**Java面试总结**

# 并发

## 1 线程

### 什么是进程?

进程是程序的一次执行，是操作系统**资源分配的基本单位**。系统运行一个程序即是一个进程从创建、运行到销毁的过程。

### 什么是线程?

线程是现代操作**系统调度的基本单位**，在一个进程中可以创建多个线程。相比进程来说，线程的创建、销毁和切换开销更小，所以线程也被叫做**轻量级进程**。多个线程可以**共享进程的内存空间**，通信方便高效。

### 线程与进程的区别？：单位、开销、通信

进程是程序的一次执行，是资源分配的基本单位。线程是系统调度的基本单位，在一个进程中可以创建多个线程。相比进程来说，线程的创建、销毁和切换开销更小，而且多个线程可以共享进程的内存空间，通信更加方便高效。

### 为什么要引入线程？（线程相比进程的优势）

进程的创建、销毁和切换开销较大，而且进程间通信，因为要经过内核，开销也较大。而线程的创建、销毁和切换开销更小，多个线程可以共享进程的内存空间，通信方便快捷。所以引入了线程，来进一步提高程序的执行效率。

### 线程有哪些分类？

Java线程分为守护线程和用户线程两类。守护线程一般由JVM创建，如GC线程，用户线程由程序创建。JVM会等待用户线程结束再退出，但不会等待守护线程。只要所有的用户线程都结束了，所有守护线程也必须结束。

### 什么是守护线程，如何创建守护线程？

Java可以创建两种线程，用户线程和守护线程。守护线程也就是后台线程，是用来给用户线程提供服务的。只要当前JVM实例中尚存在没有结束的非守护线程，守护线程就全部工作；只有当最后一个非守护线程结束时，守护线程随着JVM一同结束工作。常见的守护线程如GC线程、Finalizer线程等。

使用Thread类的setDaemon(true)方法可以将线程设置为守护线程。需要注意的是，需要在start()方法前调用这个方法，否则会抛出IllegalThreadStateException异常。

### 说说线程的生命周期和状态?：6状态、状态转换

Java线程在生命周期内有6种状态，Java线程在生命周期中在这6个状态间不停转换。当Thread对象被创建，但还没有调用start()方法前，处于新建状态（NEW）。当Thread对象调用start()方法后，开始运行，线程处于可运行状态状态（RUNNABLE）。当调用不带超时时间参数的Object::wait()、Thread::join()、LockSupport::park()方法后，线程进入等待状态（WAITING）。进入等待状态的线程需要依靠其它线程的通知才能够返回到可运行状态，比如其它线程调用对应的Object::notify()、Object::notifyAll()、LockSupport::unpark()方法，或者Object::join()方法等待的线程结束。当调用带超时时间参数的Object::wait(long)、Object::join(long)、Thread::sleep(long)、LockSupport::parkNanos(long)、LockSupport::parkUntil(long)方法会使线程进入超时等待状态(TIMED\_WAITING)，当超时时间到达或等待事件发生后线程将会返回到可运行状态。当线程调用同步方法时，在没有获取到锁的情况下，线程将会进入到阻塞状态（BLOCKED），当线程成功获取锁后，线程会返回到可运行状态。线程在执行完run()方法后进入到终止状态(TERMINATED)。

### 你对线程优先级的理解是什么？：1-10优先级、java与os优先级映射

Java中线程**优先级分为1-10**。在两个线程同时处于就绪状态时，优先级越高的任务越容易被操作系统选择执行。因为Java目前主流平台主流JVM采用的都是内核线程来实现的Java线程，所以线程的调度是交给操作系统来完成的。Java线程优先级只是JVM给操作系统的一个建议，并不是一项稳定的调节手段。因为在优先级数量比Java优先级数量少的操作系统中，Java优先级的多个优先级可能在操作系统中对应同一个优先级。而且操作系统本身也会对线程的优先级做一些优化调节。所以我们并不能在程序中通过线程优先级来完全准确判断一组都为就绪状态的线程将会先执行哪一个。

### 创建线程的方式有哪些，Callable返回值作用是什么？

创建线程的方式有继承Thread类、实现Runnable接口、实现Callable接口。Runnable没有返回值，Callable会返回一个Future对象。Future对象用于查询异步任务的状态、获取最终结果、取消任务等。

### callable，runnable有什么区别？

两者都可以用于创建线程，都可以使用在Executor框架中。Callable接口的call()方法允许有返回值，通过泛型标识返回值类型，允许抛出异常。Runnable接口的run()方法没有返回值，不允许抛出异常。Thread类只支持Runnable接口不支持Callable接口。

### 可以直接调用Thread类的run ()方法么？（为什么我们调用start()方法时会执行run()方法，为什么我们不能直接调用run()方法？）

直接调用Thread的run()方法不会开启新线程去执行run()方法的代码，就像调用一个普通方法一样，由调用run()方法的线程本身去执行。调用start()方法则会开启一个新线程，并将run()方法中的代码作为新线程的执行代码。

### 如何正确地终止线程？

对于不清楚其内部运行状态的线程，应该调用其封装好的取消和关闭方法来终止线程。对于清楚其内部运行状态或没有提供取消和关闭方法的线程，可以使用两种方法来终止线程。第一种是使用volatile共享变量标志线程状态，在状态改为终止标记时结束线程，此方式无法终止处于等待和超时等待状态的线程。第二种是通过interrupt()方法、isInterrupted()方法和处理InterruptedException异常配合使用终止线程，中断机制属于协作机制，不能终止处于阻塞状态的线程。

### 线程不正常终止会发生什么，线程占用的空间具体是哪，是寄存器还是内存还是其他地方?

线程不正常终止可能会导致线程持有的**资源不能正确释放**，通常是没有给予线程完成资源释放工作的机会，因此会导致程序可能工作在不确定状态下。Java线程占用的是操作系统分配给JVM进程的用来分配线程私有内存的区域，线程占用空间对应运行时区域中的虚拟机栈、本地方法栈、程序计数器，而虚拟机栈的部分区域对应于JMM中的线程工作内存，线程的工作内存会优先存储于寄存器和高速缓存中。

### 如何确保main()方法所在的线程是Java 程序最后结束的线程？

可以使用Thread::join()方法来等待所有用户线程结束，然后再结束main线程。

### 如何让正在运行的线程暂停一段时间？

可以使用Thread::sleep(long)方法来让线程暂停执行一段时间，当暂停时间到了之后，线程会进入可执行状态。

### Java中你怎样唤醒一个阻塞的线程？

对于在同步方法或同步块中使用Object::wait()方法进入等待状态的线程，可以使用Object:notify()随机唤醒一个在锁对象上等待的线程，也可以使用Object::notifyAll()方法唤醒全部在锁对象上等待的线程。当被唤醒的线程获取到锁之后，线程才可以进入同步块或同步方法中执行。

### Java的线程和Linux的线程有什么区别，为什么需要Java的线程？

在主流平台的主流JVM中，Java线程是使用内核线程实现的，内核线程直接由操作系统支持。

### 一个线程运行时发生异常会怎样？

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部接口。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。

### 线程通信的方式有哪些？（线程之间是如何通信的？如何在两个线程间共享数据？java如何实现多线程之间的通讯和协作？）

在命令式编程中线程通信的机制有两种，**共享内存**和**消息传递**。在共享内存的并发模型里线程间共享程序的公共状态，通过读写内存中的公共状态实现隐式通信。在消息传递的并发模型里线程间没有公共状态，必须通过发送消息来进行显示通信。Java并发采用共享内存模型，线程之间的通信总是隐式进行，整个通信过程对程序员完全透明。Java

使用volatile修饰的共享变量进行通信。

使用synchronized的可见性保证来实现通信。

使用wait()、notify()、notifyAll()方法实现的等待通知机制来进行通信。

使用join()方法来进行通信，join()方法底层也是使用的wait()、notify()、notifyAll()方法实现的等待通知机制。

使用管道输入输出流进行通信，管道流底层使用一个默认大小为1KB的循环缓冲数组，输入流从循环缓冲数组中读数据，输出流从循环缓冲数组中写数据，当缓冲数组已满时，输出流所在线程进入等待状态，当缓冲数组首次为空时，输入流所在线程进入等待状态。

### 为什么要引入多线程？

首先是为了进一步提高程序的运行速度。还有就是提高对硬件**资源的利用率**，因为摩尔定律逐渐失效，现代计算机芯片采用**横向扩展**，多核CPU多个核心可以并行地运行多个线程，采用多线程可以更高效地利用硬件资源。

### 多线程一定快吗?

不一定。因为多线程程序存在着大量的线程创建、销毁和上下文切换动作，如果这些动作太过于频繁，将会降低程序吞吐量，使程序运行效率降低。

### 进程的切换过程是怎样的？

进程的切换过程就是被中断的进程和待运行进程的上下文切换。

### 什么是多线程中的上下文切换？

CPU的每一个核心在同一时刻都只能执行一个线程，CPU为了保证每个线程都能及时响应，采取了时间片轮转的调度算法。当前线程执行一个时间片后会切换到另一个线程。但是，在切换前会保存上一个线程的状态，以便下次切换回这个线程时，可以再加载这个线程的状态。所以线程从保存到再加载的过程就是一次上下文切换。

