Qt程序设计方法

Qt对象模型与容器类

School of Computer Science
and Engineering

2024-2025-暑期课程

Qt对象模型与容器类

主 要 内 容

- 对象模型
- 容器类
- 正则表达式

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：

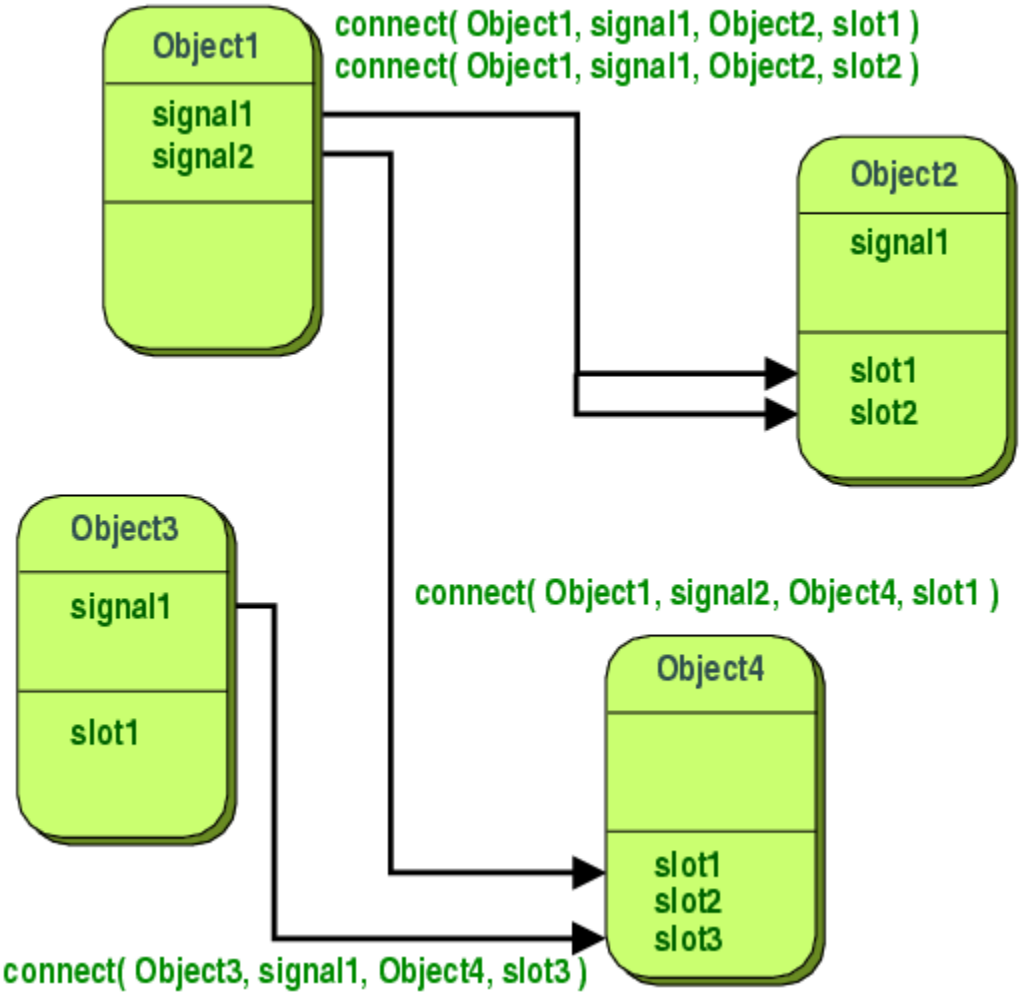
- 一个强大的无缝对象通信机制——信号和槽（signals and slots）；
- 可查询和可设计的对象属性系统（object properties）；
- 强大的事件和事件过滤器（events and event filters）；
- 通过上下文进行国际化的字符串翻译机制（string translation for internationalization）；
- 完善的定时器（timers）驱动，使得可以在一个事件驱动的GUI中处理多个任务；
- 分层结构的、可查询的对象树（object trees），它使用一种很自然的方式来组织对象拥有权（object ownership）；
- 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
不同于其他
总希望其他
和其他对象
窗口的clos
了回调（ca
当一个特殊
一个函数，
定义了一些
信号和槽来

- 一个信号可
一个信号还
那么，当这
联顺序相同



特征，也是Qt
了一个部件时，
何对象都可以
希望可以执行
些工具包中使用
行对象间的通信。
击；而槽就是
的部件类中已经
后添加自己的

槽上，甚至，
信号相关联，
执行顺序与关

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。

自学：

如果不使用设计模式生成，自定义的信号和槽应该如何使用自动关联，需要注意哪些事项？

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]
           [REVISION int]
           [DESIGNABLE bool]
           [SCRIPTABLE bool]
           [STORED bool]
           [USER bool]
           [CONSTANT]
           [FINAL])
```

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(“yafei”);                   // 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。

7.2 容器类

Qt库提供了一组通用的基于模板的容器类（container classes）。这些容器类可以用来存储指定类型的项目（items），例如，如果大家需要一个QString类型的可变大小的数组，那么可以使用QVector(QString)。与STL(Standard Template Library，C++的标准模板库)中的容器类相比，Qt中的这些容器类更轻量，更安全，更容易使用。

- Qt的容器类简介
- 遍历容器
- 通用算法
- QString
- QByteArray和QVariant

Qt的容器类简介

QSet<T>	它提供了一个快速查询单值的数学集。
QMap<Key,T>	它提供了一个字典（关联数组），将 Key 类型的键值映射到 T 类型的值上。一般每一个键关联一个单一的值。QMap 使用键顺序来存储它的数据；如果不关心存储顺序，那么可以使用 QHash 来代替它，因为 QHash 更快速。
QMultiMap<Key,T>	它是 QMap 的一个便捷类，提供了实现多值映射的方便的接口函数，例如一个键可以关联多个值。
QHash<Key,T>	它与 QMap 拥有基本相同的接口，但是它的查找速度更快。QHash 的数据是以任意的顺序存储的。
QMultiHash<Key,T>	它是 QHash 的一个便捷类，提供了实现多值散列的方便的接口函数。
	进行了插入或者删除操作，那么这个迭代器就无效了。）
QVector<T>	它在内存的相邻位置存储给定类型的值的一个数组。在 vector 的前面或者中间插入项目是非常缓慢的，因为这样可能导致大量的项目要在内存中移动一个位置。
QStack<T>	它是 QVector 的一个便捷子类，提供了后进先出（LIFO）语义。它添加了 push()，pop()和 top()等函数。
QQueue<T>	它是 QList 的一个便捷子类，提供了先进先出（FIFO）语义。它添加了 enqueue()，dequeue()和 head()等函数。

QList是一个模板类，它提供了一个列表。QList<T>实际上是一个T类型项目的指针数组，所以它支持基于索引的访问，而且当项目的数目小于1000时，可以实现在列表中间进行快速的插入操作。QList提供了很多方便的接口函数来操作列表中的项目，例如：

- 插入操作insert();
- 替换操作replace();
- 移除操作removeAt();
- 移动操作move();
- 交换操作swap();
- 在表尾添加项目append();
- 在表头添加项目prepend();
- 移除第一个项目removeFirst();
- 移除最后一个项目removeLast();
- 从列表中移除一项并获取这个项目takeAt()，还有相应的takeFirst()和takeLast();
- 获取一个项目的索引indexOf();
- 判断是否含有相应的项目contains();
- 获取一个项目出现的次数count()。

对于QList，可以使用“<<”操作符来向列表中插入项目，也可以使用“[ ]”操作符通过索引来访问一个项目，其中项目是从0开始编号的。不过，对于只读的访问，另一种方法是使用at()函数，它比“[ ]”操作符要快很多。

例如：

