

第9章 QT图形界面应用程序开发基础



目 录

9.1 Qt简介

9.2 Qt5概述

9.3 信号和插槽机制

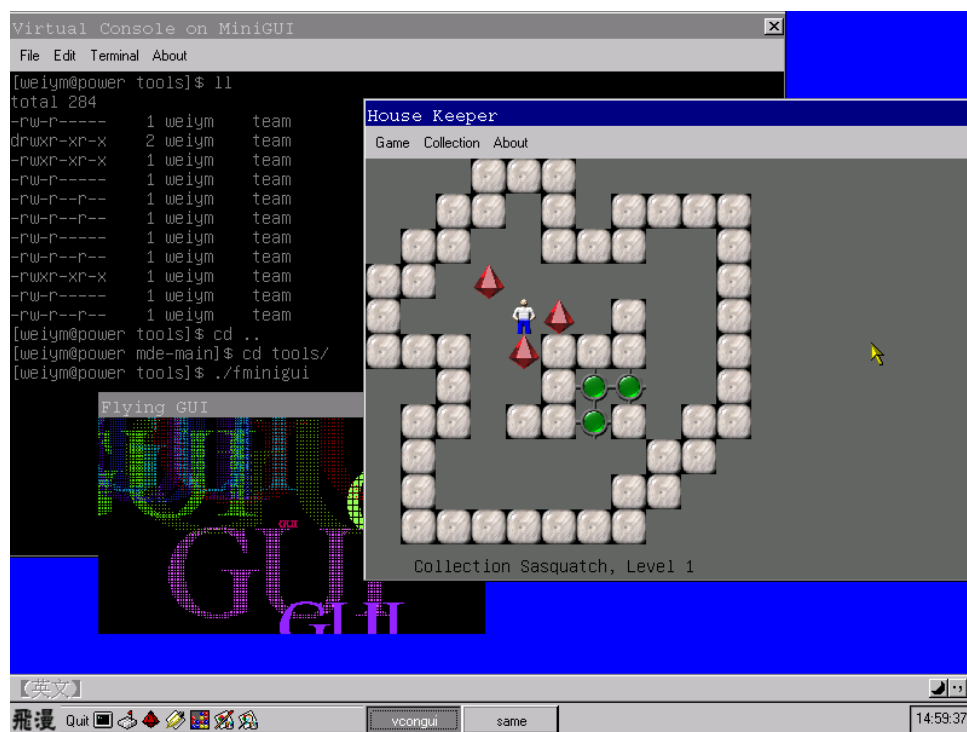
9.4 Qt程序设计

9.5 Qt数据库应用

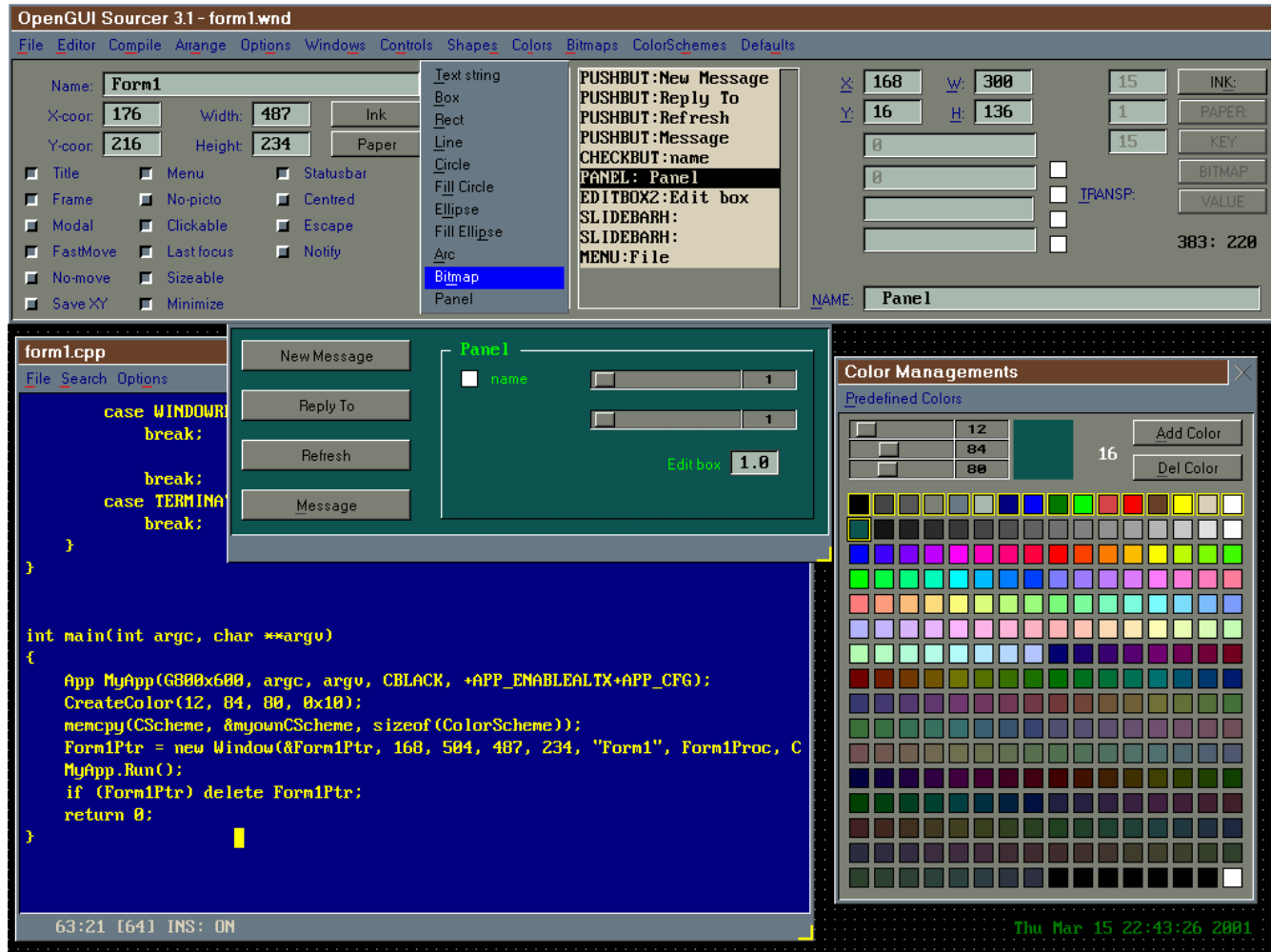
与普通**GUI**不同的是，嵌入式**GUI**的要求是轻量级的，如在嵌入式linux中使用的图形界面系统。同时嵌入式**GUI**还具有可定制，高可靠性，可裁减性等特点。嵌入式**GUI**的开发系统主要有X Window、MiniGUI、OpenGL、Qt、OpenGUI等。

MiniGUI是由北京飞漫软件技术有限公司主持的一个自由软件项目(遵循**GPL**条款)，其目标是为基于**Linux**的实时嵌入式系统提供一个轻量级的图形用户界面支持系统。

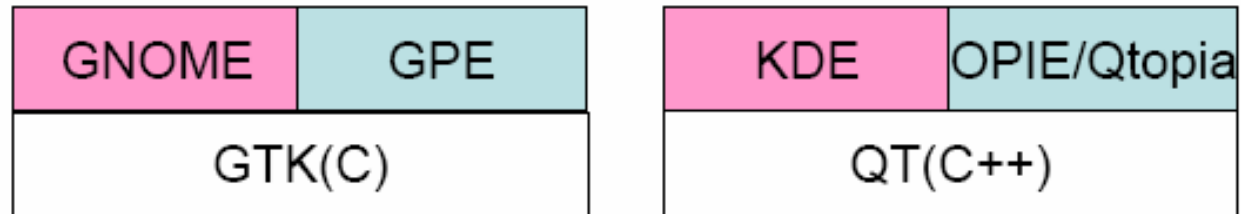
MiniGUI为应用程序定义了一组轻量级的窗口和图形设备接口。利用这些接口，每个应用程序可以建立多个窗口，而且可以在这些窗口中绘制图形。用户也可以利用**MiniGUI**建立菜单、按钮、列表框等常见的**GUI**元素。




OpenGUI基于一个用汇编实现的**x86**图形内核，
提供了一个高层的**C/C++**图形/窗口接口，它的资源
消耗小，可移植性差，不支持多进程。



主流的Linux窗口系统



 PC

 嵌入式

9.1

Part One

Qt简介

Qt是一个跨平台应用程序和图形用户界面**GUI**开发框架。

Qt是挪威**Trolltech**公司的标志性产品，于**1991**年推出。**2008**年，**Trolltech**被诺基亚公司收购，**QT**也因此成为诺基亚旗下的编程语言工具。**2012**年**8**月芬兰IT业务供应商**Digia**全面收购诺基亚**Qt**业务及其技术。

Qt概念

- ❑ Qt作为跨平台开发框架,是开源KDE桌面的基石.
- ❑ Mathematica, Autodesk Maya, WPS, Wireshark, KDE, Dropbox, Virtualbox, Google Earth, Skype, Opera, Adobe Photoshop Album, 咪咕音乐等著名软件都是基于Qt编写的.
- ❑ 和java的”一次编写到处运行”所不同的是,Qt是源代码级的跨平台一次编写到处编译.一次开发的Qt应用程序可以移植到不同平台.
- ❑ 目前Qt支持的平台有:Mac, Windows, unix, linux, 嵌入式linux

Qt/Embedded(简称QtE)是一个专门为嵌入式系统设计图形用户界面的工具包。**Qt**是挪威**Trolltech**软件公司的产品，它为各种系统提供图形用户界面的工具包，**QtE**就是**Qt**的嵌入式版本。



Qt/Embedded 也可以看成是一组用于访问嵌入式设备的 **Qt C++ API**;

