目录

一、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艺术级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语言

2012 年 Qt 又被 Digia 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。

当前 Qt 最新版本为 5.13.2（2019.12 之前）

1.3 支持的平台

- 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

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 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。

1.5 Qt 的下载与安装

1.5.1 下载地址

http://www.qt.io/download-open-source/
或者
http://download.qt.io/archive/qt/

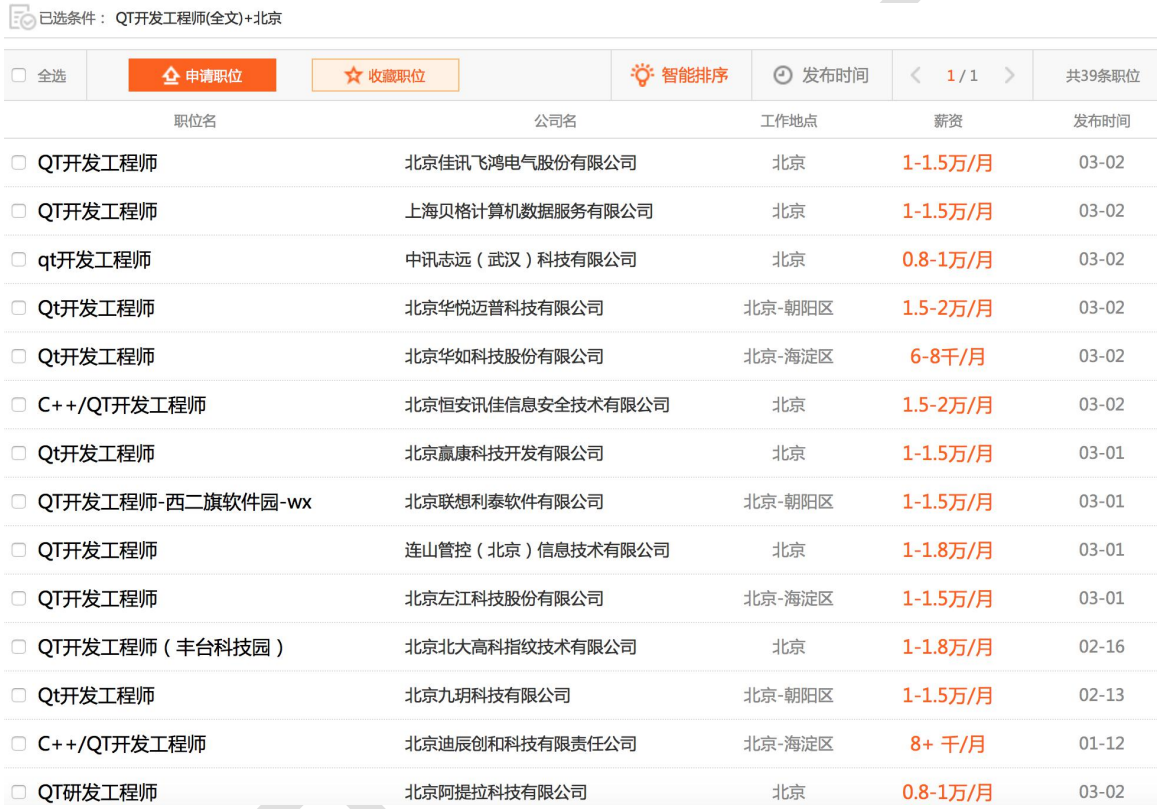
Name	Last modified	Size	Metadata
Parent Directory		-	
submodules/	20-Jan-2017 13:19	-	
single/	20-Jan-2017 13:14	-	
qt-opensource-windows-x86-winrt-msvc2015-5.8.0.exe	20-Jan-2017 12:54	1.2G	Details
qt-opensource-windows-x86-winrt-msvc2013-5.8.0.exe	20-Jan-2017 12:53	1.2G	Details
qt-opensource-windows-x86-msvc2015_64-5.8.0.exe	20-Jan-2017 12:52	1.0G	Details
qt-opensource-windows-x86-msvc2015-5.8.0.exe	20-Jan-2017 12:59	1.0G	Details
qt-opensource-windows-x86-msvc2013_64-5.8.0.exe	20-Jan-2017 12:51	958M	Details
qt-opensource-windows-x86-msvc2013-5.8.0.exe	20-Jan-2017 12:50	947M	Details
qt-opensource-windows-x86-mingw530-5.8.0.exe	20-Jan-2017 12:49	1.2G	Details
qt-opensource-windows-x86-android-5.8.0.exe	20-Jan-2017 12:48	1.3G	Details
qt-opensource-mac-x64-clang-5.8.0.dmg	20-Jan-2017 12:45	1.3G	Details
qt-opensource-mac-x64-android-ios-5.8.0.dmg	20-Jan-2017 12:44	3.4G	Details
qt-opensource-mac-x64-android-5.8.0.dmg	20-Jan-2017 12:40	1.4G	Details
qt-opensource-linux-x64-android-5.8.0.run	20-Jan-2017 12:34	817M	Details
qt-opensource-linux-x64-5.8.0.run	20-Jan-2017 12:34	766M	Details
md5sums.txt	22-Nov-2017 13:16	1.1K	Details
android-patches-5.8-2017_11_16.tar.gz	22-Nov-2017 10:40	4.8K	Details

1.5.2 安装

按照默认安装即可
Qt 对不同的平台提供了不同版本的安装包，可根据实际情况自行下载安装，本文档使用 qt-opensource-windows-x86-mingw530-5.8.0 版本进行讲解（32 位和 64 位均可安装）

1.6 Qt 的优点

- 跨平台，几乎支持所有的平台
- 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。



已选条件： QT开发工程师(全文)+北京

全选 | 申请职位 | 收藏职位 | 智能排序 | 发布时间 | 1 / 1 | 共39条职位

职位名	公司名	工作地点	薪资	发布时间
QT开发工程师	北京佳讯飞鸿电气股份有限公司	北京	1-1.5万/月	03-02
QT开发工程师	上海贝格计算机数据服务有限公司	北京	1-1.5万/月	03-02
qt开发工程师	中讯志远（武汉）科技有限公司	北京	0.8-1万/月	03-02
Qt开发工程师	北京华悦迈普科技有限公司	北京-朝阳区	1.5-2万/月	03-02
Qt开发工程师	北京华如科技股份有限公司	北京-海淀区	6-8千/月	03-02
C++/QT开发工程师	北京恒安讯佳信息安全技术有限公司	北京	1.5-2万/月	03-02
Qt开发工程师	北京赢康科技开发有限公司	北京	1-1.5万/月	03-01
QT开发工程师-西二旗软件园-wx	北京联想利泰软件有限公司	北京-朝阳区	1-1.5万/月	03-01
QT开发工程师	连山管控（北京）信息技术有限公司	北京	1-1.8万/月	03-01
QT开发工程师	北京左江科技股份有限公司	北京-海淀区	1-1.5万/月	03-01
QT开发工程师（丰台科技园）	北京北大高科指纹技术有限公司	北京	1-1.8万/月	02-16
Qt开发工程师	北京九玥科技有限公司	北京-朝阳区	1-1.5万/月	02-13
C++/QT开发工程师	北京迪辰创和科技有限责任公司	北京-海淀区	8+ 千/月	01-12
QT研发工程师	北京阿提拉科技有限公司	北京	0.8-1万/月	03-02

1.7 成功案例

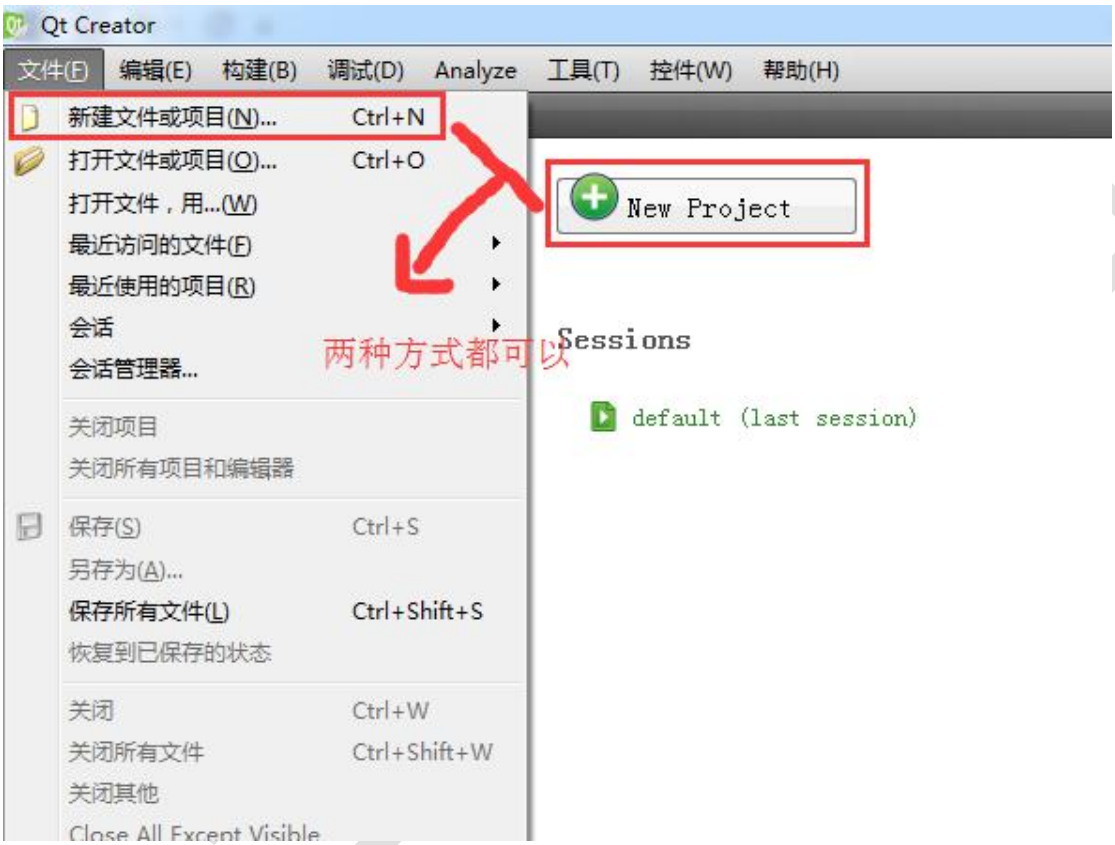
- Linux 桌面环境 KDE
- Skype 网络电话
- Google Earth 谷歌地图
- VLC 多媒体播放器
- VirtualBox 虚拟机软件
- 咪咕音乐
- WPS Office
- 极品飞车

...

