[java多线（2019年）](https://www.cnblogs.com/xiaowangbangzhu/p/10443289.html)

**50) 写出3条你遵循的多线程最佳实践**

这种问题我最喜欢了，我相信你在写并发代码来提升性能的时候也会遵循某些最佳实践。以下三条最佳实践我觉得大多数Java程序员都应该遵循：

* 给你的线程起个有意义的名字。  
  这样可以方便找bug或追踪。OrderProcessor, QuoteProcessor or TradeProcessor 这种名字比 Thread-1. Thread-2 and Thread-3 好多了，给线程起一个和它要完成的任务相关的名字，所有的主要框架甚至JDK都遵循这个最佳实践。
* 避免锁定和缩小同步的范围  
  锁花费的代价高昂且上下文切换更耗费时间空间，试试最低限度的使用同步和锁，缩小临界区。因此相对于同步方法我更喜欢同步块，它给我拥有对锁的绝对控制权。
* 多用同步类少用wait 和 notify  
  首先，CountDownLatch, Semaphore, CyclicBarrier 和 Exchanger 这些同步类简化了编码操作，而用wait和notify很难实现对复杂控制流的控制。其次，这些类是由最好的企业编写和维护在后续的JDK中它们还会不断 优化和完善，使用这些更高等级的同步工具你的程序可以不费吹灰之力获得优化。
* 多用并发集合少用同步集合  
  这是另外一个容易遵循且受益巨大的最佳实践，并发集合比同步集合的可扩展性更好，所以在并发编程时使用并发集合效果更好。如果下一次你需要用到map，你应该首先想到用ConcurrentHashMap。

**51) 如何强制启动一个线程？**

这个问题就像是如何强制进行Java垃圾回收，目前还没有觉得方法，虽然你可以使用System.gc()来进行垃圾回收，但是不保证能成功。在Java里面没有办法强制启动一个线程，它是被线程调度器控制着且Java没有公布相关的API。

**52) Java中的fork join框架是什么？**

fork join框架是JDK7中出现的一款高效的工具，Java开发人员可以通过它充分利用现代服务器上的多处理器。它是专门为了那些可以递归划分成许多子模块 设计的，目的是将所有可用的处理能力用来提升程序的性能。fork join框架一个巨大的优势是它使用了工作窃取算法，可以完成更多任务的工作线程可以从其它线程中窃取任务来执行。

**53） Java多线程中调用wait() 和 sleep()方法有什么不同？**

Java程序中wait 和 sleep都会造成某种形式的暂停，它们可以满足不同的需要。wait()方法用于线程间通信，如果等待条件为真且其它线程被唤醒时它会释放锁，而 sleep()方法仅仅释放CPU资源或者让当前线程停止执行一段时间，但不会释放锁。

[**多线程常见面试题及答案**](https://www.cnblogs.com/JavaHxm/p/10832835.html)

1、如何在Java中实现线程(4种)？

1.继承Thread类，重写run方法（其实Thread类本身也实现了Runnable接口）

2.实现Runnable接口，重写run方法

3.实现Callable接口，重写call方法（有返回值）

4.使用线程池（有返回值）

https://www.cnblogs.com/duanjiapingjy/p/9434244.htmlhttps://www.cnblogs.com/duanjiapingjy/p/9434244.html

2、在具体多线程编程实践中，如何选用Runnable还是Thread？

  Java中实现多线程有两种方法：继承Thread类、实现Runnable接口，在程序开发中只要是多线程，肯定永远以实现Runnable接口为主，因为实现Runnable接口相比继承Thread类有如下优势：

    1、可以避免由于Java的单继承特性而带来的局限；

    2、增强程序的健壮性，代码能够被多个线程共享，代码与数据是独立的；

适合多个相同程序代码的线程区处理同一资源的情况。

3、Thread类中的start()和run()方法有什么区别？

 start（）方法来启动线程，真正实现了多线程运行，这时无需等待run方法体代码执行完毕而直接继续执行下面的代码： 通过调用Thread类的start()方法来启动一个线程，这时此线程是处于就绪状态，并没有运行。然后通过此Thread类调用方法run()来完成其运行操作的，这里方法run()称为线程体，它包含了要执行的这个线程的内容，Run方法运行结束，此线程终止，而CPU再运行其它线程。

  run（）方法当作普通方法的方式调用，程序还是要顺序执行，还是要等待run方法体执行完毕后才可继续执行下面的代码： 而如果直接用run方法，这只是调用一个方法而已，程序中依然只有主线程–这一个线程，其程序执行路径还是只有一条，这样就没有达到多线程的目的。

4、Java中Runnable和Callable有什么不同

相同点：

1. 两者都是接口；（废话）

2. 两者都可用来编写多线程程序；

3. 两者都需要调用Thread.start()启动线程；

不同点：

1. 两者最大的不同点是：实现Callable接口的任务线程能返回执行结果；而实现Runnable接口的任务线程不能返回结果；

2. Callable接口的call()方法允许抛出异常；而Runnable接口的run()方法的异常只能在内部消化，不能继续上抛；

注意点：

• Callable接口支持返回执行结果，此时需要调用FutureTask.get()方法实现，此方法会阻塞主线程直到获取‘将来’结果；当不调用此方法时，主线程不会阻塞！

5、如何避免死锁？

1. 加锁顺序

按照顺序加锁是一种有效的死锁预防机制。但是，这种方式需要你事先知道所有可能会用到的锁(并对这些锁做适当的排序)，但总有些时候是无法预知的。

2. 加锁时限

另外一个可以避免死锁的方法是在尝试获取锁的时候加一个超时时间，这也就意味着在尝试获取锁的过程中若超过了这个时限该线程则放弃对该锁请求。

3.死锁检测

死锁检测是一个更好的死锁预防机制，它主要是针对那些不可能实现按序加锁并且锁超时也不可行的场景。

每当一个线程获得了锁，会在线程和锁相关的数据结构中（map、graph等等）将其记下。除此之外，每当有线程请求锁，也需要记录在这个数据结构中。

当一个线程请求锁失败时，这个线程可以遍历锁的关系图看看是否有死锁发生。例如，线程A请求锁7，但是锁7这个时候被线程B持有，这时线程A就可以检查一下线程B是否已经请求了线程A当前所持有的锁。如果线程B确实有这样的请求，那么就是发生了死锁（线程A拥有锁1，请求锁7；线程B拥有锁7，请求锁1）。

当然，死锁一般要比两个线程互相持有对方的锁这种情况要复杂的多。线程A等待线程B，线程B等待线程C，线程C等待线程D，线程D又在等待线程A。线程A为了检测死锁，它需要递进地检测所有被B请求的锁。从线程B所请求的锁开始，线程A找到了线程C，然后又找到了线程D，发现线程D请求的锁被线程A自己持有着。这是它就知道发生了死锁。

6、Java多线程中调用wait() 和 sleep()方法有什么不同？

**共同点：**

1. 他们都是在多线程的环境下，都可以在程序的调用处阻塞指定的毫秒数，并返回。

2. wait()和sleep()都可以通过interrupt()方法 **打断线程的暂停状态** ，从而使线程立刻抛出InterruptedException。

   如果线程A希望立即结束线程B，则可以对线程B对应的Thread实例调用interrupt方法。如果此刻线程B正在wait/sleep /join，则线程B会立刻抛出InterruptedException，在catch() {} 中直接return即可安全地结束线程。

   需要注意的是，InterruptedException是线程自己从内部抛出的，并不是interrupt()方法抛出的。**对某一线程调用 interrupt()时，如果该线程正在执行普通的代码，那么该线程根本就不会抛出InterruptedException。但是，一旦该线程进入到 wait()/sleep()/join()后，就会立刻抛出InterruptedException** 。

**不同点：**

1. Thread类的方法：sleep(),yield()等

   Object的方法：wait()和notify()等

2. **每个对象都有一个锁来控制同步访问**。Synchronized关键字可以和对象的锁交互，来实现线程的同步。

   sleep方法没有释放锁，而wait方法释放了锁，使得其他线程可以使用同步控制块或者方法。

3.**wait，notify和notifyAll只能在同步控制方法或者同步控制块里面使用，而sleep可以在任何地方使用**

4. sleep必须捕获异常，而wait，notify和notifyAll不需要捕获异常

**7、什么是Executor框架**

我们知道线程池就是线程的集合，线程池集中管理线程，以实现线程的重用，降低资源消耗，提高响应速度等。线程用于执行异步任务，单个的线程既是工作单元也是执行机制，从JDK1.5开始，为了把工作单元与执行机制分离开，Executor框架诞生了，他是一个用于统一创建与运行的接口。Executor框架实现的就是线程池的功能。

**8、在Java中Executor和Executors的区别**

Executor是**http://lib.csdn.net/base/javaee" \o "Java EE知识库" \t "https://blog.csdn.net/qq991029781/article/details/\_blankJava**线程池的顶级**接口；**

Executors是一个**类，** Executors类提供了若干个静态方法，用于生成不同类型的线程池：

**9、什么是多线程中的上下文切换？**

即使是单核CPU也支持多线程执行代码，CPU通过给每个线程分配CPU时间片来实现这个机制。时间片是CPU分配给各个线程的时间，因为时间片非常短，所以CPU通过不停地切换线程执行，让我们感觉多个线程时同时执行的，时间片一般是几十毫秒（ms）。

CPU通过时间片分配算法来循环执行任务，当前任务执行一个时间片后会切换到下一个任务。但是，在切换前会保存上一个任务的状态，以便下次切换回这个任务时，可以再次加载这个任务的状态，**从任务保存到再加载的过程就是一次上下文切换**。

这就像我们同时读两本书，当我们在读一本英文的技术书籍时，发现某

**10、什么是线程安全**

线程安全的代码是多个线程同时执行也能工作的代码

如果一段代码可以保证多个线程访问的时候正确操作共享数据，那么它是线程安全的

如果你的代码所在的进程中有多个线程在同时运行，而这些线程可能会同时运行这段代码。如果每次运行结果和单线程运行的结果是一样的，而且其他的变量的值也和预期的是一样的，

就是线程安全的。

或者说:一个类或者程序所提供的接口对于线程来说是原子操作或者多个线程之间的切换不会导致该接口的执行结果存在二义性,也就是说我们不用考虑同步的问题。

**11、如何检测死锁？怎么预防死锁？**

利用Java自带工具 https://www.baidu.com/s?wd=JConsole&tn=24004469\_oem\_dg&rsv\_dl=gh\_pl\_sl\_csd" \t "https://blog.csdn.net/wolfcode\_cn/article/details/\_blankJConsole.界面化查看死锁。

**利用jstack 命令检测死锁 jpg jstack -l pid >wenjian.txt**

1. 破坏“不可剥夺”条件：一个进程不能获得所需要的全部资源时便处于等待状态，等待期间他占有的资源将被隐式的释放重新加入到 系统的资源列表中，可以被其他的进程使用，而等待的进程只有重新获得自己原有的资源以及新申请的资源才可以重新启动，执行。

2. 破坏”请求与保持条件“：第一种方法静态分配即每个进程在开始执行时就申请他所需要的全部资源。第二种是动态分配即每个进程在申请所需要的资源时他本身不占用系统资源。

3. 破坏“循环等待”条件：采用资源有序分配其基本思想是将系统中的所有资源顺序编号，将紧缺的，稀少的采用较大的编号，在申请资源时必须按照编号的顺序进行，一个进程只有获得较小编号的进程才能申请较大编号的进程。

**12、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

(时间片轮转法、优先级调度法、多级反馈队列调度法等

**13、Java中如何获取到线程dump文件**

jmap -dump:format=b,file=F:/heamdump.out 16540”命令即可以生成

**jpg jstack -l pid >wenjian.txt**

**14、池技术有什么作用，常见的池技术有哪些**

**起到对象复用**

**15、用线程池有什么好处，请谈谈线程池的使用场景**

1、避免重复创建线程，减少在创建和 销毁线程时所花时间，及系统的整体开销

2、避免系统创建大量线程而消耗系统资源

3、用户提交的数据能够及时得到处理，响应速度快

4、能够更好的监控和管理线程

常量池 线程池 数据库连接池

**16、线程池的技术原理是什么**

预先启动一些线程，线程无限循环从任务队列中获取一个任务进行执行，直到线程池被关闭。如果某个线程因为执行某个任务发生异常而终止，那么重新创建一个新的线程而已。如此反复。

**17、线程池有哪些种类，各自的使用场景是什么？**

newCachedThreadPool：

• 底层：返回ThreadPoolExecutor实例，corePoolSize为0；maximumPoolSize为Integer.MAX\_VALUE；keepAliveTime为60L；unit为TimeUnit.SECONDS；workQueue为SynchronousQueue(同步队列)

• 通俗：当有新任务到来，则插入到SynchronousQueue中，由于SynchronousQueue是同步队列，因此会在池中寻找可用线程来执行，若有可以线程则执行，若没有可用线程则创建一个线程来执行该任务；若池中线程空闲时间超过指定大小，则该线程会被销毁。

• 适用：执行很多短期异步的小程序或者负载较轻的服务器

newFixedThreadPool：

• 底层：返回ThreadPoolExecutor实例，接收参数为所设定线程数量nThread，corePoolSize为nThread，maximumPoolSize为nThread；keepAliveTime为0L(不限时)；unit为：TimeUnit.MILLISECONDS；WorkQueue为：new LinkedBlockingQueue<Runnable>() 无解阻塞队列

• 通俗：创建可容纳固定数量线程的池子，每隔线程的存活时间是无限的，当池子满了就不在添加线程了；如果池中的所有线程均在繁忙状态，对于新任务会进入阻塞队列中(无界的阻塞队列)

• 适用：执行长期的任务，性能好很多

newSingleThreadExecutor:

• 底层：FinalizableDelegatedExecutorService包装的ThreadPoolExecutor实例，corePoolSize为1；maximumPoolSize为1；keepAliveTime为0L；unit为：TimeUnit.MILLISECONDS；workQueue为：new LinkedBlockingQueue<Runnable>() 无解阻塞队列

• 通俗：创建只有一个线程的线程池，且线程的存活时间是无限的；当该线程正繁忙时，对于新任务会进入阻塞队列中(无界的阻塞队列)

• 适用：一个任务一个任务执行的场景

NewScheduledThreadPool:

• 底层：创建ScheduledThreadPoolExecutor实例，corePoolSize为传递来的参数，maximumPoolSize为Integer.MAX\_VALUE；keepAliveTime为0；unit为：TimeUnit.NANOSECONDS；workQueue为：new DelayedWorkQueue() 一个按超时时间升序排序的队列

• 通俗：创建一个固定大小的线程池，线程池内线程存活时间无限制，线程池可以支持定时及周期性任务执行，如果所有线程均处于繁忙状态，对于新任务会进入DelayedWorkQueue队列中，这是一种按照超时时间排序的队列结构

• 适用：周期性执行任务的场景

**18、线程池有哪些重要的参数？**

a.核心线程数

b 最大线程数

c 线程空闲时间

c 时间单位

d 阻塞队列大小：queueCapacity

e 任务拒绝处理器 ：rejectedExceptionHandler

**19、你们在具体的设计开发过程中是如何设置这些重要参数的？**

**根据任务的特性具体方案 具体定制 参见17**

**20、单例的使用场景是什么，如何实现单例**

系统中只存在一个实力，一种是枚举，还有一种私有静态内部类

单例对象的类必须保证只有一个实例存在。许多时候整个系统只需要拥有一个的全局对象，这样有利于我们协调系统整体的行为

**21、如何在Java中创建线程安全的Singleton**

public class StaticSingleton {

02 private StaticSingleton(){

03 System.out.println("StaticSingleton is create");

04 }

05 private static class SingletonHolder {

06 private static StaticSingleton instance = new StaticSingleton();

07 }

08 public static StaticSingleton getInstance() {

09 return SingletonHolder.instance;

10 }

**22、11 }**

**23、synchronzied关键词的使用**

synchronized包裹代码块：

I . synchronized(对象){}

II . synchronized(类名.class){}

III. synchronized(this){}

synchronized修饰方法：

I .public synchronized void memberMethod(){};

II.public static synchronized void staticMethod(){};

**24、ReentrantLock和synchronized使用的场景是什么，机制有何不同**

在资源竞争不是很激烈的情况下，偶尔会有同步的情形下，synchronized是很合适的。原因在于，编译程序通常会尽可能的进行优化

synchronize，另外可读性非常好，不管用没用过5.0多线程包的程序员都能理解。

ReentrantLock 类实现了 Lock ，它拥有与 synchronized 相同的并发性和内存语义，但是添加了类似轮询锁、定时锁等候和可中断锁等候的一些特性。此外，它还提供了在激烈争用情况下更佳

的性能。

其实ReentrantLock是一个可重入的互斥锁，重入锁是一种递归无阻塞的同步机制。ReentrantLock由最近成功获取锁，还没有释放的线程所拥有，当锁被另一个线程拥有时，调用lock的线程可以成功获取锁。如果锁已经被当前线程拥有，当前线程会立即返回

ReentrantLock可以等同于synchronized使用，但是比synchronized有更强的功能、可以提供更灵活的锁机制、同时减少死锁的发生概率。在确实需要一些 synchronized 所没有的特性的时候，比如时间锁等候、可中断锁等候、无块结构锁、多个条件变量或者轮询锁。 ReentrantLock 还具有可伸缩性的好处，应当在高度争用的情况下使用它，但是请记住，大多数 synchronized 块几乎从来没有出现过争用，所以可以把高度争用放在一边。我建议用 synchronized 开发，直到确实证明 synchronized 不合适，而不要仅仅是假设如果使用 ReentrantLock “性能会更好”。请记住，这些是供高级用户使用的高级工具。（而且，真正的高级用户喜欢选择能够找到的最简单工具，直到他们认为

**25、什么是ThreadLocal变量**

ThreadLocal，Thread：线程，这个毫无疑问。那Local呢？本地的，局部的。也就是说，ThreadLocal是线程本地的变量，只要是本线程内都可以使用，线程结束了，那么相应的线程本地变量也就跟随着线程消失了。

**26、ThreadLocal技术原理是什么，它在架构中常常用来做什么？**

**27、java多线程有哪些常见的锁，各自用法是什么？**

synchronized (同步)synchronized关键字修饰的代码相当于数据库上的互斥锁。确保多个线程在同一时刻只能由一个线程处于方法或同步块中，确保线程对变量访问的可见和排它，获得锁的对象在代码结束后，会对锁进行释放。

       synchronzied使用方法有两个：①加在方法上面锁定方法，②定义synchronized块。

condition (配合lock使用 类似 object.wait)Condition.await()方法相当于Object.wait()方法,而Condition.signal()方法相当于Object.notify()方法。当然它也有对应的Condition.signalAll()方法。同样的在调用Condition.await()之后,线程占用的锁会被释放。这样在Condition.signal()方法调用的时候才获取到锁。

需要注意的是Condition.signal()方法调用之后,被唤醒的线程因为需要重新获取锁。所以需要等到调用Condition.signal()的线程释放了锁(调用ReentrantLock.unlock())之后才能继续执行。

**lockInterruptibly()方法**比较特殊，当通过这个方法去获取锁时，如果线程正在等待获取锁，则这个线程能够响应中断，即中断线程的等待状态。也就使说，当两个线程同时通过lock.lockInterruptibly()想获取某个锁时，假若此时线程A获取到了锁，而线程B只有在等待，那么对线程B调用threadB.interrupt()方法能够中断线程B的等待过程。由于lockInterruptibly()的声明中抛出了异常，所以lock.lockInterruptibly()必须放在try块中或者在调用lockInterruptibly()的方法外声明抛出InterruptedException。

ReentrantLock，意思是“可重入锁”，ReentrantLock是唯一实现了Lock接口的类，并且ReentrantLock提供了更多的方法。

分为读锁和写锁，多个读锁不互斥，读锁与写锁互斥，这是由jvm自己控制的，我们只要上好相应的锁即可。如果你的代码只读数据，可以很多人同时读，但不能同时写，那就上读锁；如果你的代码修改数据，只能有一个人在写，且不能同时读取，那就上写锁。总之，读的时候上读锁，写的时候上写锁！读写锁接口：**ReadWriteLock**，它的具体实现类为：**ReentrantReadWriteLock**。

**28、CountDownLatch用于多线程的什么场景**

1. ountDownLatch类。这个类是一个同步辅助类。用于一个线程等待多个操作完成之后再执行，也就是这个当前线程会一直阻塞，直到它所等待的多个操作已经完成。await方法，需要等到其他操作先完成的那个线程调用的，先将线程休眠，直到其他操作完成，计数器减为0，才会唤醒因此休眠的线程

2. countDown方法，每个被等待的事件在完成之后调用，会将计数器减一

**29、volatile适用于高并发的什么场景**

volatile最适用一个线程写，多个线程读的场合。

   如果有多个线程并发写操作，仍然需要使用锁或者线程安全的容器或者原子变量来代替。(摘自Netty权威指南)

您只能在有限的一些情形下使用 volatile 变量替代锁。要使 volatile 变量提供理想的线程安全，必须同时满足下面两个条件：

• 对变量的写操作不依赖于当前值。

• 该变量没有包含在具有其他变量的不变式中。

**30、多线程join方法用于什么场景？**

，主线程创建并启动了子线程，如果子线程中需要进行大量的耗时运算，主线程往往将早于子线程结束之前结束，如果主线程想等待子线程执行完毕后，获得子线程中的处理完的某个数据，就要用到join方法了，方法join（）的作用是等待线程对象呗销毁；

**join底层是wait方法，所以它是会释放https://www.baidu.com/s?wd=%E5%AF%B9%E8%B1%A1%E9%94%81&tn=24004469\_oem\_dg&rsv\_dl=gh\_pl\_sl\_csd" \t "https://blog.csdn.net/liuyifeng1920/article/details/\_blank对象锁的，而sleep在同步的方法中是不释放对象锁的，只有同步方法执行完毕，其他线程才可以执行。**

**31、java多线程中让所有子线程执行完毕的方法有哪几种？**

1、用sleep方法，让主线程睡眠一段时间，当然这个睡眠时间是主观的时间，是我们自己定的，这个方法不推荐，但是在这里还是写一下，毕竟是解决方法

2、使用Thread的join()等待所有的子线程执行完毕，主线程在执行，thread.join()把指定的线程加入到当前线程，可以将两个交替执行的线程合并为顺序执行的线程。比如在线程B中调用了线程A的Join()方法，直到线程A执行完毕后，才会继续执行线程B。

3、countDownLatch不可能重新初始化或者修改CountDownLatch对象内部计数器的值，一个线程调用countdown方法happen-before另外一个线程调用await方法

4、同步屏障CyclicBarrier方法可以使用reset()方法重置，所以CyclicBarrier方法可以能处理更为复杂的业务场景。

**32、高并发环境下的计数器如何实现？**

**33、HashTable、HashMap、ConcurrentHashMap各自的技术原理和使用场景是什么？**

HashMap

实现了Map接口，实现了将唯一键隐射到特定值上。允许一个NULL键和多个NULL值。非线程安全。

HashTable

类似于HashMap，但是不允许NULL键和NULL值，比HashMap慢，因为它是同步的。HashTable是一个线程安全的类，它使用synchronized来锁住整张Hash表来实现线程安全，即每次锁住整张表让线程独占。

ConcurrentHashMap

ConcurrentHashMap允许多个修改操作并发进行，其关键在于使用了锁分离技术。它使用了多个锁来控制对hash表的不同部分进行的修改。ConcurrentHashMap内部使用段(Segment)来表示这些不同的部分，每个段其实就是一个小的Hashtable，它们有自己的锁。只要多个修改操作发生在不同的段上，它们就可以并发进行。

**34、LinkedBlockingQueue、ConcurrentLinkedQueue各自技术原理和使用场景是什么?**

 LinkedBlockingQueue是一个线程安全的阻塞队列，基于链表实现，一般用于生产者与消费者模型的开发中。采用锁机制来实现多线程同步，提供了一个构造方法用来指定队列的大小，如果不指定大小，队列采用默认大小（Integer.MAX\_VALUE，即整型最大值）。

ConcurrentLinkedQueue是一个线程安全的非阻塞队列，基于链表实现。java并没有提供构造方法来指定队列的大小，因此它是无界的。为了提高并发量，它通过使用更细的锁机制，使得在多线程环境中只对部分数据进行锁定，从而提高运行效率

**35、Java中如何停止一个线程?**

  使用退出标志，使线程正常退出，也就是当run方法完成后线程终止。

    2.  使用stop方法强行终止线程（这个方法不推荐使用，因为stop和suspend、resume一样，也可能发生不可预料的结果）。

1.     3.  使用interrupt方法中断线程。

**36、Java中Semaphore是什么?**

Semaphore是用来保护一个或者多个共享资源的访问，Semaphore内部维护了一个计数器，其值为可以访问的共享资源的个数。一个线程要访问共享资源，先获得信号量，如果信号量的计数器值大于1，意味着有共享资源可以访问，则使其计数器值减去1，再访问共享资源。

如果计数器值为0,线程进入休眠。当某个线程使用完共享资源后，释放信号量，并将信号量内部的计数器加1，之前进入休眠的线程将被唤醒并再次试图获得信号量。

Semaphore除了控制资源的多个副本的并发访问控制，也可以使用二进制信号量来实现类似synchronized关键字和Lock锁的并发访问控制功能。

**37、java多线程中有哪些并发流量控制工具类？**

别为CountDownLatch、CyclicBarrier、Semaphore和Exchanger，;

1、CountDownLatch，它是一种计数器的方式保证线程同步；它不去控制多个子线程之间的前后关系，只保证某一线程能够在这些子线程执行完成后再执行。

2、CyclicBarrier，通过设置屏障的方式使得多线程同步，能够控制多个线程在屏障处等等其他线程也执行到屏障点，可以实现CountDownLatch具有的功能，但是比CountDownLatch功能强大；

3、Semaphore，信号量，用于控制访问某一公共资源的并发线程数；

4、Exchanger，用于两个线程之间的数据交换。

**38、如何理解动态代理?**

   代理模式是常用的java设计模式，他的特征是代理类与委托类有同样的接口，代理类主要负责为委托类预处理消息、过滤消息、把消息转发给委托类，以及事后处理消息等。代理类与委托类之间通常会存在关联关系，一个代理类的对象与一个委托类的对象关联，代理类的对象本身并不真正实现服务，而是通过调用委托类的对象的相关方法，来提供特定的服务。简单的说就是，我们在访问实际对象时，是通过代理对象来访问的，代理模式就是在访问实际对象时引入一定程度的间接性，因为这种间接性，可以附加多种用途;

**39、什么是线程安全？**

如果你的代码所在的进程中有多个线程在同时运行，而这些线程可能会同时运行这段代码。如果每次运行结果和单线程运行的结果是一样的，而且其他的变量的值也和预期的是一样的，就是https://www.baidu.com/s?wd=%E7%BA%BF%E7%A8%8B%E5%AE%89%E5%85%A8&tn=SE\_PcZhidaonwhc\_ngpagmjz&rsv\_dl=gh\_pc\_zhidao" \t "https://zhidao.baidu.com/question/\_blank线程安全的。

**40、能举几个不是线程安全的数据结构么？**

Hashmap 、ArrayList 、LinkedList 、TreeMap、

**41、常见的多线程数据结构有哪些，你用过其中的哪些多线程数据结构？**

Vector、CopyOnWriteArrayList、CopyOnWriteArraySet、、HashTable(子类LinkedHashMap)ConcurrentHashMap

ConcurrentLinkedQueue采用的是无锁的方式，所以其性能在高并发中很好。

　　　　BlockingQueue采用的是生产者消费者模式的方式进行加锁。Blocking的具体实现有ArrayBlockingQueue和LinkedBlockingQueue

**42、多线程的常见设计模式，你用过其中的哪些设计模式**

Future模式，Master-Worker模式，生产者-消费者模型

**43、什么是Master-Worker模式？如何实现Master-Worker模式？**

Master-Worker模式是常用的并行计算模式。他的核心思想是系统由两类进程协作工作：Master进程和Worker进程.Maseter负责接收和分配任务, Worker负责处理子任务。当各个Worker子进行处理完成后，会将结果返回给Master，由Msster做归纳总结，好处是能将一个大任务分解成若干个小任务，并行执行，从而提高系统的吞吐量

**44、什么是Producer-Consumer模式？如何实现Producer-Consumer模式；**

在生产-消费模式中:通常由两类线程，即若干个生产者和若干个消费者的线程。生产者负责提交用户数据，消费者负责具体处理生产者提交的任务，在生产者和消费者之间通过共享内存缓存区进行通信。

**45、什么是Future模式？如何实现Future模式**

Future模式类似于异步请求

**46、多线程使用场景是什么？**

1、数据库的数据分析(待分析的数据太多)，数据迁移。

2、servlet多线程。

3、FTP下载，多线程操作文件。

4、数据库用到的多线程。

5、分布式计算。

6、tomcat，tomcat内部采用多线程，上百个客户端访问同一个WEB应用，tomcat接入后就是把后续的处理扔给一个新的线程来处理，这个新的线程最后调用我们的servlet程序，比如doGet或者dpPost方法。

7、后台任务：如定时向大量(100W以上)的用户发送邮件；定期更新配置文件、任务调度(如quartz)，一些监控用于定期信息采集。

8、自动作业处理：比如定期备份日志、定期备份数据库。

9、异步处理：如发微博、记录日志。

10、页面异步处理：比如大批量数据的核对工作(有10万个手机号码，核对哪些是已有用户)。

**47、多线程有优缺点？**

何时使用多线程技术,何时避免用它,是我们需要掌握的重要课题。多线程技术是一把双刃剑,在使用时需要充分考虑它的优缺点。

多线程处理可以同时运行多个线程。由于多线程应用程序将程序划分成多个独立的任务,因此可以在以下方面显著提高性能:

(1)多线程技术使程序的响应速度更快 ,因为用户界面可以在进行其它工作的同时一直处于活动状态;

(2)当前没有进行处理的任务时可以将处理器时间让给其它任务;

(3)占用大量处理时间的任务可以定期将处理器时间让给其它任务;

(4)可以随时停止任务;

(5)可以分别设置各个任务的优先级以优化性能。

 是否需要创建多个线程取决于各种因素。在以下情况下,最适合采用多线程处理:

(1)耗时或大量占用处理器的任务阻塞用户界面操作;

(2)各个任务必须等待外部资源 (如远程文件或 Internet连接)。

同样的 ,多线程也存在许多缺点 ,在考虑多线程时需要进行充分的考虑。多线程的主要缺点包括:

(1)等候使用共享资源时造成程序的运行速度变慢。这些共享资源主要是独占性的资源 ,如打印机等。

(2)对线程进行管理要求额外的 CPU开销。线程的使用会给系统带来上下文切换的额外负担。当这种负担超过一定程度时,多线程的特点主要表现在其缺点上,比如用独立的线程来更新数组内每个元素。

(3)线程的死锁。即较长时间的等待或资源竞争以及死锁等多线程症状。

(4)对公有变量的同时读或写。当多个线程需要对公有变量进行写操作时,后一个线程往往会修改掉前一个线程存放的数据,从而使前一个线程的参数被修改;另外 ,当公用变量的读写操作是非原子性时,在不同的机器上,中断时间的不确定性,会导致数据在一个线程内的操作产生错误,从而产生莫名其妙的错误,而这种错误是程序员无法预知的。

**48、假设某系统的某个接口的峰值TPS为2w/s(其它接口的并发峰值至多为200每秒)，且该接口会保存数据至数据库，如何提升该接口的性能？**

利用多线程 将并发数改成200；

创建一个任务队列里面存入要存放任务书2.1W;创建线程2个100或1个200处理请求。如果实时，处理不来可以设置超时返回错误。

**49、是否熟悉java concurrent包的内容，请讲讲concurrent包有哪些重要的内容？**

locks部分：显式锁(互斥锁和速写锁)相关；

atomic部分：原子变量类相关，是构建非阻塞算法的基础；

executor部分：线程池相关；

collections部分：并发容器相关；

tools部分：同步工具相关，如信号量、闭锁、栅栏等功能；

**50、请讲讲并发编程的CAS理论**

CAS 操作包含三个操作数 -- 内存位置、预期数值和新值。CAS 的实现逻辑是将内存位置处的数值与预期数值想比较，若相等，则将内存位置处的值替换为新值。若不相等，则不做任何操作。

**51、请讲讲并发队列和阻塞队列**

ConcurrentLinkedQueue : 是一个适用于高并发场景下的队列，通过无锁的方式，实现

了高并发状态下的高性能，通常ConcurrentLinkedQueue性能好于BlockingQueue.它

是一个基于链接节点的无界线程安全队列。该队列的元素遵循先进先出的原则。头是最先

加入的，尾是最近加入的，该队列不允许null元素。

阻塞队列（BlockingQueue）是一个支持两个附加操作的队列。

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。

BlockingQueue即阻塞队列，从阻塞这个词可以看出，在某些情况下对阻塞队列的访问可能会造成阻塞。被阻塞的情况主要有如下两种：1. 当队列满了的时候进行入队列操作2. 当队列空了的时候进行出队列操作

