目录

1 Qt 概述

1.1 什么是 Qt

Qt 是一个跨平台的 C++图形用户界面应用程序框架。它为应用程序开发者提供建立图形界面所需的所有功能。它是完全面向对象的，很容易扩展，并且允许真正的组件编程。

1.2 Qt 的发展史

- 1991 年 Qt 最早由芬兰奇趣科技开发
- 1996 年 进入商业领域，它也是目前流行的 Linux 桌面环境 KDE 的基础
- 2008 年 奇趣科技被诺基亚公司收购，Qt 称为诺基亚旗下的编程基础
- 2012 年 Qt 又被 Digia 公司（芬兰一家软件公司）收购
- 2014 年 4 月 跨平台的集成开发环境 Qt Creator3.1.0 发布，同年 5 月 20 日配发了 Qt5.3 正式版，至此 Qt 实现了对 iOS、Android、WP 等各平台的全面支持。

1.3 Qt 的优势

- 跨平台，几乎支持所有的平台
  - Windows – XP、Vista、Win7、Win8、Win2008、Win10
  - Uinux/X11 – Linux、Sun Solaris、HP-UX、Compaq Tru64 UNIX、IBM AIX、SGI IRIX、FreeBSD、BSD/OS、和其他很多 X11 平台
  - Macintosh – Mac OS X
  - Embedded – 有帧缓冲支持的嵌入式 Linux 平台，Windows CE
- 接口简单，容易上手，学习 QT 框架对学习其他框架有参考意义。
- 一定程度上简化了内存回收机制
- 开发效率高，能够快速的构建应用程序。
- 有很好的社区氛围，市场份额在缓慢上升。

- 可以进行嵌入式开发。

1.4 Qt 版本

Qt 按照不同的版本发行，分为商业版和开源版

■ 商业版

为商业软件提供开发，他们提供传统商业软件发行版，并且提供在商业有效期内的免费升级和技术支持服务。

■ 开源的 LGPL 协议版本：

为了开发自有而设计的开放源码软件，它提供了和商业版本同样的功能，在 GNU 通用公共许可下，它是免费的。

目前我们学习上使用的就是这个版本。可以到官网下载最新版本：

http://download.qt.io/archive/qt/

1.5 成功案例

- Linux 桌面环境 KDE（K Desktop Environment）
- WPS Office 办公软件
- Skype 网络电话
- Google Earth 谷歌地球
- VLC 多媒体播放器
- VirtualBox 虚拟机软件
- 等等

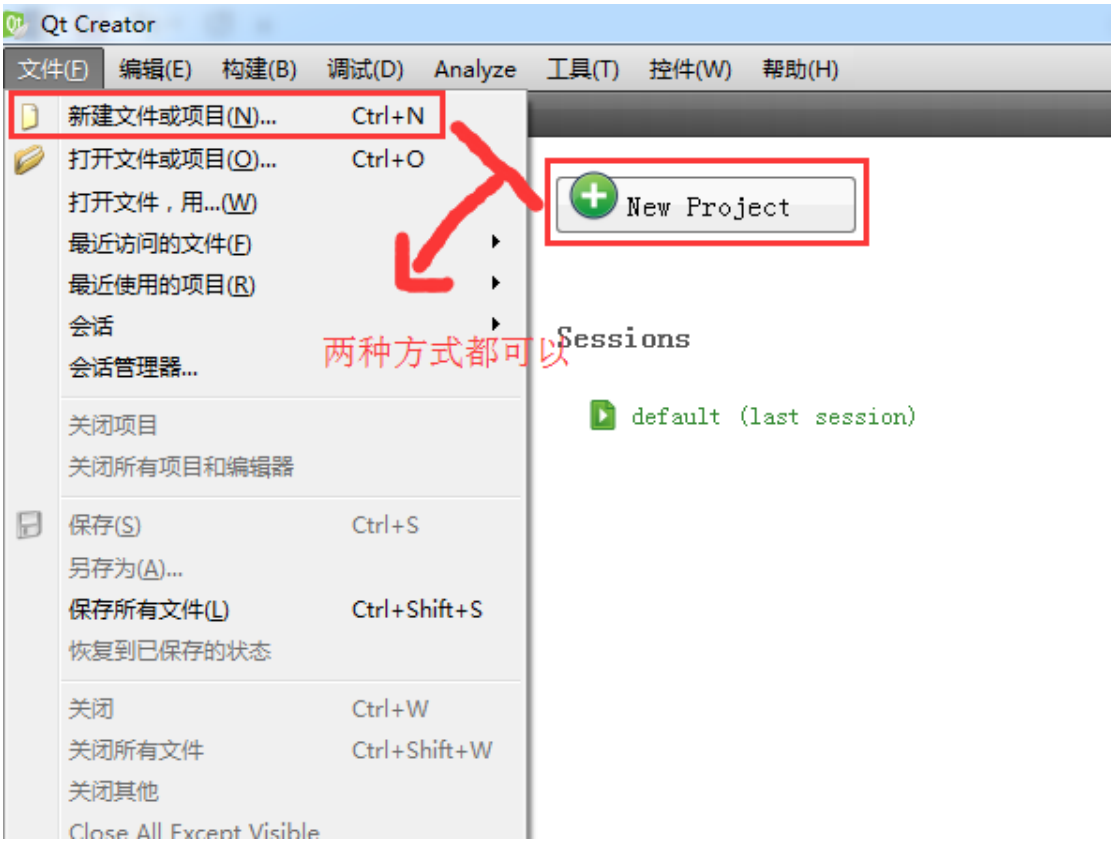
1.6 近期就业行情

职位名称	反馈率	公司名称	职位月薪	工作地点	发布日期
Qt初/中级软件工程师	94%	北京数之行科技有限公司 会员	8000-12000	北京	最新
QT高级软件工程师	100%	北京数之行科技有限公司 会员	10000-18000	北京	最新
C++（QT）中级软件工程师	98%	北京数之行科技有限公司 会员	12000-18000	北京	最新
QT开发工程师	65%	致导科技(北京)有限公司 会员	10000-20000	北京-海淀区	最新
QT开发工程师		北京炫我科技有限公司 会员	6001-8000	北京	最新
QT开发工程师/C++研发工程师		北京乐易考教育科技集团有限公司 会员	10001-15000	北京	最新
智能地图事业部-研发部-QT/C++研发工程师-北京		四维图新 名企	面议	北京	最新
Qt 软件开发工程师	58%	华北计算技术研究所 名企	8001-10000	北京	最近
资深/高级C++研发工程师（QT））		北京思源互联科技有限公司 名企	15000-25000	北京	招聘中
Siemens Healthcare SHC QT Regulatory Affaires Specialist (Beijing) 西门子医疗 法规事务高级专员（工作地点:北京）		Siemens Ltd. China/西门子(中国)有限公司 名企	面议	北京	招聘中
风控经理（QT17002-2）		钱通资本管理(北京)有限公司 名企	12000-18000	北京-朝阳区	招聘中
嵌入式QT开发工程师J11021		北京超思电子技术有限责任公司 名企	10001-15000	北京	最近
移动端测试工程师-MUG（H3D-MUG-QT110003）		北京永航科技有限公司 名企	8000-15000	北京	招聘中
黑盒测试工程师（H3D-QT110001）		北京永航科技有限公司 名企	8000-12000	北京	最近
游戏测试工程师（H3D-MUG-QT110001）		北京永航科技有限公司 名企	8000-15000	北京	最近
游戏测试工程师（H3D-MUG-QT110002）		北京永航科技有限公司 名企	10001-15000	北京	最近
董事长助理（QT17003）		钱通资本管理(北京)有限公司 名企	10000-15000	北京-朝阳区	招聘中

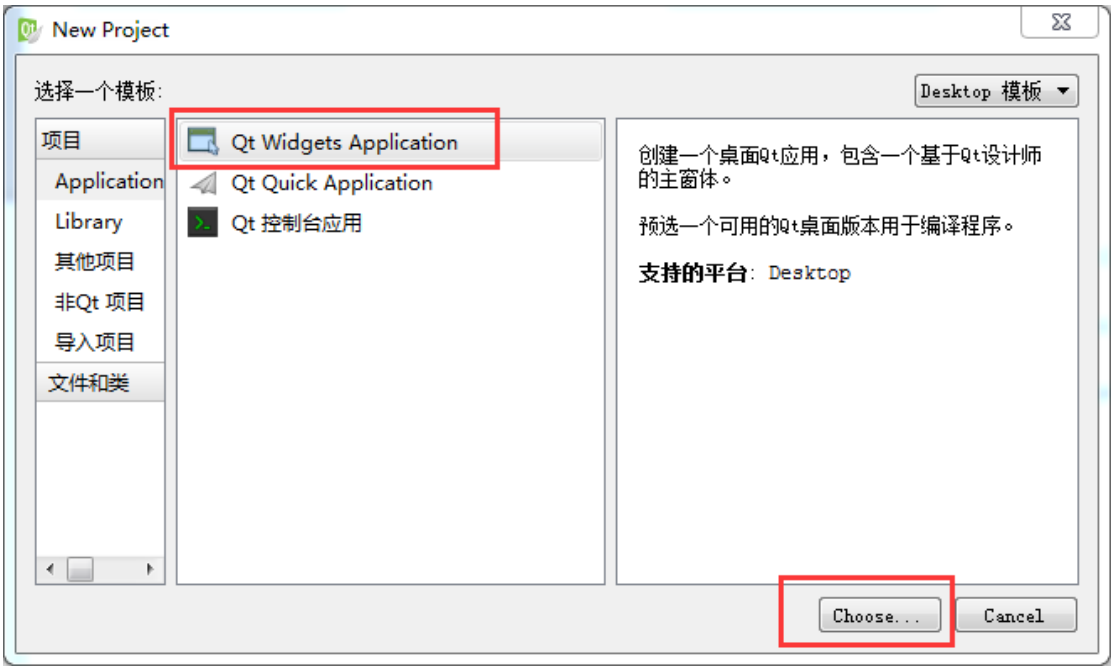
2 创建 Qt 项目

2.1 使用向导创建

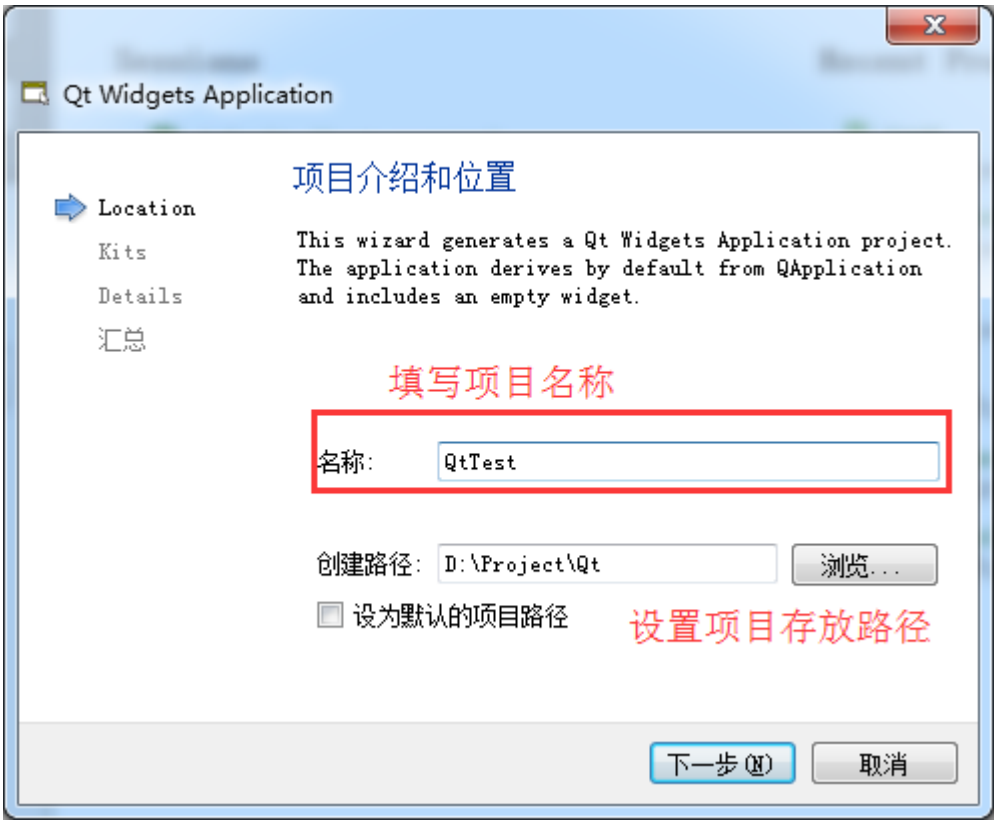
打开 Qt Creator 界面选择 New Project 或者选择菜单栏 【文件】-【新建文件或项目】菜单项



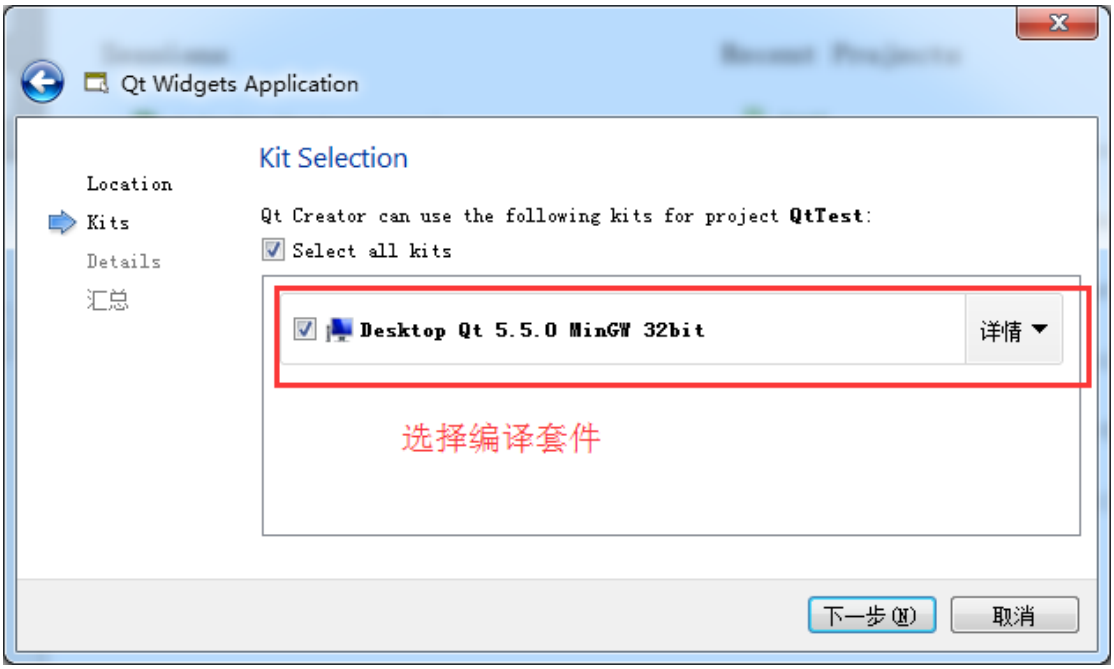
弹出 New Project 对话框，选择 Qt Widgets Application，



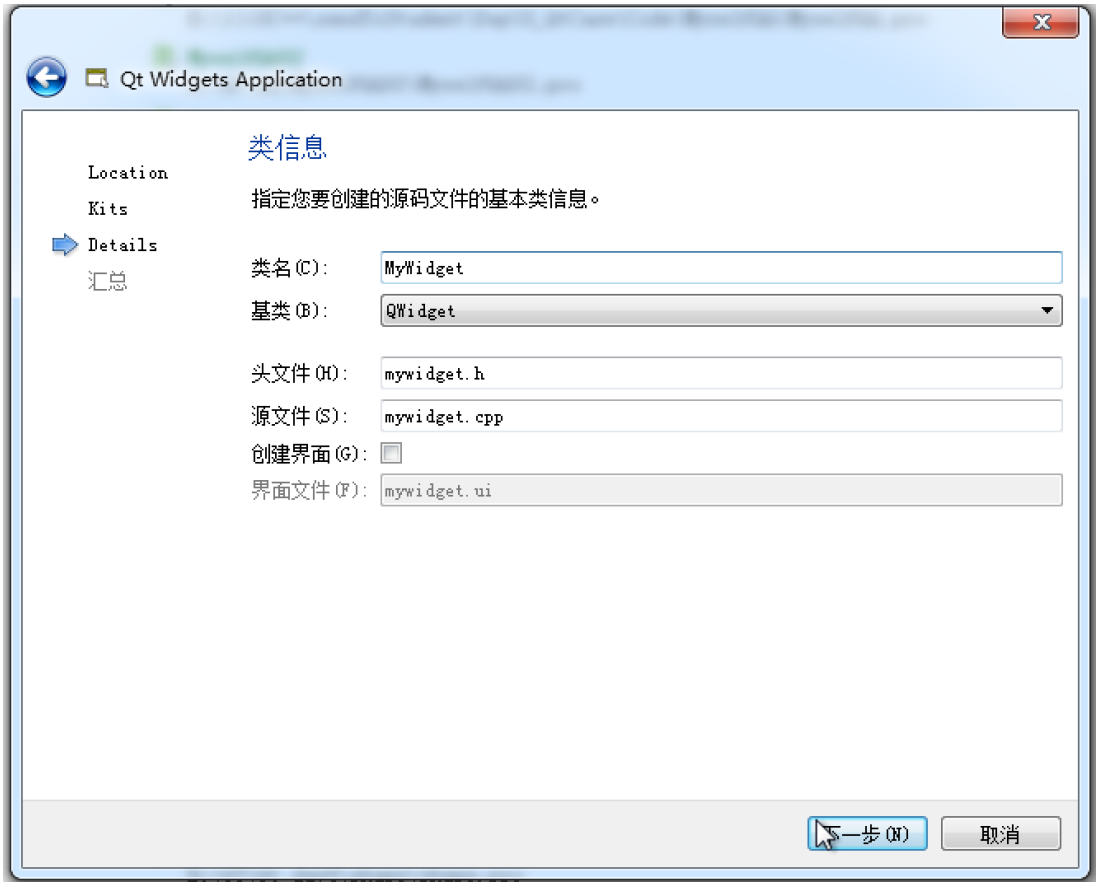
选择【Choose】按钮，弹出如下对话框



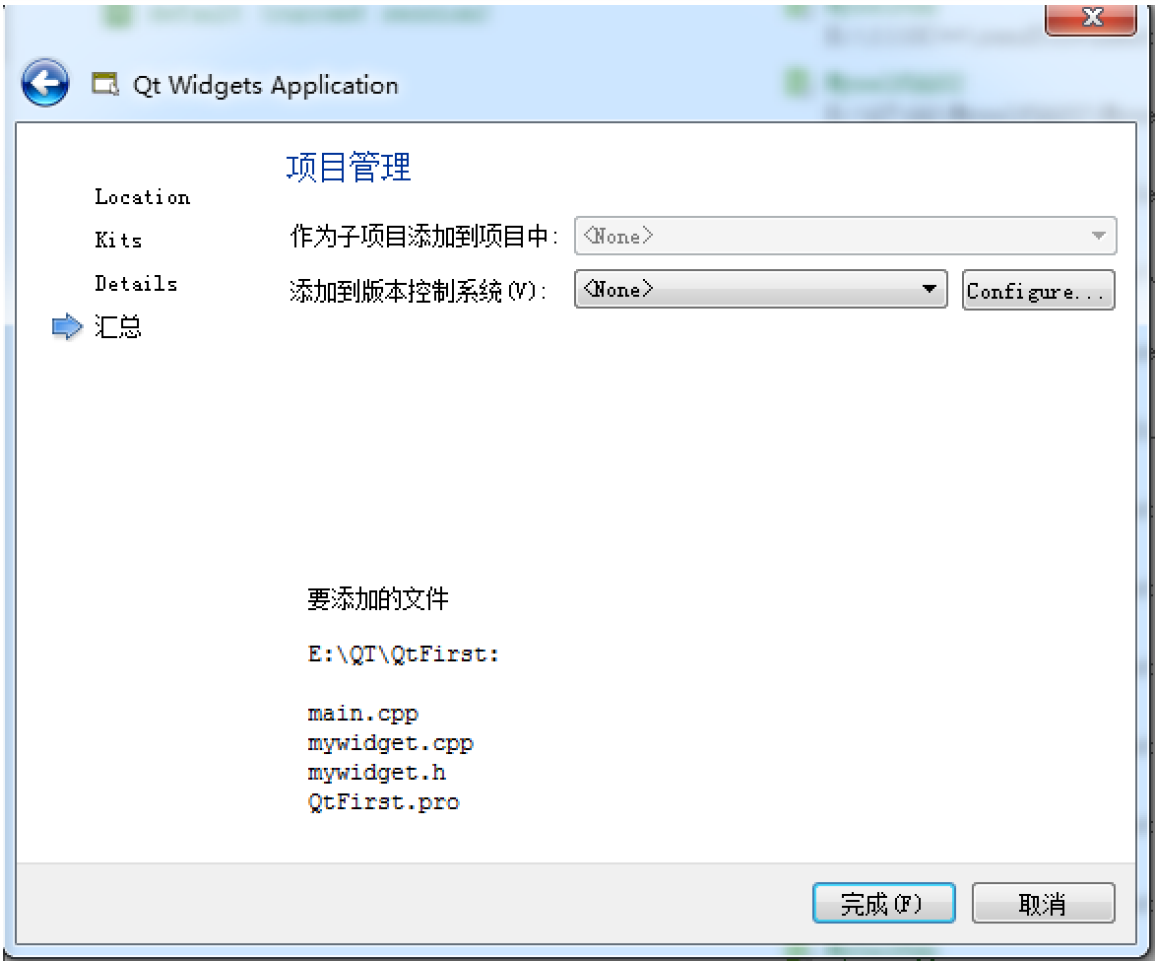
设置项目名称和路径，按照向导进行下一步，



选择编译套件



向导会默认添加一个继承自 QMainWindow 的类，可以在此修改类的名字和基类。默认的基类有 QMainWindow、QWidget 以及 QDialog 三个，我们可以选择 QWidget（类似于空窗口），这里我们可以先创建一个不带 UI 的界面，继续下一步



