

传智播客 - C++学院

Qt5 教程

目录

目录.....	1
1 Qt 概述.....	3
1.1 什么是 Qt.....	3
1.2 Qt 的发展史.....	4
1.3 支持的平台.....	4
1.4 Qt 版本.....	4
1.5 Qt 的安装.....	5
Linux Host.....	5
OS X Host.....	5
Windows Host.....	5
1.6 Qt 的优点.....	5
2 创建 Qt 项目.....	6
2.1 使用向导创建.....	6
2.2 手动创建.....	9
2.3 .pro 文件.....	10
2.4 一个最简单的 Qt 应用程序.....	12
3 信号和槽机制.....	13
3.1 信号和槽.....	13
3.2 自定义信号槽.....	15
自定义信号槽需要注意的事项.....	18
信号槽的更多用法.....	18
3.3 Lambda 表达式.....	19
4 Qt 窗口系统.....	21
4.1 Qt 窗口坐标体系.....	21
坐标体系.....	21
4.2 QWidget.....	21

4.2.1 对象模型.....	21
4.3 QMainWindow.....	23
4.3.1 菜单栏.....	24
4.3.2 工具栏.....	25
4.3.3 状态栏.....	25
4.4 资源文件.....	26
4.5 对话框 QDialog.....	29
4.5.1 基本概念.....	29
4.5.2 标准对话框.....	30
4.5.3 自定义消息框.....	31
4.5.4 消息对话框.....	33
4.5.5 标准文件对话框.....	36
4.6 常用控件.....	39
4.6.1 QLabel 控件使用.....	39
4.6.2 QLineEdit.....	41
4.6.3 其他控件.....	43
4.7 布局管理器.....	43
4.7.1 水平/垂直/网格布局.....	44
4.7.2 自定义控件.....	46
5 Qt 消息机制和事件.....	50
5.1 事件.....	50
5.2 event ()	52
5.3 事件过滤器.....	55
5.4 总结.....	59
5.5 不规则窗体.....	62
6 绘图和绘图设备.....	63
6.1 QPainter.....	63
6.2 绘图设备.....	65
6.2.1 QPixmap、QBitmap、QImage.....	66
6.2.2 QPicture.....	69
7 文件系统.....	70
7.1 基本文件操作.....	72

7.2 二进制文件读写.....	74
7.3 文本文件读写.....	75
8 Socket 通信.....	77
8.1 TCP/IP.....	77
服务器端.....	78
客户端.....	80
8.2 UDP.....	81
广播.....	82
组播.....	83
8.3 TCP/IP 和 UDP 的区别.....	83
9 多线程.....	84
9.1 线程介绍.....	84
9.2 多线程的使用.....	87
9.3 使用线程绘图.....	89
10 数据库操作.....	92
10.1 数据库操作.....	92
10.2 使用模型操作数据库.....	98
查询操作.....	98
插入操作.....	99
更新操作.....	99
删除操作.....	100
10.3 可视化显示数据库数据.....	101
11 Qt 程序打包.....	102

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 Creator 3.1.0 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 5.5.0

1.3 支持的平台

- Windows - XP、Vista、Win7、Win8、Win2008、Win10
- Unix/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 的安装

Linux Host

- [Qt 5.5.0 for Linux 32-bit \(535 MB\) __ \(info\)](#)
- [Qt 5.5.0 for Linux 64-bit \(532 MB\) __ \(info\)](#)
- [Qt 5.5.0 for Android \(Linux 64-bit, 605 MB\) __ \(info\)](#)
- [Qt 5.5.0 for Android \(Linux 32-bit, 608 MB\) __ \(info\)](#)

OS X Host

- [Qt 5.5.0 for Mac \(588 MB\) __ \(info\)](#)
- [Qt 5.5.0 for Android \(Mac, 652 MB\) __ \(info\)](#)
- [Qt 5.5.0 for Android and iOS \(Mac, 1.7 GB\) __ \(info\)](#)

Windows Host

- [Qt 5.5.0 for Windows 64-bit \(VS 2013, 650 MB\) __ \(info\)](#)
- [Qt 5.5.0 for Windows 32-bit \(VS 2013, 633 MB\) __ \(info\)](#)
- [Qt 5.5.0 for Windows 32-bit \(VS 2012, 587 MB\) __ \(info\)](#)
- [Qt 5.5.0 for Windows 32-bit \(VS 2010, 585 MB\) __ \(info\)](#)
- [Qt 5.5.0 for Windows 32-bit \(MinGW 4.9.2, 959 MB\) __ \(info\)](#)
- [Qt 5.5.0 for Android \(Windows 32-bit, 1.0 GB\) __ \(info\)](#)
- [Qt 5.5.0 for Windows RT 32-bit \(621 MB\) __ \(info\)](#)

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 5.5.0 for Windows 32-bit \(MinGW 4.9.2, 959 MB\) __ \(info\)](#) 版本就行讲解。

MinGW32 --> Minimalist GNU for Windows 32

1.6 Qt 的优点

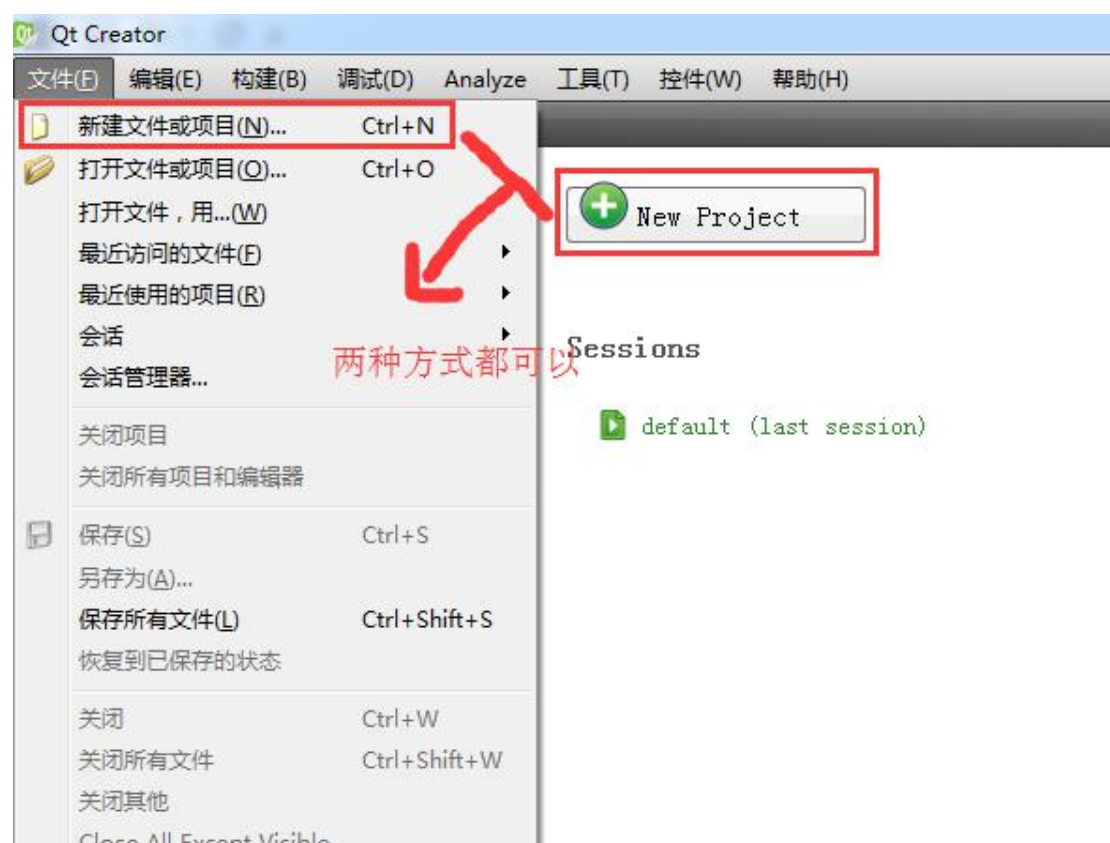
- 跨平台，几乎支持所有的平台
- 接口简单，容易上手，学习 QT 框架对学习其他框架有参考意义。
- 一定程度上简化了内存回收机制

- 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。

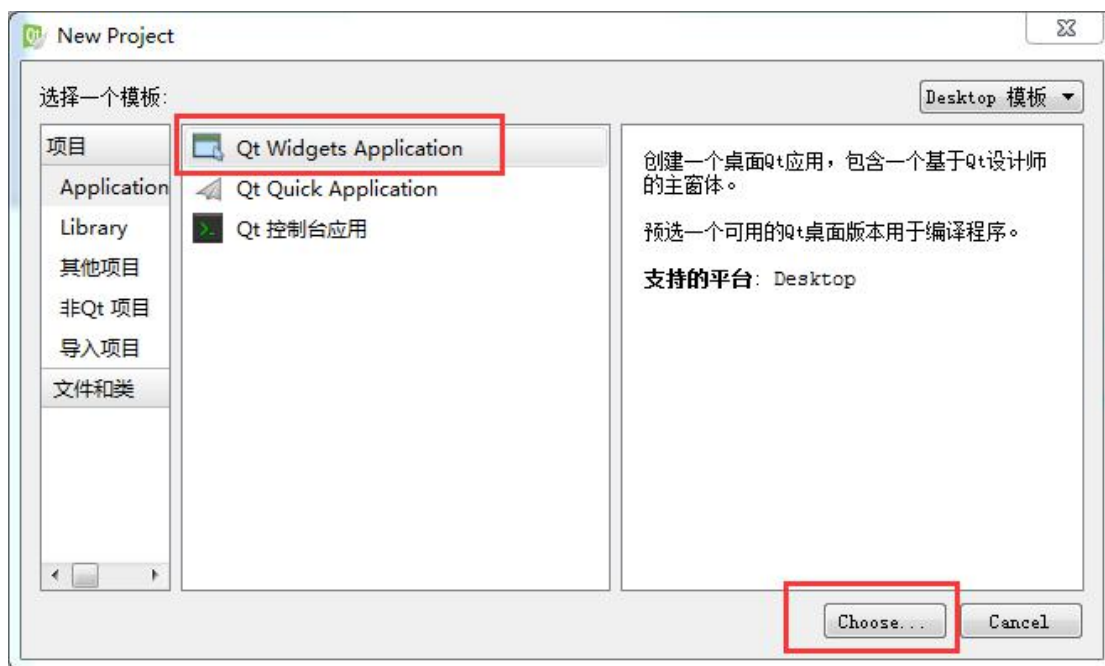
2 创建 Qt 项目

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



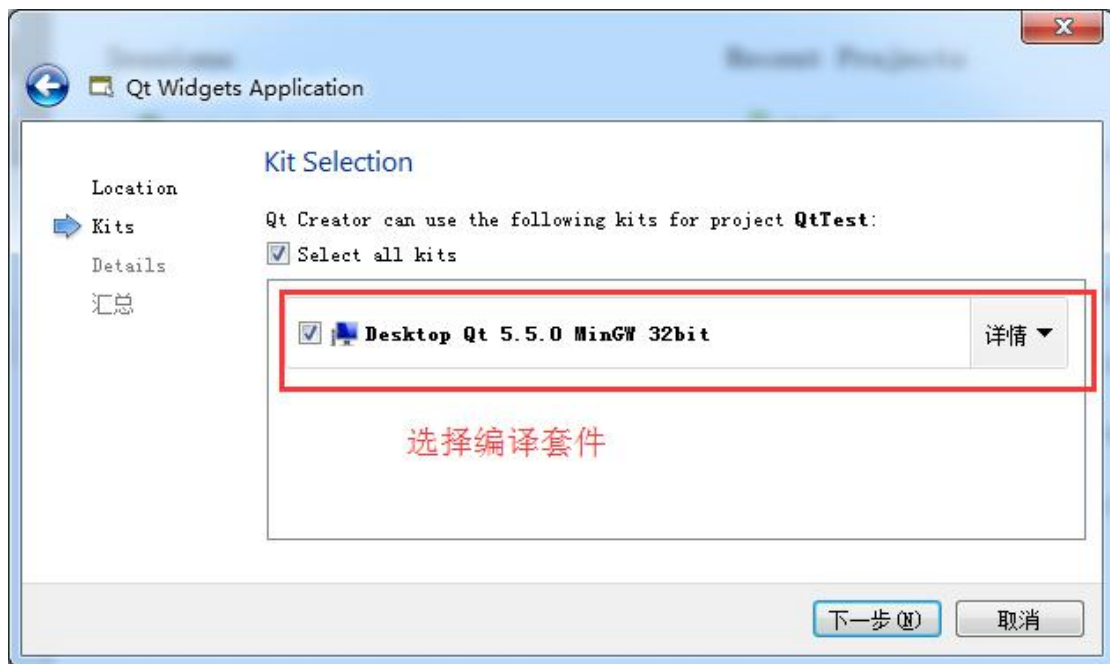
弹出 New Project 对话框，选择 Qt Widgets Application，



