

### 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。

进程和线程的主要差别在于它们是不同的操作系统资源管理方式。进程有独立的地址空间，一个进程崩溃后，在保护模式下不会对其余进程产生影响，而线程只是一个进程中的不同执行路径。线程有自己的堆栈和局部变量，但线程之间没有单独的地址空间，一个线程死掉就等于整个进程死掉，所以多进程的程序要比多线程的程序健壮，但在进程切换时，耗费资源较大，效率要差一些。但对于一些要求同时进行并且又要共享某些变量的并发操作，只能用线程，不能用进程。

### 1) 继承 Thread 类实现多线程

### 2) 实现 Runnable 接口方式实现多线程

### 3) 使用 ExecutorService、Callable、Future 实现有返回结果的多线程

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。

`stop` 终止，不推荐。

**NEW:** 毫无疑问表示的是刚创建的线程，还没有开始启动。

**RUNNABLE:** 表示线程已经触发 `start()` 方式调用，线程正式启动，线程处于运行中状态。

**BLOCKED:** 表示线程阻塞，等待获取锁，如碰到 `synchronized`、`lock` 等关键字等

占用临界区的情况，一旦获取到锁就进行 `RUNNABLE` 状态继续运行。

**WAITING:** 表示线程处于无限制等待状态，等待一个特殊的事件来重新唤醒，如通过 `wait()` 方法进行等待的线程等待一个 `notify()` 或者 `notifyAll()` 方法，通过 `join()` 方法进行等待的线程等待目标线程运行结束而唤醒，一旦通过相关事件唤醒线程，线程就进入了 `RUNNABLE` 状态继续运行。

**TIMED\_WAITING:** 表示线程进入了一个有时限的等待，如 `sleep(3000)`，等待 3 秒后线程重新进行 `RUNNABLE` 状态继续运行。

**TERMINATED:** 表示线程执行完毕后，进行终止状态。需要注意的是，一旦线程通过 `start` 方法启动后就再也不能回到初始 `NEW` 状态，线程终止后也不能再回到 `RUNNABLE` 状态

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

`Synchronized` 关键字，`Lock` 锁实现，分布式锁等。

死锁就是两个线程相互等待对方释放对象锁。

`wait/notify`

实现 `Callable` 接口。

一个非常重要的问题，是每个学习、应用多线程的 Java 程序员都必须掌握的。理解 `volatile` 关键字的作用的前提是要理解 Java 内存模型，这里就不讲 Java 内存模型了，可以参见第31点，`volatile` 关键字的作用主要有两个：

1) 多线程主要围绕可见性和原子性两个特性而展开，使用 `volatile` 关键字修饰的变量，保证了其在多线程之间的可见性，即每次读取到 `volatile` 变量，一定是最新的数据

2) 代码底层执行不像我们看到的高级语言----Java 程序这么简单，它的执行是 Java 代码-->字节码-->根据字节码执行对应的 C/C++代码-->C/C++代码被编译成汇编语言-->和硬件电路交互，现实中，为了获取更好的性能 JVM 可能会对指令进行重排序，多线程下可能会出现一些意想不到的问题。使用 `volatile` 则会对禁止语义重排序，当然这也一定程度上降低了代码执行效率从实践角度而言，`volatile` 的一个重要作用就是和 `CAS` 结合，保证了原子性，详细的可以参见 `java.util.concurrent.atomic` 包下的类，比如 `AtomicInteger`。

用 `join` 方法。

用 `Semaphore`。

我们知道不用线程池的话，每个线程都要通过 `new Thread(xxRunnable).start()` 的方式来创建并运行一个线程，线程少的话这不是问题，而真实环境可能会开启多个

---

线程让系统和程序达到最佳效率，当线程数达到一定数量就会耗尽系统的 CPU 和内存资源，也会造成 GC 频繁收集和停顿，因为每次创建和销毁一个线程都是要消耗系统资源的，如果为每个任务都创建线程这无疑是一个很大的性能瓶颈。所以，线程池中的线程复用极大节省了系统资源，当线程一段时间不再有任务处理时它也会自动销毁，而不会长驻内存。