因此，当一个线程试图对一个已经满了的队列进行入队列操作时，它将会被阻塞，除非有另一个线程做了出队列操作；同样，当一个线程试图对一个空队列进行出队列操作时，

它将会被阻塞，除非有另一个线程进行了入队列操作。

在Java中，BlockingQueue的接口位于java.util.concurrent 包中(在Java5版本开始提供)，由上面介绍的阻塞队列的特性可知，阻塞队列是线程安全的。

在新增的Concurrent包中，BlockingQueue很好的解决了多线程中，如何高效安全“传输”数据的问题。通过这些高效并且线程安全的队列类。

**52、多线程yield方法使用于什么场景？**

Thread.yield()方法作用是：暂停当前正在执行的线程对象（及放弃当前拥有的cup资源），并执行其他线程。

yield()做的是让当前运行线程回到可运行状态，以允许具有相同优先级的其他线程获得运行机会。因此，使用yield()的

目的是让相同优先级的线程之间能适当的轮转执行。但是，实际中无法保证yield()达到让步目的，因为让步的线程还有可能被

线程调度程序再次选中。

**53、请讲讲线程异步处理的原理及关键组件？**

在http://lib.csdn.net/base/javase" \o "Java SE知识库" \t "https://www.cnblogs.com/tuojunjie/p/\_blankJava平台,实现异步调用的角色有如下三个角色：调用者、 提货单 、真实数据，一个调用者在调用耗时操作,不能立即返回数据时,先返回一个提货单

.然后在过一断时间后凭提货单来获取真正的数据.去蛋糕店买蛋糕，不需要等蛋糕做出来(假设现做要很长时间)，只需要领个提货单就可以了(去干别的

事情)，等到蛋糕做好了，再拿提货单取蛋糕就可以了。

**54、在实际项目（产品）研发过程中，你是否有使用过多线程，和线程池，如果有，请举例说明（要用STAR模型）；**

**55、什么是多线程的原子操作？Java 中有哪些原子操作？**

即不能被线程调度机制中断的操作。原子操作不需要进行同步控制。

原子操作可以是一个步骤，也可以是多个操作步骤，但是其顺序不可以被打乱，也不可以被切割而只执行其中的一部分，将整个操作视作一个整体是原子性的核心特征；

原子更新基本类型

1）除long和double之外的基本类型的赋值操作

2）所有引用reference的赋值操作

3）java.concurrent.Atomic.\* 包中所有类的一切操作

AtomicBoolean ：原子更新布尔类型

AtomicInteger： 原子更新整型

AtomicLong: 原子更新长整型

原子更新数组

AtomicIntegerArray ：原子更新整型数组里的元素

AtomicLongArray :原子更新长整型数组里的元素

AtomicReferenceArray : 原子更新引用类型数组的元素

AtomicBooleanArray ：原子更新布尔类型数组的元素

原子更新引用类型

AtomicReference ：原子更新引用类型

AtomicReferenceFieldUpdater ：原子更新引用类型里的字段

AtomicMarkableReference：原子更新带有标记位的引用类型。可以原子更新一个布尔类型的标记位和应用类型

原子更新字段类

AtomicIntegerFieldUpdater:原子更新整型的字段的更新器

AtomicLongFieldUpdater：原子更新长整型字段的更新器

AtomicStampedReference:原子更新带有版本号的引用类型。该类将整型数值与引用关联起来，可用于原子的更新数据和数据的版本号，可以解决使用CAS进行原子更新时可能出现的ABA问题。

**56、多线程的原子操作类的使用场景是什么，你在项目的实际研发过程中是否有使用过原子操作类？**

原子操作可以是一个步骤，也可以是多个操作步骤，但是其顺序不可以被打乱，也不可以被切割而只执行其中的一部分，将整个操作视作一个整体是原子性的核心特征；

原子更新基本类型

计数器 可以用原子操作

**57、如何在多个线程间共享数据？**

如果每个线程执行的代码相同，可以使用同一个Runnable对象，这个Runnable对象中有那个共享数据，例如，卖票系统就可以这么做。

将共享数据封装成另外一个对象，然后将这个对象逐一传递给各个Runnable对象，每个线程对共享数据的操作方法也分配到那个对象身上完成，这样容易实现针对数据进行各个操作的互斥和通信

将Runnable对象作为一个类的内部类，共享数据作为这个类的成员变量，每个线程对共享数据的操作方法也封装在外部类，以便实现对数据的各个操作的同步和互斥，作为内部类的各个Runnable对象调用外部类的这些方法。

**58、线程的状态有哪些，线程状态的使用场景是什么？**

　1、新状态：线程对象已经创建，还没有在其上调用start()方法。

　　2、可运行状态：当线程有资格运行，但调度程序还没有把它选定为运行线程时线程所处的状态。当start()方法调用时，线程首先进入可运行状态。在线程运行之后或者从阻塞、等待或睡眠状态回来后，也返回到可运行状态。

　　3、运行状态：线程调度程序从可运行池中选择一个线程作为当前线程时线程所处的状态。这也是线程进入运行状态的唯一一种方式。

　　4、等待/阻塞/睡眠状态：这是线程有资格运行时它所处的状态。实际上这个三状态组合为一种，其共同点是：线程仍旧是活的，但是当前没有条件运行。换句话说，它是可运行的，但是如果某件事件出现，他可能返回到可运行状态。

5、死亡态：当线程的run()方法完成时就认为它死去。这个线程对象也许是活的，但是，它已经不是一个单独执行的线程。线程一旦死亡，就不能复生。如果在一个死去的线程上调用start()方法，会抛出java.lang.IllegalThreadStateException异常。

**59、有多个线程T1，T2，T3，怎么确保它们按顺序执行？**

1）可以在线程里面加入join方法 一次等待前面的线程执行完 在执行后面的

2）用Executors.newSingleThreadExecutor();分别依次提交 开启，这样就执行有序了。

**60、volatile变量和atomic变量有什么不同？**

Volatile是让变量在所有线程中变得可见。操作时不一定保证原子性，线程安全。

Atomic是原子性，线程安全的。他的修改是sysnizied。

**61、wait/notify/notifyAll一般使用于什么场景？**

如果线程调用了对象的wait（）方法，那么线程便会处于该对象的等待池中，等待池中的线程不会去竞争该对象的锁。

   当有线程调用了对象的notifyAll（）方法（唤醒所有wait线程）或notify（）方法（只随机唤醒一个wait线程），被唤醒的的线程便会进入该对象的锁池中，锁池中的线程会去竞争该对象锁。

   优先级高的线程竞争到对象锁的概率大，假若某线程没有竞争到该对象锁，它还会留在锁池中，唯有线程再次调用wait（）方法，它才会重新回到等待池中。而竞争到对象锁的线程则继续往下执行，直到执行完了synchronized代码块，它会释放掉该对象锁，这时锁池中的线程会继续竞争该对象锁。

**62、什么是JVM ?**

JVM是Java Virtual Machine（Javahttps://baike.baidu.com/item/%E8%99%9A%E6%8B%9F%E6%9C%BA" \t "https://baike.baidu.com/item/JVM/\_blank虚拟机）的缩写，JVM是一种用于计算设备的规范，它是一个虚构出来的计算机，是通过在实际的计算机上仿真模拟各种计算机功能来实现的。

https://baike.baidu.com/item/Java%E8%AF%AD%E8%A8%80" \t "https://baike.baidu.com/item/JVM/\_blankJava语言的一个非常重要的特点就是与平台的无关性。而使用Java虚拟机是实现这一特点的关键。一般的高级语言如果要在不同的平台上运行，至少需要编译成不同的https://baike.baidu.com/item/%E7%9B%AE%E6%A0%87%E4%BB%A3%E7%A0%81/9407934" \t "https://baike.baidu.com/item/JVM/\_blank目标代码。而引入Java语言虚拟机后，Java语言在不同平台上运行时不需要重新编译。Java语言使用Java虚拟机屏蔽了与具体平台相关的信息，使得Java语言https://baike.baidu.com/item/%E7%BC%96%E8%AF%91%E7%A8%8B%E5%BA%8F/8290180" \t "https://baike.baidu.com/item/JVM/\_blank编译程序只需生成在Java虚拟机上运行的目标代码（https://baike.baidu.com/item/%E5%AD%97%E8%8A%82%E7%A0%81/9953683" \t "https://baike.baidu.com/item/JVM/\_blank字节码），就可以在多种平台上不加修改地运行。Java虚拟机在执行字节码时，把字节码解释成具体平台上的https://baike.baidu.com/item/%E6%9C%BA%E5%99%A8%E6%8C%87%E4%BB%A4/8553126" \t "https://baike.baidu.com/item/JVM/\_blank机器指令执行。这就是Java的能够“一次编译，到处运行”的原因。

**63、Java中堆和栈有什么区别？**

各司其职：

最主要的区别就是栈内存用来存储局部变量和方法调用。

而堆内存用来存储Java中的对象。无论是成员变量，局部变量，还是类变量，它们指向的对象都存储在堆内存中。

独有还是共享：

栈内存归属于单个线程，每个线程都会有一个栈内存，其存储的变量只能在其所属线程中可见，即栈内存可以理解成线程的私有内存。

而堆内存中的对象对所有线程可见。堆内存中的对象可以被所有线程访问。

异常错误：

如果栈内存没有可用的空间存储方法调用和局部变量，JVM会抛出java.lang.StackOverFlowError。

而如果是堆内存没有可用的空间存储生成的对象，JVM会抛出java.lang.OutOfMemoryError。

空间大小：

栈的内存要远远小于堆内存，如果你使用递归的话，那么你的栈很快就会充满。如果递归没有及时跳出，很可能发生StackOverFlowError问题。

你可以通过-Xss选项设置栈内存的大小。-Xms选项可以设置堆的开始时的大小，-Xmx选项可以设置堆的最大值。

这就是Java中堆和栈的区别。理解好这个问题的话，可以对你解决开发中的问题，分析堆内存和栈内存使用，甚至性能调优都有帮助。

**64、请说说jvm的基本结构？**

它包括：类加载器子系统、运行时数据区、执行引擎和本地方法接口。

运行时数据区是JVM从操作系统申请来的堆空间和操作系统给JVM分配的栈空间的总称。JVM为了运行Java程序，又进一步对运行时数据区进行了划分，划分为Java方法区、Java堆、Java栈、PC寄存器、本地方法栈等，这里JVM从操作系统申请来的堆空间被划分为方法区和Java堆，操作系统给JVM分配的栈空间构成Java栈。

**65、堆空间的结构**

运行时数据区中Java的方法区和Java堆（图中显示的是：永久、新生、老年，这是分代垃圾回收时的术语，实际上永久代和Java方法区对应，https://www.baidu.com/s?wd=%E6%96%B0%E7%94%9F%E4%BB%A3&tn=24004469\_oem\_dg&rsv\_dl=gh\_pl\_sl\_csd" \t "https://blog.csdn.net/zhangzl4321/article/details/\_blank新生代和老年代和Java堆对应），也就说Java方法区和Java堆其实都是JVM堆的一部分。JVM的栈区构成了Java的线程栈。

**66、为何新生代要设置两个survivor区，jvm的设计上有何目的？**

1、Survivor的存在意义，就是减少被送到老年代的对象，进而减少Full GC的发生，Survivor的预筛选保证，只有经历16次Minor GC还能在新生代中存活的对象，才会被送到老年代。

2、设置两个Survivor区最大的好处就是解决了碎片化，永远有一个survivor space是空的，另一个非空的survivor space无碎片。

设计目的：

应该建立两块Survivor区，刚刚新建的对象在Eden中，经历一次Minor GC，Eden中的存活对象就会被移动到第一块survivor space S0，Eden被清空；等Eden区再满了，就再触发一次Minor GC，Eden和S0中的存活对象又会被复制送入第二块survivor space S1（这个过程非常重要，因为这种复制算法保证了S1中来自S0和Eden两部分的存活对象占用连续的内存空间，避免了碎片化的发生）。S0和Eden被清空，然后下一轮S0与S1交换角色，如此循环往复。如果对象的复制次数达到16次，该对象就会被送到老年代中。下图中每部分的意义和上一张图一样，就不加注释了。

**67、垃圾回收中的复制算法适用于在什么场景下使用？**

将内存分为（大小相等）两部分，每次只使用其中一块进行内存分配，当内存使用完后，就出发GC，将存活的对象直接复制到另一块空闲的内存中，然后对当前使用的内存块一次性清除所有，然后转到另一块内存进行使用。

优点：简单，高效。

缺点：浪费内存，因为每次都有另一块内存空闲着。

Eden survivor 垃圾回收

**68、老年代的垃圾回收一般用什么算法？**

标记-压缩-清理算法进行垃圾回收，将标记对象移动到堆的另一端，同时更新对象的引用地址

1、mark\_sweep\_phase1： 标记活跃对象

2、mark\_sweep\_phase2： 计算活跃对象在压缩完成之后的新地址

**69、怎么获取 Java 程序使用的内存？堆使用的百分比？**

jhat：内存分析工具：

主要是对java应用程序的资源和性能进行实时的命令行监控，包括了对heap size和垃圾回收状况的监控。

 jstat -<option> [-t] [-h<lines>] <vmid> [<interval> [<count>]]

option：我们经常使用的选项有gc、gcutil

vmid：java进程id

interval：间隔时间，单位为毫秒

count：打印次数

jmap -heap pid:查看堆使用情况

jmap -histo pid：查看堆中对象数量和大小

jmap -dump:format=b,file=heapdump pid：将内存使用的详细情况输出到文件

序列号、Class实例的数量、内存的占用、类限定名

如果是内部类，类名的开头会加上\*，如果加上live子参数的话，如jmap -histo：live pid，这个命名会触发一次FUll GC，只统计存活对象

**70、GC回收机制？**

**71、jmap命令是有什么用途？jstat命令是有什么用途？**

https://www.baidu.com/s?wd=Jmap&tn=24004469\_oem\_dg&rsv\_dl=gh\_pl\_sl\_csd" \t "\_blankmap是一个可以输出所有内存中对象的工具，甚至可以将VM 中的heap，以二进制输出成文本。打印出某个java进程（使用pid）内存内的，

Jstat是JDK自带的一个轻量级小工具。全称“Java Virtual Machine statistics monitoring tool”，它位于java的bin目录下，主要利用JVM内建的指令对Java应用程序的资源和性能进行实时的命令行的监控，包括了对Heap size和垃圾回收状况的监控。可见，Jstat是轻量级的、专门针对JVM的工具，非常适用。

**72、有哪些常见的jvm命令，说说各自的用途是什么？**

**Jps:JVM Process Status Tool,显示指定系统内所有的HotSpot虚拟机进程**

Jstat: 是用于监视虚拟机运行时状态信息的命令，它可以显示出虚拟机进程中的类装载、内存、垃圾收集、JIT编译等运行数据。

Jmap:(JVM Memory Map)命令用于生成heap dump文件，如果不使用这个命令，还阔以使用-XX:+HeapDumpOnOutOfMemoryError参数来让虚拟机出现OOM的时候·自动生成dump文件。

jmap不仅能生成dump文件，还阔以查询finalize执行队列、Java堆和永久代的详细信息，如当前使用率、当前使用的是哪种收集器等。

jhat(JVM Heap Analysis Tool)命令是与jmap搭配使用，用来分析jmap生成的dump，jhat内置了一个微型的HTTP/HTML服务器，生成dump的分析结果后，可以在浏览器中查看。在此要注意，一般不会直接在服务器上进行分析，因为jhat是一个耗时并且耗费硬件资源的过程，一般把服务器生成的dump文件复制到本地或其他机器上进行分析。

jstack用于生成java虚拟机当前时刻的线程快照。线程快照是当前java虚拟机内每一条线程正在执行的方法堆栈的集合，生成线程快照的主要目的是定位线程出现长时间停顿的原因，如线程间死锁、死循环、请求外部资源导致的长时间等待等。 线程出现停顿的时候通过jstack来查看各个线程的调用堆栈，就可以知道没有响应的线程到底在后台做什么事情，或者等待什么资源。 如果java程序崩溃生成core文件，jstack工具可以用来获得core文件的java stack和native stack的信息，从而可以轻松地知道java程序是如何崩溃和在程序何处发生问题。另外，jstack工具还可以附属到正在运行的java程序中，看到当时运行的java程序的java stack和native stack的信息, 如果现在运行的java程序呈现hung的状态，jstack是非常有用的。

info(JVM Configuration info)这个命令作用是实时查看和调整虚拟机运行参数。

之前的jps -v口令只能查看到显示指定的参数，如果想要查看未被显示指定的参数的值就要使用jinfo口令

**73、、GC有哪些算法（\*\*\*\*\*）**

引用计数,没有被Java采用

标记-清除

标记-压缩 标记-整理算法

复制算法 新生代

**74、、什么是线程中断。tips: stop the world，简称STW，参考billy1.GC算法与种类**

Java中一种全局暂停的现象

全局停顿，所有Java代码停止，native代码可以执行，但不能和JVM交互

**75、MGC、FGC分别是什么意思，它们在什么情况下会发生**

，YG用来放新产生的对象，经过几次回收还没回收掉的对象往OG中移动，对YG进行垃圾回收又叫做MinorGC，对 OG垃圾回收又叫MajorGC，.

1、当eden满了，触发young GC；

2.young GC做2件事：一，去掉一部分没用的object；二，把老的还被引用的object发到survior里面，等下几次GC以后，survivor再放到old里面。

3.当old满了，触发full GC。full GC很消耗内存，把old，young里面大部分垃圾回收掉。这个时候用户线程都会被block。

**76、、请讲讲jvm的分代，为什么要分代，jvm分代有什么好处？**

虚拟机中的共划分为三个代：年轻代（Young Generation）、年老点（Old Generation）和持久代（Permanent Generation）。其中持久代主要存放的是Java类的类信息，与垃圾收集要收集的Java对象关系不大。年轻代和年老代的划分是对垃圾收集影响比较大的。

利用对象存活的生命不同。利用的算法不同。

所有新生成的对象首先都是放在年轻代的。年轻代的目标就是尽可能快速的收集掉那些生命周期短的对象。年轻代分三个区。一个Eden区，两个Survivor区(一般而言)。大部分对象在Eden区中生成。当Eden区满时，还存活的对象将被复制到Survivor区（两个中的一个），当这个Survivor区满时，此区的存活对象将被复制到另外一个Survivor区，当这个Survivor去也满了的时候，从第一个Survivor区复制过来的并且此时还存活的对象，将被复制“年老区(Tenured)”。需要注意，Survivor的两个区是对称的，没先后关系，所以同一个区中可能同时存在从Eden复制过来 对象，和从前一个Survivor复制过来的对象，而复制到年老区的只有从第一个Survivor去过来的对象。而且，Survivor区总有一个是空的。同时，根据程序需要，Survivor区是可以配置为多个的（多于两个），这样可以增加对象在年轻代中的存在时间，减少被放到年老代的可能。

**77、、你知道哪些jvm调优工具么？**

uptime 系统时间 运行时间 连接数 1,5,15分钟内的系统平均负载

top

vmstat可以统计系统的CPU，内存，swap，io等情况

pidstat细致观察进程

任务管理器

Perfmon

Process Explorer

pslist

**78、、在jvm中，年轻代如何向老年代转变的？年轻代向老年代转换的重要参数是什么?**

当eden满了，触发young GC；

2.young GC做2件事：一，去掉一部分没用的object；二，把老的还被引用的object发到survior里面，等下几次GC以后，survivor再放到old里面。

3.当old满了，触发full GC。full GC很消耗内存，把old，young里面大部分垃圾回收掉。这个时候用户线程都会被block。

-Xms 和 -Xmx (-XX:InitialHeapSize 和 -XX:MaxHeapSize)：指定JVM初始占用的堆内存和最大堆内存。JVM也是一个软件，也必须要获取本机的物理内

-XX:MaxTenuringThreshold= 设置熬过年轻代多少次收集后移入老人区，CMS中默认为0，熬过第一次GC就转入，可以用-XX:+PrintTenuringDistribution 查看

调用16次

**79、、直接内存使用场景是什么，使用直接内存可能会存在什么问题？tips**

80、、堆内存有哪些重要参数？

默认空余堆内存小于40%时，JVM就会增大堆直到-Xmx的最大限制，可以由 -XX:MinHeapFreeRatio=指定。

默认空余堆内存大于70%时，JVM会减少堆直到-Xms的最小限制，可以由 -XX:MaxHeapFreeRatio=指定。

服务器一般设置-Xms、-Xmx相等以避免在每次GC后调整堆的大小，所以上面的两个参数没啥用。

**oung(Nursery)：年轻代**

研究表明大部分对象都是朝生暮死，随生随灭的。所以对于年轻代在GC时都采取复制收集算法，具体算法参考下面的描述；

Young的默认值为4M，随堆内存增大，约为1/15，JVM会根据情况动态管理其大小变化。

Young里面又分为3 个区域，一个Eden，所有新建对象都会存在于该区，两个Survivor区，用来实施复制算法。

-XX:NewRatio= 参数可以设置Young与Old的大小比例，-server时默认为1:2，但实际上young启动时远低于这个比率？如果信不过JVM，也可以用 -Xmn硬性规定其大小，有文档推荐设为Heap总大小的1/4。

-XX:SurvivorRatio= 参数可以设置Eden与Survivor的比例，默认为32。Survivio大了会浪费，小了的话，会使一些年轻对象潜逃到老人区，引起老人区的不安，但这个参数对性能并不太重要。

**Old(Tenured)：年老代**

年轻代的对象如果能够挺过数次收集，就会进入老人区。老人区使用标记整理算法。因为老人区的对象都没那么容易死的，采用复制算法就要反复的复制对象，很不合算，只好采用标记清理算法，但标记清理算法其实也不轻松，每次都要遍历区域内所有对象，所以还是没有免费的午餐啊。

-XX:MaxTenuringThreshold= 设置熬过年轻代多少次收集后移入老人区，CMS中默认为0，熬过第一次GC就转入，可以用-XX:+PrintTenuringDistribution 查看。

**Permanent：持久代**

装载Class信息等基础数据，默认64M，如果是类很多很多的服务程序，需要加大其设置 -XX:MaxPermSize=，否则它满了之后会引起fullgc()或Out of Memory。 注意Spring，Hibernate这类喜欢AOP动态生成类的框架需要更多的持久代内存。一般情况下，持久代是不会进行GC的，除非通过 -XX:+CMSClassUnloadingEnabled -XX:+CMSPermGenSweepingEnabled进行强制设置。

81、如何设置堆大小，是否有一些经验值？

JVM 中最大堆大小有三方面限制：相关操作系统的数据模型（32-bt还是64-bit）限制；系统的可用虚拟内存限制；系统的可用物理内存限制。32位系统 下，一般限制在1.5G~2G；64为操作系统对内存无限制。我在Windows Server 2003 系统，3.5G物理内存，JDK5.0下测试，最大可设置为1478m。

**典型JVM参数设置：**

java -Xmx3550m -Xms3550m -Xmn2g -Xss128k

-Xmx3550m：设置JVM最大可用内存为3550M。

-Xms3550m：设置JVM促使内存为3550m。此值可以设置与-Xmx相同，以避免每次垃圾回收完成后JVM重新分配内存。

-Xmn2g：设置年轻代大小为2G。整个堆大小=年轻代大小 + 年老代大小 + 持久代大小。持久代一般固定大小为64m，所以增大年轻代后，将会减小年老代大小。此值对系统性能影响较大，Sun官方推荐配置为整个堆的3/8。

-Xss128k：设置每个线程的堆栈大小。JDK5.0以后每个线程堆栈大小为1M，以前每个线程堆栈大小为256K。更具应用的线程所需内存大小进行 调整。在相同物理内存下，减小这个值能生成更多的线程。但是操作系统对一个进程内的线程数还是有限制的，不能无限生成，经验值在 3000~5000 左右。

java -Xmx3550m -Xms3550m -Xss128k -XX:NewRatio=4 -XX:SurvivorRatio=4 -XX:MaxPermSize=16m -XX:MaxTenuringThreshold=0

-XX:NewRatio=4:设置年轻代（包括Eden和两个Survivor区）与年老代的比值（除去持久代）。设置为4，则年轻代与年老代所占比值为1：4，年轻代占整个堆栈的1/5

-XX:SurvivorRatio=4：设置年轻代中Eden区与Survivor区的大小比值。设置为4，则两个Survivor区与一个 Eden区的比值为2:4，一个Survivor区占整个年轻代的1/6

-XX:MaxPermSize=16m:设置持久代大小为16m。

-XX:MaxTenuringThreshold=0：设置垃圾最大年龄。如果设置为0的话，则年轻代对象不经过Survivor区，直接进入年老代。 对于年老代比较多的应用，可以提高效率。如果将此值设置为一个较大值，则年轻代对象会在Survivor区进行多次复制，这样可以增加对象再年轻代的存活 时间，增加在年轻代即被回收的概论。

82、如何打印JVM日志？

-XX:+PrintGCDetails -Xloggc:../logs/gc.log -XX:+PrintGCTimeStamps

83、请介绍常见的jvm参数

-XX:+PrintGCTimeStamps:

打印此次垃圾回收距离jvm开始运行的所耗时间

-XX:+PrintGCDeatils

打印垃圾回收的细节信息

-Xloggc:<filename>

将垃圾回收信息输出到指定文件

-XX:+PrintGCDateStamps

需要打印日历形式的时间戳选项

-XX:+PrintGCApplicationStoppedTime

-XX:+PrintGCApplicationConcurrentTime

打印应用程序由于执行VM安全点操作而阻塞的时间以及两个安全点操作之间应用程序的运行时间

-XX:+PrintSafepointStatistics

可以将垃圾回收的安全点与其他的安全点区分开

ν -XX:+UseSerialGC：在新生代和老年代使用串行收集器

ν -XX:SurvivorRatio：设置eden区大小和survivior区大小的比例

ν -XX:NewRatio:新生代和老年代的比

ν -XX:+UseParNewGC：在新生代使用并行收集器

ν -XX:+UseParallelGC ：新生代使用并行回收收集器

ν -XX:+UseParallelOldGC：老年代使用并行回收收集器

ν -XX:ParallelGCThreads：设置用于垃圾回收的线程数

ν -XX:+UseConcMarkSweepGC：新生代使用并行收集器，老年代使用CMS+串行收集器

ν -XX:ParallelCMSThreads：设定CMS的线程数量

ν -XX:CMSInitiatingOccupancyFraction：设置CMS收集器在老年代空间被使用多少后触发

ν -XX:+UseCMSCompactAtFullCollection：设置CMS收集器在完成垃圾收集后是否要进行一次内存碎片的整理

ν -XX:CMSFullGCsBeforeCompaction：设定进行多少次CMS垃圾回收后，进行一次内存压缩

ν -XX:+CMSClassUnloadingEnabled：允许对类元数据进行回收

ν -XX:CMSInitiatingPermOccupancyFraction：当永久区占用率达到这一百分比时，启动CMS回收

ν -XX:UseCMSInitiatingOccupancyOnly：表示只在到达阀值的时候，才进行CMS回收

84、CMS收集器有什么特点？

尽可能降低停顿

会影响系统整体吞吐量和性能

比如，在用户线程运行过程中，分一半CPU去做GC，系统性能在GC阶段，反应速度就下降一半

清理不彻底

因为在清理阶段，用户线程还在运行，会产生新的垃圾，无法清理

因为和用户线程一起运行，不能在空间快满时再清理

85、G1收集器有什么特点？

并行于并发：G1能充分利用CPU、多核环境下的硬件优势，使用多个CPU（CPU或者CPU核心）来缩短stop-The-World停顿时间。部分其他收集器原本需要停顿Java线程执行的GC动作，G1收集器仍然可以通过并发的方式让java程序继续执行。

2、分代收集：虽然G1可以不需要其他收集器配合就能独立管理整个GC堆，但是还是保留了分代的概念。它能够采用不同的方式去处理新创建的对象和已经存活了一段时间，熬过多次GC的旧对象以获取更好的收集效果。

3、空间整合：与CMS的“标记--清理”算法不同，G1从整体来看是基于“标记整理”算法实现的收集器；从局部上来看是基于“复制”算法实现的。

4、可预测的停顿：这是G1相对于CMS的另一个大优势，降低停顿时间是G1和ＣＭＳ共同的关注点，但Ｇ１除了追求低停顿外，还能建立可预测的停顿时间模型，能让使用者明确

86、垃圾回收器有哪些？

串行垃圾回收器（Serial Garbage Collector）

并行垃圾回收器（Parallel Garbage Collector）

并发标记扫描垃圾回收器（CMS Garbage Collector）

G1垃圾回收器（G1 Garbage Collector）

87、java内存模型

1、主内存和工作内存

    （1）所有变量均存储在主内存（虚拟机内存的一部分）

    （2）每个线程都对应着一个工作线程，主内存中的变量都会复制一份到每个线程的自己的工作空间，线程对变量的操作都在自己的工作内存中，操作完成后再将变量更新至主内存；

    （3）其他线程再通过主内存来获取更新后的变量信息，即线程之间的交流通过主内存来传递

Note：JMM的空间划分和JVM的内存划分不一样，非要对应的话，关系如下：

    （1）JMM的主内存对应JVM中的堆内存对象实例数据部分

    （2）JMM的工作内存对应JVM中栈中部分区域

88、什么是类加载器，类加载器有哪些，类加载器的加载顺序是什么？

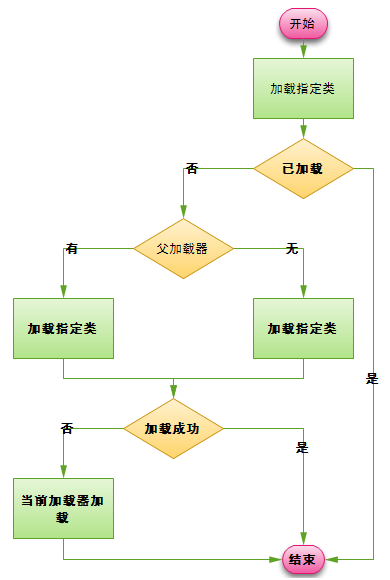
类加载器是一个用来加载类文件的类。Java源代码通过javac编译器编译成类文件。然后JVM来执行类文件中的字节码来执行程序。类加载器负责加载文件系统、网络或其他来源的类文件。有三种默认使用的类加载器：Bootstrap类加载器、Extension类加载器和System类加载器（或者叫作Application类加载器）。

1）Bootstrap类加载器 – JRE/lib/rt.jar

2) Extension类加载器 – JRE/lib/ext或者java.ext.dirs指向的目录

3) Application类加载器 – CLASSPATH环境变量, 由-classpath或-cp选项定义,或者是JAR中的Manifest的classpath属性定义.

VM并不是把所有的类一次性全部加载到JVM中的，也不是每次用到一个类的时候都去查找，对于JVM级别的类加载器在启动时就会把默认的JAVA\_HOME/lib里的class文件加载到JVM中，因为这些是系统常用的类，对于其他的第三方类，则采用用到时就去找，找到了就缓存起来的，下次再用到这个类的时候就可以直接用缓存起来的类对象了，ClassLoader之间也是有父子关系的，没个ClassLoader都有一个父ClassLoader,在加载类时ClassLoader与其父ClassLoader的查找



89、简述java内存分配与回收策略

1、 当eden满了，触发young GC；

2.young GC做2件事：一，去掉一部分没用的object；二，把老的还被引用的object发到survior里面，等下几次GC以后，survivor再放到old里面。

3.当old满了，触发full GC。full GC很消耗内存，把old，young里面大部分垃圾回收掉。这个时候用户线程都会被block。

•Tips：eden、Sruvivor、老年代、永久代（元空间）

90、JDK1.8之后Perm Space有哪些变动? MetaSpace大小默认是无限的么? 还是你们会通过什么方式来指定大小?

