拼多多电商部二面整理

1、并发编程三要素？

（1）原子性
原子性指的是一个或者多个操作，要么全部执行并且在执行的过程中不被其他操作打断，要么就全部都不执行。

（2）可见性
可见性指多个线程操作一个共享变量时，其中一个线程对变量进行修改后，其他线程可以立即看到修改的结果。

（3）有序性
有序性，即程序的执行顺序按照代码的先后顺序来执行。

2、实现可见性的方法有哪些？

synchronized 或者 Lock：保证同一个时刻只有一个线程获取锁执行代码，锁释放之前把最新的值刷新到主内存，实现可见性。

3、多线程的价值？

（1）发挥多核 CPU 的优势
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，采用多线程的方式去同时完成几件事情而不互相干扰。

（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。

4、创建线程的有哪些方式？

（1）继承 Thread 类创建线程类

（2）通过 Runnable 接口创建线程类

（3）通过 Callable 和 Future 创建线程

（4）通过线程池创建

5、创建线程的三种方式的对比？

（1）采用实现 Runnable、Callable 接口的方式创建多线程。
优势是：

线程类只是实现了 Runnable 接口或 Callable 接口，还可以继承其他类。在这种方式下，多个线程可以共享同一个 target 对象，所以非常适合多个相同线程来处理同一份资源的情况，从而可以将 CPU、代码和数据分开，形成清晰的模型，较好地体现了面向对象的思想。

劣势是：

编程稍微复杂，如果要访问当前线程，则必须使用 Thread.currentThread()方法。

（2）使用继承 Thread 类的方式创建多线程
优势是：

编写简单，如果需要访问当前线程，则无需使用 Thread.currentThread()方法，直接使用 this 即可获得当前线程。

劣势是：

线程类已经继承了 Thread 类，所以不能再继承其他父类。

（3）Runnable 和 Callable 的区别
1、Callable 规定（重写）的方法是 call()，Runnable 规定（重写）的方法是 run()。

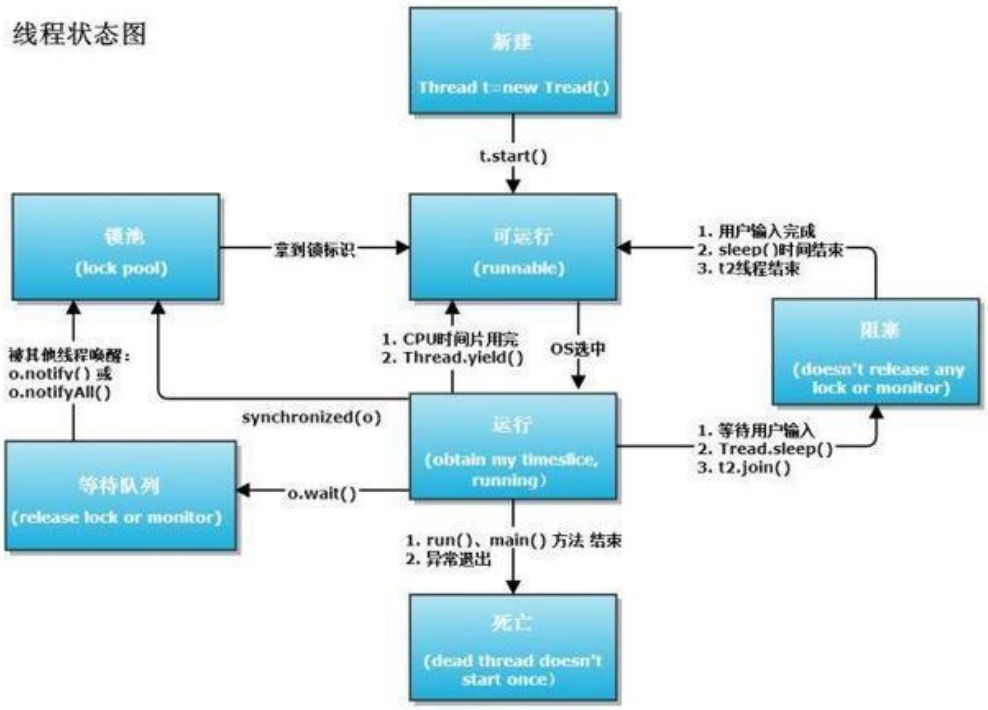
2、Callable 的任务执行后可返回值，而 Runnable 的任务是不能返回值的。

