPushButton("DDD");
    pb->setSizePolicy(QSizePolicy::Preferred, QSizePolicy::Fixed);
    pb1->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Fixed);
    pb2->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Fixed);
    QHBoxLayout *pg=new QHBoxLayout;
```

```

pg->addWidget(pb);
pg->addStretch(1); //添加一个拉伸因子为 1 的间距 (QSpacerItem)
pg->addWidget(pb1);
pg->addStretch(2); //添加一个拉伸因子为 2 的间距
pg->addWidget(pb2);
w.setLayout(pg);
w.resize(300, 100);      w.show();      return a.exec(); }

```

运行结果及说明



这是添加的两个间距(虚线框实际显示时是没有的), 这两个间距将按照 1: 2 的比例显示, **QSpacerItem** 间距相当于空白的部件, 因此这里相当于添加了两个含了拉伸因子的空白部件, 这会导致所有按钮的大小策略将不起作用, 即按钮不会被拉伸(注: 含有拉伸因子的部件会使其他部件的大小策略失效)

示例 5.6: 间距与空白部件

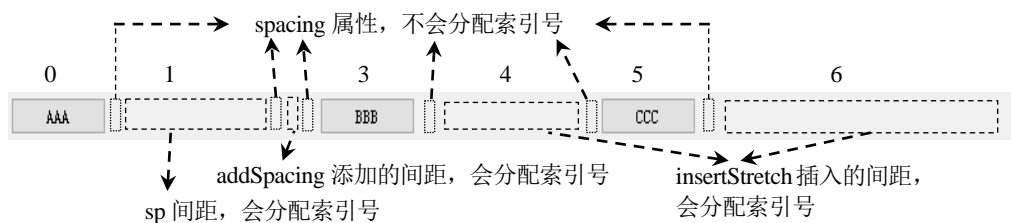
```

#include<QtWidgets>
#include <iostream>
using namespace std;
int main(int argc, char *argv[]) {    QApplication a(argc, argv);
    QWidget w;
    QPushButton *pb=new QPushButton("AAA");
    QPushButton *pb1=new QPushButton("BBB");
    QPushButton *pb2=new QPushButton("CCC");
    QHBoxLayout *pg=new QHBoxLayout;
    QSpacerItem *sp=new QSpacerItem(22, 22); //创建一个宽和高都为 22 的 QSpacerItem
    pg->setSpacing(22); /*设置 spacing 属性, 使每个子部件之间的间距为 22, spacing 属性设置的间距不是空白部件, 不会被分配索引号*/

    pg->addWidget(pb);
    pg->addSpacing(22); //添加一个间距(QSpacerItem), 该间距是空白间距, 会分配一个索引号。
    pg->addWidget(pb1);
    pg->addWidget(pb2);
    pg->insertSpacerItem(1, sp); /*把 QSpacerItem 实例 sp 插入索引为 1 的位置, 注意 sp 是空白部件, 会分配一个索引号。*/
    pg->setStretch(1, 1); //把索引为 1 的部件(即上一行插入的 sp)的拉伸因子设置为 1
    pg->insertStretch(4, 1); //插入一个拉伸因子为 1 的间距(QSpacerItem)
    pg->insertStretch(6, 2);
    w.setLayout(pg);      w.resize(300, 100);      w.show();      return a.exec(); }

```

运行结果及说明



五、嵌套布局及布局位于容器中

- 1、嵌套布局就是指的布局管理器中安装其他布局管理器的情况。使用嵌套布局，需使用布局管理器类中的 `addLayout()`或 `insertLayout()`函数将子布局安装进来，因各种类型的布局管理器对嵌套布局的处理情况不相同，因此本小节以 `QBoxLayout` 及其子类为例进行讲解。
- 2、`QBoxLayout` 类中用于添加布局的函数的原型如下

```
void QBoxLayout::addLayout(QLayout* layout, int stretch = 0);    //在末尾添加布局。
```

```
void QBoxLayout::insertLayout(int index, QLayout* layout, int stretch = 0);
```

在指定索引处插入布局，并设置其拉伸因子为 `stretch`，若 `index` 为负，则添加到末尾。

示例 5.7：嵌套布局的使用

```
#include <QtWidgets>
#include <iostream>
using namespace std;
int main(int argc, char *argv[]) {    QApplication a(argc, argv);
    QWidget w;        QPushButton *pb, *pb1, *pb2, *pb3, *pb4, *pb5, *pb6, *pb7, *pb8;
    pb=new QPushButton("AAA");        pb1=new QPushButton("BBB");        pb2=new QPushButton("CCC");
    pb3=new QPushButton("DDD");        pb4=new QPushButton("EEE");        pb5=new QPushButton("FFF");
    pb6=new QPushButton("GGG");        pb7=new QPushButton("HHH");        pb8=new QPushButton("III");

    QBoxLayout *pg=new QHBoxLayout;        QVBoxLayout *pv=new QVBoxLayout;
    QVBoxLayout *pv1=new QVBoxLayout;
    pg->addWidget(pb);        pg->addLayout(pv, 1);    //添加子布局后，子布局会占一个索引位置
    pg->addWidget(pb1);        pg->addWidget(pb2);
    pg->insertLayout(3, pv1, 2);    //插入布局
    pv->addWidget(pb3);        pv->addWidget(pb4);        pv->addWidget(pb5);
    pv1->addWidget(pb6);        pv1->addWidget(pb7);        pv1->addWidget(pb8);
    w.setLayout(pg);        w.resize(300, 100);        w.show();    return a.exec(); }
```

运行结果及说明



这是添加进来的子布局，他们将按照 1：2 的比例缩放。

3、布局位于容器中：布局也可添加到一个容器中，然后把该容器添加到窗口中，可以使用 QWidget 作为容器。因为比较简单，下面列举一示例作为演示

示例 5.8：布局位于容器中

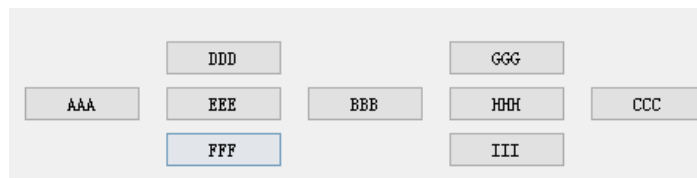
```
#include<QtWidgets>
#include <iostream>
using namespace std;
int main(int argc, char *argv[]) {    QApplication a(argc, argv);
    QWidget w;                      QPushButton *pb, *pb1, *pb2, *pb3, *pb4, *pb5, *pb6, *pb7, *pb8;
    QWidget *pw=new QWidget;        QWidget *pw1=new QWidget;    //创建容器
    pb=new QPushButton("AAA");      pb1=new QPushButton("BBB");      pb2=new QPushButton("CCC");
    pb3=new QPushButton("DDD");      pb4=new QPushButton("EEE");      pb5=new QPushButton("FFF");
    pb6=new QPushButton("GGG");      pb7=new QPushButton("HHH");      pb8=new QPushButton("III");

    QHBoxLayout *pg=new QHBoxLayout;    QVBoxLayout *pv=new QVBoxLayout;
    QVBoxLayout *pv1=new QVBoxLayout;

    pg->addWidget(pb);
    pg->addWidget(pw);                //把容器添加到布局 pg 中
    pg->addWidget(pb1);      pg->addWidget(pw1);      pg->addWidget(pb2);

    pv->addWidget(pb3);      pv->addWidget(pb4);      pv->addWidget(pb5);
    pv1->addWidget(pb6);      pv1->addWidget(pb7);      pv1->addWidget(pb8);
    w.setLayout(pg);
    pw->setLayout(pv);    //为容器安装布局
    pw1->setLayout(pv1);
    w.resize(300, 100);      w.show();      return a.exec();    }
```

运行结果及说明



5.2 各布局管理器类

QBoxLayout、QBoxLayout、QFormLayout

一、QBoxLayout 及其子类(盒式布局)

1、QBoxLayout 盒式布局管理器，可以创建水平或垂直方向的布局管理器，通常使用的是该类的两个子类 QHBoxLayout 和 QVBoxLayout，因为子类有比较方便的构造函数，水平和垂直布局管理器在前面一小节已使用得比较多了

2、QBoxLayout 类中的枚举见下表

QBoxLayout::Direction 枚举(无标志)

作用：用于描述盒式布局的方向

成员	值	说明	成员	值	说明
QBoxLayout::LeftToRight	0	水平从左到右	QBoxLayout::TopToBottom	2	垂直从上到下
QBoxLayout::RightToLeft	1	水平从右到左	QBoxLayout::BottomToTop	3	垂直从下到上

3、QBoxLayout 类中的成员函数

- ①、`QBoxLayout(Direction dir, QWidget* parent = Q_NULLPTR);` //构造一个方向为 dir 的盒式布局
- ②、`void addLayout(QLayout* layout, int stretch = 0);`
`void insertLayout(int index, QLayout* layout, int stretch = 0);`
把布局 layout 添加到末尾或在指定索引 index 处插入布局，并设置其拉伸因子为 stretch，若 index 为负，则添加到末尾。
- ③、`void addWidget(QWidget* widget, int stretch = 0, Qt::Alignment m = Qt::Alignmnet());`
`void insertWidget(int index, QWidget* widget, int stretch = 0, Qt::Alignment m = Qt::Alignmnet());`
把部件 widget 添加到布局的末尾或添加到索引 index 处，并设置其拉伸因子为 stretch，对齐方式为 m，默认对齐方式为 0，表示部件填充整个单元格。若 index 为负，则插入到末尾。Qt::Alignment 属性见公共枚举章节。注意：对齐方式会对部件的拉伸效果产生影响，非零对齐表示不应拉伸以填充其可用空间。
- ④、`void addSpacing(int size);`
`void insertSpacing(int index, int size);`
在末尾添加或在索引 index 处插入大小为 size 的不可拉伸间距(QSpacerItem)。若索引为负，则添加到末尾。
- ⑤、`void addStretch(int stretch = 0);`
`void insertStretch(int index, int stretch = 0);`
在末尾添加或在索引 index 处，插入最小大小为 0，拉伸因子为 stretch 的可拉伸间距(QSpacerItem)。若索引为负，则添加到末尾。
- ⑥、`void addSpacerItem(QSpacerItem* spacerItem);`
`void insertSpacerItem(int index, QSpacerItem* sp);`

在末尾添加或在索引 index 处，插入最小大小为 0，拉伸因子为 0 的 spacerItem。若索引为负，则添加到末尾。

- ⑦、`int spacing() const;` //QLayout::spacing()的重新实现，设置 spacing 属性
- `void setSpacing(int spacing);` //QLayout::setSpacing()的重新实现，设置 spacing 属性
- ⑧、`void setStretch(int index, int stretch);`
- `bool setStretchFactor(QWidget* widget , int stretch);`
- `bool setStretchFactor(QLayout* layout, int stretch);`

以上函数分别表示，设置索引为 index 的部件或布局 layout 或部件 widget 的拉伸因子为 stretch，若 layout 或 widget(不包括子布局)在布局中，则返回 true，否则返回 false。

- ⑨、`int stretch(int index) const;` //返回索引为 index 的拉伸因子。
- ⑩、`Direction direction() const;` //返回布局的方向。
- ⑪、`void setDirection(Direction dir);` //设置布局的方向为 dir
- ⑫、`virtual int count() const;` //虚函数，QLayout::count()的重新实现。返回布局中的项目数。
- ⑬、`void addStrut(int size);`

把盒式布局的正交尺寸限制在最小值 size 大小。其中正交尺寸是指，对于垂直布局则是指水平方向的尺寸，对于水平布局则是指垂直方向的尺寸。

- ⑭、`void insertItem(int index, QLayoutItem* item);`
- ⑮、`virtual QLayoutItem* takeAt(int index)` //虚函数，QLayout::takeAt()的重新实现

从布局中删除指定索引 index 处的项目，并返回该项目，若没有这样的项目，则返回 0，删除项目后，布局中的其他项目会重新编号索引。安全删除的方式为

`QLayoutItem *p; if(p=layout->takeAt(0)!=0) delete p;`

- ⑯、`virtual QLayoutItem* itemAt(int index) const;` //虚函数，QLayout::itemAt()的重新实现

返回指定索引 index 处的项目，若没有这样的项目，则返回 0。

4、QHBoxLayout 和 QVBoxLayout 类中的成员函数(仅有构造函数和析构函数)

```
QHBoxLayout();           QHBoxLayout(QWidget* parent);
QVBoxLayout();           QVBoxLayout(QWidget* parent);
```

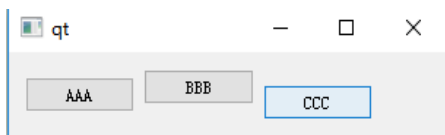
示例 5.9：对齐方式、正交尺寸

```
#include<QtWidgets>
#include <iostream>
using namespace std;
int main(int argc, char *argv[]) {    QApplication a(argc, argv);
    QWidget w;
    QPushButton *pb=new QPushButton("AAA");    QPushButton *pb1=new QPushButton("BBB");
    QPushButton *pb2=new QPushButton("CCC");    QHBoxLayout *pg=new QHBoxLayout;

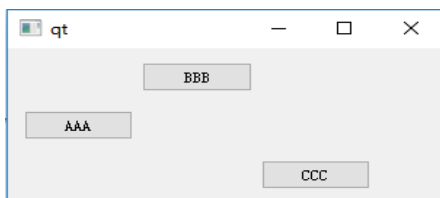
    pg->addWidget(pb,1,Qt::AlignLeft);    //pb 设置为在可用空间中左对齐
    pg->addWidget(pb1,2,Qt::AlignTop);
    pg->addWidget(pb2,3,Qt::AlignBottom|Qt::AlignLeft);
    pg->addStrut(100);    //把水平布局的垂直最小高度设置为 100
    w.setLayout(pg);
    w.resize(300,55);    //主窗口大小为 45
```

```
w.show();    return a.exec(); }
```

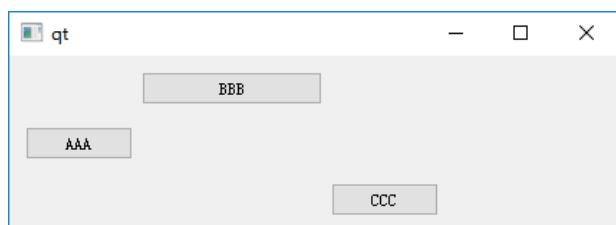
运行结果及说明



注释掉 `pg->addStruct(100)` 的效果



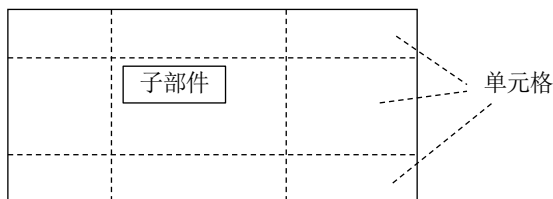
未注释掉 `pg->addStruct(100)` 的效果，此时主窗口的高度不能小于 100(不含标题栏和边框等)



把主窗口拉伸之后的效果，可见各按钮在其可用空间内按各自的方式进行对齐。在垂直方向按钮默认是不可拉伸的，可见水平方向的对齐方式会影响部件在水平方向的拉伸。

二、QGridLayout 类(网格布局)

1、网格布局原理(见下图):



基本原理是把窗口划分为若干个单元格，每个子部件被放置于一个或多个单元格之中，各单元格的大小可由拉伸因子和一行或列中单元格的数量来确定，若子部件的大小(由 `sizeHint()` 确定)小于单元格，则可以设置该子部件位于单元格的什么位置(顶部、左侧、底部等)，还可设置该子部件是否可以拉伸以填满该单元格等等。前文讲的 `QBoxLayout` 布局就可以看成是只有一行或一列的网格布局。

2、QGridLayout 类中的属性

`horizontalSpacing: int` 访问函数: `int horizontalSpacing() const; void setHorizontalSpacing(int spacing);`

`verticalSpacing: int` 访问函数: `int verticalSpacing()const; void setVerticalSpacing(int spacing);`

以上属性用于获取或设置垂直和水平方向子部件之间的间距。以上属性其实设置的是 `spacing` 属性(注意: 不是 `QSpacerItem`)，布局不会为这些间距分配索引号。

3、QGridLayout 类中的函数

①、`QGridLayout()`:

`QGridLayout(QWidget* parent);` //构造函数

```

②、 void addLayout(QLayout* layout, int row, int column, Qt::Alignment alignment = Qt::Alignment());
      void addLayout(QLayout* layout, int row, int column, int rowSpan, int columnSpan,
                    Qt::Alignment alignment = Qt::Alignment());
      void addWidget(QWidget* widget, int row, int column, Qt::Alignment alignment = Qt::Alignment());
      void addWidget(QWidget* widget, int row, int column, int rowSpan, int columnSpan,
                    Qt::Alignment alignment = Qt::Alignment());
      void addItem(QLayoutItem* item, int row, int column, int rowSpan = 1, int columnSpan = 1,
                  Qt::Alignment alignment = Qt::Alignment());

```

以上函数表示，在第 row 行，第 column 列单元格添加一个跨越 rowSpan 行 columnSpan 列单元格的项目(layout、widget 或 item)，其对齐方式为 alignment。若 rowSpan 或 columnSpan 为-1，则将 item 分别扩展至底部或右侧边缘。默认对齐方式为 0，表示部件会填充整个单元格，非零对齐方式不应增长以填充可用空间。左上角为第 0 行 0 列。注：addItem 是向 QGridLayout 布局中添加 QSpacerItem 间距的方法，QSpacerItem 间距是布局中的项目，它会占据一个位置，布局会为其分配索引号。

```

③、 int columnCount() const;           返回列数
      int rowCount() const;             返回行数
④、 int columnMinimumWidth(int column) const;  返回 column 列最小宽度
      void setColumnMinimumWidth(int column, int minSize); 把 column 列的最小宽度设置为 minSize 像素。
      int rowMinimumHeight(int row) const;      返回 row 行的最小高度。
      void setRowMinimumHeight(int row, int minSize); 把 row 行的最小高度设置为 minSize 像素。

```

注意：以上函数设置的是单元格的最小高度和宽度。

```

⑤、 int columnStretch(int column) const;      返回 column 列的拉伸因子
      void setColumnStretch(int column, int stretch); 把 column 列的拉伸因子设置为 stretch。
      int rowStretch(int row) const;           返回 row 行的拉伸因子。
      void setRowStretch(int row, int stretch); 把 row 行的拉伸因子设置为 stretch。
⑥、 void setSpacing(int spacing);             把垂直和水平间距都设置为 spacing
      int spacing() const;                     若垂直和水平间距相等，则返回该值，否则返回-1。

```

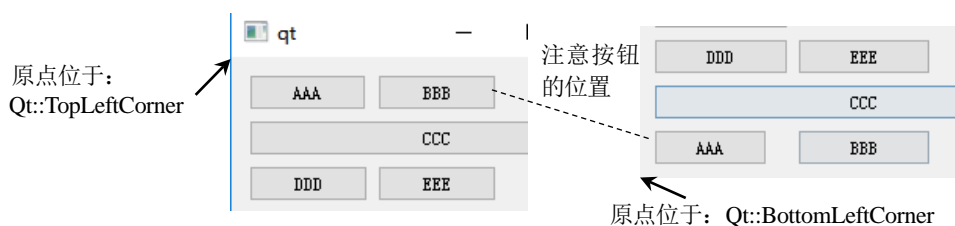
以上两函数设置的是 spcing 属性，它不是一个 QSpacerItem。

```

⑦、 Qt::Corner originCorner() const;          返回用于网格原点的角落
      void setOriginCorner(Qt::Corner corner); 把 corner 设置为网格的原点位置，即(0,0);

```

以上两个函数用于设置原点位于布局的哪个角落(左上角(默认)、右下角等)，见下面的图示，其中 Qt::Corner 枚举对角落进行了描述，详见公用枚举章节。



⑧、`QRect cellRect(int row, int column) const;`

返回(row, column)处的单元格的几何形状。若 row 或 column 在网格之外，则返回无效的矩形。

⑨、`void getItemPosition(int index, int *row, int *column, int *rowSpan, int *columnSpan) const;`

返回索引为 index 的项目的位置信息。比如

```
int i,j, k ,h;          getItemPosition(1, &i,&j,&k,&h);
```

假设 i=0, j=1, k=1, h=3, 则表示索引为 1 的项目位于第 0 行第 1 列，占据 1 行 3 列。

⑩、`QLayoutItem* itemAtPosition(int row, int column) const;`

返回(row,column)处的布局中项目。若单元格为空，则返回 0。

⑪、`virtual int count() const;` //虚函数，`QLayout::count()`的重新实现，返回布局中的项目数

⑫、`virtual QLayoutItem takeAt(int index);` //虚函数，

`QLayout::takeAt()`的重新实现。从布局中删除索引 index 处的项目，并返回该项目

⑬、`virtual QLayoutItem* itemAt(int index) const;` //虚函数，`QLayout::itemAt()`的重新实现

返回指定索引 index 处的项目，若没有这样的项目，则返回 0。

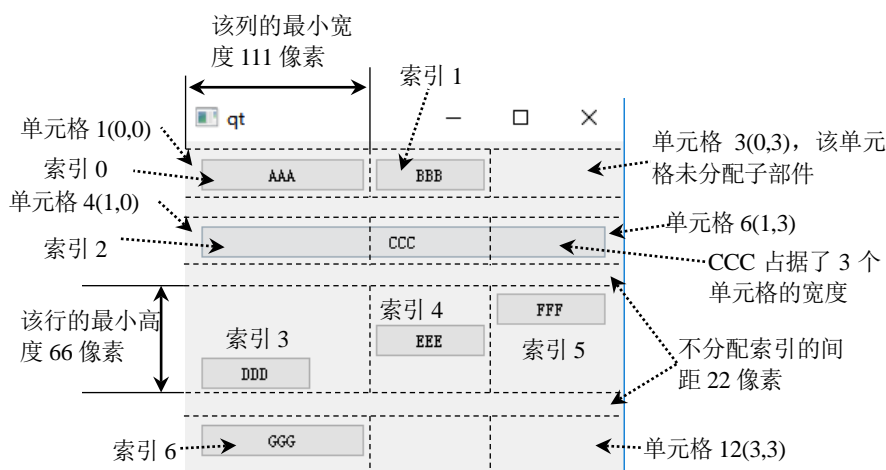
示例 5.10: QGridLayout (网格布局) 的使用

```
#include<QtWidgets>
#include <iostream>
using namespace std;
int main(int argc, char *argv[]) {    QApplication a(argc, argv);
    QWidget w;          QPushButton *pb, *pb1, *pb2, *pb3, *pb4, *pb5, *pb6;
    pb=new QPushButton("AAA");    pb1=new QPushButton("BBB");    pb2=new QPushButton("CCC");
    pb3=new QPushButton("DDD");    pb4=new QPushButton("EEE");
    pb5=new QPushButton("FFF");    pb6=new QPushButton("GGG");

    QGridLayout *pg=new QGridLayout;
    pg->setVerticalSpacing(22);    //设置各子部件之间的垂直间距，不会为该间距分配索引。
    pg->addWidget(pb, 0, 0, 1, 1);    //把按钮 pb 添加到第 0 行 0 列，该按钮占据 1 行和 1 列单元格。
    pg->addWidget(pb1, 0, 1, 1, 1);
    pg->addWidget(pb2, 1, 0, 1, 3);    //该按钮占据 3 列单元格的宽度
    pg->addWidget(pb3, 2, 0, 1, 1, Qt::AlignBottom|Qt::AlignLeft);    //pb3 在其单元格内左下对齐
    pg->addWidget(pb4, 2, 1, 1, 1);
    pg->addWidget(pb5, 2, 2, 1, 1, Qt::AlignTop|Qt::AlignRight);
    pg->addWidget(pb6, 3, 0, 1, 1);
    w.setLayout(pg);
    pg->setRowMinimumHeight(2, 66);    //把第 2 行的最小高度设置为 66 像素
    pg->setColumnMinimumWidth(0, 111);    //把第 0 列的最小宽度设置为 111 像素
    cout<<pg->columnCount()<<endl;    //输出列数，本例为 3
    cout<<pg->rowCount()<<endl;    //输出行数，本例为 4。
    cout<<pg->rowMinimumHeight(2)<<endl;    //输出第 2 行的最小高度 66
    cout<<pg->columnMinimumWidth(0)<<endl;    //输出第 0 列的最小宽度 111
    int i, j, k, l;
    pg->getItemPosition(2, &i, &j, &k, &l);    //获取索引为 2(本例为 pb3)的子部件的位置信息。
    cout<<i<<","<<j<<","<<k<<","<<l<<endl;    /*输出 (1, 0, 1, 3)，即该部件位于第 1 行第 0 列，占据 1 行
                                                3 列的单元格大小。*/

    w.show();    return a.exec(); }
```

运行结果及说明



其他说明：虚线框实际程序是没有的，索引是分配给布局中的子部件的。

若为单元格设置了拉伸因子，单元格的大小还会按拉伸因子的比例调整单元格的大小，各子部件只会在其分配的单元格内部进行拉伸和压缩。

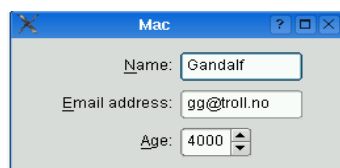
为子部件设置了对齐方式后，子部件只会在其自身所在的单元格内进行对齐，如按钮 DDD 和 FFF。按钮在垂直方向默认是不拉伸的，若要使按钮能拉伸需设置其大小策略。

三、QFormLayout 类(表单布局)

- 1、QFormLayout 布局以两列的形式列出其子项目，
- 2、QFormLayout 被分成两列，左列是标签(label)部分，通常由标签 QLabel 组成，右列是由字段(field)部分，通常是 QLineEdit 行编辑器，QSpinBox 旋转框等部件，注意：QFormLayout 的两列也可以是任意其他的部件。
- 3、使用 QFormLayout 可方便的管理“标签----字段”对形式布局的组件，使用该布局可快速的为 QLabel 设置其伙伴部件。其方法是使用成员函数 `addRow()` 如下所示
`formLayout->addRow("&Name", lineEdit);` //添加一个名称为 Name 的标签，其伙伴为 lineEdit
- 4、通常，QFormLayout 布局是使用 QGridLayout 实现的。
- 5、QFormLayout 根据不同的平台，设置了不同的默认外观，见下图



早期版本的 windows、KDE 外观。
标签左对齐，字段拉伸填充可用空间。



macOS 风格，标签右对齐，字段大小为大小提示，且表单是水平居中的。



KDE 程序的推荐样式。标签右对齐，表单左对齐，且字段拉伸填充可用空间。



Qt 的默认样式，标签右对齐，字段拉伸以填充可用空间，且对长行启用换行

6、QFormLayout 类中的枚举

QFormLayout::ItemRole 枚举(无标志)

作用：描述表单布局中部件的类型

成员	值	说明
QFormLayout::LabelRole	0	标签部件
QFormLayout::FieldRole	1	字段部件
QFormLayout::SpanningRole	2	跨越标签和字段的部件(标签和字段间的间距?)

7、QFormLayout 类中的属性

①、fieldGrowthPolicy: FieldGrowthPolicy

访问函数：FieldGrowthPolicy fieldGrowthPolicy() const; void setFieldGrowthPolicy(FieldGrowthPolicy);
 设置或获取表单布局字段部分的增长方式(注意：标签部分是不会增长的，即使标签部分部件的大小策略被设置为 Expanding，该部件也不会增长)。默认值取决于样式。
 QFormLayout::FieldGrowthPolicy 枚举见下表

QFormLayout::FieldGrowthPolicy 枚举(无标志)

作用：描述表单布局字段的增长方式(即增长策略)

成员	值	说明
QFormLayout::FieldsStayAtSizeHint	0	字段不超出他们的大小提示，QMacStyle 默认值
QFormLayout::ExpandingFieldsGrow	1	只有水平方向的大小策略为 Expanding 或 MinimumExpanding 的字段可以增长以填充可用的空间，其他字段不会超出他们的大小提示(原理见图示)。这是 QCommonStyle(比如 windows)的默认值
QFormLayout::AllNonFixedFieldsGrow	2	所有具有允许增长的大小策略的字段都会增长以填充可用空间，这是大多数样式的默认值(比如 Qt Extended 样式)。

②、formAlignment: Qt::Alignment

访问函数：Qt::Alignment formAlignment() const; void setFormAlignment(Qt::Alignment);
 设置或获取表单布局内容的对齐方式，即“标签，字段”对的对齐方式，默认值取决于样式。对于 QMacStyle，默认值为 Qt::AlignHCenter | Qt::AlignTop。对于其他样式，默认值为 Qt::AlignLeft | Qt::AlignTop。注意：该对齐方式需要在字段部分不增长的情况

下才会显示出来，否则字段会增长以填充可用空间，不会看到对齐的效果(具体见后面的示例)。

③、**labelAlignment**: Qt::Alignment

访问函数: Qt::Alignment labelAlignment() const; void setLabelAlignment(Qt::Alignment)

设置和获取标签部分的水平对齐方式，默认值取决于样式，对于 QCommonStyle，除了 QPultiqueStyle 外，默认值为 Qt::AlignLeft，对于其他样式，默认为 Qt::AlignRight。

④、**rowWrapPolicy**: RowWrapPolicy

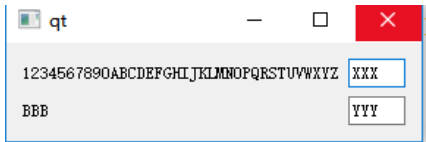
访问函数: RowWrapPolicy rowWrapPolicy() const; void setRowWrapPolicy(RowWrapPolicy)

设置和获取表单行的换行方式，默认值取决于样式，对于 QtExtended，默认为 WrapLongRows，对于其他样式，默认为 DontWrapRow。

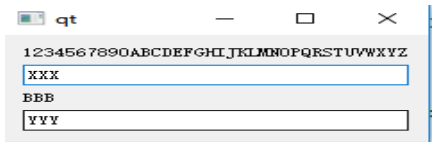
QFormLayout::RowWrapPolicy 枚举(无标志)

作用：描述表单行的换行方式

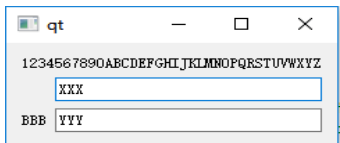
成员	值	说明
QFormLayout::DontWrapRows	0	字段始终位于标签旁边，这是除 Qt Extended 样式外的所有样式的默认策略。
QFormLayout::WrapLongRows	1	标签获得足够宽的水平空间，以适应最宽的标签，其余空间分配给字段。若“标签，字段”对的最小大小比可用空间大，则字段被换到下一行，这是 QtExtended 样式的默认策略。
QFormLayout::WrapAllRows	2	字段始终位于标签下方



DontWrapRows 策略，字段始终位于标签的旁边



WrapAllRows 策略，字段始终位于标签下方



WrapLongRows 策略，注意：在调整窗口大小时，需为使窗口有足够的垂直高度，否则字段部会不会自动切换到一下行。

⑤、**horizontalSpacing**: int **访问函数**: int horizontalSpacing() const; void setHorizontalSpacing(int);

verticalSpacing: int **访问函数**: int verticalSpacing() const; void setVerticalSpacing(int);

这两个属性用于设置和获取部件之间的水平和垂直间距(不分配索引号)。

⑥、QFormLayout 属性基本都依赖于样式，因此在使用 QFormLayout 布局时，最好是显示设置其属性。

示例 5.11：理解 QFormLayout::ExpandingFieldsGrow 增长策略

```
#include<QtWidgets>
#include<QDebug>
int main(int argc, char *argv[]) {    QApplication a(argc, argv);
    QWidget w;        QPushButton *pb,*pbl;    QLineEdit *pe,*pel;    QLabel *pt,*ptl;
```

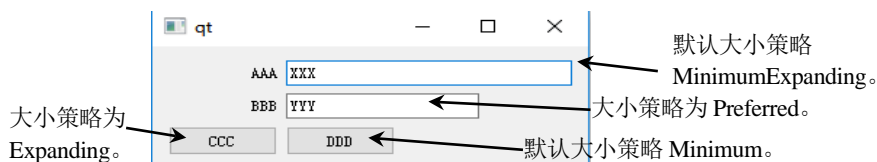
```

pb=new QPushButton("CCC");      pbl=new QPushButton("DDD");
pe=new QLineEdit("XXX");        pel=new QLineEdit("YYY");
pt=new QLabel("AAA");           ptl=new QLabel("BBB");
QFormLayout *pg=new QFormLayout;
pb->setSizePolicy(QSizePolicy::Expanding,QSizePolicy::Preferred); //设置 pb 的大小策略
pel->setSizePolicy(QSizePolicy::Preferred,QSizePolicy::Fixed); //设置 pel 的大小策略
//向布局中添加部件
pg->addRow(pt, pe);      pg->addRow(ptl, pel);      pg->addRow(pb, pbl);
pg->setLabelAlignment(Qt::AlignRight); //设置标签的对齐方式为右对齐
pg->setFieldGrowthPolicy(QFormLayout::ExpandingFieldsGrow); //设置增长策略。

QSizePolicy p=pbl->sizePolicy(); /*验证按钮的默认大小策略为 Minimum，注意 QLineEdit 在水
                                平方向的默认大小策略是 MinimumExpanding，读者可自行验证。*/
qDebug()<<p.horizontalPolicy()<<endl; //输出 Minimum
w.setLayout(pg);      w.resize(300,55);      w.show();      return a.exec(); }

```

运行结果及说明



- 1、由图可见，即使编辑器 YYY 和按钮 DDD 的大小策略是允许增长的，但他们仍不会增长。只有水平方向的大小策略为 Expanding 或 MinimumExpanding 的字段才可以增长
- 2、在布局的标签部分，即使部件的大小策略为 Expanding，也是不能增长的。
- 3、表单布局的两列，不一定必须是“标签”和行编辑器，也可以是任意的 QWidget 组件。

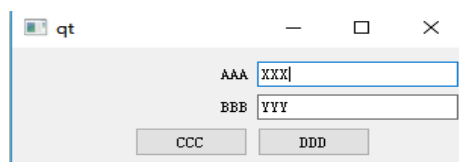
示例 5.12: 理解 formAlignment 属性

```

#include<QtWidgets>
int main(int argc, char *argv[]) {    QApplication a(argc, argv);
    QWidget w;      QPushButton *pb,*pbl;    QLineEdit *pe,*pel;    QLabel *pt,*ptl;
    pb=new QPushButton("CCC");          pbl=new QPushButton("DDD");
    pe=new QLineEdit("XXX");            pel=new QLineEdit("YYY");
    pt=new QLabel("AAA");                ptl=new QLabel("BBB");
    QFormLayout *pg=new QFormLayout;
    pg->addRow(pt, pe);      pg->addRow(ptl, pel);      pg->addRow(pb, pbl);
    pg->setLabelAlignment(Qt::AlignRight); //设置标签部分的对齐方式
    pg->setFormAlignment(Qt::AlignRight); //设置表单布局内容的对齐方式
    pg->setFieldGrowthPolicy(QFormLayout::FieldsStayAtSizeHint); //增长策略为不超出大小提示
    w.setLayout(pg);      w.resize(300,55);      w.show();      return a.exec(); }

```

运行结果及说明



8、QFormLayout 类中的函数

注意：QFormLayout 类没有设置拉伸因子的函数。

- ①、`QFormLayout(QWidget* parent = Q_NULLPTR);` //构造函数
②、`void addRow(QWidget* label, QWidget* field);`
`void addRow(QWidget* label, QLayout* field);` //把 label 和 field 添加到末尾

- ③、`void addRow(const QString& labelText, QWidget* field);`
使用 labelText 作为文本创建一个 QLabel，并把其伙伴设置为 field

- ④、`void addRow(const QString& labelText, QLayout* field);`
使用 labelText 作为文本创建一个 QLabel，并把该标签和 field 添加到末尾。

- ⑤、`void addRow(QWidget* widget);` //在末尾添加 widget，该部件占两列的宽度
`void addRow(QLayout* layout);` //在末尾添加 layout，该部件占两列的宽度

- ⑥、`void insertRow(int row, QWidget* label, QWidget* field);`
`void insertRow(int row, QWidget* label, QLayout* field);`
`void insertRow(int row, const QString &labelText, QWidget* field);`
`void insertRow(int row, const QString &labelText, QLayout* field);`
`void insertRow(int row, QWidget* widget);`
`void insertRow(int row, QLayout* layout);`

以上函数为相应的插入函数，其参数与 addRow 相同，其中若行 row 超出范围，则表示把项目添加到末尾

- ⑦、`void setItem(int row, ItemRole role, QLayoutItem* item);` //该函数添加的项目 item 会占据两列
`void setLayout(int row, ItemRole role, QLayout* layout);`
`void setWidget(int row, ItemRole role, QWidget* widget);`

以上函数表示，把行 row 和角色 role 设置为项目 item、layout 或 widget，若有必要，使用空行扩展布局。若单元格已被占用，则不插入项目，并向控制台发送错误报告，这意味着以上函数不能修改表格布局的原有内容。通常应使用 addRow()或 insertRow()来代替上述函数。

- ⑧、`void removeRow(int row);` //qt5.8
`void removeRow(QWidget* widget);` //qt5.8
`void removeRow(QLayout* layout);` //qt5.8
`TakeRowResult takeRow(int row);` //qt5.8
`TakeRowResult takeRow(QWidget* widget);` //qt5.8
`TakeRowResult takeRow(QLayout* layout);` //qt5.8

- 以上函数表示，从表单布局中删除(或移除)行 row 或与 widget 和 layout 相对应的行，其中 takeRow 函数不会删除任何内容，而 removeRow 会把其内容删除。
- row 必须是非负的，且小于 rowCount()。
- 调用以上函数后，rowCount()会减 1，占用此行的字段部分和标签部分的部件都会被删除，随后的所有行向上移一行，释放的空间在其余行中重新分配，使用这些函数可以撤销使用 addRow()和 insertRow()添加的项目。
- TakeRowResult 是 qt5.8 引进的新类，该类仅有两个成员变量，用于保存与

QFormLayout 布局相对应的两个部分的项目。TakeRowResult 类的成员如下

QLayoutItem* fieldItem; QLayoutItem* labelItem;

- ⑨、QWidget* **labelForField**(QWidget* field) const;
QWidget* **labelForField**(QLayout* field) const; //返回与字段 field 关联的标签
- ⑩、void **getItemPosition**(int index, int *rowPtr, ItemRole* rolePtr) const;
void **getLayoutPosition**(QLayout* layout, int *rowPtr, ItemRole* rolePtr) const;
void **getWidgetPosition**(QWidget* widget, int *rowPtr, ItemRole* rolePtr) const;
返回指定索引 index、部件 widget 或布局 layout 处的行和角色,并将索引存储在*rowPtr 中,把角色存储在*rolePtr 中,若索引超出界限或 widget 或 layout 不存在,则*rowPtr 为-1。
- ⑪、int **rowCount**() const; //返回表单布局中的行数
virtual int **count**() const; //虚函数, QLayout::count()的重新实现, 返回布局中的项目数
- ⑫、virtual void **addItem**(QLayoutItem* item); //虚函数, QLayout::addItem()的重新实现
这是向 QFormLayout 布局添加 QSpacerItem 的唯一方法。
- ⑬、QLayoutItem* **itemAt**(int row, ItemRole role) const;
virtual QLayoutItem* **itemAt**(int index) const; //虚函数, QLayout::itemAt()的重新实现
返回指定索引 index 或指定行 row, 指定角色 role 处的项目, 若没有这样的项目, 则返回 0。
- ⑭、virtual QLayoutItem **takeAt**(int index); //虚函数, QLayout::takeAt()的重新实现。
从布局中删除索引 index 处的项目, 并返回该项目
- ⑮、void **setSpacing**(int spacing); //把垂直和水平间距都设置为 spacing
int **spacing**() const; //若垂直和水平间距相等, 则返回该值, 否则返回-1。
以上两函数设置的是 spcing 属性, 它不是一个 QSpacerItem。

示例 5.13: QFormLayout 布局的使用

//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>
#include<QDebug>
#include <iostream>
using namespace std;
class B:public QWidget{    Q_OBJECT
public:    QPushButton *pb,*pb1,*pb2,*pb3,*pb4;
    QLineEdit *pe,*pe1;    QLabel *pt;    QFormLayout *pg;    QHBoxLayout *ph;
    B(){    pb=new QPushButton("CCC");        pb1=new QPushButton("DDD");
        pb2=new QPushButton("remove");        pb3=new QPushButton("take");
        pb4=new QPushButton("show");        pe=new QLineEdit("XXX");
        pe1=new QLineEdit("YYY");        pt=new QLabel("AAA");
        ph=new QHBoxLayout;        pg=new QFormLayout;
        QSpacerItem *pi=new QSpacerItem(22, 23); //创建一个空白间距(QSpacerItem)
        ph->addWidget(pb2);    ph->addWidget(pb3);    ph->addWidget(pb4);
        pg->addRow(ph);        //把水平布局 ph 添加到表格布局中
        pg->addItem(pi);        //把空白间距添加到表格布局中。
    }
};
```

```

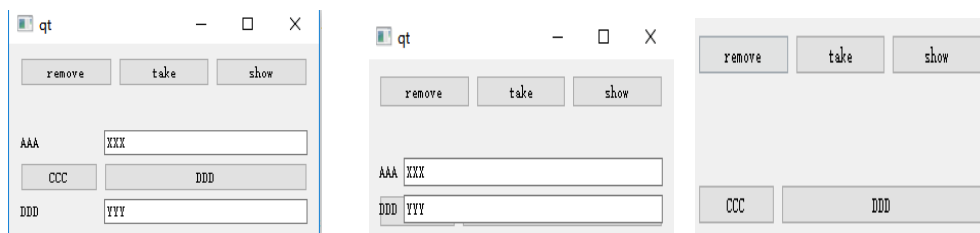
    pg->addRow(pt, pe);
    pg->addRow(pb, pb1);
    pg->addRow("&DDD", pe1); //使用文本 DDD 创建一个标签，并使 pe1 自动成为该标签的伙伴。
    setLayout(pg);
    QObject::connect(pb2, &QPushButton::clicked, this, &B::f);
    QObject::connect(pb3, &QPushButton::clicked, this, &B::f1);
    QObject::connect(pb4, &QPushButton::clicked, this, &B::f2);    }    //构造函数结束

public slots:
    void f() {    pg->removeRow(1);    }    //从表格布局 pg 中删除索引为 1 的行(会删除内容)。
    void f1() {    pg->takeRow(pb);    }    //从表格布局 pg 中移除 pb 所对应的行(不会删除内容)。
    void f2() {    cout<<pg->rowCount()<<endl;    cout<<pg->count()<<endl;    }    };
#endif // M_H

//m.cpp 文件的内容
#include "m.h"
int main(int argc, char *argv[]) {    QApplication a(argc, argv);
    B w;    w.resize(300, 55);    w.show();    return a.exec();    }

```

运行结果及说明



按以下步骤测试

首先按下 **show**，输出 5 和 8，表示该布局有 5 行 8 个项目(注，有一个空白间距)。

接着按下 **take** 按钮，效果见右中间的图，可见 **takeRow()** 函数只是把部件从表格布局中移除了，但并未删除布局中的部件。

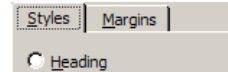
再按下 **show** 按钮，输出 4 和 6，表示该布局有 4 行 6 个项目。

接着连续按下 **remove** 按钮，可把表格布局中剩下的内容(除第一行外)全部删除掉，其效果见最右图。

5.3 实现多页面切换

QStackedLayout、QStackedWidget、QTabBar、QTabWidget

一、QStackedLayout 类(分组布局或栈布局)



- 1、使用 QStackedLayout 可以实现一个多页面切换的界面，多页面切换就是类似于选项卡(如右图)类型的界面。
- 2、QStackedLayout 并没有直接实现多页面切换的办法，只是我们可以通过该类实现多页面切换的功能，因此要使用 QStackedLayout 类实现多页面切换，需要自行进行设计。
- 3、QStackedLayout 布局的基本原理是，在同一时间只显示其中的一个子部件，当前显示的子部件被称为**当前部件(可见部件)**，也就是说若子部件是当前部件，它就是可见的。
- 4、实现多页面切换的基本步骤
 - ①、使用 QComboBox(组框)或 QListWidget(列表)之类的部件，提供给用户一个选择需要显示的页面的选项。
 - ②、创建一个容器(比如 QWidget 实例)，然后在该容器中添加需要显示给用户的内容。
 - ③、把容器添加到 QStackedLayout 布局中。
 - ④、把 QComboBox 的 activated 信号(或类似信号)与 QStackedLayout 的 setCurrentIndex 槽相关联，以实现当用户选择 QComboBox 中的选项时，显示 QStackedLayout 布局中的子部件。setCurrentIndex 槽的作用就是使 QStackedLayout 布局中的某一个子部件可见。
 - ⑤、下面为实现多页面切换的具体代码

```
QWidget w;           //主窗口
QComboBox *pc= new QComboBox; pc->addItem("page1"); pc->addItem("page2");
QWidget* pw=new QWidget(&w); //为当前显示的容器指定父部件，这样可避免初次运行程序时的
                           闪烁问题。
QWidget* pw1=new QWidget; //容器。
..... //向容器中添加需要显示的内容，略。
QStackedLayout *ps=new QStackedLayout; //创建分组布局
ps->addWidget(pw); ps->addWidget(pw1); //把容器添加到分组布局中。
QVBoxLayout *pv=new QVBoxLayout; //垂直布局
pv->addWidget(pc); pv->addLayout(pv); //把组框 pc 和分组布局 pv 添加到垂直布局中。
connect( pc, SIGNAL(activated(int)), ps, SLOT(setCurrentIndex(int))); //关联信号和槽。
w.setLayout(pv); //为主窗口添加布局
```

5、QStackedLayout 类中的属性

- ①、**count:** const int **访问函数:** virtual int count() const;
QLayout::count()的重新实现，返回布局中的项目数
- ②、**currentIndex:** int **访问函数:** int currentIndex() const; void setCurrentIndex(int index) //槽
信号: currentChanged(int index);
获取和设置当前部件(即可见部件)，若没有当前部件，则索引为-1
- ③、**stackingMode:** StackingMode
访问函数: StackingMode stackingMode() const; void setStackingMode(StackingMode);

获取和设置子部件的显示方式，默认为 StackOne。StackingMode 枚举取值如下：

QStackedLayout::StackOne(值为 0)：只有当前部件可见(默认值)

QStackedLayout::StackAll(值为 1)：所有部件均可见，当前部件只是被提升(即当前部件被提升到其他部件的前面)

6、QStackedLayout 类中的函数

①、**QStackedLayout()**;

QStackedLayout(QWidget* parent);

QStackedLayout(QLayout* parentLayout);

②、**int addWidget(QWidget* widget);**

int insertWidget(int index, QWidget* widget);

把部件 widget 添加到布局的末尾或插入到索引 index 处，并返回部件的索引号，若在调用此函数之前布局为空，则部件 widget 成为当前部件。若索引 index 超出范围，则添加到末尾。在小于或等于当前索引的索引处插入新部件将增加当前部件的索引，但当前部件仍是该部件，也就是说不会改变正在显示的部件。

③、**QWidget* currentWidget() const;** //返回当前部件，若布局中没有部件，则返回 0

④、**void setCurrentWidget(QWidget* widget);** //槽

把当前部件设置为 widget。新的当前部件必须包含在布局中。

⑤、**QWidget* widget(int index) const;** //返回索引 index 处的部件，若给定位置没有部件，则返回 0。

⑥、**virtual void addItem(QLayoutItem* item);** //虚函数，QLayout::addItem()的重新实现

把 item 添加到布局中。

⑦、**virtual QLayoutItem* itemAt(int index) const;** //虚函数，QLayout::itemAt()的重新实现

返回指定索引 index 处的项目，若没有这样的项目，则返回 0。

⑧、**virtual QLayoutItem takeAt(int index);** //虚函数，QLayout::takeAt()的重新实现。

从布局中删除索引 index 处的项目，并返回该项目

⑨、**void widgetRemoved(int index);** //信号

当部件从布局中移除时发送此信号。

⑩、**void currentChanged(int index);** //信号

当布局中的当前部件更改时，发送此信号。索引为更改后的新的当前部件的索引，若没有新部件(比如布局中没有部件)，则为-1。

7、QStackedLayout 类的核心是 setCurrentIndex(int)和 setCurrentWidget(QWidget*)槽函数，要删除布局中的部件，可使用 QLayout::removeWidget()函数，要获取布局中子部件的索引需使用 QLayout::indexOf()函数。

示例 5.14：使用 QStackedLayout 布局实现简单的多页面切换

```
#include<QtWidgets>
```

```
int main(int argc, char *argv[]) {    QApplication a(argc, argv);
```

```
    QWidget w;
```

```
    QWidget *pw=new QWidget(&w);    //初次显示时的当前部件需指定父部件以避免闪烁。
```

```
    QWidget *pw1=new QWidget;
```

```
    QPushButton *pb,*pb1,*pb2;    //容器 pw 中包含的三个按钮
```

```
    QRadioButton *pr,*pr1,*pr2;    //容器 pw1 中包含的三个单选按钮
```

```
    QStackedLayout *ps=new QStackedLayout;    QVBoxLayout *pv=new QVBoxLayout;
```



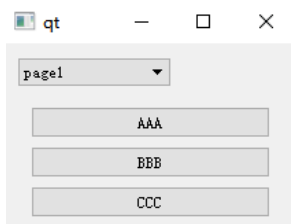
```

    QVBoxLayout *pv1=new QVBoxLayout;          QVBoxLayout *pv2=new QVBoxLayout;
    QComboBox *pc=new QComboBox;    //组框，用于选择分页
//初始化各部件
    pb=new QPushButton("AAA");    pb1=new QPushButton("BBB");    pb2=new QPushButton("CCC");
    pr=new QRadioButton("DDD");    pr1=new QRadioButton("EEE");    pr2=new QRadioButton("FFF");
    pc->addItem("page1");    pc->addItem("page2");    pc->setMinimumWidth(111);
    pc->setSizePolicy(QSizePolicy::Fixed,QSizePolicy::Fixed);
//设置容器 pw 的内容。
    pv1->addWidget(pb);    pv1->addWidget(pb1);    pv1->addWidget(pb2);    pw->setLayout(pv1);
//设置容器 pw1 的内容。
    pv2->addWidget(pr);    pv2->addWidget(pr1);    pv2->addWidget(pr2);    pw1->setLayout(pv2);
    ps->addWidget(pw);    ps->addWidget(pw1);    /*把容器 pw 和 pw1 添加到分组布局中，这样容器和
    组框就可以通过信号和槽产生联系了。*/

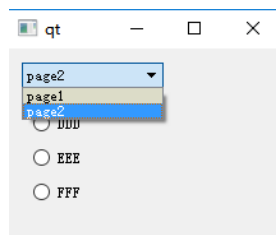
    pv->addWidget(pc);    pv->addLayout(ps);
    w.setLayout(pv);
//关联信号和槽以实现多页面切换(这是关键步骤)。
    QObject::connect(pc, SIGNAL(activated(int)), ps, SLOT(setCurrentIndex(int)));
    w.resize(300,155);    w.show();    return a.exec();    }

```

运行结果及说明



在组框中选择"page1"显示的界面

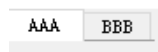


在组框中选择"page2"显示的界面

二、QStackedWidget 类

QStackedWidget 类是在 QStackedLayout 之上构造的一个便利的部件，其使用方法与步骤和 QStackedLayout 是一样的。QStackedWidget 类的成员函数与 QStackedLayout 类也基本上是一致的，使用该类就和使用 QStackedLayout 一样，因此该类就不重复讲述了。

三、QTabBar 类(选项卡栏)



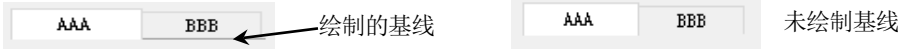
- 1、QTabBar 类直接继承自 QWidget。该类提供了一个选项卡栏，该类仅提供了一个选项卡，并没有为每个选项卡提供相应的页面，因此要使选项卡栏实际可用，需要自行行为每个选项卡设置需要显示的页面，可以通过 QStackedLayout 布局为选项卡提供页面，另外 Qt 也提供了一个现成的选项卡部件 QTabWidget。
- 2、创建选项卡栏的步骤，如下：

```
QTabBar *pt = new QTabBar;    pt->addTab("AAA");    pt->Tab("BBB");
```

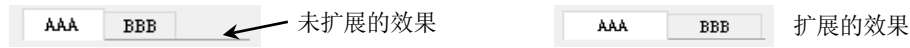
3、QTabBar 类的属性

QTabBar 类属性速查表			
属性名	说明	属性名	说明
expanding	是否扩展选项卡	documentMode	是否以适合主窗口的模式呈现
movable	选项卡是否可移动	currentIndex	获取和设置当前选项卡
shape	描述选项卡的形状	elideMode	选项卡中文本的省略方式
count	获取选项卡的数量	autoHide	选项卡仅 1 个时是否隐藏
drawBase	是否绘制基线。	usesScrollButtons	是否使用滚动按钮
tabsClosable	是否显示选项卡上的关闭按钮	selectionBehaviorOnRemove	当删除当前选项卡时，使用哪个选项卡作为当前选项卡
iconSize	选项卡栏中的图标大小	changeCurrentOnDrag	拖动选项卡时，当前选项卡是否会自动更改。

- ①、**autoHide**: bool 访问函数: bool autoHide() const; void setAutoHide(bool); //qt5.4
若该属性为 true，则当选项卡少于 2 个(即只有 1 个)时会自动隐藏，默认为 false。
- ②、**count**: const int 访问函数: int count() const;
获取选项卡栏中的选项卡数量。
- ③、**currentIndex**: int 访问函数: int currentIndex() const; void setCurrentIndex(int); //槽
信号: void currentChanged(int index);
获取和设置选项卡栏上当前选项卡(即可见选项卡)的索引，若没有当前选项卡，则索引为-1。此属性的槽函数 setCurrentIndex 是个重要的函数
- ④、**drawBase**: bool 访问函数: bool drawBase() const; void setDrawBase(bool);
此属性描述是否绘制选项卡的基线，若为 true，则绘制基线(效果见下图)



- ⑤、**elideMode**: Qt::TextElideMode
访问函数: Qt::TextElideMode elideMode() const; void setElideMode(Qt::TextElideMode);
此属性描述如何省略选项卡中的文本(如右图)。Qt::TextElideMode 枚举见公用枚举章节
- ⑥、**expanding**: bool 访问函数: bool expanding() const; void setExpanding(bool);
此属性描述是否扩展选项卡，默认为 true。效果见下图

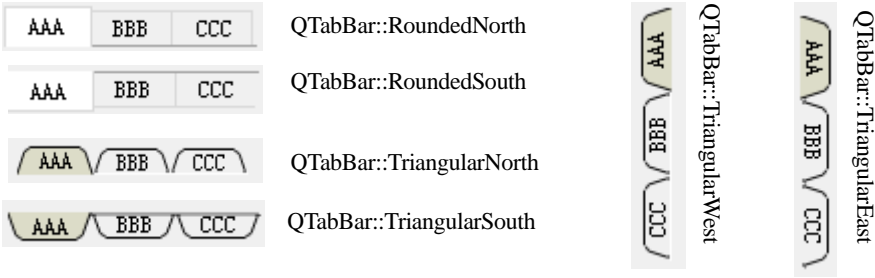


- ⑦、**iconSize**: QSize 访问函数: QSize iconSize() const; void setIconSize(const QSize&);
此属性描述选项卡栏中保存的图标大小。
- ⑧、**movable**: bool 访问函数: bool movable() const; void setMovable(bool);
此属性描述用户是否可以移动选项卡区域内的选项卡。默认为 false。
- ⑨、**selectionBehaviorOnRemove**: SelectionBehavior
访问函数: SelectionBehavior selectionBehaviorOnRemove() const;
void setSelectionBehaviorOnRemove(SelectionBehavior);
当使用 removeTab()删除的选项卡是当前选项卡时，应将哪个选项卡设置为当前选项卡。默认为 SelectRightTab。SelectionBehavior 枚举见下表

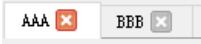
QTabBar::SelectionBehavior 枚举(无标志)		
作用：指定当删除当前选项卡时，使用哪个选项卡作为当前选项卡		
成员	值	说明
QTabBar::SelectLeftTab	0	使用左侧的选项卡作为当前选项卡
QTabBar::SelectRightTab	1	使用左侧的选项卡作为当前选项卡(默认值)
QTabBar::SelectPreviousTab	2	使用先前选择的选项卡作为当前选项卡

- ⑩、
shape: Shape
访问函数：Shpae shpae() const; void setShape(Shape);
此属性描述选项卡栏中选项卡的形状。形状由 Shape 枚举描述，见下表及图

QTabBar::Shape 枚举(无标志)		
作用：描述选项卡的形状，以下形状只是一个建议，某些样式可能不支持这些形状。		
成员	值	说明
QTabBar::RoundedNorth	0	正常的圆角外形位于上方。
QTabBar::RoundedSouth		正常的圆角外形位于下方。
QTabBar::RoundedWest		正常的圆角外形位于左侧。
QTabBar::RoundedEast		正常的圆角外形位于右侧。
QTabBar::TriangularNorth		三角形外形位于上方
QTabBar::TriangularSouth		三角形外形位于下方
QTabBar::TriangularWest	1	三角形外形位于左侧
QTabBar::TriangularEast	2	三角形外形位于右侧



- ⑪、
tabsClosable: bool
访问函数：bool tabsClosable() const; void setTabsClosable(bool);
此属性描述是否应在每个选项卡上放置关闭按钮。若该属性为 true，则关闭按钮出现在左侧或右侧依样式而定。当点关闭按钮被点击时发送 tabCloseRequested 信号。默认为 false。



- ⑫、
usesScrollButtons: bool
访问函数：bool usesScrollButtons() const; void setUsesScrollButtons(bool);
此属性描述，当选项卡太多时，是否使用滚动按钮，默认值取决于样式。滚动按钮的效果见右图



- ⑬、
documentMode: bool
访问函数：bool documentMode() const; void setDocumentMode(bool);

此属性描述选项卡栏是否以适合主窗口的模式呈现。此属性用作样式的提示，以便以不同的方式绘制选项卡。

- ⑭、`changeCurrentOnDrag`: bool //qt5.4

访问函数: bool changeCurrentOnDrag() const; void setChangeCurrentOnDrag(bool);

若为 true，则当拖动选项卡时，当前选项卡会自动更改，默认为 false。注意：要使用此属性需把 QWidget::acceptDrops 属性设置为 true。

4、QTabBar 类中的函数

- ①、`QTabBar`(QWidget *parent = Q_NULLPTR); //构造函数

- ②、int `addTab`(const QString &text);
int `addTab`(const QIcon& icon, const QString &text);
int `insertTab`(int index, const QString &text);
int `insertTab`(int index, const QIcon& icon, const QString &text);

以上函数用于在末尾添加或在索引 index 处插入带有图标 icon 或文本 text 的新选项卡。若 index 超出范围则，则选项卡追加到末尾，若在小于或等于当前选项卡索引的索引处插入一个新选项卡，将使当前选项卡的索引号增加，但当前选项卡保持不变。以上函数返回的是新选项卡的索引。

- ③、void `moveTab`(int from, int to); //把选项卡从索引 from 移动到索引 to。

- ④、void `removeTab`(int index); //删除索引 index 处的选项卡。

- ⑤、QRect `tabRect`(int index) const;

返回索引 index 处的选项卡的矩形尺寸。若索引超出范围，则返回空矩形。

- ⑥、int `tabAt`(const QPoint& position) const;

返回位置 position 处的选项卡的索引。点(0,0)为第一个选项卡的坐标位置。

- ⑦、void `setTabEnabled`(int index, bool enabled); //设置选项卡的启用/禁用状态，禁用状态呈现出灰色。

bool `isTabEnabled`(int index) const; //获取选项卡的启用/禁用状态

void `setTabIcon`(int index, const QIcon& icon); //设置选项卡的图标

QIcon `tabIcon`(int index) const; //获取选项卡的图标。

void `setTabText`(int index, const QString &text); //设置选项卡的文本

QString `tabText`(int index) const; //获取选项卡的文本。

void `setTabTextColor`(int index, const QColor& color); //设置选项卡的文本颜色

QColor `tabTextColor`(int index) const; //获取选项卡的文本颜色。

void `setTabToolTip`(int index, const QString &tip); //设置选项卡的提示文本

QString `tabToolTip`(int index) const; //获取选项卡的提示文本。

void `setTabWhatsThis`(int index, const QString &text); //设置选项卡的帮助文本

QString `tabWhatsThis`(int index) const; //获取选项卡的帮助文本。

void `setTabData`(int index, const QVariant& data); //设置选项卡的数据

QVariant `tabData`(int index) const; //获取选项卡的数据。

void `setAccessibleTabName`(int index, const QString &name); //设置选项卡的 accessibleName 名称

QString `accessibleTabName`(int index) const; //获取选项卡的 accessibleName 名称。通常不需要该名称。

- ⑧、void `setTabButton`(int index, ButtonPosition position, QWidget* widget); //在选项卡上安装部件

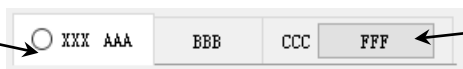
```
QWidget* tabBarButton(int index, ButtonPosition position) const; //获取选项卡上安装的部件
```

以上函数根据位置 `position` 决定把部件 `widget` 安装在选项卡上面的左侧还是右侧，其效果如下图。其中 `ButtonPosition` 枚举取值如下：

`QTabBar::LeftSide`(值为 0)：选项卡左侧。

`QTabBar::RightSide`(值为 1)：选项卡右侧。

该选项卡的文本为 AAA，而 XXX 是安装在其上的单选按钮



该选项卡的文本为 CCC，而 FFF 是安装在其上的按钮

5、QTabBar 类中的信号

- ①、`void currentChanged(int index);` //信号

当选项卡栏上的当前选项卡发生更改时发送此信号，`index` 为新选项卡的索引，若没有新的索引，则为 -1(比如 `QTabBar` 中没有选项卡)。该信号比较重要。

- ②、`void tabBarClicked(int index);` //信号，qt5.2

`void tabBarDoubleClicked(int index);` //信号，qt5.2

以上信号表示，单击或双击 `index` 处的选项卡时发送此信号，`index` 是单击选项卡的索引，若光标下没有选项卡，则为 -1。

- ③、`void tabCloseRequested(int index);` //信号

当点击选项卡上的关闭按钮时发送此信号，`index` 为应删除的选项卡的索引。

- ④、`void tabMoved(int from, int to);` //信号

当选项卡已经从索引 `from` 处移动到索引 `to` 处时，发送此信号，注意：当信号从选项卡栏发出时，`QTabWidget` 会自动移动页面。

6、当需要自定义选项卡的外观时，可能需要重新实现以下虚函数(都是受保护的)：

```
virtual void tabSizeHint(int index) const; //选项卡的大小提示。
```

```
virtual void tabRemoved(int index); //从 index 处删除选项卡后调用。
```

```
virtual void tabLayoutChange(); //只要选项卡的布局更改，就会调用此函数。
```

```
virtual void tabInserted(int index); //在 index 处添加或插入新选项卡后调用此函数。
```

```
virtual void paintEvent(QPaintEvent*); //绘制所有选项卡。
```

示例 5.15：使用 QTabBar 类和 QStackedLayout 布局实现多页界面切换

//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include <QtWidgets>
class B:public QTabBar{    Q_OBJECT
public:                    B(QWidget* p=0):QTabBar(p){ }
public slots:              void f(){ moveTab(0,2); } }; //把选项卡从索引 0 移至索引 2 处。
#endif // M_H
```

//m.cpp 文件内容

```
#include "m.h"
int main(int argc, char *argv[]){    QApplication a(argc,argv);
```

```

//创建子部件
QWidget w;          B *pt=new B;          QPushButton *pb;
QRadioButton *pr,*pr1,*pr2,*pr3,*pr4,*pr5;
pr=new QRadioButton("AAA");   pr1=new QRadioButton("BBB");   pr2=new QRadioButton("CCC");
pr3=new QRadioButton("DDD");   pr4=new QRadioButton("EEE");   pr5=new QRadioButton("FFF");
pt->addTab("AAA");   pt->addTab("BBB");   pt->addTab("CCC");
pb=new QPushButton("moveTab");

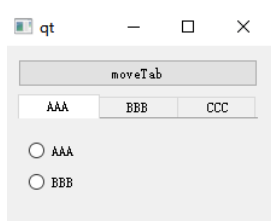
//创建容器
QWidget *pw=new QWidget(&w);   QWidget *pw1=new QWidget;   QWidget *pw2=new QWidget;

QStackedLayout *ps=new QStackedLayout;   /*三个容器 pw, pw1, pw2 位于该布局内，这样容器和
选项卡就可以通过信号和槽进行关联了。*/
ps->addWidget(pw);   ps->addWidget(pw1);   ps->addWidget(pw2);
QVBoxLayout *pv=new QVBoxLayout;   //容器 pw 使用的布局
pv->addWidget(pr);   pv->addWidget(pr1);   pw->setLayout(pv);
QVBoxLayout *pv1=new QVBoxLayout;   //容器 pw1 使用的布局
pv1->addWidget(pr2);   pv1->addWidget(pr3);   pw1->setLayout(pv1);
QVBoxLayout *pv2=new QVBoxLayout;   //容器 pw2 使用的布局
pv2->addWidget(pr4);   pv2->addWidget(pr5);   pw2->setLayout(pv2);

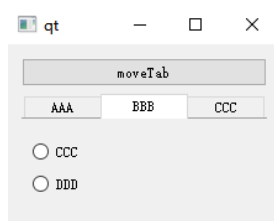
QHBoxLayout *ph=new QHBoxLayout;   ph->addWidget(pb);   //此布局仅有一个按钮
QVBoxLayout *pv3=new QVBoxLayout;   //把布局 ph, ps 和部件 pt 放于此容器内。
pv3->addLayout(ph);   pv3->addWidget(pt);   pv3->addLayout(ps);
w.setLayout(pv3);
//关联信号和槽以实现多页面切换(这是关键步骤)。
QObject::connect(pt, SIGNAL(currentChanged(int)), ps, SLOT(setCurrentIndex(int)));
QObject::connect(pb, &QPushButton::clicked, pt, &B::f);
w.resize(300, 155);   w.show();   return a.exec(); }

```

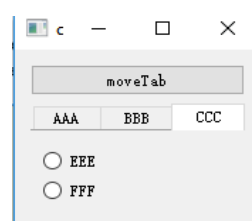
运行结果及说明



AAA 选项卡显示的界面



BBB 选项卡显示的界面



CCC 选项卡显示的界面

可见，选择不同的选项卡，呈现出了不同的界面，点击按钮 moveTab 会把第 1 个选项卡移到最后一个位置。

四、QTabWidget 类(选项卡部件)

- 1、QTabWidget 类直接继承自 QWidget。该类提供了一个选项卡栏(QTabBar)和一个相应的页面区域，用于显示与每个选项卡相对应的页面。与 QStackedLayout 布局原理相同，只有当前页面(即可见页面)是可见的，所有其他页面都不可见，用户可通过选择不同的选项卡

- 来显示其对应的其他页面。
- 2、页面或页面部件：其实就是容器(通常使用 QWidget 创建)
 - 3、QTabWidget 类，是一个实现多页面切换的类，该类已经实现了多页面切换的部分功能，只需再对其进行少量的设计(主要是要设计页面中的内容)便可实现多页面切换了。因此，使用该类实现多页面切换时，就不需要再使用 QStackedLayout 布局把页面与选项卡相关联，也不需要使用类似 QVBoxLayout 的布局把选项卡和页面放置在一起。
 - 4、QTabWidget 类的大部分功能由 QTabBar(主要处理选项卡部分)和 QStackedWidget(主要处理组织页面的功能)提供。
 - 5、使用 QTabWidget 的步骤为：
 - ①、创建一个 QTabWidget。
 - ②、为每个选项卡创建一个页面(容器)，通常为 QWidget(不要指定父部件)。
 - ③、把子部件插入到页面部件(即容器)中。
 - ④、使用 addTab()或 insertTab()把页面部件放入选项卡部件。
 - ⑤、下面为大致代码


```
QTabWidget pt;    //选项卡部件
QWidget *pw, *pw1, *pw2,...;    //创建容器。
.....          //向容器中添加需要显示的内容，略。
pt.addTab(pw,"AAA");    //把容器 pw 作为选项卡 AAA 的页面。
pt.addTab(pw1,"BBB");    //把容器 pw1 作为选项卡 BBB 的页面。
.....
```
 - ⑥、注：若容器中的内容不可见，则使用 resize()函数设置 QTabWidget 的大小使其可见
 - 6、QTabWidget 类中的属性和函数大多与 QTabBar 中的属性和函数是相同的，对于相同的属性和函数此处仅列出，只讲解不相同的属性和函数。

7、QTabWidget 类中的属性

QTabWidget 类属性

该表中的属于与 QTabBar 中的属性相同，请参阅 QTabBar 类的讲解

属性名	说明	属性名	说明
count	获取选项卡的数量	documentMode	是否以适合文档页面的模式呈现
movable	选项卡是否可移动	currentIndex	获取和设置当前选项卡
elideMode	选项卡中文本的省略方式	tabsClosable	是否显示选项卡上的关闭按钮
iconSize	选项卡栏中的图标大小	usesScrollButtons	是否使用滚动按钮

- ①、**tabBarAutoHide**: bool //qt5.4
 访问函数: bool tabBarAutoHide(); void setTabBarAutoHide(bool);
 若为 true，则当选项卡只有 1 个时，会自动隐藏，默认为 false。该属性对应于 QTabBar 的 autoHide 属性。
- ②、**tabPosition**: TabPosition
 访问函数: TabPosition tabPosition(const; void setTabPosition(TabPosition);
 获取或设置选项卡的位置(即选项卡位于上、下、左、右)。默认为 North(即上)。

TabPosition 枚举见下表

QTabWidget::TabPosition 枚举(无标志)					
作用：描述选项卡的位置					
成员	值	说明	成员	值	说明
QTabWidget::North	0	北面(即上面)	QTabWidget::West	2	西面(即左侧)
QTabWidget::South	1	南面(即下面)	QTabWidget::East	3	东面(即右侧)

- ③、**tabShape**: TabShape **访问函数**: TabShape tabShape() const; void setTabShape(TabShape);
此属性描述选项卡的形状，该属性对应于 QTabBar 的 shape 属性，其外观可参阅该类

QTabWidget::TabShape 枚举(无标志)					
作用：描述选项卡的形状					
成员	值	说明	成员	值	说明
QTabWidget::Rounded	0	圆形外观(默认)	QTabWidget::Triangular	2	三角形外观

7、QTabWidget 类中的函数

- ①、**QTabWidget**(QWidget* parent = Q_NULLPTR); //构造函数
- ②、**int addTab**(QWidget* page, const QString &label);
int addTab(QWidget* page, const QIcon &icon, const QString &label);
int insert(int index, QWidget* page, const QString &label);
int insert(int index, QWidget* page, const QIcon &icon, const QString &label);
- 以上函数表示，把页面部件 page 和具有文本 label 和(或)图标 icon 的选项卡添加到 QTabWidget 部件的末尾或插入到索引 index 处，并返回选项卡栏上该选项卡的索引。
 - 其中 label 和 icon 会成为选项卡的文本和图标。
 - 可在 label 的文本中使用&符号为选项卡设置快捷键。
 - 若在小于或等于当前选项卡索引的索引处插入一个新选项卡，将使当前选项卡的索引号增加，但当前选项卡保持不变。
 - 若以上函数在 show()之后调用，则可能会导致闪烁现象。
- ③、**void clear**();
移除所有页面，但不删除它们。调用此函数相当于调用 removeTab()函数直到选项卡部件为空。
- ④、**void removeTab**(int index); //移除索引 index 处的选项卡，页面不会被删除。
- ⑤、**int indexOf**(QWidget* w) const; //返回部件 w 的索引位置，若没有该部件则返回-1
- ⑥、**QWidget* widget**(int index) const; //返回索引 index 处的页面部件。
- ⑦、**QWidget* currentWidget**() const; //返回指向当前页面部件的指针。
void setCurrentWidget(QWidget* widget); //槽，把 widget 设置为当前页面(可见页面)。
- ⑧、**QTabBar* tabBar**() const; //返回当前的 QTabBar。
void setTabBar(QTabBar* tb); //受保护的。

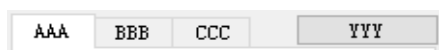
使用 `tb` 替换选项卡部件的 `QTabBar` 选项卡栏，注意：必须在添加任何其他选项卡之前调用此函数，否则将是未定义的。

- ⑨、`QWidget* cornerWidget(Qt::Corner corner = Qt::TopRightCorner) const;`

返回角落 `corner` 处的小部件或 0。

```
void setCornerWidget(QWidget* widget, Qt::Corner corner = Qt::TopRightCorner);
```

把部件 `widget` 添加到指定的角落 `corner` 处(效果见下图)。仅选项卡位于上方或下方时才能使用。



在右上角添加一个按钮的效果

- ⑩、`void setTabEnabled(int index, bool enabled);` //设置选项卡的启用/禁用状态，禁用状态呈现出灰色。
`bool isTabEnabled(int index) const;` //获取选项卡的启用/禁用状态
`void setTabIcon(int index, const QIcon& icon);` //设置选项卡的图标
`QIcon tabIcon(int index) const;` //获取选项卡的图标。
`void setTabText(int index, const QString &text);` //设置选项卡的文本
`QString tabText(int index) const;` //获取选项卡的文本。
`void setTabToolTip(int index, const QString &tip);` //设置选项卡的提示文本
`QString tabToolTip(int index) const;` //获取选项卡的提示文本。
`void setTabWhatsThis(int index, const QString &text);` //设置选项卡的帮助文本
`QString tabWhatsThis(int index) const;` //获取选项卡的帮助文本。

8、QTabWidget 类中的信号

- ①、`void currentChanged(int index);` //信号
当选项卡栏上的当前选项卡发生更改时发送此信号，`index` 为新选项卡的索引，若没有新的索引，则为-1(比如 `QTabBar` 中没有选项卡)。该信号比较重要。
- ②、`void tabBarClicked(int index);` //信号，qt5.2
`void tabBarDoubleClicked(int index);` //信号，qt5.2
以上信号表示，单击或双击 `index` 处的选项卡时发送此信号，`index` 是单击选项卡的索引，若光标下没有选项卡，则为-1。
- ③、`void tabCloseRequested(int index);` //信号
当点击选项卡上的关闭按钮时发送此信号，`index` 为应删除的选项卡的索引。

示例 5.16：使用 QTabWidget 类(选项卡部件)实现多页面切换

//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>
#include <iostream>
using namespace std;
class B:public QTabWidget{    Q_OBJECT
public:    B(QWidget* p=0):QTabWidget(p) { }
```



```

        public slots:    void f(){    removeTab(0);    } }; //移除索引为零的选项卡
#endif // M_H

//m.cpp 文件的内容
#include "m.h"
int main(int argc, char *argv[]){    QApplication a(argc, argv);
    QWidget w;
    QWidget *pw=new QWidget;    QWidget *pw1=new QWidget;    QWidget *pw2=new QWidget; //容器
    QRadioButton *pr,*pr1,*pr2,*pr3,*pr4,*pr5;
    QPushButton *pb=new QPushButton("remove",&w);
    pr=new QRadioButton("AAA");    pr1=new QRadioButton("BBB");    pr2=new QRadioButton("CCC");
    pr3=new QRadioButton("DDD");    pr4=new QRadioButton("EEE");    pr5=new QRadioButton("FFF");

    QVBoxLayout *pv=new QVBoxLayout;    //由容器 pw 使用的布局
    QVBoxLayout *pv1=new QVBoxLayout;    QVBoxLayout *pv2=new QVBoxLayout;
//把子部件添加到容器中
    pv->addWidget(pr);    pv->addWidget(pr1);    pw->setLayout(pv);
    pv1->addWidget(pr2);    pv1->addWidget(pr3);    pw1->setLayout(pv1);
    pv2->addWidget(pr4);    pv2->addWidget(pr5);    pw2->setLayout(pv2);

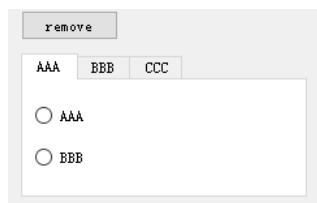
    B *pt=new B(&w);    //创建 QTabWidget 部件
//把容器添加到对应的选项卡之下。
    pt->addTab(pw,"AAA");    pt->addTab(pw1,"BBB");    pt->addTab(pw2,"CCC");
//以下步骤可使用布局代替，以避免设置部件的位置和大小
    pb->move(22,22);    pt->move(22,55);    pt->resize(222,111);

    cout<<pt->indexOf(pw1)<<endl;    //输出 1。返回 pw1 所在的索引
    cout<<pt->indexOf(pr)<<endl; //输出 -1，因为 pr 是 pw 的子部件，而不是 pt 的子部件。
    QObject::connect(pb, &QPushButton::clicked, pt, &B::f);

//使用 QTabWidget 可省略类似以下的选项卡与容器的信号和槽的关联步骤。
//QObject::connect(pt, SIGNAL(currentChanged(int)), ps, SLOT(setCurrentIndex(int)));
    w.resize(300,200);    w.show();    return a.exec(); }

```

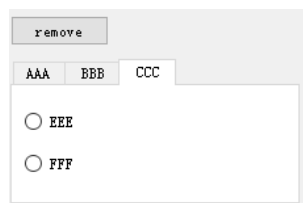
运行结果及说明



选项卡 AAA 显示的界面



选项卡 BBB 显示的界面



选项卡 CCC 显示的界面

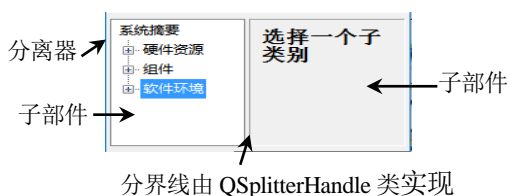
可见，选择不同的选项卡，呈现出了不同的界面，点击按钮 remove 会移除第 1 个选项卡。

5.4 QSplitter(分离器或分隔符)

QSplitter、QSplitterHandle

一、QSplitter 类(分离器)

- 1、QSplitter 类继承自 QFrame 类，也就是说该类是一个带有边框的可视部件。
- 2、QSplitter 类实现分离器，分离器用于分离两个部件，用户可通过拖动部件之间的分界线来调整子部件的大小。



- 3、QSplitter 的原理(见上图): QSplitter 的实现原理与 QBoxLayout 布局的原理类似,即 QSplitter 把子部件以水平或垂直的方式添加到 QSplitter 中,只不过在这些子部件之间多了一条分界线,另外 QSplitter 继承自 QFrame,因此 QSplitter 是有边框的。而且它可以作为容器和窗口使用,但不推荐使用 QPushButton pb(&splitter)的形式向 QSplitter 中添加子部件。

4、分界线:

- 分界线的创建:分界线是由 QSplitterHandle 类实现的,QSplitter 类本身不实现分界线。因此 QSplitterHandle 是一个部件,而 QSplitter 是另一个部件,这是两个部件,只不过这两个部件通过 Qt 的内部设计,让他们关联在一起,产生了一定的联系。
- 背景色: QSplitter 的分界线默认有可能是看不见的(因为颜色与 QSplitter 背景色相同,所以看不见),因此要使分界线可见,需设置分界线的背景色。
- 分界线的索引:其索引从 0 开始编号,分界线的数量与子部件的数量一样多。但索引为 0 的分界线始终是隐藏的。对于垂直分离器,索引为 0 的子部件上方的分界线索引为 0,对于水平界面(对于从左到右的语言),则索引为 0 的子部件左侧的分界线的索引为 0,对于从右到左的语言,则索引为 0 的子部件右侧的分界线的索引为 0。

- 5、动态调整:是指移动分界线时部件大小随之动态的改变,若不是动态调整的,则在移动分界线时部件大小不会改变,当完成分界线的移动时(即松开鼠标时),部件的大小才改变。

- 6、分离器的特点:分离器中的子部件分随着分离器大小的改变而改变,当分离器大小改变时,分离器会重新分配空间,以使其所有子部件的相对大小保持相同的比例不变。
子部件的大小策略对分离器不起作用,分离器会把子部件填充满整个空间,即使子部件的大小策略设置为 Fixed,仍会被拉伸。

7、使用 QSplitter 的步骤如下:

- ①、创建一个 QSplitter。
- ②、使用 addWidget()函数把子部件添加到 QSplitter 中。
- ③、代码如下:

```
QSplitter *ps = new QSplitter; //创建分离器,默认为水平排列子部件。
```

```

QPushButton *pb, *pb1, *pb2; .....;    //创建子部件
.....//初始化子部件。
ps->addWidget(pb);    //把子部件添加到分离器 ps 中
ps->addWidget(pb1);
.....    //添加其他子部件
ps->setStyleSheet("QSplitter::handle{ background-color: blue}");    //使用样式表把 QSplitter 分界线的
背景色设置为蓝色，这是一种快速设置所有分界线背景色的方法。

```

8、QSplitter 类的属性

①、**childrenCollapsible**: bool

访问函数: bool childrenCollapsible() const; void setChildrenCollapsible(bool);

- 子部件是否可以被折叠(即调整到 0 的大小)，默认为可折叠。
- 若部件可折叠，即使该部件的最小大小(minimumSize())不为零，用户也可将其大小调整为 0。
- 设置该属性会作用于所有子部件上，使用函数 setCollapsible()可设置单个子部件是否可折叠。

②、**handleWidth**: int 访问函数: int handleWidth() const; void setHandleWidth(int);

设置和获取分界线的宽度，默认取决于平台和样式。若设置为 0 或 1，则实际抓取区域会增加到各自部件相重叠的几个像素。

③、**opaqueResize**: bool 访问函数: bool opaqueResize() const; void setOpaqueResize(bool opaque = true);

部件是否动态调整，即分界线是否是不透明的。默认为动态调整(即为 true)。

④、**orientation**: Qt::Orientation

访问函数: Qt::Orientation orientation() const; void setOrientation(Qt::Orientation);

获取或设置 QSplitter 的方向，默认为水平方向，可取值为 Qt::Horizontal 和 Qt::Vertical。

9、QSplitter 类的函数

①、**QSplitter**(QWidget* parent = Q_NULLPTR);

QSplitter(Qt::Orientation orientation, QWidget* parent = Q_NULLPTR);

②、void **addWidget**(QWidget* widget);

void **insertWidget**(int index, QWidget* widget);

把 widget 添加到末尾或插入到指定索引 index 处，若 widget 已在分离器中，则将其移至新位置。注意：分离器会获得部件的所有权。

③、void **setCollapsible**(int index, bool collapse);

bool **isCollapsible**(int index) const;

设置或返回索引 index 处的子部件是否可折叠，要设置所有子部件都可折叠，请设置 childrenCollapsible 属性。

④、void **setStretchFactor**(int index, int stretch);

设置索引为 index 的子部件的拉伸因子。

⑤、void **setSizes**(const QList<int>& list);

QList<int> **sizes**() const;

- 使用列表 `list` 设置子部件的大小(以像素为单位), 其规则为, 若分离器是水平的, 则使用列表中的值按从左到右的顺序设置每个子部件的宽度。若分离器是垂直的, 则使用列表中的值按从上到下的顺序设置子部件的高度。若列表的值过少, 则结果未定义, 但程序仍能运行。
- 若为子部件指定的大小为 0, 则该子部件将不可见, 若指定的值小于子部件的最小大小, 则使用最小大小提示的值替换。
- 注意: 分离器的总体大小不会受到影响, 也就是说分离器的大小不会改变, 仍是之前那么大, 因此, 当使用该函数为所有的子部件设置的值小于分离器总体的大小时, 则其余多余的空间会根据设置的子部件大小的相对权重, 在各子部件之间分配。
- 示例: 假设分离器为垂直的, 且高度为 150, 共有 3 个子部件, 则

```
QList<int> q={30,30,30};           ps->setSizes(q);
```

设置之后各子部件的高度分别为 50, 50, 50, 而不是 30, 30, 30, 因为分离器剩余的空间按子部件大小的相对权重, 分配给了各个子部件。

⑥、QByteArray **saveState()** const;

```
bool restoreState(const QByteArray& sate);
```

以上函数用于保存和恢复分离器的状态, 通常应与 `QSettings` 类一起使用, 也可单独使用, 可以使用这两个函数来存储或恢复分离器的默认状态。

⑦、int **count()** const; //返回分离器中子部件的数量。

⑧、int **indexOf**(QWidget* widget) const; 返回 widget 在分离器中的索引, 该函数也可用于分界线。

⑨、QWidget* **widget**(int index) const; 返回索引 index 处的子部件。

⑩、QWidget* **replaceWidget**(int index, QWidget* widget); //qt5.9

把索引 index 处的子部件替换为 widget, 若 index 有效, 且 widget 不是分离器的子部件, 则返回被替换掉的子部件, 否则返回 null, 而不会进行替换, 新插入的子部件会继承被替换的子部件的属性(比如大小、可折叠装态等)。注意: widget 可能不会被立即设置, 需在接收到适到的事件之后才会被设置。

⑪、void **getRange**(int index, int *min, int *max) const;

返回索引为 index 的分界线的有效范围, 并存储在 *min 和 *max 中, 即获取分界线可以调整大小的范围。

⑫、QSplitterHandle* **handle**(int index) const; 返回索引 index 处的分界线。

⑬、void **refresh**(); //更新分离器的状态, 通常不需要设用这个函数。

⑭、QSplitterHandle* **createHandle**(); //虚拟的, 受保护的。

把返回的分界线作为分离器的分界线, 该函数可在子类中重新实现, 以提供自定义的分界线。

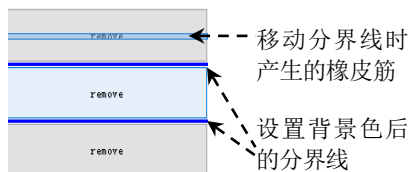
⑮、void **childEvent**(QChildEvent* c); //虚拟的, 受保护的。QObject::childEvent()的重新实现
当子部件已被插入或删除时产生该事件。

⑯、void **moveSplitter**(int pos, int index); //受保护的

把索引为 index 的分界线的左侧(或顶边), 尽可能的移至位置 pos 处, pos 是距离分离器左侧或顶边的距离。注: 对于从右到左的语言, pos 是距离分离器右侧的距离。

⑰、void **setRubberBand**(int pos); //受保护的

在位置 `pos` 处显示橡皮筋，若 `pos` 为负数，则移除橡皮筋。当分离器不是动态调整，用户移动分界线时，看到的那根线条就是橡皮筋，效果见右图。



⑱、void `splitterMoved`(int pos, int index); //信号。

当索引为 `index` 的分界线移至位置 `pos` 时，发送此信号。

示例 5.17: QSplitter 类(分离器)的使用

```
#include<QtWidgets>
int main(int argc, char *argv[]){    QApplication a(argc, argv);
    QWidget w;                        QSplitter *ps=new QSplitter(Qt::Vertical);
    QPushButton *pb=new QPushButton("AAA");    QPushButton *pb1=new QPushButton("BBB");
    QPushButton *pb2=new QPushButton("CCC");
    QPushButton *pb3=new QPushButton("DDD", ps);    //不推荐此种方式向分离器中添加子部件

    pb1->setSizePolicy(QSizePolicy::Fixed, QSizePolicy::Fixed); //大小策略不起作用
    pb->setMaximumSize(175, 55);    //最大最小值仍可限制子部件
    ps->addWidget(pb);        ps->addWidget(pb1);    ps->addWidget(pb2);
    ps->addWidget(pb3);    //pb3 已在分离器 ps 中，因此把 pb3 移至新位置
    //设置分离器的边框
    ps->setFrameShadow(QFrame::Raised);        ps->setFrameShape(QFrame::Box);
    ps->setLineWidth(5);

    ps->setChildrenCollapsible(0);    //子部件不可折叠
    ps->setOpaqueResize(1);    //动态调整子部件
    ps->setStretchFactor(0, 1);    //设置拉伸因子，以调整初始显示时各子部件间的位置
    ps->setStretchFactor(1, 2);    ps->setStretchFactor(2, 2);

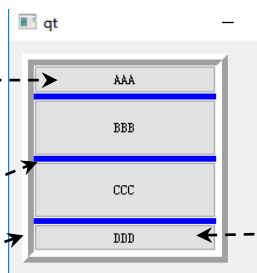
    ps->setHandleWidth(5);    //设置分界线的宽度
    //设置分界线的背景色为蓝色，若不设置背景色，则分界线可能会不可见
    ps->setStyleSheet("QSplitter::handle{background-color: blue}");
    QVBoxLayout *pv=new QVBoxLayout;    pv->addWidget(ps);    //把分离器 ps 添加到布局 pv 中
    w.setLayout(pv);        w.resize(300, 200);        w.show();    return a.exec(); }
```

运行结果及说明

当调整此按钮的大小时，最大只能拉伸至 55 个像素的高度，这说明最大大小的限制仍起作用。

蓝色的分界线

带边框的分离器



初始显示时各子部件的大小按设置的拉伸因子进行计算。

按钮 BBB 的大小策略未起作用。

此按钮被重新移至末尾。

调整主窗口大小改变分离器大小时，分离器中的各子部件的大小随之改变，但仍会保持之前的大小比例。

示例 5.18: QSplitter 综合示例(设计的界要和要求见示例后的图示)

//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>
#include <iostream>
using namespace std;
class B:public QSplitter{    Q_OBJECT
public:    static QString ddd;    //用于存储来自行编辑器输入的文本
        B(Qt::Orientation o=Qt::Horizontal,QWidget* p=0):QSplitter(o,p) {}
public slots:
    void f() {
        int j=0;        //计数器
        int i[3]={0};    //用于存储从文本 ddd 中提出取来的数字。
        QString sl=ddd;
        while(!sl.isEmpty()){
            sl=ddd.section(",",j,j);    //提取文本中以","分隔的数字
            if(j>2)return;    //如果文本中的数字个数大于等于 3 个，则退出函数。
            i[j]=sl.toInt();    //把提取出来的数字赋值给数组 i。
            j++;}
        moveSplitter(i[0],i[1]);    /*使用文本中输入的前两个数字设置分界线的位置。注意：这个函数是受保护的，若索引(第二个参数)超出范围，该函数会产生错误。*/
    }    };

class C:public QWidget{    Q_OBJECT
public:
    B *ps;        //分离器
    QLineEdit *pe,*pe1;    QLabel *pl,*pl1;
    QPushButton *pb,*pb1,*pb2,*pb3,*pb4,*pb5;
    QByteArray b;    //用于保存和恢复分离器的状态

    C(QWidget* p=0):QWidget(p) {        //构造函数
        ps=new B(Qt::Vertical);
        pl=new QLabel("EnterSize");        pe=new QLineEdit;
        pl1=new QLabel("pose,index");        pe1=new QLineEdit;
        pe->setClearButtonEnabled(1);    //显示行编辑器的清除按钮
        pe1->setClearButtonEnabled(1);        pb=new QPushButton("AAA");
        pb1=new QPushButton("BBB");        pb2=new QPushButton("CCC");
        pb3=new QPushButton("save");        pb4=new QPushButton("restore");
//布置分离器 ps 的子部件。
        ps->addWidget(pb);    ps->addWidget(pb1);    ps->addWidget(pb2);
//设置分离器的边框及其他属性
        ps->setFrameShadow(QFrame::Raised);    ps->setFrameShape(QFrame::Box);
        ps->setLineWidth(5);        ps->setChildrenCollapsible(0);
        //设置分离器的边框样式后，重新设置分离器的大小策略，否则分离器可能不会扩展。
        ps->setSizePolicy(QSizePolicy::Expanding,QSizePolicy::Expanding);
//设置分界线的背景色为蓝色，若不设置背景色，则分界线可能会不可见
        ps->setHandleWidth(5);    ps->setStyleSheet("QSplitter::handle{background-color: blue}");
//布置主窗口的子部件
        QVBoxLayout *pv=new QVBoxLayout;    //主窗口使用的主布局
        QFormLayout *pf=new QFormLayout;        pf->addRow(pl,pe);        pf->addRow(pl1,pe1);
        QHBoxLayout *ph=new QHBoxLayout;        ph->addWidget(pb3);        ph->addWidget(pb4);
```

```

//把布局 pf, ph 和分离器, 添加到主布局中
pv->addLayout(ph);    pv->addLayout(pf);    pv->addWidget(ps);    setLayout(pv);
QObject::connect(pe, &QLineEdit::returnPressed, this, &C::fSet);
QObject::connect(pb3, &QPushButton::clicked, this, &C::fSave);
QObject::connect(pb4, &QPushButton::clicked, this, &C::fRestore);
QObject::connect(pe1, &QLineEdit::returnPressed, this, &C::f);
QObject::connect(pe1, &QLineEdit::returnPressed, ps, &B::f);    } //构造函数结束

public slots:
void fSet() {          //使用 setSizes 函数设置分离器各子部件的大小
    QString ss=pe->text(); //获取行编辑器 pe 的文本
    int j=0; //计数器
    QString sl=ss;
    QList<int> b; //使用列表存储从文本 ss 中提取出来的数字
    while(!sl.isEmpty()) {
        sl=ss.section(",", j, j); //提取以","分隔的数字。
        b.append(sl.toInt()); //把提取出来的数字追加到列表 b 的末尾。
        j++;
    }
    if(b.size()<=3) return; //如果输入的数字个数少于 3 个, 则退出函数。
    ps->setSizes(b); //使用列表 b 设置各分离器子部件的位置
}

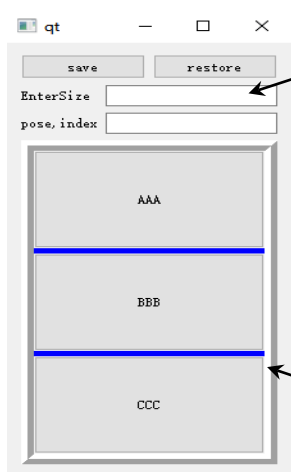
void fSave() { b=ps->saveState();    cout<<"save OK"<<endl;} //保存分离器的状态。
void fRestore() { ps->restoreState(b); } //恢复分离器的状态
void f() { B::ddd=pe1->text(); } //把行编辑器 pe1 输入的内容存储到静态变量 B::ddd 中
};

#endif // M_H

//m.cpp 文件的内容
#include "m.h"
QString B::ddd; //初始化静态变量, 注意: 静态变量不应在头文件中初始化, 否则会出现重定义错误。
int main(int argc, char *argv[]) {    QApplication a(argc, argv);
    C w;    w.resize(300, 500);    w.show();    return a.exec(); }

```

运行结果及说明



此行编辑器关联到的是 C::fSet 函数, 该输入框需要输入 3 个以逗号分隔的数字。按下回车后, 分离器中的三个按钮将按照输入的数字调整高度, 读者可自行验证 setSizes 函数的规律

此行编辑器关联到的是 B::fSet 函数, 该输入框需要输入 2 个以逗号分隔的数字。按下回车后会把由第 2 个数字指定的分界线移至第 1 个数字指定的位置。注意: 索引 0 的分界线是隐藏不可见的

按下 save 按钮, 可随时保存分离器各子部件的状态, 按下 restore 可恢复其状态。

带边框的分离器

示例 5.19: QSplitter 的嵌套使用

该示例实现的界面如下图

```
#include<QtWidgets>
int main(int argc, char *argv[]) {
    QApplication a(argc, argv);
    QWidget w;
    QSplitter *ps=new QSplitter(Qt::Vertical);
    QSplitter *ps1=new QSplitter(Qt::Vertical);
    QSplitter *ps2=new QSplitter(Qt::Horizontal);
    QTextEdit *pt=new QTextEdit;
    QTextEdit *pt1=new QTextEdit;
    QTextEdit *pt2=new QTextEdit;

    //设置分界线的背景色及边框
    ps->setHandleWidth(1);    ps->setStyleSheet("QSplitter::handle{background-color: blue}");
    ps2->setFrameShadow(QFrame::Raised);    ps2->setFrameShape(QFrame::Box);
    ps2->setLineWidth(2);        ps2->setHandleWidth(5);
    ps2->setStyleSheet("QSplitter::handle{background-color: red}");
    //布局子部件
    ps->addWidget(pt);        ps->addWidget(pt1);        ps1->addWidget(pt2);
    ps2->addWidget(ps1);        ps2->addWidget(ps);
    QHBoxLayout *ph=new QHBoxLayout;    ph->addWidget(ps2);    w.setLayout(ph);
    w.resize(300, 200);        w.show();    return a.exec(); }
```



二、QSplitterHandle 类(分界线)

- 1、QSplitterHandle 类继承自 QWidget，该类主要用于实现 QSplitter(分离器)的分界线，因此，通常与分离器一起使用。
- 2、QSplitterHandle 类比较简单，只有如下几个公有函数
QSplitterHandle(Qt::Orientation orientation, QSplitter* parent); //构造函数
QSplitter* **splitter**() const; //获取与此分界线关联的分离器
void **setOrientation**(Qt::Orientation orientation); //设置分界线的方向
Qt::Orientation **orientation**() const; //返回分界线的方向
bool **opaqueResize**() const; //设置是否不透明(动态调整)，该值由 QSplitter 控制。
- 3、该类需结合 QSplitter 的如下两个函数使用，
QSplitterHandle* QSplitter::**handle**(int index) const; //通过该函数可间接的修改分界线。
QSplitterHandle* QSplitter::**createHandle**(); //虚拟的，受保护的，重写此函数，可实现自定义的分界线。

示例 5.20: QSplitterHandle(分界线)的使用

//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>

class B:public QSplitter{    Q_OBJECT
public:    B(Qt::Orientation o=Qt::Horizontal, QWidget* p=0):QSplitter(o, p) {}
```



```

    public slots:
    QSplitterHandle *createHandle() {    //重新实现该虚函数，创建自定义的分界线
        QSplitterHandle *ph=new QSplitterHandle(orientation(),this);    //自定义的分界线
        ph->setMinimumSize(22,22);    //设计分界线的最小大小
        ph->setPalette(QPalette(QColor(111,1,1)));    //设置分界线的颜色
        return ph;    //返回自定义的分界线
    };
#endif // M_H

```

//m.cpp 文件的内容

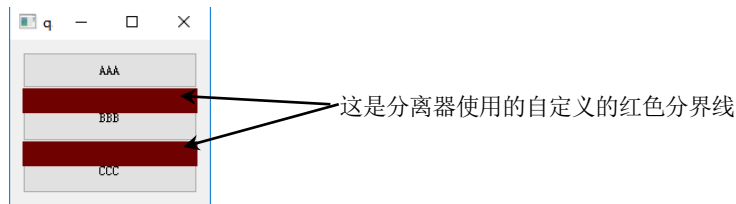
```

#include "m.h"
int main(int argc, char *argv[]) {    QApplication a(argc, argv);
    QWidget w;    B *ps=new B(Qt::Vertical);

    QPushButton *pb=new QPushButton("AAA");    QPushButton *pb1=new QPushButton("BBB");
    QPushButton *pb2=new QPushButton("CCC");
    ps->addWidget(pb);    ps->addWidget(pb1);    ps->addWidget(pb2);
    QHBoxLayout *ph=new QHBoxLayout;    ph->addWidget(ps);    w.setLayout(ph);
    w.resize(300,200);    w.show();    return a.exec();    }

```

运行结果及说明



5.5 自定义布局管理器

QLayout、QLayoutItem、QSpacerItem、QWidgetItem

- 1、自定义布局需要使用 QLayout 和 QLayoutItem 类，其中 QLayoutItem 类描述了 QLayout 布局中的项目信息。

一、QLayout 抽象类中的公有成员函数

- 1、QLayout 继承自 QObject 和 QLayoutItem 类，该类是一个抽象类。该类中的成员在前文基本上都见过了
- 2、QLayout 类中的属性

sizeConstraint: SizeConstraint //大小约束，详见前文

spacing: int //设置部件之间的空白距离，详见前文。

3、QLayout 类中的公有函数

①、QLayout()

QLayout(QWidget* parent); //构造函数

②、void addWidget(QWidget *w); //将 w 添加到布局，该函数使用 addItem()

③、QWidget* parentWidget() const;

返回该布局的父部件，若未安装在任何部件上则返回 0。若该布局是子布局，则返回父布局的父部件。

④、void removeItem(QLayoutItem* item);

void **removeWidget**(QWidget* widget);

以上函数表示，从布局中移除部件(注意：并未删除)widget 或布局项目 item，在调用之后，调用者有责任给 widget 一个合理的几何尺寸，或者把该部件放回布局中，或者明确隐藏。对于 item，则调用方有责任将其删除。注意：widget 的所有权与添加时相同。item 可以是布局(因为 QLayout 是 QLayoutItem 的子类)。

⑤、QLayoutItem* replaceWidget(QWidget* from, QWidget* to,

Qt::FindChildOptions options = Qt::FindChildrenRecursively); //qt5.2

使用部件 to 替换部件 from(注意：from 并未被删除)，如果成功，则返回包含部件 from 的布局项目，若 options 为 Qt::FindChildrenRecursively(默认值)，则还会搜索子布局，因此返回的布局项目有可能不属性此布局，而是属于子布局的。返回的布局项目不再属于此布局，应把该项目删除或插入到其他布局中，部件 from 不再由布局管理，可能需要删除或隐藏，from 的父部件保持不变。此函数适用于 Qt 内部调用，可能不适用于自定义布局。

⑥、bool isEnabled() const;

void **setEnabled**(bool enable);

以上函数用于获取和设置布局是否启用。若布局被禁用，其行为就像该布局不存在一样，默认情况下布局都可用。

⑦、QRect contentsRect() const; //返回布局的几何尺寸(即布局的大小)，包括内容边距。

- ⑧、QMargins **contentsMargins()** const;
void **getContentsMargins**(int *left, int *top , int *right, int *bottom) const;
void **setContentsMargins**(int left, int top, int right, int bottom);
void **setContentsMargins**(const QMargins &margins);
以上函数用于设置或获取内容边距(即页边距)，大多数平台上，所有方向的边距默认都为 11 像素。
- ⑨、bool **setAlignment**(QWidget*w , Qt::Alignment alignment);
bool **setAlignment**(QLayout* l, Qt::Alignment alignment);
将部件 w 或 l 的对齐方式设置为 alignment，若在该布局(不包括子布局)中找到 w 或 l，则返回 true，否则返回 false。
- ⑩、void **setMenuBar**(QWidget* widget);
把菜单栏部件 widget，放置在 parentWidget()的 QWidget::contentsMargins()之外的顶部，所有子窗口部件都放置在菜单栏底部的下方。
- ⑪、QWidget* **menuBar()** const; //返回布局的菜单栏，若没有菜单栏则返回 0。
- ⑫、void **update()**;
bool **activate()**
以上函数分别表示，更新或重新设置 parentWidget()的布局，以上函数在合适的时候会
自动调用，因此通常不需调用。若布局重设，则 activate()返回 true。
- ⑬、virtual int **indexOf**(QWidget* widget) const; //虚拟的
返回 widget 的索引，若没有找到，则返回-1，默认实现使用 itemAt()迭代所有项。

二、QLayoutItem、QSpacerItem、QWidgetItem 类

- 1、布局项目(QLayoutItem)：指的是添加到布局中由布局管理的元素，布局管理器并不能直接管理 QWidget 类型的子对象，而是管理 QLayoutItem 及其子类型的对象(为讲解方便，把其称为 QLayoutItem 对象)，对于 QWidget 这种非 QLayoutItem 对象的对象，需要把其转换为 QLayoutItem 对象，才能使用布局进行管理。
- 2、QLayoutItem 类是用于描述 QLayoutItem 对象的一个抽象基类，除非需要创建自定义的 QLayoutItem 对象，否则不需要用户子类化该类，因此该类通常很少被使用，而是使用他的子类 QSpacerItem、QWidgetItem、QLayout，自定义布局时通常子类化 QLayout 类，另外，该类除了纯虚函数(见自定义布局的实现小节)外，以下函数被子类(比如 QWidgetItem 类)重新实现后，也会被用到
- ①、virtual QWidget* **QLayoutItem::widget()** //虚函数，返回项目管理的 QWidget 类型的部件
- ②、virtual QSizePolicy::ControlTypes **QLayoutItem::controlTypes()** const; //虚函数
返回项目所管理的部件的类型，部件的类型使用 QSizePolicy::ControlType 枚举描述，其取值如下表

QSizePolicy::ControlType 枚举			
标志：QSizePolicy::ControlTypes			
成员	值	成员	值
QSizePolicy::DefaultTeyp	0x0000 0001	QSizePolicy::LineEdit	0x0000 0100

QSizePolicy::ButtonBox	0x0000 0002	QSizePolicy::PushButton	0x0000 0200
QSizePolicy::CheckBox	0x0000 0004	QSizePolicy::RadioButton	0x0000 0400
QSizePolicy::ComboBox	0x0000 0008	QSizePolicy::Slider	0x0000 0800
QSizePolicy::Frame	0x0000 0010	QSizePolicy::SpinBox	0x0000 1000
QSizePolicy::GroupBox	0x0000 0020	QSizePolicy::TabWidget	0x0000 2000
QSizePolicy::Label	0x0000 0040	QSizePolicy::ToolButton	0x0000 4000
QSizePolicy::Line	0x0000 0080		

3、QSpacerItem 类，该类在前文已讲解并使用过，这个类主要用来创建一个空白项目，以使布局看起来更合理，在此就不重述了。

4、QWidgetItem 类，其主要作用是可以根据 QWidget 部件产生一个可由布局管理的 QWidgetItem 对象，使用方法如下：

```
QWidget *w = new QWidget;
```

```
QWidgetItem *pi = new QWidgetItem(w); //把 w 转换为 QWidgetItem 对象，并存储在 pi 中。
```

使用 QWidgetItem 类重新实现的 widget()函数可实现上述相反的转换，比如

```
QWidget *pw = pi->widget(); //返回由项目 pi 管理的部件
```

示例 5.21：部件的移除与删除(设计的界面见右下图)

//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>
class B:public QWidget{    Q_OBJECT
public:
    QVBoxLayout *pv;
    QHBoxLayout *ph;
    LayoutItem* pi;    QWidget *pw; //用于保存值
    QPushButton *pb,*pb1,*pb2,*pb3,*pb4,*pb5,*pb6,*pb7;
    B(QWidget* p=0):QWidget(p) {
        pv=new QVBoxLayout;
        pw=0;    pi=0; //初始化为 0;
        pb=new QPushButton("AAA");    pb1=new QPushButton("BBB");    pb2=new QPushButton("CCC");
        pb3=new QPushButton("DDD");    pb4=new QPushButton("replace");
        pb5=new QPushButton("remove");    pb6=new QPushButton("take");    pb7=new QPushButton("del");
        pv->addWidget(pb);    pv->addWidget(pb1);    pv->addWidget(pb2);
        ph=new QHBoxLayout;
        ph->addWidget(pb4);    ph->addWidget(pb5);    ph->addWidget(pb6);    ph->addWidget(pb7);
    }
//主布局
    QVBoxLayout *pv1=new QVBoxLayout;    pv1->addLayout(ph);    pv1->addLayout(pv);
    setLayout(pv1);
    connect(pb4,&QPushButton::clicked,this,&B::f); //替换 replaceWidget
    connect(pb5,&QPushButton::clicked,this,&B::f1); //移除 removeWidget
    connect(pb6,&QPushButton::clicked,this,&B::f2); //takeAt
    connect(pb7,&QPushButton::clicked,this,&B::f3);    } //彻底删除，构造函数结束
public slots:
    void f() { pi=pv->replaceWidget(pb,pb3); //把 pb 替换为 pb3，但 pb 未被删除。
        if(pi->controlTypes()==QSizePolicy::PushButton) { //若 pi 管理的部件是按钮。
            pw=pi->widget();    } //把 pi 管理的部件赋值给 pw
    }
    void f1() { pv->QLayout::removeWidget(pb1); //移除但不删除 pb1,
```



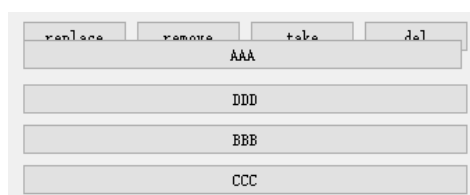
```

        pw=pbl;    }
void f2() {    pi=pv->QBoxLayout::takeAt(0);    /*使用 QBoxLayout 类重新实现的 takeAt 函数，移
        除但不删除索引为 0 的部件，注：QLayout 并未实现 takeAt 纯虚函数*/
        if(pi->controlTypes()==QSizePolicy::PushButton){    pw=pi->widget();}    }
void f3() {    //该函数用于删除项目和部件
    //若仅仅删除布局项目 pi，并不会把布局所管理的部件 pw 删除掉，因此部件需明确的删除。
        if(pi!=0) {delete pi; pi=0;}        if(pw!=0) {delete pw; pw=0;}    }    };
#endif // M_H

//m.cpp 文件的内容
#include "m.h"
int main(int argc, char *argv[]) {    QApplication a(argc, argv);
    B w;    w.resize(300,200);    w.show();    return a.exec();    }

```

运行结果及说明



不要重复替换已经替换掉的按钮(即不能点击两次 replace 按钮)，否则会发生错误。点击 replace 按钮，并改变窗口大小后，可发现，被替换掉的按钮 AAA 并未删除，仍然存在，按钮 AAA 此时还可用作其他用途(比如添加到其他布局等)，本示例选择删除按钮 AAA，只需点击 del 即可将其删除，按钮 AAA 删除之后将不能再用作其他用途。点击 remove 和 take 按钮后，其对应的按钮都未被删除，效果与 replace 类似。

三、自定义布局的实现

- 1、注意：布局不能管理非 QLayoutItem 对象，QWidget 对象需转换为 QLayoutItem 对象才能被布局管理。
- 2、纯虚函数：虽然纯虚函数可由用户自行实现任意功能，但是这些纯虚函数通常需要由 Qt 内部调用，因此，若用户实现的纯虚函数不满足 Qt 内部的要求，则不但达不到预期的效果，还有可能使程序出错。因此纯虚函数的功能也是需要了解的。
- 3、要自定义布局，需要重新实现以下纯虚函数

count(); addItem(); itemAt(); takeAt(); sizeHint(); setGeometry()

其中，setGeometry()函数不是必须重新实现的，但该函数管理着怎样对子部件进行布置(设置其大小、位置等)，因此通常还需要重新实现该函数。

4、以下为 QLayout 类中的纯虚函数，自定义布局时，必须重新实现以下函数

- ①、virtual int count() const = 0; //纯虚函数

必须在子类中重新实现，作用是返回布局中项目的数量。

- ②、virtual void addItem(QLayoutItem* item)=0; //纯虚函数

把项目 item 添加到布局中，该函数必须在子类中重新实现，以添加特定于每个子类的

项目，该函数通常不需要在程序中调用，因为 `QLayout` 的子类提供了相关的函数，比如要添加布局，可使用子类的 `addLayout()` 函数，要添加部件，使用 `addWidget()` 函数等。

- ③、virtual `QLayoutItem itemAt(int index) const = 0;` //纯虚函数

必须在子类中实现，以返回索引 `index` 处的布局项目，若没有这样的项目，则必须返回 0，项目从为开始编号，若项目被删除，则其他项目将被重新编号。

- ④、virtual `QLayoutItem* takeAt(int index) = 0;` //纯虚函数

必须在子类中实现，以从布局中删除索引 `index` 处的布局项目，并返回该项目。若没有这样的项目，该函数必须什么也不做并返回 0，项目的编号从 0 开始，若项目被删除，则其他项目会被重新编号。下面是安全删除布局中项目的方法

```
QLayoutItem *c;    while((c = layout -> takeAt(0)) != 0) { .....; delete c; }
```

5、以下函数为 `QLayout` 类重新实现的父类 `QLayoutItem` 中的纯虚函数

注意：`QLayoutItem` 类共有 7 个纯虚函数，但 `QLayout` 只实现了 6 个，其中 `sizeHint()` 函数未重新实现。

- ①、virtual `maximumSize() const;` ②、virtual `QSize minimumSize() const;`

- ③、virtual `bool isEmpty() const;` ④、virtual `QRect geometry() const;`

- ⑤、virtual `setGeometry(const QRect& r);`

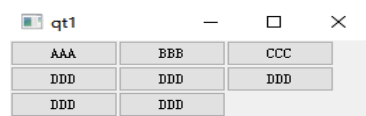
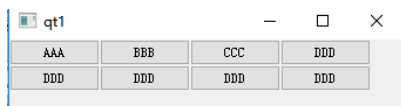
当实现自定义布局时，此函数的作用就是对布局中的子部件进行布置，即子部件的大小、排列等都在此函数内完成的，因此若要实现自定义布局，则此函数需重新实现。

- ⑥、virtual `Qt::Orientations expandingDirections() const;`

设置此布局的拉伸方式(即是否可以获得比 `QLayoutItem::sizeHint()` 更多的空间)，若值为 `Qt::Vertical` 或 `Qt::Horizontal` 则只能在垂直或水平方向拉伸，若值为 `Qt::Vertical | Qt::Horizontal` (默认)则可在两个方向拉伸。子类会根据子部件的大小策略，重新实现该函数，以返回有意义的值。

示例 5.22：自定义布局

这是一个简单的示例，该示例实现如下效果的布局，当调整窗口大小时，子部件会自动调整至上一行或下一行，其中子部件不会拉伸或压缩



//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>
class B : public QLayout{
public:    B(QWidget *parent): QLayout(parent) {}    B() {}    ~B();
//声明需要实现的成员函数
    void addItem(QLayoutItem *item);    QSize sizeHint() const;
    int count() const;    QLayoutItem *itemAt(int) const;
    QLayoutItem *takeAt(int);    void setGeometry(const QRect &rect);
```

```

    void addw(QWidget* pw); /*使用一个自定义的函数向布局中添加 QWidget 对象, addItem 函数不能
                             直接接收 QWidget 对象, 该函数主要起类型转换的作用。*/
    QList<QLayoutItem*> list; //使用 QList 存储布局需要管理的对象。
}; //类 B 的声明结束

void B::addItem(QLayoutItem *item){ list.append(item);} //把元素添加到列表。
void B::addw(QWidget* p) {
    addItem(new QWidgetItem(p)); /*把 p 转换为 QLayoutItem 对象, 非 QLayoutItem 对象不能由布局
                                   管理。*/
    //addItem((QLayoutItem*)p); /*错误, 强制类型转换指针的类型, 会使内存的内容被重新解释, 这
                                   可能会产生内存错误。比如 int a=1; int *p=&a; 假设 int 占 4 字节, double
                                   占 8 字节, 则*p 只会读取 4 字节的内容, 但是*(double*)p;则会读取 8 字节
                                   的内容(详见 C++语法详解一书)。*/
}

QLayoutItem *B::itemAt(int i) const{return list.value(i);} //返回索引 i 处的项目。
QLayoutItem *B::takeAt(int i) //删除索引 idx 处的项目
{ return i >= 0 && i < list.size() ? (QLayoutItem*)list.takeAt(i) : 0; }
int B::count() const{ return list.size(); } //返回布局中的项目数量
B::~~B() { //因为 QLayoutItem 未继承自 QObject, 因此必须手动删除 QLayoutItem 对象。
    QLayoutItem *item; while ((item = takeAt(0))) delete item;}
void B::setGeometry(const QRect &r) { //布置布局中的子项目
    QSize s=parentWidget()->size(); //获取布局所在父部件的大小
    int w=sizeHint().width(); int h=sizeHint().height();
    int x=0; int y=0; //部件左上角的坐标。
    for(int i=0;i<list.size();i++){
        list.at(i)->setGeometry(QRect(x, y, w, h));
        x=x+w; //第二个项目的水平坐标向后移第一个部件的宽度
        if(x+w>s.width()) /*如果新添加的项目占据的位置超过了父部件的大小, 则该部件添加到下
                           一行的开头*/
            { y=y+h; x=0;} }
    QSize B::sizeHint() const{ return QSize(77, 22); }
#endif // M_H

//m.cpp 文件的内容
#include "m.h"
int main(int argc, char *argv[]){ QApplication a(argc, argv);
    QWidget w;
    QPushButton *pb=new QPushButton("AAA"); QPushButton *pb1=new QPushButton("BBB");
    QPushButton *pb2=new QPushButton("CCC"); QPushButton *pb3=new QPushButton("DDD");
    QPushButton *pb4=new QPushButton("DDD"); QPushButton *pb5=new QPushButton("DDD");
    QPushButton *pb6=new QPushButton("DDD"); QPushButton *pb7=new QPushButton("DDD");
    B *ph=new B;
    ph->addWidget(pb); ph->addWidget(pb1); ph->addWidget(pb2);
    ph->addWidget(pb3); ph->addWidget(pb4); ph->addWidget(pb5);
    ph->addWidget(pb6); ph->addWidget(pb7);
    w.setLayout(ph); w.resize(300,200); w.show(); return a.exec(); }

```

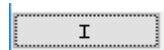
作者：黄邦勇帅(原名：黄勇)

2018-4-12

5.6 Qt 焦点系统

QFocusFrame 及 QWidget 类中与焦点有关的函数

焦点如右图所示，焦点通常使用 tab 键进行移动，在同一个窗口中，使用 tab 键来移动焦点时，焦点通常是在各控件之间循环移动的



按钮 I 周围的虚线框就是焦点

一、焦点链(焦点循环)

- 1、焦点循环：在使用 tab 键移动焦点时，焦点会在各部件之间循环移动。因此可使用一个循环链表的结构来管理焦点，该链表称为焦点链。Qt 会自动生成窗口中各子部件的焦点链
- 2、焦点顺序(或 tab 顺序)：按下 tab 时，焦点的移动顺序称为焦点顺序或 tab 顺序。
- 3、以下函数为与焦点链有关的函数

- ①、QWidget* **QWidget::nextInFocusChain()** const; //返回此部件焦点链中的下一个部件
- ②、QWidget* **QWidget::previousInFocusChain()** const; //返回此部件焦点链中的前一个部件
- ③、static void **QWidget::setTabOrder**(QWidget* first, QWidget* second); //静态的

将焦点顺序中的部件 second 放置在部件 first 之后，该函数把部件 second 从其焦点链中移除，并插入在部件 first 之后，若 first 或 second 拥有焦点代理，则该函数会正确处理焦点代理。注意：子部件 first 和子部件 second 需位于同一个窗口之中。

- 4、当使用 setTabOrder()函数时，只会改变该焦点链中各部件之间的顺序，比如，若默认的焦点链顺序为 a-b-c-d，则

setTabOrder(d,c); //改变后焦点链的顺序为 a-b-d-c;

setTabOrder(b,a); //焦点链的顺序为 b-a-d-c，最终焦点的移动顺序为 b-a-d-c-b-a-d-c....

- 5、窗口接受焦点的策略

- ①、若窗口没有子部件，或其子部件都不接受焦点，窗口仍然可能会拥有焦点。
- ②、当用户把焦点移至某个窗口时，应用程序必须确定窗口中的哪个部件应该获得焦点，通常的策略是，若在此之前窗口拥有过焦点，则最后一个具有焦点的部件将重新获得焦点，若未对此进行设置，则 Qt 会自动执行此操作。若在此之前窗口从未拥有过焦点，则通常是焦点链中的第一个部件获得焦点。

示例 5.23: 使用 setTabOrder 函数设置焦点链的顺序

```
#include<QtWidgets>
int main(int argc, char *argv[]) {    QApplication aa(argc, argv);
    QWidget w,w1; //创建两个窗口
    QGridLayout *pg=new QGridLayout;;    QPushButton *a,*b,*c,*d,*e,*f,*g,*h,*i;
//设置容器 pw 中的部件
    QWidget *pw=new QWidget(&w); //pw 用作容器，不用作窗口
    a=new QPushButton("A");        b=new QPushButton("B");        c=new QPushButton("C");
    d=new QPushButton("D");        e=new QPushButton("E");        f=new QPushButton("F");
    pg->addWidget(a, 0, 0);        pg->addWidget(b, 0, 1);        pg->addWidget(c, 1, 0);
```

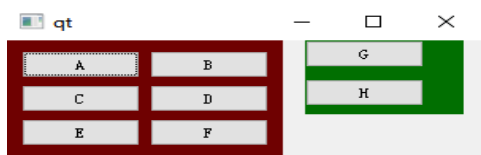


```

pg->addWidget(d, 1, 1);          pg->addWidget(e, 2, 0);          pg->addWidget(f, 2, 1);
pw->setLayout(pg);
//设置容器 pw1 中的部件
QWidget *pw1=new QWidget(&w);    pw1->move(188, 0);
g=new QPushButton("G", pw1);      h=new QPushButton("H", pw1);      h->move(0, 33);
//设置窗口 w1 中的部件
i=new QPushButton("I", &w1);
//设置容器 pw 和 pw1 的背景色
QPalette pt, pt1;    //创建调色板
pt.setColor(QPalette::Window, QColor(111, 1, 1)); //设置背景色
pw->setPalette(pt);    //为 pw 安装调色板
pw->setAutoFillBackground(true); //自动填充背景，此函数必须调用，否则背景可能不会被更新。
pt1.setColor(QPalette::Window, QColor(1, 111, 1));
pw1->setPalette(pt1);    pw1->setAutoFillBackground(true);
//设置焦点顺序，默认焦点顺序为 a-b-c-d-e-f-g-h
w.setTabOrder(d, g);    //a-b-c-d-g-e-f-h
w.setTabOrder(d, c);    //a-b-d-c-g-e-f-h
w.setTabOrder(c, a);    //b-d-c-a-g-e-f-h
//w.setTabOrder(h, i); //无效设置，h 和 i 位于不同的窗口中
w1.resize(222, 111);    w1.show();    w.resize(300, 200);    w.show();    return aa.exec();}

```

运行结果及说明



焦点的循环次序为
b-d-c-a-g-e-f-h-b-d....

二、获取焦点信息

1、QWidget* **QWidget::focusWidget()** const;

返回该部件(比如作为窗口的部件，该窗口不一定要处于激活状态)中最后一次获得焦点的子部件，对于顶级窗口，这是在窗口激活时，将获得焦点的部件。

2、static QWidget* **QApplication::focusWidget()**; //静态，返回当前活动窗口中具有焦点的部件。

3、**focus**: const bool **访问函数**: bool hasFocus() const; //QWidget 类中的属性

部件是否具有焦点，获取此属性的值，相当于检查 **QApplication::focusWidget()**是否指向该部件。默认为 false。

4、**isActiveWindow**: const bool **访问函数**: bool isActiveWindow() const; //QWidget 类中的属性

该部件的窗口是否是活动窗口，默认为 false。

5、void **QWidget::activateWindow()**;

把包含此部件的顶级部件设置为活动窗口，此函数与在顶级窗口标题栏上单击鼠标执行相同的操作。注意：窗口必须可见，否则该函数不起作用。在 X11 上，该函数的结果取决于窗口管理器。

示例 5.24: QWidget::focusWidget 与 QApplication::focusWidget 的区别

//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>
#include<QDebug>
class B:public QWidget{    Q_OBJECT
public:    QPushButton *a,*b,*c,*d,*e,*f,*g,*h;    QWidget *pw;    //pw 用作窗口

    B(QWidget* p=0):QWidget(p) {

//设置窗口 pw 中的部件
    pw=new QWidget;    QGridLayout *pg=new QGridLayout;
    a=new QPushButton("&A");    b=new QPushButton("B");
    c=new QPushButton("C");    d=new QPushButton("D");
    pg->addWidget(a,0,0);    pg->addWidget(b,0,1);
    pg->addWidget(c,1,0);    pg->addWidget(d,1,1);
    pw->setLayout(pg);

//设置容器 pw1 中的部件
    QWidget *pw1=new QWidget(this);    QGridLayout *pg1=new QGridLayout;
    e=new QPushButton("E");    f=new QPushButton("F");
    g=new QPushButton("G");    h=new QPushButton("H");
    pg1->addWidget(e,0,0);    pg1->addWidget(f,0,1);
    pg1->addWidget(g,1,0);    pg1->addWidget(h,1,1);
    pw1->setLayout(pg1);

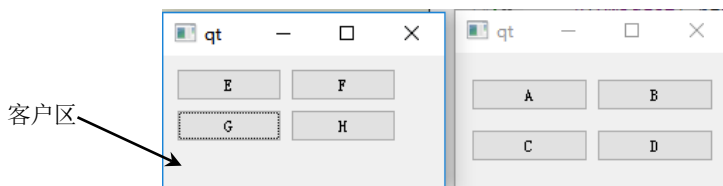
//设置各按钮的对象名
    a->setObjectName("A");    b->setObjectName("B");    c->setObjectName("C");
    d->setObjectName("D");    e->setObjectName("E");    f->setObjectName("F");
    g->setObjectName("G");    h->setObjectName("H");

    pw->resize(333,222);    pw->show();    //显示窗口
    connect(a,&QPushButton::clicked,this,&B::f1);    }    //构造函数结束
public slots:
    void mousePressEvent(QMouseEvent* e){
        a->activateWindow();    }    //按下鼠标使拥有部件 a 的窗口成为激活状态。
    void f1() {    //测试 QWidget::focusWidget 与 QApplication::focusWidget 的不同
        QWidget *pw=this->focusWidget();    qDebug()<<pw->objectName();
        QWidget *pa=QApplication::focusWidget();    qDebug()<<pa->objectName();
    };
#endif // M_H
```

//m.cpp 文件的内容

```
#include "m.h"
int main(int argc, char *argv[]){    QApplication aa(argc,argv);
    B w;    w.resize(300,200);    w.show();    return aa.exec();    }
```

运行结果及说明



测试方法:

点击左侧窗口标题栏(注意,一定要点击标题栏),然后把焦点移至按钮 G,然后点击左侧窗口客户区(工作区)的空白处,产生鼠标事件,调用 `a->activateWindow()` 函数,使右侧窗口成为活动窗口,然后移动焦点到按钮 C,再按下 `Alt+A` 键,此时程序输出 G 和 C,其中 G 是 `this->focusWidget()` 的结果,这是左侧窗口最后一个拥有焦点的部件, C 是 `Application::focusWidget()` 的结果,是当前活动的右侧窗口当前拥有焦点的部件。

三、焦点代理(Focus Proxy)

1、焦点代理(focus proxy):就是类似于标签和行编辑器之间的伙伴关系,当设置焦点代理后,焦点代理会实际获得并处理该部件的焦点,比如部件 2 是部件 1 的焦点代理,则当部件 1 获得焦点时,实际获得并处理焦点的是部件 2。

2、以下函数为与焦点代理有关的函数

①、`QWidget* QWidget::focusProxy() const;` //返回该部件的焦点代理,若没有代理则返回 0。

②、`void QWidget::setFocusProxy(QWidget* w);`

把此部件的焦点代理设置为部件 w,若 w 为 0,则重置该部件为没有焦点代理。若该部件有焦点代理,则 `setFocus()` 和 `hasFocus()` 函数将对焦点代理进行操作。

四、设置焦点及焦点策略

1、焦点策略:焦点策略描述了部件接受焦点的方式(比如可通过 `tab` 接受焦点,可通过鼠标滚轮接受焦点等)。

2、焦点改变的原因:是指部件是因为哪一个动作(按下 `tab`、鼠标点击或窗口失去焦点等)而使其获得或失去焦点

3、下面为需要用到函数和属性

①、`focusPolicy`: `Qt::FocusPolicy` //QWidget 类中的属性

访问函数: `Qt::FocusPolicy focusPolicy()const;` `void setFocusPolicy(Qt::FocusPolicy);`

获取和设置焦点策略,若该部件拥有焦点代理,则焦点策略会传递给焦点代理。

`Qt::FocusPolicy` 枚举描述了焦点策略,取值如下表

Qt::FocusPolicy 枚举(无标志)

作用:描述部件接受焦点的方式

成员	值	说明
<code>Qt::TabFocus</code>	<code>0x1</code>	通过 <code>tab</code> 键接受焦点
<code>Qt::ClickFocus</code>	<code>0x2</code>	通过鼠标点击接受焦点
<code>Qt::StrongFocus</code>	<code>TabFocus ClickFocus 0x8</code>	通过 <code>tab</code> 和鼠标点击接受焦点。
<code>Qt::WheelFocus</code>	<code>StrongFocus 0x4</code>	<code>Qt::StrongFocus +</code> 可使用鼠标滚轮接受焦点
<code>Qt::NoFocus</code>	<code>0</code>	不接受焦点

②、void **QWidget::setFocus**(); //槽

void **QWidget::setFocus**(Qt::FocusReason reason);

- 若此部件或其父部件之一是活动窗口，则为此部件(或其焦点代理)设置焦点。若窗口不是活动的，则当窗口 变为活动时，部件将获得焦点。
- 参数 **reason** 被传递到从该函数发送的任何焦点事件中，**reason** 用于解释是什么导致部件获得焦点。
- 无论焦点策略如何，该函数都会把焦点赋予给部件，但不会清除任何键盘捕获。
- 若该部件处于隐藏状态，则在显示该部件之前它不会接收到焦点。
- 注意：如果从可能被 **focusOutEvent()** 或 **focusInEvent()** 函数调用的函数中调用 **setFocus()**，可能会产生无限递归。
- 枚举 **Qt::FocusReason** 在这方面进行了描述，如下表

Qt::FocusReason 枚举(无标志)		
作用：描述部件焦点改变的原因		
成员	值	说明
Qt::MouseFocusReason	0	由于鼠标点击
Qt::TabFocusReason	1	由于按下 tab 键
Qt::BacktabFocusReason	2	由于产生了一个 Backtab，即 tab 的逆序动作，比如按下 Shift+tab 键时。
Qt::ActiveWindowFocusReason	3	由于窗口活动状态的改变
Qt::PopupFocusReason	4	程序打开/关闭一个弹出窗口，该弹出窗口捕获/释放了焦点。
Qt::ShortcutFocusReason	5	由于键入了好友的快捷键。
Qt::MenuBarFocusReason	6	菜单栏焦点
Qt::OtherFocusReason	7	其他原因

③、void **QWidget::clearFocus**(); 移出此部件拥有的焦点。

示例 5.25：焦点策略与焦点代理

```
//m.h 文件的内容
#ifndef M_H
#define M_H
#include<QtWidgets>
#include<QFocusEvent>
#include<QDebug>
class B:public QPushButton{    Q_OBJECT
public:    B(QString t="",QWidget* p=0):QPushButton(t,p){
public slots:
    void f(){ QObject *p=sender(); //QObject::sender() 的作用是返回指向发送信号的对象的指针。
        qDebug() <<p->objectName();    } //输出发送信号的对象的名称。
};
#endif // M_H

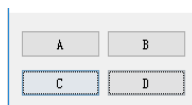
//m.cpp 文件的内容
#include "m.h"
int main(int argc, char *argv[]){    QApplication aa(argc,argv);
```

```

    QWidget w;    B *a,*b,*c,*d;    QGridLayout *pg=new QGridLayout;
a=new B("&A");    b=new B("&B");    c=new B("&C");    d=new B("&D");
pg->addWidget(a,0,0);    pg->addWidget(b,0,1);
pg->addWidget(c,1,0);    pg->addWidget(d,1,1);
w.setLayout(pg);
a->setObjectName("A");    //设置对象名
b->setObjectName("B");    c->setObjectName("C");    d->setObjectName("D");
//以下代码把 c 和 d 发出的信号关联到同一个槽
QObject::connect(c,&QPushButton::clicked,c,&B::f);
QObject::connect(d,&QPushButton::clicked,d,&B::f);
c->setFocusProxy(d);    //把 d 设置为 c 的焦点代理
b->setFocusPolicy(Qt::ClickFocus);    //设置 b 的焦点接受方式为只能使用鼠标点击
w.resize(300,200);    w.show();    return aa.exec(); }

```

运行结果及说明



测试方法:

使用 tab 移动焦点时, 按钮 B 不会接受到焦点(因为按钮 B 被设置为只能使用鼠标点击获取焦点), 按钮 C 也不会接收焦点(因为按钮 C 设置了焦点代理 D)

鼠标点击按钮 C, 焦点会自动移至按钮 D, 此时程序输出 C, 表示由按钮 C 发出了信号, 说明按钮 C 仍会响应鼠标点击事件。接着按下键盘的空格键, 输出 D, 表示按钮 D 发出了信号, 说明按钮 C 未响应键按下事件, 而是由按钮 C 的焦点代理 D 响应了键按下事件。

五、焦点事件

1、焦点事件产生的规则: 当焦点从一个部件移动到另一个部件时, 则各种焦点事件产生顺序如下:

- ①、首先 focusChangeEvent(焦点更改事件)被发送到之前拥有焦点的部件(如果有的话), 以告诉它, 它将失去焦点,
- ②、然后焦点被更改, 并把 focusOutEvent(焦点输出事件)发送到前一个焦点项, 也就是说前一个即将失去焦点的部件会产生两个事件(焦点更改和焦点输出事件)
- ③、最后把 focusInEvent 焦点输入事件发送到新项以告诉它, 它刚刚接收到焦点。
- ④、若焦点输入和焦点输出的部件相同, 则不会发生任何事情。

2、tab 键与文本编辑: 在文本编辑器中按下 tab 键是移动焦点还是产生制表符字符, Qt 的处理方式是把 Ctrl+tab 视为 tab, 把 Ctrl+Shift+Tab 视为 Shift+Tab, 但这不是一种完美的解决方案, 因此用户通常需要自行处理 tab 键, 其方法是重新实现 QWidget::event()函数, 以屏蔽 Qt 的默认实现方案。

3、下面为与焦点事件有关的函数

- ①、void **QWidget::focusOutEvent**(QFocusEvent* event); //虚拟的, 受保护的
当失去焦点时触发该事件, 默认实现为更新部件。

- ②、void **QWidget::focusInEvent**(QFocusEvent* event); //虚拟的，受保护的
当接收到焦点时触发该事件，默认实现为更新部件。
 - ③、bool **QWidget::event**(QEvent* event); //虚拟的，受保护的，该函数详见元对象系统章节
- 4、下面为 QEvent::Type 枚举定义的与焦点有关的事件类型。

- ①、**QEvent::FocusIn**: 部件获得焦点
- ②、**QEvent::FocusOut**: 部件失去焦点。
- ③、**QEvent::FocusAboutToChange**: 部件焦点即将被更改。

5、QFocusEvent 类:

该类有如下几个函数，其中枚举 Qt::FocusReason 参见 QWidget::setFocus() 函数

- ①、**QFocusEvent**(Type type, Qt::FocusReason reason = Qt::OtherFocusReason); //构造函数
参数 type 必须是 QEvent::FocusIn 或 Event::FocusOut
- ②、bool **gotFocus**() const; //若事件类型是 QEvent::FocusIn，则返回 true，否则返回 false
- ③、bool **lostFocus**() const; //若事件类型是 QEvent::FocusOut，则返回 true，否则返回 false
- ④、Qt::FocusReason **reason**() const; //返回产生该焦点事件的原因。

示例 5.26：焦点事件的使用(示例目的和界面见代码后的说明)

//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>
#include<QFocusEvent>
#include<QDebug>
class B:public QPushButton{    Q_OBJECT
public:    B(QString t="",QWidget* p=0):QPushButton(t,p) {}
public slots:
    void focusOutEvent(QFocusEvent* e) {
        qDebug()<<"Out="<<objectName();    //当失去焦点时输出去焦点的部件名称
        update();    }    //必须调用 update() 函数，否则焦点不会被更新
    void focusInEvent(QFocusEvent* e) {        qDebug()<<"In="<<objectName();    update();    }
    bool event(QEvent *e) {
        QFocusEvent *pe=(QFocusEvent*)e;
        if(pe->type()==QEvent::FocusAboutToChange) {    //处理焦点更改事件类型
            qDebug()<<"Changed="<<objectName();    //输出产生焦点更改事件的部件的名称
            qDebug()<<pe->reason();    //输出产生焦点更改事件的原因
            return true;    }    //焦点更改事件类型处理完毕，不需进一步处理。
        return QPushButton::event(e);    //其余未处理的事件交给父类处理(此步不能省略)
    }
};

class C:public QLineEdit{    Q_OBJECT    //子类化 QLineEdit 以使该部件能输出制表符。
public:    C(QString t="",QWidget* p=0):QLineEdit(t,p) {
public slots:
    bool event(QEvent *e) {    //重新实现该函数，以对按下 Tab 键时作专门的处理
        if(e->type()==QEvent::KeyPress) {
            QKeyEvent *pe=(QKeyEvent*)e;
            //以下语句，应先处理按下的 Ctrl+Tab 键，因为按下 Ctrl+Tab 键也是按下 Tab 键，若先处理 Tab
            键，则对 Ctrl+Tab 按键的处理将得不到执行。
            if(pe->modifiers()==Qt::ControlModifier&&pe->key()==Qt::Key_Tab)
```

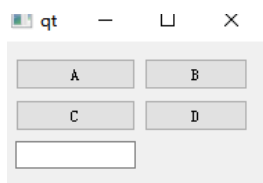
```

        { parentWidget()->setFocus(); //把焦点传递给该部件的父部件。
          return true;          } //对 Ctrl+Tab 的按键处理完毕。
    if (pe->key()==Qt::Key_Tab){ //若按下的是 Tab 键
        insert("\t"); //在 QLineEdit 中插入制表符。
        return true;}
    }
    return QLineEdit::event(e); //其余事件交给父部件处理，此步不能省略，否则其余事件将得不到
                                处理。*/
    }
};
#endif // M_H

//m.cpp 文件的内容
#include "m.h"
int main(int argc, char *argv[]){    QApplication aa(argc,argv);
    QWidget w;    B *a,*b,*c,*d;    C *pe;    QGridLayout *pg=new QGridLayout;
//初始化各部件
    pe=new C;    a=new B("&A");    b=new B("B");    c=new B("C");    d=new B("D");
//布局各子部件
    pg->addWidget(a,0,0);    pg->addWidget(b,0,1);    pg->addWidget(c,1,0);
    pg->addWidget(d,1,1);    pg->addWidget(pe,2,0);
    w.setLayout(pg);
//设置子部件的对象名
    a->setObjectName("A");    b->setObjectName("B");
    c->setObjectName("C");    d->setObjectName("D");
    w.resize(300,200);    w.show();    return aa.exec();    }

```

运行结果及说明



当焦点在各子部件间移动时，程序会输出相应的信息，比如当焦点从 B 移到 C 时，会输出
 Changed="B" //表示产生焦点更改事件的部件来自按钮 B

Qt::FocusReason(TabFocusReason) //表示产生焦点更改事件的原因是由于按下了 Tab 键。

Out="B" //表示产生焦点失去事件的是按钮 B(即按钮 B 失去了焦点)

In="C"; //表示产生焦点输入事件的是按钮 C(即按钮 C 获得了焦点)

QLineEdit(行编辑器)默认是不会输入制表符的，也就是说默认情况下按下 Tab 键会使 QLineEdit 失去焦点，本示例对 QLineEdit 对按下 Tab 键作了专门的处理以使其能输入制表符，同时按下 Ctrl+Tab 可把焦点传递给 QLineEdit 的父部件。

六、自定义焦点循环

- 1、要定义自己的焦点循环顺序，可以重新实现 QWidget::focusNextPrevChild()函数，该函数会被 Qt 内部调用(在按下 Tab 或 Shift+Tab 键时会调用该函数)，我们只需按照 Qt 内部规

则重新实现该函数即可，下面为该函数的原型，重新实现时的规则见示例代码。

bool **QWidget::focusNextPrevChild**(bool next); //虚拟的，受保护的

查找为 Tab 和 Shift+Tab 键提供焦点的部件，若找到则返回 true，若未找到则返回 false。

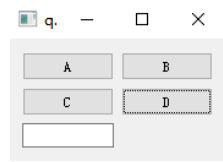
若参数 next 为 true，则向前搜索，若为 false 则向后搜索。重新实现该函数，可以控制所有子部件的焦点循环。

示例 5.27：实现自定义焦点循环顺序

本示例实现的规则是，按下 tab 键焦点逆序传递，即顺序为 D-C-B-A-行编辑器-D-C...，窗口外观如下图。

//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>
class B:public QWidget{    Q_OBJECT
public:    QPushButton *a,*b,*c,*d;    QLineEdit *pe;
        QList<QWidget*> t;    //使用列表 t 来保存窗口中的所有子部件
//构造函数开始
    B(QWidget* p=0):QWidget(p){
        QGridLayout *pg=new QGridLayout;
        pe=new QLineEdit;        a=new QPushButton("&A");
        b=new QPushButton("B");    c=new QPushButton("C");    d=new QPushButton("D");
        pg->addWidget(a,0,0);    pg->addWidget(b,0,1);    pg->addWidget(c,1,0);
        pg->addWidget(d,1,1);    pg->addWidget(pe,2,0);
        setLayout(pg);
        t=findChildren<QWidget*>();    //获取当前窗口中的所有子部件，并将其保存在列表 t 中。
    }    //构造函数结束
```



```
bool focusNextPrevChild(bool next){    //重新实现该虚函数
    //QList<QWidget*> t=findChildren<QWidget*>();    /*本示例不需要此步骤，把列表创建于此处的好处是可以实时跟踪窗口中子部件的更新，因为只要 Qt 内置函数调用该函数(在按下 tab 或 Shift+tab 时会调用该函数)，都会重新获取窗口中所有子部件的列表。*/
    QWidget* pw=focusWidget();    //获取该窗口中当前拥有焦点的部件。
    if(pw==0){        //若窗口当前没有部件拥有焦点，则把列表 t 中的第一个对象获得焦点。
        /*必须为 setFocus 函数传递一个 Qt::FocusReason 枚举值，而且值必须是 Qt::TabFocusReason 或 Qt::BacktabFocusReason，即焦点必须是由 tab 键产生的，否则，可能无法正确绘制焦点。*/
        t.at(0)->setFocus(Qt::TabFocusReason);}
    int i=t.indexOf(pw);    //获取当前拥有焦点的部件在列表 t 中的索引
    //如果拥有焦点的部件是列表中的第 1 个部件，则把焦点传递给列表中的最后一个部件
    if(i-1<0){ t.at(t.size()-1)->setFocus(Qt::TabFocusReason);}
    //否则把焦点传递给列表中的前一个部件。
    else {t.at(i-1)->setFocus(Qt::TabFocusReason);}
    return true;    /*必须返回 true，以满足 Qt 内部对此函数所要求的规则，即表示找到了为 Tab 和 Shift+Tab 键提供焦点的部件，Qt 内置的函数会根据该函数的返回值作一些其他设置，所以此处必须返回 true。*/
}
#endif // M_H
```

//m.cpp 文件的内容


```
#include "m.h"
int main(int argc, char *argv[]) {    QApplication aa(argc, argv);
    B w;    w.resize(300,200);    w.show();    return aa.exec(); }
```

七、QFocusFrame 类(焦点框, 自定义焦点框的外形)

1、QFocusFrame 类直接继承自 QWidget 类。



2、QFocusFrame 类用于向部件提供一个焦点框架的外形, 其实就是在部件周围的一个方框(见上图)。

3、QFocusFrame 会跟踪对部件的更改, 并自动调整自身大小将自己放在该部件周围。如果被监控的部件被删除, QFocusFrame 将其设置为零。

4、QFocusFrame 类比较简单, 只有几个函数, 只需使用 QFocusFrame::setWidget()函数, 就可把焦点框安装到部件上, 但这样安装的焦点框不会随着焦点移动, 因此要使焦点框能移动还需要重新实现 QWidget::focusNextPrevChild()函数自定义焦点循环顺序。通常不需要创建自己的焦点框。

5、QFocusFrame 类中的函数

- ①、**QFocusFrame**(QWidget* parent = Q_NULLPTR); //构造函数
- ②、void **setWidget**(QWidget* w); //把部件 w 的焦点框设置为此焦点框
- ③、QWidget* **widgit**() const; //返回部件拥有的焦点框。

示例 5.28: QFocusFrame 类(自定义焦点框的外形)

该示例的效果见示例后的说明

//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>
class B:public QWidget{    Q_OBJECT
public:
    QPushButton *a,*b,*c,*d;    QLineEdit *pe;        QList<QWidget*> t;
    QFocusFrame *pf;        //焦点框
//构造函数开始
    B(QWidget* p=0):QWidget(p) {
        QGridLayout *pg=new QGridLayout;
        pe=new QLineEdit;    a=new QPushButton("&A");    b=new QPushButton("B");
        c=new QPushButton("C");    d=new QPushButton("D");
        pg->addWidget(a, 0, 0);    pg->addWidget(b, 0, 1);    pg->addWidget(c, 1, 0);
        pg->addWidget(d, 1, 1);    pg->addWidget(pe, 2, 0);
        setLayout(pg);

        t=findChildren<QWidget*>();    //获取所有子部件
        pf=new QFocusFrame;        //初始化焦点框
//设置焦点框的颜色
        QPalette pt;        //创建调色板
        pt.setColor(QPalette::WindowText, QColor(111,1,1)); //焦点框需设置前景色
```

```

    pf->setPalette(pt);    //为 pf 安装调色板
    pf->setAutoFillBackground(true);    //自动填充背景
}    //构造函数结束

bool focusNextPrevChild(bool next){    //自定义焦点循环顺序的原理见上一小节
    //pf=new QFocusFrame;    /*不应这样创建焦点框，因为每调用一次 focusNextPrevChild() 函数
                                都会重新创建一个焦点框，本示例不需要那么多焦点框。*/
    //QFocusFrame pf;    /*使用局部对象也是不正确的，因为该函数调用结束后，局部对象就自动
                                结束其生命期，被销毁了，这样的话就看不到创建的焦点框了。*/
    QWidget* pw=focusWidget();
    if(pw==0) {                pf->setWidget(t.at(0));    //把部件 t.at(0) 的焦点框设置为 pf
                                t.at(0)->setFocus(Qt::TabFocusReason);    }
    int i=t.indexOf(pw);
    if(i-1<0) {                pf->setWidget(t.at(t.size()-1));
                                t.at(t.size()-1)->setFocus(Qt::TabFocusReason);    }
    else {                    pf->setWidget(t.at(i-1));    t.at(i-1)->setFocus(Qt::TabFocusReason);    }
    return true;    }
};

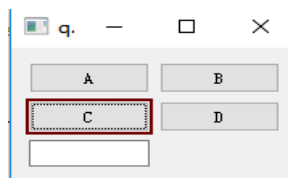
#endif // M_H

//m.cpp 文件的内容
#include "m.h"
int main(int argc, char *argv[]) {    QApplication aa(argc, argv);
    B w;    w.resize(300,200);    w.show();    return aa.exec(); }

```

运行结果及说明

本示例 Tab 顺序为逆序循环，按下 Tab 时自定义的红色焦点框，将随焦点的更改而移动，当调整主窗口大小更改子部件的大小时，焦点框会随之而改变，以使其始终位于部件的周围。注：使用鼠标等其他方式移动焦点不会改变焦点框的位置(因为本示例未处理这种情况)



作者：黄邦勇帅(原名：黄勇)

2018-4-12

本章公用枚举

Qt::TextElideMode 枚举(无标志)

作用：描述如何省略文本

成员	值	说明	成员	值	说明
QT::ElideLeft	0	省略号位于开头	QT::ElideMiddle	2	省略号位于中间
QT::ElideRight	1	省略号位于末尾	QT::ElideNone	3	无省略号

Qt::Corner 枚举(无标志)

作用：描述矩形中的角落位置

成员	值	说明	成员	值	说明
Qt::TopLeftCorner	0x00000	左上角	Qt::BottomLeftCorner	0x00002	左下角
Qt::TopRightCorner	0x00001	右上角	Qt::BottomRightCorner	0x00003	右下角

作者：黄邦勇帅(原名：黄勇)

2018-4-12

本文作者：黄邦勇帅(原名：黄勇)

本文要求读者已经非常熟悉 C++ 语法。若读者不熟悉 C++ 语法，推荐参阅《C++ 语法详解》(作者：黄勇)一书，电子工业出版社出版。

本文主要讲解了 Qt 的对话框，本文对 Qt 的对话框作了比较全面详细的深入讲解，详细讲解了各种对话框的每一个性质，并列举了详细的示例进行说明，同时本文也是非常方便、快捷的编写 Qt 程序的查阅资料，可方便的查阅到相关内容的原理，以及怎样使用该内容。本文内容由浅入深，易学易懂。

本文使用的是 windows 10 操作系统，Qt 版本为 Qt5.10.1，Qt Creator 的版本为 Qt Creator 4.5.1
本文内容完全属于个人见解与参考文献的作者无关，限于水平有限，其中难免有误解之处，望指出更正。

声明：禁止抄袭、复印、转载本文，本文作者拥有完全版权。

主要参考文献：

- 1、C++语法详解 黄勇 编著 电子工业出版社 2017 年 7 月
- 2、Qt Creator 快速入门(第 3 版) 霍亚飞 编著 北京航空航天大学出版社 2017 年 1 月
- 3、C++ GUI Qt4 编程(第 2 版) [加拿大] Jasmin Blanchette [英] Mark Summerfield 著 闫锋欣 曾泉人 张志强 译 电子工业出版社 2008 年 8 月
- 4、Qt 5 开发及实例 陆文周 主编 电子工业出版社 2014 年 1 月

第 6 章 Qt 对话框目录

6.1 QDialog 类(对话框)

6.1.1 对话框与窗口

6.1.2 模态与非模态对话框

6.1.3 对话框返回的信息

6.1.4 对话框与窗口的关闭和隐藏

6.2 QMessageBox 类(消息对话框)

6.3 QErrorMessage 类(错误消息对话框)

6.4 QColorDialog 类(颜色对话框)

6.5 QFontDialog 类(字体对话框)

6.6 QFileDialog 类(文件对话框)

6.6.1 文件对话框基础

6.6.2 QFileDialog 类中的属性

6.6.3 文件过滤器

6.6.4 QFileDialog 类中的函数

6.6.5 QFileDialog 类中的信号

6.7 QInputDialog 类(输入对话框)

6.7.1 输入对话框基础

6.7.2 QInputDialog 类中的属性

6.7.3 QInputDialog 类中的函数

6.7.4 QInputDialog 类中的信号

6.8 QProgressDialog 类(进度对话框)和 QProgressBar(进度条)

6.8.1 进度条原理

6.8.2 QProgressDialog 类(进度对话框)

6.8.3 QProgressBar 类(进度条)

6.9 QWizard 类(向导)和 QWizardPage 类(向导页)

6.9.1 向导基础

6.9.2 向导外观

6.9.3 向导中的按钮

6.9.4 向导中的页面

6.9.5 验证页面中的内容

6.9.6 各页面间的通信(字段)

6.9.7 实现非线性向导

6.9.8 QWizard 类中的属性

6.9.9 QWizard 类中的函数

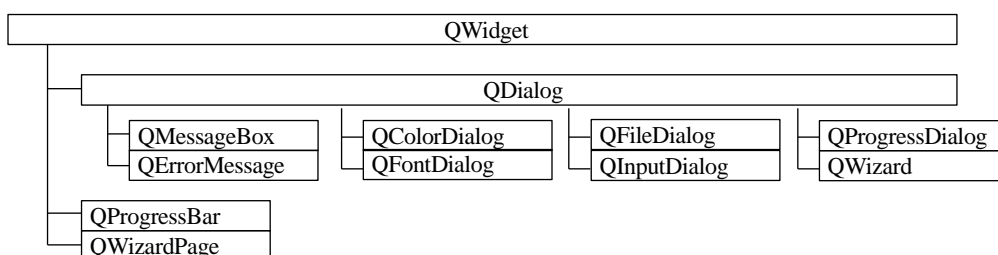
6.9.10 QWizardPage 类中的属性和函数

第 6 部分 Qt 对话框

注意：本程序都假设读者在 pro 文件中已添加了正确的 `QT+=widgets` 语句，文中不再重复累述添加此语句。

本文注重讲解原理，因此使用的是手写的 Qt 程序，对于使用 Qt 设计师快速设计 Qt 程序会在专门章节讲解。

本章讲解的类及继承关系如下图所示



6.1 QDialog 类(对话框)

QDialog 类继承自 QWidget 类，该类是对话框的基类。对话框是一个用于与用户短期交互的顶层窗口。QDialog 类的子类，是 Qt 提供的预定义标准对话框。

一、对话框与窗口

- 1、QDialog 继承自 QWidget，那么 QDialog 就应遵守 QWidget 的基本规则，QWidget 是描述 Qt 部件的基础类，该类对部件的类型进行了描述和规定，对话框需要遵循这些规则。
- 2、窗口标志：用于指定窗口的类型和窗口的外观，窗口类型描述了部件是什么类型的窗口，比如部件是对话框、窗口、弹出窗口、子窗口等，部件只能是一种窗口类型，窗口外观描述了窗口的外观，比如窗口是否有关闭按钮、是否有最大化最小化按钮、是否有标题栏等，窗口外观可以同时指定多个。
- 3、窗口标志使用 `Qt::WindowType` 枚举(标志为 `Qt::WindowFlags`)描述(其取值详见第 3 章)，该枚举值被分为两部件，即窗口类型部分和窗口提示部分，其中窗口类型位于低位字段，这些字段的任一组合都是另一种窗口类型，这间接限制了部件只能有一种窗口类型。窗口提示位于高位字段，它描述了部件的外观，窗口提示可以有多个，且仅影响顶级窗口。
- 4、窗口的特点：
 - ①、窗口默认不可见，需调用 `show()` 等函数才能使其可见，
 - ②、在大多数平台上，窗口都会列在桌面的任务栏中。
 - ③、若窗口具有父级，则根据窗口管理系统的不同，子窗口通常总是位于其各自的父窗口之上，并且没有自己的任务栏条目，通常是共享父部件的任务栏条目，比如，即使子窗口处于未激活状态而父窗口处于激活状态时，子窗口仍会位于父窗口之上。此规则对对话框同样适用。

④、，窗口标志包含 Qt::Window 的部件都是窗口

5、对话框的窗口标志默认为 Qt::Dialog，值为 0x0000 0002 | Qt::Window，可见默认情况下它是一个窗口，需使用 show()等函数才能显示出来。同理，改变 QDialog 的窗口类型，可以使 QDialog 不再是一个对话框。

6、对话框与窗口的明显区别在于对话框通常没有最大最小化按钮。

7、需用到的函数原型

- ①、`QDialog(QWidget* parent = Q_NULLPTR, Qt::WindowFlags f = Qt::WindowFlags());` //构造函数
- ②、`void QWidget::setParent(QWidget* parent);`
- ③、`void QWidget::setParent(QWidget* parent, Qt::WindowFlags f);`

更改该部件的父部件，并清除窗口标志或重置窗口标志，若新父部件和旧父部件相同，则此函数不执行任何操作，但是第 2 个函数若窗口标志不相同，会修改窗口标志。调用该函数作出更改后，即使该部件以前是可见的，也会变为不可见，因此需调用 show() 重新使该部件再次可见。

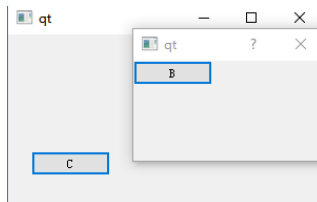
示例 6.1：窗口与非窗口的区别

```
#include<QtWidgets>
int main(int argc, char *argv[]) {    QApplication aa(argc, argv);
    QWidget w;    QPushButton *a,*b,*c;
    QDialog *pd1=new QDialog(&w);
    QDialog *pd2=new QDialog(&w);
    QDialog *pd3=new QDialog(&w, Qt::SubWindow);    //改变 pd3 的窗口标志，注意 SubWindow 不是窗口

    a=new QPushButton("&A",pd1);    b=new QPushButton("B",pd2);    c=new QPushButton("C",pd3);
    pd1->move(22,22);    pd2->move(22,77);    pd3->move(22,111);
    pd2->show();    //因为 pd2 是窗口，即使为其指定了父窗口也要明确的显示，该窗口才会可见。
    w.resize(300,200);    w.show();    return aa.exec(); }
```

运行结果及说明

对话框 pd3 中的按钮，因 pd3 被重新指定了窗口标志，使 pd3 不再是窗口，因此 pd3 作为子部件被添加到窗口 w 之中。



对话框 pd2 始终显示在其父窗口 w 之上，即使 pd2 失去了焦点，而 w 获得焦点，但 pd2 仍位于 w 之上(这是为窗口指定父窗口的特点)。

本示例，对话框 pd1 未显示，因为 pd1 是窗口，未使用 show()明确显示，所以 pd1 不可见。

示例 6.2：使用 setParent() 函数修改窗口标志

```
#include<QtWidgets>
#include<QDebug>
int main(int argc, char *argv[]) {    QApplication aa(argc, argv);
    QWidget w;    QPushButton *a,*b,*c;    QWidget *pw=new QWidget;
    QDialog *pd1=new QDialog(pw);    //父窗口指定为 pw
    QDialog *pd2=new QDialog(&w);
```

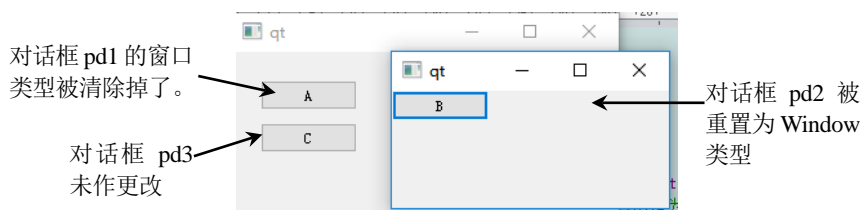
```

QDialog *pd3=new QDialog(&w, Qt::SubWindow);
a=new QPushButton("&A", pd1);    b=new QPushButton("B", pd2);    c=new QPushButton("C", pd3);

pd1->setParent(&w); /*重新指定父窗口，此函数会清除 pd1 的窗口标志，因此，pd1 不再是窗口，而是窗口 w 的子部件，此步骤会使 pd1 不可见，但 pd1 可随着 w 的显示而随之显示出来。*/
QDebug()<<pd1->>windowFlags(); //输出的值将只有窗口提示而无窗口类型的值。
pd1->move(22, 22);
pd2->resize(222, 111);
pd2->show(); //明确显示 pd2
pd2->setParent(&w, Qt::Window); /*修改 pd2 的窗口类型为 Window,即使在之前明确的调用了 show() 显示 pd2，但该步骤还是会使 pd2 不可见。*/
pd2->show(); //再次显示 pd2，若注释掉本行，则 pd2 将不可见。
pd3->setParent(&w); //因为新的父窗口与旧的父窗口相同，此步骤未执行任何操作。
QDebug()<<pd3->>windowFlags(); //从输出的值可以看到 pd3 的窗口类型仍是 SubWindow
pd3->move(22, 55);
w.resize(300, 200);
w.show(); //显示窗口，此步骤会使 pd1 显示出来。
return aa.exec(); }

```

运行结果及说明：



二、模态与非模态对话框

- 1、模态对话框：在同一个程序中，模态对话框会阻止用户与其他可见窗口的交互，模态对话框通常用于需要返回值的情形，比如获取用户按下的是 **Ok** 还是 **Cancel** 键等。
- 2、模态对话框有两种模式，**application modal** 应用程序模式(默认)和 **window modal** 窗口模式
 - ①、应用程序模式对话框：该模式会阻止用户与应用程序中的所有窗口的交互，直到完成与对话框的交互，并关闭该对话框，然后才能访问程序中的其他窗口。
 - ②、窗口模式对话框：只阻止与对话框相关联的窗口的访问，允许用户在程序中继续使用其他窗口。该模式会阻止与对话框的父窗口、所有祖先窗口及父窗口和祖先窗口的所有兄弟姐妹窗口的交互，除这些窗口外，该模式仍可与其他窗口交互。
 - ③、注意：模态对话框不会阻止与其子窗口的交互。
- 3、设置模态对话框的方法：
 - ①、使用 **QDialog::exec()** 函数，这是常用的方法。调用 **QDialog::exec()** 函数后不会立即返回，需结束对话框才会返回该函数。
 - ②、使用 **QDialog::setModal()** 或 **QWidget::setWindowModality()**，然后使用 **show()** 函数。与 **QDialog::exec()** 不同的是 **show()** 会立即把控制权返回给调用方。

4、需用到的函数和属性

- ①、virtual int **QDialog::exec()**; //虚拟的，槽
把对话框以模态对话框(默认为应用程序模式)的形式显示，该函数不会立即返回，需结束对话框才会返回该函数。此函数返回一个结果代码(见后文)的值，每次调用该函数时都会把结果代码重置为 0。
- ②、virtual void **QDialog::open()**; //虚拟的，槽
把对话框显示为窗口模式对话框，并立即返回。
- ③、**modal**: bool **该问函数**: bool isModal() const; void setModal(bool); //QDialog 类中的属性
设置此属性为 true，对话框将以应用程序模式显示，该函数通常与 show()一起使用，exe()函数会忽略此属性的值。
- ④、**modal**: const bool **该问函数**: bool isModal() const; //QWidget 类中的属性
部件是否是模态的，默认为 false。此属性只适用于 windows。
- ⑤、**windowModality**: Qt:: WindowModality //QWidget 类中的属性
该问函数: Qt::WindowModality windowModality() const;
void setWindowModality(Qt::WindowModality);

获取和设置窗口的模态类型，默认为 Qt::NonModal，当窗口可见时更改此属性不起作用，使用此属性需要先隐藏部件，然后再显示。此属性只适用于 windows。枚举 Qt::WindowModality 的取值如下

Qt::WindowModality 枚举(无标志)		
作用：描述模态窗口的类型		
成员	值	说明
Qt::NonModal	0	非模态窗口
Qt::WindowModal	1	窗口模式
Qt::ApplicationModal	2	应用程序模式

示例 6.3：exec() 函数与 show() 函数的区别

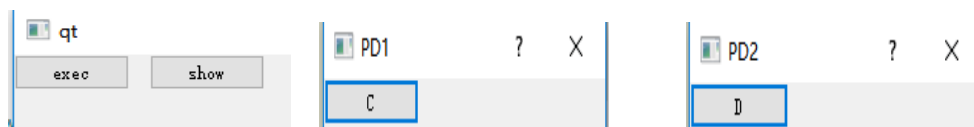
```
//m.h 文件的内容
#ifndef M_H
#define M_H
#include<QtWidgets>
class B:public QWidget{   Q_OBJECT
public:   QDialog *pd1,*pd2,*pd3;   QPushButton *a,*b,*c,*d;
        B(QWidget* p=0):QWidget(p) {
            pd1=new QDialog;   pd2=new QDialog;   pd1->resize(222,111);   pd2->resize(222,111);
            pd1->setWindowTitle("PD1");   pd2->setWindowTitle("PD2");
            a=new QPushButton("exec",this);   b=new QPushButton("show",this);   b->move(88,0);
            c=new QPushButton("C",pd1);   d=new QPushButton("D",pd2);
            QObject::connect(a,&QPushButton::clicked,this,&B::f);
            QObject::connect(b,&QPushButton::clicked,this,&B::f1);
        }
public slots:
        void f() {
            pd1->exec();   //调用该函数后需关闭对话框，才会返回该函数，然后才会执行之后的语句。
            pd2->show();   }
        void f1() {   pd1->show();   pd2->show();   }
};
```

```
#endif // M_H
```

//m.cpp 文件的内容

```
#include "m.h"
int main(int argc, char *argv[]) {    QApplication aa(argc, argv);
    B w;    w.resize(300,200);    w.show();    return aa.exec(); }
```

运行结果及说明



点击 exec 按钮会立即显示对话框 PD1，但 PD2 不会被显示，当把 PD1 关闭后，才会显示 PD2。而点击 show 按钮，则 PD1 和 PD2 会同时显示出来。

示例 6.4：应用程序模式与窗口模式的区别

//m.h 文件的内容

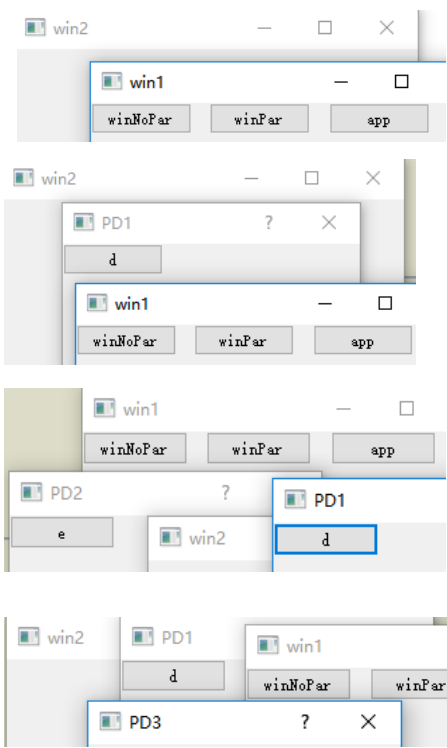
```
#ifndef M_H
#define M_H
#include<QtWidgets>
class B:public QWidget{    Q_OBJECT
public:    QDialog *pd1,*pd2,*pd3;    QPushButton *a,*b,*c,*d,*e;
    B(QWidget* p=0):QWidget(p) {    }
    void g() {
        pd1=new QDialog;        //窗口模式
        pd2=new QDialog(this);    //窗口模式
        pd3=new QDialog;        //应用程序模式
        pd1->setWindowTitle("PD1"); pd2->setWindowTitle("PD2");pd3->setWindowTitle("PD3");
        a=new QPushButton("winNoPar", this);
        b=new QPushButton("winPar", this);    b->move(88, 0);
        c=new QPushButton("app", this);    c->move(177, 0);
        d=new QPushButton("d", pd1);    e=new QPushButton("e", pd2);
        pd1->resize(222, 111);    pd2->resize(222, 111);    pd3->resize(222, 111);
        QObject::connect(a,&QPushButton::clicked, this, &B::f);
        QObject::connect(b,&QPushButton::clicked, this, &B::f1);
        QObject::connect(c,&QPushButton::clicked, this, &B::f2);
    }
public slots:
    void f() {    pd1->setWindowModality(Qt::WindowModal);    pd1->show();    }
    void f1() {    pd2->setWindowModality(Qt::WindowModal);    pd2->show();    }
    void f2() {    pd3->setWindowModality(Qt::ApplicationModal);    pd3->show();    }};
#endif // M_H
```

//m.cpp 文件的内容

```
#include "m.h"
int main(int argc, char *argv[]) {    QApplication aa(argc, argv);
    B w,w1;    w.g();    w.resize(300,200);    w.show(); }
```

```
w.setWindowTitle("win1");    w1.setWindowTitle("win2");
w.resize(300,200);    w1.show();    return aa.exec();    }
```

运行结果及说明



初次运行时两个窗口

点击按钮 winNoPar 会弹出对话框 PD1，该对话框模态类型为窗口模式，但无父窗口，因此该对话框阻止不了对窗口 win1 和 win2 的交互。

接着点击按钮 winPar 会弹出对话框 PD2，该对话框模态类型为窗口模式，父窗口为 win1，因此该对话框阻止不了对窗口 win2、对话框 PD1 的交互，但能阻止与窗口 win1 的交互。

关闭 PD2，点击按钮 app 会弹出对话框 PD3，该对话框模态类型为应用程序模式，无父窗口，该对话框会阻止与应用程序的所有窗口进行交互，因此 win1，win2，PD1 都不能在关闭 PD3 之前被激活。

三、对话框返回的信息

- 1、使用对话框时，通常需要知道用户点击了对话框中的哪个按钮，这就要求对话框能给程序返回一些信息。
- 2、Qt 把对话框返回的信息存储在“结果代码”中，但是这个变量对程序员是隐藏的，无法直接访问，可使用其他 QDialog 的成员函数间接的设置和读取结果代码的值，结果代码需要一个 int 类型的值，但通常应使用 QDialog::DialogCode 枚举值，该枚举成员如下表

QDialog::DialogCode 枚举(无标志)			
作用：用作结果代码的值			
成员	值	成员	值
QDialog::Accepted	1	QDialog::Rejected	0

- 4、需用到的函数、槽和信号
 - ①、int QDialog::result() const;

返回对话框的结果代码，通常为 Accepted 或 Rejected。注意：在 QMessageBox 实例上调用此函数时，返回的值是 QMessageBox::StandardButton 枚举的值。若对话框是使用 Qt::WA_DeleteOnClose 属性构造的，则不要调用此函数，即，对话框被删除后，不要调用此函数。

②、void **QDialog::setResult**(int i);

把对话框的结果代码设置为 i，建议使用 QDialog::DialogCode 枚举的值之一。

③、virtual void **QDialog::accept**(); //虚拟的，槽

隐藏对话框，并把结果代码设置为 Accepted，该函数其实质是使用参数 Accepted 调用 done(int)函数。

④、void **QDialog::reject**(); //虚拟的，槽

隐藏对话框，并把结果代码设置为 Rejected。该函数其实质是使用参数 Rejected 调用 done(int)函数。

⑤、void **QDialog::done**(int r); //虚拟的，槽

隐藏对话框，并把结果代码设置为 r，若对话框是使用 exec()显示的，则 exec()返回 r。

⑥、void **QDialog::accepted**(); //信号

当对话框被用户接受，或调用 accept()函数，或通过参数 QDialog::Accepted 调用 done()函数时，发送此信号。注意：当调用 hide()或 setVisible(false)隐藏对话框时，不会发送此信号，包括在对话框可见时删除对话框。

⑦、void **QDialog::rejected**(); //信号

当对话框被用户拒绝，或调用 reject()函数，或通过参数 QDialog::Rejected 调用 done()函数时，发送此信号。注意：当调用 hide()或 setVisible(false)隐藏对话框时，不会发送此信号，包括在对话框可见时删除对话框。

⑧、void **QDialog::finished**(int result); //信号

当对话框被用户、或调用 reject()、accept()、done()设置了对话框的结果代码时，发送此信号。注意：当调用 hide()或 setVisible(false)隐藏对话框时，不会发送此信号，包括在对话框可见时删除对话框。

5、通常把默认按钮连接到 accept()槽，把 Cancel 按钮连接到 reject()槽。或者使用 Accepted 或 Rejected 调用 done()槽。

示例 6.5: accept() 槽和 reject() 槽函数的使用

//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>
#include <iostream>
using namespace std;
class B:public QWidget{    Q_OBJECT
public:    QDialog *pd1,*pd2;    QPushButton *pb,*pb1,*pb2,*pb3,*pb4,*pb5;
    B(QWidget* p=0):QWidget(p){
        pd1=new QDialog(this);                pd2=new QDialog(this);
        pd1->setWindowTitle("PD1");            pd2->setWindowTitle("PD2");
        pb=new QPushButton("dialog1",this);    pb1=new QPushButton("dialog2",this);
        pb2=new QPushButton("Yes",pd1);        pb3=new QPushButton("No",pd1);
    }
};
```

```

        pb4=new QPushButton("Yes",pd2);          pb5=new QPushButton("No",pd2);
        pb1->move(88,0);          pb3->move(88,0);          pb5->move(88,0);
        pd1->resize(222,111);    pd2->resize(222,111);
        connect(pb,&QPushButton::clicked,this,&B::f);
        connect(pb1,&QPushButton::clicked,this,&B::f1);
        connect(pb2,&QPushButton::clicked,pd1,&QDialog::accept);
        connect(pb3,&QPushButton::clicked,pd1,&QDialog::reject);
        connect(pb4,&QPushButton::clicked,pd2,&QDialog::accept);
        connect(pb5,&QPushButton::clicked,pd2,&QDialog::reject);    }    //构造函数结束

public slots:
    void f() {
        pd1->setWindowModality(Qt::ApplicationModal);
        pd1->show();
        /*注意：调用 show() 之后会立即返回调用处继续执行之后的语句，因此，result() 返回的是上次
        的结果代码值。*/
        if(pd1->result()==1)    cout<<"OK"<<endl;
        if(pd1->result()==0) cout<<"Cancel"<<endl;    }//f 结束
    void f1() {
        int i=pd2->exec();
        /*因为 exec() 需等到用户关闭对话框才会返回调用处继续执行之后的语句，所以 result() 返回的是当
        前用户操作之后的结果代码值，这也是对话框通常使用 exec() 显示的原因。*/
        if(pd2->result()==1)cout<<"OK"<<endl;
        if(i==0) cout<<"Cancel"<<endl;    }    };    //类 B 结束
#endif // M_H

```

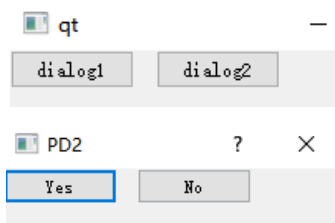
//m.cpp 文件的内容

```

#include "m.h"
int main(int argc, char *argv[]){    QApplication aa(argc,argv);
    B w;    w.resize(300,200);    w.show();    return aa.exec(); }

```

运行结果及说明



点击 dialog2 弹出下图所示对话框 PD2, 点击 PD2 中的 Yes 按钮, 关闭(此处是隐藏并未删除)对话框, 然后程序输出 OK, 若点击的是 No 按钮, 则关闭对话框, 输出 Cancel。
若点击的是 dialog1 则会弹出与 PD2 类似的对话框 PD1, 此时立即输出 Cancel, 然后点击 PD1 中的 Yes 按, 对话框会关闭, 但程序不会输出值, 再次点击主窗口的 dialog1 按钮, 程序立即输出 Ok(这是上次的结果代码值)。

四、对话框与窗口的关闭和隐藏

1、为方便讲解，本文对以下概念作一区别

- 删除：是指窗口被销毁，也就是说窗口不存在了。比如窗口使用 new 创建的，则表示窗口被 delete 了，被销毁的窗口不能被再次使用，否则会发生内存错误。
- 隐藏：是指窗口不可见，但窗口并未被销毁，使用 show()等函数，可以让该窗口再

次可见。

- 关闭：是指窗口不可见，但窗口有可能是被删除了，也有可能是被隐藏了，这要视情况而定。
- 窗口被删除时，会同时删除其子对象，而隐藏则不会。

2、一个(应用)程序通常拥有多个窗口，关闭(或删除)一个窗口，并一定会使程序终止，Qt 中关闭窗口使用 `QWidget::close()`槽函数，终止程序使用的是 `QCoreApplication::quit()`静态槽函数或 `QCoreApplication::exit()`静态函数

3、与关闭部件和终止程序有关的属性如下(即 `Qt::WidgetAttribute` 枚举的取值)，属性可使用 `QWidget::setAttribute()`函数进行设置和清除。

①、`Qt::WA_DeleteOnClose` 属性：表示当部件接受到 `QCloseEvent` 事件时，是否让 Qt 删除部件。若该属性为 `true`，则删除部件，否则部件只是隐藏。注意：设置了该属性的部件需要使用 `new` 创建，否则会产生内存错误。

②、`Qt::WA_QuitOnClose` 属性：表示当拥有该属性的最后一个部件接受到 `QCloseEvent` 事件时，让 Qt 终止应用程序。默认情况下，所有 `Qt::Window` 类型的部件都具有该属性。

4、`QWidget` 类中与关闭窗口有关的函数如下：

①、`bool QWidget::close();` //槽

关闭(即删除或隐藏)部件，若部件关闭成功，则返回 `true`，否则返回 `false`。

②、`virtual void QWidget::closeEvent(QCloseEvent* e);` //虚拟的，受保护的

这是 `QCloseEvent` 事件的处理函数，默认情况下，该函数接受 `QCloseEvent` 事件。该函数通常被重新实现，以确定用户是否需要关闭窗口。

5、`close()`函数的执行过程如下：

①、首先，向该部件发送 `QCloseEvent` 事件(不管部件是否可见)

②、然后判断部件是否接受 `QCloseEvent` 事件

- 若部件接受该事件(默认值)，则继续下一步操作。

- 若部件忽略该事件，则取消关闭操作，结束后续的操作。其中最重要的是，不会对 `Qt::WA_DeleteOnClose` 属性进行判断，此时该属性不起作用。

③、接着判断部件是否被隐藏了，若未被隐藏，则隐藏，若已被隐藏，则什么也不做。然后继续下一步。

④、再接着判断部件的 `Qt::WA_QuitOnClose` 属性，当具有 `Qt::WA_QuitOnClose` 属性的最后一个可见主窗口(即没有父窗口的窗口)被关闭时，会发送 `QApplication::lastWindowClosed()`信号。

⑤、最后判断部件的 `Qt::WA_DeleteOnClose` 属性，若该属性为 `true`，则删除该部件，否则什么也不做。至此整个过程结束。

⑥、总结：从以上过程可见，若部件接受 `QCloseEvent` 事件，且设置了 `Qt::WA_DeleteOnClose` 属性，则会删除该部件，若未设置该属性则只会隐藏该部件。若部件忽略 `QCloseEvent` 事件，则直接取消对该部件的关闭操作，该部件既不会被隐藏也不会被删除。由此可见对 `QCloseEvent` 事件接受还是忽略决定着对窗口关闭的处理方式，同时对该事件的处理方式与其他事件也是不同的，`QCloseEvent::ignore()`表示取消关闭操作(也就是说 `QCloseEvent` 事件不会被传递给父对象)，而 `QCloseEvent::accept()`则表示让 Qt 继续关闭操作。

6、QCloseEvent 事件的发送时机如下：

从窗口菜单选择“关闭”，单击标题栏上的 X 按钮，调用 QWidget::close()函数时。

7、下面为用于终止程序的函数原型：

- ①、static void **QCoreApplication::quit()**; //静态的，槽
退出程序，并返回代码 0(成功)，此函数与调用 QCoreApplication::exit(0)等同。该槽函数通常与信号连接使用，比如

```
QPushButton *p = new QPushButton("quit");
```

```
QObject::connect( p ,& QPushButton::clicked, &app, &QCoreApplication::quit);
```

- ②、static void **QCoreApplication::exit**(int returnCode = 0); //静态的
使用 returnCode 退出程序，通常 returnCode 为 0，表示成功，任何非零值都表示错误。
该函数会使程序离开主事件循环，并返回到调用 QCoreApplication::exec()处，exec()函数会返回 returnCode 的值，若事件循环未运行，则该函数什么都不做。

8、可使用以下方式终止程序

- ①、直接调用 quit()或 exit()函数
②、最后一个具有 Qt::WA_QuitOnClose 属性的主窗口关闭时，终止程序，若没有这样的主窗口，即使所有的窗口都关闭了程序也不会结束。

9、对话框的关闭过程：对话框的 reject()、accept()、done()函数，与 QWidget::close()函数相同，唯一的区别是对话框不会发送 QCloseEvent 事件，因此不能通过 QCloseEvent 事件来阻止对话框的关闭。注意：此规则仅限于上述 3 个函数，比如点击对话框窗口的 X 按钮或右击标题栏选择“关闭”时，仍会发送 QCloseEvent 事件。若用户在对话框中按下 Esc 键，会调用 QDialog::reject()。为了修改对话框的关闭行为，可以重新实现 accept()、reject()或 done()函数。

10、删除 QObject 对象时，会发送 destroyed()信号，该信号原型如下：

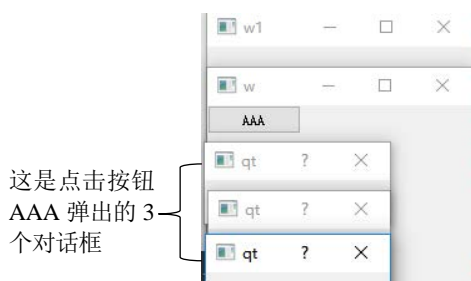
```
void QObject::destroyed(QObject* obj = Q_NULLPTR); //信号。
```

当对象 obj 被销毁之前发送，且不能被阻止，发送该信号后，对象 obj 的孩子都会被立即销毁。该信号需配合 Qt::WA_DeleteOnClose 属性使用，当 Qt::WA_DeleteOnClose 属性为 true 的对象被删除时，会发送该信号，直接终止程序的运行，是不会发送该信号的。

示例 6.6：关闭一个窗口就结束程序(理解 Qt::WA_QuitOnClose 属性)

```
#include<QtWidgets>
int main(int argc, char *argv[]) {    QApplication aa(argc, argv);
    QWidget w,w1;    QDialog pd;    QDialog pd1;    QDialog pd2;
    QPushButton *pb=new QPushButton("AAA",&w);
    w.setWindowTitle("w");    w1.setWindowTitle("w1");
    //除窗口 w1 外，窗口 w 和对话框的 Qt::WA_QuitOnClose 属性都设置为 0。
    w.setAttribute(Qt::WA_QuitOnClose,0);    pd.setAttribute(Qt::WA_QuitOnClose,0);
    pd1.setAttribute(Qt::WA_QuitOnClose,0);    pd2.setAttribute(Qt::WA_QuitOnClose,0);
    //点击按钮 pb，弹出 3 个对话框。
    QObject::connect(pb,&QPushButton::clicked,&pd,&QDialog::show);
    QObject::connect(pb,&QPushButton::clicked,&pd1,&QDialog::show);
    QObject::connect(pb,&QPushButton::clicked,&pd2,&QDialog::show);
    w.resize(300,200);    w.show();    w1.resize(300,200);    w1.show();    return aa.exec();}
```


运行结果及说明



所有窗口(含对话框窗口)都不存在父子关系,因为本例只有 w1 的 Qt::Wa_QuitOnClose 属性为 true,因此关闭窗口 w1(点击右上角的 X 按钮),会使程序终止运行,关闭其他窗口都不会使程序终止,只会使该窗口被隐藏。若所有窗口的该属性都是 false,则即使把所有窗口都关闭了,程序也不会终止,此时需点击下图的红色正方形按钮来结束程序。



示例 6.7: 理解 Qt::WA_DeleteOnClose 属性和 destroyed 信号

//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>
#include<QDebug>
class A:public QObject{Q_OBJECT
public slots: void f(QObject *p){QDebug()<<"del="<<p->objectName();} //输出被删除对象的名称
};
#endif//M_H
```

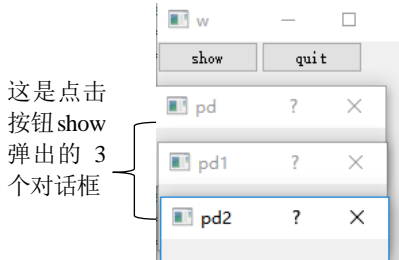
//m.cpp 文件的内容

```
#include "m.h"
int main(int argc, char *argv[]){ QApplication aa(argc,argv);
    QWidget w;    QDialog pd;    QDialog *pd1=new QDialog;    QDialog *pd2=new QDialog;
    QPushButton *pb=new QPushButton("show",&w);
    QPushButton *pbl=new QPushButton("quit",&w); pbl->move(77,0);
//设置窗口标题
    w.setWindowTitle("w");          pd.setWindowTitle("pd");
    pd1->setTitle("pd1");          pd2->setTitle("pd2");
//设置 Qt::WA_DeleteOnClose 属性
    pd.setAttribute(Qt::WA_DeleteOnClose,1);
    pd1->setAttribute(Qt::WA_DeleteOnClose,1);
//设置对象名
    w.setObjectName("w");    pd.setObjectName("pd");    pd1->setName("pd1");
    pd2->setName("pd2");
    A ma;
//单击按钮 pb 弹出 3 个对话框
    QObject::connect(pb,&QPushButton::clicked,&pd,&QDialog::show);
    QObject::connect(pb,&QPushButton::clicked,pd1,&QDialog::show);
    QObject::connect(pb,&QPushButton::clicked,pd2,&QDialog::show);
//单击按钮 pbl 直接终止程序
    QObject::connect(pbl,&QPushButton::clicked,&aa,&QApplication::quit);
//把 destroyed 信号连接到槽 f
    QObject::connect(&pd,&QDialog::destroyed,&ma,&A::f);
    QObject::connect(pd1,&QDialog::destroyed,&ma,&A::f);
```



```
QObject::connect(pd2, &QDialog::destroyed, &ma, &A::f);
w.resize(300, 200);    w.show();    return aa.exec(); }
```

运行结果及说明



测试步骤

- 1、点击 show 弹出对话框之后，若关闭(点击右上角的 X 按钮)pd，则程序可能会崩溃(因为pd未使用new创建)，关闭pd1则会销毁pd1，程序此时输出"del=pd1"，关闭pd2，则只会隐藏pd2，因为各窗口之间不存在父子关系，因此本例需点击quit才能正常结束程序(因为最终在关闭窗口pd时，可能会崩溃)。
- 2、重新运行程序，点击 show 按钮，然后关闭pd1和pd2(注意，不要关闭pd)，然后再点击 show 按钮，对话框pd2会被再次显示，但pd1未被显示(因为被销毁了)。
- 3、再次重新运行程序，点击 show，弹出3个对话框，然后点击quit直接退出程序，程序无任何输出，可见，信号destroyed未被发射。

示例 6.8：关闭窗口时询问用户(QCloseEvent 事件的应用)

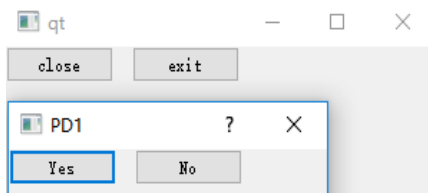
```
#ifndef M_H
#define M_H
#include<QtWidgets>
class B:public QWidget{    Q_OBJECT
public:    QDialog *pd1;    QPushButton *pb,*pb1,*pb2,*pb3;
    B(QWidget* p=0):QWidget(p){
        pd1=new QDialog(this);    pd1->setWindowTitle("PD1");    pd1->resize(222,111);
        pb=new QPushButton("close",this);
        pb1=new QPushButton("exit",this);    pb1->move(88,0);
        pb2=new QPushButton("Yes",pd1);
        pb3=new QPushButton("No",pd1);    pb3->move(88,0);
        connect(pb,&QPushButton::clicked,this,&B::f);
        connect(pb1,&QPushButton::clicked,this,&B::f1);
        connect(pb2,&QPushButton::clicked,pd1,&QDialog::accept);
        connect(pb3,&QPushButton::clicked,pd1,&QDialog::reject);
    }
    void f(){    close();    }    //调用 close() 函数关闭窗口，该函数会发送 QCloseEvent 事件。
    void f1(){    QApplication::quit();    }    //注意：quit() 函数不会产生 QCloseEvent 事件。
    void closeEvent(QCloseEvent* e){    //重新实现 closeEvent 函数，以处理 QCloseEvent 事件
        int i = pd1->exec();    //弹出对话框 pd1
        //根据用户在对话框中选择的按钮，决定怎样关闭窗口
        if(i==1)e->accept();    //接受 QCloseEvent 事件，继续关闭窗口。
        if(i==0)e->ignore();    //忽略 QCloseEvent 事件，阻止关闭窗口。
    }
};
#endif // M_H
```

//m.cpp 文件的内容

```
#include "m.h"
int main(int argc, char *argv[]){    QApplication aa(argc,argv);
```

```
B w;    w.resize(300,200);    w.show();    return aa.exec(); }
```

运行结果及说明



当用户点击窗口 qt 右上角的 X 按钮或点击 close 按钮时，会弹出对话框 PD1，以确定用户是否需要关闭窗口，当用户点击 PD1 中的 Yes 按钮时，会关闭窗口 qt，若点击 No 按钮，则 qt 不会被关闭。若用户直接点击 qt 中的 exit 按钮，则程序直接终止，不会弹出对话框。

示例 6.9：对话框与 QCloseEvent 事件

//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>
#include <iostream>
using namespace std;
class D:public QDialog{    Q_OBJECT
public:    D(QWidget* p=0):QDialog(p) {    }
    void closeEvent(QCloseEvent* e){    cout<<"D"<<endl;    setResult(2);    }    };
class B:public QWidget{    Q_OBJECT
public:    D *pd1;    QPushButton *pb,*pb1,*pb2,*pb3,*pb4,*pb5;
    B(QWidget* p=0):QWidget(p) {
        pd1=new D(this);    pd1->setWindowTitle("PD1");    pd1->resize(222,111);

        pb=new QPushButton("exec",this);
        pb1=new QPushButton("show",this);    pb1->move(88,0);
        pb2=new QPushButton("Yes",pd1);
        pb3=new QPushButton("No",pd1);    pb3->move(88,0);
        pb4=new QPushButton("result",pd1);    pb4->move(0,33);
        connect(pb,&QPushButton::clicked,this,&B::f);
        connect(pb1,&QPushButton::clicked,this,&B::f1);
        connect(pb2,&QPushButton::clicked,pd1,&QDialog::accept);
        connect(pb3,&QPushButton::clicked,pd1,&QDialog::reject);
        connect(pb4,&QPushButton::clicked,this,&B::f2);
    }

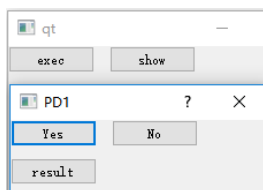
    void f() {    int i=pd1->exec();    cout<<"exec="<<i<<endl;    }
    void f1() {    pd1->show();    cout<<"show="<<pd1->result()<<endl;    }
    void f2() {    cout<<"result="<<pd1->result()<<endl;    }

};
#endif // M_H
```

//m.cpp 文件的内容

```
#include "m.h"
int main(int argc, char *argv[]){    QApplication aa(argc,argv);
    B w;    w.resize(300,200);    w.show();    return aa.exec(); }
```

运行结果及说明



1、验证 QCloseEvent 事件未触发

点击 exec 以使用 exec()函数显示对话框 PD1

点击 Yes 按钮，程序输出 exec=1，可见 QDialog::accept()槽函数未触发 QCloseEvent 事件。

2、验证 exec()函数重置结果代码为 0

再次点击 exec 按钮，显示对话框 PD1，然后点击 result 按钮，程序输出 result=0，可见 exec()函数把对话框的结果代码重置为 0 了。

3、验证 QCloseEvent 事件被触发

然后点击 PD1 右上角的 X 关闭对话框，此时程序输出 D，可见此时触发了 QCloseEvent 事件，此时 PD1 的结果代码为 2。

4、验证 show()不会重置结果代码为 0

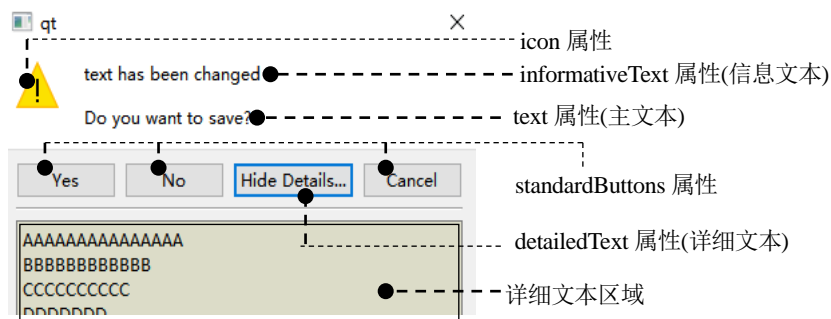
然后点击 show 按钮，以使用 show()函数显示对话框 PD1，

此时输出 show=2(注意 show()函数调用后会立即返回); 然后点击 result，输出 result=2;可见 PD1 的结果代码此时为 2。

然后点击 yes 按钮关闭对话框，再点击 show 按钮显示对话框，此时输出 show=1;然后点击 result，输出 result=1;可见 show()函数未把结果代码重置为 0。

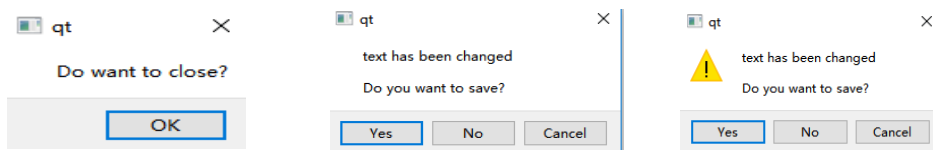
6.2 QMessageBox 类(消息对话框)

- 1、消息对话框用于向用户显示一些信息，或者接收用户的答案。
- 2、QMessageBox 类继承自 QDialog，是 Qt 内部预定义的一种对话框。
- 3、消息对话框有很多种形式，下图是一个综合的消息对话框，其中描述了每个位置 Qt 使用什么属性进行描述。



textFormat 属性：用于描述 text 属性或 informativeText 属性是否启用富文本

- 4、下面是几种其他常见形式的消息对话框样式



- 5、通常使用以下两种方式创建消息对话框

- 设置属性的方式：

```
QMessageBox pm;    //创建对象
pm.setText("XXX"); //设置文本(属性之一)
pm.exec();         //显示对话框
```

- 使用静态函数的方式，

```
QMessageBox::question(&w, "AAA", "BBB"); //使用静态函数创建一个询问消息对话框并显示，其标题为 AAA，文本为 BBB
```

使用静态函数创建的消息对话框不需要明确的显示。

- 6、向消息对话框中添加按钮：方法如下：

- 把自定义按钮添加到消息对话框中：使用 QMessageBox::addButton()函数把按钮添加到消息对话框，添加时只需为按钮指定一个 ButtonRole(按钮角色)即可。
- 添加预定义的标准按钮：使用 standardButtons 属性直接指定 Qt 预定义标准按钮。

- ①、按钮角色：使用 QMessageBox::ButtonRole 枚举描述，见下表

QMessageBox::ButtonRole 枚举(无标志)		
用于描述消息对话框中按钮的角色或作用(Role)。		
成员	值	说明
QMessageBox::InvalidRole	-1	该按钮无效
QMessageBox::AcceptRole	0	对话框被接受(如 OK 按钮)
QMessageBox::RejectRole	1	对话框被拒绝(如 cancel 按钮)
QMessageBox::DestructiveRole	2	点击该按钮会产生破坏性更改(如放弃当前的更改) 并关闭对话框
QMessageBox::ActionRole	3	单击该按钮会导致对话框中的元素进行更改。
QMessageBox::HelpRole	4	用于请求帮助(如帮助按钮)
QMessageBox::YesRole	5	“是”按钮
QMessageBox::NoRole	6	“否”按钮
QMessageBox::ApplyRole	8	该按钮应用当前的更改，比如“应用”按钮。
QMessageBox::ResetRole	7	重置为默认值。

- ②、Qt 预定义的标准按钮见下表

QMessageBox::StandardButton 枚举		
标志：QMessageBox::StandardButtons		
描述了 Qt 内预定义的标准按钮，这些按钮都由 ButtonRole 枚举指定了作用(Role)		
成员	值	说明
QMessageBox::Ok	0x0000 0400	“OK 按钮”，由 AcceptRole 定义
QMessageBox::Open	0x0000 2000	“Open 按钮”，由 AcceptRole 定义
QMessageBox::Save	0x0000 0800	“Save 按钮”，由 AcceptRole 定义
QMessageBox::Cancel	0x0040 0000	“Cancel 按钮”，由 RejectRole 定义
QMessageBox::Close	0x0020 0000	“Close 按钮”，由 RejectRole 定义
QMessageBox::Discard	0x0080 0000	“Discar(丢弃)按钮”，使用 DestructiveRole 定义，取决于平台
QMessageBox::Apply	0x0200 0000	“Apply 按钮”，由 ApplyRole 定义
QMessageBox::Reset	0x0400 0000	“Reset 按钮”，由 ResetRole 定义
QMessageBox::RestoreDefaults	0x0800 0000	“Restore Defaults 按钮”，由 ResetRole 定义
QMessageBox::Help	0x0100 0000	“Help 按钮”，由 HelpRole 定义
QMessageBox::SaveAll	0x0000 1000	“Save All 按钮”，由 AcceptRole 定义
QMessageBox::Yes	0x0000 4000	“Yes 按钮”，由 YesRole 定义
QMessageBox::YesToAll	0x0000 8000	“Yes To All 按钮”，由 YesRole 定义
QMessageBox::No	0x0001 0000	“No 按钮”，由 NoRole 定义
QMessageBox::NoToAll	0x0002 0000	“No To All 按钮”，由 NoRole 定义
QMessageBox::Abort	0x0004 0000	“Abort 按钮”，由 RejectRole 定义
QMessageBox::Retry	0x0008 0000	“Retry 按钮”，由 AcceptRole 定义
QMessageBox::Ignore	0x0010 0000	“Ignore 按钮”，由 AcceptRole 定义
QMessageBox::NoButton	0x0000 0000	无效按钮







- 7、注意：QDialog::result()函数返回的是 StandardButtons 值，而不再是 QDialog::DialogCode 值。

8、EscapeButton 按钮，是指当按下 Esc 键时激活的按钮。若未使用 setEscapeButton()函数设置 EscapeButton 按钮，则默认规则如下：

- 若只有一个按钮，则设置该按钮为 EscapeButton
- 若有“Cancle”按钮，则将其设置为 EscapeButton
- 仅适用于 Mac，若只有一个按钮具有角色 QMessageBox::RejectRole，则将其设置为 EscapeButton
- 若无法检测到 EscapeButton 按钮，则按下 Esc 键不起作用。

9、QMessageBox 类中的属性

- ①、**text**: QString **访问函数**: QString text() const; void setText(const QString &);
设置或获取消息对话框的主文本，默认为空字符串，文本被解释为富文本还是纯文本由 textFormat 属性设置，其默认格式为 Qt::AutoText(自动检测)。
- ②、**informativeText**: QString
访问函数: QString informativeText()const; void setInformativeText(const QString &);
设置或获取消息对话框的信息文本，信息文本可用于向用户提供更多的信息，默认为空字符串。在 Mac 上，信息文本会以小系统字体显示，在其他平台上，只是简单的附加到现有的文本中。文本被解释为富文本还是纯文本由 textFormat 属性设置。
- ③、**detailedText**: QString //详细文本
访问函数: QString detailedText() const; void setDetailedText(const QString &text);
设置和获取详细文本区域中显示的文本，该属性总是被解释为纯文本，默认为空字符串。详细文本用于向用户提供更详细的信息。设置该属性后，对话框会多出一个"Show Details..."按钮，点击该按钮会在对话框下面显示详细的文字描述。
- ④、**textFormat**: Qt::TextFormat
访问函数: Qt::TextFormat textFormat() const; void setTextFormat(Qt::TextFormat);
设置和获取消息对话框的文本是否可以以富文本的形式显示，该属性会影响 text 和 informativeText 属性。其中 Qt::TextFormat 枚举的取值为：Qt::PlainText(纯文本)、Qt::RichText(富文本)，Qt::AutoText(自动检测，默认值)。
- ⑤、**icon**: Icon **访问函数**: Icon icon()const; void setIcon(Icon);
设置和获取消息对话框的预定义图标，图标所使用的像素图(图片)取决于当前的 GUI 样式，该属性用于设置标准的 Qt 预定义图标，对于自定义的图标需使用 iconPixmap 属性设置。Qt 的预定义图标由 QMessageBox::Icon 枚举定义，见下表：

QMessageBox::Icon 枚举(无标志)		
作用：Qt 的预定义消息对话框的图标		
成员	值	说明
QMessageBox::NoIcon	0	无图标(默认值)
QMessageBox::Question	4	询问图标。  , 
QMessageBox::Information	1	消息图标。  , 
QMessageBox::Warning	2	警告图标。 
QMessageBox::Critical	3	严重警告图标。 

⑥、**iconPixmap**: QPixmap

访问函数: QPixmap iconPixmap() const; void setIconPixmap(const QPixmap&);

设置和获取消息对话框的自定义图标

⑦、**standardButtons**: StandardButtons

访问函数: StandardButtons standardButtons() const; void setStandardButtons (StandardButtons);

设置和获取消息对话框使用的标准按钮, 默认不包含标准按钮。标准按钮可使用按位或来指定多个按钮, 按钮的显示顺序取决于平台, 比如在 windows 上"Save"按钮位于"Cancel"按钮的左侧, 但在 Mac OS 上则相反。

⑧、**textInteractionFlags**: Qt::TextInteractionFlags //qt5.1

访问函数: Qt::TextInteractionFlags textInteractionFlags() const;

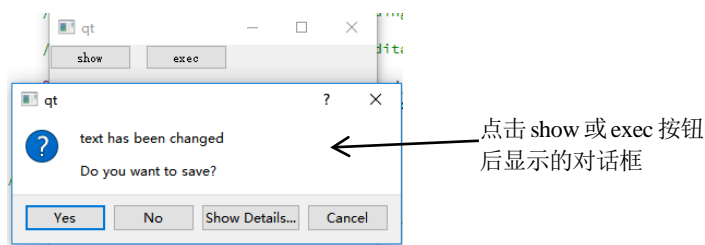
void setTextInteractionFlags(Qt::TextInteractionFlags);

描述消息对话框的标签(即显示的文本)与用户的交互方式, 比如是否可选择文本, 是否可编辑文本, 显示的链接是否可激活等。默认值取决于样式。该属性与 QLabel 类的 textInteractionFlags 属性是相同的, 详见 QLabel 类(第 4 章)关于该属性的讲解。

示例: 使用 QMessageBox 的属性创建消息对话框

```
#include<QtWidgets>
int main(int argc, char *argv[]) {    QApplication aa(argc, argv);
    QWidget w;
    QPushButton *pb=new QPushButton("show",&w);
    QPushButton *pb1=new QPushButton("exec",&w);pb1->move(88,0);
    QMessageBox pml;
    //设置各种属性
    pml.setText("text has been changed");
    pml.setInformativeText("Do you want to save?");
    pml.setDetailedText("AAAAAAAAAAAAAA\nBBBBBBBBBBBBBB\nCCCCCCCCCCC\nDDDDDDDD");
    pml.setStandardButtons(QMessageBox::Yes|QMessageBox::No|QMessageBox::Cancel);
    pml.setIcon(QMessageBox::Question);
    pml.setWindowModality(Qt::NonModal);    //把对话框设置为非模态的。
    QObject::connect(pb,&QPushButton::clicked,&pml,&QMessageBox::show);    //显示为非模态对话框
    QObject::connect(pb1,&QPushButton::clicked,&pml,&QMessageBox::exec);    //显示为模态对话框
    w.resize(300,200);    w.show();    return aa.exec(); }
```

运行结果及说明



10、QMessageBox 类中的函数

1)、构造函数

- ①、`QMessageBox(QWidget* parent = Q_NULLPTR);`
`QMessageBox(Icon icon, const QString &title, const QString &text,`
`StandardButtons buttons = NoButton, QWidget* parent = Q_NULLPTR,`
`Qt::WindowFlags = Qt::Dialog | Qt::MSWindowsFixedSizeDialogHint);`

使用给定的图标 icon、标题 title、文本 text、标准按钮 buttons 构造一个应用程序模式的消息对话框, 在 Mac 上, 若 parent 不为0, 且希望消息对话框显示为该父级的 Qt::Sheet, 则需把消息对话框设置为 Qt::WindowModal 模式, 否则消息对话框是标准的对话框。

2)、用于创建消息对话框的静态函数

- ②、`static StandardButton critical(QWidget* parent, const QString &title, const QString &text,`
`StandardButtons buttons = Ok, StandardButton defaultButton = NoButton);` //静态的。
`static StandardButton information(QWidget* parent, const QString &title, const QString &text,`
`StandardButtons buttons = Ok, StandardButton defaultButton = NoButton);` //静态的。
`static StandardButton question(QWidget* parent, const QString &title, const QString &text,`
`StandardButtons buttons = StandardButtons(Yes | No),`
`StandardButton defaultButton = NoButton);` //静态的。
`static StandardButton warning(QWidget* parent, const QString &title, const QString &text,`
`StandardButtons buttons = Ok, StandardButton defaultButton = NoButton);` //静态的。

- 以上函数表示, 在指定的父部件 parent 前面打开一个具有给定标题 title, 文本 text 和标准按钮 buttons 的严重警告、消息、询问、警告消息对话框
- 默认按钮 defaultButton 必须是已经存在的按钮, 若默认按钮为 QMessageBox::NoButton, 则系统自动选择一个合适的默认按钮。
- 创建的对话框是应用程序模式对话框,
- 该函数返回单击的标准按钮 StandardButton 值, 若按下 Esc, 则返回 EscapeButton 按钮。
- 注意: 不要在消息对话框运行期间删除父部件, 若需要这样操作, 应使用 QMessageBox 的构造函数创建消息对话框。

- ③、`static void about(QWidget* parent, const QString &title, const QString &text);` //静态的

显示一个带有标题 title 和文本 text 的“关于”对话框, 该对话框有一个 OK 按钮。在 Mac 上, 该对话框是非模态的, 在其他平台上是应用程序模式模态对话框。该对话框的图标在以下地方寻找:

- 若 parent>icon()存在, 则使用该图标。
- 否则尝试寻找包含父部件的顶级部件。
- 若失败, 则尝试寻找活动窗口。
- 最后, 若未找到则使用信息图标。

- ④、`static void aboutQt(QWidget* parent, const QString &title = QString());` //静态的

显示一个具有标题 title 的关于 Qt 的简单消息对话框, 消息包含 Qt 的版本号, 该对话框用于程序的帮助菜单很合适。Mac 上, 该对话框是非模态的, 在其他平台上是应用程序模式模态对话框。

3)、用于添加标准按钮的函数(注意 addButton 函数一次只能添加一个按钮)

- ⑤、`void addButton(QAbstractButton* button, ButtonRole role);`

把按钮 button 以指定的角色 role 添加到消息对话框中。

`QPushButton* addButton(const QString &text, ButtonRole role);`

使用文本 text 创建一个按钮，并把该按钮以指定的角色 role 添加到消息对话框中，然后返回这个按钮。

`QPushButton* addButton(StandardButton button);` //注意：StandardButton 后没有字母 s

把标准按钮 button 添加到消息对话框中，并返回这个按钮。

4)、显示消息对话框的函数

- ⑥、`virtual int exec();` //虚拟的，槽

这是对 `QDialog::exec()` 的重新实现。表示把消息对话框显示为模态对话框，直到用户关闭它为止，该函数返回用户单击的标准按钮 `StandardButton` 值，若标准按钮是自定义的，则该函数返回一个不明确的值。

`void open(QObject *receiver, const char* member);`

显示对话框，并把 `QDialog::finished()` 或 `QMessageBox::buttonClicked()` 信号连接到由 receiver 和 member 指定的槽函数，槽 member 具体连接到哪个信号，由指定槽时的参数决定，比如，若槽的参数为 `QAbstractButton*`，则连接到 `buttonClicked()` 信号。关闭对话框时，会断开信号和槽的联结。

5)、移除按钮

- ⑦、`void removeButton(QAbstractButton* button);` //移除而不删除按钮 button。

6)、获取消息对话框按钮

- ⑧、`QAbstractButton* button(StandardButton which) const;`

返回指向标准按钮 which 所对应的指针，若 which 不存在，则返回 0。

- ⑨、`StandardButton standardButton(QAbstractButton* button) const;`

返回按钮 button 所对应的标准按钮 `StandardButton` 枚举值，若 button 不是标准按钮，则返回 `NoButton`。

- ⑩、`QList<QAbstractButton*> buttons() const;` //返回消息对话框中所有按钮的列表。

- ⑪、`ButtonRole buttonRole(QAbstractButton* button) const;`

返回按钮 button 的按钮角色，若 button 为 0 或未添加到消息对话框中，则返回 `InvalidRole`

- ⑫、`QAbstractButton* clickedButton() const;`

返回用户点击的按钮，若 `exec()` 还未设用，则返回 0；若点击的是 `Esc` 键，且没有设置 `EscapeButton` 则返回 0。

7)、默认按钮

- ⑬、`void setDefaultButton(StandardButton button);` //把 button 设置为默认按钮

`void setDefaultButton(QPushButton* button);` //把 button 设置为默认按钮

`QPushButton* defaultButton() const;` //返回默认按钮，若未设置默认按钮，则返回 0。

8)、复选按钮

- ⑭、`void setCheckBox(QCheckBox* cb);` //qt5.2

把 cb 设置为消息对话框的复选按钮，若 cb 为 0，表示移除复选按钮。

`QCheckBox* checkBox() const;` //qt5.2，返回消息对话框的复选按钮。

9)、EscapeButton 按钮

- ⑮、`void setEscapeButton(QAbstractButton* button);` //设置 EscapeButton 按钮
`void setEscapeButton(StandardButton button);` //设置 EscapeButton 按钮
`QAbstractButton* escapeButton() const;` //返回按下 Esc 键时激活的按钮，

10)、模态模式及标题

- ⑯、`void setWindowModality(Qt::WindowModality windowModality);`
设置消息对话框的模态类型为 `windowModality`，该函数会隐藏 `QWidget::setWindowModality()`。在 Mac 上，若模态类型为 `Qt::WindowModal`，且有父级，则消息对话框将是 `Qt::Sheet`，否则是标准对话框。
- ⑰、`void setWindowTitle(const QString &title);`
把消息对话框的标题设置为 `title`，该函数会隐藏 `QWidget::setWindowTitle()`，在 macOS 上，标题会被忽略。

11)、信号

- ⑱、`void buttonClicked(QAbstractButton* button);` //信号
只要在消息对话框中点击按钮，就会发送此信号，被点击的按钮保存 `button` 中。

示例：使用静态函数创建消息对话框

//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>
#include<QDebug>
class B:public QWidget{    Q_OBJECT
public:    QWidget w;    QPushButton *pb;
    B(QWidget* p=0):QWidget(p){
        pb=new QPushButton("show",this);
        QObject::connect(pb,&QPushButton::clicked,this,&B::f);
        w.setWindowTitle("WWW");    w.resize(222,111);    w.show();    }
public slots:
    void f(){
        QMessageBox pm;
        //注意，通过静态函数 question 创建的消息对话框与 pm 是两个不同的实例，他们是互不影响的，
        //因此对 pm 属性的设置不会影响到由 question 函数创建的消息对话框。
        pm.setWindowModality(Qt::NonModal);    //把 pm 设置为非模态对话框
        pm.question(&w,"AAA","BBB");    /*创建的消息框仍为应用程序模式模态对话框，显示该消息
                                           对话框时，会显示在父窗口 w 的前面。*/

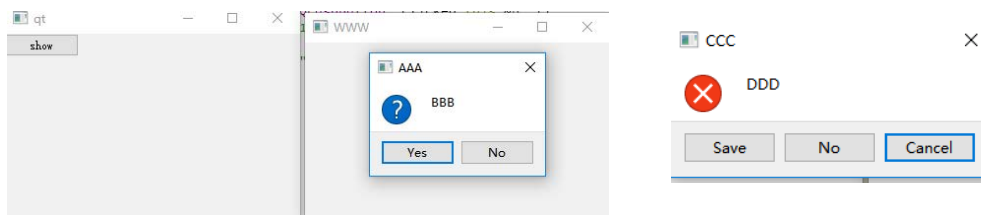
        //创建一个严重警告消息对话框。
        QMessageBox::StandardButton ps=
            QMessageBox::critical(    //该函数返回用户按下的标准按钮枚举值
                0,                //消息对话框的父部件也可以为 0。
                "CCC","DDD",    //设置标题和文本
                QMessageBox::Save|QMessageBox::No|QMessageBox::Cancel,    //设置消息框的按钮
                QMessageBox::Cancel);    //设置默认按钮

        qDebug()<<ps;
    }
};
#endif // M_H
```

//m.cpp 文件的内容

```
#include "m.h"
int main(int argc, char *argv[]) {    QApplication aa(argc, argv);
    B w;    w.resize(300,200);    w.show();    return aa.exec(); }
```

运行结果及说明



初始运行时程序显示两个窗口 qt 和 WWW，点击 show 按钮会弹出一个消息对话框 AAA，无论窗口 WWW 移至何处，点击 show 按钮弹出的消息框 AAA 都会显示在 WWW 的前面，关闭 AAA 之后，程序弹出 CCC 消息框，该消息框不一定会出现在 WWW 的前面。

示例：获取消息框的信息

//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>
#include<QDebug>
class B:public QWidget{    Q_OBJECT
public:    QPushButton *pb,*pb1,*xpb,*xpb1;    QMessageBox pml;    int i;
    B(QWidget* p=0):QWidget(p) {
        pb=new QPushButton("show",this);    pb1=new QPushButton("cusCane1");
        pml.setWindowTitle("AAA");    pml.setText("BBBBBBBBBBBBBBBBBBBB");
        xpb=pml.addButton(QMessageBox::Yes);    //添加一个标准按钮
        xpb1=pml.addButton("cusNo",QMessageBox::NoRole);    /*使用文本 cusNo 创建一个按钮，并将其添加到消息框中*/
        pml.addButton(pb1,QMessageBox::RejectRole);    //添加按钮 pb1 到消息框中
        pb1->setObjectName("cusCane1");    xpb->setObjectName("Yes");
        xpb1->setObjectName("cusNo");
        pml.setIcon(QMessageBox::Warning);    //向消息框中添加一个警告图标
        pml.setEscapeButton(xpb);    //把 Yes 按钮设置为 EscapeButton
        QCheckBox *pc=new QCheckBox("DDD");
        pml.setCheckBox(pc);    //向消息框中添加一个复选按钮
        QObject::connect(pb,&QPushButton::clicked,this,&B::f);
        QObject::connect(&pml,&QMessageBox::buttonClicked,this,&B::f1);    }

public slots:
    void f() {    i=pml.exec();    qDebug()<<"exec="<<i;    }
    void f1(QAbstractButton* p) {
        qDebug()<<"signals="<<p;    //查看 buttonClicked 信号的参数指向的按钮。
        qDebug()<<"clicked="<<pml.clickedButton();    //返回当用户点击的按钮
        qDebug()<<"Esc="<<pml.escapeButton();    //当按下 Esc 键激活的按钮。
        qDebug()<<"Yes="<<pml.button(QMessageBox::Yes);    //标准按钮 Yes 所对应的按钮的指针
        qDebug()<<xpb;
        qDebug()<<"No="<<pml.button(QMessageBox::No);    //标准按钮 No 所对应的按钮的指针。
    }
};
```

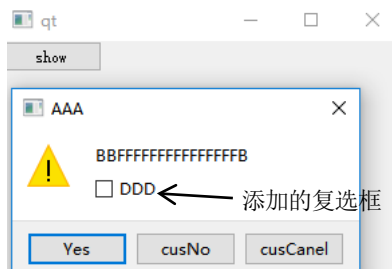
```

};
#endif // M_H

//m.cpp 文件的内容
#include "m.h"
int main(int argc, char *argv[]) {    QApplication aa(argc, argv);
    B w;    w.resize(300, 200);    w.show();    return aa.exec(); }

```

运行结果及说明



```

signals= QPushButton(0x9042a8, name="Yes")
clicked= QPushButton(0x9042a8, name="Yes")
Esc= QPushButton(0x9042a8, name="Yes")
Yes= QPushButton(0x9042a8, name="Yes")
QPushButton(0x9042a8, name="Yes")
No= QWidget(0x0)|
exec= 16384

```

按下 Yes 输出的内容