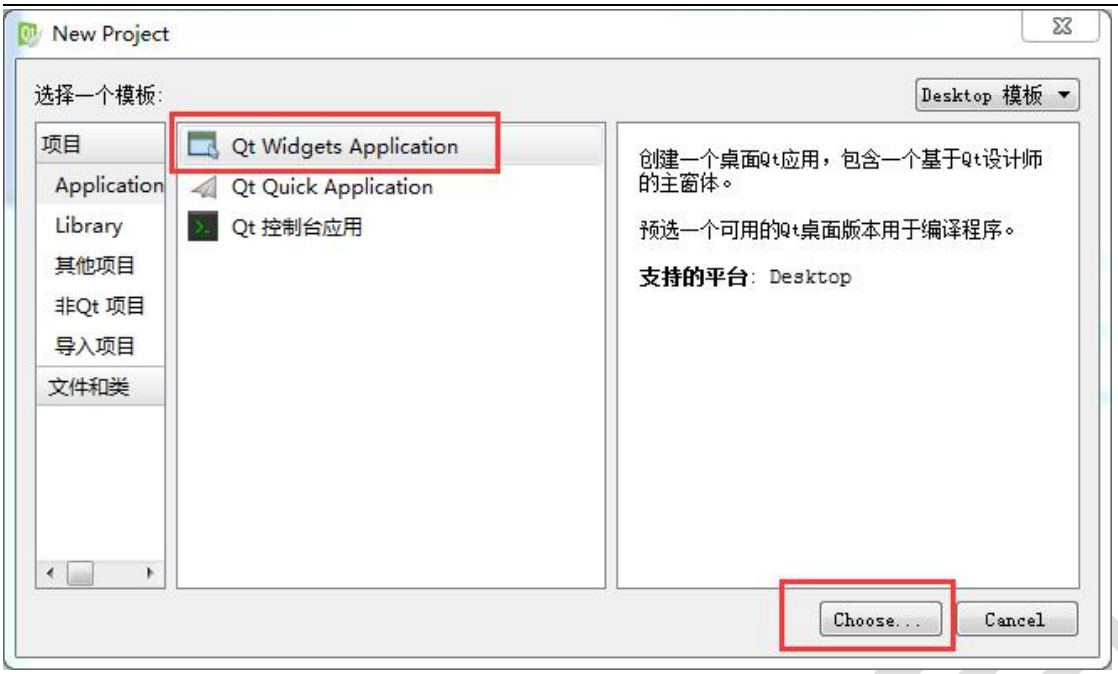
二、创建 Qt 项目

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

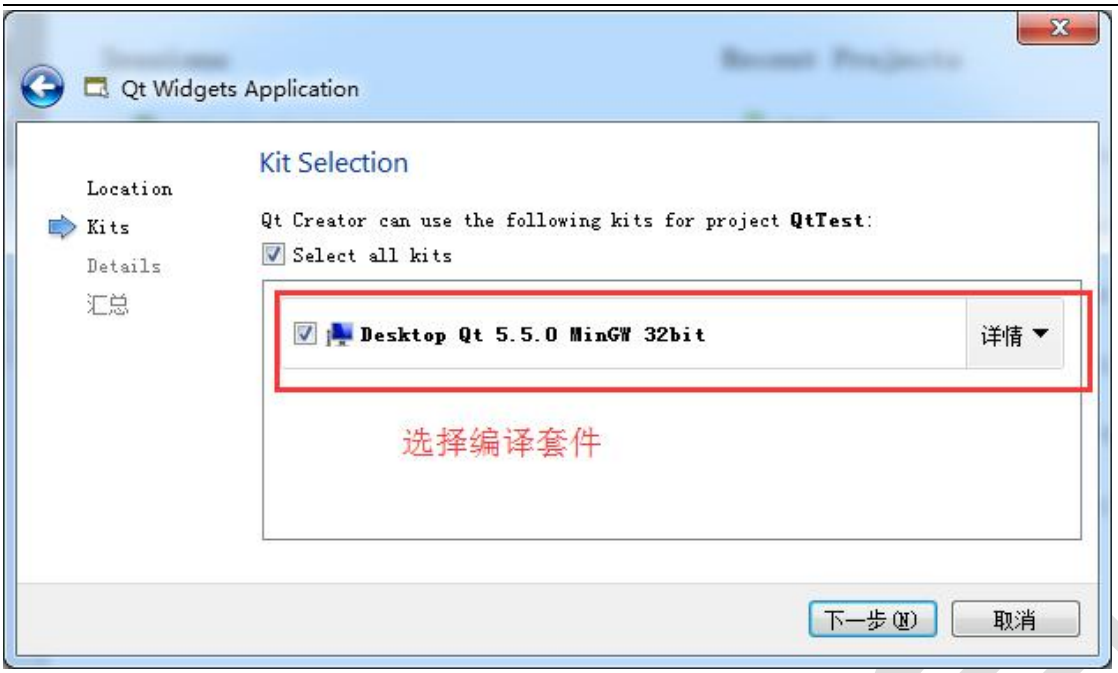


弹出 New Project 对话框，选择 Qt Widgets Application，

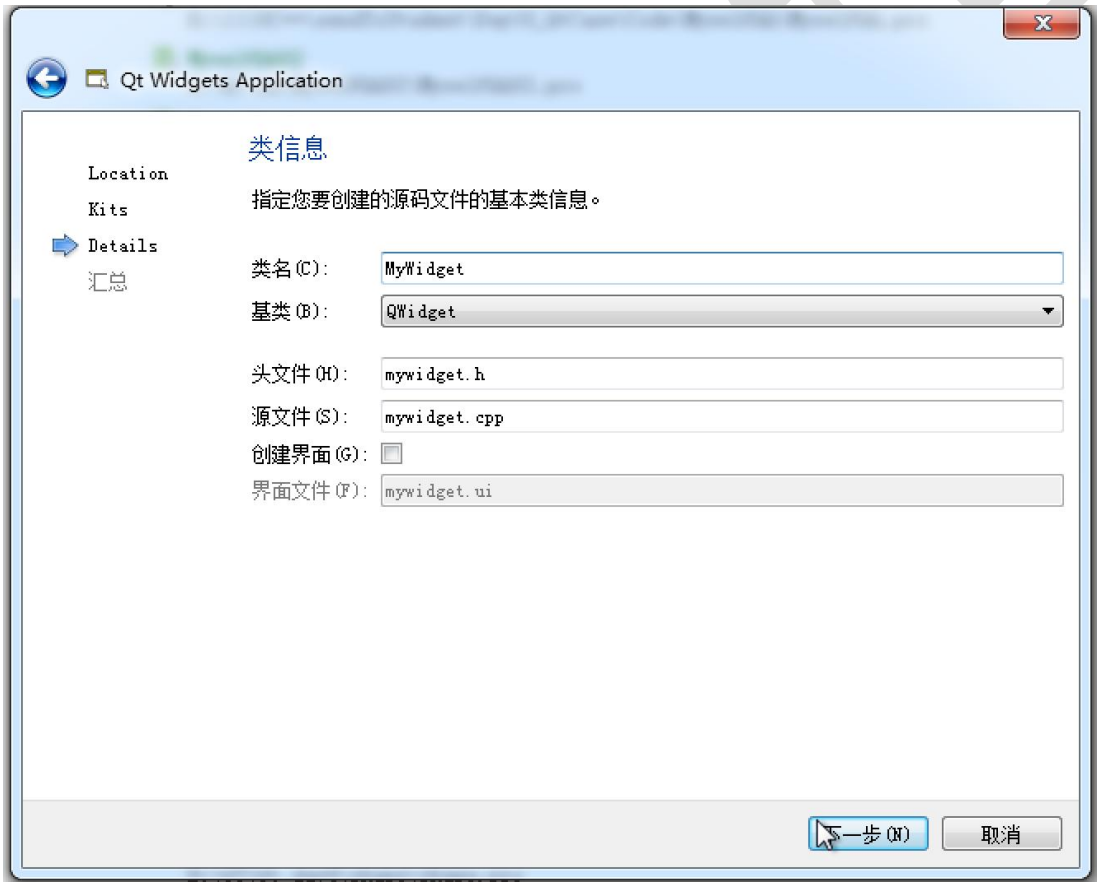


选择【Choose】按钮，弹出如下对话框

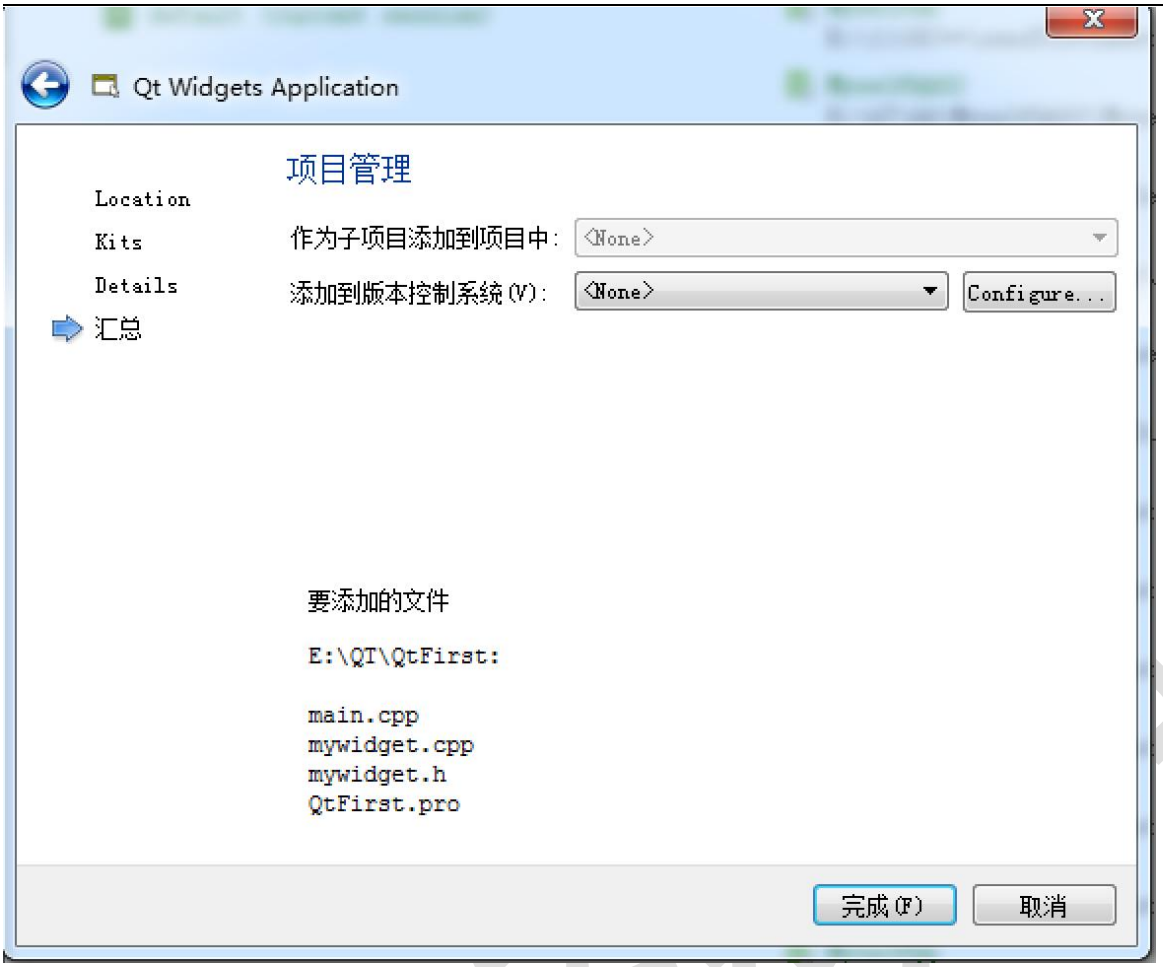
设置项目名称和路径，按照向导进行下一步，



选择编译套件



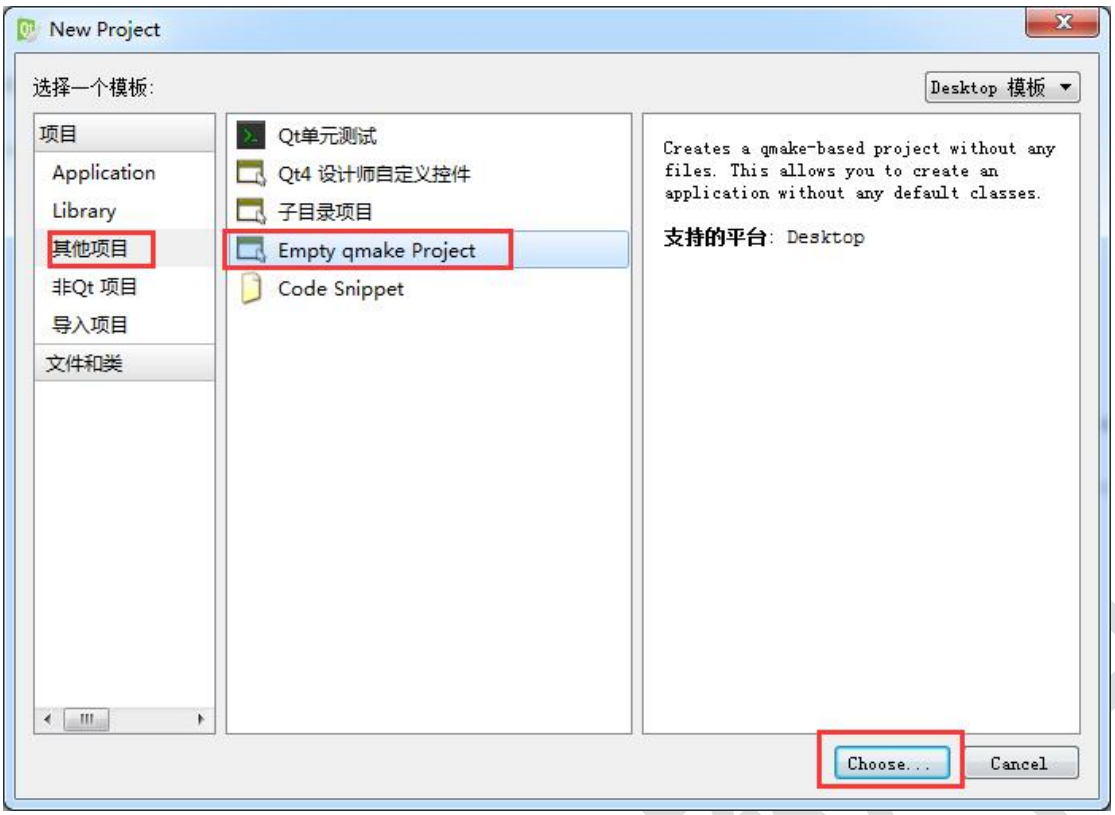
向导会默认添加一个继承自 CMainWindow 的类，可以在此修改类的名字和基类。默认的基类有 QMainWindow、QWidget 以及 QDialog 三个，我们可以选择 QWidget（类似于空窗口），这里我们可以先创建一个不带 UI 的界面，继续下一步



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。

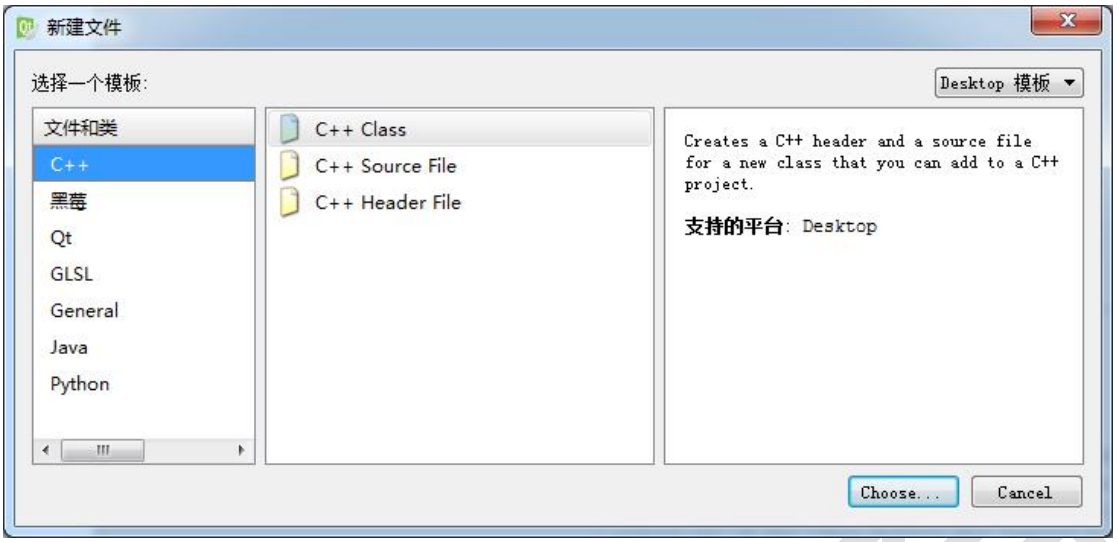
2.2 手动创建

添加一个空项目



选择【choose】进行下一步。设置项目名称和路径 —> 选择编译套件 --> 修改类信息 --> 完成（步骤同上），生成一个空项目。在空项目中添加文件：在项目名称上单击鼠标右键弹出右键菜单，选择【添加新文件】

弹出新建文件对话框



在此对话框中选择要添加的类或者文件，根据向导完成文件的添加。

2.3 .pro 文件

在使用 Qt 向导生成的应用程序.pro 文件格式如下：

```
QT       += core gui  //包含的模块
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets //大于 Qt4 版本 才包含 widget 模块
TARGET = QtFirst  //应用程序名  生成的.exe 程序名称
TEMPLATE = app    //模板类型    应用程序模板
SOURCES += main.cpp\   //源文件
        mywidget.cpp
HEADERS  += mywidget.h   //头文件
```