什么是线程池？

很简单，简单看名字就知道是装有线程的池子，我们可以把要执行的多线程交给线程池来处理，和连接池的概念一样，通过维护一定数量的线程池来达到多个线程的复用。

线程池的好处

我们知道不用线程池的话，每个线程都要通过 `new Thread(xxRunnable).start()` 的方式来创建并运行一个线程，线程少的话这不会有问题，而真实环境可能会开启多个线程让系统和程序达到最佳效率，当线程数达到一定数量就会耗尽系统的 CPU 和内存资源，也会造成 GC 频繁收集和停顿，因为每次创建和销毁一个线程都是要消耗系统资源的，如果为每个任务都创建线程这无疑是一个很大的性能瓶颈。所以，线程池中的线程复用极大节省了系统资源，当线程一段时间不再有任务处理时它也会自动销毁，而不会长驻内存。

线程池核心类

在 `java.util.concurrent` 包中我们能找到线程池的定义，其中 `ThreadPoolExecutor` 是我们线程池核心类，首先看看线程池类的主要参数有哪些。

如何提交线程

如可以先随便定义一个固定大小的线程池 `ExecutorService es = Executors.newFixedThreadPool(3)`；提交一个线程 `es.submit(xxRunnable)`；  
`es.execute(xxRunnable)`；

`submit` 和 `execute` 分别有什么区别呢？

`execute` 没有返回值，如果不需要知道线程的结果就使用 `execute` 方法，性能会好很多。

`submit` 返回一个 `Future` 对象，如果想知道线程结果就使用 `submit` 提交，而且它能在主线程中通过 `Future` 的 `get` 方法捕获线程中的异常。

如何关闭线程池 `es.shutdown()`；

不再接受新的任务，之前提交的任务等执行结束再关闭线程池。

`es.shutdownNow()`；

不再接受新的任务，试图停止池中的任务再关闭线程池，返回所有未处理的线程 `list` 列表。

`execute` 没有返回值，如果不需要知道线程的结果就使用 `execute` 方法，性能会好很多。

`submit` 返回一个 `Future` 对象，如果想知道线程结果就使用 `submit` 提交，而且它能在主线程中通过 `Future` 的 `get` 方法捕获线程中的异常。

两个看上去有点像的类，都在 `java.util.concurrent` 下，都可以用来表示代码运行到某个点上，二者的区别在于：

1. `CyclicBarrier` 的某个线程运行到某个点上之后，该线程即停止运行，直到所有的线程都到达了这一点，所有线程才重新运行；`CountDownLatch` 则不是，某线程运行到某个点上之后，只是给某个数值-1而已，该线程继续运行

- 
- 2.CyclicBarrier 只能唤起一个任务，CountDownLatch 可以唤起多个任务  
3.CyclicBarrier 可重用，CountDownLatch 不可重用，计数值为 0 该 CountDownLatch就不可再用了

死锁、活锁、饥饿是关于多线程是否活跃出现的运行阻塞障碍问题，如果线程出现了这三种情况，即线程不再活跃，不能再正常地执行下去了。

### 死锁

死锁是多线程中最差的一种情况，多个线程相互占用对方的资源的锁，而又相互等待对方释放锁，此时若无外力干预，这些线程则一直处理阻塞的假死状态，形成死锁。举个例子，A 同学抢了 B 同学的钢笔，B 同学抢了 A 同学的书，两个人都相互占用对方的东西，都在让对方先还给自己自己再还，这样一直争执下去等待对方还而又得不到解决，老师知道此事后就让他们相互还给对方，这样在外力的干预下他们才解决，当然这只是个例子没有老师他们也能很好解决，计算机不像人如果发现这种情况没有外力干预还是会一直阻塞下去的。

