计算机二级:Java多线程编程的常见陷阱计算机二级考试 PDF转换可能丢失图片或格式，建议阅读原文 https://www.100test.com/kao_ti2020/581/2021_2022__E8_AE_A1_E7_AE_97_E6_9C_BA_E4_c97_581239.htm 1、在构造函数中启动线程 我在很多代码中都看到这样的问题，在构造函数中启动一个线程，类似这样： public class A{ public A(){ this.x=1. this.y=2. this.thread=new MyThread(). this.thread.start(). } ..... } 这个会引起什么问题呢？如果有个类B继承了类A，依据java类初始化的顺序，A的构造函数一定会在B的构造函数调用前被调用，那么thread线程也将在B被完全初始化之前启动，当thread运行时使用到了类A中的某些变量，那么就可能使用的不是你预期中的值，因为在B的构造函数中你可能赋给这些变量新的值。也就是说此时将有两个线程在使用这些变量，而这些变量却没有同步。 解决这个问题有两个办法：将A设置为final，不可继承；或者提供单独的start方法用来启动线程，而不是放在构造函数中。 2、不完全的同步 都知道对一个变量同步的有效方式是用synchronized保护起来，synchronized可能是对象锁，也可能是类锁，看你是类方法还是实例方法。但是，当你将某个变量在A方法中同步，那么在变量出现的其他地方，你也需要同步，除非你允许弱可见性甚至产生错误值。类似这样的代码： class A{ int x. public int getX(){ return x. } public synchronized void setX(int x) { this.x=x. } } x的setter方法有同步，然而getter方法却没有，那么就无法保证其他线程通过getX得到的x是最新的值。事实上，这里的setX的同步是没有必要的，因为对int的写入是原子的，这一点JVM规范已经保证，多个同步没有任何意义；当然，如果

这里不是int，而是double或者long，那么getX和setX都将需要同步，因为double和long都是64位，写入和读取都是分成两个32位来进行（这一点取决于jvm的实现，有的jvm实现可能保证对long和double的read、write是原子的），没有保证原子性。类似上面这样的代码，其实都可以通过声明变量为volatile来解决。 3、在使用某个对象当锁时，改变了对象的引用，导致同步失效。 这也是很常见的错误，类似下面的代码： synchronized(array[0]) { ..... array[0]=new A(). ..... } 同步块使用array[0]作为锁，然而在同步块中却改变了array[0]指向的引用。分析下这个场景，第一个线程获取了array[0]的锁，第二个线程因为无法获取array[0]而等待，在改变了array[0]的引用后，第三个线程获取了新的array[0]的锁，第一和第三两个线程持有的锁是不一样的，同步互斥的目的就完全没有达到了。这样代码的修改，通常是将锁声明为final变量或者引入业务无关的锁对象，保证在同步块内不会被修改引用。 4、没有在循环中调用wait（）。 wait和notify用于实现条件变量，你可能知道需要在同步块中调用wait和notify，为了保证条件的改变能做到原子性和可见性。常常看见很多代码做到了同步，却没有在循环中调用wait，而是使用if甚至没有条件判断： synchronized(lock) { if(isEmpty() lock.wait(). } 对条件的判断是使用if，这会造成什么问题呢？在判断条件之前可能调用notify或者notifyAll，那么条件已经满足，不会等待，这没什么问题。在条件没有满足，调用了wait（）方法，释放lock锁并进入等待休眠状态。如果线程是在正常情况下，也就是条件被改变之后被唤醒，那么没有任何问题，条件满足继续执行下面的逻辑操作。问题在于线程可能被意外甚至恶意唤

醒，由于没有再次进行条件判断，在条件没有被满足的情况下，线程执行了后续的操作。意外唤醒的情况，可能是调用了notifyAll，可能是有人恶意唤醒，也可能是很少情况下的自动苏醒（称为“ 伪唤醒 ”）。因此为了防止这种条件没有满足就执行后续操作的情况，需要在被唤醒后再次判断条件，如果条件不满足，继续进入等待状态，条件满足，才进行后续操作。 synchronized(lock) { while(isEmpty() lock.wait(). } 没有进行条件判断就调用wait的情况更严重，因为在等待之前可能notify已经被调用，那么在调用了wait之后进入等待休眠状态后就无法保证线程苏醒过来。 5、同步的范围过小或者过大。 同步的范围过小，可能完全没有达到同步的目的；同步的范围过大，可能会影响性能。同步范围过小的一个常见例子是误认为两个同步的方法一起调用也是将同步的，需要记住的是Atomic Atomic ！=Atomic. Map map=Collections.synchronizedMap(new HashMap()). if(!map.containsKey("a")){ map.put("a", value). } 这是一个很典型的错误，map是线程安全的，containskey和put方法也是线程安全的，然而两个线程安全的方法被组合调用就不一定是线程安全的了。因为在containsKey和put之间，可能有其他线程抢先put进了a，那么就可能覆盖了其他线程设置的值，导致值的丢失。解决这一问题的方法就是扩大同步范围，因为对象锁是可重入的，因此在线程安全方法之上再同步相同的锁对象不会有问题。 Map map = Collections.synchronizedMap(new HashMap()). synchronized (map) { if (!map.containsKey("a")) { map.put("a", value). } } 注意，加大锁的范围，也要保证使用的是同一个锁，不然很可能造

成死锁。 Collections.synchronizedMap（new HashMap（））使用的锁是map本身，因此没有问题。当然，上面的情况现在更推荐使用ConcurrentHashMap，它有putIfAbsent方法来达到同样的目的并且满足线程安全性。 同步范围过大的例子也很多，比如在同步块中new大对象，或者调用费时的IO操作（操作数据库，webservice等）。不得不调用费时操作的时候，一定要指定超时时间，例如通过URLConnection去invoke某个URL时就要设置connect timeout和read timeout，防止锁被独占不释放。同步范围过大的情况下，要在保证线程安全的前提下，将不必要同步的操作从同步块中移出。 100Test 下载频道开通，各类考试题目直接下载。详细请访问 www.100test.com