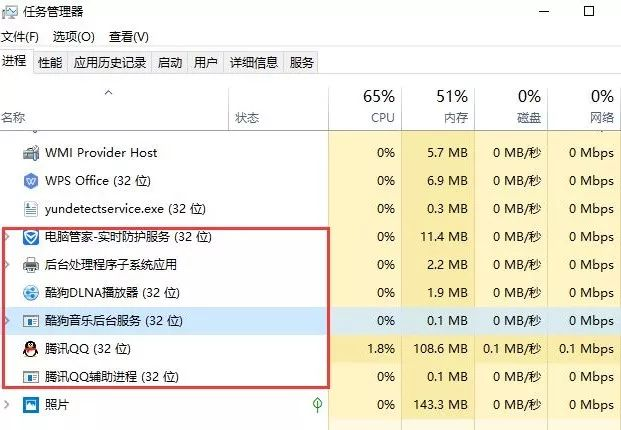
记得今年3月份刚来杭州面试的时候，有一家公司的技术总监问了我这样一个问题：你来说说有哪些线程安全的类？我心里一想，这我早都背好了，稀里哗啦说了一大堆。

他又接着问：那你再来说说什么是线程安全？——然后我就GG了。说真的，我们整天说线程安全，但是对于什么是线程安全我们真的了解吗？之前的我真的是了解甚微，那么我们今天就来聊聊这个问题。

在探讨线程安全之前，我们先来聊聊什么是进程。

什么是进程？

电脑中时会有很多单独运行的程序，每个程序有一个独立的进程，而进程之间是相互独立存在的。比如下图中的QQ、酷狗播放器、电脑管家等等。

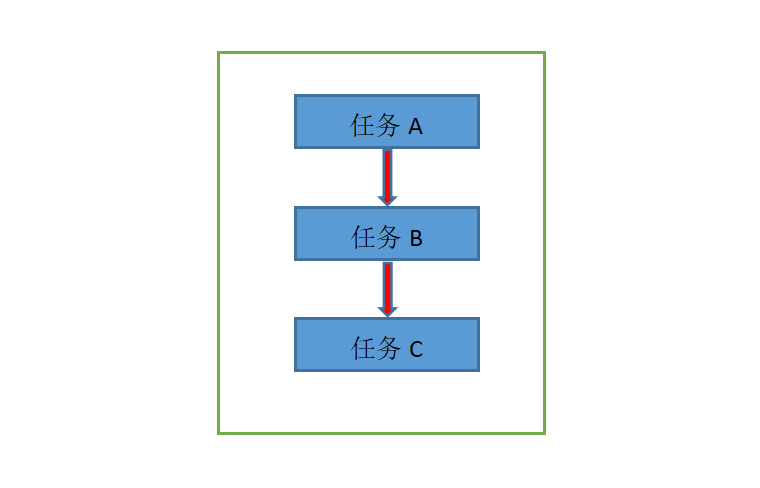


什么是线程？

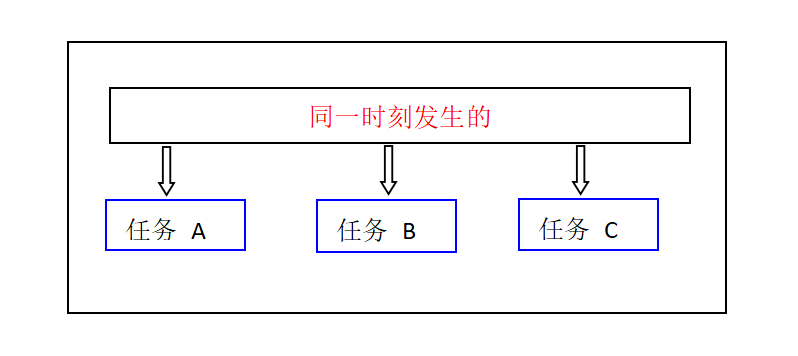
进程想要执行任务就需要依赖线程。换句话说，就是进程中的最小执行单位就是线程，并且一个进程中至少有一个线程。