1、 JDK 1.8后用元空间替代了 Perm Space；字符串常量存放到堆内存中。

2、 MetaSpace大小默认没有限制，一般根据系统内存的大小。JVM会动态改变此值。

3、 -XX:MetaspaceSize：分配给类元数据空间（以字节计）的初始大小（Oracle逻辑存储上的初始高水位，the initial high-water-mark）。此值为估计值，MetaspaceSize的值设置的过大会延长垃圾回收时间。垃圾回收过后，引起下一次垃圾回收的类元数据空间的大小可能会变大。

4、 -XX:MaxMetaspaceSize：分配给类元数据空间的最大值，超过此值就会触发Full GC，此值默认没有限制，但应取决于系统内存的大小。JVM会动态地改变此值。

91、Perm Space中保存什么数据？会引起OutOfMemory吗？

加载class文件。

会引起，出现异常可以设置 -XX:PermSize 的大小。JDK 1.8后，字符串常量不存放在永久带，而是在堆内存中，JDK8以后没有永久代概念，而是用元空间替代，元空间不存在虚拟机中，二是使用本地内存。

详细查看Java8内存模型—永久代(PermGen)和元空间(Metaspace)

92、java类加载全过程，从架构角度理解，类加载和反射、动态代理有什么关系？

93、简述java类加载机制？tips：看ClassLoader源码讲解类加载机制，理解记忆

**加载----验证----准备----解析-----初始化----使用-----卸载**

94、GC收集器有哪些？CMS收集器与G1收集器的特点

串行垃圾回收器（Serial Garbage Collector）

并行垃圾回收器（Parallel Garbage Collector）

并发标记扫描垃圾回收器（CMS Garbage Collector）

G1垃圾回收器（G1 Garbage Collector）

95、类加载器双亲委派模型机制，“双亲委派”中的双亲是什么意思？tips：演示ClassLoaderTest2，讲解双亲委派流程图

如果一个类加载器收到类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器完成。每个类加载器都是如此，只有当父加载器在自己的搜索范围内找不到指定的类时（即ClassNotFoundException），子加载器才会尝试自己去加载。

96、什么情况下会出现永久代内存溢出，如何解决此类问题？

生成大量的类，增大Perm区 允许Class回收

97、什么情况下会出现堆内存溢出，如何解决此类问题？

占用大量堆空间，直接溢出

增大堆空间，及时释放内存

98、什么情况下会出现直接内存溢出，如何解决此类问题？

ByteBuffer.allocateDirect()无法从操作系统获得足够的空间

解决方法：**减少**堆内存 有意触发GC

99、什么情况下会出现过多线程导致内存溢出的问题，如何解决此类问题？

多线程 没有释放内存

– 1、OOM由于保存多线程过多引起，可以考虑增加堆大小

– 2. 如果应用允许，缩短多线程的过期时间，使得session可以及时过期，并回收

100、什么情况下会出现CPU使用率过高的问题，如何解决此类问题？

多线程竞争资源，多线程上下文切换太频繁

合理设置线程最大开启数量，并发数量

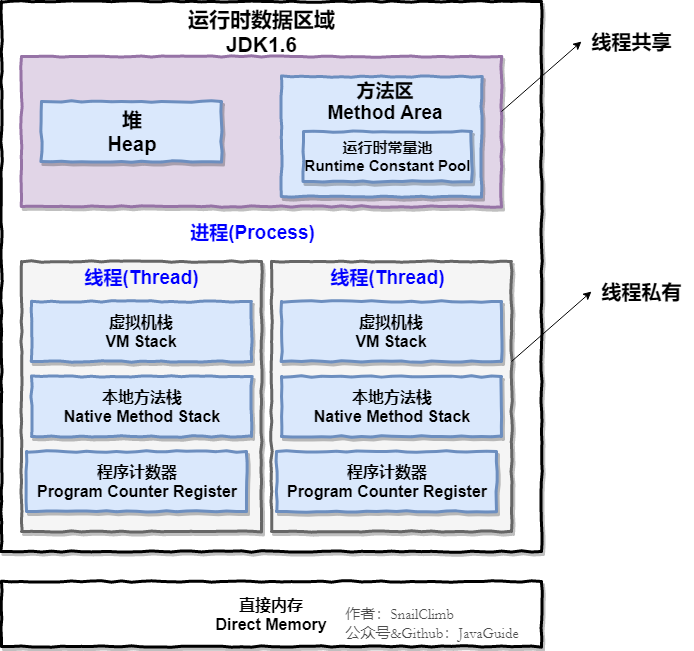
[**多线程面试题总结**](https://www.cnblogs.com/toria/p/11234323.html)

**1、什么是线程和进程?**

**进程**：在操作系统中能够独立运行，并且作为资源分配的基本单位。它表示运行中的程序。系统运行一个程序就是一个进程从创建、运行到消亡的过程。

**线程**：是一个比进程更小的执行单位，能够完成进程中的一个功能，也被称为轻量级进程。一个进程在其执行的过程中可以产生多个线程。

【注】线程与进程不同的是：同类的多个线程**共享进程的堆和方法区资源**，但**每个线程有自己的程序计数器、虚拟机栈和本地方法栈**，所以系统在产生一个线程，或是在各个线程之间作切换工作时，负担要比进程小得多。



**为什么程序计数器、虚拟机栈和本地方法栈是线程私有的呢？为什么堆和方法区是线程共享的呢？**

* **程序计数器为什么是私有的?**

程序计数器主要有下面两个作用：

* 1. 字节码解释器通过改变程序计数器来依次读取指令，从而实现代码的流程控制，如：顺序执行、选择、循环、异常处理。
  2. 在多线程的情况下，程序计数器用于记录当前线程执行的位置，从而当线程被切换回来的时候能够知道该线程上次运行到哪儿了。

（需要注意的是，如果执行的是 native 方法，那么程序计数器记录的是 undefined 地址，只有执行的是 Java 代码时程序计数器记录的才是下一条指令的地址。）

所以，程序计数器私有主要是为了**线程切换后能恢复到正确的执行位置**。

* **虚拟机栈和本地方法栈为什么是私有的?**
* **虚拟机栈**： **每个 Java 方法在执行的同时会创建一个栈帧用于存储局部变量表、操作数栈、常量池引用等信息。从方法调用直至执行完成的过程，就对应着一个栈帧在 Java 虚拟机栈中入栈和出栈的过程**。
* **本地方法栈**： 和虚拟机栈所发挥的作用非常相似，区别是： 虚拟机栈为虚拟机执行 Java 方法 （也就是字节码）服务，而本地方法栈则为虚拟机使用到的 Native 方法服务。 在 HotSpot 虚拟机中和 Java 虚拟机栈合二为一。

所以，为了**保证线程中的局部变量不被别的线程访问到**，虚拟机栈和本地方法栈是线程私有的。

**堆和方法区是所有线程共享的资源**，其中：

* **堆是进程中最大的一块内存，主要用于存放新创建的对象 (所有对象都在这里分配内存)**。
* **方法区主要用于存放已被加载的类信息、常量、静态变量、即时编译器编译后的代码等数据**。

**2、什么是上下文切换?**

　　即使单核处理器也支持多线程执行代码，**CPU通过给每个线程分配CPU时间片来实现这个机制**。时间片是CPU分配给各个线程的时间，因为时间片非常短，所以CPU通过不停地切换线程执行，让我们感觉多个线程是同时执行的。（时间片一般是几十毫秒）

　　CPU通过时间片分配算法来循环执行任务，当前任务执行一个时间片后会切换到下一个任务。但是，在切换前会保存上一个任务的状态，以便下次切换回这个任务时，可以再加载这个任务的状态。所以**任务从保存到加载的过程就是一次上下文切换**。**上下文切换会影响多线程的执行速度**。

**3、并发与并行？**

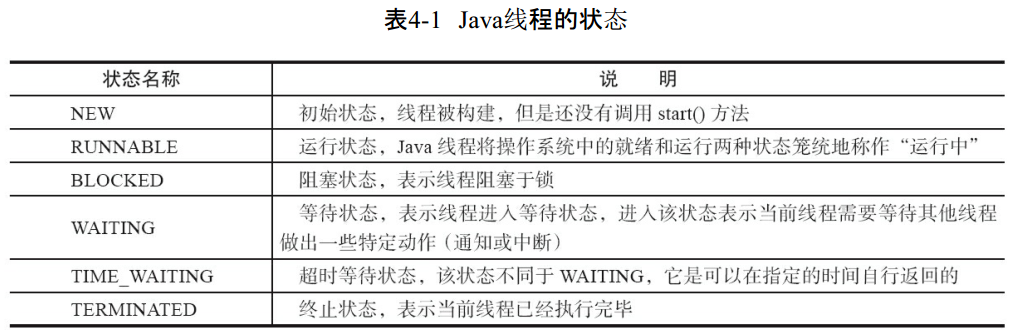
**并发指的是多个任务交替进行，并行则是指真正意义上的“同时进行”**。

　　实际上，如果系统内只有一个CPU，使用多线程时，在真实系统环境下不能并行，只能通过切换时间片的方式交替进行，从而并发执行任务。真正的并行只能出现在拥有多个CPU的系统中。

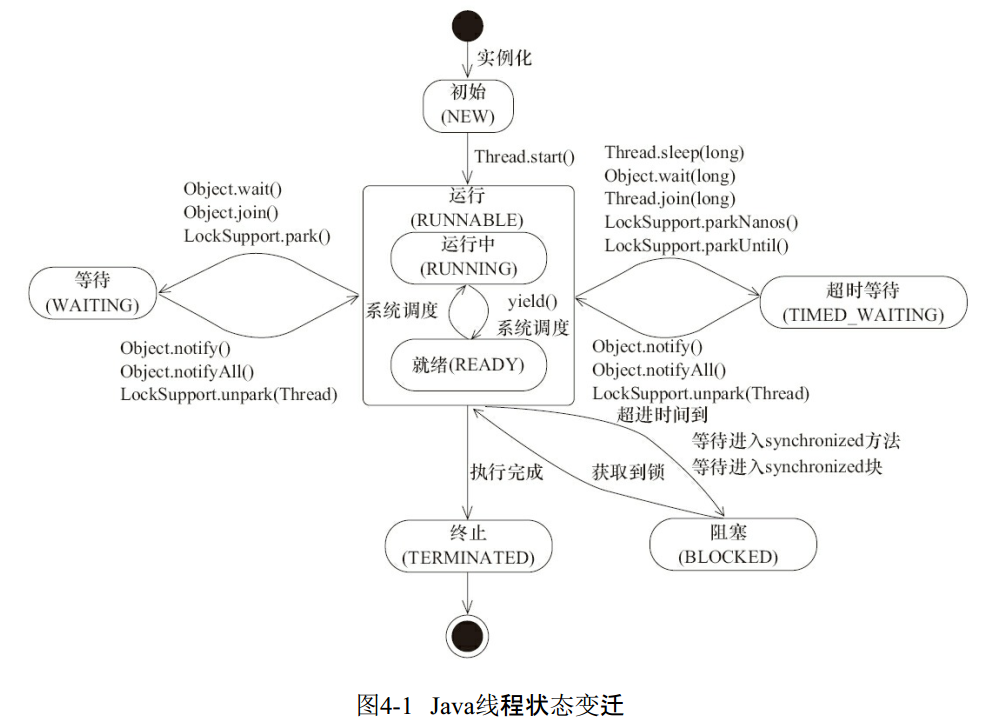
**4、线程的生命周期和状态?（重要！）**

 Java 线程在运行的生命周期中的指定时刻只可能处于下面 6 种不同状态的其中一个状态：

**初始状态、运行状态、阻塞状态、等待状态、超时等待状态、终止状态**



**线程在生命周期中并不是固定处于某一个状态而是随着代码的执行在不同状态之间切换**：



* 线程创建之后它将处于 **初始状态**(NEW)，**调用 start() 方法后开始运行，线程这时候处于 可运行状态**(READY)。
* **可运行状态的线程获得了 CPU 时间片后就处于 运行状态**(RUNNING)。
* 当**线程执行 wait()方法之后，线程进入 等待状态**(WAITING)，**进入等待状态的线程需要依靠其他线程的通知才能够返回到运行状态**【notify()】。 而 **超时等待状态**(TIME\_WAITING)**相当于在等待状态的基础上增加了超时限制，【sleep（long millis）/wait（long millis）】，当超时时间到达后 Java 线程将会返回到运行状态**。
* **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)。
* **线程在执行 Runnable 的run()方法之后将会进入到** **终止状态**(TERMINATED)。

**5、什么是线程死锁?如何避免死锁?**

 　　多个线程同时被阻塞，它们中的一个或者全部都在等待某个资源被释放。由于线程被无限期地阻塞，因此程序不可能正常终止。

　　假如线程 A 持有资源 2，线程 B 持有资源 1，他们同时都想申请对方的资源，所以这两个线程就会互相等待而进入死锁状态。

**避免死锁的几个常见方法：**

* **避免一个线程同时获取多个锁**
* **避免一个线程在锁内同时占用多个资源，尽量保证每个锁只占用一个资源。**
* **尝试使用定时锁，使用 lock.tryLock(timeout) 来代替使用内部锁机制。**
* **对于数据库锁，加锁和解锁必须在一个数据库连接里，否则会出现解锁失败的情况。**

**6、sleep() 方法和 wait() 方法区别和共同点?（重要！）**

**相同点：**

　　两者都可以暂停线程的执行，都会让线程进入等待状态。

**不同点：**

* **sleep()方法没有释放锁，而 wait()方法释放了锁。**
* **sleep()方法属于Thread类的静态方法，作用于当前线程；而wait()方法是Object类的实例方法，作用于对象本身。**
* **执行sleep()方法后，可以通过超时或者调用interrupt()方法唤醒休眠中的线程；执行wait()方法后，通过调用notify()或notifyAll()方法唤醒等待线程。**

**7、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？**

**new 一个 Thread，线程进入初始状态；调用 start()方法，会启动一个线程并使线程进入了就绪状态，当分配到时间片后就可以开始运行了。 start() 会执行线程的相应准备工作，然后自动执行 run() 方法的内容，这是真正的多线程工作**。 而**直接执行 run() 方法，会把 run 方法当成一个 main 线程下的普通方法去执行，并不会在某个线程中执行它，所以这并不是多线程工作**。

**总结： 调用 start 方法可启动线程并使线程进入就绪状态，而 run 方法只是 thread 的一个普通方法调用，还是在主线程里执行**。

**8、多线程开发带来的问题与解决方法？（重要）**

使用多线程主要会带来以下几个问题：

**（一）线程安全问题**

　　线程安全问题**指的是在某一线程从开始访问到结束访问某一数据期间，该数据被其他的线程所修改，那么对于当前线程而言，该线程就发生了线程安全问题，表现形式为数据的缺失，数据不一致等**。

　　线程安全问题发生的条件：

　　　　1）多线程环境下，即存在包括自己在内存在有多个线程。

　　　　2）多线程环境下存在共享资源，且多线程操作该共享资源。

　　　　3）多个线程必须对该共享资源有非原子性操作。

　　线程安全问题的解决思路：

**1）尽量不使用共享变量，将不必要的共享变量变成局部变量来使用**。

**2）使用synchronized关键字同步代码块，或者使用jdk包中提供的Lock为操作进行加锁**。

**3）使用ThreadLocal为每一个线程建立一个变量的副本，各个线程间独立操作，互不影响**。

**（二）性能问题**

　　线程的生命周期开销是非常大的，**一个线程的创建到销毁都会占用大量的内存**。同时如果不合理的创建了多个线程，cup的处理器数量小于了线程数量，那么将会有很多的线程被闲置，闲置的线程将会占用大量的内存，为垃圾回收带来很大压力，同时cup在分配线程时还会消耗其性能。

　　解决思路：

**利用线程池**，模拟一个池，预先创建有限合理个数的线程放入池中，当需要执行任务时从池中取出空闲的先去执行任务，执行完成后将线程归还到池中，这样就**减少了线程的频繁创建和销毁，节省内存开销**和减小了垃圾回收的压力。同时因为任务到来时本身线程已经存在，减少了创建线程时间，提高了执行效率，而且合理的创建线程池数量还会使各个线程都处于忙碌状态，提高任务执行效率，线程池还提供了拒绝策略，当任务数量到达某一临界区时，线程池将拒绝任务的进入，保持现有任务的顺利执行，减少池的压力。

**（三）活跃性问题**

　　1）**死锁**，**假如线程 A 持有资源 2，线程 B 持有资源 1，他们同时都想申请对方的资源，所以这两个线程就会互相等待而进入死锁状态**。多个线程环形占用资源也是一样的会产生死锁问题。

　　解决方法：

* **避免一个线程同时获取多个锁**
* **避免一个线程在锁内同时占用多个资源，尽量保证每个锁只占用一个资源。**
* **尝试使用定时锁，使用 lock.tryLock(timeout) 来代替使用内部锁机制。**

　　想要避免死锁，可以**使用无锁函数**（cas）**或者使用重入锁（ReentrantLock），通过重入锁使线程中断或限时等待可以有效的规避死锁问题**。

　　2）**饥饿**，饥饿**指的是某一线程或多个线程因为某些原因一直获取不到资源，导致程序一直无法执行**。如某一线程优先级太低导致一直分配不到资源，或者是某一线程一直占着某种资源不放，导致该线程无法执行等。

　　解决方法：

　　与死锁相比，饥饿现象还是有可能在一段时间之后恢复执行的。**可以设置合适的线程优先级来尽量避免饥饿的产生**。

　　3）**活锁**，活锁体现了一种谦让的美德，每个线程都想把资源让给对方，但是由于机器“智商”不够，可能会产生一直将资源让来让去，导致资源在两个线程间跳动而无法使某一线程真正的到资源并执行，这就是活锁的问题。

**（四）阻塞**

　　阻塞是用来形容多线程的问题，**几个线程之间共享临界区资源，那么当一个线程占用了临界区资源后，所有需要使用该资源的线程都需要进入该临界区等待，等待会导致线程挂起，一直不能工作，这种情况就是阻塞**，**如果某一线程一直都不释放资源，将会导致其他所有等待在这个临界区的线程都不能工作**。当我们使用synchronized或重入锁时，我们得到的就是阻塞线程，如论是synchronized或者重入锁，都会在试图执行代码前，得到临界区的锁，如果得不到锁，线程将会被挂起等待，知道其他线程执行完成并释放锁且拿到锁为止。

　　解决方法：

**可以通过减少锁持有时间，读写锁分离，减小锁的粒度，锁分离，锁粗化等方式来优化锁的性能**。

临界区：

　　临界区是用来表示一种公共的资源（共享数据），它可以被多个线程使用，但是在每次只能有一个线程能够使用它，当临界区资源正在被一个线程使用时，其他的线程就只能等待当前线程执行完之后才能使用该临界区资源。

　　比如办公室办公室里有一支笔，它一次只能被一个人使用，假如它正在被甲使用时，其他想要使用这支笔的人只能等甲使用完这支笔之后才能允许另一个人去使用。这就是临界区的概念。

参考 <https://www.cnblogs.com/Eternally-dream/p/9678314.html>

**9、 synchronized 关键字**

synchronized关键字可以保证被它修饰的方法或者代码块在任意时刻只能有一个线程执行。

synchronized关键字最主要的三种使用方式：**修饰实例方法:、修饰静态方法、修饰代码块。**

* **对于普通同步方法，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。**
* **对于同步代码块，锁是synchronized括号里配置的对象。**

　　当一个线程试图访问同步代码块时，它首先必须得到锁，退出或抛出异常时必须释放锁。

**synchronized在JVM里是怎么实现的？**

**synchronized 同步语句块的实现使用的是 monitorenter 和 monitorexit 指令，其中 monitorenter 指令指向同步代码块的开始位置，monitorexit 指令则指明同步代码块的结束位置**。 当执行 monitorenter 指令时，线程试图获取锁也就是获取 monitor的持有权。当计数器为0则可以成功获取，获取后将锁计数器设为1也就是加1。相应的在执行 monitorexit 指令后，将锁计数器设为0，表明锁被释放。如果获取对象锁失败，那当前线程就要阻塞等待，直到锁被另外一个线程释放为止。

(monitor对象存在于每个Java对象的对象头中，synchronized 锁便是通过这种方式获取锁的，也是为什么Java中任意对象可以作为锁的原因)

**synchronized用的锁是存在哪里的？**

　　synchronized用到的锁是存在**Java对象头**里的。

**10、说说 JDK1.6 之后的synchronized 关键字底层做了哪些优化，可以详细介绍一下这些优化吗**

　　JDK1.6 对锁的实现引入了大量的优化，如**偏向锁、轻量级锁、自旋锁、适应性自旋锁、锁消除、锁粗化**等技术来减少锁操作的开销。

**锁主要存在四种状态，依次是：无锁状态、偏向锁状态、轻量级锁状态、重量级锁状态**，他们会随着竞争的激烈而逐渐升级。注意**锁可以升级不可降级**，这种策略是为了提高获得锁和释放锁的效率。

　　关于这几种优化的详细信息可以查看这篇文章：<https://gitee.com/SnailClimb/JavaGuide/blob/master/docs/java/Multithread/synchronized.md>

**11、synchronized和 Lock 的区别？（重要）**

1）**Lock是一个接口，而synchronized是Java中的关键字**，synchronized是内置的语言实现；

2）**synchronized在发生异常时，会自动释放线程占有的锁，因此不会导致死锁现象发生**；而**Lock在发生异常时，如果没有主动通过unLock()去释放锁，则很可能造成死锁现象，因此使用Lock时需要在finally块中释放锁**；

3）**Lock可以让等待锁的线程响应中断**，而**synchronized却不行，使用synchronized时，等待的线程会一直等待下去，不能够响应中断**；

4）**通过Lock可以知道有没有成功获取锁（tryLock()方法：如果获取锁成功，则返回true）**，而**synchronized却无法办到**。

5）**Lock可以提高多个线程进行读操作的效率**。

　　在性能上来说，如果**竞争资源不激烈，两者的性能是差不多的**，而当**竞争资源非常激烈时（即有大量线程同时竞争），此时Lock的性能要远远优于synchronized**。所以说，在具体使用时要根据适当情况选择。

参考<https://blog.csdn.net/qq_38200548/article/details/82943222>

**12、synchronized和ReentrantLock（重入锁） 的区别？**

* 两者都是**可重进入锁**，就是**能够支持一个线程对资源的重复加锁。sychnronized关键字隐式的支持重进入**，比如一个sychnronized修饰的递归方法，在方法执行时，执行线程在获取了锁之后仍能连续多次地获取该锁。**ReentrantLock**虽然没能像sychnronized关键字一样隐式的重进入，但是**在调用lock()方法时，已经获取到锁的线程，能够再次调用lock()方法获取锁而不被阻塞**。
  + 线程重复n次获取了锁，随后在第n次释放该锁后，其他线程能够获取到该锁。锁的最终释放要求**锁对于获取进行计数自增，计数表示当前锁被重复获取的次数，而锁被释放时，计数自减，当计数等于0时表示锁已经成功被释放**。
* **synchronized 依赖于 JVM 而 ReentrantLock 依赖于 API**。ReentrantLock 是 JDK 层面实现的（也就是 API 层面，需要 lock() 和 unlock() 方法配合 try/finally 语句块来完成）
* **ReentrantLock 比 synchronized 增加了一些高级功能，主要有3点：①等待可中断；②可实现公平锁；③可实现选择性通知（锁可以绑定多个条件）**
  + **ReentrantLock提供了一种能够中断等待锁的线程的机制**，也就是说正在等待的线程可以选择放弃等待，改为处理其他事情。通过lock.lockInterruptibly()来实现这个机制。
  + **ReentrantLock可以指定是公平锁还是非公平锁。而synchronized只能是非公平锁。**（公平锁就是先等待的线程先获得锁）
  + synchronized关键字与wait()和notify()/notifyAll()方法相结合可以实现等待/通知机制。ReentrantLock类当然也可以实现，但是需要借助于Condition接口与newCondition() 方法。用ReentrantLock类结合Condition实例可以实现“选择性通知” 。如果执行notifyAll()方法的话就会通知所有处于等待状态的线程这样会造成很大的效率问题，而Condition实例的signalAll()方法 只会唤醒注册在该Condition实例中的所有等待线程

**13、volatile关键字**

**保证共享变量的“可见性”**。可见性的意思是当一个线程修改一个共享变量时，另外一个线程能读到这个修改的值。

　　把变量声明为volatile，这就指示 JVM每次使用它都到主存中进行读取。

**14、synchronized 关键字和 volatile 关键字的区别**

* volatile关键字是线程同步的轻量级实现，所以**volatile性能比synchronized关键字要好**。但是**volatile关键字只能用于变量而synchronized关键字可以修饰方法以及代码块**。
* **多线程访问volatile关键字不会发生阻塞，而synchronized关键字可能会发生阻塞**。、
* **volatile**关键字主要用于**解决变量在多个线程之间的可见性**，而 **synchronized**关键字解决的是**多个线程之间访问资源的同步性**。
* volatile关键字能保证数据的可见性，但不能保证数据的原子性。synchronized关键字两者都能保证。

**15、使用线程池的好处？**

1. **降低资源消耗**。通过**重复利用已创建的线程，降低线程创建和销毁造成的消耗**。
2. **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. **提高线程的可管理性**。**线程是稀缺资源，如果无限制地创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一分配、调优和监控**。

**16、说一说几种常见的线程池及适用场景？（重要）**

　　可以创建（**Executors.newXXX**）3种类型的ThreadPoolExecutor：**FixedThreadPool**、**SingleThreadExecutor**、**CachedThreadPool**。

* **FixedThreadPool**：**可重用固定线程数的线程池**。（适用于负载比较重的服务器）
  + **FixedThreadPool使用无界队列LinkedBlockingQueue作为线程池的工作队列**
  + 该线程池中的线程数量始终不变。当有一个新的任务提交时，线程池中若有空闲线程，则立即执行。若没有，则新的任务会被暂存在一个任务队列中，待有线程空闲时，便处理在任务队列中的任务。
* **SingleThreadExecutor**：**只会创建一个线程执行任务。**（适用于需要保证顺序执行各个任务；并且在任意时间点，没有多线程活动的场景。）
  + **SingleThreadExecutorl也使用无界队列LinkedBlockingQueue作为工作队列**
  + 若多余一个任务被提交到该线程池，任务会被保存在一个任务队列中，待线程空闲，按先入先出的顺序执行队列中的任务。
* **CachedThreadPool：是一个会根据需要调整线程数量的线程池。**（大小无界，适用于执行很多的短期异步任务的小程序，或负载较轻的服务器）
  + **CachedThreadPool使用没有容量的SynchronousQueue作为线程池的工作队列，但CachedThreadPool的maximumPool是无界的。**
  + 线程池的线程数量不确定，但若有空闲线程可以复用，则会优先使用可复用的线程。若所有线程均在工作，又有新的任务提交，则会创建新的线程处理任务。所有线程在当前任务执行完毕后，将返回线程池进行复用。
* **ScheduledThreadPool**：继承自ThreadPoolExecutor。它主要用来**在给定的延迟之后运行任务，或者定期执行任务**。使用DelayQueue作为任务队列。

**16、线程池都有哪几种工作队列？（重要）**

* **ArrayBlockingQueue**：是一个**基于数组结构的有界阻塞队列**，此队列按FIFO（先进先出）原则对元素进行排序。
* **LinkedBlockingQueue**：是一个**基于链表结构的阻塞队列**，此队列按FIFO排序元素，吞吐量通常要高于ArrayBlockingQueue。静态工厂方法Executors.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，吞吐量通常要高于Linked-BlockingQueue，静态工厂方法Executors.newCachedThreadPool使用了这个队列。
* **PriorityBlockingQueue**：一个**具有优先级的无限阻塞队列**。

**17、创建线程的几种方式？（重要）**

有4种方式：**继承Thread类**、**实现Runnable接口**、**实现Callable接口**、**使用Executor框架来创建线程池**。

**（1）通过继承Thread类创建线程**

public class MyThread **extends Thread**{//继承Thread类

**//重写run方法**

　　public void run(){

　　}

}

----------------------------------------------------------------------------------

public class Main {

　　public static void main(String[] args){

　　　　new MyThread()**.start()**; //创建并启动线程

　　}

}

**（2）通过实现Runnable接口来创建线程**

public class MyThread2 **implements Runnable** {//实现Runnable接口

**//重写run方法**

　　public void run(){

　　}

}

------------------------------------------------------------------------------------------

public class Main {

　　public static void main(String[] args){

　　　　//创建并启动线程

　　　　MyThread2 myThread=new MyThread2();

　　　　Thread thread=new Thread(myThread);

　　　　thread()**.start()**;

　　　　//或者    new Thread(new MyThread2()).start();

　　}

}

不管是继承Thread还是实现Runnable接口，多线程代码都是通过运行Thread的start()方法来运行的。

**（3）实现Callable接口来创建线程**

与实现Runnable接口类似，和Runnable接口不同的是，Callable接口提供了一个call() 方法作为线程执行体，call()方法比run()方法功能要强大：call()方法可以有返回值、call()方法可以声明抛出异常。

public class Main {

　　public static void main(String[] args){

　　　MyThread3 th=new MyThread3();

　　　//使用Lambda表达式创建Callable对象

　　   //使用FutureTask类来包装Callable对象

**Future**Task<Integer> future=new FutureTask<Integer>(

　　　　(Callable<Integer>)()->{

　　　　　　return 5;

　　　　}

　　  );

　　　new Thread(task,"有返回值的线程")**.start()**;//实质上还是以Callable对象来创建并启动线程

　　  try{

　　　　System.out.println("子线程的返回值："+future.get());//get()方法会阻塞，直到子线程执行结束才返回

 　　 }catch(Exception e){

　　　　ex.printStackTrace();

　　　}

　　}

}

**（4）使用Executor框架来创建线程池**

**Executors.newXXXX： newFixedThreadPool(int )、newSingleThreadExecutor、newCachedThreadPool、newScheduledThreadPool(int)**

通过Executors的以上四个静态工厂方法获得**ExecutorService**实例**，**而后可以执行Runnable任务或Callable任务。

* **Executor执行Runnable任务**：

　　通过Executors的以上四个静态工厂方法获得 ExecutorService实例，而后调用该实例的execute（Runnable command）方法即可。一旦Runnable任务传递到execute（）方法，该方法便会自动在一个线程上。

[复制代码](javascript:void(0);)

public class TestCachedThreadPool{

public static void main(String[] args){

ExecutorService executorService = Executors.newCachedThreadPool();

for (int i = 0; i < 5; i++){

executorService.execute(new TestRunnable());

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\* a" + i + " \*\*\*\*\*\*\*\*\*\*\*\*\*");

}

executorService.shutdown();

}

}

class TestRunnable implements Runnable{   
 //重写run方法

public void run(){

System.out.println(Thread.currentThread().getName() + "线程被调用了。");

}

[复制代码](javascript:void(0);)

* **Executor执行Callable任务：**

　　当将一个Callable的对象传递给ExecutorService的submit方法，则该call方法自动在一个线程上执行，并且**会返回执行结果Future对象**。同样，将Runnable的对象传递给ExecutorService的submit方法，则该run方法自动在一个线程上执行，并且会返回执行结果Future对象，但是在该Future对象上调用get方法，将返回null。

[复制代码](javascript:void(0);)

public class CallableDemo{

public static void main(String[] args){

ExecutorService executorService = Executors.newCachedThreadPool();

List<Future<String>> resultList = new ArrayList<Future<String>>();

//创建10个任务并执行

for (int i = 0; i < 10; i++){

//使用ExecutorService执行Callable类型的任务，并将结果保存在future变量中

Future<String> future = executorService.submit(new TaskWithResult(i));

//将任务执行结果存储到List中

resultList.add(future);

}

//遍历任务的结果

for (Future<String> fs : resultList){

try{

while(!fs.isDone);//Future返回如果没有完成，则一直循环等待，直到Future返回完成

System.out.println(fs.get()); //打印各个线程（任务）执行的结果

}catch(InterruptedException e){

e.printStackTrace();

}catch(ExecutionException e){

e.printStackTrace();

}finally{

//启动一次顺序关闭，执行以前提交的任务，但不接受新任务

executorService.shutdown();

}

}

}

}

class TaskWithResult implements Callable<String>{

private int id;

public TaskWithResult(int id){

this.id = id;

}

// 重写call()方法

public String call() throws Exception {

System.out.println("call()方法被自动调用！！！ " + Thread.currentThread().getName());

//该返回结果将被Future的get方法得到

return "call()方法被自动调用，任务返回的结果是：" + id + " " + Thread.currentThread().getName();

}

}

[复制代码](javascript:void(0);)

实现Runnable接口和Callable接口的区别？

 Runnable接口或Callable接口实现类都可以被ThreadPoolExecutor或ScheduledThreadPoolExecutor执行。两者的区别在于 **Runnable 接口不会返回结果但是 Callable 接口可以返回结果**。

**执行execute()方法和submit()方法的区别是什么呢？**

1)**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与否；

2) **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，而使用 get（long timeout，TimeUnit unit）方法则会阻塞当前线程一段时间后立即返回，这时候有可能任务没有执行完。

**18.线程池参数？**

①**corePoolSize：线程池的基本大小**，当提交一个任务到线程池时，线程池会创建一个线程来执行任务，即使其他空闲的基本线程能够执行新任务也会创建线程，等到需要执行的任务数大于线程池基本大小时就不再创建。说白了就是，即便是线程池里没有任何任务，也会有corePoolSize个线程在候着等任务。

