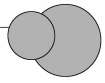
# 第3章

# 初识Qt开发框架



## 3.1 Qt的历史渊源

Qt是1991年由挪威的奇趣科技(Trolltech)公司开发的跨平台C++图形用户界面(GUI)应用程序开发框架。2008年,奇趣科技公司被诺基亚公司收购,Qt也因此成为诺基亚公司旗下的编程语言工具,曾称霸一时的Symbian手机操作系统就是基于Qt开发的。2012年,Qt被芬兰软件公司Digia收购。

Qt最早在Linux系统上大放异彩,它是Linux著名的桌面系统KDE的开发平台。后来又被很多软件公司用来开发重量级产品,其中包括三维动画软件Maya、办公套件WPS、即时通信软件Skype等。它既可以开发GUI程序,也可用于开发非GUI程序。

目前Qt在不同的行业中都取得了不小的成绩,例如能源、医疗、军工和国防、汽车、游戏动画和视觉效果、芯片、消费电子、工业自动化、计算机辅助设计和制造等。以华为公司、中石油公司为代表的客户已经说明了Qt实力的雄厚。

历经二十多年不断发展进步,Qt已经发展成为一个完善的C++开发框架,可以开发出强大的、互动的并且独立于平台的应用程序。Qt的应用程序可以在本地桌面、嵌入式和移动主机系统上运行,其具有的性能远远优于其他跨平台的应用程序开发框架。

Qt具有下列突出优点:

- 优良的跨平台特性。Qt支持的操作系统包括Microsoft Windows、Apple Mac OS X、Linux/X11, Embedded Linux、Windows Embedded、RTOS以及手机上的Android、IOS等。
- 面向对象。Qt的良好封装机制使得Qt的模块化程度非常高,可重用性较好,对于用户开发来说是非常方便的。Qt提供了一种称为signal/slot(信号/槽)的通信机制,这使得各个元件之间的协同工作变得更为简单和安全。
- 丰富的API。Qt包括多达250个以上的C++类,除了用于用户界面开发,还可用于文件操作、数据库处理、网络通信、2D/3D图形渲染、XML操作等。

## 3.2 安装Qt开发系统

### 3.2.1 Qt系统下载

Qt系统可以在官方下载网站http://download.qt.io/archive或者中文的Qtcn开发

网 http://www.qtcn.org 上下载。

本书采用的是 qt-opensource-windows-x86-mingw482_opengl-5.3.2.exe 软件包，也就是 Qt 5.3.2 版本，其官方下载界面如图 3-1 所示。由于在 Qt 发展过程中，其结构有时会有较大变动，因此在本书学习过程中，请尽量选用 Qt 5.3 或更新的版本。

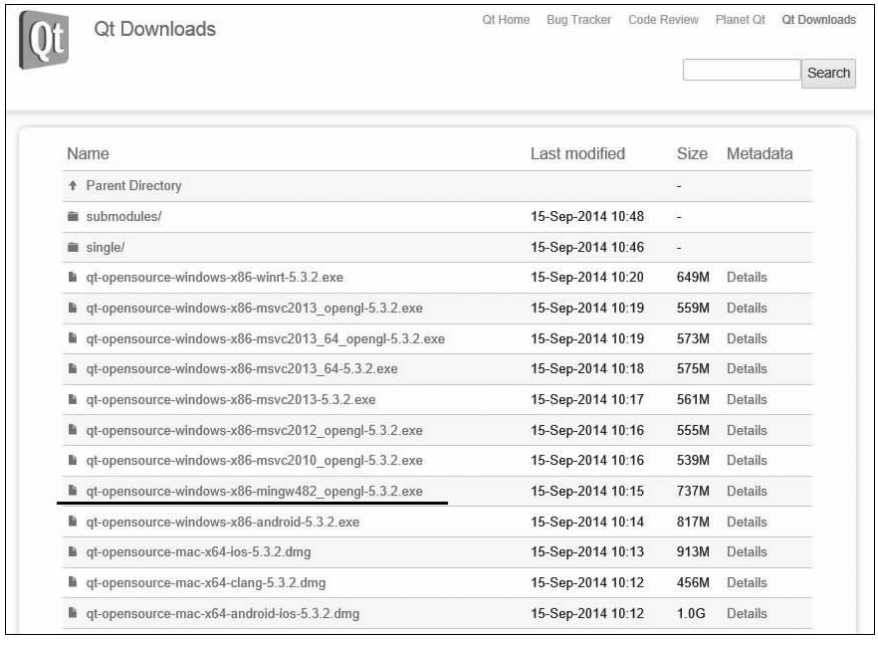



图 3-1　Qt 系统官方下载界面

Qt 本质上是一套 C++ 类库，用于编写 C++ 源程序。Qt 本身没有 C++ 的编译系统，而是采用开源的 MinGW（基于 gcc、gdb）或者微软公司的 msvc 编译系统。本书采用整合 MinGW 的软件包，该软件包无须微软公司的 VC 2012 或 VC 2013 开发环境的支持。

在安装过程中有一步是选择组件，这时一定要将 Tools 中的 MinGW 4.8.2 选中，如图 3-2 所示。另外，在选择软件授权协议的界面，选择 LGPL 协议（GNU 宽通用公共许可证）即可。其他步骤可按默认方式操作。安装完成后，Qt 类库、集成开发环境 Qt Creator、官方例程以及 MinGW 系统就一并安装好了。


图 3-2　在 Qt 安装界面上选择组件

## 3.2.2　Qt Creator 简介

除了可以用手工方式编写基于 Qt 的程序代码，也可以使用官方开发的集成开发环境 Qt Creator。Qt Creator 提供了图形化的界面设计器 Qt Designer，该工具提供了 Qt 基本的窗体部件，如 QWidget（基本窗口）、QLabel（标签）、QPushButton（按钮）等，可以在设计器中通过鼠标直接拖曳这些窗口部件并将其布置到