```
QList<QString> list;
list << "aa" << "bb" << "cc"; // 插入项目
if(list[1] == "bb") list[1] = "ab";
list.replace(2,"bc");        // 将“cc”换为“bc”
qDebug() << "the list is: ";  // 输出整个列表
for(int i=0; i<list.size(); ++i){
    qDebug() << list.at(i);   // 现在列表为aa ab bc
}
list.append("dd");            // 在列表尾部添加
list.prepend("mm");           // 在列表头部添加
QString str = list.takeAt(2); // 从列表中删除第3个项目，并获取它
qDebug() << "at(2) item is: " << str;
qDebug() << "the list is: ";
for(int i=0; i<list.size(); ++i)
{
    qDebug() << list.at(i);   // 现在列表为mm aa bc dd
}
```

QMap类是一个容器类，它提供了一个基于跳跃列表的字典（a skip-list-based dictionary）。QMap<Key,T>是Qt的通用容器类之一，它存储（键，值）对并提供了与键相关的值的快速查找。QMap中提供了很多方便的接口函数，例如：

- 插入操作insert();
- 获取值value();
- 是否包含一个键contains();
- 删除一个键remove();
- 删除一个键并获取该键对应的值take();
- 清空操作clear();
- 插入一键多值insertMulti()。

可以使用“[ ]”操作符插入一个键值对或者获取一个键的值，不过当使用该操作符获取一个不存在的键的值时，会默认向map中插入该键，为了避免这个情况，可以使用value()函数来获取键的值。当使用value()函数时，如果指定的键不存在，那么默认会返回0，可以在使用该函数时提供参数来更改这个默认返回的值。QMap默认是一个键对应一个值的，但是也可以使用insertMulti()进行一键多值的插入，对于一键多值的情况，更方便的是使用QMap的子类QMultiMap。

例如：

```
QMap<QString,int> map;
 map["one"] = 1;          // 向map中插入("one",1)
 map["three"] = 3;
map.insert("seven",7);  // 使用insert()函数进行插入
 // 获取键的值，使用“[ ]”操作符时，如果map中没有该键，那么会自动插入
 int value1 = map["six"];
 qDebug() << "value1:" << value1;
qDebug() << "contains 'six' ?" << map.contains("six");
 // 使用value()函数获取键的值，这样当键不存在时不会自动插入
 int value2 = map.value("five");
 qDebug() << "value2:" << value2;
qDebug() << "contains 'five' ?" << map.contains("five");
 // 当键不存在时，value()默认返回0，这里可以设定该值，比如这里设置为9
 int value3 = map.value("nine",9);
 qDebug() << "value3:" << value3;
```

嵌套和赋值

- 容器也可以嵌套使用，例如QMap<QString,QList<int> >，这里键的类型是QString，而值的类型是QList<int>，需要注意，在后面的“> >”符号之间要有一个空格，不然编译器会将它当做“>>”操作符对待。

- 在各种容器中所存储的值的类型可以是任何的可赋值的数据类型，该类型需要有一个默认的构造函数，一个拷贝构造函数和一个赋值操作运算符，像基本的类型double，指针类型，Qt的数据类型如QString、QDate、QTime等。但是QObject以及QObject的子类都不能存储在容器中，不过，可以存储这些类的指针，例如QList<QWidget*>。

遍历容器

遍历一个容器可以使用迭代器（iterators）来完成，迭代器提供了一个统一的方法来访问容器中的项目。Qt的容器类提供了两种类型的迭代器：

- Java风格迭代器
- STL风格迭代器

如果只是想按顺序遍历一个容器中的项目，那么还可以使用Qt的：

- foreach关键字

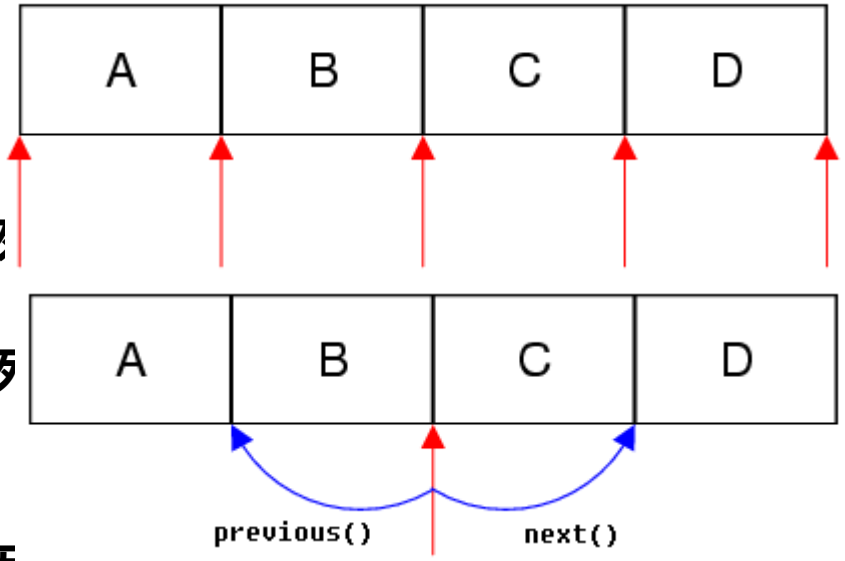
Java风格迭代器

Java风格迭代器在使用时比STL风格迭代器要方便很多，但是在性能上稍微弱于后者。对于每一个容器类，都有两个Java风格迭代器数据类型：一个提供只读访问，一个提供读写访问。

容器	只读迭代器	读写迭代器
QList<T>, QQueue<T>	QListIterator<T>	QMutableListIterator<T>
QLinkedList<T>	QLinkedListIterator<T>	QMutableLinkedListIterator<T>
QVector<T>, QStack<T>	QVectorIterator<T>	QMutableVectorIterator<T>
QSet<T>	QSetIterator<T>	QMutableSetIterator<T>
QMap<Key, T>, QMultiMap<Key, T>	QMapIterator<Key, T>	QMutableMapIterator<Key, T>
QHash<Key, T>, QMultiHash<Key, T>	QHashIterator<Key, T>	QMutableHashIterator<Key, T>

QList示例：

```
QList<QString> list;
list << "A" << "B" << "C" << "D";
QListIterator<QString> i(list); // 创建
qDebug() << "the forward is :";
while (i.hasNext())              // 正向遍历
    qDebug() << i.next();
qDebug() << "the backward is :";
while (i.hasPrevious())          // 反向遍历列表，结果为D，C，B，A
    qDebug() << i.previous();
```



这里先创建了一个QList列表list，然后使用list作为参数创建了一个列表的只读迭代器。这时，迭代器指向列表的第一个项目的前面（这里是指向项目“A”的前面）。然后使用hasNext()函数来检查在该迭代器后面是否还有项目，如果还有项目，那么使用next()来跳过这个项目，next()函数会返回它所跳过的项目。当正向遍历结束后，迭代器会指向列表最后一个项目的后面，这时可以使用hasPrevious()和previous()来进行反向遍历。可以看到，Java风格迭代器是指向项目之间的，而不是直接指向项目。所以，迭代器或者指向容器的最前面，或者指向两个项目之间，或者指向容器的最后面。

QMap示例：

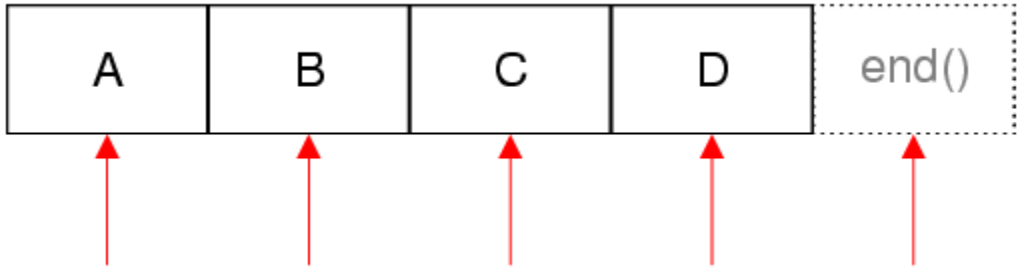
```
QMap<QString, QString> map;
map.insert("Paris", "France");
map.insert("Guatemala City", "Guatemala");
map.insert("Mexico City", "Mexico");
map.insert("Moscow", "Russia");
QMapIterator<QString,QString> i(map);
while(i.hasNext()) {
     i.next();
    qDebug() << i.key() << " : " << i.value();
}
 if(i.findPrevious("Mexico"))
        qDebug() << “find ‘Mexico’”；// 向前查找键的值
```

这里在QMap中存储了一些（首都，国家）键值对，然后删除了包含以“City”字符串结尾的键的项目。对于QMap的遍历，可以先使用next()函数，然后再使用key()和value()来获取键和值的信息。

STL风格迭代器

STL风格迭代器兼容Qt和STL的通用算法（generic algorithms），而且在速度上进行了优化。对于每一个容器类，都有两个STL风格迭代器类型：一个提供了只读访问，另一个提供了读写访问。因为只读迭代器比读写迭代器要快很多，所以应尽可能使用只读迭代器。

容器	只读迭代器	读写迭代器
QList<T>, QQueue<T>	QList<T>::const_iterator	QList<T>::iterator
QLinkedList<T>	QLinkedList<T>::const_iterator	QLinkedList<T>::iterator
QVector<T>, QStack<T>	QVector<T>::const_iterator	QVector<T>::iterator
QSet<T>	QSet<T>::const_iterator	QSet<T>::iterator
QMap<Key, T>, QMultiMap<Key, T>	QMap<Key, T>::const_iterator	QMap<Key, T>::iterator
QHash<Key, T>, QMultiHash<Key, T>	QHash<Key, T>::const_iterator	QHash<Key, T>::iterator



QList示例：

