公告

昵称：夜&枫
园龄：6年1个月
粉丝：112
关注：49
+加关注

Qt经典—线程、事件与Qobject

介绍

> You’re doing it wrong. — Bradley T. Hughes

30 31 1 2 3 4 5

搜索

找找看

谷歌搜索

最新随笔

1. 2016年的互联网创业
2. 《失控》对当下的互联网投资有哪些借鉴意义
3. 值得推荐的C/C++框架和库
4. 学会自我管理
5. 互联网的寒冬来了，BAT都不社招了
6. 十种数据采集滤波的方法和编程实例
7. 好老板VS坏老板
8. 不耐烦，不淡定，不会好好说话，为何如今遍地戾气？
9. [纯干货] MySQL索引背后的数据结构及算法原理

线程是qt channel里最流行的讨论话题之一。许多人加入了讨论并询问如何解决他们在运行跨线程编程时所遇到的问题。

快速检阅一下他们的代码，在发现的问题当中，十之八九遇到得最大问题是他们在某个地方使用了线程，而随后又坠入了并行编程的陷阱。Qt中创建、运行线程的“易用”性、缺乏相关编程尤其是异步网络编程知识或是养成的使用其它工具集的习惯、这些因素和Qt的信号槽架构混合在一起，便经常使得人们自己把自己射倒在了脚下。此外，Qt对线程的支持是把双刃剑：它即使得你在进行Qt多线程编程时感觉十分简单，但同时你又必须对Qt所新添加许多的特性尤为小心，特别是与QObject的交互。

本文的目的不是教你如何使用线程、如何适当地加锁，也不是教你如何进行并行开发或是如何写可扩展的程序；关于这些话题，有很多好书，比如这个链接给的推荐读物清单. 这篇文章主要是为了向读者介绍Qt 4的事件循环以及线程使用，其目的在于帮助读者们开发出拥有更好结构的、更加健壮的多线程代码，并回避Qt事件循环以及线程使用的常见错误。

先决条件

考虑到本文并不是一个线程编程的泛泛介绍，我们希望你有如下相关知识：

10. 看懂需要勇气，33张人性图！

随笔分类(167)

c# 编程(28)
linux(1)
Qt 编程(41)
Web编程(1)
测试(5)
串口通信(5)
打包部署(1)
分布式开发(4)
框架/模式(2)
面试(1)
设计(3)
数据库(7)
算法(6)
网络编程(4)
闲话闲聊(22)
项目管理(2)
移动开发(12)
正能量(22)

随笔档案(198)

C++基础； Qt 基础：QOjbects , 信号/槽，事件处理； 了解什么是线程、线程与进程间的关系和操作系统； 了解主流操作系统如何启动、停止、等待并结束一个线程； 了解如何使用mutexes, semaphores 和以及wait conditions 来创建一个线程安全/可重入的函数、数据结构、类。 本文我们将沿用如下的名词解释，即

可重入 一个类被称为是可重入的：只要在同一时刻至多只有一个线程访问同一个实例，那么我们说多个线程可以安全地使用各自线程内自己的实例。 一个函数被称为是可重入的：如果每一次函数的调用只访问其独有的数据（译者注：全局变量就不是独有的，而是共享的），那么我们说多个线程可以安全地调用这个函数。 也就是说，类和函数的使用者必须通过一些外部的加锁机制来实现访问对象实例或共享数据的序列化。 线程安全 如果多个线程可以同时使用一个类的对象，那么这个类被称为是线程安全的；如果多个线程可以同时使用一个函数体里的共享数据，那么这个函数被称为线程安全的。 (译者注： 更多可重入(reentrant)和t线程安全(thread-safe)的解释： 对于类，如果它的所有成员函数都可以被不同线程同时调用而不相互影响——即使这些调用是针对同一个类对象，那么该类被定义为线程安全。 对于类，如果其不同实例可以在不同线程中被同时使用而不相互影响，那么该类被定义为可重入。在Qt的定义中，在类这个层次，thread-safe是比reentrant更严格的要求)

事件与事件循环

Qt作为一个事件驱动的工具集，其事件和事件派发起到了核心的作用。本文将不会全面的讨论这个话题，而是会聚焦于与线程相关的一些关键概念。想要了解更多的Qt事件系统专题参见 (这里 [doc.qt.nokia.com] 和 这里 [doc.qt.nokia.com]) （译者注：也欢迎参阅译者写的博文：浅议Qt的事件处理机制一，二）

一个Qt的事件是代表了某件另人感兴趣并已经发生的对象；事件与信号的主要区别在于，事件是针对于与我们应用中一个具体目标对象（而这个对象决定了我们如何处理这个事件），而信号发射则是“漫无目的”。从代码的角度来说，所有的事件实例是QEvent [doc.qt.nokia.com]的子类，并且所有的QObject的派生类可以重载虚函数QObject::event(),从而实现对目标对象实例事件的处理。