窗口界面中，从而实现所见即所得的设计。

Qt Creator 启动界面如图 3-3 所示。它的中间部分是主窗口，上部是菜单栏，左侧工具栏主要是模式选择器和一些常用按钮。

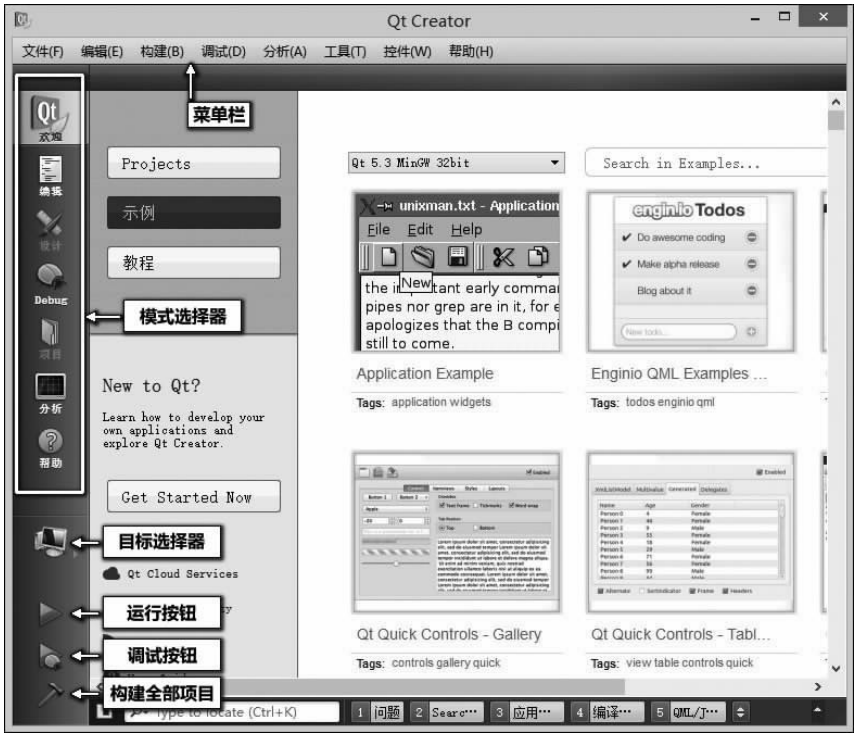



图 3-3　Qt Creator 启动界面

### 1. 菜单栏(Menu Bar)

菜单栏包括以下 8 个菜单。

- “文件”菜单：包含新建、打开、关闭项目和文件，打印文件和退出等基本功能。
- “编辑”菜单：包含撤销、剪切、复制、查找等常用功能，在高级菜单中还有标示空白符、折叠代码、改变字体大小等功能。
- “构建”菜单：包含构建和运行项目等相关的功能。
- “调试”菜单：包含调试程序等相关的功能。
- “分析”菜单：包含 QML 分析器、Valgrind 内存分析器等功能菜单。QML 是 Qt 开发团队创立的一种脚本语言，可以通过描述的方式创建窗体程序。QML 分析器可以分析一段脚本执行过程中出现的问题。而 Valgrind 是一个免费的工具包，用来检测程序运行时内存泄露、越界等问题。
- “工具”菜单：提供了快速定位菜单、版本控制工具菜单和界面编辑器菜单等。其中的“选项”菜单中包含 Qt Creator 各个方面的设置选项，包括环境设置、快捷键设置、编辑器设置、帮助设置、Qt 版本设置、Qt 设计师设置和版本控制设置等。

- “控件”菜单：包含设置窗口布局的一些菜单项，如全屏显示和隐藏边栏等。
- “帮助”菜单：包含 Qt 帮助、Qt Creator 版本信息和插件管理等菜单项。

**2. 模式选择器（Mode Selector）**

Qt Creator 包含欢迎、编辑、设计、调试(Debug)、项目、分析和帮助 6 个模式，各个模式完成不同的功能。也可以使用快捷键来更换模式，对应的快捷键依次是 Ctrl＋1～6。下面简单介绍主要的几种模式。

- 编辑模式：主要用来查看和编辑程序代码，管理项目文件。Qt Creator 的编辑器具有关键字特殊颜色显示、代码自动补全、声明定义间快捷切换、函数原型提示、Fl 键快速打开相关帮助和在项目中进行查找等功能。
- 设计模式：整合了 Qt 设计师的功能。可以在这里设计图形界面，进行部件属性设置、信号和槽设置、布局设置等操作。
- 调试模式：Qt Creator 默认使用 gdb 进行调试，支持设置断点、单步调试和远程调试等功能，包含局部变量、监视器、断点、线程以及快照等查看窗口。
- 项目模式：包含对特定项目的构建设置、运行设置、编辑器设置和依赖关系等页面。构建设置中可以对项目的版本、使用 Qt 的版本和编译步骤进行设置；编辑器设置中可以设置文件的默认编码。

**3. 常用按钮**

Qt Creator 启动界面左下角包含目标选择器、运行按钮、调试按钮和构建全部项目 4 个按钮图标。目标选择器用来选择要构建哪个平台的项目，这对于多个 Qt 库的项目很有用。还可以选择编译项目的 debug 版本或 release 版本。运行按钮可以实现项目的构建和运行。调试按钮可以进入调试模式。构建全部项目按钮可以构建所有打开的项目。

## 3.3　创建一个简单程序

本节以手工编码和图形化操作方式建立两个同样的“Hello Qt!”程序。

**注意**：*在建立项目时，项目的路径和名称都不要使用中文。*

### 3.3.1　手工编码方式

**【例 3-1】** 利用手工编码方式建立“Hello Qt!”程序。

第 1 步，利用 Qt Creator 的菜单“文件→新建文件或项目”打开新建对话框，选择“其他项目→空的 Qt 项目”建立一个名为 3_1 的工程。这时工程中除了名为 3_1.pro 工程文件外无任何其他文件。

