



# Qt基础

华清远见

# 版权声明

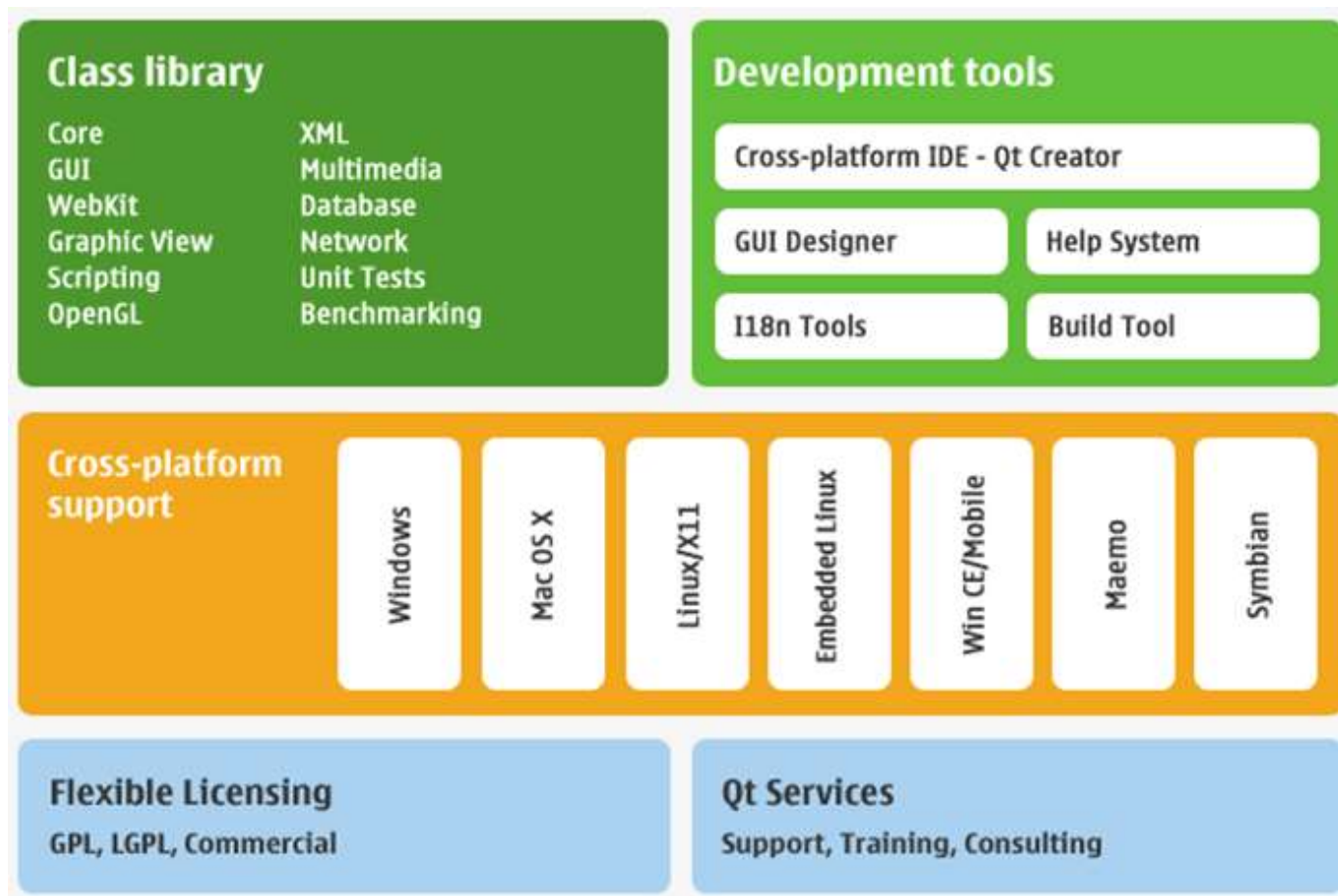
- 华清远见教育集团版权所有；
- 未经华清远见明确许可，不得为任何目的以任何形式复制或传播此文档的任何部分；
- 本文档包含的信息如有更改，恕不另行通知；
- 华清远见教育集团保留所有权利。

# 课程大纲

- ▶ Qt 简介
- ▶ Qt 应用范围
- ▶ 开发环境
- ▶ 第一个Qt程序
- ▶ 信号与槽

# Qt 简介

- ▶ Qt是挪威Trolltech开发的多平台C++图形用户界面应用程序框架



- ▶ 1996 Sep 24 Qt1.0
- ▶ 1996 Oct KDE 组织成立
- ▶ 1998 Apr 05 Trolltech 的程序员在5天之内将Netscape5.0从Motif移植到Qt上
- ▶ 1998 Apr 08 KDE Free Qt 基金会成立
- ▶ 1998 Jul 12 KDE 1.0 发布
- ▶ 1999 Jun 25 Qt 2.0 发布
- ▶ 2000 Mar 20 嵌入式 Qt 发布
- ▶ 2000 Sep 04 Qt free edition 开始使用 GPL
- ▶ 2008 Aug 4.4发布, 集成Webkit和Phonon
- ▶ 2009 May 11 Qt源代码管理系统面向公众开放
- ▶ 2012 Aug 9 Digia宣布已完成对诺基亚Qt业务及软件技术的全面收购, 并计划将Qt应用到Android、iOS及Windows 8平台上。
- ▶ 2014 May 20 QT 5.3发布

## ▶ 优良的跨平台特性

- ▶ Qt支持下列操作系统：Windows、Linux、Solaris、SunOS、FreeBSD、BSD/OS、SCO、AIX、OS390、QNX、android等等。

## ▶ 面向对象

- ▶ Qt的良好封装机制使得Qt的模块化程度非常高，可重用性较好，对于用户开发来说是非常方便的。Qt提供了一种称为signals/slots的安全类型来替代callback，这使得各个元件之间的协同工作变得十分简单。

## ▶ 丰富的API

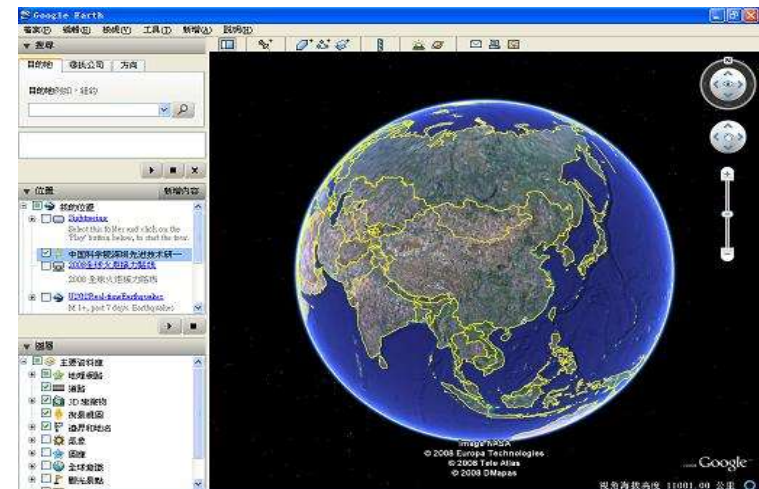
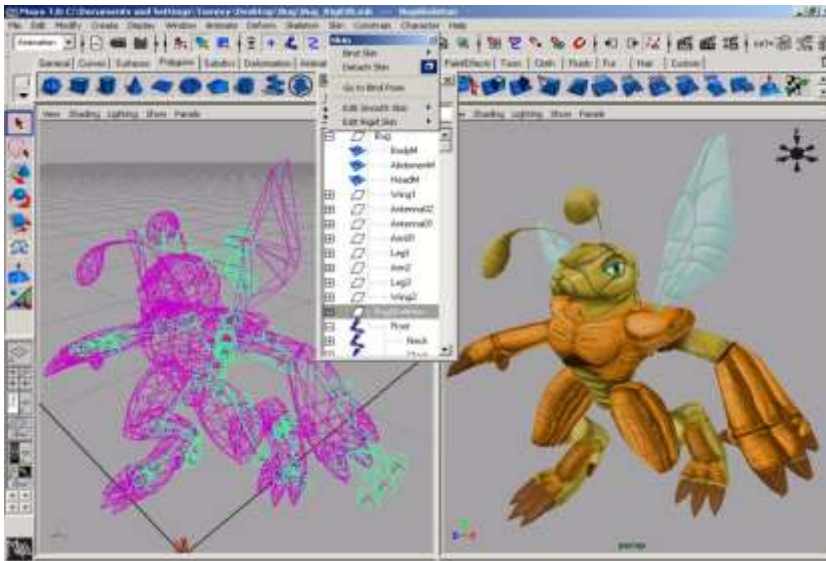
- ▶ Qt包括多达500个以上的C++类，还替供基于模板的collections，serialization，file，I/O device，directory management，date/time类。甚至还包括正则表达式的处理功能。

## ▶ 大量的开发文档

## ▶ Network/XML/OpenGL/Database/webkit/...

# Qt 应用范围

- ← KDE
- ← Maya
- ← Google earth
- ← Opera浏览器
- ← Skype网络电话
- ← QCad
- ← Adobe Photoshop Album
- ← CGAL计算几何库
- ← .....



## Qt 应用范围

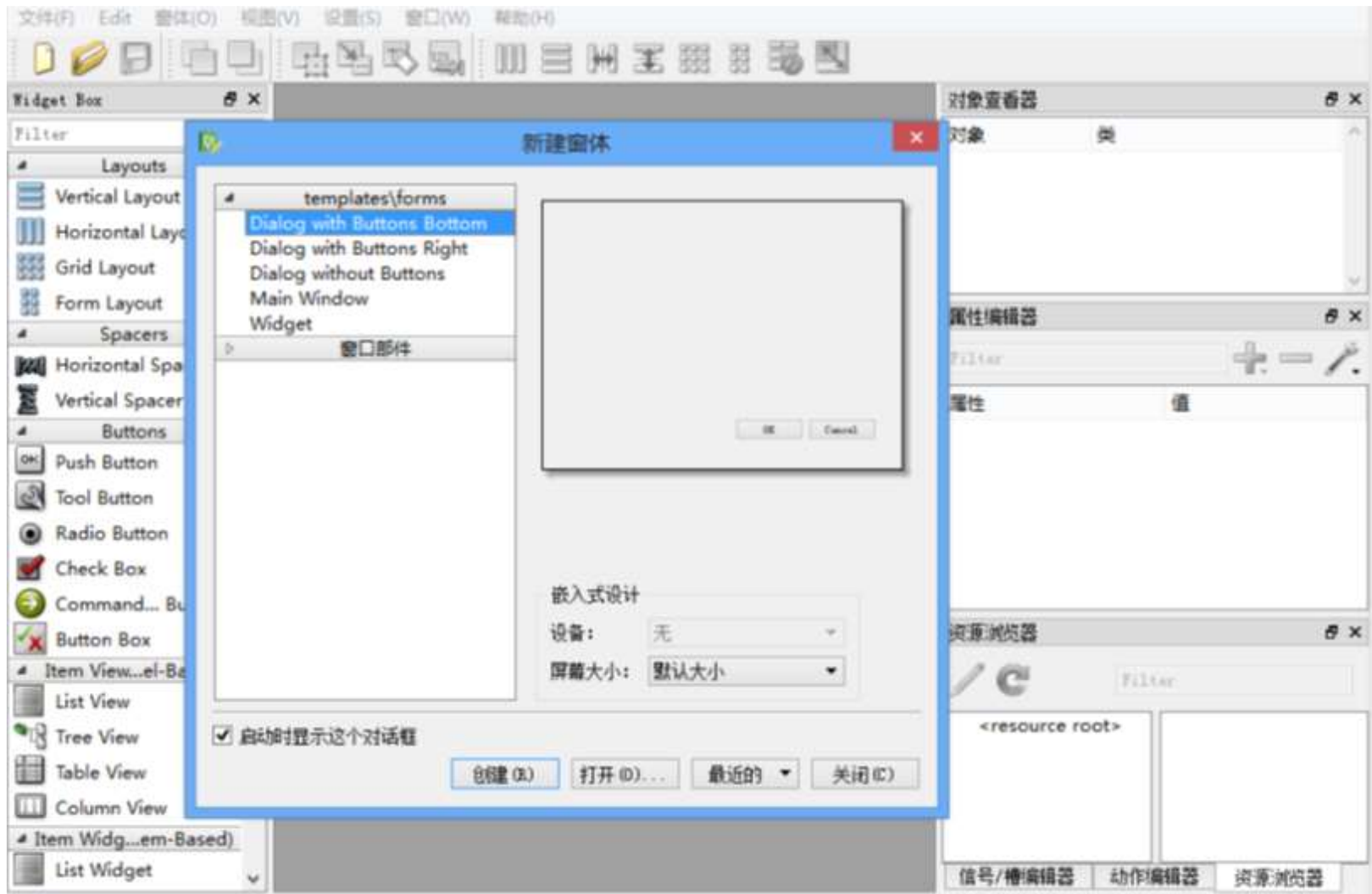
---

- ▶ Qt 在石油， 天然气行业的应用
- ▶ Qt 在家庭媒体中的应用
- ▶ Qt 在ip 通信的使用
- ▶ Qt 在虚拟测试当中的应用
- ▶ Qt 在仿真方面的应用