事件可以产生于应用程序的内部，也可以来源于外部；比如：

QKeyEvent和QMouseEvent对象代表了与键盘、鼠标相关的交互事件，它们来自于视窗管理程序。 当计时器开始计时，QTimerEvent对象被发送到QObject对象中，它们往往来自于操作系统。 当一个子类对象被添加或删除时，QChildEvent对象会被发送到一个QObject对象重，而它们来自于你的应用程序内部 对于事件来讲，一个重要的事情在于它们并没有在事件产生时被立即派发，而是列入到一个事件队列（Event queue）中，等待以后的某一个时刻发送。分配器（dispatcher ）会遍历事件队列，并且将入栈的事件发送到它们的目标对象当中，因此它们被称为事件循环（Event loop）. 从概念上讲, 下段代码描述了一个事件循环的轮廓:

最新评论

1. Re:解决linux环境下qt groupbox边框不显示问题

```
1:  while (is_active)
2:  {
3:      while (!event_queue_is_empty)
4:          dispatch_next_event();
5:
6:      wait_for_more_events();
7:  }
```

我们是通过运行QCoreApplication::exec()来进入Qt的主体事件循环的；这会引发阻塞，直至QCoreApplication::exit() 或者QCoreApplication::quit() 被调用，进而结束循环。

这个“wait_for_more_events()” 函数产生阻塞，直至某个事件的产生。 如果我们仔细想想，会发现所有在那个时间点产生事件的实体必定是来自于外部的资源（因为当前所有内部事件派发已经结束，事件队列里也没有悬而未决的事件等待处理），因此事件循环被这样唤醒：

- 视窗管理活动（键盘按键、鼠标点击，与视窗的交互等等）；
- socket活动 （有可见的用来读取的数据或者一个可写的非阻塞Socket, 一个新的Socket连接的产生）；
- timers (即计时器开始计时）

你要是把代码贴出来是不是会更好呢？； P

--kangear

2. Re:C# 中静态调用C++dll 和C# 中动态调用C++dll

正是我需要的，谢谢博主

--张杨

3. Re:如何实现双机热备

你这个实现了吗，现在我们也要搞这个，蛋疼。。。

--雨之秋水

4. Re:使用Qt编写模块化插件式应用程序

diao,黧黑，

--cd2011blog

5. Re:Qt 一步一步实现dll调用（附源码）

博客园后台有“文件”标签页，上传后拷贝下载链接即可。

--findumars

阅读排行榜

- 其它线程Post的事件（见后文）。 Unix系统中，视窗管理活动（即X11）通过Socket（Unix 域或者TCP/IP）通知应用程序（事件的产生），因为客户端使用它们与X服务器进行通讯。如果我们决定用一个内部的socketpair(2)来实现跨线程的事件派发，那么视窗管理活动需要唤醒的是

- sockets;
- timers;

这也是*select(2)* 系统调用所做的： 它为视窗管理活动监控了一组描述符，如果一段时间内没有任何活动，它会超时。Qt所要做的是把系统调用select的返回值转换为正确的QEvent子类对象，并将其列入事件队列的栈中，现在你知道事件循环里面装着什么东西了吧:)

为什么需要运行事件循环？

下面的清单并不全，但你会有一幅全景图，你应该能够猜到哪些类需要使用事件循环。

- *Widgets 绘图与交互*： 当派发QPaintEvent事件时，QWidget::paintEvent() 将会被调用。QPaintEvent可以产生于内部的QWidget::update() ，也可以产生于外部的视窗管理（比如，一个显示被隐藏的窗口）。同样的，各种各样的交互（键盘、鼠标等）所对应的事件均需要事件循环来派发。

- **Timers**: 长话短说，当select(2)或相类似的调用超时时，计时器开始计时，因此需要让Qt通过返回事件循环让那些调用为你工作。
- **Networking**: 所以底层的Qt网络类(QTcpSocket, QUdpSocket, QTcpServer等)均被设计成异步的。当你调用read()时，它们仅仅是返回已经可见的数据而已； 当你调用write()时，它们仅是将写操作列入执行计划表待稍后执行。 真正的读写仅发生于事件循环返回的时候。 请注意虽然Qt网络类提供了相应的同步方法（waitFor* 一族），但它们是不被推荐使用的，原因在于他们阻塞了正在等待的事件循环。向QNetworkAccessManager这样的上层类，并不提供同步API而且需要事件循环。

阻塞事件循环

在讨论为什么*你永远都不要阻塞事件循环*之前，让我们尝试着再进一步弄明白到底“阻塞”意味着什么。假定你有一个按钮widget，它被按下时会emit一个信号；还有一个我们下面定义的Worker对象连接了这个信号，而且这个对象的槽做了很多耗时的事情。当你点击完这个按钮后，从上之下的函数调用栈如下所示：

```
1 main(int, char **)
2 QApplication::exec()
3 [...]
```

```
4  QWidget::event(QEvent *)
5  Button::mousePressEvent(QMouseEvent *)
6  Button::clicked()
7  [...]
8  Worker::doWork()
```