```

signals= QPushButton(0xa74748, name="cusNo")
clicked= QPushButton(0xa74748, name="cusNo")
Esc= QPushButton(0xa74aa8, name="Yes")
Yes= QPushButton(0xa74aa8, name="Yes")
QPushButton(0xa74aa8, name="Yes")
No= QWidget(0x0)
exec= 0

```

按下 cusNo 输出的内容

由输出可见,当按下自定义按钮 cusNo 时, exec()函数返回的是一个不明确的值,否则返回的是该按钮的 StandardButton 枚举值,标准按钮 No 并未添加到该消息框中,所以 button()函数始终返回 0,另外,本例点击复选框 DDD 不会有任何输出。

```

signals= QPushButton(0xa74aa8, name="Yes")
clicked= QPushButton(0xa74aa8, name="Yes")
Esc= QPushButton(0xa74aa8, name="Yes")
Yes= QPushButton(0xa74aa8, name="Yes")
QPushButton(0xa74aa8, name="Yes")
No= QWidget(0x0)
exec= 16384

```

按下 Esc 键输出的内容

示例：改变消息对话框的大小

消息对话框的大小默认是不可更改的,若要改变其大小,需重新实现消息对话框的 showEvent() 函数,因为默认在该函数内部把消息对话框设置为了固定大小。另外要想对话框大小可调整,还应把消息对话框的 Qt::MSWindowsFixedSizeDialogHint 属性设置为 0。

```

//m.h 文件的内容
#ifndef M_H
#define M_H
#include<QtWidgets>
class D:public QMessageBox{    Q_OBJECT
public:
    D(QWidget* p=0):QMessageBox(p) {    }

```

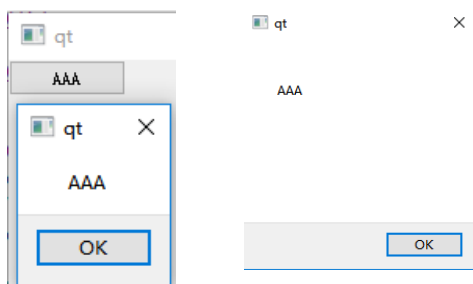
```

void showEvent(QShowEvent* e){    //重新实现 showEvent 函数
    QMessageBox::showEvent(e); //父类的该函数应先调用，因为该函数会调用 setFixedSize()。
    //清除对窗口大小的固定，清除之后窗口大小可调整。
    setFixedSize(QWIDGETSIZE_MAX, QWIDGETSIZE_MAX);
    resize(222, 222);    //重新设置窗口的大小。
}    };
#endif // M_H

//m.cpp 文件的内容
#include "m.h"
int main(int argc, char *argv[]){    QApplication aa(argc, argv);
    QWidget w;    QPushButton *pb=new QPushButton("AAA", &w);
    D pm;    pm.setText("AAA");
    //清除窗口的固定大小属性，以使窗口可调整
    pm.setWindowFlag(Qt::MSWindowsFixedSizeDialogHint, 0);
    QObject::connect(pb, &QPushButton::clicked, &pm, &QMessageBox::exec);
    w.resize(300, 200);    w.show();    return aa.exec(); }

```

运行结果及说明



消息对话框第一次显示时仍为 Qt 设置的固定大小，关闭消息对话框然后再次重新显示该对话框，可以看到对话框的大小为自定义的大小，且大小可调整了。

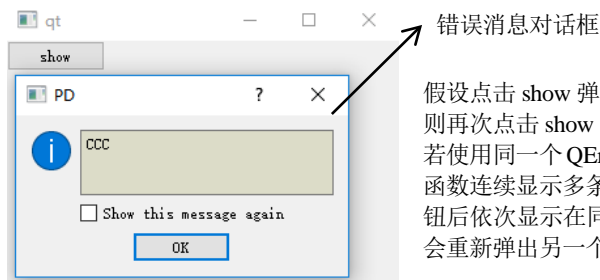
6.3 QMessageBox 类(错误消息对话框)

1、QErrorMessage 类用于向用户提供一些错误消息，该类比较简单，只需使用其构造函数和 showMessage()槽函数即可，其函数原型如下：

- `QErrorMessage(QWidget* parent = Q_NULLPTR);` //构造函数
- `void showMessage(const QString &mess);` //槽
- `void showMessage(const QString &mess, const QString &type);` //槽

显示一个错误消息对话框，显示的内容为 mess。其中 type 表示消息的类型，若指定了 type，则在这之后的与 type 相同的消息将不再显示。

2、错误消息对话框的样式及特点见下图



假设点击 show 弹出对话框 PD，若取消复选框的勾，则再次点击 show 时将不会再弹出该对话框。若使用同一个 QMessageBox 实例调用 showMessage() 函数连续显示多条信息，则这些信息会在点击 Ok 按钮后依次显示在同一个对话框的文本区域。注意，不会重新弹出另一个对话框

示例：理解 QMessageBox::showMessage() 函数

//m.h 文件的内容

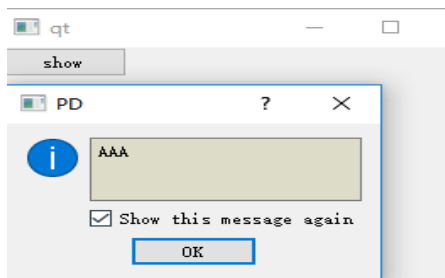
```
#ifndef M_H
#define M_H
#include<QtWidgets>
class B:public QWidget{    Q_OBJECT
public:    QMessageBox pe;    QPushButton *pb;
    B(QWidget* p=0):QWidget(p) {
        pb=new QPushButton("show", this);    pe.setWindowTitle("PD");
        QObject::connect(pb, &QPushButton::clicked, this, &B::f);
    }
public slots:
    void f() {
        pe.showMessage("AAA", "D");    //类型为 D
        pe.showMessage("CCC", "D");    //类型为 D
        pe.showMessage("EEE", "D");    //类型为 D
        pe.showMessage("D", "E");    //类型为 E，而非 D
        pe.showMessage("FFF");    //无类型 }
    };
#endif // M_H
```

//m.cpp 文件的内容

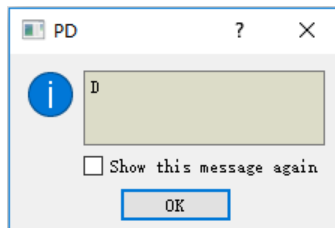
```
#include "m.h"
```

```
int main(int argc, char *argv[]) {    QApplication aa(argc, argv);  
    B w;    w.resize(300, 200);    w.show();    return aa.exec(); }
```

运行结果及说明



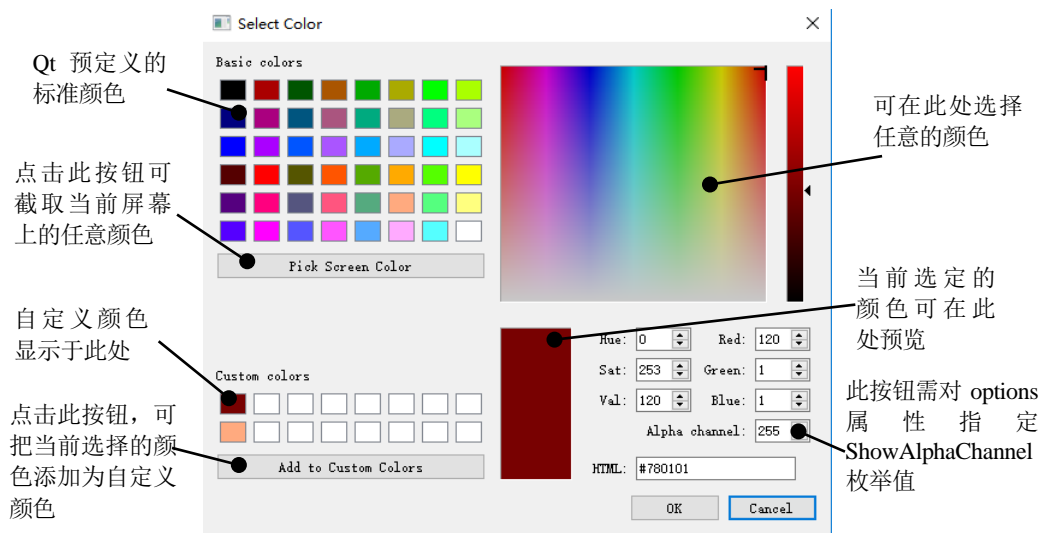
点击 show 之后，初始显示的对话框



在左侧的对话框中把复选框前面的勾去掉，然后点击 **Ok** 按钮之后，对话框显示的内容为 **D**，跳过了程序中类型为 **D**，内容为"CCC"和"EEE"的显示。

6.4 QColorDialog 类(颜色对话框)

- 1、颜色对话框主要用于向用户提供一个选择颜色的对话框，该对话框也是 Qt 预定义的标准对话框。
- 2、颜色对话框的样式见下图



3、QColorDialog 类中的属性

①、**currentColor**: QColor

访问函数: QColor currentColor() const; void setCurrentColor(const QColor&);

信号: currentColorChanged(const QColor&);

获取和设置对话框中当前选择的颜色

②、**options**: QColorDialogOptions

访问函数: QColorDialogOptions options() const; void setOptions(QColorDialogOptions);

该属性包含描述对话框界面外观的所有选项，这些选项应在对话框显示之前设置，在可见时设置不能保证立即对对话框产生效果。默认情况下，所有选项都是被禁用的。
QColorDialogOption 枚举如下表

QColorDialog::ColorDialogOption 枚举

标志: QColorDialog::ColorDialogOptions

作用: 描述颜色对话框外观的各种选项

成员	值	说明
QColorDialog::ShowAlphaChannel	0x0000 0001	允许用户选择颜色的 alpha 分量
QColorDialog::NoButtons	0x0000 0002	不显示 Ok 和 Cancel 按钮

QColorDialog::DontUseNativeDialog	0x0000 0004	使用 Qt 的标准颜色对话框，而不使用系统的本地颜色对话框。
-----------------------------------	-------------	--------------------------------

4、QColorDialog 类中的函数

- ①、`QColorDialog(QWidget* parent = Q_NULLPTR);`

`QColorDialog(const QColor& initial, QWidget* parent = Q_NULLPTR);` //构造函数

第 2 个函数表示，使用指定的初始颜色 initial 和父部件构造一个颜色对话框。

- ②、`void open(QObject* receiver, const char* member);`

显示颜色对话框，并把 colorSelected() 信号连接到由 receiver 和 member 指定的槽，关闭对话框时，信号和槽将被断开。

- ③、`static QColor getColor(const QColor &initial = Qt::white, QWidget* parent = Q_NULLPTR,`

`const QString &title = QString(),`

`ColorDialogOptions options = ColorDialogOptions());` //静态的

显示一个模态颜色对话框，该对话框具有标题 title(默认标题为"Select Color")、初始颜色 initial、父部件 parent、和选项属性 options。该函数返回用户选择的颜色，若未选择颜色则返回一个无效颜色。使用该函数可用于需要使用 QColor 的地方为用户快速的提供一个所见即所得的颜色选择对话框。比如

//把用户选择的颜色作为 QPalette 的参数

`QWidget w1; w1.setPalette(QPalette(QColorDialog::getColor()));`

- ④、`static QColor customColor(int index);` //静态的，返回索引 index 处的自定义颜色

`static void setCustomColor(int index, QColor color);` //静态的。

设置索引 index 处的自定义颜色，此函数不适用于 Mac 的本地颜色对话框。

- ⑤、`static int customCount();` //静态的，返回自定义颜色的数量。

- ⑥、`static QColor standardColor(int index);` //静态的，返回索引 index 处的标准色。qt5.0

`static void setStandardColor(int index, QColor color);` //静态的

把索引 index 处的标准颜色设置为 color。此函数不适用于 Mac 的本地颜色对话框。

- ⑦、`bool testOption(ColorDialogOption option) const;`

`void setOption(ColorDialogOption option, bool on = true);`

以上函数表示设置或返回 option 是否启用，以上函数是对属性 options 的设置。

- ⑧、`QColor selectedColor() const;`

返回用户通过单击 Ok 或等效按钮所选择的颜色。注意：该函数返回的是最终用户确定选择的颜色，在最终选定颜色之前，用户可以选择其他不同的颜色，因此此函数返回的颜色不一定与 currentColor 属性的颜色是相同的。

- ⑨、`void colorSelected(const QColor& color);` //信号

当用户点击确定以选择所要使用的最终颜色时发送此信号，color 为用户选择的颜色。注意：当用户改变当前颜色时，不会发送此信号。

- ⑩、`void currentColorChanged(const QColor& color);` //信号

当颜色对话框中的颜色发生改变时发送此信号，color 为当前颜色。

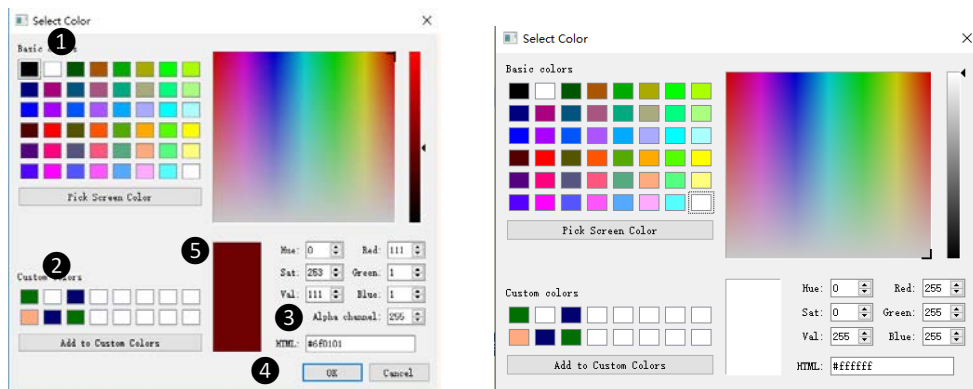
- 5、静态函数属于整个类不属于某个对象，因此从 QColorDialog 类中的静态函数可知，对标准颜色和自定义颜色的更改会影响到所有颜色对话框，也就是说颜色对话框共享相同的自定义颜色和标准颜色。
- 6、标准颜色和自定义颜色的索引标示规则为从上到下，从左到右(见右图)。

0	3	6
1	4	7
2	5	8

示例：颜色对话框

```
#include<QtWidgets>
int main(int argc, char *argv[]){    QApplication aa(argc, argv);
    QWidget w;
    QPushButton *pb=new QPushButton("AAA",&w);
    QPushButton *pb1=new QPushButton("BBB",&w);    pb1->move(88, 0);
    QColorDialog pd(QColor(111, 1, 1), 0);    QColorDialog pd1;
    //以下静态函数的设置会影响到所有的颜色对话框。
    QColorDialog::setCustomColor(3, QColor(1, 1, 111)); //设置索引为 3 的自定义颜色
    QColorDialog::setCustomColor(0, QColor(1, 111, 1));
    QColorDialog::setCustomColor(4, QColor(1, 1, 111));
    QColorDialog::setCustomColor(5, QColor(1, 111, 1));
    QColorDialog::setStandardColor(6, QColor(255, 255, 255));
    //以下函数为非静态函数，其设置只会影响到指定的对象。
    pd.setOptions(QColorDialog::ShowAlphaChannel);
    pd1.setOptions(QColorDialog::NoButtons);
    QObject::connect(pb, &QPushButton::clicked, &pd, &QColorDialog::exec);
    QObject::connect(pb1, &QPushButton::clicked, &pd1, &QColorDialog::exec);
    w.resize(300, 200);    w.show();    return aa.exec();    }
```

运行结果及说明(下图省略了主窗口)



- 图左侧为点击按钮 AAA 弹出的颜色对话框 pd，右侧为点击 BBB 弹出的颜色对话框 pd1
- ①、②两处，两个对话框是相同的，因为这两个地方都是使用的静态函数设置的，即标准颜色和自定义颜色是共享的。
- ③、④两处，右侧对话框没有相应的按钮，因为这两个按钮是属性 options，该属性是属于颜色对象框对象所特有的，不是共享的，因此不相同。
- ⑤处是创建对话框时指定的初始颜色。

示例：获取颜色对话框的信息

//m.h 文件的内容

```
#ifndef M_H
```

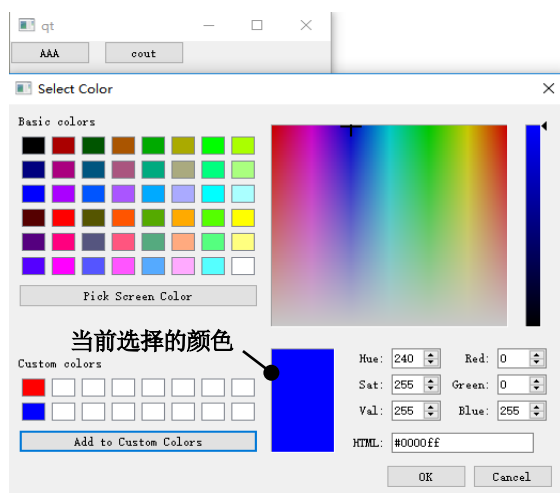
```

#define M_H
#include<QtWidgets>
#include<QDebug>
class B:public QWidget{    Q_OBJECT
public:
    QColorDialog pd; //局部对象应创建在构造函数之外，否则构造函数一结束，局部对象便被销毁了
    B(QWidget* p=0):QWidget(p) {
        QPushButton *pb=new QPushButton("AAA",this);
        QPushButton *pbl=new QPushButton("cout",this);    pbl->move(88,0);
        //以非模态形式弹出颜色对话框
        QObject::connect(pb,&QPushButton::clicked,&pd,&QColorDialog::show);
        QObject::connect(pbl,&QPushButton::clicked,this,&B::f2);
        QObject::connect(&pd,&QColorDialog::colorSelected,this,&B::f);
        QObject::connect(&pd,&QColorDialog::currentColorChanged,this,&B::f1); }
public slots:
    void f(const QColor& c){QDebug()<<"colorSelected";    qDebug()<<c<<endl; }
    void f1(const QColor& c){QDebug()<<"currentColorChanged";qDebug()<<c<<endl; }
    void f2() {QDebug()<<"result";
        qDebug()<<"cus="<<QColorDialog::customColor(1);        //输出索引为 1 处的自定义颜色
        qDebug()<<"std="<<QColorDialog::standardColor(1);        //输出索引为 1 处的标准颜色
        qDebug()<<"count="<<QColorDialog::customCount();        //输出自定义颜色的数量
        qDebug()<<"curren="<<pd.currentColor();        //输出当前选择的颜色
        qDebug()<<"select="<<pd.selectedColor()<<endl;        //输出最终确定选择的颜色
    }
};
#endif // M_H

//m.cpp 文件的内容
#include "m.h"
int main(int argc, char *argv[]){    QApplication aa(argc,argv);
    B w;    w.resize(300,200);    w.show();    return aa.exec(); }

```

运行结果及说明



```

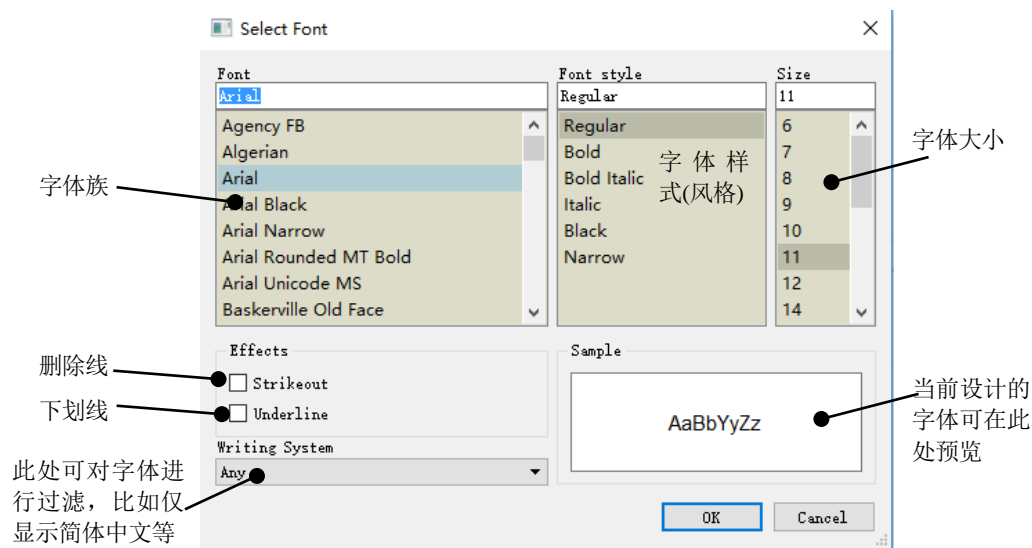
result
cus= QColor(ARGB 1, 0, 0, 1)
std= QColor(ARGB 1, 0, 0, 0.498039)
count= 16
curren= QColor(ARGB 1, 0, 0, 1)
select= QColor(Invalid)

```

添加如右图所示自定义颜色，并选择蓝色作为当前颜色，然后点击 cout 按钮，程序中的 f2 函数输出如上图，说明当前颜色 curren 为蓝色，最终确定选择的颜色 select 此时为无效(因为还未选定)，点击对话框的 Ok 按钮后才会有最终确定选择的颜色，程序中对颜色对话框两个信号的测试，读者可自行观察其不同以及信号的发送时机。

6.5 QFontDialog 类(字体对话框)

- 1、字体对话框主要用于向用户提供一个选择字体的对话框，该对话框也是 Qt 预定义的标准对话框。
- 2、字体是由多个方面的属性(比如粗细、倾斜、下划线等)进行描述的，因此一个字体需要对多方面进行设计
- 3、字体对话框的样式见下图



3、QColorDialog 类中的属性

①、currentFont: QFont

访问函数: QFont currentFont() const; void setCurrentFont(const QFont&);

信号: void currentFontChanged(const QFont&);

获取和设置对话框中当前设计的字体

②、options: QFontDialogOptions

访问函数: QFontDialogOptions options() const; void setOptions(QFontDialogOptions);

该属性包含描述对话框界面外观的所有选项，这些选项应在对话框显示之前设置，在可见时设置不能保证立即对对话框产生效果。默认情况下，所有选项都是被禁用的。QFontDialogOption 枚举如下表

QFontDialog::FontDialogOption 枚举		
标志: QFontDialog::FontDialogOptions		
作用: 描述字体对话框外观的各种选项，注意，某些平台可能不支持字体过滤选项		
成员	值	说明

QFontDialog::NoButtons	0x0000 0001	不显示 Ok 和 Cancel 按钮
QFontDialog::DontUseNativeDialog	0x0000 0002	在 Mac 上使用 Qt 的标准颜色对话框，而不使用系统的本地颜色对话框。
QFontDialog::ScalableFonts	0x0000 0004	显示可缩放字体
QFontDialog::NonScalableFonts	0x0000 0008	显示不可缩放字体
QFontDialog::MonospacedFonts	0x0000 0010	显示等宽字体
QFontDialog::ProportionalFonts	0x0000 0020	显示比例字体

4、QColorDialog 类中的函数

- ①、**QFontDialog**(QWidget* parent = Q_NULLPTR);

QFontDialog(const QFont& initial, QWidget* parent = Q_NULLPTR);

第 2 个函数表示，使用指定的初始字体 initial 和父部件构造一个字体对话框。

- ②、void **open**(QObject* receiver, const char* member);

显示字体对话框，并把 **fontSelected()** 信号连接到由 receiver 和 member 指定的槽，关闭对话框时，信号和槽将被断开。

- ③、static QFont **getFont**(bool *ok, const QFont& initial, QWidget* parent = Q_NULLPTR,

const QString &title = QString(),

FontDialogOptions options = FontDialogOptions()); //静态的

static QFont **getFont**(bool *ok, QWidget* parent = Q_NULLPTR); //静态的

以上函数表示，显示一个模态字体对话框，且返回用户选择的字体，若用户选择“Cancel”，则第 1 个函数返回初始字体 initial，第 2 个函数返回 Qt 的默认字体。各参数意义如下：

- title 表示标题、initial 表示初始字体、parent 表示父部件、options 表示选项属性。
- 参数 ok 若不是空指针，则若用户点击的是 Ok 按钮，则*ok 的值为 1，若点击的是 Calcel 按钮，则*ok 的值为 0，该参数可用于判断用户点击的是对话框中的哪个按钮。
- 使用以上函数可用于需要使用 QFont 的地方为用户快速的提供一个所见即所得的字体选择对话框。比如

//把用户选择的字体作为 setFont 的参数

QWidget w1; w1.setFont(QFontDialog::getFont(0,w.font())); 或 w1.setFont(QFontDialog::getFont(0));

- ④、void **setOption**(FontDialogOption option, bool on = true);

bool **testOption**(FontDialogOption option) const;

以上函数表示设置或返回 option 是否启用，以上函数是对属性 options 的设置。

- ⑤、QFont **selectedFont**() const;

返回用户通过单击 Ok 或等效按钮所选择的字体。注意：该函数返回的是最终用户确定选择的字体，在最终选定字体之前，用户可以选择其他不同的字体，因此此函数返回的字体不一定与 currentFont 属性的字体是相同的。

- ⑥、void **currentFontChanged**(const QFont& font); //信号

当字体对话框中的当前字体发生改变时发送此信号，font 为当前字体。

- ⑦、void **fontSelected**(const QFont& font); //信号

当用户点击确定以选择所要使用的最终字体时发送此信号，font 为用户选择的字体。

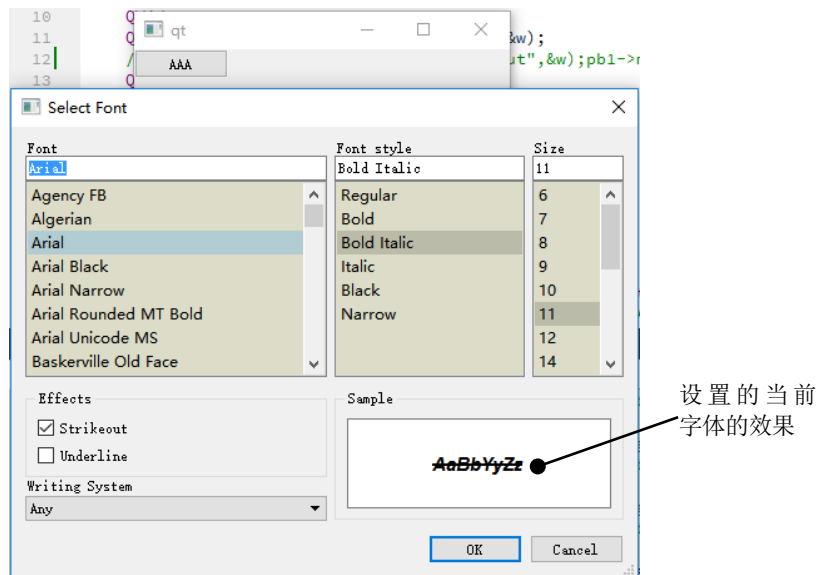
注意：当用户改变当前字体时，不会发送此信号。

5、QFontDialog 类中的属性和函数与 QColorDialog 类是相似的，更多的示例可参阅 QColorDialog 类。

示例：显示一个字体对话框

```
#include<QtWidgets>
int main(int argc, char *argv[]) {    QApplication aa(argc, argv);
    QWidget w;
    QPushButton *pb=new QPushButton("AAA",&w);
    QFontDialog pf;                //创建一个字体对话框对象。
    QFont f;                      //创建字体
    f.setBold(true);              //设置为粗体
    f.setItalic(1);              //设置为斜体
    f.setPointSize(11);          //字体大小为 11
    f.setStrikeOut(1);           //字体带删除线
    f.setFamily("Arial");        //字体族为"Arial"
    pf.setCurrentFont(f);        //把字体对话框的当前字体设置为 f
    QObject::connect(pb,&QPushButton::clicked,&pf,&QFontDialog::show);
    w.resize(300,200);    w.show();    return aa.exec(); }
```

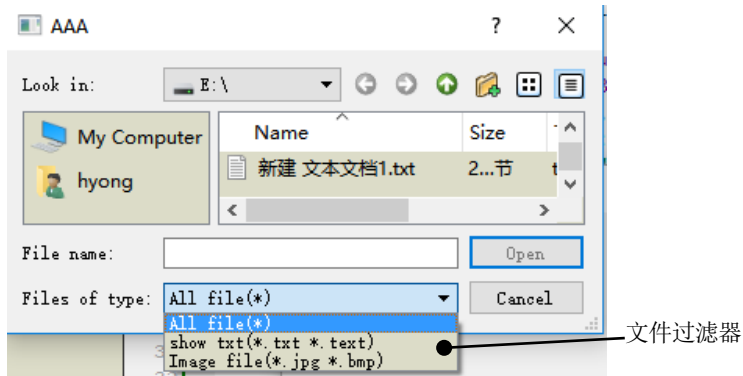
运行结果及说明



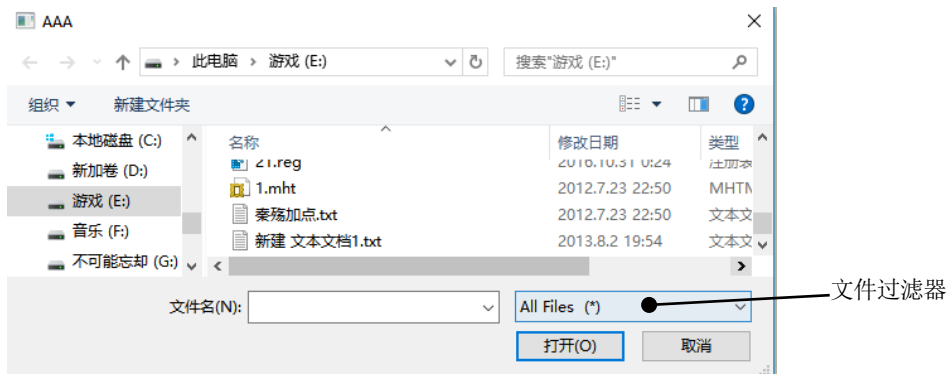
6.6 QFileDialog 类(文件对话框)

一、文件对话框基础

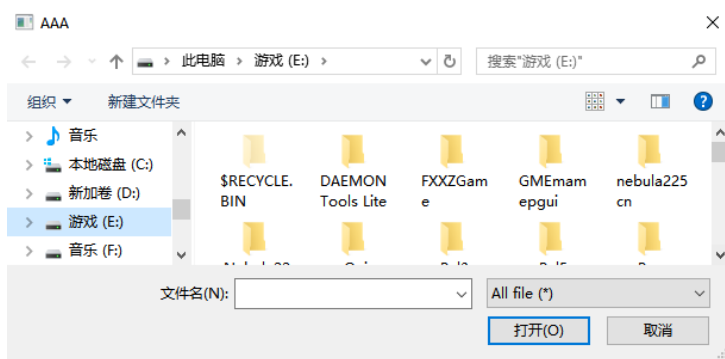
- 1、文件对话框主要用于向用户提供一个选择文件或目录的对话框，该对话框也是 Qt 预定义的标准对话框。
- 2、对需要选择的文件，需要从多方面进行描述，比如文件的路径，文件的类型，文件的显示方式等。
- 3、文件对话框分为 Qt 文件对话框和本地文件对话框(即各平台下的文件对话框)
下图为 Qt 文件对话框的样式



下图为 windows 系统下的文件对话框样式

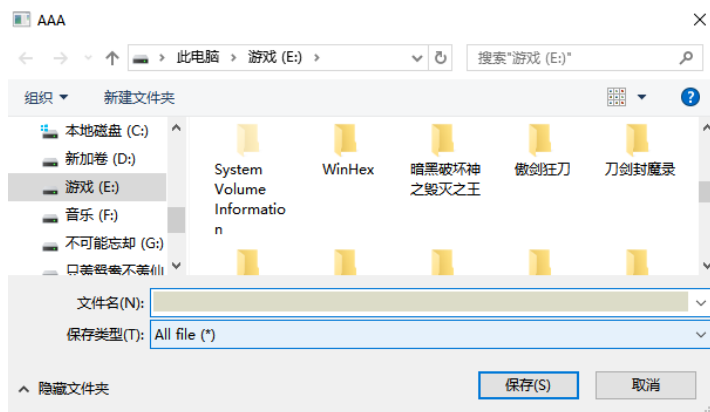


- 5、使用 QUrl 指定本地路径的方式为：QUrl q("file:///E:/"); 表示位于 E 盘根目录下。
- 6、文件对话框的类型、样式及创建方法见下图



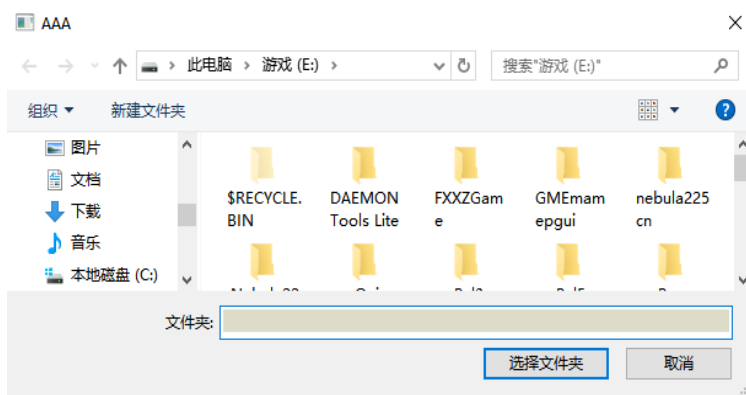
打开文件对话框

QFileDialog::getOpenFileName();或 QFileDialog::getOpenFileNames();



保存文件对话框

QFileDialog::getSaveFileName();



目录文件对话框

QFileDialog::getExistingDirectory();

二、QFileDialog 类中的属性

①、acceptMode: AcceptMode

访问函数: AcceptMode acceptMode() const; void setAcceptMode(AcceptMode);

此属性描述文件对话框是“打开”文件对话框，还是“保存”文件对话框，明显的区别就是文件名右侧的按钮是"Open"还是"Save"。默认为 AcceptOpen(打开)。AcceptMode 枚举如下表

QFileDialog::AcceptMode 枚举(无标志)			
作用：描述文件对话框是打开文件对话框还是保存文件对话框			
成员	值	成员	值
QFileDialog::AcceptOpen	0(默认)	QFileDialog::AcceptSave	1

②、defaultSuffix: QString

访问函数: QString defaultSuffix() const; void setDefaultSuffix(const QString&);

若文件名未指定后缀，则向文件名中添加一个后缀。此函数配合 selectedFiles()使用，便能看到其效果。

③、fileMode: FileMode

访问函数: FileMode fileMode() const; void setFileMode(FileMode);

获取和设置文件对话框的文件模式，文件模式描述了用户可以在对话框中选择的项目的类型和数量(比如只能选择目录，只能选择单个文件等)。FileMode 枚举见下表

QFileDialog::FileMode 枚举(无标志)		
作用：描述用户可在文件对话框中选择的内容。		
成员	值	说明
QFileDialog::AnyFile	0	只能选择单个文件，无论该文件是否存在(默认值)
QFileDialog::ExistingFile	1	只能选择单个已存在的文件
QFileDialog::Directory	2	只能选择目录。会同时显示文件和目录，但，本地 Windows 文件对话框不支持在目录选择器中显示文件。
QFileDialog::ExistingFiles	3	可以选择多个已存在的文件
说明：文件是否已存在的区别 AnyFile 在“打开文件对话框”中，可输入一个不存在的文件名称，并打开，保存类似。 但 ExistingFile 在打开文件或保存文件对话框中必须选择一个已经存在的文件。		

④、options: Options 访问函数: Options options() const; void setOptions(Options);

该属性包含描述对话框界面外观的所有选项，这些选项应在对话框显示之前设置，在可见时设置不能保证立即对对话框产生效果。默认情况下，所有选项都是被禁用的。Option 枚举如下表

QFileDialog::Option 枚举	
标志: QFileDialog::Options	

作用：描述文件对话框外观的各种选项。		
成员	值	说明
QFileDialog::ShowDirsOnly	0x0000 0001	仅显示目录。默认为文件和目录都会显示
QFileDialog::DontResolveSymlinks	0x0000 0002	不解析符号链接(类似于快捷方式)。默认为解析。
QFileDialog::DontConfirmOverwrite	0x0000 0004	若选择了已存在的文件不需要确认，默认为需要确认。
QFileDialog::DontUseNativeDialog	0x0000 0010	不使用本地文件对话框，默认情况下，除非使用了含有 Q_Object 宏的 QFileDialog 子类，或平台没有所需类型的本地文件对话框，否则使用本地文件对话框。以上规则说明使用 QFileDialog 类创建的对话框是 Qt 文件对话框，而不是本地文件对话框。
QFileDialog::ReadOnly	0x0000 0020	只读模式
QFileDialog::HideNameFilterDetails	0x0000 0040	指示文件过滤器的详细信息是否隐藏。
QFileDialog::DontUseSheet	0x0000 0008	该属性不再被支持。
QFileDialog::DontUseCustomDirectoryIcons	0x0000 0080	始终使用默认的目录图标，有些平台允许用户设置不同的图标(即自定义图标)。

⑤、viewMode: ViewMode

访问函数：ViewMode viewMode() const; void setViewMode(ViewMode);

描述对话框中文件的显示方式(列表还是详细)，默认为 Detail(详细信息)。ViewMode 枚举如下表

QFileDialog::ViewMode 枚举(无标志)		
作用：描述对话框中文件的显示方式。		
成员	值	说明
QFileDialog::Detail	0	显示项目的详细信息
QFileDialog::List	1	仅显示项目的名称和图标。

⑥、supportedSchemes: QStringList

访问函数：QStringList supportedSchemes() const; void setSupportedSchemes(const QStringList&);

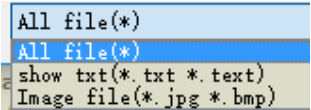
描述文件对话框允许导航到的 URL 方案。默认为空，即没有限制。

三、文件过滤器

- 1、文件过滤器：用于过滤需要被显示的文件，比如若过滤器被调置为"Image(*.jpg *.bmp)", 则只有后缀为 jpg 和 bmp 的文件被显示。
- 2、文件过滤器的分类：
 - 文件的后缀通常用于表示文件的类型，比如.txt 文件表示文本文件，根据对文件后缀的不同描述方法，把文件过滤器分为名称过滤器和 MIME 类型过滤器
 - 文件还具有只读、可修改、系统文件等性质，从这些性质对文件进行过滤的过滤器称为模型过滤器。
- 3、默认过滤器：是指对话框初次显示时使用的过滤器。

4、名称过滤器：

- ①、安装方法：使用 setNameFilter()函数、setNameFilters()函数、QFileDialog 的构造函数、getXXX()系列静态函数安装。
- ②、创建方法：名称过滤器使用字符串来创建，其命名规则如下：
 - 过滤的文件之间使用空格进行分格，比如"*.jpg *.bmp"，在 jpg 后有一个空格，表示仅显示后缀为 jpg 和 bmp 的文件。
 - 若过滤器含有小括号，则仅把括号中包含的文本用作过滤器，比如"show Image(*.jpg *.bmp)"; 表示显示后缀为 jpg 和 bmp 的文件。
 - 多个过滤器之间使用两个分号(也可使用"\n")进行分隔，比如"All file(*);; show txt(*.txt *.text);; Image file(*.jpg *.bmp)"，其效果如右图

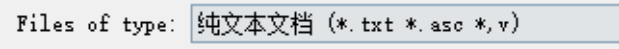


5、MIME 类型过滤器

- ①、安装方法：使用 setMimeTypeFilters()函数安装。
- ②、MIME 简介：MIME 全称是多用途互联网邮件扩展(Multipurpose Internet Mail Extensions)，最初是为了使纯文本格式的电子邮件能够支持多种格式(比如多媒体、图片等)而制定的，MIME 类型的优点是可以为每种类型提供所有可能的后缀，比如 jpeg 图像有三个可能的后缀(jpeg、jpg、jpe)，使用 MIME 类型可以直接描述这三种类型，其形式为"image/jpeg"，MIME 类型最常见的形式是一个主类型加一个子类型，并用斜线分隔，有效的主类型为：text、image、audio、video、applications、multipart、message。
- ③、创建方法：MIME 类型过滤器使用字符串来创建，在使用时只需把 MIME 类型使用双引号括起来即可，比如"text/plain" 表示纯文本文档类型*.txt *.asc *.v，代码为：

```
QStringList s; s<<"text/plain"; setMimeTypeFilters(s);
```

其效果如下图



注意：任意的二进制文件都被叫做 application/octet-stream 类型，因此该类型可以表示所有文件(即常见的 All files(*))，下表列出一些 Qt 中常见的 MIME 类型与实际类型的对照表(详细内容请查阅相关文章)

MIME 类型	Qt 中对应的类型
application/octet-stream	All files(*)
text/plain	纯文本文档(*.txt *.asc *.v);
text/html	HTML 文档(*.html *.htm)
image/jpeg	JPEG 图像(*.jpeg *.jpg *.jpe)
audio/mpeg	MP3 音频(*.mp3 *.mpga)
video/x-ms-asf	ASF 视频(*.asf)
video/mpeg	MPEG 视频(*.mpeg *.mpg *.mp2 *.mpe *.vob [0-9][0-9][0-9].vdr)

video/x-msvideo	AVI 视频(*.avi *.avf *.divx)
video/mp4	MPEG-4 视频(*.mp4 *.m4v *.f4v *.lrv);

6、模型过滤器

- ①、安装方法：使用 `setFilter()` 函数安装。
- ②、创建方法：模型过滤器由 `QDir::Filter` 枚举指定，比如 `QDir::Hidden` 表示列出隐藏文件，`QDir::Modified` 表示列出已被修改的文件，`QDir::System` 表示列出系统文件等，其详细的取值详见 `QDir` 类。

四、QFileDialog 类中的函数

1、构造函数及显示文件对话框的函数

- ①、`QFileDialog(QWidget* parent, Qt::WindowFlags flags);`

`QFileDialog(QWidget* parent = Q_NULLPTR, const QString &caption = QString(),
const QString &directory = QString(), const QString &filter = QString());`

各参数意义为：caption 表示对话框的标题，directory 表示对话框最初显示该目录的内容。filter 表示文件过滤器。

- ②、`void open(QObject* receiver, const char* member);`

显示文件对话框，并把信号连接到由 receiver 和 member 指定的槽，关闭对话框时，信号和槽将被断开。如果 fileMode 是 ExistingFiles 则信号是 filesSelected()，若 fileMode 是其他类型，则信号是 fileSelected()，注意，与 filesSelected 相差一个字母 s。

2、以下静态函数可用于快速的创建各种类型的文件对话框。

- ③、`static QString getExistingDirectory(QWidget* parent=Q_NULLPTR, const QString &caption=QString(),
const QString &dir = QString(), Options options = ShowDirsOnly);` //静态的

创建并显示一个模态目录文件对话框，该对话框只能选择目录。各参数意义如下：

- parent: 对话框的父窗口，若不为 0，则对话框会显示在父窗口的中央。
- caption: 对话框的标题，若为空，则使用默认标题
- dir: 对话框的工作目录，若为空则使用当前目录。
- options: 文件对话框的 options 属性(详见该属性)。
- 返回值: 返回用户选择的已存在的目录。
- 在 windows 和 Mac 上，此函数默认使用本地文件对话框，而不是使用 Qt 的文件对话框(即使用 QFileDialog 创建的对话框)
- 默认情况下，本地 widows 在目录选择器中不会显示文件，若要使其显示文件需设置 options 参数包含 DontUseNativeDialog，并去掉 showDirsOnly 参数。

- ④、`static QString getOpenFileName(QWidget* parent = Q_NULLPTR, const QString &caption = QString(),
const QString &dir = QString(), const QString &filter = QString(),`

`QString *selectedFilter = Q_NULLPTR, Options options = Options());` //静态的

`static QStringList getOpenFileNames(QWidget* parent = Q_NULLPTR,`

`const QString &caption = QString(), const QString &dir = QString(),`

`const QString &filter = QString(), QString *selectedFilter = Q_NULLPTR,`

`Options options = Options());` //静态的

创建并显示一个模态打开文件对话框，其中第 1 个函数只能选择单个文件，第 2 个函数可同时选择多个文件。各参数意义如下：

- **parent**(父部件)、**caption**(标题)、**dir**(工作目录)、**options**(options 属性)、**filter**(文件过滤器)
- **selectedFilter** 参数：表示对话框初始显示时，默认使用的文件过滤器，该参数需是 **filter** 参数的子项。比如

```
QFileDialog::getOpenFileName(0,"A","E:/",
                             "All file(*.jpg *.bmp)", //filter 参数的值
                             new QString("Image(*.jpg *.bmp)"); //selectedFilter 参数的值
    则当对话框初次显示时，将使用"Image(*.jpg *.bmp)" 作为默认过滤器。
```

- 返回值：返回用户选择的文件。
- 在 windows 和 Mac 上，此函数默认使用本地文件对话框，而不是使用 Qt 的文件对话框(即使用 QFileDialog 创建的对话框)

- ⑤、static QString **getSaveFileName**(QWidget* **parent** = Q_NULLPTR, const QString &**caption** = QString(), const QString &**dir** = QString(), const QString &**filter** = QString(), QString ***selectedFilter** = Q_NULLPTR, Options **options** = Options()); //静态的
- 创建并显示一个模态保存文件对话框，该函数的各参数的意义与 **getOpenFileName()** 完全相同。该函数返回用户选择的文件，但该文件不一定存在。

- ⑥、static QUrl **getExistingDirectoryUrl**(QWidget* **parent** = Q_NULLPTR, const QString &**caption** = QString(), const QUrl &**dir** = QUrl(), Options **options** = ShowDirsOnly, const QStringList &**supportedSchemes** = QStringList()); //静态的，qt5.2
- static QUrl **getOpenFileUrl**(QWidget* **parent** = Q_NULLPTR, const QString &**caption** = QString(), const QUrl &**dir** = QUrl(), const QString &**filter** = QString(), QString ***selectedFilter** = Q_NULLPTR, Options **options** = Options(), const QStringList & **supportedSchemes** = QStringList()); //静态的，qt5.2
- static QList<QUrl> **getOpenFileUrls**(QWidget* **parent** = Q_NULLPTR, const QString &**caption** = QString(), const QUrl &**dir** = QUrl(), const QString &**filter** = QString(), QString ***selectedFilter** = Q_NULLPTR, Options **options** = Options(), const QStringList & **supportedSchemes** = QStringList()); //静态的，qt5.2
- static QUrl **getSaveFileUrl**(QWidget* **parent** = Q_NULLPTR, const QString &**caption** = QString(), const QUrl &**dir** = QUrl(), const QString &**filter** = QString(), QString ***selectedFilter** = Q_NULLPTR, Options **options** = Options(), const QStringList & **supportedSchemes** = QStringList()); //静态的，qt5.2

以上函数与对应的 **getExistingDirectory()**、**getOpenFileName()**、**getOpenFileNames()**、**getSaveFileName()** 函数的功能相同，其区别在于以上函数为用户提供了选择远程目录的功能，在不支持远程文件的平台上，Qt 允许只选择本地文件。其中 **supportedSchemes** 参数，表示限制用户能够选择的 Url 的类型，若为空(默认)，则表示没有限制

3、以下函数为文件对话框中的基本设置

- ⑦、`QDir directory() const;` //返回当前对话框显示的目录
`void setDirectory(const QString& directory);` //设置文件对话框的当前目录
`void setDirectory(const QDir& directory);` //设置文件对话框的当前目录。

- ⑧、`void setOption(Option option, bool on = true);`
`bool testOption(Option option) const;`

以上函数表示设置或返回 option 是否启用，以上函数是对属性 options 的设置。

- ⑨、`void selectFile(const QString &filename);` //在文件对话框中选中给定的文件，此函数可用于设置文件对话框初次显示时，默认选中的文件。

`QStringList selectedFiles() const;` //返回被选中的文件的绝对路径，

4、名称过滤器

- ⑩、`QStringList nameFilters() const;` //返回对话框中的名称过滤器。
`QString selectedNameFilter() const;` //返回用户当前选择的名称过滤器。
`void setNameFilter(const QString& filter);` //设置文件对话框中使用的名称过滤器。
`void setNameFilters(const QStringList& filters);` //设置文件对话框中使用的名称过滤器。
`void selectNameFilter(const QString &filter);`

选择过滤器 filter，filter 应是 setNameFilter()或 setNameFilters()函数为对话框设置的过滤器的子项。此函数可把过滤器 filter 设置为对话框初次显示时的默认过滤器。

5、MIME 类型过滤器

- ⑪、`QStringList mimeTypeFilters() const;` //返回当前使用的 MIME 类型过滤器，qt5.2
`QString selectedMimeTypeFilter() const;` //返回用户选择的 MIME 类型过滤器，qt5.9
`void setMimeTypeFilters(const QStringList& filters);` //qt5.2
 设置 MIMI 类型过滤器。使用该函数设置的过滤器，将覆盖之前设置的名称过滤器，并更改 nameFilters()的返回值。
`void selectMimeTypeFilter(const QString &filter);` //qt5.2

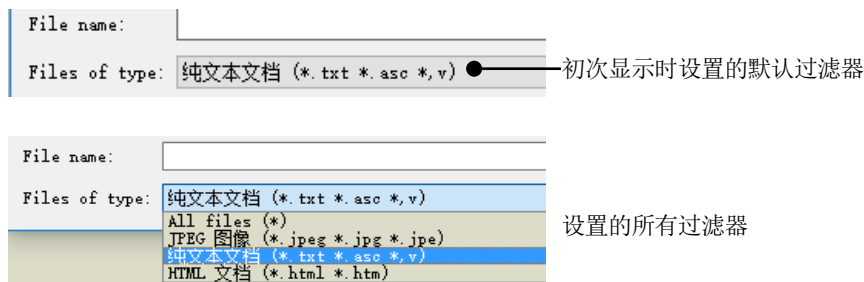
选择过滤器 filter，filter 应是 setMimeTypeFilters()函数为对话框设置的过滤器的子项。此函数可把过滤器 filter 设置为对话框初次显示时的默认过滤器。

6、模型过滤器

- ⑫、`void setFilter(QDir::Filters filters);` //把由模型使用的过滤器设置为 filters
`QDir::Filters filter() const;` //返回显示文件时使用的过滤器

过滤器示例(其效果见下图):

```
QFileDialog *pfl=new QFileDialog;
QStringList s;
s<<"application/octet-stream"; //所有文件
s<<"image/jpeg"; //jpeg 图像文件
s<<"text/plain"; //纯文本文件
s<<"text/html"; //html 文本文件
pfl->setMimeTypeFilters(s);
pfl->selectMimeTypeFilter("text/palin"); //把过滤器 text/palin 设置为默认过滤器
pfl->show();
```



7、设置文件对话框各处的标签文本。

⑬、QString **labelText**(DialogLabel label) const;

void **setLabelText**(DialogLabel label, const QString& text);

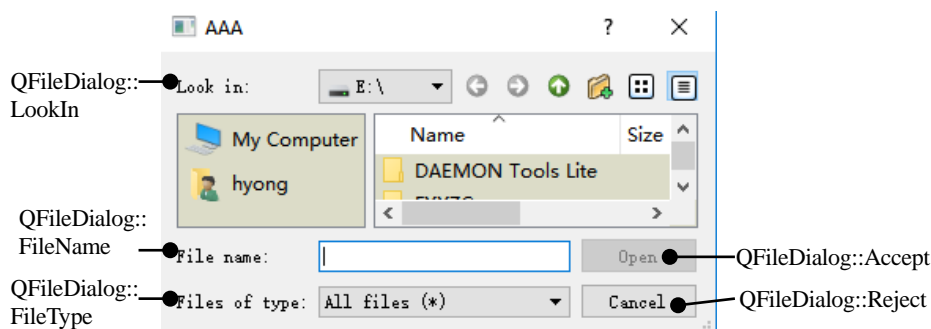
以上函数用于获取和设置 文件对话框中由 label 指定的位置处的文本

DialogLabel 枚举用于指定文件对话框中标签文本的位置，其取值和位置见下表及图

QFileDialog::DialogLabel 枚举(无标志)

作用：指定文件对话框中各标签文本的位置(见下图)。

成员	值	成员	值	成员	值
QFileDialog::LookIn	0	QFileDialog::FileType	2	QFileDialog::Reject	4
QFileDialog::FileName	1	QFileDialog::Accept	3		



8、保存文件对话框的状态及浏览历史记录

⑭、QByteArray **saveState**() const;

bool **restoreState**(const QByteArray& state);

以上函数用于保存和恢复对话框的状态。通常与 QSettings 一起使用。若有错误，则第 2 个函数返回 false。

⑮、QStringList **history**() const; //返回文件对话框的浏览历史记录。

void **setHistory**(const QStringList& paths);//设置文件对话框的浏览历史记录。

9、设置文件对话框左侧工具栏的地址。

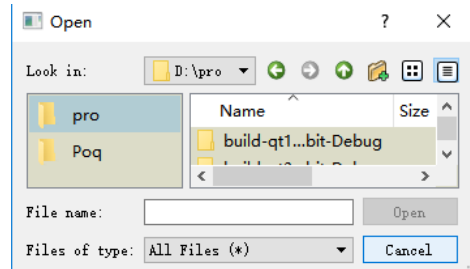
⑯、QList<QUrl> **sidebarUrls**() const; //返回当前侧面工具栏(sidebar)中的 Url 列表。

void **setSidebarUrls**(const QList<QUrl> &urls); //设置侧面工具栏(sidebar)中的 Url。

以上函数需设置 `QFileDialog::DontUseNativeDialog` 为 true 才有效。

示例(效果见右图):

```
QFileDialog *pf=new QFileDialog;
pf->setOptions(QFileDialog::DontUseNativeDialog);
QList<QUrl> urls;
urls << QUrl::fromLocalFile("D:/pro")
      << QUrl::fromLocalFile("E:/Poq");
pf->setSidebarUrls(urls);
pf->show();
```



10、以下函数与模型视图有关

QAbstractItemDelegate* **itemDelegate**() const; //返回文件对话框中视图的项目的项目委托。

void **setItemDelegate**(QAbstractItemDelegate* delegate); //设置文件对话框中视图的项目的项目委托。

QAbstractProxyModel* **proxyModel**() const;

void **setProxyModel**(QAbstractProxyModel* proxyModel);

返回或设置文件对话框的代理模型。现有的代理模型将被移除，但不会被删除。文件对话框将获得 proxyModel 的所有权。

QFileIconProvider* **iconProvider**() const; //返回图标提供程序

void **setIconProvider**(QFileIconProvider* provider); //设置图标提供程。用于自定义图标。

11、以下函数与 Url 有关

void **setDirectoryUrl**(const QUrl& directory); //设置对话框的当前 Url 目录, qt5.2

QUrl **directoryUrl**() const; //返回当前对话框显示的目录的 Url, qt5.2

QList<QUrl> **selectedUrls**() const; //返回被选中的文件的 Url 路径, qt5.2

void **selectUrl**(const QUrl &url); //在文件对话框中选给定的 url, qt5.2

五、QFileDialog 类中的信号

1、void **currentChanged**(const QString &path); //

当文件更改时，发送此信号。path 为新文件的路径(包含文件名)。

2、void **directoryEntered**(const QString &directory); //信号，

用户进入目录时发送此信号，directory 为新目录的路径。

3、void **filterSelected**(const QString &filter); //当过滤器被选中时发送此信号。

4、void **fileSelected**(const QString &file); //信号

void **filesSelected**(const QStringList &selected); //信号，注意 file 后多了一个 s

以上信号表示，当选择更改且对话框被接受时，发送此信号，其中 file 为用户选定的文件，selected 为用户选定的文件列表。

5、void **currentUrlChanged**(const QUrl &url); //当文件更改时，发送此信号。url 为新文件的 Url。qt5.2

6、void **directoryUrlEntered**(const QUrl &directory); //用户进入目录时发送此信号。qt5.2

7、void **urlSelected**(const QUrl& url); //信号，qt5.2

void **urlsSelected**(const QList<QUrl> &urls); //信号，qt5.2

以上信号表示，当选择更改并且接受对话框时，信号与用户选择的 url(有可能为空)或

urls(有可能为空)一起被发送。

示例：文件对话框

//m.h 文件内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>
#include<QDebug>

class B:public QWidget{    Q_OBJECT
public:    QFileDialog *pf;
    B(QWidget* p=0):QWidget(p) {
        QPushButton *pb=new QPushButton("AAA",this);
        pf=new QFileDialog(0,"AAA","E:/");
        QStringList s;
        s<<"application/octet-stream"; //所有文件
        s<<"image/jpeg"; //jpeg 图像文件
        s<<"text/plain"; //纯文本文件
        s<<"text/html"; //html 文本文件
        pf->setMimeTypeFilters(s); //安装 MIME 类型过滤器
        pf->selectMimeTypeFilter("text/plain"); //设置初次显示时的默认过滤器
        pf->setDefaultSuffix("XXX"); //若未为文件名指定后缀，则为其添加后缀 XXX。
        pf->setDirectory("E:/sss"); /*此设置会使构造函数的目录失效,对于本人使用的机器这是一个无效的目录，因此会 Qt 使用默认目录*/
        pf->selectFile("aaa"); /*设置初次显示时选中的默认文件，即使该文件可能不存在，仍会在"file name"栏处显示该文件。*/
        pf->setLabelText(QFileDialog::LookIn,"directory"); //修改 Look In 处的标签文本。
        pf->setLabelText(QFileDialog::Reject,"Close"); /*把文件对话框中名称默认为 Cancel 的按钮，修改为名称 Close*/

        QObject::connect(pb,&QPushButton::clicked,pf,&QFileDialog::show);
        QObject::connect(pf,&QFileDialog::currentChanged,this,&B::f1);
        QObject::connect(pf,&QFileDialog::directoryEntered,this,&B::f2);
        QObject::connect(pf,&QFileDialog::filterSelected,this,&B::f3);
        QObject::connect(pf,&QFileDialog::fileSelected,this,&B::f4);    } //构造函数结束

public slots:
    void f1(const QString &path) {
        qDebug()<<"currentChanged";
        qDebug()<<path;
    }
    void f2(const QString &dir) {
        qDebug()<<"directoryEntered";
        qDebug()<<dir;
        qDebug()<<pf->directory().dirName();
        qDebug()<<pf->directory().path();
        qDebug()<<pf->directoryUrl().url();    }
    void f3(const QString &filter) {
        qDebug()<<"filterSelected";
        qDebug()<<filter;
        qDebug()<<pf->filter();
        qDebug()<<pf->selectedNameFilter();
        qDebug()<<pf->mimeTypeFilters();
        qDebug()<<pf->nameFilters();    }
```

```

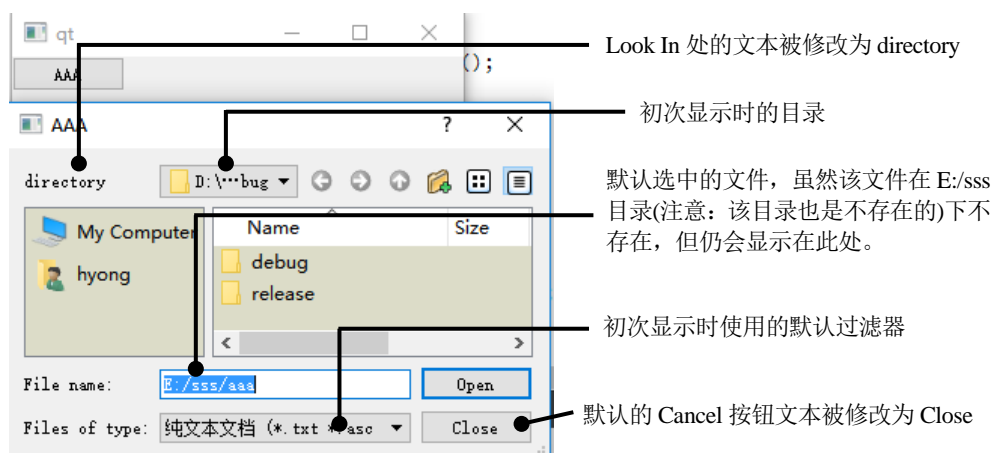
void f4(const QString &file){
    qDebug()<<"fileSelected";
    qDebug()<<file;
    qDebug()<<pf->selectedFiles(); }
#endif // M_H

//m.cpp 文件内容
#include "m.h"
int main(int argc, char *argv[]){    QApplication aa(argc, argv);
    B w;    w.resize(300,200);    w.show();    return aa.exec(); }

```

运行结果及说明

1、初次显示时的界面



2、选中如右图所示过滤器 程序输出如下图

Files of type: JPEG 图像 (*.jpeg *.jpg *.jpe)

```

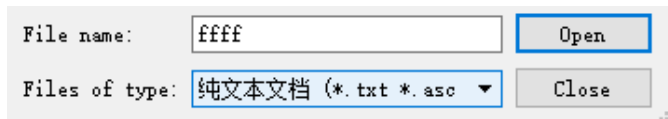
① filterSelected
② "JPEG 图像 (*.jpeg *.jpg *.jpe)"
③ QDir::Filters( Dirs|AllDirs|Files|Drives|NoDot|NoDotDot|AllEntries )
④ "JPEG 图像 (*.jpeg *.jpg *.jpe)"
⑤ ("application/octet-stream", "image/jpeg", "text/plain", "text/html")
⑥ ("All files (*)", "JPEG 图像 (*.jpeg *.jpg *.jpe)", "纯文本文档 (*.txt *.asc *.v)", "HTML 文档 (*.html *.htm)")

```

- ①、只产生了 filterSelected 信号
- ②、filterSelected 信号传递的参数为当前选择的过滤器
- ③、模型过滤器
- ④、selectedNameFilter()函数返回当前选择的过滤器。
- ⑤、mimeTypeFilters()函数返回的是整个文件过滤器的内容。
- ⑥、nameTypeFilters()函数返回的也是整个文件过滤器的内容。

3、在"File name"处输入一个不存在的文件的名称且不指定后缀(如下图), 然后按回车键(或按

下 Open 按钮)



此时会调用 f2 函数，输出结果如下图

```
fileSelected  
"./ffff.XXX"  
("./ffff.XXX")
```

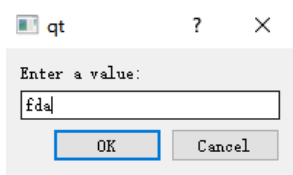
本例指定了 defaultSuffix 属性，因此获得的文件增加了
后缀名 XXX

4、每次改变对文件的选择时会调用 f1()槽函数，每次进入文件夹时会调用 f2()槽函数。

6.7 QInputDialog 类(输入对话框)

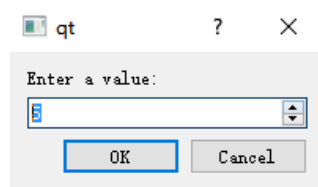
一、输入对话框基础

- 1、输入对话框主要用于向用户提供一个对话框，以获取来自用户输入的值，该对话框也是 Qt 预定义的标准对话框。
- 2、用户输入的值可以是字符串、数字或其他内容，通常对话框的标签会告诉用户应输入什么类型的值。
- 3、输入对话框的类型、样式及创建方法，见下图



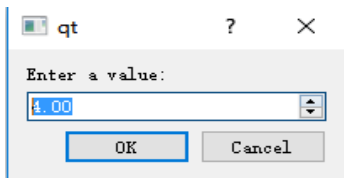
单行文本输入对话框

创建方法: `QInputDialog pi;`
或: `QInputDialog::getText(0,"qt",`
`"Enter a value");`



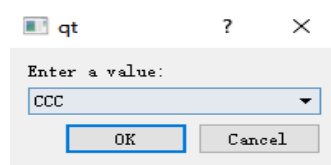
整数输入对话框

```
QInputDialog pi;  
pi.setInputMode(QInputDialog::IntInput);  
或: QInputDialog::getInt(0,"qt",  
                           "Enter a value");
```



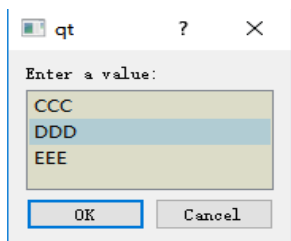
浮点数输入对话框

```
QInputDialog pi;  
pi.setInputMode(QInputDialog::DoubleInput);  
或: QInputDialog::getDouble(0,"qt",  
                             "Enter a value");
```



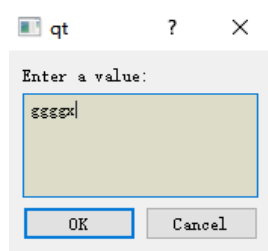
下拉列表(组合框)输入对话框

```
QInputDialog pi;  
QStringList s; s<<"CCC"<<"DDD";  
pi.setComboBoxItems (s);  
或: QInputDialog::getItem(0,"qt",  
                           "Enter a value", s);
```



列表输入对话框

```
QInputDialog pi;
QStringList s; s<<"CCC"<<"DDD";
pi.setComboBoxItems(s);
pi.setOption(QInputDialog::
    UseListViewForComboBoxItems);
无静态函数创建此类型对话框
```



多行文本输入对话框

```
QInputDialog pi;
pi.setOption(QInputDialog::
    UsePlainTextEditForTextInput);
或: QInputDialog::getMultiLineText(0,"qt",
    "Enter a value");
```

4、除列表输入对话框和多行文本输入对话框可调整大小外，其余类型的对话框的高度默认具有固定的大小，不可调整大小，但宽度可调整大小。

二、QInputDialog 类中的属性

1、各类型输入对话框的共同属性

①、cancelButtonText: QString

访问函数: QString cancelButtonText() const; void setCancelButtonText(const QString&);

okButtonText: QString

访问函数: QString okButtonText() const; void setOkButtonText(const QString&);

以上属性表示，设置或获取 Cancel 按钮和 OK 按钮的文本。

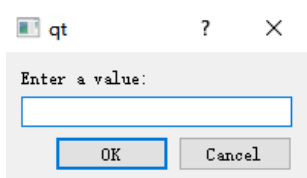
②、labelText: QString

访问函数: QString labelText() const; void setLabelText(const QString&);

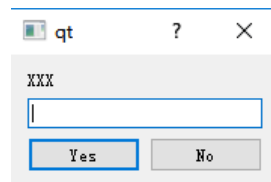
设置或获取提示用户输入内容的标签的文本。

示例:

```
QInputDialog *pil=new QInputDialog;
pil->setOkButtonText("Yes");
pil->setCancelButtonText("No");
pil->setLabelText("XXX");
pil->show();
```



默认输入对话框的形式



使用以上代码修改后的样式

③、inputMode: InputMode

访问函数：InputMode inputMode() const; void setInputMode(InputMode);
获取和设置输入对话框的类型，其中 InputMode 枚举见下表

QInputDialog::InputMode 枚举(无标志)		
作用：描述输入对话框的类型		
成员	值	说明
QInputDialog::TextInput	0	单行文本输入对话框
QInputDialog::IntInput	1	整数输入对话框
QInputDialog::DoubleInput	2	浮点数输入对话框

④、options: InputDialogOptions

访问函数：InputDialogOptions options() const; void setOptions(InputDialogOptions);
获取和设置输入对话框的外观及另两种类型，默认所有选项都被禁用。其中 InputDialogOption 枚举见下表

QInputDialog::InputDialogOption 枚举		
标志：QInputDialog::InputDialogOptions		
作用：描述输入对话框的外观及另两种类型		
成员	值	说明
QInputDialog::NoButtons	0x0000 0001	不显示 Ok 和 Cancel 按钮。
QInputDialog::UseListViewForComboBoxItems	0x0000 0002	使用 QListView(即列表形式)来显示由 setComboBoxItems()设置的不可编辑的下拉列表对话框，注意：若下拉列表是可编辑的，则此设置无效。
QInputDialog::UsePlainTextEditForTextInput	0x0000 0004	使用 QPlainTextEdit 代替单行文本对话框以使其可进行多行文本输入，也就是说该设置对于整数输入对话框、浮点数输入对话框、下拉列表输入对话框是无效的。qt5.2

2、设置文本输入对话框的属性(以下属性仅对 QInputDialog::TextInput 类型的对话框有意义)

⑤、textEchoMode: QLineEdit::EchoMode

访问函数：QLineEdit::EchoMode textEchoMode() const;void setTextEchoMode(QLineEdit::EchoMode);
获取和设置输入对话框的回显模式。其中 QLineEdit::EchoMode 枚举见 QLineEdit 类(第 4 章)。

⑥、textValue: QString

访问函数：QString textValue() const; void setTextValue(const QString&);
信号：textValueChanged(const QString&);
获取和设置输入对话框的文本值。

3、设置整数输入对话框的属性(以下属性仅对 QInputDialog::IntInput 类型的对话框有意义)

- ⑦、intMaximum: int 访问函数：int intMaximum() const; void setIntMaximum(int);
- intMinimum: int 访问函数：int intMinimum() const; void setIntMinimum(int);

以上属性描述输入对话框能接受的最大和最小整数值。

- ⑧、**intStep**: int **访问函数**: int intStep() const; void setIntStep(int);
设置整数值增长或减少的步长。
- ⑨、**intValue**: int **访问函数**: int intValue() const; void setIntValue(int);
信号: intValueChanged(int);
获取和设置输入对话框中的整数值。

4、设置浮点数输入对话框属性(以下属性仅对 **QInputDialog::DoubleInput** 类型对话框有意义)

- ⑩、**doubleMinimum**: double
访问函数: double doubleMinimum() const; void setDoubleMinimum(double);
doubleMaximum: double
访问函数: double doubleMaximum() const; void setDoubleMaximum(double);
以上属性描述输入对话框的最大和最小浮点数值(包括整数部分和小数部分在内)。
 - ⑪、**doubleDecimals**: int
访问函数: int doubleDecimals() const; void setDoubleDecimals(int);
获取和设置输入对话框中的浮点数的精度(这里指小数的位数)。
 - ⑫、**doubleStep**: double **访问函数**: double doubleStep() const; void setDoubleStep(double); //qt5.10
设置浮点数值增长或减少的步长。
 - ⑬、**doubleValue**: int
访问函数: double doubleValue() const; void setDoubleValue(double);
信号: doubleValueChanged(double);
获取和设置输入对话框中的浮点数值。
- 5、设置下拉列表输入对话框属性(以下属性可用于创建下拉列表输入对话框)
- ⑭、**comboBoxItems**: QStringList
访问函数: QStringList comboBoxItems() const; void setComboBoxItems(const QStringList&);
获取和设置下拉列表输入对话框中的下拉列表中的项目。
 - ⑮、**comboBoxEditable**: bool
访问函数: bool isComboBoxEditable() const; void setComboBoxEditable(bool);
下拉列表输入框中的下拉列表是否可编辑。

三、QInputDialog 类中的函数

- ①、**QInputDialog**(QWidget* parent = Q_NULLPTR, Qt::WindowFlags flags = Qt::WindowFlags());
- ②、static double **getDouble**(QWidget* parent, const QString &title, const QString &label, double value = 0,
double min = -2147483647, double max = 2147483647, int decimals = 1,
bool *ok = Q_NULLPTR, Qt::WindowFlags flags = Qt::WindowFlags()); //静态的
static double **getDouble**(QWidget* parent, const QString &title, const QString &label, double value,
double min, double max, int decimals, bool *ok,
Qt::WindowFlags flags, double step); //静态的
static int **getInt**(QWidget* parent, const QString &title, const QString &label, int value = 0,
int min = -2147483647, int max = 2147483647, int step = 1,
bool *ok = Q_NULLPTR, Qt::WindowFlags flags = Qt::WindowFlags()); //静态的

```

static QString getItem(QWidget* parent , const QString &title, const QString &label,
                        const QStringList &items, int current = 0, bool editable = true,
                        bool* ok = Q_NULLPTR, Qt::WindowFlags flags = Qt::WindowFlags(),
                        Qt::InputMethodHints inputMethodHints = Qt::ImhNone); //静态的

static QString getMultiLineText(QWidget* parent , const QString &title, const QString &label,
                                const QString &text = QString(), bool* ok = Q_NULLPTR,
                                Qt::WindowFlags flags = Qt::WindowFlags(),
                                Qt::InputMethodHints inputMethodHints = Qt::ImhNone); //静态的, qt5.2

static QString getText(QWidget* parent , const QString &title, const QString &label,
                        QLineEdit::EchoMode mode = QLineEdit::Normal,
                        const QString &text = QString(), bool* ok = Q_NULLPTR,
                        Qt::WindowFlags flags = Qt::WindowFlags(),
                        Qt::InputMethodHints inputMethodHints = Qt::ImhNone); //静态的

```

以上静态函数分别用于快速创建浮点数、整数、下拉列表、多文本、单文本输入对话框。各参数的意义如下：

- **parent**: 父部件。**title**: 对话框的标题。**flags**: 窗口标志。
- **label**: 对话框的输入标签(提示用户应输入什么文字)。
- **value**: 整数和浮点数输入对话框中的默认值
- **min**、**max**: 对话框中允许的最小、最大值。
- **decimals**: 浮点数的精度(小数位数)。
- **ok**: 若用户点击了对话框中的 **Ok** 按钮, 则*ok 为 true, 若点击的是 **Cancel** 则*ok 为 false。
- **step**: 数值增长或减少的步长
- **items**: 下拉列表对话框中的元素
- **current**: 下拉列表对话框当前显示的默认项目(以索引指定项目)
- **editable**: 下拉列表对话框是否可编辑。
- **text**: 文本输入对话框中显示的默认文本。
- **mode**: 单行文本输入对话框中文本的回显模式, 取值详见 **QLineEdit** 类(第 4 章)。
- **inputMethodHints**: 输入法枚举标志(**Qt::InputMethodHint** 枚举详见相关章节)。

各函数返回值规则如下：

- **getDouble()**和 **getInt()**: 若点击 **Ok** 按钮, 则返回对话框中的当前值, 若点击的是 **Cancel** 按钮, 则返回默认值。
- **getItem()**: 始终返回对话框当前项目的文本。
- **getText()**和 **getMultiLineText()**: 若点击 **Ok** 按钮, 则返回对话框中的当前文本, 若点击的是 **Cancel** 按钮, 则返回一个空字符串。

③、void **open**(QObject* receiver, const char* member);

显示输入对话框, 并把信号连接到由 **receiver** 和 **member** 指定的槽, 关闭对话框时, 信号和槽将被断开。具体发送的信号, 其规则如下：

若 **member** 的第一个参数是 **QString**, 则发送 **textValueSelected()**信号

若 **member** 的第一个参数是 **int**, 则发送 **intValueSelected()**信号

若 member 的第一个参数是 double，则发送 doubleValueSelected()信号
若 member 没有参数，则发送 accepted()信号

④、void **setDoubleRange**(double min, double max);

void **setIntRange**(int min, int max);

以上函数分别用于设置浮点数和整数的范围，min 和 max 分别为最小值和最大值。

⑤、void **setOption**(InputDialogOption option, bool on = true);

bool **testOption**(InputDialogOption option) const;

以上函数是对属性 InputDialogOption 的设置。

四、QInputDialog 类中的信号

1、以下信号仅对 QInputDialog::TextInput 类型的对话框有意义

①、void **textValueChanged** (const QString &text); //信号

只要输入框中的文本改变就会发送此信号

②、void **textValueSelected** (const QString &text); //信号

当用户确定选择输入框中的文本时(比如点击确定按钮)，发送此信号。

2、以下信号仅对 QInputDialog::IntInput 类型的对话框有意义

③、void **intValueChanged** (int value); //信号

只要输入框中的整数值改变就会发送此信号

④、void **intValueSelected** (int value); //信号

当用户确定选择输入框中的数值时(比如点击确定按钮)，发送此信号。

3、以下信号仅对 QInputDialog::DoubleInput 类型对话框有意义

⑤、void **doubleValueChanged**(double value); //信号

只要输入框中的浮点数值改变就会发送此信号

⑥、void **doubleValueSelected**(double value); //信号

当用户确定选择输入框中的数值时(比如点击确定按钮)，发送此信号。

6.8 QProgressDialog 类(进度对话框)

和 QProgressBar(进度条)

QProgressBar 继承自 QWidget

一、进度条原理

- 1、进度条 QProgressBar 的实现原理：首先，需要给进度条指定最小和最大值(即进度条表示的范围)，然后为进度条设置一个当前值，当进度条显示时，Qt 会使进度条显示为当前值与进度条范围的百分比，百分比的计算方式为：

$$(\text{当前值}-\text{最小值})/(\text{最大值}-\text{最小值})$$

比如，进度条的最小值为 0，最大值为 100，当前值为 20，则显示的效果如下图所示



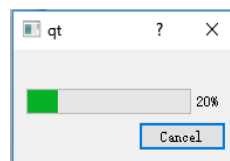
- 2、进度条的显示原理：进度条通常是动态的从 0% 显示到 100%，在这之间进度条会不断更新。而设置进度条的当前值，只能使进度条显示在一个位置，而不会自动增长其值，因此要使进度条从 0% 显示到 100%，需要不断的为进度条设置当前值，通常的方法是使用循环语句，也可使用计时器让其每隔一定时间自动增长，因此，显示进度条的代码通常为

```
for(int i = min; i < max; i++) {设置进度条的当前值;}
```

以上语句 min 表示进度条的最小值，max 表示进度条的最大值。使用以上语句便可使进度条从 0% 动态显示到 100%，在这期间 Qt 会自动更新进度条，我们不需担心更新以及更新时的闪烁问题。

二、QProgressDialog 类(进度对话框)

- 1、进度对话框主要用于向用户反应当前操作进度的对话框(见右图)，该对话框也是 Qt 预定义的标准对话框。
- 2、进度对话框包含一个进度条(QProgressBar)，以及其他一些相关的子部件。
- 3、Qt 实现的进度对话框的原理及执行过程



- ①、QProgressDialog 的构造函数，会启动一个默认为 4000 毫秒的计时器，当计时器超时，会显示该进度对话框，也就是说只要创建了 QProgressDialog 对象，即使程序中没有显示该对话框的 show() 语句(或类似语句)，该对话框 4 秒之后仍会被显示出来。因此使用进度对话框时，不需要显示的调用 show() 语句(或类似语句)显示该对话框，程序会在超时之后自动显示。
- ②、重置计时器：QProgressDialog::setValue() 函数可以重新启动计时器，其规则如下：
 - 需使用 0 或 QProgressDialog::minimum() 调用 setValue() 函数。
 - 此时使用 setMinimumDuration() 函数设置的时间才能作为计时器的超时时间，也就是说该函数此时才起作用，否则该函数不起作用。

示例：

```
QProgressDialog *pp = new QProgressDialog;
pp->setMinimumDuration(10000); //使对话框在 10 秒之后显示
pp->setValue(0); //必须使用此步骤，否则 setMinimumDuration()函数设置的值将不起作用。
//不需使用 pp->show()或类似语句来显示对话框 pp。
```

以上程序会在 10 秒之后自动显示一个进度为 0%的进度对话框。

- ③、若进度对话框的完成时间比超时时间更短，则进度对话框不会被显示。比如

```
QProgressDialog *pp = new QProgressDialog;
for(int i=0; i<1001;i++) pp->setValue(i*100/1000); //对话框不会显示，很明显计算机计算 1001 次循环
不需要 4 秒时间，因此最终该对话框未被显示出来。
```

4、进度对话框的一些其他规则

- ①、基本规则：进度对话框在操作完成时(即当前值等于最大值或显示为 100%时)，默认会自动重置并且隐藏对话框。隐藏和重置由 `autoClose` 和 `autoReset` 属性控制(详见下文)
- ②、若指定的新的当前值超出了进度条的最大或最小值，则对之前设置的当前值没有影响。若当前值超出了进度条新设置的最大或最小值，则重置进度条。这两条规则对设置最大最小值和当前值的顺序作了规定，不同顺序会产生不同影响。

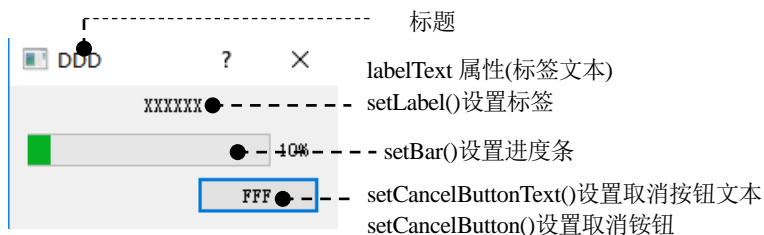
示例：

```
QProgressDialog *pp = new QProgressDialog; //默认最小值为 0，最大值为 100。
pp->setValue(80);
pp->setValue(222); //当前值仍为 80。
pp->setMaximum(40); //当前值 80 大于新设置的最大值 40，重置进度条为 0。
```

- ③、若重新设置最大值为当前值，则进度对话框不会被隐藏。此规则规定了设置最大值应在设置当前值之前。
- ④、若进度条的最大和最小值都设置为 0，则进度条不再显示百分比，而会指示一个繁忙的状态，见下图。



5、下图为 Qt 对进度对话框外观进行的描述



6、QProgressDialog 类中的属性

- ①、**autoClose**: bool 访问函数: bool autoClose() const; void setAutoClose(bool);
指示 reset()函数是否隐藏对话框，若 autoClose()为 true，则 reset()函数会隐藏对话框，若为 false 则不会隐藏对话框，默认为 true。
- ②、**autoReset**: bool 访问函数: bool autoReset() const; void setAutoReset(bool);

进度对话框是否在 `value()` 等于 `maximum` 时立即调用 `reset()`, 若为 `true` 则立即调用 `reset()` 函数, 若为 `false` 则不调用 `reset()` 函数。默认为 `true`。

以上两属性可对进度对话框实现如下操作

- `autoClose = false; autoReset = true;` 不隐藏但重置对话框
- `autoClose = false; autoReset = false;` 既不隐藏也不重置对话框
- `autoClose = true; autoReset = false;` 因为未调用 `reset()` 函数, 所以既不重置也不隐藏对话框。也就是说不能设置为既隐藏对话框而又不重置对话框。
- `autoClose = true; autoReset = true;` 隐藏且重置对话框。

- ③、**labelText**: `QString` **访问函数**: `QString labelText() const; void setLabelText(const QString&);`
获取和设置进度对话框的标签文本。默认为空字符串。
- ④、**maximum**: `int` **访问函数**: `int maximum() const; void setMaximum(int);`
获取和设置进度条所表示的最大值。默认为 100。
- ⑤、**minimum**: `int` **访问函数**: `int minimum() const; void setMinimum(int);`
获取和设置进度条所表示的最小值。默认为 0。
- ⑥、**value**: `int` **访问函数**: `int value() const; void setValue(int);`
获取和设置进度对话框的当前值。注意: 若进度对话框是模态的, 则 `setValue()` 调用 `QApplication::processEvents()`, 因此不要在 `paintEvent()` 函数中使用进度对话框。
- ⑦、**wasCanceled**: `const bool` **访问函数**: `bool wasCanceled() const;`
若进度对话框已被取消则为 1, 否则为 0。若该属性为 1, 则直到进度对话框被重置(即调用 `reset()` 函数)为止都会一直为 1。
- ⑧、**minimumDuration**: `int` **访问函数**: `int minimumDuration() const; void setMinimumDuration(int);`
显示进度对话框之前需经过的时间, 若为 0, 则会立即显示进度对话框, 默认为 4000。

7、QProgressDialog 类中的函数

- ①、**QProgressDialog**(`QWidget* parent = Q_NULLPTR, Qt::WindowFlags f = Qt::WindowFlags();`
QProgressDialog(`const QString &labelText, const QString &cancelButtonText, int min,`
`int max, QWidget* parent = Q_NULLPTR, Qt::WindowFlags f = Qt::WindowFlags());`
以上为构造函数, 各参数意义如下:
labelText: 进度对话框的标签文本。
cancelButtonText: 取消按钮文本, 若该参数为 `QString()`(注意: 不是空字符串), 则不显示取消按钮。
min 和 **max** 分别表示进度条的最小和最大值。
- ②、**void open**(`QObject* receiver, const char* member`);
显示进度对话框, 并把 `canceled()` 信号连接到由 `receiver` 和 `member` 指定的槽, 关闭对话框时, 信号和槽将被断开。
- ③、**void cancel**(); //槽
隐藏并重置进度对话框, 且设置 `wasCanceled` 属性为 1。
- ④、**void reset**(); //槽
重置进度对话框, 若 `autoClose` 属性为 `true`, 则进度对话框会隐藏, 若该属性为 `false`, 则不会隐藏进度对话框。注意: 该函数会把 `wasCancel` 属性重置为 `false`。
- ⑤、**void setRange**(`int min, int max`); //槽

设置进度条的范围(即最小值和最大值)。

- ⑥、void **setBar**(QProgressBar* bar);

把进度对话框的进度条设置为 bar，注意：进度对话框会获取进度条 bar 的所有权，在必要时会删除进度条，因此不要使用在栈上分配的进度条。

- ⑦、void **setLabel**(QLabel* label);

把进度对话框的标签设置为 label。注意：进度对话框会获取 label 的所有权，在必要时会将其删除，因此不要在栈上分配 label。

- ⑧、void **setCancelButton**(QPushButton* cancelButton);

把进度对话框的取消按钮设置为 cancelButton，若设置为 0，则不会显示取消按钮。注意：进度对话框会获取 cancelButton 的所有权，在必要时会将其删除，因此不要使用在栈上分配的按钮。

- ⑨、void **setCancelButtonText**(const QString &cancelButtonText); //槽

把取消按钮的文本设置为 cancelButtonText，若设置为 QString() (注意：不是空字符串)，则删除取消按钮。

- ⑩、void **forceShow**(); //槽，受保护的

QProgressDialog 的构造函数使用此函数显示进度对话框，注意：该函数不是虚拟的，因此重写该函数并不能改变构造函数的默认行为。

- ⑪、void **canceled**(); //信号

当点击 Cancel 按钮时发送此信号，此信号默认连接到 QProgressDialog::cancel() 槽函数

示例：QProgressDialog 类(进度对话框)

//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>
#include <iostream>
using namespace std;
class B:public QWidget{    Q_OBJECT
public:    QProgressDialog *pp;
    B(QWidget* p=0):QWidget(p) {
        QPushButton *pb=new QPushButton("AAA", this);
        QPushButton *pb1=new QPushButton("cout", this);pb1->move(88, 0);
        QPushButton *pb2=new QPushButton("reset", this);pb2->move(0, 33);
        QPushButton *pb3=new QPushButton("show", this);pb3->move(88, 33);
        pp=new QProgressDialog(this);
        pp->setWindowTitle("DDD");
        pp->setLabelText("Please Wait...");    /*设置进度对话框标签文本，向用户提示该进度条正在做什么。*/
        pp->setAutoClose(0);    //当进度条显示到 100%时不隐藏。
        pp->setAutoReset(0);    //当进度条显示到 100%时不调用 reset() 函数重置进度条。
        pp->setCancelButtonText("No");    //重新设置 Cancel 按钮的文本
        pp->setMinimumDuration(0);    //立即显示对话框。
        pp->setValue(0);    //设置当前值为 0，从而使上一条语句生效，否则上条语句将不会起作用。
        QObject::connect(pb, &QPushButton::clicked, pp, &QProgressDialog::show);
        QObject::connect(pb1, &QPushButton::clicked, this, &B::f);
    }
    //点击按钮 pb2 调用 reset 重置对话框。
};
```

```

        QObject::connect(pb2, &QPushButton::clicked, pp, &QProgressDialog::reset);
        QObject::connect(pb3, &QPushButton::clicked, this, &B::f2);    }

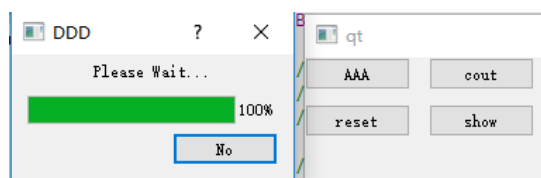
public slots:
    void f() {
        for(int i=0; i<100001; i++) { //使用循环让进度条动态增长, 若执行过快, 可增大循环次数。
            pp->setValue(i*100/100000);
            //QCoreApplication::processEvents(); //若在执行过程中界面有冻结现象, 可调用此函数避免冻结。
            if(pp->wasCanceled()) break; //若用户在中途点击了Cancel 按钮, 则退出循环。
        }
    }

    void f2() { cout<<pp->wasCanceled()<<endl; } };
#endif // M_H

//m.cpp 文件的内容
#include "m.h"
int main(int argc, char *argv[]) {    QApplication aa(argc, argv);
    B w;    w.resize(300, 200);    w.show();    return aa.exec(); }

```

运行结果及说明



程序初次运行时进度对话框会与主窗口同时显示。
 点击 AAA 按钮会使隐藏的进度对话框再次显示。
 点击 cout 按钮, 进度对话框会动态从 0%增长到 100%。
 点击 reset 按钮进度对话框会被重置为 0。
 当击 show 按钮用于测试 wasCanceled()函数返回的值,
 点击 No 按钮或进度对话框右上角的 X 按钮隐藏对话框, 此时会重置进度条为 0, 并使 wasCanceled()函数返回 1, 点击 AAA 使对话框再次显示, 此时可发现 wasCanceled()函数仍返回 1, 点击 reset 按钮重置对话框, 此时才会使 wasCanceled()函数返回 0。

三、QProgressBar 类(进度条)

注: QProgressBar 类继承自 QWidget, 它是一个 QWidget 部件, QProgressBar 除了将其放置于进度对话框之中外, 还可将其放置于窗口的状态栏等其他部件中。

1、QProgressDialog 类中的属性

①、**alignment**: Qt::Alignment

访问函数: Qt::Alignment alignment()const; void setAlignment(Qt::Alignment);
 获取和设置进度条的对齐方式

②、**format**: QString

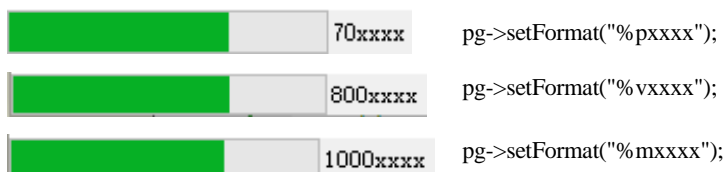
访问函数: QString format() const; void setFormat(constQString&); void resetFormat();
 设置进度条右侧描述性文字的显示格式, 默认为"%p%", 具体规定如下:
 %p: 显示的数值为百分比。

%v: 显示的数值为当前设置的值。

%m: 显示的数值为进度条的范围值。

示例(各情形见图示):

```
QProgressBar *pg=new QProgressBar;  
pg->setMinimum(100);      pg->setMaximum(1100);      pg->setValue(800);
```



③、**invertedAppearance**: bool

访问函数: bool invertedAppearance() const; void setInvertedAppearance(bool);

是否反转进度条, 若该属性为 true, 则进度条会从另一个方向增长(比如从右到左), 默认为 false。

④、**maximum**: int **访问函数**: int maximum() const; void setMaximum(int);

进度条的最大值。默认为 100。

⑤、**minimum**: int **访问函数**: int minimum() const; void setMinimum(int);

进度条的最小值。默认为 0。



当最小值和最大值都为 0 时, 进度条会显示为一个繁忙的状态 (如上图所示)。

若指定的新的当前值超出了进度条的最大或最小值, 则对之前设置的当前值没有影响。

若当前值超出了进度条新设置的最大或最小值, 则重置进度条(示例见 QProgressDialog 类)。

⑥、**value**: int **访问函数**: int value() const; void setValue(int);

信号: valueChanged(int);

获取和设置进度条的当前值, 若指定的当前值超出了进度条的最大或最小值, 则对之前设置的当前值没有影响。

⑦、**orientation**: Qt::Orientation

访问函数: Qt::Orientation orientation() const; void setOrientation(Qt::Orientation);

进度条的方向, 必须是 Qt::Horizontal (水平, 默认) 或 Qt::Vertical (垂直)

⑧、**text**: const QString **访问函数**: virtual QString text() const;

返回进度条右侧的描述性文本, 比如返回"50%"等。

⑨、**textDirection**: Direction

访问函数: QProgressBar::Direction textDirection()const;void setTextDirection(QProgressBar::Direction);

描述垂直进度条描述性文字的阅读方向, 该属于对水平进度条无影响。默认为 QProgressBar::TopToBottom (从上到下)

⑩、**textVisible**: bool **访问函数**: bool isTextVisible() const; void setTextVisible(bool);

描述是否显示进度条的描述性文字(通常显示为百分比), 默认为 true。该属性可能会被样式忽略。

2、QProgressDialog 类中的函数

- ①、**QProgressBar**(QWidget* parent = Q_NULLPTR); //构造函数
 - ②、void **reset**(); //槽，重置进度条
 - ③、void **setRange**(int min, int max); //槽，设置进度条的范围(即最小值和最大值)。
 - ④、void **valueChanged**(int value); //信号
- 当进度条中显示的值发生变化时发送此信号，value 为进度条显示的新值。

作者：黄邦勇帅(原名：黄勇)

2018-4-30

6.9 QWizard 类(向导)和 QWizardPage(向导页)

QWizardPage 继承自 QWidget，因此 QWizardPage 具有普通的 QWidget 部件的性质，可以向其中添加按钮、复选框等其他子部件，也可以作为窗口单独显示，但 QWizardPage 通常是作为 QWizard 类的页面，而添加到 QWizard 中的。

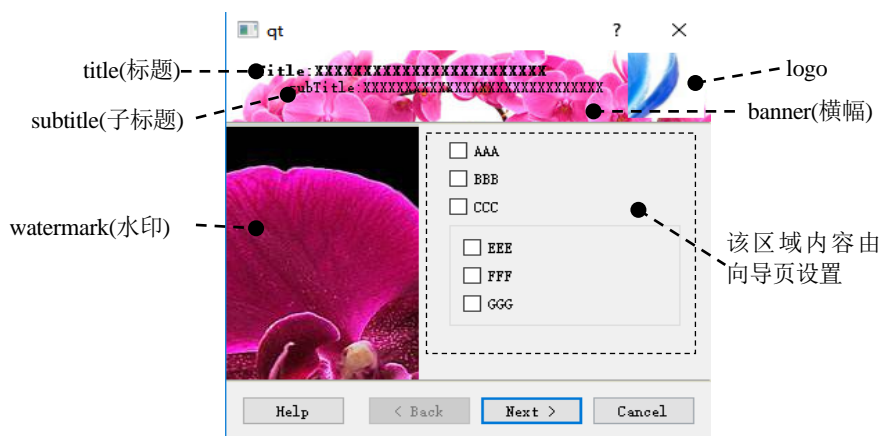
一、向导基础

- 1、向导：向导是由一系列页面组成的对话框，其主要目的是引导用户逐步完成一些复杂的任务。
- 2、向导实现原理：
向导 QWizard 拥有一些共同的外观和特性，向导页 QWizardPage(也称为页面、页)就是一个 QWidget 部件，但该部件可以被添加到 QWizard 之中，同一时刻只有一个向导页被显示，可点击按钮显示向导的上一页或下一页。向导会为每个向导页分配一个从 0 开始的 ID。
- 3、由 QWizard 类设置的项目通常是作用于整个向导的，而使用 QWizardPage 类设置的项目，通常只作用于当前页，比如使用 QWizard 类的 setPixmap() 设置的横幅(banner)图片(作用于整个向导)，当改变页面时，图片不会改变，而使用 QWizardPage 类的相同函数 setPixmap() 设置的横幅(banner)图片(只作用于当前页)，会因当前显示的页面不同，而显示不同的图片。

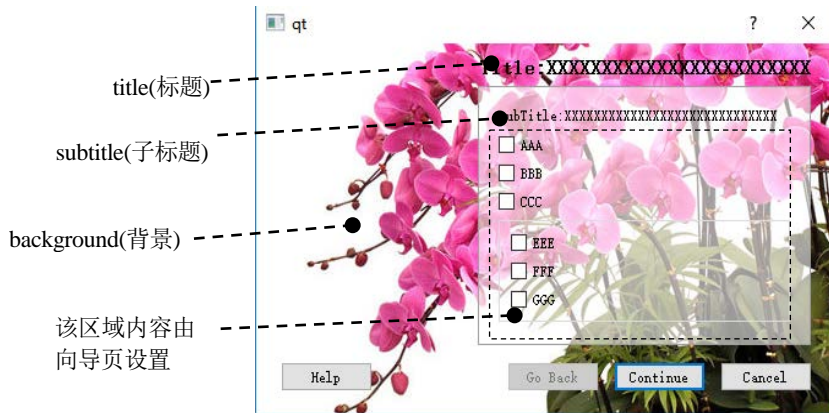
二、向导外观

1、向导的框架样式

下图为 windows 下向导的框架样式



下图为 Mac 下向导的框架样式



2、以下函数或属性用于设置向导框架(函数及属性原型请参阅后文)

- ①、向导样式由 `QWizard::setWizardStyle()` 函数设置，其样式类型由枚举 `QWizard::WizardStyle` 指定(比如 `QWizard::MacStyle` 表示 Mac 样式);
- ②、title: 由 `QWizardPage::setTitle()` 函数设置
- ③、subtitle: 由 `QWizardPage::setSubTitle()` 函数设置，注意：标题和子标题虽然由 `QWizardPage` 设置，但不会显示在 `QWizardPage` 部件上，而是显示在 `QWizard` 部件上面。
- ④、watermark、logo、banner、background: 由 `QWizard` 或 `QWizardPage` 的 `setPixmap()` 函数设置，其类型由 `QWizard::WizardPixmap` 枚举指定(比如 `QWizard::LogoPixmap` 表示 logo)
- ⑤、向导页由 `QWizard::addPage()` 函数添加到向导中。

示例：创建一个简单的向导

```
#include<QtWidgets>
int main(int argc, char *argv[]){    QApplication aa(argc, argv);
    QWizard *pw=new QWizard;          //向导
    QWizardPage *pwg,*pwg1;    //向导页
    pwg=new QWizardPage;    pwg1=new QWizardPage;
    QCheckBox *pc=new QCheckBox("AAA");    QCheckBox *pc1=new QCheckBox("BBB");
    QCheckBox *pc2=new QCheckBox("CCC");    QCheckBox *pc3=new QCheckBox("DDD");
    QCheckBox *pc4=new QCheckBox("EEE");    QCheckBox *pc5=new QCheckBox("FFF");
    QCheckBox *pc6=new QCheckBox("GGG");
    //设置向导的样式，读者可依次验证可样式的外观。
    //pw->setWizardStyle(QWizard::ClassicStyle); //经典 windows 样式
    pw->setWizardStyle(QWizard::ModernStyle);    //现代 windows 样式
    //pw->setWizardStyle(QWizard::MacStyle);    //Mac 样式
    //pw->setWizardStyle(QWizard::AeroStyle);    //Aero 样式(windows vista)

    //设置向导页 pwg 的内容
    //注：标题和子标题虽然是由 QWizardPage 类指定的，但不会在其上显示，只会显示在 QWizard 上。
    pwg->setTitle("Title:XXXXXXXXXXXXXXXXXXXXXXX");    //标题
    pwg->setSubTitle("subTitle:XXXXXXXXXXXXXXXXXXXXXXX");    //子标题
```

```

//向水印区域添加图片
pwg->setPixmap(QWizard::WatermarkPixmap, QPixmap("F:/lvs. png"));
//添加背景图片(仅适用于 Mac 样式)
//pwg->setPixmap(QWizard::BackgroundPixmap, QPixmap("F:/4. jpg"));
//向横幅和 logo 区域添加图片(Mac 样式不适用)
pwg->setPixmap(QWizard::BannerPixmap, QPixmap("F:/4hm. png")); //ModernStyle 样式不适用
pwg->setPixmap(QWizard::LogoPixmap, QPixmap("F:/3i. png"));

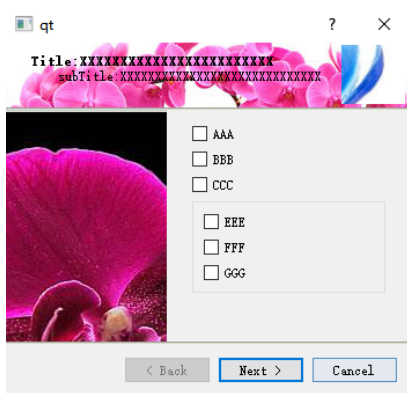
//向向导页 pwg 中添加一些按钮部件。
QGroupBox *pgb=new QGroupBox ;    QVBoxLayout *pv2=new QVBoxLayout;
pv2->addWidget(pc4);    pv2->addWidget(pc5);    pv2->addWidget(pc6);
pgb->setLayout(pv2);
QVBoxLayout *pv=new QVBoxLayout;
pv->addWidget(pc);    pv->addWidget(pc1);    pv->addWidget(pc2);    pv->addWidget(pgb);
pwg->setLayout(pv);

//设置向导页 pwg1 的内容
pwg1->setPixmap(QWizard::WatermarkPixmap, QPixmap("F:/3vs. png"));
pwg1->setTitle("YYYYYYY");    pwg1->setSubTitle("MMMMMMM");
QVBoxLayout *pv1=new QVBoxLayout;    pv1->addWidget(pc3);    pwg1->setLayout(pv1);

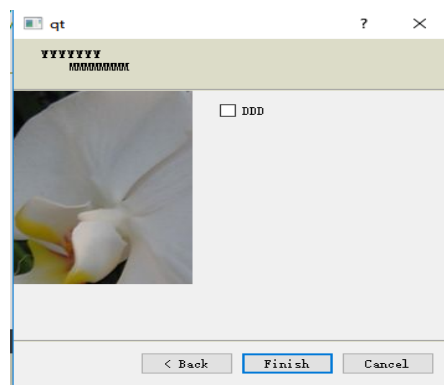
//把向导页添加到向导中, 并显示向导
pw->addPage(pwg);    pw->addPage(pwg1);    pw->show();    return aa.exec(); }

```

运行结果及说明



第一页的内容



点击按钮 Next 后第二页的内容

三、向导中的按钮(所用到的函数及属性原型请参阅后文)

- 1、只能向向导中添加 3 个自定义的按钮, 其余按钮只能使用向导中设置好的按钮。默认情况下, 不显示自定义按钮。
- 2、向导中的按钮由 QWizard::options 和 QWizard::WizardButton 属性描述
- 3、按钮的位置可使用 QWizard::setButtonLayout()函数调整

- 3、要使用自定义按钮需使用 `QWizard::setOption()` 函数使其可见，使用 `QWizard::setButton()` 添加按钮，使用 `QWizard::setButtonText()` 函数 `QWizardPage::setButtonText()` 或设置其文本
- 4、默认情况下 Next 和 Finish 按钮是互斥的，同一时刻要么是 Next 按钮，要么是 Finish 按钮。

示例：向导中的按钮

```
#include<QtWidgets>
int main(int argc, char *argv[]) {    QApplication aa(argc, argv);
    QWizard pw;                        QWizardPage *pwg=new QWizardPage;
    QWizardPage *pwg1=new QWizardPage;    QWizardPage *pwg2=new QWizardPage;
    QPushButton *pb=new QPushButton("AAA");    QPushButton *pb1=new QPushButton("BBB");
    QRadioButton *pr=new QRadioButton("DDD");    QCheckBox *pc=new QCheckBox("EEE");
    QCheckBox *pc1=new QCheckBox("Page 1", pwg);    QCheckBox *pc2=new QCheckBox("Page 2", pwg1);
    QCheckBox *pc3=new QCheckBox("Page 3", pwg2);
    pw.setWizardStyle(QWizard::ModernStyle); //本示例使用现代 Windows 样式。
    //pw.setWizardStyle(QWizard::MacStyle);
    //添加 3 个自定义按钮(最多只能添加 3 个自定义按钮)
    pw.setButton(QWizard::CustomButton1, pb);    pw.setButton(QWizard::CustomButton2, pr);
    pw.setButton(QWizard::CustomButton3, pc);
    //使用按钮 pb1 替换默认的 Cancel 按钮
    pw.setButton(QWizard::CancelButton, pb1);
    //使自定义按钮可见，否则自定义按钮不可见
    pw.setOption(QWizard::HaveCustomButton1);    pw.setOption(QWizard::HaveCustomButton2);
    pw.setOption(QWizard::HaveCustomButton3);
    //把 Cancel 按钮的文本修改为 No(使用 QWizard 类设置的文本会作用于所有页面)
    pw.setButtonText(QWizard::CancelButton, "No");
    //修改 Next 按钮的文本，使用 QWizardPage 类设置的文本只会作用于当前页
    pwg->setButtonText(QWizard::NextButton, "Page 1");
    pwg1->setButtonText(QWizard::NextButton, "Page 2");
    pw.setOption(QWizard::HaveNextButtonOnLastPage); //使最后一页显示 Next 按钮
    pwg2->setButtonText(QWizard::NextButton, "Page End");
    //自定义按钮的文本也需使用 setButtonText 函数修改
    pw.setButtonText(QWizard::CustomButton1, "Start");
    pwg2->setButtonText(QWizard::CustomButton3, "End");
    //使用以下方式不能成功修改按钮的文本。
    QAbstractButton *pbb=pw.button(QWizard::NextButton);
    pbb->setText("Up");
    //被添加到向导之后，即使自定义按钮也不能使用 setTextButton() 之外的函数修改文本
    pr->setText("JJJJ"); //不会使 pr 的文本修改为 JJJ
    //把向导页添加到向导中
    pw.addPage(pwg);    pw.addPage(pwg1);    pw.addPage(pwg2);

    //修改按钮的默认显示顺序，其顺序依按钮添加到列表中的顺序显示。
    //注意：setButtonLayout() 是最终的设置按钮的函数，若未出现在该函数参数的列表中的按钮将不会被显示，在列表中增加的按钮会被显示。*/
    QList<QWizard::WizardButton> y;
    y<<QWizard::CancelButton    //使用<<运算符向列表添加元素。
    <<QWizard::HelpButton    //此步骤会使 Help 按钮可见
    <<QWizard::FinishButton
    <<QWizard::Stretch    //添加一个空白拉伸间隙
    <<QWizard::NextButton
    <<QWizard::BackButton
```

```

<<QWizard::Stretch          //添加一个空白拉伸间隙
<<QWizard::CustomButton3
<<QWizard::CustomButton2
<<QWizard::CustomButton1;
pw.setButtonLayout(y);      //设置向导中按钮的顺序
pw.show();    return aa.exec(); }

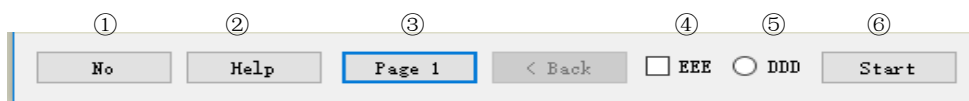
```

运行结果及说明

1、未使用 setButtonLayout()函数设置按钮位置时的默认顺序



2、使用 setButtonLayout()函数更改按钮位置后的顺序



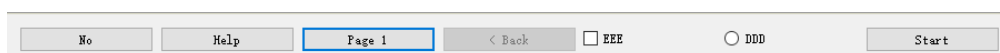
- ①、Cancel 按钮被修改为了文本 No
- ②、Help 按钮被使用 setButtonLayout()添加到了向导中
- ③、第一页的 Next 按钮被修改为了文本 Page 1
- ④、⑤、自定义按钮 3 和自定义按钮 2
- ⑥、自定义按钮 1 的文本被修改为了 Start

3、点击 Page 1 和之后的 Page2 进入到最后一页，并拉伸向导后的按钮情形



- ①、最后一页会显示 Finish 按钮
- ②、④、是添加到向导中的两个空白间隙，在向导拉伸时，这个间隙会占据多余的空间，若未添加空白间隙，则各按钮都会被拉伸以占据多余的空间，见下图。
- ③、默认情况下，最后一个不会显示 Next 按钮(即 Page End 按钮)，使用 options 属性使其显示，但其状态为禁用的。
- 由⑤和③可以看到，使用 QWizard::setButtonText()函数修改的 Cancele 按钮文本 No，是作用于整个向导的，在切换到其他页面时，该按钮的文本不会改变，而使用 QWizardPage::setButtonText()函数修改的 Next 按钮文本，只作用于当前页，当显示另一页时其文本可能会不相同，本例显示第一页时，Next 按钮为 Page1，第二页为 Page 2，第三页(最后一页)为 Page End

4、未添加任何拉伸空白间隙的效果，即未向 setButtonLayout()的参数的列表加添加任何的 QWizard::Stretch 枚举。由图可见，当拉伸向导时，各按钮都会被拉伸，



四、向导中的页面(所用到的函数及属性原型请参阅后文)

- 1、提交页：是指不能通过单点 Back 或 Cancel 来撤消的页面，在提交页面上使用 Commit 按钮替换 Next 按钮，单击 Commit 按钮就像单击 Next 一样。直接点击提交页上 Commit 按钮进入下一页，则下一页的 Back 按钮是被禁用的。
- 2、最终页面：就是指的向导的最后一页，在最后一页默认会显示 Finish 按钮，此时 QWizard 和 QWizardPage 的 nextId() 返回-1。
- 3、使用 QWizardPage::isCommitPage() 函数可判断该页是否是提交页，使用 QWizardPage::setCommitPage() 函数把当前页设置为提交页。
- 4、使用 QWizardPage::setFinalPage() 把当前页设置为最终页从而可以使用户提前完成向导，使用 QWizardPage::isFinalPage 可判断该页是否是最终页。
- 5、重新实现 QWizardPage::isComplete() 虚函数可用于确定是否启用 Next 或 Finish 按钮，该函数返回 true 则启用，返回 false 则禁用。

示例：提交页、最终页及控制 Finish 和 Next 按钮的禁用/启用状态 (isComplete 虚函数)

//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>
class D:public QWizardPage{    Q_OBJECT
public:    D(QWidget* p=0):QWizardPage(p) {}
    bool isComplete()const{ //重新实现该虚函数，以控制 Finish 和 Next 按钮的禁用/启用状态
        QWizard *p=wizard();
        QList<int> t=p->pageIds(); //使用列表 t.size(), 间接获取页面的数量
        //若当前页是倒数第二页，则禁用 Next 和 Finish 按钮
        if(p->currentId()==t.size()-2){return 0;}
        return 1;    }; //D 结束
#endif // M_H
```

//m.cpp 文件的内容

```
#include "m.h"
int main(int argc, char *argv[]){    QApplication aa(argc,argv);
    QWizard pw;
    D *pwg1=new D;    D *pwg2=new D;    D *pwg3=new D;    D *pwg4=new D;
    D *pwg5=new D;    D *pwg6=new D;
    QPushButton *pb=new QPushButton("Up");    QPushButton *pbl=new QPushButton("Dwon");
    QCheckBox *pc1=new QCheckBox("Page 1",pwg1); QCheckBox *pc2=new QCheckBox("Page 2",pwg2);
    QCheckBox *pc3=new QCheckBox("Page 3",pwg3); QCheckBox *pc4=new QCheckBox("Page 4",pwg4);
    QCheckBox *pc5=new QCheckBox("Page 5",pwg5); QCheckBox *pc6=new QCheckBox("Page 6",pwg6);

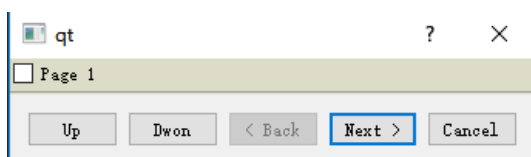
    pw.setWizardStyle(QWizard::ModernStyle); //本示例使用现代 Windows 样式。
    //添加自定义按钮，以用于以编程的方式显示上一页和下一页
    pw.setButton(QWizard::CustomButton1, pb);    pw.setOption(QWizard::HaveCustomButton1);
    pw.setButton(QWizard::CustomButton2, pbl);    pw.setOption(QWizard::HaveCustomButton2);
    //把向导页添加到向导中
    pw.addPage(pwg1);    pw.addPage(pwg2);    pw.addPage(pwg3);
    pw.addPage(pwg4);    pw.addPage(pwg5);    pw.addPage(pwg6);
    /
    //设置 pwg2 为提交页
    pwg2->setCommitPage(true);
```

```

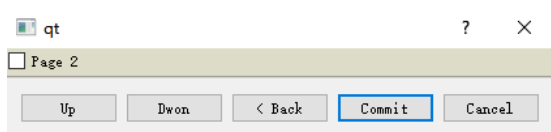
//使 Finish 按钮被显示, 该按钮是否可用取决于 isComplete() 函数的返回值
pwg4->setFinalPage(true); //本例此页显示的 Finish 按钮可用
pwg5->setFinalPage(true); //本例此页显示的 Finish 按钮和 Next 按钮均不可用
//点击按钮 pb 时, 显示上一页
QObject::connect(pb, &QPushButton::clicked, &pw, &QWizard::back);
//点击按钮 pb1 时, 显示下一页
QObject::connect(pb1, &QPushButton::clicked, &pw, &QWizard::next);
pw.show(); return aa.exec(); }

```

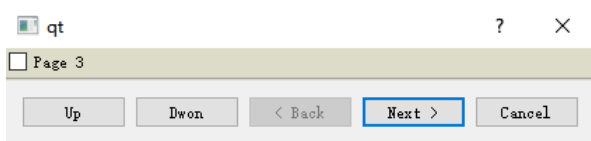
运行结果及说明



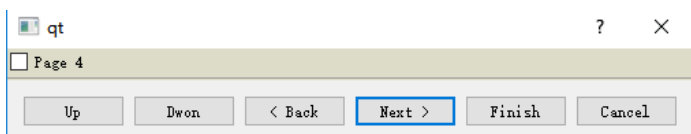
最初显示时的状态



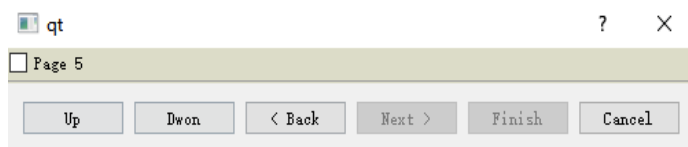
第 2 页的状态, 可看到该页的 Next 按钮被替换为了 Commit 按钮



点击第 2 页的 Commit 按钮(该按钮就像 Next 一样)进入第 3 页的状态, 可看到, 第 3 页的 Back 按钮被禁用了, 但仍可点击 Up 按钮以编程的方式进入上一页



第 4 页的状态, 可见, 第 4 页多了一个 Finish 按钮, 点击该按钮可提前结束向导



第 5 页的状态, 可见, 第 5 页的 Finish 和 Next 按钮都被禁用了(因为此时 isComplete()返回 false), 但仍可点击 Down 按钮以编程的方式进入下一页



第 6 页的状态

五、验证页面中的内容(所用到的函数及属性原型请参阅后文)

- 1、页面内容的验证: 向导通常会要求用户输入一些信息, 并验证用户输入信息的合法性, 比如, 若输入合法则可进行下一页, 否则向用户发出一个错误提示信息等。

- 2、对用户输入的验证，可通过重新实现 `QWizard::validateCurrentPage()` 或 `QWizardPage::validatePage()` 虚函数来实现，由于 `QWizard::validateCurrentPage()` 默认实现是调用的 `QWizardPage::validatePage()` 虚函数，因此，只需重新实现 `QWizardPage::validatePage()` 虚函数即可。此函数的验证时机为，当用户单击 Next 或 Finish 按钮时。
- 3、重新实现 `QWizardPage::isComplete()` 虚函数，以控制对 Next 或 Finish 按钮的启用/禁用状态，也可达到验证用户输入的目的，且效果比重新实现 `QWizardPage::validatePage()` 虚函数更好。若重新实现该函数，当页面变得完整或不完整时应在合适的地方发送 `QWizardPage::completeChanged()` 信号(可把另一个信号连接到该信号来发送该信号)，以更新按钮的启用/禁用状态。注：发送 `completeChanged` 信号会导致 `isComplete()` 函数被调用。
- 4、由以上可知 `isComplete()` 函数先于 `validatePage()` 函数调用。比如对于页面中的 `QLineEdit` 部件，若用户未输入任何内容时，可使用 `isComplete()` 函数使 Next 按钮不可用，当用户最终输入完内容点击 Next 按钮时，再使用 `validatePage()` 函数进行验证。

示例：验证用户输入的内容(`isComplete` 和 `validatePage` 函数)

//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>
class D:public QWizardPage{    Q_OBJECT    //子类化 QWizardPage
public:    QLabel *p;    QLineEdit *pe;
    D(QWidget* pw=0):QWizardPage(pw) {
        p=new QLabel("input value:",this);
        pe=new QLineEdit(this);pe->move(77, 0);
        /*通过发送 QLineEdit 中的信号而发送 completeChanged 信号，也就是说当用户在 QLineEdit 中按下回车键时会发送 completeChanged 信号。注：returnPressed 信号只有在按下回车键时才会发送*/
        QObject::connect(pe,&QLineEdit::returnPressed,this,&QWizardPage::completeChanged);
    }
    bool isComplete()const {
        //若用户未输入值或输入了"close"则禁用 Finish 和 Next 按钮
        if(pe->text().isEmpty()||pe->text()=="disable next"){
            //emit completeChanged();//错误,因为 isComplete 是 const 函数,只能调用 const 函数。
            return 0;}
        else {        return 1;}
    }
    bool validatePage() { //该虚函数用于在点击 Next 或 Finish 按钮时，对输入的内容进行最后的验证

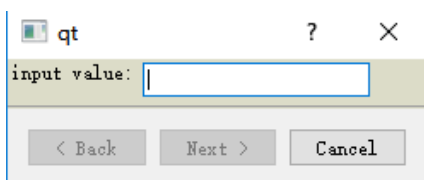
        if(pe->text()=="BBB") { return 1; } //只有用户在输入了文本 BBB 时才能通过验证
        else if(pe->text()=="close") { //若用户输入 close 则禁用 Next 按钮。
            pe->setText("disable next"); //提示用户 Next 按钮会被禁用
            emit completeChanged(); /*发送该信号以使按钮 Next 被禁用。注：发送该信号会导致 isComplete() 函数被调用。*/
            return 0;        }
        else { //对于其他输入，则提示用户输入了错误的值，并使验证无法通过。
            pe->setText("value error,enter again"); //提示用户重新输入。
            //该 else 语句中未发送 completeChanged 信号，因此不会对按钮 Next 的启用/禁用状态作出更新
            return 0;}    } }; //D 结束
#endif // M_H
```


//m.cpp 文件的内容

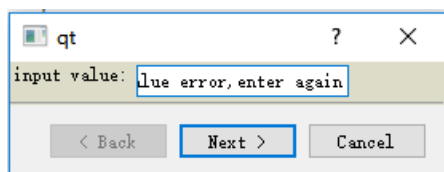
```
#include "m.h"

int main(int argc, char *argv[]) {    QApplication aa(argc, argv);
    QWizard pw;    D *pwgl=new D;    QWizardPage *pwp=new QWizardPage;
    QLineEdit *pe=new QLineEdit(pwp);    pw.setWizardStyle(QWizard::ModernStyle);
    pw.addPage(pwgl);    pw.addPage(pwp);
    pw.show();    return aa.exec(); }
```

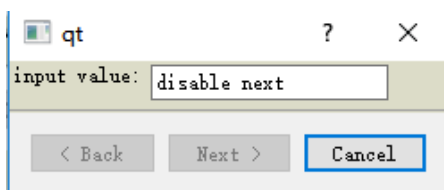
运行结果及说明



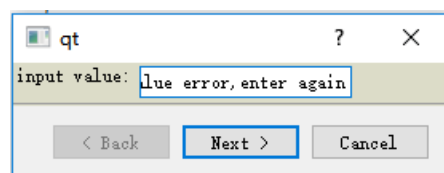
初次运行时按钮 Next 是被禁用的。本示例只有输入 BBB 才能顺利进入下一页。



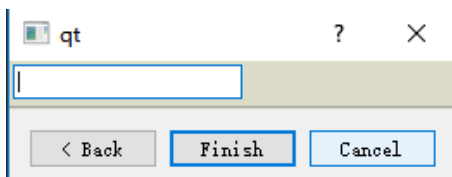
输入除了 BBB 和 close 以外的文本，比如输入 D，然后按下回车之后情形。可见，此时程序提示这是一个错误的值，同时按钮 Next 被启用了。



再次输入文本 close，然后点击 Next 按钮，此时显示提示信息"disable next"，并且 Next 按钮被禁用。



再次输入 D，以启用 Next 按钮，然后在输入除 BBB 和 close 之外的其他任何文本，然后点击 Next 按钮，此时该按钮不会被禁用(只有输入 close 才会禁用该按钮)，并且每次都会提示用户重新输入。



最后输入 BBB，然后点击 Next 按钮，顺利进入下一页。

六、各页面间的通信(字段)(所用到的函数及属性原型请参阅后文)

- 1、字段：在向导中，通常会遇到之前页面的内容会影响到后续页面内容的情况，此时需要在页面间进行通信，QWizard 使用字段用于在各页面间通信。
- 2、在 QWizard 和 QWizardPage 类中，没有用于获取页面之中部件的函数，比如在页面 1 之

中有一个 QLineEdit 部件 pe，在页面 2 中想要使用在 pe 中输入的文本用于设置该页中按钮 pb 的文本，此时就需要得到指向部件 pe 的指针，但在 QWizard 和 QWizardPage 类中没有相应的函数，因此无法获取页面 1 之中的任何部件。对于这种情形的解决方法就是使用字段。

- 3、字段就是一个名称，但该名称附带了一些信息，其他页面可以获取和更改这个字段所附带的信息，从而达到通信的目的。比如字段：("xxx", pe, "text"); 表示，字段名为 xxx，附带的信息是 QLineEdit 部件 pe 的 text 属性，然后其他页面便可通过字段名 xxx 获取到 pe 部件 text 的属性，也可对该属性进行更改。
- 4、对于整个向导来讲，字段是全局的。
- 5、创建字段的步骤：
 - ①、首先需要子类化 QWizardPage，因为需要调用其中的受保护的函数注册字段。
 - ②、调用受保护的函数 QWizardPage::registerField()注册字段
 - ③、注册之后，各页面都可以使用 field()函数获取到字段保存的值，也可使用 setField()重新设置字段的值。注：QWizard 和 QWizardPage 类中都有 field()和 setField()函数。
- 6、initializePage()虚函数：该函数用于在显示页面之前初始化该页面，也就是说在前一页点击 Next 按钮之后，显示该页之前，会调用该函数来对该页面进行一些调置。在 QWizard 和 QWizardPage 类中都有该函数，因为默认实现，在 QWizard 类中是调用的 QWizardPage 类中的 initializePage()虚函数，因此，通常只需重新实现 QWizardPage::initializePage()函数即可，默认情况下 QWizardPage::initializePage()函数什么都没做。

示例：字段和 initializePage() 虚函数的使用

//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>
class D:public QWizardPage{    Q_OBJECT
public:    QLabel *p;    QLineEdit *pe;    QPushButton *pb;
    D(QWidget* pw=0):QWizardPage(pw){
        p=new QLabel("input value:",this);    p->move(0,33);
        pe=new QLineEdit(this);    pe->move(77,33);
        pb=new QPushButton("XXX",this);    pb->move(0,66);
        registerField("AAA",pb,"text");    //注册字段。该字段附加的信息是按钮 pb 的 text 属性
    }    //构造函数结束

public slots: void f(){ setField("AAA",pe->text());} //把 pe 中输入的文本设置为字段 AAA 的值。
    void f1(QString t){ setField("AAA",t); } //把 t 设置为字段 AAA 的值。
};    //D 结束

class D1:public QWizardPage{    Q_OBJECT
public:    QLabel *p;    QLineEdit *pe;
    D1(QWidget* pw=0):QWizardPage(pw){
        p=new QLabel("input value:",this);    p->move(0,33);
        pe=new QLineEdit(this);    pe->move(77,33);
        //setField("AAA","BBB");    //此处对字段的设置不会成功
    }    //构造函数结束

/*initializePage()函数用于在显示该页面之前初始化页面，也就是说在前一页点击 Next 按钮之后，显示
该页之前，会调用该函数。*/
```

```

void initializePage() {
    pe->setText(field("AAA").toString()); } //把 pe 的文本设置为字段 AAA 的值。
}; //D1 结束

//D2 未重新实现 initializePage() 函数，因此在显示该页之前，无法对其进行初始化。
class D2:public QWizardPage{    Q_OBJECT
public:    QLabel *p;    QLineEdit *pe;    QPushButton *pb;
    D2(QWidget* pw=0):QWizardPage(pw) {
        p=new QLabel("input value:",this);        p->move(0,33);
        pe=new QLineEdit(this);        pe->move(77,33);
        pb=new QPushButton("set text",this);        pb->move(0,66);
        pe->setEnabled(0);        //pe 不可被编辑
        //通过点击该页上的按钮 pb 来设置 pe 的文本。
        QObject::connect(pb,&QPushButton::clicked,this,&D2::f); } //构造函数结束
public slots:    void f() {    pe->setText(field("AAA").toString()); } //D2 结束
#endif // M_H

//m.cpp 文件的内容
#include "m.h"
int main(int argc, char *argv[]) {    QApplication aa(argc,argv);
    QWizard pw;    D *pwg1=new D;    D1 *pwg2=new D1;    D2 *pwg3=new D2;
    QWizardPage *pwp=new QWizardPage;    QLineEdit *pe=new QLineEdit(pwp); pe->move(0,33);
    pw.setWizardStyle(QWizard::ModernStyle); //设置向导样式
//添加一个自定义按钮，点击该按钮才会设置字段的值
    QPushButton* pb=new QPushButton("set field");
    pw.setButton(QWizard::CustomButton1,pb);    pw.setOptions(QWizard::HaveCustomButton1);
//添加一个侧边部件，使用该侧边部件也可设置字段的值。
    QWidget *w=new QWidget;
    QLabel *pa=new QLabel("set field");    QLineEdit *pel=new QLineEdit;
    QHBoxLayout *pv=new QHBoxLayout;
    pv->addWidget(pa);    pv->addWidget(pel);    w->setLayout(pv);
    pw.setSideWidget(w);
//为每一页添加一个标签，以指明当前是哪一页
    QLabel *pa1=new QLabel("page 1",pwg1);    QLabel *pa2=new QLabel("page 2",pwg2);
    QLabel *pa3=new QLabel("page 3",pwg3);    QLabel *pa4=new QLabel("page 4",pwp);

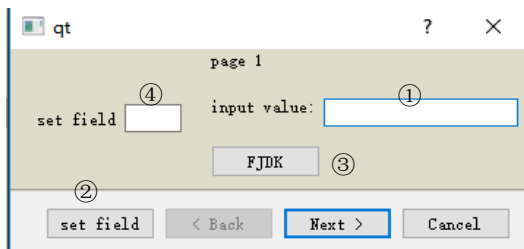
    //pw.setOptions(QWizard::IndependentPages); //设置该属性可禁用字段的传递。
    //pw.setField("AAA","ZZZZ"); //字段 AAA 还未注册成功，此处的设置不起作用。
    pw.addPage(pwg1);    /*字段 AAA 创建于页面 pwg1 之中，因此至少要在 pwg1 添加到向导之后，字段才会注册成功。*/
    pw.setField("AAA","FJDK");    /*设置字段 AAA 的值。此处调用的是 QWizard 类中的 setField() 函数，
        设置的字段的值会作用于所有页面。*/
    pe->setText(pw.field("AAA").toString()); //把 pe 的值设置为字段 AAA 所附带的值，此时为 FJDK
    pw.addPage(pwg2);    pw.addPage(pwg3);    pw.addPage(pwp);

//把部件 pb 的 clicked 信号关联到槽 D::f，该槽会设置字段的值。
    QObject::connect(pb,&QPushButton::clicked,pwg1,&D::f);
//把侧边部件 pel 的 textChanged 信号关联到槽 D::f1，该槽会设置字段的值。
    QObject::connect(pel,&QLineEdit::textChanged,pwg1,&D::f1);
    pw.show();    return aa.exec(); }

```

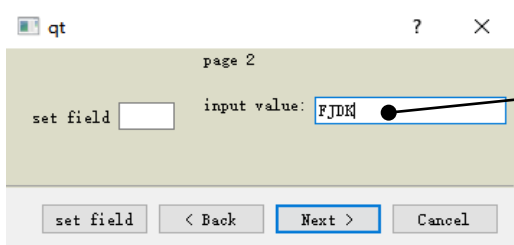
运行结果及说明

1、初次运行时的界面



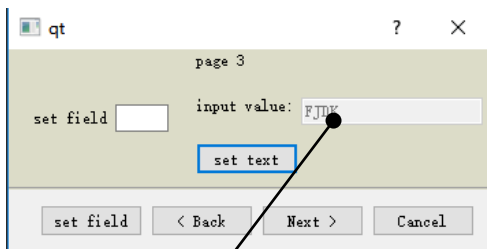
- 设置的字段是③处按钮 FJDK 的文本
- 在①处输入文本，然后点击②处的按钮，会把字段的值设置为该文本，此时③处的按钮的文本会改变为①处输入的文本。
- 在④处也可设置字段的值。在 4 处输入的值会立即显示到 3 处的按钮上。

2、点击 Next 按钮进入第 2 页

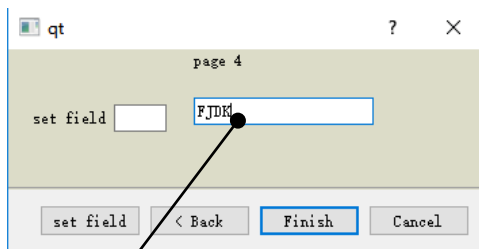


此处的文本是使用的字段的值。

3、进入第 3 页和第 4 页

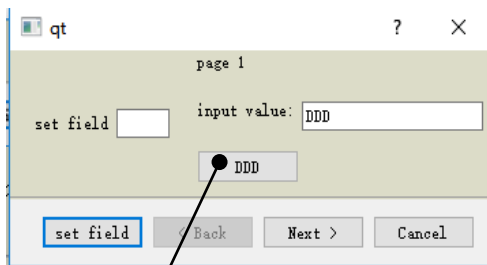


点击 set text 按钮后，会把此处的文本设置为字段的值。

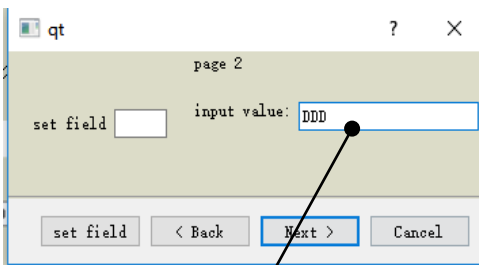


此处的文本此时是使用的字段的值。

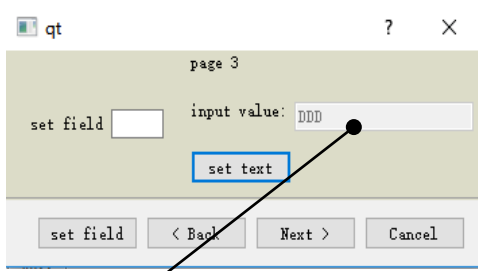
4、在第一页行编辑器中输入字符 DDD，并点击 set field 按钮



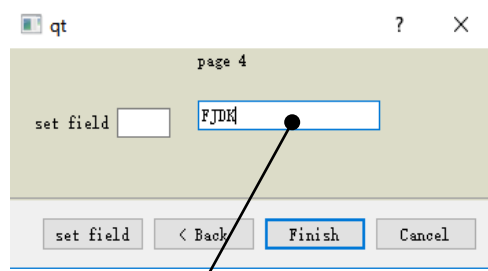
在第 1 页的此按钮的文本被设置为了 DDD



点击 Next 按钮进行第 2 页后，此处的文本被自动设置为了 DDD，因为 D1 类重新实现了 initializePage() 函数



在第 3 页需要点击 set text 按钮，才会把此处的文本被设置为 DDD，因为 D2 类未重新实现 initializePage()函数



第 4 页此处的文本未被改变，因为对此处文本的设置是在 main()函数中进行的。

7、必填 字段与信号：

- ①、必填字段是以星号(*)结尾的字段，只有在所有必填字段都被填写时，才会启用 Next 或 Finish 按钮。
- ②、若字段是必填字段，则此时需要为字段指定信号，以通知 QWizard 重新检查必填字段的值，若不为必填字段指定信号，则即使必填字段已拥有值，Next 或 Finish 按钮也不会被启用。

示例：必填字段与信号

//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>
class D:public QWizardPage{    Q_OBJECT
public:    QLabel *p,*p1,*p2,*p3;    QLineEdit *pe,*pe1,*pe2;    QPushButton *pb;
    D(QWidget* pw=0):QWizardPage(pw) {
        p=new QLabel("page 1");                p1=new QLabel("input value:");
        p2=new QLabel("input value:");          p3=new QLabel("input value:");
        pe=new QLineEdit;                        pe1=new QLineEdit;                pe2=new QLineEdit;
//布局页面的部件
        QFormLayout *pf=new QFormLayout;
        pf->addRow(p);    pf->addRow(p1,pe);    pf->addRow(p2,pe1);    pf->addRow(p3,pe2);
        setLayout(pf);
        registerField("AAA*",pe,"text",SIGNAL(textChanged(const QString)));//注册必填字段。
        registerField("BBB*",pe1,"text",SIGNAL(editingFinished()));//注册必填字段。
        registerField("CCC",pe2,"text");    }    };    //类 D 结束

class D1:public QWizardPage{    Q_OBJECT
public:    QLabel *p,*p1;    QLineEdit *pe;
    D1(QWidget* pw=0):QWizardPage(pw) {
        p=new QLabel("page 2");
        p1=new QLabel("input value:",this);    p1->move(0,33);
        pe=new QLineEdit(this);                pe->move(77,33);
        registerField("DDD*",pe,"text");    //注册必填字段。
    }    };    //类 D1 结束
```

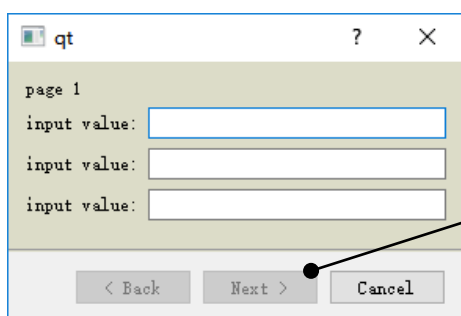
```
#endif // M_H
```

//m.cpp 文件的内容

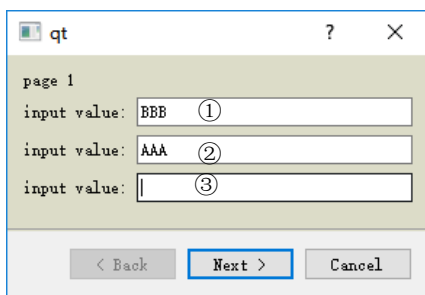
```
#include "m.h"

int main(int argc, char *argv[]) {    QApplication aa(argc, argv);
    QWizard pw;          D *pwg1=new D;    D1 *pwg2=new D1;
    QWizardPage *pwp=new QWizardPage;
    QLabel *pa4=new QLabel("page 3", pwp);    QLineEdit *pe=new QLineEdit(pwp); pe->move(0, 33);
    pw.setWizardStyle(QWizard::ModernStyle);
    pw.addPage(pwg1);        pw.addPage(pwg2);    pw.addPage(pwp);
    pw.show();    return aa.exec(); }
```

运行结果及说明

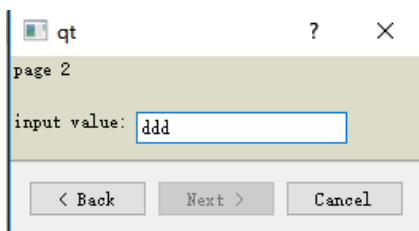


初次运行时，因为存在必填字段，所以 Next 按钮是被禁用的。



若首先在 2 处输入值，然后在 1 处只要输入值按钮 Next 会立即启用。

若首先在 1 处输入值，然后在 2 处输入值，此时需离开 2 处的行编辑器(比如点击 3 处的行编辑器)，或按下回车才会启用 Next 按钮(因为 2 处的行编辑器使用的是 editingFinished 信号)



进入第 2 页，其中行编辑器是必填字段，但程序未对该字段指定信号，因此，无论怎样在行编辑器中输入值，都不会使 Next 按钮被启用，从而无法进入第 3 页。

七、实现非线性向导(所用到的函数及属性原型请参阅后文)

- 1、非线性向导是指点击 Next 按钮时，向导会进入到指定的页面(不一定是下一页)。
- 2、要实现非线性向导，需要重新实现 nextId()虚函数，在 QWizard 和 QWizardPage 类中都有

nextId()函数，默认实现是 QWizard::nextId()调用 QWizardPage::nextId();

3、注意：QWizard 和 QWizardPage 中的 nextId()虚函数会被 Qt 调用多次，其规则如下

- 初次运行时会调用 2 次。
- 点击 Next 按钮时会调用 3 次。
- 点击 Back 按钮是会调用 2 次。

4、Qt 会根据 nextId()函数返回的值，对向导产生不同的设置，下面是 QWizard::nextId()的规则
点击 Next 按钮时，

- 第 1 次调用 nextId(), 根据其返回值设置需要显示的页面。
- 第 2 次调用 nextId(), 根据其返回值，确定是否设置 Next 按钮，若此时 nextId()返回 -1，则不设置 Next 按钮，否则设置 Next 按钮。
- 第 3 次调用 nextId(), 根据其返回值，确定是否设置 Finish 按钮，若此时 nextId()返回 -1，则设置 Finish 按钮，否则不设置 Finish 按钮。

点击 Back 按钮时，

- 第 1 次调用 nextId(), 根据其返回值，确定是否设置 Next 按钮，若此时 nextId()返回 -1，则不设置 Next 按钮，否则设置 Next 按钮。
- 第 2 次调用 nextId(), 根据其返回值，确定是否设置 Finish 按钮，若此时 nextId()返回 -1，则设置 Finish 按钮，否则不设置 Finish 按钮。

由以上规则可见，重新实现 nextId()时，需要根据情形(比如最后一页)，而返回不同的值。

示例：重新实现 nextId() 以实现非线性向导

//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>
#include <iostream>
using namespace std;

//使用枚举表示页面的 ID
class A{public:enum {one=1, two, three,four};};

//子类化 QWizard，因为该类可以控制所有页面。
class D:public QWizard{    Q_OBJECT
public:    QLabel *pa;    QLineEdit *pe;    QWizardPage *pwg1,*pwg2,*pwg3,*pwg4;
    QLabel *pa1,*pa2,*pa3,*pa4,*pa5;
    int i;        //用于保存 pe 中输入的数值
    D(QWidget* pw=0):QWizard(pw){
        i=0;
        pwg1=new QWizardPage;    pwg2=new QWizardPage;    pwg3=new QWizardPage;
        pwg4=new QWizardPage;
        //设置页面的内容
        pa1=new QLabel("page 1",pwg1);    pa2=new QLabel("page 2",pwg2);
        pa3=new QLabel("page 3",pwg3);    pa4=new QLabel("page 4",pwg4);
        pa5=new QLabel("input value",pwg1);    pa5->move(0,33);
        pe=new QLineEdit(pwg1);    pe->move(77,33);
        setWizardStyle(QWizard::ModernStyle);    //设置向导样式
        //向向导中添加页面
```

```

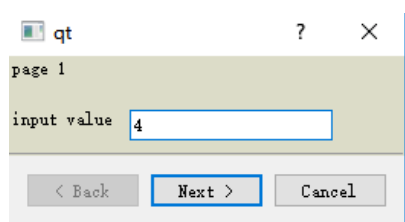
        setPage(A::one, pwg1);        setPage(A::two, pwg2);
        setPage(A::three, pwg3);       setPage(A::four, pwg4);
//使用槽函数 f1 获取 pe 中输入的文本内容。这样可避免在 nextId() 中重复获取。
QObject::connect(pe, &QLineEdit::textChanged, this, &D::f1);    } //构造函数结束

int nextId() const {
    static int j=0;    cout<<"A="<<j++<<endl;    //用于测试 nextId() 的调用次数。
    if(i==2) return A::two;    //如果输入 2, 则显示第 2 页
    if(i==3) return A::three;    //如果输入 3, 则显示第 3 页
    if(i>=4) {    //如果输入 4, 因为是最后一页, 所以作以下处理。
        static int k=0;
        if(k==0) {k=1;return 4;}    //第 1 次的返回值决定显示哪一页。
        //第 2 次的返回值决定是否设置 Next 按钮, 返回-1 表示不设置。
        else if(k==1) {k++;return -1;}
        //第 3 次的返回值决定是否设置 Finish 按钮, 返回-1 表示设置。
        else if(k==2) {k++;return -1;}
        //第 4 次的返回值由点击 Back 按钮时使用, 用于决定是否设置 Next 按钮, 返回非-1 表示设置。
        else if(k==3) {k++;return 1;}
        /*第 5 次的返回值也由点击 Back 按钮时使用, 用于决定是否设置 Finish 按钮, 返回非-1 表示不
        设置。*/
        else if(k==4) {k=0;return 1;}
        else return 4;    //其他情况返回 4。
    }
    return QWizard::nextId();    //输入的其他值, 使用父类的 nextId() 处理
    }    //nextId() 结束
    public slots:    void f1() {    i=pe->text().toInt();    };    //类 D 结束
#endif // M_H

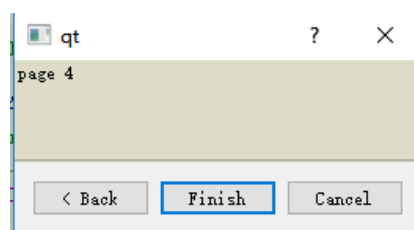
//m.cpp 文件的内容
#include "m.h"
int main(int argc, char *argv[]) {    QApplication aa(argc, argv);
    D pw;    pw.show();    return aa.exec();    }

```

运行结果及说明



初次运行的界面, 输入 2、3、4 程序会跳到指定的页面, 其他值会进入第 2 页, 只要页面进行了非线性的跳转, 点击 Back 按钮都会直接返回该页面



进入 4 页后的界面, 注意下面按钮的不同。点击 Back 按钮会返回到第 1 页

八、QWizard 类中的属性

①、**currentId**: const int 访问函数: int currentId() const; 信号: void currentIdChanged(int);

获取当前显示向导页的 ID, 该属性不能被直接设置, 要更改当前页需使用 `next()`、`back()`、`restart()` 函数。默认为 -1 (表示没有页面显示)。

- ②、**startId**: `int` **访问函数**: `int startId() const; void setStartId(int);`

获取和设置向导的起始页, 若未插入页面则为 -1, 默认为向导中的最低页 ID。

- ③、**subTitleFormat**: `Qt::TextFormat`

访问函数: `Qt::TextFormat subTitleFormat() const; void setSubTitleFormat(Qt::TextFormat)`

titleFormat: `Qt::TextFormat`

访问函数: `Qt::TextFormat titleFormat() const; void setTitleFormat(Qt::TextFormat)`

以上属性用于设置标题和子标题的文本格式 (即是使用纯文本还是富文本), `Qt::TextFormat` 枚举取值为 `Qt::PlainText` (纯文本)、`Qt::RichText` (富文本)、`Qt::AutoText` (自动检测)

- ④、**wizardStyle**: `WizardStyle`

访问函数: `WizardStyle wizardStyle(); void setWizardStyle (WizardStyle);`

设置向导的样式, `WizardStyle` 枚举的取值如下, 具体使用哪种样式依部件当前的样式而定。

QWizard::WizardStyle 枚举(无标志)

作用: 描述向导的样式

成员	值	说明
<code>QWizard::ClassicStyle</code>	0	经典 Windows 外观
<code>QWizard::ModernStyle</code>	1	现代 Windows 外观
<code>QWizard::MacStyle</code>	2	macOS 外观
<code>QWizard::AeroStyle</code>	3	Aero 外观 (Window Vista); 默认情况下, 向导仅在启用了 Alpha 组合的 Windows Vista 系统上使用此样式。

- ⑤、**options**: `WizardOptions`

访问函数: `WizardOptions options() const; void setOptions(WizardOptions);`

有关向导外观的各种选项 (主要是对按钮的设置), 在 windows 下, 默认取值为 `HelpButtonOnRight`; 在 macOS 上, 默认取值为 `NoDefaultButton` 和 `NoCancelButton`。其中 `WizardOption` 枚举见下表

QWizard::WizardOption 枚举

标志 `QWizard::WizardOptions`

作用: 描述向导的外观

成员	值	说明
<code>QWizard::IndependentPages</code>	0x0000 0001	页面彼此独立 (即各页面间不会相互导出值)
<code>QWizard::IgnoreSubTitles</code>	0x0000 0002	即使设置了子标题, 也不显示子标题
<code>QWizard::ExtendedWatermarkPixmap</code>	0x0000 0004	把水印下降到窗口边缘
<code>QWizard::NoDefaultButton</code>	0x0000 0008	使 Next 或 Finish 按钮不是默认按钮
<code>QWizard::NoBackButtonOnStartPage</code>	0x0000 0010	在起始页面不显示 Back 按钮
<code>QWizard::NoBackButtonOnLastPage</code>	0x0000 0020	在最后一页不显示 Back 按钮。

QWizard::DisabledBackButtonOnLastPage	0x0000 0040	禁用最后一页上的 Back 按钮。
QWizard::HaveNextButtonOnLastPage	0x0000 0080	在最后一页显示 Next 按钮(状态为禁用)
QWizard::HaveFinishButtonOnEarlyPages	0x0000 0100	在非最终页面上显示 Finish 按钮(状态为禁用)。
QWizard::NoCancelButton	0x0000 0200	不显示 Cancel 按钮
QWizard::CancelButtonOnLeft	0x0000 0400	把 Cancel 放在 Back 的左侧(默认为 Finish 或 Next 的右侧)
QWizard::HaveHelpButton	0x0000 0800	显示 Help 按钮
QWizard::HelpButtonOnRight	0x0000 1000	把 Help 按钮放在最右侧(默认为最左侧)
QWizard::HaveCustomButton1	0x0000 2000	显示用户自定义按钮(CustomButton1)
QWizard::HaveCustomButton2	0x0000 4000	显示用户自定义按钮(CustomButton2)
QWizard::HaveCustomButton3	0x0000 8000	显示用户自定义按钮(CustomButton3)
QWizard::NoCancelButtonOnLastPage	0x0001 0000	在最后一页不显示 Cancel 按钮

九、QWizard 类中的函数

1、与向导外观有关的函数

- ①、QWizard(QWidget* parent = Q_NULLPTR, Qt::WindowFlags flags = Qt::WindowFlags()); //构造函数
- ②、void setOption(WizardOption option, bool on = true);
bool testOption(WizardOption option) const;
以上函数是对 options 属性的设置。
- ③、void back(); //槽，上一页，相当于按下 Back 按钮。
void next(); //槽，下一页，相当于按下 Next 或 Commit 按钮。
void restart(); //槽，回到起始页。
- ④、QPixmap pixmap(WizardPixmap which) const;
void setPixmap(WizardPixmap which, const QPixmap &pixmap);
获取和设置位置 which 处的图片，该函数设置的是整个向导的图片，也就是说在显示其他页时其图片不会改变，QWizardPage::setPixmap()函数设置的只是当前页的图片，当显示另一页时，将不再是该图片。WizardPixmap 枚举的取值如下表：

QWizard::WizardPixmap 枚举(无标志)		
作用：描述向导的图片位置		
成员	值	说明
QWizard::WatermarkPixmap	0	ClassicStyle 和 ModernStyle 样式水印区域
QWizard::LogoPixmap	1	ClassicStyle 和 ModernStyle 样式 logo 区域
QWizard::BannerPixmap	2	banner 区域 (仅适用于 ModernStyle 样式)
QWizard::BackgroundPixmap	3	背景图片(仅适用于 MacStyle 样式)

- ⑤、QWidget* sideWidget() const; //返回向导左侧的部件(默认情况下，左侧不存在部件)。
void setSideWidget(QWidget* widget);
把部件 widget 设置在向导的左侧，对于有水印的样式(ClassicStyle 和 ModernStyle)，部件显示在水印上面，对于其他样式，部件显示在向导的左侧。设置为 0 表示没有侧面部件，新设置的部件会隐藏之前的任何部件。当向导被销毁时，会销毁侧面部件。

2、与向导中的按钮有关的函数。

QWizard::WizardButton 枚举(无标志)		
作用：描述向导的按钮		
成员	值	说明
QWizard::BackButton	0	Back 按钮(macOS 上为 Go Back)
QWizard::NextButton	1	Next 按钮(macOS 上为 Continue)
QWizard::CommitButton	2	Commit(提交)按钮
QWizard::FinishButton	3	Finish 按钮(macOS 上为 Done)
QWizard::CancelButton	4	Cancel 按钮
QWizard::HelpButton	5	Help 按钮
QWizard::CustomButton1	6	用户自定义按钮 1
QWizard::CustomButton2	7	用户自定义按钮 2
QWizard::CustomButton3	8	用户自定义按钮 3
QWizard::Stretch	9	这是一个水平拉伸空白间隙。

- ⑥、QAbstractButton* **button**(WizardButton which) const; //返回角色为 which 的按钮
void **setButton**(WizardButton which, QAbstractButton *button);把角色为 which 的按钮设置为 button。
- ⑦、QString **buttonText**(WizardButton which) const; //返回按钮上的文本。
void **setButtonText**(WizardButton which, const QString &text);
把角色为 which 的按钮文本设置为 text, 按钮文本还可使用 QWizardPage::setButtonText() 函数按页指定, 这样, 同一个按钮在不同页面会显示为不同的文本。
- ⑧、void **setButtonLayout**(const QList<WizardButton>& layout);
设置按钮的布局, 使用该函数可以设置按钮的排列顺序。

3、与页面有关的函数

- ⑨、int **addPage**(QWizardPage* page);
把向导页 page 添加到向导中, 并返回该页的 ID, 该 ID 一定大于向导中的任何其他 ID。
- ⑩、void **setPage**(int id, QWizardPage* page); 把向导页 page 添加到向导, 并设置其 ID 为 id。
QWizardPage* **page**(int id) const; //返回 ID 为 id 的向导页, 若没有, 则返回 0。
QWizardPage* **currentPage**() const; //返回指向当前页的指针, 若没有, 则返回 0。
void **removePage**(int id); //移除 ID 为 id 的页面, 若有需要, 将调用 cleanupPage() 函数。
QList<int> **pageIds**() const; //返回页面的 ID 列表。
- ⑪、QList<int> **visitedPages**() const;
按照向导页被访问的顺序返回其 ID 列表, 按下 Back 键, 将使当前页再次标记为“未访问”。
- ⑫、bool **hasVisitedPage**(int id) const;
若被访问向导面的历史记录包含 id, 则返回 true, 否则, 返回 false。按下 Back 键, 将使当前页再次标记为“未访问”。

4、与字段有关的函数

- ⑬、void **setField**(const QString &name, const QVariant& value);

QVariant **field**(const QString &name) const;

以上函数表示获取和设置名为 **name** 字段的值，以上函数可用于设置向导上任何页面的字段。与其相关的函数有 `QWizardPage::registerField()`;

- ⑭、void **setDefaultProperty**(const char* className, const char* property, const char* changedSignal);

把名为 **className** 字段的默认属性设置为 **property**，并把相关的更改信号设置为 **changedSignal**。当 **className**(或其子类之一)的实例传递给 `QWizardPage::registerField()` 且未指定属性时，默认属性会被使用。

对于常见的 Qt 部件(或其子类)，不需要指定 **property** 和 **changedSignal**，`QWizard` 知道这些部件的属性和更改信号，下表为不需指定属性和更改信号的部件。

部件	属性	更改信号
QAbstractButton	bool checked	toggled()
QAbstractSlider	int value	valueChanged()
QComboBox	int currentIndex	currentIndexChanged()
QDateTimeEdit	QDateTime dateTime	dateTimeChanged()
QLineEdit	QString text	textChanged()
QListWidget	int currentRow	currentRowChanged()
QSpinBox	int value	valueChanged()

5、QWizard 类中的虚函数

- ⑮、virtual bool **validateCurrentPage**(); //虚拟的

- 当用户单击 **Next** 或 **Finish** 执行最后的验证时，此虚函数由 `QWizard` 调用。
- 若返回 **true**，则显示下一页(或向导结束)，否则保持当前页不动。
- 默认实现是调用 `QWizardPage::validatePage()` 虚函数。
- 若可能，则通过指定必填字段或重新实现 `isComplete()` 来禁用 **Next** 或 **Finish** 按钮，比重新实现该函数更好。
- 当用户点击 **Next** 或 **Finish** 时，验证页面输入的内容是否合法，比如若用户输入了不完整的信息，则显示一个错误消息等。

- ⑯、virtual int **nextId**() const; //虚拟的

该虚函数由 `QWizard` 调用，表示当用户按下 **Next** 按钮时显示的下一个页面，返回值是下一个页面的 ID，若无后续页面，则返回 -1，默认实现是调用 `QWizardPage::nextId()` 虚函数。重新实现该函数可控制页面的显示顺序。

- ⑰、virtual void **initializePage**(int id); //虚拟的，受保护的

- 该虚函数由 `QWizard` 调用，该函数用于在显示页面之前准备页面(就是初始化页面)。
- 但是，若设置了 `QWizard::IndependentPages` 选项，则只在第一次显示页面时调用此函数，
- 默认实现是调用 `QWizardPage::initializePage()` 虚函数。
- 重新实现该函数可以确保页面根据前一页的字段进行正确初始化。

- ⑱、virtual **cleanupPage**(int id); //虚拟的，受保护的

- 该虚函数由 QWizard 调用，表示在用户离开它之前通过点击 Back 按钮来清除 ID 为 id 的页面(除非设置了 QWizard::IndependentPages 选项)。
- 默认实现为调用页面 ID 为 id 的 QWizardPage::cleanUpPage()虚函数。

6、QWizard 类中的信号

void **currentIdChanged**(int id); //信号，当当前页发生更改时，发送此信号。id 为新的当前页的 ID。
 void **customButtonClicked**(int which); //信号，当用户点击自定义按钮时，发送此信号。
 void **helpRequested**(); //信号，当用户点击 Help 按钮时，发送此信号。
 void **pageAdded**(int id); //信号，当向导添加向导页时，发送此信号，id 为该页面的 ID。
 void **pageRemoved**(int id); //信号，当从向导中移除页面时，发送此信号 id 为移除页面的 ID。

十、QWizardPage 类中的属性和函数

1、属性

subTitle: QString **访问函数**: QString subTitle() const; void setSubTitle(const QString&);
title: QString **访问函数**: QString title() const; void setTitle(const QString&);

以上属性分别描述向导的子标题和标题，这些属性虽然由 QWizardPage 设置，但不会显示在 QWizardPage 部件上，而是显示在 QWizard 部件上面。标题和子标题可以是富文本和纯文本的，具体取决于 QWizard::titleFormat 和 QWizard::subTitleFormat 属性。

2、函数

①、QWizardPage(QWidget *parent = Q_NULLPTR); //构造函数

②、QString **buttonText**(QWizard::WizardButton which) const;

void **setButtonText**(QWizard::WizardButton which, const QString& text);

以上函数用于获取和设置向导上按钮的文本，详见 QWizard::setButtonText()函数。

③、QPixmap **pixmap**(QWizard::WizardPixmap which) const;

void **setPixmap**(QWizard::WizardPixmap which, const QPixmap &pixmap);

以上函数用于获取和设置向导上图片，详见 QWizard::setPixmap()函数。

④、bool **isCommitPage**() const; //若该页是提交页面，则返回 true。

void **setCommitPage**(bool commitPage);

若 commitPage 为 true，则将该页面设置为提交页，否则，设置为普通页。提交页是指不能通过单点 Back 或 Cancel 来撤消的页面，在提交页面上使用 Commit 按钮替换 Next 按钮，单击 Commit 按钮就像单击 Next 一样。直接点击提交页上 Commit 按钮进入下一页，则下一页的 Back 按钮是被禁用的。

⑤、void **setFinalPage**(bool finalPage);

若 finalPage 为 true，则在该页面上显示 Finish 按钮，这样用户可提前完成向导。

bool **isFinalPage**() const;

若返回 true，则在该页面上显示 Finish 按钮，该函数由 QWizard 调用，用于确定该页面是否应显示 Finish 按钮。默认情况下，若没有下一页(即 nextId()返回-1，也就是最后一页)，则返回 true，否则返回 false。

3、虚函数、受保护的函数、信号

⑥、virtual bool **isComplete**() const; //虚拟的

- 若返回 `true`，则启用 `Next` 或 `Finish` 按钮。
- `QWizard` 调用该函数，用于确定是否启用 `Next` 或 `Finish` 按钮。若填写了所有必填字段，则默认返回 `true`，否则返回 `false`。
- 重新实现该函数可强制用户执行某一操作，才能进行下一步。
- 若重新实现该函数，当页面变得完整或不完整时应在合适的地方发送 `QWizardPage::completeChanged()` 信号(可把另一个信号连接到该信号来发送该信号)，以更新按钮的启用/禁用状态。

⑦、`virtual void cleanupPage();` //虚拟的

- 当用户点击 `Back` 按钮离开页面时，由 `QWizard::cleanupPage()` 调用该函数(除非设置了 `QWizard::IndependentPages` 选项)。
- 默认实现为把页面的字段重置为原始值(在调用 `initializePage()` 之前的值)。
- 当用户点击 `Back` 按钮时，会调用此函数以重置页面的内容。

⑧、`virtual bool validatePage();` //虚拟的

- 当用户单击 `Next` 或 `Finish` 执行最后的验证时，此虚函数由 `QWizard::validateCurrentPage()` 调用。
- 若返回 `true`，则显示下一页(或向导结束)，否则保持当前页不动。
- 默认实现返回 `true`。
- 若可能，则通过指定必填字段或重新实现 `isComplete()` 来禁用 `Next` 或 `Finish` 按钮，比重新实现该函数更好。

⑨、`virtual void initializePage();` //虚拟的

- 该虚函数由 `QWizard::initializePage()` 调用，当用户单击 `Next` 按钮时，会调用该函数以初始化页面的内容，即该函数在显示下一页之前准备页面。
- 但是，若设置了 `QWizard::IndependentPages` 选项，则只在第一次显示页面时调用此函数，
- 默认实现该函数什么也不做。
- 重新实现该函数可以确保页面根据前一页的字段进行正确初始化。
- 若想要使用用户在前一页输入的内容来设置当前页的默认值，则需要重新实现该函数。

⑩、`virtual int nextId() const;` //虚拟的

该虚函数由 `QWizard::nextId()` 调用，以找出当用户按下 `Next` 按钮时显示的下一个页面，返回值是下一个页面的 ID，若无后续页面，则返回 -1，默认情况下，该函数返回大于当前页 ID 的最低 ID，若没有此 ID，则返回 -1。重新实现该函数可控制页面的显示顺序。

⑪、`QWizard* wizard() const;` //受保护的

返回与此页相关联的向导，若该页还未添加到 `QWizard` 中，则返回 0。

⑫、`QVariant field(const QString &name) const;` //受保护的

`void setField(const QString &name, const QVariant &value);` //受保护的

以上函数表示获取和设置名为 `name` 字段的值，以上函数也可用于向导上任何页面的字段。相当于 `wizard()->setField(name, value);` 和 `wizard()->field(name);`;

⑬、`void registerField(const QString &name, QWidget* widget, const char *property = Q_NULLPTR,`

`const char* changedSignal = Q_NULLPTR;` //受保护的

- 创建一个名称为 `name` 的字段，该字段与部件 `widget` 的属性 `property` 相关联，在这之后，便可使用 `field()`和 `setField()`函数访问该属性。
- 字段对于整个向导而言是全局的。
- 若 `name` 是以星号(*)结尾的，则该字段是必填字段，只有在所有必填字段都被填写时，才能启用 `Next` 或 `Finish` 按钮，此时需要为字段指定信号，以通知 `QWizard` 重新检查必填字段的值；若不为必填字段指定信号，则即使必填字段已拥有值，`Next` 或 `Finish` 按钮也不会被启用。信号由 `changedSignal` 指定(通常是更改信号)，注意：指定信号时需要使用 `SIGNAL` 宏。
- `QWizard` 支持的默认属性和更改信号见 `QWizard::setDefaultField()`函数。

示例：

```
QLineEdit *pe=new QLineEdit;  
registerField("AAA*", pe, "text", SIGNAL(editingFinished()));
```

以上代码表示：注册一个名为 `AAA` 的必填字段，该字段包含 `pe` 的 `text` 属性，当编辑完成时使用 `editingFinished()`信号通知 `QWizard` 检查必填字段的值。

- ⑭、void `completeChanged()`; //信号

当页面的 `complete` 状态(即 `isComplete()`返回的值)发生变化时，发送此信号，发送该信号会导致 `isComplete()`函数被调用。

作者：黄邦勇帅(原名：黄勇)

2018-4-30

本文作者：黄邦勇帅(原名：黄勇)

本文要求读者已经非常熟悉 C++ 语法。若读者不熟悉 C++ 语法，推荐参阅《C++ 语法详解》(作者：黄勇)一书，电子工业出版社出版。

本文主要讲解了 Qt 的主窗口，本文对 Qt 的主窗口作了比较全面详细的深入讲解，详细讲解了主窗口的每一个性质，并列举了详细的示例进行说明，同时本文也是非常方便、快捷的编写 Qt 程序的查阅资料，可方便的查阅到相关内容的原理，以及怎样使用该内容。本文内容由浅入深，易学易懂。

本文使用的是 windows 10 操作系统，Qt 版本为 Qt5.10.1，Qt Creator 的版本为 Qt Creator 4.5.1
本文内容完全属于个人见解与参考文献的作者无关，限于水平有限，其中难免有误解之处，望指出更正。

声明：禁止抄袭、复印、转载本文，本文作者拥有完全版权。

主要参考文献：

- 1、C++语法详解 黄勇 编著 电子工业出版社 2017 年 7 月
- 2、Qt Creator 快速入门(第 3 版) 霍亚飞 编著 北京航空航天大学出版社 2017 年 1 月
- 3、C++ GUI Qt4 编程(第 2 版) [加拿大] Jasmin Blanchette [英] Mark Summerfield 著 闫锋欣 曾泉人 张志强 译 电子工业出版社 2008 年 8 月
- 4、Qt 5 开发及实例 陆文周 主编 电子工业出版社 2014 年 1 月

第 7 章 Qt 主窗口目录

7.1 QMainWindow 类主窗口基础

7.2 QMenu 类、QMenuBar 类、QAction 类基础

7.2.1 基本概念

7.2.2 创建菜单的方法

7.2.3 部件的所有权

7.2.4 QAction 动作基础

7.3 QShortcut 类、快捷键

7.3.1 快捷键基础

7.3.1 QShortcut 类中的属性

7.3.2 QShortcut 类中的函数

7.4 QKeySequence 类、键序列

7.4.1 键序列基础

7.4.2 QKeySequence 类中的枚举

7.4.3 QKeySequence 类中的函数

7.5 QAction 类、QActionGroup 类

7.5.1 动作基本规则

7.5.2 QAction 类中的属性

7.5.3 QAction 类中的函数

7.5.4 QAction 类中的槽和信号

7.5.5 QWidget 类中与 QAction 有关的函数

7.5.6 QActionGroup 类动作组

7.6 QMenu 类、菜单

7.6.1 菜单基本规则

7.6.2 QMenu 类中的属性

7.6.3 QMenu 类中的函数

7.7 QMenuBar 类、菜单栏

7.7.1 菜单栏基本规则

7.7.2 QMenuBar 类中的属性

7.7.3 QMenuBar 类中的函数

7.8 QToolBar 类、工具栏

7.8.1 工具栏基本规则

7.8.2 QToolBar 类中的属性

7.8.3 QToolBar 类中的函数

7.8.4 QToolBar 类中的信号

7.9 QStatusBar 类、状态栏

7.9.1 状态栏基本规则

7.9.2 QStatusBar 类中的属性

7.9.3 QStatusBar 类中的函数

7.10 QDockWidget 类、可停靠窗口、悬浮窗口

7.10.1 可停靠窗口基本规则

7.10.2 QDockWidget 类中的属性

7.10.3 QDockWidget 类中的函数

7.10.4 QDockWidget 类中的信号

7.11 QMainWindow 类、主窗口

7.11.1 QMainWindow 类中的属性

7.11.2 QMainWindow 类中的函数

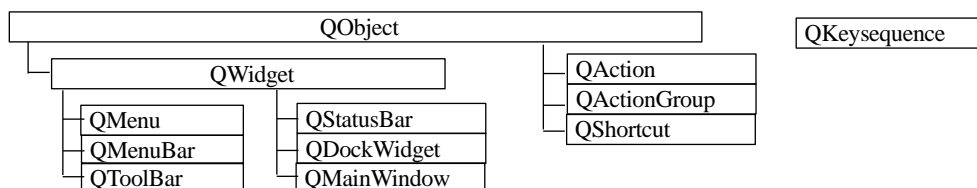
7.11.3 QMainWindow 类中的信号

第 7 部分 Qt 主窗口

注意：本程序都假设读者在 pro 文件中已添加了正确的 `QT+=widgets` 语句，文中不再重复累述添加此语句。

本文注重讲解原理，因此使用的是手写的 Qt 程序，对于使用 Qt 设计师快速设计 Qt 程序会在专门章节讲解。

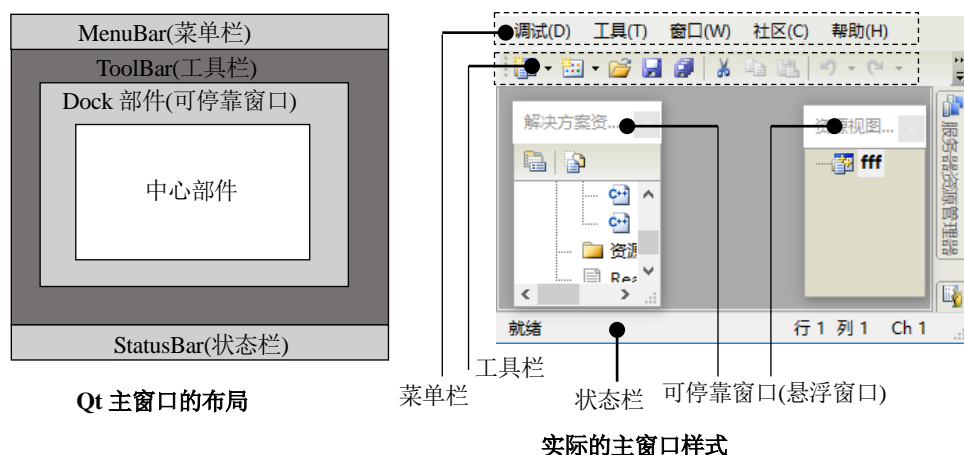
本章讲解的类及继承关系如下图所示



是否应在第 7.11.2 节增加 `QWidget::contextMenuPolicy` 属性，

7.1 QMainWindow 类(主窗口)基础

- 1、`QMainWindow` 类继承自 `QWidget` 类，因此主窗口就是一个普通的部件，只不过主窗口拥有自己的布局，在主窗口中特定的位置只能添加特定的子部件而已。因为主窗口有自己的布局，因此不能在主窗口上设置布局管理器。
- 2、下图为主窗口的布局图



- ①、中心部件：主窗口必须有且只能有一个中心部件。中心部件可以是任意类型的部件(比如 `QTextEdit`、`QLineEdit` 等)。中心部件使用 `setCentralWidget()` 函数设置
- ②、状态栏：状态栏主要用于向用户显示一些程序的状态信息，状态栏位于主窗口底部，一个主窗口只能有一个状态栏。使用 `setStatusBar()` 函数可设置状态栏。

- ③、工具栏：工具栏通常是由一系列的类似按钮的部件组成的。使用 `addToolBar()` 可以把工具栏添加到主窗口。工具栏是由 `QToolBar` 类实现的。一个主窗口可以有多个工具栏，工具栏可以位于主窗口的上、下、左、右四个方向上。
- ④、Dock 部件(悬浮窗口、可停靠窗口)：一个主窗口可以有多个可停靠窗口。使用 `addDockWidget()` 可以把可停靠窗口添加到主窗口
- ⑤、菜单与菜单栏：菜单位于菜单栏之中，一个主窗口只能有一个菜单栏，一个菜单栏中可以包含多个菜单。`QMainWindow` 有一个默认的菜单栏，该菜单栏使用 `menuBar()` 函数返回。

7.2 QMenu 类(菜单)、QMenuBar 类(菜单栏)、QAction 类(动作)基础

QMenu 和 QMenuBar 类都继承自 QWidget。

一、基本概念



- 1、菜单：为便于讲解本文会把单独的一个菜单项和含有其他菜单项的菜单统称为菜单，因此菜单这一概念比较易产生混淆。
- 2、弹出菜单、下拉菜单：展开后包含其他菜单或菜单项的菜单称为下拉菜单或弹出菜单。通常顶级菜单是弹出菜单。
- 3、主菜单：如上图所示，“文件”、“编辑”、“视图”等都是主菜单。
- 4、子菜单：位于弹出菜单中的弹出菜单称为子菜单，因此子菜单也是弹出菜单。通常子菜单在右侧有一个向右的箭头。比如上图中的“新建”、“打开”都是子菜单。
- 5、系统菜单：在标题栏最左边的小图标上左击或右击鼠标，会激活系统菜单。
- 6、快捷菜单(上下文菜单)：指的是点击鼠标右键弹出的菜单，这种菜单也是弹出菜单。
- 7、标记菜单：菜单项可以被选中，即在菜单文本左侧显示一个小的选中标记(一般为一个勾形符号)，这种类型的菜单被称为标记菜单。顶级菜单不能被选中。
- 8、助记符：是指使用“ALT+某个字符”形式的快捷键，通常这个字符会加上下划线或被小括号括起来，如上图中的“文件(F)”，其中的 F 就是助记符。
- 9、快捷键、加速键、助记符：把 Ctrl+C(复制)这种类型的快捷键称为加速键，其实助记符和加速键都是快捷键，为避免引起概念混淆，本文尽量注意区分。

二、创建菜单的方法

- 1、QMenu 类可用于创建菜单，该类可创建单独的一个菜单项，并可使用 QMenu::addMenu() 函数把一个菜单项添加到另一个菜单项中，从而创建一个子菜单。
- 2、QMenuBar 是菜单栏，使用 QMenu 创建的菜单通常会添加到菜单栏中。使用 QMenuBar::addMenu() 函数可把菜单添加到菜单栏之中。

- 3、一个菜单只应使用 addMenu()函数添加一次，后续的添加将是无效的。
- 4、QMenu 和 MenuBar 类都继承自 QWidget，可见，他们其实就是一个 QWidget 部件，只不过他们有些特殊的限制。因此 QMenuBar 可以以窗口的形式使用 show()被单独显示出来，但 QMenu 应使用 exec()来显示。
- 5、下面是 addMenu 函数的原型(QMenu::addMenu 和 QMenuBar::addMenu 功能是相同的)

①、QAction* addMenu(QMenu *menu);

添加菜单 menu 到菜单栏或菜单中，并返回与此菜单关联的 Action(动作)。此函数不会使菜单或菜单栏拥有 menu 的所有权(关于所有权，见后文)。

②、QMenu* addMenu(const QString &title);

QMenu* addMenu(const QIcon& icon, const QString &title);

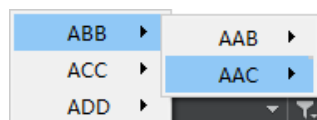
使用 title 和 icon 创建一个新菜单，并把其添加到菜单或菜单栏，并返回该新菜单，菜单栏或菜单拥有该新菜单的所有权。

示例：创建及显示 QMenu

```
#include<QtWidgets>
int main(int argc, char *argv[]) {    QApplication aa(argc, argv);
    QMenu pm1("AAA");                //创建菜单，仅需一个字符串即可。
    //向菜单 pm1 中添加菜单项
    QMenu *pm11=pm1.addMenu("ABB");   //pm1 拥有 addMenu() 函数返回的菜单的所有权。
    QMenu *pm12=pm1.addMenu("ACC");   QMenu *pm13=pm1.addMenu("ADD");
    //向菜单 pm11 中添加菜单项
    pm11->addMenu("AAB");              pm11->addMenu("AAC");
    pm1.exec();                        //使用 exec() 函数显示菜单 pm1
    return aa.exec(); }

```

运行结果及说明



该菜单会在屏幕左上角显示，注意：在显示菜单之前不要点击鼠标和使用键盘，以免使显示的菜单消息了。



关闭菜单后，程序不会结束，此时请使用此按钮结束程序

示例：把菜单添加到菜单栏中，并使菜单栏作为单独的窗口显示

```
#include<QtWidgets>
int main(int argc, char *argv[]) {    QApplication aa(argc, argv);
    QMenuBar pmb;                    //创建一个菜单栏
    //向菜单栏中添加顶级菜单
    QMenu *pm1=pmb.addMenu("AAA");    pmb.addMenu("BBB");        pmb.addMenu("CCC");
    //向顶级菜单 AAA 中添加菜单项
    QMenu *pm11=pm1->addMenu("ABB");  pm1->addMenu("ACC");        pm1->addMenu("ADD");
    //向菜单 ABB 中添加菜单项
    pm11->addMenu("AAB");              pm11->addMenu("AAC");
    /*还可向使用 QWidget 一样，向 QMenuBar 中添加按钮，同样也可向 QMenu 中添加按钮。但这种使用方法不推荐。*/
}

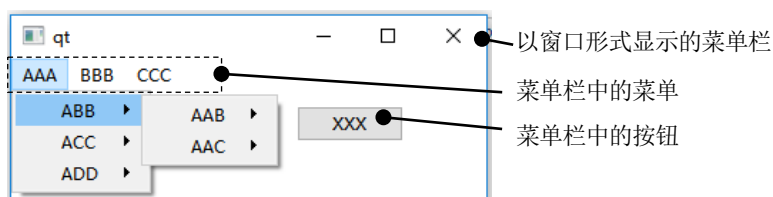
```

```

QPushButton *pb=new QPushButton("XXX",&pmb);    pb->move(200,33);
//设置菜单栏的大小,并以窗口的形式显示菜单栏
pmb.resize(333,222);    pmb.show();    return aa.exec(); }

```

运行结果及说明



示例：菜单栏与窗口

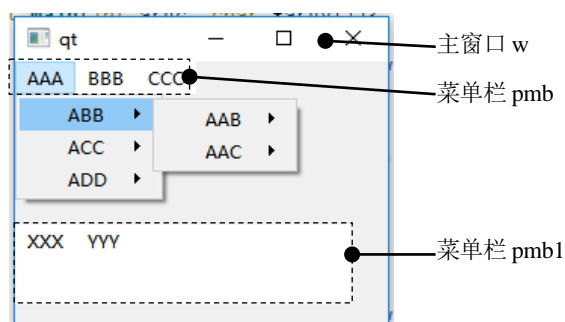
```

#include<QtWidgets>
int main(int argc, char *argv[]) {    QApplication aa(argc, argv);
    QWidget w;
    QMenuBar *pmb=new QMenuBar(&w);    //为菜单栏指定父窗口
    QMenu *pml=pmb->addMenu("AAA");    pmb->addMenu("BBB");    pmb->addMenu("CCC");
    QMenu *pml1=pml->addMenu("ABB");    pml->addMenu("ACC");    pml->addMenu("ADD");
    pml1->addMenu("AAB");    pml1->addMenu("AAC");

    QMenuBar *pmb1=new QMenuBar(&w);    pmb1->move(0,111);
    QMenu *pp=pmb1->addMenu("XXX");    pmb1->addMenu("YYY");
    pp->addMenu("X11");    pp->addMenu("X22");
    pmb1->resize(222,55);    //设置菜单栏的大小
    w.resize(333,222);    w.show();    return aa.exec(); }

```

运行结果及说明



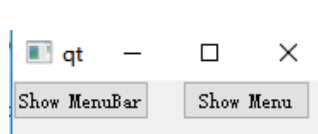
三、部件的所有权

- 1、所有权的意义：比如部件 1 若拥有部件 2 的所有权，则当部件 1 被删除(销毁)时，会删除部件 2。

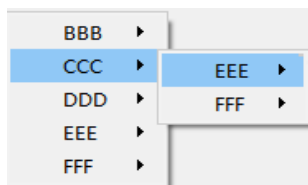
示例：部件所有权

```
#include<QtWidgets>
int main(int argc, char *argv[]) {    QApplication aa(argc, argv);
    QWidget w;    QMenuBar *pmb=new QMenuBar();
    /*以下 4 个使用 new 分配的对象，在本例没有显示的使用 delete 进行删除，也没有对象拥有他们的所有权，
    他们也没有父对象，因此在本例以下 4 个对象不会被 delete，也就是说会产生内存泄漏的问题，但由于程
    序较小，所以不会有什么影响。*/
    QMenu *pm1=new QMenu("AAA");          QMenu *pm2=new QMenu("BBB");
    QMenu *pm4=new QMenu("DDD");          QMenu *pm6=new QMenu("FFF");
    /*为菜单 pb3 指定父对象 pmb，此时 pmb 拥有对 pm3 的所有权，当 pmb 销毁时 pm3 也会被销毁，这也是为菜
    单指定父对象的主要作用。*/
    QMenu *pm3=new QMenu("CCC", pmb);
    //pm3 拥有菜单 EEE 的所有权，当 pm3 销毁时 EEE 也会被销毁。
    QMenu *pm5=pm3->addMenu("EEE");
    //以下对 addMenu() 函数的调用，都不会获得对实参对象的所有权。
    pm3->addMenu(pm6);    pm1->addMenu(pm2);    pm1->addMenu(pm3);    pmb->addMenu(pm4);
    pm1->addMenu(pm5);    pm1->addMenu(pm6);    pmb->addMenu(pm1);
    pmb->setAttribute(Qt::WA_DeleteOnClose, 1); //当 pmb 关闭时销毁 pmb，该属性的使用见第 6 章
    QPushButton *pb=new QPushButton("Show MenuBar", &w);
    QPushButton *pb1=new QPushButton("Show Menu", &w);    pb1->move(99, 0);
    pm1->move(444, 444);    //使菜单 pm1 在屏幕上(444, 444)处显示
    QObject::connect(pb, &QPushButton::clicked, pmb, &QMenuBar::show);
    QObject::connect(pb1, &QPushButton::clicked, pm1, &QMenu::show);
    w.resize(333, 222);    w.show();    return aa.exec(); }
```

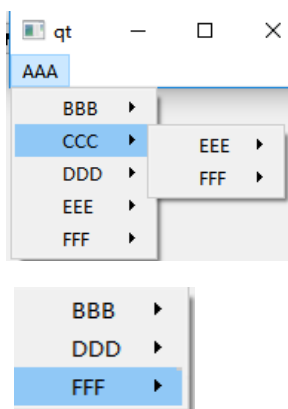
运行结果及说明



1、主窗口界面



2、首先点击 Show Menu 按钮，此时会在屏幕(444,444)位置处弹出如图所示菜单，此时的菜单是完整的菜单



3、

- 再点击 Show MenuBar 弹出菜单栏，此时菜单栏中的菜单也是完整的菜单，其中菜单栏拥有对菜单 CCC 的所有权，菜单 CCC 拥有菜单 EEE 的所有权。
- 点击右上角的 X 按钮关闭菜单栏(本例会销毁菜单栏)，此时 pmb 会销毁 CCC，而 CCC 又会销毁 EEE。
- 再次在主窗口点击 Show MenuBar 按钮，此时菜单栏不在显示(因为菜单栏被销毁了)。
- 再次点击主窗口的 Show Menu 按钮，显示如下图所有菜单，可以看到菜单 CCC 和 EEE 已被销毁，不再被显示出来，但 FFF 未被销毁，因此仍会被显示。

四、QAction(动作)基础(与 QAction 有关的函数原型及更详细的说明见 7.4 节)

- 1、QAction 用于使来自不同地方的命令都能够以相同的方式执行,比如 QAction 可以使菜单、工具栏和快捷键执行相同的操作。
- 2、QAction 就是一个动作,只不过这个动作会执行某一操作,而且 QAction 还可以包含有图标、文本、快捷方式、状态文字、What this 文字和工具提示。其中 QAction 的文本通常用作菜单的文本。
- 3、使用 QAction 需要完成以下关键步骤:
 - 把 QAction 对象使用 QWidget::addAction()函数添加到部件。必须把 QAction 添加到部件才能使用。
 - 把 QAction 的 triggered()信号连接到需要执行的槽函数。
 - 激活 QAction。
 - 若已把 QAction 添加到菜单中,则直接点击该菜单项即可;
 - 若 QAction 含有快捷键,则直接使用快捷键便可激活 QAction;
 - 若 QAction 是添加到某个 QWidget 部件的(比如 QPushButton),则可以把该部件的某个信号连接到 QAction::trigger()槽或 QAction::triggered()信号,来激活 QAction。
- 4、注意:若添加 QAction 的部件被禁用,则该动作将不会被激活。

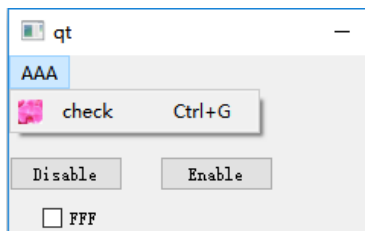
示例: QAction 的使用

```
#include<QtWidgets>
int main(int argc, char *argv[]) {    QApplication aa(argc, argv);
    QWidget w;
    //以下是创建 QAction 的方法
    QAction *pa=new QAction(QIcon("F:/li.png"), "check", &w); //含一图标、文本
    QAction *pal=new QAction("disable", &w); //仅有文本
    QAction *pa2=new QAction(&w); //既无图标也无文本
    QAction *pa3=new QAction("xxx", &w);
    //为 QAction 添加快捷键
    pa->setShortcut(QKeySequence("Ctrl+G"));    pal->setShortcut(QKeySequence("Ctrl+D"));
    pa2->setShortcut(QKeySequence("Ctrl+E"));    pa3->setShortcut(QKeySequence("Ctrl+H"));
    //创建菜单和菜单栏
    QMenuBar *pmb=new QMenuBar(&w);    QMenu *pml=pmb->addMenu("AAA");
    pml->addAction(pa); //动作 pa 的文本和图标会作为菜单项被添加到 pml 之中。
    //创建按钮部件
    QPushButton *pb=new QPushButton("Disable", &w);    pb->move(0, 66);
    QPushButton *pb1=new QPushButton("Enable", &w);    pb1->move(99, 66);
    QCheckBox *pc=new QCheckBox("FFF", &w);    pc->move(22, 99);
    //把 QAction 添加到按钮中
    pb->addAction(pal);    pb1->addAction(pa2);
    //把 QAction 的 triggered 信号连接到相应的槽函数
    //动作 pa 用于切换 pc 的选中状态
    QObject::connect(pa, &QAction::triggered, pc, &QCheckBox::toggle);
    //动作 pal 用于禁用 pc, 注意: triggered 信号会传递一个默认为 false 的 bool 参数。
    QObject::connect(pal, &QAction::triggered, pc, &QCheckBox::setEnabled);
    //动作 pa2 用于启用 pc
    QObject::connect(pa2, &QAction::triggered, pc, &QCheckBox::setDisabled);
    /*把 pb 的 clicked 信号关联到动作 pal, 这样点击 pb 将激活动作 pal。注意: clicked 信号也会传递一个默认为 false 的 bool 参数。*/
```



```
QObject::connect(pb, &QPushButton::clicked, pa1, &QAction::triggered);  
//使动作 pa3 切换 pc 的选中状态，但是该动作在本例不会有任何反应，因为未把该动作添加到任何部件。  
QObject::connect(pa3, &QAction::triggered, pc, &QCheckBox::toggle);  
w.resize(333, 222);    w.show();    return aa.exec(); }
```

运行结果及说明



测试方法如下：

- 1、点击菜单 check 或按下 Ctrl+G 可使复选框 FFF 在选中和未选中之间切换。
- 2、按下 Disable 或按下 Ctrl+D 可禁用复选框 FFF。
- 3、按下 Ctrl+E 可启用复选框 FFF，但按下 Enable 按钮不能启用复选框，因为该按钮未与动作 pa2 相关联。
- 4、按下 Ctrl+H 不会有任何反应，因为动作 pa3 未添加到任何部件。

7.3 QShortcut 类(快捷键)

一、快捷键基础

- 1、QShortcut 类对快捷键进行了描述，该类继承自 QObject。
- 2、助记符：是指使用“Alt+某个字符”形式的快捷键，通常这个字符会加上下划线或被小括号括起来。
- 3、快捷键、加速键、助记符：把 Ctrl+C(复制)这种类型的快捷键称为加速键，其实助记符和加速键都是快捷键。
- 4、快捷键的实现原理：首先使用指定的 QKeySequence(键序列)创建一个快捷键 QShortcut，然后把 QShortcut::activated()信号(激活快捷键时会发送此信号)关联到需要执行的槽函数即可。下面为其示例代码 QKeySequence 类见后文(此小节只需知道可以这样指定键序列即可)

方法 1:

```
QPushButton *pb=new QPushButton("AAA");
QShortcut *ps=new QShortcut(QKeySequence("Ctrl+D"),pb); //为部件 pb 创建快捷键 Ctrl+D。
//当按下 Ctrl+D 时，执行 pb 的 click 槽函数。
QObject::connect(ps,&QShortcut::activated,pb,&QPushButton::click);
```

方法 2:

```
QPushButton *pb=new QPushButton("AAA");
//为部件 pb 创建快捷键 Ctrl+D，当按下该快捷键时执行 pb 的槽函数 click。
QShortcut *ps1=new QShortcut(QKeySequence("Ctrl+D"),pb,SLOT(click()));
```

- 5、QShortcutEvent 事件：当用户按下一个键组合时会产生该事件。

二、QShortcut 类中的属性

- 1、**autoRepeat**: bool **访问函数**: bool autoRepeat() const; void setAutoRepeat(bool);
描述快捷键是否可自动重复，若为 true，则只要系统启用了键盘自动重复，则快捷键启用自动重复。默认为 true。自动重复详见第 2 章键盘事件小节
- 2、**context**: Qt::ShortcutContext
访问函数: Qt::ShortcutContext context() const; void setContext(Qt::ShortcutContext);
描述在什么情况下允许激活(或触发)快捷键，即快捷键的激活方式，默认为 Qt::WindowShortcut。其中 Qt::ShortcutContext 枚举见下表。

Qt::ShortcutContext 枚举(无标志)

描述在什么情况下激活快捷键。以下的父部件是指创建快捷键时指定的父部件，比如 QShortcutContext s(QKeySequence("Ctrl+D"), &w); 其父部件是指 w。

成员	值	说明
Qt::WidgetShortcut	0	当父部件拥有焦点时，快捷键被激活。
Qt::WidgetWithChildrenShortcut	3	当父部件或其子部件拥有焦点时，快捷键可被激活。


```

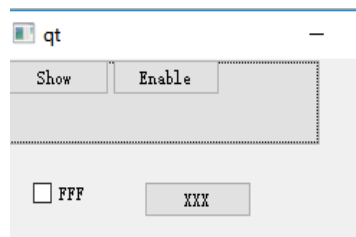
//创建 4 个快捷键并分别对应 4 种不同的
    QShortcut *ps=new QShortcut(QKeySequence("Ctrl+D"),pb1,SLOT(click()));
//以下 3 个快捷键都可禁用复选框 pc
    QShortcut *ps1=new QShortcut(QKeySequence("Ctrl+E"),pw);
    QShortcut *ps2=new QShortcut(QKeySequence("Ctrl+F"),pw);
    QShortcut *ps3=new QShortcut(QKeySequence("Ctrl+G"),pw);
//设置各快捷键的激活方式
    ps->setContext(Qt::ApplicationShortcut);    ps1->setContext(Qt::WindowShortcut);
    ps2->setContext(Qt::WidgetShortcut);    ps3->setContext(Qt::WidgetWithChildrenShortcut);

//激活快捷键 ps1、ps2、ps3 时调用 pb3 的 click 槽函数(即相当于点击 pb3)
    QObject::connect(ps1,&QShortcut::activated,pb3,&QPushButton::click);
    QObject::connect(ps2,&QShortcut::activated,pb3,&QPushButton::click);
    QObject::connect(ps3,&QShortcut::activated,pb3,&QPushButton::click);
//点击 pb3 禁用复选框 pc
    QObject::connect(pb3,&QPushButton::clicked,pc,&QCheckBox::setEnabled);
//点击 pb1 启用复选框 pc
    QObject::connect(pb1,&QPushButton::clicked,pc,&QCheckBox::setDisabled);
//点击 pb 显示对话框
    QObject::connect(pb,&QPushButton::clicked,pd,&QDialog::show);
    w.resize(333,222);    w.show();    return aa.exec(); }

```

运行结果及说明

- 1、Ctrl+E、Ctrl+F、Ctrl+G 都是禁用复选框 FFF 的快捷键
- 2、Ctrl+D 可重新启用复选框 FFF
- 3、把焦点移至 pw(即 show 和 Enable 的父部件)，此时按下 Ctrl+E、Ctrl+F、Ctrl+G 都可禁用复选框 FFF。
- 4、把焦点移至按钮 Show，此时按下 Ctrl+F 不能禁用 FFF，因为按钮 Show 不是快捷键 Ctrl+F 的父部件。
- 5、把焦点移至按钮 XXX，此时按下 Ctrl+F、Ctrl+G 不能禁用 FFF。因为 XXX 即不是 Ctrl+G 的父部件也不是他的子部件。Ctrl+F 的原因同第 4 点
- 6、点击按钮 Show 弹出对话框，使对话框成为活动的，此时按下 Ctrl+E、Ctrl+F、Ctrl+G 都不能禁用复选框 FFF。因为 Ctrl+E 的父部件 pw 不是活动顶层窗口 pd 的子部件。Ctrl+F 的原因同第 4 点，Ctrl+G 的原因同第 5 点
- 7、只要该程序的任一窗口是活动的，按下 Ctrl+D 都能启用 FFF



7.4 QKeySequence 类(键序列)

QKeySequence 是一个单独的父亲类，未继承其他类。

一、键序列基础

- 1、QKeySequence 类(键序列)：描述了组合成快捷键的键，比如 Ctrl+C、Ctrl+D 等。该类是一个顶级类，未继承自任何其他类。注意：QKeySequence 类只能创建一个按键的组合，但该组合具体能执行什么操作，还需要结合其他类使用，比如与 QShortcut 或 QAction 类结合使用。
- 2、可使用标准键、字符串和硬编码 3 种方式来创建键序列，示例如下：

```
QKeySequence(QKeySequence::Copy);    //使用标准键创建键序列
QKeySequence("Ctrl+C"); 或 QKeySequence("Ctrl+c"); //使用字符串创建键序列
QKeySequence(Qt::CTRL+Qt::Key_P);    //使用硬编码创建键序列
```

- 3、键盘布局与键序列：在使用字符串创建键序列时，使用字母指定的键是不区分大小写的，比如 Ctrl+C 与 Ctrl+c 是相同的。但对于 Ctrl++这种类型的快捷键，对于不同键盘布局就会产生不同的结果，比如对于常用的英式键盘布局，因为需要使用 Shift 才能输入 + 符号，所以键序列 Ctrl++需要按下 Ctrl+Shift+=才能被激活，这就使得 Ctrl++与 Ctrl+Shift+=两个键序列是相同的，对于其他键盘布局(比如挪威键盘)可以直接使用 Ctrl++，因此在编程时需要注意键盘布局对键序列的影响，使用 Qt 预定义的标准键序列可以解决键盘布局的问题。
- 4、使用逗号进行分隔的多个键组合(最多 4 个)，可以创建按指定顺序激活的键序列，比如
QKeySequence("Ctrl+C, Ctrl+D"); //按下 Ctrl+C，然后按下 Ctrl+D，用户在输入时，可以按住 Ctrl 不放，然后按下 C，再按下 D，注意：按下 C 之后需要紧接着按下 D。因此此方法创建的快捷键类似于 Ctrl+C+D。

```
QKeySequence("Ctrl+C,D");    //按下 Ctrl+C，然后按下 D
QKeySequence(Qt::CTRL+D, Qt::Key_D);    //按下 Ctrl+C，然后按下 D
QKeySequence("Ctrl+C+D");    //这是无效的键序列
QKeySequence("Ctrl+Alt+D");    //按下 Ctrl+Alt+D
```

- 5、键序列列表：是指同一个操作对应多个快捷键的情形，比如 Ctrl+C 和 Ctrl+Ins 都表示复制的快捷键。Qt 使用键序列列表表来完成以上功能。键序列列表表需要使用静态函数 listFromString()创建，其步骤如下：
QList<QKeySequence> qs;
qs<<QKeySequence::listFromString("Ctrl+D");
qs<<QKeySequence::listFromString("Ctrl+E"); //此时按下 Ctrl+D 和 Ctrl+E 会执行相同的操作。

二、QKeySequence 类中的枚举

Qt::SequenceFormat 枚举(无标志)		
成员	值	说明

QKeySequence::NativeText	0	键序列为特定于平台的字符串。最好使用此枚举值。
QKeySequence::portableText	1	键序列以可移植格式给出，适用于读取和写入文件。

Qt::SequenceMatch 枚举(无标志)

成员	值	说明
QKeySequence::NoMatch	0	键序列不匹配。
QKeySequence::PartialMatch	1	键序列部分匹配，但不相同。
QKeySequence::ExactMatch	2	键序列是相同的。

Qt::StandardKey 枚举(无标志)

预定的标准键

以下为常用的标准键

成员	值	说明	Windows	Mac
QKeySequence::AddTab	19	添加标签	Ctrl+T	Ctrl+T
QKeySequence::Back	13	回退	Alt+Left, Backspace	Ctrl+[
QKeySequence::Backspace	69	删除	无	Meta+H
QKeySequence::Bold	27	加粗字体	Ctrl+B	Ctrl+B
QKeySequence::Close	4	关闭	Ctrl+F4, Ctrl+W	Ctrl+F4, Ctrl+W
QKeySequence::Copy	9	复制	Ctrl+C, Ctrl+Ins	Ctrl+C
QKeySequence::Cut	8	剪切	Ctrl+X, Shift+Del	Ctrl+X, Meta+K
QKeySequence::Forward	14	向前	Alt+Right, Shift+Backspace	Ctrl+]
QKeySequence::HelpContents	1	打开帮助	F1	Ctrl+?
QKeySequence::Italic	28	斜体文本	Ctrl+I	Ctrl+I
QKeySequence::New	6	新建	Ctrl+N	Ctrl+N
QKeySequence::NextChild	20	下一选项卡或子窗口	Ctrl+Tab, Forward, Ctrl+F6	Ctrl+}, Forward, Ctrl+Tab
QKeySequence::Open	3	打开	Ctrl+O	Ctrl+O
QKeySequence::Paste	10	粘贴	Ctrl+V, Shift+Ins	Ctrl+V, Meta+Y
QKeySequence::Preferences	64	打开首选项对话框	无	Ctrl+,
QKeySequence::PreviousChild	21	上一选项卡或子窗口	Ctrl+Shift+Tab, Back, Ctrl+Shift+F6	Ctrl+{, Back, Ctrl+Shift+Tab
QKeySequence::Print	18	打印	Ctrl+P	Ctrl+P
QKeySequence::Quit	65	退出	无	Ctrl+Q
QKeySequence::Redo	12	重做	Ctrl+Y, Shift+Ctrl+Z, Alt+Shift+Backspace	Ctrl+Shift+Z
QKeySequence::Refresh	15	刷新	F5	F5
QKeySequence::Replace	25	查找并替换	Ctrl+H	无
QKeySequence::SaveAs	63	另存为	无	Ctrl+Shift+S
QKeySequence::Save	5	保存	Ctrl+S	Ctrl+S
QKeySequence::SelectAll	26	选择所有文本	Ctrl+A	Ctrl+A
QKeySequence::Deselect	67	取消选择。qt5.1	无	无
QKeySequence::Underline	29	下划线	Ctrl+U	Ctrl+U
QKeySequence::Undo	11	撤销	Ctrl+Z, Alt+Backspace	Ctrl+Z
QKeySequence::WhatsThis	2	激活"what's this"	Shift+F1	Shift+F1
QKeySequence::ZoomIn	16	放大	Ctrl+Plus	Ctrl+Plus

QKeySequence::ZoomOut	17	缩小	Ctrl+Minus	Ctrl+Minus
QKeySequence::FullScreen	66	全屏	F11, Alt+Enter	Ctrl+Meta+F
QKeySequence::Cancel	70	取消	Escape	Escape, Ctrl+.
QKeySequence::UnknownKey	0	未绑定		

以下为不常用的标准键

QKeySequence::Delete	7	删除	Del	Del, Meta+D
QKeySequence::DeleteEndOfLine	60	删除行尾	无	无
QKeySequence::DeleteEndOfWord	59	删除光标后/	Ctrl+Del	无
QKeySequence::DeleteStartOfWord	58	前的一个单词	Ctrl+Backspace	Alt+Backspace
QKeySequence::DeleteCompleteLine	68	删除整行	无	无
QKeySequence::Find	22	查找	Ctrl+F	Ctrl+F
QKeySequence::FindNext	23	查找下一个	F3, Ctrl+G	Ctrl+G
QKeySequence::FindPrevious	24	向前查找	Shift+F3, Ctrl+Shift+G	Ctrl+Shift+G
QKeySequence::InsertLineSeparator	62	插入新行	Shift+Enter	Meta+Enter, Meta+O
QKeySequence::InsertParagraphSeparator	61	插入新段落	Enter	Enter
QKeySequence::MoveToEndOfBlock	41	光标移至块的 末尾/开始	无	Alt+Down, Meta+E
QKeySequence::MoveToStartOfBlock	40		无	Alt+Up, Meta+A
QKeySequence::MoveToEndOfDocument	43	光标移至文档 的末尾/开始	Ctrl+End	Ctrl+Down, End
QKeySequence::MoveToStartOfDocument	42		Ctrl+Home	Ctrl+Up, Home
QKeySequence::MoveToEndOfLine	39	光标移至行的 末尾/行首	End	Ctrl+Right, Meta+Right
QKeySequence::MoveToStartOfLine	38		Home	Ctrl+Left, Meta+Left
QKeySequence::MoveToNextChar	30	光标移至下/ 上一个字符	Right	Right, Meta+F
QKeySequence::MoveToPreviousChar	31		Left	Left, Meta+B
QKeySequence::MoveToNextLine	34	光标移至下/ 上一行	Down	Down, Meta+N
QKeySequence::MoveToPreviousLine	35		Up	Up, Meta+P
QKeySequence::MoveToNextPage	36	光标移至下/ 上一页	PgDown	PgDown, Alt+PgDown, Meta+Down, Meta+PgDown, Meta+V
QKeySequence::MoveToPreviousPage	37		PgUp	PgUp, Alt+PgUp, Meta+Up, Meta+PgUp
QKeySequence::MoveToNextWord	32	光标移至下/ 上一个单词	Ctrl+Right	Alt+Right
QKeySequence::MoveToPreviousWord	33		Ctrl+Left	Alt+Left
QKeySequence::SelectEndOfBlock	65	选择直到文本 块的末尾开头	无	Alt+Shift+Down, Meta+Shift+E
QKeySequence::SelectStartOfBlock	54		无	Alt+Shift+Up, Meta+Shift+A
QKeySequence::SelectEndDocument	57	选择直到文档 结束/开头	Ctrl+Shift+End	Ctrl+Shift+Down, Shift+End
QKeySequence::SelectStartOfDocument	56		Ctrl+Shift+Home	Ctrl+Shift+Up, Shift+Home
QKeySequence::SelectEndOfLine	53	选择直到行尾 /行首。	Shift+End	Ctrl+Shift+Right
QKeySequence::SelectStartOfLine	52		Shift+Home	Ctrl+Shift+Left
QKeySequence::SelectNextChar	44	选择直到下/ 上一个字符	Shift+Right	Shift+Right
QKeySequence::SelectPreviousChar	45		Shift+Left	Shift+Left
QKeySequence::SelectNextLine	48	选择直到下/	Shift+Down	Shift+Down

QKeySequence::SelectPreviousLine	49	上一行	Shfit+Up	Shift+Up
QKeySequence::SelectNextPage	50	选择直到下/ 上一页	Shift+PgDown	Shift+PgDown
QKeySequence::SelectPreviousPage	51		Shift+PgUp	Shigt+PgUp
QKeySequence::SelectNextWord	46	选择直到下/ 上一个单词	Ctrl+Shift+Right	Alt+Shift+Right
QKeySequence::SelectPreviousWord	47		Ctrl+Shift+Left	Alt+Shift+Left

三、QKeySequence 类中的函数

1、构造函数

①、`QKeySequence();` //默认构造函数

②、`QKeySequence(const QString &key, SequenceFormat format = NativeText);`

使用字符串 `key` 及格式 `format` 创建一个键序列，其中字符串"Ctrl"、"Shift"、"Alt"、"Meat"能被识别，字符串通常与 `QObject::tr()` 函数一起使用，比如 `QKeySequence(tr("Ctrl+O"))`;

③、`QKeySequence(int k1, int k2 = 0, int k3 = 0, int k4 = 0);`

使用键 `k1`, `k2`, `k3`, `k4` 创建一个键序列，`k1`, `k2`, `k3`, `k4` 是键 `Qt::Key` 枚举中的值，并可与 `Qt::Modifier` 枚举中的修饰符相结合使用。比如 `QKeySequence(Qt::CTRL+Qt::Key_C)` ;表示 `Ctrl+C`

④、`QKeySequence(StandardKey key);`

使用标准键 `key` 创建一个键序列，具体的键组合取决于平台。比如 `QKeySequence(QKeySequence::Copy)` ;表示用于复制的快捷键，通常为 `Ctrl+C`

⑤、`QKeySequence(const QKeySequence &keySequence);` //复制构造函数

2、`int count() const;` //返回键序列中键组合的数量，最大值为 4。

3、`bool isEmpty() const;` //若键序列为空，则返回 true，否则返回 false

4、`static QList<QKeySequence> keyBindings(StandardKey key);` //静态的

返回键 `key` 的键绑定列表。返回值依平台不同而不同。

5、`SequenceMatch matches(const QKeySequence &seq) const;`

把该键序列与 `seq` 进行匹配。若成功，则返回 `ExactMatch`，若不完全匹配(即部分匹配)，则返回 `PartialMatch`，若完全不匹配则返回 `NoMatch`，若 `seq` 更短，也返回 `NoMatch`。部分匹配，指的是每个键序列中键组合的匹配，具体规则以示例说明：

```
QKeySequence k("F");           QKeySequence k1("Ctrl");       QKeySequence k2("Ctrl+F");
QKeySequence k3("E");           QKeySequence k4("Ctrl+F,E");    QKeySequence k5("e, Ctrl+F");
QDebug() << k.matches(k2);      //不匹配
QDebug() << k1.matches(k2);     //不匹配
QDebug() << k3.matches(k4);     //不匹配
QDebug() << k2.matches(k4);     //部分匹配
QDebug() << k4.matches(k2);     //不匹配，因为 k2 比 k4 短
QDebug() << k3.matches(k5);     //部分匹配
QDebug() << k2.matches(k5);     //不匹配
```

6、`static QKeySequence mnemonic(const QString &text);` //静态的

返回 `text` 中的助记符键序列，若未找到助记符则返回空键序列。比如"Exit"，返回 `Qt::ALT+Qt::Key_X`。

7、`void swap(QKeySequence& other);` //把该键序列与 `other` 交换。

8、`static QKeySequence fromString(const QString &str, SequenceFormat format = PortableText);` //静态的
由字符串 `str` 及格式 `format` 返回一个键序列。

9、`QString toString(SequenceFormat format = PortableText) const;`

依据格式 `format`, 返回键序列的字符串表示形式, 若键序列没有键组合, 则返回空字符串。

比如 `Qt::CTRL+Qt::Key_C`, 该函数会返回"`Ctrl+C`", 若键序列中有多个键组合, 则使用逗号分隔。

10、`static QList<QKeySequence> listFromString(const QString &str, SequenceFormat format = PortableText);` //静态的, qt5.1

由字符串 `str` 及格式 `format` 返回一个键序列的列表。

11、`static QString listToString(const QList<QKeySequence> &list, SequenceFormat format = PortableText);` //静态的, qt5.1

依据格式 `format`, 返回 `list` 的字符串表示形式,

12、重载的运算符函数有: `!=`、`<`、`<=`、`=`、`==`、`>`、`>=`、`[]`

7.5 QAction 类(动作)、QActionGroup 类(动作组)

QAction 继承自 QObject

一、基本规则

- 1、QAction 的基础及使用见 7.1，本小节重点介绍与 QAction 有关的类及函数原型。
- 2、图标(icon 属性)、文本(text 属性)、图标文本(iconText 属性)、快捷键(shortcut 属性)
 - ①、若把动作添加到菜单中，
 - 则菜单项将显示图标(icon 属性)、文本(text 属性)、快捷键(shortcut 属性)。
 - 若 text 属性未设置，则动作的图标文本(即 iconText 属性)将用作文本，否则显示 text 属性的文本。
 - ②、若把动作添加到工具栏中，
 - 若动作有图标、文本、快捷键、图标文本，则工具栏默认只显示图标。
 - 若动作未设置图标，则显示图标文本。
 - 若动作图标和图标文本都未设置，则显示文本(text 属性)，
 - 工具栏显示的文本会被去掉一些字符，比如"&BBB..."会被显示为"BBB"。
 - 快捷键始终不会被显示。
 - 若工具栏的 QToolBar::toolButtonStyle 属性被设置为允许显示文本，则默认显示图标文本(注意：菜单显示的是文本)，若图标文本为未设置，则显示文本。

二、QAction 类中的属性

以下属性除了 checked 的相关信号是 void toggled(bool)信号外，其他属性都是 void changed() 信号。

QAction 属性速查表			
属性名	说明	属性名	说明
autoRepeat	是否启用自动重复	checkable	是否可被选中
text	文本	checked	选中状态
shortcut	快捷键	enabled	是否启用
shortcutContext	快捷键激活方式	font	动作的字体
icon	图标	menuRole	菜单角色
iconText	图标文本	priority	优先级
visible	动作是否可见	statusTip	状态提示
iconVisibleInMenu	图标是否可见	toolTip	工具提示
shortcutVisibleInContextMenu	快捷键是否可见	whatsThis	what's this 帮助

- 1、autoRepeat: bool 访问函数: bool autoRepeat() const; void setAutoRepeat(bool);
描述动作是否可自动重复，若为 true，则只要系统启用了键盘自动重复，则动作启用自动重复。默认为 true。自动重复详见第 2 章键盘事件小节
- 2、text: QString 访问函数: QString text() const; void setText(const QString &);
动作的文本。默认没有文本。

3、**shortcut**: QKeySequence

访问函数: QKeySequence shortcut() const; void setShortcut(const QKeySequence&);
描述动作的主要快捷键。默认没有快捷键。

4、**shortcutContext**: Qt::ShortcutContext

访问函数: Qt::ShortcutContext shortcutContext() const; void setShortcutContext(Qt::ShortcutContext);
描述动作的快捷键的 context 属性(即快捷键的激活方式)。

5、**icon**: QIcon

访问函数: QIcon icon() const; void setIcon(const QIcon&);

描述动作的图标。默认没有图标。若把此属性设置为空(QIcon::isNull()), 则会清除该操作的图标。

6、**iconText**: QString

访问函数: QString iconText() const; void setIconText(const QString&);

该属性描述动作的图标文本。默认为一个空字符串。

7、**visible**: bool

访问函数: bool isVisible() const; void setVisible(bool);

描述该动作是否可见(比如在菜单或工具栏中)。不可见的动作不会显示在菜单中(注意: 不是显示为灰色)。默认为 true(即可见)。

8、**iconVisibleInMenu**: bool

访问函数: bool isIconVisibleInMenu() const; void setIconVisibleInMenu(bool);

描述该动作是否应在菜单中显示图标, 通常菜单仅使用文本, 工具栏使用图标。显示设置此属性将覆盖 Qt::AA_DontShowIconsInMenus 属性(使用 QCoreApplication::setAttribute() 函数设置此属性)

9、**shortcutVisibleInContextMenu**: bool //qt5.10

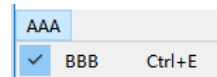
访问函数: bool isShortcutVisibleInContextMenu() const; void setShortcutVisibleInContextMenu(bool);

描述动作是否应在上下文菜单中显示快捷键。显示设置此属性将覆盖 Qt::AA_DontShowShortcutsInContextMenus 属性(使用 QCoreApplication::setAttribute() 函数设置此属性)

10、**checkable**: bool

访问函数: bool isCheckable() const; void setCheckable(bool);

描述该动作是否是可选中动作, 可选中动作是指具有开/关状态的动作, 比如字处理器工具栏中的文字加粗按钮。设置该属性并把该动作添加到菜单后, 当菜单处于选中状态时, 会在菜单的旁边显示一个被选中形式的勾形符号(如图所示)。对于工具栏, 则对应的按钮会呈现出被按下的状态。不是切换动作的动作是命令动作, 命令动作只简单地执行一个操作, 比如文件保存等。默认此属性为 false。



11、**checked**: bool

访问函数: bool isChecked() const; void setChecked(bool);

描述该动作是否被选中, 只能可选中动作才能被选中, 默认为 false(未被选中)。

12、**enabled**: bool

访问函数: bool isEnabled() const; void setEnabled(bool);

该属性描述是否启用该动作。只要设置了 QAction::whatsThis 属性, 则对于禁用的动作, 仍然可以使用 what's this 帮助。当添加动作的部件被禁用(通常显示为灰色)或不可见时, 动作将被禁用, 当某个动作被禁用时, 不能通过它的快捷键激活它。默认为 true(启用)。

13、**font**: QFont

访问函数: QFont font() const; void setFont(const QFont&);

描述动作的字体。默认包含应用程序的默认字体。

14、**menuRole**: MenuRole

访问函数: MenuRole menuRole() const; void setMenuRole(MenuRole);

描述动作在 macOS 应用程序菜单中的角色，默认为 TextHeuristicRole。MenuRole 枚举见下表

QAction::MenuRole 枚举(无标志)		
描述如何把动作移动到 macOS 应用程序的菜单中		
成员	值	说明
QAction::NoRole	0	不应放入应用程序菜单中
QAction::TextHeuristicRole	1	根据 QMenuBar 文档中描述的动作文本放入应用程序菜单中
QAction::ApplicationSpecificRole	2	应放入应用程序特定角色的应用程序菜单中
QAction::AboutQtRole	3	该动作处理 About Qt 菜单项
QAction::AboutRole	4	应放入应用程序中 About 菜单项的位置。
QAction::PreferencesRole	5	应放入应用程序菜单中的"Preference..."菜单项中。
QAction::QuitRole	6	应放入应用程序菜单中的"Quit"菜单项的位置。

- 15、`priority`: Priority 访问函数: Priority priority()const; void setPriority(Priority);
- 描述动作的优先级，比如当工具栏设置了 Qt::ToolButtonTextBesideIcon 模式时，具有 LowPriority 的动作将不会显示文本标签。Priority 枚举见下表

QAction::Priority 枚举的取值		
QAction::LowPriority	QAction::NormalPriority	QAction::HightPriority

- 16、`statusTip`: QString 访问函数: QString statusTip() const; void setStatusTip(const QString &);
- 描述动作的状态提示，状态提示显示在动作的顶级父部件提供的状态栏上。默认为一个空字符串。
- 17、`toolTip`: QString 访问函数: QString toolTip() const; void setToolTip(const QString&);
- 描述动作的工具提示文本。若未设置此属性，则使用动作的文本(text 属性)。默认为动作的文本。
- 18、`whatsThis`: QString 访问函数: QString whatsThis() const; void setWhatsThis(const QString&);
- 描述动作的 what's this 帮助文本。该文本是对动作的简短描述。默认没有该文本。

三、QAction 类中的函数

- 1、`QAction`(QObject* parent = nullptr); //qt5.7 开始参数 parent 是可选的。父部件主要用于销毁该动作
- `QAction`(const QString &text, QObject *parent = nullptr);
- `QAction`(const QIcon& icon, const QString &text, QObject *parent = nullptr);
- 2、`QList`<QKeySequence> `shortcuts`() const;
- `void setShortcuts`(const QList<QKeySequence>& shortcuts);
- 获取或设置激活(或触发)该动作的快捷键列表，列表中的第一个元素是主要快捷键。
- 3、`void setShortcuts`(QKeySequence::StandardKey key); //把标准键 key 设置为该动作的快捷键。
- 4、`QList`<QWidget*> `associatedWidgets`() const; 返回该动作被添加到的部件列表。
- 5、`QWidget*` `parentWidget`() const; //返回此动作的父部件
- 6、`QMenu*` `menu`() const;

```
void setMenu(QMenu* menu);
```

返回和设置包含该动作的菜单。包含菜单的动作可用于创建带有子菜单的菜单项，或插入到工具栏以创建带有弹出式菜单的按钮。

- 7、bool isSeparator() const; 若此动作是分隔符，则返回 true，否则返回 false

```
void setSeparator(bool b);
```

若 b 为 true，则该动作会被视为分隔符。分隔符的表示方式与动作被插入到的部件有关，通常，分隔符会忽略文本、子菜单、图标。

- 8、void activate(ActionEvent event);

使动作发送由 event 指定的信号，即调用此函数会发送 hovered()信号或 triggered()信号。

ActionEvent 枚举见下表

QAction::ActionEvent 枚举(无标志)		
成员	值	说明
QAction::Trigger	0	发送 QAction::triggered()信号
QAction::Hover	1	发送 QAction::hovered()信号

- 9、QActionGroup* actionGroup() const;

```
void setActionGroup(QActionGroup* group);
```

获取和设置该动作的动作组(见后文)，设置动作组后，该动作会自动添加到动作组的列表中。

- 10、bool showStatusText(QWidget* widget = NULL_PTR);

通过发送 QStaturTipEvent 事件到其父部件来更新部件 widget 的相关状态栏。若事件被发送则返回 true，否则返回 false。若 widget 为空，则把该事件发送给该动作的父部件。

- 11、QVariant data() const; //返回动作的数据

```
void setData(const QVariant& userData);      //设置动作的数据。
```

- 12、QList<QGraphicsWidget*> associatedGraphicsWidgets() const; //返回该动作被添加到的部件列表。