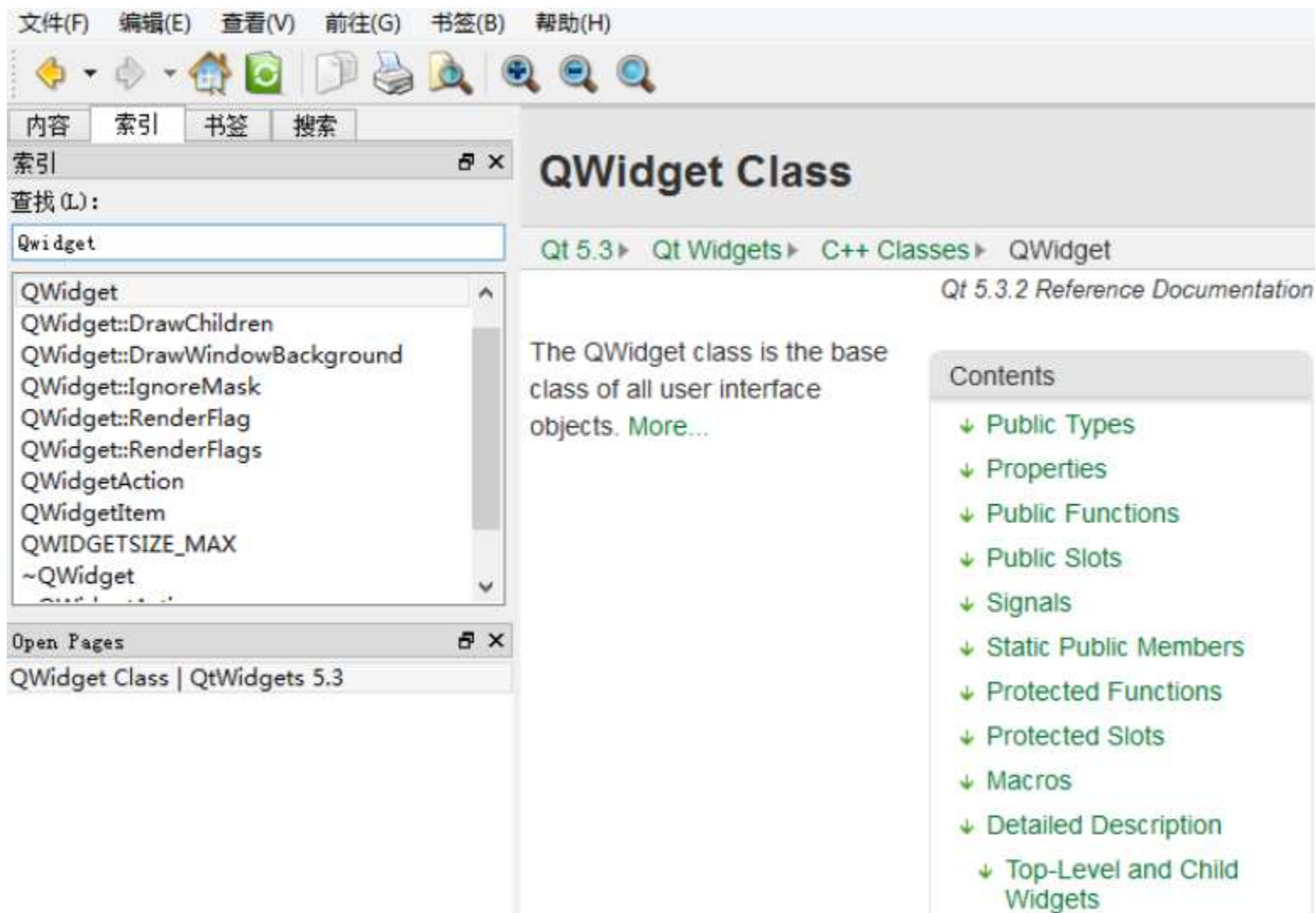


- ▶ 可视化工具：Qt Designer、Qt Assistant、Qt Linguist、Qt Creator
  - ▶ Qt Designer：界面设计编辑工具
  - ▶ Qt Assistant：Qt技术文档文档浏览器
  - ▶ Qt Linguist：国际化语言翻译工具
  - ▶ Qt Creator：集成开发环境（IDE）
- ▶ 命令行程序：lupdate、lrelease、qmake、uic、moc
  - ▶ lupdate：根据项目文件（.pro）读取源码中需要翻译的内容生成翻译文件（.ts）
  - ▶ lrelease：把翻译文件（.ts）翻译成二进制的.qm文件
  - ▶ qmake：Makefile生成器，能根据工程文件（.pro）产生不同平台下Makefile
  - ▶ uic：把ui文件（xml语法的界面文件）自动生成相应的C++代码
  - ▶ moc：元对象编译器
- ▶ Qt SDK开发包
- ▶ Qvfb
  - ▶ 用X11的应用程序虚拟缓冲帧。和物理的显示设备在每个像素上保持一致

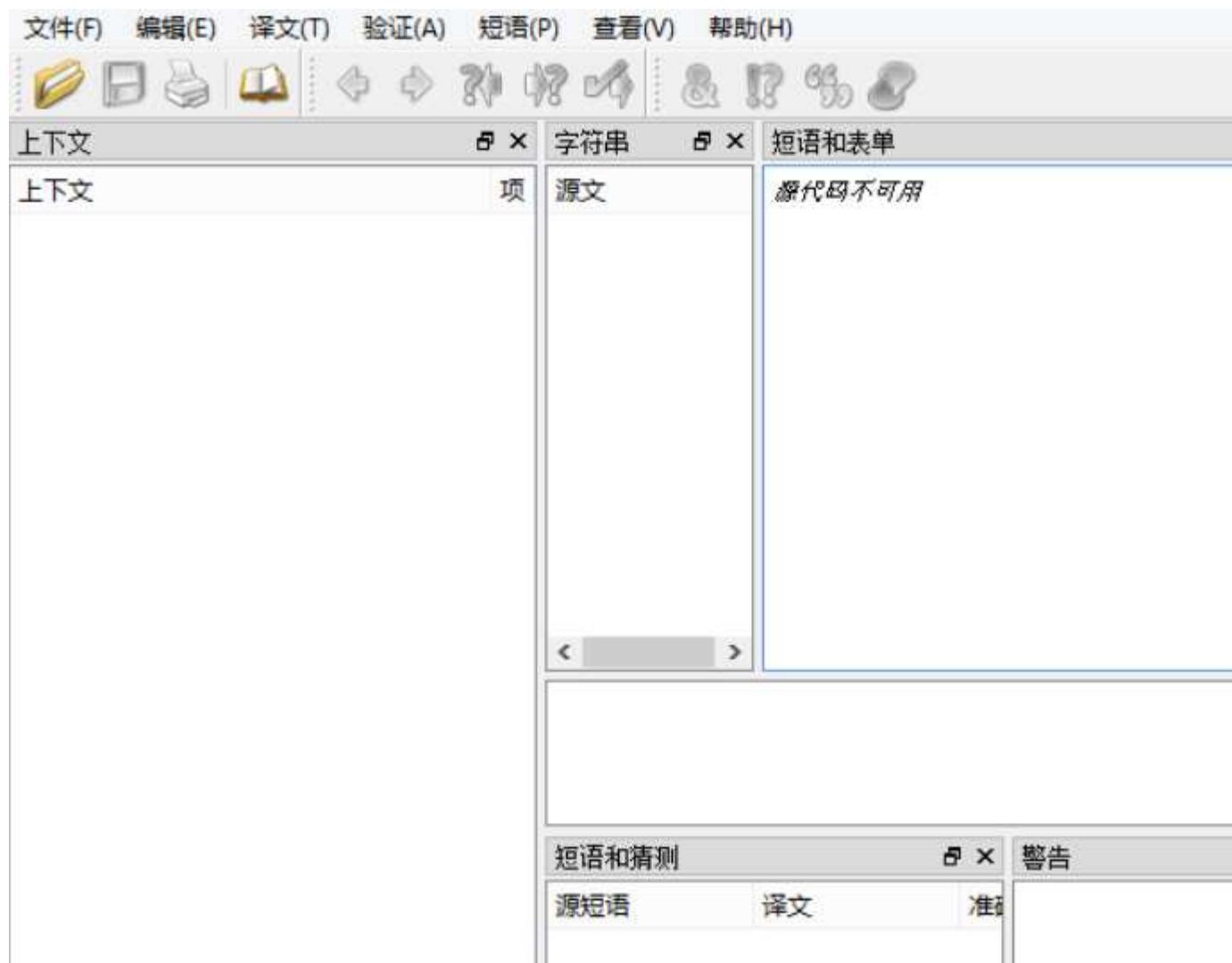
# 开发环境-Qt Designer



# 开发环境-Qt Assistant



# 开发环境-Qt Linguist

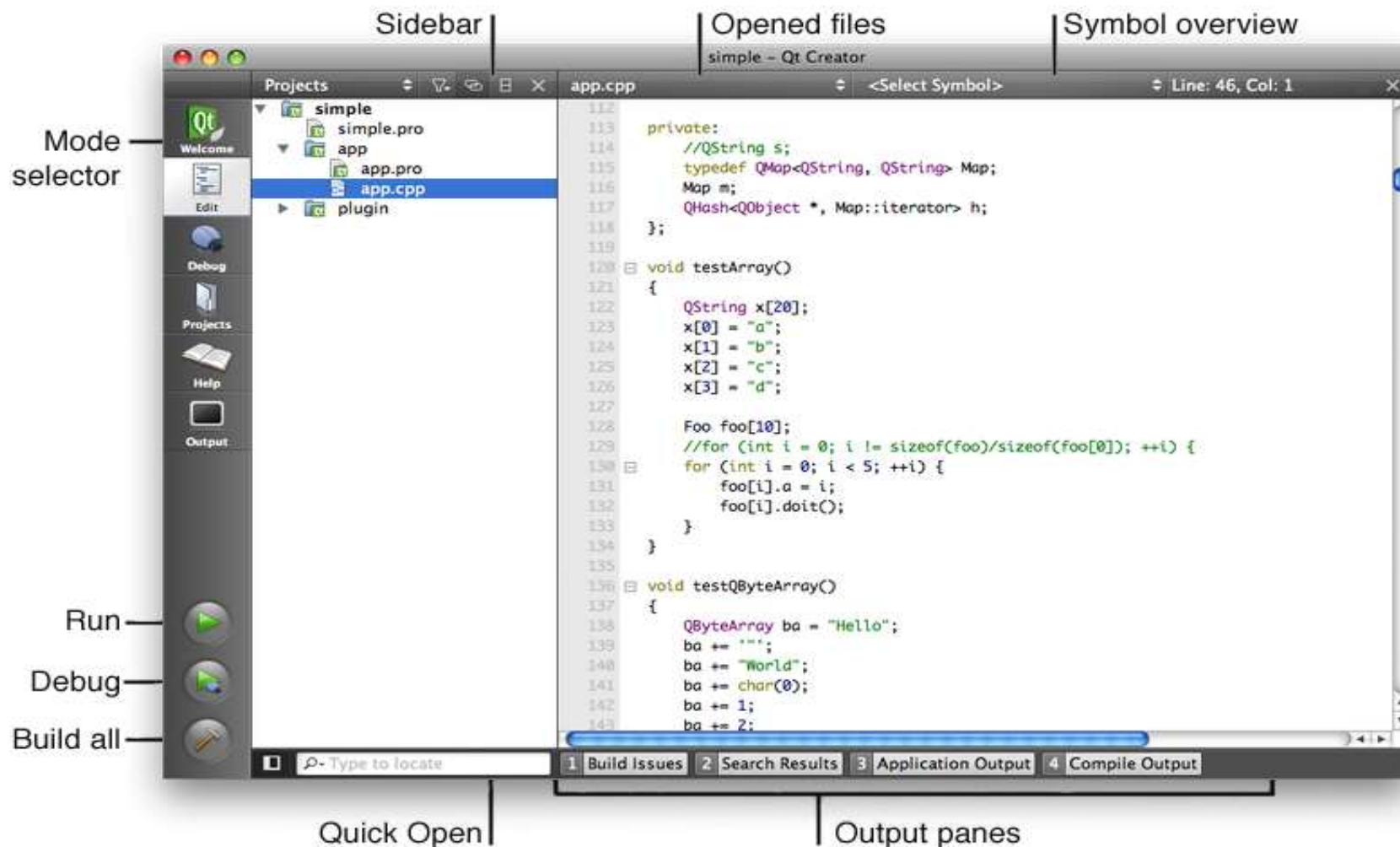


# 开发环境-Qt Creator

---

- ▶ Qt Creator是专为满足Qt开发人员需求而量身定制的跨平台集成开发环境(IDE)。
- ▶ Qt Creator可在Windows、Linux/X11和 Mac OS X 桌面操作系统上运行，供开发人员针对多个桌面和移动设备平台创建应用程序。
  - ▶ UI设计与代码开发的无缝协作
  - ▶ 支持Qt Quick 开发
  - ▶ 支持移动设备上的应用开发（Android，黑莓，iOS）
  - ▶ 支持嵌入式设备开发

# 开发环境-Qt Creator



# 开发环境-翻译发布管理器

---

- ▶ 翻译发布管理器提供了两个工具：lupdate、lrelease
- ▶ lupdate的用法
  - ▶ lupdate xxx.pro
  - ▶ 将生成或更新项目文件中的翻译源文件ts文件
- ▶ lrelease的用法
  - ▶ lrelease xxx.ts
  - ▶ 将把项目中的翻译源文件ts文件生成二进制qm文件

# 开发环境-qmake

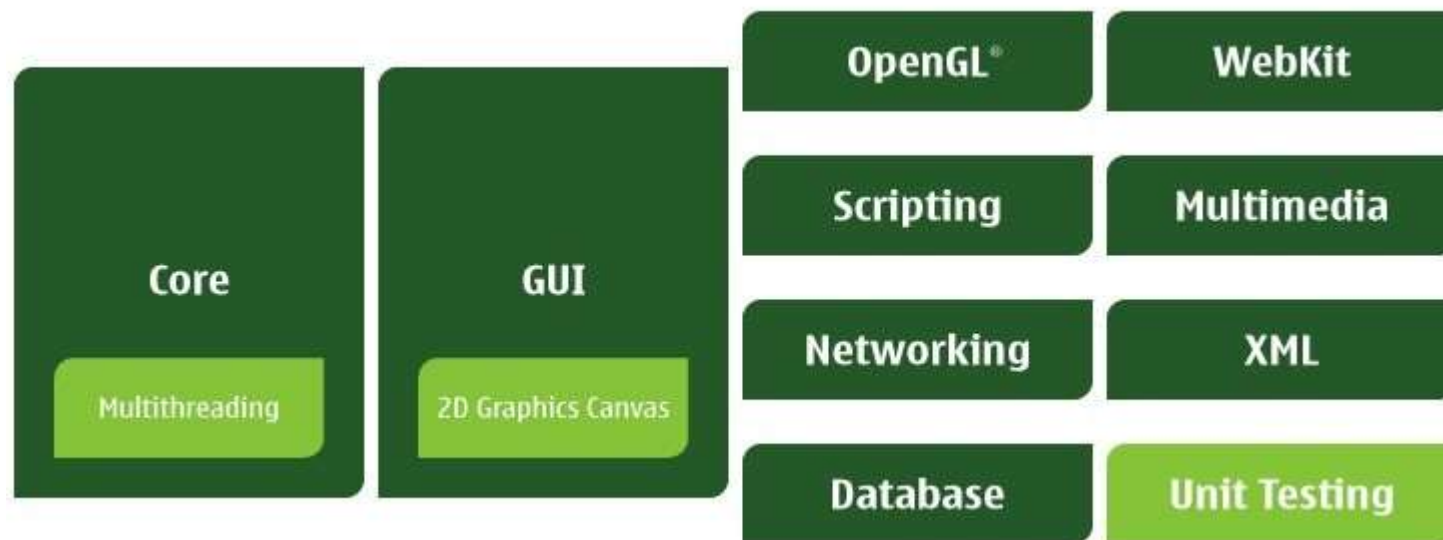
---

- ▶ 编译Qt程序
  - ▶ 生成工程文件: `qmake -project`
    - ▶ `#helloworld/hellowold.pro`
    - ▶ `TEMPLATE = app`
    - ▶ `CONFIG-= moc`
    - ▶ `DEPENDPATH+= .`
    - ▶ `INCLUDEPATH+= .`
    - ▶ `#Input`
    - ▶ `SOURCES+= main.cpp`
  - ▶ 生成Makefile: `qmake`
  - ▶ 编译工程: `make`



# 开发环境-Qt SDK 开发包

- ▶ Qt SDK 开发包：Qt核心库，包含直观的C++ API以及类CSS/JavaScript编程的Qt Quick



# Qt Creator 安装

---

## ► 下载

## ► <http://www.qt.io/download-open-source>

### Qt Online Installers

Qt online installer is a small executable which downloads content over internet based on your selections. It provides binary and source packages for different Qt library versions and latest Qt Creator.

