11 Qt 模板库

Qt 模板库（QT Template Library 简称 QTL）是一套提供对象容器的模板。如果你的编译器没有适当的 STL（标准模板库）可用，QTL 将被代替使用。QTL 提供了对象的链表、对象的矢量（动态数组）、从一个类型到另一个类型的映射（或称为字典）和相关的迭代器和算法。一个容器是包含和管理其它对象的一个对象，并且提供迭代器对被包含的对 象进行访问。 Qt 模板类说明如表 2。

表 2 Qt 模板类说明

模板类名称	说明
QMap	提供基于值的一个字典的模板类
QMapConstIterator	QMap 的常量迭代器
QMapIterator	QMap 的迭代器
QPair	提供基于值的一对元素的模板类
QValueList	提供基于值的一个双向链表的模板类
QValueListConstIterator	QValueList 的常量迭代器
QValueListIterator	QValueList 的迭代器
QValueStack	提供基于值的一个堆栈模板类
QValueVector	提供基于值的一个动态数组模板类

QTL 类的命名约定与其他 Qt 类一致（比如，count()、isEmpty()）。它们还提供额外的函数来兼容 STL 算法，比如 size()和 empty()。可以像使用 STL 的函数 map 一样来使用它们。

与 STL 相比，QTL 仅仅包含了 STL 容器应用程序接口的最重要的特性，没有平台差异，通常要慢一些并且经常扩展为更少的对象代码。

如果你不想拷贝存储对象，你最好使用 QPtrCollection 及派生类。它们就是被设计用来处理各种类指针的。QObject 没有拷贝 构造函数，因此 QObject 不能作为一个值使用。但可以存储指向 QObject 的指针到 QValueList。当然，直接使用 QPtrList 更好。 QPtrList 像所有其它的基于 QPtrCollection 的容器一样，提供了比速度优化了、基于值的容器更多健全的检查。

如果你有一些使用值的对象，并且在你的目标平台没有可用的 STL，Qt 模板库就可以替代它。使用值的对象至少需要一个拷贝构造函数、一个赋值操作符和一个默认构造函数（如：一个没有任何参数的构造函数）。

注意一个快速的拷贝构造函数对于容器的高性能是关键的，因为许多拷贝操作将会发生。如果你想排序你的数据，你必须在你的数据类中实现 operator<()。

Qt 模板库是为高性能而设计，迭代器是非常快的。为了实现这样的性能，Qt 模板库比基于 QPtrCollection 的集合类做更少的错误 检查。一个 QTL 容器，例如：QTL 容器没有跟踪任何相关的迭代器。这样在诸如删除条目时没有执行有效性检查，但它提供了很好的执行性能。

11.1 迭代器（Iterators）

Qt 模板库打交道的是值对象，而不是指针对象。迭代器是最好的遍历容器方法。遍历一个容器可使用像下面的循环：

```
typedef QValueList<int> List;
List l;
for( List::Iterator it = l.begin(); it != l.end(); ++it )
        printf( "Number is %i\n", *it );
```

begin()返回第一个元素的迭代器，end()返回的是最后一个元素之后的一个迭代器。end()标明的是一个无效的位置，它永远不能被解除引用。它 只是任何一次迭代的终止条件，迭代可以从 begin()或 end()开始。同样的概念也适用于其它容器类，例如，用于 QMap 和 QValueVector 的迭代方法如下：

```
typedef QMap<QString,QString> Map;
Map map;
for( Map::iterator it = map.begin(); it != map.end(); ++it )
       printf( "Key=%s Data=%s\n", it.key().ascii(), it.data().ascii() );

typedef QValueVector<int> Vector;
Vector vec;
for( Vector::iterator it = vec.begin(); it != vec.end(); ++it )
       printf( "Data=%d\n", *it );
```

11.2 算法

Qt 模板库定义了大量操作容器的算法。这些算法用模板库函数实现，还提供了有迭代器的容器的通用代码。例如：qHeapSort()和 qBubbleSort()提供了著名的堆排序和冒泡排序算法。你可以象下面这样使用它们：

```
typedef QValueList<int> List;
List l;
l << 42 << 100 << 1234 << 12 << 8;
qHeapSort( l );

List l2;
l2 << 42 << 100 << 1234 << 12 << 8;
List::Iterator b = l2.find( 100 );
```

```
List::Iterator e = l2.find( 8 );
qHeapSort( b, e );

double arr[] = { 3.2, 5.6, 8.9 };
qHeapSort( arr, arr + 3 );
```

第一个例子对整个列表排序。第二个例子对两个迭代器之间的所有元素排序，即 100、1234 和 12。第三个例子表明迭代器是作为指针使用的。

一些常用的模板函数说明如下：

（1）函数 qSwap()用来交换两个变量的值，例如：

```
QString second( "Einstein" );
        QString name( "Albert" );
qSwap( second, name );
```

（2）函数 qCount()用于统计容器中一个值出现的次数。例如：

```
QValueList<int> l;
        l.push_back( 1 );       //放入 1 到 l 链表中
        l.push_back( 1 );
        l.push_back( 1 );
        l.push_back( 2 );
        int c = 0;
qCount( l.begin(), l.end(), 1, c ); //统计 1 的个数 c， c = 3
```

（3）函数 qFind()用于查找容器中一个值的第一次出现位置。例如：

```
QValueList<int> l;
        l.push_back( 1 );
        l.push_back( 1 );
        l.push_back( 1 );
        l.push_back( 2 );
       //查找 2 所在的位置
        QValueListIterator<int> it = qFind( l.begin(), l.end(), 2 );
```

（4）函数 qFill()用于将一个值拷贝填充到一个范围。例如：

```
QValueVector<int> v(3);
qFill( v.begin(), v.end(), 99 ); //将 99 填充整个 v 数组， v 包含 99, 99, 99
```

（5）函数 qEqual()用来比较两个范围的元素是否相等，两个范围的元素个数不一定相等。只要第一个范围的元素与第二个范围的对应元素都相等时，就认为这两个范围相等。例如：

```
QValueVector<int> v1(3);
    v1[0] = 1;
    v1[2] = 2;
    v1[3] = 3;

    QValueVector<int> v2(5);
    v1[0] = 1;
    v1[2] = 2;
    v1[3] = 3;
    v1[4] = 4;
    v1[5] = 5;

    bool b = qEqual( v1.begin(), v2.end(), v2.begin() );
    // b == TRUE
```

（6）函数 qCopy()用于拷贝一个范围的元素到输出迭代器，例如：

```
QValueList<int> l;
    l.push_back( 100 );
    l.push_back( 200 );
    l.push_back( 300 );
    QTextOStream str( stdout );
   //拷贝 l 中所有元素到输出迭代器 QTextOStreamIterator
    qCopy( l.begin(), l.end(), QTextOStreamIterator(str) );
```