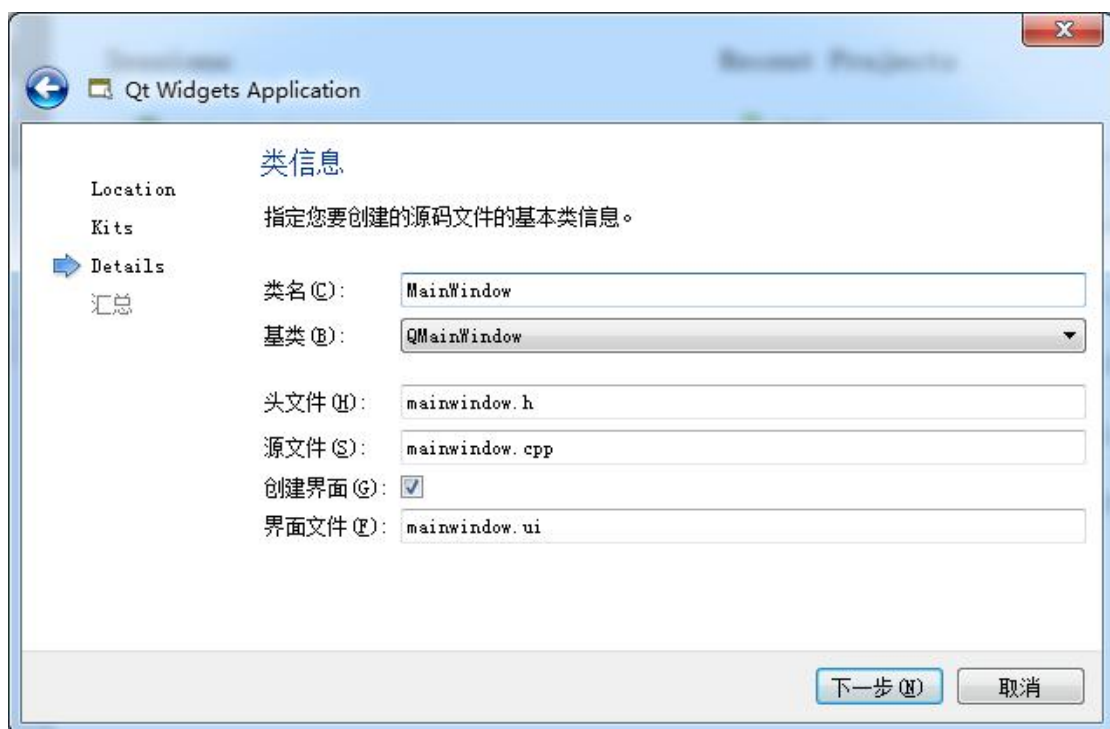
选择【Choose】按钮，弹出如下对话框



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。

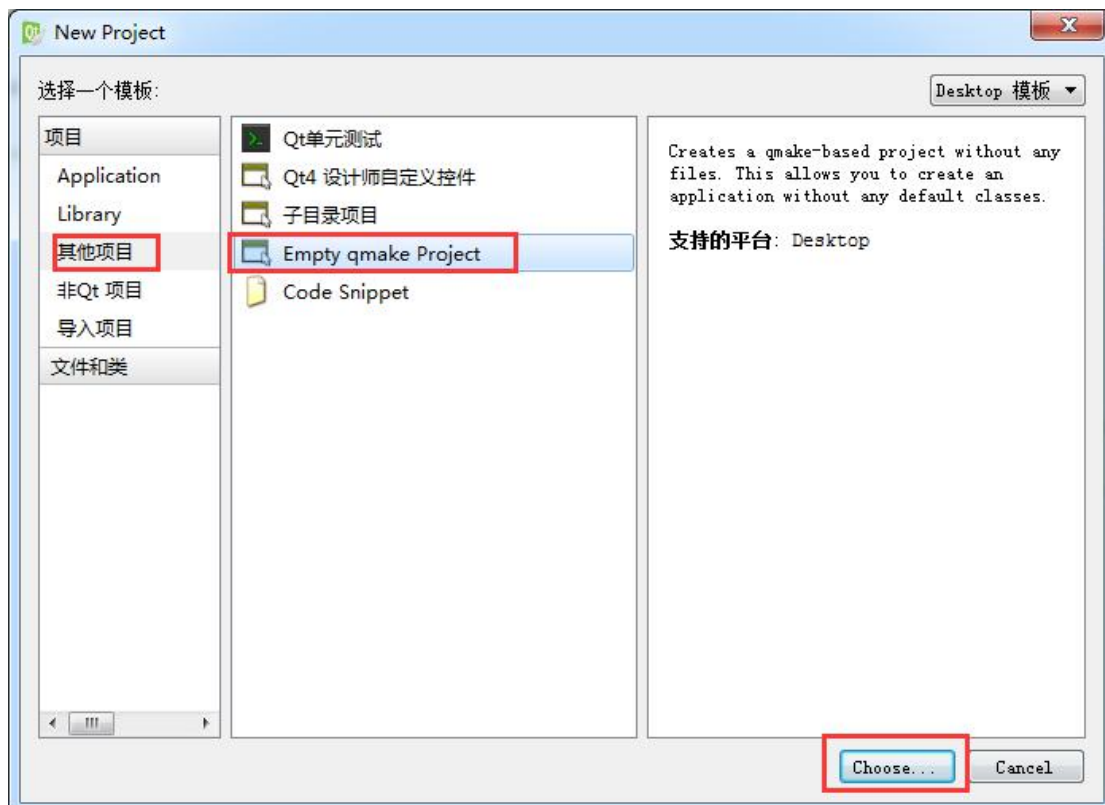
继续下一步



即可创建出一个 Qt 桌面程序。

2.2 手动创建

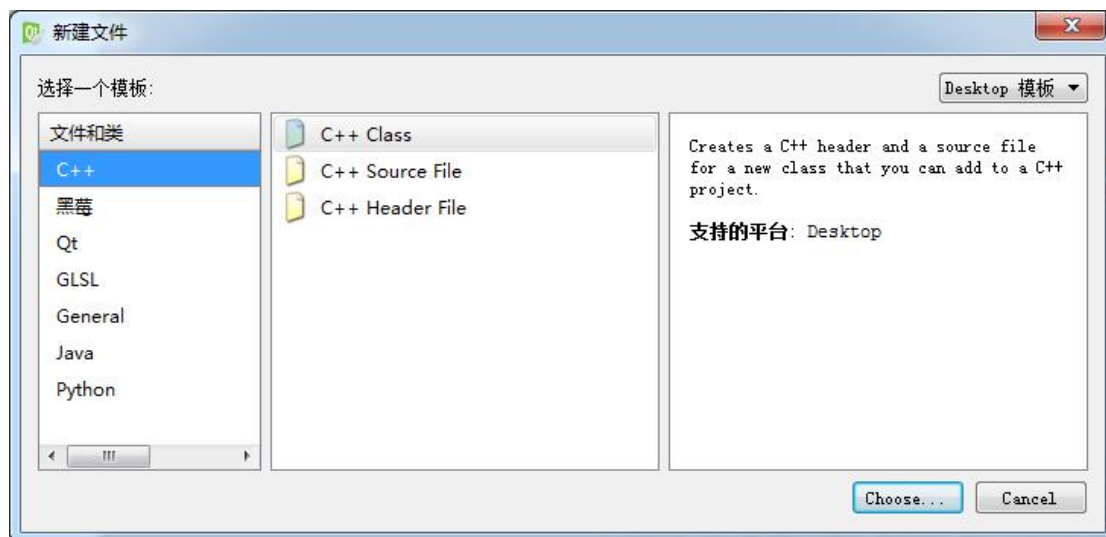
添加一个空项目



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    //模块的名字
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets

TARGET = test    //应用程序名
TEMPLATE = app    //生成的 makefile 的模板类型

//源文件
SOURCES += main.cpp\
           mainwindow.cpp

//头文件
HEADERS += mainwindow.h

//窗口设计文件
FORMS    += mainwindow.ui
```

.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/painter.cpp`

- #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

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

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

    return a.exec();
}
```

解释：

- Qt 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。

3 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。（这里提一句，Qt 的信号槽使用了额外的处理来实现，并不是 GoF 经典的观察者模式的实现方式。）

3.1 信号和槽

为了体验一下信号槽的使用，我们以一段简单的代码说明：

- Qt5 的书写方式

```
#include <QApplication>
#include <QPushButton>

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

```
QApplication app(argc, argv);

QPushButton button("Quit");
QObject::connect(&button, &QPushButton::clicked,
                &app, &QApplication::quit);

button.show();
return app.exec();
}
```

我们按照前面文章中介绍的在 Qt Creator 中创建工程的方法创建好工程，然后将 main() 函数修改为上面的代码。点击运行，我们会看到一个按钮，上面有“Quit”字样。点击按钮，程序退出。

connect() 函数最常用的一般形式：

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

参数：

- 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

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），但是不能说信号根本没有这个数据，你就要在槽函数中使用（就是槽函数的参数比信号的多，这是不允许的）。

如果信号槽不符合，或者根本找不到这个信号或者槽函数，比如我们改成：

```
connect(&button, &QPushButton::clicked, &QApplication::quit2);
```

由于 QApplication 没有 quit2 这样的函数，因此在编译时会有编译错误：

```
'quit2' is not a member of QApplication
```

这样，使用成员函数指针我们就不会担心在编写信号槽的时候出现函数错误。

● Qt4 的书写方式：

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

```
QPushButton *button = new QPushButton("Quit");
connect(button, SIGNAL(clicked()), &a, SLOT(quit()));
button->show();
return a.exec();
}
```

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

3.2 自定义信号槽

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，实现一个报纸和订阅者的例子：

有一个报纸类 `Newspaper`，有一个订阅者类 `Subscriber`。`Subscriber` 可以订阅 `Newspaper`。这样，当 `Newspaper` 有了新的内容的时候，`Subscriber` 可以立即得到通知。

```
#include <QObject>
////////// newspaper.h //////////
class Newspaper : public QObject
{
    Q_OBJECT
public:
    Newspaper(const QString & name) :
        m_name(name)
    {
    }

    void send()
    {
        emit newPaper(m_name);
    }
}
```

```
signals:
    void newPaper(const QString &name);

private:
    QString m_name;
};

////////// reader.h //////////
#include <QObject>
#include <QDebug>

class Reader : public QObject
{
    Q_OBJECT
public:
    Reader() {}

    void receiveNewspaper(const QString & name)
    {
        qDebug() << "Receives Newspaper: " << name;
    }
};

////////// main.cpp //////////
#include <QCoreApplication>

#include "newspaper.h"
#include "reader.h"

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

    Newspaper newspaper("Newspaper A");
    Reader reader;
    QObject::connect(&newspaper, &Newspaper::newPaper,
                    &reader,    &Reader::receiveNewspaper);
    newspaper.send();
}
```



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

- 首先看 Newspaper 这个类。这个类继承了 QObject 类。**只有继承了 QObject 类的类，才具有信号槽的能力。**所以，为了使用信号槽，必须继承 QObject。**凡是 QObject 类（不管是直接子类还是间接子类），都应该在第一行代码写上 Q_OBJECT。**不管是不是使用信号槽，都应该添加这个宏。这个宏的展开将为我们提供信号槽机制、国际化机制以及 Qt 提供的不基于 C++ RTTI 的反射能力。
- Newspaper 类的 public 和 private 代码块都比较简单，只不过它新加了一个 signals。signals 块所列出的，就是该类的信号。**信号就是一个一个的函数名，返回值是 void（因为无法获得信号的返回值，所以也就无需返回任何值），参数是该类需要让外界知道的数据。信号作为函数名，不需要在 cpp 函数中添加任何实现。**
- Newspaper 类的 send() 函数比较简单，只有一个语句 emit newPaper(m_name);。emit 是 Qt 对 C++ 的扩展，是一个关键字（其实也是一个宏）。emit 的含义是发出，也就是发出 newPaper() 信号。感兴趣的接收者会关注这个信号，可能还需要知道是哪份报纸发出的信号？所以，我们将实际的报纸名字 m_name 当做参数传给这个信号。当接收者连接这个信号时，就可以通过槽函数获得实际值。这样就完成了数据从发出者到接收者的一个转移。
- Reader 类更简单。因为这个类需要接受信号，所以我们将其继承了 QObject，并且添加了 Q_OBJECT 宏。后面则是默认构造函数和一个普通的成员函数。**Qt 5 中，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。**与信号函数不同，槽函数必须自己完成实现代码。槽函数就是普通的成员函数，因此作为成员函数，也会受到 public、private 等访问控制符的影响。（如果信号是 private 的，这个信号就不能在类的外面连接，也就没有任何意义。）

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);
- 使用 `signals` 标记信号函数,信号是一个函数声明,返回 `void`,不需要实现函数代码;
- 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 `QObject::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

信号槽的更多用法

- 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。

- 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。我们的代码可以写成下面这样:

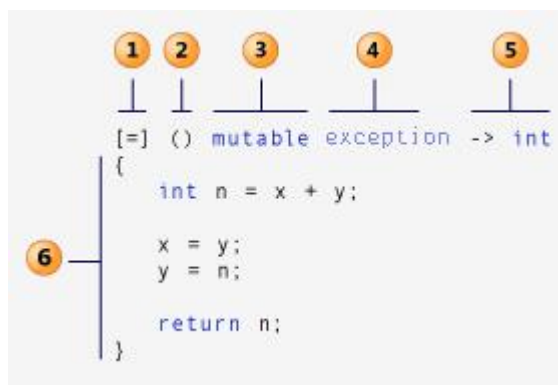
```
QObject::connect(&newspaper, static_cast<void (Newspaper:: *)  
                (const QString &)>(&Newspaper::newPaper),  
                [=](const QString &name)  
                { /* Your code here. */ }  
                );
```

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。

3.3 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：



[函数对象参数] (操作符重载函数参数) mutable 或 exception -> 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。

```
int m = 0, n = 0;
[=] (int a) mutable { m = ++n + a; } (4);
[&] (int a) { m = ++n + a; } (4);

[=,&m] (int a) mutable { m = ++n + a; } (4);
[&,&m] (int a) mutable { m = ++n + a; } (4);

[m,n] (int a) mutable { m = ++n + a; } (4);
[&m,&n] (int a) { m = ++n + a; } (4);
```

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）。

④ 错误抛出标示符；

`exception` 声明，这部分也可以省略。`exception` 声明用于指定函数抛出的异常，如抛出整数类型的异常，可以使用 `throw(int)`

⑤ 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。

⑥ 是函数体；

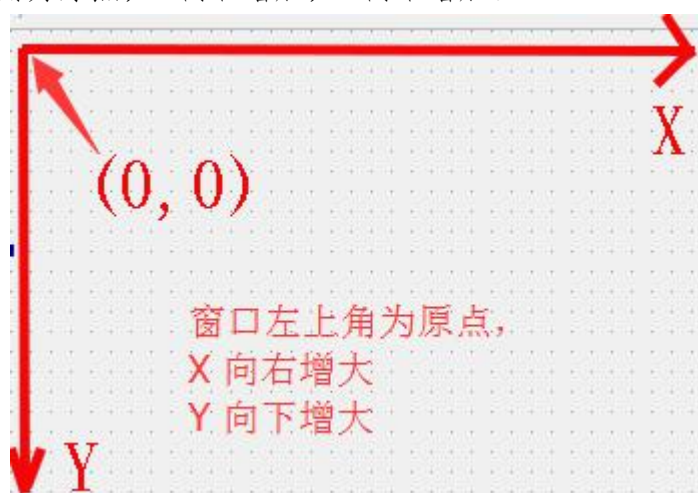
{}, 标识函数的实现, 这部分不能省略, 但函数体可以为空。

4 Qt 窗口系统

4.1 Qt 窗口坐标体系

坐标体系

以左上角为原点, 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。

4.2 QWidget

所有窗口及窗口控件都是从 QWidget 直接或间接派生出来的。

4.2.1 对象模型

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！）

这种机制在 GUI 程序设计中相当有用。例如，一个按钮有一个 `QShortcut`（快捷键）对象作为其子对象。当我们删除按钮的时候，这个快捷键理应被删除。这是合理的。

- `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("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：

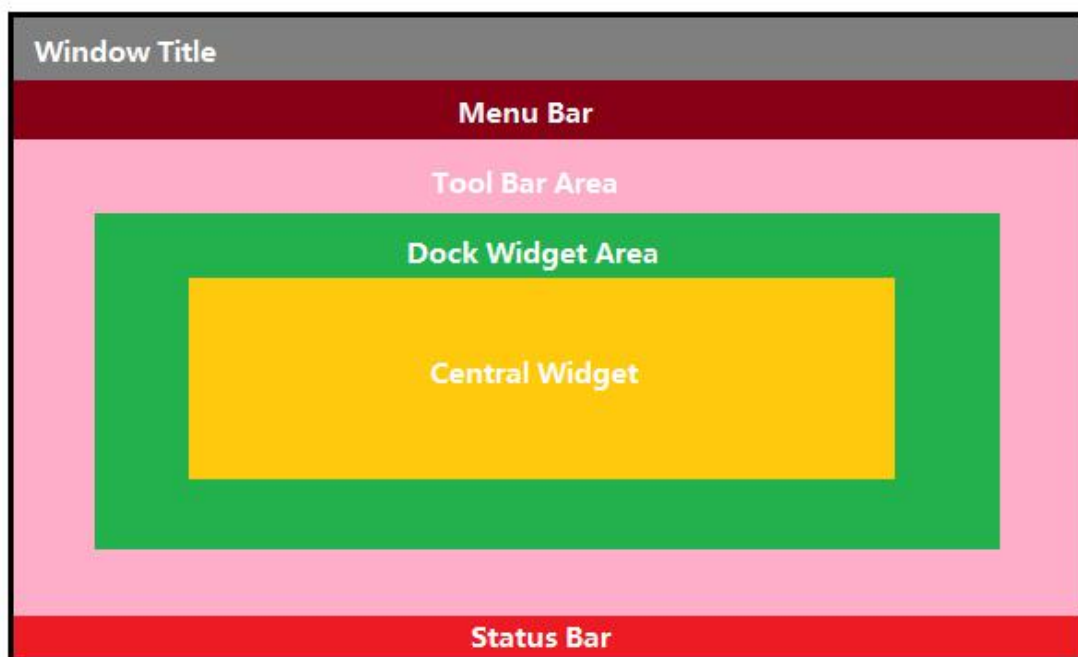
```
{
    QPushButton quit("Quit");
    QWidget window;
    quit.setParent(&window);
}
```

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。

4.3 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)、多个锚接部件 (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。



4.3.1 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。

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