系统会默认给我们添加 main.cpp、mywidget.cpp、 mywidget.h 和一个.pro 项目文件，点击完成，即可创建出一个 Qt 桌面程序。

2.2 一个最简单的 Qt 应用程序

2.2.1 main 函数中

```
#include "widget.h"
#include <QApplication>


int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    Widget w;
    w.show();
```

```
    return a.exec();
}
```

解释：

- Qt 系统提供的类头文件没有.h 后缀
- Qt 一个类对应一个头文件，类名和头文件名一致
- QApplication 应用程序类
  - 管理图形用户界面应用程序的控制流和主要设置。
  - 是 Qt 生命，一个程序要确保一直运行，就肯定至少得有一个循环，**这就是 Qt 主消息循环**，在其中完成来自窗口系统和其它资源的**所有事件消息处理和调度**。它也处理**应用程序的初始化和结束**，并且**提供对话管理**。
  - 对于任何一个使用 Qt 的图形用户界面应用程序，都正好存在一个 QApplication 对象，不论这个应用程序在同一时刻有多少个窗口。
- a.exec()

  程序进入消息循环，等待对用户输入进行响应。这里 main()把控制权转交给 Qt，Qt 完成事件处理工作，当应用程序退出的时候 exec()的值就会返回。**在 exec()中，Qt 接受并处理用户和系统的事件并且把它们传递给适当的窗口部件。**

2.2.2 类头文件

```
#include <QWidget>

class MyWidget : public QWidget
{
    //引入 Qt 信号和槽机制的一个宏
    Q_OBJECT
```

```
public:
//构造函数中 parent 是指父窗口
//如果 parent 是 0，那么窗口就是一个顶层的窗口
    MyWidget (QWidget *parent = 0);
    ~ MyWidget ();
};
```

2.3 .pro 文件

.pro 就是工程文件(project)，它是 qmake 自动生成的用于生产 makefile 的配置文件。类似于 VS 中的.sln 和 vsproj 文件

以下是.pro 文件的一个案例：

```
#引入 Qt 的模块，core gui
QT       += core gui
#如果 qt 版本大于 4，那么引入 widgets 模块
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
#生成最终文件的文件名，可执行文件 exe
TARGET = 01_MyWidget
#项目类型，生成什么类型的文件，可执行程序还是库文件
TEMPLATE = app
#要编译的源文件列表
SOURCES += \
        main.cpp \
        mywidget.cpp
#要编译的头文件列表
HEADERS += \
        mywidget.h
```

.pro 文件的规则：

- 注释

  从“#”开始，到这一行结束。

- 模块引入

  QT += 模块名，表示当前项目引入 Qt 哪些模块。

  引入模块的意思就简单理解为引入 C/C++头文件搜索路径，如果没引入对应模块就使用该头文件的话会报错说找不到该头文件。当然不必要的模块还是别引入，因为引入模块不仅仅是引入头文件搜索路径那么简单，还包括引入连接的库等一系列操作，会让程序变臃肿。

  Qt 详细模块有哪些可以参照附录。

- 模板变量告诉 qmake 为这个应用程序生成哪种 makefile。下面是可供使用的选择：**TEMPLATE** = app

  - app -建立一个应用程序的 makefile。这是默认值，所以如果模板没有被指定，这个将被使用。
  - lib - 建立一个库的 makefile。
  - vcapp - 建立一个应用程序的 VisualStudio 项目文件。
  - vclib - 建立一个库的 VisualStudio 项目文件。
  - subdirs -这是一个特殊的模板，它可以创建一个能够进入特定目录并且为一个项目文件生成 makefile 并且为它调用 make 的 makefile。

- 指定生成的应用程序名：

  TARGET = QtDemo

- 工程中包含的头文件

  HEADERS += include/painter.h

- 工程中包含的.ui 设计文件

  FORMS += forms/painter.ui

- 工程中包含的源文件

  SOURCES += sources/main.cpp sources

- 工程中包含的资源文件

RESOURCES += qrc/painter.qrc

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

这条语句的含义是，如果 QT_MAJOR_VERSION 大于 4（也就是当前使用的 Qt5 及更高版本）需要增加 widgets 模块。如果项目仅需支持 Qt5，也可以直接添加“QT += widgets”一句。不过为了保持代码兼容，最好还是按照 QtCreator 生成的语句编写。

- 配置信息

CONFIG 用来告诉 qmake 关于应用程序的配置信息。

CONFIG += c++11 //使用 c++11 的特性（qt5.6 以上版本默认使用 C++11）

在这里使用“+=”，是因为我们添加我们的配置选项到任何一个已经存在中。这样做比使用“=”那样替换已经指定的所有选项更安全。

2.4 命名规范

类名：单词首字母大写，单词和单词之间直接连接，无需连接字符 。

```
MyClass，QPushButton
class MainWindow
```

Qt 中内置的类型，头文件和类命名同名。

```
#include <QString>
QSring str;

#include <QWidget>
QWidget w;
```

函数名字，变量名：首字母小写，之后每个单词首字母大写，单词和单词之间直接连接，无需连接字符

```
void connectTheSignal();
```

类的成员变量设置函数用使用 set+成员变量名，获取成员变量的函数直接

用成员变量名(如果是 bool 类型,有可能会用一些表示状态的术语,如 isVisilble, hasFocus)：

```
//普通成员变量设置和获取
void setText(QString text);
QString text()const;
//bool 的成员变量设置和获取
void setEnabled(bool enabled);
bool isEnabled()const;
```

2.5 QtCreator 常用快捷键

运行 ctrl +R

编译 ctrl +B

帮助文档 F1 ，点击 F1 两次跳到帮助界面

跳到符号定义 F2 或者 ctrl + 鼠标点击

注释 ctrl+/

字体缩放 ctrl + 鼠标滚轮

整行移动代码 ctrl + shift + ↑或↓

自动对齐 ctrl + i

同名之间的.h 和.cpp 文件跳转 F4

3 Qt 按钮小程序

3.1 按钮的创建和父子关系

在 Qt 程序中，最常用的控件之一就是按钮了，首先我们来看下如何创建一个按钮

```
#include <QPushButton>


QPushButton * btn = new QPushButton;
```

```
    //设置父亲
    btn->setParent(this);
    //设置文字
    btn->setText("德玛西亚");
    //移动位置
    btn->move(100,100);

    //第二种创建
    QPushButton * btn2 = new QPushButton("孙悟空",this);
    //重新指定窗口大小
    this->resize(600,400);

    //设置窗口标题
    this->setWindowTitle("第一个项目");

    //限制窗口大小
    this->setFixedSize(600,400);
```

上面代码中，一个按钮其实就是一个 QPushButton 类的对象，如果只是创建出对象，是无法显示到窗口中的，所以我们需要依赖一个父窗口，也就是指定一个父亲，利用 setParent 函数或者按钮创建的时候通过构造函数传参，此时我们称两个窗口建立了**父子关系**。在有父窗口的情况下，窗口调用 show 会显示在父窗口中，如果没有父窗口，那么窗口调用 show 显示的会是一个顶层的窗口（顶层窗口是能够在任务栏中找到的，不依赖于任何一个窗口而独立存在）（按钮也是继承于 QWidget，也属于窗口）。

如果想设置按钮上显示的文字可以用 setText，移动按钮位置用 move。

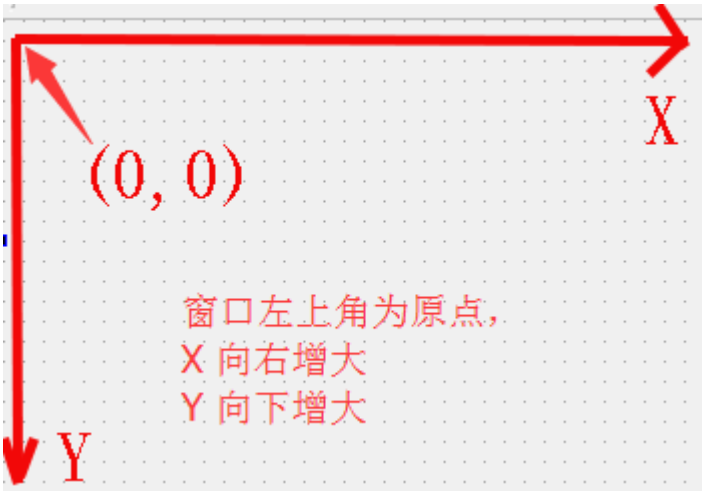
对于窗口而言，我们可以修改左上角窗口的标题 setWindowTitle，重新指定窗口大小：resize，或者设置固定的窗口大小 setFixedSize。

3.2 Qt 窗口坐标体系

通过以上代码可以看出 Qt 的坐标体系。

以左上角为原点（0,0），以向右的方向为 x 轴的正方向，以向下方向为 y 轴

的正方向。



对于嵌套窗口，其坐标是**相对于父窗口**来说的。顶层窗口的父窗口就是屏幕。

3.3 对象树模型

QObject 是 Qt 里边绝大部分类的根类

- QObject 对象之间是以对象树的形式组织起来的。
  - 当两个 QObject（或子类）的对象建立了**父子关系**的时候。子对象就会加入到父对象的一个成员变量叫 children（孩子）的 list（列表）中。
  - **当父对象析构的时候**，这个列表中的**所有对象也会被析构**。（注意，这里是说父对象和子对象，不要理解成父类和子类）
- QWidget 是能够在屏幕上显示的一切组件的父类
  - **QWidget 继承自 QObject，因此也继承了这种对象树关系。一个孩子自动地成为父组件的一个子组件**。我们向某个窗口中添加了一个按钮或者其他控件（建立父子关系），当用户关闭这个窗口的时候，该窗口就会被析构，之前添加到他上边的按钮和其他控件也会被一同析构。这个结果也是我们开发人员所期望的。
  - 当然，**我们也可以手动删除子对象。当子对象析构的时候会发出一个信号 destroyed，父对象收到这个信号之后就会从 children 列表中将它剔除。**比如，当我们删除了一个按钮时，其所在的主窗口会自动将该按钮从其子对象列表（children）中删除，并且自动调整屏幕显示，按钮在屏

幕上消失。当这个窗口析构的时候，children 列表里边已经没有这个按钮子对象，所以我们手动删除也不会引起程序错误。

Qt 引入对象树的概念，在一定程度上解决了内存问题。

- 对象树中对象的顺序是没有定义的。这意味着，销毁这些对象的顺序也是未定义的。
- 任何对象树中的 QObject 对象 delete 的时候，如果这个对象有 parent，则自动将其从 parent 的 children()列表中删除；如果有孩子，则自动 delete 每一个孩子。Qt 保证没有 QObject 会被 delete 两次，这是由析构顺序决定的。

如果 QObject 在栈上创建，Qt 保持同样的行为。正常情况下，这也不会发生什么问题。来看下下面的代码片段：

```
{
    QWidget window;
    QPushButton quit("Quit", &window);
}
```

作为父组件的 window 和作为子组件的 quit 都是 QObject 的子类（事实上，它们都是 QWidget 的子类，而 QWidget 是 QObject 的子类）。这段代码是正确的，quit 的析构函数不会被调用两次，因为标准 C++要求，**局部对象的析构顺序应该按照其创建顺序的相反过程**。因此，这段代码在超出作用域时，会先调用 quit 的析构函数，将其从父对象 window 的子对象列表中删除，然后才会再调用 window 的析构函数。

但是，如果我们使用下面的代码：

```
{
    QPushButton quit("Quit");
    QWidget window;
    quit.setParent(&window);
}
```

情况又有所不同，析构顺序就有了问题。我们看到，在上面的代码中，作为父对象的 window 会首先被析构，因为它是最后一个创建的对象。在析构过程中，它会调用子对象列表中每一个对象的析构函数，也就是说， quit 此时就被析构了。然后，代码继续执行，在 window 析构之后，quit 也会被析构，因为 quit 也是一个局部变量，在超出作用域的时候当然也需要析构。但是，这时候已经是

第二次调用 quit 的析构函数了，C++ 不允许调用两次析构函数，因此，程序崩溃了。

由此我们看到，Qt 的对象树机制虽然帮助我们在一定程度上解决了内存问题，但是也引入了一些值得注意的事情。这些细节在今后的开发过程中很可能时不时跳出来烦扰一下，所以，我们最好从开始就养成良好习惯，在 Qt 中，尽量在构造的时候就指定 parent 对象，并且大胆在堆上创建。

4 信号和槽机制

信号：各种事件

槽： 响应信号的动作

当某个事件发生后，如某个按钮被点击了一下，它就会发出一个被点击的信号（signal）。

某个对象接收到这个信号之后，就会做一些相关的处理动作（称为槽 slot）。

但是 Qt 对象不会无故收到某个信号，要想让一个对象收到另一个对象发出的信号，这时候需要建立连接（connect）

4.1 系统自带的信号和槽

下面我们完成一个小功能，上面我们已经学习了按钮的创建，但是还没有体现出按钮的功能，按钮最大的功能也就是点击后触发一些事情，比如我们点击按钮，就把当前的窗口给关闭掉，那么在 Qt 中，这样的功能如何实现呢？

其实两行代码就可以搞定了，我们看下面的代码

```
QPushButton * quitBtn = new QPushButton("关闭窗口",this);
connect(quitBtn,&QPushButton::clicked,this,&MyWidget::close);
```

第一行是创建一个关闭按钮，这个之前已经学过，第二行就是核心了，也就是信号槽的使用方式

connect 函数是建立信号发送者、信号、信号接收者、槽四者关系的函数：

connect(sender, signal, receiver, slot);

参数解释：

- sender：信号发送者
- signal：信号
- receiver：信号接收者

- slot：接收对象在接收到信号之后所需要调用的函数（槽函数）

这里要注意的是 connect 的四个参数都是指针，信号和槽是函数指针。

系统自带的信号和槽如何查找呢，这个就需要利用帮助文档了，在帮助文档中比如我们上面的按钮的点击信号，在帮助文档中输入 QPushButton，首先我们可以在 Contents 中寻找关键字 signals，信号的意思，但是我们发现并没有找到，这时候我们应该想到也许这个信号的被父类继承下来的，因此我们去他的父类 QAbstractButton 中就可以找到该关键字，点击 signals 索引到系统自带的信号有如下几个

Signals

```
void    clicked(bool checked = false)
void    pressed()
void    released()
void    toggled(bool checked)
```

- 3 signals inherited from QWidget
- 2 signals inherited from QObject

这里的 clicked 就是我们要找到，槽函数的寻找方式和信号一样，只不过他的关键字是 slot。

4.2 自定义信号和槽

Qt 框架默认提供的标准信号和槽不足以完成我们日常应用开发的需求，比如说点击某个按钮让另一个按钮的文字改变，这时候标准信号和槽就没有提供这样的函数。但是 Qt 信号和槽机制提供了允许我们自己设计自己的信号和槽。

4.2.1 自定义信号使用条件

1) 声明在类的 signals 域下
2) 没有返回值，void 类型的函数
3) 只有函数声明，没有定义
4) 可以有参数，可以重载
5) 通过 emit 关键字来触发信号，形式：emit object->sig(参数);

4.2.2 自定义槽函数使用条件

1) qt4 必须声明在 private/public/protected slots 域下面，qt5 之后可以声明 public 下，同时还可以是静态的成员函数，全局函数，lambda 表达式
2) 没有返回值，void 类型的函数
3) 不仅有声明，还得要有实现
4) 可以有参数，可以重载

4.2.3 使用自定义信号和槽

定义场景：下课了，老师跟同学说肚子饿了（信号），学生请老师吃饭（槽）

首先定义一个学生类和老师类：

老师类中声明信号 饿了 hungry

```
signals:
        void hungry();
```

学生类中声明槽 请客 treat

```
    public slots:
        void treat();
```

在窗口中声明一个公共方法下课，这个方法的调用会触发老师饿了这个信号，而响应槽函数学生请客

```
    void MyWidget::ClassIsOver()
{
    //发送信号
    emit teacher->hungry();
}
```

学生响应了槽函数，并且打印信息

//自定义槽函数 实现

```
void Student::treat()
{
        qDebug() << "Student treat teacher";
}
```

在窗口中连接信号槽

```
teacher = new Teacher(this);
student = new Student(this);
connect(teacher,&Teacher::hungury,student,&Student::treat);
```

并且调用下课函数，测试打印出相应 log

自定义的信号 hungry 带参数，需要提供重载的自定义信号和 自定义槽

```
void hungry(QString name);   自定义信号
void treat(QString name );     自定义槽
但是由于有两个重名的自定义信号和自定义的槽，直接连接会报错，所以需要利用函数指针来指向函数地址， 然后在做连接
void (Teacher:: * teacherSingal)(QString) = &Teacher:: hangry;
void (Student:: * studentSlot)(QString) = &Student::treat;
connect(teacher,teacherSingal,student,studentSlot);
也可以使用 static_cast 静态转换挑选我们要的函数
connect(
    teacher,
    static_cast<void(Teacher:: *)(QString)>(&Teacher:: hangry),
    student,
    static_cast<void(Student:: *)(QString)>(& Student::treat));
```

4.3 信号和槽的拓展

- **一个信号可以和多个槽相连**

如果是这种情况，这些槽会一个接一个的被调用，但是**槽函数调用顺序是不确定的**。像上面的例子，可以将一个按钮点击信号连接到关闭窗口

的槽函数，同时也连接到学生请吃饭的槽函数，点击按钮的时候可以看到关闭窗口的同时也学生请吃饭的 log 也打印出来。

- **多个信号可以连接到一个槽**

只要任意一个信号发出，这个槽就会被调用。如：一个窗口多个按钮都可以关闭这个窗口。

- **一个信号可以连接到另外的一个信号**

当第一个信号发出时，第二个信号被发出。除此之外，这种信号-信号的形式和信号-槽的形式没有什么区别。注意这里还是使用 connect 函数，只是信号的接收者和槽函数换成另一个信号的发送者和信号函数。如上面老师饿了的例子，可以新建一个按钮 btn。

```
connect(btn,&QPushButton::clicked,teacher,&Teacher::hungry);
```

- **信号和槽可以断开连接**

可以使用 **disconnect** 函数，当初建立连接时 connect 参数怎么填的，disconnect 里边 4 个参数也就怎么填。这种情况并不经常出现，因为**当一个对象 delete 之后，Qt 自动取消所有连接到这个对象上面的槽。**

- **信号和槽函数参数类型和个数必须同时满足两个条件**

1) 信号函数的参数个数必须大于等于槽函数的参数个数

2) 信号函数的参数类型和槽函数的参数类型必须一一对应

4.4 Qt4 版本的信号槽写法

```
connect(
    teacher,
    SIGNAL(hungry(QString)),
    student,
    SLOT(treat(QString))
);
```

这里使用了 **SIGNAL 和 SLOT 这两个宏，宏的参数是信号函数和槽函数的函数原型。**

因为直接填入了函数原型，所有这里边编译不会出现因为重载导致的函数指针二义性的问题。但问题是如果函数原型填错了，或者不符合信号槽传参个数类型约定，编译期间也不会报错，只有运行期间才会看到错误 log 输出。

原因就是这两个宏将后边参数（函数原型）转化成了字符串。目前编译器还没有那么智能去判断字符串里边的内容符不符合运行条件。

4.5 Lambda 表达式

C++11 中的 Lambda 表达式**用于定义匿名的函数对象**，以简化编程工作。首先看一下 Lambda 表达式的基本构成：

分为四个部分：[局部变量捕获列表]、(函数参数)、函数额外属性设置 opt、函数返回值->retype、{函数主体}

```
[capture](parameters) opt ->retType
{
    ……;
}
```

4.5.1 局部变量引入方式

[ ]，标识一个 **Lambda 的开始**。由于 lambda 表达式可以定义在某一个函数体 A 里边，所以 lambda 表达式有可能会去访问 A 函数中的局部变量。中括号里边内容是描述了在 lambda 表达式里边可以使用的外部局部变量的列表：

- []

  表示 lambda 表达式不能访问外部函数体的任何局部变量

- [a]

  在函数体内部使用值传递的方式访问 a 变量

- [&b]

  在函数体内部使用引用传递的方式访问 b 变量

- [=]

  函数外的所有局部变量都通过值传递的方式使用，函数体内使用的是副本

- [&]

引用的方式使用 lambda 表达式外部的所有变量

- [=, &foo]

  foo 使用引用方式，其余是值传递的方式

- [&,foo]

  foo 使用值传递方式，其余是引用传递的方式

- [this]

在函数内部可以使用类的成员函数和成员变量，=和&形式也都会默认引入

由于引用方式捕获对象会有局部变量释放了而 lambda 函数还没有被调用的情况。如果执行 lambda 函数那么引用传递方式捕获进来的局部变量的值不可预知。

所以在无特殊情况下建议使用[=](){}的形式

4.5.2 函数参数

(params)表示 lambda 函数对象接收的参数，类似于函数定义中的小括号表示函数接收的参数类型和个数。参数可以通过按值（如：(int a,int b)）和按引用（如：(int &a,int &b)）两种方式进行传递。函数参数部分可以省略，省略后相当于无参的函数。

4.5.3 选项 Opt

Opt 部分是可选项，最常用的是 mutable 声明，这部分可以省略。外部函数局部变量通过值传递引进来时，其默认是 const，所以不能修改这个局部变量的拷贝，加上 mutable 就可以

```
int a = 10 ;
[=]()
{
    a=20;//编译报错，a 引进来是 const
}
```

```
[=]()mutable
{
    a=20;//编译成功
};
```

4.5.4 函数返回值 ->retType

->retType，标识 lambda 函数返回值的类型。这部分可以省略，但是省略了并不代表函数没有返回值，编译器会自动根据函数体内的 return 语句判断返回值类型，但是如果有多条 return 语句，而且返回的类型都不一样，编译会报错，如：

```
[=]()mutable
{
    int b = 20;
    float c = 30.0;
    if(a>0)
        return b;
    else
        return c;//编译报错，两条 return 语句返回类型不一致
};
```

4.5.5 是函数主体{}

{}，标识函数的实现，这部分不能省略，但函数体可以为空。

4.5.6 槽函数使用 Lambda 表达式

以 QPushButton 点击事件为例：

```
connect(btn,&QPushButton::clicked,[=](){
    qDebug()<<"Clicked";
});
```