②**maximumPoolSize:最大线程数**，不管你提交多少任务，线程池里最多工作线程数就是maximumPoolSize。

③**keepAliveTime:线程的存活时间**。当线程池里的线程数大于corePoolSize时，如果等了keepAliveTime时长还没有任务可执行，则线程退出。

⑤**unit**：这个用来指定keepAliveTime的单位，比如秒:TimeUnit.SECONDS。

⑥**workQueu**e：用于保存等待执行任务的**阻塞队列**，提交的任务将会被放到这个队列里。

⑦**threadFactory：线程工厂，用来创建线程**。主要是为了给线程起名字，默认工厂的线程名字：pool-1-thread-3。

⑧**handler：拒绝策略**，即当线程和队列都已经满了的时候，应该采取什么样的策略来处理新提交的任务。默认策略是AbortPolicy（抛出异常），其他的策略还有：CallerRunsPolicy(只用调用者所在线程来运行任务)、DiscardOldestPolicy(丢弃队列里最近的一个任务，并执行当前任务)、DiscardPolicy(不处理，丢弃掉)

**19.线程池执行流程?**

　　任务被提交到线程池，会先判断当前线程数量是否小于corePoolSize，如果小于则创建线程来执行提交的任务，否则将任务放入workQueue队列，如果workQueue满了，则判断当前线程数量是否小于maximumPoolSize,如果小于则创建线程执行任务，否则就会调用handler，以表示线程池拒绝接收任务。

**1、什么是线程,什么是进程,它们有什么区别和联系,一个进程里面是否必须有个线程**

        进程本质是一个正在执行的程序,一个进程可以有多个线程.线程是进程的最小执行单位,一个进程至少有一个线程

区别:1:多进程程序不受java控制,而多线程则受java控制,

        2:多线程比多进程需要更少的管理成本

**2、实现一个线程有哪几种方式,各有什么优缺点,比较常用的是那种,为什么**

        实现一个线程有三种方式:

1)      继承Thread类,需要重写run方法,无返回值

a)        优点:可以直接调用start方法启动线程

b)        缺点:java只能单继承,如果已经有了父类,不能用这种方法

2)      实现Runnable接口,需要重写run()方法

a)        优点:即使自己定义的线程类有了父类也可以实现接口,而且接口是多实现

b)        缺点:需通过构造一个Thread把自己传进去,才能实现Thread的方法,代码复杂

3)      实现Callable接口,需要重写call()方法

a)        优点:可以抛出异常,有返回值

b)        缺点:只有jdk1.5以后才支持,结合FuntureTask和Thread类一起使用,最后调用start启动线程

一般用第二种,实现Runnable接口,比较方便,扩展性高

**3、一般情况下我们实现自己线程时候要重写什么方法**

        使用Thread类,要重写run()方法

        实现Runnable接口时,要实现run()方法

        使用Callable接口时,要重写call()方法

**4、start方法和run方法有什么区别,我们一般调用的那个方法,系统调用的是那个方法**

        Start用来启动线程

        当调用start后,线程并不会马上运行,而是处于就绪状态,是否要运行取决于cpu

        Run用来子类重写来实现线程的功能

        我们一般调用的是start方法,系统调用的是run方法

**5、sleep方法有什么作用,一般用来做什么**

        Sleep是一个Thread类的静态方法

        作用:让调用它的线程休眠指定的时间,用于暂停线程,但不会把线程锁让给其它线程,休眠时间结束,线程进入就绪状态,等待cpu分配的执行机会

**6、讲下join,yield方法的作用,以及什么场合用它们**

        Join()有严格的先后顺序,调用它的线程需要执行完,其它线程才会执行

        Yield()是暂停当前正在执行的线程对象,把时间让给其它线程

**7、线程中断是否能直接调用stop,为什么?**

        线程终端不能直接调用stop()方法

        Stop()方法是从外部强行终止一个线程,会导致不可预知的错误,比如线程锁没有归还,io流不能关流        线程只能调用interrupt()方法中断,而且不是立即中断,     只是发出了一个类似于信号量的东西,通过修改了被调用线程的中断状态来告知那个线程,说它被中断了,至于什么时候中断,这个系统会判断)系统会在一个合适的时候进行中断处理

**8、列举出一般情况下线程中断的几种方式,并说明他们之间的优缺点,并且说明那种中断方式最好**

        线程中断有4种方式

由interrupt发出中断信号,用户接收中断信号,通过isInterrupted判断线程是否中断

由interrupt发出中断信号,系统接收中断信号,通过sleep抛出中断异常,并把中断信号清除,只能抛出一次

用户自定义中断,将中断信号发出,自己接收该中断信号

调用interrupted(),会把中断信号清除,并中断线程

**9、线程有几种状态,他们是怎么转化的**

        线程一般分为:新生,就绪,运行,阻塞,死亡五中状态

        当创建一个线程后,并没有运行,处于新生状态,需要通过调用start方法,让线程处于就绪状态,但是否运行取决于cpu分配的执行机会,当得到cpu的执行机会后马上运行,一个正在执行的线程可以通过很多方式进入阻塞状态,当执行完所有操作后就进入死亡状态

**10、在实现Runnable的接口中怎么样访问当前线程对象,比如拿到当前线程的名字**

        通过Thread.currentThread().getName()可获得当前线程名字

**11、讲下什么是守护线程,以及在什么场合来使用它**

守护线程是用来监听其它线程是否挂掉,这个线程具有最低优先级

用于为系统中的其它线程提供服务

守护线程就像象棋里的车马相仕,非守护线程就是老帅,老帅挂掉,守护线程也就挂掉了

        当主线程和主线程创建的子线程全部退出,守护线程一定会跟着退出

        (比如QQ主程序和QQ聊天窗口,主程序退出,QQ聊天窗口也随之关掉)

**12、一般的线程优先级是什么回事,线程优先级高的线程一定会先执行吗?如果不设置优先级的话,那么线程优先级是多少,设置线程优先级用那个函数**

        线程的优先级就是设置哪个线程先执行,但不是绝对,只是让优先级高的线程优先运行的概率高一些,

        线程默认优先级是NORM\_PRIORITY = 5;

        最小是 MIN\_PRIORITY = 1;

        最大是 MAX\_PRIORITY = 10;

        设置线程优先级有setPriority()方法

**13、为什么Thread里面的大部分方法都是final的**

        线程很多方法都是由系统调用的,不能通过子类重写去改变他们的行为

**14、什么是线程同步,什么是线程安全**

        同步:当两个或两个以上的线程需要共享资源,通过同步方法限制资源在一次仅被一个线程占用

        线程安全:线程安全就是多线程操作同一个对象不会产生数据污染,

        线程同步一般来保护线程安全,final修饰的也是线程安全

**15、讲下同步方法和同步块的区别,以及什么时候用它们**

        同步方法:被synchronized修饰的方法,同步整个方法,且整个方法都会被锁住,同一时间只有一个线程可以访问该方法,

        同步块: 被synchronized修饰的代码块,可以同步一小部分代码

什么时候用?

        同步块越小性能越好,当性能要求比较高,用同步代码块

**16、简单说下Lock对象的实现类的锁机制和同步方法或同步块有什么区别**

        是JDK1.5才出现的,Lock对象比synchronized更加灵活,可以控制什么时候上锁,什么时候解锁,而使用synchronized必须等代码执行完才会解锁.

       Synchronized在锁定时如果方法块抛出异常,JVM会自动释放锁,而Lock出现异常必须在finally将锁释放,否则将引起死锁

**17、同步块里面的同步监视器是怎么写的,默认的同步方法里面的同步监视器是那个**

        Synchronized(对象){

                        //代码块

        }

       默认的同步监视器是this

**18、讲下什么是死锁,死锁发生的几个条件是什么**

        死锁就是当有两个或两个以上的线程都获得对方的资源,但彼此都不肯放开,处于僵持阶段,此时就造成了死锁

        条件:两个或两个以上的线程,同时想要获得对方的资源,彼此又不肯放开

**19、线程间是怎么通信的,通过调用几个方法来交互的**

        线程是通过wait , notify等方法相互作用进行协作通信；

        wait()方法使得当前线程必须要等待，直到到另外一个线程调用notify()或者notifyAll()方法唤醒

        -- wait()和notify()方法要求在调用时线程已经获得了对象锁，因此对这两个方法的调用需要放           在  synchronized修饰的方法或代码块中。

**20、wait,notify,notifyAll在什么地方使用才有效,他们是那个类的方法**

        Wait,notify,notifyAll都必须在synchronized修饰的方法或代码块中使用,都属于Object的方法,可以被所有类继承,都是final修饰的方法,不能通过子类重写去改变他们的行为

**21、wait和sleep有什么区别和联系,他们执行的时候是否都会释放锁**

        Wait和sleep都可以使线程暂停,但wait必须在synchronized修饰的方法或代码块中使用,

       Sleep()方法是线程类的静态方法,调用此方法会让当前线程暂停执行指定的时间,将执行机会(CPU)让给其它线程,但是对象的锁依然保持,因此休眠时间结束后会自动恢复(就绪状态),

       wait()是Object类的方法,调用对象的wait()方法会让当前线程放弃对象的锁(线程暂停执行),进入对象的等待池(wait pool),只有调用对象的Notify()或notifyAll()方法才能唤醒等待池中的线程进入等锁池(lock pool),如果线程重新获得对象的锁就可以进入就绪状态

**22、yield,sleep方法有什么区别和联系**

        Yield和sleep都可以让线程暂停,

1)      Sleep()方法给其它线程运行机会时不考虑线程的优先级,因此会给低优先级的线程以运行机会…yield()方法只会给相同优先级或更高优先级的线程以运行机会

2)      线程执行sleep()方法后转入阻塞状态,而执行yield()方法后转入就绪状态

3)      Sleep()方法声明抛出InterruptedException,而yield()方法没有声明任何异常

4)      Sleep()方法比yield()方法具有更好的可移植性

**23、线程的启动是哪个方法，调用的是哪个方法？**

        Start用于启动线程,当调用start后,线程并不会马上运行,而是处于就绪状态,是否运行取决于cpu给的时间片

        Run()方法用于子类重写来实现线程的功能

        我们一般调用的是start方法,系统调用的是run方法

**24、线程安全与线程不安全的区别**

              线程安全就是多线程访问时,采用了加锁机制,

              当一个线程访问类中某个数据,进行保护,其它线程不能进行访问,直到该线程执行完,其它线程才可使用---不会导致数据污染

              线程不安全就是不提供数据访问保护,有可能出现多个线程先后更改数据造成所得到的数据是脏数据

**25、线程的实现方式，线程的生命周期等**

              实现方式

1)  通过继承Thread类

2)  实现Runnable接口

3)  实现Callable接口

              生命周期

              新建

              就绪

              运行

              堵塞

              死亡

**26、如何处理线程不安全问题(有2种解决方法)**

              第一，是采用原子变量,线程安全问题最根本上是由于全局变量和静态变量引起的，定义变量用sig\_atomic\_t和volatile。

              第二，就是实现线程间同步,让线程有序访问变量

**27、线程中常用方法的区别**

              线程中用到最多的是start方法,它的作用是启动一个线程

              Sleep()让线程休眠一会,但不会释放锁

              Join()强制执行一个线程

              Wait()是Object类中的方法,会让当前线程放弃对象锁,进入对象等待池

              Yield()礼让线程,让另外一个线程执行一会,自己再执行,只能让同优先级的线程有执行的机会

              currentThread()用于获得当前线程对象,可以调用getName()方法获得当前线程名字

**28、多线程和单线程有什么区别？**

        干一件事情一个人干和两个人同时干的区别

**29、线程的5种状态，\_\_、\_、\_、\_\_\_。启动调用\_方法，启动后会调用\_\_方法。**

              新生

              就绪

              运行

              阻塞

              死亡

              启动调用start()

**30、用同步块与同步方法的区别？**

              同步方法是被synchronized修饰的方法,同步整个方法, ,同一时间只有一个线程可以访问该                   方法

              同步块是被synchronized修饰的代码块,可以同步一小部分代码

              同步代码越少性能越好,当性能要求比较高时,用同步块

**31、什么是同步和异步，分别用例子说明,同步有几种方式？**

        同步是排队去做事情,异步就是各做各的

**32、什么是对象锁？**

1.  java中的每个对象都有一个锁，当访问某个对象的synchronized方法时，表示将该对象上锁，此时其他任何线程都无法在去访问该syncronized 方法了，直到之前的那个线程执行方法完毕后，其他线程才有可能去访问该synchronized方法。

2.如果一个对象有多个synchronized方法，某一时刻某个线程已经进入到某个synchronzed方法，那么在该方法没有执行完毕前，其他线程无法访问该对象的任何synchronzied 方法的，但可以访问非synchronzied方法。

3.如果synchronized方法是static的，那么当线程访问该方法时，它锁的并不是synchronized方法所在的对象，而是synchuronized方法所在对象的对应的Class对象，

        [类锁（synchronized修饰的静态方法）]

**33、什么是死锁？**

       死锁就是当有两个或两个以上的线程都获得对方的资源，但彼此有不肯放开，处于僵持状态，此时便造成了死锁。

**java实现线程有哪几种方式**

1.继承Thread类实现多线程  
2.实现Runnable接口方式实现多线程  
3.使用线程池：如ExecutorService，Callable，Future  
第一种和第二种方式相比：java类只能允许继承一个父类，可以实现多个接口；其次，在第一种方式中用this可以获取当前线程，第二种方式获取当前线程只能通过Thread.currentThread()  
第三种：线程的创建和释放，需要占用不小的内存和资源。如果每次需要使用线程时，都new 一个Thread的话，难免会造成资源的浪费，而且可以无限制创建，之间相互竞争，会导致过多占用系统资源导致系统瘫痪， ExecutorService是Java提供的线程池

public ThreadPoolExecutor(int corePoolSize,

int maximumPoolSize,

long keepAliveTime,

TimeUnit unit,

BlockingQueue<Runnable> workQueue,

ThreadFactory threadFactory,

corePoolSize:核心线程数  
maximumPoolSize：最大线程数  
poolSize：线程池中当前线程的数量  
那么poolSize、corePoolSize、maximumPoolSize三者的关系是如何的呢？  
当新提交一个任务时：  
（1）如果poolSize<corePoolSize，新增加一个线程处理新的任务。  
（2）如果poolSize=corePoolSize，新任务会被放入阻塞队列等待。  
（3）如果阻塞队列的容量达到上限，且这时poolSize<maximumPoolSize，新增线程来处理任务。  
（4）如果阻塞队列满了，且poolSize=maximumPoolSize，那么线程池已经达到极限，会根据饱和策略RejectedExecutionHandler拒绝新的任务  
ExecutorService分为以下几种线程池：可缓存线程池newCachedThreadPool()，其中创建的都是非核心线程，；单线程newSingleThreadExecutor()；  
固定线程池newFixedThreadPool(3);定时线程池

**线程同步有哪几种方法**

Synchronized,Lock锁，分布式锁  
Synchronized,Lock锁的区别：  
1.Synchronized是java内置关键字，Lock是java 类 Lock lock = new ReentrantLock()  
2.synchronized无法判断是否获取锁的状态，Lock可以判断是否获取锁lock.tryLock()  
3.synchronized的锁可重入、不可中断、非公平，而Lock锁可重入、可判断、可公平  
4.Lock锁适合大量同步的代码的同步问题，synchronized锁适合代码少量的同步问题  
5.synchronized会自动释放锁(a 线程执行完同步代码会释放锁 ；b 线程执行过程中发生异常会释放锁)，Lock需在finally中手工释放锁（unlock()方法释放锁），否则容易造成线程死锁；  
6.用synchronized关键字的两个线程1和线程2，如果当前线程1获得锁，线程2线程等待。如果线程1阻塞，线程2则会一直等待下去，而Lock锁就不一定会等待下去，如果尝试获取不到锁，线程可以不用一直等待就结束了//lock.tryLock(3000, TimeUnit.MILLISECONDS) //尝试获取锁 获取不到锁，就等3秒，如果3秒后还是获取不到就返回false

**synchronized与volatile区别**

同步块（或方法）和 volatile 变量。这两种机制的提出都是为了实现代码线程的安全性。其中 Volatile 变量的同步性较差（但有时它更简单并且开销更低），而且其使用也更容易出错。volatile关键字用于声明简单类型变量，如int、float、boolean等数据类型。如果这些简单数据类型声明为volatile，对它们的操作就会变成原子级别的。但这有一定的限制。并不是只要简单类型变量使用volatile修饰，对这个变量的所有操作都是原来操作，当变量的值由自身的上一个决定时，如n=n+1、n++等，volatile关键字将失效，只有当变量的值和自身上一个值无关时对该变量的操作才是原子级别的，如n = m + 1，这个就是原级别的。所以在使用volatile关键时一定要谨慎，如果自己没有把握，可以使用synchronized来代替volatile。

**java中notify和notifyAll的区别**

notify方法不能唤醒某个具体的线程，而notifyAll唤醒所有的线程并允许他们争夺锁，确保至少有一个线程能继续运行

**为什么wait/notify/notifyAll这些方法不在thread类里面**

一个明显的原因就是java提供的锁是对象级而不是线程级。每个对象都有锁，通过线程获得。如果线程需要等待某些锁那么调用对象中wait方法才有意义

**什么是死锁**

死锁是两个线程相互等待对方释放对象锁

**启动线程方法start()和run()有什么区别**

只有调用了start方法，才会表现多线程的特性，不同线程的run方法里面的带脉交替执行。如果只是调用了run方法，那么代码还是同步执行，必须等待一个线程的run方法里面的代码全部执行完，才会执行其自己的run方法里面的代码

**多线程通信**

wait/notify

**java中用到了什么线程调度算法**

抢占式。一个线程的cpu用完后，操作系统会根据线程优先级，线程饥饿情况等情况分配某个线程

**synchronized有哪几种用法**

锁类，锁方法，锁代码块

**活锁，饥饿，无锁，死锁**

**死锁**：多个线程互相占用对方资源的锁，又相互等待对方释放锁，若无外力干涉，这些线程则一直处于阻塞你的假死状态，形成死锁  
**活锁**：指拿到资源却又相互释放不执行，这个资源在多个线程之间跳动又得不到执行  
**饥饿**：指优先级高的线程一直占着优先级低的资源，导致优先级低的线程无法得到执行，处于长久等待中

[**如何判断线程是否死亡:**](https://www.jianshu.com/p/5e0bf2046df5)

可以调用线程对象的isAlive方法，当处于就绪、运行、阻塞时，返回true。当处于新建或死亡时，返回false

[**能不能对一个死亡的线程重新调用start方法让它重新启动：**](https://www.jianshu.com/p/5e0bf2046df5)

不能，死亡就是死亡，该线程将不可再次作为线程执行

[**就绪状态的线程如何获取处理器资源：**](https://www.jianshu.com/p/5e0bf2046df5)

线程从就绪状态到运行状态不受程序控制，而是根据系统线程调用所决定，当处于就绪状态的线程获得到处理器资源时，会进入运行状态。而当处于运行状态的线程失去处理器资源时，会处于就绪状态。但是yield方法可以使线程从运行状态转入就绪状态

[**Java程序每次运行至少启动几个线程**](https://www.jianshu.com/p/5e0bf2046df5)

两个线程，一个是主线程，一个是垃圾回收机制的线程

[**如果没有启动start而是直接启动线程的run方法会怎么样：**](https://www.jianshu.com/p/5e0bf2046df5)

虚拟机会把这个线程当做一个普通的对象，而run方法也会被看做是一个普通的对象中的方法

[**Join()在多线程中的作用:**](https://www.jianshu.com/p/5e0bf2046df5)

让一个线程等待另一个线程完成，比如：我创建了一个MyThread线程：

MyThread myThread = new MyThread();

然后在main()方法中调用（本质是在main线程中调用）myThread.join()方法，那么main线程会进入阻塞状态直到MyThread线程中的run方法的执行完毕后才会继续执行。

[**什么是后台线程：**](https://www.jianshu.com/p/5e0bf2046df5)

有一种线程，是在后台运行的，它的任务是为其他线程提供服务，又称为“守护线程”，

比如JVM的垃圾回收线程就是一个后台线程。后台线程有一个特征，就是如果所有前台线程全部死亡，那么后台线程会自动死亡。

[**线程睡眠Sleep（long millis）：**](https://www.jianshu.com/p/5e0bf2046df5)

可以让线程暂停一段时间,当线程进入睡眠状态后，该线程不会获得执行机会，即使系统中没有其他可以执行的线程，处于sleep中的线程也不会执行，直到睡眠结束

[**sleep()和wait()的区别:**](https://www.jianshu.com/writer)

最大的区别是，sleep()在睡眠后不会释放掉锁，而wait()在睡眠后会释放掉锁。

[**线程让步yield():**](https://www.jianshu.com/p/5e0bf2046df5)

将当前正在执行的线程暂停，但是是将线程进入到就绪状态，释放CPU资源

[**线程优先级是什么，如何改变线程优先级：**](https://www.jianshu.com/p/5e0bf2046df5)

每个线程都有一定的优先级，优先级高的线程会获得到更多的执行机会。每个线程的优先级和创建它的父线程的优先级相同，Thread提供了setPriority方法来设定优先级，参数范围为1-10

[**什么是线程安全，什么是线程不安全（synchronized的作用或者由来）**](https://www.jianshu.com/p/5e0bf2046df5)

线程的run方法不具有同步安全性，当多个线程同时处理一个共享数据时，可能导致数据混乱，这就是线程不安全。所以引入了synchronized来解决此问题。被synchronized锁定的同步代码块，只能同时被一个线程获取到，当该线程执行完此方法后，会释放掉该锁，这时其他线程才可以继续获取该锁。Synchronized()括号内可以传任何参数，即Obj类型，但通常是被共同访问的共享资源来作为同步监视器

[**同步方法是什么，如何使用：**](https://www.jianshu.com/p/5e0bf2046df5)

被synchronized修饰的方法，成为同步方法，如：

Public synchronized void getPerson(){};

[**当内部类中的代码块被synchronized（this）修饰时，请问这个this指的是内部类还是其父类：**](https://www.jianshu.com/p/5e0bf2046df5)

内部类

[**当一个线程访问某个类中的被synchronized修饰的方法时，另一个线程可以访问该类中其他没有被synchronized修饰的方法吗：**](https://www.jianshu.com/p/5e0bf2046df5)

可以

[**当一个线程访问某个类中的被synchronized修饰的方法时，另一个线程可以访问该类中其他被synchronized修饰的方法吗：**](https://www.jianshu.com/p/5e0bf2046df5)

不可以，因为Java中的每个对象都有一个锁（lock），或者叫做监视器（monitor），当一个线程访问某个对象的synchronized方法时，**将该对象上锁**，其他任何线程都无法再去访问该对象的synchronized方法了（这里是指所有的同步方法，而不仅仅是同一个方法），直到之前的那个线程执行方法完毕后（或者是抛出了异常），才将该对象的锁释放掉，其他线程才有可能再去访问该对象的synchronized方法。

[**Synchronized如何被释放:**](https://www.jianshu.com/p/5e0bf2046df5)

1、当前线程的同步方法，同步代码块执行结束

2、出现了未处理的Error或Exception，导致代码块结束

3、当线程正在执行加锁的代码块或同步方法时，调用了wait()方法，那么会使线程进入阻塞状态，并且释放锁

4、注意：当线程正在执行加锁的代码块或同步方法时，调用sleep()方法或者yield()方法，不会释放锁

[**什么是死锁，什么情况下会出现死锁:**](https://www.jianshu.com/p/5e0bf2046df5)

当所有线程处于阻塞状态，整个程序没有发生异常，也不会有任何提示，就进入了死锁状态；

当两个线程相互等待对方释放锁时，就会发生死锁。

[**什么是Lock锁，和synchronized有什么区别：**](https://www.jianshu.com/p/5e0bf2046df5)

1、Lock是一个接口，而synchronized是Java中的关键字，synchronized是内置的语言实现；

2、synchronized在发生异常时，会自动释放线程占有的锁，因此不会导致死锁现象发生；而Lock在发生异常时，如果没有主动通过unLock()去释放锁，则很可能造成死锁现象，因此使用Lock时需要在finally块中释放锁；

3、Lock可以通过tryLock()方法知道是否成功获取到锁，但是synchronized不行

4、Lock可以让等待锁的线程响应中断，而synchronized却不行，使用synchronized时，等待的线程会一直等待下去，不能够响应中断；

5、Lock可以提高多个线程进行读操作的效率（读写锁）。

6、Lock可以实现公平锁，Synchronized不保证公平性。

[**线程间通信：**](https://www.jianshu.com/p/5e0bf2046df5)

wait()/notify()机制：

ThreadA处于阻塞状态（wait方法），ThreadB在run方法中对某个数据做操作，当数据达到我设定的某个条件时，通过notify()来唤醒ThreadA。

好处：提高了CPU的利用率

缺点：如果通知过早，会打乱执行的逻

其他通信方法请参考：

<http://www.importnew.com/26850.html>

[**线程池:**](https://www.jianshu.com/p/5e0bf2046df5)

为了避免重复创建线程，线程池可以的出现可以让线程进行复用，当需要时，从线程池中取出一个线程，当工作完成后，并不是直接关闭线程，而是将线程归还给线程池供其他任务使用。

[**常用的线程池：**](https://www.jianshu.com/p/5e0bf2046df5)

**newFixedThreadPool：**

固定大小的线程池，可以指定线程池的大小，该线程池corePoolSize和maximumPoolSize相等，阻塞队列使用的是LinkedBlockingQueue，大小为整数最大值。

**newSingleThreadExecutor：**

单个线程线程池，只有一个线程的线程池，阻塞队列使用的是LinkedBlockingQueue,若有多余的任务提交到线程池中，则会被暂存到阻塞队列，待空闲时再去执行。按照先入先出的顺序执行任务。

**newCachedThreadPool：**

缓存线程池，缓存的线程默认存活60秒。线程的核心池corePoolSize大小为0，核心池最大为Integer.***MAX\_VALUE***,阻塞队列使用的是SynchronousQueue。是一个直接提交的阻塞队列，他总会迫使线程池增加新的线程去执行新的任务。在没有任务执行时，当线程的空闲时间超过keepAliveTime（60秒），则工作线程将会终止被回收，当提交新任务时，如果没有空闲线程，则创建新线程执行任务，会导致一定的系统开销。如果同时又大量任务被提交，而且任务执行的时间不是特别快，那么线程池便会新增出等量的线程池处理任务，这很可能会很快耗尽系统的资源。

**newScheduledThreadPool：**

定时线程池，该线程池可用于周期性地去执行任务，通常用于周期性的同步数据。

1、什么是进程，什么是线程，为什么需要多线程编程？

进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动，是操作系统进行资源分配和调度的一个独立单位；

线程是进程的一个实体，是CPU调度和分派的基本单位，是比进程更小的能独立运行的基本单位。线程的划分尺度小于进程，这使得多线程程序的并发性高；进程在执行时通常拥有独立的内存单元，而线程之间可以共享内存。

使用多线程的编程通常能够带来更好的性能和用户体验，但是多线程的程序对于其他程序是不友好的，因为它可能占用了更多的CPU资源。当然，也不是线程越多，程序的性能就越好，因为线程之间的调度和切换也会浪费CPU时间。时下很时髦的Node.js就采用了单线程异步I/O的工作模式。

**2、什么是线程安全**

如果你的代码在多线程下执行和在单线程下执行永远都能获得一样的结果，那么你的代码就是线程安全的。

这个问题有值得一提的地方，就是线程安全也是有几个级别的：

* 不可变。像String、Integer、Long这些，都是final类型的类，任何一个线程都改变不了它们的值，要改变除非新创建一个，因此这些不可变对象不需要任何同步手段就可以直接在多线程环境下使用
* 绝对线程安全。不管运行时环境如何，调用者都不需要额外的同步措施。要做到这一点通常需要付出许多额外的代价，Java中标注自己是线程安全的类，实际上绝大多数都不是线程安全的，不过绝对线程安全的类，Java中也有，比方说CopyOnWriteArrayList、CopyOnWriteArraySet
* 相对线程安全。相对线程安全也就是我们通常意义上所说的线程安全，像Vector这种，add、remove方法都是原子操作，不会被打断，但也仅限于此，如果有个线程在遍历某个Vector、有个线程同时在add这个Vector，99%的情况下都会出现ConcurrentModificationException，也就是fail-fast机制。
* 线程非安全。这个就没什么好说的了，ArrayList、LinkedList、HashMap等都是线程非安全的类
* **5、简述synchronized 和java.util.concurrent.locks.Lock的异同？**
* Lock是Java 5以后引入的新的API，和关键字synchronized相比主要相同点：Lock 能完成synchronized所实现的所有功能；主要不同点：Lock有比synchronized更精确的线程语义和更好的性能，而且不强制性的要求一定要获得锁。synchronized会自动释放锁，而Lock一定要求程序员手工释放，并且最好在finally 块中释放（这是释放外部资源的最好的地方）。

**7、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.

**8、举例说明同步和异步。**

如果系统中存在临界资源（资源数量少于竞争资源的线程数量的资源），例如正在写的数据以后可能被另一个线程读到，或者正在读的数据可能已经被另一个线程写过了，那么这些数据就必须进行同步存取（数据库操作中的排他锁就是最好的例子）。当应用程序在对象上调用了一个需要花费很长时间来执行的方法，并且不希望让程序等待方法的返回时，就应该使用异步编程，在很多情况下采用异步途径往往更有效率。事实上，所谓的同步就是指阻塞式操作，而异步就是非阻塞式操作。

**9、启动一个线程是调用run()还是start()方法？**

启动一个线程是调用start()方法，使线程所代表的虚拟处理机处于可运行状态，这意味着它可以由JVM 调度并执行，这并不意味着线程就会立即运行。run()方法是线程启动后要进行回调（callback）的方法。

**10、为什么需要run()和start()方法，我们可以只用run()方法来完成任务吗？**

我们需要run()&start()这两个方法是因为JVM创建一个单独的线程不同于普通方法的调用，所以这项工作由线程的start方法来完成，start由本地方法实现，需要显示地被调用，使用这俩个方法的另外一个好处是任何一个对象都可以作为线程运行，只要实现了Runnable接口，这就避免因继承了Thread类而造成的Java的多继承问题。

**11、什么是线程池（thread pool）？**

在面向对象编程中，创建和销毁对象是很费时间的，因为创建一个对象要获取内存资源或者其它更多资源。

在Java中更是如此，虚拟机将试图跟踪每一个对象，以便能够在对象销毁后进行垃圾回收。所以提高服务程序效率的一个手段就是尽可能减少创建和销毁对象的次数，特别是一些很耗资源的对象创建和销毁，这就是“池化资源”技术产生的原因。线程池顾名思义就是事先创建若干个可执行的线程放入一个池（容器）中，需要的时候从池中获取线程不用自行创建，使用完毕不需要销毁线程而是放回池中，从而减少创建和销毁线程对象的开销。

Java 5+中的Executor接口定义一个执行线程的工具。它的子类型即线程池接口是ExecutorService。要配置一个线程池是比较复杂的，尤其是对于线程池的原理不是很清楚的情况下，因此在工具类Executors面提供了一些静态工厂方法，生成一些常用的线程池，如下所示：

* newSingleThreadExecutor：创建一个单线程的线程池。这个线程池只有一个线程在工作，也就是相当于单线程串行执行所有任务。如果这个唯一的线程因为异常结束，那么会有一个新的线程来替代它。此线程池保证所有任务的执行顺序按照任务的提交顺序执行。
* newFixedThreadPool：创建固定大小的线程池。每次提交一个任务就创建一个线程，直到线程达到线程池的最大大小。线程池的大小一旦达到最大值就会保持不变，如果某个线程因为执行异常而结束，那么线程池会补充一个新线程。
* newCachedThreadPool：创建一个可缓存的线程池。如果线程池的大小超过了处理任务所需要的线程，那么就会回收部分空闲（60秒不执行任务）的线程，当任务数增加时，此线程池又可以智能的添加新线程来处理任务。此线程池不会对线程池大小做限制，线程池大小完全依赖于操作系统（或者说JVM）能够创建的最大线程大小。
* newScheduledThreadPool：创建一个大小无限的线程池。此线程池支持定时以及周期性执行任务的需求。
* newSingleThreadExecutor：创建一个单线程的线程池。此线程池支持定时以及周期性执行任务的需求。

**13、Java中如何实现序列化，有什么意义？**

序列化就是一种用来处理对象流的机制，所谓对象流也就是将对象的内容进行流化。可以对流化后的对象进行读写操作，也可将流化后的对象传输于网络之间。序列化是为了解决对象流读写操作时可能引发的问题（如果不进行序列化可能会存在数据乱序的问题）。

要实现序列化，需要让一个类实现Serializable接口，该接口是一个标识性接口，标注该类对象是可被序列化的，然后使用一个输出流来构造一个对象输出流并通过writeObject(Object)方法就可以将实现对象写出（即保存其状态）；如果需要反序列化则可以用一个输入流建立对象输入流，然后通过readObject方法从流中读取对象。序列化除了能够实现对象的持久化之外，还能够用于对象的深度克隆。

**14、产生死锁的条件**

1. 互斥条件：一个资源每次只能被一个进程使用。
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:进程已获得的资源，在末使用完之前，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。

**15、什么是线程饿死，什么是活锁？**

线程饿死和活锁虽然不想是死锁一样的常见问题，但是对于并发编程的设计者来说就像一次邂逅一样。

当所有线程阻塞，或者由于需要的资源无效而不能处理，不存在非阻塞线程使资源可用。JavaAPI中线程活锁可能发生在以下情形：

* 当所有线程在程序中执行Object.wait(0)，参数为0的wait方法。程序将发生活锁直到在相应的对象上有线程调用Object.notify()或者Object.notifyAll()。
* 当所有线程卡在无限循环中。

**16、什么导致线程阻塞**

阻塞指的是暂停一个线程的执行以等待某个条件发生（如某资源就绪），学过操作系统的同学对它一定已经很熟悉了。Java 提供了大量方法来支持阻塞，下面让我们逐一分析。

方法说明sleep()sleep() 允许 指定以毫秒为单位的一段时间作为参数，它使得线程在指定的时间内进入阻塞状态，不能得到CPU 时间，指定的时间一过，线程重新进入可执行状态。典型地，sleep() 被用在等待某个资源就绪的情形：测试发现条件不满足后，让线程阻塞一段时间后重新测试，直到条件满足为止suspend() 和 resume()两个方法配套使用，suspend()使得线程进入阻塞状态，并且不会自动恢复，必须其对应的resume() 被调用，才能使得线程重新进入可执行状态。典型地，suspend() 和 resume() 被用在等待另一个线程产生的结果的情形：测试发现结果还没有产生后，让线程阻塞，另一个线程产生了结果后，调用 resume() 使其恢复。yield()yield() 使当前线程放弃当前已经分得的CPU 时间，但不使当前线程阻塞，即线程仍处于可执行状态，随时可能再次分得 CPU 时间。调用 yield() 的效果等价于调度程序认为该线程已执行了足够的时间从而转到另一个线程。wait() 和 notify()两个方法配套使用，wait() 使得线程进入阻塞状态，它有两种形式，一种允许 指定以毫秒为单位的一段时间作为参数，另一种没有参数，前者当对应的 notify() 被调用或者超出指定时间时线程重新进入可执行状态，后者则必须对应的 notify() 被调用.