Qt/Embedded ,Qt/X11,Qt/Windows 和**Qt/Mac**版本提供的都是相同的**API**和工具。

Qtopia是基于 **Qt**编写的一个用于手持设备的 用户信息管理软件，它集成了很多实用的程序。

市面上买到的预装 **linux** 操作系统的 **arm9**开发版，开机后看到的图形界面多是 **qtopia**。



Qt和Qtopia之间的关系

- ❑ Qt泛指Qt的所有桌面版本，比如Qt/X11，Qt Windows，Qt Mac等。由于Qt最早是在Linux中随着KDE流行开来的，因此通常很多人说的Qt都指用于Linux/Unix的Qt/X11。
- ❑ Qt/E（Qt/Embedded）是用于嵌入式Linux系统的Qt版本。Qt/E去掉了X Lib的依赖而直接工作于Frame Buffer上，因而效率更高，但它并不是Qt的子集，而应该是超集，部分机制（如QCOP等）不能用于Qt/X11中。
- ❑ Qtopia是一个构建于Qt/E之上的类似桌面系统的应用环境。相比之下，Qt/E是基础类库。
- ❑ Qtopia Core：就是原来的Qt/E，从Qt 4开始改名，把Qtopia Core并到Qtopia的产品线中去了。但实际上Qtopia Core就相当于原来的Qt/E，仍然作为基础类库。

Qt 具有下列优点:

(1)面向对象

(2)丰富的 **API**函数和直观的**C++** 类库

(3)支持 **2D/3D** 图形渲染, 支持 **OpenGL**

(4)具有跨平台 **IDE** 的集成开发工具

(5)跨桌面和嵌入式操作系统的移植性

(6)大量的开发文档

(7)国际化

按不同的版本发行：

Qt商业版：提供给商业软件开发。它们提供传统商业软件发行版并且提供在协议有效期内的免费升级和技术支持服务。

Qt开源版：仅仅为了开发自由和开放源码软件，提供了和商业版本同样的功能。**GNU**通用公共许可证下，它是免费的。

2009年3月发布的Qt 4.5 起，诺基亚为Qt增添开源**LGPL**授权选择。

9.2

Part Two

Qt5概述

9.2.1 Qt 5简介

2012年12月19日，Digia宣布正式发行Qt 5.0。Qt 5.0是一个全新的流行于跨平台应用程序和用户界面开发框架的版本，可应用于桌面、嵌入式和移动应用程序

Qt 5的主要优势包括:

- 图形质量;
- 出色的图像处理与表现能力:
- 更高效和灵活的研发:
- 跨平台的移植变得更加简单:
- Qt 通过使用 **OpenGL ES**, 大大的增加了交付令人印象深刻的图形的能力

Qt Creator的下载和安装

以Qt 5.6.1版本为例，其中包含了Qt Creator 4.0.1。

- 地址：
http://download.qt.io/official_releases/qt/5.6/5.6.1-1/
- 下载文件：qt-opensource-windows-x86-mingw492-5.6.1-1.exe

版本介绍

Qt安装包:

qt-opensource-windows-x86-mingw492-5.6.1-1.exe

- **opensource**表示开源版本
- **windows-x86**表示Windows 32位平台
- **mingw**表示使用MinGW编译器
- **5.6.1-1**是当前版本号

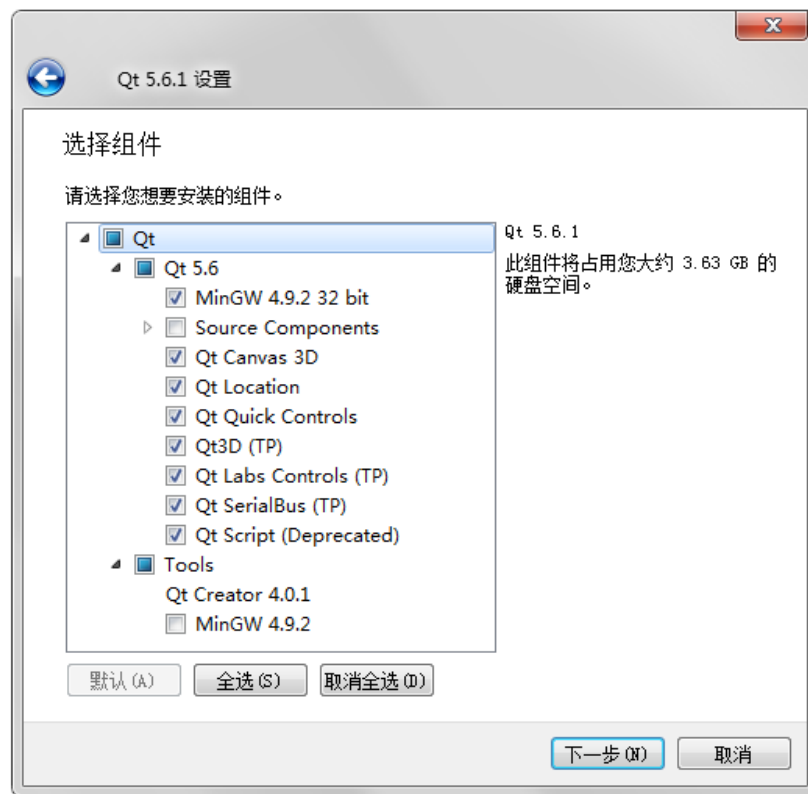


提示

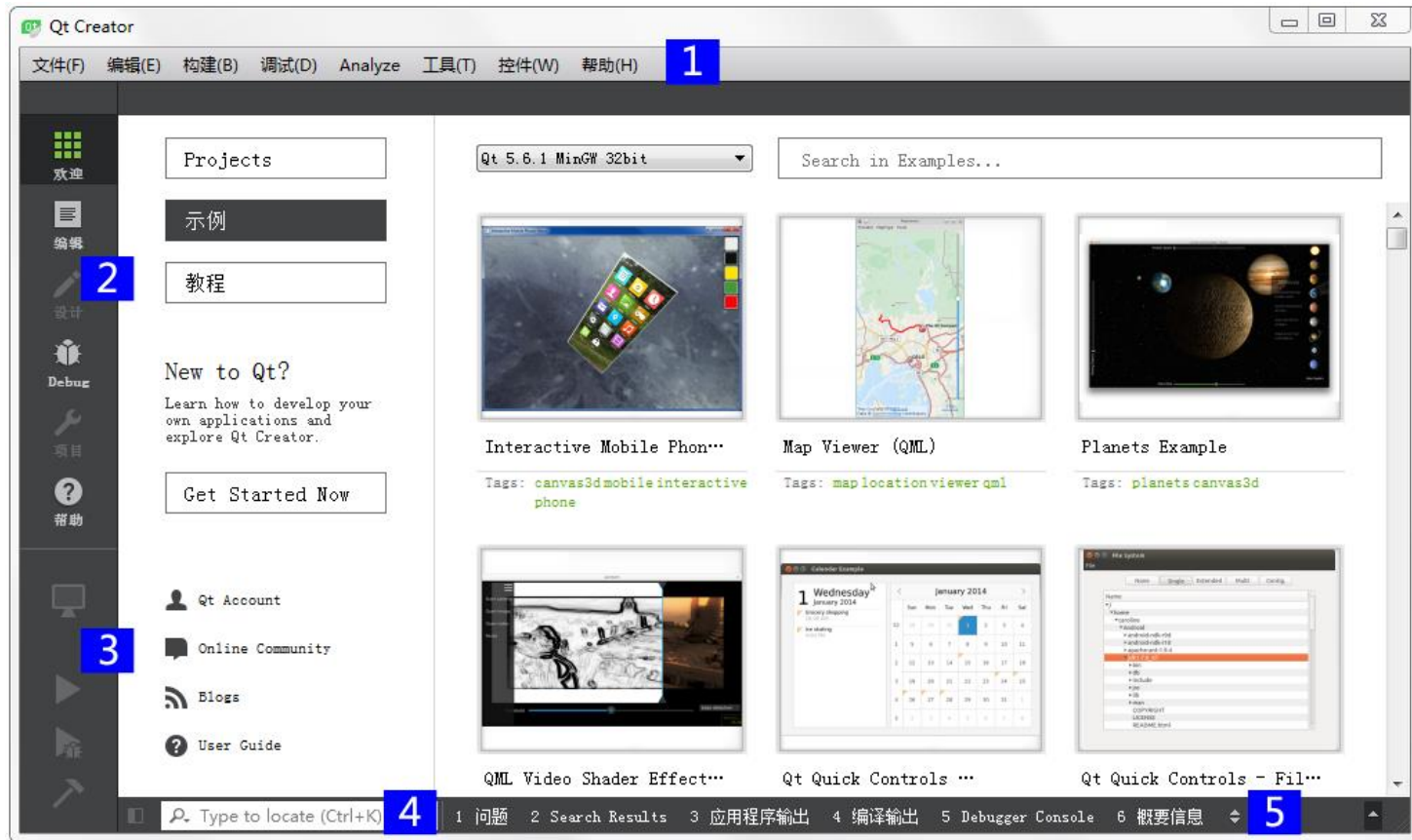
MinGW即Minimalist GNU For Windows，是将GNU开发工具移植到Win32平台下的产物，是一套Windows上的GNU工具集。用其开发的程序不需要额外的第三方DLL支持就可以直接在Windows下运行。更多内容请查看<http://www.mingw.org>。

注意：

- 安装路径中不能有中文。
- 安装开始是否登陆或者注册Qt账号，不影响程序的安装，可以直接跳过。
- 在选择组件界面，可以选择安装一些模块，单击一个组件，可以在右侧显示该组件的详细介绍，初学者建议保持默认选择。



