QT对象模型和正则表达式

软件学院
马玲 张圣林

主 要 内 容

- 对象模型
- 正则表达式
- 小结

对象模型

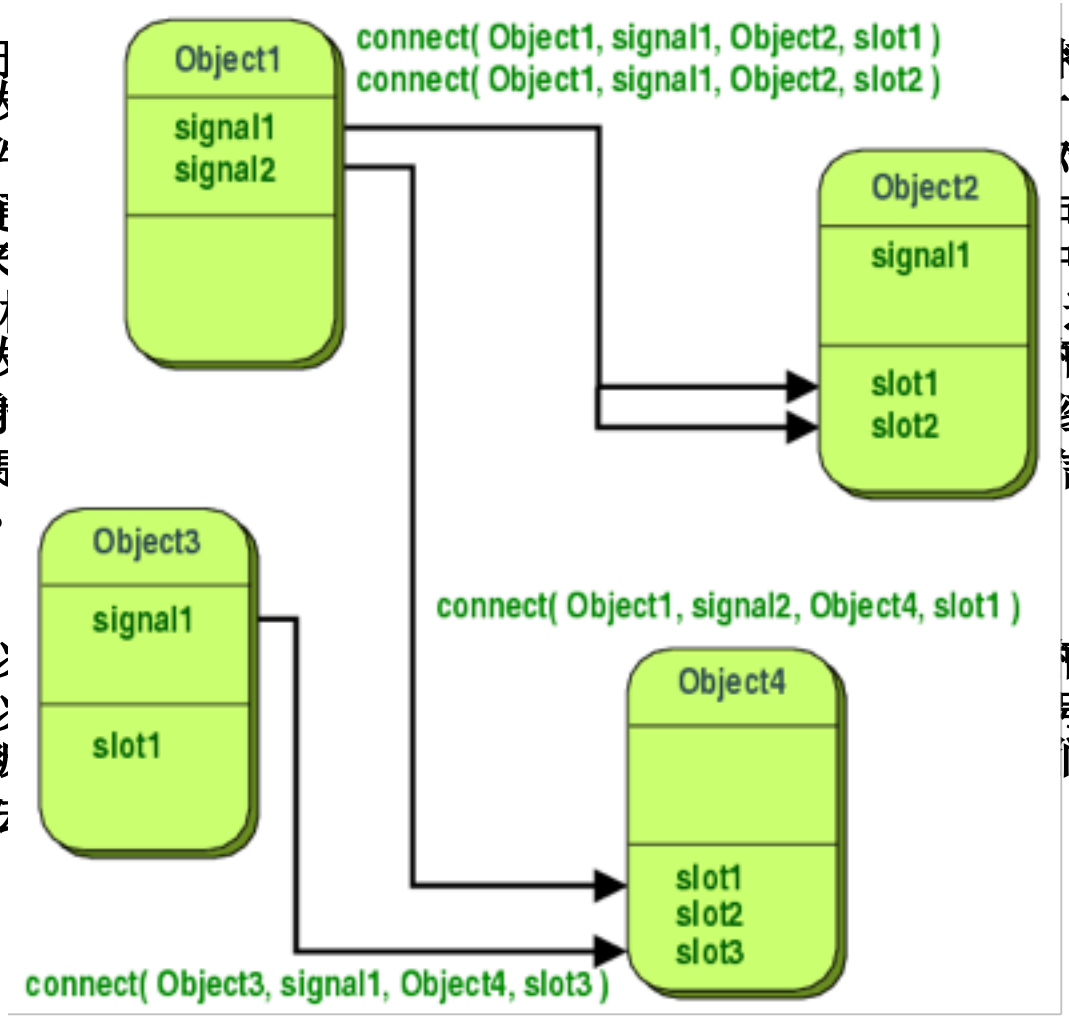
标准C++对象模型可以在运行时非常有效的支持对象范式（object paradigm），但是它的静态特性在一些问题领域中不够灵活。图形用户界面编程不仅需要运行时的高效性，还需要高度的灵活性。为此，Qt在标准C++对象模型的基础上添加了一些特性，形成了自己的对象模型。这些特性有：

