<https://www.cnblogs.com/lvdongjie/p/6564949.html>

## 使用线程

基本上有种使用线程的场合：

* 通过利用处理器的多个核使处理速度更快。
* 为保持GUI线程或其他高实时性线程的响应，将耗时的操作或阻塞的调用移到其他线程。

### 何时使用其他技术替代线程

开发人员使用线程时需要非常小心。启动线程是很容易的，但确保所有共享数据保持一致很难。遇到问题往往很难解决，这是由于在一段时间内它可能只出现一次或只在特定的硬件配置下出现。**在创建线程来解决某些问题之前，应该考虑一些替代的技术** ：

|  |  |
| --- | --- |
| **替代技术** | **注解** |
| QEventLoop::processEvents() | 在一个耗时的计算操作中反复调用QEventLoop::processEvents() 可以防止界面的假死。尽管如此，这个方案可伸缩性并不太好，因为该函数可能会被调用地过于频繁或者不够频繁。 |
| QTimer | 后台处理操作有时可以方便地使用Timer安排在一个在未来的某一时刻执行的槽中来完成。在没有其他事件需要处理时，时间隔为0的定时器超时事件被相应 |
| QSocketNotifier  QNetworkAccessManager  QIODevice::readyRead() | 这是一个替代技术，替代有一个或多个线程在慢速网络执行阻塞读的情况。只要响应部分的计算可以快速执行，这种设计比在线程中实现的同步等待更好。与线程相比这种设计更不容易出错且更节能(energy efficient)。在许多情况下也有性能优势。 |

一般情况下，建议只使用安全和经过测试的方案而避免引入特设线程的概念。QtConcurrent 提供了一个将任务分发到处理器所有的核的易用接口。线程代码完全被隐藏在 QtConcurrent 框架下，所以你不必考虑细节。尽管如此，QtConcurrent 不能用于线程运行时需要通信的情况，而且它也不应该被用来处理阻塞操作。

### 应该使用 Qt 线程的哪种技术？

有时候，你需要的不仅仅是在另一线程的上下文中运行一个函数。您可能需要有一个生存在另一个线程中的对象来为 GUI线程提供服务。也许你想在另一个始终运行的线程中来轮询硬件端口并在有关注的事情发生时发送信号到GUI线程。Qt为开发多线程应用程序提供了多种 不同的解决方案。**解决方案的选择依赖于新线程的目的以及线程的生命周期。**

|  |  |  |
| --- | --- | --- |
| **生命周期** | **开发任务** | **解决方案** |
| 一次调用 | 在另一个线程中运行一个函数，函数完成时退出线程 | 编写函数，使用QtConcurrent::run 运行它 |
| 派生QRunnable，使用QThreadPool::globalInstance()->start()运行它 |  |  |
| 派生QThread，重新实现QThread::run() ，使用QThread::start()运行它 |  |  |
| 一次调用 | 需要操作一个容器中所有的项。使用处理器所有可用的核心。一个常见的例子是从图像列表生成缩略图。 | QtConcurrent 提供了map()函你数来将操作应用到容器中的每一个元素，提供了fitler()函数来选择容器元素，以及指定reduce函数作为选项来组合剩余元素。 |
| 一次调用 | 一个耗时运行的操作需要放入另一个线程。在处理过程中，状态信息需要发送会GUI线程。 | 使用QThread，重新实现run函数并根据需要发送信号。使用信号槽的queued连接方式将信号连接到GUI线程的槽函数。 |
| 持久运行 | 生存在另一个线程中的对象，根据要求需要执行不同的任务。这意味着工作线程需要双向的通讯。 | 派生一个QObject对象并实现需要的信号和槽，将对象移动到一个运行有事件循环的线程中并通过queued方式连接的信号槽进行通讯。 |
| 持久运行 | 生存在另一个线程中的对象，执行诸如轮询端口等重复的任务并与GUI线程通讯。 | 同上，但是在工作线程中使用一个定时器来轮询。尽管如此，处理轮询的最好的解决方案是彻底避免它。有时QSocketNotifer是一个替代。 |

## Qt线程基础

QThread是一个非常便利的跨平台的对平台原生线程的抽象。启动一个线程是很简单的。让我们看一个简短的代码：生成一个在线程内输出"hello"并退出的线程。

// hellothread/hellothread.h class HelloThread : public QThread { Q\_OBJECT private: void run(); };

我们从QThread派生出一个类，并重新实现run方法。

// hellothread/hellothread.cpp void HelloThread::run() { qDebug() << "hello from worker thread " << thread()->currentThreadId(); }

run方法中包含将在另一个线程中运行的代码。在本例中，一个包含线程ID的消息被打印出来。  QThread::start() 将在另一个线程中被调用。

int main(int argc, char \*argv[]) { QCoreApplication app(argc, argv); HelloThread thread; thread.start(); qDebug() << "hello from GUI thread " << app.thread()->currentThreadId(); thread.wait(); // do not exit before the thread is completed! return 0; }