在main()中，我们通过调用QApplication::exec() （如上段代码第2行所示）开启了事件循环。视窗管理者发送了鼠标点击事件，该事件被Qt内核捕获，并转换成QMouseEvent ，随后通过QApplication::notify() （notify并没有在上述代码里显示）发送到我们的widget的event()方法中（第4行）。因为Button并没有重载event()，它的基类QWidget方法得以调用。 QWidget::event() 检测出传入的事件是一个鼠标点击，并调用其专有的事件处理器，即Button::mousePressEvent() (第5行)。我们重载了mousePressEvent方法，并发射了Button::clicked()信号（第6行），该信号激活了我们worker对象中十分耗时的Worker::doWork()槽（第8行）。（译者注：如果你对这一段所描述得函数栈的更多细节，请参见浅议Qt的事件处理机制一，二）

当worker对象在繁忙的工作时，事件循环在做什么呢？ 你也许猜到了答案：什么也没做！它分发了鼠标点击事件，并且因等待event handler返回而被阻塞。我们阻塞了事件循环，也就是说，在我们的doWork()槽（第8行）干完活之前再不会有事件被派发了，也再不会有pending的事件被处理。

当事件派发被就此卡住时，widgets 也将不会再刷新自己（QPaintEvent对象将在事件队列里静候），也不能有进一步地与widgets交互的事件发生，计时器也不会在开始计时，网络通讯也将变得迟钝、停滞。更严重的是，许多视窗管理程序会检测到你的应用不再处理事件，从而告诉用户你的程序不再有响应（not responding）. 这就是为什么快速的响应事件并尽可能快的返回事件循环如此重要的原因

强制事件循环

那么，对于需要长时间运行的任务，我们应该怎么做才会不阻塞事件循环？ 一个可行的答案是将这个任务移动另一个线程中：在一节，我们会看到如果去做。一个可能的方案是，在我们的受阻塞的任务中，通过调用QCoreApplication::processEvents() 人工地强迫事件循环运行。QCoreApplication::processEvents() 将处理所有事件队列中的事件并返回给调用者。

另一个可选的强制地重入事件的方案是使用QEventLoop [doc.qt.nokia.com] 类，通过调用QEventLoop::exec() ，我们重入了事件循环，而且我们可以把信号连接到QEventLoop::quit() 槽上使得事件循环退出，如下代码所示：

```
1:  QNetworkAccessManager qnam;
2
```

```
2:  QNetworkReply *reply =
qnam.get(QNetworkRequest(QUrl(...)));
3:  QEventLoop loop;
4:  QObject::connect(reply, SIGNAL(finished()), &loop,
SLOT(quit()));
5:  loop.exec();
6:  /* reply has finished, use it */
```

QNetworkReply 没有提供一个阻塞式的API，而且它要求运行一个事件循环。我们进入到一个局部QEventLoop，并且当回应完成时，局部的事件循环退出。

当重入事件循环是从“其他路径”完成的则要非常小心：它可能会导致无尽的递归循环！让我们回到Button这个例子。如果我们再在doWork() 槽里面调用QCoreApplication::processEvents() ，这时用户又一次点击了button，那么doWork()槽将会再次被调用：

```
main(int, char **)
QApplication::exec()
[...]
QWidget::event(QEvent *)
Button::mousePressEvent(QMouseEvent *)
Button::clicked()
[...]
Worker::doWork() // 实现，内部调用
```

```
QCoreApplication::processEvents() // 我们人工的派发事件而且…
[...]
QWidget::event(QEvent *) // 另一个鼠标点击事件被发送给Button
Button::mousePressEvent(QMouseEvent *)
Button::clicked() // 这里又一次emit了clicked() …
[...]
Worker::doWork() // 完蛋！我们已经递归地调用了doWork槽
```

一个快速并且简单的临时解决办法是把 QEventLoop::ExcludeUserInputEvents 传递给 QCoreApplication::processEvents(), 也就是说，告诉事件循环不要派发任何用户输入事件（事件将简单的呆在队列中）。

同样地，使用一个对象的deleteLater() 来实现异步的删除事件（或者，可能引发某种“关闭（shutdown）”的任何事件）则要警惕事件循环的影响。 （译者注：deleteLater()将在事件循环中删除对象并返回）

```
1:  QObject *object = new QObject;
2:  object->deleteLater();
3:  QEventLoop loop;
4:  loop.exec();
```

```
5:  /* 现在object是一个野指针！ */
```

可以看到，我们并没有用QCoreApplication::processEvents() (从Qt 4.3之后，删除事件不再被派发 ），但是我们确实用到了其他的局部事件循环（像我们QEventLoop 启动的这个循环，或者下面将要介绍的QDialog::exec()）。

切记当我们调用QDialog::exec()或者 QMenu::exec()时，Qt进入了一个局部事件循环。Qt 4.5 以后的版本，QDialog 提供了QDialog::open() 方法用来再不进入局部循环的前提下显示window-modal式的对话框

```
1:  QObject *object = new QObject;
2:  object->deleteLater();
3:  QDialog dialog;
4:  dialog.exec();
5:  /* 现在object是一个野指针！ */
```