Qt Creator环境介绍



Qt Creator主要由主窗口区、菜单栏、模式选择器、构建套件选择器、定位器和输出窗格等部分组成

①**菜单栏（Menu Bar）**。这里有8个菜单选项，包含了常用的功能菜单。

- **文件菜单**。其中包含了新建、打开和关闭项目和文件、打印文件和退出等基本功能菜单。
- **编辑菜单**。这里有撤销、剪切、复制、查找和选择编码等常用功能菜单，在高级菜单中还有标示空白符、折叠代码、改变字体大小和使用vim风格编辑等功能菜单。
- **构建菜单**。包含构建和运行项目等相关的菜单。
- **调试菜单**。包含调试程序等相关的功能菜单。
- **Analyze分析菜单**。包含QML分析器、Valgrind内存和功能分析器等相关菜单。
- **工具菜单**。这里提供了快速定位菜单、版本控制工具菜单和外部工具菜单等。这里的选项菜单中包含了Qt Creator各个方面的设置选项：环境设置、文本编辑器设置、帮助设置、构建和运行设置、调试器设置和版本控制设置等。
- **控制菜单**。这里包含了设置窗口布局的一些菜单，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、报告bug和插件管理等菜单。

②**模式选择器 (Mode Selector)**。Qt Creator包含欢迎、编辑、设计、调试、项目和帮助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，它们对应的快捷键依次是Ctrl + 数字1~6。

- 欢迎模式。这里主要提供了一些功能的快捷入口，如打开帮助教程、打开示例程序、打开项目、新建项目、快速打开以前的项目和会话、联网查看Qt官方论坛和博客等。

- 编辑模式。这里主要用来查看和编辑程序代码，管理项目文件。也可以在“工具→选项”菜单项中对编辑器进行设置。

- 设计模式。这里整合了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。可以在“工具→选项”菜单项中对设计师进行设置。

- 调试模式。支持设置断点、单步调试和远程调试等功能，包含局部变量和监视器、断点、线程以及快照等查看窗口。可以在“工具→选项”菜单项中设置调试器的相关选项。

- 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、代码风格设置和依赖关系等页面。也可以在“工具→选项”菜单项中对项目进行设置。

- 帮助模式。在帮助模式中将Qt助手整合了进来，包含目录、索引、查找和书签等几个导航模式。可以在“工具→选项”菜单中对帮助进行相关设置。

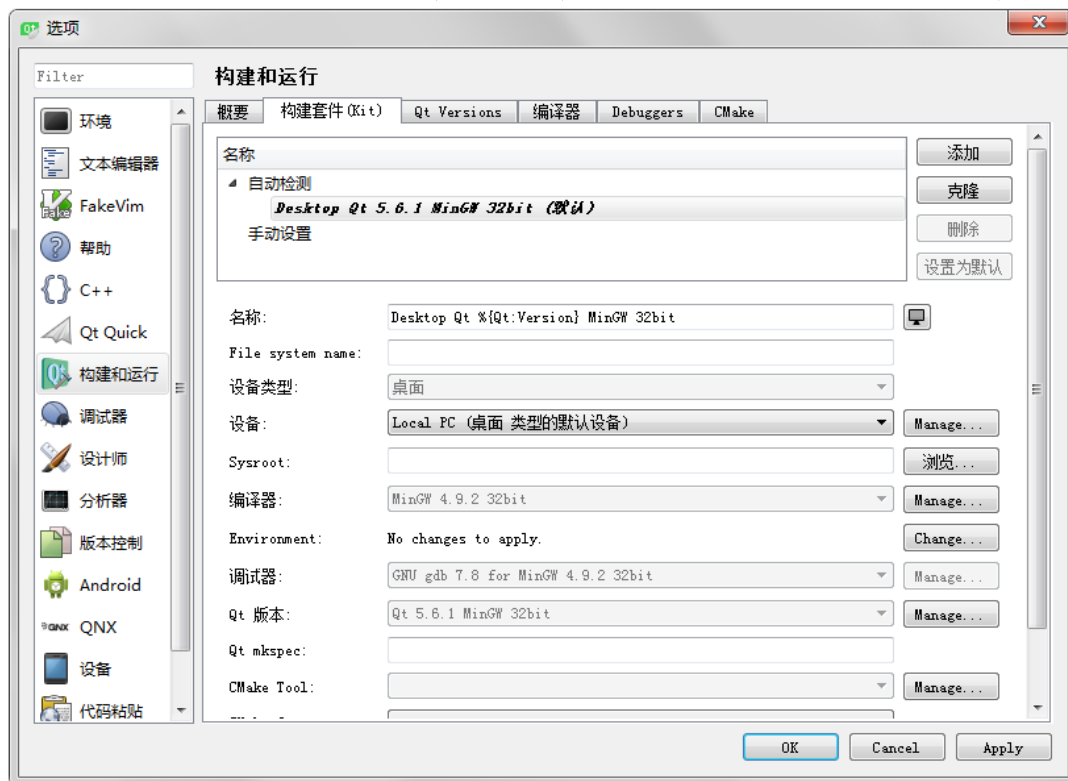
③**构建套件选择器 (Kit Selector)**。包含了目标选择器 (Target selector)、运行按钮 (Run)、调试按钮 (Debug) 和构建按钮 (Building) 4个图标。目标选择器用来选择要构建哪个项目，使用哪个Qt库，这对于多个Qt库的项目很有用。这里还可以选择编译项目的debug版本或是release版本。运行按钮可以实现项目的构建和运行；调试按钮可以进入调试模式，开始调试程序；构建按钮完成项目的构建。

④**定位器 (Locator)**。在Qt Creator中可以使用定位器来快速定位项目、文件、类、方法、帮助文档以及文件系统。可以使用过滤器来更加准确地定位要查找的结果。可以在“工具→选项”菜单项中设置定位器的相关选项。

⑤**输出窗格 (Output panes)**。这里包含了问题、搜索结果、应用程序输出、编译输出、Debugger Console、概要信息、版本控制7个选项，它们分别对应一个输出窗口，相应的快捷键依次是Alt + 数字1~7。问题窗口显示程序编译时的错误和警告信息；搜索结果窗口显示执行了搜索操作后的结果信息；应用程序输出窗口显示在应用程序运行过程中输出的所有信息；编译输出窗口显示程序编译过程输出的相关信息；版本控制窗口显示版本控制的相关输出信息。

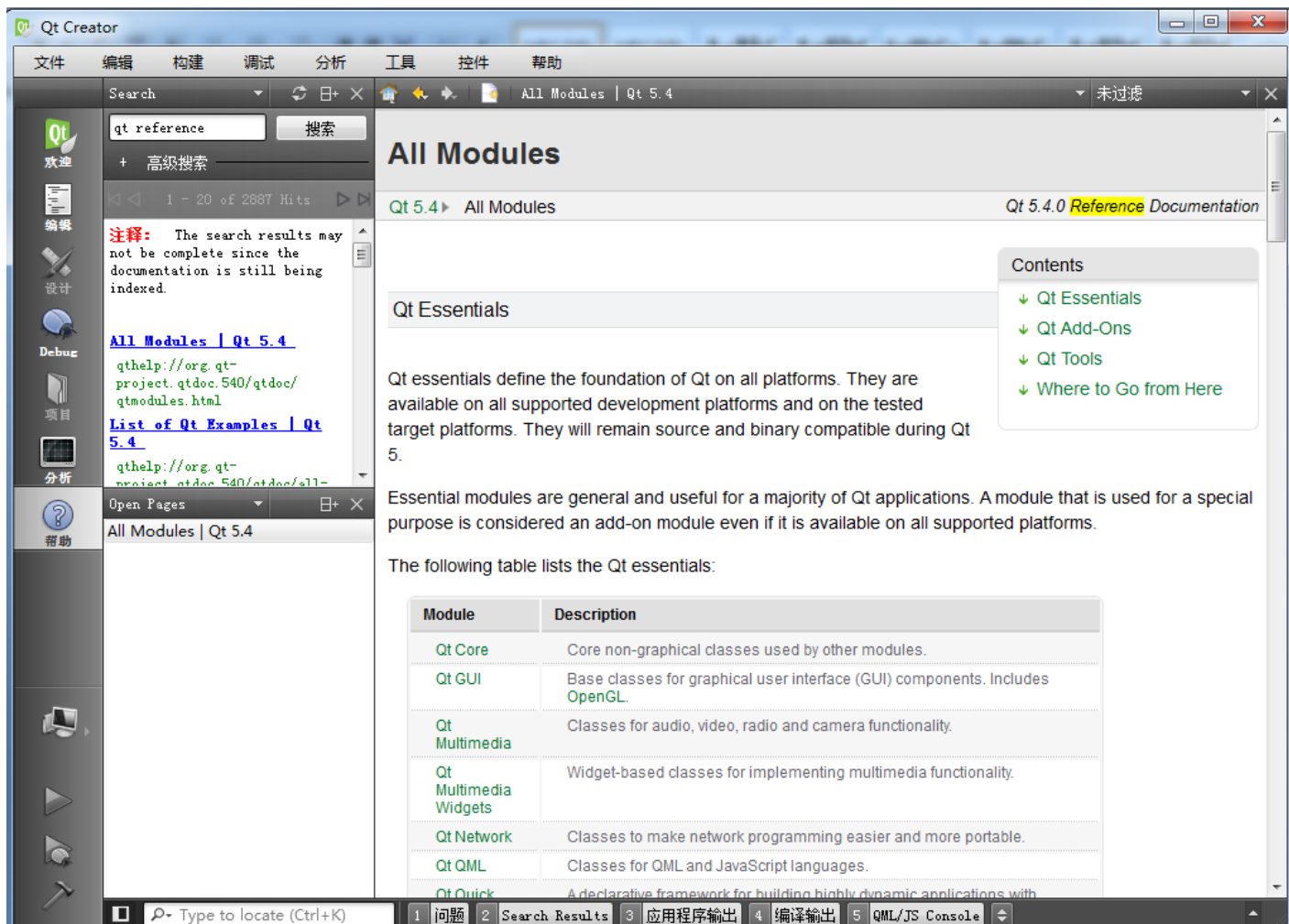
将Qt Creator与Qt库进行关联

选择“工具→选项”菜单项，然后选择“构建和运行”项。可以看到构建套件中已经自动检测到了Qt版本、编译器和调试器。如果以后需要添加其他版本的Qt，可以在这里先添加编译器，然后添加Qt版本，最后添加构建套件。



