Qt编程 学习笔记

付亚军

**专题1 Qt全局宏和变量**

1. Qt 全局宏定义

Qt版本号:

QT\_VERSION :　 (major << 16) + (minor << 8) + patch

检测版本号：

QT\_VERSION\_CHECK(major, minor, patch)((major<<16)|(minor<<8)|(patch))

当使用命名空间时的一些宏定义：

namespace QT\_NAMESPACE {}　//命名空间定义

# define QT\_PREPEND\_NAMESPACE(name) ::QT\_NAMESPACE::name

# define QT\_USE\_NAMESPACE using namespace ::QT\_NAMESPACE;

# define QT\_BEGIN\_NAMESPACE namespace QT\_NAMESPACE {

# define QT\_END\_NAMESPACE }

# define QT\_BEGIN\_INCLUDE\_NAMESPACE }

# define QT\_END\_INCLUDE\_NAMESPACE namespace QT\_NAMESPACE {

# define QT\_BEGIN\_MOC\_NAMESPACE QT\_USE\_NAMESPACE

# define QT\_END\_MOC\_NAMESPACE

# define QT\_FORWARD\_DECLARE\_CLASS(name) \

QT\_BEGIN\_NAMESPACE class name; QT\_END\_NAMESPACE \

using QT\_PREPEND\_NAMESPACE(name);

# define QT\_FORWARD\_DECLARE\_STRUCT(name) \

QT\_BEGIN\_NAMESPACE struct name; QT\_END\_NAMESPACE \

using QT\_PREPEND\_NAMESPACE(name);

Q\_GLOBAL\_STATIC(type, name)用于定义一个全局的静态变量（线程安全的）．用它来初始化一个对象指针（函数指针）．

Q\_GLOBAL\_STATIC\_WITH\_ARGS(type, name, arguments)：同上，可带参数．

**Q\_UNUSED（）**:通常用于某个函数的参数，表明该参数未在方法体中使用．

常用数据类型：

qint8, quint8, qint16, quint16, qint32, quint32, qint64, quint64, qlonglong(=int64), qulonglong(=uint64)

quintptr, qptrdiff, 作为指针是一样大的．如：

sizeof(void \*) == sizeof(quintptr)

&& sizeof(void \*) == sizeof(qptrdiff)

Qt有用的类型定义：uchar, ushort, uint, ulong

实用内联函数：qAbs(), qRound(), qRound64(), qMin(), qMax(), qBound();

try-catch宏：

#ifdef QT\_NO\_EXCEPTIONS

# define QT\_TRY if (true)

# define QT\_CATCH(A) else

# define QT\_THROW(A) qt\_noop()

# define QT\_RETHROW qt\_noop()

#else

# define QT\_TRY try

# define QT\_CATCH(A) catch (A)

# define QT\_THROW(A) throw A

# define QT\_RETHROW throw

#endif

每个QT\_TRY对应一个ＱT\_CATCH，不要在catch语句块中使用异常实例．

调试相关函数：

qDebug(), qWarnig(), qCritical(), Q\_ASSERT()

其他一些函数：

该宏用于定义某个类，禁止拷贝

#define **Q\_DISABLE\_COPY(Class)** \

Class(const Class &); \

Class &operator=(const Class &);

qgetenv(), qputenv(), qIntCast(double 或float). qsrand(uint seed), qrand()

2. Qt的内存管理函数

void \*qMalloc(size\_t size);

void qFree(void \*ptr);

void \*qRealloc(void \*ptr, size\_t size);

void \*qMallocAligned(size\_t size, size\_t alignment)

void \*qReallocAligned(void \*oldptr, size\_t newsize, size\_t oldsize, size\_t alignment)

void qFreeAligned(void \*ptr)

void \*qMemCopy(void \*dest, const void \*src, size\_t n);

void \*qMemSet(void \*dest, int c, size\_t n);

**注:** 数组下标为-1的地址对于数组来说是越界访问了，但是这个地址是有意义的.这个地址就是所申请的数组存储空间的首地址的向前偏移一个单位(也就是偏移一个当前数组类型所对应的字节数)所对应的地址。这个地址由于没有跟着数组空间一起初始化，所以其中的数据是不确定的.如果是正在被系统或者其他应用程序使用中的地址空间，那么可以被访问，其中的数据的意义取决于被系统或者其他应用程序所写入的数据，但是访问后，有可能会引起系统或者其他应用程序异常。如果是没有被使用的地址，那么就是一个野地址，那么其中的数据是随机的，无意义的

３.　环境变量设置函数

qputenv()：与putenv()基本类似，由于VC2005后废除了putenv()函数，所有定义了此函数，在VC环境中调用替代的函数，其他平台调用标准Ｃ库函数putenv()．

qgetenv():获取某个环境变量的值．

４.自定义输出函数

QtMsgHandler qInstallMsgHandler(QtMsgHandler h)

其中：typedef void (\*QtMsgHandler)(QtMsgType, const char \*);

利用该函数可定制自己的调试信息输出．

5. 断言

Q\_ASSERT(bool test)：当断言失败时，打印出警告信息，包含了源码文件名和行号．

void Q\_ASSERT\_X(bool test, const char \*where, const char \*what)

void Q\_CHECK\_PTR(void \*pointer)：检测非法指针，如果指针为０，打印警告信息．

６.系统信息类QsysInfo

**专题二　Ｉ/O处理**

1. 类QabstractFileEngineHandler

该类提供了一种在应用程序中注册定制的文件引擎的方法．它是创建QabstractFileEngine对象的一个工厂，这些对象由ＱFile, QfileInfo和Qdir内部使用．当打开一个文件时，Ｑt通过从Qfile或Qdir传递过来的文件名从内部注册的文件引擎处理器列表中选择一个合适的文件引擎．第一个识别文件名的处理器用于创建引擎．Qt已经提供了处理普通文件和资源的内部文件引擎，当然，你也可以注册自己的QabstractFileEngine子类．

1. QabstractFileEngine

该类提供了访问文件系统的一个抽象．

**专题三　状态机框架**

状态机框架提供了创建和执行状态图的一些类．这些概念和表示都是基于Ｈarel状态图中的一些概念和记法．它也是UML状态图表的基础．状态机执行的语义是基于状态图XML(SCXML)．

状态图提供了一个图形的方式来对一个系统建模以反应它怎么响应外部触发．这是通过定义系统可能进入的一些状态以及系统怎么从一个状态转换到另一个状态（不同状态之间转变）来实现的．事件驱动系统的一个关键的特征（例如Qt应用程序）就是行为通常不仅取决于上次或当前事件，还取决于在它之前的一些事件．用状态图，这个信息非常容易表达．

状态机框架提供了一个API和执行模型，可有效地将状态图的元素和语义嵌入到Ｑt应用程序当中．该框架与Qt的元对象系统结合紧密：例如，不同状态之间的转变可由信号触发且状态可配置用来设置QObject的属性和方法．Qt的事件系统用于驱动状态机．

状态机框架中的状态图是分层的．状态可嵌套在另一个状态内．状态机的当前配置包含一些当前活跃的状态．状态机中的一个合法的配置中的所有状态都有一个共同的祖先．

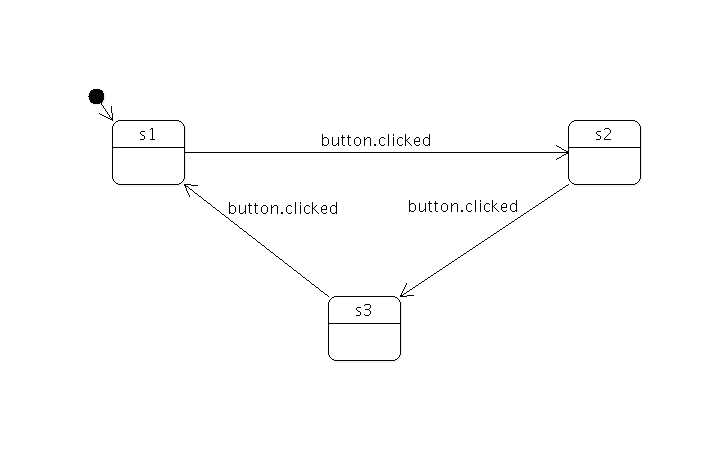
**状态机框架中的类**

qt提供了这些类来创建事件驱动的状态机．

|  |  |
| --- | --- |
| **[QAbstractState](http://doc.qt.nokia.com/4.6/qabstractstate.html)** | The base class of states of a QStateMachine |
| **[QAbstractTransition](http://doc.qt.nokia.com/4.6/qabstracttransition.html)** | The base class of transitions between QAbstractState objects |
| **[QEventTransition](http://doc.qt.nokia.com/4.6/qeventtransition.html)** | QObject-specific transition for Qt events |
| **[QFinalState](http://doc.qt.nokia.com/4.6/qfinalstate.html)** | Final state |
| **[QHistoryState](http://doc.qt.nokia.com/4.6/qhistorystate.html)** | Means of returning to a previously active substate |
| **[QKeyEventTransition](http://doc.qt.nokia.com/4.6/qkeyeventtransition.html)** | Transition for key events |
| **[QMouseEventTransition](http://doc.qt.nokia.com/4.6/qmouseeventtransition.html)** | Transition for mouse events |
| **[QSignalTransition](http://doc.qt.nokia.com/4.6/qsignaltransition.html)** | Transition based on a Qt signal |
| **[QState](http://doc.qt.nokia.com/4.6/qstate.html)** | General-purpose state for QStateMachine |
| **[QStateMachine](http://doc.qt.nokia.com/4.6/qstatemachine.html)** | Hierarchical finite state machine |
| **[QStateMachine::SignalEvent](http://doc.qt.nokia.com/4.6/qstatemachine-signalevent.html)** | Represents a Qt signal event |
| **[QStateMachine::WrappedEvent](http://doc.qt.nokia.com/4.6/qstatemachine-wrappedevent.html)** | Holds a clone of an event associated with a QObject |

**一个简单的状态机**

为了演示状态机API的核心功能，让我们来看一个小例子：一个状态机有三个状态s1,s2和s3．状态机由一个按钮来控制；当点击按钮时，状态机转换到另一个状态．刚开始时，状态机处于状态s1．该状态机的状态图如下所示：



下面代码段显示了创建一个这样的状态机所需的代码．首先，我们创建一个状态机和一些状态：

QStateMachine machine;

QState \*s1 = new QState();

QState \*s2 = new QState();

QState \*s3 = new QState();

然后，我们使用[QState::addTransition](http://doc.qt.nokia.com/4.6/qstate.html" \l "addTransition)()函数创建转换：

s1->addTransition(button, SIGNAL(clicked()), s2);

s2->addTransition(button, SIGNAL(clicked()), s3);

s3->addTransition(button, SIGNAL(clicked()), s1);

接下来，我们将这些状态加入状态机中并设置它的初始状态：

machine.addState(s1);

machine.addState(s2);

machine.addState(s3);

machine.setInitialState(s1);

最后，我们启动状态机：

machine.start();

状态是异步执行的，例如，它成为你的应用程序事件循环的一部分．

**在状态入口和出口做有用的工作**

上面的状态机仅仅从一个状态转换到另一个状态，并没有执行任何操作．[QState::assignProperty](http://doc.qt.nokia.com/4.6/qstate.html" \l "assignProperty)()函数可用于当进入某个状态时设置某个Qobject的一个属性．在下面的代码段中，为每个状态指定了应当赋给Qlabel的text属性的值．

s1->assignProperty(label, "text", "In state s1");

s2->assignProperty(label, "text", "In state s2");

s3->assignProperty(label, "text", "In state s3");

当进入了这些状态中的任何一个，标签的值就会相应地改变．

当进入某个状态时，就会发出[QState::entered](http://doc.qt.nokia.com/4.6/qabstractstate.html" \l "entered)()信号．当离开这个状态时，就会发出[QState::exited](http://doc.qt.nokia.com/4.6/qabstractstate.html" \l "exited)() 信号．在下面的代码段中，按钮的showMaximize()槽在进入状态s3时被调用．当退出状态s3时调用showMinimized()：

QObject::connect(s3, SIGNAL(entered()), button, SLOT(showMaximized()));

QObject::connect(s3, SIGNAL(exited()), button, SLOT(showMinimized()));

自定义的状态可以重新实现方法[QAbstractState::onEntry](http://doc.qt.nokia.com/4.6/qabstractstate.html" \l "onEntry)() 和 [QAbstractState::onExit](http://doc.qt.nokia.com/4.6/qabstractstate.html" \l "onExit)().