（7）函数 qCopyBackward()用于拷贝一个容器或者它的一部分到一个输出迭代器，拷贝的次序是从后面开始，例如：

```
QValueVector<int> vec(3);
    vec.push_back( 100 );
    vec.push_back( 200 );
    vec.push_back( 300 );
    QValueVector<int> another;
   // “another”包含的是按倒序排列的（300、200、100）。
    qCopyBackward( vec.begin(), vec.end(), another.begin() );
```

如果你写了新的算法，请考虑把它们写成模板函数，这样就可以使它们能够用在尽可能多的容器上了。在上一个例子中，你可以很容易地使用 qCopy()打印出一个标准 C++数组，方法列出如下：

```
int arr[] = { 100, 200, 300 };
    QTextOStream str( stdout );
    qCopy( arr, arr + 3, QTextOStreamIterator( str ) );
```

11.3 数据流串行化

所有提到的容器（如：QValueList 、QStringList、QValueStack 和 QMap 等）都可被相应的流操作符串行化。下面是一个例子。

```
QDataStream str(...);
    QValueList<QRect> l;
    // ……在这里填充这个列表
    str << l;
```

容器还能象下面这样被再一次读入：

```
QValueList<QRect> l;
    str >> l;
```

12　集合类

一个集合类是装有多个条目的容器，每个条目是某种数据结构，集合类能执行对容器中的条目的插入、删除及查找等操作。

Qt 有几个基于值和基于指针的集合类。基于指针的集合类使用指向条目的指针来工作，而基于值的集合类存储着它们条目的拷贝。基于值的集合类类似于 STL 容器类，能和 STL 算法和容器一起使用。

基于值的集合类说明如表 4 所示：

表 4　基于值的集合类表

类名称	说明
QValueList	基于值的链表
QValueVector	基于值的矢量结构
QValueStack	基于值的栈结构
QMap	基于值的字典结构

基于指针的集合类说明如表 5 所示：

表 5　基于指针的集合类表

类名称	说明
QCache 和 QIntCache	LRU (least recently used)缓存结构。
QDict、QIntDict 和 QPtrDict	字典结构。
QPtrList	双向链接的链表结构。
QPtrQueue	FIFO 先进先出(first in,first out)队列结构。
QPtrStack	LIFO 后进先出(last in, first out)栈结构。
QPtrVector	矢量结构。

QMemArray 是一个例外，它既不是基于指针也不是基于值，而是基于内存的结构。用于在有简单数据结构的数组中使用 QMemArray 效率最高，QMemArray 在拷贝和数组元素比较时使用位逻辑运算符操作。

这些类中有一些具有迭代器，迭代器是遍历集合类中的条目的类。在 Qt 模板库里，基于值的集合和算法集成在一起。下面讨论基于指针的容器。

12.1 基于指针的容器的结构

基于指针的容器有 4 个内部基类(QGCache, QGDict, QGList 和 QGVector)操作 void 类型指针。通过增加/删除条目指针，一个由这 4 个类组成的薄模板层实现了实际的集合。

允许 Qt 的模板类的策略使得在空间上很经济（实现这些模板类仅增加了对基类的内联调用），而且还不影响执行效率。

示例：QPtrList 使用

下面的例子说明了如何存储 Employee 条目到一个链表，并将它们以相反的次序打印出来。

```
#include <qptrlist.h>
#include <qstring.h>
#include <stdio.h>

class Employee
{
public:
    Employee( const char *name, int salary ) { n=name; s=salary; }
    const char *name()   const          { return n; }
    int        salary() const           { return s; }
private:
    QString    n;
    int        s;
};

int main()
{
    QPtrList<Employee> list;       // 指向 Employee 的指针链表。
list.setAutoDelete( TRUE );    //当链表条目被移动时，删除条目。

    list.append( new Employee("Bill", 50000) );   //链表追加新的对象。
    list.append( new Employee("Steve",80000) );
    list.append( new Employee("Ron",  60000) );

    QPtrListIterator<Employee> it(list); //遍历 Employee 链表。
    for ( it.toLast(); it.current(); --it) ) {   //从尾向头遍历。
        Employee *emp = it.current();
        printf( "%s earns %d\n", emp->name(), emp->salary() );
    }
```

```
    return 0;
}
```

程序运行结果如下：

```
Ron earns 60000
Steve earns 80000
Bill earns 50000
```

12.2 管理集合条目

所有基于指针的集合继承了 QPtrCollection 基类。这个类仅知道集合中的条目个数和删除策略。

当集合中的条目被移去时，缺省时它们不被删除。QPtrCollection::setAutoDelete()定义了删除策略。在上述 QPtrList 使用示例子，我们激活了自动删除功能来进行链表删除。

当插入一个条目到一个集合时，仅指针被拷贝，而不是拷贝条目本身。这称为浅拷贝。当插入一个条目时，拷贝所有条目的数组到集合中也是可能的，这称为深拷贝。

所有的集合类函数在插入条目时调用虚拟函数 QPtrCollection::newItem()。如果你想进行深拷贝，你需要重载它。

当从一个链表中移去一个条目时，调用虚拟函数 QPtrCollection::deleteItem()。如果自动删除功能被激活，在所有集合类中的缺省实现函数被调用来删除条目。

基于指针的集合类，如：QPtrList<type>，定义了指向对象的指针集合。我们在这里只讨论 QPtrList 类，其它的基于指针的集合类和所有集合类迭代器都有同样的使用方法。

模板实例化方法如下：

```
QPtrList<Employee> list;
```

在这个例子中，条目的类或类型是 Employee，它必须在链表定义之前被定义。例如：

```
class Employee {
    ...
};
QPtrList<Employee> list;
```

12.3 迭代器（Iterators）

QPtrListIterator 能在链表被修改的同时非常安全的遍历链表。在同一个集合上，多个迭代器能独立地工作。

QPtrList 有一个指向所有迭代器的内部链表，这些迭代器当前操作链表。当一个链表条目被移去时，链表更新所有的指向这个条目的迭代器。

QDict 和 QCache 集合没有遍历函数。为了遍历集合，你必须使用 QDictIterator 或 QCacheIterator。

Qt 预定义的集合类有字符串链表：QStrList, QStrIList (在 qstrlist.h 中)和 QStringList (在 qstringlist.h 中)。在绝大多数情况下你将选择 QStringList，它是一个共享的 QString Unicode 字符串的值链表。QPtrStrList 和 QPtrStrIList 仅存储字符指针，而不是字符串本身。