9.2.2 通过”帮助”了解Qt 5的组成——模块

打开Qt Creator，进入帮助模式，然后选择“Qt Reference”进行搜索。选择“All Qt Modules”选项来查看所有的Qt模块。如下图所示。



在“**All Qt Modules**”页面Qt的模块被分为了三部分：

Qt 基本模块（Qt Essentials）、

Qt扩展模块（Qt Add-Ons）、

Qt工具（Qt Tools）

模块	描述
Qt Core	使用其它模块的核心非图形类
Qt GUI	图形用户界面（GUI）组件的基础类，包括OpenGL
Qt Multimedia	处理音频、视频、广播、摄像头功能的类
Qt Network	使网络编程更容易，更方便的类
Qt QML	QML和JavaScript的类
Qt Quick	自定义用户界面构建高度动态的应用程序的声明性框架
Qt SQL	使用SQL集成数据库的类
Qt Test	进行Qt应用程序和库单元测试的类
Qt WebKit	基于WebKit2实现的一个新的QML API类
Qt WebKit Widgets	Qt4中，WebKit1和QWidget-based类
Qt Widgets	用C++部件扩展Qt图形界面的类

表9-1 Qt基本模块组成

9.4

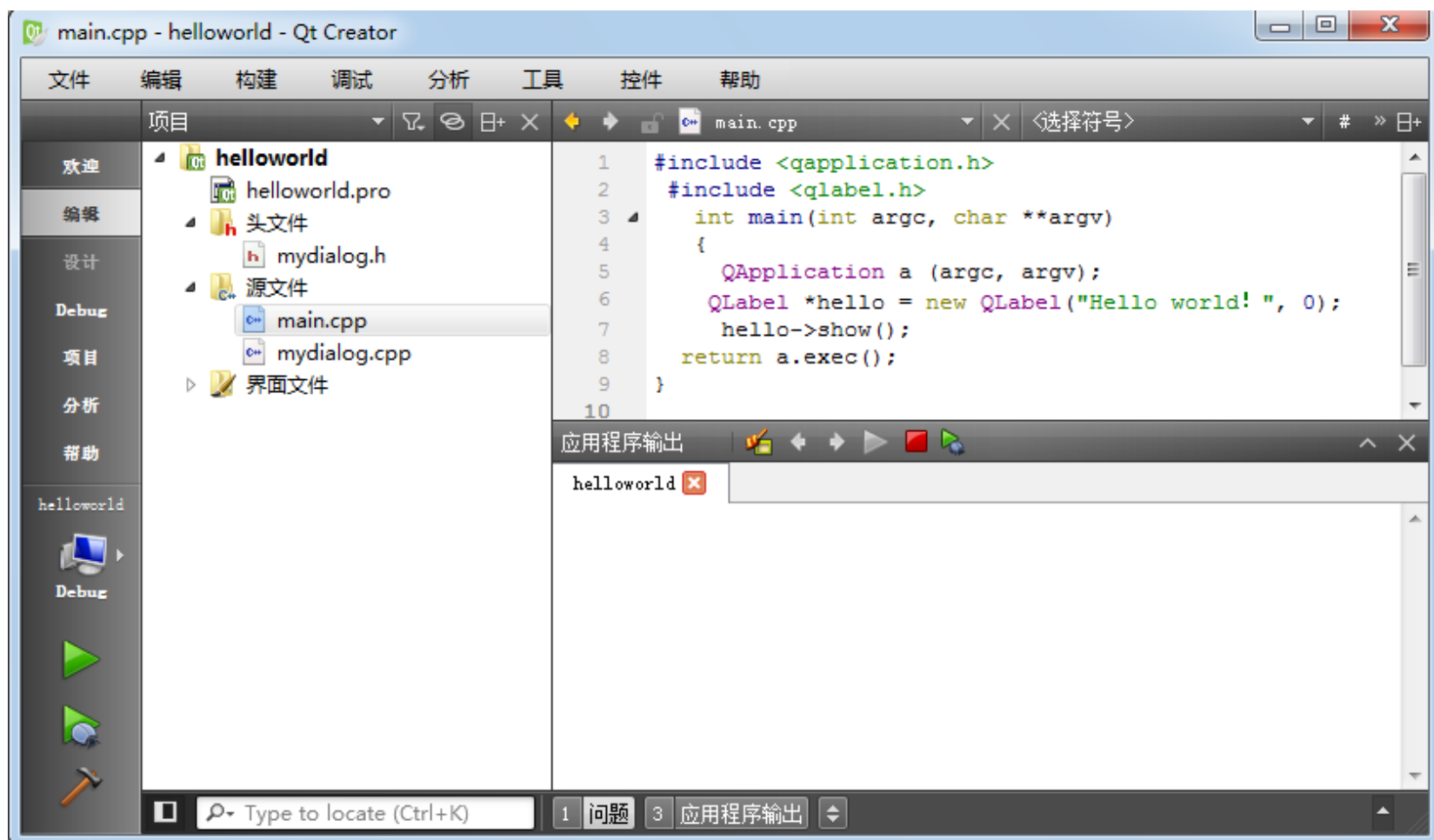
Part Four

Qt程序设计

9.4.1 Helloworld程序

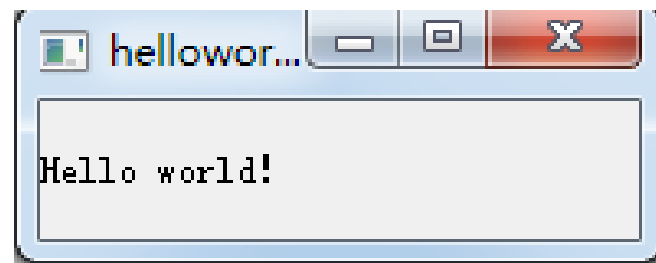
新建一个helloworld项目，该项目使用的类信息中将基类选择为QDialog。项目构成如下图所示。

图9-4 helloworld项目构成



在源文件 **Main.cpp**中输入源码，如下：

```
#include <qapplication.h>
#include <qlabel.h>
int main(int argc, char **argv)
{
    QApplication a (argc, argv);//创建了一个QApplication类的对象a
    QLabel *hello = new QLabel("Hello world! ", 0);//创建了一个静态文本，将
                                                    label设置为“Hello world!”
    hello->show();//调用show（）方法使窗口部件可见
    return a.exec();//exec（）中qt接收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
}
```



9.4.2 多窗口应用程序（演示）

9.3 Part Three

信号和插槽机制

信号和插槽

■ 在Qt程序中，利用信号（**signal**）和插槽（**slot**）机制进行对象间的通信

- 事件处理的方式也是回调
- 当对象状态发生改变的时候，发出**signal**通知所有的**slot**接收**signal**，尽管它并不知道哪些函数定义了**slot**，而**slot**也同样不知道要接收怎样的**signal**

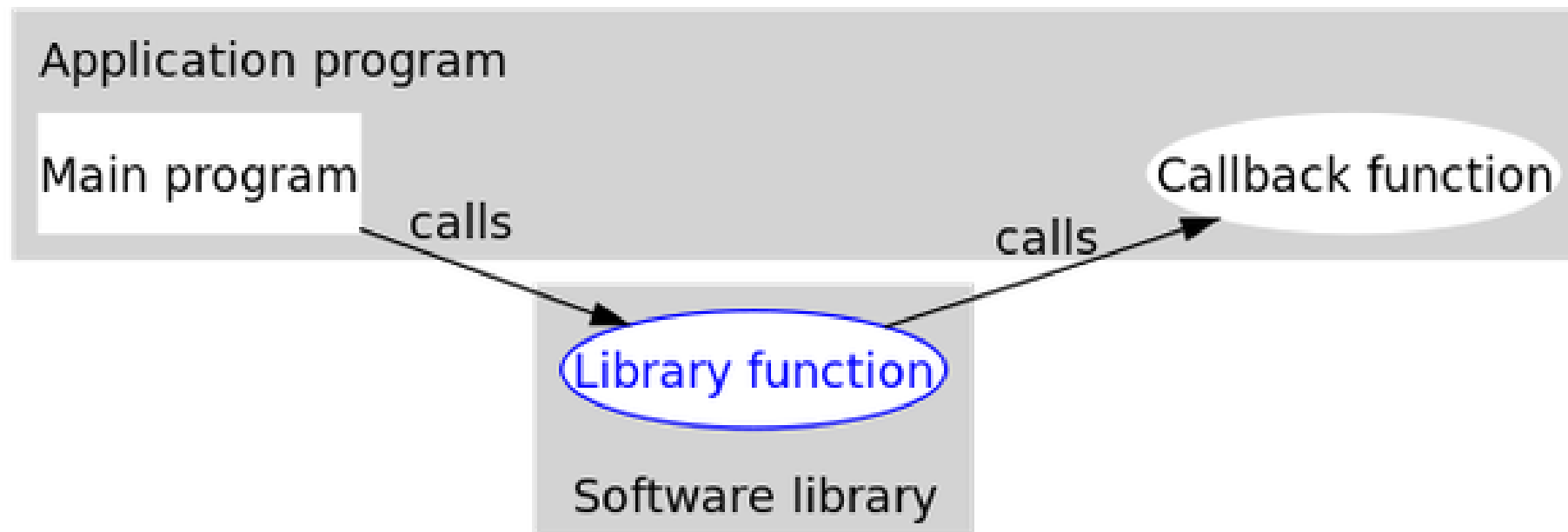
■ **signal**和**slot**机制真正实现了封装的概念，**slot**除了接收**signal**之外和其它的成员函数没有什么不同，而且**signal**和**slot**之间也不是一一对应的。

