# 1、在java中守护线程和本地线程区别？

java中的线程分为两种：守护线程（Daemon）和用户线程（User）。

任何线程都可以设置为守护线程和用户线程，通过方法Thread.setDaemon(bool on)；true则把该线程设置为守护线程，反之则为用户线程。Thread.setDaemon()必须在Thread.start()之前调用，否则运行时会抛出异常。

两者的区别：

虚拟机（JVM）何时离开，Daemon是为其他线程提供服务，如果全部的User Thread已经撤离，Daemon 没有可服务的线程，JVM撤离。也可以理解为守护线程是JVM自动创建的线程（但不一定），用户线程是程序创建的线程；比如JVM的垃圾回收线程是一个守护线程，当所有线程已经撤离，不再产生垃圾，守护线程自然就没事可干了，当垃圾回收线程是Java虚拟机上仅剩的线程时，Java虚拟机会自动离开。

扩展：Thread Dump打印出来的线程信息，含有daemon字样的线程即为守护进程，可能会有：服务守护进程、编译守护进程、windows下的监听Ctrl+break的守护进程、Finalizer守护进程、引用处理守护进程、GC守护进程。

# 2、线程与进程的区别？

进程是操作系统分配资源的最小单元，线程是操作系统调度的最小单元。

一个程序至少有一个进程,一个进程至少有一个线程。

# 3、什么是多线程中的上下文切换？

多线程会共同使用一组计算机上的CPU，而线程数大于给程序分配的CPU数量时，为了让各个线程都有执行的机会，就需要轮转使用CPU。不同的线程切换使用CPU发生的切换数据等就是上下文切换。

# 4、死锁与活锁的区别，死锁与饥饿的区别？

死锁：是指两个或两个以上的进程（或线程）在执行过程中，因争夺资源而造成的一种互相等待的现象，若无外力作用，它们都将无法推进下去。

产生死锁的必要条件：

互斥条件：所谓互斥就是进程在某一时间内独占资源。

请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。

不剥夺条件:进程已获得资源，在末使用完之前，不能强行剥夺。

循环等待条件:若干进程之间形成一种头尾相接的循环等待资源关系。

活锁：任务或者执行者没有被阻塞，由于某些条件没有满足，导致一直重复尝试、失败、尝试、失败。

活锁和死锁的区别在于，处于活锁的实体是在不断的改变状态，所谓的“活”， 而处于死锁的实体表现为等待；活锁有可能自行解开，死锁则不能。

饥饿：一个或者多个线程因为种种原因无法获得所需要的资源，导致一直无法执行的状态。

Java中导致饥饿的原因：

高优先级线程吞噬所有的低优先级线程的CPU时间。

线程被永久堵塞在一个等待进入同步块的状态，因为其他线程总是能在它之前持续地对该同步块进行访问。

线程在等待一个本身也处于永久等待完成的对象(比如调用这个对象的wait方法)，因为其他线程总是被持续地获得唤醒。

# 5、Java中用到的线程调度算法是什么？

采用时间片轮转的方式。可以设置线程的优先级，会映射到下层的系统上面的优先级上，如非特别需要，尽量不要用，防止线程饥饿。

# 6、什么是线程组，为什么在Java中不推荐使用？

ThreadGroup类，可以把线程归属到某一个线程组中，线程组中可以有线程对象，也可以有线程组，组中还可以有线程，这样的组织结构有点类似于树的形式。

为什么不推荐使用？因为有很多的安全隐患吧，如果需要使用，推荐使用线程池。

# 7、为什么使用Executor框架？

每次执行任务创建线程 new Thread()比较消耗性能，创建一个线程是比较耗时、耗资源的。

调用 new Thread()创建的线程缺乏管理，被称为野线程，而且可以无限制的创建，线程之间的相互竞争会导致过多占用系统资源而导致系统瘫痪，还有线程之间的频繁交替也会消耗很多系统资源。

