# 多线程的40个面试题总结（上）

周末在微信公共号看到一篇关于《线程的40个道题》的文章，由于今年工作之前参加过几次面试，所以觉得这篇文章总结的很好

只要读者朋友们耐心看完，并且在阅读过程中遇到自己疑惑的地方时自己能动手查一查做一做，我相信以后遇到很多关于线程上的问题都会迎刃而解。

打算用**两个部分**来写，第一便于读者阅读，第二谁也不能一口吃个胖子，相信我有的知识是需要时间的积累才能获得的，没有时间的积累谁也不能成为专家，只有时间才能将你打磨的更加完美。

*原文出自“去哪技术沙龙”公众号 40个多线程问题总结，没有地址实在无法标注原文地址*

这些多线程的问题，有些来源于各大网站、有些来源于自己的思考。可能有些问题网上有、可能有些问题对应的答案也有、也可能有些各位网友也都看过，但是本文写作的重心就是所有的问题都会按照自己的理解回答一遍，不会去看网上的答案。

40个问题汇总

1、多线程有什么用？

一个可能在很多人看来很扯淡的一个问题：我会用多线程就好了，还管它有什么用？在我看来，这个回答更扯淡。所谓”知其然知其所以然”，”会用”只是”知其然”，”为什么用”才是”知其所以然”，只有达到”知其然知其所以然”的程度才可以说是把一个知识点运用自如。OK，下面说说我对这个问题的看法：

（1）发挥多核CPU的优势

随着工业的进步，现在的笔记本、台式机乃至商用的应用服务器至少也都是双核的，4核、8核甚至16核的也都不少见，如果是单线程的程序，那么在双核CPU上就浪费了50%，在4核CPU上就浪费了75%。单核CPU上所谓的”多线程”那是假的多线程，同一时间处理器只会处理一段逻辑，只不过线程之间切换得比较快，看着像多个线程”同时”运行罢了。多核CPU上的多线程才是真正的多线程，它能让你的多段逻辑同时工作，多线程，可以真正发挥出多核CPU的优势来，达到充分利用CPU的目的。

（2）防止阻塞

从程序运行效率的角度来看，单核CPU不但不会发挥出多线程的优势，反而会因为在单核CPU上运行多线程导致线程上下文的切换，而降低程序整体的效率。但是单核CPU我们还是要应用多线程，就是为了防止阻塞。试想，如果单核CPU使用单线程，那么只要这个线程阻塞了，比方说远程读取某个数据吧，对端迟迟未返回又没有设置超时时间，那么你的整个程序在数据返回回来之前就停止运行了。多线程可以防止这个问题，多条线程同时运行，哪怕一条线程的代码执行读取数据阻塞，也不会影响其它任务的执行。

（3）便于建模

这是另外一个没有这么明显的优点了。假设有一个大的任务A，单线程编程，那么就要考虑很多，建立整个程序模型比较麻烦。但是如果把这个大的任务A分解成几个小任务，任务B、任务C、任务D，分别建立程序模型，并通过多线程分别运行这几个任务，那就简单很多了。

2、创建线程的方式

比较常见的一个问题了，一般就是两种：

（1）继承Thread类

（2）实现Runnable接口

至于哪个好，不用说肯定是后者好，因为实现接口的方式比继承类的方式更灵活，也能减少程序之间的耦合度，面向接口编程也是设计模式6大原则的核心。

3、start()方法和run()方法的区别

只有调用了start()方法，才会表现出多线程的特性，不同线程的run()方法里面的代码交替执行。如果只是调用run()方法，那么代码还是同步执行的，必须等待一个线程的run()方法里面的代码全部执行完毕之后，另外一个线程才可以执行其run()方法里面的代码。

7、什么是线程安全

又是一个理论的问题，各式各样的答案有很多，我给出一个个人认为解释地最好的：如果你的代码在多线程下执行和在单线程下执行永远都能获得一样的结果，那么你的代码就是线程安全的。