**17、怎么检测一个线程是否持有对象监视器**

Thread类提供了一个holdsLock(Object obj)方法，当且仅当对象obj的监视器被某条线程持有的时候才会返回true，注意这是一个static方法，这意味着”某条线程”指的是当前线程。

**18、请说出与线程同步以及线程调度相关的方法。**

* wait()：使一个线程处于等待（阻塞）状态，并且释放所持有的对象的锁；
* sleep()：使一个正在运行的线程处于睡眠状态，是一个静态方法，调用此方法要处理InterruptedException异常；
* notify()：唤醒一个处于等待状态的线程，当然在调用此方法的时候，并不能确切的唤醒某一个等待状态的线程，而是由JVM确定唤醒哪个线程，而且与优先级无关；
* notityAll()：唤醒所有处于等待状态的线程，该方法并不是将对象的锁给所有线程，而是让它们竞争，只有获得锁的线程才能进入就绪状态；

**19、sleep() 、join（）、yield（）有什么区别**

* sleep()方法给其他线程运行机会时不考虑线程的优先级，因此会给低优先级的线程以运行的机会；yield()方法只会给相同优先级或更高优先级的线程以运行的机会；
* 线程执行sleep()方法后转入阻塞（blocked）状态，而执行yield()方法后转入就绪（ready）状态；
* sleep()方法声明抛出InterruptedException，而yield()方法没有声明任何异常；
* sleep()方法比yield()方法（跟操作系统CPU调度相关）具有更好的可移植性。

**20、wait(),notify()和suspend(),resume()之间的区别**

初看起来它们与 suspend() 和 resume() 方法对没有什么分别，但是事实上它们是截然不同的。区别的核心在于，前面叙述的所有方法，阻塞时都不会释放占用的锁（如果占用了的话），而这一对方法则相反。上述的核心区别导致了一系列的细节上的区别。

首先，前面叙述的所有方法都隶属于 Thread 类，但是这一对却直接隶属于 Object 类，也就是说，所有对象都拥有这一对方法。初看起来这十分不可思议，但是实际上却是很自然的，因为这一对方法阻塞时要释放占用的锁，而锁是任何对象都具有的，调用任意对象的 wait() 方法导致线程阻塞，并且该对象上的锁被释放。而调用 任意对象的notify()方法则导致从调用该对象的 wait() 方法而阻塞的线程中随机选择的一个解除阻塞（但要等到获得锁后才真正可执行）。

其次，前面叙述的所有方法都可在任何位置调用，但是这一对方法却必须在 synchronized 方法或块中调用，理由也很简单，只有在synchronized 方法或块中当前线程才占有锁，才有锁可以释放。同样的道理，调用这一对方法的对象上的锁必须为当前线程所拥有，这样才有锁可以释放。因此，这一对方法调用必须放置在这样的 synchronized 方法或块中，该方法或块的上锁对象就是调用这一对方法的对象。若不满足这一条件，则程序虽然仍能编译，但在运行时会出现IllegalMonitorStateException 异常。

wait() 和 notify() 方法的上述特性决定了它们经常和synchronized关键字一起使用，将它们和操作系统进程间通信机制作一个比较就会发现它们的相似性：synchronized方法或块提供了类似于操作系统原语的功能，它们的执行不会受到多线程机制的干扰，而这一对方法则相当于 block 和wakeup 原语（这一对方法均声明为 synchronized）。它们的结合使得我们可以实现操作系统上一系列精妙的进程间通信的算法（如信号量算法），并用于解决各种复杂的线程间通信问题。

关于 wait() 和 notify() 方法最后再说明两点：

第一：调用 notify() 方法导致解除阻塞的线程是从因调用该对象的 wait() 方法而阻塞的线程中随机选取的，我们无法预料哪一个线程将会被选择，所以编程时要特别小心，避免因这种不确定性而产生问题。

第二：除了 notify()，还有一个方法 notifyAll() 也可起到类似作用，唯一的区别在于，调用 notifyAll() 方法将把因调用该对象的 wait() 方法而阻塞的所有线程一次性全部解除阻塞。当然，只有获得锁的那一个线程才能进入可执行状态。

谈到阻塞，就不能不谈一谈死锁，略一分析就能发现，suspend() 方法和不指定超时期限的 wait() 方法的调用都可能产生死锁。遗憾的是，Java 并不在语言级别上支持死锁的避免，我们在编程中必须小心地避免死锁。

以上我们对 Java 中实现线程阻塞的各种方法作了一番分析，我们重点分析了 wait() 和 notify() 方法，因为它们的功能最强大，使用也最灵活，但是这也导致了它们的效率较低，较容易出错。实际使用中我们应该灵活使用各种方法，以便更好地达到我们的目的。

**21、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

**22、wait()方法和notify()/notifyAll()方法在放弃对象监视器时有什么区别**

wait()方法和notify()/notifyAll()方法在放弃对象监视器的时候的区别在于：wait()方法立即释放对象监视器，notify()/notifyAll()方法则会等待线程剩余代码执行完毕才会放弃对象监视器。

**23、Runnable和Callable的区别**

Runnable接口中的run()方法的返回值是void，它做的事情只是纯粹地去执行run()方法中的代码而已；Callable接口中的call()方法是有返回值的，是一个泛型，和Future、FutureTask配合可以用来获取异步执行的结果。

这其实是很有用的一个特性，因为多线程相比单线程更难、更复杂的一个重要原因就是因为多线程充满着未知性，某条线程是否执行了？某条线程执行了多久？某条线程执行的时候我们期望的数据是否已经赋值完毕？无法得知，我们能做的只是等待这条多线程的任务执行完毕而已。而Callable+Future/FutureTask却可以方便获取多线程运行的结果，可以在等待时间太长没获取到需要的数据的情况下取消该线程的任务。

**24、Thread类的sleep()方法和对象的wait()方法都可以让线程暂停执行，它们有什么区别？**

sleep()方法（休眠）是线程类（Thread）的静态方法，调用此方法会让当前线程暂停执行指定的时间，将执行机会（CPU）让给其他线程，但是对象的锁依然保持，因此休眠时间结束后会自动恢复。

wait()是Object类的方法，调用对象的wait()方法导致当前线程放弃对象的锁（线程暂停执行），进入对象的等待池（wait pool），只有调用对象的notify()方法（或notifyAll()方法）时才能唤醒等待池中的线程进入等锁池（lock pool），如果线程重新获得对象的锁就可以进入就绪状态。

**25、线程的sleep()方法和yield()方法有什么区别？**

1. sleep()方法给其他线程运行机会时不考虑线程的优先级，因此会给低优先级的线程以运行的机会；yield()方法只会给相同优先级或更高优先级的线程以运行的机会；
2. 线程执行sleep()方法后转入阻塞（blocked）状态，而执行yield()方法后转入就绪（ready）状态；
3. sleep()方法声明抛出InterruptedException，而yield()方法没有声明任何异常；
4. sleep()方法比yield()方法（跟操作系统CPU调度相关）具有更好的可移植性。

**26、为什么wait,nofity和nofityAll这些方法不放在Thread类当中**

一个很明显的原因是JAVA提供的锁是对象级的而不是线程级的，每个对象都有锁，通过线程获得。如果线程需要等待某些锁那么调用对象中的wait()方法就有意义了。如果wait()方法定义在Thread类中，线程正在等待的是哪个锁就不明显了。简单的说，由于wait，notify和notifyAll都是锁级别的操作，所以把他们定义在Object类中因为锁属于对象。

**27、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。

**28、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。

**29、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。

**30、一个线程如果出现了运行时异常怎么办？**

如果这个异常没有被捕获的话，这个线程就停止执行了。另外重要的一点是：如果这个线程持有某个某个对象的监视器，那么这个对象监视器会被立即释放

**31、Java当中有哪几种锁**

自旋锁: 自旋锁在JDK1.6之后就默认开启了。基于之前的观察，共享数据的锁定状态只会持续很短的时间，为了这一小段时间而去挂起和恢复线程有点浪费，所以这里就做了一个处理，让后面请求锁的那个线程在稍等一会，但是不放弃处理器的执行时间，看看持有锁的线程能否快速释放。为了让线程等待，所以需要让线程执行一个忙循环也就是自旋操作。在jdk6之后，引入了自适应的自旋锁，也就是等待的时间不再固定了，而是由上一次在同一个锁上的自旋时间及锁的拥有者状态来决定

偏向锁: 在JDK1.之后引入的一项锁优化，目的是消除数据在无竞争情况下的同步原语。进一步提升程序的运行性能。偏向锁就是偏心的偏，意思是这个锁会偏向第一个获得他的线程，如果接下来的执行过程中，改锁没有被其他线程获取，则持有偏向锁的线程将永远不需要再进行同步。偏向锁可以提高带有同步但无竞争的程序性能，也就是说他并不一定总是对程序运行有利，如果程序中大多数的锁都是被多个不同的线程访问，那偏向模式就是多余的，在具体问题具体分析的前提下，可以考虑是否使用偏向锁。

轻量级锁: 为了减少获得锁和释放锁所带来的性能消耗，引入了“偏向锁”和“轻量级锁”，所以在Java SE1.6里锁一共有四种状态，无锁状态，偏向锁状态，轻量级锁状态和重量级锁状态，它会随着竞争情况逐渐升级。锁可以升级但不能降级，意味着偏向锁升级成轻量级锁后不能降级成偏向锁

**32、如何在两个线程间共享数据**

通过在线程之间共享对象就可以了，然后通过wait/notify/notifyAll、await/signal/signalAll进行唤起和等待，比方说阻塞队列BlockingQueue就是为线程之间共享数据而设计的

**33、如何正确的使用wait()?使用if还是while？**

wait() 方法应该在循环调用，因为当线程获取到 CPU 开始执行的时候，其他条件可能还没有满足，所以在处理前，循环检测条件是否满足会更好。下面是一段标准的使用 wait 和 notify 方法的代码：

synchronized (obj) {

while (condition does not hold)

obj.wait(); // (Releases lock, and reacquires on wakeup)

... // Perform action appropriate to condition

}

**34、什么是线程局部变量ThreadLocal**

线程局部变量是局限于线程内部的变量，属于线程自身所有，不在多个线程间共享。Java提供ThreadLocal类来支持线程局部变量，是一种实现线程安全的方式。但是在管理环境下（如 web 服务器）使用线程局部变量的时候要特别小心，在这种情况下，工作线程的生命周期比任何应用变量的生命周期都要长。任何线程局部变量一旦在工作完成后没有释放，Java 应用就存在内存泄露的风险。

**35、ThreadLoal的作用是什么？**

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.

**36、ThreadLocal 原理分析**

ThreadLocal为解决多线程程序的并发问题提供了一种新的思路。ThreadLocal，顾名思义是线程的一个本地化对象，当工作于多线程中的对象使用ThreadLocal维护变量时，ThreadLocal为每个使用该变量的线程分配一个独立的变量副本，所以每一个线程都可以独立的改变自己的副本，而不影响其他线程所对应的副本。从线程的角度看，这个变量就像是线程的本地变量。

ThreadLocal类非常简单好用，只有四个方法，能用上的也就是下面三个方法：

* void set(T value)：设置当前线程的线程局部变量的值。
* T get()：获得当前线程所对应的线程局部变量的值。
* void remove()：删除当前线程中线程局部变量的值。

ThreadLocal是如何做到为每一个线程维护一份独立的变量副本的呢？在ThreadLocal类中有一个Map，键为线程对象，值是其线程对应的变量的副本，自己要模拟实现一个ThreadLocal类其实并不困难，代码如下所示：

import java.util.Collections;

import java.util.HashMap;

import java.util.Map;

public class MyThreadLocal<T> {

private Map<Thread, T> map = Collections.synchronizedMap(new HashMap<Thread, T>());

public void set(T newValue) {

map.put(Thread.currentThread(), newValue);

}

public T get() {

return map.get(Thread.currentThread());

}

public void remove() {

map.remove(Thread.currentThread());

}

}

**37、如果你提交任务时，线程池队列已满，这时会发生什么**

如果你使用的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；如果你使用的是有界队列比方说ArrayBlockingQueue的话，任务首先会被添加到ArrayBlockingQueue中，ArrayBlockingQueue满了，则会使用拒绝策略RejectedExecutionHandler处理满了的任务，默认是AbortPolicy。

**38、为什么要使用线程池**

避免频繁地创建和销毁线程，达到线程对象的重用。另外，使用线程池还可以根据项目灵活地控制并发的数目。

**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、Thread.sleep(0)的作用是什么**

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

**41、什么是CAS**

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

**42、什么是乐观锁和悲观锁**

乐观锁：乐观锁认为竞争不总是会发生，因此它不需要持有锁，将比较-替换这两个动作作为一个原子操作尝试去修改内存中的变量，如果失败则表示发生冲突，那么就应该有相应的重试逻辑。

悲观锁：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。

**43、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？

**44、ConcurrentHashMap的工作原理**

ConcurrentHashMap在jdk 1.6和jdk 1.8实现原理是不同的.

jdk 1.6: ConcurrentHashMap是线程安全的，但是与Hashtablea相比，实现线程安全的方式不同。Hashtable是通过对hash表结构进行锁定，是阻塞式的，当一个线程占有这个锁时，其他线程必须阻塞等待其释放锁。ConcurrentHashMap是采用分离锁的方式，它并没有对整个hash表进行锁定，而是局部锁定，也就是说当一个线程占有这个局部锁时，不影响其他线程对hash表其他地方的访问。具体实现:ConcurrentHashMap内部有一个Segment

jdk 1.8 在jdk 8中，ConcurrentHashMap不再使用Segment分离锁，而是采用一种乐观锁CAS算法来实现同步问题，但其底层还是“数组+链表->红黑树”的实现。

**45、CyclicBarrier和CountDownLatch区别**

这两个类非常类似，都在java.util.concurrent下，都可以用来表示代码运行到某个点上，二者的区别在于：

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

CyclicBarrier只能唤起一个任务，CountDownLatch可以唤起多个任务

CyclicBarrier可重用，CountDownLatch不可重用，计数值为0该CountDownLatch就不可再用了

**46、java中的++操作符线程安全么？**

不是线程安全的操作。它涉及到多个指令，如读取变量值，增加，然后存储回内存，这个过程可能会出现多个线程交差

**47、有三个线程T1，T2，T3，怎么确保它们按顺序执行？**

在多线程中有多种方法让线程按特定顺序执行，你可以用线程类的join()方法在一个线程中启动另一个线程，另外一个线程完成该线程继续执行。为了确保三个线程的顺序你应该先启动最后一个(T3调用T2，T2调用T1)，这样T1就会先完成而T3最后完成。

**48、如何在Java中创建Immutable对象？**

这个问题看起来和多线程没什么关系， 但不变性有助于简化已经很复杂的并发程序。Immutable对象可以在没有同步的情况下共享，降低了对该对象进行并发访问时的同步化开销。可是Java没有@Immutable这个注解符，要创建不可变类，要实现下面几个步骤：通过构造方法初始化所有成员、对变量不要提供setter方法、将所有的成员声明为私有的，这样就不允许直接访问这些成员、在getter方法中，不要直接返回对象本身，而是克隆对象，并返回对象的拷贝。

**49、你有哪些多线程开发良好的实践？**

* 给线程命名
* 最小化同步范围
* 优先使用volatile
* 尽可能使用更高层次的并发工具而非wait和notify()来实现线程通信,如BlockingQueue,Semeaphore
* 优先使用并发容器而非同步容器.
* 考虑使用线程池

**50、可以创建Volatile数组吗？**

Java 中可以创建 volatile类型数组，不过只是一个指向数组的引用，而不是整个数组。如果改变引用指向的数组，将会受到volatile 的保护，但是如果多个线程同时改变数组的元素，volatile标示符就不能起到之前的保护作用了

**51、Volatile关键字的作用**

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

* 多线程主要围绕可见性和原子性两个特性而展开，使用volatile关键字修饰的变量，保证了其在多线程之间的可见性，即每次读取到volatile变量，一定是最新的数据
* 代码底层执行不像我们看到的高级语言—-Java程序这么简单，它的执行是Java代码–>字节码–>根据字节码执行对应的C/C++代码–>C/C++代码被编译成汇编语言–>和硬件电路交互，现实中，为了获取更好的性能JVM可能会对指令进行重排序，多线程下可能会出现一些意想不到的问题。使用volatile则会对禁止语义重排序，当然这也一定程度上降低了代码执行效率

从实践角度而言，volatile的一个重要作用就是和CAS结合，保证了原子性，详细的可以参见java.util.concurrent.atomic包下的类，比如AtomicInteger。

**52、volatile能使得一个非原子操作变成原子操作吗？**

一个典型的例子是在类中有一个 long 类型的成员变量。如果你知道该成员变量会被多个线程访问，如计数器、价格等，你最好是将其设置为 volatile。为什么？因为 Java 中读取 long 类型变量不是原子的，需要分成两步，如果一个线程正在修改该 long 变量的值，另一个线程可能只能看到该值的一半（前 32 位）。但是对一个 volatile 型的 long 或 double 变量的读写是原子。

一种实践是用 volatile 修饰 long 和 double 变量，使其能按原子类型来读写。double 和 long 都是64位宽，因此对这两种类型的读是分为两部分的，第一次读取第一个 32 位，然后再读剩下的 32 位，这个过程不是原子的，但 Java 中 volatile 型的 long 或 double 变量的读写是原子的。volatile 修复符的另一个作用是提供内存屏障（memory barrier），例如在分布式框架中的应用。简单的说，就是当你写一个 volatile 变量之前，Java 内存模型会插入一个写屏障（write barrier），读一个 volatile 变量之前，会插入一个读屏障（read barrier）。意思就是说，在你写一个 volatile 域时，能保证任何线程都能看到你写的值，同时，在写之前，也能保证任何数值的更新对所有线程是可见的，因为内存屏障会将其他所有写的值更新到缓存。

**53、volatile类型变量提供什么保证？**

volatile 主要有两方面的作用:1.避免指令重排2.可见性保证.例如，JVM 或者 JIT为了获得更好的性能会对语句重排序，但是 volatile 类型变量即使在没有同步块的情况下赋值也不会与其他语句重排序。volatile 提供 happens-before 的保证，确保一个线程的修改能对其他线程是可见的。某些情况下，volatile 还能提供原子性，如读 64 位数据类型，像 long 和 double 都不是原子的(低32位和高32位)，但 volatile 类型的 double 和 long 就是原子的.

**54、Java 中，编写多线程程序的时候你会遵循哪些最佳实践？**

这是我在写Java 并发程序的时候遵循的一些最佳实践：

* 给线程命名，这样可以帮助调试。
* 最小化同步的范围，而不是将整个方法同步，只对关键部分做同步。
* 如果可以，更偏向于使用 volatile 而不是 synchronized。
* 使用更高层次的并发工具，而不是使用 wait() 和 notify() 来实现线程间通信，如 BlockingQueue，CountDownLatch 及 Semeaphore。
* 优先使用并发集合，而不是对集合进行同步。并发集合提供更好的可扩展性。

**55、说出至少 5 点在 Java 中使用线程的最佳实践。**

这个问题与之前的问题类似，你可以使用上面的答案。对线程来说，你应该：

* 对线程命名
* 将线程和任务分离，使用线程池执行器来执行 Runnable 或 Callable。
* 使用线程池

**56、Java中如何获取到线程dump文件**

死循环、死锁、阻塞、页面打开慢等问题，打线程dump是最好的解决问题的途径。所谓线程dump也就是线程堆栈，获取到线程堆栈有两步：

* 获取到线程的pid，可以通过使用jps命令，在Linux环境下还可以使用ps -ef | grep java
* 打印线程堆栈，可以通过使用jstack pid命令，在Linux环境下还可以使用kill -3 pid

另外提一点，Thread类提供了一个getStackTrace()方法也可以用于获取线程堆栈。这是一个实例方法，因此此方法是和具体线程实例绑定的，每次获取获取到的是具体某个线程当前运行的堆栈。

**57、高并发、任务执行时间短的业务怎样使用线程池？并发不高、任务执行时间长的业务怎样使用线程池？并发高、业务执行时间长的业务怎样使用线程池？**

这是我在并发编程网上看到的一个问题，把这个问题放在最后一个，希望每个人都能看到并且思考一下，因为这个问题非常好、非常实际、非常专业。关于这个问题，个人看法是：

1. 高并发、任务执行时间短的业务，线程池线程数可以设置为CPU核数+1，减少线程上下文的切换
2. 并发不高、任务执行时间长的业务要区分开看：

* 假如是业务时间长集中在IO操作上，也就是IO密集型的任务，因为IO操作并不占用CPU，所以不要让所有的CPU闲下来，可以加大线程池中的线程数目，让CPU处理更多的业务
* 假如是业务时间长集中在计算操作上，也就是计算密集型任务，这个就没办法了，和（1）一样吧，线程池中的线程数设置得少一些，减少线程上下文的切换

1. 并发高、业务执行时间长，解决这种类型任务的关键不在于线程池而在于整体架构的设计，看看这些业务里面某些数据是否能做缓存是第一步，增加服务器是第二步，至于线程池的设置，设置参考（2）。
2. 业务执行时间长的问题，也可能需要分析一下，看看能不能使用中间件对任务进行拆分和解耦。

**58、作业(进程)调度算法**

1. 先来先服务调度算法(FCFS) 每次调度都是从后备作业队列中选择一个或多个最先进入该队列的作业，将它们调入内存，为它们分配资源、创建进程，然后放入就绪队列。
2. 短作业(进程)优先调度算法(SPF) 短作业优先(SJF)的调度算法是从后备队列中选择一个或若干个估计运行时间最短的作业，将它们调入内存运行。缺点:长作业的运行得不到保证
3. 优先权调度算法(HPF) 当把该算法用于作业调度时，系统将从后备队列中选择若干个优先权最高的作业装入内存。当用于进程调度时，该算法是把处理机分配给就绪队列中优先权最高的进程，这时，又可进一步把该算法分成如下两种。可以分为:

* 非抢占式优先权算法
* 抢占式优先权调度算法

1. 高响应比优先调度算法(HRN) 每次选择高响应比最大的作业执行，响应比=(等待时间+要求服务时间)/要求服务时间。该算法同时考虑了短作业优先和先来先服务。

* 如果作业的等待时间相同，则要求服务的时间愈短，其优先权愈高，因而该算法有利于短作业。
* 当要求服务的时间相同时，作业的优先权决定于其等待时间，等待时间愈长，其优先权愈高，因而它实现的是先来先服务。
* 对于长作业，作业的优先级可以随等待时间的增加而提高，当其等待时间足够长时，其优先级便可升到很高，从而也可获得处理机。简言之，该算法既照顾了短作业，又考虑了作业到达的先后次序，不会使长作业长期得不到服务。因此，该算法实现了一种较好的折衷。当然，在利用该算法时，每要进行调度之前，都须先做响应比的计算，这会增加系统开销。

1. 时间片轮转法（RR） 在早期的时间片轮转法中，系统将所有的就绪进程按先来先服务的原则排成一个队列，每次调度时，把CPU分配给队首进程，并令其执行一个时间片。时间片的大小从几ms到几百ms。当执行的时间片用完时，由一个计时器发出时钟中断请求，调度程序便据此信号来停止该进程的执行，并将它送往就绪队列的末尾；然后，再把处理机分配给就绪队列中新的队首进程，同时也让它执行一个时间片。这样就可以保证就绪队列中的所有进程在一给定的时间内均能获得一时间片的处理机执行时间。换言之，系统能在给定的时间内响应所有用户的请求。
2. 多级反馈队列调度算法 它是目前被公认的一种较好的进程调度算法。

* 应设置多个就绪队列，并为各个队列赋予不同的优先级。第一个队列的优先级最高，第二个队列次之，其余各队列的优先权逐个降低。该算法赋予各个队列中进程执行时间片的大小也各不相同，在优先权愈高的队列中，为每个进程所规定的执行时间片就愈小。例如，第二个队列的时间片要比第一个队列的时间片长一倍，……，第i+1个队列的时间片要比第i个队列的时间片长一倍。
* 当一个新进程进入内存后，首先将它放入第一队列的末尾，按FCFS原则排队等待调度。当轮到该进程执行时，如它能在该时间片内完成，便可准备撤离系统；如果它在一个时间片结束时尚未完成，调度程序便将该进程转入第二队列的末尾，再同样地按FCFS原则等待调度执行；如果它在第二队列中运行一个时间片后仍未完成，再依次将它放入第三队列，……，如此下去，当一个长作业(进程)从第一队列依次降到第n队列后，在第n 队列便采取按时间片轮转的方式运行。
* 仅当第一队列空闲时，调度程序才调度第二队列中的进程运行；仅当第1～(i-1)队列均空时，才会调度第i队列中的进程运行。如果处理机正在第i队列中为某进程服务时，又有新进程进入优先权较高的队列(第1～(i-1)中的任何一个队列)，则此时新进程将抢占正在运行进程的处理机，即由调度程序把正在运行的进程放回到第i队列的末尾，把处理机分配给新到的高优先权进程。

**59、讲讲线程池的实现原理**

首先要明确为什么要使用线程池，使用线程池会带来什么好处？

* 线程是稀缺资源，不能频繁的创建。
* 应当将其放入一个池子中，可以给其他任务进行复用。
* 解耦作用，线程的创建于执行完全分开，方便维护。

**60、创建一个线程池**

以一个使用较多的

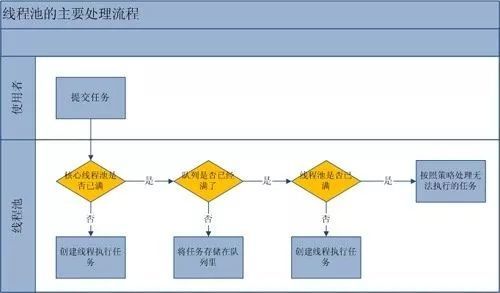
ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue, RejectedExecutionHandler handler);

为例：

* 其中的 corePoolSize 为线程池的基本大小。
* maximumPoolSize 为线程池最大线程大小。
* keepAliveTime 和 unit 则是线程空闲后的存活时间。
* workQueue 用于存放任务的阻塞队列。
* handler 当队列和最大线程池都满了之后的饱和策略。

**61、处理流程**

当提交一个任务到线程池时它的执行流程是怎样的呢？



首先第一步会判断核心线程数有没有达到上限，如果没有则创建线程(会获取全局锁)，满了则会将任务丢进阻塞队列。

如果队列也满了则需要判断最大线程数是否达到上限，如果没有则创建线程(获取全局锁)，如果最大线程数也满了则会根据饱和策略处理。

常用的饱和策略有:

* 直接丢弃任务。
* 调用者线程处理。
* 丢弃队列中的最近任务，执行当前任务。

所以当线程池完成预热之后都是将任务放入队列，接着由工作线程一个个从队列里取出执行。

**62、合理配置线程池**

线程池并不是配置越大越好，而是要根据任务的熟悉来进行划分：如果是 CPU 密集型任务应当分配较少的线程，比如 CPU 个数相当的大小。

如果是 IO 密集型任务，由于线程并不是一直在运行，所以可以尽可能的多配置线程，比如 CPU 个数 \* 2 。

当是一个混合型任务，可以将其拆分为 CPU 密集型任务以及 IO 密集型任务，这样来分别配置。

**63、synchronize 实现原理**

众所周知 Synchronize 关键字是解决并发问题常用解决方案，有以下三种使用方式:

* 同步普通方法，锁的是当前对象。
* 同步静态方法，锁的是当前 Class 对象。
* 同步块，锁的是 {} 中的对象。

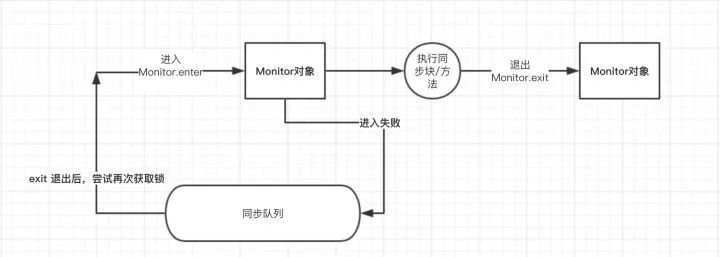
实现原理：JVM 是通过进入、退出对象监视器( Monitor )来实现对方法、同步块的同步的。

具体实现是在编译之后在同步方法调用前加入一个 monitor.enter 指令，在退出方法和异常处插入 monitor.exit 的指令。

其本质就是对一个对象监视器( Monitor )进行获取，而这个获取过程具有排他性从而达到了同一时刻只能一个线程访问的目的。

而对于没有获取到锁的线程将会阻塞到方法入口处，直到获取锁的线程 monitor.exit 之后才能尝试继续获取锁。

流程图如下:



synchronize 很多都称之为重量锁，JDK1.6 中对 synchronize 进行了各种优化，为了能减少获取和释放锁带来的消耗引入了偏向锁和轻量锁。

**64、轻量锁**

当代码进入同步块时，如果同步对象为无锁状态时，当前线程会在栈帧中创建一个锁记录(Lock Record)区域，同时将锁对象的对象头中 Mark Word 拷贝到锁记录中，再尝试使用 CAS 将 Mark Word 更新为指向锁记录的指针。

如果更新成功，当前线程就获得了锁。

如果更新失败 JVM 会先检查锁对象的 Mark Word 是否指向当前线程的锁记录。

如果是则说明当前线程拥有锁对象的锁，可以直接进入同步块。

不是则说明有其他线程抢占了锁，如果存在多个线程同时竞争一把锁，轻量锁就会膨胀为重量锁。

**65、解锁**

轻量锁的解锁过程也是利用 CAS 来实现的，会尝试锁记录替换回锁对象的 Mark Word 。如果替换成功则说明整个同步操作完成，失败则说明有其他线程尝试获取锁，这时就会唤醒被挂起的线程(此时已经膨胀为重量锁)

轻量锁能提升性能的原因是：

认为大多数锁在整个同步周期都不存在竞争，所以使用 CAS 比使用互斥开销更少。但如果锁竞争激烈，轻量锁就不但有互斥的开销，还有 CAS 的开销，甚至比重量锁更慢。

**66、偏向锁**

为了进一步的降低获取锁的代价，JDK1.6 之后还引入了偏向锁。

偏向锁的特征是:锁不存在多线程竞争，并且应由一个线程多次获得锁。

当线程访问同步块时，会使用 CAS 将线程 ID 更新到锁对象的 Mark Word 中，如果更新成功则获得偏向锁，并且之后每次进入这个对象锁相关的同步块时都不需要再次获取锁了。

**67、释放锁**

当有另外一个线程获取这个锁时，持有偏向锁的线程就会释放锁，释放时会等待全局安全点(这一时刻没有字节码运行)，接着会暂停拥有偏向锁的线程，根据锁对象目前是否被锁来判定将对象头中的 Mark Word 设置为无锁或者是轻量锁状态。

偏向锁可以提高带有同步却没有竞争的程序性能，但如果程序中大多数锁都存在竞争时，那偏向锁就起不到太大作用。可以使用 -XX:-userBiasedLocking=false 来关闭偏向锁，并默认进入轻量锁。

1、什么是进程，什么是线程，为什么需要多线程编程？

进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动，是操作系统进行资源分配和调度的一个独立单位；

线程是进程的一个实体，是CPU调度和分派的基本单位，是比进程更小的能独立运行的基本单位。线程的划分尺度小于进程，这使得多线程程序的并发性高；进程在执行时通常拥有独立的内存单元，而线程之间可以共享内存。

使用多线程的编程通常能够带来更好的性能和用户体验，但是多线程的程序对于其他程序是不友好的，因为它可能占用了更多的CPU资源。当然，也不是线程越多，程序的性能就越好，因为线程之间的调度和切换也会浪费CPU时间。时下很时髦的Node.js就采用了单线程异步I/O的工作模式。

2、什么是线程安全

如果你的代码在多线程下执行和在单线程下执行永远都能获得一样的结果，那么你的代码就是线程安全的。

这个问题有值得一提的地方，就是线程安全也是有几个级别的：

* 不可变。像String、Integer、Long这些，都是final类型的类，任何一个线程都改变不了它们的值，要改变除非新创建一个，因此这些不可变对象不需要任何同步手段就可以直接在多线程环境下使用
* 绝对线程安全。不管运行时环境如何，调用者都不需要额外的同步措施。要做到这一点通常需要付出许多额外的代价，Java中标注自己是线程安全的类，实际上绝大多数都不是线程安全的，不过绝对线程安全的类，Java中也有，比方说CopyOnWriteArrayList、CopyOnWriteArraySet
* 相对线程安全。相对线程安全也就是我们通常意义上所说的线程安全，像Vector这种，add、remove方法都是原子操作，不会被打断，但也仅限于此，如果有个线程在遍历某个Vector、有个线程同时在add这个Vector，99%的情况下都会出现ConcurrentModificationException，也就是fail-fast机制。
* 线程非安全。这个就没什么好说的了，ArrayList、LinkedList、HashMap等都是线程非安全的类

3、编写多线程程序有几种实现方式？

Java 5以前实现多线程有两种实现方法：一种是继承Thread类；另一种是实现Runnable接口。 两种方式都要通过重写run()方法来定义线程的行为，推荐使用后者，因为Java中的继承是单继承，一个类有一个父类，如果继承了Thread类就无法再继承其他类了，显然使用Runnable接口更为灵活。

Java 5以后创建线程还有第三种方式：实现Callable接口，该接口中的call方法可以在线程执行结束时产生一个返回值。

4、synchronized关键字的用法？

synchronized关键字可以将对象或者方法标记为同步，以实现对对象和方法的互斥访问，可以用synchronized(对象) { … }定义同步代码块，或者在声明方法时将synchronized作为方法的修饰符。

5、简述synchronized 和java.util.concurrent.locks.Lock的异同？

Lock是Java 5以后引入的新的API，和关键字synchronized相比主要相同点：Lock 能完成synchronized所实现的所有功能；主要不同点：Lock有比synchronized更精确的线程语义和更好的性能，而且不强制性的要求一定要获得锁。synchronized会自动释放锁，而Lock一定要求程序员手工释放，并且最好在finally 块中释放（这是释放外部资源的最好的地方）。

6、当一个线程进入一个对象的synchronized方法A之后，其它线程是否可进入此对象的synchronized方法B？

不能。其它线程只能访问该对象的非同步方法，同步方法则不能进入。因为非静态方法上的synchronized修饰符要求执行方法时要获得对象的锁，如果已经进入A方法说明对象锁已经被取走，那么试图进入B方法的线程就只能在等锁池（注意不是等待池哦）中等待对象的锁。

7、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：

* ReentrantLock可以对获取锁的等待时间进行设置，这样就避免了死锁
* ReentrantLock可以获取各种锁的信息
* 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.

