第 3 章

CHAPTER 3

Qt 信号槽机制及图片轮播

3.1 Qt 信号槽机制及应用

5min

信号槽(Signal & Slot)是 Qt 编程的基础,也是 Qt 的一大创新。因为有了信号槽的编程机制,在 Qt 中处理界面的各个组件在进行交互操作时变得更加直观和简单。信号槽是 Qt 框架引以为豪的机制之一。所谓信号槽,实际就是观察者模式。当某个事件发生后,例如按钮检测到自己被单击了一下,它就会发出一个信号(Signal)。这种发出是没有目的的,类似广播。如果有对象对这个信号感兴趣,则可以使用连接(Connect)函数将想要处理的信号和自己的一个槽函数(Slot)绑定,以此来处理这个信号。也就是说,当信号发出时,被连接的槽函数会自动被回调。信号槽机制是 Qt GUI 编程的基础,使用信号槽机制可以比较容易地将信号与响应代码关联起来。

1. 信号

信号就是在特定情况下被发射的事件,例如下压式按钮(PushButton)最常见的信号就是鼠标单击时发射的 clicked()信号,而一个组合下拉列表(ComboBox)最常见的信号是选择的列表项在变化时发射的 CurrentIndexChanged()信号。GUI 程序设计的主要内容就是对界面上各组件的信号进行响应,只需知道什么情况下发射哪些信号,合理地去响应和处理这些信号就可以了。信号是一个特殊的成员函数声明,返回值的类型为 void,只能声明不能定义实现。信号必须用 signals 关键字声明,访问属性为 protected,只能通过 emit 关键字调用(发射信号)。当某个信号对其客户或所有者发生的内部状态发生改变时,信号被一个对象发射。只有定义过这个信号的类及其派生类能够发射这个信号。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。信号槽机制完全独立于任何 GUI 事件循环。只有当所有的槽返回以后发射函数(emit)才返回。如果存在多个槽与某个信号相关联,则当这个信号被发射时,这些槽将会一个接一个地执行,但执行的顺序将会是随机的,不能人为地指定哪个先执行哪个后执行。信号的声明是在头文件中进行的,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,代码如下:

```
signals:
    void mycustomsignals();
```

signals 是 QT 的关键字,而非 C/C++的。信号可以重载,但信号却没有函数体定义,并且信号的返回类型都是 void,不要指望能从信号返回什么有用信息。信号由 MOC 自动产生,不应该在.cpp 文件中实现。

2. 槽

槽就是对信号响应的函数。槽就是一个函数,与一般的 C++函数一样,可以定义在类的任何部分(public、private 或 protected),可以具有任何参数,也可以被直接调用。槽函数与一般函数的不同点在于:槽函数可以与一个信号关联,当信号被发射时,关联的槽函数被自动执行。槽也能够声明为虚函数。槽的声明也是在头文件中进行的,代码如下:

```
public slots:
    void setValue(int value);
```

只有 QObject 的子类才能自定义槽,定义槽的类必须在类声明的最开始处使用 Q_OBJECT,类中声明槽需要使用 slots 关键字,槽与所处理的信号在函数签名上必须一致。

3. 信号槽的关联

信号槽的关联是用 QObject::connect()函数实现的,其代码如下:

```
//chapter3/qt-help-apis.txt
//QObject::connect(sender, SIGNAL(signal()), receiver, SLOT(slot()));
bool QObject::connect (const QObject * sender, const char * signal,
                       const QObject * receiver, const char * method,
                       Qt::ConnectionType type = Qt::AutoConnection );
```

connect()函数是 QObject 类的一个静态函数,而 QObject 是所有 Qt 类的基类,在实际调用时可以忽略前面的限定符,所以可以直接写为

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

其中,sender 是发射信号的对象的名称,signal()是信号名称。信号可以看作特殊的函数,需要带圆括号,当有参数时还需要指明参数。receiver 是接收信号的对象名称,slot()是槽函数的名称,需要带圆括号,当有参数时还需要指明参数。SIGNAL 和 SLOT 是 Qt 的宏,用于指明信号和槽,并将它们的参数转换为相应的字符串。一段简单的代码如下:

```
QObject::connect(btnClose, SIGNAL(clicked()), Widget, SLOT(close()));
```

这行代码的作用就是将 btnClose 按钮的 clicked()信号与窗体(Widget)的槽函数 close()相关联,当单击 btnClose 按钮(界面上的 Close 按钮)时,就会执行 Widget 的 close()槽函数。

当信号槽没有必要继续保持关联时，可以使用 disconnect 函数来断开连接，代码如下：

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

disconnect()函数用于断开发射者中的信号与接收者中的槽函数之间的关联。在 disconnect()函数中 0 可以用作一个通配符，分别表示任何信号、任何接收对象、接收对象中的任何槽函数，但是发射者 sender 不能为 0，其他 3 个参数的值可以等于 0。以下 3 种情况需要使用 disconnect()函数断开信号槽的关联。

(1) 断开与某个对象相关联的任何对象，代码如下：

```
disconnect(sender, 0, 0, 0);
sender->disconnect();
```

(2) 断开与某个特定信号的任何关联，代码如下：

```
disconnect(sender, SIGNAL(mySignal()), 0, 0);
sender->disconnect(SIGNAL(mySignal()));
```

(3) 断开两个对象之间的关联，代码如下：

```
disconnect(sender, 0, receiver, 0);
sender->disconnect(receiver);
```

4. 信号槽的注意事项

Qt 利用信号槽机制取代传统的回调函数机制(Callback)进行对象之间的沟通。当操作事件发生时，对象会发射一个信号，而槽则是一个函数，用于接收特定信号并且运行槽本身设置的动作。信号与槽之间，需要通过 QObject 的静态方法 connect()函数连接。信号在任何运行点皆可发射，甚至可以在槽里再发射另一个信号，信号槽的连接不限定为一对一的连接，一个信号可以连接到多个槽或多个信号连接到同一个槽，甚至信号也可连接到信号。以往的回调缺乏类型安全，在调用处理函数时，无法确定是传递正确形态的参数，但信号和其接收的槽之间传递的数据形态必须相匹配，否则编译器会发出警告。信号和槽可接收任何数量、任何形态的参数，所以信号槽机制是完全类型安全。信号槽机制也确保了低耦合性，发送信号的类并不知道哪个槽会接收，也就是说一个信号可以调用所有可用的槽。此机制会确保当“连接”信号和槽时，槽会接收信号的参数并且正确运行。关于信号槽的使用，需要注意以下规则。