这个问题有值得一提的地方，就是线程安全也是有几个级别的：

（1）不可变

像String、Integer、Long这些，都是final类型的类，任何一个线程都改变不了它们的值，要改变除非新创建一个，因此这些不可变对象不需要任何同步手段就可以直接在多线程环境下使用

（2）绝对线程安全

不管运行时环境如何，调用者都不需要额外的同步措施。要做到这一点通常需要付出许多额外的代价，Java中标注自己是线程安全的类，实际上绝大多数都不是线程安全的，不过绝对线程安全的类，Java中也有，比方说CopyOnWriteArrayList、CopyOnWriteArraySet

（3）相对线程安全

相对线程安全也就是我们通常意义上所说的线程安全，像Vector这种，add、remove方法都是原子操作，不会被打断，但也仅限于此，如果有个线程在遍历某个Vector、有个线程同时在add这个Vector，99%的情况下都会出现ConcurrentModificationException，也就是fail-fast机制。

（4）线程非安全

这个就没什么好说的了，ArrayList、LinkedList、HashMap等都是线程非安全的类

9、一个线程如果出现了运行时异常会怎么样

如果这个异常没有被捕获的话，这个线程就停止执行了。另外重要的一点是：如果这个线程持有某个某个对象的监视器，那么这个对象监视器会被立即释放

10、如何在两个线程之间共享数据

通过在线程之间共享对象就可以了，然后通过wait/notify/notifyAll、await/signal/signalAll进行唤起和等待，比方说阻塞队列BlockingQueue就是为线程之间共享数据而设计的

11、sleep方法和wait方法有什么区别

这个问题常问，sleep方法和wait方法都可以用来放弃CPU一定的时间，不同点在于如果线程持有某个对象的监视器，sleep方法不会放弃这个对象的监视器，wait方法会放弃这个对象的监视器

12、生产者消费者模型的作用是什么

这个问题很理论，但是很重要：

（1）通过平衡生产者的生产能力和消费者的消费能力来提升整个系统的运行效率，这是生产者消费者模型最重要的作用

（2）解耦，这是生产者消费者模型附带的作用，解耦意味着生产者和消费者之间的联系少，联系越少越可以独自发展而不需要收到相互的制约

13、ThreadLocal有什么用

简单说ThreadLocal就是一种以空间换时间的做法，在每个Thread里面维护了一个以开地址法实现的ThreadLocal.ThreadLocalMap，把数据进行隔离，数据不共享，自然就没有线程安全方面的问题了

14、为什么wait()方法和notify()/notifyAll()方法要在同步块中被调用

这是JDK强制的，wait()方法和notify()/notifyAll()方法在调用前都必须先获得对象的锁

15、wait()方法和notify()/notifyAll()方法在放弃对象监视器时有什么区别

wait()方法和notify()/notifyAll()方法在放弃对象监视器的时候的区别在于：wait()方法立即释放对象监视器，notify()/notifyAll()方法则会等待线程剩余代码执行完毕才会放弃对象监视器。

16、为什么要使用线程池

避免频繁地创建和销毁线程，达到线程对象的重用。另外，使用线程池还可以根据项目灵活地控制并发的数目。

17、怎么检测一个线程是否持有对象监视器

我也是在网上看到一道多线程面试题才知道有方法可以判断某个线程是否持有对象监视器：Thread类提供了一个holdsLock(Object obj)方法，当且仅当对象obj的监视器被某条线程持有的时候才会返回true，注意这是一个static方法，这意味着“某条线程”指的是当前线程。

18、synchronized和ReentrantLock的区别

synchronized是和if、else、for、while一样的关键字，ReentrantLock是类，这是二者的本质区别。既然ReentrantLock是类，那么它就提供了比synchronized更多更灵活的特性，可以被继承、可以有方法、可以有各种各样的类变量，ReentrantLock比synchronized的扩展性体现在几点上：