- 一个强大的无缝对象通信机制——信号和槽（signals and slots）；
- 可查询和可设计的对象属性系统（object properties）；
- 强大的事件和事件过滤器（events and event filters）；
- 通过上下文进行国际化的字符串翻译机制（string translation for internationalization）；
- 完善的定时器（timers）驱动，使得可以在一个事件驱动的GUI中处理多个任务；
- 分层结构的、可查询的对象树（object trees），它使用一种很自然的方式来组织对象拥有权（object ownership）；
- 守卫指针即QPointer，它在引用对象被销毁时自动将其设置为0；
- 动态的对象转换机制（dynamic cast）；

Qt的这些特性都是在遵循标准C++规范内实现的，使用这些特性都必须要继承自QObject类。其中对象通信机制和动态属性系统，还需要元对象系统（Meta-Object System）的支持。

信号和槽

- 信号和槽用于
同于其他开发
希望其他部件
他对象进行通
close()函数来
（callback）
特殊的事情发
它在信号发射
号和槽，但是
想要的功能。



特征，也是Qt不
个部件时，总
对象都可以和其
可以执行窗口的
使用了回调
通信。当一个
槽就是一个函数，
经定义了一些信
信号和槽来实现

- 一个信号可以
个信号还可以
当这个信号被
随机的，无法

槽上，甚至，一
号相关联，那么，
门执行的顺序是

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 Widget::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,SIGNAL(dlgReturn(int)),this,SLOT(showValue(int)));
```

connect()函数原型如下：

```
bool QObject::connect ( const QObject * sender, const char * signal, const QObject * receiver, const char *
    method, Qt::ConnectionType type = Qt::AutoConnection )
```

- 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，这里是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，这里是SLOT(showValue(int))。
- 对于信号和槽，必须使用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。
- 信号和槽的参数只能有类型，不能有变量，例如写成SLOT(showValue(int value))是不对的。对于信号和槽的参数问题，基本原则是信号中的参数类型要和槽中的参数类型相对应，而且信号中的参数可以多于槽中的参数，但是不能反过来，如果信号中有多余的参数，那么它们将被忽略。

关联方式

connect()函数的最后一个参数，它表明了关联的方式，其默认值是Qt::AutoConnection，这里还有其他几个选择，具体功能如下表所示 。

常量	描述
Qt::AutoConnection	如果信号和槽在不同的线程中，同 Qt::QueuedConnection；如果信号和槽在同一个线程中，同Qt::DirectConnection
Qt::DirectConnection	发射完信号后立即执行槽，只有槽执行完成返回后，发射信号处后面的代码才可以执行
Qt::QueuedConnection	接收部件所在线程的事件循环返回后再执行槽，无论槽执行与否，发射信号处后面的代码都会立即执行
Qt::BlockingQueuedConnection	类似 Qt::QueuedConnection，只能用在信号和槽在不同的线程的情况下
Qt::UniqueConnection	类似 Qt::AutoConnection，但是两个对象间的相同的信号和槽只能有唯一的关联
Qt::AutoCompatConnection	类似Qt::AutoConnection，它是Qt 3 中的默认类型

Qt 5中新加的关联形式

connect()函数另一种常用的基于函数指针的重载形式：

```
[static] QMetaObject::Connection QObject::connect(
                const QObject *sender, PointerToMemberFunction signal,
                const QObject *receiver,PointerToMemberFunction method,
                Qt::ConnectionType type = Qt::AutoConnection)
```

与前者最大的不同就是，指定信号和槽两个参数时不用再使用SIGNAL()和SLOT()宏，并且槽函数不再必须是使用slots关键字声明的函数，而可以是任意能和信号关联的成员函数。要使一个成员函数可以和信号关联，那么这个函数的参数数目不能超过信号的参数数目，但是并不要求该函数拥有的参数类型与信号中对应的参数类型完全一致，只需要可以进行隐式转换即可。使用这种重载形式，前面程序中的关联可以使用如下代码代替：

