# 第二版：多线程 75 道

目录

我们的网站：https://tech.souyunku.com

**关注我们的公众号：搜云库技术团队，回复以下关键字**

回复:【进群】邀请您进「技术架构分享群」

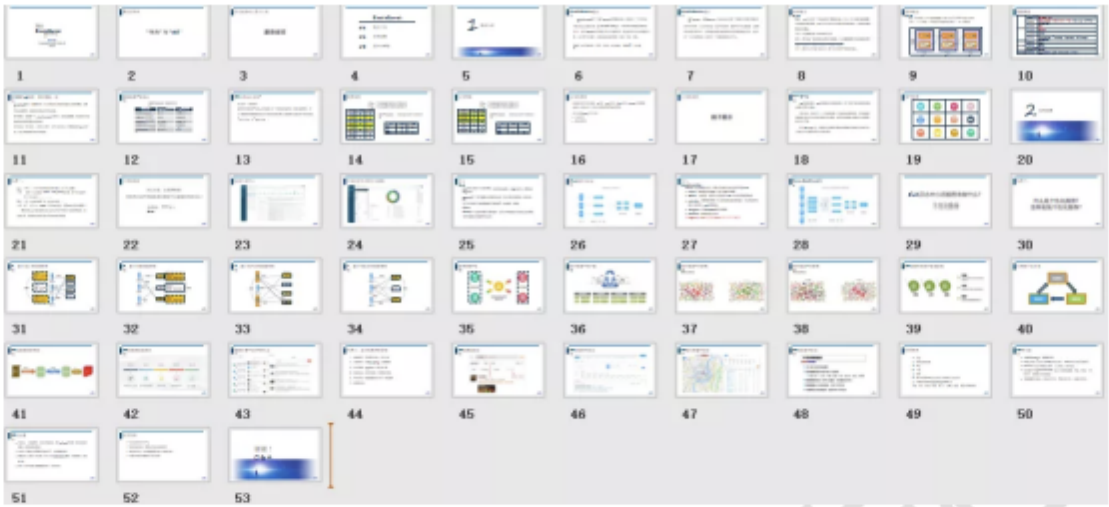
回复:【内推】即可进：北京，上海，广周，深圳，杭州，成都，武汉，南京，郑州，西安，长沙「程序员工作内推群」

回复 【1024】送 4000G 最新架构师视频

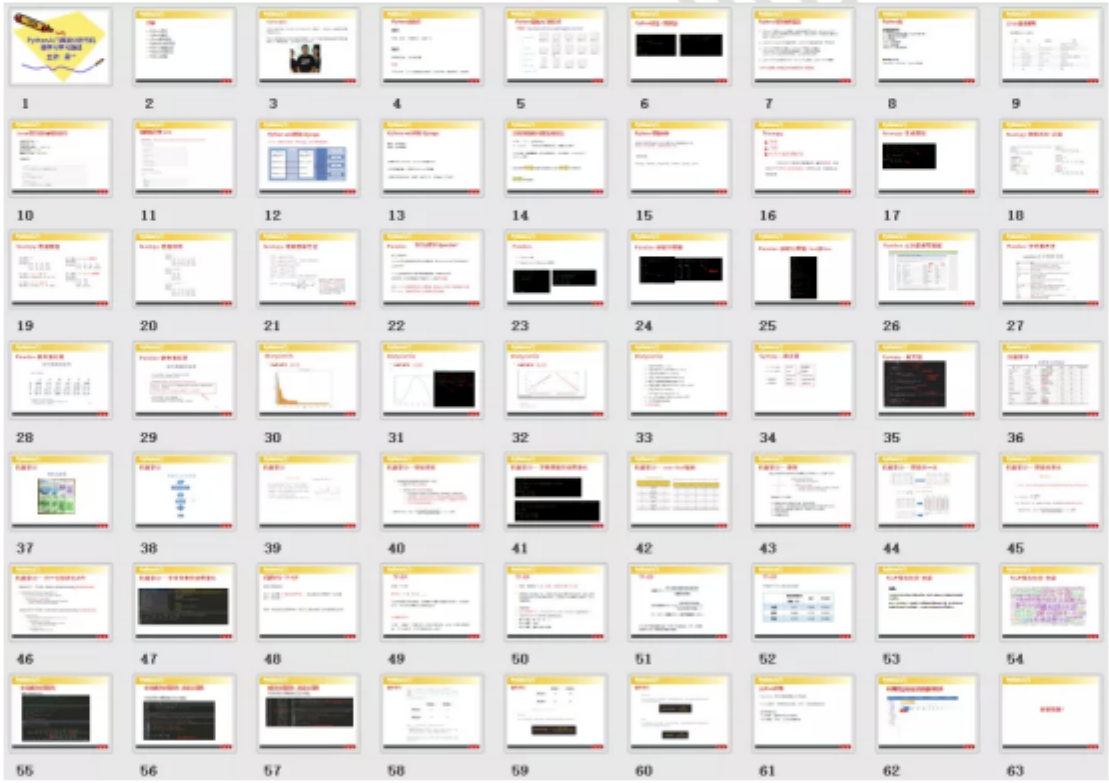
回复 【PPT】即可无套路获取，以下最新整理调优 PPT!