接使用new Thread() 启动的线程不利于扩展，比如定时执行、定期执行、定时定期执行、线程中断等都不便实现。

# 8、在Java中Executor和Executors的区别？

Executors 工具类的不同方法按照我们的需求创建了不同的线程池，来满足业务的需求。

Executor 接口对象能执行我们的线程任务。

ExecutorService接口继承了Executor接口并进行了扩展，提供了更多的方法我们能获得任务执行的状态并且可以获取任务的返回值。

使用ThreadPoolExecutor 可以创建自定义线程池。

Future 表示异步计算的结果，他提供了检查计算是否完成的方法，以等待计算的完成，并可以使用get()方法获取计算的结果。

# 9、什么是原子操作？

原子操作是指一个不受其他操作影响的操作任务单元。原子操作是在多线程环境下避免数据不一致必须的手段。

处理器使用基于对缓存加锁或总线加锁的方式来实现多处理器之间的原子操作。

在Java中可以通过锁和循环CAS的方式来实现原子操作。 CAS操作——Compare & Set，或是 Compare & Swap，现在几乎所有的CPU指令都支持CAS的原子操作。

int++并不是一个原子操作，所以当一个线程读取它的值并加1时，另外一个线程有可能会读到之前的值，这就会引发错误。

为了解决这个问题，必须保证增加操作是原子的，在JDK1.5之前我们可以使用同步技术来做到这一点。到JDK1.5，java.util.concurrent.atomic包提供了int和long类型的原子包装类，它们可以自动的保证对于他们的操作是原子的并且不需要使用同步。

java.util.concurrent这个包里面提供了一组原子类。其基本的特性就是在多线程环境下，当有多个线程同时执行这些类的实例包含的方法时，具有排他性，即当某个线程进入方法，执行其中的指令时，不会被其他线程打断，而别的线程就像自旋锁一样，一直等到该方法执行完成，才由JVM从等待队列中选择一个另一个线程进入，这只是一种逻辑上的理解。

# 10、Java Concurrency API中的Lock接口是什么？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：

可以使锁更公平

可以使线程在等待锁的时候响应中断

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

可以在不同的范围，以不同的顺序获取和释放锁

整体上来说Lock是synchronized的扩展版，Lock提供了无条件的、可轮询的、定时的、可中断的、可多条件队列的锁操作。另外Lock的实现类基本都支持非公平锁和公平锁，synchronized只支持非公平锁，当然，在大部分情况下，非公平锁是高效的选择。

# 11、什么是Executors框架？

Executor框架是一个根据一组执行策略调用，调度，执行和控制的异步任务的框架。

无限制的创建线程会引起应用程序内存溢出。所以创建一个线程池是个更好的的解决方案，因为可以限制线程的数量并且可以回收再利用这些线程。利用Executors框架可以非常方便的创建一个线程池。

# 12、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？

阻塞队列是一个支持两个附加操作的队列。在队列为空时，获取元素的线程会等待队列变为非空。当队列满时，存储元素的线程会等待队列可用。

阻塞队列常用于生产者和消费者的场景，生产者是往队列里添加元素的线程，消费者是从队列里拿元素的线程。阻塞队列就是生产者存放元素的容器，而消费者也只从容器里拿元素。

JDK7提供了7个阻塞队列。分别是：

ArrayBlockingQueue ：一个由数组结构组成的有界阻塞队列。

LinkedBlockingQueue ：一个由链表结构组成的有界阻塞队列。

PriorityBlockingQueue ：一个支持优先级排序的无界阻塞队列。

DelayQueue：一个使用优先级队列实现的无界阻塞队列。

SynchronousQueue：一个不存储元素的阻塞队列。

LinkedTransferQueue：一个由链表结构组成的无界阻塞队列。

LinkedBlockingDeque：一个由链表结构组成的双向阻塞队列。

Java 5之前实现同步存取时，可以使用普通的一个集合，然后在使用线程的协作和线程同步可以实现生产者，消费者模式，主要的技术就是用好，wait 、notify、notifyAll、sychronized这些关键字。而在java 5之后，可以使用阻塞队列来实现，此方式大大简少了代码量，使得多线程编程更加容易，安全方面也有保障。

BlockingQueue接口是Queue的子接口，它的主要用途并不是作为容器，而是作为线程同步的的工具，因此他具有一个很明显的特性，当生产者线程试图向BlockingQueue放入元素时，如果队列已满，则线程被阻塞，当消费者线程试图从中取出一个元素时，如果队列为空，则该线程会被阻塞，正是因为它所具有这个特性，所以在程序中多个线程交替向BlockingQueue中放入元素，取出元素，它可以很好的控制线程之间的通信。

阻塞队列使用最经典的场景就是socket客户端数据的读取和解析，读取数据的线程不断将数据放入队列，然后解析线程不断从队列取数据解析。

# 13、什么是Callable和Future?

Callable接口类似于Runnable，但是Runnable不会返回结果，并且无法抛出返回结果的异常，而Callable功能更强大一些，被线程执行后，可以返回值，这个返回值可以被Future拿到，也就是说Future可以拿到异步执行任务的返回值。