**完成的状态机**

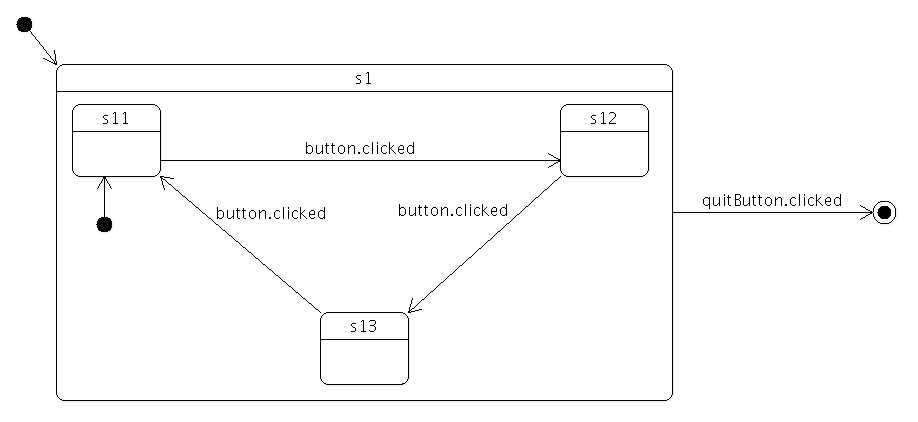
前面部分定义的状态机从不完成．为了使一个状态机能够完成，它需要拥有一个顶层的最终状态（[QFinalState](http://doc.qt.nokia.com/4.6/qfinalstate.html) 对象）．当状态机进入一个顶层最终状态时，该状态机将会释放[QStateMachine::finished](http://doc.qt.nokia.com/4.6/qstate.html" \l "finished)()信号并停止．

在图中引入一个最终状态，所有你需要做的就是创建一个[QFinalState](http://doc.qt.nokia.com/4.6/qfinalstate.html)对象且使用它作为一个或多个转换的目标．

**通过对状态进行分组来共享转换**

假设我们想让用户能够通过点击Quit揿钮在任何时刻能够退出应用程序．为了完成这个目标，我们需要创建一个最终状态并将其作为与Quit按钮的clicked()信号相关联的转换的目标．我们可以从状态s1,s2,s3中添加一个转换；但是，这看起来像是多余的，并且，我们不得不记住从每个将来新加入的状态添加一个这样的转换．

我们可以通过将状态s1,s2,s3分组取得相同的行为（即点击Quit按钮将退出状态机，无论该状态机处于哪个状态）．这是通过创建一个新的顶层状态并使三个原先的状态成为新状态的孩子．如下图显示了新状态机．



三个原先的状态已经重命名为s11,s12和s13以反映它们现在已经是新的顶层状态s1的孩子．孩子状态隐含地继承它们的父状态的转换．这意味着现在增加一个从状态s1到最终状态s2的转换已经足够了．新加入s1的状态也将自动继承这个转换．

将状态分组的所有工作就是当创始状态时，指定合适的父状态．你也需要指定哪个子状态是初始状态（例如，哪个子状态将是进入父状态时应该处于的状态）．

QState \*s1 = new QState();

QState \*s11 = new QState(s1);

QState \*s12 = new QState(s1);

QState \*s13 = new QState(s1);

s1->setInitialState(s11);

machine.addState(s1);

QFinalState \*s2 = new QFinalState();

s1->addTransition(quitButton, SIGNAL(clicked()), s2);

machine.addState(s2);

QObject::connect(&machine, SIGNAL(finished()), QApplication::instance(), SLOT(quit()));

在本例子中，我们想让当状态机完成后，应用程序退出，因此状态机的finished()信号连接到应用程序的quit()槽．

一个子状态可以覆盖一个继承过来的转换．例如，如下代码添加了一个转换，它有效地造成了当状态机处于状态s12时，Quit按钮将被忽略．

s12->addTransition(quitButton, SIGNAL(clicked()), s12);

一个转换可以将任何状态作为它的目标，例如，目标状态不一定要与源状态处于相同的层次．

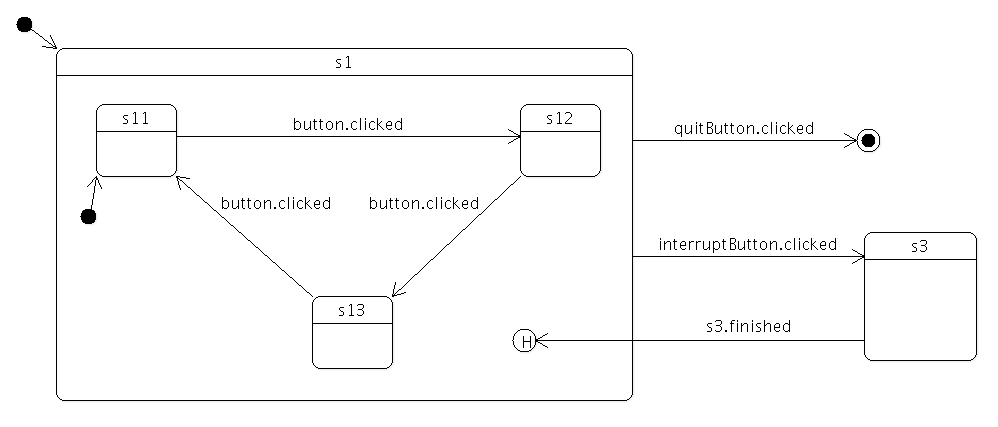
**使用历史状态来保存和恢复当前状态**

假设我们要增加一个“中断”机制到前面提到的例子当中；用户应该能够点击一个按钮使状态机执行一些不相关的任务，任务完成后状态机应该能够恢复到之前执行的任何任务。（例如，返回到旧状态，在此例子中中s11,s12,s13中的一个）。

这样的行为很容易地使用历史状态建模。一个历史状态（QhistoryState对象）是一个伪状态，它代表父状态最后退出时所处的孩子状态。

一个历史状态创建为某个状态的孩子，用于为其记录当前的孩子状态；当状态机在运行时检测到有这个的一个状态存在时，它在父状态退出时自动地记录当前的孩子状态。到该历史状态的一个转变实际上是到状态机之前保存的出乱子状态的转变。状态机自动地“转发”到真正孩子状态的转变。

下图显示了加入了中断机制后的状态机。



下面的代码显示了怎么去实现这种机制；在本例中，我们在进入s3时简单地显示一个信息框，然后通过历史状态立即返回到s1之前的孩子状态中。

QHistoryState \*s1h = new QHistoryState(s1);

QState \*s3 = new QState();

s3->assignProperty(label, "text", "In s3");

QMessageBox \*mbox = new QMessageBox(mainWindow);

mbox->addButton(QMessageBox::Ok);

mbox->setText("Interrupted!");

mbox->setIcon(QMessageBox::Information);

QObject::connect(s3, SIGNAL(entered()), mbox, SLOT(exec()));

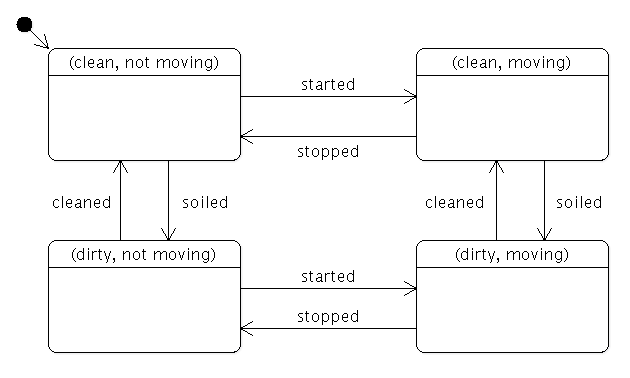
s3->addTransition(s1h);

machine.addState(s3);

s1->addTransition(interruptButton, SIGNAL(clicked()), s3);

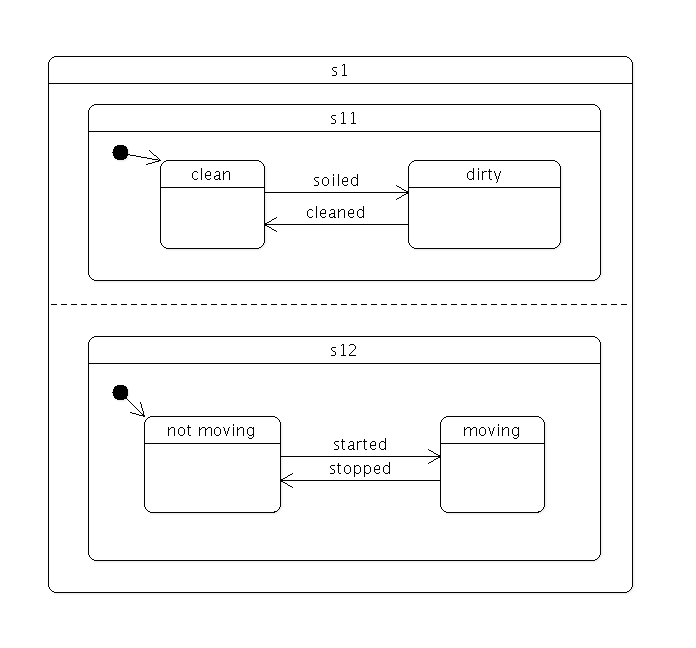
**使用并行状态以避免状态的组合爆发**

假设你想要在一个状态机中建立一些相互排斥的属性。比如说，我们感兴趣的属性是Clean VS Dirty和Moving VS Not moving。需要采用四个互斥的状态和八个转变才能描述该状态机，并能在各个可能的组合中自由的移动。