### 如何减少上下文切换？

减少上下文切换的方法有无锁并发编程、CAS算法、使用最少线程和使用协程等。

**无锁并发编程**：多线程竞争锁时，会引起上下文切换，所以多线程处理数据时，可以采用一些办法来避免使用锁，如将数据进行分段，不同的线程处理不同分段的数据。

**CAS算法**：Java的Atomic包提供了CAS算法来更新数据。CAS会调用CPU的Compare And Swap指令来进行原子的“读-改-写”操作，不需要加锁。

**使用最少线程**：避免创建不必要的线程，比如任务很少，但是创建了大量的线程来处理，这样会造成大量线程都处于等待状态。

**协程**：在单线程里实现多任务的调度，并在单线程里维持多个任务间的切换。

### 什么是死锁，如何避免死锁，活锁是什么？

死锁是指由于竞争资源或者通信关系，两个或更多线程在执行中出现，永远互相等待只能由其他线程引发的事件。

避免死锁只需要破环形成死锁的4个必要条件之一就可以了。形成死锁的4个必要条件是：

**互斥访问资源**：可以把互斥的共享资源封装成可同时访问。

**持有资源并等待其他资源**：可以要求程序请求资源时，不能持有其他资源，必须一次请求所有需要的资源。

**不可抢占**：如果程序请求了不能立即分配的资源，则释放已占有资源。

**循坏等待**：可以对资源排序，要求程序按顺序请求资源。

活锁与死锁相似，死锁是线程都在等待对方先释放资源。活锁则是线程彼此释放资源又同时占用对方释放的资源，当此情况持续发生时，尽管资源的状态不断改变，但每个线程都不能获取到所需的全部资源，从而发生了活锁。活锁与死锁的区别在于，活锁的资源状态是在不断改变的，而死锁的资源状态是静止的。活锁可以自行解开，死锁不能通过外力解开。

### 多线程同步和互斥有几种实现方法，都是什么？

同步是指控制不同线程间操作发生的相对顺序。互斥是指多个线程对于共享资源进行排它性访问，当有若干个线程都要使用某一共享资源时，任何时刻最多只允许一个线程去使用，其它要使用该资源的线程必须等待，直到占用资源者释放该资源。操作系统实现同步互斥的方法有原子操作、临界区、**信号量**、管程等。

### 使用多线程可能带来什么问题?

并发访问问题（死锁、数据不一致）、内存泄漏、频繁的上下文切换导致效率降低等。

### 为什么你应该在循环中检查等待条件?

因为线程被唤醒后，可能等待的条件状态又发生了变化，此时需要继续等待。

### 线程安全的本质是什么？

当多个线程同时访问一个对象时，如果不考虑这些线程在运行时环境下的交替执行，也不需要进行额外的同步，或者在调用方进行任何其它的协调操作，调用这个对象的行为都可以获得正确的结果，那就称这个对象是线程安全的。（串行执行与并发执行都能得到正确的结果）

### Java如何保证线程安全？（怎么实现线程安全性？）

保证线程安全可以采用以下手段：

使用**不可变对象**，不可变对象一定是线程安全的。

进行**同步互斥**，包括**synchronized**、**volatile**、**锁**、**等待通知机制**、**原子类**的**循环CAS**操作等。

**线程隔离**，让多个线程不再共享资源和数据，包括让不同的线程处理不同的数据段，使用**ThreadLocal**线程变量等。

### 什么是线程调度器(Thread Scheduler)和时间分片(Time Slicing )？

线程调度器是操作系统的一个子系统，它负责为就绪状态的线程分配CPU时间。只有被分配了CPU时间的线程才可以运行。时间分片是指将CPU的一个时间片分配给可用的就绪状态的线程的过程。

### 同步和非同步、阻塞和非阻塞是什么？

同步和非同步一般说的是针对若干个任务是同步串行地执行还是异步并发地执行。阻塞和非阻塞一般是说调用一个方法是阻塞获取结果还是立即返回获取结果。

### 线程有哪些方法？

start()、run()、sleep()、join()、yield()、interrupt()、activeCount()、setPriority()、setDaemen()、setName()、isInterrupted()、isAlive()。

### 为什么wait, notify 和 notifyAll这些方法不在Thread类里面，而被定义在Object类里？

Java提供的锁是对象级别而不是线程级别的，每个对象都有锁，通过线程获得。当线程需要使用等待通知机制时，直接调用同一个对象的对应方法即可。

### Thread类中的yield方法有什么作用？

当前线程主动让出CPU，避免线程过度占用。重新竞争时，cpu 也有可能重新选中自己。yield()方法只会给相同优先级或者更高优先级的线程以运行的机会。

### 为什么wait(), notify()和notifyAll ()必须在同步方法或者同步块中被调用？

如果使用wait()、notify()这些方法没有同步的话，那么容易出现竞态条件。竞态条件是指多个线程访问或操作同一个对象，最终结果与时序有关。

### 为什么Thread类的sleep()和yield ()方法是静态的？

Thread类的sleep()和yield()方法将在当前正在执行的线程上运行。所以在其他处于等待状态的线程上调用这些方法是没有意义的。这就是为什么这些方法是静态的。它们可以在当前正在执行的线程中工作，并避免程序员错误地认为可以在其他非运行的线程上调用这些方法。

### interrupt(), interrupted(), isInterrupted()的区别？

Java中终止一个线程的主要机制是中断，中断不是强迫一个线程终止，而是一种协作机制，给线程传递一个信号，由线程决定何时如何退出。

interrupt()方法和isInterrupted()方法是实例方法，interrupted()方法是静态方法。

isInterrupted()方法是返回对应线程的中断标志位是否为true，interrupted()静态方法会调用当前线程的isInterrupted()方法并清空中断标志位。

Interrupt()尝试中断该线程。对线程的影响与线程的状态有关。

1. RUNNABLE状态下仅设置中断标志位。
2. WAITING、TIMED\_WAITING状态下，会使该线程抛出InterruptedException异常。在抛出异常后清空中断标志位。
3. BLOCKED状态会设置中断标志位，但是线程会依旧处于BLOCKED状态。也就是说，其不能真正使一个等待锁的线程中断。

### 说 sleep() 方法和 wait() 方法区别和共同点?

相同点：

两者都让线程进入到 TIMED\_WAITING 状态，并且可以设置等待的时间。

不同点：

wait 是Object类的方法，sleep是Thread 类的方法。

sleep不会释放锁，沉睡的时候，其它线程是无法获得锁的，但 wait 会释放锁。

### 调用sleep(0)有什么效果？

Thread.sleep(0) 是你的线程暂时放弃cpu，也就是释放一些未用的时间片给其他线程或进程使用，可以达到与yield类似的效果。区别是yield()只会给相同优先级或者更高优先级的线程以运行的机会，而sleep(0) 方法给其他线程运行机会时不考虑线程的优先级。sleep()方法比yield()方法具有更好的可移植性。

### 什么是线程组，为什么在Java中不推荐使用？

Java中通过ThreadGroup类来实现线程组。线程组表示一组线程，其目的是为了方便线程的管理。线程组可以包括线程和其他线程组，形成一个树状结构，其中除初始线程组之外的每个线程组都有父节点。其允许线程组访问自己的子线程组的信息，但不允许访问父线程组和任何其他线程组的信息。

## 2 Java并发机制的底层实现

### volatile有什么作用?底层实现是怎样的?

**作用**：

保证变量对所有线程的可见性。禁止指令重排序优化，保证并发执行时的有序性。

**适用场景**:

1如果共享对象只有一个，操作为不依赖原值的get/set操作，不存在竞态条件问题，只需要保障内存可见性。

2一写多读，只有单一的线程修共享对象。

**底层实现**:

volatile通过在机器代码中添加lock前缀指令实现可见性，Lock前缀的指令在多核处理器下会引发两件事情：

1将当前处理器缓存行的数据写回到系统内存。

2这个写回内存的操作会使在其他CPU里缓存了该内存地址的数据无效。

为了实现volatile禁止指令重排的语义，编译器会在生成字节码时，在指令序列中插入内存屏障来禁止特定类型的处理器重排序。

### volatile不能解决什么问题，用什么解决？

不能解决依赖变量当前值的“读改写”操作这种复合操作的原子性问题，会产生竞态条件问题。可以使用CAS、锁、同步块等方式解决。

### volatile 变量和 atomic 变量有什么不同？

volatile仅保障变量的内存可见性。atomic内部成员使用volatile修饰，还提供部分原子的组合操作。

### synchronized的底层实现是怎样的?

JVM基于进入和退出**monito**r对象来实现方法同步和代码块同步。代码块同步使用**monitorenter**和**monitorexit**指令实现，而方法同步是使用另外一种方式实现的，细节在JVM规范里并没有详细说明。但是，方法的同步同样可以使用这两个指令来实现。

monitorenter指令是在编译后插入到同步代码块的开始位置，而monitorexit是插入到代码块结束处和异常处，JVM要保证每个monitorenter必须有对应的monitorexit与之配对。**任何对象都有一个monitor与之关联**，当一个monitor被持有后，它将处于锁定状态。线程执行到monitorenter指令时，将会尝试获取对象所对应的monitor的所有权。

**详细版本**：

每个Java对象都有一个关联的monitor，monitor是一个使用c++实现的叫objectmonitor的c++对象，其中记录了当前持有这个monitor的线程id。使用 synchronized 时 JVM 会根据monitor 的状态进行加解锁的判断。如果成功加锁就会成为该 monitor 的唯一持有者，monitor 在被释放前不能再被其他线程获取。

同步代码块使用 monitorenter 和 monitorexit 这两个字节码指令获取和释放 monitor。monitorenter指令是在编译后插入到同步代码块的开始位置，而monitorexit是插入到代码块结束处和异常处，JVM要保证每个monitorenter必须有对应的monitorexit与之配对。这两个字节码指令都需要一个引用类型的参数指明要锁定和解锁的对象，对于实例同步方法，锁是当前实例对象；对于静态同步方法，锁是当前类的 Class 对象；对于同步方法块，锁是 synchronized 括号里的对象。

执行 monitorenter 指令时，首先尝试获取对象锁。如果这个对象没有被锁定，或当前线程已经持有锁，就把锁的计数器加 1，执行 monitorexit 指令时会将锁计数器减 1。一旦计数器为 0 锁随即就被释放。