那什么是多线程？提到多线程这里要说两个概念，就是串行和并行，搞清楚这个，我们才能更好地理解多线程。

**串行:**其实是相对于单条线程来执行多个任务来说的，我们就拿下载文件来举个例子：当我们下载多个文件时，在串行中它是按照一定的顺序去进行下载的，也就是说，必须等下载完A之后才能开始下载B，它们在时间上是不可能发生重叠的。



**并行**：下载多个文件，开启多条线程，多个文件同时进行下载，这里是严格意义上的，在同一时刻发生的，并行在时间上是重叠的。



了解了这两个概念之后，我们再来说说什么是**多线程**。举个例子，我们打开腾讯管家，腾讯管家本身就是一个程序，也就是说它就是一个进程，它里面有很多的功能，我们可以看下图，能查杀病毒、清理垃圾、电脑加速等众多功能。

按照单线程来说，无论你想要清理垃圾、还是要病毒查杀，那么你必须先做完其中的一件事，才能做下一件事，这里面是有一个执行顺序的。

如果是多线程的话，我们其实在清理垃圾的时候，还可以进行查杀病毒、电脑加速等等其他的操作，这个是严格意义上的同一时刻发生的，没有执行上的先后顺序。



以上就是，一个进程运行时产生了多个线程。

在了解完这个问题后，我们又需要去了解一个使用多线程不得不考虑的问题——线程安全。

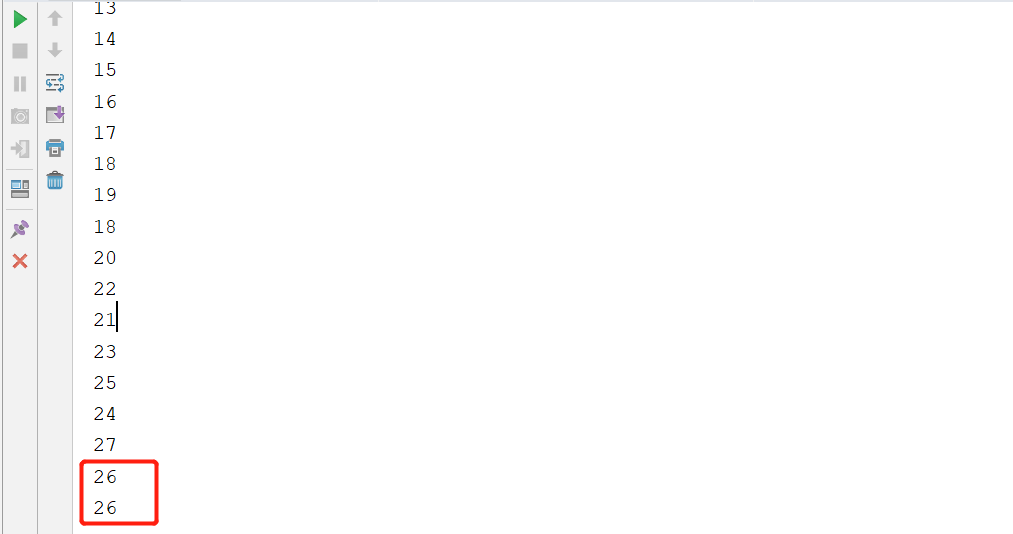
今天我们不说如何保证一个线程的安全，我们聊聊什么是线程安全？因为我之前面试被问到了，说真的，我之前真的不是特别了解这个问题，我们好像只学了如何确保一个线程安全，却不知道所谓的安全到底是什么！

什么是线程安全？

既然是线程安全问题，那么毫无疑问，所有的隐患都是在多个线程访问的情况下产生的，也就是我们要确保在多条线程访问的时候，我们的程序还能按照我们预期的行为去执行，我们看一下下面的代码。

Integer count = 0;  
   public void getCount() {  
       count ++;  
       System.out.println(count);  
   }