第 2 步，再次打开新建对话框，选择 C++ 项目下的 C++ Source File，添加一个 C++ 源程序 q1.cpp(名称可以任取)。

第 3 步，单击打开工程文件 3_1.pro，在末尾行添加文字：QT ＋＝widgets。这样便可以在工程中使用可视化的部件。

第 4 步，在源程序 q1.cpp 中添加如下代码：

```
#include <QApplication>
#include <QDialog>
#include <QLabel>
int main(int argc, char * argv[])
{
    QApplication a(argc,argv);
    QDialog w;
    QLabel label(&w);
    label.setText("Hello Qt!");
    label.setGeometry(10,10,100,100);
    w.show();
    return a.exec();
}
```

至此，一个完整的 Qt 程序就完成了。单击运行按钮（图标为▶，对应快捷方式为 Ctrl＋R 键），即可得到图 3-4 所示的窗体。


图 3-4　工程 3_1 运行界面

代码说明：

第 1～3 行包含了头文件。其中 2、3 两行说明可使用对话框类和标签类。

第 4 行是 C++ 中的 main 函数，它有两个参数，用来接收命令行参数。

第 6 行新建 QApplication 类对象，用于管理应用程序的各种设置，并执行事件处理工作，任何一个 Qt GUI 程序都要有一个 QApplication 对象。该对象需要 argc 和 argv 两个参数。

第 7 行新建一个 QDialog 对象，实现一个对话框界面。

第 8 行新建了标签 QLabel 对象，并将 QDialog 对象 w 作为参数，表明对话框 w 是它的父窗口，也就是说这个标签放在对话框窗口中。

第 9 行给标签设置要显示的字符。

第 10 行设置标签相对于对话框的位置和大小，使用了函数 void setGeometry(int x, int y,int w,int h)，其中 x、y 设置标签在对话框中的坐标，w 为宽，h 为高。GUI 控件都有这个函数。

第 11 行将对话框显示出来。在默认情况下，窗口部件对象是不可见的，要使用 show 函数让它们显示出来。

第 12 行的 exec 函数让 QApplication 对象进入事件循环，这样 Qt 应用程序在运行时便可以接收产生的事件，例如鼠标单击和键盘按下等事件。

## 3.3.2　无 UI 的向导方式

所谓 UI 是指程序界面描述文件，可用于可视化界面设计。

**【例 3-2】**　利用无 UI 的应用程序向导建立“Hello Qt!”程序。

第 1 步，建立无 UI 的工程。

利用 Qt Creator 的菜单的“文件→新建文件或项目”打开新建对话框，选择其中“应用程序”项目的 Qt Widgets Application 选项建立名为 3_2 的工程。在设置类信息的界面(图 3-5)中选择类名为 QDialog，同时取消勾选“创建界面”项目。其他设置采用默认值，最终将建立工程 3_2，其结构如图 3-6 所示。


图 3-5　设置类信息


图 3-6　工程 3_2 的结构

从图 3-5 中会发现向导为窗体程序提供了 3 个基类，分别为 QMainWindow、QWidget、QDialog，3 个基类的区别说明如下。

- QMainWindow 类提供一个有菜单栏、工具栏和一个状态栏的应用程序窗口。
- QWidgt 类是所有用户界面对象的基类。它从窗口系统接收鼠标、键盘和其他事件，并且在屏幕上绘制自己。
- QDialog 类是对话框窗口的基类。对话框窗口是主要用于短期任务以及和用户进行简要通信的顶级窗口。

工程建立完毕后，有 4 个文件：3_2. pro、dialog. h、dialog. cpp、main. cpp。其中头文件 dialog. h 和源文件 dialog. cpp 共同实现了仅属于本项目的对话框类 Dialog。类 Dialog 派生自 Qt 的基本对话框类 QDialog。双击图 3-6 中任一文件即可在主窗口编辑器中打开。

在 main 函数头部可找到如下语句：

```
#include "dialog.h"
```

这样在 main 函数中就可以使用向导自动生成的 Dialog 类。事实上，在主函数中的确定义了 Dialog 类对象 w，并将其显示出来。这些均是自动生成的。

有了派生类 Dialog，只需在其中添加控件即可。

第 2 步，在派生类 Dialog 中添加控件。

修改头文件 dialog. h 如下：

```
#include <QDialog>
#include <QLabel>                              //添加头文件
class Dialog : public QDialog
{
    Q_OBJECT
    QLabel * label;                            //添加标签指针
public:
    Dialog(QWidget * parent=0);
    ~Dialog();
};
```

在 dialog.cpp 中修改构造函数：

```
Dialog::Dialog(QWidget * parent) : QDialog(parent)
{
    resize(150,150);                           //设定对话框大小
    label=new QLabel(this);                    //新建 QLabel 对象
    label->setText("Hello Qt!");               //设置文字内容
    label->setGeometry(0,0,100,100);           //设置标签位置、大小
}
```

至此，程序编写完毕，编译运行即可。运行界面和例 3-1 一致。

利用无可视化设计界面的向导生成的工程比手工方式多了一个类。这样使得控件的添加、设定都在这个类中进行，程序模块化更好一些。

### 3.3.3 Qt 设计器方式

借助 Qt 设计器(Qt Designer)可以以所见即所得的方式构建 GUI 程序。

**【例 3-3】** 利用 Qt 设计器建立“Hello Qt!”程序。

第 1 步，建立含有 UI 的工程。

本步骤与例 3-2 中建立工程的步骤唯一的不同是在“设置类信息”界面(图 3-5)中保持“创建界面”为勾选状态。

建立的工程为 3_3，其中有 5 个文件：3_3.pro、dialog.cpp、dialog.h、main.cpp、dialog.ui。其中 dialog.h 和 dialog.cpp 共同实现了 Dialog 类。dialog.ui 是用于可视化工具 Qt 设计器的文件。