如果我们增加第三个属性（比如，Red VS Blue），状态的总数将会翻倍，到8个，且如果我们添加第四个属性（比如，Enclosed VS Convertible），状态的总数将再次翻倍到16个。

使用并行状态，状态的总数和转变数会随着属性的不断增加线性地增长，而不是指数地增长。而且，从并行状态中添加或移除状态不会影响它们的兄弟状态。



为了创建一个并行状态组，传递[QState::ParallelStates](http://doc.qt.nokia.com/4.6/qstate.html" \l "ChildMode-enum) 到Qstate构造函数中。

QState \*s1 = new QState(QState::ParallelStates);

*// s11 and s12 will be entered in parallel*

QState \*s11 = new QState(s1);

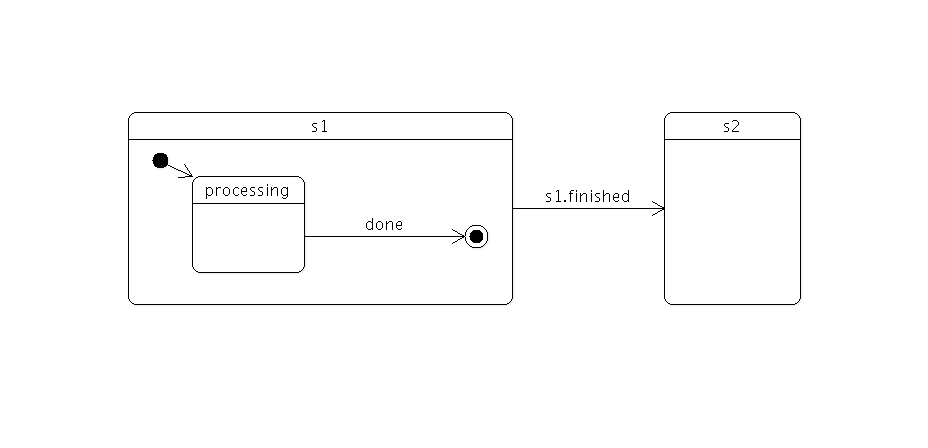
QState \*s12 = new QState(s1);

当一个并行状态组进入时，所有的子状态将会同时进入。每个子状态里的转变正常执行。但是，任何一个子状态可以执行存在于父状态中的一个转变。当这发生时，父状态以及所有的子状态将退出。

状态机框架的并行机制遵循如下一种交错的语义。所有并行操作将以单步，原子地进行，没有事件可以中断并行操作。但是，事件仍然会被顺序地处理，因为状态机本身是单线程的。举个例子：考虑这样的一个情形，有两个转变从相同的状态组中退出，并且它们的（退出）条件同时变为真。在这种情况下，被处理的事件中的后一个将不会产生任何效果，因为第一个事件已经促使状态机从并行状态中退出了。

**检测某个组合状态已经完成**

一个孩子状态可为最终状态（一个[QFinalState](http://doc.qt.nokia.com/4.6/qfinalstate.html)对象）。当进入最终状态时，父状态发出[QState::finished](http://doc.qt.nokia.com/4.6/qstate.html" \l "finished)()信号。下图显示了一个组合状态s1，在进入最终状态之前执行一些处理：



当进入s1的最终状态时，s1会自动地发出finished()。我们使用一个转变来促使这个事件触发一个状态改变：

s1->addTransition(s1, SIGNAL(finished()), s2);

在组合状态中使用最终状态是有用的，当你想隐藏一个组合状态的内部细节时；例如，位于该组合状态之外的世界只需能进入到该状态并在该状态完成了其工作时获得通知。在构建复杂的状态机（深度嵌套）时，这是一种非常强大的抽象和封装机制。（在以上例子中，当然你可以创建一个直接从s1 的done状态开始的一个转变，而不依赖s1的finished()信号，但是，会造成s1的实现细节暴露并依赖它。）。

对于并行状态组，当所有孩子状态进入了最终状态时会发出[QState::finished](http://doc.qt.nokia.com/4.6/qstate.html" \l "finished)()信号。

**无目标转变**

一个转变不需要一个目标状态。无目标的转变可与其他转变一样的方式被触发； 不同之处在于当无目标转变被触发时，它不会造成任何状态的改变。这可以允许你在当状态机处于某个特定状态时，对信号或事件作出响应而不用离开那个状态。例如：

QStateMachine machine;

QState \*s1 = new QState(&machine);

QPushButton button;

QSignalTransition \*trans = new QSignalTransition(&button, SIGNAL(clicked()));

s1->addTransition(trans);

QMessageBox msgBox;

msgBox.setText("The button was clicked; carry on.");

QObject::connect(trans, SIGNAL(triggered()), &msgBox, SLOT(exec()));

machine.setInitialState(s1);

该信息框在每次按钮被点击时显示，但是状态机仍然处于当前状态(s1)。然而，如果目标状态显式地设置为s1，s1会退出并且每次点击的时候进入（例如，会发出[QAbstractState::entered](http://doc.qt.nokia.com/4.6/qabstractstate.html" \l "entered)()和[QAbstractState::exited](http://doc.qt.nokia.com/4.6/qabstractstate.html" \l "exited)()信号）。

**事件，转变和哨卫**

一个[QStateMachine](http://doc.qt.nokia.com/4.6/qstatemachine.html) 运行在自己的事件循环里，对于信号转变（[QSignalTransition](http://doc.qt.nokia.com/4.6/qsignaltransition.html)对象），当它截获了相应地信号，[QStateMachine](http://doc.qt.nokia.com/4.6/qstatemachine.html)会自动地发送一个[QStateMachine::SignalEvent](http://doc.qt.nokia.com/4.6/qstatemachine-signalevent.html)到自身。类似地，对于[QObject](http://doc.qt.nokia.com/4.6/qobject.html)事件转变（[QEventTransition](http://doc.qt.nokia.com/4.6/qeventtransition.html)对象），会发送一个[QStateMachine::WrappedEvent](http://doc.qt.nokia.com/4.6/qstatemachine-wrappedevent.html)。

你可以使用[QStateMachine::postEvent](http://doc.qt.nokia.com/4.6/qstatemachine.html" \l "postEvent)()将自己的事件发送到状态机。

当发送一个自定义的事件到状态机，你一般也拥有一个或更多个自定义的转变，这些转变可以由这种类型的事件触发。 为了创建一个这样的转变，你要创建一个[QAbstractTransition](http://doc.qt.nokia.com/4.6/qabstracttransition.html) 子类并重新实现[QAbstractTransition::eventTest](http://doc.qt.nokia.com/4.6/qabstracttransition.html" \l "eventTest)()方法，在该方法中，你检测某个事件是否与你的事件类型匹配（也可以采用其他的判断规则，如事件对象的属性）。下面我们定义了自已的事件类型，StringEvent，用于向状态机中发送字符串：

struct StringEvent : public QEvent

{

StringEvent(const QString &val)

: QEvent(QEvent::Type(QEvent::User+1)),

value(val) {}

QString value;

};

接下来，我们定义一个转变，仅当事件的字符与某个特定的字符串（一个哨卫转变）匹配时才触发它。

class StringTransition : public QAbstractTransition

{

public:

StringTransition(const QString &value)

: m\_value(value) {}

protected:

virtual bool eventTest(QEvent \*e) const

{

if (e->type() != QEvent::Type(QEvent::User+1)) *// StringEvent*

return false;

StringEvent \*se = static\_cast<StringEvent\*>(e);

return (m\_value == se->value);

}

virtual void onTransition(QEvent \*) {}

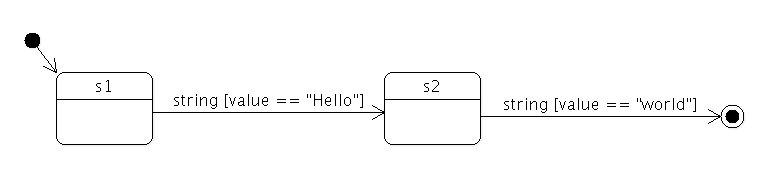
private:

QString m\_value;

};

在eventTest()的重载中，我们首先检测了事件类型是否是我们想要的类型。如果是的，我们将事件转换为一个StringEvent并执行字符串比较操作。

如下是一个使用了自定义事件和转变的状态图：



该状态图的实现代码如下：

QStateMachine machine;

QState \*s1 = new QState();

QState \*s2 = new QState();

QFinalState \*done = new QFinalState();

StringTransition \*t1 = new StringTransition("Hello");

t1->setTargetState(s2);

s1->addTransition(t1);

StringTransition \*t2 = new StringTransition("world");

t2->setTargetState(done);

s2->addTransition(t2);

machine.addState(s1);

machine.addState(s2);

machine.addState(done);

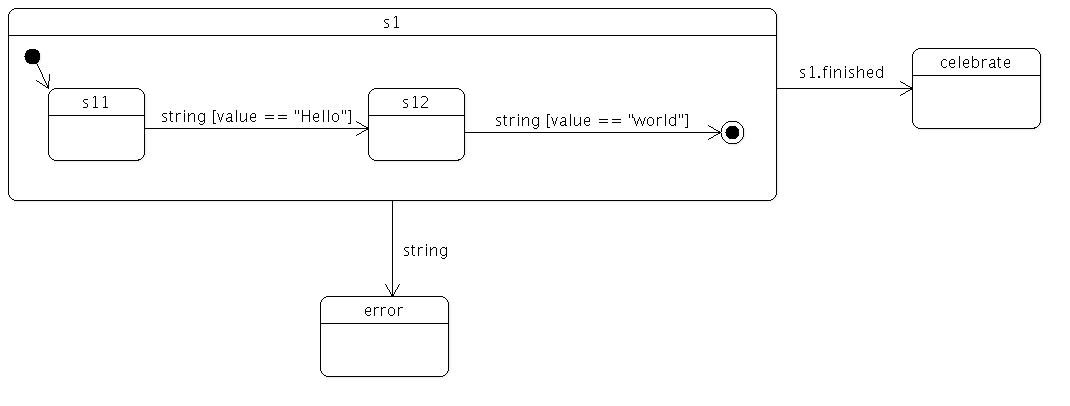
machine.setInitialState(s1);

一旦状态机启动，我们可以将事件发送给它。

machine.postEvent(new StringEvent("Hello"));

machine.postEvent(new StringEvent("world"));

没有被任何相关的转变处理的事件将自动由状态处理。这对于分组状态和提供这样的事件的一个默认处理是有用的；例如，如下状态图：



对于深度嵌套的状态图，你可以添加这样的“后退（fallback）”转变

**使用恢复策略自动地恢复属性**

在一些状态机中，在精力集中在对状态中的属性进行赋值是有用的，而不是当状态不再活跃时恢复它们。如果你知道当状态机进入某个状态时，并且在该状态下没有显式地给属性一个值，属性总是应该恢复到它的初始状态，你可以设置全局的策略为[QStateMachine::RestoreProperties](http://doc.qt.nokia.com/4.6/qstatemachine.html" \l "RestorePolicy-enum)。