### QObject与线程

QObject有线程关联(thread affinity)[如何翻译？关联？依附性？dbzhang800 20110618]，换句话说，它生存于一个特定的线程。这意味着，在创建时QObject保存了到当前线程的指针。当事件使用postEvent()被 派发时，这个信息变得很有用。事件被放置到相应线程的事件循环中。如果QObject所依附的线程没有事件循环，该事件将永远不会被传递。

要启动事件循环，必须在run()内调用exec()。线程关联可以通过moveToThread()来更改。

如上所述，当从其他线程调用对象的方法时开发人员必须始终保持谨慎。线程关联不会改变这种状况。 Qt文档中将一些方法标记为线程安全。postEvent()就是一个值得注意的例子。一个线程安全的方法可以同时在不同的线程被调用。

通常情况下并不会并发访问的一些方法，在其他线程调用对象的非线程安全的方法在出现造成意想不到行为的并发访问前 数千次的访问可能都是工作正常的。编写测试代码不能完全确保线程的正确性，但它仍然是重要的。在Linux上，Valgrind和Helgrind有助于 检测线程错误。

QThread的内部结构非常有趣：

* QThread并不生存于执行run()的新线程内。它生存于旧线程中。
* QThread的大多数成员方法是线程的控制接口，并设计成从旧线程中被调用。不要使用moveToThread()将该接口移动到新创建的线程中；调用moveToThread(this)被视为不好的实践。
* exec()和静态方法usleep()、msleep()、sleep()要在新创建的线程中调用。
* QThread子类中定义的其他成员可在两个线程中访问。开发人员负责访问的控制。一个典型的策略是在start()被调用前设置成员变量。一旦 工作线程开始运行，主线程不应该操作其他成员。当工作线程终止后，主线程可以再次访问其他成员。这是一个在线程开始前传递参数并在结束后收集结果的便捷的 策略。

QObject必须始终和parent在同一个线程。对于在run()中生成的对象这儿有一个惊人的后果：

void HelloThread::run() { QObject \*object1 = new QObject(this); //error, parent must be in the same thread QObject object2; // OK QSharedPointer <QObject> object3(new QObject); // OK }

### 使用互斥量保护数据的完整

互斥量是一个拥有lock()和unlock()方法并记住它是否已被锁定的对象。互斥量被设计为从多个线程调 用。如果信号量未被锁定lock()将立即返回。下一次从另一个线程调用会发现该信号量处于锁定状态，然后lock()会阻塞线程直到其他线程调用 unlock()。此功能可以确保代码段将在同一时间只能由一个线程执行。

### 使用事件循环防止数据破坏

Qt的事件循环对线程间的通信是一个非常有价值的工具。每个线程都可以有它自己的事件循环。在另一个线程中调用一个槽的一个安全的方法是将调用放置到另一个线程的事件循环中。这可以确保目标对象调用另一个的成员函数之前可以完成当前正在运行的成员函数。

那么，如何才能把一个成员调用放于一个事件循环中？ Qt的有两种方法来做这个。一种方法是通过queued信号槽连接；另一种是使用QCoreApplication::postEvent()派发一个事 件。queued的信号槽连接是异步执行的信号槽连接。内部实现是基于posted的事件。信号的参数放入事件循环后信号函数的调用将立即返回。

连接的槽函数何时被执行依赖于事件循环其他的其他操作。

通过事件循环通信消除了我们使用互斥量时所面临的死锁问题。这就是我们为什么推荐使用事件循环，而不是使用互斥量锁定对象的原因。

### 处理异步执行

一种获得一个工作线程的结果的方法是等待线程终止。在许多情况下，一个阻塞等待是不可接受的。阻塞等待的替代方法 是异步的结果通过posted事件或者queued信号槽进行传递。由于操作的结果不会出现在源代码的下一行而是在位于源文件其他部分的一个槽中，这会产 生一定的开销，因为，但在位于源文件中其他地方的槽。 Qt开发人员习惯于使用这种异步行为工作，因为它非常相似于GUI程序中使用的的事件驱动编程。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

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：

**[cpp]**  [view plain](http://blog.csdn.net/fuyajun01/article/details/7075979) [copy](http://blog.csdn.net/fuyajun01/article/details/7075979)