可以认为是带有回调的Runnable。

Future接口表示异步任务，是还没有完成的任务给出的未来结果。所以说Callable用于产生结果，Future用于获取结果。

# 14、什么是FutureTask?

在Java并发程序中FutureTask表示一个可以取消的异步运算。它有启动和取消运算、查询运算是否完成和取回运算结果等方法。

只有当运算完成的时候结果才能取回，如果运算尚未完成get方法将会阻塞。一个FutureTask对象可以对调用了Callable和Runnable的对象进行包装，由于FutureTask也是调用了Runnable接口所以它可以提交给Executor来执行。

# 15、什么是并发容器的实现？

何为同步容器：可以简单地理解为通过synchronized来实现同步的容器，如果有多个线程调用同步容器的方法，它们将会串行执行。比如Vector、Hashtable、Collections.synchronizedSet、synchronizedList等方法返回的容器。

可以通过查看Vector，Hashtable等这些同步容器的实现代码，可以看到这些容器实现线程安全的方式就是将它们的状态封装起来，并在需要同步的方法上加上关键字synchronized。

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伸缩性，例如在ConcurrentHashMap中采用了一种粒度更细的加锁机制，可以称为分段锁，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的访问map，同时允许一定数量的写操作线程并发地修改map，所以它可以在并发环境下实现更高的吞吐量。

# 16、多线程同步和互斥有几种实现方法，都是什么？

线程同步是指线程之间所具有的一种制约关系，一个线程的执行依赖另一个线程的消息，当它没有得到另一个线程的消息时应等待，直到消息到达时才被唤醒。

线程互斥是指对于共享的进程系统资源，在各单个线程访问时的排它性。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。线程互斥可以看成是一种特殊的线程同步。

线程间的同步方法大体可分为两类：用户模式和内核模式。顾名思义，内核模式就是指利用系统内核对象的单一性来进行同步，使用时需要切换内核态与用户态，而用户模式就是不需要切换到内核态，只在用户态完成操作。

用户模式下的方法有：原子操作、临界区。内核模式下的方法有：事件、信号量、互斥量。

# 17、什么是竞争条件？你怎样发现和解决竞争？

当多个进程都企图对共享数据进行某种处理，而最后的结果又取决于进程运行的顺序时，则我们认为这发生了竞争条件。

# 18、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？

当你调用start()方法时你将创建新的线程，并且执行在run()方法里的代码。

但是如果你直接调用run()方法，它不会创建新的线程也不会执行调用线程的代码，只会把run方法当作普通方法去执行。

# 19、Java中你怎样唤醒一个阻塞的线程？

在Java发展史上曾经使用suspend()、resume()方法对于线程进行阻塞唤醒，但随之出现很多问题，比较典型的还是死锁问题。

解决方案可以使用以对象为目标的阻塞，即利用Object类的wait()和notify()方法实现线程阻塞。

首先，wait、notify方法是针对对象的，调用任意对象的wait()方法都将导致线程阻塞，阻塞的同时也将释放该对象的锁，相应地，调用任意对象的notify()方法则将随机解除该对象阻塞的线程，但它需要重新获取改对象的锁，直到获取成功才能往下执行；其次，wait、notify方法必须在synchronized块或方法中被调用，并且要保证同步块或方法的锁对象与调用wait、notify方法的对象是同一个，如此一来在调用wait之前当前线程就已经成功获取某对象的锁，执行wait阻塞后当前线程就将之前获取的对象锁释放。

# 20、CycliBarriar和CountdownLatch有什么区别？

CyclicBarrier可以重复使用，而CountdownLatch不能重复使用。

Java的concurrent包里面的CountDownLatch其实可以把它看作一个计数器，只不过这个计数器的操作是原子操作，同时只能有一个线程去操作这个计数器，也就是同时只能有一个线程去减这个计数器里面的值。

你可以向CountDownLatch对象设置一个初始的数字作为计数值，任何调用这个对象上的await()方法都会阻塞，直到这个计数器的计数值被其他的线程减为0为止。

所以在当前计数到达零之前，await 方法会一直受阻塞。之后，会释放所有等待的线程，await的所有后续调用都将立即返回。这种现象只出现一次——计数无法被重置。如果需要重置计数，请考虑使用 CyclicBarrier。