```
connect(dlg, &MyDialog::dlgReturn, this, &Widget::showValue);
```

使用这种方式与前一种相比，还有一个好处就是可以在编译时进行检查，信号或槽的拼写错误、槽函数参数数目多于信号的参数数目等错误在编译时就能够被发现。

使用信号和槽注意事项

- 需要继承自QObject或其子类；
- 在类声明的最开始处添加Q_OBJECT宏；
- 槽中的参数的类型要和信号的参数的类型相对应，且不能比信号的参数多；
- 信号只用声明，没有定义，且返回值为void类型。

信号和槽自动关联

信号和槽还有一种自动关联方式，比如在设计模式直接生成的“确定”按钮的单击信号的槽，就是使用的这种方式：

on_pushButton_clicked()

它由“on”、部件的objectName和信号三部分组成，中间用下划线隔开。这样组织的名称的槽就可以直接和信号关联，而不用再使用connect()函数。

信号和槽的高级应用

- 有时希望获得信号发送者的信息，在Qt中提供了QObject::sender()函数来返回发送该信号的对象的指针。

- 但是如果有多个信号关联到了同一个槽上，而在该槽中需要对每一个信号进行不同的处理，使用上面的方法就很麻烦了。对于这种情况，便可以使用QSignalMapper类。QSignalMapper可以被叫做信号映射器，它可以实现对多个相同部件的相同信号进行映射，为其添加字符串或者数值参数，然后再发射出去。

信号和槽机制的特色和优越性

信号和槽机制的特色和优越性：

- 信号和槽机制是类型安全的，相关联的信号和槽的参数必须匹配；
- 信号和槽是松耦合的，信号发送者不知道也不需要知道接受者的信息；
- 信号和槽可以使用任意类型的任意数量的参数。

  虽然信号和槽机制提供了高度的灵活性，但就其性能而言，还是慢于回调机制的。当然，这点性能差异通常在一个应用程序中是很难体现出来的。

属性系统

Qt提供了强大的基于元对象系统的属性系统，可以在能够运行Qt的平台上支持任意的标准C++编译器。要声明一个属性，那么该类必须继承自QObject类，而且还要在声明前使用Q_PROPERTY()宏：

Q_PROPERTY(type name

（READ getFunction [WRITE setFunction] |

MEMBER memberName [(READ getFunction | WRITE setFunction)])

[RESET resetFunction]

[NOTIFY notifySignal]

...

其中type表示属性的类型，它可以是QVariant所支持的类型或者是用户自定义的类型。而如果是枚举类型，还需要使用Q_ENUMS()宏在元对象系统中进行注册，这样以后才可以使用QObject::setProperty()函数来使用该属性。name就是属性的名称。READ后面是读取该属性的函数，这个函数是必须有的，而后面带有“[ ]”号的选项表示这些函数是可选的。

一个属性类似于一个数据成员，不过添加了一些可以通过元对象系统访问的附加功能：

- 一个读（READ）操作函数。如果MEMBER变量没有指定，那么该函数是必须有的，它用来读取属性的值。这个函数一般是const类型的，它的返回值类型必须是该属性的类型，或者是该属性类型的指针或者引用。例如，QWidget::focus是一个只读属性，其READ函数是QWidget::hasFocus()。
- 一个可选的写（WRITE）操作函数。它用来设置属性的值。这个函数必须只有一个参数，而且它的返回值必须为空void。例如，QWidget::enabled的WRITE函数是QWidget::setEnabled()。
- 如果没有指定READ操作函数，那么必须指定一个MEMBER变量关联，这样会使给定的成员变量变为可读写的而不用创建READ和WRITE操作函数。
- 一个可选的重置（RESET）函数。它用来将属性恢复到一个默认的值。这个函数不能有参数，而且返回值必须为空void。例如，QWidget::cursor的RESET函数是QWidget::unsetCursor()。
- 一个可选的通知（NOTIFY）信号。如果使用该选项，那么需要指定类中一个已经存在的信号，每当该属性的值改变时都会发射该信号。如果使用MEMBER变量时指定NOTIFY信号，那么信号最多只能有一个参数，并且参数的类型必须与属性的类型相同。

- 一个可选的版本（REVISION）号。如果包含了该版本号，它会定义属性及其通知信号只用于特定版本的API（通常暴露给QML），如果不包含，则默认为0。
- 可选的DESIGNABLE表明这个属性在GUI设计器（例如Qt Designer）的属性编辑器中是否可见。大多数属性的该值为true，即可见。
- 可选的SCRIPTABLE表明这个属性是否可以被脚本引擎（scripting engine）访问，默认值为true。
- 可选的STORED表明是否在当对象的状态被存储时也必须存储这个属性的值，大部分属性的该值为true。
- 可选的USER表明这个属性是否被设计为该类的面向用户或者用户可编辑的属性。一般，每一个类中只有一个USER属性，它的默认值为false。例如，QAbstractButton::checked是按钮的用户可编辑属性。
- 可选的CONSTANT表明这个属性的值是一个常量。对于给定的一个对象实例，每一次使用常量属性的READ方法都必须返回相同的值，但对于类的不同的实例，这个常量可以不同。一个常量属性不可以有WRITE方法和NOTIFY信号。
- 可选的FINAL表明这个属性不能被派生类重写。

其中的READ，WRITE和RESET函数可以被继承，也可以是虚的（virtual），当在多继承时，它们必须继承自第一个父类。

自定义属性示例