（1）ReentrantLock可以对获取锁的等待时间进行设置，这样就避免了死锁

（2）ReentrantLock可以获取各种锁的信息

（3）ReentrantLock可以灵活地实现多路通知

另外，二者的锁机制其实也是不一样的。ReentrantLock底层调用的是Unsafe的park方法加锁，synchronized操作的应该是对象头中mark word，这点我不能确定。

19、ConcurrentHashMap的并发度是什么

ConcurrentHashMap的并发度就是segment的大小，默认为16，这意味着最多同时可以有16条线程操作ConcurrentHashMap，这也是ConcurrentHashMap对Hashtable的最大优势，任何情况下，Hashtable能同时有两条线程获取Hashtable中的数据吗？

20、ReadWriteLock是什么

首先明确一下，不是说ReentrantLock不好，只是ReentrantLock某些时候有局限。如果使用ReentrantLock，可能本身是为了防止线程A在写数据、线程B在读数据造成的数据不一致，但这样，如果线程C在读数据、线程D也在读数据，读数据是不会改变数据的，没有必要加锁，但是还是加锁了，降低了程序的性能。

因为这个，才诞生了读写锁ReadWriteLock。ReadWriteLock是一个读写锁接口，ReentrantReadWriteLock是ReadWriteLock接口的一个具体实现，实现了读写的分离，读锁是共享的，写锁是独占的，读和读之间不会互斥，读和写、写和读、写和写之间才会互斥，提升了读写的性能。

# Qt之QThread（深入理解）

**简述**

为了让程序尽快响应用户操作，在开发应用程序时经常会使用到线程。对于耗时操作如果不使用线程，UI界面将会长时间处于停滞状态，这种情况是用户非常不愿意看到的，我们可以用线程来解决这个问题。

前面，已经介绍了QThread常用的两种方式：

* Worker-Object
* 子类化QThread

下面，我们来看看子类化QThread在日常中的应用。

* [简述](http://blog.csdn.net/liang19890820/article/details/52186626#%E7%AE%80%E8%BF%B0)
* [子类化QThread](http://blog.csdn.net/liang19890820/article/details/52186626#%E5%AD%90%E7%B1%BB%E5%8C%96qthread)
* [线程休眠](http://blog.csdn.net/liang19890820/article/details/52186626#%E7%BA%BF%E7%A8%8B%E4%BC%91%E7%9C%A0)
* [在主线程中更新UI](http://blog.csdn.net/liang19890820/article/details/52186626#%E5%9C%A8%E4%B8%BB%E7%BA%BF%E7%A8%8B%E4%B8%AD%E6%9B%B4%E6%96%B0ui)
* [避免多次connect](http://blog.csdn.net/liang19890820/article/details/52186626#%E9%81%BF%E5%85%8D%E5%A4%9A%E6%AC%A1connect)
* [优雅地结束线程](http://blog.csdn.net/liang19890820/article/details/52186626#%E4%BC%98%E9%9B%85%E5%9C%B0%E7%BB%93%E6%9D%9F%E7%BA%BF%E7%A8%8B)
* [更多参考](http://blog.csdn.net/liang19890820/article/details/52186626#%E6%9B%B4%E5%A4%9A%E5%8F%82%E8%80%83)

大多数情况下，多线程耗时操作会与UI进行交互，比如：显示进度、加载等待。。。让用户明确知道目前的状态，并对结果有一个直观的预期，甚至有趣巧妙的设计，能让用户爱上等待，把等待看成一件很美好的事。

**子类化QThread**

下面，是一个使用多线程操作UI界面的示例 - 更新进度条。与此同时，分享在此过程中有可能遇到的问题及解决方法。

定义一个WorkerThread类，让其继承自QThread，并重写run()函数，每隔50毫秒更新当前值，然后发射resultReady()信号（用于更新进度条）。