CountDownLatch的一个非常典型的应用场景是：有一个任务想要往下执行，但必须要等到其他的任务执行完毕后才可以继续往下执行。假如我们这个想要继续往下执行的任务调用一个CountDownLatch对象的await()方法，其他的任务执行完自己的任务后调用同一个CountDownLatch对象上的countDown()方法，这个调用await()方法的任务将一直阻塞等待，直到这个CountDownLatch对象的计数值减到0为止

CyclicBarrier一个同步辅助类，它允许一组线程互相等待，直到到达某个公共屏障点 。在涉及一组固定大小的线程的程序中，这些线程必须不时地互相等待，此时 CyclicBarrier 很有用。因为该 barrier 在释放等待线程后可以重用，所以称它为循环 的 barrier。

# 21、什么是不可变对象，它对写并发应用有什么帮助？

不可变对象即对象一旦被创建它的状态就不能改变，反之即为可变对象。 不可变对象的类即为不可变类。Java平台类库中包含许多不可变类，如String、基本类型的包装类、BigInteger和BigDecimal等。

不可变对象天生是线程安全的。它们的常量（域）是在构造函数中创建的。既然它们的状态无法修改，这些常量永远不会变。

不可变对象永远是线程安全的。

只有满足如下状态，一个对象才是不可变的；

它的状态不能在创建后再被修改；

所有域都是final类型；并且，

它被正确创建（创建期间没有发生this引用的逸出）。

# 22、什么是多线程中的上下文切换？

在上下文切换过程中，CPU会停止处理当前运行的程序，并保存当前程序运行的具体位置以便之后继续运行。

从这个角度来看，上下文切换有点像我们同时阅读几本书，在来回切换书本的同时我们需要记住每本书当前读到的页码。在程序中，上下文切换过程中的“页码”信息是保存在进程控制块（PCB）中的。PCB还经常被称作“切换桢”，“页码”信息会一直保存到CPU的内存中，直到他们被再次使用。

上下文切换是存储和恢复CPU状态的过程，它使得线程执行能够从中断点恢复执行。上下文切换是多任务操作系统和多线程环境的基本特征。

# 23、Java中用到的线程调度算法是什么？

计算机通常只有一个CPU,在任意时刻只能执行一条机器指令,每个线程只有获得CPU的使用权才能执行指令.所谓多线程的并发运行,其实是指从宏观上看,各个线程轮流获得CPU的使用权,分别执行各自的任务.在运行池中,会有多个处于就绪状态的线程在等待CPU,JAVA虚拟机的一项任务就是负责线程的调度,线程调度是指按照特定机制为多个线程分配CPU的使用权.

有两种调度模型：分时调度模型和抢占式调度模型。

分时调度模型是指让所有的线程轮流获得cpu的使用权,并且平均分配每个线程占用的CPU的时间片这个也比较好理解。

java虚拟机采用抢占式调度模型，是指优先让可运行池中优先级高的线程占用CPU，如果可运行池中的线程优先级相同，那么就随机选择一个线程，使其占用CPU。处于运行状态的线程会一直运行，直至它不得不放弃CPU。

# 24、什么是线程组，为什么在Java中不推荐使用？

线程组和线程池是两个不同的概念，他们的作用完全不同，前者是为了方便线程的管理，后者是为了管理线程的生命周期，复用线程，减少创建销毁线程的开销。

# 25、为什么使用Executor框架比使用应用创建和管理线程好？

为什么要使用Executor线程池框架？

每次执行任务创建线程 new Thread()比较消耗性能，创建一个线程是比较耗时、耗资源的。

调用 new Thread()创建的线程缺乏管理，被称为野线程，而且可以无限制的创建，线程之间的相互竞争会导致过多占用系统资源而导致系统瘫痪，还有线程之间的频繁交替也会消耗很多系统资源。