基于指针的集合类和相关的迭代器类说明如表 4。

表 4　基于指针的集合类和相关的迭代器类列表

QAsciiCache	基于 char* keys 的缓存的模板类。
QAsciiCacheIterator	QAsciiCache 集合的迭代器。
QAsciiDict	基于 char* keys 的字典的模板类。
QAsciiDictIterator	QAsciiDict 集合的迭代器。
QBitArray	Bit 位数组。
QBitVal	与 QBitArray 一起用的内部类
QBuffer	在 QByteArray 上操作的 I/O 设备。
QByteArray	字节数组
QCache	基于 QString keys 的模板类。
QCacheIterator	QCache 集合的迭代器。
QCString	C 语言中 0 结尾的字符数组(char *)。
QDict	基于 QString keys 的字典的模板类。
QDictIterator	QDict collections 的迭代器。
QIntCache	基于 long keys 的缓存的模板类。
QIntCacheIterator	QIntCache 集合的迭代器。
QIntDict	基于 long keys 的字典的模板类。
QIntDictIterator	QIntDict 集合的迭代器。
QObjectList	QObjects 的 QPtrList 链表。
QObjectListIt	QObjectLists 的迭代器。

QPtrCollection	大多数基于指针的 Qt 集合的基类。
QPtrDict	基于 void* keys 的字典的模板类。
QPtrDictIterator	QPtrDict 的迭代器。
QPtrList	双向链表的模板类。
QPtrListIterator	QPtrList 集合的迭代器。
QPtrQueue	一个队列的模板类。
QStrIList	对于大小写敏感的 char*双向链接的链表。
QStrList	char* 双向链接的链表。

13 Qt 线程

Qt 对线程提供了支持，它引入了一些基本与平台无关的线程类、线程安全传递事件的方式和全局 Qt 库互斥量允许你从不同的线程调用 Qt 的方法。Qt 中与线程应用相关的类如表 6 所示。

表 6　Qt 中与线程相关的类

QMutex	线程之间的互斥锁
QSemaphore	整数信号量
QThread	与平台无关的线程
QWaitCondition	线程之间允许等待/唤醒的条件

使用线程需要 Qt 提供相应的线程库的支持，因此，在编译安装 Qt 时，需要加上线程支持选项。

当在 Windows 操作系统上编译 Qt 时，线程支持是在一些编译器上的一个选项。在 Windows 操作系统上编译应用程序时，通过在 qconfig.h 文件中增加一个选项来解决来解决这个问题。

在 Mac OS X 和 Unix 上编译 Qt 时，你应在运行 configure 脚本时添加-thread 选项。在 Unix 平台上，多线程程序必须用特殊的线程支持库连接，多线 程程序必须连接线程支持库 libqt-mt，而不是标准的 Qt 库。编译应用程序时，你应该使用宏定义 QT_THREAD_SUPPORT 来编译（如：编译 时使用-DQT_THREAD_SUPPORT）。

13.1 线程类 QThread

在 Qt 中提供了 QThread 线程类，它提供了创建一个新线程的方法。线程通过重载 QThread::run()函数开始执行的，这一点与 Java 中的线程类相似。

示例 1：一个简单的线程

下面的例子实现了一个简单的继承自 QThread 的用户线程类，并运行 2 个线程，线程 b 在线程 a 运行完后运行。代码列出如下：

```
class MyThread : public QThread {
public:
	virtual void run();
};

void MyThread::run()		//运行线程
{
	for( int count = 0; count < 20; count++ ) {
		sleep( 1 );
		qDebug( "Ping!" );
	}
}

int main()
{
	MyThread a;
	MyThread b;
	a.start();	//通过调用 run()函数来执行
	b.start();
	a.wait();
	b.wait();
}
```

只有一个线程类是不够的，对于支持多线程的程序来说，还需要保护两个不同的线程对数据的同时访问，因此 Qt 提供了 QMutex 类，一个线程可以锁住互斥量，当互斥量被锁住时，将阻塞其它线程访问临界数据，直到这个线程释放互斥量。这样，可以保护临界数据一次只能被一个线程访问。

Qt 库互斥量（qApp->lock()和 qApp->unlock()）是在访问 Qt 的 GUI 界面资源时用到的互斥量。在 Qt 中没有使用互斥量而调用一个函数通常会导致不可预知的行为。从另外一个线程中调用 Qt 的一个 GUI 相关函数需要使用 Qt 库互斥量。在这种情况下，所有 访问图形或窗口系统资源的函数都与 GUI 相关。如果对象仅被一个线程访问，使用容器类，字符串或者输入/输出类不需要任何互斥量。

13.2 线程安全的事件传递

在 Qt 中，一个线程总是一个事件线程，线程从窗口系统中拉出事件并且把它们分发给窗口部件。静态方法 QThread::postEvent 从线程中 邮递事件，而不是从事件线程。事件线程被唤醒并且事件象一个正常窗口系统的事件一样在事件线程中被分发。例如，你可以从不同的线程强制一个窗口部件进行重 绘，方法如下：

```
QWidget *mywidget;
QThread::postEvent( mywidget, new QPaintEvent( QRect(0, 0, 100, 100) ) );
```

上述代码将异步地使 mywidget 在它区域中重绘一块 100*100 的正方形区域。

另外，还需要一些机制使得处于等待状态的线程在给定条件下被唤醒。QWaitCondition 类就提供了这种功能。线程等待的条件 QWaitCondition 满足，QWaitCondition 表明发生了什么事情，它阻塞直到这件事情发生。当发 生给定的事情时，QWaitCondition 将唤醒等待该事情的所有线程或者唤醒任意一个被选中的线程。（这和 POSIX 线程条件变量具有相同功能，是 Unix 上的一种实现。）