示例：键序列列表、使用 QAction::setMenu() 为动作设置子菜单

```
#include<QtWidgets>
#include<QDebug>
int main(int argc, char *argv[]) {      QApplication aa(argc, argv);
    QWidget w;
    QCheckBox *pc=new QCheckBox("FFF",&w);      pc->move(22,77);
    QMenuBar *pmb=new QMenuBar(&w);      QMenu* pml=pmb->addMenu("AAA");
    QAction *pal=pml->addAction("BBB");      /*此处使用的是 QMenu::addAction(), 此时 pml 会拥有该动
                                           作的所有权，因此不必为该动作指定父对象。*/

    //键序列列表的使用
    QList<QKeySequence> qs;
    qs<<QKeySequence::listFromString("Ctrl+D");      qs<<QKeySequence::listFromString("Ctrl+E");
    pal->setShortcuts(qs);      //此时快捷键 Ctrl+D 和 Ctrl+E 对应于同一个动作 pal
    //使用 QAction::setMenu() 函数添加子菜单
    QAction *pa2=pml->addAction("BBB");
    QMenu* pm2=new QMenu("CCC");
    pa2->setMenu(pm2);      //为动作设置菜单后，可向该动作中添加子菜单
```

```

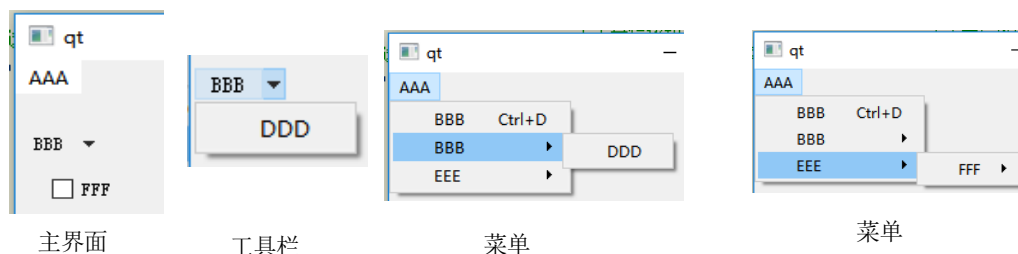
pm2->addAction("DDD");           //菜单 DDD 将作为 BBB 的子菜单
QToolBar *pt=new QToolBar(&w);   pt->move(0, 44);   //创建工具栏
pt->addAction(pa2);              //该工具栏具有下拉列表的菜单
//以下为另一种添加子菜单的方式
QMenu* pm3=new QMenu("EEE",&w);   QMenu* pm4=new QMenu("FFF",&w);
pm1->addMenu(pm3);               pm3->addMenu(pm4);

qDebug()<<pa1->menu();           //此处返回一个空指针，因为 pa1 未使用 setMenu() 设置菜单。
qDebug()<<(pa2->menu()==pm2);    //相比较的结果为 1。
qDebug()<<pa1->associatedWidgets(); //此函数返回的列表只包含一个菜单 pm1
qDebug()<<pa2->associatedWidgets(); /*此函数返回的列表包含一个菜单 pm1，一个工具栏，和一个工具栏按钮。*/

//激活动作 pa1，以使复选框在选中和未选中之间切换
QObject::connect(pa1,&QAction::triggered,pc,&QCheckBox::toggle);
w.resize(333,222);   w.show();   return aa.exec();   }

```

运行结果及说明



按下 Ctrl+D 或 Ctrl+E 都可使复选框 FFF 在选中和未选中之间切换。

四、QAction 类中的槽和信号

- 1、void **setEnabled**(bool b); //槽，若 b 为 true 则禁用动作，若为 false 则启用动作
- 2、void **hover**(); //槽，该槽函数相当于是调用 activate(QAction::Hover)。
- 3、void **toggle**(); //槽，该函数会使动作在选中和未选中之间切换。
- 4、void **trigger**(); //槽，该槽函数相当于是调用 activate(QAction::Trigger)。
- 5、void **toggled**(bool checked); //信号
当可选中动作的选中状态发生改变时，发送此信号。若动作被选中，则 checked 为 true，未选中，则为 false。
- 6、void **triggered**(bool checked = false); //信号
当用户激活动作时，发送此信号。比如当用户单击菜单项、工具栏按钮，或按下快捷键，或调用 trigger()函数时。注意：调用 setChecked()或 toggle()函数不会发送此信号。若该动作是可选中的，则当动作被选中时，checked 为 true，未选中时，为 false。
- 7、void **hovered**(); //信号
当用户高亮显示某个动作时，发送此信号。比如当用户把光标停留在菜单项、工具栏上，或按下快捷键时。
- 8、void **changed**(); //信号

当动作发生变化时发送此信号。由 QAction 类的属性可知，除了 checked 属性外，只要 QAction 类的其他属性发生改变就会发送此信号。若只对某个部件的动作发生变化感兴趣，可以重新实现 QWidget::actionEvent()函数，并对 QEvent::ActionChanged 事件类型进行处理。

示例：QAction 类的信号和槽

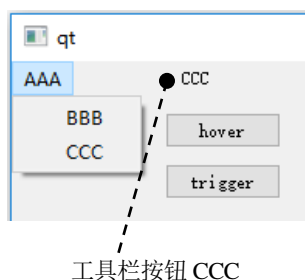
//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>
#include <iostream>
using namespace std;
class B:public QWidget{    Q_OBJECT
public:    QAction *pa1,*pa2;
    B(QWidget *p=0):QWidget(p){
        QMenuBar *pmb=new QMenuBar(this);    QMenu* pml=pmb->addMenu("AAA");
        pa1=pml->addAction("BBB");    pa2=pml->addAction("CCC");
        QPushButton *pb1=new QPushButton("hover",this);    pb1->move(99,33);
        QPushButton *pb2=new QPushButton("trigger",this);    pb2->move(99,66);
        //创建一个工具栏
        QToolBar *pt=new QToolBar(this);    pt->move(99,0);    pt->addAction(pa2);
        //把动作 pa1 的 changed 信号与槽 f1 关联，以验证 changed 信号的发送时机。
        QObject::connect(pa1,&QAction::changed,this,&B::f1);
        QObject::connect(pa2,&QAction::hovered,this,&B::f2);    //验证 hovered 信号
        QObject::connect(pa2,&QAction::triggered,this,&B::f3);    //验证 triggered 信号
        /*调用以下函数相当于发送 QAction::triggered() 信号，该函数需位于以上语句之后，因为这时发送的信号才与相应的槽函数相关联。*/
        pa2->activate(QAction::Trigger);
        //槽函数 hover() 相当于是发送 QAction::hovered() 信号。
        QObject::connect(pb1,&QPushButton::clicked,pa2,&QAction::hover);
        //槽函数 trigger() 相当于是相当于是发送 QAction::triggered() 信号。
        QObject::connect(pb2,&QPushButton::clicked,pa2,&QAction::trigger);    }
public slots:    void f1(){ cout<<"A"<<endl;    }
    void f2(){    cout<<"B"<<endl;    }
    void f3(){    cout<<"C"<<endl;
        /*使 autoRepeat 属性在启用和禁用之间切换。本例使用 autoRepeat 属性的改变，来验证
        QAction::changed() 信号的发送时机。*/
        pa1->setAutoRepeat(pa1->autoRepeat()^1);    }    };
#endif // M_H
```

//m.cpp 文件的内容

```
#include "m.h"
int main(int argc, char *argv[]){    QApplication aa(argc,argv);
    B w;    w.resize(333,222);    w.show();    return aa.exec();    }
```

运行结果及说明



- 1、初次运行程序时会输出 C、A，因为 pa2->activate()函数发送了一个 Triggered 信号，该信号与 f3 关联，而 f3 中的语句 pa1->setAutoRepeat()改变了属性 autoRepeat 的状态，因此这会导致 pa1 发送 changed()信号，从而输出 A。
- 2、当把加亮条移至菜单 CCC 时会输出 B(此时 pa2 发送 hovered 信号)，当选择菜单 CCC 时会输出 C、A(此时 pa2 会发送 triggered 信号)。
- 3、点击按钮 hover 会输出 B，点击按钮 trigger 会输出 C、A。
- 4、把鼠标移至工具栏按钮 CCC 会输出 B，点击工具栏按钮 CCC 会输出 C、A

五、QWidget 类中与 QAction 有关的函数

- 1、`QList<QAction*> actions() const;` //返回该部件的动作列表(可能为空)
- 2、`void addAction(QAction* action);`
`void addActions(QList<QAction*> actions);`
 把 actions 添加到此部件的动作列表中。所有 QWidget 部件都有一个 QAction 列表(这意味着按钮、复选框等部件都可以添加动作)，该列表的默认用法是创建一个上下文菜单，一个 QWidget 应该只有动作列表中的一个动作，并且已经拥有的动作不会导致相同的动作在部件中出现两次。调用该函数的部件不会获得动作的所有权。
- 3、`void insertAction(QAction* before, QAction* action);`
`void insertActions(QAction* before, QList<QAction*> actions);`
 把 action 插入到部件动作列表的动作 before 之前，若 before 是 0 或不是一个有效的动作，则追加动作。
- 4、`void removeAction(QAction* action);` //从部件的动作列表中移除动作 action。
- 5、`virtual void actionEvent(QActionEvent* event);` //虚拟的，受保护的
 只要部件的动作发生变化，就会调用此事件处理程序。

六、QActionGroup 类(动作组)

- 1、QActionGroup 类继承自 QObject，该类用于把动作组合在一起，从而形成一个动作组。
- 2、在同一个动作组中的动作是互斥的，任何时候只有一个动作是活动的。文本的对齐方式(左对齐、右对齐、居中对齐等)就是动作组的一个典型应用。
- 3、动作组中的动作默认是具有排他性的(独占性)，可使用 `QActionGroup::setExclusive(bool)` 函数来改变这一状态。
- 4、使用 `QActionGroup::addAction()` 可把动作添加到动作组中。使用 `QWidget::addActions()` 函数可把动作组添加到 QWidget 部件中。

5、QActionGroup 类中的属性

- ①、**enabled**: bool 访问函数: `bool isEnabled()const; void setEnabled(bool);`
 是否启用动作组。
- ②、**exclusive**: bool 访问函数: `bool isExclusive()const; void setExclusive(bool);`
 描述动作组是否具有排他性，默认为 true(即具有排他性)
- ③、**visible**: bool 访问函数: `bool isVisible() const; void setVisible(bool);`

描述动作组是否可见。

6、QActionGroup 类中的函数

- ①、`QActionGroup(QObject *parent);` //构造函数
- ②、`QAction* addAction(QAction *action);`
`QAction* addAction(const QString &text);`
`QAction* addAction(const QIcon& icon, const QString &text);`
以上函数表示，创建一个动作，并把该动作添加到该动作组中，通常使用 `QAction::setActionGroup()` 函数设置动作组，因此以上函数通常不被使用。
- ③、`QList<QAction *> actions() const;` //返回该动作组的列表(有可能为空)
- ④、`QAction* checkedAction() const;`
返回动作组中当前被选中的动作，若没有动作被选中，则返回 0。
- ⑤、`void removeAction(QAction* action);`
从该动作组中移除动作 `action`，被移除后动作 `action` 将没有父母(这意味着该动作不会被父部件销毁，需手动销毁)。
- ⑥、`void setDisabled(bool b);` //槽，若 `b` 为 `true` 则禁用动作组，若为 `false` 则启用动作组
- ⑦、`void triggered(QAction* action);` //信号
当用户激活动作时，发送此信号。比如当用户单击菜单项、工具栏按钮，或按下快捷键，或调用 `trigger()` 函数时。`action` 是当前激活的动作。
- ⑧、`void hovered(QAction* action);` //信号
当用户高亮显示某个动作时，发送此信号。比如当用户把光标停留在菜单项、工具栏上，或按下快捷键时。`action` 是当前高亮显示的动作。

示例：动作组、分隔符的使用

```
#include<QtWidgets>
int main(int argc, char *argv[]) {    QApplication aa(argc, argv);
    QWidget w;
    QActionGroup *pag1=new QActionGroup(&w);    QActionGroup *pag2=new QActionGroup(&w);
    QActionGroup *pag3=new QActionGroup(&w);
    QMenuBar *pmb=new QMenuBar(&w);    QMenu* pml=pmb->addMenu("AAA");
    //方法 1: 使用 QAction::setActionGroup() 把动作添加到动作组
    QAction *pa1=pml->addAction("BBB");    QAction *pa2=pml->addAction("CCC");
    pa1->setActionGroup(pag1);    pa2->setActionGroup(pag1);
    QAction *paa=pml->addAction("SSS");
    paa->setSeparator(true);    //paa 被设置为分隔符，因此 paa 的文本、图标等会被忽略。
    //方法 2: 使用 QActionGroup::addAction() 把动作添加到动作组
    QAction *pa3=pml->addAction("DDD");    QAction *pa4=pml->addAction("EEE");
    pag2->addAction(pa3);    pag2->addAction(pa4);
    QAction *paal=pml->addAction("SSS");    paal->setSeparator(true);
    //方法 3: 使用 QWidget::addActions() 把动作添加到动作组
    QList<QAction*> pp;
    QAction* pa5=pag3->addAction("FFF");    QAction* pa6=pag3->addAction("GGG");
    pp<<pa5<<pa6;    pml->addActions(pp);
    QAction *paa2=pml->addAction("SSS");    pa2->setSeparator(true);
    //以下动作不属于任何动作组
    QAction *pa7=pml->addAction("HHH");    QAction *pa8=pml->addAction("III");
```

//使各动作都可被选中

```
pa1->setCheckable(true);    pa2->setCheckable(true);    pa3->setCheckable(true);  
pa4->setCheckable(true);    pa5->setCheckable(true);    pa6->setCheckable(true);  
pa7->setCheckable(true);    pa8->setCheckable(true);
```

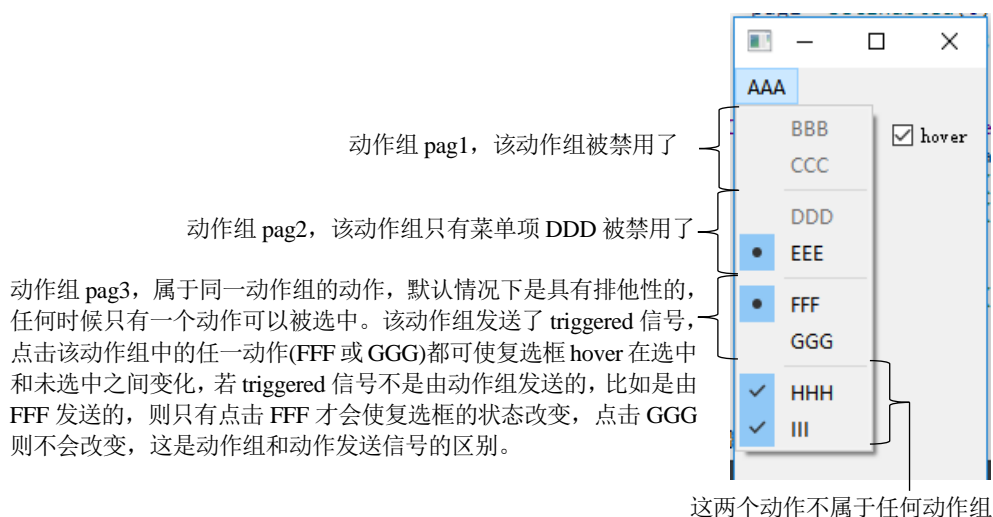
```
pag1->setEnabled(0); //该函数会把属于此动作组中的所有动作一起禁用  
pa3->setEnabled(0); //只禁用动作 pa3。
```

```
QCheckBox *pc=new QCheckBox("hover",&w);pc->move(99,33);
```

//验证动作组发出的 triggered() 信号

```
QObject::connect(pag3,&QActionGroup::triggered,pc,&QCheckBox::toggle);  
w.resize(333,222);    w.show();    return aa.exec(); }
```

运行结果及说明

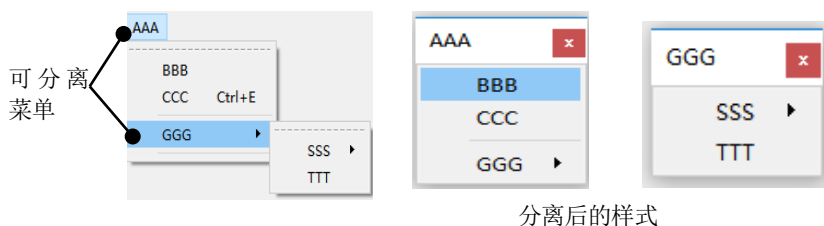


7.6 QMenu 类(菜单)

QMenu 继承自 QWidget

一、基本规则

- 1、可分离菜单：通常在顶部显示为虚线，点击该虚线可使菜单分离，菜单分离后存在于一个单独的窗口中，并会创建一个与原始菜单有相同菜单项目的副本(见下图)。



- 2、菜单应使用 `QMenu::exec()` 或 `QMenu::popup()` 显示，而不应使用 `show()` 显示。
- 3、不能使用单击来触发子菜单的 `triggered()` 信号，但可以使用快捷键来触发。
- 4、菜单都会被视为一个 `QAction`，包括 `QMenu`。
- 5、向菜单中添加各种菜单项的方法
 - 子菜单使用 `addMenu()` 或 `insertMenu()` 添加，子菜单在右侧有一个右箭头，子菜单不一定会有自己的菜单项。
 - 不是子菜单的菜单项使用 `addAction()` 或 `insertAction()` 函数添加。
 - 分隔符使用 `addSeparator()`、`insertSeparator()`、`addSection()`、`insertSection()` 函数添加

二、QMenu 类中的属性

- 1、**icon**: QIcon **访问函数**: QIcon icon() const; void setIcon(const QIcon&);
菜单的图标，默认为空图标
- 2、**separatorsCollapsible**: bool
访问函数: bool separatorsCollapsible() const; void setSeparatorsCollapsible(bool);
描述是否应折叠连续分隔符，若为 `true`，则菜单中的连续分隔符在视觉上会被折叠为单个分隔符，此时菜单开始或结束处的分隔符也会被隐藏。默认为 `true`。



- 3、**tearOffEnabled**: bool **访问函数**: bool isTearOffEnabled() const; void setTearOffEnabled(bool);
描述菜单是否是可分离菜单，默认为 `false`。
- 4、**title**: QString **访问函数**: QString title() const; void setTitle(const QString&);

菜单的标题，默认为空字符串。

- 5、**toolTipsVisible**: bool **访问函数**: bool toolTipsVisible() const; void setToolTipsVisible(bool); //qt5.1
描述菜单的工具提示是否可见。默认为 false。

三、QMenu 类中的函数

1、构造函数

QMenu(QWidget* parent = Q_NULLPTR);

QMenu(const QString &title, QWidget* parent = Q_NULLPTR);

构造函数，为菜单指定父部件，可在父部件销毁时自动销毁该菜单。

2、向菜单中添加和插入菜单项、分隔符的函数

①、QAction* **addAction**(const QString &text);

QAction* **addAction**(const QIcon& icon, const QString &text);

QAction* **addAction**(const QString &text, const QObject* receiver, const char* member,
const QKeySequence& shortcut = 0);

QAction* **addAction**(const QIcon& icon, const QString &text, const QObject* receiver,
const char* member, const QKeySequence& shortcut = 0);

QAction* **addAction**(const QString &text, const QObject* receiver, PointerToMemberFunction method,
const QKeySequence& shortcut = 0); //qt5.6

QAction* **addAction**(const QIcon &icon, const QString &text, const QObject* receiver,
PointerToMemberFunction method, const QKeySequence& shortcut = 0); //qt5.6

QAction* **addAction**(const QString &text, Functor functor, const QKeySequence& shortcut = 0); //qt5.6

QAction* **addAction**(const QString &text, const QObject* context, Functor functor,
const QKeySequence& shortcut = 0); //qt5.6

QAction* **addAction**(const QIcon& icon, const QString &text, Functor functor,
const QKeySequence& shortcut = 0); //qt5.6

QAction* **addAction**(const QIcon& icon, const QString &text, const QObject* context, Functor functor,
const QKeySequence& shortcut = 0); //qt5.6

以上函数表示创建一个新的动作，并把该动作添加到菜单的动作列表中，并返回该动作。QMenu 拥有返回的动作的所有权。各参数的意义如下：

- text: 动作的文本，菜单项会显示该文本。
- icon: 动作的图标，图标会显示在菜单项的左侧。
- receiver、member、method: 表示用于接收动作 triggered()信号的对象和槽函数。其中 member 需使用 SLOT 宏指定，method 可直接使用成员函数的地址指定，其用法与 QObject::connect()函数相同。
- shortcut: 表示该动作的快捷键。
- functor: 表示用于接收动作 triggered()信号函数，只是该参数支持全局函数、静态函数等

②、QAction* **addMenu**(QMenu* menu);

把菜单 menu 作为子菜单添加到该菜单中。注意：该函数返回的是与该菜单所关联的 QAction(即 menuAction()函数返回的动作)。，而且该菜单不会获取 menu 的所有权。

③、QMenu* **addMenu**(const QString &title);

QMenu* **addMenu**(const QIcon& icon, const QString &title);

使用 icon 和 title 创建一个新菜单并添加到该菜单中，并返回新的菜单。该菜单拥有新菜单的所有权。

④、QAction* **insertMenu**(QAction* before, QMenu* menu);

在动作 before 之前插入菜单 menu，并返回与此菜单所关联的 QAction(即 menuAction() 函数返回的动作)。

⑤、QAction* **insertSeparator**(QAction* before);

QAction* **addSeparator**();

QAction* **addSection**(const QString& text); //qt5.1

QAction* **addSection**(const QIcon& icon, const QString& text); //qt5.1

QAction* **insertSection**(QAction* before, const QString &text); //qt5.1

QAction* **insertSection**(QAction* before, const QIcon& icon, const QString &text); //qt5.1

以上函数都会新创建一个分隔符动作，表示把新创建的动作插入到动作 before 之前或添加到菜单中，并返回这个新创建的分隔符动作，该菜单拥有返回的新动作的所有权。

addSection()与 addSeparator()的区别在于，前者新创建的动作拥有文本、图标，而后都创建的动作没有。

3、菜单的显示

⑥、QAction* **exec**();

QAction* **exec**(const QPoint& p, QAction* action = Q_NULLPTR);

QAction* **exec**(QList<QAction*> actions, const QPoint& pos, QAction* action = Q_NULLPTR,

QWidget* parent = Q_NULLPTR); //静态的

void **popup**(const QPoint& p, QAction *atAction = Q_NULLPTR);

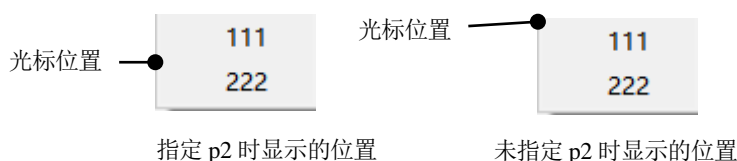
以上函数用于显示菜单，以使动作 action 出现在位置 p 处。第 3 个函数可用于快速创建菜单，其创建的菜单项由 actions 指定，parent 是该菜单的父部件。exec()函数返回用户激活的菜单项(即用户选择的菜单项)。使用以上函数可用于创建快捷菜单(上下文菜单，即点击鼠标右键出现的菜单)，因此通常位置是鼠标光标的位置(即 QCursor::pos())。

示例

//可把以下代码放入处理鼠标事件的处理函数中，以使点击鼠标右键时弹出该菜单：

```
QList<QAction*> q;  
QAction* p1=new QAction("111"); QAction* p2=new QAction("222");  
q<<p1<<p2;  
QMenu::exec(q, QCursor::pos(), p2);
```

结果及说明见下图



4、与可分离菜单有关的函数

⑦、void `hideTearOffMenu()`; 强行隐藏可分离菜单。

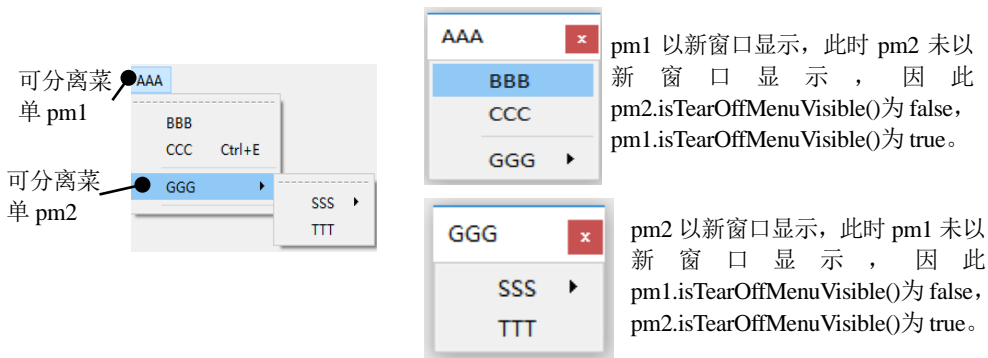
⑧、void `showTearOffMenu`(const QPoint &pos); //qt5.7

void `showTearOffMenu`(); //qt5.7

以上函数表示，强制显示可分离菜单于指定的位置 pos 处(原点位于桌面左上角)，或显示在鼠标光标下。

⑨、bool `isTearOffMenuVisible`() const;

当可分离菜单以新窗口的形式显示时返回 true，若未显示则为 false(见下图)。

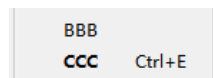


5、默认与高亮显示的动作

⑩、QAction* `defaultAction`() const;

void `setDefaultAction`(QAction* act);

获取和设置菜单的默认动作。默认动作根据 QStyle，会在视觉上有不同的显示，通常以粗体字显示菜单项(如右图)。



⑪、void `setActiveAction`(QAction *act);

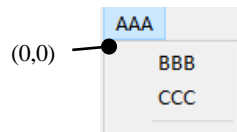
QAction *`activeAction`() const;

以上函数用于设置和返回该菜单中高亮显示的动作，若没有动作则返回 0。

6、其他函数

⑫、QAction* `actionAt`(const QPoint& pt) const;

返回位置 pt 处的菜单项，若没有菜单项，则返回 0。此函数的位置 pt 需在菜单展开的情形下才有效，位置的指定规则见图，需要注意的是可能边框会占两个像素，因此 BBB 的开始位置可能会是(3,3)，而不是(0,0);



⑬、QRect `actionGeometry`(QAction* act) const;

获取该菜单中动作 act 的几何尺寸，坐标的原点与 actionAt()相同。

⑭、QAction * `menuAction`() const; //返回与此菜单关联的动作。

⑮、bool `isEmpty`() const; //若菜单中没有可见的动作则返回 true，否则返回 false。

⑯、void `clear`(); //删除菜单中的所有动作。

7、用于 macOS 上的函数

⑰、void `setAsDockMenu`(); //qt5.2,

通过点击应用程序可停靠栏图标，使菜单设置为可停靠菜单。仅用于 macOS 上。

⑱、NSMenu* `toNSMenu()`; // 返回此菜单的本机 NSMenu，仅在 macOS 上可用。qt5.2

8、QMenu 中的信号

⑲、void `hovered`(QAction* action); //信号

当用户高亮显示某个动作时，发送此信号。比如当用户把光标停留在菜单项、工具栏上，或按下快捷键时。`action` 是当前高亮显示的动作。该信号会作用于该菜单中的所有菜单项，也就是说当该菜单中的任一菜单项被高亮显示时都会发送此信号。

⑳、void `triggered`(QAction* action); //信号

当用户激活动作时，发送此信号。比如当用户单击菜单项、工具栏按钮，或按下快捷键，或调用 `trigger()` 函数时。`action` 是当前激活的动作。该信号会作用于该菜单中的所有菜单项，也就是说当激活该菜单中的任一菜单项时都会发送此信号

(21)、void `aboutToHide`(); //信号，隐藏菜单之前发送此信号。

void `aboutToShow`(); //信号，显示菜单之前发送此信号。

7.7 QMenuBar 类(菜单栏)

QMenuBar 继承自 QWidget

一、基本规则

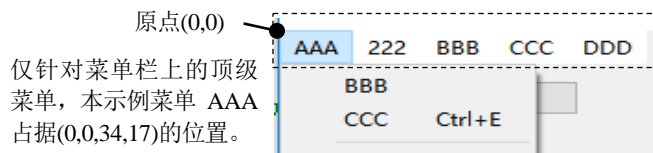
- 1、菜单栏通常不需要手动布局，默认会自动将菜单栏设置到父部件的顶部，并在父部件大小调整时调整其大小。
- 2、使用 QMenuBar::addMenu()函数可把菜单添加到菜单栏上。

二、QMenuBar 类中的属性

- 1、**defaultUp**: bool **访问函数**: bool isDefaultUp() const; void setDefaultUp(bool);
描述菜单的弹出方向，若为 true 则菜单向上弹出，若为 false 则向下弹出，若菜单不适合于屏幕时，会自动从另一个方向弹出。默认为向下弹出。
- 2、**nativeMenuBar**: bool **访问函数**: bool isNativeMenuBar() const; void setNativeMenuBar(bool);
描述菜单栏是否使用平台上的本地菜单栏，目前只支持 macOS 和 Linux，在其他平台上，此属性不起作用，且该属性始终返回 false。设置该属性会覆盖 Qt::AA_DontUseNativeMenuBar 属性(使用 QApplication::setAttribute()设置)

三、QMenuBar 类中的函数

- 1、QMenuBar(QWidget* parent = Q_NULLPTR); //构造函数
 - 2、以下函数用于向菜单栏中添加或插入动作、菜单、分隔符，这些函数的使用方法和作用与 QMenu 类中的相应函数是相同的，只不过中把他们添加到菜单栏上而不是在菜单上。
 - ①、QAction* **addAction**(const QString& text);
QAction* **addAction**(const QString &text, const QObject *receiver, const char *member);
 - ②、QAction* **addMenu**(QMenu* menu);
QMenu* **addMenu**(const QString &title);
QMenu* **addMenu**(const QIcon &icon, const QString &title);
QAction* **insertMenu**(QAction* before, QMenu* menu);
 - ③、QAction* **addSeparator**(); //添加分隔符。
QAction* **insertSeparator**(QAction* before);
 - 3、以下函数需要 QAction 类型的形参，因为菜单栏上的菜单通常是 QMenu 类型的，此时可使用 QMenu::menuAction()函数返回该菜单的动作。
 - ④、QAction* **activeAction**() const; //返回菜单栏上高亮显示的动作
void **setActiveAction**(QAction* act); //把动作 act 设置为高亮显示。
 - ⑤、QAction* **actionAt**(const QPoint& pt) const; //返回位于菜单栏 pt 处的动作
QRect **actionGeometry**(QAction* act) const; //返回菜单栏中的动作 act 的几何尺寸。
- 以上函数指的是菜单栏上的顶级菜单，不包含顶级菜单中的菜单项。具体见下图



4、菜单栏中的部件

⑥、`QWidget* cornerWidget(Qt::Corner corner = Qt::TopRightCorner) const;`

`void setCornerWidget(QWidget* widget, Qt::Corner corner = Qt::TopRightCorner);`

以上函数表示返回或设置位于角落 `corner` 处的部件，使用第 2 个函数可把部件 `widget` 添加到菜单栏第一个菜单项的左侧或最后一个菜单项的右侧，此时菜单栏拥有该部件的所有权。参数 `corner` 应取值为 `Qt::TopRightCorner` 或 `Qt::TopLeftCorner`，对于其他值会导致 Qt 发出警告，并且可能会无效。

5、其他函数

⑦、`void clear();` //删除菜单栏上的所有动作

⑧、`NSMenu* toNSMenu();` //返回此菜单栏的本机 `NSMenu`，仅在 macOS 上可用。qt5.2

⑨、`virtual void setVisible(bool visible);` //虚函数，槽，对 `QWidget::setVisible()` 的重新实现

6、信号

⑩、`void triggered(QAction * action);` //信号

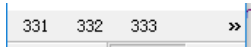
⑪、`void hovered(QAction* action);` //信号

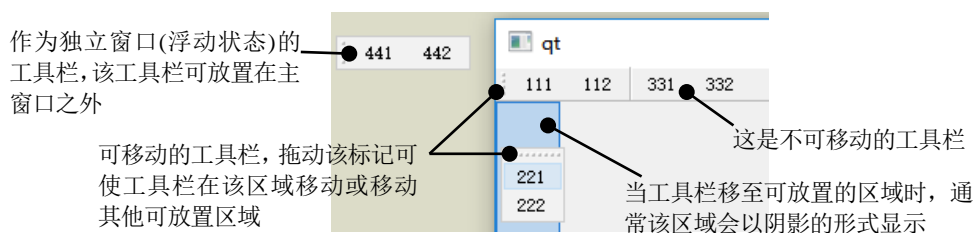
以上信号的发送时机与 `QMenu` 类中的相应信号是相同的(详见 `QMenu` 类)，但是该类中的这两个信号其范围更广，只要激活了菜单栏中的动作及菜单栏上顶级菜单中的菜单项，都会发送 `triggered()` 信号，注意，只有激活动作时才会发送，若激活的是菜单栏上的菜单(`QMenu` 类型的)，是不会发送 `triggered()` 信号的。只要高亮显示菜单栏(包括顶级菜单中的菜单项、菜单栏上的 `QMenu` 类型的菜单)，都会发送 `hovered()` 信号。

7.8 QToolBar 类(工具栏)

QToolBar 类继承自 QWidget

一、基本规则

- 1、QToolBar 类提供了一个可移动的带有一组控件的面板。由于 QToolBar 就是一个 QWidget 部件，默认情况下，该部件就是一个 QWidget，在其中没有任何内容，因此要使 QToolBar 类有意义，需要向其中添加项目。
- 2、因为 QToolBar 类的大部分属性需要在 QMainWindow 类中才有意义，因此本小节只讲解工具栏位于 QMainWindow 之中的情形。
- 3、工具栏可以被固定在某个位置(比如窗口顶部)，也可在工具栏区域内移动。相关函数有 setMovable()、isMovable()、allowedAreas()、isAreaAllowed()
- 4、若工具栏无法显示所包含的所有项目时，会在工具栏上显示一个扩展按钮(如图)。
- 5、向工具栏中添加项目的方法
 - 可把动作(使用 addAction()或 insertAction()添加)作为工具栏上的工具栏按钮。
 - 使用 addSeparator()或 insertSeparator()函数可向工具栏添加分隔符。
 - 还可使用 addWidget()或 insertWidget()向工具栏上插入 QWidget 部件(常见的有 QSpinBox、QComboBox 等)。
- 6、按下工具栏按钮时会发送 actionTriggered()信号。
- 7、工具栏的三种属性：可移动、可放置区域、可作为独立窗口(浮动状态)，见下图



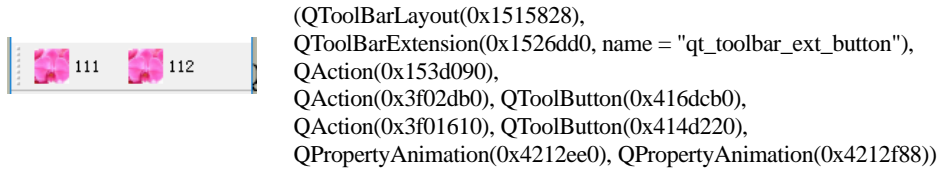
8、工具栏的组成

- 工具栏是一个由多个对象组合而成的部件, 使用 QObject::findChildren()函数可以看到工具栏的组成, 其语法为
- ```
qDebug()<<toolbar->findChildren<QObject*>();
```
- 下图为一个简单的工具栏及使用 qDebug()输出的结果, 由结果可见, 工具栏拥有自己的布局 QToolBarLayout, 并且还有两个工具栏按钮 QToolButton 以及相应的 QAction。
  - 注意: QToolBarLayout、QToolBarExtension 是 Qt 内部类, 在帮助文档中是看不到的, 但在 Qt 安装程序的文件夹中能找到这两个类的源代码, 其路径为

F:\app\Qt5.10.1\5.10.1\Src\qtbase\src\widgets\widgets,

其文件名为 qtoolbarXXX.cpp 或 qtoolbarXXX.h, 在 qtoolbarextension\_p.h 文件中可看

到 `QToolBarExtension` 类是继承自 `QToolButton` 的。



## 二、QToolBar 类中的属性

1、**allowedAreas**: `Qt::ToolBarAreas`

访问函数: `Qt::ToolBarAreas allowedAreas() const; void setAllowedAreas(Qt::ToolBarAreas);`

信号: `void allowedAreasChanged(Qt::ToolBarAreas);`

- 设置工具栏可以被放置在哪些区域，此属性仅在 `QMainWindow` 中才有意义，默认为 `Qt::AllToolBarAreas`。
- 注意：该属性并不能改变工具栏所在的区域，要使工具栏以编程的方式放置在 `QMainWindow` 中的不同区域，应使用 `QMainWindow::addToolBar()` 函数，`QToolBar` 类之中没有相应的函数或属性来设置此功能。
- 枚举 `Qt::ToolBarAreas` 见下表

| Qt::ToolBarArea 枚举                |                  |                                    |                               |
|-----------------------------------|------------------|------------------------------------|-------------------------------|
| 标志: <code>Qt::ToolBarAreas</code> |                  |                                    |                               |
| 作用：描述工具栏在主窗口中的位置                  |                  |                                    |                               |
| 成员                                | 值                | 成员                                 | 值                             |
| <code>Qt::LeftToolBarArea</code>  | <code>0x1</code> | <code>Qt::BottomToolBarArea</code> | <code>0x8</code>              |
| <code>Qt::RightToolBarArea</code> | <code>0x2</code> | <code>Qt::AllToolBarAreas</code>   | <code>ToolBarArea_Mask</code> |
| <code>Qt::TopToolBarArea</code>   | <code>0x4</code> | <code>Qt::NoToolBarArea</code>     | <code>0</code>                |

- 2、**floatable**: `bool`      访问函数: `bool isFloatable()const; void setFloatable(bool);`  
描述工具栏是否可以作为独立的窗口(浮动状态)。默认为 `true`。
- 3、**floating**: `const bool`      访问函数: `bool isFloating() const;`  
描述当前工具栏是否正处于独立窗口的状态(浮动状态)。注：工具栏不能编程的方式设置为浮动状态，即没有 `setFloating()` 函数。
- 4、**iconSize**: `QSize`      访问函数: `QSize iconSize() const; void setIconSize(const QSize&);`  
信号: `void iconSizeChanged(const QSize&);`  
工具栏中的图标大小，默认大小取决于样式。这是图标的最大尺寸，较小的图标不会被放大。
- 5、**movable**: `bool`      访问函数: `bool isMovable() const; void setMovable(bool);`  
信号: `void movableChanged(bool);`  
描述工具栏是否可在工具栏区域或工具栏区域之间移动，默认为 `true`。此属性仅在 `QMainWindow` 中才有意义

## 6、orientation: Qt::Orientation

访问函数: Qt::Orientation orientation() const; void setOrientation(Qt::Orientation);

信号: void orientationChanged(Qt::Orientation);

描述工具栏的方向, 默认为 Qt::Horizontal。不应在 QMainWindow 中使用此属性。

## 7、toolButtonStyle: Qt::ToolButtonStyle

访问函数: Qt::ToolButtonStyle toolButtonStyle() const; void setToolButtonStyle(Qt::ToolButtonStyle);

信号: void toolButtonStyleChanged(Qt::ToolButtonStyle);

描述把 QAction 添加到工具栏上时工具按钮的样式(即工具栏的图标和文本应如何显示), 默认为 Qt::ToolButtonIconOnly。注意, 若是使用 QToolButton::addWidget()把部件添加到 QToolButton 的, 则不受此属性的影响。Qt::ToolButtonStyle 枚举见下表

### Qt::ToolBarStyle 枚举

作用: 描述工具栏的图标和文本应如何显示

| 成员                           | 值 | 说明       |
|------------------------------|---|----------|
| Qt::ToolButtonIconOnly       | 0 | 只显示图标    |
| Qt::ToolButtonTextOnly       | 1 | 只显示文字    |
| Qt::ToolButtonTextBesideIcon | 2 | 文字位于图标旁边 |
| Qt::ToolButtonTextUnderIcon  | 3 | 文字位于图标下方 |
| Qt::ToolButtonFollowStyle    | 4 | 依样式      |

## 示例: QToolBar 类的属性

//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>
using namespace std;
class B:public QWidget{ Q_OBJECT
public: QToolBar *pt1,*pt2,*pt3,*pt4; QPushButton *pbl; QMainWindow *pw;
 B(QWidget *p=0):QWidget(p){
 pw=new QMainWindow; //主窗口
 pbl=new QPushButton("AAA",this);
 pw->setCentralWidget(this); //设置主窗口的中心部件
//初始化工具栏
 pt1=new QToolBar("TTT",pw); //标题 TTT 不会显示为工具栏中的工具按钮文本
 pt2=new QToolBar("TTT",pw); pt3=new QToolBar("TTT",pw); pt4=new QToolBar("TTT",pw);
//把工具栏添加到主窗口 pw 中
 pw->addToolBar(pt1); pw->addToolBar(pt2); pw->addToolBar(pt3);
 pw->addToolBar(Qt::RightToolBarArea,pt4); //把工具栏 pt4 添加到右侧的工具栏区域
//向各工具栏中添加动作
 pt1->addAction(QIcon("F:/1i.png"),"111"); pt1->addAction(QIcon("F:/1i.png"),"112");
 pt2->addAction(QIcon("F:/1i.png"),"221"); pt2->addAction(QIcon("F:/1i.png"),"222");
 pt3->addAction(QIcon("F:/1i.png"),"331"); pt3->addAction(QIcon("F:/1i.png"),"332");
 pt4->addAction(QIcon("F:/1i.png"),"441"); pt4->addAction(QIcon("F:/1i.png"),"442");
//设置各工具栏的属性
 pt2->setToolButtonStyle(Qt::ToolButtonTextBesideIcon); //把文字显示在图标的侧面
 pt3->setToolButtonStyle(Qt::ToolButtonTextOnly); //仅显示文字
 pt4->setToolButtonStyle(Qt::ToolButtonTextUnderIcon); //文字显示在图标的下面
```

```

 pt2->setFloatable(0); //pt2 不能成为独立窗口
 pt3->setMovable(0); //pt3 不可移动
 //pt4 只可在上、左、右三个区域放置，但不能放置在底部
 pt4->setAllowedAreas(Qt::TopToolBarArea|Qt::LeftToolBarArea|Qt::RightToolBarArea);
 pw->resize(333,222); pw->show();
 //点击按钮 pb1 可把工具栏放置在右侧
 QObject::connect(pb1,&QPushButton::clicked,this,&B::f1); }
public slots:
 void f1() {
 //使用 QMainWindow::addToolBar() 以编程的方式设置工具栏的位置，QToolBar 类中没有相应的函数
 pw->addToolBar(Qt::RightToolBarArea,pt1); } };
#endif // M_H

```

//m.cpp 文件的内容

```

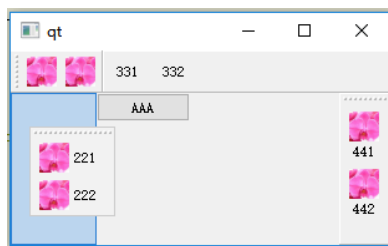
#include "m.h"
int main(int argc, char *argv[]){ QApplication aa(argc,argv); B w; return aa.exec(); }

```

## 运行结果及说明



拖动工具栏到可放置区域，中心部件会自动适应其变化，见下图

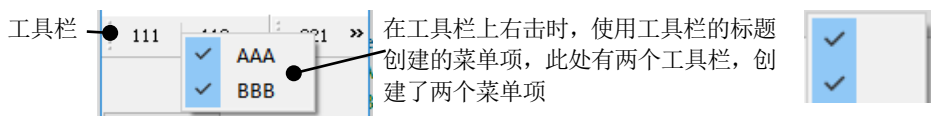


## 三、QToolBar 类中的函数

1、**QToolBar**(QWidget\* parent = Q\_NULLPTR); //构造函数

**QToolBar**(const QString &title, QWidget\* parent = Q\_NULLPTR);

工具栏的标题 **title**: 当在主窗口的工具栏上右击时，会弹出一个弹出菜单，该标题文本会作为该菜单中菜单项的名称，通过取消/选中名称为 **title** 的菜单项的选中符号(通常为勾形符号)，可隐藏/显示该工具栏，见下图，注意：应为工具栏指定标题，否则右击弹出的菜单会显示一个空字符串的菜单列表(见下图右侧)。



- 2、以下函数详见 `QMenu::AddAction()` 函数，其参数意义和用法是相同的，表示创建一个动作并添加到工具栏的末尾，若指定了槽函数，则该动作的 `triggered()` 信号会发送到该槽函数

`QAction* addAction(const QString &text);`

`QAction* addAction(const QIcon &icon, const QString &text);`

`QAction* addAction( const QString &text, const QObject *receiver, const char *member);`

`QAction* addAction( const QIcon &icon, const QString &text, const QObject *receiver, const char *member);`

`QAction* addAction( const QString &text, const QObject *receiver, PointerToMemberFunction method);//qt5.6`

`QAction* addAction(const QIcon &icon, const QString &text, const QObject *receiver, PointerToMemberFunction method);//qt5.6`

`QAction* addAction( const QString &text, Functor functor); //qt5.6`

`QAction* addAction( const QIcon &icon, const QString &text, Functor functor); //qt5.6`

`QAction* addAction( const QString &text, const QObject* context, Functor functor); //qt5.6`

`QAction* addAction( const QIcon &icon, const QString &text, const QObject* context, Functor functor); //qt5.6`

- 3、`QAction* addSeparator();` //把分隔符添加到工具栏的末尾。

`QAction* insertSeparator(QAction* before);` //把分隔符插入到与动作 `before` 关联项目之前。

- 4、`QAction* addWidget(QWidget* widget);`

`QAction* insertWidget (QAction *before, QWidget *widget);`

- 以上函数表示，把部件 `widget` 添加到工具栏的末尾或插入到动作为 `before` 的项目之前。
- 工具栏会获得 `widget` 的所有权。
- 使用以上方法添加的工具栏按钮，不会遵守 `ToolButtonStyle` 属性。
- 应使用 `QAction::setVisible()` 更改 `widget` 的可见性，使用 `QWidget::setVisible()`，`QWidget::show()`、`QWidget::hide()` 将不起作用。

- 5、`QAction* actionAt(const QPoint &p) const;` //返回点 `p` 处的动作，若无该动作则返回 0。

`QAction* actionAt(int x, int y) const;` //返回点 `(x,y)` 处的动作，若无该动作则返回 0。

工具栏中动作的位置会由于工具栏的大小和样式而有所不同，见下图



- 6、`QWidget* widgetForAction(QAction* action) const;`

返回与指定动作关联的部件，使用该函数可以获取工具栏上与 `action` 关联的工具按钮 (`QToolButton`)。

- 7、`QAction* toggleViewAction() const;`

返回可用于隐藏或显示此工具栏的可选中动作，即在工具栏上右击时使用该工具栏的标题创建的菜单项(详见构造函数)所关联的动作。

- 8、`bool isAreaAllowed(QToolBarArea area) const;`

若工具栏在区域 `area` 是可停靠的，则返回 `true`，否则返回 `false`。

9、void `clear()`; //移除工具栏中的所有动作。

## 四、QToolBar 类中的信号

1、void `actionTriggered`(QAction\* action); //信号

当工具栏中的动作被激活时，发送此信号。比如按下包含该动作的工具栏按钮。

2、void `allowedAreasChanged`(Qt::ToolBarAreas allowedAreas); //信号

当工具栏允许被放置的区域发生变化时，发送此信号，参数 `allowedAreas` 为发生变化后的新区域。

3、void `iconSizeChanged`(const QSize& iconSize); //信号

当图标的大小发生变化时，发送此信号，`iconSize` 为新图标的大小。

4、void `movableChanged`(bool movable); //信号

当工具栏变得可移动或不可移动时，发送此信号，若工具栏可移动，则 `movable` 为 `true`，否则为 `false`。

5、void `orientationChanged`(Qt::Orientation orientation); //信号

当工具栏的方向改变时，发送此信号，`orientation` 为工具栏的新方向。

6、void `toolButtonStyleChanged`(Qt::ToolButtonStyle toolButtonStyle); //信号

当工具栏按钮的样式改变时，发送此信号，`toolButtonStyle` 为工具样的新样式。

7、void `topLevelChanged`(bool topLevel); //信号

当 `floating` 属性改变时，发送此信号，若工具栏处于浮动状态(即独立窗口状态)时，`topLevel` 为 `true`，否则为 `false`。

8、void `visibilityChanged`(bool visible); //信号

当工具栏变得可见或不可见时，发送此信号。

## 示例：工具栏的使用

//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>
#include<QDebug>
class B:public QWidget{ Q_OBJECT
public: QToolBar *pt1,*pt2;QAction *pa1,*pa2;
 QPushButton *pb1,*pb2,*pb3,*pb4; QMainWindow *pw;
 B(QWidget *p=0):QWidget(p){
 pw=new QMainWindow; //主窗口
 pb1=new QPushButton("background",this);
 pb2=new QPushButton("toggleView",this); pb2->move(0,33);
 pb3=new QPushButton("no use",this); pb3->move(0,66);
 pb4=new QPushButton("CCC");
 pw->setCentralWidget(this); //设置主窗口的中心部件
 pt1=new QToolBar("AAA",pw); pt2=new QToolBar("BBB",pw);
 //把工具栏添加到主窗口 pw 中
 pw->addToolBar(pt1); pw->addToolBar(pt2);
 //向各工具栏中添加项目
```

```

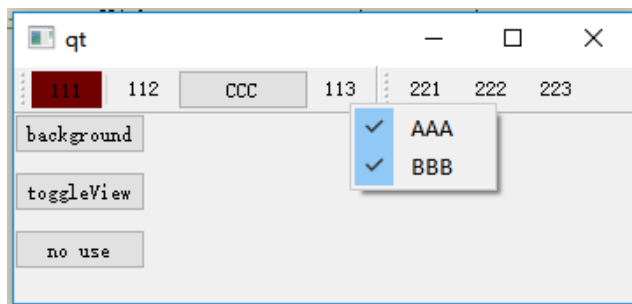
 pal=pt1->addAction("111");
 pt1->addSeparator(); //添加一个分隔符
 pt1->addAction("112");
 pt1->addWidget(pb4); //添加一个按钮部件
 pt1->addAction("113");
 pt2->addAction("221"); pt2->addAction("222"); pt2->addAction("223");
 pw->resize(333,222); pw->show();
//获取在工具栏上右击时使用工具栏 pt1 的标题 AAA 创建的菜单项所关联的动作
 pa2=pt1->toggleViewAction();

 QObject::connect(pt1,&QToolBar::actionTriggered,this,&B::f1);
 QObject::connect(pb1,&QPushButton::clicked,this,&B::f2);
 QObject::connect(pb2,&QPushButton::clicked,this,&B::f3);
 QObject::connect(pb3,&QPushButton::clicked,this,&B::f4); }
public slots: void f1() { qDebug() << "A";}
 void f2() { //设置工具栏 pt1 中与动作 pal 相关联的部件的背景色(需使用样式表设置)。
 pt1->widgetForAction(pal)->setStyleSheet("background-color: rgb(111,1,1);");}
 void f3() {pa2->trigger();} //显示或隐藏动作 pa2 所关联的工具栏 pt1
 void f4() { pa2->setChecked(0); } /*setChecked() 函数不会触发动作的 triggered() 信号,
 因此此处把动作设置为未选中状态不会使工具栏 pt1 隐藏。*/
};
#endif // M_H

//m.cpp 文件的内容
#include "m.h"
int main(int argc, char *argv[]) { QApplication aa(argc,argv); B w; return aa.exec(); }

```

## 运行结果及说明



- 1、点击工具栏 pt1 上的 111、112、113 都会输出 A，但是点击 CCC 不会输出 A，因为 CCC 不是工具栏上的动作。
- 2、点击按钮 background 会使工具栏 pt1 上的工具按钮 111 的背景色设置为红色。
- 3、点击按钮 toggleView 可使工具栏 pt1 在隐藏和显示之间转换，点击该按钮相当于选中图示在工具栏上右击弹出的菜单中的 AAA 菜单项。
- 4、点击 no use 按钮只会使弹出菜单 AAA 的菜单项变为未选中状态，但工具栏 pt1 不会因此而被隐藏，因为 setChecked() 函数不会发送 triggered() 信号。



## 7.9 QStatusBar 类(状态栏)

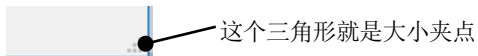
QStatusBar 类继承自 QWidget

### 一、基本规则

- 1、状态栏主要用于向用户显示一些程序的状态信息，状态栏通常位于窗口底部，
- 2、由于状态栏就是一个 QWidget 部件，默认情况下，状态栏什么也不会显示，因此需要自行编写代码处理状态栏上显示的信息，比如需要处理当鼠标进入某个部件时在状态栏显示该部件的一些信息，当离开时清除该信息的显示等。
- 3、由于 QMainWindow(主窗口)已经由 Qt 内部实现了状态栏的功能(比如状态位始终位于窗口底部，当鼠标进入部件时显示信息，离开时清除信息等)，因此在主窗口中使用状态栏更方便，当然程序员也可以实现自己的状态栏(见后文)。
- 4、主窗口(QMainWindow)只能有一个状态栏。使用 QMainWindow::setStatusBar()函数可设置状态栏。
- 5、状态栏可以显示 3 种类型的信息，
  - 临时信息：通常用于显示部件的状态信息(即使用 setStatusTip()函数设置的信息)，临时信息的显示方式通常是，当鼠标进入该部件时显示该信息，离开该部件时清除该信息。临时信息使用 QStatusBar::showMessage()函数进行设置。
  - 正常信息：正常信息会一直显示在状态栏，但是当需要显示临时信息时，临时信息可能会隐藏正常信息。正常信息通过一个 QWidget 部件来进行显示(比如 QLabel、QProgressBar 等)，可使用 QStatusBar::addWidget()函数向状态栏添加这些部件。
  - 永久信息：永久信息显示在状态栏的最右侧，永久信息不会被临时信息隐藏。永久信息也通过一个 QWidget 部件来进行显示(比如 QLabel、QProgressBar 等)，可使用 QStatusBar::addPermanentWidget()函数向状态栏添加这些部件

### 二、QStatusBar 类中的属性

**sizeGripEnabled:** bool      **访问函数:** bool isSizeGripEnabled() const; void setSizeGripEnabled(bool);  
描述是否启用右下角的大小夹点(QSizeGrip)，默认为启用。



### 三、QStatusBar 类中的函数

- 1、**QStatusBar**(QWidget\* parent = Q\_NULLPTR);
- 2、void **showMessage**(const QString &message, int timeout = 0); //槽  
隐藏正常信息，并显示指定毫秒的临时信息(即消息 message 只显示 timeout 毫秒，然后被删除)，若 timeout 为 0，则信息会一直显示，直到调用 clearMessage()删除信息或再次调用 showMessage()更改信息。
- 3、void **addWidget**(QWidget\* widget, int stretch = 0);  
int **insertWidget**(int index, QWidget\* widget, int stretch = 0);

把部件 `widget` 添加或插入到状态栏，`widget` 可能会被临时信息隐藏。若 `widget` 不是该状态栏的子部件，则重新指定该部件的父对象，`stretch` 表示拉伸因子，0 表示部件以最小空间的形式显示。

4、void `addPermanentWidget`(QWidget\* widget, int stretch =0);

int `insertPermanentWidget`(int index, QWidget\* widget, int stretch =0);

把部件 `widget` 永久添加或插入到状态栏，`widget` 位于状态栏的最右侧。永久意味着不会被临时信息所隐藏。若 `widget` 不是该状态栏的子部件，则重新指定该部件的父对象，`stretch` 表示拉伸因子，0 表示部件以最小空间的形式显示。

5、void `removeWidget`(QWidget\* widget);

从状态栏中移除而不删除(即只是隐藏)部件 `widget`(包括永久部件)，若需要再次添加该部件，须同时调用 `addWidget`()或 `addPermanentWidget`()和 `show`()函数。

6、QString `currentMessage`() const; //返回当前显示的临时信息，若无信息，则为空字符串

7、void `clearMessage`(); //槽，删除使用 `showMessage`()显示的临时信息。

8、void `hideOrShow`(); //受保护的，该函数由 `showMessage`()和 `clearMessage`()函数使用。

9、void `messageChanged`(const QString &message); //信号

当临时信息更改时，发送此信号。`message` 为更改后的新信息。

10、void `reformat`(); //受保护的，

## 示例：状态栏的使用

//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>
class B:public QMainWindow{ Q_OBJECT
public: QMainWindow pw; QStatusBar *ps1; QLabel *pt; QPushButton *pb1,*pb2,*pb3,*pb4,*pb5;
 B(QWidget *p=0):QMainWindow(p) {
 ps1=new QStatusBar(this); //状态栏
 //设置中心部件
 QWidget *pw1=new QWidget; pb1=new QPushButton("remove",pw1);
 pb2=new QPushButton("add",pw1); pb2->move(77,0);
 //以下部件会添加到状态栏
 pb3=new QPushButton("CCC"); pb4=new QPushButton("DDD");
 pb5=new QPushButton("EEE"); pt=new QLabel("Label");
 //向主窗口中添加状态栏和中心部件
 setStatusBar(ps1); setCentralWidget(pw1);
 //设置状态栏的正常信息(使用 QWidget 部件显示)
 ps1->addWidget(pt); ps1->addWidget(pb3);
 //设置状态栏的永久信息(使用 QWidget 部件显示)
 ps1->addPermanentWidget(pb4); ps1->addPermanentWidget(pb5);
 //设置各部件的状态信息提示
 pb1->setStatusTip("hide widget DDD"); pb2->setStatusTip("add permanent widget DDD");
 QObject::connect(pb1,&QPushButton::clicked,this,&B::f1);
 QObject::connect(pb2,&QPushButton::clicked,this,&B::f2); }
 public slots: void f1() { ps1->removeWidget(pb4); }
 void f2() { ps1->addPermanentWidget(pb4);
 pb4->show(); } //必须使用 show() 否则重新添加的 pb4 将不可见
```

```

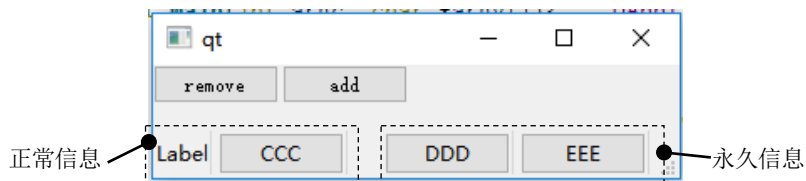
 };
#endif // M_H

//m.cpp 文件的内容
#include "m.h"
int main(int argc, char *argv[]) { QApplication aa(argc, argv);
 B w; w.resize(333, 222); w.show(); return aa.exec(); }

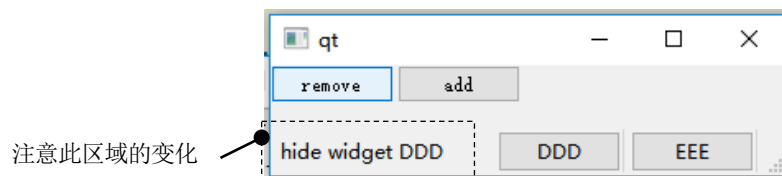
```

## 运行结果及说明

### 1、初始界面如下图



### 2、当把鼠标移至按钮 remove 上时，状态栏的正常信息区域，会被该按钮的状态信息提示所隐藏(如下图)，当把鼠标移至该按钮之外时，状态栏的正常信息又会被重新显示出来。



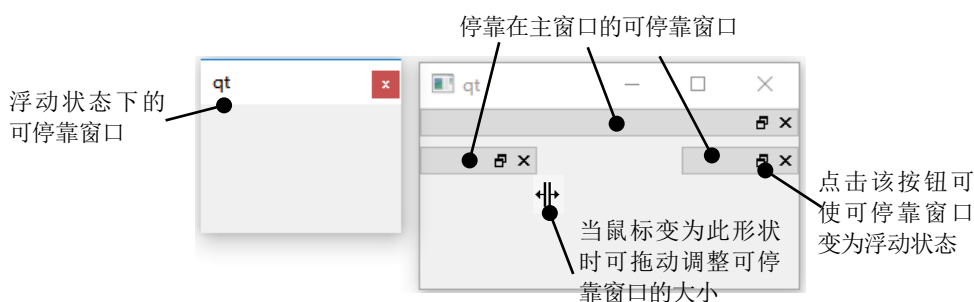
### 3、点击 remove 按钮，会隐藏永久信息区域的 DDD 按钮，点击 add 按钮又会把该按钮重新添加到该区域中。

## 7.10 QDockWidget 类(可停靠窗口、悬浮窗口)

QDockWidget 类继承自 QWidget。

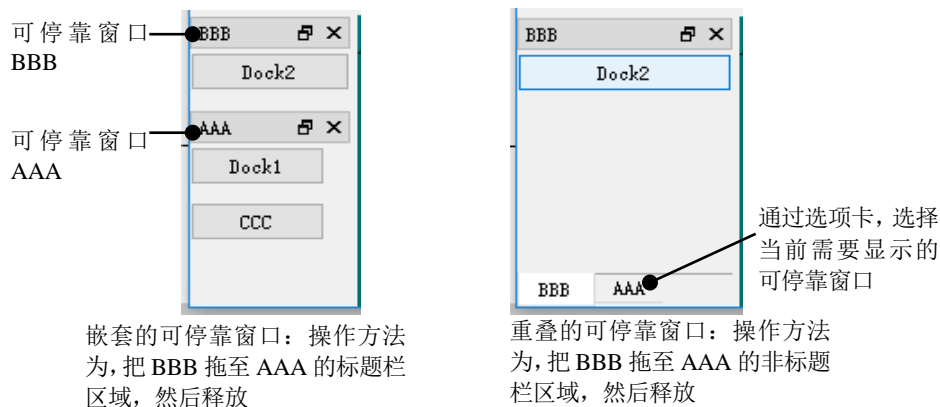
### 一、基本规则

- 1、QDockWidget 实现了可停靠窗口，可停靠窗口可以停靠在主窗口中，也可作为一个浮动窗口使用，见下图



以上的可停靠窗口和主窗口都未包含任何其他部件(仅有一个标题栏)

- 2、在主窗口中的可停靠窗口还可重叠和嵌套，见下图



- 3、由于 QDockWidget 就是一个 QWidget 部件，默认情况下，该部件只提供了一个有标题栏的框架窗口样式，在其中没有任何内容，因此要使 QDockWidget 类有意义，需要向其中添加部件，使用 QDockWidget::setWidget()函数可向其中添加部件。
- 4、QDockWidget 也是由多个对象组合而成的部件，没有任何 QWidget 部件的可停靠窗口，使用以下语句

```
qDebug()<<"dock->findChildren<QObject*>()";
```

输出结果如下：

```
(QDockWidgetLayout(0x141f360),
```

```
//可停靠窗口使用的布局
```

```
QDockWidgetTitleButton(0x142b970, name = "qt_dockwidget_floatbutton"), //可停靠窗口上的 Float 按钮
```

```
QDockWidgetTitleButton(0x142bb68, name = "qt_dockwidget_closebutton"), //可停靠窗口上的 X 按钮
QAction(0x1428690)) //可停靠窗口的动作
```

二、QDockWidget 类中的属性

1、allowedAreas: Qt::DockWidgetAreas

**访问函数:** Qt::DockWidgetAreas allowedAreas() const; void setallowedAreas(Qt::DockWidgetAreas);

**信号:** allowedAreasChanged(Qt::DockWidgetAreas)

可停靠窗口允许放置的区域，默认为 Qt::AllDockWidgetAreas，注意：该属性并不能改变可停靠窗口所在的区域，要使可停靠窗口以编程的方式放置在 QMainWindow 中的不同区域，应使用 QMainWindow::addDockWidget()函数，QDockWidget 类之中没有相应的函数或属性来设置此功能。

枚举 Qt::DockWidgetArea 见下表

| Qt::DockWidgetArea 枚举   |     |                          |                     |
|-------------------------|-----|--------------------------|---------------------|
| 标志: Qt::DockWidgetAreas |     |                          |                     |
| 作用: 描述可停靠窗口在主窗口中的位置     |     |                          |                     |
| 成员                      | 值   | 成员                       | 值                   |
| Qt::LeftDockWidgetArea  | 0x1 | Qt::BottomDockWidgetArea | 0x8                 |
| Qt::RightDockWidgetArea | 0x2 | Qt::AllDockWidgetAreas   | DockWidgetArea_Mask |
| Qt::TopDockWidgetArea   | 0x4 | Qt::NoDockWidgetArea     | 0                   |

2、features: DockWidgetFeatures

**访问函数:** DockWidgetFeatures features() const; void setFeatures(DockWidgetFeatures);

**信号:** featuresChanged(QDockWidget::DockWidgetFeatures);

描述可停靠窗口的特征，即，是否可移动、关闭、悬浮等，默认是 DockWidgetClosable、DockWidgetMovable、DockWidgetFloatable。注意：没有关闭单个特片的函数，要关闭这些选项需要进行运算，比如要清除 DockWidgetMovable 应使用如下语句

```
dock->setFeatures(dock->features()&0x05); //0x05 的二进制形式为 0000 0101
```

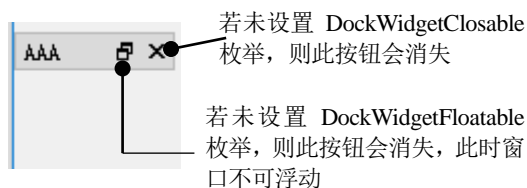
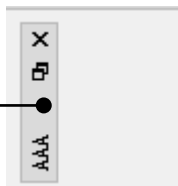
或 dock->setFeatures(QDockWidget::DockWidgetClosable|QDockWidget::DockWidgetFloatable)

枚举 DockWidgetFeature 见下表

| QDockWidget::DockWidgetFeature 枚举       |      |                                                                     |
|-----------------------------------------|------|---------------------------------------------------------------------|
| 标志: QDockWidget::DockWidgetFeatures     |      |                                                                     |
| 作用: 描述可停靠窗口的特征                          |      |                                                                     |
| 成员                                      | 值    | 说明                                                                  |
| QDockWidget::DockWidgetClosable         | 0x01 | 可停靠窗口可以被关闭。                                                         |
| QDockWidget::DockWidgetMovable          | 0x02 | 可停靠窗口可移动                                                            |
| QDockWidget::DockWidgetFloatable        | 0x04 | 可停靠窗口可从主窗口中分离作为一个独立窗口(即，可浮动)                                        |
| QDockWidget::DockWidgetVerticalTitleBar | 0x08 | 可停靠窗口在左侧显示一个垂直标题栏。                                                  |
| QDockWidget::AllDockWidgetFeatures      |      | 已弃用，这是 DockWidgetClosable、DockWidgetMovable、DockWidgetFloatable 的组合 |

|                                   |      |                  |
|-----------------------------------|------|------------------|
| QDockWidget::NoDockWidgetFeatures | 0x00 | 可停靠窗口无法关闭、移动、浮动。 |
|-----------------------------------|------|------------------|

DockWidgetVerticalTitleBar 枚举的效果，但是，当窗口悬浮显示时标题栏不会显示在左侧



若未设置 DockWidgetMovable 枚举，可停靠窗口在外观上没变化，但窗口不可被移动

3、floating: bool                      访问函数: bool isFloating() const; void setFloating(bool);  
描述可停靠窗口的浮动状态，可使用此属性以编程的方式使可停靠窗口浮动。

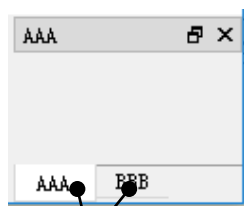
4、windowTitle: QString  
访问函数: QString windowTitle() const; void setWindowTitle(const QString&);  
信号: windowTitleChanged(const QString &);  
可停靠窗口的标题(见构造函数对标题的讲解)，默认为空字符串。

### 三、QDockWidget 类中的函数

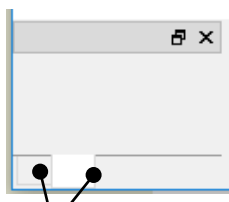
1、QDockWidget(QWidget\* parent = Q\_NULLPTR, Qt::WindowFlags flags = Qt::WindowFlags());

QDockWidget(const QString &title, QWidget\* parent = Q\_NULLPTR,  
Qt::WindowFlags flags = Qt::WindowFlags());

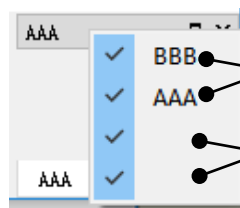
可停靠窗口的标题 title: 当在主窗口中的可停靠窗口标题栏上右击时，会弹出一个弹出菜单，该标题文本会作为该菜单中菜单项的名称，通过取消/选中名称为 title 的菜单项的选中符号(通常为勾形符号)，可隐藏/显示该可停靠窗口，另外当可停靠窗口停靠在可停靠窗口中时(即重叠的可停靠窗口)，标题也会作为选项卡的名称，见下图，注意：应为可停靠窗口指定标题，否则菜单项和选项卡都会以空字符串代替。



选项卡上的文字



选项卡上没有文字



有标题的可停靠窗口

无标题的可停靠窗口

有标题时重叠的可停靠窗口

无标题时重叠的可停靠窗口

2、void setWidget(QWidget\* widget);

把部件 widget 添加到可停靠区域中，后添加的部件会覆盖之前添加的部件，从而使之前添加的部件不可见。

3、QWidget\* widget() const;

返回可停靠窗口中的部件，若使用 setWidge()向可停靠窗口中添加了多个部件，则该函数

只能获取最后添加的那个部件。

4、void **setTitleBarWidget**(QWidget\* widget);

QWidget\* **titleBarWidget**() const;

- 以上函数用于设置和获取可停靠窗口的自定义标题栏。
- 自定义标题栏可以是任何 QWidget 部件，
- 设置自定义标题栏需要对该标题栏部件进行必要的处理，比如需要处理点击该标题栏时的鼠标事件(比如当拖动标题栏时移动，当双击时停靠等鼠标事件)。
- 自定义的标题栏部件还必须具有有效的 QWidget::sizeHint() 和 QWidget::minimumSizeHint() 尺寸，否则可能会因 sizeHint() 和 minimumSizeHint() 为 0 而无法显示自定义的标题栏部件。可通过 setMinimumSize() 来间接设置大小提示和最小大小提示。
- 可停靠窗口的标题栏是无法移除的，要移除标题栏，可把默认构造的 QWidget 部件(其最小大小默认为 0) 设置为可停靠窗口的标题栏，从而达到类似移除标题栏的效果。

5、bool **isAreaAllowed**(Qt::DockWidgetArea area) const;

若在 area 区域允许被停靠，则返回 true，否则返回 false。

6、QAction\* **toggleViewAction**() const;

返回可用于隐藏或显示该可停靠窗口的可选中动作，即在可停靠窗口的标题栏上右击时使用该可停靠窗口的标题创建的菜单项(详见构造函数)所关联的动作。

## 四、QDockWidget 类中的信号

1、void **topLevelChanged**(bool topLevel); //信号

当 floating 属性改变时，发送此信号，若可停靠窗口当前是浮动的，则 topLevel 为 true，否则为 false。

2、void **visibilityChanged**(bool visible); //信号

当可停靠窗口变得可见或不可见时，发送此信号。当在重叠的可停靠窗口中切换其选项卡使可停靠窗口可见或不可见时，也会发送此信号。

3、void **allowedAreasChanged**(Qt::DockWidgetAreas allowedAreas); //信号

当 allowedAreas 属性(即允许放置的区域)更改时，发送此信号，allowedAreas 为更改后的新区域。

4、void **dockLocationChanged**(Qt::DockWidgetArea area); //信号

当可停靠窗口移动到另一个可停靠区域或移动到当前可停靠区域的另一位置时，发送此信号，当以编程的方式改变位置时，也会发送此信号。

5、void **featuresChanged**(QDockWidget::DockWidgetFeatures features); //信号

当 features 属性改变时，发送此信号，参数 features 为改变后的新值。

### 示例：QDockWidget 的使用及自定义标题栏部件

//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>
#include<QDebug>
```



```

class B:public QWidget{ Q_OBJECT //最好继承 QMainWindow 类
public: QDockWidget *pd1,*pd2,*pd3; QMainWindow *pw;
 QPushButton *pb,*pb1,*pb2,*pb3,*pb4,*pb5,*pb6;
 B(QWidget *p=0):QWidget(p) {
 pw=new QMainWindow;
 QWidget *pw1=new QWidget; QWidget *pw2=new QWidget;
 QWidget *pw3=new QWidget; QWidget *pw4=new QWidget;
 pd1=new QDockWidget; pd2=new QDockWidget; pd3=new QDockWidget;
//布局中心部件
 pb=new QPushButton("AAA",pw1); pw->setCentralWidget(pw1);
 pw1->setStyleSheet("background-color: rgb(1,111,111);"); //设置中心部件的背景色
//布局可停靠窗口 pd1
 pw->addDockWidget(Qt::LeftDockWidgetArea,pd1); //把 pd1 添加到主窗口的左侧
 pb1=new QPushButton("Dock1",pw2);
 pb2=new QPushButton("CCC",pw2); pb2->move(0,33);
 pd1->setWidget(pw2); //设置 pd1 中的内容
 pd1->setWindowTitle("AAA"); //设置 pd1 的标题
 //获取 QDockWidget 布局的大小约束(默认为 SetMinAndMaxSize), 实际使用时有可能会更改此约束
 qDebug()<<pd1->layout()->sizeConstraint();
//布局可停靠窗口 pd2
 pw->addDockWidget(Qt::RightDockWidgetArea,pd2);
 pb3=new QPushButton("Float",pw3); pb4=new QPushButton("Hide",pw3); pb4->move(77,0);
 pb5=new QPushButton("Dock2");
 pw3->setStyleSheet("background-color: rgb(111,1,1);"); //设置 pw3 的背景色
 pd2->setTitleBarWidget(pw3); //把 pw3 设置为 pd2 的标题栏(设置自定义标题栏)
 //设置可停靠窗口和标题栏部件的最小大小, 若设置不合适, 会使可停靠窗口无法显示某些部件。
 pd2->setMinimumSize(155,144); pw3->setMinimumSize(155,55);
 //pw3->resize(111,111); //此函数设置的大小不起作用。
 //设置 pd2 的标题及其内容
 pd2->setWindowTitle("BBB"); pd2->setWidget(pb5);
//布局可停靠窗口 pd3
 pw->addDockWidget(Qt::TopDockWidgetArea,pd3);
 pb6=new QPushButton("Dock3"); pd3->setWidget(pb6);
 pd3->setTitleBarWidget(pw4); /*因为 pw4 的最小大小默认为 0, 因此设置 pw4 为 pd3 的标题栏, 会使 pd3 的标题栏不可见(因为大小为 0)。*/
 //点击 pb 使 pd3 处于悬浮状态, 主要用于验证 pd3 是一个可停靠窗口
 QObject::connect(pb,&QPushButton::clicked,this,&B::f);
 QObject::connect(pb3,&QPushButton::clicked,this,&B::f3);//点击 pb3 使 pd2 处于悬浮状态
 QObject::connect(pb4,&QPushButton::clicked,this,&B::f4);//点击 pb3 使 pd2 停靠在主窗口
 pw->resize(333,222); pw->show(); }
public slots: void f() { pd3->setFloating(1); }
 void f3() { pd2->setFloating(1); }
 void f4() { pd2->setFloating(0); } };
#endif // M_H

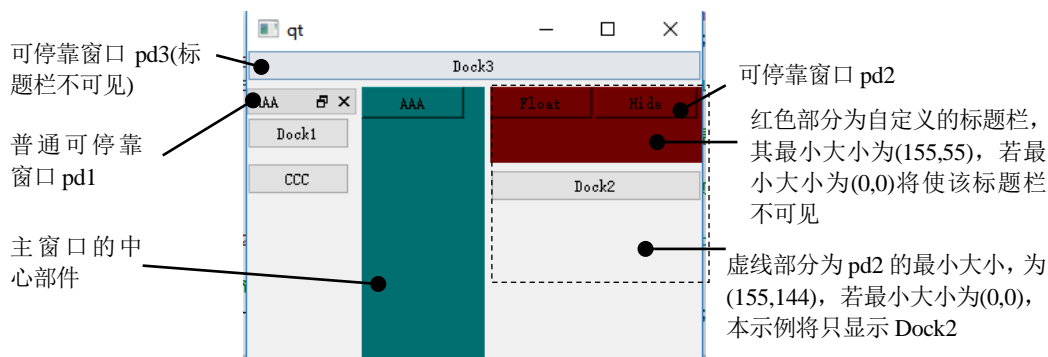
//m.cpp 文件的内容
#include "m.h"
int main(int argc, char *argv[]){ QApplication aa(argc,argv); B w; return aa.exec(); }

```

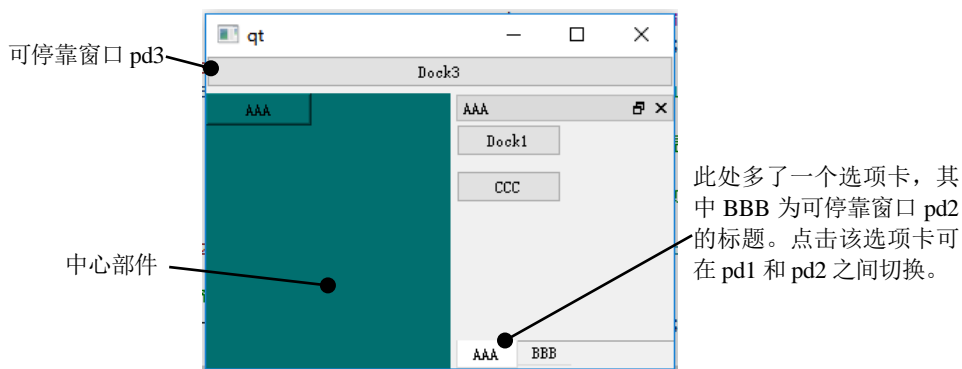
运行结果及说明

1、初次运行时的界面

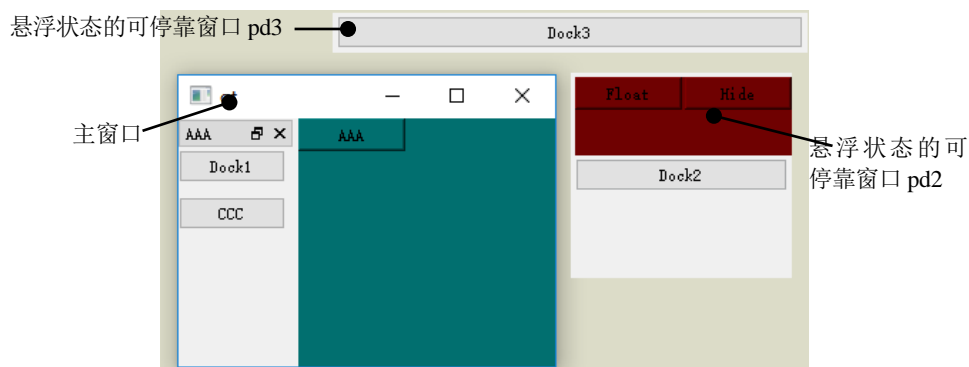




2、把 pd1 拖至 pd2 下面的效果见下图



3、点击 pd2 自定义标题栏中的按钮 Float 和中心部件中的按钮 AAA, 并把主窗口移至其他位置, 此时可看到 pd3 和 pd2 现在成为了独立的悬浮窗口, 但是无法移动这两个窗口, 如下图



4、点击 pd2 自定义标题栏上的按钮 Hide 可使 pd2 重新停靠到主窗口原来的位置。

## 7.11 QMainWindow 类(主窗口)

QMainWindow 类继承自 QWidget。主窗口把前面各小节讲解的部件组织在一个窗口中，关于主窗口的基本规则见 7.1，本小节主要讲解 QMainWindow 类中的属性和函数。

### 一、QMainWindow 类中的属性

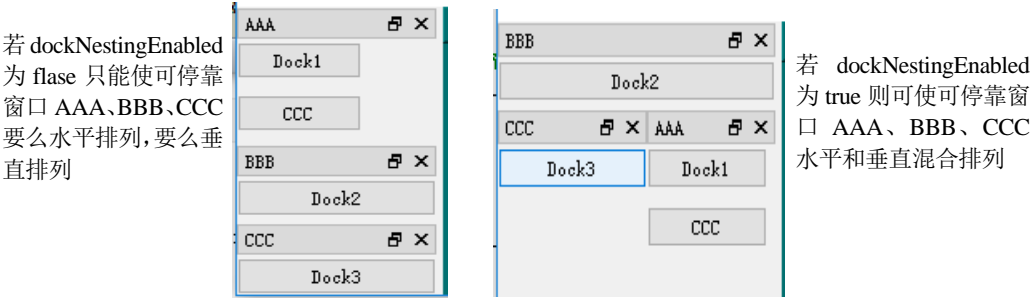
1、**animated**: bool            **访问函数**: bool isAnimated() const; void setAnimated(bool);

是否具有动画效果。当在主窗口上拖动工具栏和可停靠窗口时，主窗口会调整其内容，若此属性为 true 则会以平滑的动画效果移动其内容。默认为 true，设置此属性与 setDockOptions(QMainWindow::AnimatedDocks)相同。

2、**dockNestingEnabled**: bool

**访问函数**: bool isDockNestingEnabled() const; void setDockNestingEnabled(bool);

此属性影响嵌套的可停靠窗口，若该属性为 false，则嵌套的可停靠窗口只能在其可停靠区域放置于单独的方向(水平或垂直)，若此属性为 true，则可在多个方向上放置，设置此属性与 setDockOptions(QMainWindow::AllowNestedDocks)相同，见下图



3、**dockOptions**: DockOptions

**访问函数**: DockOptions dockOptions() const; void setDockOptions(DockOptions);

该属性描述主窗口的可停靠窗口的可停靠行为。默认值为 AnimatedDocks|AllowTabbedDocks。枚举 DockOption 如下表

| QMainWindow::DockOption 枚举         |      |                                                       |
|------------------------------------|------|-------------------------------------------------------|
| 标志: QMainWindow::DockOptions       |      |                                                       |
| 作用: 描述主窗口的可停靠窗口的可停靠行为(注: 只影响可停靠窗口) |      |                                                       |
| 成员                                 | 值    | 说明                                                    |
| QMainWindow::AnimateDocks          | 0x01 | 与 animated 属性相同。                                      |
| QMainWindow::AllowNestedDocks      | 0x02 | 与 dockNestingEnabled 属性相同                             |
| QMainWindow::AllowTabbedDocks      | 0x04 | 允许把一个可停靠部件放置于另一个可停靠部件之上(重叠), 并出一个选项卡标签, 用于选择使哪一个部件可见。 |
| QMainWindow::ForceTabbedDocks      | 0x08 | 强制使用选项卡式可停靠窗口, 也就是说, 可停靠窗口                            |

|                                           |                   |                                                                                                              |
|-------------------------------------------|-------------------|--------------------------------------------------------------------------------------------------------------|
|                                           |                   | 只可重叠，不可嵌套。若设置此选项，则 <code>AllowNestedDocks</code> 将不起作用。                                                      |
| <code>QMainWindow::VerticalTabs</code>    | <code>0x10</code> | 当可停靠窗口重叠时，选项卡标签垂直显示，若未设置此选项，选项卡标签都是显示在其底部。设置此选项，意味着同时设置了 <code>AllowTabbedDocks</code> 选项。                   |
| <code>QMainWindow::GroupedDragging</code> | <code>0x20</code> | 当拖动一个可停靠部件的标题栏时，将拖动重叠在一起的所有可停靠窗口，未设置此选项时，只会拖动当前的可停靠窗口。若 <code>QDockWidgets</code> 在某些区域中有限制，则可能不会正常的工作。Qt5.6 |

- 4、**documentMode**: bool      **访问函数**: `bool documentMode() const; void setDocumentMode(bool);`  
此属性描述选项卡是否以适合主窗口的模式呈现。另请参阅第 5 章，`QTabBar` 类。
- 5、**iconSize**: `QSize`      **访问函数**: `QSize iconSize() const; void setIconSize(const QSize&);`  
描述主窗口中工具栏图标的大小，图标只能缩小不能放大，默认是 GUI 样式的默认工具栏图标大小。
- 6、**tabShape**: `QTabWidget::TabShape`  
**访问函数**: `QTabWidget::TabShape tabShape() const; void setTabShape(QTabWidget::TabShape);`  
描述选项卡的形状，默认是 `QTabWidget::Rounded`，详见第 5 章 `QTabWidget::tabShape` 属性。
- 7、**toolButtonStyle**: `Qt::ToolButtonStyle`  
**访问函数**: `Qt::ToolButtonStyle toolButtonStyle() const; void setToolButtonStyle(Qt::ToolButtonStyle);`  
描述主窗口中的工具栏按钮的样式(即工具栏的图标和文本应如何显示)，默认为 `Qt::ToolButtonIconOnly`。详见前文 `QToolBar::toolButtonStyle` 属性。
- 8、**unifiedTitleAndToolBarOnMac**: bool      //qt5.2  
**访问函数**: `bool unifiedTitleAndToolBarOnMac() const; void setUnifiedTitleAndToolBarOnMac(bool);`  
描述在 macOS 上是否使用统一的标题和工具栏外观。Qt5 有一些限制，如下：
  - 不支持在带有 OpenGL 内容的窗口中使用。
  - 使用可停靠或可移动的工具栏时，可能会导致绘制错误，因此不建议使用。

## 二、QMainWindow 类中的函数

- 1、**QMainWindow**(`QWidget* parent = Q_NULLPTR, Qt::WindowFlags flags = Qt::WindowFlags();` //构造函数

### 2、与中心部件和状态栏有关的函数

- 1)、**void setCentralWidget**(`QWidget* widget;`  
把 `widget` 设置为中心部件，`QMainWindow` 获取 `widget` 的所有权
- 2)、**QWidget\* centralWidget**() const;  
返回主窗口的中心部件，若未设置中心部件，则返回 0。
- 3)、**QWidget\* takeCentralWidget**();      //qt5.2  
移除中心部件，被移除的部件的所有权被传递给调用方。
- 4)、**void setStatusbar**(`QStatusBar* statusbar;`  
把 `statusbar` 设置为主窗口的状态栏，若 `statusbar` 为 0，则会将状态栏从主窗口删除。  
`QMainWindow` 获取 `statusbar` 的所有权

5)、`QStatusBar* statusBar() const;`

返回主窗口的状态栏，若状态栏不存在，则该函数会创建并返回一个空的状态栏。

### 3、与工具栏有关的函数

6)、`void addToolBar(Qt::ToolBarArea area, QToolBar *toolbar);`

把工具栏 `toolbar` 添加到主窗口的区域 `area` 中。工具栏被放置在当前工具栏的末尾，若主窗口已有该工具栏，则只会把工具栏 `toolbar` 移动到区域 `area`，因此使用此函数可以以编程的方式移动工具栏。`Qt::ToolBarArea` 枚举详见 `QToolBar` 类。

7)、`void addToolBar(QToolBar *toolbar);`

相当于 `addToolBar(Qt::TopToolBarArea, toolbar);`

8)、`QToolBar *addToolBar(const QString &title);`

创建一个新工具栏，并把其添加到工具栏区域的顶部，注意：参数 `title` 只是工具栏的标题，也就是说该函数创建的工具栏没有任何内容，还需要使用 `QToolBar::addAction()` 等函数向新创建的工具栏中添加内容。

9)、`void insertToolBar(QToolBar* before, QToolBar* toolbar);`

把工具栏 `toolbar` 插入到工具栏 `before` 的前面。

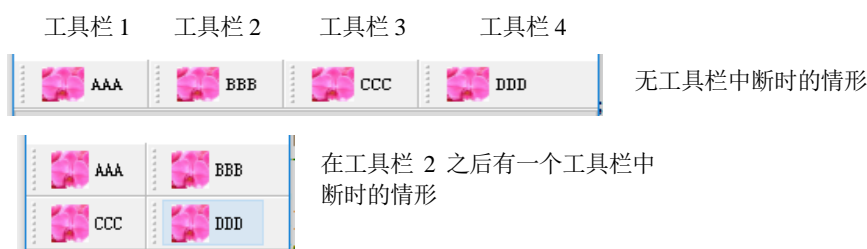
10)、`void removeToolBar(QToolBar *toolbar);`

从主窗口中移除而不删除(即隐藏)工具栏 `toolbar`。

11)、`Qt::ToolBarArea toolbarArea(QToolBar* toolbar) const;`

返回工具栏 `toolbar` 所在的区域，若 `toolbar` 未添加到主窗口，则返回 `Qt::NoToolBarArea`。

### 4、工具栏中断



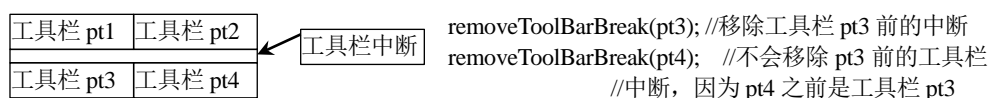
12)、`void addToolBarBreak(Qt::ToolBarArea area = Qt::TopToolBarArea);` //把工具栏中断添加到区域 `area`。

13)、`void insertToolBarBreak(QToolBar* before);` //把工具栏中断插入到工具栏 `before` 之前。

14)、`bool toolbarBreak(QToolBar* toolbar) const;` //返回 `toolbar` 之前是否是工具栏中断。

15)、`void removeToolBarBreak(QToolBar* before);`

移除在工具栏 `before` 之前插入的工具栏中断。见下图



### 5、与菜单和菜单栏有关的函数

16)、`void setMenuBar(QMenuBar* menuBar);`

```
void setMenuWidget(QWidget* menuBar);
```

以上函数都可以把主窗口的菜单栏设置为 `menuBar`，主窗口拥有 `menuBar` 的所有权。其中第 2 个函数可用于创建自定义的菜单栏。

17)、`QMenuBar* menuBar() const;`

返回主窗口的菜单栏，若菜单栏不存在，则返回一个空的菜单栏，返回空菜单栏意味着主窗口可以不使用 `QMainWindow::setMenuBar()` 函数添加的菜单栏，而使用这个空的菜单栏(通常使用的就是这个方法)，比如

```
QMainWindow *pw = new QManiWindow;
```

```
QMenu*pm1= pw->menuBar()->addMenu("AAA"); //向主窗口的菜单栏添加菜单 AAA
```

```
QMenu*pm2= pw->menuBar()->addMenu("BBB"); //向主窗口的菜单栏添加菜单 BBB
```

```
pm1.addAction("CCC");
```

18)、`QWidget* menuWidget() const;` //返回主窗口的菜单栏，若菜单栏还未创建，则返回 `null`。

## 6、自定义主窗口的上下文菜单(需要重新实现以下虚函数之一)

19)、`virtual void contextMenuEvent(QContextMenuEvent* event);` //虚拟的，受保护的

`QWidget::contextMenuEvent` 的重新实现

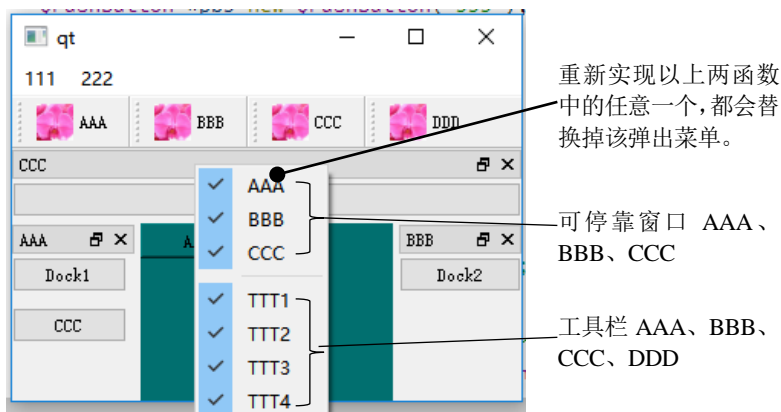
20)、`virtual QMenu* createPopupMenu();` //虚拟的

返回一个弹出菜单，其中包含在主窗口中出现的工具栏和可停靠窗口的可选中项目。当用户激活上下文菜单时(通常在工具栏或可停靠窗口标题栏上右击鼠标激活)，该函数会由主窗口调用，因此重新实现此函数，可创建自定义的弹出菜单，弹出菜单的所有权会转移给调用方。

以上两函数的区别如下：

- `createPopupMenu()` 函数只需返回一个 `QMenu`，然后在主窗口的工具栏和可停靠窗口的标题栏上右击时，则会弹出该菜单，但在主窗口的其他地方(比如中心部件处)右击，则不会弹出该菜单。
- `contextMenuEvent()` 是一个事件处理函数，该函数先于 `createPopupMenu()` 函数而被调用，若重新实现了 `contextMenuEvent()` 函数，则 `createPopupMenu()` 函数就不起作用了。
- 若重新实现 `contextMenuEvent()` 函数来弹出一个菜单，则可在主窗口的任意位置(标题栏除外)右击弹出菜单。
- 重新实现以上两个函数中的任意一个，都会代替主窗口中默认的在工具栏和可停靠窗口上右击时弹出的菜单。

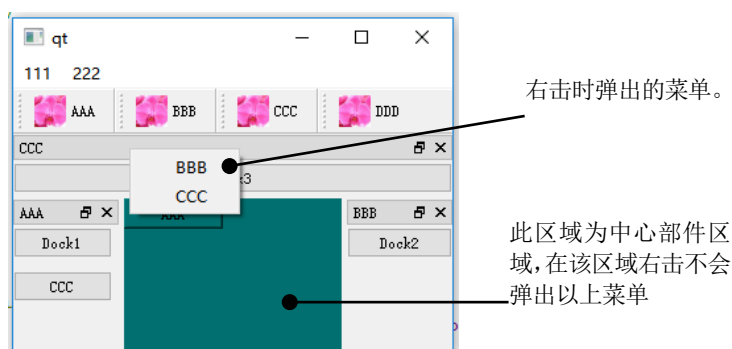
示例：假设创建的主窗口界面如下图



以下为重新实现的 createPopupMenu()函数

```
QMenu * QMainWindow::createPopupMenu() {
 QMenu *pml=new QMenu("AAA"); pml->addAction("BBB"); pml->addAction("CCC");
 return pml; }
```

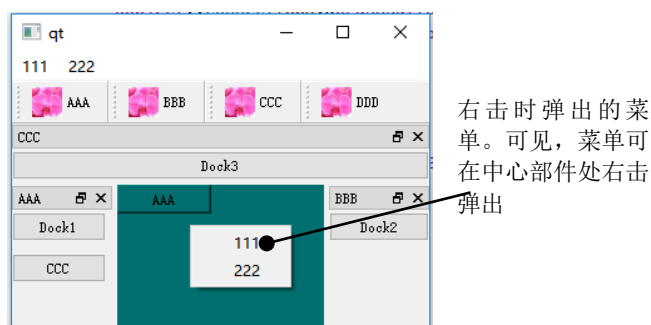
效果如下图:



以下为重新实现的 contextMenuEvent()函数

```
void QMainWindow::contextMenuEvent(QContextMenuEvent* event) {
 QMenu *pml=new QMenu("AAA"); pml->addAction("111"); pml->addAction("222");
 pml->exec(QCursor::pos()); } //菜单需要明确显示。
```

效果如下图:



## 7、与可停靠窗口有关的函数

21)、`void addDockWidget(Qt::DockWidgetArea area, QDockWidget* dockwidget);`

`void addDockWidget(Qt::DockWidgetArea area, QDockWidget* dockwidget, Qt::Orientation orientation);`

把可停靠窗口 `dockwidget` 按指定的方向 `orientation` 添加到指定的主窗口区域 `area` 中。

22)、`void removeDockWidget(QDockWidget* dockwidget);`

从主窗口中移除而不删除(即隐藏)可停靠窗口 `dockwidget`。

23)、`Qt::DockWidgetArea dockWidgetArea(Qt::DockWidget* dockwidget) const;`

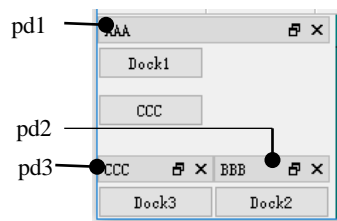
返回可停靠窗口 `dockwidget` 所在的区域, 若 `dockwidget` 未添加到主窗口, 则返回 `Qt::NoDockWidgetArea`。

24)、`void splitDockWidget(QDockWidget* first, QDockWidget* second, Qt::Orientation orientation);`

该函数用于拆分可停靠窗口在主窗口中的放置区域, 说简单点, 就是该函数可实现以编程的方式布局嵌套的可停靠窗口。规则如下:

- 若 `orientation` 为 `Qt::Horizontal`(水平方向), 则把 `second` 放置于 `first` 的右侧。
- 若 `orientation` 为 `Qt::Vertical`(垂直方向), 则把 `second` 放置于 `first` 的下面。
- 注意: 该函数还受 `Qt::LayoutDirection` 的影响, 若布局是从右到左的方向时, 以上规则会反向。
- 示例: 假设 `pd1`、`pd2` 和 `pd3` 是可停靠窗口(其中的内容、标题等需自行设计, 此处略), 设有如下语句, 则效果如右图所示

```
mainwindow->splitDockWidget(pd1, pd3, Qt::Vertical);
mainwindow->splitDockWidget(pd3, pd2, Qt::Horizontal);
```



25)、`Qt::DockWidgetArea corner(Qt::Corner corner) const;`

返回指定角落 `corner` 的可停靠部件区域。

26)、`void setCorner(Qt::Corner corner, Qt::DockWidgetArea area);`

把指定的可停靠区域 `area` 设置为占据指定的角落 `corner`。该函数用于设置水平和垂直区域相交处的角落, 应由哪个区域占据。原理见下图(以左上角为例)



27)、`void setTabPosition(Qt::DockWidgetAreas areas, QTabWidget::TabPosition tabPosition);`

`QTabWidget::TabPosition tabPosition(Qt::DockWidgetArea area) const;`

以上函数用于设置和获取指定的可停靠区域 `area` 的选项卡的位置(即上、下、左、右), 默认位于底部。 `setDockOptions(QMainWindow::VerticalTabs);` 会覆盖此函数的设置。

28)、`void resizeDocks(const QList<QDockWidget*> &docks, const QList<int> &sizes,`

`Qt::Orientation orientation);` //qt5.6

把列表 `docks` 中的可停靠窗口的大小, 调整为列表 `sizes` 中的相应大小(以像素为单位)。若 `orientation` 为 `Qt::Horizontal`(水平方向), 则调整宽度, 若 `orientation` 为 `Qt::Vertical`(垂

直方向), 则调整高度。主窗口的大小不会受到此函数的影响。

29)、`QList<QDockWidget*> tabifiedDockWidgets(QDockWidget* dockwidget) const;`

返回与 dockwidget 一起被选项卡化的可停靠窗口。原理见下图



30)、`void tabifyDockWidget(QDockWidget* first, QDockWidget* second);`

把 second 移至 first 处, 并使这两个可停靠窗口被选项卡化。也就是说使用此函数, 可以实现以编程的方式, 把两个可停靠窗口以选项卡的形式重叠在一起。原理见下图



## 8、存储和恢复可停靠窗口的状态

31)、`QByteArray saveState(int version = 0) const;`

32)、`bool restoreState(const QByteArray &state, int version = 0);`

以上函数用于保存和恢复工具栏和可停靠窗口的状态, 包括 `setCorner()` 设置的角落, 若状态已恢复, 则 `restoreState()` 返回 `true`。`version`(版本号)会在 `saveState()` 和 `restoreState()` 之间传递和比较。注意: 对象名(即 `objectName` 属性)会用于识别工具栏和可停靠窗口, 因此应确保为每个工具栏和可停靠窗口设置唯一的对象名。

示例代码如下:

```
QByteArray q = saveState(1); //保存当前状态, 版本为 1
restoreState(q,1); //恢复到状态 q, 版本为 1 的状态。
```

33)、`bool restoreDockWidget(QDockWidget* dockwidget);`

若 dockwidget 是在调用 `restoreState()` 函数之后创建的, 则恢复其状态。若状态已经恢复, 则返回 `true`, 否则返回 `false`。

## 三、QMainWindow 类中的信号

1、`void tabifiedDockWidgetActivated(QDockWidget* dockWidget);` //信号, qt5.8

当通过选择选项卡, 激活被选项卡化的可停靠窗口时, 发送此信号, dockWidget 为激活的可停靠窗口。

2、`void toolButtonStyleChanged(Qt::ToolButtonStyle toolButtonStyle);` //信号

当工具按钮的样式(即工具栏的图标和文本应如何显示)变化时, 发送此信号,



toolButtonStyle 为变化后的新样式。

3、void `iconSizeChanged`(const QSize &iconSize); //信号

当主窗口中使用的图标大小发生变化时，即 QMainWindow::iconSize 属性变化时，发送此信号。

作者：黄邦勇帅(原名：黄勇)

**2018-5-16**

本文作者：黄邦勇帅(原名：黄勇)

本文要求读者已经非常熟悉 C++ 语法。若读者不熟悉 C++ 语法，推荐参阅《C++ 语法详解》(作者：黄勇)一书，电子工业出版社出版。

本文主要讲解了 Qt 模型/视图框架，本文对模型/视图框架作了全面细致的分析，内容全面、详细且深入。本文内容由浅入深，易学易懂。

本文使用的是 windows 10 操作系统，Qt 版本为 Qt5.10.1，Qt Creator 的版本为 Qt Creator 4.5.1  
本文内容完全属于个人见解与参考文献的作者无关，限于水平有限，其中难免有误解之处，望指出更正。

声明：禁止抄袭、复印、转载本文，本文作者拥有完全版权。

主要参考文献：

- 1、C++语法详解 黄勇 编著 电子工业出版社 2017 年 7 月
- 2、Qt Creator 快速入门(第 3 版) 霍亚飞 编著 北京航空航天大学出版社 2017 年 1 月
- 3、C++ GUI Qt4 编程(第 2 版) [加拿大] Jasmin Blanchette [英] Mark Summerfield 著 闫锋欣 曾泉人 张志强 译 电子工业出版社 2008 年 8 月
- 4、Qt 5 开发及实例 陆文周 主编 电子工业出版社 2014 年 1 月
- 5、Qt 高级编程 [英] Mark Summerfield 著 白建平 王军锋 闫锋钦 白净 译 电子工业出版社 20011 年 4 月

## 第8章 Qt 模型、视图框架目录

### 第1篇 自定义模型/视图框架

- [8.1 模型、视图原理](#)
- [8.2 模型: QAbstractItemModel 类](#)
- [8.3 视图: QAbstractItemView 类 \(视图基类\)](#)
- [8.4 选择: QItemSelectionModel 类与 QItemSelection 类](#)
- [8.5 委托: QAbstractItemDelegate 与 QStyleOptionViewItem](#)
- [8.6 索引: QModelIndex 类](#)
- [8.7 自定义视图示例](#)

### 第2篇 Qt 实现的标准模型/视图框架相关类

- [8.8 标准模型: QStandardItemModel 类及 QStandardItem 类](#)
- [8.9 列表模型: QAbstractListModel 类、QAbstractTableModel 类、QStringListModel 类](#)
- [8.10 文件系统模型: QFileSystemModel 类](#)
- [8.11 表格视图: QTableView 类](#)
- [8.12 列表视图: QListView 类](#)
- [8.13 树视图: QTreeView 类](#)
- [8.14 标头视图: QHeaderView 类](#)
- [8.15 列视图: QColumnView 类](#)
- [8.16 项目委托: QStyleItemDelegate 类](#)

### 第3篇 使用现成的模型/视图部件

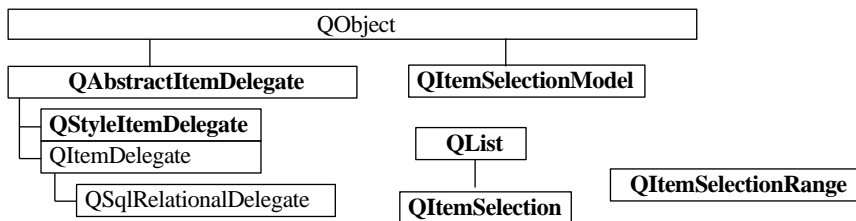
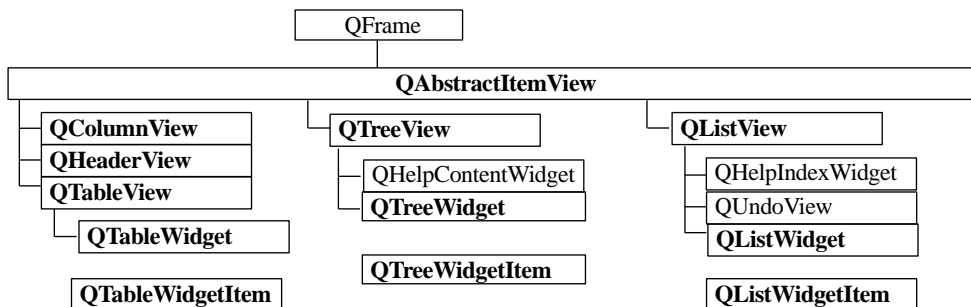
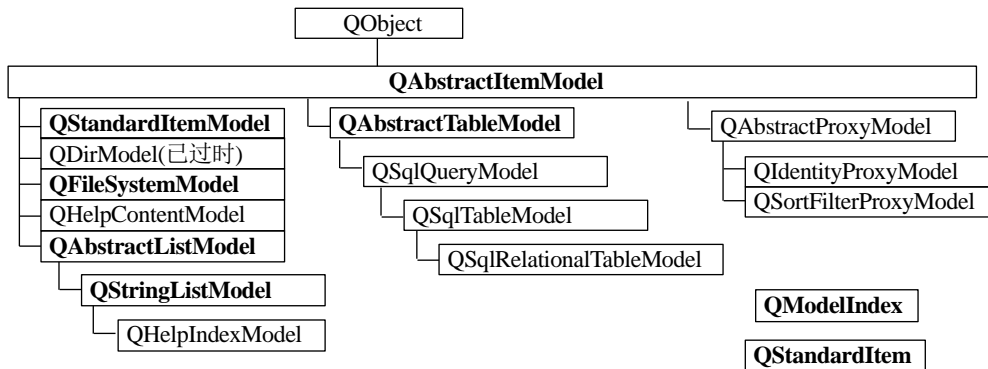
- [8.17 表格部件: QTableWidget 类](#)
- [8.18 列表部件: QListWidget 类](#)
- [8.19 树部件: QTreeWidget 类](#)

## 第 8 部分 Qt 模型/视图框架

注意：本章节程序都假设读者在 pro 文件中已添加了正确的 `QT+=widgets` 语句，文中不再重复累述添加此语句。

本文注重讲解原理，因此使用的是手写的 Qt 程序，。

本章讲解的类及继承关系如下图所示(注：限于篇幅本章只讲解图中粗体所示的类)



# 第 1 篇 自定义模型/视图框架

## 8.1 模型/视图原理

数据通常由若干个数据项(item)组成。

### 一、模型/视图基本原理

1、MVC 把图形界面分为三个部分：模型(Model)、视图(View)、控制器(Controller)。

- 模型：用于管理数据，注意，数据不一定需要位于模型之中
- 视图：就是呈现在用户面前的界面外观，视图负责把模型中的数据显示给用户。
- 控制器：用于处理用户在用户界面的输入。

2、MVC 把需要处理的数据及其显示分离开来。

3、Qt 实现的 MVC 模型(见图)：

①、Qt 把视图和控制器组合在一起，从而形成模型/视图结构。

②、模型直接与数据进行通信，并为视图和委托提供访问数据的接口。

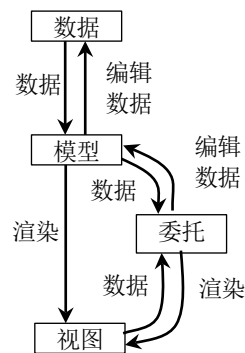
③、数据与模型：数据不一定需要存储在模型中，数据也可以存储在文件、数据库等其他地方，存储的数据不一定拥有一种数据结构，但是模型通常会把这些数据组织成一种数据结构(比如列表(list)结构、树形(tree 结构等)，这种结构只是逻辑上的结构，数据本身不一定拥有这种结构，然后视图根据模型提供的逻辑结构显示数据，比如列表视图(QListView)可以显示把数据组织成列表结构的模型(QStandardItemModel 模型实现了该种结构)中的数据；

当然，也可以使用列表视图显示组织为树形结构的数据，但这样做需要做一些比较麻烦的处理才能正确显示，所以通常应使用列表视图显示列表模型的数据，树形视图显示树形结构模型的数据。

④、为了对用户的输入进行灵活的处理，Qt 引入了委托 Delegate(或代理)的概念，委托其实就是把用户输入的数据委托给 Qt 的某个部件处理，比如委托 QSpinBox 部件来处理用户输入的整数等。另外，委托还会绘制(或渲染)视图中的个别数据项。

⑤、模型、视图、委托之间的通信使用 Qt 的信号和槽机制来完成。

⑥、下面以实际的图形界面为例，讲解 MVC 各部分间的功能



Qt 模型/视图理论结构

模型: 表格中的数据都是由模型管理的

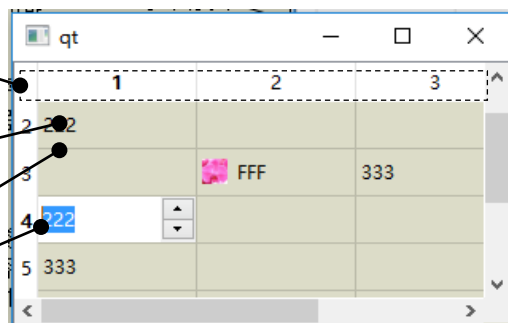
视图: 负责管理水平/垂直标头、滚动、对数据项的选择、网格线等

委托: 负责单元格(数据项)的绘制, 以及编辑单元格的内容, 比如此单元格就是委托 QSpinBox 部件编辑该数据项

水平标头

数据项

单元格边框线  
(网格线)



实际的 Qt 界面程序

#### 4、Qt 对模型/视图结构的具体实现

①、模型: Qt 使用抽象类 `QAbstractItemModel` 来描述模型, 所有的模型都是通过子类化该抽象类而实现的。Qt 实现了一些标准的现成模型, 下面是一简介:

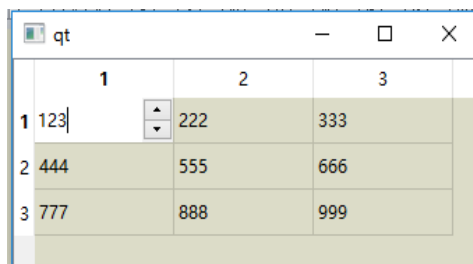
- `QStringListModel`: 用于存储 `QString` 项目的列表。
- `QStandardItemModel`: 该模型可以被当作列表模型、表格模型、树模型来使用。
- `QFileSystemModel`: 该模型提供本地文件系统中的文件和目录信息, 模型本身没有任何的数据项目。
- `QSqlQueryModel`、`QSqlTableModel`、`QSqlRelationalTableModel` 与数据库有关。

②、视图: Qt 使用抽象类 `QAbstractItemView` 来描述视图, 所有的视图都是通过子类化该抽象类而实现的。Qt 实现了一些标准的现成视图, 比如 `QListView`(列表视图), `QTableView`(表格视图), `QTreeView`(树视图)等。

③、委托: Qt 使用抽象类 `QAbstractItemDelegate` 来描述委托, Qt 实现了两个委托类, `QStyledItemDelegate` 和 `QItemDelegate`, 这两个委托之中只能使用其中一个, 其区别在于 `QItemDelegate` 总是使用一种默认的样式绘制数据项, 而 `QStyledItemDelegate` 使用当前的样式来绘制数据项, 通常使用的是 `QStyledItemDelegate`。Qt 默认使用 `QStyledItemDelegate`。

#### 5、使用 Qt 模型/视图的步骤。代码如下(界面见下图):

```
QStandardItemModel model(3, 3, this); //创建一个 3 行 3 列的表格结构的模型
QTableView v1; //创建一个表格视图
//设置模型的数据, 使用索引的形式设置每个数据项的值
model.setData(model.index(0, 0), 123);
model.setData(model.index(0, 1), 222);
model.setData(model.index(0, 2), 333);
model.setData(model.index(1, 0), 444);
model.setData(model.index(1, 1), 555);
model.setData(model.index(1, 2), 666);
model.setData(model.index(2, 0), 777);
model.setData(model.index(2, 1), 888);
model.setData(model.index(2, 2), 999);
v1.setModel(&model); //设置视图 v1 的模型
v1.show(); //显示视图
```



## 二、定位模型中的数据与模型索引

### 1、模型的结构

在 Qt 中，无论数据被存储为何种数据结构，模型总是以层次结构(即树形结构)来表示数据，视图按照此约定来访问模型中的数据，若数据是列表(list)或表格(tab)结构的数据，则可以把其看作是只含有顶层节点，不含任何子节点的树形结构。也就是说，我们在子类化 `QAbstractItemModel` 来自定义自己的模型结构时，始终应以树形结构为出发点，来组织自己的模型结构。

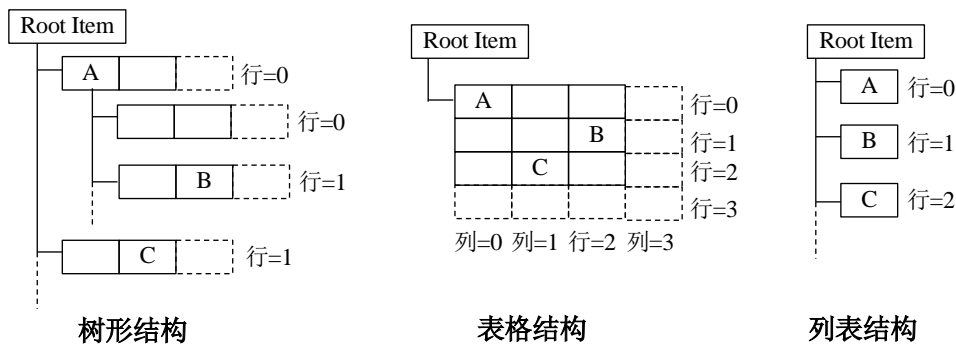
### 2、模型索引(简称索引):

- Qt 使用模型索引来使数据的表示和访问相分离，视图和委托使用索引来访问模型中的数据项，因此，只有模型知道怎样获取数据。
- 模型索引不像普通索引仅使用一个数字就能描述，模型索引需要使用 3 个属性进行描述：行号、列号、父模型索引。虽然模型索引使用行号、列号来定位数据项，但这并不意味着，数据是存储在数组或表格结构中的，使用行号、列号只是允许模型与视图、代理进行通信的一种约定。之所以需要父模型索引，是因为 Qt 的模型都是以层次结构(树形结构)组织的，Qt 具体实现时，顶级数据项的父模型索引，使用“无效模型索引”来表示。
- 模型索引包含一个指向创建它们的模型的指针，当使用多个模型时，可避免混淆。

### 3、Qt 对模型索引的实现

- Qt 使用 `QModelIndex` 类实现模型索引，该类提供的索引是一个临时索引，因为模型可能会随时重新组织其内部结构(即数据项可能被随时发生变化，比如删除、增加新数据项等)，因此模型索引可能会变得无效，所以模型索引不需要也不应被存储，若需要对数据项进行长时间的引用，应使用 `QPersistentModelIndex` 类创建一个持久模型索引。
- 使用模型索引引用模型中数据项的方法是使用 `QAbstractItemModel::index()` 函数。
- 无效模型索引：使用零参数的 `QModelIndex` 类的构造函数创建，即 `QModelIndex()` 就表示创建了一个无效模型索引。

4、下面以代码的形式来说明模型索引，下图为 Qt 常见的模型结构。



#### ①、树形结构:

//获取数据项 A 的模型索引，A 的父索引由一个无效模型索引指定。

```
QModelIndex iA = model.index(0,0, QModelIndex());
//获取数据项 B 的模型索引, B 位于父索引 iA 的第 1 行第 1 列位置。
QModelIndex iB = model.index(1,1, iA);
```

## ②、表格结构

```
//获取数据项 B 的模型索引, B 是顶级数据项, 因此其父索引是一个无效索引。
QModelIndex iB = model.index(1,2, QModelIndex());
```

## ③、列表结构

```
//获取数据项 B 的模型索引, B 是顶级数据项, 因此其父索引是一个无效索引。
QModelIndex iB = model.index(1,0, QModelIndex());
```

# 三、数据的角色与数据的类型

1、同一个类型的数据可以作为不同的角色(或作用)来使用, 比如对于字符串"AAA", 可把该字符串以文本的形式显示在视图的相应位置上, 也可把该字符串作为工具提示使用, 还可把该字符串作为 what's this 的帮助提示等。由此可见, 数据的角色, 决定了该数据在视图中的显示方式, 角色不同, 显示方式也不同。

2、数据项与数据元素: 位于同一个位置的数据项, 并不仅仅只有一个数据元素, 本文把组成数据项的数据称为数据元素, 每个数据元素都有其自身的角色, 比如一个数据项可能会同时含有图标数

|   | 1   | 2   |
|---|-----|-----|
| 1 | 123 | 222 |
| 2 | 444 | 555 |

据元素(角色为 Qt::DecorationRole)、文本数据元素(角色为 Qt::DisplayRole)、工具提示数据元素(角色为 Qt::ToolTipRole)等(见图第 1 行第 2 列的数据项"222")。

3、因为数据项由多个数据元素组成, 因此把数据项使用一个单独的类来管理是比较方便的, 比如对于 QStandardItemModel 模型的数据项, 就使用了类 QStandardItem 来专门管理其数据项。

4、为了避免概念上的混乱, 下面对数据、数据项、项(项目)、节点、单元格、数据元素、模型索引(索引)做一简介

①、数据: 是一个统称, 既可以是数据项也可以是数据元素。

②、数据项: 是由多个数据元素组成的, 每个数据元素都有自己的角色。

③、单元格、节点、项目: 这三个概念通常用于指模型中的某一个数据项所在的位置, 只是对于不同的模型结构会有不同的称呼, 比如树形结构通常称为节点, 表格结构通常称为单元格, 而项目是一种更通用的称呼, Qt 中通常称其为项目(item)或数据项; 有时也会对以上概念加以区别对待, 比如单元格 XXX 的数据项, 此时单元格表示位置, 数据项就表示实际存储的数据。不管怎样, 模型中某个数据项所在位置都是需要使用模型索引来指定的, 因此单元格、节点、项目、数据项、模型索引这几个概念通常会根据不同的情形用于表示模型中的某个位置。

5、标准的数据角色由枚举 Qt::ItemDataRole 来描述, 见下表。数据角色由 QAbstractItemModel::setData()函数的第 3 个参数指定, 其使用方法如下, 其效果见第 2 点图示中第 1 行第 2 列的单元格:

```
model.setData(model.index(0,1), 222, Qt::DisplayRole); //设置显示的文本
model.setData(model.index(0,1), QIcon("F:/li.png"), Qt::DecorationRole); //设置图标
```



```
model.setData(model.index(0, 1), "EEE", Qt::ToolTipRole); //设置工具提示
```

**Qt::ItemDataRole 枚举(无标志)**

作用：描述数据元素的角色

| 成员                       | 值  | 说明                                   |
|--------------------------|----|--------------------------------------|
| 通用角色                     |    |                                      |
| Qt::DisplayRole          | 0  | 数据以文本形式显示(QString 类型)                |
| Qt::DecorationRole       | 1  | 数据以图标形式显示(QColor、QIcon、QPixmap)      |
| Qt::EditRole             | 2  | 数据可以编辑(QString)                      |
| Qt::ToolTipRole          | 3  | 数据作为工具提示(QString)                    |
| Qt::StatusTipRole        | 4  | 数据作为状态提示(QString)                    |
| Qt::WhatsThisRole        | 5  | 数据作为 What's this 帮助文档(QString)       |
| Qt::SizeHintRole         | 13 | 数据项的大小提示(QSize 类型)                   |
| 描述外观的角色                  |    |                                      |
| Qt::FontRole             | 6  | 使用默认委托时渲染的数据项的字体(QFont)              |
| Qt::TextAlignmentRole    | 7  | 使用默认委托时渲染的数据项的文本对齐方式(Qt::Alignment)  |
| Qt::BackgroundRole       | 8  | 使用默认委托时渲染的数据项的背景画刷(QBrush)           |
| Qt::BackgroundColorRole  | 8  | 已过时，改为 BackgroundRole                |
| Qt::ForegroundRole       | 9  | 使用默认委托时渲染的数据项的前景画刷(通常为文字颜色)(QBrush)  |
| Qt::TextColorRole        | 9  | 已过时，改为 ForegroundColorRole           |
| Qt::CheckStateRole       | 10 | 数据项的选中状态(Qt::CheckState 枚举)          |
| Qt::InitialSortOrderRole | 14 | 标题视图部分的初始排序顺序(即升序、降序)(Qt::SortOrder) |

**四、选择视图中的数据项(选择模型)**

1、选择模型：对视图内项目的选择 Qt 使用 QItemSelectionModel 类来实现。所有的标准视图都有自己默认的选择模型。视图可以使用 QAbstractItemView::selectionModel() 函数和 QAbstractItemView::setSelectionModel() 来获取和设置选择模型。通常不需要对选择模型进行设置。

**五、Qt 实现的便利类**

Qt 通过模型/视图框架结构实现了一些标准的方便使用的部件，使用这些便利部件比较简单，这些类是 QListWidget、QTreeWidget、QTableWidget。

## 8.2 QAbstractItemModel 类(模型基类)

- 1、QAbstractItemModel 类继承自 QObject，该类是 Qt 所有模型类的基类，用于管理模型/视图结构中的数据。Qt 的所有模型都需要子类化该类。注意：该类是抽象类，因此不能创建该类的对象。
- 2、该类的构造函数，原型为：**QAbstractItemModel**(QObject\* parent = Q\_NULLPTR); //构造函数

### 一、QAbstractItemModel 类中的纯虚函数及有效模型索引

- 1、有效模型索引的创建：模型索引是由 QModelIndex 类进行描述的，但该类只有一个默认构造函数，而使用默认构造函数创建的模型索引是无效模型索引，因此要创建一个有效的模型索引，需要使用工厂函数 QAbstractItemModel::createIndex()来创建，在重新实现纯虚函数 index()和 parent()时，都有可能调用该工厂函数来创建模型索引。

- 2、下面为相关的函数及其原型

- 1)、virtual QModelIndex **index**(int row, int column,

const QModelIndex &parent = QModelIndex ()) const = 0; //纯虚函数

返回由行 row、列 column、父索引 parent 指定的数据项的模型索引，当子类重新实现此函数时，模型索引需调用 createIndex()函数来创建。

- 2)、virtual QModelIndex **parent**(const QModelIndex &index) const = 0; //纯虚函数

返回索引 index 的父模型索引，若没有父模型索引，则返回无效的模型索引。重新实现该函数时需要小心调用 QModelIndex 中的成员函数(比如 QModelIndex::parent());因为自定义的模型索引只会调用自定义的实现，因此 QModelIndex::parent()会调用此处重新实现的该函数，从而导致无限递归。重新实现该函数时，通常也使用 createIndex()函数创建模型索引。

- 3)、virtual int **rowCount**(const QModelIndex &parent = QModelIndex ()) const = 0; //纯虚函数

virtual int **columnCount**(const QModelIndex &parent = QModelIndex ()) const = 0; //纯虚函数

以上函数表示，返回父模型索引 parent 下的行/列数，在实现基于表格的模型时，若父模型索引有效，则以上函数都应返回 0。

- 4)、virtual QVariant **data**(const QModelIndex &index, int role = Qt::DisplayRole) const = 0; //纯虚函数

返回索引 index 所引用的项在给定角色 role 下存储的数据。注意：若没有需要返回的值，应返回无效的 QVariant，而不是返回 0。该函数用于向视图和委托提供项目数据，也就是说视图和委托是显示的该函数返回的值。

- 5)、virtual bool **setData**(const QModelIndex &index, const QVariant &value, int role=Qt::EditRole); //虚拟的  
设置索引 index 所引用数据项的值为 value，其角色为 role。若设置成功则返回 true，并且应发送 dataChanged()信号，否则返回 false。默认实现为返回 false，虽然此函数不是纯虚函数，但若模型是可编辑模型，则必须重新实现此函数，

- 6)、QModelIndex **createIndex**(int row, int column, void \*ptr = Q\_NULLPTR); //受保护的

QModelIndex **createIndex**(int row, int column, quintptr id); //受保护的

- 以上函数表示使用内部指针 ptr 或内部 ID 为给定的行 row 和列 column 创建一个模型索引。
- 当重新实现纯虚函数 index()时，需要调用该函数创建模型索引。
- 内部指针可由 QModelIndex::internalPointer()函数获取，内部 ID 可由 QModelIndex::internalId()函数获取。内部指针其实质就是指指向模型实际所管理的数据，因此不一定需要使用 internalPointer()函数来获取。
- quintptr 类型在 32 位指针的系统上是 quint32，在 64 位指针的系统上是 quint64，其最终结果都是使用 typedef 重命的 unsigned int 类型(只是系统不同长度不同)

7)、bool **hasIndex**(int row, int column, const QModelIndex &parent =QModelIndex ()) const;

若根据 row、column、parent 返回的模型索引是有效索引，则返回 true，否则返回 false。

8)、virtual bool **hasChildren**(const QModelIndex &parent = QModelIndex ())const; //虚拟的

若 parent 拥有任何子女，则返回 true，否则返回 false，注意，若设置标志为 Qt::ItemNeverHasChildren，则使用此方法的行为是未定义的。在分层模型中，查找数据项的子项目数量是一项昂贵的操作，因此 rowCount()函数应在确有必要时进行调用，通过首先调用此函数判断数据项是否有子项，然后再决定是否调用 rowCount()函数是一种有效的方法。

## 二、插入和删除行/列

1、要使模型能插入行/列和删除行/列，子类需要重新实现以下虚函数：insertRows(); insertColumns(); removeRows(); removeColumns();

2、下面以 insertRows()虚函数为例，讲解其规则(其余函数，原理类同)

在将新行插入到任何基础数据结构之前，必须调用 beginInsertRows()函数(称其为 begin 函数)，该函数会通知其他组件(比如视图或委托)行数将要发生变化，完成插入操作之后，还需要调用 endInsertRows()函数(称其为 end 函数)以通知其他组件，该模型的行数已经更改，若 insertRows()插入成功，则返回 true，否则返回 false。

3、更改模型结构的另一种方法

通常使用 begin 和 end 函数就能够达到通知其他组件模型结构变化的目的，但对于结构比较复杂的模型，则这种方法可能会比较低效，比如若有一个有 300 百万行的模型，需要删除所有偶数行，这将有可能使用 beginRemoveRows 和 endRemoveRows 达到 150 万次之多，这显然是低效的。此时可使用以下步骤来更新模型结构

- 发送 layoutAboutToBeChanged()信号
- 更新模型结构的内部数据。
- 使用 chnagePersistentIndexList()更新持久索引。
- 发送 layoutChanged()信号。

以上步骤可用于更新任何结构的模型。

4、下面为相关的函数及其原型

### 插入

9)、virtual bool **insertColumns**(int column, int count, const QModelIndex &parent=QModelIndex ()) //虚拟的

virtual bool **insertRows**(int row, int count, const QModelIndex &parent = QModelIndex ()); //虚拟的

- QAbstractItemModel 类对以上虚函数的默认实现什么也没有做，并返回 false，因此要想模型支持插入操作，需要重新实现以上虚函数。重新实现时需要调用相应的 begin 函数和 end 函数。
- 以上函数表示在指定的列 column/行 row 之前插入 count 行/列，插入的新行/列将是 parent 模型索引所指数据项的子项，若插入成功则返回 true，否则返回 false。

10)、bool **insertRow**(int row, const QModelIndex &parent = QModelIndex ());

bool **insertColumn**(int column, const QModelIndex &parent = QModelIndex ());

以上函数分别调用虚函数 insertRows()和 insertColumns()

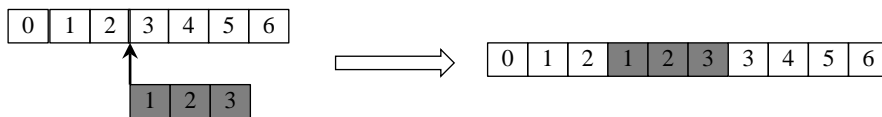
11)、void **beginInsertColumns**(const QModelIndex &parent, int first, int last); //受保护的

void **beginInsertRows**(const QModelIndex &parent, int first, int last); //受保护的

void **endInsertColumns**(); //受保护的

void **endInsertRows**(); //受保护的

- 以上函数分别是插入列和行时的 begin 和 end 函数，其中在数据被插入之前 beginInsertColumns()函数会发送 columnsAboutToBeInserted()信号，beginInsertRows()函数会发送 rowsAboutToBeInserted()信号，视图或代理通过连接到以上信号，以更新其视图显示，若视图或代理未对以上信号做出处理，则不会正确显示插入的行(需验证信号问题)。
- 参数 parent 表示被插入到新列/行的父索引，first 和 last 分别表示新列插入后的开始和结束列/行号，下面以 beginInsertColumns()函数为例进行讲解，beginInsertRows()函数原理是相同的(原理见下图)。



假设要在第 3 列之前插入 3 列，其语句为：beginInsertColumns(parent, 3,5);

## 删除

12)、virtual bool **removeColumns**(int column,int count,const QModelIndex &parent=QModelIndex ()); //虚拟  
virtual bool **removeRows**(int row, int count, const QModelIndex &parent = QModelIndex ()); //虚拟的

- QAbstractItemModel 类对以上虚函数的默认实现什么也没有做，并返回 false，因此要想模型支持删除操作，需要重新实现以上虚函数。重新实现时需要调用相应的 begin 函数和 end 函数。
- 以上函数表示删除模型索引 parent 所指数据项之下从列 column/行 row 开始的 count 行/列。若删除成功则返回 true，否则返回 false。

13)、bool **removeRow**(int row, const QModelIndex &parent = QModelIndex ());

bool **removeColumn**(int column, const QModelIndex &parent = QModelIndex ());

以上函数分别调用虚函数 removeRows()和 removeColumns()

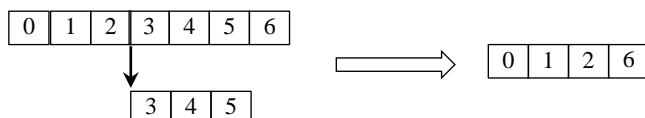
14)、void **beginRemoveColumns**(const QModelIndex &parent, int first, int last); //受保护的

void **beginRemoveRows**(const QModelIndex &parent, int first, int last); //受保护的

void **endRemoveColumns**(); //受保护的

void **endRemoveRows**(); //受保护的

- 以上函数分别是删除列和行时的 `begin` 和 `end` 函数，其中在数据被删除之前 `beginRemoveColumns()` 函数会发送 `columnsAboutToBeRemoved()` 信号，`beginRemoveRows()` 函数会发送 `rowsAboutToBeRemoved()` 信号，视图或代理通过连接到以上信号，以更新其视图显示，若视图或代理未对以上信号做出处理，则不会正确显示删除的行。
- 参数 `parent` 表示被删除的列/行的父索引，`first` 和 `last` 分别表示被删除的数据项的开始和结束列/行号，下面以 `beginRemovedCloumns()` 函数为例进行讲解，`beginRemovedRows()` 函数原理是相同的(原理见下图)。



假设要删除第 3~5 列的三列，其语句为：`beginRemovedColumns(parent, 3,5);`

## 移动

15)、virtual bool `moveColumns`(const QModelIndex &*sourceParent*, int *sourceColumn*, int *count*,  
const QModelIndex &*destinationParent*, int *destinationChild*); //虚拟的  
virtual bool `moveRows`(const QModelIndex &*sourceParent*, int *sourceRow*, int *count*,  
const QModelIndex &*destinationParent*, int *destinationChild*); //虚拟的

- `QAbstractItemModel` 类对以上虚函数的默认实现什么也没有做，并返回 `false`，因此要想模型支持移动操作，需要重新实现以上虚函数。重新实现时需要调用相应的 `begin` 函数和 `end` 函数。
- 以上函数表示从源模型索引 `sourceParent` 所指数据项之下，从列 `column`/行 `row` 开始的 `count` 行/列，移至目标模型索引 `destinationParent` 所指数据项之下的 `destinationChild` 之前。若移动成功则返回 `true`，否则返回 `false`。

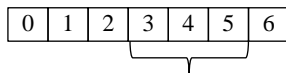
16)、bool `moveRow`(const QModelIndex &*sourceParent*, int *sourceRow*,  
const QModelIndex &*destinationParent*, int *destinationChild*);  
bool `moveColumn`(const QModelIndex &*sourceParent*, int *sourceColumn*,  
const QModelIndex &*destinationParent*, int *destinationChild*);

以上函数分别调用虚函数 `moveRows()` 和 `moveColumns()`

17)、bool `beginMoveColumns`(const QModelIndex &*sourceParent*, int *sourceFirst*, int *sourceLast*,  
const QModelIndex &*destinationParent*, int *destinationChild*); //受保护的  
bool `beginMoveRows`(const QModelIndex &*sourceParent*, int *sourceFirst*, int *sourceLast*,  
const QModelIndex &*destinationParent*, int *destinationChild*); //受保护的  
void `endMoveColumns`(); //受保护的  
void `endMoveRows`(); //受保护的

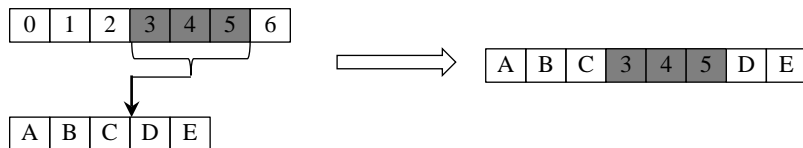
- 以上函数分别是移动列和行时的 `begin` 和 `end` 函数，`begin` 函数简化了模型中实体的移动，负责移动模型中的持久索引，否则需要自己完成持久索引的移动。使用以上的 `begin` 和 `end` 函数是 `changePersistentIndex()` 函数及发送 `layoutChanged()` 和 `layoutAboutToBeChanged()` 信号的替代方法(验证)。

- 参数 `sourceParent` 表示被移动的列/行的父索引, `destinationPart` 表示移动后所在新列/行的父索引, 以上函数表示把从 `sourceFirst` 开始到 `sourceLast` 结束的数据项移至 `destinationChild` 表示的列/行之前。
- 当在同一父索引中移动时, 必须确保 `destinationChild` 位于  $(sourceFirst, sourceLast+1)$  的范围之外(原理见下图)。另外, 不要将列/行移至自己的子项或其祖先, 若发生以上两种情形之一, 则以上函数应返回 `false`。



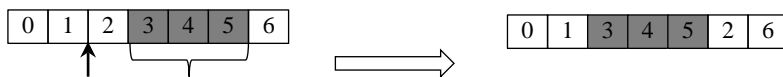
假设要在同一父索引中移动第 3~5 列, 则 `destinationChild` 必须位于 (3,5) 之外。

- 下面以 `beginMoveCloumns()` 函数为例进行讲解, `beginMoveRows()` 函数原理是相同的(原理见下图)。



假设要把第 3~5 列的三列移至 D 之前, 其语句为:

```
beginMoveColumns(sourceParent, 3,5,destinationParent, 3);
```



假设要把第 3~5 列移至第 2 列之前, 其语句为: `beginMoveColumns(parent, 3,5,parent, 2);`

### 三、拖放支持及 MIME 类型

本小节需要对拖放操作有一定了解, 下面作一简介

1、Qt 拖放的过程: 拖放操作分为拖动(Drag)和放下或放置(Drop)两种操作, 当拖动时, 需要把拖动的数据进行存储, 存储数据这一步骤称为编码, 数据以 MIME 类型(见 6.6 节)存储为 `QMimeData` 类型的对象(称为放置数据), 当执行放下操作时, 需要对放置数据进行处理, 若需要使用这些数据, 则需要把存储的数据读取出来(即解码), 然后进行处理; 当然, 若不需要这些放置数据, 也可以直接丢弃。

2、自定义拖放操作的步骤

①、启用视图的拖放支持: 标准视图自动支持内部拖放, 默认情况下, 这些视图的拖放功能未启用, 要启用视图的拖放支持, 需进行以下设置:

- 启用项目拖放: 设置视图的 `dragEnabled` 属性为 `true`。
- 拖放的模式: 即视图是否支持拖放, 是否接受放置数据, 是否接受来自自身的移动操作等。需要设置视图的 `dragDropMode` 属性。
- 若要允许用户在视图内放置内部或外部项目, 需将视图的 `viewport()` 的 `acceptDrops`



属性设置为 `true`。

- 若要向用户显示拖动的项目放置位置(通常该位置会以不同的形式显示,比如以阴影形式显示等),此时需设置视图的 `showDropIndicator` 属性。

下面为具体的代码

```
v->setDragEnabled(1); //启用视图的拖放支持
v->setAcceptDrops(1); //接受放置数据
v->setDropIndicatorShown(1); //显示放置位置
v->setDragDropMode(QAbstractItemView::DragDrop); //设置拖放模型为支持拖放
```

- ②、启用数据项的拖放支持: 重新实现 `QAbstractItemModel::flags()` 函数以提供合适的标志来指示哪些项目可以被拖动, 哪些项目将接受放置(Drop)。
- ③、编码数据: 重新实现 `QAbstractItemModel::mimeData()` 函数, 把编码后的数据保存在该函数返回的 `QMimeData` 对象中。
- ④、处理放置数据:
  - 内置模型对放置数据的处理: 不同类型的模型以不同的方式处理放置数据。列表模型和表格模型只提供了存储数据项的平面结构。因此, 他们可能会在数据放入视图中的现有项目时插入新行(和列), 或者可能会使用提供的一些数据覆盖模型中项目的内容。树模型通常能够将包含新数据的子项添加到其基础数据存储中, 因此就用户而言, 其行为将更具可预测性。
  - 对放置数据的自行处理: 通过重新实现 `QAbstractItemModel::dropMimeData()` 函数来处理放置数据, 此时需要对放置数据进行解码(即读取出 `QMimeData` 对象存储的数据的内容), 并将其插入模型的底层数据结构中(或进行其他处理), 若该函数修改了数据项或模型的尺寸, 则必须注意确保发出所有相关的信号。因为该函数需要插入或删除等操作, 所以简单的调用 `QAbstractItemModel` 子类中的已经实现了的 `setData()`, `insertRows()` 和 `insertColumns()` 等函数, 会更方便。另外, 还可以使用 `dropMimeData()` 函数的默认实现来处理放置数据, `dropMimeData()` 函数的默认实现不会覆盖模型中的任何数据, 它会将放置数据作为项目的同胞插入, 或者作为该项目的子项插入。若要使用该函数的默认实现, 就需要重新实现以下函数: `insertRows()`、`insertColumns()`、`setData()`、`setItemData()`。

### 3、自定义拖放操作的其他设置

- ①、模型能接受的拖放操作的类型:

重新实现 `QAbstractItemModel::supportedDropActions()` 函数可以指示模型能接受的拖放操作的类型(比如移动、复制等), 虽然可以给出 `Qt::DropActions` 的值的任意组合, 但需要编写代码来支持它们, 比如, 对于 `Qt::MoveAction`(移动)操作, 则模型就需要提供 `QAbstractItemModel::removeRows()` 的实现。

- ②、对多种 MIME 类型的支持

默认情况下, 内置模型和视图只支持一种 MIME 类型, 即

`application/x-qabstractitemmodeldatalist`。

若要让模型支持多种 MIME 类型, 则需要重新实现 `QAbstractItemModel::mimeTypes()`, 该函数返回拖放操作时模型可以处理的 MIME 类型列表, 注意, 自定义数据类型必须声明为元对象。

- ③、禁止在项目上放置数据

重新实现 `QAbstractItemModel::canDropMimeData()` 函数可禁止在项目上放置数据。

#### 4、下面为相关的函数及其原型

18)、virtual `Qt::ItemFlags flags(const QModelIndex &index)` const; //虚拟的

返回索引 `index` 的数据项的标志组合，默认实现为 `Qt::ItemIsEnabled` |

`Qt::ItemIsSelectable`，重新实现该函数可控制数据项的属性(比如是否可被选中，可拖放等)。 `Qt::ItemFlag` 枚举见下表

| Qt::ItemFlag 枚举                       |     |                                                                                     |
|---------------------------------------|-----|-------------------------------------------------------------------------------------|
| 标志: Qt::ItemFlags                     |     |                                                                                     |
| 作用: 该枚举描述了数据项的属性。                     |     |                                                                                     |
| 成员                                    | 值   | 说明                                                                                  |
| <code>Qt::NoItemFlags</code>          | 0   | 无属性                                                                                 |
| <code>Qt::ItemIsSelectable</code>     | 1   | 可被选择                                                                                |
| <code>Qt::ItemIsEditable</code>       | 2   | 可被编辑                                                                                |
| <code>Qt::ItemIsDragEnabled</code>    | 4   | 可拖动                                                                                 |
| <code>Qt::ItemIsDropEnabled</code>    | 8   | 是否可被丢弃                                                                              |
| <code>Qt::ItemIsUserCheckable</code>  | 16  | 可以被用户选中或取消选中                                                                        |
| <code>Qt::ItemIsEnabled</code>        | 32  | 数据项被启用                                                                              |
| <code>Qt::ItemIsAutoTristate</code>   | 64  | 三态，即该数据项的状态取决于其子项的状态，比如对于树形结构，若所有子项都被选中，则选中该选项，若所有子项未选中，则该项未选中，若子项部分选中，则该项处于部分选中状态。 |
| <code>Qt::ItemIsTristate</code>       |     | 已弃用，改为 <code>ItemIsAutoTristate</code>                                              |
| <code>Qt::ItemNeverHasChildren</code> | 128 | 该数据项从来没有子项，仅用于优化目的。                                                                 |
| <code>Qt::ItemIsUserTristate</code>   | 256 | 用户可以循环改变三种不同的状态。qt5.5                                                               |

19)、virtual `QMimeType* mimeType(const QModelIndexList &indexes)` const; //虚拟的

返回一个包含与索引列表 `indexes` 相对应的序列化数据项。若 `indexes` 为空或没有支持的 MIME 类型，应返回 0 而不是序列化的空列表。该函数用于编码拖放的数据。

20)、bool `dropMimeType(const QMimeType *data, Qt::DropAction action, int row, int column,`

`const QModelIndex &parent)`; //虚拟的

- 处理以动作 `action` 结束的拖放操作提供的的数据 `data`，即解码放置数据 `data` 并进行处理。若数据已由模型处理，则返回 `true`，否则返回 `false`。参数 `row`、`column`、`parent` 表示操作结束时数据项的位置。
- 该函数的默认实现不会覆盖模型中的任何数据，它会将放置数据作为项目的同胞插入，或者作为该项目的子项插入，比如：对 `QTreeView` 中的数据项执行放置操作时，可能会导致新数据项被插入到 `row`、`column`、`parent` 所指数据项的子项，或作为该数据项的兄弟项。当 `row` 和 `column` 为-1 时，意味着放置数据应直接位于父数据项之上，这通常意味着把数据项追加为父数据项的子项，若 `row` 和 `column` 大于或等于 0，则意味着应在指定的 `row` 和 `column` 之前发生。
- 该函数会调用 `mimeType()` 函数，以获取可接受的 MIME 类型的列表，`mimeType()` 函数，默认只返回一个默认的 MIME 类型，若要使 `mimeType()` 返回多个 MIME 类型，则需要重新实现该函数。



- 21)、virtual bool **canDropMimeData**(const QMimeData \**data*, Qt::DropAction *action*, int *row*, int *column*,  
const QModelIndex &*parent*) const; //虚拟的
- 若模型可以接受放置数据 *data*，则返回 true。默认实现仅检查 *data* 是否在 *mimeType*() 列表中至少有一种格式，以及 *action* 是否在模型的 *supportedDropActions*() 中。若要在 *row*、*column*、*parent* 上测试数据是否可以放置(drop)，则需要重新实现此函数，若不需要进行此种测试，就不必重新实现该函数。
- 22)、virtual QStringList **mimeType**() const; //虚拟的
- 返回允许的 MIME 类型列表，默认情况下，内置模型和视图使用的 MIME 类型为：*application/x-qabstractitemmodeldatalist*。若在自定义模型中实现拖放支持时，需要处理默认类型以外的格式，则需要重新实现此函数，以返回需要的 MIME 类型列表。若重新实现此函数，还必须重新实现调用该函数的 *mimeType*() 和 *dropMimeData*() 函数。
- 23)、virtual Qt::DropActions **supportedDragActions**() const; //虚拟的
- 返回模型中数据支持的动作，默认实现返回 *supportedDropActions*()。当发生拖放时，该函数的返回值被用作 *QAbstractItemView::startDrag*() 的默认值。
- 24)、virtual Qt::DropActions **supportedDropActions**() const; //虚拟的
- 返回此模型支持的放置动作，默认返回 *Qt::CopyAction*。若重新实现此函数，还必须重新实现 *dropMimeData*() 函数以处理其他动作。*Qt::DropAction* 枚举见下表

| Qt::DropAction 枚举    |        |                                                                                       |
|----------------------|--------|---------------------------------------------------------------------------------------|
| 标志: Qt::DropActions  |        |                                                                                       |
| 作用: 描述拖动时的放置动作       |        |                                                                                       |
| 成员                   | 值      | 说明                                                                                    |
| Qt::CopyAction       | 0x1    | 将数据复制到目标                                                                              |
| Qt::MoveAction       | 0x2    | 把数据从源移动到目标                                                                            |
| Qt::LinkAction       | 0x4    | 创建从源到目标的链接                                                                            |
| Qt::ActionMask       | 0xff   |                                                                                       |
| Qt::IgnoreAction     | 0x0    | 忽略动作(对数据不做任何事情)                                                                       |
| Qt::TargetMoveAction | 0x8002 | 在 Windows 上，当目标应用程序应接管 D&D 数据的所有权时使用此值，即源应用程序不应删除数据时。在 X11 上，此值用于执行移动操作，在 Mac 上不使用此值。 |

#### 四、模型标头

- 25)、virtual QVariant **headerData**(int *section*, Qt::Orientation *orientation*,  
int *role* = Qt::DisplayRole) const; //虚拟的
- virtual bool **setHeaderData**(int *section*, Qt::Orientation *orientation*, const QVariant &*value*,  
int *role* = Qt::EditRole); //虚拟的
- 以上函数用于获取和设置位于方向 *orientation* 和位置 *section* 的标头数据，其数据角色为 *role*(见下图)。重新实现 *setHeaderData* 函数时，必须明确地发送 *headerDataChanged*() 信号。

|   | 1   | GGG | 3   |
|---|-----|-----|-----|
| 1 | 123 | EEE | FFF |
| 2 | 444 | 555 | 666 |
| 3 | 777 | 888 |     |

```

pmodel->setHeaderData(1,Qt::Horizontal,"GGG");
pmodel->setHeaderData(1,Qt::Horizontal,"FFF",
 Qt::ToolTipRole);

```

## 五、另一种设置数据项的函数

26)、virtual bool **setItemData**(const QModelIndex &*index*, const QMap<int, QVariant> &*roles*); //虚拟的  
virtual QMap<int, QVariant> **itemData**(const QModelIndex &*index*) const; //虚拟的

- 以上函数主要用于获取和设置组成数据项的各数据元素的信息，
- 注意，QMap<int,QVariant>中的 int 指的是 Qt::ItemDataRole 枚举成员对应的 int 值，比如 QMap<int, QVariant> mp; mp.insert(6,f); setItemData(index, mp);表示把数据项的字体角色(Qt::FontRole 对应的 int 值为 6)设置为字体 f。
- itemData 函数是模型/视图结构中唯一能获取项目角色的函数。

示例如下：

//示例 1: itemData()函数的使用

//在索引为(0,1QModelIndex())的数据项由三个数据元素组成，代码如下

```

pmodel->setData(pmodel->index(0,1,QModelIndex()),"EEE");
pmodel->setData(pmodel->index(0,1,QModelIndex()),QIcon("F:/1.png"),Qt::DecorationRole);
pmodel->setData(pmodel->index(0,1,QModelIndex()),222,Qt::ToolTipRole);
QDebug()<<pmodel->itemData(pmodel->index(0,1,QModelIndex())); //输出如下
/* QMap(0, QVariant(QString, "EEE"))
 (1, QVariant(QIcon, QIcon(availableSizes[normal,Off]=(QSize(56, 55)),cacheKey=0x100000000)))
 (3, QVariant(int, 222))) //3 表示 Qt::ToolTipRole 对应的整数
*/

```

//示例 2: 使用 setItemData()函数设置数据项(效果见图中 EEE)

```

QFont f;
f.setPixelSize(22);
QMap<int, QVariant> mp; //创建 QMap
mp.insert(0,"EEE"); //设置 Qt::DisplayRole 角色的数据
mp.insert(1,QIcon("F:/1i.png")); //设置 Qt::DecorationRole 角色的数据
mp.insert(3,"222"); //设置角色 Qt::ToolTipRole 的数据
mp.insert(6,f); //设置角色 Qt::Font 的数据
//以下数据项由以上 4 个数据元素组成。
pmodel->setItemData(pmodel->index(0,1,QModelIndex()),mp);

```

|   | 1   | 2   | 3   |
|---|-----|-----|-----|
| 1 | 123 | EEE |     |
| 2 | 444 | 555 | 666 |

## 六、大量数据的性能优化和持久索引

27)、virtual bool **canFetchMore**(const QModelIndex &*parent*) const; //虚拟的

若 parent 可获得更多的数据，则返回 true，否则返回 false。默认实现返回 false，若该函数返回 true，则还应重新实现 fetchMore()函数以填充模型。

28)、virtual void **fetchMore**(const QModelIndex &*parent*); //虚拟的

获取 parent 所指父数据项的所有可用数据。默认实现什么也不做。

以上函数的使用例子是动态填充树模型，比如当树模型的分枝被扩展时，就需要重新实现 `fetchMore()` 以把新数据添加到模型中，此时可能还需要调用 `beginInsertRows()`、`endInsertRows()` 等函数，另外 `canFetchMore()` 函数也应重新实现。

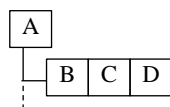
- 29)、void **changePersistentIndex**( const QModelIndex &*from*, const QModelIndex &*to*); //受保护的  
void **changePersistentIndexList**( const QModelIndexList &*from*,  
const QModelIndexList &*to*); //受保护的

以上函数用于更改持久模型索引

- 30)、QModelIndexList **persistentIndexList**() const; //受保护的  
返回模型中的持久模型索引列表。

## 七、其他函数

- 31)、virtual QHash<int, QByteArray> **roleNames**() const; //虚拟的，返回模型的角色名。  
32)、virtual QModelIndex **sibling**(int *row*, int *column*, const QModelIndex &*index*) const; //虚拟的  
返回在位置(*row*,*column*)处与索引 *index* 所指数据项同级的数据项的索引(原理见下图)，  
若没有则返回无效索引。此函数可依情况而重新实现。



假设 D 的索引为 *index*，则  
`sibling(0,1,index);` //返回 C 的索引  
`sibling(0,0,index);` //返回 B 的索引

- 33)、virtual void **sort**(int *column*, Qt::SortOrder *order* = Qt::AscendingOrder); //虚拟的  
使模型的列 *column* 按给定的顺序 *order* 排序，默认实现什么也没做。Qt::SortOrder 可取值为 Qt::AscendingOrder(升序)，Qt::DescendingOrder(降序)。  
34)、virtual QModelIndexList **match**(const QModelIndex &*start*, int *role*, const QVariant &*value*, int *hits* = 1, Qt::MatchFlags *flags* = Qt::MatchFlags(Qt::MatchStartsWith | Qt::MatchWrap)) const; //虚拟的  
从索引 *start* 开始查找与角色 *role*、值 *value* 匹配的数据项的索引列表。参数 *hits* 表示匹配数据项的数量(即命中数)，若 *hits* = -1，则会返回所有匹配的数据项的索引。参数 *flags* 用于指定搜索的方式(比如是否基于字符串的匹配，是否区分大小写等)，其中 Qt::MatchWrap 表示若搜索到达末尾则循环从重头搜索。注意：若使用的是代理模型，则返回的列表中的结果顺序可能与模型中的顺序不一致。该函数默认实现仅搜索列  
35)、virtual QModelIndex **buddy**(const QModelIndex &*index*) const; //虚拟的  
返回模型索引 *index* 表示的数据项的伙伴的模型索引。默认实现为每个项目为自己的伙伴。  
36)、virtual void **revert**(); //槽，虚拟的，丢弃缓存信息，此函数常用于行编辑。  
37)、virtual bool **submit**(); //槽，虚拟的  
将缓冲信息提交给永久存储，若没有错误，则返回 true，否则返回 false，该函数通常用于行编辑。  
38)、virtual QSize **span**(const QModelIndex &*index*) const; //虚拟的  
返回由 *index* 表示的数据项的行和列的跨度，注：目前，不使用跨度。  
39)、void **beginResetModel**(); //受保护的

开始重置模型，该函数会发送 `modelAboutToBeReset()` 信号。必须在重置模型或代理模型中的数据结构之前调用此函数。

40)、void `endResetModel()`; //受保护的

完成模型重置操作，该函数会发送 `modelReset()` 信号。必须在重置模型或代理模型中的数据结构之后调用此函数。

41)、void `resetInternalData()`; //槽，受保护的

该函数在模型的内部数据重置时才被调用，该槽为代理模型的子类提供了便利，比如子类化 `QSortFilterProxyModel` 可以维护额外的数据。由于错误的原因，此槽未出现在 qt5.0 中。

## 八、QAbstractItemModel 类中的信号

1、void `columnsAboutToBeInserted`(const QModelIndex &*parent*, int *first*, int *last*); //私有信号

2、void `columnsAboutToBeMoved`(const QModelIndex &*sourceParent*, int *sourceFirst*, int *sourceEnd*,  
const QModelIndex &*destinationParent*, int *destinationColumn*); //私有信号

3、void `columnsAboutToBeRemoved`(const QModelIndex &*parent*, int *first*, int *last*); //私有信号

4、void `rowsAboutToBeInserted`(const QModelIndex &*parent*, int *start*, int *end*); //私有信号

5、void `rowsAboutToBeMoved`(const QModelIndex &*sourceParent*, int *sourceStart*, int *sourceEnd*,  
const QModelIndex &*destinationParent*, int *destinationRow*); //私有信号

6、void `rowsAboutToBeRemoved`(const QModelIndex &*parent*, int *first*, int *last*); //私有信号

以上信号表示当列/行在模型中插入、移动、删除之前发送。这些信号用于使组件适应模型尺寸的变化，以上信号都是私有信号

7、void `columnsInserted`(const QModelIndex &*parent*, int *first*, int *last*); //私有信号

8、void `columnsMoved`(const QModelIndex &*parent*, int *start*, int *end*, const QModelIndex &*destination*,  
int *column*); //私有信号

9、void `columnsRemoved`(const QModelIndex &*parent*, int *first*, int *last*); //私有信号

10、void `rowsInserted`(const QModelIndex &*parent*, int *first*, int *last*); //私有信号

11、void `rowsMoved`(const QModelIndex &*parent*, int *start*, int *end*,  
const QModelIndex &*destination*, int *row*); //私有信号

12、void `rowsRemoved`(const QModelIndex &*parent*, int *first*, int *last*); //私有信号

以上信号表示当列/行在模型中插入、移动、删除之后发送。这些信号用于使组件适应模型尺寸的变化，以上信号都是私有信号

13、void `modelAboutToBeReset`(); //私有信号

在模型的内部状态(比如持久模型索引)失效之前，调用 `beginResetModel()` 时，发送此信号，该信号是私有信号

14、void `modelReset`(); //私有信号

在模型的内部状态(比如持久模型索引)失效之后，调用 `endResetModel()` 时，发送此信号，该信号是私有信号

15、void `layoutAboutToBeChanged`(  
const QList<QPersistentModelIndex> &*parent* = QList<QPersistentModelIndex> (),  
QAbstractItemModel::LayoutChangeHint *hint* =

QAbstractItemModel::NoLayoutChangeHint); //信号, qt5.0  
在模型布局改变之前发送该信号。该信号用于使组件适应模型布局的变化, 子类在发送该信号后应更新持久模型索引。参数 parent 表示模型布局的哪些部分正在更改, 若 parent 为空, 则表示对整个模型布局的更改, parent 中元素的顺序并不重要。参数 hint 用于提示模型正在重新布局时发生的情况, 枚举 LayoutChangeHint 取值如下表

| QAbstractItemModel::LayoutChangeHint 枚举(无标志) |   |       |
|----------------------------------------------|---|-------|
| 成员                                           | 值 | 说明    |
| QAbstractItemModel::NoLayoutChangeHint       | 0 | 无提示可用 |
| QAbstractItemModel::VerticalSortHint         | 1 | 行正在排序 |
| QAbstractItemModel::HorizontalSortHint       | 2 | 列正在排序 |

16、void layoutChanged(  
const QList<QPersistentModelIndex> &parent = QList<QPersistentModelIndex> (),  
QAbstractItemModel::LayoutChangeHint hint =  
QAbstractItemModel::NoLayoutChangeHint); //信号, qt5.0  
当模型的项的布局发生变化时, 发送此信号, 比如, 当模型被排序时。当视图接收到此信号时, 应更新项目的布局来反映这个变化。当子类化 QAbstractItemModel 或 AbstractProxyModel 时, 应确保在项的顺序更改之前或向视图展示的数据结构更改之前发送 layoutAboutToBeChanged()信号, 并在更改布局之后发送该信号, 且在发送该信号之前更新持久模型索引。参数的意义见 layoutAboutToBeChanged()信号。

17、void dataChanged(const QModelIndex &topLeft, const QModelIndex &bottomRight,  
const QVector<int> &roles = QVector<int>()); //信号  
当项目中的数据发生变化时, 发送此信号, 若项目具有相同的父级, 则受影响的项目在 topLeft 和 bottomRight 之间, 若没有相同的父级, 则行为是未定义的。参数 roles 表示实际修改了哪些数据角色, 空的 roles 表示所有角色都应被修改, roles 中元素的顺序没有任何关联。当重新实现 setData()函数时, 必须明确的发送此信号。

18、void headerDataChanged(Qt::Orientation orientation, int first, int last); //信号  
当更改标头时, 发送此信号。当重新实现 setHeaderData()函数时, 必须明确的发送此信号。

九、自定义模型

- 1、自定义模型至少需要实现以下 5 个纯虚函数  
columnCout()、rowCount()、index()、parent()、data()  
为了能添加自己的数据到模型中, 通常还需要重新实现 setData()函数, 而不重新实现 setData()则无法向模型中添加数据。
- 2、基本原理及步骤
- ①、数据: 实际数据可使用 QList、数组、整型、或单独的一个类来保存, 数据可存放在模型中, 也可存放在文件等其他地方。

②、columnCout()、rowCount()、index()、parent()这4个函数用于共同设计模型的结构，因为使用索引表示模型中的某个数据项的位置，因此设计模型索引的结构就是设计模型的结构。

- 行数和列数的设计：比如对于3行4列的表格结构columnCout()应返回4，rowCount()应返回3；对于列表结构，则因为列表只需要1列，所以columnCout()应总是返回1，rowCount()返回该列表的行数；对于树形结构模型，则更复杂，需要根据当前父节点的情况进行判断，以返回该父节点拥有的列数和行数。
- parent()函数(父模型索引)的设计：因为表格结构中的所有单元格都属于同一个父索引之下，所以可把所有单元格都视为顶级节点，因此他们的父索引可以以无效模型索引作为父索引，因此parent()可以返回一个无效模型索引；对于列表结构的模型，同样只需返回一个无效模型索引即可；对树形结构模型，此步骤比较复杂，可以通过获取当前节点的父节点及其行号和列号，然后使用createIndex()创建该父节点的索引。
- index()函数的设计：该函数用于为模型中的每个数据项创建索引，创建索引需要使用createIndex()函数，对于表格结构，只需向createIndex()函数传递当前数据项所在的行号、列号及使用的数据的指针即可；对于列表结构，则列号始终为0，其余同表格结构；对于树形结构，需要向该函数传递当前数据项位于父索引中的行号、列号及使用的数据的指针。

③、data()函数的返回值决定了视图上应显示的数据，也就是说在界面上用户看到的数据是由该函数返回的，若返回不当的值，则数据无法正常显示在视图上，下面以使用内置的标准视图类为例来讲解怎样设计此函数。data()函数会被视图类调用多次，视图每次都会向data传递一个不同的role(角色)参数值，然后视图根据data返回的值，设置该role的数据，因此在设计data函数的返回值时，需要根据role的不同值返回不同的数据，以使视图正确的显示。

## 示例：自定义表格模型

//m.h 文件内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>
#include <iostream>
using namespace std;
class D:public QAbstractItemModel{ //继承抽象类QAbstractItemModel
public:
 int col,row; //表格模型的列数和行数
 QList<QVariant> s1; //这是模型管理的数据
 QList<int> rol; //存储数据的角色
//构造函数表示创建二个i行j列的表格模型
 D(int i, int j):row(i),col(j){ for(int k=0;k<col*row;k++){ s1<<QVariant(); rol<<-1;} }
//①、返回表格模型的行数
 int rowCount(const QModelIndex &parent = QModelIndex ()) const{return row; }
//②、返回表格模型的列数
 int columnCount(const QModelIndex &parent =QModelIndex ()) const{return col;}
//③、返回表格模型的父索引，因为所有单元格都是顶级节点，所以使用无效节点作为父节点
```

读者可把这两项单独封装在一个类(比如 xxxItem)中进行管理，Qt 内置模型就是这样处理的



```

 QModelIndex parent(const QModelIndex &index) const{ return QModelIndex();}
//④、为每个单元格创建一个唯一的索引
 QModelIndex index(int row, int column, const QModelIndex &parent = QModelIndex ()) const
 {
 cout<<"index"<<endl; //测试语句
 //其他模型可能需要判断传递进来的索引是否有效。
 //if(!hasIndex(row, column, parent))return QModelIndex();
 /*本示例仅需简单的为传递进来的单元格创建一个索引即可，注意第3个参数的使用，请参阅
 createIndex()原型。*/
 return createIndex(row, column, (void*)&(s1.at(row*column+column))); }

//⑤、返回视图上显示的数据，该函数会被视图多次调用(注：其他虚函数同样会被 Qt 调用多次)
 QVariant data(const QModelIndex &index, int role = Qt::DisplayRole) const{
 //测试用，读者可看到每次调用时 Qt 内部传递进来的 role(角色)的值。
 cout<<"data="<<role<<endl;
 int i=index.row()*col+index.column(); //单元格所在数据 s1 中的位置
//以下对不同角色返回的数据会被视图使用以使数据正确的显示在其单元格中。
 /*对于 QListView 和 QTreeView 必须对所在节点设置大小，否则这两个视图不会显示任何内容(因
 为节点大小为 0)*/
 //if(role==Qt::SizeHintRole)return QSize(55,55);
 /*设置角色 CheckStateRole 的数据，本示例返回一个无效的 QVariant 作为该角色的数据，若返
 回有效值，会使单元格的左侧显示一个复选框。*/
 if(role==Qt::CheckStateRole)return QVariant();
 //设置单元格中数据的对齐方式，本示例为左侧垂直居中对齐
 if(role==Qt::TextAlignmentRole) return Qt::AlignLeft|Qt::AlignVCenter;
 //设置角色 DecorationRole(图片)的数据
 if(role==Qt::DecorationRole)
 //若用户设置了 DecorationRole 角色的数据，则返回用户为该单元格设置的数据。
 if(rol.at(i)==Qt::DecorationRole) return s1.at(i);
 //若用户为角色 EditRole 或 DisplayRole 角色设置了数据，则返回用户为该单元格设置的数据。
 if(role==Qt::EditRole|role==Qt::DisplayRole)
 if(rol.at(i)==Qt::EditRole|rol.at(i)==Qt::DisplayRole) return s1.at(i);
 //其余角色使用无效数据
 return QVariant(); } //data() 结束
//⑥、重载 setData 以使用户可以向模型中添加数据
 bool setData(const QModelIndex &index,const QVariant &value, int role=Qt::EditRole)
 {
 /*使用数据 value 和角色 role 分别替换列表 s1 和 rol 中原有的值，使用 replace 便于下一个示
 例(拖放)的使用。*/
 s1.replace(index.row()*col+index.column(),value);
 rol.replace(index.row()*col+index.column(),role);
 emit dataChanged(index, index); //数据改变后，需要发送此信号。
 return true; } //返回 true，表示数据设置成功。
};
#endif // M_H

//m.cpp 文件内容
#include "m.h"
int main(int argc, char *argv[]){ QApplication aa(argc,argv);
 D *d=new D(3,3); //创建一个 3 行 3 列的表格模型
 QTableView *pv2=new QTableView; //使用表格视图来显示以上模型管理的数据

```

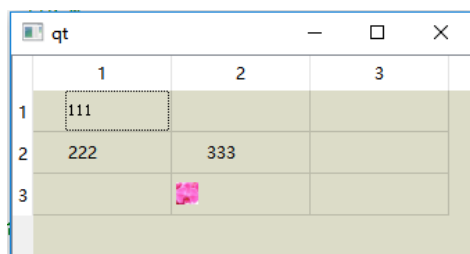
```

/*读者也可使用以下视图来显示模型 d 中的数据，但要去掉 data() 中对 if(role==Qt::SizeHintRole)
 的注释，还要注意，若使用树形视图显示，有可能会使程序崩溃，因为本示例未完整的实现树形模
 型结构*/
//QListView *pv2=new QListView;
//QTreeView *pv2=new QTreeView;

//向模型中添加数据
d->setData(d->index(0,0,QModelIndex()),"111",Qt::DisplayRole);
d->setData(d->index(1,0,QModelIndex()),222);
d->setData(d->index(1,1,QModelIndex()),333);
d->setData(d->index(2,1,QModelIndex()),QIcon("F:/li.png"),Qt::DecorationRole);
pv2->setModel(d); pv2->resize(333,222); pv2->show(); return aa.exec(); }

```

运行结果及说明(由图可见，数据显示正确)



### 示例：自定义拖放操作

注：本示例只需在上一示例的 `main` 函数中增加以下代码(启用视图的拖放功能)。

```

pv2->setDragEnabled(true);
pv2->viewport()->setAcceptDrops(true);
pv2->setDropIndicatorShown(true);
pv2->setDragDropMode(QAbstractItemView::DragDrop);

```

然后再在类 `D` 之中增下如下函数的定义即可

```

//①、重新实现 flags 函数，以开启模型的拖放功能
Qt::ItemFlags flags(const QModelIndex &index) const{
 return Qt::ItemIsEditable|Qt::ItemIsDragEnabled|Qt::ItemIsDropEnabled|
 Qt::ItemIsEnabled|Qt::ItemIsSelectable; }
//②、重新实现 supportedDropActions 以使模型即可复制又可移动
Qt::DropActions supportedDropActions() const{return Qt::CopyAction|Qt::MoveAction;}
//③、重新实现 canDropMimeData 函数，以决定单元格是否允许放置拖动过来的数据。
bool canDropMimeData(const QMimeData *data,Qt::DropAction action, int row, int column,
 const QModelIndex &parent) const
{ /*本示例 parent 参数有效，row 和 column 是无效的，以下代码表示，第 3 行第 3 列不接受来自拖
 放的数据。*/
 if(parent.row()==2&&parent.column()==2) return false;else return true; }
//④、重新实现 mimeData 函数，以编码拖动时的数据(就是把拖动时的数据保存起来，并返回)
QMimeData* mimeData(const QModelIndexList &indexes) const{
 QMimeData *const pm=new QMimeData(); //创建一个 QMimeData 对象。

```

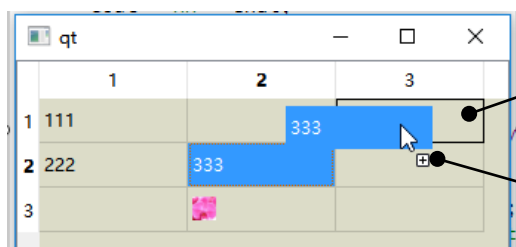


```

/*注意: indexes 是被拖动的数据项的索引, 若同时拖动了多个数据项, 则 indexes 才会包含多个数据项。*/
for(int k=0;k<indexes.size();k++){
 const QModelIndex &i=indexes.at(k);
 if (i.isValid()) {
 QByteArray t;
 //提取数据项的索引为 i 角色为 DisplayRole 的数据
 QVariant v=data(i, Qt::DisplayRole);
 t.append(v.toString()); //把提取的数据保存在 t 之中
 pm->setData("text/plain", t); //然后把 t 中的数据以 MIME 类型的形式编码到 pm 中。
 } //if 结束
} //for 结束
return pm; } //返回编码后的数据
//⑤、重新实现 dropMimeData 函数, 以解码拖动时的数据并对其进行处理
bool dropMimeData(const QMimeData *data, Qt::DropAction action,int row, int column,
const QModelIndex &parent)
{
 /* 本示例 parent 参数有效, row 和 column 无效, 以下代码表示保存单元格所在数据 s1 中的位置。索引 parent 是拖动之后需要放置的位置的索引。*/
 int i=parent.row()*col+parent.column();
 QByteArray t=data->data("text/plain"); //解码由 mimeType 函数编码后的数据。
 //若是复制操作, 则把新位置 parent 处的数据重置为 t
 if(action==Qt::CopyAction) { setData(parent,t); }
 //若是移动操作, 则把新位置 parent 处的数据重置为 t
 if(action==Qt::MoveAction) { setData(parent,t); }
 return true; } //返回 true 表示数据已处理。

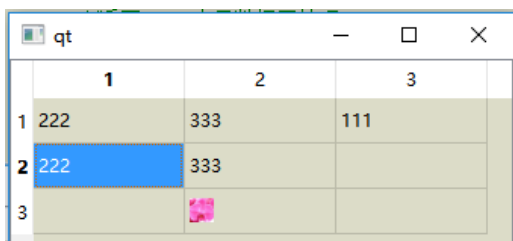
```

## 运行结果及说明



假设要拖至此单元格 (0,2), 则 dropMimeData() 的 parent 参数的索引是 (0,2) 单元格的, mimeType() 的 indexes 参数的索引是 (1,1) 单元格的

此处的 “+” 表示正在执行复制操作



本示例可成功复制或移动内容为文本的单元格, 并使用被移动单元格的内容替换新单元格的内容。注意, 只能在已有的 3 行 3 列之间移动或复制, 超出这范围程序会崩溃。另外含有图片的单元格不能成功复制或移动, 因为程序未对非文本内容单元格进行处理。

|   | 1   | 2   | 3   |
|---|-----|-----|-----|
| 1 | 111 |     |     |
| 2 | 222 | 333 |     |
| 3 |     |     | 333 |

本示例禁止把拖动的内容放置于单元格(2,2)。

### 示例：自定义插入/删除操作

由于篇幅原因，请读者自行完成自定义 `insertColumns`、`insertRows`、`removeColumns`、`removeRows` 等函数，有了以上的示例，实现这几个函数是不难的。

## 8.3 QAbstractItemView 类(视图基类)

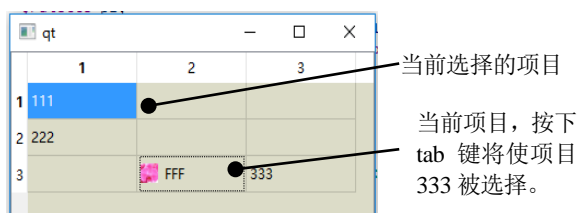
### 一、基本概念

- 1、QAbstractItemView 类继承自 QAbstractScrollArea，后者又继承自 QFrame，该类是 Qt 所有视图类的基类，Qt 的所有视图都需要子类化该类。注意：该类是抽象类，因此不能创建该类的对象。
- 2、该类的构造函数原型为：**QAbstractItemView**(QWidget\* parent = Q\_NULLPTR); //构造函数
- 3、该类支持以下的键盘操作

| QAbstractItemView 类支持的键盘操作 |                           |
|----------------------------|---------------------------|
| 方向键                        | 改变当前项目并选择该项目              |
| Ctrl+方向键                   | 改变当前项目但不选择该项目             |
| Shift+方向键                  | 改变当前项目并选择该项目(之前选择的项目不会取消) |
| Ctrl+Space                 | 切换当前项目的选择                 |
| Tab/Backtab                | 使后/前一个项目成为当前项目            |
| Home/End                   | 选择第一个/最后一个项目              |
| Page up/Page down          | 向上/下滚动显示可见的行              |
| Ctrl+A                     | 选择所有项目                    |
|                            |                           |

- 4、当前项目(或当前索引)：这是重要概念

- 当前项目用于键盘导航和焦点指示(原理见下图)，若按下编辑键 F2，将会编辑当前项目
- 当前项目不一定是当前已被选择(高亮)的项目，当前项目可以处于被选择状态，也可以不处于被选择状态。
- 只能有一个当前项目，但可以同时有多个被选择的项目。
- 当前项目通常是具有焦点的项目。



- 5、本小节暂时不介绍怎样实现自定义的视图，因为自定义视图需要自行绘制(通常在 paintEvent()函数内完成)，所以需在讲解委托之后讲解怎样自定义视图。

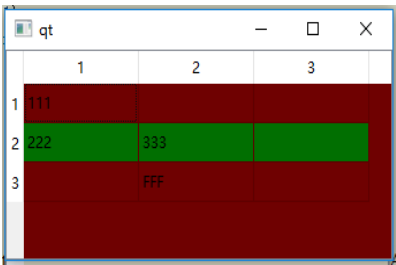
### 二、QAbstractItemView 类中的属性

与滚动有关的属性

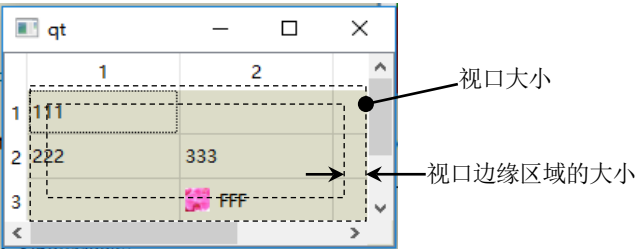
- 1、**autoScroll**: bool      **访问函数**: bool hasAutoScroll() const; void setAutoScroll(bool);  
是否启用自动滚动，默认为 true(启用)
- 2、**alternatingRowColors**: bool  
**访问函数**: bool alternatingRowColors() const; void setAlternatingRowColors(bool);  
是否使用交替的颜色绘制背景色，若为 true，则使用 QPalette::AlternateBase 和 QPalette::Base 绘制背景色，否则使用 QPalette::Base 绘制背景色。默认为 false。

示例，效果见右图

```
QTableView *pv21=new QTableView;
... //其他代码
QPalette p1;
p1.setColor(QPalette::Base,QColor(111,1,1));
p1.setColor(QPalette::AlternateBase,
 QColor(1,111,1));
pv21->setPalette(p1);
pv21->setAlternatingRowColors(true);
```



- 3、**autoScrollMargin**: int      **访问函数**: int autoScrollMargin() const; void setAutoScrollMargin(int);  
描述触发自动滚动时视口边缘区域的大小，当用户拖动至该区域时视图将自动滚动，默认为 16 像素。原理见下图



- 4、**horizontalScrollMode**: ScrollMode  
**访问函数**: ScrollMode horizontalScrollMode() const; void setHorizontalScrollMode(ScrollMode);  
void resetHorizontalScrollMode();
- 5、**verticalScrollMode**: ScrollMode  
**访问函数**: ScrollMode verticalScrollMode() const; void setVerticalScrollMode(ScrollMode );  
void resetVerticalScrollMode();

以上属性分别表示视图在水平或垂直方向上的滚动模式，即一次滚动一个像素还是一个项目的内容，默认值由 QStyle::SH\_ItemView\_ScrollMode 决定。ScrollMode 枚举取值如下

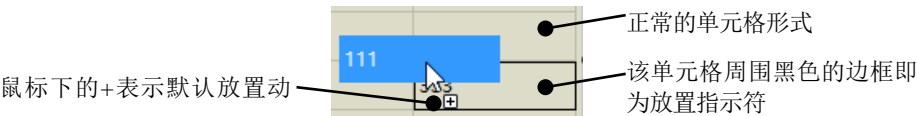
| QAbstractItemView::ScrollMode 枚举(无标志) |   |                                   |
|---------------------------------------|---|-----------------------------------|
| 作用：描述视图的滚动模式                          |   |                                   |
| 成员                                    | 值 | 说明                                |
| QAbstractItemView::ScrollPerItem      | 0 | 一次滚动一个项目的内容                       |
| QAbstractItemView::ScrollPerPixel     | 1 | 一次滚动一个像素的内容，此模式的滚动距离还与滚动时的单步大小有关。 |

与拖放有关的属性

6、**dragEnabled**: bool                   访问函数: bool dragEnabled() const; void setDragEnabled(bool);  
是否启用拖放操作，注意，视图默认未启用拖放。

7、**defaultDropAction**: Qt::DropAction  
访问函数: Qt::DropAction defaultDropAction() const; void setDropAction(Qt::DropAction);  
此属性描述拖放时的默认放置动作(见属性 showDropIndicator 的图示)。若未设置该属性，则当支持的放置动作支持 Qt::CopyAction 时，放置操作为 Qt::CopyAction。枚举 Qt::DropAction 见 QAbstractItemModel::supportedDropActions()函数。

8、**showDropIndicator**: bool  
访问函数: bool showDropIndicator() const; void setDropIndicatorShow(bool);  
拖动项目在项目放下时，是否向用户显示放置指示符(通常该位置会以不同的形式显示，比如以阴影形式显示等，见下图)



9、**dragDropMode**: DragDropMode  
访问函数: DragDropMode dragDropMode() void setDragDropMode(DragDropMode);  
描述视图支持的拖放模式。DragDropMode 枚举见下表

| QAbstractItemView::DragDropMode 枚举(无标志) |   |                   |
|-----------------------------------------|---|-------------------|
| 作用：描述视图的拖放模式                            |   |                   |
| 成员                                      | 值 | 说明                |
| QAbstractItemView::NoDragDrop           | 0 | 不支持拖放             |
| QAbstractItemView::DragOnly             | 1 | 仅支持拖动操作           |
| QAbstractItemView::DropOnly             | 2 | 仅支持放置操作           |
| QAbstractItemView::DragDrop             | 3 | 支持拖动和放置操作         |
| QAbstractItemView::InternalMove         | 4 | 仅接受来自其自身(不是复制)的操作 |

10、**dragDropOverwriteMode**: bool  
访问函数: bool dragDropOverwriteMode() const; void setDragDropOverwriteMode(bool);  
描述施放操作放置时的覆盖行为，若该值为 true，则选定的数据将在放置时覆盖现在的数  
据，若为 false，则在放置数据时，所选数据将作为新项目被插入。对于 QListView 和  
QTreeView 默认为 false，对于 QTableView 默认为 true。**该属性对不同视图的行为并不一  
致。**

编辑项目

11、**editTriggers**: EditTriggers  
访问函数: EditTriggers editTriggers() const; void setEditTriggers(EditTriggers);  
描述编辑项目的触发方式，枚举 EditTrigger 见下表

## QAbstractItemView::EditTrigger 枚举

### 标志: QAbstractItemView::EditTriggers

作用: 描述编辑项目的触发方式

| 成员                                 | 值  | 说明                                                                               |
|------------------------------------|----|----------------------------------------------------------------------------------|
| QAbstractItemView::NoEditTriggers  | 0  | 无法编辑                                                                             |
| QAbstractItemView::CurrentChanged  | 1  | 只要当前项目发生变时, 即开始编辑。比如该项目由未被选择状态变为选择状态(即高亮)时即开始编辑, 若该项已是选择状态, 则在该项上单击或双击都不会进入编辑状态。 |
| QAbstractItemView::DoubleClicked   | 2  | 双击项目时开始编辑                                                                        |
| QAbstractItemView::SelectedClicked | 4  | 单击已选定的选项目时开始编辑, 注意: 若该项目之前不是选择状态(高亮), 则单击该项目不会进入编辑状态。                            |
| QAbstractItemView::EditKeyPressed  | 8  | 按下编辑键(通常为 F2)时开始编辑                                                               |
| QAbstractItemView::AnyKeyPressed   | 16 | 按下任意键开始编辑                                                                        |
| QAbstractItemView::AllEditTriggers | 31 | 以上所有方式的组合。                                                                       |

## 选择项目

### 12、selectionBehavior: SelectionBehavior

访问函数: QAbstractItemView::SelectionBehavior selectionBehavior() const;

void setSelectionBehavior(QAbstractItemView::SelectionBehavior)

描述视图的选择行为(比如按行或列选择等)。SelectionBehavior 枚举见下表

## QAbstractItemView::SelectionBehavior 枚举(无标志)

作用: 描述视图的选择行为

| 成员                               | 值 | 说明                                |
|----------------------------------|---|-----------------------------------|
| QAbstractItemView::SelectItems   | 0 | 一次选择一个项目                          |
| QAbstractItemView::SelectRows    | 1 | 按行选择, 即选择某个项目时, 会同时选中该项目所在行的所有项目。 |
| QAbstractItemView::SelectColumns | 2 | 按列选择, 即选择某个项目时, 会同时选中该项目所在列的所有项目。 |

### 13、selectionMode: SelectionMode

访问函数: QAbstractItemView::SelectionMode selectionMode() const;

void setSelectionMode(QAbstractItemView::SelectionMode)

描述视图的选择模式(比如是否可以同时选择多个项目等)。SelectionMode 枚举见下表

## QAbstractItemView::SelectionMode 枚举(无标志)

作用: 描述视图的选择模式

| 成员                                 | 值 | 说明                              |
|------------------------------------|---|---------------------------------|
| QAbstractItemView::NoSelection     | 0 | 项目不能被选择                         |
| QAbstractItemView::SingleSelection | 1 | 只能同时选中一个项目。即选择该项目时, 会取消其他已选中项目。 |

|                                        |   |                                                                                                               |
|----------------------------------------|---|---------------------------------------------------------------------------------------------------------------|
| QAbstractItemView::MultiSelection      | 2 | 可以同时选中多个项目。当选择一个项目时，会使该项目的选择状态在选择(高亮)/取消选择之间切换，同时，其他项目的选择状态保持不变，可通过鼠标拖放来切换多个项目的选择状态。                          |
| QAbstractItemView::ExtendedSelection   | 3 | 选择该项目时，会取消其他已选中项目，但是若在单击某个项目时按住了 Ctrl 键，则该项目的选择状态会在选择或取消选择之间切换，若在单击某个项目时按住了 Shift 键，则会选择或取消选择当前项目和单击项目之间的所有项。 |
| QAbstractItemView::ContiguousSelection | 4 | 选择该项目时，会取消其他已选中项目，但是若在单击某个项目时按住了 Shift 键，则会选择或取消选择当前项目和单击项目之间的所有项。                                            |

## 其他属性

- 14、**tabKeyNavigation**: bool      **访问函数**: bool tabKeyNavigation() const; void setTabKeyNavigation(bool);  
描述 tab 键是否用于项目导航，若为 true，则按下 tab 键会选择(高亮) 下一个项目；若为 false，则按下 tab 键没有任何反应，但是若 setEditTriggers()函数的设置包含 QAbstractItemView::AnyKeyPressed，则按下 tab 键会直接编辑下一个项目。
- 15、**textElideMode**: Qt::TextElideMode  
**访问函数**: Qt::TextElideMode textElideMode() const; void setTextElideMode(Qt::TextElideMode);  
描述省略符 “...” 的位置，所有视图的默认值都是 Qt::ElideRight(即右侧)。枚举 Qt::TextElideMode 见第 4 章部件公用枚举章节。
- 16、**iconSize**: QSize          **访问函数**: QSize iconSize() const; void setIconSize(const QSize &);  
**信号**: iconSizeChanged(const QSize&);  
描述视图项目图标的大小。

## 三、QAbstractItemView 类中的函数

### 1、纯虚函数

- 1)、virtual void **setSelection**(const QRect &*rect*, QItemSelectionModel::SelectionFlags *flags*) = 0; //受保护的  
将选择标志 flags 应用于矩形 rect 中的项目，或 rect 所触及到的项目。在实现自己的视图时，该函数应该调用 selectionModel()->select(selection,flags)，其中的 selection 要么是空的 QModelIndex，要么是包含在 rect 中的所有项目的 QItemSelection。枚举 QItemSelectionModel::SelectionFlags 参见 QItemSelectionModel 类
- 2)、virtual QRect **visualRect**(const QModelIndex &*index*) const = 0;    //纯虚函数  
返回由索引 index 所指项目占用的视口上的矩形(注意：矩形是包括位置和大小的)。若项目显示在多个区域，则该函数应返回包含索引的主区域，而不是索引可能包含、触及或导致绘图完整区域。
- 3)、virtual QRegion **visualRegionForSelection**(const QItemSelection &*selection*) const = 0;    //受保护的  
从 selection 选择的项目的视口返回区域。
- 4)、virtual int **horizontalOffset**() const = 0;    //纯虚函数，受保护的
- 5)、virtual int **verticalOffset**() const = 0;    //纯虚函数，受保护的

返回视图的水平/垂直偏移量，必须返回视口在窗口部件中 x 和 y 坐标的偏移量(通常滚动条的值(value())就是需要的偏移量)。

- 6)、virtual QModelIndex **indexAt**(const QPoint &*point*) const = 0; //纯虚函数  
返回视口坐标点 *point* 处的模型索引
- 7)、virtual bool **isIndexHidden**(const QModelIndex &*index*) const = 0; //纯虚函数，受保护的  
若索引 *index* 所引用的项目隐藏在视图中，则返回 true，否则返回 false，隐藏是视图特定的特性。
- 8)、virtual QModelIndex **moveCursor**(CursorAction *cursorAction* ,  
Qt::KeyboardModifiers *modifiers*) = 0; //纯虚函数，受保护的  
根据 *cursorAction* 和键盘修饰符 *modifiers*, 返回指向视图中的下一个对象的模型索引。  
枚举 CursorAction 见下表

| QAbstractItemView::CursorAction 枚举(无标志) //受保护的 |   |             |
|------------------------------------------------|---|-------------|
| 成员                                             | 值 | 说明          |
| QAbstractItemView::MoveUp                      | 0 | 移至当前项目上方的项目 |
| QAbstractItemView::MoveDown                    | 1 | 移至当前项目下方的项目 |
| QAbstractItemView::MoveLeft                    | 2 | 移至当前项目左侧的项目 |
| QAbstractItemView::MoveRight                   | 3 | 移至当前项目右侧的项目 |
| QAbstractItemView::MoveHome                    | 4 | 移至左上角的项目    |
| QAbstractItemView::MoveEnd                     | 5 | 移至右下角的项目    |
| QAbstractItemView::MovePageUp                  | 6 | 在当前项目上方移动一页 |
| QAbstractItemView::MovePageDown                | 7 | 在当前项目下方移动一页 |
| QAbstractItemView::MoveNext                    | 8 | 移至当前项目之后的项目 |
| QAbstractItemView::MovePrevious                | 9 | 移至当前项目之前的项目 |

- 9)、virtual void **scrollTo**(const QModelIndex &*index*, ScrollHint *hint* = EnsureVisible) = 0; //纯虚函数  
必要时滚动视图，以确保 *index* 处的项目是可见的，视图根据参数 *hint* 来定位项目。  
ScrollHint 枚举见下表

| QAbstractItemView::ScrollHint 枚举(无标志) |   |                 |
|---------------------------------------|---|-----------------|
| 成员                                    | 值 | 说明              |
| QAbstractItemView::EnsureVisible      | 0 | 滚动以确保项目可见       |
| QAbstractItemView::PositionAtTop      | 1 | 滚动，以将项目定位于视口顶部。 |
| QAbstractItemView::PositionAtBottom   | 2 | 滚动，以将项目定位于视口底部。 |
| QAbstractItemView::PositionAtCenter   | 3 | 滚动，以将项目定位于视口中心。 |

## 2、与模型有关的函数

- 10)、QAbstractItemModel \***model**() const; //返回该视图使用的模型。  
virtual void **setModel**(QAbstractItemModel \**model*); //虚拟的
- 设置视图的模型，
  - 除非视图被设置为模型的父对象，否则视图不会获取模型的所有权。



- 注意：使用该函数后，还会重新创建并设置一个新的选择模型 (QItemSelectionModel)，并且会替换之前由 setSelectionModel() 设置的任何选择模型，但是旧的选择模型不会被删除，可使用如下代码删除旧的选择模型

```
QItemSelectionModel *p = view->selectionModel();
```

```
view->setModel(model);
```

```
delete p;
```

若旧模型和旧的选择模型都没有父项，或父项是长期存在的对象，则最好调用它们的 QObject::deleteLater() 函数来显示删除。

### 3、与拖放有关的函数

11)、DropIndicatorPosition **dropIndicatorPosition()** const; //受保护的

返回放置指示符(见属性 showDropIndicator)相对于最近项目的位置。枚举

DropIndicatorPosition 见下表

| QAbstractItemView::DropIndicatorPosition 枚举(无标志) //受保护的 |   |                                                    |
|---------------------------------------------------------|---|----------------------------------------------------|
| 成员                                                      | 值 | 说明                                                 |
| QAbstractItemView::OnItem                               | 0 | 项目将放置在索引处                                          |
| QAbstractItemView::AboveItem                            | 1 | 项目将放置在索引的上方                                        |
| QAbstractItemView::BelowItem                            | 2 | 项目将放置在索引的下方                                        |
| QAbstractItemView::OnViewport                           | 3 | 该项目将被放置到没有项目的视口区域中。每个视图处理放置在视口上的项目的方式取决于所使用的模型的行为。 |

12)、virtual void **startDrag**(Qt::DropActions *supportedActions*); //虚拟的，受保护的

通过使用参数 supportedActions 调用 drag->exec() 来启用拖动。

### 4、与委托有关的函数

13)、void **setItemDelegate**(QAbstractItemDelegate \**delegate*);

将该视图和模型的委托设置为 delegate，现有的委托将被移除，但不会被删除，

QAbstractItemView 不会接受委托的所有权。设置委托后，便可以完全控制项目的编辑和显示。注意：最好不要在视图之间共享同一个实例的委托。

14)、void **setItemDelegateForColumn**(int *column*, QAbstractItemDelegate \**delegate*);

void **setItemDelegateForRow**(int *row*, QAbstractItemDelegate \**delegate*);

将该视图和模型所在列 column 或行 row 的委托设置为 delegate，设置委托后，列或行上的所有项目都将由 delegate 进行绘制和管理，而不再使用默认委托(即 itemDelegate()), 现有的委托将被移除，但不会被删除，QAbstractItemView 不会接受委托的所有权。若委托已分配给行和列，则行/列委托将优先管理相交的单元索引。

15)、QAbstractItemDelegate\* **itemDelegate()** const;

QAbstractItemDelegate\* **itemDelegate**(const QModelIndex &*index*) const;

QAbstractItemDelegate\* **itemDelegateForColumn**(int *column*) const;

QAbstractItemDelegate\* **itemDelegateForRow**(int *row*) const;

以上函数表示获取视图和模型在索引 index 或列 column 或行 row 处使用的项目委托。

## 5、与编辑有关的函数

- 16)、void **edit**(const QModelIndex &*index*); //槽  
若索引 *index* 指定的项目是可编辑的，则编辑该项目。
- 17)、virtual bool **edit**(const QModelIndex &*index*, EditTrigger *trigger*, QEvent \**event*); //虚拟的，受保护的  
在索引 *index* 处开始编辑项目，必要时创建编辑器，若视图的 State(状态)为 EditingState，则返回 true，否则返回 false。参数 *trigger* 指定引起编辑的方式，相关的事件由 *event* 描述。若把 *trigger* 指定为 QAbstractItemView::AllEditTriggers，则可以强制编辑，EditTrigger 枚举见 editTriggers 属性。
- 18)、virtual void **commitData**(QWidget \**editor*); //槽，虚拟的，受保护的  
将编辑器 *editor* 中的数据提交给模型。
- 19)、virtual **closeEditor**(QWidget \**editor*, QAbstractItemDelegate::EndEditHint *hint*); //槽，虚拟的，受保护的  
关闭编辑器 *editor*，然后释放它，参数 *hint* 用于指示视图应如何响应编辑操作结束时的行为，比如 *hint* 可以表示当编辑操作结束时，打开视图中的下一个项目进行编辑。枚举 QAbstractItemDelegate::EndEditHint *hint* 见 QAbstractItemDelegate 类。
- 20)、virtual void **editorDestroyed**(QObject \**editor*); //槽，虚拟的，受保护的  
销毁编辑器 *editor*。
- 21)、void **openPersistentEditor**(const QModelIndex &*index*);  
在索引 *index* 处打开持久编辑器，若没有编辑器，则委托将创建一个新的编辑器。普通编辑器当离开该项目时会自动关闭，而持久编辑器则会一直存在，直到调用 closePersistentEditor() 关闭。
- 22)、bool **isPersistentEditorOpen**(const QModelIndex &*index*) const; //qt5.10  
返回索引 *index* 处项目的持久编辑器是否已打开。
- 23)、void **closePersistentEditor**(const QModelIndex &*index*);  
关闭索引 *index* 处的持久编辑器

## 6、与选择有关的函数

- 24)、QItemSelectionModel \***selectionModel**() const; //返回当前的选择模型
- 25)、virtual void **setSelectionModel**(QItemSelectionModel \**selectionModel*); //虚拟的  
把当前的选择模型设置为 *selectionModel*。注意：若在此函数之后调用 setModel() 函数，则该函数设置的选择模型会被调用 setModel() 函数时创建的选择模型替换。旧的选择模型应由应用程序来删除，也就是说，若旧选择模型未被其他视图使用时，当删除其父对象时，旧选择模型会被自动删除，但是，若旧选择模型没有父对象，或父对象是一个长期存在的对象，则最好调用它的 QObject::deleteLater() 函数来显示删除。
- 26)、void **clearSelection**(); //槽  
取消所有选择的项目，当前索引不会被改变。
- 27)、virtual void **selectAll**(); //槽，虚拟的  
选择视图中的所有项目，
- 28)、virtual QModelIndexList **selectedIndexes**() const; //虚拟的，受保护的

返回视图中所有可被选择项目和非隐藏项目的索引列表，该列表不包含重复项目，且未排序。

- 29)、virtual QModelIndex::SelectionFlags **selectionCommand**(const QModelIndex &*index*,  
const QEvent \**event* = Q\_NULLPTR) const; //虚拟的，受保护的  
返回在更新包含索引 *index* 的选择时，使用的 SelectionFlags(选择标志)，事件 *event* 是用户输入事件，比如鼠标或键盘事件。重新实现此函数可定义自己的选择行为。枚举 QModelIndex::SelectionFlags 参见 QModelIndex 类
- 30)、virtual void **selectionChanged**(const QModelIndex &*selected*,  
const QModelIndex &*deselected*) const; //槽，虚拟的，受保护的  
当选择改变时，调用此槽函数，*deselected* 表示之前的选择(可能为空)，*selected* 表示新的选择。

## 7、与索引有关的函数

- 31)、QModelIndex **currentIndex**() const; //返回当前项目的模型索引
- 32)、void **setCurrentIndex**(const QModelIndex &*index*); //槽
- 把索引 *index* 处的项目设置为当前项目。除非选择模式是 NoSelection，否则该项目会被选择(高亮显示)。
  - 注意：此函数还会更新用户执行的任何新选择的起始位置。
  - 另外，还可使用 QModelIndex 类中的函数来设置当前项目，其方法如下：  
QAbstractItemView::selectionModel()->setCurrentIndex()。
- 33)、QModelIndex **rootIndex**() const;  
返回模型根项目的模型索引，根项目是视图的顶级项目的父项目，根目录的索引可能是无效索引。
- 34)、virtual void **setRootIndex**(const QModelIndex &*index*); //槽，虚拟的  
把索引 *index* 所指项目设置为根项目

## 8、与滚动有关的函数

- 35)、void **scrollToBottom**(); //槽  
void **scrollToTop**(); //槽  
以上函数表示将视图滚动到底/顶部
- 36)、void **setDirtyRegion**(const QRegion &*region*); //受保护的  
将区域 *region* 标记为脏区域(Dirty Region)，并安排更新，只有在自定义视图子类时，才需调用此函数。
- 37)、QPoint **dirtyRegionOffset**() const; //受保护的  
返回视图中 Dirty Region(脏区域)的偏移量。若使用 ScrollDirtyRegion()并在该类的子类中实现了 paintEvent()，则应使用从该函数返回的偏移量来转换 paint 事件给出的区域。
- 38)、void **scrollDirtyRegion**(int *dx*, int *dy*); //受保护的  
通过在相反方向移动脏区域，以使视图滚动(dx,dy)像素。只有在自定义视图子类实现滚动视口时，才需调用此函数。若在该类的子类中重新实现

QAbstractScrollArea::scrollContentsBy()函数，则应在调用视口上的 QWidget::scroll()之前调用此函数，或者只调用 QAbstractItemView::update()函数。

39)、virtual QSize **viewportSizeHint**() const; //虚拟的，受保护的，qt5.2

QAbstractScrollArea::viewportSizeHint()的重新实现。返回视口的大小提示(不含滚动条)

40)、virtual bool **viewportEvent**(QEvent \*event); //虚拟的，受保护的

QAbstractScrollArea::viewportEvent()的重新实现。若 event 是 QEvent::ToolTip 或 QEvent::WhatsThis，则此函数用于处理工具提示、What's this 模式，对于其他事件，则传递给父类的 viewportEvent()函数处理。

## 9、与外观有关的函数

41)、virtual QStyleOptionViewItem **viewOptions**() const; //虚拟的，受保护的

返回由视图的调色板(palette)、字体(font)、状态、对齐等填充的 QStyleOptionViewItem 结构。

42)、QSize **sizeHintForIndex**(const QModelIndex &index) const;

返回索引 index 所指项目的大小提示或无效索引的无效大小

43)、virtual int **sizeHintForColumn**(int column) const; //虚拟的

返回列 column 的宽度大小提示，若没有模型，则返回-1，此函数用于具有水平标头的视图，以根据列 column 查找标头部分的大小提示。

44)、virtual int **sizeHintForRow**(int row) const; //虚拟的

返回行 row 的高度大小提示，若没有模型，则返回-1，此函数用于具有垂直标头的视图，以根据列 row 查找标头部分的大小提示。注意：要控制行的高度，必须重新实现 QAbstractItemDelegate::sizeHint()函数。

45)、void **update**(const QModelIndex &index); //槽

更新索引 index 占用的区域

46)、virtual void **updateGeometries**(); //槽，虚拟的，受保护的

更新视图的子部件的几何形状。

## 10、其他

47)、QWidget\* **indexWidget**(const QModelIndex &index) const;返回索引 index 处的 QWidget 部件。

48)、void **setIndexWidget**(const QModelIndex &index, QWidget \*widget);

把部件 widget 设置到索引 index 所指项目上，并把部件 widget 的所有权传递给视口。若索引无效(比如根索引)，则此函数什么也不做，必须把 widget 的 autoFillBackground 属性设置为 true，否则部件的背景将是透明的。若部件 A 被替换为部件 B，则部件 A 将被删除。

49)、State **state**() const; //受保护的

void **setState**(State state); //受保护的

以上函数表示，返回或设置项目视图的状态，通常只在重新实现自己的视图时才有意义。枚举 State 见下表

---

QAbstractItemView::State 枚举(无标志) //受保护的

---

| 作用：描述视图的状态                            |   |              |
|---------------------------------------|---|--------------|
| 成员                                    | 值 | 说明           |
| QAbstractItemView::NoState            | 0 | 默认状态         |
| QAbstractItemView::DraggingState      | 1 | 用户正在拖动项目     |
| QAbstractItemView::DragSelectionState | 2 | 用户正在选择项目     |
| QAbstractItemView::EditingState       | 3 | 用户正在编辑项目     |
| QAbstractItemView::ExpandingState     | 4 | 用户正在打开一个项目分支 |
| QAbstractItemView::CollapsingState    | 5 | 用户正在关闭一个项目分支 |
| QAbstractItemView::AnimatingState     | 6 | 视图正在执行动画。    |

50)、virtual void **reset()**; //槽，虚拟的

重置视图的内部状态。此函数将重置编辑器、滚动条位置、选择等，并且现在的更改不会被提交(即在编辑器中改变的数据不会提交给模型)，若要在重置视图时保存更改，需要重新实现此函数。

51)、virtual void **keyboardSearch**(const QString &search); //虚拟的

移动并选择与字符串 search 最匹配的项，若没有该项目，则什么都不会发生。在默认实现中，若 search 为空，或自上次搜索以后的时间间隔超过 QApplication::keyboardInputInterval()，则会重置搜索。

52)、virtual void **dataChanged**(const QModelIndex &topLeft, const QModelIndex &bottomRight,

const QVector<int> &roles = QVector<int>()); //槽，虚拟的，受保护的

当在模型中更改具有角色 roles 的项目时，会调用此槽函数，更改的项目范围为 topLeft 到 bottomRight，若只有一个项目被改变，则 topLeft == bottomRight。若 roles 为空，则意味着所有项目都已更改。

53)、virtual **currentChanged**(const QModelIndex &current,

const QModelIndex &previous); //槽，虚拟的，受保护的

当新项目成为当前项目时，会调用此槽函数，previous 表示上一个当前项目，current 表示新项目。

54)、virtual void **rowsAboutToBeRemoved**(const QModelIndex &parent,

int start, int end); //槽，虚拟的，受保护的

当行即将被移除时，调用此槽函数，移除的行是父级 parent 下从 start 开始到 end 结束的行。

55)、virtual void **rowsInserted**(const QModelIndex &parent, int start, int end); //槽，虚拟的，受保护的

在插入行时调用此槽函数，新行是父级 parent 下从 start 开始到 end 结束的行。

56)、void **executeDelayedItemsLayout**(); //受保护的

执行预定的布局(Scheduled Layout)，而无需等待事件处理开始。

57)、void **scheduleDelayedItemsLayout**(); //受保护的

计划视图中项目的布局，以便在事件处理开始时执行。即使在事件处理之前多次调用该函数，视图也只会执行一次布局。

## 11、重新实现的父类中的函数

1)、以下函数为重新实现的 QWidget 类中的相关函数。

```

virtual void dragEnterEvent(QDragEnterEvent* event); //虚拟的，受保护的
virtual void dragLeaveEvent(QDragLeaveEvent* event); //虚拟的，受保护的
virtual void dragMoveEvent(QDragMoveEvent* event); //虚拟的，受保护的
virtual void dropEvent(QDropEvent* event); //虚拟的，受保护的
virtual void focusInEvent(QFocusEvent* event); //虚拟的，受保护的
virtual focusNextPrevChild(bool next); //虚拟的，受保护的
virtual focusOutEvent(QFocusEvent* event); //虚拟的，受保护的
virtual void inputMethodEvent(QInputMethodEvent* event); //虚拟的，受保护的
virtual QVariant inputMethodQuery(Qt::InputMethodQuery query); //虚拟的
virtual void keyPressEvent(QKeyEvent* event); //虚拟的，受保护的
virtual void mouseMoveEvent(QMouseEvent* event); //虚拟的，受保护的
virtual void resizeEvent(QResizeEvent* event); //虚拟的，受保护的

```

- 2)、virtual void **mouseDoubleClickEvent**(QMouseEvent\* *event*); //虚拟的，受保护的  
若双击位于有效项目上，该函数会发送 QAbstractItemView::doubleClicked()信号，并调用该项目上的 edit()函数。
- 3)、virtual void **mousePressEvent**(QMouseEvent\* *event*); //虚拟的，受保护的  
该函数会发送 QAbstractItemView::pressed()信号
- 4)、virtual void **mouseReleaseEvent**(QMouseEvent\* *event*); //虚拟的，受保护的  
若项目被按下，该函数会发送 QAbstractItemView::clicked()信号。
- 5)、virtual void **timerEvent**(QTimerEvent\* *event*); //虚拟的，受保护的  
该函数为重新实现的 QObject 类中的相关函数。

## 四、QAbstractItemView 类中的信号

- 1、void **pressed**(const QModelIndex &*index*); //信号  
当按下鼠标时，发送此信号，index 为鼠标按下的项目索引，只有索引有效时才会发送此信号，可使用 QApplication::mouseButtons()函数来获取按下鼠标按钮的状态。
- 2、void **clicked**(const QModelIndex &*index*); //信号  
当单击鼠标左键时发出此信号，index 为鼠标点击时的项目索引，只有索引有效时才会发送此信号。
- 3、void **activated**(const QModelIndex &*index*); //信号  
当用户激活索引 index 指定的项目时，发送此信号，如何激活信号取决于平台，通常可以双击若按下回车键激活项目。
- 4、void **viewportEntered**(); //信号  
当鼠标进入视口时，发送此信号，需启用鼠标跟踪功能才能使用此信号。
- 5、void **entered**(const QModelIndex &*index*); //信号  
当鼠标进入索引 index 指定的项目时，发送此信号，需启用鼠标跟踪功能才能使用此信号。
- 6、void **doubleClicked**(const QModelIndex &*index*); //信号  
双击鼠标时，发送此信号，index 表示双击的项目的索引，只有在 index 有效时，才会发送此信号。



## 五、经常使用的函数

QAbstractItemView 类中的函数虽然比较多,但由于受保护的函数只能在子类化时在子类的成员函数中调用,因此若不自定义视图,而使用 Qt 预定义的标准视图(比如 QTableView 等),则能使用的函数就并不多了,现整理如下

| QAbstractItemView 类中的常用函数 |                                                                                                                                      |
|---------------------------|--------------------------------------------------------------------------------------------------------------------------------------|
| 纯虚函数                      | indexAt() //返回点所在模型索引<br>visualRect() //返回索引所指单元格的矩形大小<br>scrollTo() //将索引所指单元格滚动至指定位置                                               |
| 模型                        | model()、setModel() //设置和获取模型                                                                                                         |
| 委托                        | setItemDelegate()、setItemDelegateForColumn()、setItemDelegateForRow();<br>itemDelegate()、itemDelegateForColumn()、itemDelegateForRow() |
| 编辑                        | edit() //编辑某数据项<br>openPersistentEditor()、closePersistentEditor()、isPersistentEditorOpen() //持久编辑器                                   |
| 选择                        | selectionModel()、setSelectionModel()、clearSelection()、selectAll()                                                                    |
| 索引                        | currentIndex()、setCurrentIndex()、rootIndex()、setRootIndex()                                                                          |
| 滚动                        | scrollToBottom(); scrollToTop()                                                                                                      |
| 其他                        | indexWidget()、setIndexWidget()                                                                                                       |

### //示例: QAbstractItemView 类中常用函数的使用

//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>

class B:public QWidget{ Q_OBJECT
public: QStandardItem *d; QTableView *pv2; QPushButton *pb1,*pb2,*pb3,*pb4,*pb5;
 B(QWidget *p=0):QWidget(p){
 pv2=new QTableView(this); //视图是 QFrame 部件,可添加到 QWidget 中
 pv2->move(22,22); pv2->resize(333,155);
 d=new QStandardItem(3,3,this);
 //向模型中添加数据
 d->setData(d->index(0,0,QModelIndex()),"111",Qt::DisplayRole);
 d->setData(d->index(1,0,QModelIndex()),222);
 d->setData(d->index(2,2,QModelIndex()),333);
 d->setData(d->index(2,1,QModelIndex()),QIcon("F:/li.png"),Qt::DecorationRole);
 d->setData(d->index(2,1,QModelIndex()),"FFF");
 pv2->setModel(d); //设置视图 pv2 关联的模型
 //向窗口中添加按钮
 pb1=new QPushButton("edit",this); pb1->move(22,199);
 pb2=new QPushButton("openPerEdit",this); pb2->move(99,199);
 pb3=new QPushButton("colsePerEdit",this); pb3->move(177,199);
 pb4=new QPushButton("selAll",this); pb4->move(22,233);
 pb5=new QPushButton("clearSel",this); pb5->move(99,233);

 QObject::connect(pb1,&QPushButton::clicked,this,&B::f1);
 QObject::connect(pb2,&QPushButton::clicked,this,&B::f2);
 QObject::connect(pb3,&QPushButton::clicked,this,&B::f3);
```

```

//选择所有数据项
QObject::connect(pb4, &QPushButton::clicked, pv2, &QTableView::selectAll);
//清除所选数据项
QObject::connect(pb5, &QPushButton::clicked, pv2, &QTableView::clearSelection);
}

public slots:
void f1() {
 if(pv2->currentIndex().isValid()) //若当前项是有效的，则编辑当前项
 pv2->edit((pv2->currentIndex()));
 else{
 QAbstractItemModel *p=pv2->model(); //获取 pv2 关联的模型
 //把索引(0,0)的项设置为当前项目
 pv2->setCurrentIndex(p->index(0,0,QModelIndex()));
 pv2->edit((pv2->currentIndex())); //编辑当前项目
 }
}

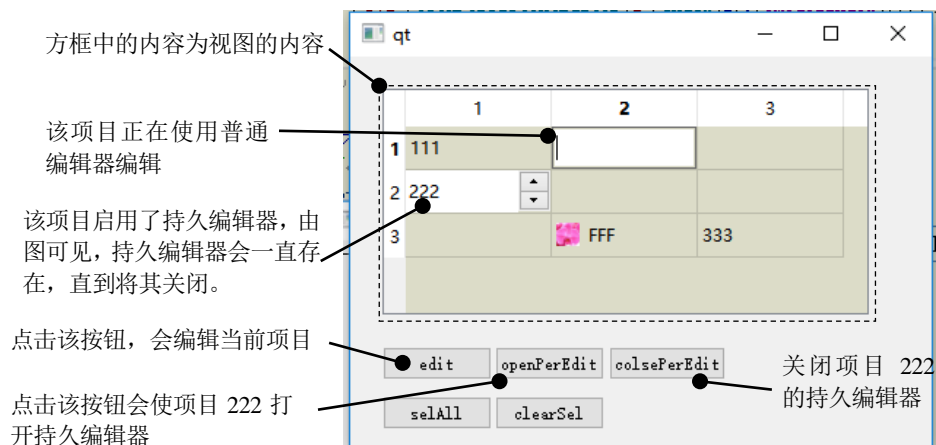
void f2() { pv2->openPersistentEditor(d->index(1,0,QModelIndex())); } //打开永久编辑器
void f3() { pv2->closePersistentEditor(d->index(1,0,QModelIndex())); } //关闭永久编辑器
};

#endif // M_H

//m.cpp 文件的内容
#include "m.h"
int main(int argc, char *argv[]){
 QApplication aa(argc, argv);
 B w;
 w.resize(444, 333);
 w.show();
 return aa.exec();
}

```

## 运行结果及说明





## 8.4 QItemSelectionModel 类(选择模型)与

### QItemSelection 类(选择范围)

#### 一、基本原理

1、QItemSelectionModel 类继承自 QObject，该类主要用于跟踪视图中所选择的项目。

2、重要概念：当前项目或当前索引见 QAbstractItemView 章节的讲解

3、使用选择模型的步骤：

要使选择模型的设置显示在视图上，需要使选择模型关联与视图相同的项目模型 (QAbstractItemModel)，并且应使用视图的 setSelectionModel() 函数把该选择模型作为视图的选择模型。具体代码类似如下：

```
QStandardItemModel *pd.... //模型
QTableView *pv.... //视图
QItemSelectionModel *ps //选择模型
.... //其他代码
pv->setModel(pd); //设置与视图关联的项目模型，注意：视图的 setModel() 函数会删除之前
 //设置的选择模型，所以此步骤应位于 setSelectionModel() 之前。
ps->setModel(pd); //设置与选择模型关联的项目模型
pv->setSelectionModel(ps); //设置与视图关联的选择模型，注意：此步必须在 pv->setModel(pd); 之后
```

#### 二、QItemSelection 类

1、QItemSelection 类继承自 QList 类，该类的原型如下：