```
QList<QString> list;
list << "A" << "B" << "C" << "D";
QList<QString>::iterator i;      // 使用读写迭代器
qDebug() << "the forward is :";
for (i = list.begin(); i != list.end(); ++i) {
    *i = (*i).toLower();         // 使用QString的toLower()函数转换为小写
    qDebug() << *i;              // 结果为a，b，c，d
}
qDebug() << "the backward is :";
while (i != list.begin()) {
    --i;
    qDebug() << *i;              // 结果为d，c，b，a
}
QList<QString>::const_iterator j; // 使用只读迭代器
qDebug() << "the forward is :";
for (j = list.constBegin(); j != list.constEnd(); ++j)
    qDebug() << *j;              // 结果为a，b，c，d
```

STL风格迭代器的API模仿了数组的指针，例如，使用“++”操作符来向后移动迭代器使其指向下一个项目；使用“*”操作符返回迭代器指向的项目等。需要说明的是，不同于Java风格迭代器，STL风格迭代器是直接指向项目的。其中一个容器的begin()函数返回了一个指向该容器中第一个项目的迭代器，end()函数也返回一个迭代器，但是这个迭代器指向该容器的最后一个项目的下一个假想的虚项目，end()标志着一个无效的位置，当列表为空时，begin()函数等价于end()函数。

在STL风格迭代器中“++”和“--”操作符即可以作为前缀（++i，--i）操作符，也可以作为后缀（i++，i--）操作符。当作为前缀时会先修改迭代器，然后返回修改后的迭代器的一个引用；当作为后缀时，在修改迭代器以前会对其进行复制，然后返回这个复制。如果在表达式中不会对返回值进行处理，那么最好使用前缀操作符（++i，--i），这样会更快一些。对于非const迭代器类型，使用一元操作符“*”获得的返回值可以用在赋值运算符的左侧。STL风格迭代器的常用API如下表所示。

表达式	行为
*i	返回当前项目
++i	前移迭代器到下一个项目
i += n	使迭代器前移 n 个项目
--i	使迭代器往回移动一个项目
i -= n	使迭代器往回移动 n 个项目
i - j	返回迭代器 i 和迭代器 j 之间的项目的数目

QMap示例：

```
QMap<QString, int> map;
map.insert("one",1);
map.insert("two",2);
map.insert("three",3);
QMap<QString, int>::const_iterator p;
qDebug() << "the forward is :";
for (p = map.constBegin(); p != map.constEnd(); ++p)
    qDebug() << p.key() << ":" << p.value();// 结果为
    (one,1),(three,3),(two,2)
```

这里创建了一个QMap，然后使用STL风格的只读迭代器对其进行了遍历，输出了其中所有项目的键和值 。

Foreach关键字

foreach是Qt向C++语言中添加的一个用来进行容器的顺序遍历的关键字，它使用预处理器来进行实施。 例如：

```
QList<QString> list;
 list.insert(0, "A");
 list.insert(1, "B");
 list.insert(2, "C");
 qDebug() <<"the list is :";
 foreach (QString str,list) {    // 从list中获取每一项
    qDebug() << str;            // 结果为A，B，C
 }

 QMap<QString,int> map;
 map.insert("first", 1);
 map.insert("second", 2);
 map.insert("third", 3);
 qDebug() << endl << "the map is :";
 foreach (QString str, map.keys())   // 从map中获取每一个键
    qDebug() << str << " : " << map.value(str);
 // 输出键和对应的值，结果为(first,1),(second,2),(third,3)
```

通用算法

在<QtAlgorithms>头文件中，Qt提供了一些全局的模板函数，这些函数是可以使用在容器上的十分常用的算法。我们可以在任何提供了STL风格迭代器的容器类上使用这些算法，包括QList、QLinkedList、QVector、QMap和QHash。

如果在目标平台上可以使用STL，那么可以使用STL的算法来代替Qt的这些算法，因为STL提供了更多的算法，而Qt只提供了其中最重要的一些算法。

示例：

```
QStringList list;
 list << "one" << "two" << "three";
 qDebug() << QObject::tr("qCopy()算法：");
QVector<QString> vect(3);
// 将list中所有项目复制到vect中
 std:copy(list.begin(), list.end(), vect.begin());
qDebug() << vect; //结果为one,two,three

qDebug() << endl << QObject::tr("qEqual()算法：");
// 从list的开始到结束的所有项目与vect的开始及其后面的等数量的项目进行比较，
// 全部相同则返回true
bool ret1 = std:equal(list.begin(), list.end(), vect.begin());
qDebug() << "euqal: " << ret1; //结果为true

qDebug() << endl << QObject::tr("qFind()算法：");
// 从list中查找"two",返回第一个对应的值的迭代器，如果没有找到则返回end()
QList<QString>::iterator i = std:find(list.begin(), list.end(), "two");
qDebug() << *i;     // 结果为"two"
```

QString

- QString类提供了一个Unicode（Unicode是一种支持大部分文字系统的国际字符编码标准）字符串。其实在第一个Hello World程序就用到了它，而几乎所有的程序中都会使用到它。

- QString存储了一串QChar，而QChar提供了一个16位的Unicode 4.0字符。在后台，QString使用隐式共享（implicit sharing）来减少内存使用和避免不必要的数据拷贝，这也有助于减少存储16位字符的固有开销。

隐式共享

- 隐式共享（Implicit Sharing）又称为写时复制（copy-on-write）。Qt中很多C++类使用隐式数据共享来尽可能的提高资源使用率和尽可能的减少复制操作。使用隐式共享类作为参数传递是既安全又有效的，因为只有一个指向该数据的指针被传递了，只有当函数向它写入时才会复制该数据。

- 共享的好处是程序不需要进行不必要的数据复制，这样可以减少数据的拷贝和使用更少的内存，对象也可以很容易地被分配，或者作为参数被传递，或者从函数被返回。隐式共享在后台进行，在实际编程中我们不必去关注它。Qt中主要的隐式共享类有：QByteArray、QCursor、QFont、QPixmap、QString、QUrl、QVariant和所有的容器类等等 。

例如：

```
QPixmap p1, p2;
p1.load("image.bmp");
p2 = p1;                        // p1与p2共享数据
QPainter paint;
paint.begin(&p2);               // p2被修改
paint.drawText(0,50, "Hi");
paint.end();
```

当处理共享对象时，有两种复制对象的方法：深拷贝（deep copy）和浅拷贝（shallow copy）。

- 深拷贝意味着复制一个对象，而浅拷贝则是复制一个引用（仅仅是一个指向共享数据块的指针）。一个深拷贝是非常昂贵的，需要消耗很多的内存和CPU资源；而浅拷贝则非常快速，因为它只需要设置一个指针和增加引用计数的值。
- 当隐式共享类使用“=”操作符时就是使用浅拷贝，如上面的“p2 = p1；”语句。但是当一个对象被修改时，就必须进行一次深拷贝，比如上面程序中“paint.begin(&p2);”语句要对p2进行修改，这时就要对数据进行深拷贝，使p2和p1指向不同的数据结构，然后将p1的引用计数设为1，p2的引用计数也设为1。

编辑操作

在QString中提供了多个方便的函数来操作字符串，例如：

- append()和prepend()分别实现了在字符串后面和前面添加字符串或者字符；replace()替换指定位置的多个字符；
- insert()在指定位置添加字符串或者字符；
- remove()在指定位置移除多个字符；
- trimmed()除去字符串两端的空白字符，这包括‘\t’、‘\n’、‘\v’、‘\f’、‘\r’和‘ ’；
- simplified()不仅除去字符串两端的空白字符，还将字符串中间的空白字符序列替换为一个空格；
- split()可以将一个字符串分割为多个子字符串的列表等等。

对于一个字符串，也可以使用“[ ]”操作符来获取或者修改其中的一个字符，还可以使用“+”操作符来组合两个字符串。在QString类中一个null字符串和一个空字符串并不是完全一样的。一个null字符串是使用QString的默认构造函数或者在构造函数中传递了0来初始化的字符串；而一个空字符串是指大小为0的字符串。一般null字符串都是空字符串，但一个空字符串不一定是一个null字符串，在实际编程中一般使用isEmpty()来判断一个字符串是否为空。

示例：

