

《Qt Creator 快速入门》第 3 版 实验讲义

2017 年 2 月

目 录

实验 1 Qt 开发环境的搭建	- 2 -
目的与要求.....	- 2 -
实验准备.....	- 2 -
实验内容.....	- 2 -
实验 2 编译和发布 Qt 程序	- 12 -
目的与要求.....	- 12 -
实验准备.....	- 12 -
实验内容.....	- 12 -
实验 3 使用 Qt 资源文件	- 20 -
目的与要求.....	- 20 -
实验准备.....	- 20 -
实验内容.....	- 20 -
实验 4 创建登陆对话框	- 26 -
目的与要求.....	- 26 -
实验准备.....	- 26 -
实验内容.....	- 26 -
实验 5 定时器和随机数	- 32 -
目的与要求.....	- 32 -
实验准备.....	- 32 -
实验内容.....	- 32 -
实验 6 编译 MySQL 数据库驱动	- 35 -
目的与要求.....	- 35 -
实验准备.....	- 35 -
实验内容.....	- 35 -
实验 7 数据库基本操作	- 44 -
目的与要求.....	- 44 -
实验准备.....	- 44 -
实验内容.....	- 44 -
实验 8 Qt 数据库应用编程（综合设计）	- 50 -
目的与要求.....	- 50 -
实验准备.....	- 50 -
实验内容.....	- 50 -

实验 1 Qt 开发环境的搭建

目的与要求

- (1) 掌握 Qt SDK 安装方法
- (2) 掌握 Qt Creator 的基本使用方法
- (3) 了解 Qt Creator 的界面布局
- (4) 了解 Qt 相关工具软件

实验准备

- (1) 对 Qt 及 Qt Creator 有基本了解
- (2) 了解 MinGW
- (3) 下载 Qt 5.6.1 安装包

实验内容

1. 安装 Qt Creator

(1) 双击运行 qt-opensource-windows-x86-mingw492-5.6.1-1 安装包，将出现如图 1.1 所示的安装向导欢迎界面，这里提示如果没有 Qt 帐号，可以在后面的步骤中创建一个。单击“下一步”按钮。

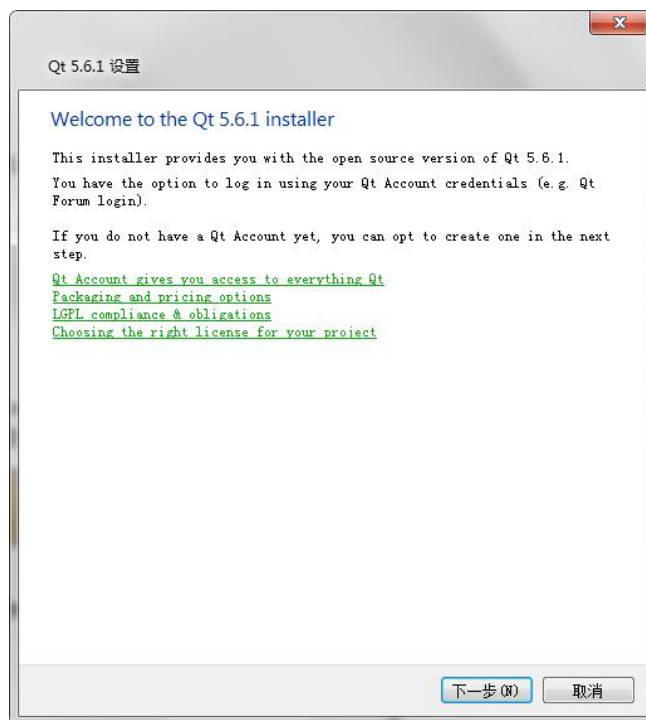


图 1.1 Qt Creator 安装向导

(2) 这里可以直接登录 Qt 帐号，如果没有可以在这里注册一个，不过登录或注册与否都不影响安装，所以直接单击 Skip 按钮跳过这一步即可，如图 1.2 所示。

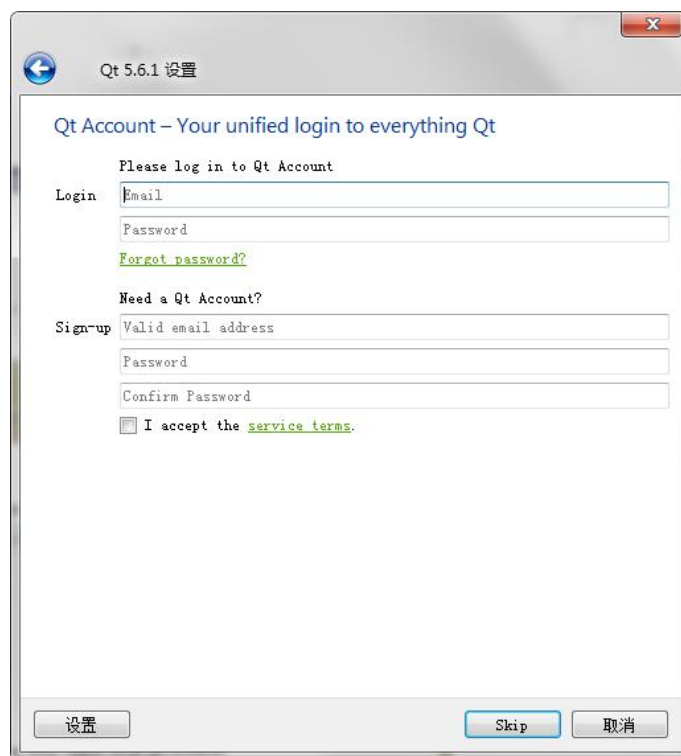


图 1.2 登录 Qt 帐号

(3) 选择安装位置界面如图 1.3 所示。这里默认安装在 C 盘，选中“Associate common file types with Qt Creator”选项会为 Qt Creator 关联相应类型的文件，以后双击相应类型的文件（如 Qt 项目文件）可直接在 Qt Creator 中打开。这里保持默认设置，单击“下一步”按钮。



图 1.3 选择安装位置

(4) 在选择组件界面可以选择需要安装的组件，如图 1.4 所示，这里一般保持默认即可。



图 1.4 选择组件

(5) 在如图 1.5 所示的“许可协议”界面，选择接受许可即可。然后单击“下一步”按钮。



图 1.5 许可协议界面

(6) 在设置“开始菜单快捷方式”界面，可以在系统开始菜单中创建 Qt SDK 的快捷方式，可以选择显示的位置和更改显示名称。如图 1.6 所示。这里保持默认即可。单击“下一步”按钮。

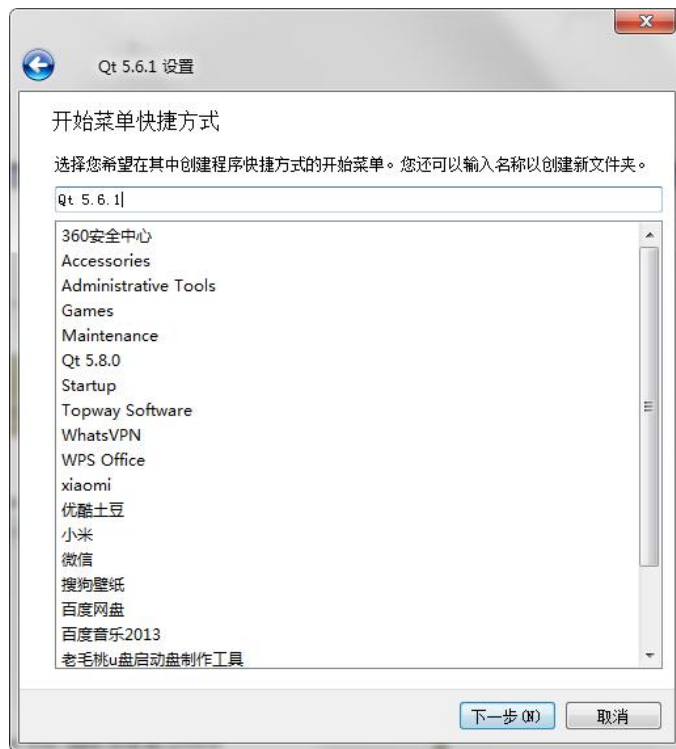


图 1.6 设置开始菜单快捷方式

(5) 在“已做好安装准备”界面，单击“安装”按钮开始安装。如图 1.7 所示。



图 1.7 已做好安装准备界面

(6) 完成安装。在完成安装界面，单击“完成”按钮结束安装。因为默认勾选了“Launch Qt Creator”，所以完成安装后会自动运行 Qt Creator。如图 1.8 所示。

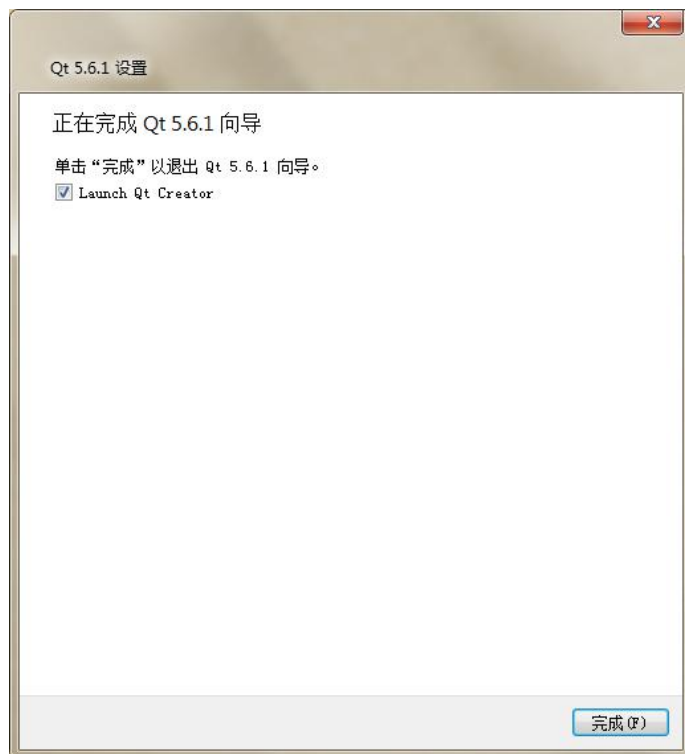


图 1.8 完成安装

(7) Qt Creator 的主界面如图 1.9 所示。可以点击各个菜单和功能图标，对 Qt Creator 界面进行初步了解。

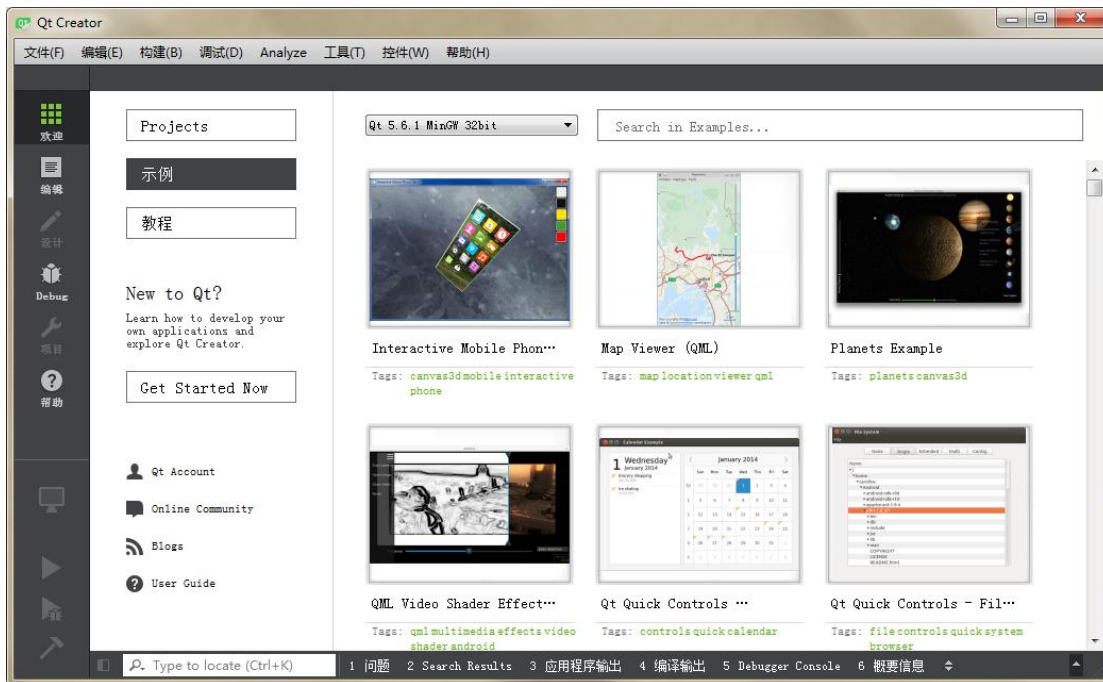


图 1.9 Qt Creator 主界面

2.运行一个示例程序

(1) 在欢迎界面单击“示例”页面就可以看到所有示例程序了，它们几乎涉及到了 Qt 支持的所有功能。这里还提供了一个搜索栏，可以进行示例程序的查找，比如查找所有和对话框相关的例子，可以输入“dialog”关键字，如图 1.10 所示。

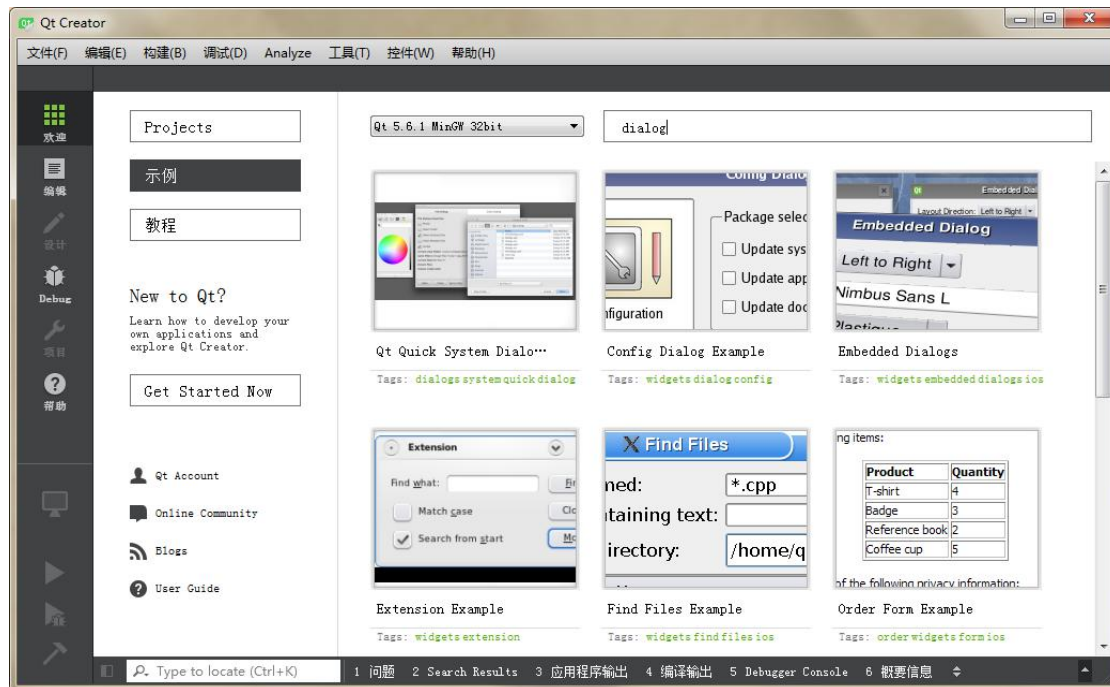


图 1.10 示例程序

(2) 下面选择 Embedded Dialogs 示例程序，这时会自动在新窗口打开该示例的帮助文档，可以对该示例进行了解。如图 1.11 所示。

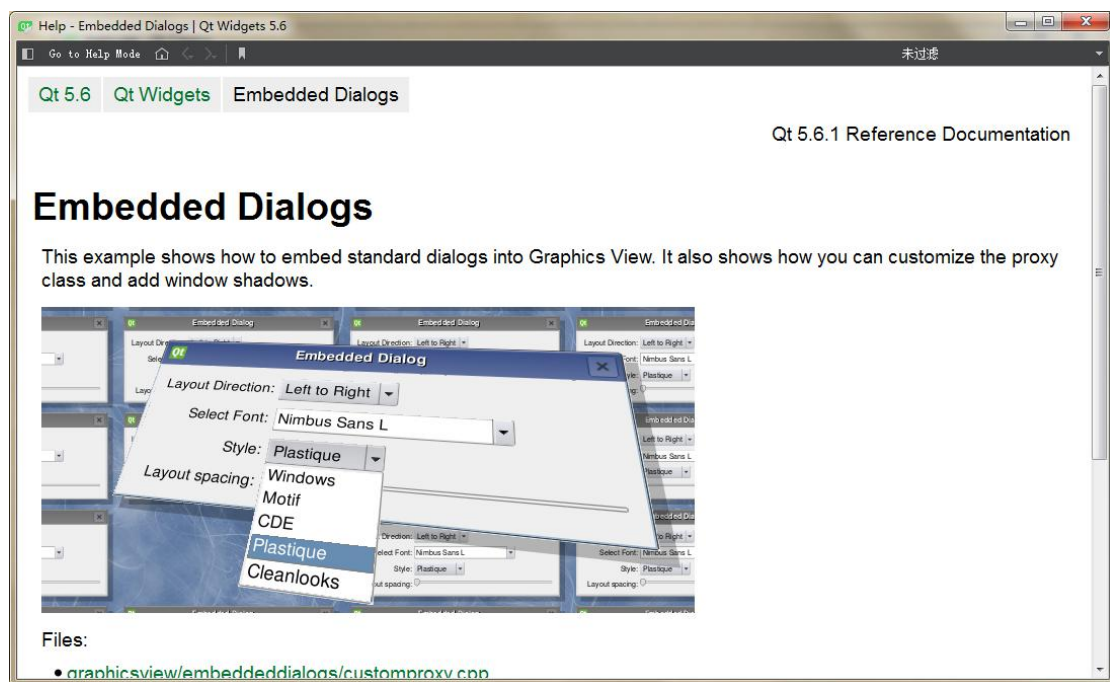


图 1.11 示例程序的帮助文档

(3) 首次打开程序会让配置构建套件，因为这里只有一个桌面版的 Qt，所以保持默认，选择 Configure Project 即可。如图 1.12 所示。

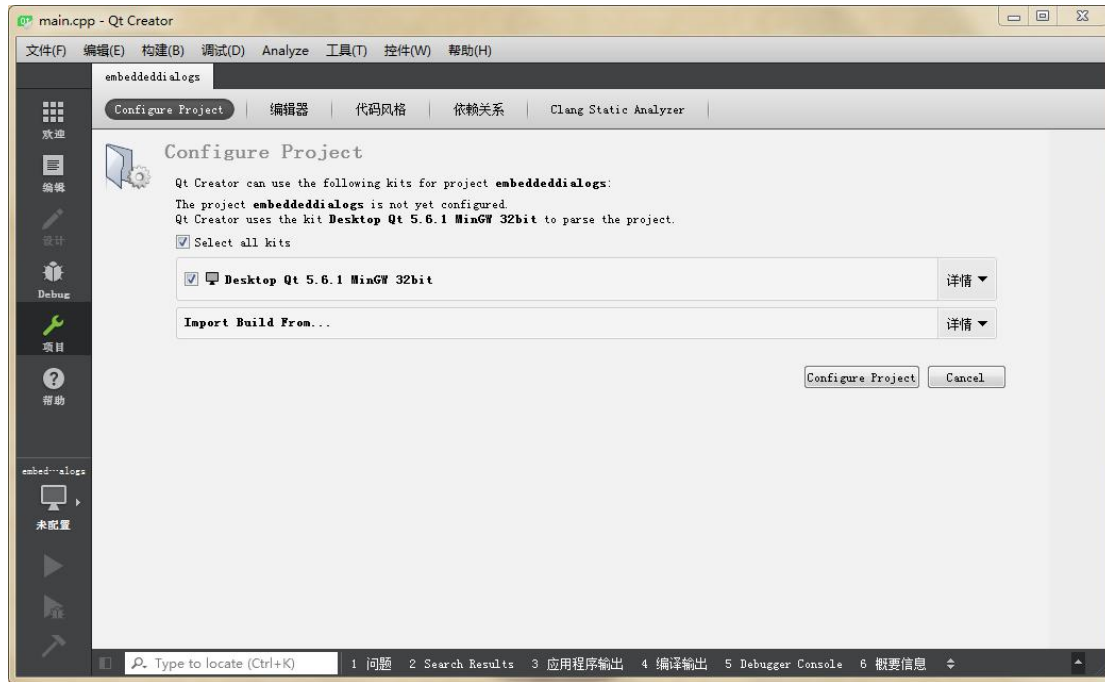


图 1.12 选择构建套件

(4) 这时便进入了编辑模式。每当打开一个示例程序，Qt Creator 便会自动打开该程序的项目文件，然后进入编辑模式。可以在项目文件列表中查看该示例的源代码。如图 1.13 所示。