dialog.ui 实质是一个 XML 文件，用于描述 GUI 界面。双击 UI 文件就启动了 Qt 设计器，如图 3-7 所示。

Qt 设计器主要有以下 6 个区域(分别对应图 3-7 中的①～⑥)。

(1) 设计区：就是图 3-7 正中间的部分，主要用来布置各个窗口部件。

(2) 部件列表窗：这里分类罗列了各种常用的标准部件，可以使用鼠标将这些部件拖入主设计区中，放到主设计区的界面上。

(3) 对象查看器：这里列出了界面上所有部件的对象名称和父类，而且以树形结构显示了各个部件的所属关系。可以在这里单击对象来选中某个部件。

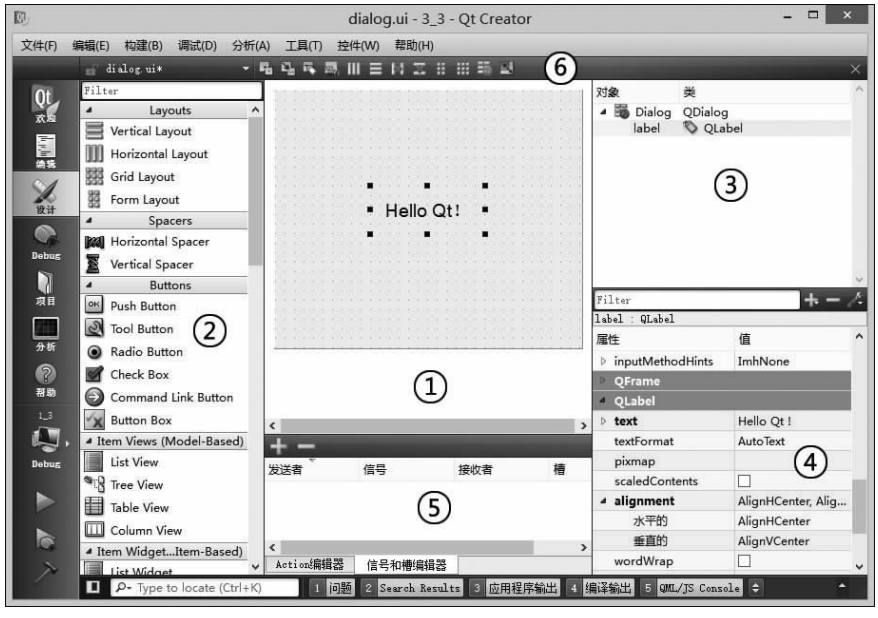



图 3-7　Qt Designer 界面

(4) 属性编辑器：这里显示了各个部件的常用属性信息，可以在这里更改部件的一些属性，如大小、位置等。这些属性按照从祖先继承的属性、从父类继承的属性和自己的属性的顺序进行了分类。

(5) Action(动作)编辑器、信号和槽编辑器：这些和菜单命令、控件事件响应等相关，放到以后使用时再介绍。

(6) 常用功能工具栏：该工具栏中前 4 个按钮用于进入相应的模式，分别是窗口部件编辑模式(这是默认模式)、信号/槽编辑模式、伙伴编辑模式和 Tab 顺序编辑模式。后面几个按钮用来实现添加布局管理器以及调整控件大小等功能。

第 2 步，在对话框中添加控件。

从部件列表窗中拖曳一个 Label(标签)到主窗体中，可以在标签属性编辑器中修改字体、对齐方式等属性，如图 3-7 所示。

程序编写完成，编译运行即可。

在这一过程中 Qt 设计器究竟做了哪些工作呢？下面就一探究竟。

首先看一下系统自动生成 main.cpp 的内容：

```
#include "dialog.h"
#include <QApplication>
int main(int argc, char * argv[])
{
    QApplication a(argc, argv);
    Dialog w;                    //创建一个用设计器设计的对话框对象
    w.show();
    return a.exec();
```

```
}
```

显然，这里没有任何添加标签控件的信息。但可以看出，要搞清程序执行过程，关键是搞清创建 w 对象时程序做了什么，也就是 Dialog 类的构造函数做了什么。所以下面先看一下 dialog.h 的内容：

```
#include <QDialog>
namespace Ui {
    class Dialog;                          //放到 Ui 中以便和下面的 Dialog 类区分
}
class Dialog : public QDialog
{
    Q_OBJECT
public:
    explicit Dialog(QWidget *parent=0);
    ~Dialog();
private:
    Ui::Dialog *ui;
};
```

在 Dialog 类中，仍然没有标签控件的踪迹。这里比例 3-2 的 Dialog 类多了 namespace Ui 的声明，是因为在后面的构造函数中要调用 Ui 中的 Dialog 类，也就是 Ui::Dialog 类。注意此 Ui::Dialog 和 Dialog 是完全不同的两个类。这里的变量 ui 指针将指向一个 Ui::Dialog 类对象。

这个 ui 指向的对象是如何被使用的呢？先看看 dialog.cpp 中的内容：

```
#include "dialog.h"
#include "ui_dialog.h"
Dialog::Dialog(QWidget *parent) :
    QDialog(parent),
    ui(new Ui::Dialog)                          //创建 Ui::Dialog 类对象
{
    ui->setupUi(this);                          //调用 Ui::Dialog 中的函数
}
Dialog::~Dialog()
{
    delete ui;
}
```

这里，源程序 dialog.cpp 首先包含了头文件 ui_dialog.h（该文件定义了 Ui::Dialog 类）。然后在构造函数中创建了一个 Ui::Dialog 类对象，其指针就是 ui。指针 ui 在构造函数中立即调用了 Ui::Dialog 中的函数 setupUi。

于是问题的关键是类 Ui::Dialog 是什么，它做了什么事情。在工程的结构图中找不到 Ui::Dialog 类或者 ui_dialog.h 文件。于是我们在 Windows 资源管理器中打开与工程 3_3 目录并列的 build-3_3-Desktop_Qt_5_3_MinGW_32bit-Debug 目录（这个目录用于