回调callback

- 程序跑起来时，一般情况下，应用程序（**application program**）会时常通过**API**调用库里所预先备好的函数。但是有些库函数（**library function**）却要求应用先传给它一个函数，好在合适的时候调用，以完成目标任务。这个被传入的、后又被调用的函数就称为回调函数（**callback function**）。

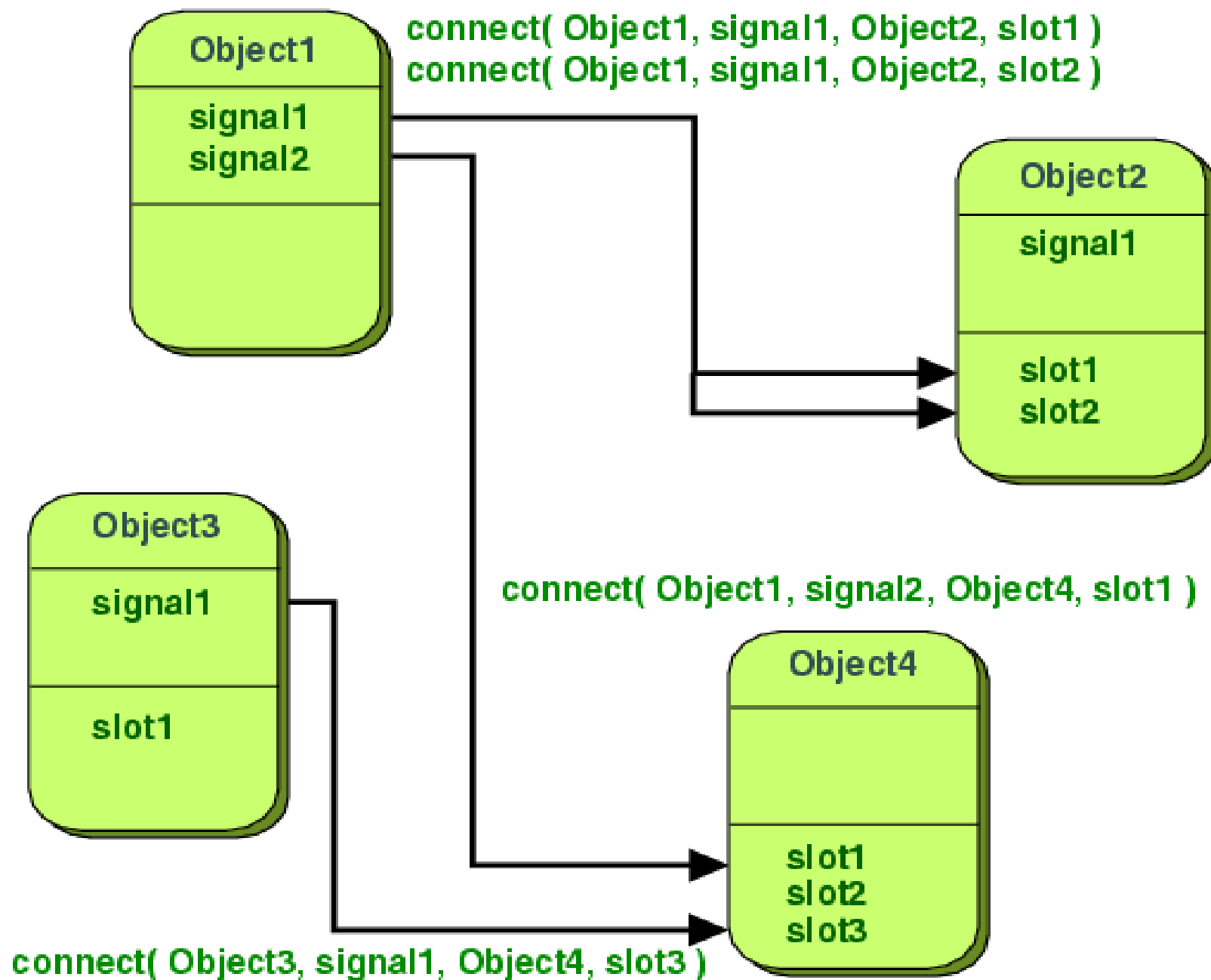
回调callback

- 打个比方，有一家旅馆提供叫醒服务，但是要求旅客自己决定叫醒的方法。可以是打客房电话，也可以是派服务员去敲门，睡得死怕耽误事的，还可以要求往自己头上浇盆水。这里，“叫醒”这个行为是旅馆提供的，相当于库函数，但是叫醒的方式是由旅客决定并告诉旅馆的，也就是回调函数。而旅客告诉旅馆怎么叫醒自己的动作，也就是把回调函数传入库函数的动作，称为**登记回调函数**（**to register a callback function**）。如下图所示



- 信号和插槽用于两个对象之间的通信，信号和插槽（**signal/slot**）机制是**Qt**的核心特征，信号和插槽是**Qt**自定义的一种通信机制，它独立于标准的**C/C++**语言，所有从**QObject**或其子类派生的类都能够包含信号和插槽。
- 在**GUI**编程中，当改变了一个部件时，总希望其他部件也能了解到该变化。更一般来说，我们希望任何对象都可以和其他对象进行通信。例如，如果用户点击了关闭按钮，我们希望可以执行窗口的**close()**函数来关闭窗口。当一个特殊的事情发生时便可以发射一个信号，比如按钮被单击；而槽就是一个函数，它在信号发射后被调用，来响应这个信号。在**Qt**的部件类中已经定义了一些信号和槽，但是更多的做法是子类化这个部件，然后添加自己的信号和槽来实现想要的功能。

图9-3 信号与插槽的对应关系



信号

声明一个信号，例如：

signals:

```
void dlgReturn(int);           // 自定义的信号
```

- 声明一个信号要使用**signals**关键字。
- 在**signals**前面不能使用**public**、**private**和**protected**等限定符，因为只有定义该信号的类及其子类才可以发射该信号。
- 信号只用声明，不需要也不能对它进行定义实现。
- 信号没有返回值，只能是**void**类型的。
- 只有**QObject**类及其子类派生的类才能使用信号和槽机制，使用信号和槽，还必须在类声明的最开始处添加**Q_OBJECT**宏。

发射信号

例如：

```
void MyDialog::on_pushButton_clicked() // 确定按钮
{
    int value = ui->spinBox->value(); // 获取输入的数值
    emit dlgReturn(value);           // 发射信号
    close();                          // 关闭对话框
}
```

当单击确定按钮时，便获取**spinBox**部件中的数值，然后使用自定义的信号将其作为参数发射出去。发射一个信号要使用**emit**关键字，例如程序中发射了**dlgReturn()**信号。

槽

自定义槽的声明：

private slots:

```
void showValue(int value);
```

实现：

```
void MyWidget::showValue(int value)    // 自定义槽
{
    ui->label->setText(tr("获取的值是： %1").arg(value));
}
```

声明一个槽需要使用**slots**关键字。一个槽可以是**private**、**public**或者**protected**类型的，槽也可以被声明为虚函数，这与普通的成员函数是一样的，也可以像调用一个普通函数一样来调用槽。槽的最大特点就是可以和信号关联。

信号和槽的关联

例如：

```
MyDialog *dlg = new MyDialog(this);  
connect(dlg, &MyDialog::dlgReturn(int),this, &MyWidget::(showValue(int)));
```

connect()函数原型如下：

```
bool QObject::connect ( const QObject * sender, const char * signal, const QObject * receiver,  
    const char * method)
```

- 它的第一个参数为发送信号的对象，例如这里的**dlg**；
- 第二个参数是要发送的信号，这里可用**SIGNAL(dlgReturn(int))**；
- 第三个参数是接收信号的对象，这里是**this**，表明是本部件，即**Widget**，当这个参数为**this**时，也可以将这个参数省略掉，因为**connect()**函数还有另外一个重载形式，该参数默认为**this**；
- 第四个参数是要执行的槽，这里可用**SLOT(showValue(int))**。
- 对于信号和槽，QT4使用**SIGNAL()**和**SLOT()**宏，它们可以将其参数转化为**const char*** 类型。**connect()**函数的返回值为**bool**类型，当关联成功时返回**true**。

下面再举例来说明信号/插槽机制。

```
#include <QObject>
class Counter : public QObject
{
    Q_OBJECT
public:
    Counter() { m_value = 0; }
    int value() const { return m_value; }
public slots:
    void setValue(int value);
signals:
    void valueChanged(int newValue);
private:
    int m_value;
};
```

Counter类通过发射信号**valueChanged**来通知其他对象它的状态发生了变化，同时该类还具有一个插槽**setValue**，其他对象可以发信号给这个插槽。插槽**setValue**的定义如下：

```
void Counter::setValue(int value)
{
    if (value != m_value) {
        m_value = value;
        emit valueChanged(value);
    }
}
```

在声明信号/插槽后，使用**connect**（）函数将它们关联起来。

```
Counter a, b;
```

```
QObject::connect(&a, &Counter::valueChanged,  
                &b, &Counter::setValue);
```

```
a.setValue(18);    // a.value() == 18, b.value() == 18
```

```
b.setValue(42);    // a.value() == 18, b.value() == 42
```

当信号与插槽没有必要继续保持关联时，用户可以使用**disconnect()**函数来断开连接。其定义如下所示：

```
bool QObject::disconnect (const QObject *  
sender, const char * signal,const Object *  
receiver, const char * slot) [static]
```

这个函数断开发射者中的信号与接收者中的插槽函数之间的关联。

示例：关联方式一：使用connect()关联

- mywidget.h文件写上槽的声明：

public slots:

```
void showChildDialog();
```

- 在mywidget.cpp文件中将槽的实现：

```
void MyWidget::showChildDialog()
```

```
{
```

```
    QDialog *dialog = new QDialog(this);
```

```
    dialog->show();
```