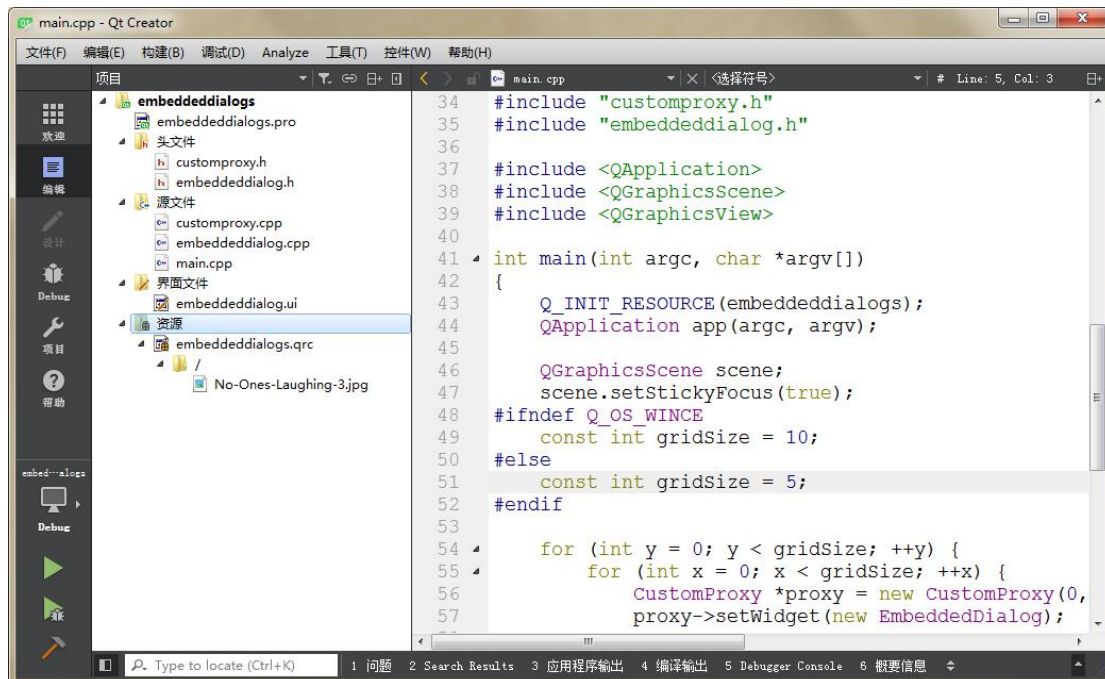



图 1.13 示例程序的编辑模式

(4) 现在单击左下角的  运行按钮或者 Ctrl+R 快捷键，程序便开始编译运行，在下面的“应用程序输出”栏会显示程序的运行信息和调试输出信息。程序运行结果如图 1.14 所示。

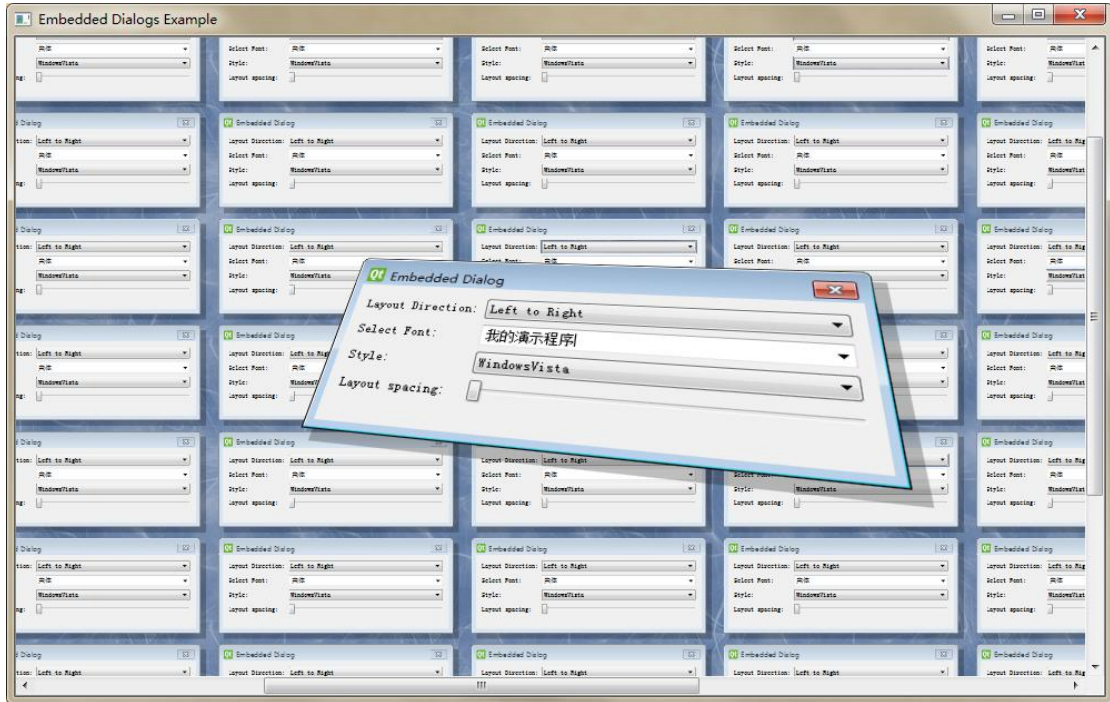


图 1.14 程序运行效果

3 查看构建套件

打开 Qt Creator，选择“工具→选项”菜单项，然后选择左侧的“构建和运行”项。在“构建套件(Kit)”中可以看到已经自动检测到了名称为 Desktop Qt 5.6.1 MinGW 32bit 的构建套件，这里还可以查看编译器、调试器、Qt 版本等信息。因为这里只有一个默认的 Qt 版本和编译器，所以现在无需设置，如果同时安装了多个版本的 Qt，只想使用一个 Qt Creator 进行开发，那么可以在这里进行添加。如图 1.15 所示。

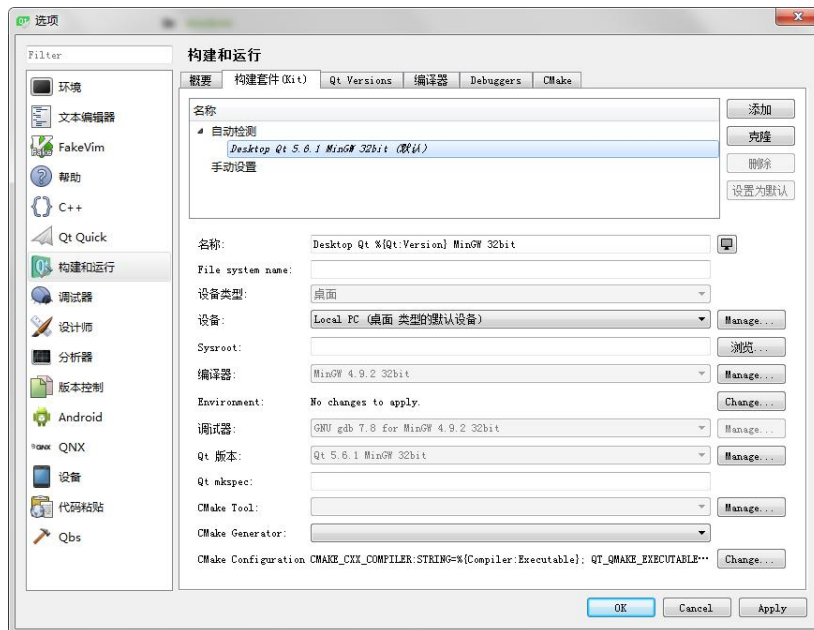


图 1.15 查看构建套件

4. 查看 Qt 工具

前面安装的 Qt 5.6.1 中包含了几个很有用的工具，分别是 Qt Assistant（Qt 助手）、Qt Designer（Qt 设计师）和 Qt Linguist（Qt 语言家）。可以从开始菜单启动它们。如图 1.16 所示。现在先来运行这些工具，对其有一个大概了解。



图 1.16 Qt 工具菜单目录

(1) 运行 Qt Assistant（Qt 助手），如图 1.17 所示。

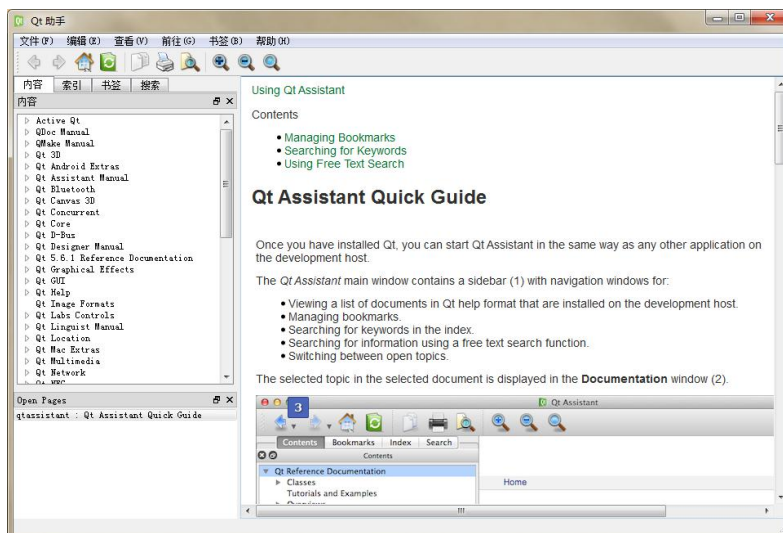


图 1.17 Qt Assistant

(2) 运行 Qt Designer（Qt 设计师），如图 1.18 所示。

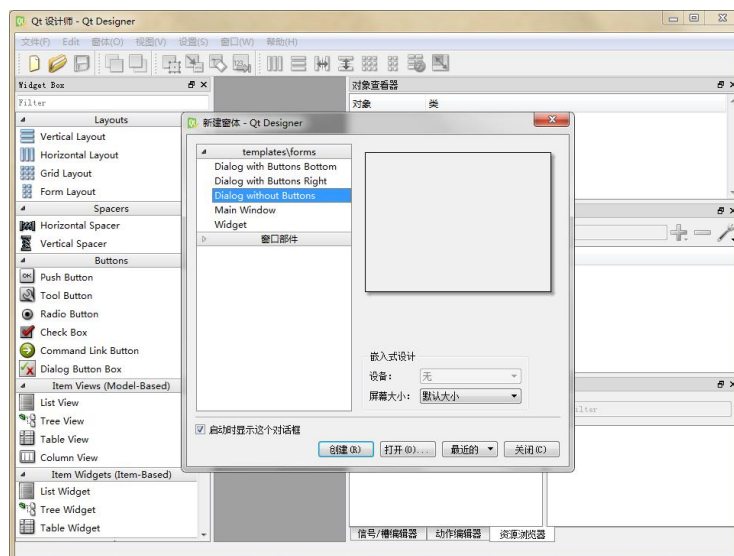


图 1.18 Qt Designer

(4) 运行 Qt Linguist (Qt 语言家), 如图 1.19 所示。

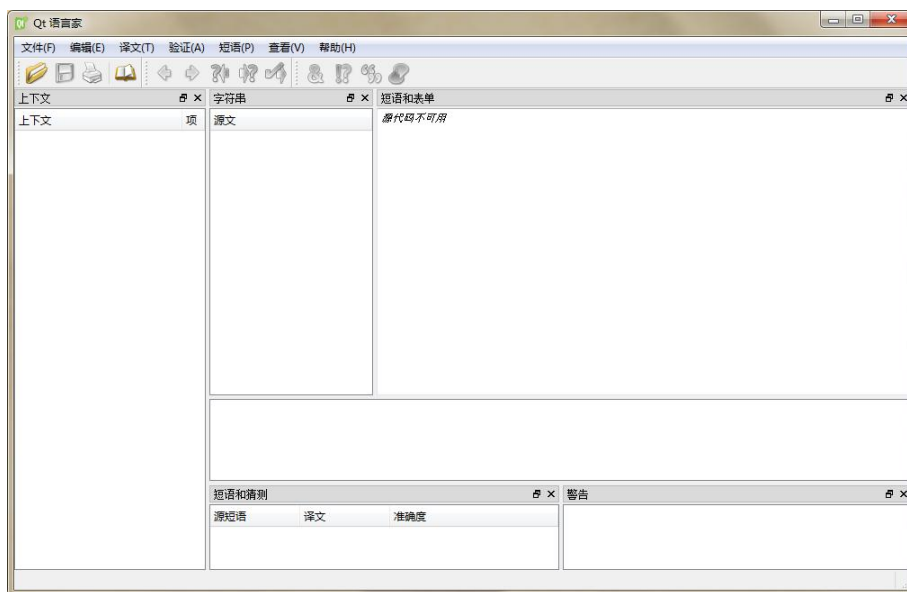


图 1.19 Qt Linguist

思考题： Qt Creator、Qt 和 MinGW 各有什么作用？

实验 2 编译和发布 Qt 程序

目的与要求

- (1) 掌握创建 Qt 程序的方法
- (2) 掌握发布 Qt 程序的方法
- (3) 学会为 Qt 程序添加应用程序图标
- (4) 了解 Qt 发布需要的 DLL 动态库文件

实验准备

- (1) 搭建好 Qt 开发环境
- (2) 了解 Qt Creator 设计模式的基本使用方法
- (3) 了解 DLL 动态库文件的作用
- (4) 了解 Debug 版本和 Release 版本的区别

实验内容

1. 创建 hello world 程序

(1) 运行 Qt Creator，打开“文件→新建文件或项目”菜单项（也可以直接按下 Ctrl+N 快捷键），在选择模板页面选择 Application 中的“Qt Widgets Application”一项，然后单击“Choose”按钮，如图 2.1 所示。

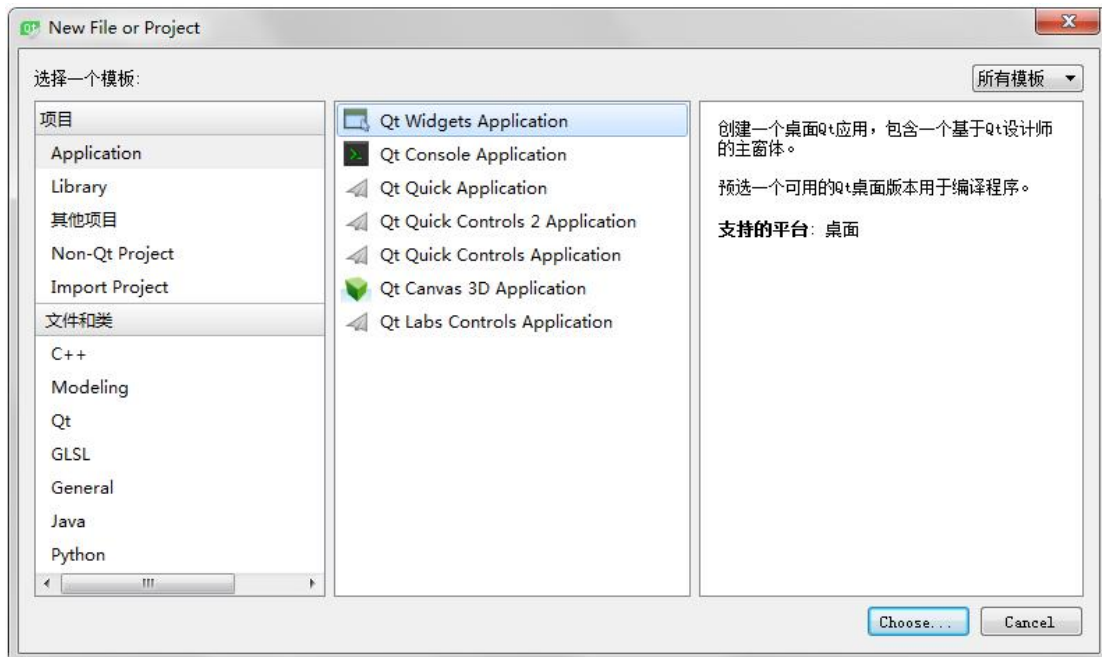


图 2.1 选择模板

(2) 输入项目信息。在“项目介绍和位置”页面输入项目的名称为 helloworld，然后

单击创建路径右边的“浏览”按钮选择源码路径，例如这里是“E:\app\src\02\2-1”。如果选中了这里的“设置默认的项目路径”，那么以后创建的项目会默认使用该目录，如图 2.2 所示。单击“下一步”进入下个页面。（注意：项目名和路径中都不能出现中文。）



图 2.2 项目介绍与位置

(3) 选择构建套件。这里显示的 Desktop Qt 5.6.1 MinGW 32bit 就是在实验 1 看到的构建套件，下面默认为 Debug 版本和 Release 版本分别设置了两个不同的目录，如图 2.3 所示。然后单击“下一步”。

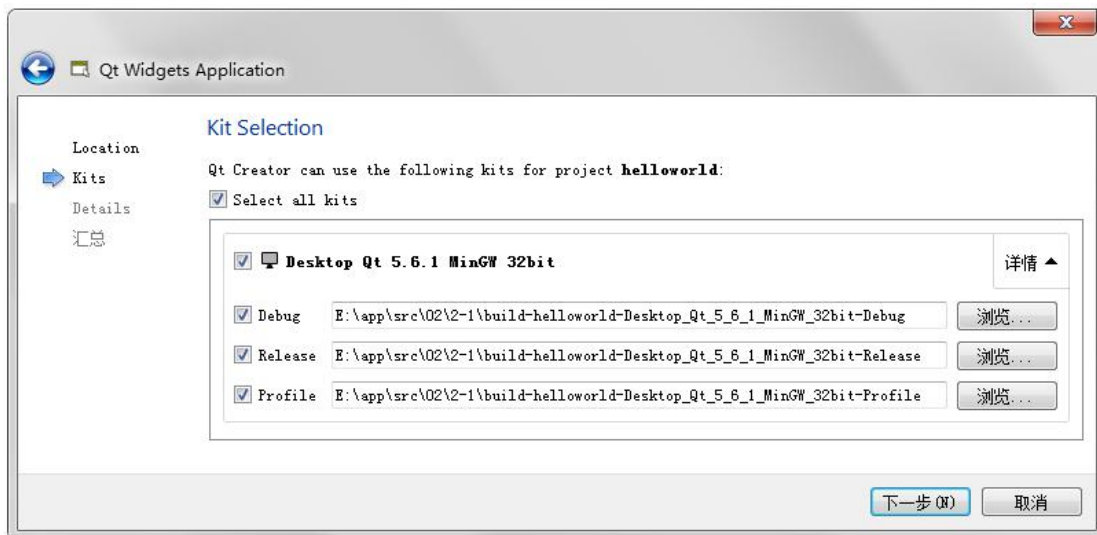


图 2.3 选择构建套件

(4) 输入类信息。在“类信息”页面中创建一个自定义类。这里设定类名为 HelloDialog，基类选择 QDialog，表明该类继承自 QDialog 类，使用这个类可以生成一个对话框界面。这时下面的头文件、源文件和界面文件都会自动生成，保持默认即可，如图 2.4 所示。然后单击“下一步”。