```
class MyClass : public QObject
{
   Q_OBJECT
   Q_PROPERTY(QString userName READ getUserName WRITE setUserName
         NOTIFY userNameChanged); // 注册属性userName
public:
   explicit MyClass(QObject *parent = 0);
   QString getUserName() const       // 实现READ读函数
   {return m_userName;}
   void setUserName(QString userName)  // 实现WRITE写函数
   {
      m_userName = userName;
      emit userNameChanged(userName); // 当属性值改变时发射该信号
   }
signals:
   void userNameChanged(QString);     // 声明NOTIFY通知消息
private:
   QString m_userName;              // 私有变量，存放userName属性的值
};
```

使用自定义的属性：

```
MyClass *my = new MyClass(this);          // 创建MyClass类实例

connect(my,SIGNAL(userNameChanged(QString)),this,
        SLOT(userChanged(QString)));

my->setUserName(“software”);              // 设置属性的值

qDebug() << “userName:” << my->getUserName(); // 输出属性的值

// 使用QObject类的setProperty()函数设置属性的值
my->setProperty("userName","linux");

// 输出属性的值，这里使用了QObject类的property()函数，它返回值类型为QVariant
qDebug() << "userName:" << my->property("userName").toString();
```

对象树和拥有权

Qt中使用对象树（object tree）来组织和管理所有的QObject类及其子类的对象。当创建一个QObject时，如果使用了其他的对象作为其父对象（parent），那么这个QObject就会被添加到父对象的children()列表中，这样当父对象被销毁时，这个QObject也会被销毁。实践表明，这个机制非常适合于管理GUI对象。例如，一个QShortcut（键盘快捷键）对象是相应窗口的一个子对象，所以当用户关闭了这个窗口时，这个快捷键也可以被销毁。

QWidget作为能够在屏幕上显示的所有部件的基类，扩展了对象间的父子关系。一个子对象一般也就是一个子部件，因为它们要显示在父部件的区域之中。例如，当关闭一个消息对话框（message box）后要销毁它时，消息对话框中的按钮和标签也会被销毁，这也正是我们所希望的，因为按钮和标签是消息对话框的子部件。当然，也可以自己来销毁一个子对象。

示例

- 自定义继承自QPushButton的MyButton 类，添加析构函数的声明：

```
~MyButton();
```

- 定义析构函数：

```
MyButton::~MyButton()
{
   qDebug() << "delete button";
}
```

这样当MyButton的对象被销毁时，就会输出相应的信息。

- 在主窗口Widget类的构造函数中创建自定义的按钮部件：

```
MyButton *button = new MyButton(this);   // 创建按钮部件，指定widget为父部件
button->setText(tr("button"));
```

- 更改Widget类的析构函数：