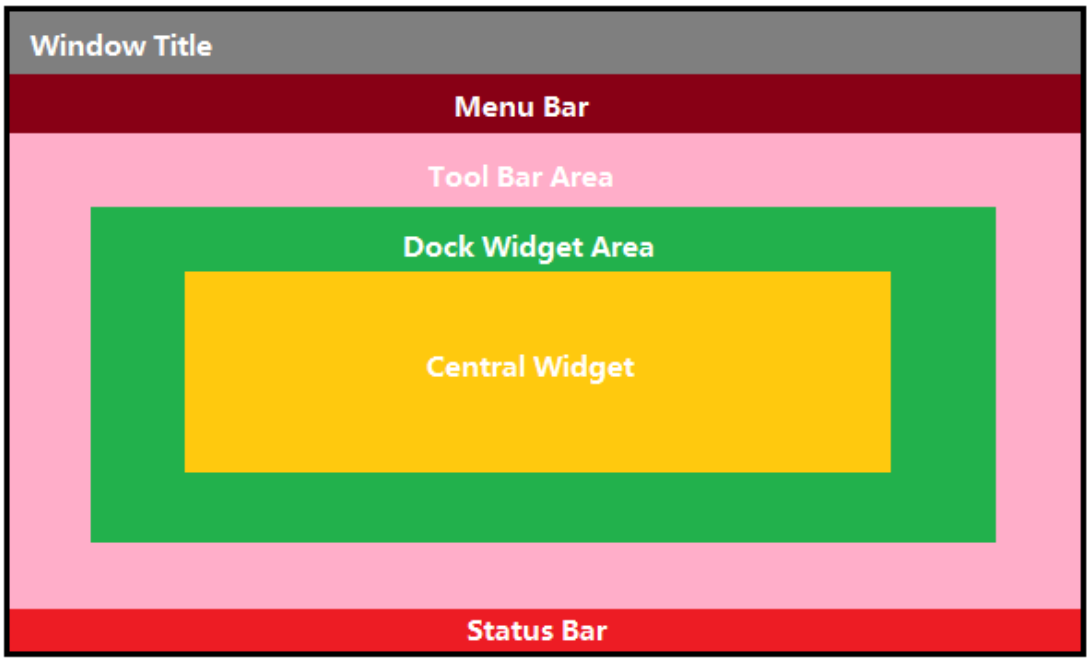
这里可以看出使用 Lambda 表达式作为槽的时候不需要填入信号的接收者。当点击按钮的时候，clicked 信号被触发，lambda 表达式也会直接运行。当然 lambda 表达式还可以指定函数参数，这样也就能够接收到信号函数传递过来的参数了。

由于 lambda 表达式比我们自己自定义槽函数要方便而且灵活得多，所以在实现槽函数的时候优先考虑使用 Lambda 表达式。一般我们的使用习惯也是 lambda 表达式外部函数的局部变量全部通过值传递捕获进来，也就是：

[=](){ }的形式

5 QMainWindow

QMainWindow 是一个为用户提供主窗口程序的类，包含一个菜单栏（menu bar）、多个工具栏(tool bars)、多个停靠部件(dock widgets)、一个状态栏(status bar)及一个中心部件(central widget)，是许多应用程序的基础，如文本编辑器，图片编辑器等。



5.1 菜单栏

一个主窗口最多只有一个菜单栏。位于主窗口顶部、主窗口标题栏下面。

- 通过 QMainWindow 类的 menubar（）函数获取主窗口菜单栏指针，如果当前窗口没有菜单栏，该函数会自动创建一个。

```
QMenuBar *    menuBar() const;
```

- 创建菜单，调用 QMenu 的成员函数 addMenu 来添加菜单

```
QAction* addMenu(QMenu * menu);
QMenu* addMenu(const QString & title);
QMenu* addMenu(const QIcon & icon, const QString & title);
```

- 创建菜单项，调用 QMenu 的成员函数 addAction 来添加菜单项

```
    QAction* activeAction() const;
    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);
```

Qt 并没有专门的菜单项类，只是使用一个 QAction 类，抽象出公共的动作。当我们把 QAction 对象添加到菜单，就显示成一个菜单项，添加到工具栏，就显示成一个工具按钮。用户可以通过点击菜单项、点击工具栏按钮、点击快捷键来激活这个动作。

5.2 工具栏

主窗口的工具栏上可以有多个工具条，通常采用一个菜单对应一个工具条的的方式，也可根据需要进行工具条的划分。

- 调用 QMainWindowd 对象的成员函数 addToolBar（）,该函数每次调用都会创建一个新的工具栏，并且返回该工具栏的指针。
- 插入属于工具条的项，这时工具条上添加项也是用 QAction。

  通过 QToolBar 类的 addAction 函数添加。
- 工具条是一个可移动的窗口，它的停靠区域由 QToolBar 的 allowAreas 决定，包括（以下值可以通过查帮助文档 allowAreas 来索引到）：
  - Qt::LeftToolBarArea 停靠在左侧
  - Qt::RightToolBarArea 停靠在右侧
  - Qt::TopToolBarArea 停靠在顶部
  - Qt::BottomToolBarArea 停靠在底部
  - Qt::AllToolBarAreas 以上四个位置都可停靠

  使用 setAllowedAreas（）函数指定停靠区域：

setAllowedAreas（Qt::LeftToolBarArea | Qt::RightToolBarArea）

使用 setFloatable（trueOrFalse）函数来设定工具栏可否浮动

使用 setMoveable（trueOrFalse）函数设定工具栏的可移动性：

setMoveable（false）//工具条不可移动，只能停靠在初始化的位置上

5.3 状态栏

一个 QMainWindow 的程序最多只有一个状态栏。QMainWindow 中可以有多个的部件都使用 add...名字的函数，而只有一个的部件，就直接使用获取部件的函数，如 menuBar。同理状态栏也提供了一个获取状态栏的函数 statusBar()，没有就自动创建一个并返回状态栏的指针。

```
QMenuBar *    menuBar() const;
```

- 添加小部件（从状态栏左侧添加）

```
void addWidget(QWidget * widget, int stretch = 0);
//插入小部件
int insertWidget(int index, QWidget * widget, int stretch = 0);
//删除小部件
void removeWidget(QWidget * widget);
```

- 添加小部件（从状态栏右侧添加）

```
void addPermenentWidget (QWidget *widget, int stretch = 0);
```

5.4 停靠部件（也称为铆接部件、浮动窗口）

停靠部件 QDockWidget，也称浮动窗口，可以有多个。

```
QDockWidget * dock = new QDockWidget("标题",this);
//添加停靠部件到 mainWindow 中，并且设定默认停靠在左边
addDockWidget(Qt::LeftDockWidgetArea,dock);
//设定停靠部件允许停靠的范围
dock->setAllowedAreas(Qt::LeftDockWidgetArea |
      Qt::RightDockWidgetArea | Qt::TopDockWidgetArea);
```

5.5 核心部件（中心部件）

除了以上几个部件，中心显示的部件都可以作为核心部件，例如一个记事本程序中，就是一个 QTextEdit（编辑框控件）做核心部件

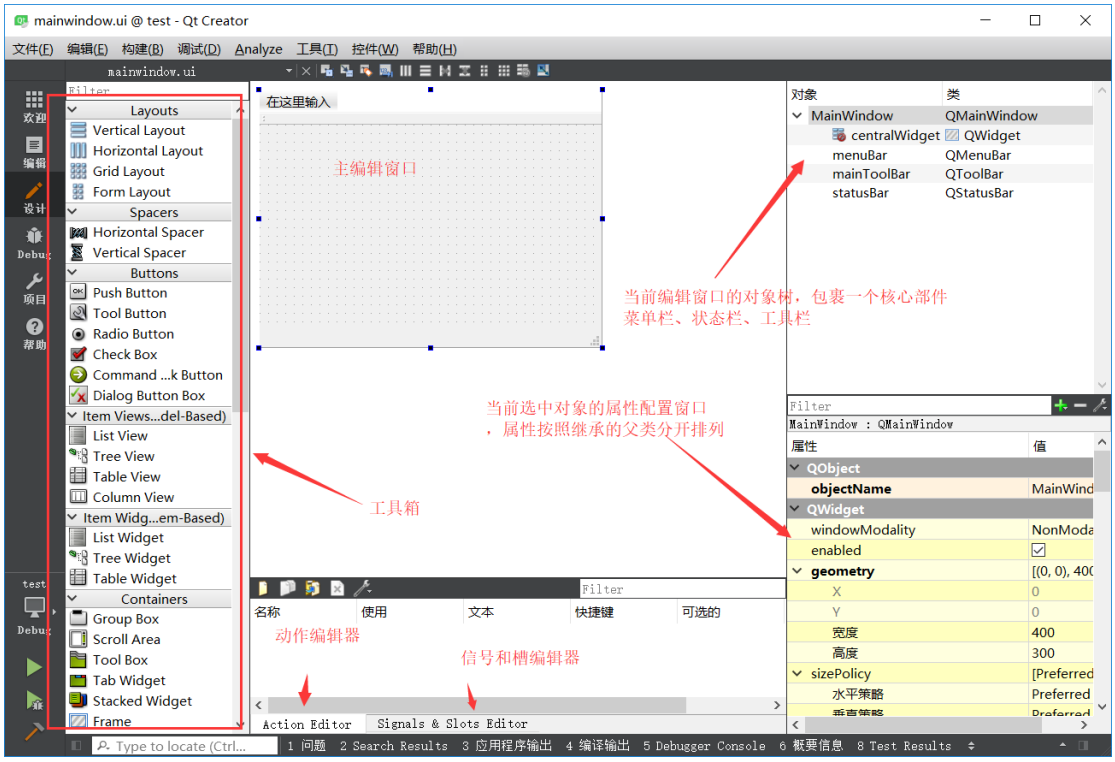
```
QTextEdit * edit = new QTextEdit(this);
```

```
//设置 mainWindow 的核心部件
setCentralWidget(edit);
```

5.6 使用 UI 文件创建窗口

创建工程的时候把 UI 文件留着

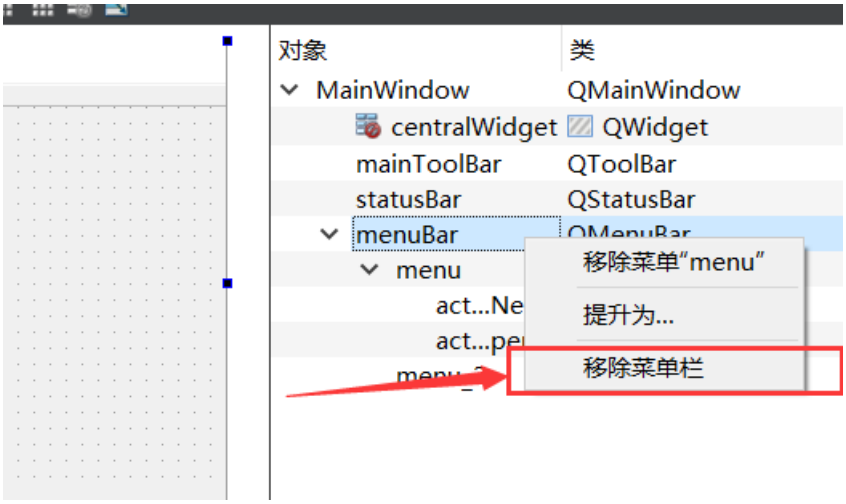
5.6.1 UI 设计窗口介绍



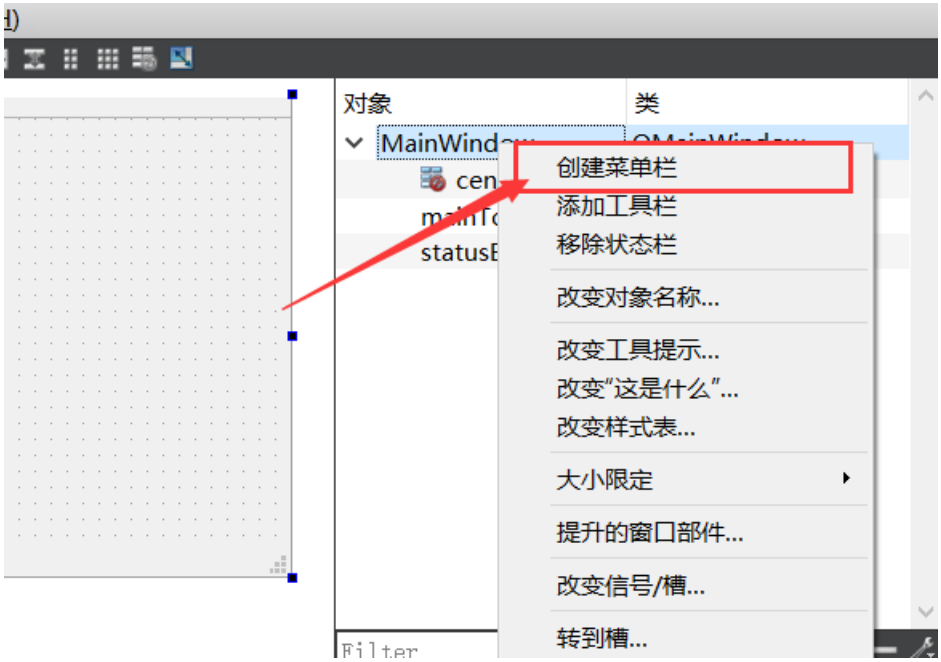
5.6.2 菜单栏

5.6.2.1 添加/删除菜单栏

默认情况下 QMainWindow 项目一创建就自带了菜单栏，可以在对象树窗口中，右键菜单栏对象，移除菜单栏：

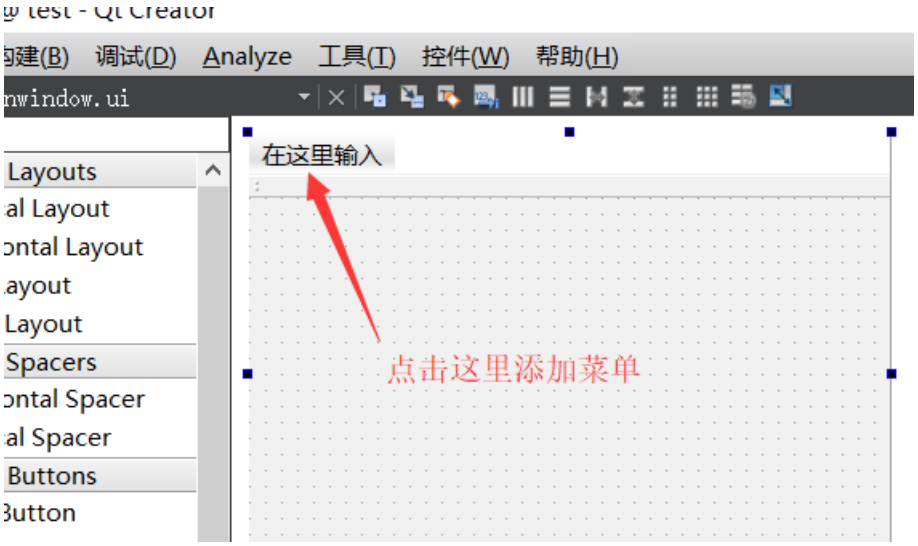


删除后也可以创建菜单栏，此时在对象树中右键 MainWindow 对象，菜单里边会多了创建菜单栏的功能

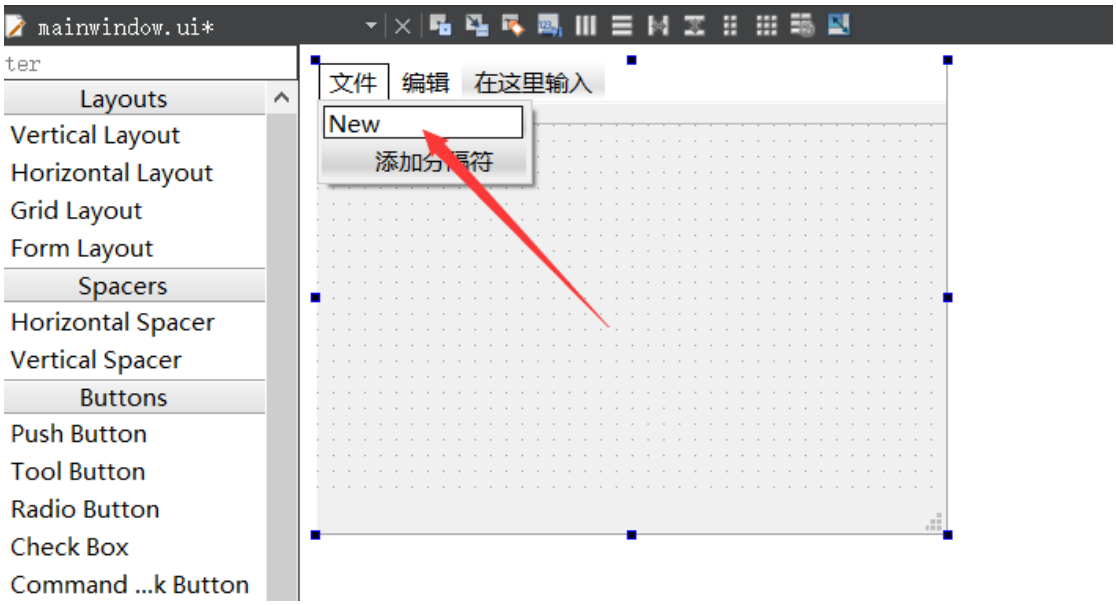


5.6.2.2 添加菜单

点击菜单栏的“在这里输入”可以输入一个菜单名字创建一个菜单。

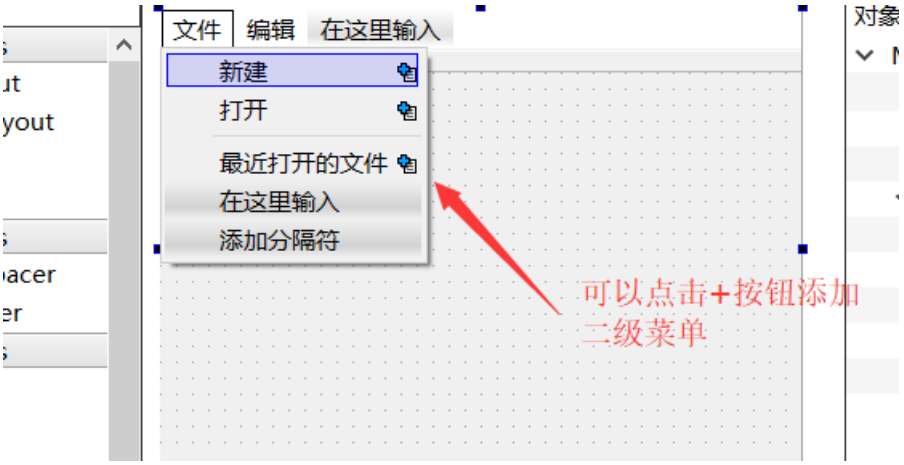


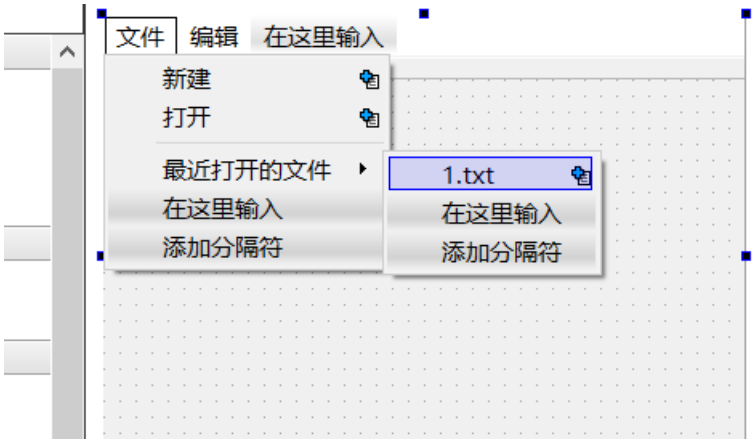
5.6.2.3 添加菜单项



在 UI 界面中添加菜单项只能用英文，因为此时会创建一个 QAction 对象，会用输入的值作为对象名，所以不能用中文，得添加后再属性窗口改中文。

5.6.2.4 添加多级菜单





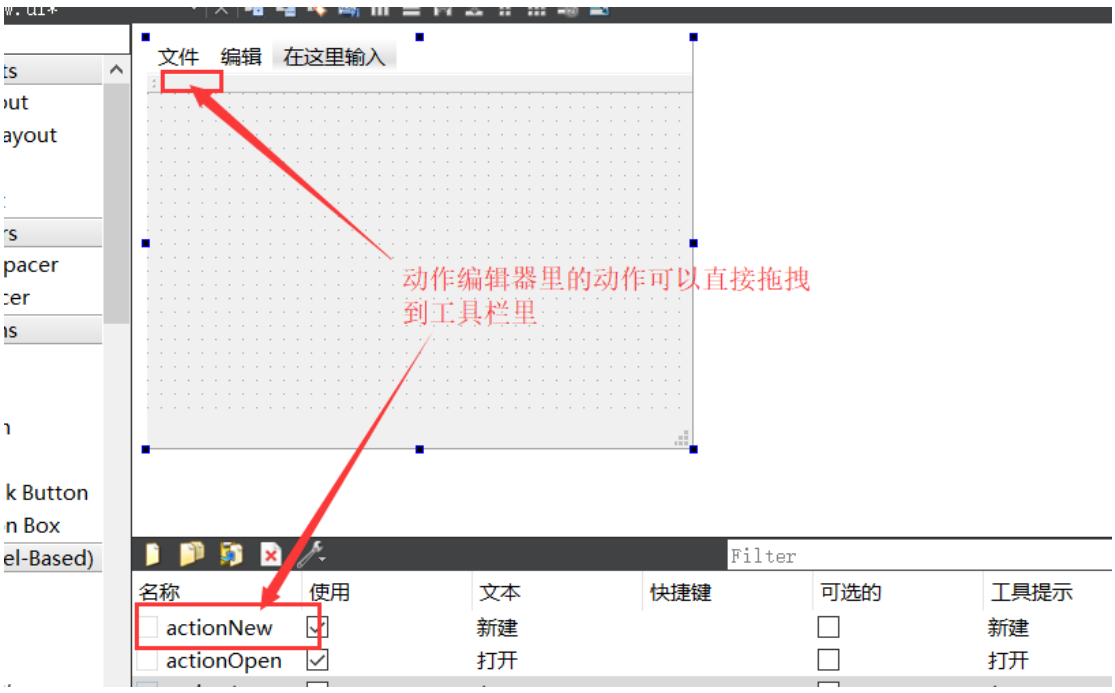
5.6.3 工具栏

5.6.3.1 添加/删除工具栏

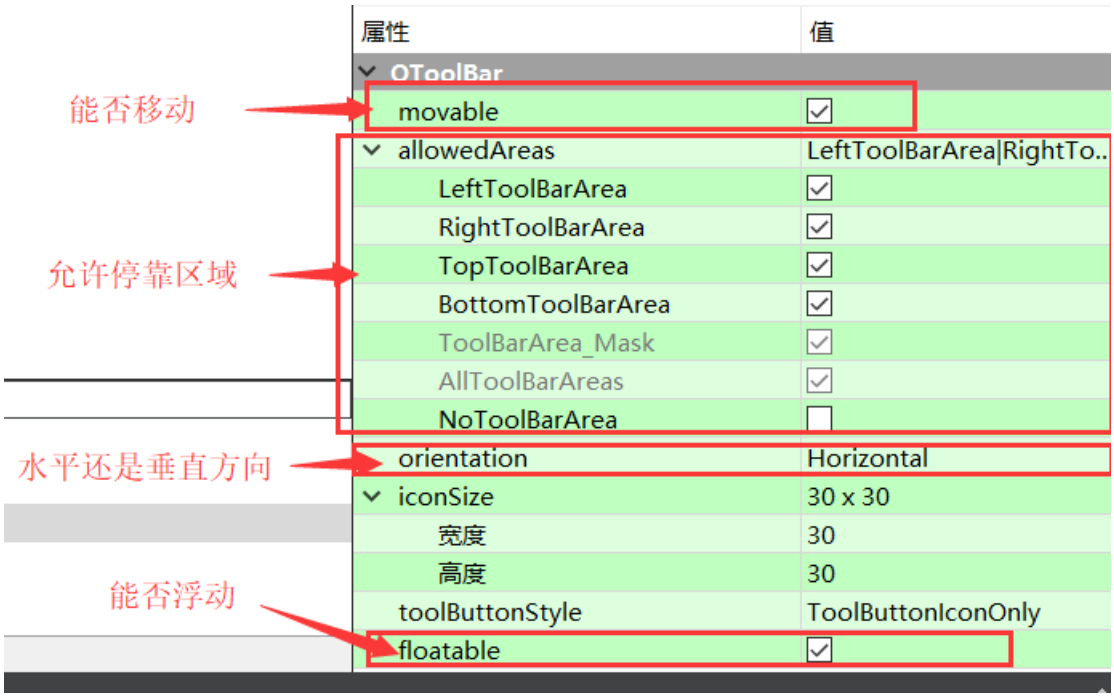
删除工具栏方法和删除菜单栏方法一样，不过工具栏可以有多个，所以每次右键 MainWindow 对象，都可以看到添加工具栏的选项。

