Qt GUI编程入门（1）

计算机系

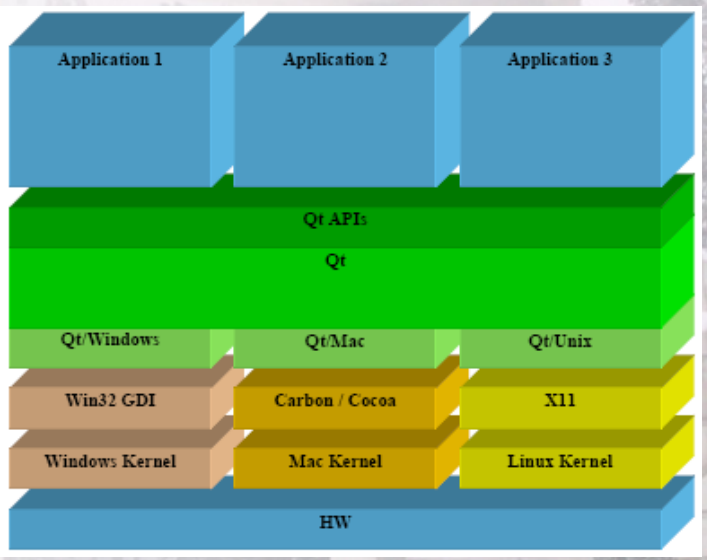
课程主要内容

- Qt简介
- Qt Creator
- Qt基础类
- 信号/槽机制
- GUI界面设计
  - 基于Qt设计器
  - Qt通用部件
  - 基于纯代码
- 元对象系统
- 信号/槽示例
- 实例讲解：温度转换器

Qt简介

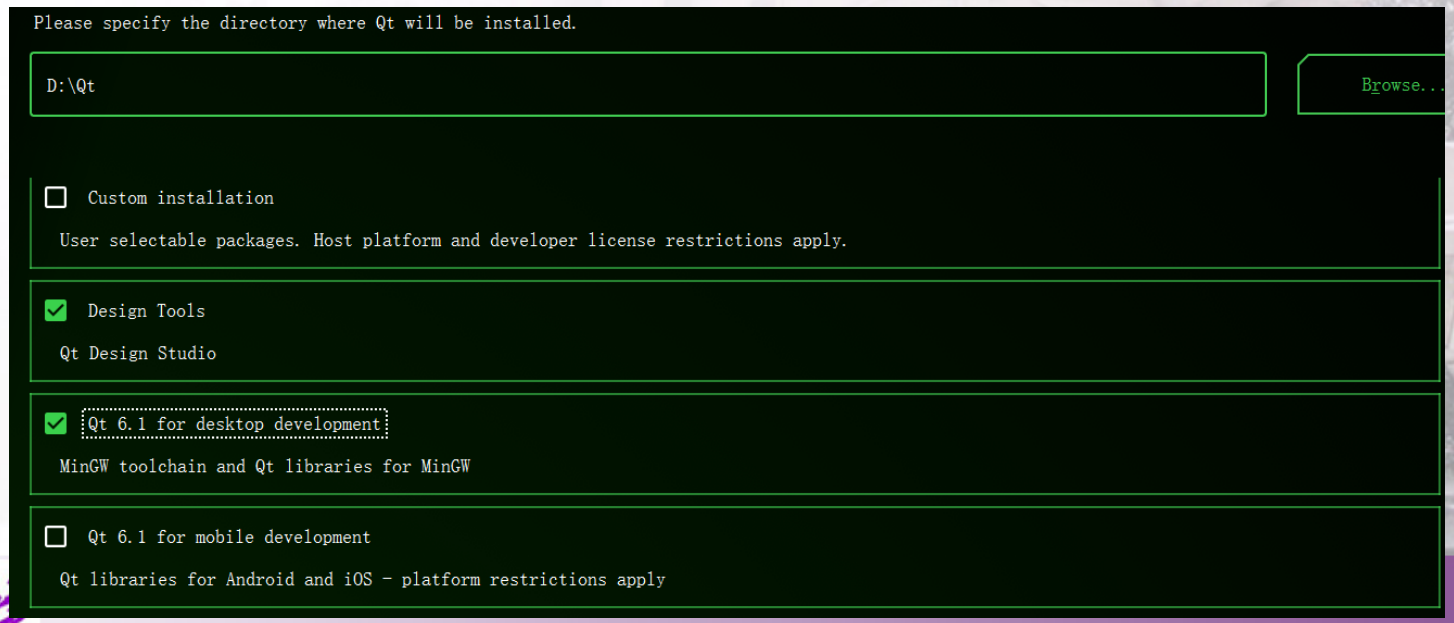
Qt简介

- Qt是跨平台的C++图形用户界面（GUI）开发类库
- Qt兼容多种平台，源程序可以在不同平台编译运行
  - PC/服务器：Windows, Linux, macOS
  - 移动和嵌入式系统：iOS, Android, Embedded Linux, WinRT
- 分为商业许可和开源许可（LGPLV3和GPLV2/GPLV3）
- Qt开源IDE：Qt Creator
- Qt应用举例
  - Autodesk Maya
  - Google Earth
  - Skype
  - WPS Office



Qt版本与安装

- 建议用最新版本：Qt 6.1.2
- 开源IDE下载（Qt Creator）
  - https://www.qt.io/download-open-source
  - 安装时选上Qt 6.1 for desktop development (MinGW)
  - 可不选Design Tools (Qt Design Studio, 用于开发Qt Quick项目)
  - 如果选用其他版本MinGW（不推荐），需要进行环境变量配置，如CPLUS_INCLUDE_PATH/LD_LIBRARY_PATH等



Qt学习资源

- 推荐参考书
  - Qt 5.9 C++开发指南，王维波等著，人民邮电出版社，2018
- 在线学习资源：
  - Qt官网： https://qt.io
  - Qt文档： https://doc.qt.io/ （Qt Creator也自带帮助文档）
  - Qt Wiki：https://wiki.qt.io/Main
  - 指导样例： https://doc.qt.io/qt-6/qtexamplesandtutorials.html
    或Qt Creator自带的Examples

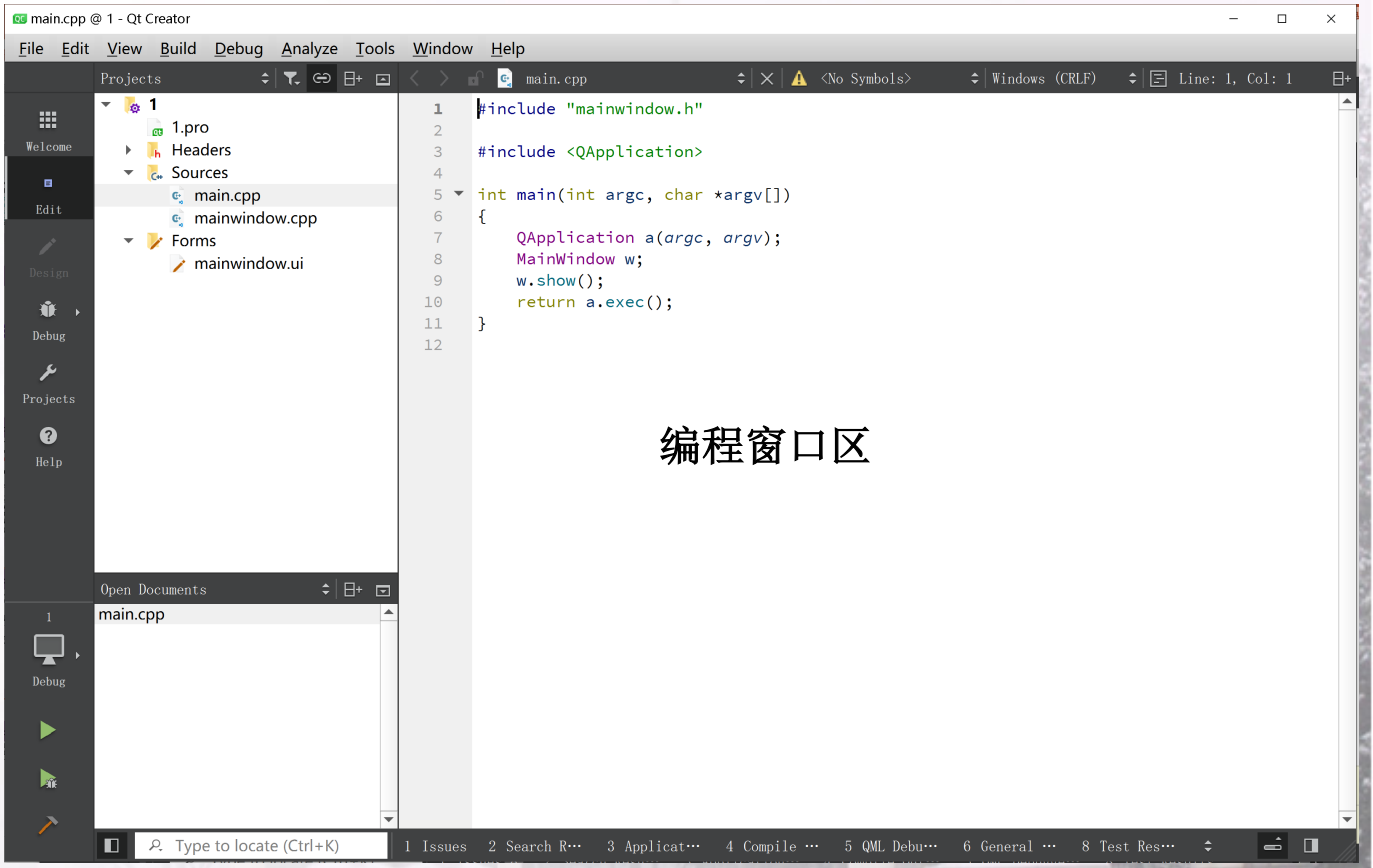
Qt Creator

Qt Creator创建项目

- 文件->新建文件或项目，选择*Application (Qt)*，选项：
  - Qt Widgets Application：一般选这个，带有GUI的应用程序；
  - Qt Console Application：控制台应用程序，无GUI，可以编译C/C++程序；
- 选择目录（*Choose*） -> Build system默认*qmake*（*cmake*等也可以）
- 选择需要创建界面的基类：
  - QMainWindow：主窗口类（缺省选择），拥有菜单栏、工具栏、状态栏
  - QDialog：对话框类，创建基于对话框的界面
  - QWidget：所有可视界面类的基类，支持各种组件
- 勾选*Generate form*复选框自动创建*ui*文件
- Translation选*none* -> Kits选择*MinGW*编译器
- Summary版本控制选择*None*，完成项目创建

Qt Creator主界面

- 目录树
- .pro: 项目文件，保存配置信息
- Headers: 头文件 (.h)
- Sources: C++源文件 (.cpp)
- Forms: 界面文件 (.ui)



编程窗口区

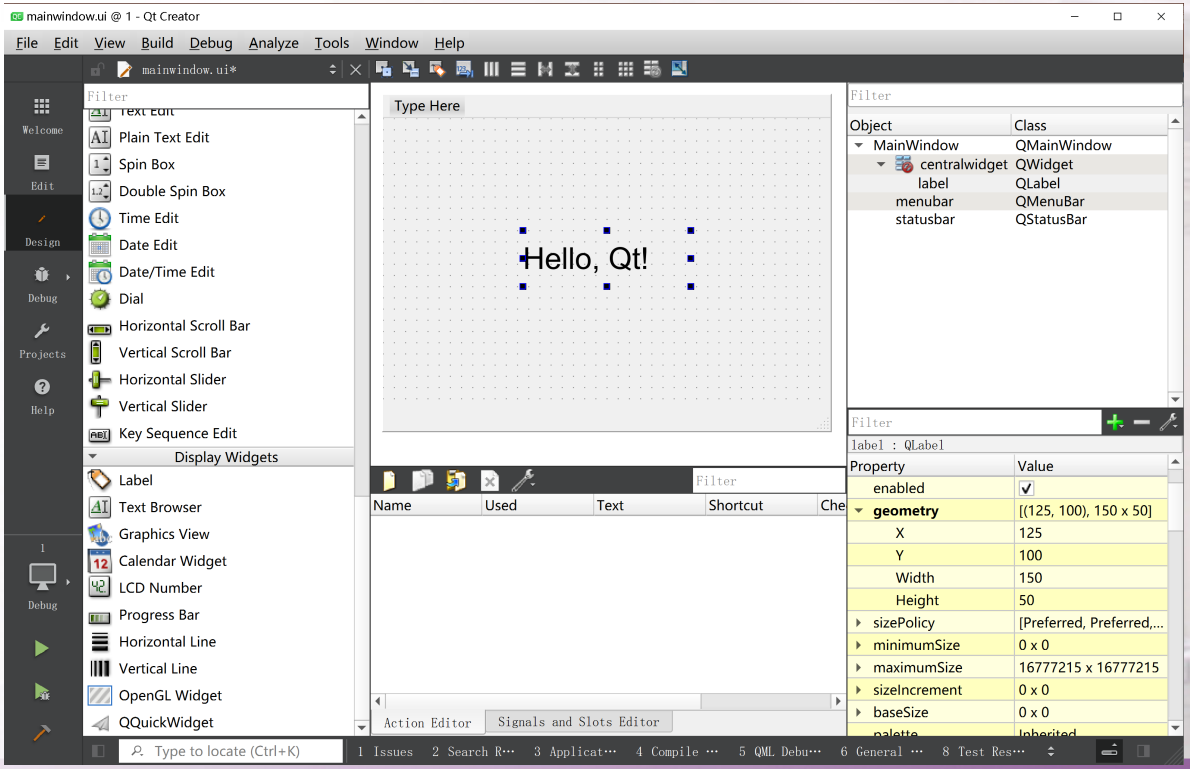
运行按钮

调试按钮

编译按钮

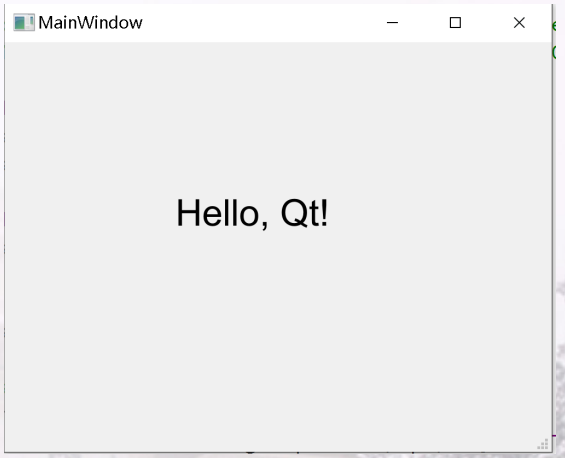
使用Qt设计器（Designer）

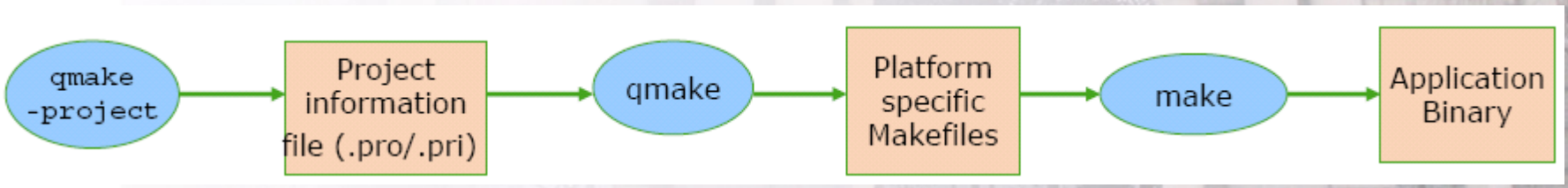
- 双击目录树中的界面文件(mainwindow.ui)，出现窗体设计（Qt Designer）界面
- 左侧是分组的组件面板，从*Display Widgets*中将*Label*组件拖放到设计窗体上，通过拖放设置大小，通过双击编辑文字内容，在右下角属性窗口内设置属性值。