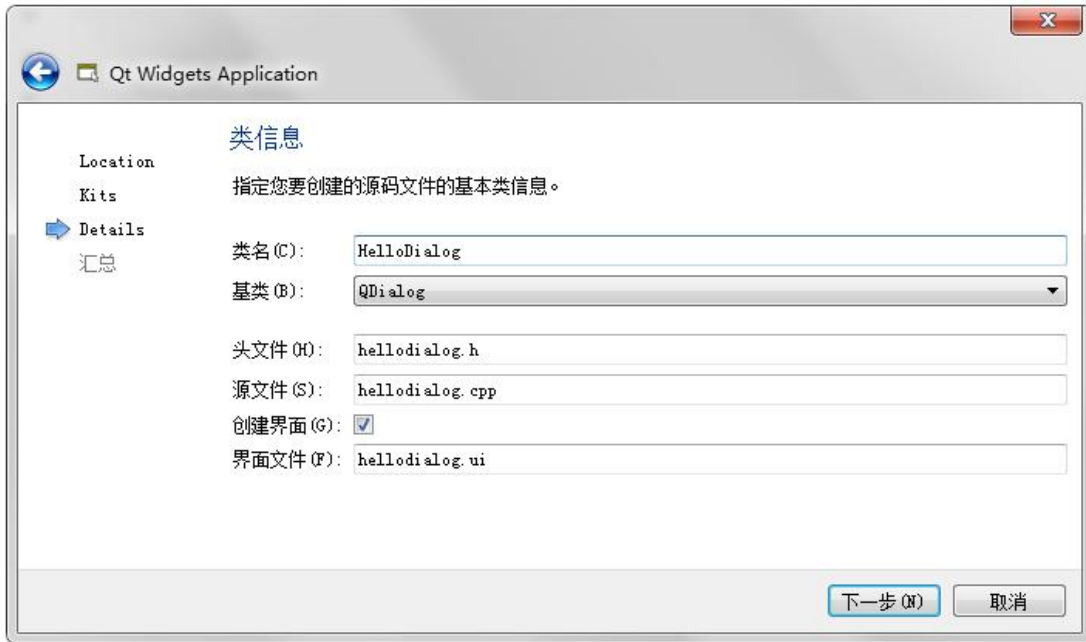


图 2.4 类信息

(5)设置项目管理。在这里可以看到这个项目的汇总信息，还可以使用版本控制系统，这个项目不会涉及，所以可以直接单击“完成”按钮完成项目的创建。如图 2.5 所示。



图 2.5 项目管理

(6) 项目建立完成后会直接进入编辑模式。界面的右边是编辑器，可以阅读和编辑代码。如果觉得字体太小，可以使用快捷键 Ctrl + “+”（即同时按下 Ctrl 键和+号键）来放大字体，使用 Ctrl + “-”（减号）来缩小字体，也可以使用 Ctrl 键+鼠标滚轮来缩放字体。使用 Ctrl+0（数字）可以使字体还原到默认大小。在左边侧边栏，罗列了项目中的所有文件，如图 2.6 所示。

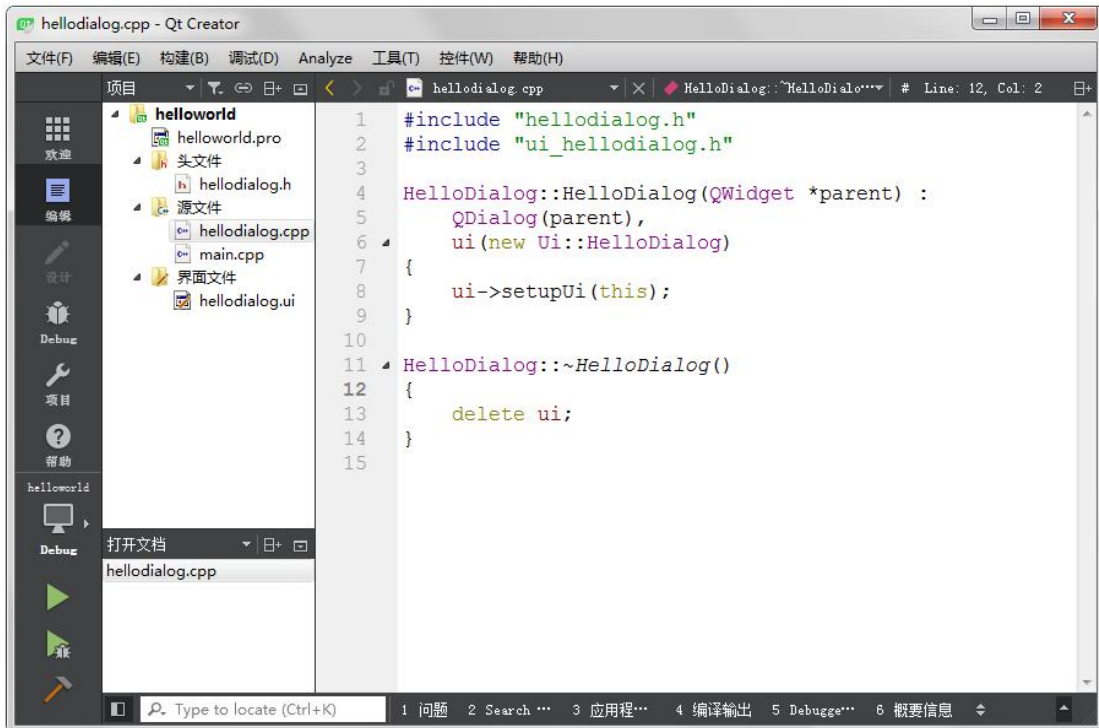


图 2.6 编辑模式

(7) 在 Qt Creator 的编辑模式下双击项目文件列表中界面文件分类下的 `helloworld.ui` 文件，这时便进入了设计模式，如图 2.7 所示。

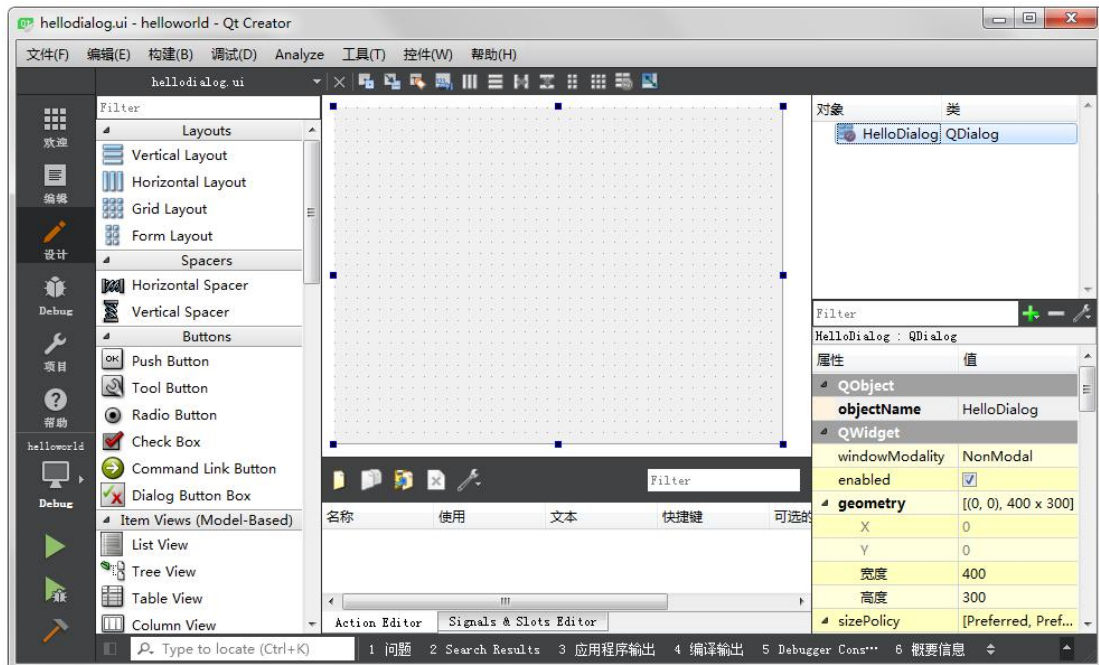


图 2.7 设计模式

(8) 从部件列表中找到 `Label` (标签) 部件，然后按着鼠标左键将它拖到主设计区的界面上，再双击它进入编辑状态后输入“Hello World! 你好 Qt!”字符串。如图 2.8 所示。

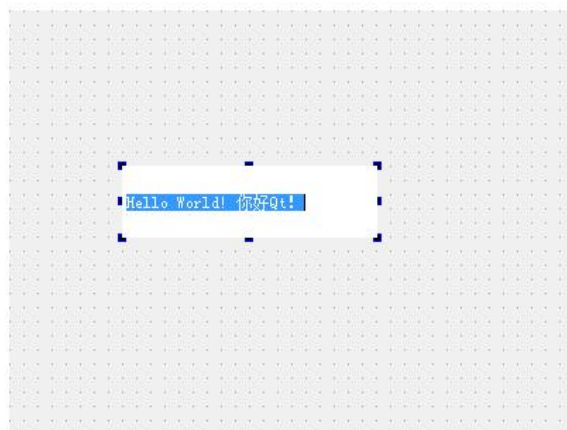


图 2.8 添加标签部件

2.运行并发布程序

(1) 可以使用快捷键 **Ctrl+R** 或者通过按下左下角的运行按钮来运行程序。这时可能会弹出“保存修改”对话框，这是因为刚才在设计模式更改了界面，而 `helloworld.ui` 文件被修改了但是还没有保存。现在要编译运行该程序，就要先保存所有文件。可以选中“构建之前总是先保存文件”选项，以后再运行程序时就可以自动保存文件了。

(2) 程序运行效果如图 2.9 所示。



图 2.9 Helloworld 程序运行效果

(3) 要发布程序时，要使用 `release` 版本。在 Qt Creator 中对 `helloworld` 程序进行 `release` 版本的编译，需要在左下角的目标选择器（Target selector）中将构建目标设置为 `Release`，如图 2.10 所示，然后单击运行图标编译运行程序。



图 2.10 目标选择器

(4) 编译完成之后再查看项目目录中：

E:\app\src\02\2-1\build-helloworld-Desktop_Qt_5_6_1_MinGW_32bit-Release 文件夹的 release 目录中，已经生成了 helloworld.exe 文件。

(5) 在桌面上新建一个文件夹，重命名为“我的第一个 Qt 程序”，然后将 release 文件夹中的 helloworld.exe 复制过来，再去 Qt 安装目录的 bin 目录中将 libgcc_s_dw2-1.dll、libstdc++-6.dll、libwinpthread-1.dll、Qt5Core.dll、Qt5Gui.dll 和 Qt5Widgets.dll 这 6 个文件复制过来。另外，还需要将 C:\Qt\Qt5.6.1\5.6\mingw49_32\plugins 目录中的 platforms 文件夹复制过来（不要修改该文件夹名称），里面只需要保留 qwindows.dll 文件即可。

3. 设置应用程序图标

在程序发布时，一般会给可执行文件设置一个漂亮的图标。下面是在 Windows 系统上设置应用程序图标的方法。

(1) 创建 .ico 文件。将 .ico 图标文件复制到工程文件夹的 helloworld 目录中，重命名为“myico.ico”。完成后 helloworld 文件夹中的内容如图 2.11 所示。

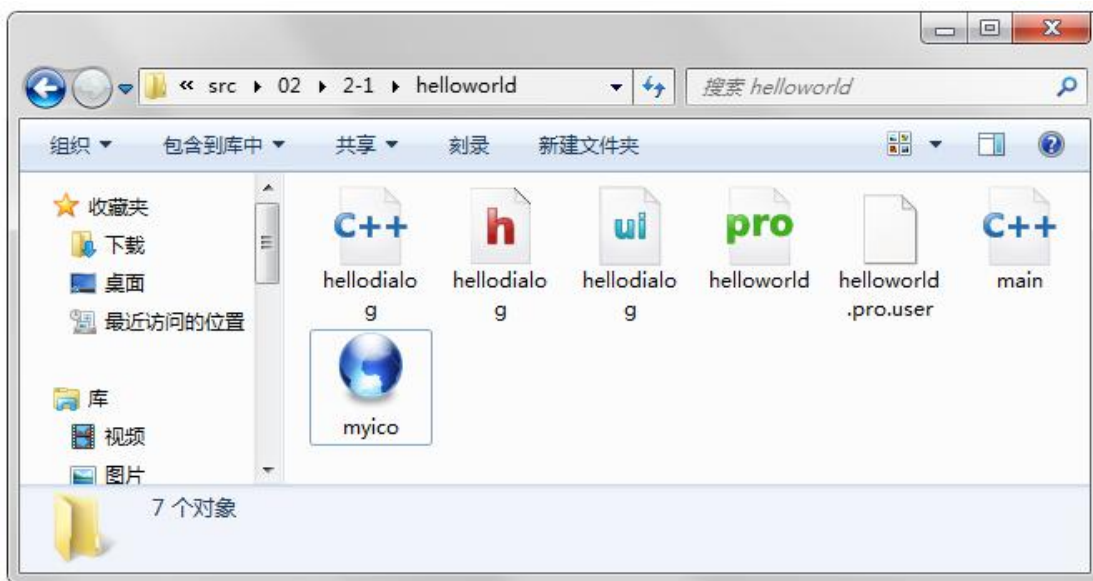
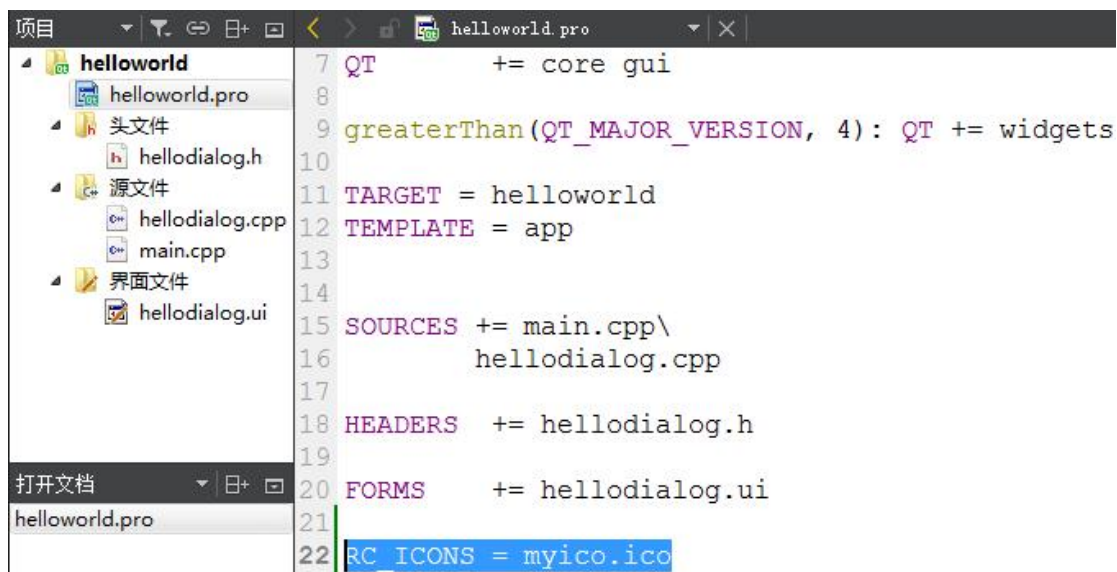


图 2.11 helloworld 目录

(2) 修改项目文件。在 Qt Creator 中的编辑模式双击 helloworld.pro 文件，在最后面添加下面一行代码，如图 2.12 所示。

```
RC_ICONS = myico.ico
```



```
7 QT += core gui
8
9 greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
10
11 TARGET = helloworld
12 TEMPLATE = app
13
14
15 SOURCES += main.cpp\
16             hellodialog.cpp
17
18 HEADERS += hellodialog.h
19
20 FORMS += hellodialog.ui
21
22 RC_ICONS = myico.ico
```

图 2.12 编辑工程文件

(3) 运行程序。如图 2.13 所示，可以看到窗口左上角的图标已经更换了。



图 2.13 更换了图标的程序运行界面

(4) 查看一下 release 文件夹中的文件，可以看到现在 exe 文件已经更换了新的图标，如图 2.14 所示。

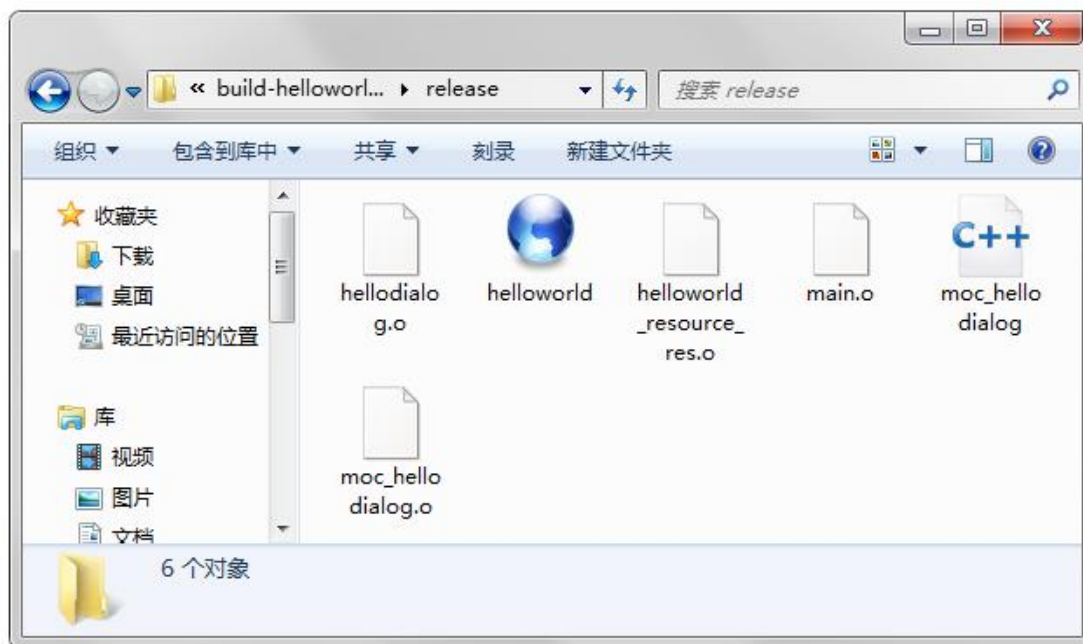


图 2.14 更换了图标后的 release 目录

(6) 现在可以将更改了图标的 `helloworld.exe` 文件复制到程序发布目录，然后就可以将程序发布目录压缩打包进行发布了。发布的程序可以在没有安装 Qt 的 Windows 系统上运行。