直接使用new Thread() 启动的线程不利于扩展，比如定时执行、定期执行、定时定期执行、线程中断等都不便实现。

使用Executor线程池框架的优点：

能复用已存在并空闲的线程从而减少线程对象的创建从而减少了消亡线程的开销。

可有效控制最大并发线程数，提高系统资源使用率，同时避免过多资源竞争。

框架中已经有定时、定期、单线程、并发数控制等功能。

综上所述使用线程池框架Executor能更好的管理线程、提供系统资源使用率。

# 26、java中有几种方法可以实现一个线程？

继承 Thread 类

实现 Runnable 接口

实现 Callable 接口，需要实现的是 call() 方法

# 27、如何停止一个正在运行的线程？

使用共享变量的方式

在这种方式中，之所以引入共享变量，是因为该变量可以被多个执行相同任务的线程用来作为是否中断的信号，通知中断线程的执行。

使用interrupt方法终止线程

如果一个线程由于等待某些事件的发生而被阻塞，又该怎样停止该线程呢？这种情况经常会发生，比如当一个线程由于需要等候键盘输入而被阻塞，或者调用Thread.join()方法，或者Thread.sleep()方法，在网络中调用ServerSocket.accept()方法，或者调用了DatagramSocket.receive()方法时，都有可能导致线程阻塞，使线程处于处于不可运行状态时，即使主程序中将该线程的共享变量设置为true，但该线程此时根本无法检查循环标志，当然也就无法立即中断。这里我们给出的建议是，不要使用stop()方法，而是使用Thread提供的interrupt()方法，因为该方法虽然不会中断一个正在运行的线程，但是它可以使一个被阻塞的线程抛出一个中断异常，从而使线程提前结束阻塞状态，退出堵塞代码。

# 28、notify()和notifyAll()有什么区别？

当一个线程进入wait之后，就必须等其他线程notify/notifyall,使用notifyall,可以唤醒所有处于wait状态的线程，使其重新进入锁的争夺队列中，而notify只能唤醒一个。

如果没把握，建议notifyAll，防止notigy因为信号丢失而造成程序异常。

# 29、什么是Daemon线程？它有什么意义？

所谓后台(daemon)线程，是指在程序运行的时候在后台提供一种通用服务的线程，并且这个线程并不属于程序中不可或缺的部分。因此，当所有的非后台线程结束时，程序也就终止了，同时会杀死进程中的所有后台线程。

反过来说， 只要有任何非后台线程还在运行，程序就不会终止。必须在线程启动之前调用setDaemon()方法，才能把它设置为后台线程。注意：后台进程在不执行finally子句的情况下就会终止其run()方法。

比如：JVM的垃圾回收线程就是Daemon线程，Finalizer也是守护线程。

# 30、java如何实现多线程之间的通讯和协作？

中断和共享变量

# 31、什么是可重入锁？

举例来说明锁的可重入性：

public class UnReentrant{

   Lock lock = new Lock();

   public void outer(){

       lock.lock();

       inner();

       lock.unlock();

   }

   public void inner(){

       lock.lock();

       //do something

       lock.unlock();

   }

}

outer中调用了inner，outer先锁住了lock，这样inner就不能再获取lock。其实调用outer的线程已经获取了lock锁，但是不能在inner中重复利用已经获取的锁资源，这种锁即称之为 不可重入可重入就意味着：线程可以进入任何一个它已经拥有的锁所同步着的代码块。

synchronized、ReentrantLock都是可重入的锁，可重入锁相对来说简化了并发编程的开发。

# 32、当一个线程进入某个对象的一个synchronized的实例方法后，其它线程是否可进入此对象的其它方法？

如果其他方法没有synchronized的话，其他线程是可以进入的。

所以要开放一个线程安全的对象时，得保证每个方法都是线程安全的。

# 33、乐观锁和悲观锁的理解及如何实现？

悲观锁：总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。再比如Java里面的同步原语synchronized关键字的实现也是悲观锁。

乐观锁：每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制。乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库提供的类似于write\_condition机制，其实都是提供的乐观锁。在Java中java.util.concurrent.atomic包下面的原子变量类就是使用了乐观锁的一种实现方式CAS实现的。

# 34、SynchronizedMap 和 ConcurrentHashMap有什么区别？

SynchronizedMap一次锁住整张表来保证线程安全，所以每次只能有一个线程来访为map。

ConcurrentHashMap使用分段锁来保证在多线程下的性能。ConcurrentHashMap中则是一次锁住一个桶。ConcurrentHashMap默认将hash表分为16个桶，诸如get,put,remove等常用操作只锁当前需要用到的桶。这样，原来只能一个线程进入，现在却能同时有16个写线程执行，并发性能的提升是显而易见的。

另外ConcurrentHashMap使用了一种不同的迭代方式。在这种迭代方式中，当iterator被创建后集合再发生改变就不再是抛出ConcurrentModificationException，取而代之的是在改变时new新的数据从而不影响原有的数据 ，iterator完成后再将头指针替换为新的数据 ，这样iterator线程可以使用原来老的数据，而写线程也可以并发的完成改变。

# 35、CopyOnWriteArrayList可以用于什么应用场景？

CopyOnWriteArrayList(免锁容器)的好处之一是当多个迭代器同时遍历和修改这个列表时，不会抛出ConcurrentModificationException。在CopyOnWriteArrayList中，写入将导致创建整个底层数组的副本，而源数组将保留在原地，使得复制的数组在被修改时，读取操作可以安全地执行。