### 活锁

活锁这个概念大家应该很少有人听说或理解它的概念，而在多线程中这确实存在。活锁恰恰与死锁相反，死锁是大家拿不到资源都占用着对方的资源，而活锁是拿到资源却又相互释放不执行。当多线程中出现了相互谦让，都主动将资源释放给别的线程使用，这样这个资源在多个线程之间跳动而又得不到执行，这就是活锁。

### 饥饿

我们知道多线程执行中有线程优先级这个东西，优先级高的线程能够插队并优先执行，这样如果优先级高的线程一直抢占优先级低线程的资源，导致低优先级线程无法得到执行，这就是饥饿。当然还有一种饥饿的情况，一个线程一直占着一个资源不放而导致其他线程得不到执行，与死锁不同的是饥饿在以后一段时间内还是能够得到执行的，如那个占用资源的线程结束了并释放了资源。

### 无锁

无锁，即没有对资源进行锁定，即所有的线程都能访问并修改同一个资源，但同时只有一个线程能修改成功。无锁典型的特点就是一个修改操作在一个循环内进行，线程会不断的尝试修改共享资源，如果没有冲突就修改成功并退出否则就会继续下一次循环尝试。所以，如果有多个线程修改同一个值必定会有一个线程能修改成功，而其他修改失败的线程会不断重试直到修改成功。之前的文章我介绍过 JDK 的 CAS 原理及应用即是无锁的实现。

可以看出，无锁是一种非常良好的设计，它不会出现线程出现的跳跃性问题，锁使用不当肯定会出现系统性能问题，虽然无锁无法全面代替有锁，但无锁在某些场合下是非常高效的。

原子性、可见性、有序性是多线程编程中最重要的几个知识点，由于多线程情况复杂，如何让每个线程能看到正确的结果，这是非常重要的。

### 原子性

原子性是指一个线程的操作是不能被其他线程打断，同一时间只有一个线程对一个变量进行操作。在多线程情况下，每个线程的执行结果不受其他线程的干扰，比如说多个线程同时对同一个共享成员变量  $n++100$  次，如果  $n$  初始值为 0， $n$  最后的值应该是 100，所以说它们是互不干扰的，这就是传说中的原子性。但  $n++$  并不是原子性的操作，要使用 AtomicInteger 保证原子性。

### 可见性

可见性是指某个线程修改了某一个共享变量的值，而其他线程是否可以看见该共享变量修改后的值。在单线程中肯定不会有这种问题，单线程读到的肯定都是最新的



---

值，而在多线程编程中就不一定了。每个线程都有自己的工作内存，线程先把共享变量的值从主内存读到工作内存，形成一个副本，当计算完后再把副本的值刷回主内存，从读取到最后刷回主内存这是一个过程，当还没刷回主内存的时候这时候对其他线程是不可见的，所以其他线程从主内存读到的值是修改之前的旧值。像 CPU 的缓存优化、硬件优化、指令重排及对 JVM 编译器的优化，都会出现可见性的问题。

### 有序性

我们都知道程序是按代码顺序执行的，对于单线程来说确实是如此，但在多线程情况下就不是如此了。为了优化程序执行和提高 CPU 的处理性能，JVM 和操作系统都会对指令进行重排，也就是说前面的代码并不一定都会后面的代码前面执行，即后面的代码可能会插到前面的代码之前执行，只要不影响当前线程的执行结果。所以，指令重排只会保证当前线程执行结果一致，但指令重排后势必会影响多线程的执行结果。虽然重排序优化了性能，但也是会遵守一些规则的，并不能随便乱排序，只是重排序会影响多线程执行的结果。

什么是守护线程？与守护线程相对应的就是用户线程，守护线程就是守护用户线程，当用户线程全部执行完结束之后，守护线程才会跟着结束。也就是守护线程必须伴随着用户线程，如果一个应用内只存在一个守护线程，没有用户线程，守护线程自然会退出。

如果异常没有被捕获该线程将会停止执行。Thread.UncaughtExceptionHandler 是用于处理未捕获异常造成线程突然中断情况的一个内嵌接口。当一个未捕获异常将造成线程中断的时候 JVM 会使用 Thread.getUncaughtExceptionHandler() 来查询线程的 UncaughtExceptionHandler 并将线程和异常作为参数传递给 handler 的 uncaughtException() 方法进行处理。