**46 页《JVM 深度调优，演讲 PPT》**

53 页《Elasticsearch 调优演讲 PPT》



63 页《Python 数据分析入门 PPT》




## 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(boolean 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**：
唯一的区别是判断虚拟机(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**：

1、互斥条件：所谓互斥就是进程在某一时间内独占资源。

2、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。

3、不剥夺条件:进程已获得资源，在未使用完之前，不能强行剥夺。

4、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：**

1、高优先级线程吞噬所有的低优先级线程的 CPU 时间。

2、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。

3、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？因为使用有很多的安全隐患吧，没有具体追究，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、如何在 Windows 和 Linux 上查找哪个线程使用的 CPU 时间最长?

参考：

http://daiguahub.com/2016/07/31/使用 jstack 找出消耗 CPU 最多的线程代码/

## 10、什么是原子操作？在 Java Concurrency API 中有哪些原子类(atomic classes)？

原子操作（atomic operation）意为”不可被中断的一个或一系列操作” 。
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。

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。
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。

原子类：AtomicBoolean，AtomicInteger，AtomicLong，AtomicReference
原子数组：AtomicIntegerArray，AtomicLongArray，AtomicReferenceArray
原子属性更新器：AtomicLongFieldUpdater，AtomicIntegerFieldUpdater，AtomicReferenceFieldUpdater

解决 ABA 问题的原子类：AtomicMarkableReference（通过引入一个 boolean 来反映中间有没有变过），AtomicStampedReference（通过引入一个 int 来累加来反映中间有没有变过）

## 11、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：**

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(tryLock 方法)、定时的(tryLock 带参方法)、可中断的(lockInterruptibly)、可多条件队列的(newCondition 方法)锁操作。另外 Lock 的实现类基本都支持非公平锁(默认)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。

## 12、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。

## 13、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（BlockingQueue）是一个支持两个附加操作的队列。

这两个附加的操作是：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。

## 14、什么是 Callable 和 Future?

Callable 接口类似于 Runnable，从名字就可以看出来了，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。

## 15、什么是 FutureTask?使用 ExecutorService 启动任务。

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。

## 16、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，以及 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。

## 17、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（例如一个单一的全局变量），临界区。内核模式下的方法有：事件，信号量，互斥量。

## 18、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（race condition）。

## 19、你将如何使用 thread dump？你将如何分析 Thread dump？

**新建状态（New）**

用 new 语句创建的线程处于新建状态，此时它和其他 Java 对象一样，仅仅在堆区中被分配了内存。

**就绪状态（Runnable）**

当一个线程对象创建后，其他线程调用它的 start()方法，该线程就进入就绪状态，Java 虚拟机会为它创建方法调用栈和程序计数器。处于这个状态的线程位于可运行池中，等待获得 CPU 的使用权。

**运行状态（Running）**

处于这个状态的线程占用 CPU，执行程序代码。只有处于就绪状态的线程才有机会转到运行状态。

**阻塞状态（Blocked）**

阻塞状态是指线程因为某些原因放弃 CPU，暂时停止运行。当线程处于阻塞状态时，Java 虚拟机不会给线程分配 CPU。直到线程重新进入就绪状态，它才有机会转到运行状态。

**阻塞状态可分为以下 3 种：**

**位于对象等待池中的阻塞状态（Blocked in object’s wait pool）：**

当线程处于运行状态时，如果执行了某个对象的 wait()方法，Java 虚拟机就会把线程放到这个对象的等待池中，这涉及到“线程通信”的内容。

**位于对象锁池中的阻塞状态（Blocked in object’s lock pool）：**

当线程处于运行状态时，试图获得某个对象的同步锁时，如果该对象的同步锁已经被其他线程占用，Java 虚拟机就会把这个线程放到这个对象的锁池中，这涉及到“线程同步”的内容。