很简单的一段代码，下面我们就来统计一下这个方法的访问次数，多个线程同时访问会不会出现什么问题，我开启的3条线程，每个线程循环10次，得到以下结果：



我们可以看到，这里出现了两个26，出现这种情况显然表明这个方法根本就不是线程安全的，出现这种问题的原因有很多。

最常见的一种，就是我们A线程在进入方法后，拿到了count的值，刚把这个值读取出来，还没有改变count的值的时候，结果线程B也进来的，那么导致线程A和线程B拿到的count值是一样的。

那么由此我们可以了解到，这确实不是一个线程安全的类，因为他们都需要操作这个共享的变量。其实要对线程安全问题给出一个明确的定义，还是蛮复杂的，我们根据我们这个程序来总结下什么是线程安全。

当多个线程访问某个方法时，不管你通过怎样的调用方式、或者说这些线程如何交替地执行，我们在主程序中不需要去做任何的同步，这个类的结果行为都是我们设想的正确行为，那么我们就可以说这个类是线程安全的。

搞清楚了什么是线程安全，接下来我们看看Java中确保线程安全最常用的两种方式。先来看段代码。

public void threadMethod(int j) {    int i = 1;    j = j + i;}

大家觉得这段代码是线程安全的吗？

毫无疑问，它绝对是线程安全的，我们来分析一下，为什么它是线程安全的？

我们可以看到这段代码是没有任何状态的，就是说我们这段代码，不包含任何的作用域，也没有去引用其他类中的域进行引用，它所执行的作用范围与执行结果只存在它这条线程的局部变量中，并且只能由正在执行的线程进行访问。当前线程的访问，不会对另一个访问同一个方法的线程造成任何的影响。

两个线程同时访问这个方法，因为没有共享的数据，所以他们之间的行为，并不会影响其他线程的操作和结果，所以说无状态的对象，也是线程安全的。

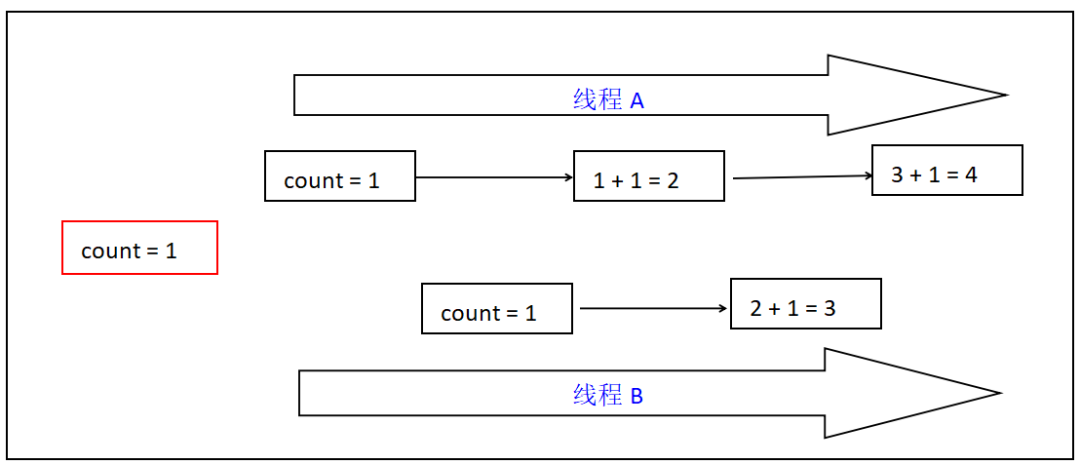
添加一个状态呢？

如果我们给这段代码添加一个状态，添加一个count，来记录这个方法并命中的次数，每请求一次count+1，那么这个时候这个线程还是安全的吗？

public class ThreadDemo {   int count = 0; *// 记录方法的命中次数   public void threadMethod(int j) {       count++ ;       int i = 1;       j = j + i;   }}*