```
class Q_CORE_EXPORT QItemSelection : public QList<QItemSelectionRange>{ ...};
```

可见该类存储的是 QItemSelectionRange 类型的列表。QItemSelectionRange 类用于管理模型中选定项目范围的信息，该类通常通过 QItemSelection 进行使用，很少被直接使用，该类的详细内容详见帮助文档。

2、QItemSelection 类用于描述一个选择范围，其方法为指定左上角和右下角项目的索引，原理见下图。注意：指定了一个选择范围，不代表该范围内的项目会被选择(高亮显示)，项目是否被选择需由选择模型 QItemSelectionModel 进行设置。

|   | 1   | 2   | 3   |
|---|-----|-----|-----|
| 1 | 111 |     |     |
| 2 | 222 |     |     |
| 3 |     | FFF | 333 |

图中被选择部分的选择范围可使用左上角项目 (1,1) 的索引和右下角项目 (2,2) 的索引来指定。

#### 2、QItemSelection 类的函数

1)、[QItemSelection\(\)](#); //默认构造函数

- 2)、 **QItemSelection**(const QModelIndex &*topLeft*, const QModelIndex &*bottomRight*);  
构造一个从左上角 *topLeft* 到右下角 *bottomRight* 的选择范围。通常使用该构造函数
- 3)、 void **select**(const QModelIndex &*topLeft*, const QModelIndex &*bottomRight*);  
该函数通常于默认构造函数结合使用，如下  
**QItemSelection** \*s =new **QItemSelection**();  
s->select(*topLeft*, *bottomRight*);
- 4)、 bool **contains**(const QModelIndex &*index*) const;  
或选择范围包含索引 *index* 则返回 true，否则返回 false。
- 5)、 QModelIndexList **indexes**() const;  
返回选择范围的索引列表。
- 6)、 void **merge**(const QItemSelection &*other*, QItemSelectionModel::SelectionFlags *command*);  
使用选择标志 *command* 将选择 *other* 与当前选择范围合并。  
注意：选择标志只支持 QItemSelectionModel::Select、QItemSelectionModel::Deselect、  
QItemSelectionModel::Toggle。  
选择标志 QItemSelectionModel::SelectionFlag 见 QItemSelectionModel 类。
- 7)、 static void **split**(const QItemSelectionRange &*range*, const QItemSelectionRange &*other*,  
QItemSelection \**result*); //静态的  
拆分范围，使用选择范围 *other* 拆分选择范围 *range*，从 *range* 中移除所有 *other*，并将结果存入 *result* 之中。

### 三、QItemSelectionModel 类中的函数和属性

#### 1、属性及构造函数

- 1)、 **selectedIndexes**: const QModelIndexList //qt5.5  
访问函数：QModelIndexList selectedIndexes()const;  
信号：void selectionChanged(const QItemSelection &*selected*, const QItemSelection &*deselected*)  
获取被选择项目索引的列表，该列表不包含重复项，也不进行排序。
- 2)、 **QItemSelectionModel**(QAbstractItemModel \**model* = Q\_NULLPTR); //构造函数  
**QItemSelectionModel**(QAbstractItemModel \**model*, QObject \**parent*);

#### 2、模型

- 3)、 void **setModel**(QAbstractItemModel \**model*); //qt5.5  
设置与该选择模型相关联的项目模型为 *model*，并发送 modelChanged()信号。
- 4)、 const QAbstractItemModel \***model**() const;  
QAbstractItemModel \***model**(); //qt5.5  
以上函数表示返回与该选择模型相关联的项目模型

#### 3、以编程的方式选择项目

- 5)、 virtual void **select**(const QModelIndex &*index*, QItemSelectionModel::SelectionFlags *command*); //槽  
virtual void **select**(const QItemSelection &*selection*, QItemSelectionModel::SelectionFlags *command*); //槽

- 以上函数表示，使用给定的选择标志 `command` 选择模型项目索引 `index` 或选择项目 `selection`，并发送 `selectionChanged()` 信号。 `SelectionFlag` 枚举见下表，
- 使用第一个函数一次只能选择一个项目，若想同时选择多个项目，需要使用第二个函数，并由 `QItemSelection` 对象指定一个选择范围。

#### 4、当前项目(或当前索引)

- 6)、 `QModelIndex` `currentIndex()` `const`;
- 返回当前项目的模型索引，若没有当前项目，则返回无效索引
- 7)、 `virtual void` `setCurrentIndex(const QModelIndex &index,`  

`QItemSelectionModel::SelectionFlags command`); //槽，虚拟的

把 `index` 所指项目设置为当前项目，并发送 `currentChanged()` 信号。根据指定的 `command`，`index` 也可以成为当前选择的一部分。 `SelectionFlag` 枚举见下表

| QItemSelectionModel::SelectionFlag 枚举  |        |                            |
|----------------------------------------|--------|----------------------------|
| 标志：QItemSelectionModel::SelectionFlags |        |                            |
| 作用：选择标志，描述选择模型的更新方式                    |        |                            |
| 成员                                     | 值      | 说明                         |
| QItemSelectionModel::NoUpdate          | 0x0000 | 不做出选择                      |
| QItemSelectionModel::Clear             | 0x0001 | 清除已选择索引                    |
| QItemSelectionModel::Select            | 0x0002 | 选择指定的索引，该选项不会清除之前的选择。      |
| QItemSelectionModel::Deselect          | 0x0004 | 取消选择指定的索引                  |
| QItemSelectionModel::Toggle            | 0x0008 | 切换选择/取消选择                  |
| QItemSelectionModel::Current           | 0x0010 | 更新当前选择(注意：当前选择不是当前索引和当前项目) |
| QItemSelectionModel::Rows              | 0x0020 | 将索引扩展至整行                   |
| QItemSelectionModel::Columns           | 0x0040 | 将索引扩展至整列                   |
| QItemSelectionModel::SelectCurrent     |        | Select 和 Current 的组合       |
| QItemSelectionModel::ToggleCurrent     |        | Toggle 和 Current 的组合       |
| QItemSelectionModel::ClearAndSelect    |        | Clear 和 Select 的组合         |

#### 5、清除内容

- 8)、 `virtual void` `reset()`; //槽，虚拟的
- 清除选择模型(该函数会同时清除已选择的索引和当前索引)，不发送任何信号
- 9)、 `virtual void` `clear()`; //槽，虚拟的
- 清除选择模型(该函数会同时清除已选择的索引和当前索引)，并发送 `selectionChanged()` 和 `currentChanged()` 信号。
- 10)、 `virutal void` `clearCurrentIndex()`; //槽，虚拟的
- 清除当前索引(该函数不会清除已选择的索引)，发送 `currentChanged()` 信号
- 11)、 `void` `clearSelection()`; //槽
- 清除选择模型中的选择(该函数不会清除当前索引)，发送 `selectionChanged()` 信号

#### 6、获取所选择项目的信息

12)、const QModelIndex **selection**() const;

返回存储在选择模型中的选择范围

13)、bool **isSelected**(const QModelIndex &**index**) const;

若模型索引 **index** 被选择，则返回 true。

14)、bool **hasSelection**() const;

若选择模型包含任意的选择范围，则返回 true，否则返回 false。

15)、QModelIndexList **selectedColumns**(int **row** = 0) const;

返回“行 **row** 所在列中”的所有行被选中的列的索引(原理见下图)。

16)、QModelIndexList **selectedRows**(int **column** = 0) const;

返回“列 **column** 所在行中”的所有列被选中的行的索引(原理见下图)。

|   | 1   | 2   | 3   |
|---|-----|-----|-----|
| 1 | 111 |     |     |
| 2 | 222 |     |     |
| 3 |     | FFF | 333 |

**selectedColumns(1);**

返回空列表，因为第 1 行(行 222 所在行)所在的 3 列，没有任一列是被全部选中的。

|   | 1   | 2   | 3   |
|---|-----|-----|-----|
| 1 | 111 |     |     |
| 2 | 222 |     |     |
| 3 |     | FFF | 333 |

**selectedColumns(1);**

返回第 1 行 1 列(注：从 0 开始编号)单元格的索引，因为第 1 行所在的 3 列当中，只有第 1 列的 3 行(所有行)被全部选中，所以返回第 1 行第 1 列的索引。

|   | 1   | 2   | 3   |
|---|-----|-----|-----|
| 1 | 111 |     |     |
| 2 | 222 |     |     |
| 3 |     | FFF | 333 |

**selectedColumns(1);**

返回第 1 行 2 列(注：从 0 开始编号)单元格的索引，原因同上一图示。

|   | 1   | 2   | 3   |
|---|-----|-----|-----|
| 1 | 111 |     |     |
| 2 | 222 |     |     |
| 3 |     | FFF | 333 |

**selectedColumns(1);**

返回空列表，因为第 1 行(行 222 所在行)所在的 3 列，没有任一列是被全部选中的。

17)、bool **columnIntersectsSelection**(int **column**, const QModelIndex &**parent**) const

bool **rowIntersectsSelection**(int **row**, const QModelIndex &**parent**) const;

以上函数表示如果在列 **column**/行 **row** 中选择了父项 **parent** 中的任意项目，则返回 true。

18)、bool **isColumnSelected**(int **column**, const QModelIndex &**parent**) const;

bool **isRowSelected**(int **row**, const QModelIndex &**parent**) const;

以上函数表示如果在列 **column**/行 **row** 中选择了父项 **parent** 中的所有项目，则返回 true。

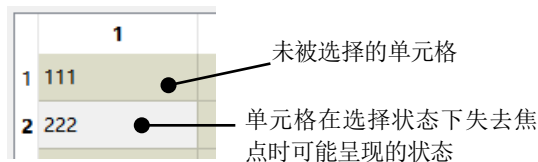
以上函数比在同一列或行的所有项目上依次调用 **isSelected()** 进行判断更快，且忽略不可选项目。

## 7、其他

- 19)、void **emitSelectionChanged**(const QTableWidgetItem &*newSelection*,  
const QTableWidgetItem &*oldSelection*); //受保护的  
比较 newSelection 和 oldSelection, 并发送 selectionChanged()信号。

## 8、信号

- 1)、void **selectionChanged**(const QTableWidgetItem &*selected*, const QTableWidgetItem &*deselected*) //信号  
当选择变化时, 发送此信号。选择的变化是指项目在取消选择和被选择之间变化, 注意: 当前索引的改变不会发送此信号, 另外, 重置项目模型时也不会发送此信号。
- 2)、void **currentChanged**(const QModelIndex &*current*, const QModelIndex &*previous*); //信号  
当当前项目改变时, 发送此信号。之前的项目索引 previous 被替换为选择的当前项目索引 current。重置项目模型时不会发送此信号。
- 3)、void **currentRowChanged**(const QModelIndex &*current*, const QModelIndex &*previous*); //信号  
void **currentColumnChanged**(const QModelIndex &*current*, const QModelIndex &*previous*); //信号  
以上信号表示, 若当前项目 current 发生变化, 且与前一个当前项目 previous 的行/列不相同, 发送此信号。重置项目模型时不会发送此信号。
- 4)、void **modelChanged**(QAbstractItemModel \**model*); //信号, qt5.5  
使用 setModel()成功设置模型时, 发送此信号
- 9、下图为单元格在选择状态下失去焦点时的外观



## 示例：选择标志及清除函数的使用 (设计的界面及说明见程序之后的说明)

//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>
class B:public QWidget{ Q_OBJECT
public: QStandardItemModel *d; QTableView *pv2; QTableWidgetItemSelectionModel *ps;
 QPushButton *pb1,*pb2,*pb3,*pb4,*pb5,*pb6,*pb7,*pb8,*pb9,*pb10;
 B(QWidget *p=0):QWidget(p){
 pv2=new QTableView(this); //视图是 QFrame 部件, 可添加到 QWidget 中
 pv2->move(22, 22); pv2->resize(333, 144); //设置视图的位置和大小
 d=new QStandardItemModel(3, 3, this);
//向模型中添加数据
 d->setData(d->index(0, 0, QModelIndex()), "111", Qt::DisplayRole);
 d->setData(d->index(1, 0, QModelIndex()), 222);
 d->setData(d->index(2, 2, QModelIndex()), 333);
 d->setData(d->index(2, 1, QModelIndex()), QIcon("F:/li.png"), Qt::DecorationRole);
 d->setData(d->index(2, 1, QModelIndex()), "FFF");
//选择模型
```

```

 QTableWidgetItem *pp=pv2->selectionModel(); //获取视图 pv2 的选择模型
pv2->setModel(d); //该设置会同时清除 pv2 之前的选择模型
ps=new QTableWidgetItem;
ps->setModel(d); //设置选择模型关联的项目模型
pv2->setSelectionModel(ps); //设置视图 pv2 的选择模型，注意：该语句应位于
//pv2->setModel(d)之后
delete pp; //删除 pv2 之前的选择模型
//向窗口中添加按钮和标签
 QLabel *pa=new QLabel("Select()", this);pa->move(22, 177);
 pb1=new QPushButton("Select", this); pb1->move(22, 199);
 pb2=new QPushButton("Clear|Select", this);pb2->move(99, 199);
 pb3=new QPushButton("Toggle", this);pb3->move(177, 199);
 pb4=new QPushButton("Rows", this);pb4->move(255, 199);
 pb5=new QPushButton("Rows|Select", this);pb5->move(333, 199);

 QLabel *pal=new QLabel("Current()", this);pal->move(22, 233);
 pb6=new QPushButton("CSelect", this);pb6->move(22, 255);
 pb7=new QPushButton("NoUpdate", this);pb7->move(99, 255);

 QLabel *pa2=new QLabel("Clear()", this);pa2->move(22, 288);
 pb8=new QPushButton("Clear", this);pb8->move(22, 311);
 pb9=new QPushButton("clearCurrentIndex", this);pb9->move(99, 311);
 pb10=new QPushButton("clearSelection", this);pb10->move(211, 311);
//关联信号和槽
 QObject::connect(pb1, &QPushButton::clicked, this, &B::f1);
 QObject::connect(pb2, &QPushButton::clicked, this, &B::f1);
 QObject::connect(pb3, &QPushButton::clicked, this, &B::f1);
 QObject::connect(pb4, &QPushButton::clicked, this, &B::f1);
 QObject::connect(pb5, &QPushButton::clicked, this, &B::f1);
 QObject::connect(pb6, &QPushButton::clicked, this, &B::f1);
 QObject::connect(pb7, &QPushButton::clicked, this, &B::f1);
 QObject::connect(pb8, &QPushButton::clicked, this, &B::f1);
 QObject::connect(pb9, &QPushButton::clicked, this, &B::f1);
 QObject::connect(pb10, &QPushButton::clicked, this, &B::f1);
}
public slots: void f1() {
 pv2->setFocus(); //使视图获得焦点。
 QPushButton *po=(QPushButton*)sender(); //获取发送信号的部件
 if(po->text()=="Select") //若点击的是 Select 按钮
 ps->select(d->index(2, 1, QModelIndex()), QTableWidgetItem::Select);
 if(po->text()=="Clear|Select") //若点击的是 Clear|Select 按钮
 ps->select(d->index(2, 1, QModelIndex()), QTableWidgetItem::ClearAndSelect);
 if(po->text()=="Toggle")
 ps->select(d->index(2, 1, QModelIndex()), QTableWidgetItem::Toggle);
 if(po->text()=="Rows")
 ps->select(d->index(2, 1, QModelIndex()), QTableWidgetItem::Rows);
 if(po->text()=="Rows|Select")
 ps->select(d->index(2, 1, QModelIndex()),
 QTableWidgetItem::Select|QTableWidgetItem::Rows);
 if(po->text()=="CSelect")
 ps->setCurrentIndex(d->index(2, 1, QModelIndex()), QTableWidgetItem::Select);
 if(po->text()=="NoUpdate")
 ps->setCurrentIndex(d->index(2, 1, QModelIndex()), QTableWidgetItem::NoUpdate);
}

```

```

 if(po->text()=="Clear")
 if(po->text()=="clearCurrentIndex")
 if(po->text()=="clearSelection")
 });
#endif // M_H

```

//m.cpp 文件的内容

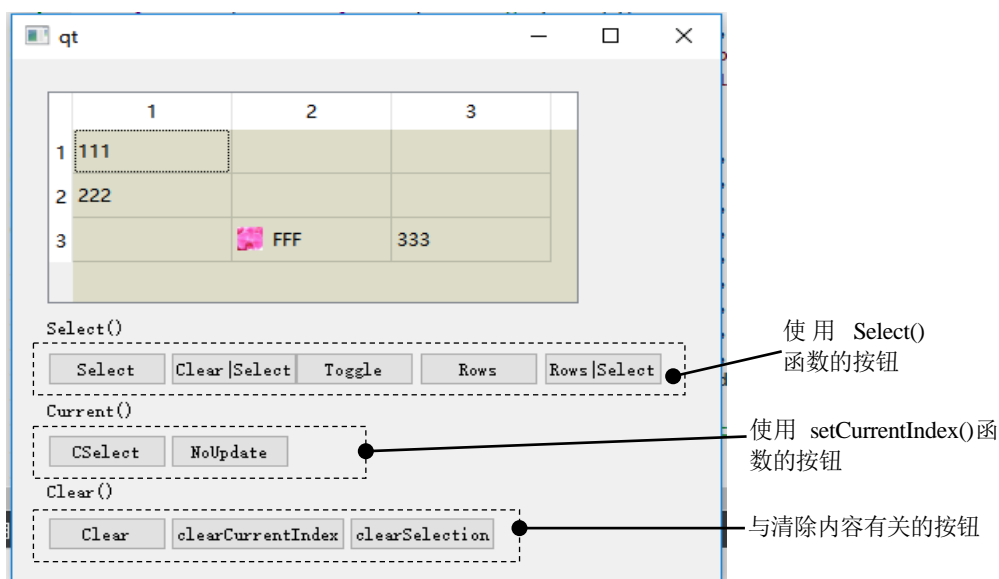
```

#include "m.h"
int main(int argc, char *argv[]) {
 QApplication aa(argc, argv);
 B w;
 w.resize(444, 355);
 w.show();
 return aa.exec();
}

```

运行结果及说明

## 1、初始界面



## 2、选择标志 QTableWidgetItem::Select

按住 Ctrl 先选择项目 111 再选择项目 222，然后点击 Select 按钮后的效果，可见 QTableWidgetItem::Select 未清除之前的选择。注意：此时的当前项目为 222，按下 F2 将编辑此项目而不会编辑选中的项目 FFF

|   | 1   | 2   |     |
|---|-----|-----|-----|
| 1 | 111 |     |     |
| 2 | 222 |     |     |
| 3 |     | FFF | 333 |

## 3、选择标志 QTableWidgetItem::ClearAndSelect

按住 Ctrl 先选择项目 111 再选择项目 222，然后点击按钮 Clear|Select 的效果，可见 QTableWidgetItem::ClearAndSelect 清除了之前的选择并选择了指定的项目 FFF。注意：此时的当前项目为 222，按下 F2 将编辑此项目而不会编辑选中的项目 FFF

|   | 1   | 2   |     |
|---|-----|-----|-----|
| 1 | 111 |     |     |
| 2 | 222 |     |     |
| 3 |     | FFF | 333 |

## 4、选择标志 QTableWidgetItem::Toggle

按住 Ctrl 先选择项目 111 再选择项目 222，然后重复点击按钮 Toggle，此时数据项 FFF 将在选择和取消选择之间切换，但是当前项目始终为 222，按下 F2 将编辑项目 222，而不会编辑项目 FFF

|   | 1   | 2   |     |
|---|-----|-----|-----|
| 1 | 111 |     |     |
| 2 | 222 |     |     |
| 3 |     | FFF | 333 |

5、选择标志 QItemSelectionModel::Rows

按住 Ctrl 先选择项目 111 再选择项目 222，然后点击按钮 Rows，视图没有变化，因为此时选择标志既未指定 Select 也未指定 Clear，所以项目 FFF 即不会选中也不会被清除

|   | 1   | 2   |     |
|---|-----|-----|-----|
| 1 | 111 |     |     |
| 2 | 222 |     |     |
| 3 |     | FFF | 333 |

6、选择标志 QItemSelectionModel::Select|QItemSelectionModel::Rows

按住 Ctrl 先选择项目 111 再选择项目 222，然后点击按钮 Rows|Select，此时项目 FFF 所在行的所有数据项都被选中了

|   | 1   | 2   | 3   |  |
|---|-----|-----|-----|--|
| 1 | 111 |     |     |  |
| 2 | 222 |     |     |  |
| 3 |     | FFF | 333 |  |

7、更改当前项目

按住 Ctrl 先选择项目 111 再选择项目 222，然后点击按钮 NoUpdate，此时数据项 FFF 成为当前项，但未被选中，按下 F2 将编辑该项目。若点击按钮 CSelect，则会同时使 FFF 被选中。

|   | 1   | 2   |     |
|---|-----|-----|-----|
| 1 | 111 |     |     |
| 2 | 222 |     |     |
| 3 |     | FFF | 333 |

8、清除内容

按住 Ctrl 先选择项目 111 再选择项目 222，然后点击按钮 NoUpdate。使视图先呈现出如右图所示情形，然后分别按下图所示按钮进行测试

|   | 1   | 2   |     |
|---|-----|-----|-----|
| 1 | 111 |     |     |
| 2 | 222 |     |     |
| 3 |     | FFF | 333 |

在上图的基础上按下 Clear 按钮，此时所有选择被清除(包括当前项目)，按下 F2 不会有任何项目被编辑(因为没有当前项目)

|   | 1   | 2   |     |
|---|-----|-----|-----|
| 1 | 111 |     |     |
| 2 | 222 |     |     |
| 3 |     | FFF | 333 |

在第一幅图的基础上按下 ClearCurrentIndex 按钮，此时当前项目会被清除，但所选择的项目未被清除，按下 F2 不会有任何项目被编辑(因为没有当前项目)

|   | 1   | 2   |     |
|---|-----|-----|-----|
| 1 | 111 |     |     |
| 2 | 222 |     |     |
| 3 |     | FFF | 333 |

在第一幅图的基础上按下 ClearSelection 按钮，此时被选择的项目(高亮)会被清除，但当前项目未被清除，按下 F2 会编辑项目 FFF。

|   | 1   | 2   |     |
|---|-----|-----|-----|
| 1 | 111 |     |     |
| 2 | 222 |     |     |
| 3 |     | FFF | 333 |



## 8.5 QAbstractItemDelegate 类(委托基类)与

### QStyleOptionViewItem 类

#### 一、基本原理

- 1、QAbstractItemDelegate 类继承自 QObject,
- 2、委托用于显示视图中的单个项目, 并处理模型数据的编辑。
- 3、若需要以自定义方式渲染项目, 则必须重新实现 paint()和 sizeHint()函数。
- 4、可使用如下两种方法实现自定义的编辑:
  - 1)、方法 1: 创建一个编辑器部件, 并将其设置为项目的编辑器, 此方法必须重新实现 createEditor()函数, 并使用 setEditorData()函数从模型中获取数据用于编辑器, 使用 setModelData 把编辑器的内容写入模型中。
  - 2)、方法 2: 重新实现 editorEvent()函数, 直接处理用户事件。
- 4、QAbstractDelegate 的子类 QItemDelegate 和 QStyleItemDelegate 是 Qt 提供的对 QAbstractDelegate 类的默认实现。

#### 二、QAbstractItemDelegate 类中的函数

注意: QAbstractItemDelegate 类中的函数都是虚函数, 这些函数的参数都附带有必要的信息, 比如对于 paint() 函数的 index 参数, 就是表示需要绘制的模型的索引, 且 index.data()函数包含有来自模型的数据。在重新实现这些虚函数时, 应合理使用这些参数所附带的信息。

- 1、QAbstractItemDelegate(QObject\* *parent* = Q\_NULLPTR); //构造函数

#### 2、纯虚函数(自定义渲染)

- 1)、virtual void paint(QPainter \**painter*, const QStyleOptionViewItem &*option*,  
const QModelIndex &*index*) const = 0; //纯虚函数

若要提供自定义的渲染, 则必须重新实现此函数, 使用 painter 和外观选项 option 来渲染索引 index 所指的项目。若重新实现该函数, 还必须重新实现 sizeHint()函数。

- 2)、virtual QSize sizeHint(const QStyleOptionViewItem &*option*,  
const QModelIndex &*index*) const = 0; //纯虚函数

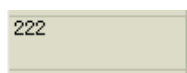
若要提供自定义的渲染, 则必须重新实现此函数, 若重新实现该函数, 还必须重新实现 sizeHint()函数。

#### 3、编辑器

- 3)、virtual QWidget\* createEditor(QWidget\* *parent*, const QStyleOptionViewItem &*option*,  
const QModelIndex &*index*) const; //虚拟的

返回用于编辑索引 index 所指数据项的编辑器, 注意: index 包含正在使用的模型的信息。parent 表示编辑器的父部件, 项目选项由 option 指定。默认实现返回 0, 若要使用自定义的编辑器, 需要重新实现该函数。

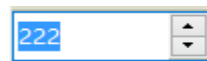
- 4)、virtual void **setEditorData**(QWidget \**editor*, const QModelIndex &*index*) const; //虚拟的  
将编辑器 *editor* 的内容设置为索引 *index* 所指项目的数据。默认实现什么也不做。若  
要使用自定义的编辑器，则需要重新实现此函数。其原理见下图



单元格的初始内容

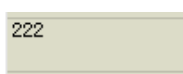


未把编辑器的数据设置  
为项目的数据

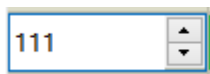


正确的把编辑器的数据  
设置为项目的数据

- 5)、virtual void **setModelData**(QWidget \**editor*, QAbstractItemModel \**model*,  
const QModelIndex &*index*) const; //虚拟的  
将模型 *model* 中由索引 *index* 所指项目的数据设置为编辑器 *editor* 的内容。默认实现  
什么也不做。若要使用自定义的编辑器，则需要重新实现此函数。原理见下图



1、单元格的初始内容



2、把项目的内容修改  
为值 111



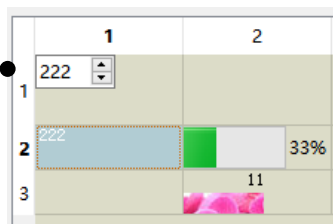
3、未把项目的数据设置为编辑器  
的数据，单元格恢复为初始值



4、把项目的数据正确设置为编辑器的数据后的效果

- 6)、virtual void **updateEditorGeometry**(QWidget \**editor*, const QStyleOptionViewItem &*option*,  
const QModelIndex &*index*) const; //虚拟的  
根据 *option* 中的矩形，更新索引 *index* 所指项目的编辑器的几何尺寸，该函数决定编  
辑器在视图中的位置和大小，因此是比较重要的，默认实现什么也不做。若要使用自  
定义的编辑器，则需要重新实现此函数。原理见下图

未正确设置编辑器的位置和大小  
时，编辑器明显小于单元格的大小，  
若所有单元格都未正确设置编辑器  
的几何尺寸，则有可能所有单元格的  
编辑器都会显示在此位置(左上  
角)



- 7)、virtual void **destroyEditor**(QWidget \**editor*, const QModelIndex &*index*) const; //虚拟的，qt5.0  
当编辑器 *editor* 不再需要编辑索引 *index* 所指的数据项且应该被销毁时调用，也就  
说，该函数用于销毁编辑器 *editor*。默认行为是调用编辑器 *editor* 的 `QObject::deleteLater()`  
函数(该函数用于销毁对象)。重新实现该函数可以避免使用默认行为。

## 4、事件

- 8)、virtual bool **editorEvent**(QEvent \**event*, QAbstractItemModel \**model*,  
const QStyleOptionViewItem &*option*, const QModelIndex &*index*); //虚拟的

在开始编辑项目时，会调用这个函数，event 为触发该编辑的事件，model、index 分别为被编辑项目的模型和索引，option 为渲染项目的选项。即使鼠标事件没有开始编辑该项目，也会被发送给 editorEvent()，比如当在项目上按下鼠标右键试图打开一个上下文菜单时，默认实现返回 false(即什么也没做)。

- 9)、virtual bool **helpEvent**(QHelpEvent \*event, QAbstractItemView \*view, const QStyleOptionViewItem &option, const QModelIndex &index); //虚拟的
- 当发生帮助事件时，会调用这个函数，若委托处理该事件则返回 true，否则返回 false。对于成功处理的 QEvent::ToolTip 和 QEvent::WhatsThis 事件，根据系统的配置可能会显示相关的弹出窗口。

5、信号

- 1)、void **closeEditor**(QWidget \*editor, QAbstractItemDelegate::EndEditHint hint = NoHint); //信号
- 当用户使用编辑器 editor 完成对项目的编辑时，发送此信号。hint 为委托提供了编辑完成后影响模型和视图行为的方式，可以指示组件接下来执行什么操作，比如项 hint 为 EidtNextItem 则视图应使用委托打开下一个项目的编辑器。枚举 EndEditHint 取值见下表

| QAbstractItemDelegate::EndEditHint 枚举   |   |                                    |
|-----------------------------------------|---|------------------------------------|
| 成员                                      | 值 | 说明                                 |
| QAbstractItemDelegate::NoHint           | 0 | 无推荐可执行                             |
| QAbstractItemDelegate::EditNexItem      | 1 | 应在下一个项目上打开编辑器                      |
| QAbstractItemDelegate::EditPreviousItem | 2 | 应在上一个项目上打开编辑器                      |
| QAbstractItemDelegate::SubmitModelCache | 3 | 若模型缓存数据，则应把缓存的数据写入底层的数据存储          |
| QAbstractItemDelegate::RevertModelCache | 4 | 若模型缓存数据，则应放弃缓存的数据并将其替换为底层数据存储中的数据。 |

- 2)、void **commitData**(QWidget \*editor); //信号
- 当编辑器 editor 编辑完数据并且要将其写回模型中时，必须发送此信号。也就是说当数据写回模型时，会发送此信号。
- 3)、void **sizeHintChanged**( const QModelIndex &index); //信号
- 当索引 index 的 sizeHint()变化时，必须发送此信号，视图自动连接到此信号，并根据需要重新布局项目。

四、QStyleOptionViewItem 类

- 1、QStyleOptionViewItem 类继承自 QStyleOption。
- 2、Qt 绘制控件基本原理: Qt 内置的部件的外观几乎都是由 QStyle 类的成员函数进行绘制的，使用这些函数绘制部件时需要向函数提供一些所需绘制图形元素的信息，而这些信息是由 QStyleOption 类进行描述的，QStyleOption 类的不同子类描述了不同图形元素所需的信息，比如 QStyleOptionButton 描述了绘制按钮所需的有关信息等。因此模型/视图结构中的 QStyleOptionViewItem 类描述了绘制数据项所需的有关信息
- 3、QStyleOptionViewItem 类中的函数

`QStyleOptionViewItem();` //构造函数

`QStyleOptionViewItem(const QStyleOptionViewItem& other);` //复制构造函数

4、QStyleOptionViewItem 类中的成员变量

注：项目的装饰(decoration)：通常是指图标，与角色 Qt::DecorationRole 是相对应的。

- 1)、QBrush `backgroundBrush`  
用于绘制项目背景的画刷
- 2)、Qt::CheckState `checkState`  
若项目是可选中的(即枚举成员 ViewItemFeature::HasCheckIndicator 为 true)，若该项目被选中，则为 true，否则为 false。
- 3)、Qt::Alignment `decorationAlignment`  
项目装饰的对齐方式，默认为 Qt::AlignLeft。
- 4)、Position `decorationPosition`  
项目装饰的位置，默认为 Left。枚举 Position 取值如下表

| QStyleOptionViewItem::Position 枚举 |   |      |
|-----------------------------------|---|------|
| 作用：描述项目装饰的位置                      |   |      |
| 成员                                | 值 | 说明   |
| QStyleOptionViewItem::Left        | 0 | 文本左侧 |
| QStyleOptionViewItem::Right       | 1 | 文本右侧 |
| QStyleOptionViewItem::Top         | 2 | 文本上方 |
| QStyleOptionViewItem::Bottom      | 3 | 文本下方 |

- 5)、QSize `decorationSize`  
项目装饰的大小，默认为 QSize(-1,-1)，即无效大小。若要在项目上绘制图标需设置此变量的大小。
- 6)、Qt::Alignment `displayAlignment`  
项目显示值的对齐方式，默认值为 Qt::AlignLeft。通常用于设置与角色 Qt::DisplayRole 对应的值的对齐方式。
- 7)、ViewItemFeatures `features`  
描述项目的特征，即该项目可以包含哪些类型的数据，枚举 ViewItemFeature 取值见下表

| QStyleOptionViewItem::ViewItemFeature 枚举  |      |                                                                       |
|-------------------------------------------|------|-----------------------------------------------------------------------|
| 标志：QStyleOptionViewItem::ViewItemFeatures |      |                                                                       |
| 作用：描述项目的特征                                |      |                                                                       |
| 成员                                        | 值    | 说明                                                                    |
| QStyleOptionViewItem::None                | 0x00 | 正常项目                                                                  |
| QStyleOptionViewItem::WrapText            | 0x01 | 项目的文本可以自行换行                                                           |
| QStyleOptionViewItem::Alternate           | 0x02 | 项目使用 alternateBase 渲染背景，参见 QAbstractItemView::alternatingRowColors 属性 |
| QStyleOptionViewItem::HasCheckIndicator   | 0x04 | 项目具有可选中指示符(即左侧有一个可被选中的勾形符号的方框)                                        |

|                                     |      |                         |
|-------------------------------------|------|-------------------------|
| QStyleOptionViewItem::HasDisplay    | 0x08 | 项目具有 Qt::DisplayRole    |
| QStyleOptionViewItem::HasDecoration | 0x10 | 项目具有 Qt::DecorationRole |

- 8)、QFont **font**  
项目的字体，默认为应用程序的默认字体
- 9)、QIcon **icon**  
绘制在项目中的图标
- 10)、QModelIndex **index**  
需要绘制的模型的索引
- 11)、bool **showDecorationSelected**  
当项目被选择时，是否突出显示装饰。默认为 false。
- 12)、QString **text**  
在项目中绘制的文本。
- 13)、Qt::TextElideMode **textElideMode**  
当项目文本太长时，省略号应出现的位置，默认为 Qt::ElideMiddle，即省略号位于文本中间。
- 14)、ViewItemPosition **viewItemPosition**  
该项目相对于其他项目的位置，ViewItemPosition 枚举见下表

| QStyleOptionViewItem::ViewItemPosition 枚举                |   |                  |
|----------------------------------------------------------|---|------------------|
| 作用：描述项目在一行中的位置，可根据项目的位置不同而绘制项目的不同外形，比如为开始和结束处的项目绘制圆形边缘等。 |   |                  |
| 成员                                                       | 值 | 说明               |
| QStyleOptionViewItem::Invalid                            | 0 | 未知的，应忽略          |
| QStyleOptionViewItem::Beginning                          | 1 | 项目位于行的开头         |
| QStyleOptionViewItem::Middle                             | 2 | 项目位于行的中间         |
| QStyleOptionViewItem::End                                | 3 | 项目位于行的末尾         |
| QStyleOptionViewItem::OnlyOne                            | 4 | 项目是唯一的(即该行只有一项)。 |

- 5、除以上成员变量外，从 QStyleOption 继承而来的成员变量对项目的绘制也有影响，其中有两个比较重要，如下
  - 1)、QRect QStyleOption::rect  
表示绘制的项目的区域，也就是说，图形是在 rect 所指区域中绘制的。默认为空矩形。
  - 2)、QStyle::State QStyleOption::state  
绘制控件时的样式标志。state 用于指示控件是否已启用，是否被选择等状态。

**示例：使用委托绘制自定义的项目**

```
//m.h 文件的内容
#ifndef M_H
#define M_H
#include<QtWidgets>
#include<QDebug>
```

```

//本示例仅重新实现了 QAbstractItemDelegate 类中的两个纯虚函数
class T:public QAbstractItemDelegate{
public: T(QObject *parent = 0):QAbstractItemDelegate(parent){} //构造函数
//重新实现 paint 函数，以绘制每个单元格(项目)
 void paint(QPainter *painter, const QStyleOptionViewItem &pl,
 const QModelIndex &index) const{
//为第 1 行 1 列的项目绘制一个进度条
 if (index.row()==1&&index.column() == 1) {
 int i = index.data().toInt();
 QStyleOptionProgressBar p; //该类用于设置绘制进度条时所需的信息
 p.rect = pl.rect; /*设置绘制进度条的区域(使用 Qt 自动获取的区域)。此项必须设置，
 否则图形不知绘制于何处。*/
 p.minimum = 0; //进度条的最小值
 p.maximum = 100; //进度条的最大值
 p.progress = i; //进度条的当前进度来自模型中的数据
 p.text = QString::number(i) + "%"; //进度条当前显示的值
 p.textVisible = true;
 /*使用 QStyle::drawControl() 成员函数绘制进度条，由此可见，项目上具体需要显示什么，完
 全可以由程序员自行绘制。*/
 QApplication::style()->drawControl(
 QStyle::CE_ProgressBar, //该参数表示绘制进度条
 &p, painter);
 }
//绘制第 2 行 1 列的项目，该项目会绘制一个图标和一个文本
 else if(index.row()==2 && index.column() == 1){
 QStyleOptionViewItem p; //该类用于设置绘制模型/视图的数据项时所需的信息
 p.index=index; //设置需要绘制的项目的索引，此项不是必须设置项
 p.rect=pl.rect; /*绘制项目的区域(使用 Qt 自动获取的区域)，此项必须设置，否则图
 形不知绘制于何处。*/
 //要显示图标，以下选项必须设置
 p.features=QStyleOptionViewItem::HasDecoration //要显示图标，必须包含该特征
 |QStyleOptionViewItem::HasDisplay;
 p.decorationSize=QSize(55, 55); /*设置装饰(通常为图标)的大小，若不设置该选项，则
 因装饰大小为无效大小(-1, -1)而无法绘制。*/
 p.icon=index.data(Qt::DecorationRole).value<QIcon>(); //需要绘制的图标
 //要使项目被选择时高亮显示，需设置以下选项
 p.state=pl.state; //设置项目的样式标志(即项目是否被选择，是否启用等)
 qDebug()<<pl.state; //读者可观察项目样式标志的变化情况
 p.showDecorationSelected=true; //设置为 true。
 if (pl.state & QStyle::State_Selected) //若项目被选择，则高亮绘制其矩形
 painter->fillRect(pl.rect, pl.palette.highlight());
 //其他一些设置
 p.decorationPosition=QStyleOptionViewItem::Bottom; //装饰(图标)显示在文字下方
 p.displayAlignment=Qt::AlignLeft|Qt::AlignHCenter; //设置项目文本的对齐方式
 p.text=index.data().toString(); //设置项目需要显示的文本
 //绘制项目
 QApplication::style()->drawControl(
 QStyle::CE_ItemViewItem, //该参数表示，绘制模型/视图的项目
 &p, painter);
 }
//绘制第 2 行 0 列的项目，本示例将该项目绘制于其他地方(即该项目未位于视图的单元格之内)

```

```

else if(index.row()==2&&index.column() == 0){
 QStyleOptionViewItem p;
 p.features=QStyleOptionViewItem::HasDisplay
 |QStyleOptionViewItem::HasCheckIndicator; //该项目可被选中
 p.rect=QRect(199,144,111,44); //该项目位置于此处
 p.state=pl.state;
 p.showDecorationSelected=true;
 p.text=index.data().toString();
 p.backgroundBrush=QBrush(QColor(111,1,1)); //设置背景色
 QFont f; f.setPixelSize(22); p.font=f; //设置字体
 p.displayAlignment=Qt::AlignLeft|Qt::AlignVCenter; //设置对齐方式
 QApplication::style()->drawControl(QStyle::CE_ItemViewItem, &p, painter);
}
//绘制其他项目
else { QStyleOptionViewItem p;
 p.features=QStyleOptionViewItem::HasDisplay;
 p.index=index; p.rect=pl.rect; p.state=pl.state;
 p.showDecorationSelected=true; /*设置为 true, 否则没有内容的项目被选择时不会
 高亮显示。*/

 p.text=index.data().toString();
 QApplication::style()->drawControl(QStyle::CE_ItemViewItem,&p, painter);
} } //函数 paint 结束
//此函数对本例没有影响, 可返回任意值。
QSize sizeHint(const QStyleOptionViewItem &option,const QModelIndex &index) const
{ return QSize(0,0); }
}; //类 T 结束

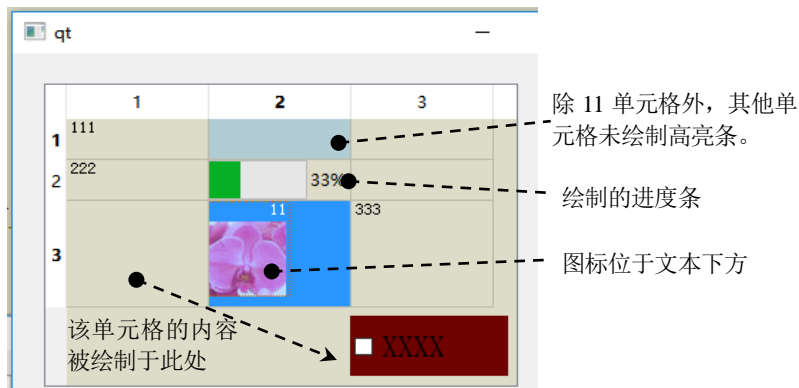
class B:public QWidget{ Q_OBJECT
public: QStandardItemModel *d; QTableView *pv2; T *pt;
 B(QWidget *p=0):QWidget(p) {
 pv2=new QTableView(this); pv2->move(22,22); pv2->resize(333,222);
 d=new QStandardItemModel(3,3,this);
 //向模型中添加数据
 d->setData(d->index(0,0,QModelIndex()),"111",Qt::DisplayRole);
 d->setData(d->index(1,0,QModelIndex()),222);
 d->setData(d->index(1,1,QModelIndex()),33);
 d->setData(d->index(2,0,QModelIndex()),"XXXX");
 d->setData(d->index(2,1,QModelIndex()),QIcon("F:/li.png"),Qt::DecorationRole);
 d->setData(d->index(2,1,QModelIndex()),11);
 d->setData(d->index(2,1,QModelIndex()),11,Qt::ToolTipRole);
 d->setData(d->index(2,2,QModelIndex()),333);
 pv2->setModel(d); //设置模型
 //设置视图的委托为自定义委托
 pt=new T(this); pv2->setItemDelegate(pt); };
#endif // M_H

//m.cpp 文件的内容
#include "m.h"
int main(int argc, char *argv[]){ QApplication aa(argc,argv);
 B w; w.resize(444,355); w.show(); return aa.exec(); }

```



## 运行结果及说明



### 示例：使用自定义的编辑器

//本示例只需把以下代码直接复制到上一示例类 T 之中即可。

//重新实现 createEditor 以便为每个单元格创建各自的编辑器

```
QWidget* createEditor(QWidget* parent, const QStyleOptionViewItem &option,
 const QModelIndex &index) const {
 //创建两种类型的编辑器，以用于不同类型
 QLineEdit *pe=new QLineEdit(parent); QSpinBox *ps=new QSpinBox(parent);
 pe->setObjectName("EEE"); ps->setObjectName("SSS");
 //若单元格的数据是文本或无效数据，则使用 pe 编辑器进行编辑
 if(index.data().type()==QMetaType::QString||index.data().isNull()){
 pe->setFocusPolicy(Qt::TabFocus); return pe; }
 //否则使用 ps 编辑器进行编辑
 else { ps->setMaximum(1000); return ps; }
}
```

//重新实现 setEditorData 以便把单元格的数据读入到编辑器中

```
void setEditorData(QWidget *e, const QModelIndex &index) const{
 //若单元格之前是文本或无效数据，则把模型中的数据读入到 QLineEdit 类型的编辑器中
 if(index.data().type()==QMetaType::QString||index.data().isNull()){
 ((QLineEdit*)e)->setText(index.data().toString()); }
 //否则把模型中的数据读入到 QSpinBox 类型的编辑器中
 else { ((QSpinBox*)e)->setValue(index.data().toInt());}
}
```

//重新实现 setModelData 以便把编辑器中的数据写入到单元格(即模型)

```
void setModelData(QWidget *e, QAbstractItemModel *m, const QModelIndex &index) const{
 //若单元格之前是文本或无效数据，则把 QLineEdit 类型编辑器的数据写入模型
 if(index.data().type()==QMetaType::QString||index.data().isNull()){
 { m->setData(index, ((QLineEdit*)e)->text()); }
 //否则把 QSpinBox 类型编辑器的数据写入模型
 else { m->setData(index, ((QSpinBox*)e)->value());}
}
```

//设置编辑器的几何尺寸

```
void updateEditorGeometry(QWidget *e, const QStyleOptionViewItem &option,
 const QModelIndex &index) const{
 //只需把编辑器的尺寸(位置和大小)设置为单元格的尺寸即可。
```



```

 e->setGeometry(option.rect);
 }
//该函数可以不需重新实现而使用默认的实现方式
void destroyEditor(QWidget *e, const QModelIndex &index) const{
 qDebug()<<e->objectName(); //查看删除的编辑器
 delete e; } //使用 delete 删除编辑器，当然，你也可以不删除编辑器。

```

运行结果及说明



## 8.6 QModelIndex 类

- 1、QModelIndex 类是一个独立的类
- 2、QModelIndex 类在前面章节已使用得比较多了，该类用于管理项目的模型索引，主要在以下几点，
  - 模型索引需要使用 QAbstractItemModel::createIndex()函数来创建。
  - 无效模型索引通常是模型中顶级项目的父索引。
  - 无效模型索引使用 QModelIndex 类的默认构造函数创建。
  - 模型索引应立即使用，然后丢弃，因为在改变模型结构或删除项目之后，模型索引有可能会不再有效。
  - QModelIndex 类提供的索引是一个临时索引，若需要对数据项进行长时间的引用，应使用 QPersistentModelIndex 类创建一个持久模型索引
- 3、QModelIndex 类中的函数
  - 1)、**QModelIndex**() //构造函数，用于创建无效模型索引
  - 2)、int **column**() const;
    - int **row**() const;
    - 以上函数表示，返回模型索引所引用的列号和行号(从 0 开始计算)。
  - 3)、QVariant **data**(int **role** = Qt::DisplayRole) const;
    - 返回由该索引所指项目的角色 **role** 的数据。
  - 4)、Qt::ItemFlags **flags**() const;

获取索引所指项目的项目标志，该函数的作用及 Qt::ItemFlags 枚举详见 QAbstractItemModel::flags() 函数。

5)、quintptr **internalId**() const;

void \***internalPointer**() const;

以上函数分别表示返回该索引所指项目与内部数据结构所关联的 void \* 指针或 quintptr

6)、bool **isValid**() const;

若模型索引有效则返回 true，否则返回 false，有效模型索引具有非负的行号和列号。

7)、const QAbstractItemModel \***model**() const;

返回索引所指项目的模型的指针，注意，返回的是 const 指针。

8)、QModelIndex **parent**() const;

返回模型索引的父索引，若没有父索引，则返回 QModelIndex()。

9)、QModelIndex **sibling**(int *row*, int *column*) const;

返回位于行 row 和列 column 处的兄弟索引，若在此位置没有兄弟，则返回无效的模型索引，其原理详见 QAbstractItemModel::sibling() 函数。

10)、重载的运算符函数有：!=、<、==。

## 8.7 自定义视图示例

### 1、自定义视图基本原则

- 1)、视图需要自行绘制，通常在 `paintEvent()` 函数内完成，所以除了必须实现的纯虚函数外，`paintEvent()` 也应重新实现。另外若需要对单元格进行重新绘制、更新滚动条等，还需要重新实现 `resizeEvent()` 函数。
  - 2)、自定义视图需要完成显示的单元格的大小和位置的计算、单元格轮廓线的绘制、滚动的计算、对单元格的选择作出处理、若有必要还需要绘制标头。
  - 3)、另外需要记住的是视图就是一个 `QFrame`，也就是说直接使用 `show()` 显示视图，那么视图只是一个什么也没有的窗口而已，因此窗口中的内容需要由程序员设计，也就是说你也可以完全不继承 `QAbstractItemView` 类，而子类化 `QFrame` 类来实现视图中内容的绘制，当然这样会失去对 `QAbstractItemView` 类中由 Qt 已实现的内部函数的使用。
- 2、以下示例为最小的自定义视图实现，也就是说以下代码实现的视图功能并不完善。完整的实现需要比较复杂的计算量，以下代码主要是为了让读者明白，视图究竟用来做什么，做了什么。从而加深对模型/视图结构的理解。作为最小实现，本示例只处理单元格轮廓线的绘制、单元格中数据项的绘制，以及简单的选择操作
- 3、本示例的基本规则：使用 `paintEvent()` 函数完成由 `visualRect()` 函数返回的矩形的轮廓线及其数据项的绘制，其中数据项调用委托绘制，

### 示例：简单的自定义视图

//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>

class V:public QAbstractItemView{
public:
//1、以下函数用于计算项目所占据的矩形(即位置和大小)
 QRect visualRect(const QModelIndex &index) const{
 //该函数在初次运行时便会由 Qt 调用，调用次数依模型而定，本例 3*3 的表格模型，
 //该函数会被调用 18 次。参数 index 包含模型的索引，index 会在调用时循环传递。
 //比如对于本例，第一次调用时的索引为(0,0)，第二次为(0,1)，第3次为(0,2)...

 //计算项目的矩形：项目大小始终为(110,33)，位置随索引而不同。
 return QRect(index.column()*110, index.row()*33+20, 110, 33); }

//2、以下函数返回鼠标光标所在位置的项目的索引
 QModelIndex indexAt(const QPoint &point) const{ //该函数在点击鼠标时 Qt 会调用。
 //参数 point 包含了鼠标光标的坐标位置(视图坐标)
 int r=(point.y()-20)/33; //计算光标位于哪一行。
 int c=point.x()/110; //计算光标位于哪一列。
 return model()->index(r,c); } //返回该项目的索引。
};
```

//3、以下两个函数主要用于处理对项目的选择，当选择视图中的项目时，Qt 才会调用他们。

//当不需要选择项目时，以下两个函数可以不用实现。

```
void setSelection(const QRect &rect, QItemSelectionModel::SelectionFlags flags) {
 //参数 rect 包含了所选项目的矩形(位置和大小，使用视图坐标)
 //参数 flags 包含了选择项目时的选择标志。
 int r=(rect.y()-20)/33; //计算选中的是哪一行。
 int c=rect.x()/110; //计算选中的是哪一列。
 selectionModel()->select(model()->index(r,c),flags); } //选择所选中的索引。
```

```
QRegion visualRegionForSelection(const QItemSelection &s) const {
 //此函数用于计算所有被选择的项目占据的区域(即位置和大小)。
 //参数 s 包含了所选择的项目的范围。
 return QRegion(); } }
```

//4、以下函数用于计算视图的滚动，本例是最小实现，不需要滚动，所以不需要实现以下 3 个函数。

```
int horizontalOffset() const{ return 0; }
int verticalOffset() const { return 0; }
void scrollTo(const QModelIndex &index, ScrollHint hint = EnsureVisible) {}
```

//5、本示例不需要隐藏项目，所以以下函数直接返回 0 即可。

```
bool isIndexHidden(const QModelIndex &index) const{ return 0; }
```

//6、以下函数用于处理键盘按键(比如按下左键应返回左侧的项目索引等)，本示例不处理键盘按键，  
//所以不需要实现。

```
QModelIndex moveCursor(CursorAction cursorAction ,Qt::KeyboardModifiers modifiers){
 return QModelIndex(); } }
```

//7、以下函数是核心，用于绘制视图的外观，也就是说若没有以下函数，则视图什么也不会显示。

```
void paintEvent(QPaintEvent *e){
 QPainter pt(viewport()); //在视口上绘制图形
 //使用 QAbstractItemView::viewOptions() 获取需要绘制的图形的信息(此步骤比较重要)
 QStyleOptionViewItem po=viewOptions();
 //循环遍历模型的大小。
 for(int r=0;r<model()->rowCount();r++){
 for(int c=0;c<model()->columnCount();c++){
 QModelIndex i=model()->index(r,c);
 QRect rect=visualRect(i); //获取索引 i 所指项目的矩形(位置和大小)
 po.rect=visualRect(i);
 //处理项目被选择的情形
 if(selectionModel()->isSelected(i)){ po.state |= QStyle::State_Selected;}
 //使用代理绘制数据项(即项目)，这里也可使用自定义的代理(若已添加)
 itemDelegate()->paint(&pt,po,i);
 //以下代码用于绘制视图单元格的轮廓线。
 pt.save();
 pt.setPen(QPen(QColor(111,1,1))); //创建画笔。
 pt.drawLine(rect.bottomLeft(),rect.bottomRight());
 pt.drawLine(rect.bottomRight(),rect.topRight());
 pt.restore(); } //for 结束
 } //paintEvent 结束
};
```

```
class B:public QWidget{ Q_OBJECT
```

```

public: QStandardItemModel *d; V *pv;
B(QWidget *p=0):QWidget(p) {
 d=new QStandardItemModel(3,3);
 V* pv=new V; //使用自定义的视图
 //向模型中添加数据
 d->setData(d->index(0,0), "AAA"); d->setData(d->index(0,1), "BBB");
 d->setData(d->index(1,0), "CCC"); d->setData(d->index(1,2), "DDD");
 d->setData(d->index(1,1), "EEE"); d->setData(d->index(2,0), "FFF");
 d->setData(d->index(1,1), QIcon("F:/li.png"), Qt::DecorationRole);

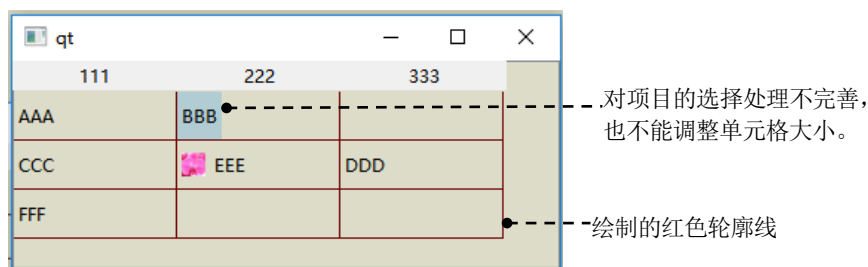
 //向视图中添加标签,以用于视图的标头,以下代码主要演示视图还可以像使用一个 QFrame 那样使用。
 QLabel *pp=new QLabel("111", pv);
 pp->setAutoFillBackground(1); //使标签不透明。
 pp->setAlignment(Qt::AlignCenter); pp->resize(111,20);
 QLabel *ppl=new QLabel("222", pv); ppl->setAutoFillBackground(1);
 ppl->setAlignment(Qt::AlignCenter); ppl->resize(111,20); ppl->move(111,0);
 QLabel *pp2=new QLabel("333", pv); pp2->setAutoFillBackground(1);
 pp2->setAlignment(Qt::AlignCenter); pp2->resize(111,20); pp2->move(222,0);

 //pv->setItemDelegate(pt); //也可以添加自定义代理,以使用自定义代理绘制数据项。
 pv->setModel(d); pv->resize(333,222); pv->show(); }
};
#endif // M_H

//m.cpp 文件内容
#include "m.h"
int main(int argc, char *argv[]) { QApplication app(argc,argv); B w; return app.exec(); }

```

运行结果及说明



作者：黄邦勇帅(原名：黄勇)

2018-6-8

## 第 2 篇 Qt 实现的标准模型/视图框架相关类

### 8.8 QStandardItemModel 类及 QStandardItem 类

本小节使用项目表示模型中的位置，数据项表示该位置的数据，比如位于(3,2)处项目的数据项为 xx

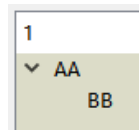
#### 一、QStandardItemModel 基本原理

- 1、QStandardItemModel 继承自 QAbstractItemModel。QStandardItem 是一个独立的类。
- 2、QStandardItemModel 类是 Qt 实现的标准模型，该类的数据项(项目)由类 QStandardItem 类描述
- 3、QStandardItemModel 类实现了 3 种结构的模型，即列表、树、表格。
- 4、创建表格模型的方法为：向构造函数传递表格的大小(行数和列数)以创建一个表格模型，然后使用 setItem()或 setData()向模型中添加数据。代码如下

```
QStandardItemModel d(3, 3); //创建一个 3 行 3 列的表格模型
d.setData(d.index(0, 0, QModelIndex()), 222); //向表格的第 0 行 0 列添加数据 222。
//或使用 setItem() 函数向表格添加数据
QStandardItem *pm = new QStandardItem("AA"); //创建一个数据项
d.setItem(0, 0, pm); //把数据项添加到表格的 0 行 0 列。
```

- 5、创建列表或树形结构的方法为：创建一个空的 QStandardItemModel，然后使用 appendRow() 函数将数据项添加到模型中。创建树形结构的代码如下(效果见右图)

```
QTreeView *pv2=new QTreeView //创建树视图
QStandardItemModel *d=new QStandardItemModel(this); //创建模型
QStandardItem *pr = d->invisibleRootItem(); //创建根数据项
QStandardItem *pm = new QStandardItem("AA"); //创建数据项
QStandardItem *pml = new QStandardItem("BB");
pr->appendRow(pm); //把 pm 添加为 pr 的子节点
pm->appendRow(pml); //把 pml 添加为 pm 的子节点
pv2->setModel(d);
pv2->show();
```



- 6、注意：在使用 QAbstractItemModel::setData()或插入函数 QStandardItemModel::insertXXX() 添加数据时，若指定的位置不存在，则可能需要使用相应的函数(比如 setRowCount())先增加模型的行数和列数，才能向模型中正确插入数据。因此建议使用 appendXXX()函数或 QStandardItemModel::setItem()函数向模型中添加数据，因为这两个函数都会自动增加模型的行数和列数。

#### 二、QStandardItemModel 类中的属性和函数(重新实现的基类函数未重复列出)

注：对 QAbstractItemModel 类中重新实现的函数未列出

##### 1、属性和构造函数

1)、**sortRole**: int

访问函数: int sortRole() const; void setSortRole(int);

描述用于排序时使用的角色,比如可设置为按照 Qt::ToolTipRole 角色的数据进行排序,默认为 Qt::DisplayRole

- 2)、`QStandardItemModel(QObject *parent = Q_NULLPTR);` //构造函数  
`QStandardItemModel(int rows, int columns, QObject *parent = Q_NULLPTR);`

## 2、追加、插入、添加、移除数据项

- 3)、void `setItem`(int row, int column, QStandardItem \*item);

void `setItem`(int row, QStandardItem \*item);

把位于(row, column)处项目的数据项设置为 item。若有必要会增加模型的行数和列数以适应数据项,若给定的位置已存在数据项,则之前的数据项被删除。设置数据项后,该模型获取该数据项的所有权。

- 4)、void `appendColumn`(const QList<QStandardItem\*> &items);

void `appendRow`(const QList<QStandardItem\*> &items);

以上函数用于向模型中的列或行追加数据项 item。

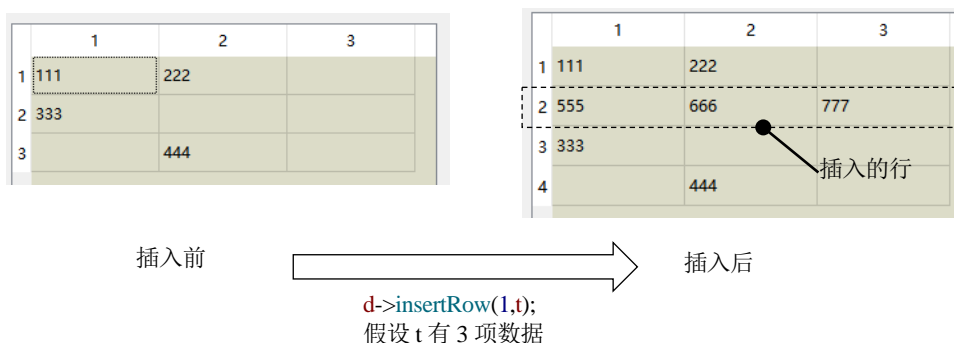
- 5)、void `insertColumn`(int column, const QList<QStandardItem\*> &items);

bool `insertColumn`(int column, const QModelIndex &parent = QModelIndex());

void `insertRow`(int row, const QList<QStandardItem\*> &items);

bool `insertRow`(int row, const QModelIndex &parent = QModelIndex());

以上函数用于向模型插入一行或一列,仅仅只能插入一行或一列,也就是说,若插入行,不会因为数据项 items 有多项而插入多行,而是把数据项 items 按水平方向填充列(原理见下图),参数 column/row 表示需要插入的列/行的位置, parent 表示插入位置处项目的父项目。若插入成功,有返回值的函数则返回 true,否则返回 false。注意 `QAbstractItemModel::insertColumns()`和 `QAbstractItemModel::insertRows()`也可用于向模型插入行或列。



- 6)、void `appendRow`(QStandardItem\* item);

void `insertRow`(int row, QStandardItem \*item);

以上函数用于在只有一列的列表或树时,提供便捷的追加或插入单个新数据项的方法。

- 7)、QList<QStandardItem\*> `takeColumn`(int column);

QList<QStandardItem\*> `takeRow`(int row);

以上函数表示,移除(而不删除)列 column 或行 row 处的数据项,并返回指向已移除数

据项的指针列表,若列或行中的数据项未设置,则对应的指针将为 0。调用以上函数后,模型释放对数据项的所有权。

8)、`QStandardItem *takeItem(int row, int column = 0);`

移除(而不删除)位于(row,column)处的数据项,并返回指向该数据项的指针,调用该函数后,模型释放对该数据项的所有权。

### 3、标头

以下函数需要注意与 `QAbstractItemModel::setHeaderData()` 函数的区别,该函数只能为已存在的标头设置标签,但以下函数可能会增加数据项。

9)、`void setHorizontalHeaderItem(int column, QStandardItem *item);`

`void setVerticalHeaderItem(int row, QStandardItem *item);`

`QStandardItem* horizontalHeaderItem(int column) const;`

`QStandardItem *verticalHeaderItem(int row) const;`

以上函数用于设置和获取水平或垂直标头数据项,若有必要会增加模型的数据项,若设置的列或行位于已存在的标头位置,则之前的标头数据项被删除。设置标头数据项后,该模型获取该数据项的所有权。原理见下图

|   | 1   | 2 | 3 |
|---|-----|---|---|
| 1 | 111 |   |   |

初始状态

|   | 1   | 2 | 3 | 4 | AAA |
|---|-----|---|---|---|-----|
| 1 | 111 |   |   |   |     |

增加了两列

```
QStandardItem *pml = new QStandardItem("AAA");
d->setHorizontalHeaderItem(4, pml);
```

|   | 1   | AAA | 3 |
|---|-----|-----|---|
| 1 | 111 |     |   |

```
QStandardItem *pml = new QStandardItem("AAA");
d->setHorizontalHeaderItem(1, pml);
```

10)、`void setHorizontalHeaderLabels(const QStringList &labels);`

`void setVerticalHeaderLabels(const QStringList &labels);`

以上函数用于设置水平/垂直标头的标签文本,若有必要会增加模型的数据项。原理见下图

|   | 1   | 2 | 3 |
|---|-----|---|---|
| 1 | 111 |   |   |

初始状态

|   | 111 | 222 | 333 | 444 |
|---|-----|-----|-----|-----|
| 1 | 111 |     |     |     |

增加了 1 列

```
QStringList s;
s << "111" << "222" << "333" << "444";
d->setHorizontalHeaderLabels(s);
```

11)、`QStandardItem *takeHorizontalHeaderItem(int column);`

`QStandardItem *takeVerticalHeaderItem(int row);`

移除(而不删除)水平或垂直标头数据项,并返回指向该数据项的指针,调用以上函数



后，模型释放对该数据项的所有权。

#### 4、获取项目的信息

12)、QStandardItem\* **item**(int *row*, int *column* = 0) const;

返回(row, column)处的数据项(若已设置)，否则返回 0。

13)、QModelIndex **indexFromItem**(const QStandardItem\* *item*) const;

返回与数据项 *item* 相关联的模型索引。QStandardItem::index()函数相当于调用此函数。

14)、QStandardItem\* **itemFromIndex**(const QModelIndex &*index*) const;

返回指向与索引 *index* 相关联的 QStandardItem 的指针，若索引 *index* 是无效索引，则返回 0。

#### 5、其他函数及信号

15)、QStandardItem\* **invisibleRootItem**() const;

返回模型的不可见根数据项。从该数据项上检索到的索引是无效的。

16)、void **clear**();

删除模型中的所有数据项(包括标头数据项)，并将行数和列数设置为 0。

17)、void **setColumnCount**(int *columns*);

void **setRowCount**(int *rows*);

把模型的列或行的数量设置为 *columns* 或 *rows*。若设置的列/行数小于当前的列/行数，则删除多余的列/行中的数据。

18)、QList<QStandardItem\*> **findItems**(const QString &*text*, Qt::MatchFlags *flags* = Qt::MatchExactly,

int *column* = 0) const;

查找模型中的数据项，*text* 表示需要查找的文本，*flags* 表示匹配的方式(比如是否区分大小写等)，*column* 表示在该列中查找。Qt::MatchFlag 枚举见第 4 章公共枚举部分

19)、void **setItemRoleNames**(const QHash<int, QByteArray> &*roleNames*);

设置项目角色的名称为 *roleNames*，此函数与 QAbstractItemModel::roleNames()函数相对应。

20)、const QStandardItem\* **itemPrototype**() const;

void **setItemPrototype**(const QStandardItem\* *item*);

获取和设置模型的项目原型。项目原型依靠 QStandardItem::clone() 函数充当 QStandardItem 工厂，当需要按需构造新项目时(比如，当视图或委托调用 setData()时)，使用项目原型作为项目工厂。

21)、void **itemChanged**(QStandardItem\* *item*); //信号

当数据项 *item* 的数据发生变化时，发送此信号。

### 三、QStandardItem 基本原理

1、QStandardItem 数据项是一个二维的子项目表(即 *n* 行 *m* 列的表)，这使得数据项可以以层次结构构建，典型的层次结构是树形结构。注意：此处说明了数据项不是一个单独的元素，而是由多个子项目组成的。

2、可使用以下函数设置子表的维数(即行和列)

- `setRowCount()`和 `setColumnCount()`设置行数和列数,
  - 使用 `setChild()`可把数据项放置于子表中,
  - `insertRow()`和 `insertColumn`、`appendRow()`和 `appendColumn()`可向子表中插入或追加行或列。
  - 删除子表的行或列使用函数 `removeRow()`、`takeRow()`、`removeColumn()`或 `takeColumn()`
- 3、注意：因为数据项具有层次结构，模型的项目也具有层次结构，因此在使用时应设计好模型、视图和数据项三者之间的结构，若设计不合理，可能会无法正确显示。若数据项 `QStandardItem` 组织为树形结构，则 `QStandardItemModel` 和树形视图默认只会使用 1 列来显示数据项，因此若数据项的某个节点有多余 1 列的数据需要显示，则需要增加模型 `QStandardItemModel` 的列数，否则数据将不会被显示。

## 四、QStandardItem 类中的函数

### 1、QStandardItem()

`QStandardItem`(const `QString` &*text*)

`QStandardItem`(const `QIcon` &*icon*, const `QString` &*text*) //使用图标和文本构造一个数据项

`QStandardItem`(int *rows*, int *columns* = 1) //构造一个有 *rows* 行, *columns* 列的子项目的数据项

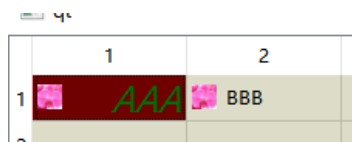
### 2、以下函数主要用于设置单个数据项的外观，比较简单，因此仅列出其原型

| 数据项的外观                                                                          |                                                          |                |
|---------------------------------------------------------------------------------|----------------------------------------------------------|----------------|
| void <b>setText</b> (const <code>QString</code> & <i>text</i> )                 | <code>QString</code> <b>text</b> () const                | 文本             |
| void <b>setTextAlignment</b><br>( <code>Qt::Alignment</code> <i>alignment</i> ) | <code>Qt::Alignment</code> <b>textAlignment</b> () const | 文本对齐方式         |
| void <b>setIcon</b> (const <code>QIcon</code> & <i>icon</i> )                   | <code>QIcon</code> <b>icon</b> () const                  | 图标             |
| void <b>setFont</b> (const <code>QFont</code> & <i>font</i> )                   | <code>QFont</code> <b>font</b> () const                  | 字体             |
| void <b>setBackground</b> (const <code>QBrush</code> & <i>brush</i> )           | <code>QBrush</code> <b>background</b> () const           | 背景色            |
| void <b>setForeground</b> (const <code>QBrush</code> & <i>brush</i> )           | <code>QBrush</code> <b>foreground</b> () const           | 前景色(如文字颜色)     |
| void <b>setToolTip</b> (const <code>QString</code> & <i>toolTip</i> )           | <code>QString</code> <b>toolTip</b> () const             | 工具提示           |
| void <b>setStatusTip</b> (const <code>QString</code> & <i>statusTip</i> )       | <code>QString</code> <b>statusTip</b> () const           | 状态提示           |
| void <b>setWhatsThis</b> (<br>const <code>QString</code> & <i>whatsThis</i> )   | <code>QString</code> <b>whatsThis</b> () const           | what's this 帮助 |

virtual void **setData**(const `QVariant` &*value*, int *role* = `Qt::UserRole` + 1)  
virtual `QVariant` **data**(int *role* = `Qt::UserRole` + 1) const  
综合设置上述选项，其使用方法同 `QAbstractItemModel` 类中的对应函数，另外除以上的数据外，其他数据(比如整型、浮点型数据等)也需要使用此函数进行设置。

### 示例：设置单个数据项(效果见图)

```
QTableView *pv2=new QTableView;
QStandardItemModel *d=new QStandardItemModel(3,3);
QStandardItem *ps=new QStandardItem;
QFont f; f.setPixelSize(22); f.setItalic(true);
ps->setFont(f);
ps->setIcon(QIcon("F:/li.png"));
ps->setText("AAA");
ps->setTextAlignment(Qt::AlignRight);
ps->setToolTip("xxx");
ps->setBackground(QBrush(QColor(111,1,1)));
```



```

ps->setForeground(QBrush(QColor(1, 111, 1)));

QStandardItem *ps1=new QStandardItem;
ps1->setData("BBB", Qt::DisplayRole);
ps1->setData(QIcon("F:/li.png"), Qt::DecorationRole);
d->setItem(0, 0, ps);
d->setItem(0, 1, ps1);
pv2->setModel(d); //该设置会同时清除 pv2 之前的选择模型
pv2->show();

```

3、以下函数用于设置数据项的状态(如选中、启用等)

| 状态                                                                                                                                                              |                                           |                               |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------|-------------------------------|
| 注：要启用三态(tristate)需要启用项目的可选中状态，若三态是由 setAutoTristate()启用的，则会由 QTreeWidgetItem 自动控制(注意：QTreeView 视图不会自动控制)，即当用户在部分选中其子项时，父项呈现部分选中状态，若子项全都选中，则父项选中，若子项都未选中，则父项未选中。 |                                           |                               |
| void <b>setCheckable</b> (bool <i>checkable</i> )                                                                                                               | bool <b>isCheckable</b> () const          | 是否启用可选中状态                     |
| void <b>setCheckState</b> (Qt::CheckState <i>state</i> )                                                                                                        | Qt::CheckState <b>checkState</b> () const | 选中状态                          |
| void <b>setAutoTristate</b> (bool <i>tristate</i> )                                                                                                             | bool <b>isAutoTristate</b> () const       | 是否启用三态并由 QTreeWidgetItem 自动控制 |
| void <b>setUserTristate</b> (bool <i>tristate</i> )                                                                                                             | bool <b>isUserTristate</b> () const       | 是否启用三态并由用户控制                  |
| void <b>setSelectable</b> (bool <i>selectable</i> )                                                                                                             | bool <b>isSelectable</b> () const         | 是否可被选择                        |
| void <b>setEditable</b> (bool <i>editable</i> )                                                                                                                 | bool <b>isEditable</b> () const           | 是否可被编辑                        |
| void <b>setEnabled</b> (bool <i>enabled</i> )                                                                                                                   | bool <b>isEnabled</b> () const            | 是否启用该项目                       |
| void <b>setSizeHint</b> (const QSize & <i>size</i> )                                                                                                            | QSize <b>sizeHint</b> () const            | 大小提示                          |

| 拖动                                                    |                                     |                      |
|-------------------------------------------------------|-------------------------------------|----------------------|
| 注：要使用拖动功能，需使视图开启拖动，详见 QAbstractItemModel 类有关拖动的讲解     |                                     |                      |
| void <b>setFlags</b> (Qt::ItemFlags <i>flags</i> )    | Qt::ItemFlags <b>flags</b> () const | 见 QAbstractItemModel |
| void <b>setDragEnabled</b> (bool <i>dragEnabled</i> ) | bool <b>isDragEnabled</b> () const  | 是否可被拖动               |
| void <b>setDropEnabled</b> (bool <i>dropEnabled</i> ) | bool <b>isDropEnabled</b> () const  | 是否可接受放置              |

4、以下函数用于设置数据项中的子项目(注：应使用对应的树形视图进行显示)

- 1)、void **appendColumn**(const QList<QStandardItem\*> &*items*)  
void **appendRow**(const QList<QStandardItem\*> &*items*)  
void **appendRows**(const QList<QStandardItem\*> &*items*)  
void **appendRow**(QStandardItem \**item*)
- 2)、void **insertColumn**(int *column*, const QList<QStandardItem\*> &*items*)  
void **insertColumns**(int *column*, int *count*) //从列 column 开始插入 count 列空列  
void **insertRow**(int *row*, const QList<QStandardItem\*> &*items*)  
void **insertRow**(int *row*, QStandardItem \**item*)  
void **insertRows**(int *row*, const QList<QStandardItem\*> &*items*)  
void **insertRows**(int *row*, int *count*) //从行 row 开始插入 count 行空行

说明：以行为例，insertRow() 只会插入一行，因此插入的数据项只会位于该项，而 insertRows() 函数(注意：多一个 s 字母)用于插入多行，因此插入的数据项会分别填充这些新行，追加函数原理相同，还应注意的是，在插入或追加项目时，若插入或追加的位置不存在，则需要增加相应的列数或行数，才能使数据正确插入并显示，原理见以下示例

### 插入示例：

```
QTreeView *pv2=new QTreeView(this);
QStandardItemModel *d=new QStandardItemModel;
QStandardItem *ps=d->invisibleRootItem();
QStandardItem *ps1=new QStandardItem("111");
QStandardItem *ps3=new QStandardItem("333");
QStandardItem *ps5=new QStandardItem("555");
QStandardItem *ps6=new QStandardItem("666");
QStandardItem *ps7=new QStandardItem("777");
QList<QStandardItem*> t; t<<ps5<<ps6<<ps7;
ps->appendRow(ps1); ps1->appendRow(ps3);
//以下代码必不可少，用于增加列数以正确显示数据项 666 和 777。
//否则数据项 666 和 777 将不会正确显示，注意，以下代码
//增加的是模型的列，而不是数据项的列。
d->setColumnCount(3);
插入方案 1:
ps1->insertRow(1, t); //在第 1 行插入行(仅能插入 1 行)
插入方案 2:
//以下代码不能成功插入，因为父节点 111 之下第 2 行之前的行不存在。
//ps1->insertRows(2,t);
插入方案 3:
ps1->insertRows(1, t); //在第 1 行插入多行，行数视 t 而定
插入方案 4:
ps1->insertRows(2, 2); //在第 2 行处插入 2 行空行，此步必不可
 //少，否则下一行代码插入的行将无法显示。
ps1->insertRow(2, t);
```

|       |
|-------|
| 1     |
| ▼ 111 |
| 333   |

插入前

增加的两列模型

|       |     |     |
|-------|-----|-----|
| 1     | 2   | 3   |
| ▼ 111 |     |     |
| 333   |     |     |
| 555   | 666 | 777 |

方案 1

|       |   |   |
|-------|---|---|
| 1     | 2 | 3 |
| ▼ 111 |   |   |
| 333   |   |   |
| 555   |   |   |
| 666   |   |   |
| 777   |   |   |

方案 3

首先在此处插入两行空行，然后再在此处插入一行，因此之前的两行空行下移

|       |     |     |
|-------|-----|-----|
| 1     | 2   | 3   |
| ▼ 111 |     |     |
| 333   |     |     |
|       |     |     |
| 555   | 666 | 777 |
|       |     |     |

方案 4

### 3)、void setChild(int row, int column, QStandardItem \*item)

void setChild(int row, QStandardItem \*item)

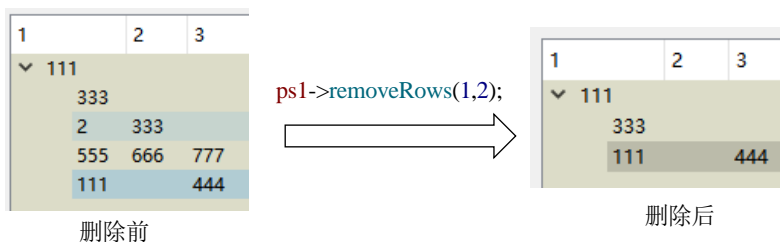
把(row, column)处子项目的数据项设置为 item。该数据项(父数据项)获得 item 的所有权，该函数会增加行数和列数，若 item 为空指针，将删除该位置处的数据项。

### 4)、void setColumnCount(int columns); //设置列的数量 columns

void setRowCount(int rows); //设置行的数量为 rows

### 5、移除或删除数据项中的子项

- 5)、void **removeColumn**(int *column*) //删除列 *column* 处的数据项  
 void **removeColumns**(int *column*, int *count*) //删除从列 *column* 开始的 *count* 列(原理见下图)  
 void **removeRow**(int *row*)  
 void **removeRows**(int *row*, int *count*)
- 6)、QList<QStandardItem\*> **takeColumn**(int *column*) //移除而不删除列 *column* 处的数据项  
 QList<QStandardItem\*> **takeRow**(int *row*)
- 7)、QStandardItem\* **takeChild**(int *row*, int *column* = 0)  
 移除而不删除(row,column)处的数据项, 该函数不会影响子表的维数



## 6、获取有关子项目的信息

- 8)、int **column**() const  
 int **row**() const //返回该数据项所在父项中的行位置
- 9)、int **columnCount**() const  
 int **rowCount**() const //返回该数据项包含的子项列的数量。
- 10)、QStandardItem\* **child**(int *row*, int *column* = 0) const //返回位置(row,column)处的子项目
- 11)、bool **hasChildren**() const //若该项有子项则返回 true
- 12)、QStandardItem\* **parent**() const //返回该项的父项, 若没有父项(或顶级父项)则返回 0。
- 13)、QModelIndex **index**() const //返回与该数据项关联的模型索引, 若未与索引关联, 则返回无效索引。
- 14)、QStandardItemModel\* **model**() const //返回该项目所属的 QStandardItemModel 模型

## 7、排序规则

在进行排序时, 数据项的比较规则由 **operator < ()** 函数定义, 该函数默认实现使用 **QStandardItemModel::sortRole** 属性设置的角色数据项进行比较, 否则使用 **Qt::DisplayRole** 角色数据项进行比较。若想要实现自定义的排序规则, 可以重新实现 **operator < ()** 函数。

- 15)、void **sortChildren**(int *column*, Qt::SortOrder *order* = Qt::AscendingOrder)  
 virtual bool **operator<**(const QStandardItem &*other*) const //虚拟的  
 使用给定顺序(升序或降序)对列 *column* 进行排序, 该函数递归执行, 因此会对其子项、孙子项等等进行排序。

## 8、其他

- 16)、virtual int **type**() const //虚拟的  
 返回该数据项的类型, 返回的类型主要用于区分是否是自定义类型, 可以重新实现此函数以使用自定义的类型。返回的值应是枚举 **ItemType** 的成员所对应的整数值, 见下表

|                                   |
|-----------------------------------|
| <b>QStandardItem::ItemType 枚举</b> |
|-----------------------------------|

| 成员                      | 值    | 说明                       |
|-------------------------|------|--------------------------|
| QStandardItem::Type     | 0    | 标准数据项的默认类型               |
| QStandardItem::UserType | 1000 | 自定义类型的最小值，小于此值的值由 Qt 保留。 |

17)、virtual void **read**(QDataStream &*in*) //虚拟的

virtual void **write**(QDataStream &*out*) const //虚拟的

将数据项读取或写入流，不含子项，且只读取或写入数据项的数据和标志。

18)、virtual QStandardItem \* **clone**() const //虚拟的

返回该数据项的副本(不含子项目)，若希望能够按需创建自定义数据项类型的实例，可重新实现此函数向 QStandardItemModel 提供一个工厂，该函数通常与 QStandardItemModel 类中的 setItemPrototype() 和 itemPrototype 及 QStandardItem::operator = ()一起使用。

## 8.9 QAbstractListModel 类、QAbstractTableModel 类 及 QStringListModel 类

### 一、QAbstractListModel 类、QAbstractTableModel 类

- 1、QAbstractListModel 类、QAbstractTableModel 类继承自 QAbstractItemModel 这两个类同样是抽象类，且只部分实现了其父类的虚函数，因此使用这两个抽象类与使用其父类类似。通常使用的是他们的子类，QAbstractTableModel 类的子类与数据库有关，不属于本文讲解内容，因此本小节着重讲解 QAbstractListModel 类的子类 QStringListModel 类。

### 二、QStringListModel 类(字符串列表模型)

- 1、该类提供了一个由视图使用的字符串列表模型，可用于需要在视图部件中显示字符串的情况，比如用于 QListView 或 QComboBox 类。QStringListModel 模型将字符串列表中的数据表示为一列和等于列表中项目数行数的列表模型。

- 2、创建 QStringListModel 模型的方法

可使用现有的字符串列表直接构建一个 QStringListModel 模型，也可在稍后使用 setStringList() 设置其内容，还可使用 insertRows() 向模型中插入数据。代码如下：

```
QStringList s;
s<<"AAA"<<"BBB"<<"CCC";
QStringListModel *m = new QStringListModel(s);
或
QStringListModel *ml= new QStringListModel;
ml-> setStringList(s);
```

- 3、QStringListModel 类中的函数

- 1)、QStringListModel(QObject \*parent = Q\_NULLPTR) //构造函数

QStringListModel(const QStringList &strings, QObject \*parent = Q\_NULLPTR)

- 2)、void setStringList(const QStringList &strings)

把模型内部的字符串列表(即该模型存储数据的变量)设置为 strings，该模型将通知视图，基础数据已更改。

- 3)、QStringList stringList() const //返回存储数据的字符串列表。

以下函数为重新实现的 QAbstractItemModel 类中的相应函数(这些函数会使用到)。

- 4)、virtual bool setData(const QModelIndex &index, const QVariant &value, int role = Qt::EditRole)

把索引 index 处指定角色 role 所关联的数据设置为 value，若数据项被改变则发送 QAbstractItemModel::dataChanged() 信号。

- 5)、virtual QVariant data(const QModelIndex &index, int role = Qt::DisplayRole) const

返回索引 index 处指定角色 role 所关联的数据。若索引无效，则返回一个无效的 QVariant。

- 6)、virtual Qt::ItemFlags flags(const QModelIndex &index) const

返回索引 index 所指项目的标志，标志详见 QAbstractItemModel 类。

- 7)、virtual bool insertRows(int row, int count, const QModelIndex &parent = QModelIndex())



virtual bool **removeRows**(int **row**, int **count**, const QModelIndex &**parent** = QModelIndex())

以上函数表示从行 row 开始插入或删除 count 行，参数 parent 是可选的，仅用于与 QAbstractItemModel 保持一致，默认为空索引，表示在模型的顶层插入行。

8)、virtual int **rowCount**(const QModelIndex &**parent** = QModelIndex()) const

返回模型中的行数，该值对应于模型内部字符串列中的项目数量。因为该模型是列表，所有 parent 应是一个无效索引，若 parent 是一个有效索引，则该函数将返回 0。

9)、virtual QModelIndex **sibling**(int **row**, int **column**, const QModelIndex &**idx**) const

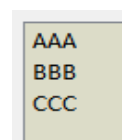
virtual void **sort**(int **column**, Qt::SortOrder **order** = Qt::AscendingOrder)

virtual Qt::DropActions **supportedDropActions**() const

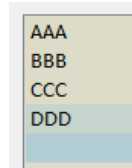
以上函数详见父类 QAbstractItemModel 对应函数。

### 示例：QStringListModel 的使用

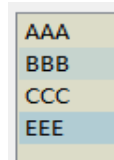
```
QListView *pv2=new QListView(this);
QStringListModel *d1=new QStringListModel;
QStringList s; s<<"AAA"<<"BBB"<<"CCC";
//初始状态：见右图
//把模型 d1 的内部数据设置为列表 s
d1->setStringList(s);
//插入行，见右图
//在第 3 行处增加 2 行，必须增加行数，
//否则 setData() 函数无法把数据添加到新位置
d1->insertRows(3, 2);
//setData() 函数也会修改 d1 的内部数据。
d1->setData(d1->index(3, 0), "DDD", Qt::DisplayRole);
//重置内部数据(最终效果)，见右图
s<<"EEE";
//重新设置 d1 的内部数据。注意：列表 s 和模型的内部数据
//是两个相互独立的部分，s 并不是模型的内部数据。
d1->setStringList(s);
pv2->setModel(d1); //该设置会同时清除 pv2 之前的选择模型
pv2->show();
//也可将 QStringListModel 模型放于 QComboBox 中，代码如下，见右图
QComboBox *pc=new QComboBox(this);
pc->setModel(d1);
```



初 始  
状态



插入行  
之后



重置内部  
数据之后



将模型用  
于列表



## 8.10 QFileSystemModel 类

(目录模型或文件系统模型)

### 一、基本规则

- 1、QFileSystemModel 类继承自 QAbstractItemModel，该类替代 QDirModel 类，这是该类称为目录模型的原因。注：QDirModel 类已过时。
- 2、QFileSystemModel 类提供了一个本地文件系统的数据库模型，该类提供了对本地文件系统的访问，比如可以重命名、删除文件或目录等，该类可以与合适的显示部件一起使用，通常与 QTreeView 一起使用，当然也可与其他视图使用(比如 QListView 等)。
- 3、QFileSystemModel 类的数据来自于本地文件系统。
- 4、在调用 setRootPath()之前，QFileSystemModel 不会获取任何文件或目录，这样可避免在文件系统上的不必要的查询，也就是说，文件系统模型的数据需要调用 setRootPath()函数才会被读入到模型。
- 5、与 QDirModel 不同，QFileSystemModel 使用单独的线程来填充自己，因此在查询文件系统时不会导致主线程挂起。
- 6、直到模型填充目录，调用 rowCount()将返回 0
- 7、fileInfo()、isDir()、fileName()、filePath 等函数提供了关于底层文件和目录的信息，使用 mkdir()函数可以创建目录，rmdir()函数可以删除目录
- 8、使用 QFileSystemModel 类的示例

```
QTreeView *pt = new QTreeView;
QFileSystemModel *dl;
//读入数据到模型，目录路径可以任意，
//并不影响结果。
dl->setRootPath("");
pt->setModel(dl);
pt->show();
```



| Name       | Size | Type  | Date Modified   |
|------------|------|-------|-----------------|
| C:         |      | Drive | 2018.5.31 12:21 |
| 新加卷 (D:)   |      | Drive | 2018.6.4 0:05   |
| 游戏 (E:)    |      | Drive | 2018.5.31 12:21 |
| 音乐 (F:)    |      | Drive | 2018.5.31 12:21 |
| 不可能忘却 (G:) |      | Drive | 2018.6.3 23:40  |
| 飞来横福 (H:)  |      | Drive | 2018.5.31 12:21 |
| 中流砥柱 (I:)  |      | Drive | 2018.5.31 12:21 |
| 新加卷 (J:)   |      | Drive | 2018.5.31 12:21 |

显示结果因本地文件系统而异

### 二、QFileSystemModel 类中的属性

- 1、nameFileterDisables: bool

访问函数: bool nameFileterDisables() const; void setNameFileterDisables(bool);

描述是否隐藏由名称过滤器过滤掉的文件，默认为 true(不隐藏)，若文件未被隐藏，将被显示为被禁用的状态(通常为灰色)

- 2、readOnly: bool 访问函数: bool isReadOnly() const; void setReadOnly(bool);

只读模式，即是否允许目录模型写入文件系统，若为 false 则允许重命名、复制、删除文件和目录等操作。默认为 true(即只读)。

- 3、**resolveSymlinks**: bool      **访问函数**: bool resolveSymlinks() const; void setResolveSymlinks(bool);  
仅适用于 windows，描述目录模型是否应解析符号链接(类似于快捷方式)。默认为 true。

### 三、QFileSystemModel 类中的函数(重新实现的基类函数未重复列出)

- 1、**QFileSystemModel**(QObject \*parent = Q\_NULLPTR)      //构造函数

#### 2、根目录

QDir **rootDirectory**() const      //当前设置的根目录

QString **rootPath**() const      //当前设置的根路径

QModelIndex **setRootPath**(const QString &newPath)      //重要函数

- 通过在模型上安装文件系统监视程序(QFileSystemWatcher)，将模型正在监视的目录设置为 newPath，该目录中的任何文件和目录的更改都会反映到模型中，若路径被更改，则发送 rootPathChanged()信号。
- 注意：该函数不会更改模型的结构或修改视图的数据，也就是说，模型的“根”不会更改为仅包含文件系统中由 newPath 指定的目录内的文件和目录。
- 以上规则说明，无论该函数设置为何种目录，模型的根不会被更改(也没有可以更改的函数)，对于 Windows 而言，模型的根是“我的电脑”(即直接显示电脑下的硬盘分区)。另外，该函数的设置也不会影响视图的显示。
- setRootPath()函数的最终结果是，对文件系统模型的设置将只作用于由该函数指定的目录及其子目录，比如

```
QFileSystemModel *d=new QFileSystemModel;
```

```
d->setRootPath("F:"); //无论此处指定哪个目录，模型将始终读取本地文件系统的根目录，
```

```
 //也就是说模型将始终为“我的电脑”下的硬盘分区。
```

```
 //视图也不会更改为显示该目录。
```

```
//以下对模型的设置将只作用于目录"F:/"及其子目录，对于其他目录，比如 C:/、D:/等将不起作用。
```

```
d->sort(0);
```

```
d->setReadOnly(false);
```

```
...
```

#### 3、过滤器

- 1)、void **setFilter**(QDir::Filters filters)

QDir::Filters **filter**() const

以上函数表示设置和返回目录模型的过滤器，设置的过滤器应始终包含 QDir::AllDirs，否则 QFileSystemModel 将无法读取目录结构。若未设置过滤器，则默认过滤器是 QDir::AllEntries|QDir::NoDotAndDotDot | QDir::AllDirs，即列出目录、文件、驱动器、符号连接，且不列出特殊条目 "...”，QDir::Filter 枚举参见 QDir 类。

- 2)、void **setNameFilters**(const QStringList &filters)

QStringList **nameFilters**() const

以上函数用于设置和返回模型的名称过滤器列表。此处的名称过滤器的设置方法与第 6 章文件对话框章节的有些不同，规则如下

过滤器使用字符串列表的形式指定，一次只能指定一种类型，且不支持 MIME 类型，

比如

```
QStringList s; s<<"*.jpg"<<"*.txt"; //等同于"*.jpg *.txt"
//以下设置的名称过滤器都是无效的。
s<<"*.jpg *.bmp"; s<<"All file(*)"; s<<"text/plain";
```

#### 4、删除、创建

- 3)、QModelIndex **mkdir**(const QModelIndex &parent, const QString &name)

在父模型索引 parent 中创建一个名称为 name 的目录。

- 4)、bool **rmdir**(const QModelIndex &index)

移除并删除文件系统模型中索引 index 所对应的目录。若删除成功则返回 true，否则返回 false。注意：该函数并不会把目录放置于可以恢复的位置(比如回收站)，而是直接删除。

- 5)、bool **remove**(const QModelIndex &index)

从文件系统模型中移除模型索引 index，并删除对应的文件，若删除成功，则返回 true，否则返回 false。注意：该函数并不会把文件放置于可以恢复的位置(比如回收站)，而是直接删除。

#### 5、获取信息

- 6)、QIcon **fileIcon**(const QModelIndex &index) const

QFileInfo **fileInfo**(const QModelIndex &index) const

QString **fileName**(const QModelIndex &index) const

QString **filePath**(const QModelIndex &index) const

以上函数分别表示，返回索引 index 所指项目的图标、QFileInfo(文件信息)、文件名、路径。QFileInfo 类用于描述文件的信息(比如创建日期、时间、大小等)，此类暂时不用深入理解。

- 7)、QModelIndex **index**(const QString &path, int column = 0) const

返回给定路径 path 和列 column 的项目的模型索引。

- 8)、bool **isDir**(const QModelIndex &index) const

若索引 index 所指的项目表示目录则返回 true，否则返回 false。

- 9)、QDateTime **lastModified**(const QModelIndex &index) const

返回上次修改索引的日期和时间。

- 10)、QFile::Permissions **permissions**(const QModelIndex &index) const

返回索引 index 所指项目的文件权限，QFile::Permission 枚举详见 QFileDevice 类(即 QFile 的父类)。

- 11)、qint64 **size**(const QModelIndex &index) const

返回索引 index 所指项目的大小，若该文件不存则，则返回 0。qint64 是一个使用 typedef 重命名的保证为 64 位的 long long int 类型。

- 12)、QString **type**(const QModelIndex &index) const

返回索引 index 所指项目的类型，比如"jpeg"、"txt"、"Directory"等。

- 13)、QVariant **myComputer**(int role = Qt::DisplayRole) const

返回项目“我的电脑”中与角色 role 关联的数据。

## 6、图标

14)、void **setIconProvider**(QFileIconProvider \**provider*)

QFileIconProvider \***iconProvider**() const

以上函数表示设置和返回此目录模型的文件图标提供程序，QFileIconProvider 类为 QFileSystemModel 提供文件图标

## 7、信号

15)、void **directoryLoaded**(const QString &*path*); //信号

当线程完成路径 *path* 的加载时，发送此信号。

16)、void **fileRenamed**(const QString &*path*, const QString &*oldName*, const QString &*newName*); //信号

当文件 *oldName* 被成功重命名为 *newName* 时，发送此信号。参数 *path* 为文件的路径。

17)、void **rootPathChanged**(const QString &*newPath*); //信号

当根路径更改为 *newPath* 时，发送此信号。

## 示例：一个简单的文件浏览器

//m.h 文件的内容

```
#ifndef M_H
```

```
#define M_H
```

```
#include<QtWidgets>
```

```
class B:public QWidget{ Q_OBJECT
```

```
public: QTreeView *pv2; QListView *pv21; QComboBox *pc1; QFileSystemModel *d;
```

```
 B(QWidget *p=0):QWidget(p) {
```

```
 //初始化各部件
```

```
 pv2=new QTreeView(this); pv21=new QListView(this); d=new QFileSystemModel;
```

```
 pc1=new QComboBox; QVBoxLayout *ph=new QVBoxLayout;
```

```
 //添加分隔条
```

```
 QSplitter *ps=new QSplitter; ps->addWidget(pv2); ps->addWidget(pv21);
```

```
 //设置列表框 pc1
```

```
 pc1->setSizePolicy(QSizePolicy::Expanding,QSizePolicy::Fixed); //设置大小策略。
```

```
 QStringList s; s<<"*"<<"*.jpg;*.bmp"<<"*.txt;*.text"<<"*.jpg;*.txt;*.text";
```

```
 pc1->addItems(s);
```

```
 //布局部件。
```

```
 ph->addWidget(ps); ph->addWidget(pc1); setLayout(ph);
```

```
 //设置模型及视图
```

```
 pv2->setHeaderHidden(1); //隐藏标头
```

```
 d->sort(0); //使用第 1 列进行排序(默认为升序)
```

```
 d->setRootPath(""); //空路径将监视所有目录
```

```
 d->setReadOnly(false); //设置为非只读模式
```

```
 //两个视图使用同一个模型
```

```
 pv2->setModel(d); pv21->setModel(d);
```

```
 //以下函数必须位于视图设置模型之后，否则这些函数将不起作用
```

```
 d->setNameFilterDisables(0); //隐藏被名称过滤器过滤掉的文件。
```

```
 pv2->hideColumn(1); //隐藏视图的第 1 列
```

```
 pv2->hideColumn(2);
```

```
 pv2->hideColumn(3);
```

```

 pv21->setRootIndex(d->index("F:/app")); //设置视图 pv21 的根索引(即显示该目录的内容)。
//单击 pv2 或双击 pv21 中的项目时，都将使 pv2 显示该项目的内容。
 QObject::connect(pv2, &QTreeView::clicked, this, &B::f1);
 QObject::connect(pv21, &QTreeView::doubleClicked, this, &B::f1);
//将选择 pc1 中的项设置为模型 d 的名称过滤器
 QObject::connect(pc1, SIGNAL(activated(QString)), this, SLOT(f2(QString)));
}
public slots:
 void f1(QModelIndex i){ pv21->setRootIndex(i);} //设置视图 pv21 显示的项目。
 void f2(QString t){
 QStringList s;
 s=t.split(";"); //使用分号将字符串拆分为字符串列表，
 //假设 t="a;bb;c"，则列表 s=["a","bb","c"]
 d->setNameFilters(s); } //把 s 设置为模型 d 的名称过滤器
};
#endif // M_H

```

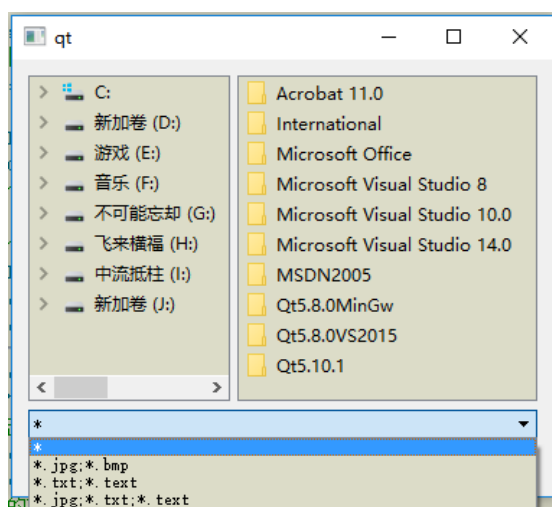
//m.cpp 文件的内容

```

#include "m.h"
int main(int argc, char *argv[]){ QApplication app(argc, argv);
 B w; w.resize(444, 355); w.show(); return app.exec(); }

```

运行结果及说明



单击左侧的目录或双击右侧的目录，都将显示该目录的内容，选择下方的过滤器，将过滤掉不需要显示的文件。

## 8.11 QTableView 类(表格视图)

1、QTableView 继承自 QAbstractItemView。该类是 Qt 实现的标准视图之一，实现了一个显示表格的视图。

### 一、QTableView 类中的属性

1、**cornerButtonEnabled**: bool

**访问函数**: bool isCornerButtonEnabled() const; void setCornerButtonEnabled(bool);

是否启用表格左上角的按钮，若为 true(启用)，则点击该按钮会使表格视图中的所有单元格被选择。默认为 true。

2、**gridStyle**: Qt::PenStyle **访问函数**: Qt::PenStyle gridStyle() const; void setGridStyle(Qt::PenStyle);

描述绘制表格的画笔样式，比如 Qt::PenStyle 取值如下：

Qt::SolidLine(实线)，Qt::DashLine(虚线)，Qt::DotLine(点线)，Qt::DashDotLine(点破折线)，Qt::DashDotDotLine(点点破折线)，Qt::NoPen(无画笔)

3、**showGrid**: bool **访问函数**: bool showGrid() const; void setShowGrid(bool); //槽

是否绘制网格线，默认为 true(绘制)

4、**sortingEnabled**: bool **访问函数**: bool isSortingEnabled() const; void setSortingEnabled(bool);

是否启用排序。默认为 false(不启用)

5、**wordWrap**: bool **访问函数**: bool wordWrap() const; void setWordWrap(bool);

数据项的文字是否自动换行。默认为 true(换行)

### 二、QTableView 类中的函数(重新实现的基类函数未重复列出)

1、**QTableView**(QWidget \*parent = Q\_NULLPTR) //构造函数

2、单元格大小

1)、void **setSpan**(int row, int column, int rowSpanCount, int columnSpanCount)

把(row,column)处的单元格的跨度设置为跨越 rowSpanCount 行或 columnSpanCount 列。

2)、int **rowSpan**(int row, int column) const

int **columnSpan**(int row, int column) const

以上函数表示返回(row, column)处单元格的行/列跨度，默认值为 1。

3)、void **clearSpans**() //删除视图中的所有行和列跨度。

4)、void **setColumnWidth**(int column, int width)

int **columnWidth**(int column) const

void **setRowHeight**(int row, int height)

int **rowHeight**(int row) const

以上函数表示设置和获取列 column 的宽度或行 row 的高度。

3、坐标(单元格位置)

注：以下函数的坐标都是指的内容坐标，即坐标原点位于视图的左上角(不含标头)

5)、int **columnAt**(int x) const //返回 x 方向上位于坐标 x 处的行

int **rowAt**(int *y*) const //返回 *y* 方向上位于坐标 *y* 处的列

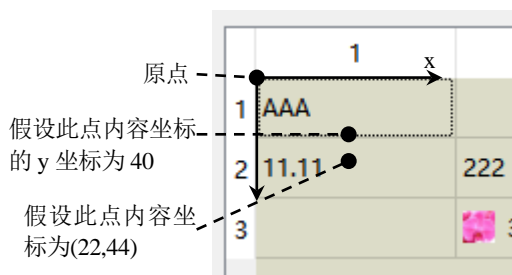
6)、int **columnViewportPosition**(int *column*) const

int **rowViewportPosition**(int *row*) const

以上函数表示返回列 *column*/行 *row* 的 *x* 坐标/*y* 坐标。

7)、virtual QModelIndex **indexAt**(const QPoint &*pos*) const; //虚拟的

这是对 QAbstractItemView::indexAt() 的重新实现，表示返回内容坐标 *pos* 所在位置的数据项的索引。



rowAt(44); //返回 1(即第一行)  
rowViewportPosition(1); //返回 40。  
indexAt(QPoint(22,44)); //返回数据项  
//11.11 的索引

#### 4、标头

8)、QHeaderView \***horizontalHeader**() const

void **setHorizontalHeader**(QHeaderView \**header*)

QHeaderView \***verticalHeader**() const

void **setVerticalHeader**(QHeaderView \**header*)

以上函数表示设置视图的水平和垂直标头。QHeaderView 见 QHeaderView 类。

#### 5、隐藏单元格

9)、bool **isColumnHidden**(int *column*) const

bool **isRowHidden**(int *row*) const

以上函数表示列 *column* 或行 *row* 若是隐藏的，则返回 true。

10)、void **setRowHidden**(int *row*, bool *hide*)

void **setColumnHidden**(int *column*, bool *hide*)

以上函数表示若 *hide* 为 true，则隐藏列 *column* 或行 *row*

#### 6、排序

11)、void **sortByColumn**(int *column*, Qt::SortOrder *order*) //按列 *column* 进行排序(升序/降序)。

#### 7、槽

12)、void **resizeColumnToContents**(int *column*)

void **resizeColumnsToContents**()

void **resizeRowToContents**(int *row*)

void **resizeRowsToContents**()

以上槽函数表示根据委托的大小提示设置列 *column*/行 *row* 或所有列/行的大小

13)、void **hideColumn**(int *column*)

void **hideRow**(int *row*)

void **showColumn**(int *column*)

void **showRow**(int *row*)

以上槽函数表示，隐藏和显示列 `column` 或行 `row`

14)、void `selectColumn`(int *column*)

void `selectRow`(int *row*)

以上槽函数表示选择整列 `column` 或整行 `row`

## 8、受保护的槽

15)、void `columnCountChanged`(int *oldCount*, int *newCount*)

void `rowCountChanged`(int *oldCount*, int *newCount*)

当添加或删除列/行时，调用以上槽函数，`oldCount` 表示之前的列数/行，`newCount` 表示新的列数/行数。

16)、void `columnMoved`(int *column*, int *oldIndex*, int *newIndex*)

void `rowMoved`(int *row*, int *oldIndex*, int *newIndex*)

更改视图中列 `column`/行 `row` 的索引，`oldIndex` 表示旧索引，`newIndex` 表示新索引。

17)、void `columnResized`(int *column*, int *oldWidth*, int *newWidth*)

void `rowResized`(int *row*, int *oldHeight*, int *newHeight*)

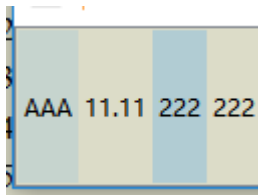
更改列 `column`/行的宽度/高度，`oldWidth/oldHeight` 表示旧的宽度/高度，`newWidth/newHeight` 表示新的宽度/高度。



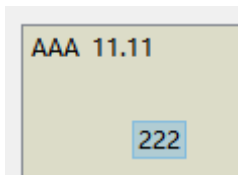
## 8.12 QListView 类(列表视图)

### 一、QListView 类基础

- 1、QListView 继承自 QAbstractItemView。该类是 Qt 实现的标准视图之一，实现了一个显示列表的视图。
- 2、列表视图没有水平和垂直标头。
- 3、列表视图可以按水平或垂直方向显示(由 flow 属性设置)。
- 4、列表视图可以 ListMode 模式和 IconMode 模式显示(由 viewMode 属性设置)。
- 5、列表视图中的项目还可以自由移动和捕捉(由 movement 属性设置)，自由移动是指数据项可以移至视图中的任何位置，而捕捉则只能将数据项移至视图的概念网格中。



水平方向显示的列表视图



列表视图的 IconMode 模式与自由移动

### 二、QListView 类中的属性

| QListView 类属性速查表     |             |             |                |
|----------------------|-------------|-------------|----------------|
| 属性                   | 说明          | 属性          | 说明             |
| flow                 | 项目排列方向      | spacing     | 项目周围的空白空间大小    |
| resizeMode           | 是否自动布局      | gridSize    | 网格大小           |
| viewMode             | 视图模式(图标或列表) | modelColumn | 显示模型中的哪一列      |
| selectionRectVisible | 选择矩形是否可见    | wordWrap    | 项目中的文字是否自动换行   |
| uniformItemSizes     | 项目是否具有相同大小  | isWrapping  | 项目是否自动换行       |
| layoutMode           | 布局模式        | movement    | 移动方式(静态、自由、捕捉) |
| batchSize            | 批处理数量       |             |                |

- 1、**flow**: Flow                      **访问函数**: Flow flow() const;   void setFlow(Flow);  
描述视图中数据项的排列方向，默认为 TopToBottom(从上到下)，注意：即使视图是从左到右排列，但显示的仍是模型中数据项的列。Flow 枚举见下表

| QListView::Flow 枚举(无标志) |   |              |
|-------------------------|---|--------------|
| 作用：描述项目的排列方向            |   |              |
| 成员                      | 值 | 说明           |
| QListView::LeftToRight  | 0 | 项目从左到右排列     |
| QListView::TopToBottom  | 1 | 项目从上到下排列(默认) |

- 2、 **viewMode**: ViewMode      **访问函数**: ViewMode viewMode() const; void setViewMode(ViewMode);  
设置视图的视图模式，ViewMode 枚举见下表。

| QListView::ViewMode 枚举(无标志) |   |                                     |
|-----------------------------|---|-------------------------------------|
| 作用：描述列表视图的视图模式              |   |                                     |
| 成员                          | 值 | 说明                                  |
| QListView::ListMode         | 0 | 列表模式，项目默认使用 TopToBottom 布局，且具有静态移动。 |
| QListView::IconMode         | 1 | 图标模式，项目默认使用 LeftToRight 排列，具有自由移动   |

- 3、 **movement**: Movement      **访问函数**: Movement movement() const; void setMovement(Movement);  
描述项目的移动方式，Movement 见下表

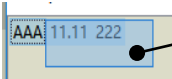
| QListView::Movement 枚举(无标志) |   |                                     |
|-----------------------------|---|-------------------------------------|
| 作用：描述项目的移动方式                |   |                                     |
| 成员                          | 值 | 说明                                  |
| QListView::Static           | 0 | 静态移动，即项目不能被用户移动(ListMode 模式的默认值)    |
| QListView::Free             | 1 | 自由移动，即项目可移动至视图任意位置(IconMode 模式的默认值) |
| QListView::Snap             | 2 | 捕捉，项目移动时会被捕捉到指定的网格。                 |

- 4、 **modelColumn**: int      **访问函数**: int modelColumn() const; void setModelColumn(int);  
设置视图应显示模型中的哪一列，默认为 0(即显示第一列)。注意：该属性的设置需位于视图添加模型之后，否则将不起作用。

- 5、 **selectionRectVisible**: bool

**访问函数**: bool isSelectionRectVisible() const; void setSelectionRectVisible(bool);

描述是否显示选择矩形(见图)，默认为 false(不显示)。  
该属性仅在可以选择多个项目时才有效，若选择模式 (QAbstractItem::SelectionMode 属性)是



阴影部分即为  
选择矩形

QAbstractItemView::SingleSelection(即只能选择一个)，则此属性将不起作用，列表视图默认只能选择一个项目。

- 6、 **uniformItemSizes**: bool      **访问函数**: bool uniformItemSizes() const;void setUniformItemSizes(bool);

描述视图中的项目是否具有相同的大小，默认为 false。

- 7、 **wordWrap**: bool      **访问函数**: bool wordWrap() const; void setWordWrap(bool);

设置数据项中的文字是否自动换行，默认为 false(不换行)

- 8、 **isWrapping**: bool      **访问函数**: bool isWrapping() const; void setWrapping(bool);

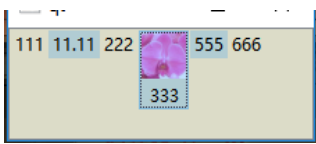
描述项目是否应自动换行，默认为 false(不换行)

- 9、 **resizeMode**: ResizeMode      **访问函数**: ResizeMode resizeMode() const;void setResizeMode(ResizeMode);

描述在调整视图大小时，是否可以再次布局项目，默认为 Fixed。ResizeMode 枚举见下表

| QListView::ResizeMode 枚举(无标志) |  |  |
|-------------------------------|--|--|
| 作用：描述项目改变大小时的布局方式             |  |  |

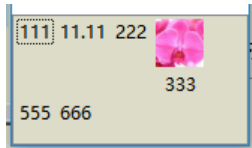
| 成员                | 值 | 说明                             |
|-------------------|---|--------------------------------|
| QListView::Fixed  | 0 | 项目仅在第一次显示时布局，在视图调整大小时，项目将固定不动。 |
| QListView::Adjust | 1 | 每次调整视图大小时都对项目进行布局              |



以 IconMode 模式显示的初始状态

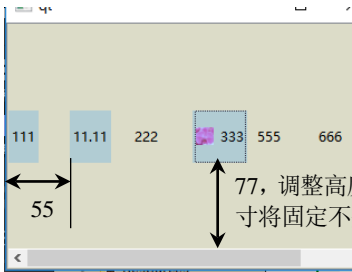


Fixed 布局方式在改变视图大小时，项目未变动(或布局)

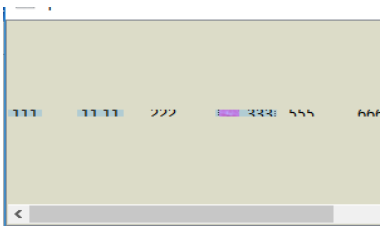


Adjust 布局方式在改变视图大小时，项目被重新布局，注意：isWrapping 属性还应为 true

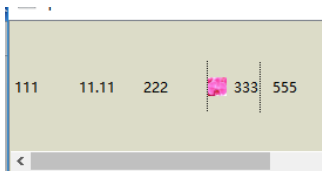
- 10、**gridSize**: QSize
访问函数: QSize gridSize() const; void setGridSize(const QSize&);
描述网格的大小，默认是空白大小(即没有网格)。设置此属性将忽略 spacing 属性。
- 11、**spacing**: int
访问函数: int spacing() const; void setSpacing(int);
描述项目周围的空白空间大小，默认值为 0。设置间距后，在改变视图的大小时，数据项可能会被压缩，原理见下图



初始大小及代码  
pv2->setGridSize(QSize(55,44));  
pv2->setSpacing(77);



调整高度时，数据项被压缩



继续调低高度，数据项再次显示出来，但此时数据项将无法被选择了。

- 12、**batchSize**: int
访问函数: int batchSize() const; void setBatchSize(int);
若 layoutMode 属性设置为 Batch(批处理)，则此属性用于描述每批次布局的项目数量。默认为 100。
- 13、**layoutMode**: LayoutMode
访问函数: LayoutMode layoutMode() const; void setLayoutMode(LayoutMode);
描述项目的布局模式，当模式为 Batched(批处理)时，则在批处理项目的同时还可以处理

事件，这样，便可以即时查看可见项目并与其交互，而其他项目还正在布局中，默认为 SinglePass。LayoutMode 枚举见下表

| QListView::LayoutMode 枚举(无标志) |   |             |
|-------------------------------|---|-------------|
| 作用：描述项目的布局模式                  |   |             |
| 成员                            | 值 | 说明          |
| QListView::SinglePass         | 0 | 项目一次性排列(默认) |
| QListView::Batched            | 1 | 项目分批批量排列。   |

### 三、QListView 类中的函数

- 1、QListView(QWidget \*parent = Q\_NULLPTR) //构造函数
  - 2、void clearPropertyFlags()  
清除 QListView 的特定属性标志，比如，在调用 setMovement()或 setViewMode()时，QAbstractItemView::dragEnabled 属性和 QWidget::acceptDrops 属性是由 QListView 设置的。
  - 3、bool isRowHidden(int row) const
  - 4、void setRowHidden(int row, bool hide)  
以上函数用于获取和设置行 row 的隐藏状态。
  - 5、QRect rectForIndex(const QModelIndex &index) const; //受保护的  
返回模型中索引 index 所指项目的矩形(使用内容坐标)
  - 6、void setPositionForIndex( const QPoint &position, const QModelIndex &index); //受保护的  
将索引 index 所指项目的位置设置为 position，若列表视图为静态移动模式，则此函数将不起作用。
  - 7、void indexesMoved(const QModelIndexList &indexes); //信号  
当索引 indexes 在视图中移动时，发送此信号。
- 示例参阅 QFileSystemModel 类

## 8.13 QTreeView 类(树视图)

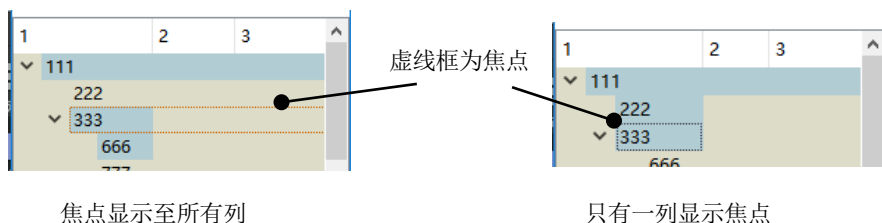
1、QTreeView 继承自 QAbstractItemView。该类是 Qt 实现的标准视图之一，实现了一个显示树形结构的视图。

### 一、QTreeView 类中的属性

#### 1、allColumnsShowFocus: bool

访问函数: `bool allColumnsShowFocus() const; void setAllColumnsShowFocus(bool);`

设置键盘焦点是否显示至所有列，默认为 false(即只有一列显示焦点)，原理见下图



#### 2、animated: bool

访问函数: `bool isAnimated() const; void setAnimated(bool);`

是否启用动画，若为 false(默认值)，则树视图会立即展开或折叠分支；若为 true，则会动画展开和折叠分支。

#### 3、autoExpandDelay: int

访问函数: `int autoExpandDelay() const; void setAutoExpandDelay(int);`

自动扩展延迟，在拖放操作时自动打开或关闭树节点之前的延迟时间(以毫秒为单位)，若小于 0，则禁用自动扩展，默认为-1。

#### 4、expandsOnDoubleClick: bool

访问函数: `bool expandsOnDoubleClick() const; void setExpandsOnDoubleClick(bool);`

是否可使用双击展开和折叠节点。

#### 5、headerHidden: bool

访问函数: `bool isHeaderHidden() const; void setHeaderHidden(bool);`

是否隐藏标头，默认为 false(不隐藏)。

#### 6、indentation: int

访问函数: `int indentation() const; void setIndentation(int); void resetIndentation();`

描述节点的缩进量(以像素为单位)，顶级节点是指从视口边缘到第 1 列的水平距离，子节点的缩进量相对于父节点。默认取决于样式。

#### 7、itemsExpandable: bool

访问函数: `bool itemsExpandable() const; void setItemsExpandable(bool);`

节点是否可由用户展开和折叠，默认为 true(允许展开和折叠)

#### 8、rootIsDecorated: bool

访问函数: `bool rootIsDecorated() const; void setRootIsDecorated(bool);`

是否显示顶级项目用于展开和折叠顶级项目的控件，默认为 true(显示)。

#### 9、sortingEnabled: bool

访问函数: `bool isSortingEnabled() const; void setSortingEnabled(bool);`

是否启用排序，默认为 false(不启用)。为提高性能，建议在项目插入树中之后启用排序。

#### 10、uniformRowHeights: bool

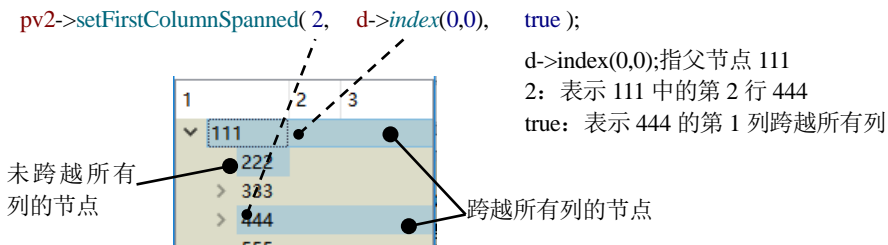
访问函数: `bool uniformRowHeights() const; void setUniformRowHeights(bool);`

确保所有数据项是否具有相同的高度，默认为 false(不确保具有相同高度)

- 11、**wordWrap**: bool                      **访问函数**: bool wordWrap() const; void setwordWrap(bool);  
设置数据项中的文字是否自动换行，默认为 false(不换行)

## 二、QTreeView 类中的函数

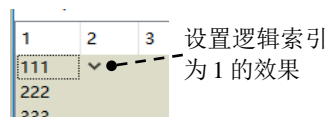
- 1、**QTreeView**(QWidget \*parent = Q\_NULLPTR)    //构造函数
- 2、void **setExpanded**(const QModelIndex &index, bool expanded)  
根据 expanded 的值，将索引 index 所指节点设置为展开或折叠
- 3、bool **isExpanded**(const QModelIndex &index) const  
若索引 index 所指节点是展开的则返回 true，否则返回 false。
- 4、int **columnAt**(int x) const                      //返回 x 坐标(内容坐标)所在的列。  
int **columnViewportPosition**(int column) const    //返回视口中列 column 的水平位置(内容坐标)。
- 5、int **columnWidth**(int column) const  
void **setColumnWidth**(int column, int width)    //以上函数表示返回或设置列 column 的宽度。
- 6、QHeaderView \***header**() const  
void **setHeader**(QHeaderView \*header)  
获取和设置视图的标头，设置标头后，该视图拥有该标头的所有权。
- 7、void **setRowHidden**(int row, const QModelIndex &parent, bool hide)  
void **setColumnHidden**(int column, bool hide)  
若 hide 为 true，则隐藏行 row 或列 column，否则显示该行或列。
- 8、bool **isRowHidden**(int row, const QModelIndex &parent) const  
bool **isColumnHidden**(int column) const  
若列 column 或父节点 parent 中的行 row 是隐藏的，则返回 true，否则返回 false
- 9、QModelIndex **indexAbove**(const QModelIndex &index) const  
QModelIndex **indexBelow**(const QModelIndex &index) const  
返回索引 index 上方或下方的模型索引。
- 10、bool **isFirstColumnSpanned**(int row, const QModelIndex &parent) const  
若父级 parent 的行 row 中第 1 列的节点跨越所有列，则返回 true，否则返回 false。
- 11、void **setFirstColumnSpanned**(int row, const QModelIndex &parent, bool span)  
若 span 为 true，则把父级 parent 的行 row 中第 1 列的节点设置为跨越所有列，否则显示在行 row 中的所有项。原理见下图



12、void **setTreePosition**(int *index*) //qt5.2

int **treePosition**() const //qt5.2

以上函数表示，设置和返回树的逻辑索引(见右图)，  
若为-1，则表示树放置在可视索引 0 上。



13、void **sortByColumn**(int *column*, Qt::SortOrder *order*) //按列 *column* 进行排序(升序/降序)。

## 槽

14、void **collapse**(const QModelIndex &*index*)

void **collapseAll**()

void **expand**(const QModelIndex &*index*)

void **expandAll**()

以上槽函数表示，折叠或展开索引 *index* 所指节点或所有节点。

15、void **expandToDepth**(int *depth*); //将所有可展开的节点，展开至给定的深度 *depth*

16、void **resizeColumnToContents**(int *column*) //根据其内容调整列 *column* 的大小。

17、void **showColumn**(int *column*) //显示列 *column*

18、void **hideColumn**(int *column*) //隐藏列 *column*，初始化模型后调用此函数才有效。

## 受保护的槽

19、void **columnCountChanged**(int *oldCount*, int *newCount*)

通知树视图列数已从 *oldCount* 更改为 *newCount*。

20、void **columnMoved**() //当列被移除时，调用此槽函数。

21、void **columnResized**(int *column*, int *oldSize*, int *newSize*) //当标头的大小更改时，调用该槽函数。

22、void **rowsRemoved**(const QModelIndex &*parent*, int *start*, int *end*)

通知视图，从 *start* 开始的行到 *end* 结束的行已从父索引 *parent* 中移除。

## 受保护的函数

23、virtual void **drawBranches**(QPainter \**painter*, const QRect &*rect*, const QModelIndex &*index*) const; //虚拟的  
使用绘制器 *painter* 在与索引 *index* 相同的行上，由 *rect* 指定的矩形中绘制树视图中的分支。

24、virtual void **drawRow**(QPainter \**painter*, const QStyleOptionViewItem &*option*,  
const QModelIndex &*index*) const; //虚拟的  
使用绘制器 *painter* 在包含索引 *index* 的树视图中绘制行。

25、void **drawTree**(QPainter \**painter*, const QRegion &*region*) const

使用绘制器 *painter* 绘制树中与区域 *region* 相交的部分。

26、int **indexRowSizeHint**(const QModelIndex &*index*) const //返回索引 *index* 所指的行的尺寸提示。

27、int **rowHeight**(const QModelIndex &*index*) const //返回索引 *index* 所指行的高度。

## 信号

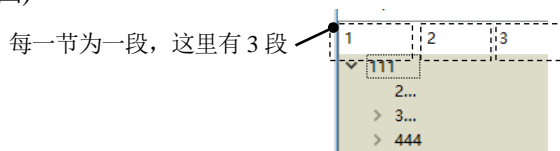
28、void **collapsed**(const QModelIndex &*index*)

void **expanded**(const QModelIndex &*index*) //当 *index* 所指节点折叠或展开时，发送以上信号。

## 8.14 QHeaderView 类(标头视图)

### 一、QHeaderView 类基础

- 1、QHeaderView 继承自 QAbstractItemView。该类是 Qt 实现的标准视图之一，为项目视图提供了标头列和标头行。该类通常与其他视图一起使用，比如 QTreeView 视图可以使用 QTreeView::header() 获取标头视图，使用 QTreeView::setHeader() 设置标头视图。QTableView 视图可以使用 QTableView::horizontalHeader() 获取水平标头视图，QTableView::setHorizontalHeader() 设置水平标头视图等。
- 2、并不是所有数据角色(Qt::ItemDataRole 枚举)都能应用于标头视图，标头视图支持以下角色：Qt::DisplayRole、Qt::FontRole、Qt::DecorationRole(图标)、Qt::ForegroundRole、Qt::BackgroundRole、Qt::TextAlignmentRole。若想要使用其他角色的数据，则需要子类化 QHeaderView 并重新实现 paintEvent() 函数自行绘制。
- 3、标头会渲染每个段自身的数据，并不依赖于委托，因此调用标头视图的 setItemDelegate() 函数将不起作用。
- 4、重要概念：段(section，见下图)



### 二、QHeaderView 类中的属性

#### 1、cascadingSectionResizes: bool

**访问函数：**bool cascadingSectionResizes() const; void setCascadingSectionResizes(bool);

当用户调整大小的段达到最小大小后，是否级联到下一个段(原理见下图)。此属性仅影响调整大小模式(使用 setSectionResizeMode() 函数设置)为 Interactive 的段。默认为 false。



#### 2、defaultAlignment: Qt::Alignment

**访问函数：**Qt::Alignment defaultAlignment() const; void setDefaultAlignment(Qt::Alignment);

标头段中文本的对齐方式。

#### 3、defaultSectionSize: int

**访问函数：**int defaultSectionSize() const; void setDefaultSectionSize(int); void resetDefaultSectionSize();

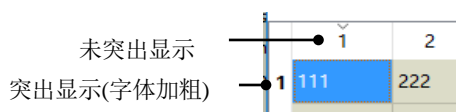
标头段的默认大小，此属性仅影响调整大小模式为 Interactive 或 Fixed 的段。设置此属性将调整标头段的大小。

#### 4、highlightSections: bool



**访问函数:** `bool highlightSections() const; void setHighlightSections(bool);`

是否突出显示所选项目的段，默认为 `false`(不突出显示)。原理见下图



5、**maximumSectionSize:** `int //qt5.2`

**访问函数:** `int maximumSectionSize() const; void setMaximumSectionSize(int);`

标头段的最大大小，默认值为 1048575，若设置为 -1，将重置为最大的段大小。

6、**minimumSectionSize:** `int`

**访问函数:** `int minimumSectionSize() const; void setMinimumSectionSize(int);`

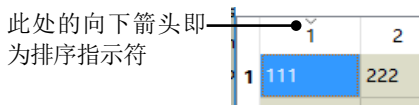
标头段的最小大小，若设置为 -1，将使用 `QApplication::globalStrut` 属性或 `QFontMetrics::fontMetrics()` 的最大大小。

7、**showSortIndicator:** `bool`

**访问函数:** `bool isSortIndicatorShown() const;`

`void setSortIndicatorShown(bool);`

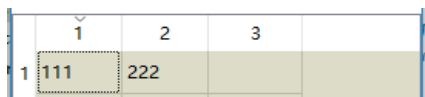
是否显示排序指示符(右图)，默认为 `false`(不显示)。



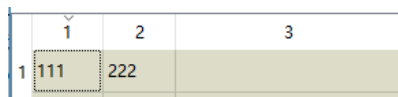
8、**stretchLastSection:** `bool`

**访问函数:** `bool stretchLastSection() const; void setStretchLastSection(bool);`

标头中最后一个可见段是否拉伸至所有可用空间(原理见下图)。默认为 `false`(不拉伸)。注意: `QTreeView` 的水平标头被设置为 `true`。



第 3 段未拉伸的情形



第 3 段拉伸的情形

### 三、QHeaderView 类中的函数

1、**QHeaderView**(`Qt::Orientation orientation, QWidget *parent = Q_NULLPTR`) //构造函数

#### 2、隐藏/显示标头

注: 若隐藏标头的某一段会同时隐藏该段所对应的模型数据，要想只隐藏标头而不隐藏模型数据，可以使用 `QWidget::setHidden()` 函数，不过该函数会隐藏整个标头。

1)、`void hideSection(int logicalIndex);` //隐藏由 `logicalIndex` 指定的段。

`void setSectionHidden(int logicalIndex, bool hide)`

若 `hide` 为 `true`，则隐藏由 `logicalIndex` 指定的段，否则显示该段。

2)、`void showSection(int logicalIndex)` //显示由 `logicalIndex` 指定的段。

3)、`int hiddenSectionCount() const` //返回标头中已隐藏的段的数量。

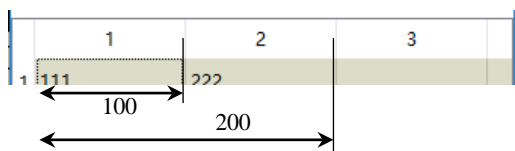
4)、`bool isSectionHidden(int logicalIndex) const`

若 `logicalIndex` 所指的段对用户是隐藏的，则返回 `true`，否则返回 `false`。

5)、bool **sectionsHidden**() const //若标头中的段被隐藏，则返回 true，否则返回 false。

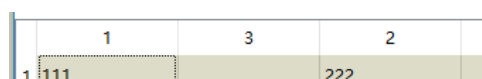
### 3、标头的索引和位置

- ①、可视索引 **visualIndex**、逻辑索引 **logicalIndex**、位置 **position**，原理见下图；
- ②、索引以 0 开始，位置以像素为单位。
- ③、逻辑索引：为标头的段因定所拥有的，也主就是说无论把标头移至何处，逻辑索引不会改变。
- ④、可视索引：即视图上所看到的段的索引，即使该段隐藏，也不会隐藏可视索引。
- ⑤、位置：是从单元格的左侧或顶部开始计算的。



标头段 2 的参数如下：

逻辑索引：1； 可视索引：1； 位置：100



移动后标头段 2 的参数如下：

逻辑索引：1； 可视索引：2； 位置：200

6)、int **logicalIndex**(int **visualIndex**) const

返回可视索引 **visualIndex** 所指段的逻辑索引。若 **visualIndex** 小于 0，或大于等于段的数量，则返回-1。

7)、int **logicalIndexAt**(int **position**) const

int **logicalIndexAt**(int **x**, int **y**) const

int **logicalIndexAt**(const QPoint &**pos**) const

以上函数表示，返回位置 **position** 或坐标(x,y)或点 **pos** 处段的逻辑索引。若是水平标头则使用 **x** 坐标，垂直标头则使用 **y** 坐标。

8)、int **visualIndex**(int **logicalIndex**) const

返回逻辑索引 **logicalIndex** 所指段的可视索引，隐藏的段的可视索引仍然有效。

9)、int **visualIndexAt**(int **position**) const //返回位置 **position** 处段的可视索引。

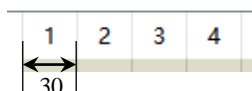
10)、int **sectionPosition**(int **logicalIndex**) const

返回逻辑索引 **logicalIndex** 的位置。若该段被隐藏了，则返回-1。

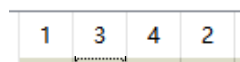
11)、int **sectionViewportPosition**(int **logicalIndex**) const

返回逻辑索引 **logicalIndex** 的视口位置。若该段被隐藏了，则返回的值未定义。

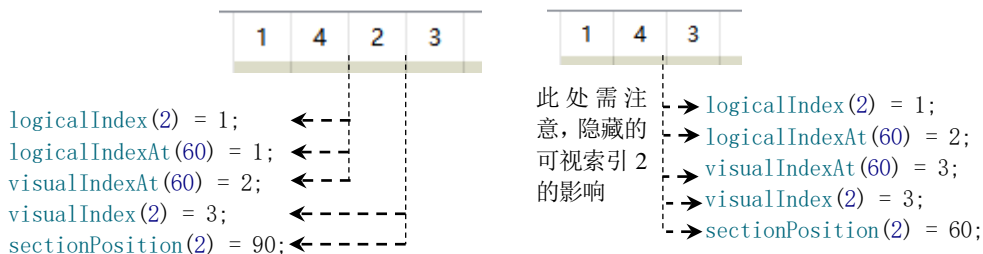
综合示例：



`logicalIndex(2) = 2;` //可视索引 2 的逻辑索引  
`logicalIndexAt(60) = 2;` //位置 60 的逻辑索引  
`visualIndexAt(60) = 2;` //位置 60 的可视索引  
`visualIndex(2) = 2;` //逻辑索引 2 的可视索引  
`sectionPosition(2) = 60;` //逻辑索引 2 的位置

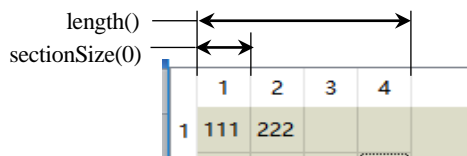


`logicalIndex(2) = 3;`  
`logicalIndexAt(60) = 3;`  
`visualIndexAt(60) = 2;`  
`visualIndex(2) = 1;`  
`sectionPosition(2) = 30;`



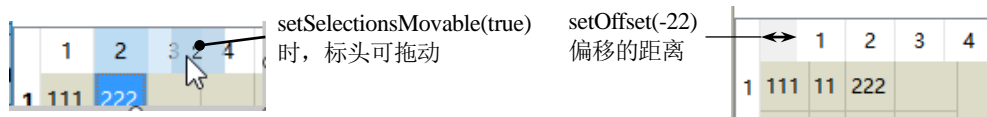
#### 4、获取标头基本信息

- 12)、int **count**() const //返回标头的段数。
- 13)、int **length**() const //返回标头的长度(或高度)
- 14)、int **sectionSize**(int **logicalIndex**) const //返回逻辑索引 **logicalIndex** 所指段的宽度(或高度)
- 15)、Qt::Orientation **orientation**() const //返回标头的方向(即水平或垂直)
- 16)、int **sectionSizeHint**(int **logicalIndex**) const //返回逻辑索引 **logicalIndex** 所指段的大小提示



#### 5、移动标头

- 17)、bool **sectionsMovable**() const //qt5.0 若用户可以移动标头, 则返回 true, 否则返回 false
- void **setSectionsMovable**(bool **movable**)
- 若 **movable** 为 true, 则用户可以拖动标头, 否则不能拖动, 但可使用编程方式移动。
- 18)、bool **sectionsMoved**() const //若标头中的段已被移动, 则返回 true, 否则返回 false。
- 19)、void **moveSection**(int **from**, int **to**) //将可视索引 **from** 所指的段移至可视索引 **to** 所指的段。
- 20)、int **offset**() const //获取标头的偏移量。
- void **setOffset**(int **offset**) //设置标头的偏移量(以像素为单位), 槽。
- 21)、void **swapSections**(int **first**, int **second**) //交换可视索引 **first** 指定的段与可视索引 **second** 指定的段。



#### 6、修改标头的大小

- 22)、ResizeMode **sectionResizeMode**(int **logicalIndex**) const //返回大小调整模式, qt5.0
- void **setSectionResizeMode**(ResizeMode **mode**) //qt5.0
- void **setSectionResizeMode**(int **logicalIndex**, ResizeMode **mode**) //qt5.0
- 设置标头或由逻辑索引 **logicalIndex** 的段的大小调整模式, 逻辑索引 **logicalIndex** 应存在。注意: 若 **stretchLastSection** 属性为 true, 则此设置将在标头的最后一段被忽略。
- ResizeMode 枚举见下表

## QHeaderView::ResizeMode 枚举

作用：描述标头的大小调整模式。

| 成员                            | 值 | 说明                                                         |
|-------------------------------|---|------------------------------------------------------------|
| QHeaderView::Interactive      | 0 | 用户可调整段的大小，另见 <code>cascadingSectionResizes</code> 属性       |
| QHeaderView::Fixed            | 2 | 用户不能调整段的大小，但可使用 <code>resizeSection()</code> 函数以编程的方式调整大小。 |
| QHeaderView::Stretch          | 1 | 标头视图自动调整段的大小以填充可用空间，大小不能由用户或以编程的方式更改。                      |
| QHeaderView::ResizeToContents | 3 | 标头视图将根据行或列的内容自动调整段的最佳大小，大小不能由用户或以编程的方式更改。                  |

23)、void `resizeSections(QHeaderView::ResizeMode mode)`

根据大小调整模式 `mode` 调整各段的大小，忽略当前的大小调整模式。注意：该函数不会把标头视图的大小调整模式设置为 `mode`。比如

```
setSectionResizeMode(QHeaderView::Stretch); //设置大小调整模式为 Stretch(拉伸)
```

```
resizeSections(QHeaderView::ResizeToContents); //根据内容调整标头各段的大小，但调整大小后，
大小调整模式仍为 Stretch，之后改变视图大小仍会使标头各段
随之拉伸。
```

24)、int `stretchSectionCount()` const //返回大小调整模式被设置为 `Stretch`(拉伸)的段的数量。

25)、void `resizeSection(int logicalIndex, int size)`

调整由逻辑索引 `logicalIndex` 所指段的大小为 `size`(以像素为单位)，参数 `size` 必须大于或等于 0，不建议使用 0(此时应隐藏该段)。

26)、int `resizeContentsPrecision()` const //qt5.2

```
void setResizeContentsPrecision(int precision) //qt5.2
```

设置标头视图在使用大小调整模式 `ResizeToContents` 调整大小时的精确度。默认值为 1000，意味着在自动执行大小调整计算水平列的大小时最多可以查看 1000 行。值 0 意味着只查看可见区域，值 -1 表示查看所有元素。重新实现，，函数将使该函数不起作用。较高的值将提供更精确的调整，但速度可能会更慢，较低的值虽然不太精确，但速度更快。该函数设置的值被用于 `QTableView::sizeHintForColumn()`、`QTableView::sizeHintForRow()`、`QTreeView::sizeHintForColumn()`，若重新实现这些函数将使该函数不起作用。

## 7、排序及保存/恢复标头

27)、void `setSortIndicator(int logicalIndex, Qt::SortOrder order)`

- 为由逻辑索引 `logicalIndex` 所指的段按照 `order` 指定的顺序设置排序指示符，并移除其他段中的排序指示符。该函数只负责设置排序指示符的位置，不一定具有排序功能。
- 注意：该函数需要开启 `showSortIndicator` 属性，否则排序指示符不可见。
- 若 `logicalIndex` 为 -1，则不会显示排序指示符。
- 注意：并非所有模型都支持此函数，甚至可能会崩溃。

- 28)、Qt::SortOrder **sortIndicatorOrder**() const  
返回排序指示符的顺序，若该段没有排序指示符，则返回值是未定义的。
- 29)、int **sortIndicatorSection**() const //返回具有排序指示符的段的逻辑索引，默认为 0。
- 30)、QByteArray **saveState**() const //保存标头视图的状态。  
bool **restoreState**(const QByteArray &state) //恢复标头视图的状态。

## 8、标头是否可点击(以下设置会影响部分信号的发送)

- 31)、bool **sectionsClickable**() const //若标头是可点击的，则返回 true，否则返回 false。qt5.0  
void **setSectionsClickable**(bool clickable) //若 clickable 为 true，则标头将响应单击。qt5.0

## 9、槽

- 32)、void **headerDataChanged**(Qt::Orientation orientation, int logicalFirst, int logicalLast)  
使用方向 orientation 更新已更改的标头段，从 logicalFirst 更新到 logicalLast
- 33)、void **setOffsetToLastSection**() //设置偏移量以使最后一段可见。
- 34)、void **setOffsetToSectionPosition**(int visualSectionNumber)  
把偏移量设置为 visualSectionNumber 处的段的起始位置。

## 10、受保护的槽

- 35)、void **resizeSections**()  
根据段的大小提示调整段的大小，通常不需要调用此槽函数。
- 36)、void **sectionsAboutToBeRemoved**(const QModelIndex &parent, int logicalFirst, int logicalLast)  
从父级 parent 删除段时，调用此槽函数，logicalFirst 和 logicalLast 表示被删除的段的位置，若只有一个节点，他们是相等的。
- 37)、void **sectionsInserted**(const QModelIndex &parent, int logicalFirst, int logicalLast)  
当段插入到父级 parent 时，调用此槽函数，logicalFirst 和 logicalLast 表示被删除的段的位置，若只有一个节点，他们是相等的。

## 11、受保护的函数

- 38)、virtual void **paintSection**(QPainter \*painter, const QRect &rect, int logicalIndex) const  
使用给定的 painter、矩形 rect 绘制由逻辑索引 logicalIndex 指定的段。通常不需要调用此函数。
- 39)、virtual QSize **sectionSizeFromContents**(int logicalIndex) const  
返回由逻辑索引 logicalIndex 指定的段的内容的大小。

## 12、信号

- 1)、void **geometriesChanged**()  
当标头的几何形状发生变化时，发送此信号。改变段的大小并未改变标头的几何形状，在最小化时会发送此信号。
- 2)、void **sectionClicked**(int logicalIndex)  
当点击标头的某个段时发送此信号，logicalIndex 表示被点击的段的逻辑索引。注意：

还会发送 `sectionPressed()` 信号。

- 3)、void `sectionPressed`(int *logicalIndex*)

当点击标头的某个段时发送此信号，`logicalIndex` 表示被点击的段的逻辑索引。

- 4)、void `sectionDoubleClicked`(int *logicalIndex*)

当双击某个段时，发送此信号。`logicalIndex` 表示被双击的段的逻辑索引

- 5)、void `sectionHandleDoubleClicked`(int *logicalIndex*)

当双击某个段时，发送此信号。`logicalIndex` 表示被双击的段的逻辑索引

- 6)、void `sectionEntered`(int *logicalIndex*)

当某段上按下鼠标左键不放并将光标移至另一段时，发送此信号，`logicalIndex` 表示光标移动到的段的逻辑索引

- 7)、void `sectionMoved`(int *logicalIndex*, int *oldVisualIndex*, int *newVisualIndex*)

当移动某个段时，发送此信号，`logicalIndex` 表示被移动的段的逻辑索引，`oldVisualIndex` 表示旧索引，`newVisualIndex` 表示新索引

- 8)、void `sectionCountChanged`(int *oldCount*, int *newCount*)

当段数量发生变化时(比如添加或删除段)，发送此信号。`oldCount` 表示旧的数量，`newCount` 表示新的数量。

- 9)、void `sectionResized`(int *logicalIndex*, int *oldSize*, int *newSize*)

当段被调整大小时，发送此信号，`logicalIndex` 为该段的逻辑索引，`oldSize` 为旧的大小，`newSize` 为调整后的大小。

- 10)、void `sortIndicatorChanged`(int *logicalIndex*, Qt::SortOrder *order*)

当含有排序指示符的段或指示的顺序发生变化时，发送此信号，该段的逻辑索引由 `logicalIndex` 指定，排序顺序由 `order` 指定。

## 8.15 QColumnView 类(列视图)

- 1、QColumnView 继承自 QAbstractItemView。该类是 Qt 实现的标准视图之一，实现了一个列视图。
- 2、列视图是一个级联的视图，在其中的每一列都通过使用一个 QListView 来显示其数据。

### QColumnView 类中的属性和函数

- 1、**resizeGripsVisible**: bool

访问函数: bool resizeGripsVisible() const; void setResizeGripsVisible(bool);

调整大小的夹点是否可见，若不可见，则列视图将不能调整位置。默认为 true(可见)。

- 2、**QColumnView**(QWidget \*parent = Q\_NULLPTR) //构造函数

- 3、QList<int> **columnWidths**() const //获取视图中列的宽度

- 4、void **setColumnWidths**(const QList<int> &list)

设置列的宽度。列表中多余的值被保留，在创建新列时会使用。

- 5、QWidget \***previewWidget**() const //返回预览部件，若没有则返回 0。

- 6、void **setPreviewWidget**(QWidget \*widget)

设置预览部件，该部件成为列的子项，并在删除列区域或设置新的部件时被销毁。

- 7、void **updatePreviewWidget**(const QModelIndex &index) //信号

当预览部件被更新时发送此信号。

- 8、以下两个函数用于在自定义列视图时使用(详见稍后的示例)。

virtual QAbstractItemView \* **createColumn**(const QModelIndex &index)

返回一个视图用于列视图中显示内容。QColumnView 自动获取返回的视图的所有权。index 为分配给模型的根索引。

void **initializeColumn**(QAbstractItemView \*column) const

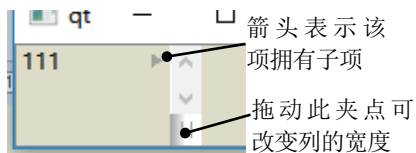
主要用于设置列视图的内容，比如设置列视图的滚动条、拖放等。

### 示例：列视图的使用

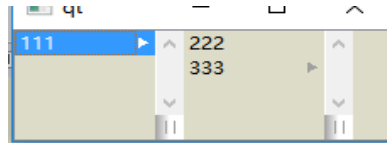
```
#include<QtWidgets>
int main(int argc, char *argv[]) { QApplication app(argc, argv);
 QStandardItemModel *d=new QStandardItemModel; //模型
 QColumnView *pv2=new QColumnView; pv2->resize(333, 222);pv2->move(20, 20); //列视图
 QStandardItem *ps=d->invisibleRootItem(); //获取不可见根项目
 QStandardItem *ps1=new QStandardItem("111");QStandardItem *ps2=new QStandardItem("222");
 QStandardItem *ps3=new QStandardItem("333");QStandardItem *ps4=new QStandardItem("444");
 //以树形结构设置模型数据项
 ps->appendRow(ps1); ps1->appendRow(ps2); ps1->appendRow(ps3) ps3->appendRow(ps4);
 pv2->setSelectionMode(QAbstractItemView::ExtendedSelection); //可同时选择多个
 //设置列视图的列宽
 QList<int> i; i.append(88); i.append(88); i.append(88); pv2->setColumnWidths(i);
 //设置列视图的预览部件
 QPushButton *pb=new QPushButton("BBB");pb->resize(22, 22); pv2->setPreviewWidget(pb);
 pv2->setModel(d); pv2->show(); return app.exec(); }
```



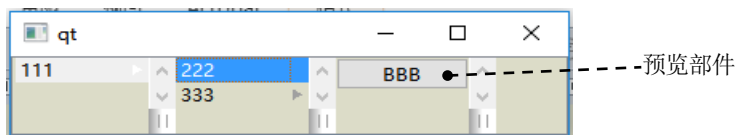
运行结果见下图



初始显示只有一列



点击项目 111，自动在右侧展开第 2 列。



当点击没有子项的项目时，会在右侧的新列中显示设置的预览部件。

### 示例：简单的自定义列视图(本示例只有部分代码)

```
class E:public QColumnView{Q_OBJECT
public:
 QAbstractItemView *createColumn(const QModelIndex &index){
 QListView *v = new QListView(); //使用列表视图显示内容
 initializeColumn(v); /*调用 initializeColumn() 函数，也可直接把 initializeColumn() 函数
 数内的代码复制到该函数中来。从而不使用 initializeColumn 函数。*/
 v->setRootIndex(index); //此代码应位于设置模型之后。
 return v;}
 void initializeColumn(QAbstractItemView *column) const{
 column->setDragDropMode(dragDropMode()); //拖放
 column->setFrameShape(QFrame::NoFrame);
 column->setVerticalScrollBarPolicy(Qt::ScrollBarAlwaysOn); //显示垂直滚动条
 //.....其他设置
 column->setModel(model()); //此步骤是关键，必须设置模型，否则视图没有数据显示。
 }
};
```



## 8.16 QStyledItemDelegate 类

注: QItemDelegate 与 QStyledItemDelegate 类似, 因此就不再对 QItemDelegate 进行讲解

1、QStyledItemDelegate 是所有 Qt 标准项目视图的默认委托。

2、委托主要有绘制和编辑项目两大功能,

- QStyledItemDelegate 绘制委托的方法是根据每个数据元素的角色为数据项进行不同的绘制, 以使项目在视图中显示不同的外观, 下表是该委托可以处理的角色和数据类型, 自定义绘制与子类化 QAbstractItemDelegate 的方法相同, 此处不重述。

| 角色                | 接受的类型   | 角色                    | 接受的类型                       |
|-------------------|---------|-----------------------|-----------------------------|
| Qt::DisplayRole   | QString | Qt::DecorationRole    | QIcon、QPixmap、QImage、QColor |
| Qt::EditRole      |         | Qt::TextAlignmentRole | Qt::Alignment               |
| Qt::FontRole      | QFont   | Qt::CheckStateRole    | Qt::CheckState              |
| Qt::QSizeHintRole | QSize   | Qt::ForegroundRole    | QBrush                      |
|                   |         | Qt::BackgroundRole    | QBrush                      |

- QStyledItemDelegate 使用 QItemEditorFactory 类(编辑器工厂)创建编辑器, 当然也可子类化该类并重新实现 createEditor()等函数创建自定义的编辑器, 这样就不需要使用 QItemEditorFactory 了。子类化 QStyledItemDelegate 的方法与子类化 QAbstractItemDelegate 相同, 详见 QAbstractItemDelegate 类。本小节只简单的讲解怎样使用 QItemEditorFactory 创建编辑器。

3、QStyledItemDelegate 类中的函数:

该类都是重新实现的基类的函数, 除构造/析构函数外总共只有 4 个自己的函数, 而经常使用的就是如下两个设置和返回编辑器工厂的函数

```
QItemEditorFactory * itemEditorFactory() const //返回编辑器工厂
void setItemEditorFactory(QItemEditorFactory *factory) //设置编辑器工厂
```

4、使用编辑器工厂指定自定义的编辑器

1)、编辑器的最终创建者是谁?

QStyledItemDelegate 使用 QItemEditorFactory 创建编辑器, QItemEditorFactory 又通过抽象类 QItemEditorCreatorBase(该类被称为编辑器创建者)创建编辑器, 因此最终编辑器是由重新实现 QItemEditorCreatorBase::createWidget() 函数的抽象类 QItemEditorCreatorBase 的具体子类创建的。QItemEditorCreatorBase 有两个子类, 分别是 QStandardItemEditorCreator 和 QItemEditorCreator。具体实现时可以使用模板类来实现编辑器的创建(Qt 也是这样实现的), 代码如下

//类 E 完全由用户创建, 这就是自定义的编辑器。

```
class E:public QSpinBox{ public:E(QWidget *p=0):QSpinBox(p) {} };
//类 A 就是编辑器创建者, 负责创建编辑器。当然可直接使用 QStandardItemEditorCreator 类
//此处写出以下代码只是让读者明白其原理。
```

```
template<class T> class A :public QItemEditorCreatorBase{public:
 QWidget *createWidget(QWidget *parent) const {
 return new T(parent); } //返回的实例就是新创建的自定义的编辑器
```

```

 QByteArray valuePropertyName() const { //这是必须实现的另一个纯虚函数
 return QByteArray(); }
};

```

`QItemEditorCreatorBase* pa= new A<E>();` //这样就创建了一个类型为 E 的自定义编辑器了。

## 2)、把自定义编辑器传递给 `QStyledItemDelegate` 委托

- 首先需要在编辑器工厂中注册自定义的编辑器，其方法为

```

QItemEditorFactory *e = new QItemEditorFactory;

```

```

e->registerEditor(QVariant::String, pa); /*注册编辑器，此步骤表示，凡是 QString 类型的
 对象，都使用由编辑器创建者 pa 创建的编辑器编辑。*/

```

- 然后使用委托安装编辑器工厂

//以下的 pv 为某个视图类的实例

```

QStyledItemDelegate *pr=(QStyledItemDelegate*)pv->itemDelegate();

```

```

pr->setItemEditorFactory(e); /*安装编辑器工厂，这样就可以使用编辑器工厂创建(间接创建)的编辑器编辑数据了。*/

```

## 第3篇 使用现成的模型/视图部件

### 8.17 QTableWidget 类(表格部件)

- 1、QTableWidget 类继承自 QTableView，该类是一个由 Qt 实现的标准的表格部件，该类的数据项由 QTableWidgetItem 类管理。
- 2、当前单元格(或当前项目)与当前索引或当前选择是相同的，即可以同时选择多个单元格，但只能有一个当前单元格，当编辑单元格时，只会编辑当前单元格。当前单元格通常具有焦点边框。
- 3、单元格和项目的区别(重要概念):  
项目是指 QTableWidgetItem 类的对象，因此空单元格是不含有项目的。当在空单元格上单击鼠标时，会发送 cellClicked()信号，但不会发送 itemClicked()信号。因此当前单元格和当前项目也是有区别的。

#### 一、QTableWidget 类中的属性和函数

##### 1、属性和构造函数

- 1)、**columnCount**: int                      访问函数: int columnCount() const; void setColumnCount(int);
- 2)、**rowCount**: int                      访问函数: int rowCount() const; void setRowCount(int);  
    以上属性描述表格的行数和列数
- 3)、**QTableWidget**(QWidget \*parent = Q\_NULLPTR)      //构造函数  
    **QTableWidget**(int rows, int columns, QWidget \*parent = Q\_NULLPTR)

##### 2、设置、移除、获取项目基本信息

- 4)、void **setItem**(int row, int column, QTableWidgetItem \*item)  
    把(row, column)处的项目设置为 item，该表格取得 item 的所有权。注意：若启用了排序(按列)，则会将行移至由排序确定的位置。
- 5)、QTableWidgetItem \* **takeItem**(int row, int column);      //从表中移除而不删除(row, column)处的项目。
- 6)、QTableWidgetItem \***item**(int row, int column) const      //返回(row, column)处的项目，否则返回 0。
- 7)、QTableWidgetItem \***itemAt**(const QPoint &point) const; //返回位置 point 处的项目。使用内容坐标。
- 8)、QTableWidgetItem \***itemAt**(int ax, int ay) const      //返回位置(ax, ay)处的项目。使用内容坐标。
- 9)、QRect **visualItemRect**(const QTableWidgetItem \*item) const  
    返回项目 item 在视口上所占用的矩形(即位置和大小)。
- 10)、int **column**(const QTableWidgetItem \*item) const      //返回项目 item 所在的列
- 11)、int **row**(const QTableWidgetItem \*item) const      //返回项目 item 所在的行。
- 12)、QList<QTableWidgetItem \*> **items**(const QMimeData \*data) const;      //返回包含在 data 中的项目列表。

##### 3、当前项目

13)、void **setCurrentCell**(int *row*, int *column*)

void **setCurrentCell**(int *row*, int *column*, QTableWidgetItem::SelectionFlags *command*)

void **setCurrentItem**(QTableWidgetItem *\*item*)

void **setCurrentItem**(QTableWidgetItem *\*item*, QTableWidgetItem::SelectionFlags *command*)

以上函数用于设置当前单元格或当前项目，除非选择模式为 NoSelection，否则当前项目会同时被选中。

14)、QTableWidgetItem **currentItem**() const //返回当前项目

15)、int **currentColumn**() const //返回当前项目所在的列。

16)、int **currentRow**() const //返回当前项目所在的行。

## 4、选择项目

17)、QList<QTableWidgetItem\*> **selectedItems**() const

返回所有被选择项目的列表(不含空单元格)。函数 QTableWidgetItem::selectedIndexes() 包含空单元格。

18)、QList<QTableWidgetItemSelectionRange> **selectedRanges**() const

返回所有被选择项目的范围列表(含空单元格)。

19)、void **setRangeSelected**(const QTableWidgetItemSelectionRange &*range*, bool *select*)

选择或取消选择范围 *range* 指定的项目。QTableWidgetItemSelectionRange 类类似于 QTableWidgetItemSelection 用于指示选择的范围。使用方法如下：

QTableWidgetItemSelectionRange r(1,1,4,4); //从左上角(1,1)到右下角(4,4)的范围

r.setRangeSelected(r, true); //选择 r 指定的范围

## 5、排序、查找、编辑、模型索引

20)、void **sortItems**(int *column*, Qt::SortOrder *order* = Qt::AscendingOrder)

使表格按照列 *column* 进行排序，排序规则(升序或降序)由 *order* 指定)。

21)、void **editItem**(QTableWidgetItem *\*item*) //开始编辑项目 *item*

22)、QList<QTableWidgetItem\*> **findItems**(const QString &*text*, Qt::MatchFlags *flags*) const

使用文本 *text* 查找项目，Qt::MatchFlag 枚举见第 4 章公共枚举章节

23)、QModelIndex **indexFromItem**(QTableWidgetItem *\*item*) const; //返回与 *item* 关联的模型索引。

24)、QTableWidgetItem **itemFromIndex**(const QModelIndex &*index*) const; //返回与 *index* 关联的项目

## 6、标头

25)、int **visualColumn**(int *logicalColumn*) const

返回逻辑列 *logicalColumn* 处的可视列(与 QHeaderView 类中的逻辑索引原理相同)。

26)、int **visualRow**(int *logicalRow*) const; //返回逻辑行 *logicalColumn* 处的可视行。

//以下函数用于设置标头，其原理与 QStandardItemView 类中的相应函数相同。

27)、QTableWidgetItem **horizontalHeaderItem**(int *column*) const

void **setHorizontalHeaderItem**(int *column*, QTableWidgetItem *\*item*) //设置水平标头的项目

void **setHorizontalHeaderLabels**(const QStringList &*labels*) //设置水平标头的文本

QTableWidgetItem **verticalHeaderItem**(int *row*) const

```

void setVerticalHeaderLabels(const QStringList &labels)
void setVerticalHeaderItem(int row, QTableWidgetItem *item)
QTableWidgetItem *takeVerticalHeaderItem(int row) //移除而不删除
QTableWidgetItem *takeHorizontalHeaderItem(int column)

```

## 7、持久编辑器、部件、项目原型

```

28)、QWidget *cellWidget(int row, int column) const //返回单元格(row, column)处的部件
29)、void setCellWidget(int row, int column, QWidget *widget); //把部件设置到位置(row, column)处
30)、void removeCellWidget(int row, int column) //删除单元格(row, column)处的部件。
31)、void closePersistentEditor(QTableWidgetItem *item) //关闭持久编辑器(见 QAbstractItemView)
32)、void openPersistentEditor(QTableWidgetItem *item) //打开持久编辑器
33)、bool isPersistentEditorOpen(QTableWidgetItem *item) const //持久编辑器是否打开。 qt5.10
34)、const QTableWidgetItem *itemPrototype() const //返回该表格的项目原型
35)、void setItemPrototype(const QTableWidgetItem *item)

```

设置 item 为该表格的项目原型。该表格取得项目原型的所有权。

## 8、槽

```

35)、void clear() //删除视图中的所有项目(包含标头)
36)、void clearContents() //删除视图中的数据项(不含标头)
37)、void insertColumn(int column) //在 column 处插入一个空列
38)、void insertRow(int row) //在 row 处插入一个空行
39)、void removeColumn(int column) //移除列 column 及其上的项目
40)、void removeRow(int row) ///移除行 row 及其上的项目
41)、void scrollToItem(const QTableWidgetItem *item, QAbstractItemView::ScrollHint hint = EnsureVisible)

```

滚动视图以使项目 item 可见。QAbstractItemView::ScrollHint 枚举见相关类

## 9、信号

```

1)、void cellActivated(int row, int column) 激活单元格时发送。
void itemActivated(QTableWidgetItem *item) //项目被激活时发送。
windows 下激活是指在单元格上按下回车键，具体取决于系统。
2)、void cellClicked(int row, int column) //单击单元格时发送
void itemClicked(QTableWidgetItem *item) //单击表格中的项目就发送此信号
3)、void cellPressed(int row, int column) //单击单元格时发送
void itemPressed(QTableWidgetItem *item) //当按下表格中的项目时，发送此信号
4)、void cellDoubleClicked(int row, int column) //双击单元格时发送
void itemDoubleClicked(QTableWidgetItem *item) //双击表格中的项目就发送此信号
5)、void cellEntered(int row, int column) //鼠标进入(需开启鼠标跟踪)单元格时发送
void itemEntered(QTableWidgetItem *item) //当鼠标光标进入项目时(需开启鼠标跟踪)发送
6)、void cellChanged(int row, int column) //单元格中的数据发生改变时发送。
void itemChanged(QTableWidgetItem *item) //当项目的数据项变化时，发送此信号。
7)、void currentCellChanged(int currentRow, int currentColumn, int previousRow, int previousColumn)

```

当前单元格发生变化时，发送此信号。

8)、void **currentItemChanged**(QTableWidgetItem \**current*, QTableWidgetItem \**previous*)

当当前项目发生变化时，发送此信号。

9)、void **itemSelectionChanged**() //当选择改变时，发送此信号

## 10、受保护的函数(主要用于处理拖放，与 QAbstractItemModel 类中的相应函数类似)

virtual bool **dropMimeData**(int *row*, int *column*, const QMimeData \**data*, Qt::DropAction *action*)

virtual QMimeData \* **mimeData**(const QList<QTableWidgetItem \*> *items*) const

virtual QStringList **mimeTypes**() const

virtual Qt::DropActions **supportedDropActions**() const

## 二、QTableWidgetItem 类

1、QTableWidgetItem 是一个独立的类，该类用于向 QTableWidgetItem 类提供数据项

2、**QTableWidgetItem**(int *type* = Type) //构造函数

**QTableWidgetItem**(const QString &*text*, int *type* = Type)

**QTableWidgetItem**(const QIcon &*icon*, const QString &*text*, int *type* = Type)

**QTableWidgetItem**(const QTableWidgetItem &*other*)

注意：参数 *type* 指的是类型，而不是值，该类型是枚举 ItemType(见 type()函数)描述的

3、以下函数主要用于设置项目数据，在前面章节已见过，下面以表格的形式列出

| 设置函数                                                         | 读取函数                                      | 说明             |
|--------------------------------------------------------------|-------------------------------------------|----------------|
| void <b>setText</b> (const QString & <i>text</i> )           | QString <b>text</b> () const              | 文本             |
| void <b>setTextAlignment</b> (int <i>alignment</i> )         | int <b>textAlignment</b> () const         | 文本对齐方式         |
| void <b>setBackground</b> (const QBrush & <i>brush</i> )     | QBrush <b>background</b> () const         | 背景色            |
| void <b>setForeground</b> (const QBrush & <i>brush</i> )     | QBrush <b>foreground</b> () const         | 前景色            |
| void <b>setIcon</b> (const QIcon & <i>icon</i> )             | QIcon <b>icon</b> () const                | 图标             |
| void <b>setFont</b> (const QFont & <i>font</i> )             | QFont <b>font</b> () const                | 字体             |
| void <b>setToolTip</b> (const QString & <i>toolTip</i> )     | QString <b>toolTip</b> () const           | 工具提示文本         |
| void <b>setStatusTip</b> (const QString & <i>statusTip</i> ) | QString <b>statusTip</b> () const         | 状态提示文本         |
| void <b>setWhatsThis</b> (const QString & <i>whatsThis</i> ) | QString <b>whatsThis</b> () const         | what's this 文本 |
| void <b>setSizeHint</b> (const QSize & <i>size</i> )         | QSize <b>sizeHint</b> () const            | 大小提示           |
| void <b>setCheckState</b> (Qt::CheckState <i>state</i> )     | Qt::CheckState <b>checkState</b> () const | 选中状态           |
| void <b>setSelected</b> (bool <i>select</i> )                | bool <b>isSelected</b> () const           | 选择状态           |
| void <b>setFlags</b> (Qt::ItemFlags <i>flags</i> )           | Qt::ItemFlags <b>flags</b> () const       | 项目标志           |

4、QTableWidgetItem \* **tableWidget**() const //返回该项目的表格部件

5、int **column**() const//返回表格中项目所在的列，若项目不在表格中，则返回-1。

6、int **row**() const //返回表格中项目所在的行，若项目不在表格中，则返回-1。

7、virtual QVariant **data**(int *role*) const

virtual void **setData**(int *role*, const QVariant &*value*)

把角色 *role* 的数据设置为 *value*，该函数可用于添加整型、浮点型等类型的数据。

8、virtual void **read**(QDataStream &*in*); //从流中读取项目

9、virtual void **write**(QDataStream &*out*) const; //把项目写入流中

10、virtual QTableWidgetItem \***clone**() const; //创建该项目的副本。

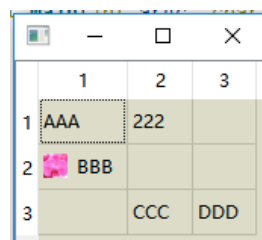
## 11、int type() const

返回该数据项的类型，返回的类型主要用于区分是否是自定义类型，返回的值应是枚举 ItemType 的成员所对应的整数值，见下表

| QTableWidgetItem::ItemType 枚举 |      |                          |
|-------------------------------|------|--------------------------|
| 成员                            | 值    | 说明                       |
| QTableWidgetItem::Type        | 0    | 数据项的默认类型                 |
| QTableWidgetItem::UserType    | 1000 | 自定义类型的最小值，小于此值的值由 Qt 保留。 |

## 示例：QTableWidget 部件的使用(效果见图)

```
#include<QtWidgets>
int main(int argc, char *argv[]) { QApplication app(argc, argv);
 //创建一个 3 行 3 列的表格
 QTableWidget *pt=new QTableWidget(3,3);
 QTableWidgetItem *pi1=new QTableWidgetItem("AAA");
 QTableWidgetItem *pi2=new QTableWidgetItem;
 QTableWidgetItem *pi3=new QTableWidgetItem;
 QTableWidgetItem *pi4=new QTableWidgetItem("CCC");
 QTableWidgetItem *pi5=new QTableWidgetItem("DDD");
 //整数值需使用 setData() 函数添加
 pi2->setData(Qt::DisplayRole, 222);
 pi3->setIcon(QIcon("F:/li.png"));
 pi3->setText("BBB");
 //把数据项添加到表格
 pt->setItem(0,0,pi1); pt->setItem(0,1,pi2); pt->setItem(1,0,pi3);
 pt->setItem(2,1,pi4); pt->setItem(2,2,pi5);
 //开启拖放，这两函数位于 QTableWidget 类的父类
 pt->setDragEnabled(1);pt->setAcceptDrops(1);
 pt->resize(333,222); pt->show(); return app.exec(); }
```





## 8.18 QListWidget 类(列表部件)

## 一、QListWidget 类基础

- 1、QListWidget 类继承自 QListView，该类是一个由 Qt 实现的标准的列表部件，该类的数据项由 QListWidgetItem 类管理。
- 2、使用 QListWidget 部件的方法如下

- 方法 1: 在创建 `QListWidgetItem` 项目时直接把项目添加到列表部件

```
QListWidget *pt = new QListWidget;
QListWidgetItem *pi1= QListWidgetItem("AAA", pt);
QListWidgetItem *pi2= QListWidgetItem("BBB", pt);
```

- 方法 2：创建 `QListWidgetItem` 项目后使用 `QListWidget` 类的添加或插入函数向 `QListWidget` 部件中添加项目。

```
QListWidget *pt = new QListWidget;
QListWidgetItem *pi1 = QListWidgetItem("AAA");
QListWidgetItem *pi2 = QListWidgetItem("BBB");
pt->addItem(pi1);
pt->addItem("CCC");
pt->insertItem(1, pi2);
```

## 二、QListWidget 类的属性和函数

## 1、属性和构造函数

- ```

1)、count: const int           访问函数: int count() const;           //返回列表中的项目数量, 包括隐藏项目
2)、currentRow: int           //描述当前行
   访问函数: int currentRow() const;   void setCurrentRow(int);
   void setCurrentRow(int row, QItemSelectionModel::SelectionFlags command);
   信号: void currentRowChanged(int);
3)、sortingEnabled: bool           //是否启用排序, 默认为 false(不启用)
   访问函数: bool isSortingEnabled() const;   void setSortingEnabled(bool);
4)、QListWidget(QWidget *parent = Q_NULLPTR)           //构造函数

```

2、设置、移除项目

- 5)、void **addItem**(QListWidgetItem **item*) //在末尾追加项目
- void **addItem**(const QString &*label*);
- void **addItems**(const QStringList &*labels*)
- 6)、void **insertItem**(int *row*, QListWidgetItem **item*) //在 row 处插入项目
- void **insertItem**(int *row*, const QString &*label*)
- void **insertItems**(int *row*, const QStringList &*labels*)
- 7)、QListWidgetItem ***takeItem**(int *row*) //移除而不删除 row 处的项目

说明：项目被添加到列表部件后，列表部件将获得项目的所有权。同一个项目只能添加到

QListWidget 中一次，多次相同的添加会导致未定义的行为。

3、获取列表部件的基本信息

- 8)、int **row**(const QListWidgetItem **item*) const //返回项目 item 所在的行。
- 9)、QListWidgetItem **item*(int **row**) const //返回行 row 处的项目，否则返回 0。
- 10)、QListWidgetItem **itemAt*(const QPoint &*p*) const //返回位置 p 处的项目。使用内容坐标。
- 11)、QListWidgetItem **itemAt*(int *x*, int *y*) const //返回位置(x, y)处的项目。使用内容坐标。
- 12)、QRect **visualItemRect**(const QListWidgetItem **item*) const
返回项目 item 在视口上所占用的矩形(即位置和大小)。
- 13)、QList<QListWidgetItem*> **items**(const QMimeData **data*) const; //返回包含在 data 中的项目列表。

4、当前项目和当前行

- 14)、QListWidgetItem **currentItem*() const
void **setCurrentItem**(QListWidgetItem **item*)
void **setCurrentItem**(QListWidgetItem **item*, QItemSelectionModel::SelectionFlags *command*)
- 15)、int **currentRow**() const
void **setCurrentRow**(int *row*)
void **setCurrentRow**(int *row*, QItemSelectionModel::SelectionFlags *command*)
以上函数用于设置当前行或当前项目，除非选择模式为 NoSelection，否则当前项目会同时被选中。其中最后 3 个函数是 **currentRow** 属性的函数，列在此处主要是为了与当前项目相区别。

5、选择项目、排序

- 16)、QList<QListWidgetItem*> **selectedItems**() const //返回列表部件中所有被选择项目的列表。
- 17)、void **sortItems**(Qt::SortOrder *order* = Qt::AscendingOrder)
使所有项目按照顺序 order 进行排序(升序或降序)。需开启 **sortingEnabled** 属性。

6、编辑、查找、模型索引

- 18)、void **editItem**(QListWidgetItem **item*); //开始编辑项目 item
- 19)、QList<QListWidgetItem*> **findItems**(const QString &*text*, Qt::MatchFlags *flags*) const
使用文本 text 查找项目，Qt::MatchFlag 枚举见第 4 章公共枚举章节
- 20)、QModelIndex **indexFromItem**(QListWidgetItem **item*) const; //返回与 item 关联的模型索引。
QListWidgetItem **itemFromIndex*(const QModelIndex &*index*) const; //返回与 index 关联的项目

7、持久编辑器、部件

- 21)、void **closePersistentEditor**(QListWidgetItem **item*); //关闭持久编辑器(见 QAbstractItemView)
bool **isPersistentEditorOpen**(QListWidgetItem **item*) const; //持久编辑器是否打开。qt5.10
void **openPersistentEditor**(QListWidgetItem **item*) //打开持久编辑器
- 22)、void **setItemWidget**(QListWidgetItem **item*, QWidget **widget*) //把部件设置到项目 item 中显示
QWidget **itemWidget*(QListWidgetItem **item*) const //返回项目 item 处显示的部件
void **removeItemWidget**(QListWidgetItem **item*)//删除项目 item 处的部件。

8、槽

- 23)、void **clear**(); //删除所有项目，注意：项目会被永久删除。
- 24)、void **scrollToItem**(const QListWidgetItem *item, QAbstractItemView::ScrollHint hint=EnsureVisible)
滚动视图以使项目 item 可见。QAbstractItemView::ScrollHint 枚举见相关类

9、信号

- 1)、void **currentRowChanged**(int currentRow)当前行变化时，发送此信号
- 2)、void **currentItemChanged**(QListWidgetItem *current, QListWidgetItem *previous); //当前项变化时发送
- 3)、void **currentTextChanged**(const QString ¤tText) //当前项变化时，发送此信号
- 4)、void **itemActivated**(QListWidgetItem *item)
当项目激活时发送此信号，具体激活方式取决于系统，在 windows 上是按下 Return 键。
- 5)、void **itemChanged**(QListWidgetItem *item) //当项目的数据变化时发送。
- 6)、void **itemClicked**(QListWidgetItem *item) //当单击项目时发送
- 7)、void **itemDoubleClicked**(QListWidgetItem *item) //当双击项目时发送。
- 8)、void **itemEntered**(QListWidgetItem *item) //当鼠标光标进入项目时发送(需开启鼠标跟踪)
- 9)、void **itemPressed**(QListWidgetItem *item) //当按下鼠标按钮时发送。
- 10)、void **itemSelectionChanged**() //当选择改变时发送。

10、受保护的函数(主要用于处理拖放，与 QAbstractItemModel 类中的相应函数类似)

virtual bool **dropMimeData**(int index, const QMimeData *data, Qt::DropAction action)
virtual QMimeData ***mimeData**(const QList<QListWidgetItem *> items) const
virtual QStringList **mimeTypes**() const
virtual Qt::DropActions **supportedDropActions**() const

三、QListWidgetItem 类的属性和函数

- 1、QListWidgetItem 是一个独立的类，该类用于向 QListWidget 类提供数据项
- 2、**QListWidgetItem**(QListWidget *parent = Q_NULLPTR, int type = Type)
QListWidgetItem(const QString &text, QListWidget *parent = Q_NULLPTR, int type = Type)
QListWidgetItem(const QIcon &icon, const QString &text, QListWidget *parent = Q_NULLPTR, int type = Type)
构造函数创建的项目可以直接添加到由 parent 指定的列表部件中。参数 type 指的是类型，而不是值，该类型是枚举 ItemType(见 type()函数)描述的
- 3、以下函数主要用于设置项目数据，在前面章节已见过，下面以表格的形式列出

设置函数	读取函数	说明
void setText (const QString &text)	QString text () const	文本
void setTextAlignment (int alignment)	int textAlignment () const	文本对齐方式
void setBackground (const QBrush &brush)	QBrush background () const	背景色
void setForeground (const QBrush &brush)	QBrush foreground () const	前景色
void setIcon (const QIcon &icon)	QIcon icon () const	图标

void setFont (const QFont & <i>font</i>)	QFont font () const	字体
void setToolTip (const QString & <i>toolTip</i>)	QString toolTip () const	工具提示文本
void setStatusTip (const QString & <i>statusTip</i>)	QString statusTip () const	状态提示文本
void setWhatsThis (const QString & <i>whatsThis</i>)	QString whatsThis () const	what's this 文本
void setSizeHint (const QSize & <i>size</i>)	QSize sizeHint () const	大小提示
void setCheckState (Qt::CheckState <i>state</i>)	Qt::CheckState checkState () const	选中状态
void setSelected (bool <i>select</i>)	bool isSelected () const	选择状态
void setFlags (Qt::ItemFlags <i>flags</i>)	Qt::ItemFlags flags () const	项目标志
void setHidden (bool <i>hide</i>)	bool isHidden () const	是否隐藏该项

4、QListWidget ***listWidget**() const; //返回该项目的列表部件

5、virtual QVariant **data**(int *role*) const

virtual void **setData**(int *role*, const QVariant &*value*)

把角色 role 的数据设置为 value，该函数可用于添加整型、浮点型等类型的数据。

6、virtual void **read**(QDataStream &*in*); //从流中读取项目

7、virtual void **write**(QDataStream &*out*) const; //把项目写入流中

8、virtual QListWidgetItem ***clone**() const; //创建该项目的副本。

9、、int **type**() const

返回该数据项的类型，返回的类型主要用于区分是否是自定义类型，返回的值应是枚举 ItemType 的成员所对应的整数值，见下表

QListWidgetItem::ItemType 枚举		
成员	值	说明
QListWidgetItem::Type	0	数据项的默认类型
QListWidgetItem::UserType	1000	自定义类型的最小值，小于此值的值由 Qt 保留。

8.19 QTreeWidgetItem 类(树部件)

一、QTreeWidgetItem 类基础

1、QTreeWidgetItem 类继承自 QTreeView，该类是一个由 Qt 实现的标准的树部件，该类的数据项由 QTreeWidgetItem 类管理。

QTreeWidgetItem 的数据项由两部分构建，QTreeWidgetItem 用于创建顶级项目，QTreeWidgetItem 用于创建子级项目。

2、使用 QTreeWidgetItem 部件的方法见后文对 QTreeWidgetItem 构造函数的使用。

二、QTreeWidgetItem 类的属性和函数

1、属性和构造函数

- 1)、**columnCount**: int **访问函数**: int columnCount() const; void setColumnCount(int);
树部件显示的列数，默认为 1。
- 2)、**topLevelItemCount**: const int **访问函数**: int topLevelItemCount() const;
返回顶级项目的数量，默认为 0。
- 3)、**QTreeWidgetItem**(QWidget *parent = Q_NULLPTR) //构造函数

2、设置、移除项目

- 4)、void **addTopLevelItem**(QTreeWidgetItem *item)
void **addTopLevelItems**(const QList<QTreeWidgetItem*> &items)
以上函数表示把项目 item 或项目列表 items 追加为部件的顶级项目。
- 5)、void **insertTopLevelItem**(int index, QTreeWidgetItem *item)
void **insertTopLevelItems**(int index, const QList<QTreeWidgetItem*> &items)
以上函数表示把项目 item 或项目列表 items 插入到部件顶级项目的索引 index 处。
- 6)、QTreeWidgetItem ***takeTopLevelItem**(int index)
移除而不删除 index 处的顶级项目，并返回该项目。

3、获取树部件的基本信息

- 7)、QTreeWidgetItem ***itemAbove**(const QTreeWidgetItem *item) const; //返回 item 上方的项目
QTreeWidgetItem ***itemBelow**(const QTreeWidgetItem *item) const; //返回 item 下方的可见项目
- 8)、QTreeWidgetItem ***itemAt**(const QPoint &p) const; //返回位置 point 处的项目。使用内容坐标。
QTreeWidgetItem ***itemAt**(int x, int y) const; //返回位置(x, y)处的项目。使用内容坐标。
- 9)、QTreeWidgetItem ***topLevelItem**(int index) const;
返回索引 index 处的顶层项目，若该项目不存在，则返回 0。
- 10)、QRect **visualItemRect**(const QTreeWidgetItem *item) const
返回项目 item 在视口上所占用的矩形(即位置和大小)。
- 11)、int **indexOfTopLevelItem**(QTreeWidgetItem *item) const; //返回顶级项目 item 的索引。
- 12)、QTreeWidgetItem ***invisibleRootItem**() const;

返回不可见根项目。不可见根项目用于对树部件顶层项目的访问。

4、当前项目和当前列

13)、int **currentColumn**() const;

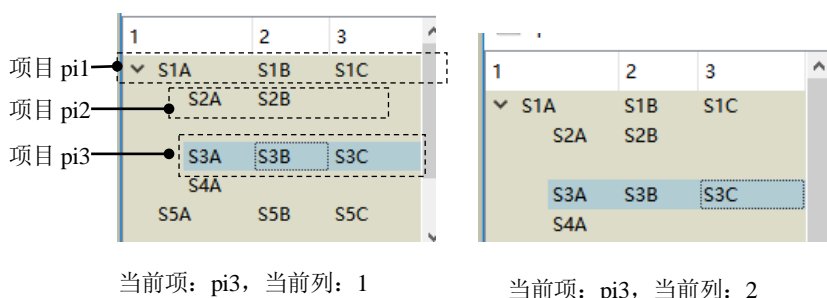
QTreeWidgetItem * **currentItem**() const;

void **setCurrentItem**(QTreeWidgetItem **item*)

void **setCurrentItem**(QTreeWidgetItem **item*, int *column*)

void **setCurrentItem**(QTreeWidgetItem **item*, int *column*, QItemSelectionModel::SelectionFlags *command*)

以上函数用于设置或获取当前列和当前项目，除非选择模式为 NoSelection，否则当前项目会同时被选中，原理见下图。



5、选择项目、排序、跨列

14)、QList<QTreeWidgetItem*> **selectedItems**() const; //返回所有被选择的非隐藏项目的列表。

15)、void **sortItems**(int *column*, Qt::SortOrder *order*)

使项目按照列 *column* 进行排序(升序或降序，由 *order* 指定)。

16)、int **sortColumn**() const; //返回用于对项目进行排序的列。

17)、bool **isFirstItemColumnSpanned**(const QTreeWidgetItem **item*) const

若项目 *item* 跨越所有列，则返回 true，否则返回 false。

18)、void **setFirstItemColumnSpanned**(const QTreeWidgetItem **item*, bool *span*)

若 *span* 为 true，则把父级 *item* 设置为跨越所有列，否则项目显示在各自的列中。原理见 QTreeView 类中相应函数。

6、编辑、查找、模型索引

19)、void **editItem**(QTreeWidgetItem **item*, int *column* = 0); //开始编辑列 *column* 中的项目 *item*。

20)、QList<QTreeWidgetItem*> **findItems**(const QString &*text*, Qt::MatchFlags *flags*, int *column* = 0) const

使用文本 *text* 在列 *column* 中查找项目，Qt::MatchFlag 枚举见第 4 章公共枚举章节

21)、QModelIndex **indexFromItem**(const QTreeWidgetItem **item*, int *column* = 0) const

返回列 *column* 中与 *item* 关联的模型索引。

22)、QTreeWidgetItem **itemFromIndex**(const QModelIndex &*index*) const

返回与模型索引 *index* 关联的项目

7、持久编辑器、部件

- 23)、void **closePersistentEditor**(QTreeWidgetItem *item, int column = 0);
 bool **isPersistentEditorOpen**(QTreeWidgetItem *item, int column = 0) const; //t5.10
 void **openPersistentEditor**(QTreeWidgetItem *item, int column = 0);
 以上函数分别表示，关闭、打开持久编辑器，以及判断持久编辑器是否打开。持久编辑器见 QAbstractItemView
- 24)、void **setItemWidget**(QTreeWidgetItem *item, int column, QWidget *widget)
 把部件设置到项目 item 和列 column 指定的单元格中显示。调用该函数之前，应把 item 添加到树部件中。树部件获得部件 widget 的所有权
- 25)、QWidget ***itemWidget**(QTreeWidgetItem *item, int column) const
 返回由项目 item 和列 column 指定的单元格中显示的部件
- 26)、void **removeItemWidget**(QTreeWidgetItem *item, int column)
 删除由项目 item 和列 column 指定的单元格的部件。

8、标头

//以下函数用于设置标头，其原理与 QStandardItemView 类中的相应函数相同。

- 27)、QTreeWidgetItem ***headerItem**() const; //返回标头的项目
 void **setHeaderItem**(QTreeWidgetItem *item); //设置标头的项目
 void **setHeaderLabel**(const QString &label);
 void **setHeaderLabels**(const QStringList &labels);

9、槽

- 28)、void **clear**() //清除树部件的所有项目。
- 29)、void **collapseItem**(const QTreeWidgetItem *item); //折叠项目 item。
 void **expandItem**(const QTreeWidgetItem *item); //展开项目 item。
- 30)、void **scrollToItem**(const QTreeWidgetItem *item, QAbstractItemView::ScrollHint hint = EnsureVisible)
 滚动视图以使项目 item 可见。QAbstractItemView::ScrollHint 枚举见相关类

10、信号

- 1)、void **currentItemChanged**(QTreeWidgetItem *current, QTreeWidgetItem *previous)
 当前项目更改时发送此信号。
- 2)、void **itemActivated**(QTreeWidgetItem *item, int column)
 当项目激活时发送此信号，具体激活方式取决于系统，在 windows 上是按下 Return 键。
- 3)、void **itemClicked**(QTreeWidgetItem *item, int column); //点击项目时发送。
- 4)、void **itemDoubleClicked**(QTreeWidgetItem *item, int column) //当双击项目时发送。
- 5)、void **itemPressed**(QTreeWidgetItem *item, int column) //当在项目上按下按钮时发送。
- 6)、void **itemEntered**(QTreeWidgetItem *item, int column)
 当鼠标光标进入项目时发送(需开启鼠标跟踪)
- 7)、void **itemCollapsed**(QTreeWidgetItem *item)
 void **itemExpanded**(QTreeWidgetItem *item)
 以上信号表示，当项目拆叠或展开时发送。注意：当 QTreeView::collapseAll()被调用后，

然后折叠或展开项目，不会发送相应信号。

8)、void **itemChanged**(QTreeWidgetItem **item*, int *column*) //当项目更改时发送。

9)、void **itemSelectionChanged**() //当选择发生更改时发送。

11、受保护的函数(主要用于处理拖放，与 QAbstractItemModel 类中的相应函数类似)

virtual bool **dropMimeData**(QTreeWidgetItem **parent*, int *index*, const QMimeData **data*, Qt::DropAction *action*);

virtual QMimeData ***mimeData**(const QList<QTreeWidgetItem *> *items*) const

virtual QStringList **mimeTypes**() const

virtual Qt::DropActions **supportedDropActions**() const

三、QTreeWidgetItem 类的属性和函数

1、QTreeWidgetItem 是一个独立的类，该类用于向 QTreeWidget 类提供数据项

2、以下函数主要用于设置项目数据，在前面章节已见过，下面以表格的形式列出

	设置函数	读取函数
以下各项的含义如下：1、文本，2、文本对齐，3、背景色，4、前景色，5、图标，6、字体，7、选中状态，8、选择状态，9、是否隐藏，10、是否禁用，11、是否展开，12、是否扩展第1列，13、工具提示，14、状态提示，15、what's this 文本，16、大小提示，17、项目标志		
1	void setText (int <i>column</i> , const QString & <i>text</i>)	QString text (int <i>column</i>) const
2	void setTextAlignment (int <i>column</i> , int <i>alignment</i>)	int textAlignment (int <i>column</i>) const
3	void setBackground (int <i>column</i> , const QBrush & <i>brush</i>)	QBrush background (int <i>column</i>) const
4	void setForeground (int <i>column</i> , const QBrush & <i>brush</i>)	QBrush foreground (int <i>column</i>) const
5	void setIcon (int <i>column</i> , const QIcon & <i>icon</i>)	QIcon icon (int <i>column</i>) const
6	void setFont (int <i>column</i> , const QFont & <i>font</i>)	QFont font (int <i>column</i>) const
7	void setCheckState (int <i>column</i> , Qt::CheckState <i>state</i>)	Qt::CheckState checkState (int <i>col</i>) const
8	void setSelected (bool <i>select</i>)	bool isSelected () const
9	void setHidden (bool <i>hide</i>)	bool isHidden () const
10	void setDisabled (bool <i>disabled</i>)	bool isDisabled () const
11	void setExpanded (bool <i>expand</i>)	bool isExpanded () const
12	void setFirstColumnSpanned (bool <i>span</i>)	bool isFirstColumnSpanned () const
13	void setToolTip (int <i>column</i> , const QString & <i>toolTip</i>)	QString toolTip (int <i>column</i>) const
14	void setStatusTip (int <i>column</i> , const QString & <i>statusTip</i>)	QString statusTip (int <i>column</i>) const
15	void setWhatsThis (int <i>column</i> , const QString & <i>whatsThis</i>)	QString whatsThis (int <i>column</i>) const
16	void setSizeHint (int <i>column</i> , const QSize & <i>size</i>)	QSize sizeHint (int <i>column</i>) const
17	void setFlags (Qt::ItemFlags <i>flags</i>)	Qt::ItemFlags flags () const

3、构造函数

以下的参数 *type* 指的是类型，而不是值，该类型是枚举 *ItemType*(见 *type()* 函数)描述的

1)、**QTreeWidgetItem**(int *type* = *Type*)

QTreeWidgetItem(const QStringList &*strings*, int *type* = *Type*)

构造一个项目，在之后被添加到树部件中。

2)、**QTreeWidgetItem**(QTreeWidget **parent*, int *type* = *Type*)

QTreeWidgetItem(QTreeWidget **parent*, const QStringList &*strings*, int *type* = *Type*)

构造一个项目，该项目被直接添加到树部件 *parent* 之中。

3)、**QTreeWidgetItem**(QTreeWidgetItem **parent*, int *type* = *Type*)

QTreeWidgetItem(QTreeWidgetItem *parent, const QStringList &strings, int type = Type)

构造一个项目，该项目被添加到项目(或节点)parent 之中作为其子项目。

4)、**QTreeWidgetItem**(QTreeWidgetItem *parent, QTreeWidgetItem *preceding, int type = Type)

QTreeWidgetItem(QTreeWidgetItem *parent, QTreeWidgetItem *preceding, int type = Type)

以上两个函数用于构造一个空项目，该项目位于父级 parent 的子项目 preceding 之后。

示例：QTreeWidgetItem 构造函数的使用(效果见图示)

```
QTreeWidgetItem *pi1,*pi2,*pi3,*pi4,*pi5,*pi6,*pi7,*pi8;  
QTreeWidgetItem *pt;           QStringList s1,s2,s3,s4,s5,s6;
```

//初始化列表

```
s1<<"S1A"<<"S1B"<<"S1C";    s2<<"S2A"<<"S2B";
```

```
s3<<"S3A"<<"S3B"<<"S3C";    s4<<"S4A";
```

```
s5<<"S5A"<<"S5B"<<"S5C";    s6<<"S6A";
```

//初始化各部件

```
pt=new QTreeWidgetItem;        //创建一个树部件
```

```
pt->setColumnCount(3);          //设置树部件的列数
```

//向父项目(或父节点)pt 之中添加 3 个子项目。

```
pi1=new QTreeWidgetItem(pt, s1);
```

```
pi5=new QTreeWidgetItem(pt, s5);
```

```
pi6=new QTreeWidgetItem(pt, s6);
```

//向父项目(或父节点)pi1 之中添加 3 个子项目。

```
pi2=new QTreeWidgetItem(pi1, s2);
```

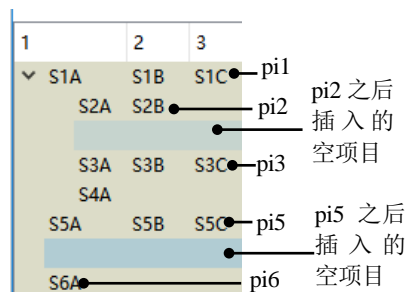
```
pi3=new QTreeWidgetItem(pi1, s3);
```

```
pi4=new QTreeWidgetItem(pi1, s4);
```

```
pi7=new QTreeWidgetItem(pi1, pi2);    //在 pi1 的子项目 pi2 之后插入一个空项目
```

```
pi8=new QTreeWidgetItem(pt, pi5);     //在 pt 的子项目 pi5 之后插入一个空项目
```

```
pt->resize(333, 222);    pt->show();
```



4、添加、插入、删除、移除子项

5)、void **addChild**(QTreeWidgetItem *child) //向该项目添加子项

void **addChildren**(const QList<QTreeWidgetItem *> &children)

6)、void **insertChild**(int index, QTreeWidgetItem *child)

void **insertChildren**(int index, const QList<QTreeWidgetItem *> &children)

把子项(child 或 children)插入到索引 index 处，若子项已插入其他地方，则不会再被插入。

7)、QList<QTreeWidgetItem *> **takeChildren**()

QTreeWidgetItem ***takeChild**(int index)

移除而不删除该项目的子项，并返回子项

8)、void **removeChild**(QTreeWidgetItem *child)

移除而不删除子项 child，注意：该函数也不会删除(销毁)child。

9)、virtual QVariant **data**(int column, int role) const; //setData()的读取函数

virtual void **setData**(int column, int role, const QVariant &value)

把该项目位于列 column 的数据项的角色 role 的数据设置为 value，该函数可用于添加整型、浮点型等类型的数据。

5、获取项目的信息

- 10)、QTreeWidgetItem *parent() const; //返回该项目的父项目
- 11)、QTreeWidgetItem *treeWidget() const; //返回包含该项目的树部件
- 12)、int columnCount() const; //返回该项目的列数
- 13)、QTreeWidgetItem *child(int index) const; //返回该项目位于索引 index 处的子项目
- 14)、int indexOfChild(QTreeWidgetItem *child) const; //返回子项 child 的索引。
- 15)、int childCount() const //返回该项目拥有的子项目的数量。

6、展开/折叠指示符、排序

- 16)、QTreeWidgetItem::ChildIndicatorPolicy childIndicatorPolicy() const
void setChildIndicatorPolicy(QTreeWidgetItem::ChildIndicatorPolicy policy)
以上函数用于设置或获取项目指示符策略，该策略决定何时显示树的展开/折叠指示符。
枚举 QTreeWidgetItem::ChildIndicatorPolicy 见下表

QTreeWidgetItem::ChildIndicatorPolicy 枚举(无标志)		
成员	值	说明
QTreeWidgetItem::ShowIndicator	0	即使没有子项，也显示展开/折叠指示符
QTreeWidgetItem::DontShowIndicator	1	即使有子项，也不会显示展开/折叠指示符
QTreeWidgetItem::DontShowIndicatorWhenChildless	2	若有子项，则显示展开/折叠指示符

- 17)、void sortChildren(int column, Qt::SortOrder order)
使子项按照列 column 进行排序，排序规则(升序或降序)由 order 指定)。

7、其他

- 18)、virtual void read(QDataStream &in); //从流中读取项目
- 19)、virtual void write(QDataStream &out) const; //把项目写入流中
- 20)、virtual QTreeWidgetItem *clone() const; //创建该项目的副本。
- 21)、int type() const
返回该数据项的类型，返回的类型主要用于区分是否是自定义类型，返回的值应是枚举 ItemType 的成员所对应的整数值，见下表

QTreeWidgetItem::ItemType 枚举		
成员	值	说明
QTreeWidgetItem::Type	0	数据项的默认类型
QTreeWidgetItem::UserType	1000	自定义类型的最小值，小于此值的值由 Qt 保留。

本文作者：黄邦勇帅(原名：黄勇)

本文要求读者已经非常熟悉 C++ 语法。若读者不熟悉 C++ 语法，推荐参阅《C++ 语法详解》(作者：黄勇)一书，电子工业出版社出版。

本文主要讲解了 Qt 的放施和剪贴板，本文列举了详细的示例进行说明，同时本文也是非常方便、快捷的编写 Qt 程序的查阅资料，可方便的查阅到相关内容的原理，以及怎样使用该内容。本文内容由浅入深，易学易懂。

本文使用的是 windows 10 操作系统，Qt 版本为 Qt5.10.1，Qt Creator 的版本为 Qt Creator 4.5.1 本文内容完全属于个人见解与参考文献的作者无关，限于水平有限，其中难免有误解之处，望指出更正。

声明：禁止抄袭、复印、转载本文，本文作者拥有完全版权。

主要参考文献：

- 1、C++语法详解 黄勇 编著 电子工业出版社 2017 年 7 月
- 2、Qt Creator 快速入门(第 3 版) 霍亚飞 编著 北京航空航天大学出版社 2017 年 1 月
- 3、C++ GUI Qt4 编程(第 2 版) [加拿大] Jasmin Blanchette [英] Mark Summerfield 著 闫锋欣 曾泉人 张志强 译 电子工业出版社 2008 年 8 月
- 4、Qt 5 开发及实例 陆文周 主编 电子工业出版社 2014 年 1 月

第9章 Qt 拖放、剪贴板

[9.1 拖放原理](#)

[9.1.1 拖放基本原理](#)

[9.1.2 拖放动作或称为放置动作](#)

[9.1.3 使用拖放打开文件](#)

[9.2 与拖放事件有关的类及函数](#)

[9.2.1 QDropEvent 类](#)

[9.2.2 QDragMoveEvent 类](#)

[9.2.3 QDragEnterEvent 类和 QDragLeaveEvent 类](#)

[9.2.4 QWidget 类中与拖放有关的函数](#)

[9.3 QDrag 类](#)

[9.4 QMimeData 类与拖放自定义类型数据](#)

[9.4.1 基本规则](#)

[9.4.2 QMimeData 类中的函数](#)

[9.4.3 子类化 QMimeData](#)

[9.4.4 重新实现 QMimeData 类中的虚函数](#)

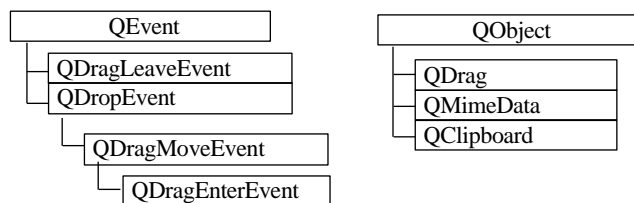
[9.5 QClipboard 类\(剪贴板\)](#)

第 9 部分 Qt 拖放与剪贴板

注意：本程序都假设读者在 pro 文件中已添加了正确的 `QT+=widgets` 语句，文中不再重复累述添加此语句。

本文注重讲解原理，因此使用的是手写的 Qt 程序。

本章讲解的类及继承关系如下图所示



本章所讲的类

9.1 拖放原理

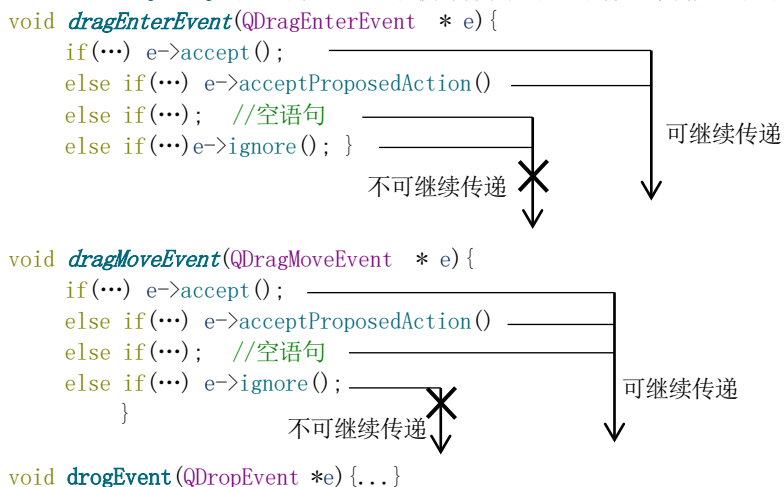
一、拖放基本原理

- 1、拖放操作包括两个动作：拖动(drag)和放下(drop 或称为放置)。当被拖动时拖动的数据会被存储为 MIME 类型(见第 6 章文件对话框)的对象，MIME 类型使用 `QMimeData` 类来描述。MIME 类型通常由剪贴板和拖放系统使用，以识别不同类型的数据。
- 2、拖动点(drag site)：拖动的起始位置。
- 3、放下点(drop site)：被拖动的对象放下的位置，若部件不能接受拖动的对象，Qt 会改变光标的形状(一个禁用形状)来向用户进行说明。
- 4、拖放的启动和结束：
 - 1)、启动拖放：拖放通过调用 `QDrag::exec()` 函数而启动，该函数是一个阻塞函数(但不会阻塞主事件循环)，这意味着在拖放操作结束之前，不会返回该函数，调用 `QDrag::exec()` 函数后，Qt 拥有对拖动对象的所有权，并会在必要时将其删除。
 - 2)、结束拖放：当用户放下拖动或取消拖动操作时结束拖放。
- 5、拖放产生的过程及事件
 - 1)、启动拖放后，会使数据被拖动，这时需要按住鼠标按键才能拖动需要拖动的数据，松开鼠标按键时意味着拖动结束。在这期间会产生如下事件
 - 2)、默认情况下，部件不接受放下事件。使用 `QWidget::setAcceptDrops()` 函数可设置部件是否接受放下事件(即，拖放完成时发送的事件)。只有在部件接受放下事件的情形下，才会产生以下事件(见下图)。

- 3)、QDragEnterEvent: 拖动进入事件。当拖动操作进入部件时, 该事件被发送到部件, 忽略该事件, 将会导致后续的拖放事件不能被发送。通常在该部件上光标会在外观上显示为禁用的图形。
- 4)、QDragMoveEvent: 拖动移动事件。当拖动操作正在进行时, 以及当具有焦点时按下键盘的修饰键(比如 Ctrl)时, 发送该事件, 要使部件能接收到该事件, 则该部件必须接受 QDragEnterEvent 事件。
- 5)、QDropEvent: 放下事件。在完成拖放操作时发送该事件, 即当用户在部件上放下一个对象时, 发送此事件。要使部件能接收到该事件, 则该部件必须接受 QDragEnterEvent 事件, 且不能忽略 QDragMoveEvent 事件。
- 6)、QLeaveEvent: 当拖放操作离开部件时发送该事件, 注意: 要使部件能接收到该事件, 必须要使拖动先进入该部件(即产生 QDragEnterEvent 事件), 然后再离开该部件, 才会产生 QLeaveEvent 事件。因很少使用该事件, 因此本文不做重点介绍。
- 7)、注: 必须接受是指必须重新实现该事件的处理函数并接受该事件, 不能忽略是指在事件处理函数中不明确调用 ignore() 函数忽略该事件, 这意味着可以不必重新实现该事件的处理函数。
- 8)、以上事件产生的顺序为: QDragEnterEvent、QDragMoveEvent、QDropEvent

事件传递规则图示, 注: acceptProposedAction() 也表示接受事件, 该函数见后文。

xxx->setAcceptDrops(true); //必须使部件接受放置事件, 才会产生以下事件



6、编写拖放程序的步骤

- 1)、在需要接受放下数据的部件上调用 QWidget::setAcceptDrops() 函数以使该部件能接受拖放事件。
- 2)、启动拖放: 通常在 mousePressEvent() 或 mouseMoveEvent() 函数中启动拖放, 记住启动拖放就是调用 QDrag 对象的 exec() 函数, 因此也可以在 keyPressEvent() 等函数中启动拖放(因很少这样做, 所以本文不介绍这种情况下的拖放)。在此步把需要拖动的数据保存在 QMimeData 对象中。
- 3)、重新实现需要接受放下数据的部件的 dragEnterEvent() 事件处理函数。

4)、根据需要重新实现 dragMoveEvent 或 dropEvent()函数

下面以实现代码为例进行讲解

示例：简单的拖放

//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>
class C:public QPushButton(Q_OBJECT)
public:    C(QString t="",QWidget *p=0):QPushButton(t,p) {}
    void mousePressEvent(QMouseEvent *e) { //在该事件中启动拖放
        QDrag *dg=new QDrag(this);
        //将需要拖动的数据放入 QMimeData 对象中，该对象用于保存需要传递的数据，数据的内
        //容完全由程序员自行设定。通常为界面上所选择内容。
        QMimeData *md=new QMimeData;
        md->setText("FFF"); //这是 md 中存储的内容(即拖放时传递的数据)。
        dg->setMimeData(md); //步骤 1：设置拖动的数据。该函数会获得 md 的所有权。
        dg->exec(); //步骤 2：启动拖放
    void dragEnterEvent(QDragEnterEvent * e) {
        //步骤 3：处理是否接受拖动事件。
        e->accept(); 接受拖动进入事件
        //e->ignore(); /*若忽略该事件，则不会再发送之后的事件，拖放至此结束，这会导致鼠
        标光标显示为禁用的图形。*/
    }
    void dropEvent(QDropEvent * e) {
        //步骤 4：处理拖动中的数据(当然也可不作任何处理)
        setText(e->mimeTypeData()->text()); //设置此部件的文本为拖动对象中的文本。
        //此事件不影响后续事件，可接受也可忽略。
        //e->accept();
        //e->ignore();
    }
};

class B:public QWidget{    Q_OBJECT
public: B(QWidget *p=0):QWidget(p) {
    C *pb1=new C("AAA",this);    pb1->move(22,22);    pb1->setIcon(QIcon("F:/li.png"));
    C *pb2=new C("BBB",this);    pb2->move(99,22);
    pb2->setAcceptDrops(true);    //pb2 接受放下事件
    pb1->setAcceptDrops(0);    //pb1 禁止放下事件。
}
};
#endif // M_H
```

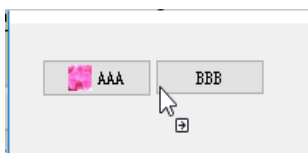
//m.cpp 文件的内容

```
#include "m.h"
int main(int argc, char *argv[]) {    QApplication app(argc,argv);
    B w;    w.resize(444,355);    w.show();    return app.exec(); }
```

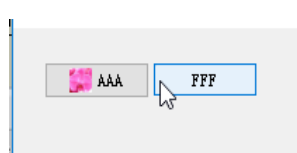
运行结果及说明



在按钮 AAA 上按下鼠标不放并拖动至如图所示位置，由于主窗口不接受放下事件，因此光标显示为禁用的图形



当光标移进入按钮 BBB 的范围时，会发送 QDragEnterEvent 事件，同时光标改变为以上形状，表示 BBB 可接受拖动的数据



当在 BBB 上放下鼠标按钮时，此时发送 QDropEvent 事件，按钮 BBB 的文本被修改为 FFF(即拖动对象中保存的数据)。至此拖动结束

本示例可把数据从按钮 AAA 拖至按钮 BBB

二、拖放动作(或称为放置动作)

1、拖放动作是指用户希望怎样处理拖放的数据，比如移动、复制、还是创建由目标到源的链接等。拖放动作由 Qt::DropAction 枚举描述(其取值见第 8 章)

2、可能的拖放动作，实际的拖放动作，建议的拖放动作

1)、**可能的拖放动作**：是指用户在拖放时可能会执行的拖放动作，用户在拖放时通常可由用户选择，比如可以选择移动、复制或链接等动作中的一种，这些动作都是可能的拖放动作。可能的拖放动作在 QDrag::exec()函数的第 1 个参数中指定，同时该函数的返回值是最终的实际拖放动作。

2)、**实际的拖放动作**：是指拖放被最终放置时实际执行的动作，实际拖放动作在 dropEvent()函数中使用 QDropEvent::setDropAction()函数(还需在之后调用 accept()函数)设置，该函数会影响 QDrag::exec()函数的返回值。

3)、**建议的拖放动作**：是指当用户执行拖动而不使用修饰键时的默认动作，建议拖放动作可在 QDrag::exec()函数的第 2 个参数中指定，该参数的设置会影响拖动时鼠标光标右下角的外观，另外 QDropEvent::acceptProposedAction()函数表示设置执行操作为建议操作并接受该事件。

4)、以上三种拖放动作常常相互关联，比如用户在拖动时通常可以执行移动、复制或链接等动作(可能的拖放动作)中的一种，然而应用程序在拖放被最终放置时并不知道用户到底需要执行哪种操作，若用户未指定需要执行的可能的拖放动作中的哪一种动作时，应用程序可以使用设置的建议动作，作为需要执行的动作。

3、各拖放动作之间的关系

1)、QDrag::exec()函数的规则

- 若 QDrag::exec()未指定建议拖放动作，则依顺序移动、复制、链接进行选择
- 若 QDrag::exec()函数在第 2 个参数上指定了建议拖放动作，但该动作不在可能的拖放动作组合之中，则使用默认的复制拖放动作。比如

```
QDrag *dg = new QDrag(this);
```

```
.....
```

```
dg->exec(Qt::MoveAction|Qt::CopyAction, Qt::LinkAction); //建议动作为复制。
```

2)、QDropEvent::setDropAction()函数的规则

- 使用 setDropAction()函数设置拖放动作之后应使用 accept()函数，而不应使用 QDropEvent::acceptProposedAction()函数(因为该函数会重置拖放动作为建议拖放动作)
- 若 QDropEvent::setDropAction()函数设置的拖放动作不在可能的拖放动作组合之中，则使用建议拖放动作。

3)、dropEvent()函数的规则，该函数是否接受事件直接影响到 QDrag::exec()函数的返回值，其规则如下

- 若在该函数内调用 ignore()，则 exec()函数返回 Qt::IgnoreAction
- 若在该函数内调用 accept()，则 exec()函数返回在该函数中使用 setDropAction()函数设置的拖放动作。详细规则见下面的示例程序。

各种拖放动作示例

```
Qt::DropAction d= QDrag::exe(Qt::MoveAction | Qt::MoveAction);    //可能的拖放动作
...
void dragEvent(QDropEvent *e) {
    //下面分情形讲解该函数对 exe() 返回值(即以上语句的 d)的影响
    //情形 1: d = Qt::MoveAction
    if(...) { e->setDropAction(Qt::MoveAction);      e->accept();}
    //情形 2: d = Qt::MoveAction
    else if(...) { e->setDropAction(Qt::CopyAction);      e->accept();}
    //情形 3: d = 建议的拖放动作
    else if(...) { e->acceptProposedAction();      }
    //情形 4: d = Qt::IgnoreAction
    else if(...) { e->ignore();      } //该部件不接受放置动作
    //情形 5: d = 建议的拖放动作
    else if(...) {
        e->setDropAction(Qt::MoveAction);
        e->acceptProposedAction(); /*使用该函数会把最终拖放动作重置为建议的拖放动作，因此调用 setDropAction() 之后应调用 accept() 函数。*/
        e->accept();}
    //情形 6: d = Qt::IgnoreAction
    else if(...) {
        e->setDropAction(Qt::MoveAction);
        e->ignore();} //该函数表示不接受放置操作，这会之前设置的拖放动无效。
    //情形 6: d = 建议的拖放动作，因为 Qt::LinkAction 不是可能的拖放动作之一。
    else if(...) {e->setDropAction(Qt::LinkAction);e->accept();} }
```

4、拖放动作及拖放的程序设计原则

- 1)、若在 mouseMoveEvent()函数中启动拖放，则可以编写避免用户因为手握鼠标抖动而产生的拖动，这比在 mousePressEvent()函数中启动拖放效果更好。
- 2)、在 QDrag::exe()函数的参数中指定可能的拖放动作，比如在其中同时指定移动、复制、链接等；但最终是否接受这些动作，由后续的事件处理函数进行判断，详见后文。另外需要注意的是 QDrag::exec()函数虽是阻塞函数，但在执行完该函数(比如释放鼠标按

钮完成拖放时)后程序会返回该函数，然后接着执行之后的语句，exec()函数返回的是用户实际执行的动作。

- 3)、dragEnterEvent()函数通常根据该部件或实际使用情况进行筛选，比如若该部件不接受图片数据，则忽略对该动作的接受，从而阻止事件被进一步传递。
- 4)、若重新实现了 dragMoveEvent()函数，则还可以在该函数内进行进一步的设置，比如默认为复制动作，若用户拖动时同时按下了 Shift 键，则设置为移动动作，若按下了 Ctrl 键，则为复制动作，若按下了 Alt 键则为链接动作等，在该函数中的设置可以影响鼠标光标在外观上的显示，比如复制会在光标右下角显示一个“+”符号等。另外，在该函数内还可以设置用户拖动到该部件中的范围，比如拖动到某矩形范围内时，该部件才接受拖放，否则被忽略等。注意：若在 dragEnterEvent()函数内也设置了拖放动作，同样会改变光标的外观但只会改变进入部件时的外观，光标最终的外观以 dragMoveEvent()函数设置的为准(因为该函数位于 dragEnterEvent()之后执行)。
- 5)、在 dropEvent()函数内最终决定对拖放的数据的处理，以及用户实际执行的拖放动作，因此该函数决定着 QDrag::exec()函数的返回值，这里要注意的是，对于移动动作，通常原始数据应由源部件(启动拖放的部件)进行删除，因此当 dropEvent()处理完数据之后，应把拖放动作设置为移动，QDrag::exec()函数会返回在 dropEvent()函数中设置的动作，然后源部件根据 QDrag::exec()的返回值是否是移动动作，而作出是否删除原始数据的决定。注：dragEnterEvent()和 dragMoveEvent()对拖放动作的设置不会影响 QDrag::exec()的返回值。
- 6)、注意：在源部件中创建的 QMimeData 和 QDrag 对象不应由程序员销毁，因为 Qt 会自动销毁，若程序员销毁了，则可能会出现多次 delete 同一个指针的错误。
- 7)、以上只是通常在各函数中的做法，当然你也可以不按照这些步骤来实现，从之前的拖放示例可以看到，对拖放的处理完全是任意的。
- 8)、注意：Qt 为某些部件提供了一些标准的拖放支持，在继承这些部件实现拖放时需要重新实现 dragEnterEvent()、dropEvent()，另外还可能需重新实现 dragMoveEvent()函数，以避免与标准实现的拖放支持相冲突或产生预料之外的结果。

示例：拖放动作

//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>
#include<QDebug>

class C:public QPushButton{Q_OBJECT
public:    QPoint p;    //用于保存点击鼠标的位置
    C(QString t="",QWidget *pp=0):QPushButton(t,pp) {}
    void mousePressEvent(QMouseEvent *e){    p=e->pos();    }    //第一次点击鼠标的位置
    void mouseMoveEvent(QMouseEvent *e){
        //若拖动的距离大于 5 个像素则启用拖放(避免抖动)，本例使用更简单直观的算法。
        if ((e->pos()-p).x()>=5|| (e->pos()-p).y()>=5) {
            QDrag *dg=new QDrag(this);    QMimeData *md=new QMimeData;
            md->setText("FFF");    dg->setMimeData(md);
            //启动拖放，该数据可复制、移动、链接，具体是否接受这些动作，需由后续程序决定。
        }
    }
};
```

```

//若 dragEvent() 函数把拖放动作设置为移动，则需要对原始数据作进一步处理，注意：在完成拖放
//后会返回 exec() 函数继续执行其后的语句。
    if(dg->exec(Qt::CopyAction|Qt::MoveAction|Qt::LinkAction)==Qt::MoveAction)
        qDebug()<<"AAA"; //本示例没有可删除的原始数据，因此只简单输出字符串。
    } //if 结束
//注：鼠标移动距离也可使用以下语句判断，其中 manhattanLength() 表示曼哈顿距离，startDragDistance
//表示系统推荐的拖动起始距离。
    //if((e->pos()-p).manhattanLength()>=QApplication::startDragDistance())
    {

void dragEnterEvent(QDragEnterEvent * e){
    //若拖动的数据中不包含文本 FFF 则忽略该事件，否则接受该事件。
    if(e->mimeTypeData()->text()!="FFF") e->ignore(); else e->accept(); }

void dragMoveEvent(QDragMoveEvent * e){
    /*以下设置会改变鼠标光标的外观。若拖动时同时按下了 CTRL、ALT、SHIFT 键，则把拖放动作设
    置为复制、移动、链接，否则为复制。*/
    if(e->keyboardModifiers()==Qt::CTRL) e->setDropAction(Qt::CopyAction);
    else if(e->keyboardModifiers()==Qt::SHIFT) e->setDropAction(Qt::MoveAction);
    else if(e->keyboardModifiers()==Qt::ALT) e->setDropAction(Qt::LinkAction);
    else e->setDropAction(Qt::CopyAction);
    //若光标位于矩形 r 之内，则接受该事件，否则忽略该事件。
    QRect r(0,0,111,33);
    if(r.contains(e->pos())){ e->accept(); } else e->ignore();
}

void dropEvent(QDropEvent * e){
    /*若拖动的源和目标在同一个部件，则什么也不做。注意应使用 return;跳出函数，若使用 ignore()
    或 accept()，程序还会继续执行之后的语句。*/
    if(e->source()==this) return;
    setText(e->mimeTypeData()->text());
    e->setDropAction(Qt::MoveAction); //把拖放动作设置为移动。
    e->accept(); //使用此步骤会把 QDrag::exec() 函数的返回值设置为上面设置的移动动作
}

};

class B:public QWidget{ Q_OBJECT
public:
    B(QWidget *p=0):QWidget(p){
        C *pb1=new C("AAA",this); pb1->move(22,22);
        C *pb2=new C("BBB",this); pb2->resize(111,66); pb2->move(99,22);
        pb1->setIcon(QIcon("F:/li.png"));
        pb2->setAcceptDrops(true); pb1->setAcceptDrops(0);
    }
};

#endif // M_H

//m.cpp 文件的内容
#include "m.h"

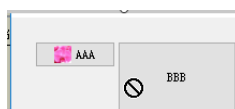
int main(int argc, char *argv[]){ QApplication app(argc,argv);
    B w; w.resize(444,355); w.show(); return app.exec(); }

```

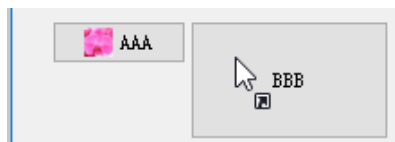
运行结果及说明，当拖放被成功放置到按钮 BBB 上时，程序会输出字符串 AAA。



把光标移至按钮 **BBB** 上半部分的情形



当光标移至按钮 **BBB** 的下半部分时，拖放被禁止



拖动时按下 **ALT** 键，改变了光标右下角的图形



拖动时按下 **SHIFT** 键，也会改变光标右下角的图形

三、使用拖放打开文件

拖放文件的基本步骤：文件需要使用 **QFile** 类来打开，然后才能读取或存入其内容，因此对拖放的文件进行处理，其实就是获取文件的地址，而地址是使用 **URL** 来表示的，因此首先需要判断拖放的数据是否含有 **URL**，然后读取出 **URL** 中保存的文件的地址，再打开文件，然后读取文件的内容，有关流和文件的内容本章暂时不用深入了解，明白以下示例代码的作用即可。下面是具体的示例步骤。

示例：拖放文件

//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>
class C:public QPushButton{Q_OBJECT
public:    C(QString t="",QWidget *p=0):QPushButton(t,p) {}
    void dragEnterEvent(QDragEnterEvent * e){
        //若拖动的数据包含一个 URL 则接受该事件，否则忽略该事件。
        if(e->mimeType()->hasUrls()){    e->accept();    }    else e->ignore();    }

    void dropEvent(QDropEvent * e){
        const QMimeData *pm=e->mimeType();
        QList<QUrl> u=pm->urls();    //读取出 URL 的地址列表。
        QString pth=u.at(0).toLocalFile();    //将地址转换为 QString
        if(!pth.isEmpty()){    //判断地址 pth 是否为空
            QFile file(pth);    //创建文件 file
            if(!file.open(QIODevice::ReadOnly));    //以只读方式打开文件
                QTextStream in(&file);    //创建流用于读取文件的内容。
                setText(in.readAll());    } }    //读出文件的内容，并设置为该部件的文本
    };
class B:public QWidget{    Q_OBJECT
public:
    B(QWidget *p=0):QWidget(p) {
```

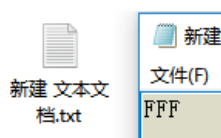
/*注：实际编程时只需把按钮替换为 QTextEdit 之类的部件即可，此处为避免复杂性及明白其原理，使用简单的按钮就可以了。*/

```
C *pbl=new C("AAA",this);    pbl->move(22,22);    pbl->setAcceptDrops(true);    }  
};  
#endif // M_H
```

//m.cpp 文件的内容

```
#include "m.h"  
int main(int argc, char *argv[]) {    QApplication app(argc,argv);  
    B w;    w.resize(444,355);    w.show();    return app.exec(); }
```