被 synchronized 修饰的同步块对一条线程来说是可重入的，并且同步块在持有锁的线程释放锁前会阻塞其他线程进入。从执行成本的角度看，持有锁是一个重量级的操作。Java 线程是映射到操作系统的内核线程上的，要阻塞或唤醒一条线程，需要切换到内核态。

### ****synchronized同步代码块还有同步方法本质上锁住的是谁？为什么？优劣势？****

Java锁定的是对象而非线程。Synchronized同步代码块锁定的是指定的锁对象，静态同步方法锁定的是方法所在类的Class对象，实例同步方法锁定的是实例对象。

同步代码块可自定义锁对象，更加灵活。实例中不同的实例同步方法不可并行。

### 怎么检测一个线程是否拥有锁（持有synchronized锁对象）？

使用Thread ::holdsLock(obj)，当且仅当当前线程拥有某个具体对象的锁时才返回true。

### volatile与synchronized的区别

synchronized可以实现并发过程中的可见性、原子性和有序性，而volatile仅保障可见性和有序性。

大多数场景下，volatile的总开销要比synchronized要低，在符合volatile使用场景的地方应该优先使用volatile。

### CAS是什么，有什么问题，如何解决？

Compare And Swap，是在多线程中保证同步的一个原子指令。可以保证读改写操作的原子性。相比synchronized来说，CAS是一种乐观的、非阻塞的同步机制。

可能导致问题：

1. ABA问题。解决思路：可以给变量增加版本号，atomic包中提供了AtomicStampedReference类来解决这个问题。
2. 如果CAS长期不成功，会造成循环时间长开销大的问题。
3. 只能保证一个共享变量的原子操作。

### 锁有几种状态，有什么限制？

锁一共有四种状态，级别由低到高分别是：无锁状态，偏向锁状态，轻量级锁状态，重量级锁状态，几个状态会随着竞争情况逐渐升级。锁可以升级但不能降级，不降级是为了提高获得锁和释放锁的效率。

### 锁优化有哪些策略？

JDK 6 对 synchronized 做了很多优化，引入了自适应自旋、锁消除、锁粗化、偏向锁和轻量级锁等优化措施来提高synchronized加解锁的效率，锁一共有 4 个状态，级别从低到高依次是：无锁、偏向锁、轻量级锁和重量级锁，状态会随竞争情况升级。锁可以升级但不能降级，这种只能升级不能降级的锁策略是为了提高锁获得和释放的效率。

### 自旋锁是什么？

同步互斥中对性能最大的影响是阻塞，挂起和恢复线程的操作都需要转入内核态完成。而根据经验法则，许多应用上共享数据的锁定只会持续很短的时间，为了这段时间去挂起和恢复线程并不值得。如果机器有多个处理器核心，我们可以让后面请求锁的线程稍等一会，但不放弃处理器的执行时间，看看持有锁的线程是否很快会释放锁。为了让线程等待只需让线程执行一个忙循环，这项技术就是自旋锁。

自旋锁在 JDK1.4 就已引入，默认关闭，在 JDK6 中改为默认开启。自旋不能代替阻塞，虽然避免了线程切换开销，但要占用处理器时间，如果锁被占用的时间很短，自旋的效果就会非常好，反之只会白白消耗处理器资源。如果自旋超过了限定的次数仍然没有成功获得锁，就应挂起线程，自旋默认限定次数是 10。

### 如何实现自旋锁？

采用循环CAS实现自旋锁。

### 设置重量级锁自旋次数的参数是什么？

自旋次数的默认值是10次，可用通过-XX：PreBlockSpin参数来自行更改自旋次数。

### 什么是自适应自旋？

JDK6 对自旋锁进行了优化，自旋时间不再固定，而是由前一次在同一个锁上的自旋时间及锁的拥有者的状态决定。

如果在同一个锁对象上，自旋等待刚刚成功获得过锁，并且持有锁的线程正在运行中，那么虚拟机会认为这次自旋也很可能成功，进而允许自旋持续更久。如果自旋很少成功，以后获取锁时将可能直接省略掉自旋，避免浪费处理器资源。

有了自适应自旋，随着程序运行时间的增长，虚拟机对程序锁的状况预测就会越来越精准。

### 锁消除是什么？

锁消除指虚拟机即时编译器在运行时，对代码要求同步，但又检测到不可能存在共享数据竞争的锁进行消除。之所以会出现这种情况，是因为有许多同步措施并不是程序员自己加入的，而是编译的时候编译器自动加入的。

锁消除的主要判定依据来源于逃逸分析的数据支持，如果判断一段代码中堆上的所有数据都只被一个线程访问，就可以把它们当作栈上的数据对待，认为它们是线程私有的而无须同步。

### 锁粗化是什么？

在原则上，我们在编写代码时需要将同步块的作用范围限制得尽量小，只在共享数据的实际作用域中进行同步，这是为了使同步操作变少，即使存在同步，等待锁的线程也能尽可能快地拿到锁。

但如果一系列的连续操作都对同一个锁对象反复加锁和解锁，甚至加锁操作是出现在循环体之中的，那即使没有线程竞争，频繁地进行互斥同步操作也会导致不必要的性能消耗。因此如果虚拟机探测到有一串零碎的操作都对同一个对象加锁，将会把同步的范围扩展到整个操作序列的外部。

### 什么是偏向锁，偏向锁的获取与撤销过程是怎样的，偏向锁升级为轻量级锁的过程是怎样的？

偏向锁是jdk6中引入的一项锁优化措施，它的目的是**消除数据在无竞争情况下的同步原语，从而提高加锁和解锁的效率**。如果线程已经获取到偏向锁，以后该线程在进入和退出同步代码块时不需要再进行CAS操作来加解锁，只需简单地测试一下锁对象对象头Mark Word中是否存储着指向当前线程的偏向锁即可。

**获取**：

当线程即将进入同步块的时候，它首先会去判断锁对象的Mark Word中的锁标志位，如果是‘01’，则表示锁状态处于无锁状态或偏向锁状态。然后再根据偏向锁标志位判断锁到底处于哪个状态。如果是‘0’，也就是处于无锁状态，则进行CAS操作把当前线程ID记录到Mark Word中，如果操作成功，则成功获取到锁，如果操作失败，则进行偏向锁撤销。如果是‘1’，也就是处于偏向锁状态，则检查Mark Word中记录的线程ID是否是当前线程ID，如果是，则表示已经获取到锁，无需再进行加锁。如果不是，则进行CAS操作替换Mark Word中的线程ID，替换成功，则成功获取到锁，替换失败，则进行偏向锁撤销。

**撤销**：

偏向锁的撤销是由获取偏向锁失败的线程来执行的，它首先会等待持有偏向锁的线程执行到全局安全点，然后暂停它，再判断持有偏向锁的线程的状态，如果其未退出同步代码块，则进行偏向锁升级为轻量级锁的操作。如果其处于未活动状态或已经退出了同步代码块，则将锁对象重置为无锁状态，然后唤醒原来持有偏向锁的线程，再去重新竞争偏向锁。

**偏向锁升级为轻量级锁**：

偏向锁升级为轻量级锁是由获取偏向锁失败的线程来执行的，它首先会在持有偏向锁的线程的栈帧中创建用于存放锁记录的空间，并将锁对象的Mark Word复制到持有偏向锁的线程的锁记录中，然后将对象头中的Mark Word替换成指向锁记录的指针，并修改锁标志位为’00‘，表示锁处于轻量级锁状态，完成这些也就意味着原持有偏向锁的线程已经成功获取到了升级后的轻量级锁了，然后获取偏向锁失败的线程再唤醒原持有偏向锁的线程，让它从安全点开始继续执行，自己则去参加新一轮的锁竞争。

### 什么是轻量级锁，轻量级锁的加锁与解锁过程是怎样的？

轻量级锁是jdk6中引入的一项锁优化措施，它是在没有多线程竞争的前提下，使用CAS来代替重量级的互斥量，从而减少传统的重量级锁产生的大量开销。

**加锁**：

当线程即将进入同步块时，它首先会去判断锁对象对象头中Mark Word中的锁标志，如果是‘00‘，即锁处于轻量级锁状态。则JVM会在当前线程的栈帧中创建用于存放锁记录的空间，并将锁对象的Mark Word复制到其锁记录中。然后线程尝试使用CAS将对象头中的Mark Word替换成指向锁记录的指针，替换成功则表示成功获取锁，替换失败则进行自旋来替换Mark Word，当自旋替换成功后，则表示成功获取到锁，若自旋超过一定次数仍未成功，则将轻量级锁升级为重量级锁。

**解锁**：

解锁时，当前线程会首先判断锁对象的Mark Word是否仍然指向当前线程栈帧中的锁记录，如果是，那就用CAS操作将Mark Word替换回去。如果替换成功，那解锁过程就完成了，如果替换失败或者是锁对象的Mark Word已经不再指向当前线程栈帧中的锁记录了，那么说明有其他线程尝试过获取该锁，就要在释放锁的同时，唤醒被挂起的线程，然后这些线程会进行新一轮的锁竞争。

## 3 Java内存模型

### JMM 的作用是什么？

Java 线程的通信由 JMM 控制，JMM 的主要目的是定义程序中各种变量的访问规则。变量包括实例字段、静态字段和构成数组对象的元素，但不包括局部变量与方法参数，因为它们是线程私有的，不存在多线程竞争。JMM 遵循一个基本原则：只要不改变程序执行结果，编译器和处理器可以进行任意优化操作。例如编译器分析某个锁只会被单线程访问就消除锁，某个volatile变量只会被单线程访问就把它设置为普通变量。