很明显已经不是了，单线程运行起来确实是没有任何问题的，但是当出现多条线程并发访问这个方法的时候，问题就出现了，我们先来分析下count+1这个操作。

进入这个方法之后首先要读取count的值，然后修改count的值，最后才把这把值赋值给count，总共包含了三步过程：“读取”一>“修改”一>“赋值”，既然这个过程是分步的，那么我们先来看下面这张图，看看你能不能看出问题：



可以发现，count的值并不是正确的结果，当线程A读取到count的值，但是还没有进行修改的时候，线程B已经进来了，然后线程B读取到的还是count为1的值，正因为如此所以我们的count值已经出现了偏差，那么这样的程序放在我们的代码中，是存在很多的隐患的。

如何确保线程安全？

既然存在线程安全的问题，那么肯定得想办法解决这个问题，怎么解决？我们说说常见的几种方式。

**1、synchronized**

synchronized关键字，就是用来控制线程同步的，保证我们的线程在多线程环境下，不被多个线程同时执行，确保我们数据的完整性，使用方法一般是加在方法上。

public class ThreadDemo {   int count = 0; *// 记录方法的命中次数   public synchronized void threadMethod(int j) {       count++ ;       int i = 1;       j = j + i;   }}*

这样就可以确保我们的线程同步了，同时这里需要注意一个大家平时忽略的问题，首先synchronized锁的是括号里的对象，而不是代码，其次，对于非静态的synchronized方法，锁的是对象本身也就是this。

当synchronized锁住一个对象之后，别的线程如果想要获取锁对象，那么就必须等这个线程执行完释放锁对象之后才可以，否则一直处于等待状态。

注意点：虽然加synchronized关键字，可以让我们的线程变得安全，但是我们在用的时候，也要注意缩小synchronized的使用范围，如果随意使用时很影响程序的性能，别的对象想拿到锁，结果你没用锁还一直把锁占用，这样就有点浪费资源。

**2、Lock**

先来说说它跟synchronized有什么区别吧，Lock是在Java1.6被引入进来的，Lock的引入让锁有了可操作性，什么意思？就是我们在需要的时候去手动的获取锁和释放锁，甚至我们还可以中断获取以及超时获取的同步特性，但是从使用上说Lock明显没有synchronized使用起来方便快捷。我们先来看下一般是如何使用的：

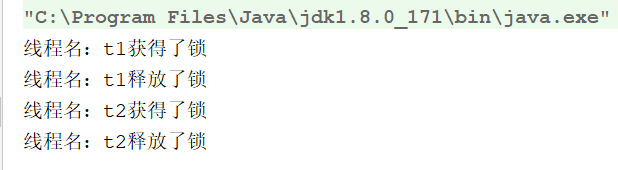
private Lock lock = new ReentrantLock(); *// ReentrantLock是Lock的子类   private void method(Thread thread){       lock.lock(); // 获取锁对象       try {           System.out.println("线程名："+thread.getName() + "获得了锁");           // Thread.sleep(2000);       }catch(Exception e){           e.printStackTrace();       } finally {           System.out.println("线程名："+thread.getName() + "释放了锁");           lock.unlock(); // 释放锁对象       }   }*

进入方法我们首先要获取到锁，然后去执行我们业务代码，这里跟synchronized不同的是，Lock获取的所对象需要我们亲自去进行释放，为了防止我们代码出现异常，所以我们的释放锁操作放在finally中，因为finally中的代码无论如何都是会执行的。

写个主方法，开启两个线程测试一下我们的程序是否正常：

public static void main(String[] args) {       LockTest lockTest = new LockTest();       *// 线程1       Thread t1 = new Thread(new Runnable() {           @Override           public void run() {               // Thread.currentThread()  返回当前线程的引用               lockTest.method(Thread.currentThread());           }       }, "t1");       // 线程2       Thread t2 = new Thread(new Runnable() {           @Override           public void run() {               lockTest.method(Thread.currentThread());           }       }, "t2");       t1.start();       t2.start();   }*