生成编译的结果文件),在该目录下发现了文件 ui_dialog.h,其内容如下:

```
/********************************************************************************
** Form generated from reading UI file 'dialog.ui'
** Created by: Qt User Interface Compiler version 5.3.2
** WARNING! All changes made in this file will be lost when recompiling UI file!
********************************************************************************/
#ifndef UI_DIALOG_H
#define UI_DIALOG_H
#include <QtCore/QVariant>
#include <QtWidgets/QAction>
#include <QtWidgets/QApplication>
#include <QtWidgets/QButtonGroup>
#include <QtWidgets/QDialog>
#include <QtWidgets/QHeaderView>
#include <QtWidgets/QLabel>
QT_BEGIN_NAMESPACE
class Ui_Dialog
{
public:
    QLabel * label;
    void setupUi(QDialog * Dialog)
    {
        if (Dialog->objectName().isEmpty())
          Dialog->setObjectName(QStringLiteral("Dialog"));
        Dialog->resize(323, 271);                          //对话框设置大小
        label=new QLabel(Dialog);                          //新建标签对象
        label->setObjectName(QStringLiteral("label"));
        label->setGeometry(QRect(100, 100, 121, 51));      //设置标签位置和大小
        QFont font;
        font.setFamily(QStringLiteral("Arial"));           //设置标签字体
        font.setPointSize(14);
        label->setFont(font);
        label->setAlignment(Qt::AlignCenter);              //设置标签字体对齐方式
        retranslateUi(Dialog);
        QMetaObject::connectSlotsByName(Dialog);
    } //setupUi
    void retranslateUi(QDialog * Dialog)
    {
        Dialog->setWindowTitle(QApplication::translate("Dialog", "Dialog", 0));
        label->setText(QApplication::translate("Dialog","Hello Qt\357\274\
        201", 0));
    } //retranslateUi
};
```

```
namespace Ui {
    class Dialog: public Ui_Dialog {};                    //这里是 public 继承
} //namespace Ui
QT_END_NAMESPACE
#endif //UI_DIALOG_H
```

通过这个文件，可以知道以下几点：

(1) ui_dialog.h 是由 Qt 设计器自动生成的。

(2) 在 ui_dialog.h 中定义了 Ui 命名空间，其中定义了 Ui::Dialog 类，该类实质就是上面文件中的 Ui_Dialog 类。

(3) 在 Ui_Dialog 类中可以发现利用设计器生成的对象。例如 label 就是放入标签控件后自动产生的。

(4) 在 Ui_Dialog 类的 setupUi 函数中传入了一个窗体(即 main 中的对话框 w)，并在其中设置了控件的大小、位置等。

至此，Qt 设计器所做的工作就全清楚了。

首先，Qt 设计器自动生成 Ui::Dialog 类。

然后，在 main 函数被执行时，要创建一个 Dialog 对象，利用如下语句创建：

```
Dialog w;
```

而在创建 w 的时候调用了 Dialog 类的构造函数，其中包含创建 Ui::Dialog 类对象(即 ui 所指向的对象)的语句。

特别注意，在生成 Ui::Dialog 对象时，对话框 Dialog 对象 w 将自身指针(即 this)传递给这个 Ui::Dialog 对象。

最后，Ui::Dialog 对象执行 setupUi 函数，该函数设置了主函数中 Dialog 对象 w 的属性，同时生成各种控件并将其布置在对象 w 中。

与前面利用无 Ui 文件的向导创建工程的不同之处是：前面是直接在 Dialog 类中添加控件，而这里是利用 Ui::Dialog 对象添加控件。由于 GUI 设计器需要自动生成代码，所以才有了 Ui::Dialog 类。注意，它并不是可视化窗体类，而是一个辅助窗体布置控件的类。它有效分离了自动生成的代码和人工输入的代码。

## 3.4 信号和槽通信机制

信号和槽机制是 Qt 的核心机制，可以让编程人员将互不相关的对象绑定在一起，实现对象之间的通信。

声明了信号的对象，当其状态改变时，信号就由该对象发送出去，而且该对象只负责发送信号，它不知道另一端是谁在接收这个信号。槽用于接收和处理信号，一个槽并不知道是否有任何信号与自己相连接。槽实际上只是普通的对象成员函数。当一个信号被发射时，与其相关联的槽将被立刻执行，就像一个正常的函数调用一样。信号与槽机制完全独立于任何 GUI 事件循环。

### 3.4.1　信号

信号(signal)的声明是在一个类的头文件中进行的,Qt 的 signals 关键字指出进入了信号声明区,随后即可声明自己的信号。例如,下面的语句定义了一个信号:

```
signals:
    void stateChanged(int nNewVal);          //定义信号
```

这里 signals 是 Qt 的关键字,而非 C++ 的关键字。信号函数 stateChanged 定义了信号 stateChanged,这个信号带有参数 nNewVal。

信号函数应该满足以下语法约束:

(1) 函数返回值是 void 类型,因为触发信号函数的目的是执行与其绑定的槽函数,无须信号函数返回任何值。

(2) 开发人员只能声明而不能实现信号函数,Qt 的 moc 工具会实现它。

(3) 信号函数被 moc 自动设置为 protected,因而只有包含一个信号函数的那个类及其派生类才能使用该信号函数。

(4) 信号函数的参数个数、类型由开发人员自由设定,这些参数的职责是封装类的状态信息,并将这些信息传递给槽函数。

(5) 只有 QObject 及其派生类才可以声明信号函数。

### 3.4.2　槽

槽(slot)函数和普通的 C++ 成员函数一样,可以被正常调用,槽唯一的特殊性就是很多信号可以与其相关联。当与其关联的信号被发送时,这个槽就会被调用。槽可以有参数,但槽的参数不能有默认值。关键字 slots 表明进入了槽函数声明区。

槽的声明也是在头文件中进行的。例如,下面声明了一个槽:

```
public slots:                                    //此语句说明后面是槽函数
    void Function(int nNewVal)
    {
        qDebug() <<"new Values=" <<nNewVal;  //显示变量
    }
```