JMM 规定所有变量都存储在主内存中，而每个线程都有自己的工作内存，工作内存中保存被该线程使用的变量的主内存副本，线程对变量的所有操作都必须在自己的工作内存中进行，不能直接读写主内存数据。不同线程间无法直接访问其他线程工作内存中的数据，线程间通信必须经过主内存。

### as-if-serial 是什么？

不管怎么重排序，单线程程序的执行结果都不能改变，编译器和处理器必须遵循 as-if-serial 语义。

为了遵循 as-if-serial，编译器和处理器不会对存在数据依赖关系的操作进行重排序，因为这种重排序会改变执行结果。但是如果操作之间不存在数据依赖关系，这些操作就可能被编译器和处理器重排序。

as-if-serial 把单线程程序保护起来，给程序员造成了一种“单线程程序是按照程序语句的顺序执行”的感觉。

### 什么是happens-before原则，有哪些常见的happens-before原则？

happens-before原则也叫先行发生原则，先行发生是JMM 定义的两项操作间的偏序关系，是判断数据是否存在竞争、线程是否安全的重要手段。

JMM 将 happens-before 要求禁止的重排序按是否会改变程序执行结果分为两类。对于会改变结果的重排序 JMM 要求编译器和处理器必须禁止，对于不会改变结果的重排序，JMM 不做要求。

JMM 存在一些天然的 happens-before 关系，无需任何同步器协助就已经存在。如果两个操作的关系不在此列，并且无法从这些规则推导出来，则它们就没有顺序性保障，虚拟机可以对它们随意进地进行重排序。天然的8条happens-before规则如下:

程序次序规则：一个线程内书写在前面的操作先行发生于书写在后面的操作。

管程锁定规则： unlock 操作先行发生于后面对同一个锁的 lock 操作。

volatile 规则：对volatile变量的写操作先行发生于后面对这个变量的读操作。

线程启动规则：线程的 start 方法先行发生于线程的每个动作。

线程终止规则：线程中所有操作先行发生于对线程的终止检测。

线程中断规则：对线程interrupt方法的调用先行发生于线程检测到中断事件的发生。

对象终结规则：对象的初始化完成先行发生于它的 finalize 方法的开始。

传递性：如果操作 A 先行发生于操作 B，操作 B 先行发生于操作 C，那么操作 A 先行发生于操作 C 。

### as-if-serial 和 happens-before 有什么区别？

as-if-serial 保证单线程程序的执行结果不变，happens-before 保证正确同步的多线程程序的执行结果不变。这两种语义的目的都是为了在不改变程序执行结果的前提下尽可能提高程序执行并行度。

### 什么是重排序，有哪些重排序？

在执行程序时，为了提高性能，编译器和处理器常常会对指令进行重排序。重排序分3种类型：

**编译器优化的重排序**。编译器在不改变单线程程序语义的前提下，可能重新安排语句的执行顺序。

**指令级并行的重排序**。现代处理器采用了指令级并行技术来将多条指令重叠执行。如果不存在数据依赖性，处理器可以改变语句对应机器指令的执行顺序。

**内存系统的重排序**。由于处理器使用缓存和读写缓冲区，这使得加载和存储操作看上去可能是在乱序执行。

### 内存重排序都会发生在哪，为什么要进行内存重排序？

内存系统重排序是由于处理器使用缓存和读写缓冲区，这使得加载和存储操作看上去可能是在乱序执行。进行内存系统重排序是为了提高程序运行效率，因为线程的工作内存对应于硬件中的寄存器、高速缓冲、读写缓冲区等，而主内存对应于硬件中的物理内存。每个线程对自己工作内存的共享变量副本进行操作，比直接操作主内存的共享变量要高效和安全。

### 什么是缓存一致性问题，如何解决？

为了加快程序的执行速度，数据内容的修改优先在缓存中进行。在多处理器环境中，每个核心的修改对其他核心不可见，这就造成了缓存一致性问题。可通过缓存一致性协议解决缓存一致性问题。Java中通过volatile、final、同步锁、显示锁等可以保证内存可见性的机制解决缓存一致性问题。

### final 可以保证可见性吗？

可以。在不发生this逃逸的情况下，final的内存语义会保证在构造函数中对final变量的赋值操作不会被重排到把这个构造对象的引用赋值给一个引用变量之后。

初次读一个包含final域的对象的引用，与随后初次读这个final域，这两个操作之间不能重排序。

### 原子性、可见性、有序性分别是什么？

**原子性**:指一组操作要么全执行要么全不执行。

Java中基本数据类型的访问都具备原子性，除了long和double在32位的机器上可能不是原子的。如果应用场景需要更大范围的原子性保证，JMM 还提供了 lock 和 unlock 操作来满足需求，尽管 JVM 没有把这两种操作直接开放给用户使用，但是提供了更高层次的字节码指令 monitorenter 和 monitorexit，这两个字节码指令反映到Java代码中就是 synchronized。

**可见性**:指一个线程修改了共享变量时，其他线程能够立即得知这个修改。

JMM 通过在变量修改后将值同步回主内存，在变量读取前从主内存刷新的方式实现可见性。Java保证可见性的手段有volatile、synchronized 和final。同步块可见性由"对一个变量执行unlock前必须先把此变量同步回主内存这条规则获得。final 的可见性是指：被final修饰的字段在构造方法中一旦初始化完成，并且没有发生this逃逸，那么其他线程就能看到 final 字段的值。

**有序性**:因为存在指令重排序和工作内存与主内存间的同步延迟，多线程环境中，从一个线程看另一个线程，所有的操作都是无序的。有序性指保证多个线程间操作执行顺序是和书写的顺序一致的或者可以达到结果一致的。

Java 提供 volatile 和 synchronized 来保证有序性，volatile本身就包含禁止指令重排序的语义，而 synchronized 会保证一个变量在同一时刻只允许一条线程对其进行 lock 操作，从而确保持有同一个锁的两个同步块只能串行地进入。

## 4 显示锁

### AQS是什么，AQS的底层原理是怎样的？

AQS也就是抽象队列同步器，它是用来构建锁或其他同步组件的基础框架。AQS开放出表示同步状态的state字段，让子类根据state字段决定是否可获得锁。对不可获得锁线程的管理则由AQS来实现，无需子类关心。

AQS底层由同步队列和条件队列构成。同步队列管理得不到锁的线程的排队与释放操作。条件队列是对同步队列的补充，用来管理在特定条件上等待的线程的排队与释放操作，比如在生产者消费者模型中当缓冲区满时不能进行生产，当缓冲区空时不能进行消费。

AQS围绕两个队列提供四大场景操作，分别是：获得锁、释放锁、条件队列阻塞、条件队列唤醒。

1. 获取锁：通过Lock.lock()尝试上锁，内部会通过组合的AQS的acquire系列方法上锁。
   1. 获取排它锁：
      1. 先调用tryAcquire()方法尝试获取锁。
      2. 若尝试获取锁失败则使用addWaiter()方法将当前线程放入同步队列尾部(先尝试CAS将node加入队尾，失败则自旋加入队尾)。
      3. 调用acquireQueued()方法自旋尝试将前置节点状态变为 signal，然后阻塞自己；获得锁的线程执行完成释放锁时，会把阻塞的node唤醒令其自旋尝试获得锁。
   2. 获取共享锁：
      1. 先调用tryAcquireShared()方法尝试获取共享锁。
      2. 失败则调用doAcquireShared()获取锁。该方法内部首先会调用addWaiter()将线程节点加入同步队列尾部，然后再在同步队列中通过自旋来获取锁，在获取到排它锁时会通过setHead()将当前节点设置为同步队列队头，而在获取到共享锁时则会通过setHeadAndPropagate()不仅将当前节点设置为头结点而且会唤醒后续共享模式的等待节点。
2. 释放锁：释放锁通过Lock.unLock () 方法，会调用AQS的release系列方法进行解锁。
   1. 释放排它锁:
      1. 使用tryRelease()方法释放锁。
      2. 调用unparkSuccessor()唤醒下一个节点。从队头开始，找它的下一个节点，如果下一个节点是空的，就会从队尾开始找，一直找到状态不是取消的节点，然后唤醒该节点。
   2. 释放共享锁: 释放共享锁的方法是 releaseShared
      1. 调用tryReleaseShared ()尝试释放当前共享锁，失败返回false。
      2. 然后调用doReleaseShared()唤醒当前节点的后续阻塞节点，该方法会唤醒后续共享模式的所有等待节点。
3. 条件队列阻塞：调用Condition.await()方法
   1. 先调用addConditionWaiter()将当前线程节点加入条件队列队尾，若加入前队尾节点状态不为CONDITION则会触发unlinkCanceledWaiter()删除所有不为CONDITION状态的节点（通常是CHANCELED状态的节点）。
   2. 调用fullyRelease()方法释放当前线程持有的锁资源。
   3. 再次确认线程节点在条件队列中，然后阻塞自己，等待被唤醒。
   4. 当其他线程调用signal、signalAll方法时阻塞线程被唤醒，然后执行acquireQueued方法自旋获取锁。
4. 条件队列唤醒：调用Condition.signal或Condition.signalAll方法
   1. 调用signal方法会从等待队列的头结点开始,执行doSignal方法将条件队列中的节点转移到同步队列中，并将其前 置节点状态置为SIGNAL。

### AQS 有哪两种模式？

AQS有独占模式和共享模式，独占模式表示锁只会被一个线程占用，其他线程必须等到持有锁的线程释放锁后才能获取锁，同一时间只能有一个线程获取到锁。

共享模式表示多个线程获取同一个锁有可能成功，ReadLock就采用共享模式。

独占模式通过 acquire 和 release 方法获取和释放锁，共享模式通过 acquireShared 和 releaseShared 方法获取和释放锁。

### AQS中为什么只有前驱节点是头节点时才能尝试获取锁？

因为一个节点被唤醒可能是因为被前驱节点唤醒，也可能是因为被中断唤醒。而持有锁的节点是头结点，**如果前驱节点不是头结点，那么该节点就是被中断唤醒**，不能尝试获取锁。同时这也可以维护同步队列的**FIFO**原则。