运行结果及说明



新建一个文本文件，右侧为文件中的内容。



拖动该文件到按钮 AAA 上面，然后释放鼠标



文本图标被设置为文本文件中的内容。

9.2 与拖放事件有关的类及函数

本小节所讲的内容在上一小节几乎都已介绍了，此处仅列出其原型并作一简要介绍。

一、QDropEvent 类

1、QDropEvent 继承自 QEvent，在完成拖放操作时发送该事件，具体规则见前文

2、QDropEvent 类中的函数

1)、**QDropEvent**(const QPointF &*pos*, Qt::DropActions *actions*, const QMimeData **data*, Qt::MouseButtons *buttons*, Qt::KeyboardModifiers *modifiers*, Type *type* = Drop);

构造函数，因很少需要创建该事件，所以仅列出其原型

2)、const QMimeData ***mimeType**() const; //返回被放置到部件上的 QMimeData 对象。

3)、QPoint **pos**() const; //返回鼠标光标的位置

4)、const QPointF &**posF**() const; //返回鼠标光标的位置

5)、Qt::KeyboardModifiers **keyboardModifiers**() const; //返回按下的键盘修饰键(比如 Ctrl)

6)、Qt::MouseButtons **mouseButtons**() const; //返回按下的鼠标按钮。

7)、Qt::DropAction **dropAction**() const; //返回数据执行的拖放动作

void **setDropAction**(Qt::DropAction *action*)

设置数据执行的拖放动作，使用该函数后应调用 **accept()**而不是 **acceptProposedAction()** 函数。

8)、void **acceptProposedAction**(); //把拖放操作设置为建议的操作并接受该事件。

9)、Qt::DropActions **possibleActions**() const;

返回可能的拖放动作或其组合，即 QDrag::exe()函数中指定的参数。

10)、Qt::DropAction **proposedAction**() const; //返回建议的拖放动作。

11)、QObject ***source**() const

若拖动操作的源部件是该程序中的一个部件，则返回该源部件，否则返回 0，源部件是实例化 QDrag 对象的第一个参数。比如 QDrag *dg=new QDrag(this);则该函数将返回部件 this。

二、QDragMoveEvent 类

1、QDragMoveEvent 继承自 QDropEvent，当拖动操作正在进行时，以及当具有焦点时按下键盘的修饰键(比如 Ctrl)时，发送该事件，具体规则见前文

2、QDragMoveEvent 类中的函数

1)、**QDragMoveEvent**(const QPoint &pos, Qt::DropActions actions, const QMimeData *data,

Qt::MouseButtons buttons, Qt::KeyboardModifiers modifiers, Type type = DragMove)

构造函数，因很少需要创建该事件，所以仅列出其原型

2)、void **accept**(const QRect &rectangle)

与 accept()相同，但若以后的移动仍在矩形 rectangle 内，则是可以接受的，这对于源在计时器事件中滚能非常有用。该函数可提高性能，但有可能被底层系统忽略。

3)、void **ignore**(const QRect &rectangle)

accept(const QRect&)相反，在矩形内移动是不可接受的，且将被忽略。

4)、QRect **answerRect**() const

若 accept()函数的参数设置了矩形，则返回该矩形，否则返回的该矩形的左上角是鼠标光标的位置，其大小为 1*1，即返回的矩形为(pos().x(), pos.y(), 1, 1);

三、QDragEnterEvent 类和 QDragLeaveEvent 类

1、QDragEnterEvent 继承自 QDragMoveEvent，当拖动操作进入部件时，该事件被发送到部件，具体规则见前文

2、QDragEnterEvent 类仅有一个构造函数，原型如下所示

QDragEnterEvent(const QPoint &point, Qt::DropActions actions, const QMimeData *data,

Qt::MouseButtons buttons, Qt::KeyboardModifiers modifiers)

3、QDragLeaveEvent 继承自 QEvent，当拖放操作离开部件时发送该事件，具体规则见前文，不应创建一个 QDragLeaveEvent 类，该类依赖于 Qt 的内部状态。该类仅有一个默认构造函数。

四、QWidget 类中与拖放有关的函数

1、QWidget 类中与拖放有关的函数就是拖放事件的处理函数，这些函数都是受保护的虚拟函数，现列出如下：

virtual QWidget::**dragEnterEvent**(QDragEnterEvent *e); //受保护的，虚拟的

virtual QWidget::**dragMoveEvent**(QDragMoveEvent *e); //受保护的，虚拟的

```
virtual QWidget::dropEvent(QDropEvent *e); //受保护的，虚拟的
virtual QWidget::dragLeaveEvent(QDragLeaveEvent *e); //受保护的，虚拟的
```

9.3 QDrag 类

1、QDrag 继承自 QObject，该类支持基于 QMimeData 对象数据的拖放。该类主要成员函数的使用方式在前文已介绍过，该类的其他成员函数可设置拖放时的外观等其他设置，比如拖放时显示一个图标而不是鼠标光标等。

2、QDrag 类中的函数

1)、QDrag(QObject *dragSource)

构造函数，注意：该类没有默认构造函数，必须为其指定一个参数，这个参数通常会作为 source()函数返回的对象。

2)、Qt::DropAction exec(Qt::DropActions supportedActions = Qt::MoveAction)

Qt::DropAction exec(Qt::DropActions supportedActions, Qt::DropAction defaultDropAction)

- 以上函数表示，启用拖放操作，并在拖放完成时返回最终的拖放动作的值。
- 可能的拖放动作由 supportedActions 指定，参数 defaultDropAction 用于设置建议的拖放动作，若该动作不在 supportedActions 的组合中，则建议的拖放动作为复制。若未使用 defaultDropAction 参数，则默认建议拖放动作按照顺序移动、复制、链接进行选择。
- 注：在 Linux 和 macOS 上，拖放操作不会阻止事件循环，其他事件仍会传递给应用程序。但在 windows 上，Qt 事件循环在操作过程中会被阻塞。
- 该函数的具体使用见本章第 1 节。

3)、QMimeData *mimeData() const; //返回拖动的 QMimeData 对象

void setMimeData(QMimeData *data)

设置要拖动的 QMimeData 对象(即拖动数据)，该函数会获得 data 的所有权。

4)、QPixmap dragCursor(Qt::DropAction action) const

void setDragCursor(const QPixmap &cursor, Qt::DropAction action)

设置相对于拖放动作的拖动光标(即拖动时的光标)为 cursor，该函数可使拖动时使用本地系统之外的光标(即可改变光标的外形)，若 cursor 为空，则恢复本地系统的光标，action 只能是 CopyAction、MoveAction、LinkAction，其他值将会被忽略。示例(见右图)

```
QDrag *dg = new QDrag(this);
.....
dg->setDragCursor(QPixmap("F:/1i.png"), Qt::CopyAction);
dg->exec(Qt::MoveAction|Qt::CopyAction, Qt::CopyAction);
```



注意：这一张图片是拖动时的光标外形。

5)、QPixmap pixmap() const

void **setPixmap**(const QPixmap &*pixmap*)

把拖动操作中的数据显示的像素图设置为 *pixmap*。见下图



这是光标

把拖动时的数据显示为该图标



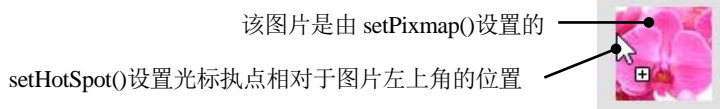
这是常见的操作

拖动时数据显示为该图标

6)、QPoint **hotSpot**() const

void **setHotSpot**(const QPoint &*hotspot*)

将热点(即光标的实际有效点)设置为相对于像素图左上角的位置 *hotspot*, 注: 在 X11 上, 像素图可能无法跟上鼠标移动。原理见下图



7)、static void **cancel**(); //取消由 Qt 启动的拖放操作。静态的。qt5.7

8)、Qt::DropAction **defaultAction**() const; //返回该拖放操作的默认建议放置动作。

9)、Qt::DropActions **supportedActions**() const; //返回此拖放可能的拖放动作组合。

10)、QObject ***source**() const; //返回拖动对象的源部件(即启用拖放的部件)。

11)、QObject ***target**() const;

返回拖放的目标部件(即拖放对象被放下的部件)。可把该函数放置于 **exec()** 之后来获取放置的目标, 比如

```
QDrag *dg = new QDrag(this);
```

```
.....
```

```
dg->target(); //此时不能成能获取放置的目标部件。
```

```
dg->exec();
```

```
dg->target(); //获取放置的目标部件。
```

12)、void **actionChanged**(Qt::DropAction *action*); //当与拖动相关的动作改变时, 发送该信号。信号

13)、void **targetChanged**(QObject **newTarget*); //当拖放的目标部件发生变化时, 发送该信号。信号

9.4 QMimeData 类与拖放自定义类型数据

一、基本规则

- 1、QMimeData 类继承自 QObject，QMimeData 对象保存与 MIME 类型相关联的数据。
- 2、QMimeData 类用于描述存储在剪贴板中，并能通过拖放机制进行传输的信息。
- 3、QMimeData 对象通常使用 new 创建，并提供给 QDrag 或 QClipboard 对象。QMimeData 对象可同时使用几种不同的格式来存储相同的数据。
- 5、MIME 格式并不总是能直接映射为剪贴板格式，Qt 提供 QWinMime 类用于把 MIME 类型映射为 windows 剪贴板格式。QMacPasteboardMime 类用于把 MIME 类型映射为 Mac 风格的剪贴板格式。
- 6、可使用如下方法把自定义的 MIME 类型数据存储在 QMimeData 对象中
 - 1)、使用 setData 函数把自定义类型以 QByteArray 的形式直接存储在 QMimeData 中，使用此方法一次只能对一个 MIME 类型进行处理。
 - 2)、子类化 QMimeData

二、QMimeData 类中的函数

- 1、QMimeData() //构造函数，注意：不能为该类指定父对象。
- 2、与常见 MIME 类型相关的函数

QMimeData 提供的与常见 MIME 类型相关的函数				
测试函数	读取函数	设置函数	MIME 类型	Qt 类型
hasText()	text()	setText	text/plain	QString
hasHtml()	html()	setHtml()	text/html	QString
hasUrls()	urls()	setUrls()	text/uri-list	QUrl
hasImage	imageData()	setImageData()	image/*	QVariant
hasColor()	colorData()	setColorData()	application/x-color	QVariant

注意：颜色和图像使用 QVariant 进行存储，而不是 QColor 和 QImage，因此在实际使用时需要在 QColor 或 QImage 和 QVariant 之间进行转换，其中 QColor 或 QImage 到 QVariant 的转换是隐式的，但从 QVariant 到 QColor 或 QImage 之间的转换需要使用 qvariant_cast()，比如 QColor c = qvariant_cast<QColor>(mimeData->colorData());

- 1)、void setText(const QString &text); //把该对象文本数据设置为 text
bool hasText() const; //若该对象可以返回文本，则返回 true，否则返回 false
QString text() const; //返回该对象存储的 URL 列表。
- 2)、void setUrls(const QList<QUrl> &urls); //把该对象中的 URL 设置为 urls。
bool hasUrls() const; //若该对象可以返回 URL 列表，则返回 true，否则返回 false
QList<QUrl> urls() const; //返回该对象存储的文本。
- 3)、void setHtml(const QString &html); //把该对象中的 HTML 数据设置为 html
bool hasHtml() const; //若该对象可以返回 HTML，则返回 true，否则返回 false

- QString **html**() const //若存储在该对象中的数据表示 HTML，则返回该字符串，
否则返回一个空字符串
- 4)、void **setImageData**(const QVariant &*image*); //把该对象中的图像数据设置为 image。
bool **hasImage**() const; //若该对象可以返回图像，则返回 true，否则返回 false
QVariant **imageData**() const //若存储在该对象中的数据表示图像，则返回该图像，否则
返回一个空的 QVariant。
- 5)、void **setColorData**(const QVariant &*color*); //把该对象中的颜色数据设置为 color
bool **hasColor**() const; //若该对象可以返回颜色，则返回 true，否则返回 false
QVariant **colorData**() const //若存储在该对象中的数据表示颜色，则返回该颜色，否
则返回一个空的 QVariant。

3、其他函数

- 6)、void **removeFormat**(const QString &*contentType*); //删除与 contentType 类型关联的数据项。
7)、void **clear**(); //删除该对象中的所有 MIME 类型和数据项。

4、与自定义 MIME 类型相关的函数(使用 QByteArray 存储数据)

- 8)、QByteArray **data**(const QString &*contentType*) const
返回以 contentType 描述的格式存储在该对象中的数据。

- 9)、void **setData**(const QString &*contentType*, const QByteArray &*data*)
把与 contentType 描述的格式相关联的数据设置为 data。注意：若要在模型/视图拖放操作中使用自定义数据类型，必须使用 Q_DECLARE_METATYPE()宏注册为 Qt 元类型，并为该类型重新实现流操作符(即<<和>>)，且流操作符必须使用
qRegisterMetaTypeStreamOperators()函数进行注册。

5、虚函数，与自定义 MIME 类型相关的函数(不使用 QByteArray 存储数据)

- 10)、virtual QStringList **formats**() const; //虚拟的
返回该对象支持的格式列表。
- 11)、virtual bool **hasFormat**(const QString &*contentType*) const; //虚拟的
若对象可以返回 contentType 指定的 MIME 类型的数据，则返回 true，否则返回 false。
- 12)、virtual QVariant **retrieveData**(const QString &*contentType*, QVariant::Type *type*) const; //虚拟的
返回包含由 contentType 指定的 MIME 类型的数据，该类型的 Qt 类型为 type。若该对象不支持给定的 MIME 类型或 QVariant 类型，则返回一个空的 QVariant。该函数一般被
读取函数 data()和 text()、html()、urls()、imageData()、colorData()调用。若要使用自定义
类型的数据，而不是 QByteArray 存储数据，则需要重新实现该函数，另外还需要重新
实现 hasFormat()和 formats()，子类化 QMimeData 时对这些虚函数的实现都不是必须
的。

示例：使用 setData() 函数处理自定义 MIME 类型的数据

```
//m.h 文件的内容
#ifndef M_H
#define M_H
#include<QtWidgets>
```

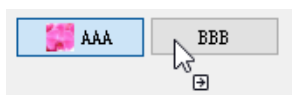
```

class C:public QPushButton{Q_OBJECT
public:    C(QString t="",QWidget *p=0):QPushButton(t,p) {}
    void mouseMoveEvent(QMouseEvent *e){
        QDrag *dg=new QDrag(this);                QMimeData *md=new QMimeData;
        QIcon ic("F:/li.png");        QString s="ZZZ";        QByteArray ba;
        //使用流把数据 ic 和 s 写入 ba 中(流暂时先不用深入了解)。
        QDataStream in(&ba,QIODevice::WriteOnly); //以只写方式创建一个流
        in<<ic<<s;                //把 ic 和 s 的写入 ba 中
        md->setData("XXX",ba); /*设置自定义的 MIME 类型数据,其中 XXX(名称任意)为自定义
                                的 MIME 类型,ba 为存储该类型数据的对象。*/
        dg->setMimeData(md);    dg->exec();    }
    void dragEnterEvent(QDragEnterEvent * e){        e->acceptProposedAction();    }
    void dropEvent(QDropEvent * e){
        const QMimeData *dg=e->mimeData();
        if(dg->hasFormat("XXX")){ //判断是否有自定义 MIME 类型 XXX。
            //读取自定义 MIME 类型 XXX 的数据,并保存在 ba 中
            QByteArray ba = dg->data("XXX");
            //以下语句表示使用流把 ba 中的数据读取出来
            QDataStream out (&ba,QIODevice::ReadOnly);
            QIcon ic;        QString s;                out>>ic>>s;
            //设置该部件的图标为 ic,文本为 s。
            setIcon(ic);                setText(s);
        }
        e->accept();
    }    };
class B:public QWidget{    Q_OBJECT
public:
    B(QWidget *p=0):QWidget(p){
        C *pb1=new C("AAA",this);    pb1->move(22,22);    pb1->setIcon(QIcon("F:/li.png"));
        C *pb2=new C("BBB",this);    pb2->move(99,22);
        pb2->setAcceptDrops(true);    pb1->setAcceptDrops(false);
    };
#endif // M_H

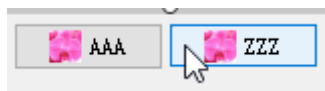
//m.cpp 文件的内容。
#include "m.h"
int main(int argc, char *argv[]){    QApplication app(argc,argv);
    B w;    w.resize(444,355);    w.show();    return app.exec(); }

```

运行结果及说明



由 AAA 向 BBB 拖动时



拖动完成后按钮 BBB 的图标和文本都被成功修改。

三、子类化 QMimeData

1、MIME 类型数据的存储

MIME 类型不属于 Qt 类型或 C++ 类型，与这些类型关联的数据需要被存储，为此需要指定一个存储 MIME 类型的 Qt 类型(或 C++ 类型)，并使用一个对象来存储该类型的数据，比如对于 MIME 类型 XXX，与 XXX 关联的数据为两个字符串"SSS"和"TTT"，那么"SSS"和"TTT"需要被存储在应用程序中，比如可以使用 QByteArray 类型的对象 ba 来存储它们(也可使用 QList 或其他类来存储)，那么与 MIME 类型相关联的类型是 QByteArray 类型，MIME 类型的数据"SSS"和"TTT"被存储在 ba 中。

- 2、由以上原理可知，QMimeData 的本质就是用于把需要用于传输的数据保存在该类的对象中，因此最小的子类化 QMimeData 类只需在该类中定义一些保存数据的成员变量就可以了。并不需要重新实现 hasFormat()、formats()、retrieveData()等虚函数，但这样做的话，没有任何对 MIME 类型的限制和指定，因此 MIME 类型就变得毫无意义了，见下面的示例

//示例：最小子类化 QMimeData(即 QMimeData 本质)

//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>
```

//对于此类 MIME 类型毫无意义。只需声明以下 3 个成员变量来保存需要传输的数据即可。

```
class D:public QMimeData{public:   QIcon c;   QString s;   int i;   };

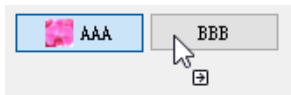
class C:public QPushButton{Q_OBJECT
public:   C(QString t="",QWidget *p=0):QPushButton(t,p) {}
    void mousePressEvent(QMouseEvent *e) {
        QDrag *dg=new QDrag(this);        D *md=new D;
        //直接访问类 D 的成员变量，并把需要传输的数据直接保存在成员变量中。
        md->c=QIcon("F:/li.png");        md->s="ZZZ";
        dg->setMimeData(md);        dg->exec();        }
    void dragEnterEvent(QDragEnterEvent * e) {        e->acceptProposedAction();        }
    void dropEvent(QDropEvent * e) {
        const D *dg=(D*)e->mimeTypeData(); //强制转换为子类 D
        //直接访问其成员变量
        setIcon(dg->c);        setText(dg->s);        e->accept();        } };

class B:public QWidget{   Q_OBJECT
public:
    B(QWidget *p=0):QWidget(p) {
        C *pb1=new C("AAA",this);        pb1->move(22,22);        pb1->setIcon(QIcon("F:/li.png"));
        C *pb2=new C("BBB",this);        pb2->move(99,22);
        pb2->setAcceptDrops(true);        pb1->setAcceptDrops(false);        } };
#endif // M_H
```

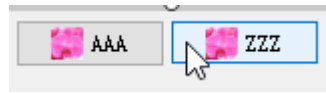
//m.cpp 文件的内容。

```
#include "m.h"
int main(int argc, char *argv[]) {   QApplication app(argc,argv);
    B w;        w.resize(444,355);        w.show();        return app.exec(); }
```

运行结果及说明



由 AAA 向 BBB 拖动时



拖动完成后按钮 BBB 的图标和文本都被成功修改。

四、重新实现 QMimeData 类中的虚函数

1、QMimeData 类中的虚函数 hasFormat()、formats()、retrieveData() 主要用于对 MIME 类型进行判别，但并未提供存储数据的功能，因此仅仅实现这些虚函数是无法存储数据的，要使数据被存储，可以在 QMimeData 中声明成员变量，并把数据存储在其中，若想更完美，则可以像 QMimeData 一样，再定义一些读取函数、设置函数和测试函数。

示例：重新实现 QMimeData 类中的虚函数

//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>

class E{public:QIcon c;QString s;};
class D:public QMimeData{public:
    E me;    //使用自定义类型 E 存储自定义的 MIME 类型的数据
    QStringList st; //用于存储自定义的 MIME 类型
    //存储 MIME 类型的数据
    void setXXX(QString m,E e){
        if(!hasFormat(m)) st.append(m); //若 MIME 类型 m 不存在于列表 st 之中，则追加到 st 中。
        me.c=e.c;me.s=e.s; } //存储数据
    //若 MIME 类型存在，则返回 me
    E XXX(QString m)const{ if(hasFormat(m)) return me;else return E();}

    QStringList formats() const{ return st; } //返回自定义的 MIME 类型列表。

    bool hasFormat(const QString &m) const{
        if(st.contains(m)) return true; //判断自定义类型 m 是否包含在 st 之中
        //调用父类的函数以处理 text/html 等常见 MIME 类型的数据。
        else return QMimeData::hasFormat(m); }

    //该函数通常由读取函数调用，本示例不需要调用此函数。
    QVariant retrieveData(const QString &m, QVariant::Type type) const{
        return QMimeData::retrieveData(m,type);} };

class C:public QPushButton{Q_OBJECT
public:    C(QString t="",QWidget *p=0):QPushButton(t,p) {}
    void mousePressEvent(QMouseEvent *e){
        QDrag *dg=new QDrag(this);    D *md=new D;
        E me;    me.c=QIcon("F:/li.png");    me.s="111";
        md->setXXX("XXX",me); //创建一个自定义的 MIME 类型 XXX，其包含的数据位于 me 之中。

        //创建两个自定义的 MIME 类型 CCC 和 DDD。由此可见，本示例可以一次创建多个 MIME 类型。
```

```

md->st<<"CCC"<<"DDD";
E me1;          me1.c=QIcon("F:/li.png");          me1.s="222";
md->setXXX("CCC",me1); //自定义 MIME 类型包含的数据位于 me1 之中。
dg->setMimeData(md);          dg->exec();          }

void dragEnterEvent(QDragEnterEvent * e){          e->acceptProposedAction(); }

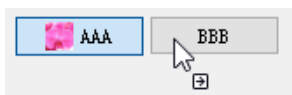
void dropEvent(QDropEvent * e){
    const D *dg=(D*)e->mimeTypeData(); //强制转换为子类 D
    //若 CCC 是合格的自定义 MIME 类型，则将其图标和文本设置为该类型所关联的数据。
    //本例未使用自定义 MIME 类型 XXX 的数据。
    if(dg->hasFormat("CCC")){ setIcon(dg->XXX("CCC").c); setText(dg->XXX("CCC").s);}
    e->accept();          }
};

class B:public QWidget{          Q_OBJECT
public:
    B(QWidget *p=0):QWidget(p){
        C *pb1=new C("AAA",this);          pb1->move(22,22);          pb1->setIcon(QIcon("F:/li.png"));
        C *pb2=new C("BBB",this);          pb2->move(99,22);
        pb2->setAcceptDrops(true);          pb1->setAcceptDrops(false);          }
};
#endif // M_H

//m.cpp 文件的内容。
#include "m.h"
int main(int argc, char *argv[]){          QApplication app(argc,argv);
    B w;          w.resize(444,355);          w.show();          return app.exec(); }

```

运行结果及说明



由 AAA 向 BBB 拖动时



拖动完成后按钮 BBB 的图标和文本都被成功修改为自定义 MIME 类型 CCC 所携带的数据。

9.5 QClipboard 类(剪贴板)

一、剪贴板基础

- 1、QClipboard 类继承自 QObject，该类主要提供对系统剪贴板的访问。
- 2、剪贴板提供了一种在应用程序之间复制和粘贴数据的机制。剪贴板的原理与拖放类似，其差别在于剪贴板把需要传输的数据放置于剪贴板上，需要传输的数据都是通过 QMimeData 对象存储的。
- 3、为便于讲解，把剪贴板分为三种类型如下：
 - 全局鼠标选择：是从全局鼠标选择中存储和检索数据，其好处是可以在以后使用鼠标中键来粘贴其数据，但仅在具有全局鼠标选择的系统上提供支持(比如 X11)，Windows 和 macOS 都不支持这种剪贴板。
 - 查找(或搜索)缓冲区：表示从查找缓冲区中存储和检索数据，该方式用于 macOS 上保存搜索的字符串。
 - 全局剪贴板：Windows 只支持全局剪贴板，这种剪贴板是完全的全局资源。
 - 以上三种剪贴板使用 QClipboard::Mode 枚举进行描述，见下表

QClipboard::Mode 枚举(无标志)		
用于描述剪贴板的类型		
成员	值	说明
QClipboard::Clipboard	0	全局剪贴板
QClipboard::Selection	1	全局鼠标选择
QClipboard::FindBuffer	2	查找缓冲区

4、使用剪贴板

剪贴板对象(即 QClipboard 对象)需要从静态函数 QApplication::clipboard()获取，不能使用 QClipboard 类直接创建 QClipboard 对象，因为 QClipboard 类没有公有的构造函数可用。获取剪贴板后，只需调用剪贴板的设置函数(比如 setText())把数据存储在剪贴板上，然后在需要的地方使用读取函数(比如 text())把剪贴板上的数据读取出来即可，剪贴板常用于鼠标右键的复制、剪切和粘贴。剪贴板使用比较简单，下面为使用剪贴板的步骤，实际需根据情况使用

```
QClipboard *pc = QApplication::clipboard(); //获取剪贴板
```

```
pc->setText("FFF"); //把文本 FFF 存储在剪贴板中
```

```
.....
```

```
QClipboard *pc = QApplication::clipboard(); //在需要使用剪贴板数据的地方再次获取剪贴板。
```

```
QString s=pc->text(); //把剪贴板中存储的文本赋给字符串 s，然后便可把 s 用于需要使用的地方
```

二、QClipboard 类中的函数

- 1、注意：该类没有公有的默认构造函数，对于函数中使用的 `QClipboard::Mode` 枚举见上面的表。下表为该类中所有函数的简表

QClipboard 类函数一览		
<code>setText()</code>	<code>text()</code>	文本
<code>setImage()</code>	<code>image()</code>	图像
<code>setPixmap()</code>	<code>pixmap()</code>	像素图
<code>setMimeData</code>	<code>mimeData()</code>	<code>QMimeData</code> 对象
<code>clear()</code>		清除内容
<code>supportsSelection()</code>	<code>supportsFindBuffer()</code>	判断是否支持所指剪贴板
以下函数用于测试剪贴板中是否包含指定类型的数据		
<code>ownsClipboard()</code>	<code>ownsFindBuffer()</code>	<code>ownsSelection()</code>

- 2、void `setText`(const `QString &text`, `Mode mode` = `Clipboard`); //把文本 `text` 复制到剪贴板中
- 3、`QString text`(`Mode mode` = `Clipboard`) const
返回剪贴板中的文本，若剪贴板没有文本，则返回空字符串。
- 4、`QString text`(`QString &subtype`, `Mode mode` = `Clipboard`) const
返回剪贴板中子类型为 `subtype` 的文本，若 `subtype` 为 `null`，则任何子类型都是可接受的。若剪贴板没有文本，则返回空字符串。`subtype` 的常见值是 `"plain"` 和 `"html"`。注意，重复调用此函数可能会很慢，此时应使用 `dataChnaged()` 信号。
- 5、void `setImage`(const `QImage &image`, `Mode mode` = `Clipboard`)
把图像 `image` 复制到剪贴板中
- 6、`QImage image`(`Mode mode` = `Clipboard`) const
返回剪贴板中的图像，若剪贴板没有图像，或图像不被支持，则返回空图像。
- 7、void `setPixmap`(const `QPixmap &pixmap`, `Mode mode` = `Clipboard`)
把像素图 `pixmap` 复制到剪贴板中，该函数比使用 `setImage()` 更慢，因为该函数需要先把 `QPixmap` 转换为 `QImage`。
- 8、`QPixmap pixmap`(`Mode mode` = `Clipboard`) const
返回剪贴板中的像素图，若剪贴板不含像素图，则返回 `null`，比如像素图是 24 位，显示是 8 位，则结果被转换为 8 位，若像素图具有 `alpha` 通道，则结果仅有掩码。
- 9、void `setMimeData`(`QMimeData *src`, `Mode mode` = `Clipboard`)
将剪贴板的数据设置为 `src`，`src` 的所有权被转移到剪贴板，若要删除数据，可使用新数据再次调用该函数，若使用 `clear()` 函数。
- 10、const `QMimeData *mimeData`(`Mode mode` = `Clipboard`) const
返回剪贴板中的 `QMimeData` 对象。注意：当剪贴板内容发生变化时，返回的指针可能会失效。
- 11、void `clear`(`Mode mode` = `Clipboard`)
清除剪贴板的内容。`mode` 用于指定清除的范围，若 `mode` 为 `QClipboard::Clipboard` 则清除全局剪贴板的内容。若 `mode` 为 `QClipboard::Selection`，则清除全局鼠标选择的内容，若 `mode` 为 `QClipboard::FindBuffer`，则清除搜索字符串缓冲区。
- 12、bool `ownsClipboard`() const

若该剪贴板对象拥有剪贴板数据则返回 `true`，否则返回 `false`

13、bool `ownsFindBuffer()` const

若该剪贴板对象拥有查找缓冲区数据则返回 `true`，否则返回 `false`

14、bool `ownsSelection()` const

若该剪贴板对象拥有鼠标选择数据则返回 `true`，否则返回 `false`

15、bool `supportsFindBuffer()` const

若剪贴板支持搜索缓冲区，则返回 `true`，否则返回 `false`

16、bool `supportsSelection()` const

若剪贴板支持鼠标选择，则返回 `true`，否则返回 `false`

二、QClipboard 类中的信号

1、void `changed(QClipboard::Mode mode)`

当剪贴板模式 `mode` 的数据改变时，发送此信号。

2、void `dataChanged()`

当剪贴板的数据发生变化时，发送此信号。在 macOS 或 Qt4.3 及以上版本，只有当应用程序被激活时，才会检测到其他应用程序对剪贴板所做的更改。

3、void `findBufferChanged()`

当查找缓冲区改变时，发送此信号，仅适用于 macOS。在 macOS 或 Qt4.3 及以上版本，只有当应用程序被激活时，才会检测到其他应用程序对剪贴板所做的更改。

4、void `selectionChanged()`

当选择改变时，发送此信号。Windows 和 macOS 不支持该函数。

作者：黄邦勇帅(原名：黄勇)

2018-6-11

本文作者：黄邦勇帅(原名：黄勇)

本文要求读者已经非常熟悉 C++ 语法。若读者不熟悉 C++ 语法，推荐参阅《C++ 语法详解》(作者：黄勇)一书，电子工业出版社出版。

本文主要讲解了 Qt 的滚动条，本文详细讲解了怎样自定义滚动区域及其原理，并列举了详细的示例进行说明，同时本文也是非常方便、快捷的编写 Qt 程序的查阅资料，可方便的查阅到相关内容的原理，以及怎样使用该内容。本文内容由浅入深，易学易懂。

本文使用的是 windows 10 操作系统，Qt 版本为 Qt5.10.1，Qt Creator 的版本为 Qt Creator 4.5.1 本文内容完全属于个人见解与参考文献的作者无关，限于水平有限，其中难免有误解之处，望指出更正。

声明：禁止抄袭、复印、转载本文，本文作者拥有完全版权。

主要参考文献：

- 1、C++语法详解 黄勇 编著 电子工业出版社 2017 年 7 月
- 2、Qt Creator 快速入门(第 3 版) 霍亚飞 编著 北京航空航天大学出版社 2017 年 1 月
- 3、C++ GUI Qt4 编程(第 2 版) [加拿大] Jasmin Blanchette [英] Mark Summerfield 著 闫锋欣 曾泉人 张志强 译 电子工业出版社 2008 年 8 月
- 4、Qt 5 开发及实例 陆文周 主编 电子工业出版社 2014 年 1 月

第 10 章 Qt 滚动目录

[10.1 滚动条、滑块\(QAbstractSlider 类、QScrollBar 类、QSlider 类\)](#)

[10.1.1 基本原理](#)

[10.1.2 最大、最小值和步长](#)

[10.1.3 跟踪 Tracking 与当前值 Value、当前位置 Position](#)

[10.1.4 QAbstractSlider 类中的属性和函数](#)

[10.1.5 QAbstractSlider 类中的信号](#)

[10.1.6 QScrollBar 类](#)

[10.1.7 QSlider 类](#)

[10.1.8 QDial 类](#)

[10.2 QScrollArea 类、\(滚动区域\)](#)

[10.3 QAbstractScrollArea 类\(抽象滚动区域\)](#)

[10.3.1 QAbstractScrollArea 类中的属性](#)

[10.3.2 QAbstractScrollArea 类中的函数](#)

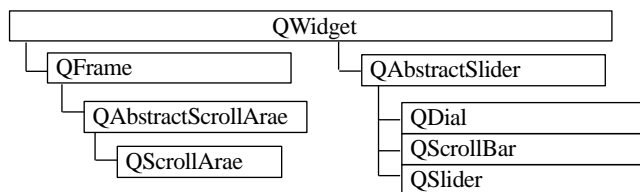
[10.3.3 自定义滚动区域](#)

第 10 部分 Qt 滚动

注意：本程序都假设读者在 pro 文件中已添加了正确的 `QT+=widgets` 语句，文中不再重复累述添加此语句。

本文注重讲解原理，因此使用的是手写的 Qt 程序。

本章讲解的类及继承关系如下图所示



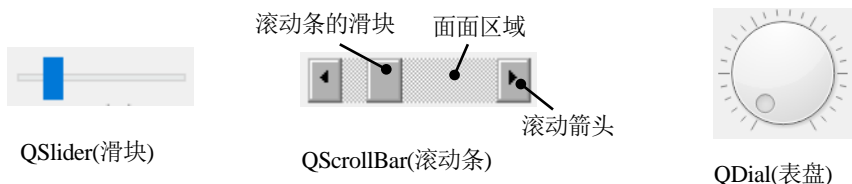
本章所讲的类

10.1 滚动条、滑块

QAbstractSlider 类、QScrollBar 类、QSlider 类

一、基本原理

- 1、QAbstractSlider 继承自 QWidget，该类主要用于提供一个范围内的整数值，
- 2、QAbstractSlider 类是 QScrollBar 类(滚动条)、QSlider 类(滑块)、QDial 类(表盘)的父类，因此该类的属性和函数对这 3 个类都是可行的。
- 3、对于滚动条和滑块主要就是对其相关的一些值的设置，对于其外观样式比较单一，所以滚动条和滑块的主要功能集中在 QAbstractSlider 类中，这个类提供了滚动条和滑块的共同作用，主要是对其值作了描述，对于他们的外形，分别由 QScrollBar 类和 QSlider 决定。所以单独使用 QAbstractSlider 类什么也干不了，通常需要使用他的子类，若使用 `show()` 显示该类的对象，他就是一个空的什么也没有的窗口。
- 4、注意：滚动条默认是不接受键盘焦点的，要使滚动条接受键盘焦点需使用 `QWidget::setFocusPolicy()` 函数设置焦点策略。
- 5、滚动条、滑块、表盘的外观样式见下表



二、最大/最小值和步长

1、最大/最小值，步长，滚动范围，滚动条的滑块大小，文档长度

- 1)、单个步长：是指按一下滚动条(垂直方向)的向上/下箭头或按下键盘的上/下键时滑块移动的距离。水平方向类似
 - 2)、页面步长：通常是指在滚动条上按下 `page up` 和 `page down` 键时移动的距离。默认值通常为 10。
 - 3)、最大和最小值是个逻辑意义上的值，它们不会改变滑块或滚动条的长度，详见后文。
 - 4)、滚动范围：是指滚动条滑块可移动的距离。滚动范围始终是最大值减去最小值。
- 具体见下面的图示。



2、滑块像素大小(即实际大小)

- 1)、像素大小就是指滑块本身的大小(即使用 `resize()` 函数设置的大小)，这个大小是以像素为单位的，以上讲解的最大/最小值、滚动范围、页面步长、文档长度都是逻辑长度，是没有单位的。
- 2)、滑块像素大小受到页面步长的影响，
- 3)、本小节会把按下 `page up` 或 `page down` 时滚动到另一端的次数简称为滚动次数
- 4)、在不影响滑块像素大小的最小大小和最大大小时，可按如下公式计算(可参阅上一点的图示)

● 滚动次数 = (最大值 - 最小值) 除以 (页面步长)

滑块像素大小 = (文档长度的像素大小) 除以 (滚动次数 + 1)。

● 滚动范围 = 最大值 - 最小值、

● 文档逻辑长度 = 最大值 - 最小值 + 页面步长 = 滚动范围 + 页面步长

● 滑块逻辑大小 = 页面步长

● 滚动一次的像素距离 = (文档像素大小 - 滑块像素大小) 除以 滚动次数

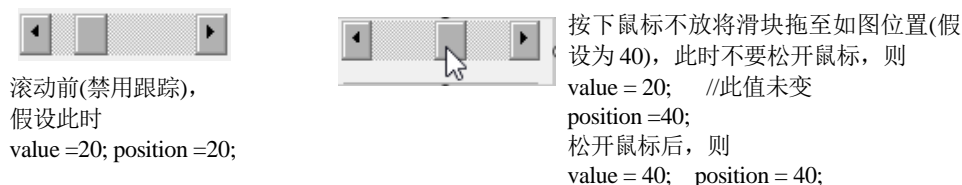
比如，假设文档像素大小为 40，滚动次数为 1，则滑块像素大小为 20 像素，若滚动次数为 2，则滑块像素大小为 $40/3=13.333$ 像素。由此可见滚动次数越多，滑块像素大小就越小，滚动次数越少，滑块像素大小就越大。

- 5)、当按以上公式计算出来的滑块像素大小小于滑块的最小像素大小时，滑块的像素大小始终为最小像素大小保持不变。此时滚动次数、最小/最大值、滚动范围都不会改变，但是文档逻辑长度和滑块逻辑大小不能再按以上公式进行计算。但滚动一次的像素距离比较好计算，其公式为

滚动一次的像素距离 = (文档像素大小 - 滑块像素最小大小) 除以 滚动次数

三、跟踪(Tracking)与当前值(Value)、当前位置(Position)

- 1、跟踪: 若启用跟踪, 则在拖动滑块或滚动条时会发送 `valueChanged()` 信号, 若禁用了跟踪, 则只在用户释放滑块或滚动条时, 才会发送 `valueChanged()` 信号
- 2、当启用了跟踪时, 当前值与当前位置是相同的,
- 3、若未启用跟踪, 则当前值与当前位置是不同的, 原理见下图



四、QAbstractSlider 类中的属性和函数

- 1、**orientation** : Qt::Orientation

访问函数: Qt::Orientation orientation() const; void setOrientation(Qt::Orientation); //槽
设置滚动条或滑块的方向, 只能是 Qt::Vertical(默认)或 Qt::Horizontal

- 2、**sliderDown** : bool **访问函数:** bool isSliderDown() const; void setSliderDown(bool);
描述滑块是否被按下, 设置此属性后在外观上可能会没有变化, 但会发送 `sliderPressed()` 信号

- 3、**maximum** : int **访问函数:** int maximum() const; void setMaximum (int);

- 4、**minimum** : int **访问函数:** int minimum() const; void setMinimum (int);
设置最大和最小值。

- 6、**pageStep** : int **访问函数:** int pageStep () const; void setPageStep(int);
页面步长,

- 7、**singleStep** : int **访问函数:** int singleStep() const; void setSingleStep (int);
单个步长

- 8、**sliderPosition** : int **访问函数:** int sliderPosition() const; void setSliderPosition (int);

信号: void sliderMoved(int value);

滑块的当前位置, 若启用了跟踪, 则此值与 `value` 属性的值相同。

- 9、**tracking** : bool **访问函数:** bool hasTracking () const; void setTracking (bool);

是否启用跟踪, 若启用跟踪, 则在拖动滑块或滚动条时会发送 `valueChanged()` 信号, 若禁用了跟踪, 则只在用户释放滑块或滚动条时, 才会发送 `valueChanged()` 信号。

- 10、**value** : int **访问函数:** int value()const; void setValue(int); //槽

信号: void valueChanged(int value);

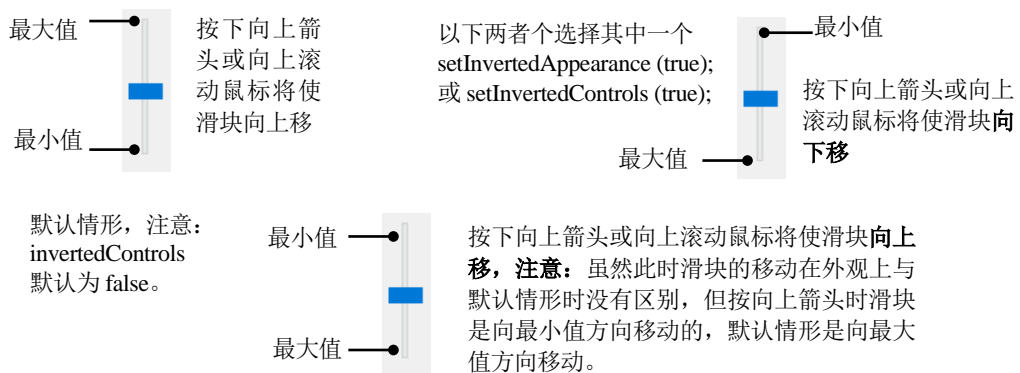
滑块的当前值, 该值被强制在最大值和最小值的范围内。

- 11、**invertedAppearance** : bool

访问函数: bool invertedAppearance() const; void setInvertedAppearance (bool);

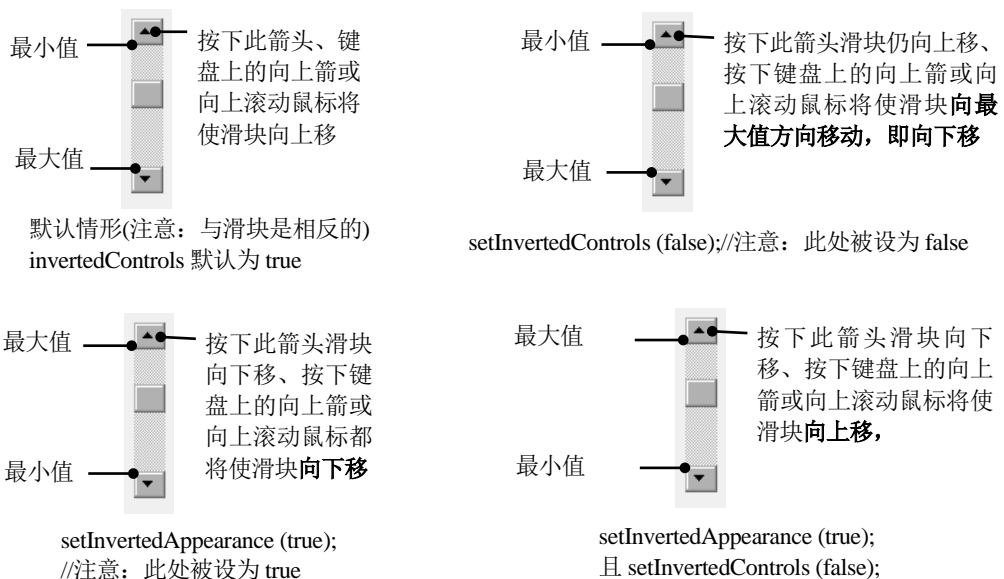
最大值和最小值是否出现在相反的位置，若为 false(默认)，则不反转，若为 true 则反转。该属性对滑块和表盘(dial)更有意义，对于滚动条则取决于样式，大多数样式会忽略滚动条的此属性。具体原理见下图

- 12、**invertedControls** : bool **访问函数**: bool invertedControls() const; void setInvertedControls (bool);
- 是否反转鼠标滚轮和键盘事件，若为 false，则鼠标滚轮向上滚或使用向上键将使值向最小值调整。注意：该属性是使用滚轮或键盘上的方向键移动滑块的方向为最大/最小值方向，且影响的是键盘和鼠标滚轮事件，也就是说该属性不会影响滚动条上的箭头。具体原理见下图



setInvertedAppearance (true); 且 setInvertedControls (true);

滑块的反转



滚动条的反转

- 13、QAbstractSlider(QWidget *parent = Q_NULLPTR); //构造函数
- 14、void setRange(int min, int max); //槽，
设置滑块的最大/最小值与 maximum 和 minimum 属性相同，只不过使用该函数更方便
- 15、void triggerAction(SliderAction action)
触发滑块，SliderAction 枚举见下表，此函数可以用来以编程的方式调整滑块的位置，比如 triggerAction(QAbstractSlider::SliderPageStepAdd)表示把滑块向最大值方长移动一个页面步长。

QAbstractSlider::SliderAction 枚举(无标志)		
作用：描述触发滑块动作的方式		
成员	值	说明
QAbstractSlider::SliderNoAction	0	无动作
QAbstractSlider::SliderSingleStepAdd	1	向最大值方向移动一个单个步长的距离
QAbstractSlider::SliderSingleStepSub	2	向最小值方向移动一个单个步长的距离
QAbstractSlider::SliderPageStepAdd	3	向最大值方向移动一个页面步长的距离
QAbstractSlider::SliderPageStepSub	4	向最小值方向移动一个页面步长的距离
QAbstractSlider::SliderToMinimum	5	把滑块移至最小值(home 键)
QAbstractSlider::SliderToMaximum	6	把滑块移至最大值(end 键)
QAbstractSlider::SliderMove	7	移动滑块

五、QAbstractSlider 类中的信号

- 2、void rangeChanged(int min, int max)
当最大/最小值改变时，发送此信号。
- 3、void sliderMoved(int value)
当 sliderDown 属性为 true，且滑块移动时，发送此信号，即使关闭跟踪(tracking 属性)，也会发送此信号。通常表示用户使用鼠标拖动滑块时，注意：使用键盘方向键或 page up、page down 或按下滚动条上的向上/下箭头都不会触发该信号，使用鼠标时需要按住滑块再拖动滑块，才会发送此信号。
- 4、void sliderPressed()
- 5、void sliderReleased()
当用户用鼠标按下或释放滑块时发送以上信号，可使用 setSliderDown()函数以编程的方式发送以上信号。注意，以上信号是鼠标信号，也就是说对键盘可能会无效。
- 6、void valueChanged(int value)
当滑块的值改变时，发送此信号，tracking 属性对此信号有影响。
- 7、void actionTriggered(int action)
触发滑块时发送，action 表示触发滑块时的动作，见 triggerAction()函数。比如，若滑块以是单个步长增长，则 action 为 1(即 QAbstractSlider::SliderSingleStepAdd)，单击 end 可触发 QAbstractSlider::SliderToMaximum，此时 action 为 6，同理单击 home 可触发 QAbstractSlider::SliderToMinimum，使用其他方式不会触发最大/最小值。

六、QScrollBar 类

1、QScrollBar 类就只有两个构造函数，其默认为取值为垂直，最小值为 0，最大值为 100，单个步长为 1，页面步长为 10，初始位置为 0。原型如下：

```
QScrollBar(QWidget *parent = Q_NULLPTR);
QScrollBar(Qt::Orientation orientation, QWidget *parent = Q_NULLPTR);
```

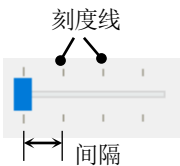
七、QSlider 类

1、QSlider 类，除了从父类继承来的特性，就仅有刻度线的绘制了，详见下文。

2、QSlider (QWidget *parent = Q_NULLPTR); //构造函数，默认为垂直。

```
QSlider (Qt :: Orientation orientation , QWidget * parent = Q_NULLPTR)
```

3、tickInterval: int 访问函数: int tickInterval() const; void setTickInterval(int);
刻度线之间的间隔(见右图)，间隔值是一个逻辑值而不是像素值，若为 0(默认)，将在单个步长和页面步长之间选择。



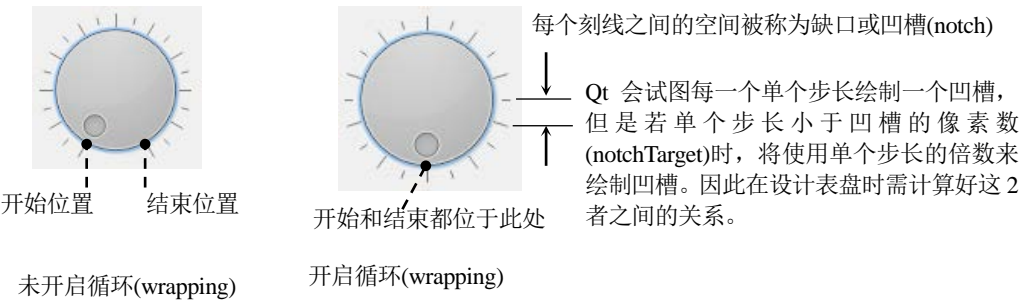
4、tickPosition: TickPosition

访问函数: TickPosition tickPosition() const; void setTickPosition(TickPosition);
描述刻度线的位置，默认为 QSlider::NoTicks(无刻度线)，枚举 TickPosition 见下表

QSlider::TickPosition 枚举(无标志)		
作用：描述刻度线的位置		
成员	值	说明
QSlider::NoTicks	0	无刻度线
QSlider::TicksBothSides	3	在两侧绘制刻度线
QSlider::TicksAbove	1	在(水平)滑块上方绘制刻度线
QSlider::TicksBelow	2	在(水平)滑块下方绘制刻度线
QSlider::TicksLeft	TicksAbove	在(垂直)滑块左侧绘制刻度线
QSlider::TicksRight	TicksBelow	在(垂直)滑块右侧绘制刻度线

八、QDial 类

1、表盘的原理见下图



2、QDial 类中的属性

```
1)、QDial(QWidget *parent = Q_NULLPTR); //构造函数
2)、notchesVisible: bool          访问函数: bool notchesVisible () const;void setNotchesVisible (bool);
```


是否显示凹槽(即刻度线), 默认为 false(不显示)

3)、**notchSize** : const int **访问函数:** int notchSize () const;

返回凹槽的大小, 凹槽的大小的原理见上图示例, 默认为 1。

4)、**notchTarget** : qreal **访问函数:** qreal notchTarget() const; void setNotchTarget (double);

设置凹槽的像素数, 默认为 3.7 像素, 为了便于计算, 可设置为 1。

5)、**wrapping** : bool **访问函数:** bool wrapping () const; void setWrapping (bool);

是否开启循环, 默认为 false(未开启)

10.2 QScrollArea 类(滚动区域)

- 1、QScrollArea 类继承自 QAbstractScrollArea 类。
- 2、QScrollArea 类实现了一个完整的滚动区域，用于显示区域内的子部件的内容，当子部件超过区域的大小时，提供滚动条，以便能查看子部件的完整区域。
- 3、QScrollArea 类的使用方法是使用 setWidget()函数，示例如下：

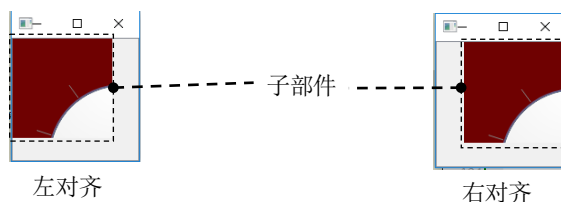
```
.....  
QScrollArea *ps = new QScrollArea;  
ps->setWidget(pw);    //pw 是需要滚动的部件(即添加到滚动区域的子部件)
```

4、QScrollArea 类中的属性和函数

1)、alignment : Qt::Alignment

访问函数：Qt::Alignment alignment() const; void setAlignment(Qt::Alignment)

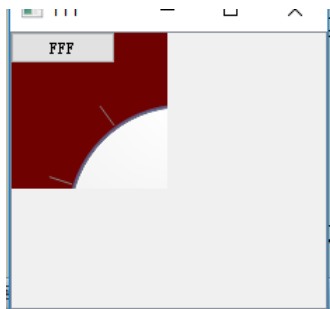
设置添加的子部件相对于滚动区域的对齐方式，效果见下图



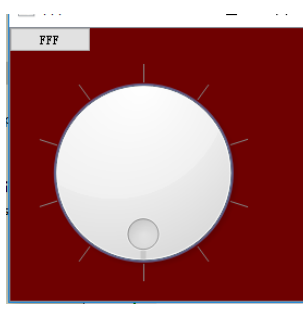
2)、widgetResizable : bool

访问函数：bool widgetResizable() const; void setWidgetResizable(bool resizable)

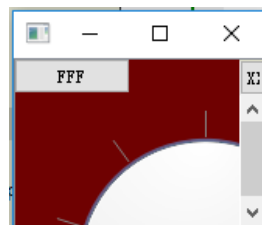
- 是否自动调整子部件的大小，默认为 false(不调整)，原理见下图
- 注：若部件的大小策略为 QSizePolicy::Fixed，则即使该属性为 true，也无法调整子部件的大小。
- 若此属性为 true，则子部件需要使用 QWidget::setMinimumSize()设置其最小大小，否则子部件可能不会被显示，且当滚动区域(即 QScrollArea)的大小小于子部件的最小大小时才会出现滚动条。



widgetResizable 为 false 时，
子部件未完整显示



widgetResizable 为 true 时，子部
件完整显示(调整了子部件)



widgetResizable 为 true 时，
因水平方向滚动区域还未达到
子部件的最小大小，所以
还未出现滚动条。

3)、 **QScrollArea**(QWidget *parent = Q_NULLPTR); //构造函数

4)、 void **addWidget**(QWidget *widget);

把子部件 widget 添加到 QScrollArea 中，QScrollArea 获取该部件的所有权。注意：对子部件的 setLayout()函数的调用，应位于该函数之后，否则会导致 setLayout()设置的布局中的部件不可见。比如

```
QWidget *pw = new QWidget;
QHBoxLayout *ph = new QHBoxLayout;
QScrollArea *ps = QScrollArea;
....
ps->addWidget(pw);
pw->setLayout(ph); //该语句应位于 setWidget()之前，否则 ph 中的内容会不可见。
```

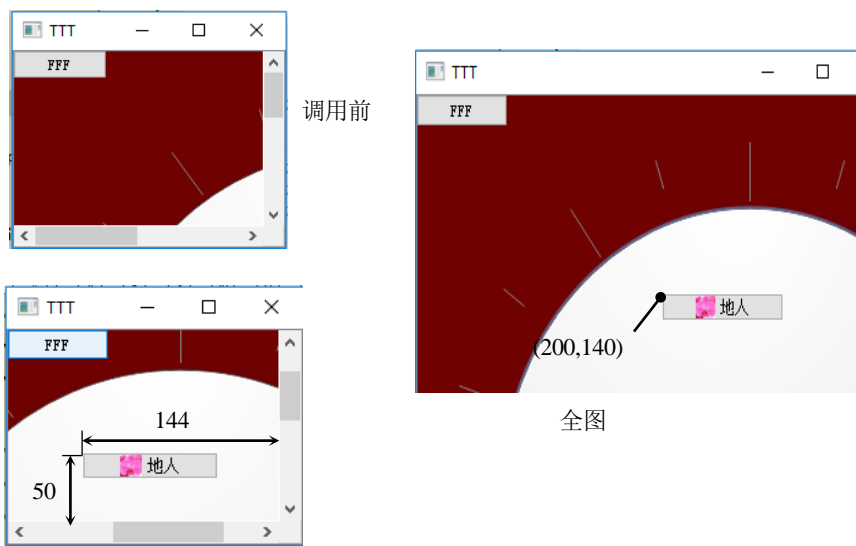
5)、 QWidget ***takeWidget**(); //删除子部件，并把子部件的所有权传递给调用者。

QWidget ***widget**() const; //获取子部件

6)、 void **ensureVisible**(int x, int y, int xmargin = 50, int ymargin = 50)

void ensureWidgetVisible(QWidget *childWidget, int xmargin = 50, int ymargin = 50)

以上函数表示，使点(x,y)或子部件 childWidget 在滚动区域中可见，原理见下图。



调用 ensureVisible(200,140,144,50);

示例：QScrollArea(滚动区域)的使用

```
#include<QtWidgets>
int main(int argc, char *argv[]) {    QApplication app(argc, argv);
    QWidget *pw=new QWidget;        //创建一个容器
    QLabel *pb=new QLabel;
    pb->setPixmap(QPixmap("F:/2. jpg")); //把图片加载到标签中
    QPushButton *pb1=new QPushButton("AAA");
    QPushButton *pb2=new QPushButton("BBB");
    pb1->setSizePolicy(QSizePolicy::Fixed, QSizePolicy::Fixed); //设置大小策略。
    //布局容器 pw 的部件
    QVBoxLayout *pv=new QVBoxLayout;    //主布局
```

```

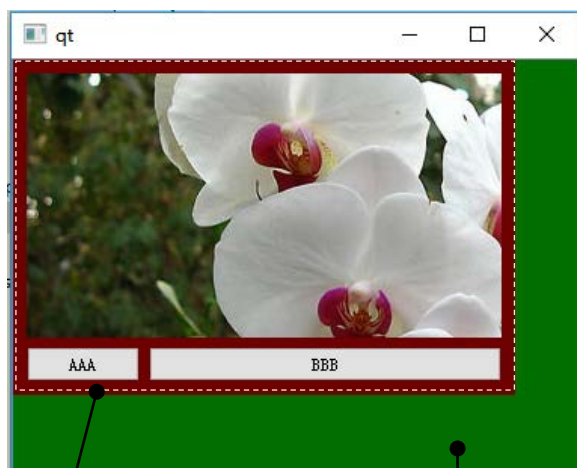
    QHBoxLayout *ph=new QHBoxLayout;
    ph->addWidget(pb1);      ph->addWidget(pb2);
    pv->addWidget(pb);       pv->addLayout(ph);
    pw->setLayout(pv);       //此语句应位于 ps.setWidget(pw);之前。
//设置容器 pw 的背景色为红色
    QPalette p1;            p1.setColor(QPalette::Background,QColor(111,1,1));
    pw->setPalette(p1);

    QScrollArea ps;         /*创建一个滚动区域，记住，QScrollArea 的祖先是 QWidget，因此，还可以像使用 QWidget 一样来使用 QScrollArea。*/

    ps.setWidget(pw);
//设置滚动区域 ps 的背景色为蓝色
    QPalette p2;            p2.setColor(QPalette::Background,QColor(1,111,1));
    ps.setPalette(p2);
//在调用 show() 显示 QScrollArea 之前需为子部件设置大小，否则子部件会不可见。
    pw->resize(333,222);
/*若 widgetResizable 属性为 true，还必须设置最小大小，此时，即使使用 resize() 设置了子部件大小，子部件也会不可见。*/
    pw->setMinimumSize(333,222);
    ps.show();              //显示 QScrollArea，也可把 ps 作为子部件添加到其他部件中。
#####
//以下部分内容用于验证视口部件，不是必须代码
    QWidget *pw1=ps.viewport(); //获取 ps 的视口
//以下语句视情况修改为 0 或 1，用于填充背景，若为 0 背景将透明。
    pw1->setAutoFillBackground(0);
    QPalette p3;            p3.setColor(QPalette::Background,QColor(1,1,111));
    pw1->setPalette(p3); //设置视口的背景色。
    return app.exec();      }

```

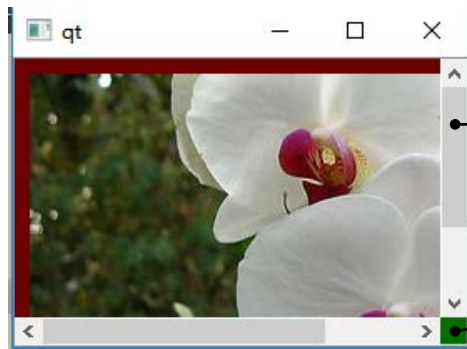
运行结果及说明



虚线框之中红色背景部分为 ps 的子部件 pw，其中的图片和按钮是 pw 的子部件。

最外层的蓝色背景为滚动区域 ps 的颜色。

本示例使用背景色是为了让读者明白，滚动区域 ps 是由多层组成的(为下一个类 QAbstractScrollArea 的讲解打下基础)，每一层都有一个容器(通常为 QWidget)，其实 ps 应有 3 层，最底层为 ps 本身(ps 是 QFrame 的孙子类)，第二层是添加到 ps 中的视口(这通常是一个 QWidget 部件)，第 3 层即为我们添加进去的子部件 pw，pw 通常应位于视口之上。在 QScrollArea 的实现中，视口通常与滚动区域本身始终保持大小一致。把示例代码的 pw1->setAutoFillBackground(0); 中的 0 修改为 1，再次运行程序，此时滚动区域的绿色背景会被视口的蓝色背景替代，因为视口位于滚动区域 ps 之上。



窗口变小后的效果，自动出现了滚动条。

注意此角落的绿色背景色

10.3 QAbstractScrollArea 类(抽象滚动区域)

QAbstractScrollArea 继承自 QFrame，注意：该类不是抽象的(没有纯虚函数)，只是类的功能不够完整，所以才称为抽象的。该类主要用于被继承以实现自定义的滚动区域，首先，先了解其中的属性和函数吧。

一、QAbstractScrollArea 类中的属性

1、horizontalScrollBarPolicy : Qt::ScrollBarPolicy

访问函数：Qt::ScrollBarPolicy horizontalScrollBarPolicy() const;

void setHorizontalScrollBarPolicy(Qt::ScrollBarPolicy)

水平滚动条的策略。默认为 Qt :: ScrollBarAsNeeded。枚举 Qt::ScrollBarPolicy 见下表

Qt::ScrollBarPolicy 枚举(无标志)		
作用：描述滚动条的策略		
成员	值	说明
Qt::ScrollBarAsNeeded	0	按需要显示滚动条
Qt::ScrollBarAlwaysOff	1	从不显示滚动条
Qt::ScrollBarAlwaysOn	2	始终显示滚动条(在 10.7 以上版本的 Mac 上会被忽略)

2、verticalScrollBarPolicy : Qt::ScrollBarPolicy

访问函数：Qt::ScrollBarPolicy verticalScrollBarPolicy() const ;

void setVerticalScrollBarPolicy(Qt::ScrollBarPolicy)

垂直滚动条的策略。默认为 Qt :: ScrollBarAsNeeded。

3、sizeAdjustPolicy : SizeAdjustPolicy //qt5.2

访问函数：SizeAdjustPolicy sizeAdjustPolicy() const ; void setSizeAdjustPolicy(SizeAdjustPolicy policy)

该属性描述当视口大小发生变化时滚动区域的大小如何变化的策略。默认为

QAbstractScrollArea::AdjustIgnored。改变这个属性可能实际上调整了 Scrollarea 的大小。

SizeAdjustPolicy 枚举见下表

QAbstractScrollArea::SizeAdjustPolicy 枚举(无标志) //qt5.2		
作用：描述视口大小发生变化时滚动区域的大小如何变化		
成员	值	说明
QAbstractScrollArea::AdjustIgnored	0	滚动区域不做调整
QAbstractScrollArea::AdjustToContents	2	滚动区域始终调整到视口
QAbstractScrollArea::AdjustToContentsOnFirstShow	1	第一次显示时，滚动区域调整到视口

二、QAbstractScrollArea 类中的函数

1、常用函数

- 1)、QAbstractScrollArea::QAbstractScrollArea(QWidget *parent = Q_NULLPTR); //构造函数
- 2)、void addScrollBarWidget(QWidget *widget, Qt::Alignment alignment)
 - 该函数用于把部件 widget 添加到滚动条上，比如
addScrollBarWidget(w, Qt::AlignTop); //把 w 添加到垂直滚动条的顶部
 - 参数 alignment: 对于水平滚动条必须是 Qt::Alignleft 或 Qt::AlignRight, 对于垂直滚动条必须是 Qt::AlignTop 或 Qt::AlignBottom。
 - 要删除 widget 可直接删除，若另外添加一个部件来代替 widget。
 - widget 会自动调整大小以适合滚动条的几何图形。比如，对于水平滚动条，则 widget 的高度将被设置为与滚动条的高度相匹配。其宽度可使用 QWidget::setMinimumWidth()和 QWidget::setMaximumWidth()来设置。
- 3)、QWidgetList scrollBarWidgets(Qt::Alignment alignment)
返回当前设置的滚动条小部件的列表。 对齐可以是四个位置标志的任意组合。
- 4)、QWidget *cornerWidget() const
void setCornerWidget(QWidget *widget)
以上函数用于获取或设置两个滚动条之间角落中的部件，设置 widget 后，之前设置的角落部件被隐藏，若 widget 为 0，则取消角落部件。滚动区域获得 widget 的所有权。默认情况下，不存在角落部件。
- 5)、QScrollBar *horizontalScrollBar() const; //返回水平滚动条。
QScrollBar *verticalScrollBar() const; //返回垂直滚动条。
void setHorizontalScrollBar(QScrollBar *scrollBar)
void setVerticalScrollBar(QScrollBar *scrollBar)
用滚动条 scrollBar 替换现有的水平/垂直滚动条，并在新滚动条上设置所有前滚动条的滚动条属性，然后删除之前的滚动条。QAbstractScrollArea 默认已经提供了水平和垂直滚动条，使用以上函数可设置自己的滚动条。
- 6)、QSize maximumViewportSize() const //返回视口的大小
- 7)、void setViewport(QWidget *widget)
将视口设置为 widget。 QAbstractScrollArea 将获得给定 widget 的所有权。若 widget 为 0， QAbstractScrollArea 将为视口分配一个新的 QWidget 实例。
- 8)、QWidget *viewport() const
返回视口部件。使用 QScrollArea::widget()函数来检索视口部件的内容。

2、以下为需要重新实现的函数

- 9)、virtual void setupViewport(QWidget *viewport); //虚拟的
在 setViewport()被调用之后，由 QAbstractScrollArea 调用。在使用新的视口之前，在 QAbstractScrollArea 的子类中重新实现此函数以初始化新视口。
- 10)、virtual bool viewportEvent(QEvent *event); //虚拟的，受保护的

- 滚动区域的主事件处理程序(viewport()部件)。 它处理指定的事件，并且可以由子类调用以提供合理的默认行为。
- 若返回 true 表明事件已处理，不需要进一步处理，若返回 false，则该事件应进一步传播。
- QAbstractScrollArea 使所有视口事件处理程序在此函数中都可用，在有意义的情况下，QWidget 的专用处理程序会被重新映射到视口事件，重新映射的专用处理程序是：paintEvent(), resizeEvent(), mousePressEvent(), mouseReleaseEvent(), mouseDoubleClickEvent(), mouseMoveEvent(), wheelEvent(), dragEnterEvent(), dragMoveEvent(), dragLeaveEvent(), dropEvent(), contextMenuEvent() 。
- 子类化 QAbstractScrollArea 时，建议重新实现 QWidget 的专用处理程序，而不是重新实现 viewportEvent()函数，因为 viewportEvent()需要处理的事件太多，鼠标，拖放，绘制等事件都在该函数内需要处理，一不小心就会出现错误。

11)、virtual QSize **viewportSizeHint()** const; //虚拟的，受保护的，qt5.2

返回视口的建议大小。 默认实现返回 viewport()-> sizeHint()。 注意，大小只是视口的大小，没有任何滚动条

12)、virtual void **scrollContentsBy**(int dx, int dy); //虚拟的，受保护的

- 当通过 dx, dy 移动滚动条时，调用此函数，因此应该相应地滚动视口的内容。
- 参数 dx 和 dy 是为方便而存在的，这样类就知道应该滚动多少(做像素移位时有用)，当然也可以直接滚动到滚动条指示的位置。
- 默认实现仅在整個 viewport()上调用 update()，子类可以重新实现此处理程序以进行优化，或者像 QScrollArea 一样移动内容部件。
- 不应在此函数中以编程的方式滚动滚动条。

13)、QMargins **viewportMargins()** const; //返回滚动区域的边距。 受保护的，qt5.5

void **setViewportMargins**(int left, int top, int right, int bottom); //受保护的

void **setViewportMargins**(const QMargins &margins); //受保护的

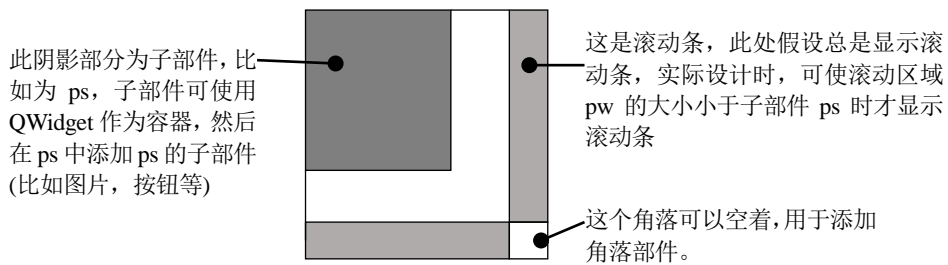
设置滚动区域周围的边距。注意，该函数通常由 QTreeView 和 QTableView 调用，所以边距必须由 QAbstractScrollArea 子类实现。另外，如果要在模型/视图中使用子类，则不应调用此函数。默认情况下，所有边距都为零。

三、自定义滚动区域

1、基本思想：自定义滚动区域，分为两个步骤，即设计外观和计算滚动。下面先讲解外观的设计，只要外观设计好了，计算方法都是相同的。

2、设计外观思想：

- 1)、所谓滚动区域，其实就是一个 QWidget 部件，比如为 pw，然后在 pw 中添加一个子部件，比如为 ps，然后当 pw 的大小小于 ps 时，显示滚动条，原理见下图。
- 2)、由以上可见，可使用任何部件来实现滚动区域，只不过子类化 QAbstractScrollArea 类，可以使用由 Qt 实现的比较方便的函数，比如可以使用 setHorizontalScrollBar()方便的设置水平滚动条，若不然，我们需要把水平滚动条添加到滚动区域的底部，并把其添加到一个布局，比如 QHBoxLayout 布局中，并设计滚动条的大小策略为垂直方向不可拉伸，水平方向拉伸。

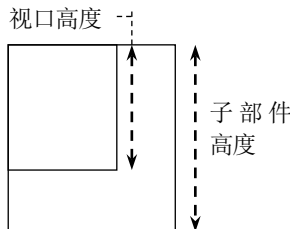


滚动区域为容纳上面所有部件的部件(比如为 pw), QAbstractScrollArea 把容纳上面所有部件的部件, 使用另一个称为视口部件的部件(比如为 pv)来封装, 然后把视口部件(简称视口)添加到滚动区域, 并设计 pv 的大小总是与滚动区域一样大, 这样的话, 我们改变滚动区域的大小时就改变了视口的大小, 而且实际绘图时, 也是在视口上绘制的。因此, 滚动区域被分为 3 层, 即, 滚动区域自身(pw), 在滚动区域之上的视口(pv), 视口之上的子部件(ps)。

3、计算: 计算主要需要计算两个方面, 首先, 当移动滚动条时, 怎样绘制子部件(可简单的移动子部件即可), 其次, 当不需要滚动条时, 恢复子部件为最初的位置。下面分别介绍一种简单的计算方法

1)、移动滚动条的计算(以垂直滚动条为例)

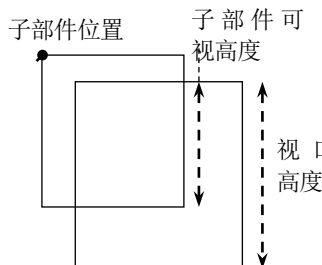
设计滚动条的范围(即最大/最小值), 当需要垂直滚动时, 通常子部件的大小大于视口的大小, 计算原理如下图所示



垂直滚动条假设为 vs, 子部件假设为 ps
vs 的最小值设置为 0
vs 的最大值为: 子部件像素高度 - 视口像素高度, 即
vs.setMaximum(ps->height() - vs->height());
设置拖动滚动条时子部件的位置
ps->move(0, -vs->value()); //注意 y 坐标为负值
以上语句表示, 当拖动垂直滚动条时, 把子部件向上移动 vs->value() 个像素。

2)、恢复子部件最初位置的计算

恢复子部件时, 其视口的大小大于子部件的大小, 计算原理如下图所示



垂直滚动条假设为 vs, 子部件假设为 ps, 视口假设为 pv
获取子部件的位置(即左上角的坐标)
int i = ps->geometry().y() //该值为负值
int j = ps->geometry().x() //该值也为负值
判断是否需要恢复子部件的位置
若视口高度大于子部件可视高度即表示需要恢复子部件的位置。代码如下:
if(ps->height() + i < pv->height();) {
 ps->move(j, -vs->value()); **}**
if 语句中的意思是: 移动子部件 ps 的位置, 其 x 坐标保持不变, y 坐标随垂直滚动条的值的改变而改变, 当垂直滚动条消失时(此时值为 0), 子部件的垂直位置恢复为初始位置

示例：自定义滚动区域(子类化 QAbstractScrollArea)

//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>
class B:public QAbstractScrollArea{    Q_OBJECT
public:    QScrollBar *psh,*psv;    QWidget *pw1,*psub;
    B(QWidget *p=0):psub(0),QAbstractScrollArea(p) {
        //初始化各部件
        pw1=new QWidget;    psh=new QScrollBar;    psv=new QScrollBar;
        //设置滚动区域的外观
        //设置滚动条和滚动策略。
        setVerticalScrollBar(psv);    setHorizontalScrollBar(psh);
        setHorizontalScrollBarPolicy(Qt::ScrollBarAlwaysOn);    //水平滚动条总是可见
        setViewport(pw1);    //把 pw1 设为视口部件。

//以下语句不是必须语句，主要是用于区分各部件而为其设置的背景色。
        //把视口的背景色设置为绿色。
        QPalette p1;    p1.setColor(QPalette::Background,QColor(1,111,1));
        pw1->setAutoFillBackground(1);    //填充背景，否则部件可能会是透明的。
        pw1->setPalette(p1);
        //设置滚动区域的颜色为蓝色。
        QPalette p2;    p2.setColor(QPalette::Background,QColor(1,1,111));
        setAutoFillBackground(1);    setPalette(p2);
        //设置水平滚动条的颜色为红色(需使用样式表设置)
        psh->setStyleSheet("background-color: rgb(111,1,1);");

//联接信号和槽，主要用于当拖动滚动条时移动子部件的位置。
        QObject::connect(psh,&QScrollBar::actionTriggered,this,&B::f1);
        QObject::connect(psv,&QScrollBar::actionTriggered,this,&B::f1);
    }    //构造函数结束

void add(QWidget *p){                //新增一个函数，用于添加用户需要拖动的子部件。
    QWidget *pp=viewport();          //获取视口部件
    p->setParent(pp);                 //把子部件的父部件重新设置为视口部件(即 pw1)
    psub=p;    }                    //把子部件保存在 psub 中，这样 psub 便指向了需要拖动的子部件。

void js(){                            //此函数用于计算和设置滚动条的滚动范围
    //设置页面步长。此步不是必须，但为了使滚动条滑块的大小随视口的改变而改变，
    //需要设置此步(页面步长可影响滚动条滑块的大小，原理见 QScrollBar 章节)。
    psv->setPageStep(viewport()->height());    psh->setPageStep(viewport()->width());

    //设置滚动条的最大最小值，使用像素单位的好处是以后可直接使用滚动条的当前值
    //来调整子部件的位置，以下公式的计算原理见正文。
    psv->setRange(0, psub->height() - viewport()->height());
    psh->setRange(0, psub->width() - viewport()->width());    }

bool viewportEvent(QEvent *e) {
```

```

/*处理 QResizeEvent 事件，此步非常重要，因为该事件在很多情况下(比如初次运行时，改变部件大小时)都会被发送。建议使用专门的 QWidget::resizeEvent() 处理函数来处理 QResizeEvent() 事件，而不是在 viewportEvent() 函数中处理。*/
if (psub!=0&&e->type()==QEvent::Resize) { //判断是否是 QResizeEvent 事件
    int hv = horizontalScrollBar()->value();int vv = verticalScrollBar()->value();
    //QRect r=psub->rect(); //不能获取子部件左上角的坐标，需使用 geometry() 函数。

    //以下代码处理恢复子部件位置时的情况。具体计算原理见正文。
    if (psub->height()+psub->geometry().y()<viewport()->height()) {
        psub->move(psub->geometry().x(),-vv); } //移动子部件位置
    if (psub->width()+psub->geometry().x()<viewport()->width()) {
        psub->move(-hv,psub->geometry().y()); }
    js(); //设置滚动条的滚动范围，此步重要，否则不会显示滚动条。
} //if 结束

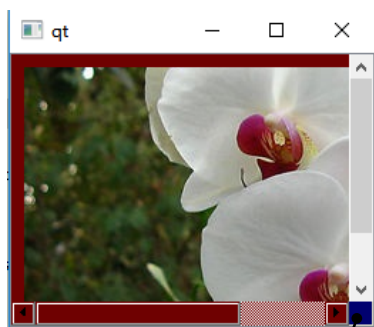
//本示例不需要处理 QPaintEvent 事件，因为 QPaintEvent 事件在某些时候不会被发送，
//比如初次显示界面时就不会发送 QPaintEvent 事件，但会发送 QResizeEvent 事件，
//因此本示例处理 QResizeEvent 事件，而不处理 QPaintEvent 事件
if (psub!=0&&e->type()==QEvent::Paint) { }
//viewport()->update(); // 不能调用该函数，否则会无限循环。
return QAbstractScrollArea::viewportEvent(e);
} //viewportEvent 结束
public slots:
    void f1() {psub->move(-psh->value(),-psv->value());} //当拖动滚动条时，移动子部件的位置。
}; //类 B 结束
#endif // M_H

//m.cpp 文件的内容
#include "m.h"
int main(int argc, char *argv[]) {    QApplication app(argc,argv);
    QWidget *pw=new QWidget; //容器，这将是滚动区域的子部件
    QLabel *pb=new QLabel;    pb->setPixmap(QPixmap("F:/2.jpg"));
    QPushButton *pb1=new QPushButton("AAA");
    QPushButton *pb2=new QPushButton("BBB");
    pb1->setSizePolicy(QSizePolicy::Fixed,QSizePolicy::Fixed);
    //布局容器中的内容
    QVBoxLayout *pv=new QVBoxLayout;    QHBoxLayout *ph=new QHBoxLayout;
    ph->addWidget(pb1); ph->addWidget(pb2); pv->addWidget(pb); pv->addLayout(ph);
    pw->setLayout(pv);
    //设置容器 pw 的背景色为红色
    QPalette pl;    pl.setColor(QPalette::Background,QColor(111,1,1));
    pw->setAutoFillBackground(1);    pw->setPalette(pl);
    pw->resize(333,222);

    B ps; //使用自定义的滚动区域
    ps.add(pw); //添加子部件
    ps.show();    return app.exec(); }

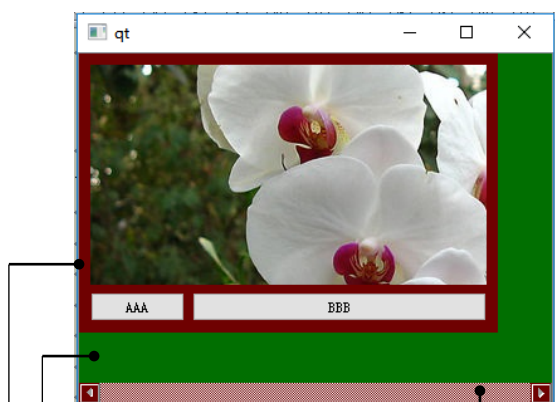
```

运行结果及说明



初次运行

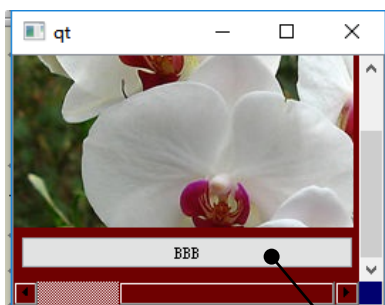
蓝色角落为滚动区域 ps 自身的背景色



绿色为视口的颜色
红色为子部件 pw 的颜色

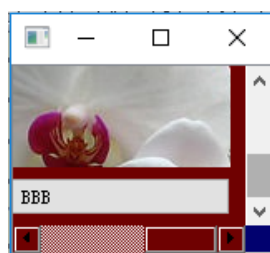
水平滚动条始终显示且为红色

展开后的全景

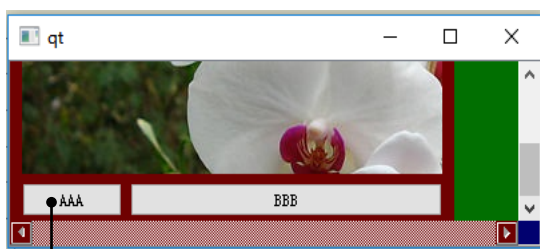


调整滚动条后的效果

由此处可见，子部件被移动了



当主窗口变小时滚动条的滑块随之变小。



由此处可见，当不需要水平滚动条时，子部件的水平位置回到了初始位置

当调整窗口以增大其垂直方向的高度时，子部件会随着高度的增加而逐渐上移，当垂直滚动条消失时，子部件在垂直方向恢复其初始位置。

作者：黄邦勇帅(原名：黄勇)

2018-6-13

本文作者：黄邦勇帅(原名：黄勇)

本文要求读者已经非常熟悉 C++ 语法。若读者不熟悉 C++ 语法，推荐参阅《C++ 语法详解》(作者：黄勇)一书，电子工业出版社出版。

本文主要讲解了 Qt 的文本系统，本文对 Qt 实现文本的原理进行了详细深入全面的讲解，并列举了详细的示例进行说明，同时本文也是非常方便、快捷的编写 Qt 程序的查阅资料，可方便的查阅到相关内容的原理，以及怎样使用该内容。本文内容由浅入深，易学易懂。

本文使用的是 windows 10 操作系统，Qt 版本为 Qt5.10.1，Qt Creator 的版本为 Qt Creator 4.5.1 本文内容完全属于个人见解与参考文献的作者无关，限于水平有限，其中难免有误解之处，望指出更正。

声明：禁止抄袭、复印、转载本文，本文作者拥有完全版权。

主要参考文献：

- 1、C++语法详解 黄勇 编著 电子工业出版社 2017 年 7 月
- 2、Qt Creator 快速入门(第 3 版) 霍亚飞 编著 北京航空航天大学出版社 2017 年 1 月
- 3、C++ GUI Qt4 编程(第 2 版) [加拿大] Jasmin Blanchette [英] Mark Summerfield 著 闫锋欣 曾泉人 张志强 译 电子工业出版社 2008 年 8 月
- 4、Qt 5 开发及实例 陆文周 主编 电子工业出版社 2014 年 1 月

第 11 章 Qt 文本系统目录

[11.1 重要基本概念及原理](#)

[11.2 QPlainTextEdit 类](#)

[11.3 QTextEdit 类](#)

[11.4 表格：QTextTable 和 QTextTableFormat 类](#)

[11.5 框架：QTextFrame 和 QTextFrameFormat 类](#)

[11.6 文本块：QTextBlock、QTextBlockFormat 类](#)

[11.7 列表：QTextList、QTextListFormat 类](#)

[11.8 图像：QTextImageFormat 类和文本片段：QTextFragment 类](#)

[11.9 插入自定义文档对象（文档元素）与总结](#)

[11.10 QTextCharFormat 类及 QTextFormat 和 QTextObject 类简介](#)

[11.11 QTextCursor 类](#)

[11.12 QTextDocument 类](#)

[11.13 其他类：QTextOption、QTextDocumentFragment 等](#)

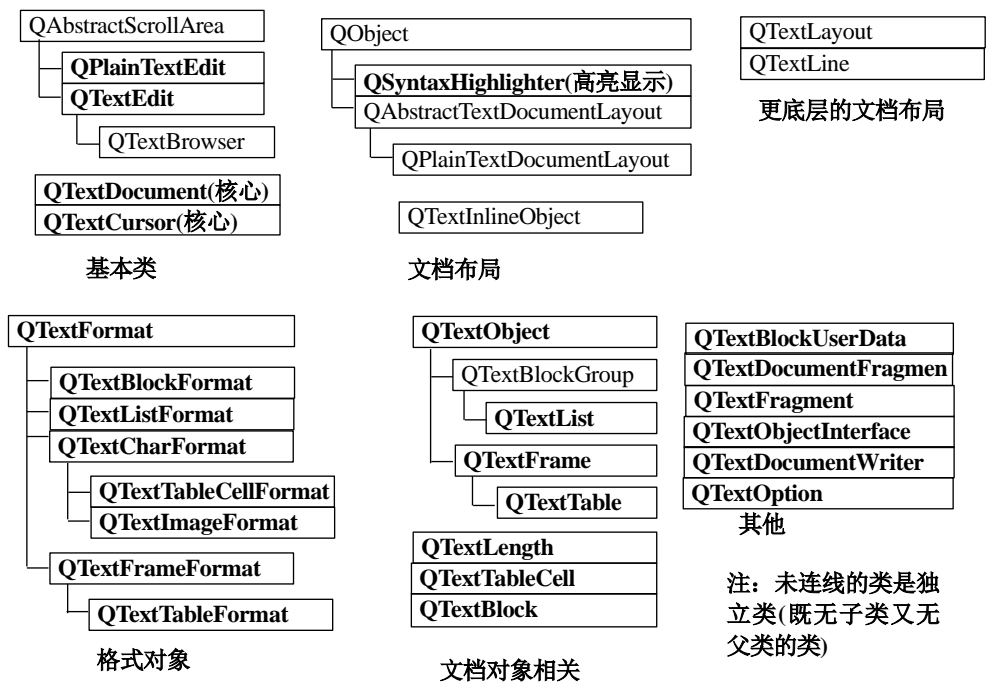
[11.14 语法高亮：QSyntaxHighlighter 类](#)

第 11 部分 Qt 文本系统

注意：本程序都假设读者在 pro 文件中已添加了正确的 QT+=widgets 语句，文中不再重复累述添加此语句。

本文注重讲解原理，因此使用的是手写的 Qt 程序。

本章讲解的类及继承关系如下图所示(仅讲解粗体字的类)



注：从本章起，会经常遇到 qreal、qint32、qint64、qint8 等 Qt 自定义类型，这些类型是使用 typedef 重命名后的相应类型，其作用是为了保证类型的长度不变，比如 qint8 是表示 8 位长度的 int 类型，qreal 是 double 类型(除非在 Qt 配置了 -qreal float 选项)。注意：C++ 语法只规定了 short、int、long 等类型的最低长度(详见《C++ 语法详解》)，并未规定最高长度，所以在不同编译器上可能会出现长度不一致的情形。

11.1 重要基本概念及原理

一、文档的基本分类

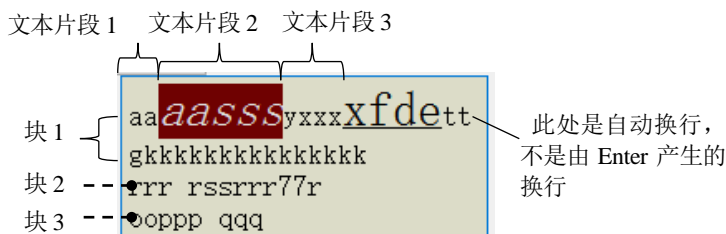
- 1、富文本：就是指的符合 HTML 语言规范的文本。Qt 的富文本文档是指由段落、框架、表格、列表、HTML 等组成的文本，因此范围更广。
- 2、文档(document)、文本(text)、段落(paragraph)、字符(char)、文档元素(文档对象)
 - ①、文档由文本组成，文本由段落、文档元素组成，段落由字符组成。
 - ②、文档元素：

文档(document)除了可以包含纯文本外，还可包含表格、列表、图像等其他对象，这些对象被称为文档元素。
 - ③、段落(paragraph)：理论上讲，一个段落可以包含表格、列表、字符、图像等内容，但实际实现时可能只会包含纯文本。

二、Qt 对文档的描述

- 1、文本片段简称片段(fragment)

文本片段是指在一文本块中具有相同属性的一部分字符的集合。
- 2、文本块简称块(block):
 - ①、块由换行符分隔，在文本编辑器中每按一次 Enter 键就产生一个文本块，即使该块什么内容也没有。注意：自动换行产生的换行不会形成一个文本块(因为不是由换行符分隔的)。
 - ②、文本块将具有不同字符格式的文本片段组合在一起，并用于表示文档中的段落(即文本块就是段落，因此这两个概念是相同的)。因此文本块包含一个或多个文本片段。
 - ③、文本块在 Qt 中用于分隔其他文档元素，由于这个原因，文本块不能包含表格、框架等对象



文本片段与文本块 (块)

- 3、文档对象(文档元素):

Qt 的文档通由常见的文档元素组成，即文本块、框架、表格和列表、图像。在 Qt 中，每种文档元素都使用一个类进行描述，比如 QTextTable 类用于描述一个表格文档元素，由于这些类大多继承自 QTextObject 类，因此文档元素也被称为文档对象。
- 4、框架(frame)

框架就是一个带边框的矩形，框架提供了文档各部分之间的逻辑分离，框架中可以包含文档的所有文档元素，也就是说框架中可以含有文本块、框架自身(即框架可嵌套)、表格、列表、图像、HTML 等。
- 5、表格(table):

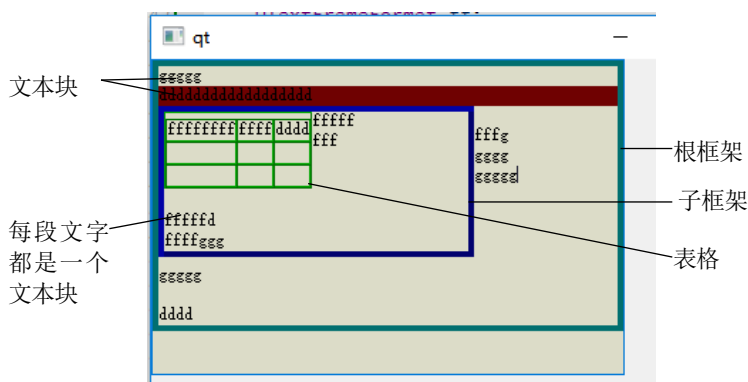
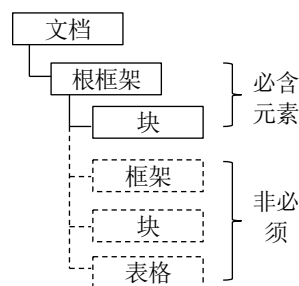
表格是一种特殊类型的框架，由许多单元格组成，每个单元格可以包含更多的框架、文本块、表格等，也就是说表格也可以嵌套。

6、图像(image):

图像与文档中的其他文档元素不同，图像由特殊格式的文本片段表示。这使得图像可以与周围的文本排成一行。

三、Qt 文档的组织结构(见右下图)

- 1、每个文档都包含一个根框架，除根框架之外的所有框架都具有父框架。
- 2、每个框架必须至少包含一个文本块(即使该文本块是空的)，以使文本光标可在其中插入新的文档元素。
- 3、即使文本块不含任何信息，框架和表格的也总是由文本块分隔开来，这确保了可以在结构之间插入新元素
- 4、由以上规则可见，每个框架至少包含一个文本块，以及零个或多个子框架。框架和表格主要用于对其他结构进行分组，而文本块才是实际上显示的信息。



Qt 文档实际结构

四、Qt 文档的实现

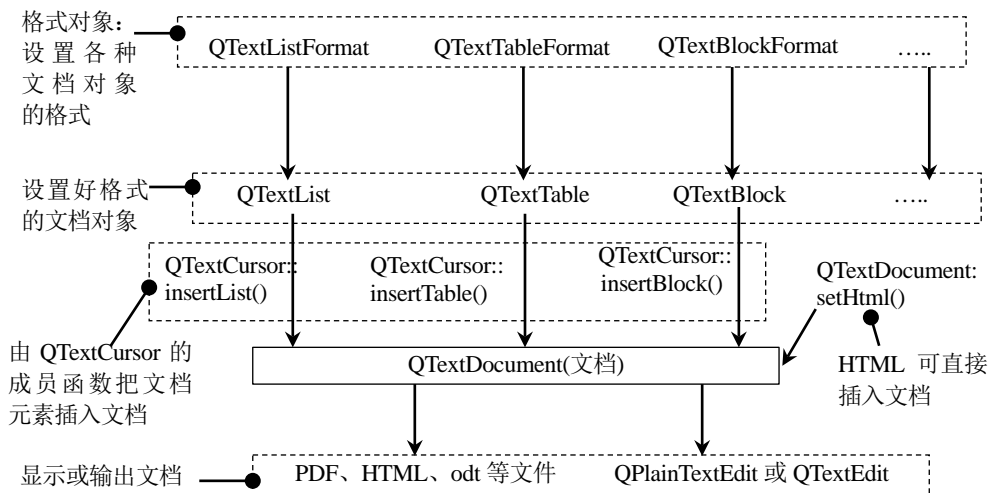
- 1、文档元素：每种文档元素都使用一个类进行描述，比如 QTextTable、QTextBlock 等。
- 2、格式对象：每种文档元素的格式(大多是外观)都使用一种相对应的格式类进行描述，比如 QTextTable 使用 QTextTableFormat 类来描述其格式，这些描述格式的类都继承自 QTextFormat 类，本文把描述文档元素格式的类称为格式对象或格式类。
- 3、QTextDocument 类：
 - ①、现实中，不同的文档元素存储在一个文档中，因此，需要把使用由各种类描述的文档对象，使用一个类来统一存储，这个类就是 QTextDocument，该类可以容纳各种文档元素，因此 QTextDocument 是文档元素的一个容器。
 - ②、由上一条原理可见，要编辑文档，首先需要创建一个 QTextDocument 对象，然后把设计好的其他文档元素(文档对象)添加到 QTextDocument 对象中。
- 4、编辑文档(QTextCursor 类):

- ①、虽然 `QTextDocument` 类是用来描述文档的，但是该类并不直接提供对文档元素修改的接口(API)，也未提供添加其他文档元素(除 HTML 外)的接口，因此要修改或向 `QTextDocument` 中添加文档元素，需要使用另一个类 `QTextCursor` 来完成。
- ②、`QTextCursor` 类主要用于管理插入符，还能够把表格或列表等复杂对象插入到 `QTextDocument` 中，并处理选择。该类可以创建/删除选择、并检索文本的内容。

5、显示文档

- ①、在 `QTextDocument` 类中的内容编辑好之后需要显示出来，有两种方法可显示 `QTextDocument` 中的内容。
- ②、直接输出到文件，比如将 `QTextDocument` 输出为 PDF、odt、ps、HTML 等文件。使用这种方式显示 `QTextDocument` 文档是完全以编程的方式编辑的 `QTextDocument` 文档，不够直观。
- ③、使用 `QPlainTextEdit` 或 `QTextEdit` 类来显示 `QTextDocument` 的内容并以所见即所得的方式进行编辑，但是除纯文本外，图像、表格等元素，还是需要以编程的方式进行编辑。注意：`QPlainTextEdit` 或 `QTextEdit` 类默认拥有一个内置的 `QTextDocument` 对象，使用这两个类时就是在编辑这个内置的 `QTextDocument` 对象。

6、具体流程见下图



注意：格式对象不一定是由相应的类创建的

Qt 文档编辑流程图

- 7、因为 `QTextDocument` 在大多数情况下仅仅是作为一个容器，在其他章节只需知道怎样获取 `QTextDocument` 以及怎样把文档元素添加到 `QTextDocument` 中即可。因此本文会把 `QTextDocument` 和 `QTextCursor` 类放在比较靠后的章节进行讲解，

11.2 QPlainTextEdit 类

为避免与鼠标光标的区别，本文有时会使用插入符表示文档光标。

鼠标光标可使用 viewport() 返回视口部件，然后对光标进行修改。

QPlainTextEdit 和下一小节讲解的 QTextEdit 类，其图形界面的文本编辑是比较简单的，麻烦的是与 QTextDocument 和 QTextCursor 有关的成员函数，因为 QTextDocument 和 QTextCursor 关系到 Qt 文本系统的全局，因此本小节和下一小节先列出 QPlainTextEdit 和 QTextEdit 类中的函数，对于有关 QTextDocument 和 QTextCursor 类的函数不作深讲。

一、基本原理

1、QPlainTextEdit 类继承自 QAbstractScrollArea 类，该类用于编辑和显示纯文本，该类不支持表格和嵌入式框架，并且不使用高精度的像素滚动方式，而是使用逐行逐段滚动的方式，但是该类更有性能优势，并且能处理更大的文档。

2、向 QPlainTextEdit 类中添加和获取内容的函数如下

1)、setDocument()、document(); //文档

2)、setPlainText()、insertPlainText()、appendPlainText()、toPlainText(); //纯文本

3)、appendHtml(); //这是向 QPlainTextEdit 中添加 HTML 文本的唯一方法

3、要设置字符的格式，需使用以下函数：

setCurrentCharFormat() 设置格式、currentCharFormat() 获取格式、mergeCurrentCharFormat() 合并格式

4、要实现自定义弹出菜单，需重新实现以下函数

contextMenuEvent(); //弹出菜单在此函数返回

createStandardContextMenu(); //获取编辑框内置的弹出菜单

5、要实现拖放或粘贴自定义的 MIME 数据需重新实现以下函数

canInsertFromMimeData() //检测拖放进来的数据是否合法(复制/粘贴不需要此函数)

createMimeDataFromSelection() //创建需要拖放(或粘贴)的数据

insertFromMimeData() //拖放和粘贴的数据在此函数内处理。

6、因为 QPlainTextEdit 是所见即所得的文本编辑部件，因此对于可视化的操作比较简单，就不多讲了，比较麻烦的在于以编程的方式来编辑文本，下列为大致步骤

1)、获取和设置需要编辑的底层文档：document()、setDocument()

2)、光标(插入符)的获取和设置：textCursor()、setTextCursor()、moveCursor()

7、QPlainTextEdit 类默认支持以下表格的键盘操作

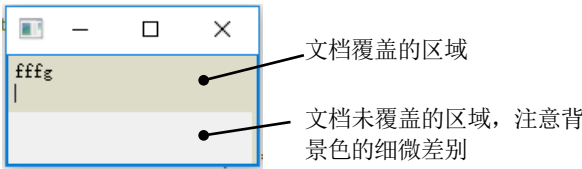
QPlainTextEdit 类默认支持键盘操作	
按键	说明
四个方向键	向指定方向移动一个字符
Ctrl + 四个方向键	向指定方向移动一个单词
Shift + 四个方向键	向指定方向选择文本。
Home、End	移至行的开头或结尾
Ctrl + Home、Ctrl+End	移至文本开头或结尾

PageUp、PageDown	向上或向下移动一个页面(视口)。
Ctrl + A、Ctrl+C、Ctrl+V、Ctrl+X	全选、复制、粘贴、剪切
Ctrl+K	删除到行尾
Ctrl+Z、Ctrl+Y	撤消、重做
Backspace、Delete	删除左侧或右侧字符
Shift+Delete、Shift+Insert	剪切、粘贴
Alt + 鼠标滚轮	水平滚动
Ctrl + 鼠标滚轮	放大文本

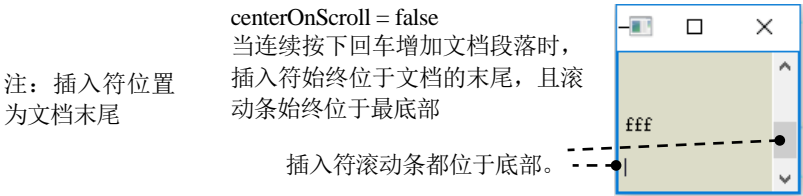
二、QPlainTextEdit 类中的属性

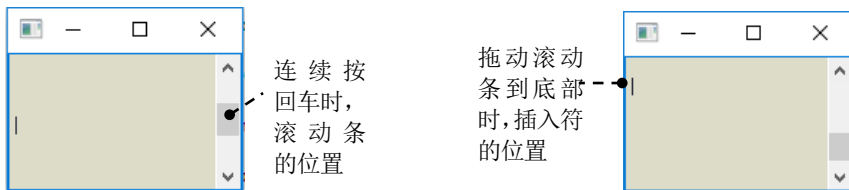
QPlainTextEdit 类属性速查表			
属性	说明	属性	说明
backgroundVisible	调色板背景	placeholderText	占位符文本
blockCount	当前块数	plainText	获取或重置文本
centerOnScroll	滚动使插入符居中	readOnly	是否只读
cursorWidth	插入符宽度	tabChangesFocus	tab 是否输入制表符
documentTitle	文档标题	tabStopDistance	制表符距离
maximumBlockCount	最大块数	textInteractionFlags	与文本交互的方式
overwriteMode	插入(覆盖)模式	undoRedoEnabled	是否启用撤消重做
lineWrapMode	换行模式(不断字)	wordWrapMode	换行模式(可断字)

- 1、backgroundVisible : bool 访问函数: bool backgroundVisible() const; void setBackgroundVisible(bool)
- 是否设置调色板背景，该属性可用于在视觉上区分哪些区域是文档未覆盖的区域，哪些是文档覆盖的区域，默认为 false(既未区分)。效果见下图



- 2、blockCount : const int 访问函数: int blockCount() const
- 获取文档的块数，默认情况下，空文档为 1。对于纯文本，通常一个段落(使用换行符分隔)就相当于 1 块。
- 3、centerOnScroll : bool 访问函数: bool centerOnScroll() const; void setCenterOnScroll(bool)
- 滚动时是否可以使插入符居中，设置该属性后，还允许编辑器滚动到文档的末尾之下。默认为 false，原理见下图





`centerOnScroll = true`

注：插入符位置为文档末尾

当连续按下回车增加文档段落时，插入符可以位于文档的中间，且滚动条也位于中间，并且拖动滚动条，可使其滚动到文档末尾之后的位置。

4、**cursorWidth** : int 访问函数: `int cursorWidth() const; void setCursorWidth(int)`

以像素为单位设置插入符的宽度，默认为 1。

5、**documentTitle** : QString

访问函数: `QString documentTitle() const; void setDocumentTitle(const QString &)`

获取和设置文档的标题，注意：不是 `QPlainTextEdit` 的标题。默认为空字符串。

6、**maximumBlockCount** : int

访问函数: `int maximumBlockCount() const; void setMaximumBlockCount(int)`

文档的最大块数。若文档中的块数多于设置的最大块数，则从文档的开头删除，此属性会立即作用于文档，且该属性还会禁用撤消/重做历史。若该属性为负或 0，则表示有无限的块数，默认为 0。

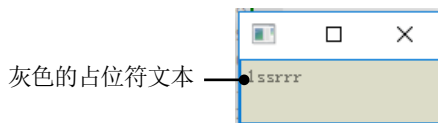
7、**overwriteMode** : bool 访问函数: `bool overwriteMode() const; void setOverwriteMode(bool)`

用户输入的文本是否覆盖现有文本，设置此属性类似于 office 中的插入功能。默认为 false(不会覆盖)。

8、**placeholderText** : QString qt5.3

访问函数: `QString placeholderText() const; void setPlaceholderText(const QString &)`

设置占位符文本，只要 `document()` 为空，便会使编辑器显示一个灰色的占位符文本。



9、**plainText** : QString 访问函数: `QString toPlainText() const; void setPlainText(const QString &);` 槽

信号: `void textChanged()`

获取或设置编辑器的内容，设置此属性时，之前的内容将被删除，且撤消/重做历史会被重置。默认情况下对于没有内容的编辑器，此属性为空字符串。

10、**readOnly** : bool 访问函数: `bool isReadOnly() const; void setReadOnly(bool)`

编辑器是否为只读。默认为 false。

11、**tabChangesFocus** : bool 访问函数: `bool tabChangesFocus() const; void setTabChangesFocus(bool)`

描述 tab 键是用于输入制表符，还是用于接受焦点。默认为 false(不接受焦点，输出制表符)。此属性若为 true，需在按下 tab 键有另一个部件接受焦点时才会起作用。

12、**tabStopDistance** : qreal //qt5.10

访问函数: `qreal tabStopDistance() const; void setTabStopDistance(qreal)`
设置制表符的距离(以像素为单位), 默认为 80。

13、**textInteractionFlags** : `Qt::TextInteractionFlags`

访问函数: `Qt::TextInteractionFlags textInteractionFlags() const;`

`void setTextInteractionFlags(Qt::TextInteractionFlags)`

设置在显示文本时, 应如何与用户输入交互。默认取决于编辑器是否是只读。该属性的使用详见第 4 章中 `QLabel` 部件的讲解。

Qt::TextInteractionFlag 枚举

标志: `Qt::TextInteractionFlags`

作用: 描述文本显示部件与用户的交互方式

成员	值	说明
<code>Qt::NoTextInteraction</code>	0	不可与文本交互
<code>Qt::TextSelectableByMouse</code>	1	可使用鼠标选择文本, 并可使用鼠标的上下文菜单或键盘快捷键把文本复制到剪贴板。
<code>Qt::TextSelectableByKeyboard</code>	2	可使用键盘上的光标键选择文本, 并显示一个文本光标
<code>Qt::LinksAccessibleByMouse</code>	4	链接可使用鼠标激活
<code>Qt::LinksAccessibleBykeyboard</code>	8	链接可使用 <code>tab</code> 键获得焦点, 并使用 <code>enter</code> 键激活。
<code>Qt::TextEditable</code>	16	文本是可编辑的。
<code>Qt::TextEditorInteraction</code>	<code>TextSelectableByMouse TextSelectableByKeyboard TextEditable</code> 文本编辑器的默认值	
<code>Qt::TextBrowserInteraction</code>	<code>TextSelectableByMouse LinksAccessibleByMouse LinksAccessibleBykeyboard</code> , 这是 <code>QTextBrowser</code> 的默认值。	

14、**undoRedoEnabled** : `bool`

访问函数: `bool isUndoRedoEnabled() const; void setUndoRedoEnabled(bool)`

是否启用撤消/重做, 默认为 `true`(启用)。

15、**lineWrapMode** : `LineWrapMode`

访问函数: `LineWrapMode lineWrapMode() const; void setLineWrapMode(LineWrapMode)`

设置文本的换行模式。枚举 `LineWrapMode` 就只有两个值, 即 `QPlainTextEdit::NoWrap`(不换行)和 `QPlainTextEdit::WidgetWidth`(默认)。`WidgetWidth` 表示在右侧空白处换行, 以保持整个单词是完整的。若想使用其他换行方式, 可使用 `wordWrapMode` 属性。

16、**wordWrapMode** : `QTextOption::WrapMode`

访问函数: `QTextOption::WrapMode wordWrapMode() const;`

`void setWordWrapMode(QTextOption::WrapMode)`

设置文本的换行模式。默认为 `QTextOption::WrapAtWordBoundaryOrAnywhere`。枚举 `QTextOption::WrapMode` 见后面章节 `QTextOption` 类

三、`QPlainTextEdit` 类中的函数

1、**QPlainTextEdit**(`QWidget *parent = Q_NULLPTR`); //构造函数

QPlainTextEdit(`const QString &text, QWidget *parent = Q_NULLPTR`)

2、字符格式

- 1)、QTextCharFormat **currentCharFormat()** const
返回插入新文本时使用的字符格式。
- 2)、void **setCurrentCharFormat**(const QTextCharFormat &*format*)
设置字符格式，若有选择的文本，则格式应用于该文本，否则格式应用于插入符处(若在该处接着输入文本，该文本会以设置的格式显示)。
- 3)、void **mergeCurrentCharFormat**(const QTextCharFormat &*modifier*);
把字符格式 *modifier* 合并到文本。若有选择的文本，则格式应用于该文本，否则格式应用于插入符处(若在该处接着输入文本，该文本会以设置的格式显示)。比如，若之前的文本为斜体，若调用此函数再把粗体合并到该文本，则最后的结果是粗斜体。

3、插入符

- 4)、void **ensureCursorVisible**(); //滚动文本，以确保插入符可见。
- 5)、void **setTextCursor**(const QTextCursor &*cursor*)
QTextCursor **textCursor()** const
设置或返回当前可见光标的 QTextCursor 副本，注意：返回的光标不会影响编辑器当前的光标。
- 6)、QTextCursor **cursorForPosition**(const QPoint &*pos*) const
在 *pos* 位置(视口坐标)返回一个 QTextCursor。
- 7)、QRect **cursorRect**(const QTextCursor &*cursor*) const; //返回包含 *cursor* 的矩形(视口坐标)
- 8)、QRect **cursorRect**() const
返回文本编辑器插入符的矩形(视口坐标)。该函数可获取编辑器中的插入符的坐标位置，比如(52,18,1,14)，则表示插入符位于(52,18)处，其宽度为 1，高度为 14。
- 9)、QList<QTextEdit::ExtraSelection> **extraSelections**() const
void **setExtraSelections**(const QList<QTextEdit::ExtraSelection> &*selections*)
使用由 *selections* 指定的颜色临时标记文档中的某些区域。QTextEdit::ExtraSelection 枚举参见 QTextEdit 类。
- 10)、void **moveCursor**(QTextCursor::MoveOperation operation,
QTextCursor::MoveMode mode = QTextCursor::MoveAnchor)
通过 *operation* 移动插入符，此函数可用于以编程的方式操作插入符。两个枚举形参见 QTextCursor 类。第 2 个参数用于控制插入符是否固定不动，比如
`moveCursor(QTextCursor::Left, QTextCursor::MoveAnchor); //向左移动一个字符。`
`moveCursor(QTextCursor::Left, QTextCursor::KeepAnchor); //向左移动一个字符并选择该字符，相当于移动的同时按住了 Shift 键。`
- 11)、QTextDocument ***document**() const; //返回指向底层文档的指针。
void **setDocument**(QTextDocument **document*)
把 *document* 设置为编辑器的新文档，编辑器不会获取 *document* 所有权，*document* 必须具有继承 QPlainTextDocumentLayout 的文档布局。

4、查找、锚点、打印、资源及判断是否可粘贴

- 12)、bool **find**(const QString &exp, QTextDocument::FindFlags options = QTextDocument::FindFlags())
 bool **find**(const QRegExp &exp, QTextDocument::FindFlags options = QTextDocument::FindFlags())
- 使用给定的规则 options(即向前、向后查找、是否区分大小写)查找下一个与 exp 匹配的字符串, 若找到 exp 则选择该文本, 并返回 true。枚举 QTextDocument::FindFlag 见 QTextDocument 类。
 - 第 2 个函数会忽略 QTextDocument::FindCaseSensitively(即是否忽略大小写), 而使用 QRegExp::caseSensitivity。
- 13)、QString **anchorAt**(const QPoint &pos) const
 返回点 pos 处锚点的引用, 若 pos 处不存在锚点, 则返回一个空字符串。
- 14)、bool **canPaste**() const
 返回文本是否可以从剪贴板粘贴到编辑器中。比如若剪贴板中是从屏幕上剪切的图片时, 便不能被粘贴到编辑器中。注意, 若复制的是文件, 则会粘贴该文件的路径, 因此是可粘贴的。
- 15)、void **print**(QPagedPaintDevice *printer) const
 使用 printer 打印编辑器的文档。该函数还支持 QPrinter::Selection 作为打印范围。
- 16)、virtual QVariant **loadResource**(int type, const QUrl &name); //虚拟的
 加载由 type 和 name 指定的资源, 该函数是 QTextDocument::loadResource()的扩展。

5、槽

- 17)、void **copy**(); void **cut**(); void **paste**(); void **undo**(); void **redo**(); void **selectAll**();
 以上槽表示复制、剪切、粘贴、撤消、重做、选择所有文本
- 18)、void **clear**()
 删除编辑器中的所有文本, 注意: 还会清除撤消/重做历史记录。
- 19)、void **appendHtml**(const QString &html); //向编辑器追加一个带有 html 的新段落。
 void **appendPlainText**(const QString &text); //向编辑器追加一个带有 text 的新段落。
 void **insertPlainText**(const QString &text); //在当前插入符位置插入文本 text。
- 20)、void **zoomOut**(int range = 1); //缩小。range 为缩小系数, 指定大于 1 的数就会缩小。
 void **zoomIn**(int range = 1); //放大, range 为放大系数, 指定大于 1 的数就会放大。
- 21)、void **centerCursor**()
 滚动文档并使插入符垂直居中, 也就是把文档滚动到插入符所在处, 并使插入符垂直居中。

6、受保护的函数

- 22)、QRectF **blockBoundingGeometry**(const QTextBlock &block) const; //受保护的
 以内容坐标的形式返回文本块 block 的边界矩形, 使用 contentOffset()函数转换矩形以获得视口上的坐标。
- 23)、QRectF **blockBoundingRect**(const QTextBlock &block) const; //受保护的
 返回文本块 block 在块自身坐标中的边界矩形
- 24)、QTextBlock **firstVisibleBlock**() const; //返回第一个可见的块。 受保护的
- 25)、QPointF **contentOffset**() const; //受保护的

返回在视口坐标中内容的原点。编辑器内容的原点始终位于第一个可见文本块的左上角。以下情况内容偏移不同于(0,0),

- 可见文本不以第一个可见块的第一行开始时, 比如, 当文本已被水平滚动或第一个可见块已经部分在屏幕上滚动时。
- 当第一个可见块是第一块且编辑器显示一个页边距时。

26)、QAbstractTextDocumentLayout::PaintContext **getPaintContext()** const; //受保护的
返回视口的 PaintContext(绘图上下文), 仅在重新实现 paintEvent()时有用。

7、信号

1)、void **blockCountChanged**(int *newBlockCount*);

当块数发生变化时, 发送此信号, newBlockCount 为新的块数。

2)、void **selectionChanged**(); //每当选择改变时, 都会发送此信号。

3)、void **copyAvailable**(bool *yes*);

在编辑器中的文本被选择或取消所有选择时发送此信号, 若有文本被选择, 则 yes 为 true, 若没有文本被选择则 yes 为 false, 与 selectionChanged()信号不同, 该信号只要选择在改变就会发送, 但是仍可能会有文本被选择, 此时就不会发送 copyAvailable 信号。
槽 copy()可使用此函数的参数 yes 的值来决定是否可以复制文本。

4)、void **cursorPositionChanged**(); //当插入符的位置改变时, 发送此信号。

5)、void **modificationChanged**(bool *changed*);

只要文档被修改, 就发送此信号, 若 changed 为 true, 则表示文档已修改,

6)、void **textChanged**()

只要文档内容发生变化就发送此信号, 该信号与 modificationChanged 的区别是, 比如我们在同一个字符上重复使用同一个格式, 此时会连续发送该信号, 但 modificationChanged 信号只在第一次时发送, 因为只有第一次文档才是修改文档。

7)、void **redoAvailable**(bool *available*); //只要重做操作变为可用或不可用, 就会发送此信号。

8)、void **undoAvailable**(bool *available*); //只要撤消操作变为可用或不可用, 就会发送此信号。

9)、void **updateRequest**(const QRect &*rect*, int *dy*)

当文本文档需要更新矩形 rect 时, 发送此信号。只要文档有更新就会发送该信号, 比如若文档中的插入符正在闪烁, 拖动滚动条(如果有)都会发送此信号, 当拖动垂直滚动条时, dy 表示滚动一次时移动的像素数, 若不拖动滚动条, 则 dy 为 0。

8、重新实现以下虚函数可处理自定义拖放的数据。

27)、virtual bool **canInsertFromMimeData**(const QMimeData **source*) const; //虚拟的, 受保护的

若 MIME 数据 source 的内容可解码并可插入到文档中, 则返回 true。该函数会在拖动操作期间当鼠标进入该部件时调用, 此时应确定是否可接受拖放的数据, 若返回 false, 则拖放的数据不会被接受。

28)、virtual QMimeData ***createMimeDataFromSelection**() const; //虚拟的, 受保护的

返回一个新的 MIME 数据对象(注意: 返回的是一个新对象), 对象的内容为编辑器当前所选择的文本, 返回的对象的所有权被传递给调用者。该函数在执行拖放操作或把数据复制到剪贴板时会被调用, 返回的对象就是剪贴板中的内容。

29)、virtual void **insertFromMimeData**(const QMimeData *source); //虚拟的, 受保护的

把由 source 指定的 MIME 数据插入到当前插入符位置处。当使用剪贴板插入或编辑器接受拖放操作中的数据时, 会调用此函数。注意: 此函数内不能调用 paste() 函数粘贴数据, 否则会陷入死循环。

9、菜单处理

30)、virtual void **contextMenuEvent**(QContextMenuEvent* e); //虚拟的, 受保护的

对 QWidget::contextMenuEvent 的重新实现, 显示使用 createStandardContextMenu() 创建的上下文菜单(即点击鼠标右键弹出的菜单), 若需要自定义或扩展标准上下文菜单需要重新实现此函数。若要禁用弹出菜单, 可设置 QWidget::contextMenuPolicy 属性为值 Qt::NoContextMenu。

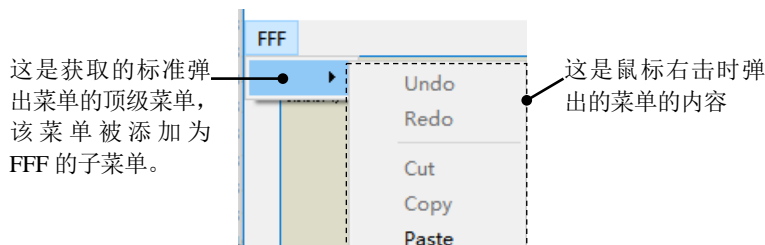
31)、QMenu ***createStandardContextMenu**()

QMenu ***createStandardContextMenu**(const QPoint &position); //qt5.5

创建标准上下文菜单, 默认由 contextMenuEvent() 函数调用。弹出菜单的所有权被转移给调用者。以上函数可用于快速创建与该编辑器完全相同的菜单, 比如

QMenu *pm = ptxt->createStandardContextMenu();

则菜单 pm 与编辑器 ptxt 的弹出菜单完全相同, 注意: 若要把 pm 添加到其他菜单或菜单栏时, 需要注意的是 createStandardContextMenu() 返回的是一个弹出菜单, 这个菜单的顶级菜单是没有文本的。效果见下图



示例: setExtraSelections 函数的使用

//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>
class B:public QWidget{    Q_OBJECT
public:    QPushButton *pb1;    QPlainTextEdit *pt;
    B(QWidget *p1=0):QWidget(p1){
        pt=new QPlainTextEdit(this); pt->resize(333,222);
        pb1=new QPushButton("Blod",this);    pb1->move(22,244);
        pb1->setShortcut(QKeySequence("Ctrl+F"));
        QObject::connect(pb1,&QPushButton::clicked,this,&B::f1);
    }
public slots:
    void f1(){

        QTextCursor cur=pt->textCursor();    //获取编辑器内部文档的光标
```

```

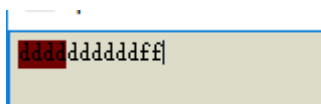
cur.movePosition(QTextCursor::Start); //移动光标至文档开头
cur.movePosition(QTextCursor::Right, QTextCursor::KeepAnchor, 4); //向右选择 4 个字符

QTextCharFormat f; //创建字符格式
f.setBackground(QBrush(QColor(111, 1, 1))); //设置背景色为红色
QTextEdit::ExtraSelection t; //创建 ExtraSelection 对象
t.cursor=cur; t.format=f; //给对象 t 的成员变量赋值
QList<QTextEdit::ExtraSelection> s;
s.append(t); //把 t 添加到列表
pt->setExtraSelections(s); //使用 setExtraSelections 函数设置字符的格式
}};
#endif // M_H

//m.cpp 文件内容
#include "m.h"
int main(int argc, char *argv[]) { QApplication app(argc, argv);
    B w; w.resize(444, 333); w.show(); return app.exec(); }

```

运行结果及说明



在文档中输入一些字符，然后按下"Ctrl+F"文本的前 4 个字符便被设置为红色的背景

示例：菜单、拖放、复制/粘贴

```

//m.h 文件的内容
#ifndef M_H
#define M_H
#include<QtWidgets>
class C:public QPlainTextEdit{
public:
    C(QWidget *parent=0):QPlainTextEdit(parent) { }
    //自定义弹出菜单
    void contextMenuEvent(QContextMenuEvent* e){
        QMenu *pm=createStandardContextMenu(); //创建一个与 Qt 内部弹出菜单内容相同的菜单
        pm->addAction("XXX");
        pm->exec(QCursor::pos()); //在鼠标位置弹出菜单
    }
}
//本示例仅演示以下 3 个函数对复制/粘贴和拖放的影响
bool canInsertFromMimeData(const QMimeData *s) const{
    /*当在文档中拖放数据时，会调用此函数，此时可判断拖放的数据 s(即剪贴板中的数据)是否符合要求，
    以启用或禁用拖放。*/
    //验证剪贴板中的数据 s 就是 createMimeDataFromSelection() 返回的对象。
    qDebug()<<"AAA="<<s->text();
    return true;
    //return false; //①、若返回 0 拖放将被禁止
}

QMimeData *createMimeDataFromSelection() const{
    //当拖放或复制/剪切时会调用此函数，该函数的返回值就是拖放或复制/剪切后剪贴板中的内容。
}

```

```

        QMimeData *md=new QMimeData;        md->setText("YYY");        return md;
    }
    //当拖放完成或粘贴时调用以下函数，也就是说粘贴时向编辑器中最终插入的数据由此函数决定。
    //void insertFromMimeData(const QMimeData *s){ insertPlainText("ZZZ");} //②
};

class B:public QWidget{    Q_OBJECT
public:    C *pt;
    B(QWidget *p1=0):QWidget(p1){
        pt=new C(this); pt->resize(111,111);pt->move(22,33);
        //向编辑器中添加一个菜单
        QMenuBar *pmb=new QMenuBar(this);
        //创建一个与 Qt 内部弹出菜单内容相同的菜单
        QMenu *pml=pt->createStandardContextMenu(QPoint(111,111));
        pmb->addMenu(pml);    pml->addAction("SSS");    pml->menuAction()->setText("TTT");
    };
#endif // M_H

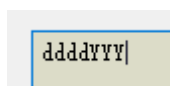
//m.cpp 文件内容
#include "m.h"
int main(int argc, char *argv[]){    QApplication app(argc,argv);
    B w;    w.resize(444,333);    w.show();    return app.exec(); }

```

1、运行结果及说明



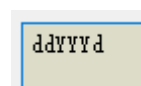
剪切图中文字



然后粘贴



拖动图中文字

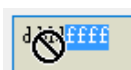


然后放下

由图可见，无论我们在编辑器中复制/剪切的是什么内容，最后都将被粘贴为 createMimeDataFromSelection() 函数返回的数据，拖动类似。因此可见 createMimeDataFromSelection() 函数返回的数据就是剪贴板中的数据。另外，当我们拖动数据时，程序会不断的输出一些字符，此时调用的是 canInsertFromMimeData() 函数，复制/剪切时不会调用该函数。

2、把示例中①处的注释取消掉，并把 return true; 注释掉，

此时将禁用拖放，复制/粘贴不受影响，如右图



拖放被禁用了

3、把②处代码的注释取消掉，以使 insertFromMimeData() 函数可用，此时无论复制/剪切或拖放的是什么数据，在粘贴时，都会插入字符串"ZZZ"，而不是插入的剪贴板中的内容。

4、示例实现的菜单比较简单，就不举图示说明了。

11.3 QTextEdit 类

该类的大部分操作与 QPlainTextEdit 相同，因此本章将只列出其属性和函数。

1、QTextEdit 类继承自 QAbstractScrollArea 类，该类用于编辑和显示纯文本和富文本，该类可显示表格、图像、列表等。对于富文本，则使用 HTML 4 进行描述。使用 QLabel 可显示一小段富文本。

2、若 QTextEdit 提供的富文本功能不能满足要求，则可使用 Qt WebKit

3、QTextCursor 类能够把表格或列表等复杂对象插入到文档中，并处理选择。该类可以创建/删除选择、并检索文本的内容。要设置选择，只需在 QTextCursor 对象上创建一个选择，然后使用 setTextCursor()把插入符设置为可见插入符，然后就可以通过 copy()或 cut()来复制数据了。

2、QTextEdit 默认支持的键盘操作与 QPlainTextEdit 类相同

2、向 QTextEdit 类中添加和获取内容的函数如下

1)、setDocument()、document(); //文档

2)、setPlainText()、toPlainText()、insertPlainText()、appendPlainText(); //纯文本

3)、setHtml()、toHtml()、insertHtml(); //富文本(新增)

4)、append()、setText(); //添加纯文本和富文本(新增)。

该类没有 appendHtml()函数，但相对于 QPlainTextEdit 增加了不少函数

3、要设置字符的格式，可使用以下函数：

setCurrentCharFormat()设置格式、currentCharFormat()获取格式、mergeCurrentCharFormat()合并格式
还可使用以下函数

setCurrentFont()字体、setFontFamily()字体族、setFontItalic()斜体、setFontPointSize()大小、

setFontUnderline()下划线、setFontWeight()权重(粗细)

其读取函数只需去掉前面的 set 并把首字母改为小写即可。

4、设置背/前景色、对齐方式使用以下函数

setAlignment()对齐方式、setTextBackgroundColor()背景色、setTextColor()文本颜色

5、其余的函数的 QPlainTextEdit 类相同。

二、QTextEdit 类中的属性

QTextEdit 类属性速查表

属性	说明	属性	说明
acceptRichText	是否接受富文本	lineWrapColumnOrWidth	自动换行
html	HTML 接口	lineWrapMode	
autoFormatting	自动套用格式		
以下属性与 QPlainTextEdit 类中的相应属性相同，仅在此列出简表			
cursorWidth	插入符宽度	placeholderText	占位符文本
document	文档	plainText	获取或重置文本
documentTitle	文档标题	readOnly	是否只读

overwriteMode	插入(覆盖)模式	undoRedoEnabled	是否启用撤消重做
tabChangesFocus	tab 是否输入制表符	wordWrapMode	换行模式
tabStopDistance	制表符距离	textInteractionFlags	与文本交互的方式

1、**acceptRichText** : bool **访问函数**: bool acceptRichText() const; void setAcceptRichText(bool)
是否接受用户输入的富文本。默认为 true。

2、**autoFormatting** : AutoFormatting
访问函数: AutoFormatting autoFormatting() const;void setAutoFormatting(AutoFormatting)
自动格式(即自动套用格式)。AutoFormatting 枚举见下表

QTextEdit::AutoFormatting 枚举(无标志)		
作用：自动格式		
成员	值	说明
QTextEdit::AutoNone	0	不启用自动格式(默认值)。
QTextEdit::AutoBulletList	0x0000 0001	自动创建项目符号列表，比如在现有列表中按 enter 键时。
QTextEdit::AutoAll	0xffff ffff	应用全部自动格式。目前仅支持 AutoBulletList 一种。

3、**html** : QString **访问函数**: QString toHtml() const; void setHtml(const QString &); //槽
该属性为编辑器提供了一个 HTML 接口。读取函数 toHtml 以 HTML 的形式返回编辑器的文本。设置函数 setHtml()更改编辑器的文本，并删除之前的任何文本，且清除撤消/重做历史。

4、**lineWrapMode** : LineWrapMode
访问函数: LineWrapMode lineWrapMode() const; void setLineWrapMode(LineWrapMode)

5、**lineWrapColumnOrWidth** : int
访问函数: int lineWrapColumnOrWidth() const; void setLineWrapColumnOrWidth(int)
以上两属性用于设置文本的换行模式，若想使用其他换行方式，可使用 wordWrapMode 属性。枚举 LineWrapMode 取值及意义如下表：

QTextEdit::LineWrapMode 枚举(无标志)		
作用：自动格式		
成员	值	说明
QTextEdit::NoWrap	0	不换行
QTextEdit::WidgetWidth	1	(默认)。WidgetWidth 表示在右侧空白处换行，以保持整个单词是完整的
使用以下枚举，需要使用 setLineWrapColumnOrWidth()设置换行时固定的宽度。		
QTextEdit::FixedPixelWidth	2	在固定的宽度(像素)处换行
QTextEdit::FixedColumnWidth	3	在固定的宽度(列数，即字符数)处换行

二、QTextEdit 类中的函数

与 QPlainTextEdit 类相同的函数速查表(仅在此列出简表)		
类别	函数名	说明
其他	anchorAt()	获取锚点文本
	canPaste()	能否粘贴
	loadResource()	加载资源
	print()	打印文本
	find()	查找文本
插入符(光标)和选择	cursorForPosition()	获取 pos 处的光标
	cursorRect()	获取光标的矩形
	moveCursor()	以编程方式移动光标
	ensureCursorVisible()	滚动以使光标可见
	textCursor()	获取光标
	setTextCursor()	设置光标
	setExtraSelections()	设置选择
	extraSelections()	获取选择
字符格式	mergeCurrentCharFormat()	合并格式
	currentCharFormat()	获取当前格式
	setCurrentCharFormat()	设置当前格式
重新实现以拖放或粘贴自定义 MIME	canInsertFromMimeData()	能否插入 MIME
	createMimeDataFromSelection()	创建 MIME
	insertFromMimeData()	插入 MIME
重新实现以自定义弹出菜单	contextMenuEvent()	
	createStandardContextMenu()	获取弹出菜单

1、**QTextEdit**(QWidget *parent = Q_NULLPTR); //构造函数

QTextEdit(const QString &text, QWidget *parent = Q_NULLPTR)

2、以下函数除了读取函数外，其余函数都是槽函数。

QTextEdit 类的设置函数(以 set 开始的函数)几乎都有以下特点：

若有选择的文本，则设置应用于该文本，否则设置应用于插入符处(若在该处接着输入文本，该文本会以设置的格式显示)

3、其他函数

1)、void **copy**()、void **cut**()、void **paste**()、void **undo**()、void **redo**()、void **selectAll**()、

void **zoomIn**(int range = 1)、void **zoomOut**(int range = 1)

分别表示复制、剪切、粘贴、撤消、重做、全选、放大、缩小。

2)、void **clear**(); //清除所有文本，同时撤消/重做历史也被清除。

3)、void **scrollToAnchor**(const QString &name);

滚动编辑器以使名称为 name 的锚点可见，若 name 为空、或已显示或未找到，则不执行任何操作。

4、设置字体(大小、粗细、斜体等)

4)、QFont **currentFont**() const; //返回当前格式的字体

void **setCurrentFont**(const QFont &f); //设置当前格式的字体，槽

- 5)、QString **fontFamily**() const; //返回当前格式的字体族(字体系列)
void **setFontFamily**(const QString &*fontFamily*); //设置当前格式的字体族(字体系列)
- 6)、bool **fontItalic**() const; //字体是否是斜体, true 表示斜体
void **setFontItalic**(bool *italic*); //设置字体是否为斜体。
- 7)、qreal **fontPointSize**() const; //返回当前格式的字体大小(磅值, 这是点大小)
void **setFontPointSize**(qreal s); //设置字体的点大小, 若 s 为 0 或负数, 是未定义的。
- 8)、bool **fontUnderline**() const; //字体是否带下划线, true(带下划线)
void **setFontUnderline**(bool underline); //设置字体的下划线
- 9)、int **fontWeight**() const; //返回当前格式的字体权重(即粗细程度)
void **setFontWeight**(int weight); //设置字体的权重(粗细), QFont::Weight 枚举描述了一个范围。

5、添加内容(另外还有属性的 **setHtml()**、**setPlainText()**、**setDocument()**)

- 10)、void **append**(const QString &*text*);
把文本 *text* 追加到编辑器的末尾, 追加的段落与当前段落具有相同的字符和块格式。
- 11)、void **insertHtml**(const QString &*text*)
把文本以 HTML 的格式插入到当前插入符位置。注意: 在样式表中使用此函数时, 样式表只适用于文档中的当前块。要在整个文档中应用样式表, 请使用
QTextDocument::setDefaultStyleSheet()。
- 12)、void **insertPlainText**(const QString &*text*); 把本插入到当前插入符位置
- 13)、void **setText**(const QString &*text*)
设置编辑器的文本为 *text*, 该函数会猜测文本是纯文本还是 HTML, 使用 **setHtml()**或 **setPlainText()**可避免这种猜测。该函数会删除之前的所有文本。

6、设置文本格式(背/前景色、对齐方式)

- 14)、Qt::Alignment **alignment**() const; //返回当前段落的对齐方式
- 15)、void **setAlignment**(Qt::Alignment a);
设置当前段落的对齐方式, 有效对齐方式为 Qt::AlignLeft, Qt::AlignRight, Qt::AlignJustify, Qt::AlignCenter。
- 16)、void **setTextBackgroundColor**(const QColor &c); //设置当前格式的文本背景色。
QColor **textBackgroundColor**() const; //返回当前格式的文本背景色。
- 17)、void **setTextColor**(const QColor &c); //设置当前格式文本的颜色。
QColor **textColor**() const; //返回当前格式文本的颜色。

7、信号

- 1)、void **copyAvailable**(bool *yes*)、 void **cursorPositionChanged**()、 void **redoAvailable**(bool *available*)
void **undoAvailable**(bool *available*)、 void **selectionChanged**()、 void **textChanged**()
以上信号详见 QPlainTextEdit 类, 原理相同。
- 2)、void **currentCharFormatChanged**(const QTextCharFormat &*f*)
若当前字符格式发生了变化(比如由于光标位置变化), 则发送此信号。

三、QTextEdit 类中的嵌套结构 QTextEdit::ExtraSelection

- 1、QTextEdit::ExtraSelection 主要用于 setExtraSelections()函数，以实现以指定的颜色临时标记文档中的某些区域。
- 2、QTextEdit::ExtraSelection 结构拥有如下成员变量
QTextcursor **cursor**; QTextCharFormat **format**;

11.4 表格(QTextTable 和 QTextTableFormat 类)

QTextTableCell、QTextTableCellFormat、QTextLength

QTextTable 继承自 QTextFrame 类，QTextTableFormat 类继承自 QTextFrameFormat 类。其他类的继承关系参见本文开始的继承表

下面为各个类的主要作用

- 1、QTextTable: 表格类，主要作用是拆分和合并单元格
- 2、QTextTableFormat: 表格格式，该类主要设置单元格的内部边距、单元格之间的距离，其他格式需调用其父类的函数设置。
- 3、QTextTableCell: 单元格类，需通过 QTextTable::cellAt() 获取单元格对象。该类只能获取一些单元格的信息
- 4、QTextTableCellFormat: 单元格格式，该类主要设置单元格的内部边距，其他格式需调用其父类的函数设置。
- 5、QTextLength: 长度，主要作用是为某一类型(比如表格的单元格)，指定某种形式的长度。比如把表格的某个单元格指定为固定的长度或可变的长度等。该类需使用 QTextTable::setColumnWidthConstraints(QVector<QTextLength>) 函数

一、表格初步

- 1、QTextTable 用于描述 QTextDocument 中的表格元素，表格是按行和列按序的一组单元格，一个表格至少有一行和一列。每个单元格包含一个块，并被框架(frame)包围。
- 2、注意：QTextTable 没有公有的构造函数，也就是说不能创建该类的对象，QTextTable 类的对象通常是通过 QTextCursor::insertTable() 的返回值创建的。函数原型如下：

```
QTextTable *insertTable(int rows, int columns, const QTextTableFormat &format)
```

```
QTextTable *insertTable(int rows, int columns)
```

- 3、向 QTextEdit 类中添加表格的方法如下(注 QPlainTextEdit 不支持表格):

示例：向 QTextEdit 中插入一个表格

//以下代码复制到 main 函数中即可运行。

```
QWidget w;    QTextEdit *pt=new QTextEdit(&w);
QTextCursor cursor(pt->textCursor());    //获取 pt 的光标，此步必不可少。
cursor.movePosition(QTextCursor::Start);    //把光标移至开始位置
//使用 QTextCursor 的成中函数创建并向 pt 插入一个 3 行 4 列的表格。
QTextTable *pta = cursor.insertTable(3, 4);    //注意：这是创建 QTextTable 对象的方法。
w.show();
```

运行结果如下图，



初始运行效果



在表格中输入数据后的效果

- 4、以上代码添加的表格不够完美，因为还对对表格进行格式设置，为此还需要用到如下一些类，QTextTableFormat、QTextTableCell、QTextTableCellFormat、QTextLength。
- 5、设置表格宽度：QTextLength 类的使用
- 该类提供了在 QTextDocument 中使用的不同类型的长度值，比如提供一个固定宽度的长度值，或提供一个以最大宽度的百分比表示的长度值等。该类需在 QTextLength 类的构造函数中指定长度，在表格中还需要通过 QTextTableFormat 的接口函数

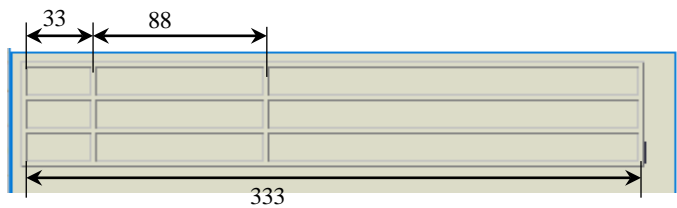
```
void QTextTableFormat::setColumnWidthConstraints(QVector<QTextLength>);
```

来使用 QTextLength 类。下面为示例代码

示例：使用 QTextLength 设置表格的宽度(以下代码复制到 main 函数中即可运行)

```
QWidget w;          QTextEdit *pt=new QTextEdit (&w);
QTextCursor cursor(pt->textCursor());    cursor.movePosition(QTextCursor::Start);
//设置表格格式
QTextTableFormat pf;
QTextLength p1(QTextLength::FixedLength,33);    //创建一个固定长度 33 的值
QTextLength p2(QTextLength::FixedLength,88);    //创建一个固定长度为 88 的值
QVector<QTextLength> v;    //把以上值 p1,p2 添加到向量
v.append(p1);    //向量的第 1 个元素设置第 1 列的宽度
v.append(p2);    //第 2 个元素设置第 2 列的宽度
pf.setColumnWidthConstraints(v);    //设置向量。
pf.setWidth(333);    /*设置表格的总宽度。这个函数是 QTextTableFormat 从父类 QTextFrame 继承来的，该函数会使表格的宽度固定为 333。其影响详见 QTextFrame 类的讲解*/
QTextTable *pta=cursor.insertTable(3, 3,pf);    //向 pt 中插入一个 3*3 使用格式 pf 描述的表格。
pt->resize(355,222);    w.resize(444,333);    w.show();
//pf. (...); .....
//pta ->setFormat(pf);    //表格的格式还可在之后使用 QTextTable::setFormat() 函数设置
```

运行结果如下：



二、QTextLength 类中的函数(该类的使用见上面的示例)

1、QTextLength()

```
QTextLength(Type type, qreal value);
```

必须使用该构造函数来指定长度，value 的意义依 type 而定，详见下表。枚举 Type 见下表(固定宽度与可变宽度的区别见后面的图示)

QTextLength::Type 枚举(无标志)

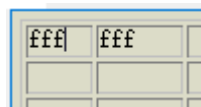
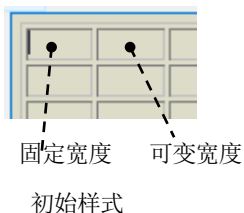
作用：描述长度的类型		
成员	值	说明
QTextLength::VariableLength	0	可变宽度。此归 value 设置的值不起作用。
QTextLength::FixedLength	1	固定宽度，此时 value 的值为像素值。
QTextLength::PercentageLength	2	宽度为最大宽度的百分比，此时 value 的值为百分比(0~100)

- 2、qreal **rawValue**() const; //返回设置的长度值，若 Type 为 VariableLength 则返回 0。
- 3、Type **type**() const; //返回设置的长度对象的类型。
- 4、qreal **value**(qreal **max**) const; //返回有效长度，由 Type 和指定的参数 max 限制。
- 5、以下为重新实现的操作符函数

operator QVariant() const

bool operator!=(const QTextLength &**other**) const

bool operator==(const QTextLength &**other**) const



由图可见，在输入内容时，固定宽度未达到右边界之前单元格不会扩展，而可变宽度的单元格，一旦输入字符就会扩展单元格

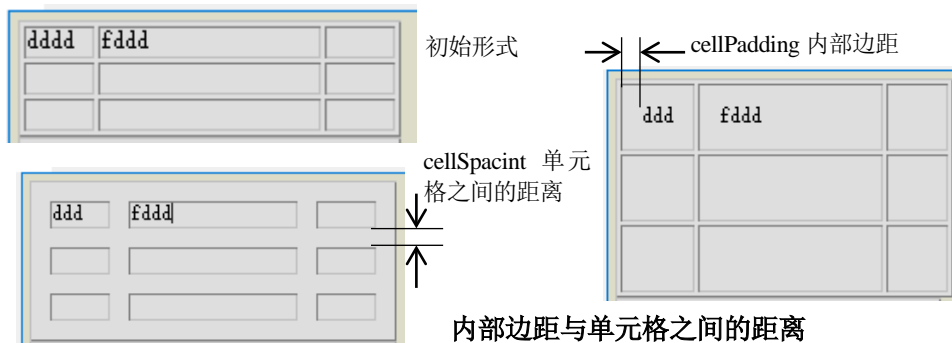
固定宽度与可变宽度的区别

三、QTextTableFormat 类中的函数(父类的函数请参阅 QTextFrameFormat、QTextFormat)

该类描述表格的格式信息，比如相邻单元格之间的间距、表格的背景色，单元格的长度等。该类设置的格式会作用于整个表格(除长度有特例外)

- 1、**QTextTableFormat**(); //构造函数
- 2、Qt::Alignment **alignment**() const; //返回表格在其父框架内的对齐方式
void **setAlignment**(Qt::Alignment **alignment**) //设置表格在其父框架内的对齐方式
- 3、qreal **cellPadding**() const; //返回表格单元格的内部边距(即单元格边界与内容之间的距离)
void **setCellPadding**(qreal **padding**) //设置表格单元格的内部边距 (见后面图示)
- 4、qreal **cellSpacing**() const; //返回表格单元格之间的距离(见后面图示)
void **setCellSpacing**(qreal **spacing**) //设置表格单元格之间的距离
- 5、QVector<QTextLength> **columnWidthConstraints**() const
void **setColumnWidthConstraints**(const QVector<QTextLength> &constraints)
以上函数用于返回和设置表格单元格的列宽度约束，其使用方法在前文已讲过。
- 6、void **clearColumnWidthConstraints**() //清除单元格的列宽度约束。
- 7、int **columns**() const; //返回表格格式约束的表格的列数
- 8、int **headerRowCount**() const; //返回标头行。表格标头在跨页时，会在页边距重复
void **setHeaderRowCount**(int **count**); 设置第 count 行为表格标头，。
- 9、bool **isValid**() const; //若表格格式有效，则返回 true。

10、设置背景色可以调用父类的 setBackground()函数



四、QTextTableCell 类中的函数

这是个独立的类，该类用于描述表格中的单个单元格，其设置只作用于该单元格，QTextTableCell 类需要使用 QTextTable::cellAt()函数来获取，示例如下：

```
QTextTableCell pc = table->cellAt(1,1); //获取第 1 行 1 列的单元格  
pc.setFormat(...); //设置该单元格的格式，其格式只作用于该单元格
```

- 1、**QTextTableCell()**; //构造函数
QTextTableCell(const QTextTableCell &other)
- 2、QTextCharFormat **format()** const; //返回单元格的字符格式。
void **setFormat(const QTextCharFormat &format)**; //设置单元格的字符格式
- 3、int **column()** const; //返回此单元格所在的列号。
int **row()** const; //返回此单元格所在的行号。
int **columnSpan()** const; //返回此单元格所跨越的列数。默认为 1。注意：跨度不能由单元格更改。
int **rowSpan()** const; //返回此单元格所跨越的行数。默认为 1。
- 4、int **tableCellFormatIndex()** const; //在文档的内部格式列表中返回表格单元格格式的索引。
- 5、QTextCursor **firstCursorPosition()** const; //返回此单元格中第一个有效的光标位置。
QTextCursor **lastCursorPosition()** const; //返回此单元格中最后一个有效的光标位置。
- 6、bool **isValid()** const; //若是有效单元格，则返回 true
- 7、QTextFrame::iterator **begin()** const; //返回指向单元格开始处的框架(frame)迭代器(其使用见 QTextFrame 类)。
QTextFrame::iterator **end()** const
- 8、以下为重新实现的操作符函数
bool operator!=(const QTextTableCell &other) const
QTextTableCell &operator=(const QTextTableCell &other)
bool operator==(const QTextTableCell &other) const

五、QTextTableCellFormat 类中的函数(父类的函数请参阅 QTextCharFormat、QTextFormat)

QTextTableCellFormat 类设置的格式只能作用于该单元格，因此可用该类可把表格的每个单元格设置成各自不同的格式，单元格中的内容通常为字符，因此通常设置的就是字符的格式，这是为什么该类继承自 QTextCharFormat 的原因。该类主要用于设置单元格的内部边距(padding)，其余格式需要使用从父类继承的函数设置。

- 1、QTextTableCellFormat(); //构造函数
- 2、void **setTopPadding**(qreal *padding*); //设置单元格上方的内部边距
void **setBottomPadding**(qreal *padding*)
void **setLeftPadding**(qreal *padding*)
void **setRightPadding**(qreal *padding*)
- 3、qreal **topPadding**() const; //获取单元格上方的内部边距。
qreal **bottomPadding**() const
qreal **leftPadding**() const
qreal **rightPadding**() const
- 4、void **setPadding**(qreal *padding*); //该函数可一次性设置 4 个方向的内部边距
- 5、bool **isValid**() const; //若是有效单元格格式，则返回 true
- 6、设置背景色可以调用父类的 setBackground()函数

六、QTextTable 类中的函数(父类的函数请参阅 QTextFrame、QTextFormat)

注意：QTextTable 没有公有的构造函数，该类主要作用是拆分和合并单元格
获取单元格的接口函数

- 1、QTextTableCell **cellAt**(int *row*, int *column*) const; //返回 row 行 column 列(从 0 开始计数)的单元格
QTextTableCell **cellAt**(int *position*) const; //返回位置 position 处的单元格。
QTextTableCell **cellAt**(const QTextCursor &*cursor*) const; //返回光标 cursor 所在的单元格

格式、光标

- 2、QTextTableFormat **format**() const; //返回表格的格式
void **setFormat**(const QTextTableFormat &*format*); //设置表格的格式
- 3、QTextCursor **rowEnd**(const QTextCursor &*cursor*) const;返回光标 cursor 所在行的行尾的光标。
QTextCursor **rowStart**(const QTextCursor &*cursor*) const;返回光标 cursor 所在行的开头的光标。

追加、删除、插入单列/行数

- 4、void **appendColumns**(int *count*); //向表格右侧追加 count 列
void **appendRows**(int *count*); //向表格底部追加 count 行。
- 5、void **insertColumns**(int *index*, int *columns*); //在索引 index 之前，插入 columns 列
void **insertRows**(int *index*, int *rows*); //在索引 index 之前，插入 rows 行
- 6、void **removeRows**(int *index*, int *rows*); //从索引 index 处开始，移除 rows 行。
void **removeColumns**(int *index*, int *columns*); //从索引 index 处开始，移除 columns 列
- 7、void **resize**(int *rows*, int *columns*); //重新设置表格的行数和列数
- 8、int **columns**() const; //返回表中的列数
int **rows**() const; //返回表中的行数

合并、拆分单元格(原理见下图)

- 9、void **mergeCells**(int *row*, int *column*, int *numRows*, int *numCols*)

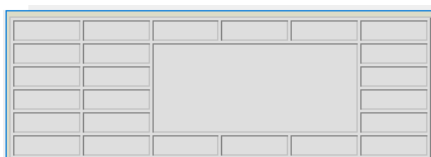
从(row, column)处的单元格向右合并 numRows 行，向下合并 numCols 列。新单元格跨越 numRows 行、numCols 列，若 numRows 和 numCols 小于单元格生成的当前行数或列数，则此方法不执行任何操作。合并后单元格的行数和列数不会改变

10、void **mergeCells**(const QTextCursor &cursor);

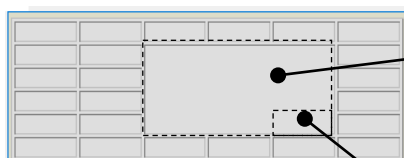
合并由光标 cursor 所选择的单元格

11、void **splitCell**(int row, int column, int numRows, int numCols)

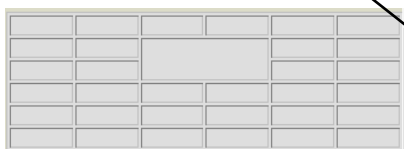
拆分单元格，只能拆分跨越多行或多列的单元格。把 row 列 column 列处的单元格拆分为多个单元格，拆分后原单元格(即拆分前的大单元格)跨越 numRows 行 numCols 列。这意味着要使单元格还原为合并之前的样式，应使 numRows 和 numCols 为 1。



mergeCells(1,2,3,3);把第 1 行 2 列的单元格向右合并 3 行，向下合并 3 行。



指定这个范围内的原来的单元格，都指向了这个大的被合并的单元格，因此都可以拆分，比如 splitCell(1,2,1,1), splitCell(1,3,1,1); splitCell(1,4,1,1); splitCell(2,2,1,1); splitCell(2,4,1,1); 等，都可以拆分这个合并的大单元格



splitCell(3,4,2,2)，使用(3,4)来指定拆分前的大单元格，拆分后大单元格仍然跨越 2 行 2 列，效果如右图

七、总结(使用表格的全部步骤)

```
QWidget w;          QTextEdit *pt=new QTextEdit (&w);    //创建文本编辑器
//1、获取当前光标，关键重点，因为要使用光标把表格插入编辑器
QTextCursor cursor= pt->textCursor();
.....    //光标位置等设置
//2、设置表格格式及长度
QTextTableFormat pf;
//2.1 设置长度(步骤较多，所以列出来)
QTextLength pl (...);QVector<QTextLength> v; v.append(pl); pf.setColumnWidthConstraints(v);
.....    //2.2 使用 pf 进行表格的其他设置
//3、使用格式 pf 插入并获取插入的表格。
QTextTable *pta =cursor.insertTable(3, 3, pf);
//pf.setFormat(...); 还可在以后使用 setFormat() 函数设置表格的格式。
5、获取单元格(以下是获取单元格的方法，获取单元格需在表格创建完成之后)
QTextTableCell pc=pta->cellAt(0, 1);
6、设置单元格格式
QTextTableCellFormat pcf;
.....
pc.setFormat(pcf);    //设置单元格格式
7、上面是一个大致过程，实际编程时应使用快捷键或关联按钮的槽来完成对表格的编辑
```

//示例：向文本编辑器中插入表格综合示例

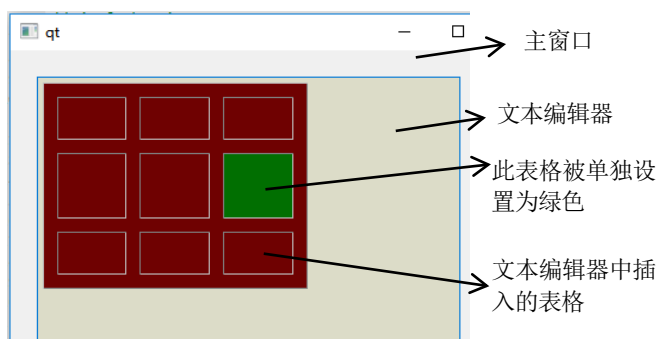
//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>
class B:public QWidget{    Q_OBJECT
public:
    QPushButton *pb1,*pb2,*pb3,*pb4,*pb5,*pb6,*pb7,*pb8,*pb9,*pb10;
    QTextEdit *pt;        //保存编辑器
    QTextTableFormat tabf; //保存表格格式
    QTextTable *tab;      //保存表格
    QTextTableCell cel;   //保存单元格
    QTextTableCellFormat celf; //保存单元格格式
B(QWidget *p=0):QWidget(p){
    //pt = new D("<a href=http://aaa>AAAA bbbb cccc dddd eeee ffff gggg kkkk</a>",this);
    pt = new QTextEdit(this); //创建文本编辑器
    pt->move(22,22);
//1、获取当前光标，
    QTextCursor cursor= pt->textCursor(); //重点，因为要使用光标把表格插入编辑器
    cursor.movePosition(QTextCursor::Start); //设置光标位置
//2、设置表格格式及长度
//2.1 设置长度和宽度
    QTextLength pl(QTextLength::PercentageLength,33);
    QVector<QTextLength> v;    v.append(pl); //向量
    tabf.setColumnWidthConstraints(v); //设置长度。
//2.2 使用 pf 进行表格的其他设置
    tabf.setWidth(222); //设置表格的总宽度。
    tabf.setCellPadding(10); //设置单元格的内部边距
    tabf.setCellSpacing(11);
    tabf.setBackground(QColor(111,1,1)); //设置整个表格的背景色为红色
//3、向 pt 中插入并返回该表格。
    tab =cursor.insertTable(3, 3,tabf); //重点:这是获取创建的表格的方法
//4、获取单元格以设置每个单元格的格式(需在创建表格之后设置)
    cel=tab->cellAt(1,2); //重点：这是获取单元格的方法
    celf.setBackground(QColor(1,111,1)); //设置单元格的背景色为绿色
    celf.setFontPointSize(22); //设置字体大小
    celf.setFontItalic(1); //字体为斜体
    cel.setFormat(celf);
    pt->resize(355,222);
}    };
#endif // M_H
```

//m.cpp 文件的内容

```
#include "m.h"
int main(int argc, char *argv[]){    QApplication app(argc,argv);
    B w;    w.resize(444,333);    w.show();    return app.exec(); }
```

运行结果见下图



11.5 框架(QTextFrame 和 QTextFrameFormat 类)

框架在外观上就是一个矩形边框。

一、文本框架(简称框架)基础

- 1、文本框架有一个边框，它把一个或多个文本块组合在一起，从而提供比段落更大的结构层，因此框架被用作其他文档元素的通用容器。框架使用 `QTextCursor::insertFrame()` 创建。
- 2、每个框架至少包含一个文本块，以使文本光标可以在其中插入新的文档元素。
- 3、框架可用于在文档中创建分层结构，每个文档都有一个根框架(`QTextDocument::rootFrame()`)，且根框架下的每个框架都有一个父框架和一个(可能是空的)子框架列表。也就是说文档是由一个或多个框架嵌套组成的。
- 4、父框架可以通过 `QTextFrame::parentFrame()` 找到，`QTextFrame::childFrames()` 函数提供了一个子框架列表。
- 5、框架通常不直接创建，而是使用 `QTextCursor::insertFrame()` 创建，原型如下：

```
QTextFrame *insertFrame(const QTextFrameFormat &format)
```

二、QTextFrame 类中的函数

- 1、`QTextFrame(QTextDocument *document);` //构造函数
- 2、`void setFrameFormat(const QTextFrameFormat &format);` //设置框架的框架格式
`QTextFrameFormat frameFormat() const;` //返回框架的框架格式
- 3、`QList<QTextFrame *> childFrames() const;` //返回该框架的子框架(可能为空)列表。
`QTextFrame *parentFrame() const;` //返回该框架的父框架，若父框架是根框架，则返回 0。
- 4、`QTextCursor firstCursorPosition() const;` //返回框架内的第一个光标位置。
`QTextCursor lastCursorPosition() const;` //返回框架内的最后一个光标位置。
- 5、`int firstPosition() const;` //返回框架内的第一个文档位置。
`int lastPosition() const;` //返回框架内的最后一个文档位置
- 6、`iterator begin() const;` //返回指向框架开头的迭代器。
`iterator end() const;`

`iterator` 是 `QTextFrame` 的内部内，可使用该类遍历框架内部的子对象，并读取其内容(不能修改)，可使用的成员函数如下

- 1)、`bool iterator::atEnd() const;` //若当前项是框架中的最后一项，同返回 `true`。
 - 2)、`QTextBlock iterator::currentBlock() const;`
返回迭代器指向的当前块，若迭代器指向的是一个子框架，则返回的块是无效的。
 - 3)、`QTextFrame iterator::currentFrame() const;`
返回迭代器指向的当前框架，若迭代器指向的是块，则返回 0。
 - 4)、`QTextFrame *iterator::parentFrame() const;` //返回当前框架的父框架
- 迭代器示例如下：

```

QTextCursor cur=pt->textCursor(); //获取文档的光标
.....
QTextFrame *pf=cur.insertFrame(...); //插入框架
.....
QTextFrame::iterator it=pf->begin(); //把指向框架开头的迭代器赋给迭代器 it
for ( ; !(it.atEnd()); ++it) { //遍历迭代器 it
    QTextBlock cf = it.currentBlock(); //获取当前迭代器所指向的文本块
    if(cf.isValid()) qDebug()<<cf.text(); } //若块有效，则输出该块的文本。

```

三、QTextFrameFormat 类中的函数

框架格式指定框架在屏幕上的渲染和定位方式，不直接指定其中的文本格式的行为，仅对其子级的布局提供约束。主要设置了边框的高度、宽度、页边距和内部边距

- 1、**QTextFrameFormat()**; 构造函数
- 2、bool **isValid()** const; //若框架有效则返回 true，否则返回 false。

框架边框

- 3、qreal **border()** const; //返回框架边框的宽度(像素)，默认为 0，要使边框可见，须设置值
void **setBorder**(qreal **width**)
- 4、QBrush **borderBrush()** const; //返回框架边框的画笔
void **setBorderBrush**(const QBrush &**brush**)
- 5、BorderStyle **borderStyle()** const; //返回框架边框的样式，BorderStyle 枚举见下表
void **setBorderStyle**(BorderStyle **style**)

QTextFrameFormat::BorderStyle 枚举

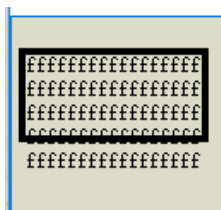
作用：描述框架的边框样式。

成员	值	说明
QTextFrameFormat::BorderStyle_None	0	无
QTextFrameFormat::BorderStyle_Dotted	1	点线
QTextFrameFormat::BorderStyle_Dashed	2	虚线
QTextFrameFormat::BorderStyle_Solid	3	实线
QTextFrameFormat::BorderStyle_Double	4	双线
QTextFrameFormat::BorderStyle_DotDash	5	点划线
QTextFrameFormat::BorderStyle_DotDotDash	6	双点划线
QTextFrameFormat::BorderStyle_Groove	7	槽
QTextFrameFormat::BorderStyle_Ridge	8	脊线
QTextFrameFormat::BorderStyle_Inset	9	内凹
QTextFrameFormat::BorderStyle_Outset	10	外凸

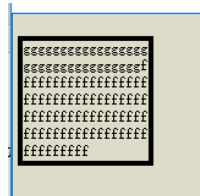
框架高度/宽度

- 6、QTextLength **height()** const; //返回框架的高度
void **setHeight**(const QTextLength &**height**); //QTextLength 类见 QTextTable 类的讲解
void **setHeight**(qreal **height**) //设置固定高度
- 7、void **setWidth**(const QTextLength &**width**); //设置框架的宽度(可设置可变宽度)，原理见下图
void **setWidth**(qreal **width**)

QTextLength **width**() const; //返回框架的宽度



固定高度, 由图可见, 固定高度的框架, 当内容超过其高度时, 不会自动扩展。



可变高度, 当内容超过其高度时, 会自动扩展。

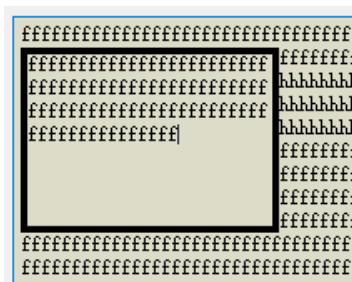
可变高度与固定高度

框架外部边距(效果见下图)

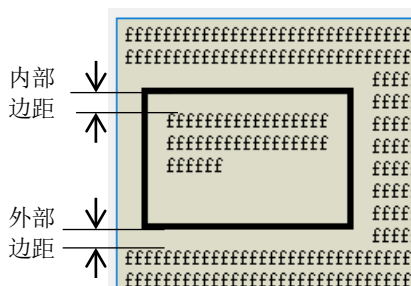
- 8、qreal **margin**() const; //返回框架的外部边距(像素)
void **setMargin**(qreal *margin*); //设置框架的外部边距(像素), 同时设置四个方向。
- 9、qreal **bottomMargin**() const; //返回框架底部的边距(像素)
void **setBottomMargin**(qreal *margin*); //设置框架底部的边距(像素)
qreal **topMargin**() const
void **setTopMargin**(qreal *margin*)
qreal **leftMargin**() const;
void **setLeftMargin**(qreal *margin*)
qreal **rightMargin**() const
void **setRightMargin**(qreal *margin*)

框架内部边距(效果见下图)

- 10、qreal **padding**() const; //返回框架的内部边距(像素)
void **setPadding**(qreal *width*)



未设置边距的效果



设置边距后的效果

分页策略和定位策略

- 11、PageBreakFlags **pageBreakPolicy**() const; //返回框架/表格的分页策略, 默认为 PageBreak_Auto
void **setPageBreakPolicy**(PageBreakFlags *policy*)

注意: 枚举 PageBreakFlag 是从父类继承来的, 见下表

QTextFormat::PageBreakFlag 枚举

标志: QTextFormat::PageBreakFlags

作用: 描述在打印时如何分页。映射到相应的 CSS 属性。

成员	值	说明
QTextFormat::PageBreak_Auto	0	由当前页上的可用空间自动确定
QTextFormat::PageBreak_AlwaysBefore	1	页面始终在段落/表格之前被分页
QTextFormat::PageBreak_AlwaysAfter	2	页面始终在段落/表格之后被分页

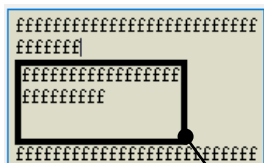
12、Position **position()** const; 返回框架相对于周围文本的定位策略。Position 枚举见下表

void **setPosition**(Position *policy*)

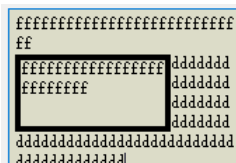
QTextFrameFormat::Position 枚举

作用: 描述框架相对于周围文本的位置。

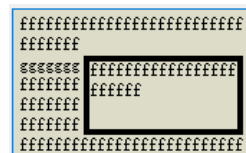
成员	值	说明
QTextFrameFormat::InFlow	0	原理见下图
QTextFrameFormat::FloatLeft	1	
QTextFrameFormat::FloatRight	2	



InFlow 框架的边框



FloatLeft



FloatRight

示例: 框架的使用

```
#include<QtWidgets>
int main(int argc, char *argv[]) {    QApplication app(argc, argv);
    QWidget w;
    QTextEdit *pt = new QTextEdit(&w); //创建文本编辑器
    QTextFrameFormat tff; QTextFrame *tf;    //创建框架和框架格式
    //1、获取当前光标,
    QTextCursor cursor= pt->textCursor(); //重点, 因为要使用光标把框架插入编辑器
    cursor.movePosition(QTextCursor::Start); //设置光标位置
    tff.setWidth(222);    //设置框架宽度
    //把框架高度设置为可变, 这样, 高度就会随输入的内容而变化了。
    tff.setHeight(QTextLength(QTextLength::VariableLength, 111));
    //tff.setBackground(QColor(111, 1, 1));
    tff.setBorder(4); //设置框架边框的宽度, 必须设置, 否则边框不可见。
    tff.setBorderBrush(QBrush(QColor(0, 0, 0))); //设置框架边框的颜色为黑色
    tff.setBorderStyle(QTextFrameFormat::BorderStyle_Solid); //设置框架边框为实线。
    tff.setPosition(QTextFrameFormat::FloatLeft); //设置框架的定位策略。
    //tff.setPageBreakPolicy(QTextFormat::PageBreak_Auto);
    tff.setPadding(11); //设置内部边距
    tff.setMargin(11); //设置外部边距
```

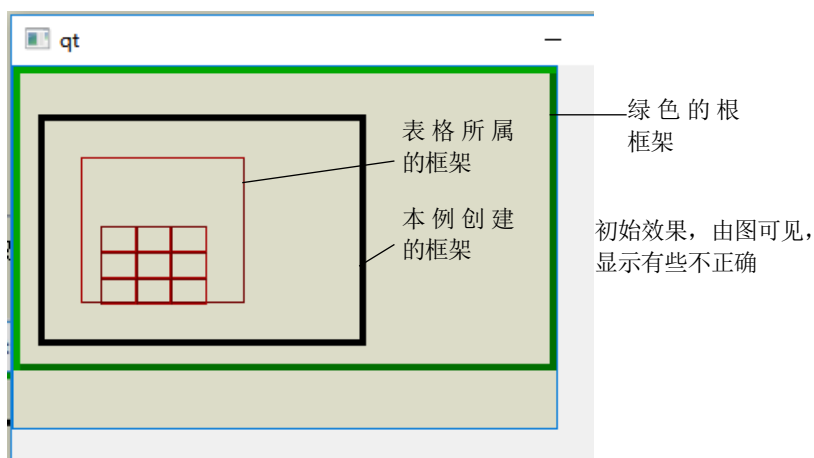
```

        tf=cursor.insertFrame(tff);    //插入框架到文档
//下面内容用于向框架中插入表格
    QTextTableFormat f; //表格格式
    f.setCellSpacing(0); //设置单元格的外部边距(即单元格之间的距离)为0
//以下函数是继承来的父类的函数
f.setWidth(111); //设置表格总宽度(这是固定宽度)，可变宽度使用 QTextLength 设置
f.setBorder(1); //设置表格边框线的宽度
f.setBorderBrush(QBrush(QColor(111,0,0))); //设置边框线的颜色为红色
f.setMargin(11); //设置整个表格的外部间距
f.setPosition(QTextFrameFormat::FloatLeft); //设置表格的位置策略
//以下代码用于验证表格是嵌套于框架之内的
f.setHeight(111); //把高度设置为大于表格的默认高度，便能明显看到表格所属的框架了
f.setPadding(11); //设置表格的内部间距(该距离是表格相对于其自身所属的框架的距离)

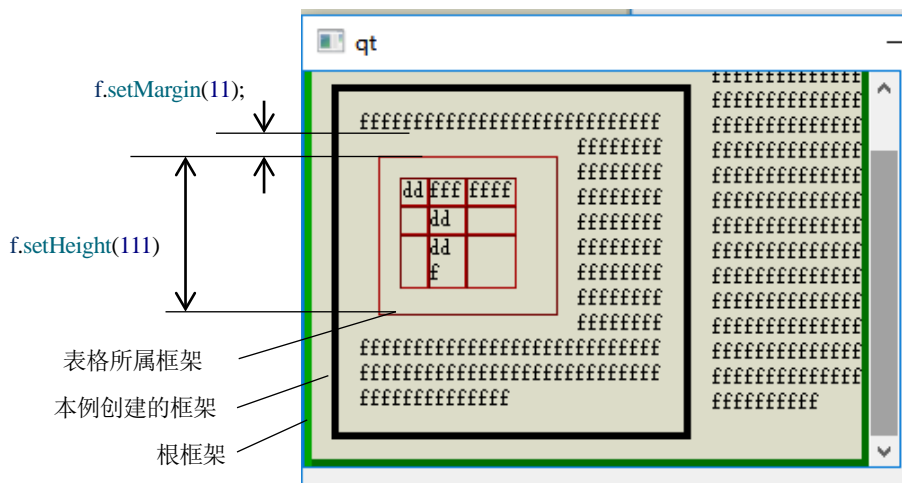
    cursor=tf->firstCursorPosition(); //获取在框架中的光标位置
    cursor.insertTable(3,3,f); //把表 格插入到框架中
//以下代码用于验证文档的根框架。
    QTextDocument *pd=pt->document(); //获取编辑器的内部文档
    QTextFrame *pm=pd->rootFrame(); //获取文档的根框架
    QTextFrameFormat ff; //设置框架格式以使根框架可见
ff.setBorder(4); //设置宽度
ff.setBorderBrush(QColor(1,111,1)); //背景色
pm->setFrameFormat(ff); //设置根框架的框架格式，这样便能显示根框架了
pt->resize(333,222); w.resize(444,333); w.show(); return app.exec(); }

```

初次显示的效果



输入一些内容后的效果



在表格中输入内容后，系统对表格在其自身框架中的位置作了调整，而且可看到使用 `f.setPadding(11);` 设置间距后表格相对于其自身框架的距离在各个方向并不一致

11.6 块(QTextBlock、QTextBlockFormat 类)

一、QTextBlock 基础

- 1、块的基本知识在第 1 节已讲了，不再重述，需要注意的，在编辑文档的过程中也会产生块。
- 2、因为块仅用于处理文本，因此，对块的格式设置包括对字符的格式设置(比如字体大小、粗细等)，还有块的格式设置，比如文本在块内的对齐方式、块的缩进、背景色等。
- 3、因为在文本编辑器中，编辑文字的行就是块，所以块与其他文档元素有些不同，块的格式修改和块的创建都需要通过 QTextCursor 的成员函数进行，因为其他类(包括 QTextBlock 类)都无法修改块的格式(即没有 setFormat()函数)，因此，即使我们获取到了 QTextBlock 对象，也只能对其进行只读访问，不能直接修改块的格式。
- 4、修改块的格式需使用以下函数

以下函数会清楚之前设置的格式

```
void QTextCursor::setBlockFormat(const QTextBlockFormat&); //设置块格式
void QTextCursor::setBlockCharFormat(const QTextCharFormat&); //设置块中字符的格式
```

以下函数不会清楚之前设置的格式

```
void QTextCursor::mergeBlockFormat(const QTextBlockFormat&); //合并块格式
void QTextCursor::mergeBlockCharFormat(const QTextCharFormat&); //合并块中字符的格式
```

注意：setBlockCharFormat()和 mergeBlockCharFormat()需在文本块为空的状态下，其设置的格式才会起作用。

- 5、块由 QTextCursor::insertBlock()函数插入文档，但要注意，该函数不会返回块的对象。
- 6、使用 QTextCursor::block();函数，可获取光标所在处的块。

示例代码：

```
QTextEdit *pt = new QTextEdit;
QTextCursor cur = pt->textCursor();
QTextBlockFrame f; //创建块格式
f.setBackground(QColor(111,1,1)); //设置背景色以使其插入的块与其他块有所区别
.....
cur.insertBlock(f); //使用块格式 f 向文档中插入一文本块
QTextBlock b=cur.block(); //获取光标处所在的块
qDebug()<<b.text(); //输出该块的文本。
```

二、QTextBlock 类中的函数

- 1、**QTextBlock**(const QTextBlock &other); //复制构造函数，块没有公有构造函数。

2、块的文本、方向、列表、文档

- 1)、QString **text**() const; //以纯文本形式返回块的内容。
- 2)、Qt::LayoutDirection **textDirection**() const; //返回块的文字方向(Qt::LeftToRight 或 Qt::RightToLeft)。
- 3)、QTextList ***textList**() const; //若块表示一个列表项，则返回该项所属的列表，否则返回 0。

4)、const QTextDocument *document() const; //返回该块所属的文档，若块不属于任何文档，则返回 0。

3、块的格式

5)、QTextBlockFormat blockFormat() const; //返回块的块格式。

6)、QTextCharFormat charFormat() const; //返回该块的字符格式。

7)、int blockFormatIndex() const; //返回文档内部块格式列表的索引

8)、int charFormatIndex() const; //返回文档内部字符格式列表的索引

9)、QVector<QTextLayout::FormatRange> textFormats() const; //返回块的文本格式选项。

4、块的位置

10)、bool contains(int position) const; //若位置 position 位于文本块内，则返回 true。

11)、QTextBlock next() const; //返回文档中此块的下一个块，若是最后一个块，则返回空块。

12)、QTextBlock previous() const; //返回文档中此块的前一个块，若是第一个块，则返回空块。

13)、int position() const; //返回文档中该块的第一个字符的索引。

14)、int blockNumber() const; //返回块的编号，若无效则返回-1。

5、块的行数、长度

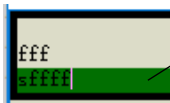
15)、int length() const; //以字符形式返回块的长度，返回的长度包含所有的格式化字符，比如换行符。

16)、int lineCount() const; //返回行数，注：并非所有布局都支持此功能。

void setLineCount(int count); //设置块的行数

17)、int firstLineNumber() const; //返回块的行号，若块无效，则返回-1，通常行号与块编号相同。

//setLineCount()设置了行数后对该函数有影响。



对于此行，各函数输出如下

position() = 5; //该块第一个字符 s，位于文档的第 5 个索引处(在之前有三个字符 f，第 1 块和第 2 块回车换行符)

blockNumber() = 2; //块的编号，该块是文档中的第 2 个块(从 0 开始编号)

length() = 6; //该块有 6 个字符(含末尾的空字符)

lineCount(); //该函数返回的值与 setLineCount()设置的值相同，若未设置，则返回 1。

firstLineNumber() = 2; //与块的编号相同。若 setLineCount()设置了值，则返回值就不是 2 了

6、可见性、有效性、版本

18)、bool isValid() const; //若块有效，则返回 true，否则返回 false。

19)、bool isVisible() const; //若块可见，则返回 true，否则返回 false

void setVisible(bool visible); //设置块是否可见

20)、int revision() const; //返回块的修订次数，每修改一次块，便会增加一次。

void setRevision(int rev); //设置块的修订次数，设置后修订次数不会再增加

7、存局及其他

21)、QTextLayout *layout() const; //返回该块的布局，布局用于显示和布局块的内容。

void clearLayout(); //清除用于布局块的布局。

22)、void setData(QTextBlockUserData *data); //把数据 data 附加到文本块(可用于自定义数据)。

QTextBlockUserData *userData() const

23)、void setUserState(int state); //在文本块中存储整数值 state，在语法高亮(QSyntaxHighlighter)中可用。

```
int userState() const;
```

8、迭代器(可用于访问块中的文本片段)

24)、iterator **begin**() const //迭代器, 返回指向文本块开头的迭代器

iterator **end**() const

iterator 是 QTextBlock 的内部, 可使用如下两个成员函数

25)、bool iterator::atEnd() const; //若当前项是文本块中的最后一项, 同返回 true。

26)、QTextFragment iterator::fragment() const; //返回迭代器当前指向的文本片段。

迭代器示例如下:

```
QTextCursor cur=pt->textCursor(); //获取光标
QTextBlock b=cur.block(); //获取光标所在处的块
QTextBlock::iterator it=b.begin(); //把指向文本块开头的迭代器赋给迭代器 it
for ( ; !(it.atEnd()); ++it) { //遍历迭代器 it
    QTextFragment cf = it.fragment(); //获取当前所指向的文本片段
    qDebug()<<cf.text(); } //输出文本片段的文本。
```



该文本块, 使用以上代码将输出三个字符串, 分别是"ff d"、"dd"、 "dd"

9、以下为重新实现的操作符函数

```
bool operator!=(const QTextBlock &other) const
```

```
bool operator<(const QTextBlock &other) const
```

```
QTextBlock &operator=(const QTextBlock &other)
```

```
bool operator==(const QTextBlock &other) const
```

三、QTextBlockFormat 类中的函数

该类主要设置文本对齐方式、边距、文本方向、缩进等, 背景色需使用父类的 setBackground() 设置。

1、**QTextBlockFormat**(); //构造函数

对齐方式、断行

2、Qt::Alignment **alignment**() const; //返回块中文本的对齐方式

void **setAlignment**(Qt::Alignment **alignment**); //设置块中文本的对齐方式

3、bool **nonBreakableLines**() const; //若块中的行不可断行, 则返回 true。

void **setNonBreakableLines**(bool **b**); //若 b 为 true, 则不可断行(即不会自动换行)。

边距

4、qreal **topMargin**() const; //返回块的顶部边距

qreal **bottomMargin**() const; //返回块的底部边距

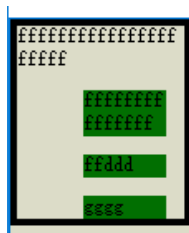
qreal **leftMargin**() const; //返回块左侧的边距

qreal **rightMargin**() const; //返回块右侧的边距

void **setTopMargin**(qreal **margin**); //设置块的顶部边距

void **setBottomMargin**(qreal **margin**)

void **setLeftMargin**(qreal **margin**)



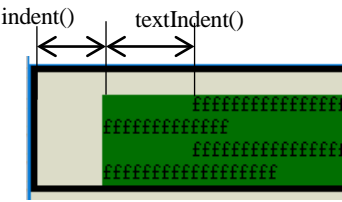
边距效果

绿色背景
的文本块
为设置了
边距的块

void **setRightMargin**(qreal *margin*)

缩进量(原理见图示)

- 5、int **indent**() const; //返回块的缩进
- void **setIndent**(int *in*); //设置块的缩进，缩进量是 QTextDocument::indentWidth 的倍数。
- 6、void **setTextIndent**(qreal *indent*); 设置块中文本第一行的缩进量(像素单位)
- qreal **textIndent**() const; //返回块中文本第一行的缩进量。



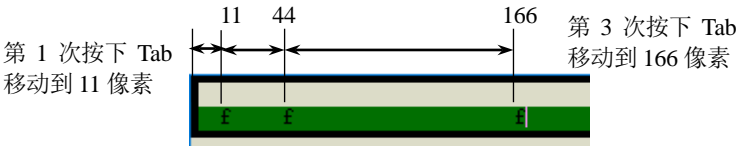
缩进量效果

制表符

- 7、QList<QTextOption::Tab> **tabPositions**() const; //返回文本块定义的制表符位置列表。
- void **setTabPositions**(const QList<QTextOption::Tab> &*tabs*); //设置块中的制表符位置。

下面举一个简单的示例(效果见图示):

```
QTextOption::Tab tt1;    tt1.position=11;    //第 1 个制表符位于像素 11
QTextOption::Tab tt2;    tt1.position=44;    //第 2 个制表符位于像素 44
QTextOption::Tab tt3;    tt1.position=166;   //第 3 个制表符位于像素 166
QList<QTextOption::Tab> s; s.append(tt1); s.append(tt2); s.append(tt3);
QTextBlockFrame bf; bf.setTabPositions (s); bf.setBackground(QColor(1,111,1));
cursor.insertBlock(bf);
```



制表符效果

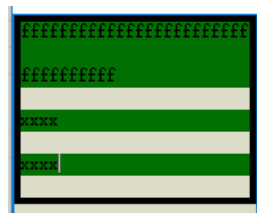
行间距(效果见后面的图示)

- 8、void **setLineHeight**(qreal *height*, int *heightType*)
- 根据枚举 LineHeightTypes(即参数 heightType 的值)所描述的方式，设置块的行间距为 height。枚举 LineHeightTypes 见下表

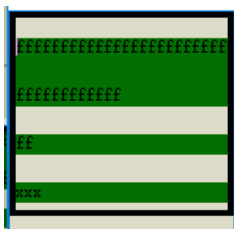
QTextBlockFormat::LineHeightTypes 枚举(无标志)		
描述行间距		
成员	值	说明
QTextBlockFormat::SingleHeight	0	默认行间距
QTextBlockFormat::ProportionalHeight	1	与行按比例设置行间距，比如 setLineHeight(200,1);表示 2 倍行间距
QTextBlockFormat::FixedHeight	2	设置固定像素的行间距

QTextBlockFormat::MinimumHeight	3	设置最小行间距(像素)
QTextBlockFormat::LineDistanceHeight	4	在行之间添加指定像素高度的行间距

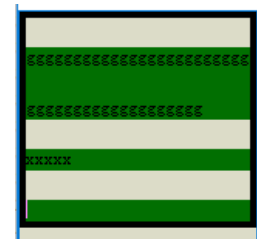
- 9、int **lineHeightType**() const//返回块的 QTextFrame::LineHeightType 属性(即 LineHeightTypes 枚举值)
- 10、qreal **lineHeight**() const; //返回块的行间距
- 11、qreal **lineHeight**(qreal *scriptLineHeight*, qreal *scaling*) const
- 把行行间距与 scriptLineHeight 和 scaling 进行计算后，返回计算后的行间距，具体计算规则略。



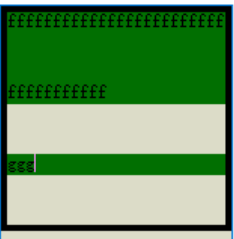
setLineHeight(200,1);
双倍行间距，空白间距添加于行的底部



setLineHeight(33,2);
固定高度，注意顶部和底部都有一些空白



setLineHeight(33,3);
最小行间距，空白间距添加于行的顶部



setLineHeight(33,4);
在行与行之间指定高度，空白间距添加于行的底部

行间距效果

有效性及分页策略

- 12、bool **isValid**() const; //若块格式有效，则返回 true，否则返回 false
- 13、PageBreakFlags **pageBreakPolicy**() const; //返回该块的分页策略，默认为 QTextFormat::PageBreak_Auto
- 14、void **setPageBreakPolicy**(PageBreakFlags *policy*); //设置该块的分页策略，

11.7 列表(QTextList、QTextListFormat 类)

一、QTextList 基础

- 1、列表包含一系列的文本块，每个列表前都有一个编号，编号可以是数字、字母、符号等，具体由 QTextListFormat 类设置。若编号是数字或有序字母，编号可以自动编号。
- 2、因为 QTextList 是 QTextBlockGroup 的子类，所以列表并不把其项目分组为子元素，而仅仅是提供了管理这些项目的功能。因此，在访问文档中的文本块时，该文本块有可能是一个列表的项目，也有可能不是。
- 3、有两种方法创建文本块，如下：

```
QTextList *insertList(const QTextListFormat &format)
```

```
QTextList *insertList(QTextListFormat::Style style)
```

插入一个空的文本块，并使该块成为列表中的第一项。

```
QTextList *createList(const QTextListFormat &format)
```

```
QTextList *createList(QTextListFormat::Style style)
```

表示创建一个列表，并把光标所在的当前块设置为列表中的第一项。

- 4、使用 QTextList::add()函数可向现有列表中添加文本块。

二、QTextList 类中的函数

- 1、QTextListFormat format() const; //返回列表的列表格式
void setFormat(const QTextListFormat &format); //设置列表的列表格式
- 2、void add(const QTextBlock &block); //把块 block 添加到列表
- 3、int count() const; //返回列表中的项目数量
- 4、QTextBlock item(int i) const; //返回列表中第 i 个文本块
- 5、QString itemText(const QTextBlock &block) const; //返回与块 block 对应的列表项的编号文本。
- 6、void remove(const QTextBlock &block); //把块 block 从列表中删除。
- 7、int itemNumber(const QTextBlock &block) const;
返回与块 block 对应的列表项的索引(从 0 开始)，若块不在列表中，则返回-1。索引是指块位于列表中的第几个位置，这是与编号不同的，比如块< 2 >，他的编号是"< 2 >"，而索引是 1(即第 1 项)，即编号是从 1 开始的，索引是从 0 开始的，而且编号包括前后缀，而索引就是一个数值。
- 8、void removeItem(int i);
删除列表中位置为 i 的项目，当列表中的最后一项被删除时，该列表会被拥有它的 QTextDocument 自动删除。

三、QTextListFormat 类中的函数

- 1、QTextListFormat(); //构造函数
- 2、int indent() const; //返回列表的缩进量，
void setIndent(int in); //设置列表的缩进量。缩进量为 in 乘以 QTextDocument::indentWidth 属性的值

- 3、bool **isValid()** const; //若此列表有效，则返回 true，否则返回 false。
- 4、void **setNumberPrefix**(const QString &*numPrefix*); //设置列表项目编号的前缀(对不能排序的编号无影响)。
void **setNumberSuffix**(const QString &*numSuffix*); //设置列表项目编号的后缀(对不能排序的编号无影响)。
QString **numberPrefix**() const; //返回列表项目编号的前缀。
QString **numberSuffix**() const; //返回列表项目编号的后缀
- 5、Style **style**() const; //返回列表编号的样式
void **setStyle**(Style *style*); //设置列表编号的样式，Style 枚举见下表

QTextListFormat::Style 枚举(无标志)

描述项目编号的样式

成员	值	说明
QTextListFormat::ListDisc	-1	实心圆圈
QTextListFormat::ListCircle	-2	空心圆圈
QTextListFormat::ListSquare	-3	正方形
QTextListFormat::ListDecimal	-4	十进制数值
QTextListFormat::ListLowerAlpha	-5	小写字母
QTextListFormat::ListUpperAlpha	-6	大写字母
QTextListFormat::ListLowerRoman	-7	小写罗马数字
QTextListFormat::ListUpperRoman	-8	大写罗马数字

示例：QTextList 列表的使用

//m.h 文件的内容

```

#ifndef M_H
#define M_H
#include<QtWidgets>
class B:public QWidget{    Q_OBJECT
public:    QPushButton *pb1,*pb2;    QTextEdit *pt;                QTextList *px1,*px2;
    B(QWidget *p=0):QWidget(p){
        pt=new QTextEdit(this);
        QTextCursor cur= pt->textCursor(); //获取光标(插入符)
        cur.movePosition(QTextCursor::Start); //设置光标位置
        QTextListFormat f;
        f.setStyle(QTextListFormat::ListDecimal);                //设置列表编号样式
        f.setNumberPrefix("<"); f.setNumberSuffix(">");            //设置列表项目编号前/后缀
        px1=cur.insertList(f);                //创建列表 px1

        QTextListFormat f1;    f1.setStyle(QTextListFormat::ListDecimal);
        px2=cur.insertList(f1);                //创建列表 px2
    } //把文本块添加到列表 px1 中
    QTextBlockFormat bf;    //设置块格式
    bf.setBackground(QColor(111,1,1));
    cur.insertBlock(bf);    //创建块
    QTextBlock b1; //注意：这样直接创建的块不能被直接添加到列表中(因为该块未插入文档)。
    b1=cur.block(); //获取文档中上面创建的块，并赋值给 b1
    px1->add(b1);    //把 b1 添加到列表中。
} //显示根边框
QTextDocument *pd=pt->document(); QTextFrame *pf=pd->rootFrame(); //获取根框架

```

```

    QTextFrameFormat ff;
    ff.setBorder(4);          ff.setBorderBrush(QColor(0,0,0)); //设置框架边框宽度及颜色
    ff.setBorderStyle(QTextFrameFormat::BorderStyle_Solid);    //框架边框线设置为实现
    pf->setFrameFormat(ff);   //设置边框格式
    pt->resize(333,222);

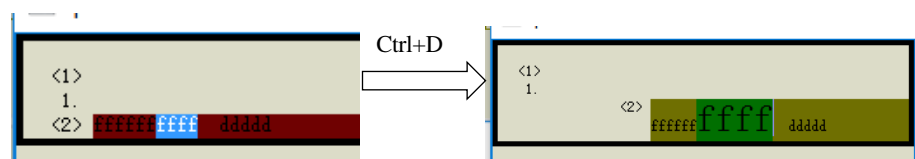
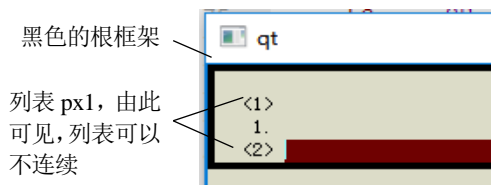
//设置按钮及快捷键，读者还可设置更多的快捷键，以更方便的处理文本。
    pb1=new QPushButton("Blod",this);   pb1->move(22,244);
    pb2=new QPushButton("FontSize22",this); pb2->move(99,244);
    pb1->setShortcut(QKeySequence("Ctrl+F")); //设置 pb1 的快捷键
    pb2->setShortcut(QKeySequence("Ctrl+D"));
    QObject::connect(pb1,&QPushButton::clicked,this,&B::f1);
    QObject::connect(pb2,&QPushButton::clicked,this,&B::f2); }

public slots:
    void f1() {
        QTextCursor cur=pt->textCursor();
        QTextBlock b1=cur.block();          //获取当前光标处的块
        if(px1->itemNumber(b1)!=-1){          //判断块 b1 是否是列表中的项目
            qDebug() <<b1.text();             //输出块的文本
            qDebug() <<px1->itemText(b1);     //输出块的项目编号文本
            qDebug() <<px1->itemNumber(b1);   //输出块的项目编号
        }
    }
    void f2() {qDebug() <<"BBB";
        QTextCursor cur= pt->textCursor();   QTextBlockFormat bf;   QTextCharFormat cf;
        bf.setBackground(QColor(111,111,1)); bf.setIndent(2);
        cf.setFontPointSize(22);              cf.setBackground(QColor(1,111,1));
        cur.setCharFormat(cf);
        cur.mergeBlockFormat(bf); //合并块的格式
        //cur.setBlockFormat(bf); //注意：若使用此函数设置块的格式，会把列表的项目删除
    } };
#endif // M_H

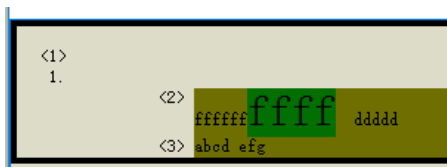
//m.cpp 文件的内容
#include "m.h"
int main(int argc, char *argv[]){   QApplication app(argc,argv);
    B w;    w.resize(444,333);    w.show();    return app.exec(); }

```

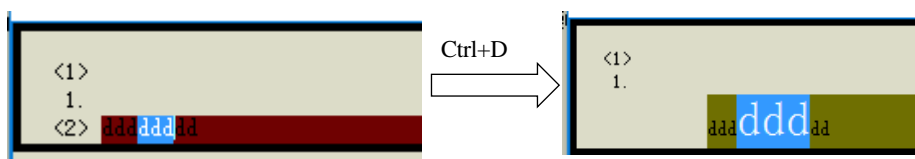
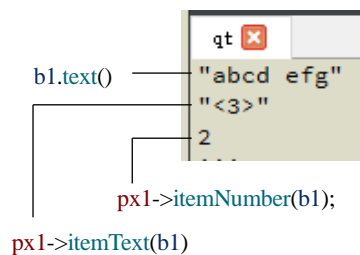
运行结果及说明



输入并选择图示内容，然后按下 Ctrl+D



把光标移至列表项目<2>的末尾，并按下回车，可以看到列表的编号会自动编号，在项目<3>的块中，输入图示内容，然后按下 Ctrl+F，结果输出，如右图所示



注释掉示例中的语句：cur.mergeBlockFormat(bf);

并把语句 cur.setBlockFormat(bf);前的注释删掉

然后重新运行程序，并输入且选中图中内容，然后按下 Ctrl+D，结果如右图，从图中可见，列表 px1 的项目<2>被删掉了。

11.8 图像(QTextImageFormat 类)和

文本片段(QTextFragment 类)

一、图像基础

- 1、图像与文档中的其他文档元素不同，图像由特殊格式的文本片段表示。这使得图像可以与周围的文本排成一行。因此没有与图像对象相对应的 QTextImage 类，只有描述图像格式的 QTextImageFormat 类。
- 2、图像的来自于外部的图片文件(jpg、png)等资源，因此使用图像时只需指定图像文件的路径或资源名称即可
- 3、可使用 QTextCursor::insertImage()函数插入图像，其原型如下：

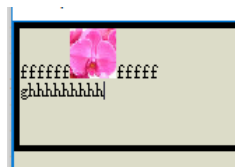
```
void insertImage(const QTextImageFormat &format)
void insertImage(const QTextImageFormat &format, QTextImageFormat::Position alignment)
void insertImage(const QString &name); //该函数可直接使用文件路径或资源名插入图片。
void insertImage(const QImage &image, const QString &name = QString())
```

- 4、注意：在 QTextCursor 类中没有直接获取图像或图像格式的函数，要获取图像格式可使用 QTextFormat::toImageFormat()函数间接获取，方法如下：

```
QTextCursor cur = pt->textCursor();
QTextImageFormat f = cur.charFormat().toImageFormat(); //获取光标的字符格式，再转换为图像格式
qDebug()<< f.name(); //输出图像的名称。
```

- 4、下面为示例代码(结果见右图)

```
QTextEdit *pt = new QTextEdit;
QTextCursor cur = pt->textCursor();
QTextImageFormat f; //创建图像格式。
f.setHeight(33); //设置显示图像时的高度
f.setWidth(33); //设置显示图像时的宽度
f.setName("F:/1i.png"); //设置需要显示的图像的位置，这里直接指定路径名称，也可使用资源名称
.....
cur.insertImage(f);
QTextImageFormat f1 = cur.charFormat().toImageFormat(); //获取光标处的图像格式
```



周围的文字是插入图像后输入的，由此可见，图像是被当作文本片段处理的

二、QTextImageFormat 类中的函数

- 1、QTextImageFormat(); //构造函数
- 2、qreal height() const; //返回图像占据的高度
qreal width() const; //返回图像占据的宽度
void setHeight(qreal height); //设置图像占据的高度
void setWidth(qreal width); //设置图像占据的宽度
- 3、QString name() const; //返回图像的名称，该名称是程序资源名称或文件路径字符串
void setName(const QString &name); //设置图像的名称(即图像的路径或资源名)
- 4、bool isValid() const; //若图像格式有效，则返回 true，否则返回 false

三、QTextFragment 类中的函数

1、QTextFragment 类仅描述了文本片段的一些信息(比如长度、位置、文本等)，因此不需要修改文本片段。

2、文本片段需要通过 QTextBlock::iterator 类中的 fragment()函数来获取，步骤如下：

```
QTextCursor cur=pt->textCursor(); //获取光标
QTextBlock b=cur.block();          //获取光标所在处的块
QTextBlock::iterator it=b.begin(); //把指向文本块开头的迭代器赋给迭代器 it
QTextFragment cf = it.fragment();  //获取当前所指向的文本片段
QDebug() <<cf.text();              //输出文本片段的文本。
```

3、QTextFragment(); //构造函数

QTextFragment(const QTextFragment &other);

4、QString text() const; //以纯文本形式返回文本片段包含的文本。

5、int length() const; //返回文本片段中的字符数

6、int position() const; //返回文档中此文本片段的位置。

7、bool isValid() const; //若文本片段有效，则返回 true，否则返回 false。

8、bool contains(int position) const; //若在文档中的位置 position 处文本片段包含文本则返回 true。

9、QTextCharFormat charFormat() const; //获取该文本片段的字符格式

10、int charFormatIndex() const; //返回该文本片段的字符格式在文档内部字符格式列表中的索引。

11、QList<QGlyphRun> glyphRuns(int pos = -1, int len = -1) const; //返回文本片段的字形。

12、以下为重新实现的操作符函数

bool operator!=(const QTextFragment &other) const

bool operator<(const QTextFragment &other) const

QTextFragment &operator=(const QTextFragment &other)

bool operator==(const QTextFragment &other) const

11.9 插入自定义文档对象(文档元素)与总结

一、插入自定义文档对象

- 1、前面章节在 QTextEdit 中显示的文档元素都是常见的文档元素，本小节将简单的介绍怎样使用 QTextEdit 显示自定义的文档元素。
- 2、基本思想：使用 QTextCharFormat 把自定义的文档元素以文本片段的形式插入到 QTextDocument 中。而自定义的文档元素，可以子类化 QTextObjectInterface 类，并在该类中使用 QPainter 自由绘制你想显示的任何内容。
- 3、对象类型与格式类型：

每一种文档元素在 QTextFormat(所有格式类型的顶级父类)中，都定义了一个相对应的对象类型，每种文档格式也定义了一个相对应的格式类型，每种对象类型与一种格式类型相关联。对象类型使用 QTextFormat::ObjectTypes 枚举描述，使用 QTextFormat::setObjectType(int)进行设置，格式类型使用 QTextFormat::FormatType 枚举描述，下面为这些枚举的原型

```
enum QTextFormat::ObjectTypes{NoObject, ImageObject, TableObject, ... , UserObject};
```

```
enum QTextFormat::FormatType{InvalidFormat, BlockFormat, CharFormat, ... , UserFormat};
```

- 3、插入自定义文档对象，分为以下几步

- ①、创建自己的文档对象，即子类化 QTextObjectInterface 类绘制自定义文档对象

```
class C:public QObject, public QTextObjectInterface { ....}; //具体绘制方法见示例代码
```

- ②、使用如下函数在当前文档使用的文档布局上注册绘制的文本对象

```
void QAbstractTextDocumentLayout::registerHandler(int objectType, QObject * component)
```

以上函数表示把组件 component 注册为对象类型 objectType。

使用示例如下：

```
C *pc = new C; //创建由第一步实现的对象。
```

```
//把 pc 注册为用户自定义对象类型，类型的值必须大于等于 UserObject
```

```
pt->document()->documentLayout()->registerHandler(QTextFormat::UserObject + 1,pc);
```

- ③、把对象类型使用 QTextFormat::setObjectType()设置在 QTextCharFormat 上。方法如下：

```
QTextCharFormat cf;
```

```
cf.setObjectType(QTextFormat::UserObject + 1); //即，与格式 cf 相关联的对象类型是 UserObject+1
```

- ④、使用以上设置了对象类型的 QTextCharFormat 格式类型把 QChar::ObjectReplacementCharacter 插入到文档中。QChar::ObjectReplacementCharacter 用于表示对象(比如图像等)，方法如下：

```
cur.insertText(QString(QChar::ObjectReplacementCharacter), cf); //把自定义对象类型插入文档
```

- ⑤、第 3 步和第 4 步也可使用如下方法代替(更简洁)

```
QTextImageFormat mf; //注意：QTextImageFormat 是 QTextCharFormat 的子类
```

```
mf.setObjectType(QTextFormat::UserObject + 1);
```

```
cur.insertImage(mf);
```

- 4、QTextObjectInterface 类的成员函数(该类没有公有构造函数)

①、virtual void **drawObject**(QPainter *painter, const QRectF &rect, QTextDocument *doc,

int posInDocument, const QTextFormat &format) = 0

绘制自定义的文档对象，painter 表示绘制器，rect 表示把图形绘制于该矩形内，该参数是由以下的函数 intrinsicSize()返回的值，doc 表示在该文档上绘制，posInDocument 表示文档中的位置(以字符数表示)，format 表示绘制时的格式。若不需要在文档中指定位置以指定格式绘制对象，则后面三个参数可以不使用。

②、virtual QSizeF **intrinsicSize**(QTextDocument *doc, int posInDocument, const QTextFormat &format) = 0

该函数的返回值，决定了绘制的图像的大小。

示例：插入自定义文档对象

//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>
class C : public QObject, public QTextObjectInterface { //继承的第一个类必须是 QObject
    Q_OBJECT
    Q_INTERFACES(QTextObjectInterface) //必须使用该宏让Qt知道该类实现了QTextObjectInterface。
public:
    QSizeF intrinsicSize(QTextDocument *doc, int pos, const QTextFormat &format)
    {
        return QSizeF(88, 88); } //这是绘制的图像的大小，本例直接返回一个大小即可。

void drawObject(QPainter *painter, const QRectF &rect, QTextDocument *doc,
                int posInDocument, const QTextFormat &format)
{
    QImage im("F:/li.png"); //需要绘制的图像
    painter->drawImage(rect, im); //绘制图像
    //再随意绘制一系列线条(你也可以随便绘制想绘制的图形)
    //注意：坐标点需由 rect 来指定，否则绘制的图形将不会移动。
    painter->drawLine(rect.topLeft()+QPoint(11, 0), rect.topRight());
    painter->drawLine(rect.topLeft()+QPoint(11, 11), rect.topRight());
    painter->drawLine(rect.topLeft()+QPoint(11, 22), rect.topRight());
    painter->drawLine(rect.topLeft()+QPoint(11, 33), rect.topRight());
    painter->drawLine(rect.topLeft()+QPoint(11, 44), rect.topRight());
    painter->drawLine(rect.topLeft()+QPoint(11, 55), rect.topRight());
}

};

class B:public QWidget{    Q_OBJECT
public:
    B(QWidget *p=0):QWidget(p){
        QTextEdit *pt=new QTextEdit(this);
        QTextCursor cur= pt->textCursor(); //获取光标
        cur.movePosition(QTextCursor::Start); //设置光标位置

        C *pmc=new C; //创建自定义的文档对象
        pmc->setParent(this); //设置父对象，以便能正确销毁。
        //注册自定义文档对象 pmc 的类型，类型的值必须大于或等于 UserObject
        pt->document()->documentLayout()->registerHandler(QTextFormat::UserObject+1, pmc);

        QTextCharFormat cf; //创建字符格式
```

```

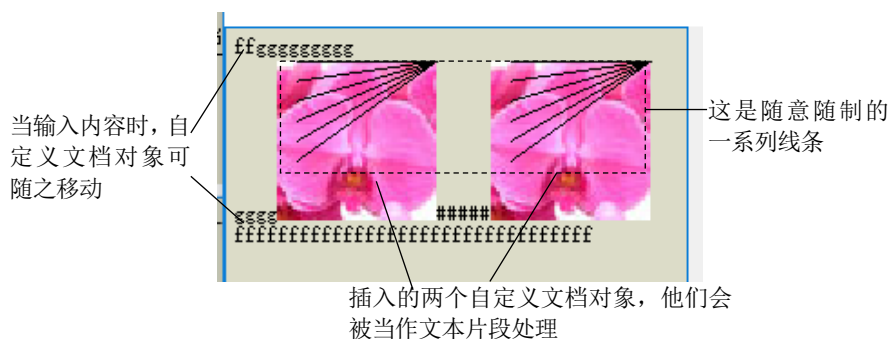
//设置格式 QTextCharFormat 的对象类型为用户自定义对象类型,
//即格式 cf 相关联的对象类型是(UserObject+1);
    cf.setObjectType(QTextFormat::UserObject + 1);
//把 QChar::ObjectReplacementCharacter 使用格式 cf 插入到文档中。
    cur.insertText(QString(QChar::ObjectReplacementCharacter), cf);

    cur.insertText("#####"); //插入一些文字
//以下代码的原理同 QTextCharFormat, 只是更简洁
    QTextImageFormat mf;
    mf.setObjectType(QTextFormat::UserObject + 1);
    cur.insertImage(mf);
} };
#endif // M_H

//m.cpp 文件的内容
#include "m.h"
int main(int argc, char *argv[]) {    QApplication app(argc, argv);
    B w;    w.resize(444, 333);    w.show();    return app.exec(); }

```

运行结果及说明



二、总结

下面把各种文档对象和格式对象的接口函数总结出来,。

文档对象和格式对象接口函数总结

以下使用的 cur 为文档的 QTextCursor 对象

表格对象	单元格长度	QTextTableFormat f; f.setColumnWidthConstraints(...)
	插入和获取表格对象	QTextTable *pt1=cur.insertTable(...);
	设置和获取表格格式	QTextTable::setFormat(...); QTextTable::format(...);
	获取单元格对象	QTextTableCell ptc=pt1->cellAll(...);
	设置/获取单元格格式	QTextTableCell::setFormat(...); QTextTableCell::format(...);
框架对象	插入/获取框架对象	QTextFrame *pt2=cur.insertFrame(...);
	设置/获取框架格式	QTextFrame::setFormat(...); QTextFrame::format(...);
块对象	插入块对象	cur.insertBlock(...); //注意: 未返回块对象
	获取块对象	QTextBlock b=cur.block(); //仅限于只读访问
	获取块格式和字符格	QTextBlock::blockFormat();

	式(只读访问)	QTextBlock::charFormat();
	获取/修改块格式和字符格式(可修改)	QTextCursor::blockFormat(); QTextCursor::blockCharFormat(); QTextCursor::setBlockFormat(); QTextCursor::setBlockCharFormat(); QTextCursor::mergeBlockFormat();QTextCursor::mergeBlockCharFormat();
列表对象	插入/获取列表对象	QTextList *pt3=cur.insertList(...); QTextList *pt4=cur.createList(...);
	向列表中添加块	QTextList::add(const QTextBlock&);
	设置/获取列表格式	QTextList::setFormat(...); QTextList::format(...);
	获取列表关联的块	QTextList::item(int);
图像对象	插入图像对象(没有获取图像对象的函数)	cur.insertImage(...); //注意无返回对象
	获取图像格式	QTextImageFormat f=cur.charFormat().toImageFormat();
文本片段	获取文本片段的方法 (没有文本片段格式)	QTextBlock::iterator it=cur.block().begin();
		QTextFragment cf=it.fragment();

11.10 QTextCharFormat 类及其他类

一、QTextFormat 和 QTextObject 类简介

1、QTextFormat 类

①、QTextFormat 类是前面章节介绍的所有格式类的父类，通常不需要使用该类来设置各文档对象的格式，而应使用该类的子类以设置特定文档元素的格式。因此，本文不会详细讲解该类。

②、QTextFormat 类定义了大量的属性，除了使用该类的子类来设置各文档元素的格式外，还可设置该类的属性来间接设置文档元素的格式，使用的函数原型如下

```
void setProperty(int propertyId, const QVariant &value);
```

propertyId 表示需要设置的属性，value 表示设置的值。

```
QVariant property(int propertyId) const; //获取属性 propertyId 的值。
```

示例如下：

```
QTextCursor cur= pt->textCursor(); //获取光标
cur.movePosition(QTextCursor::Start); //设置光标位置
QTextBlockFormat f;
//使用属性设置背景色。
f.setProperty(QTextFormat::BackgroundBrush, QBrush(QColor(111, 1, 1)));
cur.insertBlock(f);
```

2、QTextObject 类

该类用于描述文档对象，是 QTextFrame 和 QTextBlockGroup 类的父类，该类很少需要直接使用。该类只有少数的几个成员函数，如下所示

1)、QTextObject(QTextDocument *document); //受保护的

~QTextObject(); //受保护的

构造函数和析构函数都是受保护的，构造函数只能从 QTextDocument::createObject() 调用，以创建一个文档对象。

2)、void setFormat(const QTextFormat &format); //设置文本对象的格式。受保护的

QTextFormat format() const; //返回该文本对象的格式

3)、QTextDocument *document() const; //返回该文本对象所关联的文档

4)、int formatIndex() const; //返回文档内部格式列表中对象格式的索引

5)、int objectIndex() const; //返回该文本对象的对象索引，可与 QTextFormat::setObjectIndex() 一起使用。

二、QTextCharFormat 类

1、QTextCharFormat 类用于描述字符的格式信息(比如粗体、斜体等)，该类的绝大部分属性与 QFont 类是相同的，具体的细节详见 QFont 类的讲解。

2、字符的颜色使用 QTextFormat::setForeground() 设置，
字符背景色使用 QTextFormat::setBackground() 设置

3、QTextCharFormat 类中的函数

下表的函数比较简单，直接以表的形式列出

QTextCharFormat 类中的函数		
QString fontFamily() const	void setFontFamily (const QString & <i>family</i>)	字体族
bool fontFixedPitch() const	void setFontFixedPitch (bool <i>fixedPitch</i>)	是否使用等宽字体
bool fontItalic() const	void setFontItalic (bool <i>italic</i>)	斜体
bool fontKerning() const	void setFontKerning (bool <i>enable</i>)	字距调整
qreal fontLetterSpacing() const	void setFontLetterSpacing (qreal <i>spacing</i>)	字母之间的间距
bool fontOverline() const	void setFontOverline (bool <i>overline</i>)	上划线
qreal fontPointSize() const	void setFontPointSize (qreal <i>size</i>)	字体大小
int fontStretch() const	void setFontStretch (int <i>factor</i>)	拉伸因子
bool fontStrikeOut() const	void setFontStrikeOut (bool <i>strikeOut</i>)	删除线
bool fontUnderline() const	void setFontUnderline (bool <i>underline</i>)	下划线
int fontWeight() const	void setFontWeight (int <i>weight</i>)	字体重量(即粗细)
qreal fontWordSpacing() const	void setFontWordSpacing (qreal <i>spacing</i>)	两单词间的间距
QString toolTip() const	void setToolTip (const QString & <i>text</i>)	文本片段的工具提示
QColor underlineColor() const	void setUnderlineColor (const QColor & <i>col</i>)	下划线颜色

- 1)、**QTextCharFormat()**; //构造函数
- 2)、bool **isValid()** const; //若字符格式有效，则返回 true，否则返回 false。
- 3)、QFont::SpacingType **fontLetterSpacingType()** const; //qt5.0
void **setFontLetterSpacingType**(QFont::SpacingType *letterSpacingType*)
字母间距的类型，即字母间距是以百分比设置还是以像素设置。
- 4)、QFont::Capitalization **fontCapitalization()** const
void **setFontCapitalization**(QFont::Capitalization *capitalization*)
设置文本字体的大写模式(比如首字母大写，或全部大写等)，枚举取值见 QFont 类。
- 5)、QFont::StyleHint **fontStyleHint()** const; //字体样式提示(即优先匹配哪一个字体样式)
void **setFontStyleHint**(QFont::StyleHint *hint*, QFont::StyleStrategy *strategy* = QFont::PreferDefault)
- 6)、QFont::StyleStrategy **fontStyleStrategy()** const; //样式提示策略，比如位图字体、轮廓字体等
void **setFontStyleStrategy**(QFont::StyleStrategy *strategy*)
- 7)、QFont::HintingPreference **fontHintingPreference()** const
void **setFontHintingPreference**(QFont::HintingPreference *hintingPreference*)
字体提示首选项，主要用于选择优化字体方式。枚举值见 QFont 类。
- 8)、void **setVerticalAlignment**(VerticalAlignment *alignment*)
VerticalAlignment **verticalAlignment()** const
设置字符的垂直对齐方式，枚举 VerticalAlignment 见下表

QTextCharFormat::VerticalAlignment 枚举(无标志)

描述字符的垂直对齐方式

成员	值	说明
QTextCharFormat::AlignNormal	0	以标准方式对齐
QTextCharFormat::AlignSuperScript	1	字符位于基线下方
QTextCharFormat::AlignSubScript	2	字符位于基线上方
QTextCharFormat::AlignMiddle	3	对象中心与基线垂直对齐(仅适用于嵌入对象)

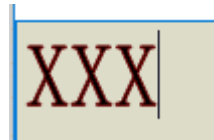
QTextCharFormat::AlignBottom	4	对象的底部与底线垂直对齐
QTextCharFormat::AlignTop	5	对象的顶部与基线垂直对齐
QTextCharFormat::AlignBaseline	6	字符基线对齐

- 9)、void **setUnderlineStyle**(UnderlineStyle style)
UnderlineStyle **underlineStyle**() const
下划线样式，UnderlineStyle 枚举见下表

QTextCharFormat::UnderlineStyle 枚举(无标志)		
描述下划线样式		
成员	值	说明
QTextCharFormat::NoUnderline	0	无下划线
QTextCharFormat::SingleUnderline	1	实线
QTextCharFormat::DashUnderline	2	虚线
QTextCharFormat::DotLine	3	点线
QTextCharFormat::DashDotLine	4	点划线
QTextCharFormat::DashDotDotLine	5	双点划线
QTextCharFormat::WaveUnderline	6	波浪线
QTextCharFormat::SpellCheckUnderline	7	取决于 QPlatformTheme 的 SpellCheckUnderlineStyle

- 10)、QPen **textOutline**() const
void **setTextOutline**(const QPen &pen)
设置绘制字符轮廓线的画笔。示例如下(效果如右图)

```
QTextCursor cur= pt->textCursor();
cur.movePosition(QTextCursor::Start);
QPen pen(QColor(111, 1, 1));
QTextCharFormat cf;
cf.setFontPointSize(33);
cf.setTextOutline(pen);
cur.insertText("XXX", cf);
```



- 11)、bool **isAnchor**() const;
void **setAnchor**(bool anchor);
QString **anchorHref**() const
void **setAnchorHref**(const QString &value)
QStringList **anchorNames**() const
void **setAnchorNames**(const QStringList &names);

- 以上文本用于设置超链接，要设置超链接首先应使用 **setAnchor**()启用超链接，然后使用 **setAnchorHref**()设置超链接的目标地址(通常为一个网址)，**setAnchorNames**()用于设置超链接显示的名称，比如 [abcdef](http://www.aaa.com) 若关联的目标网址是"http://www.aaa.com"，则 **anchorNames**()就是字符串"abcdef"，而 **anchorHref**()是字符串"http://www.aaa.com"。
- 注意：我们通常见到的超链接是带下划线的蓝色文本，这些属性都需要我们进行设置，否则即使该文本是超链接，也不会是带下划线的蓝色文本，他看起来会像普通文本一样。
- 要使设置的超链接在外观上有所区别，还需要设置 QTextEdit 的 **textInteractionFlags** 属性包含 Qt::LinksAccessibleByMouse(即超链接可使用鼠标激活)，否则超链接在 QTextEdit 中看起来与普通文本是一样的。

示例(效果见图示):

```
QTextEdit *pt=new QTextEdit(this);
QTextCursor cur= pt->textCursor(); //获取光标
cur.movePosition(QTextCursor::Start); //设置光标位置
QTextCharFormat ff;
pt->setTextInteractionFlags(Qt::TextEditable|Qt::LinksAccessibleByMouse);
ff.setAnchor(true);
ff.setAnchorHref("http://www.baidu.com");
ff.setForeground(QBrush(QColor(1, 1, 111)));
ff.setFontUnderline(1);
ff.setFontPointSize(33);
cur.insertText("SSS", ff);
```



设置成功后,鼠标点击该文本,会在其周围显示一个虚线框

12)、QFont **font()** const; //返回文本格式的字体

```
void setFont(const QFont &font);
```

```
void setFont(const QFont &font, FontPropertiesInheritanceBehavior behavior)
```

设置文本格式的字体, FontPropertiesInheritanceBehavior 取值如下:

- QTextCharFormat :: FontPropertiesAll: 未被 font 显示设置的字体属性, 按照默认值设置
- QTextCharFormat :: FontPropertiesSpecifiedOnly: 未被 font 显示设置的字体属性将被忽略, 并保持相应的属性值不变。
- 注意: 以上枚举是对与 QFont 类相对应的字体属性才有效。

示例(效果见图示):

```
QTextCursor cur= pt->textCursor(); //获取光标
cur.movePosition(QTextCursor::Start); //设置光标位置
QTextCharFormat cf;
cf.setForeground(QBrush(QColor(111, 1, 1)));//设置字符颜色为红色, 该属性不是 QFont 属性
cf.setFontItalic(1);
QFont f; f.setPointSize(22);
cf.setFont(f, QTextCharFormat::FontPropertiesSpecifiedOnly); //保持原有的属性
cur.insertText("ABC", cf); //文本"ABC"有 22 磅值大小, 且为红色、斜体。
cf.setFont(f, QTextCharFormat::FontPropertiesAll); //把未由 f 设置的属性恢复为默认值
cur.insertText("SSS", cf); //文本"SSS"将只有 22 磅值大小, 显示为红色但不是斜体。
```



11.11 QTextCursor 类

一、QTextCursor 类基础

- 1、QTextCursor 类主要用于管理插入符(即光标)，还能够把表格或列表等复杂对象插入到 QTextDocument 中，并处理选择。该类可以创建/删除选择、并检索文本的内容。
- 2、QTextCursor 类用来实现以编程的方式创建并编辑 QTextDocument。
- 3、在 QPlainTextEdit 和 QTextEdit 类中使用 QTextCursor 的方法如下：
在文档中使用 textCursor()函数获取光标，然后编辑文档，然后把修改后的光标使用函数 setTextCursor()设置为文档的光标以确保使设置生效。
- 4、如果光标有选择，则给定的格式将应用于当前选择。请注意，当只选择块的一部分时，块格式将应用于整个块。光标移动仅限于有效的光标位置。
- 5、当前字符：指的是文档中光标位置 position()之前的字符。“当前块”是包含光标位置 position()的块。