```
QString str = "hello";
qDebug() << QObject::tr("字符串大小：") << str.size(); // 大小为5
str[0] = QChar('H');     // 将第一个字符换为‘H'
qDebug() << QObject::tr("第一个字符：") << str[0]; // 结果为‘H'
str.append(" Qt");       // 向字符串后添加"Qt"
str.replace(1,4,"i");    // 将第1个字符开始的后面4个字符替换为字符串"i"
str.insert(2," my");     // 在第2个字符后插入" my"
qDebug() << QObject::tr("str为：") << str; // 结果为Hi my Qt
str = str + "!!!";       // 将两个字符串组合
qDebug() << QObject::tr(“str为：”) << str; // 结果为Hi my Qt！！！

str = " hi\r\n Qt!\n  ";
qDebug() << QObject::tr("str为：") << str;
QString str1 = str.trimmed();   // 除去字符串两端的空白字符
qDebug() << QObject::tr("str1为：") << str1;
QString str2 = str.simplified(); // 除去字符串两端和中间多余的空白字符
qDebug() << QObject::tr("str2为：") << str2; //结果为hi Qt!
```

查询操作

- 在QString中提供了right()、left()和mid()函数分别来提取一个字符串的最右面，最左面和中间的含有多个字符的子字符串；
- 使用indexOf()函数来获取一个字符或者子字符串在该字符串中的位置；
- 使用at()函数可以获取一个指定位置的字符，它比“[ ]”操作符要快很多，因为它不会引起深拷贝；
- 可以使用contains()函数来判断该字符串是否包含一个指定的字符或者字符串；
- 可以使用count()来获得字符串中一个字符或者子字符串出现的次数；
- 而使用startsWith()和endsWidth()函数可以用来判断该字符串是否是以一个字符或者字符串开始或者结束的；
- 对于两个字符串的比较，可以使用“>”和“<=”等操作符，也可以使用compare()函数。

示例：

```
qDebug() << endl << QObject::tr("以下是在字符串中进行查询的操作：") <<endl;

str = "yafeilinux";
qDebug() << QObject::tr("字符串为：") << str;

// 执行下面一行代码后，结果为linux
qDebug() << QObject::tr("包含右侧5个字符的子字符串：") << str.right(5);

// 执行下面一行代码后，结果为fei
qDebug() << QObject::tr("包含第2个字符以后3个字符的子字符串：") <<
    str.mid(2,3);

qDebug() << QObject::tr("'fei'的位置：") <<str.indexOf("fei"); //结果为2

qDebug() << QObject::tr("str的第0个字符：") << str.at(0); //结果为y

qDebug() << QObject::tr("str中'i'字符的个数：") << str.count('i'); //结果为2

// 执行下面一行代码后，结果为true
qDebug() << QObject::tr("str是否以”ya“开始？") << str.startsWith("ya");
```

转换操作

- QString中的toInt()、toDouble()等函数可以很方便的将字符串转换为整型或者double型数据，当转换成功后，它们的第一个bool型参数会为true；

- 使用静态函数number()可以将数值转换为字符串，这里还可以指定要转换为哪种进制；

- 使用toLower()和toUpper()函数可以分别返回字符串小写和大写形式的副本。

示例：

```
qDebug() << endl << QObject::tr("以下是字符串的转换操作：") << endl;

str = "100";
qDebug() << QObject::tr("字符串转换为整数：") << str.toInt(); // 结果为100

int num = 45;
qDebug() << QObject::tr("整数转换为字符串：") << QString::number(num);//结果为
    “45”

str = "FF";
bool ok;
int hex = str.toInt(&ok,16);
qDebug() << "ok: "<< ok << QObject::tr("转换为十六进制：") << hex; // 结果为ok:
    true 255

num = 26;
qDebug() << QObject::tr("使用十六进制将整数转换为字符串：")
         << QString::number(num,16);//结果为1a

str = "123.456";
qDebug() << QObject::tr("字符串转换为浮点型：") << str.toFloat();//结果为123.456

str = "abc";qDebug() << QObject::tr("转换为大写：") << str.toUpper();// 结果为ABC

str = "ABC";qDebug() << QObject::tr("转换为小写：") <<str.toLower();// 结果为
```

arg( )函数

示例：

```
int age = 25;
QString name = "yafei";

// name代替%1，age代替%2
str = QString("name is %1, age is %2").arg(name).arg(age);
// 结果为name is yafei,age is 25
qDebug() << QObject::tr("更改后的str为：") << str;

str = "%1 %2";
qDebug() << str.arg("%1f","hello");     // 结果为%1f hello
qDebug() << str.arg("%1f").arg("hello"); // 结果为hellof %2

str = QString("ni%1").arg("hi",5,'*');
qDebug() << QObject::tr("设置字段宽度为5，使用'*'填充：") << str;//结果为
    ni***hi

qreal value = 123.456;
str = QString("number: %1").arg(value,0,'f',2);
qDebug() << QObject::tr("设置小数点位数为两位：") << str;  //结果为
    "number:123.45
```

- arg()函数中的参数可以取代字符串中相应的“%1”等标记，在字符串中可以使用的标记在1到99之间，arg()函数会从最小的数字开始对应，比如QString(“%5,%2,%7”).arg(“a”).arg(“b”)，那么“a”会代替“%2”，“b”会代替“%5”，而“%7”会直接显示。

- arg()的一种重载形式是：
  arg (const QString & a1, const QString & a2)，
  它与使用str.arg(a1).arg(a2)是相同的，不过当参数a1中含有“%1”等标记时，两者的效果是不同的，这个可以在前面的程序中看到。

- 该函数的另一种重载形式为：
  arg (const QString & a, int fieldWidth = 0, const QChar & fillChar = QLatin1Char(‘ ’))，
  这里可以设定字段宽度，如果第一个参数a的宽度小于fieldWidth的值，那么就可以使用第三个参数设置的字符来进行填充。这里的fieldWidth如果为正值，那么文本是右对齐的，比如前面程序中的结果为“ni***hi”。而如果为负值，那么文本是左对齐的，例如将前面的程序中的fieldWidth改为-5，那么结果就应该是“nihi***”。

- arg()还有一种重载形式：
  arg (double a, int fieldWidth = 0, char format = ‘g’, int precision = -1,
  const QChar & fillChar = QLatin1Char(‘ ’))，
  它的第一个参数是double类型的，后面的format和precision分别可以指定其类型和精度。

QByteArray

- QByteArray类提供了一个字节数组，它可以用来存储原始字节（包括‘\0’）和传统的以‘\0’结尾的8位字符串。
- 使用QByteArray比使用const char * 要方便很多，在后台，它总是保证数据以一个‘\0’结尾，而且使用隐式共享来减少内存的使用和避免不必要的数据拷贝。
- 但是除了当需要存储原始二进制数据或者对内存保护要求很高（如在嵌入式Linux上）时，一般都推荐使用QString，因为QString是存储16位的Unicode字符，使得在应用程序中更容易存储非ASCII和非Latin-1字符，而且QString全部使用的是Qt的API。
- QByteArray类拥有和QString类相似的接口函数，除了arg()以外，在QByteArray中都有相同的用法。

QVariant

- QVariant类像是最常见的Qt的数据类型的一个共用体（union），一个QVariant对象在一个时间只保存一个单一类型的一个单一的值（有些类型可能是多值的，比如字符串列表）。
- 可以使用toT()（T代表一种数据类型）函数来将QVariant对象转换为T类型，并且获取它的值。这里toT()函数会复制以前的QVariant对象，然后对其进行转换，所以以前的QVariant对象并不会改变。
- QVariant是Qt中一个很重要的类，比如前面讲解属性系统时提到的QObject::property()返回的就是QVariant类型的对象。

示例：

```
QVariant v1(15);
qDebug() << v1.toInt();                    // 结果为15

QVariant v2(12.3);
qDebug() << v2.toFloat();                   // 结果为12.3

QVariant v3("nihao");
qDebug() << v3.toString();                  // 结果为"nihao “

QColor color = QColor(Qt::red);
QVariant v4 = color;
qDebug() << v4.type();                     // 结果为QVariant::QColor
qDebug() << v4.value<QColor>();            // 结果为QColor(ARGB
    1,1,0,0)