- [Qt Online Installer for Linux 64-bit \(22.2 MB\)](#) [\(info\)](#)
- [Qt Online Installer for Linux 32-bit \(22.9 MB\)](#) [\(info\)](#)
- [Qt Online Installer for Mac \(9.5 MB\)](#) [\(info\)](#)
- [Qt Online Installer for Windows \(13.7 MB\)](#) [\(info\)](#)

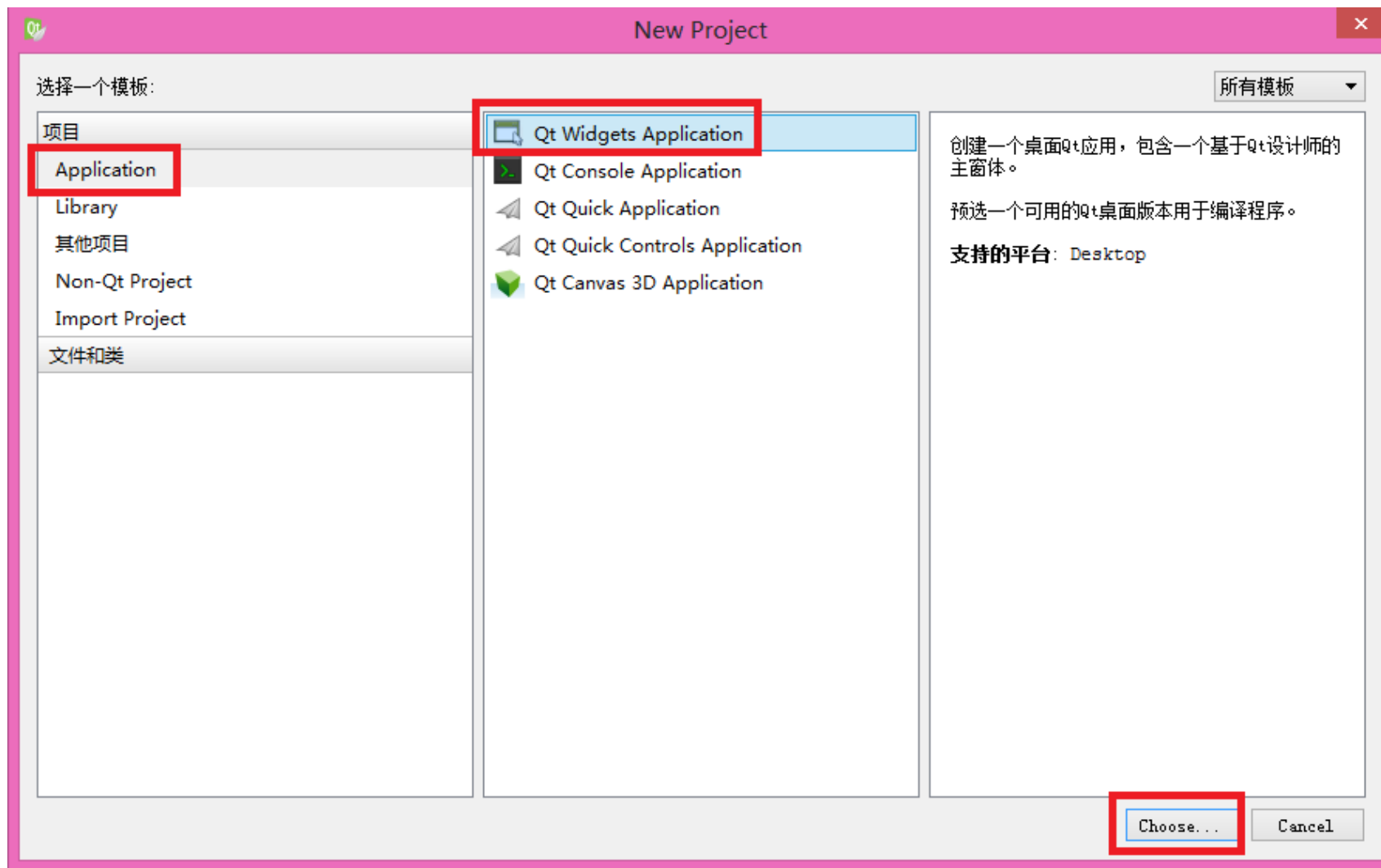
### Offline Installers

Qt offline installer is a stand-alone binary package including Qt libraries and Qt Creator.

### Linux Host

- [Qt 5.3.2 for Linux 32-bit \(449.3 MB\)](#) [\(info\)](#)
- [Qt 5.3.2 for Linux 64-bit \(446.8 MB\)](#) [\(info\)](#)

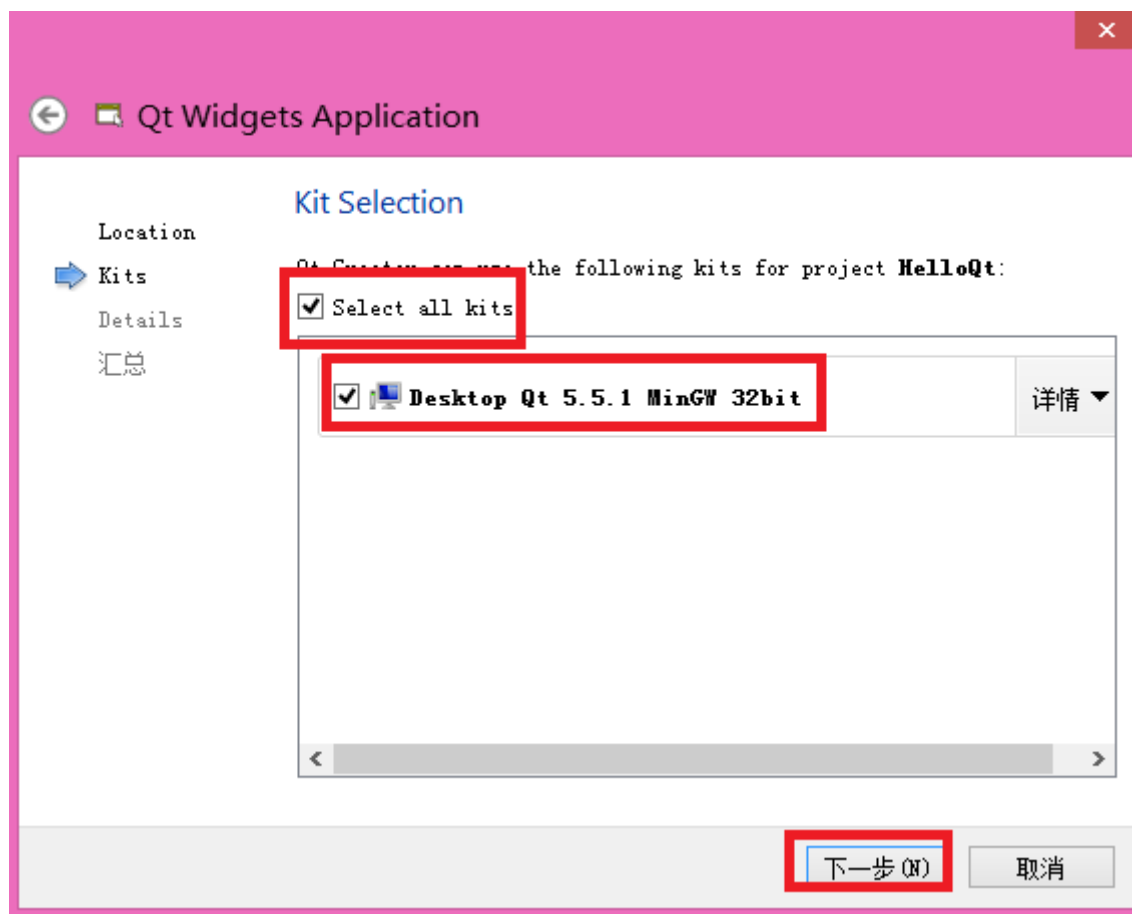
# 第一个Qt程序



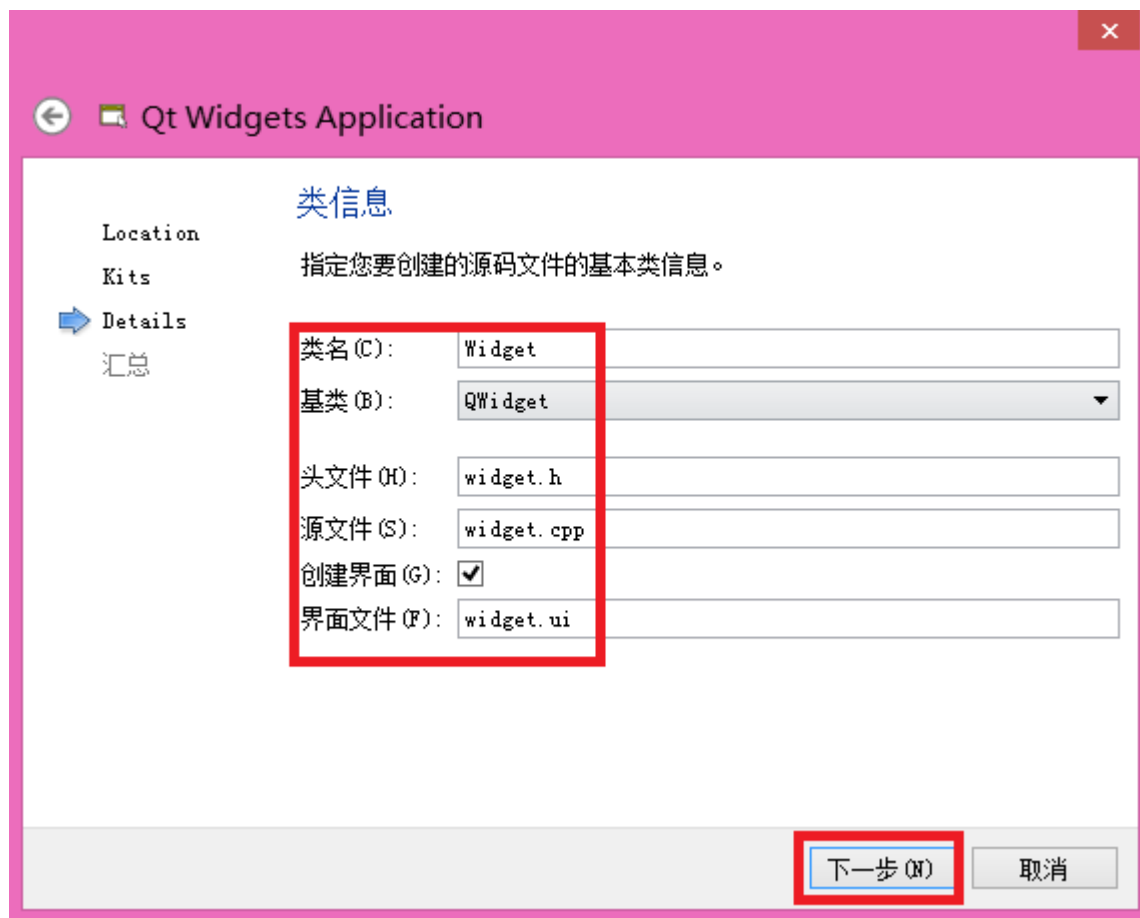
# 第一个Qt程序



# 第一个Qt程序



# 第一个Qt程序



The image shows the 'Qt Widgets Application' wizard dialog box. It has a pink title bar with a close button. On the left is a sidebar with 'Location', 'Kits', 'Details' (selected with a blue arrow), and 'Summary'. The main area is titled '类信息' (Class Information) and contains the instruction '指定您要创建的源码文件的基本类信息。' (Specify the basic class information for the source files you want to create). A red rectangle highlights the following fields: '类名 (C):' with text 'Widget', '基类 (B):' with a dropdown menu showing 'QWidget', '头文件 (H):' with text 'widget.h', '源文件 (S):' with text 'widget.cpp', '创建界面 (G):' with a checked checkbox, and '界面文件 (F):' with text 'widget.ui'. At the bottom right, another red rectangle highlights the '下一步 (N)' (Next) button, with a '取消' (Cancel) button next to it.

Qt Widgets Application

类信息

指定您要创建的源码文件的基本类信息。

Location  
Kits  
Details  
汇总

类名 (C): Widget

基类 (B): QWidget

头文件 (H): widget.h

源文件 (S): widget.cpp

创建界面 (G): ☒

界面文件 (F): widget.ui

下一步 (N) 取消

# 第一个Qt程序

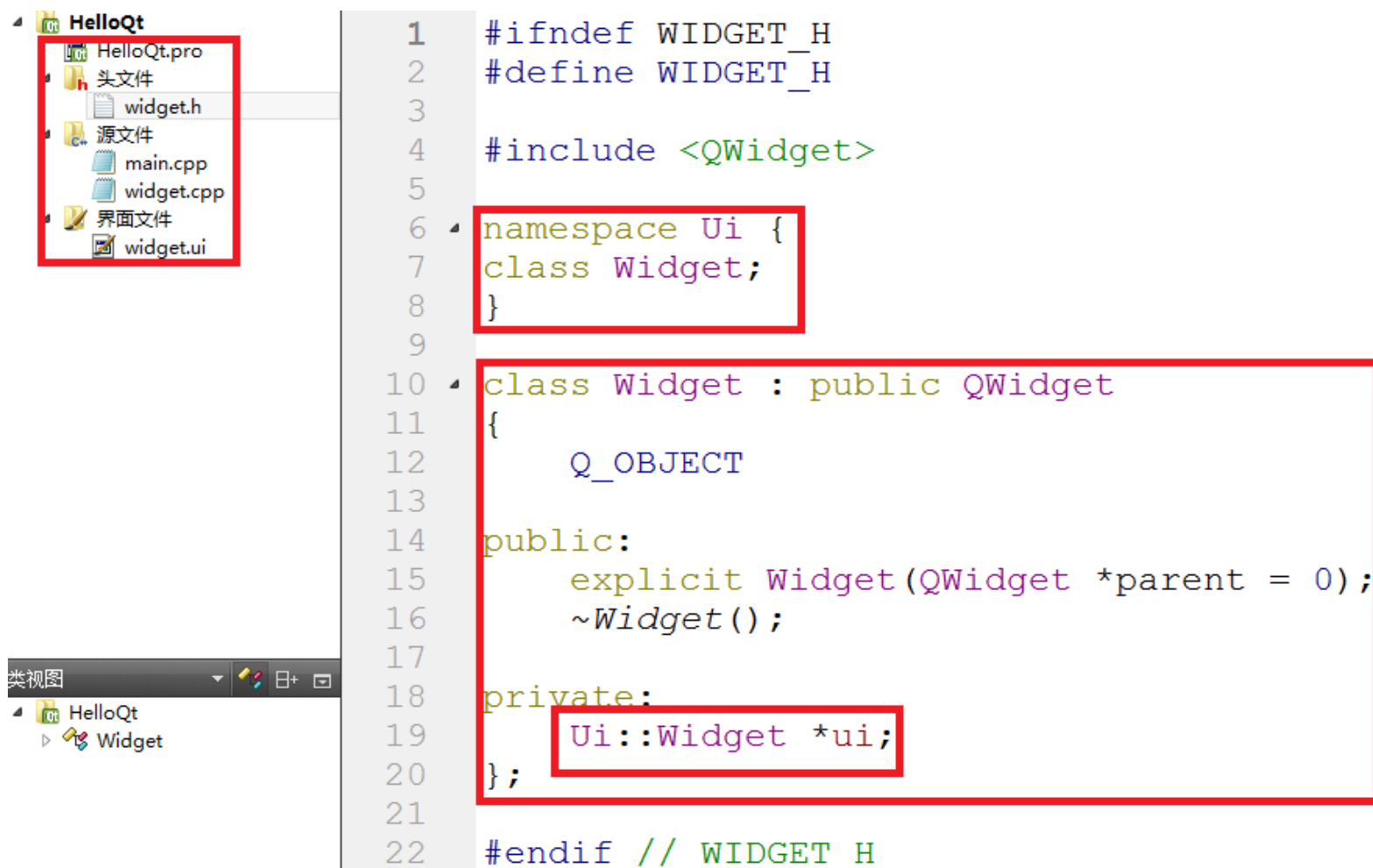


# 第一个Qt程序





# 第一个Qt程序



The screenshot displays the Qt Creator IDE interface. On the left, the 'HelloQt' project is expanded, showing the '头文件' (Header Files) folder containing 'widget.h'. Below it, the '源文件' (Source Files) folder contains 'main.cpp' and 'widget.cpp', and the '界面文件' (UI Files) folder contains 'widget.ui'. The '类视图' (Class View) at the bottom left shows the 'Widget' class under the 'HelloQt' project. The main editor area displays the content of 'widget.h', which is a header file for a Qt widget. The code is as follows:

```
1  #ifndef WIDGET_H
2  #define WIDGET_H
3
4  #include <QWidget>
5
6  namespace Ui {
7      class Widget;
8  }
9
10 class Widget : public QWidget
11 {
12     Q_OBJECT
13
14 public:
15     explicit Widget(QWidget *parent = 0);
16     ~Widget();
17
18 private:
19     Ui::Widget *ui;
20 };
21
22 #endif // WIDGET_H
```