#include <QThread>

class WorkerThread : public QThread

{

Q\_OBJECT

public:

explicit WorkerThread(QObject \*parent = 0)

: QThread(parent)

{

qDebug() << "Worker Thread : " << QThread::currentThreadId();

}

protected:

virtual void run() Q\_DECL\_OVERRIDE {

qDebug() << "Worker Run Thread : " << QThread::currentThreadId();

int nValue = 0;

while (nValue < 100)

{

// 休眠50毫秒

msleep(50);

++nValue;

// 准备更新

emit resultReady(nValue);

}

}

signals:

void resultReady(int value);

};

构建一个主界面 - 包含按钮、进度条，当点击“开始”按钮时，启动线程，更新进度条。

class MainWindow : public CustomWindow

{

Q\_OBJECT

public:

explicit MainWindow(QWidget \*parent = 0)

: CustomWindow(parent)

{

qDebug() << "Main Thread : " << QThread::currentThreadId();

// 创建开始按钮、进度条

QPushButton \*pStartButton = new QPushButton(this);

m\_pProgressBar = new QProgressBar(this);

//设置文本、进度条取值范围

pStartButton->setText(QString::fromLocal8Bit("开始"));

m\_pProgressBar->setFixedHeight(25);

m\_pProgressBar->setRange(0, 100);

m\_pProgressBar->setValue(0);

QVBoxLayout \*pLayout = new QVBoxLayout();

pLayout->addWidget(pStartButton, 0, Qt::AlignHCenter);

pLayout->addWidget(m\_pProgressBar);

pLayout->setSpacing(50);

pLayout->setContentsMargins(10, 10, 10, 10);

setLayout(pLayout);

// 连接信号槽

connect(pStartButton, SIGNAL(clicked(bool)), this, SLOT(startThread()));

}

~MainWindow(){}

private slots:

// 更新进度

void handleResults(int value)

{

qDebug() << "Handle Thread : " << QThread::currentThreadId();

m\_pProgressBar->setValue(value);

}

// 开启线程

void startThread()

{

WorkerThread \*workerThread = new WorkerThread(this);

connect(workerThread, SIGNAL(resultReady(int)), this, SLOT(handleResults(int)));

// 线程结束后，自动销毁

connect(workerThread, SIGNAL(finished()), workerThread, SLOT(deleteLater()));

workerThread->start();

}

private:

QProgressBar \*m\_pProgressBar;

WorkerThread m\_workerThread;

};

显然，UI界面、Worker构造函数、槽函数处于同一线程（主线程），而run()函数处于另一线程（次线程）。

Main Thread : 0x34fc   
Worker Thread : 0x34fc   
Worker Run Thread : 0x4038   
Handle Thread : 0x34fc

由于信号与槽连接类型默认为“Qt::AutoConnection”，在这里相当于“Qt::QueuedConnection”。

也就是说，槽函数在接收者的线程（主线程）中执行。

**注意：**信号与槽的连接类型，请参考：[Qt之Threads和QObjects](http://blog.csdn.net/liang19890820/article/details/52054152)中“跨线程的信号和槽”部分。

**线程休眠**

上述示例中，通过在run()函数中调用msleep(50)，线程会每隔50毫秒让当前的进度值加1，然后发射一个resultReady()信号，其余时间什么都不做。在这段空闲时间，线程不占用任何的系统资源。当下一次CPU时钟来临时，它会继续执行。

QThread提供了静态的、平台独立的休眠函数：sleep()、msleep()、usleep()，允许秒，毫秒和微秒来区分，函数接受整型数值作为参数，以表明线程挂起执行的时间。当休眠时间结束，线程就会获得CPU时钟，将继续执行它的指令。