结果：



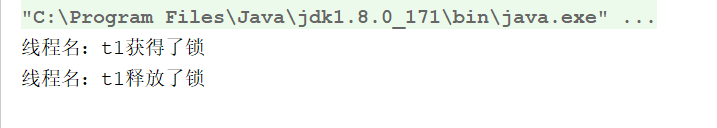
可以看出我们的执行，是没有任何问题的。

其实在Lock还有几种获取锁的方式，我们这里再说一种，就是tryLock()这个方法跟Lock()是有区别的，Lock在获取锁的时候，如果拿不到锁，就一直处于等待状态，直到拿到锁，但是tryLock()却不是这样的，tryLock是有一个Boolean的返回值的，如果没有拿到锁，直接返回false，停止等待，它不会像Lock()那样去一直等待获取锁。

我们来看下代码：

private void method(Thread thread){       *// lock.lock(); // 获取锁对象       if (lock.tryLock()) {           try {               System.out.println("线程名："+thread.getName() + "获得了锁");               // Thread.sleep(2000);           }catch(Exception e){               e.printStackTrace();           } finally {               System.out.println("线程名："+thread.getName() + "释放了锁");               lock.unlock(); // 释放锁对象           }       }   }*

结果：我们继续使用刚才的两个线程进行测试可以发现，在线程t1获取到锁之后，线程t2立马进来，然后发现锁已经被占用，那么这个时候它也不在继续等待。

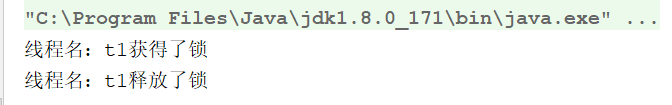


似乎这种方法，感觉不是很完美，如果我第一个线程，拿到锁的时间，比第二个线程进来的时间还要长，是不是也拿不到锁对象？

那我能不能，用一中方式来控制一下，让后面等待的线程，可以等待5秒，如果5秒之后，还获取不到锁，那么就停止等，其实tryLock()是可以进行设置等待的相应时间的。

private void method(Thread thread) throws InterruptedException {       *// lock.lock(); // 获取锁对象       // 如果2秒内获取不到锁对象，那就不再等待       if (lock.tryLock(2,TimeUnit.SECONDS)) {           try {               System.out.println("线程名："+thread.getName() + "获得了锁");               // 这里睡眠3秒               Thread.sleep(3000);           }catch(Exception e){               e.printStackTrace();           } finally {               System.out.println("线程名："+thread.getName() + "释放了锁");               lock.unlock(); // 释放锁对象           }       }   }*

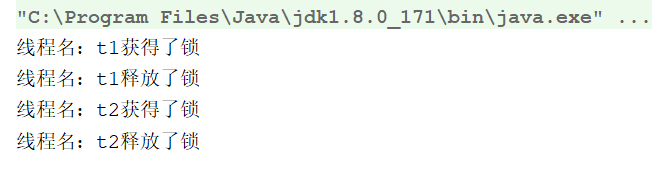
结果：看上面的代码，我们可以发现，虽然我们获取锁对象的时候，可以等待2秒，但是我们线程t1在获取锁对象之后，执行任务缺花费了3秒，那么这个时候线程t2是不在等待的。



我们再来改一下这个等待时间，改为5秒，再来看下结果：

private void method(Thread thread) throws InterruptedException {       *// lock.lock(); // 获取锁对象       // 如果5秒内获取不到锁对象，那就不再等待       if (lock.tryLock(5,TimeUnit.SECONDS)) {           try {               System.out.println("线程名："+thread.getName() + "获得了锁");           }catch(Exception e){               e.printStackTrace();           } finally {               System.out.println("线程名："+thread.getName() + "释放了锁");               lock.unlock(); // 释放锁对象           }       }   }*

结果：这个时候我们可以看到，线程t2等到5秒获取到了锁对象，执行了任务代码。



以上就是使用Lock，来保证我们线程安全的方式。