Qt 线程类

Qt对线程的支持已经有很多年了（发布于2000年九月22日的Qt2.2引入了QThread类），Qt 4.0版本的release则对其所有所支持平台默认

地是对多线程支持的。（当然你也可以关掉对线程的支持，参见这里）。现在Qt提供了不少类用于处理线程，让你我们首先预览一下：

QThread

QThread 是Qt中一个对线程支持的核心的底层类。 每个线程对象代表了一个运行的线程。由于Qt的跨平台特性，QThread成功隐藏了所有在不同操作系统里使用线程的平台相关性代码。

为了运用QThread从而让代码在一个线程里运行，我们可以创建一个QThread的子类，并重载QThread::run() 方法：

```
class Thread : public QThread {
protected:
void run() {
/* your thread implementation goes here */
}
};
```

接着，我们可以使用：

class Thread : public QThread { protected: void run() { /* your thread implementation goes here */ } };

来真正的启动一个新的线程。 请注意，Qt 4.4版本之后，QThread不再支持抽象类；现在虚函数QThread::run()实际上是简单调用了QThread::exec(),而它启动了线程的事件循环。（更多信息见后文）

QRunnable 和 QThreadPool

QRunnable [doc.qt.nokia.com] 是一种轻量级的、以“run and forget”方式来在另一个线程开启任务的抽象类，为了实现这一功能，我们所需要做的全部事情是派生QRunnable 类，并实现纯虚函数方法run()

class Task : public QRunnable { public: void run() { /* your runnable implementation goes here */ } };

事实上，我们是使用QThreadPool 类来运行一个QRunnable 对象，它维护了一个线程池。通过调用QThreadPool::start(runnable) ，我们把一个QRunnable 放入了QThreadPool的运行队列中；只要线程是可见得，QRunnable 将会被拾起并且在那个线程里运行。尽管所有的Qt应用程序都有一个全局的线程池，且它是通过调用QThreadPool::globalInstance()可见得，但我们总是显式地创建并管理一个私有的QThreadPool 实例。

请注意，QRunnable 并不是一个QObject类，它并没有一个内置的与其他组件显式通讯的方法。你必须使用底层的线程原语（比如收集结构的枷锁保护队列等）来亲自编写代码。

QtConcurrent

QtConcurrent 是一个构建在QThreadPool之上的上层API，它用于处理最普通的并行计算模式：map [en.wikipedia.org], reduce [en.wikipedia.org], and filter [en.wikipedia.org] 。同时，QtConcurrent::run()方法提供了一种便于在另一个线程运行一个函数的方法。

不像QThread 以及QRunnable，QtConcurrent 没有要求我们使用底层的同步原语，QtConcurrent 所有的方法会返回一个QFuture 对象，它包含了结果而且可以用来查询线程计算的状态（它的进度），从而暂停、继续、取消计算。QFutureWatcher 可以用来监听一个 QFuture 进度，并且通过信号和槽与之交互（注意QFuture是一个基于数值的类，它并没有继承自QObject）.

功能比较

\	QThread	QRunnable	QtConcurrent[1]
High level API	✗	✗	✓
Job-oriented	✗	✓	✓
Builtin support for pause/resume/cancel	✗	✗	✓

Can run at a different priority	✓	✗	✗
Can run an event loop	✓	✗	✗

线程与QObjects

线程的事件循环

我们在上文中已经讨论了事件循环，我们可能理所当然地认为在Qt的应用程序中只有一个事件循环，但事实并不是这样：QThread对象在它们所代表的线程中开启了新的事件循环。因此，我们说main 事件循环是由调用main()的线程通过QCoreApplication::exec() 创建的。它也被称做是GUI线程，因为它是界面相关操作唯一允许的进程。一个QThread的局部事件循环可以通过调用QThread::exec() 来开启（它包含在run()方法的内部）

```
class Thread : public QThread {
protected:
void run() {
/* ... initialize ... */
exec();
}
};
```

正如我们之前所提到的，自从Qt 4.4 的QThread::run() 方法不再是一个纯虚函数，它调用了QThread::exec()。就像QCoreApplication，QThread 也有QThread::quit() 和QThread::exit()来停止事件循环。

一个线程的事件循环为驻足在该线程中的所有QObjects派发了所有事件，其中包括在这个线程中创建的所有对象，或是移植到这个线程中的对象。我们说一个QObject的线程依附性（thread affinity）是指某一个线程，该对象驻足在该线程内。我们在任何时间都可以通过调用QObject::thread()来查询线程依附性，它适用于在QThread对象构造函数中构建的对象。

```
class MyThread : public QThread
{
public:
MyThread()
{
otherObj = new QObject;
}
private:
QObject obj;
QObject *otherObj;
QScopedPointer<QObject> yetAnotherObj;
};
```