(1) 一个信号可以连接多个槽，代码如下：

```
connect(spinNum, SIGNAL(valueChanged(int)), this, SLOT(addFun(int));
connect(spinNum, SIGNAL(valueChanged(int)), this, SLOT(updateStatus(int));
```

当一个对象 spinNum 的数值发生变化时，所在窗体有两个槽函数进行响应，一个

addFun()函数用于计算，另一个 updateStatus()函数用于更新状态。当一个信号与多个槽函数关联时，槽函数按照建立连接时的顺序依次执行。当信号和槽函数带有参数时，在connect()函数里，要写明参数的类型，但可以不写参数名称。

(2) 多个信号可以连接同一个槽，例如让 3 个选择颜色的 RadioButton 的 clicked()信号关联到相同的一个自定义槽函数 setTextFontColor()，代码如下：

```
//chapter3/qt - help - apis.txt
connect(ui - > rBtnBlue,SIGNAL(clicked()),this,SLOT(setTextFontColor()));
connect(ui - > rBtnRed,SIGNAL(clicked()),this,SLOT(setTextFontColor()));
connect(ui - > rBtnBlack,SIGNAL(clicked()),this,SLOT(setTextFontColor()));
```

当任何一个 RadioButton 被单击时都会执行 setTextFontColor() 槽函数。

(3) 一个信号可以连接另外一个信号，代码如下：

```
connect(spinNum, SIGNAL(valueChanged(int)), this, SIGNAL
(refreshInfo(int));
```

当一个信号发射时，也会发射另外一个信号，实现某些特殊的功能。

(4) 在严格的情况下，信号槽的参数个数和类型需要一致，至少信号的参数不能少于槽的参数。如果不匹配，则会出现编译错误或运行错误。

(5) 在使用信号槽的类中，必须在类的定义中加入宏 Q_OBJECT。

(6) 当一个信号被发射时，与其关联的槽函数通常会被立即执行，就像正常调用一个函数一样。只有当信号关联的所有槽函数执行完毕后，才会执行发射信号处后面的代码。

5. 元对象工具

元对象编译器对 C++文件中的类声明进行分析并产生用于初始化元对象的 C++代码，元对象包含全部信号和槽的名字及指向槽函数的指针。

元对象编译器读 C++源文件，如果发现有 Q_OBJECT 宏声明的类，就会生成另外一个C++源文件，新生成的文件中包含该类的元对象代码。假设有一个头文件 mysignal.h，在这个文件中包含信号或槽的声明，那么在编译之前元对象编译器工具就会根据该文件自动生成一个名为 mysignal.moc.h 的 C++源文件并将其提交给编译器；对应的 mysignal.cpp 文件元对象编译器工具将自动生成一个名为 mysignal.moc.cpp 的文件提交给编译器。

元对象代码是信号槽机制所必需的。用元对象编译器产生的 C++源文件必须与类实现一起进行编译和连接，或者用#include 语句将其包含到类的源文件中。元对象编译器并不扩展#include 或者#define 宏定义，只是简单地跳过所遇到的任何预处理指令。

信号和槽函数的声明一般位于头文件中，同时在类声明的开始位置必须加上 Q_OBJECT 语句，Q_OBJECT 语句将告诉编译器在编译之前必须先应用元对象编译器工具进行扩展。关键字 signals 是对信号的声明，signals 的默认属性为 protected。关键字 slots 是对槽函数的声明，slots 有 public、private、protected 等属性。signals、slots 关键字是 Qt 自己定义的，不是 C++中的关键字。信号的声明类似于函数的声明而非变量的声明，左边要有

类型，右边要有括号，如果要向槽中传递参数，则应在括号中指定每个形式参数的类型，而形式参数的个数可以多于一个。关键字 slots 指出随后开始槽的声明，这里 slots 用的也是复数形式。槽的声明与普通函数的声明一样，可以携带零个或多个形式参数。既然信号的声明类似于普通 C++ 函数的声明，那么，信号也可采用 C++ 中虚函数的形式进行声明，即同名但参数不同。例如，第 1 次定义的 void mySignal()没有带参数，而第 2 次定义的却带有参数，从这里可以看出 Qt 的信号机制是非常灵活的。信号槽之间的联系必须事先用 connect()函数进行指定。如果要断开二者之间的联系，则可以使用 disconnect()函数。

6. 标准信号槽案例应用

新建一个 Qt Widgets Application 项目（笔者的项目名称为 MySignalSlotsDemo），基类选择 QWidget，如图 3-1 所示，然后在构造函数中动态地创建一个按钮，实现单击按钮关闭窗口的功能。编译并运行该程序，效果如图 3-2 所示。

注意：该案例的完整工程代码可参考本书源码中的 chapter3/MySignalSlotsDemo，建议读者先下载源码将工程运行起来，然后结合本书进行学习。

图 3-1　Qt Widgets 项目的基类选择

图 3-2　Qt 信号槽的运行效果

本项目包含的代码如下：

```
//chapter3/MySignalSlotsDemo/widget.h
//widget.h 头文件////
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>

namespace Ui {
class Widget;
}

class Widget : public QWidget
{
    Q_OBJECT

public:
    explicit Widget(QWidget *parent = nullptr);
    ~Widget();

private:
    Ui::Widget *ui;
};
#endif //WIDGET_H


/////widget.cpp 文件//////
#include "widget.h"
#include "ui_widget.h"
#include <QPushButton>

Widget::Widget(QWidget *parent) :
    QWidget(parent),
    ui(new Ui::Widget)
{
    ui->setupUi(this);

    //创建一个按钮
    QPushButton *btn = new QPushButton;
    //btn->show(); //以顶层方式弹出窗口控件
    //让 btn 依赖在 myWidget 窗口中
    btn->setParent(this); //this 指当前窗口
    btn->setText("关闭");
    btn->move(100,100);

    //关联信号和槽:单击按钮关闭窗口
    //参数 1:信号发送者;参数 2:发送的信号(函数地址)
    //参数 3:信号接收者;参数 4:处理槽函数地址
    connect(btn, &QPushButton::clicked, this, &QWidget::close);
```

```
}

Widget::~Widget()
{
    delete ui;
}
```