二、QTextCursor 类中的函数

- 1、QTextCursor 类中有关处理文档元素和文档元素格式的函数在前面各章小节已讲解和使用过了，本小节仅把这些函数以表的形式列出其原型。

QTextCursor 类中与文档元素和文档元素格式有关的函数		
插入文档对象	QTextDocument *document() const;	光标关联的文档
	QTextList *createList(const QTextListFormat &format)	创建和插入列表
	QTextList *createList(QTextListFormat::Style style)	
	QTextList *insertList(const QTextListFormat &format)	
	QTextList *insertList(QTextListFormat::Style style)	
	void insertBlock()	插入块
	void insertBlock(const QTextBlockFormat &format)	
	void insertBlock(const QTextBlockFormat &format, const QTextCharFormat &charFormat)	
	void insertFragment(const QTextDocumentFragment &fragment)	插入文档片段
	QTextFrame *insertFrame(const QTextFrameFormat &format)	插入框架
	void insertHtml(const QString &html)	插入 HTML
	void insertImage(const QTextImageFormat &format)	插入图像
	void insertImage(const QTextImageFormat &format, QTextFrameFormat::Position alignment)	
	void insertImage(const QString &name)	
	void insertImage(const QImage &image, const QString &name = QString())	
文档元	QTextTable *insertTable(int rows, int columns, const QTextTableFormat &format)	插入表格
	QTextTable *insertTable(int rows, int columns)	
	void insertText(const QString &text)	插入文本
	void insertText(const QString &text, const QTextCharFormat &format)	
	void setBlockCharFormat(const QTextCharFormat &format)	设置和合并格式，其中 set 函数将替换当前格式，而
	void setBlockFormat(const QTextBlockFormat &format)	

素 格 式	void setCharFormat (const QTextCharFormat & <i>format</i>)	merge 函数将格式添加到当前格式。
	void mergeBlockCharFormat (const QTextCharFormat & <i>modifier</i>)	
	void mergeBlockFormat (const QTextBlockFormat & <i>modifier</i>)	
	void mergeCharFormat (const QTextCharFormat & <i>modifier</i>)	
获 取 文 档 对 象 信 息	QTextCharFormat blockCharFormat () const	获取块字符格式
	QTextBlockFormat blockFormat () const	获取块格式
	QTextCharFormat charFormat () const; //获取光标之前的字符格式，若光标位于非空文本块的开头，则返回光标后面的字符格式	获取字符格式
	QTextFrame * currentFrame () const	获取框架
	QTextList * currentList () const	获取列表
	QTextTable * currentTable () const	获取表格
	QTextBlock block () const	获取块
	int blockNumber () const //获取块的编号(从 0 开始)，仅适用于无复杂对象(如表格或框架)的文档	

2、构造函数

1)、**QTextCursor**(); //构造函数

QTextCursor(QTextDocument **document*); //构造一个指向文档开头的光标

QTextCursor(QTextFrame **frame*); //构造一个指向框架开头的光标

QTextCursor(const QTextBlock &*block*)

QTextCursor(const QTextCursor &*cursor*)

3、光标位置

2)、bool **atBlockEnd**() const; //若光标位于块的末尾，则返回 true，否则返回 false

bool **atBlockStart**() const; //若光标位于块的开头，则返回 true，否则返回 false

3)、bool **atEnd**() const; //若光标位于文档的末尾，则返回 true，否则返回 false

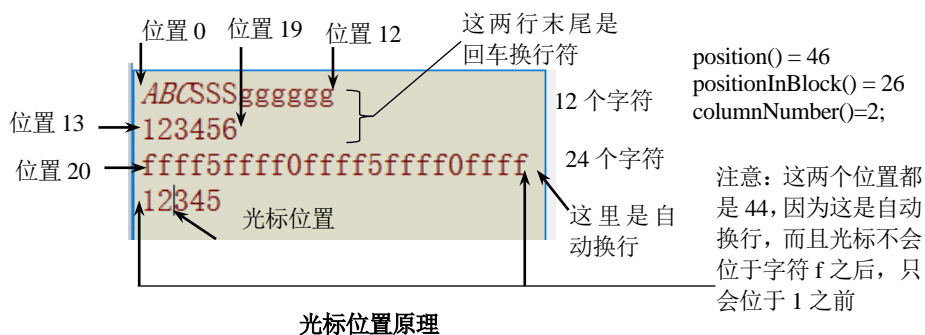
bool **atStart**() const; //若光标位于文档的开头，则返回 true，否则返回 false

4)、int **anchor**() const; //返回锚点位置。

5)、int **columnNumber**() const; //返回光标所在的列号，注意，这不是相对于块所在的列，见下图

int **positionInBlock**() const; //返回光标相对于块内的位置。原理见下图

int **position**() const; //返回文档中光标的绝对位置(即相对于整个文档的位置)，原理见下图



6)、void **setPosition**(int *pos*, MoveMode *m* = MoveAnchor);

使用模式 *m*，把光标移至文档中的绝对位置 *pos* 处(即相对于整个文档的位置 *pos* 处)。

注意：应调用 `QTextEdit::setTextCursor()` 以使光标的设置生效。

7)、bool **movePosition**(MoveOperation *operation*, MoveMode *mode* = MoveAnchor, int *n* = 1)

- 移动光标,通过模式 *mode* 和选项 *operation* 移动光标 *n* 次,若 *mode* 为 `KeepAnchor`,则光标移动的同时会选择所移过的文本,这与按住 `Shift` 再移动光标的效果相同。
- 若该函数操作成功,则返回 `true`, 否则返回 `false`(比如到达文档的末尾)。
- 注意：应调用 `QTextEdit::setTextCursor()` 以使光标的设置生效
- `MoveOperation` 和 `MoveMode` 枚举见下表

QTextCursor::MoveMode 枚举(无标志)

成员	值	说明
<code>QTextCursor::MoveAnchor</code>	0	移动光标时移动锚点
<code>QTextCursor::KeepAnchor</code>	1	保持锚点位置(若此时移动光标,会选择光标移过的文本)

QTextCursor::MoveOperation 枚举(无标志)

成员	值	说明
<code>QTextCursor::NoMove</code>	0	光标保持不动
<code>QTextCursor::Start</code>	1	移至文档开头
<code>QTextCursor::StartOfLine</code>	3	移至当前行的开头
<code>QTextCursor::StartOfBlock</code>	4	移至当前块的开头
<code>QTextCursor::StartOfWord</code>	5	移至当前单词的开头
<code>QTextCursor::PreviousBlock</code>	6	移至前一块的开头
<code>QTextCursor::PreviousCharacter</code>	7	移至上一个字符
<code>QTextCursor::PreviousWord</code>	8	移至前一个单词的开头
<code>QTextCursor::Up</code>	2	向上移一行
<code>QTextCursor::Left</code>	9	向左移一个字符
<code>QTextCursor::WordLeft</code>	10	向左移一个单词
<code>QTextCursor::End</code>	11	移至文档的末尾
<code>QTextCursor::EndOfLine</code>	13	移至当前行的末尾
<code>QTextCursor::EndOfWord</code>	14	移至当前单词的末尾
<code>QTextCursor::EndOfBlock</code>	15	移至当前块的末尾
<code>QTextCursor::NextBlock</code>	16	移至下一块的开头
<code>QTextCursor::NextCharacter</code>	17	移至下一个字符
<code>QTextCursor::NextWord</code>	18	移至下一个单词的开头
<code>QTextCursor::Down</code>	12	向下移一行
<code>QTextCursor::Right</code>	19	向右移一个字符
<code>QTextCursor::WordRight</code>	20	向右移一个单词
<code>QTextCursor::NextCell</code>	21	移至当前表格的下一个单元格的开头,若单元格是该行中的最后一个单元格,则光标移至下一行中的第一个单元格
<code>QTextCursor::PreviousCell</code>	22	移至当前表格的上一个单元格的开头,若单元格是该行中的第一个单元格,则光标移至上一行中的最后一个单元格
<code>QTextCursor::NextRow</code>	23	移至当前表格中下一行的第一个单元格的开头
<code>QTextCursor::PreviousRow</code>	24	移至当前表格中上一行的第一个单元格的开头

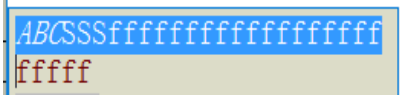
4、选择

- 8)、void `clearSelection()`; //清除选择，通过把锚点设置为光标位置来清除选择。
- 9)、void `removeSelectedText()`; //若有选择，则删除所选的内容，否则什么也不做。
- 10)、void `select(SelectionType selection)`; //该函数是以编程方式进行选择的函数
根据选择类型 `selection` 选择文档中的文本。该函数应调用 `QTextEdit::setTextCursor()` 以使光标的设置生效。`SelectionType` 枚举见下表

QTextCursor::SelectionType 枚举(无标志)		
成员	值	说明
QTextCursor::Document	3	选择整个文档
QTextCursor::BlockUnderCursor	2	选择光标下的文本块
QTextCursor::LineUnderCursor	1	先择光标下的文本行
QTextCursor::WordUnderCursor	0	选择光标下的单词



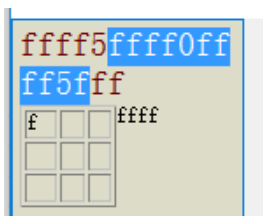
`select(QTextCursor::BlockUnderCursor)`



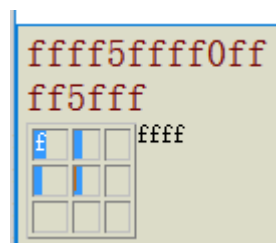
`select(QTextCursor::LineUnderCursor)`

块与行的区别

- 11)、QString `selectedText()` const;
返回当前选择的文本(不含格式信息，即纯文本)，注意：该函数返回的文本，可能会含有特殊的 Unicode 字符，比如，若从编辑器(比如 `QTextEdit`)中选择的文本跨越了行(即含换行符)，则文本将包含 Unicode U+2029 的段落分隔符而不是换行符'\n'，这可以使用 `QString::replace()`进行替换。
- 12)、QTextDocumentFragment `selection()` const
返回所选择的内容，包含其格式信息。`QTextDocumentFragment` 是一个文档片段，文档片段是文档中的一部分内容，也可以是整个文档。
- 13)、int `selectionEnd()` const
int `selectionStart()` const
返回光标的开始或结束位置，若没有选择则返回位置 `position()`。
- 14)、bool `hasSelection()` const; //若光标包含选择，则返回 true，否则返回 false。
- 15)、void `selectedTableCells(int *firstRow, int *numRows, int *firstColumn, int *numColumns)` const
获取所选择单元格的信息，并将其存储在该函数的参数中。若选择跨越表格中的单元格，则 `firstRow` 和 `firstColumn` 分别表示选择区域中的第一行和第一列的编号，`numRows` 和 `numColumns` 分别表示选区中的行数和列数。若选择未跨越单元格，则结果是无害的，但未定义(通常结果值为-1)。原理见下图
- 16)、bool `hasComplexSelection()` const;
若是复杂选择则返回 true，复杂选择是指不能简单的由 `selectionStart()`和 `selectionEnd()` 表示的选择。原理见下图

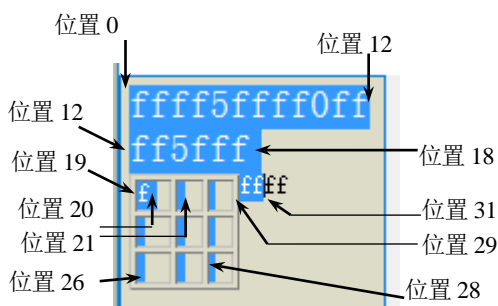


```
w = x = y = z = -1;
hasComplexSelection() = false;
selectionStart() = 5;
selectionEnd() = 16;
```



```
w = y = 0; x = z = 2;
hasComplexSelection() = true;
selectionStart() = 19;
selectionEnd() = 24;
```

```
int w,x,y,z;
selectedTableCells(&w, &x, &y, &z);
```



```
w = x = y = z = -1;
hasComplexSelection() = false;
selectionStart() = 0;
selectionEnd() = 31;
注意：此种情况，看上去是选择了表格中的表格单元，实际并未选择。
```

含有表格时的选择及位置原理

5、分组操作

17)、void beginEditBlock()

void endEditBlock()

以上函数用于把操作分组，即从撤消/重做角度看，位于这两个函数之间的动作被视为单个动作处理。比如

```
QTextEdit *pt=new QTextEdit;        QTextCursor cur=pt->textCursor();
cur.beginEditBlock();                cur.insertText("AAA");        cur.endEditBlock();
cur.beginEditBlock();                cur.insertText("BBB");        cur.endEditBlock();
pt->undo(); //将撤消对 BBB 的插入，最终插入 AAA。
```

18)、void joinPreviousEditBlock()

该函数会使在该函数之后的操作加入到之前的操作分组中，比如

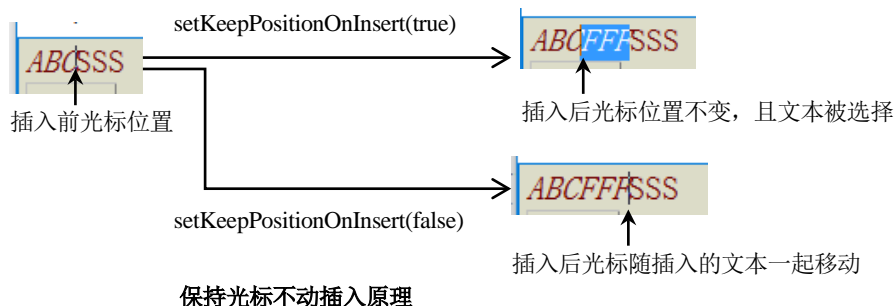
```
cur.beginEditBlock();                cur.insertText("AAA");        cur.endEditBlock();
.....
cur.joinPreviousEditBlock(); //加入以下操作到上一个操作分组
cur.insertText("BBB");
cur.endEditBlock();                //若调用 undo()操作，将撤消对"AAA", "BBB"二段文本的插入操作。
```

6、删除、保持光标不动插入、垂直移动 x 位置

- 19)、void **deleteChar**(); //删除光标位置之后或所选择的字符。
void **deletePreviousChar**(); //删除光标位置之前或所选择的字符。

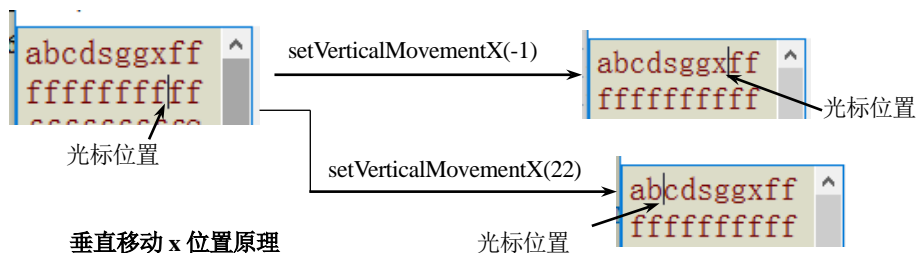
- 20)、bool **keepPositionOnInsert**() const
void **setKeepPositionOnInsert**(bool *b*)

在光标当前位置处插入文本时，光标是否应保持当前位置。默认为 false(不保持)，注意：若在光标当前位置之前插入文本，则光标始终移动。该函数应调用 `QTextEdit::setTextCursor()` 以使光标的设置生效，原理见下图。



- 21)、int **verticalMovementX**() const
void **setVerticalMovementX**(int *x*);

垂直移动时的 *x* 位置(注意：该位置不是按字符数计算的)，即当使用上下箭移动光标时，光标会移动至上一行或下一行的 *x* 位置。当光标水平移动时，自动清除 *x* 位置，当光标垂直移动时 *x* 位置保持不变。若 *x* 为 -1 表示未定义 *x* 位置。该函数应调用 `QTextEdit::setTextCursor()` 以使光标的设置生效，原理见下图



7、其他

- 22)、bool **isCopyOf**(const QTextCursor &*other*) const; 若该光标与光标 *other* 是彼此的副本，则返回 true。
23)、bool **isNull**() const; 若光标为空，则返回 true。空光标由默认构造函数创建。
24)、void **swap**(QTextCursor &*other*); 把此光标与光标 *other* 交换。qt5.0
25)、bool **visualNavigation**() const
void **setVisualNavigation**(bool *b*);

设置可视化导航，可视化导航意味着会跳过隐藏的文本段落，默认为 false。

8、以下为重新实现的操作符函数

bool operator!=(const QTextCursor &*other*) const


```
bool operator<(const QTextCursor &other) const
bool operator<=(const QTextCursor &other) const
QTextCursor &operator=(QTextCursor &&other)
QTextCursor &operator=(const QTextCursor &cursor)
bool operator==(const QTextCursor &other) const
bool operator>(const QTextCursor &other) const
bool operator>=(const QTextCursor &other) const
```

11.12 QTextDocument 类

- 1、QTextDocument 类是一个独立的类，该类用于处理格式化文本。
- 2、QTextDocument 类是结构化富文本文档的容器，支持样式化文本和各种类型的文档元素(比如列表、表格、框架和图像)，它们可以在 QTextEdit 中使用，也可以独立使用。

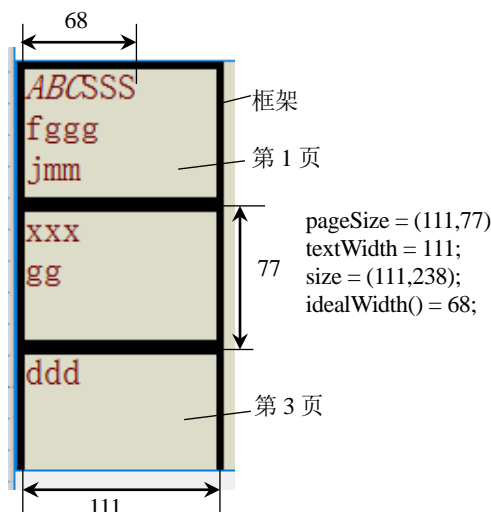
一、QTextDocument 类中的属性

- 1、**blockCount** : const int **访问函数:** int blockCount() const
获取文本块的数量，若文档中拥有表格工框架，则属性值未定义。默认值为 1。
- 2、**defaultFont** : QFont **访问函数:** QFont defaultFont() const; void setDefaultFont(const QFont &)
文档中文本使用的默认字体。
- 3、**defaultStyleSheet** : QString
访问函数: QString defaultStyleSheet() const; void setDefaultStyleSheet(const QString &)
默认样式表，默认样式表应用于插入到文档中的最近的 HTML 格式文本。更改默认样式表不会对文档的现有内容产生影响。
- 4、**defaultTextOption** : QTextOption
访问函数: QTextOption defaultTextOption() const; void setDefaultTextOption(const QTextOption &)
在创建 QTextBlock 时将其在其 QTextLayout 上设置该属性，这允许设置文档的全局属性。QTextOption 类封装了富文本的一些属性，包括文本对齐、布局方向、文字换行等，详见该类的讲解。
- 5、**documentMargin** : qreal **访问函数:** qreal documentMargin() const; void setDocumentMargin(qreal)
文档的边距
- 6、**indentWidth** : qreal **访问函数:** qreal indentWidth() const; void setIndentWidth(qreal)
获取和设置列表和文本块缩进的宽度，默认为 40。注：QTextListFormat::setIndent()和 QTextBlockFormat::setIndent 设置的缩进是此值的倍数。
- 7、**maximumBlockCount** : int
访问函数: int maximumBlockCount() const; void setMaximumBlockCount(int)
 - 文档最大块数，若文档中的块数大于该属性的值，则从文档的开头删除内容。负值和零表示文档有无限的块。默认为 0。
 - 该属性会立即作用于文档内容，该属性还会禁用撤消重做历史，且在有表格或框架的文档中未定义。
- 8、**modified** : bool **访问函数:** bool isModified() const; void setModified(bool m = true); //槽
获取和设置文档是否已被用户修改。注意：此处的修改只是一种状态，比如假设文档之前未改变(此属性为 false)，若用户插入一个字符，则此时文档变为被修改状态(true)，之后再插入或删除等字符文档都会保持已被修改的状态不变，也就是说一旦创建文档，该文档就会一直处于修改状态(true)，除非用户将其设置为 false。
- 9、**undoRedoEnabled** : bool **访问函数:** bool isUndoRedoEnabled() const; void setUndoRedoEnabled(bool)
是否启用撤消/重做，默认为 true。

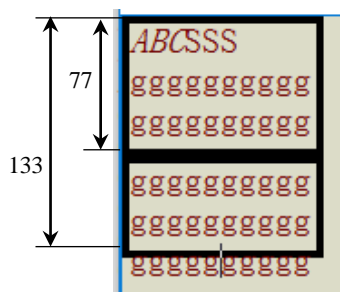
- 10、**useDesignMetrics** : bool **访问函数**: bool useDesignMetrics() const; void setUseDesignMetrics(bool)
- 是否启用设计度量来提高文本布局的准确性，若不使用设计度量，将使用 QAbstractTextDocumentLayout::setPaintDevice()上设置的绘制设备的度量标准。若使用设计度量，则布局的宽度将不再依赖于提示和像素舍入。默认为 false(不启用)

文档的页面

- 11、**pageSize** : QSizeF **访问函数**: QSizeF pageSize() const; void setPageSize(const QSizeF &)
- 描述文档的页面大小，默认情况下，对于空的新创建的文档，此属性的大小未定义。
 - 页面大小的单位由绘制到的设备决定，当绘制到屏幕上时，大小以逻辑像素为单位，当绘制到打印机上时，以点(1/72 英寸)为单位。页面原理见下图
 - 设置页面大小，可把文档分解为多个页面即超过一页，会使用第二页。
 - 若把宽度或高度设置为-1，则文档会在宽度或高度方向自动扩展或收缩。
- 12、**size** : const QSizeF **访问函数**: QSizeF size() const
- 返回文档的实际大小，相当于 documentLayout()->documentSize();默认情况下，对于新建的空文档，该属性的值依赖于配置。注意：宽度始终大于等于 pageSize().width()。页面原理见下图
- 13、**textWidth** : qreal **访问函数**: qreal textWidth() const; void setTextWidth(qreal)
- 设置文本的宽度，该函数其实就是 pageSize 属性的宽度值。页面原理见下图
- 14、qreal **idealWidth**() const; //这是个函数
- 文档实际使用宽度(理想宽度)，它总是小于等于 size().width()。页面原理见下图



文档页面



框架和页面是分离的
假设根框架高度设置为固定值 133，页面高度设置为 77，由图可见，当文档内容超过设置的固定框架的高度时，框架以下的内容不会显示，但实际上在框架之下仍有内容，因此在设置页面时应把框架垂直高度设置为自动扩展。

框架与页面

二、QTextDocument 类中的函数

- 1、**QTextDocument**(QObject *parent = Q_NULLPTR); //构造函数

QTextDocument(const QString &text, QObject *parent = Q_NULLPTR)

2、基本信息

- 1)、QTextFrame ***rootFrame**() const; //返回文档的根框架
- 2)、QTextObject ***objectForFormat**(const QTextFormat &f) const; //返回与格式 f 相关联的文本对象
- 3)、QTextObject ***object**(int **objectIndex**) const; //返回与 objectIndex 相关联的文本对象。
- 4)、bool **isEmpty**() const; //若文档为空则返回 true，否则返回 false。
- 5)、int **pageCount**() const; //返回文档中的页数
- 6)、QChar **characterAt**(int **pos**) const; //返回位置 pos 处的字符(相对于文档从 0 开始计数光标后的字符)
- 7)、int **characterCount**() const; //返回此文档的字符数
- 8)、QVector<QTextFormat> **allFormats**() const; //以向量的形式返回列表中所使用的所有格式。
- 9)、QTextBlock **begin**() const; //返回文档的第一个文本块。
QTextBlock **firstBlock**() const; //返回文档的第一个文本块
QTextBlock **end**() const; //返回文档的最后有效文本块
QTextBlock **lastBlock**() const; //返回文档的最后有效文本块
以上函数可用于遍历文档的块内容，代码如下：

```
for(QTextBlock t = doc->begin(); t != doc->end(); t = t.next()) qDebug()<< t.text();
```

3、基本操作(插入/获取 HTML、纯文本、打印、资源等)

- 10)、void **adjustSize**(); //该函数可根据文档页面的内容自动调整文档的大小。
- 11)、virtual void **clear**(); //清除文档，虚拟的
- 12)、QTextDocument ***clone**(QObject *parent = Q_NULLPTR) const; 创建一个该文档的副本。
- 13)、void **print**(QPagedPaintDevice ***printer**) const;
将文档打印到打印机，若文档已使用 **pageSize** 属性的高度进行分页，则按原样打印，若未分页，则创建文本的临时副本，并根据设置的大小把副本分成多个页面，默认情况下，文档有 2 厘米的边距，且页码打印在每页的底部。
- 14)、void **setHtml**(const QString &**html**);
使用给定的 HTML 格式文本 **html** 替换文档的全部内容(即，会清除文档之前的内容)。此函数还会重置撤消/重做历史。
- 15)、QString **toHtml**(const QByteArray &**encoding** = QByteArray()) const;
返回以 HTML 形式表示的字符串形式，注意，返回的文本会比较多，包含了 html 头等生成 HTML 文件所需的信息。参数 **encoding** 用于在返回的 html 标头中指字符串字符集属性的值，若未指定 **encoding** 则不会产生这些信息。
- 16)、void **setPlainText**(const QString &**text**);
使用文本 **text** 替换文档的全部内容(即，会清除文档之前的内容)。此函数还会重置撤消/重做历史。
- 17)、QString **toPlainText**() const;
QString **toRawText**() const; //qt5.9
返回文档中包含的纯文本(包含文档中的所有文本)。这两个函数返回相同的值，**toRawText** 函数返回的文本可能会含有特殊的 Unicode 字符，比如，若从编辑器(比如

QTextEdit)中选择的文本跨越了行(即含换行符), 则文本将包含 Unicode U+2029 的段落分隔符而不是换行符'\n', 而 toPlainText()返回的是'\n'。嵌入对象使用 unicode 值 U+FFFC 表示。

18)、QVariant **resource**(int *type*, const QUrl &*name*) const;

void **addResource**(int *type*, const QUrl &*name*, const QVariant &*resource*);

把资源 resource 使用名称 name 添加到资源缓存, 其中 type 是资源的类型, 其取值应是 QTextDocument::ResourceType 枚举的值, 见下表

QTextDocument::ResourceType 枚举(无标志)		
成员	值	说明
QTextDocument::HtmlResource	1	资源包含 HTML
QTextDocument::ImageResource	2	资源包含图像数据。目前不支持 QVariant::Icon
QTextDocument::StyleSheetResource	3	资源包含 CSS
QTextDocument::UserResource	100	用户自定义类型的资源

示例:

```
QTextEdit pt=new QTextEdit ;
QTextCursor cur=pt->textCursor();
QTextDocument *pd=pt->document();
//把 QImage 对象添加到资源缓存, 名称为 FFF
pd->addResource(QTextDocument::ImageResource, QUrl("FFF"),
               QVariant(QImage("F:/li.png")));

//使用 QTextCursor 插入资源
cur.insertImage("FFF");
QTextImageFormat mf;   mf.setName("FFF");      cur.insertImage(mf);
//也可使用 HTML 的形式插入资源
pt->append("<img src=\"FFF\" />");
pt->insertHtml("<img src=\"FFF\" />");
```

19)、Qt::CursorMoveStyle **defaultCursorMoveStyle**() const;

void **setDefaultCursorMoveStyle**(Qt::CursorMoveStyle *style*);

光标的默认移动样式, 默认为 Qt::LogicalMoveStyle, 枚举 Qt::CursorMoveStyle 见下表

Qt::CursorMoveStyle 枚举(无标志)		
成员	值	说明
Qt::LogicalMoveStyle	0	在从左到右的布局中, 按左箭头时减少光标位置, 按右键时增加光标位置, 若文本是从右到左的, 则按相反方向进行。
Qt::VisualMoveStyle	1	无论文本的方向如何, 按左箭头时都向左移动, 按右箭头向右移动

4、查找

20)、QTextCursor **find**(const QString &*subString*, const QTextCursor &*cursor*,

FindFlags *options*=FindFlags()) const;

QTextCursor **find**(const QString &*subString*, int *position* = 0, FindFlags *options* = FindFlags()) const;

- 在文档中查找字符串 subString,

- 参数 `cursor` 和 `position` 表示开始查找的位置,
- `options` 表示是向前还是向后查找、或是否区分大小写等。枚举 `FindFlag` 见下表
- 若找到字符串 `subString`, 则返回位于该处的光标, 否则返回空的光标。
- 若光标 `cursor` 已经选中了文本 `subString`, 则在选中的文本之后开始查找。此规则不适用于使用 `position` 参数(即第 2 个函数)指定的查找。
- 默认情况下, 搜索区分大小写, 且可在整个文档中查找。

QTextDocument::FindFlag 枚举 标志: QTextDocument::FindFlags		
成员	值	说明
QTextDocument::FindBackward	0x00001	向后查找
QTextDocument::FindCaseSensitively	0x00002	区分大小写
QTextDocument::FindWholeWords	0x00004	只查找完全匹配的单词

QTextCursor **find**(const QRegExp &*expr*, int *from* = 0, FindFlags *options* = FindFlags()) const;

QTextCursor **find**(const QRegExp &*expr*, const QTextCursor &*cursor*,
FindFlags *options* = FindFlags()) const;

QTextCursor **find**(const QRegularExpression &*expr*, int *from* = 0, FindFlags *options* = FindFlags()) const;

QTextCursor **find**(const QRegularExpression &*expr*, const QTextCursor &*cursor*,
FindFlags *options* = FindFlags()) const;

以上函数与前面介绍的查找函数相同, 只是查找的是与正则表达式 `expr` 匹配的文本。

21)、QTextBlock **findBlock**(int *pos*) const;

返回包含第 `pos` 个字符的文本块, `pos` 从 0 计数, 回车换行符需要计算一个字符。
比如 `findBlock(2);` //返回包含第 2 个字符的文本块。

22)、QTextBlock **findBlockByLineNumber**(int *lineNumber*) const;

QTextBlock **findBlockByNumber**(int *blockNumber*) const;

返回行编辑号为 `blockNumber` 或块编辑号为 `blockNumber` 的文本块, 这里的行编辑号与块编辑号通常是相同的, 这与 `QTextCursor::select()` 函数有些不同。另见 `QTextBlock::setLineCount()` 函数。

5、撤消重做

23)、bool **isRedoAvailable**() const; //若重做可用则返回 true, 否则返回 false。

bool **isUndoAvailable**() const; //若撤消可用则返回 true, 否则返回 false。

24)、int **availableRedoSteps**() const; //返回可用的重做步骤数量。

int **availableUndoSteps**() const; //返回可用的撤消步骤数量。

25)、int **revision**() const; //返回文档的修订次数(若启用了撤消)。每修改一次文档, 便会加 1。

26)、void **redo**(); //重做, 槽

void **undo**(); //撤消, 槽

void **redo**(QTextCursor **cursor*); //重做

void **undo**(QTextCursor **cursor*); //撤消

27)、void **clearUndoRedoStacks**(Stacks *stacksToClear* = UndoAndRedoStacks);

清除由 `stacksToClear` 指定的撤消或重做或两者(默认)堆栈上的命令。若命令被清楚, 会发送 `undoAvailable()`或 `redoAvailable()`信号, `Stacks` 枚举见下表

QTextDocument::ResourceType 枚举(无标志)		
成员	值	说明
QTextDocument::UndoStack	0x01	撤消堆栈
QTextDocument::RedoStack	0x02	重做堆栈
QTextDocument::UndoAndRedoStacks		UndoStack RedoStack

6、文档布局

- 28)、QAbstractTextDocumentLayout ***documentLayout**() const; //返回文档的文档布局
void **setDocumentLayout**(QAbstractTextDocumentLayout **layout*); //设置文档的文档布局
- 29)、int **lineCount**() const; //若布局支持, 则返回文档的行数, 否则与块的数量相同。
- 30)、void **markContentsDirty**(int position, int *length*);
把由位置 `position` 和长度 `length` 指定的内容标记为"dirty(脏)", 通知文档需要重新布局。
- 31)、void **drawContents**(QPainter **p*, const QRectF &*rect* = QRectF());
使用绘制器 `p` 绘制文档内容, 内容裁剪为 `rect` 大小, 若 `rect` 为空, 则不裁剪绘制的内容。
- 32)、QString **metaInformation**(MetaInformation *info*) const;
void **setMetaInformation**(MetaInformation *info*, const QString &*string*);
返回或设置由 `info` 指定的类型的文档的元信息。`MetaInformation` 枚举见下表

QTextDocument::MetaInformation 枚举(无标志)		
成员	值	说明
QTextDocument::DocumentTitle	0	文档的标题
QTextDocument::DocumentUrl	1	文档的网址。 <code>loadResource()</code> 函数加载相关资源时使用此 <code>url</code> 作为基

7、受保护的函数

- 33)、virtual QTextObject ***createObject**(const QTextFormat &*format*); //虚拟的
根据格式 `format` 创建并返回一个新的 `QTextObject` 对象, `QTextObject` 对象始终通过此方法创建, 重新实现该函数可实现自定义 `QTextObject` 对象。
- 34)、virtual QVariant **loadResource**(int *type*, const QUrl &*name*); //虚拟的
使用给定的名称 `name` 从资源加载指定类型的数据。该函数由富文本引擎调用。该函数应使用 `addResource()`将资源添加到缓存中。

8、信号

- 1)、void **blockCountChanged**(int *newBlockCount*);
文档中文本块的总数变化时, 发送此信号。`newBlockCount` 为新的总数。
- 2)、void **contentsChange**(int *position*, int *charsRemoved*, int *charsAdded*);

只要文档内容变化，就发送此信号(注意：应用格式(比如加粗字体)也会发送此信号)。
position 表示发生变化的字符的位置，charsRemoved 表示删除的字符数，charsAdded 表示添加的字符数。

3)、void **contentsChanged**(); //只要文档内容变化，就发送此信号。

4)、void **cursorPositionChanged**(const QTextCursor &cursor);

只要光标位置变化，就发送此信号。还可使用 QTextEdit::cursorPositionChanged() 信号。

5)、void **documentLayoutChanged**(); //设置新文档布局时，发送此信号

6)、void **modificationChanged**(bool *changed*); //当修改文档时，发送此信号。

7)、void **redoAvailable**(bool *available*); //只要重做操作变为可用或不可用，就会发送此信号。

void **undoAvailable**(bool *available*); //只要撤消操作变为可用或不可用，就会发送此信号。

8)、void **undoCommandAdded**();

每次增加新的撤消时，发送此信号。比如插入一个字符，再删除该字符，就增加了两次撤消，所以会发送两次该信号。

Qt 的文档通常没有单位，通常，打印的单位为磅值(point)，其他情形的单位为像素(pixel)。

示例：把文档内容输出为 PDF 和 odt 文件

说明：本示例需要在 pro 文件中加入 QT+=printsupport，并在头文件中使用#include<QPrinter>，注意：最好使用 MinGW 编译器，若使用 VC++ 有可能会无法打开 QPrinter 头文件。

//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>
#include<QPrinter> //需包含此头文件
class B:public QWidget{ Q_OBJECT
public:
    QPushButton *pb1;    QTextEdit *pt;
    B(QWidget *p=0):QWidget(p){
        pt=new QTextEdit(this); pt->resize(333,222);
        QTextCursor cur= pt->textCursor(); //获取光标
        cur.movePosition(QTextCursor::Start); //设置光标位置
        QTextCharFormat cf;
        cf.setForeground(QBrush(QColor(111,1,1))); //设置字符颜色为红色
        cf.setFontPointSize(22);
        cur.insertText("abcde",cf);
        QTextDocument *pd=pt->document();
        pd->setPageSize(QSize(111,77)); //设置页面大小
        //创建按钮并添加快捷键
        pb1=new QPushButton("Blod",this); pb1->move(22,244);
        QAction *pppl=new QAction("AAA");
        pppl->setShortcut(QKeySequence("Ctrl+F"));
        pb1->addAction(pppl);
        QObject::connect(pb1,&QPushButton::clicked,this,&B::f1);
        QObject::connect(pppl,&QAction::triggered,this,&B::f1);
    }
public slots:
    void f1() {QTextDocument *pd=pt->document(); //获取文档
               QPrinter pr; //创建打印机
               pr.setOutputFileName("F:/1.pdf"); //设置输出文件的路径和名称
            }
```



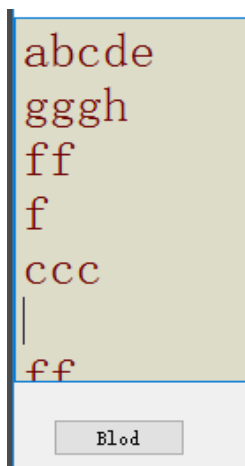
```

pr.setOutputFormat(QPrinter::PdfFormat); //设置输出文件的格式
pr.setPageSize(QPrinter::A7); //设置打印页面大小(此处为 A7 纸大小)
pd->print(&pr); //调用 print() 函数打印文档
//使用 QTextDocumentWriter 类把文档保存为.odt 文件，该类详见后文
QTextDocumentWriter wr("F:/1.odt"); //输出文件的路径和名称
wr.write(pd); //把文档 pd 的内容输出到 1.odt 文件
});
#endif // M_H

//m.cpp 文件内容
#include "m.h"
int main(int argc, char *argv[]) { QApplication app(argc, argv);
    B w; w.resize(444,333); w.show(); return app.exec(); }

```

运行结果及说明



运行后在文档中随便输入内容，然后按下"Blod"按钮或按下 Ctrl+F 按钮便可把文档内容输出到PDF和odt文件，在 F:/盘下可看到下图的两个文件，打开 PDF 文件后，第 1 页的内容见右图



1.pdf



1.odt

abcde
gggh
ff
f
ccc

页数是 Qt 加上去的，我们无法控制

11.13 其他类

QTextOption、QTextDocumentFragment、QTextDocumentWriter 类

一、QTextOption 类

QTextOption 类封装了富文本的一些属性，包括文本对齐、布局方向、文字换行等，该类的设置会影响到整个文档，使用该类设置的属性需要使用 QTextDocument::defaultTextOption 属性将其添加到文档，下面为该类的成员函数

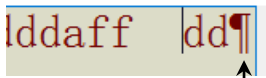
- 1、QTextOption(); //构造函数
- QTextOption(Qt::Alignment alignment)
- QTextOption(const QTextOption &other)
- 2、Qt::Alignment alignment() const
- void setAlignment(Qt::Alignment alignment); //设置文本的对齐方式
- 3、Qt::LayoutDirection textDirection() const; //文本布局的方向(即从左到右，还是从右到左)
- void setTextDirection(Qt::LayoutDirection direction);
- 4、bool useDesignMetrics() const;
- void setUseDesignMetrics(bool enable);
- 是否启用设计度量来提高文本布局的准确性，若不使用设计度量，将使用绘制设备的度量标准(默认行为)
- 5、void setWrapMode(WrapMode mode)
- WrapMode wrapMode() const
- 设置换行模式，枚举 WrapMode 见下表

QTextOption::WrapMode 枚举(无标志)		
作用：描述文本应怎样换行		
成员	值	说明
QTextOption::NoWrap	0	文本不换行
QTextOption::WordWrap	1	文本在单词边界处换行
QTextOption::ManualWrap	2	与 NoWrap 相同
QTextOption::WrapAnywhere	3	即使在单词的中间也将文本换行
QTextOption::WrapAtWordBoundaryOrAnywhere	4	若能在单词边界处换行则在此处换行，否则会在行的适当位置换行，即使是在一个单词的中间。

- 6、Flags flags() const
- void setFlags(Flags flags)
- 设置标志，枚举 Flag 见下表

QTextOption::Flag 枚举	
标志：QTextOption::Flags	

成员	值	说明
QTextOption::IncludeTrailingSpaces	0x80000000	设置该选项将使 QTextLine::naturalTextWidth() 返回的值包含文本中尾随空格的宽度。
QTextOption::ShowTabsAndSpaces	0x1	用点显示空格，箭头显示制表符
QTextOption::ShowLineAndParagraphSeparators	0x2	显示段落分隔符的符号
QTextOption::ShowDocumentTerminator	0x10	显示文档末尾的符号，qt5.7
QTextOption::AddSpaceForLineAndParagraphSeparators	0x4	换行时考虑为绘制分隔符而添加的距离(原理见下图)。
QTextOption::SuppressColors	0x8	禁止字符格式中的颜色更改(主选择除外)



这是显示的段落分隔符

若设置了 AddSpaceForLineAndParagraphSeparators，则再输入一个字符文本就会换行了，若未设置该选项，则还要输入两个字符才会换行。

以下函数用于设置制表符，制表符的原理可查看 QTextBlockFormat::setTabPositions ()函数

7、QList<qreal> **tabArray**() const; //选项卡位置列表。

void **setTabArray**(const QList<qreal> &*tabStops*);

8、void **setTabStopDistance**(qreal *tabStopDistance*); //qt5.10

qreal **tabStopDistance**() const; //qt5.10

制表符的默认距离设置为由 tabStopDistance 指定的值。

9、QList<Tab> **tabs**() const;

void **setTabs**(const QList<Tab> &*tabStops*);

设置制表符列表。QTextOption::Tab 是一个嵌套类，如下：

Tab(); //构造函数，制表符距离默认为 80。

Tab(qreal pos, TabType type, QChar delim=QChar());

Tab 类的成员变量如下

qreal Tab::**position** //制表符移动的距离

QChar Tab::**delimiter** //若 type 为 DelimiterTab，则制表符直到在文本中找到此字符为止。

TabType Tab::**type** //制表符的类型，TabType 枚举见下表

QTextOption::TabType 枚举(无标志)

作用：描述制表符的类型

成员	值	说明
QTextOption::LeftTab	0	左选项卡
QTextOption::RightTab	1	右选项卡
QTextOption::CenterTab	2	居选项卡
QTextOption::DelimiterTab	3	制表符停止在特定的分隔字符处

二、QTextDocumentFragment 类(文档片段)

1、文档片段是文档中的一部分内容，也可以是文档中的一块整式文本，也可以是整个文档。

2、文档片段可通过以下方式创建

QTextDocumentFragment 的构造函数

QTextCursor::selection();函数会返回一个文档片段。

使用 QTextDocumentFragment 类的静态函数 fromPlainText()和 fromHtml()创建。

3、文档片段可使用 QTextCursor::insertFragment()函数来插入文档。

4、文档片段的内容可使用该类的成员函数 toPlainText()和 toHtml()获取。

5、QTextDocumentFragment 类中的函数

1)、QTextDocumentFragment(); //默认构造函数

QTextDocumentFragment(const QTextDocumentFragment &other); //复制构造函数

2)、QTextDocumentFragment(const QTextDocument *document)

把 document 转换为文档片段(不含文档标题等元信息)

3)、QTextDocumentFragment(const QTextCursor &cursor)

从光标选择的内容创建一个文档片段，若没有选择则创建一个空文档片段。

4)、static QTextDocumentFragment fromHtml(const QString &text); //静态的

static QTextDocumentFragment fromHtml(const QString &text,

const QTextDocument *resourceProvider); //静态的

根据 text 中的一段 HTML 格式的文本创建一个文档片段，尽可能保留格式。若提供的 HTML 包含对外部资源的引用，则由 resourceProvider 加载。

5)、static QTextDocumentFragment fromPlainText(const QString &plainText); //静态的

返回包含 plainText 的文档片段。

6)、bool isEmpty() const; //若文档片段为空则返回 true，否则返回 false。

7)、QString toHtml(const QByteArray &encoding = QByteArray()) const

使用指定的编码(比如 UTF-8，ISO 8859-1)将文档片段的内容作为 HTML 的格式返回

8)、QString toPlainText() const; //以纯文本(即无格式信息)返回文档片段的文本。

三、QTextDocumentWriter 类

1、QTextDocumentWriter 类用于向文件或其他设备写入 QTextDocument 文档。

2、使用 QTextDocumentWriter 类的步骤如下：

使用文件名或设备创建一个 QTextDocumentWriter 对象，并为其指定要写入的文档的格式，然后使用 write()函数把文档写入到前面指定的文件或设备。文件名或设备、文档的格式也可在以后设置。文档的格式是使用字符串(不区分大小写)进行描述的。示例如下：

QTextDocumentWriter wr("F:/1.odf", "odf"); //准备把文档以 odf 格式写入到文件 1.odf 中。

wr.write(doc); //写入文档到文件。

3、QTextDocumentWriter 类中的函数

1)、QTextDocumentWriter(); //构造函数

QTextDocumentWriter(QIODevice *device, const QByteArray &format);

QTextDocumentWriter(const QString &fileName, const QByteArray &format = QByteArray());

- 2)、void **setFileName**(const QString &*fileName*); //设置将要写入的文件的文件名。
 QString **fileName**() const;
- 3)、void **setFormat**(const QByteArray &*format*); //设置将要写入的文档的格式，格式用字符串描述
 QByteArray **format**() const;
- 4)、void **setDevice**(QIODevice **device*); //设置将要写入的设备(会删除旧设备)。
 QIODevice ***device**() const;
- 5)、void **setCodec**(QTextCodec **codec*); //设置写入时的编码格式，默认为 UTF-8
 QTextCodec ***codec**() const;
- 6)、bool **write**(const QTextDocument **document*); //把文档 document 写入指定的设备或文件。
 bool **write**(const QTextDocumentFragment &*fragment*); //把文档片段 fragment 写入指定的设备或文件
- 7)、QList <QByteArray> **supportedDocumentFormats**();
 返回目前支持的文档格式列表，默认可写入的格式有"ODF"(OpenDocument format),
 "HTML", "plaintext"(纯文本)。

11.14 QSyntaxHighlighter 类(语法高亮)

一、语法高亮基础

- 1、语法高亮，是用于使编辑的文本以各种不同的格式(比如颜色)呈现在用户面前，比如我们使用的 Qt Creator 的文本编辑器就是个例子，见下图

```
QTextCursor cur=pt->textCursor();  
// QTextCharFormat ff;  
QTextDocument *pd=pt->document(); //获取文档
```

语法高亮的效果

- 2、实现语法高亮的思想其实很简单，就是另外使用一个类来专门设置文档中显示的文本的格式，通常使用 QTextCharFormat 类来设置文本的格式。下面是实现 Qt 语法高亮的步骤
 - 1)、子类化 QSyntaxHighlighter。
 - 2)、重新实现 QSyntaxHighlighter::highlightBlock()函数(必须实现，这是纯虚函数)，在该函数内根据您的要求和规则，重新实现需要语法高亮的文本。
 - 3)、创建 QSyntaxHighlighter 子类的对象，并为其指定一个需要语法高亮的文档。
- 3、注意：语法高亮只能同时作用于一个文档。
- 4、下面举一下实现语法高亮的最小示例作为讲解。

示例：最小的语法高亮示例

//m.h 文件的内容

```
#ifndef M_H  
#define M_H  
#include<QtWidgets>  
class H:public QSyntaxHighlighter{public:    //子类化该类  
    H(QTextDocument *parent=0):QSyntaxHighlighter(parent) {}  
    void highlightBlock(const QString &text) { //每当在文档中插入或删除文本时都会调用该函数。  
        qDebug()<<text;    //text 参数是文档中当前块的内容。  
        QTextCharFormat f;  
        f.setFontPointSize(22);    //设置更大一些的字体  
        f.setForeground(Qt::darkMagenta);    //设置字体的颜色为紫色  
        setFormat(3, 7, f);    //把从第 3 个字符(从 0 计数)开始的 7 个字符设置为格式 f。  
    } };  
class B:public QWidget{    Q_OBJECT  
public:    QTextEdit *pt;  
    B(QWidget *p1=0):QWidget(p1) {  
        pt=new QTextEdit(this); pt->resize(333,222);  
        QTextCursor cur= pt->textCursor();  
        cur.movePosition(QTextCursor::Start);  
        QTextCharFormat cf;    //设置字符格式  
        cf.setForeground(QBrush(QColor(111,1,1)));  
        cur.insertText("abcde",cf);    //随意插入一些文本  
        cur.insertTable(3,3);    //随意插入一个表格  
        //可把以下代码放于一个槽函数中，以便在需要时才使用高亮语法。  
        QTextDocument *pd=pt->document();
```

```

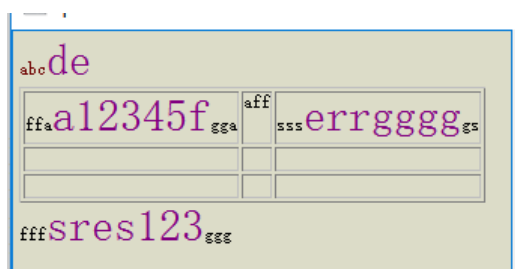
        H *q=new H(pd);    //把当前文档传递给类H，以使语法高亮作用于该文档。
    }
};

#endif // M_H

//m.cpp 文件内容
#include "m.h"
int main(int argc, char *argv[]) {    QApplication app(argc,argv);
    B w;    w.resize(444,333);    w.show();    return app.exec(); }

```

运行结果及说明



由图可见，当在文档中输入内容时，每个文本块从第 3 个字符开始的 7 个字符都被自动设置为了 22 点大小的紫色文字。

二、QSyntaxHighlighter 类中的函数

1、公有函数

- 1)、**QSyntaxHighlighter**(QObject *parent); //构造函数
- QSyntaxHighlighter**(QTextDocument *parent);
- 2)、QTextDocument *document() const; //返回安装了该语法高亮的文档。
- 3)、void setDocument(QTextDocument *doc); //把语法高亮安装到文档 doc 上。
- 4)、void rehighlight(); //把高亮显示重新应用于整个文档，槽
- 5)、void rehighlightBlock(const QTextBlock &block); //把高亮显示重新应用于块 block，槽

2、受保护的函数

- 6)、virtual void highlightBlock(const QString &text) = 0; //受保护的，**纯虚函数**
这是必须实现的纯虚函数，在该函数内根据您的要求和规则，重新实现需要语法高亮的文本。每当在文档中插入或删除文本时都会调用该函数，参数 text 为当前块的文本。
 - 7)、QTextBlock currentBlock() const; //返回当前文本块，受保护的
 - 8)、void setFormat(int start, int count, const QColor &color); //受保护的
void setFormat(int start, int count, const QFont &font); //受保护的
void setFormat(int start, int count, const QTextCharFormat &format); //受保护的
- 把字符格式 format 重置或合并为从 start 开始的 count 个字符，若 count 为 0，则不执行任何操作，start 从 0 开始计数。
 - 以上函数只能作用于当前文本块。
 - 注意：文档本身并不会被修改，这意味着，若之前 QTextDocument 的 modified 属性为 false，则使用该函数设置文本字符的格式后 modified 仍为 false。
 - 另外还要注意，第 3 个函数是合并而不重置格式。
 - 第 1 个函数仅用于设置文本的颜色，其他规则相同。

- 第 2 个函数用于设置文本的字体, 但会把文档的其他属性重置为默认值(即不是合并格式), 其他规则相同。

9)、QTextCharFormat **format**(int *position*) const; //受保护的

返回当前文本块内位置 *position* 处的字符格式

10)、int **previousBlockState**() const; //返回当前块的前一个文本块的状态, 受保护的

int **currentBlockState**() const; //返回当前文本块的状态, 受保护的

void **setCurrentBlockState**(int *newState*); //设置当前文本块的状态, 受保护的

注: 文本状态就是一个存储整数值的函数, 使用这个整数值可以把高亮语法作用于多个文本块, 比如像注释 “/*.....*/” 这样的情形, 以上函数的具体使用, 该类的帮助文档有详细的算法。

11)、QTextBlockUserData ***currentBlockUserData**() const; //受保护的

返回之前添加到当前文本块的 QTextBlockUserData 对象(即用户数据元素)

12)、void **setCurrentBlockUserData**(QTextBlockUserData **data*); //受保护的

- 把用户数据元素 *data* 添加到当前文本块, *data* 的所有权被传递给底层的文档。使用用户数据元素, 可以实现括号的匹配, 具体使用见后文示例。
- QTextBlockUserData 类仅有一个析构函数, 除此之外包括公有的构造函数都没有, 所以这个类的主要作用就是子类化该类, 然后在该类中存储一些信息以在其他地方使用, 比如

```
class C:public QTextBlockUserData{public: QString s; };
//然后就可以使用以上两个函数来使用类 C, 比如
C *pc = new C; pc->s="("; //赋值
setCurrentBlockUserData(pc); //设置用户数据元素
.....
C *pc1 = (C*)currentBlockUserData(); //读取用户数据元素
qDebug()<<pc1.s;
```

3、最后需要注意的是, 在纯虚函数 **highlightBlock()** 中使用 **setFormat()** 函数设置的格式不会被保留, 下面举一示例说明这一问题

示例: 格式的保留与否

//m.h 文件的内容

```
#ifndef M_H
```

```
#define M_H
```

```
#include<QtWidgets>
```

```
class H:public QSyntaxHighlighter{ public: int i; //用于存储一个值。
H(QTextDocument *parent=0):QSyntaxHighlighter(parent){ i=0; } //初始化 i。
```

```
void highlightBlock(const QString &text){
    QTextCharFormat f; f.setFontPointSize(33);
    setFormat(i++, 1, f); } //依次修改各个字符的格式
```

```
class B:public QWidget{ Q_OBJECT
```

```
public: QTextEdit *pt;
```

```
B(QWidget *p1=0):QWidget(p1){
```

```
pt=new QTextEdit(this); pt->resize(333, 222);
```

```
pt->setText("12345"); //随意插入一些文本
```

```
QTextDocument *pd=pt->document(); H *q=new H(pd); //安装语法高亮。
```



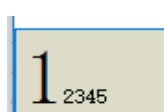
```

    });
#endif // M_H

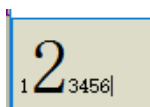
//m.cpp 文件内容
#include "m.h"
int main(int argc, char *argv[]) {    QApplication app(argc,argv);
    B w;    w.resize(444,333);    w.show();    return app.exec(); }

```

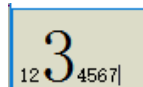
运行结果及说明如下图



初始状态



在 5 之后输入一个字符，则设置字符 2 的格式，但字符 1 的格式被取消



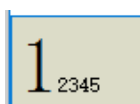
继续在 6 之后输入一个字符，则设置字符 3 的格式，但字符 2 的格式又被取消了

//使用相同原理的如下代码代替上例的 H 类，其效果见后图

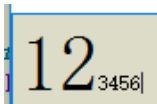
```

class H:public QSyntaxHighlighter{
public:    QTextCursor cur;    QTextDocument *pd;    //文档和光标
    H(QTextDocument *parent=0):QSyntaxHighlighter(parent){
        pd=document();    //获取与语法高亮相关的文档
        cur=QTextCursor(pd);    //创建与 pd 相关联的光标
        cur.movePosition(QTextCursor::Start);    //移动光标至文档开头
    }
void highlightBlock(const QString &text){
    QTextCharFormat f;    f.setFontPointSize(33);
    cur.movePosition(QTextCursor::Right,QTextCursor::KeepAnchor);    //向右选择一个字符。
    cur.setCharFormat(f);    //设置格式。
    cur.clearSelection();    //清除选择
}    };

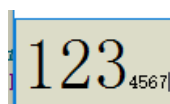
```



初始状态



在 5 之后输入一个字符，设置字符 2 的格式，其他字符的格式被保留



继续在 6 之后输入一个字符，设置字符 3 的格式，其他字符的格式仍被保留

作者：黄邦勇帅(原名：黄勇)

2018-6-21

本文作者：黄邦勇帅(原名：黄勇)

本文要求读者已经非常熟悉 C++ 语法。若读者不熟悉 C++ 语法，推荐参阅《C++ 语法详解》(作者：黄勇)一书，电子工业出版社出版。

本文主要讲解了 Qt 的 2D 绘图及一些图形方面的知识，本文列举了详细的示例进行说明，同时本文也是非常方便、快捷的编写 Qt 程序的查阅资料，可方便的查阅到相关内容的原理，以及怎样使用该内容。本文内容由浅入深，易学易懂。

本文使用的是 windows 10 操作系统，Qt 版本为 Qt5.10.1，Qt Creator 的版本为 Qt Creator 4.5.1 本文内容完全属于个人见解与参考文献的作者无关，限于水平有限，其中难免有误解之处，望指出更正。

声明：禁止抄袭、复印、转载本文，本文作者拥有完全版权。

主要参考文献：

- 1、C++语法详解 黄勇 编著 电子工业出版社 2017 年 7 月
- 2、Qt Creator 快速入门(第 3 版) 霍亚飞 编著 北京航空航天大学出版社 2017 年 1 月
- 3、C++ GUI Qt4 编程(第 2 版) [加拿大] Jasmin Blanchette [英] Mark Summerfield 著 闫锋欣 曾泉人 张志强 译 电子工业出版社 2008 年 8 月
- 4、Qt 5 开发及实例 陆文周 主编 电子工业出版社 2014 年 1 月
- 5、计算机图形学(第 4 版) [美] Donald Hearn、M. Pauline Baker、Warren R.Carithers 著 蔡士杰 杨若瑜 译 电子工业出版社 2014 年 11 月
- 6、计算机图形学原理及实践 C 语言描述 [美] James D.Foley、Andries van Dam、Steven K.Feiner、John E.Hughes 著 唐泽圣 董上海 李华 吴恩华 汪国平 等译 机械工业出版社 2004 年 3 月
- 7、计算机图形学原理及算法教程(Visual C++版) (第 2 版) 和青芳 编著 清华大学出版社 2010 年 12 月
- 8、计算机图形学(第 3 版) 孙家广 等编著 清华大学出版社 1998 年 9 月
- 9、图像工程(上册)----图像处理(第 2 版) 章毓晋 编著 清华大学出版社 2006 年 3 月
- 10、计算机图形学及学用编程技术 李春雨 等编著 北京航空航天大学出版社 2009 年 3 月

第 12 章 Qt 2d 绘图目录

[12.1 二 D 绘图基础](#)

[12.2 绘制直线与 QLineF 类](#)

[12.3 绘制矩形与 QRectF 类](#)

[12.4 绘制椭圆、弧、弦、扇形、圆角矩形](#)

[12.5 绘制点、折线、多边形（QPolygonF 类）](#)

[12.6 QPainterPath 类（路径）](#)

[12.7 绘制文本](#)

[12.8 QPen 类（画笔）](#)

[12.9 QBrush 类（画刷）与渐变（QGradient 类及其子类）](#)

[12.10 填充](#)

[12.11 裁剪区域（QRegion 类）](#)

[12.12 坐标变换（QTransform 类）](#)

[12.13 绘制图像（QImage、QPixmap、QBitmap）](#)

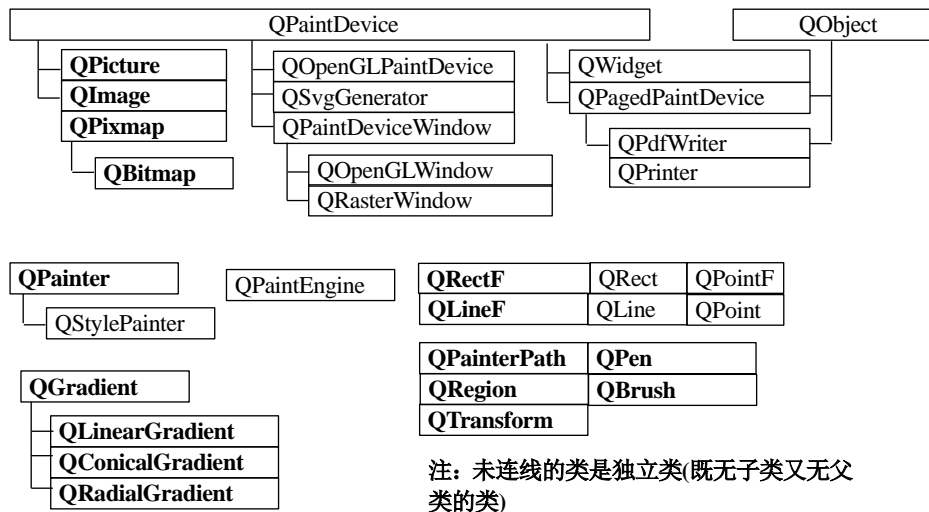
[12.14 抗锯齿和图像合成](#)

第 12 部分 Qt 2D 绘图

注意：本程序都假设读者在 `pro` 文件中已添加了正确的 `QT+=widgets` 语句，文中不再重复累述添加此语句。

本文注重讲解原理，因此使用的是手写的 Qt 程序。

本章讲解的类及继承关系如下图所示(仅讲解粗体字的类)



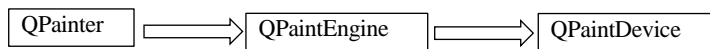
2D 绘图系统

Qt 图形系统有 3 个部分，2D 绘图、图形/视图框架、3D 绘图，限于篇幅本文仅介绍 2D 绘图。本文会遇到 `QPoint` 和 `QPointF` 之类的类，其中 `QPoint` 是点的整型版本，而 `QPointF` 是点的浮点型版本，除此之外他们几乎没有差别。其余类也是类似的，比如 `QRect` 和 `QRectF` 等。

12.1 2D 绘图基础

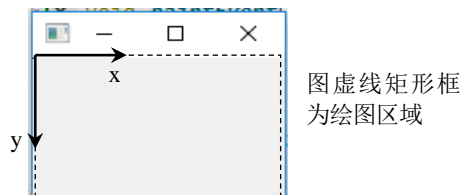
- 1、2D 绘图主要是 QPainter、QPaintDevice、QPainterEngine 三个类，
- 2、主要类的作用(其关系见图示)
 - QPainter(绘制器)是用来执行绘图的操作，用于描述需要绘制的图形，比如需要绘制线、矩形、圆形等。
 - QPaintDevice(绘图设备)是抽象出来的需要绘制的绘图设备，即可以在什么上面绘制图形，比如可在 QWidget、QImage、QPrinter 等上面绘图，

- QPainter(绘图引擎)提供 QPainter 绘制在不同类型设备上的接口，比如光栅绘图引擎，OpenGL 绘图引擎等。该类通常由 QPainter 和 QPaintDevice 内部使用，除非创建自定义的设备类型，否则通常不需要使用该类。



2D 绘图主要类的作用关系

- 3、QPainter 的核心功能就是绘图，该类提供了非常多的函数来完成大多数图形的绘制，可以绘制线条、圆形、文本、图像等。QPainter 可以在继承自 QPaintDevices 类的任何对象上绘制图形。QPainter 类的函数提供的功能几乎包括了所有的 2D 绘图功能，因此本文将把这些函数分散在各个小节进行讲解，而不集中讲解。
- 4、默认的绘图区域及坐标系统见下图：



- 5、使用 QPainter 绘制图形的步骤：

- ①、创建一个 QPainter 对象
- ②、调用 QPainter::begin(QPaintDevice *); 指定绘图设备并开始绘制，此步骤也可在 QPainter 的构造函数中完成。注意：每次调用 begin() 函数都会把 QPainter 的设置重置为默认值。
- ③、调用 QPainter 的成员函数绘制图形，调用 QPainter::end() 结束绘制。
- ④、注意：若绘制设备是一个 QWidget 部件，则 QPainter 只能在 paintEvent() 处理函数中使用(即，需要子类化 QWidget 部件，并重新实现该函数)。

示例：

```

void paintEvent(QPaintEvent *e){
    QPainter pr(this); //创建 pr 对象，并立即开始在部件 this 上绘制图形。
    pr.drawLine(QPoint(11,11), QPoint(111,111)); //绘制一条从点(11,11)到点(111,111)的直线
} //函数结束时，在 QPainter 的析构函数中调用 end() 函数结束绘制。
  
```

以上代码与以下代码等效。

```

void paintEvent(QPaintEvent *e){
    QPainter pr; //创建对象。
    pr.begin(this); //开始在这 this 上绘制图形
    pr.drawLine(QPoint(11,11), QPoint(111,111)); //绘制一条从点(11,11)到点(111,111)的直线
    pr.end(); //结束绘制。
}
  
```

- 6、以下为完整的绘图代码(结果见图示)，在此之后，本文通常只会列出 paintEvent() 函数内的代码，而不再列出完整的示例程序。

示例：完整的绘图代码

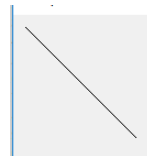
//m.h 文件的内容

```

#ifndef M_H
#define M_H
#include<QtWidgets>
class B:public QWidget{    Q_OBJECT    //需要子类化 QWidget 部件
public:
    B(QWidget *p1=0):QWidget (p1) {        }
    //需要重新实现该事件处理函数
    void paintEvent(QPaintEvent *e) {
        QPainter pr (this);
        pr.drawLine(QPoint(11,11),QPoint(111,111)); } //绘制一直线
};
#endif // M_H

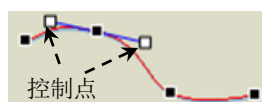
//m.cpp 文件内容
#include "m.h"
int main(int argc, char *argv[]){    QApplication app(argc,argv);
    B w;    w.resize(444,333);    w.show();    return app.exec(); }

```



7、贝塞尔(Bezier)曲线简介

- 1)、贝塞尔曲线是应用于二维图形程序的数学曲线，在计算机中有着重要的应用，比如绘图工具上常用的钢笔工具、TrueType 字体(使用二次贝塞尔曲线描述其轮廓)等。
- 2)、贝塞尔曲线的特点是易于绘制和控制，通过调节贝塞尔曲线的控制点可调节曲线的形状(见图示)。



- 3)、贝塞尔曲线是由一系列多边折线(称为特征多边形)的各顶点(称为控制点)惟一的定义出来的曲线，在多边形的各顶点中，只有最后一点和第一点在曲线上，其余各点可用于定义曲线的形状及阶次，改变多边折线的顶点位置可改变曲线的形状。
- 4)、数学公式：贝塞尔曲线的数学原理需要在起点和终点之间构建插值多项式的混合函数，通常由 $n+1$ 个顶点定义一个 n 次多项式，贝塞尔曲线上各点坐标的插值公式为

$$B(t) = \sum_{i=0}^n C_n^i P_i (1-t)^{n-i} t^i$$

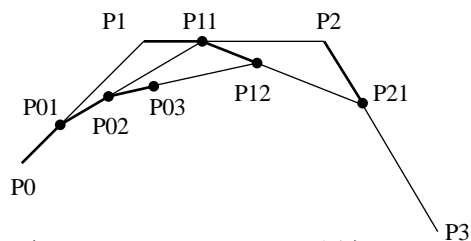
$$= C_0^n P_0 (1-t)^n t^0 + C_1^n P_1 (1-t)^{n-1} t^1 + \dots + C_{n-1}^n P_{n-1} (1-t)^1 t^{n-1} + C_n^n P_n (1-t)^0 t^n, \quad t \in [0,1]$$

- 5)、通常使用的是二次和三次的贝塞尔曲线，也就是 $n=2$ 和 $n=3$ 时的贝塞尔曲线，
 一次贝塞尔曲线公式： $B(t) = (1-t) P_0 + t P_1, t \in [0,1]$;
 二次贝塞尔曲线公式： $B(t) = (1-t)^2 P_0 + 2t(1-t)P_1 + t^2 P_2, t \in [0,1]$;
 三次贝塞尔曲线公式： $B(t) = (1-t)^3 P_0 + 3t(1-t)^2 P_1 + 3t^2(1-t)P_2 + t^3 P_3, t \in [0,1]$;

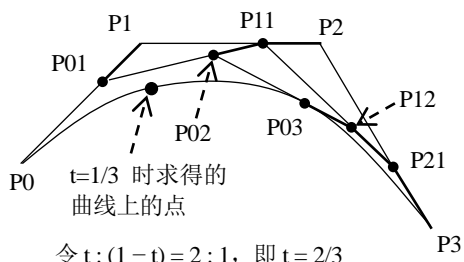
- 6)、贝塞尔曲线的绘制方法简介(其数学原理详见《计算机图形学》相关课程)

任意指定 t 值，得到 t 与 $1-t$ 的比值，即 $t:(1-t)$ ；然后依次对特征多边形的每一条边按照该比例进行分割，求得一分点，然后再连接各分点继续以上操作，又得到一系统分点，如此重复下去，对于 n 次贝塞尔曲线，第 n 次的分割点便是贝塞尔曲线上的点 $B(t)$ 。

下面以 3 次贝塞尔曲线为例进行讲解



令 $t : (1 - t) = 1 : 2$, 即 $t = 1/3$, 图中 P03 为最终求得的贝塞尔曲线上的点



令 $t : (1 - t) = 2 : 1$, 即 $t = 2/3$

说明: P1 和 P2 为控制点, P0 和 P3 是起点和终点, P01 第 2 个数字 1 表示分割的次数。

步骤: 以左图为例(右图类似), 依次求出 P0P1, P1P2, P2P3 线段上比例为 1:2 的点 P01、P11、P21, 然后依次连接 P01、P11 和 P21 得到两个分线段, 然后再在分线段上进行以上操作, 得到 P02、P12 两点, 连接两点又得到一分线段, 在该线段上找到比例为 1:2 的点 P03, 该点即为 $t=1/3$ 时贝塞尔曲线上的点。

令 $t=0 \sim 1$ 之间的数, 依次按以上方法作图, 便可得到贝塞尔曲线上的所有点。

作图法绘制三次贝塞尔曲线

12.2 绘制直线与 QLineF 类

一、绘制直线

1、需使用如下函数

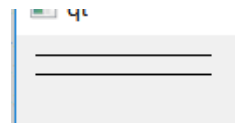
QPainter 类中绘制直线的函数

注：以下函数都省略了返回类型 void

<code>drawLine(const QLineF &line)</code>	根据 line 绘制一条直线
<code>drawLine(const QLine &line)</code>	
<code>drawLine(int x1, int y1, int x2, int y2)</code>	绘制从点(x1,y1)或 p1 到(x2,y2)或 p2 的一条直线
<code>drawLine(const QPoint &p1, const QPoint &p2)</code>	
<code>drawLine(const QPointF &p1, const QPointF &p2)</code>	
<code>drawLines(const QLine *lines, int lineCount)</code>	一次绘制多条直线。
<code>drawLines(const QLineF *lines, int lineCount)</code>	1、lines 和 pointPairs 需要数组实参，lineCount 表示需要绘制直线的数量。 2、因为绘制直线需要指定两点，因此数组的元素数至少应为 lineCount*2。 3、若为 QVector 指定奇数个点，则最后一个点会被忽略。
<code>drawLines(const QPoint *pointPairs, int lineCount)</code>	
<code>drawLines(const QPointF *pointPairs, int lineCount)</code>	
<code>drawLines(const QVector<QLine> &lines)</code>	
<code>drawLines(const QVector<QLineF> &lines)</code>	
<code>drawLines(const QVector<QPoint> &pointPairs)</code>	
<code>drawLines(const QVector<QPointF> &pointPairs)</code>	

示例：绘制多条直线(结果如右图)

```
QPainter pr(this);
QPoint p1(11, 11);    QPoint p2(111, 11);
QPoint p3(11, 22);    QPoint p4(111, 22);
QPoint p[]={p1, p2, p3, p4};    pr.drawLines(p, 2);
//或使用以下向量形式指定的相同结果的代码
//QVector<QPoint> v;        v<<p1<<p2<<p3<<p4;
//pr.drawLines(v);
//或使用以下使用 QLineF 类的相同代码
//QLineF n1(11, 11, 111, 11), n2(11, 22, 111, 22);
//QVector<QLineF> v1;    v1.append(n1); v1.append(n2);
//pr.drawLines(v1);
```



一次绘制了两条直线

2、QLine 和 QLineF 类

QLine 是整型版本，成员函数较少，QLineF 是精度更高的浮点型版本，本文以 QLineF 类进行讲解。

QLineF 类提供了一个二维向量，使用 QLineF 类绘制直线可以利用该类中的成员函数方便的对线条的属性进行一些设置，比如可使用 setAngle() 设置线条的角度，使用 angle() 可获取线条的角度等。下面为该类中的函数

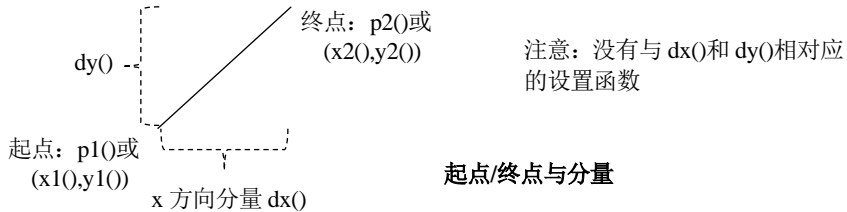
1)、QLineF(); QLineF(const QLine &line)

`QLineF(const QPointF &p1, const QPointF &p2);` //构造从 p1 到 p2 的线

`QLineF(qreal x1, qreal y1, qreal x2, qreal y2);` //构造从(x1,y1)到(x2,y2)的线

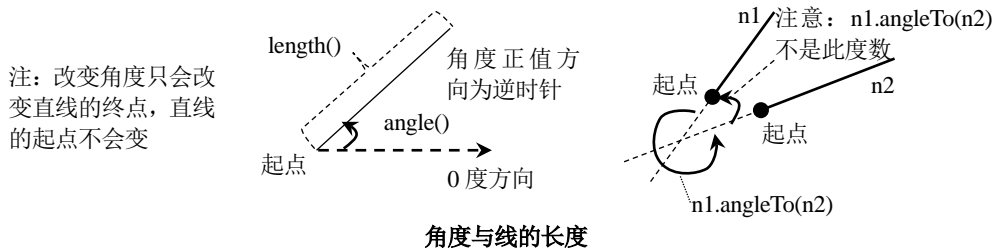
2)、起点/终点与分量

QPointF **p1**() const; QPointF **p2**() const; qreal **dx**() const; qreal **dy**() const;
 qreal **x1**() const; qreal **y1**() const; qreal **x2**() const; qreal **y2**() const
 void **setP1**(const QPointF &p1); void **setP2**(const QPointF &p2);
 void **setLine**(qreal x1, qreal y1, qreal x2, qreal y2); void **setPoints**(const QPointF &p1, const QPointF &p2)

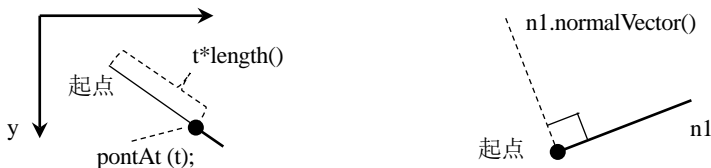


3)、角度与线的长度

qreal **angle**() const; //返回直线的角度, 范围为 0~359 度
 void **setAngle**(qreal **angle**)
 qreal **length**() const; void **setLength**(qreal **length**)
 qreal **angleTo**(const QLineF &line) const; //该线需要旋转多少度才与线 line 的度数相同(注意, 这不是指的两线相交的角度)



- 4)、QPointF **pointAt**(qreal t) const; //返回位于直线上 t*length()处的点。
 QLineF **normalVector**() const; //返回与此线有相同起点和长度并与该线垂直的线
 QLineF **unitVector**() const; //返回与此线有相同起点和方向, 长度为 1.0 的线, 即单位向量。



5)、相交

IntersectType **intersect**(const QLineF &line, QPointF *intersectionPoint) const

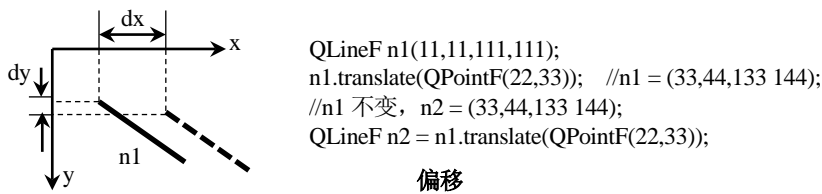
该线是否与线 line 相交，交点保存于 intersectionPoint 指针中，若线条平行，则交点不确定。枚举 IntersectType 表示相交的类型，见下表。

QLineF::IntersectType 枚举(无标志)		
作用：两线相交的类型		
成员	值	说明
QLineF::NoIntersection	0	两线不相交(即平行)。
QLineF::UnboundedIntersection	2	两线相交，但交点不位于两条线之上
QLineF::BoundedIntersection	1	两线相交，交点位于两条线之上。

6)、偏移

```
void translate(const QPointF &offset);
void translate(qreal dx, qreal dy);
QLineF translated(const QPointF &offset) const
QLineF translated(qreal dx, qreal dy) const
```

把直线沿 x 轴移动距离 dx，沿 y 轴移动距离 dy，负值向反方向移动。注意：移动的是距离而不是位置(虽然 offset 是以点的形式表示的)，其中 translate() 会改变原直线，而 translated() 不会改变原直线，但会返回一个偏移后的副本。原理见图



7)、其他

```
QPointF center() const; //返回该线的占点，相当于 0.5*p1() + 0.5*p2(); qt5.8
QLine toLine() const; //返回此线的整数副本(注意：相应数值会被四舍五入为最近的整数)
bool isNull() const; //若线未设置有效的起点和终点，则返回 true，否则返回 false。
static QLineF fromPolar(qreal length, qreal angle); //静态的
//返回长度为 length，角度为 angle 的线，该线起点位于原点。
bool operator!=(const QLineF &line) const;
bool operator==(const QLineF &line) const;
```

3、绘制图形时需要注意变量的作用域的问题，比如

```
class B:public QWidget{    Q_OBJECT
public: QLineF n1;
    B(QWidget *p1=0):QWidget(p1){    n1=QLineF(11, 11, 111, 111);    } //初始化直线
    void paintEvent(QPaintEvent *e) {
        QPainter pr(this);
```

/*以下代码会使直线 n1 被不断地偏移，因为只要 QWidget 一旦更新界面(比如调整大小，移动等)就会调用一次 paintEvent() 函数，每调用一次就会偏移一次直线，这会使直接被不断地偏移。*/

```
n1.translate(QPointF(22,33));  
pr.drawLine(n1);  
}};
```

可把直线偏移的代码放在槽函数中，以使在需要时调用槽函数来偏移，比如

```
void paintEvent(QPaintEvent *e) {  
    QPainter pr(this);  
    /*在该函数中对 n1 赋值，会导致每调用一次 paintEvent() 都会对 n1 重新赋值。因此要使槽函数  
    中的 n1 能被偏移，就不能在该函数中对 n1 重新赋值。*/  
    //n1=QLineF(11, 11, 111, 111);  
    pr.drawLine(n1);  
}  
public slots:  
void f1() {    n1.translate(QPointF(22, 33));    //偏移 n1。  
              update(); } //偏移后记得更新界面，这样才会使窗口重绘。
```

12.3 绘制矩形与 QRectF 类

1、需要使用到的 QPainter 类中的函数

QPainter 类中绘制矩形的函数

注：以下函数都省略了返回类型 void

<code>drawRect(int x, int y, int width, int height)</code>	绘制一个矩形，其中(x,y)是左上角坐标，width 是宽度，height 是高度。
<code>drawRect(const QRect &rectangle)</code>	
<code>drawRect(const QRectF &rectangle)</code>	
<code>drawRects(const QRect *rectangles, int rectCount)</code>	一次绘制多个矩形。rectangles 需要数组实参，rectCount 表示需要绘制矩形的数量。其使用方法类似 QLineF，详见 QLineF 类用法。
<code>drawRects(const QRectF *rectangles, int rectCount)</code>	
<code>drawRects(const QVector<QRect> &rectangles)</code>	
<code>drawRects(const QVector<QRectF> &rectangles)</code>	

2、QRect 和 QRectF 类

QRect 是整型版本，QRectF 是精度更高的浮点型版本，本文以 QRectF 类进行讲解。

QRectF 类对矩形进行了描述，下面为该类中的函数

3、对于 QRect，由于历史原因，bottom()和 right()函数返回的值并不是真正的矩形的右下角，right()返回的值与 left()+width()-1 相同，bottom()返回的值与 top()+height()-1 相同，同理 bottomRight()、topRight()、bottomLeft()函数与 bottom()和 right()类似，详见下图。为避免这种情形，建议使用 QRectF 而不是 QRect。

4、QRectF 类的构造函数

1)、QRectF()

`QRectF(const QPointF &topLeft, const QSizeF &size);` //由左上角的点 topLeft 和大小构造矩形

`QRectF(const QPointF &topLeft, const QPointF &bottomRight);` //由左上角和右下角的点构造矩形

`QRectF(qreal x, qreal y, qreal width, qreal height)`

`QRectF(const QRect &rectangle)`

5、获取和设置位置和大小的函数(见下表)

QRectF 类中的位置和大小函数

省略说明：

1、读取函数的返回类型与设置函数的形参类型是相同的，并且省略了读取函数小括号后的 const。

2、移动函数和设置函数都省略了返回类型 void，并且形参类型 QPointF 和 QSizeF 前的 const 也省略了。

函数说明：

1、形参 x 指的是 x 轴方向的 x 坐标位置，y 类似。

2、pos 是指的点的坐标位置。

3、height 和 width 是指的矩形的高度和宽高，注意，这不是坐标位置。QSizeF 也是指的宽度和高度。

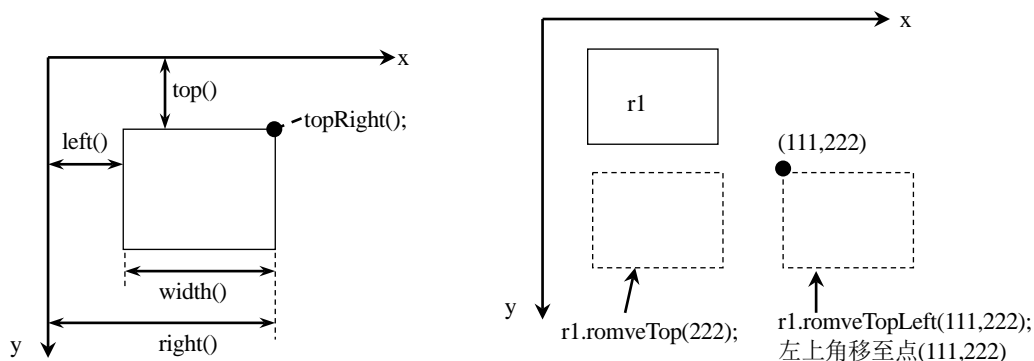
4、比如 top()返回的是顶部的 y 轴坐标位置，setTop(11);表示设置矩形顶部的位置到 y 轴坐标位置 11 处，底部保持不变，moveTop(11);表示垂直移动矩形，以使顶部位于 y 轴坐标位置 11 处。详见图示

读取函数	移动函数	设置函数
<code>top()</code> 、 <code>y()</code>	<code>moveTop(qreal y);</code>	<code>setY(qreal y);</code> <code>setTop(qreal y);</code> //从顶部改变高度
<code>bottom()</code>	<code>moveBottom(qreal y);</code>	<code>setBottom(qreal y);</code> //从底部改变高度
<code>left()</code> 、 <code>x()</code>	<code>moveLeft(qreal x);</code>	<code>setLeft(qreal x);</code>

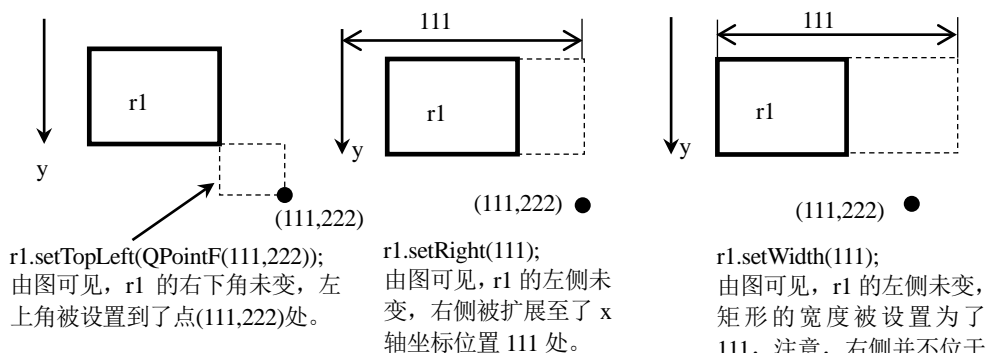
		<code>setX(qreal x);</code> //从左侧改变宽度
<code>right()</code>	<code>moveRight(qreal x);</code>	<code>setRight(qreal x);</code> //从右侧改变宽度
<code>topLeft()</code>	<code>moveTo(qreal x, qreal y);</code> <code>moveTo(QPointF &pos);</code> <code>moveTopLeft(QPointF &pos);</code>	<code>setTopLeft(QPointF &pos);</code> //从左上角改变大小
<code>topRight()</code>	<code>moveTopRight(QPointF &pos);</code>	<code>setTopRight(QPointF &pos);</code> //从右上角改变大小
<code>bottomLeft()</code>	<code>moveBottomLeft(QPointF &pos);</code>	<code>setBottomLeft(QPointF &pos);</code> //从左下角改变大小
<code>bottomRight()</code>	<code>moveBottomRight(QPointF &pos);</code>	<code>setBottomRight(QPointF &pos);</code>
<code>height()</code>		<code>setHeight(qreal height);</code> //从底部改变高度
<code>width()</code>		<code>setWidth(qreal width);</code> //从右侧改变宽度
<code>size()</code>		<code>setSize(QSizeF &size);</code> //设置大小, 左上角不变
<code>center()</code>	<code>moveCenter(QPointF &pos);</code>	//中心位置

其余位置和大小函数

- 1)、void `setRect`(qreal x, qreal y, qreal **width**, qreal **height**);
设置矩形左上角坐标为(x,y), 宽为 width, 高为 height
- 2)、void `setCoords`(qreal **x1**, qreal **y1**, qreal **x2**, qreal **y2**);
设置矩形左上角的坐标为(x1,y1), 右下角坐标为(x2,y2);
- 3)、void `getRect`(qreal *x, qreal *y, qreal ***width**, qreal ***height**) const;
把矩形的左上角坐标提取到(x,y)中, 宽度提取到 width, 高度提取到 height 中。
- 4)、void `getCoords`(qreal ***x1**, qreal ***y1**, qreal ***x2**, qreal ***y2**) const;
把矩形的左上角坐标提取到(x1,y1)中, 左上角坐标提取到(x2,y2)中。



读取函数和移动函数原理



设置函数原理

6、判断矩形及点

1)、bool **contains**(const QPointF &point) const; 若点 point 位于矩形内(含边缘), 则返回 true。

bool **contains**(const QRectF &rectangle) const; 若矩形 rectangle 位于该矩形内, 则返回 true。

bool **contains**(qreal x, qreal y) const; //若点(x,y)在矩形内(含边缘), 则返回 true。

2)、bool **isEmpty**() const; //矩形是否为空, 详见下文。

bool **isNull**() const; //矩形是否不存在, 详见下文。

bool **isValid**() const; //矩形是否无效, 详见下文。

若 宽度<=0 或 高度<=0, 则 isEmpty()返回 true。

若 宽度=0 且 高度 = 0, 则 isNull()返回 true。

空矩形意味着是无效的, 即 isEmpty()==!isValid();

注意: 宽度或高宽 < 0 不代表该矩形不存在, 若宽度或高度小于 0, 矩形会向相反的方向绘制。

示例:

QRectF r1(11,11, 0, 111); //r1.isEmpty() = true; r1.isNull() = false; r1.isValid() = false。

QRectF r2(11,11, 0, 0); //r2.isEmpty() = true; r2.isNull() = true; r2.isValid() = false。

QRectF r3(11,11, -111, -111); //r3.isEmpty() = true; r3.isNull() = false; r3.isValid() = false。

QRectF r4(11,11, 111, 111); //r4.isEmpty() = false; r4.isNull() = false; r4.isValid() = true。

3)、QRectF **normalized**() const;

返回一个规范化矩形(即有效矩形), 若 width() < 0, 则交换左右角, 若 height()<0, 则交换上下角。比如 QRectF r1(11,11, -111,111); QRectF r2 = r1.normalized(); //r2 = (11,11,111,111);

7、平移矩形

1)、void **translate**(qreal dx, qreal dy); void **translate**(const QPointF &offset);

QRectF **translated**(qreal dx, qreal dy) const; QRectF **translated**(const QPointF &offset) const;

把矩形沿 x 轴移动距离 dx, 沿 y 轴移动距离 dy, 负值向反方向移动。注意: 移动的是距离, 而不是位置, 其中 translate()会改变原矩形, 而 translated()不会改变原矩形, 但会返回一个偏移后的副本。以上函数的原理与 QLineF 类中的相应函数是相同的, 详见 QLineF 类。

2)、QRectF **transposed**() const;

返回交换该矩形宽度和高度后的副本。比如

QRectF r1(11,33, 77,99);

QRectF r2=r1.transposed(); //r1 不变, r2 是 r1 交换高宽后的结果, 即 r2=(11,33,99,77);

8、调整矩形

1)、void **adjust**(qreal dx1, qreal dy1, qreal dx2, qreal dy2)

QRectF **adjusted**(qreal dx1, qreal dy1, qreal dx2, qreal dy2) const

把矩形的左上角坐标加上(dx1, dy1);右下角坐标加上(dx2, dy2); 其中 adjust()会改变原矩形的坐标值, adjusted()不会改变原矩形的坐标值, 但会返回一个新矩形, 示例如下:

QRectF r1(QPointF(11,22), QPointF(111,122));

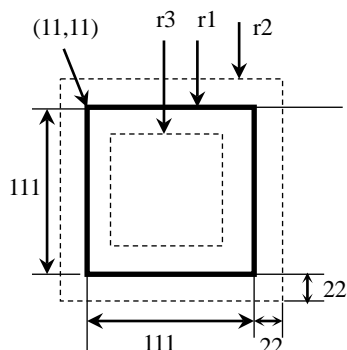
r1.adjust(33,33,155,155); //r1 左上角坐标为(44,55), 右上角坐标为(266,277);

QRectF r2=r1.adjusted(33,33,155,155); //r1 不变, r2 左上角坐标为(44,55), 右上角坐标为(266,277);

2)、QRectF **marginsAdded**(const QMarginsF &margins) const; //qt5.3

QRectF **marginsRemoved**(const QMarginsF &*margins*) const; //qt5.3

返回一个在现有矩形上增加或减小了边距 *margins* 的新矩形(即调整矩形的宽度和高度)



```
QRectF r1(33,33,111,111);
//r2=(11,11,155,155);
QRectF r2=r1.marginsAdded(QMarginsF(22,22,22,22));
//r3=(55,55,67,67);
QRectF r3=r1.marginsRemoved(QMarginsF(22,22,22,22));
```

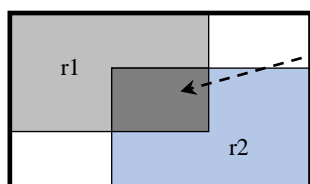
9、矩形的交集和并集

1)、QRectF **intersected**(const QRectF &*rectangle*) const

返回该矩形与矩形 *rectangle* 的交集(即两矩形的重叠区域)，注意：*r1.intersected(r2)* 相当于 *r1&r2*。

2)、bool **intersects**(const QRectF &*rectangle*) const; //若该矩形与矩形 *rectangle* 相交，则返回 true。

3)、QRectF **united**(const QRectF &*rectangle*) const; //返回该矩形与矩形 *rectangle* 的边界矩形。



QRectF r3=r1.intersected(r2);

QRectF r4=r1.united(r2);
//r4 为外围粗线框所围矩形。

10、浮点型与整型间的转换及其他

QRect **toAlignedRect**() const; //返回完全包含该矩形的最小 QRect(整数)矩形。

QRect **toRect**() const; //根据该矩形返回一个 QRect(整数)矩形，其值将被四舍五入到最接近的整数。

CGRect **toCGRect**() const; //从 QRectF 创建 CGRect， qt5.8

static **QRectF** fromCGRect(CGRect *rect*); //从 rect 创建一个 QRectF， 静态的， qt5.8

11、重载的操作符函数

QRectF operator&(const QRectF &*rectangle*) const; //返回两矩形的交集。

QRectF &operator&=(const QRectF &*rectangle*);

QRectF &operator+=(const QMarginsF &*margins*); //返回增加边距后的矩形

QRectF &operator-=(const QMarginsF &*margins*); //返回缩小边距后的矩形。

QRectF operator|(const QRectF &*rectangle*) const; //返回边界矩形(并集)。

QRectF &operator|=(const QRectF &*rectangle*)

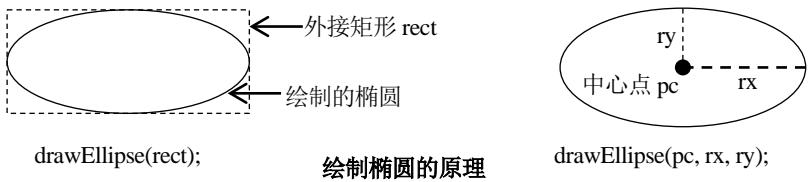
12.4 绘制椭圆、弧、弦、扇形、圆角矩形

1、需要使用到的 QPainter 类中的函数

QPainter 类中绘制椭圆、弧、弦的函数	
注：以下函数都省略了返回类型 void 以下的弧、弦、扇形是指椭圆弧、椭圆弦、椭圆扇形，圆是椭圆的一个特例。	
<code>drawEllipse(const QRectF &rectangle)</code> <code>drawEllipse(const QRect &rectangle)</code> <code>drawEllipse(int x, int y, int width, int height)</code> <code>drawEllipse(const QPointF &center, qreal rx, qreal ry)</code> <code>drawEllipse(const QPoint &center, int rx, int ry)</code>	绘制椭圆
<code>drawArc(const QRectF &rectangle, int startAngle, int spanAngle)</code> <code>drawArc(const QRect &rectangle, int startAngle, int spanAngle)</code> <code>drawArc(int x, int y, int width, int height, int startAngle, int spanAngle)</code>	绘制弧，注意：startAngle, spanAngle 为指定角度的 1/16, 角度逆时针方向为正(详见正文)
<code>drawChord(const QRectF &rectangle, int startAngle, int spanAngle)</code> <code>drawChord(int x, int y, int width, int height, int startAngle, int spanAngle)</code> <code>drawChord(const QRect &rectangle, int startAngle, int spanAngle)</code>	绘制弦，其他同上。
<code>drawPie(const QRectF &rectangle, int startAngle, int spanAngle)</code> <code>drawPie(int x, int y, int width, int height, int startAngle, int spanAngle)</code> <code>drawPie(const QRect &rectangle, int startAngle, int spanAngle)</code>	绘制扇形(饼形)。其他同上
<code>drawRoundedRect(const QRectF &rect, qreal xRadius, qreal yRadius, Qt::SizeMode mode=Qt::AbsoluteSize)</code> <code>drawRoundedRect(int x, int y, int w, int h, qreal xRadius, qreal yRadius, Qt::SizeMode mode=Qt::AbsoluteSize)</code> <code>drawRoundedRect(const QRect &rect, qreal xRadius, qreal yRadius, Qt::SizeMode mode=Qt::AbsoluteSize)</code> 以上函数用于绘制带圆角的矩形(原理见正文)	

2、绘制椭圆的方法有

绘制给定矩形的内接椭圆和根据中心点与椭圆 x 方向和 y 方向的半径绘制，原理见下图



3、绘制弧、弦、扇形的原理：

- 1)、弧是椭圆上的一段曲线，因此其绘制方法就是首先绘制一个椭圆，然后指定一段从起点到结束点的曲线作为弧。
- 2)、弧、弦、扇形的原理是相同的，只是形式不同，下图是他们的样式



- 2)、本文以弧为例讲解其绘制的原理，下面我们看看画弧的函数，原型如下：

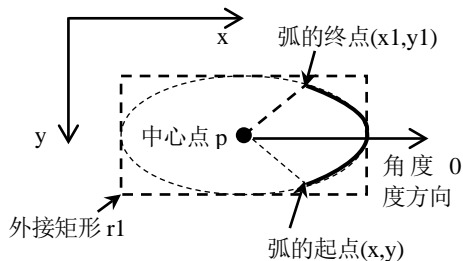
```
drawArc(const QRectF &rectangle, int startAngle, int spanAngle);
```


- 第 1 个参数 `rectangle` 用于指定该矩形的内接椭圆。
- `startAngle` 用于指定起始角，`spanAngle` 表示跨越的度数。注意：`startAngle` 和 `spanAngle` 是用于以下椭圆参数方程中的角 θ 的，

$$x = a \cos \theta, y = b \sin \theta; \quad (a \text{ 表示椭圆 } x \text{ 方向的半轴长, } b \text{ 表示椭圆 } y \text{ 方向的半轴长})$$

也就是说 `startAngle` 并不是指的弧的中心点与起点所连直线与 x 方向的夹角，`startAngle` 是用于计算弧的起点坐标的，同理 `spanAngle` 和 `startAngle` 共同用于计算弧的终点坐标。

- 下面以图示进行讲解



假设起始角度为 δ ，跨越角度为 η ，

则弧的起点坐标计算如下：

$$x = (r1.width() / 2) * \cos \delta + p.x();$$

$$y = p.y() - (r1.height() / 2) * \sin \delta;$$

则弧的终点坐标计算如下：

$$x1 = (r1.width() / 2) * \cos(\delta + \eta) + p.x();$$

$$y1 = p.y() - (r1.height() / 2) * \sin(\delta + \eta);$$

绘制弧的原理

示例：弧的绘制

本示例绘制一条跟踪弧起点和一条跟踪弧终点直线。

//m.h 文件的内容

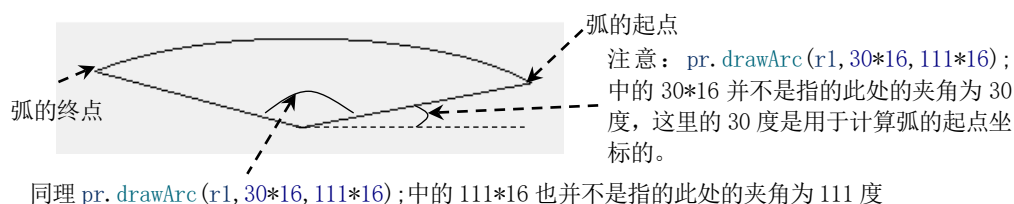
```
#ifndef M_H
#define M_H
#include<QtWidgets>
#include<math.h> //C++数学函数
class B:public QWidget{ Q_OBJECT
public:
    B(QWidget *p1=0):QWidget(p1) {}
    void paintEvent(QPaintEvent *e) {
        QPainter pr(this);
        QRectF r1=QRectF(10,10,333,111); //椭圆的外接矩形
        QPointF p1=r1.center(); //矩形(即椭圆)中心点。
        pr.drawArc(r1,30*16,111*16); //绘制一段弧。注意：需使用实际度数乘以16。
        qreal pi=3.1415926; //指定 pi
        qreal w=pi/180;
//绘制跟踪弧起点的直线
        qreal x=(r1.width()/2)*cos(30*w)+p1.x();//注：三角函数是以弧度形式指定的，而不是角度
        qreal y=p1.y()-(r1.height()/2)*sin(30*w); //公式原理详见前文图示。
        QPointF p2=QPointF(x,y); //弧起点的坐标
        pr.drawLine(p1,p2); //绘制一条从椭圆中心到弧起点的坐标
//绘制跟踪弧终点的直线
        qreal x1=(r1.width()/2)*cos(141*w)+p1.x();
        qreal y1=p1.y()-(r1.height()/2)*sin(141*w);
        QPointF p3=QPointF(x1,y1); //弧终点的坐标
        pr.drawLine(p1,p3); //绘制一条从椭圆中心到弧终点的坐标
    }
};
```

```
#endif // M_H
```

//m.cpp 文件内容

```
#include "m.h"
int main(int argc, char *argv[]) {    QApplication app(argc,argv);
    B w;    w.resize(444,333);    w.show();    return app.exec(); }
```

运行结果及说明



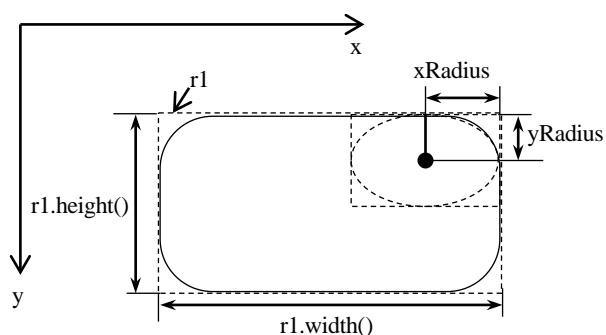
4、绘制带圆角的矩形的原理,圆角矩形的圆角是使用椭圆的四个象限分别对矩形的四个角进行圆角的。下面为绘制带圆角的矩形的函数原型

```
void drawRoundedRect(const QRectF &rect, qreal xRadius, qreal yRadius,
                    Qt::SizeMode mode=Qt::AbsoluteSize)
```

`rect` 表示需要被圆角的矩形, `xRadius` 和 `yRadius` 分别表示椭圆 `x` 方向的半轴长度和 `y` 方向的半轴长度, `mode` 用于指定 `xRadius` 和 `yRadius` 的取值方式, 取值如下:

- `Qt::AbsoluteSize`: 使用绝对值指定 `xRadius` 和 `yRadius` 的大小。
- `Qt::RelativeSize`: 使用相对于矩形 `rect` 的宽和高的一半的百分比来指定 `xRadius` 和 `yRadius` 的大小, 此时值的范围为 0.0 ~ 100.0。

下面以图示进行讲解



由图可见, 只要计算出 `xRadius` 和 `yRadius`, 就能计算出被圆角的矩形每个角上的椭圆的大小及位置。示例:

```
QRectF r1(11,11,222,111);
pr.drawRoundedRect(r1, 55,22);
则 xRadius =55, yRadius=22;
pr.drawRoundedRect(r1, 55,22,
                    Qt::RelativeSize);
则 xRadius = (222/2) *0.55 = 110.45 ,
    yRadius = (111/ 2) *0.22 = 12.21;
```

12.5 绘制点、折线、多边形(QPolygonF 类)

1、需要使用到的 QPainter 类中的函数

QPainter 类中绘制椭圆、弧、弦的函数

注：以下函数都省略了返回类型 void

多边形会隐式将最后一点连接到第一点，而折线则不会。

drawPoint (const QPointF & <i>position</i>) drawPoint (const QPoint & <i>position</i>) drawPoint (int x, int y)	绘制点(很简单, 略)
drawPoints (const QPointF * <i>points</i> , int <i>pointCount</i>) drawPoints (const QPoint * <i>points</i> , int <i>pointCount</i>)	绘制多个点, <i>points</i> 需要数组实参, <i>pointCount</i> 指示需要绘制的点的数量。
drawPoints (const QPolygonF & <i>points</i>) drawPoints (const QPolygon & <i>points</i>)	绘制多个点, QPolygonF、QPolygon 见正文。
drawConvexPolygon (const QPointF * <i>points</i> , int <i>pointCount</i>) drawConvexPolygon (const QPoint * <i>points</i> , int <i>pointCount</i>) drawConvexPolygon (const QPolygonF & <i>polygon</i>) drawConvexPolygon (const QPolygon & <i>polygon</i>)	绘制凸多边形, <i>points</i> 需要数组实参, <i>pointCount</i> 指示需要绘制的数量。若绘制的多边形不是凸的, 即包含一个至少大于 180 度的角, 则结果是不确定的。
drawPolyline (const QPointF * <i>points</i> , int <i>pointCount</i>) drawPolyline (const QPoint * <i>points</i> , int <i>pointCount</i>) drawPolyline (const QPolygonF & <i>ploy</i>) drawPolyline (const QPolygon & <i>ploy</i>)	绘制多段线(即折线), <i>points</i> 需要数组实参, <i>pointCount</i> 指示需要绘制的数量。
drawPolygon (const QPointF * <i>points</i> , int <i>pointCount</i> , Qt::FillRule <i>fillRule</i> = Qt::OddEvenFill) drawPolygon (const QPoint * <i>points</i> , int <i>pointCount</i> , Qt::FillRule <i>fillRule</i> = Qt::OddEvenFill) drawPolygon (const QPolygonF & <i>points</i> , Qt::FillRule <i>fillRule</i> = Qt::OddEvenFill) drawPolygon (const QPolygon & <i>points</i> , Qt::FillRule <i>fillRule</i> = Qt::OddEvenFill)	
以上函数用于绘制多边形, 枚举 Qt::FillRule 见 QPainterPath 类, 主本影响填充背景, 此处暂不讲解。	

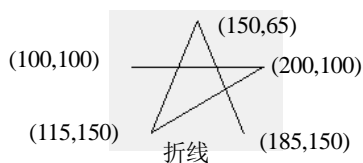
2、绘制折线和多边形有两种方法

一种是是, 根据提供的一系列坐标点, 和折线的段数, 依次把提供的坐标点相连, 从而绘制成指定段数的折线。一系列的坐标点可以通过数组提供。

一种是提供一个 QPolygonF 或 QPolygon 绘制一个折线或多边形, 其实这两个类的本质也就是一系列的点, 只是使用这两个类可提供一些方便使用的函数。

示例:

```
void paintEvent(QPaintEvent *e){
    QPainter pr(this);
    QPointF p[] = {QPointF(100,100), QPointF(200,100), QPointF(115,150), QPointF(150,65),QPointF(185,150) };
    pr.drawPolygon(p, 5);    pr.drawPolyline(p, 5);    }
```



注: drawPolygon()和 drawPolyline()应分别绘制, 否则两个图形会重合在一起。

由图可见, 折线第一点和最后一点未自动连接

3、QPolygonF 类

注意: QPolygonF 类继承自 QVector, 该类可用于快速的提供一系列点的集合, 该类还提供了一些函数可以方便的处理多边形, 最简单的添加点的方法就是使用<<运算符, 比如:

```
QPolygonF p; p<<QPointF(11,11)<<QPointF(111,111)<<QPointF(111,222); pr.drawPolygon(p);
```

下面为该类中的函数

1)、QPolygonF() //构造函数

QPolygonF(int size)

QPolygonF(const QVector<QPointF> &points)

QPolygonF(QVector<QPointF> &&v)

QPolygonF(const QRectF &rectangle)

QPolygonF(const QPolygon &polygon)

QPolygonF(const QPolygonF &polygon)

QPolygonF(QPolygonF &&other)

2)、void translate(const QPointF &offset); //平移所有点, 原理与 QRectF 类相同。

void translate(qreal dx, qreal dy)

QPolygonF translated(const QPointF &offset) const

QPolygonF translated(qreal dx, qreal dy) const

3)、bool containsPoint(const QPointF &point, Qt::FillRule fillRule) const;

判断点 point 是否位于多边形内部, Qt::FillRule 参见 QPainterPath 类

4)、QPolygon toPolygon() const; //转换为整型版本的 QPolygon

5)、bool isClosed() const; //若多边形的起点与终点相等(此情形称为闭合), 则返回 true。

6)、void swap(QPolygonF &other);

把该多边形与多边形 other 交换。比如

```
QPolygonF p1; p1<<QPointF(11,11)<<QPointF(11,33)<<QPointF(33,55)<<QPointF(11,22);
```

```
QPolygonF p2; p2<<QPointF(22,22)<<QPointF(22,44)<<QPointF(44,77);
```

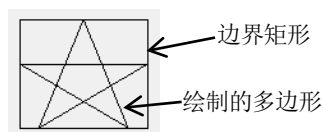
```
p1.swap(p2); //则 p1={ (22,22), (22,44), (44,77) }
```

7)、QPolygonF intersected(const QPolygonF &r) const; //返回此多边形与 r 的交集

bool intersects(const QPolygonF &p) const; //若两多边形相交, 则返回 true。

8)、QRectF boundingRect() const;

返回多边形的边界矩形, 若多边形为空, 则返回 QRectF(0,0,0,0);原理见下图

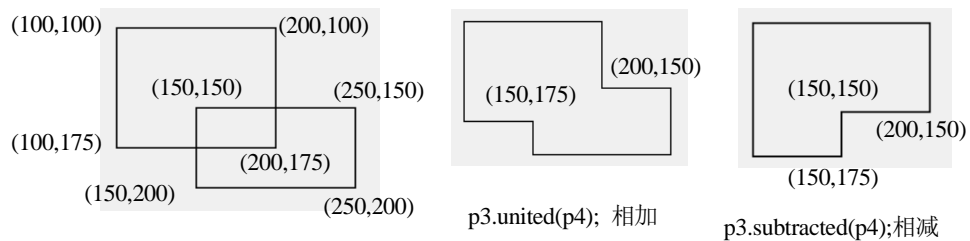


9)、QPolygonF united(const QPolygonF &r) const;

返回此多边形与 r 的并集(注意: 与 QRectF::united()原理不相同)。原理见下图

10)、QPolygonF subtracted(const QPolygonF &r) const;

返回从该多边形中减去多边形 r 后的新多边形。原理见下图



```

QPolygonF p3;
p3<<QPointF(100, 100)<<QPointF(200, 100)<<QPointF(200, 175)<<QPointF(100, 175);
QPolygonF p4;
p4<<QPointF(150, 200)<<QPointF(150, 150)<<QPointF(250, 150)<<QPointF(250, 200);
QPolygonF p5=p3.united(p4);
p5 = {(100,100), (200,100), (200,150), (250,150), (250,200), (150,200), (150,175), (100,175), (100,100)}
QPolygonF p6=p3.subtracted(p4);
p6 = {(100,100), (200,100), (200,150),(150,150), (150,175), (100,175), (100,100)}

```

12.6 QPainterPath 类(路径)

一、路径基本原理

1、需要使用到的 QPainter 类中的函数就一个，原型如下：

```
void QPainter::drawPath(const QPainterPath &path);
```

2、QPainterPath 类是一个容器，可把图形形状保存其中，需要时可再次使用，也就是说，复杂的图形只需要使用路径创建一次，然后就可以调用 QPainter::drawPath()函数多次绘制它们。一个路径就是由多个矩形、椭圆、线条等图形组成的对象，路径可以是封闭的也可以是非封闭的。

3、子路径：路径是由多个图形组成的对象，每个形状构成一个子路径。

4、图形的绘制步骤：路径始终从当前点开始绘制，下面为其步骤

1)、创建 QPainterPath 对象。

2)、使用 moveTo 把当前点移至需要绘制图形的开始位置

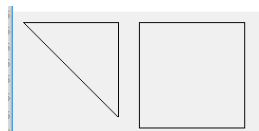
3)、使用 lineTo()、arcTo()等函数绘制直线、弧等图形，还可使用 addRect()、addEllipse()等函数把封闭子路径添加到路径。

4)、使用 QPainter::drawPath()函数绘制路径所描述的图形。

示例(注意：实际绘制时需要注意作用域问题，以下示例本着简洁原则，暂不考虑作用域的影响)：

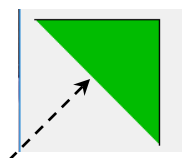
```
QPainter pr(this);
QPainterPath ph;
ph.moveTo(11, 11);    //把当前点移至(11, 11)，即从(11, 11)处开始绘制。
ph.lineTo(111, 11);   //绘制一条从(11, 11)到(111, 11)的直线
ph.lineTo(111, 111);  //绘制一条从(111, 11)到(111, 111)的直线
ph.lineTo(11, 11);    //绘制一条从(111, 111)到(11, 11)的直线，最终形成一个三角形
ph.addRect(133, 11, 111, 111); //添加一个封闭了矩形到路径。
pr.drawPath(ph);      //绘制路径中的图形。
```

绘制的图形见图



5、QPainterPath 对象可用于填充、轮廓和裁剪，也就是说即使绘制的路径不是封闭的，也会被视为是隐式关闭的，因此可被填充，比如

```
QPainter pr(this);
QBrush bs(QColor(1, 188, 1));
pr.setBrush(bs); //设置画刷，以填充路径
QPainterPath ph;
ph.moveTo(11, 11);
ph.lineTo(111, 11);
ph.lineTo(111, 111);
pr.drawPath(ph);
```

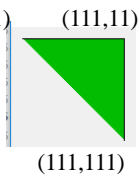


注意：此处并没有一条封闭该图形的直线，即，该图形不是封闭的

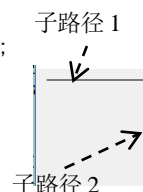
6、常用函数及基本概念

- 1)、**moveTo()**: 该函数可移动当前点, 移动当前点会启用一个新的子路径, 并隐式关闭之前的当前路径。**moveTo()**只是移动当前点, 不会绘制图形。

```
QPainter pr(this);
QBrush bs(QColor(1, 188, 1));
pr.setBrush(bs);
QPainterPath ph;
ph.moveTo(11, 11);
ph.lineTo(111, 11);
ph.lineTo(111, 111);
pr.drawPath(ph);
```



```
QPainter pr(this);
QBrush bs(QColor(1, 188, 1));
pr.setBrush(bs);
QPainterPath ph;
ph.moveTo(11, 11);
ph.lineTo(111, 11);
//增加一条 moveTo 语句
ph.moveTo(111, 11);
ph.lineTo(111, 111);
pr.drawPath(ph);
```

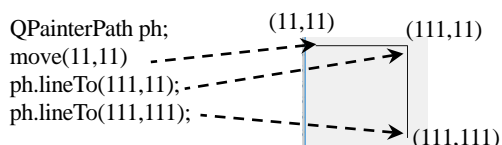


增加一条 **moveTo** 语句后, 变为两个子路径, 每个子路径都只是一条线段, 不能构成封闭(或隐式封闭)图形, 所以最终结果未被填充。

moveTo()及子路径原理

- 2)、**currentPosition()**: 该函数可获取当前点的位置, 当前点的位置始终是添加的最后一个子路径的结束位置。
- 3)、当前点的位置: 使用 **lineTo()**函数等函数绘制图形后, 会更新当前点的位置, 具体更新规则依使用的函数而不同, 使用 **lineTo()**会绘制图形后, 当前点会被更新为直线的终点, 使用 **addRect()**绘制图形后, 当前点会被更新为矩形的左上角, **addPolygon()**把当前点更新为最后一点。当前点的原理见下图

```
QPainterPath ph;
move(11,11)
ph.lineTo(111,11);
ph.lineTo(111,111);
```



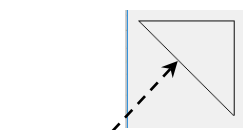
箭头指向使用相应语句后当前点的位置

当前点原理

- 4)、**closeSubpath()**: 该函数通过画一条到子路径开头的线来关闭当前子路径, 并开启一个新的子路径, 新路径当前点为(0,0); 比如:

```
QPainterPath ph;
ph.moveTo(11, 11);
ph.lineTo(111, 11);
ph.lineTo(111, 111);
//ph.closeSubPath()
```

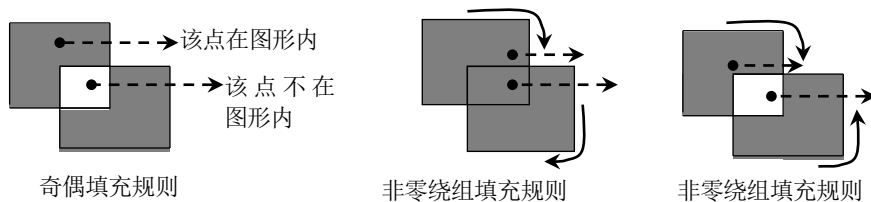
未使用 **ph.closeSubPath()** 时的样式



使用 **ph.closeSubPath()**后会绘制此条直线以封闭子路径

- 7、**填充规则(Fill Rule)**: 填充规则用于判断应该填充图形的哪个区域, 或者说用于判断某个点是否位于图形内, 有如下两种填充规则(原理见图)

- 1)、奇偶(odd even)填充规则：从该点向图形外绘制一水平直线，若该线与图形交点的个数为奇数，则表示该点位于图形之中。
- 2)、非零绕组(non zero winding)填充规则：从该点向图形外绘制一水平直线，然后确定该直线与图形每个交点处的边线的方向，比如把顺时针方向绘制的边线记为 1，把逆时针方向绘制的边线计为-1，然后把所有数值相加，若结果不为 0，则该点位于图形内部。



由图可见，若采用非零绕组填充规则，则图形的绘制方向不同，其最后的填充结果是不一样的

- 8、元素(element)：路径中的每一个绘制步骤被称为元素，比如 `ph.moveTo(...); ph.lineTo(...);` 这里包含两个元素。
- 9、注意：`addEllipse()`、`addPath()`、`addPolygon()`、`addRect()`、`addRegion()`和 `addText()`这些函数实现上都是使用 `moveTo()`、`lineTo()`、`cubicTo()`函数来完成绘制的，因此 `ph.addRect(11,11,111,111);`这里包含 5 个元素，即


```
ph.moveTo(11,11);      ph.lineTo(122,11);      ph.lineTo(122,122);
ph.lineTo(11,122);     ph.lineTo(11,11);
```
- 10、使用路径时的作用域问题：

```
void paintEvent(QPaintEvent *e){
    QPainter pr(this);
    QPainterPath ph;
    ph.moveTo(11, 11);      ph.lineTo(111, 11);      .....    pr.drawPath(ph);
    qDebug()<<ph.elementCount();} //返回路径中的元素数。
```

始终应注意 `paintEvent()`函数在每次更新界面时，都会被调用，因此，以上代码在每次更新界面时都会调用一次 `paintEvent()`，这会导致每更更新一次就会在界面上重复绘制一次相同的路径，随着更新次数的增加绘制的图形会越来越多，若次数过多，比如达到几十万次甚至百万次时(路径稍微复杂一点就很容易达到这个数值)，会使计算机速度变得很慢。这可从 `ph.elementCount();`函数输出的结果可以看到。因此，应重新组织以上程序的代码，比如

```
class B:public QWidget{    Q_OBJECT
public: QPainterPath ph;
    B(QWidget *p1=0):QWidget(p1){
        ph.moveTo(11, 11);  ph.lineTo(111, 11);  ..... } //把路径放于构造函数中
    void paintEvent(QPaintEvent *e){
        QPainter pr(this);    pr.drawPath(ph);    }; //该函数中只需绘制路径即可。
```


二、QPainterPath 类中的函数

1、**QPainterPath**(); //构造函数

QPainterPath(const QPointF &*startPoint*)

QPainterPath(const QPainterPath &*path*)

2、常用绘制函数(见表)

QPainterPath 类中的常用函数	
以下函数省略了返回类型 void	
moveTo (const QPointF & <i>point</i>) moveTo (qreal <i>x</i> , qreal <i>y</i>)	移动当前点到位置 <i>point</i>
closeSubpath ();	画一条到子路径开头的线，然后关闭当前路径。
QPointF currentPosition () const;	返回路径的当前位置
lineTo (const QPointF & <i>endPoint</i>); lineTo (qreal <i>x</i> , qreal <i>y</i>);	绘制一条到点 <i>endPoint</i> 的直线
addRect (const QRectF & <i>rectangle</i>) addRect (qreal <i>x</i> , qreal <i>y</i> , qreal <i>width</i> , qreal <i>height</i>)	添加矩形，顺时针方向。
addEllipse (const QRectF & <i>boundingRectangle</i>) addEllipse (qreal <i>x</i> , qreal <i>y</i> , qreal <i>width</i> , qreal <i>height</i>) addEllipse (const QPointF & <i>center</i> , qreal <i>rx</i> , qreal <i>ry</i>)	添加椭圆，顺时针方向
addPath (const QPainterPath & <i>path</i>) addPolygon (const QPolygonF & <i>polygon</i>)	添加路径 添加多边形(注意: 多边形不会自动封闭)
addRoundedRect (const QRectF & <i>rect</i> , qreal <i>xRadius</i> , qreal <i>yRadius</i> , Qt::SizeMode <i>mode</i> = Qt::AbsoluteSize) void addRoundedRect (qreal <i>x</i> , qreal <i>y</i> , qreal <i>w</i> , qreal <i>h</i> , qreal <i>xRadius</i> , qreal <i>yRadius</i> , Qt::SizeMode <i>mode</i> = Qt::AbsoluteSize)	添加圆角矩形，顺时针方向
addText (const QPointF & <i>point</i> , const QFont & <i>font</i> , const QString & <i>text</i>) addText (qreal <i>x</i> , qreal <i>y</i> , const QFont & <i>font</i> , const QString & <i>text</i>) addRegion (const QRegion & <i>region</i>)	添加文本 添加区域
Qt::FillRule fillRule () const; void setFillRule (Qt::FillRule <i>fillRule</i>);	路径的填充规则，默认为 Qt::OddEvenFill(奇偶填充)，另一个取值为 Qt::WindingFill(非零绕组填充)。

3、其他绘制函数

1)、void **arcMoveTo**(const QRectF &*rectangle*, qreal *angle*)

void **arcMoveTo**(qreal *x*, qreal *y*, qreal *width*, qreal *height*, qreal *angle*)

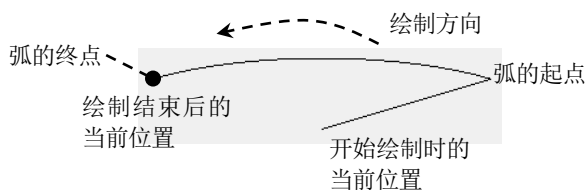
移动当前点的位置，其位置位于矩形 *rectangle* 的内接椭圆上，在椭圆上的具体坐标位置由 *angle* 根据椭圆参数方程 $x = a \cos(\text{angle})$; $y = b \sin(\text{angle})$; 计算，详见

QPainter::drawArc()

2)、void **arcTo**(const QRectF &*rectangle*, qreal *startAngle*, qreal *sweepLength*)

void **arcTo**(qreal *x*, qreal *y*, qreal *width*, qreal *height*, qreal *startAngle*, qreal *sweepLength*)

弧的绘制原理详见 **QPainter::drawArc**()，以上的角度是直接以度数指定的(而不是 1/16 度)，注意：弧是按逆时针方向绘制的，开始绘制时会把弧的起点连接到当前位置，绘制完成后，不会自动封闭图形，且当前位置位于弧的最后一个点上。原理见下图



4、修改路径

3)、bool **isEmpty**() const; 若路径中没有元素，或仅有唯一元素 MoveToElement，则返回 true。

4)、qreal **length**() const; //返回当前路径的长度

5)、int **elementCount**() const; //返回路径中元素的数量

6)、void **setElementPositionAt**(int **index**,qreal **x**,qreal **y**);

把索引 **index** 处的元素的 **x** 和 **y** 坐标设置为 **x** 和 **y**,索引就是绘制图形时使用的 moveTo()、lineTo()等函数的顺序(从 0 开始)。比如

```
QPainterPath ph;
ph.moveTo(11,11);    //索引 0
ph.lineTo(111,11);   //索引 1
ph.lineTo(111,111);  //索引 2
ph.lineTo(11,111);   //索引 3
ph.setElementPositionAt(3,55,55); //把索引 3 的元素 lineTo(11,111)修改为 lineTo(55,55);
```

7)、QPainterPath::Element **elementAt**(int **index**) const;

返回索引 **index** 处的元素。元素是使用嵌套类 QPainterPath::Element 描述的，该类拥有如下表的成员函数及变量

QPainterPath::Element 嵌套类中的成员	
bool isCurveTo () const	若元素是曲线，则返回 true
bool isLineTo () const	若元素是线条，则返回 true
bool isMoveTo () const	若元素正在移动当前位置，则返回 true
ElementType type	该变量保存元素的类型(ElementType 枚举见下表)
qreal x ; qreal y ;	分别保存元素的 x 和 y 坐标。

QPainterPath::ElementType 枚举(无标志)		
作用：描述元素的类型		
成员	值	说明
QPainterPath::MoveToElement	0	一个新的子路径
QPainterPath::LineToElement	1	一条线
QPainterPath::CurveToElement	2	一条曲线
QPainterPath::CurveToDataElement	3	描述曲线所需的额外数据。

5、路径运算的交集、并集、相减、平移

8)、QPainterPath **intersected**(const QPainterPath &**p**) const;

返回该路径与路径 **p** 的填充区域相交的路径。非封闭路径会被视为隐式关闭。贝塞尔曲线可能会变为线段。

9)、bool **intersects**(const QRectF &**rectangle**) const;

10)、bool **intersects**(const QPainterPath &p) const;

若该路径与矩形 rectangle 或路径 p 相交，则返回 true。

11)、QPainterPath **united**(const QPainterPath &p) const;

返回该路径与路径 p 的填充区域的并集。非封闭路径会被视为隐式关闭。贝塞尔曲线可能会变为线段。其算法与 QPolygonF::united()类似。

12)、QPainterPath **subtracted**(const QPainterPath &p) const;

返回从该路径中减去路径 p 后的路径，非封闭路径会被视为隐式关闭。贝塞尔曲线可能会变为线段。其算法与 QPolygonF::subtracted()类似。

13)、void **translate**(qreal dx, qreal dy);

void **translate**(const QPointF &offset);

QPainterPath **translated**(qreal dx, qreal dy) const;

QPainterPath **translated**(const QPointF &offset) const;

把路径中的所有点偏移(dx,dy)，即向 x 方向偏移 dx，向 y 方向偏移 dy。

6、其他运算(边界矩形、连接、包含、交换、反转、简化)

14)、QRectF **boundingRect**() const; //返回路径的边界矩形。其算法与 QPolygonF::boundingRect()类似

15)、QRectF **controlPointRect**() const;

返回包含路径中所有点和控制点的矩形，该函数返回的矩形总是 boundingRect()函数返回的矩形的超集。

16)、void **connectPath**(const QPainterPath &path)

把该路径与路径 path 相联，并从该路径的最后一个元素添加一条线到路径 path 的第一个元素。

17)、bool **contains**(const QPointF &point) const; //若点 point 位于路径中(包含轮廓线)，则返回 true。

bool **contains**(const QRectF &rectangle) const;

bool **contains**(const QPainterPath &p) const

若该路径包含矩形 rectangle 或路径 p，则返回 true。若 rectangle 或 p 与该路径有任何交集则返回 false，也就是说两个完全重合的路径会返回 false。

18)、void **swap**(QPainterPath &other); //把该路径与路径 other 相交换。

19)、QPainterPath **toReversed**() const;

返回一下与该路径相反的路径，比如，若路径以 moveTo(), lineTo(), arcTo()创建，则此函数返回的路径将是 arcTo(), lineTo(), moveTo()。

20)、QPainterPath **simplified**() const;

返回此路径的简化版本，即，合并所有相交子路径，返回没有交叉边的路径，还会合并连续的平行线。简化路径将始终使用填充规则 Qt::OddEvenFile。注：贝塞尔曲线可能会变为线段。

7、贝塞尔曲线

21)、void **cubicTo**(const QPointF &c1, const QPointF &c2, const QPointF &endPoint);

void **cubicTo**(qreal c1X, qreal c1Y, qreal c2X, qreal c2Y, qreal endPointX, qreal endPointY)

在当前位置与终点 `endPoint` 之间添加一条 3 次贝塞尔曲线(Bezier)，其中 `c1` 和 `c2` 是控制点。添加曲线后，当前点更新为点 `endPoint`

22)、void **quadTo**(const QPointF &*c*, const QPointF &*endPoint*);

void **quadTo**(qreal *cx*, qreal *cy*, qreal *endPointX*, qreal *endPointY*);

在当前位置和 `endPoint` 之间添加一条二次贝塞尔曲线，其中控制点为 `c`。添加曲线后，当前点更新为点 `endPoint`

以下函数若路径中有曲线，则百分比参数将映射到贝塞尔曲线方程的 `t` 参数，此时百分比关于长度不是线性的。

23)、qreal **angleAtPercent**(qreal *t*) const //返回路径在百分比 *t* 处的角度，*t* 必须位于 0 到 1 之间。

24)、qreal **slopeAtPercent**(qreal *t*) const; //返回路径在百分比 *t* 处的斜率，*t* 必须位于 0 到 1 之间。

25)、qreal **percentAtLength**(qreal *len*) const; //返回长度 *len* 在整个路径的百分比。

26)、QPointF **pointAtPercent**(qreal *t*) const; //返回路径在百分比 *t* 处的点。*t* 必须在 0 到 1 之间。

8、路径转换为多边形

27)、QPolygonF **toFillPolygon**(const QTransform &*matrix*) const;

QPolygonF **toFillPolygon**(const QMatrix &*matrix* = QMatrix()) const;

使用 `matrix` 把该路径转换为多边形，然后返回多边形。

28)、QList<QPolygonF> **toFillPolygons**(const QTransform &*matrix*) const;

QList<QPolygonF> **toFillPolygons**(const QMatrix &*matrix* = QMatrix()) const;

使用 `matrix` 把该路径转换为多边形列表，然后返回多边形列表。

29)、QList<QPolygonF> **toSubpathPolygons**(const QTransform &*matrix*) const;

QList<QPolygonF> **toSubpathPolygons**(const QMatrix &*matrix* = QMatrix()) const;

使用 `matrix` 把该路径转换为多边形列表，然后返回多边形列表。该函数会为每个子路径都创建一个多边形，而不管相交的子路径。

9、以下为重新实现的操作符函数

QPainterPath operator&(const QPainterPath &*other*) const; //交集

QPainterPath &operator&=(const QPainterPath &*other*);

QPainterPath operator+(const QPainterPath &*other*) const; //并集，与 |() 相同。

QPainterPath &operator+=(const QPainterPath &*other*);

QPainterPath operator|(const QPainterPath &*other*) const; //并，与 +() 同。

QPainterPath &operator|=(const QPainterPath &*other*);

QPainterPath operator-(const QPainterPath &*other*) const; //相减

QPainterPath &operator-=(const QPainterPath &*other*);

QPainterPath &operator=(const QPainterPath &*path*);

QPainterPath &operator=(QPainterPath &&*other*);

bool operator==(const QPainterPath &*path*) const;

bool operator!=(const QPainterPath &*path*) const;

12.7 绘制文本

1、需要使用到的 QPainter 类中的函数

QPainter 类中绘制文本的函数

注：以下函数都省略了返回类型 void

drawText(const QPointF &*position*, const QString &*text*);

drawText(const QPoint &*position*, const QString &*text*);

drawText(int *x*, int *y*, const QString &*text*);

在给定点 *position* 处绘制文本。以上函数不会处理“\n”字符，也不会自动换行，其中 *y* 坐标为显示文字时的基线。默认情况下，QPainter 绘制文本抗锯齿。

drawText(const QRectF &*rectangle*, int *flags*, const QString &*text*, QRectF **boundingRect*=Q_NULLPTR);

drawText(const QRect &*rectangle*, int *flags*, const QString &*text*, QRect **boundingRect*=Q_NULLPTR)

drawText(int *x*, int *y*, int *width*, int *height*, int *flags*, const QString &*text*, QRect **boundingRect*=Q_NULLPTR);

drawText(const QRectF &*rectangle*, const QString &*text*, const QTextOption &*option* = QTextOption());

使用标志 *flag* 在给定的矩形框 *rectangle* 中绘制文本。以上函数的顶部对齐矩形的 *y* 坐标。默认情况下，QPainter 绘制文本抗锯齿。QTextOption 类见第 11 章

drawStaticText(const QPointF &*topLeftPosition*, const QStaticText &*staticText*);

drawStaticText(const QPoint &*topLeftPosition*, const QStaticText &*staticText*);

drawStaticText(int *left*, int *top*, const QStaticText &*staticText*);

绘制静态文本，以上函数的顶部对齐 *y* 坐标，静态文本是为了提高绘制时的性能而使用的文本，本文不准备详细讲解此类，只简单介绍一下怎样使用。

void **setFont**(const QFont &*font*); //设置绘制文本时的字体。实际使用的字体不一定与设置的字体相同。

//若不存在匹配的字体，则使用最匹配的已安装字体。

const QFont & **font**() const; //返回使用 setFont()设置的字体，

QFontInfo **fontInfo**()const; //返回实际使用的字体

Qt::LayoutDirection **layoutDirection**() const;

void **setLayoutDirection**(Qt::LayoutDirection *direction*);

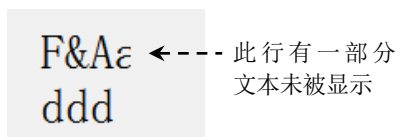
绘制文本时的布局方向(即 Qt::LeftToRight 还是 Qt::RightToLeft)，默认为 Qt::LayoutDirectionAuto(自动布局)

2、下面讲解以下函数的用法

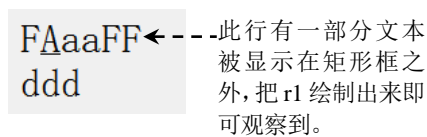
void **drawText**(const QRectF &*rectangle*, int *flags*, const QString &*text*, QRectF **boundingRect*=Q_NULLPTR);

以上函数表示，把文本 *text* 绘制在给定的矩形 *rectangle* 中，其中 *boundingRect* 用于指示包围文本的边框矩形，*flags* 用于指示文本的一些属性，*flags* 只能取下表中所列枚举的按位 OR 值(原理见图示)。

所属枚举	取值
Qt::AlignmentFlag	Qt::AlignLeft; Qt::AlignRight; Qt::AlignTop; Qt::AlignBottom; Qt::AlignCenter; Qt::AlignHCenter; Qt::AlignVCenter Qt::AlignJustify 描述文本的对齐方式
Qt::TextFlag 该枚举的某些取值 仅在打印环境下才有意义。	Qt::TextDontClip; //若不能在给定的范围内显示，则会在外面打印 Qt::TextSingleLine; //把所有空白视为空格，并只打印一行。 Qt::TextExpandTabs; //使 U+0009(即 ASCII tab)移至下一个制表符。 Qt::TextShowMnemonic; //把字符串"&S"显示为 <u>S</u> 。 Qt::TextWordWrap; //在合适的位置换行，比如单词的边界处。



```
QRect r1(55,55,55,133);
pr.drawText(r1, "F&AaFF ddd");
```



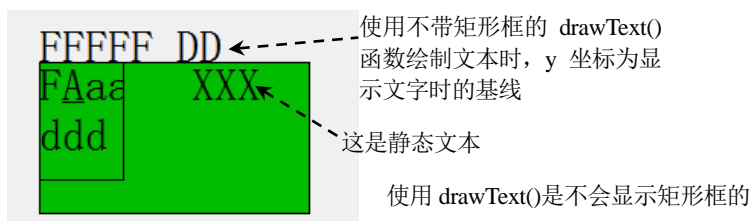
```
QRect r1(55,55,55,133);
pr.drawText(r1, Qt::TextDontClip|
Qt::TextShowMnemonic|
Qt::TextWordWrap,"F&AaFF ddd");
```

示例：文本的绘制

```
void paintEvent(QPaintEvent *e){
    QPainter pr(this);
    QBrush bs(QColor(1,188,1));    pr.setBrush(bs);
    QFont f;    f.setPointSize(22);    pr.setFont(f);    //设置字体
    QRectF r1(55,55,55,77);    QRectF r2(55,55,177,99);    //绘制矩形，注意绘制顺序。
    pr.drawRect(r2);    pr.drawRect(r1);
    pr.drawText(r1, Qt::TextShowMnemonic|Qt::TextWordWrap,"F&AaFF ddd",&r2);
    pr.drawText(QPointF(55,55),"FFFFF\n DD");
    pr.drawStaticText(QPointF(155,55),QStaticText("XXX")); }

```

运行结果及说明



12.8 QPen 类(画笔)

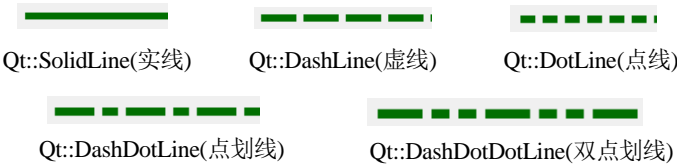
一、画笔基础

1、需要使用到的 QPainter 类中的函数原型如下：

```
void setPen(const QPen &pen);    //设置画笔，
void setPen(const QColor &color); //设置画笔，该笔样式为 Qt::SolidLine、宽度为 1，颜色由 color 指定
void setPen(Qt::PenStyle style); //设置画笔，该笔样式由 style 指定、宽度为 1，颜色为黑色
const QPen &pen() const;        //返回画笔
```

- 2、画笔定义了怎样绘制线条的形状和轮廓，还定义了文本的颜色。画笔决定了线条的粗细、颜色、样式(即线条的虚实)等属性。
- 3、默认画笔的宽度为 1，颜色为黑色、端点样式为 Qt::SquareCap、联接样式为 Qt::BevelJoin。
- 4、画笔样式：使用 setStyle()函数设置，使用枚举 Qt::PenStyle 进行描述，具体见表和图示

Qt::PenStyle 枚举(无标志)					
成员	值	说明	成员	值	说明
Qt::SolidLine	1	实线	Qt::DashDotLine	4	点划线
Qt::DashLine	2	虚线	Qt::DashDotDotLine	5	双点划线
Qt::DotLine	3	点线	Qt::CustomDashLine	6	自定义(使用 setDashPattern()函数设置)
Qt::NoPen	0	无画笔，即只绘制填充区域，但无边界线			



画笔样式(Qt::PenStyle 枚举)

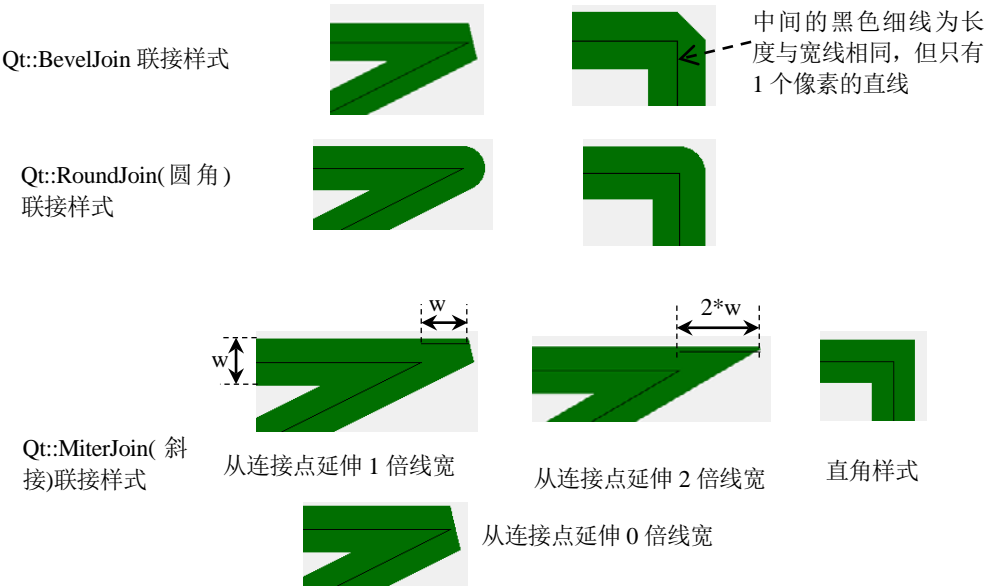
5、画笔端点样式：使用 setCapStyle()函数设置，使用枚举 Qt::PenCapStyle 进行描述，其取值和外观见图示



Qt::SquareCap 样式是一方形线端，把端点延长到线宽的一半，Qt::FlatCap 样式也是一个方形线端，但不会延长端点，Qt::RoundCap 是一个圆形的线端。

端点样式(Qt::PenCapStyle 枚举)

6、画笔联接样式：使用 `setJoinStyle()` 函数设置，使用枚举 `Qt::PenJoinStyle` 进行描述，其取值和外观见图示，斜接样式(`Qt::MiterJoin`)的延长距离使用 `setMiterLimit()` 函数进行设置，该函数以线的笔宽为单位来指定其延长距离，比如若笔宽为 11，若 `setMiterLimit(3)`;则表示延长 3 个笔宽单位这么长(即 $3*11=33$ 像素)。



Qt::SvgMiterJoin 联接样式：该样式用于与 SVG 标准的斜接联接相对接

联接样式(Qt::PenJoinStyle)枚举

二、QPen 类中的函数

- 1、**QPen()**; //构造一个宽度为 1，颜色为黑色、端点样式为 `Qt::SquareCap`、联接样式为 `Qt::BevelJoin` 的画笔
- QPen**(Qt::PenStyle *style*);
- QPen**(const QColor &*color*);
- QPen**(const QBrush &*brush*, qreal *width*, Qt::PenStyle *style* = Qt::SolidLine, Qt::PenCapStyle *cap* = Qt::SquareCap, Qt::PenJoinStyle *join* = Qt::BevelJoin);
- QPen**(const QPen &*pen*);
- QPen**(QPen &&*pen*);

QPen 类中的常用函数

注：设置函数都省略了返回类型 void
读取函数都省略了圆括号后面的 const

读取函数	设置函数	说明
int width () ; qreal widthF () ;	setWidth (int <i>width</i>); setWidthF (qreal <i>width</i>);	画笔宽度
QColor color () ; QBrush brush () ;	setColor (const QColor & <i>color</i>); setBrush (const QBrush & <i>brush</i>);	画笔颜色，setBrush()还可为画笔设置渐变色
Qt::PenStyle style () ;	setStyle (Qt::PenStyle <i>style</i>);	画笔样式

Qt::PenCapStyle capStyle() ;	setCapStyle (Qt::PenCapStyle <i>style</i>);	端点样式
Qt::PenJoinStyle joinStyle() ;	setJoinStyle (Qt::PenJoinStyle <i>style</i>);	联接样式
qreal miterLimit() ;	setMiterLimit (qreal <i>limit</i>);	斜接样式的延长距离，仅在联接样式为 Qt::MitreJoin 时，该函数有效。

2、自定义虚线

1)、 QVector<qreal> **dashPattern()** const; //返回画笔的虚线图案

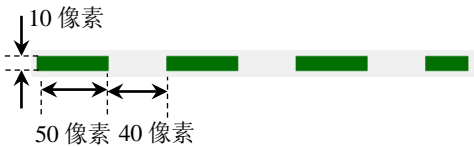
void **setDashPattern**(const QVector<qreal> &*pattern*);

- 设置画笔的虚线图案(即自定义虚线的样式)，该函数必须指定偶数个正项，其中奇数项是破折号，偶数项是空格。
- 虚线图案的单位是以笔宽为单位的，比如，宽度为 11 像素，若设置其长度为 5，即 5 个笔宽单位这么长，因此短划线的长度为 5*11=55 像素。
- 注意：0 宽度相当于是 1 像素宽的装饰笔。
- 另外还要注意，端点样式也会影响虚线的长短，比如若端点样式为 Qt::SquareCap(默认值)，则会在线的端点处延长宽度的一半(详见正文)。
- 设置该函数会隐式的把画笔样式转换为 Qt::CustomDashLine。

```

QPainter pr(this);
QPen pn;
pn.setWidth(10); //设置线宽
QVector<qreal> v; v<<5<<4;
pn.setDashPattern(v);
//不延长端头
pn.setCapStyle(Qt::FlatCap);
pn.setColor(QColor(1,111,1));
pr.setPen(pn);
pr.drawLine(33,33,333,33);

```

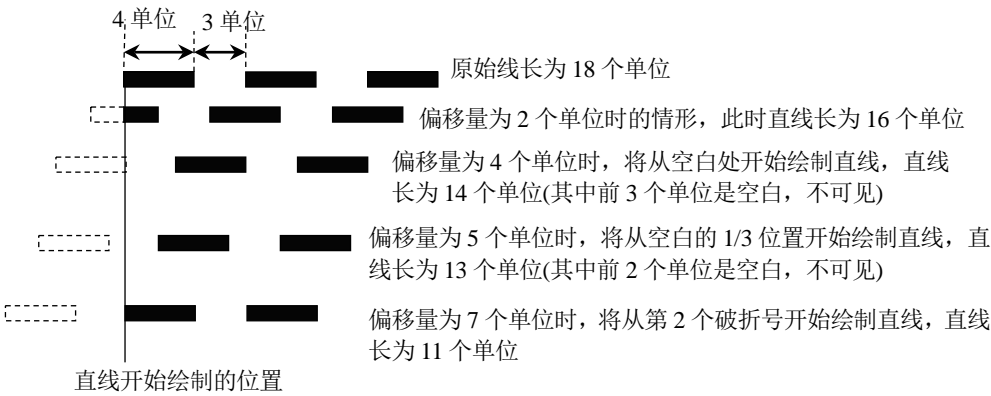


自定义虚线示例

2)、 qreal **dashOffset()** const; //返回画笔的虚线偏移量。

void **setDashOffset**(qreal *offset*);

设置虚线偏移量(虚线图案的起点)，单位由虚线图案(即 setDashPattern()函数)而定，设置该函数将导致画笔样式转换为 Qt::CustomDashLine。虚线偏移量原理见图示



虚线偏移量原理

3、bool **isCosmetic**() const;

void **setCosmetic**(bool *cosmetic*);

设置装饰笔(cosmetic pen)，装饰笔是指具有恒定宽度的笔，使用装饰笔绘制的形状，可确保其轮廓在不同缩放因素下具有相同的厚度。默认情况下，0 宽度的笔就是装饰笔。

4、bool **isSolid**() const;

若画笔具有实心填充，则返回 true。注意：虚线也是实心填充，非实心填充是指使用 QPen::setBrush() 设置的画刷，当画刷是渐变或纹理时的情形。

5、void **swap**(QPen &other); //把此笔与 other 交换。

6、重新实现的操作符函数

operator **QVariant**() const;

bool operator!=(const QPen &*pen*) const;

QPen &operator=(const QPen &*pen*);

QPen &operator=(QPen &&*other*);

bool operator==(const QPen &*pen*) const;

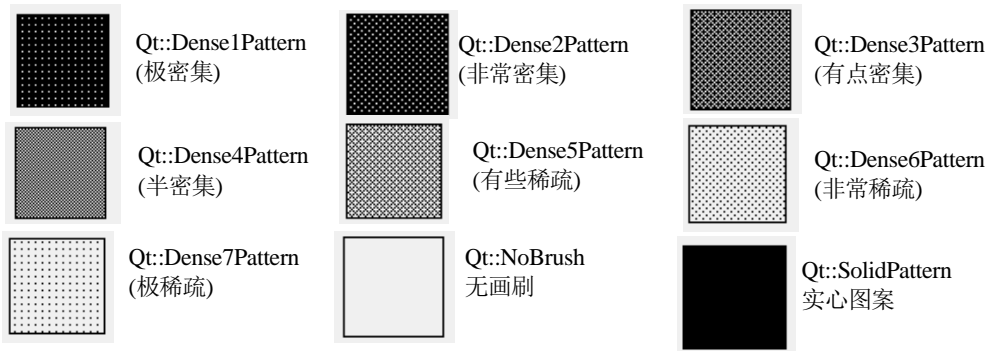
12.9 QBrush 类(画刷)与渐变(QGradient 类及其子类)

一、画刷基础

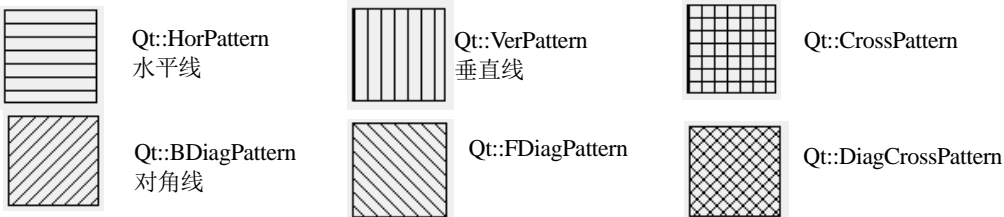
1、需要使用到的 QPainter 类中的函数原型如下：

QPainter 类中与画刷有关的函数	
void setBrush (const QBrush & <i>brush</i>) void setBrush (Qt::BrushStyle <i>style</i>) const QBrush & brush () const	画刷
void setOpacity (qreal <i>opacity</i>) qreal opacity () const	设置画刷的透明度，值在 0 到 1 之间，0 表示完全透明，1 表示完全不透明。默认为 1。

- 2、QBrush 类(画刷)用于描述 QPainter 绘制的形状的填充颜色或图案。
- 3、画刷定义了样式、颜色、渐变和纹理(其实就是一个图片)，下面分别介绍。
- 画刷颜色使用 setColor()函数或在 QBrush()的构造函数中设置。
 - 画刷的渐变色需要使用 QGradient 类及其子类，并在 QBrush()的构造函数中设置。
 - 画刷的纹理可使用 setTexture()函数或在 QBrush()的构造函数中设置。
 - 画刷的样式见下文。
- 4、画刷的样式(Qt::BrushStyle 枚举)：
- 画刷样式使用 Qt::BrushStyle 枚举进行描述(见下图和表)，画刷样式可在 QBrush 类的构造函数进行设置，或使用 setStyle()函数设置。



画刷样式(Qt::BrushStyle 枚举)

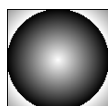




Qt::LinearGradientPattern, 线性渐变
需使用专用的
QBrush 构造函数



Qt::ConicalGradientPattern, 圆锥渐变
需使用专用的
QBrush 构造函数



Qt::RadialGradientPattern, 径向渐变
需使用专用的
QBrush 构造函数



Qt::TexturePattern, 自定义图案(即画刷的纹理)需使用
QBrush::setTexture()函数

画刷样式(Qt::BrushStyle 枚举)

二、QBrush 类中的函数

1、构造函数，见下表

QBrush 类的构造函数	
QBrush(); // 构造一个默认为黑色，样式为 Qt::NoBrush(即不会填充形状)的画刷	
QBrush(Qt::BrushStyle style); QBrush(const QColor &color, Qt::BrushStyle style = Qt::SolidPattern); QBrush(Qt::GlobalColor color, Qt::BrushStyle style = Qt::SolidPattern);	使用颜色 color 和样式 style 构造一个画刷
QBrush(const QColor &color, const QPixmap &pixmap); QBrush(Qt::GlobalColor color, const QPixmap &pixmap); QBrush(const QPixmap &pixmap); QBrush(const QImage &image);	使用颜色 color 和/或像素图 pixmap 构造一个画刷，样式被设置为 Qt::TexturePattern，这些构造函数用于设置画刷的纹理，color 仅对 QPixmap 有影响。另见 setTexture() 和 setTextureImage() 函数
QBrush(const QGradient &gradient);	这是构造渐变色画刷的方法，关于 QGradient 类及其子类，详见后文。
QBrush(const QBrush &other);	

2、颜色、样式、透明性

1)、const QColor &color() const; //返回画刷的颜色

void setColor(const QColor &color);

void setColor(Qt::GlobalColor color);

画刷的颜色，注意：若画刷是渐变、纹理(纹理是 QPixmap 除外)，则调用 setColor() 函数不会有作用。

2)、void setStyle(Qt::BrushStyle style); //设置画刷的样式

Qt::BrushStyle style() const; //返回画刷的样式

3)、const QGradient *gradient() const; //返回此画刷的渐变

4)、bool isOpaque() const;

若画刷是完全不透明的，则返回 true，否则返回 false。以下情况被认为是透明的

- color() 的 alpha 分量是 255(即不透明)，使用 QColor::setAlpha() 设置。比如

QColor c(255,1,1); c.setAlpha(254); QBrush bs(c); 则 bs.isOpaque() = false

- 纹理(texture())不透明，因为没有 alpha 通道。

- 渐变(`gradient()`)中的颜色都具有 255 的 alpha 分量(即不透明)。
- 扩展的辐射渐变(透明), `QBitmap`(透明)。

注意: 该函数与 `QPainter::setOpacity()` 没有关系。

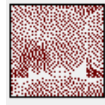
3、纹理(即自定义图案)

5)、`QPixmap texture()` const; //返回画刷图案。

void `setTexture`(const `QPixmap &pixmap`);

设置画刷的像素图为 `pixmap`, 样式被设置为 `Qt::TexturePattern`(纹理), 此时画刷只对单色像素图(即 `QPixmap::depth()==1`)有作用, 比如(效果见图):

```
QPainter pr(this);
QBrush bs(QColor(111,1,1), QBitmap("F:/li.png"));
pr.setBrush(bs);
pr.drawRect(11,11,55,55);
```



6)、`QImage textureImage()` const; //返回画刷图案, 若纹理被设置为 `QPixmap`, 则将其转换为 `QImage`

void `setTextureImage`(const `QImage &image`);

设置画刷的图像为 `image`, 样式被设置为 `Qt::TexturePattern`(纹理), 注意: 该函数与使用 `QBitmap` 调用 `QBrush::setTexture()` 不同, 当前画刷不会对单色图像产生作用, 若要改变单色图像的颜色, 可使用 `QBitmap::fromImage()` 把图像转换为 `QBitmap`, 并把生成的 `QBitmap` 设置为纹理。

4、坐标变换

7)、const `QMatrix &matrix()` const; //返回画刷的当前变换矩阵

void `setMatrix`(const `QMatrix &matrix`);

设置画刷的变换矩阵, 把画刷的变换矩阵与 `QPainter` 的变换矩阵合并, 以产生最终的结果。

8)、`QTransform transform()` const; //返回画刷的变换矩阵

void `setTransform`(const `QTransform &matrix`);

设置画刷的变换矩阵为 `matrix`, 把画刷变换矩阵与 `QPainter` 变换矩阵合并, 以产生最终的结果。

5、其他

9)、void `swap`(`QBrush &other`); //把该画刷与 `other` 交换。

6、重新实现的操作符函数

operator `QVariant`() const;

bool operator!=(const `QBrush &brush`) const

`QBrush &operator`=(const `QBrush &brush`)

`QBrush &operator`=(`QBrush &&other`)

bool operator==(const `QBrush &brush`) const

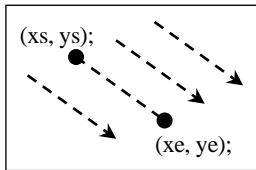
三、渐变(`QGradient` 类及其子类)

1、`QGradient` 类与 `QBrush` 类一起用于指定渐变填充。

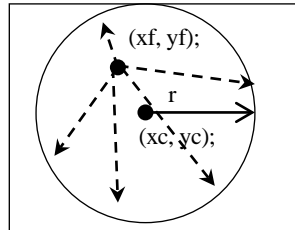
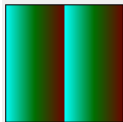
2、Qt 目前支持 3 种类型的渐变填充, 如下

- 线性渐变: 使用 `QLinearGradient` 类描述, 在起点(`xs, ys`)和终点(`xe, ye`)之间插入颜色, 在起点和终点之间的区域为线性渐变区域。原理见下图。

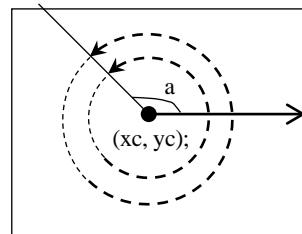
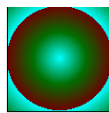
- 辐射(径向)渐变: 由 `QRadialGradient` 类描述。由中心点 (x_c, y_c) 和半径 r 定义一个圆(该圆即为辐射渐变区域), 然后颜色从焦点 (x_f, y_f) 向外扩散, 原理见下图。
- 圆锥渐变: 由 `QConicalGradient` 类描述。由一个中心点 (x_c, y_c) 和一个角度 a 定义, 然后颜色在中心点周围像钟表那样扩散。原理见下图。



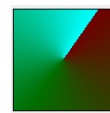
线性渐变(`QLinearGradient`)



辐射渐变(`QRadialGradient`)



圆锥渐变(`QConicalGradient`)



3、渐变颜色的设置

- 渐变颜色不能使用 `QBrush::setColor()` 函数设置,
- 渐变颜色需要使用停止点和颜色两个属性来指定, 可使用 `QGradient::setColorAt()` 函数来设置单个的停止点和颜色, 还可使用 `QGradient::setStops()` 来一次定义多个<停止点, 颜色>对, `setStops()` 函数需要使用 `QGradientStop` 类型的参数。注: 停止点使用的是 0~1 之间的数值来表示的, 0 表示起点, 1 表示终点。
- <位置, 颜色>对是使用 `QGradientStop` 类型来描述的, `QGradientStop` 类型是

`typedef QPair<qreal, QColor> QGradientStop;`

其中 `QPair` 是一个模板类, 用于存储对项, 比如 `QPair<T1, T2> pr`, 则 `pr` 可存储一个 `T1` 类型的值和一个 `T2` 类型的值。

下面为其示例(效果见右图):

```
QPainter pr(this);
//创建线性渐变, 起点为(33, 50), 终点为(111, 50);
QLinearGradient g(QPointF(33, 50), QPointF(144, 50));
//颜色设置需位于 QBrush bs(g) 之前。
g.setColorAt(0, QColor(111, 1, 1)); //设置起始点颜色为红色
g.setColorAt(0.5, QColor(1, 111, 1)); //在起点和终点之间设置颜色为绿色
g.setColorAt(1, QColor(255, 255, 2)); //设置终点颜色为黄色。
QBrush bs(g);
pr.setBrush(bs);
pr.drawRect(33, 50, 111, 111);
```



4、渐变色的传播方式(spread)

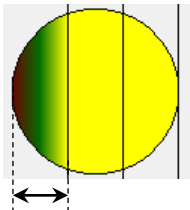
渐变色的传播是指在渐变区域以外的区域渐变色是怎样进行扩散的。可使用 `QGradient::setSpread()` 函数进行设置, 其传播方式使用枚举 `QGradient::Spread` 进行描述, 取值如下表(外观及原理见图示)。

注意：传播方式仅对线性渐变和辐射渐变有作用，因为这两种类型的渐变是有边界的，而锥形渐变其渐变范围是 0~360 度的圆，因此没有渐变边界，所以传播方式不适用于锥形渐变。

QGradient::Spread 枚举(无标志)

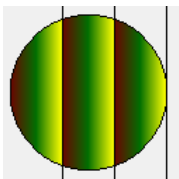
作用：描述渐变色的传播方式

成员	值	说明
QGradient::PadSpread	0	填充传播。渐变在区域外使用与终止颜色最接近的色， 默认值 。
QGradient::RepeatSpread	2	重复传播。渐变在区域外重复传播
QGradient::ReflectSpread	1	反射传播。渐变以相反的方式在区域外传播。



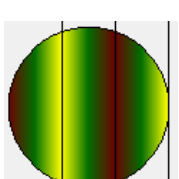
线性渐变区域，颜色顺序为红，绿，黄，传播的区域都为黄色

PadSpread(填充传播)



渐变在线性渐变区域之外，按照原顺序(红、绿、黄)重复传播

RepeatSpread(重复传播)



渐变在线性渐变区域之外，按照原顺序相反的顺序(黄、绿、红)交替重复传播

ReflectSpread(反射传播)

渐变色传播方式

5、坐标模式：就是指的怎样指定渐变色的坐标，共有 3 种方法，使用枚举

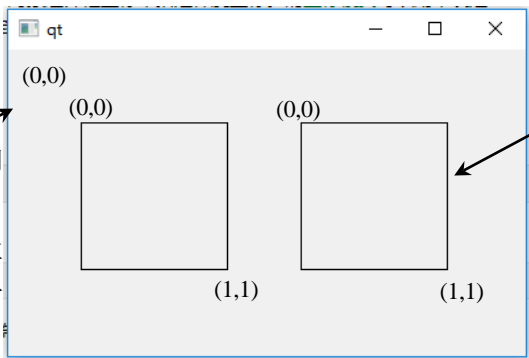
QGradient::CoordinateMode 进行描述(见下表及图), 可使用 QGradient::setCoordinateMode() 函数进行设置。

QGradient::CoordinateMode 枚举(无标志)

作用：描述渐变的坐标模式

成员	值	说明
QGradient::LogicalMode	0	渐变的坐标与对象坐标相同， 默认值 。
QGradient::StretchToDeviceMode	1	设备边界模式。渐变坐标位于绘制设备的边界矩形内，左上角为(0,0)，右下角为(1,1)
QGradient::ObjectBoundingMode	2	对象边界模式。渐变坐标位于对象的边界矩形内，左上角为(0,0)，右下角为(1,1)

该窗口部件就是绘制设备
StretchToDeviceMode 模式的坐标范围就位于该部件内，取值从(0,0)到(1,1)



若渐变色填充到该矩形的话，则该矩形就是对象。
ObjectBoundingMode 模式的坐标范围就在该矩形内，取值从(0,0)到(1,1)

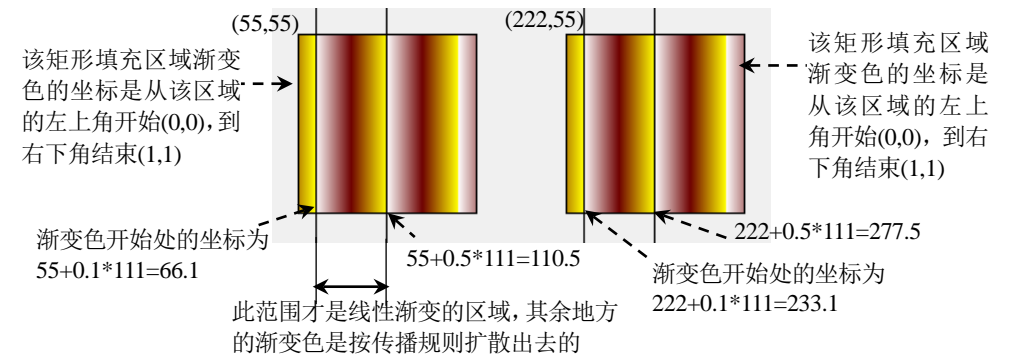
(1,1)

渐变坐标模式

示例：渐变坐标模式

```
void paintEvent(QPaintEvent *e) {
    QPainter pr(this);
    QLinearGradient g(QPointF(0.1, 0.50), QPointF(0.50, 0.50)); //起始/终点坐标以小数点指定
    g.setCoordinateMode(QGradient::ObjectBoundingMode); //设置渐变坐标模式
    g.setColorAt(0, QColor(255, 255, 255)); //渐变起始颜色
    g.setColorAt(0.5, QColor(111, 1, 1));
    g.setColorAt(1, QColor(255, 255, 1)); //渐变最终停止点颜色
    g.setSpread(QGradient::RepeatSpread); //设置渐变色传播模式
    QBrush bs(g); //创建画刷
    pr.setBrush(bs);
    pr.drawRect(55, 55, 111, 111); //渐变填充区域 1
    pr.drawRect(222, 55, 111, 111); //渐变填充区域 2
    pr.drawLine(66.1, 0, 66.1, 444); pr.drawLine(110.5, 0, 110.5, 444); //用于测量
    pr.drawLine(233.1, 0, 233.1, 444); pr.drawLine(277.5, 0, 277.5, 444); //用于测量
}
```

运行结果及说明



6、QGradient 类中的函数

QGradient 类中的函数

注：1、该类没有公有构造函数。2、设置函数都省略了返回类型 void。3、读取函数都省略了圆括号后面的 const。

读取函数	设置函数	说明
CoordinateMode <code>coordinateMode()</code> ;	<code>setCoordinateMode(CoordinateMode mode);</code>	渐变的坐标模式
Spread <code>spread()</code> ;	<code>setSpread(Spread method);</code>	渐变传播方式
QGradientStops <code>stops()</code> ;	<code>setStops(const QGradientStops &stopPoints);</code>	设置渐变<位置, 颜色>对
<code>void setColorAt(qreal position, const QColor &color);</code> //把停止位置 position 处的颜色设置为 color		
Type <code>type()</code> const; //渐变的类型, 枚举见下表		
<code>typedef QPair<qreal, QColor> QGradientStop;</code>		
<code>typedef QVector<QGradientStop> QGradientStops;</code> //这是新定义的类型。setStops()的参数		

QGradient::Type 枚举(无标志)

作用：描述渐变的类型

成员	值	说明
QGradient::LinearGradient	0	线性渐变(即 QLinearQGradient)。
QGradient::RadialQGradient	1	辐射渐变(即 QRadialQGradient)
QGradient::ConicalQGradient	2	焦点渐变(即 QConicalQGradient)

7、QLinearGradient 类(线性渐变)中的函数

- 1)、**QLinearGradient**(); 在(0,0)到(1,1)之间构造一个线性渐变。
- 2)、**QLinearGradient**(const QPointF &*start*, const QPointF &*finalStop*);
QLinearGradient(qreal *x1*, qreal *y1*, qreal *x2*, qreal *y2*);
在开始点(*start* 或(*x1*,*y1*))与结束点 *finalStop* 或(*x2*,*y2*))之间构造一个线性渐变, 参数值预期以像素为单位。
- 3)、QPointF **finalStop**() const; //返回渐变的最终停止点(逻辑坐标)
void **setFinalStop**(const QPointF &*stop*); //在逻辑坐标中设置渐变的最终停止点(就是渐变的结束点)
void **setFinalStop**(qreal *x*, qreal *y*);
- 4)、void **setStart**(const QPointF &*start*); //在逻辑坐标中设置线性渐变的起点。
void **setStart**(qreal *x*, qreal *y*);
QPointF **start**() const; //返回渐变的起点(逻辑坐标)

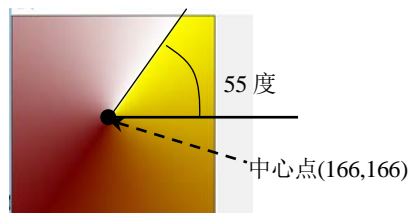
8、QConicalGradient 类(圆锥渐变)中的函数

- 1)、**QConicalGradient**(); //以(0,0)为中心, 0 为开始角度, 构造一个圆锥渐变。
- 2)、**QConicalGradient**(const QPointF &*center*, qreal *angle*);
QConicalGradient(qreal *cx*, qreal *cy*, qreal *angle*);
以 *center* 或(*cx*, *cy*)为中心, 以 *angle*(0~360)为开始角度, 构造一个圆锥渐变。
- 3)、qreal **angle**() const; //以逻辑坐标返回圆锥渐变的起始角度。
void **setAngle**(qreal *angle*); //以逻辑坐标设置圆锥渐变的起始角度为 *angle*
- 4)、QPointF **center**() const; //以逻辑坐标返回圆锥渐变的中心。
void **setCenter**(const QPointF &*center*); //以逻辑坐标设置圆锥渐变的中心为 *center*
void **setCenter**(qreal *x*, qreal *y*);

示例: 圆锥渐变

```
void paintEvent(QPaintEvent *e) {
    QPainter pr(this);
    QConicalGradient g(QPointF(166, 166), 55);
    g.setColorAt(0, QColor(255, 255, 255));
    g.setColorAt(0.5, QColor(111, 1, 1));
    g.setColorAt(1, QColor(255, 255, 1));
    //传播方式对于圆锥渐变不起作用
    g.setSpread(QGradient::RepeatSpread);
    QBrush bs(g); pr.setBrush(bs);
    pr.drawRect(1, 1, 333, 333);}

```

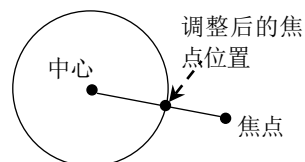


9、QRadialGradient 类(辐射渐变)中的函数

- 1)、**QRadialGradient**(); //以(0, 0)为中心和焦点, 半径为 1, 构造一个简单的辐射渐变。
- 2)、**QRadialGradient**(const QPointF &*center*, qreal *radius*);
QRadialGradient(qreal *cx*, qreal *cy*, qreal *radius*);
以(*cx*, *cy*)或 *center* 为中心, *radius* 为半径, 构造一个简单的辐射渐变
- 3)、**QRadialGradient**(const QPointF &*center*, qreal *radius*, const QPointF &*focalPoint*);
QRadialGradient(qreal *cx*, qreal *cy*, qreal *radius*, qreal *fx*, qreal *fy*);

以(*cx*, *cy*)或 *center* 为中心, *radius* 为半径, (*fx*, *fy*)或 *focalPoint* 为焦点, 构造一个简单的辐射渐变。

注意: 若焦点位于由中心和半径定义的圆之外, 则焦点将被调整为位于中心和焦点的连线与圆



的交点上(原理见右图)。

4)、**QRadialGradient**(const QPointF &*center*, qreal *centerRadius*, const QPointF &*focalPoint*,
qreal *focalRadius*);

QRadialGradient(qreal *cx*, qreal *cy*, qreal *centerRadius*, qreal *fx*, qreal *fy*, qreal *focalRadius*);

以(cx, cy)或 *center* 为中心, *radius* 为半径, (fx, fy)或 *focalPoint* 为焦点, *focalRadius* 为焦点半径, 构造一个扩展的辐射渐变。

QRadialGradient 类中的函数

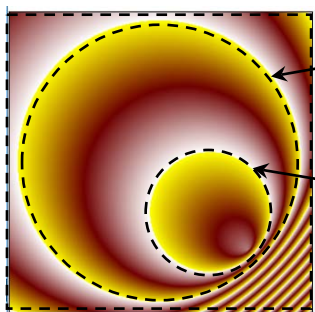
注: 1、设置函数都省略了返回类型 void。2、读取函数都省略了圆括号后面的 const。

3、以下函数都是逻辑坐标

读取函数	设置函数	说明
QPointF center () ;	setCenter (const QPointF & <i>center</i>); setCenter (qreal <i>x</i> , qreal <i>y</i>);	辐射渐变的中心
qreal centerRadius () ; qreal radius () ;	setCenterRadius (qreal <i>radius</i>); setRadius (qreal <i>radius</i>);	辐射渐变的中心半径
QPointF focalPoint () ;	setFocalPoint (const QPointF & <i>focalPoint</i>); setFocalPoint (qreal <i>x</i> , qreal <i>y</i>);	辐射渐变的焦点
qreal focalRadius () ;	setFocalRadius (qreal <i>radius</i>);	辐射渐变的焦点半径

示例: 扩展的辐射渐变

```
void paintEvent(QPaintEvent *e) {  
    QPainter pr(this);  
    QRadialGradient g(QPointF(166, 166), 155, QPointF(222, 222), 66);  
    g.setColorAt(0, QColor(255, 255, 255)); //开始颜色(白色)  
    g.setColorAt(0.5, QColor(111, 1, 1)); //中间颜色(红色)  
    g.setColorAt(1, QColor(255, 255, 1)); //终点颜色(黄色)  
    g.setSpread(QGradient::RepeatSpread); //传播方式(重复)  
    QBrush bs(g);    pr.setBrush(bs);  
    pr.drawRect(1, 1, 333, 333);}
```



辐射渐变的区域圆, 该圆的半径就是中心半径, 该圆的圆心就是中心。在该区域之外的渐变是由传播方式扩散的

焦点圆, 该圆的半径就是焦点半径, 该圆的圆心就是焦点

扩展的辐射渐变

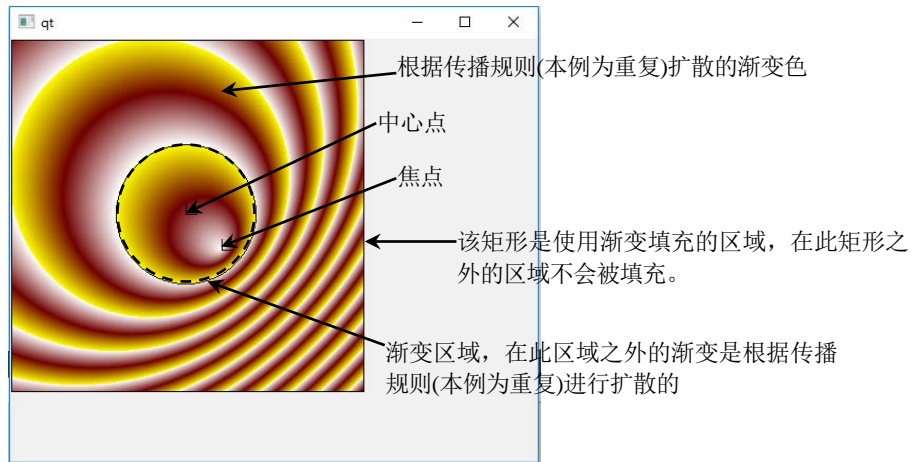
示例: 简单的辐射渐变与传播方式的效果

```
void paintEvent(QPaintEvent *e) {  
    QPainter pr(this);  
    QRadialGradient g(QPointF(166, 166), 66, QPointF(200, 200));  
    g.setColorAt(0, QColor(255, 255, 255));  
    g.setColorAt(0.5, QColor(111, 1, 1));  
    g.setColorAt(1, QColor(255, 255, 1));  
    g.setSpread(QGradient::RepeatSpread); //传播方式为重复传播
```

```

QBrush bs(g);          pr.setBrush(bs);
pr.drawRect(1, 1, 333, 333); //渐变填充的区域
pr.drawEllipse(QPointF(166, 166), 66, 66); //绘制一个圆，该圆的大小与辐射渐变区域相同。
//以下代码用于标出中心点和焦点的位置。
pr.drawLine(166, 166, 176, 166);          pr.drawLine(166, 166, 166, 156);
pr.drawLine(200, 200, 210, 200);          pr.drawLine(200, 200, 200, 190);}

```



简单的辐射渐变

12.10 填充

1、需要使用到的 QPainter 类中的函数原型如下：

QPainter 类中与填充有关的函数	
注：以下函数都省略了返回类型 void	
fillPath (const QPainterPath & <i>path</i> , const QBrush & <i>brush</i>)	使用画刷 <i>brush</i> 填充路径，填充后没有轮廓线条
strokePath (const QPainterPath & <i>path</i> , const QPen & <i>pen</i>)	使用该函数可绘制由 <i>fillPath</i> ()填充后，路径的轮廓线条。
fillRect (const QRectF & <i>rectangle</i> , const QBrush & <i>brush</i>) fillRect (const QRect & <i>rectangle</i> , const QBrush & <i>brush</i>) fillRect (int <i>x</i> , int <i>y</i> , int <i>width</i> , int <i>height</i> , const QBrush & <i>brush</i>)	使用画刷 <i>brush</i> 填充矩形区域，填充后没有轮廓线条
fillRect (const QRectF & <i>rectangle</i> , const QColor & <i>color</i>) fillRect (const QRect & <i>rectangle</i> , const QColor & <i>color</i>) fillRect (int <i>x</i> , int <i>y</i> , int <i>width</i> , int <i>height</i> , const QColor & <i>color</i>)	使用颜色 <i>color</i> 填充矩形区域，填充后没有轮廓线条
fillRect (int <i>x</i> , int <i>y</i> , int <i>width</i> , int <i>height</i> , Qt::GlobalColor <i>color</i>) fillRect (const QRect & <i>rectangle</i> , Qt::GlobalColor <i>color</i>) fillRect (const QRectF & <i>rectangle</i> , Qt::GlobalColor <i>color</i>)	使用预定义的全局颜色 <i>color</i> 填充矩形区域，填充后没有轮廓线条
fillRect (int <i>x</i> , int <i>y</i> , int <i>width</i> , int <i>height</i> , Qt::BrushStyle <i>style</i>) fillRect (const QRect & <i>rectangle</i> , Qt::BrushStyle <i>style</i>) fillRect (const QRectF & <i>rectangle</i> , Qt::BrushStyle <i>style</i>)	使用画刷样式填充矩形区域，填充后没有轮廓线条
eraseRect (const QRectF & <i>rectangle</i>) eraseRect (int <i>x</i> , int <i>y</i> , int <i>width</i> , int <i>height</i>) eraseRect (const QRect & <i>rectangle</i>)	擦除矩形区域的填充
void setBrushOrigin (const QPointF & <i>position</i>) void setBrushOrigin (const QPoint & <i>position</i>) void setBrushOrigin (int <i>x</i> , int <i>y</i>) QPointF brushOrigin () const	画刷原点。影响画刷纹理图案(即填充图片)和渐变
void setBackground (const QBrush & <i>brush</i>) const QBrush & background () const	背景画刷(在透明模式(默认)下不起作用)
void setBackgroundMode (Qt::BGMode <i>mode</i>) Qt::BGMode backgroundMode () const	背景模式，是否用背景颜色填充背景，比如虚线空白间隙的颜色， <i>mode</i> 可取值有 Qt::TransparentMode(透明)，Qt::OpaqueMode(不透明)

- 2、填充区域：当绘制封闭的形状时，在其内部可以填充各种颜色、图案等属性。但并不是所有封闭形状都可以被填充，比如使用 QPainter::drawLine()函数绘制的封闭形状就不能被填充，以下把可被填充的形状，称为填充区域。
- 3、使用 QPainter 对象绘制的直线、弧不是填充区域，使用 QPainter 对象绘制的矩形、椭圆、弦、扇形是填充区域，使用路径 QPainterPath 绘制的图形是填充区域。另外，使用 QPen 绘制的轮廓也是可被填充的。
- 4、注意：填充区域不一定有轮廓线，比如使用 Qt::NoPen 画笔样式绘制的填充区域就没有轮廓线。
- 5、使用画刷即可向填充区域中填充颜色、图案、渐变等属性。
- 6、填充直线、虚线的空白处、字体背景

要填充虚线的空白和字体背景，需要使用 `setBackground()` 函数设置背景画刷，且还需使用 `setBackgroundMode()` 函数把背景模式设置为 `Qt::OpaqueMode` (不透明)，下面为其示例

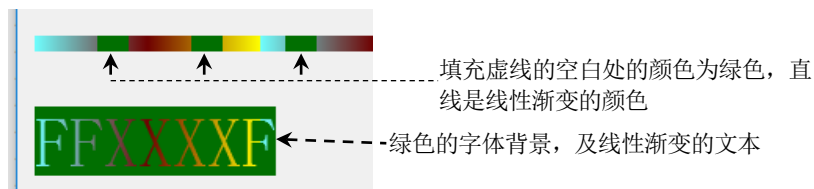
示例：填充直线、虚线的空白处、字体背景

```
void paintEvent(QPaintEvent *e) {
    QPainter pr(this);
    QLinearGradient g(QPointF(11, 11), QPointF(155, 11));    //创建直线渐变
    g.setColorAt(0, QColor(111, 255, 255));    //青色
    g.setColorAt(0.5, QColor(111, 1, 1));    //红色
    g.setColorAt(1, QColor(255, 255, 1));    //黄色
    g.setSpread(QGradient::RepeatSpread);
    //创建两个画刷
    QBrush bs(g);    QBrush bs1(QColor(1, 111, 1));    //bs1 为绿色
    //设置画笔
    QPen pn;
    pn.setCapStyle(Qt::FlatCap); //该端头样式可消除端头延长部分的影响
    pn.setWidth(10);    //线宽
    pn.setStyle(Qt::DashLine); //设置为虚线
    pn.setBrush(bs);    //设置画笔的画刷为 bs (渐变色)
    pr.setPen(pn);    //设置画笔
    //设置字体
    QFont f;    f.setPointSize(33);    pr.setFont(f);

    pr.setBackground(bs1); //设置背景画刷为 bs1 (纯色)
    pr.setBackgroundMode(Qt::OpaqueMode); //设置背景模式为不透明

    pr.drawText(11, 111, "FFXXXXF"); //绘制文字
    pr.drawLine(11, 33, 151, 33); } //绘制直线
```

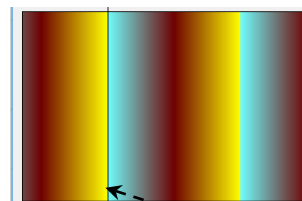
运行结果及说明



7、画刷原点对于使用渐变填充的区域和使用画刷填充纹理图案时会有一定影响，下面举示例说明。

示例：画刷原点与渐变填充

```
void paintEvent(QPaintEvent *e) {
    QPainter pr(this);
    QLinearGradient g(QPointF(0, 11), QPointF(155, 11));
    g.setColorAt(0, QColor(111, 255, 255));
    g.setColorAt(0.5, QColor(111, 1, 1));
    g.setColorAt(1, QColor(255, 255, 1));
    g.setSpread(QGradient::RepeatSpread);
    QBrush bs(g);    pr.setBrush(bs);
    pr.setBrushOrigin(111, 111);
    pr.drawRect(11, 11, 333, 222);
    pr.drawLine(111, 0, 111, 444); }
```



示例：画刷原点与纹理图案

```
void paintEvent(QPaintEvent *e) {  
    QPainter pr(this);  
    QBrush bs1(QPixmap("F:/5b.jpg"));  
    pr.setBrush(bs1);  
    pr.setBrushOrigin(111, 111);    //把画刷原点设置为与矩形的左上角相同，以使画刷从  
                                     //图片 5b.jpg 的左上角开始填充矩形。  
    pr.drawRect(111, 111, 400, 300); }
```



此时显示的是完整的图片



注释掉 pr.setBrushOrigin(111,111);之后的效果

12.11 裁剪区域(QRegion 类)

一、裁剪区域基础

1、需要使用到的 QPainter 类中的函数原型如下：

- 1)、void **setClipPath**(const QPainterPath &path, Qt::ClipOperation operation = Qt::ReplaceClip);
void **setClipRect**(const QRectF &rectangle, Qt::ClipOperation operation = Qt::ReplaceClip);
void **setClipRect**(const QRect &rectangle, Qt::ClipOperation operation = Qt::ReplaceClip);
void **setClipRect**(int x, int y, int width, int height, Qt::ClipOperation operation = Qt::ReplaceClip);
void **setClipRegion**(const QRegion ®ion, Qt::ClipOperation operation = Qt::ReplaceClip);
把路径 path 或矩形 rectangle 或区域 region 设置为裁剪区域。枚举 ClipOperation 见下表

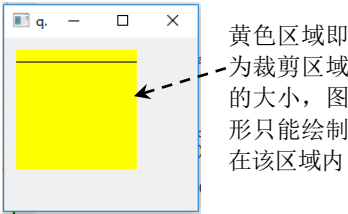
Qt::ClipOperation 枚举(无标志)		
作用：描述裁剪区域的操作方式		
成员	值	说明
Qt::NoClip	0	关闭裁剪
Qt::ReplaceClip	1	替换当前的裁剪区域
Qt::IntersectClip	2	把该裁剪区域与当前裁剪区域相关

- 2)、QRectF **clipBoundingRect**() const;
QPainterPath **clipPath**() const;
QRegion **clipRegion**() const;
以逻辑坐标返回当前的裁剪区域，注意：QPainter 不会显式地存储组合裁剪。
- 3)、bool **hasClipping**() const; //若已启用裁剪，则返回 true。
void **setClipping**(bool enable); //若 enable 为 true，则启用裁剪，否则禁用裁剪

2、裁剪区域的作用是把图形的绘制限制在裁剪区域内，裁剪区域通常没有轮廓线。见以下示例

```
void paintEvent(QPaintEvent *e) {  
    QPainter pr(this);  
    QBrush bs(QColor(255, 255, 1));  
    pr.setBrush(bs);  
    pr.setClipRect(11, 11, 111, 111); //创建区域  
    pr.drawRect(0, 0, 444, 333);  
    pr.drawLine(0, 22, 333, 22); }  

```



二、QRegion 类中的函数

1、QRegion 类主要用于创建裁剪区域，该类与 QRectF 类类似，只是使用该类创建的区域没有轮廓线。

2、构造函数

- 1)、QRegion(); //构造一个空白区域
- 2)、QRegion(int x, int y, int w, int h, RegionType t = Rectangle);

QRegion(const QRect &*r*, RegionType *t* = Rectangle);

构造一个区域, 若 *t* 为 QRegion::Ellipse, 则该区域为椭圆区域, 若 *t* 为 QRegion::Rectangle, 则该区域为矩形区域。

3)、**QRegion**(const QPolygon &*a*, Qt::FillRule *fillRule* = Qt::OddEvenFill);

根据填充规则 *fillRule* 创建一个多边形区域, Qt::FillRule 参见 QPainterPath 类。

4)、**QRegion**(const QBitmap &*bm*); //从位图 *bm* 构造一个区域。

5)、**QRegion**(const QRegion &*r*);

QRegion(QRegion &&*other*);

3、对区域的判断

6)、QRect **boundingRect**() const;

返回此区域的边界矩形。若是空区域, 则返回的矩形是 QRect::isNull()的。

7)、bool **contains**(const QPoint &*p*) const; //若区域包含点 *p*, 则返回 true。

8)、bool **contains**(const QRect &*r*) const; //若区域与矩形 *r* 部分重叠, 则返回 true。

9)、bool **isEmpty**() const; //若区域为空, 则返回 true, 否则返回 false。

10)、bool **isNull**() const; //与 isEmpty()相同。qt5.0

4、组合多个区域

11)、int **rectCount**() const; //返回将在 rects()中返回的矩形数量

12)、QVector<QRect> **rects**() const; //返回组成区域的非重叠矩形的数量。

13)、void **setRects**(const QRect *rects, int number);

使用矩形数组 *rects* 和数量 *number* 设置区域, 这些矩形必须按 Y-X 排序, 且应遵守以下规则

- 矩形不能相交
- 所有具有给定顶部坐标的矩形, 必须具有相同的高度。
- 两个矩形不能水平邻接(此时, 应该把他们组合成一个更宽的矩形)
- 矩形必须按升序排序, 其中 Y 为主排序键, X 为次排序键。

5、区域的集合运算(与 QRectF、QPainterPath 的计算规则相同)

14)、QRegion **intersected**(const QRegion &*r*) const; //返回该区域与区域 *r* 的交集。

QRegion **intersected**(const QRect &*rect*) const; //返回该区域与矩形 *rect* 的交集。

bool **intersects**(const QRegion &*region*) const; //若该区域与 *region* 相交, 则返回 true。

bool **intersects**(const QRect &*rect*) const; //若该区域与 *rect* 相交, 则返回 true。

15)、QRegion **subtracted**(const QRegion &*r*) const; //返回从该区域中减去区域 *r* 后的区域。

void **translate**(int *dx*, int *dy*); //把区域沿 x 轴平移 *dx*, 沿 y 轴平移 *dy*

void **translate**(const QPoint &*point*); //把区域沿 x 轴平移 *point.x()*, 沿 y 轴平移 *point.y()*

QRegion **translated**(int *dx*, int *dy*) const; //返回该区域平移后的副本。

QRegion **translated**(const QPoint &*p*) const; //返回该区域平移后的副本。

16)、QRegion **united**(const QRegion &*r*) const; //返回该区域与区域 *r* 的并集。

QRegion **united**(const QRect &*rect*) const; //返回该区域与 *rect* 的并集。

17)、QRegion **xored**(const QRegion &r) const; //返回该区域与区域 r 的异或运算。

18)、void **swap**(QRegion &other); //将该区域与 other 交换。

6、迭代器相关，以下函数为 qt5.8 引入

19)、const_iterator: 构成区域 QRect 的迭代器

const_reverse_iterator: 构成区域 QRect 的反向迭代器

20)、const_iterator **begin**() const;

const_iterator **cbegin**() const;

按 rects()返回的顺序，返回一个指向组成矩形范围开始的 const_iterator。

21)、const_iterator **end**() const;

const_iterator **cend**() const;

按 rects()返回的顺序，返回一个指向组成矩形范围的末尾的 const_iterator。

22)、const_reverse_iterator **rbegin**() const;

const_reverse_iterator **rbegin**() const;

按 rects()返回的相反顺序，返回一个指向组成矩形范围开始的 const_reverse_iterator。

23)、const_reverse_iterator **rend**() const;

const_reverse_iterator **rend**() const;

按 rects()返回的相反顺序，返回一个指向组成矩形范围末尾的 const_reverse_iterator。

7、以下为重新实现的操作符函数

operator **QVariant**() const;

bool operator!=(const QRegion &other) const;

QRegion operator&(const QRegion &r) const; //交集，比如 r1&r2 与 r1.intersted(r2);相同

QRegion operator&(const QRect &r) const;

QRegion &operator&=(const QRegion &r); //r1&=r2 与 r1=r1.intersected(r2)相同

QRegion &operator&=(const QRect &r);

QRegion operator+(const QRegion &r) const; 并集，r1+r2 与 r1.united(r2);相同

QRegion operator+(const QRect &r) const;

QRegion &operator+=(const QRegion &r);

QRegion &operator+=(const QRect &r);

QRegion operator-(const QRegion &r) const; //相减，r1 - r2 与 r1.subtracted(r2)相同。

QRegion &operator-=(const QRegion &r); //

QRegion &operator-=(const QRegion &r);

QRegion &operator=(QRegion &&other);

bool operator==(const QRegion &r) const;

QRegion operator^(const QRegion &r) const; //异或，r1^r2 与 r1.xored(r2)相同。

QRegion &operator^=(const QRegion &r);

QRegion operator|(const QRegion &r) const; //并集，同 operator +();

QRegion &operator|=(const QRegion &r);

12.12 坐标变(转)换(QTransform 类)

注：本小节需要一些数学知识和计算机图形学方面的知识

一、基本数学知识

1、齐次坐标

- 1)、二维空间中的点可使用(x, y)表示,但在计算机图形学中使用齐次坐标表示点更为方便,齐次坐标把点表示为三元组,即在(x,y)基础上增加一维表示为(x, y, w),其中 w 是一个非零值
- 2)、每个点有很多个不同的齐次坐标表示(只要其中一个是另一个的倍数即可),比如(1, 2, 5), (2, 4, 10), (4, 8, 20);都表示同一个点,通常会使用 w 去除齐次坐标,从而一个点被表示为(x/w, y/w, 1),这样,每一个二维坐标点都可以使用三维的齐次坐标来表示,同理,三维坐标点(x,y,z);可使用四维的齐次坐标表示为(x, y, z, 1);比如(2,5,9)的齐次坐标为(2,5,9,1);
- 3)、使用齐次坐标后,所有的坐标变换公式都可以使用如下矩阵相乘的形式来表示,

$$\begin{bmatrix} x1 & y1 & z1 \end{bmatrix} = \begin{bmatrix} x & y & z \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} =$$
$$\begin{bmatrix} xb_{11} + yb_{21} + zb_{31} & xb_{12} + yb_{22} + zb_{32} & xb_{13} + yb_{23} + zb_{33} \end{bmatrix} \quad (1)$$

其中 $\begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix}$ 被称为**变换矩阵**, 记为 **M**, 因此上式可简化为 **P1=P·M**。

2、坐标变换公式

以下各式中, dx 和 dy 表示平移距离, Sx 和 Sy 表示缩放系数, θ 表示绕原点逆时针旋转的角度, P1(x1, y1)为坐标变换后的点, P(x,y)为变换前的点。

- ①、坐标平移公式: $x1 = x + dx; \quad y1 = y + dy;$ (2)

齐次坐标形式为: $\begin{bmatrix} x1 & y1 & 1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ dx & dy & 1 \end{bmatrix}$ (3)

简记为: **P1 = P · T(dx,dy);**

- ②、坐标缩放公式为: $x1 = x \cdot Sx; \quad y1 = y \cdot Sy;$ (4)

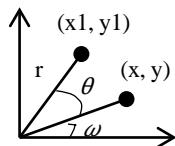
齐次坐标形式为: $\begin{bmatrix} x1 & y1 & 1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} Sx & 0 & 0 \\ 0 & Sy & 0 \\ 0 & 0 & 1 \end{bmatrix}$ (5)

简记为: **P1 = P · S(sx,sy);**

- ③、坐标旋转公式为: $x1 = x \cdot \cos \theta - y \cdot \sin \theta, \quad y1 = x \cdot \sin \theta + y \cdot \cos \theta$ (6)

$$\text{齐次坐标形式为: } \begin{bmatrix} x_1 & y_1 & 1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} \cos \theta & \sin \theta & 0 \\ -\sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}; \quad (7)$$

简记为: $\mathbf{P1} = \mathbf{P} \cdot \mathbf{R}(\theta)$;



$$\begin{aligned} x_1 &= r \cos(\omega + \theta) = r \cos \omega \cos \theta - r \sin \omega \sin \theta \\ y_1 &= r \sin(\omega + \theta) = r \sin \omega \cos \theta + r \cos \omega \sin \theta \\ x &= r \cos \omega; \quad y = r \sin \omega; \quad \text{解得} \\ x_1 &= x \cdot \cos \theta - y \cdot \sin \theta; \quad y_1 = x \cdot \sin \theta + y \cdot \cos \theta; \end{aligned}$$

坐标旋转公式的推导

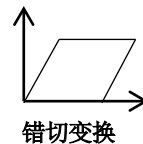
④、对称变换公式为(关于 x 轴, 仅举一例): $x_1 = x; \quad y_1 = -y;$ (8)

$$\text{齐次坐标形式为: } \begin{bmatrix} x_1 & y_1 & 1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (9)$$

⑤、错切(shear)变换公式(效果见图示):

沿 x 方向错切的公式为: $x_1 = cy + x; \quad y_1 = y;$ (10)

$$\text{齐次坐标形式为: } \begin{bmatrix} x_1 & y_1 & 1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ c & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (11)$$



$$\text{沿 y 方向错切的齐次坐标形式为: } \begin{bmatrix} x_1 & y_1 & 1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} 1 & c & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (12)$$

3、坐标的复合变换: 即多个坐标变换序列的组合。

①、连续平移变换(其结果是平移相加)

把点 P 平移至点 P1 再平移至 P2, 则其平移公式的推导步骤为

$$\mathbf{P1} = \mathbf{P} \cdot \mathbf{M1}; \quad \mathbf{P2} = \mathbf{P1} \cdot \mathbf{M2} = \mathbf{P} \cdot \mathbf{M1} \cdot \mathbf{M2};$$

由以上推导过程可见, 对复合的坐标变换序列, 只需计算其变换矩阵即可, 以上规则同样适用于其他更复杂的复合变换, 对于之后的变换, 将只写出变换矩阵的计算。

下面为坐标平移复合变换后的计算过程。

$$\mathbf{P2} = \begin{bmatrix} x_2 & y_2 & 1 \end{bmatrix} =$$

$$\begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ dx_1 & dy_1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ dx_2 & dy_2 & 1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ dx_1 + dx_2 & dy_1 + dy_2 & 1 \end{bmatrix}$$

由结果可见, 连续平移后的结果是平移相加, 连续平移可简写为以下公式

$$\mathbf{P2} = \mathbf{P} \cdot \mathbf{T}(t_{1x}, t_{1y}) \cdot \mathbf{T}(t_{2x}, t_{2y}) = \mathbf{P} \cdot \mathbf{T}(t_{1x} + t_{2x}, t_{1y} + t_{2y}); \quad (13)$$

②、连续缩放变换的结果是缩放相乘(推导略), 即

$$\mathbf{P1} = \mathbf{P} \cdot \mathbf{S}(s_{1x}, s_{1y}) \cdot \mathbf{S}(s_{2x}, s_{2y}) = \mathbf{P} \cdot \mathbf{S}(s_{1x} \cdot s_{2x}, s_{1y} \cdot s_{2y}); \quad (14)$$

③、连续旋转变换的结果是旋转相加(推导略), 即

$$\mathbf{P1} = \mathbf{P} \cdot \mathbf{R}(\theta_1) \cdot \mathbf{R}(\theta_2) = \mathbf{P} \cdot \mathbf{R}(\theta_1 + \theta_2); \quad (15)$$

4、矩阵计算注意事项:

$$M1 \cdot M2 \cdot M3 = (M1 \cdot M2) \cdot M3 = M1 \cdot (M2 \cdot M3); \quad \text{但是} \quad (16)$$

$$M1 \cdot M2 \neq M2 \cdot M1; \quad (17)$$

公式 17 表示坐标复合变换时不能交换其顺序，也就是说先平移再旋转与选旋转再平移是不同的。

- 5、由以上讲解可知，只要给出变换矩阵，便可对其进行坐标变换，因此在计算机编程中，通常只需给出一个变换矩阵，然后给矩阵各元素赋予需要的值便可对其进行各种坐标变换。
- 6、对于三维空间的坐标变换，其原理与二维类似，其坐标变换矩阵为

$$\begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix}$$

可把以上矩阵从功能上划分为多个部分，其中 $\begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix}$ 可产生缩放、旋转、错切

等几何变换， $[b_{41} \ b_{42} \ b_{43}]$ 可产生平移变换， $[b_{14} \ b_{24} \ b_{34}]^T$ 可产生透视变换， $[b_{44}]$ 可产生整体比例变换。

- 7、仿射(affine)变换：平移、旋转、缩放、对称、错切这些变换都是仿射变换的特例，任何仿射变换总可表示为这几种变换的组合。另外还有投影变换，投影变换分为透视变换和平行变换。
- 8、除以上坐标变换之外，还有窗口/视口变换，该内容位于后续内容讲解。

二、使用 QPainter 类中的基本变换函数进行坐标变换

- 1、默认情况下，QPainter 是在自己所关联的绘制设备的坐标系(通常为像素)上运行的，绘制设备默认坐标系的原点位于左上角，X 轴向右增长，Y 轴向下增长。

- 2、QPainter 类中的基本坐标变换函数

以下函数都是在 QPainter 的变换矩阵(QTransform 类)上进行的变换，可使用 QPainter::worldTransform() 函数获取该变换矩阵。QTransform 类详见下文。

- 1)、void QPainter::translate(const QPointF &offset); //平移坐标系。
void QPainter::translate(const QPoint &offset); //平移坐标系。
void QPainter::translate(qreal dx, qreal dy); //平移坐标系。
- 2)、void QPainter::scale(qreal sx, qreal sy); //缩放坐标系
- 3)、void QPainter::rotate(qreal angle); //顺时针旋转坐标系，angle 以度为单位。
- 4)、void QPainter::shear(qreal sh, qreal sv); //使用(sh, sv)错切坐标系。

- 3、绘制状态

- 8)、void QPainter::save();

保存当前绘制器状态(把状态压入堆栈)，包括当前的坐标系。一个 save() 函数必须跟随一个 restore() 函数。

- 9)、void QPainter::restore(); //恢复当前绘制器的状态(从堆栈中弹出保存的状态)。

- 10)、void QPainter::resetTransform();

重置使用以下函数进行的任何转换：translate()、scale()、shear()、rotate()、setWorldTransform()、setViewport()、setWindow()

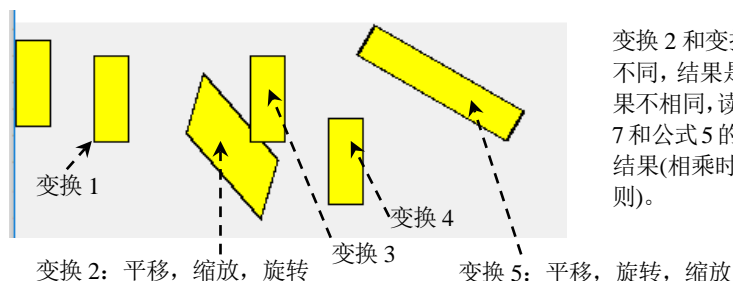
11)、const QTransform &QPainter::deviceTransform() const;

返回从逻辑坐标转换为平台相关绘图设备的设备坐标的矩阵，只有在平台相关句柄 (Qt::HANDLE)上使用平台绘图命令时，才需要此函数。

示例：使用 QPainter 类中的基本坐标变换函数变换坐标

```
void paintEvent(QPaintEvent *e) {
    QPainter pr(this);
    QBrush bs(QColor(255, 255, 1));    pr.setBrush(bs);    QRectF r(0, 11, 22, 55);
    pr.drawRect(r);
    pr.translate(50, 10);    pr.drawRect(r); //变换 1: 平移(50, 10)
    pr.save();                //保存状态
    //变换 2: 平移后位于(100, 60)，缩放(Y向2倍)，旋转(逆时针)
    pr.translate(50, 50);    pr.scale(1, 2);    pr.rotate(-60);    pr.drawRect(r);
    pr.restore();            //恢复状态，此时坐标系位于(50, 10)处。
    pr.translate(100, 0);    pr.drawRect(r);    //变换 3: 平移后位于(150, 10)
    //变换 4: 平移后位于(200, 10)，旋转(逆时针)，缩放(Y向2倍)，注意连续变换与变换 2 的顺序。
    pr.translate(50, 0);    pr.rotate(-60);    pr.scale(1, 2);    pr.drawRect(r);
    pr.resetTransform();      //重置为初始状态
    pr.translate(200, 50);    pr.drawRect(r);} //变换 5: 平移后位于(200, 50);
```

运行结果及说明



变换 2 和变换 5 说明，变换顺序不同，结果是不同的。为什么效果不相同，读者可自行使用公式 7 和公式 5 的变换矩阵相乘查看结果(相乘时注意公式 17 的规则)。

三、使用变换矩阵(QTransform 类)进行坐标变换

1、QPainter 类中与 QTransform 类有关的函数

1)、void QPainter::setWorldMatrixEnabled(bool enable);

若 enable 为 true 则启用世界转换，否则禁用世界转换，世界变换矩阵不会改变。

2)、bool worldMatrixEnabled() const; //若启用了世界转换，则返回 true。

3)、void QPainter::setWorldTransform(const QTransform &matrix, bool combine = false);

void QPainter::setTransform(const QTransform &transform, bool combine = false);

设置世界变换矩阵，若 combine 为 true，则把当前矩阵与 transform 组合，否则 transform 会取代当前矩阵。

4)、const QTransform &QPainter::transform() const; //返回世界变换矩阵

const QTransform &QPainter::worldTransform() const; //返回世界变换矩阵

2、使用 QPainter 类中的基本坐标变换可轻松的实现坐标的变换，下面用示例说明

3、QTransform 类

- ①、QTransform 类主要用于创建一个 3*3 的变换矩阵，该矩阵用于坐标系的 2D 变换。该类取代了 QMatrix 类(此类已过时)。QTransform 类通过操控变换矩阵来实现坐标变换。另外 Transform 类还可对矩阵进行操作，比如可进行矩阵的加、乘等运算，还可对矩阵类型进行判断(比如是否是满秩矩阵等)。
- ②、QTransform 类除了可通过操控其矩阵进行坐标变换外，还可使用 QTransform 类中内置的基本变换函数(比如 QTransform::scale()、QTransform::scale()等)对坐标进行变换，这些函数的使用方法与 QPainter 类中的相应函数是相同的。
- ③、简单的坐标变换完全可使用 QPainter 类中的基本坐标变换函数来完成，使用 QTransform 类可以把多个坐标变换组织在一起，然后在需要时使用。
- ④、QTransform 类的变换矩阵如下

$$\begin{bmatrix} m11 & m12 & m13 \\ m21 & m22 & m23 \\ m31(dx) & m32(dy) & m33 \end{bmatrix}$$

下面为 QTransform 类使用的公式，其中 x1, y1 为变换后的点，x, y 为变换前的点

$$x1 = m11*x + m21*y + dx; \quad (18)$$

$$y1 = m22*y + m12*x + dy; \quad (19)$$

以上两公式用于仿射变换，若不是仿射变换(即投影变换)使用以下公式

$$w1 = m13*x + m23*y + m33; \quad (20)$$

$$x1 = x1 / w1; \quad (21)$$

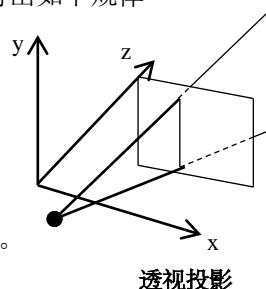
$$y1 = y1 / w1; \quad (22)$$

- 通过对变换矩阵的元素设置不同的值，可以实现不同的坐标变换，比如
 $m21 = m12 = m13 = m23 = 0; \quad m11 = m22 = m33 = 1;$ 则公式 18 和 19 分别变为
 $x1 = m11*x + dx; \quad y1 = m12*x + dy;$ //这就是坐标平移公式。
- 注：公式 18~22 其实就是使用的如下矩阵乘法计算出来的(读者可自行计算)，在没有投影变换的情形下，w1 直接取整数值 1 即可。因此在使用 QTransform 类进行坐标变换时，也可使用如下的矩阵形式计算变换后的坐标值。

$$\begin{bmatrix} x1 & y1 & w1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} m11 & m12 & m13 \\ m21 & m22 & m23 \\ m31(dx) & m32(dy) & m33 \end{bmatrix} \quad (23)$$

- ⑤、把 QTransform 类的变换矩阵与坐标变换矩阵相对比，可得出如下规律

其中 m31 和 m32 用于平移，m11 和 m22 用于缩放，m21 和 m12 用于错切(shear)，m13 和 m23 用于投影变换，m33 是一个额外的投影因子，设置 m11, m12, m21, m22 可实现旋转变换。注：投影变换是一个比较复杂的变换，包括透视投影(右图为透视投影的一个简图)和平行投影，关于投影变换的内容请参阅《计算机图形学》课程。



四、QTransform 类中的函数

1、构造函数

1)、**QTransform**(); //构造一个单位矩阵，即 m11=m22=m33=1;其余元素全为 0。

QTransform(qreal *m11*, qreal *m12*, qreal *m13*, qreal *m21*, qreal *m22*, qreal *m23*, qreal *m31*,
qreal *m32*, qreal *m33* = 1.0);

QTransform(qreal *m11*, qreal *m12*, qreal *m21*, qreal *m22*, qreal *dx*, qreal *dy*);

QTransform(const QMatrix &*matrix*);

QTransform(QTransform &&*other*);

QTransform(const QTransform &*other*);

2、设置和获取变换矩阵的元素

2)、qreal **m11**() const; qreal **m12**() const; qreal **m13**() const;

qreal **m21**() const; qreal **m22**() const; qreal **m23**() const;

qreal **m31**() const; qreal **m32**() const; qreal **m33**() const;

qreal **dx**() const; qreal **dy**() const;

3)、void **setMatrix**(qreal *m11*, qreal *m12*, qreal *m13*, qreal *m21*, qreal *m22*, qreal *m23*, qreal *m31*,
qreal *m32*, qreal *m33*);

4、QTransform 内置的基本坐标变换函数

4)、QTransform &**rotate**(qreal *angle*, Qt::Axis *axis* = Qt::ZAxis);

把坐标系沿轴 *axis* 逆时针旋转角度 *angle*，并返回该矩阵的引用，注意：若将矩阵应用于部件坐标系中的点，则旋转方向为顺序针方向(因为 y 轴是向下的)。Qt::Axis 枚举可取值有：Qt::XAxis, Qt::YAxis, Qt::ZAxis。

5)、QTransform &**rotateRadians**(qreal *angle*, Qt::Axis *axis* = Qt::ZAxis);

该函数与 rotate()函数相同，但角度 *angle* 是以弧度指定的。

6)、QTransform &**scale**(qreal *sx*, qreal *sy*); //缩放坐标系，并返回该矩阵的引用。

static QTransform **fromScale**(qreal *sx*, qreal *sy*); //与 scale();相同，但速度更快。静态的

7)、QTransform &**translate**(qreal *dx*, qreal *dy*); //平移

static QTransform **fromTranslate**(qreal *sx*, qreal *sy*); //与 translate();相同，但速度更快。静态的

8)、QTransform &**shear**(qreal *sh*, qreal *sv*); //错切变换。

5、对变换矩阵的判断

9)、bool **isRotating**() const; //若矩阵表示旋转变换，则返回 true。180 度或 360 度的旋转被视为缩放。

10)、bool **isScaling**() const; //若矩阵表示缩放变换，则返回 true。

11)、bool **isTranslating**() const; //若矩阵表示平移变换，则返回 true。

12)、bool **isAffine**() const; //若矩阵表示的是仿射(affine)变换，则返回 true，否则返回 false。

13)、TransformationType **type**() const;

返回此矩阵的转换类型，注意：返回的值是最大枚举值，比如若矩阵即是缩放又是错切，则返回的值是 TxShear，因为 TxShear 的枚举值比 TxRotate 的枚举值更大。

QTransform::TransformationType 枚举(无标志)

作用：描述变换矩阵的类型

成员	值	说明	成员	值	说明
QTransform::TxNone	0x00	无	QTransform::TxRotate	0x04	旋转
QTransform::TxTranslate	0x01	平移	QTransform::TxShear	0x08	错切
QTransform::TxScale	0x02	缩放	QTransform::TxProject	0x10	投影

6、构建变换矩阵

- 14)、void **reset()**; //把矩阵重置为单位矩阵。
- 15)、static bool **squareToQuad**(const QPolygonF &*quad*, QTransform &*trans*); //静态的
创建一个把单位正方形映射到一个四边多边形 *quad* 的变换矩阵 *trans*。若构造了 QTransform 则返回 true，否则返回 false。
- 16)、static bool **quadToSquare**(const QPolygonF &*quad*, QTransform &*trans*); //静态的
创建一个把四边多边形 *quad* 映射到单位正方形的变换矩阵 *trans*。若构造了 QTransform 则返回 true，否则返回 false。
- 17)、static bool **quadToQuad**(const QPolygonF &*one*, const QPolygonF &*two*, QTransform &*trans*); //静态的
创建一个把四边多边形 *one* 映射到另一个四边多边形 *two* 的变换矩阵 *trans*。若构造了 QTransform 则返回 true，否则返回 false。
- 18)、const QMatrix &**toAffine**() const;
把此矩阵作为仿射矩阵返回，注意：若为透视转换，则转换后将导致数据丢失。

7、与线性代数有关的函数

- 19)、QTransform **adjoint**() const; //返回该矩阵的伴随(adjoint)矩阵
- 20)、qreal **determinant**() const; //返回该矩阵的行列式
- 21)、QTransform **transposed**() const; //返回此矩阵的转置矩阵
- 22)、QTransform **inverted**(bool **invertible* = Q_NULLPTR) const;
返回此矩阵的逆矩阵，若矩阵是奇异的(非可逆的)，则返回的矩阵是单位矩阵，若参数 *invertible* 有效(即不为 0)，则若矩阵可逆，则将其设置为 true，否则将其设置为 false。
- 23)、bool **isIdentity**() const; //若矩阵是单位矩阵，则返回 true。
- 24)、bool **isInvertible**() const; //若矩阵是可逆的，则返回 true。

8、使用变换矩阵转换图形坐标

以下函数用于转换坐标，比如 QPoint p1=matrix.map(point);表示把点使用变换矩阵 *matrix* 进行转换，然后返回该点的副本。等效于 p1= point*matrix;具体应用见后文示例

- 25)、QPointF **map**(const QPointF &*p*) const;
QPoint **map**(const QPoint &*point*) const;
QPoint operator*(const QPoint &*point*, const QTransform &*matrix*);
QPointF operator*(const QPointF &*point*, const QTransform &*matrix*);
- 26)、QLine **map**(const QLine &*l*) const;
QLineF **map**(const QLineF &*line*) const;
QLineF operator*(const QLineF &*line*, const QTransform &*matrix*);
QLine operator*(const QLine &*line*, const QTransform &*matrix*);

27)、QPolygonF **map**(const QPolygonF &*polygon*) const;
 QPolygon **map**(const QPolygon &*polygon*) const;
 QPolygonF operator*(const QPolygonF &polygon, const QTransform &matrix);
 QPolygon operator*(const QPolygon &polygon, const QTransform &matrix);

28)、QRegion **map**(const QRegion &*region*) const;
 QRegion operator*(const QRegion ®ion, const QTransform &matrix);

29)、QPainterPath **map**(const QPainterPath &*path*) const;
 QPainterPath operator*(const QPainterPath &path, const QTransform &matrix);

30)、void **map**(qreal x, qreal y, qreal *tx, qreal *ty) const; //把 x,y 的转换结果保存在 tx 和 ty 中。
 void **map**(int x, int y, int *tx, int *ty) const;

31)、QRectF **mapRect**(const QRectF &*rectangle*) const;
 QRect **mapRect**(const QRect &*rectangle*) const;
 注意：若变换矩阵有旋转或错切，则以上函数返回的是转换后的边界矩形。要获取其准确的矩形，需使用函数 mapToPolygon()。

32)、QPolygon **mapToPolygon**(const QRect &*rectangle*) const;

9、下面为重新实现的操作符函数

```
operator QVariant() const;
bool operator!=(const QTransform &matrix) const;
QTransform operator*(const QTransform &matrix) const; //矩阵相乘
QTransform &operator*=(const QTransform &matrix);
QTransform &operator*=(qreal scalar);
QTransform &operator+=(qreal scalar);
QTransform &operator-=(qreal scalar);
QTransform &operator/=(qreal scalar);
QTransform &operator=(QTransform &other);
QTransform &operator=(const QTransform &matrix);
bool operator==(const QTransform &matrix) const;
```

示例：使用 QTransform 类(变换矩阵)进行坐标变换

```
void paintEvent(QPaintEvent *e) {
    QPainter pr(this);    QBrush bs(QColor(255, 255, 1));    pr.setBrush(bs);
    QRectF r(0, 55, 22, 55);
    //变换 1
    QTransform t(1, 0, 0, 1, 20, 0); //x 方向平移 20
    t.rotate(-60); //使 t 再逆时针旋转 60 度。
    pr.setTransform(t); //设置变换矩阵为 t
    qDebug() << t; //输出内容为: QTransform(type=TxRotate,
    // 11=0.5 12=-0.866025 13=0 21=0.866025 22=0.5 23=0 31=20 32=0 33=1)
    pr.drawRect(r); //绘制矩形
    //变换 2
    QTransform t1; t1.setMatrix(1, 0, 0, 0, 2, 0, 0, 1); //Y 方向放大两倍。
    pr.setTransform(t1); pr.drawRect(r);
    //变换 3
    t1=t1*t; //使用 QTransform 重载的*运算符进行矩阵乘运算，此时 t1 的变换如下：
    //首先向 X 方向平移 20，再逆时针旋转 60 度，最后把 Y 轴方向放大两倍。
    pr.setTransform(t1);
```

```

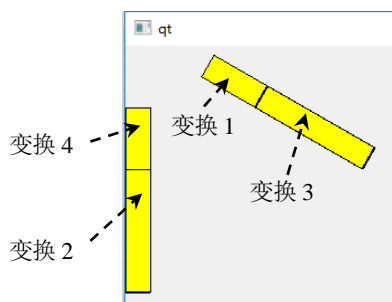
QDebug() << t1;          //输出内容为: QTransform(type=TxShear,
                          // 11=0.5 12=-0.866025 13=0 21=1.73205 22=1 23=0 31=20 32=0 33=1)

pr.drawRect(r);
t1.reset();              //重置矩阵为单位矩阵。
pr.setTransform(t1);     //变换矩阵的值更改后需要使用 QPainter 重新设置变换矩阵,
QDebug() << t1;          //输出内容为: QTransform(type=TxNone,
                          //11=1 12=0 13=0 21=0 22=1 23=0 31=0 32=0 33=1)

pr.drawRect(r);}

```

运行结果及说明

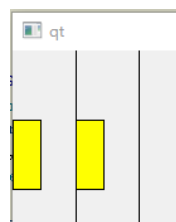


示例: QTransform::map() 函数的使用

```

void paintEvent(QPaintEvent *e) {
    QPainter pr(this);    QBrush bs(QColor(255, 255, 1));
    pr.setBrush(bs);
    QRectF r(0, 55, 22, 55);    QLineF n(50, 0, 50, 555);
    pr.drawLine(n);            pr.drawRect(r);
    QTransform t;              t.translate(50, 0);    //x 方向平移 50
    QRectF r1=t.mapRect(r);    //使用 t 转换 r 的坐标。
    QLineF n1=t.map(n);        //使用 t 转换 n 的坐标
    //n1 也可使用如下等效语句创建, 更简洁
    //QLineF n1=n*t;           //注意 t*n 是错误的
    pr.drawRect(r1);           pr.drawLine(n1);}    //绘制转换后的图形

```

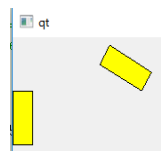


示例: 使用变换矩阵计算变换后的坐标值