.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。.pro 文件的写法如下：

- 注释

  从“#”开始，到这一行结束。

- 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

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 一个最简单的 Qt 应用程序

main 入口函数中

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

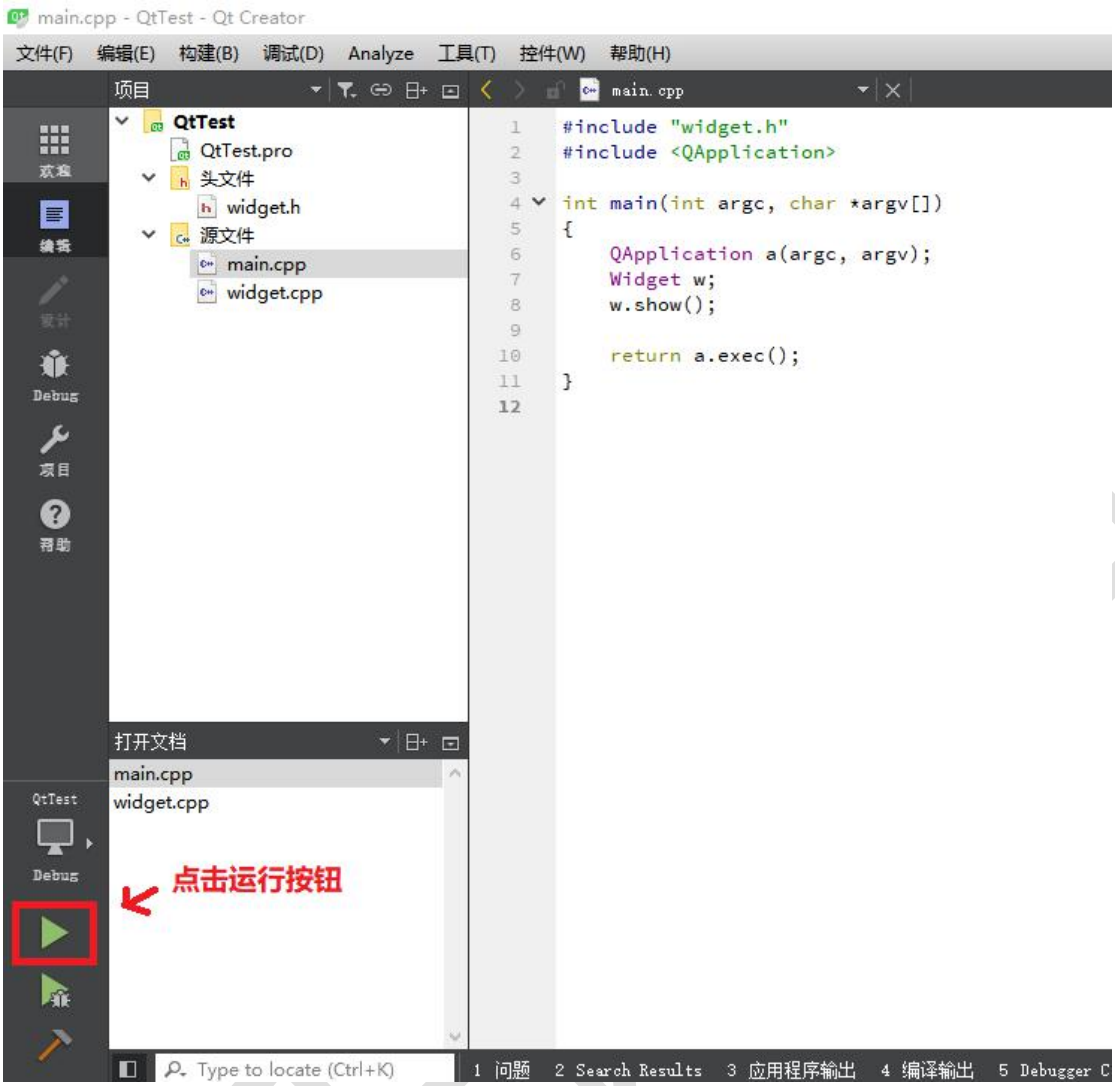
```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    Widget w;
    w.show();

    return a.exec();
}
```

解释：

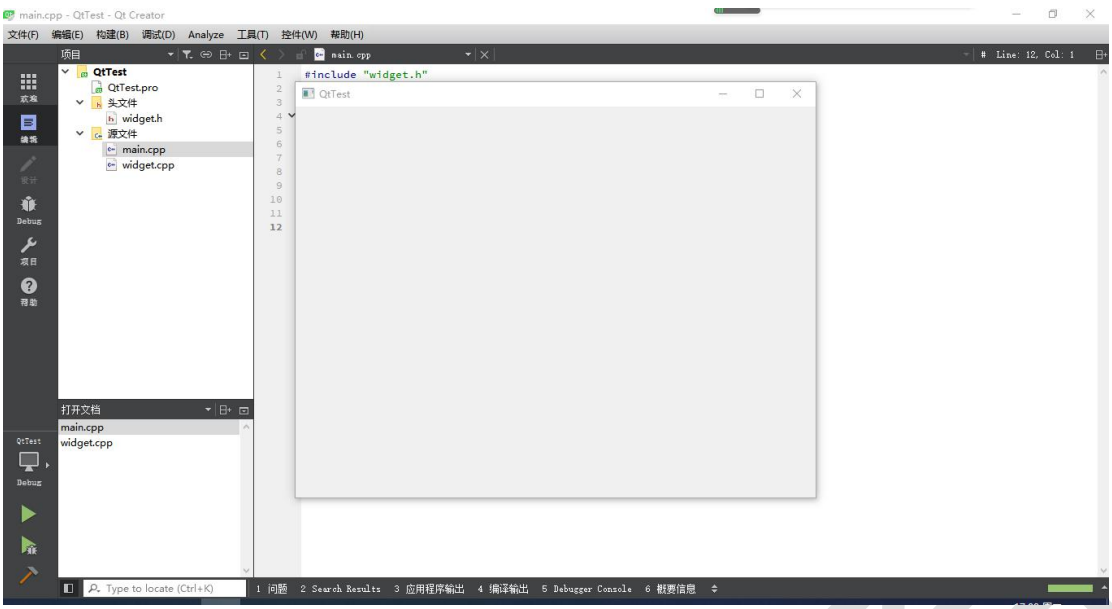
- Qt 系统提供的标准类名声明头文件没有.h 后缀
- Qt 一个类对应一个头文件，类名就是头文件名
- QApplication 应用程序类
  - 管理图形用户界面应用程序的控制流和主要设置。
  - 是 Qt 的整个后台管理的命脉它包含主事件循环，在其中来自窗口系统和其它资源的所有事件处理和调度。它也处理应用程序的初始化和结束，并且提供对话管理。
  - 对于任何一个使用 Qt 的图形用户界面应用程序，都正好存在一个 QApplication 对象，而不论这个应用程序在同一时间内是不是有 0、1、2 或更多个窗口。
- a.exec()

  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。

2.5 运行代码



运行结果，显示基本窗口



三、设置窗口属性

3.1 代码书写位置

通过之前内容见讲解，新创建的 qt 项目工程的文件结构如图所示

那么如果要设置窗口的属性或者添加一些控件，代码应该在哪个文件书写呢？
.pro 文件用于生成可执行文件，.h 存放头文件、类等，widget.cpp 中主要存放.h 文件中函数的实现过程，main.cpp 主要是主程序

main.cpp 文件中有一句代码

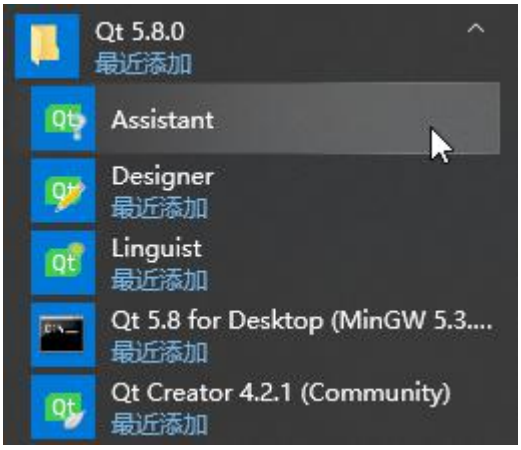
```
Widget w;
```

Widget 类实例化的对象 w，此时会调用构造函数，对如果对当前 Widget 类中的构造函数中写入设置的属性或者添加一些控件，当实例化对象是就会调用构造函数，从而实现想要达到的目的，并且这样做也不会使得主函数中有多余的代码

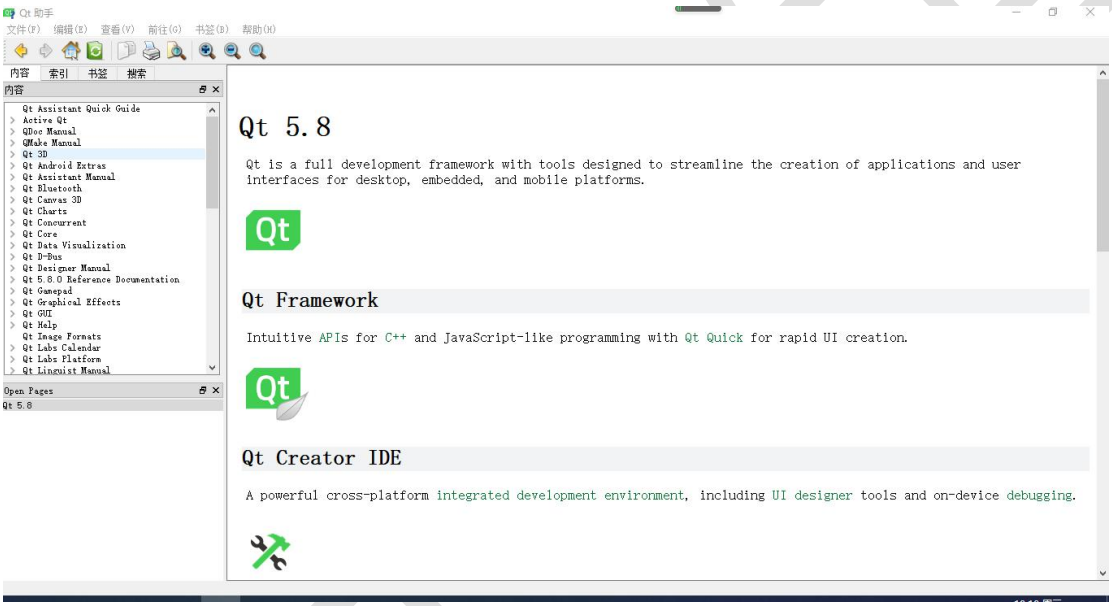
所以一般窗口的属性和添加控件以及对控件的操作都会在类的构造函数中书写

3.2 Qt 助手

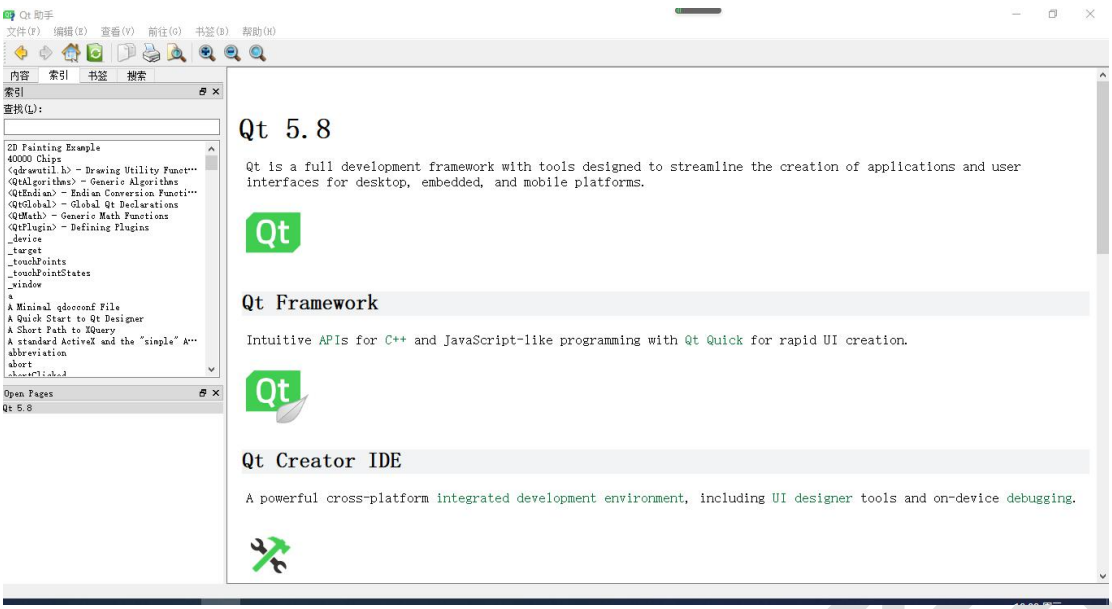
通常在安装 Qt 时都会自带一个 Qt 助手，方便去查询 Qt 中的要使用的类和函数等
一般会在开始菜单的 Qt 目录中，一般叫做 Assistant



点击进入后可以看到所呈现的内容

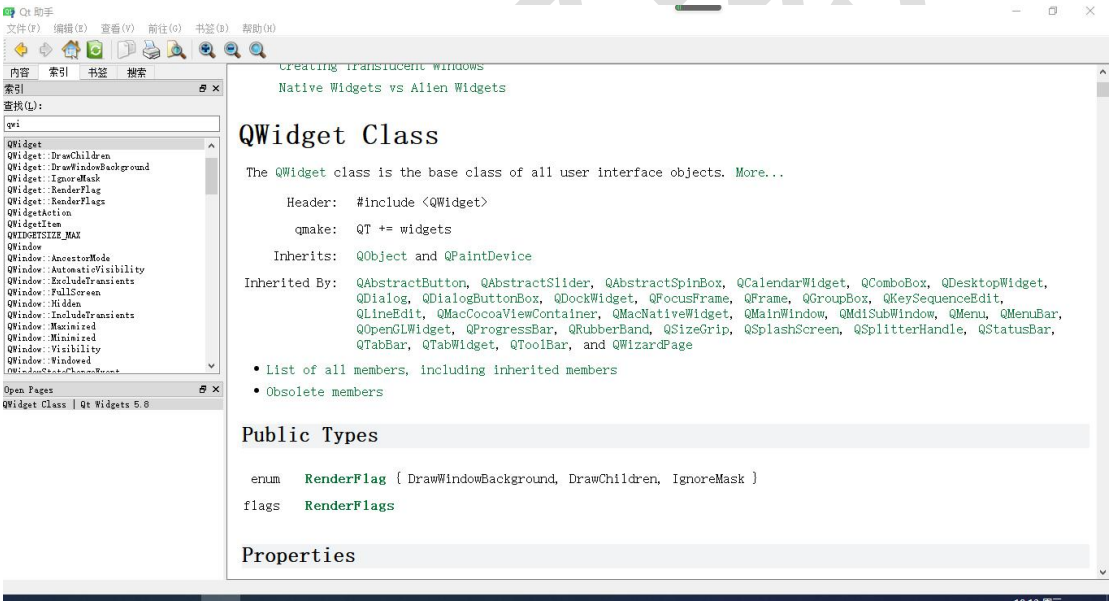


一般点击索引后即可查找相应的内容



3.3 设置窗口属性

由于现在设置的是窗口的属性，Widget 是自定义的类，所以要设置一些默认属性需要在其父类中寻找，所以在 Qt 手册中搜索 QWidget 即可找到一些要寻找的设置属性的函数