7. 自定义信号槽案例应用

图 3-3　Qt 项目中 Add New 添加新项

当 Qt 提供的标准信号和槽函数无法满足需求时，就需要用到自定义信号槽，可以使用 emit 关键字来发射信号。例如定义老师和学生两个类（都继承自 QObject），当老师发出“下课”信号时，学生响应“去吃饭”的槽功能。由于“下课”不是 Qt 标准的信号，所以需要用到自定义信号槽机制。这里不再创建新的 Qt 项目，直接使用上文的 MySignalSlotsDemo 项目，先添加两个自定义类 Teacher 和 Student，它们都继承自 QObject。用鼠标右击项目名称 MySignalSlotsDemo，在弹出的快捷菜单中选择 Add New...菜单选项，如图 3-3 所示，然后在弹出的“新建文件”对话框中，单击左侧的 C++模板，在右侧选择 C++Class，如图 3-4 所示，然后在弹出的 C++Class 对话框中，输入 Class name(Teacher)，在 Base class 下拉列表中选择 QObject，如图 3-5 所示，然后以同样的步骤创建 Student 类，成功后，项目中多了两个类(Teacher 和 Student)，如图 3-6 所示。

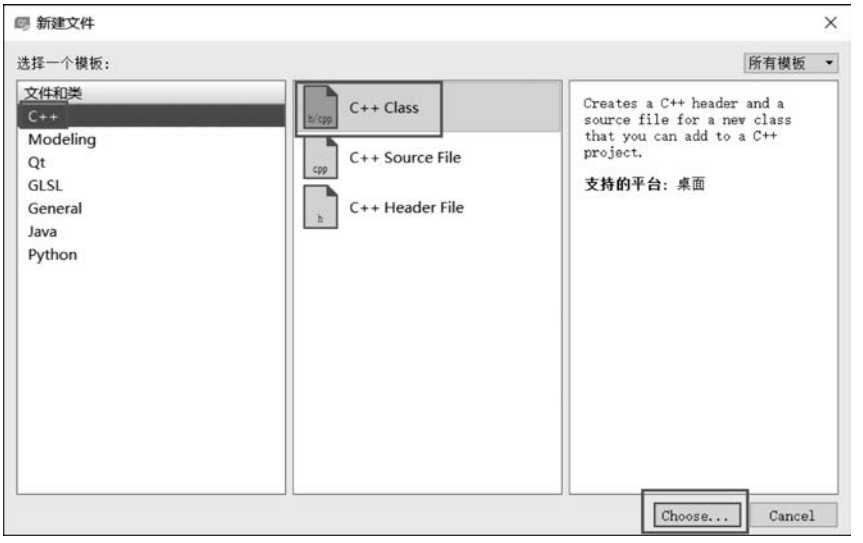

图 3-4　Qt 项目中选择 C++Class

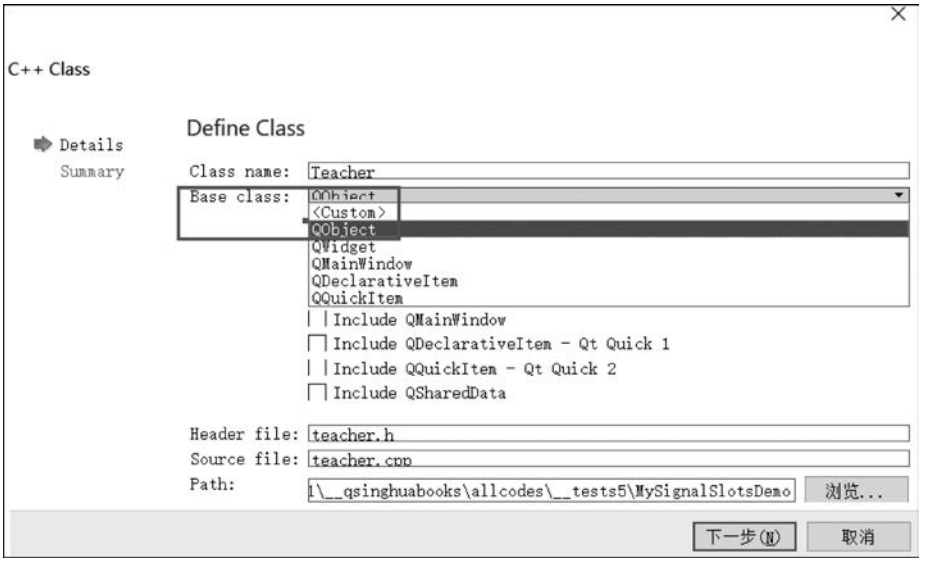

图 3-5　Qt 项目中添加新类并选择 QObject 基类

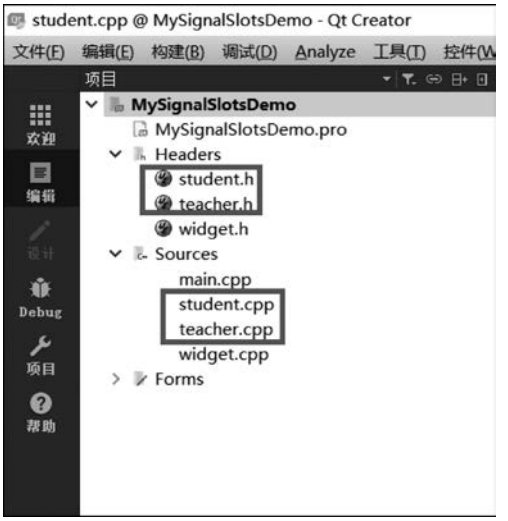

图 3-6　Qt 项目中添加了两个类

在 Teacher 类中添加一个“下课”信号(finishClass),代码如下:

```
//chapter3/MySignalSlotsDemo/student.h
signals:
    //自定义信号,写到 signals 下
    //返回值为 void,只用申明,不需要实现
    //可以有参数,可以重载
    void finishClass();
```