思考题： Qt 程序从编写、编译、运行，到最后的发布，整个流程是怎样的？

实验 3 使用 Qt 资源文件

目的与要求

- (1) 掌握设置菜单栏方法
- (2) 掌握使用 Qt 资源文件的方法
- (3) 会使用代码添加菜单
- (4) 了解资源文件的构成

实验准备

- (1) 搭建好 Qt 开发环境
- (2) 了解使用 Qt Creator 创建 Qt 应用程序的流程
- (3) 了解应用程序主窗口的组成
- (4) 准备好需要使用的图标文件

实验内容

1. 创建主窗口菜单

(1) 新建 Qt Widgets 应用，项目名称为 myMainWindow，基类选择 QMainWindow，类名为 MainWindow。

(2) 创建完项目后，双击 mainwindow.ui 文件进入设计模式。可以看到界面左上角的“在这里输入”，可以在这里添加菜单。双击“在这里输入”，将其更改为“文件(&F)”，然后按下回车键，效果如图 3.1 所示。这里的&F 表明将菜单的快捷键设置为了 Alt + F，可以看到，实际的显示效果中&符号是隐藏的。

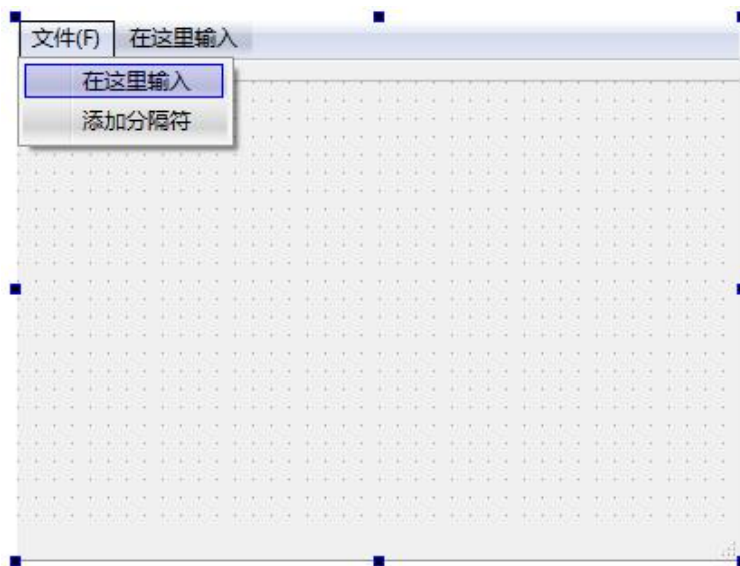


图 3.1 创建菜单

(3) 同样的方法，在文件菜单中添加“新建(&N)”菜单项（如果无法输入中文，可以从别处复制粘贴），效果如图 3.2 所示。菜单后面的那个加号图标是用来创建下一级菜单的。



图 3.2 创建菜单项

2. 添加菜单图标

(1) Qt 中的一个菜单项被看做是一个 Action，在设计器下面的 Action 编辑器中可以看到刚才添加的“新建”动作，如图 3.3 所示。



图 3.3 动作编辑器

(2) 双击 action_N 条目，会弹出编辑动作对话框，这里可以进行各项设置，比如可以设置动作的快捷键，点击一下快捷键后面的行编辑器，然后按下键盘上的 Ctrl + N，这样就可以将该菜单的快捷键设置为 Ctrl + N。如图 3.4 所示。

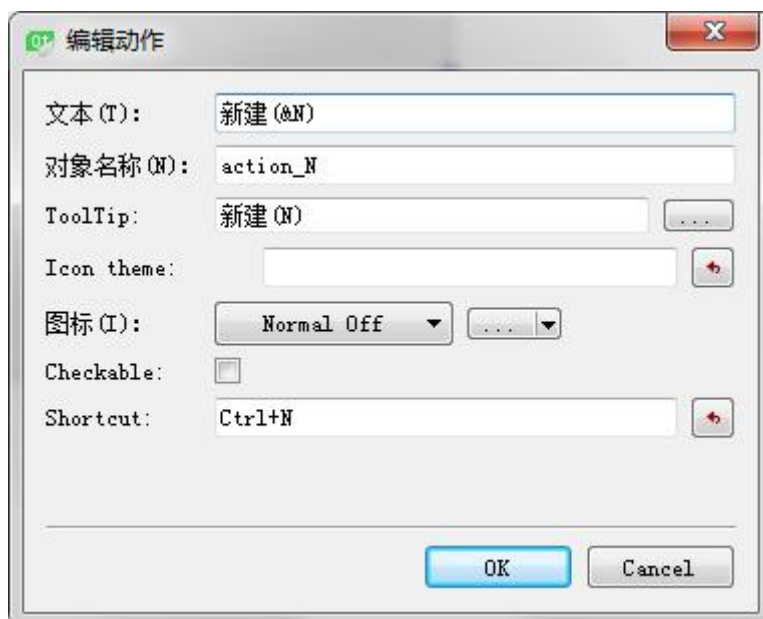



图 3.4 编辑动作

(3) 在编辑动作对话框中的图标后面的  黑色箭头下拉框可以选择使用资源还是使用文件来设置图标，如果使用文件的话，那么就可以直接在弹出的文件对话框中选择本地磁盘上的一个图标文件。如果直接单击这个按钮默认是使用资源。现在先按下编辑动作对话框的“OK”按钮关闭它。

3. 添加资源文件

Qt 中可以使用资源文件将各种类型的文件添加到最终生成的可执行文件中，这样就可以避免使用外部文件可能出现的一些问题。而且，在编译时 Qt 还会将资源文件进行压缩，可以使可执行文件的体积尽可能缩小。

(1) 向项目中添加新文件，模板选择 Qt 分类中的 Qt Resource File，如图 3.5 所示。然后将名称设置为“myResources”。

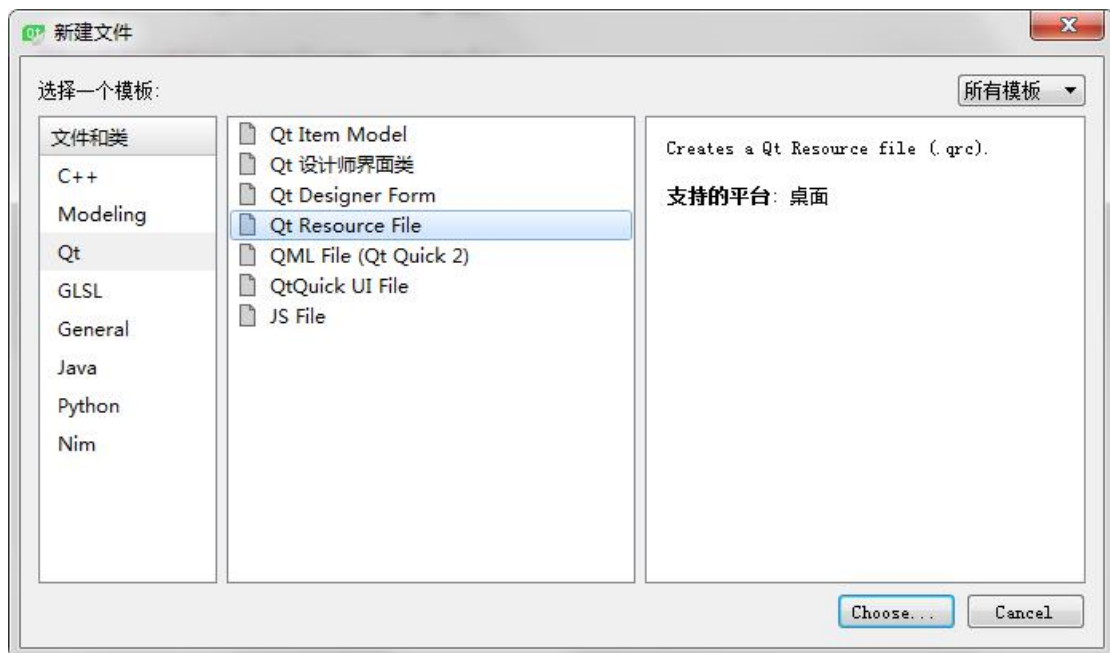


图 3.5 创建资源文件

(2) 创建完文件后会打开该资源文件，这里需要先在下面添加前缀，点击添加按钮，然后选择前缀，默认的前缀是“/new/prefix1”，这个可以随意修改（不要出现中文字符）。这里因为要添加图片，所以修改为/myImages。然后再按下添加按钮来添加文件，这里最好将所有要用到的图片放到项目目录中。比如这里在项目目录中新建了一个 images 文件夹，然后将需要的图标文件复制进去。添加完文件后，如图 3.6 所示。

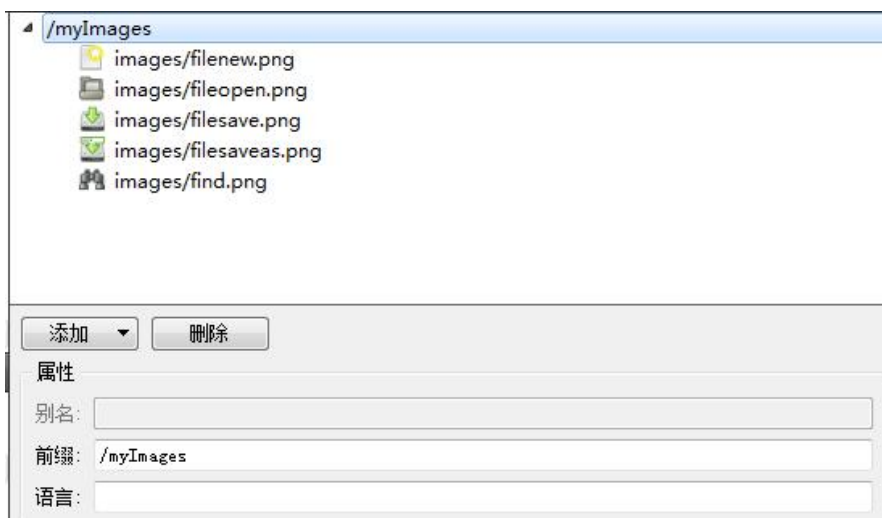



图 3.6 添加图片

(3) 当添加完资源后，一定要按下 Ctrl + S 来保存资源文件，不然在后面可能无法显示已经添加的资源。

4.使用资源文件

(1) 重新到设计模式打开新建菜单的编辑动作对话框，然后添加图标。在打开的选择资源对话框中，第一次可能无法显示已经存在的资源，可以按下左上角的绿箭头  来更新显示。效果如图 3.7 所示。

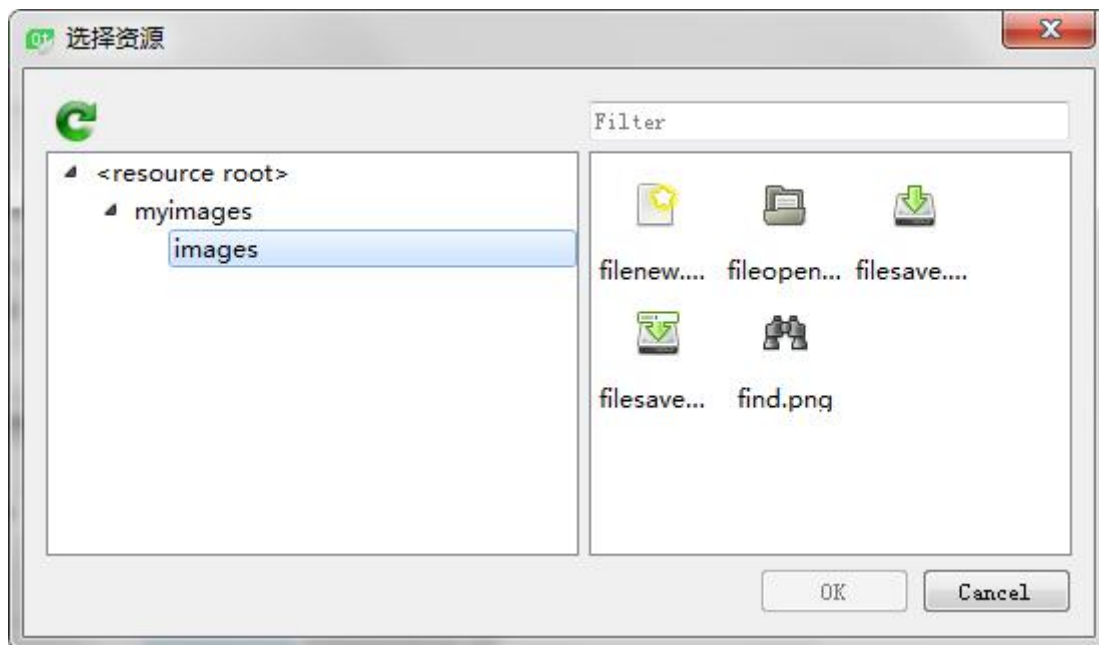


图 3.7 添加资源图标

(2) 点击这里需要的新建图标 filenew.png，按下“OK”按钮即可。现在按下 Ctrl + R 键运

行程序，效果如图 3.8 所示。

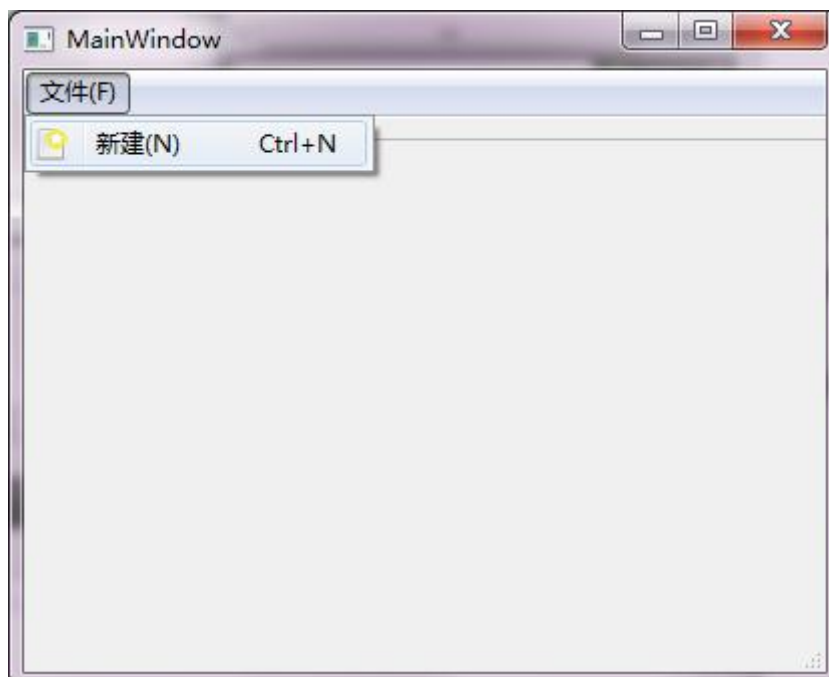


图 3.8 添加图标运行效果

5.使用代码来添加菜单和图标

(1) 对于添加的资源文件，在项目目录中可以看到，即 `myResources.qrc`，使用写字板程序将其打开，可以发现它其实就是一个 XML 文档：

```
<RCC>
  <qresource prefix="/myImages">
    <file>images/filenew.png</file>
    <file>images/fileopen.png</file>
    <file>images/filesave.png</file>
    <file>images/filesaveas.png</file>
    <file>images/find.png</file>
  </qresource>
</RCC>
```

(2) 前面是在设计模式添加了图标文件，下面使用代码再来添加一个菜单，并为其设置图标。在编辑模式打开 `mainwindow.cpp` 文件，并在构造函数中添加如下代码：

```
// 创建新的动作
 QAction *openAction = new QAction(tr("&Open"), this);
// 添加图标
 QIcon icon(":/myImages/images/fileopen.png");
 openAction->setIcon(icon);
// 设置快捷键
```

```
openAction->setShortcut(QKeySequence(tr("Ctrl+O")));
```

```
// 在文件菜单中设置新的打开动作
```

```
ui->menu_F->addAction(openAction);
```

这里添加图标时，就是使用的资源文件中的图标。使用资源文件，需要在最开始使用冒号，然后添加前缀，后面是文件的路径。在代码中使用文件菜单，就是使用其 `objectName` 属性。

(3) 现在运行程序查看效果。

思考题： 使用资源文件和直接使用本地的图片文件，这两种方式有什么优缺点？

实验 4 创建登陆对话框

目的与要求

- (1) 掌握自定义部件类的方法
- (2) 学会使用信号和槽机制
- (3) 了解如何创建登录对话框
- (4) 了解密码显示的方式

实验准备

- (1) 搭建好 Qt 开发环境
- (2) 会使用 Qt 设计模式设计界面
- (3) 了解信号和槽的基本概念

实验内容

编写程序，实现在程序主界面出现以前，弹出登录对话框，并可以填写用户名和密码，当按下登录按钮，如果用户名和密码均正确则进入主窗口，如果有错则弹出警告对话框。

1.创建程序

(1) 新建 Qt Widgets 应用，项目名称为“login”，类名和基类保持 QMainWindow 和 QMainWindow 不变。

(2) 完成项目创建后，向项目中添加新的 Qt 设计师界面类，模板选择 Dialog without Buttons，类名更改为“LoginDialog”。完成后向界面上添加两个标签 Label、两个行编辑器 Line Edit 和两个按钮 Push Button，设计界面如图 4.1 所示。



图 4.1 设计界面效果

(3) 在属性编辑器中将用户名后面的行编辑器的 object Name 属性更改为“usrLineEdit”，密码后面的行编辑器的 object Name 属性更改为“pwdLineEdit”，登录按钮的 object Name 属性更改为“loginBtn”，退出按钮的 object Name 属性更改为“exitBtn”。如图 4.2 所示。

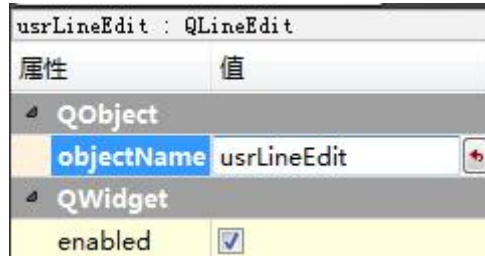


图 4.2 修改对象名称属性

(4) 在设计模式下面的信号和槽编辑器（Signals & Slots Editor）中，先点击左上角的绿色加号添加关联，然后选择发送者为 exitBtn，信号为 clicked()，接收者为 LoginDialog，槽为 close()。如图 4.3 所示。这样，当单击退出按钮时，就会关闭登录对话框。



图 4.3 添加信号和槽关联

(5) 右击登录按钮，在弹出的菜单中选择“转到槽...”，然后选择 clicked()信号并确定。转到相应的槽以后，添加函数调用：

```
void LoginDialog::on_loginBtn_clicked()
{
    accept();
}
```

(6) 下面到 main.cpp 文件，更改内容如下：

```
#include <QApplication>
#include "mainwindow.h"
#include "logindialog.h"
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
```

```
MainWindow w;  
LoginDialog dlg;  
if (dlg.exec() == QDialog::Accepted)  
{  
    w.show();  
    return a.exec();  
}  
else return 0;  
}
```