编译运行Qt程序

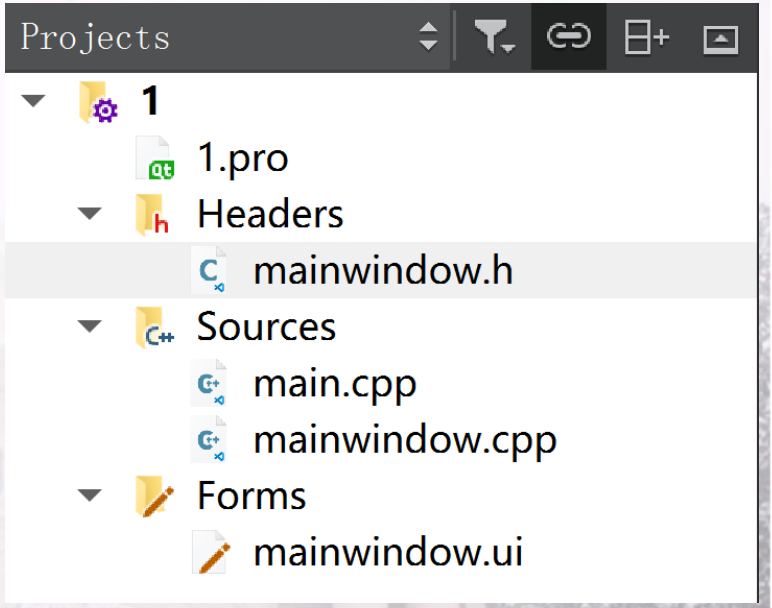
- 方法一：在Qt Creator中编译运行
- 方法二：命令行编译
  - 在项目所在目录下执行“qmake -project”，自动创建Qt工程文件(.pro)。也可以手工创建该文件。
  - 执行“qmake”
    - 缺省输入为工程文件(.pro)，产生平台相关的Makefile(s)
    - Makefile(s)包含编译规则，为包含Q_OBJECT宏的头文件调用moc编译器
  - 执行“make”编译程序

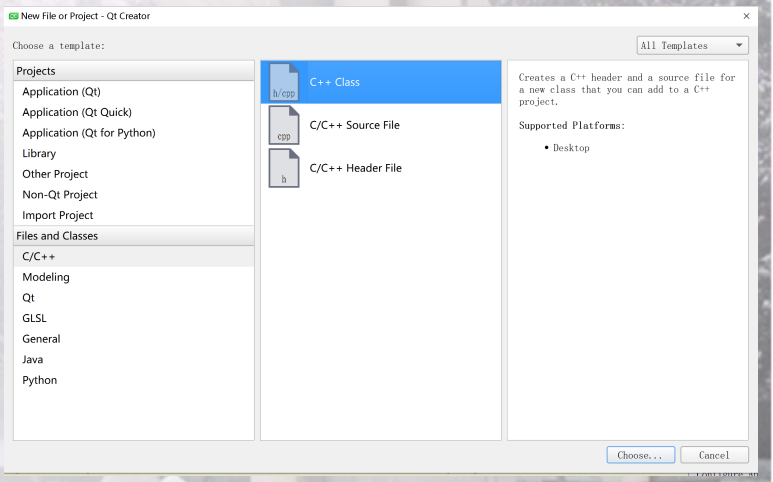




项目文件

- .pro文件存储项目配置信息
- main.cpp：main()入口函数
- 窗体界面文件mainwindow.ui：XML格式文件，存储窗体上的元件及其布局信息
- mainwindow.h：窗体类头文件
- mainwindow.cpp：窗体类实现文件
- 除了以上自动生成的文件，还可以自行添加新类
  - *File -> New File or Project*创建新的C++类，选择继承的基类，自动生成.h/.cpp文件





项目管理文件

- .pro文件
  - 项目模块（QT）
  - 配置（CONFIG）
  - 源代码（SOURCES）
  - 头文件（HEADERS）
  - 窗体文件（FORMS）等信息
- 使用QT Creator添加新件时，将自动更新.pro文件
- 如果手动添加新文件，需要相应修改.pro文件的内容

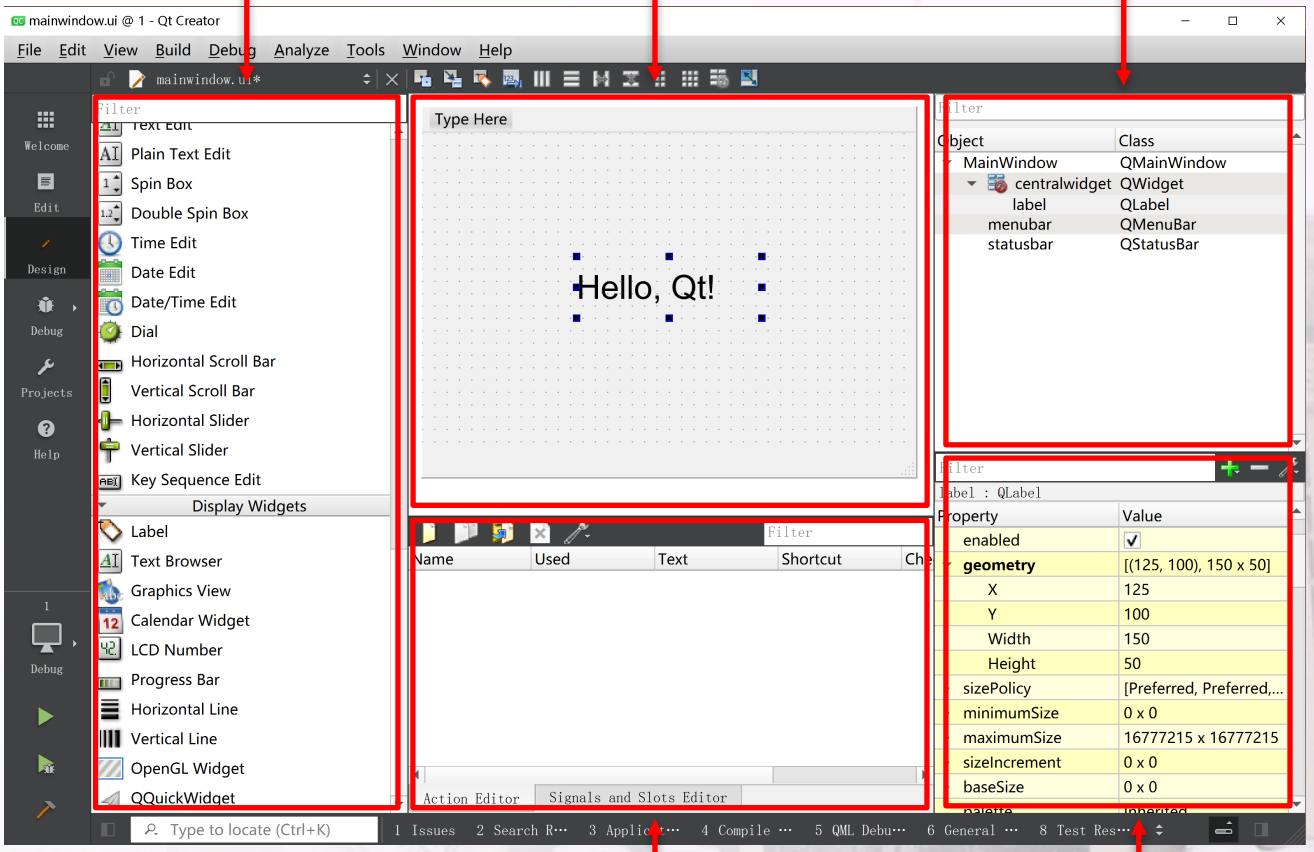
```
QT       += core gui

greaterThan(QT_MAJOR_VERSION, 4): QT += widgets

CONFIG += c++11

# You can make your code fail to compile if it uses dep
# In order to do so, uncomment the following line.
#DEFINES += QT_DISABLE_DEPRECATED_BEFORE=0x060000    #

SOURCES += \
    main.cpp \
    mainwindow.cpp

HEADERS += \
    mainwindow.h

FORMS += \
    mainwindow.ui

# Default rules for deployment.
qnx: target.path = /tmp/${TARGET}/bin
else: unix:!android: target.path = /opt/${TARGET}/bin
!isEmpty(target.path): INSTALLS += target

```

Qt设计器界面

组件面板，包括
常见组件

待设计窗体

对象浏览器，显示组件
对象之间的关系



Action编辑器、信号/槽编辑器

属性编辑器，可设置
组件的属性值

应用程序入口(main.cpp)

- 创建应用程序
  - QApplication是Qt的标准应用程序类，第7行定义了一个实例a，是应用程序对象
- 创建窗口
  - 第8行定义了MainWindow类型的变量w
- 显示窗口
  - 用w.show()显示窗口
- 运行程序
  - 调用a.exec()开始应用程序执行
  - 进入消息循环和事件处理

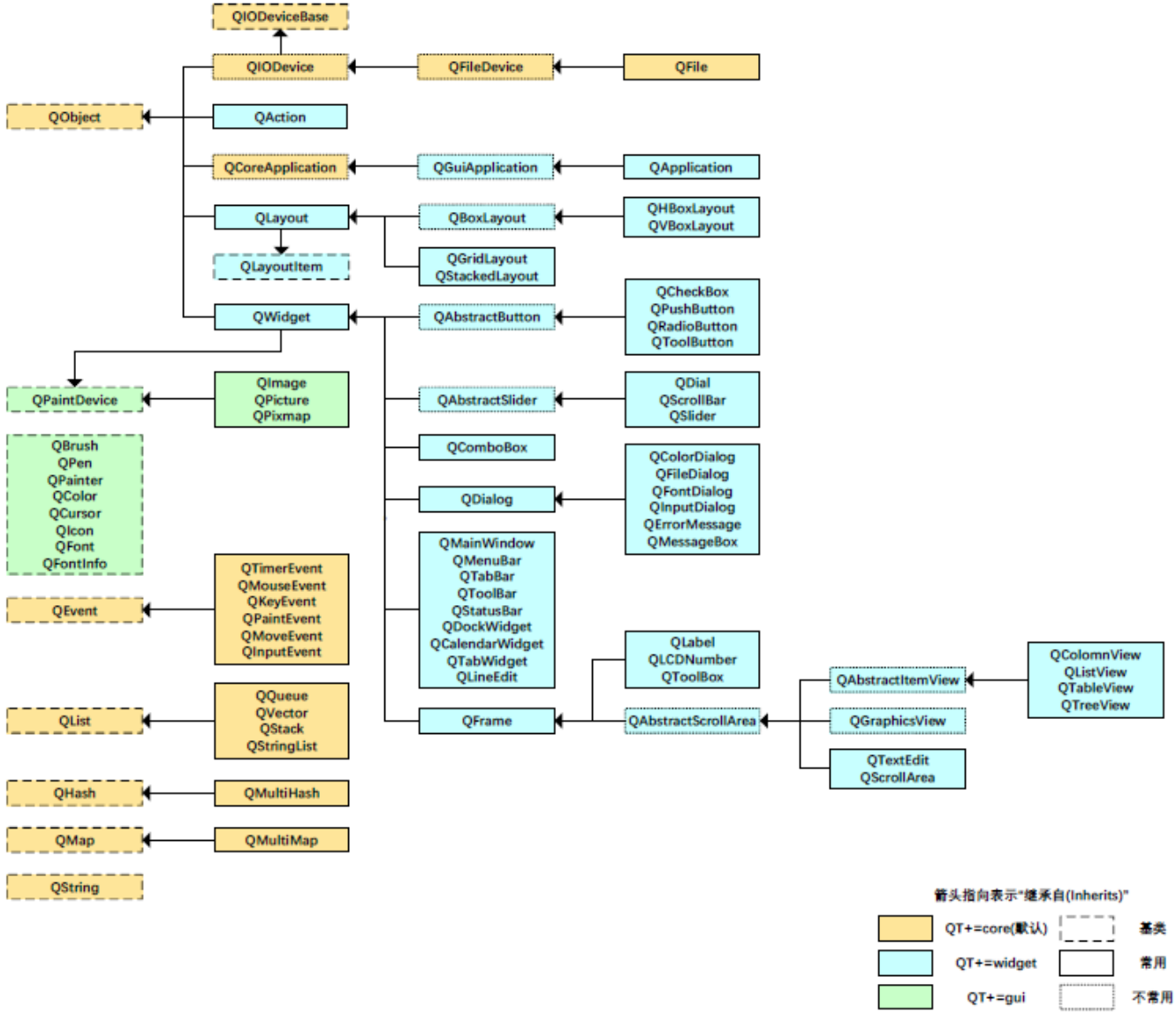
```
1   #include "mainwindow.h"
2
3   #include <QApplication>
4
5   int main(int argc, char *argv[])
6   {
7       QApplication a(argc, argv);
8       MainWindow w;
9       w.show();
10      return a.exec();
11  }
12
```