示例 2：QWaitCondition 类应用

下面这个例子的功能是：当你按下按钮，这个程序就会唤醒 worker 线程，这个线程在按钮上显示工作状态：等待（Waiting）还是正在 工作（Working）。当按钮被按下时，worker 线程正在工作，那么对线程不产生影响。当 run 函数再次循环到 mycond.wait()时，线程 阻塞并等待。当按钮再被按下时，触发 slotClicked()函数运行，唤醒等待的线程。

```
#include <qapplication.h>
#include <qpushbutton.h>

// 全局条件变量
QWaitCondition mycond;

// Worker 类实现
class Worker : public QPushButton, public QThread
{
    Q_OBJECT

public:
    Worker(QWidget *parent = 0, const char *name = 0)
            : QPushButton(parent, name)
    {
        setText("Start Working");

        // 将 QPushButton 继承来的信号与槽 slotClicked()连接起来。
        connect(this, SIGNAL(clicked()), SLOT(slotClicked()));
```

```
        // 调用从 QThread 继承来的 start()方法开始线程的执行
        QThread::start();
    }

public slots:
    void slotClicked()
    {
        // 唤醒等待这个条件变量的一个线程
        mycond.wakeOne();
    }

protected:
    void run()   //重载 run 函数
    {
            while ( TRUE ) {
                // 锁定应用程序互斥锁，并且设置窗口标题来表明我们正在等待开始工作
                qApp->lock();
                setCaption( "Waiting" );
                qApp->unlock();

                // 等待直到我们被告知可以继续
                mycond.wait();

                // 如果到了这里，表示我们已经被另一个线程唤醒
                qApp->lock();
                setCaption( "Working!" );// 设置标题，表示正在工作
                qApp->unlock();
            }
    }
};

int main( int argc, char **argv )
{
    QApplication app( argc, argv );

    // 创建一个 worker
```

```
    Worker firstworker( 0, "worker" );

    app.setMainWidget( &worker ); //将 worker 设置为应用程序的主窗口。
    worker.show();

    return app.exec();
}
```

当进行线程编程时，需要注意的一些事项：

（1）在持有 Qt 库互斥量时不要做任何阻塞操作。这将冻结事件循环。

（2）确认你锁定一个递归 QMutex 的次数等于解锁的次数，不能多也不能少。

（3）在调用除了 Qt 容器和工具类外的任何东西之前锁定 Qt 应用程序互斥量。

（4）谨防隐含的共享类，如果你需要在线程之间指定它们，你应该用 detach()分离它们。

（5）小心没有被设计成线程安全的 Qt 类，例如，QPtrList 的 API 接口不是线程安全的，并且如果不同的线程需要遍历一个 QPtrList，它们应该在调用 QPtrList::first()之前锁住，在到达终点后解锁。

（6）确信仅在 GUI 线程中创建继承自 QWidget、QTimer 和 QSocketNotifier 的对象。在一些平台上，创建在线程中而不是 GUI 线程的对象永远不会接收到底层窗口系统的事件。

（7）和上面很相似，只在 GUI 线程中使用 QNetwork 类。因为所有的 QNetwork 类都是异步的，没必要把 QSocket 用在多线程中。

（8）永远不要尝试在不是 GUI 线程的线程中调用 processEvents()函数。这也包括 QDialog::exec()、QPopupMenu::exec()、QApplication::processEvents()和其它一些函数。

（9）在你的应用程序中，不要把普通的 Qt 库和支持线程的 Qt 库混合使用。这意味着如果你的程序使用了支持线程的 Qt 库，你就不能连接普通 的 Qt 库、动态的载入普通 Qt 库或者动态地连接其它依赖普通 Qt 库的库或者插件。在一些系统上，这样做会导致 Qt 库中使用的静态数据崩溃。

14　鼠标拖放

拖放提供了一种用户在应用程序之间或之内传递信息的一种简单可视机制。在术语中，这被称为"直接操作模型"。拖放在功能上类似剪贴板的剪切和粘贴机 制。拖放机制包括拖动、放下、剪贴板、拖放操作、添加新的拖放类型、高级拖放以及和其它应用程序之间的操作几个方面。下面从这几个方面分别进行说明：

（1）拖动

开始一个拖动，比如是在鼠标移动事件，创建一个适合你的媒体的 QDragObject 的子类的对象，例如：对于文本使用 QTextDrag，对于图片使用 QImageDrag。然后调用 drag()方法。例如，从一个窗口部件中开始拖动一些文本：

```
void MyWidget::startDrag()
    {
        QDragObject *d = new QTextDrag( myHighlightedText(), this );
        d->dragCopy();    //拷贝选中文本
        // 不要删除 d。
    }
```

注意在拖动之后，QDragObject 没有被删除。在拖放明显完成后，这个 QDragObject 需要被保存。因为它还可能需要与其它进程通信。最后 Qt 会删除这个对象。如果拥有拖动对象的窗口部件在删除拖动对象之前被删除，那么任何没有完成的放下操作将会被取消，并且拖动对象会被删除。因为这个原 因，你应该小心对待对象引用。

（2）放下

为了能在一个窗口部件中接收被放下的媒体，这个窗口部件调用 setAcceptDrops(TRUE)（如：在它的构造函数中），并且重载 事件处理方法 dragEnterEvent()和 dropEvent()。对于更复杂的应用程序，重载 dragMoveEvent()和 dragLeaveEvent()也是必需的。

例如，当拖动后放下文本或图片时，窗口部件接受并处理放下操作的代码如下：

```
MyWidget::MyWidget(...) :
        QWidget(...)
    {
            ...
            setAcceptDrops(TRUE); //接收被放下的媒体。
    }
//当一个拖动正在进行并且鼠标进入这个窗口部件，这个事件处理函数被调用。
    void MyWidget::dragEnterEvent(QDragEnterEvent* event)
    {
          event->accept( QTextDrag::canDecode(event) ||
                QImageDrag::canDecode(event)  );
    }
//当拖动在这个窗口部件上被放下，这个事件处理器被调用。
    void MyWidget::dropEvent(QDropEvent* event)
```

```
{
    QImage image;
    QString text;

    if ( QImageDrag::decode(event, image) ) {//解码图片
        insertImageAt(image, event->pos());   //在窗口部件中插入图片
    } else if ( QTextDrag::decode(event, text) ) {
        insertTextAt(text, event->pos());
    }
}
```

（3）剪贴板

QDragObject、QDragEnterEvent、QDragMoveEvent 和 QDropEvent 类都是 QMimeSource（提供类型信息的类）的子类。如果你在 QDragObject 中基于你的数据进行传递，你不仅可使用拖放，而且还可以使用传统的剪 切和粘贴。QClipboard 有两个函数：

```
setData(QMimeSource*)
    QMimeSource* data()const
```

使用这些函数，你可以把你的拖放初始信息放到剪贴板中：

```
void MyWidget::copy()
{
    QApplication::clipboard()->setData( new QTextDrag(myHighlightedText()) );
}

void MyWidget::paste()
{
    QString text;
    if ( QTextDrag::decode(QApplication::clipboard()->data(), text) )
        insertText( text );
}
```