5.6.3.2 工具栏添加动作

新添加的 QAction 对象会在动作编辑器里找到（Action Editor），可以直接拖拽上来添加到工具栏里。



可以对工具栏设定停靠区域、能否浮动、能否移动等



5.6.4 状态栏

添加和删除状态栏的方法和添加删除菜单栏方法一样。

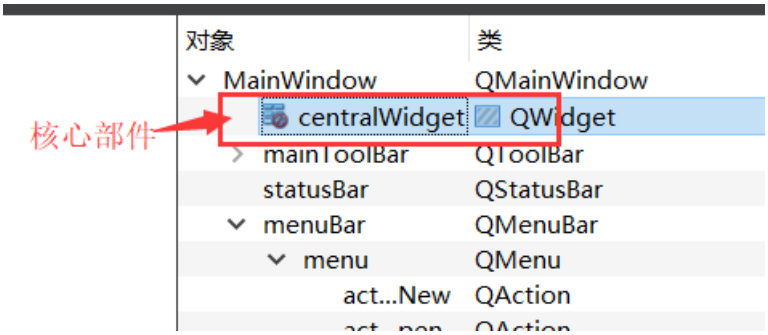
状态栏添加左侧控件、右侧控件只能通过代码来添加。

5.6.5 停靠部件

从工具箱中拖出一个停靠部件就行。也可以像设定工具栏停靠范围一样，在停靠部件的属性窗口中设定他可以停靠的范围。

5.6.6 核心部件

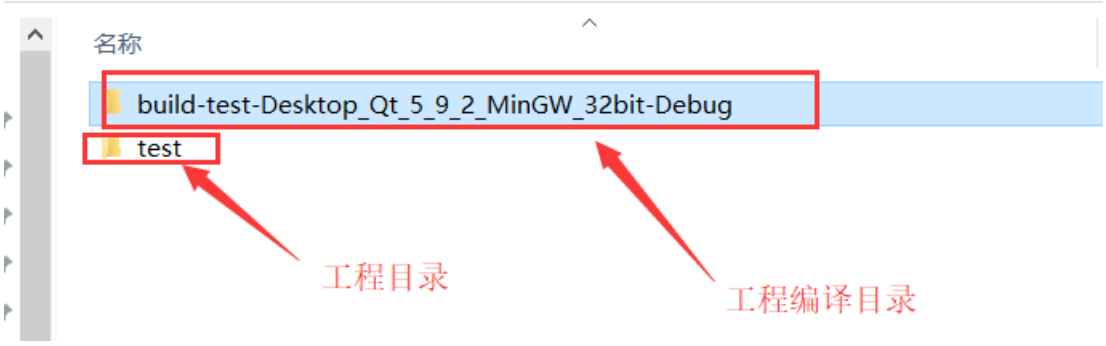
UI 窗口中默认核心部件就是一个 widget

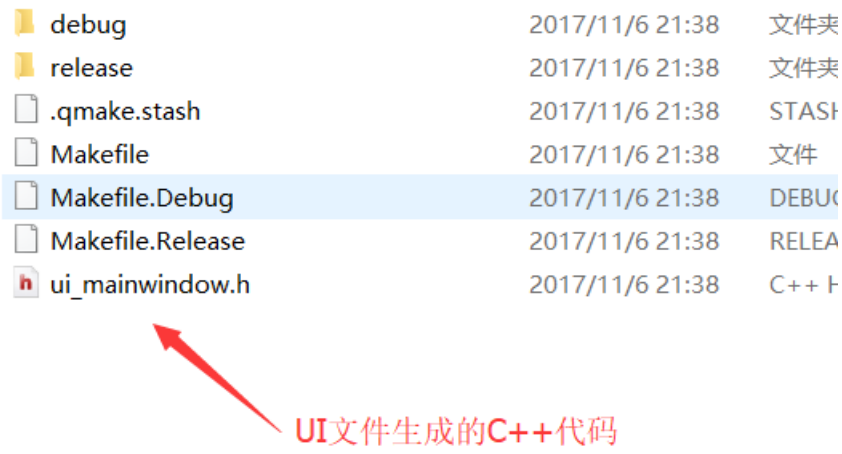


5.7 UI 文件原理

使用 UI 文件创建界面很轻松很便捷，他的原理就是每次我们保存 UI 文件的时候，QtCreator 就自动帮我们将 UI 文件翻译成 C++的图形界面创建代码。可以通过以下步骤查看代码

到工程编译目录，一般就是工程同级目录下会生成另一个编译目录，会找到一个带 ui_前缀跟 ui 文件同名的.h 文件，这就是代码





代码内容：

```
class Ui_MainWindow
{
public:
    QAction *actionNew;
    QAction *actionOpen;
    QAction *action1_txt;
    QWidget *centralWidget;
    QToolBar *mainToolBar;
    QStatusBar *statusBar;
    QMenuBar *menuBar;
    QMenu *menu_2;
    QMenu *menu;
    QMenu *menu_3;

    void setupUi(QMainWindow *MainWindow)
    {
        if (MainWindow->objectName().isEmpty())
            MainWindow->setObjectName(QStringLiteral("MainWindow"));
        MainWindow->resize(400, 300);
        actionNew = new QAction(MainWindow);
        actionNew->setObjectName(QStringLiteral("actionNew"));
        actionOpen = new QAction(MainWindow);
        actionOpen->setObjectName(QStringLiteral("actionOpen"));
        action1_txt = new QAction(MainWindow);
        action1_txt->setObjectName(QStringLiteral("action1_txt"));
        centralWidget = new QWidget(MainWindow);
        centralWidget->setObjectName(QStringLiteral("centralWidget"));
        MainWindow->setCentralWidget(centralWidget);
        mainToolBar = new QToolBar(MainWindow);
        mainToolBar->setObjectName(QStringLiteral("mainToolBar"));
        MainWindow->addToolBar(Qt::TopToolBarArea, mainToolBar);
```

MainWindow构造函数中会调用该函数来构建窗口的初始化界面

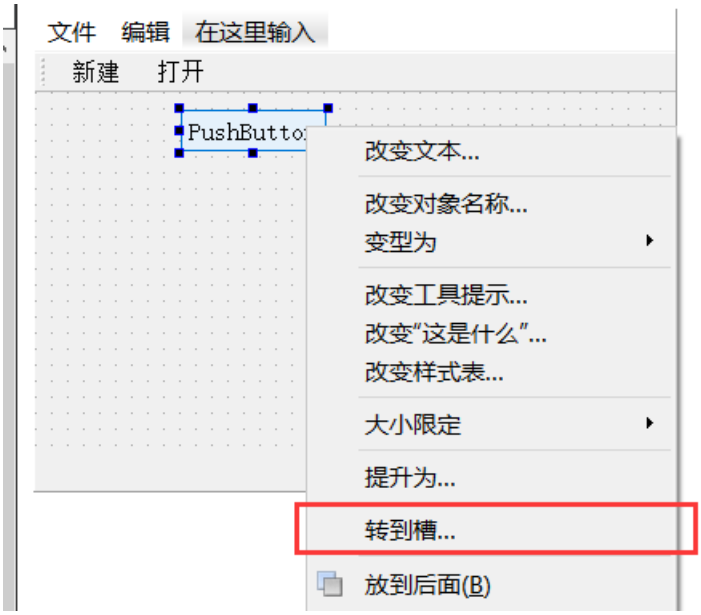
在项目 MainWindow 的构造函数中会调用这个函数来初始化窗口，其实这里边就是初始化我们的各个控件。

```
MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    //如果想要使用 ui 里边的控件对象
    //代码必须写在 setupUi 之下
    //否则 ui 各个控件没有初始化时使用会出问题
    ui->pushButton->setText("Hello");
}
```

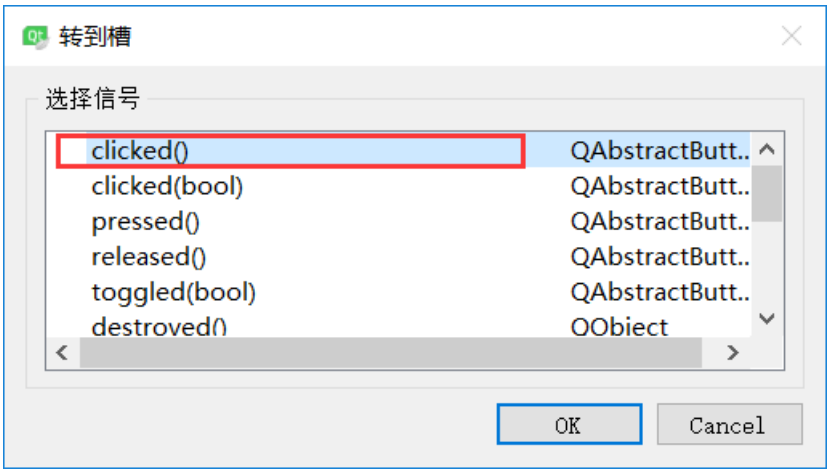
5.8 UI 文件下使用信号和槽

5.8.1 转到槽

在 UI 编辑界面中使用信号和槽很方便，比如，拖出一个 Button 到窗口上，右键这个 button，选择转到槽：



此时会出现这个控件（QPushButton）可以连接的各个信号，我们可以根据具体需求选中信号来创建一个连接这个信号的槽函数：



以 click(bool)信号为例，创建了一个槽函数

```
19  }
20
21  void MainWindow::on_pushButton_clicked(bool checked)
22  {
23
24  }
25
```

这个槽函数是 QtCreator 自动帮我们创建的，而且也使用生成 C++代码的方式帮我们做好了连接，我们可以直接在这个函数体内实现功能就行。很方便，比使用 Lambda 表达式还方便。

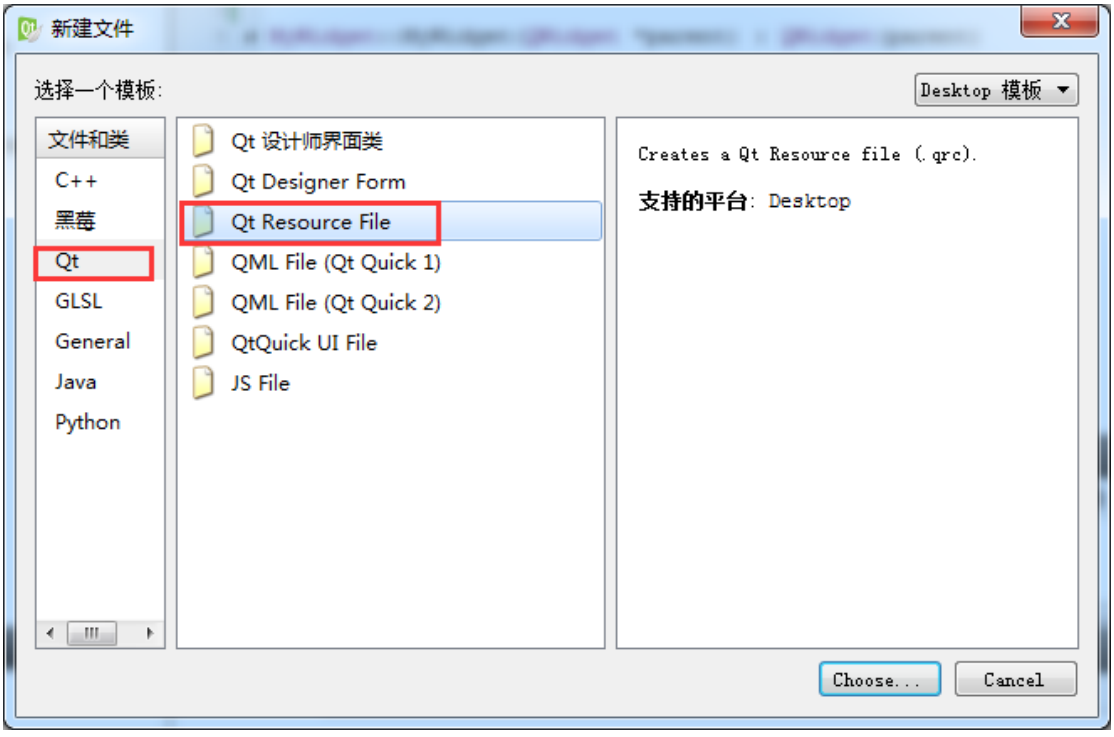
5.8.2 信号槽编辑器

可以使用动作编辑器旁边的信号槽编辑器，里边也可以添加信号和槽的连接，比如添加 actionQuit 的 triggered 信号和窗口 close 槽的连接：

5.9 资源文件

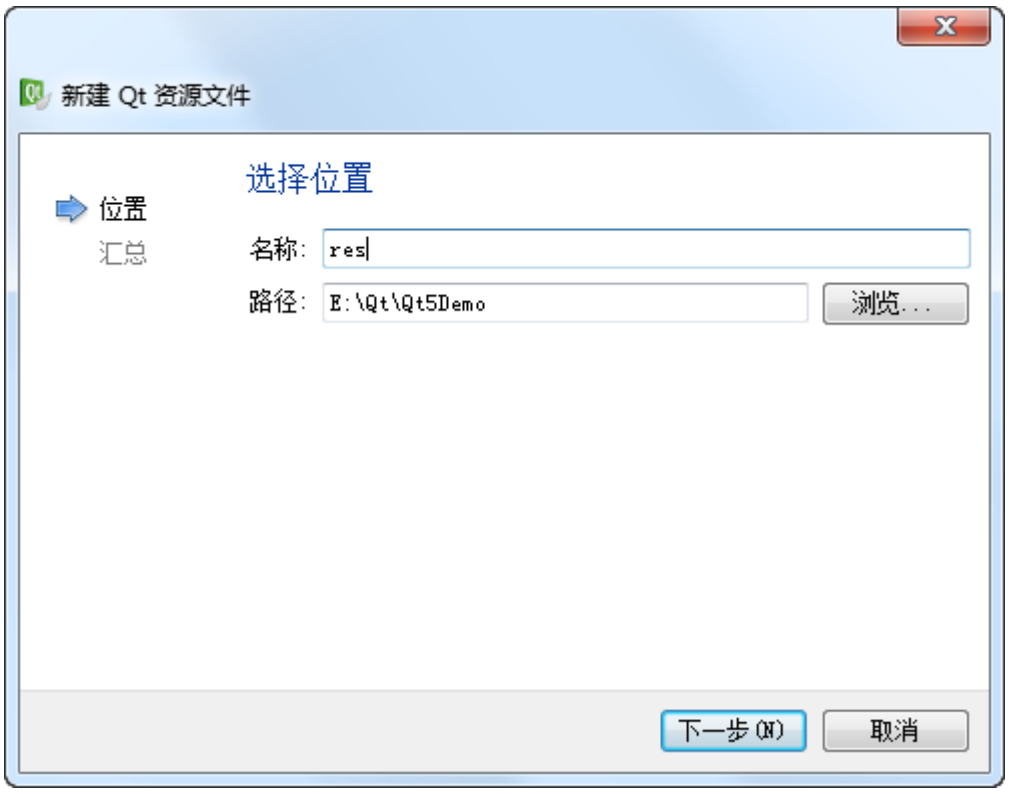
Qt 资源系统是一个跨平台的资源机制，用于将程序运行时所需要的资源以二进制的形式存储于可执行文件内部。如果你的程序需要加载特定的资源（图标、文本翻译等），那么，将其放置在资源文件中，就再也不需要担心这些文件的丢失。也就是说，如果你将资源以资源文件形式存储，它是会编译到可执行文件内部。

使用 Qt Creator 可以很方便地创建资源文件。我们可以在工程上点右键，选择“添加新文件…”，可以在 Qt 分类下找到“Qt 资源文件”：

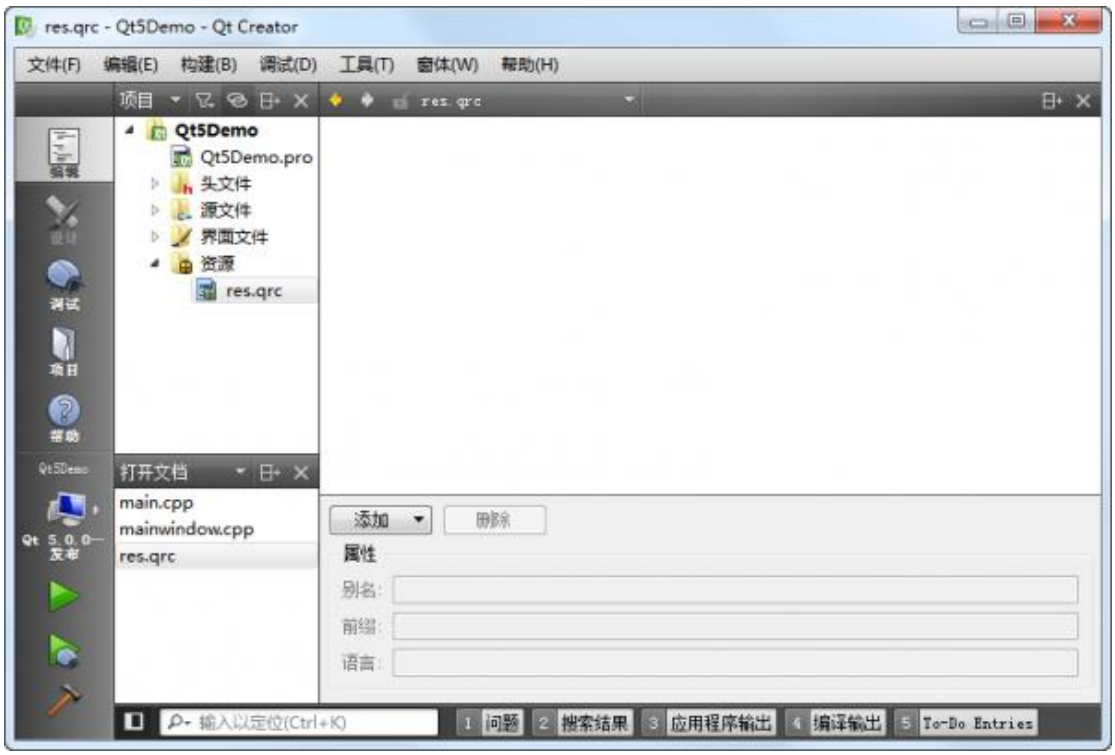


点击“选择…”按钮，打开“新建 Qt 资源文件”对话框。在这里我们输入

资源文件的名字和路径：



点击下一步，选择所需要的版本控制系统，然后直接选择完成。我们可以在 Qt Creator 的左侧文件列表中看到“资源文件”一项，也就是我们新创建的资源文件：

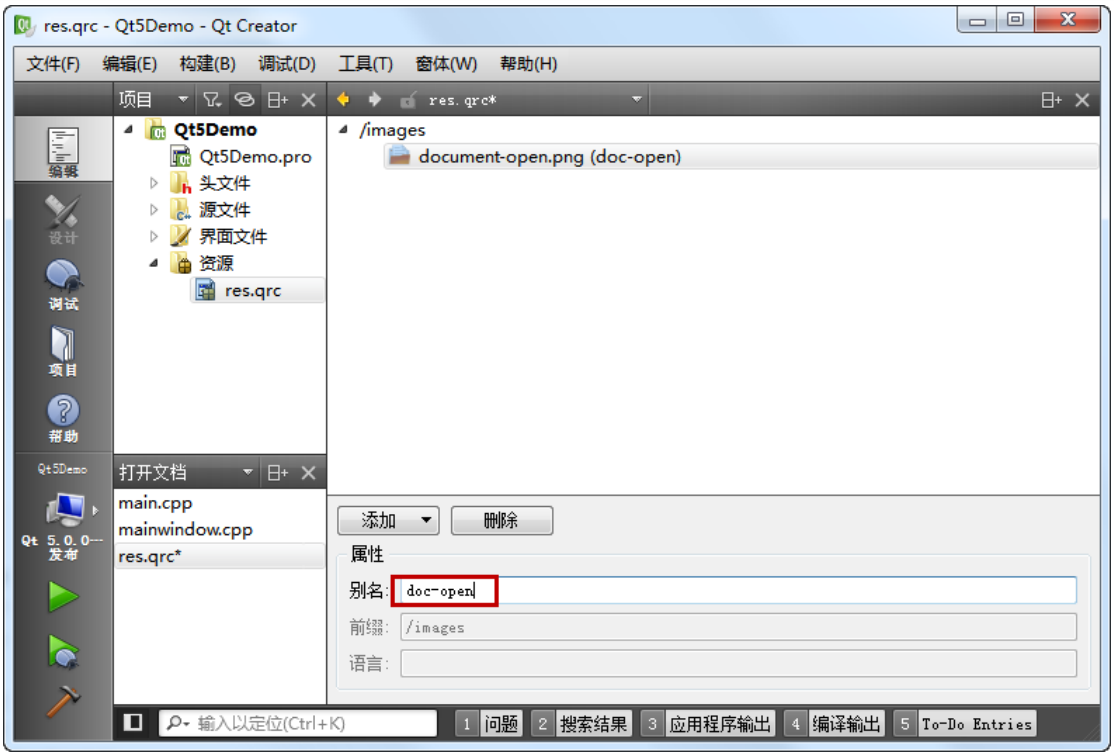


右侧的编辑区有个“添加”，我们首先需要添加前缀，比如我们将前缀取名

为 images。然后选中这个前缀，继续点击添加文件，可以找到我们所需添加的文件。这里，我们选择 document-open.png 文件。当我们完成操作之后，Qt Creator 应该是这样子的：