```
Widget::~Widget(){
delete ui;
qDebug() << "delete widget";
}
```

当Widget窗口被销毁时，将输出信息。

运行程序，然后关闭窗口，在Qt Creator的应用程序输出栏中的输出信息为：

delete widget

delete button

可以看到，当关闭窗口后，因为该窗口是顶层窗口，所以应用程序要销毁该窗口部件（如果不是顶层窗口，那么关闭时只是隐藏，不会被销毁），而当窗口部件销毁时会自动销毁其子部件。这也就是为什么在Qt中经常只看到new操作而看不到delete操作的原因。

在main.cpp文件，其中Widget对象是建立在栈上的：

Widget w;

w.show();

这样对于对象w，在关闭程序时会被自动销毁。而对于Widget中的部件，如果是在堆上创建（使用new操作符），那么只要指定Widget为其父窗口就可以了，也不需要进行delete操作。整个应用程序关闭时，会去销毁w对象，而此时又会自动销毁它的所有子部件，这些都是Qt的对象树所完成的。

所以，对于规范的Qt程序，要在main()函数中将主窗口部件创建在栈上，例如“Widget w;”，而不要在堆上进行创建（使用new操作符）。对于其他窗口部件，可以使用new操作符在堆上进行创建，不过一定要指定其父部件，这样就不需要再使用delete操作符来销毁该对象了。

元对象系统

Qt中的元对象系统（Meta-Object System）提供了对象间通信的信号和槽机制、运行时类型信息和动态属性系统。元对象系统是基于以下三个条件的：

- 该类必须继承自QObject类；
- 必须在类的私有声明区声明Q_OBJECT宏（在类定义时，如果没有指定public或者private，则默认为private）；
- 元对象编译器Meta-Object Compiler（moc），为QObject的子类实现元对象特性提供必要的代码。

其中moc工具读取一个C++源文件，如果它发现一个或者多个类的声明中包含有Q_OBJECT宏，便会另外创建一个C++源文件（就是在项目目录中的debug目录下看到的以moc开头的C++源文件），其中包含了为每一个类生成的元对象代码。这些产生的源文件或者被包含进类的源文件中，或者和类的实现同时进行编译和链接。

元对象系统主要是为了实现信号和槽机制才被引入的，不过除了信号和槽机制以外，元对象系统还提供了其他一些特性：

- QObject::metaObject()函数可以返回一个类的元对象，它是QMetaObject类的对象；
- QMetaObject::className()可以在运行时以字符串形式返回类名，而不需要C++编辑器原生的运行时类型信息（RTTI）的支持；
- QObject::inherits()函数返回一个对象是否是QObject继承树上一个类的实例的信息；
- QObject::tr()和QObject::trUtf8()进行字符串翻译来实现国际化；
- QObject::setProperty()和QObject::property()通过名字来动态设置或者获取对象属性；
- QMetaObject::newInstance()构造该类的一个新实例。

正则表达式

正则表达式（regular expression），就是在一个文本中匹配子字符串的一种模式（pattern），它可以简写为“regexp”，。一个regexp主要应用在以下几个方面：

- 验证。一个regexp可以测试一个子字符串是否符合一些标准。例如，是一个整数或者不包含任何空格等。
- 搜索。一个regexp提供了比简单的子字符串匹配更强大的模式匹配。例如，匹配单词mail或者letter，而不匹配单词email或者letterbox。
- 查找和替换。一个regexp可以使用一个不同的字符串替换一个字符串中所有要替换的子字符串。例如，使用Mail来替换一个字符串中所有的M字符，但是如果M字符后面有ail时不进行替换。
- 字符串分割。一个regexp可以识别在哪里进行字符串分割。例如，分割制表符隔离的字符串。

Qt中的QRegExp类实现了使用正则表达式进行模式匹配。QRegExp是以Perl的正则表达式语言为蓝本的，它完全支持Unicode。QRegExp中的语法规则可以使用setPatternSyntax()函数来更改。