你甚至能使用 QDragObject 的子类作为文件 I/O 部分。例如，如果你的程序有一个 QDragObject 的子类把 CAD 设计编码成 DXF 格式，你可以象下面这样存储和装载这个格式的文件：

```
void MyWidget::save()
{
    QFile out(current_file_name);
    out.open(IO_WriteOnly);
    MyCadDrag tmp(current_design); // MyCadDrag 是 QDragObject 的子类
    out.writeBlock( tmp->encodedData( "image/x-dxf" ) );
}

void MyWidget::load()
{
    QFile in(current_file_name);
    in.open(IO_ReadOnly);
    if ( !MyCadDrag::decode(in.readAll(), current_design) ) {
        QMessageBox::warning( this, "Format error",
            tr("The file \"%1\" is not in any supported format")
              .arg(current_file_name)
        );
    }
}
```

（4）拖放操作

在一些简单的情况下，拖放的目标接收一个被拖动的数据的拷贝，并且由源来决定是否删除初始的拖动对象。这是 QDropEvent 中 的"Copy"操作。目标也可以选择理解其它操作，特别是"Move"和"Link"操作。如果目标理解了"Move"操作，目标负责拷贝和删除操作，源 不会尝试删除数据。如果目标理解为"Link"操作，它存储它自己的引用到初始信息中，并且源不会删除初始信息。最通用的拖放操作是在同一个窗口部件中执 行一个"Move"操作。

拖动操作的另一个主要用途是当使用一个引用类型，比如 text/uri-list，实际上被拖动的数据是文件或对象的引用。

（5）添加新的拖放类型

拖放不仅仅局限于文本和图片，任何信息都可以被拖放。为了在应用程序之间拖放信息，两个应用程序必须指明彼此都能接受和产生的数据格式。这 个可以通过使用 MIME 类型来获得。拖动的源提供一个它能产生的 MIME 类型列表（按从最合适的到最少合适的顺序排列），并且放下的目标选择一种它能接受 的类型。例如，QTextDrag 提供了"text/plain"MIME 类型（普通的没有格式的文本），还有"text/utf16"和"text /utf8"的 Unicode 格式的类型。QImageDrag 提供了"image/*"类型，*是 QImageIO 支持的任何一种图片格式，并且 QUriDrag 子类提供了"text/uri-list"的支持，它是传输一个文件名列表（或 URL）的标准格式。

为了实现一些还没有可用 QDragObject 子类的信息类型的拖放，首先和最重要的步骤是查找合适的存在格式：IANA（Internet Assigned Numbers Authority）在 ISI（Information Sciences Institute）提供了一个 MIME 媒体类型的分级列表。使用标准的 MIME 类型将会使你的应用程序现在及未来能更好地与其它软件互相操作。

为了支持另外的媒体类型，从 QDragObject 或 QStoredDrag 派生类。当你需要提供多种媒体类型的支持时，从 QDragObject 派生类。当一个类型足够时，就从更简单的 QStoredDrag 派生类。

QDragObject 的子类将会重载 const char* format(int i) const 和 QByteArray encodedData(const char* mimetype) const 成员，并且提供一套方法编码媒体数据，提供静态成员 canDecode()和 decode()解码输入的数据，QImageDrag 的成员函数 bool canDecode(QMimeSource*) const 和 QByteArray decode(QMimeSource*) const 在子类中需要类似的重载。

QStoredDrag 的子类提供了提供一套方法编码媒体数据，静态成员 canDecode()和 decode()对进入的数据进行解码。

（6）高级拖放

在剪贴板模式中，用户可以剪切或复制资源信息，然后粘贴它。相似地，在拖放模式中，用户可以拖动信息的拷贝或者拖动信息本身到一个新的位置（移动它）。拖放模式对于程序员来说都是更多的复杂性：程序直到放下（粘贴）完成才会知道用户是想剪切还是复制。在应用程序之间拖动，这个没有什么区别， 但是在一个应用程序之内进行拖动，应用程序必须小心不要将拷贝粘贴到同一个地方。例如，在同上窗口部件中拖动文本，拖动的开始点和放下事件处理函数应象下 面这样重载：

```
void MyEditor::startDrag()
{
    QDragObject *d = new QTextDrag(myHighlightedText(), this);
    if ( d->drag() && d->target() != this )
        cutMyHighlightedText();   //剪切选中的文本
}

void MyEditor::dropEvent(QDropEvent* event)
{
```

```
    QString text;

    if ( QTextDrag::decode(event, text) ) {
        if ( event->source() == this && event->action() == QDropEvent::Move ) {
            // 在同一个窗口部件时，不能使用粘贴拷贝，而应是移到到这个位置。
            event->acceptAction();
            moveMyHighlightedTextTo(event->pos());
        } else {
            pasteTextAt(text, event->pos());    //粘贴拷贝
        }
    }
}
```

一些窗口部件在数据被拖动到它们上面时需要指定"是"或"否"接收。例如，一个 CAD 程序也许只接收在视图中的文本对象上放下的文本。在这种情况下，dragMoveEvent()被使用并且给定接受或者忽略拖动的区域。代码列出如下：

```
void MyWidget::dragMoveEvent(QDragMoveEvent* event)
{
    if ( QTextDrag::canDecode(event) ) {
        MyCadItem* item = findMyItemAt(event->pos());
        if ( item )
            event->accept();
    }
}
```

（7）和其它应用程序之间的操作

在 X11 上，拖动使用公有的 XDND 协议，而 Qt 在 Windows 上使用 OLE 标准，Qt 在 Mac 上使用 Carbon 拖动管理器。在 X11 上，XDND 使用 MIME，所以不需要转换。Qt 的应用编程接口与平台无关。在 Windows 上，识别 MIME 的应用程序可以通过使用 MIME 类型的剪贴 板格式名字进行通信。一些 Windows 应用程序已经对它们的剪贴板格式使用 MIME 命名规范了。在内部，Qt 有能力在专有的剪贴板格式和 MIME 类型之间转换。在 X11 上，Qt 也支持使用 Motif 拖放协议的拖动。

15 键盘焦点

Qt 的窗口部件在图形用户界面中按用户的习惯的方式来处理键盘焦点。基本出发点是用户的击键能定向到屏幕上窗口中的任何一个，和在窗口中任何一个部 件中。当用户按下一个键，他们期望键盘焦点能够到达正确的位置，并且软件必须尽量满足这种希望。系统必须确定击键定位在哪一个应用程序、应用程序中的哪一 个窗口和窗口中的哪一个窗口部件。