如上述代码，我们在创建了MyThread 对象后，obj, otherObj, yetAnotherObj 的线程依附性是怎么样的？要回答这个问题，我们必须要看一下创建他们的线程：是这个运行MyThread 构造函数的线程创建了他们。因此，这三个对象并没有驻足在MyThread 线程，而是驻足在创建MyThread 实例的线程中。

要注意的是在QCoreApplication 对象之前创建的QObjects没有依附于某一个线程。因此，没有人会为它们做事件派发处理。（换句话说，QCoreApplication 构建了代表主线程的QThread 对象）

我们可以使用线程安全的QCoreApplication::postEvent() 方法来为某个对象分发事件。它将把事件加入到对象所驻足的线程事件队列中。因此，除非事件对象依附的线程有一个正在运行的事件循环，否则事件不会被派发。

理解QObject和它所有的子类不是线程安全的（尽管是可重入的）非常重要；因此，除非你序列化对象内部数据所有可访问的接口、数据，否则你不能让多个线程同一时刻访问相同的QObject（比如，用一个锁来保护）。请注意，尽管你可以从另一个线程访问对象，但是该对象此时可能正在处理它所驻足的线程事件循环派发给它的事件! 基于这种原因，你不能从另一个线程去删除一个QObject，一定要使用QObject::deleteLater()，它会Post一个事件，目标删除对象最终会

在它所生存的线程中被删除。（译者注：QObject::deleteLater作用是，当控制流回到该对象所依附的线程事件循环时，该对象才会被“本”线程中删除）。

此外，QWidget 和它所有的子类，以及所有与GUI相关的类（即便不是基于QObject的，像QPixmap）并不是可重入的。它们必须专属于GUI线程。

我们可以通过调用QObject::moveToThread()来改变一个QObject的依附性；它将改变这个对象以及它的孩子们的依附性。因为QObject不是线程安全的，我们必须在对象所驻足的线程中使用此函数；也就是说，你只能将对象从它所驻足的线程中推送到其他线程中，而不能从其他线程中拉回来。此外，Qt要求一个QObject的孩子必须与它们的双亲驻足在同一个线程中。这意味着：

你不能使用QObject::moveToThread()作用于有双亲的对象； 你千万不要在一个线程中创建对象的同时把QThread对象自己作为它的双亲。 （译者注：两者不在同一个线程中）：

```
1  class Thread : public QThread {
2  void run() {
3  QObject obj = new QObject(this); // WRONG!!!
4  }
5  };
```

这是因为，QThread 对象驻足在另一个线程中，即QThread 对象它自己被创建的那个线程中。

Qt同样要求所有的对象应该在代表该线程的QThread对象销毁之前得以删除；实现这一点并不难：只要我们所有的对象是在QThread::run() 方法中创建即可。（译者注：run函数的局部变量，函数返回时得以销毁）。

跨线程的信号与槽

接着上面讨论的，我们如何应用驻足在其他线程里的QObject方法呢？Qt提供了一种非常友好而且干净的解决方案：向事件队列post一个事件，事件的处理将以调用我们所感兴趣的方法为主（当然这需要线程有一个正在运行的事件循环）。而触发机制的实现是由moc提供的内省方法实现的（译者注：有关内省的讨论请参见我的另一篇文章Qt的内省机制剖析）：因此，只有信号、槽以及被标记成Q_INVOKABLE的方法才能够被其它线程所触发调用。

静态方法QMetaObject::invokeMethod() 为我们做了如下工作：

```
1 QMetaObject::invokeMethod(object, "methodName",
2 Qt::QueuedConnection,
3 Q_ARG(type1, arg1),
4 Q_ARG(type2, arg2));
```

请注意，因为上面所示的参数需要被在构建事件时进行硬拷贝，参数的自定义型别所对应的类需要提供一个共有的构造函数、析构函数以及拷贝构造函数。而且必须使用注册Qt型别系统所提供的qRegisterMetaType() 方法来注册这一自定义型别。

跨线程的信号槽的工作方式相类似。当我们把信号连接到一个槽的时候，QObject::connect的第五个可选输入参数用来特化这一连接类型：

- direct connection 是指：发起信号的线程会直接触发其所连接的槽；
- queued connection 是指：一个事件被派发到接收者所在的线程中，在这里，事件循环会之后的某一时间将该事件拾起并引起槽的调用；
- blocking queued connection 与queued connection的区别在于，发送者的线程会被阻塞，直至接收者所在线程的事件循环处理发送者发送（入栈）的事件，当连接信号的槽被触发后，阻塞被解除；
- automatic connection (缺省默认参数) 是指： 如果接收者所依附的线程和当前线程是同一个线程，direct connection会被使用。否则使用queued connection。

请注意，在上述四种连接方式当中，发送对象驻足于哪一个线程并不重要！对于automatic connection，Qt会检查触发信号的线程，并且与接收者所驻足的线程相比较从而决定到底使用哪一种连接类型。特别要指出的是：当前的Qt文档的声明(4.7.1) 是错误的：