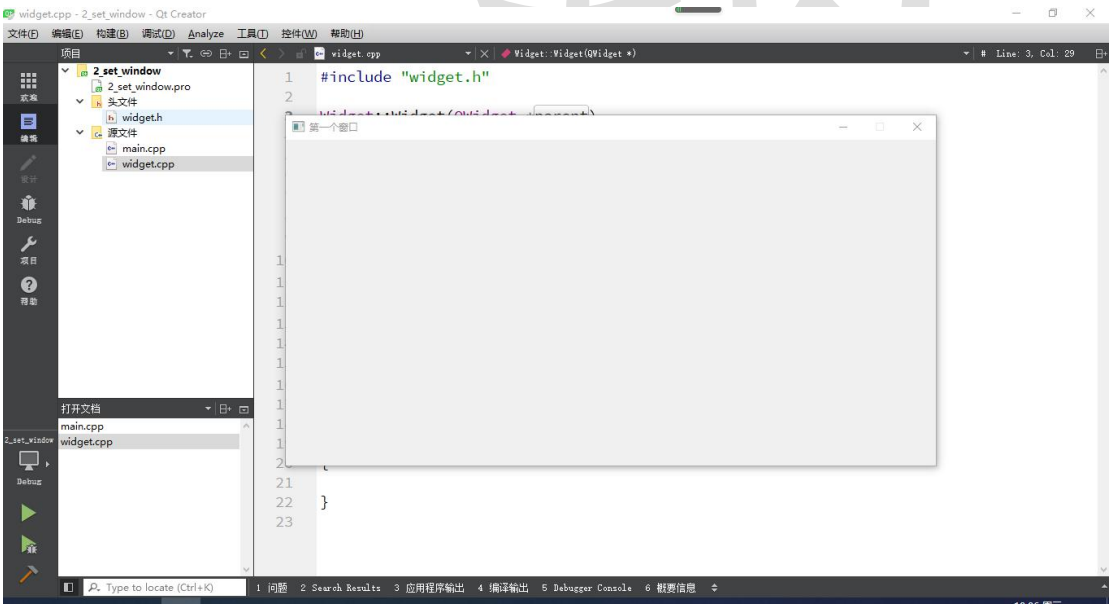
如果当前类中没有所需要的函数，可以从当前类的父类中再去寻找

设置窗口属性代码如下：

```
#include "widget.h"


Widget::Widget(QWidget *parent)
```

```
    : QWidget(parent)
{
    //一般对窗口的信息的设置都放在构造函数当中

    //****************设置窗口属性******************
    //设置窗口标题
    this->setWindowTitle("第一个窗口");

    //设置窗口大小，设置完毕后窗口还可以拉伸
    this->resize(400, 200);

    //设置固定大小，设置完毕后窗口大小无法再改变
    this->setFixedSize(800, 400);
}

Widget::~Widget()
{

}
```

运行结果



四、按钮的创建和属性设置

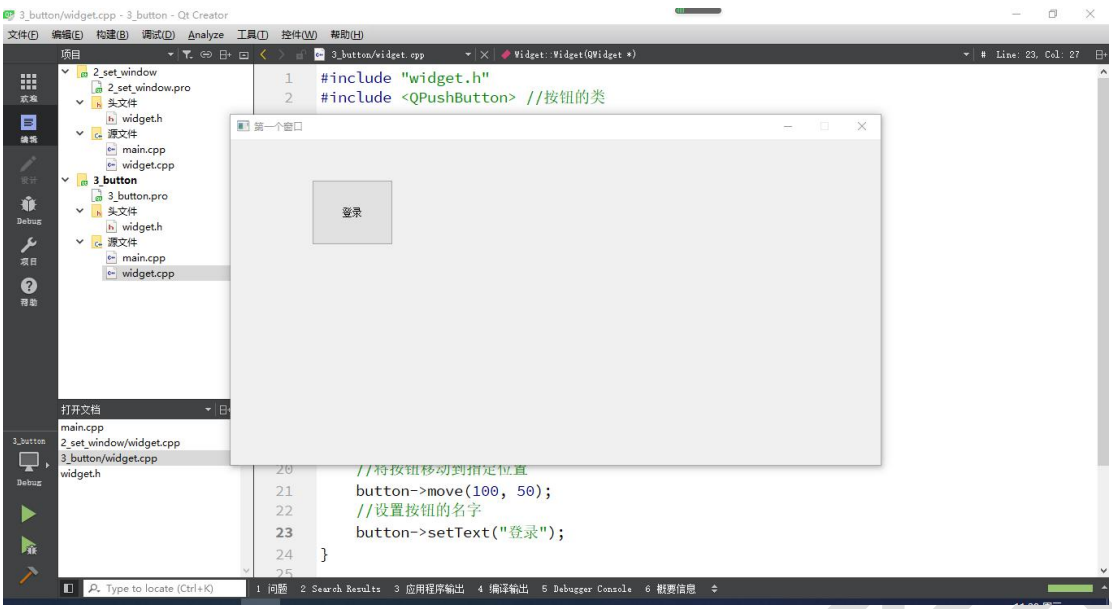
类名：QPushButton
头文件：#include <QPushButton>

4.1 按钮的创建

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 "widget.h"
#include <QPushButton> //按钮的类

Widget::Widget(QWidget *parent)
    : QWidget(parent)
{
    //****************设置窗口属性********************
    this->setWindowTitle("第一个窗口");
    this->resize(400, 200);
    this->setFixedSize(800, 400);

    //****************创建按钮并设置按钮的属性**********
    //创建一个按钮
    //注意：QT 中类实例化对象时，一般都要在堆区开辟空间
    QPushButton *button = new QPushButton;
    //设置按钮的父对象为当前窗口，如果不设置，则无法再当前窗口显示
    button->setParent(this);
    //设置按钮的大小
    button->setFixedSize(100, 80);
    //将按钮移动到指定位置
    button->move(100, 50);
    //设置按钮的名字
    button->setText("登录");
}

Widget::~Widget()
{

}
```

运行结果

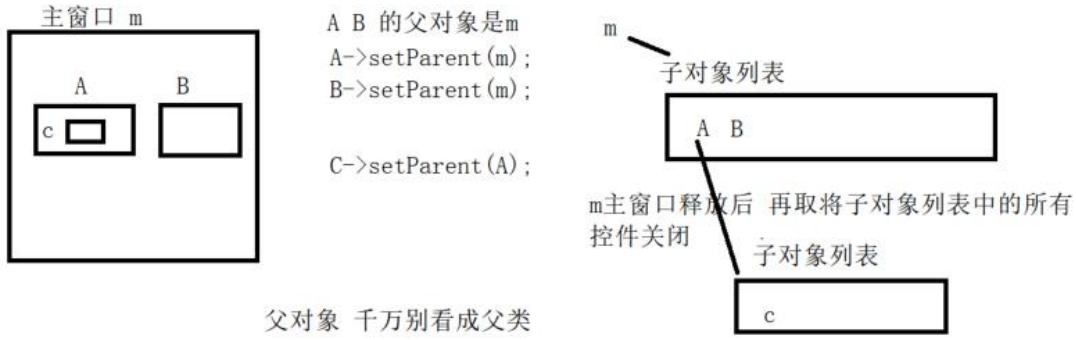


4.2 对象模型（对象树）

在 Qt 中创建对象的时候会提供一个 Parent 对象指针，下面来解释这个 parent 到底是干什么的。

- QObject 是以对象树的形式组织起来的。
  - 当你创建一个 QObject 对象时，会看到 QObject 的构造函数接收一个 QObject 指针作为参数，这个参数就是 parent，也就是父对象指针。

    这相当于，**在创建 QObject 对象时，可以提供一个其父对象，我们创建的这个 QObject 对象会自动添加到其父对象的 children()列表。**
  - **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，这里的父对象并不是继承意义上的父类！）**
- 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**。因此，它会显示在父组件的坐标系统中，被父组件的边界剪裁。例如，当用户关闭一个对话框的时候，应用程序将其删除，那么，我们希望属于这个对话框的按钮、图标等应该一起被删除。事实就是如此，因为这些都是对话框的子组件。
  - 当然，**我们也可以自己删除子对象，它们会自动从其父对象列表中删除。**比如，当我们删除了一个工具栏时，其所在的主窗口会自动将该工具栏从其子对象列表中删除，并且自动调整屏幕显示。



Qt 引入对象树的概念，在一定程度上解决了内存问题。

- 当一个 QObject 对象在堆上创建的时候，Qt 会同时为其创建一个对象树。不过，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 = QPushButton ("退出", &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：

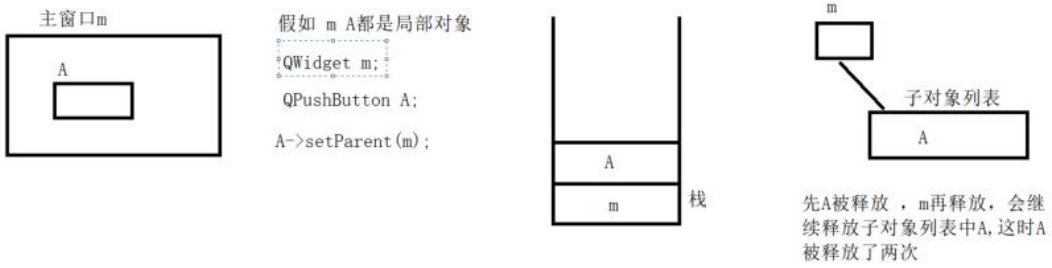
```
{
    QPushButton quit("Quit");
    QWidget window;
    quit.setParent(&window);
}
```

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。

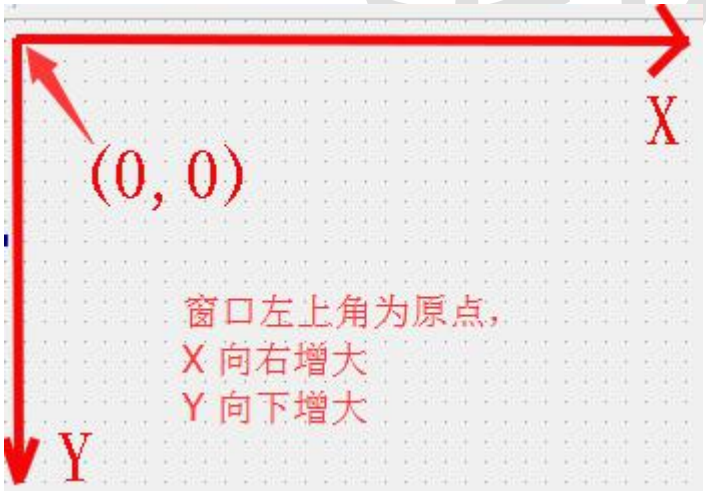
由此我们看到，Qt 的对象树机制虽然帮助我们在一定程度上解决了内存问题，但是也引入了一些值得注意的事情。这些细节在今后的开发过程中很可能时不时跳出来烦扰一下，所以，我们最好从开始就养成良好习惯。

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。



4.3 Qt 窗口坐标体系

坐标体系：
以左上角为原点（0,0），X 向右增加，Y 向下增加。

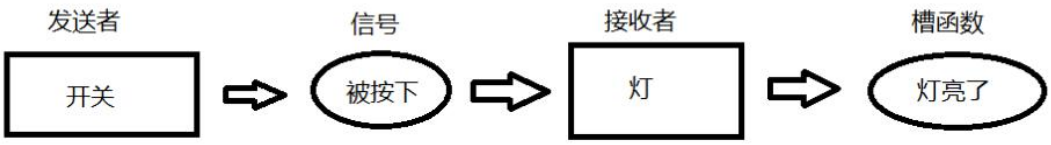


对于嵌套窗口，其坐标是**相对于父窗口**来说的。

五、信号和槽机制

信号槽是 Qt 框架引以为豪的机制之一。所谓信号槽，实际就是观察者模式。**当某个事件发生之后**，比如，按钮检测到自己被点击了一下，**它就会发出一个信号（signal）**。这种发出是没有目的的，

类似广播。**如果有对象对这个信号感兴趣，它就会使用连接（connect）函数**，意思是，**将想要处理的信号和自己的一个函数（称为槽（slot））绑定来处理这个信号**。也就是说，**当信号发出时，被连接的槽函数会自动被回调**。这就类似观察者模式：当发生了感兴趣的事件，某一个操作就会被自动触发。



5.1 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？

其实无法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()函数最常用的一般形式：

```
connect(sender, signal, receiver, slot);
```

参数解释：

- sender：发出信号的对象
- signal：发送对象发出的信号
- receiver：接收信号的对象
- slot：接收对象在接收到信号之后所需要调用的函数（槽函数）

5.2 按钮常用的信号

那么系统自带的信号和槽通常如何查找呢，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。

5.2 自定义信号和槽

使用 connect()可以让我们连接系统提供的信号和槽。但是，Qt 的信号槽机制并不仅仅是使用系统提供的那部分，还会允许我们自己设计自己的信号和槽。

下面我们看看使用 Qt 的信号槽：

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 hungury();
```

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->hungury();
}
```

学生响应了槽函数，并且打印信息

```
//自定义槽函数 实现
void Student::eat()
{
    qDebug() << "该吃饭了！";
}
```

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 “该吃饭了”

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 hungury(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::hungury;
void (Student:: * studentSlot)(QString) = &Student::treat;
connect(teacher,teacherSingal,student,studentSlot);
```