槽函数的返回值是 void 类型,因为信号和槽机制是单向的:信号被发送后,与其绑定的槽函数会被执行,但不要求槽函数返回任何执行结果。和信号函数一样,只有 QObject 及其派生类才可以定义槽函数。

既然槽函数是普通的成员函数,因此与其他的函数一样,它们也有存取权限(public、protected、private)。也就是说,人们能够控制其他类能够以怎样的方式调用一个槽函数。而且这些关键字并不影响 QObject::connect 函数(该函数关联信号和槽)。可以将 protected 甚至 private 的槽函数和一个信号函数绑定。当该信号被发射后,即使是 private 的槽函数也会被执行。从某种意义上讲,Qt 的信号和槽机制破坏了 C++ 的存取控制规则,但是这种机制带来的灵活性远胜于可能导致的问题。

### 3.4.3 关联信号与槽

通过调用 QObject::connect 函数可以绑定一个信号函数和一个槽函数，该函数最常用的格式如下：

```
connect(sender, SIGNAL(signal_func()),receiver,SLOT(slot_func()))
```

其中 sender 及 receiver 都是指向 QObject(或其子类)对象的指针，前者指向发送信号的对象，后者指向处理信号的对象，两者分别被称为“发送者”及“接收者”。signal_func 以及 slot_func 分别是这两个对象中定义的信号函数和槽函数。当指定信号 signal 时一般使用 Qt 的宏 SIGNAL，当指定槽函数时必须使用宏 SLOT。

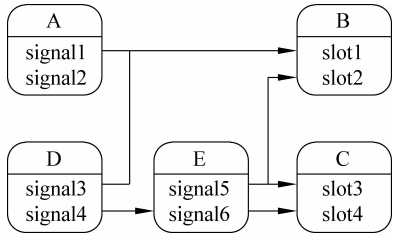


图 3-8 信号函数与槽函数的各种对应关系

一个信号函数可以和多个槽函数绑定。例如在图 3-8 中，对象 E 的 signal5 和 B 的 slot2 以及 C 的 slot3 绑定，当 signal5 被发送时，两个槽函数都会被执行。

多个信号函数可以和一个槽函数绑定。图 3-8 中对象 A 的 signal1 以及 D 的 signal3 都和 B 的 slot1 绑定，其中任何一个信号被发送，槽函数就会被执行。Qt 的信号和槽机制甚至支持两个信号函数之间的绑定。例如 D 的 signal4 和 E 的 signal6 绑定，后者再和 C 的 slot4 绑定。当 signal4 被发送时，signal6 也会随即被发送，导致 slot4 被执行。

使用信号和槽机制时应注意以下问题：

(1) 信号和槽机制与普通函数的调用一样，如果使用不当，在程序执行时也有可能产生死循环。因此，在定义槽函数时一定要注意避免间接形成无限循环，即在槽中再次发送所接收到的同样信号。

(2) 如果一个信号与多个槽相联系，那么当这个信号被发送时，与之相关的槽被激活的顺序将是随机的。

(3) 宏定义不能用在信号和槽的参数中。

(4) 信号和槽的参数个数与类型必须一致。

### 3.4.4 信号和槽举例

**【例 3-4】** 无图形用户界面的简单信号和槽的例子。

信号和槽机制并不需要 GUI 界面，只要发送和接收信号的类都从 CObject 继承即可，同时在类的声明部分要加入 Q_OBJECT 宏。

第 1 步，利用 Qt Creator 的菜单“文件→新建文件或项目”打开新建对话框，选择“其他项目→空的 Qt 项目”建立一个名为 3_4 的工程。

第 2 步，再次打开新建对话框，选择 C++ 项目下的 C++ Header File，添加一个 C++ 头文件 exampleA.h，修改其内容为

```
#include <QCoreApplication>
class CExampleA : public QObject
{
    Q_OBJECT
public:
    CExampleA() { m_Value=0; }
    void SetValue(int nNewVal)
    {
        if(m_Value ==nNewVal)
        {
            return ;
        }
        m_Value=nNewVal;
        emit stateChanged(m_Value);                    //①
    }
signals:
    void stateChanged(int nNewVal);                    //②
private:
    int m_Value;
};
```

第 3 步，再次打开新建对话框，选择 C++ 项目下的 C++ Header File，添加一个 C++ 头文件 exampleB.h，修改其内容为

```
#include <QDebug>
#include <QCoreApplication>
class CExampleB : public QObject
{
    Q_OBJECT
public:
    CExampleB() {  }
public slots:                                          //③
    void Function(int nNewVal)
    {
        qDebug() <<"new Values=" <<nNewVal;
    }
};
```

第 4 步，再次打开新建对话框，选择 C++ 项目下的 C++ Source File，添加一个 C++ 主函数文件 main.cpp，修改其内容为

```
#include "exampleA.h"
#include "exampleB.h"
int main()
{
```

```
    CExampleA a;
    CExampleB b;
    QObject::connect(&a,SIGNAL(stateChanged(int)),&b,SLOT(Function(int)));
    a.SetValue(100);
    return 0;
}
```

编译运行后，在输出窗口看到运行结果如下：

```
Starting F:\QT_learn\a1\build-1_4-Desktop_Qt_5_3_MinGW_32bit-Debug\debug\
1_4.exe...
new Values=100
F:\QT_learn\a1\build-1_4-Desktop_Qt_5_3_MinGW_32bit-Debug\debug\1_4.exe
exited with code 0
```

代码分析：

类 CExampleA 在行②使用 Qt 的关键字 signals 定义了信号函数 stateChanged，当类 CExampleA 的状态改变时，行①“调用”这个信号函数。此处的调用格式与一般情形下的不同，行①使用了 Qt 的关键字 emit，可将其称为“发送”一个信号。

对于 Qt 的关键字 signals、emit 及 slots 等，Qt 的预处理器 moc 会将此处的关键字 emit 转换为符合 C++ 语法标准的语句。在使用信号和槽机制的时候，要确保包含信号和槽的类必须是类 QObject 的派生类。而且在定义该类时应该在首部嵌入宏 Q_OBJECT。尤其需要注意的是，应该把这个类的声明放置在单独的头文件中（而不是潜入在源文件中），否则链接阶段会报错。