# 第一个Qt程序

- ▶ pro文件说明:
  - ▶ QT += core gui
  - ▶ greaterThan(QT\_MAJOR\_VERSION, 4): QT += widgets
  - ▶ TARGET = HelloQt
  - ▶ TEMPLATE = app
  - ▶ SOURCES += main.cpp\  
    widget.cpp
  - ▶ HEADERS += widget.h
  - ▶ FORMS += widget.ui
- ▶ QT: 指定使用类的模块
- ▶ TARGET: 目标文件名
- ▶ TEMPLATE: 编译方法
- ▶ SOURCES: 源文件
- ▶ HEADERS: 头文件
- ▶ FORMS: 指定ui文件

# 第一个Qt程序

- ▶ `#include "widget.h"`//引用窗口类声明的头文件
- ▶ `#include <QApplication>`
- ▶ `int main(int argc, char *argv[])`//入口函数
- ▶ {
- ▶     `QApplication a(argc, argv);`//GUI事件处理对象
- ▶     `Widget w;`//定义窗口类对象
- ▶     `w.show();`//显示窗口
- ▶     `return a.exec();`//事件循环
- ▶ }

# 第一个Qt程序

## ▶ 事件循环的伪代码

```
▶ int QApplication::exec(){  
    ▶ while(1){  
        ▶ 等事件发生  
        ▶ 分发信号  
        ▶ 处理信号  
    }  
}
```

## ▶ 界面或操作分离

```
▶ class Widget : public QWidget{  
▶     Q_OBJECT//元对象编译器宏  
▶ public:  
▶     explicit Widget(QWidget *parent = 0);  
▶     ~Widget();  
▶     ...           //操作代码（后面讲的槽函数或普通成员函数）  
▶ private:  
▶     Ui::Widget *ui;//界面构造对象指针  
▶     ...           //其他成员属性  
▶ };
```

# 第一个Qt程序

## ▶ Ui文件说明

```
<ui version="4.0">
  <class>Widget</class>
  <widget class="QWidget" name="Widget">
    <property name="geometry">
      <rect>
        <x>0</x>
        <y>0</y>
        <width>400</width>
        <height>300</height>
      </rect>
    </property>
    <property name="windowTitle">
      <string>Widget</string>
    </property>
  </widget>
  <layoutdefault spacing="6" margin="11"/>
  <resources/>
  <connections/>
</ui>
```

窗口类

窗口继承基类 QWidget

窗口对象 Widget

窗口大小描述

窗口标题描述

窗口对象描述

# 第一个Qt程序

---

- ▶ Makefile说明:
- ▶ MAKEFILE = Makefile
- ▶ .....
- ▶ debug: FORCE
  - ▶ \$(MAKE) -f \$(MAKEFILE).Debug
- ▶ Makefile.Debug说明:
- ▶ ui\_widget.h: ../HelloQt/widget.ui
  - ▶ C:\Qt\Qt5.5.1\5.5\mingw492\_32\bin\uic.exe ../HelloQt/widget.ui -o ui\_widget.h

# 第一个Qt程序

## ► Ui文件自动生成代码说明:

```
class Ui_Widget
{
public:

    void setupUi(QWidget *Widget)
    {
        if (Widget->objectName().isEmpty())
            Widget->setObjectName(QStringLiteral("Widget"));
        Widget->resize(400, 300);

        retranslateUi(Widget);

        QObject::connectSlotsByName(Widget);
    } // setupUi

    void retranslateUi(QWidget *Widget)
    {
        Widget->setWindowTitle(QApplication::translate("Widget", "Widget", 0));
    } // retranslateUi

};
namespace Ui {
    class Widget: public Ui_Widget {};
} // namespace Ui
```

setupUi 函数，组装界面

retranslateUi 函数，窗口标题

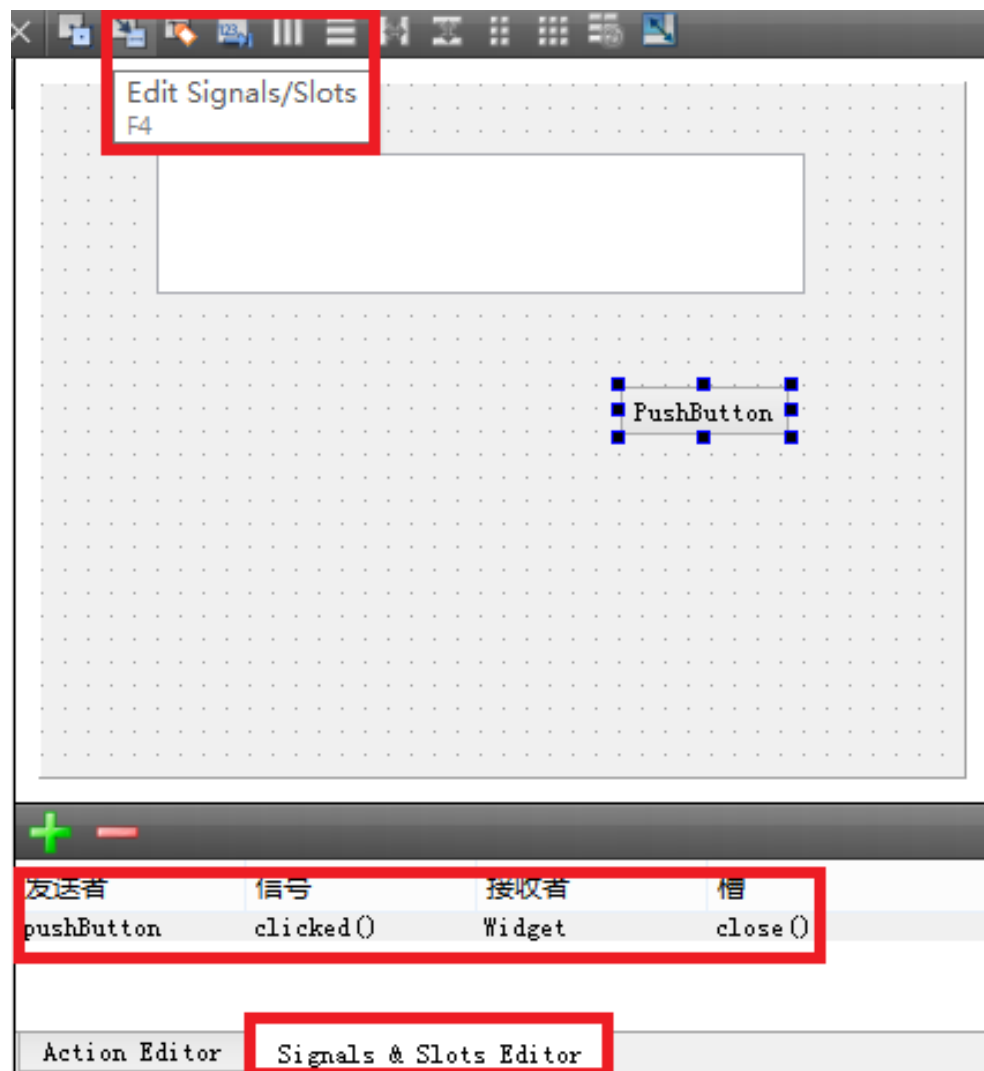
名字空间 Ui 声明定义类 Widget

# 信号与槽

- ▶ 信号和槽机制是Qt的一个主要特征，是Qt与其它工具包最不相同的部分。通过反馈的方式动态地或松散地将事件和状态变化联系起来。
- ▶ Qt工作的原理：事件驱动，信号槽机制。
- ▶ 回调（callback）是一个函数指针，当一个事件发生时被调用，任何函数都可以被安排作为回调。
  - ▶ 没有类型安全
  - ▶ 总是以直接调用方式工作
- ▶ 信号和槽的方式更加动态
  - ▶ 一个更通用的机制
  - ▶ 更容易互连两个已存在的类
  - ▶ 相关类之间涉及更少的知识共享
- ▶ Qt采用信号和槽实现对象部件之间的通信。



# 信号与槽



- ▶ 能携带任意数量和任意类型的参数,取代原始的回调和消息映射机制
- ▶ 面向对象, 独立于标准C/C++, 必须借助QT工具moc (Meta Object Compiler), C++预处理程序, 为高层次事件处理自动生成所需要附加代码
- ▶ 必须把事件和相关代码联系起来, 才能对事件做出响应。才能使不同类型的对象之间能够进行通信



- ▶ 当信号被发射时，QT代码将回调与其相连接的槽函数
- ▶ 信号将由元对象处理moc自动翻译成C++代码
- ▶ 信号的声明不在.cpp文件中，而在头文件中
  - ▶ Q\_OBJECT
  - ▶ .....
  - ▶ signals:
    - ▶ void mySignal();
    - ▶ void mySignal(int x);
    - ▶ void mySignalParam(int x,int y);

- ▶ 槽函数是普通的C++成员函数，可以被正常调用
- ▶ 槽函数可以有返回值，也可以没有。
- ▶ 槽函数的访问权限三种：public slots、private slots和protected slots。槽函数的存取权限决定了谁能够与其相关联。
- ▶ 头文件中中声明
  - ▶ Q\_OBJECT
  - ▶ .....
  - ▶ public slots:
    - ▶ void mySlot();
    - ▶ void mySlot(int x);
    - ▶ void mySignalParam(int x,int y);

- ▶ 原型：
  - ▶ `QMetaObject::Connection QObject::connect(const QObject * sender, const char * signal, const QObject * receiver, const char * method, Qt::ConnectionType type = Qt::AutoConnection);`
- ▶ 槽函数执行方式分为：自动、直接、队列、阻塞队列等等。
- ▶ 信号与槽关联
  - ▶ `QObject::connect( sender, SIGNAL(signal),receiver, SLOT(method) );`
- ▶ 信号与信号相连
  - ▶ `QObject::connect( sender, SIGNAL(signal), receiver, SIGNAL(signal) );`
- ▶ 同一个信号连接到多个槽
  - ▶ `QObject::connect( sender, SIGNAL(signal),receiver, SLOT(method1) );`
  - ▶ `QObject::connect( sender, SIGNAL(signal),receiver, SLOT(method2) );`
  - ▶ .....
- ▶ 多个信号连接到同一个槽
  - ▶ `QObject::connect( sender, SIGNAL(signal1),receiver, SLOT(method) );`
  - ▶ `QObject::connect( sender, SIGNAL(signal2),receiver, SLOT(method) );`
  - ▶ .....

# 信号与槽：连接

## ▶ 信号槽的参数对应关系：

Signals		Slots
<code>rangeChanged(int,int)</code>	—————	<code>setRange(int,int)</code>
<code>rangeChanged(int,int)</code>	—————	<code>setValue(int)</code>
<code>rangeChanged(int,int)</code>	—————	<code>updateDialog()</code>
<code>valueChanged(int)</code>	<del>—————</del>	<code>setRange(int,int)</code>
<code>valueChanged(int)</code>	—————	<code>setValue(int)</code>
<code>valueChanged(int)</code>	—————	<code>updateDialog()</code>
<code>textChanged(QString)</code>	<del>—————</del>	<code>setValue(int)</code>
<code>clicked()</code>	<del>—————</del>	<code>setValue(int)</code>
<code>clicked()</code>	—————	<code>updateDialog()</code>

# 信号与槽：发送信号

---

- ▶ **signal**一般是在事件处理时候Qt发出，如果需要程序自己触发信号，则使用emit。
- ▶ 使用语法如下：
  - ▶ emit signal

## 信号与槽：取消连接

- ▶ 如果不需要连接信号槽的时候，可以取消连接。
- ▶ 函数原型（有多个版本，函数重载）：
  - ▶ `bool QObject::disconnect(const QObject * sender, const char * signal, const QObject * receiver, const char * method);`
- ▶ 取消一个连接不是很常用，因为Qt会在一个对象被删除后自动取消这个对象所包含的所有连接





# 窗口部件

华清远见

- ▶ 常用类介绍
- ▶ 内置部件介绍
- ▶ 基础窗口部件（QWidget）
- ▶ 布局管理器

# Qt 常用类介绍

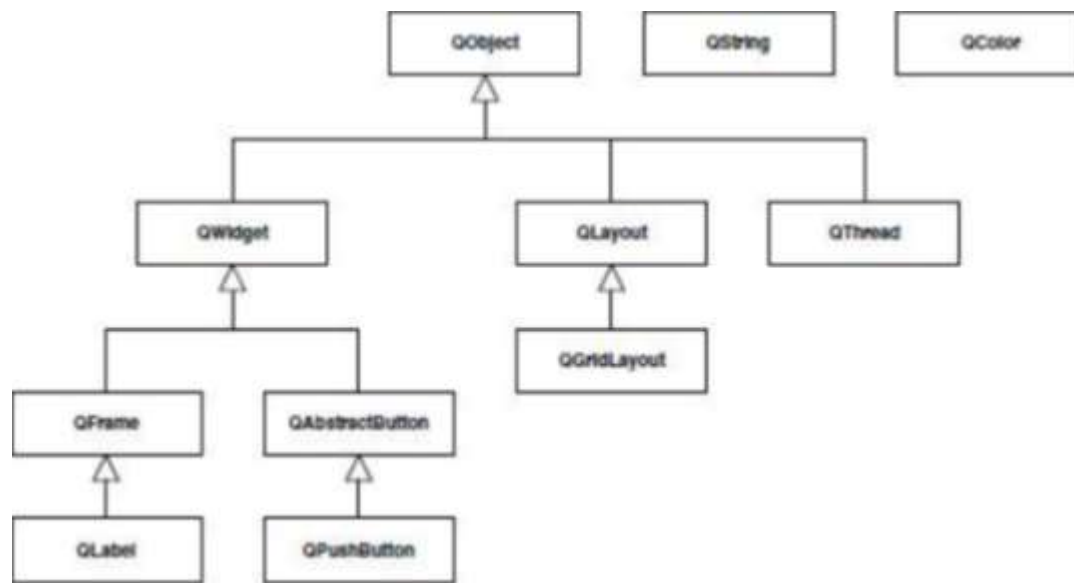
- ▶ 类、部件、组件
- ▶ Qt中最基本的类

- ▶ QObject派生类

- ▶ QWidget
- ▶ QLayout
- ▶ QThread
- ▶ QTcpSocket

- ▶ 非QObject派生类

- ▶ QString: 处理字符串
- ▶ QImage: 加载与保存图像
- ▶ QColor: 处理色彩
- ▶ .....