接下来，我们还可以添加另外的前缀或者另外的文件。这取决于你的需要。当我们添加完成之后，我们可以像前面一章讲解的那样，通过使用 : 开头的路径来找到这个文件。比如，我们的前缀是 /images，文件是 document-open.png，那么就可以使用:/images/document-open.png 找到这个文件。

这么做带来的一个问题是，如果以后我们要更改文件名，比如将 docuemnt-open.png 改成 docopen.png，那么，所有使用了这个名字的路径都需要修改。所以，更好的办法是，我们给这个文件去一个“别名”，以后就以这个别名来引用这个文件。具体做法是，选中这个文件，添加别名信息：

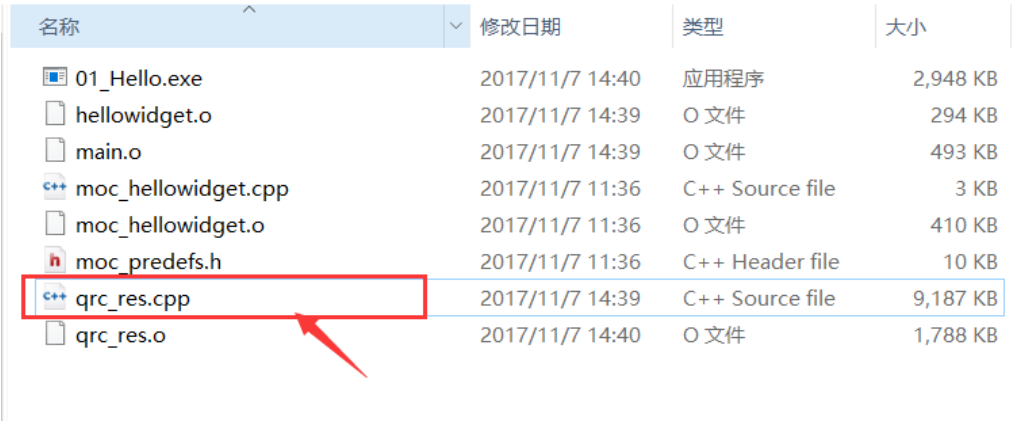


这样，我们可以直接使用:/images/doc-open 引用到这个资源，无需关心图片的真实文件名。

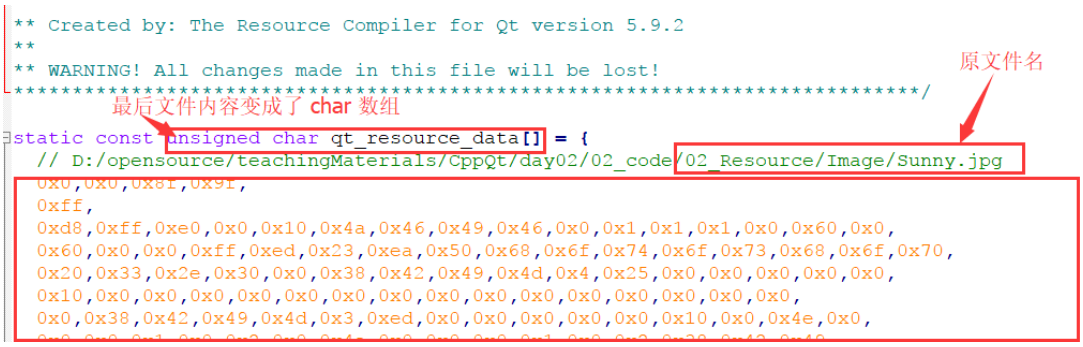
Qrc 文件只是记录了我们要用到的资源文件在项目路径哪个位置的一个信息，如果我们使用文本编辑器打开 res.qrc 文件，就会看到以下内容：

```
<RCC>
    <qresource prefix="/images">
        <file alias="doc-open">document-open.png</file>
    </qresource>
    <qresource prefix="/images/fr" lang="fr">
        <file alias="doc-open">document-open-fr.png</file>
    </qresource>
</RCC>
```

当我们编译工程之后，我们可以在构建目录中找到 qrc_res.cpp 文件。



这就是 Qt 将我们的资源编译成了 C++ 代码：



可以看出 Qt 帮我们将资源文件内容一个字节一个字节的读出来最终放到了代码里，使用字符数组的形式保存着，所以程序启动的时候这些资源就会以数组的形式占用到程序内存里。当我们使用 qt 的 qrc 资源文件时要考虑内存占用的问题，如果 Qt 程序资源很多，而且并不是每次运行程序都会加载所有的资源，比如制作一个游戏，所需要的图片声音资源量很大，可能超过了机器内存大小，这时候使用 qrc 加载资源的方式很不合适。

可以考虑动态加载资源的方式，当切入到某个游戏场景的时候才加载场景相关的图片声音资源。Qt 有提供 rcc 的方式动态加载资源，不过不常用，所以一般游戏资源都放在可执行文件 exe 所在目录或者子目录下，程序运行的时候就从游戏 exe 文件路径去搜寻资源。

6 QDialog 对话框

6.1 基本概念

对话框是 GUI 程序中不可或缺的组成部分。很多不能或者不适合放入主窗口的功能组件都必须放在对话框中，比如用于完成一次性任务的功能（如登录功能、选择某个文件打开、保存文件）。对话框通常会是一个顶层窗口，出现在程序最上层，用于实现短期任务或者简洁的用户交互。

Qt 中使用 QDialog 类实现对话框，但是声明一个 QDilaog 对象的时候，不管这个对话框对象跟哪个窗口建立了父子关系，当他显示出来的时候都还是一个顶层的窗口。

对话框分为模态对话框和非模态对话框。

- 模态对话框，当对话框打开时，不能操作同一个应用程序的其他窗口，只有当对话框关闭的时候才可以操作。

  模态对话框很常见，比如“打开文件”功能。你可以尝试一下记事本的打开文件，当打开文件对话框出现时，我们是不能对除此对话框之外的窗口部分进行操作的。

- 与此相反的是非模态对话框，例如查找对话框，我们可以在显示着查找对话框的同时，继续对记事本的内容进行编辑。

6.1.1 模态对话框

Qt 有两种级别的模态对话框：

- 应用程序级别的模态

  当该种模态的对话框出现时，用户必须首先对对话框进行交互，直到关闭对话框，然后才能访问程序中其他的窗口。

  使用 QDialog::exec()实现应用程序级别的模态对话框

- 窗口级别的模态

  该模态仅仅阻塞与对话框关联的窗口，但是依然允许用户与程序中其它窗口交互。窗口级别的模态尤其适用于多窗口模式。

  使用 QDialog::open()实现窗口级别的模态对话框

  一般情况下我们只使用应用程序级别的模态对话框。

在下面的示例中，我们调用了 exec()将对话框显示出来，因此这就是一个模态对话框。当对话框出现时，我们不能与主窗口进行任何交互，直到我们关闭了该对话框。

```
QDialog dialog;
dialog.setWindowTitle(tr("Hello, dialog!"));
```

6.1.2 非模态对话框

下面我们试着将 exec()修改为 show()，看看非模态对话框：

```
QDialog dialog(this);
dialog.setWindowTitle(tr("Hello, dialog!"));
dialog.show();
```

是不是事与愿违？对话框竟然一闪而过！这是因为，**show()函数不会阻塞当前线程，对话框会显示出来，然后函数立即返回，代码继续执行。**注意，dialog 是建立在栈上的，show()函数返回，MainWindow::open()函数结束，dialog 超出作用域被析构，因此对话框消失了。知道了原因就好改了，我们将 dialog 改成堆上建立，当然就没有这个问题了：

```
QDialog *dialog = new QDialog;
dialog->setWindowTitle(tr("Hello, dialog!"));
dialog->show();
```

如果你足够细心，应该发现上面的代码是有问题的：dialog 存在内存泄露！dialog 使用 new 在堆上分配空间，却一直没有 delete。解决方案也很简单：将 MainWindow 的指针赋给 dialog 即可。还记得我们前面说过的 Qt 的对象树吗？

不过，这样做有一个问题：如果我们的对话框不是在一个界面类中出现呢？由于 QWidget 的 parent 必须是 QWidget 指针，那就限制了我们不能将一个普通的 C++ 类指针传给 Qt 对话框。另外，如果对内存占用有严格限制的话，当我们将主窗口作为 parent 时，主窗口不关闭，对话框就不会被销毁，所以会一直占用内存。在这种情景下，我们可以设置 dialog 的 WindowAttribute：

```
QDialog *dialog = new QDialog;
dialog->setAttribute(Qt::WA_DeleteOnClose);
dialog->setWindowTitle(tr("Hello, dialog!"));
dialog->show();
```

setAttribute()函数设置对话框关闭时，自动销毁对话框。

6.2 标准对话框

所谓标准对话框，是 Qt 内置的一系列对话框，用于简化开发。事实上，有很多对话框都是通用的，比如打开文件、设置颜色、打印设置等。这些对话框在所有程序中几乎相同，因此没有必要在每一个程序中都自己实现这么一个对话框。

Qt 的内置对话框大致分为以下几类：

- QMessageBox：　　模态对话框，用于显示信息、询问问题等；
- QColorDialog：　　选择颜色；

- QFontDialog：选择字体；
- QFileDialog：选择文件或者目录；
- QInputDialog：允许用户输入一个值，并将其值返回；
- QPageSetupDialog：为打印机提供纸张相关的选项；
- QPrintDialog：打印机配置；
- QPrintPreviewDialog：打印预览；
- QProgressDialog：显示操作过程。

6.3 消息对话框

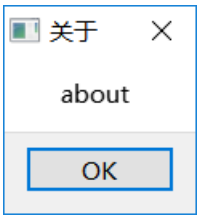
QMessageBox 用于显示消息提示。我们一般会使用其提供的几个 static 函数：

- About

显示关于对话框。

```
void about(QWidget * parent, const QString & title, const QString & text)
```

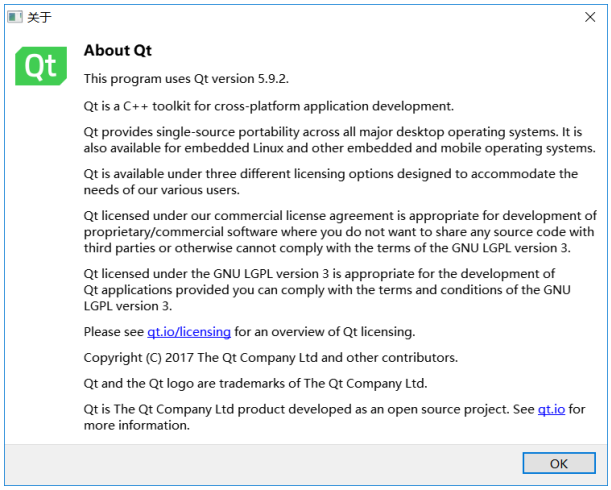
这是一个最简单的对话框，其标题是 title，内容是 text，父窗口是 parent。对话框只有一个 OK 按钮。



- AboutQt

显示关于 Qt 对话框。该对话框用于显示有关 Qt 的信息。

```
void aboutQt(QWidget * parent, const QString & title = QString());
```

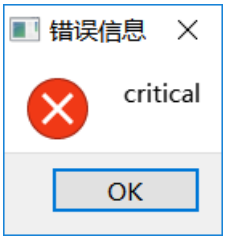


- Critical

显示严重错误对话框。

```
StandardButton critical(QWidget * parent,
const QString & title,
const QString & text,
StandardButtons buttons = Ok,
StandardButton defaultButton = NoButton);
```

这个对话框将显示一个红色的错误符号。我们可以通过 buttons 参数指明其显示的按钮。默认情况下只有一个 Ok 按钮，我们可以使用 StandardButtons 类型指定多种按钮。

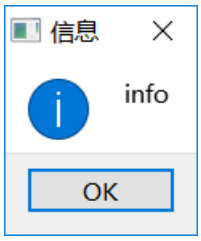


- Infomation

与 critical 类似，不同之处在于这个对话框提供一个普通信息图标。

```
StandardButton information(QWidget * parent,
const QString & title,
const QString & text,
```

```
StandardButtons buttons = Ok,
StandardButton defaultButton = NoButton)
```



- Question

与 QMessageBox::critical ()类似，不同之处在于这个对话框提供一个问号图标，并且其显示的按钮是“是”和“否”。

```
StandardButton question(QWidget * parent,
const QString & title,
const QString & text,
StandardButtons buttons = StandardButtons( Yes | No ),
StandardButton defaultButton = NoButton)
```

- 第一个参数是对话框的父窗口是 this。

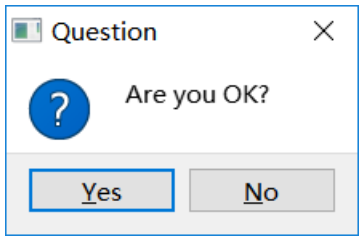
QMessageBox 是 QDialog 的子类，这意味着它的初始显示位置将会是在 parent 窗口的中央。

- 第二个参数是对话框的标题。
- 第三个参数是我们想要显示的内容。
- 第四个参数是关联的按键类型，我们可以使用或运算符（|）指定对话框应该出现的按钮。比如我们希望是一个 Yes 和一个 No。
- 最后一个参数指定默认选择的按钮。

这个函数有一个返回值，用于确定用户点击的是哪一个按钮。按照我们的写法，应该很容易的看出，这是一个模态对话框，因此我们可以直接获取其返回值。

我们可以通过以下代码来判断问题对话框的返回值：

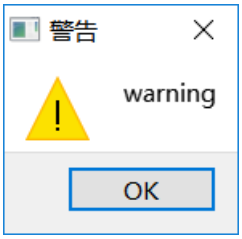
```
if (QMessageBox::Yes == QMessageBox::question(this,
                tr("Question"), tr("Are you OK?"),
                QMessageBox::Yes | QMessageBox::No,
                QMessageBox::Yes))
{
    QMessageBox::information(this, tr("Hmmm..."),
tr("I'm glad to hear that!"));
}
else
{
    QMessageBox::information(this, tr("Hmmm..."),
tr("I'm sorry!"));
}
```



- warning

与 QMessageBox::critical()类似，不同之处在于这个对话框提供一个黄色叹号图标。

```
StandardButton warning(QWidget * parent,
const QString & title,
const QString & text,
StandardButtons buttons = Ok,
StandardButton defaultButton = NoButton)
```



QMessageBox 类的 static 函数优点是方便使用，缺点也很明显：非常不

灵活。我们只能使用简单的几种形式。为了能够定制 QMessageBox 细节，我们必须使用 QMessageBox 的属性设置 API。如果我们希望制作一个询问是否保存的对话框，我们可以使用如下的代码：

```
QMessageBox msgBox;
msgBox.setText(tr("The document has been modified."));
msgBox.setInformativeText(tr("Do you want to save your changes?"));
msgBox.setDetailedText(tr("Differences here..."));
msgBox.setStandardButtons(QMessageBox::Save
                          | QMessageBox::Discard
                          | QMessageBox::Cancel);
msgBox.setDefaultButton(QMessageBox::Save);
int ret = msgBox.exec();
switch (ret)
{
case QMessageBox::Save:
    qDebug() << "Save document!";
    break;
case QMessageBox::Discard:
    qDebug() << "Discard changes!";
    break;
case QMessageBox::Cancel:
    qDebug() << "Close document!";
    break;
}
```

msgBox 是一个建立在栈上的 QMessageBox 实例。我们设置其主要文本信息为“The document has been modified.”，informativeText 则是会在对话框中显示的简单说明文字。下面我们使用了一个 detailedText，也就是详细信息，当我们点击了详细信息按钮时，对话框可以自动显示更多信息。我们自己定义的对话框的按钮有三个：保存、丢弃和取消。然后我们使用了 exec()是其成为一个模态对话框，根据其返回值进行相应的操作。

6.4 标准文件对话框

QFileDialog，也就是文件对话框。在本节中，我们将尝试编写一个简单的文本文件编辑器，我们将使用 QFileDialog 来打开一个文本文件，并将修改过的文件保存到硬盘。

首先，我们需要创建一个带有文本编辑功能的窗口。借用我们前面的程序代码，应该可以很方便地完成：

```
openAction = new QAction(QIcon(":/images/file-open"),tr("&Open..."),
this);
openAction->setStatusTip(tr("Open an existing file"));
saveAction = new QAction(QIcon(":/images/file-save"), tr("&Save..."),
this);
saveAction->setStatusTip(tr("Save a new file"));
QMenu *file = menuBar()->addMenu(tr("&File"));
file->addAction(openAction);
file->addAction(saveAction);

QToolBar *toolBar = addToolBar(tr("&File"));
toolBar->addAction(openAction);
toolBar->addAction(saveAction);

textEdit = new QTextEdit(this);
setCentralWidget(textEdit);
```

我们在菜单和工具栏添加了两个动作:打开和保存。接下来是一个 QTextEdit 类，这个类用于显示富文本文件。也就是说，它不仅仅用于显示文本，还可以显示图片、表格等等。不过，我们现在只用它显示纯文本文件。QMainWindow 有一个 setCentralWidget()函数，可以将一个组件作为窗口的中心组件，放在窗口中央显示区。显然，在一个文本编辑器中，文本编辑区就是这个中心组件，因此我们将 QTextEdit 作为这种组件。

我们使用 connect()函数，为这两个 QAction 对象添加响应的动作：

```
    connect(openAction, &QAction::triggered,
    this, &MainWindow::openFile);
    connect(saveAction, &QAction::triggered,
    this, &MainWindow::saveFile);
```

下面是最主要的 openFile()和 saveFile()这两个函数的代码：

```
//打开文件
void MainWindow::openFile()
{
```

```
    QString path = QFileDialog::getOpenFileName(this,
                                                 tr("Open File"), ".",
tr("Text Files(*.txt)"));
    if(!path.isEmpty())
    {
        QFile file(path);
        if (!file.open(QIODevice::ReadOnly | QIODevice::Text))
        {
            QMessageBox::warning(this, tr("Read File"),
                                 tr("Cannot open
file:\n%1").arg(path));
            return;
        }
        QTextStream in(&file);
        textEdit->setText(in.readAll());
        file.close();
    }
    else
    {
        QMessageBox::warning(this, tr("Path"),
                             tr("You did not select any file."));
    }
}

//保存文件
void MainWindow::saveFile()
{
    QString path = QFileDialog::getSaveFileName(this,
                                                 tr("Open File"), ".",
tr("Text Files(*.txt)"));
    if(!path.isEmpty())
    {
        QFile file(path);
        if (!file.open(QIODevice::WriteOnly | QIODevice::Text))
        {
            QMessageBox::warning(this, tr("Write File"),
                                 tr("Cannot open
file:\n%1").arg(path));
            return;
        }
        QTextStream out(&file);
        out << textEdit->toPlainText();
        file.close();
    }
```

```
    else
    {
        QMessageBox::warning(this, tr("Path"),
                             tr("You did not select any file."));
    }
}
```

在 openFile()函数中，我们使用 QFileDialog::getOpenFileName()来获取需要打开的文件的路径。这个函数原型如下：

```
QString getOpenFileName(QWidget * parent = 0,
                        const QString & caption = QString(),
                        const QString & dir = QString(),
                        const QString & filter = QString(),
                        QString * selectedFilter = 0,
                        Options options = 0);
```

不过注意，它的所有参数都是可选的，因此在一定程度上说，这个函数也是简单的。这六个参数分别是：

- parent：父窗口。
- caption：对话框标题；
- dir：对话框打开时的默认目录
  - “.” 代表程序运行目录
  - “/” 代表当前盘符的根目录（特指 Windows 平台；Linux 平台当然就是根目录），这个参数也可以是平台相关的，比如“C:\\”等；
- filter：过滤器。

  我们使用文件对话框可以浏览很多类型的文件，但是，很多时候我们仅希望打开特定类型的文件。比如，文本编辑器希望打开文本文件，图片浏览器希望打开图片文件。**过滤器就是用于过滤特定的后缀名**。如果我们使用“Image Files(*.jpg *.png)”，则只能显示后缀名是 jpg 或者 png 的文件。**如果需要多个过滤器，使用“;;”分割**，比如“JPEG Files(*.jpg);;PNG Files(*.png)”；
- selectedFilter：默认选择的过滤器；
- options：对话框的一些参数设定