QString str = "hello";
QVariant v5 = str;
qDebug() << v5.canConvert(QVariant::Int);  // 结果为true
qDebug() << v5.toString();                 // 结果为"hello"
qDebug() << v5.convert(QVariant::Int);     // 结果为false
qDebug() << v5.toString();                 // 转换失败，v5被清空，结果为"
```

- QVariant类的toInt()函数返回int类型的值，toFloat()函数返回float类型的值。
- 因为QVariant是QtCore库的一部分，所以它没有提供对QtGui中定义的数据类型（例如QColor、QImage等）进行转换的函数，也就是说，这里没有toColor()这样的函数。不过，我们可以使用QVariant::value()函数或者qVariantValue()模板函数来完成这样的转换，例如上面程序中对QColor类型的转换。
- 对于一个类型是否可以转换为一个特殊的类型，可以使用canConvert()函数来判断，如果可以转换，则该函数返回true。
- 也可以使用convert()函数来将一个类型转换为其他不同的类型，如果转换成功则返回true，如果无法进行转换，variant对象将会被清空，并且返回false。
- 需要说明，对于同一种转换，canConvert()和convert()函数并不一定返回同样的结果，这通常是因为canConvert()只报告QVariant进行两个类型之间转换的能力。也就是说，如果在提供了合适的数据时，这两个类型间可以进行转换，但是，如果提供的数据不合适，那么转换就会失败，这样convert()的返回值就与canConvert()不同了。例如上面程序中的QString类型的字符串str，当str中只有数字字符时，它可以转换为int类型，比如str = “123”，因为它有这个能力，所以canConvert()返回为true。但是，现在str中包含了非数字字符，真正进行转换时会失败，所以convert()返回为false。

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。

- 示例

```
QRegExp rx("^\\d\\d?$");          // 两个字符都必须为数字，第二个字符可以没有
qDebug() << rx.indexIn("a1");          // 结果为-1，不是数字开头
qDebug() << rx.indexIn("5");          // 结果为0
qDebug() << rx.indexIn("5b");          // 结果为-1，第二个字符不是数字
qDebug() << rx.indexIn("12");          // 结果为0
qDebug() << rx.indexIn( “123” );          // 结果为-1，超过了两个字符

rx.setPattern("\\b(mail|letter)\\b");    // 匹配mail或者letter单词
qDebug() << rx.indexIn("emailletter");   // 结果为-1，mail不是一个单词
qDebug() << rx.indexIn("my mail");       // 返回3
qDebug() << rx.indexIn("my email letter"); // 返回9

rx.setPattern("M(?!ail)");              // 匹配字符M，其后面不能跟有ail字符
QString str1 = "this is M";
str1.replace(rx,"Mail");                // 使用"Mail"替换匹配到的字符
qDebug() << "str1: " << str1;           // 结果为this is Mail
QString str2 = "my M,your Ms,his Mail";
str2.replace(rx,"Mail");
qDebug() << "str2: " << str2;           // 结果为my Mail,your Mails,his Mail
```

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

第一个0。

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。

例如：

QRegExp rx3("*.txt");

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

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。

示例：

```
QRegExp rx4("(\\d+)");
QString str4 = "Offsets: 12 14 99 231 7";
QStringList list;
int pos2 = 0;
while ((pos2 = rx4.indexIn(str4, pos2)) != -1)
{
    list << rx4.cap(1);                          // 第一个捕获到的文本
    pos2 += rx4.matchedLength();                 // 上一个匹配的字符串的长度
}
qDebug() << list;                            // 结果12,14,99,231,7

QRegExp rxlen("(\\d+)(?:\\s*)(cm|inch)");
int pos3 = rxlen.indexIn("Length: 189cm");
if (pos3 > -1) {
    QString value = rxlen.cap(1);       // 结果为189
    QString unit = rxlen.cap(2);            // 结果为cm
    QString string = rxlen.cap(0);          // 结果为189cm
    qDebug() << value << unit << string;
}
```

新的QRegularExpression类

- 在Qt 5中引入了新的QRegularExpression类，实现了与Perl兼容的正则表达式，并在QRegExp基础上进行了很大的改进。建议编写Qt 5程序时使用QRegularExpression来代替QRegExp。

- 在QRegularExpression中，一个正则表达式由两部分构成：一个模式字符串和一组模式选项，模式选项用来更改模式字符串的含义。可以在构造函数中直接设置模式字符串：

```
QRegularExpression re("a pattern");
```

- 也可以使用setPattern()为已有的QRegularExpression对象设置模式字符串：

```
QRegularExpression re;
re.setPattern("another pattern");
```

- 可以通过pattern()来获取已设置的模式字符串。在QRegularExpression中通过模式选项来改变模式字符串的含义，例如，可以通过设置QRegularExpression::CaseInsensitiveOption使匹配时不区分字母大小写：

```
QRegularExpression re("Qt rocks",
                      QRegularExpression::CaseInsensitiveOption);
```

- 这时除了匹配Qt rocks，还会匹配QT rocks、QT ROCKS、qT rOcKs等字符串。也可以通过setPatternOptions()来设置模式选项，通过patternOptions()获取设置的模式选项：

```
QRegularExpression re("^\\d+$");
re.setPatternOptions(QRegularExpression::MultilineOption);
QRegularExpression::PatternOptions options =
 re.patternOptions();
```

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；也学习了容器类及其相关的QString，QByteArray和QVariant等类；还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。

- 什么是信号和槽，怎么使用信号和槽？
- 简单介绍Qt的对象树？
- Qt中常用的容器类有哪些？
- 使用Qt中的正则表达式来验证一个字符串？

界面外观

一个完善的应用程序不仅应该有实用的功能，还要有一个漂亮的外观，这样才能使应用程序更加友善，更加吸引用户。

作为一个跨平台的UI开发框架，Qt提供了强大而灵活的界面外观设计机制。这一章将学习在Qt中设计应用程序外观的相关知识，在本章开始会对Qt风格QStyle和调色板QPalette进行简单介绍，然后再对Qt样式表（Qt Style Sheets）进行重点讲解，最后还会涉及不规则窗体和透明窗体的实现方法。

主 要 内 容

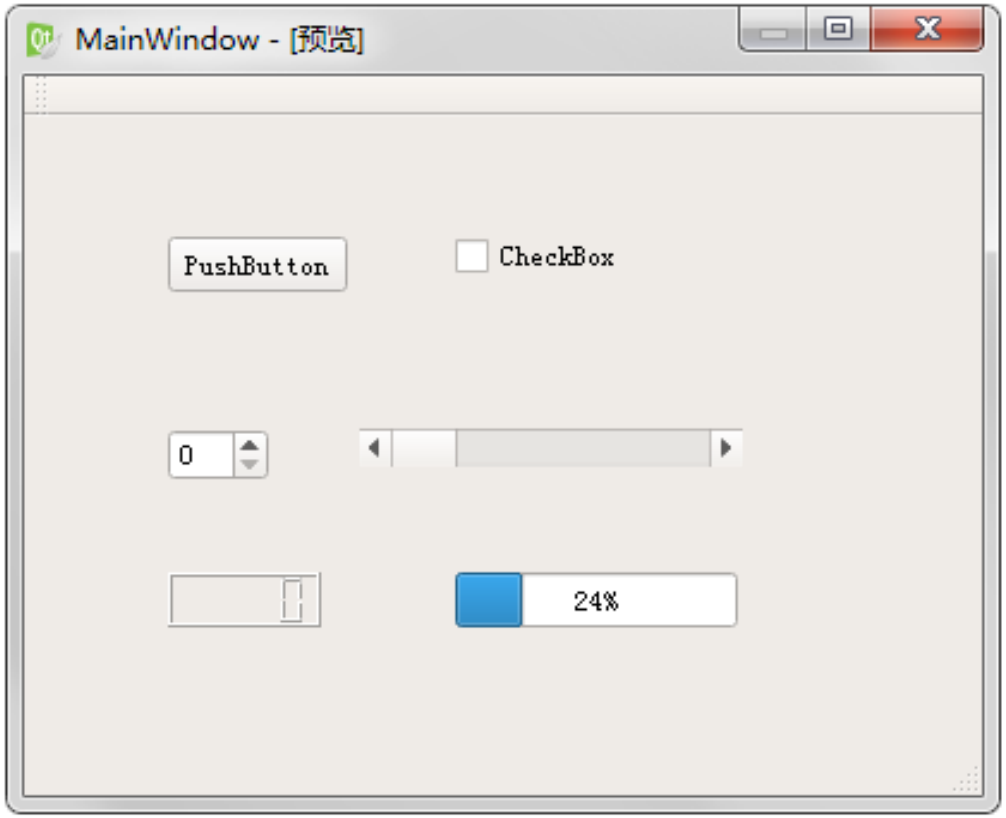
- Qt风格
- Qt样式表
- 特殊效果窗体

Qt风格

Qt中的各种风格是一组继承自QStyle的类。QStyle类是一个抽象基类，封装了一个GUI的外观，Qt的内建（built-in）部件使用它来执行几乎所有的绘制工作，以确保它们看起来可以像各个平台上的本地部件一样。QStyleFactory类可以创建一个QStyle对象，首先通过keys()函数获取可用的风格，然后使用create()函数创建一个QStyle对象。一般windows风格和fusion风格是默认可用的，而有些风格只在特定的平台上才有效，例如windowsxp风格、windowsvista风格、gtk风格和macintosh风格。

使用不同风格预览程序

首先进入设计模式，可以先修改界面，然后选择“工具→Form Editor→Preview in”菜单项，这里列出了现在可用的几种风格，选择“Fusion风格”，预览效果如下图所示。也可以使用其他几种风格进行预览。



使用不同风格运行程序

- 如果想使用不同的风格来运行程序，那么只需要调用QApplication的setStyle()函数指定要使用的风格即可。

例如，在main()函数的“QApplication a(argc, argv);”一行代码后添加如下一行代码：