- ▶ QObject类是所有能够处理signal， slot和事件的Qt对象的基类

## QObject作用

- Qt对象模型的核心
  - 是绝大多数类的基类
  - 所有的QWidget都是QObject
  - 提供对象树和对象的关系
  - QObject在整个Qt的概念体系中处在一个非常重要的位置
  - 提供了信号-槽的通信机制
- 具有三个作用
  - 内存管理
  - Introspection（自省）
  - 事件处理



# Qt 常用类介绍

---

- ▶ QApplication类负责GUI应用程序的控制流和主要的设置，它包括主事件循环体，负责处理和调度所有来自窗口系统和其他资源的事件
- ▶ 处理应用程序的开始，结束以及会话管理
- ▶ QApplication是QObject类的子类

# Qt 常用类介绍

---

- ▶ 在非图形程序中，QCoreApplication类接管了QApplication类在GUI应用程序中的角色：它使得事件循环机制能够使用。如果你需要异步通讯的话，这将是非常有用的，或者不同的线程之间，或者通过网络套接字。
- ▶ QCoreApplication是QObject类的子类

# Qt 常用类介绍

---

- ▶ QWidget类继承了QObject类的属性
- ▶ QWidget类是所有用户接口对象的基类
- ▶ 组件是用户界面的单元组成部分，接收鼠标，键盘和从其它窗口系统来的事件
- ▶ QWidget类有很多成员函数，但一般不直接使用，而通过子类继承来使用其函数功能

# Qt 常用类介绍

---

## ▶ 基本库QtCore

- ▶ 基本数据类型，例如：QString、QByteArray
- ▶ 基本数据结构，例如：QList、QVector、QHash
- ▶ 输入输出类，例如：QIODevice、QTextStream、QFile
- ▶ 多线程编程用到的类，例如：QThread、QWaitCondition
- ▶ 提供QObject和QCoreApplication



# Qt 常用类介绍

---

## ▶ GUI库QtGui

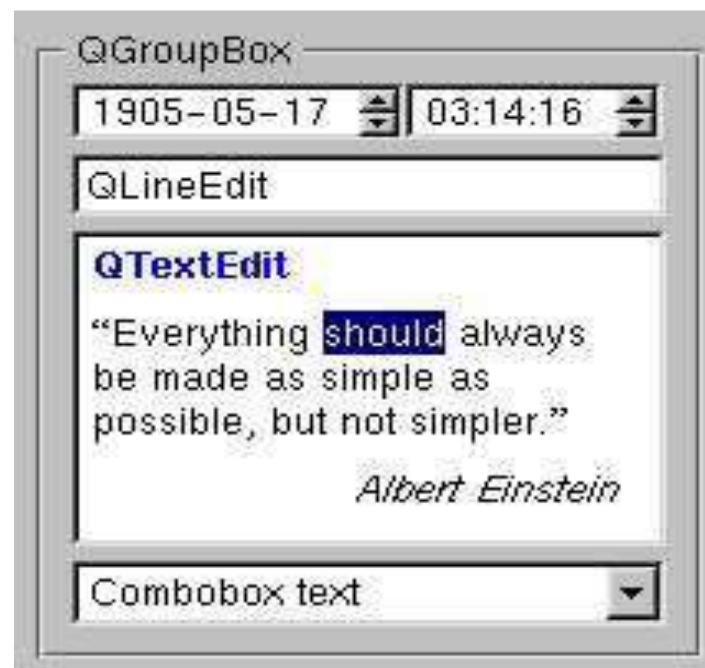
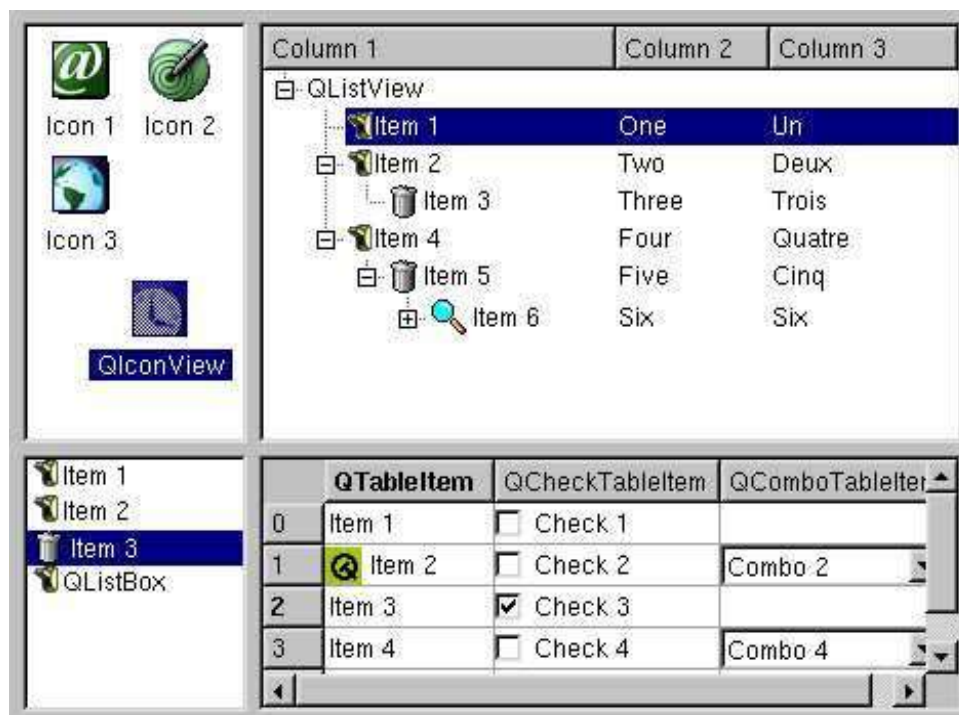
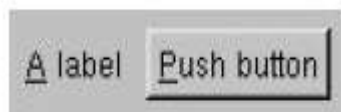
- ▶ QWidget类，以及由它派生出来的类，例如：QLabel、QPushButton
- ▶ 布局类，例如：QVBoxLayout、QHBoxLayout、QGridLayout
- ▶ 主窗口类，QMainWindow、QMenu
- ▶ 绘图类，例如：QPainter、QPen、QBrush
- ▶ 提供准备好使用的(ready-to-use)对话框类，例如：QFileDialog、QPrintDialog
- ▶ 应用程序类QApplication

# Qt 常用类介绍

---

- ▶ 网络库QtNetwork
  - ▶ QTcpSocket
  - ▶ QUdpSocket
  - ▶ QHttp
  - ▶ QFtp
- ▶ OpenGL库QtOpenGL
  - ▶ QGLWidget
- ▶ Database库QtSql
- ▶ XML库QtXml
- ▶ 兼容库Qt3Support

# 内置部件介绍



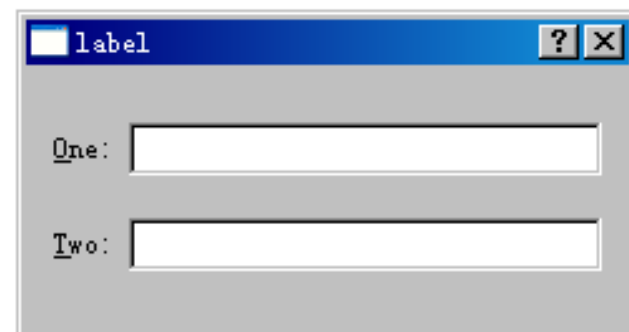
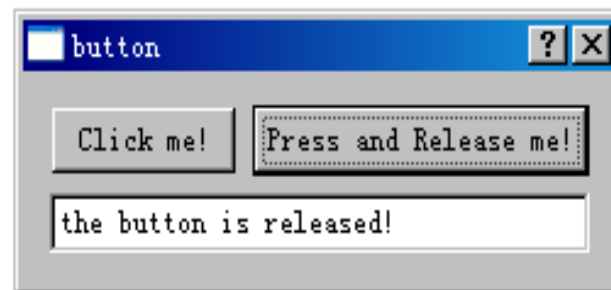
# 内置部件

---

- ▶ 按钮（QPushButton）
- ▶ 标签（QLabel）
- ▶ 复选框（QCheckBox）
- ▶ 单选按钮（QRadioButton）
- ▶ 分组框（QGroupBox）
- ▶ 列表部件框（QListWidget）
- ▶ 组合框（QComboBox）
- ▶ 自旋框（QSpinBox）
- ▶ 滑动条（QSlider）
- ▶ 进度条（QProgressBar）

# 内置部件

- ▶ 按钮 (QPushButton)
  - ▶ clicked() (signal)
  - ▶ pressed() (signal)
  - ▶ release() (signal)
- ▶ 标签 (QLabel)
  - ▶ setBuddy(QWidget\*)



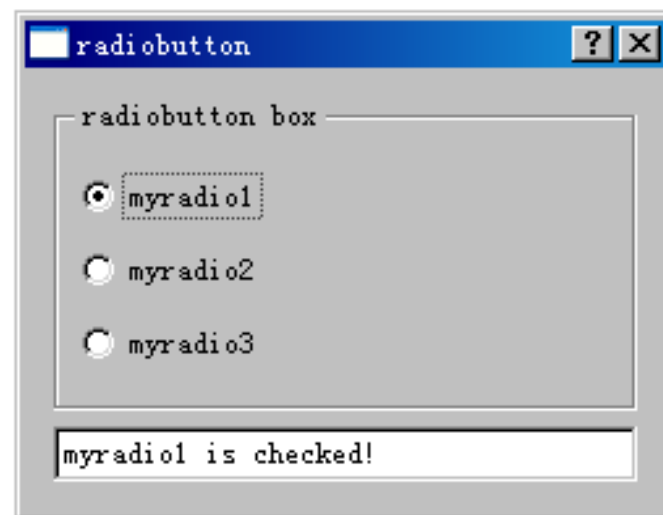
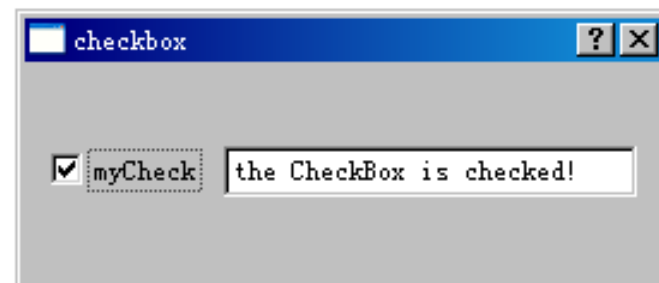
# 内置部件

- ▶ 行编辑框（QLineEdit）与文本编辑框（QTextEdit）
  - ▶ text()
  - ▶ setText(const QString &) (slot)
  - ▶ textChanged(const QString&) (signal)
  - ▶ textEdited(const QString&) (signal)
  - ▶ setMaxLength(int)
  - ▶ setEchoMode(EchoMode)



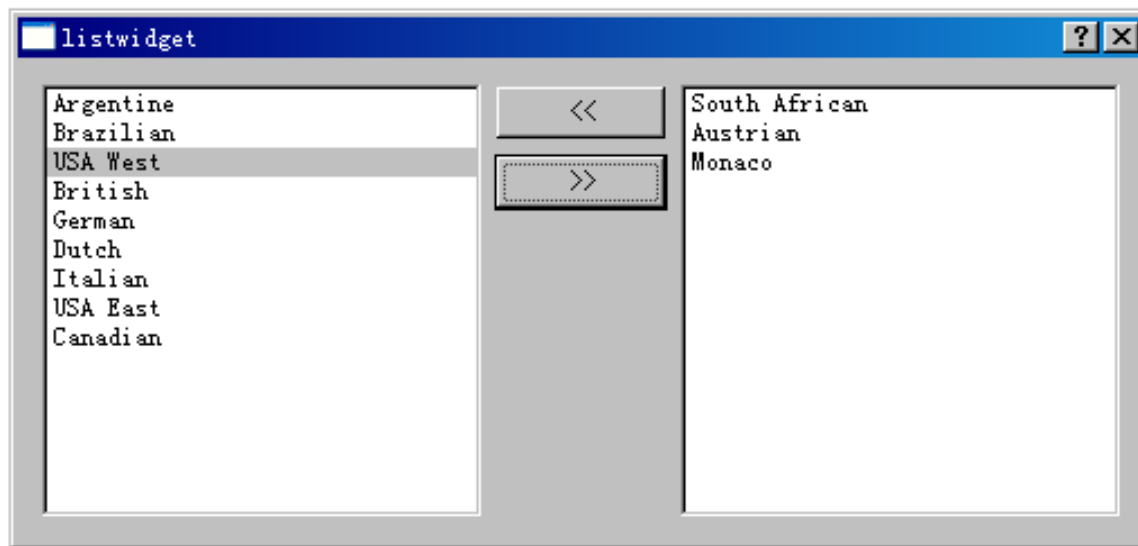
# 内置部件

- ▶ 复选框（QCheckBox）
  - ▶ isChecked()
  - ▶ stateChanged(int) (signal)
- ▶ 单选按钮（QRadioButton）
  - ▶ clicked() (signal)
  - ▶ isChecked()
- ▶ 分组框（QGroupBox）



# 内置部件

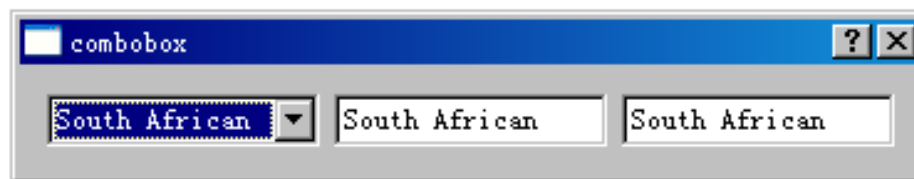
- ▶ 列表部件框（QListWidget）
  - ▶ addItem(const QString&)
  - ▶ addItems(const QStringList&)
  - ▶ selectedItems()
  - ▶ item(int row)
  - ▶ takeItem(int row)
  - ▶ setSelectionMode(QAbstractItemView::SelectionMode mode)





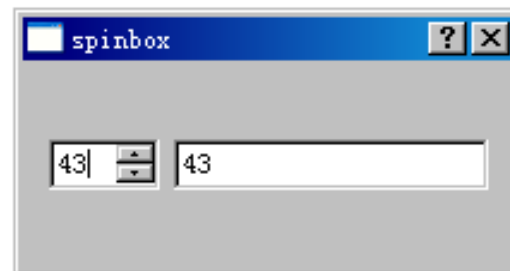
# 内置部件

- ▶ 组合框 (QComboBox)
  - ▶ setEditable()
  - ▶ activated() (signal)
  - ▶ currentIndexChanged() (signal)
  - ▶ editTextChanged() (signal)
  - ▶ currentIndex() / currentText()