5.3 自定义信号槽需要注意的事项

- 发送者和接收者都需要是 QObject 的子类（当然，槽函数是全局函数、Lambda 表达式等无需接收者的时候除外）；
- 信号和槽函数返回值是 void
- 信号只需要声明，不需要实现
- 槽函数需要声明也需要实现
- 槽函数是普通的成员函数，作为成员函数，会受到 public、private、protected 的影响；
- 使用 emit 在恰当的位置发送信号；
- 使用 connect()函数连接信号和槽。
- 任何成员函数、static 函数、全局函数和 Lambda 表达式都可以作为槽函数
- 信号槽要求信号和槽的参数一致，所谓一致，是参数类型一致。
- 如果信号和槽的参数不一致，允许的情况是，槽函数的参数可以比信号的少，即便如此，槽函数存在的那些参数的顺序也必须和信号的前面几个一致起来。这是因为，你可以在槽函数中选择忽略信号传来的数据（也就是槽函数的参数比信号的少）。

5.4 信号槽的拓展

- 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，但是它们的调用顺序是不确定的。

- 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。

- 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。

- 槽可以被取消链接

  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。

- 信号槽可以断开

  利用 disconnect 关键字是可以断开信号槽的

- 使用 Lambda 表达式

  在使用 Qt 5 的时候，能够支持 Qt 5 的编译器都是支持 Lambda 表达式的。

  在连接信号和槽的时候，槽函数可以使用 Lambda 表达式的方式进行处理。后面我们会详细介绍什么是 Lambda 表达式

5.5 Qt4 版本的信号槽写法

```
connect(zt,SIGNAL(hungry(QString)),st,SLOT(treat(QString)));
```

这里使用了 SIGNAL 和 SLOT 这两个宏，将两个函数名转换成了字符串。注意到 connect()函数的 signal 和 slot 都是接受字符串，一旦出现连接不成功的情况，Qt4 是没有编译错误的（因为一切都是字符串，编译期是不检查字符串是否匹配），而是在运行时给出错误。这无疑会增加程序的不稳定性。

Qt5 在语法上完全兼容 Qt4，而反之是不可以的。

5.6 Lambda 表达式

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

```
[capture](parameters) mutable ->return-type
{
statement
}
```

[函数对象参数](操作符重载函数参数)mutable ->返回值{函数体}

① 函数对象参数；

[]，标识一个 Lambda 的开始，这部分必须存在，不能省略。函数对象参数是传递给编译器自动生成的函数对象类的构造函数的。函数对象参数只能使用那些到定义 Lambda 为止时 Lambda 所在作用范围内可见的局部变量（包括 Lambda 所在类的 this）。函数对象参数有以下形式：

- 空。没有使用任何函数对象参数。
- =。函数体内可以使用 Lambda 所在作用范围内所有可见的局部变量（包括 Lambda 所在类的 this），并且是值传递方式（相当于编译器自动为我们按值传递了所有局部变量）。
- &。函数体内可以使用 Lambda 所在作用范围内所有可见的局部变量（包括 Lambda 所在类的 this），并且是引用传递方式（相当于编译器自动为我们按引用传递了所有局部变量）。
- this。函数体内可以使用 Lambda 所在类中的成员变量。
- a。将 a 按值进行传递。按值进行传递时，函数体内不能修改传递进来的 a 的拷贝，因为默认情况下函数是 const 的。**要修改传递进来的 a 的拷贝**，可以添加 mutable 修饰符。
- &a。将 a 按引用进行传递。
- a，&b。将 a 按值进行传递，b 按引用进行传递。
- =，&a，&b。除 a 和 b 按引用进行传递外，其他参数都按值进行传递。
- &，a，b。除 a 和 b 按值进行传递外，其他参数都按引用进行传递。

② 操作符重载函数参数；

标识重载的()操作符的参数，没有参数时，这部分可以省略。参数可以通过按值（如：(a,b)）和按引用（如：(&a,&b)）两种方式进行传递。

③ 可修改标示符；

mutable 声明，这部分可以省略。按值传递函数对象参数时，加上 mutable 修饰符后，可以修改按值传递进来的拷贝（注意是能修改拷贝，而不是值本身）。

```
QPushButton * myBtn = new QPushButton (this);
QPushButton * myBtn2 = new QPushButton (this);
myBtn2->move(100,100);
int m = 10;

connect(myBtn,&QPushButton::clicked,this,[m] ()mutable { m = 100 + 10; qDebug() << m; });

connect(myBtn2,&QPushButton::clicked,this,[=] ()    { qDebug() << m; });

qDebug() << m;
```

④ 函数返回值；

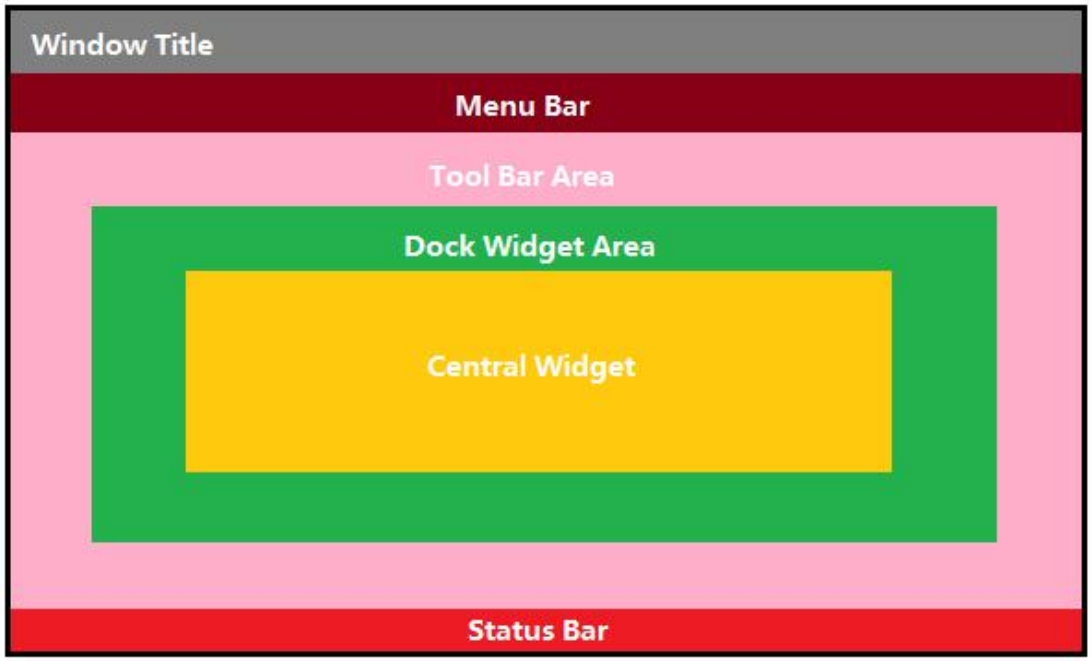
->返回值类型，标识函数返回值的类型，当返回值为 void，或者函数体中只有一处 return 的地方（此时编译器可以自动推断出返回值类型）时，这部分可以省略。

⑤ 是函数体；

{}，标识函数的实现，这部分不能省略，但函数体可以为空。

五、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)、多个铆接部件(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。


Window Title
Menu Bar
Tool Bar Area
Dock Widget Area
Central Widget
Status Bar

5.1 菜单栏


Qt Creator
文件(F)
编辑(E)
构建(B)
调试(D)
Analyze
工具(T)
控件(W)
帮助(H)
菜单栏
新建文件或项目(N)... Ctrl+N
打开文件或项目(O)... Ctrl+O
打开文件，用...(W)
最近访问的文件(F)
最近使用的项目(R)
Sessions
会话管理器...
菜单项
关闭项目
关闭所有项目和编辑器
Save Ctrl+S
Save As...
保存所有文件(L) Ctrl+Shift+S
恢复到已保存的状态
关闭 Ctrl+W
关闭所有文件 Ctrl+Shift+W
关闭其他
Close All Except Visible
打印(P)... Ctrl+P
退出(X) Ctrl+Q
菜单
<no document>

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。

- 创建菜单栏，通过 QMainWindow 类的 menubar（）函数获取主窗口菜单栏指针

```
创建菜单栏：
    #include <QMenuBar>
    QMenuBar -->  QMenuBar(QWidget *parent = Q_NULLPTR)
添加菜单栏：
    QMainWindow -->  void setMenuBar(QMenuBar *menuBar)
```

- 创建菜单，调用 QMenu 的成员函数 addMenu 来添加菜单

```
创建菜单：
    #include <QMenu>
    QMenu -->  QMenu(const QString &title, QWidget *parent = Q_NULLPTR)
添加菜单：
    MenuBar -->  QAction *addMenu(QMenu *menu)
```

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

```
创建菜单项：
    #include <QAction>
    QAction -->  QAction(const QString &text, QObject *parent = nullptr)
添加菜单项：
    QMenu -->  addAction(const QAction *action)
```

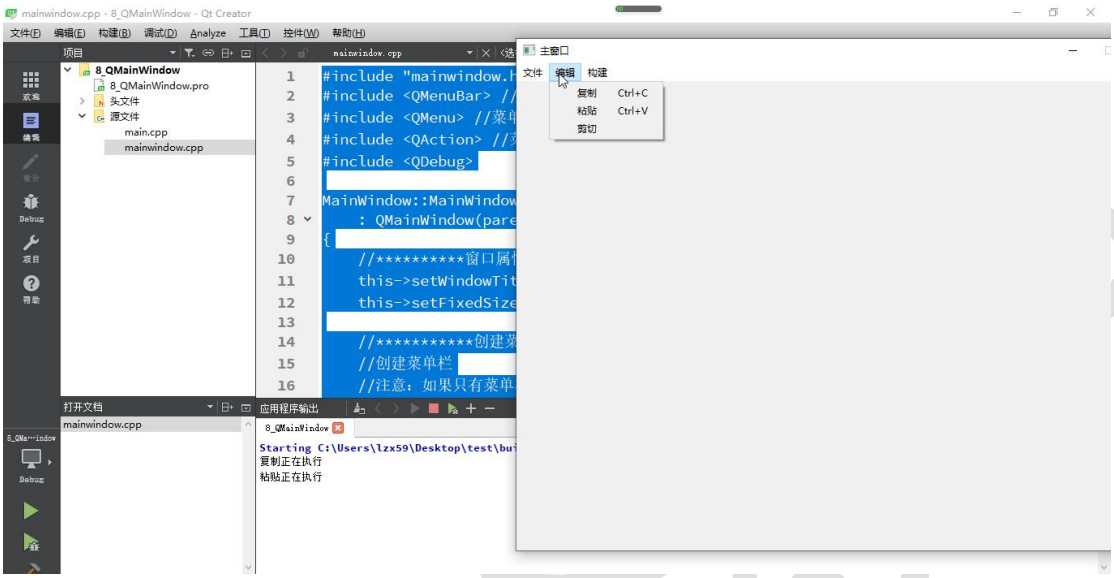
Qt 并没有专门的菜单项类，只是使用一个 QAction 类，抽象出公共的动作。当我们把 QAction 对象添加到菜单，就显示成一个菜单项，添加到工具栏，就显示成一个工具按钮。用户可以通过点击菜单项、点击工具栏按钮、点击快捷键来激活这个动作。

```
#include "mainwindow.h"
#include <QMenuBar> //菜单栏
#include <QMenu> //菜单
#include <QAction> //菜单项
#include <QDebug>

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
{
    //**********窗口属性设置****************
    this->setWindowTitle("主窗口");
    this->setFixedSize(800, 600);

    //***********创建菜单栏*****************
    //创建菜单栏
    //注意：如果只有菜单栏，则在窗口处没有现象
    QMenuBar *menu_bar = new QMenuBar(this);
    this->setMenuBar(menu_bar);
```

```
//创建菜单
QMenu *menu1 = new QMenu("文件", this);
menu_bar->addMenu(menu1);

QMenu *menu2 = new QMenu("编辑", this);
menu_bar->addMenu(menu2);

QMenu *menu3 = new QMenu("构建", this);
menu_bar->addMenu(menu3);

//创建菜单项
QAction *act1 = new QAction("新建文件或项目", this);
menu1->addAction(act1);

QAction *act2 = new QAction("打开文件或项目", this);
menu1->addAction(act2);

//添加分割线
menu1->addSeparator();

QAction *act3 = new QAction("保存", this);
menu1->addAction(act3);

QAction *act4 = new QAction("另存为", this);
menu1->addAction(act4);

QAction *act5 = new QAction("复制", this);
act5->setShortcut(QKeySequence(Qt::CTRL + Qt::Key_C));
menu2->addAction(act5);

QAction *act6 = new QAction("粘贴", this);
act6->setShortcut(QKeySequence(tr("Ctrl+V")));
menu2->addAction(act6);

QAction *act7 = new QAction("剪切", this);
menu2->addAction(act7);

//当单击菜单项时，做出相应的处理
connect(act5, &QAction::triggered, [=]{
    qDebug()<<"复制正在执行";
});

connect(act6, &QAction::triggered, [=]{
```

```
        qDebug()<<"粘贴正在执行";
    });
}

MainWindow::~MainWindow()
{

}
```

运行结果



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。、

创建工具栏：

```
#include <QToolBar>
QToolBar --> QToolBar(QWidget *parent = Q_NULLPTR)
```

添加工具栏：

```
QMainWindow -->
    void addToolBar(QToolBar *toolbar)
    void addToolBar(Qt::ToolBarArea area, QToolBar *toolbar)
        Qt::LeftToolBarArea 左边显示
        Qt::RightToolBarArea 右边显示
```

Qt::TopToolBarArea 上边显示

Qt::BottomToolBarArea 下边显示

- 直接调用 QMainWindow 类的 addToolBar（）函数获取主窗口的工具条对象，每增加一个工具条都需要调用一次该函数。
- 插入属于工具条的动作，即在工具条上添加操作。

  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：
  - 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 setMoveable（）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

```
//创建工具栏
//注意：工具栏可以添加多个
QToolBar *toolbar = new QToolBar(this);
//将工具栏添加到窗口
//this->addToolBar(toolbar);//默认在最上面显示
this->addToolBar(Qt::LeftToolBarArea, toolbar); //设置默认在左边显示

//工具栏添加菜单项和分割线
QAction *act_tool1 = new QAction("欢迎", this);
QAction *act_tool2 = new QAction("编辑", this);
toolbar->addAction(act_tool1);
toolbar->addSeparator();
toolbar->addAction(act_tool2);

//设置工具栏的浮动状态，true：可以悬浮在窗口 false：不可以
toolbar->setFloatable(false);
//设置运行工具栏的位置，设置为左边或者右边
toolbar->setAllowedAreas(Qt::LeftToolBarArea | Qt::RightToolBarArea);
```

