编写多线程Java应用程序常见问题 PDF转换可能丢失图片或格式，建议阅读原文
https://www.100test.com/kao_ti2020/145/2021_2022__E7_BC_96_E5_86_99_E5_A4_9A_E7_c104_145445.htm 几乎所有使用 AWT 或 Swing 编写的画图程序都需要多线程。但多线程程序会造成许多困难，刚开始编程的开发者常常会发现他们被一些问题所折磨，例如不正确的程序行为或死锁。 在本文中，我们将探讨使用多线程时遇到的问题，并提出那些常见陷阱的解决方案。 线程是什么？一个程序或进程能够包含多个线程，这些线程可以根据程序的代码执行相应的指令。多线程看上去似乎在并行执行它们各自的工作，就像在一台计算机上运行着多个处理机一样。在多处理机计算机上实现多线程时，它们确实可以并行工作。和进程不同的是，线程共享地址空间。也就是说，多个线程能够读写相同的变量或数据结构。编写多线程程序时，你必须注意每个线程是否干扰了其他线程的工作。可以将程序看作一个办公室，如果不需要共享办公室资源或与其他人交流，所有职员就会独立并行地工作。某个职员若要和其他人交谈，当且仅当该职员在“ 听 ”且他们两说同样的语言。此外，只有在复印机空闲且处于可用状态（没有仅完成一半的复印工作，没有纸张阻塞等问题）时，职员才能够使用它。在这篇文章中你将看到，在 Java 程序中互相协作的线程就好像是在一个组织良好的机构中工作的职员。 在多线程程序中，线程可以从准备就绪队列中得到，并在可获得的系统 CPU 上运行。操作系统可以将线程从处理器移到准备就绪队列或阻塞队列中，这种情况可以认为是处理器“ 挂起 ”了该线程。同样，Java 虚拟机 (JVM) 也可以控

制线程的移动在协作或抢先模型中从准备就绪队列中将进程移到处理器中，于是该线程就可以开始执行它的程序代码。协作式线程模型允许线程自己决定什么时候放弃处理器来等待其他的线程。程序开发员可以精确地决定某个线程何时会被其他线程挂起，允许它们与对方有效地合作。缺点在于某些恶意或是写得不好的线程会消耗所有可获得的 CPU 时间，导致其他线程“饥饿”。 在抢占式线程模型中，操作系统可以在任何时候打断线程。通常会在它运行了一段时间（就是所谓的一个时间片）后才打断它。这样的结果自然是没有线程能够不公平地长时间霸占处理器。然而，随时可能打断线程就会给程序开发员带来其他麻烦。同样使用办公室的例子，假设某个职员抢在另一人前使用复印机，但打印工作在未完成的时候离开了，另一人接着使用复印机时，该复印机上可能就还有先前那名职员留下来的资料。抢占式线程模型要求线程正确共享资源，协作式模型却要求线程共享执行时间。由于 JVM 规范并没有特别规定线程模型，Java 开发员必须编写可在两种模型上正确运行的程序。在了解线程以及线程间通讯的一些方面之后，我们可以看到如何为这两种模型设计程序。 线程和 Java 语言为了使用 Java 语言创建线程，你可以生成一个 Thread 类（或其子类）的对象，并给这个对象发送 start() 消息。（程序可以向任何一个派生自 Runnable 接口的类对象发送 start() 消息。）每个线程动作的定义包含在该线程对象的 run() 方法中。run 方法就相当于传统程序中的 main() 方法；线程会持续运行，直到 run() 返回为止，此时该线程便死了。 上锁大多数应用程序要求线程互相通信来同步它们的动作。在 Java 程序中最简单实现同步的方法就是上锁

。为了防止同时访问共享资源，线程在使用资源的前后可以给该资源上锁和开锁。假想给复印机上锁，任一时刻只有一个职员拥有钥匙。若没有钥匙就不能使用复印机。给共享变量上锁就使得 Java 线程能够快速方便地通信和同步。某个线程若给一个对象上了锁，就可以知道没有其他线程能够访问该对象。即使在抢占式模型中，其他线程也不能够访问此对象，直到上锁的线程被唤醒、完成工作并开锁。那些试图访问一个上锁对象的线程通常会进入睡眠状态，直到上锁的线程开锁。一旦锁被打开，这些睡眠进程就会被唤醒并移到准备就绪队列中。 在 Java 编程中，所有的对象都有锁。线程可以使用 synchronized 关键字来获得锁。在任一时刻对于给定的类的实例，方法或同步的代码块只能被一个线程执行。这是因为代码在执行之前要求获得对象的锁。继续我们关于复印机的比喻，为了避免复印冲突，我们可以简单地对复印资源实行同步。如同下列的代码例子，任一时刻只允许一位职员使用复印资源。通过使用方法（在 Copier 对象中）来修改复印机状态。这个方法就是同步方法。只有一个线程能够执行一个 Copier 对象中同步代码，因此那些需要使用 Copier 对象的职员就必须排队等候。 class CopyMachine {public synchronized void makeCopies(Document d, int nCopies) {// only one thread executes this at a time}public void loadPaper() {// multiple threads could access this at once!synchronized(this) {// only one thread accesses this at a time// feel free to use shared resources, overwrite members, etc.}}}Fine-grain 锁在对象级使用锁通常是一种比较粗糙的方法。为什么要将整个对象都上锁，而不允许其他线程短暂地使用对象中其他同步方法来访问

共享资源？如果一个对象拥有多个资源，就不需要只为了让一个线程使用其中一部分资源，就将所有线程都锁在外面。由于每个对象都有锁，可以如下所示使用虚拟对象来上锁：class FineGrainLock {MyMemberClass x, y.Object xlock = new Object(), ylock = new Object().public void foo() {synchronized(xlock) {// access x here}// do something here - but dont use shared resourcessynchronized(ylock) {// access y here}}public void bar() {synchronized(this) {// access both x and y here}// do something here - but dont use shared resources}}若为了在方法级上同步，不能将整个方法声明为 synchronized 关键字。它们使用的是成员锁，而不是 synchronized 方法能够获得的对象级锁。 