15.1 焦点移动的方式

把焦点定位特殊的窗口部件的习惯方式有：

（1）用户按下 Tab 键（或者 Shift 键+Tab 键）（或者有时是 Enter 键）。

（2）用户点击一个窗口部件。

（3）用户按下一个键盘快捷键。

（4）用户使用鼠标滚轮。

（5）用户移动焦点到一个窗口，并且应用程序必须决定窗口中的哪个窗口部件应该得到焦点。

这些移动机制的每个都是不同的，并且不同类型的窗口部件只能接收它们中的一些方式的焦点。下面我们将按次序介绍它们。

1. Tab 或者 Shift+Tab.

按 Tab 键是到目前为止用键盘移动焦点的最通用的方法。有时在输入数据的应用程序中 Enter 键和 Tab 键的作用是一样的。我们暂时忽略这一点。

所有窗口系统中的有关焦点的最通用使用方法是：按 Tab 键移动键盘焦点到每个窗口的窗口部件循环列表中的下一个窗口部件。Tab 键按照循环 列表的一个方向移动焦点，Shift 键+Tab 键按另一个方向移动焦点。按 Tab 键从一个窗口部件到下一个窗口部件移动焦点的次序叫做 Tab 键次序。

在 Qt 中，窗口部件循环列表存放在 QFocusData 类中。每个窗口有一个 QFocusData 对象，并且当选择合适的 QWidget::FocusPolicy 焦点策略的 QWidget::setFocusPolicy()被调用的时候，窗口部件自动把它们自己追加到列 表的末尾。你可以使用 QWidget::setTabOrder()来自定义 Tab 键控制次序。如果你没有定义这个次序，那么 Tab 键会按照窗口部件构造 的顺序移动焦点。Qt designer 工具提供了一个可视化的改变 Tab 键控制次序的方法。

因为按 Tab 键是如此的常用，大多数含有焦点的窗口部件应该支持 Tab 焦点。主要例外情况是几乎没用到的窗口部件，并且在窗口部件上有一些移动焦点的键盘快捷键或者错误处理。

2. 用户点击一个窗口部件。

在使用鼠标或者其它指针设备的计算机中，用鼠标点击一个窗口部件是一种比按 Tab 键更常用的方法。

当鼠标点击把焦点移到一个窗口部件时，对于编辑器窗口部件，它也会移动文本光标（窗口部件的内部焦点）到鼠标被点击的地点。

鼠标点击移动焦点是大多数窗口部件必须支持的，有时窗口部件需要避免鼠标点击移动焦点。例如：在一个字处理程序中，当用户点击"B"（粗体）工具按钮，键盘焦点应该保留在原来的位置。在 Qt 中，只有 QWidget::setFocusPolicy()函数影响点击焦点。

3. 用户按下一个键盘快捷键。

使用键盘快捷键来移动焦点不是很常用。这种情况可能会隐含地发生在打开的模式对话框中，但是也会显式地发生在使用焦点加速器中，例如在 QLabel::setBuddy()、QGroupBox 和 QTabBar 提供的加速器中。

用户想让焦点跳到的窗口部件都应支持快捷键焦点。例如：一个 Tab 对话框为它的每一个页提供键盘快捷键，所以用户可以按下比如 Alt+P 来 跳到打印页面。但只能有少量的快捷键，并且为命令提供键盘快捷键也很重要，如：在标准快捷键列表中，Alt+P 也可以用来粘贴、播放或打印。

4. 用户使用鼠标滚轮。

在 Microsoft Windows 上，鼠标滚轮的用法是一直由有键盘焦点的窗口部件处理。在 Mac OS X 和 X11 上，它由获得其它鼠标事件的窗口部件处理。

Qt 处理这种平台差异的方法是当滚轮被使用时，让窗口部件移动键盘焦点。每个窗口部件上有合适的焦点策略，应用程序可以在 Windows、Mac OS X 和 X11 上按照习惯正确地处理焦点。

5. 用户移动焦点到这个窗口。

在这种情况下，应用程序必须决定窗口中的哪一个窗口部件接收焦点。Qt 自动实现这样的做法：如果焦点以前在这个窗口中，那么窗口中有焦点的 最后一个窗口部件应该重新获得焦点。如果以前焦点就从来没有来到过这个窗口，并且你知道焦点应该从哪里开始，就在你调用 QWidget::show()显 示它之前，在应该接收焦点的窗口部件上调用 QWidget::setFocus()。如果你不知道，Qt 会选择一个合适的窗口部件。

15.2 焦点策略及操作函数
键盘焦点的策略及操作函数说明如下：

（1） 焦点策略属性变量

focusPolicy 焦点策略属性变量保存的是窗口部件接收键盘焦点的策略。如果窗口部件通过 tab 来接收键盘焦点，这个策略就是 QWidget::TabFocus；如果窗 口部件通过点击来接收键盘焦点，这个策略就是 QWidget::ClickFocus；如果窗口部件上述两种方式都使用，是 QWidget::StrongFocus；并且如果它不接收焦点（QWidget 的默认值），是 QWidget::NoFocus。

如果一个窗口部件处理键盘事件，你必须使键盘焦点生效。这通常在窗口部件的构造函数中完成。例如，QLineEdit 的构造函数调用 setFocusPolicy(QWidget::StrongFocus)。

（2）void QWidget::setFocus() [虚槽]

函数 setFocus()把键盘输入焦点赋给这个窗口部件（或者它的焦点代理）。

首先，一个焦点移出事件会被发送给焦点窗口部件（如果有的话）告诉它关于失去焦点的事情。然后一个焦点进入事件被发送给这个窗口部件告诉它刚刚接收到焦点。（如果焦点移出和进入的窗口部件是同一个的话，就什么都没有发生。）

函数 setFocus()会把焦点给一个窗口部件，而不管它的焦点策略，但是不会清空任何键盘捕获（grabKeyboard()）。请注意如果窗口部件是被隐藏的，它将不接收焦点。

（3）void QWidget::setFocusProxy(QWidget * w) [虚]

函数 setFocusProxy 设置这个窗口部件的焦点代理为窗口部件 w。如果 w 为 0，这个函数重置这个窗口部件没有焦点代理。

一些窗口部件，比如 QComboBox，能够"拥有焦点"，但创建一个子窗口部件来实际处理这个焦点。例如，QComboBox 创建了一个 QLineEdit 来处理焦点。

当"这个窗口部件"获得焦点时，setFocusProxy()设置的这个窗口部件实际获得焦点。如果有了一个焦点代理，focusPolicy()、setFocusPolicy()、setFocus()和 hasFocus()都在这个焦点代理上操作。

16 会话管理

一个会话是一组正在运行的应用程序，它们每个都一个特殊的状态。会话被一个称为会话管理器（Session manager）的服务程序来控制。在会话里每个参与的应用程序被称为会话客户。