```
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。

4.3.2 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方式，也可根据需要进行工具条的划分。

- 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
```

4.3.3 状态栏

- 派生自 QWidget 类，使用方法与 QWidget 类似，QStatusBar 类常用成员函数：

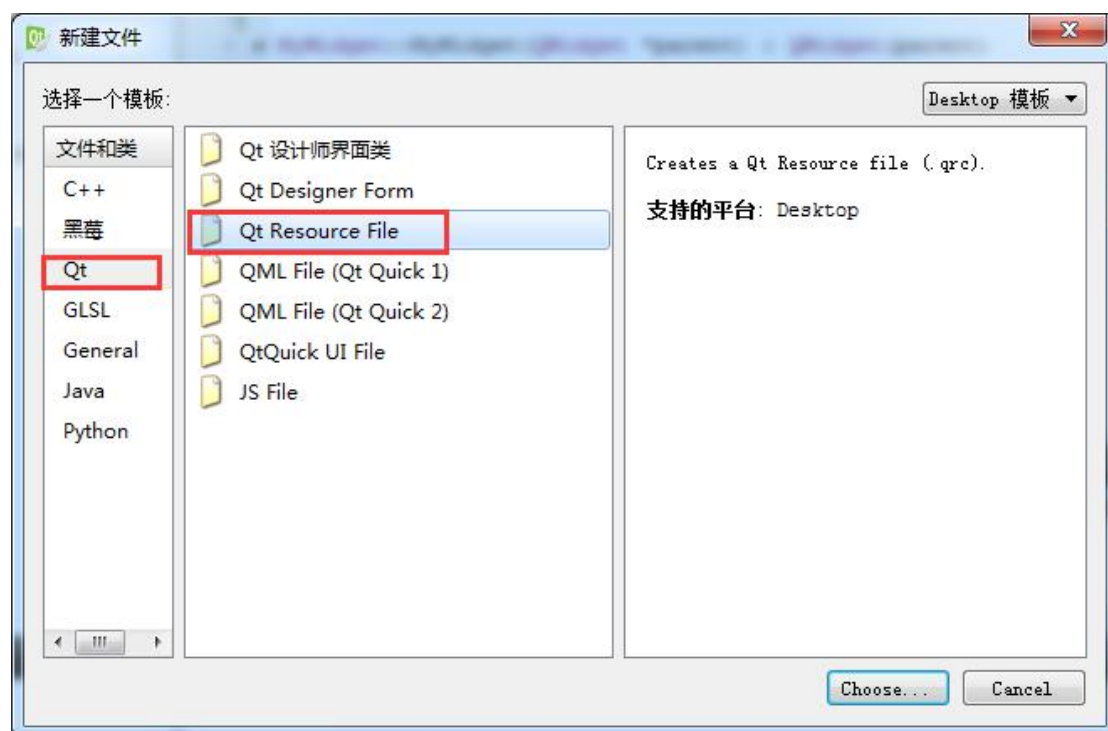
```
//添加小部件
```

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

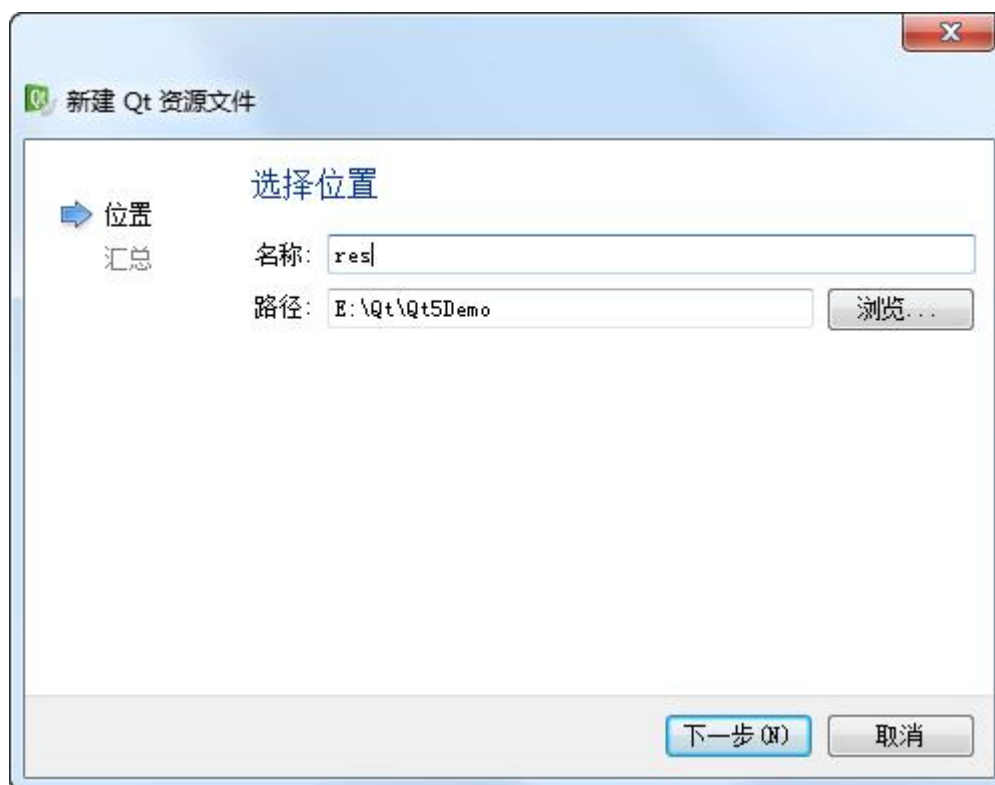
```
//插入小部件  
int insertWidget(int index, QWidget * widget, int stretch = 0)  
//删除小部件  
void removeWidget(QWidget * widget)
```

4.4 资源文件

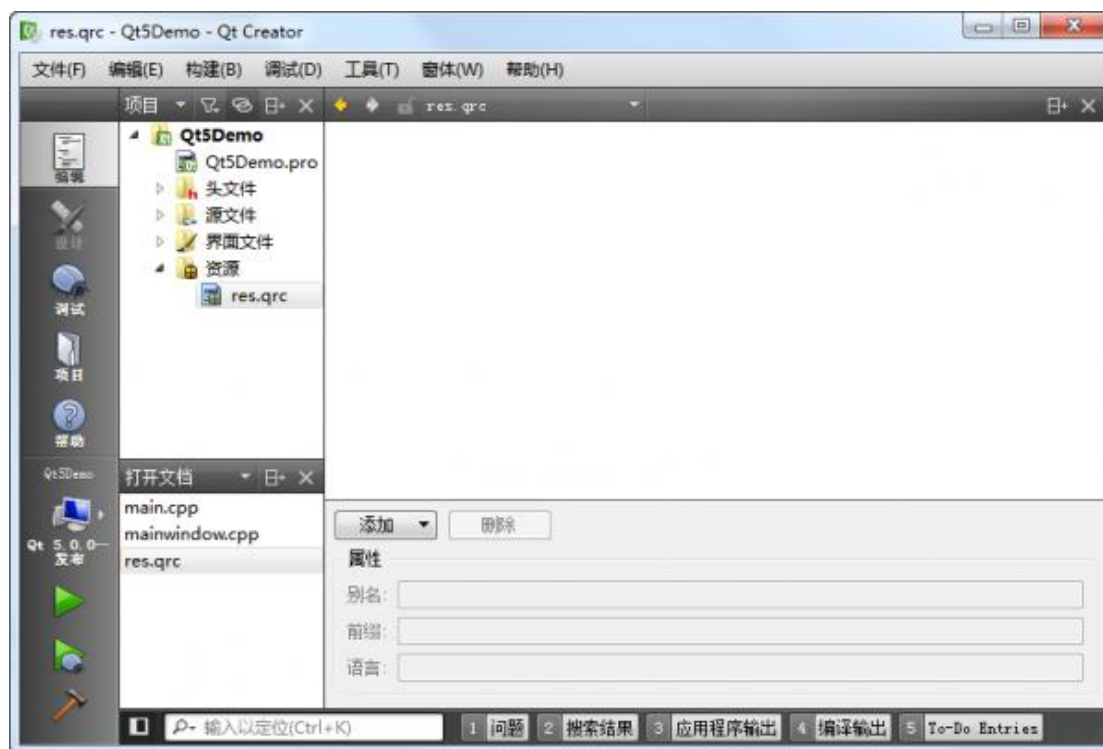
Qt 资源系统是一个跨平台的资源机制，用于将程序运行时所需要的资源以二进制的形式存储于可执行文件内部。如果你的程序需要加载特定的资源（图标、文本翻译等），那么，将其放置在资源文件中，就再也不需要担心这些文件的丢失。也就是说，如果你将资源以资源文件形式存储，它是会编译到可执行文件内部。使用 Qt Creator 可以很方便地创建资源文件。我们可以在工程上点右键，选择“添加新文件...”，可以在 Qt 分类下找到“Qt 资源文件”：



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称和路径：

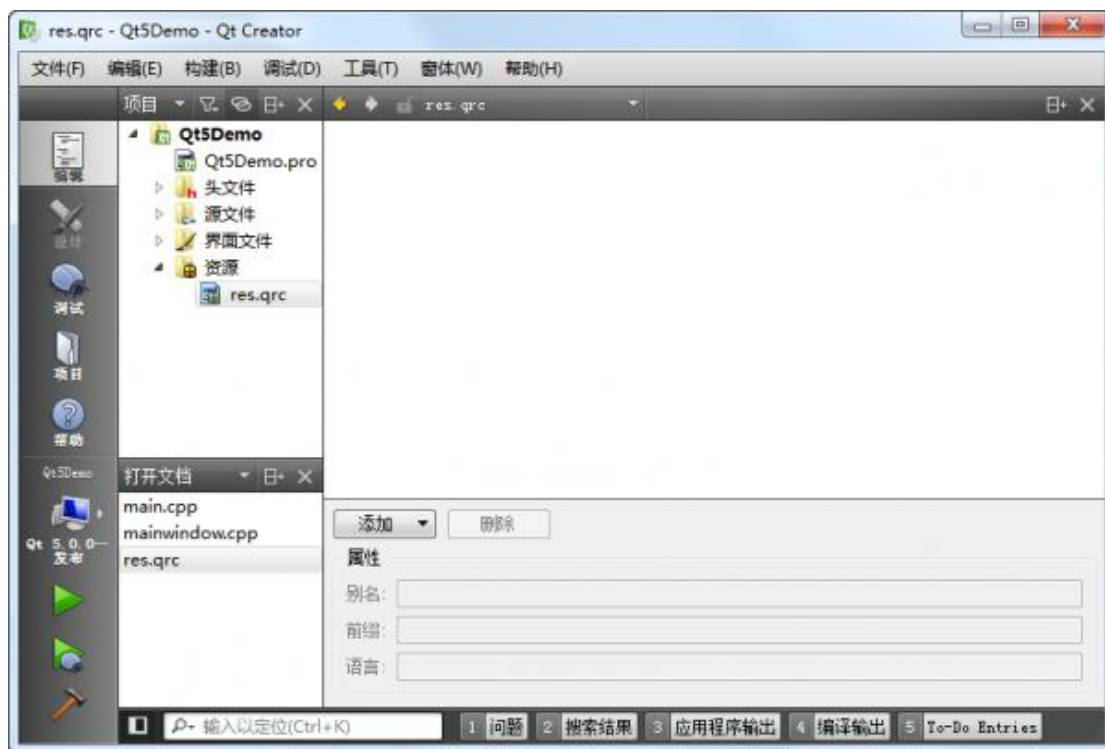


点击下一步，选择所需要的版本控制系统，然后直接选择完成。我们可以在 Qt Creator 的左侧文件列表中看到“资源文件”一项，也就是我们新创建的资源文件：



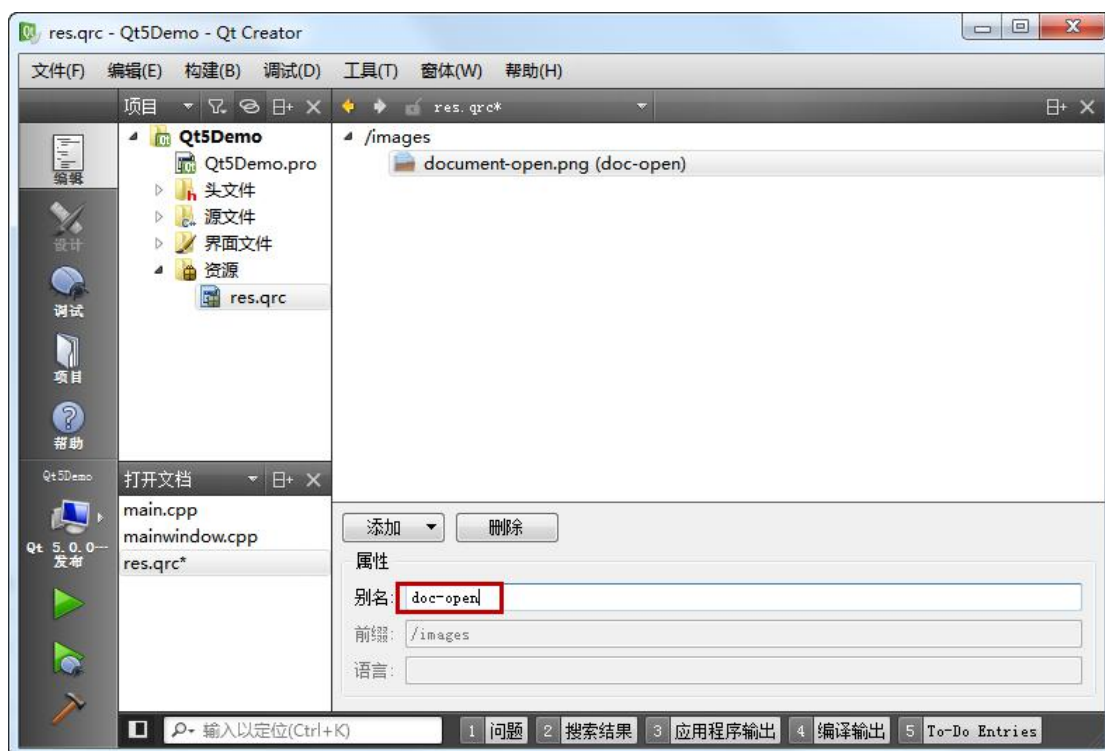
右侧的编辑区有个“添加”，我们首先需要添加前缀，比如我们将前缀取名为 images。然后选中这个前缀，继续点击添加文件，可以找到我们所需添加的文件。

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。

4.5 对话框 QDialog

4.5.1 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。

4.5.2 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。

4.5.3 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。

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。

```
void MainWindow::open()
{
    QDialog dialog;
    dialog.setWindowTitle(tr("Hello, dialog!"));
    dialog.exec();
}
```


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：

```
void MainWindow::open()
{
    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：

```
void MainWindow::open()
{
    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`：

```
void MainWindow::open()
{
    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。

4.5.4 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。

这里就是我们需要询问的文字。下面，我们使用或运算符（`|`）指定对话框应该出现的按钮。这里我们希望是一个 `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。

4.5.5 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->setShortcuts(QKeySequence::Open);
openAction->setStatusTip(tr("Open an existing file"));

saveAction = new QAction(QIcon(":/images/file-save"),
                          tr("&Save..."), this);
saveAction->setShortcuts(QKeySequence::Save);
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: %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。
Qt 的标准对话框提供静态函数，用于返回一个模态对话框；
- **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。

4.6 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的两种，所有控件的使用方法我们都可以通过帮助文档获取。

4.6.1 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。

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

```
QLabel *label = new QLabel;  
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, 如果设置为 `false`, 想要打开链接只能通过捕捉 `linkActivated()` 信号, 在自定义的槽函数中使用 `QDesktopServices::openUrl()` 打开链接, 该函数参数默认值为 `false`