QStateMachine machine;

machine.setGlobalRestorePolicy(QStateMachine::RestoreProperties);

当该策略设置了后，状态机会自动地恢复所有的属性。如果它进入了一个状态，而某个给定的属性没有设置，它会首先寻找祖先的层次结构以查看该属性是否已定义。如果是的，该属性会被恢复到最近祖先定义的值。如果不是，它会被恢复到初始值。

如下代码所示：

QStateMachine machine;

machine.setGlobalRestorePolicy(QStateMachine::RestoreProperties);

QState \*s1 = new QState();

s1->assignProperty(object, "fooBar", 1.0);

machine.addState(s1);

machine.setInitialState(s1);

QState \*s2 = new QState();

machine.addState(s2);

比如说，属性fooBar在状态机启动时值为0.0。当机器处于状态s1，属性值会为1.0，因为该状态显示地设置了该属性的值。当该机器处于状态s2，没有显示地定义该属性，因此它会被 隐式地恢复为0.0。

如果我们使用嵌套的状态，父状态为该属性定义了一个值，所有其后裔并没有显示地定义该属性的值。

QStateMachine machine;

machine.setGlobalRestorePolicy(QStateMachine::RestoreProperties);

QState \*s1 = new QState();

s1->assignProperty(object, "fooBar", 1.0);

machine.addState(s1);

machine.setInitialState(s1);

QState \*s2 = new QState(s1);

s2->assignProperty(object, "fooBar", 2.0);

s1->setInitialState(s2);

QState \*s3 = new QState(s1);

这里，s1拥有两个孩子：s2和s3。当进入s2时，属性fooBar的值为2.0，因为该状态显示地定义了该值。当状态机处于状态s3时，该状态没有定义任何值，但是s1定义了属性的值为1.0，因此，这就是将被赋给fooBar的值。

**动画属性赋值**

状态机API与动画API的连接使得当在状态中设置动画属性时，自动地animating属性。比如，我们有如下代码：

QState \*s1 = new QState();

QState \*s2 = new QState();

s1->assignProperty(button, "geometry", QRectF(0, 0, 50, 50));

s2->assignProperty(button, "geometry", QRectF(0, 0, 100, 100));

s1->addTransition(button, SIGNAL(clicked()), s2);

这里，我们定义了用户界面的两个状态，在状态s1中，button小些，在状态s2中，button大些。如果我们点击按钮，从状态s1转换到状态s2，当给定的状态进入时，该按钮的几何属性可以立即设置。但是，如果我们想让转变更为流畅，需要构造一个[QPropertyAnimation](http://doc.qt.nokia.com/4.6/qpropertyanimation.html)对象并将其添加到转变对象中。

QState \*s1 = new QState();

QState \*s2 = new QState();

s1->assignProperty(button, "geometry", QRectF(0, 0, 50, 50));

s2->assignProperty(button, "geometry", QRectF(0, 0, 100, 100));

QSignalTransition \*transition = s1->addTransition(button, SIGNAL(clicked()), s2);

transition->addAnimation(new QPropertyAnimation(button, "geometry"));

为属性添加了一个动画后，属性的赋值不再当进入状态时马上起效。相反地，动画在状态进入时开始播放并平滑地使属性赋值动起来。因为我们没有设置运行的起始值和结束值，这些将隐式地设置。动画的起始值将是动画开始时的当前值。

如果状态机的全局恢复策略设置为[QStateMachine::RestoreProperties](http://doc.qt.nokia.com/4.6/qstatemachine.html" \l "RestorePolicy-enum) ，也可以为恢复属性添加动画。

**检测某个状态下的所有属性**

当动画用于赋值时，一个状态不再定义当状态机进入该状态时的精确值。当动画正在运行时，属性可以拥有任何值，取决于动画。

在一些情况下，当能检测到某个属性被一个状态定义的实际值时是有用的。

比如，我们有如下代码：

QMessageBox \*messageBox = new QMessageBox(mainWindow);

messageBox->addButton(QMessageBox::Ok);

messageBox->setText("Button geometry has been set!");

messageBox->setIcon(QMessageBox::Information);

QState \*s1 = new QState();

QState \*s2 = new QState();

s2->assignProperty(button, "geometry", QRectF(0, 0, 50, 50));

connect(s2, SIGNAL(entered()), messageBox, SLOT(exec()));

s1->addTransition(button, SIGNAL(clicked()), s2);

当button点击后，状态机将转换到状态s2，它会设置按钮的geometry属性，然后弹出一个信息框来提示用户geometry已经改变。

在正常情况下，没有使用动画时，该操作会以预期地方式执行。但是，如果在状态s1到s2的转变中为button的属性geometry定义了一个动画，该动画将在进入s2时启动，但是，在动画结束运行之前，geometry属性并不会到达它定义的值。在这种情况下，在button的geometry属性实际被设置之前，会弹出一个信息框。

为了确保信息框直到geometry达到它的最终值的时候才弹出，我们可以使用状态的propertiesAssigned()信号，当属性被赋予最终的值时，就会发出propertiesAssigned()信号。

QMessageBox \*messageBox = new QMessageBox(mainWindow);

messageBox->addButton(QMessageBox::Ok);

messageBox->setText("Button geometry has been set!");

messageBox->setIcon(QMessageBox::Information);

QState \*s1 = new QState();

QState \*s2 = new QState();

s2->assignProperty(button, "geometry", QRectF(0, 0, 50, 50));

QState \*s3 = new QState();

connect(s3, SIGNAL(entered()), messageBox, SLOT(exec()));

s1->addTransition(button, SIGNAL(clicked()), s2);

s2->addTransition(s2, SIGNAL(propertiesAssigned()), s3);

在该例子中，当button点击时，状态机进入s2，当仍然处于状态s2直到属性geometry被设置为QRect(0, 0, 50, 50)。然后，它会转变到s3。当进入s3时，信息框会弹出。如果转变到s2有一个geometry属性的动画，那么状态机将会处于s2中直到动画完成。如果没有这样的动画，它会设置该属性并立即进入状态s3。

不管什么方式，当状态机处于状态s3，可以保证属性geometry已经被赋予了定义的值。如果全局恢复策略设置为[QStateMachine::RestoreProperties](http://doc.qt.nokia.com/4.6/qstatemachine.html" \l "RestorePolicy-enum),该状态不会发出propertiesAssigned()信号，直到这些也被执行了。

**在动画完成之前某个状态退出了会发生什么**

如果一个状态有属性赋值并且状态的转变过程中为属性设置了动画，状态有可以在动画完成之前退出。这是可能发生的，特别当从状态的转变出来的一些转变不依赖于propertiesAssigned()信号。

状态机API保证一个被状态机赋值的属性：

——拥有显示赋给该属性的一个值

——是当前正被渐进到一个是式地赋予给属性的值。

当一个状态在动画完成之前退出时，状态机的行为取决于转变的目标状态。如果目标状态显式地为属性赋予了一个值，不会采用另外的动作。属性将被赋予被目标状态定义的值。

如果目标状态没有赋予属性任何值，有两种选择：默认的，属性会被赋予它当离开的状态的值。但是，如果设置了全局恢复策略，优先采取这种选择，属性会像平常一样被恢复。

**默认动画**

正如早前所描述的一样，你可以添加动画到转变中以确保目标状态的属性赋值会被渐变。如果你想为某个给定的属性使用一个特定的动画而不管采用什么转变，你可以添加它作为状态机的一个默认的动画。

QState \*s1 = new QState();

QState \*s2 = new QState();

s2->assignProperty(object, "fooBar", 2.0);

s1->addTransition(s2);

QStateMachine machine;

machine.setInitialState(s1);

machine.addDefaultAnimation(new QPropertyAnimation(object, "fooBar"));

当状态机处于状态s2，状态机会为属性fooBar播放默认的动画，因为该属性由s2赋值。注意，显示地设置转变动画比默认动画优先级大。

**专题四　Qt中使用Ｃ++的方式**

先看一个普通的Ｃ++类的定义：

|  |
| --- |
| **#include <string>**  **using std::string;**  **class MyClass**  **{**  **public:**  **MyClass( const string& text );**  **const string& text() const;**  **void setText( const string& text );**  **int getLengthOfText() const;**  **private:**  **string m\_text;**  **}** |

上述类的定义体现了Qt的一些风格：

* 类名的第一个字母为大写，类名包含的单词使用“驼峰”风格分开，也即每个单词的第一个字母为大写。这是Qt类命名的通用方式。
* 所在方法名的第一个字母为小写。单词之间也是采用“驼峰”风格分开。这是Qt方法的命名的通用方式。
* 属性text的getter和setter方法分别命名为text(getter)和setText(setter)。这是Qt命名getter和setter方法的通用方式。

在编程中遵循这些结构化的方式将为你的实际编码节省大量时间。

在Ｑt中，所有的类都继续于QObject类。如果让上述类继承自QObject，那么，QObject将负责该类的一些对象的内存释放，只要将父类传递给该类的对象实例即可。修改后的类定义如下所示：

|  |
| --- |
| **#include <QObject>**  **#include <string>**  **using std::string;**  **class MyClass : public QObject**  **{**  **public:**  **MyClass( const string& text, QObject \*parent = 0 );**  **const string& text() const;**  **void setText( const string& text );**  **int getLengthOfText() const;**  **private:**  **string m\_text;**  **};** |

下面分别说明两种不同类定义下的使用区别：

第一种情况，不使用Ｑt的动态内存管理方案的情况：

|  |
| --- |
| **#include <iostream>**  **int main( int argc, char \*\*argv )**  **{**  **MyClass \*a, \*b, \*c;**  **a = new MyClass( "foo" );**  **b = new MyClass( "ba-a-ar" );**  **c = new MyClass( "baz" );**  **std::cout << a->text() << " (" << a->getLengthOfText() << ")" << std::endl;**  **a->setText( b->text() );**  **std::cout << a->text() << " (" << a->getLengthOfText() << ")" << std::endl;**  **int result = a->getLengthOfText() - c->getLengthOfText();**  **delete a;**  **delete b;**  **delete c;**  **return result;**  **}** |

使用Ｑt的动态内存管理的情况：

|  |
| --- |
| **#include <QtDebug>**  **int main( int argc, char \*\*argv )**  **{**  **QObject parent;**  **MyClass \*a, \*b, \*c;**  **a = new MyClass( "foo", &parent );**  **b = new MyClass( "ba-a-ar", &parent );**  **c = new MyClass( "baz", &parent );**  **qDebug() << QString::fromStdString(a->text())**  **<< " (" << a->getLengthOfText() << ")";**  **a->setText( b->text() );**  **qDebug() << QString::fromStdString(a->text())**  **<< " (" << a->getLengthOfText() << ")";**  **return a->getLengthOfText() - c->getLengthOfText();**  **}** |

说明：在Qt程序中，尽量使用qDebug()来打印信息，它能保证在所有的平台上将信息发送到正确的地方。要想关闭qDebug()，只需在编译时传递宏定义**QT\_NO\_DEBUG\_OUTPUT**即可。