5.3 状态栏

- 派生自 QWidget 类，使用方法与 QWidget 类似，QStatusBar 类常用成员函数：
- 状态栏也只能最多有一个

创建状态栏：
QStatusBar -->
将控件添加到左边栏
void addWidget(QWidget *widget, int stretch = 0)
将控件添加到右边栏
void addPermanentWidget(QWidget *widget, int stretch = 0)
添加状态栏：
QMainWindow --> void setStatusBar(QStatusBar *statusbar)

```
//创建状态栏
//状态栏只能有一个
QStatusBar *statusbar = new QStatusBar(this);
//添加状态栏
this->setStatusBar(statusbar);

//给状态栏中添加文字信息或者按钮=
QLabel *label1 = new QLabel("左边信息", this);
statusbar->addWidget(label1);

QLabel *label2 = new QLabel("右边信息", this);
statusbar->addPermanentWidget(label2);

QPushButton *button = new QPushButton("设置", this);
statusbar->addWidget(button);
```

5.4 铆接部件

铆接部件

QDockWidget，也称浮动窗口，可以有多个。

创建铆接部件：
QDockWidget -->
QDockWidget(const QString &title, QWidget *parent = Q_NULLPTR)
添加铆接部件：
QMainWindow -->
void addDockWidget(Qt::DockWidgetArea area, QDockWidget *dockwidget)
Qt::LeftDockWidgetArea 左边
Qt::RightDockWidgetArea 右边
Qt::TopDockWidgetArea 上边
Qt::BottomDockWidgetArea 下边

```
//创建铆接部件
QDockWidget *dockwidget = new QDockWidget("这是一个铆接部件", this);
this->addDockWidget(Qt::TopDockWidgetArea, dockwidget);
```

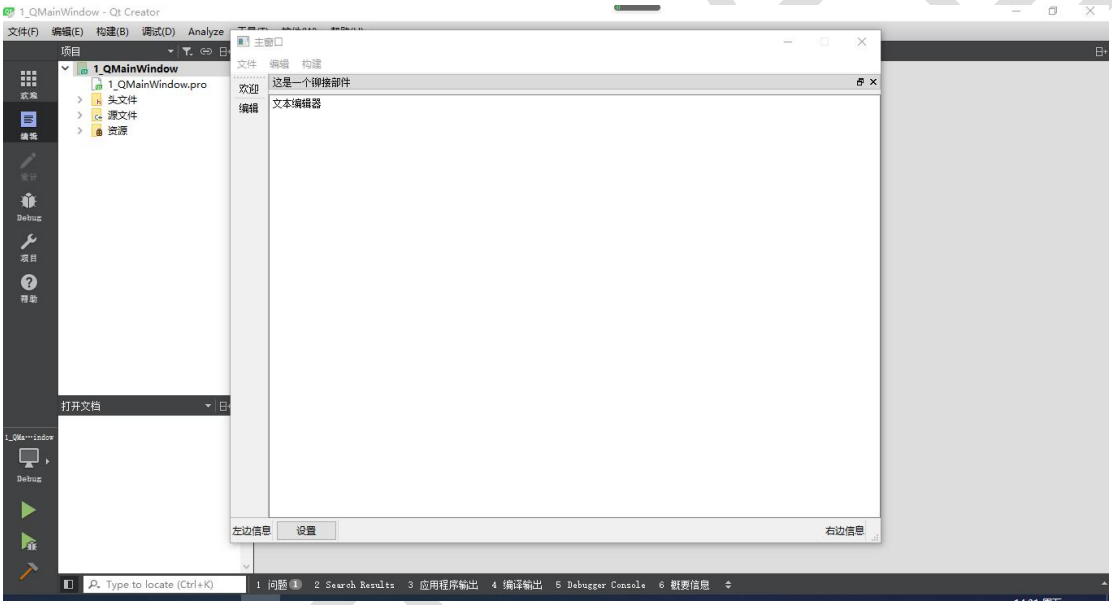
5.5 核心部件（中心部件）

除了以上几个部件，中心显示的部件都可以作为核心部件，例如一个记事本文件，可以利用 QTextEdit 做核心部件

```
添加中心部件：
    QMainWindow --> void setCentralWidget(QWidget *widget)
```

```
QTextEdit *edit = new QTextEdit("文本编辑器", this);
this->setCentralWidget(edit);
```

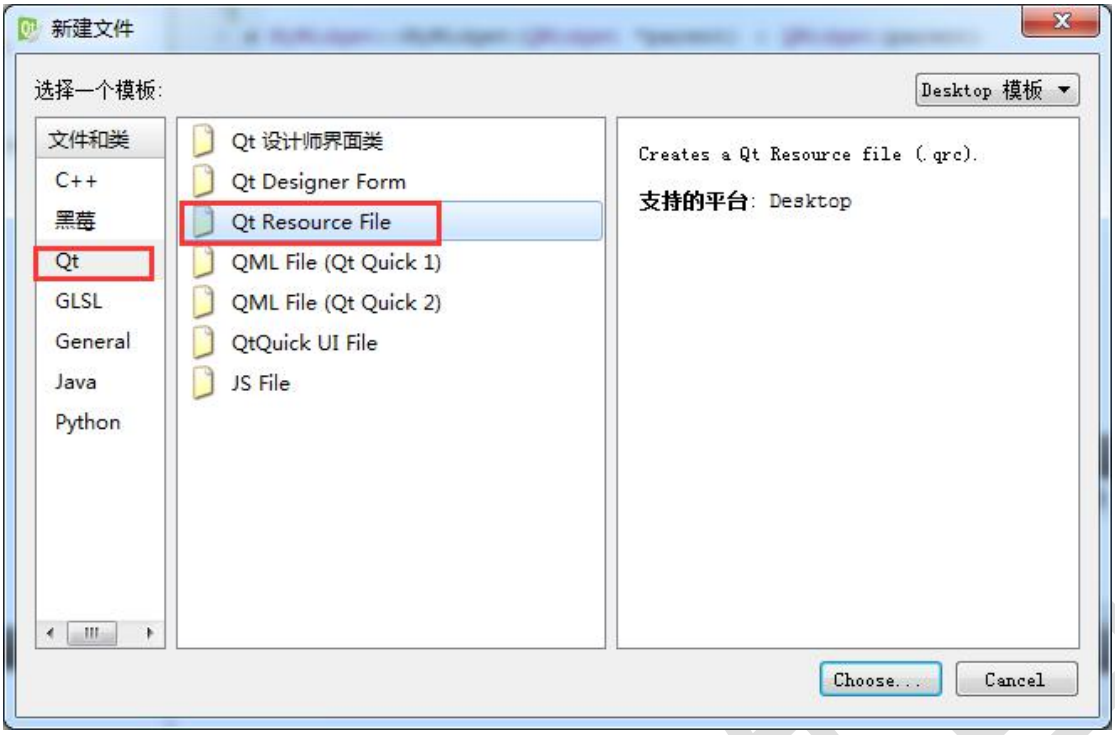
最终执行结果



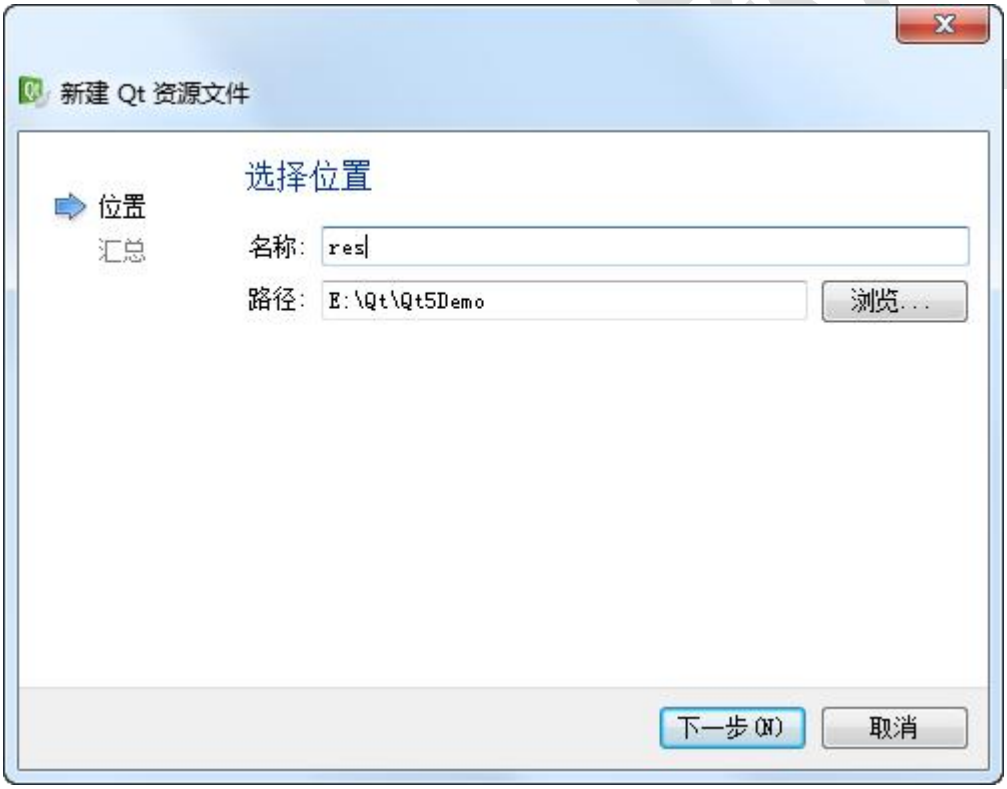
5.6 资源文件

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。

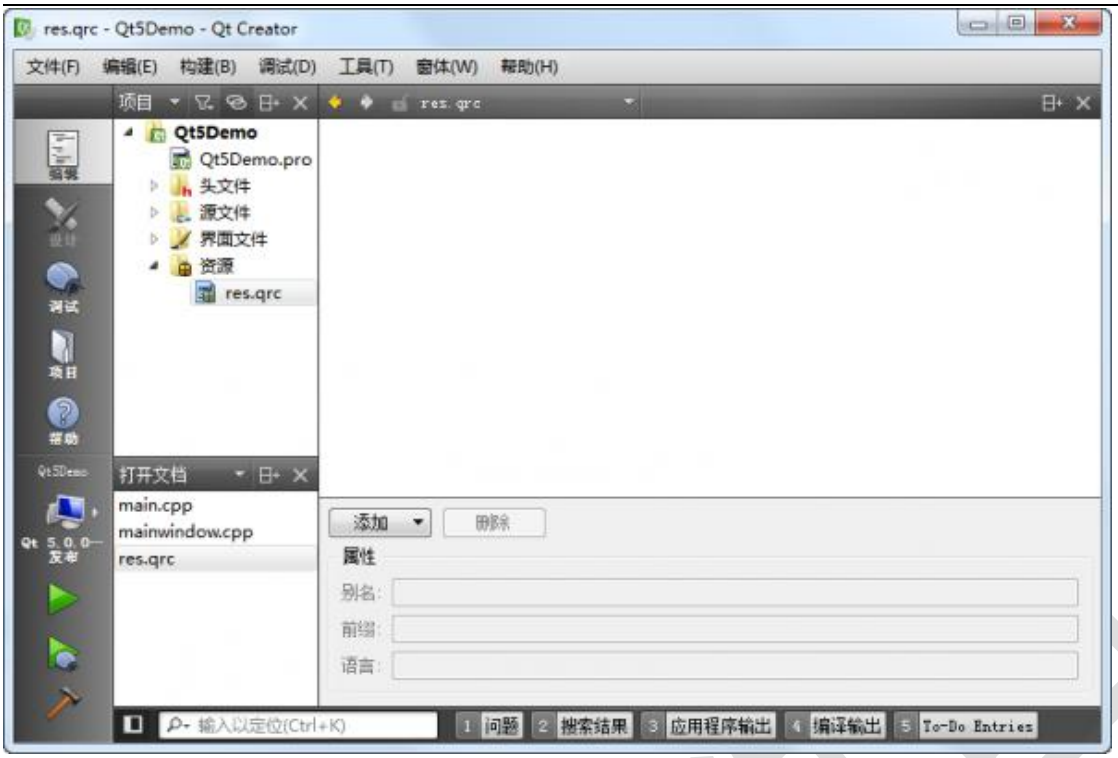
使用 Qt Creator 可以很方便地创建资源文件。我们可以在工程上点右键，选择“添加新文件…”，可以在 Qt 分类下找到“Qt 资源文件”：



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：

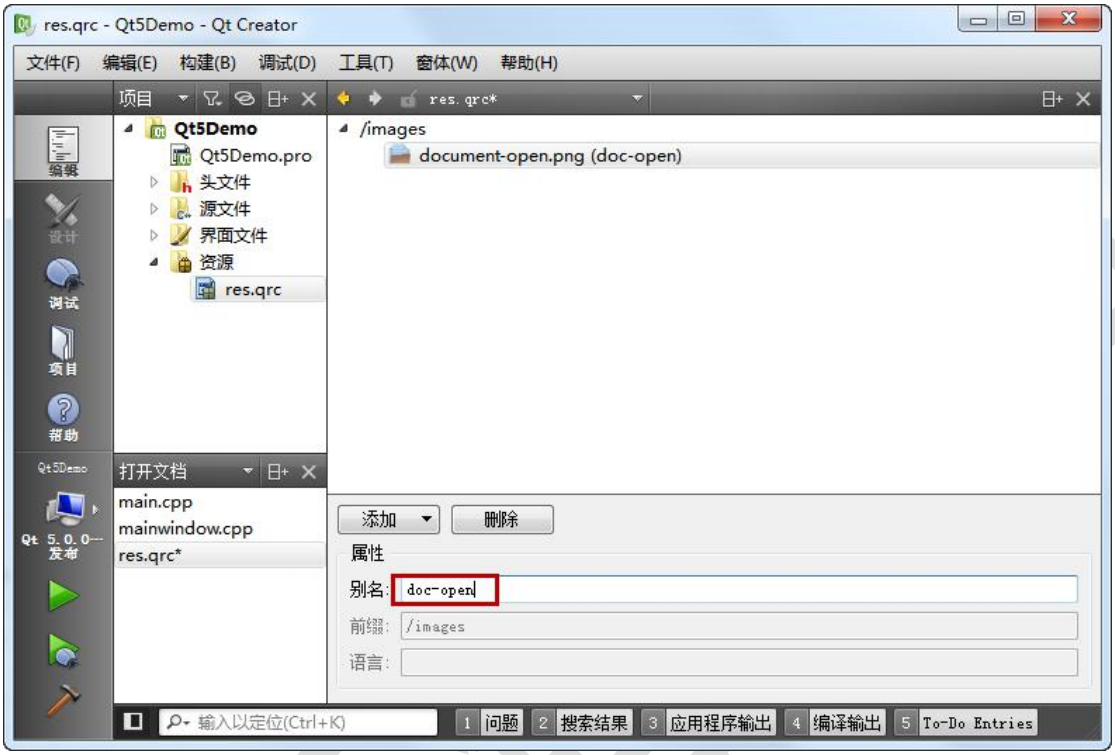


右侧的编辑区有个“添加”，我们首先需要添加前缀，比如我们将前缀取名为 images。然后选中这个前缀，继续点击添加文件，可以找到我们所需添加的文件。这里，我们选择 document-open.png 文件。当我们完成操作之后，Qt Creator 应该是这样子的：