Yield 方法可以暂停当前正在执行的线程对象，让其它有相同优先级的线程执行。它是一个静态方法而且只保证当前线程放弃 CPU 占用而不能保证使其它线程一定能占用 CPU，执行 yield() 的线程有可能在进入到暂停状态后马上又被执行。

所谓重入锁，指的是以线程为单位，当一个线程获取对象锁之后，这个线程可以再次获取本对象上的锁，而其他的线程是不可以的。

锁类、锁方法、锁代码块。

大任务自动分散小任务，并发执行，合并小任务结果。

线程过多会造成栈溢出，也有可能造成堆异常。

Java 中平时用的最多的 Map 集合就是 HashMap 了，它是线程不安全的。

看下面两个场景：

1、当用在方法内的局部变量时，局部变量属于当前线程级别的变量，其他线程访问不了，所以这时也不存在线程安全不安全的问题了。

2、当用在单例对象成员变量的时候呢？这时候多个线程过来访问的就是同一个 HashMap 了，对同个 HashMap 操作这时候就存在线程安全的问题了。

CAS，全称为 Compare and Swap，即比较-替换。假设有三个操作数：内存值 V、旧的预期值 A、要修改的值 B，当且仅当预期值 A 和内存值 V 相同时，才会将内存值修改为 B 并返回 true，否则什么都不做并返回 false。当然 CAS 一定要 volatile 变量配合，这样才能保证每次拿到的变量是主内存中最新的那个值，否则旧的预期值 A 对某条线程来说，永远是一个不会变的值 A，只要某次 CAS 操作失败，永远都不可能成功。

java.util.concurrent.atomic 包下面的 Atom\*\*\*\*类都有 CAS 算法的应用。

java.lang.Thread#holdsLock 方法

jstack

- 1、尽量缩小同步的范围，增加系统吞吐量。
- 2、分布式同步锁无意义，要使用分布式锁。
- 3、防止死锁，注意加锁顺序。

要在同步块中使用。

如果任务拆解的很深，系统内的线程数量堆积，导致系统性能严重下降；  
如果函数的调用栈很深，会导致栈内存溢出；

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

synchronized 和 volatile

ReentrantLock、ReadWriteLock

ThreadLocal 的作用是提供线程内的局部变量，这种变量在线程的生命周期内起作用，减少同一个线程内多个函数或者组件之间一些公共变量的传递的复杂度。用来解决数据库连接、Session 管理等。

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。

FutureTask 表示一个异步运算的任务，FutureTask 里面可以传入一个 Callable 的具

体实现类，可以对这个异步运算的任务的结果进行等待获取、判断是否已经完成、取消任务等操作。

如果线程是因为调用了 `wait()`、`sleep()` 或者 `join()` 方法而导致的阻塞，可以中断线程，并且通过抛出 `InterruptedException` 来唤醒它；如果线程遇到了 IO 阻塞，无能为力，因为 IO 是操作系统实现的，Java 代码并没有办法直接接触到操作系统。

不可变对象保证了对象的内存可见性，对不可变对象的读取不需要进行额外的同步手段，提升了代码执行效率。

多线程的上下文切换是指 CPU 控制权由一个已经正在运行的线程切换到另外一个就绪并等待获取 CPU 执行权的线程的过程。

？

抢占式。一个线程用完 CPU 之后，操作系统会根据线程优先级、线程饥饿情况等数据算出一个总的优先级并分配下一个时间片给某个线程执行。

由于 Java 采用抢占式的线程调度算法，因此可能会出现某条线程常常获取到 CPU 控制权的情况，为了让某些优先级比较低的线程也能获取到 CPU 控制权，可以使用 `Thread.sleep(0)` 手动触发一次操作系统分配时间片的操作，这也是平衡 CPU 控制权的一种操作。