```
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);  
connect(label, &QLabel::linkActivated,  
this, &MyWidget::slotOpenUrl);  
  
//槽函数  
void MyWidget::slotOpenUrl(const QString &link)  
{  
    QDesktopServices::openUrl(QUrl(link));  
}
```

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);
```

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 QMovie 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);
```

4.6.2 QLineEdit

Qt 提供的单行文本编辑框。

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 &)
```

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。

设置输入提示

如果我们想实现一个与百度的搜索框类似的功能：输入一个或几个字符，下边会列出几个跟输入的字符相匹配的字符串，QLineEdit 要实现这样的功能可以使用该类的成员函数 setCompleter() 函数来实现：

```
void setCompleter(QCompleter * c)
```

创建 QCompleter 对象，并初始化

```
QStringList tipList;  
tipList<< "Hello" << "how are you" << "Haha" << "oh, hello";  
// 不区分大小写  
completer->setCaseSensitivity(Qt::CaseInsensitive);  
QCompleter *completer = new QCompleter(tipList, this);
```

QCompleter 类的 setCaseSensitivity() 函数可以设置是否区分大小写，它的参数是一个枚举类型：

- Qt::CaseInsensitive 不区分大小写
- Qt::CaseSensitive 区分大小写

如果不设置该属性，默认匹配字符串时是区分大小写的。

另外我们还可以设置字符串其中某一部分匹配，此功能可通过 QCompleter 类的 setFilterMode 函数来实现，函数声明如下：

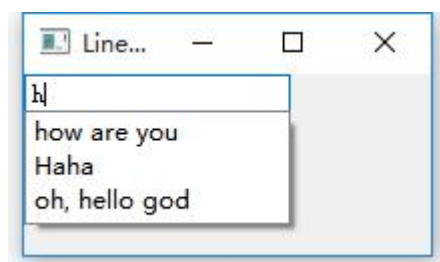
```
void setFilterMode(Qt::MatchFlags filterMode)
```

其参数为 Qt 定义的宏，有多重类型，具体可参考 Qt 帮助稳定，要实现我们上边提到的功能，参数可以使用 Qt::MatchContains：

```
completer->setFilterMode(Qt::MatchContains);
```

属性设置完成之后，将 QCompleter 对象设置到 QLineEdit 中：

```
QLineEdit *edit = new QLineEdit(this);  
edit->setCompleter(completer);
```



4.6.3 其他控件

Qt 中控件的使用方法可参考 Qt 提供的帮助文档。

4.7 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;

4.7.1 水平/垂直/网格布局

下面我们通过一个例子来学习以下水平布局管理器的使用方法:

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

    QWidget window;
    window.setWindowTitle("Enter your age");

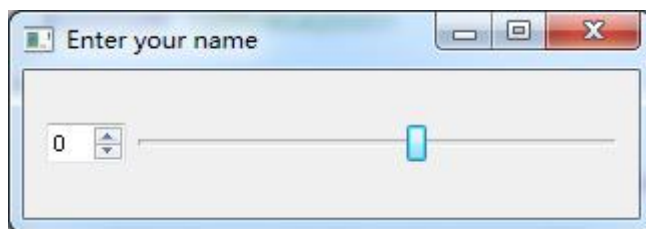
    QSpinBox *spinBox = new QSpinBox(&window);
    QSlider *slider = new QSlider(Qt::Horizontal, &window);
    spinBox->setRange(0, 130);
    slider->setRange(0, 130);

    QObject::connect(slider, &QSlider::valueChanged,
                     spinBox, &QSpinBox::setValue);
    void (QSpinBox:: *spinBoxSignal)(int) = &QSpinBox::valueChanged;
    QObject::connect(spinBox, spinBoxSignal,
                     slider, &QSlider::setValue);
    spinBox->setValue(35);

    //给控件设置布局
    QHBoxLayout *layout = new QHBoxLayout;
    layout->addWidget(spinBox);
    layout->addWidget(slider);
    window.setLayout(layout);

    window.show();
    return app.exec();
}
```

我们在这段代码中引入了两个新的组件: QSpinBox 和 QSlider。QSpinBox 就是只能输入数字的输入框, 并且带有上下箭头的步进按钮。QSlider 则是带有滑块的滑竿。



上面的代码中 `window.setLayout(layout);` 是将布局设置到窗口 `window` 中，在窗口中设置布局还有另一种写法：

```
//给控件设置布局
```

```
QHBoxLayout *layout = new QHBoxLayout (window);  
layout->addWidget(spinBox);  
layout->addWidget(slider);
```

在创建布局对象的时候给新对象指定父窗口，就等于给传入的窗口设置了布局。另外布局与布局之间是可以嵌套使用的，使用 `addLayout()` 方法。QVBoxLayout 的使用方法与 QHBoxLayout 完全相同。

关于上述代码中信号和槽连接的解释：

当数字输入框显示的内容发生改变的时候，会发出一股信息，滑块会接收这一信号，并作出改变。如果二者的信号槽连接写成下边这样：

```
QObject::connect(spinBox, &QSpinBox::valueChanged,  
                 slider, &QSlider::setValue);
```

编译器却会报错

```
no matching function for call to 'QObject::connect(QSpinBox*&,  
<unresolved overloaded function type>, QSlider*&, void  
(QAbstractSlider::*)(int))'
```

这是怎么回事呢？从出错信息可以看出，编译器认为 `QSpinBox::valueChanged` 是一个 overloaded 的函数。我们看一下 `QSpinBox` 的文档发现，`QSpinBox` 的确有两个信号：

- `void valueChanged(int)`
- `void valueChanged(const QString &)`

当我们使用 `&QSpinBox::valueChanged` 取函数指针时，编译器不知道应该取哪一个函数（记住前面我们介绍过的，`signal` 也是一个普通的函数。）的地址，因此报错。解决的方法很简单，编译器不是不能确定哪一个函数吗？那么我们就显式

指定一个函数。方法就是，我们创建一个函数指针，这个函数指针参数指定为 `int`：

```
void (QSpinBox:: *spinBoxSignal)(int) = &QSpinBox::valueChanged;
```

然后将这个函数指针作为 `signal`，与 `QSlider` 的函数连接：

```
QObject::connect(spinBox, spinBoxSignal,  
                 slider, &QSlider::setValue);
```

这样便避免了编译错误。

动手

通过布局管理器搭建如下登陆界面：



4.7.2 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`，实现了一个自定义窗口，

```
// smallwidget.h  
class SmallWidget : public QWidget  
{  
    Q_OBJECT  
public:
```

```
explicit SmallWidget(QWidget *parent = 0);

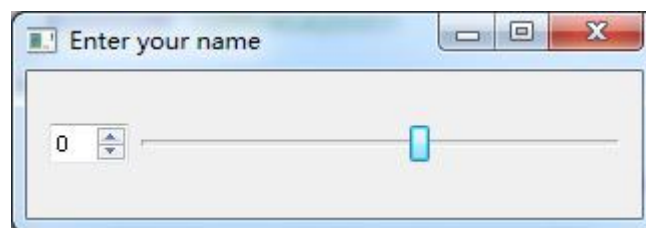
signals:

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);

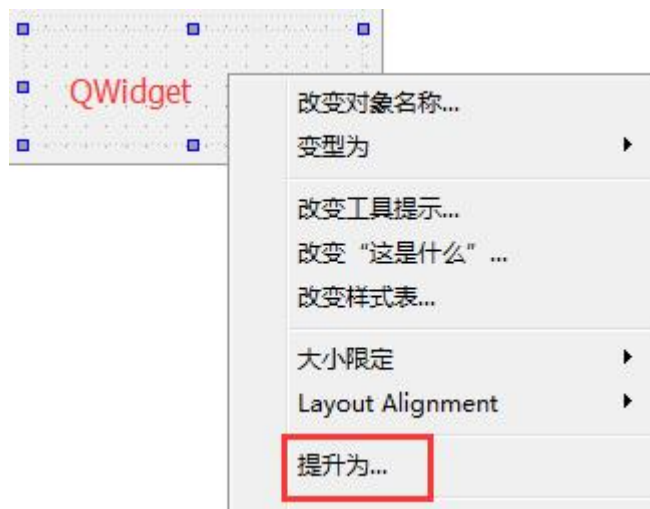
    // 创建布局对象
    QHBoxLayout* layout = new QHBoxLayout;
    // 将控件添加到布局中
    layout->addWidget(spin);
    layout->addWidget(slider);
    // 将布局设置到窗口中
    setLayout(layout);

    // 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);
}
```

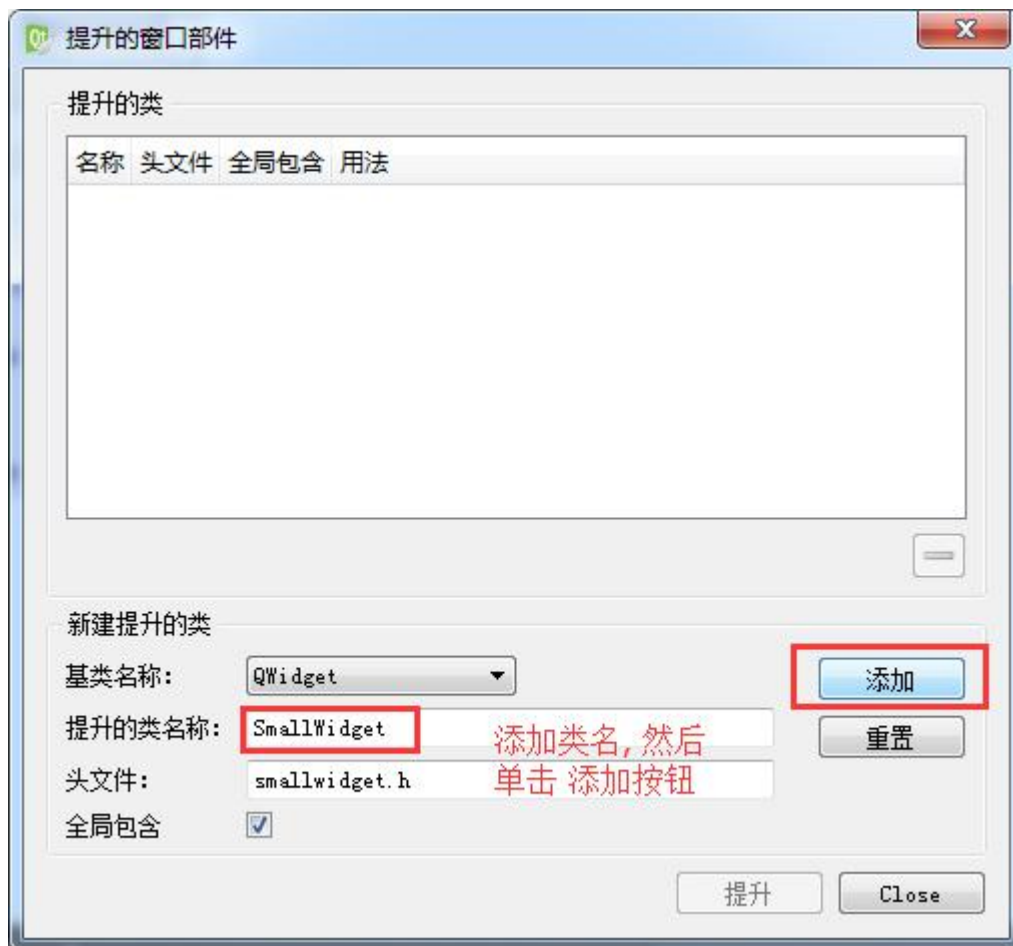


那么这个 SmallWidget 可以作为独立的窗口显示,也可以作为一个控件来使用:
打开 Qt 的 .ui 文件,因为 SmallWidget 是派生自 QWidget 类,所以需要在 ui 文件

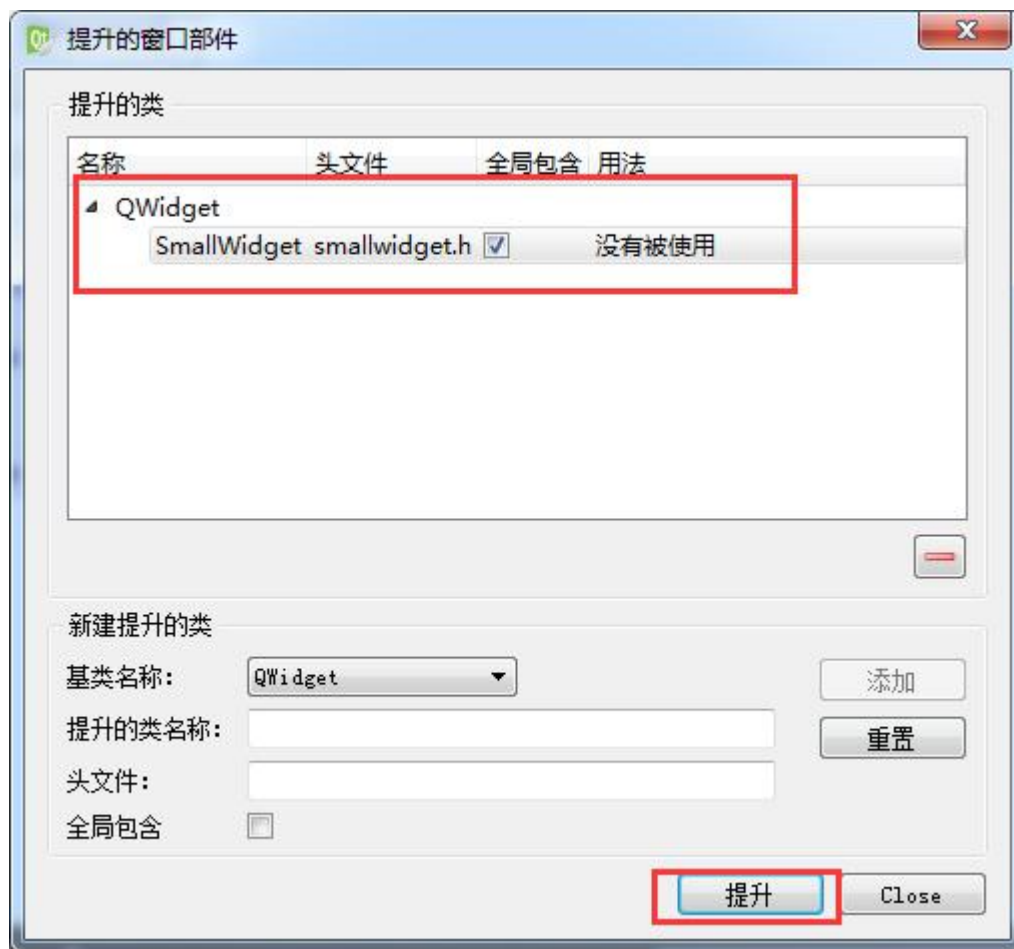
中先放入一个 QWidget 控件，然后再上边鼠标右键



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.



5 Qt 消息机制和事件

5.1 事件

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。

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 的事件循环。在执行 exec() 函数之后，程序将进入事件循环来监听应用程序的事件。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。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。

在所有组件的父类 QWidget 中，定义了很多事件处理的回调函数，如

- keyPressEvent()
- keyReleaseEvent()
- mouseDoubleClickEvent()
- mouseMoveEvent()
- mousePressEvent()
- mouseReleaseEvent() 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。

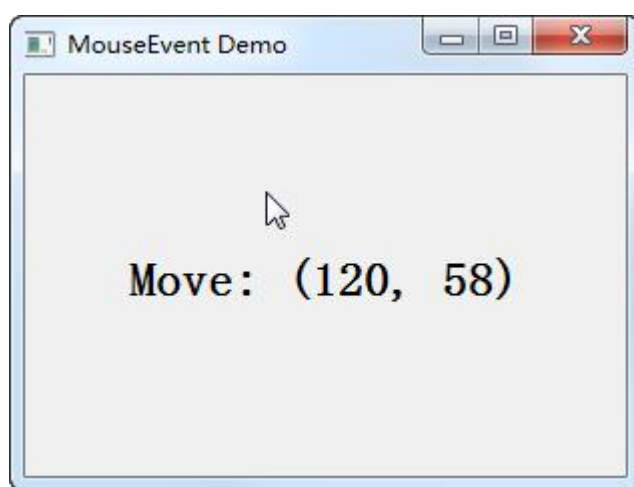
- QString 的 arg() 函数可以自动替换掉 QString 中出现的占位符。其占位符以 % 开始，后面是占位符的位置，例如 %1, %2 这种。