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。

如果我们使用文本编辑器打开 res.qrc 文件，就会看到一下内容：

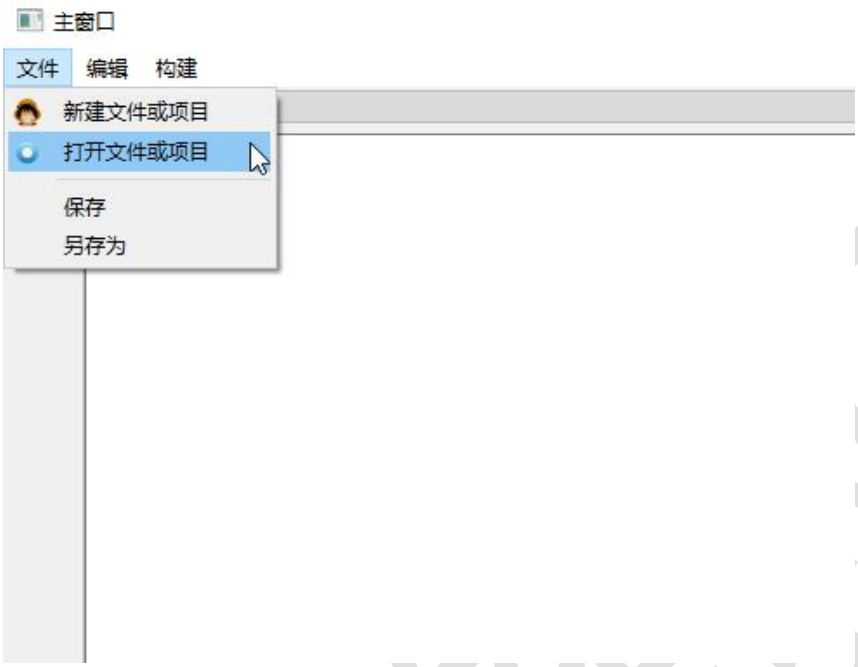
```
<RCC>
    <qresource prefix="/images">
        <file alias="doc-open">document-open.png</file>
    </qresource>
    <qresource prefix="/images/fr" lang="fr">
        <file alias="doc-open">document-open-fr.png</file>
    </qresource>
</RCC>
```

我们可以对比一下，看看 Qt Creator 帮我们生成的是怎样的 qrc 文件。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。

```
//在指定的菜单项上设置图片
//创建图片控件
QPixmap pix;
```

```
//选择图片
pix.load(":/image/Luffy.png");
//给菜单项设置图片
act1->setIcon(QIcon(pix));

pix.load(":/image/Start.png");
act2->setIcon(QIcon(pix));
```

执行结果



六、对话框 QDialog

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设置。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。就像主窗口一样，我们通常会设计一个类继承 QDialog。QDialog（及其子类，以及所有 Qt::Dialog 类型的类）的对于其 parent 指针都有额外的解释：如果 parent 为 NULL，则该对话框会作为一个顶层窗口，否则则作为其父组件的子对话框（此时，其默认出现的位置是 parent 的中心）。顶层窗口与非顶层窗口的区别在于，顶层窗口在任务栏会有自己的位置，

而非顶层窗口则会共享其父组件的位置。

对话框分为模态对话框和非模态对话框。

- 模态对话框，就是会阻塞同一应用程序中其它窗口的输入。

  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。

6.2 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：

- QColorDialog: 选择颜色；
- QFileDialog: 选择文件或者目录；
- QFontDialog: 选择字体；
- QInputDialog: 允许用户输入一个值，并将其值返回；
- QMessageBox: 模态对话框，用于显示信息、询问问题等；
- 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 自定义消息框

Qt 支持模态对话框和非模态对话框。

模态与非模态的实现：

- 使用 QDialog::exec()实现应用程序级别的模态对话框
- 使用 QDialog::open()实现窗口级别的模态对话框
- 使用 QDialog::show()实现非模态对话框。

6.3.1 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。
  - 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。

  一般默认是应用程序级别的模态。

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!"));
dialog.exec();
```

6.3.2 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 的对象系统吗？

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。

6.4 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：

- 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。

- 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());
```

- 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。

- 与 QMessageBox::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)
```

- 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)
```

- 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)
```

我们可以通过下面的代码来演示下如何使用 QMessageBox。

```
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!"));
}
```

我们使用 QMessageBox::question()来询问一个问题。

- 这个对话框的父窗口是 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。

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。

七、布局管理器

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。这就涉及到组件定位的机制。

Qt 提供了两种组件定位机制：绝对定位和布局定位。

- 绝对定位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，采用绝对定位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。

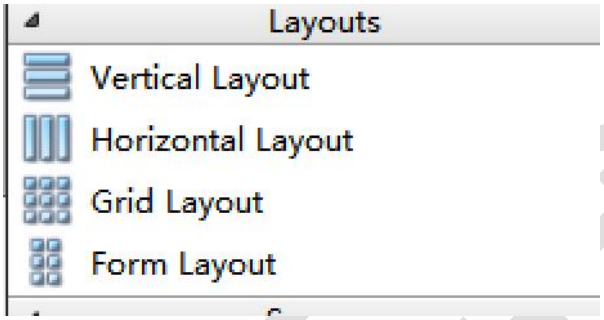
- 布局定位：你只要把组件放入某一种布局，布局由专门的布局管理器进行管理。当需要调整大小或者位置的时候，Qt 使用对应的布局管理器进行调整。

布局定位完美的解决了使用绝对定位的缺陷。

Qt 提供的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；

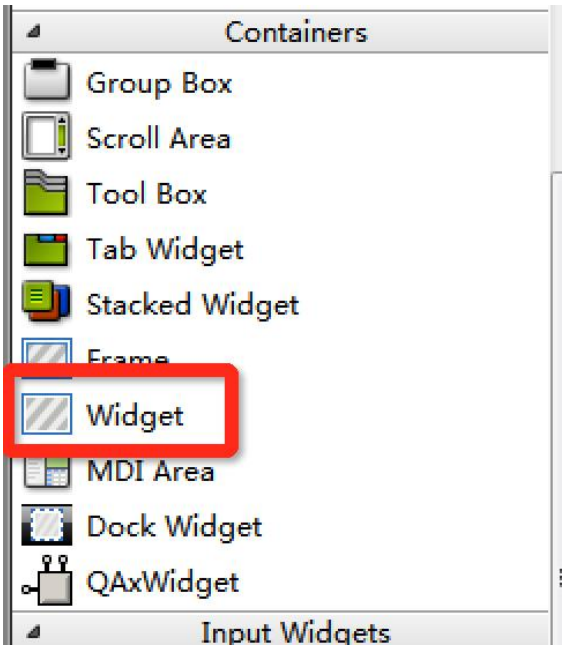
7.1 系统提供的布局控件



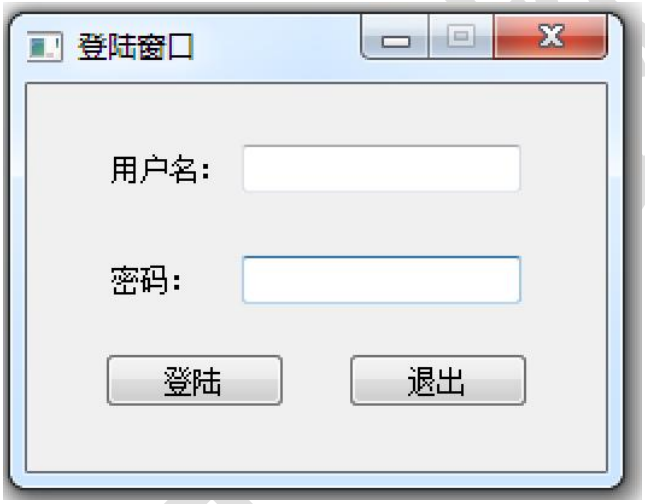
这 4 个为系统给我们提供的布局的控件，但是使用起来不是非常的灵活，这里就不详细介绍了。

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：



八、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);
```

8.1.2 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 自定义控件

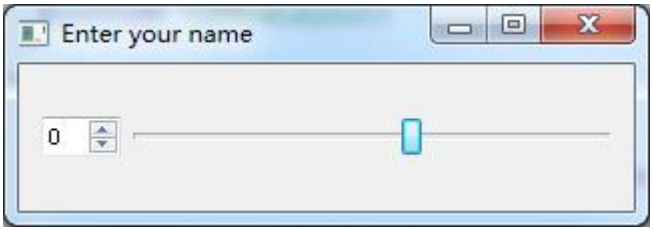
在搭建 Qt 窗口界面的时候，在一个项目中很多窗口，或者是窗口中的某个模块会被经常性的重复使用。一般遇到这种情况我们都会将这个窗口或者模块拿出来做成一个独立的窗口类，以备以后重复使用。

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，

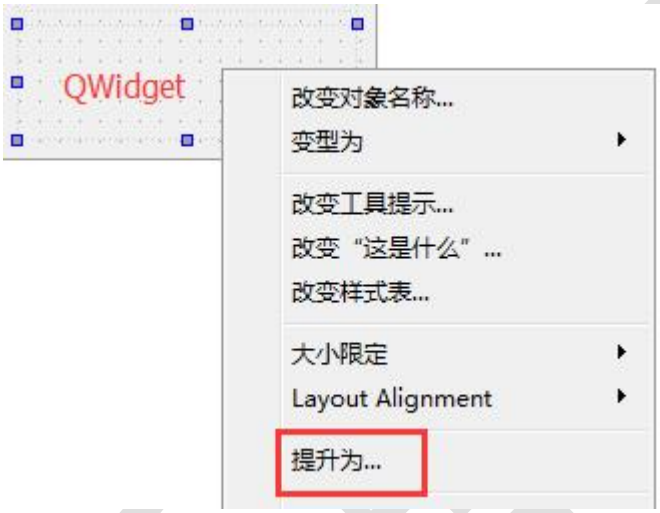
```
// smallwidget.h
class SmallWidget : public QWidget
{
    Q_OBJECT
public:
    explicit SmallWidget(QWidget *parent = 0);

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);
}
```

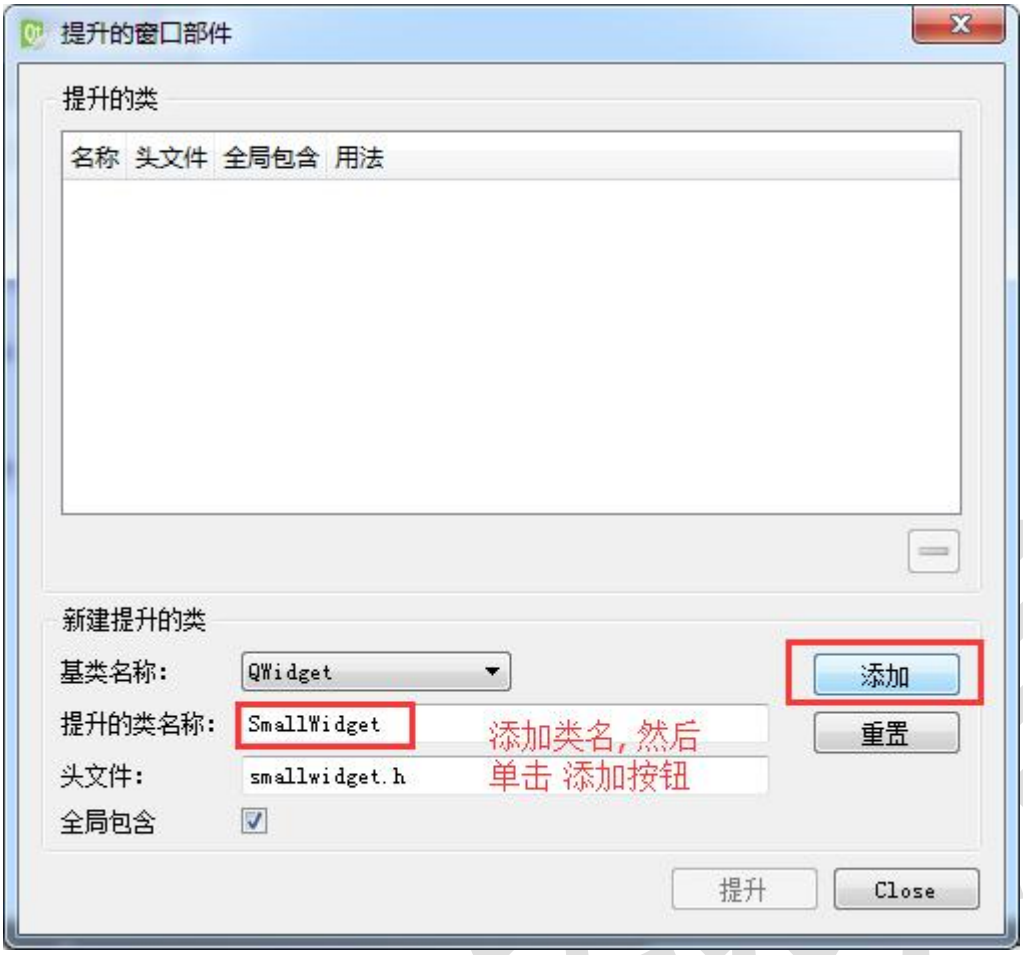


那么这个 SmallWidget 可以作为独立的窗口显示,也可以作为一个控件来使用:

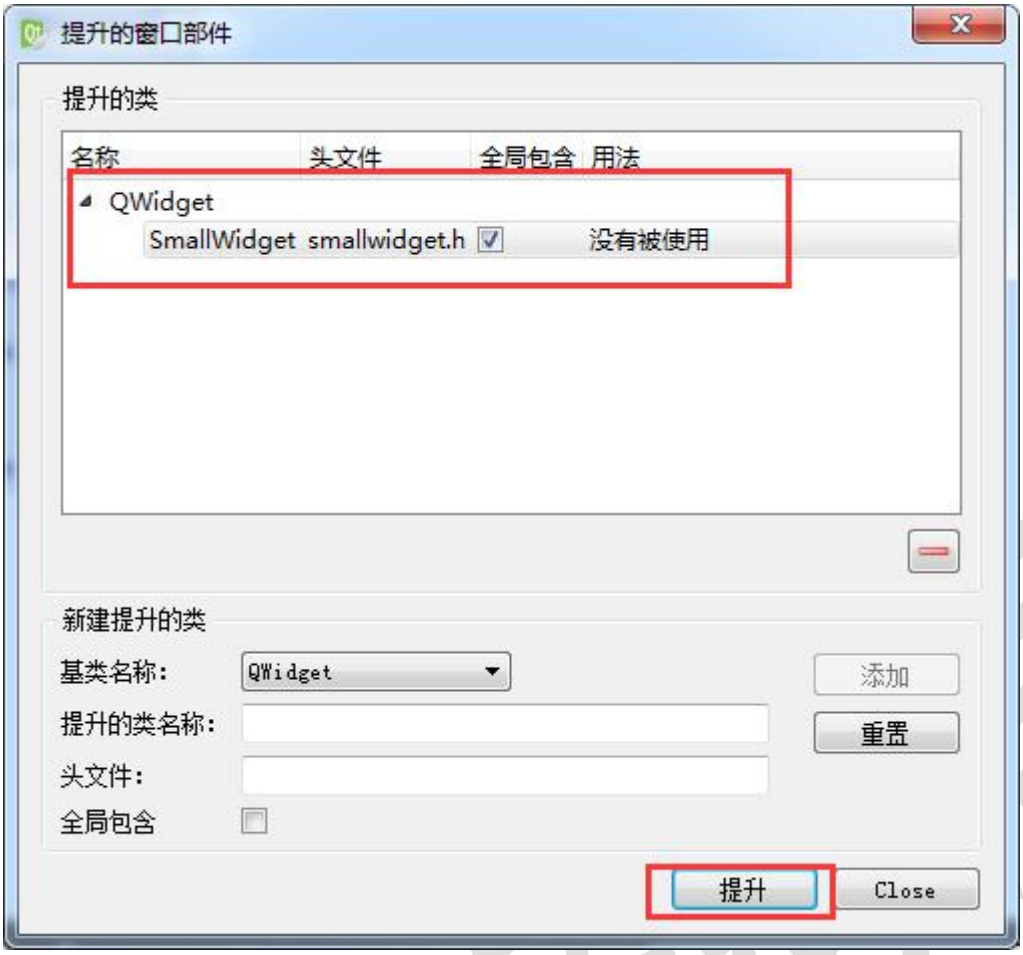
打开 Qt 的.ui 文件,因为 SmallWidget 是派生自 Qwidget 类,所以需要在 ui 文件中先放入一个 QWidget 控件, 然后再上边鼠标右键



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.

九、Qt 消息机制和事件

9.1 事件处理过程

事件（event）是由系统或者 Qt 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。

事件处理过程：

1) 在 Qt 内部，Qt 通过 QApplication::exec()启动的主事件循环不停的抓取事件队列中的事件。

2) 当事件发生时，Qt 将创建一个事件对象。Qt 中所有事件类都继承于 QEvent。

3) 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）。

event()函数主要用于事件的分发：

```
bool Widget::event(QEvent *e)
{
    if(e->type() == QEvent::MouseButtonPress)
    {
        QMouseEvent *ev = (QMouseEvent *)e;
        mousePressEvent(ev);
        return true;
    }
    else if(e->type() == QEvent::MouseMove)
    {
        QMouseEvent *ev = (QMouseEvent *)e;
        mouseMoveEvent(ev);
        return true;
```

```
    }
    else if(e->type() == QEvent::MouseButtonDblClick)
    {
        QMouseEvent *ev = (QMouseEvent *)e;
        mouseDoubleClickEvent(ev);
        return true;
}
    return QWidget::*event*(e);
}
```

9.2 常用事件处理

在所有控件的父类 QWidget 中，定义了很多事件处理的回调函数。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。实现函数的要遵循虚函数的语法规则，自定义的类中保证函数名、参数的一致性。

9.2.1 鼠标事件(QMouseEvent)

```
//重写父类的虚函数
protected:
virtual void mousePressEvent(QMouseEvent *); //鼠标点击事件
virtual void mouseReleaseEvent(QMouseEvent *); //鼠标抬起事件
virtual void mouseDoubleClickEvent(QMouseEvent *); //鼠标双击事件
```

virtual void mouseMoveEvent(QMouseEvent *); //鼠标移动事件

9.2.2 滚轮事件(QWheelEvent)

参考代码：code\day03\02_QWheelEvent

```
//virtual void wheelEvent(QWheelEvent *); //鼠标滚轮滑动

//鼠标滚轮滑动

void Widget::wheelEvent(QWheelEvent *e)

{

if(e->orientation()== Qt::Vertical) //如果方向是垂直

{

QPoint point = e->angleDelta();

QString str = QString("滚轮垂直滑动(%1, %2)")

.arg( point.x() ).arg(point.y());

ui->label->setText(str);

}

}
```

9.2.3 键盘事件(QKeyEvent)

参考代码：code\day03\03_QKeyEvent

```
//virtual void keyPressEvent(QKeyEvent *); //键盘按下事件

//virtual void keyReleaseEvent(QKeyEvent *); //键盘抬起事件

//键盘按下事件

void Widget::keyPressEvent(QKeyEvent *e)

{

QString str;
```

```
switch(e->modifiers()) //修饰键盘

{

case Qt::ControlModifier:

str = "Ctrl+";

break;

case Qt::AltModifier:

str = "Alt+";

break;

}

switch(e->key()) //普通键

{

case Qt::Key_Left:

str += "Left_Key Press";

break;

case Qt::Key_Right:

str += "Rigth_Key Press";

break;

case Qt::Key_Up:

str += "Up_Key Press";

break;

case Qt::Key_Down:

str += "Down_Key Press";

break;

case Qt::Key_Z:
```

```
str += "Z_Key Press";

break;

}

ui->label->setText(str);

}
```

9.2.4 大小改变事件(QResizeEvent)

当窗口大小发生变化时被调用，参考代码：

```
//virtual void resizeEvent(QResizeEvent *); //大小改变事件

//大小改变事件

void Widget::resizeEvent(QResizeEvent *e)

{

//变化前的窗口大小

qDebug() << "e->oldSize() = " << e->oldSize();

//变化后的窗口大小

qDebug() << "e->size() = " << e->size();

}
```

9.2.5 进入离开区域事件(enterEvent、leaveEvent)

参考代码：code\day03\05_EnterLeaveEvent

```
//virtual void enterEvent(QEvent *); //进入事件

//virtual void leaveEvent(QEvent *); //离开事件

//进入事件

void Widget::enterEvent(QEvent *)

{
```

```
qDebug() << "enterEvent";

}

//离开事件

void Widget::leaveEvent(QEvent *)

{

qDebug() << "leaveEvent";

}
```

十、

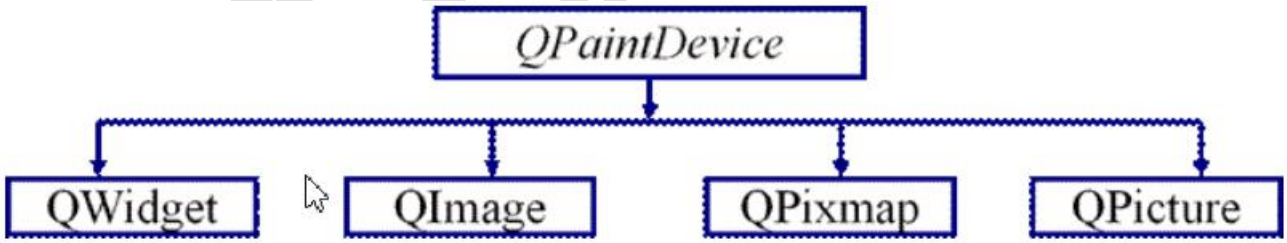
十、绘图

10.1 Qt 绘图机制

Qt 的绘图机制为屏幕显示和打印显示提供了统一的 API 接口，主要有三部分组成：QPainter 类、QPaintDevice 类和 QPaintEngine 类

l QPainter 类提供了画图操作的各种接口，可方便地绘制各种各样的图形。

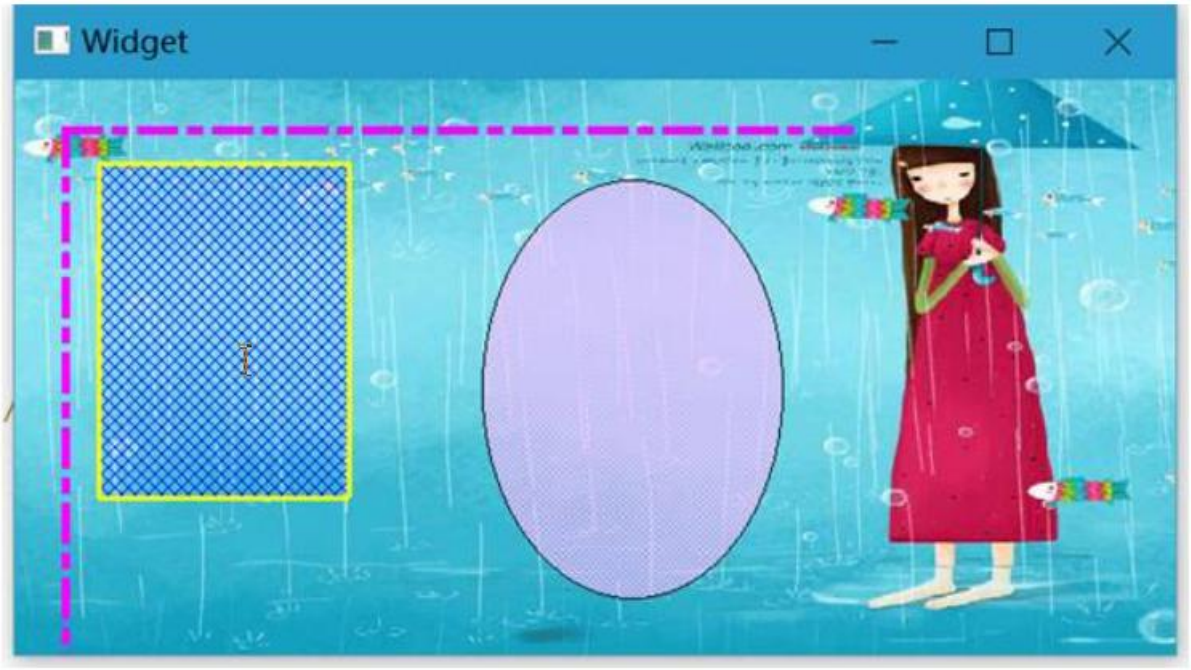
l QPaintDevice 类提供可用于画图的空间，及画图容器。



l PaintEngine 类是抽象类，提供了 QPainter 如何在指定的平台上给指定的设备绘画的抽象接口，对开发者而言，一般不会用到。

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）。

10.2 绘图事件(QPaintEvent)



如果是给控件绘图，QPainter 一般要放在 paintEvent()**里，否则会初始化失败。**

参考代码：

```
void Widget::paintEvent(QPaintEvent *e)

{

qDebug()<<"in paintEvent"<<"recet"<<e->rect(); //得到需要重新绘制的区域

QPainter paint(this); //定义画家（请了一个画家来画画），画在主窗口上

//画家画图片，作为背景图片

paint.drawPixmap(0,0,this->width(),this->height(),

QPixmap("../Image/bk.jpg"));

QPen pen; //定义一只画笔
```

```
pen.setColor(QColor(255,0,255)); //设置画笔的颜色

pen.setWidth(5); //设置画笔的宽度

pen.setStyle(Qt::DashDotLine); //设置画笔线条的风格

paint.setPen(pen); //画家使用这只画笔

paint.drawLine(30,30,500,30);//画家画水平的线条

paint.drawLine(30,30,30,500);//画家画垂直的线条

pen.setColor(Qt::yellow);//设置画笔的颜色

pen.setStyle(Qt::SolidLine);//设置画笔线条的风格

pen.setWidth(3);//设置线条的宽度

paint.setPen(pen);//画家使用这只画笔

QBrush brush(Qt::blue);//定义一把蓝色的画刷

brush.setStyle(Qt::DiagCrossPattern);//设置画刷的风格

paint.setBrush(brush);//画家使用这把画刷

paint.drawRect(50,50,150,200);//画家画矩形

paint.setPen(QPen());

paint.setBrush(QBrush(QColor(255,187,255),Qt::Dense3Pattern));

paint.drawEllipse(280,60,180,250);//画家画椭圆

}
```

10.3 刷新绘图区域(update)

参考代码：

```
//绘图事件

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

```
{
    QPainter p(this); //在窗口上绘图
    p.drawPixmap(x,200,100,100,QPixmap("../Image/face.png"));
    p.drawPixmap(i,j,100,100,QPixmap("../Image/face.png"));
}

void Widget::on_pushButton_clicked()
{
    x += 20;
    if(x > this->width())
    {
        x = 0;
    }
    this->update(); //刷新绘图区
}

void Widget::*mousePressEvent*(QMouseEvent *e)
{
```

```
    i = e->pos().x();
    j = e->pos().y();
    this->update(); //刷新绘图区域
}
```