### sychronized与ReentrantLock的区别，有sychronized为什么还要引入ReentrantLock？

ReentrantLock可以实现sychronized的**语义**，并且可以支持**非阻塞**方式获取锁、可以**响应中断**、可以**限时**、可以实现**公平锁**、可以绑定多个**条件**。

Synchronized通过JVM关键字实现，是一种**声明式编程思维**，使用更加方便。显示锁代表一种**命令式编程思维**，有更好的可操作性。

编译器可以不断**优化**synchronized实现，在较新版本JVM上，ReentrantLock和synchronized**性能**接近。

### ReentrantLock 的可重入是怎么实现的？

通过getExclusiveOwnerThread()方法判断持有锁的线程是否为当前线程，如果是，则可以重入，否则不可以。每次进入同步块都会将AQS的state加一，每次离开同步块都会将AQS的state减一，当state等于0时才表示锁已经释放。

### 什么是读写锁，锁降级是什么，state变量的高16位和低16位分别保存什么信息？

读写锁允许在同一时刻可以有多个读线程访问，但当一个写线程访问时，其他的读写线程都会被阻塞。适用于读多写少的场景。

锁降级是指把持住当前持有的写锁，再获取到读锁，随后再释放先前拥有的写锁的过程。这样做可以保证读操作可以访问到刚才修改的数据，而不会发生读操作读取到被覆盖了的数据。

state是AQS的锁状态，读写锁中state的高16位表示读状态，低16位表示写状态。若state不为0，而写状态为0，则表示读状态大于0，即读锁已经被获取。

### 如何实现的公平锁，非公平锁？

AQS同步队列中已经是满足FIFO，不公平现象仅发生于当有其他线程想要获取锁时，刚好有一个线程释放了锁，这时这个线程可能先于在同步队列中等待的线程获取到锁这种情况。实现是通过在tryAcquire方法中使用CAS获取锁前，先调用hasQueuedPredecessors()函数进行判断，**只有当同步队列为空或者当前线程就是头节点时才能获取锁**。

### 什么是悲观锁和乐观锁，什么时候用乐观锁，乐观锁和悲观锁有哪些实现方式？

悲观锁就是每次都往最坏的情况考虑，认为每次访问共享数据时都会有其他访问者去修改共享数据，所以每次访问共享数据都进行加锁。适合写多读少的场景。Java中常见的synchronized、ReentrantLock等都是悲观锁，数据库中的表锁、行锁、页锁等也都是悲观锁。

乐观锁就是每次都往最好的情况考虑。认为每次访问共享数据时都不会有其他访问者去修改共享数据，所以不会进行上锁，但修改数据时会通过某种方法判断数据是否被其他访问者修改过，如果被修改过，就进行重试。适合读所写少的情况。常见的实现乐观锁的手段有循环CAS和版本号机制。

## 5 Java并发容器与框架

### Java中的同步容器与并发容器有什么区别？

同步容器与并发容器都为并发提供了线程安全的容器，不过并发容器的并发性和可扩展性更高。

同步容器包括Vector、HashTable以及使用Collections.synchronizedList等一系列方法返回的同步容器。它们使用synchronized来实现同步，整个对象都使用一个锁进行同步。

并发容器使用了与同步容器完全不同的加锁策略来提供更高的并发性和可扩展性，例如ConcurrentHashMap中采用了一种粒度更细的加锁机制——分段锁，在这种锁机制下，允许任意数量的读线程并发地访问Map，并且执行读操作的线程和写操作的线程也可以并发访问Map，同时允许一定数量的写线程并发地修改Map，所以它可以在并发环境下实现更高的吞吐量。

### JDK7 的 ConcurrentHashMap 原理？

ConcurrentHashMap是线程安全且高效的并发容器，用于解决 HashMap 的线程不安全问题和 HashTable 的并发效率低的问题。HashTable 之所以效率低是因为所有线程都必须竞争同一把锁。而HashMap在并发更新的情况下，可能出现死循环，占满cpu，也可能导致丢失更新，死循环出现在多个线程同时扩容哈希表的时候，而不是同时更新一个链表的时候，同时更新一个链表只可能导致丢失更新。

ConcurrentHashMap实现高效并发的机制主要有两个。首先就是**锁分段**技术，将数据分为多个段，每个段都有一个独立的锁进行保护。ConcurrentHashMap由Segment数组结构和HashEntry数组结构组成，Segment是一种可重入锁，HashEntry则用于存储键值对数据。一个Segment里包含一个HashEntry数组，当对HashEntry数组的数据进行修改时，必须首先获得它对应的Segment锁。第二个机制就是**读不需要锁**，而在同步容器中读操作也需要上锁，这是通过将get()里需要使用的共享变量都定义成volatile类型实现的，保证了共享变量在多个线程之间的可见性，比如用于统计当前Segment大小的count字段和用于存储值的HashEntry的value字段。

**get操作**：get 实现简单高效，先经过一次再散列，再用这个散列值通过散列运算定位到 Segment，最后通过散列算法定位到元素。get 的高效在于不需要加锁，除非读到空值才会加锁重读。因为get 方法中将共享变量定义为 volatile类型，而在 get 操作里只需要读共享变量所以不用加锁。

**put操作**：put 必须加锁，首先定位到 Segment，然后进行插入操作，第一步判断是否需要对 Segment 里的 HashEntry 数组进行扩容，第二步定位添加元素的位置，然后将其放入数组。

**size操作**：size 操作用于统计元素的数量，必须统计每个 Segment 的大小然后求和，在统计结果累加的过程中，之前累加过的 count 变化几率很小，因此先尝试两次通过不加锁的方式统计结果，如果统计过程中容器大小发生了变化，再加锁统计所有 Segment 大小。判断容器大小是否发生变化根据 modCount 确定，在put、remove、clear方法里操作元素前都会将变量modCount加1，所以在统计size前后比较modCount是否发生了变化就可以得知容器大小是否发生了变化。

### Java中ConcurrentHashMap的并发度是什么？

并发度指的是数据段的个数，也就是预估的并行更新线程的个数，可以通过构造函数的concurrentLevel参数进行设定，其会被自动转化为2的整数次幂。

### JDK8 的 ConcurrentHashMap 原理？

主要对 JDK7的ConcurrentHashMap 做了三点改造：① **取消分段锁机制**，给每个哈希槽都配备了一个锁，进一步降低冲突概率。② **引入红黑树结构**，同一个哈希槽上的元素个数超过一定阈值后，单向链表改为红黑树结构。③ 使用了**更加优化的方式统计集合内的元素数量**。具体优化表现在：在 put、resize 和 size 方法中**涉及元素总数的更新和计算都避免了锁，使用 CAS 代替**。

get 同样不需要同步，put 操作时如果没有出现哈希冲突，就使用 CAS 添加元素，否则使用 synchronized 加锁添加元素。

当某个槽内的元素个数达到 7 且 容器中的元素数量不小于 64 时，链表转为红黑树。当某个槽内的元素减少到 6 时，由红黑树重新转为链表。在转化过程中，使用同步块锁住当前槽的首元素，防止其他线程对当前槽进行增删改操作，转化完成后利用 CAS 替换原有链表。由于 TreeNode 节点也存储了 next 引用，因此红黑树转为链表很简单，只需从 first 元素开始遍历所有节点，并把节点从 TreeNode 转为 Node 类型即可，当构造好新链表后同样用 CAS 替换红黑树。

### 红黑树和平衡二叉树的区别，HashMap和ConcurrentHashMap的优化为什么不用B+树？

红黑树不是严格意义上的平衡二叉树，它是平衡二叉树的一个变体。他不要求绝对的平衡，所以它的左右子树的高度差可能大于1。但对它进行平衡的代价更低，其平均统计性能要强于平衡二叉树，适用于插入和删除较多的场景。红黑树插入元素时最多进行2次旋转，删除元素时最多进行3次旋转，而平衡二叉树进行的旋转次数则是不确定的。

B+树是多叉树，而且每个节点存放了多个关键字，非常的矮胖。如果使用B+树的话，在数据量不是很多的情况下，数据都会“挤在”一个结点里面，这个时候就退化成链表了。相对于数据库来说，Java的HashMap数据量显然很小。

### ArrayList 的线程安全集合是什么，底层实现是怎样的，可以应用在哪些场景中？

ArrayList的线程安全集合是CopyOnWriteArrayList，它是**线程安全**的，可以被多个读写线程并发访问，但**不能被多个写线程同时访问**，每个写操作都需要先获取锁，它还**以原子方式支持一些复合操作**。它的内部也是一个数组，但这个**数组是被整体更新的，实现了读写分离**，写操作会复制一个新的集合，在新集合内添加或删除元素，修改完成后再将原集合的引用指向新集合。这样做的好处是可以高并发地进行读写操作而不需要加锁，因为当前集合不会添加任何元素。使用时注意尽量设置**容量初始值**，并且可以**使用批量添加或删除**，避免多次扩容，比如只增加一个元素却复制整个集合。

适合**读多写少**的场景，**单个添加时效率极低**。CopyOnWriteArrayList 是**弱一致性**的，并发包的集合都是这种机制，它会在安全的副本上遍历，集合修改与副本遍历没有任何关系，缺点是无法读取最新数据。这也是 CAP 理论中C和A的矛盾，即一致性与可用性的矛盾。