如果发射者和接受者在同一线程，其行为与Direct Connection相同；，如果发射者和接受者不在同一线程，其行为Queued Connection相同

因为，发送者对象的线程依附性在这里无关紧要。举例子说明

```
view plaincopy to clipboardprint?
class Thread : public QThread
{
Q_OBJECT
signals:
void aSignal();
protected:
void run() {
emit aSignal();
}
};
/* ... */
Thread thread;
Object obj;
```

```
15  QObject::connect(&thread, SIGNAL(aSignal()), &obj,
16  SLOT(aSlot()));
    thread.start();
```

如上述代码，信号aSignal() 将在一个新的线程里被发射（由线程对象所代表）；因为它并不是Object 对象驻足的线程，所以尽管Thread对象thread与Object对象obj在同一个线程，但仍然是queued connection被使用。

（译者注：这里作者分析的很透彻，希望读者仔细揣摩Qt文档的这个错误。 也就是说 发送者对象本身在哪一个线程对与信号槽连接类型不起任何作用，起到决定作用的是接收者对象所驻足的线程以及发射信号（该信号与接受者连接）的线程是不是在同一个线程，本例中aSignal()在新的线程中被发射，所以采用queued connection）。

另外一个常见的错误如下：

```
1  view plaincopy to clipboardprint?
2  class Thread : public QThread
3  {
4  Q_OBJECT
5  slots:
6  void aSlot() {
7  /* ... */
```

```
}
protected:
void run() {
/* ... */
}
};
/* ... */
Thread thread;
Object obj;
QObject::connect(&obj, SIGNAL(aSignal()), &thread,
SLOT(aSlot()));
thread.start();
obj.emitSignal();
```

当“obj”发射了一个aSignal()信号是，哪种连接将被使用呢？你也许已经猜到了：direct connection。这是因为Thread对象实在发射该信号的线程中生存。在aSlot()槽里，我们可能接着去访问线程里的一些成员变量，然而这些成员变量可能同时正在被run()方法访问：这可是导致完美灾难的秘诀。可能你经常在论坛、博客里面找到的解决方案是在线程的构造函数里加一个moveToThread(this)方法。

```
class Thread : public QThread {

```

```
3  Q_OBJECT
4
5  public:
6
7  Thread() {
8
9  moveToThread(this); // 错误
10
11 }
12
13 /* ... */
14
15 };
```

（译注：moveToThread(this)）

这样做确实可以工作(因为现在线程对象的依附性已经发生了改变），但这是一个非常不好的设计。这里的错误在于我们正在误解线程对象的目的（QThread子类）：QThread对象们不是线程；他们是围绕在新产生的线程周围用于控制管理新线程的对象，因此，它们应该用在另一个线程（往往在它们所驻足的那一个线程）

一个比较好而且能够得到相同结果的做法是将“工作”部分从“控制”部分剥离出来，也就是说，写一个QObject子类并使用QObject::moveToThread()方法来改变它的线程依附性：

```
view plaincopy to clipboardprint?
class Worker : public QObject
{
Q_OBJECT
public slots:
void doWork() {
/* ... */
}
};
/* ... */
QThread thread;
Worker worker;
connect(obj, SIGNAL(workReady()), &worker,
SLOT(doWork()));
worker.moveToThread(&thread);
thread.start();
```

我应该什么时候使用线程

当你不得不使用一个阻塞式**API**时

当你需要（通过信号和槽，或者是事件、回调函数）使用一个没有提供非阻塞式API的库或者代码时，为了阻止冻结事件循环的唯一可行的解决方案是开启一个进程或者线程。由于创建一个新的进程的开销显然要比开启一个线程的开销大，后者往往是最常见的一种选择。

这种API的一个很好的例子是地址解析 方法（只是想说我们并不准备谈论蹩脚的第三方API, 地址解析方法它是每个C库都要包含的）,它负责将主机名转化为地址。这个过程涉及到启动一个查询（通常是远程的）系统：域名系统或者叫DNS。尽管通常情况下响应会在瞬间发生，但远程服务器可能会失败：一些数据包可能会丢失，网络连接可能断开等等。简而言之，我们也许要等待几十秒才能得到查询的响应。

UNIX系统可见的标准API只有阻塞式的（不仅过时的gethostbyname(3)是阻塞式的,而且更新的getservbyname(3) 以及getaddrinfo(3)也是阻塞式的）。QHostInfo [doc.qt.nokia.com], 它是一个负责处理域名查找的Qt类，该类使用了QThreadPool 从而使得查询可以在后台进行）（参见here [qt.gitorious.com]）；如果屏蔽了多线程支持，它将切换回到阻塞式API).