```
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 mousePressEvent() 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 mousePressEvent() 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，mousePressEvent() 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 mousePressEvent() 事件。如果 mouseTracking 为 true，则 mousePressEvent() 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。

5.2 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); //强制类型转换
        if (keyEvent->key() == Qt::Key_Tab) {
            qDebug() << "You press tab.";
            return true;
        }
    }
    return QWidget::event(e); //必要的，重新处理其他事件
}
```

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;
    // ...
}
```

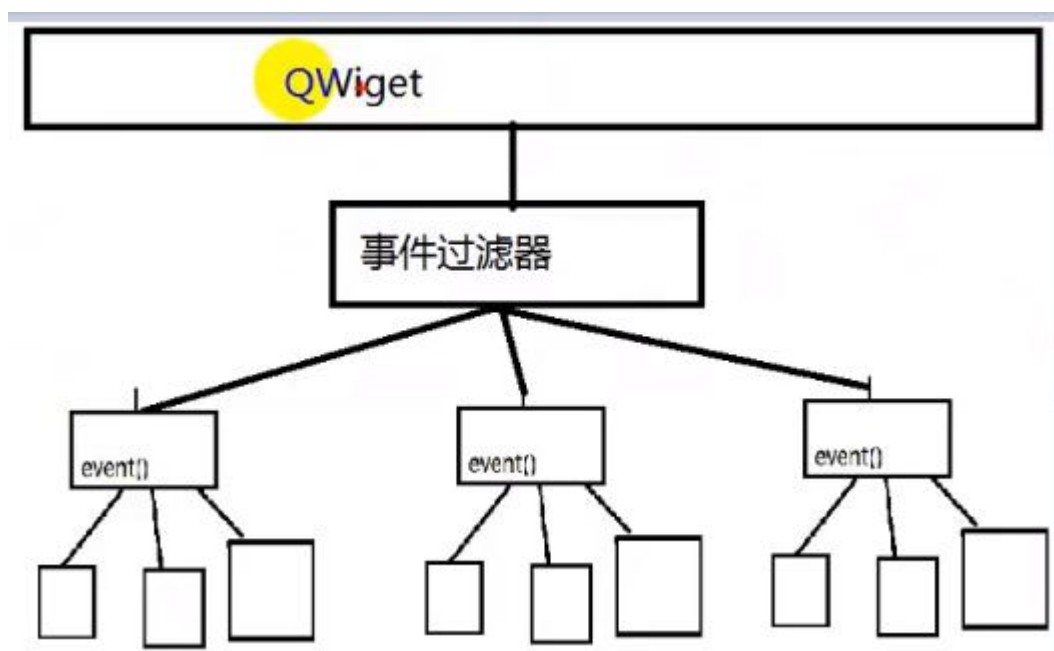
事实也的确如此。`timerEvent()` 和 `mouseMoveEvent()` 这样的函数, 就是我们前面章节所说的事件处理器 `event handler`。也就是说, `event()` 函数中实际是通过事件处理器来响应一个具体的事件。这相当于 `event()` 函数将具体事件的处理“委托”给具体的事件处理器。而这些事件处理器是 `protected virtual` 的, 因此, 我们重写了某一个事件处理器, 即可让 Qt 调用我们自己实现的版本。由此可以见, `event()` 是一个集中处理不同类型的事件的地方。如果你不想重写一大堆事件处理器, 就可以重写这个 `event()` 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問題, 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。

5.3 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，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。

5.4 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语句进行选择，其代表作就是 win32 API (**MFC 里面的**) 的 `WndProc()` 函数：

```
LRESULT CALLBACK WndProc(HWND hWnd,
```

```
UINT message,  
WPARAM wParam,  
LPARAM lParam)
```

在这个函数中，我们需要使用 switch 语句，选择 message 参数的类型进行处理，典型代码是：

```
switch(message)  
{  
    case WM_PAINT:  
        // ...  
        break;  
    case WM_DESTROY:  
        // ...  
        break;  
    ...  
}
```

- 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）。

5.5 不规则窗体

常见的窗体是各种方形的对话框，但有时候也需要非方形的窗体，如圆形，椭圆甚至是不规则形状的对话框。

实现步骤：

- 新建一个项目，比如项目名称叫做“ShapeWidget”，给此项目添加一个类“ShapeWidget”，基类选择“QWidget”。
- 为了使该不规则窗体可以通过鼠标随意拖拽，在类中重定义鼠标事件：`mousePressEvent()`、`mouseMoveEvent()`、以及绘制函数 `paintEvent()`
- “ShapeWidget”的构造函数部分是实现该不规则窗体的关键，添加具体代码如下：

```
//新建一个 QPixmap 对象
QPixmap pixmap;
//加载图片
pixmap.load(":/new/prefix1/image/sunny.png");
//固定窗口大小，将窗口大小设置为图片大小
setFixedSize( pixmap.width(), pixmap.height() );
//给窗口去掉边框，设置窗口的 flags
setWindowFlags(Qt::FramelessWindowHint | windowFlags() );
//设置透明背景
setAttribute(Qt::WA_TranslucentBackground);
```

- 重新实现鼠标事件和绘制函数

```
void ShareWidget::mousePressEvent(QMouseEvent *ev)
{
    if(ev->button() == Qt::LeftButton)
    {
        // 求出窗口移动之前的坐标
        m_dragPoint = ev->globalPos()-frameGeometry().topLeft();
    }
    if(ev->button() == Qt::RightButton)
    {
        // 鼠标右键关闭窗口
        close();
    }
}

void ShareWidget::mouseMoveEvent(QMouseEvent *ev)
{
    if(ev->buttons() & Qt::LeftButton)
    {
        // 如果是鼠标左键拖动，移动窗口
        move(ev->globalPos() - m_dragPoint);
    }
}

void ShareWidget::paintEvent(QPaintEvent *ev)
{
    Q_UNUSED(ev)
    QPainter painter(this);
    // 重新绘制图片
    painter.drawPixmap(0, 0, QPixmap(":/ButterFly"));
}
```

6 绘图和绘图设备

6.1 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** 提供了画笔（**QPainter**）在不同的设备上绘制的统一的接口。**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 **QPaintDevice** 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。

6.2 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。

6.2.1 QPixmap、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 或者其他设备上。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。

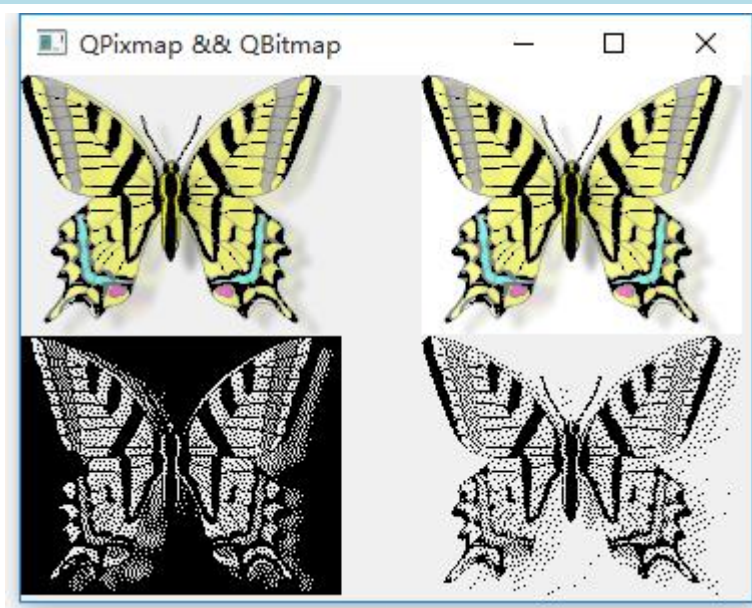
由于 QBitmap 色深小，因此只占用很少的存储空间，所以适合做光标文件和笔刷。

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。

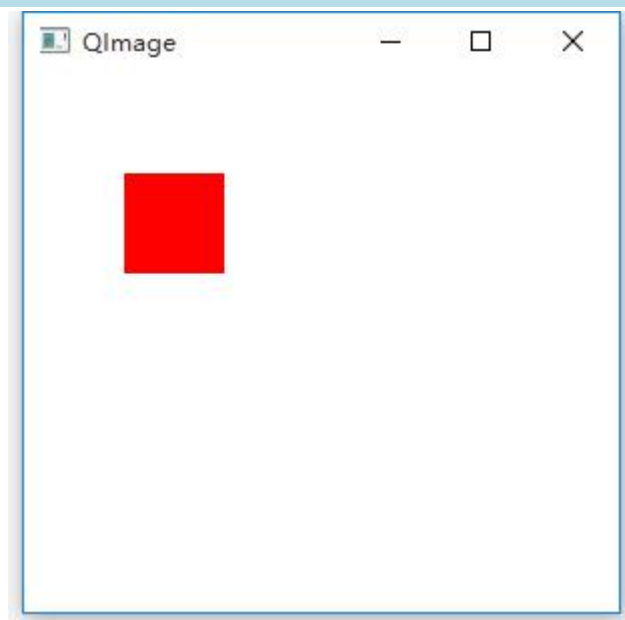
```
void PaintWidget::paintEvent(QPaintEvent *)
```

```
{
    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，使用这一方式可以大幅度提高 UI 响应速度。
- QImage 可通过 setPixel() 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
```

6.2.2 QPicture

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。

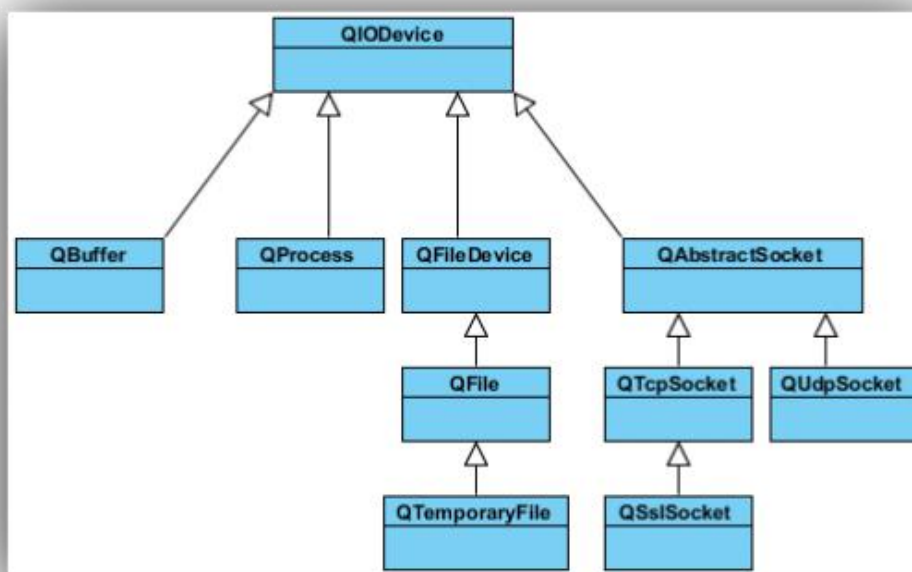
如果我们要记录下 QPainter 的命令，首先要使用 QPainter::begin() 函数，将 QPicture 实例作为参数传递进去，以便告诉系统开始记录，记录完毕后使用 QPainter::end() 命令终止。代码示例如下：

```
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();
}
```

7 文件系统

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。
- **QFile**: 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**、**QUdpSocket** 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，

7.1 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；
- `isExecutable()` 检查该文件是否是可执行文件等。
- `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"
```

7.2 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;
```

7.3 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
● <code>QIODevice::NotOpen</code>	未打开
● <code>QIODevice::ReadOnly</code>	以只读方式打开
● <code>QIODevice::WriteOnly</code>	以只写方式打开
● <code>QIODevice::ReadWrite</code>	以读写方式打开
● <code>QIODevice::Append</code>	以追加的方式打开， 新增加的内容将被追加到文件末尾
● <code>QIODevice::Truncate</code>	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。
● <code>QIODevice::Text</code>	在读取时，将行结束符转换成 <code>\n</code> ；在写入时， 将行结束符转换成本地格式，例如 Win32 平台上是 <code>\r\n</code>
● <code>QIODevice::Unbuffered</code>	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。

8 Socket 通信

Qt 中提供的所有的 `Socket` 类都是非阻塞的。

Qt 中常用的用于 `socket` 通信的套接字类：

- `QTcpServer`
用于 TCP/IP 通信，作为服务器端套接字使用
- `QTcpSocket`
用于 TCP/IP 通信，作为客户端套接字使用。
- `QUdpSocket`
用于 UDP 通信，服务器，客户端均使用此套接字。

8.1 TCP/IP

在 Qt 中实现 TCP/IP 服务器端通信的流程：

- 创建套接字
- 将套接字设置为监听模式

- 等待并接受客户端请求

可以通过 QTcpServer 提供的 void **newConnection()** 信号来检测是否有连接请求，如果有可以在对应的槽函数中调用 nextPendingConnection 函数获取到客户端的 Socket 信息（返回值为 QTcpSocket* 类型指针），通过此套接字与客户端之间进行通信。

- 接收或者向客户端发送数据

- 接收数据：使用 read () 或者 readAll () 函数

- 发送数据：使用 write () 函数

客户端通信流程：

- 创建套接字

- 连接服务器

可以使用 QTcpSocket 类的 **connectToHost ()** 函数来连接服务器。

- 向服务器发送或者接受数据

下面例子为简单的 TCP/IP 通信的实现：

服务器端

通过 Qt 提供的 QTcpServer 类实现服务器端的 socket 通信：

```
//----- tcpserver.h -----  
class TCPServer : public QMainWindow  
{  
    Q_OBJECT  
  
public:  
    explicit TCPServer(QWidget *parent = 0);  
    ~TCPServer();  
  
public slots:  
    void slotNewConnection();  
    void slotReadyRead();  
  
private:  
    Ui::TCPServer *ui;  
    // 负责监听的套接字
```

```
    QTcpServer* m_server;
    // 负责通信的套接字
    QTcpSocket* m_client;
};

//----- tcpserver.cpp -----
TCPServer::TCPServer(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::TCPServer),
    m_server(NULL),
    m_client(NULL)
{
    ui->setupUi(this);

    //创建套接字对象
    m_server = new QTcpServer(this);
    //将套接字设置为监听模式
    m_server->listen(QHostAddress::Any, 9999);

    //通过信号接收客户端请求
    connect(m_server, &QTcpServer::newConnection,
            this, &TCPServer::slotNewConnection);
}

TCPServer::~TCPServer()
{
    delete ui;
}

void TCPServer::slotNewConnection()
{
    if(m_client == NULL)
    {
        //处理客户端的连接请求
        m_client = m_server->nextPendingConnection();
        //发送数据
        m_client->write("服务器连接成功!!!");
        //连接信号，接收客户端数据
```

```
        connect(m_client, &QTcpSocket::readyRead,
                this, &TCPServer::slotReadyRead);
    }
}

void TCPServer::slotReadyRead()
{
    //接收数据
    QByteArray array = m_client->readAll();
    QMessageBox::information(this, "Client Message", array);
}
```