在 Student 类中添加一个“去吃饭”槽(gotoEat),代码如下:

```
//chapter3/MySignalSlotsDemo/student.h
public slots:
//在早期 Qt 版本中需要写到 public slots 下,高级版本可以写到 public 或全局下
//返回值为 void,需要声明,也需要实现
//可以有参数,可以重载
    void gotoEat();

//sutdent.cpp
void Student::gotoEat()
{
    qDebug() << "准备去吃饭……";
}
```

在 Widget 类中声明老师类(Teacher)和学生类(Student)的成员变量,并在构造函数中通过 new 创建实例,然后通过 connect()函数来关联老师类的“下课”信号和学生类的“去吃饭”槽,代码如下:

```
//chapter3/MySignalSlotsDemo/widget.h
private:
    Teacher * m_teacher;
    Student * m_student;

//widget.cpp
Widget::Widget(QWidget * parent) :
    QWidget(parent),
    ui(new Ui::Widget)
{
    //省略其他代码
    //创建老师对象
    this->m_teacher = new Teacher(this);
    //创建学生对象
    this->m_student = new Student(this);
    //连接老师的"下课"信号和学生的"去吃饭"槽函数
    connect(m_teacher,&Teacher::finishClass,
            m_student,&Student::gotoEat);
}
```

然后在界面上拖曳一个按钮,将文本内容修改为“下课”,用来模拟老师的下课信号,然后双击这个按钮,在 Qt 自动生成的 Widget::on_pushButton_clicked()函数中添加的代码如下:

```
//chapter3/MySignalSlotsDemo/widget.cpp
void Widget::on_pushButton_clicked()
{
    //通过 emit 发射信号
    emit this->m_teacher->finishClass();
}
```

编译并运行该程序，单击“下课”按钮，此时会在控制台输出“准备去吃饭……”，证明学生类的槽函数被成功触发，如图 3-7 所示。在本案例中老师类和学生类的相关代码如下（其余代码读者可参考源码工程）：

```
//chapter3/MySignalSlotsDemo/teacher.h
////teacher.h////
#ifndef TEACHER_H
#define TEACHER_H
#include <QObject>

class Teacher : public QObject
{
    Q_OBJECT
public:
    explicit Teacher(QObject *parent = nullptr);

signals:
    //自定义信号,写到 signals 下
    //返回值为 void,只用申明,不需要实现
    //可以有参数,可以重载
    void finishClass();

public slots:

};
#endif //TEACHER_H


////teacher.cpp////
#include "teacher.h"
Teacher::Teacher(QObject *parent) : QObject(parent)
{

}


////student.h////
#ifndef STUDENT_H
#define STUDENT_H
#include <QObject>

class Student : public QObject
{
    Q_OBJECT
public:
    explicit Student(QObject *parent = nullptr);

signals:

public slots:
```

```
//在早期 Qt 版本中需要写到 public slots 下,高级版本可以写到 public 或全局下
//返回值为 void,需要声明,也需要实现
//可以有参数,可以重载
    void gotoEat();
};
#endif //STUDENT_H


////student.cpp////
#include "student.h"
#include <QDebug>
Student::Student(QObject *parent) : QObject(parent)
{
}

void Student::gotoEat()
{
    qDebug() << "准备去吃饭……";
}
```

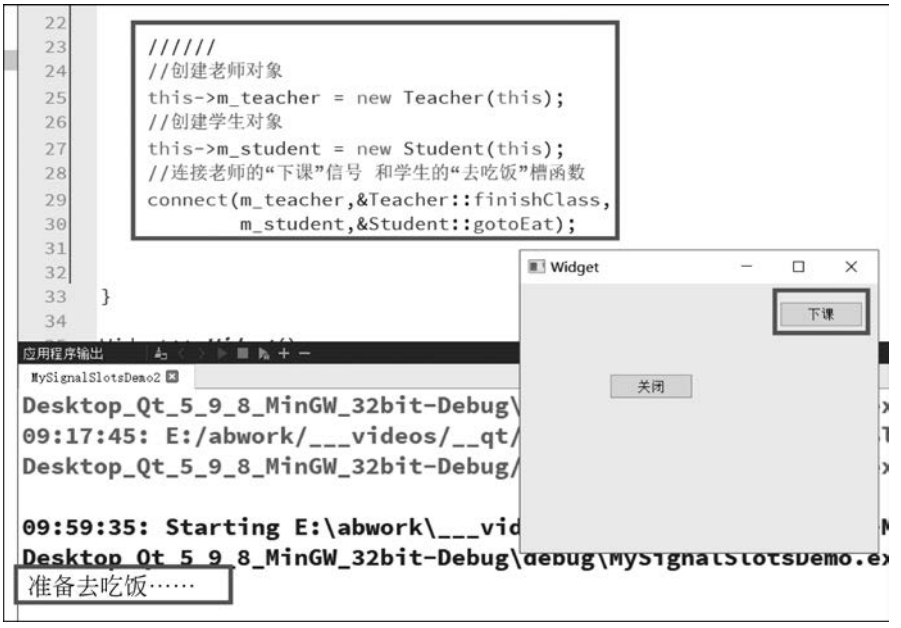

图 3-7　Qt 项目中自定义信号槽的应用

3.2　Qt 显示图像

Qt 可显示基本的图像类型,利用 QImage、QPxmap 类可以实现图像的显示,并且利用类中的方法可以实现图像的基本操作(缩放、旋转等)。Qt 可以直接读取并显示的格式有

BMP、GIF、JPG、JPEG、PNG、TIFF、PBM、PGM、PPM、XBM 和 XPM 等。可以使用 QLabel 显示图像，QLabel 类有 setPixmap()函数，可以用来显示图像。也可以直接用 QPainter 画出图像。如果图像过大，当直接用 QLabel 显示时，则会出现部分图像显示不出来的情况，这时可以用 Scroll Area 部件。