```
a.setStyle(QStyleFactory::create("fusion"));
```

这时运行程序，便会使用Fusion风格。

- 而如果不想整个应用程序都使用相同的风格，那么可以调用部件的setStyle()函数来指定该部件的风格。

调色板

调色板QPalette类包含了部件各种状态的颜色组。一个调色板包含三种状态：激活（Active）、失效（Disabled）和非激活（Inactive）。Qt中的所有部件都包含一个调色板，并且使用各自的调色板来绘制它们自身，这样可以使用户界面更容易配置，也更容易保持一致。调色板中的颜色组包括：

- 激活颜色组QPalette::Active，用于获得键盘焦点的窗口；
- 非激活颜色组QPalette::Inactive，用于其他的窗口；
- 失效颜色组QPalette::Disabled，用于因为一些原因而不可用的部件（不是窗口）。

要改变一个应用程序的调色板，可以先使用QApplication::palette()函数来获取其调色板，然后对其进行更改，最后再使用QApplication::setPalette()函数来使用该调色板。更改了应用程序的调色板，会影响到该程序的所有窗口部件。如果要改变一个部件的调色板，可以调用该部件的palette()和setPalette()函数，这样只会影响该部件及其子部件。

常量	描述
QPalette::Window	一般的背景颜色
QPalette::WindowText	一般的前景颜色
QPalette::Base	主要作为输入部件（如 QLineEdit）的背景色，也可用作 QComboBox 的下拉列表的背景色或者 QToolBar 的手柄颜色，一般是白色或其他浅色
QPalette::AlternateBase	在交替行颜色的视图中作为交替背景色
QPalette::ToolTipBase	作为 QToolTip 和 QWhatsThis 的背景色
QPalette::ToolTipText	作为 QToolTip 和 QWhatsThis 的前景色
QPalette::Text	和 Base 一起使用，作为前景色
QPalette::Button	按钮部件背景色
QPalette::ButtonText	按钮部件前景色
QPalette::BrightText	一种与深色对比度较大的文本颜色，一般用于当 Text 或者 WindowText 的对比度较差时

```
palette2.setColor(QPalette::Disabled,QPalette::Base,Qt::blue);
ui->lineEdit->setPalette(palette2);
```

设置调色板颜色时可以使用setColor()函数，这个函数需要指定颜色角色（Color Role）。在QPalette中，颜色角色用来指定该颜色所起的作用，例如是背景颜色或者是文本颜色等，主要的颜色角色如表所示。

Qt样式表

Qt样式表是一个可以自定义部件外观的十分强大的机制。Qt样式表的概念、术语和语法都受到了HTML的层叠样式表（Cascading Style Sheets，CSS）的启发，不过与CSS不同的是，Qt样式表应用于部件的世界。

- Qt样式表概述
- Qt样式表语法
- 自定义部件外观和换肤

Qt样式表概述

- 可以使用QApplication::setStyleSheet()函数将其设置到整个应用程序上；
- 也可以使用QWidget::setStyleSheet()函数将其设置到一个指定的部件（还有它的子部件）上。
- 如果在不同的级别都设置了样式表，那么Qt会使用所有有效的样式表，这被称为样式表的层叠。

使用代码设置样式表

例如：

```
// 设置pushButton的背景为黄色
ui->pushButton->setStyleSheet("background:yellow");
// 设置horizontalSlider的背景为蓝色
ui->horizontalSlider->setStyleSheet("background:blue");
```

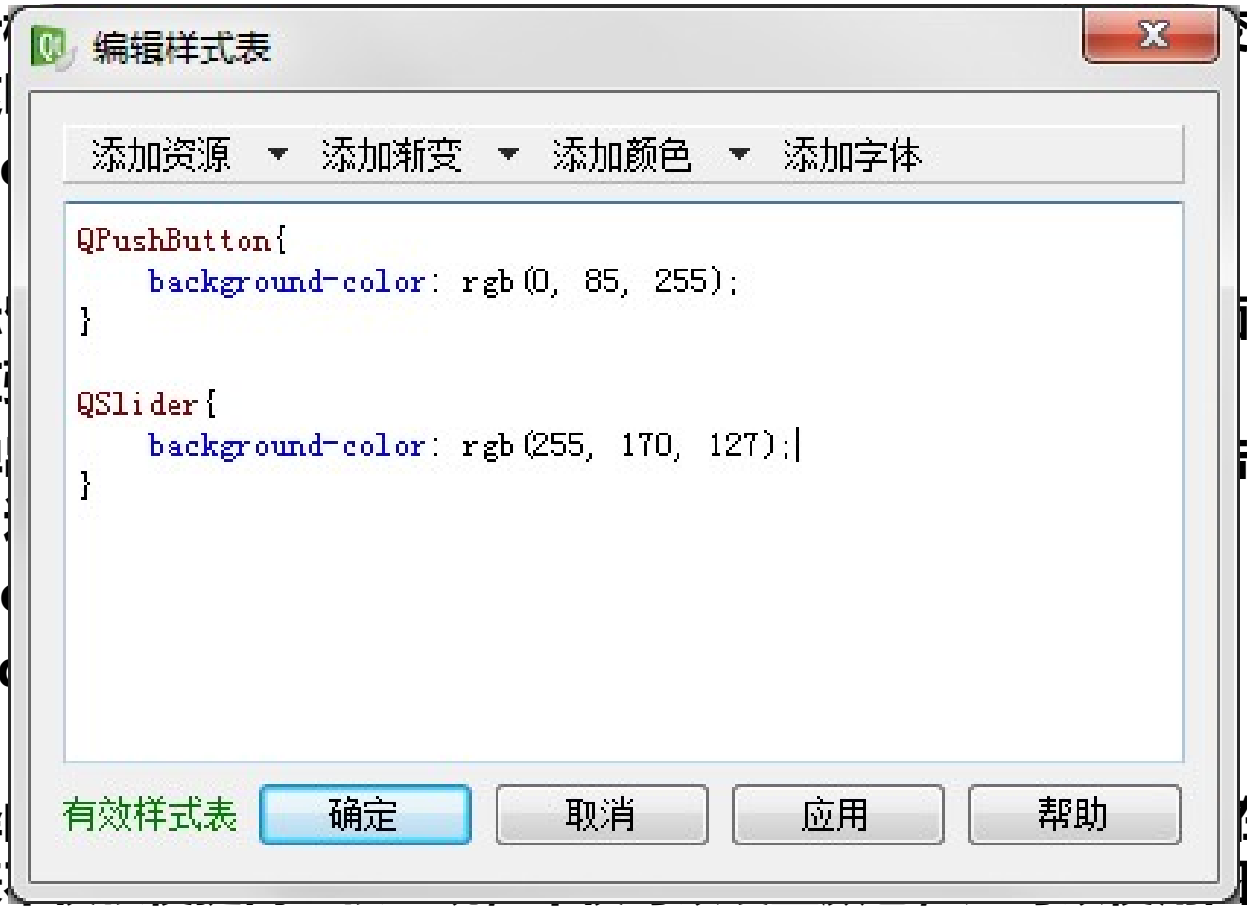
这样调用指定部件的setStyleSheet()函数只会对这个部件应用该样式表，如果想对所有的相同部件都使用相同的样式表，那么可以在它们的父部件上设置样式表。比如这里两个部件都在MainWindow上，可以为MainWindow设置样式表：