另一个简单的例子是图像装载和放大。QImageReader [doc.qt.nokia.com] 和QImage [doc.qt.nokia.com]仅仅提供了阻塞式

方法来从一个设备读取图像，或者放大图像到一个不同的分辨率。如果你正在处理一个非常大的图像，这些处理会持续数（十)秒。

当你想扩展至多核

多线程允许你的程序利用多核系统的优势。因为每个线程都是被操作系统独立调度的，因此如果你的应用运行在这样多核机器上，调度器很可能同时在不同的处理器上运行每个线程。

例如，考虑到一个通过图像集生成缩略图的应用。一个_n_ threads的线程农场（也就是说，一个有着固定数量线程的线程池），在系统中可见的CPU运行一个线程（可参见QThread::idealThreadCount()），可以将缩小图像至缩略图的工作交付给所有的进程，从而有效地提高了并行加速比，它与处理器的数量成线性关系。（简单的讲，我们认为CPU正成为一个瓶颈)。

什么时候你可能不想别人阻塞

这是一个很高级的话题，你可以忽略该小节。一个比较好的例子来自于Webkit里使用的QNetworkAccessManager 。Webkit是一个时髦的浏览器引擎，也就是说，它是一组用于处理网页的布局和显示的类集合。使用Webkit的Qt widget是QWebView。

QNetworkAccessManager 是一个用于处理HTTP任何请求和响应的Qt类，我们可以把它当作一个web浏览器的网络引擎；所有的网络访

问被同一个QNetworkAccessManager 以及它的QNetworkReplys 驻足的线程所处理。

尽管在网络处理时不使用线程是一个很好的主意，它也有一个很大的缺点：如果你没有从socket中尽快地读取数据，内核的缓存将会被填满，数据包可能开始丢失而且传输速率也将迅速下降。

Sokcet活动（即，从一个socket读取一些数据的可见性）由Qt的事件循环管理。阻塞事件循环因此会导致传输性能的损失，因为没有人会被通知将有数据可以读取（从而没人会去读数据）。

但究竟什么会阻塞事件循环呢？令人沮丧地回答: WebKit它自己！只要有数据被接收到，WebKit便用其来布局网页。不幸地是，布局处理过程相当复杂，而且开销巨大。因此，它阻塞事件循环的一小段时间足以影响到正在进行地传输（宽带连接这里起到了作用，在短短几秒内就可填满内核缓存）。

总结一下上述所发生的事情：

WebKit提出了一个请求； 一些响应数据开始到达； WebKit开始使用接收到的数据布局网页，从而阻塞了事件循环； 数据被OS接受，但没有一个正在运行的事件循环为之派发，所以并没有被QNetworkAccessManager sockets所读取； 内核缓存将被填满，传输将变慢。 网页的总体装载时间因其自发引起的传输速率降低而变得越来越坏。

诺基亚的工程师正在试验一个支持多线程的QNetworkAccessManager来解决这个问题。请注意因为QNetworkAccessManagers 和QNetworkReplys 是QObjects，他们不是线程安全的，因此你不能简单地将他们移到另一个线程中并且继续在你的线程中使用他们，原因在于，由于事件将被随后线程的事件循环所派发，他们可能同时被两个线程访问：你自己的线程以及已经它们驻足的线程。

什么时候不需要使用线程

> If you think you need threads then your processes are too fat. — Bradley T. Hughes

计时器

这也许是线程滥用最坏的一种形式。如果我们不得不重复调用一个方法（比如每秒），许多人会这样做：

```
view plaincopy to clipboardprint?
// 非常之错误
while (condition) {
doWork();
```

```
sleep(1); // this is sleep(3) from the C library
}
```

然后他们发现这会阻塞事件循环，因此决定引入线程：

```
view plaincopy to clipboardprint?
// 错误
class Thread : public QThread {
protected:
void run() {
while (condition) {
// notice that "condition" may also need volatiness and
mutex protection
// if we modify it from other threads (!)
doWork();
sleep(1); // this is QThread::sleep()
}
}
};
```

一个更好也更简单的获得相同效果的方法是使用timers,即一个QTimer[doc.qt.nokia.com]对象，并设置一秒的超时时间，并让

doWork方法成为它的槽：

```
view plaincopy to clipboardprint?
class Worker : public QObject
{
Q_OBJECT
public:
Worker() {
connect(&timer, SIGNAL(timeout()), this,
SLOT(doWork()));
timer.start(1000);
}
private slots:
void doWork() {
/* ... */
}
private:
QTimer timer;
};
```

所有我们需要做的就是运行一个事件循环，然后doWork()方法将会被每隔秒钟调用一次。

网络/状态机

一个处理网络操作非常之常见的设计模式如下：

```
1  view plaincopy to clipboardprint?
2  socket->connect(host);
3  socket->waitForConnected();
4  data = getData();
5  socket->write(data);
6  socket->waitForBytesWritten();
7  socket->waitForReadyRead();
8  socket->read(response);
9  reply = process(response);
10 socket->write(reply);
11 socket->waitForBytesWritten();
12 /* ... and so on ... */
```