8、举例说明同步和异步。

如果系统中存在临界资源（资源数量少于竞争资源的线程数量的资源），例如正在写的数据以后可能被另一个线程读到，或者正在读的数据可能已经被另一个线程写过了，那么这些数据就必须进行同步存取（数据库操作中的排他锁就是最好的例子）。当应用程序在对象上调用了一个需要花费很长时间来执行的方法，并且不希望让程序等待方法的返回时，就应该使用异步编程，在很多情况下采用异步途径往往更有效率。事实上，所谓的同步就是指阻塞式操作，而异步就是非阻塞式操作。

9、启动一个线程是调用run()还是start()方法？

启动一个线程是调用start()方法，使线程所代表的虚拟处理机处于可运行状态，这意味着它可以由JVM 调度并执行，这并不意味着线程就会立即运行。run()方法是线程启动后要进行回调（callback）的方法。

10、为什么需要run()和start()方法，我们可以只用run()方法来完成任务吗？

我们需要run()&start()这两个方法是因为JVM创建一个单独的线程不同于普通方法的调用，所以这项工作由线程的start方法来完成，start由本地方法实现，需要显示地被调用，使用这俩个方法的另外一个好处是任何一个对象都可以作为线程运行，只要实现了Runnable接口，这就避免因继承了Thread类而造成的Java的多继承问题。

11、什么是线程池（thread pool）？

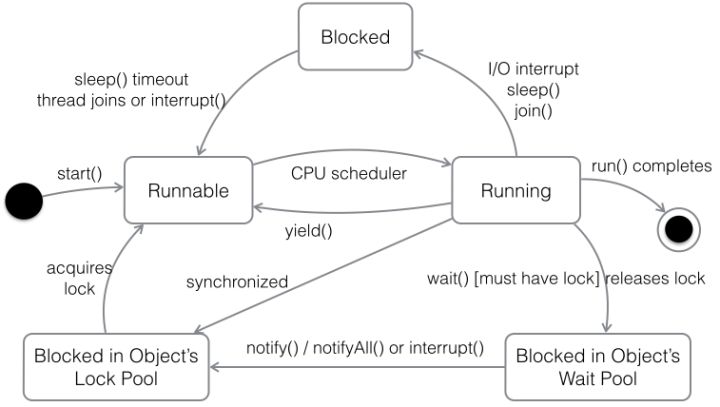
在面向对象编程中，创建和销毁对象是很费时间的，因为创建一个对象要获取内存资源或者其它更多资源。

在Java中更是如此，虚拟机将试图跟踪每一个对象，以便能够在对象销毁后进行垃圾回收。所以提高服务程序效率的一个手段就是尽可能减少创建和销毁对象的次数，特别是一些很耗资源的对象创建和销毁，这就是“池化资源”技术产生的原因。线程池顾名思义就是事先创建若干个可执行的线程放入一个池（容器）中，需要的时候从池中获取线程不用自行创建，使用完毕不需要销毁线程而是放回池中，从而减少创建和销毁线程对象的开销。

Java 5+中的Executor接口定义一个执行线程的工具。它的子类型即线程池接口是ExecutorService。要配置一个线程池是比较复杂的，尤其是对于线程池的原理不是很清楚的情况下，因此在工具类Executors面提供了一些静态工厂方法，生成一些常用的线程池，如下所示：

* newSingleThreadExecutor：创建一个单线程的线程池。这个线程池只有一个线程在工作，也就是相当于单线程串行执行所有任务。如果这个唯一的线程因为异常结束，那么会有一个新的线程来替代它。此线程池保证所有任务的执行顺序按照任务的提交顺序执行。
* newFixedThreadPool：创建固定大小的线程池。每次提交一个任务就创建一个线程，直到线程达到线程池的最大大小。线程池的大小一旦达到最大值就会保持不变，如果某个线程因为执行异常而结束，那么线程池会补充一个新线程。
* newCachedThreadPool：创建一个可缓存的线程池。如果线程池的大小超过了处理任务所需要的线程，那么就会回收部分空闲（60秒不执行任务）的线程，当任务数增加时，此线程池又可以智能的添加新线程来处理任务。此线程池不会对线程池大小做限制，线程池大小完全依赖于操作系统（或者说JVM）能够创建的最大线程大小。
* newScheduledThreadPool：创建一个大小无限的线程池。此线程池支持定时以及周期性执行任务的需求。
* newSingleThreadExecutor：创建一个单线程的线程池。此线程池支持定时以及周期性执行任务的需求。

12、线程的基本状态以及状态之间的关系？



其中Running表示运行状态；Runnable表示就绪状态（万事俱备，只欠CPU）；Blocked表示阻塞状态；阻塞状态又有多种情况，可能是因为调用wait()方法进入等待池，也可能是执行同步方法或同步代码块进入等锁池，或者是调用了sleep()方法或join()方法等待休眠或其他线程结束，或是因为发生了I/O中断。

13、Java中如何实现序列化，有什么意义？

序列化就是一种用来处理对象流的机制，所谓对象流也就是将对象的内容进行流化。可以对流化后的对象进行读写操作，也可将流化后的对象传输于网络之间。序列化是为了解决对象流读写操作时可能引发的问题（如果不进行序列化可能会存在数据乱序的问题）。

要实现序列化，需要让一个类实现Serializable接口，该接口是一个标识性接口，标注该类对象是可被序列化的，然后使用一个输出流来构造一个对象输出流并通过writeObject(Object)方法就可以将实现对象写出（即保存其状态）；如果需要反序列化则可以用一个输入流建立对象输入流，然后通过readObject方法从流中读取对象。序列化除了能够实现对象的持久化之外，还能够用于对象的深度克隆。

14、产生死锁的条件

* 互斥条件：一个资源每次只能被一个进程使用。
* 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:进程已获得的资源，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。

15、什么是线程饿死，什么是活锁？

线程饿死和活锁虽然不想是死锁一样的常见问题，但是对于并发编程的设计者来说就像一次邂逅一样。

当所有线程阻塞，或者由于需要的资源无效而不能处理，不存在非阻塞线程使资源可用。JavaAPI中线程活锁可能发生在以下情形：

* 当所有线程在程序中执行Object.wait(0)，参数为0的wait方法。程序将发生活锁直到在相应的对象上有线程调用Object.notify()或者Object.notifyAll()。
* 当所有线程卡在无限循环中。

16、什么导致线程阻塞

阻塞指的是暂停一个线程的执行以等待某个条件发生（如某资源就绪），学过操作系统的同学对它一定已经很熟悉了。Java 提供了大量方法来支持阻塞，下面让我们逐一分析。

方法说明sleep()sleep() 允许 指定以毫秒为单位的一段时间作为参数，它使得线程在指定的时间内进入阻塞状态，不能得到CPU 时间，指定的时间一过，线程重新进入可执行状态。 典型地，sleep() 被用在等待某个资源就绪的情形：测试发现条件不满足后，让线程阻塞一段时间后重新测试，直到条件满足为止suspend() 和 resume()两个方法配套使用，suspend()使得线程进入阻塞状态，并且不会自动恢复，必须其对应的resume() 被调用，才能使得线程重新进入可执行状态。典型地，suspend() 和 resume() 被用在等待另一个线程产生的结果的情形：测试发现结果还没有产生后，让线程阻塞，另一个线程产生了结果后，调用 resume() 使其恢复。yield()yield() 使当前线程放弃当前已经分得的CPU 时间，但不使当前线程阻塞，即线程仍处于可执行状态，随时可能再次分得 CPU 时间。调用 yield() 的效果等价于调度程序认为该线程已执行了足够的时间从而转到另一个线程。wait() 和 notify()两个方法配套使用，wait() 使得线程进入阻塞状态，它有两种形式，一种允许 指定以毫秒为单位的一段时间作为参数，另一种没有参数，前者当对应的 notify() 被调用或者超出指定时间时线程重新进入可执行状态，后者则必须对应的 notify() 被调用.

17、怎么检测一个线程是否持有对象监视器

Thread类提供了一个holdsLock(Object obj)方法，当且仅当对象obj的监视器被某条线程持有的时候才会返回true，注意这是一个static方法，这意味着”某条线程”指的是当前线程。

18、请说出与线程同步以及线程调度相关的方法。

* wait()：使一个线程处于等待（阻塞）状态，并且释放所持有的对象的锁；
* sleep()：使一个正在运行的线程处于睡眠状态，是一个静态方法，调用此方法要处理InterruptedException异常；
* notify()：唤醒一个处于等待状态的线程，当然在调用此方法的时候，并不能确切的唤醒某一个等待状态的线程，而是由JVM确定唤醒哪个线程，而且与优先级无关；
* notityAll()：唤醒所有处于等待状态的线程，该方法并不是将对象的锁给所有线程，而是让它们竞争，只有获得锁的线程才能进入就绪状态；

19、sleep() 、join（）、yield（）有什么区别

* sleep()方法给其他线程运行机会时不考虑线程的优先级，因此会给低优先级的线程以运行的机会；yield()方法只会给相同优先级或更高优先级的线程以运行的机会；
* 线程执行sleep()方法后转入阻塞（blocked）状态，而执行yield()方法后转入就绪（ready）状态；
* sleep()方法声明抛出InterruptedException，而yield()方法没有声明任何异常；
* sleep()方法比yield()方法（跟操作系统CPU调度相关）具有更好的可移植性。

20、wait(),notify()和suspend(),resume()之间的区别

初看起来它们与 suspend() 和 resume() 方法对没有什么分别，但是事实上它们是截然不同的。区别的核心在于，前面叙述的所有方法，阻塞时都不会释放占用的锁（如果占用了的话），而这一对方法则相反。上述的核心区别导致了一系列的细节上的区别。

首先，前面叙述的所有方法都隶属于 Thread 类，但是这一对却直接隶属于 Object 类，也就是说，所有对象都拥有这一对方法。初看起来这十分不可思议，但是实际上却是很自然的，因为这一对方法阻塞时要释放占用的锁，而锁是任何对象都具有的，调用任意对象的 wait() 方法导致线程阻塞，并且该对象上的锁被释放。而调用 任意对象的notify()方法则导致从调用该对象的 wait() 方法而阻塞的线程中随机选择的一个解除阻塞（但要等到获得锁后才真正可执行）。

其次，前面叙述的所有方法都可在任何位置调用，但是这一对方法却必须在 synchronized 方法或块中调用，理由也很简单，只有在synchronized 方法或块中当前线程才占有锁，才有锁可以释放。同样的道理，调用这一对方法的对象上的锁必须为当前线程所拥有，这样才有锁可以释放。因此，这一对方法调用必须放置在这样的 synchronized 方法或块中，该方法或块的上锁对象就是调用这一对方法的对象。若不满足这一条件，则程序虽然仍能编译，但在运行时会出现IllegalMonitorStateException 异常。

wait() 和 notify() 方法的上述特性决定了它们经常和synchronized关键字一起使用，将它们和操作系统进程间通信机制作一个比较就会发现它们的相似性：synchronized方法或块提供了类似于操作系统原语的功能，它们的执行不会受到多线程机制的干扰，而这一对方法则相当于 block 和wakeup 原语（这一对方法均声明为 synchronized）。它们的结合使得我们可以实现操作系统上一系列精妙的进程间通信的算法（如信号量算法），并用于解决各种复杂的线程间通信问题。

关于 wait() 和 notify() 方法最后再说明两点：

第一：调用 notify() 方法导致解除阻塞的线程是从因调用该对象的 wait() 方法而阻塞的线程中随机选取的，我们无法预料哪一个线程将会被选择，所以编程时要特别小心，避免因这种不确定性而产生问题。

第二：除了 notify()，还有一个方法 notifyAll() 也可起到类似作用，唯一的区别在于，调用 notifyAll() 方法将把因调用该对象的 wait() 方法而阻塞的所有线程一次性全部解除阻塞。当然，只有获得锁的那一个线程才能进入可执行状态。

谈到阻塞，就不能不谈一谈死锁，略一分析就能发现，suspend() 方法和不指定超时期限的 wait() 方法的调用都可能产生死锁。遗憾的是，Java 并不在语言级别上支持死锁的避免，我们在编程中必须小心地避免死锁。

以上我们对 Java 中实现线程阻塞的各种方法作了一番分析，我们重点分析了 wait() 和 notify() 方法，因为它们的功能最强大，使用也最灵活，但是这也导致了它们的效率较低，较容易出错。实际使用中我们应该灵活使用各种方法，以便更好地达到我们的目的。

21、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

22、wait()方法和notify()/notifyAll()方法在放弃对象监视器时有什么区别

wait()方法和notify()/notifyAll()方法在放弃对象监视器的时候的区别在于：wait()方法立即释放对象监视器，notify()/notifyAll()方法则会等待线程剩余代码执行完毕才会放弃对象监视器。

23、Runnable和Callable的区别

Runnable接口中的run()方法的返回值是void，它做的事情只是纯粹地去执行run()方法中的代码而已；Callable接口中的call()方法是有返回值的，是一个泛型，和Future、FutureTask配合可以用来获取异步执行的结果。

这其实是很有用的一个特性，因为多线程相比单线程更难、更复杂的一个重要原因就是因为多线程充满着未知性，某条线程是否执行了？某条线程执行了多久？某条线程执行的时候我们期望的数据是否已经赋值完毕？无法得知，我们能做的只是等待这条多线程的任务执行完毕而已。而Callable+Future/FutureTask却可以方便获取多线程运行的结果，可以在等待时间太长没获取到需要的数据的情况下取消该线程的任务。

24、Thread类的sleep()方法和对象的wait()方法都可以让线程暂停执行，它们有什么区别？

sleep()方法（休眠）是线程类（Thread）的静态方法，调用此方法会让当前线程暂停执行指定的时间，将执行机会（CPU）让给其他线程，但是对象的锁依然保持，因此休眠时间结束后会自动恢复。

wait()是Object类的方法，调用对象的wait()方法导致当前线程放弃对象的锁（线程暂停执行），进入对象的等待池（wait pool），只有调用对象的notify()方法（或notifyAll()方法）时才能唤醒等待池中的线程进入等锁池（lock pool），如果线程重新获得对象的锁就可以进入就绪状态。

25、线程的sleep()方法和yield()方法有什么区别？

* sleep()方法给其他线程运行机会时不考虑线程的优先级，因此会给低优先级的线程以运行的机会；yield()方法只会给相同优先级或更高优先级的线程以运行的机会；
* 线程执行sleep()方法后转入阻塞（blocked）状态，而执行yield()方法后转入就绪（ready）状态；
* sleep()方法声明抛出InterruptedException，而yield()方法没有声明任何异常；
* sleep()方法比yield()方法（跟操作系统CPU调度相关）具有更好的可移植性。

26、为什么wait,nofity和nofityAll这些方法不放在Thread类当中

一个很明显的原因是JAVA提供的锁是对象级的而不是线程级的，每个对象都有锁，通过线程获得。如果线程需要等待某些锁那么调用对象中的wait()方法就有意义了。如果wait()方法定义在Thread类中，线程正在等待的是哪个锁就不明显了。简单的说，由于wait，notify和notifyAll都是锁级别的操作，所以把他们定义在Object类中因为锁属于对象。

27、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。

28、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。

29、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。

30、一个线程如果出现了运行时异常怎么办？

如果这个异常没有被捕获的话，这个线程就停止执行了。另外重要的一点是：如果这个线程持有某个某个对象的监视器，那么这个对象监视器会被立即释放

31、Java当中有哪几种锁

自旋锁: 自旋锁在JDK1.6之后就默认开启了。基于之前的观察，共享数据的锁定状态只会持续很短的时间，为了这一小段时间而去挂起和恢复线程有点浪费，所以这里就做了一个处理，让后面请求锁的那个线程在稍等一会，但是不放弃处理器的执行时间，看看持有锁的线程能否快速释放。为了让线程等待，所以需要让线程执行一个忙循环也就是自旋操作。在jdk6之后，引入了自适应的自旋锁，也就是等待的时间不再固定了，而是由上一次在同一个锁上的自旋时间及锁的拥有者状态来决定

偏向锁: 在JDK1.之后引入的一项锁优化，目的是消除数据在无竞争情况下的同步原语。进一步提升程序的运行性能。 偏向锁就是偏心的偏，意思是这个锁会偏向第一个获得他的线程，如果接下来的执行过程中，改锁没有被其他线程获取，则持有偏向锁的线程将永远不需要再进行同步。偏向锁可以提高带有同步但无竞争的程序性能，也就是说他并不一定总是对程序运行有利，如果程序中大多数的锁都是被多个不同的线程访问，那偏向模式就是多余的，在具体问题具体分析的前提下，可以考虑是否使用偏向锁。

轻量级锁: 为了减少获得锁和释放锁所带来的性能消耗，引入了“偏向锁”和“轻量级锁”，所以在Java SE1.6里锁一共有四种状态，无锁状态，偏向锁状态，轻量级锁状态和重量级锁状态，它会随着竞争情况逐渐升级。锁可以升级但不能降级，意味着偏向锁升级成轻量级锁后不能降级成偏向锁

32、如何在两个线程间共享数据

通过在线程之间共享对象就可以了，然后通过wait/notify/notifyAll、await/signal/signalAll进行唤起和等待，比方说阻塞队列BlockingQueue就是为线程之间共享数据而设计的

33、如何正确的使用wait()?使用if还是while？

wait() 方法应该在循环调用，因为当线程获取到 CPU 开始执行的时候，其他条件可能还没有满足，所以在处理前，循环检测条件是否满足会更好。下面是一段标准的使用 wait 和 notify 方法的代码：

synchronized (obj) {

   while (condition does not hold)

     obj.wait(); // (Releases lock, and reacquires on wakeup)

     ... // Perform action appropriate to condition

}

34、什么是线程局部变量ThreadLocal

线程局部变量是局限于线程内部的变量，属于线程自身所有，不在多个线程间共享。Java提供ThreadLocal类来支持线程局部变量，是一种实现线程安全的方式。但是在管理环境下（如 web 服务器）使用线程局部变量的时候要特别小心，在这种情况下，工作线程的生命周期比任何应用变量的生命周期都要长。任何线程局部变量一旦在工作完成后没有释放，Java 应用就存在内存泄露的风险。

35、ThreadLoal的作用是什么？

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.

36、ThreadLocal 原理分析

ThreadLocal为解决多线程程序的并发问题提供了一种新的思路。ThreadLocal，顾名思义是线程的一个本地化对象，当工作于多线程中的对象使用ThreadLocal维护变量时，ThreadLocal为每个使用该变量的线程分配一个独立的变量副本，所以每一个线程都可以独立的改变自己的副本，而不影响其他线程所对应的副本。从线程的角度看，这个变量就像是线程的本地变量。

ThreadLocal类非常简单好用，只有四个方法，能用上的也就是下面三个方法：

* void set(T value)：设置当前线程的线程局部变量的值。
* T get()：获得当前线程所对应的线程局部变量的值。
* void remove()：删除当前线程中线程局部变量的值。

ThreadLocal是如何做到为每一个线程维护一份独立的变量副本的呢？在ThreadLocal类中有一个Map，键为线程对象，值是其线程对应的变量的副本，自己要模拟实现一个ThreadLocal类其实并不困难，代码如下所示：

import java.util.Collections;

import java.util.HashMap;

import java.util.Map;

public class MyThreadLocal<T> {

    private Map<Thread, T> map = Collections.synchronizedMap(new HashMap<Thread, T>());

    public void set(T newValue) {

        map.put(Thread.currentThread(), newValue);

    }

    public T get() {

        return map.get(Thread.currentThread());

    }

    public void remove() {

        map.remove(Thread.currentThread());

    }

}

37、如果你提交任务时，线程池队列已满，这时会发生什么

如果你使用的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；如果你使用的是有界队列比方说ArrayBlockingQueue的话，任务首先会被添加到ArrayBlockingQueue中，ArrayBlockingQueue满了，则会使用拒绝策略RejectedExecutionHandler处理满了的任务，默认是AbortPolicy。

38、为什么要使用线程池

避免频繁地创建和销毁线程，达到线程对象的重用。另外，使用线程池还可以根据项目灵活地控制并发的数目。

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、Thread.sleep(0)的作用是什么

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

41、什么是CAS

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

42、什么是乐观锁和悲观锁

乐观锁：乐观锁认为竞争不总是会发生，因此它不需要持有锁，将比较-替换这两个动作作为一个原子操作尝试去修改内存中的变量，如果失败则表示发生冲突，那么就应该有相应的重试逻辑。

悲观锁：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。

43、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？

44、ConcurrentHashMap的工作原理

ConcurrentHashMap在jdk 1.6和jdk 1.8实现原理是不同的.

jdk 1.6: ConcurrentHashMap是线程安全的，但是与Hashtablea相比，实现线程安全的方式不同。Hashtable是通过对hash表结构进行锁定，是阻塞式的，当一个线程占有这个锁时，其他线程必须阻塞等待其释放锁。ConcurrentHashMap是采用分离锁的方式，它并没有对整个hash表进行锁定，而是局部锁定，也就是说当一个线程占有这个局部锁时，不影响其他线程对hash表其他地方的访问。 具体实现:ConcurrentHashMap内部有一个Segment

jdk 1.8 在jdk 8中，ConcurrentHashMap不再使用Segment分离锁，而是采用一种乐观锁CAS算法来实现同步问题，但其底层还是“数组+链表->红黑树”的实现。

45、CyclicBarrier和CountDownLatch区别

这两个类非常类似，都在java.util.concurrent下，都可以用来表示代码运行到某个点上，二者的区别在于：

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

CyclicBarrier只能唤起一个任务，CountDownLatch可以唤起多个任务

CyclicBarrier可重用，CountDownLatch不可重用，计数值为0该CountDownLatch就不可再用了

46、java中的++操作符线程安全么？

不是线程安全的操作。它涉及到多个指令，如读取变量值，增加，然后存储回内存，这个过程可能会出现多个线程交差

47、有三个线程T1，T2，T3，怎么确保它们按顺序执行？

在多线程中有多种方法让线程按特定顺序执行，你可以用线程类的join()方法在一个线程中启动另一个线程，另外一个线程完成该线程继续执行。为了确保三个线程的顺序你应该先启动最后一个(T3调用T2，T2调用T1)，这样T1就会先完成而T3最后完成。

48、如何在Java中创建Immutable对象？

这个问题看起来和多线程没什么关系， 但不变性有助于简化已经很复杂的并发程序。Immutable对象可以在没有同步的情况下共享，降低了对该对象进行并发访问时的同步化开销。可是Java没有@Immutable这个注解符，要创建不可变类，要实现下面几个步骤：通过构造方法初始化所有成员、对变量不要提供setter方法、将所有的成员声明为私有的，这样就不允许直接访问这些成员、在getter方法中，不要直接返回对象本身，而是克隆对象，并返回对象的拷贝。

49、你有哪些多线程开发良好的实践？

* 给线程命名
* 最小化同步范围
* 优先使用volatile
* 尽可能使用更高层次的并发工具而非wait和notify()来实现线程通信,如BlockingQueue,Semeaphore
* 优先使用并发容器而非同步容器.
* 考虑使用线程池

50、可以创建Volatile数组吗？

Java 中可以创建 volatile类型数组，不过只是一个指向数组的引用，而不是整个数组。如果改变引用指向的数组，将会受到volatile 的保护，但是如果多个线程同时改变数组的元素，volatile标示符就不能起到之前的保护作用了

51、Volatile关键字的作用

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

* 多线程主要围绕可见性和原子性两个特性而展开，使用volatile关键字修饰的变量，保证了其在多线程之间的可见性，即每次读取到volatile变量，一定是最新的数据
* 代码底层执行不像我们看到的高级语言—-Java程序这么简单，它的执行是Java代码–>字节码–>根据字节码执行对应的C/C++代码–>C/C++代码被编译成汇编语言–>和硬件电路交互，现实中，为了获取更好的性能JVM可能会对指令进行重排序，多线程下可能会出现一些意想不到的问题。使用volatile则会对禁止语义重排序，当然这也一定程度上降低了代码执行效率

从实践角度而言，volatile的一个重要作用就是和CAS结合，保证了原子性，详细的可以参见java.util.concurrent.atomic包下的类，比如AtomicInteger。

52、volatile能使得一个非原子操作变成原子操作吗？

一个典型的例子是在类中有一个 long 类型的成员变量。如果你知道该成员变量会被多个线程访问，如计数器、价格等，你最好是将其设置为 volatile。为什么？因为 Java 中读取 long 类型变量不是原子的，需要分成两步，如果一个线程正在修改该 long 变量的值，另一个线程可能只能看到该值的一半（前 32 位）。但是对一个 volatile 型的 long 或 double 变量的读写是原子。

一种实践是用 volatile 修饰 long 和 double 变量，使其能按原子类型来读写。double 和 long 都是64位宽，因此对这两种类型的读是分为两部分的，第一次读取第一个 32 位，然后再读剩下的 32 位，这个过程不是原子的，但 Java 中 volatile 型的 long 或 double 变量的读写是原子的。volatile 修复符的另一个作用是提供内存屏障（memory barrier），例如在分布式框架中的应用。简单的说，就是当你写一个 volatile 变量之前，Java 内存模型会插入一个写屏障（write barrier），读一个 volatile 变量之前，会插入一个读屏障（read barrier）。意思就是说，在你写一个 volatile 域时，能保证任何线程都能看到你写的值，同时，在写之前，也能保证任何数值的更新对所有线程是可见的，因为内存屏障会将其他所有写的值更新到缓存。

53、volatile类型变量提供什么保证？

volatile 主要有两方面的作用:1.避免指令重排2.可见性保证.例如，JVM 或者 JIT为了获得更好的性能会对语句重排序，但是 volatile 类型变量即使在没有同步块的情况下赋值也不会与其他语句重排序。 volatile 提供 happens-before 的保证，确保一个线程的修改能对其他线程是可见的。某些情况下，volatile 还能提供原子性，如读 64 位数据类型，像 long 和 double 都不是原子的(低32位和高32位)，但 volatile 类型的 double 和 long 就是原子的.