想象一下，日常用的电脑，如果我们需要离开一段时间，可以将它设置为休眠状态，为了节约用电，同时响应国家政策 - 走绿色、环保之道。

可以尝试注释掉休眠部分的代码，这时，由于没有任何耗时操作，会造成频繁地更新UI。所以，为了保证界面的流畅性，同时确保进度的更新在人眼可接受的范围内，我们应在必要的时候加上适当时间的休眠。

**在主线程中更新UI**

当连接方式更改为“Qt::DirectConnection”时：

connect(workerThread, SIGNAL(resultReady(int)), this, SLOT(handleResults(int)), Qt::DirectConnection);

再次点击“开始”按钮，会很失望，因为它会出现一个异常，描述如下：

ASSERT failure in QCoreApplication::sendEvent: “Cannot send events to objects owned by a different thread. Current thread e346e8. Receiver customWidget’ (of type ‘MainWindow’) was created in thread 4186a0”, file kernel\qcoreapplication.cpp, line 553

显然，UI界面、Worker构造函数处于同一线程（主线程），而run()函数、槽函数处于同一线程（次线程）。

Main Thread : 0x2c6c   
Worker Thread : 0x2c6c   
Worker Run Thread : 0x4704   
Handle Thread : 0x4704

之所以会出现这种情况是因为Qt做了限制（其它大多数GUI编程也一样），不允许在其它线程（非主线程）中访问UI控件，这么做主要是怕在多线程环境下对界面控件进行操作会出现不可预知的情况。

所以，不难理解，由于在槽函数（次线程）中更新了UI，所以，会引起以上错误。

**避免多次connect**

当多次点击“开始”按钮的时候，就会多次connect()，从而启动多个线程，同时更新进度条。

为了避免这个问题，我们修改如下：

class MainWindow : public CustomWindow

{

Q\_OBJECT

public:

explicit MainWindow(QWidget \*parent = 0)

: CustomWindow(parent)

{

// ...

connect(&m\_workerThread, SIGNAL(resultReady(int)), this, SLOT(handleResults(int)));

}

~MainWindow(){}

private slots:

// ...

void startThread()

{

if (!m\_workerThread.isRunning())

m\_workerThread.start();

}

private:

WorkerThread m\_workerThread;

};

将connect添加在构造函数中，保证了信号槽的正常连接。在线程start()之前，可以使用isFinished()和isRunning()来查询线程的状态，判断线程是否正在运行，以确保线程的正常启动。

**优雅地结束线程**

如果一个线程运行完成，就会结束。可很多情况并非这么简单，由于某种特殊原因，当线程还未执行完时，我们就想中止它。

不恰当的中止往往会引起一些未知错误。比如：当关闭主界面的时候，很有可能次线程正在运行，这时，就会出现如下提示：

QThread: Destroyed while thread is still running

这是因为次线程还在运行，就结束了UI主线程，导致事件循环结束。这个问题在使用线程的过程中经常遇到，尤其是耗时操作。

在此问题上，常见的两种人：

* 直接忽略此问题。
* 强制中止 - terminate()。

大多数情况下，当程序退出时，次线程也许会正常退出。这时，虽然抱着侥幸心理，但隐患依然存在，也许在极少数情况下，就会出现Crash。

正如前面提到过terminate()，比较危险，不鼓励使用。线程可以在代码执行的任何点被终止。线程可能在更新数据时被终止，从而没有机会来清理自己，解锁等等。。。总之，只有在绝对必要时使用此函数。

举一个简单的例子：当你的哥们正处于长时间的酣睡状态时，你想要叫醒他，但是采取的措施却是泼一盆凉水，想象一下后果？这凉爽 - O(∩\_∩)O哈哈~。

所以，我们应该采取合理的措施来优雅地结束线程，一般思路：

1. 发起线程退出操作，调用quit()或exit()。
2. 等待线程完全停止，删除创建在堆上的对象。
3. 适当的使用wait()（用于等待线程的退出）和合理的算法。