# 内置部件

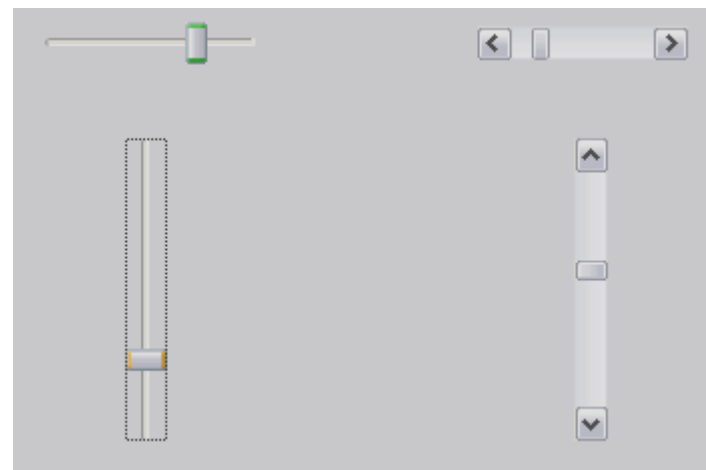
- ▶ 自旋框（QSpinBox）
  - ▶ `setMinimum(int)`
  - ▶ `setMaximum(int)`
  - ▶ `setRange(int,int)`
  - ▶ `setSingleStep(int)`
  - ▶ `valueChanged(int)(signal)`
  - ▶ `setValue(int)(slot)`
- ▶ 双精度自旋框（QDoubleSpinBox）
  - ▶ `setDecimals()`



# 内置部件

## ▶ 滑动条 (QSlider)

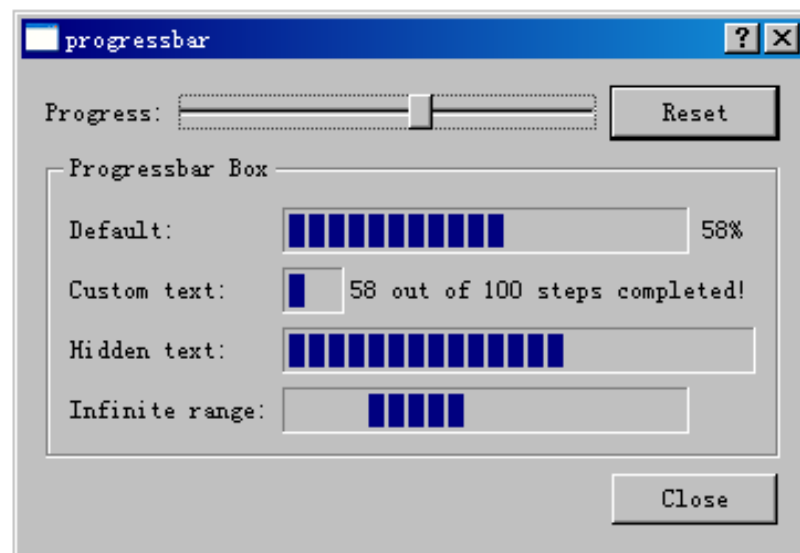
- ▶ `setMinimum(int)`
- ▶ `setMaximum(int)`
- ▶ `setRange(int,int)`
- ▶ `setSingleStep(int)`
- ▶ `setPageStep(int)`
- ▶ `setOrientation(Qt::Orientation)`
- ▶ `valueChanged(int)(signal)`



## ▶ 滚动条 (QScrollBar)

# 内置部件

- ▶ 进度条 (QProgressBar)
  - ▶ valueChanged(int)(signal)
  - ▶ setMaximum(int)(slot)
  - ▶ setMinimum(int)(slot)
  - ▶ setRange(int,int)(slot)
  - ▶ setValue(int)(slot)
  - ▶ reset()(slot)
  - ▶ setFormat(const QString&)
  - ▶ setTextVisible(bool)



# 基础窗口部件（QWidget）

## QWidget Class

Qt 5.5 ▶ Qt Widgets ▶ C++ Classes ▶ QWidget

The [QWidget](#) class is the base class of all user interface objects. [More...](#)

Header: `#include <QWidget>`

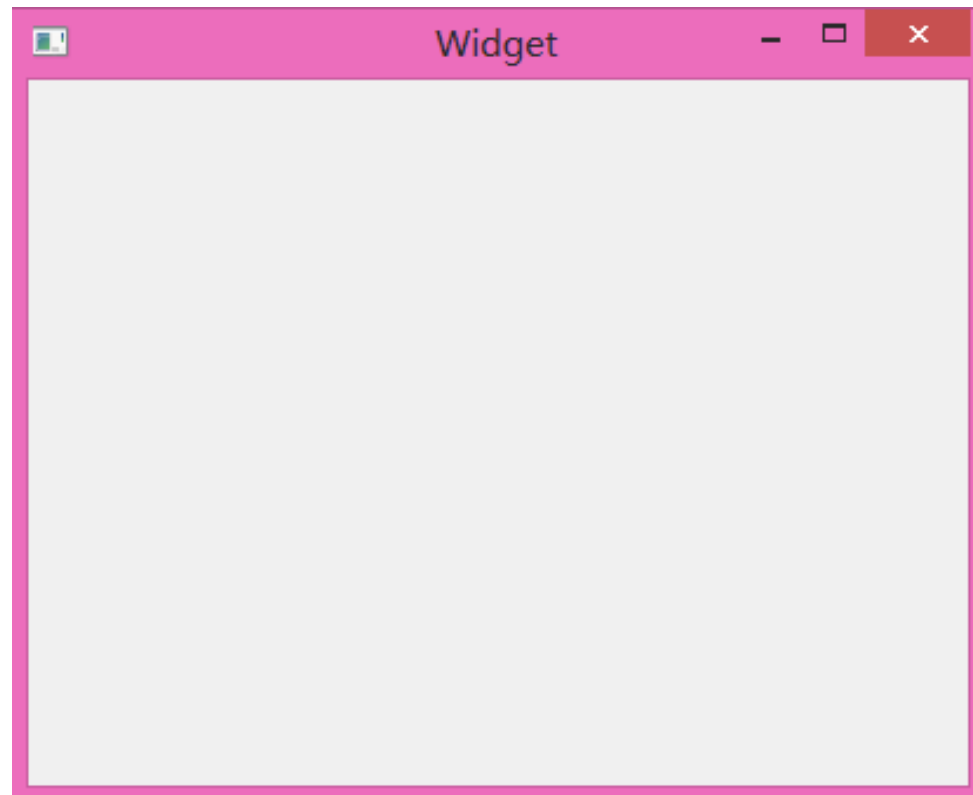
qmake: `QT += widgets`

Inherits: [QObject](#) and [QPaintDevice](#)

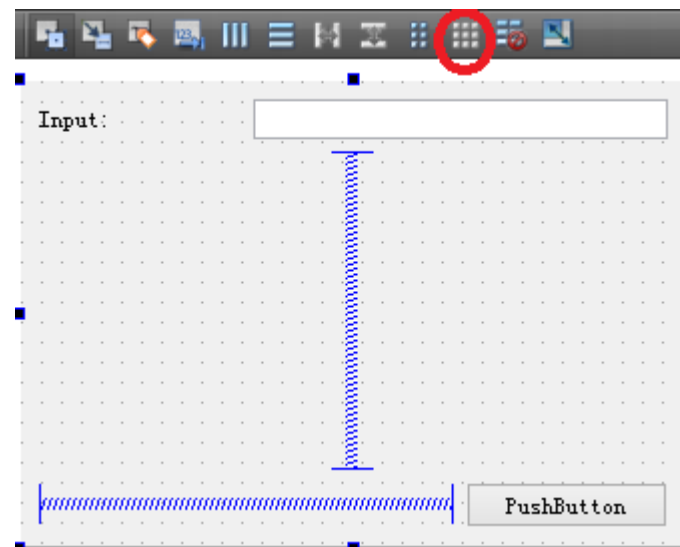
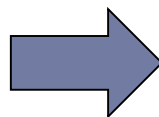
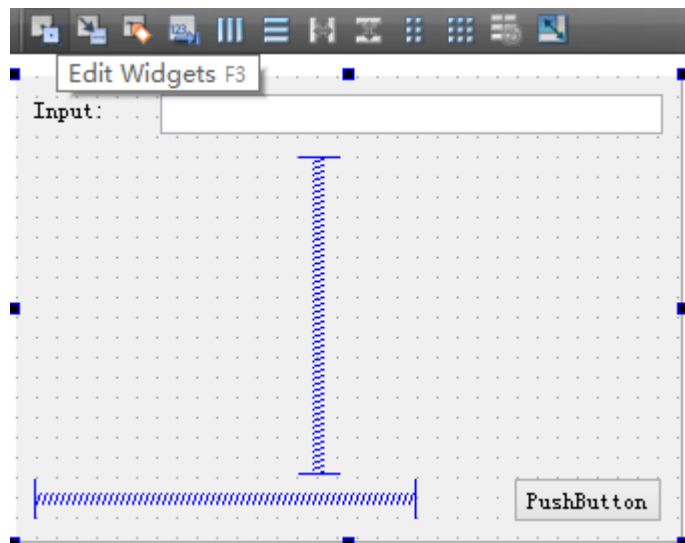
Inherited By: [QAbstractButton](#), [QAbstractSlider](#), [QAbstractSpinBox](#), [QCalendarWidget](#), [QComboBox](#), [QDesktopWidget](#), [QDialog](#), [QDialogButtonBox](#), [QDockWidget](#), [QFocusFrame](#), [QFrame](#), [QGroupBox](#), [QKeySequenceEdit](#), [QLineEdit](#), [QMacCocoaViewContainer](#), [QMacNativeWidget](#), [QMainWindow](#), [QMdiSubWindow](#), [QMenu](#), [QMenuBar](#), [QOpenGLWidget](#), [QProgressBar](#), [QRubberBand](#), [QSizeGrip](#), [QSplashScreen](#), [QSplitterHandle](#), [QStatusBar](#), [QTabBar](#), [QTabWidget](#), [QToolBar](#), and [QWizardPage](#)

# 基础窗口部件（QWidget）

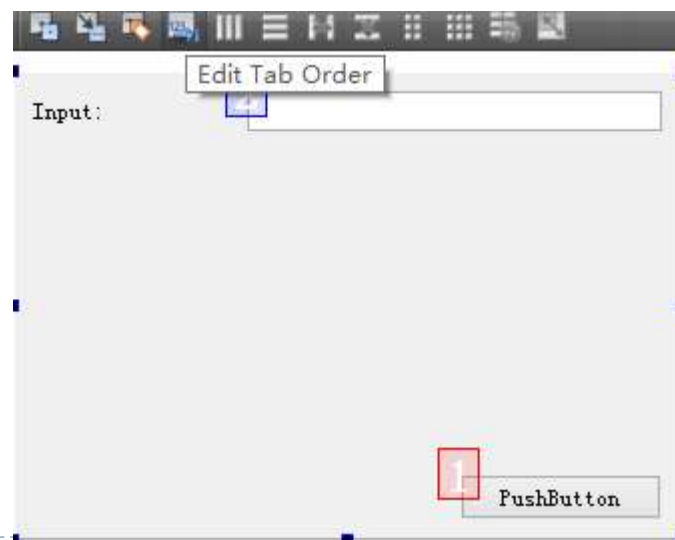
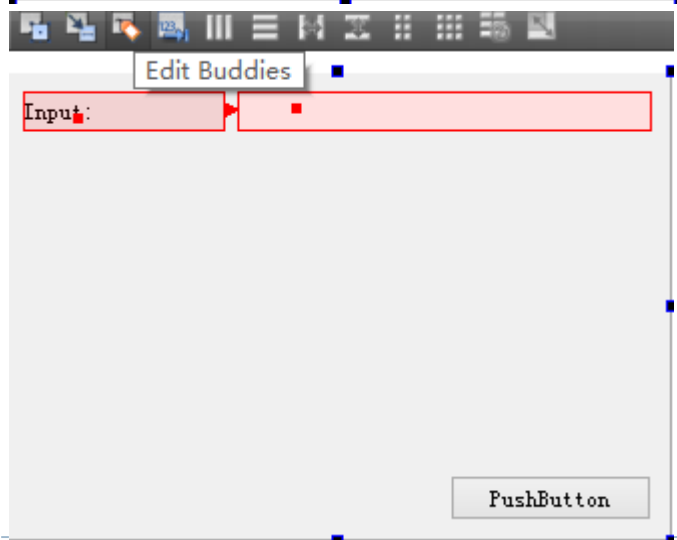
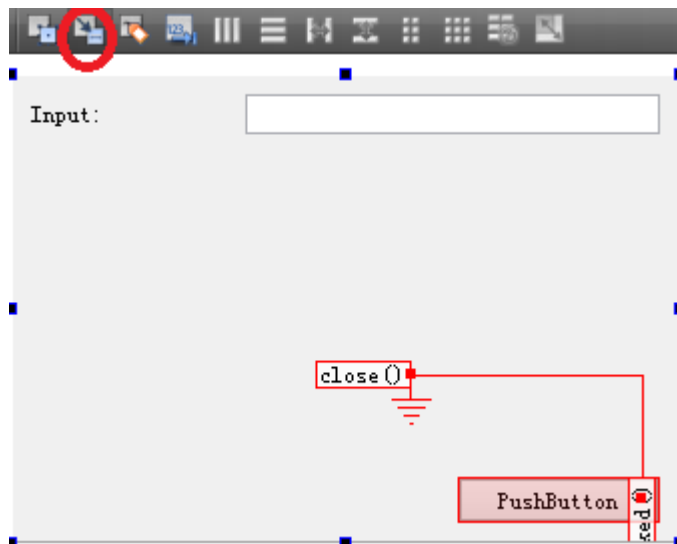
- ▶ 基础窗口部件主要用于自定义窗口。
- ▶ **QWidget**提供一个基础窗口，窗口并没有任何图形部件。通过指定图形部件的父对象来把图形部件放上基础窗口，或是通过自动布局工具把图形部件放上基础窗口。



# 基础窗口部件（QWidget）



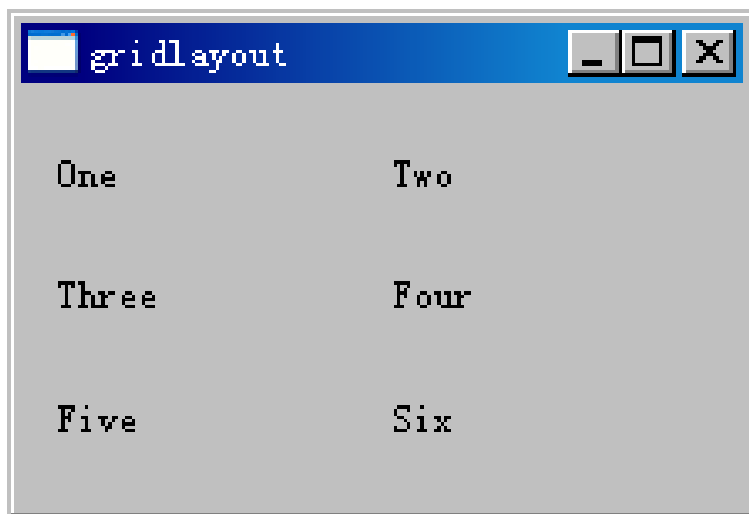
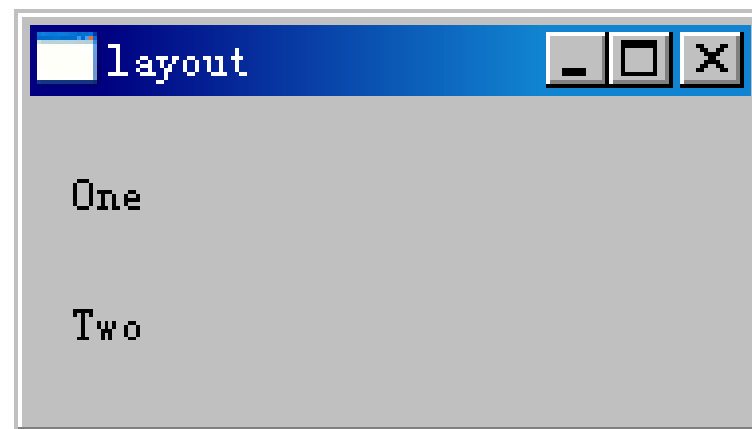
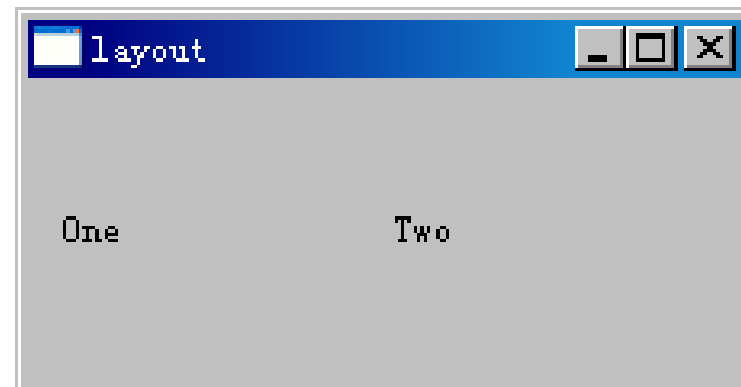
# 基础窗口部件（QWidget）