Qt基础类

基类

- QObject：核心基类
  - 几乎所有Qt类和所有部件(QWidget)的基类，已禁用拷贝构造函数。
  - 实现信号-槽的通信机制、事件处理机制、属性与内省机制、内存管理机制。
- QApplication：事件循环
  - 处理应用程序的开始、结束以及会话管理，负责系统和应用程序的参数设置，负责处理和调度所有来自窗口系统和其他资源的事件。
  - 在Qt应用程序中，不管有几个窗口，QApplication对象只能有一个，而且必须在其他对象之前创建。
  - 可以利用全局指针qApp访问该QApplication对象。
- QWidget：所有界面类的基类
  - 窗口部件从窗口系统接收鼠标、键盘等事件，并在屏幕上绘制自己。
  - 窗口部件包括按钮（button）、菜单（menu）、滚动条（scroll bars）和框架（frame）等，一个窗口部件可以包含其它窗口部件。
  - 当窗口对象被创建时，其部件总是隐藏的，必须调用show()函数使其可见。
- QMainWindow与QDialog：主窗口/对话框（继承自QWidget）
  - QMainWindow用于创建带有主菜单栏、工具栏、状态栏的主窗口，通过centralWidget来调整布局
  - QDialog用于创建一个对话框

Qt类图


QIODeviceBase
QIODevice
QFileDevice
QFile
QObject
QAction
QCoreApplication
QGuiApplication
QApplication
QLayout
QBoxLayout
QHBoxLayout
QVBoxLayout
QLayoutItem
QGridLayout
QStackedLayout
QWidget
QAbstractButton
QCheckBox
QPushButton
QRadioButton
QToolButton
QPaintDevice
QImage
QPicture
QPixmap
QAbstractSlider
QDial
QScrollBar
QSlider
QBrush
QPen
QPainter
QColor
QCursor
QIcon
QFont
QFontInfo
QComboBox
QDialog
QColorDialog
QFileDialog
QFontDialog
QInputDialog
QErrorMessage
QMessageBox
QMainWindow
QMenuBar
QTabBar
QToolBar
QStatusBar
QDockWidget
QCalendarWidget
QTabWidget
QLineEdit
QTimerEvent
QMouseEvent
QKeyEvent
QPaintEvent
QMoveEvent
QInputEvent
QEvent
QLabel
QLCDNumber
QToolBox
QAbstractItemView
QColomnView
QListView
QTableView
QTreeView
QQueue
QVector
QStack
QStringList
QList
QFrame
QAbstractScrollArea
QGraphicsView
QTextEdit
QScrollArea
QHash
QMultiHash
QMap
QMultiMap
QString
箭头指向表示"继承自(Inherits)"
QT+=core(默认)
基类
QT+=widget
常用
QT+=gui
不常用

程序退出与关闭窗口

- 程序退出
  - 调用quit()或exit(0)
  - a. / qApp-> / QApplication::三种方法效果相同
  - 程序退出时默认关闭所有窗口
- 关闭窗口
  - 对于窗口：w.close()
  - 对于应用程序：a.closeAllWindows()
  - 如果关闭窗口后，不想让应用程序退出，可设置QApplication::setQuitOnLastWindowClosed (false)

```
1   #include "mainwindow.h"
2
3   #include <QApplication>
4
5   int main(int argc, char *argv[])
6   {
7       QApplication a(argc, argv);
8       MainWindow w;
9       w.show();
10      return a.exec();
11  }
12
```

信号/槽机制

信号与槽

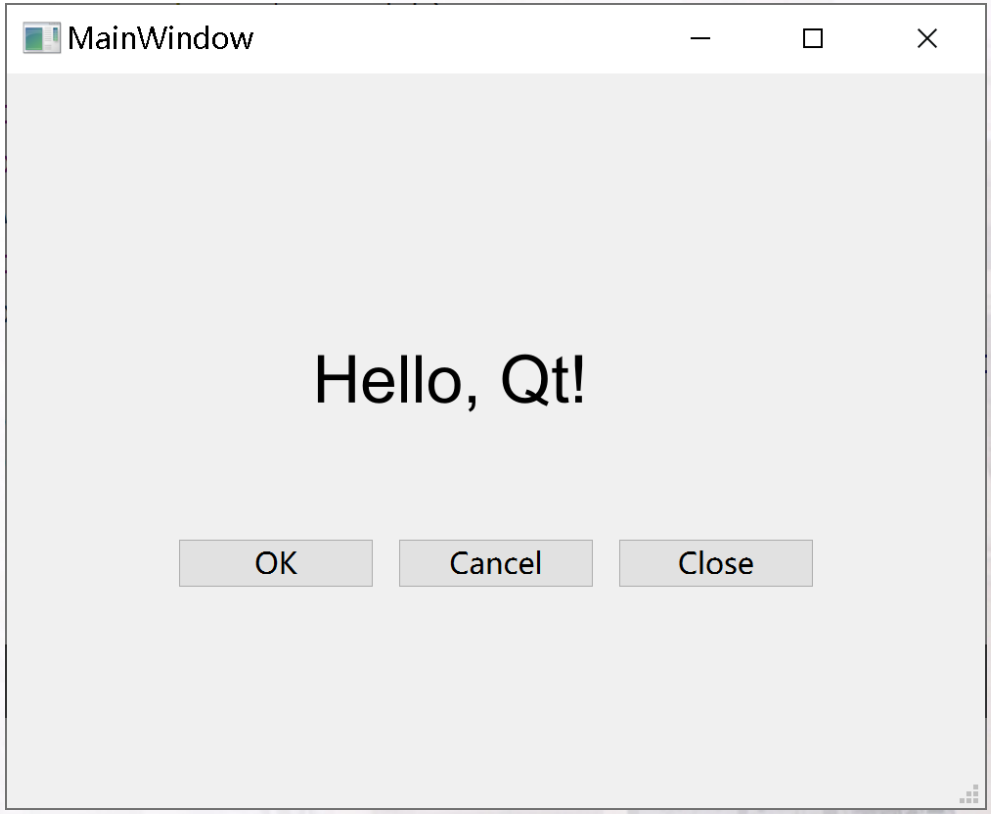
- 信号/槽(Signal/Slot)机制是Qt GUI编程的基础，也是Qt的一大创新，便于Qt事件处理以及界面组件的交互。
  - 只有继承自QObject类或QObject的子类，才能支持信号/槽
- 信号（Signal）：特定情况下发射的事件（函数调用）
  - 如PushButton在鼠标单击时发出的clicked()信号
- 槽（Slot）：对信号响应的函数
  - 槽函数可以与一个信号关联：信号被发射时，所有关联的槽函数将自动执行，类似观察者设计模式
  - 除了可以响应信号外，与普通成员函数相同
  - 可以定义为public / private / protected，可以带有参数，可以被直接调用
- 信号与槽的关联：多对多的对应关系
  - 使用QObject::connect()函数实现，支持Lambda函数
  - 除了编写代码连接信号与槽，也可以通过Qt Designer设置连接

signal和slot的执行逻辑

- 信号与槽的执行
  - 如果信号和槽实现在同一个线程中，当信号发射的时候，与它关联的槽将马上执行
  - 如果信号和槽不在同一个线程中，槽的执行可能会有延迟(next event loop)
  - 只有当与信号关联的所有槽函数执行完毕后，才会执行发射信号处后面的代码

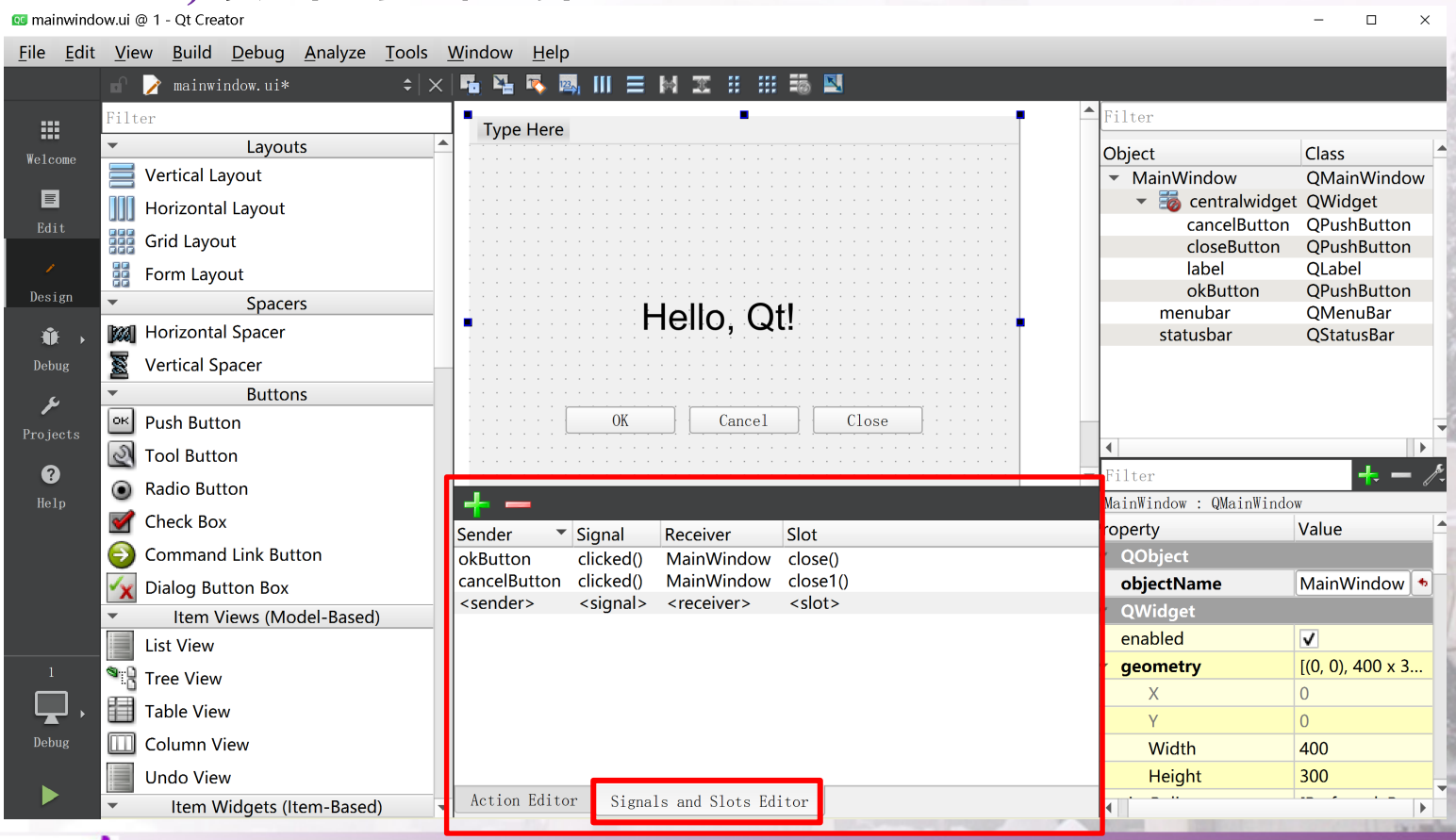
使用Qt设计器连接信号与槽

- 举例：给之前的hello world程序添加三个功能相同的关闭按键



使用Qt设计器连接信号与槽

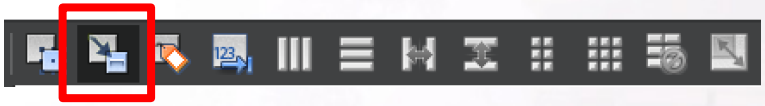
- 方法1：使用设计器直接连接已经存在的信号与槽
  - 拖动添加组件后，使用界面下方的信号/槽编辑器(*Signals and Slots Editor*)添加信号和槽的信息



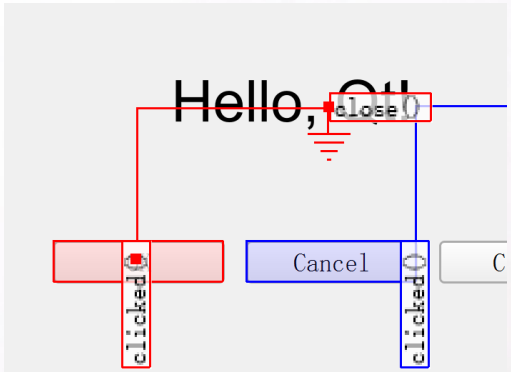
使用Qt设计器连接信号与槽

- 方法2：在设计器中拖动鼠标连接信号与槽
  - 在模式栏点击进入信号/槽编辑模式：Edit Signals/Slots（F4）
  - 从一个部件（sender）拖放鼠标到另一个部件（receiver）
  - 选中相应的signal和对应的slot
  - 在信号/槽编辑器中查看结果