槽函数的定义则非常简单，如行③所示，只是在该函数的存取控制关键字后面加上 Qt 关键字 slots 即可。

在工程 main 函数中调用 QObject 的静态成员函数 connect，绑定上述信号和槽。当修改类 CExampleA 的对象 a 的值时，行①发送信号 stateChanged，与其绑定的对象 b 的槽函数 Function 会被执行，在控制台上输出对象 a 新的值。

qDebug 函数可用于在“应用程序输出”窗口输出变量信息，这个函数的用法在 3.5 节还有介绍。

**【例 3-5】** 使用控件内部定义好的信号和槽。

很多 GUI 窗口控件（例如按钮、标签、列表、编辑框等）都预先定义好了若干信号，例如单击按钮就会发出 clicked 信号，还有诸如双击（doubleClicked）、进入（entered）、按下（pressed）等信号都是预先在控件内部定义好的。同时控件中也有一些预先定义好的槽，例如 close、clear 等。

这个例子在对话框上添加标签和按钮两个控件，当单击按钮时标签控件消失。

第 1 步，建立无 UI 文件的工程。

利用 Qt Creator 的菜单的“文件→新建文件或项目”打开新建对话框，选择其中“应用程序”项目的 Qt Widgets Application 选项建立名为 3_5 的工程。在设置“类信息”的界面选择基类名为 QDialog，同时取消勾选“创建界面”项目。

第 2 步，在对话框中添加控件。

在 dialog.h 文件头部添加包含文件，加入如下代码：

```
#include <QLabel>
#include <QPushButton>
```

在 dialog.h 文件尾部添加如下声明：

```
private:
    QLabel * label;
    QPushButton * btn;
```

在 dialog.cpp 文件的构造函数添加如下代码，从而生成控件：

```
resize(300,300);
label=new QLabel("label",this);
btn=new QPushButton("Click Me",this);
label->move(150,150);
btn->move(125,110);
```

第 3 步，关联按钮的信号和标签的槽。

在 dialog.cpp 文件的构造函数添加如下代码：

```
connect(btn,SIGNAL(clicked()),label,SLOT(close()));
```

至此就完成了程序编写，保存、编译、运行之后单击按钮即可看到标签控件消失了。

## 3.5　如何发现程序的错误

程序中的错误有两种：编码语法错误、程序逻辑错误。

(1) 编码语法错误问题。如果有这类问题，编译系统根本无法完成程序的编译，因为有一些它不认识的语法。这时编译系统会给出具体的错误提示。在图 3-9 中，编程人员误将标签 QLabel 写成 QLable，这时系统会给出错误指示，如图 3-9 所示。在 QtCreator 下部的问题窗口可以双击错误条目，从而显示错误的具体信息。这种错误需要程序员分析代码的语法，找到写错的地方。

(2) 程序逻辑错误问题。如果有这类问题，有时候程序编译可以完成，但是运行结果不对。这时就要分析程序的逻辑，看看是算法问题还是代码实现有缺陷。

一种简单直接的查找错误的方法是：在适当的程序段落中加入一些输出语句，看看这些输出和预想的结果是否一致。可以在程序文件头部加入下列语句：

```
#include <QDebug>
```

而后在需要输出的位置用下面的语法输出：

```
qDebug()<<x<<"  "<<y;
```

这里 x、y 为输出的变量，中间加入一些空格使得结果看起来更清楚。

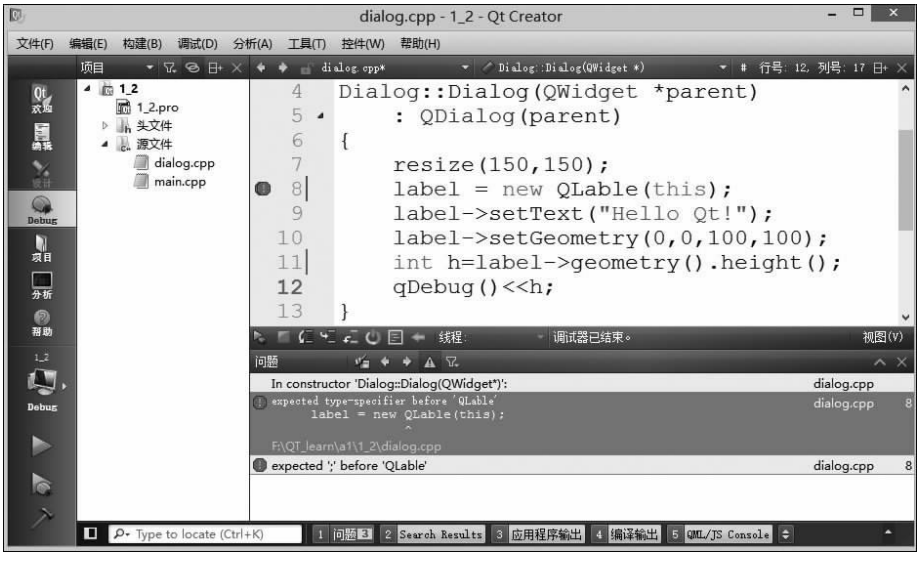



图 3-9 编译无法通过的情况

前面的例 3-4 就用了这种方法。

## 3.6 字符类和字符串类

Qt 对基本的字符类型和字符串进行了包装，将它们包装成两个功能全面的类——QChar 类和 QString 类。利用类中定义的方法，程序员可以快速实现目标。本节简单介绍这两个类。