在Ｑt编译中，尽量不要使用STL库中的类，这样使得Qt程序不依赖于第二个框架，所以上述类定义中的std:string可以全部用Qstring来替换，修改后的类定义如下：

|  |
| --- |
| **#include <QString>**  **#include <QObject>**  **class MyClass : public QObject**  **{**  **public:**  **MyClass( const QString& text, QObject \*parent = 0 );**  **const QString& text() const;**  **void setText( const QString& text );**  **int getLengthOfText() const;**  **private:**  **QString m\_text;**  **};** |

另外Qt还对普通的C++进行了一些扩展，如信号、槽以及元对象等，这些内容将在其他专题中介绍。

**专题五　动态对话框的创建**

动态对话框是指在运行时由.ui文件创建的对话框。它不是用uic将.ui文件转化为c++代码，而是使用类QuiLoader在运行时将文件加载。使用示例如下：

|  |
| --- |
| **QUiLoader uiLoader;**  **QFile file("sortdialog.ui");**  **QWidget \*sortDialog = uiLoader.load(&file);**  **if (sortDialog) {**  **...**  **}** |

可以使用QObject::findChild<T>()方法来方法来访问动态对话框中的子组件。例如：

|  |
| --- |
| **QComboBox \*primaryColumnCombo =**  **sortDialog->findChild<QComboBox \*>("primaryColumnCombo");**  **if (primaryColumnCombo) {**  **...**  **}** |

注意，该方法不适用于MSVC6，在MSVC6中，就使用全局函数qFindChild<T>(),它的使用方式与QObject::findChild<T>()一致。另外，QuiLoader类位于一个单独的库中，要在一个Qt应用程序中使用该类，必须在工程文件中添加如下信息：

CONFIG += uitools

**专题六　信号与槽**

信号与槽机制是Qt编程的基础。它使得应用程序开发人员可以将对象绑定到一起而对象之间不需要相互了解。

槽几乎与普通的C++成员函数相同。它们可以是虚函数。它们可以被重载。它们可以是公有的，可以是保护类型的，也可以是私有类型的，它们可以像其他类成员函数一样调用。它们的参数也可以是任何类型。不同之处就是它可以连接到一个信号上，每当信号发出时，它自动地被调用。

connect()语句的使用方式如下所示：

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

信号与槽的连接可以有多种组合方式，如下所示：

* 一个信号可连接到许多槽中。

connect(slider, SIGNAL(valueChanged(int)),

spinBox, SLOT(setValue(int)));

connect(slider, SIGNAL(valueChanged(int)),

this, SLOT(updateStatusBarIndicator(int)));

* 许多信号可以连接到相同的槽中。

connect(lcd, SIGNAL(overflow()),

　this, SLOT(handleMathError()));

connect(calculator, SIGNAL(divisionByZero()),

this, SLOT(handleMathError()));

* 一个信号可以连接到另外一个信号中。

connect(lineEdit, SIGNAL(textChanged(const QString &)),

this, SIGNAL(updateRecord(const QString &)));

* 信号与槽之间的连接可以被解除。

disconnect(lcd, SIGNAL(overflow()),

this, SLOT(handleMathError()));

另外，一般情况下，连接在一起的信号与槽的参数类型和参数顺序应该相同，特别地，信号的参数个数可以比连接到此信号中的槽的参数个数多，多余的参数个数会被忽略。

**信号与槽的实质**

信号与槽实质是用函数指针实现的。当使用emit发送一个信号时，实际上是调用该号。信号是一个函数，其实现代码由moc生成。该函数调用任何与之相连的槽，这些槽的信息是通过拥有槽的对象的元对象获得的。

元对象包含了指向槽的函数指针，以及它们的名字和参数类型。它们同时包含一个可用的信号和信号名字以及参数类型的一个列表。当调用connect时，实际上是通过元对象将槽加入到信号的调用列表中。如果参数匹配，连接就建立起来了。

在匹配参数时，仅检查槽所能接受的参数。这意味着，一个无参数的槽可以与所有的信号匹配。

**专题七　保存和恢复应用程序状态**

在实际应用中，常常需要应用程序能保存程序的状态以及用户的设置，如应用程序显示的大小、位置、背景颜色或用户设置参数等信息，以便下次运行程序时，能保持上次关闭的状态。

Qt提供的QSettings类能很方便地实现保存程序设置的需求。当创始一个QSettings对象时，必须传递公司名或组织名以及应用程序名称，如下所示：

QSettings settings("MySoft", "Star Runner");

如果在应用程序的多个地方用到QSettings, 可以使用[QCoreApplication::setOrganizationName](qcoreapplication.html" \l "organizationName-prop)() 和[QCoreApplication::setApplicationName](qcoreapplication.html" \l "applicationName-prop)()来指定组织名和应用程序名。这样，就可以使用QSettings的默认构造函数了。如下所示：

QCoreApplication::setOrganizationName("MySoft");

QCoreApplication::setOrganizationDomain("mysoft.com");

QCoreApplication::setApplicationName("Star Runner");

...

QSettings settings;

保存一个设置方法如下：

settings.setValue("editor/wrapMargin", 68);

它指定了一个设置的键(QString)和一个与之关联的数据(QVariant)。

读取一个设置的方法如下：

int margin = settings.value("editor/wrapMargin").toInt();

上述语句将返回与键关联的数据，如果没有与该键关联的数据，则返回值为空。

int margin = settings.value("editor/wrapMargin", 80).toInt();

上述语句提供了一个默认值，即当没有与键关联的数据时，返回默认值。

**关于段和键的语法**

可以构造层次化的键值，使用’/’作为分隔符,类似于Unix文件路径,如:

settings.setValue("mainwindow/size", win->size());

settings.setValue("mainwindow/fullScreen", win->isFullScreen());

settings.setValue("outputpanel/visible", panel->isVisible());

如果保存或恢复的设置的键值拥有相同的前缀,那么可以使用beginGroup()和endGroup(),代码如下所示:

settings.beginGroup("mainwindow");

settings.setValue("size", win->size());

settings.setValue("fullScreen", win->isFullScreen());

settings.endGroup();

settings.beginGroup("outputpanel");

settings.setValue("visible", panel->isVisible());

settings.endGroup();

**专题八　事件处理**

事件是由系统或Qt本身产生的,以应对不断发生的情况。事件可以是鼠标事件、键盘事件或窗口显示等。

在Qt中，各种Widget已经封装了对各种事件的响应代码，对外的形式是各种信号。当使用Widget时，我们只需关注信号就可以了，而当自己实现一个Widget时，则需要自己实现一些事件响应代码。

在Qt中，一个事件对象继承自QEvent。Qt处理多达几百种事件，每种事件都用一个枚举类型来表示，如鼠标按下事件：QEvent::MouseButtonPress。

事件是通过函数event()来通知给对象的，继承看QObject，在QWidget的实现中，event()将转发大部分常见的事件类型给特定的事件处理器，如mousePressEvent, keyPressEvent以及paintEvent()。

**安装事件过滤器**

Qt事件模型的一个强大的特性就是一个QObject可以监控另一个QObject对象（该对象甚至不必感知另一对象的存在）。具体做法如下：

* 注册监控对象，即在目标对象上调用installEventFilter()，如：firstNameEdit->installEventFilter(this);
* 在监控者的eventFilter()函数中处理目标对象的事件。

Qt提供了在5个不同的层次上处理和过滤事件的方法：

* 重新实现一个特定的事件处理器。

例如，重新实现一个输入事件处理函数，如：mousePressEvent(), keyPressEvent()等。

* 重新实现QObject::event()。

通过重新实现event()函数，我们可以在事件到达特定的事件处理器之前处理它。不过在实现event()函数时，必须调用基类的event()函数以处理其他的情况。

* 在某个QObject对象上安装一个事件过滤器。

使用installEventFilter()函数，与之相关的目标对象的所有事件将首先被送到监控对象的eventFilter()函数中。

* 在QApplicatioin对象上安装珍上事件过滤器。

如果为qApp注册了一个事件过滤器，那么应用程序中的每个事件都首先被发送到该eventFilter()函数中，然后才转发给其他的事件过滤器。该方法通常用于调试。

* 创建QApplicatoin类的子类，并重新实现notify()函数。

Qt调用QApplication::notify()发送一个事件。重新实现该函数是获取所有被其他事件过滤器处理之前的事件的唯一方式。eventFilter()函数可以有多个，但notify()函数却只有一个。

**QApplication::processEvents()**

在Qt应用程序中，存在一个事件循环，对应着一个事件队列。对于单线程程序来说，只有一个事件队列。一个新事件首先添加到队列中，然后等待被处理。如果某个事件处理的时间有点长，就会影响其他事件的处理时间。此时，应用程序就会对其他事件无响应。为了解决此问题，一个比较简单的方法就是不停地调用QApplication::processEvents()函数。该函数告诉Qt去处理任何末决的事件，然后将控制权返还给调用者。事件上，Qapplication::exec()的简化模型就是在一个while循环中不断调用processEvents()函数。

**专题九　Qt中的容器类**

Qt提供了两种类型的容器类：序列容器和关联容器。例如QVector<T>, QLinkedList<T>和QList<T>属于序列容器，而QMap<K,T>和QHash<K,T>属于关联容器。

Qt提供了通用的算法，对这些容器类执行一些操作，如qSort()对一个序列容器进行排序，而qBinaryFind()在一个排过序的序列容器 内执行二叉树查找。

**序列容器**

QVector<T>是一个类数组的数据结构，与C++中普通的数组不同之外是：一个向量知道本身的大小并可被重置大小。将一个向量的末尾添加一个新的项相当有效，而往其前面或中间插入一个项则代价比较昂贵。

用法：

*在提前知道大小的情况下：*

QVector<double> vect(3);

vect[0] = 1.0;

vect[1] = 0.540302;

vect[2] = -0.416147;

*在大小不可知的情况下：*

QVector<double> vect;

vect.append(1.0);

vect.append(0.540302);

vect.append(-0.416147);

或

vect << 1.0 << 0.540302 << -0.416147;

*遍历一个向量：*

double sum = 0.0;

for (int i = 0; i < vect.count(); ++i)

sum += vect[i];

QLinkedList<T>是一个链表数据结构，它不支持随机访问，但可以很高效的完成插入和删除操作。链表是通过迭代子来访问的。链表的使用如下所示：

QLinkedList<QString> list;

list.append("Clash");

list.append("Ramones");

QLinkedList<QString>::iterator i = list.find("Ramones");

list.insert(i, "Tote Hosen");

QList<T>结合了QVector<T>和QLinkedList<T>的优点，是一个“数组－链表”，它的接口是基于索引的，它是一个最合适的通用容器类。

QStack<T>和QQueue<T>是两个更实用的子类。QStack<T>在向量的基础上增加了push(),pop()和top()方法，而QQueue<T>在链表的基础上增加了enqueue(),dequeue()和head()方法。