### 什么是阻塞队列？阻塞队列的实现原理是什么？如何使用阻塞队列来实现生产者-消费者模型？

阻塞队列就是支持阻塞插入和阻塞移除的队列。阻塞插入就是说当队列满时，队列会阻塞插入元素的线程，直到队列不满。阻塞移除是指在队列空时，队列会阻塞移除元素的线程，知道队列不空。

阻塞队列是通过**通知模式**来实现的，就是当生产者往满的队列里添加元素时会阻塞住生产者，当消费者消费了一个队列中的元素后，会通知生产者当前队列可用。具体来说是使用了显示锁的**Condition**来实现的，在一个Condition的条件队列上进行等待和通知。

以阻塞队列作为生产者消费者模型中的缓冲区，然后生产者消费者使用阻塞队列的阻塞插入和阻塞移除方法put/take来生产和消费产品即可。

### Java中常见的阻塞队列有哪些，都有什么特点？

JDK提供了**7个**阻塞队列：

**ArrayBlockingQueue**，由**数组**结构组成的**有界**阻塞队列，默认情况下**不保证线程公平**，有可能先阻塞的线程最后才访问队列。

**LinkedBlockingQueue**，由**链表**结构组成的**有界**阻塞队列，队列的**默认和最大长度为 Integer 最大值**。**依照先进先出原则对元素进行排序**。

**PriorityBlockingQueue**，支持**优先级**的**无界**阻塞队列，默认情况下元素按照自然顺序升序**排序**。也可自定义 compareTo 方法指定排序规则，或者初始化时指定 Comparator 来对元素进行排序，**不能保证同优先级元素的顺序**。

**DelayQueue**，支持**延时获取元素**的**无界**阻塞队列，使用PriorityQueue实现。创建元素时**可以指定多久才能从队列中获取当前元素**，只有**延迟期满时才能从队列中获取元素**，适用于**缓存和定时调度**。

**SynchronousQueue**，是一个**不存储元素**的阻塞队列，**每一个 put 必须等待一个 take操作，否则不能继续添加元素**。**默认使用非公平策略，也支持公平策略**，**适用于传递性场景，吞吐量高。**

**LinkedTransferQueue**，**链表**组成的**无界**阻塞队列，相对于其他阻塞队列多了 tryTransfer 和 transfer 方法。**transfer**方法：如果当前有消费者正等待接收元素，可以把生产者传入的元素立刻传输给消费者，否则会**将元素放在队列的尾节点并等到该元素被消费者消费才返回**。**tryTransfer** 方法用来试探生产者传入的元素能否直接传给消费者，**如果没有消费者等待接收元素则返回 false**，和 transfer 的区别是无论消费者是否消费都会立即返回。

**LinkedBlockingDeque**，**链表**组成的**双向**阻塞队列，可从队列的两端插入和移出元素，**多线程同时入队时减少了竞争**。

### ConcurrentLinkedQueue是什么，底层实现是怎样的？

ConcurrentLinkedQueue是一个链表结构组成的无界线程安全队列。实现一个线程安全的队列有两种方式，一种是阻塞方式，它使用一个锁管理出队或入队，或者两个锁分别管理出队和入队来实现。另一种是非阻塞方式，ConcurrentLinkedQueue就是一个以非阻塞方式——循环CAS实现的线程安全队列。

**入队**：

整个入队过程主要做三件事情，第一是**定位出尾节点**；第二是**使用CAS算法将入队节点设置成尾节点的next节点**，如不成功则重新定位尾节点再重试；第三则是更新tail节点。ConcurrentLinkedQueue由head节点和tail节点组成，为了减少CAS更新tail节点的次数，tail节点并不总是尾节点，并不是每次节点入队都将tail节点更新成尾节点，而是当tail节点和尾节点的距离大于等于常量HOPS的值（默认是1）时才更新tail节点，tail节点和尾节点的距离越长，使用CAS更新tail节点的次数就会越少，但是每次入队时定位尾节点的时间就越长，但是这样仍然能够提高入队的效率，因为从本质上来看它通过对volatile变量的读操作来减少对volatile变量的写操作，而对volatile变量的写操作开销要远远大于读操作，所以入队效率会有所提升。

**出队**：

整个出队过程也是主要做三件事情，第一是定位出头结点；第二是使用CAS的方式将头结点的引用设置成null，如果设置成功，则直接返回头节点元素，如果不成功，则重新定位头节点再重试；第三是更新head节点。类似地，为了减少CAS更新head节点的次数，head节点也并不总是头节点。并不是每次出队时都更新head节点，而是当head节点和头节点的距离大于等于常量HOPS（默认是1）时才更新head节点。

### 什么是Fork/Join框架，实现原理是什么？

Fork/Join框架是一个用于并行执行任务的框架，是一个把大任务分割成若干个小任务，最终汇总每个小任务结果后得到大任务结果的框架。

在JDK中，ForkJoinTask是ForkJoin任务，它提供在任务中执行fork()和join()操作的机制。而ForkJoinTask需要通过ForkJoinPool来执行。ForkJoinPool由ForkJoinTask数组和ForkJoinWorkerThread数组组成，ForkJoinTask数组负责存放程序提交给ForkJoinPool的任务，而ForkJoinWorkerThread数组负责执行这些任务。

**ForkJoinTask的fork方法实现原理**：

当调用ForkJoinTask的fork方法时，程序会调用ForkJoinWorkerThread的pushTask方法异步地执行这个任务，然后立即返回结果。pushTask方法会把当前任务存放在ForkJoinTask数组里，然后再调用ForkJoinPool的signalWork方法唤醒或创建一个工作线程来执行任务。

**ForkJoinTask的join方法实现原理**：

Join方法的主要作用是阻塞当前线程并等待获取结果。首先，它调用了doJoin方法，通过doJoin方法得到当前任务的状态来判断返回什么结果，任务状态有4种：已完成（NORMAL）、被取消（CANCELLED）、信号（SIGNAL）和出现异常（EXCEPTIONAL）。如果任务状态是已完成，则直接返回任务结果，如果是被取消，则直接抛出CancellationException，如果是出现异常，则直接抛出对应的异常。

而在doJoin方法里，首先通过查看任务的状态，看任务是否已经执行完成，如果执行完成，则直接返回任务状态。如果没有执行完，则从任务数组里取出任务并执行。如果任务顺利执行完成，则设置任务状态为NORMAL，如果出现异常，则记录异常，并将任务状态设置为EXCEPTIONAL。

### 什么是工作窃取算法，它的优缺点是怎样的？

为了减少线程间的竞争，把很多子任务分别放到不同的队列里，并为每个队列创建一个单独的线程来执行队列里的任务，线程和队列一一对应。工作窃取算法是指某个线程在把自己队列里的任务执行完成之后，会从其它线程对应的队列里窃取任务来执行。为了减少窃取任务线程和被窃取任务线程之间的竞争，通常会使用双端队列，被窃取任务线程永远从双端队列的头部拿任务执行，而窃取任务线程则永远从双端队列的尾部拿任务执行。

**优点**：

可以充分利用线程进行并行计算，减少了线程间的竞争。

**缺点**：

需要创建多个线程和多个双端队列，会消耗更多的系统资源。在某些情况下还是存在竞争，比如双端队列里只有一个任务时。

## 6 原子类

### 有哪些原子类？

Atomic包里一共提供了12个类来对应**4种类型的原子更新方式**，分别是**原子更新基本类型、原子更新数组、原子更新引用和原子更新字段**。Atomic包里的类基本都是使用**Unsafe**实现的包装类。原子更新基本类型的类有AtomicInteger、AtomicBoolean、AtomicLong三个。其中Unsafe类只提供了compareAndSwapObject、compareAndSwapInt、compareAndSwapLong三个方法。所以AtomicBoolean是先把boolean类型转换成整型，再使用compareAndSwapInt进行CAS来实现的。类似地，如果要实现原子更新char、float、double变量也可以用类似的思路来完成。

原子更新数组的类有AtomicIntegerArray、AtomicLongArray、AtomicReferenceArray三个。主要是提供原子的方式更新数组里的元素。

原子更新引用的类有AtomicReference、AtomicStampedReference、AtomicMarkableReference三个。其中AtomicStampedReference用来原子更新带有整型值版本号的引用类型，可以解决CAS的ABA问题。而AtomicMarkableReference用来原子更新带有布尔类型标记位的引用类型，本质就是它的版本号只有两个，true和false，修改的时候在这两个版本号之间来回切换，这样做并不能解决ABA的问题，只是会降低ABA问题发生的几率而已。

原子更新字段的类有AtomicIntegerFieldUpdater、AtomicLongFieldUpdater、AtomicReferenceFieldUpdater三个。

### 原子类实现原子更新的原理是什么？

通过CAS操作来实现，JDK5之后Java类库中才开始使用CAS操作，该操作由**Unsafe类**的compareAndSwapObject、compareAndSwapInt、compareAndSwapLong这几个方法包装提供。虚拟机在内部对这些方法做了特殊处理，**即时编译出来的结果就是一条平台相关的处理器CAS指令**。Unsafe类不是提供给用户程序调用的类，因此在JDK9之前只有Java类库可以使用CAS，直到JDK9之后，Java类库才在VarHandle类里开放了面向用户程序的CAS操作。

## 7 Java中的并发工具类

### CountDownLatch 是什么？

CountDownLatch 是基于执行时间的同步类，允许一个或多个线程等待其他线程完成操作，它可以实现join的功能，并且比join的功能更多。它的构造方法接收一个 int 参数作为计数器，如果要等待 n 个点就传入 n。每次调用CountDownLatch的 countDown 方法时计数器减 1，调用CountDownLatch 的await 方法会阻塞当前线程直到计数器变为0，由于 countDown 方法可用在任何地方，所以 n 个点既可以是 n 个线程也可以是一个线程里的 n 个执行步骤。用在多个线程时，只需要把这个CountDownLatch的引用传递到线程里即可。还有就是一个线程调用countDown方法是先行发生（happens-before）于另一个线程调用await方法的，所以不会出现空等的现象。