客户端

客户端通过使用 Qt 提供的 QTcpSocket 类可以方便的实现与服务器端的通信。

```
//----- tcpclient.h -----
class TCPClient : public QMainWindow
{
    Q_OBJECT

public:
    explicit TCPClient(QWidget *parent = 0);
    ~TCPClient();

public slots:
    void slotReadyRead();
    void slotSendMsg();

private:
    Ui::TCPClient *ui;
    QTcpSocket* m_client;
};

//----- tcpclient.cpp -----
TCPClient::TCPClient(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::TCPClient)
{
```



```
    ui->setUpUi(this);
    //创建套接字
    m_client = new QTcpSocket(this);
    //连接服务器
    m_client->connectToHost(QHostAddress("127.0.0.1"), 9999);

    //通过信号接收服务器数据
    connect(m_client, &QTcpSocket::readyRead,
            this, &TCPClient::slotReadyRead);

    //发送按钮
    connect(ui->btnSend, &QPushButton::clicked,
            this, &TCPClient::slotSendMsg);
}

TCPClient::~TCPClient()
{
    delete ui;
}

void TCPClient::slotReadyRead()
{
    //接收数据
    QByteArray array = m_client->readAll();
    QMessageBox::information(this, "Server Message", array);
}

void TCPClient::slotSendMsg()
{
    QString text = ui->textEdit->toPlainText();
    //发送数据
    m_client->write(text.toUtf8());
    ui->textEdit->clear();
}
```

8.2 UDP

使用 Qt 提供的 QUdpSocket 进行 UDP 通信。在 UDP 方式下，客户端并不与服务器建立连接，它只负责调用发送函数向服务器发送数据。类似的服务器也不从客

户端接收连接，只负责调用接收函数，等待来自客户端的数据的到达。

在 UDP 通信中，服务器端和客户端的概念已经显得有些淡化，两部分做的工作都大致相同：

- 创建套接字
- 绑定套接字

在 UDP 中如果需要接收数据则需要对套接字进行绑定，只发送数据则不需要对套接字进行绑定。

通过调用 `bind()` 函数将套接字绑定到指定端口上。

- 接收或者发送数据

- 接收数据：使用 `readDatagram()` 接收数据，函数声明如下：

```
qint64 readDatagram(char * data, qint64 maxSize,  
                    QHostAddress * address = 0, quint16 * port = 0)
```

参数：

- ◆ `data`：接收数据的缓存地址
- ◆ `maxSize`：缓存接收的最大字节数
- ◆ `address`：数据发送方的地址（一般使用提供的默认值）
- ◆ `port`：数据发送方的端口号（一般使用提供的默认值）

使用 `pendingDatagramSize()` 可以获取到将要接收的数据的大小，根据该函数返回值来准备对应大小的内存空间存放将要接收的数据。

- 发送数据：使用 `writeDatagram()` 函数发送数据，函数声明如下：

```
qint64 writeDatagram(const QByteArray & datagram,  
                     const QHostAddress & host, quint16 port)
```

参数：

- ◆ `datagram`：要发送的字符串
- ◆ `host`：数据接收方的地址
- ◆ `port`：数据接收方的端口号

广播

在使用 `QUdpSocket` 类的 `writeDatagram()` 函数发送数据的时候，其中第二个参数 `host` 应该指定为广播地址：`QHostAddress::Broadcast` 此设置相当于

```
QHostAddress("255.255.255.255")
```

使用 UDP 广播的特点：

- 使用 UDP 进行广播，局域网内的其他的 UDP 用户全部可以收到广播的消息
- UDP 广播只能在局域网范围内使用

组播

我们再使用广播发送消息的时候会发送给所有用户，但是有些用户是不想接受消息的，这时候我们就应该使用组播，接收方只有先注册到组播地址中才能收到组播消息，否则则接受不到消息。另外组播是可以在 Internet 中使用的。

在使用 QUdpSocket 类的 writeDatagram() 函数发送数据的时候，其中第二个参数 host 应该指定为组播地址，关于组播地址的分类：

- 224.0.0.0~224.0.0.255 为预留的组播地址（永久组地址），地址 224.0.0.0 保留不做分配，其它地址供路由协议使用；
- 224.0.1.0~224.0.1.255 是公用组播地址，可以用于 Internet；
- 224.0.2.0~238.255.255.255 为用户可用的组播地址（临时组地址），全网范围内有效；
- 239.0.0.0~239.255.255.255 为本地管理组播地址，仅在特定的本地范围内有效。

注册加入到组播地址需要使用 QUdpSocket 类的成员函数：

```
bool joinMulticastGroup(const QHostAddress & groupAddress)
```

8.3 TCP/IP 和 UDP 的区别

	TCP/IP	UDP
是否连接	面向连接	无连接
传输方式	基于流	基于数据报
传输可靠性	可靠	不可靠
传输效率	效率低	效率高
能否广播	不能	能

9 多线程

通常情况下，应用程序都是在一个线程中执行操作。但是，当调用一个耗时操作（例如，大批量 I/O 或大量矩阵变换等 CPU 密集操作）时，用户界面常常会冻结。而使用多线程可以解决这一问题。

多线程有以下几个优势：

- 提高应用程序响应速度。

这对于图形界面开发的程序尤为重要，当一个操作耗时很长时，整个系统都会等待这个操作，程序就不能响应键盘、鼠标、菜单等操作，而使用多线程技术可将耗时长操作置于一个新的线程，避免以上问题。

- 使多 CPU 系统更加有效。

当前线程数不大于 CPU 数目时，操作系统可以调度不同的线程运行于不同的 CPU 上。

- 改善程序结构。

一个既长又复杂的进程可以考虑分为多个线程，成为独立或半独立的运行部分，这样有利于代码的理解和维护。

多线程程序有以下几个特点：

- 多线程程序的行为无法预期，当多次执行程序时，每一次的结果都可能不同。
- 多线程的执行顺序无法保证，它与操作系统的调度策略和线程优先级等因素有关。
- 多线程的切换可能发生在任何时刻、任何地点。
- 多线程对代码的敏感度高，对代码的细微修改都可能产生意想不到的结果。

基于以上这些特点，为了有效的使用线程，开发人员必须对其进行控制。

9.1 线程介绍

在 Qt 中使用 QThread 来管理线程。下面来看一个简单的例子：

```
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
{
    QWidget *widget = new QWidget(this);
```

```
QVBoxLayout *layout = new QVBoxLayout;
widget->setLayout(layout);
QLCDNumber *lcdNumber = new QLCDNumber(this);
layout->addWidget(lcdNumber);
QPushButton *button = new QPushButton(tr("Start"), this);
layout->addWidget(button);
setCentralWidget(widget);

QTimer *timer = new QTimer(this);
connect(timer, &QTimer::timeout, [=]() {
    static int sec = 0;
    lcdNumber->display(QString::number(sec++));
});

connect(button, &QPushButton::clicked, [=]() {
    timer->start(1);
    for (int i = 0; i < 2000000000; i++);
    timer->stop();
});
}
```

我们的主界面有一个用于显示时间的 LCD 数字面板还有一个用于启动任务的按钮。程序的目的是用户点击按钮，开始一个非常耗时的运算（程序中我们以一个 2000000000 次的循环来替代这个非常耗时的的工作，在真实的程序中，这可能是一个网络访问，可能是需要复制一个很大的文件或者其它任务），同时 LCD 开始显示逝去的毫秒数。毫秒数通过一个计时器 QTimer 进行更新。计算完成后，计时器停止。这是一个很简单的应用，也看不出有任何问题。但是当我们开始运行程序时，问题就来了：点击按钮之后，程序界面直接停止响应，直到循环结束才开始重新更新。

有经验的开发者立即指出，这里需要使用线程。这是因为 Qt 中所有界面都是在 UI 线程中（也被称为主线程，就是执行了 QApplication::exec() 的线程），在这个线程中执行耗时的操作（比如那个循环），就会阻塞 UI 线程，从而让界面停止响应。界面停止响应，用户体验自然不好，不过更严重的是，有些窗口管理程序会检测到你的程序已经失去响应，可能会建议用户强制停止程序，这样一来

你的程序可能就此终止，任务再也无法完成。所以，为了避免这一问题，我们要使用 QThread 开启一个新的线程：

```
class WorkerThread : public QThread
{
    Q_OBJECT
public:
    WorkerThread(QObject *parent = 0)
        : QThread(parent)
    {
    }
protected:
    void run()
    {
        for (int i = 0; i < 1000000000; i++);
        emit done();
    }
signals:
    void done();
};

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
{
    QWidget *widget = new QWidget(this);
    QVBoxLayout *layout = new QVBoxLayout;
    widget->setLayout(layout);
    lcdNumber = new QLCDNumber(this);
    layout->addWidget(lcdNumber);
    QPushButton *button = new QPushButton(tr("Start"), this);
    layout->addWidget(button);
    setCentralWidget(widget);

    QTimer *timer = new QTimer(this);
    connect(timer, &QTimer::timeout, [=]() {
        static int sec = 0;
        lcdNumber->display(QString::number(sec++));
    });
};
```

```
WorkerThread *thread = new WorkerThread(this);
connect(thread, &WorkerThread::done, timer, &QTimer::stop);
connect(thread, &WorkerThread::finished,
        thread, &WorkerThread::deleteLater);
connect(button, &QPushButton::clicked, [=]() {
    timer->start(1);
    thread->start();
});
}
```

注意，我们增加了一个 WorkerThread 类。WorkerThread 继承自 QThread 类，重写了其 run() 函数。我们可以认为，**run() 函数就是新的线程需要执行的代码**。在这里就是要执行这个循环，然后发出计算完成的信号。**run() 是线程的入口，就像 main() 对于应用程序的作用，使用 QThread::start() 函数启动一个线程**（注意，这里不是 run() 函数）。再次运行程序，你会发现现在界面已经不会被阻塞了。另外，我们将 **WorkerThread::deleteLater() 函数与 WorkerThread::finished() 信号连接起来，当线程完成时，系统可以帮我们清除线程实例**。这里的 finished() 信号是系统发出的，与我们自定义的 done() 信号无关。

这是 Qt 线程的最基本的使用方式之一（确切的说，这种方式已经不大推荐使用，不过因为看起来很清晰，而且简单使用起来也没有什么问题，所以还是有必要介绍）。代码看起来很简单，不过，如果你认为 Qt 的多线程编程也很简单，那就大错特错了。Qt 多线程的优势设计使得它使用起来变得容易，但是坑很多，稍不留神就会被绊住，尤其是涉及到与 QObject 交互的情况。稍懂多线程开发的童鞋都会知道，调试多线程开发简直就是煎熬。

9.2 多线程的使用

在 Qt4.7 及以后版本推荐使用以下的工作方式。其主要特点就是利用 Qt 的事件驱动特性，将**需要在次线程中处理的业务放在独立的模块（类）中，由主线程创建完该对象后，将其移交给指定的线程，且可以将多个类似的对象移交给同一个线程**。在这个例子中，信号由主线程的 QTimer 对象发出，之后 Qt 会将关联

的事件放到 worker 所属线程的事件队列。由于队列连接的作用，在不同线程间连接信号和槽是很安全的。

示例代码如下：

```
class Worker : public QObject
{
    Q_OBJECT
private slots:
    void onTimeout()
    {
        qDebug() << "Worker::onTimeout get called from?: "
                << QThread::currentThreadId();
    }
};

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    qDebug() << "From main thread: " << QThread::currentThreadId();

    QThread t;
    QTimer timer;
    Worker worker;

    QObject::connect(&timer, SIGNAL(timeout()),
                    &worker, SLOT(onTimeout()));

    // 启动定时器
    timer.start(1000);
    // 将类对象移交个线程
    worker.moveToThread(&t);
    // 启动线程
    t.start();

    return a.exec();
}
```

关于 QObject 类的 connect 函数最后一个参数，连接类型：

- 自动连接(AutoConnection)，默认的连接方式。
 - 如果信号与槽，也就是发送者与接受者在同一线程，等同于直接连接；

- 如果发送者与接受者处在不同线程，等同于队列连接。
- 直接连接(DirectConnection)
当信号发射时，槽函数立即直接调用。**无论槽函数所属对象在哪个线程，槽函数总在发送者所在线程执行。**
- 队列连接(QueuedConnection)
当控制权回到接受者所在线程的事件循环时，槽函数被调用。**槽函数在接受者所在线程执行。**

总结：

- * 队列连接：槽函数在接受者所在线程执行。
- * 直接连接：槽函数在发送者所在线程执行。
- * 自动连接：二者不在同一线程时，等同于队列连接

多线程使用过程中注意事项：

- 线程不能操作 UI 对象（从 Qwidget 直接或间接派生的窗口对象）
- 需要移动到子线程中处理的模块类，创建的对象的时候不能指定父对象。

9.3 使用线程绘图

根据前面讲过的知识,实现以下案例:

在窗口中有一个按钮,当点击按钮之后,在线程中绘制一张图片,然后将绘制好的图片显示到当前窗口中。

实现步骤:

将需要房屋线程中的操作放入单独的一个类中去处理:

```
class Work : public QObject
{
    Q_OBJECT
public:
    Work(QObject *parent = 0) : QObject(parent)
    {
```

```
    }

public slots:
    void slotDrawImage()
    {
        QImage image(600, 600, QImage::Format_ARGB32);
        QPainter painter(&image);
        QPoint pt[] =
        {
            QPoint(qrand()%590, qrand()%590),
            QPoint(qrand()%590, qrand()%590),
            QPoint(qrand()%590, qrand()%590),
            QPoint(qrand()%590, qrand()%590),
            QPoint(qrand()%590, qrand()%590),
        };
        painter.drawPolygon(pt, 5);
        // 将画好的图片通过信号发送出去
        emit ImageDone(image);
    }
signals:
    void ImageDone(QImage image);
};
```

在 UI 线程中(主线程)中创建 Work 类对象，并调用 moveToThread 函数将操作移入到子线程中取处理。

```
// 头文件
class MyWidget : public QWidget
{
    Q_OBJECT

public:
    explicit MyWidget(QWidget *parent = 0);
    ~MyWidget();

protected:
    void paintEvent(QPaintEvent *);

private:
```

```
    Ui::MyWidget *ui;
    QImage m_image;
};

// 源文件
MyWidget::MyWidget(QWidget *parent) :
    QWidget(parent),
    ui(new Ui::MyWidget)
{
    ui->setupUi(this);

    Work* pWork = new Work;
    connect(ui->draw, &QPushButton::clicked,
            pWork, &Work::slotDrawImage);

    QThread * pthread = new QThread(this);
    // 将操作移入子线程中处理
    pWork->moveToThread(pthread);
    // 启动子线程
    pthread->start();

    connect(pWork, &Work::ImageDone, [=](QImage image)
    {
        // 保存图片
        m_image = image;
        // 刷新窗口
        update();
    });

    connect(this, &MyWidget::destroyed, [=]()
    {
        // 退出线程
        pthread->quit();
        pthread->wait();
        delete pWork;
    });
}
```

如果需要在窗口中绘制图形,那么就需要重写 paintEvent 事件处理函数。通过

QPainter 对象将子线程中绘制的图片画到当前窗口中。如果需要刷新窗口可以调用 update () 函数，时间处理器会自动被调用。

```
void MyWidget::paintEvent(QPaintEvent *e)
{
    QPainter p(this);
    p.drawImage(0, 0, m_image);
}
```

10 数据库操作

10.1 数据库操作

Qt 提供了 QSql 模块来提供平台独立的基于 SQL 的数据库操作。这里我们所说的“平台独立”，既包括操作系统平台，有包括各个数据库平台。另外，我们强调了“基于 SQL”，因为 NoSQL 数据库至今没有一个通用查询方法，所以不可能提供一种通用的 NoSQL 数据库的操作。Qt 的数据库操作还可以很方便的与 model/view 架构进行整合。通常来说，我们对数据库的操作更多地在于对数据库表的操作，而这正是 model/view 架构的长项。