1. class HelloWorldTask : public QRunnable
2. {
3. void run()
4. {
5. qDebug() << "Hello world from thread" << QThread::currentThread();
6. }
7. }
9. HelloWorldTask \*hello = new HelloWorldTask();
10. // QThreadPool takes ownership and deletes 'hello' automatically
11. 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, waitcondition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](http://blog.csdn.net/fuyajun01/article/details/7075979#map)() | appliesa function to every item in a container, modifying the itemsin-place |
| [QtConcurrent::mapped](http://blog.csdn.net/fuyajun01/article/details/7075979#mapped)() | islike map(), except that it returns a new container with themodifications |
| [QtConcurrent::mappedReduced](http://blog.csdn.net/fuyajun01/article/details/7075979#mappedReduced)() | islike mapped(), except that the modified results are reduced orfolded into a single result. |
| [QtConcurrent::filter](http://blog.csdn.net/fuyajun01/article/details/7075979#filter)() | litems from a container based on the result of a filter function. |
| [QtConcurrent::filtered](http://blog.csdn.net/fuyajun01/article/details/7075979#filtered)() | islike filter(), except that it returns a new container with thefiltered results |
| [QtConcurrent::filteredReduced](http://blog.csdn.net/fuyajun01/article/details/7075979#filteredReduced)() | islike filtered(), except that the filtered results are reduced orfolded into a single result |
| [QtConcurrent::run](http://blog.csdn.net/fuyajun01/article/details/7075979#run)() | runsa function in another thread. |
| [QFutureIterator](http://blog.csdn.net/fuyajun01/article/details/qfutureiterator.html) | allowsiterating through results available via [QFuture](http://blog.csdn.net/fuyajun01/article/details/qfuture.html). |
| [QFutureWatcher](http://blog.csdn.net/fuyajun01/article/details/qfuturewatcher.html) | allowsmonitoring a [QFuture](http://blog.csdn.net/fuyajun01/article/details/qfuture.html) usingsignals-and-slots. |
| [QFutureSynchronizer](http://blog.csdn.net/fuyajun01/article/details/qfuturesynchronizer.html) | isa convenience class that automatically synchronizes severalQFutures. |

代码实例：

**[cpp]**  [view plain](http://blog.csdn.net/fuyajun01/article/details/7075979) [copy](http://blog.csdn.net/fuyajun01/article/details/7075979)

1. using namespace QtConcurrent;
2. void hello(QString name)
3. {
4. qDebug() << "Hello" << name << "from" << QThread::currentThread();
5. }
6. int main(int argc, char \*\*argv)
7. {
8. QApplication app(argc, argv);
9. QFuture<void> f1 = run(hello, QString("Alice"));
10. QFuture<void> f2 = run(hello, QString("Bob"));
11. f1.waitForFinished();
12. f2.waitForFinished();
13. return app.exec();
14. }

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

### [QtConcurrent::run() 的使用](http://blog.csdn.net/zengcccc/article/details/7010891)

2011-11-25 10:36  155人阅读  [评论](http://blog.csdn.net/zengcccc/article/details/7010891#comments)(0)  收藏  [举报](http://blog.csdn.net/zengcccc/article/details/7010891#report)

QFuture<T> run(const Class \*object, T (Class::\*fn)(Param1, Param2, Param3, Param4, Param5) const, const Arg1 &arg1,const Arg2 &arg2, const Arg3 &arg3, const Arg4 &arg4, const Arg5 &arg5)

run()函数的原型如上,此函数是QtConcurrent命名空间里的函数.主要功能是令启动一个线程来执行一个函数.Concurrent的英文示意就是并发的意思.

下面简要的介绍run()函数的使用方法:

1.首先要有一个需要在另外开启的线程中执行的函数:

void thread\_add(QObject \*receiver,int a,int b)

{

QString message=QString("%1 + %2 = %3").arg(a).arg(b).arg(a+b);

QApplication::postEvent(receiver,new ProgressEvent(true, message));

}

函数在线程中运行完毕后会向receiver发送一个消息,来返回结果.

2.有了要在线程中运行的函数,再来看看怎么启动线程来运行这个函数

void MainWindow::on\_pushButton\_clicked()

{

for(int i=0;i<9;i++)

for(int j=0;j<9;j++)

QtConcurrent::run(thread\_add,this,i,j);

}

点击一个按钮就会运行这段代码,然后启动8\*8=64个线程,线程要运行的函数就是thread\_add(之前定义的),消息接收对象就是MainWindow这个类的实例

3.线程得到了运行会发送消息给MainWindow,MainWindow重新实现bool MainWindow::event ( QEvent \* event )处理接收到的消息,并显示出来

bool MainWindow::event ( QEvent \* event )

{

if (event->type() ==

static\_cast<QEvent::Type>(ProgressEvent::EventId)) {

ProgressEvent \*progressEvent =

static\_cast<ProgressEvent\*>(event);

Q\_ASSERT(progressEvent);

ui->teLog->append(progressEvent->message);

return true;

}

return QMainWindow::event(event);

}

再给出自定义的消息结构

struct ProgressEvent : public QEvent

{

enum {EventId = QEvent::User};

explicit ProgressEvent(bool saved\_, const QString &message\_)

: QEvent(static\_cast<Type>(EventId)),

saved(saved\_), message(message\_) {}

const bool saved;

const QString message;

};

原文地址：<http://zjqzy03080312.blog.163.com/blog/static/1857428072012525114256880/>