1. Qt 显示图像

首先使用 QFileDialog 类的静态函数 getOpenFileName()打开一张图像，将图像文件加载进 QImage 对象中，再用 QPixmap 对象获得图像，最后用 QLabel 选择一个 QPixmap 图像对象进行显示。该过程的关键代码如下(完整代码可参考 chapter3/QtImageDemo 工程)：

```
//chapter3/QtImageDemo/widget.cpp
//Qt 显示图片
void Widget::on_btnShowImage_clicked()
{
    QString filename;
    filename = QFileDialog::getOpenFileName(this,
tr("选择图像"),"",tr("Images ( * .png * .bmp * .jpg * .tif * .GIF )"));
    if(filename.isEmpty()){
         return;
    }
    else{
        m_img = new QImage;
        if(! (m_img->load(filename) ) ) //加载图像
        {
            QMessageBox::information(this,
                        tr("打开图像失败"),tr("打开图像失败!"));
            delete m_img;
            return;
        }
        ui->lblImage->setPixmap(QPixmap::fromImage( * m_img));
    }
}
```

QImage 为图像的像素级访问进行了优化，QPixmap 使用底层平台的绘制系统进行绘制，无法提供像素级别的操作，而 QImage 则使用独立于硬件的绘制系统。编译并运行该工程，单击“显示图像”按钮，选择一张本地的图片，如图 3-8 所示。

图 3-8 Qt 使用 QImage 和 QPixmap 显示图像

2. Qt 缩放图像

Qt 缩放图像可以用 scaled()函数，函数原型的代码如下：

```
//chapter3/qt-help-apis.txt
QImage QImage::scaled (const QSize & size,Qt::
AspectRatioMode
```

```
aspectRatioMode = Qt::IgnoreAspectRatio,
Qt::TransformationModetransformMode = Qt::FastTransformation ) const;
```

利用上面已经加载成功的图像(m_img)，在 scaled()函数中 width 和 height 分别表示缩放后图像的宽和高，即将原图像缩放到 width×height 大小。例如在本案例中显示的图像的原始长和宽为 200×200，缩放后修改为 100×100，编译并运行，如图 3-9 所示，代码如下：

```
//chapter3/QtImageDemo/widget.cpp
void Widget::on_btnScale_clicked(){
    QImage * imgScaled = new QImage;
     * imgScaled = m_img -> scaled(100,100,
Qt::KeepAspectRatio);
     ui -> lblScale -> setPixmap(QPixmap::
fromImage( * imgScaled));
}
```

图 3-9　Qt 缩放图像

3. Qt 旋转图像

Qt 旋转图像可以用 QMatrix 类的 rotate()函数，代码如下：

```
//chapter3/QtImageDemo/widget.cpp
void Widget::on_btnRotate_clicked(){
    QImage * imgRotate = new QImage;
    QMatrix matrix;
    matrix.rotate(270);
     * imgRotate = m_img -> transformed(matrix);
    ui -> lblRotate -> setPixmap(QPixmap::fromImage( * imgRotate));
}
```

编译并运行该项目，使用时依次单击“显示图像”“缩放”“旋转”按钮，效果如图 3-10 所示。

图 3-10　Qt 显示、缩放和旋转图像

3.3 Qt 实现图片轮播

网页上经常看到的各种广告基本会使用图片轮播技术，也可以用 Qt 实现图片轮播的效果，实现小区域内嵌入多个广告的效果，主要包括定时自动切换广告图片和手动单击选择切换图片这两种方式。可以使用 Qt 的动画类（QPropertyAnimation）实现图片轮播效果，如图 3-11 所示。

注意：该案例的完整工程代码可参考本书源码中的 chatper3/CarouselImageWindow，建议读者先下载源码将工程运行起来，然后结合本书进行学习。

图 3-11 Qt 实现的图片轮播效果

1. 创建 Qt 项目并准备图片

首先创建 Qt Widgets Application 项目，将默认的窗口类名修改为 CarouselImageWindow，基类选择 QWidget，将项目名称修改为 CarouselImageWindow，如图 3-12 所示。创建项目成功后，需要添加资源文件（images.qrc）。右击项目名称，在弹出的对话框中选择 Qt→Qt Resource File，然后单击 Choose 按钮，如图 3-13 所示，资源名称输入 images 即可，然后在 Qt 项目中会多出一个 images.qrc 资源文件。单击右侧的“添加”按钮，在“前缀”文本框中输入“/”即可，如图 3-14 所示。在项目根目录下新建一个文件夹（QtImagesRes），存储 5 张图片，如图 3-15 所示。单击右侧的“添加”按钮并选择“文件”下拉选项，如图 3-16 所示，在弹出的页面中选择 QtImagesRes 目录下的图片，依次操作将这 5 张图片都添加到资源中，如图 3-17 所示。

2. QPropertyAnimation 动画类

在 Qt 中可以使用自带的属性动画类（QPropertyAnimation）实现动画效果。使用时需要包含头文件，先通过 new 创建出一个实例来，然后设置使用动画的控件及动画效果，一个简单的案例代码如下：