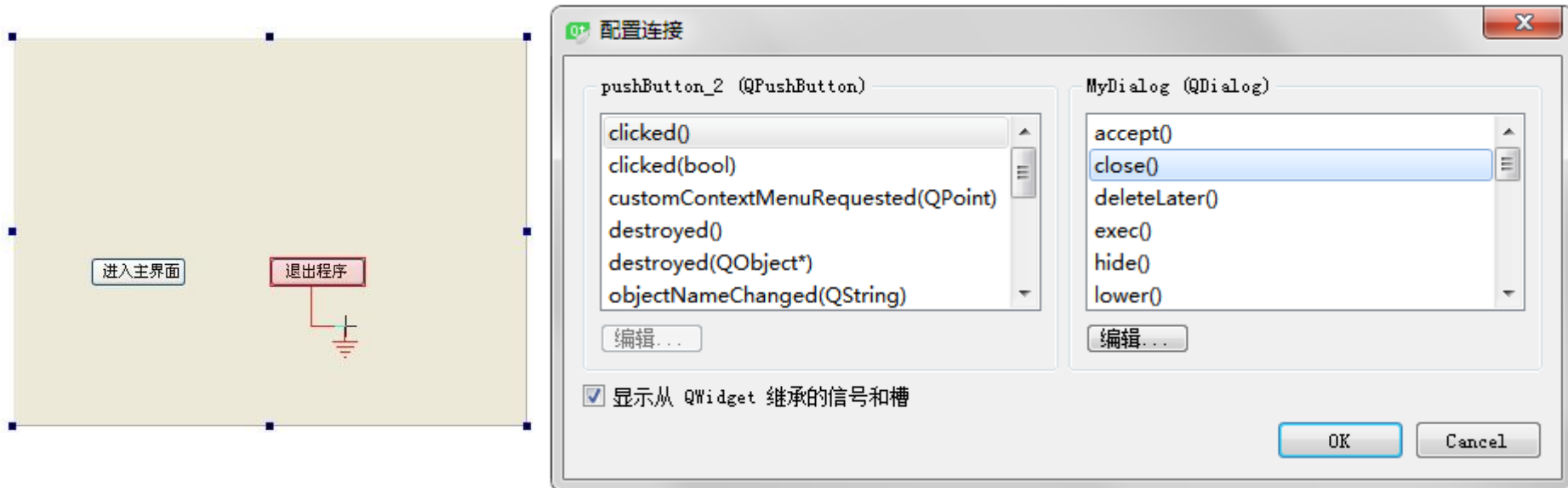
```
}
```

- 在mywidget.cpp文件的MyWidget类的构造函数中使用connect()关联按钮单击信号和自定义的槽如下：

```
connect(ui->showChildButton, &QPushButton::clicked,  
        this, &MyWidget::showChildDialog);
```

关联方式二：在设计模式关联

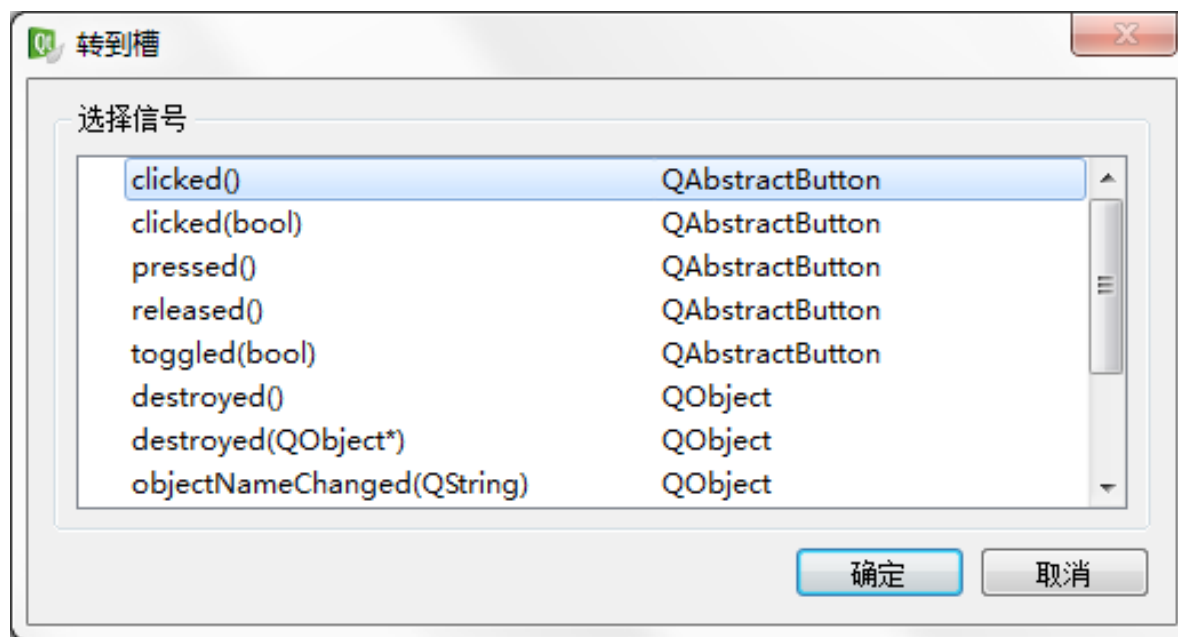
- 首先添加自定义对话框类MyDialog。在设计模式中向窗口上添加两个Push Button，并且分别更改其显示文本为“进入主界面”和“退出程序”。
- 点击设计器上方的“编辑信号/槽”图标，或者按下快捷键F4，这时便进入了部件的信号和槽的编辑模式。在“退出程序”按钮上按住鼠标左键，然后拖动到窗口界面上，这时松开鼠标左键。
- 在弹出的配置连接对话框中，选中下面的“显示从QWidget继承的信号和槽”选项，然后在左边的QPushButton栏中选择信号clicked()，在右边的QDialog栏中选择对应的槽close()，完成后按下“确定”。



关联方式三：自动关联

在“进入主界面”按钮上右击，在弹出的菜单上选择“转到槽”，然后在弹出的对话框中选择clicked()信号，并按“确定”。这时便会进入代码编辑模式，并且定位到自动生成的on_pushButton_clicked()槽中。在其中添加代码：

```
void MyDialog::on_pushButton_clicked()
{
    accept();
}
```



- 自动关联就是将关联函数整合到槽命名中。
- 例如**on_pushButton_clicked()**就是由字符“on”和发射信号的部件对象名，还有信号名组成。这样就可以去掉那个**connect()**关联函数了。每当**pushButton**被按下，就会发射**clicked()**信号，然后就会执行**on_pushButton_clicked()**槽。
- 这里**accept()**函数是**QDialog**类中的一个槽，对于一个使用**exec()**函数实现的模态对话框，执行了这个槽，就会隐藏这个模态对话框，并返回**QDialog::Accepted**值，我们就是要使用这个值来判断是哪个按钮被按下了。与其对应的还有一个**reject()**槽，它可以返回一个**QDialog::Rejected**值。其实，前面的“退出程序”按钮也可以关联这个槽。

QObject类

- **QObject**是Qt类体系的唯一基类，是Qt各种功能的源头活水，就象MFC中的CObject和Dephi中的TObject
- **对象树**：**QObject**在对象树中组织它们自己。当你以另外一个对象作为父对象来创建一个**QObject**时，它就被添加到父对象的**children()**列表中，并且当父对象被删除的时候，它也会被删除。这种机制很好的适合了图形用户界面应用对象的需要。
- **事件**：事件是由窗口系统或qt本身对各种事务的反应而产生的。当用户按下、释放一个键或鼠标按钮，一个键盘或鼠标事件被产生；当窗口第一次显示，一个绘图事件产生，从而告知最新的可见窗口需要重绘自身。大多数事件是由于响应用户的动作而产生的，但还有一些，比如定时器等，是由系统独立产生的。

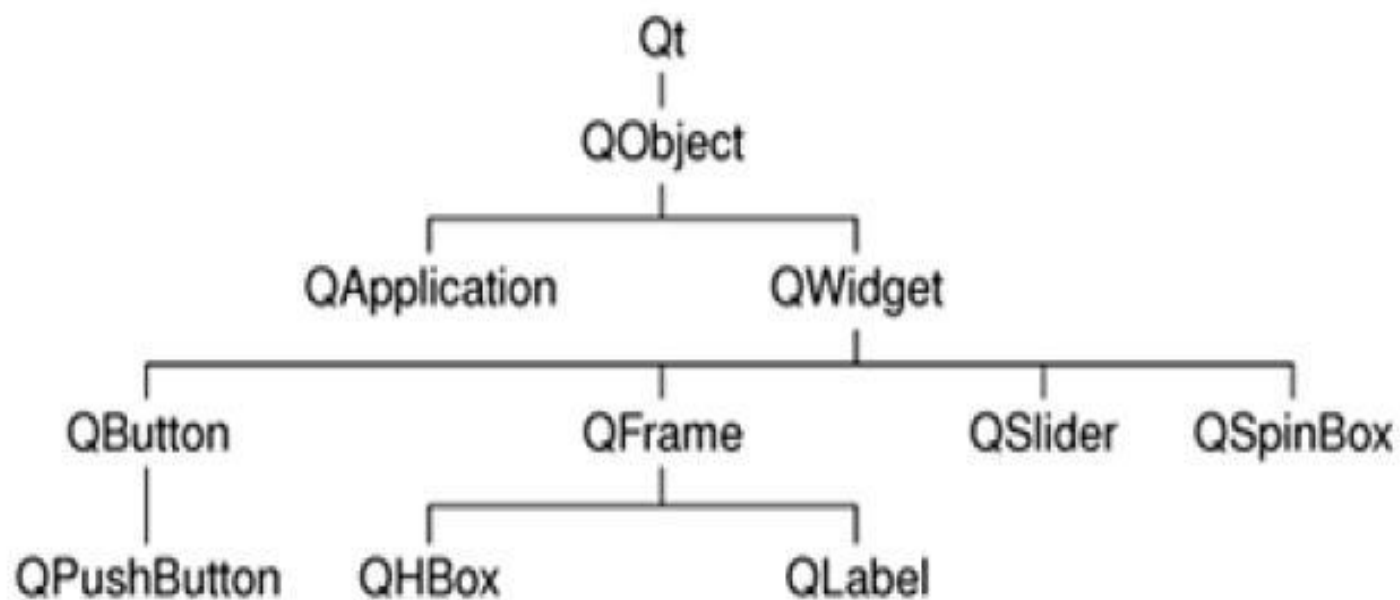
QApplication

- **QApplication**和**QWidget**都是**QObject**类的子类
- **QApplication**类负责**GUI**应用程序的控制流和主要的设置，它包括主事件循环体，负责处理和调度所有来自窗口系统和其他资源的事件，并且处理应用程序的开始、结束以及会话管理，还包括系统和应用程序方面的设置。对于一个应用程序来说，建立此类的对象是必不可少的