会话管理器为用户发出命令给它的客户。这些命令会使客户提交没有保存的变化（如：保存打开的文件），使客户为将来的会话保存状态或关机。这样的一些操作被称为会话管理。

通常情况下，一个会话由用户在其桌面上同时运行的所有应用程序组成。在 Unix/X11 下，一个会话可能包括运行在不同计算机和多个显示器上的应用程序。

16.1 会话管理

（1）关闭一个会话

一个会话可以被会话管理器关闭，通常在用户 logout 时为用户关闭的。一个系统可以在紧急情况下执行自动关闭，例如：在掉电时。正常关机 和掉电关机有很大的不同，在正常关机时，用户可能想与应用程序交互，并确定哪些文件应该保存，哪些应该删除。在掉电关机时，没有时间进行交互，甚至于用户 不在现场。

（2）不同平台上的协议和支持

Mac OS X 和 MS-Windows 对应用程序还没有完全的会话管理，如：没法恢复以前的会话。它们支持正常的 logout，在得到用户确认后，应用程序有机会取消 进程。这是与 QApplication::comm.itData()方法相对应的功能。X11 自从 X11R6 后，支持完整的会话管理。

（3）让会话管理与 Qt 一起工作

通过重载 QApplication::comm.itData()来使用你的应用程序参加正常的 logout 处理。如果你仅应用在 MS-Windows 平台上，就只能提供这个重载了。你的应用程序最好提供一个如图 5 的程序关闭对话框。

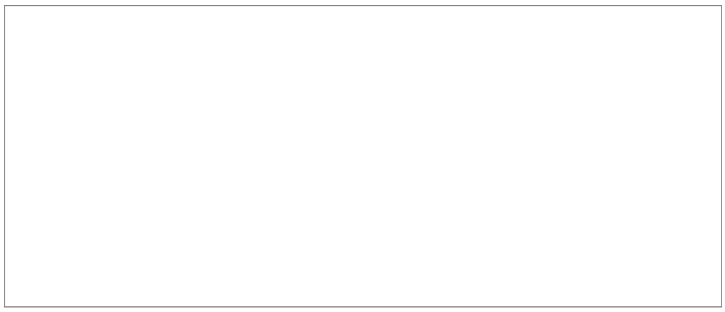

图 5　程序关闭对话框

对于完全的会话管理（目前仅 X11R6），你还应该关心应用程序的状态存储和下一个会话生命周期恢复状态。

16.2 测试和调试会话管理

在 Mac OS X 和 Windows 上的会话管理支持由于这些操作系统本身的这种功能的缺乏而受到限制。为了简单地关闭会话并验证你的应用程序是否如期望的那样执行，你最好启动一个其它的应用程序。这个应用程序将随后得到关闭消息，这样允许你取消关闭。

在 Unix 上你能使用一个支持标准的 X11R6 会话管理，或使用 X 联盟提供的会话管理器 xsm。xsm 是标准 X11R6 安装的一部分。它是一个具有图形界面的会话管理器，你可能用它来管理会话。

下面是使用 xsm 的一个简单的方法：

（1）运行 X11R6.

（2）在你的 home 目录下将创建仅包含下面一行的.xsmstartup 文件:

xterm

这将告诉 xsm 的 default/failsafe(缺省/失败安全)会话仅有 xterm。否则，xsm 将尝试触发包括窗口管理器 twm 在内多个客户，这没有什么用途。

（3）现在从另一个终端窗口启动 xsm。一个会话管理器和 xterm 将出现。xterm 有一个其它的 shell 没有的属性：在 xterm 的 shell 里，SESSION_MANAGER 环境变量指向了你刚启动的会话管理器。

（4）在新的 xterm 窗口中启动你的应用程序，应用程序将自动连接它到会话管理器。你能使用 ClientList 按钮检查连接是否成功。

注意：当你启动或关闭会话管理的客户时，不要保护 ClientList 为打开状态。否则，xsm 可能崩溃。

（5）使用会话管理器的 Checkpoint 和 Shutdown 按钮的不同设置检查你的应用程序执行行为。本地保存类型表示客户端应该保存 它们的状态。它对应着 QApplication::saveState()函数。全局保存类型请求应用程序保存它们没存储的变化到永久的全局可访问的存储 中。它触发 QApplication::commitData()。

（6）在用户桌面上 xsm 是一个有用的会话管理器。作为一个测试环境它是稳定的而有用的。

17　调试技术

Qt 应用程序的调试可以通过 DDD 进行跟踪调试和打印各种调试或警告信息。DDD（Data Display Debugger）是使用 gdb 调试工具的图形工具，它安装在 Linux 操作系统中，使用方法可参考 DDD 的帮助文档。下面说明如何打印各种调试或警告信息

17.1 命令行参数

当你运行 Q 应用程序时，你可以指定几个命令行参数来帮助你调试。这几个命令行参数说明如下：

-nograb 应用程序不再捕获鼠标或者键盘。当程序在 Linux 下运行在 gdb 调试器中时这个选项是默认的。

-dograb 忽略任何隐含的或明显得-nograb。即使-nograb 出现在命令行的最后，-dograb 也会超过-nograb 生效的。

-sync 在 X 同步模式下运行应用程序。同步模式强迫 X 服务器立即执行每一个 X 客户端的请求，而不使用缓存优化。它使得程序更加容易测试并且通常会更慢。-sync 模式只对 X11 版本的 Qt 有效。

17.2 打印警告和调试消息

Qt 使用三个全局函数 qDebug、qWarning 和 qFatal 来打印警告和调试信息到标准错误输出 stderr（它在缺省情况下为显示屏，也可指定为文件）。 这三个函数说明如下：

qDebug()用来打印调试信息，在调试版本中输出信息，在发布版本中，函数将不起作用。

qWarning()用来在程序发生错误时打印警告信息。

qFatal()用来打印致命错误消息并且退出。

这些函数的 Qt 实现在 Unix/X11 下把文本打印到标准错误输出（stderr），在 Windows 下会打印到调试器。你可以通过安装一个消息处理器，qInstallMsgHandler()来接收这些函数。

因为这 3 个函数的实现类似，这里只分析函数 qDebug，qDebug 函数的参数格式与函数 printf 类似，打印格式化字符串。qDebug 函数列出如下（在 src/tools/qglobal.cpp 中）：

```
static QtMsgHandler handler = 0;  //指向用户定义的打印输出函数的句柄。
static const int QT_BUFFER_LENGTH = 8196;    //内部 buffer 长度。
```

```
void qDebug( const char *msg, ... )   //msg 格式化的需要打印的字符串。
{
        char buf[QT_BUFFER_LENGTH];
        va_list ap;
        va_start( ap, msg );   //使用可变的参数链表。
#if defined(QT_VSNPRINTF)
        QT_VSNPRINTF( buf, QT_BUFFER_LENGTH, msg, ap );
#else
        vsprintf( buf, msg, ap );    //将需要打印的信息放入到 buf 中。
#endif
        va_end( ap );
        if ( handler ) {//如果用户指定的输出函数存在，使用它来输出信息。
  (*handler)( QtDebugMsg, buf );
        } else {
#if defined(Q_CC_MWERKS)
                mac_default_handler(buf); //mac 系统下的缺省输出函数。
#elif defined(Q_OS_TEMP)
  QString fstr( buf );
  OutputDebugString( (fstr + "\n").ucs2() );
#else
  fprintf( stderr, "%s\n", buf );  // 输出到 stderr
#endif
        }
}
```