# 布局管理器

- ▶ 布局管理器主要常用的三个类：QHBoxLayout、QVBoxLayout、QGridLayout
- ▶ QHBoxLayout：水平布局
- ▶ QVBoxLayout：垂直布局
- ▶ QGridLayout：网格布局



# 布局管理器

- ▶ 布局管理器，可以管理任意从QWidget继承出来的窗体，也布局管理器管理布局管理。
- ▶ 布局管理器添加窗体：
  - ▶ `void QLayout::addWidget(QWidget *w);`
  - ▶ `void QVBoxLayout::addWidget(QWidget *widget, int stretch = 0, Qt::Alignment alignment = Qt::Alignment());`
  - ▶ `void QGridLayout::addWidget(QWidget *widget, int fromRow, int fromColumn, int rowSpan, int columnSpan, Qt::Alignment alignment = Qt::Alignment());`
- ▶ 布局管理器添加布局管理器：
  - ▶ `void QVBoxLayout::addLayout(QLayout *layout, int stretch = 0);`
  - ▶ `void QGridLayout::addLayout(QLayout *layout, int row, int column, int rowSpan, int columnSpan, Qt::Alignment alignment = Qt::Alignment());`
- ▶ 一个窗体有且仅有一个布局管理器（其他的布局管理器可以被窗体的布局管理器管理而已），成为窗体的布局管理器的方法：
  - ▶ 1、构造布局管理器对象时设置窗体为父对象
  - ▶ 2、`void QWidget::setLayout(QLayout *layout);`



# 事件及图形系统

华清远见

- ▶ 事件处理
- ▶ 2D绘图基础

- ▶ Qt的基本工作原理是：事件驱动，信号槽机制
- ▶ Qt的事件由窗口系统或是Qt自身产生，用于响应各种需要处理的事务。
- ▶ QEvent类定义了Qt中的事件，通过enum QEvent::Type可以查询。
- ▶ 常见需要处理的有窗口系统的QKeyEvent、QMouseEvent、QPaintEvent、QResizeEvent、QMoveEvent事件，来自系统的QTimerEvent，等等。
- ▶ 处理事件的方式：
  - ▶ 重写特定事件处理器，如：
    - ▶ `QMoveEvent::QMoveEvent(const QPoint & pos, const QPoint & oldPos)`
    - ▶ `QKeyEvent::QKeyEvent(Type type, int key, Qt::KeyboardModifiers modifiers, const QString & text = QString(), bool autorep = false, ushort count = 1)`
    - ▶ .....
  - ▶ 重写QObject事件处理虚函数
    - ▶ `bool QObject::event(QEvent * event)`
  - ▶ 注册事件过滤器，并重写过滤器
    - ▶ `void QObject::installEventFilter(QObject * filterObj)`
    - ▶ `bool QObject::eventFilter(QObject * watched, QEvent * event)`
  - ▶ 发送事件
    - ▶ `bool QApplication::notify(QObject * receiver, QEvent * event)`

## ▶ 1、特定事件处理器

- ▶ 从QObject继承出来的对象，都会重写事件处理器，形成新的特定事件处理器，如：
- ▶ `void keyPressEvent(QKeyEvent * event);`
- ▶ `void keyReleaseEvent(QKeyEvent * event);`
- ▶ `void mouseDoubleClickEvent(QMouseEvent * event);`
- ▶ .....

## ▶ 2、QObject对象事件处理器

- ▶ 从QObject继承出来的对象，有一个统一的事件处理器，用于集中处理事件
- ▶ `bool event(QEvent * ev);`

## ▶ 3、事件过滤器

- ▶ 从QObject继承出来的对象，可以注册一个事件过滤器，所有的事件先交给事件过滤器处理。
- ▶ `bool eventFilter(QObject *obj, QEvent *event);`

## ▶ 发送事件

- ▶ `bool QCoreApplication::notify(QObject * receiver, QEvent * event)`
- ▶ `void QCoreApplication::postEvent(QObject * receiver, QEvent * event, int priority = Qt::NormalEventPriority)`
- ▶ `bool QCoreApplication::sendEvent(QObject * receiver, QEvent * event)`
- ▶ 可以构造一个特定事件，向指定对象（从QObject继承）发送。一般情况下不需要使用到。

## ▶ 处理事件

- ▶ `void QCoreApplication::processEvents(QEventLoop::ProcessEventsFlags flags = QEventLoop::AllEvents)`
- ▶ Qt程序是所谓的GUI程序，有前台界面和后台代码。如果存在后台代码耗时较长的情况，如果解决界面的及时响应就是一个比较重要的事情。一般建议使用信号槽来完成前后台数据交换，或是采用线程/进程技术，还有一种方式就是在耗时代码中手工调用事件处理函数，来及时处理界面。

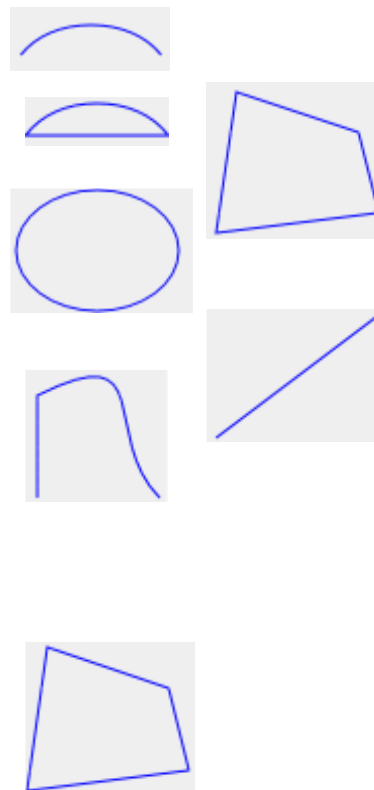
- ▶ 定时事件QTimerEvent
- ▶ 捕获定时事件
  - ▶ void QObject::timerEvent(QTimerEvent \* event)
- ▶ 起停定时事件
  - ▶ int QObject::startTimer(int interval, Qt::TimerType timerType = Qt::CoarseTimer)
  - ▶ void QObject::killTimer(int id)
- ▶ 定时器QTimer
- ▶ 起停定时器
  - ▶ void start(int msec)
  - ▶ void start()
  - ▶ void stop()
- ▶ 设置超时时间
  - ▶ void setInterval(int msec)
- ▶ 超时信号
  - ▶ void QTimer::timeout() [signal]
- ▶ 触发定时信号
  - ▶ void singleShot(int msec, const QObject \* receiver, const char \* member)
  - ▶ void singleShot(int msec, Qt::TimerType timerType, const QObject \* receiver, const char \* member)



- ▶ Qt的2D绘图系统主要是由于三个基本的类构成：QPainter、QPaintEngine、QPaintDevice。
- ▶ QPainter提供了绘图方法
  - ▶ 譬如画点、画线、画圆等等。
- ▶ QPaintDevice提供了QPainter的绘图设备
  - ▶ QWidget、QImage、QPrinter等等绘图场景都是从QPaintDevice继承出来的。
- ▶ QPaintEngine对程序员不透明，提供了不同类型设备的接口，由QPaintDevice和QPainter与其进行交互。
- ▶ 绘图工具：
  - ▶ QPen、QBrush是提供给QPainter绘制方法使用的画笔、画刷。

## ▶ QPainter绘制方法:

- ▶ drawArc: 画弧线
- ▶ drawChord: 画弦
- ▶ drawConvexPolygon: 画凸多边形
- ▶ drawEllipse: 画椭圆
- ▶ drawImage: 画QImage表示的图
- ▶ drawLine: 画线
- ▶ drawPath: 画路径
- ▶ drawPicture: 画QPicture表示的图
- ▶ drawPixmap: 画QPixmap表示的图
- ▶ drawPoint: 画点
- ▶ drawPolygon: 画多边形
- ▶ drawText: 画文字
- ▶ .....



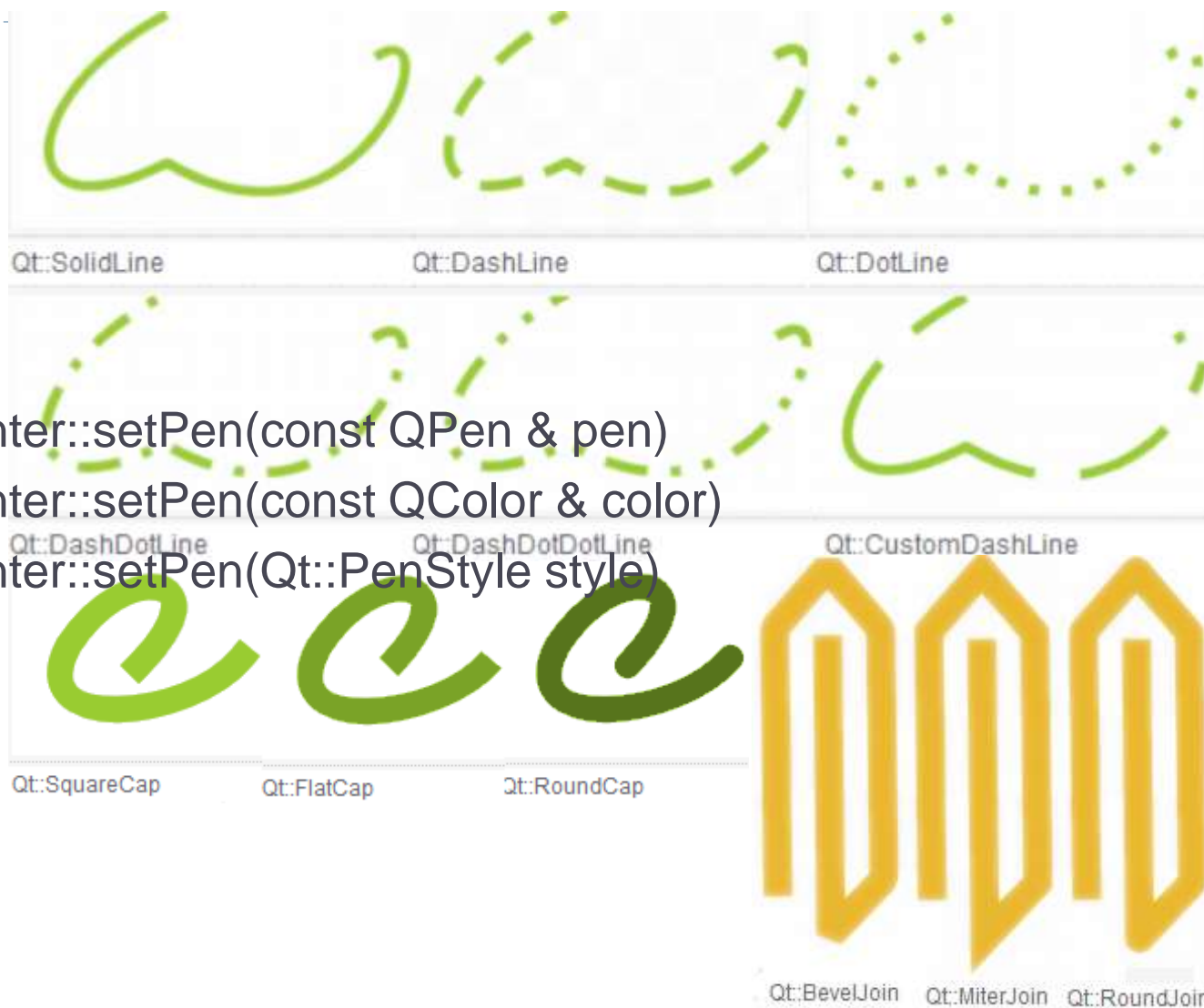
# 2D绘图基础

## ▶ 画笔QPen

- ▶ 画笔样式
- ▶ 端点样式
- ▶ 连接样式

## ▶ 使用方法

- ▶ `void QPainter::setPen(const QPen & pen)`
- ▶ `void QPainter::setPen(const QColor & color)`
- ▶ `void QPainter::setPen(Qt::PenStyle style)`



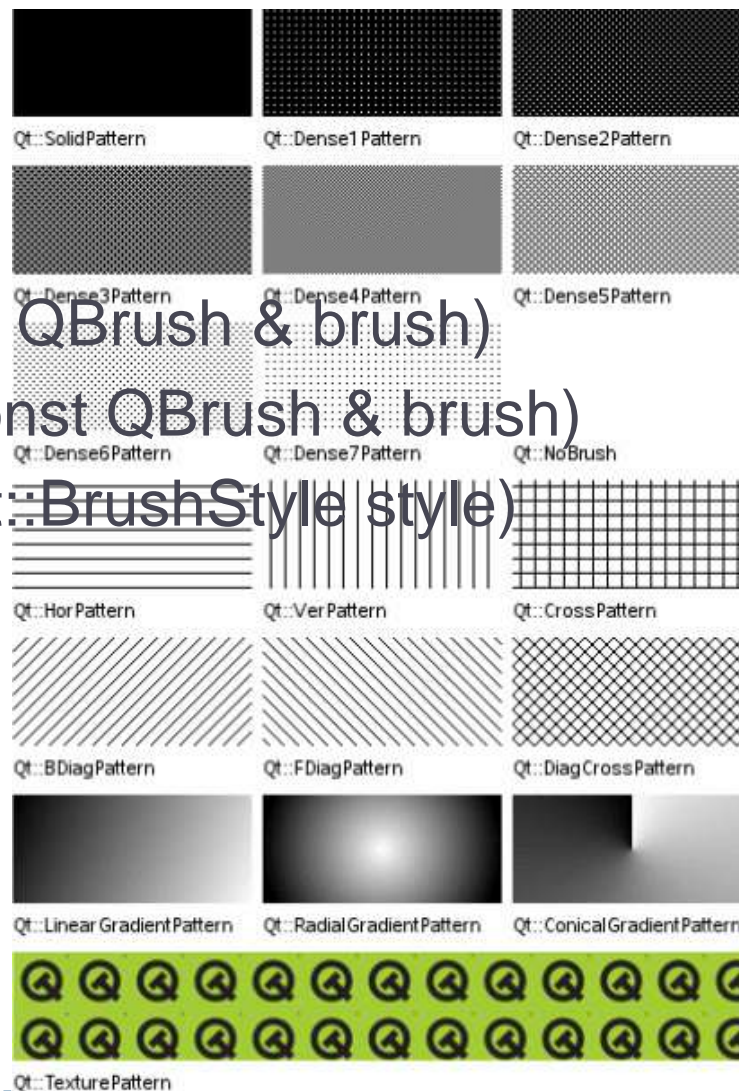
# 2D绘图基础

## ▶ 画刷QBrush

### ▶ 画刷样式

## ▶ 设置方式

- ▶ void QPen::setBrush(const QBrush & brush)
- ▶ void QPainter::setBrush(const QBrush & brush)
- ▶ void QPainter::setBrush(Qt::BrushStyle style)