正则表达式简介

- Regexps由表达式（expressions）、量词（quantifiers）和断言（assertions）组成。最简单的一个表达式就是一个字符，例如x和5。而一组字符可以使用方括号括起来，例如[ABC]将会匹配一个A或者一个B或者一个C，这个也可以简写为[A-C]，这样我们要匹配所有的英文大写字母，就可以使用[A-Z]。
- 一个量词指定了必须要匹配的表达式出现的次数。例如，x{1,1}意味着必须匹配且只能匹配一个字符x，而x{1,5}意味着匹配一列字符x，其中至少要包含一个字符x，但是最多包含5个字符x。
- 现在假设我们要使用一个regexp来匹配0到99之间的整数。因为至少要有一个数字，所以我们使用表达式 [0-9]{1,1} 开始，它匹配一个单一的数字一次。要匹配0-99，我们可以想到将表达式最多出现的次数设置为2，即 [0-9]{1,2} 。现在这个regexp已经可以满足我们假设的需要了，不过，它也会匹配出现在字符串中间的整数。如果想匹配的整数是整个字符串，那么就需要使用断言“^”和“$”，当 ^ 在regexp中作为第一个字符时，意味着这个regexp必须从字符串的开始进行匹配；当 $ 在regexp中作为最后一个字符时，意味着regexp必须匹配到字符串的结尾。所以，最终的regexp为 ^[0-9]{1,2}$ 。
- 一般可以使用一些特殊的符号来表示一些常见的字符组和量词。例如，[0-9] 可以使用 \d 来替代。而对于只出现一次的量词 {1,1} ，可以使用表达式本身代替，例如x{1,1}等价于x 。所以要匹配0-99，就可以写为 ^\d{1,2}$ 或者 ^\d\d{0,1}$ 。而 {0,1} 表示字符是可选的，就是只出现一次或者不出现，它可以使用 ？ 来代替，这样regexp就可以写为 ^\d\d?$ ，它意味着从字符串的开始，匹配一个数字，紧接着是0个或1个数字，再后面就是字符串的结尾。

- 现在我们写一个regexp来匹配单词“mail”或者“letter”其中的一个，但是不要匹配那些包含这些单词的单词，比如“email”和“letterbox”。要匹配“mail”，regexp可以写成m{1,1}a{1,1}i{1,1}l{1,1}，因为{1,1}可以省略，所以又可以简写成mail。下面就可以使用竖线“|”来包含另外一个单词，这里“|”表示“或”的意思。为了避免regexp匹配多余的单词，必须让它从单词的边界进行匹配。首先，将regexp用括号括起来，即(mail|letter)。括号将表达式组合在一起，可以在一个更复杂的regexp中作为一个组件来使用，这样也可以方便我们来检测到底是哪一个单词被匹配了。为了强制匹配的开始和结束都在单词的边界上，要将regexp包含在\b单词边界断言中，即\b(mail|letter)\b。这个\b断言在regexp中匹配一个位置，而不是一个字符，一个单词的边界是任何的非单词字符，如一个空格，新行，或者一个字符串的开始或者结束。
- 如果想使用一个单词，例如“Mail”，替换一个字符串中的字符M，但是当字符M的后面是“ail”的话就不再替换。这样我们可以使用(?!E)断言，例如这里regexp应该写成M(?!Mail)。
- 如果想统计“Eric”和“Eirik”在字符串中出现的次数，可以使用\b(Eric|Eirik)\b或者\bEi?ri[ck]\b。这里需要使用单词边界断言‘\b’来避免匹配那些包含了这些名字的单词。

表达式

元素	含义
c	一个字符代表它本身，除非这个字符有特殊的 regexp 含义。例如，c 匹配字符 c
\c	跟在反斜杠后面的字符匹配字符本身，但是本表中下面指定的这些字符除外。例如，要匹配一个字符串的开头，使用\^
\a	匹配 ASCII 的振铃（BEL,0x07）
\f	匹配 ASCII 的换页（FF,0x0C）
\n	匹配 ASCII 的换行（LF,0x0A）
\r	匹配 ASCII 的回车（CR,0x0D）
\t	匹配 ASCII 的水平制表符（HT,0x09）
\v	匹配 ASCII 的垂直制表符（VT,0x0B）
\xhhhh	匹配 Unicode 字符对应的十六进制数 hhhh（在 0x0000 到 0xFFFF 之间）