图 3-12　创建 Qt 项目并修改类名称

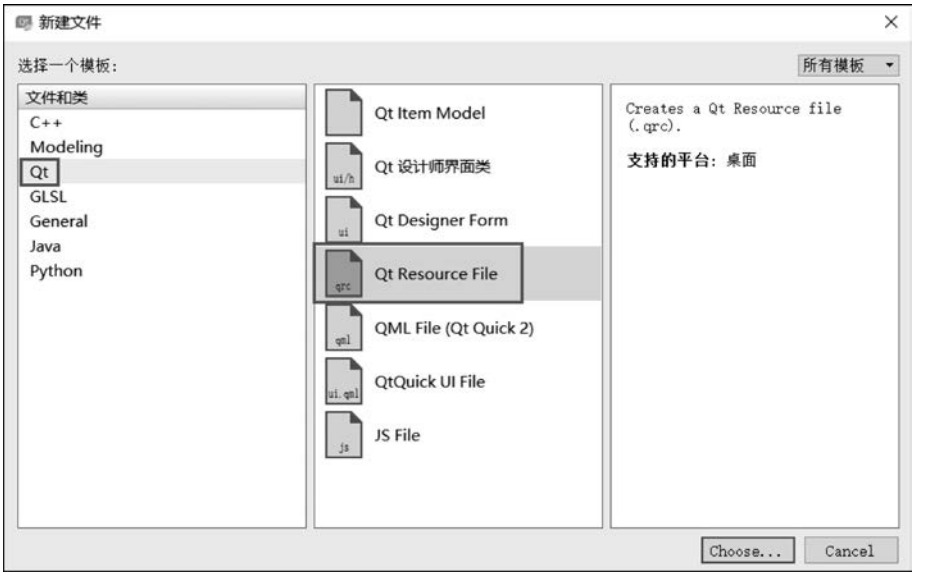

图 3-13　Qt 添加资源文件

```
//chapter3/qt - help - apis.txt
# include < QPropertyAnimation > //包含头文件
//注意,在类的头文件中定义 QPropertyAnimation 类型的成员变量
//QPropertyAnimation * m_animation;
void Animation::createAnimation(){
    m_animation = new QPropertyAnimation();                    //创建动画
    m_animation - > setTargetObject(label);                    //设置使用动画的控件
    m_animation - > setEasingCurve(QEasingCurve::Linear);      //设置动画效果
}
```