3、Call 方法可以抛出异常，run 方法不可以。

4、运行 Callable 任务可以拿到一个 Future 对象，表示异步计算的结果。它提供了检查计算是否完成的方法，以等待计算的完成，并检索计算的结果。通过 Future 对象可以了解任务执行情况，可取消任务的执行，还可获取执行结果。

6、线程的状态流转图

线程的生命周期及五种基本状态：



7、Java 线程具有五中基本状态

（1）新建状态（New）：
当线程对象对创建后，即进入了新建状态，如：Thread t= new MyThread()；

（2）就绪状态（Runnable）：
当调用线程对象的 start()方法（t.start();），线程即进入就绪状态。处于就绪状态的线程，只是说明此线程已经做好了准备，随时等待 CPU 调度执行，并不是说执行了 t.start()此线程立即就会执行；

（3）运行状态（Running）：
当 CPU 开始调度处于就绪状态的线程时，此时线程才得以真正执行，即进入到运行状态。注：就 绪状态是进入到运行状态的唯一入口，也就是说，线程要想进入运行状态执行，首先必须处于就绪状态中；

（4）阻塞状态（Blocked）：
处于运行状态中的线程由于某种原因，暂时放弃对 CPU 的使用权，停止执行，此时进入阻

塞状态，直到其进入到就绪状态，才 有机会再次被 CPU 调用以进入到运行状态。

根据阻塞产生的原因不同，阻塞状态又可以分为三种：

1）等待阻塞：运行状态中的线程执行 wait()方法，使本线程进入到等待阻塞状态；

2）同步阻塞：线程在获取 synchronized 同步锁失败(因为锁被其它线程所占用)，

它会进入同步阻塞状态；

3）其他阻塞：通过调用线程的 sleep()或 join()或发出了 I/O 请求时，线程会进入到阻塞状态。当 sleep()状态超时、join()等待线程终止或者超时、或者 I/O 处理完毕时，线程重新转入就绪状态。

（5）死亡状态（Dead）：
线程执行完了或者因异常退出了 run()方法，该线程结束生命周期。

8、什么是线程池？有哪几种创建方式？

线程池就是提前创建若干个线程，如果有任务需要处理，线程池里的线程就会处理任务，处理完之后线程并不会被销毁，而是等待下一个任务。由于创建和销毁线程都是消耗系统资源的，所以当你想要频繁的创建和销毁线程的时候就可以考虑使用线程池来提升系统的性能。

java 提供了一个 java.util.concurrent.Executor 接口的实现用于创建线程池。

9、四种线程池的创建：

（1）newCachedThreadPool 创建一个可缓存线程池

（2）newFixedThreadPool 创建一个定长线程池，可控制线程最大并发数。

（3）newScheduledThreadPool 创建一个定长线程池，支持定时及周期性任务执行。

（4）newSingleThreadExecutor 创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务。

10、线程池的优点？

（1）重用存在的线程，减少对象创建销毁的开销。

（2）可有效的控制最大并发线程数，提高系统资源的使用率，同时避免过多资源竞争，避

免堵塞。

（3）提供定时执行、定期执行、单线程、并发数控制等功能。

11、常用的并发工具类有哪些？

（1）CountDownLatch

（2）CyclicBarrier

（3）Semaphore

（4）Exchanger

12、CyclicBarrier 和 CountDownLatch 的区别

（1）CountDownLatch 简单的说就是一个线程等待，直到他所等待的其他线程都执行完成并且调用 countDown()方法发出通知后，当前线程才可以继续执行。