(7) 运行程序，按下退出按钮会退出程序，按下登录按钮会关闭登录对话框，并显示主窗口。

2. 登录设置

(1) 下面添加代码来实现使用用户名和密码登录，这里只是简单的将用户名和密码设置为了固定的字符串。到 `logindialog.cpp` 文件中将登录按钮的单击信号对应的槽更改如下：

```
void LoginDialog::on_loginBtn_clicked()  
{  
    // 判断用户名和密码是否正确，  
    // 如果错误则弹出警告对话框  
    if(ui->usrLineEdit->text() == tr("yafeilinux") &&  
        ui->pwdLineEdit->text() == tr("123456"))  
    {  
        accept();  
    } else {  
        QMessageBox::warning(this, tr("Waring"),  
                               tr("user name or password error!"),  
                               QMessageBox::Yes);  
    }  
}
```

Qt 中的 `QMessageBox` 类提供了多种常用的对话框类型，比如这里的警告对话框，还有提示对话框、问题对话框等。这里使用静态函数来设置一个警告对话框，这种方式很方便。其中的参数依次是：`this` 表明父窗口是登录对话框，然后是窗口标题，然后是显示的内容，最后一个参数是显示的按钮，这里使用了一个 `Yes` 按钮。

在 `logindialog.cpp` 添加 `QMessageBox` 类的头文件包含，即：

```
#include <QMessageBox>
```

(2) 运行程序，如果输入用户名为“yafeilinux”，密码为“123456”，那么可以登录，如果输入其他的字符，则会弹出警告对话框，如图 4.4 所示。



图 4.4 运行效果

(3) 对于输入的密码，常见的是显示成小黑点的样式。下面双击 `logindialog.ui` 文件进入设计模式，然后选中界面上的密码行编辑器部件，在其属性编辑器中将 `echoMode` 属性选择为 `Password`。这时再次运行程序，可以看到密码已经改变显示样式了。如图 4.5 所示。



图 4.5 修改密码显示效果

除了在属性编辑器中进行更改，也可以在 `loginDialog` 类的构造函数中使用 `setEchoMode(QLineEdit::Password)` 函数来设置。

(4) 在行编辑器的属性栏中还可以设置占位符，就是没有输入信息前的一些提示语句。例如将密码行编辑器的 `placeholderText` 属性更改为“请输入密码”，将用户名行编辑器的更改为“请输入用户名”，运行效果如图 4.6 所示。



图 4.6 占位符运行效果

(5) 对于行编辑器，还有一个问题，比如输入用户名，不小心在前面添加了一个空格，这时也需要保证输入是正确的，这个可以使用 `QString` 类的 `trimmed()` 函数来实现，它可以去除字符串前后的空白字符。下面将 `logindialog.cpp` 文件中登录按钮单击信号关联的槽函数中的判断代码更改为：

```
if(ui->usrLineEdit->text().trimmed() == tr("yafeilinux")
    && ui->pwdLineEdit->text() == tr("123456"))
```

这时运行程序，就可以实现相应的功能了。

(6) 当登录失败后，希望可以清空用户名和密码信息，并将光标定位到用户名输入框中。这个可以通过在判断用户名和密码错误后添加相应的代码来实现：

```
void LoginDialog::on_loginBtn_clicked()
{
    // 判断用户名和密码是否正确，如果错误则弹出警告对话框
    if(ui->usrLineEdit->text().trimmed() == tr("yafeilinux")
        && ui->pwdLineEdit->text() == tr("123456"))
    {
        accept();
    } else {
        QMessageBox::warning(this, tr("Waring"),
            tr("user name or password error!"),
            QMessageBox::Yes);
        // 清空内容并定位光标
        ui->usrLineEdit->clear();
        ui->pwdLineEdit->clear();
        ui->usrLineEdit->setFocus();
    }
}
```

下面运行程序并测试效果。

思考题： 还有没有其他方式来实现登录窗口？

实验 5 定时器和随机数

目的与要求

- (1) 掌握使用定时器的方法
- (2) 掌握随机数的设置方法
- (3) 会使用定时器实现时钟
- (4) 了解图片显示的方法

实验准备

- (1) 了解信号和槽的基本使用方法
- (2) 了解定时器和随机数的基本概念
- (3) 准备好需要使用的图片文件

实验内容

使用定时器可以在指定时间执行一些功能，而要实现随机效果就要使用随机数来实现。实验目标为新建程序实现电子时钟效果，并每隔一秒随机显示一张图片。

- (1) 新建 Qt Widgets 应用，项目名称为 myTimer，基类选择 QWidget，类名为 Widget。
- (2) 完成项目创建后，双击 widget.ui 进入设计模式，然后向界面上拖入两个 Push Button、一个 Line Edit 和一个 Label 部件，修改其显示文本，效果如图 5.1 所示。



图 5.1 设计界面效果

- (3) 然后到 widget.h 文件中添加类的前置声明：

```
class QTimer;
```

再添加一个私有槽声明：

```
private slots:  
    void timerUpdate();
```

和一个私有对象定义:

```
private:  
    QTimer *timer;
```

(4) 到 widget.cpp 中, 先添加头文件包含:

```
#include <QTimer>  
#include <QDateTime>
```

然后在构造函数中添加如下代码:

```
timer = new QTimer(this);  
//关联定时器溢出信号和相应的槽函数  
connect(timer, &QTimer::timeout, this, &Widget::timerUpdate);  
  
qrand(QTime(0, 0, 0).secsTo(QTime::currentTime()));
```

这里创建了一个定时器, 并将其溢出信号和更新槽关联起来, 最后使用 start() 函数来开启定时器。

关于随机数, 在 Qt 中是使用 qrand() 和 qsrnd() 两个函数实现的。qrand() 函数可以产生随机数, 例如 qrand()%10 可以产生 0-9 之间的随机数, 要产生 100 以内的随机数就是%100, 以此类推。在使用 qrand() 函数产生随机数之前, 一般要使用 qsrnd() 函数为其设置初值, 如果不设置初值, 那么每次运行程序, qrand() 都会产生相同的一组随机数。为了每次运行程序时都可以产生不同的随机数, 要使用 qsrnd() 设置一个不同的初值。这里使用了 QTime 类的 secsTo() 函数, 它表示两个时间点之间所包含的秒数, 比如代码中就是指从零点整到当前时间所经过的秒数。

(5) 下面在 widget.cpp 文件中添加 timerUpdate() 函数的定义:

```
void Widget::timerUpdate()  
{  
    //获取系统现在的时间  
    QDateTime time = QDateTime::currentDateTime();  
    //设置系统时间显示格式  
    QString str = time.toString("yyyy-MM-dd hh:mm:ss dddd");  
    //在标签上显示时间  
    ui->lineEdit->setText(str);  
  
    int rand = qrand() % 5;    // 产生 5 以内随机整数即 0-4  
  
    ui->label->setPixmap(QString("../myTimer/images/%1.png").arg(rand));  
}
```

这里在行编辑器中显示了当前的时间。然后使用 qrand()%5 产生 5 以内的随机数, 并使用这个随机数来获取图片名称。

(6) 前面代码中使用了 0.png、1.png、2.png、3.png、4.png 这样 5 张图片, 需要复制 5 张图片到项目目录下新建的 images 目录中。

(7) 在设计模式, 分别右击“开始”按钮和“停止”按钮, 选择转到槽, 然后选择 clicked() 信号。将对应的槽函数修改如下:

```
// 开始按钮
void Widget::on_pushButton_clicked()
{
    timer->start(1000);
}

// 停止按钮
void Widget::on_pushButton_2_clicked()
{
    timer->stop();
}
```

使用 `start()` 来启动定时器，其中参数用来指定溢出时间间隔，单位为毫秒。这里设置为 1000，表明每隔 1 秒发射一次 `timeout()` 信号。如果要停止定时器，可以调用 `stop()` 函数。

(8) 运行程序，点击“开始”按钮，会显示当前系统时间，并每隔一秒随机显示一张图片。效果如图 5.2 所示。



图 5.2 程序运行效果

思考题：怎样利用随机数使程序出现随机效果？

实验 6 安装 MySQL 数据库

目的与要求

- (1) 掌握 MySQL 数据库的安装方法
- (2) 掌握 MySQL 数据库的简单操作命令
- (3) 掌握在 Qt 程序中关联 MySQL 数据库的方法
- (4) 了解 Qt 默认支持的数据库类型

实验准备

- (1) 对 MySQL 数据库初步了解
- (2) 了解在 Qt 中使用 MySQL 数据库的基本过程
- (3) 了解 Qt 操作数据库的基本方法

实验内容

1. 安装 MySQL

(1) 从 <http://www.qter.org/portal.php?mod=view&aid=10> 下载 MySQL 安装包，具体文件为 mysql-5.6.10-win32，如图 6.1 所示。

其他相关

资源名称	资源介绍	更新日期	下载地址	备用地址
Git-1.8.3-preview20130601.exe	Windows下的git版本控制软件(官网下载)	2013-3-1	下载	地址2
mysql-5.6.10-win32	MySQL 5.6 纯净版(教程)	2013-5-1	下载	地址2
窗口中用的图标文件	这些文件是从ubuntu上拷贝过来的	2012-2-1	下载	地址2

图 6.1 下载文件

(2) 运行下载的安装包，首先出现的是向导欢迎界面。如图 6.2 所示。单击“Next”按钮。

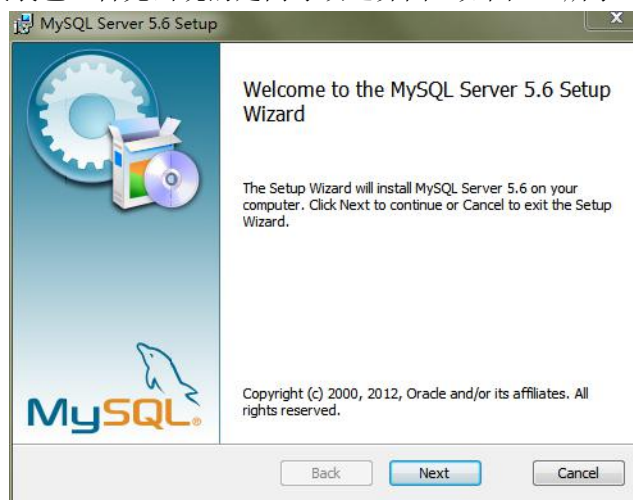


图 6.2 欢迎界面

(3) 该界面选择同意条款。如图 6.3 所示。单击“Next”按钮。

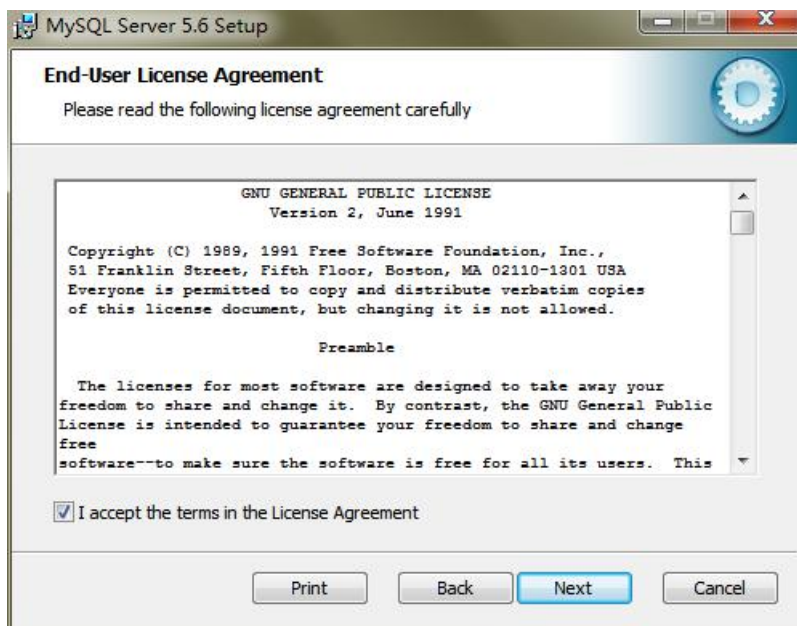


图 6.3 服务条款界面

(4) 下面选择定制安装“Custom”。如图 6.4 所示。

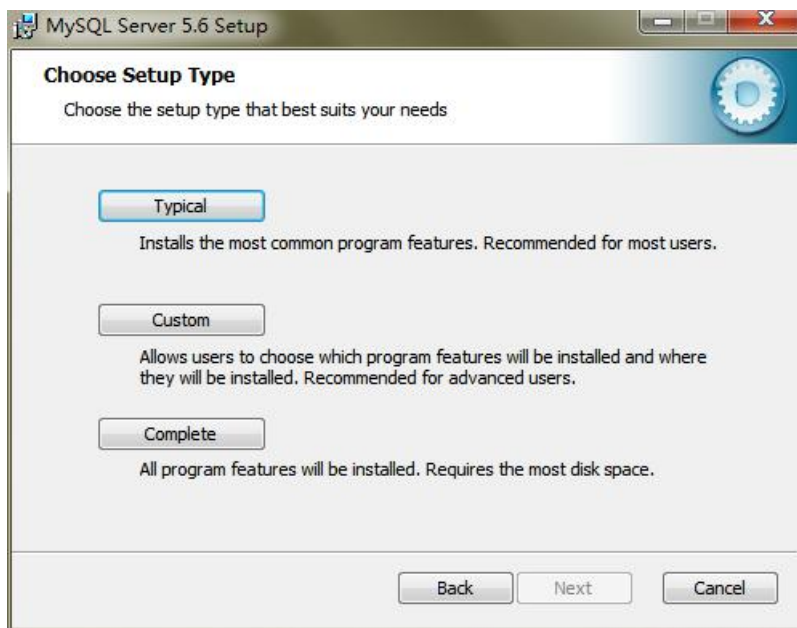


图 6.4 选择定制安装

(5) 这里需要安装所有的头文件和库，点击 Development Components 前面的下拉箭头，然后选择第二项。如图 6.5 所示。

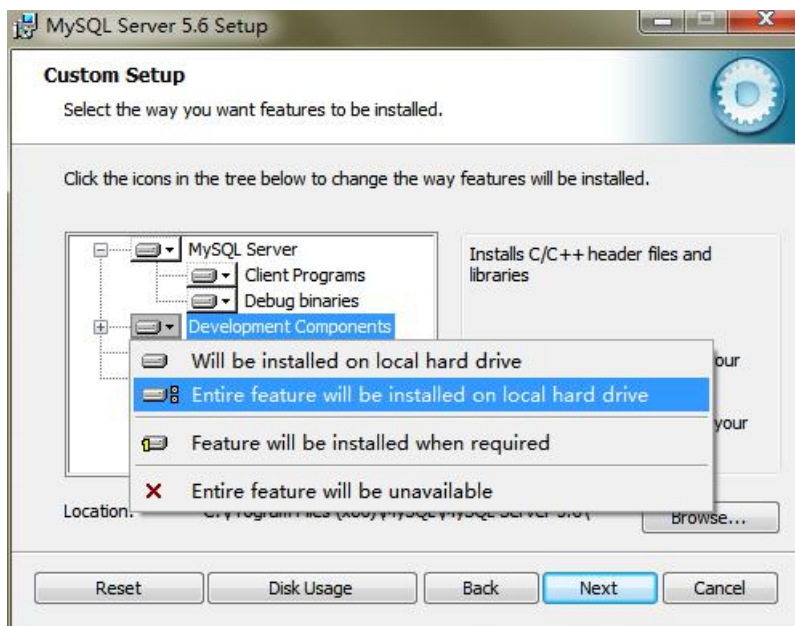


图 6.5 选择安装的文件

(6) 然后选择下面的“Browse...”按钮来更改安装路径，这里设置为 C:\MySQL\，如图 6.6 所示。

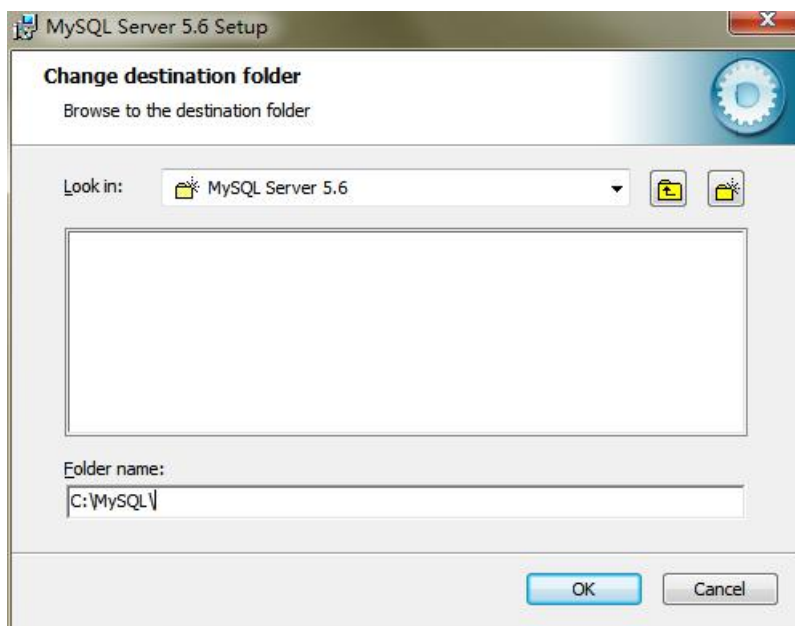


图 6.6 设置安装位置

(7) 填写完路径后单击“OK”按钮回到主页面，单击“Next”按钮来到新的页面，这里单击“Install”按钮开始安装。如图 6.7 所示。

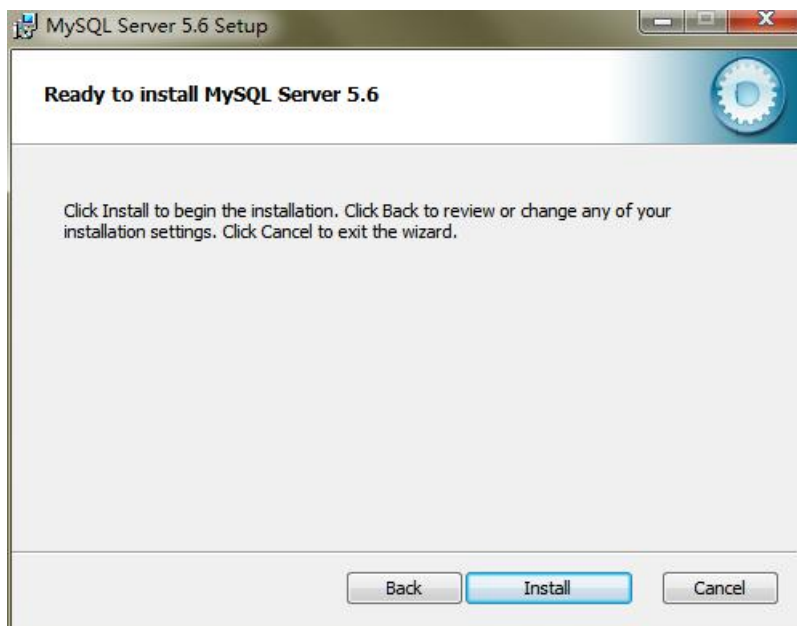


图 6.7 开始安装

(8) 等安装完毕后，点击“Finish”按钮完成安装。如图 6.8 所示。

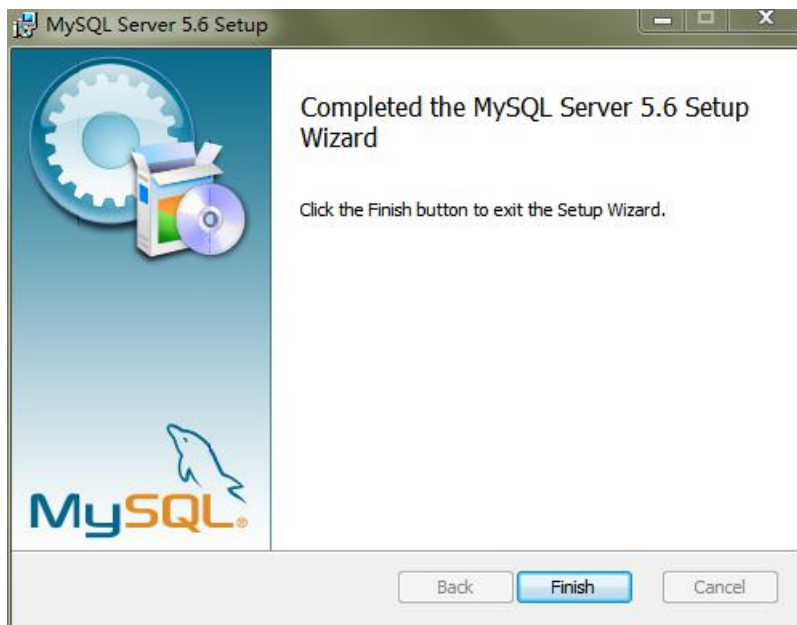


图 6.8 完成安装

2.在 MySQL 中创建数据库

(1) 下面先在安装的 MySQL 中创建一个数据库，用于后面的测试。首先到 MySQL 的安装目录 C:\MySQL\bin 目录下运行 `mysqld.exe` 程序，该程序运行完成后会自动关闭。如图 6.9 所示。