**其他阻塞状态（Otherwise Blocked）：**

当前线程执行了 sleep()方法，或者调用了其他线程的 join()方法，或者发出了 I/O 请求时，就会进入这个状态。

死亡状态（Dead）

当线程退出 run()方法时，就进入死亡状态，该线程结束生命周期。

我们运行之前的那个死锁代码 SimpleDeadLock.java，然后尝试输出信息(

```
/* 时间，jvm 信息 */
2017-11-01 17:36:28
Full thread dump Java HotSpot(TM) 64-Bit Server VM (25.144-b01 mixed mode):
```

```
/* 线程名称：DestroyJavaVM
编号：#13
优先级：5
系统优先级：0
jvm 内部线程 id：0x0000000001c88800
对应系统线程 id（NativeThread ID）：0x1c18
线程状态： waiting on condition [0x0000000000000000]  (等待某个条件)
线程详细状态：java.lang.Thread.State: RUNNABLE  及之后所有*/
"DestroyJavaVM" #13 prio=5 os_prio=0 tid=0x0000000001c88800
nid=0x1c18 waiting on condition [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE

"Thread-1" #12 prio=5 os_prio=0 tid=0x0000000018d49000
nid=0x17b8 waiting for monitor entry [0x0000000019d7f000]
/* 线程状态：阻塞（在对象同步上）
     代码位置：at
com.leo.interview.SimpleDeadLock$B.run(SimpleDeadLock.java:56)
     等待锁：0x00000000d629b4d8
     已经获得锁：0x00000000d629b4e8*/
   java.lang.Thread.State: BLOCKED (on object monitor)
     at
com.leo.interview.SimpleDeadLock$B.run(SimpleDeadLock.java:56)
     - waiting to lock <0x00000000d629b4d8> (a java.lang.Object)
     - locked <0x00000000d629b4e8> (a java.lang.Object)

"Thread-0" #11 prio=5 os_prio=0 tid=0x0000000018d44000 nid=0x1ebc
waiting for monitor entry [0x000000001907f000]
   java.lang.Thread.State: BLOCKED (on object monitor)
     at
com.leo.interview.SimpleDeadLock$A.run(SimpleDeadLock.java:34)
```

```
        - waiting to lock <0x00000000d629b4e8> (a java.lang.Object)
        - locked <0x00000000d629b4d8> (a java.lang.Object)

"Service Thread" #10 daemon prio=9 os_prio=0
tid=0x0000000018ca5000 nid=0x1264 runnable [0x0000000000000000]
    java.lang.Thread.State: RUNNABLE

"C1 CompilerThread2" #9 daemon prio=9 os_prio=2
tid=0x0000000018c46000 nid=0xb8c waiting on condition
[0x0000000000000000]
    java.lang.Thread.State: RUNNABLE

"C2 CompilerThread1" #8 daemon prio=9 os_prio=2
tid=0x0000000018be4800 nid=0x1db4 waiting on condition
[0x0000000000000000]
    java.lang.Thread.State: RUNNABLE

"C2 CompilerThread0" #7 daemon prio=9 os_prio=2
tid=0x0000000018be3800 nid=0x810 waiting on condition
[0x0000000000000000]
    java.lang.Thread.State: RUNNABLE

"Monitor Ctrl-Break" #6 daemon prio=5 os_prio=0
tid=0x0000000018bcc800 nid=0x1c24 runnable [0x00000000193ce000]
    java.lang.Thread.State: RUNNABLE
      at java.net.SocketInputStream.socketRead0(Native Method)
      at
java.net.SocketInputStream.socketRead(SocketInputStream.java:116)
      at java.net.SocketInputStream.read(SocketInputStream.java:171)
      at java.net.SocketInputStream.read(SocketInputStream.java:141)
      at sun.nio.cs.StreamDecoder.readBytes(StreamDecoder.java:284)
```

```
        at sun.nio.cs.StreamDecoder.implRead(StreamDecoder.java:326)
        at sun.nio.cs.StreamDecoder.read(StreamDecoder.java:178)
        - locked <0x00000000d632b928> (a java.io.InputStreamReader)
        at java.io.InputStreamReader.read(InputStreamReader.java:184)
        at java.io.BufferedReader.fill(BufferedReader.java:161)
        at java.io.BufferedReader.readLine(BufferedReader.java:324)
        - locked <0x00000000d632b928> (a java.io.InputStreamReader)
        at java.io.BufferedReader.readLine(BufferedReader.java:389)
        at
com.intellij.rt.execution.application.AppMainV2$1.run(AppMainV2.java:6
4)

"Attach Listener" #5 daemon prio=5 os_prio=2
tid=0x0000000017781800 nid=0x524 runnable [0x0000000000000000]