不用多说，各种各样的waitFor*()函数阻塞了调用者使其无法返回到事件循环，UI被冻结等等。请注意上面的这段代码并没有考虑到错误处理，否则它会更加地笨重。这个设计中非常错误的地方是我们正在忘却网络编程是异步的设计，如果我们构建一个同步的处理方法，则是自己给自己找麻烦。为了解决这个问题，许多人简单得将这些代码移到另一个线程中。

另一个更加抽象的例子：

```
view plaincopy to clipboardprint?
result = process_one_thing();
if (result->something())
process_this();
else
process_that();
wait_for_user_input();
input = read_user_input();
process_user_input(input);
/* ... */
```

它多少反映了网络编程相同的陷阱。

让我们回过头来从更高的角度来想一下我们这里正在构建的代码：我们想创造一个状态机，用以反映某类的输入并相对应的作某些动作。比如，上面的这段网络代码，我们可能想做如下这些事情：

空闲→ 正在连接 (当调用connectToHost()); 正在连接→ 已经连接(当connected() 信号被发射); 已经连接→ 发送录入数据 (当我们发送录入的数据给服务器); 发送录入数据 → 录入 (服务器响应一个ACK) 发送录入数据→ 录入错误(服务器响应一个NACK) 以此类推。

现在，有很多种方式来构建状态机（Qt甚至提供了QStateMachine[doc.qt.nokia.com]类），最简单的方式是用一个枚举值（及，一个整数）来记忆当前的状态。我们可以这样重写以下上面的代码：

```
class Object : public QObject
{
Q_OBJECT
enum State {
State1, State2, State3 /* and so on */
};
State state;
public:
Object() : state(State1)
{
connect(source, SIGNAL(ready()), this,
SLOT(doWork()));
}
private slots:
void doWork() {
switch (state) {
case State1:
/* ... */
state = State2;
```

```
20  break;
21  case State2:
22  /* ... */
23  state = State3;
24  break;
25  /* etc. */
26  }
27  }
    };
```

那么“souce”对象和它的信号“ready()” 究竟是什么? 我们想让它们是什么就是什么：比如说，在这个例子中，我们可能想把我们的槽连接到socket的QAbstractSocket::connected() 以及 QIODevice::readyRead() 信号中，当然，我们也可以简单地在我们的用例中加更多的槽（比如一个槽用于处理错误情况，它将会被 QAbstractSocket::error() 信号所通知）。这是一个真正的异步的，信号驱动的设计！

分解任务拆成不同的块

假如我们有一个开销很大的计算，它不能够轻易的移到另一个线程中（或者说它根本不能被移动，举个例子，它必须运行在GUI线程

中）。如果我们能将计算拆分成小的块，我们就能返回到事件循环，让它来派发事件，并让它激活处理下一个块相应的函数。如果我们还记得queued connections是怎么实现的，那么会觉得这是很容易能够做到的：一个事件派发到接收者所驻足的线程的事件循环；当事件被传递，相应的槽随之被激活。

我们可以使用特化QMetaObject::invokeMethod() 的激活类型为Qt::QueuedConnection 来得到相同的结果；这需要函数是可激活的。因此它需要一个槽或者用Q_INVOKABLE宏来标识。如果我们同时想给函数中传入参数，他们需要使用Qt元对象类型系统里的qRegisterMetaType()进行注册。请看下面这段代码：

```
1  class Worker : public QObject
2  {
3  Q_OBJECT
4  public slots:
5  void startProcessing()
6  {
7  processItem(0);
8  }
9  void processItem(int index)
10 {
11 /* process items[index] ... */
12 if (index < numberOfItems)
13 QMetaObject::invokeMethod(this,
```

```
14  "processItem",
15  Qt::QueuedConnection,
16  Q_ARG(int, index + 1));
17  }
18  };
```

作者：江南烟雨居

出处：http://www.cnblogs.com/newstart//

分类: Qt 编程

« 上一篇：C#源码500份

» 下一篇：给明年依然年轻的我们

posted on 2013-07-20 11:02 夜&枫 阅读(13272) 评论(1) 编辑 收藏

发表评论

#1楼 2015-03-02 14:59 | tadpole999

就是写的好，学习了。谢谢

支持(0) 反对(0)

刷新评论 刷新页面 返回顶部

注册用户登录后才能发表评论，请 登录 或 注册，访问网站首页。

最新**IT**新闻:

· 百度发布百度医疗大脑，在医疗人工智能领域正面PK谷歌、IBM
· Twitter CEO发内部信提振士气 绝口不提出售
· Oculus升级Gear VR头盔软件 禁止其连接Note 7
· 我国在超冷原子量子模拟领域取得重大突破
· AAFA呼吁再将阿里划入“恶名市场”名单，这名单是个什么鬼？
» 更多新闻...

最新知识库文章:

· 陈皓：什么是工程师文化？
· 没那么难，谈CSS的设计模式
· 程序猿媳妇儿注意事项
· 可是姑娘，你为什么要编程呢？
· 知其所以然（以算法学习为例）
» 更多知识库文章...