说明：对于所有的容器类，类型T可以是基本的数据类型，如整型或双精度型，指针类型或一个拥有默认构造函数，一个拷贝函数和一个赋值操作符的类，这些类包括QByteArray,QDateTime,QRegExp,QString和QVariant。从QObject类中继承的子类不能作为窗口类的项，因为它们的拷贝函数和赋值操作符被禁用了。

解决办法(Trick)：将指向这些对象的指针作为容器类的项，而不是对象本身。

T也可以是一个容器类，但在写法上要注意：

QList<QVector<double> > list;

如上，两个>之间一定要有空格，否则编译会将其解释为>>而导致出错。

**容器的遍历——迭代器**

Java风格的迭代器：只读和读写迭代器。只读迭代器类有QVectorIterator<T>，QLinkedListIterator<T>和QListItrator<T>，相应的读写迭代器在名字中有个Mutable,如QMutableVectorIterator<T>

使用Java风格的迭代器时要注意，迭代子并不直接指向容器中的项，它们可定位在第一项之前，最后一项之后或两者之间，如下所示：

QList<double> list;

... QListIterator<double> i(list);

while (i.hasNext()) {

do\_something(i.next());

}

注：在Java风格的只读迭代器中，可以直接对容器进行操作，无需一份拷贝，这些细节已经由相应的类自动完成了，如：

QListIterator<int> i(splitter->sizes());

while (i.hasNext()) {

do\_something(i.next());

}

STL风格的迭代器；

只读：QVector<T>::iterator， 读写： QVector<T>::const\_iterator。

如：

QList<int> list = splitter->sizes();

QList<int>::const\_iterator i = list.begin();

while (i != list.end()) {

do\_something(\*i);

++i;

}

注意，在STL风格下，我们只能对容器的拷贝进行迭代。

//错误写法：

while (i != splitter->sizes().end()) {

do\_something(\*i);

++i;

}

**隐式共享**

由于存在隐式共享，拷贝一个容器类代价并不高，与复制一个指针的代价差不多。仅当某个容器的拷贝内容发生变化时才进行真正的拷贝，有点像Linux中父过程与子进程之间的“写时复制”共享数据。隐式共享鼓励对象通过传值的方式返回。

Qt中所有的容器类以及许多其他的类都使用了隐式共享，包括QByteArray, Qbrush, AFont, QImage, QPixmap和QString。这些类通过传值的方式传递时非常高效，不管是作为函数参数或是作为函数返回值。

隐式共享保证了数据只有被修改时才会进行真正的复制。所以在Qt中，尽管使用只读形式的接口。

**关联容器**

Qt提供了两个主要的关联容器类：QMap<K, T>和QHash<K, T>。

访问方式如下代码所示：

*写*

QMap<QString, int> map;

map.insert("eins", 1);

map.insert("sieben", 7);

map.insert("dreiundzwanzig", 23);

或

map["eins"] = 1;

map["sieben"] = 7;

map["dreiundzwanzig"] = 23;

*读*

int val = map.value("dreiundzwanzig");

或

int seconds = map.value("delay", 30);带默认值 。

QMultiMap是一种1对多的映射容器。

QHash与QMap的接口类似，不过它是无序的，它也有一个变体QMultiHash。

**通用算法**

<QtAlgorithms>包含了一些通用的全局模板函数，实现了操作于容器类数据的通用算法。

如qFind(), qCopy(), qFill(), qSort()等。

**类容器类**

与容器类有许多共性的三个类是：QString , QByteArray, QVariant。

QString的值是16位Unicode值，包含ASCII和Latin-1。QString可认为是一个元素为QChar类型的向量，它可嵌入’\0’，length()函数返回的值包含嵌入的’\0’。QString支持+和+=连接字符串，也可以使用append()方式。也可以使用sprintf()方法连接不同的字符串，如：

str.sprintf("%s %.1f%%", "perfect competition", 100.0);

显示地将一个const char\*型字符串转换为QString类型的方法如下：

const char\* str = “This is a test”;

QString qstr = (QString)str;

或qstr = QStirng::fromAscii(str)或qstr = QString::fromLatin1();

将QString转变为const char\*型的步骤如下：

1. 先使用toAscii()或toLatin1()返回一个QByteArray类型的对象
2. 使用QByteArray::data()或 QByteArray::constData()将其转换为const char\*型。

**QString与中文显示问题**

QString本身是支持中文显示的，它默认是支持unicode的。要使QString能正常地显示中文，必须传递合适的unicode。有两种方法：

1. 利用QTextCodec的toUnicode()方法。

如： w.setWindowTitle(QTextCodec::codecForName("gb2312")->toUnicode("这是一个测试！"));

1. 利用QTextCodec的静态方法，如setCodecForCStrings等。

QTextCodec::setCodecForCStrings(QTextCodec::codecForName("GB2312"));

优先选择第二种方法。

**专题十　输入输出**

QIODevice类是对所有读和写一段字节块的一个抽象。Qt包含了下面的一些子类：

| QFile | 访问本地文件系统中的文件或嵌入的资源。 |
| --- | --- |
| QTemporaryFile | 创建或访问本地文件系统中的临时文件。 |
| QBuffer | 从一个QByteArrary中读数据或将数据写入到一个QByteArray中。 |
| QProcess | 运行外部程序并处理进程间的通信。 |
| QTcpSocket | 使用TCP协议传输一个数据流。 |
| QUdpSocket | 通过网络发送或接收UDP数据流。 |

其中，QProcess,QTcpSocket, QUdpSocket是顺序文件，只能顺序访问，而QFile, QTemporaryFile, QBuffer是随机访问文件，可随机访问，可使用QIODevice::seek()来重定位文件指针。

另外，Qt也提供了两个更高级的流处理类，可用于向任何QIODevice设备中读或写数据。其中QDataStream用于读写二进制数据，而QTextStream用于读写文本数据。

**读写文本**

QTextStream负责处理Unicode与系统本地编码格式或任何其他编码格式的转换。QTextStream使用16-位的QChar类型作为数据的基本单元。另外，QTextStream也支持c++的基本数据类型。

使用例子如下所示：

QFile file("sf-book.txt");

if (!file.open(QIODevice::WriteOnly)) {

cerr << "Cannot open file for writing: "

<< qPrintable(file.errorString()) << endl;

return;

}

QTextStream out(&file);

out << "Thomas M. Disch: " << 334 << endl;

注意：写数据可能要容易些，但读数据就要复杂些，如

out << "Norway" << "Sweden";如果使用in >> str1 >> str2;读的话，结果却是str1=”Norway Sweden”，而str2为空。因此，读数据必须使用专门的程序解析，QTextStream可以逐字符(QChar)地读取。

可以设置字符编码，如：

stream.setCodec("UTF-8");

QTextStream也支持一些流操作标志，如：

out << showbase << uppercasedigits << hex << 12345678;

或

out.setNumberFlags(QTextStream::ShowBase

| QTextStream::UppercaseDigits);

out.setIntegerBase(16);

out << 12345678;

**读写二进制文件**

QDataStream用于读写原始数据文件。在读写文件中，主要涉及两个问题：1.数据类型大小。2.Qt版本。

数据类型问题：确保读和写使用相同的数据类型。比如，对于整数，最好使用qint8, qint16和qint32，qint64，它们对于不同平台，数据类型大小都是一样的。避免使用int , short , long等数据类型，它们在不同平台上大小可能不同。

Qt版本：不同的Qt版本使用的二进制数据的编码可以不同。

QDataStream可序列化所有基本的C++类型和大部分Qt类型，如QColor, QList, QString等。要使得QDataStram能序列化一个自定义的类型，必须提供相应的<<和>>操作符重载函数。代码例子如下：

|  |
| --- |
| struct ColorText  {  QString text;  QColor color;  };  QDataStream &operator<<( QDataStream &stream, const ColorText &data )  {  stream << data.text << data.color;  return stream;  }  QDataStream &operator>>( QDataStream &stream, ColorText &data )  {  stream >> data.text;  stream >> data.color;  return stream;  }  QList<ColorText> list;  ColorText data;  data.text = "Red";  data.color = Qt::red;  list << data;  ...  QFile file( "test.dat" );  if( !file.open( QIODevice::WriteOnly ) )  return;  QDataStream stream( &file );  stream.setVersion( QDataStream::Qt\_4\_2 );  stream << list;  file.close();  QList<ColorText> list;  QFile file( "test.dat" );  if( !file.open( QIODevice::ReadOnly ) )  return;  QDataStream stream( &file );  stream.setVersion( QDataStream::Qt\_4\_2 );  stream >> list;  file.close();  foreach( ColorText data, list )  qDebug() << data.text << "("  << data.color.red() << ","  << data.color.green() << ","  << data.color.blue() << ")"; |

**访问目录**

QDir类可用于浏览路径中的目录和文件。QDir提供了一些静态方法，可以很容易地浏览文件系统。如：

* QDir::current()返回应用程序当前所在的工作目录。
* QDir::home()返回用户的主目录。
* QDir::root()返回根用户目录。
* QDir::temp()返回临时文件目录。
* QDir::drives()返回一个包含了QFileInfo对象的列表。

QFileInfo类提供了访问文件和目录信息的方法，如：

* isDir(), isFile(), isSymLink()：判断文件是否属于目录，一般文件或符号链接文件。
* dir(), absoluteDir():返回相对目录或绝对目录。
* exists();判断文件对象是否存在。
* isHidden(), isReadable, isWritable()和isExecutable():判断一个文件是否隐藏、可读、可写或可执行。
* fileName():返回文件名。
* filePath():返回文件名（包含路径）
* absoluteFilePath():返回包含绝对路径的文件名。
* completeBaseName(), completeSuffix()：返回文件名和文件名后缀。

例如如下：

foreach( QFileInfo drive, QDir::drives() )

{

qDebug() << "Drive: " << drive.absolutePath();

QDir dir = drive.dir();

dir.setFilter( QDir::Dirs );

foreach( QFileInfo rootDirs, dir.entryInfoList() )

qDebug() << "" << rootDirs.fileName();

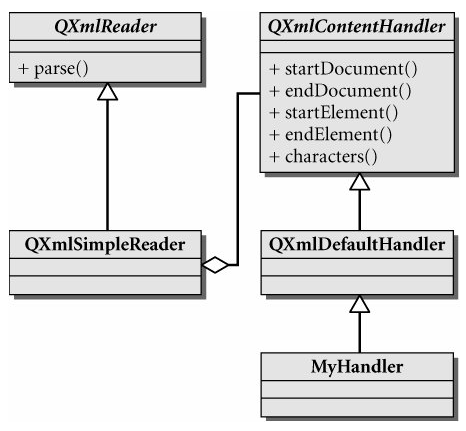
}

**专题十一　XML文件处理**

Qt的XML模块包含了两种处理xml的API，分别是SAX和DOM。SAX提供了一种低层的基于事件驱动的方式解析xml文件，而DOM提供了一种高层的接口用于解析xml文件。不同之处在于DOM受制于内存的大小，一次不能解析文件大小太大的xml文件，因为它将xml文件一次性地读入内存中。