（2）cyclicBarrier 是所有线程都进行等待，直到所有线程都准备好进入 await()方法之后，所有线程同时开始执行！

（3）CountDownLatch 的计数器只能使用一次。而 CyclicBarrier 的计数器可以使用 reset()方法重置。所以 CyclicBarrier 能处理更为复杂的业务场景，比如如果计算发生错误，可以重置计数器，并让线程们重新执行一次。

（4）CyclicBarrier 还提供其他有用的方法，比如 getNumberWaiting 方法可以获得 CyclicBarrier 阻塞的线程数量。isBroken 方法用来知道阻塞的线程是否被中断。如果被中断返回 true，否则返回 false。

13、synchronized 的作用？

在 Java 中，synchronized 关键字是用来控制线程同步的，就是在多线程的环境下，控制 synchronized 代码段不被多个线程同时执行。synchronized 既可以加在一段代码上，也可以加在方法上。

14、volatile 关键字的作用

对于可见性，Java 提供了 volatile 关键字来保证可见性。当一个共享变量被 volatile 修饰时，它会保证修改的值会立即被更新到主存，当有其他线程需要读取时，它会去内存中读取

新值。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。

15、什么是 CAS

CAS 是 compare and swap 的缩写，即我们所说的比较交换。

cas 是一种基于锁的操作，而且是乐观锁。在 java 中锁分为乐观锁和悲观锁。悲观锁是将资源锁住，等一个之前获得锁的线程释放锁之后，下一个线程才可以访问。而乐观锁采取了一种宽泛的态度，通过某种方式不加锁来处理资源，比如通过给记录加 version 来获取数据，性能较悲观锁有很大的提高。

CAS 操作包含三个操作数 —— 内存位置（V）、预期原值（A）和新值(B)。如果内存地址里面的值和 A 的值是一样的，那么就将内存里面的值更新成 B。CAS 是通过无限循环来获取数据的，若果在第一轮循环中，a 线程获取地址里面的值被 b 线程修改了，那么 a 线程需要自旋，到下次循环才有可能机会执行。

java.util.concurrent.atomic 包下的类大多是使用 CAS 操作来实现的(AtomicInteger,AtomicBoolean,AtomicLong)。

16、CAS 的问题

（1）CAS 容易造成 ABA 问题
一个线程 a 将数值改成了 b，接着又改成了 a，此时 CAS 认为是没有变化，其实是已经变化过了，而这个问题的解决方案可以使用版本号标识，每操作一次 version 加 1。在 java5 中，已经提供了 AtomicStampedReference 来解决问题。

（2）不能保证代码块的原子性
CAS 机制所保证的知识一个变量的原子性操作，而不能保证整个代码块的原子性。比如需要保证 3 个变量共同进行原子性的更新，就不得不使用 synchronized 了。

（3）CAS 造成 CPU 利用率增加
之前说过了 CAS 里面是一个循环判断的过程，如果线程一直没有获取到状态，cpu 资源会一直被占用。

17、什么是 Future？

在并发编程中，我们经常用到非阻塞的模型，在之前的多线程的三种实现中，不管是继承 thread 类还是实现 runnable 接口，都无法保证获取到之前的执行结果。通过实现 Callback 接口，并用 Future 可以来接收多线程的执行结果。

Future 表示一个可能还没有完成的异步任务的结果，针对这个结果可以添加 Callback 以便在任务执行成功或失败后作出相应的操作。

18、什么是 AQS

AQS 是 AbustactQueuedSynchronizer 的简称，它是一个 Java 提高的底层同步工具类，用一个 int 类型的变量表示同步状态，并提供了一系列的 CAS 操作来管理这个同步状态。

AQS 是一个用来构建锁和同步器的框架，使用 AQS 能简单且高效地构造出应用广泛的大量的同步器，比如我们提到的 ReentrantLock，Semaphore，其他的诸如 ReentrantReadWriteLock，SynchronousQueue，FutureTask 等等皆是基于 AQS 的。

19、AQS 支持两种同步方式：

（1）独占式
（2）共享式
这样方便使用者实现不同类型的同步组件，独占式如 ReentrantLock，共享式如 Semaphore，CountDownLatch，组合式的如 ReentrantReadWriteLock。总之，AQS 为使用提供了底层支撑，如何组装实现，使用者可以自由发挥。