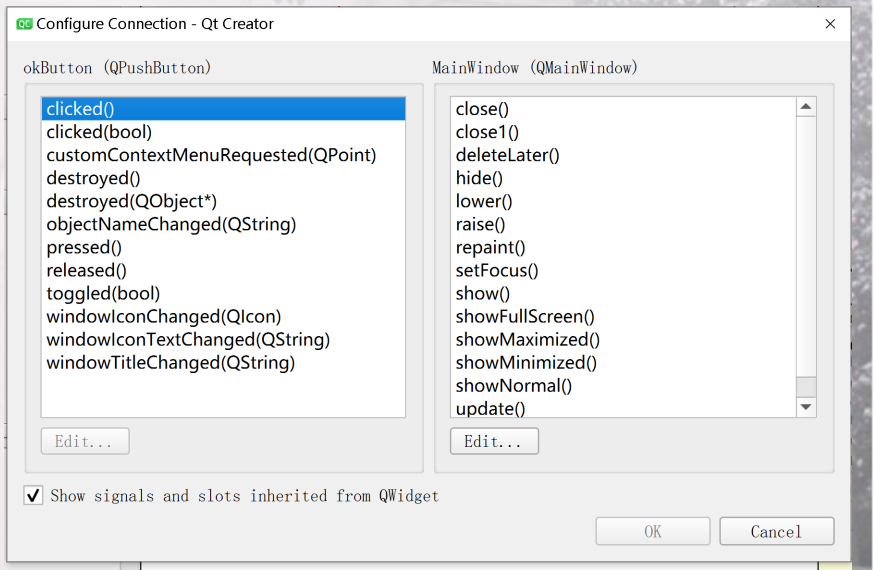
1



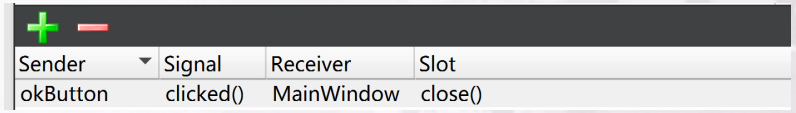
2



3

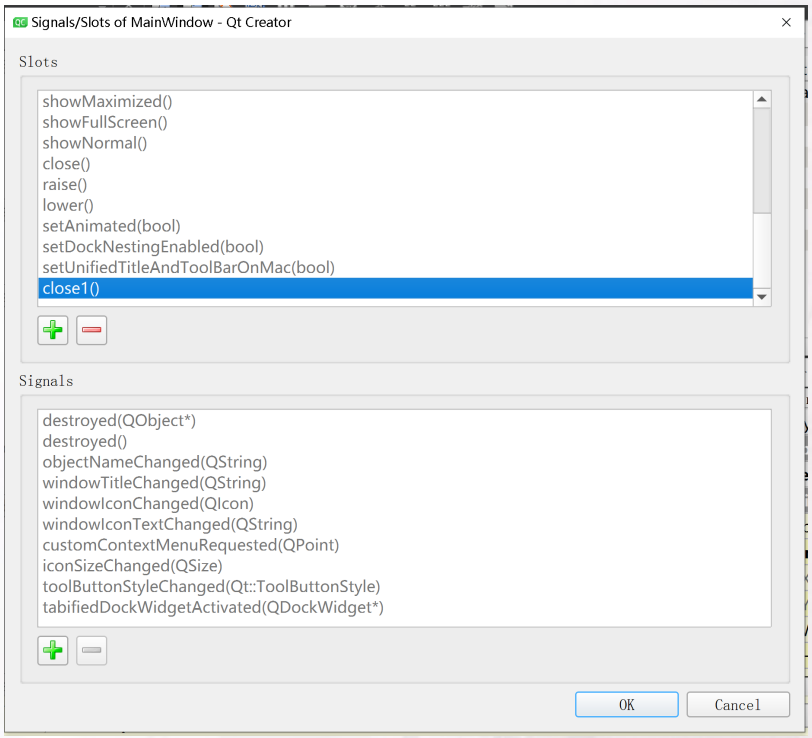


4



使用Qt设计器连接信号与槽

- 方法2：在设计器中拖动鼠标连接信号与槽
  - 可以对已有窗体类（QMainWindow的子类）添加新的槽函数
  - 在信号/槽编辑模式下，将鼠标从okButton拖动到窗体上，在弹出窗口中点击*Edit*，可添加新的槽，同时需要在代码中实现该槽函数

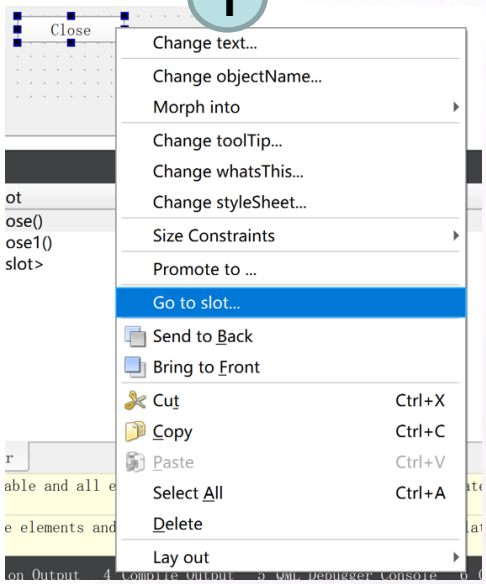



```
4   #include <QMainWindow>
5   #include <QApplication>
6
7   QT_BEGIN_NAMESPACE
8   namespace Ui { class MainWindow; }
9   QT_END_NAMESPACE
10
11  class MainWindow : public QMainWindow
12  {
13      Q_OBJECT
14
15  private slots:
16      void close1()
17      {
18          qApp->quit();
19      }
20
21  public:
22      MainWindow(QWidget *parent = nullptr);
23      ~MainWindow();
24      Ui::MainWindow *ui;
25  };
26  #endif // MAINWINDOW_H
27
```

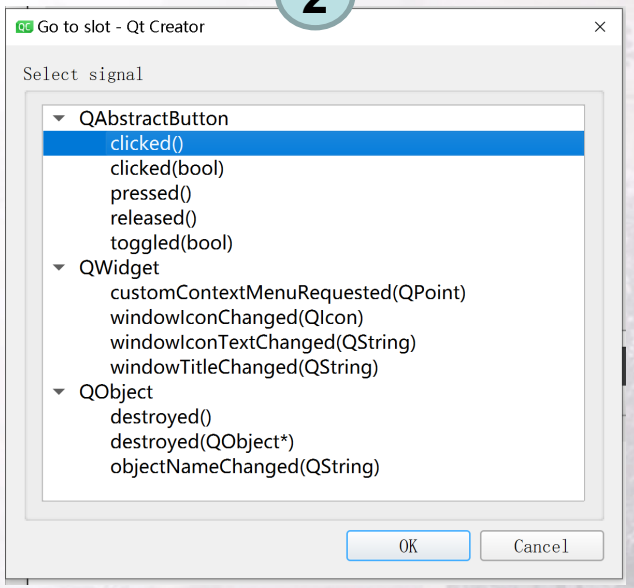
使用Qt设计器连接信号与槽

- 方法3：使用设计器+代码编写添加槽函数
  - 在*Edit Widgets*模式（F3）下右击okButton组件，选择转到槽(Go to slot)
  - 选择要该槽函数对应的信号
  - 设计器自动生成对应的槽函数代码模板，只需实现槽的函数体

1



2



3

.h文件中(自动)：

```
private slots:
    void on_closeButton_clicked(bool checked);
```

.cpp文件中：

```
void MainWindow::on_closeButton_clicked(bool checked)
{
    QApplication::exit(0);
}
```

纯代码实现信号与槽

- 信号
  - 在信号段(signals:)中定义
  - 总是返回空
  - 不必实现函数体
  - 使用*emit*关键字发射信号
  - 一个信号可以连接到任意数量的槽上，槽以任意次序被激发

```
signals:
    void aSignal();
```

```
emit aSignal();
```

- 槽
  - 在槽段（slots:）中定义
  - 当信号触发时执行槽，也可以直接调用槽函数
  - 类的成员函数，必须实现函数体
  - 可视为普通的类成员函数，可以是虚函数（virtual），可被重载，可以是public、protect或private
  - 可以传递任何类型的参数，可以使用默认参数
  - 多个信号可以连接到同一个槽

```
public slots:
    void aPublicSlot();
protected slots:
    void aProtectedSlot();
private slots:
    void aPrivateSlot();
```

纯代码实现信号与槽

- 手动定义信号和槽

```
class Student : public QObject
{
        Q_OBJECT                    ← 注意加入此行，否则signal & slot无效
public:
        Student() { myMark = 0; }
        int mark() const { return myMark; }
public slots:
        void setMark(int newMark);
signals:
        void markChanged(int newMark);
private:
        int myMark;
};
```

- 手动发射信号

```
emit markChanged(myMark);
```

纯代码连接信号与槽

- QObject::connect()函数实现信号与槽的连接
- 三种方法：

```
QPushButton *btn = new QPushButton;

// 方法一：老式写法
connect(btn, SIGNAL(clicked()), this, SLOT(close()));

// 方法二：Qt5后的新写法
connect(btn, &QPushButton::clicked, this, &MainWindow::close);

// 方法三：lambda表达式/functor
connect(btn, &QPushButton::clicked, this, [&]() {
    this->close();
});
```

纯代码连接信号与槽

- 第一种方法的函数原型

```
static QMetaObject::Connection connect(const QObject *sender,
const char *signal, const QObject *receiver, const char
*member, Qt::ConnectionType = Qt::AutoConnection);
```

- 注意：在编译时，即使信号或槽不存在也不会报错，但在执行时无效。对于C++这种静态类型语言来说，这是不友好的，不利于调试
- sender和receiver是QObject子类对象的指针
- SIGNAL()和SLOT()宏的作用是宏的参数转换成字符串
- 举例：

```
QLabel *label = new QLabel;
QScrollBar *scrollBar = new QScrollBar;
QObject::connect(scrollBar, SIGNAL(valueChanged(int)),
                 label,  SLOT(setNum(int)));
```

纯代码连接信号与槽

- 第二种方法的函数原型

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

- 这是Qt5后推荐的写法，如果编译时信号或槽不存在，则无法编译通过。
- 举例：

```
QLabel *label = new QLabel;
QLineEdit *lineEdit = new QLineEdit;
QObject::connect(lineEdit, &QLineEdit::textChanged,
                 label,  &QLabel::setText);
```

纯代码连接信号与槽

- 第三种方法的函数原型

```
inline QMetaObject::Connection connect(const QObject *sender,
const char *signal, const char *member, Qt::ConnectionType type
= Qt::AutoConnection) const;
```

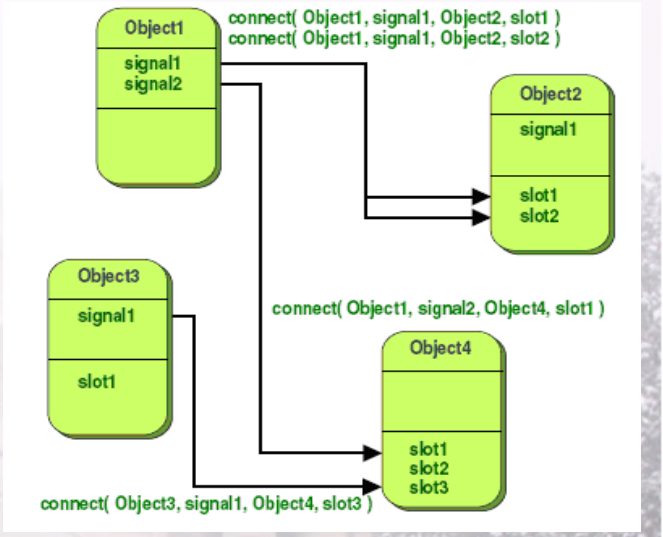
- 采用lambda表达式，更加方便快捷
- 举例：

```
QPushButton *quitButton = new QPushButton("Quit");
QObject::connect(quitButton, &QPushButton::clicked, qApp, [&]() {
            qApp->exit(0);
        });
```

信号与槽的连接：connect函数

- 多个信号可以连接到同一个槽

```
connect(lcd, SIGNAL(overflow()),
        this, SLOT(handleMathError()));
connect(calculator, SIGNAL(divisionByZero()),
        this, SLOT(handleMathError()));
```



- 一个信号可以连接到多个槽

```
connect(slider, SIGNAL(valueChanged(int)),
        spinBox, SLOT(setValue(int)));
connect(slider, SIGNAL(valueChanged(int)),
        this, SLOT(updateStatusBarIndicator(int)));
```

- 一个信号可以和另一个信号相连

```
connect(lineEdit, SIGNAL(textChanged(const QString &)),
        this, SIGNAL(updateRecord(const QString &)));
```

- 删除连接（不常用，因为对象删除后会自动删除其连接）

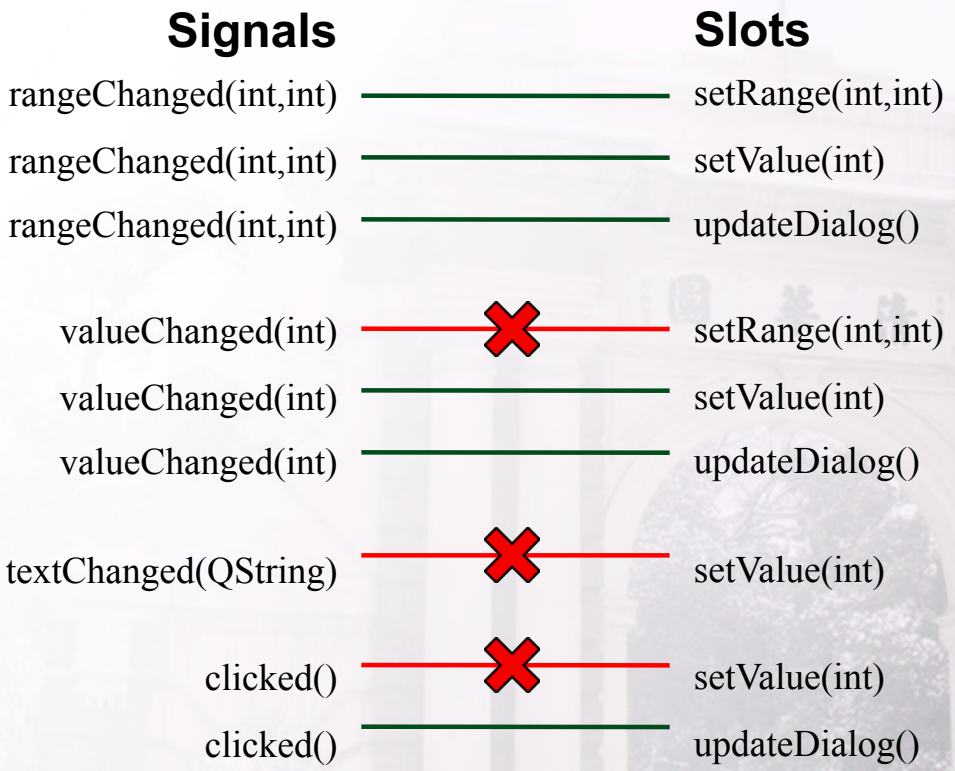
```
disconnect(lcd,SIGNAL(overflow()),this, SLOT(handleMathError()));
```

信号与槽的机制说明

- 信号与槽的执行
  - 如果信号和槽实现在同一个线程中，当信号发射的时候，与它关联的槽将马上执行
  - 如果信号和槽不在同一个线程中，槽的执行可能会有延迟(next event loop)
  - 只有当与信号关联的所有槽函数执行完毕后，才会执行发射信号处后面的代码
- 信号和槽连接必须满足的条件
  - 在定义信号与槽的类中，需要加入Q_OBJECT宏
  - 信号和槽函数必须具有相同的参数类型及顺序
  - 信号的参数可以多于槽的参数，多余的参数被忽略，反之则不行

信号与槽的机制说明

- 举例



基于Qt Designer的GUI界面设计

Qt图形用户界面（GUI）设计

- 使用Qt Designer
  - 直观且高效
  - 由于Qt Designer功能有限，可能无法支持特别的设计
- 纯代码GUI设计
  - 可以完成复杂的设计需求，但设计效率低，过程繁琐
  - 设计过程需要参考Qt文档或教程
- 可以采用混合方式进行GUI设计，部分模块使用Qt Designer，部分模块使用纯代码设计
- 基本流程：
  - 粗略放置部件在窗体上
  - 从里到外进行布局，添加必要的弹簧
  - 设计后台逻辑，进行信号连接
  - 在代码中使用Qt Designer设计的组件和纯代码设计其他组件

设计过程中需要不断调整
Practice make perfect!