Qt 使用 QSqlDatabase 表示一个数据库连接。更底层上，Qt 使用驱动(drivers)来与不同的数据库 API 进行交互。Qt 桌面版本提供了如下几种驱动：

驱动	数据库
QDB2	IBM DB2 (7.1 或更新版本)
QIBASE	Borland InterBase
QMYSQL	MySQL
QOCI	Oracle Call Interface Driver
QODBC	Open Database Connectivity (ODBC) - Microsoft SQL Server 及其它兼容 ODBC 的 数据库
QPSQL	PostgreSQL (7.3 或更新版本)
QSQLITE2	SQLite 2
QSQLITE	SQLite 3

QSYMSQL	针对 Symbian 平台的 SQLite 3
QTDS	Sybase Adaptive Server (自 Qt 4.7 起废除)

不过, 由于受到协议的限制, Qt 开源版本并没有提供上面所有驱动的二进制版本, 而仅仅以源代码的形式提供。通常, Qt 只默认搭载 QSqlite 驱动 (这个驱动实际还包括 Sqlite 数据库, 也就是说, 如果需要使用 Sqlite 的话, 只需要该驱动即可)。我们可以选择把这些驱动作为 Qt 的一部分进行编译, 也可以当作插件编译。

如果习惯于使用 SQL 语句, 我们可以选择 QSqlQuery 类; 如果只需要使用高层次的数据库接口 (不关心 SQL 语法), 我们可以选择使用 QSqlTableModel 类。在使用时, 我们可以通过

```
QSqlDatabase::drivers();
```

找到系统中所有可用的数据库驱动的名字列表。我们只能使用出现在列表中的驱动。由于默认情况下, QtSql 是作为 Qt 的一个模块提供的。为了使用有关数据库的类, 我们必须早 .pro 文件中添加这么一句:

```
QT += sql
```

这表示, 我们的程序需要使用 Qt 的 core、gui 以及 sql 三个模块。注意, 如果需要同同时使用 Qt4 和 Qt5 编译程序, 通常我们的 .pro 文件是这样的:

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

这两句也很明确: Qt 需要加载 core、gui 和 sql 三个模块, 如果主版本大于 4, 则再添加 widgets 模块。

下面来看一个简单的函数:

```
bool connect(const QString &dbName)
{
    QSqlDatabase db = QSqlDatabase::addDatabase("SQLITE");
    // db.setHostName("host");
    // db.setDatabaseName("dbname");
    // db.setUserName("username");
    // db.setPassword("password");
}
```

```
db.setDatabaseName(dbName);
if (!db.open()) {
    QMessageBox::critical(0, QObject::tr("Database Error"),
                          db.lastError().text());

    return false;
}
return true;
}
```

我们使用 `connect()` 函数创建一个数据库连接。我们使用 `QSqlDatabase::addDatabase()` 静态函数完成这一请求，也就是创建了一个 `QSqlDatabase` 实例。注意，数据库连接使用自己的名字进行区分，而不是数据库的名字。例如，我们可以使用下面的语句：

```
QSqlDatabase db=QSqlDatabase::addDatabase("SQLITE",
                                           QString("con%1").arg(dbName));
```

此时，我们是使用 `addDatabase()` 函数的第二个参数来给这个数据库连接一个名字。在这个例子中，用于区分这个数据库连接的名字是 `QString("conn%1").arg(dbName)`，而不是“SQLITE”。这个参数是可选的，如果不指定，系统会给出一个默认的名字 `QSqlDatabase::defaultConnection`，此时，Qt 会创建一个默认的连接。如果你给出的名字与已存在的名字相同，新的连接会替换掉已有的连接。通过这种设计，我们可以为一个数据库建立多个连接。

我们这里使用的是 `sqlite` 数据库，只需要指定数据库名字即可。如果是数据库服务器，比如 `MySQL`，我们还需要指定主机名、端口号、用户名和密码，这些语句使用注释进行了简单的说明。

接下来我们调用了 **QSqlDatabase 类的 `open()` 函数，打开这个数据库连接**。通过检查 `open()` 函数的返回值，我们可以判断数据库是不是正确打开。

QtSql 模块中的类大多具有 `lastError()` 函数，用于检查最新出现的错误。如果你发现数据库操作有任何问题，应该使用这个函数进行错误的检查。这一点我们也在上面的代码中进行了体现。当然，这只是最简单的实现，一般来说，更好的设计是，不要在数据库操作中混杂界面代码（并且将这个 `connect()` 函数放在一

个专门的数据库操作类中)。

接下来我们可以在 main() 函数中使用这个 connect() 函数：

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    if (connect("demo.db")) {
        QSqlQuery query;
        if (!query.exec("CREATE TABLE student ("
                        "id INT PRIMARY KEY AUTOINCREMENT,"
                        "name VARCHAR(255),"
                        "age INT)"))
        {
            QMessageBox::critical(0,
                                   QObject::tr("Database Error"),
                                   query.lastError().text());
            return 1;
        }
    } else {
        return 1;
    }
    return a.exec();
}
```

main() 函数中，我们调用这个 connect() 函数打开数据库。如果打开成功，我们通过一个 QSqlQuery 实例执行了 SQL 语句。同样，我们使用其 lastError() 函数检查了执行结果是否正确。

注意这里的 QSqlQuery 实例的创建。我们并没有指定是为哪一个数据库连接创建查询对象，此时，系统会使用默认的连接，也就是使用没有第二个参数的 addDatabase() 函数创建的那个连接（其实就是名字为 QSqlDatabase::defaultConnection 的默认连接）。如果没有这么一个连接，系统就会报错。也就是说，如果没有默认连接，我们在创建 QSqlQuery 对象时必须指明是哪一个 QSqlDatabase 对象，也就是 addDatabase() 的返回值。

我们还可以通过使用 QSqlQuery::isActive() 函数检查语句执行正确与否。如果

QSqlQuery 对象是活动的，该函数返回 true。所谓“活动”，就是指该对象成功执行了 exec() 函数，但是还没有完成。这里需要注意的是，如果存在一个活动的 SELECT 语句，某些数据库系统不能成功完成 connect() 或者 rollback() 函数的调用。此时，我们必须首先将活动的 SELECT 语句设置成不活动的。

创建过数据库表 student 之后，我们开始插入数据，然后将其独取出来：

```
if (connect("demo.db")) {
    QSqlQuery query;
    query.prepare("INSERT INTO student (name, age) VALUES (?, ?)");
    QVariantList names;
    names << "Tom" << "Jack" << "Jane" << "Jerry";
    query.addBindValue(names);
    QVariantList ages;
    ages << 20 << 23 << 22 << 25;
    query.addBindValue(ages);
    if (!query.execBatch()) {
        QMessageBox::critical(0, QObject::tr("Database Error"),
                               query.lastError().text());
    }

    query.exec("SELECT name, age FROM student");
    while (query.next()) {
        QString name = query.value(0).toString();
        int age = query.value(1).toInt();
        qDebug() << name << ": " << age;
    }
} else {
    return 1;
}
```

依旧连接到我们创建的 demo.db 数据库。我们需要插入多条数据，此时可以使用 QSqlQuery::exec() 函数一条一条插入数据，但是这里我们选择了另外一种方法：批量执行。首先，我们使用 **QSqlQuery::prepare() 函数对这条 SQL 语句进行预处理，问号 ? 相当于占位符，预示着以后我们可以使用实际数据替换这些位置**。简单说明一下，预处理是数据库提供的一种特性，它会将 SQL 语句进行编译，性能和安全性都要优于普通的 SQL 处理。在上面的代码中，我们使用一个字符串列表 names 替换掉第一个问号的位置，一个整型列表 ages 替换掉第

二个问号的位置，利用 `QSqlQuery::addBindValue()` 我们将实际数据绑定到这个预处理的 SQL 语句上。需要注意的是，`names` 和 `ages` 这两个列表里面的数据需要一一对应。然后我们调用 `QSqlQuery::execBatch()` 批量执行 SQL，之后结束该对象。这样，插入操作便完成了。

另外说明一点，我们这里使用了 ODBC 风格的 ? 占位符，同样，我们也可以使用 Oracle 风格的占位符：

```
query.prepare("INSERT INTO student (name, age) VALUES (:name, :age)");
```

此时，我们就需要使用

```
query.bindValue(":name", names);  
query.bindValue(":age", ages);
```

进行绑定。Oracle 风格的绑定最大的好处是，绑定的名字和值很清晰，与顺序无关。但是这里需要注意，`bindValue()` 函数只能绑定一个位置。比如

```
query.prepare("INSERT INTO test (name1, name2)  
              VALUES (:name, :name)");  
  
// ...  
query.bindValue(":name", name);
```

只能绑定第一个 `:name` 占位符，不能绑定到第二个。

接下来我们依旧使用同一个查询对象执行一个 `SELECT` 语句。如果存在查询结果，`QSqlQuery::next()` 会返回 `true`，直到到达结果最末，返回 `false`，说明遍历结束。我们利用这一点，使用 `while` 循环即可遍历查询结果。使用 `QSqlQuery::value()` 函数即可按照 `SELECT` 语句的字段顺序获取到对应的数据库存储的数据。

```
query.exec("select name, age from student");  
while (query.next())  
{  
    QString name = query.value(0);  
    QString age = query.value(1);  
}
```

对于数据库事务的操作，我们可以使用 `QSqlDatabase::transaction()` 开启事务，`QSqlDatabase::commit()` 或者 `QSqlDatabase::rollback()` 结束事务。使用 `QSqlDatabase::database()` 函数则可以根据名字获取所需要的数据库连接。

10.2 使用模型操作数据库

上一节我们使用 SQL 语句完成了对数据库的常规操作，包括简单的 CREATE、SELECT 等语句的使用。我们也提到过，Qt 不仅提供了这种使用 SQL 语句的方式，还提供了一种基于模型的更高级的处理方式。这种基于 QSqlTableModel 的模型处理更为高级，如果对 SQL 语句不熟悉，并且不需要很多复杂的查询，这种 QSqlTableModel 模型基本可以满足一般的需求。本节我们将介绍 QSqlTableModel 的一般使用，对比 SQL 语句完成对数据库的增删改查等的操作。值得注意的是，QSqlTableModel 并不一定非得结合 QListView 或 QTableView 使用，我们完全可以用其作一般性处理。

查询操作

首先我们来看看如何使用 QSqlTableModel 进行 SELECT 操作：

```
if (connect("demo.db")) {
    QSqlTableModel model;
    model.setTable("student");
    model.setFilter("age > 20 and age < 25");
    if (model.select()) {
        for (int i = 0; i < model.rowCount(); ++i) {
            QSqlRecord record = model.record(i);
            QString name = record.value("name").toString();
            int age = record.value("age").toInt();
            qDebug() << name << ": " << age;
        }
    }
} else {
    return 1;
}
```

我们依旧使用了上一节的 connect() 函数。接下来我们创建了 QSqlTableModel 实例，

- setTable() 函数设置所需要操作的表格；
- setFilter() 函数则是添加过滤器，也就是 WHERE 语句所需要的部分。

例如上面代码中的操作实际相当于 SQL 语句

```
SELECT * FROM student WHERE age > 20 and age < 25
```

使用 `QSqlTableModel::select()` 函数进行操作，也就是执行了查询操作。如果查询成功，函数返回 `true`，由此判断是否发生了错误。如果没有错误，我们使用 `record()` 函数取出一行记录，该记录是以 `QSqlRecord` 的形式给出的，而 `QSqlRecord::value()` 则取出一个列的实际数据值。注意，由于 `QSqlTableModel` 没有提供 `const_iterator` 遍历器，因此不能使用 `foreach` 宏进行遍历。

另外需要注意，由于 `QSqlTableModel` 只是一种高级操作，肯定没有实际 SQL 语句方便。具体来说，**我们使用 `QSqlTableModel` 只能进行 `SELECT *` 的查询，不能只查询其中某些列的数据。**

插入操作

下面一段代码则显示了如何使用 `QSqlTableModel` 进行插入操作：

```
QSqlTableModel model;
model.setTable("student");
int row = 0;
model.insertRows(row, 1);
model.setData(model.index(row, 1), "Cheng");
model.setData(model.index(row, 2), 24);
model.submitAll();
```

插入也很简单：`model.insertRows(row, 1)`；说明我们想在索引 0 的位置插入 1 行新的数据。使用 **`setData()` 函数则开始准备实际需要插入的数据**。注意这里我们向 `row` 的第一个位置写入 `Cheng`（通过 `model.index(row, 1)`，回忆一下，我们把 `model` 当作一个二维表，这个坐标相当于第 `row` 行第 1 列），其余以此类推。最后，调用 **`submitAll()` 函数提交所有修改**。这里执行的操作可以用如下 SQL 表示：

```
INSERT INTO student (name, age) VALUES ('Cheng', 24)
```

更新操作

当我们取出了已经存在的数据后，对其进行修改，然后重新写入数据库，即完成了一次更新操作：

```
QSqlTableModel model;
model.setTable("student");
model.setFilter("age = 25");
if (model.select()) {
    if (model.rowCount() == 1) {
        QSqlRecord record = model.record(0);
        record.setValue("age", 26);
        model.setRecord(0, record);
        model.submitAll();
    }
}
```

这段代码中，我们首先找到 `age = 25` 的记录，然后将 `age` 重新设置为 26，存入相同的位置（在这里都是索引 0 的位置），提交之后完成一次更新。当然，我们也可以类似其它模型一样的设置方式：`setData()` 函数。具体代码片段如下：

```
if (model.select()) {
    if (model.rowCount() == 1) {
        model.setData(model.index(0, 2), 26);
        model.submitAll();
    }
}
```

注意我们的 `age` 列是第 3 列，索引值为 2，因为前面还有 `id` 和 `name` 两列。这里的更新操作则可以用如下 SQL 表示：

```
UPDATE student SET age = 26 WHERE age = 25
```

删除操作

删除操作同更新类似：

```
QSqlTableModel model;
model.setTable("student");
model.setFilter("age = 25");
if (model.select()) {
    if (model.rowCount() == 1) {
        model.removeRows(0, 1);
        model.submitAll();
    }
}
```

如果使用 SQL 则是：

```
DELETE FROM student WHERE age = 25
```

当我们看到 `removeRows()` 函数就应该想到：我们可以一次删除多行。事实也正是如此，这里不再赘述。

10.3 可视化显示数据库数据

前面我们用了两个章节介绍了 Qt 提供的两种操作数据库的方法。显然，使用 `QSqlQuery` 的方式更灵活，功能更强大，而使用 `QSqlTableModel` 则更简单，更方便与 `model/view` 结合使用（数据库应用很大一部分就是以表格形式显示出来，这正是 `model/view` 的强项）。本章我们简单介绍使用 `QSqlTableModel` 显示数据的方法。当然，我们也可以选择使用 `QSqlQuery` 获取数据，然后交给 `view` 显示，而这需要自己给 `model` 提供数据。

我们还是使用前面一直在用的 `student` 表，直接来看代码：

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    if (connect("demo.db")) {
        QSqlTableModel *model = new QSqlTableModel;
        model->setTable("student");
        model->setSort(1, Qt::AscendingOrder);
        model->setHeaderData(1, Qt::Horizontal, "Name");
        model->setHeaderData(2, Qt::Horizontal, "Age");
        model->select();

        QTableView *view = new QTableView;
        view->setModel(model);

        view->setSelectionMode(QAbstractItemView::SingleSelection);

        view->setSelectionBehavior(QAbstractItemView::SelectRows);
        // view->setColumnHidden(0, true);
        view->resizeColumnsToContents();
        view->setEditTriggers(QAbstractItemView::NoEditTriggers);
    }
}
```

```
        QHeaderView *header = view->horizontalHeader();
        header->setStretchLastSection(true);

        view->show();
    } else {
        return 1;
    }
    return a.exec();
}
```