由于写操作的时候，需要拷贝数组，会消耗内存，如果原数组的内容比较多的情况下，可能导致young gc或者full gc；

不能用于实时读的场景，像拷贝数组、新增元素都需要时间，所以调用一个set操作后，读取到数据可能还是旧的,虽然CopyOnWriteArrayList 能做到最终一致性,但是还是没法满足实时性要求；

CopyOnWriteArrayList透露的思想

读写分离，读和写分开

最终一致性

使用另外开辟空间的思路，来解决并发冲突

# 36、什么叫线程安全？servlet是线程安全吗?

线程安全是编程中的术语，指某个函数、函数库在多线程环境中被调用时，能够正确地处理多个线程之间的共享变量，使程序功能正确完成。

Servlet不是线程安全的，servlet是单实例多线程的，当多个线程同时访问同一个方法，是不能保证共享变量的线程安全性的。

Struts2的action是多实例多线程的，是线程安全的，每个请求过来都会new一个新的action分配给这个请求，请求完成后销毁。

SpringMVC的Controller是线程安全的吗？不是的，和Servlet类似的处理流程

Struts2好处是不用考虑线程安全问题；Servlet和SpringMVC需要考虑线程安全问题，但是性能可以提升不用处理太多的gc，可以使用ThreadLocal来处理多线程的问题。

# 37、volatile有什么用？能否用一句话说明下volatile的应用场景？

volatile保证内存可见性和禁止指令重排。

volatile用于多线程环境下的单次操作(单次读或者单次写)。

# 38、为什么代码会重排序？

在执行程序时，为了提供性能，处理器和编译器常常会对指令进行重排序，但是不能随意重排序，不是你想怎么排序就怎么排序，它需要满足以下两个条件：

在单线程环境下不能改变程序运行的结果；

存在数据依赖关系的不允许重排序

需要注意的是：重排序不会影响单线程环境的执行结果，但是会破坏多线程的执行语义。

# 39、在java中wait和sleep方法的不同？

最大的不同是在等待时wait会释放锁，而sleep一直持有锁。Wait通常被用于线程间交互，sleep通常被用于暂停执行。

直接了解的深入一点吧：

在Java中线程的状态一共被分成6种：

初始态：

创建一个Thread对象，但还未调用start()启动线程时，线程处于初始态。

运行态：

在Java中，运行态包括就绪态和运行态。

就绪态该状态下的线程已经获得执行所需的所有资源，只要CPU分配执行权就能运行。所有就绪态的线程存放在就绪队列中。

运行态获得CPU执行权，正在执行的线程。由于一个CPU同一时刻只能执行一条线程，因此每个CPU每个时刻只有一条运行态的线程。

阻塞态：

当一条正在执行的线程请求某一资源失败时，就会进入阻塞态。而在Java中，阻塞态专指请求锁失败时进入的状态。由一个阻塞队列存放所有阻塞态的线程。处于阻塞态的线程会不断请求资源，一旦请求成功，就会进入就绪队列，等待执行。PS：锁、IO、Socket等都资源。

等待态：

当前线程中调用wait、join、park函数时，当前线程就会进入等待态。也有一个等待队列存放所有等待态的线程。线程处于等待态表示它需要等待其他线程的指示才能继续运行。进入等待态的线程会释放CPU执行权，并释放资源（如：锁）

超时等待态：

当运行中的线程调用sleep(time)、wait、join、parkNanos、parkUntil时，就会进入该状态；它和等待态一样，并不是因为请求不到资源，而是主动进入，并且进入后需要其他线程唤醒；进入该状态后释放CPU执行权 和 占有的资源。与等待态的区别：到了超时时间后自动进入阻塞队列，开始竞争锁。

终止态：

线程执行结束后的状态。

注意：

wait()方法会释放CPU执行权和占有的锁。

sleep(long)方法仅释放CPU使用权，锁仍然占用；线程被放入超时等待队列，与yield相比，它会使线程较长时间得不到运行。

yield()方法仅释放CPU执行权，锁仍然占用，线程会被放入就绪队列，会在短时间内再次执行。

wait和notify必须配套使用，即必须使用同一把锁调用；

wait和notify必须放在一个同步块中调用wait和notify的对象必须是他们所处同步块的锁对象。

# 40、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。

# 41、如何在两个线程间共享数据？

在两个线程间共享变量即可实现共享。

一般来说，共享变量要求变量本身是线程安全的，然后在线程内使用的时候，如果有对共享变量的复合操作，那么也得保证复合操作的线程安全性。

# 42、Java中notify 和 notifyAll有什么区别？

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。

# 43、为什么wait, notify 和 notifyAll这些方法不在thread类里面？

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。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。