自动生成的框架代码

- 使用Qt Designer设计.ui文件，会自动生成.h文件

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>

namespace Ui {
    class Widget;
}

class Widget : public QWidget {
    Q_OBJECT
public:
    Widget(QWidget *parent = 0);
    ~Widget();

private:
    Ui::Widget *ui;
};

#endif // WIDGET_H
```

Ui::Widget类的前置声明

Qwidget子类

Ui::Widget类指针ui，可以访问Widget上的所有部件

自动生成框架代码

- 使用Qt Designer设计.ui文件，会自动生成.cpp文件

```
#include "widget.h"
#include "ui_widget.h"

Widget::Widget(QWidget *parent) :
    QWidget(parent),
    ui(new Ui::Widget)
{
    ui->setupUi(this);
}

Widget::~Widget()
{
    delete ui;
}
```

调用函数setupUi，生成所有父窗体(this)上的部件

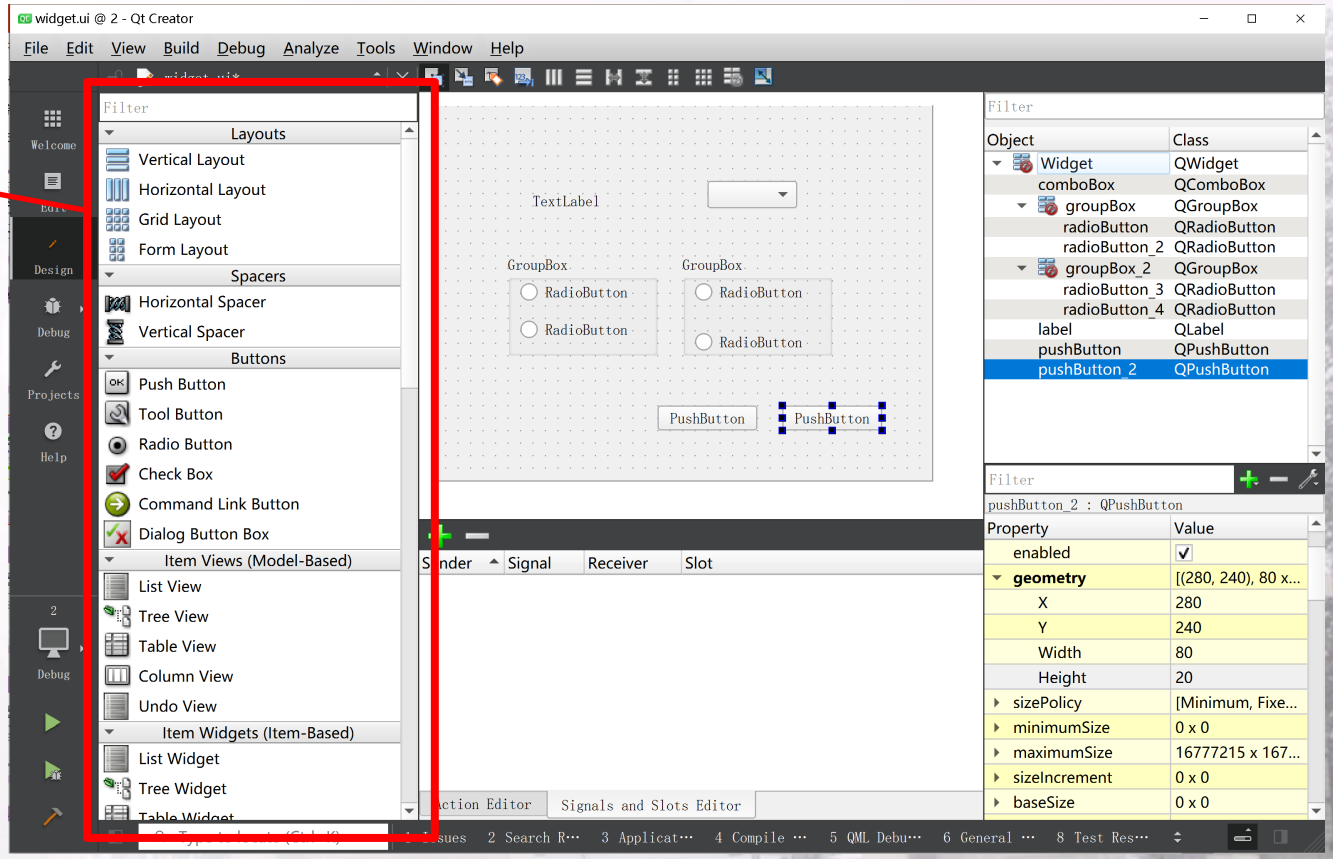
ui指向Ui::Widget的实例

释放ui对象

使用设计器实现GUI界面

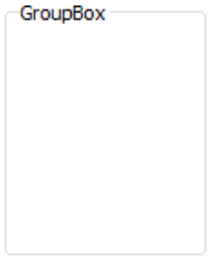
- 拖放不同部件到窗体上
- 在右下角属性编辑区设置属性值

Qt Designer提供的各种部件

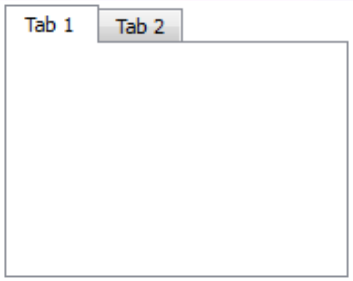


部件层次化

- 多个部件可以放在一个容器部件（如QGroupBox，QTabWidget等）中，达到分层次放置的效果



QGroupBox

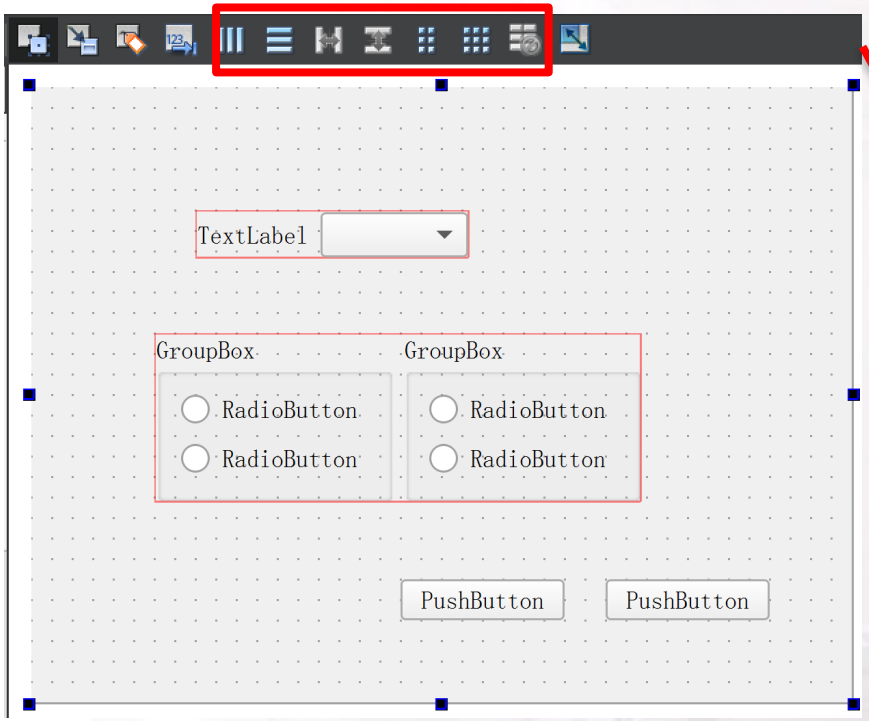


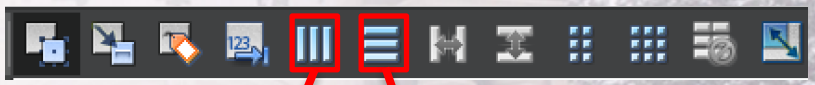
QTabWidget

- 容器类提供可视化结构，同时具有一定功能
  - 如可以将多个QRadioButton放到一个QGroupBox中，由此实现彼此间的互斥
- 部件的主要特点
  - 占据屏幕中一个矩形的区域
  - 从输入设备接收事件
  - 当部件产生变化时，发出信号
  - 一个部件中可以包含其他部件

使用设计器实现GUI界面

- 可以使用布局管理器自动设置部件的位置关系
- 例中的GroupBox可以使用垂直布局，而label和comboBox可以采用水平布局
- GroupBox间也可以使用水平布局



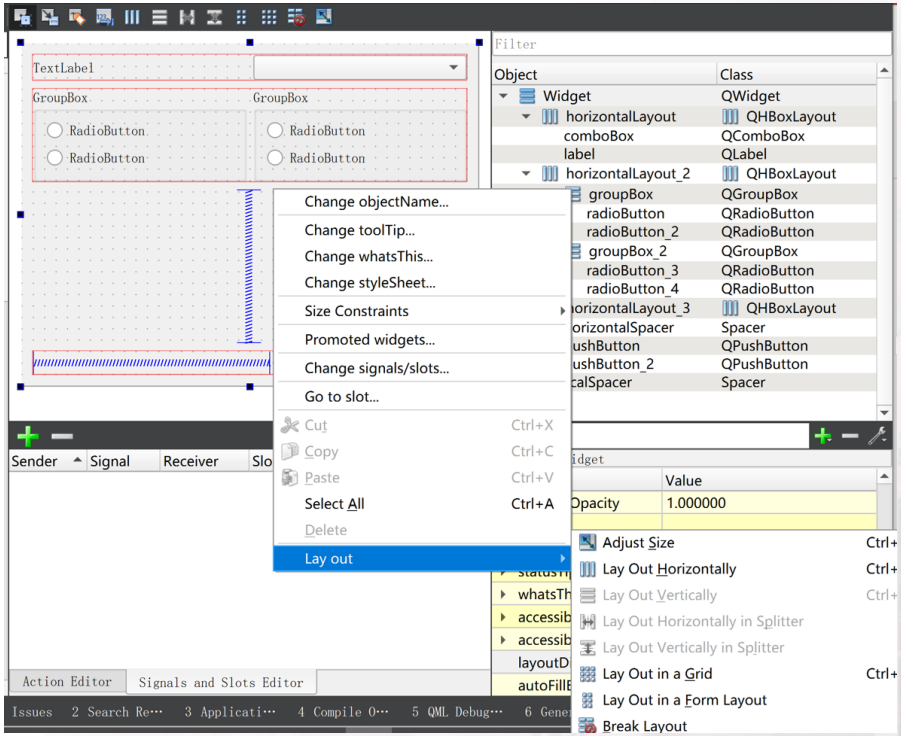


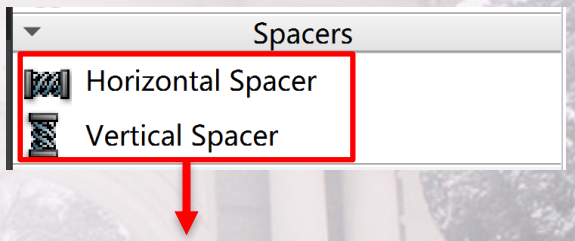
按Ctrl键选中多个部件，单击可应用水平布局

按Ctrl键选中多个部件，单击可应用垂直布局

使用设计器实现GUI界面

- 添加必要的弹簧，使部件具有弹性，可以伸缩
- 将弹簧与部件一起放入布局管理器中
- 右击窗口可以选择Layout，应用不同的布局管理

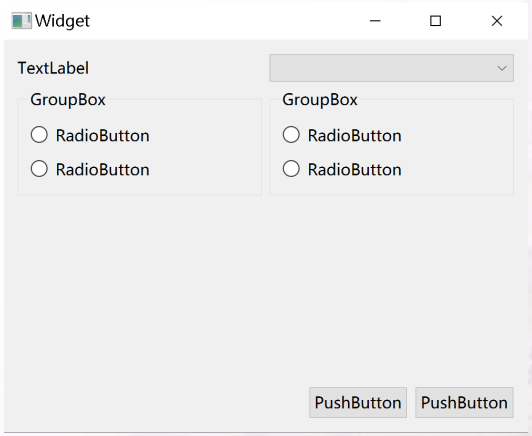




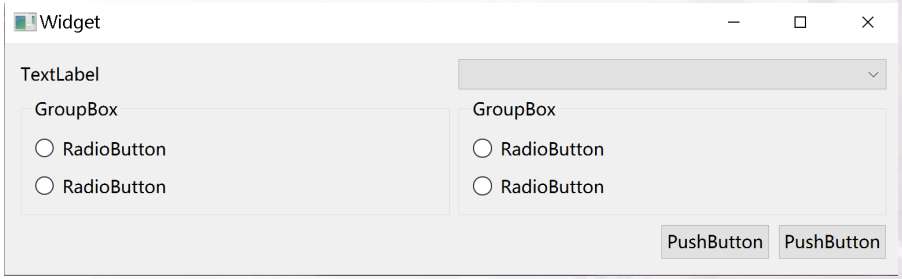
水平与垂直弹簧

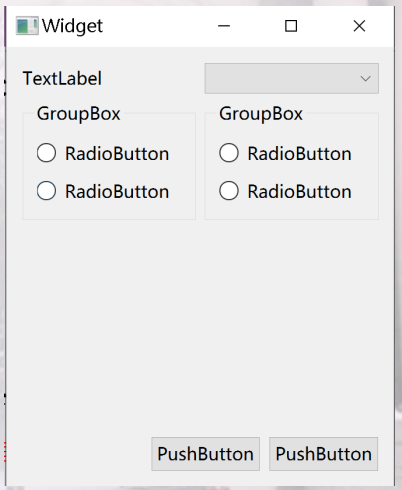
使用设计器实现GUI界面

- 界面设计结果



- 部件放置在布局管理器中的效果
  - 弹簧使界面具有弹性，易伸缩





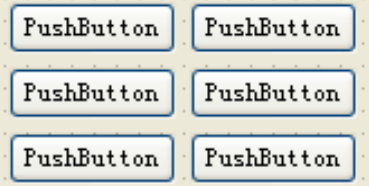
布局管理器总结

- 几种常用的布局

QVBoxLayout

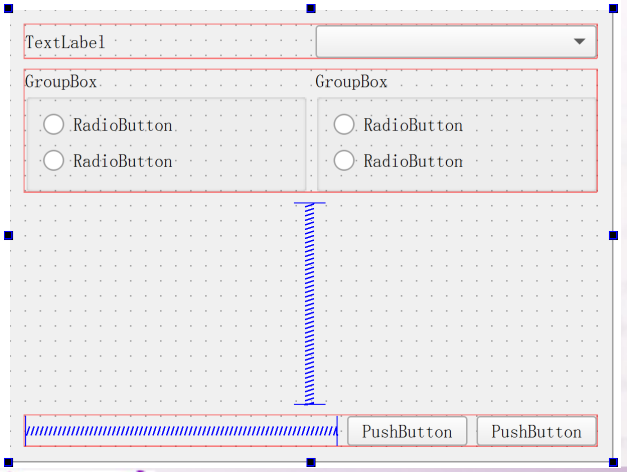
QHBoxLayout

QGridLayout



QFormLayout

- 布局管理器和部件“协商”各个部件大小与位置
- 弹簧可以用来填充空白处 
- 整个用户界面由多个层次的布局管理器和部件组成



Object	Class
Widget	QWidget
horizontalLayout	QHBoxLayout
comboBox	QComboBox
label	QLabel
horizontalLayout_2	QHBoxLayout
groupBox	QGroupBox
radioButton	QRadioButton
radioButton_2	QRadioButton
groupBox_2	QGroupBox
radioButton_3	QRadioButton
radioButton_4	QRadioButton
horizontalLayout_3	QHBoxLayout
horizontalSpacer	Spacer
pushButton	QPushButton
pushButton_2	QPushButton
verticalSpacer	Spacer

注意：布局管理器并不是其管理的部件的父对象

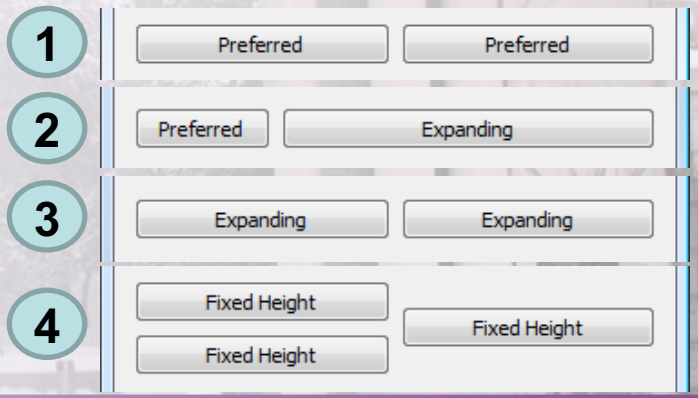
部件的尺寸策略（sizePolicy）

- 部件的布局过程是在布局管理器和部件间进行协调的过程
  - 布局管理器提供各种布局结构
  - 部件则提供各个方向上的尺寸策略
  - 在**Qt**设计器右下角的属性编辑器中，可以找到**sizePolicy**选项，双击每一栏可进行编辑

sizePolicy	[Minimum, Fixed, 0, 0]
Horizontal Policy	Minimum
Vertical Policy	Fixed
Horizontal Stret...	0
Vertical Stretch	0

部件的尺寸策略（sizePolicy）

- 每一个部件都有一个尺寸大小的示意（hint），给出水平和垂直方向上的尺寸策略
  - Fixed – 规定了widget的尺寸，固定大小（最严格）
  - Minimum – 规定了可能的最小值，可增长
  - Maximum – 规定可能的最大值，可缩小
  - Preferred – 给出最佳值，但不是必须的，可增长可缩小
  - Expanding – 同preferred，但希望增长
  - MinimumExpanding – 同minimum，但希望增长
  - Ignored – 忽略规定尺寸，widget得到尽量大的空间
- 举例
  - 2个 preferred 相邻
  - 1个 preferred, 1个 expanding
  - 2个 expanding 相邻
  - 空间不足以放置widget (fixed)



设置最大最小尺寸

- 可通过最大/最小尺寸更好地控制所有部件的大小
  - **maximumSize** – 最大可能尺寸
  - **minimumSize** – 最小可能尺寸

Property	Value
minimumSize	0 x 0
Width	0
Height	0
maximumSize	16777215 x 16777215
Width	16777215
Height	16777215

- 也可以使用代码直接修改