    java.lang.Thread.State: RUNNABLE

"Signal Dispatcher" #4 daemon prio=9 os_prio=2
tid=0x000000001778f800 nid=0x1b08 waiting on condition
[0x0000000000000000]
    java.lang.Thread.State: RUNNABLE

"Finalizer" #3 daemon prio=8 os_prio=1 tid=0x000000001776a800
nid=0xdac in Object.wait() [0x0000000018b6f000]
    java.lang.Thread.State: WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        - waiting on <0x00000000d6108ec8> (a
java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:143)
        - locked <0x00000000d6108ec8> (a
java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:164)
```

```
        at java.lang.ref.Finalizer$FinalizerThread.run(Finalizer.java:209)

"Reference Handler" #2 daemon prio=10 os_prio=2
tid=0x0000000017723800 nid=0x1670 in Object.wait()
[0x00000000189ef000]
    java.lang.Thread.State: WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        - waiting on <0x00000000d6106b68> (a
java.lang.ref.Reference$Lock)
        at java.lang.Object.wait(Object.java:502)
        at java.lang.ref.Reference.tryHandlePending(Reference.java:191)
        - locked <0x00000000d6106b68> (a java.lang.ref.Reference$Lock)
        at java.lang.ref.Reference$ReferenceHandler.run(Reference.java:153)

"VM Thread" os_prio=2 tid=0x000000001771b800 nid=0x604 runnable

"GC task thread#0 (ParallelGC)" os_prio=0 tid=0x0000000001c9d800
nid=0x9f0 runnable

"GC task thread#1 (ParallelGC)" os_prio=0 tid=0x0000000001c9f000
nid=0x154c runnable

"GC task thread#2 (ParallelGC)" os_prio=0 tid=0x0000000001ca0800
nid=0xcd0 runnable

"GC task thread#3 (ParallelGC)" os_prio=0 tid=0x0000000001ca2000
nid=0x1e58 runnable

"VM Periodic Task Thread" os_prio=2 tid=0x0000000018c5a000
nid=0x1b58 waiting on condition
```

```
JNI global references: 33


/* 此处可以看待死锁的相关信息!   */
Found one Java-level deadlock:
=============================
"Thread-1":
  waiting to lock monitor 0x0000000017729fc8 (object
0x00000000d629b4d8, a java.lang.Object),
  which is held by "Thread-0"
"Thread-0":
  waiting to lock monitor 0x0000000017727738 (object
0x00000000d629b4e8, a java.lang.Object),
  which is held by "Thread-1"

Java stack information for the threads listed above:
===================================================
=====
"Thread-1":
      at
com.leo.interview.SimpleDeadLock$B.run(SimpleDeadLock.java:56)
      - waiting to lock <0x00000000d629b4d8> (a java.lang.Object)
      - locked <0x00000000d629b4e8> (a java.lang.Object)
"Thread-0":
      at
com.leo.interview.SimpleDeadLock$A.run(SimpleDeadLock.java:34)
      - waiting to lock <0x00000000d629b4e8> (a java.lang.Object)
      - locked <0x00000000d629b4d8> (a java.lang.Object)

Found 1 deadlock.
```

```
/* 内存使用状况，详情得看 JVM 方面的书 */
Heap
 PSYoungGen      total 37888K, used 4590K [0x00000000d6100000,
0x00000000d8b00000, 0x0000000100000000)
  eden space 32768K, 14% used
[0x00000000d6100000,0x00000000d657b968,0x00000000d8100000)
  from space 5120K, 0% used
[0x00000000d8600000,0x00000000d8600000,0x00000000d8b00000)
  to   space 5120K, 0% used
[0x00000000d8100000,0x00000000d8100000,0x00000000d8600000)
 ParOldGen       total 86016K, used 0K [0x0000000082200000,
0x0000000087600000, 0x00000000d6100000)
  object space 86016K, 0% used
[0x0000000082200000,0x0000000082200000,0x0000000087600000)
 Metaspace       used 3474K, capacity 4500K, committed 4864K,
reserved 1056768K
  class space    used 382K, capacity 388K, committed 512K, reserved
1048576K
```