比如只显示文件夹等等，它的取值是 enum QFileDialog::Option，每个选项可以使用 | 运算组合起来。

QFileDialog::getOpenFileName()返回值是选择的文件路径。我们将其赋值给 path。通过判断 path 是否为空，可以确定用户是否选择了某一文件。只有当用户选择了一个文件时，我们才执行下面的操作。

在 saveFile()中使用的 QFileDialog::getSaveFileName()也是类似的。使用这种静态函数，在 Windows、Mac OS 上面都是直接调用本地对话框，但是 Linux 上则是 QFileDialog 自己的模拟。这暗示了，如果你不使用这些静态函数，而是直接使用 QFileDialog 进行设置，那么得到的对话框很可能与系统对话框的外观不一致。这一点是需要注意的。

7 布局

所谓 GUI 界面，归根结底，就是一堆组件的叠加。我们创建一个窗口，把按钮放上面，把图标放上面，这样就成了一个界面。在放置时，组件的位置尤其重要。我们必须要指定组件放在哪里，以便窗口能够按照我们需要的方式进行渲染。这就涉及到组件布局定位的机制。

Qt 提供了两种组件布局定位机制：静态布局和动态布局。

- **静态布局**就是一种最原始的定位方法：给出这个组件的坐标和长宽值。

  这样，Qt 就知道该把组件放在哪里以及如何设置组件的大小。但是这样做带来的一个问题是，如果用户改变了窗口大小，比如点击最大化按钮或者使用鼠标拖动窗口边缘，采用静态布局的组件是不会有任何响应的。这也很自然，因为你并没有告诉 Qt，在窗口变化时，组件是否要更新自己以及如何更新。或者，还有更简单的方法：禁止用户改变窗口大小。但这总不是长远之计。

- **动态布局**：你只要把组件放入某一种布局(layout)，布局由专门的布局管理器进行管理。当需要调整大小或者位置的时候，Qt 使用对应的布局管理器进行调整。

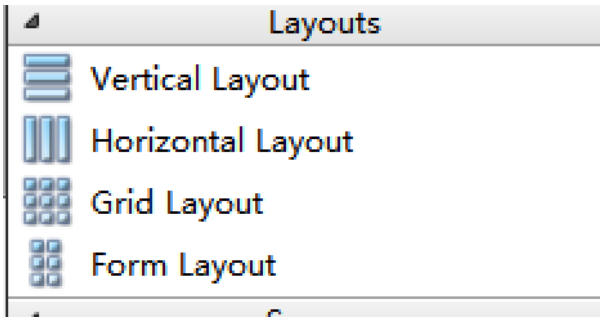
  动态布局解决了使用**静态布局**的缺陷。

  Qt 提供的动态布局中以下三种是我们最常用的：

- QHBoxLayout：按照水平方向从左到右布局；
- QVBoxLayout：按照竖直方向从上到下布局；

- QGridLayout：在一个网格中进行布局，类似于 HTML 的 table；

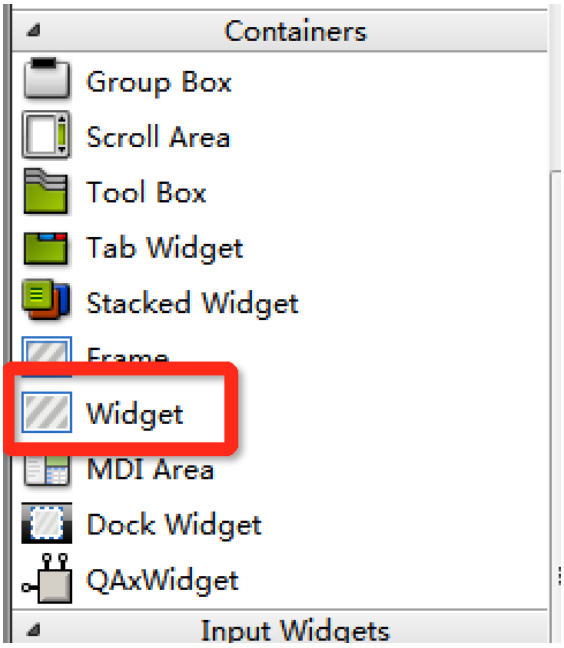
7.1 系统提供的布局控件



这 4 个为系统给我们提供的布局的控件，但是使用起来不是非常的灵活，这里就不详细介绍了。

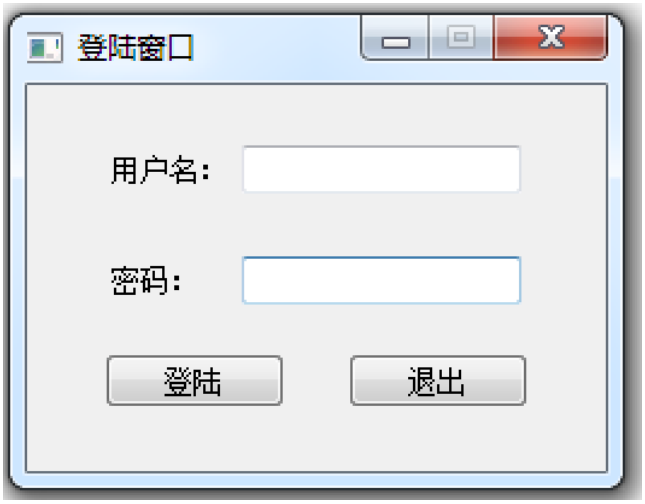
7.2 利用 widget 做布局

第二种布局方式是利用控件里的 widget 来做布局，在 Containers 中



在 widget 中的控件可以进行水平、垂直、栅格布局等操作，比较灵活。

再布局的同时我们需要灵活运用弹簧的特性让我们的布局更加的美观，下面是一个登陆窗口，利用 widget 可以搭建出如下登陆界面：



8 常用控件

Qt 为我们应用程序界面开发提供的一系列的控件，下面我们介绍两种最常用一些控件，所有控件的使用方法我们都可以通过帮助文档获取。

8.1 QLabel 控件使用

QLabel 是我们最常用的控件之一，其功能很强大，我们可以用来显示文本，图片和动画等。

8.1.1 显示文字 （普通文本、html）

通过 QLabel 类的 setText 函数设置显示的内容:

```
void    setText(const QString &)
```

- 可以显示普通文本字符串

```
QLable *label = new QLable;
label->setText(“Hello, World!”);
```

- 可以显示 HTML 格式的字符串

比如显示一个链接:

```
QLabel * label = new QLabel(this);
label ->setText("Hello, World");
label ->setText("<h1><a href=\"https://www.baidu.com\">百度一下</a></h1>");
label ->setOpenExternalLinks(true);
```

其中 setOpenExternalLinks()函数是用来设置用户点击链接之后是否自动打开链接，如果参数指定为 true 则会自动打开。

8.1.2 显示图片

可以使用 QLabel 的成员函数 setPixmap 设置图片

```
void    setPixmap(const QPixmap &)
```

首先定义 QPixmap 对象

```
QPixmap pixmap;
```

然后加载图片

```
pixmap.load(":/Image/boat.jpg");
```

最后将图片设置到 QLabel 中

```
QLabel *label = new QLabel;
label.setPixmap(pixmap);
```

8.1.3 显示动画

可以使用 QLabel 的成员函数 setMovie 加载动画，可以播放 gif 格式的文件

```
void    setMovie(QMovie * movie)
```

首先定义 QMovied 对象，并初始化:

```
QMovie *movie = new QMovie(":/Mario.gif");
```

播放加载的动画：

```
movie->start();
```

将动画设置到 QLabel 中：

```
QLabel *label = new QLabel;
label->setMovie(movie);
```

8.2 QLineEdit

Qt 提供的单行文本编辑框。

8.2.1 设置/获取内容

- 获取编辑框内容使用 text（），函数声明如下：

  QString　text() const

- 设置编辑框内容

  void　setText(const QString &)

8.2.2 设置显示模式

使用 QLineEdit 类的 setEchoMode () 函数设置文本的显示模式,函数声明:

void　setEchoMode(EchoMode mode)

EchoMode 是一个枚举类型,一共定义了四种显示模式:

- QLineEdit::Normal　模式显示方式，按照输入的内容显示。
- QLineEdit::NoEcho 不显示任何内容，此模式下无法看到用户的输入。
- QLineEdit::Password　密码模式，输入的字符会根据平台转换为特殊字符。
- QLineEdit::PasswordEchoOnEdit　编辑时显示字符否则显示字符作为密码。

另外，我们再使用 QLineEdit 显示文本的时候，希望在左侧留出一段空白的区域，那么，就可以使用 QLineEdit 给我们提供的 setTextMargins 函数：

void setTextMargins(int left, int top, int right, int bottom)

用此函数可以指定显示的文本与输入框上下左右边界的间隔的像素数。

8.3 其他控件

Qt 中控件的使用方法可参考 Qt 提供的帮助文档。

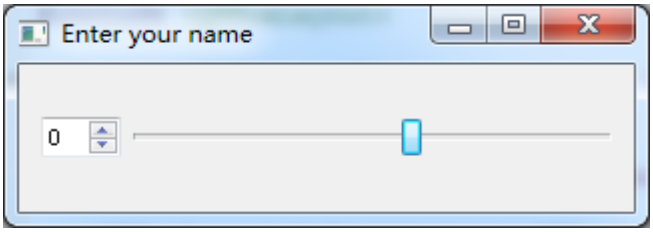
8.4 自定义控件

在搭建 Qt 窗口界面的时候，在一个项目中很多窗口，或者是窗口中的某个模块会被经常性的重复使用。一般遇到这种情况我们都会将这个窗口或者模块拿出来做成一个独立的窗口类，以备以后重复使用。

在使用 Qt 的 ui 文件搭建界面的时候，工具栏栏中只为我们提供了标准的窗口控件，如果我们想使用自定义控件怎么办？

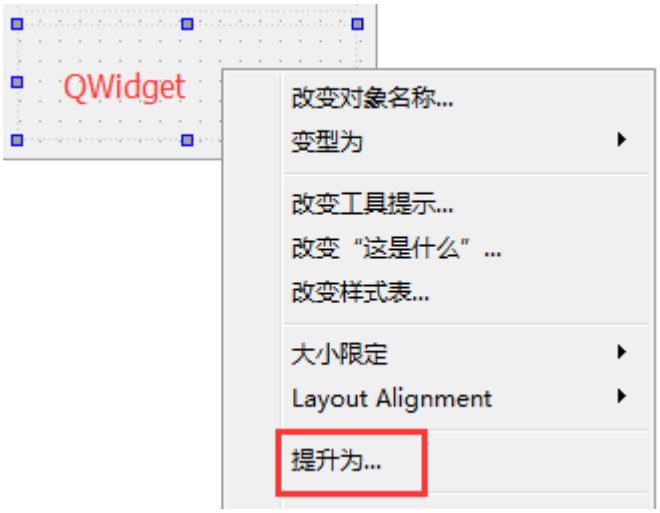
例如：我们从 QWidget 派生出一个类 SmallWidget，实现了一个自定窗口，

```
// smallwidget.h
class SmallWidget : public QWidget
{
    Q_OBJECT
public:
    explicit SmallWidget(QWidget *parent = 0);

signals:

public slots:
private:
    QSpinBox* spin;
    QSlider* slider;
};

// smallwidget.cpp
SmallWidget::SmallWidget(QWidget *parent) : QWidget(parent)
{
    spin = new QSpinBox(this);
    slider = new QSlider(Qt::Horizontal, this);

    // 创建布局对象
    QHBoxLayout* layout = new QHBoxLayout;
    // 将控件添加到布局中
    layout->addWidget(spin);
    layout->addWidget(slider);
    // 将布局设置到窗口中
    setLayout(layout);

    // 添加消息响应
    connect(spin,
            static_cast<void
(QSpinBox::*)(int)>(&QSpinBox::valueChanged),
            slider, &QSlider::setValue);
    connect(slider, &QSlider::valueChanged,
            spin, &QSpinBox::setValue);
}
```

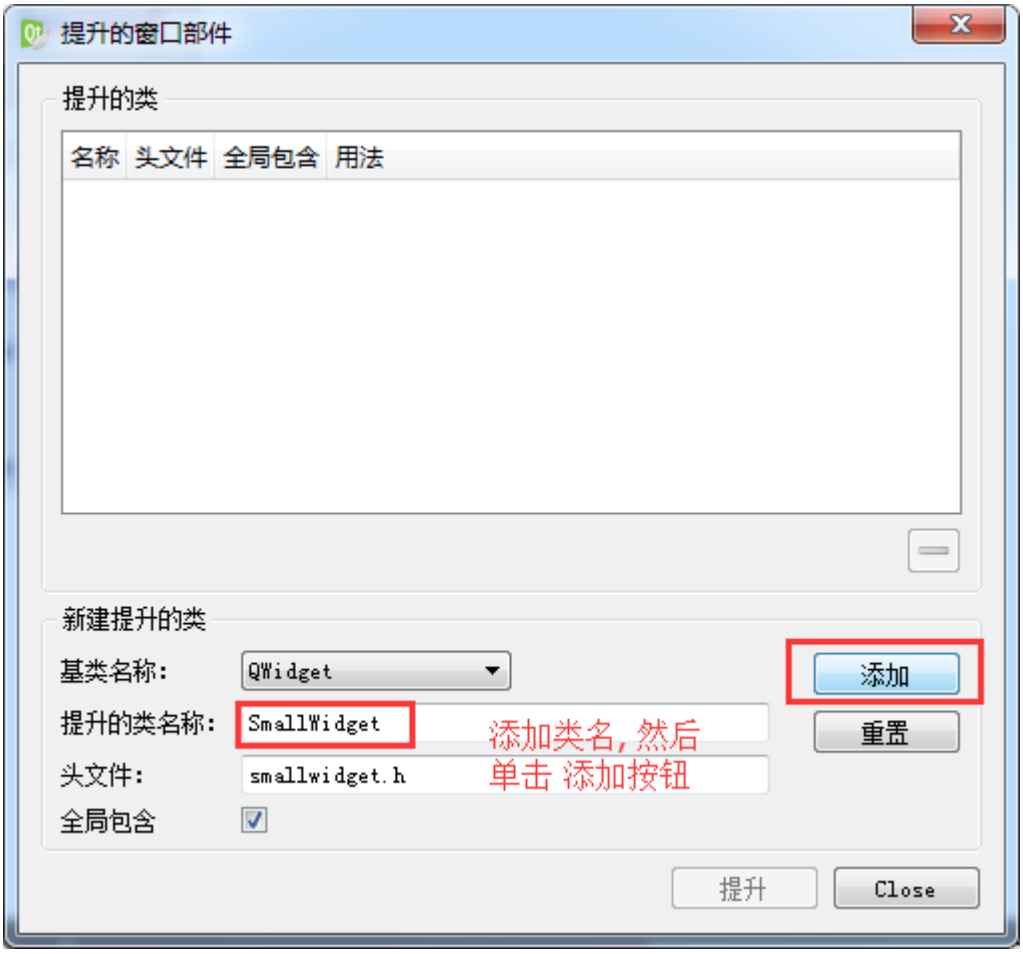


那么这个 SmallWidget 可以作为独立的窗口显示,也可以作为一个控件来使用：

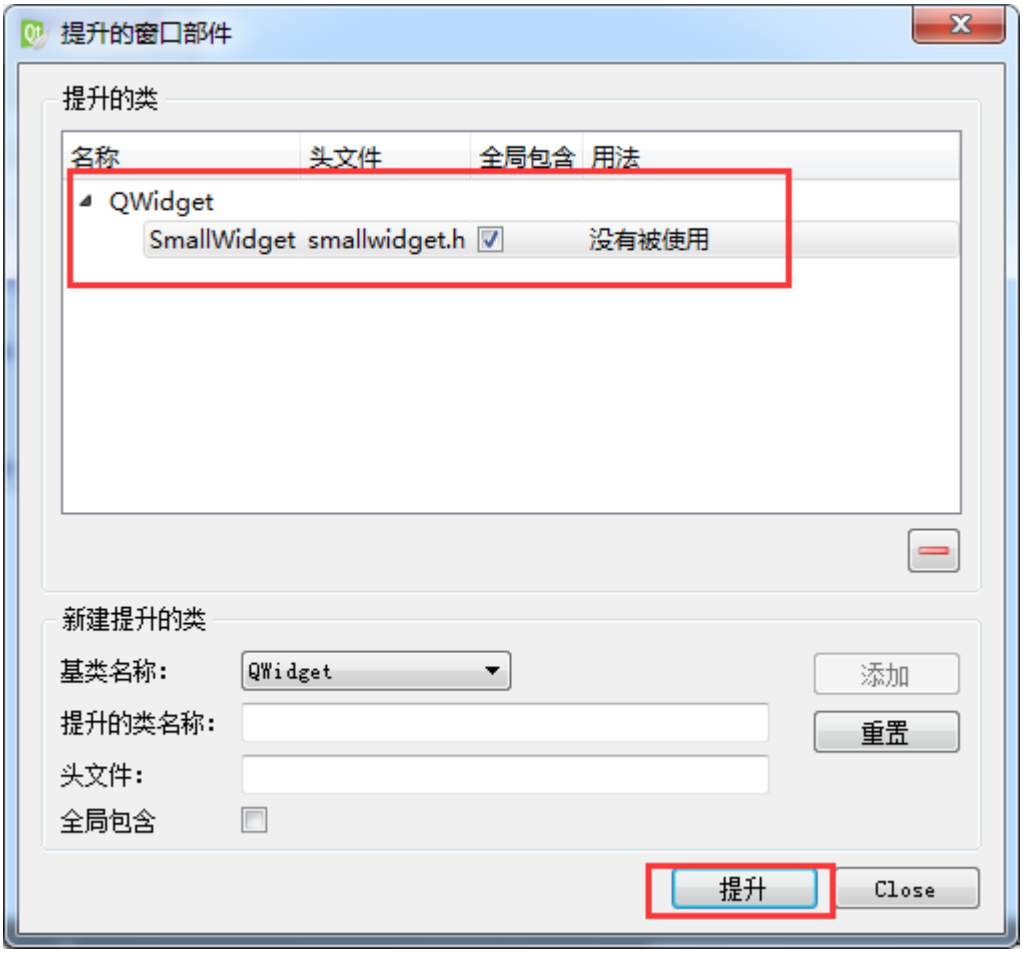
打开 Qt 的.ui 文件,因为 SmallWidget 是派生自 Qwidget 类,所以需要在 ui 文件中先放入一个 QWidget 控件，然后再上边鼠标右键



弹出提升窗口部件对话框



添加要提升的类的名字,然后选择 添加



添加之后,类名会显示到上边的列表框中,然后单击提升按钮,完成操作.

我们可以看到，这个窗口对应的类从原来的 QWidget 变成了 SmallWidget

再次运行程序,这个 widget_3 中就能显示出我们自定义的窗口了.