```
ui->pushButton->setMinimumSize(100, 150);
ui->pushButton->setMaximumHeight(250);
```

使用设计器实现GUI界面

- 在代码中访问界面上的部件
  - 可以通过ui指针访问每个子部件

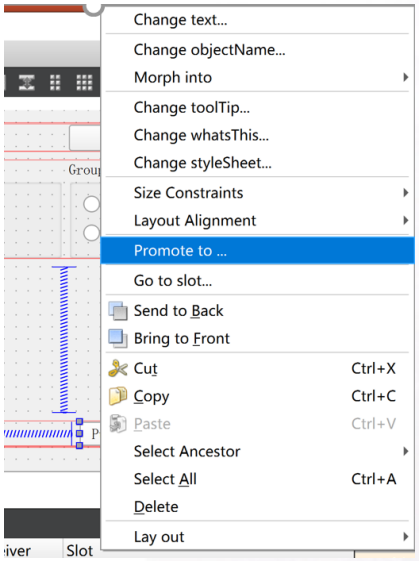
```
class Widget : public QWidget {
    ...
private:
    Ui::Widget *ui;
};
```

```
void Widget::memberFunction()
{
    ui->pushButton->setText(...);
}
```

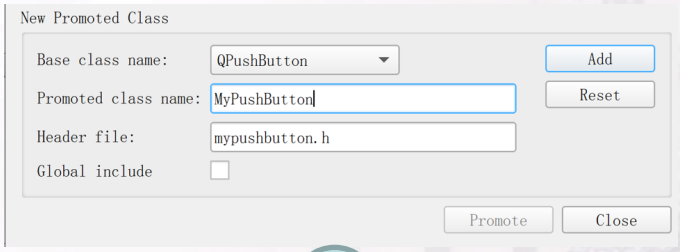
自定义部件-定义部件的子类

- 可以通过定义通用部件的子类，增加通用部件不具有的功能
- 在Qt Designer中，通过右键点击部件，选择*Promote to*（提升为），可以定义部件的子类
  - 输入所提升的类名称
  - 新建C++类 “MyPushButton”，设置基类为QPushButton

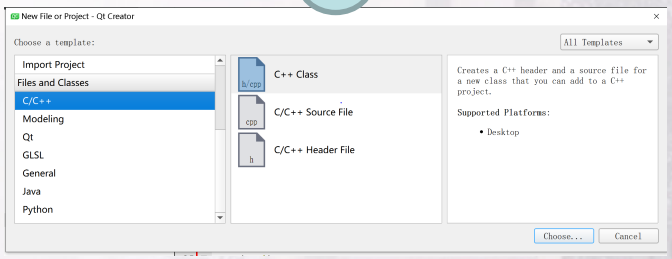
1



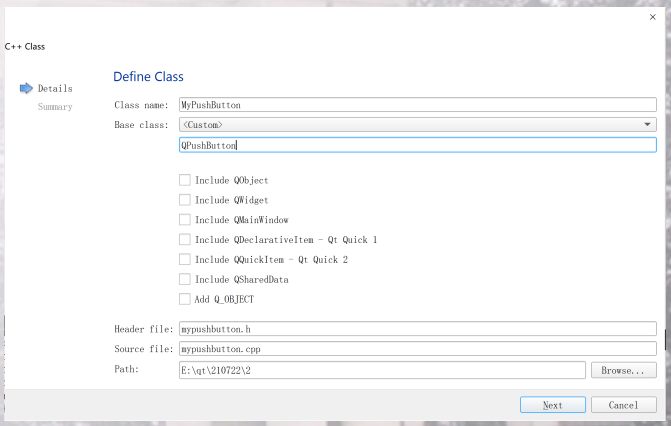
2



3



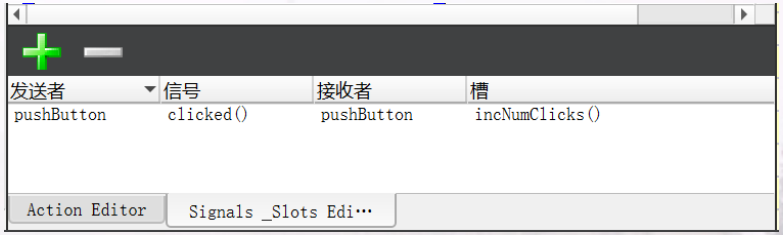
4



自定义部件-定义部件的子类

- 编写MyPushButton类，加入需要的功能（记录点击次数）
- 设置“信号/槽”的连接关系

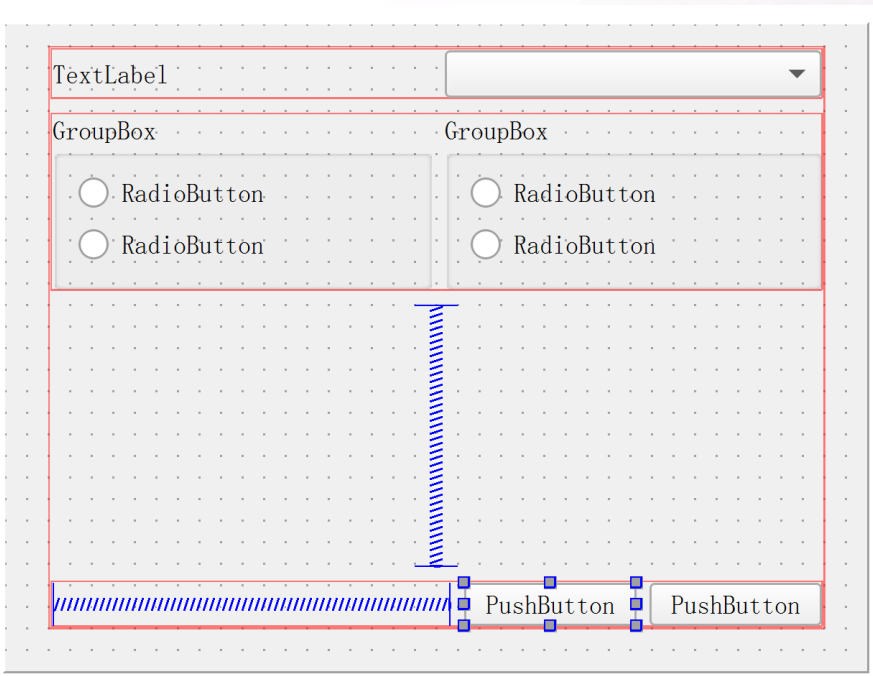
```
#ifndef MYPUSHBUTTON_H
#define MYPUSHBUTTON_H
#include<QPushButton>
class MyPushButton :
        public QPushButton {
    Q_OBJECT
public:
    MyPushButton(QWidget
*parent);
    int getNumClicks() {
        return numClicks_;
    }
public slots:
    void incNumClicks();
protected:
    int numClicks_;
};
#endif // MYPUSHBUTTON_H
```

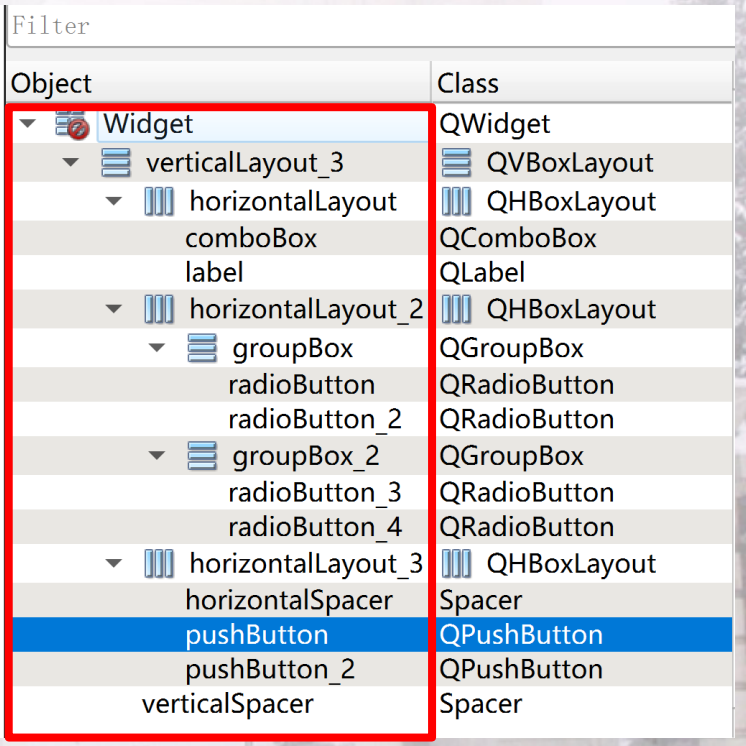



```
#include "mypushbutton.h"
MyPushButton::MyPushButton(QWidget
*parent): QPushButton(parent),
            numClicks_(0)
{
}
void MyPushButton::incNumClicks() {
  ++ numClicks_;
}
```

对象树

- 部件的层次化放置会产生一个对象树
- 每一个QObject子类的对象都可以设置一个指向父对象的指针
- 如果一个QWidget子类对象的父对象指针为空，那么它自身就是一个可以独立显示的窗口





对象树

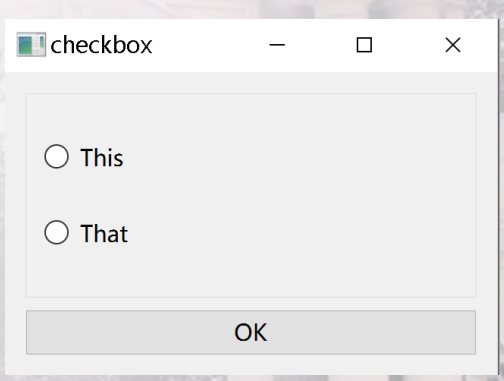
- 子对象会通知父对象自己的存在，父对象则把子对象加入到自己的孩子列表中
  - 当父部件隐藏/显示的时候，会自动隐藏/显示其子部件
    - 当父部件可见的时候，子部件也可以单独隐藏自己
  - 当父部件enable/disable时，子部件的状态也随之变化
- 所有子对象的内存管理都转移给了父对象
  - 使用new在堆上分配内存
  - 子对象可自动被父对象释放内存
  - 手动释放子对象不会引起二次释放，因为子对象释放时会通知父对象更新孩子列表
- 任何没有父对象的QObject子类对象都需要手动释放
  - 建议把这种无父对象的对象分配在栈上，避免内存泄露
  - Qt没有自动回收站的机制，只需关注对象的父子关系

对象树举例

- QObject子类的构造函数一般都带有一个父对象指针，通过该指针设置对象间的父子关系