```
setStyleSheet("QPushButton{background:yellow}QSlider{background:blue}
");
```

这样，以后再往主窗口上添加的所有QPushButton部件和QSlider部件的背景色都会改为这里指定的颜色。

在设计模式使用样式表

进入设计
表”，这
QPushButt
}
注意光标
箭头，在
这时会弹
按钮”，
QPushButt
backgroun
}
根据选择
置样式表



样式
的下拉
“确定
在这里设
片，使
用渐变颜色，或者更改字体。相似的，可以再设置QSlider的背景色。

样式表语法

样式规则：

- 样式表包含了一系列的样式规则，一个样式规则由一个选择器（selector）和一个声明（declaration）组成。选择符指定了受该规则影响的部件；声明指定了这个部件上要设置的属性。例如：

  QPushButton{color:red}

  在这个样式规则中，QPushButton是选择器，{color:red}是声明，而color是属性，red是值。这个规则指定了QPushButton和它的子类应该使用红色作为它们的前景色。
- Qt样式表中一般不区分大小写，例如color、Color、COLOR和COloR表示相同的属性。只有类名，对象名和Qt属性名是区分大小写的。
- 一些选择器可以指定相同的声明，只需要使用逗号隔开，例如：

  QPushButton,QLineEdit,QComboBox{color:red}
- 一个样式规则的声明部分是一些“属性:值”对组成的列表，它们包含在大括号中，使用分号隔开。例如：

  QPushButton{color:red;background-color:white}

选择器类型

Qt样式表支持在CSS2中定义的所有选择器。下表列出了最常用的选择器类型。

选择器	示例	说明
通用选择器	*	匹配所有部件
类型选择器	QPushButton	匹配所有 QPushButton 实例和它的所有子类
属性选择器	QPushButton[flat="false"]	匹配 QPushButton 的属性 flat 为 false 的实例
类选择器	.QPushButton	匹配所有 QPushButton 实例，但不包含它的子类
ID 选择器	QPushButton#okButton	匹配所有 QPushButton 中以 okButton 为对象名的实例
后代选择器	QDialog QPushButton	匹配所有 QPushButton 实例，它们必须是 QDialog 的子孙部件
孩子选择器	QDialog>QPushButton	匹配所有 QPushButton 实例，它们必须是 QDialog 的直接子部件

子控件

对一些复杂的部件修改样式，可能需要访问它们的子控件，例如QComboBox的下拉按钮，还有QSpinBox的向上和向下的箭头等。选择器可以包含子控件来对部件的特定子控件应用规则，例如：

```
QComboBox::drop-down{image:url(dropdown.png)}
```

这样的规则可以改变所有的QComboBox部件的下拉按钮的样式。

伪状态

- 选择器可以包含伪状态来限制规则在部件的指定的状态上应用。伪状态出现在选择器之后，用冒号隔离，例如：

  QPushButton:hover{color:white}

- 这个规则表明当鼠标悬停在一个QPushButton部件上时才被应用。伪状态可以使用感叹号来表示否定，例如要当鼠标没有悬停在一个QRadioButton上时才应用规则，那么这个规则可以写为：

  QRadioButton:!hover{color:red}

- 伪状态还可以多个连用，达到逻辑与效果，例如当鼠标悬停在一个被选中的QCheckBox部件上时才应用规则，那么这个规则可以写为：

  QCheckBox:hover:checked{color:white}

- 如果有需要，也可以使用逗号来表示逻辑或操作，例如：

  QCheckBox:hover,QCheckBox:checked{color:white}

冲突解决

当几个样式规则对相同的属性指定了不同的值时就会产生冲突。例如：

```
QPushButton#okButton { color: gray }
QPushButton { color: red }
```

这样okButton的color属性便产生了冲突。解决这个冲突的原则是：特殊的选择器优先。在这里，因为QPushButton#okButton一般代表一个单一的对象，而不是一个类所有的实例，所以它比QPushButton更特殊，那么这时便会使用第一个规则，okButton的文本颜色为灰色。

相似的，有伪状态比没有伪状态优先。如果两个选择器的特殊性相同，则后面出现的比前面的优先。Qt样式表使用CSS2规范来确定规则的特殊性。

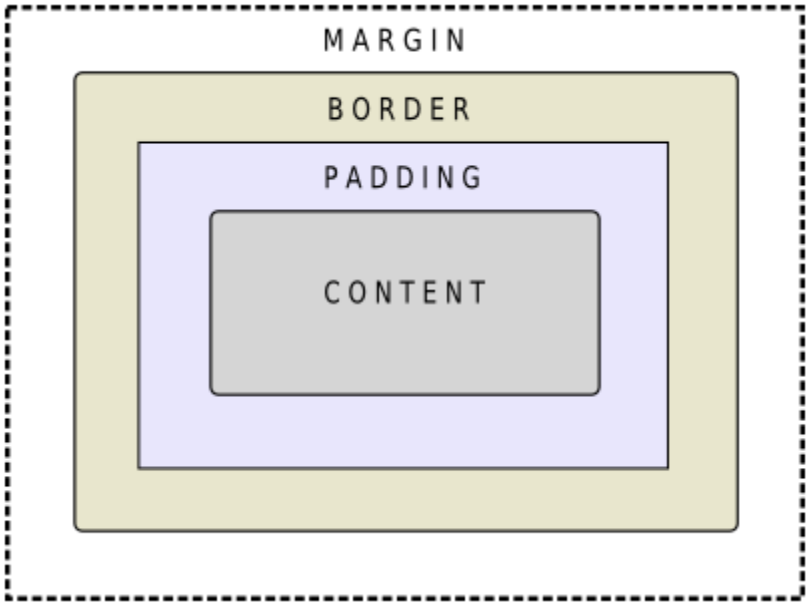
层叠

样式表可以被设置在QApplication上，或者父部件上，或者子部件上。部件有效的样式表是通过部件祖先的样式表和QApplication上的样式表合并得到的。

当发生冲突时，部件自己的样式表优先于任何继承的样式表，同样，父部件的样式表优先于祖先的样式表。

自定义部件外观

当使用样式表时，每一个部件都被看做是拥有四个同心矩形的盒子，如下图所示。这四个矩形分别是内容（content）、填衬（padding）、边框（border）和边距（margin）。边距、边框宽度和填衬等属性的默认值都是0，这样四个矩形恰好重合。



```
/****************主界面背景******************/
QMainWindow{
/*背景图片*/
background-image: url(:/image/beijing01.png);
}
/****************按钮部件******************/
QPushButton{
/*背景色*/
background-color: rgba(100, 225, 100, 30);
/*边框样式*/
border-style: outset;
/*边框宽度为4像素*/
border-width: 4px;
/*边框圆角半径*/
border-radius: 10px;
/*边框颜色*/
border-color: rgba(255, 225, 255, 30);
/*字体*/
font: bold 14px;
/*字体颜色*/
color:rgba(0, 0, 0, 100);
/*填衬*/
padding: 6px;
}
```

```
/*鼠标悬停在按钮上时*/
QPushButton:hover{
background-color:rgba(100,255,100, 100);
border-color: rgba(255, 225, 255, 200);
color:rgba(0, 0, 0, 200);
}
/*按钮被按下时*/
QPushButton:pressed {
background-color:rgba(100,255,100, 200);
border-color: rgba(255, 225, 255, 30);
border-style: inset;
color:rgba(0, 0, 0, 100);
}
/*****************滑块部件*********************
/*水平滑块的手柄*/
QSlider::handle:horizontal {
image: url(:/image/sliderHandle.png);
}
/*水平滑块手柄以前的部分*/
QSlider::sub-page:horizontal {
/*边框图片*/
border-image: url(:/image/slider.png);
}
```

实现换肤功能

Qt样式表可以存放在一个以.qss为后缀的文件中，这样我们就可以在程序中调用不同的.qss文件来实现换肤的功能。

在换肤按钮中这样设置：

```
void MainWindow::on_pushButton_clicked(){
    QFile file(":/qss/my1.qss");
    file.open(QFile::ReadOnly);
    QString styleSheet = tr(file.readAll());
    qApp->setStyleSheet(styleSheet);
}
```

现在可以运行程序了，当大家按下按钮后，便会更改界面的外观，这样就实现了换肤功能。

例如通过下面的代码使用样式表：

```
QFile file(":/qss/my.qss");
// 只读方式打开该文件
file.open(QFile::ReadOnly);
// 读取文件全部内容
QString styleSheet = QString(file.readAll());
// 为QApplication设置样式表
qApp->setStyleSheet(styleSheet);
```

这里读取了Qt样式表文件中的内容，然后为应用程序设置了样式表。

8.3 特殊效果窗体

- 不规则窗体
- 透明窗体

不规则窗体

Qt中提供了部件遮罩（mask）来实现不规则窗体。例如：

- 先在构造函数中添加如下代码：