54、Java 中，编写多线程程序的时候你会遵循哪些最佳实践？

这是我在写Java 并发程序的时候遵循的一些最佳实践：

* 给线程命名，这样可以帮助调试。
* 最小化同步的范围，而不是将整个方法同步，只对关键部分做同步。
* 如果可以，更偏向于使用 volatile 而不是 synchronized。
* 使用更高层次的并发工具，而不是使用 wait() 和 notify() 来实现线程间通信，如 BlockingQueue，CountDownLatch 及 Semeaphore。
* 优先使用并发集合，而不是对集合进行同步。并发集合提供更好的可扩展性。

55、说出至少 5 点在 Java 中使用线程的最佳实践。

这个问题与之前的问题类似，你可以使用上面的答案。对线程来说，你应该：

* 对线程命名
* 将线程和任务分离，使用线程池执行器来执行 Runnable 或 Callable。
* 使用线程池

56、Java中如何获取到线程dump文件

死循环、死锁、阻塞、页面打开慢等问题，打线程dump是最好的解决问题的途径。所谓线程dump也就是线程堆栈，获取到线程堆栈有两步：

* 获取到线程的pid，可以通过使用jps命令，在Linux环境下还可以使用ps -ef | grep java
* 打印线程堆栈，可以通过使用jstack pid命令，在Linux环境下还可以使用kill -3 pid

另外提一点，Thread类提供了一个getStackTrace()方法也可以用于获取线程堆栈。这是一个实例方法，因此此方法是和具体线程实例绑定的，每次获取获取到的是具体某个线程当前运行的堆栈。

57、高并发、任务执行时间短的业务怎样使用线程池？并发不高、任务执行时间长的业务怎样使用线程池？并发高、业务执行时间长的业务怎样使用线程池？

这是我在并发编程网上看到的一个问题，把这个问题放在最后一个，希望每个人都能看到并且思考一下，因为这个问题非常好、非常实际、非常专业。关于这个问题，个人看法是：

* 高并发、任务执行时间短的业务，线程池线程数可以设置为CPU核数+1，减少线程上下文的切换
* 并发不高、任务执行时间长的业务要区分开看：
* 假如是业务时间长集中在IO操作上，也就是IO密集型的任务，因为IO操作并不占用CPU，所以不要让所有的CPU闲下来，可以加大线程池中的线程数目，让CPU处理更多的业务
* 假如是业务时间长集中在计算操作上，也就是计算密集型任务，这个就没办法了，和（1）一样吧，线程池中的线程数设置得少一些，减少线程上下文的切换
* 并发高、业务执行时间长，解决这种类型任务的关键不在于线程池而在于整体架构的设计，看看这些业务里面某些数据是否能做缓存是第一步，增加服务器是第二步，至于线程池的设置，设置参考（2）。
* 业务执行时间长的问题，也可能需要分析一下，看看能不能使用中间件对任务进行拆分和解耦。

58、作业(进程)调度算法

* 先来先服务调度算法(FCFS) 每次调度都是从后备作业队列中选择一个或多个最先进入该队列的作业，将它们调入内存，为它们分配资源、创建进程，然后放入就绪队列。
* 短作业(进程)优先调度算法(SPF) 短作业优先(SJF)的调度算法是从后备队列中选择一个或若干个估计运行时间最短的作业，将它们调入内存运行。缺点:长作业的运行得不到保证
* 优先权调度算法(HPF) 当把该算法用于作业调度时，系统将从后备队列中选择若干个优先权最高的作业装入内存。当用于进程调度时，该算法是把处理机分配给就绪队列中优先权最高的进程，这时，又可进一步把该算法分成如下两种。 可以分为:
* 非抢占式优先权算法
* 抢占式优先权调度算法
* 高响应比优先调度算法(HRN) 每次选择高响应比最大的作业执行，响应比=(等待时间+要求服务时间)/要求服务时间。该算法同时考虑了短作业优先和先来先服务。
* 如果作业的等待时间相同，则要求服务的时间愈短，其优先权愈高，因而该算法有利于短作业。
* 当要求服务的时间相同时，作业的优先权决定于其等待时间，等待时间愈长，其优先权愈高，因而它实现的是先来先服务。
* 对于长作业，作业的优先级可以随等待时间的增加而提高，当其等待时间足够长时，其优先级便可升到很高，从而也可获得处理机。简言之，该算法既照顾了短作业，又考虑了作业到达的先后次序，不会使长作业长期得不到服务。因此，该算法实现了一种较好的折衷。当然，在利用该算法时，每要进行调度之前，都须先做响应比的计算，这会增加系统开销。
* 时间片轮转法（RR） 在早期的时间片轮转法中，系统将所有的就绪进程按先来先服务的原则排成一个队列，每次调度时，把CPU分配给队首进程，并令其执行一个时间片。时间片的大小从几ms到几百ms。当执行的时间片用完时，由一个计时器发出时钟中断请求，调度程序便据此信号来停止该进程的执行，并将它送往就绪队列的末尾；然后，再把处理机分配给就绪队列中新的队首进程，同时也让它执行一个时间片。这样就可以保证就绪队列中的所有进程在一给定的时间内均能获得一时间片的处理机执行时间。换言之，系统能在给定的时间内响应所有用户的请求。
* 多级反馈队列调度算法 它是目前被公认的一种较好的进程调度算法。
* 应设置多个就绪队列，并为各个队列赋予不同的优先级。第一个队列的优先级最高，第二个队列次之，其余各队列的优先权逐个降低。该算法赋予各个队列中进程执行时间片的大小也各不相同，在优先权愈高的队列中，为每个进程所规定的执行时间片就愈小。例如，第二个队列的时间片要比第一个队列的时间片长一倍，……，第i+1个队列的时间片要比第i个队列的时间片长一倍。
* 当一个新进程进入内存后，首先将它放入第一队列的末尾，按FCFS原则排队等待调度。当轮到该进程执行时，如它能在该时间片内完成，便可准备撤离系统；如果它在一个时间片结束时尚未完成，调度程序便将该进程转入第二队列的末尾，再同样地按FCFS原则等待调度执行；如果它在第二队列中运行一个时间片后仍未完成，再依次将它放入第三队列，……，如此下去，当一个长作业(进程)从第一队列依次降到第n队列后，在第n 队列便采取按时间片轮转的方式运行。
* 仅当第一队列空闲时，调度程序才调度第二队列中的进程运行；仅当第1～(i-1)队列均空时，才会调度第i队列中的进程运行。如果处理机正在第i队列中为某进程服务时，又有新进程进入优先权较高的队列(第1～(i-1)中的任何一个队列)，则此时新进程将抢占正在运行进程的处理机，即由调度程序把正在运行的进程放回到第i队列的末尾，把处理机分配给新到的高优先权进程。

59、讲讲线程池的实现原理

首先要明确为什么要使用线程池，使用线程池会带来什么好处？

* 线程是稀缺资源，不能频繁的创建。
* 应当将其放入一个池子中，可以给其他任务进行复用。
* 解耦作用，线程的创建于执行完全分开，方便维护。

60、创建一个线程池

以一个使用较多的

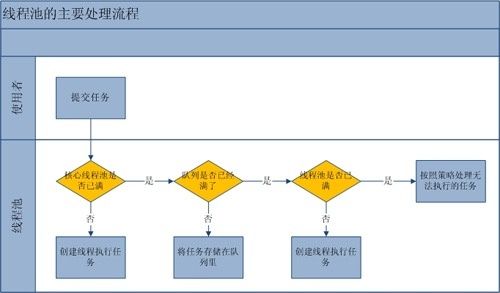
ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue, RejectedExecutionHandler handler);

为例：

* 其中的 corePoolSize 为线程池的基本大小。
* maximumPoolSize 为线程池最大线程大小。
* keepAliveTime 和 unit 则是线程空闲后的存活时间。
* workQueue 用于存放任务的阻塞队列。
* handler 当队列和最大线程池都满了之后的饱和策略。

61、处理流程

当提交一个任务到线程池时它的执行流程是怎样的呢？



首先第一步会判断核心线程数有没有达到上限，如果没有则创建线程(会获取全局锁)，满了则会将任务丢进阻塞队列。

如果队列也满了则需要判断最大线程数是否达到上限，如果没有则创建线程(获取全局锁)，如果最大线程数也满了则会根据饱和策略处理。

常用的饱和策略有:

* 直接丢弃任务。
* 调用者线程处理。
* 丢弃队列中的最近任务，执行当前任务。

所以当线程池完成预热之后都是将任务放入队列，接着由工作线程一个个从队列里取出执行。

62、合理配置线程池

线程池并不是配置越大越好，而是要根据任务的熟悉来进行划分： 如果是 CPU 密集型任务应当分配较少的线程，比如 CPU 个数相当的大小。

如果是 IO 密集型任务，由于线程并不是一直在运行，所以可以尽可能的多配置线程，比如 CPU 个数 \* 2 。

当是一个混合型任务，可以将其拆分为 CPU 密集型任务以及 IO 密集型任务，这样来分别配置。

63、synchronize 实现原理

众所周知 Synchronize 关键字是解决并发问题常用解决方案，有以下三种使用方式:

* 同步普通方法，锁的是当前对象。
* 同步静态方法，锁的是当前 Class 对象。
* 同步块，锁的是 {} 中的对象。

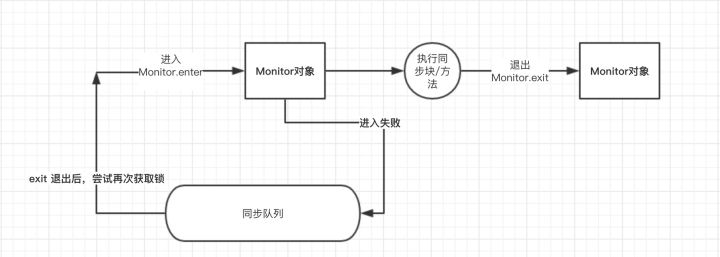
实现原理： JVM 是通过进入、退出对象监视器( Monitor )来实现对方法、同步块的同步的。

具体实现是在编译之后在同步方法调用前加入一个 monitor.enter 指令，在退出方法和异常处插入 monitor.exit 的指令。

其本质就是对一个对象监视器( Monitor )进行获取，而这个获取过程具有排他性从而达到了同一时刻只能一个线程访问的目的。

而对于没有获取到锁的线程将会阻塞到方法入口处，直到获取锁的线程 monitor.exit 之后才能尝试继续获取锁。

流程图如下:



synchronize 很多都称之为重量锁，JDK1.6 中对 synchronize 进行了各种优化，为了能减少获取和释放锁带来的消耗引入了偏向锁和轻量锁。

64、轻量锁

当代码进入同步块时，如果同步对象为无锁状态时，当前线程会在栈帧中创建一个锁记录(Lock Record)区域，同时将锁对象的对象头中 Mark Word 拷贝到锁记录中，再尝试使用 CAS 将 Mark Word 更新为指向锁记录的指针。

如果更新成功，当前线程就获得了锁。

如果更新失败 JVM 会先检查锁对象的 Mark Word 是否指向当前线程的锁记录。

如果是则说明当前线程拥有锁对象的锁，可以直接进入同步块。

不是则说明有其他线程抢占了锁，如果存在多个线程同时竞争一把锁，轻量锁就会膨胀为重量锁。

65、解锁

轻量锁的解锁过程也是利用 CAS 来实现的，会尝试锁记录替换回锁对象的 Mark Word 。如果替换成功则说明整个同步操作完成，失败则说明有其他线程尝试获取锁，这时就会唤醒被挂起的线程(此时已经膨胀为重量锁)

轻量锁能提升性能的原因是：

认为大多数锁在整个同步周期都不存在竞争，所以使用 CAS 比使用互斥开销更少。但如果锁竞争激烈，轻量锁就不但有互斥的开销，还有 CAS 的开销，甚至比重量锁更慢。

66、偏向锁

为了进一步的降低获取锁的代价，JDK1.6 之后还引入了偏向锁。

偏向锁的特征是:锁不存在多线程竞争，并且应由一个线程多次获得锁。

当线程访问同步块时，会使用 CAS 将线程 ID 更新到锁对象的 Mark Word 中，如果更新成功则获得偏向锁，并且之后每次进入这个对象锁相关的同步块时都不需要再次获取锁了。

67、释放锁

当有另外一个线程获取这个锁时，持有偏向锁的线程就会释放锁，释放时会等待全局安全点(这一时刻没有字节码运行)，接着会暂停拥有偏向锁的线程，根据锁对象目前是否被锁来判定将对象头中的 Mark Word 设置为无锁或者是轻量锁状态。

偏向锁可以提高带有同步却没有竞争的程序性能，但如果程序中大多数锁都存在竞争时，那偏向锁就起不到太大作用。可以使用 -XX:-userBiasedLocking=false 来关闭偏向锁，并默认进入轻量锁。

**volatile**

**对 volatile的理解**

volatile 是一种轻量级的同步机制。

1. 保证数据可见性
2. 不保证原子性
3. 禁止指令重排序

**JMM**

JMM（Java 内存模型）是一种抽象的概念，描述了一组规则或规范，定义了程序中各个变量的访问方式。

JVM运行程序的实体是线程，每个线程创建时 JVM 都会为其创建一个工作内存，是线程的私有数据区域。JMM中规定所有变量都存储在主内存，主内存是共享内存。线程对变量的操作在工作内存中进行，首先将变量从主内存拷贝到工作内存，操作完成后写会主内存。不同线程间无法访问对方的工作内存，线程通信（传值）通过主内存来完成。

JMM 对于同步的规定：

1. 线程解锁前，必须把共享变量的值刷新回主内存
2. 线程加锁前，必须读取主内存的最新值到自己的工作内存
3. 加锁解锁是同一把锁

JMM 的三大特性

1. 可见性
2. 原子性
3. 顺序性

原子性是不可分割，某个线程正在做某个具体业务时，中间不可以被分割，要么全部成功，要么全部失败。

重排序：计算机在执行程序时，为了提高性能，编译器和处理器常常对指令做重排序，源代码经过编译器优化重排序、指令并行重排序、内存系统的重排序之后得到最终执行的指令。

在单线程中保证程序最终执行结果和代码执行顺序执行结果一致。

多线程中线程交替执行，由于重排序，两个线程中使用的变量能否保证一致性无法确定，结果无法确定。

处理器在处理重排序时需要考虑数据的依赖性。

volatile 实现禁止指令重排序，避免多线程环境下程序乱序执行。是通过内存屏障指令来执行的，通过插入内存屏障禁止在内存屏障后的指令执行重排序优化，并强制刷出缓存数据，保证线程能读取到这些数据的最新版本。

**实例1：volatile 保证可见性**

[?](https://www.jb51.net/article/160125.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | class MyData {   //volatile int number = 0;//case2   //int number=0; //case1   public void change() {   number = 60;   }  }    public class VolatileDemo {   public static void main(String[] args) {   MyData data=new MyData();     new Thread(()->{   System.out.println(Thread.currentThread().getName()+"\t come in");   try{ TimeUnit.SECONDS.sleep(3); } catch (InterruptedException e) {e.printStackTrace();}   data.change();   System.out.println(Thread.currentThread().getName()+"\t updated number value:"+data.number);   },"A").start();     while(data.number==0){}   System.out.println(Thread.currentThread().getName()+"\t over, get number:"+data.number);     }  } |

当我们使用case1的时候，也就是number没有volatile修饰的时候，运行结果：

A  come in  
A  updated number value:60

并且程序没有执行结束，说明在main线程中由于不能保证可见性，一直在死循环。

当执行case2的时候：

A  come in  
A  updated number value:60  
main  over, get number:60

保证了可见性，因此main成功结束。

**实例2： volatile 不保证原子性**

[?](https://www.jb51.net/article/160125.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | class MyData {   volatile int number = 0;     public void change() {   number = 60;   }     public void addOne() {   number++;   }  }    public class VolatileDemo {   public static void main(String[] args) {   case2();   }     //验证原子性   public static void case2() {   MyData myData = new MyData();     for (int i = 0; i < 20; i++) {   new Thread(() -> {   for (int j = 0; j < 1000; j++) {    myData.addOne();   }   }, String.valueOf(i)).start();   }     while(Thread.activeCount()>2){   Thread.yield();   }   System.out.println(Thread.currentThread().getName()+"\t number value:"+myData.number);   }  } |

最终输出结果可以发现并不是 20000，且多次输出结果并不一致，因此说明 volatile 不能保证原子性。

如何保证原子性

1. 加锁：使用 synchronized 加锁
2. 使用 AtomicInteger

**实例3：volatile 和 单例模式**

DCL模式的单例模式

[?](https://www.jb51.net/article/160125.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | public class Singleton {     private static Singleton instance=null;   private Singleton(){   System.out.println(Thread.currentThread().getName()+" constructor");   }     //DCL 双端检锁机制 |

DCL 机制不能完全保证线程安全，因为有指令重排序的存在。

原因在于instance = new Singleton(); 可以分为三步：

1. memory=allocate();//分配内存空间

2. instance(memory);//初始化对象

3. instance=memory;//设置instance指向分配的内存地址，分配成功后，instance!=null

由于步骤2和步骤3不存在数据依赖关系，且无论重排序与否执行结果在单线程中没有改变，因此这两个步骤的重排序是允许的。也就是说指令重排序只会保证单线程串行语义的一致性（as-if-serial），但是不会关心多线程间的语义一致性。

因此，重排序之后，先执行3会导致instance!=null，但是对象还未被初始化。此时，别的线程在调用时，获取了一个未初始化的对象。

因此，在声明 instance 时，使用 volatile 进行修饰，禁止指令重排序。

[?](https://www.jb51.net/article/160125.htm)

|  |  |
| --- | --- |
| 1 | private static volatile Singleton instance = null; |

**CAS**

CAS 的全程是 CompareAndSwap，是一条 CPU 并发原语。它的功能是判断内存某个位置的值是否为预期值，如果是则更新为新的值，这个过程是原子的。

CAS 的作用是比较当前工作内存中的值和主内存中的值，如果相同则执行操作，否则继续比较直到主内存和工作内存中的值一致为止。主内存值为V，工作内存中的预期值为A，要修改的更新值为B，当且仅当A和V相同，将V修改为B，否则什么都不做。

**CAS 底层原理：**

在原子类中，CAS 操作都是通过 Unsafe 类来完成的。

[?](https://www.jb51.net/article/160125.htm)

|  |  |
| --- | --- |
| 1  2  3  4 | //AtomicInteger i++  public final int getAndIncrement(){   return unsafe.getAndAddInt(this,valueoffset,1);  } |

其中 this 是当前对象， valueoffset 是一个 long ，代表地址的偏移量。

[?](https://www.jb51.net/article/160125.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | //AtomicInteger.java  private static final Unsafe unsfae=Unsafe.getUnsafe();//unsafe对象  private static final long valueOffset;//地址偏移量    static{   try{   valueoffset=unsafe.objectFieldOffset(AtomicInteger.class.getDeclaredField("value");   }catch(Excepthion ex){throw new Error(ex);}  }    private volatile int value;//存储的数值 |

1. Unsafe

Unsafe 类是 rt.jar 下的 sun.misc 包下的一个类，基于该类可以直接操作特定内存的数据。  
Java方法无法直接访问底层系统，需要使用 native 方法访问，Unsafe 类的内部方法都是 native 方法，其中的方法可以像C的指针一样直接操作内存，Java 中的 CAS 操作的执行都依赖于 Unsafe 类的方法。

1. valueOffset

该变量表示变量值在内存中的偏移地址， Unsafe 就是根据内存偏移地址获取数据的。

**Unsafe类**

CAS 并发源于体现在 Java 中就是 Unsafe 类的各个方法。调用该类中的 CAS 方法，JVM会帮我们实现出 CAS 汇编指令，这是一种完全依赖于硬件的功能。

原语是由若干条指令组成的，用于完成某个功能的过程。原语的执行必须是连续的，执行过程不允许被中断。所以 CAS 是一条 CPU 的原子指令，不会造成数据不一致问题。

下边是 AtomicInteger 中实现 i++ 功能所调用的 Unsafe 类的函数。

[?](https://www.jb51.net/article/160125.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | //unsafe.getAndAddInt  public final int getAndAddInt(Object var1,long var2,int var4){   int var5;   do{   //获取当前的值的地址   var5=this.getIntVolatile(var1,var2);   //var1代表对象，var2和var5分别代表当前对象的真实值和期望值，如果二者相等，更新为var5+var4   }while(!this.compareAndSwapInt(var1,var2,var5,var5+var4);   return var5;  } |

在 getAndAddInt 函数中，var1 代表了 AtomicInteger 对象， var2 代表了该对象在内存中的地址， var4 代表了期望增加的数值。

首先通过 var1 和 var2 获取到当前的主内存中真实的 int 值，也就是 var5。

然后通过循环来进行数据更改，当比较到真实值和对象的当前值相等，则更新，退出循环；否则再次获取当前的真实值，继续尝试，直到成功。

在 CAS 中通过自旋而不是加锁来保证一致性，同时和加锁相比，提高了并发性。

具体情境来说：线程A和线程B并发执行 AtomicInteger 的自增操作：

1. AtomicInteger 中的 value 原始值为 3。主内存中 value 为 3， 线程A和线程B的工作内存中有 value 为 3 的副本；
2. 线程 A 通过 getIntVolatile() 获取到 value 的值为3，并被挂起。
3. 线程 B 也获取到 value 的值为3，然后执行 compareAndSwapInt 方法，比较到内存真实值也是 3，因此成功修改内存值为4.
4. 此时线程 A 继续执行比较，发现对象中的 value 3 和主内存中的 value 4 不一致，说明已经被修改，A 重新进入循环。
5. 线程 A 重新获取 value，由于 value 被 volatile 修饰，所以线程 A 此时 value 为4，和主内存中 value 相等，修改成功。

**CAS的缺点**

1. 如果CAS失败，会一直尝试。如果CAS长时间不成功，会给CPU带来很大的开销。
2. CAS 只能用来保证单个共享变量的原子操作，对于多个共享变量操作，CAS无法保证，需要使用锁。
3. 存在 ABA 问题。

ABA问题

CAS 实现一个重要前提需要取出内存中某个时刻的数据并在当下时刻比较并替换，这个时间差会导致数据的变化。

线程1从内存位置V中取出A，线程2也从V中取出A，然后线程2通过一些操作将A变成B，然后又把V位置的数据变成A，此时线程1进行CAS操作发现V中仍然是A，操作成功。尽管线程1的CAS操作成功，但是不代表这个过程没有问题。

这个问题类似于幻读问题，通过新增版本号的机制来解决。在这里可以使用 AtomicStampedReference 来解决。

AtomicStampedReference

通过 AtomicStampedReference 来解决这个问题。

[?](https://www.jb51.net/article/160125.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | public class SolveABADemo {   static AtomicStampedReference<Integer> atomicStampedReference=new AtomicStampedReference<>(100,1);     new Thread(()->{   int stamp=atomicStampedReference.getStamp();   System.out.println(Thread.currentThread().getName()+"\t 版本号："+stamp);   try {   Thread.sleep(1000);   } catch (InterruptedException e) {   e.printStackTrace();   }   atomicStampedReference.compareAndSet(100,101,atomicStampedReference.getStamp(),atomicStampedReference.getStamp()+1);   System.out.println(Thread.currentThread().getName()+"\t 版本号："+atomicStampedReference.getStamp());   atomicStampedReference.compareAndSet(101,100,atomicStampedReference.getStamp(),atomicStampedReference.getStamp()+1);   System.out.println(Thread.currentThread().getName()+"\t 版本号："+atomicStampedReference.getStamp());   },"t1").start();     new Thread(()->{   int stamp=atomicStampedReference.getStamp();   System.out.println(Thread.currentThread().getName()+"\t 版本号："+stamp);   try {   Thread.sleep(3000);   } catch (InterruptedException e) {   e.printStackTrace();   }   boolean ret=atomicStampedReference.compareAndSet(100,2019,stamp,stamp+1);   System.out.println(Thread.currentThread().getName()+"\t"+ret   +" stamp:"+atomicStampedReference.getStamp()   +" value:"+atomicStampedReference.getReference());   },"t2").start();   }  } |

t1  版本号：1  
t2  版本号：1  
t1  版本号：2  
t1  版本号：3  
t2 false stamp:3 value:100

**集合类的线程安全问题**

**ConcurrentModificationException**

这个异常也就是并发修改异常，java.util.ConcurrentModificationException。

导致这个异常的原因，是集合类本身是线程不安全的。

解决方案：

1. 使用 Vector， Hashtable 等同步容器
2. 使用 Collections.synchronizedxxx(new XX) 创建线程安全的容器
3. 使用 CopyOnWriteList, CopyOnWriteArraySet, ConcurrentHashMap 等 j.u.c 包下的并发容器。

**CopyOnWriteArrayList**

底层使用了private transient volatile Object[] array;

CopyOnWriteArrayList 采用了写时复制、读写分离的思想。

[?](https://www.jb51.net/article/160125.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | public boolean add(E e){   final ReentrantLock lock=this.lock;   try{   //旧数组   Object[] elements = getArray();   int len = elements.length;   //复制新数组   Object[] newElements = Arrays.copyOf(elements, len+1);   //修改新数组   newElements[len] = e;   //更改旧数组引用指向新数组   setArray(newElements);   return true;   }finally{   lock.unlock();   }  } |

添加元素时，不是直接添加到当前容器数组，而是复制到新的容器数组，向新的数组中添加元素，添加完之后将原容器引用指向新的容器。

这样做的好处是可以对该容器进行并发的读，而不需要加锁，因为读时容器不会添加任何元素。

CopyOnWriteArraySet 本身就是使用 CopyOnWriteArrayList 来实现的。

**Java锁**

**公平锁和非公平锁**

ReentrantLock 可以指定构造函数的 boolean 类型得到公平或非公平锁，默认是非公平锁，synchronized也是非公平锁。  
公平锁是多个线程按照申请锁的顺序获取锁，是 FIFO 的。并发环境中，每个线程在获取锁时先查看锁维护的等待队列，为空则战友，否则加入队列。

非公平锁是指多个线程不是按照申请锁的顺序，有可能后申请的线程比先申请的线程优先获取锁。高并发情况下可能导致优先级反转或者饥饿现象。并发环境中，上来尝试占有锁，尝试失败，再加入等待队列。

**可重入锁（递归锁）**

可冲入锁指的是同一线程外层函数获取锁之后，内层递归函数自动获取锁。也就是线程能进入任何一个它已经拥有的锁所同步着的代码块。

ReentrantLock 和 synchronized 都是可重入锁。

可重入锁最大的作用用来避免死锁。

**自旋锁**

自旋锁是指尝试获取锁的线程不会立即阻塞，而是采用循环的方式尝试获取锁。好处是减少线程上下文切换的消耗，缺点是循环时会消耗CPU资源。

实现自旋锁：

[?](https://www.jb51.net/article/160125.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47 | public class SpinLockDemo {  //使用AtomicReference<Thread>来更新当前占用的 Thread   AtomicReference<Thread> threadAtomicReference=new AtomicReference<>();     public static void main(String[] args) {   SpinLockDemo demo=new SpinLockDemo();   new Thread(()->{   demo.myLock();   try {   Thread.sleep(3000);   } catch (InterruptedException e) {   e.printStackTrace();   }   demo.myUnlock();   },"t1").start();       try {   Thread.sleep(1000);   } catch (InterruptedException e) {   e.printStackTrace();   }     new Thread(()->{   demo.myLock();   demo.myUnlock();   },"t2").start();     }     public void myLock(){   Thread thread=Thread.currentThread();   System.out.println(Thread.currentThread().getName()+"\t come in");     //如果当前占用的线程为null，则尝试获取更新   while(!threadAtomicReference.compareAndSet(null,thread)){     }   }     public void myUnlock(){   Thread thread=Thread.currentThread();   //释放锁，将占用的线程设置为null   threadAtomicReference.compareAndSet(thread,null);   System.out.println(Thread.currentThread().getName()+"\t unlocked");   }  } |

**读写锁**

独占锁：该锁一次只能被一个线程持有，如 ReentrantLock 和 synchronized。

共享锁：该锁可以被多个线程持有。

ReentrantReadWriteLock 中，读锁是共享锁，写锁时独占锁。读读共享保证并发性，读写互斥。

**并发工具类**

CountDownLatch

CountDownLatch 的作用是让一些线程阻塞直到另外一些线程完成一系列操作后才被唤醒。

CountDownLatch 在初始时设置一个数值，当一个或者多个线程使用 await() 方法时，这些线程会被阻塞。其余线程调用 countDown() 方法，将计数器减去1，当计数器为0时，调用 await() 方法被阻塞的线程会被唤醒，继续执行。

可以理解为，等大家都走了，保安锁门。

CyclicBarrier

CyclicBarrier 是指可以循环使用的屏障，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，屏障才会开门，被屏障拦截的线程才会继续工作，线程进入屏障通过 await() 方法。

可以理解为，大家都到齐了，才能开会。

Semaphore

信号量用于：

1. 多个共享资源的互斥使用
2. 并发线程数的控制

可以理解为，多个车抢停车场的多个车位。当进入车位时，调用 acquire() 方法占用资源。当离开时，调用 release() 方法释放资源。

**阻塞队列**

阻塞队列首先是一个队列，所起的作用如下：

1. 当阻塞队列为空，从队列中获取元素的操作将会被阻塞
2. 当阻塞队列为满，向队列中添加元素的操作将会被阻塞

试图从空的阻塞队列中获取元素的线程将会被阻塞，直到其他线程向空的队列中插入新的元素。同样的，试图向已满的阻塞队列中添加新元素的线程同样会被阻塞，直到其他线程从队列中移除元素使得队列重新变得空闲起来并后序新增。

阻塞：阻塞是指在某些情况下会挂起线程，即阻塞，一旦条件满足，被挂起的线程又会自动被唤醒。

优点：BlockingQueue 能帮助我们进行线程的阻塞和唤醒，而无需关心何时需要阻塞线程，何时需要唤醒线程。同时兼顾了效率和线程安全。

阻塞队列的架构

BlokcingQueue 接口实现了 Queue 接口，该接口有如下的实现类：

1. ArrayBlockingQueue: 由数组组成的有界阻塞队列
2. LinkedBlockingQueue： 由链表组成的有界阻塞队列（默认大小为 Integer.MAX\_VALUE）
3. PriorityBlockingQueue：支持优先级排序的无界阻塞队列
4. DelayQueue：使用优先级队列实现的延迟无界阻塞队列
5. SynchronousQueue： 不存储元素的阻塞队列，单个元素的队列，同步提交队列
6. LinkedTransferQueue：链表组成的无界阻塞队列
7. LinkedBlockingDeque：链表组成的双向阻塞队列

阻塞队列的方法

| **方法类型** | **抛出异常** | **特殊值** | **阻塞** | **超时** |
| --- | --- | --- | --- | --- |
| 插入 | add(e) | offer(e) | put(e) | offer(e,time,unit) |
| 移除 | remove() | poll() | take() | poll(time,unit) |
| 检查 | element() | peek() | 无 | 无 |

1. 抛出异常：当队列满，add(e)会抛出异常IllegalStateException: Queue full；当队列空，remove()和element()会抛出异常NoSuchElementException
2. 特殊值：offer(e)会返回 true/false。peek()会返回队列元素或者null。
3. 阻塞：队列满，put(e)会阻塞直到成功或中断；队列空take()会阻塞直到成功。
4. 超时：阻塞直到超时后退出，返回值和特殊值中的情况一样。

**生产者消费者模式**

方式1. 使用Lock

[?](https://www.jb51.net/article/160125.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75 | class ShareData {   private int number = 0;   private Lock lock = new ReentrantLock();   private Condition condition = lock.newCondition();     public void increment() throws Exception {   lock.lock();   try {   //判断   while (number != 0) {   condition.await();   }   //干活   number++;   System.out.println(Thread.currentThread().getName() + " produce\t" + number);   //通知唤醒   condition.signalAll();   } catch (Exception e) {   e.printStackTrace();   } finally {   lock.unlock();   }   }     public void decrement()throws Exception{   lock.lock();   try {   //判断   while (number == 0) {   condition.await();   }   //干活   number--;   System.out.println(Thread.currentThread().getName() + " consume\t" + number);   //通知唤醒   condition.signalAll();   } catch (Exception e) {   e.printStackTrace();   } finally {   lock.unlock();   }   }  }    /\*\*   \* 一个初始值为0的变量，两个线程交替操作，一个加1一个减1,重复5次   \* 1. 线程 操作 资源类   \* 2. 判断 干活 通知   \* 3. 防止虚假唤醒机制:判断的时候要用while而不是用if   \*/  public class ProduceConsumeTraditionalDemo {   public static void main(String[] args) {   ShareData data=new ShareData();     new Thread(()->{   for (int i = 0; i < 5 ; i++) {   try {    data.increment();   } catch (Exception e) {    e.printStackTrace();   }   }   },"A").start();     new Thread(()->{   for (int i = 0; i < 5 ; i++) {   try {    data.decrement();   } catch (Exception e) {    e.printStackTrace();   }   }   },"B").start();   }  } |

打印结果

A produce 1  
B consume 0  
A produce 1  
B consume 0  
A produce 1  
B consume 0  
A produce 1  
B consume 0  
A produce 1  
B consume 0

方法2：使用阻塞队列

[?](https://www.jb51.net/article/160125.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74 | public class ProduceConsumeBlockingQueueDemo {   public static void main(String[] args) {   SharedData data=new SharedData(new ArrayBlockingQueue<>(10));   new Thread(()-> {    System.out.println(Thread.currentThread().getName() + "\t生产线程启动");    try {    data.produce();    } catch (InterruptedException e) {    e.printStackTrace();    }   },"Producer").start();   new Thread(()-> {    System.out.println(Thread.currentThread().getName() + "\t消费线程启动");    try {    data.consume();    } catch (InterruptedException e) {    e.printStackTrace();    }   },"Consumer").start();     try {    Thread.sleep(3000);   } catch (InterruptedException e) {    e.printStackTrace();   }   data.stop();   System.out.println("停止");   }  }    class SharedData{   private volatile boolean FLAG=true;   private AtomicInteger atomicInteger=new AtomicInteger();     BlockingQueue<String> blockingQueue=null;     public SharedData(BlockingQueue<String> blockingQueue) {   this.blockingQueue = blockingQueue;   System.out.println(blockingQueue.getClass().getName());   }     public void produce() throws InterruptedException {   String data=null;   boolean ret;   while(FLAG){    data=""+atomicInteger.incrementAndGet();    ret=blockingQueue.offer(data,2L,TimeUnit.SECONDS);    if(ret){    System.out.println(Thread.currentThread().getName()+"\t插入"+data+"成功");    }else{    System.out.println(Thread.currentThread().getName()+"\t插入"+data+"失败");    }    TimeUnit.SECONDS.sleep(1);   }   System.out.println("生产结束，FLAG=false");   }     public void consume() throws InterruptedException {   String ret=null;   while(FLAG){    ret=blockingQueue.poll(2L,TimeUnit.SECONDS);    if(null==ret||ret.equalsIgnoreCase("")){    System.out.println(FLAG=false);    System.out.println(Thread.currentThread().getName()+"\t消费等待超时退出");    return;    }    System.out.println(Thread.currentThread().getName() + "\t消费" + ret + "成功");   }   }     public void stop(){   FLAG=false;   }  } |