```
QDialog *parent = new QDialog();
QGroupBox *box = new QGroupBox(parent);
QPushButton *button = new QPushButton(parent);
QRadioButton *option1 = new QRadioButton(box);
QRadioButton *option2 = new QRadioButton(box);

delete parent;
```

parent 释放 box 和 button
box 释放 option1 和 option2



顶层窗体

顶层窗体

- 没有父部件的部件自动成为一个独立显示的顶层窗体
  - QWidget – 普通窗体，通常无模式（ NonModal ）
  - QDialog – 对话框，通常期望用户输入，如OK，Cancel等
  - QMainWindow – 应用程序窗体，有菜单，工具栏，状态栏等

- QDialog 和 QMainWindow 继承自 QWidget

使用QWidget作为窗体

- 可使用Qt设计器设定窗体属性
  - windowTitle属性对应标题，可使用setWindowTitle()函数修改
  - windowModality属性对应不同模式，可使用setWindowModality()函数修改
    - NonModal – 所有窗体可立即使用
    - WindowModal – 父窗体阻塞
    - ApplicationModal – 所有其他窗体阻塞

Widget : QWidget

Property	Value
QObject	
objectName	Widget
QWidget	
enabled	☑
geometry	[(0, 0), 400 x 300]
sizePolicy	[Preferred, Preferred, 0, 0]
minimumSize	0 x 0
maximumSize	16777215 x 16777215
sizeIncrement	0 x 0
baseSize	0 x 0
palette	Inherited
font	A [SimSun, 9]
cursor	Arrow
mouseTracking	☐
tabletTracking	☐
focusPolicy	NoFocus
contextMenuPolicy	DefaultContextMenu
acceptDrops	☐
windowTitle	Widget
windowIcon	
windowOpacity	1.000000
toolTip	

使用QWidget作为窗体

- QWidget构造函数

QWidget::QWidget(QWidget *parent, Qt::WindowFlags f=0)

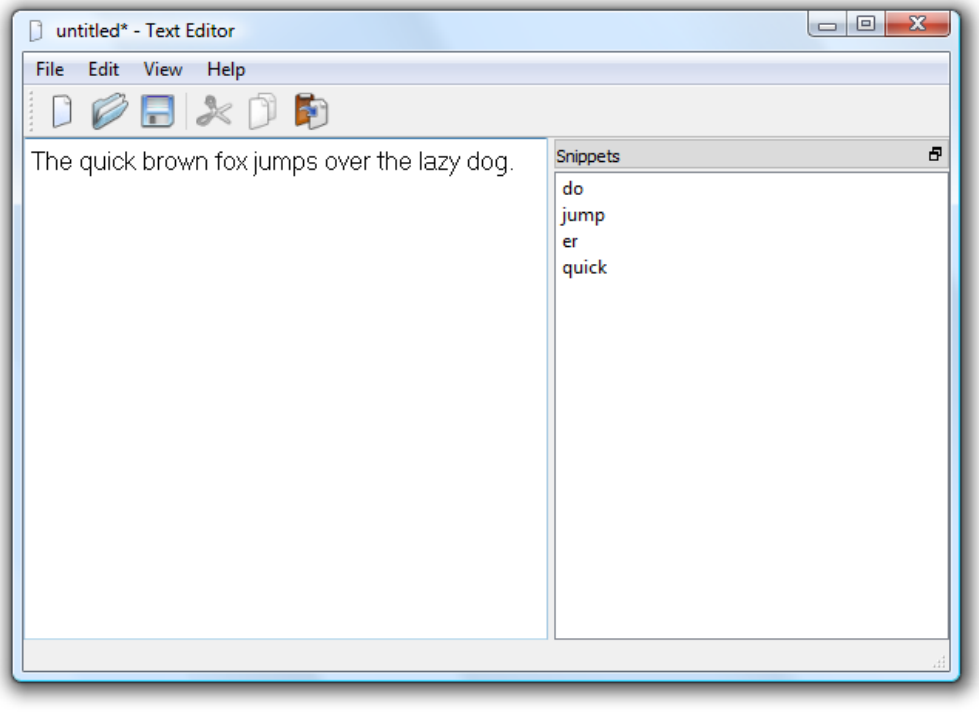
  - Qt::WindowFlags是窗体标志组合
    - 窗体类型：如Qt::Window (生成窗口), Qt::Dialog (生成对话框)
    - 窗体的Hints: 如Qt::WindowMinimizeButtonHint Qt::WindowCloseButtonHint (添加关闭按钮)

- 继承自QWidget的窗体子类
  - QSplashScreen：应用程序启动时的splash窗口，无边框
  - QMdiSubWindow：MDI（多文档）应用程序
  - QDesktopWidget：提供用户桌面信息，如屏幕个数、大小等
  - QWindow：用于底层的窗口系统，不作为独立窗体，一般作为一个父容器的嵌入式窗体

使用QMainWindow

- QMainWindow 是普通桌面程序的文档窗体

- 菜单栏
- 工具栏
- 状态栏
- 停靠窗体
- 中心部件

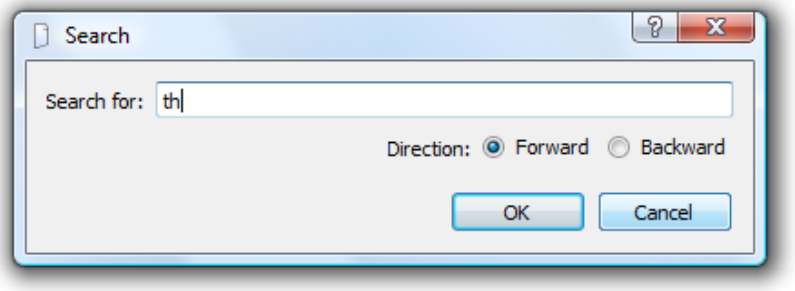


使用QDialog

- QDialog 提供标准对话框
- QDialog的子类
  - QFileDialog：文件对话框，提供打开/保存文件等功能
    - getOpenFileName() / getOpenFileNames()
    - getSaveFileName()
    - getExistingDirectory()
  - QColorDialog：颜色对话框，选择颜色
    - getColor()
  - QFontDialog：字体对话框，选择字体
    - getFont()
  - QInputDialog：输入对话框，可以输入文字、数字、字符串等
    - getText() / getInt() / getDouble() / getItem() / getMultiLineText()
  - QMessageBox：消息提示框，信息提示、是否确认、警告等
    - Information() / question() / warning() / critical() / about()

示例：基于QDialog自定义对话框

- 继承QDialog定义搜索对话框



- 使用Qt设计器或纯代码建立用户界面
- 通过QLineEdit和QRadioButton获取用户输入
- 包括OK，Cancel按钮

类的定义

```
class SearchDialog : public QDialog
{
    Q_OBJECT
public:
    explicit SearchDialog(const QString &initialText,
                          bool isBackward, QWidget *parent = 0);

    bool isBackward() const;
    const QString &searchText() const;

private:
    Ui::SearchDialog *ui;
};
```

在构造函数中初始化对话框

Getter 函数获取数据

类的实现

```
SearchDialog::SearchDialog(const QString &initialText,
                           bool isBackward, QWidget *parent) :
                       QDialog(parent), ui(new Ui::SearchDialog)
{
    ui->setupUi(this);

    ui->searchText->setText(initialText);
    if(isBackward)
        ui->directionBackward->setChecked(true);
    else
        ui->directionForward->setChecked(true);
}

bool SearchDialog::isBackward() const
{
    return ui->directionBackward->isChecked();
}

const QString &SearchDialog::searchText() const
{
    return ui->searchText->text();
}
```

初始化对话框

getter函数

使用自定义对话框

- 在代码中定义对话框对象，并执行对话框
  - 需要自行确保头文件的包含关系正确

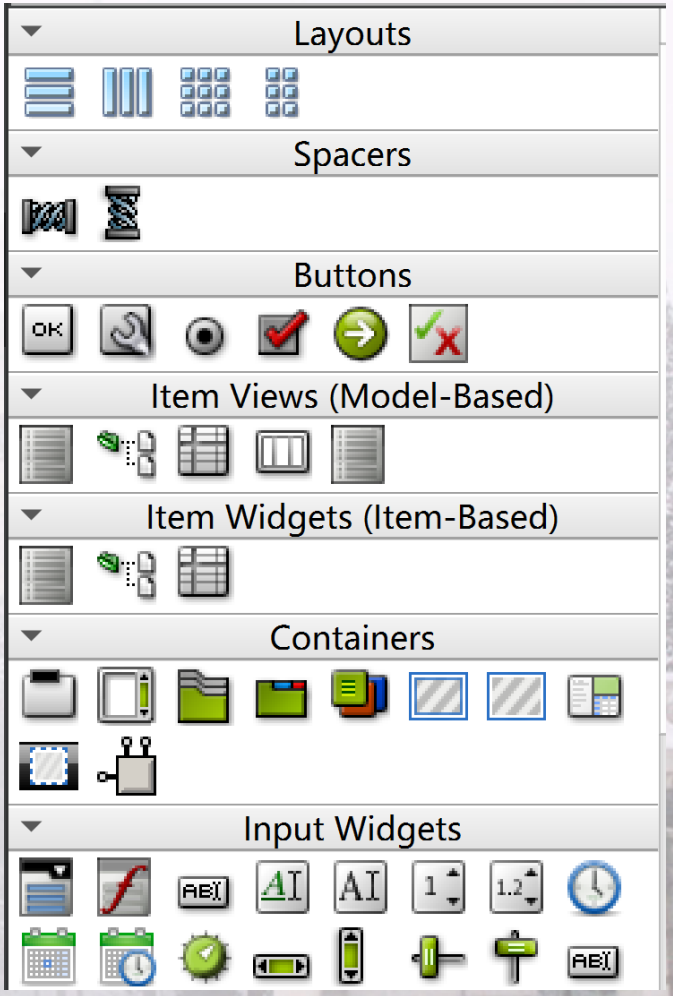
```
void MyWindow::myFunction()
{
    SearchDialog dlg(settings.value("searchText","").toString(),
                     settings.value("searchBackward", false).toBool(), this);
    if(dlg.exec() == QDialog::Accepted)
    {
        QString text = dlg.searchText();
        bool backwards = dlg.isBackward();
    }
}
```

QDialog::exec显示一个阻塞模式对话框，如果用户点击OK按钮，则返回QDialog::Accepted（需要设置正确的信号槽连接）

Qt 通用部件

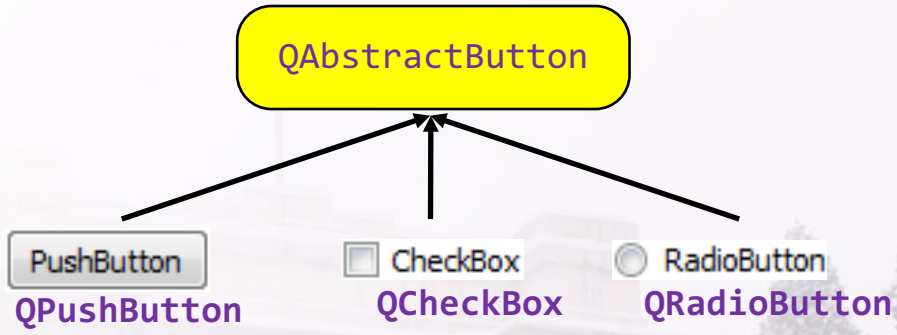
Qt通用部件

- Qt包含常见设计需求的大量通用部件
- Qt设计器为部件使用提供良好支持



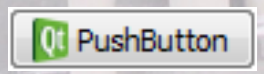
通用部件—按钮

- 所有按钮都继承自 **QAbstractButton类**



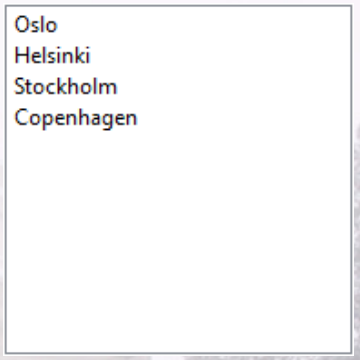
- 信号
  - **clicked() –** 当按钮被按下（并弹起后）发出。
  - **toggled(bool) –** 当按钮的状态发生改变时发出。

- 属性
  - **checkable –** 当按钮可标记时为真，使按钮激活。
  - **checked –** 当按钮被标记时为真（用于复选或单选按钮）
  - **text –** 按钮的文本。
  - **icon –** 按钮的图标（可以和文本同时显示）。

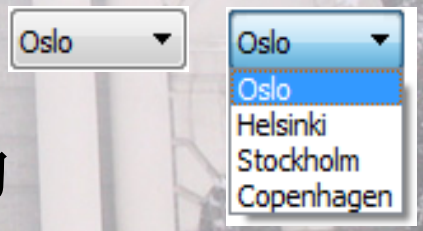


通用部件—列表项部件

- QListWidget用于显示列表项
- 添加项目
  - addItem(QString) – 将项目附加到列表末端
  - insertItem(int row, QString) – 将项目插入到指定行
- 选择项目
  - selectedItems – 返回QListWidgetItem的列表, 使用QListWidgetItem::text取得文本
- 信号
  - itemSelectionChanged – 当选择状态改变时发出



QListWidget

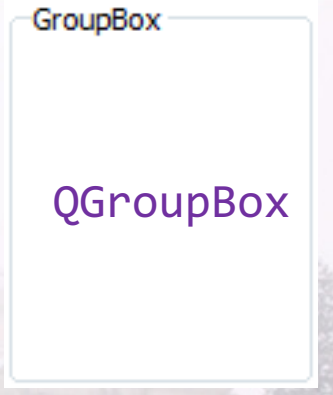


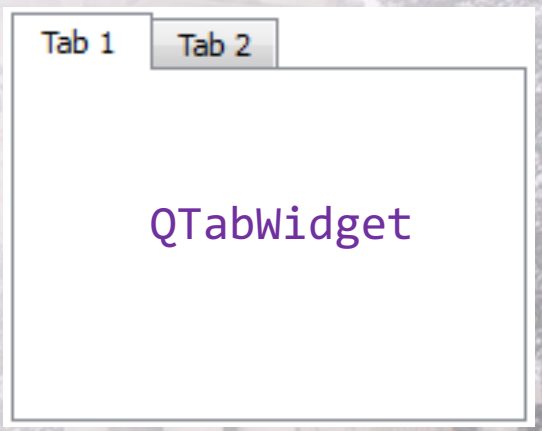
- QComboBox 以更紧密的格式展示一个单选的项目列表。

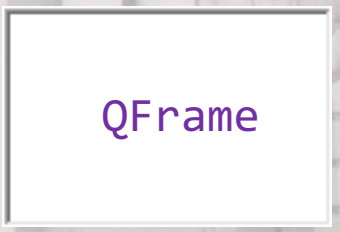
QComboBox

通用部件—容器

- 容器部件可以层次化（结构化）界面
- 一个简单的 **Qwidget**子类 对象也可当做容器来使用
- 使用**Qt**设计器：将部件放置在容器中，并为容器提供一个布局管理器
- 使用代码：为容器创建一个布局管理器，并将部件添加进布局管理器（布局管理器以容器为父对象）



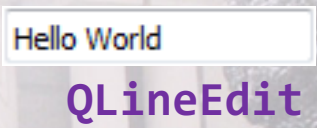