### CyclicBarrier 是什么？

循环屏障是基于同步到达某个点的信号量触发机制，作用是让一组线程到达一个屏障时被阻塞，直到最后一个线程到达屏障才会解除。构造方法中的参数表示拦截线程数量，每个线程调用 await 方法告诉 CyclicBarrier 自己已到达屏障，然后被阻塞。还支持在构造方法中传入一个 Runnable 任务，当线程到达屏障时会优先执行该任务。适用于多线程计算数据，最后合并计算结果的应用场景。

### 在Java中CyclicBarriar和CountDownLatch有什么区别？

首先是**是否支持重用**。CountDownLacth 的计数器只能用一次，而 CyclicBarrier 的**计数器可使用 reset 方法重置**，所以 CyclicBarrier 能处理更为复杂的业务场景，例如计算错误时可以重置计数器并让线程重新执行一次。

然后是**应用场景**不一样。CountDownLatch一般用于某个线程等待若干个其他线程执行完任务之后，它才执行。而CyclicBarrier一般用于一组线程互相等待至某个状态，然后这一组线程再同时执行，当然在某些情况下它们可以互换。

### Semaphore 是什么？

信号量用来**控制同时访问特定资源的线程数量**，通过协调各个线程以保证合理使用公共资源。信号量可以用于**流量控制**，特别是**公共资源有限的应用场景**，比如数据库连接。

Semaphore 的构造方法接收一个 int 类型的参数，表示可用的**许可证数量即最大并发数**。使用 acquire 方法获得一个许可证，使用 release 方法归还许可证，还可以用 tryAcquire 尝试获得许可证。

### Exchanger 是什么？

Exchanger是用于线程间协作的工具类，用于进行**线程间的数据交换**。它提供一个同步点，**在这个同步点两个线程可以交换彼此的数据**。

两个线程通过 exchange 方法交换数据，**第一个线程执行 exchange 方法后会阻塞等待第二个线程执行该方法**，当两个线程都到达同步点时这两个线程就可以交换数据，将本线程生产出的数据传递给对方。应用场景包括**遗传算法、校对工作**等。

## 8 线程池

### 使用线程池有什么好处？

1. 它可以重用线程，避免线程创建和销毁带来的开销。
2. 它可以提高响应速度，当任务到达时，任务可以不需要等到线程创建就能立即执行。
3. 它可以控制并发线程数。任务过多时，通过排队避免创建过多线程，减少系统资源消耗和竞争，确保任务有序完成。
4. 它可以隔离线程环境，可以配置独立线程池，将较慢的线程与较快的线程隔离开，避免相互影响。
5. 它可以提高线程的可管理性，对线程进行统一分配、调优和监控，提供定时执行、定期执行、并发线程数量自适应等功能，定义任务队列的缓冲策略和拒绝策略。

### 创建线程池的7大参数是什么？

1） corePoolSize：常驻核心线程数，当线程池中的线程数大于corePoolSize时，keepAliveTime为多余的空闲线程等待新任务的最长时间，超过这个时间后多余的线程将被终止。而当线程池中的线程数小于等于corePoolSize时，核心线程不会被销毁。该值设置过大会浪费资源，过小会导致线程的频繁创建与销毁。

2）maximumPoolSize：线程池能够容纳同时执行的最大线程数，必须大于等于 1，如果与核心线程数设置相同则代表是固定大小的线程池。

3）keepAliveTime：线程空闲时间，当线程池中的线程数大于corePoolSize时，keepAliveTime为多余的空闲线程等待新任务的最长时间，超过这个时间后多余的线程将被终止，避免浪费内存资源。

④ unit：keepAliveTime 的时间单位。

⑤ workQueue：工作队列，当线程请求数大于等于 corePoolSize 时线程会进入阻塞队列。

⑥ threadFactory：线程工厂，用来生产一组执行相同任务的线程。可以给线程命名，有利于分析错误。

⑦ handler：拒绝策略，默认使用 AbortPolicy 丢弃任务并抛出异常，CallerRunsPolicy 表示使用调用者所在线程重新尝试运行该任务，DiscardOldestPolicy 表示抛弃队列里等待最久的任务并把当前任务加入队列，DiscardPolicy 表示直接抛弃当前任务但不抛出异常。

### 有哪些创建线程池的方法？

可以通过ThreadPoolExecutor构造方法来创建一个线程池。

也可以通过 Executors 的静态工厂方法创建线程池：

① **newFixedThreadPool**，固定大小的线程池，核心线程数等于最大线程数，不存在空闲线程。该线程池使用的工作队列是**无界阻塞队列** **LinkedBlockingQueue**，适用于**负载较重的服务器**。

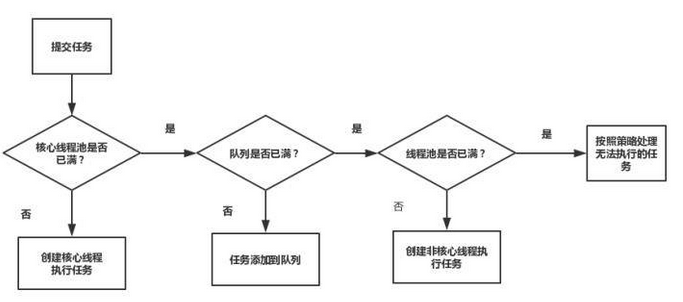
② **newSingleThreadExecutor**，使用单线程，相当于单线程串行执行所有任务，适用于需要保证顺序执行任务的场景。

③ **newCachedThreadPool**，maximumPoolSize 设置为 Integer 最大值，是高度可伸缩的线程池。该线程池使用的工作队列是没有容量的 **SynchronousQueue**，**如果主线程提交任务的速度高于线程处理的速度，线程池会不断创建新线程**，极端情况下会创建过多线程而耗尽CPU 和内存资源。适用于**执行很多短期异步任务的小程序**或**负载较轻的服务器**。

④ **newScheduledThreadPool**：它使用的工作队列是支持延时获取元素的无界阻塞队列DelayQueue，队列的最大长度为 Integer的最大值，存在 OOM 风险。支持定期及周期性任务执行，适用于需要多个后台线程执行周期任务，同时需要限制线程数量的场景。相比 Timer 更安全，功能更强，与 newCachedThreadPool 的区别是不回收工作线程。

⑤ **newWorkStealingPool**：JDK8新增，根据给定的并行度来动态创建和关闭线程，通过使用多个队列减少竞争，底层使用ForkJoinPool来实现。优势在于可以充分利用多CPU，把一个任务拆分成多个“小任务”，放到多个处理器核心上并行执行；当多个“小任务”执行完成之后，再将这些执行结果合并起来即可。而且多个队列间可以进行工作窃取。

### 线程池的工作流程是怎样的？



提交任务时，首先判断核心线程池是否已满，如果没有则创建核心线程执行任务，如果已满则判断工作队列是否已满，如果没有，将任务添加到工作队列，如果工作队列已满，则判断线程池是否已满，如果没有，则创建非核心线程执行任务，如果线程池已满，则按照拒绝策略处理。

### 线程池的拒绝策略是什么，如果让你自己定义一个拒绝策略你会怎样做？

拒绝策略又叫饱和策略，当队列和线程池都满了，说明线程池处于饱和状态，那么必须采取一种策略来处理提交的新任务。这个策略默认是AbortPolicy，表示丢弃新提交的任务并抛出异常。JDK1.5中Java线程池框架提供了4种策略，分别是：

1. AbortPolicy：丢弃新提交的任务并抛出异常。
2. CallerRunsPolicy：只使用调用者所在线程重新尝试运行该任务。
3. DiscardOldestPolicy：丢弃队列里等待最久的任务并把新提交的任务加入队列。
4. DiscardPolicy：丢弃新提交的任务，但并不抛出异常。

**自定义拒绝策略思路**：

可以根据任务的重要程度和任务的依赖顺序等来进行拒绝。

### 线程池的核心线程数量、最大线程数量如何设置？

通过corePoolSize参数和maximumPoolSize参数进行设置。将两个参数设置为一样表示创建固定大小的线程池。对于使用无界阻塞队列作为工作队列的线程池来说，最大线程数量没有意义。

### 自带的线程池为什么不建议用？

因为使用Excutors的一系列静态方法创建线程池的时候，其内部会设置一些默认的参数，这可能导致创建的线程池有隐患。比如使用newFixedThreadPool方法创建线程池，其使用的工作队列是LinkedBlockingQueue，它的默认长度也是最大长度为Integer.MAX\_VALUE，可能导致工作队列中的任务太多，堆积满了内存，从而造成内存溢出。建议使用ThreadPoolExcutor的构造函数来创建线程池，可以自定义各个参数。而且工作队列也建议使用有界队列而不是无界队列。也可以自定义自己的拒绝策略。

### 如何关闭线程池？

可以调用线程池的shutdown或shutdownNow方法来关闭线程池。它们的原理是遍历线程池中的工作线程，然后**逐个调用线程的interrupt方法**来中断线程，所以无法响应中断的任务可能永远无法终止。它们的区别是，**showdownNow**会首先将线程池的状态设置成STOP，然后尝试**停止所有的线程**（包括执行任务的和没有执行任务的），并返回正在执行任务的线程列表。而**shutdown**只是将线程池的状态设置成SHUTDOWN状态，然后**中断所有没有执行任务的线程**。

### 线程池的选择策略有什么？

选择线程池或者说合理地配置线程池，首先必须**分析任务的特性**，可以从以下几个角度来分析：

1. 任务的性质：CPU密集型任务、IO密集型任务和混合型任务
2. 任务的优先级：高、中和低。
3. 任务的执行时间：长、中和短。
4. 任务的依赖性：是否依赖其他系统资源，如数据库连接。