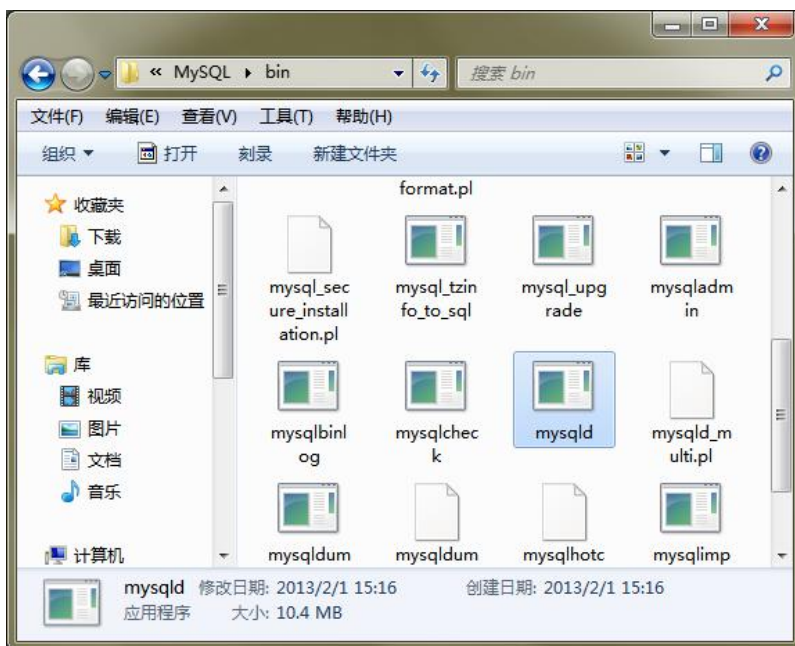


图 6.9 运行 mysqld 程序

(2) 同时按下键盘上的 Win 图标和 R 键，在弹出的对话框中输入 cmd。如图 6.10 所示。

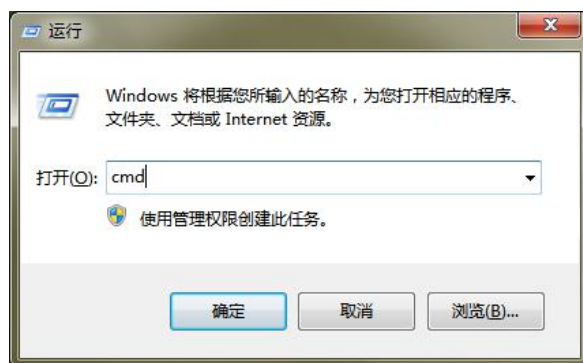


图 6.10 运行 cmd

(3) 进入终端后输入下面的命令：

```
cd C:\MySQL\bin
```

跳转到安装目录下。如图 6.11 所示。



图 6.11 跳转目录

(4) 然后输入下面的命令：

```
mysql -uroot -p
```

使用 root 用户来登录 MySQL，因为默认密码是空的，所以这里不用设置密码。运行这行代码会提示 Enter password，这时敲回车即可。如图 6.12 所示。



图 6.12 登录 MySQL

(5) 登录 MySQL 以后，使用下面的命令来查看现有的数据库：

```
show databases;
```

注意后面有个分号。如图 6.13 所示。

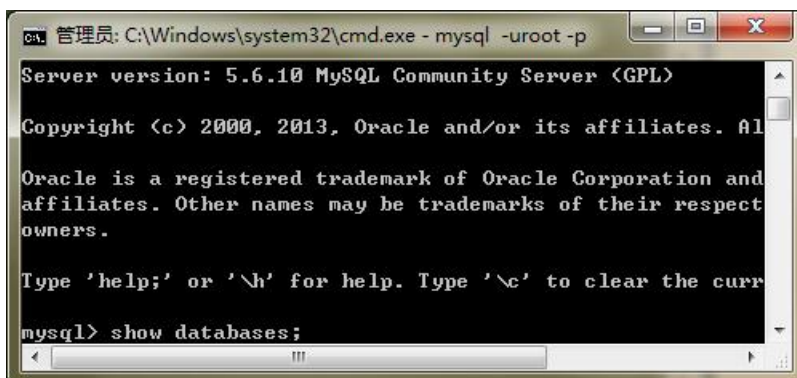


图 6.13 显示数据库

(6) 可以看到，这里现在已经有几个数据库了，它们是 MySQL 需要的。如图 6.14 所示。

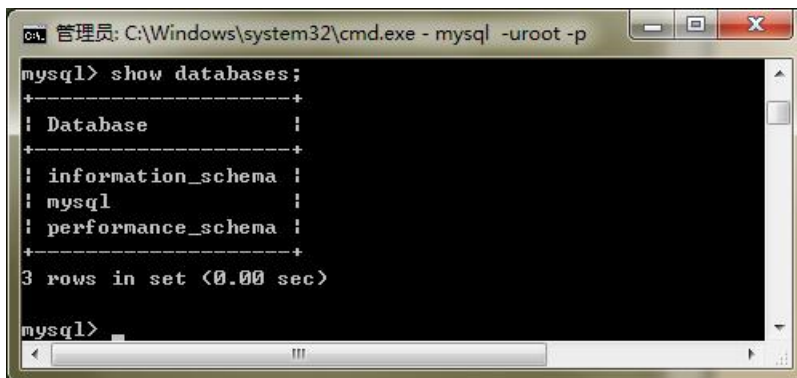
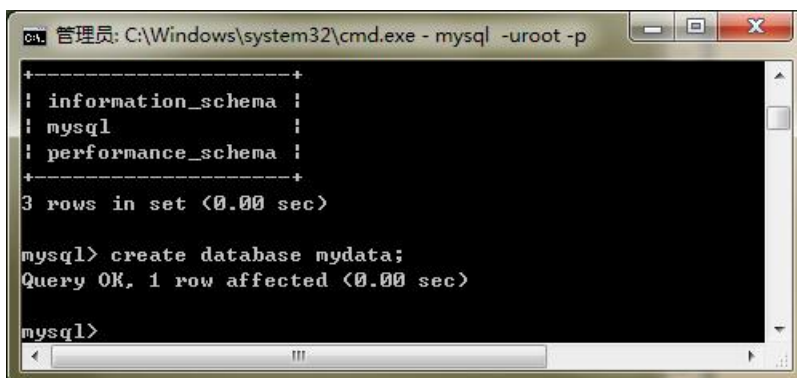


图 6.14 已有的数据库

(7) 这里不使用已有的数据库，而是新建自己的数据库，下面新建名为 `mydata` 的数据库：

```
create database mydata;
```

如图 6.15 所示。



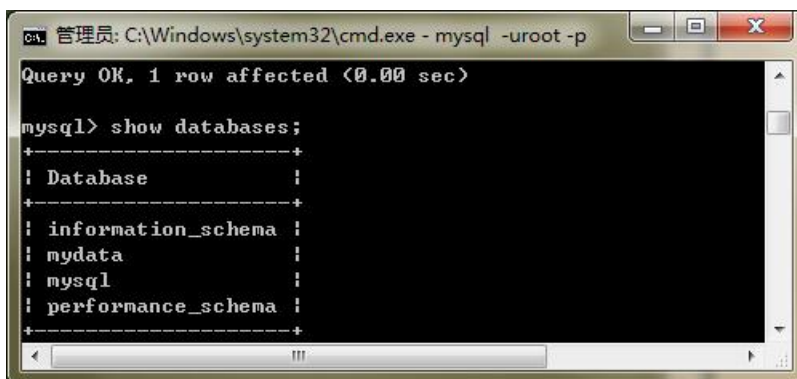
```
管理员: C:\Windows\system32\cmd.exe - mysql -uroot -p
+-----+
| information_schema |
| mysql              |
| performance_schema |
+-----+
3 rows in set (0.00 sec)

mysql> create database mydata;
Query OK, 1 row affected (0.00 sec)

mysql>
```

图 6.15 创建数据库

(8) 再次查看已经存在的数据库，发现显示出了刚才创建的数据库，如图 6.16 所示。



```
管理员: C:\Windows\system32\cmd.exe - mysql -uroot -p
Query OK, 1 row affected (0.00 sec)

mysql> show databases;
+-----+
| Database |
+-----+
| information_schema |
| mydata      |
| mysql      |
| performance_schema |
+-----+
```

图 6.16 再次查看数据库

(9) 完成后，输入 `exit` 命令退出 MySQL。

3.测试 MySQL 程序

(1) 打开 Qt Creator，新建项目，模板选择为“Qt Console Application”，项目名称为“sqldrivers”。完成后在 `sqldrivers.pro` 文件中添加如下代码：

```
QT += sql
```

然后按下 `Ctrl+S` 保存该文件。

(2) 更改 `main.cpp` 文件内容如下。

```
#include <QCoreApplication>
#include <QSqlDatabase>
#include <QDebug>
#include <QStringList>
#include <QSqlQuery>
```

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

    // 输出可用数据库
    qDebug() << "Available drivers:";
    QStringList drivers = QSqlDatabase::drivers();
    foreach(QString driver, drivers)
        qDebug() << driver;

    // 打开 MySQL
    QSqlDatabase db = QSqlDatabase::addDatabase("QMYSQL");
    db.setHostName("localhost");
    db.setDatabaseName("mydata");
    db.setUserName("root");
    db.setPassword("");
    if (!db.open())
        qDebug() << "Failed to connect to root mysql admin";
    else qDebug() << "open";

    QSqlQuery query(db);

    //注意这里 varchar 一定要指定长度，不然会出错
    query.exec("create table student(id int primary key,name varchar(20))");

    query.exec("insert into student values(1,'xiaogang')");
    query.exec("insert into student values(2,'xiaoming')");
    query.exec("insert into student values(3,'xiaohong')");

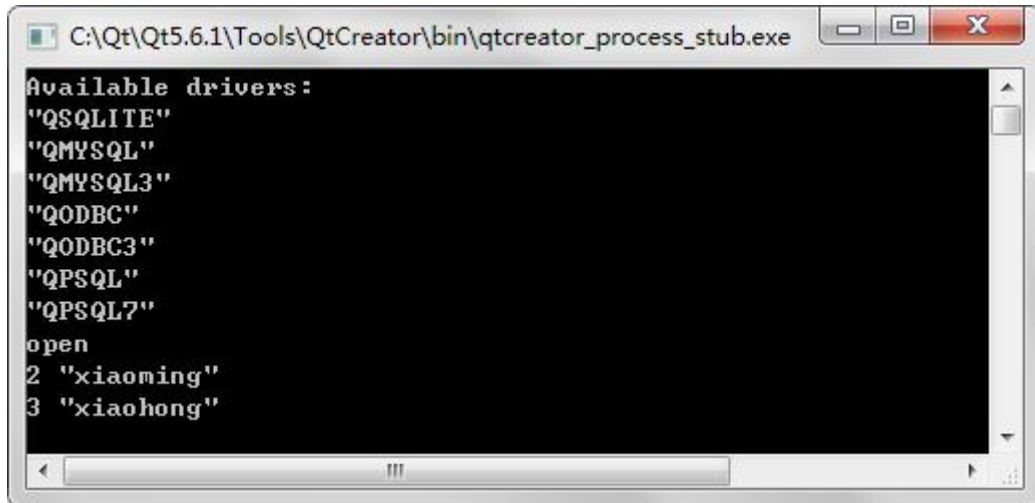
    query.exec("select id,name from student where id >= 2");

    while(query.next())
    {
        int value0 = query.value(0).toInt();
        QString value1 = query.value(1).toString();
        qDebug() << value0 << value1 ;
    }

    return a.exec();
}
```

这里注意，创建表时 `varchar` 一定要指定长度。

(3) 到 `C:\MySQL\lib` 中将 `libmysql.dll` 文件复制到 `C:\Qt\Qt5.6.1\5.6\mingw49_32\bin` 中。运行程序，结果如图 6.17 所示。



```
C:\Qt\Qt5.6.1\Tools\QtCreator\bin\qtcreator_process_stub.exe
Available drivers:
"SQLITE"
"MYSQL"
"MYSQL3"
"ODBC"
"ODBC3"
"PSQL"
"PSQL7"
open
2 "xiaoming"
3 "xiaohong"
```

图 6.17 程序运行结果

思考题： 使用 MySQL 数据库要注意哪些事项？

实验 7 数据库基本操作

目的与要求

- (1) 掌握操作数据库表格的基本方法
- (2) 掌握使用 Qt 操作 MySQL 数据库的常用操作
- (3) 掌握在 MySQL 中使用中文的方法
- (4) 了解 Qt 调用外部程序的基本方法

实验准备

- (1) 安装完成 MySQL 数据库
- (2) 会使用 Qt 创建 MySQL 数据库表
- (3) 了解 QSqlTableModel 的基本操作

实验内容

(1) 新建 Qt Widgets 应用，项目名称为 sqlModel，类名为 MainWindow，基类选择 QMainWindow。

(2) 完成后在 sqlModel.pro 文件中添加如下代码：

```
QT += sql
```

然后按下 Ctrl+S 保存该文件。

(3) 往项目中添加新的 C++ 头文件，名称为 “connection.h”，完成后在其中添加数据库连接函数的定义：

```
#ifndef CONNECTION_H
#define CONNECTION_H

#include <QMessageBox>
#include <QSqlDatabase>
#include <QSqlQuery>

static bool createConnection()
{
    QSqlDatabase db = QSqlDatabase::addDatabase("QMYSQL");
    db.setHostName("localhost");
    db.setDatabaseName("mydata");
    db.setUserName("root");
}
```

```
db.setPassword("");

if (!db.open()) {
    QMessageBox::critical(0, QObject::tr("无法打开数据库"),
        "无法创建数据库连接! ", QMessageBox::Cancel);
    return false;
}

// 下面来创建表
// 如果 MySQL 数据库中已经存在同名的表, 那么下面的代码不会执行
 QSqlQuery query(db);

// 使数据库支持中文
query.exec("SET NAMES 'Latin1'");

// 创建 course 表
query.exec("create table course (id int primary key, "
           "name varchar(20), teacher varchar(20))");
query.exec("insert into course values(0, '数学', '刘老师');");
query.exec("insert into course values(1, '英语', '张老师');");
query.exec("insert into course values(2, '计算机', '白老师');");

return true;
}

#endif // CONNECTION_H
```

(4) 打开 main.cpp 文件, 修改内容如下:

```
#include <QApplication>
#include "mainwindow.h"
#include "connection.h"
#include <QProcess>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    // 这里使用代码来运行 MySQL 数据库
    QProcess process;
    process.start("C:/MySQL/bin/mysqld.exe");

    if (!createConnection()) return 1;
    MainWindow w;
    w.show();
}
```

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

(5) 双击 mainwindow.ui 文件进入设计模式，向界面上拖入 Label、Push Button、Line Edit 和 Table View 等部件，最终效果如图 7.1 所示。



图 7.1 设计界面效果

(6) 打开 mainwindow.h 文件，添加类的前置声明：

```
class QSqlTableModel;
```

然后再定义一个私有对象：

```
private:
```

```
    QSqlTableModel *model;
```

(7) 下面到 mainwindow.cpp 文件中，添加头文件包含：

```
#include <QSqlQuery>
```

```
#include <QSqlTableModel>
```

```
#include <QSqlError>
```

```
#include <QMessageBox>
```

在构造函数中添加如下代码：

```
model = new QSqlTableModel(this);
```

```
model->setTable("course");
```

```
model->select();
```

```
// 设置编辑策略
```

```
model->setEditStrategy(QSqlTableModel::OnManualSubmit);
```

```
ui->tableView->setModel(model);
```

(8) 下面到设计模式，分别右击各个按钮，选择“转到槽”，然后选择 clicked()信号。更改各个槽函数的内容如下：

```
// 提交修改按钮
```

```
void MainWindow::on_pushButton_clicked()
```

```
{
    // 开始事务操作
    model->database().transaction();
    if (model->submitAll()) {
        model->database().commit(); //提交
    } else {
        model->database().rollback(); //回滚
        QMessageBox::warning(this, tr("tableModel"),
                               tr("数据库错误: %1").arg(model->lastError().text()));
    }
}

// 撤销修改按钮
void MainWindow::on_pushButton_2_clicked()
{
    model->revertAll();
}

// 查询按钮，进行筛选
void MainWindow::on_pushButton_7_clicked()
{
    QString name = ui->lineEdit->text();
    //根据姓名进行筛选，一定要使用单引号
    model->setFilter(QString("teacher = '%1'").arg(name));
    model->select();
}

// 显示全表按钮
void MainWindow::on_pushButton_8_clicked()
{
    model->setTable("course");
    model->select();
}

// 按 id 升序排列按钮
void MainWindow::on_pushButton_5_clicked()
{
    //id 属性，即第 0 列，升序排列
    model->setSort(0, Qt::AscendingOrder);
    model->select();
}

// 按 id 降序排列按钮
void MainWindow::on_pushButton_6_clicked()
```



```
{
    model->setSort(0, Qt::DescendingOrder);
    model->select();
}

// 删除选中行按钮
void MainWindow::on_pushButton_4_clicked()
{
    // 获取选中的行
    int curRow = ui->tableView->currentIndex().row();
    // 删除该行
    model->removeRow(curRow);
    int ok = QMessageBox::warning(this, tr("删除当前行!"),
        tr("你确定删除当前行吗? "), QMessageBox::Yes, QMessageBox::No);
    if(ok == QMessageBox::No)
    { // 如果不删除, 则撤销
        model->revertAll();
    } else { // 否则提交, 在数据库中删除该行
        model->submitAll();
    }
}