```
QGroupBox *box = new QGroupBox();
QVBoxLayout *layout = new QVBoxLayout(box);
layout->addWidget(...);
```

通用部件—输入部件

- 使用QLineEdit 实现单行文本输入
- 信号
  - textChanged(QString) – 文本状态改变时发出
  - editingFinished() – 部件失去焦点时发出
  - returnPressed() – 回车键被按下时发出
- 属性
  - text – 部件的文本
  - maxLength – 限定输入的最大长度
  - readOnly – 设置为真时文本不可编辑（允许复制）

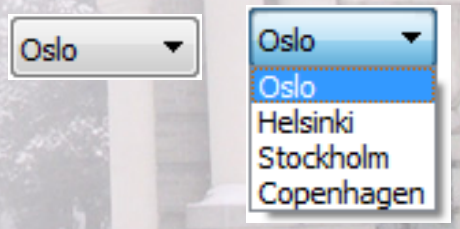


通用部件—输入部件

- 使用QTextEdit和QPlainTextEdit实现多行文本输入
  - 信号
    - textChanged() -文本状态改变时发出
  - 属性
    - plainText – 无定义格式文本
    - html – HTML格式文本
    - readOnly – 设置为真时文本不可编辑

QTextEdit

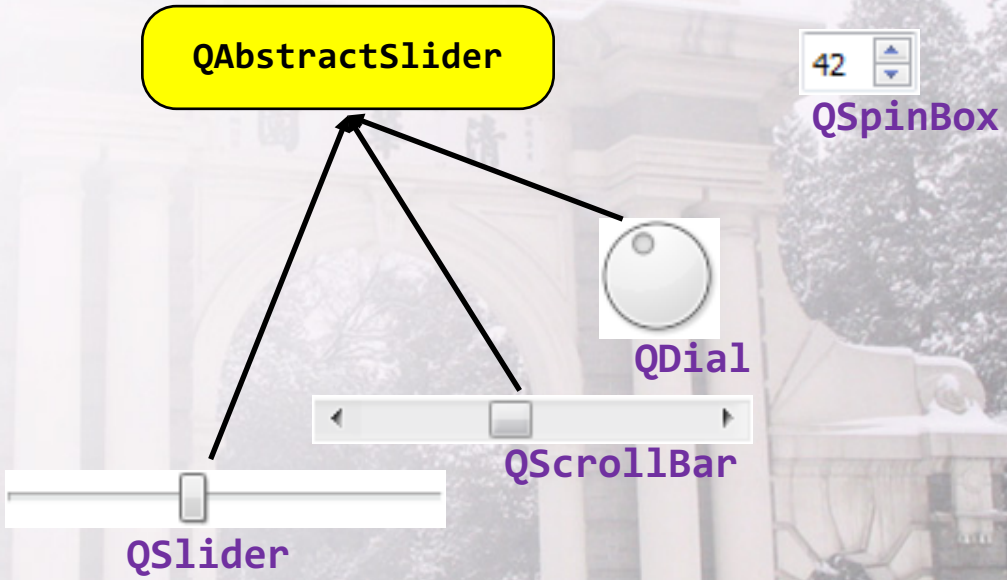
- QComboBox
  - 信号
    - editTextChanged(QString) – 当文本正被编辑时发出
  - 属性
    - currentText – combo box的当前文本



QComboBox

通用部件—输入部件

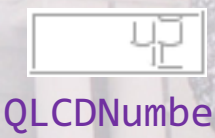
- 编辑整型数据有许多可选的输入部件
- 信号
  - valueChanged(int) – 当数值更新时发出
- 属性
  - value – 当前值
  - maximum – 最大值
  - minimum – 最小值



通用部件—显示部件

- QLabel 部件显示文本或者图片
  - 属性
    - text – 标签文本
    - pixmap – 显示的图片

- QLCDNumber 用于显示整型数值
  - 属性
    - intValue – 显示的数值

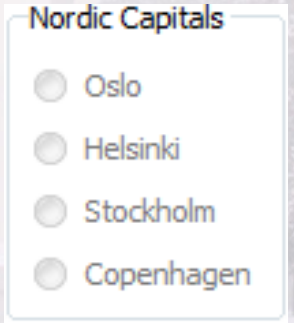


通用部件—属性

- 各个部件都有继承自QWidget类的共同属性
  - enabled – 用户交互可用或不可用

  - visible – 显示或不显示(show 或hide函数)

- 这些属性的设置，可以影响到子部件
  - 举例，使一个容器部件不可用时，容器内的RadioButton都不可选



基于纯代码的GUI界面设计

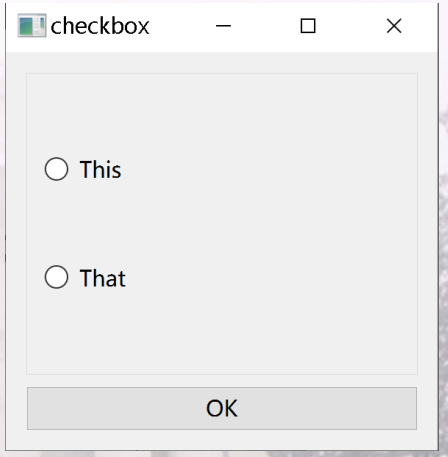
基于纯代码的界面设计

- 在创建项目时，不勾选*Generate form*复选框
  - 由此可以取消窗体创建，不生成.ui文件
- 基于代码的GUI设计中，信号与槽的连接需要通过代码来完成
  - 一般在Widget构造函数的函数体内，在setupUi函数调用之后，调用connect函数进行信号槽连接
- 对于初学者，采用纯代码的编写方式工作量较大，一般不推荐

复选框设计示例

widget.h

```
#ifndef WIDGET_H
#define WIDGET_H
#include <QWidget>
#include <QGroupBox>
#include <QPushButton>
#include <QRadioButton>
#include <QApplication>
#include <QLayout>

class Widget : public QWidget
{
    Q_OBJECT
private:
    QVBoxLayout *Vlay1, *Vlay2;
    QGroupBox *box;
    QPushButton *button;
    QRadioButton *option1;
    QRadioButton *option2;
private slots:
    void onButtonClicked(bool checked);
public:
    Widget(QWidget *parent = nullptr);
    ~Widget();
};
#endif // WIDGET_H
```

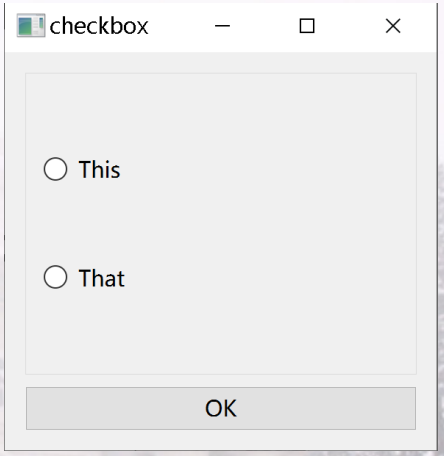


设计结果

复选框设计示例

widget.cpp

```
#include "widget.h"
Widget::Widget(QWidget *parent)
    : QWidget(parent)
{
    Vlay1 = new QVBoxLayout(this);
    box = new QGroupBox(this);
    Vlay2 = new QVBoxLayout(box);
    button = new QPushButton(this);
    option1 = new QRadioButton(box);
    option2 = new QRadioButton(box);
    Vlay1->addWidget(box);
    Vlay1->addWidget(button);
    Vlay2->addWidget(option1);
    Vlay2->addWidget(option2);
    button->setText("OK");
    option1->setText("This");
    option2->setText("That");
    setWindowTitle("checkbox");
}
void Widget::onButtonClicked(bool checked) {
    qApp->quit();
}
Widget::~Widget() {}
```



设计结果

元对象系统

元对象系统

- Qt元对象系统（Meta-Object System）
  - 提供对象之间通信的信号/槽机制、运行时类型信息、动态属性系统
  - 所有使用元对象系统的类必须继承自QObject，同时需要在类的private部分声明Q_OBJECT宏
  - MOC（元对象编译器）为每个QObject子类提供必要的代码来实现元对象系统的特性
    - 构建项目时，MOC读取C++源文件，当发现类的定义有Q_OBJECT宏时，就会为这个类生成另外一个包含有元对象支持代码的C++源文件，这个生成的源文件与类的其他源文件一起被编译和连接

元对象系统

- 元对象系统的主要用法
  - QObject::metaobject()函数返回类关联的元对象

```
Qobject *obj = new QPushButton;
obj->metaobject()->className();    //返回 "QPushButton"
```

  - QMetaObject::newInstance()函数创建类的一个新实例
  - QObject::inherits(const char *className)函数判断一个对象实例是否为className子类的实例
  - QObject::tr()和QObject::trUtf8()函数可翻译字符串
  - QObject::setProperty()和QObject::property()函数用于通过属性名称动态设置和获取属性值
  - 使用qobject_cast()函数进行动态类型转换

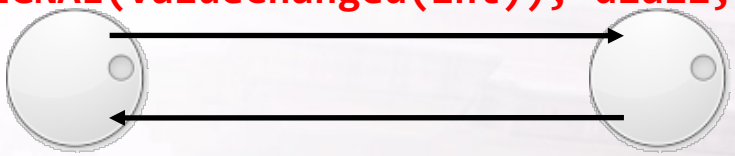
```
//假设QMyWidget是QWidget的子类且在类定义中声明了Q_OBJECT宏
QObject *obj = new QMyWidget;
QWidget *widget = qobject_cast<QWidget *>(obj);   //转换成功
QMyWidget *widget = qobject_cast<QMyWidget *>(obj);   //成功
QLabel *label = qobject_cast<QLabel *>(obj); //转换失败，返回NULL
```

信号与槽的应用实例

值同步

- 双向连接

```
connect(dial1, SIGNAL(valueChanged(int)), dial2, SLOT(setValue(int)));
```



```
connect(dial2, SIGNAL(valueChanged(int)), dial1, SLOT(setValue(int)));
```

- 注意避免**无限循环问题**：只有值发生变化时，才发射信号

```
void QDial::setValue(int v)
{
    if(v==m_value)
        return;
    emit valueChanged(v);
}
```

信号与值关联

- 对于数字键盘应用，如何把按键与发射的信号连接起来？
- 注意：按钮上的数字是按钮文本，无法自动关联到值

信号与值关联

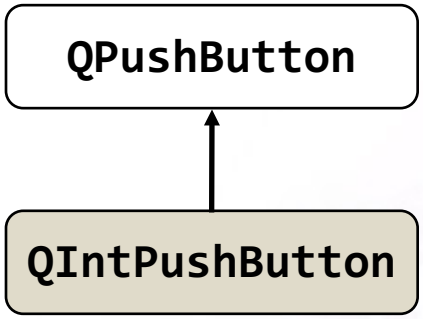
- 解决方法 #1: 每个按钮对应一个槽

connections

```
{
    ...

public slots:
    void key1Pressed();
    void key2Pressed();
    void key3Pressed();
    void key4Pressed();
    void key5Pressed();
    void key6Pressed();
    void key7Pressed();
    void key8Pressed();
    void key9Pressed();
    void key0Pressed();

    ...
}
```

信号与值关联

- 解决方法 #2: 自定义按钮子类，并增加信号




```
{
    ...

signals:
    void clicked(int);

    ...
}
```

```
{
    QIntPushButton *b;

    b=new QIntPushButton(1);
    connect(b, SIGNAL(clicked(int)),
        this, SLOT(keyPressed(int)));

    b=new QIntPushButton(2);
    connect(b, SIGNAL(clicked(int)),
        this, SLOT(keyPressed(int)));

    b=new QIntPushButton(3);
    connect(b, SIGNAL(clicked(int)),
        this, SLOT(keyPressed(int)));

    ...
}
```

解决方案评价

- #1: 每个按钮对应一个槽
  - 许多槽函数造成代码冗余
  - 难于维护 (一个小的变化影响所有槽)
  - 难于扩展 (每次都要新建槽函数)
- #2:自定义按钮子类
  - 需要定义专用类 (难于复用)
  - 难于扩展 (对于新的需求可能需要新建子类)

方案#3：用lambda函数

```
MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);

    connect(ui->pushButton, &QPushButton::clicked, [=](){ emit
keyPressed(1); });
    connect(ui->pushButton_2, &QPushButton::clicked, [=](){ emit
keyPressed(2); });

……

connect(ui->pushButton_8, &QPushButton::clicked, [=](){ emit
keyPressed(9); });
    connect(ui->pushButton_9, &QPushButton::clicked, [=](){ emit
keyPressed(0); });

    connect(this, &MainWindow::keyPressed, this,
&MainWindow::getKeyValue);
}
```

实例讲解
温度转换器

谢谢！

Q&A？