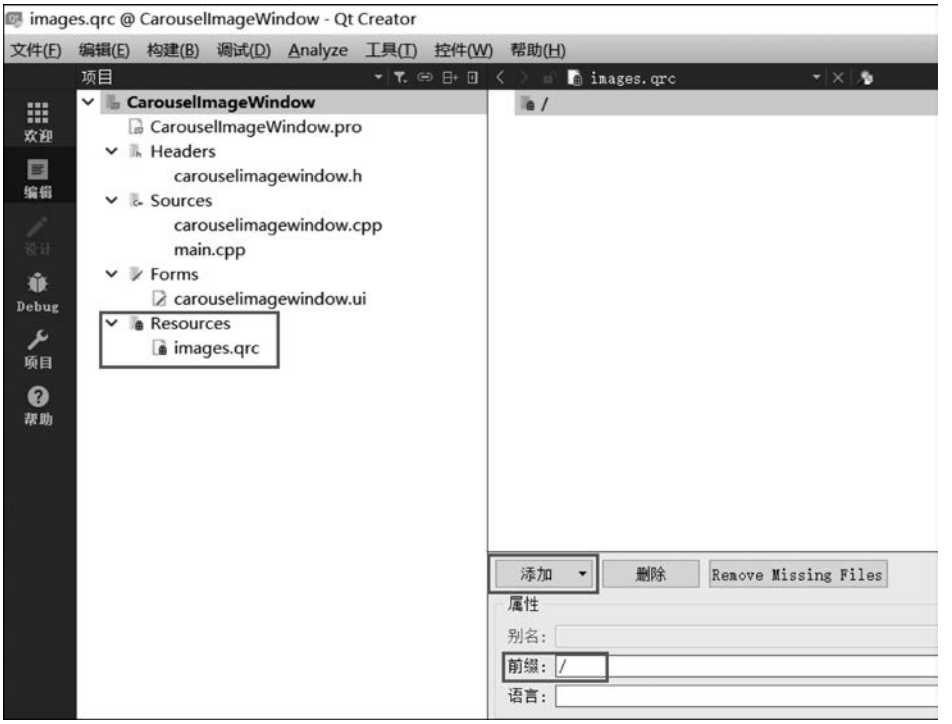

图 3-14　Qt 的资源前缀

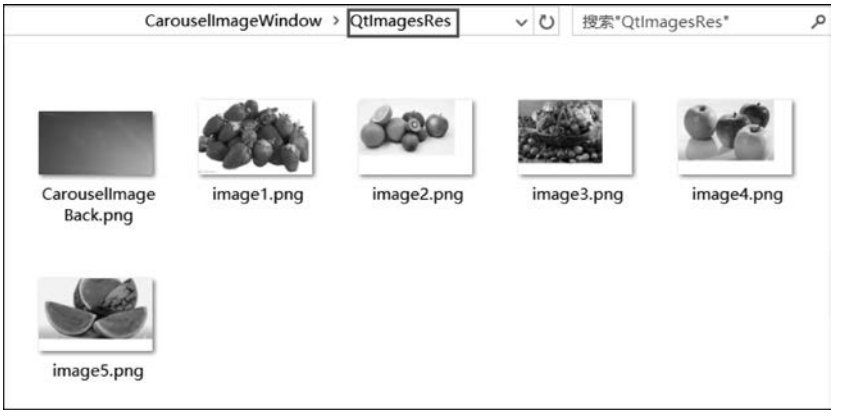

图 3-15　准备原始图片素材

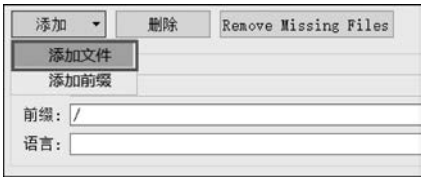

图 3-16　将本地原始图片添加到 Qt 资源中

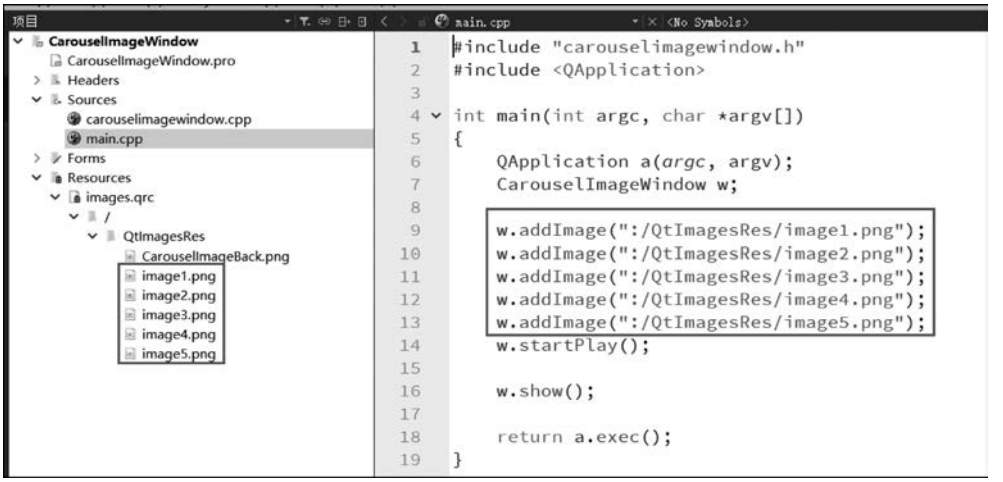

图 3-17　添加完毕后的图片资源

可以使动画按照点移动，并可以设置动画持续时间，代码如下：

```
//chapter3/qt-help-apis.txt
void Animation::moveAnimation(){
    //pos:按点移动的动画(移动)
    m_animation->setPropertyName("pos");                    //指定动画属性名
    m_animation->setDuration(3000);                         //设置动画时间(单位:毫秒)
    m_animation->setStartValue(label->pos());               //将动画起始位置设置在 label
                                                            //控件当前的 pos
    m_animation->setEndValue(label->pos() + QPoint(200, 100));   //设置动画结束位置
    m_animation->start();                                   //启动动画
}
```

可以使动画实现缩放效果，代码如下：

```
//chapter3/qt-help-apis.txt
void Animation::zoom(){
    m_animation->setPropertyName("geometry");               //指定动画属性名
    m_animation->setDuration(3000);                         //设置动画时间(单位:毫秒)
    m_animation->setStartValue(label->rect());              //设置动画起始位置

    //获取控件初始的大小
    int width = label->rect().width();
    int height = label->rect().height();

    //设置动画步长值,以及在该位置时的长和宽
    m_animation->setKeyValueAt(0.5, QRect(label->pos(),QSize(width - 20, height - 20)));
    //设置动画结束位置及其大小
    m_animation->setEndValue(QRect(label->pos(),QSize(width-100, height-100)));
    m_animation->start();                                   //启动动画
}
```

也可以设置窗口的不透明效果(只对顶级窗口有效),代码如下:

```
//chapter3/qt-help-apis.txt
void Animation::opcity(){
    //windowOpacity:不透明度(注意该效果只对顶级窗口有效)
    m_animation->setTargetObject(this);            //重设动画使用对象
    m_animation->setPropertyName("windowOpacity");//指定动画属性名
    m_animation->setDuration(2000);                //设置动画时间(单位:毫秒)

    //设置动画步长值,以及在该位置时显示的透明度
    m_animation->setKeyValueAt(0, 1);
    m_animation->setKeyValueAt(0.5, 0);
    m_animation->setKeyValueAt(1, 0);

    //动画循环次数,-1表示一直循环
    m_animation->setLoopCount(-1);          //当值为-1时,动画一直运行,直到窗口关闭
    m_animation->start();                          //启动动画
}
```

3. QTimer 定时器类

QTimer类提供了一个既可重复触发又可单次触发的定时器,它是一个高层次的应用程序接口。要使用它,只需创建一个QTimer类对象,将它的timeout()信号连接到适当的槽函数上,然后调用其start()函数开启定时器。此后,QTimer对象就会周期性地发出timeout()信号,与此关联的槽函数就会被自动触发。例如,一个1s执行一次的定时器,代码如下:

```
//chapter3/qt-help-apis.txt
QTimer *timer = new QTimer(this);              //创建定时器
//update()函数是当前窗口类中的一个函数,关联信号槽函数
connect(timer, SIGNAL(timeout()), this, SLOT(update()));
timer->start(1000);                            //在start函数中将时间间隔指定为1000ms
```

在上述代码中update()函数每隔1s就会被调用一次。当然也可以让一个QTimer对象在启动后只触发一次,只需调用该类的setSingleShot(true)函数。更简单的做法是使用该类的静态方法QTimer::singleShot(),以某个时间间隔来启动一个单次触发的定时器,代码如下:

```
QTimer::singleShot(2000, this, SLOT(updateCaption()));
```

上面这句代码执行结束,2s后会调用一次updateCaption()函数,并且只调用一次。

4. Qt 实现图片轮播

在Qt中可以使用属性动画类(QPropertyAnimation)结合定时器类(QTimer)实现图片轮播效果。在上述项目中创建的窗口类(CarouselImageWindow)继承自QWidget,为了实现图片轮播效果,需要为它添加几个成员变量和成员函数,代码如下:

```
//chapter3/CarouselImageWindow/carouselimagewindow.h
#ifndef CAROUSELIMAGEWINDOW_H
#define CAROUSELIMAGEWINDOW_H

#include <QWidget>
#include <QScrollArea>
#include <QTimer>
#include <QPropertyAnimation>
#include <QPushButton>

namespace Ui {
class CarouselImageWindow;
}

class CarouselImageWindow : public QWidget{
    Q_OBJECT

public:
    explicit CarouselImageWindow(QWidget *parent = nullptr);
    ~CarouselImageWindow();
    //设置图片列表
    void setImageList(QStringList imageFileNameList);
    //添加图片
    void addImage(QString imageFileName);
    //开始播放
    void startPlay();

private:
    //初始化图片切换按钮
    void initChangeImageButton();
    //绘图事件
    void paintEvent(QPaintEvent *event);

    //鼠标单击事件
    void mousePressEvent(QMouseEvent* event);

public slots:
    //图片切换时钟
    void onImageChangeTimeout();

    //图片切换按钮单击
    void onImageSwitchButtonClicked(int buttonId);


private:
    Ui::CarouselImageWindow *ui;

    //图片列表
    QList<QString> m_imageFileNameList;
    //切换图片
    QPixmap m_curPixmap;
```

```
    QPixmap m_nextPixmap;
    //图片切换动画类
    QPropertyAnimation * m_opacityAnimation;

    //图片切换时钟
    QTimer m_imageChangeTimer;

    //当前显示图片的 index
    int m_curDrawImageIndx;

    //按钮列表
    QList < QPushButton * > m_pButtonChangeImageList;

};

# endif //CAROUSELIMAGEWINDOW_H
```