下面介绍两种方式：

* QMutex互斥锁 + bool成员变量。

这种方式是Qt4.x中比较常用的，主要是利用“QMutex互斥锁 + bool成员变量”的方式来保证共享数据的安全性（可以完全参照下面的requestInterruption()源码写法）。

#include <QThread>

#include <QMutexLocker>

class WorkerThread : public QThread

{

Q\_OBJECT

public:

explicit WorkerThread(QObject \*parent = 0)

: QThread(parent),

m\_bStopped(false)

{

qDebug() << "Worker Thread : " << QThread::currentThreadId();

}

~WorkerThread()

{

stop();

quit();

wait();

}

void stop()

{

qDebug() << "Worker Stop Thread : " << QThread::currentThreadId();

QMutexLocker locker(&m\_mutex);

m\_bStopped = true;

}

protected:

virtual void run() Q\_DECL\_OVERRIDE {

qDebug() << "Worker Run Thread : " << QThread::currentThreadId();

int nValue = 0;

while (nValue < 100)

{

// 休眠50毫秒

msleep(50);

++nValue;

// 准备更新

emit resultReady(nValue);

// 检测是否停止

{

QMutexLocker locker(&m\_mutex);

if (m\_bStopped)

break;

}

// locker超出范围并释放互斥锁

}

}

signals:

void resultReady(int value);

private:

bool m\_bStopped;

QMutex m\_mutex;

};

为什么要加锁？很简单，是为了共享数据段操作的互斥。   
何时需要加锁？在形成资源竞争的时候，也就是说，多个线程有可能访问同一共享资源的时候。

当主线程调用stop()更新m\_bStopped的时候，run()函数也极有可能正在访问它（这时，他们处于不同的线程），所以存在资源竞争，因此需要加锁，保证共享数据的安全性。

* Qt5以后：requestInterruption() + isInterruptionRequested()

这两个接口是Qt5.x引入的，使用很方便：

class WorkerThread : public QThread

{

Q\_OBJECT

public:

explicit WorkerThread(QObject \*parent = 0)

: QThread(parent)

{

}

~WorkerThread() {

// 请求终止

requestInterruption();

quit();

wait();

}

protected:

virtual void run() Q\_DECL\_OVERRIDE {

// 是否请求终止

while (!isInterruptionRequested())

{

// 耗时操作

}

}

};

在耗时操作中使用isInterruptionRequested()来判断是否请求终止线程，如果没有，则一直运行；当希望终止线程的时候，调用requestInterruption()即可。

正如侯捷所言：「源码面前，了无秘密」。如果还心存疑虑，我们不妨来看看requestInterruption()、isInterruptionRequested()的源码：

void QThread::requestInterruption()

{

Q\_D(QThread);

QMutexLocker locker(&d->mutex);

if (!d->running || d->finished || d->isInFinish)

return;

if (this == QCoreApplicationPrivate::theMainThread) {

qWarning("QThread::requestInterruption has no effect on the main thread");

return;

}

d->interruptionRequested = true;

}

bool QThread::isInterruptionRequested() const

{

Q\_D(const QThread);

QMutexLocker locker(&d->mutex);

if (!d->running || d->finished || d->isInFinish)

return false;

return d->interruptionRequested;

}

^\_^，内部实现居然也用了互斥锁QMutex，这样我们就可以放心地使用了。

**更多参考**

* [Qt之线程基础](http://blog.csdn.net/liang19890820/article/details/52046574)
* [Qt之可重入与线程安全](http://blog.csdn.net/liang19890820/article/details/52048324)
* [Qt之Threads和QObjects](http://blog.csdn.net/liang19890820/article/details/52054152)
* [Qt之Concurrent框架](http://blog.csdn.net/liang19890820/article/details/52065512)
* [Qt之Concurrent Map和Map-Reduce](http://blog.csdn.net/liang19890820/article/details/52066381)
* [Qt之QThread](http://blog.csdn.net/liang19890820/article/details/51800024)