**SAX方式解析XML文件**

SAX方式解析XML是基于事件方式编程的。执行的流程完全取决于读入的数据。使用该方式的主要步骤是创建一个XML读取器以及一个处理器，然后将两者关联起来，之后调用parse()，涉及的各方用类图表示如下：



代码示例如下：

|  |
| --- |
| myhandler.h |
| #ifndef MYHANDLER\_H  #define MYHANDLER\_H  #include <QXmlDefaultHandler>  class QString;  class MyHandler : public QXmlDefaultHandler  {  public:  explicit MyHandler();  signals:  public slots:  public:  bool startDocument();  bool startElement( const QString & namespaceURI,  const QString & localName,  const QString & qName,  const QXmlAttributes & atts);  bool characters(const QString& text);  bool endElement( const QString & namespaceURI,  const QString & localName,  const QString & qName );  private:  QString indent;  };  #endif // MYHANDLER\_H |

|  |
| --- |
| myhandler.cpp |
| #include "myhandler.h"  #include <QObject>  #include <QTextStream>  #include <QDebug>  MyHandler::MyHandler():QXmlDefaultHandler()  {  }  bool MyHandler::startDocument() {  indent = "";  return TRUE;  }  bool MyHandler::characters(const QString& text) {  QString t = text;  qDebug() << t.remove('\n');  return TRUE;  }  bool MyHandler::startElement( const QString&,  const QString&,  const QString& qName,  const QXmlAttributes& atts) {  QString str = QString("\n%1\\%2").arg(indent).arg(qName);  qDebug() << str;  if (atts.length()>0) {  QString fieldName = atts.qName(0);  QString fieldValue = atts.value(0);  qDebug() << QString("(%2=%3)").arg(fieldName).arg(fieldValue);  }  qDebug() << "{";  indent += " ";  return TRUE;  }  bool MyHandler::endElement( const QString&,  const QString& ,  const QString& ) {  indent.remove( 0, 4 );  qDebug() << "}";  return TRUE;  } |

|  |
| --- |
| main.cpp |
| #include <QApplication>  #include <QFile>  #include <QXmlInputSource>  #include "myhandler.h"  #include <QDebug>  int main(int argc, char \*argv[])  {  QApplication a(argc, argv);  QFile xmlFile( ":/test.xml" );  QXmlInputSource source( &xmlFile );  if (!xmlFile.open(QFile::ReadOnly | QFile::Text)) {  return -1;  }  MyHandler handler;  QXmlSimpleReader reader;  reader.setContentHandler( &handler );  if( reader.parse( source ) )  qDebug()<<"File loaded!";  return a.exec();  } |

**DOM方式解析XML文件**

DOM方式只适合解析小的xml文件，它是基于SAX之上的一种高层解析XML方法。使用方式见如下代码所示：

|  |
| --- |
| #include <QtCore/QCoreApplication>  #include <QDomDocument>  #include <QFile>  #include <QDebug>  void parseFolderElement(const QDomElement &element)  {  QDomElement child = element.firstChildElement();  while (!child.isNull()) {  if (child.tagName() == "folder") {  parseFolderElement(child);  } else if (child.tagName() == "bookmark") {  QString title = child.firstChildElement("title").text();  if (title.isEmpty())  title = QObject::tr("Folder");  qDebug()<<"title="<<title;  } else if (child.tagName() == "separator") {  qDebug()<<"processing separator";  }  child = child.nextSiblingElement();  }  }  void parseXmlByDom(QIODevice \*device)  {  QDomDocument domDocument;  QString errorStr;  int errorLine;  int errorColumn;  if (!domDocument.setContent(device, true, &errorStr, &errorLine,  &errorColumn)) {  qDebug()<<"Parse error at line "  <<errorLine  <<", column "  << errorColumn  <<":\n"  <<errorStr;  return;  }  QDomElement root = domDocument.documentElement();  if (root.tagName() != "xbel") {  qDebug()<<QString( "The file is not an XBEL file.");  return;  } else if (root.hasAttribute("version")  && root.attribute("version") != "1.0") {  qDebug()<<QString("The file is not an XBEL version 1.0 file.");  return;  }  QDomElement child = root.firstChildElement("folder");  while (!child.isNull()) {  parseFolderElement(child);  child = child.nextSiblingElement("folder");  }  }  int main(int argc, char \*argv[])  {  QCoreApplication a(argc, argv);  QFile f("frank.xbel");  if( ! f.open(QIODevice::ReadOnly) )  {  qDebug()<<"failed to open file";  return -1;  }  parseXmlByDom(&f);  return a.exec();  } |

**专题十二　QProcess**

QProcess类用于启动另一个外部应用程序，并可与之进行通信。该类的运行方式是异步的。QProcess类在外部进程拥有数据或完成时发送一个信号以通知相关的对象。

QProcess的使用模板如下：

|  |
| --- |
| class XXX : public XXXX  {  Q\_OBJECT  …  private slots:  void errorHandler(QProcess::ProcessError);  void catchOutput();  void catchError();  void stateMonitor(QProcess::ProcessState *newState*);  void slotStarted();  void slotFinished(int, QProcess::ExitStatus);  …  private:  QProcess p;  …  };  XXX::XXX(…)  {  …  connect(&p, SIGNAL(started()), SLOT(slotStarted()));  connect(&p, SIGNAL(finished(int, QProcess::ExitStatus)), SLOT(slotFinished(int, QProcess::ExitStatus));  connect(&p, SIGNAL(error(QProcess::ProcessError)), SLOT(errorHandler (QProcess::ProcessError)));  connect(&p, SIGNAL(readyReadStandardError()), SLOT(catchError()));  connect(&p, SIGNAL(readyReadStandardOutput()), SLOT(catchOutput()));  …  }  XXX:: slotStarted()  {  qDebug()<<”process has started”;  }  XXX:: slotFinished(int exitcode, QProcess::ExitStatus status)  {  if (exitStatus == QProcess::CrashExit) {  …;  } else if (exitCode != 0) {  …;  } else {  …;  }  }  XXX:: errorHandler(QProcess::ProcessError error)  {  if (error == QProcess::FailedToStart) {  …  }  else if( error == QProcess:: Crashed )  {  …  }  else if( error == QProcess:: Timedout )  {  …  }  else if( error == QProcess:: WriteError )  {  …  }  else if( error == QProcess::ReadError )  {  …  }  else if( error == QProcess:: UnknownError )  {  …  }  }  XXX:: catchError()  {  QByteArray newData = process.readAllStandardError();  …  }  XXX::catchOutput()  {  QByteArray newData = process.readAllStandardOutput();  …  } |
|  |

**专题十三　多线程编程**

在Qt应用程序启动后，实际上已经产生了一个线程，叫主线程。主线程通常也称为GUI线程，因为所有的组件和其他用户界面对象都由该线程处理。

通常主线程由一个事件循环和在此线程中创建的一系列对象组成。通过继承QThread类，我们可以创建一个新的线程，拥有线程自己的事件循环和在此线程中创建的一系列对象。在一个Qt应用程序中，实现多线程非常容易：只需创建一个Thread的子类并在该子类中实现run()函数。一个简单的多线程例子如下：

|  |
| --- |
| 创建一个从QThread继承过来的子类 |
| class TextThread : public QThread  {  public:  TextThread( const QString &text );  void run();  private:  QString m\_text;  }; |

|  |
| --- |
| 实现线程子类的相关方法 |
| bool stopThreads = false;  TextThread::TextThread( const QString &text ) : QThread()  {  m\_text = text;  }  void TextThread::run()  {  while( !stopThreads )  {  qDebug() << m\_text;  sleep( 1 );  }  } |

|  |
| --- |
| 使用该线程类 |
| int main( int argc, char \*\*argv )  {  QApplication app( argc, argv );  TextThread foo( "Foo" ), bar( "Bar" );  foo.start();  bar.start();  QMessageBox::information( 0, "Threading", "Close me to stop!" );  stopThreads = true;  foo.wait();  bar.wait();  return 0;  } |

**专题十三　多线程中数据的并发访问与保护**

在多线程编程中，不可避免地要对一些共享的数据进行访问。由于线程之间对共享数据的访问是独立的，任何一个线程都可对共享数据进行访问和修改，且它们之间是异步并发进行的，特别是当需要对共享数据进行修改时，就会产生数据不一致的问题，为此，需要一种保护机制，来限制线程之间的并发访问，以保护共享的数据。

在Qt多线程应用程序中，需要对多个线程进行同步。Qt提供了如下几个同步类：QMutex, QReadWriteLock, QSemaphore以及QWaitCondition。

**QMutex**

QMutex类提供了一个保护变量或一个代码区的方法，使得每次仅有一个线程能对其进行访问。该类提供了lock()方法用于锁位mutex,以及unlock()函数用于解锁。另外，还提供了tryLock()方法，它与lock()方法的一个重要区别在于如果mutex已经被锁定，它将立即返回而不是阻塞。代码示例如下：

|  |
| --- |
| void Thread::run()  {  forever {  **mutex.lock();**  if (stopped) {  stopped = false;  mutex.unlock();  break;  }  **mutex.unlock();**  cerr << qPrintable(messageStr);  }  cerr << endl;  } |

Qt同时也提供了一个帮助类QMutexLocker用于简化mutex的操作，代码示例如下：

|  |
| --- |
| void Thread::run()  {  forever {  {  **QMutexLocker locker(&mutex);**  if (stopped) {  stopped = false;  break;  }  }  cerr << qPrintable(messageStr);  }  cerr << endl;  } |

**QReadWriteLock**

在多个线程读取共享数据而仅有一个线程要修改共享数据的情况下，可以使用QReadWriteLock类，它可以保证并发对数据的读操作而不会对性能产生负面影响。代码示例如下：

|  |
| --- |
| void ReaderThread::run()  {  ...  **lock.lockForRead();**  access\_data\_without\_modifying\_it(&data);  **lock.unlock();**  ...  } |

同样，Qt也提供了一个相应的类QReadLocker和QWriteLocker来简化QReadWriteLock的操作。

**QSemaphore**

QSemaphore是一种更通用的mutex，除了读/写锁外，信号量还可用于监视一定数目的相同资源。如下的代码片段显示了QSemaphore和QMutex之间的对应关系：

|  |
| --- |
| QSemaphore semaphore(1); | QMutex mutex;  semaphore.acquire(); | mutex.lock();  semaphore.release(); | mutex.unlock(); |

通过传递1构造函数，表示该信号量控制单个资源。信号量的一大优点就是通过传递一个大于1的数，我们可以调用acquire()函数多次以获得许多资源，代码示例如下：

|  |
| --- |
| QSemaphore s( 10 );  s.acquire(); // s.available() = 9  s.acquire(5); // s.available() = 4  s.release(2); // s.available() = 6  s.release(); // s.available() = 7  s.release(5); // s.available() = 12  **s.tryAcquire(15); // s.available() = 12** |