我们知道的 JVM 内存区域有：堆和栈，这是一种泛的分法，也是按运行时区域的一种分法，堆是所有线程共享的一块区域，而栈是线程隔离的，每个线程互不共享。线程不共享区域每个线程的数据区域包括程序计数器、虚拟机栈和本地方法栈，它们都是在新线程创建时才创建的。

**程序计数器（Program Counter Register）**

程序计数器区域一块内存较小的区域，它用于存储线程的每个执行指令，每个线程都有自己的程序计数器，此区域不会有内存溢出的情况。

**虚拟机栈（VM Stack）**

虚拟机栈描述的是 Java 方法执行的内存模型，每个方法被执行的时候都会同时创建一个栈帧（Stack Frame）用于存储局部变量表、操作数栈、动态链接、方法出口等信息。每一个方法被调用直至执行完成的过程就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。

**本地方法栈（Native Method Stack）**

本地方法栈用于支持本地方法（native 标识的方法，即非 Java 语言实现的方法）。虚拟机栈和本地方法栈，当线程请求分配的栈容量超过 JVM 允许的最大容量时抛出 `StackOverflowError` 异常。

**线程共享区域**

线程共享区域包含：堆和方法区。

**堆（Heap）**

堆是最常处理的区域，它存储在 JVM 启动时创建的数组和对象，JVM 垃圾收集也主要是在堆上面工作。

如果实际所需的堆超过了自动内存管理系统能提供的最大容量时抛出

OutOfMemoryError 异常。

方法区 (Method Area)

方法区是可供各条线程共享的运行时内存区域。存储了每一个类的结构信息，例如运行时常量池 (Runtime Constant Pool)、字段和方法数据、构造函数和普通方法的字节码内容、还包括一些在类、实例、接口初始化时用到的特殊方法。当创建类和接口时，如果构造运行时常量池所需的内存空间超过了方法区所能提供的最大内存空间后就会抛出 OutOfMemoryError

运行时常量池 (Runtime Constant Pool)

运行时常量池是方法区的一部分，每一个运行时常量池都分配在 JVM 的方法区中，在类和接口被加载到 JVM 后，对应的运行时常量池就被创建。运行时常量池是每一个类或接口的常量池 (Constant\_Pool) 的运行时表现形式，它包括了若干种常量：编译器可知的数值字面量到必须运行期解析后才能获得的方法或字段的引用。如果方法区的内存空间不能满足内存分配请求，那 Java 虚拟机将抛出一个 OutOfMemoryError 异常。栈包含 Frames，当调用方法时，Frame 被推送到堆栈。一个 Frame 包含局部变量数组、操作数栈、常量池引用。

乐观锁：就像它的名字一样，对于并发间操作产生的线程安全问题持乐观状态，乐观锁认为竞争不总是会发生，因此它不需要持有锁，将比较-替换这两个动作作为一个原子操作尝试去修改内存中的变量，如果失败则表示发生冲突，那么就应该有相应的重试逻辑。

悲观锁：还是像它的名字一样，对于并发间操作产生的线程安全问题持悲观状态，悲观锁认为竞争总是会发生，因此每次对某资源进行操作时，都会持有一个独占的锁，就像 synchronized，不管三七二十一，直接上了锁就操作资源了。

同一时间只能有一条线程执行固定类的同步方法，但是对于类的非同步方法，可以多条线程同时访问。所以，这样就有问题了，可能线程 A 在执行 Hashtable 的 put 方法添加数据，线程 B 则可以正常调用 size() 方法读取 Hashtable 中当前元素的个数，那读取到的值可能不是最新的，可能线程 A 添加了完了数据，但是没有对 size++，线程 B 就已经读取 size 了，那么对于线程 B 来说读取到的 size 一定是不准确的。而给 size() 方法加了同步之后，意味着线程 B 调用 size() 方法只有在线程 A 调用 put 方法完毕之后才可以调用，这样就保证了线程安全性 CPU 执行代码，执行的不是 Java 代码，这点很关键，一定得记住。Java 代码最终是被翻译成机器码执行的，机器码才是真正可以和硬件电路交互的代码。即使你看到 Java 代码只有一行，甚至你看到 Java 代码编译之后生成的字节码也只有一行，也不意味着对于底层来说这句语句的操作只有一个。一句 "return count" 假设被翻译成了三句汇编语句执行，一句汇编语句和其机器码做对应，完全可能执行完第一句，线程就切换了。