### 3.6.1 字符类 QChar

QChar 类是 Qt 中用于表示一个字符的类，包含在 QtCore 共享库中。QChar 类用两个字节的 Unicode 编码来表示一个字符。

#### 1. 构造函数

QChar 类提供了多个不同原型的构造函数以方便使用，例如：

QChar()，构造一个空字符，即'\0'。

QChar(char ch)，由字符数据 ch 构造。

QChar(uchar ch)，由无符号字符数据 ch 构造。

QChar(ushort code)，由无符号短整形数据 code 构造，code 是 Unicode 编码。

QChar(short code)，由短整形数据 code 构造，code 是 Unicode 编码。

QChar(uint code)，由无符号整型数据 code 构造，code 是 Unicode 编码。

QChar(int code)，由整型数据 code 构造，code 是 Unicode 编码。

实际使用时很少直接构造 QChar 类的对象，而是把这些构造函数当做类型转换来

用，让编译器自动构造所需的 QChar 类对象。也就是说，在所有需要 QChar 类作为参数的地方都可以安全地使用各种整数类型。

**2. 判断字符某种特性的函数**

QChar 类提供了很多成员函数，可以对字符的类型进行判断，例如：

bool isDigit() const，判断是否十进制数字('0'～'9')。

bool isLetter() const，判断是否字母。

bool isNumber() const，判断是否数字，包括正负号、小数点等。

bool isLetterOrNumber()，判断是否字母或数字。

bool isLower() const，判断是否小写字母。

bool isUpper() const，判断是否大写字母。

bool isNull() const，判断是否空字符'\0'。

**3. 字符转换函数**

QChar 类提供了一些成员函数进行数据的转换，例如：

char toAscii() const，得到字符的 ASCII 码。

QChar toLower() const，转换成小写字母。

QChar toUpper() const，转换成大写字母。

ushort unicode() const，得到 Unicode 编码。

注意，这几个函数都不会改变对象自身，转换的结果通过返回值得到。

**4. 字符比较**

QChar 对象可以直接使用符号！＝、＜、＜＝、＝＝、＞、＞＝比较两个对象的大小，这一点和 C/C++ 语言完全一样。比较的结果等同于两个对象转换成 ASCII 码后的比较结果，例如有如下定义：

```
QChar c1('x'), c2('y');
char ch1='x', ch2='y';
```

则 c1＜c2 和 ch1＜ch2 的结果相同，都是 false。

## 3.6.2　字符串类 QString

因为本书很多程序的输出都涉及字符串，所以这里着重讲一下 QString 类，该类用于表示和操作字符串。字符串是很常用的一个数据结构，在很多编程语言中，例如 Java、Python 甚至脚本语言中，都把字符串类作为一种基本的类来实现。在 C++ 标准中的标准模板库(STL)里也定义了字符串 string 类，而 Qt 本身就是基于 C++ 的，在编程时当然可以使用 string 类型(要包含相应的头文件)，但是由于历史及其他方面的原因，在使用 Qt 编程时，Qt 库中的 QString 类更为常用。

QString 中每个字符以 16 位 Unicode 进行编码。平常用的 ASCII 等一些编码集都

是 Unicode 编码的子集。在使用 QString 的时候，不需要担心内存分配以及关于是否以'\0'结尾的问题(C 语言字符串，即字符数组以'\0'结尾)，QString 类自身会解决这些问题。与 C 语言风格的字符串不同，用 QString 表示的字符串中间可以包含'\0'符号，而 length 函数则会返回整个字符串的长度，而不是从开始到'\0'的长度。

**1. 字符串初始化**

QString 初始化常用方式如下：

```
QString str;                          //空串
QString str="Hello";                  //初始化为 Hello
QString str('A');                     //用一个字符初始化
QString str(str2);                    //用另一个字符 str2 初始化 str,复制到 str 中
```

**2. 字符串赋值**

利用＝可以将一个串赋值给另一个变量，例如：

```
QString str="Hello", str2;
str2=str;                             //等价于 str2="Hello"
```

**3. 利用[]取得或修改一个字符**

对于字符串 str，str[0]，str[1]，…就是字符串在下标位置 0，1，…位置的字符，可以获取或修改某个位置的数据，例如：

```
QString str="abcd";
str[0]='x';                           //变为"xbcd"
str[1]=str[0] ;                       //变为"xxcd"
```

**4. 字符串比较大小**

如果要比较两个字符串的大小，可以使用 ＞、＜、＞＝、＜＝、＝＝，例如：

```
if(str ==str2) { … }
```

或者

```
if(str >str2) { … }
```

判断字符串大小的方式和 C/C++ 语言一样，都是按照字典序比较(前面小，后面大)。

**5. 加法运算**

QString 重载了＋和＋＝运算符。这两个运算符可以把两个字符串连接到一起。QString 可以自动地对占用的内存空间进行扩充。下面是这两个操作符的使用：

```
QString str="Jiaotong ";
```

```
str +="University"+"\n";     //str 为 "Jiaotong University\n"
```

**6. 在头部、尾部添加字符串**

append 函数在字符串尾部添加另一个串，例如：

```
QString str="Jiaotong ";
str.append("University\n"); //str 为 "Jiaotong University\n"
```

prepend 函数则在字符串头部添加另一个串，例如：

```
QString str="Jiaotong University";
str. prepend ("Xi'an ");      //str 为 "Xi'an Jiaotong University"
```

**7 插入子串**

insert 函数用于在某个下标位置插入一个串，例如：

```
QString str="Meal";
str.insert(1, QString("ontr"));        //在下标 1 处插入
```

结果 str 为"Montreal"。

**8. 删除子串**

remove 函数用于删除字符串中的一部分，例如：

```
QString s="Montreal";
s.remove(1, 4);                //s 结果是"Meal"
```

这里删除从下标 1 开始的 4 个字符。

**9. 替换子串**

replace 函数可以将字符串的一部分替换为其他字符串，例如：

```
QString str="rock and roll";
//将第 5 号下标位置开始的 3 个字符去掉，换为"&"
str.replace(5, 3, "&");        //str 为"rock & roll"
```

或者

```
QString str="colour behaviour flavour neighbour";
str.replace(QString("ou"), QString("o"));    //将所有 ou 换成 o
```

结果 str 为"color behavior flavor neighbor"。

**10. 获取子串**

mid 函数可以截取子串，例如：