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。

21、FutureTask 是什么

这个其实前面有提到过，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。当然，由于 FutureTask 也是 Runnable 接口的实现类，所以 FutureTask 也可以放入线程池中。

22、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。

23、什么是乐观锁和悲观锁

（1）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。

（2）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。

24、线程 B 怎么知道线程 A 修改了变量

（1）volatile 修饰变量

（2）synchronized 修饰修改变量的方法

（3）wait/notify

（4）while 轮询

25、synchronized、volatile、CAS 比较

（1）synchronized 是悲观锁，属于抢占式，会引起其他线程阻塞。

（2）volatile 提供多线程共享变量可见性和禁止指令重排序优化。

（3）CAS 是基于冲突检测的乐观锁（非阻塞）

26、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

27、ThreadLocal 是什么？有什么用？

ThreadLocal 是一个本地线程副本变量工具类。主要用于将私有线程和该线程存放的副本对象做一个映射，各个线程之间的变量互不干扰，在高并发场景下，可以实现无状态的调用，特别适用于各个线程依赖不通的变量值完成操作的场景。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。

28、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

29、多线程同步有哪几种方法？

Synchronized 关键字，Lock 锁实现，分布式锁等。

30、线程的调度策略

线程调度器选择优先级最高的线程运行，但是，如果发生以下情况，就会终止线程的运行：

（1）线程体中调用了 yield 方法让出了对 cpu 的占用权利

（2）线程体中调用了 sleep 方法使线程进入睡眠状态

（3）线程由于 IO 操作受到阻塞

（4）另外一个更高优先级线程出现

（5）在支持时间片的系统中，该线程的时间片用完

31、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？

32、Linux 环境下如何查找哪个线程使用 CPU 最长

（1）获取项目的 pid，jps 或者 ps -ef | grep java

（2）top -H -p pid，顺序不能改变

33、Java 死锁以及如何避免？

Java 中的死锁是一种编程情况，其中两个或多个线程被永久阻塞，Java 死锁情况出现至少两个线程和两个或更多资源。

Java 发生死锁的根本原因是：在申请锁时发生了交叉闭环申请。

34、死锁的原因

（1）是多个线程涉及到多个锁，这些锁存在着交叉，所以可能会导致了一个锁依赖的闭环。例如：线程在获得了锁 A 并且没有释放的情况下去申请锁 B，这时，另一个线程已经获得了锁 B，在释放锁 B 之前又要先获得锁 A，因此闭环发生，陷入死锁循环。

（2）默认的锁申请操作是阻塞的。
所以要避免死锁，就要在一遇到多个对象锁交叉的情况，就要仔细审查这几个对象的类中的所有方法，是否存在着导致锁依赖的环路的可能性。总之是尽量避免在一个同步方法中调用其它对象的延时方法和同步方法。

35、怎么唤醒一个阻塞的线程

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。

36、不可变对象对多线程有什么帮助

前面有提到过的一个问题，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。

37、什么是多线程的上下文切换

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。

38、如果你提交任务时，线程池队列已满，这时会发生什么

这里区分一下：

（1）如果使用的是无界队列 LinkedBlockingQueue，也就是无界队列的话，没关系，继续添加任务到阻塞队列中等待执行，因为 LinkedBlockingQueue 可以近乎认为是一个无穷大的队列，可以无限存放任务

（2）如果使用的是有界队列比如 ArrayBlockingQueue，任务首先会被添加到 ArrayBlockingQueue 中，ArrayBlockingQueue 满了，会根据 maximumPoolSize 的值增加线程数量，如果增加了线程数量还是处理不过来，ArrayBlockingQueue 继续满，那么则会使用拒绝策略 RejectedExecutionHandler 处理满了的任务，默认是 AbortPolicy