// 添加记录按钮
void MainWindow::on_pushButton_3_clicked()
{
    // 获得表的行数
    int rowNum = model->rowCount();
    // 添加一行
    model->insertRow(rowNum);
    model->setData(model->index(rowNum,0),rowNum);
}
```

(9) 运行程序。然后修改“刘老师”为“陈老师”，并单击“提交修改”按钮，如图 7.3 所示。然后将“张老师”修改为“李老师”并按下回车键，虽然这里已经修改了，但是并没有实际写入到数据库，单击“撤销修改”按钮，可以看到又改为了“张老师”。单击“添加记录”按钮，会在最下面添加一行，前面的星号表示该数据还没有写入到数据库，这里 id 默认为已有的最大 id 值加 1，如图 7.4 所示。现在添加 name 为“语文”，teacher 为“周老师”，然后单击“提交修改”按钮，效果如图 7.5 所示。下面单击新添加的行，然后单击“删除选中行”，就可以删除该行。在“姓名”中输入“白老师”，然后单击“查询”按钮，就可以显示“白老师”一行，效果如图 7.6 所示。单击“显示全表”按钮可以再次显示所有内容。单击“按 id 降序排列”按钮，可以按照 id 从大到小排列各行，效果如图 7.7 所示。



图 7.2 程序运行效果



图 7.3 提交修改



图 7.4 添加记录



图 7.5 提交添加的记录



图 7.6 查询效果



图 7.7 排序效果

思考题： 使用其他数据库重写编写该程序，思考 Qt 操作数据库与数据库类型有关吗？

实验 8 Qt 数据库应用编程（综合设计）

目的与要求

- (1) 掌握创建综合应用程序的方法
- (2) 掌握设计数据库程序的方法
- (3) 掌握综合程序的布局方法
- (4) 了解模块化编写应用程序的方法

实验准备

- (1) 安装完成 MySQL 数据库
- (2) 会使用 Qt 创建 MySQL 数据库表
- (3) 了解应用程序界面布局的方法
- (4) 会使用常用的 Qt 部件

实验内容

实验的目标

该实验的目标是实现一个数据管理系统，使用密码登陆系统后，通过对数据库的操作实现商品的出售、进货、查询等功能。实验中使用了电视和空调两种类型的商品作为例子来进行演示，一共创建两张数据库表：一张类型表和一张品牌表。类型表中存放所有家电的类型，这里以电视和空调为例；在品牌表中保存所有品牌所属的家电类型、总量、销售量和库存量等数据。

实验步骤

1. 创建程序

(1) 新建 Qt Widgets 应用，项目名称为“manager”，基类选择为 QWidget，类名保持“Widget”不变。

(2) 完成后进入设计模式，将主界面的宽度和高度分别设置为 750 和 500，windowTitle 属性设置为“数据管理系统”。然后向界面中拖入五个 Push Button，调整它们的大小，并分别更改其显示文本为“出售商品”、“商品入库”、“添加商品”、“商品查询”和“修改密码”。然后将它们放入一个水平布局管理器中。下面向界面上拖入一个 Stacked Widget，将其宽度和高度分别修改为 700 和 410，然后将 currentPageName 属性修改为“sellPage”。向 Stacked Widget 的当前页面即 sellPage 中拖入一个 Label 部件，将 Label 的显示文本更改为“出售商品”，将其 font 属性的点大小设置为 12，将 frameShape 属性选择为 StyledPanel，将 alignment 属性中“水平的”选择为 AlignHCenter。然后再向界面上添加 Label、Combo Box、Line Edit、Spin Box 和 Push Button 等部件，最终的效果如图 8.1 所示。

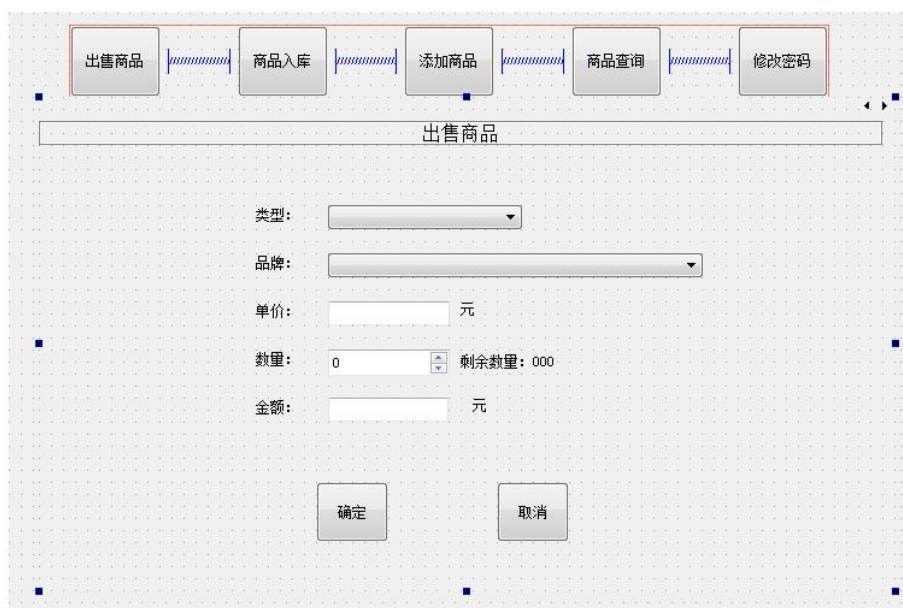


图 8.1 出售商品界面

(3) 下面来更改部分部件的 `objectName` 属性，如表 8.1 所示。

表 8.1 出售商品页面部件的 `objectName` 属性

部件	<code>objectName</code> 属性
“出售商品” Push Button	<code>sellBtn</code>
“商品入库” Push Button	<code>buyBtn</code>
“添加商品” Push Button	<code>addBtn</code>
“商品查询” Push Button	<code>queryBtn</code>
“修改密码” Push Button	<code>passwordBtn</code>
“类型:” 后面的 Combo Box	<code>sellTypeComboBox</code>
“品牌:” 后面的 Combo Box	<code>sellBrandComboBox</code>
“单价:” 后面的 Line Edit	<code>sellPriceLineEdit</code>
“数量:” 后面的 Spin Box	<code>sellNumSpinBox</code>
“剩余数量: 000” Label	<code>sellLastNumLabel</code>
“金额:” 后面的 Line Edit	<code>sellSumLineEdit</code>
“确定” Push Button	<code>sellOkBtn</code>
“取消” Push Button	<code>sellCancelBtn</code>

(4) 下面在 Stacked Widget 上右击，选择“插入页→在当前页之后”来插入新的页面。在新的页面再次拖入部件，设计效果如图 8.2 所示。部分部件的 `objectName` 属性如表 8.2 所示。

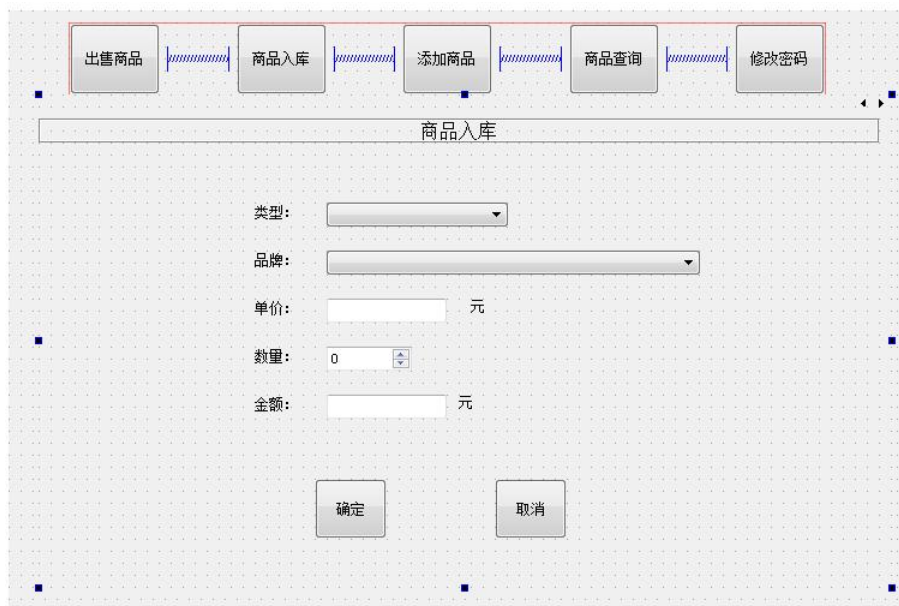


图 8.2 商品入库界面

表 8.2 商品入库页面部件的 objectName 属性

部件	objectName 属性
“类型:”后面的 Combo Box	goodsTypeComboBox
“品牌:”后面的 Combo Box	goodsBrandComboBox
“单价:”后面的 Line Edit	goodsPriceLineEdit
“数量:”后面的 Spin Box	goodsNumSpinBox
“金额:”后面的 Line Edit	goodsSumLineEdit
“确定” Push Button	goodsOkBtn
“取消” Push Button	goodsCancelBtn

(5) 再次插入新的页面，效果如图 8.3 所示。部分部件的 objectName 属性如表 8.3 所示。

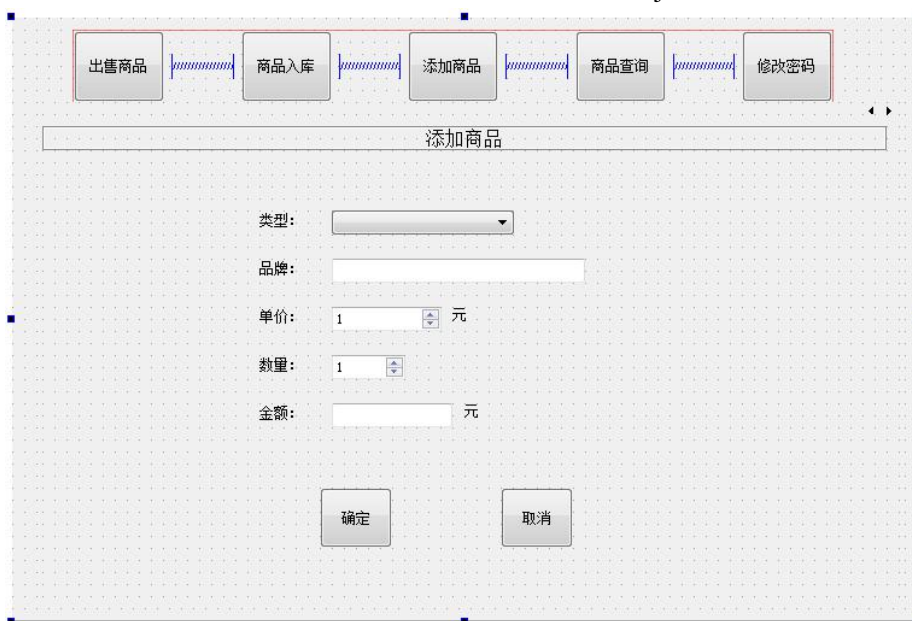


图 8.3 添加商品界面

表 8.3 添加商品页面部件的 objectName 属性

部件	objectName 属性
“类型:” 后面的 Combo Box	newTypeComboBox
“品牌:” 后面的 line Edit	newBrandLineEdit
“单价:” 后面的 Spin Box	newPriceSpinBox
“数量:” 后面的 Spin Box	newNumSpinBox
“金额:” 后面的 Line Edit	newSumLineEdit
“确定” Push Button	newOkBtn
“取消” Push Button	newCancelBtn

(6) 再次插入新的页面，拖入 Label、Combo Box、Push Button 和 Table View 等部件，效果如图 8.4 所示。部分部件的 objectName 属性如表 8.4 所示。

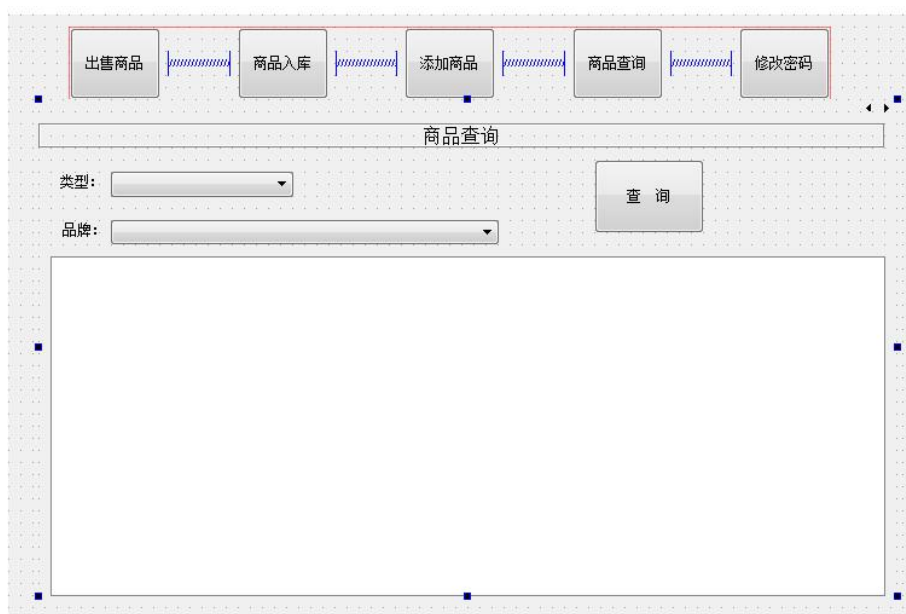


图 8.4 商品查询界面

表 8.4 商品查询页面部件的 objectName 属性

部件	objectName 属性
“类型:” 后面的 Combo Box	queryTypeComboBox
“品牌:” 后面的 Combo Box	queryBrandComboBox
“查询” Push Button	queryPushButton

(7) 再次插入新的页面，效果如图 8.5 所示。部分部件的 objectName 属性如表 8.5 所示。

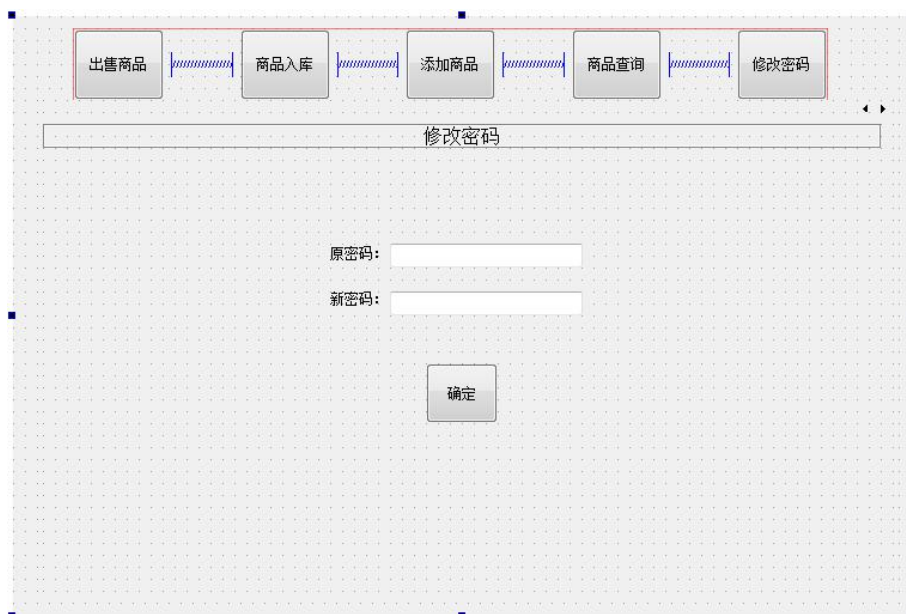


图 8.5 修改密码界面

表 8.5 商品查询页面部件的 objectName 属性

部件	objectName 属性
“原密码”后面的 Line Edit	oldPwdLineEdit
“新密码”后面的 Line Edit	newPwdLineEdit
“确定” Push Button	changePwdBtn

2.连接数据库

(1) 在项目文件 manager.pro 中添加如下代码:

```
QT += sql
```

(2) 往项目中添加新的 C++头文件 connection.h, 完成后将其内容更改如下:

```
#ifndef CONNECTION_H
#define CONNECTION_H

#include <QMessageBox>
#include <QSqlDatabase>
#include <QSqlQuery>

static bool createConnection()
{
    QSqlDatabase db = QSqlDatabase::addDatabase("QMYSQL");
    db.setHostName("localhost");
    db.setDatabaseName("mydata");
    db.setUserName("root");
}
```

```
db.setPassword("");

if (!db.open()) {
    QMessageBox::critical(0, QObject::tr("无法打开数据库"),
        "无法创建数据库连接！ ", QMessageBox::Cancel);
    return false;
}

 QSqlQuery query;

// 使数据库支持中文
query.exec("SET NAMES 'Latin1'");

// 创建分类表
query.exec("create table type(id varchar(20) primary key, name varchar(20))");
query.exec("insert into type values('0', '请选择类型')");
query.exec("insert into type values('01', '电视')");
query.exec("insert into type values('02', '空调')");

// 创建品牌表
query.exec("create table brand(id varchar(20) primary key, name varchar(30), "
    "type varchar(20), price int, sum int, sell int, last int)");
query.exec("insert into brand values('01', '海信', '电视', 3699, 50, 10, 40)");
query.exec("insert into brand values('02', '创维', '电视', 3499, 20, 5, 15)");
query.exec("insert into brand values('03', '海尔', '电视', 4199, 80, 40, 40)");
query.exec("insert into brand values('04', '王牌', '电视', 3999, 40, 10, 30)");
query.exec("insert into brand values('05', '海尔', '空调', 2899, 60, 10, 50)");
query.exec("insert into brand values('06', '格力', '空调', 2799, 70, 20, 50)");

// 创建密码表
query.exec("create table password(id varchar(20) primary key, pwd varchar(50))");
query.exec("insert into password values('01', '123456')");

return true;
}

#endif // CONNECTION_H
```

(3) 下面进入 main.cpp 文件，修改如下：

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



```
#include <QProcess>

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

    // 这里使用代码来运行 MySQL 数据库
    QProcess process;
    process.start("C:/MySQL/bin/mysqld.exe");

    if(!createConnection()) return 0;

    Widget w;
    LoginDialog dlg;
    if (dlg.exec() == QDialog::Accepted) {
        w.show();
        return a.exec();
    } else {
        return 0;
    }
}
```

这里使用了 LoginDialog 类，这个是登陆窗口类，下面来创建这个类。

3.添加登陆窗口类

(1) 向项目中添加新的 Qt 设计师界面类，模板选择 Dialog without Buttons，类名设置为“LoginDialog”。完成后往界面上拖入 Push Button、Line Edit 和 Label 等部件，最终效果如图 8.6 所示。将 Line Edit 的 objectName 属性更改为“pwdLineEdit”，然后将 echoMode 选择为 Password；将“登录”按钮的 objectName 属性设置为“loginBtn”，“退出”按钮的 objectName 属性设置为“quitBtn”。



图 8.6 登陆窗口界面

(2) 下面进入 `logindialog.cpp` 文件中，先添加头文件包含：

```
#include <QMessageBox>
#include <QSqlQuery>
```

然后在构造函数中添加如下代码：

```
setFixedSize(700, 400);
setWindowTitle(tr("登录"));
ui->pwdLineEdit->setFocus();
ui->loginBtn->setDefault(true);
```

下面从设计模式分别进入“登录”按钮和“退出”按钮的单击信号对应的槽，更改如下：