# 44、什么是ThreadLocal变量？

ThreadLocal是Java里一种特殊的变量。每个线程都有一个ThreadLocal就是每个线程都拥有了自己独立的一个变量，竞争条件被彻底消除了。它是为创建代价高昂的对象获取线程安全的好方法，比如你可以用ThreadLocal让SimpleDateFormat变成线程安全的，因为那个类创建代价高昂且每次调用都需要创建不同的实例所以不值得在局部范围使用它，如果为每个线程提供一个自己独有的变量拷贝，将大大提高效率。首先，通过复用减少了代价高昂的对象的创建个数。其次，你在没有使用高代价的同步或者不变性的情况下获得了线程安全。

# 45、Java中interrupted 和 isInterrupted方法的区别？

interrupt方法用于中断线程。调用该方法的线程的状态为将被置为”中断”状态。 注意：线程中断仅仅是置线程的中断状态位，不会停止线程。需要用户自己去监视线程的状态为并做处理。支持线程中断的方法（也就是线程中断后会抛出interruptedException的方法）就是在监视线程的中断状态，一旦线程的中断状态被置为“中断状态”，就会抛出中断异常。

interrupted查询当前线程的中断状态，并且清除原状态。如果一个线程被中断了，第一次调用interrupted则返回true，第二次和后面的就返回false了。

isInterrupted仅仅是查询当前线程的中断状态

# 46、为什么wait和notify方法要在同步块中调用？

Java API强制要求这样做，如果你不这么做，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。

# 47、为什么你应该在循环中检查等待条件?

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。

# 48、Java中的同步集合与并发集合有什么区别？

同步集合与并发集合都为多线程和并发提供了合适的线程安全的集合，不过并发集合的可扩展性更高。在Java1.5之前程序员们只有同步集合来用且在多线程并发的时候会导致争用，阻碍了系统的扩展性。

Java5介绍了并发集合像ConcurrentHashMap，不仅提供线程安全还用锁分离和内部分区等现代技术提高了可扩展性。

# 49、什么是线程池？ 为什么要使用它？

创建线程要花费昂贵的资源和时间，如果任务来了才创建线程那么响应时间会变长，而且一个进程能创建的线程数有限。为了避免这些问题，在程序启动的时候就创建若干线程来响应处理，它们被称为线程池，里面的线程叫工作线程。从JDK1.5开始，Java API提供了Executor框架让你可以创建不同的线程池。

# 50、怎么检测一个线程是否拥有锁？

在java.lang.Thread中有一个方法叫holdsLock()，它返回true如果当且仅当当前线程拥有某个具体对象的锁。

# 51、JVM中哪个参数是用来控制线程的栈堆栈小的?

-Xss 每个线程的栈大小

# 52、Thread类中的yield方法有什么作用？

使当前线程从执行状态（运行状态）变为可执行态（就绪状态）。

当前线程到了就绪状态，那么接下来哪个线程会从就绪状态变成执行状态呢？可能是当前线程，也可能是其他线程，看系统的分配了。

# 53、Java中ConcurrentHashMap的并发度是什么？

ConcurrentHashMap把实际map划分成若干部分来实现它的可扩展性和线程安全。这种划分是使用并发度获得的，它是ConcurrentHashMap类构造函数的一个可选参数，默认值为16，这样在多线程情况下就能避免争用。

在JDK8后，它摒弃了Segment（锁段）的概念，而是启用了一种全新的方式实现,利用CAS算法。同时加入了更多的辅助变量来提高并发度，具体内容还是查看源码吧。

# 54、Java中Semaphore是什么？

Java中的Semaphore是一种新的同步类，它是一个计数信号。从概念上讲，从概念上讲，信号量维护了一个许可集合。如有必要，在许可可用前会阻塞每一个 acquire()，然后再获取该许可。每个 release()添加一个许可，从而可能释放一个正在阻塞的获取者。但是，不使用实际的许可对象，Semaphore只对可用许可的号码进行计数，并采取相应的行动。信号量常常用于多线程的代码中，比如数据库连接池。