**性质不同的任务**可以用不同规模的线程池分开处理。**CPU密集型任务**应配置尽可能少的线程，如配置CPU数量+1个线程的线程池。而由于执行**IO密集型任务**的线程并不是一直在执行任务，则应配置尽可能多的线程，如2\*CPU数量。对于**混合型任务**，如果可以拆分，应将其拆分成一个CPU密集型任务和一个IO密集型任务，只要这两个任务执行的时间相差不是太大，那么分解后执行的吞吐量将高于串行执行的吞吐量。如果这两个任务执行时间相差太大，则没必要进行分解。

**优先级不同的任务**可以使用**优先级队列PriorityBlockingQueue**来处理。它可以让优先级高的任务先执行。

**执行时间不同的任务**可以**交给不同规模的线程池**来管理，或者可以使用优先级队列，让执行时间短的任务先执行。

**依赖数据库连接池的任务**，因为线程提交SQL后需要等待数据库返回结果，等待的时间越长，则CPU空闲时间就越长，那么**线程数量应该设置得更大**，这样才能更好地利用CPU。

其次就是**建议使用有界队列**。有界队列能增加系统的**稳定性和预警能力**，可以根据需要设置得大一点。

### 阻塞队列有哪些选择?

一般可以选择ArrayBlockingQueue、LinkedBlockingQueue、SynchronousQueue、PriorityBlockingQueue四种阻塞队列。**ArrayBlockingQueue**是一个数组结构组成的有界阻塞队列，此队列按照FIFO原则对元素进行排序。**LinkedBlockingQueue**是一个链表结构组成的有界阻塞队列，也是按FIFO对元素进行排序，其吞吐量通常大于ArrayBlockingQueue，Excutors.newFixedThreadPool使用了这个队列。**SynchronousQueue**是一个不存储元素的阻塞队列，每个插入操作必须等到另一个线程调用移除操作，否则插入操作一直处于阻塞状态，吞吐量通常高于LinkedBlockingQueue，Excutors.newCachedThreadPool使用了这个队列。**PriorityBlockingQueue**是一个支持优先级的无界阻塞队列，可以自定义排序规则，它不能保证同优先级元素的顺序。此外，在Excutors. newScheduledThreadPool方法创建的线程池中还使用了**DelayQueue**，它是一个支持延时获取元素的无界阻塞队列。

### 可以使用哪两种方法向线程池提交任务，有什么不同？

可以使用execute和submit两个方法向线程池提交任务。execute方法用于提交不需要返回值的任务，所以无法判断任务是否被线程池执行成功。execute方法输入的任务是一个Runnable的实现类实例。Submit方法用于提交需要返回值的任务，线程池会返回一个Future类型的对象，通过这个Future对象可以判断任务是否执行成功，并且可以通过Future的get方法来获取返回值，get方法会阻塞当前线程直到任务完成。Submit方法输入的任务是一个Runnable或Callable的实现类实例。

## 9 Excutor框架

### Executor框架是什么，为什么要引入Executor框架？

在Java中，使用线程来异步执行任务。Java的线程既是任务载体，也是执行机制。从JDK5开始，Java把任务载体和执行机制分离开来。任务载体包括Runnable和Callable，而执行机制由Executor框架提供。也就是说，Executor框架就是给异步任务提供执行机制的一个框架。

之所以要引入Executor框架是因为如果将任务载体和执行机制绑定在一块，直接由用户程序来创建和管理线程有很多缺点。这些缺点包括：

1）频繁的线程创建、销毁造成的性能损耗。

2）创建过多的线程引起激烈的资源竞争而导致系统性能急剧下降甚至瘫痪。

3）不利于扩展，比如定时执行、定期执行、线程中断等都不便实现。

而Executor框架可以将异步任务处理的任务载体和执行机制分离开，提供了线程池的执行机制，可以对异步任务并发处理中的多个线程的创建、销毁、数量等进行统一管理，也可以对异步任务进行按时间定制的执行，包括延迟执行、周期执行等。

### 在Java中Executor和Executors的区别？

Executor是一个提供异步任务执行机制的执行器接口，它常见的实现类有ThreadPoolExecutor、ScheduledThreadPoolExecutor两种。而Executors是Executor框架中的一个工具类，它提供了Executor框架中常用到的一些功能。比如使用它可以方便快捷地获得一个具体的Executor执行器，还可以对任务载体Runnable和Callable进行转化等。

### 什么是Callable和Future?

Callable和Runnable一样是一个任务载体，与Runnable不同的是它在设计出来时就是将异步任务执行中的任务载体和执行机制分离开的，所以它不能由Thread类直接运行，而只能由提供执行机制的具体的执行器来执行。Callable作为任务载体也与Runnable有所不同，它表示带有返回值的任务，而Runnable表示没有返回值的任务。

Future代表一个Callable这种带有返回值的可取消的异步任务，采用了任务载体和执行机制分离的设计原则，只能由提供执行机制的具体的Executor来执行，提供了获取异步任务结果、取消任务、查询任务状态等方法。它是一个接口，支持泛型。它的一个具体的实现类是FutureTask，使用FutureTask的get方法可以阻塞或超时阻塞地获取到异步任务的结果，也可以使用其cancel方法来取消此任务的执行。

## 10 其他JUC

### 使用过哪些Java并发包？

## 11 其他

### 什么是ThreadLocal，使用它可能产生什么问题，怎样避免？

ThreadLocal是本地线程变量，它可以存储任意对象的值。它的用途有：

1. 线程数据隔离。
2. 一个线程内跨类、方法传递数据，减少了参数传递开销。
3. 用于存储线程事务信息。
4. 对数据库连接、Session进行管理。

它会为每个线程创建单独的副本，副本值是线程私有的，互相之间不会产生影响。每个Thread对象中都有一个ThreadLocalMap对象，这个对象存储了一组以ThreadLocal.threadLocalHashCode为键，以本地线程变量为值的键值对。每一个ThreadLocal对象都包含了一个独一无二的threadLocalHashCode值，使用这个值就可以在ThreadLocalMap中找到对应的本地线程变量。

可能产生**读脏数据**的问题，由于线程池会重用Thread对象，因此与 Thread 绑定的 ThreadLocal也会被重用。如果没有调用ThreadLocal的remove清理与线程相关的 ThreadLocal 信息而且重用后的线程没有设置初始值，那么就会读到以前的脏数据。

还存在**内存泄漏**的问题，由于 ThreadLocalMap中Entry的key字段对ThreadLocal 的引用是弱引用，因此当ThreadLocal没有被程序使用后，其就将被回收。但 Entry 的 value 字段对保存的本地线程变量对象是强引用，因此当 ThreadLocal 被垃圾回收后，value对象依旧不会被释放。解决办法是当ThreadLocal使用完毕后，需要及时调用其 remove 方法进行清理value对象的操作。

### 什么是不可变对象，它对写并发应用有什么帮助？

不可变对象是指对象被创建之后，其内部状态和属性在其生命周期内保持不变的对象。常见的有String、枚举类型、基本数据类型包装类、BigInteger等大数据类型和Java8新出的日期对象LocalDate等。不可变对象一定是线程安全的，无论是对象的方法实现还是方法的调用者都不需要再进行任何的线程安全保障，不需要进行同步，可以极大地保证写并发的便捷性和高效性。

### 什么是阻塞式方法？

阻塞式方法是指程序在调用方法后，必须等待方法完成返回结果，其间一直被阻塞。比如ServerSocket的accept方法就是典型的阻塞方法。

### 什么是Java Timer 类？如何创建一个有特定时间间隔的任务？

Java.util.Timer 是一个工具类，可以用于安排一个任务在未来的某个特定时间执行。Timer 类可以用于安排一次性任务或周期性任务。它会开启一个新的线程去执行计划任务，默认不是守护线程，可以通过Timer构造函数的isDaemon参数将其设置成守护线程。java.util.TimerTask 是一个实现了 Runnable 接口的抽象类，我们需要去继承这个类来创建我们自己的定时任务并使用 Timer 去安排它的执行。

### 什么是竞态条件？你怎样发现和解决竞争？

当某个计算的正确性取决于多个线程的交替执行时序时，那么就说该计算存在竞态条件。最常见的竞态条件类型就是”先检查后执行“操作，即通过一个可能失效的观测结果来决定下一步的动作。还有依赖当前值的修改操作，可能导致在一个失效的值上进行修改，从而出错。

可以通过对依赖多个线程的交替执行时序的操作进行同步来避免竞态条件的产生，Java中常见的手段包括synchronized、显示锁等。

### 什么叫线程安全？Servlet是线程安全的吗?

当多个线程同时访问一个对象时，如果不考虑这些线程在运行时环境下的交替执行，也不需要进行额外的同步，或者在调用方进行任何其它的协调操作，调用这个对象的行为都可以获得正确的结果，那就称这个对象是线程安全的。（串行执行与并发执行都能得到正确的结果）

如果Servlet是无状态的，也就是没有字段，也没有引用其他对象的字段，那么它是线程安全的。大多数Servlet都是无状态的，只有当Servlet在处理请求时需要保存一些信息，这时候才需要使用额外的同步手段来保证Servlet的线程安全性。

### 你如何在Java中获取线程堆栈？

可以使用Jstack或JvisualVM工具。

### JVM中哪个参数是用来控制线程的堆和虚拟机栈的大小的?

-Xms:堆初始大小

-Xmx:堆最大大小

-Xmn:新生代大小（Sun官方推荐配置为整个堆的3/8）。

-Xss:每个线程的虚拟机栈大小（在相同物理内存下，减小这个值能生成更多的线程）。