QWidget

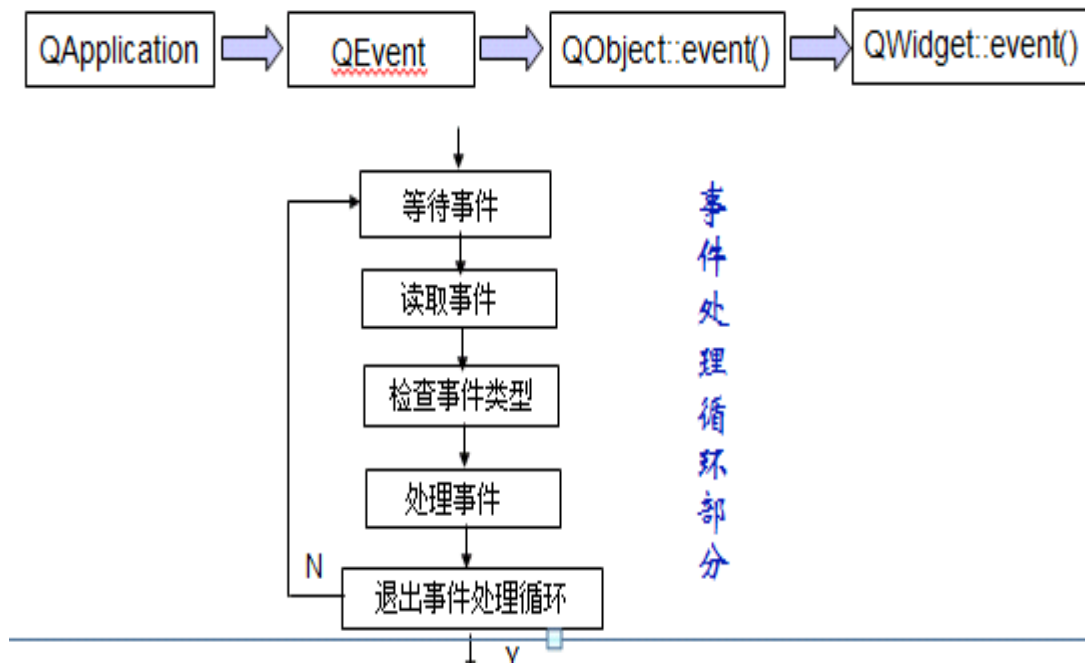
- **QWidget**类是所有用户接口对象的基类，它继承了**QObject**类的属性。组件是用户界面的单元组成部分，它接收鼠标、键盘和其它从窗口系统来的事件，并把它自己绘制在盘屏幕上
- **QWidget**类有很多成员函数，但一般不直接使用，而是通过子类继承来使用其函数功能。如，**QPushButton**、**QlistBox**等都是它的子类

对象树



事件运行机制

Qt事件的处理过程：
QApplication的事件循环体从事件队列中拾取本地窗口系统事件或其他事件，译成QEvent()，并送给QObject::event()，最后送给QWidget::event()本别对事件处理



9.5

Part Four

Qt数据库应用

Qt中的QtSql模块提供了对数据库的支持，该模块中的众多类基本上可以分为三层，分别是驱动层、**SQL**接口层和用户接口层。

驱动层为具体的数据库和**SQL**接口层之间提供了底层的桥梁；

SQL接口层提供了对数据库的访问，其中的 **QSqlDatabase**类用来创建连接， **QSqlQuery**类可以使用 **SQL**语句来实现与数据库交互，其他类对该层提供了支持；

用户接口层的几个类实现了将数据库中的数据链接到窗口部件上，这些类使用模型/视图（**model/view**）框架实现，它们是更高层次的抽象，即便不熟悉**SQL**也可以操作数据库。对应数据库部分的内容，可以在帮助中查看 **SQL Programming**关键字。

9.5.1 数据库驱动

数据库驱动名称	对应DBMS
QDB2	IBM DB2(版本7.1及以上)
QIBASE	Borland InterBase
QMYSQL	MySQL
QOCI	Oracle调用接口驱动
QODBC	ODBC
QPSQL	PostgreSQL（版本7.3及以上）
SQLITE2	SQLITE2
SQLITE	SQLITE3
QTDS	Sybase自适应服务器

也可以通过代码来查看本机Qt支持的数据库，一种典型代码如下所示：

(1)新建Qt console应用，名称为sqldriverscheck。

(2)完成后将sqldriverscheck.pro项目文件中第一行代码更改为：

QT += core sql

表明使用了sql模块。

(3)将main.cpp文件的内容更改如下：

```
#include <QCoreApplication>
```

```
#include <QSqlDatabase>
```

```
#include <QDebug>
```

```
#include <QStringList>
```

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

```
{
```

```
    QCoreApplication app(argc, argv);
```

```
    qDebug() << "Available drivers:";
```

```
    QStringList drivers = QSqlDatabase::drivers();
```

```
    foreach(QString driver, drivers)
```

```
        qDebug() << driver;
```

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

```
}
```

```
Available drivers:
```

```
"SQLITE"
```

```
"MYSQL"
```

```
"MYSQL3"
```

```
"ODBC"
```

```
"ODBC3"
```

```
"PSQL"
```

```
"PSQL7"
```

这里先使用**drivers()**函数获取了现在可用的数据库驱动，然后分别进行了输出。

9.5.2 Qt与SQLite数据库的连接

```
QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE");
db.setDatabaseName(":memory:");
if (!db.open()) {
    QMessageBox::critical(0, QApplication->tr("Cannot open database"),
        QApplication->tr("Unable to establish a database connection."
            ), QMessageBox::Cancel);
    return false;
}
QSqlQuery query;
query.exec("create table student (id int primary key, "
    "name varchar(30))");

query.exec("insert into student values(0, 'first')");
query.exec("insert into student values(1, 'second')");
```

数据库连接和表建立完毕后就可以进行查询等数据库操作了。如：

```
query.exec("SELECT *FROM student");
```

`exec()`方法执行后就可以对查询的结果集进行设置了。结果集就是查询到的所有记录的集合。在`QSqlQuery`类中提供了多个函数来操作这个集合，需要注意这个集合中的记录是从0开始编号的。最常用的操作有：

`seek(int n)`： `query`指向结果集的第`n`条记录；

`first()`： `query`指向结果集的第一条记录；

`last()`： `query`指向结果集的最后一记录；

`next()`： `query`指向下一条记录，每执行一次该函数，便指向相邻的下一条记录；

previous() : **query**指向上一条记录，每执行一次该函数，便指向相邻的上一条记录；

record() : 获得现在指向的记录；

value(int n) : 获得属性的值。其中**n**表示你查询的第**n**个属性，比方上面我们使用“**select * from student**”就相当于“**select id, name from student**”，那么**value(0)**返回**id**属性的值，**value(1)**返回**name**属性的值。该函数返回**QVariant**类型的数据，关于该类型与其他类型的对应关系，可以在帮助中查看**QVariant**。

at() : 获得现在**query**指向的记录在结果集中的编号。

9.5.3 SQL模型

除了 **QSqlQuery**，Qt还提供了3个更高层的类来访问数据库，分别是 **QSqlQueryModel**、**QSqlTableModel**和**QSqlRelationalTableModel**。

类名	用途
QSqlQueryMdoel	基于任意SQL语句的只读模型
QSqlTableModel	基于单个表的读写模型
QSqlReltionalTableModel	QSqlTableModel 的 子类，增加了外键支持

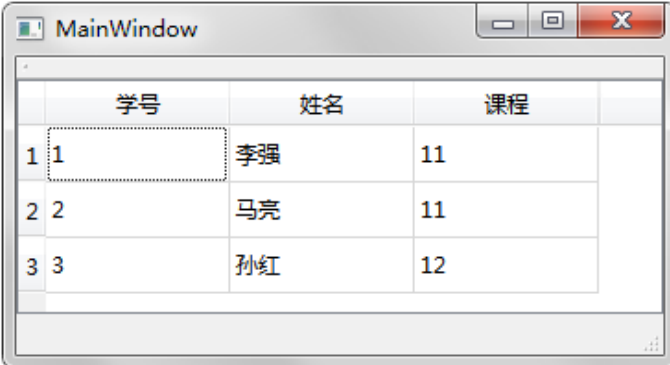
使用SQL模型类

- 这3个类都是从**QAbstractTableModel**派生来的，可以很容易地实现将数据库中的数据在**QListView**和**QTableView**等项视图类中进行显示。使用这些类的另一个好处是，这样可以使编写的代码很容易的适应其他的数据源。例如，如果开始使用了**QSqlTableModel**，而后来要改为使用**XML**文件来存储数据，这样需要做的仅是更换一个数据模型。