分析代码发现，主要在头文件中添加了图片切换动画类的实例(m_opacityAnimation)、定时器类的实例(m_imageChangeTimer)及图片列表(m_imageFileNameList)等。在该类的构造函数中需要进行初始化工作，代码如下：

```
//chapter3/CarouselImageWindow/carouselimagewindow.cpp
CarouselImageWindow::CarouselImageWindow(QWidget * parent) :
    QWidget(parent),
    ui(new Ui::CarouselImageWindow),
    m_curDrawImageIndx(0){
    ui -> setupUi(this);
    //添加 ImageOpacity 属性
    this -> setProperty("ImageOpacity", 1.0);

    //动画切换类
    m_opacityAnimation = new QPropertyAnimation(this, "ImageOpacity");
    //这里要设置的动画时间小于图片切换时间
    m_opacityAnimation -> setDuration(1500);

    //设置 ImageOpacity 属性值的变化范围
    m_opacityAnimation -> setStartValue(1.0);
    m_opacityAnimation -> setEndValue(0.0);

    //透明度变化及时更新绘图
    connect(m_opacityAnimation, SIGNAL(valueChanged(const QVariant&)), this, SLOT(update
()));

    //设置图片切换时钟的槽函数
    connect(&m_imageChangeTimer, SIGNAL(timeout()), this, SLOT(onImageChangeTimeout()));
    this -> setFixedSize(QSize(400, 250));
    this -> setWindowFlags(Qt::FramelessWindowHint);
}
```

定时器关联的 onImageChangeTimeout()槽函数通过修改索引值(m_curDrawImageIndx)

来准备下一张图片素材，然后调用 start()函数让动画类重新开始即可，代码如下：

```
//chapter3/CarouselImageWindow/carouselimagewindow.cppvoid CarouselImageWindow::
onImageChangeTimeout(){
    //设置前后的图片
    m_curPixmap = QPixmap(m_imageFileNameList.at(m_curDrawImageIndx));
    m_curDrawImageIndx++;
    if (m_curDrawImageIndx >= m_imageFileNameList.count()) {
        m_curDrawImageIndx = 0;
    }
    m_nextPixmap = QPixmap(m_imageFileNameList.at(m_curDrawImageIndx));

    m_pButtonChangeImageList[m_curDrawImageIndx]->setChecked(true);

    //动画类重新开始
    m_opacityAnimation->start();
}
```

界面的绘制工作在 paintEvent()函数中实现，它是被高度优化过的函数，本身已经自动开启并实现了双缓冲机制，因此在 Qt 中重绘操作不会引起屏幕上的任何闪烁现象。paintEvent(QPaintEvent *)函数是 QWidget 类中的虚函数，用于 UI 界面的绘制，它会在多种情况下被其他函数自动调用，例如 update()函数。重绘事件用来重绘一个部件的全部或者部分区域，下面几个原因中的任意一个都会发生重绘事件：

(1) repaint()函数或者 update()函数被调用时。

(2) 被隐藏的部件被重新显示时。

(3) 其他一些原因(例如强制绘制整个界面时)。

在本案例中主要根据图片索引通过 QPainter 类将图片显示出来，代码如下：

```
//chapter3/CarouselImageWindow/carouselimagewindow.cppvoid CarouselImageWindow::paintEvent
(QPaintEvent *event){
    QPainter painter(this);
    QRect imageRect = this->rect();

    //如果图片列表为空,则显示默认图片
    if (m_imageFileNameList.isEmpty()){
    QPixmap backPixmap = QPixmap(":/QtImagesRes/CarouselImageBack.png");
    painter.drawPixmap(imageRect, backPixmap.scaled(imageRect.size()));
    }
    //如果只有一张图片
    else if (m_imageFileNameList.count() == 1){
    QPixmap backPixmap = QPixmap(m_imageFileNameList.first());
    painter.drawPixmap(imageRect, backPixmap.scaled(imageRect.size()));
    }
    //如果有多张图片
    else if (m_imageFileNameList.count() > 1){
    float imageOpacity = this->property("ImageOpacity").toFloat();
    painter.setOpacity(1);
    painter.drawPixmap(imageRect, m_nextPixmap.scaled(imageRect.size()));
```

```
        painter.setOpacity(imageOpacity);
        painter.drawPixmap(imageRect, m_curPixmap.scaled(imageRect.size()));
        }
    }
```

图片素材和相关资源的准备工作是在 initChangeImageButton()函数中进行的,代码如下:

```
//chapter3/CarouselImageWindow/carouselimagewindow.cppvoid CarouselImageWindow::
initChangeImageButton(){
    //注意,当图片过多时按钮可能放置不下
    QButtonGroup * changeButtonGroup = new QButtonGroup;
    QHBoxLayout * hLayout = new QHBoxLayout();
    hLayout->addStretch();
    //根据图片数量来创建按钮,形状为矩形按钮
    for (int i = 0; i < m_imageFileNameList.count(); i++){
        QPushButton * pButton = new QPushButton;
        pButton->setFixedSize(QSize(16, 16));
        pButton->setCheckable(true);
        changeButtonGroup->addButton(pButton, i);
        m_pButtonChangeImageList.append(pButton);
        hLayout->addWidget(pButton);
    }
    hLayout->addStretch();
    hLayout->setSpacing(10);
    hLayout->setMargin(0);
//单击按钮也可以实现图片切换
    connect(changeButtonGroup, SIGNAL(buttonClicked(int)), this, SLOT(onImageSwitchButtonClicked
(int)));

    QVBoxLayout * mainLayout = new QVBoxLayout(this);
    mainLayout->addStretch();
    mainLayout->addLayout(hLayout);
    mainLayout->setContentsMargins(0, 0, 0, 20);
}
```

最后通过 startPlay()函数来开启动画,如果有多张图片,则需要调用 update()函数,代码如下:

```
//chapter3/CarouselImageWindow/carouselimagewindow.cppvoid CarouselImageWindow::startPlay(){
    //添加完图片之后,根据图片数量设置图片切换按钮
    initChangeImageButton();
    if (m_imageFileNameList.count() == 1){
        m_pButtonChangeImageList[m_curDrawImageIndx]->setChecked(true);
    }
    else if (m_imageFileNameList.count() > 1){
        m_pButtonChangeImageList[m_curDrawImageIndx]->setChecked(true);
        m_curPixmap = QPixmap(m_imageFileNameList.at(m_curDrawImageIndx));
        m_imageChangeTimer.start(2000);
        update();
    }
}
```