```
void LoginDialog::on_loginBtn_clicked()
{
    if (ui->pwdLineEdit->text().isEmpty()) {
        QMessageBox::information(this, tr("请输入密码"),
                                tr("请先输入密码再登录！"), QMessageBox::Ok);
        ui->pwdLineEdit->setFocus();
    } else {
        QSqlQuery query;
        query.exec("select pwd from password");
        query.next();
        if (query.value(0).toString() == ui->pwdLineEdit->text()) {
            QDialog::accept();
        } else {
            QMessageBox::warning(this, tr("密码错误"),
                                 tr("请输入正确的密码再登录！"), QMessageBox::Ok);
            ui->pwdLineEdit->clear();
            ui->pwdLineEdit->setFocus();
        }
    }
}

void LoginDialog::on_quitBtn_clicked()
{
    QDialog::reject();
}
```

当按下“登录”按钮并且输入的密码正确时执行 `QDialog::accept()` 函数，该函数会隐藏对话框并将返回码设置为 `Accepted`；当按下“退出”按钮时执行 `QDialog::reject()` 函数，该函数会隐藏对话框并将返回值设置为 `Rejected`。

这里使用了对话框的返回值来进行判断，如果返回值为 `QDialog::Accepted`，则显示主界

面，否则退出程序

4.实现功能

(1) 首先双击 `widget.ui` 文件进入设计模式，然后分别右击“出售商品”、“商品入库”、“添加商品”、“商品查询”和“修改密码”等按钮，选择“转到槽”，然后选择 `clicked` 信号，更改相应槽函数如下：

```
// 出售商品按钮
void Widget::on_sellBtn_clicked()
{
    ui->stackedWidget->setCurrentIndex(0);
}

// 商品入库按钮
void Widget::on_buyBtn_clicked()
{
    ui->stackedWidget->setCurrentIndex(1);
}

// 添加商品按钮
void Widget::on_addBtn_clicked()
{
    ui->stackedWidget->setCurrentIndex(2);
}

// 商品查询按钮
void Widget::on_queryBtn_clicked()
{
    ui->stackedWidget->setCurrentIndex(3);
}

// 修改密码按钮
void Widget::on_passwordBtn_clicked()
{
    ui->stackedWidget->setCurrentIndex(4);
}
```

(2) 在各个页面上首先是选择商品的类型，当商品类型改变时会自动修改品牌列表。分别进入各个页面，右击类型后面的 `Combo Box` 部件，选择“转到槽”，选择 `currentIndexChanged(QString)` 信号，更改各个槽函数如下：

```
// 出售商品的类型改变时
void Widget::on_sellTypeComboBox_currentIndexChanged(QString type)
```

```
{
    if (type == "请选择类型") {
        // 进行其他部件的状态设置
        on_sellCancelBtn_clicked();
    } else {
        ui->sellBrandComboBox->setEnabled(true);
        QSqlQueryModel *sellBrandModel = new QSqlQueryModel(this);
        sellBrandModel->setQuery(QString("select name from brand where type=%1")
                                .arg(type));
        ui->sellBrandComboBox->setModel(sellBrandModel);
        ui->sellCancelBtn->setEnabled(true);
    }
}

// 已有商品入库的商品类型改变时
void Widget::on_goodsTypeComboBox_currentIndexChanged(QString type)
{
    if (type == "请选择类型") {
        // 进行其他部件的状态设置
        on_goodsCancelBtn_clicked();
    } else {
        ui->goodsBrandComboBox->setEnabled(true);
        QSqlQueryModel *goodBrandModel = new QSqlQueryModel(this);
        goodBrandModel->setQuery(QString("select name from brand where type=%1")
                                .arg(type));
        ui->goodsBrandComboBox->setModel(goodBrandModel);
        ui->goodsCancelBtn->setEnabled(true);
    }
}

// 新商品入库类型改变时
void Widget::on_newTypeComboBox_currentIndexChanged(QString type)
{
    if (type == "请选择类型") {
        // 进行其他部件的状态设置
        on_newCancelBtn_clicked();
    } else {
        ui->newBrandLineEdit->setEnabled(true);
        ui->newBrandLineEdit->setFocus();
    }
}

// 查询商品类型改变时
void Widget::on_queryTypeComboBox_currentIndexChanged(const QString type)
```

```
{
    if (type == "请选择类型") {
        ui->queryBrandComboBox->setEnabled(false);
    } else {
        ui->queryBrandComboBox->setEnabled(true);
        QSqlQueryModel *brandModel = new QSqlQueryModel(this);
        brandModel->setQuery(QString("select name from brand where type='%1'").arg(type));
        ui->queryBrandComboBox->setModel(brandModel);
    }
}
```

(3) 类似的，当商品品牌改变时要查询相应的内容，分别进入各个页面，右击品牌后面的 Combo Box 部件，选择“转到槽”，选择 `currentIndexChanged(QString)` 信号，更改各个槽函数如下：

```
// 出售商品的品种改变时
void Widget::on_sellBrandComboBox_currentIndexChanged(QString brand)
{
    ui->sellNumSpinBox->setValue(0);
    ui->sellNumSpinBox->setEnabled(false);
    ui->sellSumLineEdit->clear();
    ui->sellSumLineEdit->setEnabled(false);
    ui->sellOkBtn->setEnabled(false);

    QSqlQuery query;
    query.exec(QString("select price from brand where name='%1' and type='%2'")
              .arg(brand).arg(ui->sellTypeComboBox->currentText()));
    query.next();
    ui->sellPriceLineEdit->setEnabled(true);
    ui->sellPriceLineEdit->setReadOnly(true);
    ui->sellPriceLineEdit->setText(query.value(0).toString());

    query.exec(QString("select last from brand where name='%1' and type='%2'")
              .arg(brand).arg(ui->sellTypeComboBox->currentText()));
    query.next();
    int num = query.value(0).toInt();

    if (num == 0) {
        QMessageBox::information(this, tr("提示"), tr("该商品已经售完! "),
                                QMessageBox::Ok);
    } else {
        ui->sellNumSpinBox->setEnabled(true);
        ui->sellNumSpinBox->setMaximum(num);
        ui->sellLastNumLabel->setText(tr("剩余数量: %1").arg(num));
    }
}
```

```
        ui->sellLastNumLabel->setVisible(true);
    }
}

// 已有商品入库的品牌改变时
void Widget::on_goodsBrandComboBox_currentIndexChanged(QString brand)
{
    ui->goodsNumSpinBox->setValue(0);
    ui->goodsNumSpinBox->setEnabled(true);
    ui->goodsSumLineEdit->clear();
    ui->goodsSumLineEdit->setEnabled(false);
    ui->goodsOkBtn->setEnabled(false);

    QSqlQuery query;
    query.exec(QString("select price from brand where name='%1' and type='%2'")
                .arg(brand).arg(ui->goodsTypeComboBox->currentText()));
    query.next();
    ui->goodsPriceLineEdit->setEnabled(true);
    ui->goodsPriceLineEdit->setReadOnly(true);
    ui->goodsPriceLineEdit->setText(query.value(0).toString());
}

// 新商品品牌改变时
void Widget::on_newBrandLineEdit_textChanged(QString str)
{
    if (str == "") {
        ui->newCancelBtn->setEnabled(false);
        ui->newPriceSpinBox->setEnabled(false);
        ui->newNumSpinBox->setEnabled(false);
        ui->newSumLineEdit->setEnabled(false);
        ui->newSumLineEdit->clear();
        ui->newOkBtn->setEnabled(false);
    } else {
        ui->newCancelBtn->setEnabled(true);
        ui->newPriceSpinBox->setEnabled(true);
        ui->newNumSpinBox->setEnabled(true);
        ui->newSumLineEdit->setEnabled(true);
        qreal sum = ui->newPriceSpinBox->value() * ui->newNumSpinBox->value();
        ui->newSumLineEdit->setText(QString::number(sum));
        ui->newOkBtn->setEnabled(true);
    }
}
}
```

(4) 当商品数量改变时，也要进行相应的操作，分别进入各个页面，右击数量后面的 Spin

Box，选择“转到槽”，选择 valueChanged(int)信号，更改相应槽函数内容如下：

```
// 出售商品数量改变时
void Widget::on_sellNumSpinBox_valueChanged(int value)
{
    if (value == 0) {
        ui->sellSumLineEdit->clear();
        ui->sellSumLineEdit->setEnabled(false);
        ui->sellOkBtn->setEnabled(false);
    } else {
        ui->sellSumLineEdit->setEnabled(true);
        ui->sellSumLineEdit->setReadOnly(true);
        qreal sum = value * ui->sellPriceLineEdit->text().toInt();
        ui->sellSumLineEdit->setText(QString::number(sum));
        ui->sellOkBtn->setEnabled(true);
    }
}

// 已有商品入库数量改变时
void Widget::on_goodsNumSpinBox_valueChanged(int value)
{
    if (value == 0) {
        ui->goodsSumLineEdit->clear();
        ui->goodsSumLineEdit->setEnabled(false);
        ui->goodsOkBtn->setEnabled(false);
    } else {
        ui->goodsSumLineEdit->setEnabled(true);
        ui->goodsSumLineEdit->setReadOnly(true);
        qreal sum = value * ui->goodsPriceLineEdit->text().toInt();
        ui->goodsSumLineEdit->setText(QString::number(sum));
        ui->goodsOkBtn->setEnabled(true);
    }
}

// 新商品数量改变时
void Widget::on_newNumSpinBox_valueChanged(int value)
{
    qreal sum = value * ui->newPriceSpinBox->value();
    ui->newSumLineEdit->setText(QString::number(sum));
    ui->newOkBtn->setEnabled(true);
}
```

(5) 在添加商品页面，当单价改变时也要更改金额，右击该页面单价后面的 Spin Box，选择“转到槽”，选择 valueChanged(int)信号，更改相应槽函数内容如下：

```
// 新商品单价改变时
void Widget::on_newPriceSpinBox_valueChanged(int value)
{
    qreal sum = value * ui->newNumSpinBox->value();
    ui->newSumLineEdit->setText(QString::number(sum));
    ui->newOkBtn->setEnabled(true);
}
```

(6) 各个页面的确定按钮按下时要通过操作数据库执行一定的操作，下面分别右击各个页面的“确定”按钮，选择“转到槽”，选择 clicked()信号，修改相应的槽内容如下：

```
// 出售商品的确定按钮
void Widget::on_sellOkBtn_clicked()
{
    QString type = ui->sellTypeComboBox->currentText();
    QString name = ui->sellBrandComboBox->currentText();
    int value = ui->sellNumSpinBox->value();
    // sellNumSpinBox 的最大值就是以前的剩余量
    int last = ui->sellNumSpinBox->maximum() - value;

    QSqlQuery query;

    // 获取以前的销售量
    query.exec(QString("select sell from brand where name='%1' and type='%2'"
        .arg(name).arg(type)));
    query.next();
    int sell = query.value(0).toInt() + value;

    // 事务操作
    QSqlDatabase::database().transaction();
    bool rtn = query.exec(
        QString("update brand set sell=%1,last=%2 where name='%3' and
type='%4'"
        .arg(sell).arg(last).arg(name).arg(type)));
    if (rtn) {
        QSqlDatabase::database().commit();
        QMessageBox::information(this, tr("提示"), tr("购买成功! "), QMessageBox::Ok);
    } else {
        QSqlDatabase::database().rollback();
        QMessageBox::information(this, tr("提示"), tr("购买失败, 无法访问数据库! "),
            QMessageBox::Ok);
    }
    on_sellCancelBtn_clicked();
}
```



```
}

// 已有商品入库的确定按钮
void Widget::on_goodsOkBtn_clicked()
{
    QString type = ui->goodsTypeComboBox->currentText();
    QString name = ui->goodsBrandComboBox->currentText();
    int value = ui->goodsNumSpinBox->value();

    QSqlQuery query;
    // 获取以前的总量
    query.exec(QString("select sum from brand where name='%1' and type='%2'")
                .arg(name).arg(type));
    query.next();
    int sum = query.value(0).toInt() + value;

    // 获取以前的剩余量
    query.exec(QString("select last from brand where name='%1' and type='%2'")
                .arg(name).arg(type));
    query.next();
    int last = query.value(0).toInt() + value;

    // 事务操作
    QSqlDatabase::database().transaction();
    bool rtn = query.exec(
        QString("update brand set sum=%1,last=%2 where name='%3' and type='%4'")
            .arg(sum).arg(last).arg(name).arg(type));
    if (rtn) {
        QSqlDatabase::database().commit();
        QMessageBox::information(this, tr("提示"), tr("入库成功! "), QMessageBox::Ok);
    } else {
        QSqlDatabase::database().rollback();
        QMessageBox::information(this, tr("提示"), tr("入库失败, 无法访问数据库! "),
            QMessageBox::Ok);
    }
    on_goodsCancelBtn_clicked();
}

// 新商品的确定按钮
void Widget::on_newOkBtn_clicked()
{
    QString type = ui->newTypeComboBox->currentText();
    QString brand = ui->newBrandLineEdit->text();
    qint16 price = ui->newPriceSpinBox->value();
}
```

```
qint16 num = ui->newNumSpinBox->value();

 QSqlQuery query;
 query.exec("select id from brand");
 query.last();
 qreal temp = query.value(0).toInt() + 1;

 QString id;
 if (temp < 10) {
     id = "0" + QString::number(temp);
 } else {
     id = QString::number(temp);
 }

 // 事务操作
 QSqlDatabase::database().transaction();
 bool rtn = query.exec(QString("insert into brand values('%1', '%2', '%3', %4, %5, 0, %6)")
     .arg(id).arg(brand).arg(type).arg(price).arg(num).arg(num));
 if (rtn) {
     QSqlDatabase::database().commit();
     QMessageBox::information(this, tr("提示"), tr("入库成功! "), QMessageBox::Ok);
 } else {
     QSqlDatabase::database().rollback();
     QMessageBox::information(this, tr("提示"), tr("入库失败, 无法访问数据库! "),
         QMessageBox::Ok);
 }

 on_newCancelBtn_clicked();
}
```

(7) 各个页面的取消按钮按下时要进行部件状态设置操作，下面分别右击各个页面的“取消”按钮，选择“转到槽”，选择 clicked()信号，修改相应的槽内容如下：

```
// 出售商品的取消按钮
void Widget::on_sellCancelBtn_clicked()
{
    ui->sellTypeComboBox->setCurrentIndex(0);
    ui->sellBrandComboBox->clear();
    ui->sellBrandComboBox->setEnabled(false);
    ui->sellPriceLineEdit->clear();
    ui->sellPriceLineEdit->setEnabled(false);
    ui->sellNumSpinBox->setValue(0);
    ui->sellNumSpinBox->setEnabled(false);
    ui->sellSumLineEdit->clear();
}
```

```
    ui->sellSumLineEdit->setEnabled(false);
    ui->sellOkBtn->setEnabled(false);
    ui->sellCancelBtn->setEnabled(false);
    ui->sellLastNumLabel->setVisible(false);
}

// 已有商品入库的取消按钮
void Widget::on_goodsCancelBtn_clicked()
{
    ui->goodsTypeComboBox->setCurrentIndex(0);
    ui->goodsBrandComboBox->clear();
    ui->goodsBrandComboBox->setEnabled(false);
    ui->goodsPriceLineEdit->clear();
    ui->goodsPriceLineEdit->setEnabled(false);
    ui->goodsNumSpinBox->setValue(0);
    ui->goodsNumSpinBox->setEnabled(false);
    ui->goodsSumLineEdit->clear();
    ui->goodsSumLineEdit->setEnabled(false);
    ui->goodsOkBtn->setEnabled(false);
    ui->goodsCancelBtn->setEnabled(false);
}

// 新商品入库的取消按钮
void Widget::on_newCancelBtn_clicked()
{
    ui->newTypeComboBox->setCurrentIndex(0);
    ui->newBrandLineEdit->clear();
    ui->newBrandLineEdit->setEnabled(false);
    ui->newPriceSpinBox->setEnabled(false);
    ui->newPriceSpinBox->setValue(1);
    ui->newNumSpinBox->setEnabled(false);
    ui->newNumSpinBox->setValue(1);
    ui->newSumLineEdit->setText("1");
    ui->newSumLineEdit->setEnabled(false);
    ui->newOkBtn->setEnabled(false);
    ui->newCancelBtn->setEnabled(false);
}
```

(8) 商品查询页面的“查询”按钮通过类型和品牌条件进行查询操作，下面右击“查询”按钮，选择“转到槽”，选择 clicked()信号，修改相应的槽内容如下：

```
// 查询按钮
void Widget::on_queryPushButton_clicked()
{
```

```

model->setTable("brand");
model->setHeaderData(0, Qt::Horizontal, tr("编号"));
model->setHeaderData(1, Qt::Horizontal, tr("品牌"));
model->setHeaderData(2, Qt::Horizontal, tr("分类"));
model->setHeaderData(3, Qt::Horizontal, tr("单价"));
model->setHeaderData(4, Qt::Horizontal, tr("总量"));
model->setHeaderData(5, Qt::Horizontal, tr("卖出"));
model->setHeaderData(6, Qt::Horizontal, tr("剩余"));
if(ui->queryBrandComboBox->isEnabled()) {
    QString type = ui->queryTypeComboBox->currentText();
    QString name = ui->queryBrandComboBox->currentText();
    model->setFilter(QString("type='%1' and name='%2'").arg(type).arg(name));
}
model->select();
}

```

(9) 修改密码页面的“确定”按钮通过操作数据库来修改密码，下面右击“确定”按钮，选择“转到槽”，选择 clicked()信号，修改相应的槽内容如下：

```

// 修改密码确定按钮
void Widget::on_changePwdBtn_clicked()
{
    if (ui->oldPwdLineEdit->text().isEmpty() ||
        ui->newPwdLineEdit->text().isEmpty()) {
        QMessageBox::warning(this, tr("警告"), tr("请将信息填写完整! "),
            QMessageBox::Ok);
    } else {
        QSqlQuery query;
        query.exec("select pwd from password");
        query.next();
        if (query.value(0).toString() == ui->oldPwdLineEdit->text()) {
            bool temp = query.exec(QString("update password set pwd='%1' where id='01'")
                .arg(ui->newPwdLineEdit->text()));
            if (temp) {
                QMessageBox::information(this, tr("提示"), tr("密码修改成功! "),
                    QMessageBox::Ok);
                ui->oldPwdLineEdit->clear();
                ui->newPwdLineEdit->clear();
            } else {
                QMessageBox::information(this, tr("提示"),
                    tr("密码修改失败, 无法访问数据库! "),
                    QMessageBox::Ok);
            }
        }
    }
}

```

```

        QMessageBox::warning(this, tr("警告"), tr("原密码错误, 请重新填写! "),
                               QMessageBox::Ok);
        ui->oldPwdLineEdit->clear();
        ui->newPwdLineEdit->clear();
        ui->oldPwdLineEdit->setFocus();
    }
}
}

```

(10) 下面打开 widget.h 文件, 添加类的前置声明:

```
class QSqlTableModel;
```

然后添加一个私有对象定义:

```
private:
    QSqlTableModel *model;
```

(11) 打开 widget.cpp 文件, 添加头文件包含:

```
#include <QSqlTableModel>
#include <QSqlQuery>
#include <QMessageBox>
```

然后在构造函数中添加一些初始化内容:

```

ui->stackedWidget->setCurrentIndex(0);

QSqlQueryModel *typeModel = new QSqlQueryModel(this);
typeModel->setQuery("select name from type");
ui->sellTypeComboBox->setModel(typeModel);

on_sellCancelBtn_clicked();
on_goodsCancelBtn_clicked();
on_newCancelBtn_clicked();

ui->goodsTypeComboBox->setModel(typeModel);
ui->newTypeComboBox->setModel(typeModel);

model = new QSqlTableModel(this);
ui->tableView->setModel(model);
ui->queryTypeComboBox->setModel(typeModel);
ui->queryBrandComboBox->setEnabled(false);

```

5.发布程序

- (1) 编译运行程序，选择编译运行 Release 版本程序。
- (2) 对程序进行测试，试验各种可能的情况，查看程序运行是否正确。
- (3) 完善程序，并打包发布程序。

思考题： 怎样使用 Qt 编写一个综合应用程序？