**线程之间的通信**

除了通过共享数据来在多个线程之间进行通信外，还可以使用信号与槽的机制实现线程之间的通信。

**多线程编程中注意点**

1. 子QObject必须在它的父线程中创建。

特别地，在非主线程中创建的对象不能将QThread对象作为它们的父类，因为线程对象是在另一个线程中创建的。

1. 必须先删除在一个线程中创建的所有QObject对象，然后删除线程对象本身。
2. QObject必须在创建它的线程中删除。

非GUI的QObject子类，如QTimer, QProcess以及网络类都是可重入的。可用在任何线程中，只要该线程有一个事件循环。对于非主线程，进入事件循环是通过调用QThread::exec()函数。

**专题十四　QThreadPool类和QtConcurrent**

一、QThreadPool类

　 QThreadPool管理一组线程。它负责管理和回收单个QThread对象以减少程序中线程创建的开销。每个Qt应用程序都有一个全局的QThreadPool对象，可通过方法globalInstance()获得。为了调用QThreadPool中的一个线程，需要提供一个从QRunnable继承过来的类，并实现其中的run方法。然后创建一个该类的对象，传递给QThreadPool::start()方法。代码片断如下：

|  |
| --- |
| **class HelloWorldTask : public QRunnable**  **{**  **void run()**  **{**  **qDebug() << "Hello world from thread" << QThread::currentThread();**  **}**  **}**  **HelloWorldTask \*hello = new HelloWorldTask();**  ***// QThreadPool takes ownership and deletes 'hello' automatically***  **QThreadPool::globalInstance()->start(hello);** |

默认情况下，　QThreadPool自动删除QRunnable对象。使用QRunnable::setAutoDelete()方法可以改变该默认行为。QThreadPool支持在QRunnable::run方法中通过调用tryStart(this)来多次执行相同的QRunnable。当最后一个线程退出run函数后，如果autoDelete启用的话，将删除QRunnable对象。在autoDelete启用的情况下，调用start()方法多次执行同一QRunnable会产生竞态，就避免这样做。

　　那些在一定时间内会使用的线程将会过期。默认的过期时间是30秒。可通过setExpiryTimeout()方法来设置。设置一个负数的超时值代表禁用超时机制。方法maxThreadCount()可以查询可使用的最大线程数，你也可以设置最大的线程数。activeThreadCount反应的是当前正在被使用中的线程数个数。reserveThread函数保留某个线程为外部使用，releaseThread释放该线程，这样就可以被再次使用。

二、QtConcurrent命名空间

QtConcurrent命名空间里提供了一些高级API, 利用这些API可以编写多线程程序，而不用直接使用比较低级的一些类，如mutext, lock, wait condition以及semaphore等。使用QtConcurrent命令空间的API编写的程序会根据处理器的数目自动地调整线程的个数。QtConcurrent包含了用于并行列表处理的函数式编程，包含实现共享内存系统的MapReduce和FilterReduce，　以及管理GUI应用程序中异步计算的类。相关的类说明如下：

|  |  |
| --- | --- |
| [QtConcurrent::map](qtconcurrentmap.html" \l "map)**()** | **applies a function to every item in a container, modifying the items in-place** |
| [QtConcurrent::mapped](qtconcurrentmap.html" \l "mapped)**()** | is like map(), except that it returns a new container with the modifications |
| [QtConcurrent::mappedReduced](qtconcurrentmap.html" \l "mappedReduced)**()** | is like mapped(), except that the modified results are reduced or folded into a single result. |
| [QtConcurrent::filter](qtconcurrentfilter.html" \l "filter)**()** | l items from a container based on the result of a filter function. |
| [QtConcurrent::filtered](qtconcurrentfilter.html" \l "filtered)**()** | is like filter(), except that it returns a new container with the filtered results |
| [QtConcurrent::filteredReduced](qtconcurrentfilter.html" \l "filteredReduced)**()** | is like filtered(), except that the filtered results are reduced or folded into a single result |
| [QtConcurrent::run](qtconcurrentrun.html" \l "run)**()** | runs a function in another thread. |
| [QFutureIterator](qfutureiterator.html) | allows iterating through results available via [QFuture](qfuture.html). |
| [QFutureWatcher](qfuturewatcher.html) | allows monitoring a [QFuture](qfuture.html) using signals-and-slots. |
| [QFutureSynchronizer](qfuturesynchronizer.html) | is a convenience class that automatically synchronizes several QFutures. |

代码实例：

|  |
| --- |
| **using namespace QtConcurrent;**  **void hello(QString name)**  **{**  **qDebug() << "Hello" << name << "from" << QThread::currentThread();**  **}**  **int main(int argc, char \*\*argv)**  **{**  **QApplication app(argc, argv);**  **QFuture<void> f1 = run(hello, QString("Alice"));**  **QFuture<void> f2 = run(hello, QString("Bob"));**  **f1.waitForFinished();**  **f2.waitForFinished();**  **return app.exec();**  } |

**专题十五　QT插件开发（-）**

在QT插件开发中，有两种类型：一种是扩展Qt本身的功能，另一种是扩展Qt应用程序本身的功能。本专题将探讨第二种，即通过插件来扩展Qt应用程序本身的功能。

**一、开发概述**

当我们创建一个插件时，需要声明一个插件接口，这是一个只包含纯虚函数的C++类。所有的插件将会实现该接口定义的方法 。该类将以共享库的形式存在，应用程序在运行时动态加载。要使一个Qt应用程序具备插件扩展功能，具体步骤如下：

1. 定义一组接口类（这些类只包含纯虚函数）
2. 通过宏 [Q\_DECLARE\_INTERFACE](http://qt-project.org/doc/qt-5/qtplugin.html#Q_DECLARE_INTERFACE)() 告诉Qt元对象系统这些接口相关信息
3. 使用类[QPluginLoader](http://qt-project.org/doc/qt-5/qpluginloader.html) 加载这些插件。
4. 使用 [qobject\_cast](http://qt-project.org/doc/qt-5/qobject.html#qobject_cast)()来测试某个插件是否实现了相应接口。

编写插件的步骤如下：

1. 声明一个插件类，该类从上述声明的接口以及**QObject**类派生。
2. 使用宏[Q\_INTERFACES](http://qt-project.org/doc/qt-5/qobject.html#Q_INTERFACES)()告诉Qt元对象系统这些接口信息。
3. 使用宏[Q\_PLUGIN\_METADATA](http://qt-project.org/doc/qt-5/qtplugin.html#Q_PLUGIN_METADATA)()导出该插件。
4. 使用合适的.pro文件构建插件。

二、开发实例

本例将实现一个简单的插件，来演示插件的编写过程。本插件称为**EchoPlugin**。它将实现**EchoInterface**接口，该接口只包含一个纯虚函数**echo**，接受一个QString类型的参数。

2.1 EchoInterface类的定义

|  |
| --- |
| class EchoInterface  {  public:  virtual ~EchoInterface() {}  virtual **[QString](http://qt-project.org/doc/qt-5/qstring.html)** echo(const **[QString](http://qt-project.org/doc/qt-5/qstring.html)** &message) = 0;  };  #define EchoInterface\_iid "org.qt-project.Qt.Examples.EchoInterface"  Q\_DECLARE\_INTERFACE(EchoInterface, EchoInterface\_iid) |

通过宏Q\_DECLARE\_INTERFACE 使Qt元对象函数知道该接口。这样就可以在运行时动态识别实现了该接口的插件。第二个参数主要是标识该接口，保持其唯一性。

2.2 EchoPlugin类的定义

该类是插件类，它同时从上述接口以及QObject类派生。宏Q\_INTERFACES告诉Qt它实现的是哪个接口。如果实现多个接口，则用空格隔开。宏Q\_PLUGIN\_METADATA包含了插件的IID以及指向echoplugin.json。

|  |
| --- |
| class EchoPlugin : public **[QObject](http://qt-project.org/doc/qt-5/qobject.html)**, EchoInterface  {  Q\_OBJECT  Q\_PLUGIN\_METADATA(IID "org.qt-project.Qt.Examples.EchoInterface" FILE "echoplugin.json")  Q\_INTERFACES(EchoInterface)  public:  **[QString](http://qt-project.org/doc/qt-5/qstring.html)** echo(const **[QString](http://qt-project.org/doc/qt-5/qstring.html)** &message);  };  **[QString](http://qt-project.org/doc/qt-5/qstring.html)** EchoPlugin::echo(const **[QString](http://qt-project.org/doc/qt-5/qstring.html)** &message)  {  return message;  } |

2.3 加载插件

|  |
| --- |
| bool EchoWindow::loadPlugin()  {  **[QDir](http://qt-project.org/doc/qt-5/qdir.html)** pluginsDir(qApp->applicationDirPath());  #if defined(Q\_OS\_WIN)  if (pluginsDir.dirName().toLower() == "debug" || pluginsDir.dirName().toLower() == "release")  pluginsDir.cdUp();  #elif defined(Q\_OS\_MAC)  if (pluginsDir.dirName() == "MacOS") {  pluginsDir.cdUp();  pluginsDir.cdUp();  pluginsDir.cdUp();  }  #endif  pluginsDir.cd("plugins");  foreach (**[QString](http://qt-project.org/doc/qt-5/qstring.html)** fileName, pluginsDir.entryList(**[QDir](http://qt-project.org/doc/qt-5/qdir.html)**::Files)) {  **[QPluginLoader](http://qt-project.org/doc/qt-5/qpluginloader.html)** pluginLoader(pluginsDir.absoluteFilePath(fileName));  **[QObject](http://qt-project.org/doc/qt-5/qobject.html)** \*plugin = pluginLoader.instance();  if (plugin) {  echoInterface = qobject\_cast<EchoInterface \*>(plugin);  if (echoInterface)  return true;  }  }  return false;  } |

专题 Qt编程技巧集锦

1. 检测内存分配是否成功。

int \*types = new int [typeNames.count() + 1];

**Q\_CHECK\_PTR**(types);

1. 在类定义中（头文件），私有变量（指针类型）涉及的类一般使用前向声明而不用包含相关类的头文件，这样会使编译更快些。
2. qobject\_cast<T>函数基于moc产生的元信息执行一个动态的转换。它返回一个所请求的QObject子类的指针，或０（如果对象不能转换为指定的类型）。与标准Ｃ++的dynamic\_cast<T>不同，它在整个动态库内都工作正常，而dynamic\_cast<T>则在有些编译器上不能。

元对象系统



# define METHOD(a) "0"#a

# define SLOT(a) "1"#a

# define SIGNAL(a) "2"#a

struct QmetaObject{

…

struct { // private data

const QMetaObject \*superdata;

const char \*stringdata;//class Name

const uint \*data;

const void \*extradata;

} d;

}

struct QMetaObjectPrivate

{

int revision;

int className;

int classInfoCount, classInfoData;

int methodCount, methodData;

int propertyCount, propertyData;

int enumeratorCount, enumeratorData;

int constructorCount, constructorData; //since revision 2

int flags; //since revision 3

int signalCount; //since revision 4

// revision 5 introduces changes in normalized signatures, no new members

…

}

QPointer　　Ｑt中的指针

QWeakPointer