```

void paintEvent(QPaintEvent *e) {
    QPainter pr(this);    QBrush bs(QColor(255, 255, 1));    pr.setBrush(bs);
    QRect r(0, 55, 22, 55);
    pr.drawRect(r);
    QTransform t;          t.translate(50, 0);    t.rotate(-60);
    QPolygon g1=t.mapToPolygon(r);    pr.drawPolygon(g1);
    qDebug() << t;          qDebug() << g1;}
    /*依次输出: 1、设置的变换矩阵 t
    QTransform(type=TxRotate, 11=0.5 12=-0.866025 13=0 21=0.866025 22=0.5 23=0 31=50 32=0 33=1)
    2、使用 t 把 r 变换为 g1 后各顶点的坐标值。
    QPolygon(QPoint(98, 28)QPoint(109, 8)QPoint(156, 36)QPoint(145, 55))*/

```



计算步骤如下:

变换矩阵 $t = \begin{bmatrix} 0.5 & -0.866 & 0 \\ 0.866 & 0.5 & 0 \\ 50 & 0 & 1 \end{bmatrix}$, 因此变换后各点的坐标可使用如下矩阵公式计算

$$\begin{bmatrix} x1 & y1 & 1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} 0.5 & -0.866 & 0 \\ 0.866 & 0.5 & 0 \\ 50 & 0 & 1 \end{bmatrix}$$

其中 x, y 为变换前的坐标点, $x1, y1$ 为变换后的坐标点, 把 r 四个顶点的坐标依次带入以上公式便可计算出经过变换后 r 的四个顶点的坐标, 比如 r 左上角的坐标为(0,55)代入后得到

$$\begin{bmatrix} x1 & y1 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 55 & 1 \end{bmatrix} \begin{bmatrix} 0.5 & -0.866 & 0 \\ 0.866 & 0.5 & 0 \\ 50 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 98 & 28 & 1 \end{bmatrix}$$

即左上角变换后的坐标为(98, 28); 其余坐标点的计算原理相同。当然, 也可使用公式 18 和 19 进行计算。

五、窗口视口变换原理及其使用(注: 窗口全称为裁剪窗口)

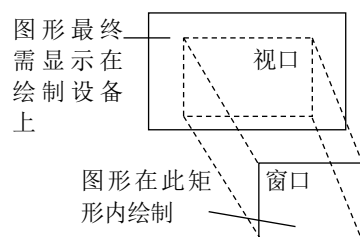
1、需要用到的 QPainter 类中的函数

- 1)、QTransform **combinedTransform()** const; //返回当前窗口/视口和世界变换的变换矩阵组合。
- 2)、void **setViewTransformEnabled**(bool enable); //若 enable 为 true, 则启用视口的转换, 否则禁用。
bool **viewTransformEnabled()** const; //若视口转换已启用则返回 true。
- 3)、QRect **viewport()** const; //返回视口矩形
QRect **window()** const; //返回窗口矩形
- 4)、void **setViewport**(const QRect &rectangle);
void **setViewport**(int x, int y, int width, int height); //设置视口矩形, 并启用视口转换。
- 5)、void **setWindow**(const QRect &rectangle);
void **setWindow**(int x, int y, int width, int height); //设置窗口矩形, 并启用视口转换。

2、理解逻辑: “逻辑”一词具有“理论上的”意思, 说简单一点, 就是“假想的”

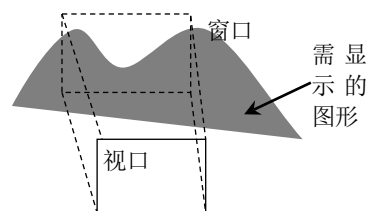
3、从图形绘制的方向理解窗口和视口(原理见图)

窗口就是一个逻辑上的(假想的)矩形, 图形在这个假想的矩形上绘制, 绘制完之后再图形映射到视口上, 视口通常是与需要显示的设备相关联的(即绘制设备, 比如 QWidget 部件, 显示屏等), 然后图形才能显示出来。



4、从图形显示方向理解窗口和视口(见右下图)

假设需要显示图中阴影部分的图形, 则, 首先把图形映射到窗口, 此时可对图形进行旋转、缩放、平移等操作, 然后把超出窗口之外的图形裁剪掉(这就是为什么窗口全称为裁剪窗口的原因), 然后把图形映射到视口显示出来, 若把视口理解为我们的眼睛, 则视口和窗口的概念就很容易理解了。窗口的所有内容只有完全映射到了视口的范围内, 才能被完全显示, 一个窗口的内容可以同时映射到多个视口, 即可以从不同的视口去观察窗口中的内容, 在窗口到视口的映射过程中, 还可对图



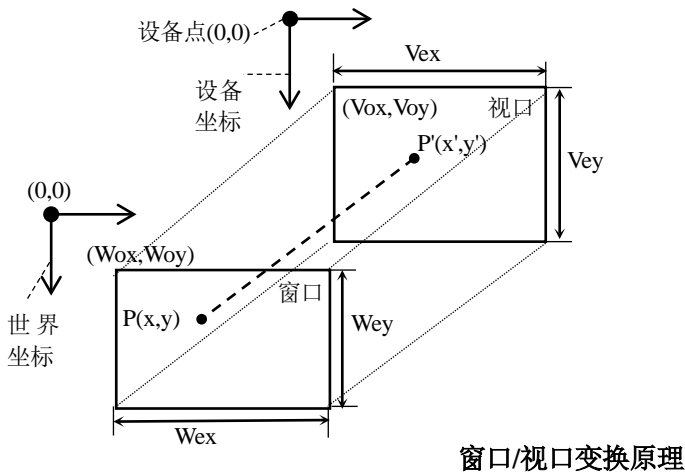
形进行缩放、平移等简单变换。注意：此处讲解的只是一种理论模型，实际实现时是否会裁剪图形，以及坐标变换是在窗口进行还是在视口进行，则视实际使用的绘图软件而不同。

5、窗口/视口主要作用

绘制设备有以像素为单位的(比如显示器)，也有以点为单位的(比如打印机)，使用窗口/视口机制，可使绘图与绘制设备相互独立，我们只需在窗口绘图，而不必关心底层绘制设备到底是什么，因为图形在窗口中绘制，因此窗口是来自真实世界的。

6、窗口/视口变换公式的推导

- 1)、因为计算机中图形的绘制和显示通常都是位于矩形区域之内的，因此窗口/视口变换就是矩形到矩形的变换。
- 2)、窗口/视口变换原理：在窗口中被映射的点(x,y)与原点所围成的矩形区域，与被映射后在视口中的点(x',y')与原点所围成的矩形区域，其所对应的长度之比，应与窗口和视口所形成的矩形的长度之比相等，对于矩形区域的宽度，其原理相同。



3)、为便于讲解引入下表所示变量及概念(意义见图解):

变量名	意义	变量名	意义
(Wox,Woy)	窗口左上角坐标	(Vox,Voy)	视口左上角坐标
Wex	窗口水平宽度	Vex	视口水平宽度
Wey	窗口垂直宽度	Vey	视口垂直宽度
(x,y)	窗口中被映射的点的坐标	(x',y')	被映射后在视口中的点的坐标
窗口/视口范围：把窗口/视口的水平宽度和垂直宽度称为窗口/视口范围，比如窗口范围就是指的 (Wex,Wey)，视口范围就是指的 (Vex,Vey)			

4)、根据变换原理，变换后的点(x',y')可由如下公式计算得出(见图):

①、 $(x'-Vox)/(x-Wox)=Vex/Wex$ $(y'-Voy)/(y-Woy)=Vey/Wey$ (24)

②、公式 24 移项后得到

$x'=(x-Wox)*Vex/Wex+Vox$ $y'=(y-Woy)*Vey/Wey+Voy$ (25)

其中 Vex/Wex 和 Vey/Wey 是坐标映射时 x 方向和 y 方向的缩放系数

③、公式 25 展开后得到

$$x' = x * Vex / Wex + Vox - Wox * Vex / Wex \quad y' = y * Vey / Wey + Voy - Woy * Vey / Wey \quad (26)$$

④、若令 $Vex/Wex=S$, $Vox-Wox*Vex/Wex=m$, $Vey/Wey=K$, $Voy-Woy*Vey/Wey=n$, 其中 S 和 K 是缩放系数, 则公式 26 简化为如下公式:

$$x' = x * S + m \quad y' = y * K + n \quad (\text{这就是数学上的坐标转换公式}) \quad (27)$$

7、理解窗口/视口变换公式

- 1)、视口中的设备就是指的实际的绘制设备, 设备坐标和设备点是不会变的, 其位置通常被设置为绘制设备的左上角, 方向是 x 轴从左向右, y 轴从上到下。在视口中的坐标值 x' , y' , Vox , Voy 都是相对于设备坐标而言的, 因此使用公式计算出来的坐标值 x' , y' 是相对于设备坐标而言的, 因为窗口通常来自于现实世界, 因此窗口中的坐标值是相对于世界坐标而言的。
- 2)、映射公式使用的数值是没有单位的, 不存在某个数表示多少毫米, 多少像素这一说法, 计算时只需把相应的数值代入公式就能计算出正确的结果

8、窗口/视口变换可实现以下坐标变换

- 1)、注意: 以下的窗口和视口范围使用的数值并不是真实的数值, 因为在坐标转换公式中, 使用的是缩放系数(即他们的比值), 因此使用真实数值并没有多大意义。
- 2)、相等变换: 即 $x'=x$; $y'=y$ 的情形, 此时视口和窗口的原点重合, 视口和窗口的缩放系数为 1。即 $(Wox, Woy) = (Vox, Voy) = (0, 0)$; $(Wex, Wey) = (Vex, Vey) = (1, 1)$;
- 3)、平移变换: 即 $x'=x+m$; $y'=y+n$ 的情形, 此时视口和窗口的缩放系数为 1。即 $S=K=Vex/Wex=Vey/Wey=1$, 或 $(Vex, Vey) = (Wex, Wey) = (1, 1)$; 因此 $m=Vox-Wox*Vex/Wex=Vox-Wox$; $n=Voy-Woy*Vey/Wey=Voy-Woy$; 由此可得出平移公式为: $x'=x+Vox-Wox$; $y'=y+Voy-Woy$;
- 4)、缩放变换: 即 $x'=x*S$; $y'=y*K$ 的情形, 此时窗口和视口原点坐标为 $(0, 0)$;
- 5)、对称变换: 若图形从水平方向翻转, 则 x 方向的缩放系数 S 为负数, 若图形从垂直方向翻转, 则 y 方向的缩放系数 K 为负数, 可见缩放系数的正负, 决定着坐标轴是否反向。
- 6)、以上各种变换的合并变换。

9、窗口/视口数据的单位(逻辑单位与设备单位)

- 1)、窗口和视口的数据在数学上来讲是没有什么实际意义的, 数据就是一个数值, 但在现实世界中绘制图形时, 需要知道图形在绘图区域的具体位置, 因此需要知道给定的数据具体表示多少个单位的距离, 这样就需要给数据指定一个单位。比如, 视口中的数据以“像素”为单位, 窗口中的数据以“毫米”为单位等。
- 2)、视口数据的单位: 来自视口的数据, 一般都以实际的物理单位为本位的, 比如若视口为显示屏, 若显示屏以像素为本位, 则数据 100 表示位于显示屏 100 个像素处的实际位置, 若显示屏以毫米为本位, 则数据 100 表示位于显示屏 100 毫米处的实际位置。若视口中数据的意义未确定, 则使用“设备单位”表示视口中数据的单位, 比如 100, 表示 100 设备单位。
- 3)、窗口数据的单位: 窗口的数据是实际绘制时希望使用的数据, 因实际绘制时可能会使用多种不同的单位, 因此在未明确窗口中数据的单位时, 使用“逻辑单位”表示窗口数据的单位。比如若想以毫米为本位绘制图形, 则 1 逻辑单位就是 1 毫米, 若想以 0.1 毫

米的精度绘制图形，则1逻辑单位就是0.1毫米。

- 4)、由以上可知，若想让绘制设备以实际窗口绘制时使用的逻辑单位为单位来显示图形，则在设备单位和逻辑单位之间需要一个单位转换，而这种转换可以使用窗口/视口变换来实现(具体见后文)。
- 5)、数据的来源：由公式可知， x' ， y' ， Vex ， Vey ， Vox ， Voy 是来自视口的数据，因此这些数据是以设备单位为本位的， x ， y ， Wex ， Wey ， Wox ， Woy 是来自窗口的数据，这些数据是以逻辑单位为本位的。

10、视口/窗口范围比 Vex/Wex (缩放系数 S)与单位转换

对于缩放系数 $Vey/Wey=K$ 的原理是相同的，仅以 S 为例讲解。

- 1)、实际绘制图形时，我们只需指定绘制时的单位，而不需关心物理设备究竟使用什么单位，比如想绘制100毫米长的直线，只需在绘制时直接输入100就能在物理设备上显示100毫米这么长的直线，而不是显示100像素或100点大小这么长的直线。本小段将会讲解使用窗口/视口变换怎样实现这种功能。
- 2)、虽然使用窗口/视口变换也可实现一些简单的坐标变换，但通常使用窗口/视口进行单位转换以实现在窗口中绘制的图形与现实世界相同，当然单位转换也可使用 $QTransform$ 变换矩阵来完成。
- 3)、视口/窗口范围比在数据无任何单位时，就仅仅表示缩放系数，即 $Vex/Wex = S$ 。
- 4)、视口/窗口范围比 S 的意义一般根据视口所表示的实际意义而定，下面以窗口的数据为未确定的逻辑单位，然后再依视口数据的单位不同，而分别对 S 的单位进行推导
 - ①、若视口的数据使用像素作为单位，则视口/窗口范围比 $Vex/Wex=S$ 表示“1个逻辑单位中的像素个数”，换句话说，就是1逻辑单位有 S 个像素，使用数学形式就是： S 像素/逻辑单位，即缩放系数 S 的单位为“像素/逻辑单位”，比如100逻辑单位就表示有“ $100*S$ ”个像素
 - ②、同理，若视口的数据使用毫米为单位，则缩放系数 S 的单位为“毫米/逻辑单位”表示1逻辑单位有 S 毫米。比如100逻辑单位就表示有 $100*S$ 毫米
 - ③、若数据各自拥有单位，则在使用坐标转换公式计算时，需要注意各数据的单位应一致，否则可能会导致不一致的结果。比如，已知缩放系数 $S = 0.1\text{mm}/\text{逻辑单位}$ ，即 $S = 0.1$ ，若 $Wox = 0$ ， $Vox = 100$ 像素， $x = 100$ 逻辑单位，则按公式 $x' = x * Vex/Wex + Vox - Wox * Vex/Wex = x * S + Vox - Wox * S = 100 * 0.1 + 100 = 110$ 在数据无单位的情况下是正确的，但在各数据有单位时，这是错误的，因为最后的结果110，即不能表示110毫米，也不能表示110像素。上式中可把 Vox 的值转换为以毫米为单位，代入公式再进行计算，计算的结果为毫米，也可把 x 、 S 、 Wox 转换为以像素为单位进行计算，计算的结果为像素，在数据有单位时，使用坐标转换公式时的单位一定要一致。在计算机中，像素可以转换为毫米、英寸等单位，因此容易出现这种错误。
 - ④、由以上讨论可知，缩放系数 S 就是表示的一个逻辑单位所对应的物理单位，使用数学公式就是

$$Vex/Wex = S \text{ 物理单位/逻辑单位}$$

因此在窗口中的1个逻辑单位，映射到视口上时，就以 S 个物理单位进行显示，比如，若 $S = 0.1 \text{ mm}/\text{逻辑单位}$ ，则窗口中的1逻辑单位，将在视口上以实际的0.1mm

进行显示。

11、单位转换示例

例1: 假设想以1逻辑单位表示0.1毫米, 屏幕的分辨率为1024像素, 屏幕实际长度为376mm, 若客户区的长度为500像素, 试确定窗口范围的尺寸, 注意: 屏幕是物理设备, 是以像素为单位来绘制图形的。

解: 主要应注意单位间的转换, 本示例只计算了X方向的情形, Y方向的情形未予计算。

方法一: 把单位转换为像素

由题知, 1逻辑单位=0.1mm, 即 $S = 0.1\text{mm} / \text{逻辑单位}$

屏幕的实际分辨率为: $X = 1024 / 376 = 2.7234 \text{ 像素/mm}$,

把S转换为以像素为单位

$S = 0.1 \text{ mm} / \text{逻辑单位} \times 2.7234 \text{ 像素/mm} = 0.27234 \text{ 像素/逻辑单位}$,

即1逻辑单位有0.27234个像素

因此 $Vex / Wex = S = 0.27234 \text{ 像素/逻辑单位}$, 又因 $Vex = 500 \text{ 像素}$, 所以

$Wex = 500 / 0.27234 = 1836 \text{ 逻辑单位}$

方法二: 把单位转换为毫米

由方法一知, 屏幕实际分辨率为2.7234 像素/mm

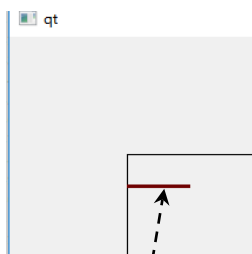
客户区的长度为500像素, 即 $Vex = 500 \text{ 像素} = 500 / 2.7234 \text{ mm} = 183.6 \text{ mm}$,

由题知, 1逻辑单位=0.1mm/逻辑单位, 即 $S = 0.1$

因此 $Vex / Wex = 183.6 / Wex = S = 0.1$, 推得 $Wex = 1836 \text{ 逻辑单位}$ 。

示例代码如下:

```
void paintEvent(QPaintEvent *e) {
    QPainter pr(this);
    QBrush bs(QColor(255, 255, 1));
    //绘制一个与视口一样大小的矩形, 以便于观察
    pr.drawRect(100, 100, 500, 500);
    QPen pn;
    pn.setColor(QColor(111, 1, 1));
    pn.setWidth(10);
    pn.setCapStyle(Qt::FlatCap);
    pr.setPen(pn);
    pr.setViewport(100, 100, 500, 500);
    pr.setWindow(0, 0, 1836, 1836);
    //注意: 在设置窗口/视口之后, 使用 drawXXX 绘制
    //图形时使用的坐标, 都需要使用公式 26 进行计算。
    pr.drawLine(0, 100, 200, 100); }
```



若您的显示器符合本示例的要求, 则该直线将只有 20 毫米长, 而不是 200 个像素的长度

12、由以上讲解可知, 通过计算窗口/视口变换可实现绘制图形时以毫米、厘米等现实世界的单位为单位, 下面再讲解一个使用窗口/视口变换实现移动绘制设备坐标的方法

1)、绘制设备的坐标默认位于左上角, 但可通过窗口/视口变换在逻辑上来移动该坐标系到我们想要的位置。具体方法如下:

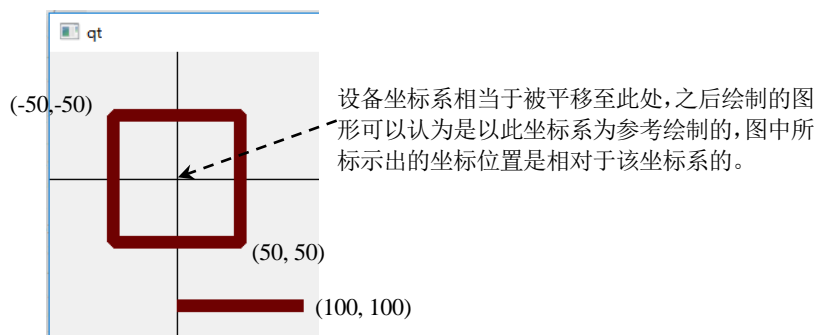
2)、使 $S = K = 1$, 即 $Vex = Wex$; $Vey = Wey$; 也就是窗口和视口具有相同的宽度和长度, 这意味着窗口和视口拥有相同的长度单位。此时, 公式26变为

$x' = x + Vox - Wox$; $y' = y + Voy - Woy$;

以上公式是一个坐标平移公式，该公式不但是一个平移点的公式，也是平移坐标系的公式。因此该公式相当于把坐标系向x方向平移了 $V_{ox} - W_{ox}$ 的距离，向y方向平移了 $V_{oy} - W_{oy}$ 的距离。在之后绘制图形时可以参考该坐标系进行绘制。若绘图软件不裁剪图形，则可简单的把窗口/视口范围设置为1，即 $V_{ex} = W_{ex} = V_{ey} = W_{ey} = 1$ ；下面以示例说明。

```
void paintEvent(QPaintEvent *e) {
    QPainter pr(this);
    QBrush bs(QColor(255, 255, 1));
    //以下绘制的交叉直线用于标示平移后的坐标位置，以方便观察
    pr.drawLine(0, 100, 555, 100);    pr.drawLine(100, 0, 100, 555);
    QPen pn;    pn.setColor(QColor(111, 1, 1));    pn.setWidth(10);
    pn.setCapStyle(Qt::FlatCap);    pr.setPen(pn);
    //设置窗口/视口，以下代码相当于把设备坐标移至(100, 100)处。
    pr.setViewport(50, 50, 200, 200);    pr.setWindow(-50, -50, 200, 200);
    pr.drawLine(0, 100, 100, 100);    pr.drawRect(-50, -50, 100, 100);}
```

运行结果及说明



12.13 绘制图像

(QImage、QPixmap、QBitmap、QPImage)

一、图像基础知识

1、分辨率(DPI 和 PPI):

1)、DPI 和 PPI: DPI 全称是 Dots Per Inch(点每英寸), PPI 全称是 Pixel Per Inch(像素每英寸), 即指的是每一英寸的长度上有多少个点(或像素)。DPI 和 PPI 越高, 显示就越清楚细腻, DPI 通常用于描述打印机或者扫描仪等的分辨率, 而 PPI 主要用于描述显示器的分辨率, 平常使用中经常混用 PPI 和 DPI 的概念, 但他们还是有一些区别, 因为像素和点代表的长度有可能并不是一样长的, 对于显示器通常使用 PPI, 比如 1024PPI, 是指每英寸有 1024 个像素, 当然, 也可对显示器使用 DPI, 比如 1024DPI 这时表示一英寸有 1024 个点, 这里一个像素和一个点具体实际代表多少长度(比如多少毫米)是需要通过计算的, 具体见下文对像素和点的讲解。

2)、像素(pixel): 像素不是长度单位, 二者不能等同, 像素表示的是一个图像的最小单位, 比如室外大屏幕一个像素可能是 1 到 2 厘米长度, 而电脑显示屏一像素大约是 0.2~0.4 毫米长度。比如 1024*768 分辨率(表示长度方向有 1024 个像素, 宽度方向有 768 个像素), 显示屏为 17 英寸的显示器(不含边框), 其 PPI 为 $[(1024*1024+768*768)^{1/2}]/17=75.2\text{PPI}$, 其一个像素的长度大约为 $25.4/75.2=0.3377$ 毫米。注: 1 英寸=25.4 毫米, 显示器的尺寸指的是显示器对角线的长度(注意, 通常是包括显示器的不可显示边框的), 所以本例对分辨率使用勾股定理求出斜边像素数, 再除以显示器尺寸就是显示器的 PPI。

3)、点(磅): 在计算机中, 一点的大小大约是 1/72 英寸, 1 磅就是指的 1 点的大小。

2、以下概念来自移动设备编程

1)、设备像素(物理像素): 就是显示设备的真实像素, 这是屏幕的固有属性, 不会改变。

2)、设备独立像素 DIP(Device Independent Pixels): 又称设备无关像素或密度独立性, 在移动设备(比如手机)中, 图形被显示在不同密度(分辨率)的屏幕上, 这时因不同移动设备的密度并不一致, 会导致在低密度的屏幕上使图形看起来更大, 而在高度密的屏幕上看起来更小, 为解决这个问题, 引入了设备独立像素, 或使用设备独立像素为单位绘制图形, 则可以保证图形在不同密度的屏幕上其大小看起来差不多一致。DIP 是由系统设置的, 通常系统设置有多个 DIP 值, 比如有 120 ppi、160ppi、240ppi 等, 通常使用 160ppi。

3)、设备像素比(dpr): 就是设备像素与设备独立像素的比值, dpr 通常也由系统设置。

4): 理解 DIP: DIP 的本质就是单位换算的问题, 下面以示例说明

示例: 若系统设置的 DIP 为 M 像素/英寸(ppi), 设备像素为 N 像素/英寸, 假设我们绘制 X 像素长度的图形, 若使用 DIP 绘制图形, 求显示屏实际显示的像素 Y 为多少。

- 所绘制图形的逻辑长度: $L = (X \text{ 像素}) / (M \text{ 像素/英寸}) = X / M$ 英寸;

以上公式要注意的是 X / M 结果的单位是英寸。

- 因此显示屏实际显示的像素为： $Y = (X / M) * N = X * (N / M) = X * \text{dpr}$ 像素
代入实际数据进行计算，

假设 DIP 为 160ppi，设备像素为 160 ppi，绘制的长度 X 为 100 像素，则在显示屏上实际显示的像素长度为：

$$100 * (160/160) = 100 \text{ 像素，其长度与逻辑长度相同。}$$

若设备像素为 320ppi(即分辨率更高)，其余不变，则实际显示的像素长度为：

$$100 * (320/160) = 200 \text{ 像素，}$$

我们还能计算出显示屏上显示的实际物理长度为：

$$200 \text{ 像素} / (320 \text{ 像素/英寸}) = 0.625 \text{ 英寸} = 0.625 * 25.4 \text{ 毫米} = 15.875 \text{ 毫米。}$$

由以上计算可见当设备像素增长或减小时，dpr 会随之增加或减小，这样就保证了所绘图形在不同分辨率的设备上显示的大小基本一致。

3、颜色基础知识

- 1)、alpha 通道用于记录图像的透明度信息，通常使用 8 位(共有 256 种级别)表示 alpha 通道，0 表示透明，255 表示不透明。

- 2)、RGB 颜色模型：

自然界中的所有颜色都可由红(R)、绿(G)、蓝(B)三种颜色组成。

计算机中，通常把红(R)、绿(G)、蓝(B)三种颜色中的每一种使用 8 位进行存储，RGB 三种颜色就需要使用 24 位(3 字节)进行存储，这样每种颜色就有 0~255 共 256 种状态，不同的红绿蓝就可以组合出 1600 多万(2 的 24 次方)种颜色，若在 24 位的基础上再增加一个表示图像透明度的 alpha 通道，假设 alpha 使用 8 位来存储，那么 ARGB 共 32 位就可以组合出更多的颜色。除了 RGB 颜色模型外还有 CMY(青色、洋红、黄色)、CMYK、HSI(色调、饱和度、亮度)、HSV 等颜色模型。

- 3)、颜色深度(简称深度)：就是指一个像素使用多少位来表示颜色，比如红绿蓝各使用 8 位共 24 位来表示一个像素的颜色，则颜色深度就是 24。因此对于 24 位深度的颜色，存储一个像素的颜色信息就需要 3 个字节的空间。颜色深度也被称为“位/像素(bpp)”。
- 4)、位平面：在计算机中设有专门用于存储图像信息的帧缓存存储器，该存储器中的每一位对应于屏幕上的一个点，当一个位上的点被设置为 1 时，屏幕上对应位置就出现一个亮点，当一个位的值为零时，则出现一个暗点。当显示器的分辨率为 640*480 时，为显示一个单色图像(深度为 1)需要 640*480 位的帧缓存存储器容量，这个容量被称为一个位平面，同理，若需要显示一个 8 位深度的图像，则需要有 640*480*8 位帧缓存的容量，即需要 8 个位平面，从此处可见位平面通常与颜色深度的值相同，但也有例外。对于更高的分辨率和显示更高深度的图像，则需要更多的帧缓存容量。
- 4)、通道是用于描述每个图像颜色的信息的，比如对于 RGB 模式的图像，共有 3 个通道，即红色通道、绿色通道、蓝色通道，若该图像还包含 alpha 通道信息，则该图像就有 4 个通道。对于 CMYK 模式的图像，有 4 个通道，分别为青色、洋红、黄色、黑色通道。
- 5)、灰度：是指黑白图像中的颜色深度，其范围通常使用 0~255 来表示，其中白色为 255，黑色为 0。对于 RGB 模式的彩色图像则 RGB 三种颜色的值相等时就是灰度色彩模式。

- 9、图像抖动(dithering)技术：该技术主要用于把高深度的图像转换为低深度图像。比如，老式的针式打印机只有黑点和白点，那么怎样打印具有灰度级别的黑白图片呢？抖动技术就

是用于解决这个问题的,抖动的基本原理是当人们以足够远的距离观察一个很小的区域时,眼睛会对区域内的细节进行平均,从而只能看到区域的总体宽度。在一个区域内黑点越多看起来就越黑,黑点越少看起来就越亮,因此通过适当的控制区域内的黑点数量及分布就能使图片呈现出不同的灰度级别,当然,这有一个专门的抖动算法(本文不讨论)。使用抖动可把灰度图像或彩色图像处理成二值图像。在彩色图像与彩色图像之间的转换(比如把 24 位的图像转换为 8 位等)也可以使用抖动技术。

10 图像的格式:

1)、像素格式:是指像素数据存储时所使用的格式,定义了像素在内存中的编码方式,注意:在实际存储图像时,对于 RGB 模式或其他颜色模式并不一定会为每种颜色都分配 8 位的存储空间,比如对于 32 位含有 alpha 通道的 RGB 模式,有可能会为红、绿、蓝各分配 10 位,而为 alpha 通道分配 2 位,当然也有为 alpha 和红、绿、蓝各分配 8 位的存储方式。

2)、查色表、颜色表、调色板、索引像素格式(简称索引格式):

对于一些包含很少几种颜色的图像,可以为每种所出现的颜色构造一张表(称为查色表或颜色表,也被称为调色板),在存储图像时只需为每种颜色存储查色表的索引即可,这样可以压缩图像的大小,使用这种方法存储图像的文件,称为索引像素格式文件,常见的文件有 GIF 和 PNG。但是颜色种类太多的图像(比如 24 位的 RGB 色),则不适合这种方法存储。比如,对于大小为 $100*100$ 具有 4 位深度的图像,若使用 RGB 模型来存储,则每个颜色分量(RGB 三色中的其中一种色)需要使用 8 位存储,因此每个像素需要 3 字节存储空间,存储该图像就需要 $100*100*3 \approx 30\text{KB}$ 的存储空间,若为 16 种颜色创建一个颜色表,则每个像素的颜色只需查找颜色表的索引即可,这样每个像素就只需 4 位(16 种状态需要 4 位)的存储空间了,该图像像素占据的存储空间为 $100*100*0.5 \approx 5\text{KB}$,颜色表的大小为 $3\text{KB}*16 = 48\text{KB}$ (注意:16 种颜色中的每种颜色仍需要 24 位来描述),总大小约为 53KB,比之前节约了为 47KB 的空间。对于具有 24 位深度的图像使用颜色表的方法存储,不但节约不了空间,反而还会使空间增大。

二、绘制图像基础

1、Qt 提供了 4 个类来处理图像: QImage、QPixmap、QBitmap、QPicture,为了对这几个类加以区分,分别称 QImage 为图像、QPixmap 为像素图、QBitmap 为位图、QPicture 为图片。

2、Qt 各个处理图像类的区别及作用

①、QImage 类提供了一个与硬件无关的图像表示方法,可以直接访问和操控像素,也就是说该类可修改或编辑图像的像素。该类还可以用于进行 I/O 处理,并对 I/O 处理操作进行了优化

②、QPixmap 类主要用于在屏幕上显示图像, QPixmap 中的像素数据是由底层窗口系统进行管理的,该类不能直接访问和操控像素,只能通过 QPainter 的相应函数或把 QPixmap 转换为 QImage 来访问和操控像素。 QPixmap 可通过标签(QLabel 类)或 QAbstractButton 的子类(icon 属性)显示在屏幕上。

③、QBitmap 是 QPixmap 的子类,用于处理颜色深度为 1 的图像,即只能显示黑白两种颜色。

- ④、QPicture 用来记录并重演 QPainter 命令，QPicture 与分辨率无关，可在不同设备上显示。该类使用一个与平台无关的格式(.pic 格式)把绘图命令序列化到 I/O 设备，所有可绘制在 QWidget 部件或 QPixmap 上的内容都可以保存在 QPicture 中，该类的主要作用是把一个绘制图设备上使用 QPainter 绘制的所有图形保存在 QPicture 之中，然后再把这些图形重新绘制在其他绘图设备上。
- 3、通常，可以使用 QImage 类来加载并操作图像数据，然后把 QImage 对象转换为 QPixmap 再显示到屏幕上，若不需对图像进行操作，也可直接使用 QPixmap 来加载并显示图像。
- 4、QImage、QPixmap、QBitmap、QPicture 都是 QPaintDevice 类的子类(直接或间接)，因此他们都是绘制设备，可以直接在其上进行图形绘制。

示例：图像绘制

```
#ifndef M_H
#define M_H
#include<QtWidgets>
class B:public QWidget{    Q_OBJECT
public:    QPushButton *pb1;    QPicture pc;    QPixmap pml;    QImage pi;
    B(QWidget *pl=0):QWidget(pl) {
        pb1=new QPushButton("AAA",this);    pb1->move(22,111);
        pb1->setShortcut(QKeySequence("Ctrl+F"));
        QObject::connect(pb1,&QPushButton::clicked,this,&B::f1);}
    void paintEvent(QPaintEvent *e) {
        QPainter pr;
        //把图像绘制到QWidget 部件上。函数 f1() 只是创建了图像中的内容，但图像需要被显示出才会可见
        //比如设置为标签或按钮的图标，或者像本示例这样直接绘制出来
        pr.begin(this);
        pr.drawPixmap(11,11,pml);    pr.drawImage(133,11,pi);    pr.drawPicture(244,11,pc);}

public slots:
    void f1() {    QPainter pr;    QBrush bs(QColor(255,255,1));
        /* QPixmap、QImage、QPicture 都是绘制设备，可在其上直接绘制图形，因为这些绘制设备都不是 QWidget
        部件，因此可以不在 paintEvent() 函数中绘制。*/
        pml.load("F:/1s.png");    //加载一幅图片
        pr.begin(&pml);    //开始在 pml 上绘制图形
        //pml.load("F:/1s.png");    //注意：load() 最好位于 begin() 之前。
        pr.drawLine(0,0,111,111);
        pr.end();

        pi=QImage(QSize(111,111),QImage::Format_RGB32);
        pr.begin(&pi);    pr.setBrush(bs);    pr.drawRect(11,11,122,122);    pr.end();

        pr.begin(&pc);
        pr.setBrush(bs); //调用 begin() 会重置 QPainter 为默认状态，因此需重新设置画刷。
        pr.drawRect(11,11,99,99);    pr.drawArc(11,11,111,111,33*16,144*16);    pr.end();
        repaint();    //立即重绘界面
    };
};
#endif // M_H

//m.cpp 文件内容
#include "m.h"
```

```
int main(int argc, char *argv[]) {    QApplication app(argc, argv);
    B w;    w.resize(444, 333);    w.show();    return app.exec(); }
```

运行结果及说明



三、使用 QPainter 类中绘制图像的函数

1、需要使用到的 QPainter 类中的函数原型如下：

- 1)、void drawImage(const QRectF &rectangle, const QImage &image)
- void drawImage(const QRect &rectangle, const QImage &image)
- void drawImage(const QPointF &point, const QImage &image)
- void drawImage(const QPoint &point, const QImage &image)
- 以上函数表示在位置 point 或矩形 rectangle 内绘制图像 image。若图像和矩形大小不一致，图像将缩放以适应其大小
- 2)、void drawImage(const QRectF &target, const QImage &image, const QRectF &source, Qt::ImageConversionFlags flags = Qt::AutoColor)
- void drawImage(const QRect &target, const QImage &image, const QRect &source, Qt::ImageConversionFlags flags = Qt::AutoColor)
- void drawImage(const QPointF &point, const QImage &image, const QRectF &source, Qt::ImageConversionFlags flags = Qt::AutoColor)
- void drawImage(const QPoint &point, const QImage &image, const QRect &source, Qt::ImageConversionFlags flags = Qt::AutoColor)
- void drawImage(int x, int y, const QImage &image, int sx = 0, int sy = 0, int sw = -1, int sh = -1, Qt::ImageConversionFlags flags = Qt::AutoColor)

以上函数的主要作用是把图像 image 的一部分图像(由 source 指定)绘制到绘制设备的位置 point 或矩形 target 内。若图像和矩形大小不一致，图像将缩放以适应其大小。参数 flags 用于指示图像怎样从高分辨率转换为低分辨率。Qt::ImageConversionFlag 枚举见下表

Qt::ImageConversionFlag 枚举		
标志：Qt::ImageConversionFlags		
作用：描述图像的转换标志，该枚举会应用于整本小节的整篇内容		
成员	值	说明
以下标志用于单色图像		
Qt::AutoColor	0x0000 0000	若图像深度为 1(即 1 位)，且仅含有黑白像素，则像素图变为黑白图像。默认

Qt::ColorOnly	0x0000 0003	像素图被抖动转换为本机显示屏深度
Qt::MonoOnly	0x0000 0002	像素图变为单色，若有必要使用抖动算法进行抖动。
以下标志为抖动模式首选项		
Qt::DiffuseDither	0x0000 0000	使用误差扩散的高质量抖动。 默认。
Qt::OrderedDither	0x0000 0010	更快的，有序的抖动。
Qt::ThresholdDither	0x0000 0020	不抖动，使用最接近的颜色。
1 位 alpha 模板(mask)的抖动模式首选项		
Qt::ThresholdAlphaDither	0x0000 0000	无抖动(默认)
Qt::OrderedAlphaDither	0x0000 0004	更快，有序的抖动。
Qt::DiffuseAlphaDither	0x0000 0008	使用误差扩散的高质量抖动
颜色匹配与抖动首选项		
Qt::PreferDither	0x0000 0040	转换为较小的色彩空间时，始终使用抖动。
Qt::AvoidDither	0x0000 0080	当源图像使用比目标格式的颜色表的大小更多的不同颜色时，则仅对索引格式使用抖动。
Qt::AutoDither	0x0000 0000	仅在向下转换为 1 或 8 位索引格式时才抖动。 默认。
Qt::NoOpaqueDetection	0x0000 0100	不检查图像是否包含非透明像素，若已经明确的知道图像是非透明的，则可以使此标志。若图像没有 alpha 通道，则此标志不起作用。
Qt::NoFormatConversion	0x0000 0200	不对图像进行任何格式转换，比如把 QImage 转换为 QPixmap 以用于一次性渲染时，会非常有用。

- 3)、void **drawPicture**(const QPointF &*point*, const QPicture &*picture*);
void **drawPicture**(const QPoint &*point*, const QPicture &*picture*);
void **drawPicture**(int *x*, int *y*, const QPicture &*picture*);
在位置 *point* 重演图片 *picture*。若 *point* = QPoint(0, 0);则以上函数与 QPicture::play()完全相同。
- 4)、void **drawPixmap**(const QPointF &*point*, const QPixmap &*pixmap*);
void **drawPixmap**(const QPoint &*point*, const QPixmap &*pixmap*);
void **drawPixmap**(int *x*, int *y*, const QPixmap &*pixmap*);
void **drawPixmap**(const QRect &*rectangle*, const QPixmap &*pixmap*);
void **drawPixmap**(int *x*, int *y*, int *width*, int *height*, const QPixmap &*pixmap*);
以上函数表示在位置 *point* 或矩形 *rectangle* 内绘制像素图 *pixmap*。若图像和矩形大小不一致，图像将缩放以适应其大小
- 5)、void **drawPixmap**(const QRectF &*target*, const QPixmap &*pixmap*, const QRectF &*source*);
void **drawPixmap**(const QRect &*target*, const QPixmap &*pixmap*, const QRect &*source*);
void **drawPixmap**(int *x*, int *y*, int *w*, int *h*, const QPixmap &*pixmap*, int *sx*, int *sy*, int *sw*, int *sh*);
void **drawPixmap**(const QPointF &*point*, const QPixmap &*pixmap*, const QRectF &*source*);
void **drawPixmap**(const QPoint &*point*, const QPixmap &*pixmap*, const QRect &*source*);
void **drawPixmap**(int *x*, int *y*, const QPixmap &*pixmap*, int *sx*, int *sy*, int *sw*, int *sh*);
以上函数表示把像素图 *pixmap* 的矩形部分(由 *source* 指定)绘制到位置 *point* 或目标矩形 *target* 内。若图像和矩形大小不一致，图像将缩放以适应其大小。
- 6)、void **drawTiledPixmap**(const QRectF &*rectangle*, const QPixmap &*pixmap*,
const QPointF &*position* = QPointF());
void **drawTiledPixmap**(const QRect &*rectangle*, const QPixmap &*pixmap*,
const QPoint &*position* = QPoint());
void **drawTiledPixmap**(int *x*, int *y*, int *width*, int *height*, const QPixmap &*pixmap*, int *sx*=0, int *sy*=0);

在矩形 `rectangle` 内绘制一个平铺像素图，其原点位于 `position`，原点是指 `pixmap` 左上角的位置，具体原理见后文示例。调用以上函数类似于多次调用 `drawPixmap()` 以使用像素图平铺一个区域。

- 7)、void `drawPixmapFragments`(const QPixmapFragment **fragments*, int *fragmentCount*, const QPixmap &*pixmap*, QPixmapFragmentHints *hints* = QPixmapFragmentHints());
- 把像素图 `pixmap` 拆分为 `fragmentCount` 个像素图片段 `fragments`，并以不同比例、旋转和透明度等在多个位置绘制这些像素图片段。
 - 该函数比多次调用 `drawPixmap()` 更快。
 - 枚举 `QPainter::PixmapFragmentHint` 就只有一个值，即 `QPainter::OpaqueHint`，表示需要绘制的像素图片段是不透明的，不透明的片段会更快绘制。
 - `PixmapFragment` 是 `QPainter` 类中的嵌套类，该类拥有如下成员函数，其余说明及变量见下表

QPainter::PixmapFragment 嵌套类中的成员变量	
1、该嵌套类需配合 <code>QPainter::drawPixmapFragments()</code> 函数使用	
2、 <code>sourceLeft</code> 、 <code>sourceTop</code> 、 <code>width</code> 、 <code>height</code> 四个变量被传入 <code>QPainter::drawPixmapFragments()</code> 函数的像素图中的源矩形。	
3、 <code>x</code> 、 <code>y</code> 、 <code>width</code> 、 <code>height</code> 四个变量被用于计算绘制的目标矩形。 <code>x</code> 和 <code>y</code> 表示目标矩形的中心。	
4、目标矩形的 <code>width</code> 和 <code>height</code> 由变量 <code>scaleX</code> 和 <code>scaleY</code> 进行缩放，然后把得到的目标矩形以 <code>x</code> 、 <code>y</code> 为中心点旋转 <code>rotation</code> 度。	
static QPixmapFragment <code>create</code> (const QPointF & <i>pos</i> , const QRectF & <i>sourceRect</i> , qreal <i>scaleX</i> = 1, qreal <i>scaleY</i> = 1, qreal <i>rotation</i> = 0, qreal <i>opacity</i> = 1);	这是静态函数，用于创建一个 <code>QPainter::PixmapFragment</code> ，注意 <code>pos</code> 指的是目标矩形的中心点而不是左上角的坐标。
qreal <code>opacity</code>	保存目标矩形的不透明度(0 透明，1 不透明)
qreal <code>rotation</code>	保存目标矩形的旋转角度(以度为单位)。目标矩形在缩放后被旋转。
qreal <code>scaleX</code> qreal <code>scaleY</code>	保存目标矩形的水平/垂直缩放比。
qreal <code>sourceLeft</code> qreal <code>sourceTop</code>	保存源矩形左侧或顶部的坐标
qreal <code>height</code> qreal <code>width</code>	保存源矩形的宽/高度，并用于计算目标矩形的宽/高度。
qreal <code>x</code> qreal <code>y</code>	保存目标矩形中心点的 <code>x</code> 和 <code>y</code> 坐标。

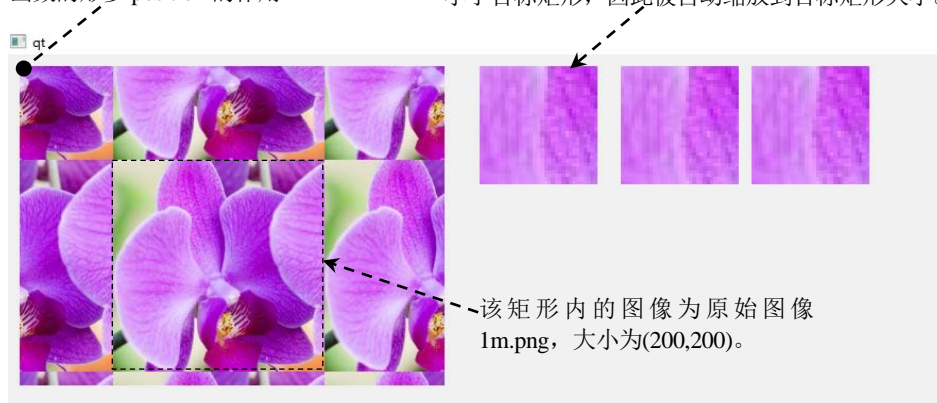
示例：读取文件的一部分到目标矩形与平铺图像

```
void paintEvent(QPaintEvent *e) {
    QPainter pr(this);    QPixmap pml("F:/1m.png");
    //把原图像 1m.png 的 (55, 55, 33, 33) 部分图形绘制到目标矩形。
    pr.drawPixmap(QRect(444, 11, 111, 111), pml, QRect(55, 55, 33, 33));
    pr.drawPixmap(QRect(577, 11, 111, 111), pml, QRect(55, 55, 33, 33));
    pr.drawPixmap(QRect(700, 11, 111, 111), pml, QRect(55, 55, 33, 33));
    pr.drawTiledPixmap(QRect(11, 11, 400, 300), pml, QPointF(111, 111)); } //图像平铺到矩形
```

运行结果及说明

原图像以(111,111)的位置开始平铺于目标矩形，这就是 `drawTiledPixmap()` 函数的形参 `position` 的作用。

原始图像 `1m.png` 的(55,55,33,33)部分被裁剪下来绘制到该目标矩形(444,11,111,111)之内，因裁剪后的图像小于目标矩形，因此被自动缩放到目标矩形大小。



示例：使用 `drawPixmapFragments()` 函数以不同形式显示原始图像的不同部分

```
void paintEvent(QPaintEvent *e) {
    QPainter pr(this);
    QPixmap pml("F:/1m.png");
    QRectF r(11, 11, 55, 55);

    QPainter::PixmapFragment fg1=QPainter::PixmapFragment::create(QPointF(111, 111), r);
    fg1.rotation=60;    //使 fg1 旋转 60 度
    QPainter::PixmapFragment fg2=QPainter::PixmapFragment::create(QPointF(222, 111), r, 2);
    QPainter::PixmapFragment fg3=QPainter::PixmapFragment::create(QPointF(333, 111), r, 1, 2, 60);
    QPainter::PixmapFragment fg4=QPainter::PixmapFragment::create(QPointF(411, 111), r, 1, 2);
    QPainter::PixmapFragment pf[4]={fg1, fg2, fg3, fg4};
    //以不同形式显示原始图像 1m.png 的各个部分(本例只裁剪了范围(11, 11, 55, 55)的部分)
    pr.drawPixmapFragments(pf, 4, pml, QPainter::OpaqueHint);
    //绘制的以下直线用于验证目标矩形的中心位置。
    pr.drawLine(0, 111, 444, 111);    pr.drawLine(111, 0, 111, 444);    pr.drawLine(222, 0, 222, 444);}
```

运行结果及说明



原始图像 `1m.png` 的各个部分以不同的形式(缩放、旋转)显示

三、QPixmap 类中的成员函数

QPixmap 默认支持的文件格式如下表

文件格式	说明	文件格式	说明	文件格式	说明
BMP	读/写	PBM	读	XBM	读/写
JPG	读/写	PGM	读	XPM	读/写
JPEG	读/写	PPM	读/写	GIF	读
PNG	读/写				

1、构造函数

- 1), **QPixmap**(); **QPixmap**(int *width*, int *height*); **QPixmap**(const QSize &*size*);
 QPixmap(const QPixmap &*pixmap*);
- 2), **QPixmap**(const QString &*fileName*, const char **format* = Q_NULLPTR,
 Qt::ImageConversionFlags *flags* = Qt::AutoColor);

使用格式 `format` 加载文件 `fileName`，若文件不存在或格式未知，则创建一个空像素图，若加载图像时未指定格式，则加载程序会尝试猜测文件格式。参数 `flags` 用于指示图像怎样从高分辨率转换为低分辨率。枚举 `Qt::ImageConversionFlag` 见 `QPainter::drawImage()`。

- 3)、**QPixmap**(const char **const*[] xpm);
使用 xpm 数据构造一个像素图，注：XPM 是一种 X11 上使用的图像格式，是一种基于 ASCII 编码的图像格式。

2、像素图的加载和存储

- 4)、bool **load**(const QString &*fileName*, const char **format* = Q_NULLPTR,
- Qt::ImageConversionFlags *flags* = Qt::AutoColor);
- 使用格式 *format* 加载文件 *fileName*，若加载成功则返回 **true**，否则返回 **false**。
 - 若加载图像时未指定格式，则加载程序会尝试猜测文件格式。
 - 参数 *flags* 用于指示图像怎样从高分辨率转换为低分辨率。枚举 Qt::ImageConversionFlag 见 QPainter::drawImage()。
 - 注意: 当从文件加载时, QPixmap 会被自动添加到 QPixmapCache 中, 使用的键(key) 是内部的, 不能获取。
 - 加载文件后对像素图的修改并不会改变原始文件 *fileName* 的内容。
- 5)、bool **loadFromData**(const uchar **data*, uint *len*, const char **format* = Q_NULLPTR,
- Qt::ImageConversionFlags *flags* = Qt::AutoColor);
- bool **loadFromData**(const QByteArray &*data*, const char **format* = Q_NULLPTR,
- Qt::ImageConversionFlags *flags* = Qt::AutoColor);
- 使用格式 *format* 从二进制数据 *data* 加载一个像素图，若加载成功则返回 **true**。
- 6)、static QPixmap **fromImageReader**(QImageReader **imageReader*,
- Qt::ImageConversionFlags *flags* = Qt::AutoColor); //静态的

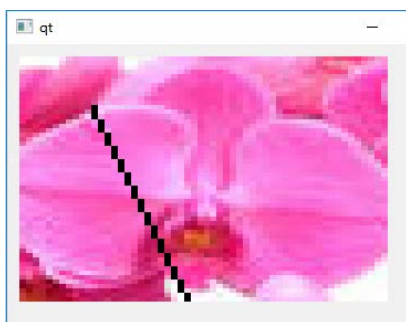
直接从 `imageReader` 读取的图像创建一个像素图。在某此系统上,把图像读入到 `QPixmap` 比读取 `QImage` 再将其转换为 `QPixmap` 使用更少的内存。枚举 `Qt::ImageConversionFlag` 见 `QPainter::drawImage()`。

- 7)、bool `save`(const `QString` &`fileName`, const char *`format` = `Q_NULLPTR`, int `quality` = -1) const;
 - 使用给定的质量 `quality` 和文件格式 `format` 把像素图保存到文件 `fileName` 中。
 - 若成功,则返回 `true`, 否则返回 `false`,
 - 质量因子 `quality` 必须在 0~100 范围内,值 0 表示小型压缩文件,100 表示未压缩文件, -1 表示使用默认值。
 - 若 `format` 为 0, 则从 `fileName` 的后缀中选择图像格式。
- 8)、bool `save`(`QIODevice` *`device`, const char *`format` = `Q_NULLPTR`, int `quality` = -1) const;
此函数表示把文件写入设备 `device` 中,比如可把文件写入 `QByteArray` 中。
- 9)、qint64 `cacheKey`() const;
返回标识此 `QPixmap` 的编号,当改变像素图时 `cacheKey()`将发生改变。
- 10)、void `detach`();
从共享像素图中分离像素图。当像素图的内容即将发生改变时(比如调用 `fill()`、`fromImage()`、`load()`等), `Qt` 就会自动分离它。

示例: 把像素图保存到文件与绘制到 `QWidget` 部件的区别

```
void paintEvent(QPaintEvent *e) {
    QPainter pr;
    QPixmap pm(200, 200);
    pm.load("F:/li.png");
    qDebug() << pm.cacheKey();    //1、验证改变像素图时 cacheKey() 的值会被改变。
    //在 pm 绘制一直线
    pr.begin(&pm);    pr.drawLine(11, 11, 44, 111);    pr.end();
    qDebug() << pm.cacheKey();    //此处输出的值与在 1 处输出的值不相同。
    pr.begin(this);
    pr.drawPixmap(11, 11, 333, 222, pm); //把 pm 绘制到 QWidget 部件的矩形(11, 11, 444, 333)上。
    pm.save("F:/zzz.png");    //保存图像
    pr.end();}
```

运行结果及说明



绘制在 `QWidget` 上的 `pm`



由图可见,绘制在 `QWidget` 上的 `pm`, 其图片的大小为 (333,222), 此时原始图像会被放大到与目标矩形大小 (333,222)一致, 因此 `pm` 中的直线和加载的图片也会被放大, 在图中可明显见到直线和图像被放大后的锯齿。保存在文件 `zzz.png` 中的 `pm`, 其图片的大小为原始图像的大小(50,50), 原始图像未被缩放。原始文件 `li.png` 的内容并未被修改(没有绘制的直线)。

3、像素图的基本信息

- 11)、int **depth**() const; //返回像素图的深度，空像素图深度为 0。
static int **defaultDepth**(); //返回该程序使用的默认像素图深度，通常返回主屏幕的深度。静态的
- 12)、int **height**() const; //返回像素图的高度。
int **width**() const; //返回像素图的宽度
QSize **size**() const; //返回像素图的大小。
QRect **rect**() const; //返回像素图的包围矩形。
- 13)、bool **isNull**() const; //若像素图是空的，则返回 true。空像素具有零宽度、零高度和零内容。
bool **isQBitmap**() const; //若像素图是一个 QPixmap 则返回 true。
- 14)、qreal **devicePixelRatio**() const; //返回设备像素比，默认为 0。
void **setDevicePixelRatio**(qreal *scaleFactor*);
设置设备像素比。默认为 1，若把其设置为其他值，将使像素图被缩放，比如像素图为 200*200，若设置设备像素比为 2，则像素图会被缩小为 100*100，因为这相当于提高了当前绘制设备的分辨率。
- 15)、bool **hasAlpha**() const; //若像素图具有 alpha 通道或蒙版，则返回 true。该函数是遗留函数。
bool **hasAlphaChannel**() const; //若像素图具有 alpha 通道格式，则返回 true。
void **setAlphaChannel**(const QPixmap &p); //不推荐使用。可使用 QPainter::CompositionMode()代替
QPixmap **alphaChannel**() const; //不推荐使用该函数，可使用 QPainter::CompositionMode()代替

4、像素图的复制、填充、替换

- 16)、void **fill**(const QColor &color = Qt::white); //使用颜色 color 填充像素图。
- 17)、QPixmap **copy**(const QRect &rectangle = QRect()) const;
QPixmap **copy**(int x, int y, int width, int height) const;
返回矩形范围内的像素图的深层副本，若 rectangle 为空，则复制整个图像，深层副本详见隐式数据共享。
- 18)、void **swap**(QPixmap &other); //把该像素图与 other 交换。

5、像素图的缩放与滚动

- 19)、QPixmap **scaled**(const QSize &size, Qt::AspectRatioMode *aspectRatioMode* = Qt::IgnoreAspectRatio, Qt::TransformationMode *transformMode* = Qt::FastTransformation) const;
QPixmap **scaled**(int width, int height, Qt::AspectRatioMode *aspectRatioMode* = Qt::IgnoreAspectRatio, Qt::TransformationMode *transformMode* = Qt::FastTransformation) const;
把像素图缩放到给定的大小，并返回缩放后的副本(注意：不会修改原始像素图)。若 size 为空或 width、height 为 0 或负，则函数返回空像素图。其中参数 aspectRatioMode 用于指定缩放时是否保持高宽比，transformMode 用于指定缩放时是否平滑图像。
Qt::AspectRatioMode 和 Qt::TransformationMode 枚举分别见下表、示例和图示。

Qt::AspectRatioMode 枚举(无标志)

作用：描述缩放时是否保持高宽比(纵横比)，原理见下面图示及示例代码

成员	值	说明
----	---	----

Qt::IgnoreAspectRatio	0	不保留纵横比，即，尺寸可自由缩放。
Qt::KeepAspectRatio	1	保持纵横比，在给定矩形内图像尺寸尽可能保持大的矩形
Qt::KeepAspectRatioByExpanding	2	保持纵横比，在给定矩形外，像素图缩放为外部尺寸尽可能小的矩形。

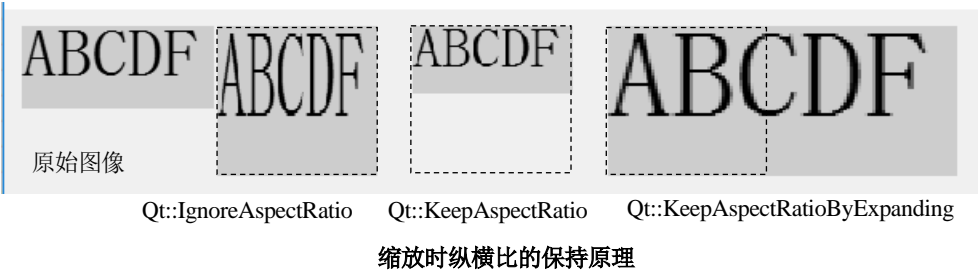
Qt::TransformationMode 枚举(无标志)

作用：描述图像变换时(比如缩放)是否平滑图像

成员	值	说明
Qt::FastTransformation	0	快速转换，没有平滑。
Qt::SmoothTransformation	1	使用双线性滤波变换图像。

示例：像素图的缩放

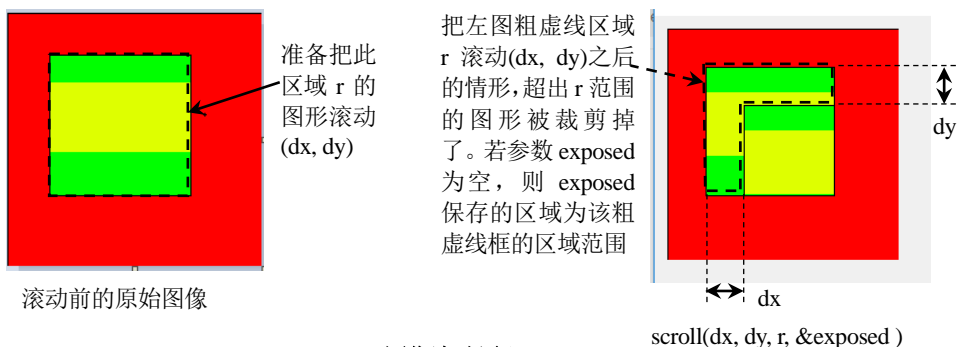
```
void paintEvent(QPaintEvent *e) {
    QPainter pr;    QPixmap pm(120, 55);    QFont f;    f.setPointSize(33);
    //在 pm 上绘制文字
    pr.begin(&pm);    pr.setFont(f);    pr.drawText(0, 33, "ABCDEF");    pr.end();
    pr.begin(this);    pr.drawPixmap(11, 11, pm);    //绘制原始文字
    //不保留纵横比
    QPixmap pml=pm.scaled(100, 100, Qt::IgnoreAspectRatio);    pr.drawPixmap(133, 11, pml);
    //在给定矩形内图像尺寸在保持纵横比的情形下，图像保持最大矩形。
    QPixmap pm2=pm.scaled(100, 100, Qt::KeepAspectRatio);    pr.drawPixmap(255, 11, pm2);
    //在给定矩形外图像尺寸在保持纵横比的情形下，图像外部尺寸保持尽可能小的矩形。
    QPixmap pm3=pm.scaled(100, 100, Qt::KeepAspectRatioByExpanding);pr.drawPixmap(377, 11, pm3);
    pr.end();}
```



- 20)、QPixmap **scaledToHeight**(int *height*, Qt::TransformationMode *mode* = Qt::FastTransformation) const;
 QPixmap **scaledToWidth**(int *width*, Qt::TransformationMode *mode* = Qt::FastTransformation) const;
 返回缩放后的像素图的副本，像素图被保持纵横比缩放到给定的高度 *height* 或宽度 *width*，未指定的宽度或高度由纵横比自动计算。若 *width*、*height* 为 0 或负，则返回空像素图。
- 21)、void **scroll**(int *dx*, int *dy*, int *x*, int *y*, int *width*, int *height*, QRegion **exposed* = Q_NULLPTR);
 void **scroll**(int *dx*, int *dy*, const QRect &*rect*, QRegion **exposed* = Q_NULLPTR);
 把像素图的矩形部分 *rect* 滚动(*dx*, *dy*)距离，区域 *exposed* 可用于保存未被改变的区域部分，当像素图上有一个活动的 QPainter 时，无法进行滚动。原理及示例代码见下图。

```
void paintEvent(QPaintEvent *e) {
    QPainter pr(this);    QPixmap pm("F:/111.png");    QRegion g;    QRect r(33, 33, 111, 111);
    pm.scroll(33, 33, r, &g);    pr.drawPixmap(11, 11, pm);
}
```

```
//输出: QRegion(size=2, bounds=(33, 33 111x111)-[(33, 33 111x33), (33, 66 33x78)])
qDebug()<<g;}
```



图像滚动原理

6、像素图变换

22)、QPixmap **transformed**(const QTransform &*transform*,

Qt::TransformationMode *mode* = Qt::FastTransformation) const;

QPixmap **transformed**(const QMatrix &*matrix*,

Qt::TransformationMode *mode* = Qt::FastTransformation) const;

返回使用 *transform* 变换后的像素图的副本, 原始像素图不会改变。转换时会在内部调整变换矩阵, 以补偿不必要的转换, 使用 *trueMatrix()* 函数可获得实际的变换矩阵。注: QMatrix 类已过时。Qt::TransformationMode 枚举见 *scaled()* 函数。

23)、static QTransform **trueMatrix**(const QTransform &*matrix*, int *width*, int *height*); //静态的

static QMatrix **trueMatrix**(const QMatrix &*m*, int *w*, int *h*); //静态的

返回转换具有宽度 *width*、高度 *height* 和变换矩阵 *matrix* 的像素图的实际变换矩阵。注: QMatrix 类已过时。

7、QPixmap 与 QImage 之间的转换

24)、QImage **toImage**() const;

将该像素图转换为 QImage(图像), 若转换失败, 则返回空图像。若像素图的深度为 1, 则返回的 QImage 也是 1 的深度。注意: 单色图像上的 alpha 蒙版将被忽略。

25)、static QPixmap **fromImage**(const QImage &*image*,

Qt::ImageConversionFlags *flags* = Qt::AutoColor); ///静态的

static QPixmap **fromImage**(QImage &&*image*,

Qt::ImageConversionFlags *flags* = Qt::AutoColor); //qt5.3, 静态的

把图像 *image* 使用标志 *flags* 转换为像素图。枚举 Qt::ImageConversionFlag 见 QPainter::drawImage()。

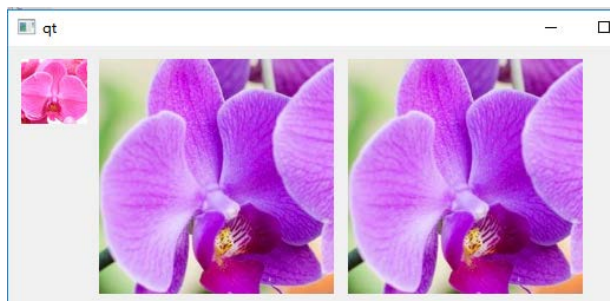
26)、bool **convertFromImage**(const QImage &*image*, Qt::ImageConversionFlags *flags* = Qt::AutoColor);

使用图像 *image* 替换此像素图的数据。枚举 Qt::ImageConversionFlag 见 QPainter::drawImage()。该函数的原理见以下示例及图示。

示例: convertFromImage() 函数的使用

```
void paintEvent(QPaintEvent *e) {
    QPainter pr(this);    QPixmap pm("F:/li.png");    QImage pi("F:/lm.png");
    pr.drawPixmap(11, 11, pm);    pr.drawImage(77, 11, pi);
    pm.convertFromImage(pi);    pr.drawPixmap(288, 11, pm);}

```



pm 原始图

pi 原始图

使用 pi 替换 pm 后的图形

converFromImage()函数原理

8、蒙版(mask)

蒙版的概念:

- ①、在绘图软件中，蒙版类似于蒙在原始图片上的一块玻璃(即蒙版)，然后我们再在这块玻璃上使用其他工具对图片进行修改，这样就不会破坏原始图片，而又能看到图片修改后的效果。
- ②、蒙版通常是黑白色的，通常通过改变蒙版(玻璃)的灰度可改变蒙版的透明度，从而可使原始图片变为可见、不可见、透明三种效果，也就是说蒙版本身只能改变透明度，并不能改变原始图像的色彩或其他性质(这些性质的改变需借助其他工具)。
- ③、蒙版(即玻璃)很少有彩色蒙版，几乎都是黑白色的(即只能通过灰度改变透明度)，虽然可使用蒙版蒙住原始图的某个通道，但蒙版本身仍是黑白色的。比如使用蒙版蒙住红色通道，则通过改变蒙版本身的透明度，可使原始图的红色变为可见、不可见或透明，但蒙版本身并不是彩色的。
- ④、Qt 通常使用 `QBitmap` 类来保存蒙版，`QBitmap` 类是一个只有 1 位深度的图像，因此 Qt 的蒙版只能在可见和不可见之间变换，不能实现透明度的效果。

27)、void `setMask`(const `QBitmap` &`mask`);

设置蒙版位图。此函数把蒙版与像素图的 `alpha` 通道合并，蒙版(即一个 `QBitmap` 图像)上的像素值为 1(黑色)，则像素图的像素不变(即仍被显示)，若蒙版的值为 0，则表示像素图的像素是透明的(即隐藏而不被显示)。蒙版必须与此像素图具有相同的大小。在绘制像素图时，此函数的效果是未定义的。该函数可能是一项昂贵的操作。

28)、`QBitmap mask`() const;

从像素图的 `alpha` 通道创建一个蒙版，该函数是遗留函数(因为操作可能会很昂贵)，不应被使用。

29)、`QBitmap createHeuristicMask`(bool `clipTight` = true) const;

为此像素图创建并返回一个启发式蒙版，该函数的工作原理是从一个角中选择一种颜色，然后从所有边缘开始切掉该颜色的像素，若 `clipTight` 为 true，则蒙版的大小足以覆盖像素，否则蒙版大于数据像素。该函数可能会很慢。

30)、QBitmap `createMaskFromColor(const QColor &maskColor,`

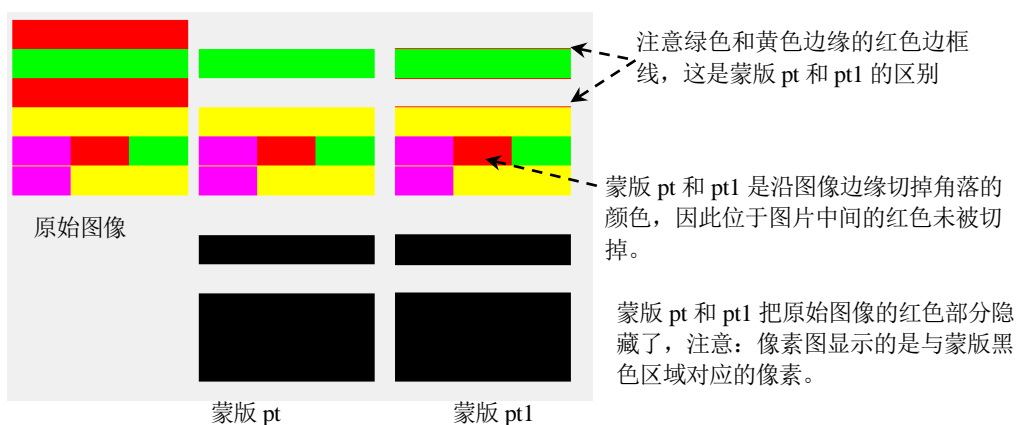
`Qt::MaskMode mode = Qt::MaskInColor)` const;

使用颜色 `maskColor` 为此像素图创建一个蒙版，若 `mode` 是 `Qt::MaskInColor`，则与 `maskColor` 匹配的所有像素都是透明的，即隐藏像素，不显示该像素，若 `mode` 是 `Qt::MaskOutColor`，则与 `maskColor` 匹配的所有像素都是不透明的，即显示该像素。注意：被蒙板黑色部分遮挡的像素会被显示，被白色部分遮挡的像素不会被显示。白色：显示、不透明。黑色：隐藏、透明。

示例：createHeuristicMask() 函数创建的蒙版

```
void paintEvent(QPaintEvent *e){
    QPainter pr(this);    QPixmap pm("F:/1z.png");    QPixmap pml("F:/1z.png");
    //创建两个蒙版
    QBitmap pt=pm.createHeuristicMask(true);    QBitmap ptl=pml.createHeuristicMask(false);
    pr.drawPixmap(11,11,pm);    //绘制原始图片
    //蒙版 pt 的效果
    pm.setMask(pt);    pr.drawPixmap(222,11,pm);    pr.drawPixmap(222,222,pt);
    //蒙版 ptl 的效果
    pml.setMask(ptl);    pr.drawPixmap(444,11,pml);    pr.drawPixmap(444,222,ptl);}
```

运行结果及说明

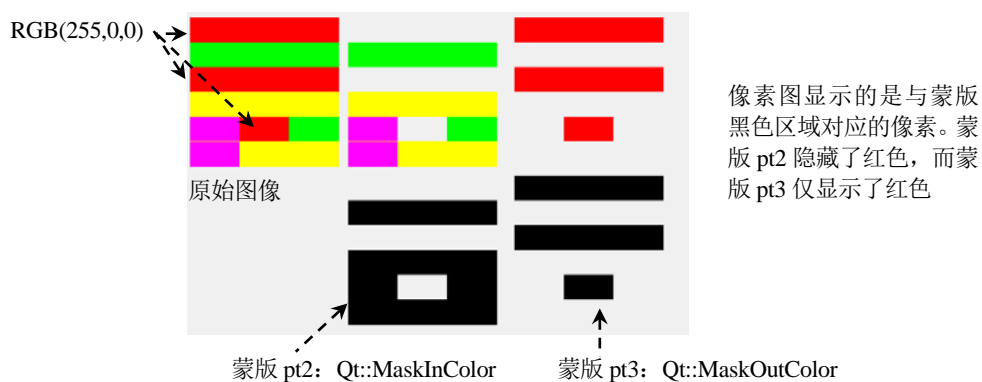


示例：createMaskFromColor() 函数创建的蒙版

```
void paintEvent(QPaintEvent *e){
    QPainter pr(this);    QPixmap pm("F:/1z.png");    QPixmap pml("F:/1z.png");
    //创建两个蒙版
    QBitmap pt2=pm.createMaskFromColor(QColor(255,0,0),Qt::MaskInColor);
    QBitmap pt3=pm.createMaskFromColor(QColor(255,0,0),Qt::MaskOutColor);
    pr.drawPixmap(11,11,pm);    //绘制原始图片
    //蒙版 pt2 的效果
    pm.setMask(pt2);    pr.drawPixmap(222,11,pm);    pr.drawPixmap(222,222,pt2);
    //蒙版 pt3 的效果
```

```
pm1.setMask(pt3);    pr.drawPixmap(444, 11, pm1);    pr.drawPixmap(444, 222, pt3);}
```

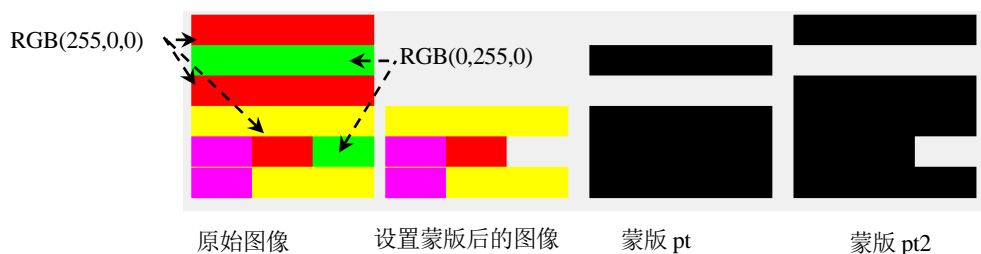
运行结果及说明



示例：setMask() 函数会合并蒙版

```
void paintEvent(QPaintEvent *e) {
    QPainter pr(this);    QPixmap pm("F:/1z.png");
    QBitmap pt=pm.createHeuristicMask(true);
    QBitmap pt2=pm.createMaskFromColor(QColor(0, 255, 0), Qt::MaskInColor);
    pm.setMask(pt);    pm.setMask(pt2);    //同一像素图上设置两个蒙板，这两个蒙板会合并。
    pr.drawPixmap(11, 11, pm);    //绘制原始图像。
    pr.drawPixmap(222, 11, pm);    pr.drawPixmap(444, 11, pt);    pr.drawPixmap(666, 11, pt2);}
```

运行结果及说明



由图可见，使用 setMask()函数把蒙版 pt 和 pt2 合并在一起了。

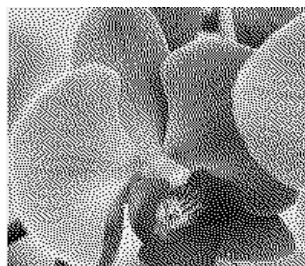
四、QBitmap 类(位图)中的成员函数

- 1、QBitmap 类继承自 QPixmap，该类描述的是 1 位深度(单色)的像素图，即只有两种色的像素图。该类主要用于创建自定义的 QCursor(光标)对象、QRegion 对象及设置图像的蒙版等。

- 2、若把深度大于 1 的像素图分配给 QBitmap，则 QBitmap 将自动抖动。
- 3、使用 Qt::color0 将位图的位设置为 0，使用 Qt::color1 把位图的位设置为 1。其中 0 表示背景(透明像素)，1 表示前景(不透明像素)。注意：使用 Qt::black(黑色)和 Qt::white(白色)没有意义。位图的效果见下图。



原始图像



位图的形式

4、QBitmap 类中的函数

- 1)、**QBitmap**(); //构造函数
 - QBitmap**(const QPixmap &*pixmap*); //使用 pixmap 构造一个位图
 - QBitmap**(int *width*, int *height*);
 - QBitmap**(const QSize &*size*);
 - QBitmap**(const QString &*fileName*, const char **format* = Q_NULLPTR);
 - QBitmap**(const QBitmap &*other*);
- 2)、void **clear**(); //清除位图，即把所有位设置为 Qt::color0。
- 3)、void **swap**(QBitmap &*other*); //把该位图与 other 交换。
- 4)、QBitmap **transformed**(const QTransform &*matrix*) const;

使用变换矩阵 matrix 变换位图，并返回变换后的副本。
- 5)、static QBitmap **fromImage**(const QImage &*image*,

Qt::ImageConversionFlags *flags* = Qt::AutoColor); //静态的

把图像 image 转换为位图，并返回其副本。枚举 Qt::ImageConversionFlag 见 QPainter::drawImage()。
- 6)、static QBitmap **fromData**(const QSize &*size*, const uchar **bits*,

QImage::Format *monoFormat* = QImage::Format_MonoLSB); //静态的

构造一个大小为 size 的位置，并把内容设置为 bits，位图数据必须是字节对齐的，并需按 monoFormat 指定的位顺序提供，单色图格式必须是 QImage::Format_Mono 或 QImage::Format_MonoLSB。

五、QImage 类(位图)中的成员函数

1、QImage 图像的存储

- ①、存储为 QImage 的图像，每个像素使用一个整数表示，
- ②、单色图像的存储：具有 1 位深度的图像(即单色图像)被存储到最多拥有两种颜色的颜色表中。单色图像有两种类型：大端(MSB)或小端(LSB)位顺序。

- ③、8 位深度图像的存储：使用一个 8 位的索引把 8 位图像存储到颜色表中，即 8 位图像的每个像素占据一字节(8 位)的存储空间，即每个像素的颜色与颜色表中某个索引号的颜色相对应。颜色表使用 QVector<QRgb> 存储，QRgb 类型是使用 typedef 定义的一个 unsigned int 类型，该类型包含一个 0xAARRGGBB 格式的四元组 ARGB 格式。
- ④、32 位图像没有颜色表，每个像素直接包含一个 QRgb 类型的值，共有 3 种类型的 32 位图像，分别是 RGB(即 0xFFRRGGBB)，ARGB 和预乘的 ARGB，在预乘格式中，红、绿、蓝通道需乘以 alpha 分量除以 255(详见下一点对 alpha 的讲解)。

2、对 alpha 通道的处理(预乘 alpha)

- ①、带 alpha 通道的图像有两种处理 alpha 通道的方法，一种是直接 alpha，别一种是预乘 alpha，使用预乘 alpha 通道的图像通常会更快。
- ②、直接 alpha 图像的 RGB 数值是原始的数值，而预乘 alpha 图像的 RGB 数值是乘以 alpha 通道后得到的数值，比如 ARGB = (a , r , g, b);则预乘 alpha 后的值为(a, a*r, a*g, a*b);
- ③、Qt 预乘 alpha 通道图像的算法是把红、绿、蓝通道的数值乘以 alpha 通道的数值再除以 255，比如使用 ARGB32 格式表示的 0x7F00004E，使用预乘 ARGB32 格式应表示为 0x7F000027，算法为(0x4E*0x7F) / 0xFF ≈ 0x27，把(0x4E*0x7F) / 0xFF 变换一下为 0x4E*(0x7F / 0xFF) 其中 0x7F/0xFF 是以小数表示的 alpha 值。再如对于透明度为 80% (1-33/FF)的 ARGB32 格式表示的 0x3337304B，使用预乘 ARGB32 格式应为 0x330B170F，以 0x37 为例计算 0x37*(0x33/0xFF)，转换为 10 进制后为 55*0.2 = 11=0xB。

3、QImage 默认支持的文件格式与 QPixmap 相同。

4、公共枚举

QImage::Format 枚举(无标志)		
说明：该枚举描述 QImage 的图像格式		
使用 Format_ARGB32_Premultiplied 比使用 Format_ARGB32 要快得多。		
成员	值	说明
QImage::Format_Invalid	0	图像无效
QImage::Format_Mono	1	MSB 顺序的单色图像(每像素使用 1 位存储)
QImage::Format_MonoLSB	2	LSB 顺序的单色图像(每像素使用 1 位存储)
QImage::Format_Indexed8	3	每像素使用 8 位索引存储图像
QImage::Format_RGB32	4	使用 32 位 RGB 格式(0xFFRRGGBB)存储图像
QImage::Format_ARGB32	5	使用 32 位 ARGB 格式(0xAARRGGBB)存储图像
QImage::Format_ARGB32_Premultiplied	6	使用预乘的 32 位 ARGB 格式(0xAARRGGBB)存储图像
QImage::Format_RGB16	7	使用 16 位 RGB 格式(5-6-5)存储图像
QImage::Format_ARGB8565_Premultiplied	8	使用预乘的 24 位 RGB 格式(8-5-6-5)存储图像
QImage::Format_RGB666	9	使用 24 位 RGB 格式(6-6-6)存储图像，未使用的最高位始终为 0。
QImage::Format_ARGB6666_Premultiplied	10	使用预乘的 24 位 ARGB 格式(6-6-6-6)存储图像
QImage::Format_RGB555	11	使用 16 位 RGB 格式(5-5-5)存储图像，未使用的最高位始终为 0。
QImage::Format_ARGB8555_Premultiplied	12	使用预乘的 24 位 ARGB 格式(8-5-5-5)存储图像
QImage::Format_RGB888	13	使用 24 位 RGB 格式(8-8-8)存储图像
QImage::Format_RGB444	14	使用 16 位 RGB 格式(4-4-4)存储图像，未使用的最高位始终为 0。

QImage::Format_ARGB4444_Premultiplied	15	使用预乘的 16 位 ARGB 格式(4-4-4-4)存储图像
QImage::Format_RGBX8888	16	使用 32 位字节排序的 RGB(x)格式(8-8-8-8)存储图像，这与 Format_RGBA8888 相同，但 alpha 必须始终为 255。
QImage::Format_RGBA8888	17	使用 32 位字节排序的 RGBA 格式(8-8-8-8)存储图像，与 ARGB32 不同的是，这是一种字节排序格式，这意味着 32 位编码在大端和小端之间会不相同。
QImage::Format_RGBA8888_Premultiplied	18	使用预乘的 32 位字节排序 RGBA 格式(8-8-8-8)存储图像
QImage::Format_BGR30	19	使用 32 位 BGR 格式(x-10-10-10)存储图像
QImage::Format_A2BGR30_Premultiplied	20	使用 32 位预乘的 ABRG 格式(2-10-10-10)存储图像
QImage::Format_RGB30	21	使用 32 位 RGB 格式(x-10-10-10)存储图像
QImage::Format_A2Rgb30_Premultiplied	22	使用 32 位预乘的 ARGB 格式(2-10-10-10)存储图像
QImage::Format_Alpha8	23	图像仅使用 8 位 alpha 格式存储
QImage::Format_Grayscale8	24	使用 8 位灰度格式存储图像。

5、构造函数

1)、QImage()

QImage(const QSize &*size*, Format *format*)

QImage(int *width*, int *height*, Format *format*)

QImage(const QImage &*image*)

QImage(QImage &&*other*)

2)、QImage(const QString &*fileName*, const char **format* = Q_NULLPTR)

使用格式 *format* 加载文件 *fileName*，若文件不存在或格式未知，则创建一个空图像，若加载图像时未指定格式，则会根据文件后缀和标题自动检测格式。

3)、QImage(const char * const[] *xpm*)

使用 *xpm* 数据构造一个像素图，注：XPM 是一种 X11 上使用的图像格式，是一种基于 ASCII 编码的图像格式。

4)、QImage(uchar **data*, int *width*, int *height*, Format *format*,

QImageCleanupFunction *cleanupFunction* = Q_NULLPTR,

void **cleanupInfo* = Q_NULLPTR)

QImage(const uchar **data*, int *width*, int *height*, Format *format*,

QImageCleanupFunction *cleanupFunction* = Q_NULLPTR,

void **cleanupInfo* = Q_NULLPTR)

QImage(uchar **data*, int *width*, int *height*, int *bytesPerLine*, Format *format*,

QImageCleanupFunction *cleanupFunction* = Q_NULLPTR,

void **cleanupInfo* = Q_NULLPTR)

QImage(const uchar **data*, int *width*, int *height*, int *bytesPerLine*, Format *format*,

QImageCleanupFunction *cleanupFunction* = Q_NULLPTR,

void **cleanupInfo* = Q_NULLPTR)

使用格式 *format* 从数据 *data* 加载一个图像。*data* 必须是 32 位对齐的，

6、加载/存储图像

- 5)、bool **load**(const QString &fileName, const char *format = Q_NULLPTR);
使用格式 format 加载文件 fileName,
- 6)、bool **load**(QIODevice *device, const char *format);
从设备 device 读取一个 QImage。
- 7)、bool **loadFromData**(const uchar *data, int len, const char *format = Q_NULLPTR);
bool **loadFromData**(const QByteArray &data, const char *format = Q_NULLPTR);
使用格式 format 从二进制数据 data 加载一个图像, 若加载成功则返回 true。
- 8)、static QImage **fromData**(const uchar *data, int size, const char *format = Q_NULLPTR); //静态的
static QImage **fromData**(const QByteArray &data, const char *format = Q_NULLPTR); //静态的
根据二进制数据 data 构造一个 QImage 图像。
- 9)、bool **save**(const QString &fileName, const char *format = Q_NULLPTR, int quality = -1) const;
 - 使用给定的质量 quality 和文件格式 format 把像素图保存到文件 fileName 中。
 - 若成功, 则返回 true, 否则返回 false,
 - 质量因子 quality 必须在 0~100 范围内, 值 0 表示小型压缩文件, 100 表示未压缩文件, -1 表示使用默认值。
 - 若 format 为 0, 则从 fileName 的后缀猜测图像格式。
- 10)、bool **save**(QIODevice *device, const char *format = Q_NULLPTR, int quality = -1) const;
此函数表示把文件写入设备 device 中, 比如可把文件写入 QByteArray 中。
- 11)、qint64 **cacheKey**() const;
返回标识此 QImage 的编号, 当改变图像时 cacheKey()将发生改变。

7、图像基本信息

- 12)、int **depth**() const; //返回图像的深度, 支持的深度有 1, 8, 16, 24, 32。
- 13)、Format **format**() const; //返回图像的格式。
QPixelFormat **pixelFormat**() const; //以 QPixelFormat 的形式返回 QImage::Format。
- 14)、int **height**() const; //返回图像的高度
int **width**() const; //返回图像的宽度。
QSize **size**() const; //返回图像的大小(宽度和高度)。
QRect **rect**() const; //返回像素图的包围矩形。
- 15)、qsizetype **sizeInBytes**() const; //以字节为单位返回图像数据的大小, qt5.10
- 16)、int **bitPlaneCount**() const;
返回图像的位平面数, 当图像格式包含未使用的位时, 位平面数与深度会不相同, 此时位平面数小于深度。
- 17)、bool **hasAlphaChannel**() const; //若像素图具有 alpha 通道格式, 则返回 true。
- 18)、bool **allGray**() const;
若图像是灰色的(即红、蓝、绿通道的值相同), 则返回 true, 否则返回 false, 注意:
对于没有颜色表的图像, 此函数会比较慢。
- 19)、bool **isGrayscale**() const;

对于 32 位图像，此函数相当于 `allGray()`，对于颜色索引图像，若 `color(i)` 是颜色表的所有索引 `QRgb(i, i, i)`，则返回 `true`，否则返回 `false`。

20)、`qreal devicePixelRatio() const;` //返回设备像素比，默认为 1.0。

`void setDevicePixelRatio(qreal scaleFactor);` //设置设备像素比，

21)、`bool isNull() const;`

若是一个空的图像，则返回 `true`，空图像的所有参数都为 0，且没有分配的数据。

22)、`bool valid(const QPoint &pos) const;`

`bool valid(int x, int y) const;`

若 `pos` 或 `(x,y)` 是图像内的有效坐标，则返回 `true`，否则返回 `false`。

8、填充和复制

23)、`void fill(uint pixelValue);`

使用值 `pixelValue` 填充整个图像。填充时使用数值的低位，比如对于 1 位深度的图像，若使用值 0 或 2，则填充 0，若使用 1 或 3 则填充 1，若深度为 8，16，则使用低 8 位、16 位。

24)、`void fill(const QColor &color);`

使用颜色 `color` 填充整个图像，若图像深度为 1，若颜色为 `Qt::color1`，则填充 1，否则填充 0。若图像深度为 8，若存在颜色表，则图像将被填充为对应的索引处的颜色，否则将填充 0。

25)、`void fill(Qt::GlobalColor color);` //使用标准的全局颜色 `color` 填充图像

26)、`QImage copy(const QRect &rectangle = QRect()) const;`

`QImage copy(int x, int y, int width, int height) const;`

返回矩形范围内的像素图的深层副本，若 `rectangle` 为空，则复制整个图像，深层副本详见隐式数据共享。

9、颜色表和操控像素

23)、`void setColorCount(int colorCount);` //设置颜色表的大小。

`int colorCount() const;`

返回颜色表的大小，注意：对于 32 位深度的图像该函数返回 0。

24)、`void setColorTable(const QVector<QRgb> colors);` //设置颜色表。

`QVector<QRgb> colorTable() const;`

返回颜色表中包含的颜色的列表，若没有颜色表，则返回空列表。

25)、`QRgb color(int i) const;`

返回位于索引 `i` (从 0 计数) 处的颜色表中的颜色，颜色表中的颜色使用 ARGB 四元组 (`QRgb`) 表示，可使用 `::qAlpha(QRgb)`，`::qRed(QRgb)`，`::qGreen(QRgb)`，`::qBlue(QRgb)` 获取这些元素。

26)、`void setColor(int index, QRgb colorValue);`

把颜色表中索引为 `index` 处的颜色设置为 `colorValue`，颜色具有 ARGB 四元组，若 `index` 超出了颜色表的当前大小，则使用 `setColorCount()` 进行扩展。

27)、`QRgb pixel(int x, int y) const;` //返回坐标 `(x,y)` 处像素的颜色。

QRgb **pixel**(const QPoint &*position*) const;

返回位置 *position* 处像素的颜色，若该位置无效，则结果是未定义的。当用于大规模像素操作时，此函数很昂贵，若需要读取多个像素，则使用 `constBits()` 或 `constScanLine()` 函数。

28)、void **setPixel**(const QPoint &*position*, uint *index_or_rgb*);

void **setPixel**(int x, int y, uint *index_or_rgb*);

把位置 *position* 或坐标(x,y)处像素的索引或颜色设置为 *index_or_rgb*。若图像的格式是单色或调色板的，则 *index_or_rgb* 必须是颜色表的索引，否则必须是 QRgb 值。

该函数很昂贵，若需要考虑性能，建议使用 `scanLine()` 或 `bits()` 直接访问像素数据。

29)、QColor **pixelColor**(const QPoint &*position*) const; //qt5.6

QColor **pixelColor**(int x, int y) const; //qt5.6

以 QColor 的形式返回位置 *position* 或坐标(x,y)处像素的颜色。

30)、void **setPixelColor**(const QPoint &*position*, const QColor &*color*); //qt.56

void **setPixelColor**(int x, int y, const QColor &*color*); //qt.56

把位置 *position* 或坐标(x,y)处像素的颜色设置为 *color*

31)、int **pixelIndex**(const QPoint &*position*) const;

int **pixelIndex**(int x, int y) const;

返回位置 *position* 或坐标(x,y)处的像素索引，若 *position* 无效，或图像不是调色板(颜色表)图像(即位深大于 8 的图像)，则结果是未定义的。

示例：颜色表及像素操控

//m.h 文件的内容

```
#ifndef M_H
```

```
#define M_H
```

```
#include<QtWidgets>
```

```
class B:public QWidget{    Q_OBJECT
```

```
public:    B(QWidget *p1=0):QWidget(p1) {}
```

```
void paintEvent(QPaintEvent *e) {
```

```
    QPainter pr;
```

```
    QImage pi(200,200,QImage::Format_Mono);
```

```
    QRgb r1=QRgb(111,1,1); //红色
```

```
    QRgb r2=QRgb(1,111,1); //绿色
```

```
    QRgb r3=QRgb(111,111,1); //黄色
```

```
//绘制单色图未填充时的默认样式
```

```
    pr.begin(this);    pr.drawImage(11,11,pi);    pr.end();
```

```
    qDebug()<<pi.colorTable(); //输出默认颜色表 QVector(4278190080, 4294967295)，其中
```

```
                //4278190080=0xFF00 0000, 4294967295=0xFFFF FFFF
```

```
    pi.fill(Qt::color1); //使用单色图的颜色表中的索引为 1 的颜色填充图像 pi。
```

```
//以下步骤用于设置颜色表
```

```
    QVector<QRgb> v;    v.append(r1);    v.append(r2);
```

```
    pi.setColorCount(2); //设置颜色表中的颜色数量
```

```
    pi.setColorTable(v); //设置一个新颜色表
```

```
//使用单色图的新颜色表逐像素绘制一个矩形(22, 22, 128, 100)
```

```
    pr.begin(&pi);
```

```

for(int j=22;j<122;j++)
    for(int i=22;i<150;i++){
        pi.setPixel(i,j,0); }//使用新颜色表中索引为0的颜色填充位于(i,j)处的像素。
pr.end();

pr.begin(this);
pr.drawImage(222,11,pi);    //在 QWidget 中绘制 pi
QDebug()<<pi.colorTable(); //输出新设置的颜色表 QVector(4285464833, 4278284033),
                        //其中 4285464833=0xFF6F 0101, 4278284033=0xFF01 6F01, 0x6F=111

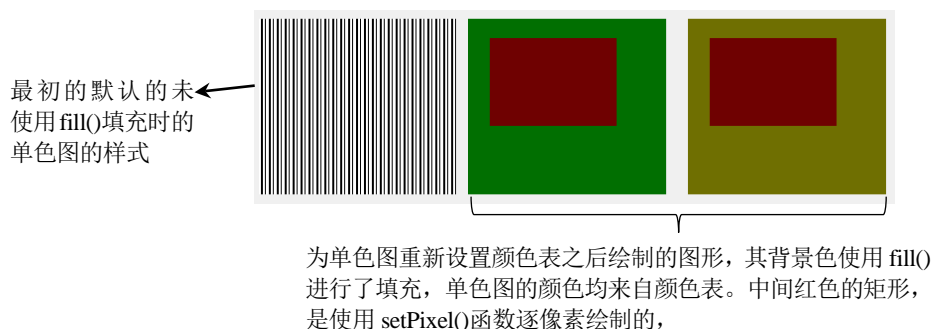
pi.setColor(1,r3); //把新颜色表中索引为1的颜色更改为r3(黄色), 注意: setColor()函数
                //改变的是颜色表中的颜色, setPixel()才能改变图像中某个位置的像素颜色。
pr.drawImage(444,11,pi); //在 QWidget 重新绘制 pi
QDebug()<<pi.colorTable(); //输出更改后的颜色表 QVector(4285464833, 4285492993)
                        //其中 4285492993=0xFF6F 6F01, 0x6F=111

pr.end();
});
#endif // M_H

//m.cpp 文件内容
#include "m.h"
int main(int argc, char *argv[]) {    QApplication app(argc,argv);
    B w;    w.resize(444,333);    w.show();    return app.exec(); }

```

运行结果及说明



10、扫描线

扫描线：是指把图像按水平方向分割成很多条线，每条线就是一条扫描线，说简单点就是一条扫描线就是图像中的一行，比如对于 300*200 的图像，共有 200 条扫描线(即，有 200 行)。

32)、uchar *bits();

const uchar *bits() const;

返回指向第一个像素数据的指针，相当于 scanLine(0)。

33)、const uchar *constBits() const; //返回指向第一个像素数据的指针。

34)、uchar ***scanLine**(int i);

const uchar ***scanLine**(int i) const;

返回指向具有索引为 I(从 0 计数)的扫描线上的像素数据的指针，扫描线数据是在 32 位边界上对齐的。若要访问 32 位深度的图像数据，可把返回的指针强制转换为 QRgb* 格式，然后再进行读/写像素值。

35)、const uchar ***constScanLine**(int i) const;

返回指向扫描线上具有索引为 i(从 0 计数)的像素数据的指针，扫描线数据是在 32 位边界上对齐的。

36)、int **bytesPerLine**() const;

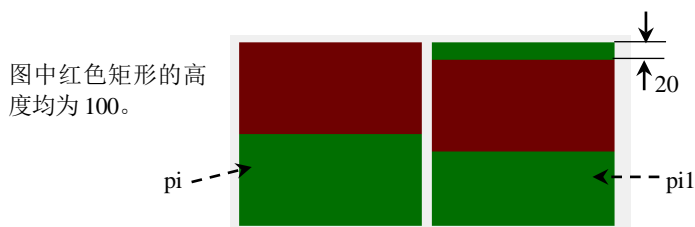
返回每个图像扫描线的字节数，若 height()不为零，则相当于 sizeInBytes()/height();

示例：扫描线的使用

```
void paintEvent(QPaintEvent *e) {
    QPainter pr;
    QImage pi(200,200,QImage::Format_ARGB32);
    QImage pil(200,200,QImage::Format_ARGB32);
    pi.fill(qRgb(1,111,1));    pil.fill(qRgb(1,111,1));    //使用绿色填充背景
    //使用扫描线函数返回的指针逐像素绘制一个矩形
    pr.begin(&pi);
    //获取指向第一行第一个像素的指针，并将其强制转换为 QRgb* 以方便修改。
    QRgb *pu=(QRgb*)pi.bits();
    for(int i=0;i<20000;i++)    //逐像素设置每个像素的颜色，pi 每行有 200 个像素，
                                //循环 2 万次意味着设置 100 行像素的颜色。
    {    *pu=qRgb(111,1,1);pu++; }
    QRgb *pul=(QRgb*)pil.scanLine(19); //获取第 20 行扫描线第一个元素的指针
    for(int i=0;i<20000;i++){    *pul=qRgb(111,1,1);pul++; }
    pr.end();
    pr.begin(this);
    pr.drawImage(11,11,pi);    pr.drawImage(222,11,pil);
    qDebug()<<pil.bytesPerLine(); //输出 800(字节)，因为 pi 是 32 位的 ARGB 图像，
                                //每个像素占据 4 字节，每一行有 200 个像素，
                                //因此每个扫描线占据 800 个字节大小。

    pr.end();}
```

运行结果及说明



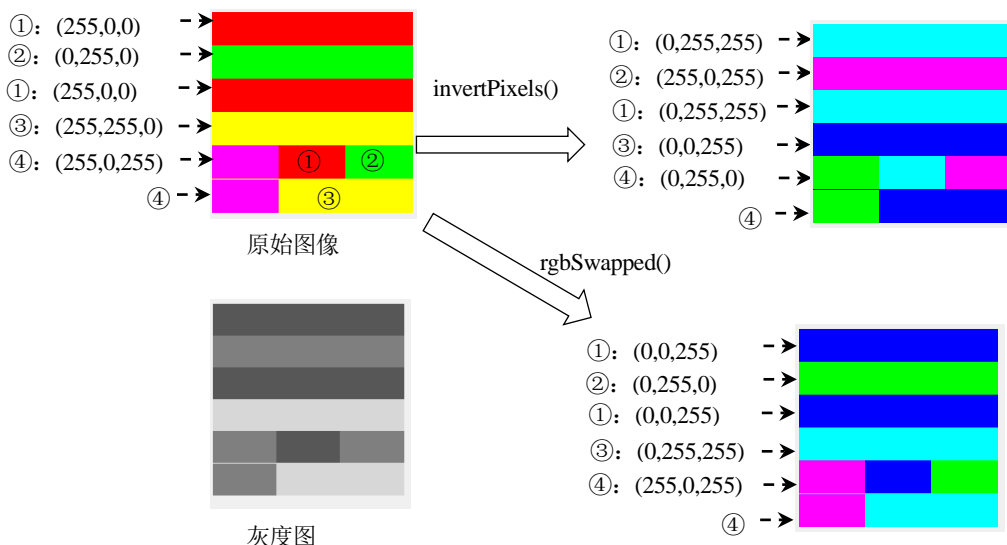
11、图像格式或类型转换

37)、QImage **convertToFormat**(Format *format*, Qt::ImageConversionFlags *flags* = Qt::AutoColor) const;

QImage **convertToFormat**(Format *format*, const QVector<QRgb> &*colorTable*,


```
pr.drawImage(222, 11, pi2);
pr.end(); }
```

运行结果及说明



12、图像变换(以下函数的使用详见 QPixmap 类中相应函数的讲解)

- 45)、QImage **scaled**(const QSize &size, Qt::AspectRatioMode *aspectRatioMode* = Qt::IgnoreAspectRatio, Qt::TransformationMode *transformMode* = Qt::FastTransformation) const;
QImage **scaled**(int *width*, int *height*, Qt::AspectRatioMode *aspectRatioMode* = Qt::IgnoreAspectRatio, Qt::TransformationMode *transformMode* = Qt::FastTransformation) const;
QImage **scaledToHeight**(int *height*, Qt::TransformationMode *mode* = Qt::FastTransformation) const;
QImage **scaledToWidth**(int *width*, Qt::TransformationMode *mode* = Qt::FastTransformation) const;
- 46)、QImage **transformed**(const QMatrix &matrix, Qt::TransformationMode *mode* = Qt::FastTransformation) const;
QImage **transformed**(const QTransform &matrix, Qt::TransformationMode *mode* = Qt::FastTransformation) const;
- 47)、static QMatrix **trueMatrix**(const QMatrix &matrix, int width, int height);静态的
static QTransform **trueMatrix**(const QTransform &matrix, int width, int height);静态的
- 48)、QImage **mirrored**(bool horizontal = false, bool vertical = true) const;
返回图像的镜像，是水平还是垂直镜像取决于形参。注意：不会改变原始图像。

13、蒙版(以下函数的使用详见 QPixmap 类中相应函数的讲解)

- 49)、QImage **createAlphaMask**(Qt::ImageConversionFlags *flags* = Qt::AutoColor) const;
QImage **createHeuristicMask**(bool clipTight = true) const;
QImage **createMaskFromColor**(QRgb color, Qt::MaskMode *mode* = Qt::MaskInColor) const;

以上函数用于创建蒙版，蒙版详见 `QPixmap` 类中的相应函数，`createAlphaMask()` 函数从图像的 alpha 缓冲区创建一个蒙版，以上函数创建的蒙版是 `QImage` 而不是 `QBitmap`，返回的 `QImage` 具有格式 `QImage::Format_MonoLSB`，可使用 `convertToFormat()` 将其转换为 `QImage::Format_Mono`。

14、图像的文本

50)、void `setText`(const `QString` &key, const `QString` &text);

设置图像的文本为 `text`，并使用键 `key` 与之关联，若只想存储单个文本块(比如注释或仅描述)，则可以使用空键。并非所有图像格式都支持嵌入文本，可使用 `QImageWrite::supportsOption()` 函数查看。

51)、`QString text`(const `QString` &key = `QString()`) const;

返回与键 `key` 关联的图像文本，若键是空字符串，则返回整个图像的文本，<键，文本>对由换行符分隔。

52)、`QStringList textKeys`() const; //返回此图像的文本键列表。

其他

53)、int `dotsPerMeterX`() const;

int `dotsPerMeterY`() const;

void `setDotsPerMeterX`(int x);

void `setDotsPerMeterY`(int y);

以上函数用于获取和设置物理水平/垂直像素数，水平/垂直像素数定义了图像的预期比例和纵横比，并确定 `QPainter` 在图像上绘制图形的比例，在其他绘制设备上绘制时，它不会更改图像的比例和纵横比。

54)、`QPoint offset`() const; //返回相对于其他图像定位时图像偏移的像素数。

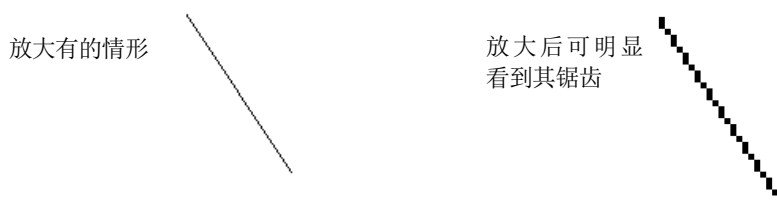
55)、void `setOffset`(const `QPoint` &offset); //设置相对于其他图像定位时图像偏移的像素数。

六、QPicture 类(该类较简单，从略)

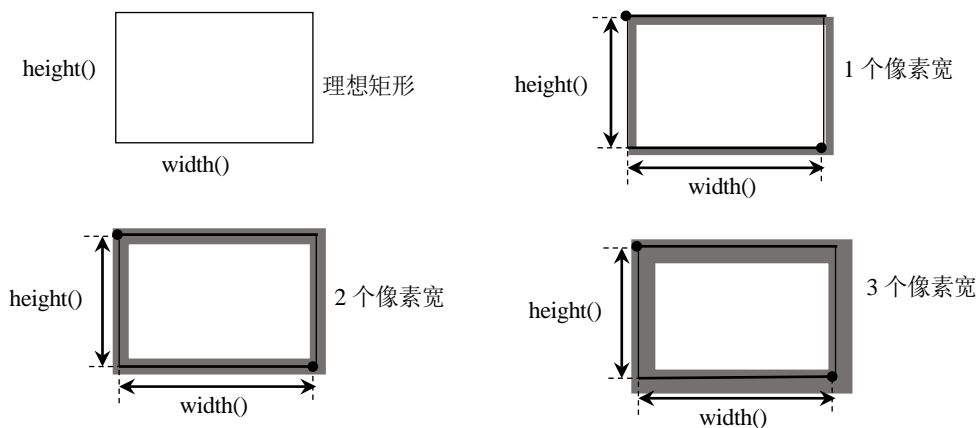
12.14 抗锯齿和图像合成

一、抗锯齿(Anti-aliased)

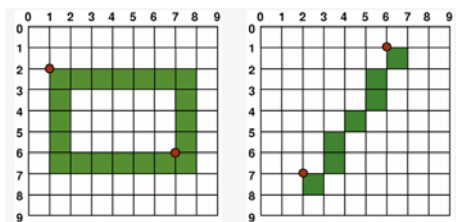
- 1、计算机上的图像都是以像素为单位显示的，像素其实就是一个一个小方块，因此显示出来的图像只有完全水平或垂直的直线才有可能是直的，对于斜线，在未放大的情况下，看起来可能是直线没有锯齿(走样，aliasing)，但一旦放大就必然会带有锯齿现象，如下图



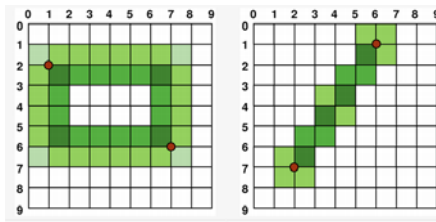
- 2、把用于减少锯齿的技术称为抗锯齿、反走样或反锯齿。抗锯齿的算法有很多种，但基本原理其实就是对轮廓线的像素设置不同的灰度值使其产生模糊的效果，从而减轻锯齿。
- 3、数学直线，是指在数学上的理想的直线，他是没有宽度的，然而实际上的图形，至少都需要占据一个像素的宽度。
- 4、当启用抗锯齿时，像素将在数学定义的点的两侧对称渲染。
- 5、未启用抗锯齿时的图形绘制原则(见下图)
 - ①、当使用一个像素宽的笔绘制图形时，像素将在数学定义的点的右侧和下方。
 - ②、当使用偶数个像素宽的笔绘制图形时，像素将在数学定义的点的周围对称地渲染。
 - ③、当使用奇数个像素宽的笔绘制图形时，多余的像素将在数学定义的点的右侧和下方。这与在一个像素宽时类似。



未启用抗锯齿时绘制的矩形



未使用抗锯齿时的效果



使用抗锯齿时的效果

6、需要用到的 QPainter 类中的函数如下表

QPainter 类与抗锯齿有关的函数	
void setRenderHint (RenderHint <i>hint</i> , bool <i>on</i> = true)	设置渲染提示。RenderHint 枚举见下表
void setRenderHints (RenderHints <i>hints</i> , bool <i>on</i> = true)	
RenderHints renderHints () const	返回渲染提示
bool testRenderHint (RenderHint <i>hint</i>) const	若设置了 hint 则返回 true。

QPainter::RenderHint 枚举

标志：QPainter::RenderHints

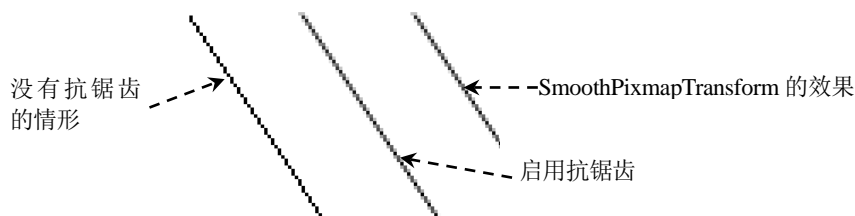
- 1、作用：描述渲染图像时的质量提示，
- 2、QPainter 默认是没有启动抗锯齿的。
- 3、注意，以下枚举只是一种提示，实际上 QPainter 可能会也可能不会执行以下的操作。

成员	值	说明
QPainter::Antialiasing	0x01	若可能，则启用抗锯齿处理
QPainter::TextAntialiasing	0x02	若可能，则文本启用抗锯齿。
QPainter::SmoothPixmapTransform	0x04	表示使用平滑的像素图像变换算法(如双线性)
QPainter::HighQualityAntialiasing	0x08	已过时
QPainter::NonCosmeticDefaultPen	0x10	已过时
QPainter::Qt4CompatiblePainting	0x20	用于与 Qt4 相兼容

示例：抗锯齿处理

```
void paintEvent(QPaintEvent *e) {
    QPainter pr(this);
    pr.drawLine(11, 11, 77, 111);
    pr.setRenderHint(QPainter::Antialiasing);    pr.drawLine(44, 11, 111, 111);
    pr.setRenderHint(QPainter::SmoothPixmapTransform);    pr.drawLine(77, 11, 144, 111);}
```

运行结果及说明



二、图像合成

1、需要用到的 QPainter 类中的函数

```
CompositionMode compositionMode() const;           //返回当前的合成模式
void setCompositionMode(CompositionMode mode);      //设置当前的合成模式
```

说明:

- ①、仅在 QImage 上使用 QPainter 才能完全支持合成模式。
- ②、枚举 CompositionMode 详见后文的讲解。

2、图像合成是基于像素的运算，即在两个图像进行合成时，逐像素进行合成。

3、下面简单介绍一下 Proter-Duff 合成公式，为此先定义以下符号

- Sc: 表示源像素的颜色，即 source color
- Dc: 表示目标像素的颜色，即 destination color
- Rc: 表示结果像素的颜色，即 result color
- Sa: 表示源像素的透明度，即 source alpha
- Da: 表示目标像素的透明度，即 destination alpha
- Ra: 表示结果像素的透明度，即 result alpha
- Fs: 表示源像素的因子，即 factor source
- Fd: 表示目标像素的因子，即 factor destination

● 注意:

- ①、alpha 值使用的是 0~1 之间的小数，也就是说若 alpha 是使用的 0~255 的值表示的，需将该值除以 255，比如对于 111 的 alpha 值，即小数值为 111/255。
- ②、颜色是使用的预乘 alpha 颜色(这样 Proter-Duff 公式会更简单)。

4、Proter-Duff 公式，如下:

```
Rc = Sc*Fs + Dc*Fd;           //结果颜色(即合成后的颜色)
Ra = Sa*Fs + Da*Fd;           //结果 alpha(即合成后的 alpha)
```

也可写为

```
[Sa*Fs + Da*Fd, Sc*Fs + Dc*Fd]; //即第一项为 alpha 值，第 2 项为颜色值
```

根据 Fs 和 Fd 的不同取值，共有 12 种不同的合成公式，比如若 Fs = 1，Fd = (1 - Sa);时，得到如下公式

$$Rc = Sc + Dc*(1 - Sa); \quad Ra = Sa + Da*(1 - Sa);$$

详细的公式详见对 QPainter::CompositionMode 枚举的讲解

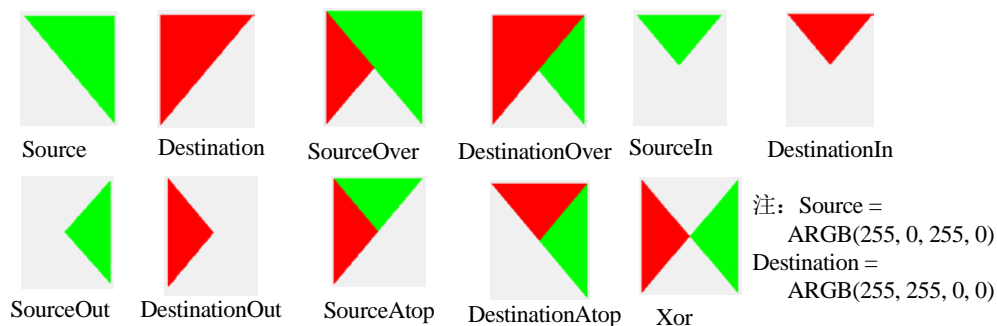
5、QPainter::CompositionMode 枚举

QPainter::CompositionMode 枚举(无标志)		
1、作用：描述图像的合成模式，即用于指定一个图像中的像素如何与另一个图像中的像素合并。		
2、RasterOp 前缀表示的模式仅在 X11 和光栅绘制引擎中起作用。带有 alpha 分量的笔和画刷不支持 RasterOp 表示的模式，打开 QPainter::Antialiasing(抗锯齿)提示，也将禁用 RasterOp 模式。		
3、最常见的模式是 SourceOver(常称为 alpha 混合)。		
4、为获得最佳性能，建议首选图像格式为 Format_ARGB32_Premultiplied		
5、图像合成的效果见表之后的图示。		
成员	值	说明

QPainter::CompositionMode_SourceOver	0	默认模式。源的 alpha 用于合成目标像素之上， $F_s = 1; F_d = (1 - S_a);$ $[S_a + (1 - S_a) * D_a, S_c + (1 - S_a) * D_c]$
QPainter::CompositionMode_DestinationOver	1	目标的 alpha 用于合成在源像素之上 $F_s = (1 - D_a); F_d = 1;$ $[D_a + (1 - D_a) * S_a, D_c + (1 - D_a) * S_c]$
QPainter::CompositionMode_Clear	2	目标的像素被清除(即设置为完全透明)
QPainter::CompositionMode_Source	3	$F_s = 1; F_d = 0$ $[S_a, S_c]$
QPainter::CompositionMode_Destination	4	$F_s = 0; F_d = 1$ $[D_a, D_c]$
QPainter::CompositionMode_SourceIn	5	$F_s = D_a; F_d = 0;$ $[S_a * D_a, S_c * D_a]$
QPainter::CompositionMode_DestinationIn	6	$F_s = 0; F_d = S_a$ $[S_a * D_a, S_a * D_c]$
QPainter::CompositionMode_SourceOut	7	$F_s = (1 - D_a); F_d = 0$ $[S_a * (1 - D_a), S_c * (1 - D_a)]$
QPainter::CompositionMode_DestinationOut	8	$F_s = 0; F_d = (1 - S_a)$ $[D_a * (1 - S_a), D_c * (1 - S_a)]$
QPainter::CompositionMode_SourceAtop	9	$F_s = D_a; F_d = (1 - S_a)$ $[D_a, S_c * D_a + D_c * (1 - S_a)]$
QPainter::CompositionMode_DestinationAtop	10	$F_s = (1 - D_a); F_d = S_a$ $[S_a, D_c * S_a + S_c * (1 - D_a)]$
QPainter::CompositionMode_Xor	11	$F_s = (1 - D_a); F_d = (1 - S_a)$ $[S_a + D_a - 2 * S_a * D_a, S_c * (1 - D_a) + D_c * (1 - S_a)]$
QPainter::CompositionMode_Plus	12	源像素和目标像素的 alpha 和颜色都加在一起。
QPainter::CompositionMode_Multiply	13	输出源颜色乘以目标颜色。 $[S_a * D_a, S_c * D_c]$
QPainter::CompositionMode_Screen	14	$[S_a + D_a - S_a * D_a, S_c + D_c - S_c * D_c]$
QPainter::CompositionMode_Overlay	15	根据目标颜色相乘或筛选颜色。
QPainter::CompositionMode_Darken	16	选择较暗的源和目标颜色 $[S_a + D_a - S_a * D_a, S_c * (1 - D_a) + D_c * (1 - S_a) + \min(S_c, D_c)]$
QPainter::CompositionMode_Lighten	17	选择较浅的源和目标颜色 $[S_a + D_a - S_a * D_a, S_c * (1 - D_a) + D_c * (1 - S_a) + \max(S_c, D_c)]$
QPainter::CompositionMode_ColorDodge	18	目标颜色变亮以反映源颜色。
QPainter::CompositionMode_ColorBurn	19	目标颜色变暗以反映源颜色。
QPainter::CompositionMode_HardLight	20	根据源颜色相乘或筛选颜色。
QPainter::CompositionMode_SoftLight	21	根据源颜色使颜色变暗或变亮。
QPainter::CompositionMode_Difference	22	从较亮的颜色中减去更深的颜色。
QPainter::CompositionMode_Exclusion	23	与 Difference 类似，但对对比度更低。

QPainter::RasterOp_SourceOrDestination	24	对源像素和目标像素执行按位 OR 运算，即 src OR dst
QPainter::RasterOp_SourceAndDestination	25	对源像素和目标像素执行按位 AND 运算， 即 src AND dst
QPainter::RasterOp_SourceXorDestination	26	对源像素和目标像素执行按位 XOR 运算， 即 src XOR dst
QPainter::RasterOp_NotSourceAndNotDestination	27	对源像素和目标像素执行按位 NOR 运算， 即(NOT src) AND (NOT dst)
QPainter::RasterOp_NotSourceOrNotDestination	28	对源像素和目标像素执行按位 NAND 运算， 即(NOT src) OR (NOT dst)
QPainter::RasterOp_NotSourceXorDestination	29	反转源像素，然后与目标像素进行异或，即

		(NOT src) XOR dst
QPainter::RasterOp_NotSource	30	按位反转源像素, 即 NOT src
QPainter::RasterOp_NotSourceAndDestination	31	(NOT src) AND dst
QPainter::RasterOp_SourceAndNotDestination	32	src AND (NOT dst)
QPainter::RasterOp_NorSourceOrDestination	33	(NOT src) OR dst
QPainter::RasterOp_ClearDestination	35	目标中的像素被清除(置为 0)。
QPainter::RasterOp_SetDestination	36	目标中的像素被置为 1。
QPainter::RasterOp_NotDestination	37	NOT dst
QPainter::RasterOp_SourceOrNotDestination	34	src OR (NOT dst)



示例：图像合成示例

```
void paintEvent(QPaintEvent *e) {
    QPainter pr;
    //1、使用预乘 alpha 格式可提高运行速度
    QImage pil(150, 200, QImage::Format_ARGB32_Premultiplied);
    QImage pi2(150, 200, QImage::Format_ARGB32_Premultiplied);
    //2、绘制两个三角形路径
    QPainterPath ph; ph.moveTo(0, 0); ph.lineTo(150, 0); ph.lineTo(150, 180); ph.lineTo(0, 0);
    QPainterPath ph1; ph1.moveTo(0, 0); ph1.lineTo(0, 180); ph1.lineTo(150, 0); ph1.lineTo(0, 0);
    //3、用于填充三角形的画刷
    QBrush bs(QColor(0, 255, 0, 100)); //QColor() 第 4 个参数就是 alpha 通道的值
    QBrush bs1(QColor(255, 0, 0, 255));
    //4、创建图像：在 pil 和 pi2 上使用路径绘制两个三角形
    //注：使用 alpha 为 0 的颜色填充整个图像，可保证在绘制的三角形区域之外的 alpha 通道的值
    //不会对合成产生影响。
    pr.begin(&pil); pil.fill(QColor(255, 255, 255, 0)); pr.fillPath(ph, bs); pr.end();
    pr.begin(&pi2); pi2.fill(QColor(255, 255, 255, 0)); pr.fillPath(ph1, bs1); pr.end();
    //5、开始合成图像，图像需要绘制在 QImage 上才能见到效果。
    pr.begin(&pi2); //这是目标图像，红色
    //可在以下语句使用 QPainter::CompositionMode 的枚举值逐一验证效果。
    pr.setCompositionMode(QPainter::CompositionMode_SourceAtop);
    pr.drawImage(0, 0, pil); //源图像，绿色
    pr.end();
    //6、在 QWidget 上绘制合成后的图像 pi2。
    pr.begin(this); pr.drawImage(11, 11, pi2); pr.end();
    //7、若使用以下语句把两个 QImage 直接在 QWidget 上合成，可能会得不到想要的结果
    /*
    pr.begin(this);
    pr.drawImage(11, 11, pi2);
    pr.setCompositionMode(QPainter::CompositionMode_SourceIn);
    pr.drawImage(11, 11, pil);
    */
}
```

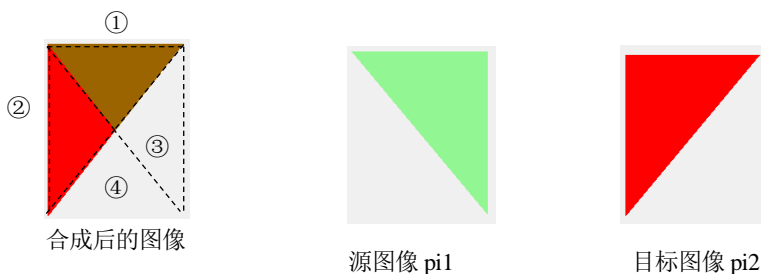


```

        pr.end();      */
//8、以下语句可输出两图像合成后相交区域的颜色。
    QColor c1= pi2.pixelColor(78,30);    qDebug()<<c1;
}

```

运行结果及说明



区域 1 的颜色为 ARGB = [1, 155,100,0]

计算步骤如下：

- 1、计算源和目标图像的 alpha 值和预乘 alpha 颜色。

源图像的 alpha 值为： $S_a = 100/255 \approx 0.392$ ，

源图像的预乘 alpha 颜色为： $S_c = [0, 255*0.392, 0] = [0, 100, 0]$

目标图像的 alpha 值为： $D_a = 255/255 = 1$ ，

目标图像的预乘 alpha 颜色为： $D_c = [255*1, 0, 0] = [255, 0, 0]$

- 2、本示例合成模式为 `QPainter::CompositionMode_SourceAtop`，其计算公式为
 $[D_a, S_c*D_a + D_c*(1 - S_a)]$

- 3、计算区域①的图像

区域 1： $S_a = 0.392$; $D_a = 1$; $D_c = [255, 0, 0]$; $S_c = [0, 100, 0]$

$a1 = [D_a, S_c*D_a + D_c*(1 - S_a)]$

$= [1, (0, 100, 0)*1 + (255, 0, 0)*(1 - 0.392)] = [1, (0, 100, 0) + (155, 0, 0)] = [1, (155, 100, 0)]$

区域 2： $S_a = 0$; $D_a = 1$; $D_c = [255, 0, 0]$; $S_c = [0, 0, 0]$

$a2 = [D_a, S_c*D_a + D_c*(1 - S_a)] = [1, (0, 0, 0)*1 + (255, 0, 0)*(1 - 0)] = [1, (255, 0, 0)]$

区域 3： $S_a = 1$; $D_a = 0$; $D_c = [0, 0, 0]$; $S_c = [0, 100, 0]$

$a3 = [0, S_c*D_a + D_c*(1 - S_a)] = [0, XXX]$ //alpha 为 0，则此区域透明。

区域 4 的 alpha 也为 0。

其余模式读者可自行计算，需要注意的是，在显示 alpha 不为 0 或 1 的图像时，显示的色号并不一定是原色号，比如本示例的 ARGB = [100, 0, 255, 0]；即 alpha 为 100 的纯绿色，显示在 QWidget 上的颜色为 RGB = [146, 246, 146]。

作者：黄邦勇帅(原名：黄勇)

2018-7-13

本文作者：黄邦勇帅(原名：黄勇)

本文要求读者已经非常熟悉 C++ 语法。若读者不熟悉 C++ 语法，推荐参阅《C++ 语法详解》(作者：黄勇)一书，电子工业出版社出版。

本文主要讲解了 Qt 的界面外观，本文列举了详细的示例进行说明，同时本文也是非常方便、快捷的编写 Qt 程序的查阅资料，可方便的查阅到相关内容的原理，以及怎样使用该内容。本文内容由浅入深，易学易懂。

本文使用的是 windows 10 操作系统，Qt 版本为 Qt5.10.1，Qt Creator 的版本为 Qt Creator 4.5.1 本文内容完全属于个人见解与参考文献的作者无关，限于水平有限，其中难免有误解之处，望指出更正。

声明：禁止抄袭、复印、转载本文，本文作者拥有完全版权。

主要参考文献：

- 1、C++语法详解 黄勇 编著 电子工业出版社 2017 年 7 月
- 2、Qt Creator 快速入门(第 3 版) 霍亚飞 编著 北京航空航天大学出版社 2017 年 1 月
- 3、C++ GUI Qt4 编程(第 2 版) [加拿大] Jasmin Blanchette [英] Mark Summerfield 著 闫锋欣 曾泉人 张志强 译 电子工业出版社 2008 年 8 月
- 4、Qt 5 开发及实例 陆文周 主编 电子工业出版社 2014 年 1 月

第 13 章 Qt 界面外观

[13.1 简单的使用 QStyle 类](#)

[13.1.1 样式基础](#)

[13.1.2 QStyleFactory 类及其函数](#)

[13.2 QPalette 类（调色版）](#)

[13.3 自定义部件的外观](#)

[13.4 子类化 QStyle](#)

[13.4.1 样式元素](#)

[13.4.2 样式绘制函数](#)

[13.4.3 子类化 QStyle 类的方法](#)

[13.5 QStyle 类的其他枚举及成员函数](#)

[13.5.1 QStyle::PixMetric 枚举及相关成员函数](#)

[13.5.2 QStyle::StandardPixmap 枚举及相关成员函数](#)

[13.5.3 QStyle::StyleHint 枚举及相关成员函数](#)

[13.5.4 其他枚举及相关成员函数](#)

[13.5.5 QStyle 类中的其他成员函数](#)

[13.6 QStyle 类中枚举的总结](#)

[13.7 QStyleOption（样式选项）及其子类](#)

[13.8 样式表](#)

[13.8.1 样式表基础](#)

[13.8.2 样式表语法基础](#)

[13.8.3 选择器](#)

[13.8.4 子控件](#)

[13.8.5 伪状态](#)

[13.9 样式表的属性](#)

[13.9.1 背景色、前景色、所选文本的颜色](#)

[13.9.2 盒子模型及相关属性](#)

[13.9.3 与位置和大小有关的属性](#)

[13.9.4 字体、文本、图标、图像、不透明度属性](#)

[13.9.5 其他属性](#)

[13.9.6 属性类型](#)

[13.10 设置各部件样式表的方法（综合示例）](#)

[13.10.1 基本规则](#)

[13.10.2 设置各部件样式表的方法](#)

[13.11 样式表的其他规则](#)

[13.11.1 层叠和继承](#)

[13.11.2 名称空间及使用 QObject 属性](#)

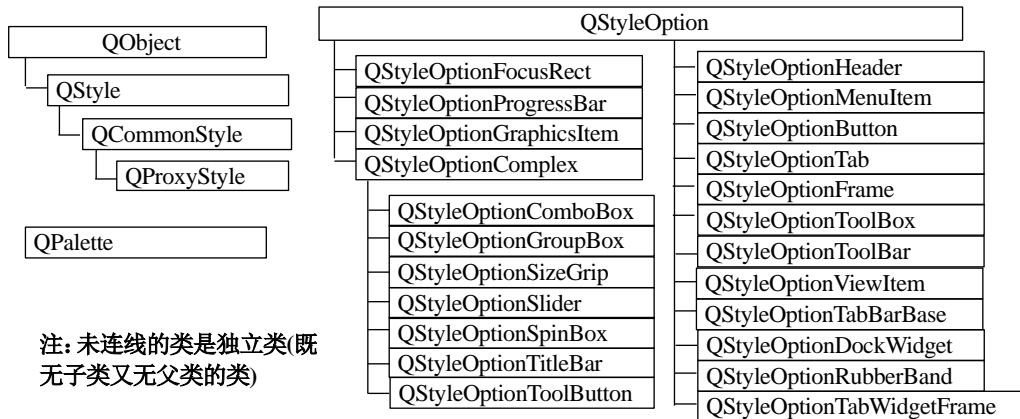
[13.11.3 冲突解决](#)

第 13 部分 Qt 界面外观

注意：本程序都假设读者在 pro 文件中已添加了正确的 `QT+=widgets` 语句，文中不再重复累述添加此语句。

本文注重讲解原理，因此使用的是手写的 Qt 程序。

本章讲解的类及继承关系如下图所示



Qt 界面外观

13.1 简单的使用 QStyle 类(风格也称为样式)

一、样式基础

- 1、QStyle 类继承自 QObject，该类是一个抽象类。
- 2、QStyle 类描述了 GUI 的界面外观，Qt 的内置部件使用该类执行几乎所有的绘制，以确保使这些部件看起来与本地部件完全相同。
- 3、Qt 内置了一系列样式，windows 样式和 fusion 样式默认是可用的，而有些样式需在特定平台上才有用，比如 windowsxp 样式、windowsvista 样式、gtk 样式、macintosh 样式等。
- 4、使用 QStyle 的步骤
 - ①、使用 `QStyleFactory::create()` 静态函数创建一个 QStyle 对象。
 - ②、然后使用以下函数把样式设置到部件或程序中
 - 使用 `QWidget::setStyle()` 函数为某个单个的部件设置样式。
 - 使用 `QApplication::setStyle()` 静态函数来设置整个程序的样式。
 - 还可由应用程序的用户使用 `-style` 命令行选项指定样式，比如
`xxx -style windows //使用 windows 样式。`

- 若未指定样式，则 Qt 将选择与用户的平台或桌面环境最合适的样式。

示例：QStyle 使用示例

```
#include<QtWidgets>
int main(int argc, char *argv[]) {    QApplication aa(argc, argv);
    QWidget w;
    QPushButton *pb1=new QPushButton("AAA", &w); pb1->move(22, 22);
    QPushButton *pb2=new QPushButton("BBB", &w); pb2->move(111, 22);
    QPushButton *pb3=new QPushButton("CCC", &w); pb2->move(222, 22);
    pb1->setStyle(QStyleFactory::create("fusion"));    //仅按钮 pb1 使用系统内置的 fusion 样式
    aa.setStyle(QStyleFactory::create("windows"));    //整个程序使用系统内置的 windows 样式
    w.resize(444, 333);    w.show();    return aa.exec(); }
```

运行结果及说明



二、QStyleFactory 类及其函数

1、QStyleFactory 类是一个独立的类，该类主要用于创建一个 QStyle 对象。

2、QStyleFactory 类的成员函数如下

1)、static QStyle *create(const QString &key); //静态的

创建并返回与键 key 匹配的 QStyle 对象，若没有匹配的 QStyle，则返回 0。来自样式插件的样式和内置样式都会被匹配。注意：key 不区分大小写。

2)、static QStringList keys(); //静态的

返回 QStyleFactory 类可以创建的 QStyle 的键。在创建样式之前，可以使用 qDebug()<<QStyleFactory::keys(); 输出可创建的 QStyle 的键，然后再使用 create()函数创建相应的 QStyle。

13.2 QPalette 类(调色版)

1、需要用到 QWidget 类中的如下属性

palette: QPalette 访问函数: `const QPalette &palette() const;` `void setPalette(const QPalette&);`

- 该属性描述了部件的调色板。在渲染标准部件时，窗口部件的样式会使用调色板，而且不同的平台或不同的样式通常具有不同的调色板。
- 该属性的默认值取决于系统环境，QApplication 维护着一个系统/主题调色板，它是所有部件的默认调色板。
- 注意：不要把此属性与 Qt 样式表联合使用。

2、Qt 的所有部件都含有一个调色板，并且使用各自的调色板来绘制自身。

3、QWidget 会把调色板角色从父级传播到子级，除非启用了 Qt::WA_WindowPropagation 属性(使用 QWidget::setAttribute()函数设置)，否则默认情况下调色板不会传播到窗口。

4、因某些样式依赖于第三方 API(比如 Mac 样式、windows vista 样式等)，而这些样式并不一定遵循调色板，因此不能保证部件的调色板分配角色会改变部件的外观。此时可使用样式表(见后文)

5、另外还可以使用 QApplication::setPalette();静态函数设置整个应用程序的默认调色板。

6、调色板设置的内容通常比较多，因此更改调色板时，通常是使用函数 QWidget::palette()或静态函数 QApplication::palette()获取调色板，然后再对其需要修改的部分进行修改，最后使用相应的 setPalette()函数设置修改后的调色板。

7、QPalette 类(调色板)包含了每个部件状态的颜色组，调色板由 3 个颜色组组成：

- ①、活动(active)：即具有键盘焦点的窗口
- ②、禁用(disabled)：即被禁用的部件(而不是窗口)。
- ③、非活动(inactive)：其他窗口

8、颜色组由枚举 QPalette::ColorGroup 描述，如下表

QPalette::ColorRole 枚举(无标志)					
作用：描述颜色角色					
成员	值	说明	成员	值	说明
QPalette::Disabled	1	标用	QPalette::Inactive	2	非活动
QPalette::Active	0	活动	QPalette::Normal		同 Active

9、QPalette 根据颜色的作用把颜色分为多个角色，并使用枚举 QPalette::ColorRole 进行描述，比如角色 QPalette::Window 表示背景色等，具体见下面的图示及枚举。

QPalette::ColorRole 枚举(无标志)		
作用：描述颜色角色		
成员	值	说明
QPalette::NoRole	17	无角色。
QPalette::Window	10	背景色

QPalette::Background		已过时，同 Window
QPalette::WindowText	0	前景色
QPalette::Foreground		已过时，同 WindowText
QPalette::Base	9	主要用于文本输入部件的背景色，也可用于组合框下拉列表、工具栏的背景，通常是白色或其他浅色。
QPalette::AlternateBase	16	用于视图中带有交替行颜色的备用背景色，详见 <code>QAbstractItemView::setAlternatingRowColors()</code> 函数的详解
QPalette::ToolTipBase	18	QToolTip 和 QWhatsThis 的背景色，QToolTip 使用 QPalette 的非活动颜色组，因为工具提示不是活动窗口。
QPalette::ToolTipText	19	QToolTip 和 QWhatsThis 的前景色
QPalette::Text	6	与 Base 一起使用的前景色，通常与 WindowText 相同，Text 通常用于文本，但也可用于线条、图标等。
QPalette::Button	1	普通按钮背景色
QPalette::ButtonText	8	与 Button 一起使用的前景色。
QPalette::BrightText	7	
以下枚举用于 3D 效果，比如使按钮看更有立体感等，以下枚举依赖于 Window		
QPalette::Light	2	比 Button 的颜色更亮
QPalette::Midlight	3	在 Button 和 Light 之间
QPalette::Dark	4	比 Button 的颜色更暗
QPalette::Mid	5	在 Button 和 Dark 之间
QPalette::Shadow	11	阴影颜色，这是一个很深的颜色，默认使用 Qt::black(黑色)
以下枚举用于选择项目时的颜色。		
QPalette::Highlight	12	突出显示所选内容，默认颜色为 Qt::darkBlue(深蓝)
QPalette::HighlightedText	13	突出显示时的文本颜色，默认颜色为 Qt::white(白色)
以下枚举与超链接有关，注意：富文本不会使用以下枚举值。		
QPalette::Link	14	未访问的超链接的颜色，默认为 Qt::blue(蓝色)
QPalette::LinkVisited	15	已访问过的超链接的颜色，默认为 Qt::magenta(洋红)

9、QPalette 类中的函数

1)、**QPalette()**; //使用应用程序的默认调色板构造一个调色板对象

2)、**QPalette(const QColor &button);**

QPalette(Qt::GlobalColor button);

根据按钮 button 的颜色构造一个调色板，其他颜色根据此颜色自动计算，窗口也将是 button 的颜色。

3)、**QPalette(const QColor &button, const QColor &window);**

根据按钮 button 和窗口颜色构造一个调色板，其他颜色自动计算

4)、**QPalette(const QBrush &windowText, const QBrush &button, const QBrush &light,**

const QBrush &dark, const QBrush &mid, const QBrush &text,

const QBrush &bright_text, const QBrush &base, const QBrush &window);

构造一个调色板，各参数用于设置指定颜色角色。

5)、**QPalette(const QPalette &p);**

QPalette(QPalette &&other);

6)、获取颜色角色的函数见下表

QPalette 类中获取颜色角色的函数

注：1、以下函数的具体意义详见 QPalette::ColorRole 枚举

2、以下函数都省略了返回类型 `const QBrush&` 和小括号后面的 `const`，比如 `base()` 函数的完整原型应为 `const QBrush& base() const`;

<code>window();</code> <code>windowText();</code> <code>base();</code> <code>alternateBase();</code> <code>toolTipBase();</code>	<code>toolTipText();</code> <code>text();</code> <code>button();</code> <code>buttonText();</code> <code>brightText();</code>	<code>light();</code> <code>midlight();</code> <code>dark();</code> <code>mid();</code> <code>shadow();</code>	<code>highlight();</code> <code>highlightedText();</code> <code>link();</code> <code>linkVisited();</code>
--	---	--	---

- 7)、void `setBrush`(ColorRole *role*, const QBrush &*brush*);
把颜色角色 *role* 的画刷 *brush* 设置为调色板中所有组的画刷。
- 8)、void `setBrush`(ColorGroup *group*, ColorRole *role*, const QBrush &*brush*);
把颜色组 *group* 中用于角色 *role* 的画刷设置为 *brush*。
- 9)、const QBrush &`brush`(ColorGroup *group*, ColorRole *role*) const; //返回颜色组 *group* 中角色 *role* 的画刷
const QBrush &`brush`(ColorRole *role*) const; //返回当前颜色组中角色 *role* 的画刷
- 10)、void `setColor`(ColorGroup *group*, ColorRole *role*, const QColor &*color*);
把颜色组 *group* 中用于角色 *role* 的颜色设置为 *color*。
- 11)、void `setColor`(ColorRole *role*, const QColor &*color*);
在所有颜色组中，把颜色角色 *role* 的颜色设置为 *color*。
- 12)、const QColor &`color`(ColorGroup *group*, ColorRole *role*) const; //返回颜色组 *group* 中角色 *role* 的颜色
const QColor &`color`(ColorRole *role*) const; //返回当前颜色组中角色 *role* 的颜色
- 13)、void `setColorGroup`(ColorGroup *cg*, const QBrush &*windowText*, const QBrush &*button*,
const QBrush &*light*, const QBrush &*dark*, const QBrush &*mid*, const QBrush &*text*,
const QBrush &*bright_text*, const QBrush &*base*, const QBrush &*window*);
设置颜色组 *cg* 中各指定颜色角色的画刷，各参数用于指定颜色角色。
- 14)、ColorGroup `currentColorGroup`() const; //返回调色板的当前颜色组
void `setCurrentColorGroup`(ColorGroup *cg*); //把调色板的当前颜色组设置为 *cg*
- 15)、bool `isBrushSet`(ColorGroup *cg*, ColorRole *cr*) const;
若在调色板上设置了颜色组 *cg* 和颜色角色 *cr*，则返回 `true`
- 16)、bool `isCopyOf`(const QPalette &*p*) const;
若此调色板是 *p* 的副本，即，其中一个创建为另一个的副本，且后来没有被修改，则返回 `true`。
- 17)、bool `isEqual`(ColorGroup *cg1*, ColorGroup *cg2*) const; //若 *cg1* 等于 *cg2* 则返回 `true`。
- 18)、QPalette `resolve`(const QPalette &*other*) const; //返回属性是从 *other* 复制的新调色板。
- 19)、void `swap`(QPalette &*other*); //把此调色板与 *other* 交换。
- 20)、qint64 `cacheKey`() const; //返回标识此 QPalette 对象内容的编号。更改调色板时，返回值会更改。

示例：QPalette 的使用

```
#include<QtWidgets>
int main(int argc, char *argv[]) {    QApplication aa(argc, argv);
    QWidget w;
    //向 w 中添加一系列部件
    QPushButton *pb1=new QPushButton("AAA",&w); pb1->move(22, 22);
    QPushButton *pb2=new QPushButton("BBB",&w); pb2->move(111, 22);
```


13.3 自定义部件的外观

一、自定义部件外观基础

- 1、有 3 种方法可实现自定义界面外观：重新实现 `paintEvent()` 函数，使用 `QStyle` 类的绘制函数，子类化 `QStyle`，本小节仅介绍方法 1 和 2 的使用方式，方法 3 见下一节。
- 2、方法一：Qt 通过 `QWidget::paintEvent()` 函数实现界面外观的绘制，因此重新实现 `QWidget::paintEvent()` 函数便可达到绘制部件自定义外观的目的，但这样做工作量比较大，下面举一简单示例对此原理作一说明，以便于对后续内容的理解。

示例：重新实现 `paintEvent()` 函数绘制部件的自定义外观

//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>
class B:public QPushButton{    Q_OBJECT
public:
    bool b;    //用于存储鼠标光标进入或离开按钮的状态
    bool press; //用于存储按钮是否被按下的状态
    B(QString s, QWidget *p1=0):QPushButton(s, p1){    b=0;                press=0;}
    void mousePressEvent(QMouseEvent *e){
        press=1;    //保存按钮被按下的状态
        update();    //更新按钮，此步不可缺少，否则按钮外观不会即时更新。
        QPushButton::mousePressEvent(e);};
    void mouseReleaseEvent(QMouseEvent *e){    press=0;    //按钮未被按下
        update();    QPushButton::mouseReleaseEvent(e);    };
    void enterEvent(QEvent *event){    b=1;    //保存鼠标进入按钮的状态
        update();    QPushButton::enterEvent(event);    }
    void leaveEvent(QEvent *event){    b=0;    //鼠标离开按钮的状态
        update();    QPushButton::leaveEvent(event);    }
    void paintEvent(QPaintEvent *e){    //自定义绘制按钮的外观
        QPainter pr(this);
        QBrush bs(QColor(111,111,111)); //灰色
        QPen pn(Qt::DotLine); //此画笔主要用于绘制焦点框
        pn.setColor(QColor(1,111,1));    pn.setWidth(4);
        QRect r=rect();    //获取设置的按钮的大小。

        if(b==0){    //若鼠标离开按钮
            pr.fillRect(r,bs);    //使用画刷 bs(灰色)填充按钮的背景，其大小为 r
            //绘制按钮上的文本，使用 text() 函数获取设置的按钮的文本。
            pr.drawText(r,Qt::AlignCenter,text());}
        if(b==1){    pr.fillRect(r,QColor(111,1,1)); //红色背景
            pr.drawText(r,Qt::AlignCenter,text()); }
        if(b==1&&press==1){    //光标进入按钮且按钮被按下
            pr.fillRect(r,QColor(222,222,222));    pr.drawText(r,Qt::AlignCenter,text());}
        if(b==0&&press==0){    //光标离开按钮且按钮未被按下
            pr.fillRect(r,bs);    pr.drawText(r,Qt::AlignCenter,text());}
        if(hasFocus())    //{按钮获得焦点
```

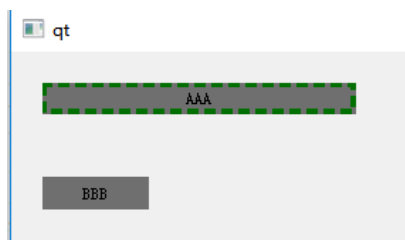
```

        pr.setPen(pn);pr.drawRect(r.adjusted(1,1,-2,-2)); } //绘制一个矩形边框
    }
#endif // M_H

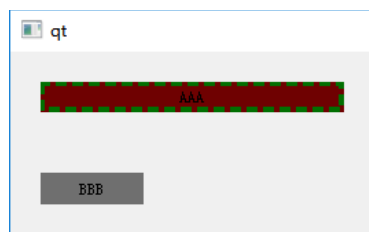
//m.cpp 文件的内容
#include "m.h"
int main(int argc, char *argv[]){    QApplication aa(argc,argv);
    QWidget w;
    //使用自定义外观的按钮
    B *pb1=new B("AAA",&w); pb1->move(22,22);    pb1->resize(221,22);
    B *pb2=new B("BBB",&w); pb2->move(22,88);
    w.resize(444,333);    w.show();    return aa.exec(); }

```

运行结果及说明



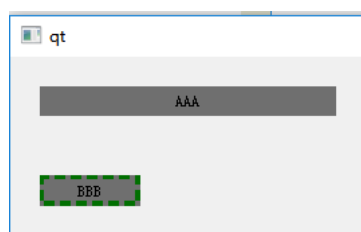
初始状态：按钮背景为灰色的



鼠标进入按钮 AAA 时，其背景变为红色



鼠标按下按钮 AAA 时，其背景变为白色



按下 tab 键，绿色虚线焦点框被移至按钮 BBB

3、方法二：自定义外观还可以使用 QStyle 类中的绘制函数来绘制，通常 QStyle 类的绘制函数需要如下 4 个参数：

- ①、一个 QStyle 枚举值：用于指定需要绘制什么类型的图形元素
- ②、一个 QStyleOption 或其子类对象(样式选项)，样式选项包含了需要绘制的图形元素的所有信息，比如包含了图形元素的文本、调色板等。根据绘制的内容，样式需要不同的样式选项类，比如 QStyle::CE_PushButton 元素，需要一个 QStyleOptionButton 类型的参数。
- ③、一个用于绘制图形的 QPainter
- ④、执行绘制的 QWidget(可选)，通常是需要绘制的元素的部件。
- ⑤、可使用 QStylePainter 类中的绘制函数代替 QStyle 类的相应绘制函数，其主要好处是可减少调用函数时的实参个数。

示例：使用 QStyle 类的绘制函数自定义部件样式

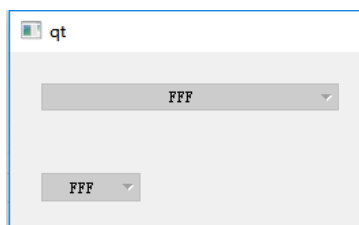
//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>
class B:public QPushButton{    Q_OBJECT    //需子类化 QPushButton
public:    B(QString s,QWidget *p1=0):QPushButton(s,p1){}
    void paintEvent(QPaintEvent *e){        //需要重新实现该函数以绘制自定义的按钮样式
        QPainter pr(this);
        QStyleOptionButton psb;    //创建一个按钮样式选项
        psb.rect=rect();    //设置绘制的按钮的大小，此步不可省略，否则绘制的按钮将不可见
        psb.text="FFF";        //设置按钮上显示的文本，要使按钮被设置为用户创建按钮时的文本，
                                //可使用 psb.text=text();代替此语句。
        psb.features=QStyleOptionButton::HasMenu;    //设置按钮的样式为带有下拉菜单的按钮
        //获取该按钮的样式对象，并使用该样式的 drawControl() 函数绘制按钮。
        style()->drawControl(QStyle::CE_PushButton, &psb, &pr, this);
        //也可使用如下代码绘制
        //QStylePainter sp;
        //与 psb 有关的代码不变
        //sp.drawControl(QStyle::CE_PushButton,&psb);
    } };
```

//m.cpp 文件的内容

```
#include "m.h"
int main(int argc, char *argv[]){    QApplication aa(argc,argv);
    QWidget w;
    B *pb1=new B("AAA",&w); pb1->move(22,22); pb1->resize(221,22);
    B *pb2=new B("BBB",&w); pb2->move(22,88);
    w.resize(444,333);    w.show();    return aa.exec(); }
```

运行结果及说明



两按钮的文本均为 FFF，在 main 函数中设置的文本不起作用。要使绘制的按钮显示设置的文本，需使用 psb.text=text();

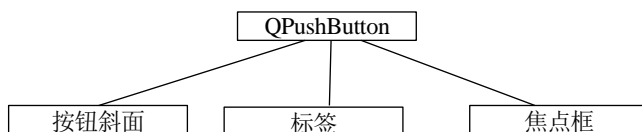
13.4 子类化 QStyle

1、把绘制自定义部件外观的步骤大致分为三大板块，如下：

- ①、样式元素：即指定需要绘制的图形元素(比如焦点框、按钮，工具栏等)。样式元素使用 `QStyle` 类中的一系列枚举(共有 11 个枚举)进行描述。
- ②、样式选项：包含了需要绘制的图形元素的所有信息，比如包含了图形元素的文本、调色板等，样式选项(见下一小节)使用 `QStyleOption` 及其子类进行描述。
- ③、样式绘制函数：即绘制图形元素的函数，这些函数是 `QStyle` 类的成员函数，比如 `drawControl()`等(在前文已见过)

一、样式元素

1、样式元素是指需要绘制的图形元素，一个部件由多个样式元素组成，比如，当样式接收到绘制按钮的请求时，会绘制标签(文本和图标)，按钮斜面(Bevel)和焦点框，下图为一个按钮元素的组成概念图。



2、共有 3 种样式元素

- ①、原始元素：由 `QStyle::PrimitiveElement` 枚举描述，使用前缀“PE_”标识，原始元素通常由部件使用，包括框架、按钮斜面、旋转框、滚动条、组合框的箭头等，原始元素不能单独存在，是其他界面元素(如复杂控件和控件元素)的组成部分，不参与用户的交互，只是被动的绘制和显示。
 - ②、控件元素：由 `QStyle::ControlElement` 枚举描述，使用前缀“CE_”标识。控件元素执行某种操作或向用户显示一些信息，控件元素会与用户交互，可以是单独的部件(比如 `QPushButton`)，也可以是其他部件的一部分(比如 `QScrollBar` 的滑块)。控件元素还可以由多个子元素组成，子元素仅用于计算边界区域，不能被绘制，然后便可在计算出的边界区域中绘制子元素，由多个子元素组成的控件使用 `QStyle::SubElement` 枚举描述(使用前缀 SE_标识)。
 - ③、复杂控件元素：由 `QStyle::ComplexControl` 枚举描述，使用前缀“CC_”标识。复杂控件元素包含子控件，比如 `QSpinBox`、`QScrollBar`、`QToolButton` 等。复杂控件元素的子控件由 `QStyle::SubControl` 枚举定义(使用前缀 SC_标识)。
 - ④、因为部件通常都是由矩形组成的，因此各种控件元素其实描述的就是一个矩形区域。
- 3、下面列出描述各种样式元素的枚举及元素的状态标志，各种样式元素之间的关系，会在后文讲解，此处暂不讲解

QStyle::ControlElement 枚举(无标志) --1

作用：控件元素，与 drawControl()函数相关

	成员	值	说明
① 按钮	QStyle::CE_PushButton	0	QPushButton, 绘制 CE_PushButtonBevel, CE_PushButtonLabel, PE_FrameFocusRect
	QStyle::CE_PushButtonBevel	1	QPushButton 的斜面
	QStyle::CE_PushButtonLabel	2	QPushButton 的标签(即文本或图标)
② 复选 按钮	QStyle::CE_CheckBox	3	QCheckBox, 绘制 CE_CheckBoxLabel、PE_IndicatorCheckBox、PE_FrameFocusRect
	QStyle::CE_CheckBoxLabel	4	QCheckBox 的标签(文本或像素图)
③ 单选 按钮	QStyle::CE_RadioButton	5	QRadioButton, 还会绘制 CE_RadioButtonLabel、PE_IndicatorRadioButton、PE_FrameFocusRect
	QStyle::CE_RadioButtonLabel	6	QRadioButton 的标签(文本或像素图)
④ 工具 栏	QStyle::CE_TabBarTab	7	QTabBar 的选项卡和标签
	QStyle::CE_TabBarTabShape	8	TabBar 的选项卡的形状
	QStyle::CE_TabBarTabLabel	9	TabBar 的标签
⑤ 进度 条	QStyle::CE_ProgressBar	10	QProgressBar, 绘制 CE_ProgressBarGroove、CE_ProgressBarContents、CE_ProgressBarBarLabel
	QStyle::CE_ProgressBarGroove	11	QProgressBar 进度指示器的凹槽
	QStyle::CE_ProgressBarContents	12	QProgressBar 的进度指示器
	QStyle::CE_ProgressBarBarLabel	13	QProgressBar 的文本标签
⑥ 菜单 相关	QStyle::CE_MenuBarItem	20	QMenuBar 中的菜单项
	QStyle::CE_MenuBarEmptyArea	21	QMenuBar 的空白区域
	QStyle::CE_MenuItem	14	QMenu 中的菜单项
	QStyle::CE_MenuScroller	15	当样式支持滚动时, 在 QMenu 中的滚动区域
	QStyle::CE_MenuTearoff	18	菜单的可分离部分
	QStyle::CE_MenuEmptyArea	19	菜单中没有菜单项的区域
	QStyle::CE_MenuHMargin	17	菜单左/右的水平额外空间
⑦ 模型/ 视图 相关	QStyle::CE_MenuVMargin	16	菜单项/底部的垂直额外空间
	QStyle::CE_Header	23	标头
	QStyle::CE_HeaderSection	24	标头段
	QStyle::CE_HeaderLabel	25	标头的标签
	QStyle::CE_StandardItem	45	模型/视图中的项目
⑧ 滚动 条	QStyle::CE_HeaderEmptyArea	43	标头视图中没有标头段的区域
	QStyle::CE_ScrollBarAddLine	31	滚动条的行增加指示器(即向下滚动)
	QStyle::CE_ScrollBarSubLine	32	滚动条的行减少指示器(即向上滚动)
	QStyle::CE_ScrollBarAddPage	33	滚动条的页面增加指示器(即向下翻页)
	QStyle::CE_ScrollBarSubPage	34	滚动条的页面减少指示器(即向上翻页)
	QStyle::CE_ScrollBarSlider	35	滚动条的滑块
	QStyle::CE_ScrollBarFirst	36	滚动条的第一行指示器(即 home)
⑨ 其他 部件	QStyle::CE_ScrollBarLast	37	滚动条的最后一行指示器(即 end)
	QStyle::CE_RubberBand	29	橡皮筋
	QStyle::CE_FocusFrame	38	样式控件的焦点框
	QStyle::CE_DockWidgetTitle	30	可停靠窗口标题
	QStyle::CE_Splitter	28	QSplitter
	QStyle::CE_ComboBoxLabel	39	不可编辑的 QComboBox 标签
	QStyle::CE_SizeGrip	27	QSizeGrip
	QStyle::CE_ToolButtonLabel	22	工具按钮的标签
	QStyle::CE_ToolBar	40	QToolBar
	QStyle::CE_ToolBoxTab	26	QToolBox 的标签和选项卡

	QStyle::CE_ToolBoxTabShape	41	工具箱的选项卡形状
	QStyle::CE_ToolBoxTabLabel	42	工具签的选项卡标签
	QStyle::CE_ShapeFrame	46	具有 QStyleOptionFrame 中指定形状的边框。
	QStyle::CE_CustomBase		自定义控件元素的基础值(其值为 0xf0000000), 自定义值必须大于此值

QStyle::ComplexControl 枚举(无标志) --2

作用：复杂控件元素，与 drawComplexControl()函数相关

	成员	值	说明
	QStyle::CC_SpinBox	0	像 QSpinBox 一样的控件
	QStyle::CC_ComboBox	1	组合框，如 QComboBox
	QStyle::CC_ScrollBar	2	滚动条，如 QScrollBar
	QStyle::CC_Slider	3	滑块，如 QSlider
	QStyle::CC_ToolButton	4	工具按钮，如 QToolButton
	QStyle::CC_TitleBar	5	标题栏，与 QMdiSubWindow 中使用的标题栏类似
	QStyle::CC_GroupBox	7	组框，如 QGroupBox
	QStyle::CC_Dial	6	表盘，如 QDial
	QStyle::CC_MdiControls	8	最大化 MDI 子窗口的菜单栏中的最小化、关闭、正常按钮
	QStyle::CC_CustomBase		自定义复杂控件的基础值(其值为 0xf0000000), 自定义值必须大于此值

QStyle::PrimitiveElement 枚举(无标志) --3

作用：原始元素，与 drawPrimitive()函数相关

	成员	值	说明
① 按钮	QStyle::PE_PanelButtonCommand	?	用于启动动作(action)的按钮，如 QPushButton
	QStyle::PE_FrameDefaultButton	1	围绕默认按钮的边框
	QStyle::PE_PanelButtonBevel		带按钮斜面的通用面板
	QStyle::PE_FrameButtonBevel		按钮斜面的面板边框
	QStyle::PE_IndicatorButtonDropDown		下拉按钮的指示器，比如显示菜单的工具按钮
	QStyle::PE_IndicatorCheckBox		QCheckBox 的开/关指示器
	QStyle::PE_IndicatorRadioButton		QRadioButton 的开/关指示器
② 行编辑器	QStyle::PE_PanelLineEdit		QLineEdit 的面板
	QStyle::PE_FrameLineEdit	5	行编辑器的边框
③ 其他	QStyle::PE_FrameFocusRect	3	焦点边框指示器
	QStyle::PE_Frame		通用边框
	QStyle::PE_Widget		一个 QWidget 面板
④ 可停靠窗口	QStyle::PE_IndicatorDockWidgetResizeHandle		调整可停靠窗口的手柄大小
	QStyle::PE_FrameDockWidget	2	可停靠窗口和工具栏的面板边框
⑤ 滚动区域	QStyle::PE_FrameWindow		围绕 MDI 窗口或可停靠窗口的边框
	QStyle::PE_PanelScrollAreaCorner		在滚动区域右下角(或左下角)的面板
组框	QStyle::PE_FrameGroupBox	4	围绕组框周围的面板边框
⑥ 工具	QStyle::PE_IndicatorToolBarHandle		工具栏的手柄
	QStyle::PE_IndicatorToolBarSeparator		工具栏的分隔符

栏	QStyle::PE_PanelToolBar		工具栏的面板
⑦ 工具按钮	QStyle::PE_PanelButtonTool		工具按钮的面板，与 QToolButton 一起使用
	QStyle::PE_FrameButtonTool		工具按钮的面板边框
⑧ 状态栏 进度条	QStyle::PE_PanelTipLabel		提示标签的面板
	QStyle::PE_PanelStatusBar		状态栏的面板
	QStyle::PE_FrameStatusBar	7	已过时，改用 PE_FrameStatusBarItem
	QStyle::PE_FrameStatusBarItem		状态栏(QStatusBar)的边框
	QStyle::PE_IndicatorProgressChunk		进度条(QProgressBar)指示器的一部分。
⑨ 微调按钮及箭头	QStyle::PE_IndicatorArrowUp		向上箭头
	QStyle::PE_IndicatorArrowDown		向下箭头
	QStyle::PE_IndicatorArrowRight		右箭头
	QStyle::PE_IndicatorArrowLeft		左箭头
	QStyle::PE_IndicatorSpinUp		微调按钮(QSpinBox)的向上符号
	QStyle::PE_IndicatorSpinDown		微调按钮的向下符号
	QStyle::PE_IndicatorSpinPlus		微调按钮的增加符号
	QStyle::PE_IndicatorSpinMinus		微调按钮的减小符号
10 选项卡相关	QStyle::PE_FrameTabBarBase		选项卡条(QTabBar)绘制的边框
	QStyle::PE_IndicatorTabTear		已过时，改用 PE_IndicatorTabTearLeft
	QStyle::PE_IndicatorTabTearLeft		当有多个选项卡时，在可见选项卡栏的左侧部分滚动的指示器。
	QStyle::PE_IndicatorTabTearRight		当有多个选项卡时，在可见选项卡栏的右侧部分滚动的指示器。
	QStyle::PE_IndicatorTabClose		选项卡条上的关闭按钮
	QStyle::PE_FrameTabWidget		选项卡部件的边框
⑪ 菜单	QStyle::PE_FrameMenu	6	弹出窗口/菜单的边框
	QStyle::PE_PanelMenuBar		菜单栏的面板
	QStyle::PE_PanelMenu		菜单的面板
	QStyle::PE_IndicatorMenuCheckMark		在菜单中使用的复选标记
⑫ 项目/视图相关	QStyle::PE_IndicatorItemViewItemDrop		在项目视图的拖放操作中，绘制拖放的项将被放置在何处的指示器。
	QStyle::PE_PanelItemViewItem		项目视图中项的背景
	QStyle::PE_PanelItemViewRow		项目视图中行的背景
	QStyle::PE_IndicatorBranch		树视图中标示树分支的线条
	QStyle::PE_IndicatorItemViewItemCheck		视图项目的开/关指示器
	QStyle::PE_IndicatorColumnViewArrow		QColumnView 中的箭头
	QStyle::PE_IndicatorHeaderArrow		列表或表标头上排序的箭头
	QStyle::PE_CustomBase		自定义原始元素的基础值(其值为 0xf00000000)，自定义值必须大于此值

QStyle::SubControl 枚举 --4

标志：QStyle::SubControls

作用：子控件，可由函数 subControlRect()获取指定枚举值对应的矩形区域

	成员	值	说明
	QStyle::SC_None	0x0000 0000	与其他子控件不匹配的特殊值
	QStyle::SC_All	0xffff ffff	与所有子控件匹配的特殊值
① 滚动	QStyle::SC_ScrollBarAddLine	0x0000 0001	滚动条增加行(即向下/向右箭头)
	QStyle::SC_ScrollBarSubLine	0x0000 0002	滚动条减少行(即向上/向左箭头)

条	QStyle::SC_ScrollBarAddPage	0x0000 0004	滚动条增加页(即 page down)
	QStyle::SC_ScrollBarSubPage	0x0000 0008	滚动条减少页(即 page up)
	QStyle::SC_ScrollBarFirst	0x0000 0010	滚动条行首(即 home)
	QStyle::SC_ScrollBarLast	0x0000 0020	滚动条行尾(即 end)
	QStyle::SC_ScrollBarSlider	0x0000 0040	滚动条滑块手柄
	QStyle::SC_ScrollBarGroove	0x0000 0080	特殊子控件, 它包含滑块手柄可移动的区域
② 微调 按钮	QStyle::SC_SpinBoxUp	0x0000 0001	微调按钮(QSpinBox)的向上/增加按钮
	QStyle::SC_SpinBoxDown	0x0000 0002	微调按钮的向下/减少按钮
	QStyle::SC_SpinBoxFrame	0x0000 0004	微调按钮的边框
	QStyle::SC_SpinBoxEditField	0x0000 0008	微调按钮的编辑字段
③ 组合 框	QStyle::SC_ComboBoxEditField	0x0000 0002	组合框(QComboBox)的编辑字段
	QStyle::SC_ComboBoxArrow	0x0000 0004	组合框的箭头按钮
	QStyle::SC_ComboBoxFrame	0x0000 0001	组合框的边框
	QStyle::SC_ComboBoxListBoxPopu p	0x0000 0008	组合框弹出窗口的参考矩形。
④ 滑块	QStyle::SC_SliderGroove	0x0000 0001	特殊子控件, 它包含滑块手柄可移动的区域
	QStyle::SC_SliderHandle	0x0000 0002	滑块手柄
	QStyle::SC_SliderTickmarks	0x0000 0004	滑块刻度线
工具 按钮	QStyle::SC_ToolButton	0x0000 0001	工具按钮(QToolButton)
	QStyle::SC_ToolButtonMenu	0x0000 0002	用于在工具按钮中打开弹出菜单的子控件
⑥ 标题 栏	QStyle::SC_TitleBarSysMenu	0x0000 0001	系统菜单按钮(即还原、关闭等)
	QStyle::SC_TitleBarMinButton	0x0000 0002	最小化按钮
	QStyle::SC_TitleBarMaxButton	0x0000 0004	最大化按钮
	QStyle::SC_TitleBarCloseButton	0x0000 0008	关闭按钮
	QStyle::SC_TitleBarLabel	0x0000 0100	窗口标题标签
	QStyle::SC_TitleBarNormalButton	0x0000 0010	正常(恢复)按钮
	QStyle::SC_TitleBarShadeButton	0x0000 0020	阴影按钮
	QStyle::SC_TitleBarUnshadeButton	0x0000 0040	无阴影的按钮
	QStyle::SC_TitleBarContextHelpBut ton	0x0000 0080	上下文帮助按钮
⑦ 表盘	QStyle::SC_DialHandle	0x0000 0002	表盘(QDial)的手柄
	QStyle::SC_DialGroove	0x0000 0001	表盘的凹槽
	QStyle::SC_DialTickmarks	0x0000 0004	表盘的刻度线
⑧ 组框	QStyle::SC_GroupBoxFrame	0x0000 0008	组框的边框
	QStyle::SC_GroupBoxLabel	0x0000 0002	组框的标题
	QStyle::SC_GroupBoxCheckBox	0x0000 0001	组框的复选框
	QStyle::SC_GroupBoxContents	0x0000 0004	组框的内容
	QStyle::SC_MdiNormalButton	0x0000 0002	菜单栏中 MDI 子窗口的常规按钮
	QStyle::SC_MdiMinButton	0x0000 0001	菜单栏中 MDI 子窗口的最小化按钮
	QStyle::SC_MdiCloseButton	0x0000 0004	菜单栏中 MDI 子窗口的关闭按钮

QStyle::SubElement 枚举(无标志) --5

作用: 子元素, 可由函数 subElementRect()获取指定枚举值对应的矩形区域

	成员	值	说明
① 按钮	QStyle::SE_PushButtonContents	0	包含标签(带有文本或像素图的图标)的区域
	QStyle::SE_PushButtonFocusRect	1	焦点区域(通常大于内容区域)
	QStyle::SE_PushButtonLayoutItem	?	父布局的区域
②	QStyle::SE_CheckBoxIndicator	2	状态指示器的区域(比如, 复选标记的区域)

复选按钮	QStyle::SE_CheckBoxContents	3	复选框的标签(文本或像素图)区域
	QStyle::SE_CheckBoxFocusRect	4	复选框的焦点区域
	QStyle::SE_CheckBoxClickRect	5	可点击区域, 默认为 SE_CheckBoxFocusRect
	QStyle::SE_CheckBoxLayoutItem	?	父布局的区域
③ 单选按钮	QStyle::SE_RadioButtonIndicator	6	状态指示器的区域
	QStyle::SE_RadioButtonContents	7	标签的区域
	QStyle::SE_RadioButtonFocusRect	8	焦点区域
	QStyle::SE_RadioButtonClickRect	9	可点击区域
	QStyle::SE_RadioButtonLayoutItem	?	父布局的区域
④ 滑块	QStyle::SE_SliderFocusRect	11	焦点区域
	QStyle::SE_SliderLayoutItem	?	父布局的区域
⑤ 进度条	QStyle::SE_ProgressBarGroove	12	凹槽区域
	QStyle::SE_ProgressBarContents	13	进度指示器的区域
	QStyle::SE_ProgressBarLabel	14	文本标签区域
	QStyle::SE_ProgressBarLayoutItem	?	父布局的区域
⑥ 选项卡部件	QStyle::SE_TabWidgetLeftCorner	21	选项卡部件左角部件的区域
	QStyle::SE_TabWidgetRightCorner	22	选项卡部件右角部件的区域
	QStyle::SE_TabWidgetTabBar	18	选项卡部件选项卡部件的区域
	QStyle::SE_TabWidgetTabContents	20	选项卡部件内容的区域
	QStyle::SE_TabWidgetTabPage	19	选项卡部件面板的区域
	QStyle::SE_TabWidgetLayoutItem	?	父布局的区域
⑦ 选项卡栏	QStyle::SE_TabBarTearIndicator	?	已过时, 改用 SE_TabBarTearIndicaotLeft
	QStyle::SE_TabBarTearIndicatorLeft		带有滚动箭头的选项卡栏左侧的可分离指示器区域
	QStyle::SE_TabBarTearIndicatorRight	?	带有滚动箭头的选项卡栏右侧的可分离指示器区域
	QStyle::SE_TabBarScrollLeftButton	?	带有滚动箭头的选项卡栏滚动左按钮的区域
	QStyle::SE_TabBarScrollRightButton	?	带有滚动箭头的选项卡栏滚动右按钮的区域
	QStyle::SE_TabBarTabLeftButton	?	选项卡栏中选项卡左侧部件的区域
	QStyle::SE_TabBarTabRightButton	?	选项卡栏中选项卡右侧部件的区域
	QStyle::SE_TabBarTabText	?	选项卡栏的文本区域
	QStyle::SE_TabBarHandle	?	工具栏手柄的区域
⑧ 可停靠窗口	QStyle::SE_DockWidgetFloatButton	?	可停靠窗口的浮动按钮
	QStyle::SE_DockWidgetTitleBarText	?	可停靠窗口标题的文本边界
	QStyle::SE_DockWidgetColseButton	?	可停靠窗口的关闭按钮
	QStyle::SE_DockWidgetIcon	?	可停靠窗口的图标
⑨ 其他部件	QStyle::SE_ComboBoxFocusRect	10	焦点区域
	QStyle::SE_ComboBoxLayoutItem		父布局的区域
	QStyle::SE_DateTimeEditLayoutItem	?	父布局的区域
	QStyle::SE_LabelLayoutItem	?	父布局的区域
	QStyle::SE_LineEditContents	?	行编辑器内容的区域
	QStyle::SE_ToolBoxTabContents	15	工具箱选项卡图标和标签的区域
	QStyle::SE_ToolButtonLayoutItem	?	父布局的区域
	QStyle::SE_DialogButtonBoxLayoutItem	?	父布局的区域
	QStyle::SE_GroupBoxLayoutItem	?	父布局的区域
	QStyle::SE_SpinBoxLayoutItem	?	父布局的区域
⑩ 其他	QStyle::SE_FrameContents	?	框架内容的区域
	QStyle::SE_FrameLayoutItem	?	父布局的区域

	QStyle::SE_ShapeFrameContents	?	使用 QStyleOptionFrame 中的 shape 的边框内容的区域
⑪ 模型/ 视图 相关	QStyle::SE_ItemViewItemDecoratio n	?	视图项目的装饰区域(图标)
	QStyle::SE_ItemViewItemText	?	视图项目的文本区域
	QStyle::SE_ItemViewItemFocusRect	?	视图项目的焦点区域
	QStyle::SE_TreeViewDisclosureItem	?	树枝中实际公开项目的区域
	QStyle::SE_HeaderArrow	17	标头的排序指示器的区域
	QStyle::SE_HeaderLabel	16	标头中的标签区域
	QStyle::SE_ItemViewItemCheckIndi cator		视图项目的复选标记区域
	QStyle::SE_CustomBase		自定义子元素的基础值(其值为 0xf00000000)，自定义值必须大于此值

QStyle::StateFlag 枚举 --6

标志: QStyle::State

作用: 元素的状态标志, 通常由 QStyleOption 及其子类使用

成员	值	说明
QStyle::State_None	0x0000 0000	无状态
QStyle::State_Active	0x0001 0000	活动状态
QStyle::State_AutoRaise	0x0000 1000	工具按钮是否自动凸起
QStyle::State_Children	0x0008 0000	项目视图分支是否具有子项
QStyle::State_DownArrow	0x0000 0040	显示向下箭头
QStyle::State_Editing	0x0040 0000	可编辑
QStyle::State_Enabled	0x0000 0001	启用部件
QStyle::State_HasEditFocus	0x0100 0000	部件当前具有编辑焦点
QStyle::State_HasFocus	0x0000 0100	部件具有焦点
QStyle::State_Horizontal	0x0000 0080	部件水平布局, 比如, 工具栏。
QStyle::State_KeyboardFocusChange	0x0080 0000	焦点是否可通过键盘更改(如, Tab、快捷键等)
QStyle::State_MouseOver	0x0000 2000	部件位于鼠标下面
QStyle::State_NoChange	0x0000 0010	用于指示三态复选框
QStyle::State_Off	0x0000 0008	部件未被选中
QStyle::State_On	0x0000 0020	部件被选中
QStyle::State_Raised	0x0000 0002	按钮是凸起状态(通常为未被按下的状态)。
QStyle::State_ReadOnly	0x0200 0000	只读
QStyle::State_Selected	0x0000 8000	被选择
QStyle::State_Item	0x0010 0000	项目视图是否应绘制水平分支
QStyle::State_Open	0x0004 0000	项目视图的树分支是否已打开
QStyle::State_Sibling	0x0020 0000	项目视图是否需要绘制垂直线
QStyle::State_Sunken	0x0000 0004	部件被按下或是凹陷状态的
QStyle::State_UpArrow	0x0000 4000	是否显示向上箭头
QStyle::State_Mini	0x0800 0000	mini 样式的 Mac 部件或按钮
QStyle::State_Small	0x0400 0000	small 样式的 Mac 部件或按钮

二、样式绘制函数

1、为讲解需要, 此处先列出需要使用到的 QStyle 类中的部分函数, 其余函数见后文

2、virtual void QStyle::drawComplexControl(ComplexControl *control*, const QStyleOptionComplex **option*, QPainter **painter*, const QWidget **widget* = Q_NULLPTR) const = 0;

- 该函数用于绘制复杂控件元素。表示使用 painter，样式选项 option 绘制控件 control。
- widget 参数是可选的，可用作绘制控件的辅助工具。
- 参数 option 是指向 QStyleOptionComplex 对象的指针，可使用 qstyleoption_cast()函数把该对象转换为正确的子类型。
- option 的 rect 成员变量必须位于逻辑坐标中，重新实现此函数，在调用 QStyle::drawPrimitive()或 QStyle::drawControl()函数之前，应使用 QStyle::visualRect()函数把逻辑坐标转换为屏幕坐标。
- 下表为复杂控件元素及其关联的样式选项子类，以及使用的相应的状态标志。

复杂控件元素及其关联的样式选项子类			
复杂控件元素	QStyleOptionComplex 的子类型	状态标志	说明
CC_SpinBox	QStyleOptionSpinBox	State_Enabled State_HasFocus	是否启用该部件 部件具有输入焦点
CC_ComboBox	QStyleOptionComboBox	State_Enabled State_HasFocus	
CC_ScrollBar	QStyleOptionSlider	State_Enabled State_HasFocus	
CC_Silder	QStyleOptionSlider	State_Enabled State_HasFocus	
CC_Dial	QStyleOptionSlider	State_Enabled State_HasFocus	
CC_ToolButton	QStyleOptionToolButton	State_Enabled	
		State_HasFocus	
		State_DownArrow	工具按钮被按下
		State_On	工具按钮是切换按钮且打开的
		State_AutoRaise	工具按钮启用了自动提升
		State_Raised	若工具按钮未按下、不是切换的，且当启用自动提升时不包含鼠标，则设置
CC_TitleBar	QStyleOptionTitleBar	State_Enabled	

3、virtual void QStyle::drawControl(ControlElement *element*, const QStyleOption **option*, QPainter **painter*, const QWidget **widget* = Q_NULLPTR) const = 0;

- 该函数用于绘制控件元素。表示使用 painter，样式选项 option 绘制元素 element。
- widget 参数是可选的，可用作绘制控件的辅助工具。
- 参数 option 是指向 QStyleOption 对象的指针，可使用 qstyleoption_cast()函数把该对象转换为正确的子类型。
- 下表为控件元素及其关联的样式选项子类，以及使用的相应的状态标志。

控件元素及其关联的样式选项子类			
控件元素	QStyleOption 的子类型	状态标志	说明
CE_MenuItem	QStyleOptionMenuIte	State_Selected	菜单项当前被选中

CE_MenuBarItem	m	State_Enabled	启用该菜单项
		State_DownArrow	指示绘制向下箭头
		State_UpArrow	指示绘制向上箭头
		State_HasFocus	具有输入焦点
CE_PushButton CE_PushButtonBevel CE_PushButtonLabel	QStyleOptionButton	State_Enabled	启用该按钮
		State_HasFocus	具有输入焦点
		State_Raised	按钮未被按下、不是切换按钮、不是平的，则设置
		State_On	按钮是切换按钮且是打开的
		State_Sunken	按钮被按下，则设置
CE_RadioButton CE_RadioButtonLabel CE_CheckBox CE_CheckBoxLabel	QStyleOptionButton	State_Enabled	启用该按钮
		State_HasFocus	具有输入焦点
		State_On	该按钮被选中
		State_Off	该按钮未被选中
		State_NoChange	若按钮处理 NoChange 状态(三态)，则设置
		State_Sunken	按钮被按下，则设置
CE_ProgressBarContents CE_ProgressBarLabel CE_ProgressBarGroove	QStyleOptionProgress Bar	State_Enabled	启用该进度条
		State_HasFocus	具有输入焦点
CE_Header CE_HeaderSection CE_HeaderLabel	QStyleOptionHeader		
CE_TabBarTab CE_TabBarTabShape CE_TabBarTabLabel	QStyleOptionTab	State_Enabled State_Selected State_HasFocus	启用选项卡栏 选项卡当前被选中 具有输入焦点
CE_ToolButtonLabel	QStyleOptionToolButt on	State_Enabled	启用工具按钮
		State_HasFocus	具有输入焦点
		State_Sunken	工具按钮被按下，则设置
		State_On	工具按钮是切换的且是打开的
		State_AutoRaise	工具按钮启用了自动提升
		State_MouseOver	鼠标位于工具按钮上方
		State_Raised	若工具按钮未按下、不是切换的，且当启用自动提升时不包含鼠标，则设置
CE_ToolBoxTab	QStyleOptionToolBox	State_Selected	选项卡当前被选中
CE_HeaderSection	QStyleOptionHeader	State_Sunken State_UpArrow State_DownArrow	该段被按下 表示排序指示器应指向上方 表示排序指示器应指向下方

4、virtual void QStyle::drawPrimitive(PrimitiveElement *element*, const QStyleOption **option*, QPainter **painter*,
const QWidget **widget* = Q_NULLPTR) const = 0;

- 该函数用于绘制原始元素。表示使用 painter，样式选项 option 绘制元素 element。
- widget 参数是可选的，可用作绘制的辅助工具。
- 下表为原始元素及其关联的样式选项子类，以及使用的相应的状态标志。

原始元素及其关联的样式选项子类			
原始元素	QStyleOption 的子类	状态标志	说明

	型		
PE_FrameFocusRect	QStyleOptionFocusRect	State_FocusAtBorder	无论焦点是在边框还是部件内部
PE_IndicatorCheckBox	QStyleOptionButton	State_NoChange State_On	指示三态复选框 指示被选中指示器
PE_IndicatorRadioButton	QStyleOptionButton	State_On	表示以选中单选按钮
PE_IndicatorBranch	QStyleOption	State_Children	指示用于展开树以显示子项的控件, 应该被绘制
		State_Item	表示应绘制水平分支(以显示子项)
		State_Open	表示树分支以展开
		State_Sibling	表示应绘制垂直线(以显示同级项)
PE_IndicatorHeaderArrow	QStyleOptionHeader	State_UpArrow	表示应该绘制处上箭头
PE_FrameGroupBox PE_Frame PE_FrameLineEdit PE_FrameMenu PE_FrameDockWidget PE_FrameWindow	QStyleOptionFrame	State_Sunken	表示边框应该是凹陷的
PE_IndicatorToolBarHandle	QStyleOption	State_Horizontal	表示应该是水平的而不是垂直的
PE_IndicatorSpinPlus PE_IndicatorSpinMinus PE_IndicatorSpinUp PE_IndicatorSpinDown	QStyleOptionSpinBox	State_Sunken	表示按钮被按下
PE_PanelButtonCommand	QStyleOptionButton	State_Enabled	启用该按钮
		State_HasFocus	具有输入焦点
		State_Raised	按钮未被按下、不是切换按钮、不是平的, 则设置
		State_On	按钮是切换的且是打开的
		State_Sunken	表示按钮被按下

5、virtual void QStyle::**drawItemPixmap**(QPainter **painter*, const QRect &*rectangle*, int *alignment*,
const QPixmap &*pixmap*) const;

表示使用 painter 根据对齐方式 alignment, 在矩形 rectangle 中绘制像素图 pixmap

6、virtual void QStyle::**drawItemText**(QPainter **painter*, const QRect &*rectangle*, int *alignment*,
const QPalette &*palette*, bool *enabled*, const QString &*text*,
QPalette::ColorRole *textRole* = QPalette::NoRole) const;

表示使用 painter, 根据对齐方式 alignment, 调色板 palette, 在矩形 rectangle 中绘制文本 text。若指定了颜色角色 textRole, 则使用使用的颜色角色的调色板颜色绘制文本, 参数 enabled 用于指示是否启用该项。重新实现此函数时, enabled 参数应影响项目的绘制方式。

7、virtual void QStyle::**polish**(QWidget **widget*);
virtual void QStyle::**polish**(QApplication **application*);
virtual void QStyle::**polish**(QPalette &*palette*);

以上函数用于初始化部件的外观，会在部件创建完成之后，在第一次显示之前被调用，默认实现什么也不做。子类化 `QStyle` 时，可利用以上函数的调用时机，对部件的一些属性进行初始化。具体使用方法见下面的示例。

8、virtual void `QStyle::unpolish`(`QWidget *widget`);

virtual void `QStyle::unpolish`(`QApplication *application`);

以上函数与 `polish()` 函数相对应，需要注意的是，只有在部件被销毁时才会被调用。

9、virtual `QRect subControlRect`(`ComplexControl control`, const `QStyleOptionComplex *option`,

`SubControl subControl`, const `QWidget *widget = Q_NULLPTR`) const = 0;

返回复杂控件 `control` 的子控件 `subControl` 的矩形

10、virtual `QRect subElementRect`(`SubElement element`, const `QStyleOption *option`,

const `QWidget *widget = Q_NULLPTR`) const = 0;

返回由样式选项 `option` 所描述的控件的子元素 `element` 的矩形，重新实现该函数的示例见下一小节

三、子类化 `QStyle` 类的方法

- 1、子类化 `QStyle` 类自定义样式时，其父类通常应选择 `QStyle` 的子类而不是 `QStyle`，因为 `QStyle` 的子类实现了 `QStyle` 类中的虚函数，子类化其子类可以减少重新实现虚函数所需要的工作量，否则其工作量是比较庞大的。
- 2、`QStyle` 共有两个子类，分别为 `QCommonStyle`，及 `QCommonStyle` 的子类 `QProxyStyle`，在子类化时可根据情况进行选择，其中 `QCommonStyle` 类实现了部件的共同界面外观，因此该类实现的界面并不一定完整，而 `QProxyStyle` 类则实现了一个 `QStyle` (通常是默认的系统样式)，因此该类的实现比较完整。
- 3、自定义部件外观样式时，可以根据需要绘制的部件而选择 `drawControl()` 或 `drawComplexControl()` 或 `drawPrimitive()` 函数进行绘制，下面以示例对此方法进行说明

示例：子类化 `QCommonStyle` 实现自定义按钮样式

//m.h 文件的内容

```
#ifndef M_H
```

```
#define M_H
```

```
#include<QtWidgets>
```

```
class B:public QCommonStyle{    Q_OBJECT    //子类化 QCommonStyle 类
```

```
public:B() {}
```

```
//重新实现 drawControl() 函数以绘制控件的自定义外观(本示例用于绘制一个 QPushButton)
```

```
void drawControl(ControlElement e, const QStyleOption *op,
```

```
QPainter *pr, const QWidget *w = Q_NULLPTR) const{
```

```
    QPen pn(QColor(1, 111, 1));    //绿色
```

```
    QBrush bs(QColor(111, 111, 111));    //灰色
```

```
    QBrush bs1(QColor(111, 1, 1));    //红色
```

```
    QBrush bs2(QColor(222, 222, 222));    //白色
```

```
//参数 op，携带了绘制部件时的状态及其他一些信息。本示例绘制的是按钮，
```

```
//因此把 op 转换为 QStyleOptionButton 类型。
```

```
const QStyleOptionButton *pb=qstyleoption_cast<const QStyleOptionButton*>(op);
```

```
QRect r=pb->rect;    //获取设置的按钮的大小
```

```
//若鼠标进入按钮则使用红色填充其背景
```

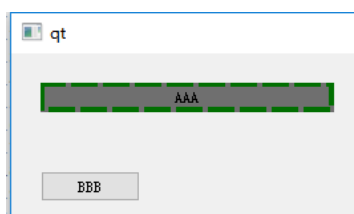
```

if(pb->state&QStyle::State_MouseOver)    {    pr->fillRect(r,bs1);    }
//若按钮处于凸起状态(通常为未选被按下状态),则使用灰色填充其背景。
//注意:凸起状态和鼠标进入按钮的状态,二者只能存在其一。
else if(pb->state&QStyle::State_Raised)    {    pr->fillRect(r,bs);    }
//若按钮被按下,则使用白色填充其背景
if(pb->state&QStyle::State_Sunken)    { pr->fillRect(r,bs2);    }
//当按钮具有焦点时,绘制按钮的焦点边框
if(pb->state&QStyle::State_HasFocus)    {
    pr->save();    pn.setWidth(4);    pn.setStyle(Qt::DashLine);    pr->setPen(pn);
    pr->drawRect(r.adjusted(1,1,-2,-2));    pr->restore();    }
//绘制按钮显示的文本。
pr->drawText(r,Qt::AlignCenter,pb->text);
}
void polish(QWidget *w) {
    //设置 Qt::WA_Hover 属性后,将使鼠标在进入或离开部件时产生绘制事件。
    w->setAttribute(Qt::WA_Hover,true);    }
void unpolish(QWidget *w) {    w->setAttribute(Qt::WA_Hover,false);    }
};
#endif // M_H

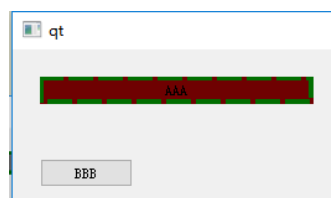
//m.cpp 文件的内容
#include "m.h"
int main(int argc, char *argv[]) {    QApplication aa(argc,argv);
    QWidget w;
    QPushButton *pb1=new QPushButton("AAA",&w); pb1->move(22,22);pb1->resize(221,22);
    QPushButton *pb2=new QPushButton("BBB",&w); pb2->move(22,88);
    pb1->setStyle(new B()); //按钮 pb1 使用自定义的样式
    w.resize(444,333);    w.show();    return aa.exec();    }

```

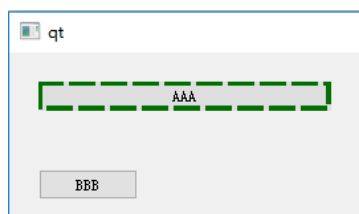
运行结果及说明



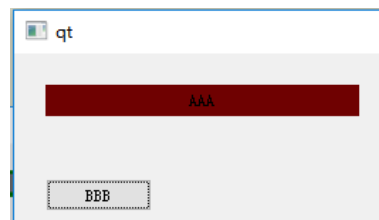
初始状态



鼠标进入按钮 AAA 之后,背景色变为红色



鼠标按下按钮 AAA 之后背景色变为白色



使用 Tab 把键盘焦点移至 BBB, 然后鼠标进入按钮 AAA, 此时 AAA 的背景变为红色, 其绿色的虚线焦点框将不被绘制, 若把焦点再次移至 AAA, 则绿色的虚线焦点框会被再次绘制

示例：子类化 QCommonStyle 实现自定义微调按钮样式

//m.h 文件的内容

```
#ifndef M_H
#define M_H
#include<QtWidgets>
class B:public QCommonStyle{    Q_OBJECT
public:B() {}
//本示例需重新实现 drawComplexControl() 函数，因为微调按钮是复杂控件元素
void drawComplexControl(ComplexControl c,const QStyleOptionComplex *op,
                        QPainter *pr, const QWidget *w = Q_NULLPTR) const{
    //参数 op，携带了绘制部件时的状态及其他一些信息。本示例绘制的是微调按钮，
    //因此把 op 转换为 QStyleOptionSpinBox 类型。
    const QStyleOptionSpinBox *pb=qstyleoption_cast<const QStyleOptionSpinBox*>(op);
    qDebug()<<pb->state;    //可输出 pb 以查看微调按钮的状态值
    //获取微调按钮向上/向下箭头和文本编辑区域所占据的矩形
    QRectF r1=subControlRect(c,op,QStyle::SC_SpinBoxUp,w);
    QRectF r2=subControlRect(c,op,QStyle::SC_SpinBoxDown,w);
    QRect r3=subControlRect(c,op,QStyle::SC_SpinBoxEditField,w);
    //创建一些画刷和画笔
    QBrush bs(QColor(111,111,111));    //灰色
    QBrush bs1(QColor(111,1,1));    //红色
    QPen pn(QColor(1,111,1));    //绿色
    //填充微调按钮箭头区域(含向上/向下箭头)的背景色为灰色
    pr->fillRect(r1.x(), r1.y(), r1.width(), r1.height()+r2.height(),bs);
    //创建向上箭头(一个指向上的三角形)的路径
    QPainterPath ph1;
    ph1.moveTo(r1.x()+r1.width()/2,r1.y());
    ph1.lineTo(r1.bottomLeft());
    ph1.lineTo(r1.bottomRight());
    ph1.closeSubpath();
    pr->drawPath(ph1);
    //创建向下箭头(一个指向下的三角形)的路径
    QPainterPath ph2;
    ph2.moveTo(r2.x(),r2.y());
    ph2.lineTo(r2.topRight());
    ph2.lineTo(r2.bottomLeft().x()+r2.width()/2,r1.height()+r2.height());
    ph2.closeSubpath();
    pr->drawPath(ph2);
    //绘制鼠标按下箭头时的外观
    if(pb->state&QStyle::State_Sunken) {
        //把当前鼠标的坐标转换为相对于微调按钮的坐标
        QPoint pt1=w->mapFromGlobal(QCursor::pos());
        pr->save();    //在绘制前保存画刷
        pr->setBrush(bs1);
        //若鼠标按下的是向上箭头则使用画刷 bs1(红色)填充其路径 ph1，否则填充 ph2。
        if(pt1.y()<r1.height()){    pr->drawPath(ph1); }
        else{    pr->drawPath(ph2); }
        //恢复画刷，若不恢复画刷，则前面设置的画刷 bs1 会继续作用于之后的绘制。
        pr->restore();    }
    //绘制当微调按钮获得焦点时的焦点边框
    if(pb->state&QStyle::State_HasFocus){
```

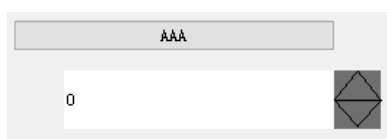
```

        pr->save();
        pr->setPen(pn);    //绿色画笔
        pr->drawRect(r3.adjusted(-1,-1,r1.width(),1));
        pr->restore();
    }
};
#endif // M_H

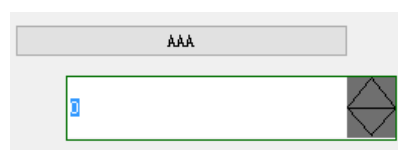
//m.cpp 文件的内容
#include "m.h"
int main(int argc, char *argv[]) {    QApplication aa(argc, argv);
    QWidget w;
    QPushButton *pbl=new QPushButton("AAA",&w); pbl->move(22,22);pbl->resize(221,22);
    QSpinBox *px=new QSpinBox(&w); px->move(55,55);    px->resize(221,44);
    px->setStyle(new B());    //微调按钮使用自定义样式
    w.resize(444,333);    w.show();    return aa.exec(); }

```

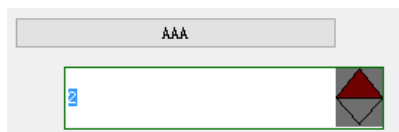
运行结果及说明



初始状态



微调按钮获得焦点后，绘制了绿色的矩形边框



当鼠标按下向上按钮时，向上按钮的背景变为红色，鼠标按下向下按钮原理类似，但要注意的是，当值为0时，按下向下按钮没有反应。

3、以上示例使用 `drawControl()`或 `drawComplexControl()`函数直接对整个部件的外观进行绘制，若使用 `drawPrimitive()`函数，则还可以绘制部件的某个元素，比如可绘制按钮的 `PE_PanelButtonCommand` 元素、`PE_FrameDefaultButton` 元素等，大至代码如下

```

void drawPrimitive(PrimitiveElement e, const QStyleOption *op, QPainter *pr,
                  const QWidget *w = Q_NULLPTR) const
{
    .....
    switch(e){.....
    case PE_PanelButtonCommand:
        //绘制元素的代码
    case PE_FrameDefaultButton:
        //绘制元素的代码
    .....
    }
    .....
}

```

13.5 QStyle 类的其他枚举及成员函数

一、QStyle::PixMetric 枚举及相关成员函数

1、virtual int QStyle::pixelMetric(PixelMetric *metric*, const QStyleOption **option* = Q_NULLPTR, const QWidget **widget* = Q_NULLPTR) const = 0;

返回 *metric* 所关联的像素度量值

2、QStyle::PixMetric 枚举见下表

QStyle::PixelMetric 枚举(无标志) --7			
作用：此枚举描述了各种像素度量(即与样式相关的大小)，与函数 pixelMetric()有关			
	成员	值	说明
① 按钮	QStyle::PM_ButtonMargin	0	按钮标签与边框之间的空白量
	QStyle::PM_ButtonDefaultIndicator	1	默认按钮边框的宽度
	QStyle::PM_ButtonShiftHorizontal	3	当按钮按下时，按钮的水平内容位移
	QStyle::PM_ButtonShiftVertical	4	当按钮按下时，按钮的垂直内容位移
	QStyle::PM_ButtonIconSize	?	按钮图标的大小
② 复选 按钮	QStyle::PM_IndicatorWidth	37	复选框指示器的宽度
	QStyle::PM_IndicatorHeight	38	复选框指示器的高度
	QStyle::PM_CheckBoxLabelSpacing	?	复选框指示器与标签之间的间距
③ 单选 按钮	QStyle::PM_ExclusiveIndicatorWidth	39	单选按钮指示器的宽度
	QStyle::PM_ExclusiveIndicatorHeight	40	单选按钮指示器的高度
	QStyle::PM_RadioButtonLabelSpacing	?	单选按钮指示器与标签之间的间距
④ 微调 按钮	QStyle::PM_SpinBoxFrameWidth	6	微调按钮的边框宽度，默认为 PM_DefaultFrameWidth
	QStyle::PM_SpinBoxSliderHeight	?	微调按钮滑块的高度
⑤ 滚动 条	QStyle::PM_MaximumDragDistance	8	拖动时鼠标与滚动条之间所允许的最大距离，超过此距离将使滑块跳回原始位置，值-1 将禁用此行为
	QStyle::PM_ScrollBarExtent	9	垂直滚动条的宽度和水平滚动条的高度。
	QStyle::PM_ScrollBarSliderMin	10	垂直滚动条滑块的最小高度和水平滚动条滑块的最小宽度。
⑥ 滑块	QStyle::PM_SliderThickness	11	滑块的厚度
	QStyle::PM_SliderControlThickness	12	滑块手柄的厚度
	QStyle::PM_SliderLength	13	滑块的长度
	QStyle::PM_SliderTickmarkOffset	14	刻度线和滑块之间的偏移量
	QStyle::PM_SliderSpaceAvailable	15	滑块移动的可用空间
⑦ 可停 靠窗 口	QStyle::PM_DockWidgetTitleBarButtonMargin	?	可停靠窗口标题栏按钮标签与边框之间的空白量
	QStyle::PM_DockWidgetSeparatorExtent	16	水平可停靠窗口分隔符的宽度和垂直可停靠窗口分隔符的高度
	QStyle::PM_DockWidgetHandleExtent	17	水平可停靠窗口手柄的宽度和垂直可停靠窗口手柄的高度
	QStyle::PM_DockWidgetFrameWidth	18	可停靠窗口的边框宽度
	QStyle::PM_DockWidgetTitleMargin	?	可停靠窗口标题的边距
⑧ 工具	QStyle::PM_ToolBarFrameWidth	?	工具栏周围边框的宽度
	QStyle::PM_ToolBarHandleExtent	?	水平工具栏中工具栏手柄的宽度和垂直工具栏中

栏			手柄的高度
	QStyle::PM_ToolBarItemMargin	?	工具栏边框和项之间的间距
	QStyle::PM_ToolBarItemSpacing	?	工具栏项之间的间距
	QStyle::PM_ToolBarSeparatorExtent	?	水平工具栏中工具栏分隔符的宽度和垂直工具栏中分隔符的高度
	QStyle::PM_ToolBarExtensionExtent	?	水平工具栏中工具栏扩展按钮的宽度和垂直工具栏中按钮的高度。
⑨ 选项 卡相 关	QStyle::PM_ToolBarIconSize	?	默认工具栏图标的大小
	QStyle::PM_TabBarTabOverlap	19	选项卡应重叠的像素数, 目前只在样式中使用, 不在 <code>QTabBar</code> 中使用。
	QStyle::PM_TabBarTabHSpace	20	添加到选项卡宽度的额外空间
	QStyle::PM_TabBarTabVSpace	21	添加到选项卡高度的额外空间
	QStyle::PM_TabBarBaseHeight	22	选项卡栏和选项卡页之间区域的高度
	QStyle::PM_TabBarBaseOverlap	23	选项卡栏与选项卡栏基础重叠的像素数
	QStyle::PM_TabBarScrollButtonWidth	?	
	QStyle::PM_TabBarTabShiftHorizontal	?	选择选项卡时的水平像素位移
	QStyle::PM_TabBarTabShiftVertical	?	选择选项卡时的垂直像素位移
	QStyle::PM_TabBar_ScrollButtonOverlap	?	选项卡栏中左右按钮之间的距度
	QStyle::PM_TabCloseIndicatorWidth	?	选项卡栏上关闭按钮的默认宽度
	QStyle::PM_TabCloseIndicatorHeight	?	选项卡栏上关闭按钮的默认高度
	QStyle::PM_TabBarIconSize	?	选项卡栏的默认图标大小
⑩ 菜单 栏	QStyle::PM_MenuBarPanelWidth	33	菜单栏的边框宽度, 默认为 <code>PM_DefaultFrameWidth</code>
	QStyle::PM_MenuBarItemSpacing	34	菜单栏项之间的间距
	QStyle::PM_MenuBarHMargin	36	菜单栏项与菜单栏左/右之间的间距
	QStyle::PM_MenuBarVMargin	35	菜单栏项与菜单栏顶/底之间的间距
⑪ 菜单	QStyle::PM_MenuPanelWidth	30	<code>QMenu</code> 的边框宽度(应用于所有侧面)
	QStyle::PM_MenuHMargin	28	<code>QMenu</code> 的附加边框(用于左/右)
	QStyle::PM_MenuVMargin	29	<code>QMenu</code> 的附加边框(用于顶/底)
	QStyle::PM_MenuScrollerHeight	27	<code>QMenu</code> 中滚动区域的高度
	QStyle::PM_MenuTearoffHeight	31	<code>QMenu</code> 中可分离区域的高度
	QStyle::PM_MenuDesktopFrameWidth	32	桌面上菜单边框的宽度
	QStyle::PM_SubMenuOverlap	?	子菜单与其父菜单之间的水平重叠部分
	QStyle::PM_MenuButtonIndicator	2	与部件高度成比例的菜单按钮指示器的宽度
⑫ 标题 栏	QStyle::PM_TitleBarHeight	26	标题栏的高度
	QStyle::PM_TitleBarButtonIconSize	?	标题栏上按钮图标的大小, qt5.8
	QStyle::PM_TitleBarButtonSize	?	标题栏上按钮的大小, qt5.8
⑬ MDI	QStyle::PM_MDIFrameWidth		已过时, 使用 <code>PM_MdiSubWindowFrameWidth</code> 代替
	QStyle::PM_MdiSubWindowFrameWidth	44	MDI 窗口边框的宽度。
	QStyle::PM_MDIMinimizedWidth		已过时, 改用 <code>PM_MdiSubWindowMinimizedWidth</code>
	QStyle::PM_MdiSubWindowMinimizedWidth	?	最小化 MDI 窗口的宽度
⑭ QLay out	QStyle::PM_LayoutLeftMargin	?	<code>QLayout</code> 的默认左边距
	QStyle::PM_LayoutTopMargin	?	<code>QLayout</code> 的默认上边距
	QStyle::PM_LayoutRightMargin	?	<code>QLayout</code> 的默认右边距
	QStyle::PM_LayoutBottomMargin	?	<code>QLayout</code> 的默认下边距
	QStyle::PM_LayoutHorizontalSpacing	?	<code>QLayout</code> 的默认水平间距
	QStyle::PM_LayoutVerticalSpacing	?	<code>QLayout</code> 的默认垂直间距
⑮	QStyle::PM_ComboBoxFrameWidth	7	给合框的边框宽度, 默认为 <code>PM_DefaultFrameWidth</code>

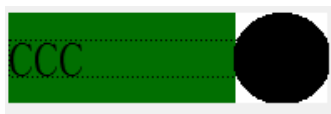
其他 控件	QStyle::PM_SizeGripSize	?	大小夹点的大小
	QStyle::PM_MessageBoxIconSize	?	消息框中标准图标的大小
	QStyle::PM_TextCursorWidth	?	行编辑器或文本编辑器光标宽度。
	QStyle::PM_ProgressBarChunkWidth	24	进度条指示器中块的宽度
	QStyle::PM_ToolTipLabelFrameWidth	?	工具提示标签的边框宽度
	QStyle::PM_SplitterWidth	25	分隔条(分离器)的宽度
⑩ 其他	QStyle::PM_SmallIconSize	?	默认小图标的大小
	QStyle::PM_LargeIconSize	?	默认大图标的大小
	QStyle::PM_FocusFrameHMargin	?	焦点边框的水平间距
	QStyle::PM_FocusFrameVMargin	?	焦点边框的垂直间距
	QStyle::PM_DefaultFrameWidth	5	默认边框宽度(通常为 2)
⑪ 模型/ 视图 相关	QStyle::PM_HeaderMarkSize	?	标头中排序指示器的大小
	QStyle::PM_HeaderGripMargin	?	标头中调整大小的夹点的大小
	QStyle::PM_HeaderMargin	?	排序指示器和文本之间的边距大小
	QStyle::PM_IconViewIconSize	?	图标视图中图标的默认大小
	QStyle::PM_ListViewIconSize	?	列表视图中图标的默认大小
	QStyle::PM_ScrollView_ScrollBarSpacing	?	使用 SH_ScrollView_FrameOnlyAroundContents 设置的边框和滚动条之间的距离
	QStyle::PM_ScrollView_ScrollBarOverlap	?	滚动条和滚动内容之间的重叠部分
	QStyle::PM_TreeViewIndentation	?	树视图中项的缩进量, qt5.4
	QStyle::PM_HeaderDefaultSectionSizeHorizontal	?	水平标头中段的默认大小, qt5.5
	QStyle::PM_HeaderDefaultSectionSizeVertical	?	垂直标头中段的默认大小, qt5.5
	QStyle::PM_CustomBase		自定义像素度量的基础值(其值为 0xf0000000), 自定义值必须大于此值

示例：自行绘制复选按钮(重新实现 pixelMetric() 和 subElementRect() 函数)

- 1、本示例比较综合，演示了怎样绘制如下图所示的复选按钮，并且演示了怎样重新实现 pixelMetric() 和 subElementRect() 函数，以及 QStyle::PixelMetric 枚举(PM_XXX) 和 QStyle::SubElement 枚举(SE_XXX) 的使用

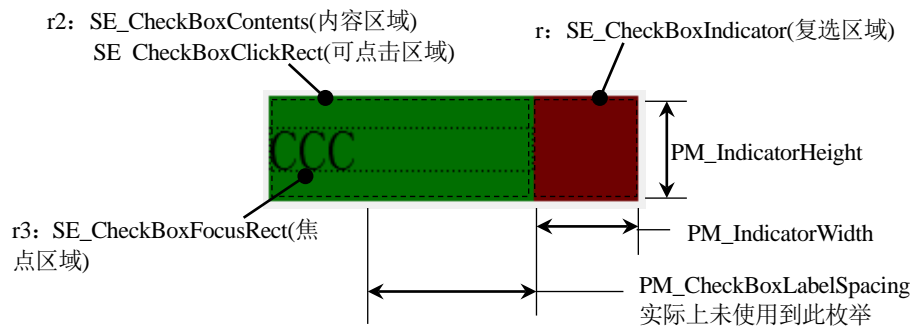


初始状态且未获得焦点时的情形
右侧部分为红色的复选区域，左侧为绿色背景
的文本区域



- 1、在左侧按下鼠标左键后，复选区域绘制一个黑色的椭圆表示该复选框被选中，
- 2、鼠标左键的点击只在复选按钮左侧的绿色背景区域有效，在复选区域按下鼠标左键没有反应。
- 3、当复选按钮获得焦点时，绘制复选按钮的焦点矩形(图中虚线)，其矩形的高度为字体的高度，长度为绿色背景区域的长度

2、下图为本示例各区域和各枚举代表的意义。注意：Qt 的内部实现其枚举所代表的意义与本示例可能会有一些不相同。



3、下面为示例代码(运行结果及说明见上文讲解)

//m.h 文件的内容

```
#ifndef M_H
```

```
#define M_H
```

```
#include<QtWidgets>
```

```
class B:public QProxyStyle{    Q_OBJECT
```

```
public:B() {}
```

//1、重新实现 pixelMetric() 函数，以自定义复选区域的宽度和高度

```
int pixelMetric(PixelMetric m, const QStyleOption *op=Q_NULLPTR,
                const QWidget *w = Q_NULLPTR) const {
    //qDebug()<<m;    //输出 m 的值，读者可以查看到复选框(本示例)使用了哪些 PM_开头的枚举
    switch(m) {
        //注意：复选区域的高度和宽度还受到 resize() 函数设置的复选框大小的影响，比如当 resize()
        //设置的高度小于此函数返回的高度时，则复选区域的高度为 resize() 函数的高度，为保持与
        //resize() 设置的高度一致，可使用 QStyleOptionButton::rect 成员变量的高度值
        //①、设置复选框复选区域的高度和宽度
        case PM_IndicatorHeight:return 54;
        case PM_IndicatorWidth:return 55;
        //②、复选框复选区域与文标签之间的间距
        case PM_CheckBoxLabelSpacing:return 88;
        //③、其他距离使用父类的实现
        default:return QProxyStyle::pixelMetric(m, op, w);
    }
}
```

//2、重新实现 subElementRect() 函数，以自定义复选按钮各区域的尺寸

```
QRect subElementRect(SubElement e, const QStyleOption *op,
                    const QWidget *w = Q_NULLPTR) const {
    //qDebug()<<e;    //可使用此函数查看 e 的值
    const QStyleOptionButton *pb=qstyleoption_cast<const QStyleOptionButton*>(op);
    QRect rl;
    //①、计算整个复选框的大小
    if(pb!=0)    rl=pb->rect;
    //②、计算 SE_CheckBoxIndicator(复选区域)，其宽度和高度使用 pixelMetric() 函数设置的值，
    //为使复选区域位于右侧，该矩形区域从复选按钮的右上角坐标计算(这样更便于计算)，
    //其宽度使用负值(即向左侧绘制矩形)。
    QRect r(rl.topRight().x(), 0, -pixelMetric(PM_IndicatorWidth, op, w),
            pixelMetric(PM_IndicatorHeight, op, w));
```

```

//③、计算 SE_CheckBoxContents(内容区域和可点击区域)
QRect r2(0,0,r1.width()+r1.height(),r1.height());
//④、计算 SE_CheckBoxFocusRect(焦点区域)，焦点区域的高度为设置的字体的高度。
QRect r3(0,0+(r2.height()-w->font().pointSize())/2,r2.width()-1,
                                                w->font().pointSize());

//⑤、返回各计算出来的各区域的矩形(比较简单)
switch(e) {
    case QStyle::SE_CheckBoxIndicator: {return r;}           //复选区域
    case QStyle::SE_CheckBoxContents: { return r2; }         //内容区域
    case QStyle::SE_CheckBoxFocusRect: {return r3;}          //焦点区域
    case QStyle::SE_CheckBoxClickRect: {return r2;}          //可点击区域
    default: {return QProxyStyle::subElementRect(e,op,w);}
} }

//3、重新实现 drawControl() 函数，以自定义复选按钮的外观
void drawControl(ControlElement e,const QStyleOption *op,
                 QPainter *pr, const QWidget *w = Q_NULLPTR) const {
    qDebug()<<e;           //输出 e 可看到复选框所使用的 CE_开头的枚举(控件元素)
    qDebug()<<op->state;    //还可查看部件的状态
//①、获取复选按钮复选区域的矩形
const QStyleOptionButton *pb=qstyleoption_cast<const QStyleOptionButton*>(op);
QRect r=subElementRect(QStyle::SE_CheckBoxIndicator,op,w);
//②、获取复选按钮除复选区域之外(即复选框的内容区域)的矩形
QRect r2=subElementRect(QStyle::SE_CheckBoxContents,op,w);
//③、获取复选按钮的焦点矩形
QRect r3=subElementRect(QStyle::SE_CheckBoxFocusRect,op,w);
//④、绘制复选按钮的初始外观
if(e==QStyle::CE_CheckBox) {
    pr->fillRect(r,QColor(111,1,1));           //使用红色填充复选区域
    pr->fillRect(r2,QColor(1,111,1));          //使用绿色填充内容区域
//⑤、绘制按下复选按钮时的外观
    if(op->state&QStyle::State_Sunken) {       //若复选按钮被按下
        pr->fillRect(r,QColor(255,255,255)); //首先使用白色画刷清除之前绘制的背景
        QBrush bs(QColor(0,0,0));
        pr->setBrush(bs);   pr->drawEllipse(r); //然后使用黑色画刷 bs 绘制一个椭圆
    }
}
//⑥、绘制复选按钮的焦点方框
if(pb->state&QStyle::State_HasFocus) { //若复选按钮获得了焦点
    //使用一个虚线画笔绘制一个无填充的矩形
    QPen pn(Qt::DotLine);   pr->setBrush(Qt::NoBrush);   pr->setPen(pn);
    pr->drawRect(r3);}
//⑦、绘制复选按钮的文本(垂直居中于 r2)，此步骤需最后绘制，否则可能会被绘制的其他图形覆盖。
pr->drawText(r2,Qt::AlignVCenter,pb->text);
} };
#endif // M_H

//m.cpp 文件的内容
#include "m.h"
int main(int argc, char *argv[]) {   QApplication aa(argc,argv);
    QWidget w;
    QPushButton *pb1=new QPushButton("AAA",&w); pb1->move(22,22);pb1->resize(221,22);
    QCheckBox *pc=new QCheckBox("CCC",&w); pc->move(99,55);

```



```

QFont f;    f.setPointSize(22);    pc->setFont(f);    //为复选按钮设置字体
pc->setStyle(new B());    //复选按钮使用自定义样式
w.resize(444, 333);    w.show();    return aa.exec(); }

```

二、QStyle::StandardPixmap 枚举及相关成员函数

- virtual QIcon **standardIcon**(StandardPixmap *standardIcon*, const QStyleOption **option* = Q_NULLPTR, const QWidget **widgit* = Q_NULLPTR) const = 0;

返回 standardIcon 指定的标准图标

- QStyle::StandardPixmap 枚举见下表

QStyle::StandardPixmap 枚举(无标志) --8			
作用：此枚举描述了可用的标准像素图，与函数 standardIcon()有关			
	成员	值	说明
1、标题栏	QStyle::SP_TitleBarMinButton	1	标题栏上的最小化按钮
	QStyle::SP_TitleBarMenuButton	0	标题栏上的菜单按钮
	QStyle::SP_TitleBarMaxButton	2	标题栏上的最大化按钮
	QStyle::SP_TitleBarCloseButton	3	标题栏上的关闭按钮
	QStyle::SP_TitleBarNormalButton	4	标题栏上的“正常(还原)”按钮
	QStyle::SP_TitleBarShadeButton	5	标题栏上的阴影按钮
	QStyle::SP_TitleBarUnshadeButton	6	标题栏上的取消阴影按钮
	QStyle::SP_TitleBarContextHelpButton	7	标题栏上的上下文帮助按钮
2、消息框	QStyle::SP_MessageBoxInformation	9	信息图标
	QStyle::SP_MessageBoxWarning	10	警告图标
	QStyle::SP_MessageBoxCritical	11	严重警告图标
	QStyle::SP_MessageBoxQuestion	12	询问图标
3、目录相关	QStyle::SP_DesktopIcon	13	桌面图标
	QStyle::SP_TrashIcon	14	垃圾图标
	QStyle::SP_ComputerIcon	15	我的电脑图标
	QStyle::SP_DriveFDIcon	16	软盘图标
	QStyle::SP_DriveHDIIcon	17	硬盘图标
	QStyle::SP_DriveCDIcon	18	CD 图标
	QStyle::SP_DriveDVDIcon	19	DVD 图标
	QStyle::SP_DriveNetIcon	20	网络图标
	QStyle::SP_DirHomeIcon	56	主目录图标
	QStyle::SP_DirOpenIcon	21	打开的目录图标
	QStyle::SP_DirClosedIcon	22	关闭的目录图标
	QStyle::SP_DirIcon	38	目录图标
	QStyle::SP_DirLinkIcon	23	目录链接的图标
	QStyle::SP_DirLinkOpenIcon	24	打开目录链接的图标
4、文件对话框	QStyle::SP_FileIcon	25	文件图标
	QStyle::SP_FileLinkIcon	26	文件链链的图标
	QStyle::SP_FileDialogStart	29	文件对话框的 Start 图标
	QStyle::SP_FileDialogEnd	30	文件对话框的 End 图标
	QStyle::SP_FileDialogToParent	31	文件对话框的父目录图标
	QStyle::SP_FileDialogNewFolder	32	文件对话框的创建新文件夹图标
	QStyle::SP_FileDialogDetailedView	33	文件对话框的详细视图图标

	QStyle::SP_FileDialogInfoView	34	文件对话框的文件信息图标
	QStyle::SP_FileDialogContentsView	35	文件对话框的内容视图图标
	QStyle::SP_FileDialogListView	36	文件对话框的列表视图图标
	QStyle::SP_FileDialogBack	37	文件对话框的 back 箭头
5、工具栏	QStyle::SP_ToolBarHorizontalExtensionButton	27	水平工具栏的扩展按钮
	QStyle::SP_ToolBarVerticalExtensionButton	28	垂直工具栏的扩展按钮
6、QDialogButtonBox	QStyle::SP_DialogOkButton	39	QDialogButtonBox 中标准 OK 按钮的图标
	QStyle::SP_DialogCancelButton	40	QDialogButtonBox 中标准 Cancel 按钮的图标
	QStyle::SP_DialogHelpButton	41	QDialogButtonBox 中标准 Help 按钮的图标
	QStyle::SP_DialogOpenButton	42	QDialogButtonBox 中标准 Open 按钮的图标
	QStyle::SP_DialogSaveButton	43	QDialogButtonBox 中标准 Save 按钮的图标
	QStyle::SP_DialogCloseButton	44	QDialogButtonBox 中标准 Close 按钮的图标
	QStyle::SP_DialogApplyButton	45	QDialogButtonBox 中标准 Apply 按钮的图标
	QStyle::SP_DialogResetButton	46	QDialogButtonBox 中标准 Reset 按钮的图标
	QStyle::SP_DialogDiscardButton	47	QDialogButtonBox 中标准 Discard 按钮的图标
	QStyle::SP_DialogYesButton	48	QDialogButtonBox 中标准 Yes 按钮的图标
	QStyle::SP_DialogNoButton	49	QDialogButtonBox 中标准 No 按钮的图标
7、箭头	QStyle::SP_ArrowUp	50	指向上方箭头的图标
	QStyle::SP_ArrowDown	51	指向下方箭头的图标
	QStyle::SP_ArrowLeft	52	指向左方箭头的图标
	QStyle::SP_ArrowRight	53	指向右方箭头的图标
	QStyle::SP_ArrowBack	54	若布局方向为 Qt::LeftToRight, 则等效于 SP_ArrowLeft, 否则等效于 SP_ArrowRight
	QStyle::SP_ArrowForward	55	若布局方向为 Qt::LeftToRight, 则等效于 SP_ArrowRight, 否则等效于 SP_ArrowLeft
8、媒体播放	QStyle::SP_MediaPlay	61	媒体开始播放图标
	QStyle::SP_MediaStop	62	媒体停止播放图标
	QStyle::SP_MediaPause	63	媒体暂停播放图标
	QStyle::SP_MediaSkipForward	64	媒体快进的图标
	QStyle::SP_MediaSkipBackward	65	媒体快退的图标
	QStyle::SP_MediaSeekForward	66	媒体向前查找的图标
	QStyle::SP_MediaSeekBackward	67	媒体向后查找的图标
	QStyle::SP_MediaVolume	68	媒体的音量控制图标
	QStyle::SP_MediaVolumeMuted	69	媒体的静音控制图标
9、其他	QStyle::SP_LineEditClearButton	70	QLineEdit 标准清除按钮的图标, qt5.2
	QStyle::SP_DockWidgetCloseButton	8	可停靠窗口上的关闭按钮
	QStyle::SP_CommandLink	57	用于指示 Vista 样式命令链接的图标
	QStyle::SP_VistaShield	58	用于指示 Vista 上的 UAC 提示的图标, 在其他平台上将返回一个空的像素图或图标
	QStyle::SP_BrowserReload	59	重新加载当前页的图标
	QStyle::SP_BrowserStop	60	停止加载页的图标
	QStyle::SP_CustomBase		自定义标准像素图的基础值(其值为 0xf0000000), 自定义值必须大于此值

示例：使用自定义的图标(重新实现 standardIcon() 函数)

//m.h 文件的内容