在 src/tools/qglobal.h 中定义了 QtMsgHandler 的函数类型，并将函数 qInstallMsgHandler 定义为从动态库中输出函数名。这两个定义列出如下：

```
typedef void (*QtMsgHandler)(QtMsgType, const char *);
// Q_EXPORT 表示动态库中输出这个函数名
Q_EXPORT QtMsgHandler qInstallMsgHandler( QtMsgHandler );
```

函数 qInstallMsgHandler 被用户用来定义一个安装处理函数，并返回以前定义的消息处理函数的指针。在一

个应用程序中只能定义一个消息处理 函数。恢复以前的消息处理函数时，调用 qInstallMsgHandler(0)。函数列出如下（在 src/tools/qglobal.cpp 中）：

```
QtMsgHandler qInstallMsgHandler( QtMsgHandler h )
{
    QtMsgHandler old = handler;
    handler = h;
    return old;
}
```

示例：应用 qInstallMsgHandler

下面的例子说明如果在一个应用程序中安装自己的程序运行信息输出函数。这个例子先定义了信息输出函数 myMessageOutput，然 后，在程序的 main 函数中安装了信息输出函数。当这个应用函数运行时，就会使用函数 myMessageOutput 输出运行信息。代码如下：

```
#include <qapplication.h>
#include <stdio.h>
#include <stdlib.h>

void myMessageOutput( QtMsgType type, const char *msg )//定义信息输出函数
{
    switch ( type ) {
  case QtDebugMsg: //输出调试信息
        fprintf( stderr, "Debug: %s\n", msg );
        break;
  case QtWarningMsg:   //输出警告信息
        fprintf( stderr, "Warning: %s\n", msg );
        break;
  case QtFatalMsg:   //输出致命信息
        fprintf( stderr, "Fatal: %s\n", msg );
        abort();   //中断运行，退出程序
    }
}

int main( int argc, char **argv )
{
```

```
    qInstallMsgHandler( myMessageOutput );   //安装信息输出函数
    QApplication a( argc, argv );
    ...
    return a.exec();
}
```

还有另外两个打印对象信息的调试函数 QObject::dumpObjectTree()和 QObject::dumpObjectInfo()。它们只在程序调试版本下，输出信息，在发布版本中，这两个函数不起作用。函数 QObject::dumpObjectInfo()打印一个对象信号连接等方面的 信息。函数 QObject::dumpObjectTree()打印出子对象树。

17.3 调试宏

在程序运行中还常使用宏 Q_ASSERT 和 Q_CHECK_PTR 来输出信息，这两个宏说明如下：

（1）Q_ASSERT(b)中的 b 是一个布尔表达式，当 b 是 FALSE 的时候，打印出类似的警告信息："ASSERT:‘b’ in file file.cpp (234)"。

（2）Q_CHECK_PTR(p)中的 p 是一个指针。如果 p 是空的话，打印出类似的警告信息："In file file.cpp, line 234: Out of memory"。

宏 Q_ASSERT 实质上是调用函数 qFatal 或 qWarning 输出信息，列出如下（在 src/tools/qglobal.h 中）：

```
#if !defined(Q_ASSERT)
#  if defined(QT_CHECK_STATE)
#    if defined(QT_FATAL_ASSERT)
#      define Q_ASSERT(x)   //打印 x，文件名，在程序源代码中的行号
#    else
#      define Q_ASSERT(x)
#    endif
#  else
#    define Q_ASSERT(x)
#  endif
#endif
```

宏 Q_CHECK_PTR 实质上调用函数 qWarning 输出信息，宏定义 Q_CHECK_PTR 列出如下（在 src/tools/qglobal.h 中）：

```
#if defined(QT_CHECK_NULL)
#  define Q_CHECK_PTR(p) (qt_check_pointer
```

```
#else
#  define Q_CHECK_PTR(p)
#endif

Q_EXPORT bool qt_check_pointer( bool c, const char *, int );
```

函数 qt_check_pointer 实现信息输出操作，函数列出如下（在 src/tools/qglobal.cpp 中）：

```
bool qt_check_pointer( bool c, const char *n, int l )
{
        if ( c )
  qWarning( "In file %s, line %d: Out of memory", n, l );
        return TRUE;
}
```

示例 2：运行宏 Q_ASSERT 和 Q_ASSERT

宏 Q_ASSERT 和 Q_ASSERT 常用来检测程序错误，下面例子使用了这两个宏：

```
char *alloc( int size )
{
        Q_ASSERT( size > 0 ); //如果 size > 0 表达式不成立，打印警告信息。
        char *p = new char[size];
        Q_CHECK_PTR( p );   //如果指针 p 为空，打印警告信息。
        return p;
}
```

Qt 基于不同的调试标记打印不同类型的警告信息。Qt 使用了下面的宏定义说明了不同的调试标记（在 src/tools/qglobal.h 中）：

QT_CHECK_STATE：检测一致的/期望的对象状态

QT_CHECK_RANGE：检测变量范围错误

QT_CHECK_NULL：检测危险的空指针

QT_CHECK_MATH：检测危险的数学，比如被 0 除

QT_NO_CHECK：关闭所有的 QT_CHECK_...标记

QT_DEBUG：使调试代码生效

QT_NO_DEBUG：关闭 QT_DEBUG 标记

默认情况下，QT_DEBUG 和所有的 QT_CHECK 标记都是打开的。如果要关闭 QT_DEBUG，请定义 QT_NO_DEBUG。如果要关闭 QT_CHECK 标记，请定义 QT_NO_CHECK。

示例 3： 打印不同类型的警告信息

下面的例子根据不同的宏定义打印不同类型的警告信息。代码如下：

```
void f( char *p, int i )
{
#if defined(QT_CHECK_NULL) //检测危险的空指针
        if ( p == 0 )
            qWarning( "f: Null pointer not allowed" );
#endif

#if defined(QT_CHECK_RANGE) //检测变量范围错误
        if ( i < 0 )
            qWarning( "f: The index cannot be negative" );
#endif
}
```