同步块，这意味着同步块之外的代码是异步执行的，这比同步整个方法更提升代码的效率。

请知道一条原则：同步的范围越小越好。

自旋锁是采用让当前线程不停地在循环体内执行实现的，当循环的条件被其他线程改变时才能进入临界区。

Java 不支持类的多重继承，但允许你实现多个接口。所以如果你要继承其他类，也



---

为了减少类之间的耦合性，`Runnable` 会更好。

`notify()`方法不能唤醒某个具体的线程，所以只有一个线程在等待的时候它才有用武之地。

而 `notifyAll()`唤醒所有线程并允许他们争夺锁确保了至少有一个线程能继续运行。

这是个设计相关的问题，它考察的是面试者对现有系统和一些普遍存在但看起来不合理的事物的看法。回答这些问题的时候，你要说明为什么把这些方法放在 `Object` 类里是有意义的，还有不把它放在 `Thread` 类里的原因。一个很明显的原因是 `JAVA` 提供的锁是对象级的而不是线程级的，每个对象都有锁，通过线程获得。如果线程需要等待某些锁那么调用对象中的`wait()`方法就有意义了。如果 `wait()`方法定义在 `Thread` 类中，线程正在等待的是哪个锁就不明显了。简单的说，由于 `wait`，`notify` 和 `notifyAll` 都是锁级别的操作，所以把他们定义在 `Object` 类中因为锁属于对象。

主要是因为 `Java API` 强制要求这样做，如果你不这么做，你的代码会抛出 `IllegalMonitorStateException` 异常。还有一个原因是为了避免 `wait` 和 `notify` 之间产生竞态条件。

处于等待状态的线程可能会收到错误警报和伪唤醒，如果不在循环中检查等待条件，程序就会在没有满足结束条件的情况下退出。因此，当一个等待线程醒来时，不能认为它原来的等待状态仍然是有效的，在 `notify()`方法调用之后和等待线程醒来之前这段时间它可能会改变。这就是在循环中使用 `wait()`方法效果更好的原因，你可以在 `Eclipse` 中创建模板调用 `wait`和 `notify` 试一试。

每个线程都有自己的栈内存，用于存储本地变量，方法参数和栈调用，一个线程中存储的变量对其它线程是不可见的。而堆是所有线程共享的一片公用内存区域。对象都在堆里创建，为了提升效率线程会从堆中弄一个缓存到自己的栈，如果多个线程使用该变量就可能引发问题，这时 `volatile` 变量就可以发挥作用了，它要求线程从主存中读取变量的值。

对于不同的操作系统，有多种方法来获得 `Java` 进程的线程堆栈。当你获取线程堆栈时，`JVM`会把所有线程的状态存到日志文件或者输出到控制台。在 `Windows` 你可以使用 `Ctrl + Break` 组合键来获取线程堆栈，`Linux` 下用 `kill -3` 命令。你也可以用 `jstack` 这个工具来获取，它对线程 `id` 进行操作，你可以用 `jps` 这个工具找到 `id`。

单例模式即一个 `JVM` 内存中只存在一个类的对象实例分类


1、懒汉式

类加载的时候就创建实例

2、饿汉式

使用的时候才创建实例

阻塞式方法是指程序会一直等待该方法完成期间不做其他事情，`ServerSocket` 的



---

`accept()`方法就是一直等待客户端连接。这里的阻塞是指调用结果返回之前，当前线程会被挂起，直到得到结果之后才会返回。此外，还有异步和非阻塞式方法在任务完成前就返回。

当线程数小于最大线程池数 `maximumPoolSize` 时就会创建新线程来处理，而线程数大于等于最大线程池数 `maximumPoolSize` 时就会执行拒绝策略。