这里的 `connect()` 函数还是我们前面使用过的 (11.1), 我们在 `main()` 函数中创建了 `QSqlTableModel` 对象, 使用 `student` 表。`student` 表有三列: `id`, `name` 和 `age`, 我们选择按照 `name` 排序, 使用 `setSort()` 函数达到这一目的。然后我们设置每一列的列头。这里我们只使用了后两列, 第一列没有设置, 所以依旧显示为列名 `id`。

在设置好 `model` 之后, 我们又创建了 `QTableView` 对象作为视图。注意这里的设置: 单行选择, 按行选择。`resizeColumnsToContents()` 说明每列宽度适配其内容; `setEditTriggers()` 则禁用编辑功能。最后, 我们设置最后一列要充满整个窗口。我们的代码中有一行注释, 设置第一列不显示。

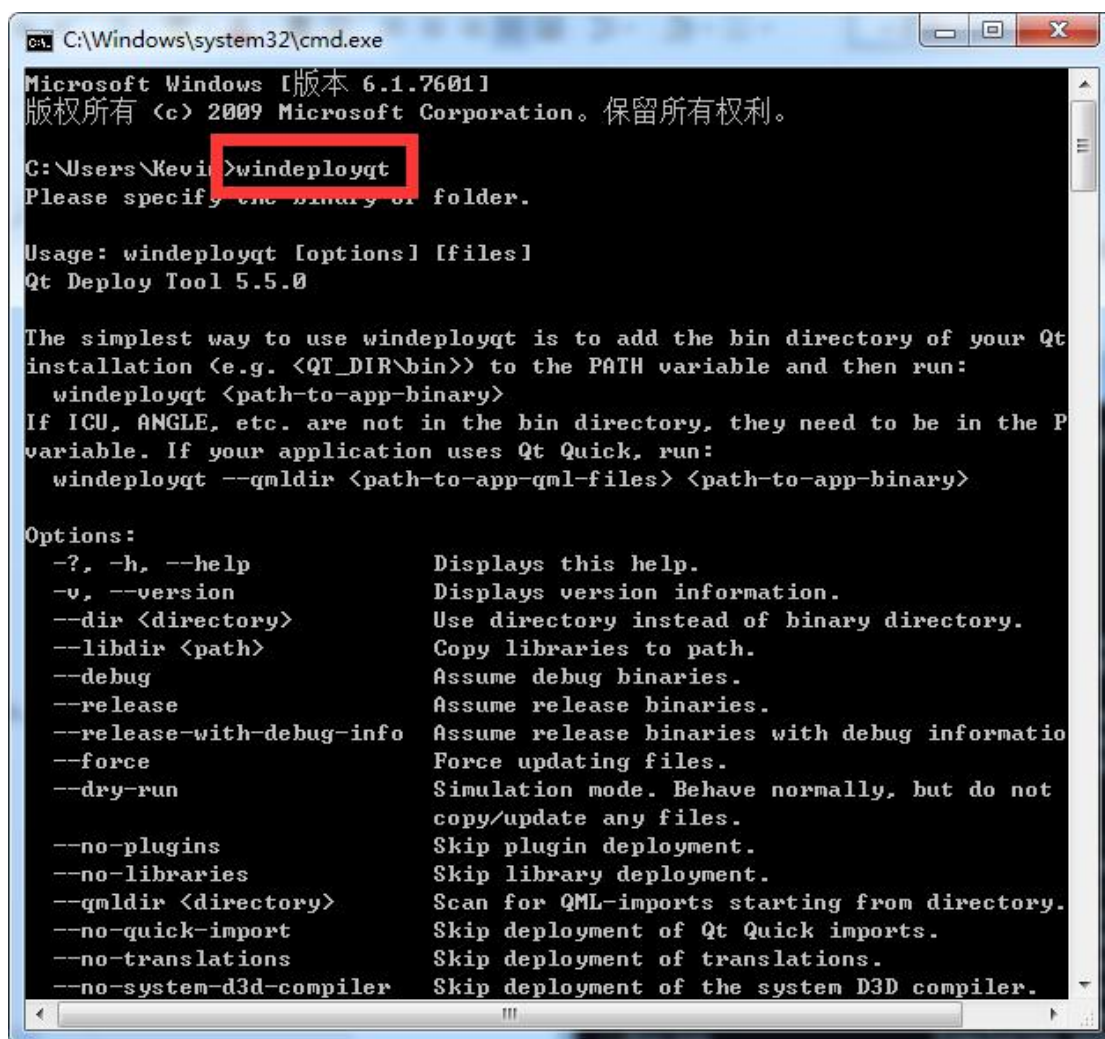
11 Qt 程序打包

Qt 的应用程序编译出来之后, 将单独的 `exe` 程序拿到其他 PC 上运行是运行不起来的, 会提示缺少对应的动态链接库。我们需要去 Qt 的安装目录下找到所有的 Qt 程序运行时所依赖的, 将他们和 `exe` 程序放到同一目录下, 程序才可以执行。根据上边的描述我们可以想象的到, 如果手动去寻找应用程序依赖的动态库, 这是一件非常麻烦的事情。其实我们完全没有必要这么辛苦, Qt 给我们提供了一个寻找依赖项的工具 `windeployqt`

`Windeployqt` 的使用方法:

如果我们一件配置好了环境变量, 在 `dos` 下输入 `windeployqt` 会有相应的信息输出, 否则需要指定该工具的完全路径才能够正常使用, 例如:

```
C:\Qt\Qt5.5.0\5.5\mingw492_32\bin\windeployqt
```



```
C:\Windows\system32\cmd.exe
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

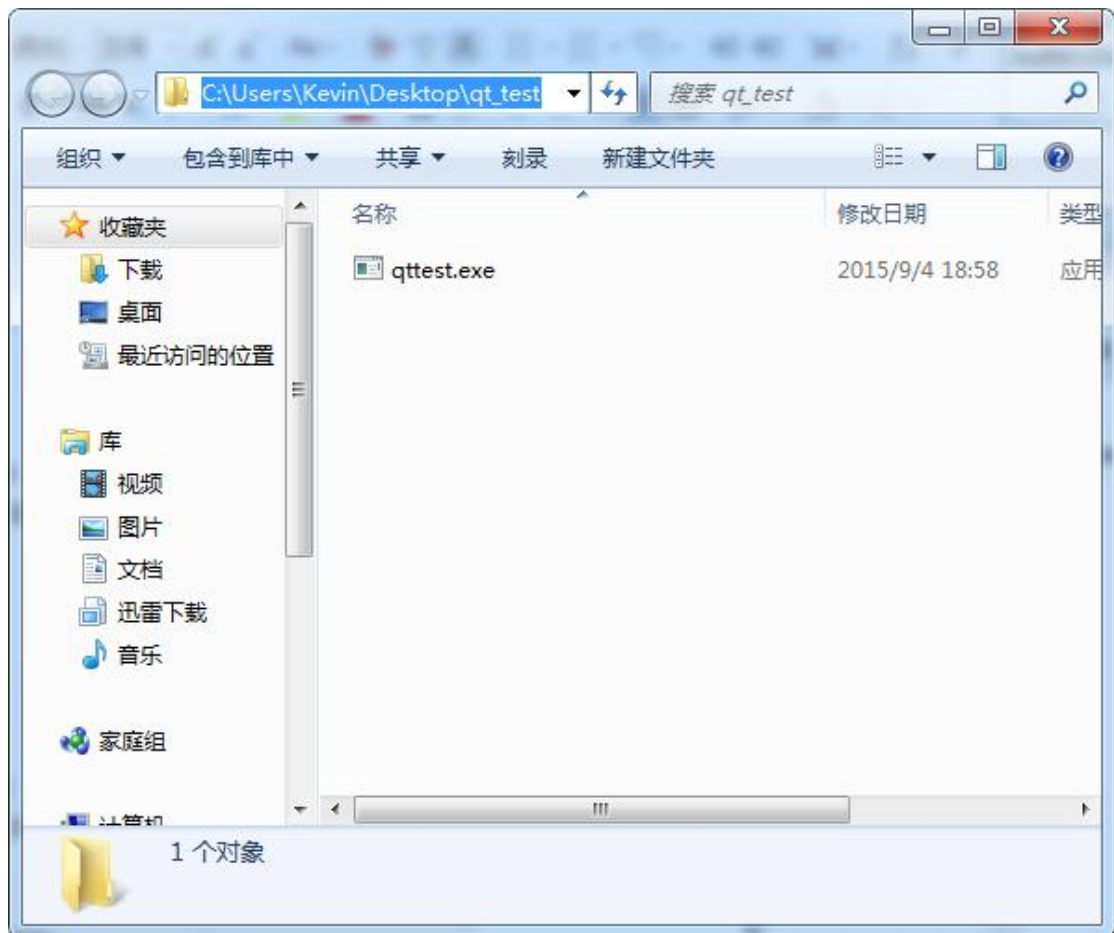
C:\Users\Kevin>windeployqt
Please specify the binary or folder.

Usage: windeployqt [options] [files]
Qt Deploy Tool 5.5.0

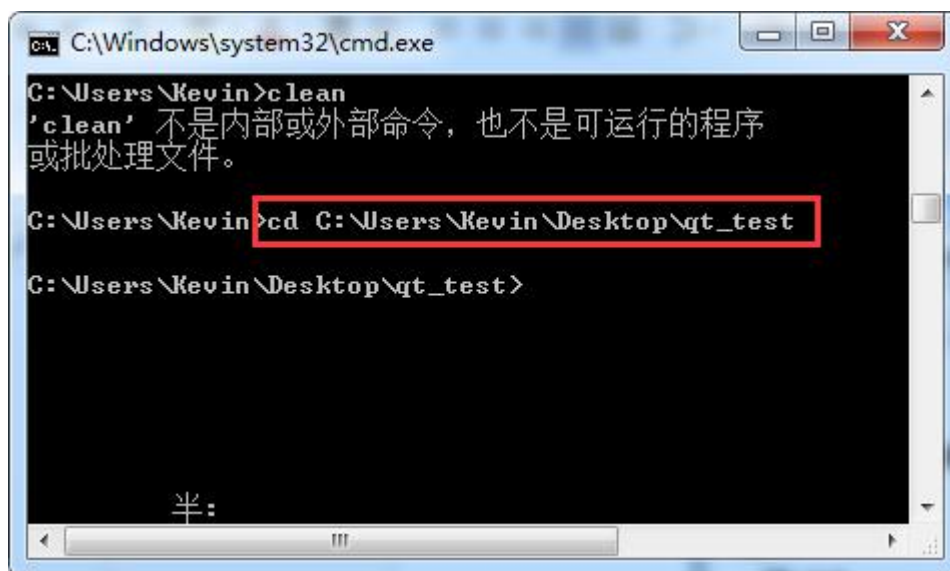
The simplest way to use windeployqt is to add the bin directory of your Qt
installation (e.g. <QT_DIR\bin>) to the PATH variable and then run:
    windeployqt <path-to-app-binary>
If ICU, ANGLE, etc. are not in the bin directory, they need to be in the P
variable. If your application uses Qt Quick, run:
    windeployqt --qmldir <path-to-app-qml-files> <path-to-app-binary>

Options:
  -?, -h, --help           Displays this help.
  -v, --version            Displays version information.
  --dir <directory>        Use directory instead of binary directory.
  --libdir <path>          Copy libraries to path.
  --debug                 Assume debug binaries.
  --release               Assume release binaries.
  --release-with-debug-info Assume release binaries with debug informatio
  --force                 Force updating files.
  --dry-run               Simulation mode. Behave normally, but do not
                           copy/update any files.
  --no-plugins            Skip plugin deployment.
  --no-libraries          Skip library deployment.
  --qmldir <directory>    Scan for QML-imports starting from directory.
  --no-quick-import       Skip deployment of Qt Quick imports.
  --no-translations        Skip deployment of translations.
  --no-system-d3d-compiler Skip deployment of the system D3D compiler.
```

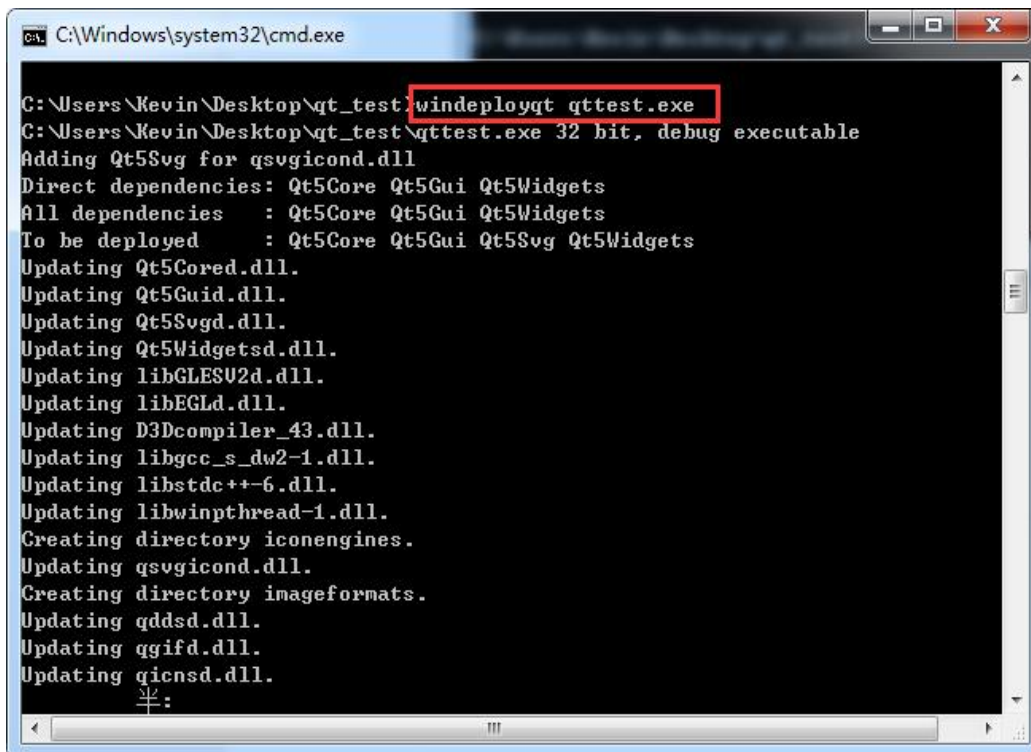
- 把新生成的 exe 文件放到指定的目录下:
例如: C:\Users\Kevin\Desktop\qt_test



- 在控制台窗口中通命令进入到上述目录中



- 执行命令 windeployqt 应用程序名 (qtttest.exe)



```
C:\Windows\system32\cmd.exe

C:\Users\Kevin\Desktop\qt_test>windeployqt qttest.exe
C:\Users\Kevin\Desktop\qt_test>qttest.exe 32 bit, debug executable
Adding Qt5Svg for qsvgicon.dll
Direct dependencies: Qt5Core Qt5Gui Qt5Widgets
All dependencies   : Qt5Core Qt5Gui Qt5Widgets
To be deployed     : Qt5Core Qt5Gui Qt5Svg Qt5Widgets
Updating Qt5Cored.dll.
Updating Qt5Guid.dll.
Updating Qt5Svgd.dll.
Updating Qt5Widgets.dll.
Updating libGLESv2.dll.
Updating libEGL.dll.
Updating D3Dcompiler_43.dll.
Updating libgcc_s_dw2-1.dll.
Updating libstdc++-6.dll.
Updating libwinpthread-1.dll.
Creating directory iconengines.
Updating qsvgicon.dll.
Creating directory imageformats.
Updating qdds.dll.
Updating qgif.dll.
Updating qicns.dll.
半:
```

应用程序所需的附加依赖项就会全部拷贝到我们指定的目录中