9 Qt 消息事件机制

9.1 事件

事件（event）是由系统或者 Qt 应用程序本身在不同的时刻发出的。当用户按下鼠标、敲下键盘，或者是窗口需要重新绘制的时候，都会发出一个相应的事件。一些事件在对用户操作做出响应时发出，如键盘事件等；另一些事件则是由系统自动发出，如计时器事件。

9.2 事件处理函数

在所有组件的父类 QWidget 中，定义了很多事件处理的函数，如

- keyPressEvent()：键盘按键按下事件
- keyReleaseEvent()：键盘按键松开事件
- mouseDoubleClickEvent()：鼠标双击事件
- mouseMoveEvent()：鼠标移动事件
- mousePressEvent()：鼠标按键按下事件
- mouseReleaseEvent() ：鼠标按键松开事件
- 等等

这些函数都是 protected virtual 的，也就是说，我们可以在子类中重新实现这些函数。下面来看一个例子：

```
class EventLabel : public QLabel
{
protected:
    void mouseMoveEvent(QMouseEvent *event);
    void mousePressEvent(QMouseEvent *event);
    void mouseReleaseEvent(QMouseEvent *event);
};

void EventLabel::mouseMoveEvent(QMouseEvent *event)
{
    this->setText(QString("<center><h1>Move:
(%1, %2)</h1></center>").arg(QString::number(event->x()),
                            QString::number(event->y())));
```

```
}

void EventLabel::mousePressEvent(QMouseEvent *event)
{

this->setText(QString("<center><h1>Press:(%1, %2)</h1></center>").arg
(QString::number(event->x()),
                          QString::number(event->y())));
}

void EventLabel::mouseReleaseEvent(QMouseEvent *event)
{
    QString msg;
    msg.sprintf("<center><h1>Release: (%d, %d)</h1></center>",
                event->x(), event->y());
    this->setText(msg);
}

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    EventLabel *label = new EventLabel;
    label->setWindowTitle("MouseEvent Demo");
    label->resize(300, 200);
    label->show();

    return a.exec();
}
```

- EventLabel 继承了 QLabel，覆盖了 mousePressEvent()、mouseMoveEvent()和 MouseReleaseEvent()三个函数。我们并没有添加什么功能，只是在鼠标按下（press）、鼠标移动（move）和鼠标释放（release）的时候，把当前鼠标的坐标值显示在这个 Label 上面。由于 QLabel 是支持 HTML 代码的，因此我们直接使用了 HTML 代码来格式化文字。

- QString 的 arg()函数可以自动替换掉 QString 中出现的占位符。其占位符以 % 开始，后面是占位符的位置，例如 %1，%2 这种。

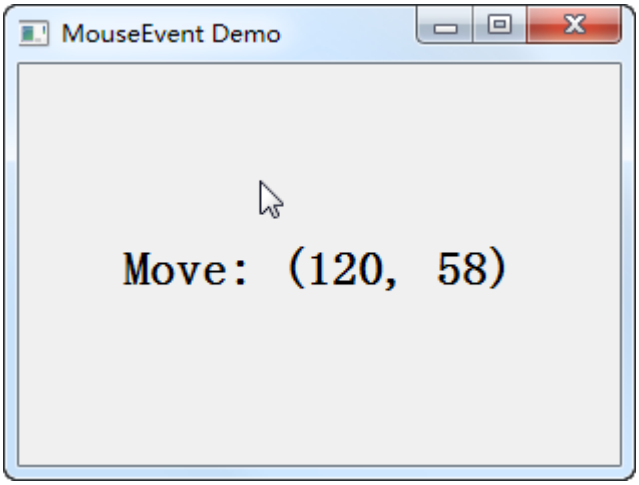
```
QString("[%1, %2]").arg(x).arg(y);
```

语句将会使用 x 替换 %1，y 替换 %2，因此，生成的 QString 为[x, y]。

- 在 mouseReleaseEvent()函数中，我们使用了另外一种 QString 的构造方法。

  我们使用类似 C 风格的格式化函数 sprintf()来构造 QString。

运行上面的代码，当我们点击了一下鼠标之后，label 上将显示鼠标当前坐标值。



为什么要点击鼠标之后才能在 mouseMoveEvent()函数中显示鼠标坐标值？

这是因为 QWidget 中有一个 mouseTracking 属性，该属性用于设置是否追踪鼠标。只有鼠标被追踪时，mouseMoveEvent()才会发出。如果 mouseTracking 是 false（默认即是），组件在至少一次鼠标点击之后，才能够被追踪，也就是能够发出 mouseMoveEvent()事件。如果 mouseTracking 为 true，则 mouseMoveEvent()直接可以被发出。

知道了这一点，我们就可以在 main()函数中添加如下代码：

```
label->setMouseTracking(true);
```

在运行程序就没有这个问题了。

9.3 事件分发函数 event()

事件对象创建完毕后，Qt 将这个事件对象传递给 QObject 的 event()函数。event()函数并不直接处理事件，而是将这些事件对象按照它们不同的类型，分发给不同的事件处理器（event handler）。

如上所述，**event()函数主要用于事件的分发**。所以，如果你希望在事件分发之前做一些操作，就可以重写这个 event()函数了。例如，我们希望在一个 QWidget 组件中监听 tab 键的按下，那么就可以继承 QWidget，并重写它的 event()函数，来达到这个目的：

```
bool CustomWidget::event(QEvent *e)
{
```

```
    if (e->type() == QEvent::KeyPress) {
        QKeyEvent *keyEvent = static_cast<QKeyEvent *>(e);
        if (keyEvent->key() == Qt::Key_Tab) {
            qDebug() << "You press tab.";
            return true;
        }
    }
    return QWidget::event(e);
}
```

CustomWidget 是一个普通的 QWidget 子类。我们重写了它的 event()函数，这个函数有一个 QEvent 对象作为参数，也就是需要转发的事件对象。函数返回值是 bool 类型。

- **如果传入的事件已被识别并且处理，则需要返回 true，否则返回 false。如果返回值是 true，那么 Qt 会认为这个事件已经处理完毕，不会再将这个事件发送给其它对象，而是会继续处理事件队列中的下一事件。**

- **在 event()函数中，调用事件对象的 accept()和 ignore()函数是没有作用的，不会影响到事件的传播。**

我们可以通过使用 QEvent::type()函数可以检查事件的实际类型，其返回值是 QEvent::Type 类型的枚举。我们处理过自己感兴趣的事件之后，可以直接返回 true，表示我们已经对此事件进行了处理；对于其它我们不关心的事件，则需要调用父类的 event()函数继续转发，否则这个组件就只能处理我们定义的事件了。为了测试这一种情况，我们可以尝试下面的代码：

```
bool CustomTextEdit::event(QEvent *e)
{
   if (e->type() == QEvent::KeyPress)
{
        QKeyEvent *keyEvent = static_cast<QKeyEvent *>(e);
       if (keyEvent->key() == Qt::Key_Tab)
{
            qDebug() << "You press tab.";
            return true;
       }
    }
    return false;
}
```

CustomTextEdit 是 QTextEdit 的一个子类。我们重写了其 event()函数，却没有调用父类的同名函数。这样，我们的组件就只能处理 Tab 键，再也无法输入任何文本，也不能响应其它事件，比如鼠标点击之后也不会有光标出现。这是

因为我们只处理的 KeyPress 类型的事件，并且如果不是 KeyPress 事件，则直接返回 false，鼠标事件根本不会被转发，也就没有了鼠标事件。

通过查看 QObject::event()的实现，我们可以理解，event()函数同前面的章节中我们所说的事件处理器有什么联系：

```
//!!! Qt5
bool QObject::event(QEvent *e)
{
    switch (e->type()) {
    case QEvent::Timer:
        timerEvent((QTimerEvent*)e);
        break;

    case QEvent::ChildAdded:
    case QEvent::ChildPolished:
    case QEvent::ChildRemoved:
        childEvent((QChildEvent*)e);
        break;
    // ...
    default:
        if (e->type() >= QEvent::User) {
            customEvent(e);
            break;
        }
        return false;
    }
    return true;
}
```

这是 Qt 5 中 QObject::event()函数的源代码（Qt 4 的版本也是类似的）。我们可以看到，同前面我们所说的一样，Qt 也是使用 QEvent::type()判断事件类型，然后调用了特定的事件处理器。比如，如果 event->type()返回值是 QEvent::Timer，则调用 timerEvent()函数。可以想象，QWidget::event()中一定会有如下的代码：

```
    switch (event->type()) {
        case QEvent::MouseMove:
            mouseMoveEvent((QMouseEvent*)event);
            break;
        // ...
    }
```

事实也的确如此。timerEvent()和 mouseMoveEvent()这样的函数，就是我们前面章节所说的事件处理器 event handler。也就是说，event()函数中实际是

通过事件处理器来响应一个具体的事件。这相当于 event()函数将具体事件的处理“委托”给具体的事件处理器。而这些事件处理器是 protected virtual 的，因此，我们重写了某一个事件处理器，即可让 Qt 调用我们自己实现的版本。

由此可以见，event()是一个集中处理不同类型的事件的地方。如果你不想重写一大堆事件处理器，就可以重写这个 event()函数，通过 QEvent::type()判断不同的事件。鉴于重写 event()函数需要十分小心注意父类的同名函数的调用，一不留神就可能出现问题，所以一般还是建议只重写事件处理器（当然，也必须记得是不是应该调用父类的同名处理器）。这其实暗示了 event()函数的另外一个作用：屏蔽掉某些不需要的事件处理器。正如我们前面的 CustomTextEdit 例子看到的那样，我们创建了一个只能响应 tab 键的组件。这种作用是重写事件处理器所不能实现的。

9.4 事件过滤器

有时候，对象需要查看、甚至要拦截发送到另外对象的事件。例如，对话框可能想要拦截按键事件，不让别的组件接收到；或者要修改回车键的默认处理。

通过前面的章节，我们已经知道，Qt 创建了 QEvent 事件对象之后，会调用 QObject 的 event()函数处理事件的分发。显然，我们可以在 event()函数中实现拦截的操作。由于 event()函数是 protected 的，因此，需要继承已有类。如果组件很多，就需要重写很多个 event()函数。这当然相当麻烦，更不用说重写 event()函数还得小心一堆问题。好在 Qt 提供了另外一种机制来达到这一目的：事件过滤器。

QObject 有一个 eventFilter()函数，用于建立事件过滤器。函数原型如下：

```
virtual bool QObject::eventFilter ( QObject * watched, QEvent * event );
```

这个函数正如其名字显示的那样，是一个“事件过滤器”。所谓事件过滤器，可以理解成一种过滤代码。事件过滤器会检查接收到的事件。如果这个事件是我们感兴趣的类型，就进行我们自己的处理；如果不是，就继续转发。这个函数返回一个 bool 类型，如果你想将参数 event 过滤出来，比如，**不想让它继续转发，就返回 true，否则返回 false。**事件过滤器的调用时间是目标对象（也就是参数里面的 watched 对象）接收到事件对象之前。也就是说，如果你在事件过滤器中停止了某个事件，那么，watched 对象以及以后所有的事件过滤器根本不会知道这么一个事件。

我们来看一段简单的代码：

```
class MainWindow : public QMainWindow
{
public:
    MainWindow();
protected:
    bool eventFilter(QObject *obj, QEvent *event);
private:
```

```
    QTextEdit *textEdit;
};

MainWindow::MainWindow()
{
    textEdit = new QTextEdit;
    setCentralWidget(textEdit);
    textEdit->installEventFilter(this);
}

bool MainWindow::eventFilter(QObject *obj, QEvent *event)
{
    if (obj == textEdit) {
        if (event->type() == QEvent::KeyPress) {
            QKeyEvent *keyEvent = static_cast<QKeyEvent *>(event);
            qDebug() << "Ate key press" << keyEvent->key();
            return true;
        } else {
            return false;
        }
    } else {
        // pass the event on to the parent class
        return QMainWindow::eventFilter(obj, event);
    }
}
```

- MainWindow 是我们定义的一个类。我们重写了它的 eventFilter()函数。为了过滤特定组件上的事件，首先需要判断这个对象是不是我们感兴趣的组件，然后判断这个事件的类型。在上面的代码中，我们不想让 textEdit 组件处理键盘按下的事件。所以，首先我们找到这个组件，如果这个事件是键盘事件，则直接返回 true，也就是过滤掉了这个事件，其他事件还是要继续处理，所以返回 false。对于其它的组件，我们并不保证是不是还有过滤器，于是最保险的办法是调用父类的函数。

- eventFilter()函数相当于创建了过滤器，然后我们需要安装这个过滤器。安装过滤器需要调用 QObject::installEventFilter()函数。函数的原型如下：

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

这个函数接受一个 QObject *类型的参数。记得刚刚我们说的，eventFilter()函数是 QObject 的一个成员函数，因此，任意 QObject 都可以作为事件过滤器

（问题在于，如果你没有重写 eventFilter()函数，这个事件过滤器是没有任何作用的，因为默认什么都不会过滤）。已经存在的过滤器则可以通过 QObject::removeEventFilter()函数移除。

- 我们可以向一个对象上面安装多个事件处理器，只要调用多次 installEventFilter()函数。如果一个对象存在多个事件过滤器，那么，最后一个安装的会第一个执行，也就是后进先执行的顺序。

还记得我们前面的那个例子吗？我们使用 event()函数处理了 Tab 键：

```
bool CustomWidget::event(QEvent *e)
{
    if (e->type() == QEvent::KeyPress) {
        QKeyEvent *keyEvent = static_cast<QKeyEvent *>(e);
        if (keyEvent->key() == Qt::Key_Tab) {
            qDebug() << "You press tab.";
            return true;
        }
    }
    return QWidget::event(e);
}
```

现在，我们可以给出一个使用事件过滤器的版本：

```
bool FilterObject::eventFilter(QObject *object, QEvent *event)
{
    if (object == target && event->type() == QEvent::KeyPress)
    {
        QKeyEvent *keyEvent = static_cast<QKeyEvent *>(event);
        if (keyEvent->key() == Qt::Key_Tab) {
            qDebug() << "You press tab.";
            return true;
        } else {
            return false;
        }
    }
    return false;
}
```

事件过滤器的强大之处在于，我们可以为整个应用程序添加一个事件过滤器。记得，installEventFilter()函数是 QObject 的函数，QApplication 或者 QCoreApplication 对象都是 QObject 的子类，因此，我们可以向 QApplication 或者 QCoreApplication 添加事件过滤器。**这种全局的事件过滤器将会在所有其它特性对象的事件过滤器之前调用。尽管很强大，但这种行为会严重降低整个应用程序的事件分发效率。**因此，除非是不得不使用的情况，否则的话我们不应

该这么做。

注意：

事件过滤器和被安装过滤器的组件必须在同一线程创建，否则，过滤器将不起作用。另外，如果在安装过滤器之后，这两个组件到了不同的线程，那么，只有等到二者重新回到同一线程的时候过滤器才会有效。

9.5 消息事件机制和信号和槽机制的关系

来看下官方文档对信号和槽以及事件机制的原文和翻译：

In Qt, we have an alternative to the callback technique: We use signals and slots. A signal is emitted when a particular event occurs. Qt's widgets have many predefined signals, but we can always subclass widgets to add our own signals to them. A slot is a function that is called in response to a particular signal. Qt's widgets have many pre-defined slots, but it is common practice to subclass widgets and add your own slots so that you can handle the signals that you are interested in.

在 Qt 中，我们有一个替代回调技术:我们使用信号和槽。当某个特定事件发生时发出一个信号。Qt 的小部件有许多预定义的信号，但我们总是可以将小部件添加到它们中。槽是响应特定信号的函数。Qt 的小部件有许多预定义的槽，但是子类化小部件并添加自己的槽是常见的做法，这样您就可以处理您感兴趣的信号。

In Qt, events are objects, derived from the abstract QEvent class, that represent things that have happened either within an application or as a result of outside activity that the application needs to know about. Events can be received and handled by any instance of a QObject subclass, but they are especially relevant to widgets.

在 Qt 中，事件是对象，从抽象的 QEvent 类派生出来，它表示应用程序中发生的事件或者应用程序需要知道的外部活动的结果。事件可以由 QObject 子类的任何实例接收和处理，但它们与小部件特别相关。

简单总结可以有以下几点

- 两者都是 Qt 中的通讯机制
- 信号和槽被用于程序内对象之间的通信，也是一种程序回调机制（回调的意思是让框架调用，某个函数叫回调函数意思就是这个函数不是我们手动调用的，而是把函数指针告诉框架，然后框架在适当时机调用）。
- 事件不仅有程序内部产生的，也有程序外部、系统产生的，如：鼠标、键盘、定时器事件
- 信号和槽都是函数，当某个信号发生了就调用相关联的槽函数。

- 事件是对象，代表着一条消息。其他对象接收到这个事件要对这个事件做相应的处理，也可以不做任何处理。
- 消息事件比信号槽更底层，一般 Qt 中的标准信号会是某些消息事件处理的结果，比如：鼠标移动到按钮上，按钮一直收到鼠标移动的事件，可是按钮没有做任何处理，但是只要当鼠标点击按钮，按钮就会收到一个鼠标点击的事件，他会处理这个事件并且主动去发送一个被点击的信号 clicked，这时候相关联的槽函数就会被调用。

在前面我们也曾经简单提到，**Qt 程序**需要在 main()函数创建一个 QApplication 对象，然后调用它的 exec()函数。这个函数就是开始 Qt 的消息事件循环，不断地处理收到的消息事件。当事件发生时，Qt 框架将创建一个事件对象。**Qt 中所有事件类都继承于 QEvent**。在事件对象创建完毕后，Qt 将这个事件对象传递给 QObject 的 event()函数。**event()函数并不直接处理事件，而是按照事件对象的类型分派给特定的事件处理函数**（event handler），关于这一点，会在后边详细说明。

9.6 总结

Qt 的事件是整个 Qt 框架的核心机制之一，也比较复杂。说它复杂，更多是因为它涉及到的函数众多，而处理方法也很多，有时候让人难以选择。现在我们简单总结一下 Qt 中的事件机制。

Qt 中有很多种事件：鼠标事件、键盘事件、大小改变的事件、位置移动的事件等等。如何处理这些事件，实际有两种选择：

- 所有事件对应一个事件处理函数，在这个事件处理函数中用一个很大的分支
- 每一种事件对应一个事件处理函数。Qt 就是使用的这么一种机制：
  - mouseEvent()
  - keyPressEvent()
  - …

Qt 具有这么多种事件处理函数，肯定有一个地方对其进行分发，否则，Qt 怎么知道哪一种事件调用哪一个事件处理函数呢？这个分发的函数，就是 event()。显然，当 QMouseEvent 产生之后，event()函数将其分发给 mouseEvent()事件处理器进行处理。

event()函数会有两个问题：

- event()函数是一个 protected 的函数，这意味着我们要想重写 event()，必须继承一个已有的类。试想，我的程序根本不想要鼠标事件，程序中所有组件都不允许处理鼠标事件，是不是我得继承所有组件，一一重写其 event()

函数？protected 函数带来的另外一个问题是，如果我基于第三方库进行开发，而对方没有提供源代码，只有一个链接库，其它都是封装好的。我怎么去继承这种库中的组件呢？

- event()函数的确有一定的控制，不过有时候我的需求更严格一些：我希望那些组件根本看不到这种事件。event()函数虽然可以拦截，但其实也是接收到了 QMouseEvent 对象。我连让它收都收不到。这样做的好处是，模拟一种系统根本没有那个事件的效果，所以其它组件根本不会收到这个事件，也就无需修改自己的事件处理函数。这种需求怎么办呢？

这两个问题是 event()函数无法处理的。于是，Qt 提供了另外一种解决方案：事件过滤器。事件过滤器给我们一种能力，让我们能够完全移除某种事件。事件过滤器可以安装到任意 QObject 类型上面，并且可以安装多个。如果要实现全局的事件过滤器，则可以安装到 QApplication 或者 QCoreApplication 上面。这里需要注意的是，如果使用 installEventFilter()函数给一个对象安装事件过滤器，那么该事件过滤器只对该对象有效，只有这个对象的事件需要先传递给事件过滤器的 eventFilter()函数进行过滤，其它对象不受影响。如果给 QApplication 对象安装事件过滤器，那么该过滤器对程序中的每一个对象都有效，任何对象的事件都是先传给 eventFilter()函数。

事件过滤器可以解决刚刚我们提出的 event()函数的两点不足：

- **首先，事件过滤器不是 protected 的，因此我们可以向任何 QObject 子类安装事件过滤器；**
- **其次，事件过滤器在目标对象接收到事件之前进行处理，如果我们将事件过滤掉，目标对象根本不会见到这个事件。**

事实上，还有一种方法，我们没有介绍。Qt 事件的调用最终都会追溯到 QCoreApplication::notify()函数，因此，最大的控制权实际上是重写 QCoreApplication::notify()。这个函数的声明是：

```
virtual bool QCoreApplication::notify ( QObject * receiver,
                                        QEvent * event );
```

该函数会将 event 发送给 receiver，也就是调用 receiver->event(event)，其返回值就是来自 receiver 的事件处理器。注意，这个函数为任意线程的任意对象的任意事件调用，因此，它不存在事件过滤器的线程的问题。不过我们并不推荐这么做，因为 notify()函数只有一个，而事件过滤器要灵活得多。

现在我们可以总结一下 Qt 的事件处理，实际上是有五个层次：

- 重写 paintEvent()、mousePressEvent()等事件处理函数。这是最普通、最简单的形式，同时功能也最简单。
- 重写 event()函数。event()函数是所有对象的事件入口，QObject 和 QWidget 中的实现，默认是把事件传递给特定的事件处理函数。
- 在特定对象上面安装事件过滤器。该过滤器仅过滤该对象接收到的事件。
- 在 QCoreApplication::instance()上面安装事件过滤器。该过滤器将过滤所有对象的所有事件，因此和 notify()函数一样强大，但是它更灵活，因为可以安装多个过滤器。全局的事件过滤器可以看到 disabled 组件上面发出的鼠标事件。全局过滤器有一个问题：只能用在主线程。
- 重写 QCoreApplication::notify()函数。这是最强大的，和全局事件过滤器一样提供完全控制，并且不受线程的限制。但是全局范围内只能有一个被使用（因为 QCoreApplication 是单例的）。

10 绘图事件和绘图设备

10.1 QPainter

Qt 的绘图系统允许使用相同的 API 在屏幕和其它打印设备上进行绘制。整个绘图系统基于 QPainter，QPainterDevice 和 QPaintEngine 三个类。

QPainter 用来执行绘制的操作；**QPaintDevice** 是一个二维空间的抽象，这个二维空间允许 QPainter 在其上面进行绘制，也就是 QPainter 工作的空间；**QPaintEngine** 提供了画家（QPainter）在不同的设备上进行绘制的统一的接口。QPaintEngine 类应用于 QPainter 和 QPaintDevice 之间，通常对开发人员是透明的。除非你需要自定义一个设备，否则你是不需要关心 QPaintEngine 这个类的。我们可以把 QPainter 理解成画笔；把 QPaintDevice 理解成使用画笔的地方，比如纸张、屏幕等；而对于纸张、屏幕而言，肯定要使用不同的画笔绘制，为了统一使用一种画笔，我们设计了 QPaintEngine 类，这个类让不同的纸张、屏幕都能使用一种画笔。

下图给出了这三个类之间的层次结构:

上面的示意图告诉我们，Qt 的绘图系统实际上是，使用 QPainter 在 QPainterDevice 上进行绘制，它们之间使用 QPaintEngine 进行通讯（也就是翻译 QPainter 的指令）。

下面我们通过一个实例来介绍 QPainter 的使用：

```
class PaintedWidget : public QWidget
{
    Q_OBJECT
public:
    PaintedWidget(QWidget *parent = 0);
protected:
    void paintEvent(QPaintEvent *);
}
```

注意我们重写了 QWidget 的 paintEvent()函数。接下来就是 PaintedWidget 的源代码：

```
PaintedWidget::PaintedWidget(QWidget *parent) :
    QWidget(parent)
{
    resize(800, 600);
    setWindowTitle(tr("Paint Demo"));
}

void PaintedWidget::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    painter.drawLine(80, 100, 650, 500);
    painter.setPen(Qt::red);
    painter.drawRect(10, 10, 100, 400);
    painter.setPen(QPen(Qt::green, 5));
    painter.setBrush(Qt::blue);
    painter.drawEllipse(50, 150, 400, 200);
}
```

在构造函数中，我们仅仅设置了窗口的大小和标题。而 paintEvent()函数则是绘制的代码。首先，我们在栈上创建了一个 QPainter 对象，也就是说，每次运行 paintEvent()函数的时候，都会重建这个 QPainter 对象。注意，这一点可能会引发某些细节问题：由于我们每次重建 QPainter，因此第一次运行时所设置的画笔颜色、状态等，第二次再进入这个函数时就会全部丢失。有时候我们希望保存画笔状态，就必须自己保存数据，否则的话则需要将 QPainter 作为类的成员变量。

QPainter 接收一个 QPaintDevice 指针作为参数。QPaintDevice 有很多子类，比如 QImage，以及 QWidget。注意回忆一下，QPaintDevice 可以理解成