\0ooo	匹配八进制的 ASCII/Latin1 字符 ooo（在 0 到 0377 之间）
.(点)	匹配任意字符（包括新行）
\d	匹配一个数字
\D	匹配一个非数字
\s	匹配一个空白字符，包括‘\t’、‘\n’、‘\v’、‘\f’、‘\r’和‘ ’
\S	匹配一个非空白字符
\w	匹配一个单词字符，包括任意一个字母或数字或下划线，即 A~Z,a~z,0~9,_ 中任意一个
\W	匹配一个非单词字符
\n	第 n 个反向引用，例如，\1，\2 等

量词

默认的，一个表达式将自动量化为{1,1}，就是说它应该出现一次。在下表中列出了量词的使用情况，其中E代表一个表达式，一个表达式可以是一个字符，或者一个字符集的缩写，或者在方括号中的一个字符集，或者在括号中的一个表达式。

量词	含义
E?	匹配 0 次或者 1 次，表明 E 是可选的，E?等价于 E{0,1}
E+	匹配 1 次或者多次，E+等价于 E{1,}，例如，0+ 匹配“0”，“00”，“000”等
E*	匹配 0 次或者多次，等价于 E{0,}
E{n}	匹配 n 次，等价于 E{n,n}，例如，x{5}等价于 x{5,5}，也等价于 xxxxx
E{n,}	匹配至少 n 次
E{,m}	匹配至多 m 次，等价于 E{0,m}
E{n,m}	匹配至少 n 次，至多 m 次

断言

断言在regexp中作出一些有关文本的声明，它们不匹配任何字符。正则表达式中的断言如下表所示，其中E代表一个表达式。

断言	含义
^	标志着字符串的开始。如果要匹配 ^ 就要使用\\^
$	标志着字符串的结尾。如果要匹配 $ 就要使用\\$
\b	一个单词的边界
\B	一个非单词的边界，当\b 为 false 时它为 true
(?=E)	表达式后面紧跟着 E 才匹配。例如，const(?=\s+char)匹配“const”且其后必须有“char”
(?!E)	表达式后面没有紧跟着 E 才匹配。例如，const(?!\s+char)匹配“const”但其后不能有“char”

通配符

QRegExp类还支持通配符（Wildcard）匹配。很多命令shell（例如bash和cmd.exe）都支持文件通配符（file globbing），可以使用通配符来识别一组文件。QRegExp的setPatternSyntax()函数就是用来在regexp和通配符之间进行切换的。通配符匹配要比regexp简单很多，它只有四个特点，如下表所示。

字符	含义
c	任意一个字符，表示字符本身
?	匹配任意一个字符，类似于 regexp 中的“.”
*	匹配 0 个或者多个任意的字符，类似于 regexp 中的“.*”
[...]	在方括号中的字符集，与 regexp 中的类似

qDebug() << rx3.exactMatch("welcome.txt.bak"); // 结果为false

文本捕获

- 在regexp中使用括号可以使一些元素组合在一起，这样既可以对它们进行量化，也可以捕获它们。例如，使用表达式 mail|letter 来匹配一个字符串，我们知道了有一个单词被匹配了，但是却不能知道具体是哪一个，使用括号就可以让我们捕获被匹配的那个单词，比如使用 (mail|letter) 来匹配字符串“I Sent you some email”，这样就可以使用cap()或者capturedTexts()函数来提取匹配的字符。
- 还可以在regexp中使用捕获到的文本，为了表示捕获到的文本，使用反向引用 \n ，其中n从1开始编号，比如 \1 就表示前面第一个捕获到的文本。例如，使用\b(\w+)\W+\1\b 在一个字符串中查询重复出现的单词，这意味着先匹配一个单词边界，随后是一个或者多个单词字符，随后是一个或者多个非单词字符，随后是与前面第一个括号中相同的文本，随后是单词边界。
- 如果使用括号仅仅是为了组合元素而不是为了捕获文本，那么可以使用非捕获语法，例如 (?:green|blue) 。非捕获括号由“(?:”开始，由“)”结束。使用非捕获括号比使用捕获括号更高效，因为regexp引擎只需做较少的工作。

小结

本章中学习了Qt的一些核心内容，比如信号和槽、元对象系统等；也学习了正则表达式的相关知识。这些知识理论性比较强，都是非常重要的内容，要在平时编程的过程中多使用相关知识点，逐渐掌握。