SQL查询模型 QSqlQueryModel

QSqlQueryModel提供了一个基于SQL查询的只读模型。例如：

```
QSqlQueryModel *model = new QSqlQueryModel(this);  
model->setQuery("select * from student");  
model->setHeaderData(0, Qt::Horizontal, tr("学号"));  
model->setHeaderData(1, Qt::Horizontal, tr("姓名"));  
model->setHeaderData(2, Qt::Horizontal, tr("课程"));  
QTableView *view = new QTableView(this);  
view->setModel(model);  
setCentralWidget(view);
```

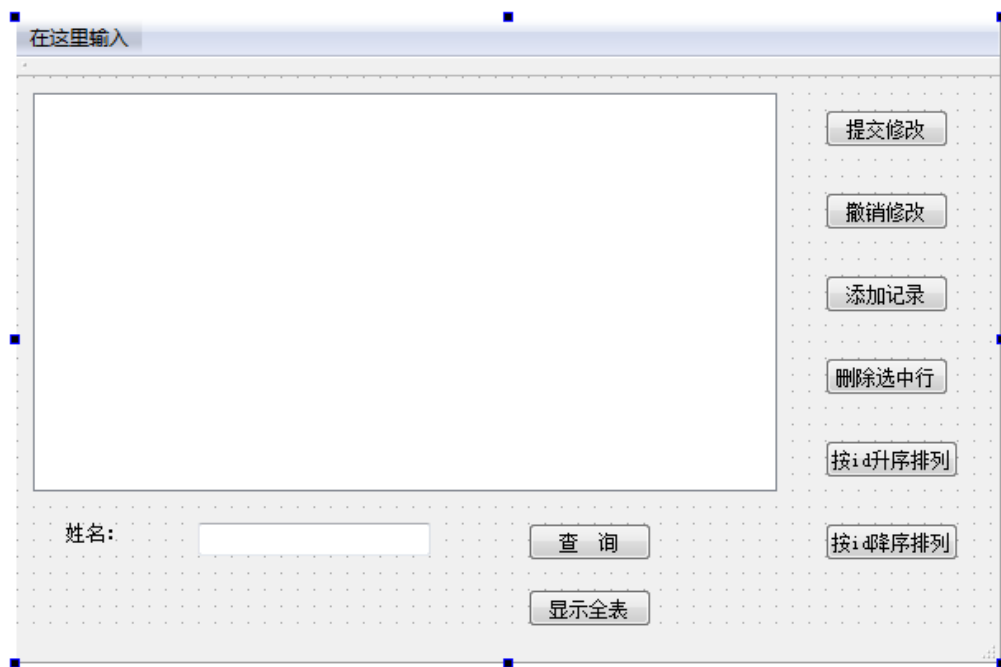


	学号	姓名	课程
1	1	李强	11
2	2	马亮	11
3	3	孙红	12

这里先创建了QSqlQueryModel对象，然后使用setQuery()来执行SQL语句查询整张student表，并使用setHeaderData()来设置显示的标头。后面创建了视图，并将QSqlQueryModel对象作为其要显示的模型。这里要注意，其实QSqlQueryModel中存储的是执行完setQuery()函数后的结果集，所以视图中显示的是结果集的内容。QSqlQueryModel中还提供了columnCount()返回一条记录中字段的个数；rowCount()返回结果集中记录的条数；record()返回第n条记录；index()返回指定记录的指定字段的索引；clear()可以清空模型中的结果集。

SQL表格模型 QSqlTableModel

- **QSqlTableModel**提供了一个一次只能操作一个**SQL**表的读写模型，它是**QSqlQuery**的更高层次的替代品，可以浏览和修改独立的**SQL**表，并且只需编写很少的代码，而且不需要了解**SQL**语法。例如：



- 创建数据表

QSqlQuery query;

// 创建student表

query.exec("create table student (id int primary key, "
"name varchar, course int)");

query.exec("insert into student values(1, '李强', 11)");

query.exec("insert into student values(2, '马亮', 11)");

query.exec("insert into student values(3, '孙红', 12)");

// 创建course表

query.exec("create table course (id int primary key, "
"name varchar, teacher varchar)");

query.exec("insert into course values(10, '数学', '王老师')");

query.exec("insert into course values(11, '英语', '张老师')");

query.exec("insert into course values(12, '计算机', '白老师')");

- 显示表:

```
model = new QSqlTableModel(this);
model->setTable("student");
model->select();
// 设置编辑策略
model->setEditStrategy(QSqlTableModel::OnManualSubmit);
ui->tableView->setModel(model);
```

这里创建一个QSqlTableModel后，只需使用**setTable()**来为其指定数据库表，然后使用**select()**函数进行查询，调用这两个函数就等价于执行了“**select * from student**”语句。这里还可以使用**setFilter()**来指定查询时的条件。在使用该模型以前，一般还要设置其编辑策略，它由**QSqlTableModel::EditStrategy**枚举类型定义。

常量	描述
QSqlTableModel::OnFieldChange	所有对模型的改变都会立即应用到数据库
QSqlTableModel::OnRowChange	对一条记录的改变会在用户选择另一条记录时被应用
QSqlTableModel::OnManualSubmit	所有的改变都会在模型中进行缓存，直到调用 <code>submitAll()</code> 或者 <code>revertAll()</code> 函数

- 修改操作

```
// 提交修改按钮
void MainWindow::on_pushButton_clicked()
{ // 开始事务操作
    model->database().transaction();
    if (model->submitAll()) {
        if(model->database().commit()) // 提交
            QMessageBox::information(this, tr("tableModel"),
                                     tr("数据修改成功! "));
    } else {
        model->database().rollback(); // 回滚
        QMessageBox::warning(this, tr("tableModel"),
                             tr("数据库错误: %1").arg(model->lastError().text()),
                             QMessageBox::Ok);
    }
}

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


- 筛选操作

// 查询按钮，进行筛选

```
void MainWindow::on_pushButton_5_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("name = '%1']").arg(name));
```

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

```
}
```

// 显示全表按钮

```
void MainWindow::on_pushButton_6_clicked()
```

```
{
```

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

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

```
}
```

- 排序操作

// 按id升序排列按钮

```
void MainWindow::on_pushButton_7_clicked()  
{
```

```
    //id字段，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_8_clicked()  
{
```

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

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

```
}
```

- 删除记录

// 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();
    }
}
```

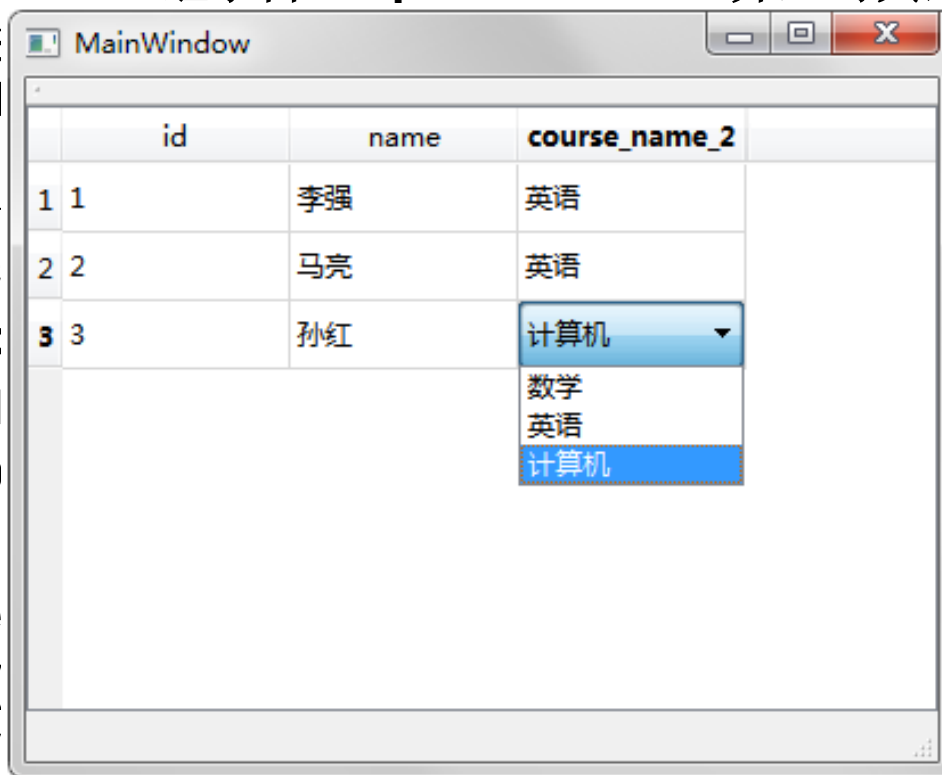
- 添加记录

// 添加记录按钮

```
void MainWindow::on_pushButton_3_clicked()  
{  
    // 获得表的行数  
    int rowNum = model->rowCount();  
    int id = 10;  
  
    // 添加一行  
    model->insertRow(rowNum);  
    model->setData(model->index(rowNum,0), id);  
  
    // 可以直接提交  
    //model->submitAll();  
}
```

SQL关系表格模型 QSqlRelationalTableModel

- **QSqlRelationalTableModel**继承自**QSqlTableModel**，并且对其进行了扩展，提供了对外键的支持。在表中的主键字段之间的一对一关系模型，就是所谓的模型，就可以将它显示为course表中的id字段course字段的值是一



	id	name	course_name_2	
1	1	李强	英语	
2	2	马亮	英语	
3	3	孙红	计算机	

```
QSqlRelationalTableModel->setTable("stud  
model->setRelation(2,  
model->select();  
QTableView *view = ne  
view->setModel(model  
setCentralWidget(view
```

- Qt中还提供了一个**QSqlRelationalDelegate**委托类，它可以为**QSqlRelationalTableModel**显示和编辑数据。这个委托为一个外键提供了一个**QComboBox**部件来显示所有可选的数据，这样就显得更加人性化了。
`view->setItemDelegate(new QSqlRelationalDelegate(view));`

del(this);