要在哪里去绘制，而现在我们希望画在这个组件，因此传入的是 this 指针。

QPainter 有很多以 draw 开头的函数，用于各种图形的绘制，比如这里的 drawLine()，drawRect()以及 drawEllipse()等。当绘制轮廓线时，使用 QPainter 的 pen()属性。比如，我们调用了 painter.setPen(Qt::red)将 pen 设置为红色，则下面绘制的矩形具有红色的轮廓线。接下来，我们将 pen 修改为绿色，5 像素宽（painter.setPen(QPen(Qt::green, 5)），又设置了画刷为蓝色。这时候再调用 draw 函数，则是具有绿色 5 像素宽轮廓线、蓝色填充的椭圆。

10.2 绘图设备

绘图设备是指继承 QPainterDevice 的子类。Qt 一共提供了四个这样的类，分别是 QPixmap、QBitmap、QImage 和 QPicture。其中，

- QPixmap 专门为图像在屏幕上的显示做了优化
- QBitmap 是 QPixmap 的一个子类，它的色深限定为 1，可以使用 QPixmap 的 isQBitmap()函数来确定这个 QPixmap 是不是一个 QBitmap。
- QImage 专门为图像的像素级访问做了优化。
- QPicture 则可以记录和重现 QPainter 的各条命令。

10.2.1QPixmap、QBitmap、QImage

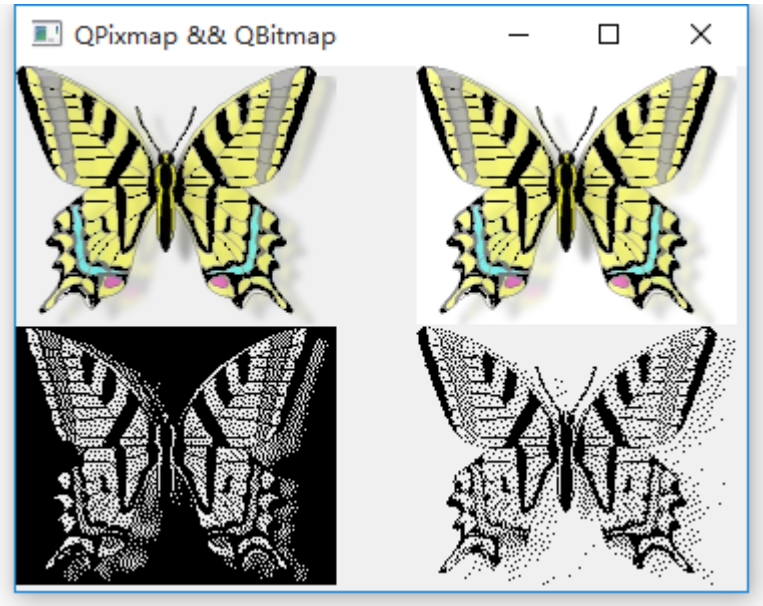
QPixmap 继承了 QPaintDevice，因此，你可以使用 QPainter 直接在上面绘制图形。QPixmap 也可以接受一个字符串作为一个文件的路径来显示这个文件，比如你想在程序之中打开 png、jpeg 之类的文件，就可以使用 QPixmap。使用 QPainter 的 drawPixmap()函数可以把这个文件绘制到一个 QLabel、QPushButton 或者其他的设备上面。**QPixmap 是针对屏幕进行特殊优化的，因此，它与实际的底层显示设备息息相关。**注意，这里说的显示设备并不是硬件，而是操作系统提供的原生的绘图引擎。所以，在不同的操作系统平台下，QPixmap 的显示可能会有所差别。

QBitmap 继承自 QPixmap，因此具有 QPixmap 的所有特性，提供单色图像。QBitmap 的色深始终为 1. 色深这个概念来自计算机图形学，是指用于表现颜色的二进制的位数。我们知道，计算机里面的数据都是使用二进制表示的。为了表示一种颜色，我们也会使用二进制。比如我们要表示 8 种颜色，需要用 3 个二进制位，这时我们就说色深是 3. 因此，所谓色深为 1，也就是使用 1 个二进制位表示颜色。1 个位只有两种状态：0 和 1，因此它所表示的颜色就有两种，黑和白。所以说，**QBitmap 实际上是只有黑白两色的图像数据。**

由于 QBitmap 色深小，因此只占用很少的存储空间，所以适合做光标文件和笔刷。

下面我们来看同一个图像文件在 QPixmap 和 QBitmap 下的不同表现：

```
void PaintWidget::paintEvent(QPaintEvent *)
{
    QPixmap pixmap(":/Image/butterfly.png");
    QPixmap pixmap1(":/Image/butterfly1.png");
    QBitmap bitmap(":/Image/butterfly.png");
    QBitmap bitmap1(":/Image/butterfly1.png");

    QPainter painter(this);
    painter.drawPixmap(0, 0, pixmap);
    painter.drawPixmap(200, 0, pixmap1);
    painter.drawPixmap(0, 130, bitmap);
    painter.drawPixmap(200, 130, bitmap1);
}
```



这里我们给出了两张 png 图片。butterfly1.png 是没有透明色的纯白背景，而 butterfly.png 是具有透明色的背景。我们分别使用 QPixmap 和 QBitmap 来加载它们。注意看它们的区别：白色的背景在 QBitmap 中消失了，而透明色在 QBitmap 中转换成了黑色；其他颜色则是使用点的疏密程度来体现（像以前黑白报纸图片打印原理一样）的。

QPixmap 使用底层平台的绘制系统进行绘制，无法提供像素级别的操作，而 QImage 则是使用独立于硬件的绘制系统，实际上是自己绘制自己，因此提供了像素级别的操作，并且能够在不同系统之上提供一个一致的显示形式。

我们声明了一个 QImage 对象，大小是 300 x 300，颜色模式是 RGB32，即使用 32 位数值表示一个颜色的 RGB 值，也就是说每种颜色使用 8 位。然后我们对每个像素进行颜色赋值，从而构成了这个图像。我们可以把 QImage 想象成一个 RGB 颜色的二维数组，记录了每一像素的颜色。

```
void PaintWidget::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    QImage image(300, 300, QImage::Format_RGB32);
    QRgb value;

    //将图片背景填充为白色
    image.fill(Qt::white);

    //改变指定区域的像素点的值
    for(int i=50; i<100; ++i)
    {
        for(int j=50; j<100; ++j)
        {
            value = qRgb(255, 0, 0); // 红色
            image.setPixel(i, j, value);
        }
    }

    //将图片绘制到窗口中
    painter.drawImage(QPoint(0, 0), image);
}
```

QImage 与 QPixmap 的区别

- QPixmap 主要是用于绘图，针对屏幕显示而最佳化设计，QImage 主要是为图像 I/O、图片访问和像素修改而设计的
- QPixmap 依赖于所在的平台的绘图引擎，故例如反锯齿等一些效果在不同

的平台上可能会有不同的显示效果，QImage 使用 Qt 自身的绘图引擎，可在不同平台上具有相同的显示效果

- 由于 QImage 是独立于硬件的，也是一种 QPaintDevice，因此我们可以在另一个线程中对其进行绘制，而不需要在 GUI 线程中处理，使用这一方式可以很大幅度提高 UI 响应速度。

- QImage 可通过 setPixpel()和 pixel()等方法直接存取指定的像素。

QImage 与 QPixmap 之间的转换:

- QImage 转 QPixmap

使用 QPixmap 的静态成员函数: fromImage()

```
QPixmap fromImage(const QImage & image,
Qt::ImageConversionFlags flags = Qt::AutoColor);
```

- QPixmap 转 QImage:

使用 QPixmap 类的成员函数: toImage()

```
QImage toImage() const;
```

10.2.2QPicture

最后一个需要说明的是 QPicture。**这是一个可以记录和重现 QPainter 命令的绘图设备。 QPicture 将 QPainter 的命令序列化到一个 IO 设备，保存为一个平台独立的文件格式。**这种格式有时候会是“元文件(meta- files)”。Qt 的这种格式是二进制的，不同于某些本地的元文件，Qt 的 pictures 文件没有内容上的限制，只要是能够被 QPainter 绘制的元素，不论是字体还是 pixmap，或者是变换，都可以保存进一个 picture 中。

QPicture 是平台无关的，因此它可以使用在多种设备之上，比如 svg、pdf、ps、打印机或者屏幕。回忆下我们这里所说的 QPaintDevice，实际上是说可以有 QPainter 绘制的对象。QPicture 使用系统的分辨率，并且可以调整 QPainter 来消除不同设备之间的显示差异。

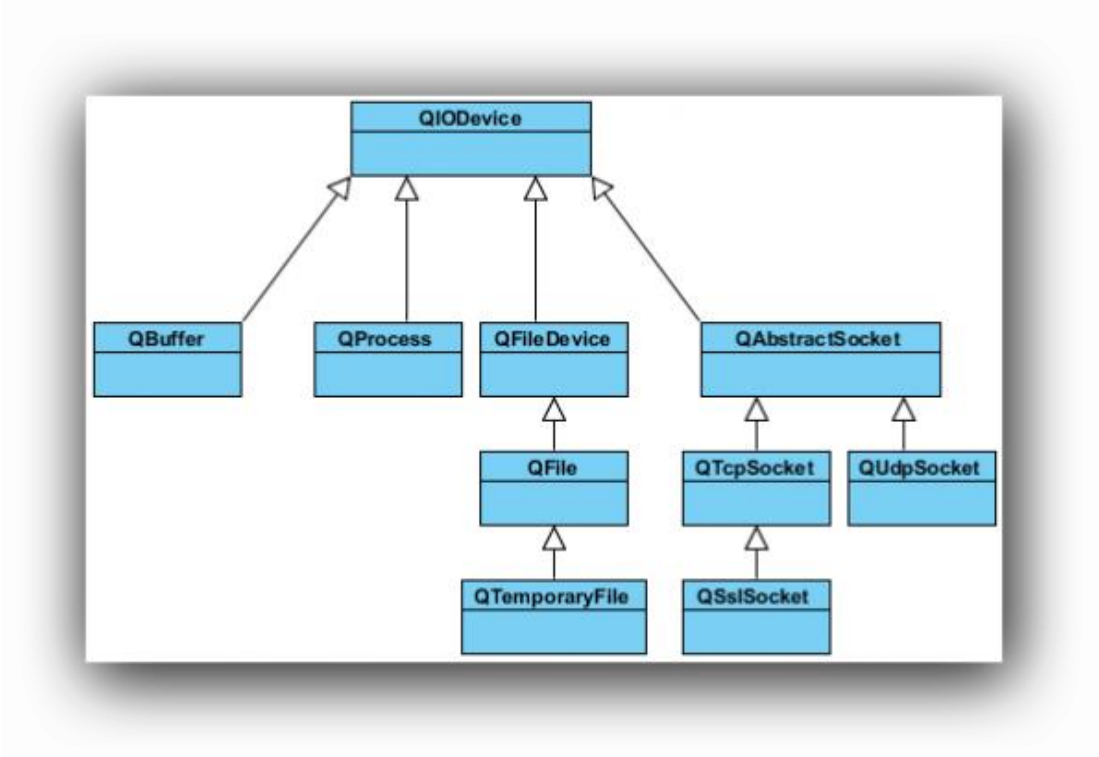
如果我们要记录下 QPainter 的命令，首先要使用 QPainter::begin()函数，将 QPicture 实例作为参数传递进去，以便告诉系统开始记录，记录完毕后使用 QPainter::end()命令终止。代码示例如下：

```
void PaintWidget::paintEvent(QPaintEvent *)
```

```
{
    QPicture pic;
    QPainter painter;
        //将图像绘制到 QPicture 中,并保存到文件
    painter.begin(&pic);
    painter.drawEllipse(20, 20, 100, 50);
    painter.fillRect(20, 100, 100, 100, Qt::red);
    painter.end();
    pic.save("D:\\drawing.pic");

        //将保存的绘图动作重新绘制到设备上
    pic.load("D:\\drawing.pic");
    painter.begin(this);
    painter.drawPicture(200, 200, pic);
    painter.end();
}
```

11 文件操作

文件操作是应用程序必不可少的部分。Qt 作为一个通用开发库，提供了跨平台的文件操作能力。Qt 通过 QIODevice 提供了对 I/O 设备的抽象，这些设备具有读写字节块的能力。下面是 I/O 设备的类图（Qt5）：



- QIODevice：所有 I/O 设备类的父类，提供了字节块读写的通用操作以及基

本接口；

- QFileDevice：Qt5 新增加的类，提供了有关文件操作的通用实现。
- QFlie：访问本地文件或者嵌入资源；
- QTemporaryFile：创建和访问本地文件系统的临时文件；
- QBuffer：读写 QbyteArray，内存文件；
- QProcess：运行外部程序，处理进程间通讯；
- QAbstractSocket：所有套接字类的父类；
- QTcpSocket：TCP 协议网络数据传输；
- QUdpSocket：传输 UDP 报文；
- QSslSocket：使用 SSL/TLS 传输数据；

文件系统分类:

- 顺序访问设备:

  是指它们的数据只能访问一遍：从头走到尾，从第一个字节开始访问，直到最后一个字节，中途不能返回去读取上一个字节，这其中，QProcess、QTcpSocket、QUdpSoctet 和 QSslSocket 是顺序访问设备。

- 随机访问设备:

可以访问任意位置任意次数，还可以使用 QIODevice::seek()函数来重新定位文件访问位置指针，QFile、QTemporaryFile 和 QBuffer 是随机访问设备，

11.1 基本文件操作

文件操作是应用程序必不可少的部分。Qt 作为一个通用开发库，提供了跨平台的文件操作能力。在所有的 I/O 设备中，文件 I/O 是最重要的部分之一。因为我们大多数的程序依旧需要首先访问本地文件（当然，在云计算大行其道的将来，这一观点可能改变）。**QFile 提供了从文件中读取和写入数据的能力。**

我们通常会将文件路径作为参数传给 QFile 的构造函数。不过也可以在创建好对象最后，使用 setFileName()来修改。QFile 需要使用 / 作为文件分隔符，不过，它会自动将其转换成操作系统所需要的形式。例如 C:/windows 这样的路径在 Windows 平台下同样是可以的。

QFile 主要提供了有关文件的各种操作，比如打开文件、关闭文件、刷新文件等。我们可以使用 QDataStream 或 QTextStream 类来读写文件，也可以使用 QIODevice 类提供的 read()、readLine()、readAll()以及 write()这样的函数。值得注意的是，有关文件本身的信息，比如文件名、文件所在目录的名字等，则是通过 QFileInfo 获取，而不是自己分析文件路径字符串。

下面我们使用一段代码来看看 QFile 的有关操作：

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

    QFile file("in.txt");
    if (!file.open(QIODevice::ReadOnly | QIODevice::Text)) {
        qDebug() << "Open file failed.";
        return -1;
    } else {
        while (!file.atEnd()) {
            qDebug() << file.readLine();
        }
    }

    QFileInfo info(file);
    qDebug() << info.isDir();
    qDebug() << info.isExecutable();
    qDebug() << info.baseName();
    qDebug() << info.completeBaseName();
    qDebug() << info.suffix();
    qDebug() << info.completeSuffix();

    return app.exec();
}
```

- 我们首先使用 QFile 创建了一个文件对象。

  这个文件名字是 in.txt。如果你不知道应该把它放在哪里，可以使用 QDir::currentPath()来获得应用程序执行时的当前路径。只要将这个文件放在与当前路径一致的目录下即可。

- 使用 open()函数打开这个文件，打开形式是只读方式，文本格式。

  这个类似于 fopen()的 r 这样的参数。open()函数返回一个 bool 类型，如果打开失败，我们在控制台输出一段提示然后程序退出。否则，我们利用

while 循环，将每一行读到的内容输出。

- 可以使用 QFileInfo 获取有关该文件的信息。

  QFileInfo 有很多类型的函数，我们只举出一些例子。比如：

  - isDir() 检查该文件是否是目录（在操作系统这种，文件夹是一种特殊的文件，我们可以通过 QFile 来打开一个文件夹或者路径，这个函数返回值就是判断他是不是一个文件夹或路径）；
  - isExecutable() 检查该文件是否可以被执行。
  - baseName() 可以直接获得文件名；
  - completeBaseName() 获取完整的文件名
  - suffix() 则直接获取文件后缀名。
  - completeSuffix() 获取完整的文件后缀

  我们可以由下面的示例看到，baseName()和 completeBaseName()，以及 suffix()和 completeSuffix()的区别：

```
QFileInfo fi("/tmp/archive.tar.gz");
QString base  = fi.baseName();  // base = "archive"
QString base  = fi.completeBaseName();  // base = "archive.tar"
QString ext   = fi.suffix();  // ext = "gz"
QString ext   = fi.completeSuffix();  // ext = "tar.gz"
```

11.2 二进制文件读写

QDataStream 提供了基于 QIODevice 的二进制数据的序列化。数据流是一种二进制流，这种流**完全不依赖**于底层操作系统、CPU 或者字节顺序（大端或小端）。例如，在安装了 Windows 平台的 PC 上面写入的一个数据流，可以不经过任何处理，直接拿到运行了 Solaris 的 SPARC 机器上读取。由于数据流就是二进制流，因此我们也可以**直接读写没有编码的二进制数据，例如图像、视频、音频**等。

QDataStream 既能够存取 C++ 基本类型，如 int、char、short 等，也可以存取复杂的数据类型，例如自定义的类。实际上，QDataStream 对于类的存储，是将复杂的类分割为很多基本单元实现的。

结合 QIODevice，QDataStream 可以很方便地对文件、网络套接字等进行读写操作。我们从代码开始看起：

```
QFile file("file.dat");

file.open(QIODevice::WriteOnly);
QDataStream out(&file);
out << QString("the answer is");

out << (qint32)42;
```

- 在这段代码中，我们首先打开一个名为 file.dat 的文件（注意，我们为简单起见，并没有检查文件打开是否成功，这在正式程序中是不允许的）。然后，我们将刚刚创建的 file 对象的指针传递给一个 QDataStream 实例 out。类似于 std::cout 标准输出流，QDataStream 也重载了输出重定向<<运算符。后面的代码就很简单了：将“the answer is”和数字 42 输出到数据流。由于我们的 out 对象建立在 file 之上，因此相当于将问题和答案写入 file。

- 需要指出一点：最好使用 Qt 整型来进行读写，比如程序中的 qint32。这保证了在任意平台和任意编译器都能够有相同的行为。

如果你直接运行这段代码，你会得到一个空白的 file.dat，并没有写入任何数据。这是因为我们的 file 没有正常关闭。**为性能起见，数据只有在文件关闭时才会真正写入**。因此，我们必须在最后添加一行代码：

```
file.close(); // 如果不想关闭文件，可以使用 file.flush();
```

接下来我们将存储到文件中的答案取出来

```
QFile file("file.dat");

file.open(QIODevice::ReadOnly);
QDataStream in(&file);
QString str;
qint32 a;

in >> str >> a;
```

唯一需要注意的是，你必须按照写入的顺序，将数据读取出来。顺序颠倒的话，程序行为是不确定的，严重时会直接造成程序崩溃。

那么，既然 QIODevice 提供了 read()、readLine()之类的函数，为什么还要有 QDataStream 呢？QDataStream 同 QIODevice 有什么区别？区别在于，

QDataStream 提供流的形式，性能上一般比直接调用原始 API 更好一些。我们通过下面一段代码看看什么是流的形式：

```
QFile file("file.dat");
file.open(QIODevice::ReadWrite);

QDataStream stream(&file);
QString str = "the answer is 42";


stream << str;
```

11.3 文本文件读写

上一节我们介绍了有关二进制文件的读写。二进制文件比较小巧，却不是人可读的格式。而文本文件是一种人可读的文件。为了操作这种文件，我们需要使用 QTextStream 类。QTextStream 和 QDataStream 的使用类似，只不过它是操作纯文本文件的。

QTextStream 会自动将 Unicode 编码同操作系统的编码进行转换，这一操作对开发人员是透明的。它也会将换行符进行转换，同样不需要自己处理。QTextStream 使用 16 位的 QChar 作为基础的数据存储单位，同样，它也支持 C++ 标准类型，如 int 等。实际上，这是将这种标准类型与字符串进行了相互转换。

QTextStream 同 QDataStream 的使用基本一致，例如下面的代码将把“The answer is 42”写入到 file.txt 文件中：

```
QFile data("file.txt");
if (data.open(QFile::WriteOnly | QIODevice::Truncate))
{
    QTextStream out(&data);
    out << "The answer is " << 42;
}
```

这里，我们在 open()函数中增加了 QIODevice::Truncate 打开方式。我们可以从下表中看到这些打开方式的区别：

枚举值	描述
QIODevice::NotOpen	未打开
QIODevice::ReadOnly	以只读方式打开
QIODevice::WriteOnly	以只写方式打开

- QIODevice::ReadWrite 以读写方式打开
- QIODevice::Append 以追加的方式打开，新增加的内容将被追加到文件末尾
- QIODevice::Truncate 以重写的方式打开，在写入新的数据时会将原有数据全部清除，游标设置在文件开头。
- QIODevice::Text 在读取时，将行结束符转换成 \n；在写入时，将行结束符转换成本地格式，例如 Win32 平台上是 \r\n
- QIODevice::Unbuffered 忽略缓存

我们在这里使用了 QFile::WriteOnly | QIODevice::Truncate，也就是以只写并且覆盖已有内容的形式操作文件。注意，QIODevice::Truncate 会直接将文件内容清空。

虽然 QTextStream 的写入内容与 QDataStream 一致，但是读取时却会有些困难：

```
QFile data("file.txt");
if (data.open(QFile::ReadOnly))
{
    QTextStream in(&data);
    QString str;
    int ans = 0;
    in >> str >> ans;
}
```

在使用 QDataStream 的时候，这样的代码很方便，但是使用了 QTextStream 时却有所不同：读出的时候，str 里面将是 The answer is 42，ans 是 0。这是因为**当使用 QDataStream 写入的时候，实际上会在要写入的内容前面，额外添加一个这段内容的长度值。而以文本形式写入数据，是没有数据之间的分隔的。**因此，使用文本文件时，很少会将其分割开来读取，而是使用诸如使用：

```
QTextStream::readLine();//读取一行
QTextStream::readAll();//读取所有文本
```

这种函数之后再对获得的 QString 对象进行处理。

默认情况下，QTextStream 的编码格式是 Unicode，如果我们需要使用另

外的编码，可以使用：

```
stream.setCodec("UTF-8");
```

这样的函数进行设置。

12 附录

12.1 附录 1：Qt 模块图