```

#ifndef M_H
#define M_H
#include<QtWidgets>

```

```
class B:public QProxyStyle{    Q_OBJECT
public:B() {}
    QIcon standardIcon(StandardPixmap ic, const QStyleOption *op = Q_NULLPTR,
                        const QWidget *w = Q_NULLPTR) const{

        qDebug()<<ic;
        //若是询问消息框, 则使用图像 li.png 作为其图标
        if(ic==QStyle::SP_MessageBoxQuestion) return QIcon("F:/li.png");
        return QProxyStyle::standardIcon(ic, op, w);
    };
#endif // M_H

//m.cpp 文件的内容
#include "m.h"
int main(int argc, char *argv[]){    QApplication aa(argc,argv);
    QMessageBox ms(QMessageBox::Question,"AAA","BBB");
    ms.setModal(0);    ms.setStyle(new B());
    ms.show();    return aa.exec(); }
```

运行结果及说明



三、QStyle::StyleHint 枚举及相关成员函数

- 1、virtual int styleHint(StyleHint hint, const QStyleOption *option = Q_NULLPTR, const QWidget *widget = Q_NULLPTR, QStyleHintReturn *returnData = Q_NULLPTR) const = 0
返回由 hint 指示的样式提示
- 2、QStyle::StyleHint 枚举比较多, 限于篇幅本文仅讲解怎样使用该枚举(以滚动条为例)以及怎样重新实现 styleHint()函数, 完整的 QStyle::StyleHint 枚举值可参阅帮助文档

QStyle::StyleHint 枚举(无标志) --9			
作用: 此枚举描述了可用的界面外观样式提示, 与函数 styleHint()有关			
	成员	值	说明
滚动条	QStyle::SH_ScrollBar_ContextMenu		滚动条是否具有上下文菜单
	QStyle::SH_ScrollBar_MiddleClickAbsolutePosition	2	若为 true, 则在滚动条上点击鼠标中键会使滑块跳转到该位置, 若为 false, 则忽略中键点击
	QStyle::SH_ScrollBar_LeftClickAbsolutePosition		若为 true, 则在滚动条上点击鼠标左键会使滑块跳转到该位置, 若为 false, 则左键单击适用于每个控件
	QStyle::SH_ScrollBar_ScrollWhenPointerLeavesControl	3	若为 true, 则单击滚动条的子控件(SubControl), 按住鼠标按钮并把鼠标移至于控件外部时, 滚动条将继续滚动; 若为 false, 则当鼠标离开子控件时, 滚动条会

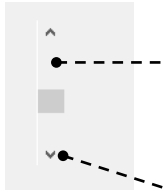
		停止滚动。
	QStyle::SH_ScrollBar_RollBetweenButtons	若为 true，则当单击滚动条按钮 (SC_ScrollBarAddLine or SC_ScrollBarSubLine) 并拖动到相反的按钮(滚动)时，将按下新按钮并释放旧按钮。若为 false，原来的按钮被释放，且无任何反应
	QStyle::SH_ScrollBar_Transient	样式是否支持临时滚动条。当需要滚动时出现临时滚动条，不需要时就消失

示例：使用样式提示(重新实现 styleHint() 函数)

```
//m.h 文件的内容
#ifndef M_H
#define M_H
#include<QtWidgets>
class B:public QProxyStyle{    Q_OBJECT
public:B() {}
    int styleHint(StyleHint h, const QStyleOption *op = Q_NULLPTR,
                  const QWidget *w = Q_NULLPTR, QStyleHintReturn *rd = Q_NULLPTR) const
    {
        qDebug()<<h;
        switch(h) { //以下设置的效果见示例后的图示讲解
            case QStyle::SH_ScrollBar_LeftClickAbsolutePosition: return true;
            case QStyle::SH_ScrollBar_MiddleClickAbsolutePosition: return true;
            case QStyle::SH_ScrollBar_ScrollWhenPointerLeavesControl: return false;
        }
        return QProxyStyle::styleHint(h, op, w);
    }
};
```

```
//m.cpp 文件的内容
#include "m.h"
int main(int argc, char *argv[]) {    QApplication aa(argc, argv);
    QWidget w;    QScrollBar *ps=new QScrollBar(&w);
    ps->resize(22,111);    ps->move(55,55);    ps->setStyle(new B());    //使用自定义样式
    w.resize(444,333);    w.show();    return aa.exec(); }
```

运行结果及说明



SH_ScrollBar_LeftClickAbsolutePosition 枚举
若该枚举返回 true，则鼠标左键点击图示位置，滑块将立即移至鼠标位置，
若该枚举返回 false，则点击一次鼠标，滑块只会移动一个步长的距离

SH_ScrollBar_ScrollWhenPointerLeavesControl 枚举
若该枚举返回 true，则点击滚动条的向上或向下箭头，并按住鼠标，然后把鼠标移至滚动条之外，则滑块会继续向上或向下移动，若该枚举为 false，则鼠标移除滚动条之外，滑块将停止移动

四、其他枚举及相关成员函数

1、virtual QSize **sizeFromContents**(ContentsType *type*, const QStyleOption **option*, const QSize &*contentsSize*,
const QWidget **widget* = Q_NULLPTR) const = 0

返回由 *type* 指定的元素的大小

QStyle::ContentsType 枚举(无标志) --10

作用：此枚举描述了可用的内容类型，以用于计算各部件内容的大小，与函数 `sizeFromContents()` 有关

成员	值	说明
QStyle::CT_CheckBox	1	复选框，如 QCheckBox
QStyle::CT_ComboBox	4	组合框，如 QComboBox
QStyle::CT_HeaderSection	19	标头部分，如 QHeader
QStyle::CT_LineEdit	14	行编辑器，如 QLineEdit
QStyle::CT_Menu	10	菜单，如 QMenu
QStyle::CT_MenuBar	9	菜单栏，如 QMenuBar
QStyle::CT_MenuBarItem	8	菜单栏项，如 QMenuBar 中的按钮
QStyle::CT_MenuItem	7	菜单项，如 QMenuItem
QStyle::CT_ProgressBar	6	进度条，如 QProgressBar
QStyle::CT_PushButton	0	按钮，如 QPushButton
QStyle::CT_RadioButton	2	单选按钮，如 QRadioButton
QStyle::CT_SizeGrip	16	大小夹点，如 QSizeGrip
QStyle::CT_Slider	12	滑块，如 QSlider
QStyle::CT_ScrollBar	13	滚动条，如 QScrollBar
QStyle::CT_SpinBox	15	旋转框(微调按钮)，如 QSpinBox
QStyle::CT_Splitter	5	分离器，如 QSplitter
QStyle::CT_TabBarTab	11	选项卡栏上的选项卡，如 QTabBar
QStyle::CT_TabWidget	17	选项卡部件，如 QTabWidget
QStyle::CT_ToolButton	3	工具按钮，如 QToolButton
QStyle::CT_GroupBox	20	组框，如 QGroupBox
QStyle::CT_ListItem	22	项目视图中的项目
QStyle::CT_MdiControls	21	菜单栏中的最小第、恢复和关闭按钮等。
QStyle::CT_CustomBase		自定义内容类型的基础值(其值为 0xf0000000)，自定义值必须大于此值

QStyle::RequestSoftwareInputPanel 枚举(无标志) --11

作用：此枚举描述了部件何时需要请求软件输入面板

成员	值	说明
QStyle::RSIP_OnMouseClickedAndAlreadyFocused	0	若用户单击部件，则请求输入面板，前提是部件已具有焦点。
QStyle::RSIP_OnMouseClicked	1	若用户单击部件，则请求输入面板

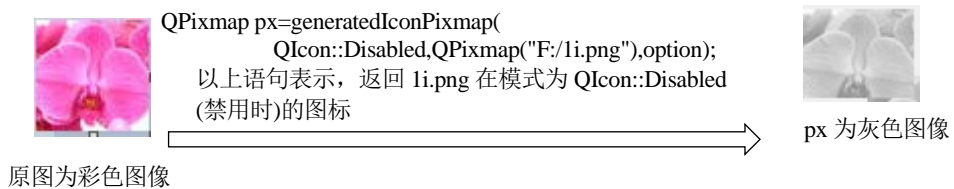
五、QStyle 类中的其他成员函数

1、以下虚函数主要用于获取部件的一些信息

1)、virtual QPixmap **generatedIconPixmap**(QIcon::Mode *iconMode*, const QPixmap &*pixmap*,

const QStyleOption *option) const = 0;

根据 iconMode 指定的模式返回 pixmap 的副本，样式为符合指定的 iconMode，并考虑 option 指定的调色板。原理见下图



2)、virtual SubControl **hitTestComplexControl**(ComplexControl *control*, const QStyleOptionComplex **option*, const QPoint &*position*, const QWidget **widget* = Q_NULLPTR) const = 0;

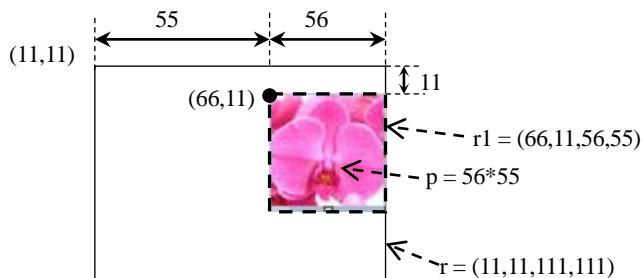
返回复杂按钮元素 control 中位置 position 处的子控件。参数 widget 是可选的。

3)、virtual QRect **itemPixmapRect**(const QRect &*rectangle*, int *alignment*, const QPixmap &*pixmap*) const; 返回在矩形 rectangle 内使用对齐方式 alignment 绘制像素图 pixmap 的区域。示例如下

QRect r (11,11,111,111);

QPixmap p("F:/li.png"); //其中 li.png 的大小为 56*55

QRect r1 = **itemPixmapRect**(r, Qt::AlignRight,p); //r1 = (66,11,56,55)，计算方法见下图



4)、virtual QRect **itemTextRect**(const QFontMetrics &*metrics*, const QRect &*rectangle*, int *alignment*, bool *enabled*, const QString &*text*) const;

返回在矩形 rectangle 内使用对齐方式 alignment 绘制像素图 pixmap 的区域。参数 enabled 指示是否启用关联项。如果给定的矩形大于绘制文本所需的面积，则返回的矩形将根据指定的对齐方式在矩形内偏移。例如，如果对齐为 QT：准直中心，则返回的矩形将在矩形内居中。如果给定的矩形小于所需区域，则返回的矩形将是足以呈现文本的最小矩形。

5)、virtual int **layoutSpacing**(QSizePolicy::ControlType *control1*, QSizePolicy::ControlType *control2*, Qt::Orientation *orientation*, const QStyleOption **option* = Q_NULLPTR, const QWidget **widget* = Q_NULLPTR) const = 0;

- 返回布局中控件 controls1 和控件 controls2 之间的间距。参数 orientation 指定控件是并排布置还是垂直堆叠，参数 widget 是可选的。

- 此函数由布局系统调用，只有当 `PM_LayoutHorizontalSpacing` 或 `PM_LayoutVerticalSpacing` 返回负值时才使用它。

6)、virtual `QPalette standardPalette()` const;

返回样式的标准调色板，注意：在支持系统颜色的平台上，不使用标准调色板，比如 window vista 和 mac 样式就不使用标准调色板，使用这些样式时不应使用 `QApplication::setPalette()` 设置调色板。

2、注意：以下函数不是虚函数也不是静态的

7)、int `combinedLayoutSpacing`(`QSizePolicy::ControlTypes controls1`,

`QSizePolicy::ControlTypes controls2`, `Qt::Orientation orientation`,

`QStyleOption *option = Q_NULLPTR`, `QWidget *widget = Q_NULLPTR`) const;

- 返回布局中控件 `controls1` 和控件 `controls2` 之间的间距。参数 `orientation` 指定控件是并排布置还是垂直堆叠，参数 `widget` 是可选的。
- `Controls1` 和 `Controls2` 是零或多个 `ControlType` 枚举类型的 OR 组合。
- 此函数由布局系统调用，只有当 `PM_LayoutHorizontalSpacing` 或 `PM_LayoutVerticalSpacing` 返回负值时才使用它。

8)、const `QStyle *proxy()` const; //返回此样式的当前代理样式。

3、以下静态函数主要用于从右到左的桌面

9)、static `Qt::Alignment visualAlignment`(`Qt::LayoutDirection direction`, `Qt::Alignment alignment`);

根据布局方向 `direction`(从左到右还是从右到右)，把不带有 `Qt::AlignAbsolute` 的对齐方式 `Qt::AlignLeft` 或 `Qt::AlignRight`(由 `alignment` 参数指定)转换为带有 `Qt::AlignAbsolute` 的对齐方式 `Qt::AlignLeft` 或 `Qt::AlignRight`

10)、static `QPoint visualPos`(`Qt::LayoutDirection direction`, const `QRect &boundingRectangle`,

const `QPoint &logicalPosition`);

返回根据指定方向 `direction` 转换为屏幕坐标的 `logicalPosition`, `boundingRectangle` 在转换时使用

11)、static `QRect visualRect`(`Qt::LayoutDirection direction`, const `QRect &boundingRectangle`,

const `QRect &logicalRectangle`);

返回根据指定方向 `direction` 转换为屏幕坐标的 `logicalRectangle`, `boundingRectangle` 在转换时使用。此函数是为了支持从右到左的桌面，通常用于重新实现的 `subControlRect()` 函数中

12)、static `QRect alignedRect`(`Qt::LayoutDirection direction`, `Qt::Alignment alignment`, const `QSize &size`,

const `QRect &rectangle`);

返回一个新的矩形，该矩形的大小为 `size`，并根据指定的对方方式 `alignment` 和方向 `direction` 与矩形 `rectangle` 对齐。

4、其他静态函数

13)、static int `sliderPositionFromValue`(int `min`, int `max`, int `logicalValue`, int `span`, bool `upsideDown = false`);

把逻辑值 `logicalValue` 转换为像素位置，`min` 参数被映射为 0，`max` 被映到 `span` 和其他值之间，若 `span` 小于 4096，则此函数可以处理整个整数范围而不会溢出。默认情况下，最大值 `max` 假定为水平项目的右侧，垂直项目的底部，参数 `upsideDown` 可反转此行为。

14)、static int **sliderValueFromPosition**(int *min*, int *max*, int *position*, int *span*, bool *upsideDown* = false);

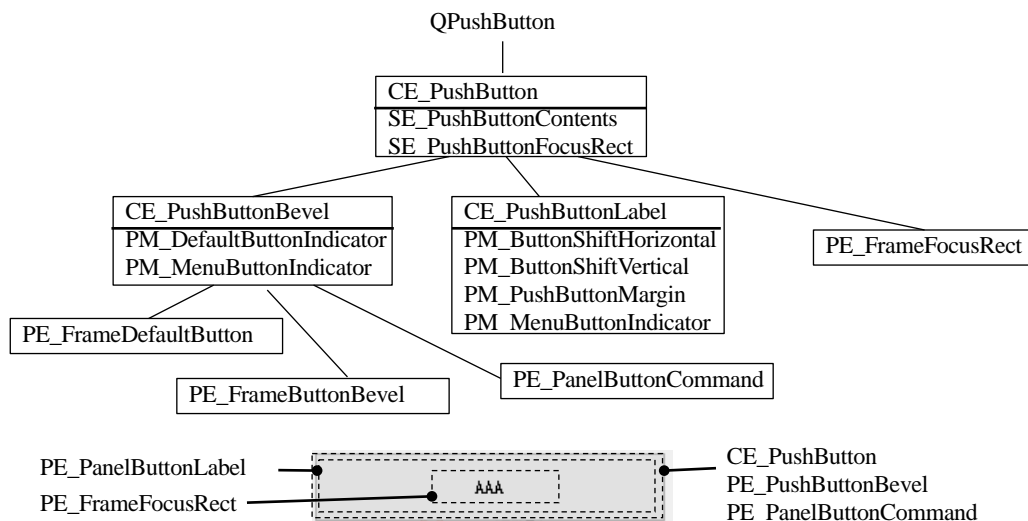
把像素位置 *position* 转换为逻辑值，其参数的意义与 `sliderPositionFromValue()`类似

13.6 QStyle 类中枚举的总结

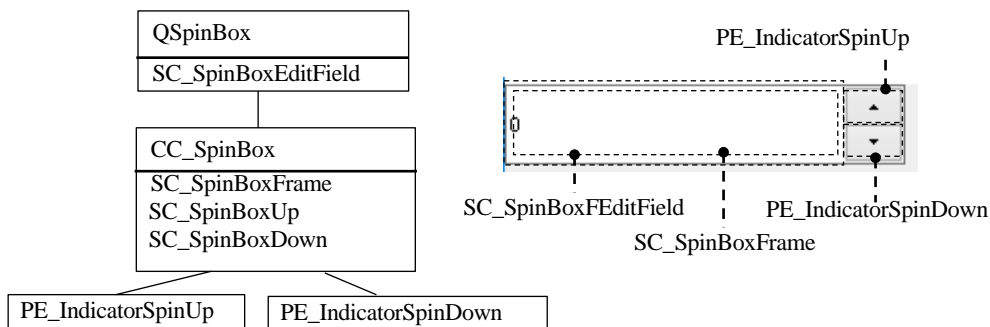
- 1、QStyle 类共有 11 个枚举，可以根据这些枚举列出各种部件界面外观的层次结构图，下面讲解一些部件的示例，其余部件读者可根据枚举值自行列出，也可在帮助文档中查看到类似的结构图(其位置为下图方框所示链接(QStyle 类帮助文档))

See also [QStyleOption](#), [QStylePainter](#), [Styles Example](#), [Styles and Style Aware Widgets](#), [QStyledItemDelegate](#), and [Styling](#).

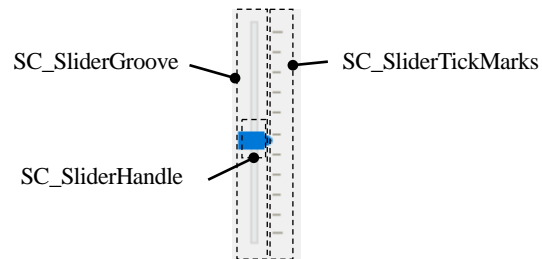
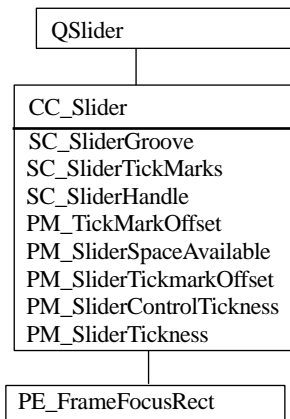
- 2、需要注意的是 CE_、CC_、PE_、SC_、SE_ 等前缀开始的枚举都是描述的一个矩形区域，比如枚举 SC_SpinBoxUp 描述的是微调按钮向上箭头的矩形区域，而不是描述的向上箭头，我们可在该枚举描述的矩形区域内绘制代表向上箭头作用的任意图形(比如可绘制“+”来代替向上箭头等)。其中 SC_(子控件)和 SE_(子元素)开始的枚举描述的矩形可通过重新实现 subControlRect()和 subElementRect()函数更改其位置和大小(具体的示例可参阅前文重新实现 subElementRect()函数的示例)。
- 2、按钮(QPushButton)的层次结构



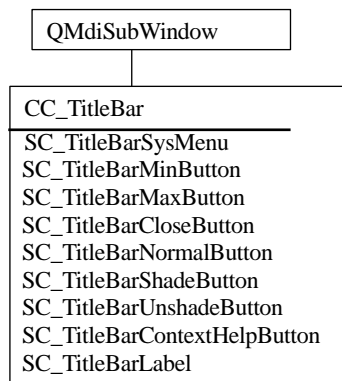
- 3、微调按钮(QSpinBox)的层次结构



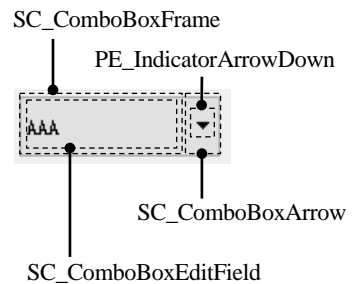
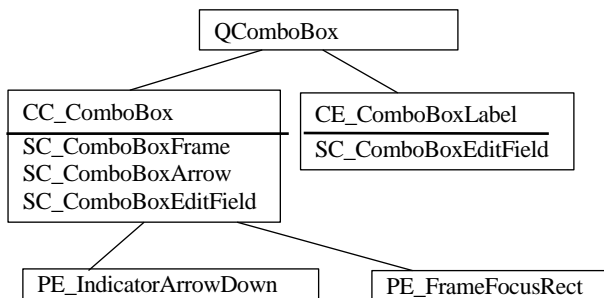
4、滑块(QSlider)的层次结构



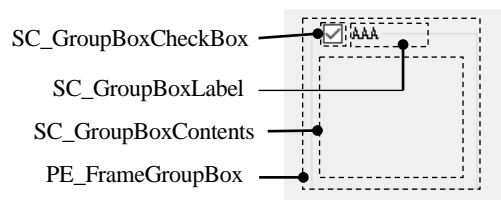
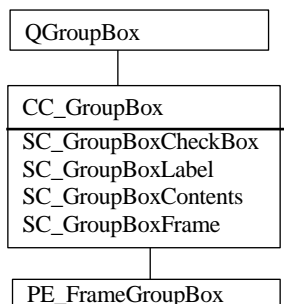
5、标题栏(QMdiSubWindow)



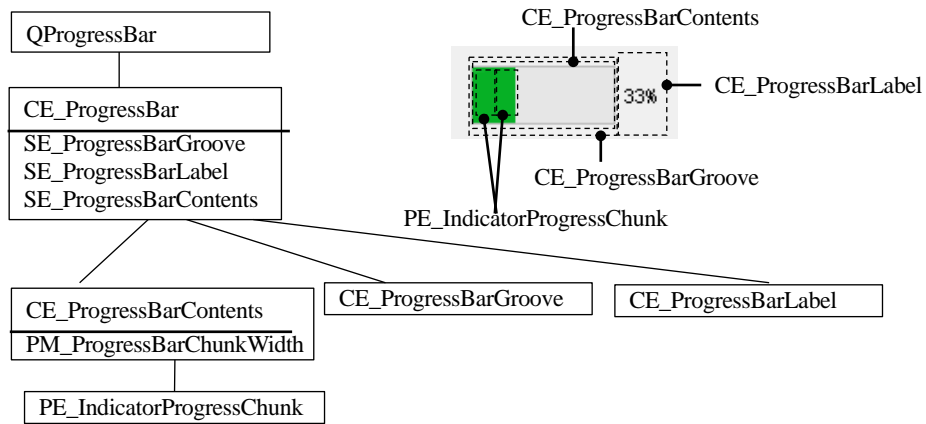
6、组合框(QComboBox)



7、组框(QGroupBox)



8、进度条(QProgressBar)



13.7 QStyleOption(样式选项)及其子类

- 1、QStyleOption 及其子类的继承关系见本章开头
- 2、QStyleOption 及其子类描述了绘制图形元素所需的一些信息，这些信息存储在公有的成员变量中，下面以表的形式列出这些类中的公有成员变量
- 3、QStyleOption 类

1、QStyleOption 类	
成员	说明
Qt::LayoutDirection direction	文本布局方向，默认为 Qt::LeftToRight
QFontMetrics fontMetrics	文本的字体度量标准，默认使用应用程序的默认字体
QPalette palette	调色板，默认使用应用程序的默认调色板
QRect rect	保存需要绘制的各种矩形区域，也就是说，对于需要绘制的不同的元素，该变量具有不同的含义，默认为空。比如，对于 QStyle::CE_PushButton，则 rect 是整个按钮的矩形区域，对于 QStyle::CE_PushButtonLabel，则只是按钮标签的矩形区域
QStyle::State state	需要绘制的元素的状态，默认为 QStyle::State_None
QObject * styleObject	正在设置的样式对象，内置支持 QWidget、QGraphicsObject 和 QQuickItem
int type	样式选项的选项类型，默认为 SO_Default，该值由 QStyleOption 及其子类内部使用。
int version	样式选项的版本，默认为 1。
void initFrom (const QWidget* w);	这是个便利函数，主要用于初始化 state、rect、palette 等成员变量，当然这些成员变量也可手动初始化。

- 4、QStyleOptionButton 类

2、QStyleOptionButton 类(按钮)	
成员	说明
ButtonFeatures features	按钮特征的按位或，枚举 ButtonFeature 见下表
QIcon icon	按钮的图标，默认为空
QSize iconSize	按钮图标的大小，默认为(-1, -1)，即无效大小
QString text	按钮的文本，默认为空

QStyleOptionButton::ButtonFeature 枚举

标志：QStyleOptionButton::ButtonFeatures

作用：描述按钮的类型

成员	值	说明
QStyleOptionButton::None	0x00	正常按钮
QStyleOptionButton::Flat	0x01	平面按钮
QStyleOptionButton::HasMenu	0x02	具有下拉菜单的按钮
QStyleOptionButton::DefaultButton	0x04	默认按钮
QStyleOptionButton::AutoDefaultButton	0x08	自动默认按钮
QStyleOptionButton::CommandLinkButton	0x10	windows vista 类型的命令链接按钮

5、QStyleOptionFocusRect 类

3、QStyleOptionFocusRect 类(焦点矩形)	
成员	说明
QColor backgroundColor	焦点矩形的背景色，默认为 RGB 值(0, 0, 0)的无效颜色。

6、QStyleOptionFrame 类

4、QStyleOptionFrame 类(边框)	
边框的原理详见 QFrame 类的讲解	
成员	说明
FrameFeatures features	边框特征的按位或，枚举 ButtonFeature 见下表
QFrame::Shape frameShape	边框的形状。
int lineWidth	边框的线宽。默认为 0
int midLineWidth	边框的中线宽，默认为 0

QStyleOptionFrame::FrameFeature 枚举
标志：QStyleOptionFrame::FrameFeatures

作用：描述边框的类型

成员	值	说明
QStyleOptionFrame::None	0x00	正常边框
QStyleOptionFrame::Flat	0x01	平面边框
QStyleOptionFrame::Rounded	0x02	圆角边框

7、QStyleOptionProgressBar 类

5、QStyleOptionProgressBar 类(进度条)	
成员	说明
bool bottomToTop	垂直进度条的文本是否从下到上读取，默认为 false
bool invertedAppearance	是否反转进度条的外观，默认为 false
int maximum	进度条的最大值，默认为 0
int minimum	进度条的最小值，默认为 0
int progress	进度条的当前进度
QString text	进度条的文本，默认为空字符串
Qt::Alignment textAlignment	进度条文本的对齐方式，默认为 Qt::AlignLeft
bool textVisible	进度条的文本是否可见。默认为 false

8、QStyleOptionRubberBand 类

6、QStyleOptionRubberBand 类(橡皮筋)	
成员	说明
bool opaque	橡皮筋是否不透明，默认为 true
QRubberBand::Shape shape	橡皮筋的形状，默认为 QRubberBand::Line

9、QStyleOptionTab 类

7、QStyleOptionTab 类(选项卡)	
成员	说明
CornerWidgets cornerWidgets	选项卡栏的角落部件，默认为 NoCornerWidgets，枚举见后表
bool documentMode	选项卡栏是否处于文档模式，默认为 false
QIcon icon	选项卡的图标，默认为空。
QSize iconSize	图标的大小，默认为(-1, -1)，即无效大小
QSize leftButtonSize	选项卡上左侧部件的大小，默认为(-1, -1)，即无效大小
TabPosition position	选项卡栏中选项卡的位置，默认为 Beginning，枚举见后表
QSize rightButtonSize	选项卡上右侧部件的大小，默认为(-1, -1)，即无效大小
int row	选项卡当前所在的行，目前只能是 0
SelectedPosition selectedPosition	所选选项卡相对于此选项卡的位置。默认值是 NotAdjacent，即该选项卡与选定的选项卡不相邻，也不是所选的选项卡。枚举见后表
QTabBar::Shape shape	选项卡的形状，默认为 QTabBar::RoundedNorth
QString text	选项卡的文本，默认为空

QStyleOptionTab::CornerWidget 枚举
标志：QStyleOptionTab::CornerWidgets

作用：描述选项卡的角落部件

成员	值	说明
QStyleOptionTab::NoCornerWidgets	0x00	无角落部件
QStyleOptionTab::LeftCornerWidgets	0x01	左角落部件
QStyleOptionTab::RightCornerWidgets	0x02	右角落部件

QStyleOptionTab::SelectedPosition 枚举(无标志)

作用：描述所选选项卡的位置

成员	值	说明
QStyleOptionTab::NotAdjacent	0	该选项卡不与所选的选项卡相邻
QStyleOptionTab::NextIsSelected	1	选择下一个选项卡(通常是右侧的选项卡)
QStyleOptionTab::PreviousIsSelected	2	选择上一个选项卡(通常是左侧的选项卡)

QStyleOptionTab::TabPosition 枚举(无标志)

作用：描述选项卡的位置

成员	值	说明
QStyleOptionTab::Beginning	0	该选项卡是选项卡栏中的第一个选项卡
QStyleOptionTab::Middle	1	该选项卡即不是第一个也不是最后一个选项卡
QStyleOptionTab::End	2	该选项卡是选项卡栏中的最后一个选项卡
QStyleOptionTab::OnlyOneTab	3	该选项卡是选项卡栏中的第一个和最后一个选项卡

10、QStyleOptionTabBarBase 类

8、QStyleOptionTabBarBase 类	
成员	说明
bool documentMode	选项卡栏是否处于文档模式，默认为 false
QRect selectedTabRect	所选选项卡的矩形，默认为空矩形
QTabBar::Shape shape	选项卡的形状，默认为 QTabBar::RoundedNorth
QRect tabBarRect	所有选项卡的矩形，默认为空矩形

11、QStyleOptionTabWidgetFrame 类

9、QStyleOptionTabWidgetFrame 类	
成员	说明
QSize leftCornerWidgetSize	选项卡上左角落部件的大小，默认为(-1, -1)，即无效大小
int lineWidth	边框的线宽。默认为 0
int midLineWidth	边框的中线宽，默认为 0
QSize rightCornerWidgetSize	选项卡上右角落部件的大小，默认为(-1, -1)，即无效大小
QRect selectedTabRect	所选选项卡的矩形，默认为空矩形
QTabBar::Shape shape	选项卡的形状，默认为 QTabBar::RoundedNorth
QRect tabBarRect	所有选项卡的矩形，默认为空矩形
QSize tabBarSize	选项卡栏的大小，默认为(-1, -1)，即无效大小

12、QStyleOptionDockWidget 类

10、QStyleOptionDockWidget 类(可停靠窗口)	
成员	说明
bool closable	是否可关闭可停靠窗口，默认为 true
bool floatable	可停靠窗口是否可浮动，默认为 true
bool movable	可停靠窗口是否可移动，默认为 false
QString title	可停靠窗口的标题，默认为空

13、QStyleOptionToolBar 类

11、QStyleOptionToolBar 类(工具栏)	
成员	说明
ToolBarFeatures features	工具栏是否可移动，默认为 None，枚举见后表
int lineWidth	工具栏的线宽。默认为 0，详见 QFrame 类的讲解
int midLineWidth	工具栏的中线宽。默认为 0
ToolBarPosition positionOfLine	工具栏行的位置，默认为 OnlyOne，枚举见后表
ToolBarPosition positionWithinLine	工具栏在一行中的位置，默认为 OnlyOne，枚举见后表
Qt::ToolBarArea toolBarArea	工具栏的位置，默认为 Qt::TopToolBarArea

QStyleOptionToolBar::ToolBarFeature 枚举

标志：QStyleOptionToolBar::ToolBarFeatures

作用：描述工具栏是否可移动

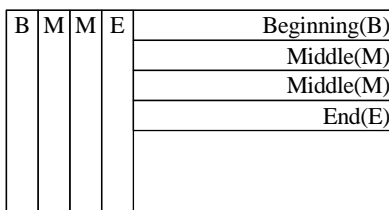
成员	值	说明
QStyleOptionToolBar::None	0x0	无法移动。
QStyleOptionToolBar::Movable	0x1	可移动。

QStyleOptionToolBar::ToolBarPosition 枚举(无标志)

作用：描述工具栏行及工具栏在行内的位置，该枚举的原理见下面的图示

成员	值	说明
QStyleOptionToolBar::Beginning	0	工具栏位于行的开头。
QStyleOptionToolBar::Middle	1	工具栏位于行的中间。

QStyleOptionToolBar::End	2	工具栏位于行的末尾。
QStyleOptionToolBar::OnlyOne	3	只有一个工具栏或行。



Beginning	Middle	Middle	End
-----------	--------	--------	-----

QStyleOptionToolBar::positionOfLine

QStyleOptionToolBar::positionWithinLine

14、QStyleOptionToolBox 类

12、QStyleOptionToolBox 类(工具箱)	
成员	说明
QIcon icon	工具箱选项卡的图标，默认为空
SelectedPosition selectedPosition	所选选项卡相对于此选项卡的位置，默认为 NotAdjacent
QString text	工具箱选项卡的文本，默认为空

QStyleOptionToolBox::SelectedPosition 枚举(无标志)

作用：描述所选选项卡的位置

成员	值	说明
QStyleOptionToolBox::NotAdjacent	0	该选项卡不与所选的选项卡相邻
QStyleOptionToolBox::NextIsSelected	1	选择下一个选项卡(通常是右侧的选项卡)
QStyleOptionToolBox::PreviousIsSelected	2	选择上一个选项卡(通常是左侧的选项卡)

15、QStyleOptionMenuItem 类

13、QStyleOptionMenuItem 类(菜单项)	
成员	说明
CheckType checkType	菜单项的复选标记类型，默认为 NotCheckable，枚举见后表
bool checked	菜单项是否被选中(checked)，默认为 false
QFont font	菜单项文本的字体
QIcon icon	菜单项的图标，默认为空
int maxIconWidth	菜单项图标的最大宽度，默认为 0，无论菜单是否有图标，必须设置此变量
bool menuHasCheckableItems	整个菜单项是否可被选中(checkable)，默认为 true
MenuItemType menuItem	菜单项的类型，默认为 Normal，枚举见后表
QRect menuRect	整个菜单的矩形，默认为空
int tabWidth	菜单项选项卡的宽度，选项卡宽度是指的菜单项文本与快捷键之间的距离，默认为 0。
QString text	菜单项的文本，默认为空

QStyleOptionMenuItem::CheckType 枚举(无标志)

作用：描述是否应绘制复选标记		
成员	值	说明
QStyleOptionMenuItem::NotCheckable	0	无复选标记
QStyleOptionMenuItem::Exclusive	1	独占复选标记(比如单选按钮)
QStyleOptionMenuItem::NonExclusive	2	非独占复选标记(比如复选按钮)

QStyleOptionMenuItem::MenuItemType 枚举(无标志)

作用：描述菜单项的类型

成员	值	说明
QStyleOptionMenuItem::Normal	0	普通菜单项
QStyleOptionMenuItem::DefaultItem	1	由 QMenu::defaultAction()返回的默认动作的菜单项
QStyleOptionMenuItem::Separator	2	菜单的分隔符
QStyleOptionMenuItem::SubMenu	3	菜单项的子菜单
QStyleOptionMenuItem::Scroller	4	弹出菜单的滚动条(目前仅适用于 macOS)
QStyleOptionMenuItem::TearOff	5	菜单的可分离手柄
QStyleOptionMenuItem::Margin	6	菜单的边距
QStyleOptionMenuItem::EmptyArea	7	菜单的空白区域

16、QStyleOptionHeader 类

14、QStyleOptionHeader 类(标头)

成员	说明
QIcon icon	标头的图标，默认为空
Qt::Alignment iconAlignment	标头图标的对齐方式，默认为 Qt::AlignLeft
Qt::Orientation orientation	标头的方向，默认为 Qt::Horizontal
SectionPosition position	标头的段相对于其他段的位置，默认为 Beginning，枚举见后表，其原理见 QStyleOptionToolBar::ToolBarPosition 枚举
int section	正在绘制标头的哪一个段，默认为 0
selectedPosition selectedPosition	所选段相对于此段的位置。默认值是 NotAdjacent，枚举见后表
SortIndicator sortIndicator	排序指示符的方向，枚举见后表
QString text	标头的文本，默认为空
Qt::Alignment textAlignment	标头文本的对齐方式，默认为 Qt::AlignLeft

QStyleOptionHeader::SectionPosition 枚举(无标志)

作用：描述该段与其他段的关系，其原理见 QStyleOptionToolBar::ToolBarPosition 枚举

成员	值	说明
QStyleOptionHeader::Beginning	0	段位于标头的开头。
QStyleOptionHeader::Middle	1	段位于标头的中间。
QStyleOptionHeader::End	2	段位于标头的末尾。
QStyleOptionHeader::OnlyOne	3	只有一个段。

QStyleOptionHeader::SelectedPosition 枚举(无标志)

作用：描述所选段与该段的位置

成员	值	说明
QStyleOptionHeader::NotAdjacent	0	该段不与所选的段相邻
QStyleOptionHeader::NextIsSelected	1	选择下一段
QStyleOptionHeader::PreviousIsSelected	2	选择上一段

QStyleOptionHeader::NextAndPreviousAreSelected	3	选择下一段和上一段
--	---	-----------

QStyleOptionHeader::SortIndicator 枚举(无标志)

作用：描述排序指示符的方向

成员	值	说明
QStyleOptionHeader::None	0	无排序指示符
QStyleOptionHeader::SortUp	1	绘制一个向上指示符
QStyleOptionHeader::SortDown	2	绘制一个向下指示符

17、QStyleOptionViewItem 类

15、QStyleOptionViewItem 类(视图)

成员	说明
QBrush backgroundBrush	视图项的背景画刷
Qt::CheckState checkState	视图项是否可被选中
Qt::Alignment decorationAlignment	视图项装饰的对齐方式，默认为 Qt::AlignLeft
Position decorationPosition	视图项装饰的位置，默认为 Left，枚举见后表
QSize decorationSize	视图项装饰的大小，默认为(-1,-1)，即无效大小
Qt::Alignment displayAlignment	项显示值的对齐方式，默认为 Qt::AlignLeft
ViewItemFeatures features	项特征的按位或，枚举见后表
QFont font	项的字体
QIcon icon	项的图标
QModelIndex index	模型索引
bool showDecorationSelected	是否应突出显示被选择项上的装饰
QString text	视图项的文本
Qt::TextElideMode textElideMode	文本太长时添加省略号的方式，默认为 Qt::ElideMiddle
ViewItemPosition viewItemPosition	该视图项相对于其他项的位置，枚举见后表

QStyleOptionViewItem::Position 枚举(无标志)

作用：描述项目装饰的位置

成员	值	说明
QStyleOptionViewItem::Left	0	在文本左侧
QStyleOptionViewItem::Right	1	文本右侧
QStyleOptionViewItem::Top	2	文本上方
QStyleOptionViewItem::Bottom	3	文本下方

QStyleOptionViewItem::ViewItemFeature 枚举

标志：QStyleOptionViewItem::ViewItemFeatures

作用：描述项的特征

成员	值	说明
QStyleOptionViewItem::None	0x00	正常项
QStyleOptionViewItem::WrapText	0x01	包含文本
QStyleOptionViewItem::Alternate	0x02	使用 alternateBase 渲染项背景(即交替背景色)
QStyleOptionViewItem::HasCheckIndicator	0x04	项具有可选中状态指示符
QStyleOptionViewItem::HasDisplay	0x08	项具有 display 角色
QStyleOptionViewItem::HasDecoration	0x10	项具有 decoration(装饰)角色

QStyleOptionViewItem::ViewItemPosition 枚举(无标志)		
作用：描述项的位置，其原理见 QStyleOptionToolBar::ToolBarPosition 枚举		
成员	值	说明
QStyleOptionViewItem::Invalid	0	未知，应被忽略
QStyleOptionViewItem::Beginning	1	项位于行的开头
QStyleOptionViewItem::Middle	2	项位于行的中间
QStyleOptionViewItem::End	3	项位于行的末尾
QStyleOptionViewItem::OnlyOne	4	项是唯一的项

18、QStyleOptionComplex 类

16、QStyleOptionComplex 类(复杂制件的父类)	
说明：该类不单独使用，而是作为其他类的父类	
成员	说明
QStyle::SubControls activeSubControls	保存激活的子控件的按位或
QStyle::SubControls subControls	保存被绘制的子控件的按位或

19、QStyleOptionComboBox 类

17、QStyleOptionComboBox 类(组合框)	
成员	说明
QIcon icon	组合框当前项的图标，默认为空
QString currentText	组合框当前项的文本，默认为空
bool editable	组合框是否可编辑，默认为 false
bool frame	组合框是否具有边框，默认为 true
QSize iconSize	组合框当前项图标的大小，默认为(-1, -1)，即无效大小
QRect popupRect	组合框的弹出矩形，默认为空，此变量目前未被使用。

20、QStyleOptionSpinBox 类

18、QStyleOptionSpinBox 类(微调按钮或旋转框)	
成员	说明
QAbstractSpinBox::ButtonSymbols buttonSymbols	旋转框按钮符号的类型，默认为 QAbstractSpinBox::UpDownArrows(即经典箭头)
bool frame	旋转框是否具有边框，默认为 false
QAbstractSpinBox::StepEnabled stepEnabled	启用了旋转框的哪些按钮(比如，向上还是向下箭头)，默认为 QAbstractSpinBox::StepNone

21、QStyleOptionSlider 类

19、QStyleOptionSlider 类(适用于滑块和滚动条)	
成员	说明
bool dialWrapping	表盘是否应换行，默认为 false
int maximum	滑块的最大值，默认为 0
int minimum	滑块的最小值，默认为 0
qreal notchTarget	凹槽之间的像素数，默认为 0.0

Qt::Orientation orientation	滑块的方向，默认为 Qt::Horizontal
int pageStep	滑块的页面步长，默认为 0
int singleStep	滑块的单个步长，默认为 0
int sliderPosition	滑块手柄的位置，默认为 0。此值与 QAbstractSlider::tracking 有关
int sliderValue	滑块的值，默认为 0。此值与 QAbstractSlider::tracking 有关
int tickInterval	刻度线之间的间隔，默认为 0
QSlider::TickPosition tickPosition	滑块刻度线的位置，默认为 QSlider::NoTicks
bool upsideDown	滑块增加的方向，默认情况下，滑块向上或向右移动时会增加(默认值，为 false)，若该值为 true，则滑块向下或向左移动时增加。

22、QStyleOptionSizeGrip 类

20、QStyleOptionSizeGrip 类(大小夹点)	
成员	说明
Qt::Corner corner	大小夹点所在的角落

23、QStyleOptionTitleBar 类

21、QStyleOptionTitleBar 类(标题栏)	
成员	说明
QIcon icon	标题栏的图标，默认为空
QString text	标题栏的文本，默认为空
Qt::WindowFlags titleBarFlags	标题栏的窗口标志，默认为 Qt::Widget
int titleBarState	标题栏的状态，默认为 0。另见 QWidget::windowState()函数。

24、QStyleOptionGroupBox 类

22、QStyleOptionGroupBox 类(组框)	
成员	说明
QStyleOptionFrame::FrameFeatures features	组框的边框特征，枚举见 QStyleOptionFrame 类
int lineWidth	边框的线宽，此变量的值始终为 1
int midLineWidth	边框的中线宽，此变量的值始终为 0
QString text	组框的文本，默认为空
Qt::Alignment textAlignment	组框标题的对齐方式，默认为 Qt::AlignLeft
QColor textColor	组框标题的颜色，默认为 RGB 值(0,0,0)的无效颜色。

25、QStyleOptionToolButton 类

23、QStyleOptionToolButton 类(工具按钮)	
成员	说明
Qt::ArrowType arrowType	工具按钮的箭头方向，默认为 Qt::DownArrow
ToolButtonFeatures features	工具按钮特征的按位或，默认为 None，枚举见后表
QFont font	使用文本的字体
QIcon icon	工具按钮的图标，默认为空
QSize iconSize	工具按钮的图标大小，默认为(-1, -1)，即无效大小
QPoint pos	工具按钮的位置，默认为(0,0)

QString text	工具按钮的文本，默认为空
Qt::ToolButtonStyle toolButtonStyle	工具按钮的样式(即怎样显示文本和图标)，默认为 Qt::ToolButtonIconOnly

QStyleOptionToolButton::ToolButtonFeature 枚举

标志: QStyleOptionToolButton::ToolButtonFeatures

作用：描述工具按钮的特征

成员	值	说明
QStyleOptionToolButton::None	0x00	普通工具按钮
QStyleOptionToolButton::Arrow	0x01	工具按钮是一个箭头
QStyleOptionToolButton::Menu	0x04	工具按钮有一个菜单
QStyleOptionToolButton::PopupDelay	0x08	显示菜单的延迟
QStyleOptionToolButton::HasMenu	0x10	该按钮是一个弹出菜单
QStyleOptionToolButton::MenuButtonPopup	Menu	该按钮应显示一个箭头，以表示菜单存在

13.8 样式表

一、样式表基础

- 1、Qt 样式表是另外一种自定义部件外观的机制。Qt 样式表的术语、语法几乎与层叠样式表 CSS(Cascading Style Sheets)相同，若已熟悉 CSS，则可快速阅读有关样式表的章节。
- 2、使用样式表与调色板(QPalette)相比，样式表更强大，因为使用调色板设置的外观，并不能保证在所有的样式中都可正常的工作，但样式表就不受这样的限制。而且使用样式表可以更方便的设置界面的外观，而不用去子类化 QStyle 类，
- 3、注意：子类化 QStyle 类的样式目前不支持样式表。这个问题将在以后解决。
- 4、可使用 QApplication::setStyleSheet()函数设置整个应用程序的样式表，使用 QWidget::setStyleSheet()设置部件及其子部件的样式表，这两个函数的原型如下：

```
void QWidget::setStyleSheet(const QString &sheet);
```

```
void QApplication::setStyleSheet(const QString &sheet);
```

示例：

```
QApplication aa(argc,argv);
```

```
aa.setStyleSheet("QPushButton{background-color:red}"); //设置按钮的背景色
```

若直接在部件上设置样式表，可忽略选择器及大括号，比如

```
QPushButton *pb1=new QPushButton ("BBB",&w);
```

```
//设置按钮的背景色，省略了选择器(即 QPushButton)和大括号
```

```
pb1->setStyleSheet("background-color:yellow");
```

- 5、样式表的使用示例：

示例：样式表的简单使用

```
#include<QtWidgets>
```

```
int main(int argc, char *argv[]) {
```

```
    QApplication aa(argc, argv);
```

```
    QWidget w;
```

```
    QPushButton *pb1=new QPushButton("AAA",&w); pb1->move(22,22);
```

```
    QPushButton *pb2=new QPushButton("BBB",&w); pb2->move(99,22);
```

```
    QCheckBox *pc=new QCheckBox("CCC",&w); pc->move(22,55);
```

```
    QCheckBox *pc1=new QCheckBox("DDD",&w); pc1->move(99,55);
```

```
//设置复选按钮 pc1 的背景色为黄色，复选按钮 pc 不受影响，注意：函数中的字符串为样式表的语法
```

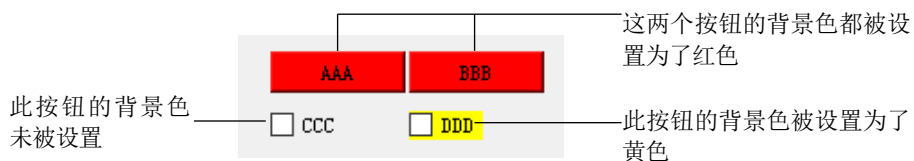
```
pc1->setStyleSheet("background-color:yellow");
```

```
//设置整个应用程序的按钮的背景色都为红色
```

```
aa.setStyleSheet("QPushButton{background-color:red}");
```

```
w.resize(444,333); w.show(); return aa.exec(); }
```

运行结果及说明



二、样式表语法基础

1、Qt 样式表通常不区分大小写，除了类名、对象名和 Qt 属性名称之外。

2、基本语法规则

- ①、样式表由选择器(selector)和声明(declaration)两部分组成，选择器指定了受影响的部件，声明指定了要设置的属性(注意与 Qt 属性的区别)。比如

```
QPushButton { color : blue }
```

其中 QPushButton 是选择器，{color: blue}是声明，color 是需要设置的属性，blue 是属性的值。以上语句表明，QPushButton 及其子类应使用 blue(蓝色)作为其前景色。

- ②、指定多个选择器

可为相同的声明指定多个选择器，选择器之间使用逗号“,”进行分隔，比如

```
QPushButton, QCheckBox, QLineEdit { color: red }
```

与以下 3 条语句相同

```
QPushButton { color: red }   QCheckBox, { color: red }   QLineEdit { color: red }
```

- ③、指定多个属性时使用分号分隔，比如

```
QPushButton { color: red; background-color: blue } //按钮前景色为红色，背景色为蓝色
```

- ④、为含有多个值的属性指定多个值时使用空隔分隔，比如

```
QLineEdit { border: 2px solid red } //表示边框线为 2 像素宽的实现，其颜色为红色
```

以上语句的属性 border 是一种对属性的简写方法，以上语句相当于以下 3 条语句：

```
QLineEdit { border-width: 2px;
              border-style: solid;
              border-color: red }
```

三、选择器

1、下表为样式表的选择器

样式表选择器		
选择器	示例	说明
全局选择器	*	匹配所有部件
类型选择器	QWidget	匹配 QWidget 及其子类的实例
属性选择器	QPushButton[flat = "false"]	1、匹配 QPushButton 的属性 flat 等于 false 的实例 2、注意：方括号内的属性是指 Qt 属性(即部件的属性)。 3、此选择器也可用于测试动态属性(setProperty()函数)。 4、也可以使用 ~= 代替 = 来测试 QStringList 类型的 Qt 属性是否包含给定的 QString。 5、注意：如果 Qt 属性的值在样式表设置后发生更改，则可能

		需要强制重新计算样式表，其方法是取消样式表并重新设置它。
类选择器	.QPushButton	匹配 QPushButton 但不匹配其子类的实例，这相当于 <code>*[class~="QPushButton"]</code>
ID 选择器	QPushButton#AAA	匹配对象名(objectName 属性)为 AAA 的所有 QPushButton 实例
后代选择器	QWidget QPushButton	匹配 QWidget 的后代(子, 孙子等)的所有 QPushButton 实例
子对象选择器	QWidget > QPushButton	匹配 QPushButton 实例，它们是 QWidget 的直接子部件

示例：样式表选择器

```

QApplication aa(argc,argv);
QWidget w;

QPushButton *pb1=new QPushButton("AAA",&w);pb1->move(22,22); pb1->resize(111,33);
QPushButton *pb2=new QPushButton("BBB",&w); pb2->move(199,22);

w.setObjectName("www"); pb1->setObjectName("AAA"); //设置对象名
/*以下语句将把按钮 AAA 的背景色设置为红色，以下语句表示，把对象名为 www 的 QWidget 的直接子部件
QPushButton 的背景色设置为红色，其中 QPushButton 的对象名为 AAA，其 x 属性和 y 属性(即 QPushButton
的位置)的值都为 22*/
aa.setStyleSheet("QWidget#www > QPushButton#AAA[x=\"22\"][y=\"22\"]{background-color:red}");

```

示例：动态属性的使用

```

QApplication aa(argc,argv);
QWidget w;
QPushButton *pb1=new QPushButton("AAA",&w); pb1->move(22,22); pb1->resize(111,33);
QPushButton *pb2=new QPushButton("BBB",&w); pb2->move(199,22);
QCheckBox *pc=new QCheckBox("CCC",&w); pc->move(22,55);
QCheckBox *pc1=new QCheckBox("DDD",&w); pc1->move(99,55);
pb1->setProperty("XXX",true); pc1->setProperty("XXX",true); //设置动态属性
//以下语句将把按钮 pb1 和 pc1 的背景色设置为红色，以下语句表示匹配所有属性"XXX = true"的部件
aa.setStyleSheet("*[XXX=true]{background-color:red}");

```

四、子控件

- 1、复杂控件通常包含一系列子控件，比如 QComboBox 的下拉按钮就是子控件，还有 QSpinBox 的向上/向下箭头等。对复杂控件使用样式表设置外观，可能需要访问其子控件。样式表中子控件以 “::” 符号开头。
- 2、下表为样式表的子控件

样式表的子控件		
注意：QScrollBar、QComboBox 等部件的一个属性或子控件是自定义的，通常还要自定义其他属性和子控件。比如，单独使用 QScrollBar 的子控件不会产生效果，需与滚动条的属性配合使用。		
	子控件	说明
滚动条、滑块	::handle	QScrollBar、QSplitter、QSlider 的手柄(滑块)
	::groove	QSlider 的凹槽
	::corner	QAbstractScrollArea 中两个滚动条之间的角落

关	::add-line	QScrollBar 增加行的按钮，即按下该按钮滚动条增加一行。
	::add-page	QScrollBar 在手柄(滑块)和增加行之间的区域
	::sub-line	QScrollBar 减少行的按钮，即按下该按钮滚动条减少一行。
	::sub-page	QScrollBar 在手柄(滑块)和减少行之间的区域
箭头 相关	::down-arrow	QComboBox、QHeaderView 排序指示器、QScrollBar、QSpinBox 的向下箭头
	::down-button	QScrollBar 或 QSpinBox 的向下按钮
	::up-arrow	QHeaderView(排序指示器)、QScrollBar、QSpinBox 的向上箭头
	::up-button	QSpinBox 的向上按钮
	::left-arrow	QScrollBar 的左箭头
	::right-arrow	QMenu 或 QScrollBar 的右箭头
模型/ 视图	::branch	QTreeView 的分支指示符
	::section	QHeaderView 的段
	::text	QAbstractItemView 的文本
其他	::chunk	QProgressBar 的进度块
	::drop-down	QComboBox 的下拉按钮
	::indicator	QAbstractItemView、QCheckBox、QRadioButton、QMenu(可被选中的)、QGroupBox(可被选中的)的指示器
选项 卡栏、	::pane	QTabWidget 的面板(边框)
	::right-corner	QTabWidget 的右角落，此控件可用于控件 QTabWidget 中右角落部件的位置
	::left-corner	QTabWidget 的左角落，此控件可用于控件 QTabWidget 中左角落部件的位置
选项 卡部 件、	::tab-bar	QTabWidget 的选项卡栏，此子控件仅用于控制 QTabBar 在 QTabWidget 中的位置，使用::tab 设置选项卡的样式。
	::tab	QTabBar 或 QToolBox 的选项卡
	::tear	QTabBar 的可分离指示器
可停 靠窗 口	::close-button	QTabBar 选项卡或 QDockWidget 上的关闭按钮
	::float-button	QDockWidget 的浮动按钮
	::title	QDockWidget 或 QGroupBox 的标题
菜单 相关	::scroller	QMenu 或 QTabBar 的滚动条
	::separator	QMenu 或 QMainWindow 中的分隔符
	::tearoff	QMenu 的可分离指示器
	::item	QAbstractItemView、QMenuBar、QMenu、QStatusBar 中的一个项
	::icon	QAbstractItemView 或 QMenu 的图标
	::menu-arrow	带有菜单的 QToolButton 的箭头
	::menu-button	QToolButton 的菜单按钮
	::menu-indicator	QPushButton 的菜单指示器

五、伪状态

1、部件含有一系列的状态，样式表的状态使用“:”符号开头。比如

`QPushButton:hover{...}` //表示鼠标悬停在按钮上

2、伪状态的规则

①、用感叹号可否定伪状态，比如

`QPushButton:!hover{...}` //表示鼠标未悬停在按钮上

②、多个伪状态连用可达到逻辑与(AND)的效果，比如

`QCheckBox:hover:checked{...}` //表示鼠标悬停在按钮上且被选中

③、多个伪状态之间使用逗号分隔可表示逻辑或(OR)的效果，比如

`QCheckBox:hover, QCheckBox:checked{...}` //表示鼠标悬停在按钮上或被选中，注意语法的写法

④、伪状态还可以子控件联合使用，比如

`QComboBox::drop-down: hover{...}`

3、下表为样式表的伪状态

样式表的伪状态		
	伪状态	说明
常见状态	:active	部件位于激活窗口中
	:focus	该项具有输入焦点
	:edit-focus	该项具有编辑焦点，此状态仅适用于 Qt Extended 应用程序
	:default	该项是默认值
	:disabled	该项已被禁用
	:enabled	该项已启用
	:hover	鼠标悬停在该项上。
	:pressed	使用鼠标按下该项
	:no-frame	该项没有边框，比如，无边框的 <code>QLineEdit</code> 等
	:flat	该项是平的(flat)，比如，一个平的 <code>QPushButton</code>
	:checked	该项被选中
	:unchecked	该项未被选中
	:off	适用于处于“关闭(off)”状态的项
	:on	适用于处于“开启(on)”状态的项
	:editable	<code>QComboBox</code> 是可编辑的
	:read-only	该项为只读，比如，只读的 <code>QLineEdit</code>
	:indeterminate	该项具有不确定状态，比如，三态的 <code>QCheckBox</code>
方位相关	:exclusive	该项是排他项目组的一部分。
	:non-exclusive	该项是非排他项目组的一部分。
	:bottom	该项位于底部
	:top	该项位于顶部
	:left	该项位于左侧，比如， <code>QTabBar</code> 的选项卡位于左侧
	:right	该项位于右侧，比如， <code>QTabBar</code> 的选项卡位于右侧
	:middle	该项位于中间，比如，不在 <code>QTabBar</code> 开头或结尾的选项卡。
	:first	该项是第一个，比如， <code>QTabBar</code> 中的第一个选项卡
其他状态	:last	该项是最后一个，比如， <code>QTabBar</code> 中的最后一个选项卡
	:horizontal	该项具有水平方向
	:vertical	该项具有垂直方向
	:maximized	该项是最大化的，比如，最大化的 <code>QMdiSubWindow</code>
	:minimized	该项是最小化的，比如，最小化的 <code>QMdiSubWindow</code>
	:floatable	该项是可浮动的。
	:movable	该项可移动，例如，可移动的 <code>QDockWidget</code>
	:only-one	该项是唯一的，比如，只有一个选项卡的 <code>QTabBar</code>
	:next-selected	下一项被选择
	:previous-selected	上一项被选择
	:selected	该项被选择
	:window	部件是一个窗口，即顶级部件
	:closable	该项可被关闭，例如，可关闭的 <code>QDockWidget</code>
	:closed	该项处于关闭状态，比如 <code>QTreeView</code> 中的非展开项
	:open	该项处于打开状态，比如 <code>QTreeView</code> 中的展开项，或带有打开菜单的 <code>QComboBox</code> 或 <code>QPushButton</code>
	:has-children	该项具有孩子，比如， <code>QTreeView</code> 中具有子项的项
	:has-siblings	该项具有兄弟姐妹(即同级的)

	:adjoins-item	QTreeView 的::branch 与项相邻时设置此状态
	:alternate	当 QAbstractItemView::alternatingRowColors()被设置为 true 时，为每个交替行设置此状态，以绘制 QAbstractItemView 的行。

13.9、样式表的属性

注：类型为属性可取值的属性类型(注意：不是 Qt 类型，属性类型及可取值见后文)

本小节把 border 也翻译为边框，因此需注意与 frame(边框)的区别，可把 border 译为边界

一、背景色、前景色、所选文本的颜色

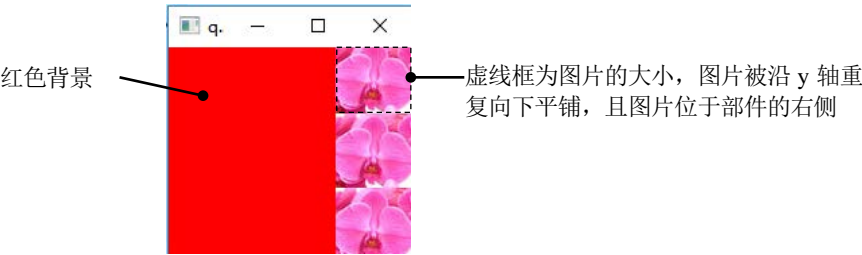
样式表的背景色、前景色、所选文本的颜色		
属性	类型	说明
1、 background	Background	设置背景的简写方法，相当于指定 background-color、background-image、background-repeat、background-position。以下类支持此属性 QAbstractItemView 子类，QAbstractSpinBox 子类，QCheckBox，QComboBox，QDialog，QFrame，QGroupBox，QLabel，QLineEdit，QRadioButton，QSplitter，QTextEdit，QToolTip 和 QWidget 注意：若仅在 QPushButton 上设置背景色，除非把 border 属性设置为某个值，否则背景可能不会被显示。
2、 background-color	Brush	部件的背景色
3、 background-image	Url	设置部件的背景图像，背景图像不会随部件的大小自动缩放，比如 <code>QWidget{ background-image: url(F:/li.png) }</code>
4、 background-repeat	Repeat	如何使用背景图像填充背景区域 background-origin，若未指定此属性，则在两个方向重复背景图像。
5、 background-position	Alignment	背景图像在 background-origin 矩形内的位置，默认为 top left
6、 background-attachment	Attachment	确定 QAbstractScrollArea 中的 background-image 是相对于视口滚动还是固定，默认值为 scroll，即，使用视口滚动
7、 background-clip	Origin	<ul style="list-style-type: none">部件绘制背景的矩形，此属性指定 background-color 和 background-image 的裁剪矩形，此属性默认值为 border(即边框矩形)，其示例见盒子模型以下类支持此属性： QAbstractItemView 子类，QAbstractSpinBox 子类，QCheckBox，QComboBox，QDialog，QFrame，QGroupBox，QLabel，QPushButton，QRadioButton，QSplitter，QTextEdit，QToolTip，QWidget
8、 background-origin	Origin	<ul style="list-style-type: none">部件背景的原点矩形，通常与 background-position 和 background-image 一起使用，默认为 padding(即填充矩形)，其示例见盒子模型以下类支持此属性： QAbstractItemView 子类，QAbstractSpinBox 子类，QCheckBox，QComboBox，QFrame，QGroupBox，QLabel，QPushButton，QRadioButton，QSplitter，QTextEdit，QToolTip，QWidget，QLineEdit，QMenu，QMenuBar(注意：没有 QDialog)
9、 color	Brush	渲染文本的颜色，所有遵守 QWidget::palette 的部件都支持此属性
10、 selection-background-color	Brush	所选文本或项的背景色，所有遵守 QWidget::palette 和显示选择文本的部件都支持此属性，默认为调色板的 QPalette::Highlight 角色的值

11、 selection-color	Brush	所选文本或项的前景色，所有遵守 QWidget::palette 和显示选择文本的部件都支持此属性，默认为调色板的 QPalette::HighlightedText 角色的值
----------------------------	-------	--

示例：使用样式表设置部件的背景色

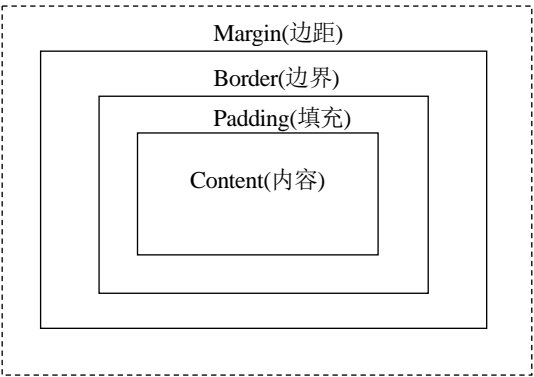
```
#include<QtWidgets>
int main(int argc, char *argv[]) {    QApplication aa(argc, argv);
    QWidget w;
    aa.setStyleSheet(
        //背景色为红色，背景图片为 li.png。注意为属性指定多个值的方法
        "QWidget {background: red url(F:/li.png);"
        "background-repeat: repeat-y;"           //在 y 轴方向重复图片
        "background-position: right;}");         //图片位于部件右侧
    w.resize(300,200);    w.show();    return aa.exec();    }
```

运行结果及说明



二、盒子模型及相关属性

1、样式表把每个部件都被视为 4 个同心矩形的盒子，如下图所示



- 说明：
- 1、Content(内容)矩形是除掉边距、边框和填充之后的部分，这意味着没有单独的与内容矩形有关的属性
 - 2、默认情况下，边距、边框和填充的值都为 0，因此这 4 个矩形恰好重合

盒子模型

2、样式表的边框属性

样式表的边框属性主要描述的是边框的边界线, 包括边界线的颜色、样式(虚线、实线等)、宽度、角落半径和填充边框的图像。下面分别介绍这些属性

注意: 由于边框(border)、填充矩形(padding)、轮廓线(outline)的边界线默认宽度为 0, 且样式为 none(即没有线条), 因此要使其边界线可见, 必须指定边界线的宽度和样式, 否则边界线不可见。

①、边框的简写方法

1)、border 类型: Border

- 设置边框的简写方法, 相当于指定 border-color, border-style, border-width
- 以下类支持此属性:
QAbstractItemView 子类, QAbstractSpinBox 子类, QCheckBox, QComboBox, QDialog, QFrame, QGroupBox, QLabel, QPushButton, QRadioButton, QSplitter, QTextEdit, QToolTip, QWidget

2)、border-top 类型: Border

设置部件顶部边框的简写方法, 相当于指定 border-top-color, border-top-style, border-top-width

3)、border-right 类型: Border

设置部件右边框的简写方法, 相当于指定 border-right-color, border-right-style, border-right-width

4)、border-bottom 类型: Border

设置部件底部边框的简写方法, 相当于指定 border-bottom-color, border-bottom-style, border-bottom-width

5)、border-left 类型: Border

设置部件左边框的简写方法, 相当于指定 border-left-color, border-left-style, border-left-width

②、边框颜色

6)、border-color 类型: Box Colors

- 边框边界线的颜色, 相当于指定 border-top-color, border-bottom-color, border-left-color, border-right-color, 默认值为 color(即部件的前景色)
- 以下类支持此属性:
QAbstractItemView 子类, QAbstractSpinBox 子类, QCheckBox, QComboBox, QFrame, QGroupBox, QLabel, QPushButton, QRadioButton, QSplitter, QTextEdit, QToolTip, QWidget, QLineEdit, QMenu, QMenuBar(注意: 没有 QDialog)

7)、border-top-color 类型: Brush //边框顶部边界线的颜色

8)、border-right-color 类型: Brush //边框右边界线的颜色

9)、border-bottom-color 类型: Brush //边框底部边界线的颜色

10)、border-left-color 类型: Brush //边框左边界线的颜色

③、边框半径

11)、border-radius 类型: Radius

- 边框角落的半径, 等效于指定 border-top-left-radius, border-top-right-radius, border-bottom-left-radius, border-bottom-right-radius, 该属性剪切元素的 background(背景), 默认为 0。
- 以下类支持此属性:
QAbstractItemView 子类, QAbstractSpinBox 子类, QCheckBox, QComboBox, QFrame, QGroupBox, QLabel, QPushButton, QRadioButton, QSplitter, QTextEdit, QToolTip, QLineEdit, QMenu, QMenuBar(注意: 没有 QDialog 和 QWidget)

12)、border-top-left-radius 类型: Radius //边框左上角的半径

13)、border-top-right-radius 类型: Radius //边框右上角的半径

14)、border-bottom-right-radius 类型: Radius //边框右下角的半径

15)、border-bottom-left-radius 类型: Radius //边框左下角的半径

④、边框样式

- 16)、**border-style** 类型: Border Style
- 边框边界线的样式(虚线、实线、点划线等), 默认为 none
 - 以下类支持此属性:
QAbstractItemView 子类, QAbstractSpinBox 子类, QCheckBox, QComboBox, QFrame, QGroupBox, QLabel, QPushButton, QRadioButton, QSplitter, QTextEdit, QToolTip, QLineEdit, QMenu, QMenuBar(注意: 没有 QDialog 和 QWidget)
- 17)、**border-top-style** 类型: Border Style //边框顶部边界线的样式
- 18)、**border-right-style** 类型: Border Style //边框右侧边界线的样式
- 19)、**border-bottom-style** 类型: Border Style //边框底部边界线的样式
- 20)、**border-left-style** 类型: Border Style //边框左侧边界线的样式

⑤、边框宽度

- 21)、**border-width** 类型: Border Lengths
- 边框的宽度, 等效于指定 border-top-width, border-bottom-width, border-left-width, border-right-width,
 - 以下类支持此属性:
QAbstractItemView 子类, QAbstractSpinBox 子类, QCheckBox, QComboBox, QFrame, QGroupBox, QLabel, QPushButton, QRadioButton, QSplitter, QTextEdit, QToolTip, QLineEdit, QMenu, QMenuBar(注意: 没有 QDialog 和 QWidget)
- 22)、**border-top-width** 类型: Length //边框顶部边界线的宽度
- 23)、**border-right-width** 类型: Length //边框右侧边界线的宽度
- 24)、**border-bottom-width** 类型: Length //边框底部边界线的宽度
- 25)、**border-left-width** 类型: Length //边框左侧边界线的宽度

⑥、边框图像

- 26)、**border-image** 类型: Border Image
- 填充边框的图像, 该图像被分割成 9 个部分, 并在必要时适当地拉伸, 详见后文。
 - 以下类支持此属性:
QAbstractItemView 子类, QAbstractSpinBox 子类, QCheckBox, QComboBox, QFrame, QGroupBox, QLabel, QPushButton, QRadioButton, QSplitter, QTextEdit, QToolTip, QLineEdit, QMenu, QMenuBar(注意: 没有 QDialog 和 QWidget)

3、样式表的边距属性

- 27)、**margin** 类型: Box Lengths
- 部件的边距, 等效于指定 margin-top, margin-right, margin-bottom, margin-left, 默认为 0
 - 以下类支持此属性:
QAbstractItemView 子类, QAbstractSpinBox 子类, QCheckBox, QComboBox, QFrame, QGroupBox, QLabel, QPushButton, QRadioButton, QSplitter, QTextEdit, QToolTip, QLineEdit, QMenu, QMenuBar(注意: 没有 QDialog 和 QWidget)
- 28)、**margin-top** 类型: Length //部件的顶部边距
- 29)、**margin-right** 类型: Length //部件的右侧边距
- 30)、**margin-bottom** 类型: Length //部件的底部边距
- 31)、**margin-left** 类型: Length //部件的左侧边距

4、样式表的填充属性

- 32)、**padding** 类型: Box Lengths
- 部件的填充矩形, 等效于指定 padding-top, padding-right, padding-bottom, padding-left, 默认为 0
 - 以下类支持此属性:
QAbstractItemView 子类, QAbstractSpinBox 子类, QCheckBox, QComboBox, QFrame, QGroupBox, QLabel, QPushButton, QRadioButton, QSplitter, QTextEdit, QToolTip, QLineEdit, QMenu, QMenuBar(注意: 没有 QDialog 和 QWidget)

33)、padding-top	类型: Length	//填充矩形顶部离边框顶部的距离
34)、padding-right	类型: Length	//填充矩形右侧离边框右侧的距离
35)、padding-bottom	类型: Length	//填充矩形底部离边框底部的距离
36)、padding-left	类型: Length	//填充矩形左侧离边框左侧的距离

5、样式表的轮廓线属性

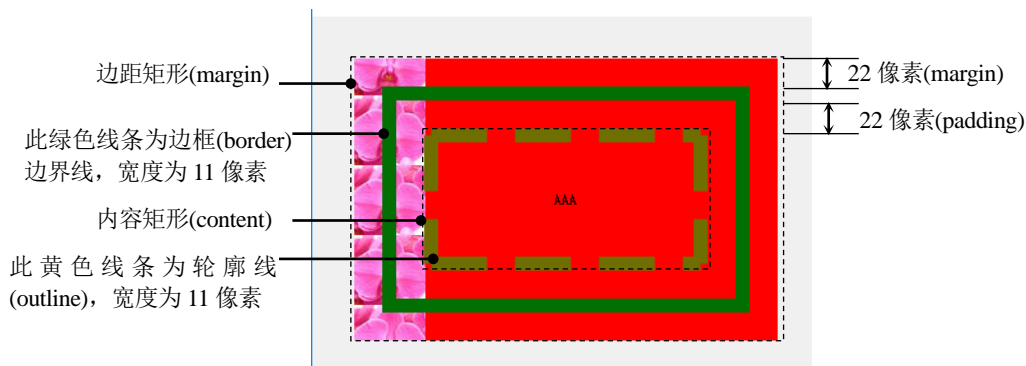
37)、outline		//绘制在对象边框上的轮廓
38)、outline-color	类型: Color	//轮廓线的颜色
39)、outline-offset	类型: Length	//轮廓与部件边框的偏移量
40)、outline-style		//绘制轮廓的图案(pattern), 另见 border-style
41)、outline-radius		//轮廓的圆角
42)、outline-bottom-left-radius	类型: Radius	//轮廓左下角的半径
43)、outline-bottom-right-radius	类型: Radius	//轮廓右下角的半径
44)、outline-top-left-radius	类型: Radius	//轮廓左上角的半径
45)、outline-top-right-radius	类型: Radius	//轮廓右上角的半径

示例：理解盒子模型

//border、outline、padding、background-clip、background-origin 属性的使用

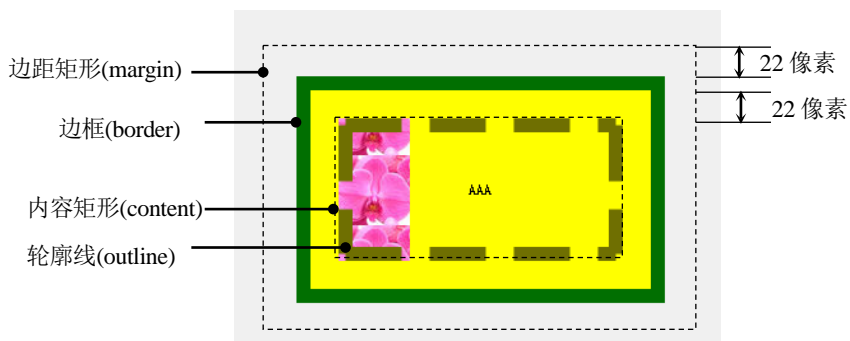
```
#include<QtWidgets>
int main(int argc, char *argv[]) {    QApplication aa(argc, argv);
    QWidget w;    QPushButton *pb=new QPushButton("AAA",&w);
    pb->move(33, 33);    pb->resize(333, 222);
    aa.setStyleSheet(
        "QPushButton{"
            "background: red url(F:/li.png);"    //背景色为红色, 背景图片为 li.png
            "background-repeat: repeat-y;"    //在 y 轴方向重复图片
            "background-position: left;"    //图片位于左侧
            //绘制一个 11 像素宽的红色实线边框线, 注意值的顺序
            "border: 11px solid rgb(1, 111, 1);"
            "margin: 22px;"    //部件的边距为 22 像素
            "padding: 22px;"    //部件填充距离为 22 像素
            "background-clip: margin;"    //背景色填充整个边距矩形
            "background-origin: margin;"    //背景图片的原点位于边界矩形
            "outline: 11 dashed rgb(111, 111, 1);}"    //绘制一个 11 像素宽的黄色虚线轮廓线
        "QPushButton:pressed {"    //当按钮按下时的样式
            "background-color: yellow;"    //背景色为黄色
            "background-clip: border;"    //背景色填充边界矩形
            "background-origin: content;}"    //背景图片的原点位于内容矩形
    );
    w.resize(400, 300);    w.show();    return aa.exec();    }
```

运行结果及说明



按钮按下前的效果:

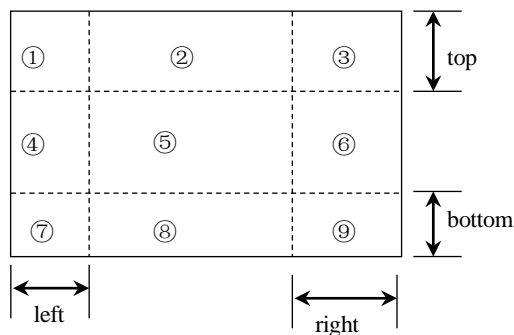
此时背景色(红色)填充整个边距矩形(margin), 背景图片原点位于边距矩形的左上角



按钮按下后的效果:

此时背景色(黄色)只填充了边界矩形(border), 背景图片原点位于内容矩形(content)的左上角

6、边框图像(border-image)原理



- 1)、如上图, 边框图像被 4 条虚线划分为 3*3 的小格, 从而把图像分为 9 个区域
- 2)、当使用边框图像填充部件背景时, 4 个角(即格子①、③、⑦、⑨)保持不变, 其他 5 个格子被拉伸或重复(即平铺)
- 3)、边框图像的设计原则如下:

- 4 条虚线分别使用从上、右、下、左边缘的距离设置，如图中 top、right、bottom、left 所示。
- 使用边框图像还必须明确的设置边框的宽度(即 border-width 属性)，通常把边框宽度设置为与 4 条虚线的值相一致，否则，4 个角的图像将被适当的拉伸或压缩以适应边框的大小。

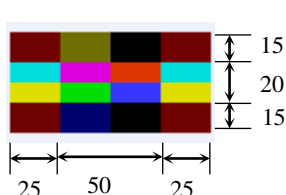
4)、边框图像(border-image)与背景图像(background-image)的显著区别是，背景图像不会随窗口部件的大小而缩放。

5)、若同时指定了边框图像和背景图像，则边框图像会绘制在背景图像之上。

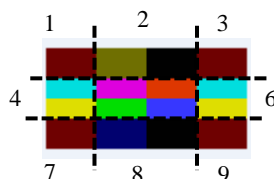
示例：边框图像原理(border-image 属性)

```
#include<QtWidgets>
int main(int argc, char *argv[]) {    QApplication aa(argc, argv);
    QWidget w;
    QPushButton *pb=new QPushButton("AAA",&w);    pb->move(33,33);    pb->resize(300,200);
    aa.setStyleSheet(
        "QPushButton{
/*设置边框图像，其上、右、下、左的边缘距离分别为 15,25,15,25，除 4 个角外，其余区域的图
像被重复绘制(平铺)，若要使其拉伸，只需把 repeat 修改为 stretch 即可。*/
    \"border-image:url(F:/1x.png) 15 25 15 25 repeat;\"
    \"border-width:55;}\"    //必须设置边框宽度
        );
    w.resize(400,300);    w.show();    return aa.exec();    }
```

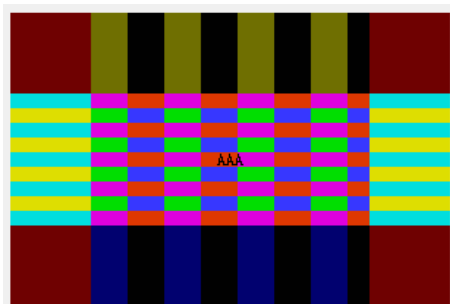
运行结果及说明



原始图像，该图像可使用 QPainter 类自行绘制

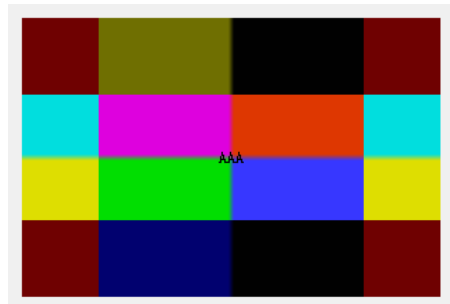


图像的划分方法



由图可见，图像的 4 个角只是被适当的拉伸或压缩，区域 2、4、5、6、8 的图形被依次按各自的方向重复绘制(即平铺)

重复边框图像的效果



由图可见，图像的 4 个角只是被适当的拉伸或压缩，区域 2、4、5、6、8 的图形被拉伸未被重复绘制

拉伸边框图像的效果

三、与位置和大小有关的属性

- 1、**subcontrol-origin** 类型: Origin //子控件的原点矩形, 默认为 padding。
- 2、**subcontrol-position** 类型: Alignment
子控件在 subcontrol-origin 属性指定的矩形内的对齐方式, 默认值取决于子控件
- 3、**position** 类型: relative | absolute
使用 left、right、top、bottom 属性的偏移是相对坐标还是绝对坐标。默认为 relative(相对的)
- 4、**spacing** 类型: Length
 - 部件的内部间距(比如复选按钮和文本之间的距离), 默认值取决于当前样式
 - 以下类支持此属性: QCheckBox, 可选中的 QGroupBox, QRadioButton, QMenuBar
- 5、**bottom** 类型: Length
left 类型: Length
top 类型: Length
right 类型: Length
以 bottom 属性为例(其余属性类似), 若 position 属性是 relative(相对的, 默认值), 则子控件向上移动一偏移量, 若 position 是 absolute(绝对的), 则 bottom 属性是指与子控件的下边缘的距离, 该距离与 subcontrol-origin 属性有关。此属性默认值为 0。
- 6、**height** 类型: Length
width 类型: Length
子控件的高度/宽度, 默认值取决于当前样式, 注意: 除非另有规定, 否则在部件上设置此属性无效, 若想要部件有一个固定的高度, 应把 min-height 和 max-height 设置为相同的值, 宽度类似。
- 7、**max-height** 类型: Length
max-width 类型: Length
 - 部件或子控件的最大高度/宽度, 以上值相对于盒子模型的内容矩形
 - 以下类支持此属性:
QAbstractItemView 子类, QAbstractSpinBox 子类, QCheckBox, QComboBox, QFrame, QGroupBox, QLabel, QPushButton, QRadioButton, QSizeGrip, QSpinBox, QSplitter, QStatusBar, QTextEdit, QToolTip, QLineEdit, QMenu, QMenuBar, (注意: 没有 QDialog 和 QWidget)
- 8、**min-height** 类型: Length
min-width 类型: Length
 - 部件或子控件的最小高度/宽度, 默认值依赖于部件的内容和样式, 该值相对于盒子模型的内容矩形
 - 以下类支持此属性:
QAbstractItemView 子类, QAbstractSpinBox 子类, QCheckBox, QComboBox, QFrame, QGroupBox, QLabel, QPushButton, QRadioButton, QSizeGrip, QSpinBox, QSplitter, QStatusBar, QTextEdit, QToolTip, QLineEdit, QMenu, QMenuBar, (注意: 没有 QDialog 和 QWidget)

示例: 设置子控件的位置和大小

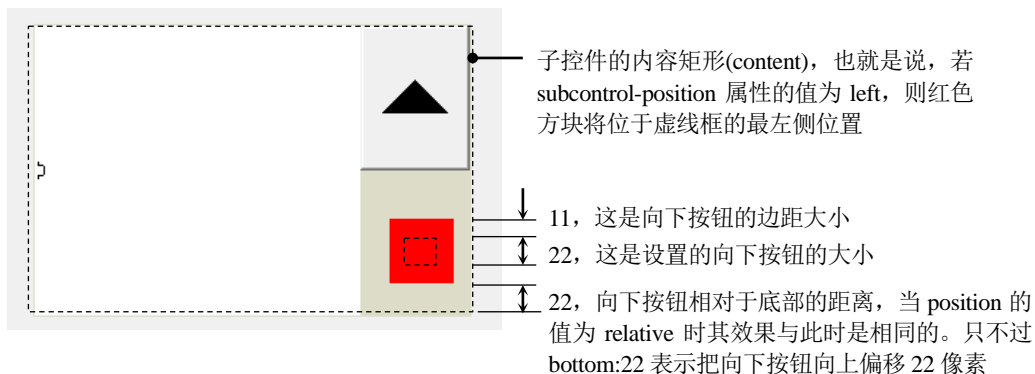
```
#include<QtWidgets>
int main(int argc, char *argv[]) {    QApplication aa(argc, argv);
    QWidget w;    QSpinBox *pb=new QSpinBox(&w);    pb->move(33,33);
    pb->resize(300,200); //为使效果比较明显, 把按钮设置得相对大一点
    aa.setStyleSheet(
        "QSpinBox::down-button{"           //设置微调按钮的向下按钮
        "subcontrol-origin:content;"       //子控件的原点矩形
        "subcontrol-position:right bottom;" //子控件相对于原点矩形的对齐方式
        "height:22px;" "width:22px;"      //设置子控件(即向下按钮)的大小(即宽度和高度)
//设置向下按钮的位置使用绝对坐标指定, 若要使用相对坐标, 只需把 absolute 修改为 relative 即可
        "position:absolute;"
        "bottom:22px;" "right:11px;"
        "margin:11px;"                     //设置向下按钮的边距
        "background-color:red;"           //使用红色填充子控件(向下按钮)背景
    );
}
```

```

        "background-clip:margin"           //填充背景的区域为边距矩形(margin)范围
    }"
    );
    w.resize(400,300);    w.show();    return aa.exec();}

```

运行结果及说明



注: 虚线的矩形框实际图形是不存在的, 因为向下按钮被红色背景填充了, 所以向下按钮不再显示为箭头形状

四、字体、文本、图标、图像、不透明度属性

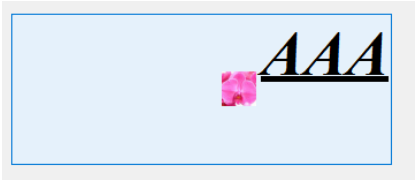
样式表的字体、文本、图像等		
属性	类型	说明
1、 icon-size	Length	部件中图标的宽度和高度, 可设置以下部件的图标大小: QCheckBox, QListView, QPushButton, QRadioButton, QTabBar, QToolBar, QToolBox, QTreeView
2、 text-align	Alignment	部件中文本和图标的对齐方式, 默认值取决于样式。
3、 text-decoration	none underline overline line-through	附加的文本的效果
4、 font	Font	设置文本字体的简写方法, 相当于指定 font-family, font-size, font-style, font-weight, 所有遵守 QWidget::font 属性的部件都支持此属性。默认为 QWidget::font 属性
5、 font-family	String	字体系列(字体族)
6、 font-size	Font Size	字体的大小, 仅支持 pt 和 px(像素)
7、 font-style	Font Style	字体的样式(比如是否倾斜)
8、 font-weight	Font Weight	字体的重量(即字体的粗细)
9、 image	Url	<ul style="list-style-type: none"> 绘制在子控件内容矩形(content)中的图像, 绘制的图像使用与 QIcon 相同的算法, 即图像不会放大, 但在必要时会缩小, 在子控件上设置该属性, 会隐式设置子控件的宽度和高度(除非使用 svg 图像)。 Qt4.3 之后可使用 image-position 来设置图像在矩形内的对齐方

		<p>式。</p> <ul style="list-style-type: none">● 此属性仅用于子控件，不支持其他元素。● 若指定了 <code>svg</code>，则图像将被缩放到内容矩形的大小，注意：<code>svg</code> 需使用 <code>svg</code> 插件。● 若还同时设置了背景图像(<code>background-image</code>)和边框图像(<code>border-image</code>)，则 <code>image</code> 绘制在 <code>border-image</code> 之上，<code>border-image</code> 绘制在 <code>background-image</code> 之上。
10、 <code>image-position</code>	Alignment	图像的位置(对齐方式)
11、 <code>opacity</code>	Number	部件的不透明度，其值为 0(透明)到 255(不透明)，目前仅支持工具提示。默认值为当前样式 <code>QStyle::SH_ToolTipLabel_Opacity</code> 指定的值。

示例：设置字体、文本、图标

```
#include<QtWidgets>
int main(int argc, char *argv[]){    QApplication aa(argc,argv);
    QWidget w;    QPushButton *pbl=new QPushButton("AAA",&w);pbl->move(22,22);
    pbl->setIcon(QIcon("F:/li.png"));    //添加一个图标
    pbl->resize(333,133);    //为使效果比较明显，把按钮设置得相对大一点
    aa.setStyleSheet(
        "QPushButton{"           //设置微调按钮的向下按钮
        "icon-size:30px;"        //注意：此处必须带上单位(px 表示像素)
        "text-align:top right;"   //文本对齐方式
        "text-decoration:underline;" //为文本添加下划线
/*设置字体，以下属性需按照字体度量、样式、大小、字体族的顺序排列，且字体大小必须带上单位，
否则属性将不起作用。*/
        "font:900 italic 55px Times New Roman;}"
    );
    w.resize(400,300);    w.show();    return aa.exec(); }
```

运行结果及说明



五、其他属性

- 1、`button-layout` 类型： Number
QDialogButtonBox 或 QMessageBox 中按钮的布局，可取值为 0，1，2，3，5，其意义如下：
0: QDialogButtonBox::WinLayout,
1: QDialogButtonBox::MacLayout,
2: QDialogButtonBox::KdeLayout,
3: QDialogButtonBox::GnomeLayout),
5: QDialogButtonBox::AndroidLayout,
默认值为 `QStyle::SH_DialogButtonLayout` 的值
- 2、`dialogbuttonbox-buttons-have-icons` 类型： Boolean //QDialogButtonBox 中的按钮是否显示图标
- 3、`lineEdit-password-character` 类型： Number

QLineEdit 的密码字符(使用 unicode 数字), 比如: `*{lineedit-password-character:9679}`

默认为 `QStyle::SH_LineEdit_PasswordCharacter` 的值

- 4、**lineedit-password-mask-delay** 类型: Number //qt5.4

QLineEdit 在显示密码字符之前的延迟时间(毫秒), 默认为 `QStyle::SH_LineEdit_PasswordMaskDelay` 的值

- 5、**messagebox-text-interaction-flags** 类型: Number

消息框中与文本的交互行为, 可取值基于 `Qt::TextInteractionFlags` 标志, 比如

`QMessageBox{messagebox-text-interaction-flags:5};` //即 `Qt::LinksAccessibleByMouse`(值为 4)与
`Qt::TextSelectableByMouse`(值为 1)的按位或。

默认值为 `QStyle::SH_MessageBox_TextInteractionFlags` 的值,

- 6、**titlebar-show-tooltips-on-buttons** 类型: bool //是否在工具栏按钮上显示工具提示

- 7、**widget-animation-duration** 类型: Number //qt5.10

动画的持续时间(毫秒), 0 意味着禁用动画, 默认值为 `QStyle::SH_Widget_Animation_Duration` 的值

- 8、**alternate-background-color** 类型: Brush

QAbstractItemView 的交替背景色, 若未设置此属性则使用 `QPalette::AlternateBase` 角色的颜色

- 9、**paint-alternating-row-colors-for-empty-area** 类型: bool

QTreeView 是否为空区域(即没有项的区域)绘制交替行颜色

- 10、**show-decoration-selected** 类型: Boolean

在 QListView 中选择是覆盖整行还是仅覆盖文本, 默认为 `QStyle::SH_ItemView_ShowDecorationSelected` 的值。

- 11、**gridline-color** 类型: Color

QTableView 中网格线的颜色, 默认为 `QStyle::SH_Table_GridLineColor` 的值

六、属性类型

1、各符号的含义

- | : 分隔符, 比如 `0|1`, 表示 0 或者 1
- * : 表示 0 或更多,
- + : 表示 1 或更多
- ? : 表示 0 或 1, 比如 `(on|off)?`, 表示可取 on 或 off 之一, 或一个都不取
- {0,4} : 表示 0 到 4, 比如 `Length{0,4}`, 表示 0 到 4 个 Length 的值

示例:

- ①、`{top|bottom|left|right|center}*` //表示可取大括号中任意值的组合比如 `top left` 表示左上

- ②、**(Font Style | Font Weight){0,2} Font Size String**

以上语法表示, 其值的格式必须按照如下顺序指定(若顺序不正确, 则取值无效):

“Font Style 或 Font Weight 的值之一或无, 字体大小, 一个字符串”,

假设 Font Style 属性类型可取值为 `italic`, Font Weight 属性类型可取值为为一整数, Font Size 属性类型可取值为为一像素值, String 为一字符串, 则以下值是有效的取值

`italic 55px "Calibri"` //斜体, 字的大小为 55 像素, 字体族为 Calibri

`900 55px "Calibri"` //字体粗细为 900, 大小为 55 像素, 字体族为 Calibri

`900 italic 55px "Calibri"` 或 `italic 900 55px "Calibri"` //Font Style 和 Font Weight 的位置无关紧要

以下为无效的取值

900 55px italic "Calibri" //Font Size 和 Font Style 的顺序不对

55px italic "Calibri" //原因同上

- 2、**Alignment** 值: {top | bottom | left | right | center}*

对齐方式, 比如 QPushButton{background-position:top right} //右上角对齐

- 3、**Url** 值: url(filename)

其中的 filename 是本地磁盘或 Qt 资源系统的文件的名称, 比如

QPushButton{background-image:url(F:/1i.png)} //读取文件 F:/1i.png 作为背景图像

- 4、**Attachment** 值: {scroll | fixed}*

滚动或固定, 属性 background-attachment 使用该类型, 以用于 QAbstractScrollArea

- 5、**Background** 值: {Brush | Url | Repeat | Alignment}* //参见各属性类型的取值

- 6、**Boolean** 值: 0 | 1 //布尔值, 取 0 或 1

- 7、**Border** 值: {Border Style | Length | Brush}* //参见各属性类型的取值

- 8、**Border Image** 值: none | Url Number{4} (stretch | repeat) {0, 2}

该值用于指定边框图像(border-image), 注意值的格式, 比如

```
QPushButton{
    border-image:url(F:/1x.png) 15 25 15 25 repeat;
    border-width:55;} //指定边框图像还需设置边框宽度
```

- 9、**Border Style**

值: dashed | dot-dash | dot-dot-dash | dotted | double | groove | inset | outset | ridge | solid | none

该属性类型用于指定边界线的样式, 比如 dashed 表示虚线, dotted 表示点线等

- 10、**Box Colors** 值: Brush{1, 4}

1 到 4 个 Brush 值, 分别指定盒子的 top、right、bottom、left 边界线, 若未指定 left, 则将与 right 相同, 若未指定 bottom, 则与 top 相同, 若未指定正确的颜色, 则与 top 的相同, 下面为示例:

QLabel{border-color:yellow} //四条边界线都为黄色

QLabel{border-color:yellow red} //top、bottom 为黄色, left 和 right 为红色

QLabel{border-color:yellow red blue} //top 为黄色, right 和 left 为红色, bottom 为蓝色

QLabel{border-color:yellow red blue green} //top 为黄色, right 为红色, bottom 为蓝色, left 为绿色

- 11、**Box Lengths** 值: Length{1, 4}

1 到 4 个 Length 值, 分别指定盒子的 top、right、bottom、left 边界线, 其原理与 Box Colors 相同。

- 12、**Brush** 值: Color | Gradient | PaletteRole

具体可参见各属性类型的取值, 注意: Brush 属性类型还可指定渐变色(即 Gradient 属性类型)

- 13、**Color** 值: rgb(r, g, b) | rgba(r, g, b, a) | hsv(h, s, v) | hsva(h, s, v, a) | #rrggbb | Color Name

指定颜色, 从其值可看到, 颜色可使用 6 种方式指定, 其中 rgb()和 rgba()可取 0~255 之间的值, 或一个百分比值, hsv()或 hsva 中的 s 和 v 必须是 0~255 之间的值, h 的值是 0~359 之间的值, 比如

QLabel{border-color: rgba(111,11,11, 70%);}

- 14、**Font** 值: (Font Style | Font Weight){0, 2} Font Size String

字体属性, 对该属性取值的讲解, 参见前文。

- 15、**Font Size** 值: Length //字体的大小

- 16、**Font Style** 值: normal | italic | oblique //字体的样式

- 17、**Font Weight** 值: normal | bold | 100 | 200 | ... | 900 //字体的重量(即粗细)

- 18、**Length** 值: Number(px | pt | em | ex)?

数字后跟一个测量单位(比如像素), 在 Qt 中, 必须指定测量单位, 为了早期的 Qt 版本兼容, 大多数情况下, 若未指定测量单位, 则被视为像素, 支持的单位如下:

px: 像素

- pt: 点(即 1/72 英寸)
- em: 字体的 em 宽度, 即字母 M 的宽度
- ex: 字体的 ex 宽度, 即字母 X 的宽度
- 19、**Number** 值: 一个 10 进制整数和实数, 比如 18, 23.46 等
- 20、**Origin** 值: margin | border | padding | content
指定盒子模型中的 4 个矩形, 详见盒子模型原理(前文)
- 21、**PaletteRole**
值: alternate-base | base | bright-text | button | button-text | dark | highlight | highlighted-text | light | link | link-visited | mid | midlight | shadow | text | window | window-text
该属性类型的取值对应于 QPalette::ColorRole 枚举。
- 22、**Radius** 值: **Length**{1,2}
1 到 2 个 Length, 若只指定了一个 Length, 则表示角的 1/4 圆的半径, 若指定两个长度, 则第一个长度是 1/4 椭圆的水平半径, 第二个长度是垂直半径
- 23、**Repeat** 值: repeat-x | repeat-y | repeat | no-repeat
表示重复性质(可实现图像的平铺), 其中 repeat-x 表示水平方向重复, repeat-y 表示垂直方向重复, repeat 表示水平和垂直方向重复, no-repeat 表示没有重复。
- 24、**Icon** 值: (Url (disabled | active | normal | selected)?(on | off) ?) *
指定图标(图标属性见下表), 示例

```
QMessageBox {
    dialogbuttonbox-buttons-have-icons: true;    //显示图标
    dialog-ok-icon: url(ok.svg);                  //ok 按钮的图标
    dialog-cancel-icon: url(cancel.png),          //cancel 按钮的图标
    url(grayed_cancel.png) disabled; } //禁用时 cancel 的图标
```

样式表的图标属性

注:

- 1、每个图标属性都有一个对应的 QStyle::SP_开头的枚举相对应, 比如 cd-icon 与 QStyle::SP_DriveCDIcon 对应, dialog-cancel-icon 与 QStyle::SP_DialogCancelButton 对应
- 2、每个图标属性可根据其名称很容易明白其属性的含义, 因此不对属性作说明, 比如 messagebox-warning-icon 表示 QMessageBox 的警告图标

dvd-icon	dialog-apply-icon	messagebox-critical-icon	file-icon
floppy-icon	dialog-cancel-icon	messagebox-information-icon	file-link-icon
harddisk-icon	dialog-close-icon	messagebox-question-icon	filedialog-contentsvew-icon
trash-icon	dialog-discard-icon	messagebox-warning-icon	filedialog-detailedview-icon
home-icon	dialog-help-icon	titlebar-contexthelp-icon	filedialog-end-icon
desktop-icon	dialog-on-icon	titlebar-maximize-icon	filedialog-infoview-icon
computer-icon	dialog-ok-icon	titlebar-menu-icon	filedialog-listview-icon
cd-icon	dialog-open-icon	titlebar-minimize-icon	filedialog-new-directory-icon
network-icon	dialog-reset-icon	titlebar-normal-icon	filedialog-parent-directory-icon
forward-icon	dialog-save-icon	titlebar-shade-icon	filedialog-start-icon
backward-icon	dialog-yes-icon	titlebar-unshade-icon	dockwidget-close-icon
downarrow-icon	rightarrow-icon	directory-closed-icon	directory-link-icon
leftarrow-icon	uparrow-icon	directory-open-icon	directory-icon

25、Gradient

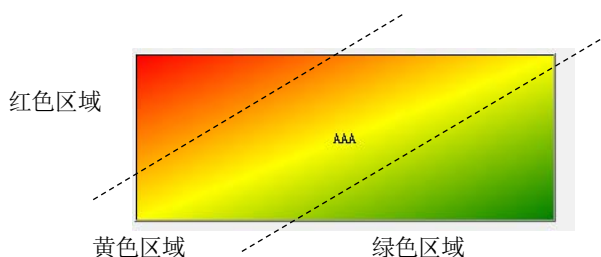
值: qlineargradient | qradialgradient | qconicalgradient

指定渐变填充, 其中 qlinearGradient 表示线性渐变, qradialgradient 表示径向渐变, qconicalgradient 表示圆锥渐变, 渐变的原理详见 Qt 2D 绘图章节, 下面列举几个示例

示例: 使用样式表设置线性渐变背景

```
#include<QtWidgets>
int main(int argc, char *argv[]){    QApplication aa(argc, argv);
    QWidget w;    QPushButton *pb1=new QPushButton("AAA",&w);    pb1->move(22,22);
    pb1->resize(333,133);    //为使效果比较明显, 把按钮设置得相对大一点
    aa.setStyleSheet(
        " QPushButton {\"background: qlineargradient(    //设置线性渐变背景
            x1:0, y1:0, x2:1, y2:1,\"                //渐变范围从左上角(0,0), 到右下角(1,1)
            //起点颜色为红色, 中点位置颜色为黄色, 终点颜色为绿色
            \"stop:0 red, stop: 0.5 yellow, stop:1 green)\"}";
    );
    w.resize(400,300);    w.show();    return aa.exec(); }
```

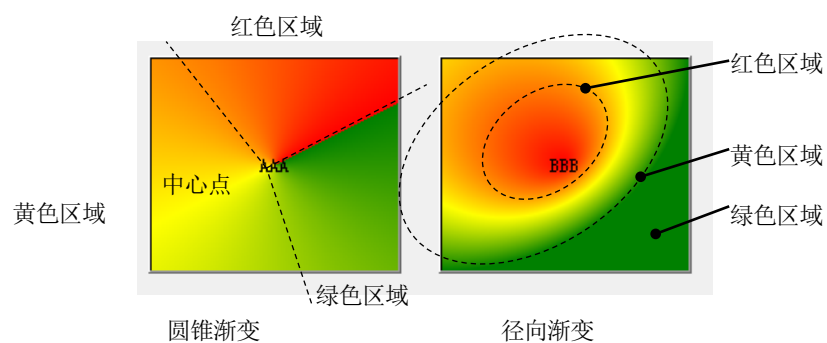
运行结果及说明



示例: 使用样式表设置圆锥渐变和径向渐变背景

```
#include<QtWidgets>
int main(int argc, char *argv[]){    QApplication aa(argc, argv);
    QWidget w;
    QPushButton *pb=new QPushButton("AAA",&w);    pb->move(22,22);    pb->resize(155,133);
    QPushButton *pb1=new QPushButton("BBB",&w);    pb1->move(200,22);pb1->resize(155,133);
    pb->setObjectName("AAA");    pb1->setObjectName("BBB");    //设置对象名
    aa.setStyleSheet(
        " QPushButton#AAA{"                                //设置按钮 AAA 的样式
            "background: qconicalgradient("                //圆锥渐变
            "cx:0.5, cy:0.5, angle:30,\"                  //指定中心点和角度
            "stop:0 red, stop:0.5 yellow,stop:1 green)\"}\"    //指定渐变颜色
        " QPushButton#BBB {"                                //设置按钮 BBB 的样式
            "background: qradialgradient("                //径向渐变
            "cx:0, cy:0, radius: 1,\"                      //中心点和半径(即渐变区域)
            "fx:0.5, fy:0.5,\"                            //焦点半径位置
            " stop:0 red,stop:0.5 yellow, stop:1 green)\"}\"    //指定渐变颜色
        );
    w.resize(400,300);    w.show();    return aa.exec(); }
```

运行结果及说明



13.10 设置各部件样式表的方法(综合示例)

一、基本规则

- 1、部件的渲染顺序：部件被视为彼此叠加的子控件的层次结构(树)，比如，QComboBox 绘制下拉子控件，然后是向下箭头子控件。因此绘制 QComboBox 样式表的顺序如下：
 - 渲染 QComboBox，即 QComboBox{....}
 - 渲染向下拉子控件，即 QComboBox::drop-down{...}
 - 渲染向下箭头，即 QComboBox::drop-arrow{...}QComboBox 的向下箭头子控件的父级是下拉子控件，下拉子控件的父级是 QComboBox 部件本身
- 2、子控件使用 subcontrol-position 和 subcontrol-origin 属性来确定在父级中的位置。
- 3、另外需要注意伪状态的情形，比如对于垂直滚动条将一直处于:vertical 伪状态，因此对垂直滚动条样式表的设置都需要指定:vertical 伪状态。

二、设置各部件样式表的方法

- 1、注：以下仅列了部件部分支持的属性、伪状态、子控件

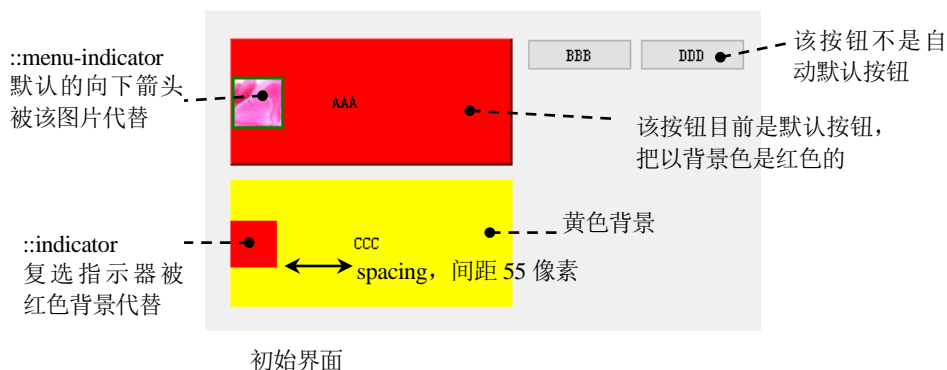
2、按钮相关

- 1)、**QPushButton** //支持盒子模型，
 - 支持:default, :flat, :checked 伪状态
 - ::menu-indicator 子控件设置拥有菜单的按钮的菜单指示器
 - :open 和:closed 伪状态可自定义可选中(checkable)按钮的外观
 - 注意：若仅在 QPushButton 上设置背景色，除非把 border 属性设置为某个值，否则背景可能不会被显示。
- 2)、**QRadioButton** //支持盒子模型，
::indicator 子控件可设置单选区域的样式，spacing 属性可设置文本和单选区域之间的间距
- 3)、**QCheckBox** //支持盒子模型
::indicator 子控件可设置复选区域的样式，spacing 属性可设置文本和复选区域之间的间距
- 4)、**QComboBox**
盒子模型可设置组合框周围的边框，::drop-down 子控件设置下拉按钮，::down-arrow 子控件设置下拉按钮中的箭头
- 5)、**QToolButton** //支持盒子模型，
 - ::menu-indicator 子控件设置拥有菜单的 QToolButton 的菜单指示器，默认情况下，指示器位于部件填充矩形的右下角。
 - 若 QToolButton 处于 QToolButton::MenuButtonPopup 模式，则::menu-button 子控件用于设置菜单按钮的样式，::menu-arrow 子控件用于设置::menu-button 子控件内的箭头，默认情况下，箭头位于::menu-button 子控件内容矩形的中心
 - ::up-arrow, ::down-arrow, ::left-arrow, ::right-arrow 用于设置显示箭头的 QToolButton
 - 注意：若仅在 QToolButton 上设置背景色，除非把 border 属性设置为某个值，否则背景可能不会被显示。
- 6)、**QSpinBox、QDateEdit、QDateTimeEdit、QDoubleSpinBox、QTimeEdit**
 - 可使用盒子模型设置微调按钮的边框样式
 - ::up-button 和::up-arrow 子控件分别用于设置向上按钮和向上箭头的样式
 - ::down-button 和::down-arrow 子控件分别用于设置向下按钮和向下箭头的样式
 - 默认情况下，向上按钮位于部件填充矩形的右上角，向下按钮位于部件填充矩形的右下角，向上箭头位于向上按钮的内容矩形的中心，向下箭头位于向下按钮内容矩形的中心

示例: QPushButton 和 QCheckBox

```
#include<QtWidgets>
int main(int argc, char *argv[]) {    QApplication aa(argc, argv);
    QWidget w;
    QPushButton *pb=new QPushButton("AAA",&w);    pb->move(20,20);    pb->resize(200,90);
    //为按钮 pb 添加菜单
    QMenu *pm=new QMenu("AAA");
    pm->addAction("111");    pm->addAction("222");    pb->setMenu(pm);
    QPushButton *pb1=new QPushButton("BBB",&w);    pb1->move(230,20);
    //把 pb 和 pb1 设置为自动默认按钮
    pb->setAutoDefault(true);    pb1->setAutoDefault(true);
    //注意: pbx 不是自动默认按钮
    QPushButton *pbx=new QPushButton("DDD",&w);    pbx->move(310,20);
    QCheckBox *pb2=new QCheckBox("CCC",&w);    pb2->move(20,120);    pb2->resize(200,90);
    aa.setStyleSheet(
        "QPushButton:default {background-color:red;}\" //默认按钮的背景色为红色
        "QPushButton:open {background-color:yellow;}\" //选中按钮时的背景色为黄色
        "QPushButton::menu-indicator {" //设置按钮的菜单指示器
            "border:2px solid green;" //绿色 2 个像素宽的实线边框边界线
            //加载图片作为菜单指示器,读者可加载符合实际需要的图片
            "background-image:url(F:/li.png);"
            "subcontrol-position:left center;" //设置子控件的位置
            "width:33px;height:33px;}\" //设置子控件的大小
        "QCheckBox{background-color:yellow;}\" //复选按钮的背景颜色为黄色
        "spacing:55px;}\" //设置文字和复选指示器之间的距离为 55 像素
    //设置复选指示器的背景颜色(红色)及大小
        "QCheckBox::indicator{background-color:red;width:33px;height:33;}\"
    //设置复选指示器处于选中状态时的背景色为绿色
        "QCheckBox::indicator:checked{ background-color:green;}\"
    );
    w.resize(400,333);    w.show();    return aa.exec(); }
```

运行结果及说明



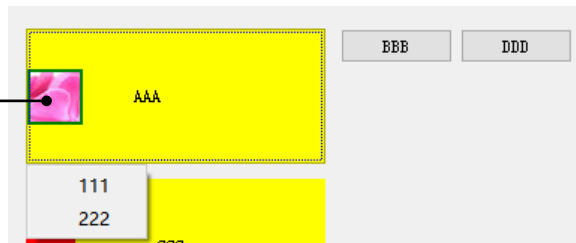


把焦点移至 BBB 时，BBB 成为默认按钮，此时使用红色背景填充 BBB

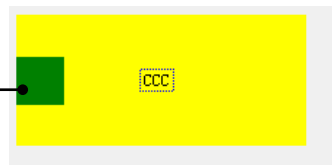


把焦点移至 DDD 时，此时没有默认按钮，所有按钮都是默认背景色

按下该图标，会弹出如下所示的菜单，此时按钮 AAA 被选中，使用黄色背景填充



::indicator:checked
当复选按钮处于选中状态时，复选指示器由红色背景变为绿色背景



示例: QSpinBox

```
#include<QtWidgets>
int main(int argc, char *argv[]){    QApplication aa(argc, argv);
    QWidget w;
    QSpinBox *pb3=new QSpinBox(&w);    pb3->move(20, 20);    pb3->resize(200, 90);
    aa.setStyleSheet(
        "QSpinBox::up-button {"        //设置向上按钮
            "subcontrol-position:left center;"        //子控件位于左侧
            "background-color:red;"                //背景色为红色
            "width:50px;height:90px;"            //子控件大小
        }
        "QSpinBox::up-arrow {"        //设置向上箭头的背景色(绿色)，大小，位置
            "background-color:green;    width:33px;height:33px;"
            "subcontrol-position:right bottom;"
        }
        "QSpinBox::down-button {"background-color:red;width:50px;height:90px;}"
        "QSpinBox::down-arrow {"background-color:green;width:33px;height:33px;}"
    );
    w.resize(400, 333);    w.show();    return aa.exec();}
```

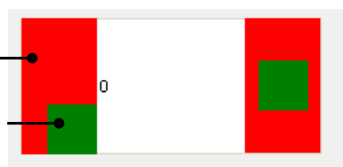
运行结果及说明

::up-button

向上按钮(红色背景)

::up-arrow

向上箭头(绿色背景)



本示例把 QSpinBox 的向上/向下按钮设置于 QSpinBox 的左/右两侧

3、对话框及文本编辑相关的部件

- 7)、**QDialog** //仅支持 background, background-clip 和 background-origin 属性
- 8)、**QDialogButtonBox** //可使用 button-layout 属性更改按钮的布局
- 9)、**QMessageBox** messagebox-text-interaction-flags 属性可设置消息框中与文本的交互方式
- 10)、**QTextEdit** //支持盒子模型,
selection-color 和 selection-background-color 属性分别设置所选项的背景和前景色
- 11)、**QLabel** //支持盒子模型,
不支持: hover 伪状态, 从 Qt4.3 开始, 在 QLabel 上设置样式表, 会自动将 QFrame::frameStyle 属性设置为 QFrame::StyledPanel 枚举值
- 12)、**QLineEdit** //支持盒子模型,
 - selection-color 和 selection-background-color 属性分别设置所选项的背景和前景色
 - lineedit-password-character 属性可设置密码字符
 - lineedit-password-mask-delay 可设置显示密码字符的延迟时间

4、主窗口、可停靠窗口、状态栏、工具栏、菜单

- 13)、**QMainWindow** //当使用 QDockWidget 时, QMainWindow 中的分隔符使用::separator 子控件设置
- 14)、**QDockWidget**
 - 当停靠时, 支持标题栏和标题栏按钮的样式
 - 可使用 border 属性设置可停靠窗口的边框样式
 - ::title 子控件可自定义标题栏
 - close 和 float 按钮分别使用: colse-button 和 float-button 子控件定位于: title 子控件
 - 当标题栏垂直时, 需设置: vertical 伪状态。
 - 根据 QDockWidget::DockWidgetFeature 设置伪状态: colsable, :floatable 和: movable
 - 注意: 当 QDockWidget 未停靠时, 样式表无效, 因为此时 Qt 使用本机顶级窗口的样式。
- 15)、**QStatusBar** //仅支持 background 属性, :item 子控件设置单个项,
- 16)、**QToolBar** //支持盒子模型,
 - :top, :left, :right, :bottom 伪状态取决于工具栏分组的区域
 - :first, :last, :middle, :only-one 伪状态用于指示工具栏在行分组中的位置(其原理与 QStyleOptionToolBar::positionWithinLine 成员变量相同)
 - :separator 子控件设置分隔符
 - :handle 子控件用于设置手柄的样式
- 17)、**QMenu** //支持盒子模型,
 - :item 子控件可设置单个的项, 该子控件除了支持通常的伪状态外, 还支持: selected, :default, :exclusive, :non-exclusive 伪状态
 - :separator 子控件设置分隔符
 - :right-arrow 和: left-arrow 子控件可设置带有子菜单的菜单项的箭头
 - :scroller 子控件设置滚动条
 - :tearoff 子控件设置可分离指示器
 - :indicator 子控件设置菜单项的可复选指示器
- 18)、**QMenuBar** //支持盒子模型,
 - :item 子控件可设置单个的项
 - spacing 属性指定菜单项之间的间距
 - 注意: 在 Qt/Mac 上运行时, 菜单栏通常被嵌入到系统范围的菜单栏中, 此时, 样式表将不起作用。

示例: QToolBar、QMenu、QMenuBar

```
#include<QtWidgets>
int main(int argc, char *argv[]) {    QApplication aa(argc, argv);
    QMainWindow w;                    //使用主窗口
    //创建并向主窗口添加 QToolBar
    QToolBar *pt=new QToolBar(&w); pt->addAction("AAA");
```

```

QToolBar *pt1=new QToolBar (&w);      pt1->addAction("BBB");
QToolBar *pt2=new QToolBar (&w);
pt2->addAction("CCC");      pt2->addSeparator();      pt2->addAction("DDD");
w.addToolBar(pt);      w.addToolBar(pt1);      w.addToolBar(pt2);

//创建并向主窗口添加菜单
QMenuBar *pb=w.menuBar(); QMenu *pm=new QMenu("111"); QMenu *pml=new QMenu("222");
pm->addMenu(pml);      pml->addAction("333");      pml->addAction("444");
pm->addAction("555");      pm->addAction("666");
pb->addMenu(pm);      pb->addAction("777");      pb->addAction("888");
aa.setStyleSheet(
    //1、工具栏位于顶部或左侧时的背景色为红色
    "QToolBar:top,QToolBar:left { background-color:red;}"

    //2、工具栏位于行分组的中间位置时的背景色为绿色
    "QToolBar:middle {background-color:green;}"

    //3、工具栏位于顶部或左侧时，其手柄背景色为紫色，宽度和高度各为 22 像素
    //注：在顶部时需设置宽度，在左侧时需设置高度，否则手柄不可见
    "QToolBar::handle:top,QToolBar::handle:left {"
        "background-color:rgb(111,1,111);width:22px;height:22px;}"

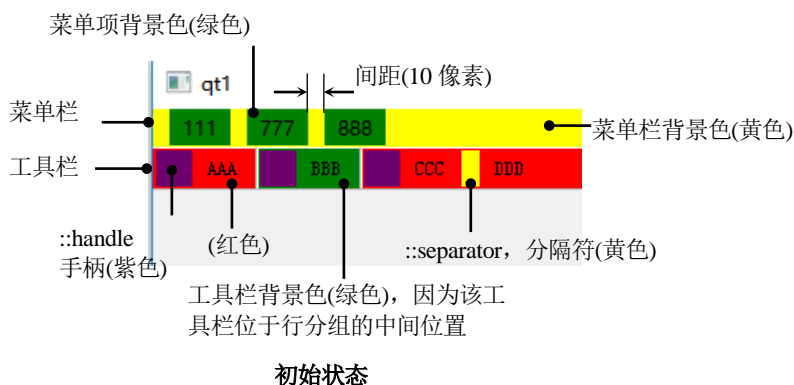
    //4、工具栏分隔符的背景色为黄色，在水平方向时宽度为 11，在垂直方向时高度为 11
    "QToolBar::separator{background-color:yellow; width:11px;height:11;}"

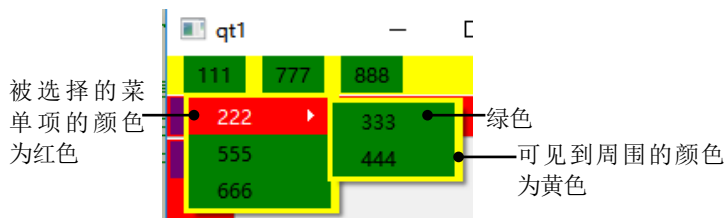
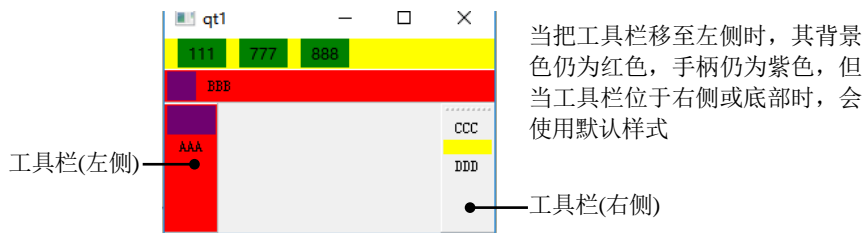
    //5、以上为对菜单和菜单栏的设置(较简单)
    "QMenuBar {background-color:yellow;spacing:10;}"
    "QMenuBar::item{background-color:green;}"
    "QMenu{background-color:yellow;}"
    "QMenu::item{background-color:green;}"
    "QMenu::item:selected{background-color:red;}"

);
w.resize(400,333);      w.show();      return aa.exec();

```

运行结果及说明





5、滚动相关

19)、**QAbstractScrollArea** //支持盒子模型，background-attachment 属性(滚动或固定)可被使用

20)、**QScrollBar** //支持盒子模型，

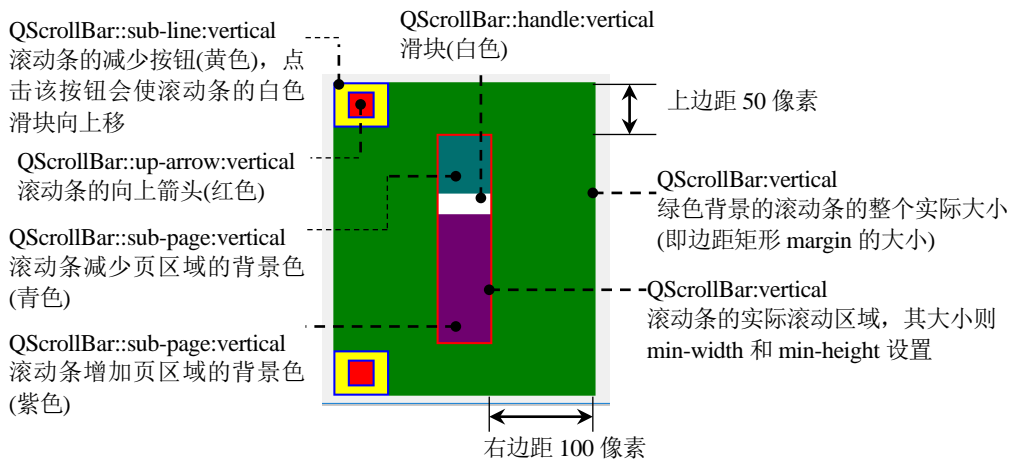
- 内容矩形被认为是滑块移动的凹槽。
- :horizontal 和:vertical 伪状态用于确定滚动条的方向
- width 和 height 属性设置 QScrollBar 的范围。
- ::handle 子控件用于设置滑块的样式
- min-width 和 min-height 属性用于限制滑块的大小
- ::add-line 子控件用于设置增加一行的按钮(比如向下或向右按钮)
- ::sub-line 子控件用于设置减少一行的按钮(比如向上或向左按钮)
- ::right-arrow 或::down-arrow 子控件用于设置向右或向下箭头，默认情况下，这两个子控件位于 add-line 子控件内容矩形的中心
- ::left-arrow 或::up-arrow 子控件用于设置向左或向上箭头，默认情况下，这两个子控件位于 sub-line 子控件内容矩形的中心
- ::sub-page 子控件用于设置减少页的滑块区域样式
- ::add-page 子控件用于设置增加页的滑块区域样式

21)、**QSlider** //支持盒子模型

- 对于水平滑块，必须提供 min-width 和 height 属性
- 对于垂直滑块，必须提供 min-height 和 width 属性
- ::groove 子控件用于设置滑块的凹槽，默认情况下，凹槽位于滑块的内容矩形中
- ::handle 子控件设置滑块的手柄，该子控件在::groove 子控件的内容矩形中移动

示例：滚动条(QScrollBar)

为了讲清楚滚动条的原理及理解滚动条的各组成部分，本示例将绘制如下图所示非传统形式的滚动条



```
#include<QtWidgets>
int main(int argc, char *argv[]){    QApplication aa(argc,argv);
    QWidget w;        QScrollBar *ps=new QScrollBar(&w);
    ps->setOrientation(Qt::Vertical);    ps->move(22,22);
    /*本示例为垂直滚动条，因此滚动条会一直处于:vertical 伪状态，所以对滚动条的每个子控件的设置
       都需要指定该状态*/
    aa.setStyleSheet(
        //1、设置整个滚动条
        "QScrollBar:vertical {border: 2px solid red;margin:50 100 50 100;}
        "background-clip:margin;    background: green; "
        //注意: width 和 height 属性仅用于子控件，因此此处使用 min-width 和 min-height
        "min-width: 50;        min-height:200;}"

        //2、设置滚动条的滑块
        "QScrollBar::handle:vertical {background: white;    min-height: 20px;}"

        //3、设置滚动条的向上按钮
        "QScrollBar::add-line:vertical {"
        "border: 2px solid blue;    background: yellow;"
        "height: 40px;    width:50px;"
        "subcontrol-position: bottom left;    subcontrol-origin: margin;}"

        //4、设置滚动条的向下按钮
        "QScrollBar::sub-line:vertical {"
        "border: 2px solid blue;    background: yellow;"
        "height: 40px;    width:50px;"
        "subcontrol-position: top left;    subcontrol-origin: margin;}"

        //5、设置向上和向下箭头
        "QScrollBar::up-arrow:vertical, QScrollBar::down-arrow:vertical {"
        "border: 2px solid blue;    width: 22px; height: 22px; background: red;}"

        //6、设置滚动条的增加页区域
        "QScrollBar::add-page:vertical{background: rgb(111,1,111);}"

        //7、设置滚动条的减少页区域
        "QScrollBar::sub-page:vertical {background: rgb(1,111,111);}"
    );
    w.resize(400,333);    w.show();    return aa.exec(); }
```


6、选项卡相关及 QToolBox

22)、QTabBar

- `::tab` 子控件可设置单个选项卡的样式，
- `::close-button` 子控件可设置关闭按钮的样式
- 选项卡支持:only-one, :first, :last, :middle, :previous-selected, :next-selected, selected 伪状态
- :top, :left, :right, :bottom 伪状态取决于选项卡的方向
- 被选中状态的重叠选项卡，使用负边距或使用 absolute 位置方案进行设置。
- `::tear` 子控件用于设置 QTabBar 的可分离指示器
- QTabBar 的滚动条使用两个 QToolButton，可使用选择器"QTabBar QToolButton"进行设置
- `::scroller` 子控件可用于设置 QTabBar 的滚动按钮的宽度。
- 若要更改 QTabWidget 中 QTabBar 的位置，请使用 `::tab-bar` 子控件，并设置其位置

23)、QTabWidget

- `::pane` 子控件用于设置 QTabWidget 部件的边框
- `::left-corner` 和 `::right-corner` 用于分别设置左角落和右角落
- `::tab-bar` 子控件可控制选项卡栏的位置
- 若要把 QTabBar 放置于中间，则需设置 `::tab-bar` 子控件的 subcontrol-position 属性
- :top, :left, :right, :bottom 伪状态取决于选项卡的方向

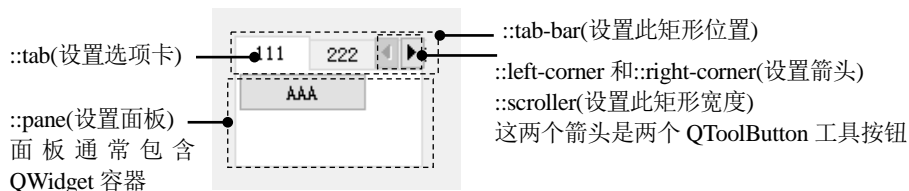
24)、QToolBox

//支持盒子模型，

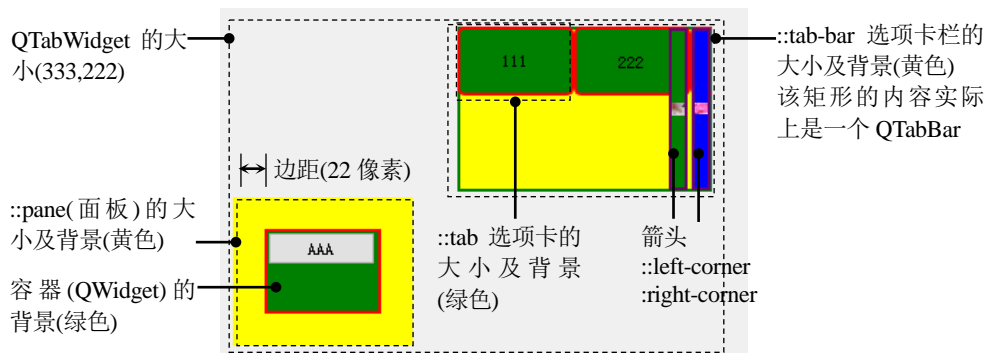
`::tab` 子控件用于设置各选项卡的样式，选项卡支持:only-one, :first, :last, :middle, :previous-selected, :next-selected, :selected 伪状态

示例：设置 QTabWidget 的样式

下图为 QTabWidget 各组成部分



下图为本示例需要设置 QTabWidget 的界面



```

#include<QtWidgets>
int main(int argc, char *argv[]) {    QApplication aa(argc, argv);
    QWidget w;
    //设置 QTabWidget 的内容
    QTabWidget *pt=new QTabWidget(&w);          pt->move(22, 22);
    QWidget *pw1=new QWidget;          QWidget *pw2=new QWidget;
    QWidget *pw3=new QWidget;          QWidget *pw4=new QWidget;
    pw1->setObjectName("pw1");          //设置容器 pw1 的对象名
    QPushButton *pb1=new QPushButton("AAA", pw1);
    QPushButton *pb2=new QPushButton("BBB", pw2);
    QPushButton *pb3=new QPushButton("CCC", pw3);
    pt->addTab(pw1, "111");          pt->addTab(pw2, "222");
    pt->addTab(pw3, "333");          pt->addTab(pw4, "444");
    pt->setUsesScrollButtons(true);          //1、使 QTabWidget 显示滚动条
    pt->resize(333, 222);          //2、设置 QTabWidget 的大小
    qDebug() << pt->findChildren<QObject*>(); //读者可使用该函数查看 QTabWidget 的个组件
    aa.setStyleSheet(
        "QWidget#pw1 {background: green;} " //3、设置容器 pw1 的背景色为绿色
        "QTabWidget::pane { "          //4、设置 QTabWidget 的面板样式
            "border: 2px solid red;      margin: 22px 22px 22px 22px;"
            "background: yellow;      background-clip: margin; background-origin: margin;"
            "width: 77;      height: 55;" //5、设置面板的大小
            "subcontrol-position: bottom left;} "          //6、面板位于左下角
    //7、QTabWidget 选项卡栏的背景色和边框需使用 QTabBar 设置
        "QTabWidget QTabBar { " //注意选择器的语法，表示设置 QTabWidget 的子部件 QTabBar
            "background: yellow;      border: 2px solid green;      min-height: 111;} "
    //8、使用::tab-bar 子控件设置选项卡栏的位置和宽度
        "QTabWidget::tab-bar {width: 177; subcontrol-position: right;} " //选项卡栏位于右侧
    //9、使用::tab 子控件设置选项卡的背景和大小
        " QTabBar::tab { background: green;      border: 2px solid red; border-radius: 5;"
            "width: 77;      height: 44;} " //选项卡的大小
    //10、使用::scroller 设置滚动区域的宽度
        "QTabBar::scroller {width: 15;} "
    //11、设置右箭头
        "QTabBar QToolButton::right-arrow { "
            "background: blue;          background-clip: margin;"
            "image: url(F:/1i.png);      border: 2px solid rgb(111, 1, 111);} "
    //12、设置左箭头
        "QTabBar QToolButton::left-arrow { "
            "background: green; border: 2px solid rgb(111, 1, 111);      image: url(F:/2i.png);} "
        );
    w.resize(400, 333);    w.show();    return aa.exec(); }

```

7、其他部件

- 25)、**QFrame** //支持盒子模型，
- 26)、**QWidget** //仅支持 background, background-clip, background-origin 属性，
- 27)、**QSizeGrip** //支持 width, height, image 属性
- 28)、**QToolTip** //支持盒子模型，opacity 属性用于设置工具提示的不透明度
- 29)、**QGroupBox** //支持盒子模型，
 - 可使用::title 子控件设置标题样式。
 - 若 QGroupBox 是可选中的(checkable)，则复选指示符使用::indicator 子控件设置。
 - spacing 属性可用于控件文本和复选指示符之间的距离

- 30)、**QProgressBar** //支持盒子模型，
::chunk 子控件设置进度条的块，块被显示在内容矩形上
text-align 属性可定位进度条的文本
:indeterminate 伪状态设置进度条的不确定状态
- 31)、**QSplitter** //支持盒子模型，::handle 子控件可设置分隔符手柄的样式

8、模型/视图相关

- 32)、**QColumnView** //::left-arrow 和::right-arrow 子控件可设置箭头指示符
- 33)、**QTreeView** //支持盒子模型，
- alternate-background-color 属性可设置交替行颜色
 - selection-color 和 selection-background-color 属性分别设置所选项的背景和前景色
 - show-decoration-selected 属性可控件选择的行为(选择整行还是仅文本)
 - ::branch 子控件用于设置 QTreeView 的分支样式，::branch 子控件支持:open, :closed, :has-sibling, :has-children 伪状态
 - ::item 子控件可更细粒度的控制 QListView 中的项
- 34)、**QTableView、QTableWidget** //支持盒子模型，
- alternate-background-color 属性可设置交替行颜色
 - selection-color 和 selection-background-color 属性分别设置所选项的背景和前景色
 - QTableView 角落部件被视为 QAbstractButton，并可使用"QTableView **QTableCornerButton::section**"选择器设置其样式
 - gridline-color 属性可设置网格的颜色
 - 注意：若仅在 **QTableCornerButton** 上设置背景色，除非把 border 属性设置为某个值，否则背景可能不会被显示。
- 35)、**QHeaderView** //支持盒子模型，
- 标头的段使用::section 子控件设置，该子控件支持:middle, :first, :last, :only-one, :next-selected, :previous-selected, :selected, :checked 伪状态
 - 排序指示器可使用::up-arrow 和::down-arrow 子控件设置
- 36)、**QListView、QListWidget** //支持盒子模型，
- alternate-background-color 可设置交替行颜色，
 - selection-color 和 selection-background-color 属性分别设置所选项的背景和前景色
 - show-decoration-selected 属性可控件选择的行为(选择整行还是仅文本)
 - ::item 子控件可更细粒度的控制 QListView 中的项

13.11 样式表的其他规则

一、层叠和继承

- 1、样式表可以在 QApplication、父部件、子部件上设置，在这种情况下，样式表会合并来自祖先的设置，当产生冲突时，无论冲突的规则如何，部件自己的样式表始终优于任何继承来的样式表，同理，父部件的样式表优于其祖父母的样式表。需要注意的是，部件之间需存在父子关系，也就是使用 QWidget::setParent()函数或其他方式设置了父子关系的部件。
- 2、默认情况下，Qt 样式表并不能自动继承其父部件的字体和颜色属性，若要使其继承父部件上设置的字体和颜色属性，可使用如下所示的语法

```
app->setStyleSheet("QGroupBox, QGroupBox *{font:33px;}"); /*之前有空格
```

示例：样式表的层叠

```
#include<QtWidgets>
int main(int argc, char *argv[]) {    QApplication aa(argc, argv);
    QWidget w;
    //w 是 pa 和 pal 的父部件，pa 是 pb1 的父部件，pb1 是 pb2 的父部件
    QLabel *pa=new QLabel("AAA", &w);      pa->move(22, 22);    pa->resize(255, 111);
    QLabel *pal=new QLabel("11111", &w);    pal->move(299, 22);
    QPushButton *pb1=new QPushButton("BBB", pa);  pb1->resize(199, 77); pb1->move(55, 33);
    QCheckBox *pb2=new QCheckBox("CCC", pb1);    pb2->move(22, 10);
    pa->setStyleSheet("*{background:green;font:22px;}");
    pb1->setStyleSheet("QPushButton, QPushButton *{background:yellow;}");
    aa.setStyleSheet("QWidget, QWidget *{background:red;}");
    w.resize(400, 333);    w.show();    return aa.exec(); }
```

运行结果及说明



- 1、标签 AAA 自己设置的样式表优于其父 QApplication 设置的样式表，虽然 QApplication 设置的样式表更特殊，但 AAA 的背景色仍优于 QApplication，为绿色
- 2、按钮 BBB 自己设置的样式表优于其父部件(标签 AAA)设置的样式表，因此其背景色为黄色，字体继承父部件，其大小为 22 像素。
- 3、复选按钮 CCC 未设置样式表，所以其背景色继承自父部件(按钮 BBB)，其颜色为黄色，字体继承自祖先部件(标签 AAA)，其大小为 22 像素。
- 4、标签 11111 与其他部件不存在父子关系，因此使用继承自 QApplication 设置的背景色红色。

二、名称空间及使用 QObject 属性

- 1、为自定义部件设置样式表时，只需使用自定义的类名即可，但该类需要有 `Q_OBJECT` 宏的声明，比如

```
class B:public QPushButton{Q_OBJECT    //需声明该宏
public:B(QString s,QWidget *p1=0):QPushButton(s,p1){ }
.....};
```

设置样式表的语法如下：

```
app->setStyleSheet("B{...}"); //直接使用类名 B
```

- 2、当自定义的类位于名称空间中时，只需把指定类名的 “:” 符号替换为 “--” (两个短横线) 以避免与样式表子控件的冲突，比如

```
namespace N{
class B:public QPushButton{Q_OBJECT
public:  B(QString s,QWidget *p1=0):QPushButton(s,p1){ }
.....}; }
```

设置样式表的语法如下：

```
app->setStyleSheet("N--B{...}"); //使用--代替作用域解析运算符::
```

- 3、从 Qt4.3 开始，任何能设计的 `Q_PROPERTY`(这是设置属性的宏)都可使用 “qproperty-属性名称” 的语法形式通过样式表来设置该属性。比如

```
aa.setStyleSheet("QPushButton{qproperty-autoDefault:1;}") //设置按钮为自动默认按钮
```

三、冲突解决

- 1、当多个样式规则对相同属性指定了不同值而产生冲突时，则优先选择更特殊的选择器，比如

```
QPushButton#AAA{color:red};    //此规则更具体，选择此规则
QPushButton{color:green};
```

- 2、若两个选择器的特殊性相同，则后出现的优先于前出现的规则，比如

```
QPushButton:enabled{color:red};
QPushButton:hover{color:green}; //选择此规则
```

- 3、当两个选择器具有继承关系时，仍然是后出现的规则优先于前出现的规则，因为对于样式表，选择器具有相同的特殊性，比如

```
QPushButton {color: red}
QAbstractButton {color: green}    //使用此规则
```

- 4、Qt 样式表规则的特殊性遵守 CSS2 规范。

作者：黄邦勇帅(原名：黄勇)

2018-8-8

本文作者：黄邦勇帅(原名：黄勇)

本文要求读者已经非常熟悉 C++ 语法。若读者不熟悉 C++ 语法，推荐参阅《C++ 语法详解》(作者：黄勇)一书，电子工业出版社出版。

本文主要讲解了 Qt 的输入输出，本文列举了详细的示例进行说明，同时本文也是非常方便、快捷的编写 Qt 程序的查阅资料，可方便的查阅到相关内容的原理，以及怎样使用该内容。本文内容由浅入深，易学易懂。

本文使用的是 windows 10 操作系统，Qt 版本为 Qt5.10.1，Qt Creator 的版本为 Qt Creator 4.5.1 本文内容完全属于个人见解与参考文献的作者无关，限于水平有限，其中难免有误解之处，望指出更正。

声明：禁止抄袭、复印、转载本文，本文作者拥有完全版权。

主要参考文献：

- 1、C++语法详解 黄勇 编著 电子工业出版社 2017 年 7 月
- 2、Qt Creator 快速入门(第 3 版) 霍亚飞 编著 北京航空航天大学出版社 2017 年 1 月
- 3、C++ GUI Qt4 编程(第 2 版) [加拿大] Jasmin Blanchette [英] Mark Summerfield 著 闫锋欣 曾泉人 张志强 译 电子工业出版社 2008 年 8 月
- 4、Qt 5 开发及实例 陆文周 主编 电子工业出版社 2014 年 1 月

第 14 章 Qt 输入输出目录

[14.1 与 Qt 输入输出有关的类简介](#)

[14.2 QDataStream 类（数据流）](#)

[14.2.1 QDataStream 类读写对象原理](#)

[14.2.2 QDataStream 类中的函数](#)

[14.3 QTextStream 类（文本流）](#)

[14.3.1 字符编码基础知识](#)

[14.3.2 QTextStream 基本规则](#)

[14.3.3 QTextStream 类中的函数](#)

[14.3.4 以其他方式设置 QTextStream 流的格式](#)

[14.4 QFile 类（文件）](#)

[14.4.1 QFile 基本原理](#)

[14.4.2 QFile 类中的函数](#)

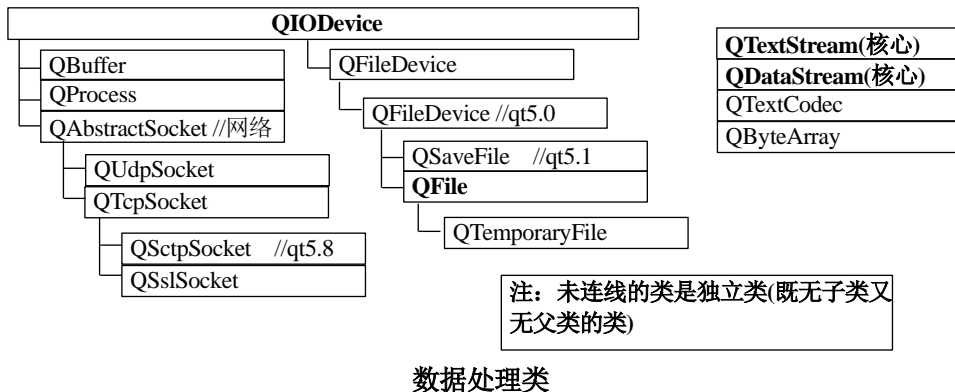
[14.5 Qt 资源简介](#)

第 14 部分 Qt 输入输出

注意：本程序都假设读者在 pro 文件中已添加了正确的 `QT+=widgets` 语句，文中不再重复累述添加此语句。

本文注重讲解原理，因此使用的是手写的 Qt 程序。

本章讲解的类及继承关系如下图所示(本文仅讲解图中粗体所示的类)



14.1 与 Qt 输入/输出有关的类简介

1、**QTextStream** 类(文本流)和 **QDataStream** 类(数据流)Qt 输入输出的两个核心类，其作用分别如下：

①、**QTextStream** 类：用于对数据进行文本格式的读/写操作，可在 **QString**、**QIODevice** 或 **QByteArray** 上运行，比如把数据输出到 **QString**、**QIODevice** 或 **QByteArray** 对象上，或进行相反的操作。

②、**QDataStream** 类：用于对数据进行二进制格式的读/写操作，**QDataStream** 只可在 **QIODevice** 或 **QByteArray** 上运行，因为 **QString** 只存放字符数据。

2、**QIODevice** 类是 Qt 中所有 I/O 设备的基础接口类(这是一个抽象类)，也就是说 **QIODevice** 及其子类描述的是 I/O 设备，该类为支持读/写数据块的设备提供了通用实现和抽象接口，比如 **QFile**、**QBuffer**、**QTcpSocket** 等。

3、**QIODevice** 把设备分为两类：随机存储设备和顺序存储设备

①、随机存储设备：可定位到任意位置(使用 `seek()` 函数)，随机存储设备有 **QFile**，**QTemporaryFile**，**QBuffer**

②、顺序存储设备：不支持任意的位置存储，顺序存储设备有 **QProcess**、**QTcpSocket**、**QUdpSocket** 和 **QSslSocket**

4、**QTextCodec** 类负责 Unicode 与各字符编码之间的转换。

- 5、QProcess 类与进程相关，QTcpSocket、QUdpSocket 等类与网络数据传输有关，这些类不属于本章所讲内容，请参阅进程和网络编程的相关文章。
- 6、QBuffer 类为 QByteArray 提供了一个 QIODevice 接口，以允许使用 QIODevice 接口来访问 QByteArray。默认情况下，创建一个 QBuffer 时，会自动在内部创建一个 QByteArray 缓冲区。
- 7、以下枚举将在本文整篇文章中使用到

QIODevice::OpenModeFlag 枚举		
标志：QIODevice::OpenMode		
作用：描述文本流的状态		
成员	值	说明
QIODevice::NotOpen	0x0000	设备未打开
QIODevice::ReadOnly	0x0001	只读模式
QIODevice::WriteOnly	0x0002	只写模式，此模式会截断，比如打开文件时会删除之前的内容
QIODevice::ReadWrite		同 ReadOnly WriteOnly，即设备即可读又可写
QIODevice::Append	0x0004	设备以追加模式打开，以便把数据写入文件末尾
QIODevice::Truncate	0x0008	若可能，设备在打开之前被截断
QIODevice::Text	0x0010	读取时，行尾符被转换为'\n'，写入时行尾符被转换为本地编码，比如 windows 会被转换为'\r\n'
QIODevice::Unbuffered	0x0020	绕过设备中的缓冲区

14.2 QDataStream 类(数据流)

一、QDataStream 类读/写对象原理

- 1、QDataStream 类负责以二进制方式读/写程序中的对象，输入源和输出目标可以是 QIODevice、QByteArray 对象。
- 2、字节序：即多字节数据(即大于一个字节的的数据)在内存中的存储顺序，有如下两种方式
 - Little-Endian(LE, 小端)：即低位字节存储在低地址端，高位字节存储在高地址端
 - Big-Endian(BE, 大端)：即高位字节存储在低地址端，低位字节存储在高地址端。这是 QDataStream 的默认字节序。
 - 比如对于整数 0x2345，若按 big-endian(大端)顺序存储，则按 0x23、0x45 的顺序存储，若按 little-endian(小端)顺序存储，则以 0x45、0x23 的顺序存储。
- 3、对象的存储和传输：若直接把一个对象保存在文件(或其他地方)上是没有意义的，因为对象中通常包含指向其他对象的指针，指针所指对象在下次运行时其内存地址很可能并不相同，因此在保存对象时，保存本次运行时指针的值就毫无意义，对此，需要采取必要的手段来解决保存对象的问题。对象的传输同样会遇到这种问题(比如在客户端和服务端传递对象时，在进程间传递对象时)，解决这一问题的方法就是序列化(serializable)或称为串行化
- 4、序列化(serializable)：是把对象状态转换为可保存或可传输的形式过程，与其对应的是反序列化，序列化和反序列化保证了数据易于存储和传输。数据通常以二进制序列的形式进行传输，因此序列化通常是把对象转换为字节序列的过程，其相反过程称为反序列化。
- 6、QDataStream 是编码信息的二进制流，它完全独立于主机的操作系统、CPU 和字节序，比如由 Windows 编写的流数据可以由运行 Solaris 的 Sun SPARC 读取。还可使用数据流来读/写原始的未编码的二进制数据
- 7、QDataStream 实现了基本的 C++ 数据类型的序列化，比如 char, short, int, char* 等。更复杂的数据类型的序列化是通过分解原始单元来完成的。
- 8、数据流与 QIODevice 紧密合作，QIODevice 表示一个能读/写数据的 I/O 设备，其中 QFile 是常见的 I/O 设备，
- 9、写入到数据流的每一项都是以预定义的二进制格式编写的，该格式根据写入项的类型而有所不同。
- 10、QDataStream 支持的 Qt 类型有 QBrush、QColor、QDateTime、QFont、QPixmap、QString、QVariant 等类型，还包括容器类型，比如 QList、QVector、QSet、QMap 等，支持的 Qt 类型的完整列表可参阅帮助文档 Serializing Qt Data Types
- 11、对于整数，建议始终转换为 Qt 整数类型(比如 qint32 等)进入写入，并将其读入为相同的 Qt 整数类型，这样可以确保获取确定的大小的整数，以避免编译器和平台差异的影响(注：C++ 语法只规定了 int, short 等类型的最小长度，未规定最大长度)
- 12、使用 QDataStream 读/写二进制数据的步骤如下(以读/写到 QFile 为例)：
 - ①、使用 QDataStream 可方便的使用 >> 和 << 运算符对数据进行读写操作。
 - ②、使用 QDataStream 读取文件步骤相对来说要多一些，需要如下步骤(详见示例)

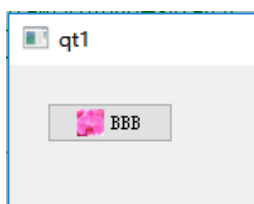
- 创建一个 QFile 对象，
- 再打开文件，
- 然后还需要创建一个 QDataStream 对象并把 QFile 对象绑定到该对象上，
- 然后才能使用>>和<<运算符进行读取操作

示例：使用 QDataStream 读/写二进制数据

以下示例读/写数据位于同一程序，实际应用时，读/写数据不一定在同一程序中。

```
#include<QtWidgets>
class B{public:int a;};          //自定义类
int main(int argc, char *argv[]){    QApplication aa(argc,argv);
    QWidget w;          QPushButton *pb=new QPushButton("AAA",&w);
    QFile f("F:/1.txt");    f.open(QIODevice::WriteOnly); //①、以只写方式打开文本 1.txt
    QIcon i("F:/li.png");    QPoint p(22,22);          //准备读/写的两个对象 i 和 p
    B mb;    mb.a=1;          //自定义类型的对象
    QDataStream out(&f);          //②、创建 QDataStream 对象并与 QFile 对象关联
    out <<i<<QString("BBB")<<p; //③、把对象 i, 字符串 BBB 和对象 p 存入文件 1.txt 中，
                                //读者可在 F 盘下找到 1.txt 文件以查看文件中的内容。
    //out<<mb;          //错误，读/写自定义类型需重载>>和<<运算符(其方法详见后文)
    f.close();          //关闭文件
    f.open(QIODevice::ReadOnly);    //④、以只读方式重读打开文本 1.txt，以读取其内容
    QIcon il;    QString s;    QPoint pl;    //这些对象用于存储从文件读入的对象。
    QDataStream in(&f);    //⑤、创建 QDataStream 对象并与 QFile 对象关联
    in>>il>>s>>pl;    //⑥、把文件 1.txt 的内容读出并存储到 il、s 和 pl 中，
                                //注意，读取数据类型的顺序应与写入时的一致
    //使用从文件读取到的数据设置按钮的图标，文本和位置
    pb->setIcon(il);    pb->setText(s);    pb->move(pl);
    f.close();    w.resize(400,333);    w.show();    return aa.exec(); }
```

运行结果及说明



由图可见，按钮的图标、文本和位置被成功设置

13、自定义类型的输入/输出: 使用 QDataStream 输入自定义类型时，需要重载<<和>>运算符，以使 QDataStream 支持新的数据类型，下面以示例进行说明。对于 Qt 类型是否支持类似的运算符，可查看该类帮助文档中的 Related Non-Members(相关的非成员)部分的函数

示例：使用 QDataStream 输入/输出自定义类型对象(本示例的运行结果及上一示例相同)

```
#include<QtWidgets>
class B{public: QIcon i;    QString s;    QPoint p; }; //①、自定义类型，保存图标，文本和位置
//②、重载<<和>>运算符(注意，是全局重载的)
QDataStream &operator<<(QDataStream &out, const B &mb){
```

```

        out<<mb.i<<mb.s<<mb.p;        return out; }        //内部代码比较简单
QDataStream &operator>>(QDataStream &in, B &mb) {in>>mb.i>>mb.s>>mb.p;    return in;    }

int main(int argc, char *argv[]) {    QApplication aa(argc,argv);
    QWidget w;    QPushButton *pb=new QPushButton("AAA",&w);
//③、创建并初始化自定义类型对象
    B mb;    mb.i=QIcon("F:/1i.png");    mb.s="BBB";    mb.p=QPoint(22,22);
//④、创建 QFile 对象并打开文件
    QFile f("F:/1.txt");    f.open(QIODevice::WriteOnly);
    QDataStream out(&f);    //⑤、创建 QDataStream 对象并与 QFile 对象关联
    out <<mb;    //⑥、把对象 mb 写入文件 1.txt
    f.close();    //⑦、关闭文件
//⑧、读取文件内容的代码与写入类似
    f.open(QIODevice::ReadOnly);
    B mb1;    QDataStream in(&f);    in>>mb1;
//⑨、使用从文件 1.txt 读入的自定义对象设置按钮的图标、文本和位置
    pb->setIcon(mb1.i); pb->setText(mb1.s); pb->move(mb1.p);
    w.resize(400,333);    w.show();    return aa.exec();    }

```

14、以下为 QDataStream 类的常用函数

- status()函数可用于读取数据时的错误检测，若发生错误，则>>操作符总是读取 0 值或空值，该函数还返回其他值，详见对该函数的讲解
- setByteOrder()函数可设置字节序
- 若到达数据末尾或若没有设置 I/O 设备，则 atEnd()函数会返回 true。
- 使用 setDevice()函数可更改 I/O 设备，I/O 设备通常在构造函数中设置。

二、QDataStream 类中的函数

1、构造函数

1)、QDataStream();

QDataStream(QIODevice *d); //构造一个使用 I/O 设备 d 的数据流

2)、QDataStream(QByteArray *a, QIODevice::OpenMode mode);

QDataStream(const QByteArray &a);

构造一个在 QByteArray 上运行的数据流，其中 mode 表示打开模式(即只读、只写、读写等模式)，第 2 个函数只能是只读模式。由于 QByteArray 不是 QIODevice 的子类，因此在内部会创建 QBuffer 来包装该对象。

2、常用函数

3)、ByteOrder **byteOrder**() const;

//返回当前的字节序，

void **setByteOrder**(ByteOrder bo);

设置字节序为 bo，默认为 QDataStream::BigEndian，除非有特殊要求，否则建议保留此设置，ByteOrder 枚举可取值为 QDataStream::BigEndian(大端)，QDataStream::LittleEndian(小端)

4)、void **setDevice**(QIODevice *d);

//设置 I/O 设备为 d，设置为 0 将取消当前的 I/O 设备

QIODevice ***device**() const;

//返回当前设置的 I/O 设备，若未设置设备，则返回 0。

5)、Status **status()** const; //返回数据流的状态
void **resetStatus()**; //重置数据流的状态
void **setStatus**(Status *status*);
设置数据流的状态为 *status*, 直到调用 **resetStatus()** 之前将忽略对 **setStatus()** 的后续调用。
Status 枚举见下表

QDataStream::Status 枚举(无标志)		
作用：描述数据流的状态		
成员	值	说明
QDataStream::Ok	0	正常运行
QDataStream::ReadPastEnd	1	已读取超过底层设备中数据的末尾
QDataStream::ReadCorruptData	2	已读取损坏的数扭
QDataStream::WriteFailed	3	无法写入底层设备

6)、bool **atEnd()** const;
若 I/O 设备已到达结束位置, 流或文件的末尾, 或没有 I/O 设备集, 则返回 **true**; 否则返回 **false**。
7)、int **version()** const; //返回序列化格式的版本号
void **setVersion**(int *v*);
设置序列化格式的版本号为 *v*, 即枚举 QDataStream::Version 的值, 见下表

QDataStream::Version 枚举(无标志)					
作用：版本号					
成员	值	说明	成员	值	说明
QDataStream::Qt_1_0	1	版本 1	QDataStream::Qt_4_8		同 Qt_4_6
QDataStream::Qt_2_0	2	版本 2	QDataStream::Qt_4_9		同 Qt_4_6
QDataStream::Qt_2_1	3	版本 3	QDataStream::Qt_5_0	13	版本 13
QDataStream::Qt_3_0	4	版本 4	QDataStream::Qt_5_1	14	版本 14
QDataStream::Qt_3_1	5	版本 5	QDataStream::Qt_5_2	15	版本 15
QDataStream::Qt_3_3	6	版本 6	QDataStream::Qt_5_3		同 Qt_5_2
QDataStream::Qt_4_0	7	版本 7	QDataStream::Qt_5_4	16	版本 16
QDataStream::Qt_4_1		同 Qt_4_0	QDataStream::Qt_5_5		同 Qt_5_4
QDataStream::Qt_4_2	8	版本 8	QDataStream::Qt_5_6	17	版本 17
QDataStream::Qt_4_3	9	版本 9	QDataStream::Qt_5_7		同 Qt_5_6
QDataStream::Qt_4_4	10	版本 10	QDataStream::Qt_5_8		同 Qt_5_6
QDataStream::Qt_4_5	11	版本 11	QDataStream::Qt_5_9		同 Qt_5_6
QDataStream::Qt_4_6	12	版本 12	QDataStream::Qt_5_10		同 Qt_5_6
QDataStream::Qt_4_7		同 Qt_4_6			

8)、FloatingPointPrecision **floatingPointPrecision()** const; //返回数据流的浮点精度
void **setFloatingPointPrecision**(FloatingPointPrecision *precision*); //qt4.6
设置数据流的精度为 *precision*。默认为 DoublePrecision。对于 Qt_4_6 之前的版本, 数据流中浮点数的精度取决于调用的流运算符, 注意: 必须在写入和读取数据流的对象上将此属性设置为相同的值。FloatingPointPrecision 枚举见下表

QDataStream::FloatingPointPrecision 枚举(无标志)

作用：读/写数据的浮点数的精度

成员	值	说明
QDataStream::SinglePrecision	0	数据流中的所有浮点数都是 32 位精度
QDataStream::DoublePrecision	1	数据流中的所有浮点数都是 64 位精度

3、事务相关

9)、void **startTransaction**(); //qt5.7

在流上启动一个新的读取事务。在读取操作序列中定义可恢复点。

10)、void **abortTransaction**(); //中止读取事务，把数据流的状态设置为 ReadCorruptData。qt5.7

11)、bool **commitTransaction**(); //qt5.7

完成读取事务，若在事务期间没有发生读取错误，则返回 true，否则返回 false，

12)、void **rollbackTransaction**(); //qt5.7

恢复读取事务，此函数通常用于在提交事务之前检测到不完整的读取时回滚事务。如果前面的流操作成功，则将数据流的状态设置为 ReadPastEnd。

4、以下函数用于读取和写入原始二进制数据

注：缓冲区 s 使用 new[] 分配，使用 delete 操作符销毁

13)、int **readRawData**(char *s, int len);

从流中读取最多 len 个字节到缓冲区 s 中，并返回读取的字节数，若产生错误，则返回-1，缓冲区 s 必须被预先分配。数据是未编码的。

14)、int **writeRawData**(const char *s, int len);

把 len 个字节的数据从缓冲区 s 写入流，并返回实际写入的字节数，若发生错误，则返回-1，注意：数据未编码。

示例：读/写原始数据 (readRawData() 和 writeRawData() 函数的使用)

```
#include<QtWidgets>
int main(int argc, char *argv[]) {
    QFile f("F:/1.txt");
    f.open(QIODevice::WriteOnly);
    QDataStream out(&f);
    out<<QString("BBB"); /*以二进制形式写入一个字符串 BBB 到文件 1.txt 中，注意：以此方式写入的字符串 BBB 是被编码的，因此实际写入文件的内容并不一定是 4 字节的大小。*/
    const char *pcl=new const char[4];    pcl="ccc";    //创建缓冲区
    int il=out.writeRawData(pcl,4); /*把缓冲区 pcl 中的 4 个字节内容写入流中，注：因为此函数写入的内容未编码，因此将直接向文件写入一个字符串"ccc" (4 字节大小)，文件 1.txt 中的最终内容见示例后的图示*/
    qDebug()<<il;    //输出实际写入的字节数，本示例为 14 字节。
    f.close();    //关闭文件

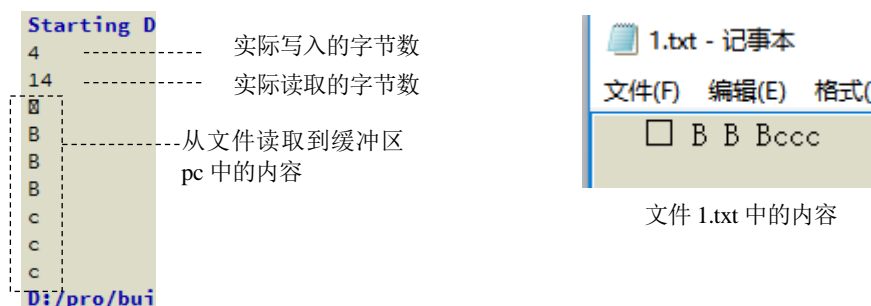
    f.open(QIODevice::ReadOnly);    //以只读模式重新打开文件
    QDataStream out1(&f);
    char *pc=new char[114];    //创建一个缓冲区
```

```

int i2=out1.readRawData(pc,114);    /*从流中读取 114 个字节内容到缓冲区 pc 中，此时 pc
                                   中保存的值就是文件 1.txt 中的原始二进制数据，注意：实
                                   际未必会读取 114 个字节*/

qDebug()<<i2;                      //输出实际读取的字节数
for(int j=0;j<i2;j++)              /*因为输出 char*时会遇到'\0'时停止输出，为输出全部内容，以循环
                                   的方式逐字符输出。*/
qDebug()<<pc[j];    }              /*输出缓冲区中的内容，注意：因为是以字符串形式输出的，而 pc 中
                                   的内容是原始二进制数据，所以输出的内容会包含乱码。，输出的内
                                   容见下图*/

```



readRawData()和 writeRawData()函数类似于标准 C++中以二进制模式使用文件流(fstream)来保存和读取对象的方法(具体使用方法详见《C++语法详解》一书，本处不重述)，下面再以一示例讲解 readRawData()和 writeRawData()函数与 C++文件流的类似用法

示例：使用 readRawData() 和 writeRawData() 函数读取/保存对象

```

#include<QtWidgets>
class B{public:int a; int b;};    //自定义类
int main(int argc, char *argv[]){
    B mb;    mb.a=11;    mb.b=22;    //创建对象 mb
    QFile f("F:/1.txt");    f.open(QIODevice::WriteOnly);
    QDataStream out(&f);
    /*保存对象 mb 到文件中，注意实参的使用方法，读者可自行打开 1.txt 文件，查看其中的内容(实际
    上是一些乱码)*/
    int i1=out.writeRawData((char*)&mb,sizeof mb);
    qDebug()<<i1;    //输出实际写入的字节数
    f.close();    //关闭文件

    f.open(QIODevice::ReadOnly);    //以只读模式重新打开文件
    QDataStream out1(&f);
    B mb1;
    //从文件中读取 sizeof mb1 个字节到&mb1 中。
    int i2=out1.readRawData((char*)&mb1,sizeof mb1);
    qDebug()<<i2;    //输出实际读取的字节数
    qDebug()<<mb1.a;    qDebug()<<mb1.b;    } /*输出 11 和 22，可见，保对象 mb 被成功保存到文
    件并被成功的读取。*/

```

15)、QDataStream &readBytes(char *&s, uint &u);

从流中读取需要读取的数据的长度,并保存在 `u` 中,然后读取 `u` 个字节到缓冲区 `s` 中,并返回对流的引用。如果字符串读取为空,则 `u` 设置为 0, `s` 设置为空指针。此函数与 `readRawData()` 的区别在于,该函数不需要指定参数 `u` 的实际值,这个值是从流中读取的,也就是说流(比如文件)中需要保存有需要读取的数据的长度。

16)、`QDataStream & writeBytes(const char *s, uint len);`

把长度 `len` 和缓冲区 `s` 写入流,并返回对流的引用, `len` 被序列化为一个 `quint32` 的整数,后面跟着 `s` 中的 `len` 字节,也就是说该函数会把长度 `len` 写入流,而 `writeRawData()` 函数不会把长度写入流。注意:数据未编码。

17)、`int skipRawData(int len);`

从设备中跳过 `len` 字节,并返回实际跳过的字节数,若出错时返回-1,这相当于在长度为 `len` 的缓冲区上调用 `readRawData` 并忽略缓冲区。另见 `QIODevice::seek()` 函数

5、`QDataStream` 类还包含如下形式的一系列的重载<<和>>运算符函数,他们的形式都是一样的,只是形参的类型不相同,其余的函数从略。

`QDataStream &operator<<(qint8 i);` `QDataStream &operator>>(qint8 &i)`

`QDataStream &operator<<(qint16 i);` `QDataStream &operator>>(qint16 &i)`

14.3 QTextStream 类(文本流)

一、字符编码基础知识

- 1、怎样将字符转换为二进制形式进行存储，存在一个编码的问题，通常都需进行两次编码，
- 2、字符集：字符的第一次编码是将字符编码为与一个数值(如一个 10 进制整数)相对应，比如把字符 A 编码为 10 进制的 65，B 编码为 66 等。把每一个字符都编码为与一个数值对应就组成了一个字符集，比如常用的 ASCII 字符集，Unicode 字符集，GB2312 字符集等。
- 3、编码(或称为编码字符集)：字符的第二次编码就是把第一次编码好的数值再编码为相应的二进制形式，这样计算机就能识别了，比如对于 Unicode 字符集有 3 种不同的二次编码方案，分别是 UTF-8(变长位)，UTF-16(16 位)和 UTF-32(32 位)，目前使用较多的是使用 UTF-8 来存储的 Unicode 字符集。本文把第二次编码后的方案简称为编码，比如 UTF-8 编码，UTF-16 编码等。
- 4、字节顺序标记 BOM(Byte Order Mark)：BOM 是出现在文本文件头部的一种用于标识文件格式的编码，UTF-16 和 UTF-32 通常使用 BOM 来表示文本的字节序，字节序对 UTF-8 没有意义，因此 UTF-8 不需要使用 BOM 来表明字节序，但可使用 BOM 来表明其编码方式，通常使用 0xEF BB BF 来表明此文本是使用的 UTF-8 编码。UTF-8 不推荐使用无意义的 BOM，但很多程序在保存 UTF-8 编码的文件时仍然带有 BOM(即在文件的开头加上 0xEF BB BF 三个字节)，比如 windows 的记事本等，因此在编辑 UTF-8 的文件时，需要注意该文件是否带有 BOM 的问题。
- 5、QString 和 QByteArray 简介(本文不会详细讲解这两个类，详见帮助文档)
QString 存储一个 16 位的 QChar 字符串，其中每个 QChar 对应一个 Unicode4.0 字符(即存储的字符含有 16 位)，对于代码值超过 65536 的 Unicode 字符使用两个连续的 QChar 表示。
QByteArray 类用于存储原始字节和传统的 8 位以'\0'终止的字符串。Qt 内部大量使用了 QString，因此通常应使用 QString，QByteArray 主要用于存储原始二进制数据。

二、QTextStream 基本规则

- 1、二进制文件格式更紧凑，但它是机器语言，不易于人工阅读和编辑，为此可使用文本格式来代替二进制格式。
- 2、QTextStream 类用于对数据进行文本格式的读/写操作，可在 QString、QIODevice 或 QByteArray 上运行，使用 QTextStream 可方便的读/写单词、行和数字，另外 QTextStream 还对字段填充、对齐和数字格式提供了格式选项的支持。
- 3、QTextStream 与编码和字符集
 - ①、QTextStream 在其内部使用 16 位(两字节)长的 QChar 类型存放每个字符，字符集使用 Unicode，这与 C++ 的 iostream 不同，iostream 每个字符的类型由模板参数 charT 指定，标准库已将其特化为 char 和 wchar_t 类型，除此之外还可为 charT 指定其他类型，而 QTextStream 的字符类型固定为 QChar 类型，使用此种方式简化了 Qt 流的总体结构，但也增加了字符占据的空间。

- ②、QTextStream 能在 Unicode 编码与系统的本地编码或其他任意编码间进行转换，且明确的处理了因系统的不同而导致的不同的行尾符的问题(比如，在 Windows 上行尾符是“\r\n”，在 UNIX 或 mac OS X 上是“\n”)，行尾符还可在打开设备时指定 QIODevice::Text 枚举来设置。
 - ③、QTextStream 使用 QTextCodec 类来支持不同的字符集，默认使用 QTextCodec::codecForLocale()返回的编码进行读/写，也可使用 QTextStream::setCodec()函数来重新设置编码。
 - ④、QTextStream 支持自动 Unicode 的 BOM 检测，当启用此功能(默认)时，QTextStream 将检测 UTF-16 或 UTF-32 的字节顺序标记 BOM(Byte Order Mark)，并在读取时切换到适当的 UTF 编解码器。默认情况下，QTextStream 不编写 BOM，但是可以通过调用 setGenerateByteOrderMark(True)来启用 BOM。
- 4、QTextStream 有 3 种读取文本文件的方式(详细使用方法见后文示例)，如下：
- ①、调用 readLine()逐行读取数据，使用 readAll()一次读取整个文件。
 - ②、一个单词接一个单词的读取，单词由空格分开，且可自动跳过前导空格。通过在 QString、QByteArray 或 char*缓冲区上使用>>操作符来实现。
 - ③、一个字符接一个字符的读取，通过在 QChar 或 char 类型上使用使用>>操作符来实现。可使用 skipWhiteSpace()来跳过空格
- 5、格式控制：
- ①、QTextStream 模仿了<iostream>的控制符(也称为操作器)，比如可使用 dec 等流控制符以 10 进制形式显示数字，另外还可使用 setIntegerBase()、setNumberFlags()等函数来设置格式。
 - ②、当从文本流中读取数字时，QTextStream 会自动检测数字的基数，比如，若数字以 0x 开始，则将被假定为 16 进制形式，若以 1~9 开头，则被假定为 10 进制形式。还可使用 dec 等流控制符、setIntegerBase()函数来设置基数，从而停止自动检测。
 - ③、QTextStream 还可以进行基本数字类型和字符串之间的转换。
- 6、写入文本数据比较容易，但读取就比较难了，比如

```
out<<"AAA"<<"BBB"    //把 AAA 和 BBB 写入流
in>>s1>>s2;           //试图从流中读取 AAA 到 s1，BBB 到 s2，
```

若使用 QTextStream 不能获得这个结果，此时 s1="AAABBB"，而 s2 什么也没有，若使用 QDataStream 则能使 s1="AAA"，s2="BBB"，因为 QDataStream 在字符串数据前面保存了每个字符串的长度。

- 7、由于文本流使用缓冲区(用于存储中间数据，这减少了对设备的访问数量)，所以不应该使用设备的相应函数直接读取，比如，若一个 QFile 直接使用 QFile::readLine()读取，而不是使用 QTextStream::readLine()，那么文本流的内部位置会与文件的位置不同步。

示例：使用 QTextStream 写入字符串

说明：1、因为 Qt 与 VC++对中文的兼容性问题，以下示例建议使用 MinGW 编译，否则中文会显示乱码。

2、QTextStream 虽然能在 Unicode 编码与其他任意编码间进行转换，但并不支持其他任间编码间的转换，因此在使用字符串时，建议使用 QString 类型的字符串，以确保输出的字符串为 Unicode 编码。

```
#include<QtWidgets>
```

```
int main(int argc, char *argv[]) {
    QFile f("F:/1.txt");          f.open(QIODevice::WriteOnly);
    QTextStream out(&f);
    //使用 QTextCodec::codecForName() 函数设置编码的好处是可以根据返回值判断设置的编码是否有效。
    QTextCodec *i=QTextCodec::codecForName("GB18030");
    if(i!=0) out.setCodec(i);
    out<<QString("a23b 检说 ui9d 极");    }
```

运行说明：

读者可找到 1.txt 文件查看其内容，若要以不同的编码方式查看输入的文本，可使用浏览器等可转换编码的软件查看，当选择编码 GB18030 查看 1.txt 文本的内容时能正确显示中文，若选择以 UTF-8 编码方式查看，则会把中文显示为乱码。

三、QTextStream 类中的函数

1、构造函数

1)、QTextStream();

QTextStream(QIODevice *device); //构造一个在设备 device 上运行的 QTextStream

2)、QTextStream(FILE *fileHandle, QIODevice::OpenMode openMode = QIODevice::ReadWrite);

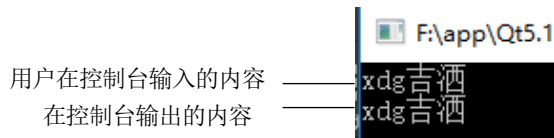
此函数用于在控制台进行输入输出，详见示例

示例：在控制台输入输出数据

使用控制台需要在 pro 文件中添加 CONFIG +=console 语句，并建议使用 MinGW 编译器运行 Qt 程序

```
#include<QtWidgets>
int main(int argc, char *argv[]) {
    QString s;
    QTextStream in(stdin, QIODevice::ReadOnly); //以只读模式从控制台输入内容到 s
    in.setCodec("GB18030"); //若要存储中文需使用此编码
    in>>s; //把控制台输入的内容赋给 s
    QTextStream out(stdout, QIODevice::WriteOnly);
    out<<s; } //把 s 的内容输出到控制台
```

运行结果及说明见下图



3)、QTextStream(QString *string, QIODevice::OpenMode openMode = QIODevice::ReadWrite);

构造一个在字符串 string 上运行的 QTextStream，打开模式由 openMode 指定。

4)、QTextStream(QByteArray *array, QIODevice::OpenMode openMode = QIODevice::ReadWrite);

构造一个在 array 上运行的 QTextStream，在内部 array 会由 QBuffer 包装。

5)、QTextStream(const QByteArray &array, QIODevice::OpenMode openMode = QIODevice::ReadOnly);

构造一个在 array 上运行的 QTextStream，无论 openMode 的值如何，array 都以只读方式访问。

2、对 QTextStream 流的操作函数

- 6)、void flush();
刷新等待写入设备的缓冲区数据，若 QTextStream 对字符串进行操作，则此函数什么也不做。如果调用此函数，QTextStream 会将写入缓冲区中的所有数据清空到设备中，并调用设备上的 flush()。该函数的作用其实就是刷新缓冲区。
- 7)、bool atEnd() const;
若没有更多的数据从 QTextStream 中读取(即到达流的末尾)，则返回 true，否则返回 false。
- 8)、void resetStatus(); //重置 QTextStream 的状态
Status status() const; //返回 QTextStream 的状态
void setStatus(Status status);
设置 QTextStream 的状态。直到调用 resetStatus()之前将忽略对 setStatus()的后续调用。
Status 枚举见下表

QTextStream::Status 枚举(无标志)		
作用：描述文本流的状态		
成员	值	说明
QTextStream::Ok	0	正常运行
QTextStream::ReadPastEnd	1	已读取超过底层设备中数据的末尾
QTextStream::ReadCorruptData	2	已读取损坏的数扭
QTextStream::WriteFailed	3	无法写入底层设备

- 9)、QIODevice *device() const; //返回与 QTextStream 关联的当前设备，若未分配设备，则返回 0
void setDevice(QIODevice *device);
把当前设备设置为 device，若已分配了设备，则 QTextStream 在更换旧设备之前调用 QTextStream::flush()。注意：此函数会把语言环境重置为默认语言环境，将编解码器重置为默认编解码器 QTextCodec::codecForLocale()
- 10)、QString *string() const; //返回分配给 QTextStream 的 string，若未分配，则返回 0
void setString(QString *string, QIODevice::OpenMode openMode = QIODevice::ReadWrite);
使用打开模式 openMode 重新设置 QTextStream 的设备为 string，若设备已分配，则 QTextStream 在更换旧设备之前调用 QTextStream::flush()

3、读取流中的数据及位置

- 11)、QString read(qint64 maxlen); //从流中读取 maxlen 个字符，并将其作为 QString 返回。
- 12)、QString readAll();
从流中读取全部内容，并将其作为 QString 返回，处理大型文件时应避免使用此函数，因为会消耗大量内存。
- 13)、QString readLine(qint64 maxlen = 0);
从流中读取一行文本，并将其作为 QString 返回，允许的最大行长度为 maxlen，若流的长度超过 maxlen，则行将在 maxlen 之后被拆分，并以部分的形式返回。若 maxlen

为 0，则行可以是任意长度。返回的行没有行尾字符("\n"或"\r\n")，若流已到达文件末尾，则返回空的 QString。

14)、bool **readLineInto**(QString *line, quint64 maxlen = 0); //qt5.5

从流中读取一行文本到 line，若 line 为 0，则不存储读取的行。允许的最大行长度为 maxlen，若流的长度超过 maxlen，则行将在 maxlen 之后被拆分，并以部分的形式返回。若 maxlen 为 0，则行可以是任意长度。返回的行没有行尾字符("\n"或"\r\n")，若流已到达文件末尾，则返回空的 QString。若流已读取到文件末尾或发生错误，则返回 false，否则返回 true。

15)、bool **seek**(quint64 pos); //查找设备中的位置 pos，若找到，则返回 true，否则返回 false

16)、quint64 **pos**() const;

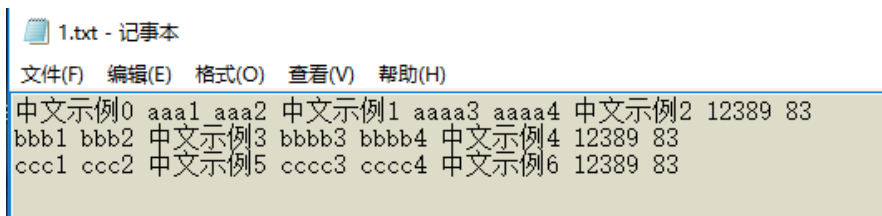
返回与流当前位置对应的设备位置，或当发生错误时返回-1(比如没有设备)，此函数可能很昂贵。

17)、void **skipWhiteSpace**();

从流中读取并丢弃空白字符，直到检测到非空白字符，或直到 atEnd()返回 true。此处的空白字符是指 QChar::isSpace()函数返回 true 的字符

示例：使用 QTextStream 流读取文件的内容

文本 1.txt 的内容如下图



```
#include<QtWidgets>
int main(int argc, char *argv[]){ //QApplication aa(argc,argv);
    QFile f("F:/1.txt");
    f.open(QIODevice::ReadOnly);
    QTextStream in(&f);      in.setCodec("GB18030");
    QChar c;
    //1、逐字符读取的方法
    in>>c;      //从 1.txt 读取第一个字符
    qDebug()<<c;    //输出\u4e2d(这是“中”字的 Unicode 编码)
    in>>c;      //继续读取第 2 个字符
    qDebug()<<c;    //输出\u6587(这是“文”字的 Unicode 编码)
    //2、一个单词接一个单词的读取
    in.seek(0);    //把位置设置为 0，以便从头开始读取
    QString s;
    in>>s;      //读取一个单词到 s 中(单词由空格分开)，
    qDebug()<<s;    //输出“中文示例 0”。
    in>>s;      //继续读取下一个单词到 s 中，
    qDebug()<<s;    /*输出“aaa1”。>>运算符会自动跳过前导空格，所以读取到的单词前面
                    没有空白符*/
    //3、读取指定个数的字符
```

```

    in.skipWhiteSpace(); //跳过空白。
    s=in.read(10);       //继续读取 10 个字符，并把读取到的字符保存到 s 中
    qDebug()<<s;        //输出“aaa2 中文示例 1”
//4、逐行读取
    s=in.readLine();    //继续读取余下的整行文本，并把结果保存到 s 中
    qDebug()<<s;        /*输出“aaaa3 aaaa4 中文示例 2 12389 83”，注意：因为为跳过空白，所以最前面有一个空白字符*/
    in.seek(0);         //重置位置
    s=in.readLine(30);  /*读取 30 个字符或整行(本行大于 30 个字符，因此只读取 30 个字符)，并把结果保存到 s 中*/
    qDebug()<<s;        //输出“中文示例 0 aaa1 aaa2 中文示例 1 aaaa3 aa”
//5、读取整个文本
    in.seek(0);
    s=in.readAll();
    qDebug()<<s;        /*输出“中文示例 0 aaa1 aaa2 中文示例 1 aaaa3 aaaa4 中文示例 2 12389
83\r\nbbb1 bbb2 中文示例 3 bbbb3 bbbb4 中文示例 4 12389 83\r\ncccc1
ccc2 中文示例 5 cccc3 cccc4 中文示例 6 12389 83\r\n”，由输出可见，
readAll() 函数包含了行尾符“\r\n”*/
//6、使用循环逐行读取整个文件。
    while(!in.atEnd())    s+=in.readLine();
    qDebug()<<s;}

```

4、编码相关

18)、QTextCodec ***codec()** const; //返回当前流的编解码器

void **setCodec**(QTextCodec ***codec**);

void **setCodec**(const char ***codecName**);

设置流的编解码器，其中 codecName 是以字符串的形式指定编解码器，若无法识别该字符串，则不会发生任何事情，codecName 的值有：“ISO 8859-1”、“UTF-8”、“UTF-16”、“GB 18030”、“Big5”等，详细的取值详见 QTextCodec 类。

19)、QLocale **locale()** const; //返回区域，默认语言环境是 C

void **setLocale**(const QLocale &**locale**); //设置流的区域为 locale，

20)、bool **autoDetectUnicode()** const; //是否启用自动 Unicode 检测，默认为 true(即，启用)

void **setAutoDetectUnicode**(bool **enabled**);

若 enabled 为 true，则 QTextStream 会尝试查看流数据来检测 Unicode 的字节顺序标记 BOM，若找到此标记，则 QTextStream 将使用 UTF 编解码器替换当前编解码器。

21)、bool **generateByteOrderMark()** const;

若 UTF 编解码器产生 BOM，则返回 true，否则返回 false。默认为 false

void **setGenerateByteOrderMark**(bool **generate**);

若 generate 为 true 且使用了 UTF 编解码器，则 QTextStream 将在把数据写入设备之前插入 BOM，若 generate 为 false，则不会插入 BOM。该函数需在写入数据之前调用。

5、格式控制

22)、void **reset()**; //重置 QTextStream 的格式选项，使其恢复到原始构造状态。

23)、int **fieldWidth()** const; //返回当前字段的宽度

void **setFieldWidth**(int *width*);

设置当前字段的宽度，若 *width* 为 0(默认值)，则字段宽度等于生成的文本的长度。字段宽度适用于调用此函数后添加到此流的每个元素

24)、QChar **padChar**() const; //返回当前的填充字符

void **setPadChar**(QChar *ch*);

设置填充字符为 *ch*，默认为 ASCII 空格字符，或 QChar 的 0x20，生成文本时，字符 *ch* 用于填充字段中的空格

25)、FieldAlignment **fieldAlignment**() const; //返回当前字段的对齐方式，FieldAlignment 枚举见下表

void **setFieldAlignment**(FieldAlignment *mode*); //设置字段的对齐方式，与 setFieldWidth()一起使用。

QTextStream::FieldAlignment 枚举(无标志)

作用：当字段宽度大于占用字段的文本时，文本的对齐方式

成员	值	说明
QTextStream::AlignLeft	0	在字段右侧填充
QTextStream::AlignRight	1	在字段左侧填充
QTextStream::AlignCenter	2	在字段两侧填充
QTextStream::AlignAccountingStyle	3	与 AlignRight 相同，除数字符号左对齐外。

26)、int **integerBase**() const; //返回整数的基数

void **setIntegerBase**(int *base*);

设置整数的基数，*base* 可取值为 2、8、10、16(分别表示二进制、8 进制等)，若 *base* 为 0，则 QTextStream 将通过检查流上的数据来检测其基数。生成数字时，QTextStream 假设 *base* 为 10，除非明确设置 *base*

27)、NumberFlags **numberFlags**() const; //返回当前的数字标志，枚举 NumberFlag 见下表

void **setNumberFlags**(NumberFlags *flags*); //设置数字标志

QTextStream::NumberFlag 枚举

标志：QTextStream::NumberFlags

作用：主要描述输出数字时的一些格式

成员	值	说明
QTextStream::ShowBase	0x1	若基数为 16、8、或 2，则显示其前缀 0x、0 和 0b
QTextStream::ForcePoint	0x2	始终显示小数分隔符(比如小数点)
QTextStream::ForceSign	0x4	始终显示数字的符号，即使是正数，比如 2，显示为+2
QTextStream::UppercaseBase	0x8	使用大写版本的基数前缀(即 0X、0B 等)
QTextStream::UppercaseDigits	0x10	使用大写字母表示数字 10 到 35，而不是小写

28)、RealNumberNotation **realNumberNotation**() const; //返回当前的实数表示法，枚举见下表

void **setRealNumberNotation**(RealNumberNotation *notation*); //设置实数表示法

QTextStream::RealNumberNotation 枚举(无标志)

作用：描述浮点数的表示法

成员	值	说明
----	---	----

QTextStream::ScientificNotation	2	科学记数法表示浮点数
QTextStream::FixedNotation	21	定点法表示浮点数
QTextStream::SmartNotation	0	科学记数法或定点法表示浮点数

- 29)、int **realNumberPrecision()** const; //返回当前实数的精度
void **setRealNumberPrecision**(int *precision*); //设置实数的精度。精度不能为负值，默认为6。

6、QTextStream 类还包含如下形式的一系列的重载<<和>>运算符函数，他们的形式都是一样的，只是形参的类型不相同，其余的函数从略。

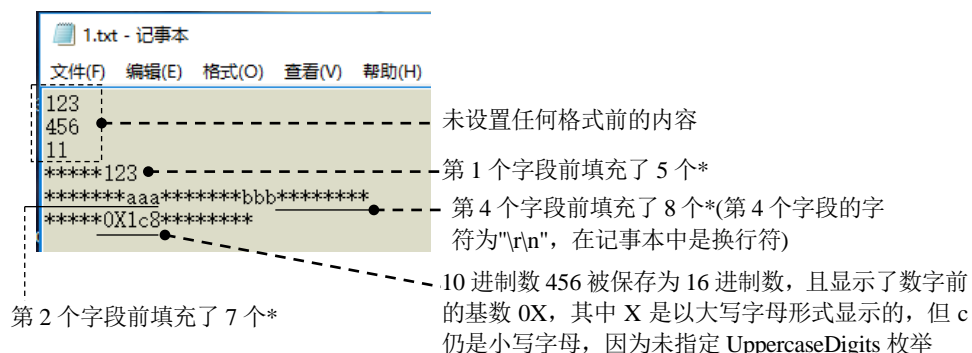
QTextStream &operator<<(QChar c); QTextStream &operator<<(char c);
QTextStream &operator<<(signed short i); QTextStream &operator<<(unsigned short i);
QTextStream &operator<<(signed int i); QTextStream &operator<<(unsigned int i);

示例：QTextStream 的格式控制

```
#include<QtWidgets>
int main(int argc, char *argv[]) {
    QFile f("F:/1.txt");           f.open(QIODevice::WriteOnly);
    QTextStream out(&f);
    /*未设置格式前输入 1.txt 的内容，QTextStream 能自动识别数字，0xB 会被保存为 10 进制的 11。
    注意：123 是字符串，不是数字。*/
    out<<"123\r\n"<<456<<"\r\n"<<0xB<<"\r\n";
    out.setFieldWidth(10);       //设置字段宽度为 10
    out.setPadChar('*');       //设置填充字符为*
    out.setFieldAlignment(QTextStream::AlignRight); //设置文本在字段间的对齐方式
    out.setIntegerBase(16);   //数字以 16 进制保存
    //显示数字基数并以大写字母形式显示基数前缀
    out.setNumberFlags(QTextStream::ShowBase|QTextStream::UppercaseBase);

    out<<"123\r\n"               /*输入第 1 个字段到 1.txt，在 123 之前填充 5 个*，注意"\r\n"会
    占两个字符宽度*/
    <<"aaa"                    /*输入第 2 个字段，在 aaa 之前填充 7 个*
    <<"bbb"<<"\r\n"       /*输入第 3 和第 4 个字段，注意：这里是两个字段而不是一个，因
    此会在 bbb 之前填充 7 个*，在"\r\n"之前填充 8 个*。*/
    <<456<<"\r\n"; } /*输入第 5 和第 6 个字段。填充字符原理同上。数字 456 被以 16 进
    制的形式输入到 1.txt 中*/
```

在 F 盘找到 1.txt 文件，其内容如下图所示



四、以其他方式设置 QTextStream 流的格式

- 1、除了可使用 QTextStream 类的成员函数控制其格式外，还可使用类似于 C++ 的 iostream 流的控制符(Qt 称为操作器)以及便捷的全局函数来控制 QTextStream 流的格式
- 2、下表为 Qt 的操作器、全局函数与 QTextStream 对应的成员函数

Qt 的操作器及与格式相关的全局函数

- 说明：1、以下全局函数的返回类型均为 QTextStream &
- 2、为便于书写，以下对 QTextStream 成员函数及其枚举的引用均省略了 QTextStream::作用域限定符的使用
- 2、具体说明可参阅相关成员函数的介绍，
- 3、操作器主要用于控制数字的格式。操作器的使用方法比较简单，如下示例所示
- ```
out<<hex<<showbase<<456<<"\r\n"<<123;
```
- 4、全局函数的使用也比较简单，上面代码的等价全局函数如下：

```
hex(out); showbase(out); out<<456<<"\r\n"<<123;
```

| 操作器                                                                     | 便捷的全局函数                                 | 等价的 QTextStream 成员函数调用                                                                                    |
|-------------------------------------------------------------------------|-----------------------------------------|-----------------------------------------------------------------------------------------------------------|
| bin                                                                     | <b>bin</b> (QTextStream &s)             | setIntegerBase(2)                                                                                         |
| oct                                                                     | <b>oct</b> (QTextStream &s)             | setIntegerBase(8)                                                                                         |
| dec                                                                     | <b>dec</b> (QTextStream &s)             | setIntegerBase(10)                                                                                        |
| hex                                                                     | <b>hex</b> (QTextStream &s)             | setIntegerBase(16)                                                                                        |
| showbase                                                                | <b>showbase</b> (QTextStream &s)        | setNumberFlags(numberFlags() ShowBase)                                                                    |
| forcesign                                                               | <b>forcesign</b> (QTextStream &s)       | setNumberFlags(numberFlags() ForceSign)                                                                   |
| forcepoint                                                              | <b>forcepoint</b> (QTextStream &s)      | setNumberFlags(numberFlags() ForcePoint)                                                                  |
| noshowbase                                                              | <b>noshowbase</b> (QTextStream &s)      | setNumberFlags(numberFlags()&~ShowBase)                                                                   |
| noforcesign                                                             | <b>noforcesign</b> (QTextStream &s)     | setNumberFlags(numberFlags()&~ForceSign)                                                                  |
| noforcepoint                                                            | <b>noforcepoint</b> (QTextStream &s)    | setNumberFlags(numberFlags()&~ForcePoint)                                                                 |
| uppercasebase                                                           | <b>uppercasebase</b> (QTextStream &s)   | setNumberFlags(numberFlags() UppercaseBase)                                                               |
| uppercasedigits                                                         | <b>uppercasedigits</b> (QTextStream &s) | setNumberFlags(numberFlags() UppercaseDigits)                                                             |
| lowercasebase                                                           | <b>lowercasebase</b> (QTextStream &s)   | setNumberFlags(numberFlags()&~UppercaseBase)                                                              |
| lowercasedigits                                                         | <b>lowercasedigits</b> (QTextStream &s) | setNumberFlags(numberFlags()&~UppercaseDigits)                                                            |
| fixed                                                                   | <b>fixed</b> (QTextStream &s)           | setRealNumberNotation(FixedNotation)                                                                      |
| scientific                                                              | <b>scientific</b> (QTextStream &s)      | setRealNumberNotation(ScientificNotation)                                                                 |
| left                                                                    | <b>left</b> (QTextStream &s)            | setFieldAlgnment(AlignLeft)                                                                               |
| right                                                                   | <b>right</b> (QTextStream &s)           | setFieldAlgnment(AlignRight)                                                                              |
| center                                                                  | <b>center</b> (QTextStream &s)          | setFieldAlgnment(AlignCenter)                                                                             |
| endl                                                                    | <b>endl</b> (QTextStream &s)            | 写入 '\n' 并刷新流，等于如下语句<br>stream<<"\n"<<flush;<br>注意：对于 Windows，若使用 QIODevice::Text 打开文件，则 '\n' 会被 '\r\n' 代替 |
| flush                                                                   | <b>flush</b> (QTextStream &s)           | 等同于 flush()                                                                                               |
| reset                                                                   | <b>reset</b> (QTextStream &s)           | 等于 reset()                                                                                                |
| ws                                                                      | <b>ws</b> (QTextStream &s)              | 等于 skipWhiteSpace()                                                                                       |
| bom                                                                     | <b>bom</b> (QTextStream &s)             | 等于 setGenerateByteOrderMark(true)                                                                         |
| 除以上全局函数外，还有以下 3 个全局函数                                                   |                                         |                                                                                                           |
| QTextStreamManipulator <b>qSetFieldWidth</b> (int <i>width</i> )        |                                         | 等于调用 setFieldWidth(width)                                                                                 |
| QTextStreamManipulator <b>qSetPadChar</b> (QChar <i>ch</i> )            |                                         | 等于调用 setPadChar(ch)                                                                                       |
| QTextStreamManipulator <b>qSetRealNumberPrecision</b> (int <i>pre</i> ) |                                         | 等于调用 setRealNumberPrecision(pre)                                                                          |

## 14.4 QFile 类(文件)

### 一、QFile 基本原理

#### 1、QFile 基础

- ①、QFile 类继承自 QFileDevice 类，QFileDevice 又继承自 QIODevice 类。
- ②、QFile 类提供了一个用于读取/写入文件的接口，是一种读写二进制文件、文本、资源的 I/O 设备。
- ③、默认情况下 QFile 假定为二进制，即不对存储在文件中的字节执行任何转换。
- ④、与其他 QIODevice 不同，QFile 不会发送 QIODevice::aboutToClose()，QIODevice::bytesWritten()、QIODevice::readyRead()信号，这意味着 QFile 不适合读取/写入某些类型的文件(比如 Unix 上的 device 文件)

#### 2、读写文件内容的方法

- ①、QFile 可通过自身直接对文件进行读写，比如，使用从 QIODevice 类继承来的 read()、readAll()、readLine()等函数进行读写，这些函数的原型详见后文；还可使用 QTextStream 或 QDataStream 进行读写(这两种方法在前文已讲解过)。
- ②、在 Unix 上，不能使用 QFileDevice::atEnd()来判断文件是否还有更多的数据，因为在 Unix 上有一些特殊的系统文件(比如，在 /proc 中)，它的 QFile::size()总是返回 0，但仍可从这样的文件中读取数据，而 QFileDevice::atEnd()函数对于大小为 0 的文件将返回 true，因此不能使用 atEnd()来判断文件是否还有数据。此时应反复的调用 readAll()、read()或 readLine()来读取数据，直到无数据读取为止。下面为其示例代码

```
QTextStream in(&f); QString s;
QString s1=in.readLine(); //读取文件内容并保存到 s1 中
while(!s1.isNull()) //判断 s1 是否为空，以此来判断是否无数据可读取
{ s += s1; //把文件的所有内容保存到 s 中
 } //其他代码
```

#### 3、文件名及目录分隔符

- ①、目录分隔符：QFile 使用的目录分隔符为"/"，不支持其他分隔符(比如"\")。
- ②、文件名：当使用 QFile、QFileInfo 和 QDir 访问 Qt 文件系统时，可以使用 Unicode 文件名。在 Unix 上，文件名被转换为 8 位编码。可以使用 QFile::encodeName()和 QFile::decodeName()函数在 Unicode 文件名和 8 位文件名之间进行转换。

#### 4、文件权限处理

- ①、在 Unix-like(类 Unix)和 Windows 上，文件权限的处理方式不同，在 Unix-like 上的不可写目录中，无法创建文件，但在 Windows 上却并非总是如此。
- ②、Qt 对文件权限的理解是有限的，这对 QFile::setPermissions()函数影响较大，在 Windows 上，Qt 将仅设置传统的 read-only(只读)标志，且只在没有任何的 Write\*标志未设置时

才设置。Qt 不处理访问控制列表(ACL, access control lists), 这使得此函数对 NTFS 卷几乎无用, 但仍可用于使用 VFAT 文件系统的 USB 存储,

5、下表是一些常用函数及其功能

| QFile 常用函数           |        |                      |            |
|----------------------|--------|----------------------|------------|
| 函数名                  | 简介     | 函数名                  | 简介         |
| QFileDevice::flush() | 刷新文件   | QFileDevice::pos()   | 获取文件当前位置   |
| QFile::size()        | 获取文件大小 | QFileDevice::seek()  | 移动到新位置     |
| QFile::remove()      | 删除文件   | QFileDevice::atEnd() | 检测是否到达文件末尾 |
| QFile::open()        | 打开文件   | QFile::setFileName() | 重新设置文件名    |
| QFileDevice::close() | 关闭文件   | QFile::exists()      | 检查文件是否存在   |

二、QFile 类中的函数

1、构造函数

```
1)、QFile();
 QFile(const QString &name);
 QFile(QObject *parent);
 QFile(const QString &name, QObject *parent);
```

2、文件操作

- 2)、virtual bool open(OpenMode mode);
- QIODevice::open()的重新实现, 使用模式 mode 打开文件, 若成功, 则返回 true, 否则返回 false。在 WriteOnly 或 ReadWrite 模式下, 若文件不存在, 则此函数在打开之前会尝试创建新文件。
- 3)、bool open(FILE \*fh, OpenMode mode, FileHandleFlags handleFlags = DontCloseHandle);
- bool open(int fd, OpenMode mode, FileHandleFlags handleFlags = DontCloseHandle);
- 在模式 mode 下打开已存在的文件句柄 fh 或文件描述符 fd, 若成功, 则返回 true, 否则返回 false。
  - 参数 handleFlags 用于指定其他选项, 其枚举见下表。

| QFileDevice::FileHandleFlag 枚举   |        |                                                                                                            |
|----------------------------------|--------|------------------------------------------------------------------------------------------------------------|
| 标志: QFileDevice::FileHandleFlags |        |                                                                                                            |
| 作用: 用于指定打开文件时的其他选项               |        |                                                                                                            |
| 成员                               | 值      | 说明                                                                                                         |
| QFileDevice::AutoCloseHandle     | 0x0001 | 传递给 open()的文件句柄应由 close()关闭, 默认行为是 close()只刷新文件, 应用程序负责关闭文件句柄。当按名称打开文件时, 将忽略此标志, 因为此时 Qt 始终拥有文件句柄, 且必须将其关闭 |
| QFileDevice::DontCloseHandle     | 0      | 若未显示关闭, 则在销毁 QFile 对象时, 底层文件句柄将保持打开状态。                                                                     |

- 使用此函数打开 QFile 时，close()的行为由 QFileDevice::AutoCloseHandle 标志控制。如果指定了 QFileDevice::AutoCloseHandle，并且此函数成功，则调用 close() 将关闭所采用的句柄。否则，close()实际上不会关闭文件，而是只刷新它。
- 注意：若 fh 不是常规文件，例如，它是 stdin、stdout 或 stderr，那么可能无法使用 seek()。在这些情况下，size()返回 0。
- 由于此函数在未指定文件名的情况下打开文件，因此不能将此 QFile 与 QFileInfo 一起使用。

4)、virtual void **QFileDevice::close**(); //虚拟的

QIODevice::close()的重新实现。调用 QFileDevice::flush()并关闭文件。

5)、bool **remove**();

static bool **remove**(const QString &fileName); //静态的

删除文件 fileName 或由 fileName()指定的文件，若成功，则返回 true，否则返回 false

6)、bool **copy**(const QString &newName);

static bool **copy**(const QString &fileName, const QString &newName); //静态的

- 把文件 fileName 或由 fileName()当前指定的文件复制到名为 newName 的文件中，若成功，则返回 true，否则返回 false。
- 注意：若 newName 已存在，则该函数会返回 false，也就是说，该函数不会覆盖 newName。
- 在复制之前，非静态函数会关闭源文件。静态函数不会关闭源文件，其实调用静态函数复制文件，根本就不需要创建与源文件关联的 QFile 对象，所以不存在关闭源文件的问题。

7)、virtual bool **resize**(qint64 sz); //QFileDevice::resize()的重新实现

static bool **resize**(const QString &fileName, qint64 sz); //静态的

- 把文件 fileName 的大小设置为 sz，若设置成功，则返回 true，否则返回 false。
- 若 sz 大于 fileName 的大小，则新字节被设置为 0，若 sz 较小，则将文件截断。
- 注意：若文件不存在，此函数可能会失败。

8)、virtual qint64 **size**() const; //QIODevice::size()的重新实现。获取文件的大小

9)、bool **exists**() const;

static bool **exists**(const QString &fileName); //静态的

若文件 fileName 或由 fileName()指定的文件存在，则返回 true，否则返回 false

### 示例：文件的复制(其原理及说明见示例后的图示)

```
#include<QtWidgets>
```

```
int main(int argc, char *argv[]) {
```

```
 QFile f("F:/1.txt"); f.open(QIODevice::ReadWrite);
```

```
 int i=QFile::copy("F:/1.txt", "F:/3.txt"); //复制失败(因为 3.txt 已存在), i=0;
```

```
 i=f.copy("F:/3.txt"); //复制失败, i1=0;原因同上。
```

```
 i=f.copy("F:/4.txt"); //复制成功, i=1, 同时关闭源文件 f。
```

```
 QTextStream out(&f); out<<444; //写入失败, 文件 f 已关闭。
```

```
//使用静态函数复制文件
```

```
 QFile f1("F:/2.txt"); f1.open(QIODevice::ReadWrite);
```

```
 i=QFile::copy("F:/2.txt", "F:/5.txt"); //复制成功, i=1;但不会关闭文件 f1,
```

```

QString out1(&f1); out1<<5555; //写入成功。
i=QFile::copy("F:/3.txt","F:/6.txt"); } /*复制成功, i=1;静态函数可直接复制文件, 不需要
创建与源文件关联的 QFile 对象。*/

```

## 运行结果及说明



复制前假设 F 盘下有如图所示 3 个文件，其文件的内容分别见右图



1.txt 的内容



2.txt 的内容



3.txt 的内容



复制后 F 盘下多了 3 个文件，每个文件的内容见下图



1.txt 的内容



2.txt 的内容



3.txt 的内容



4.txt 的内容



5.txt 的内容



6.txt 的内容

## 3、文件的读写(以下函数均继承自父类，其使用方法与 QTextStream 中的相应函数类似)

10)、qint64 QIODevice::read(char \*data, qint64 maxSize);

从设备读取 maxSize 个字节的数据到 data，并返回读取的字节数，若产生错误，返回 -1，若没有更多数据可读取时，返回 0。注意：在流的末尾读取也被认为是错误，此时也会返回-1。

11)、QByteArray QIODevice::read(qint64 maxSize);

从设备中读取 maxSize 字节的数据，并以 QByteArray 的形式返回，此函数无法报告错误

12)、qint64 QIODevice::readLine(char \*data, qint64 maxSize);

- 从设备读取一行 ASCII 字符，最大字节数为 maxSize-1，并把字符存储在 data 中，并返回读取的字节数。
- 若不能读取行，但未发生错误，则返回 0，若发生错误，则此函数返回可读取的长度，若未读取任何内容，则返回-1。
- 终止符'\0'总是附加到数据中，因此 maxSize 必须大于 1。
- 读取数据直到产生以下条件之一
  - 读取到一个'\n'字符，在 windows 上，换行符被替换为'\r\n'
  - 读取了 maxSize-1 个字节

#### ■ 检测到设备数据结束

- 13)、`QByteArray QIODevice::readLine(qint64 maxSize = 0);`  
从设备中读取一行数据，但不超过 `maxSize`，并以 `QByteArray` 的形式返回，此函数无法报告错误
- 14)、`QByteArray QIODevice::readAll();`  
从设备读取所有剩余数据，并以 `QByteArray` 的形式返回，此函数无法报告错误
- 15)、`qint64 QIODevice::write(const char *data, qint64 maxSize);`  
把最多 `maxSize` 个字节的数据 `data` 写入设备，并返回实际写入的字节数，若发生错误，则返回-1。
- 16)、`qint64 QIODevice::write(const char *data);`  
把以 0 结尾的 8 位字符串数据 `data` 写入设备，并返回实际写入的字节数，若发生错误，则返回-1。
- 17)、`qint64 QIODevice::write(const QByteArray &byteArray);`  
把内容 `byteArray` 写入设备，并返回实际写入的字节数，若发生错误，则返回-1。
- 18)、`bool QIODevice::getChar(char *c);`  
从设备读取一个字符，并将其保存在 `c` 中，若 `c` 为 0，则丢弃该字符，若读取成功，则返回 `true`，否则返回 `false`。
- 19)、`bool QIODevice::putChar(char c);`  
把字符 `c` 写入设备，若成功，则返回 `true`，否则返回 `false`。
- 20)、`void QIODevice::ungetChar(char c);`  
把字符 `c` 放回设备，除非位置为 0，否则递减当前位置，该函数通常被称为 `getChar()` 的撤销操作，若字符 `c` 未从设备读取，则行为是未定义的。注意：当事务正在进行时，此函数不可用。

## 4、文件名相关

- 21)、`virtual QString fileName() const;` //虚拟的  
`QIODevice::fileName()`的重新实现，返回由 `setFileName()`或 `QFile` 构造函数设置的文件名称
- 22)、`void setFileName(const QString &name);`  
设置文件的名称为 `name`，名称可以没有路径、相对路径或绝对路径。若文件已打开，请不要调用此函数。若 `name` 不含路径，则使用的路径是调用 `open()`时应用程序的当前目录路径。
- 23)、`bool rename(const QString &newName);`  
`static bool rename(const QString &oldName, const QString &newName);`//静态的
  - 把文件 `oldName` 或由 `fileName()`指定的文件重命名为 `newName`，若成功，则返回 `true`，否则返回 `false`，
  - 注意：若 `newName` 已存在，则该函数会返回 `false`。
  - 在复制之前，非静态函数会关闭源文件。静态函数不需要创建与源文件关联的 `QFile` 对象，所以不存在关闭文件的问题。
- 24)、`static QString decodeName(const QByteArray &localFileName);` //静态的

static QString **decodeName**(const char \**localFileName*); //静态的  
返回 *localFileName* 的 Unicode 版本，即把 *localFileName* 转换为 Unicode 版本的文件名。

- 25)、static QByteArray **encodeName**(const QString &*fileName*); //静态的  
把 *fileName* 转换为由用户区域设置确定的本地 8 位编码。硬编码到应用程序中的文件名应仅使用 7 位 ASCII 文件名字符。

## 5、文件链接相关

- 26)、bool **link**(const QString &*linkName*);  
static bool **link**(const QString &*fileName*, const QString &*linkName*); //静态的
- 创建一个名为 *linkName*() 的链接(类似于 Windows 下的快捷方式)，该链接指向文件 *fileName* 或当前由 *fileName*() 指定的文件，若成功，则返回 true，否则返回 false。
  - 注意：在 Windows 上创建链接，*linkName* 必须具有 .lnk 后缀(即文件扩展名为 .lnk)

- 27)、QString **symlinkTarget**() const;  
static QString **symlinkTarget**(const QString &*fileName*); //静态的  
返回符号链接(或快捷方式)所引用的文件或目录的绝对路径，若调用该函数的 QFile 对象不是符号链接或 *fileName* 不是符号链接，则返回空字符串。若符号链接指向一个现有文件，则 QFile::exists() 返回 true。

## 6、文件权限相关

- 28)、virtual Permissions **permissions**() const; //QIODevice::Permissions() 的重新实现  
static Permissions **permissions**(const QString &*fileName*); //静态的  
返回文件 *fileName* 的 QFile::Permission 标志的完整的按位或组合。枚举见下表
- 29)、virtual bool **setPermissions**(Permissions *permissions*);  
static bool **setPermissions**(const QString &*fileName*, Permissions *permissions*); //静态的  
QIODevice::setPermissions 的重新实现。将文件的权限设置为 *permissions*，若成功，则返回 true，若无法修改权限，则返回 false。此函数不会处理 ACL，这可能会限制其有效性。

---

### QIODevice::Permission 枚举

标志：QIODevice::Permissions

作用：描述文件的权限和所有权

| 成员                    | 值      | 说明       |
|-----------------------|--------|----------|
| QIODevice::ReadOwner  | 0x4000 | 文件所有者可读  |
| QIODevice::WriteOwner | 0x2000 | 文件所有者可写  |
| QIODevice::ExeOwner   | 0x1000 | 文件所有者可执行 |
| QIODevice::ReadUser   | 0x0400 | 用户可读     |
| QIODevice::WriteUser  | 0x0200 | 用户可写     |
| QIODevice::ExeUser    | 0x0100 | 用户可执行    |
| QIODevice::ReadGroup  | 0x0040 | 文件可由组读到  |
| QIODevice::WriteGroup | 0x0020 | 文件可由组写入  |
| QIODevice::ExeGroup   | 0x0010 | 文件可由组执行  |

|                         |        |           |
|-------------------------|--------|-----------|
| QFileDevice::ReadOther  | 0x0004 | 任何人都可读取文件 |
| QFileDevice::WriteOther | 0x0002 | 任何人都可写入文件 |
| QFileDevice::ExeOther   | 0x0001 | 任何人都可执行文件 |



## 14.5 Qt 资源简介

- 1、资源系统是一种独立于平台的机制，用于在应用程序的可执行文件中存储二进制文件(前面所讨论的数据都存储在外部设备中)。若应用程序始终需要一组特定的文件(比如图标)，则非常有用。
- 2、资源系统基于 `qmake`, `rcc` (Qt 的资源编译器，用于把资源转换为 C++ 代码) 和 `QFile`
- 3、在可执行文件中嵌入数据有利于创建独立的可执行文件，但由于必须容纳被嵌入的数据，因此，可执行文件本身会变得比较大。
- 4、资源收集文件(.qrc): 用于指定与程序关联的资源，该文件是一种基于 XML 的文件格式，该文件大致内容如下:

```
<RCC>
 <qresource prefix="/">
 <file>li.png</file>
 <file>aaa/lhs.png</file>
 <file>aaa/lm.png</file>
 </qresource>
</RCC>
```

其中 `li.png`、`aaa/lhs.png`、`lm.png` 就是 .qrc 文件列出的资源，

下面示例为使用资源设置按钮图标的示例代码

```
QPushButton *pb=new QPushButton("AAA");
pb->setIcon(QIcon(":/aaa/lm.png")); //其中"/" 是访问资源的前缀
```

### 5、资源基本规则

- ①、列出的资源文件必须与 .qrc 文件或其子目录位于同一目录中。
- ②、可以使用 "://" 前缀访问资源。比如 `"/aaa/lm.png"`
- ③、需要把 qrc 文件添加到 pro 文件中，比如

```
RESOURCES += gg.qrc //其中 gg.qrc 为需要添加的 qrc 文件
```

- ④、默认情况下，资源会被压缩，可以关闭压缩(需使用命令行参数来关闭，本文从略)

### 6、资源收集文件基本语法

- ①、资源前缀：使用 `qresource` 标记的 `prefix` 属性添加资源前缀，资源前缀主要用于对添加的资源的一些说明(可以为空)，比如

```
<qresource prefix="/image">
 <file>li.png</file>
</qresource>
```

此时访问资源方法为 `"/image/li.png"`

- ②、资源别名：使用 `file` 标记的 `alias` 属性更改，比如

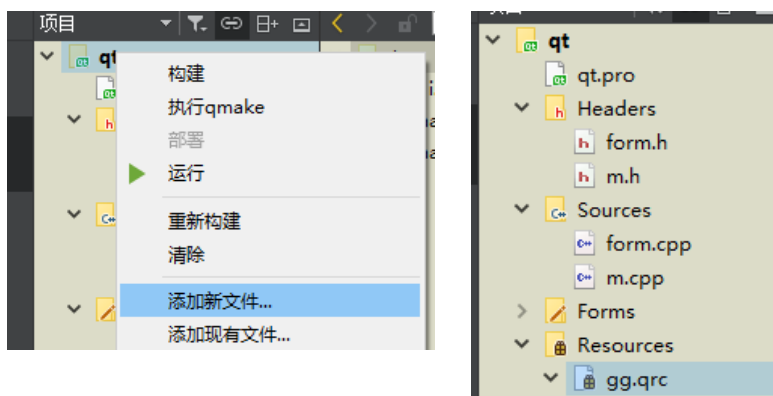
```
<qresource prefix="/image">
 <file alias = "xxx.png">aaa/lhs.png</file>
</qresource>
```

此时应这样访问资源 `"/xxx.png"`

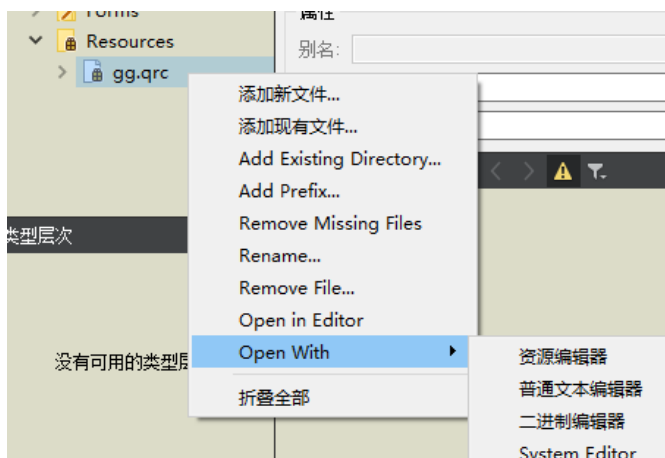
- 7、qrc 文件可在记事本中进行编辑，编辑好之后只需把记事本的后缀修改为 qrc 即可，此时需要在 pro 文件中手动添加 qrc 文件。

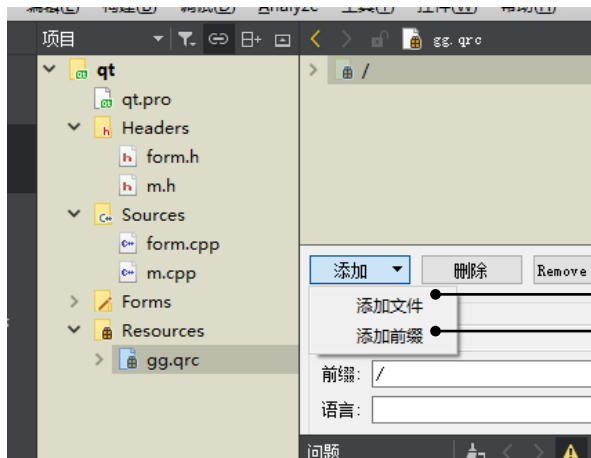
## 8、在 Qt Creator 中编辑资源的方法

- ①、创建 qrc 文件：在 Qt Creator 中创建 qrc 文件，此种方法 Qt 会自动向 pro 文件中添加创建的 qrc 文件，其步骤为：在项目名上右击(如下图左侧)，然后选择“添加新文件...”，弹出对话框，在对话框左侧选择“Qt”，然后在右侧选择“Qt Resource File”，然后点击“Choose...”按钮，接下来输入 qrc 文件的文件名，其余选项保持默认即可。添加 qrc 文件后的项目如下图右侧所示



- ②、编辑 qrc 文件：在创建的 qrc 文件上右击(见以下第 1 幅图)，然后把光标移至“Open With”，若选择“资源编辑器”，则可使用 Qt 资源编辑器编辑 qrc 文件(见以下第 2 幅图)，若选择“普通文本编辑器”，则 qrc 文件会以文本的形式打开，此时可看到 qrc 文件的内容，并可以文本的形式进行编辑。





然后选择此项，向其中添加  
文件即可  
首先需要添加前缀，即使是空的

Qt 的资源编辑器

作者：黄邦勇帅(原名：黄勇)

2018-8-14