```
QPixmap pix;
pix.load(":/image/yafeilinux.png"); // 加载图片
resize(pix.size()); // 设置窗口大小为图片大小
setMask(pix.mask()); // 为窗口设置遮罩
```

- 然后在paintEvent()函数中将图片绘制在窗口上：

```
void Widget::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    // 从窗口左上角开始绘制图片
    painter.drawPixmap(0,0,QPixmap(":/image/yafeilinux.png"));
}
```

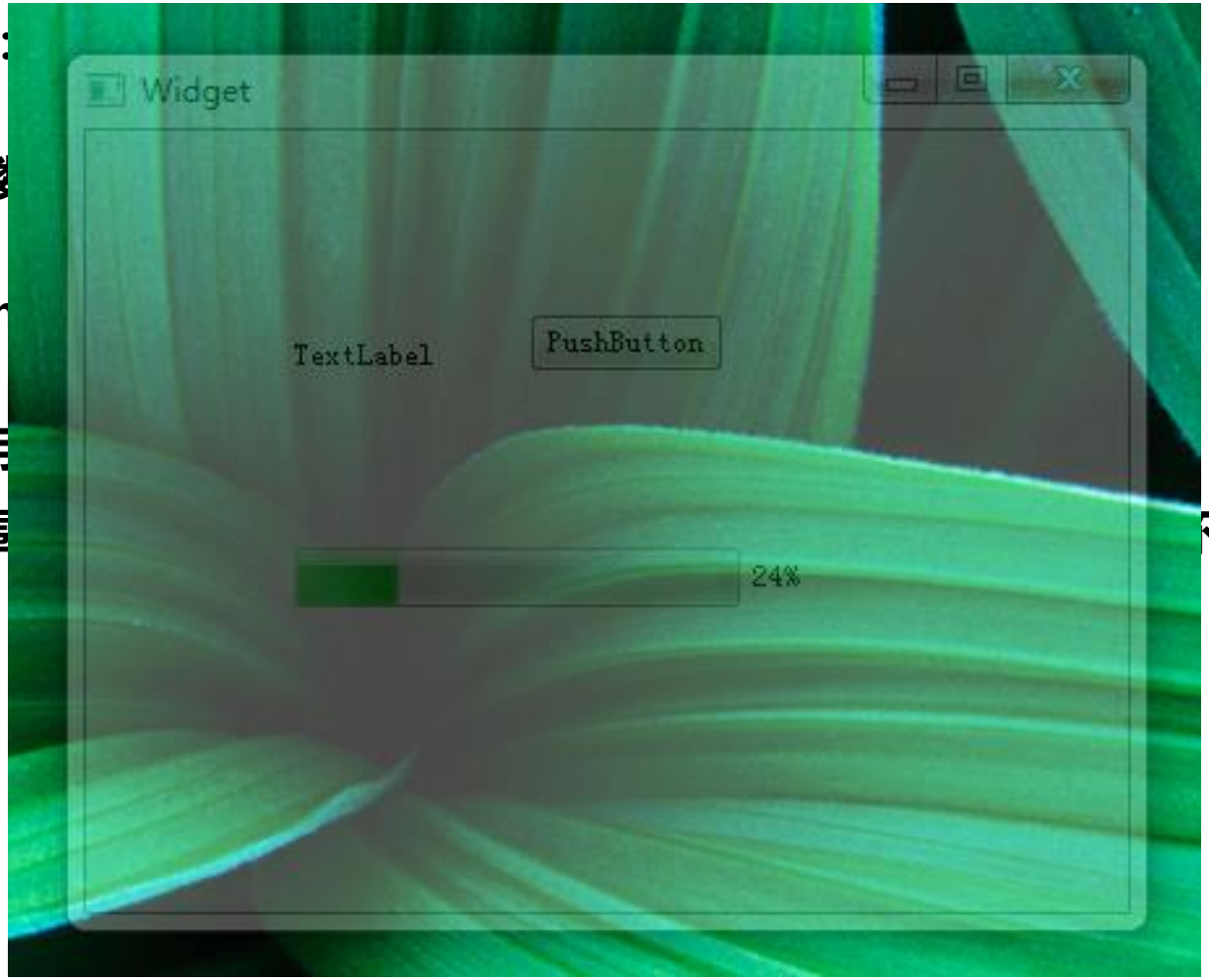
透明窗体

方式一：

构造函数

setWin

使用 它的参数取值范围 不透明。



方式二：使用setAttribute()函数。例如：

在构造函

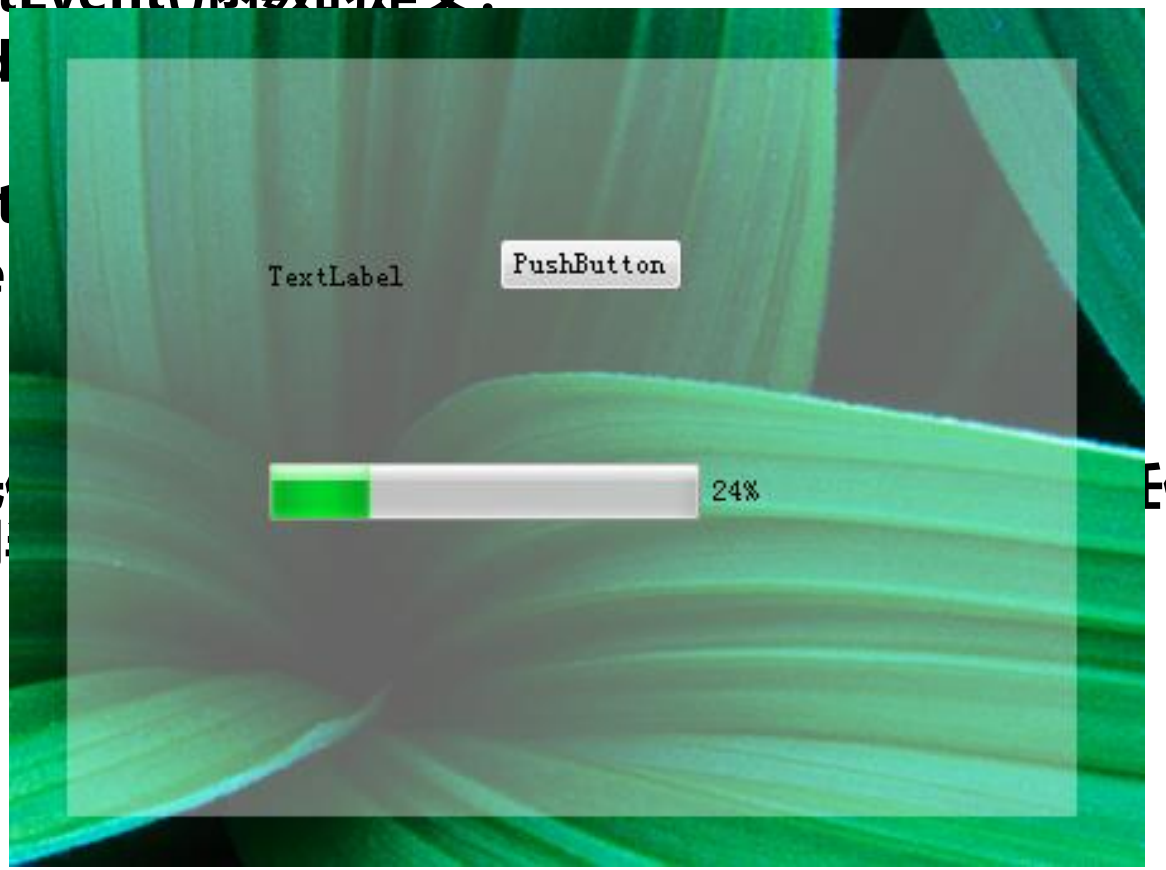
setWind
setAttrib

这里使 ckground
属性， dows下，
还要使 标志，这
样才能

方式三：在方式二的基础上修改重绘事件。例如：

进行paintEvent()函数的定义：

```
void Wid
{
    QPaint
    painte
}
```



这里先 任何边框。然后使用

自学

小结

本章要掌握最基本的更改部件样式的方法。通过综合使用本章讲到的这些知识，应该可以实现一些简单的界面效果。重点掌握Qt样式表的用法。

- 如何使用Qt调色板？
- Qt样式表语法主要包含哪几部分？
- Qt样式表中的盒子模型？
- 在Qt中如何实现不规则窗体？