# 55、Java线程池中submit() 和 execute()方法有什么区别？

两个方法都可以向线程池提交任务，execute()方法的返回类型是void，它定义在Executor接口中。

而submit()方法可以返回持有计算结果的Future对象，它定义在ExecutorService接口中，它扩展了Executor接口，其它线程池类像ThreadPoolExecutor和ScheduledThreadPoolExecutor都有这些方法。

# 56、什么是阻塞式方法？

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。

# 57、Java中的ReadWriteLock是什么？

读写锁是用来提升并发程序性能的锁分离技术的成果。

# 58、volatile 变量和 atomic 变量有什么不同？

Volatile变量可以确保先行关系，即写操作会发生在后续的读操作之前, 但它并不能保证原子性。例如用volatile修饰count变量那么 count++ 操作就不是原子性的。

而AtomicInteger类提供的atomic方法可以让这种操作具有原子性如getAndIncrement()方法会原子性的进行增量操作把当前值加一，其它数据类型和引用变量也可以进行相似操作。

# 59、可以直接调用Thread类的run ()方法么？

当然可以。但是如果我们调用了Thread的run()方法，它的行为就会和普通的方法一样，会在当前线程中执行。为了在新的线程中执行我们的代码，必须使用Thread.start()方法。

# 60、如何让正在运行的线程暂停一段时间？

我们可以使用Thread类的Sleep()方法让线程暂停一段时间。需要注意的是，这并不会让线程终止，一旦从休眠中唤醒线程，线程的状态将会被改变为Runnable，并且根据线程调度，它将得到执行。

# 61、你对线程优先级的理解是什么？

每一个线程都是有优先级的，一般来说，高优先级的线程在运行时会具有优先权，但这依赖于线程调度的实现，这个实现是和操作系统相关的(OS dependent)。我们可以定义线程的优先级，但是这并不能保证高优先级的线程会在低优先级的线程前执行。线程优先级是一个int变量(从1-10)，1代表最低优先级，10代表最高优先级。

java的线程优先级调度会委托给操作系统去处理，所以与具体的操作系统优先级有关，如非特别需要，一般无需设置线程优先级。

# 62、什么是线程调度器和时间分片？

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。 同上一个问题，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）。

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。

# 63、你如何确保main()方法所在的线程是Java 程序最后结束的线程？

我们可以使用Thread类的join()方法来确保所有程序创建的线程在main()方法退出前结束。

# 64、线程之间是如何通信的？

当线程间是可以共享资源时，线程间通信是协调它们的重要的手段。Object类中wait()、notify()、notifyAll()方法可以用于线程间通信关于资源的锁的状态。

# 65、为什么线程通信的方法wait()、notify()和notifyAll()被定义在Object 类里？

Java的每个对象中都有一个锁(monitor，也可以成为监视器) 并且wait()，notify()等方法用于等待对象的锁或者通知其他线程对象的监视器可用。在Java的线程中并没有可供任何对象使用的锁和同步器。这就是为什么这些方法是Object类的一部分，这样Java的每一个类都有用于线程间通信的基本方法。

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

当一个线程需要调用对象的wait()方法的时候，这个线程必须拥有该对象的锁，接着它就会释放这个对象锁并进入等待状态直到其他线程调用这个对象上的notify()方法。同样的，当一个线程需要调用对象的notify()方法时，它会释放这个对象的锁，以便其他在等待的线程就可以得到这个对象锁。由于所有的这些方法都需要线程持有对象的锁，这样就只能通过同步来实现，所以他们只能在同步方法或者同步块中被调用。

# 67、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，并避免程序员错误的认为可以在其他非运行线程调用这些方法。

# 68、如何确保线程安全？

在Java中可以有很多方法来保证线程安全——同步，使用原子类实现并发锁，使用volatile关键字，使用不变类和线程安全类。

# 69、同步方法和同步块，哪个是更好的选择？

同步块是更好的选择，因为它不会锁住整个对象（当然可以让它锁住整个对象）。同步方法会锁住整个对象，哪怕这个类中有多个不相关联的同步块，这通常会导致他们停止执行并需要等待获得这个对象上的锁。

同步块更要符合开放调用的原则，只在需要锁住的代码块锁住相应的对象，这样从侧面来说也可以避免死锁。

# 70、如何创建守护线程？

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，需要在调用start()方法前调用这个方法，否则会抛出IllegalThreadStateException异常。

# 71、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，可以用于安排一个线程在未来的某个特定时间执行。Timer类可以用安排一次性任务或者周期任务。

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。

目前有开源的Qurtz可以用来创建定时任务。