## ▶ 绘图事件处理函数

- ▶ `void QWidget::paintEvent(QPaintEvent * event)`

- ▶ 当窗口或是部件需要进行绘制的时候，就是触发一个绘图事件，只要重写事件处理函数，就可以定制图形绘制。

## ▶ 重绘

- ▶ `void QWidget::repaint()`

- ▶ `void QWidget::repaint(int x, int y, int w, int h)`

- ▶ `void QWidget::repaint(const QRect & rect)`

- ▶ `void QWidget::repaint(const QRegion & rgn)`

- ▶ `repaint`可以立即使得`paintEvent`被调用

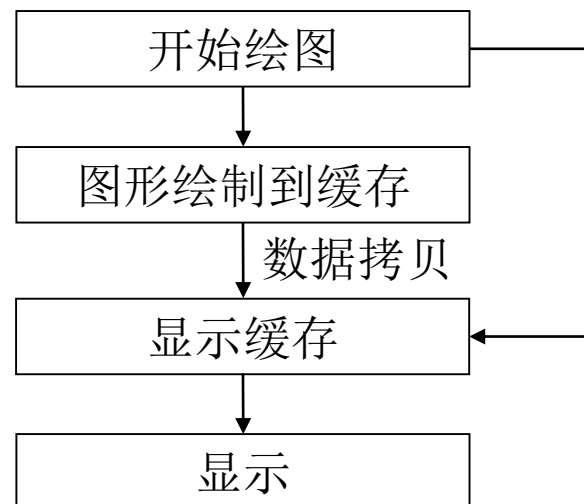
## ▶ 刷新

- ▶ `void QWidget::update()`

- ▶ .....

- ▶ `update`允许Qt来优化速度并且防止闪烁

- ▶ 双缓存绘图
- ▶ Qt中所有的窗口部件默认采用的都是缓存绘图，这样的优点是可以减轻绘制窗口时的闪烁感
- ▶ 如果需要关闭双缓存，自己来管理绘图，使用如下方法：
  - ▶ void
  - ▶ QWidget::setAttribute (Qt::WidgetAttribute attribute, bool on = true)
  - ▶ Qt::WA\_PaintOnScreen
  - ▶ 注意：这个标志仅仅支持X11





线程

华清远见

- ▶ 一般的，系统存在两种耗时任务，CPU密集操作及I/O操作。主流的操作系统都引入了线程的概念。在GUI编程中，前台界面操作和后台耗时任务如果在一个线程中，将使得GUI界面响应缓慢，类似于卡死现象。
- ▶ Qt中管理线程的是QThread
- ▶ `class MyThread : public QThread`
- ▶ `{`
- ▶ `Q_OBJECT`
- ▶ `.....`
- ▶ `protected:`
- ▶ `void run();`
- ▶ `.....`
- ▶ `};`
- ▶ `void MyThread::run() {       ..... }`



- ▶ 线程函数（重写虚函数）
  - ▶ `void QThread::run()`
- ▶ 启动线程
  - ▶ `void QThread::start(Priority priority = InheritPriority)`
- ▶ 线程优先级
  - ▶ `enumPriority { IdlePriority, LowestPriority, LowPriority, NormalPriority, ..., InheritPriority }`
- ▶ 退出线程
  - ▶ `void QThread::exit(int returnCode = 0)`
- ▶ 杀死线程
  - ▶ `void QThread::terminate()`
- ▶ 设置线程栈
  - ▶ `void QThread::setStackSize(uint stackSize)`
- ▶ 等待线程
  - ▶ `bool QThread::wait(unsigned long time = ULONG_MAX)`
- ▶ 休眠
  - ▶ `void msleep(unsigned long msecs)`
  - ▶ `void sleep(unsigned long secs)`
  - ▶ `void usleep(unsigned long usecs)`

## ► 信号

### Signals

```
void finished()
```

```
void started()
```

- 2 signals inherited from QObject

## ► 槽

### Public Slots

```
void quit()
```

```
void start(Priority priority = InheritPriority)
```

```
void terminate()
```

- 1 public slot inherited from QObject

## ▶ 事件处理循环

- ▶ `int QThread::exec()`

## ▶ 槽函数运行在新线程中

- ▶ `void QObject::moveToThread(QThread * targetThread)`

- ▶ QThread应该被看做是OS的线程接口或控制点，而不应该包含需要在新线程中运行的代码。需要运行的代码应该放到一个QObject的子类中，然后将该子类的对象moveToThread()到新线程中。



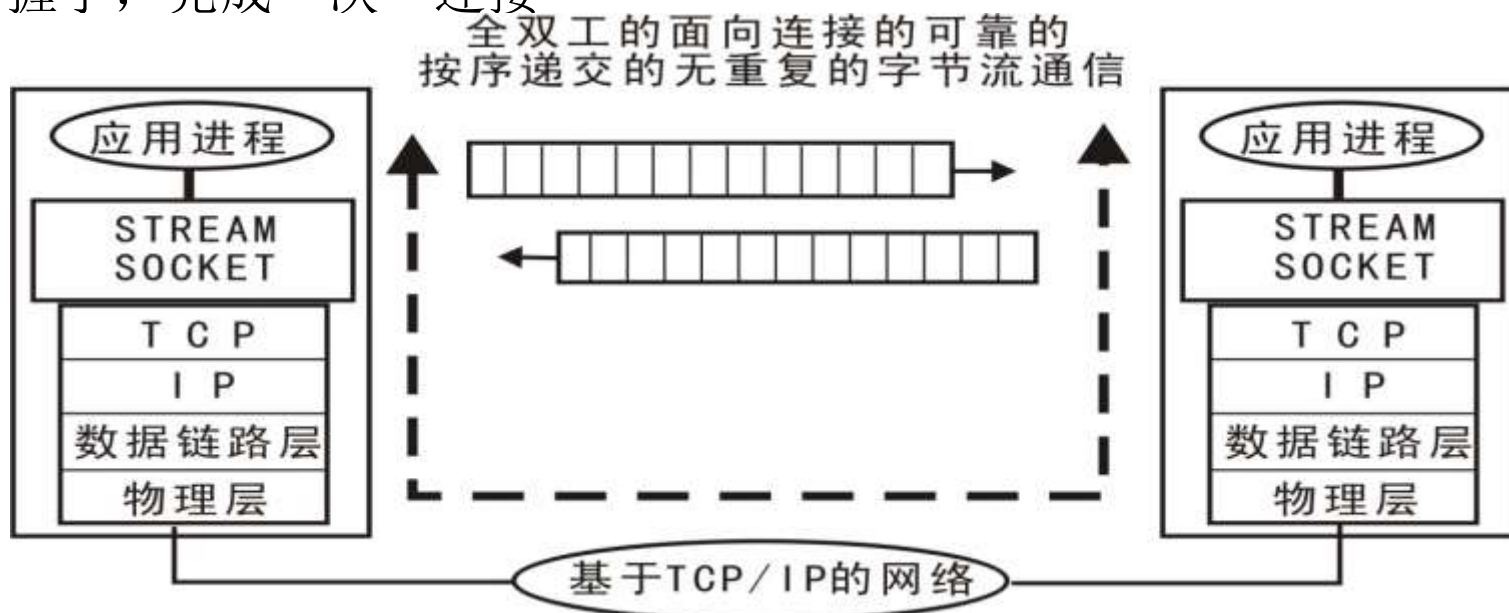
# QT网络编程

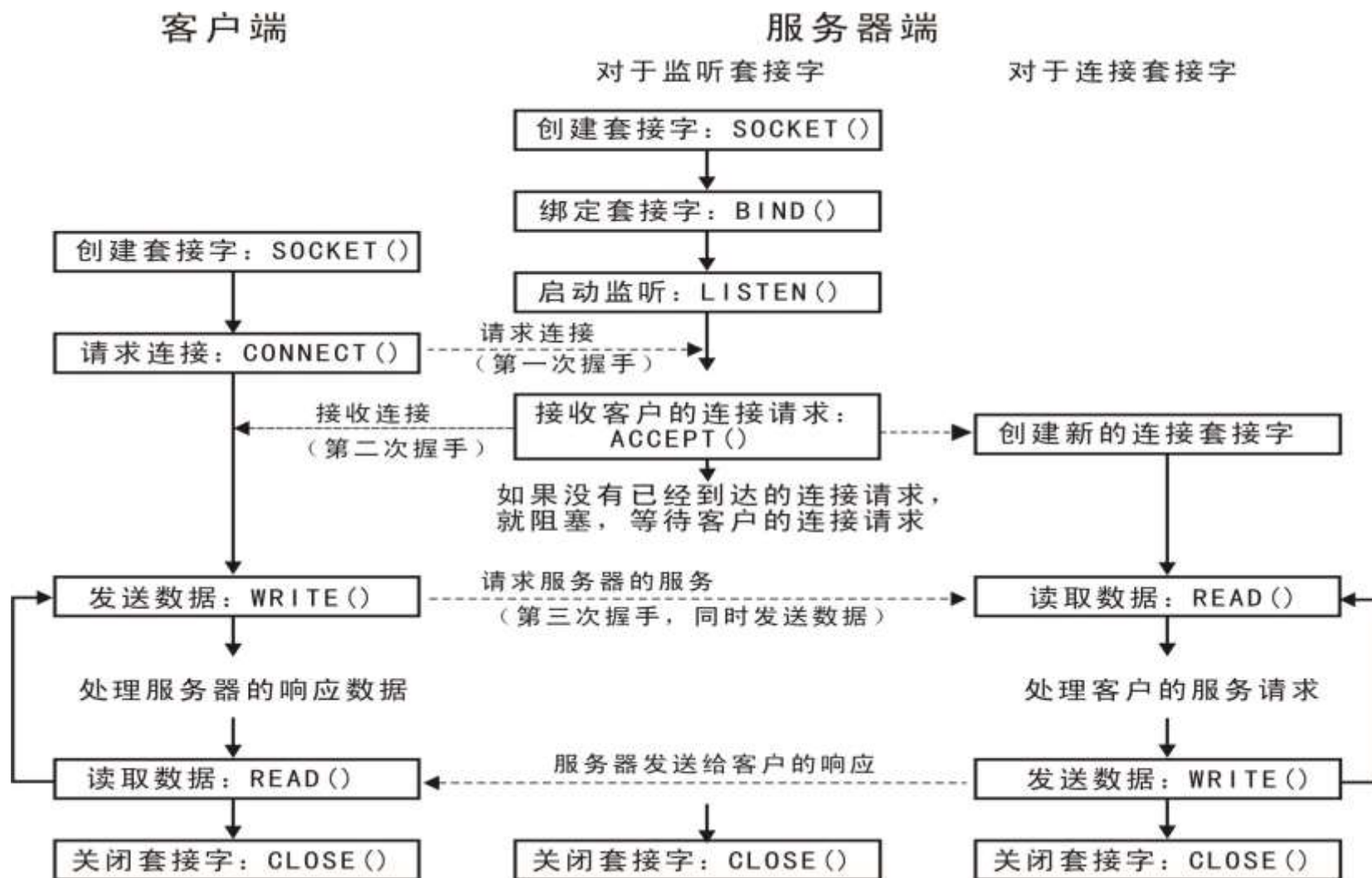
华清远见

- ▶ Qt网络编程
  - ▶ 流式套接字
  - ▶ 数据包套接字

- ▶ Qt提供了基于TCP/IP的套接字编程，引用头文件
  - ▶ `#include <QtNetwork>`
- ▶ Qt的网络模块，在项目文件中添加
  - ▶ `QT += network`
- ▶ 套接字类
  - ▶ 流式: `QTcpSocket`
  - ▶ 数据报: `QUdpSocket`
- ▶ TCP 服务器: `QTcpServer`
- ▶ 使用应用层通用协议进行网络操作
  - ▶ `QNetworkRequest`、`QNetworkReply`、`QNetworkAccessManager`
- ▶ 一个或多个访问端的抽象配置
  - ▶ `QNetworkConfiguration`、`QNetworkConfigurationManager`、`QNetworkSession`

- ▶ 流式套接字是面向连接的可靠通讯一种网络接口
- ▶ 端口：用来标识使用套接字的当前进程
- ▶ IP：用来标识使用套接字的当前主机
- ▶ 绑定：套接字必须跟端口和IP进行绑定
- ▶ 连接及等连接：C/S架构中server端等待client端发起连接，进行三次握手，完成一次“连接”







- ▶ QTcpSocket的基类是QAbstractSocket，QAbstractSocket则从QIODevice继承出来。
- ▶ 绑定：
  - ▶ `bool QAbstractSocket::bind()`
- ▶ ip：
  - ▶ Qt提供QHostAddress来管理IP
- ▶ 连接：
  - ▶ `void QAbstractSocket::connectToHost()`
  - ▶ `bool QAbstractSocket::waitForConnected(int msec = 30000)`
  - ▶ `bool QAbstractSocket::waitForDisconnected(int msec = 30000)`
- ▶ 设置接收缓存
  - ▶ `void QAbstractSocket::setReadBufferSize(qint64 size)`
- ▶ 阻塞读写
  - ▶ `virtual bool waitForBytesWritten(int msec = 30000)`
  - ▶ `virtual bool waitForReadyRead(int msec = 30000)`

# QTcpSocket

## Signals

void **connected**()

void **disconnected**()

void **error**(QAbstractSocket::SocketError *socketError*)

void **hostFound**()

void **proxyAuthenticationRequired**(const QNetworkProxy & *proxy*, QAuthenticator \* *authenticator*)

void **stateChanged**(QAbstractSocket::SocketState *socketState*)

- ▶ QTcpServer基于tcp套接字封装一个server类
- ▶ 设置监听IP和端口
  - ▶ `bool QTcpServer::listen(const QHostAddress & address = QHostAddress::Any, quint16 port = 0)`
- ▶ 处理连接
  - ▶ `int QTcpServer::maxPendingConnections()`
  - ▶ `QTcpSocket * QTcpServer::nextPendingConnection()`
  - ▶ `bool QTcpServer::hasPendingConnections()`
- ▶ 连接
  - ▶ `void QTcpServer::pauseAccepting()`
  - ▶ `void QTcpServer::resumeAccepting()`
  - ▶ `bool QTcpServer::waitForNewConnection(int msec = 0, bool * timedOut = 0)`

## Signals

`void acceptError(QAbstractSocket::SocketError socketError)`

`void newConnection()`