39、Java 中用到的线程调度算法是什么

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。

40、什么是线程调度器(Thread Scheduler)和时间分片(TimeSlicing)？

线程调度器是一个操作系统服务，它负责为 Runnable 状态的线程分配 CPU 时间。一旦我们创建一个线程并启动它，它的执行便依赖于线程调度器的实现。时间分片是指将可用的 CPU 时间分配给可用的 Runnable 线程的过程。分配 CPU 时间可以基于线程优先级或者线程等待的时间。线程调度并不受到 Java 虚拟机控制，所以由应用程序来控制它是更好的选择（也就是说不要让你的程序依赖于线程的优先级）。

41、什么是自旋

很多 synchronized 里面的代码只是一些很简单的代码，执行时间非常快，此时等待的线程

都加锁可能是一种不太值得的操作，因为线程阻塞涉及到用户态和内核态切换的问题。既然 synchronized 里面的代码执行得非常快，不妨让等待锁的线程不要被阻塞，而是在 synchronized 的边界做忙循环，这就是自旋。如果做了多次忙循环发现还没有获得锁，再阻塞，这样可能是一种更好的策略。

42、Java Concurrency API 中的 Lock 接口(Lock interface)是什么？对比同步它有什么优势？

Lock 接口比同步方法和同步块提供了更具扩展性的锁操作。他们允许更灵活的结构，可以具有完全不同的性质，并且可以支持多个相关类的条件对象。

它的优势有：

（1）可以使锁更公平

（2）可以使线程在等待锁的时候响应中断

（3）可以让线程尝试获取锁，并在无法获取锁的时候立即返回或者等待一段时间

（4）可以在不同的范围，以不同的顺序获取和释放锁

43、单例模式的线程安全性

老生常谈的问题了，首先要说的是单例模式的线程安全意味着：某个类的实例在多线程环境下只会被创建一次出来。单例模式有很多种的写法，我总结一下：

（1）饿汉式单例模式的写法：线程安全

（2）懒汉式单例模式的写法：非线程安全

（3）双检锁单例模式的写法：线程安全

44、Semaphore 有什么作用

Semaphore 就是一个信号量，它的作用是限制某段代码块的并发数。Semaphore 有一个构造函数，可以传入一个 int 型整数 n，表示某段代码最多只有 n 个线程可以访问，如果超出了 n，那么请等待，等到某个线程执行完毕这段代码块，下一个线程再进入。由此可以看出如果 Semaphore 构造函数中传入的 int 型整数 n=1，相当于变成了一个 synchronized 了。

45、Executors 类是什么？

Executors 为 Executor，ExecutorService，ScheduledExecutorService，ThreadFactory 和 Callable 类提供了一些工具方法。Executors 可以用于方便的创建线程池

46、线程类的构造方法、静态块是被哪个线程调用的

这是一个非常刁钻和狡猾的问题。请记住：线程类的构造方法、静态块是被 new 这个线程类所在的线程所调用的，而 run 方法里面的代码才是被线程自身所调用的。

如果说上面的说法让你感到困惑，那么我举个例子，假设 Thread2 中 new 了 Thread1，main 函数中 new 了 Thread2，那么：

（1）Thread2 的构造方法、静态块是 main 线程调用的，Thread2 的 run()方法是 Thread2 自己调用的

（2）Thread1 的构造方法、静态块是 Thread2 调用的，Thread1 的 run()方法是 Thread1 自己调用的

47、同步方法和同步块，哪个是更好的选择?

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。

48、Java 线程数过多会造成什么异常？

（1）线程的生命周期开销非常高
（2）消耗过多的 CPU 资源
如果可运行的线程数量多于可用处理器的数量，那么有线程将会被闲置。大量空闲的线程会占用许多内存，给垃圾回收器带来压力，而且大量的线程在竞争 CPU 资源时还将产生其他性能的开销。

（3）降低稳定性
JVM 在可创建线程的数量上存在一个限制，这个限制值将随着平台的不同而不同，并且承受着多个因素制约，包括 JVM 的启动参数、Thread 构造函数中请求栈的大小，以及底层操作系统对线程的限制等。如果破坏了这些限制，那么可能抛出 OutOfMemoryError 异常。