## 20、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。

## 21、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。

## 22、在 Java 中 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 (common barrier point)。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。

## 23、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(Immutable Objects)即对象一旦被创建它的状态（对象的数据，也即对象属性值）就不能改变，反之即为可变对象(Mutable Objects)。
不可变对象的类即为不可变类(Immutable Class)。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）。

## 24、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”（switchframe）。“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。

## 25、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。

## 26、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。

## 27、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

1、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。

2、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。

3、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**

1、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。

2、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。

3、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。

## 28、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

## 29、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。

## 30、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。

## 31、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。

## 32、java 如何实现多线程之间的通讯和协作?

中断 和 共享变量

## 33、什么是可重入锁（ReentrantLock）？

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。

## 34、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。

## 35、乐观锁和悲观锁的理解及如何实现，有哪些实现方式?

**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**：顾名思义，就是很乐观，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。

**乐观锁的实现方式**：

1、使用版本标识来确定读到的数据与提交时的数据是否一致。提交后修改版本标识，不一致时可以采取丢弃和再次尝试的策略。

2、java 中的 Compare and Swap 即 CAS ，当多个线程尝试使用 CAS 同时更新同一个变量时，只有其中一个线程能更新变量的值，而其它线程都失败，失败的线程并不会被挂起，而是被告知这次竞争中失败，并可以再次尝试。 CAS 操作中包含三个操作数 —— 需要读写的内存位置（V）、进行比较的预期原值（A）和拟写入的新值(B)。如果内存位置 V 的值与预期原值 A 相匹配，那么处理器会自动将该位置值更新为新值 B。否则处理器不做任何操作。

**CAS 缺点：**

1、**ABA 问题：**

比如说一个线程 one 从内存位置 V 中取出 A，这时候另一个线程 two 也从内存中取出 A，并且 two 进行了一些操作变成了 B，然后 two 又将 V 位置的数据变成 A，这时候线程 one 进行 CAS 操作发现内存中仍然是 A，然后 one 操作成功。尽管线程 one 的 CAS 操作成功，但可能存在潜藏的问题。从 Java1.5 开始 JDK 的 atomic 包里提供了一个类 AtomicStampedReference 来解决 ABA 问题。

2、**循环时间长开销大：**

对于资源竞争严重（线程冲突严重）的情况，CAS 自旋的概率会比较大，从而浪费更多的 CPU 资源，效率低于 synchronized。

3、**只能保证一个共享变量的原子操作：**

当对一个共享变量执行操作时，我们可以使用循环 CAS 的方式来保证原子操作，但是对多个共享变量操作时，循环 CAS 就无法保证操作的原子性，这个时候就可以用锁。

## 36、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。

## 37、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。

1、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；
2、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**

1、读写分离，读和写分开
2、最终一致性

3、使用另外开辟空间的思路，来解决并发冲突

## 38、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。

## 39、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)。

## 40、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。

## 41、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：** NEW

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：RUNNABLE
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。

## 42、用 Java 实现阻塞队列

参考 java 中的阻塞队列的内容吧，直接实现有点烦：
http://www.infoq.com/cn/articles/java-blocking-queue

## 43、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。

## 44、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。

## 45、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。

## 46、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。

## 47、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。

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

### interrupt

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

## 49、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，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。

## 50、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。

## 51、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。

## 52、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。

## 53、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。

## 54、你如何在 Java 中获取线程堆栈?

kill -3 [java pid]
不会在当前终端输出，它会输出到代码执行的或指定的地方去。比如，kill -3 tomcat pid，输出堆栈到 log 目录下。
Jstack [java pid]
这个比较简单，在当前终端显示，也可以重定向到指定文件中。
-JvisualVM：Thread Dump
不做说明，打开 JvisualVM 后，都是界面操作，过程还是很简单的。
55、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

## 56、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。

## 57、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。

## 58、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。

## 59、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。

## 60、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。

## 61、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。

## 62、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。

## 63、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。

## 64、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。

## 65、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。

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

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。

## 67、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。

## 68、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。

## 69、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。

## 70、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。

## 71、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。

## 72、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(atomic concurrent classes)，实现并发锁，使用 volatile 关键字，使用不变类和线程安全类。

## 73、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（当然你也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。

## 74、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。

## 75、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。