使用阻塞队列+原子类+volatile变量的方式。

打印结果如下：

java.util.concurrent.ArrayBlockingQueue  
Producer 生产线程启动  
Consumer 消费线程启动  
Producer 插入1成功  
Consumer 消费1成功  
Producer 插入2成功  
Consumer 消费2成功  
Producer 插入3成功  
Consumer 消费3成功  
停止  
生产结束，FLAG=false  
false  
Consumer 消费等待超时退出

**Synchronized 和 Lock 的区别**

1. 原始构成
   1. Synchronized 是关键字，属于JVM层面，底层是通过 monitorenter 和 monitorexit 完成，依赖于 monitor 对象来完成。由于 wait/notify 方法也依赖于 monitor 对象，因此只有在同步块或方法中才能调用这些方法。
   2. Lock 是 java.util.concurrent.locks.lock 包下的，是 api层面的锁。
2. 使用方法
   1. Synchronized 不需要用户手动释放锁，代码完成之后系统自动让线程释放锁
   2. ReentrantLock 需要用户手动释放锁，没有手动释放可能导致死锁。
3. 等待是否可以中断
   1. Synchronized 不可中断，除非抛出异常或者正常运行完成
   2. ReentrantLock 可以中断。一种是通过 tryLock(long timeout, TimeUnit unit)，另一种是lockInterruptibly()放代码块中，调用interrupt()方法进行中断。
4. 加锁是否公平
   1. synchronized 是非公平锁
   2. ReentrantLock 默认非公平锁，可以在构造方法传入 boolean 值，true 代表公平锁，false 代表非公平锁。
5. 锁绑定多个 Condition
   1. Synchronized 只有一个阻塞队列，只能随机唤醒一个线程或者唤醒全部线程。
   2. ReentrantLock 用来实现分组唤醒，可以精确唤醒。

案例：三个线程循环打印

[?](https://www.jb51.net/article/160125.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71 | class ShareData{   private int number=1;   private Lock lock=new ReentrantLock();     public void printA(){   lock.lock();   Condition conditionA=lock.newCondition();   try{    while(number!=1){    conditionA.await();    }    for (int i = 0; i < 5; i++) {    System.out.println(Thread.currentThread().getName()+"\t"+i);    }    number=2;    conditionA.signal();   }catch (Exception e){    e.printStackTrace();   } finally {    lock.unlock();   }   }     public void printB(){   lock.lock();   Condition conditionB=lock.newCondition();   try{    while(number!=2){    conditionB.await();    }    for (int i = 0; i < 10; i++) {    System.out.println(Thread.currentThread().getName()+"\t"+i);    }    number=3;    conditionB.signal();   }catch (Exception e){    e.printStackTrace();   } finally {    lock.unlock();   }   }     public void printC(){   lock.lock();   Condition conditionC=lock.newCondition();   try{    //判断    while(number!=3){    conditionC.await();    }    //干活    for (int i = 0; i < 15; i++) {    System.out.println(Thread.currentThread().getName()+"\t"+i);    }    number=1;    //通知    conditionC.signal();   }catch (Exception e){    e.printStackTrace();   } finally {    lock.unlock();   }   }     public static void main(String[] args) {   ShareData data=new ShareData();   new Thread(() -> data.printA(),"A").start();   new Thread(() -> data.printB(),"B").start();   new Thread(() -> data.printC(),"C").start();   }  } |

**线程池**

**创建线程**

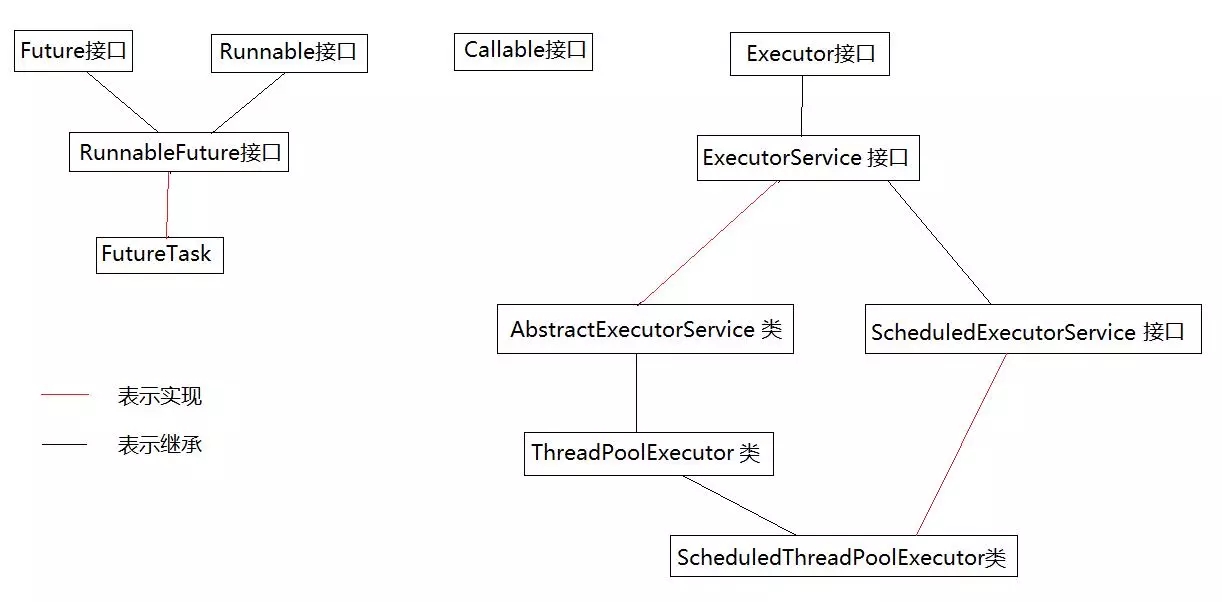
1. 实现 Runnable 接口
2. 实现 Callable 接口
3. 继承 Thread 类
4. 使用线程池

Thread的构造函数中并没有传入 Callable 的方式，但是可以传入 Runnable 接口:  
Thread thread=new Thread(Runnable runnable, String name);。为了使用 Callable 接口，我们需要使用到 FutureTask 类。 FutureTask 类实现了 RunnableFuture 这一接口，而 RunnableFutre 又是 Future 的子接口，因此 FutureTask 可以作为参数使用上述的 Thread 构造函数。同时， FutureTask 本身构造函数可以传入 Callable 。

[?](https://www.jb51.net/article/160125.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | class MyThread implements Callable<Integer>{   @Override   public Integer call() {   System.out.println("come in callable");   return 2019;   }  }  class Main{   public static void main(String [] args){   FutureTask<Integer> futureTask = new FutureTask<>(new MyThread2());   Thread t1=new Thread(futureTask,"A");   }  } |

**线程池架构**

[](https://files.jb51.net/file_images/article/201904/201942485952876.jpg?2019324908)

除此之外，还有 Executors 工具类。

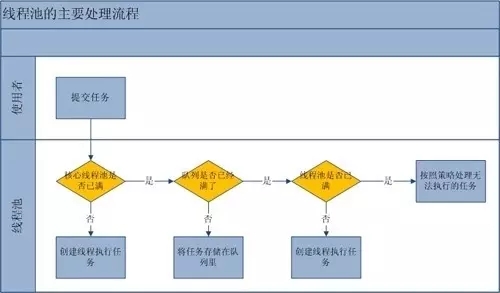
**ThreadPoolExecutor**

线程池有七大参数：

[?](https://www.jb51.net/article/160125.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | public ThreadPoolExecutor(   int corePoolSize,//线程池常驻核心线程数   int maximumPoolSize,//线程池能容纳同时执行最大线程数   long keepAliveTime,//多余的空闲线程的存活时间，当前线程池线程数量超过core，空闲时间达到keepAliveTime，多余空闲线程会被销毁直到只剩下core个   TimeUnit unit,   BlockingQueue<Runnable> workQueue,//被提交尚未被执行的任务队列   ThreadFactory threadFactory,//创建线程的线程工厂   RejectedExecutionHandler handler//拒绝策略   )  {...} |

处理流程如下：



1. 创建线程池，等待提交过来的任务请求。
2. 添加请求任务
   1. 如果运行线程数小于 corePoolSize，创建线程运行该任务
   2. 如果运行线程数大于等于 corePoolSize，将任务放入队列
   3. 队列满，且运行线程数量小于 maximumPoolSize，创建非核心线程运行任务
   4. 队列满，且运行线程数量大于等于 maximumPoolSize，线程池会启动饱和拒绝策略执行。
3. 线程完成任务，会从队列中取下一个任务来执行
4. 一个线程无事可做超过 keepAliveTime 时：
   1. 如果当前运行线程数大于 corePoolSize，该线程被停掉
   2. 线程池的所有任务完成后最终会收缩到 corePoolSize 的大小。

**拒绝策略**

在 JDK 中有四种内置的拒绝策略，均实现了 RejectedExecutionHandler 接口。

1. AbortPolicy: 直接抛出 RejectedExecutionException 异常，是默认的拒绝策略。
2. DiscardPolicy: 直接丢弃任务，不予处理也不抛出异常。如果允许任务丢失，是最好的处理策略。
3. DiscardOldestPolicy: 抛弃队列中等待最久的任务，然后把当前任务加入队列尝试再次提交。
4. CallerRunsPolicy: 调用者运行。该策略既不会抛弃任务，也不会抛出异常，而是将某些任务回退到调用者。

**三种常用线程池**

1、Executors.newFixedThreadPool(int)

创建固定容量的线程池，控制最大并发数，超出的线程在队列中等待。

[?](https://www.jb51.net/article/160125.htm)

|  |  |
| --- | --- |
| 1  2  3 | return new ThreadPoolExecutor(nThreads, nThreads,   0L, TimeUnit.MILLISECONDS,   new LinkedBlockingQueue<Runnable>()); |

其中 corePoolSize 和 maximumPoolSize 值是相等的，并且使用的是 LinkedBlockingQueue。  
适用于执行长期的任务，性能比较高。

2、Executors.newSingleThreadExecutor()

创建了一个单线程的线程池，只会用唯一的工作线程来执行任务，保证所有任务按照顺序执行。

[?](https://www.jb51.net/article/160125.htm)

|  |  |
| --- | --- |
| 1  2  3  4 | return new FinalizableDelegatedExecutorService   (new ThreadPoolExecutor(1, 1,    0L, TimeUnit.MILLISECONDS,    new LinkedBlockingQueue<Runnable>())); |

其中 corePoolSize 和 maximumPoolSize 都设置为1，使用的也是 LinkedBlockingQueue。  
适用于一个任务一个任务执行的场景。

3、Executors.newCachedThreadPool()

创建了一个可缓存的线程池，如果线程池长度超过处理需要，可以灵活回收空闲线程，没有可以回收的，则新建线程。

[?](https://www.jb51.net/article/160125.htm)

|  |  |
| --- | --- |
| 1  2  3 | return new ThreadPoolExecutor(0, Integer.MAX\_VALUE,    60L, TimeUnit.SECONDS,    new SynchronousQueue<Runnable>()); |

设置 corePoolSize 为0， maximumPoolSize 设置为 Integer.MAX\_VALUE，使用的是 SynchronousQueue。来了任务就创建线程执行，线程空闲超过60秒后销毁。

适用于执行很多短期异步的小程序或者负载比较轻的服务器。

**工作中使用什么样的线程池**

在阿里巴巴Java开发手册中有如下规定：

1. 线程资源必须通过线程池提供，不允许在应用中自行显示创建线程。
   1. 说明：使用线程池的好处是减少在创建和销毁线程上消耗的时间和系统资源的开销，解决资源不足的问题。如果不使用线程池，有可能造成系统创建大量同类线程导致消耗完内存或者过度切换。
2. 线程池不允许使用 Executors 去创建，也就是不能使用上述的三种线程池，而是要通过 ThreadPoolExecutor 的方式，这样的处理方式让写的同学更加明确线程池的运行规则，规避资源韩进的风险。
   1. FixedThreadPool 和 SingleThreadPool 都采用了 LinkedBlockingQueue，其允许的队列长度为 Integer.MAX\_VALUE，可能堆积大量的请求，导致OOM。
   2. CachedThreadPool 和 ScheduledThreadPool 允许创建的线程数量为 Integer.MAX\_VALUE，可能创建大量的线程，导致OOM。

**如何设置线程池的线程数目**

Runtime.getRuntime().availableProcessors()获取当前设备的CPU个数。

1. CPU密集型任务
   1. CPU 密集的含义是任务需要大量的运算，而没有阻塞，CPU一致全速运行
   2. CPU 密集任务只有在真正的多核 CPU 上才能得到加速（通过多线程），而在单核 CPU 上，无论开几个模拟的多线程都不能得到加速
   3. CPU 密集型任务配置尽可能少的线程数量，一般设置为 CPU 核心数 + 1
2. IO 密集型
   1. IO 密集型，是指该任务需要大量的IO，大量的阻塞
   2. 单线程上运行 IO 密集型的任务会导致浪费大量的 CPU 运算能力浪费在等待上
   3. IO 密集型任务使用多线程可以大大加速程序运行，利用了被浪费掉的阻塞时间
   4. IO 密集型时，大部分线程都阻塞，需要多配置线程数，可以采用CPU核心数 \* 2，或者采用 CPU 核心数 / (1 - 阻塞系数)，阻塞系数在0.8 ~ 0.9之间

**死锁**

**产生死锁的原因**

死锁是指两个或两个以上的进程在执行过程中，因为争夺资源造成的互相等待的现象。

死锁需要满族的四大条件如下：

1. 互斥
2. 循环等待
3. 不可抢占
4. 占有并等待

产生死锁的主要原因有：

1. 系统资源不足
2. 进程运行推进顺序不当
3. 资源分配不当

死锁实例

[?](https://www.jb51.net/article/160125.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | class HoldLockThread implements Runnable{   private String lock1;   private String lock2;     public HoldLockThread(String lock1, String lock2) {    this.lock1 = lock1;    this.lock2 = lock2;   }     @Override   public void run() {    synchronized (lock1){     System.out.println(Thread.currentThread().getName()+"\t持有"+lock1+"\t尝试获取"+lock2);     try {      Thread.sleep(2000);     } catch (InterruptedException e) {      e.printStackTrace();     }     synchronized (lock2){      System.out.println(Thread.currentThread().getName()+"\t持有"+lock1+"\t尝试获取"+lock2);     }    }   }  }    public class DeadLockDemo {   public static void main(String[] args) {    String lockA="lockA";    String lockB="lockB";      new Thread(new HoldLockThread(lockA,lockB),"Thread1").start();    new Thread(new HoldLockThread(lockB,lockA),"Thread2").start();   }  } |

输出如下结果，程序并没有终止。

Thread2 持有lockB 尝试获取lockA  
Thread1 持有lockA 尝试获取lockB

**死锁定位分析**

使用 jps ，类似于 linux 中的 ps 命令。

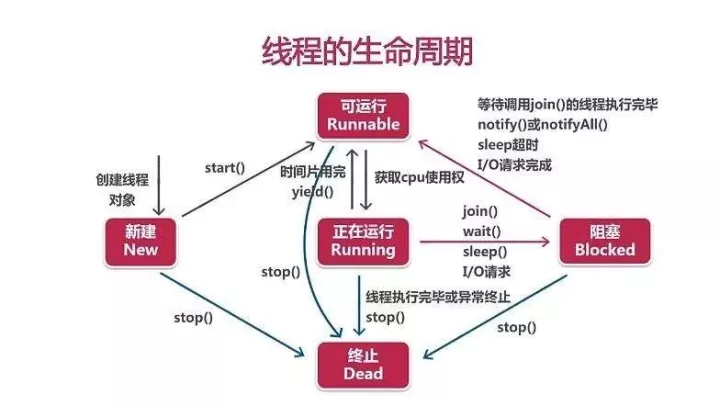
在上述 java 文件中，使用 IDEA 中的 open In Terminal，或者在该文件目录下使用 cmd 命令行工具。

首先使用 jps -l命令，类似于ls -l命令，输出当前运行的 java 线程，从中能得知 DeadLockDemo 线程的线程号。

然后，使用jstack threadId来查看栈信息。输出如下：

Java stack information for the threads listed above:  
===================================================  
"Thread2":  
at interview.jvm.deadlock.HoldLockThread.run(DeadLockDemo.java:22)  
- waiting to lock <0x00000000d6240328> (a java.lang.String)  
- locked <0x00000000d6240360> (a java.lang.String)  
at java.lang.Thread.run(Thread.java:748)  
"Thread1":  
at interview.jvm.deadlock.HoldLockThread.run(DeadLockDemo.java:22)  
- waiting to lock <0x00000000d6240360> (a java.lang.String)  
- locked <0x00000000d6240328> (a java.lang.String)  
at java.lang.Thread.run(Thread.java:748)

Found 1 deadlock.



# Java多线程面试题及回答

**1.现在有T1、T2、T3三个线程，你怎样保证T2在T1执行完后执行，T3在T2执行完后执行？**  
这个线程问题通常会在第一轮或电话面试阶段被问到，目的是检测你对”join”方法是否熟悉。这个多线程问题比较简单，可以用join方法实现。

**2. 在Java中Lock接口比synchronized块的优势是什么？你需要实现一个高效的缓存，它允许多个用户读，但只允许一个用户写，以此来保持它的完整性，你会怎样去实现它？**  
lock接口在多线程和并发编程中最大的优势是它们为读和写分别提供了锁，它能满足你写像ConcurrentHashMap这样的高性能数据结构和有条件的阻塞。Java线程面试的问题越来越会根据面试者的回答来提问。我强烈建议在你去参加多线程的面试之前认真读一下Locks，因为当前其大量用于构建电子交易终统的客户端缓存和交易连接空间。

**3.在java中wait和sleep方法的不同？**  
通常会在电话面试中经常被问到的Java线程面试问题。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。

**4.用Java实现阻塞队列。**  
这是一个相对艰难的多线程面试问题，它能达到很多的目的。第一，它可以检测侯选者是否能实际的用Java线程写程序；第二，可以检测侯选者对并发场景的理解，并且你可以根据这个问很多问题。如果他用wait()和notify()方法来实现阻塞队列，你可以要求他用最新的Java 5中的并发类来再写一次。

**5.用Java写代码来解决生产者——消费者问题。**  
与上面的问题很类似，但这个问题更经典，有些时候面试都会问下面的问题。在Java中怎么解决生产者——消费者问题，当然有很多解决方法，我已经分享了一种用阻塞队列实现的方法。有些时候他们甚至会问怎么实现哲学家进餐问题。

**6.用Java编程一个会导致死锁的程序，你将怎么解决？**  
这是我最喜欢的Java线程面试问题，因为即使死锁问题在写多线程并发程序时非常普遍，但是很多侯选者并不能写deadlock free code（无死锁代码？），他们很挣扎。只要告诉他们，你有N个资源和N个线程，并且你需要所有的资源来完成一个操作。为了简单这里的n可以替换为2，越大的数据会使问题看起来更复杂。通过避免Java中的死锁来得到关于死锁的更多信息。

**7.什么是原子操作，Java中的原子操作是什么？**  
非常简单的java线程面试问题，接下来的问题是你需要同步一个原子操作。

**8.Java中的volatile关键是什么作用？怎样使用它？在Java中它跟synchronized方法有什么不同？**  
自从Java 5和Java内存模型改变以后，基于volatile关键字的线程问题越来越流行。应该准备好回答关于volatile变量怎样在并发环境中确保可见性、顺序性和一致性。

**9.什么是竞争条件？你怎样发现和解决竞争？**  
这是一道出现在多线程面试的高级阶段的问题。大多数的面试官会问最近你遇到的竞争条件，以及你是怎么解决的。有些时间他们会写简单的代码，然后让你检测出代码的竞争条件。可以参考我之前发布的关于Java竞争条件的文章。在我看来这是最好的java线程面试问题之一，它可以确切的检测候选者解决竞争条件的经验，or writing code which is free of data race or any other race condition。关于这方面最好的书是《Concurrency practices in Java》。  
**欢迎关注我的公中浩【程序员追风】，文章都会在里面更新，整理的资料也都会放在里面。**

**10.你将如何使用thread dump？你将如何分析Thread dump？**  
在UNIX中你可以使用kill -3，然后thread dump将会打印日志，在windows中你可以使用”CTRL+Break”。非常简单和专业的线程面试问题，但是如果他问你怎样分析它，就会很棘手。

**11.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？**  
这是另一个非常经典的java多线程面试问题。这也是我刚开始写线程程序时候的困惑。现在这个问题通常在电话面试或者是在初中级Java面试的第一轮被问到。这个问题的回答应该是这样的，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。

**12.Java中你怎样唤醒一个阻塞的线程？**  
这是个关于线程和阻塞的棘手的问题，它有很多解决方法。如果线程遇到了IO阻塞，我并且不认为有一种方法可以中止线程。如果线程因为调用wait()、sleep()、或者join()方法而导致的阻塞，你可以中断线程，并且通过抛出InterruptedException来唤醒它。我之前写的《How to deal with blocking methods in java》有很多关于处理线程阻塞的信息。

**13.在Java中CycliBarriar和CountdownLatch有什么区别？**  
这个线程问题主要用来检测你是否熟悉JDK5中的并发包。这两个的区别是CyclicBarrier可以重复使用已经通过的障碍，而CountdownLatch不能重复使用。

**14.什么是不可变对象，它对写并发应用有什么帮助？**  
另一个多线程经典面试问题，并不直接跟线程有关，但间接帮助很多。这个java面试问题可以变的非常棘手，如果他要求你写一个不可变对象，或者问你为什么String是不可变的。

**15.你在多线程环境中遇到的共同的问题是什么？你是怎么解决它的？**  
多线程和并发程序中常遇到的有Memory-interface、竞争条件、死锁、活锁和饥饿。问题是没有止境的，如果你弄错了，将很难发现和调试。这是大多数基于面试的，而不是基于实际应用的Java线程问题。

**1、为什么用线程池?**

　　有时候，系统需要处理非常多的执行时间很短的请求，如果每一个请求都开启一个新线程的话，系统就要不断的进行线程的创建和销毁，有时花在创建和销毁线程上的时间会比线程真正执行的时间还长。

　　而且当线程数量太多时，系统不一定能受得了。

**使用线程池主要为了解决一下几个问题：**

　　通过重用线程池中的线程，来减少每个线程创建和销毁的性能开销。

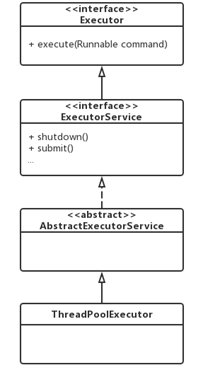
　　对线程进行一些维护和管理，比如定时开始，周期执行，并发数控制等等。

**2、线程池参数什么意思？**

　　比如去火车站买票, 有10个售票窗口, 但只有5个窗口对外开放. 那么对外开放的5个窗口称为核心线程数, 而最大线程数是10个窗口.如果5个窗口都被占用, 那么后来的人就必须在后面排队, 但后来售票厅人越来越多, 已经人满为患, 就类似于线程队列已满.这时候火车站站长下令, 把剩下的5个窗口也打开, 也就是目前已经有10个窗口同时运行. 后来又来了一批人,10个窗口也处理不过来了, 而且售票厅人已经满了, 这时候站长就下令封锁入口,不允许其他人再进来, 这就是线程异常处理策略.而线程存活时间指的是, 允许售票员休息的最长时间, 以此限制售票员偷懒的行为.

**3、讲一讲线程池中的threadpoolexecutor，每个参数干什么用的？**

　　Executor是一个接口，跟线程池有关的基本都要跟他打交道。ThreadPoolExecutor的关系



　　Executor接口很简单，只有一个execute方法。

　　ExecutorService是Executor的子接口，增加了一些常用的对线程的控制方法，之后使用线程池主要也是使用这些方法。

　　AbstractExecutorService是一个抽象类。ThreadPoolExecutor就是实现了这个类。

**ThreadPoolExecutor的参数：**



**corePoolSize**

　　核心线程数，默认情况下核心线程会一直存活，即使处于闲置状态也不会受存keepAliveTime限制。除非将allowCoreThreadTimeOut设置为true。

**maximumPoolSize**

　　线程池所能容纳的最大线程数。超过这个数的线程将被阻塞。当任务队列为没有设置大小的LinkedBlockingDeque时，这个值无效。

**keepAliveTime**

　　非核心线程的闲置超时时间，超过这个时间就会被回收。

**unit**

　　指定keepAliveTime的单位，如TimeUnit.SECONDS。当将allowCoreThreadTimeOut设置为true时对corePoolSize生效。

**workQueue**

　　线程池中的任务队列.

　　常用的有三种队列，SynchronousQueue,LinkedBlockingDeque,ArrayBlockingQueue。

**threadFactory**

　　线程工厂，提供创建新线程的功能。ThreadFactory是一个接口，只有一个方法



　　通过线程工厂可以对线程的一些属性进行定制。

**默认的工厂：**



**RejectedExecutionHandler**

　　RejectedExecutionHandler也是一个接口，只有一个方法:

　　public interface RejectedExecutionHandler {

　　 void rejectedExecution(Runnable var1, ThreadPoolExecutor var2);

　　}

　　当线程池中的资源已经全部使用，添加新线程被拒绝时，会调用RejectedExecutionHandler的rejectedExecution方法。

**4、说一下线程池内部使用规则**

　　线程池的线程执行规则跟任务队列有很大的关系。

**下面都假设任务队列没有大小限制：**

　　如果线程数量<=核心线程数量，那么直接启动一个核心线程来执行任务，不会放入队列中。

　　如果线程数量>核心线程数，但<=最大线程数，并且任务队列是LinkedBlockingDeque的时候，超过核心线程数量的任务会放在任务队列中排队。

　　如果线程数量>核心线程数，但<=最大线程数，并且任务队列是SynchronousQueue的时候，线程池会创建新线程执行任务，这些任务也不会被放在任务队列中。这些线程属于非核心线程，在任务完成后，闲置时间达到了超时时间就会被清除。

　　如果线程数量>核心线程数，并且>最大线程数，当任务队列是LinkedBlockingDeque，会将超过核心线程的任务放在任务队列中排队。也就是当任务队列是LinkedBlockingDeque并且没有大小限制时，线程池的最大线程数设置是无效的，他的线程数最多不会超过核心线程数。

　　如果线程数量>核心线程数，并且>最大线程数，当任务队列是SynchronousQueue的时候，会因为线程池拒绝添加任务而抛出异常。

　　任务队列大小有限时

　　当LinkedBlockingDeque塞满时，新增的任务会直接创建新线程来执行，当创建的线程数量超过最大线程数量时会抛异常。

　　SynchronousQueue没有数量限制。因为他根本不保持这些任务，而是直接交给线程池去执行。当任务数量超过最大线程数时会直接抛异常。

　　看代码 ThreadPoolExecutorTest1-7

**5、用过AtomicInteger吗？怎么用的？**

　　AtomicInteger是int类型的原子操作类，对于全局变量的数值类型操作 num++，若没有加synchronized关键字则是线程不安全的，num++解析为num=num+1，明显，这个操作不具备原子性，多线程时必然会出现问题

　　看代码

**要是换成volatile修饰count变量呢？**

　　volatile修饰的变量能够在线程间保持可见性，能被多个线程同时读但是又能保证只被单线程写，并且不会读取到过期值（由java内存模型中的happen-before原则决定的）volatile修饰字段的写入操作总是优先于读操作，即使多个线程同时修改volatile变量字段，总能保证获取到最新的值

　　看代码

**atomicinteger常用方法：**



**6、用过threadlocal吗？怎么用的？**

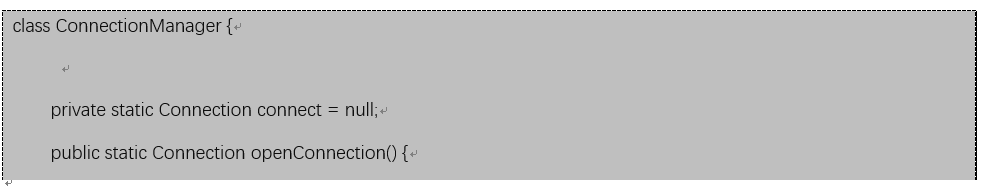
　　早在JDK 1.2的版本中就提供java.lang.ThreadLocal，ThreadLocal为解决多线程程序的并发问题提供了一种新的思路。使用这个工具类可以很简洁地编写出优美的多线程程序。

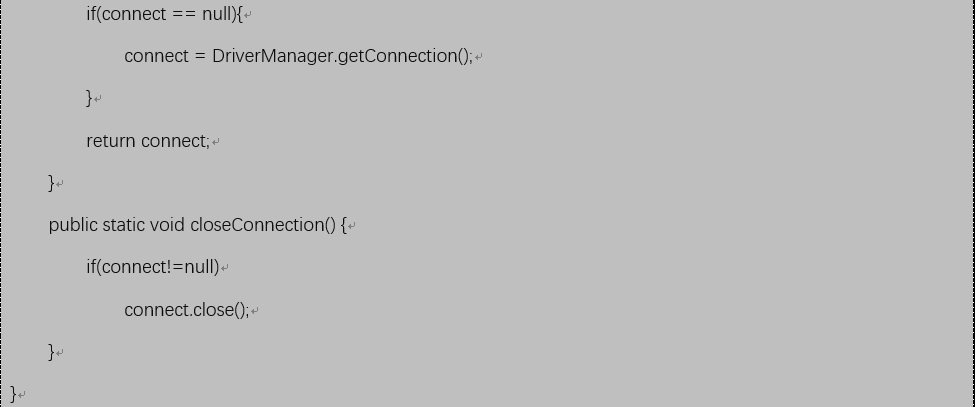
　　ThreadLocal很容易让人望文生义，想当然地认为是一个“本地线程”。

　　其实，ThreadLocal并不是一个Thread，而是Thread的局部变量，也许把它命名为ThreadLocalVariable更容易让人理解一些。

　　ThreadLocal为变量在每个线程中都创建了一个副本，那么每个线程可以访问自己内部的副本变量。

　　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。





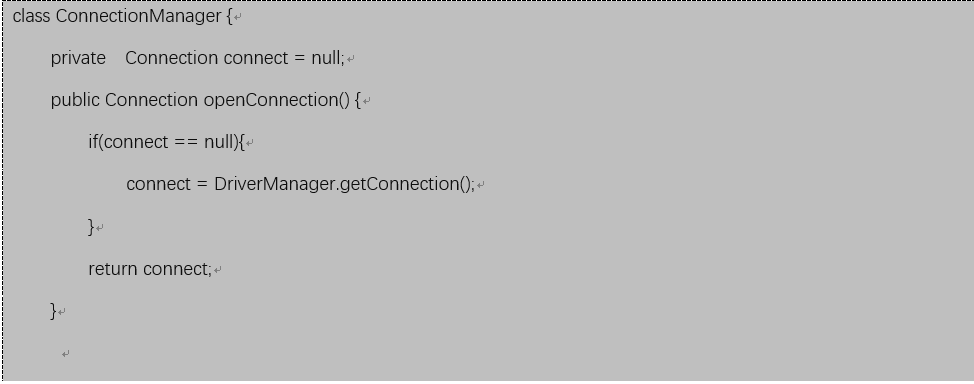
　　 假设有这样一个数据库链接管理类，这段代码在单线程中使用是没有任何问题的，但是如果在多线程中使用呢？很显然，在多线程中使用会存在线程安全问题：第一，这里面的2个方法都没有进行同步，很可能在openConnection方法中会多次创建connect；第二，由于connect是共享变量，那么必然在调用connect的地方需要使用到同步来保障线程安全，因为很可能一个线程在使用connect进行数据库操作，而另外一个线程调用closeConnection关闭链接。

　　所以出于线程安全的考虑，必须将这段代码的两个方法进行同步处理，并且在调用connect的地方需要进行同步处理。

　　这样将会大大影响程序执行效率，因为一个线程在使用connect进行数据库操作的时候，其他线程只有等待。

　　那么大家来仔细分析一下这个问题，这地方到底需不需要将connect变量进行共享？事实上，是不需要的。假如每个线程中都有一个connect变量，各个线程之间对connect变量的访问实际上是没有依赖关系的，即一个线程不需要关心其他线程是否对这个connect进行了修改的。

　　到这里，可能会有朋友想到，既然不需要在线程之间共享这个变量，可以直接这样处理，在每个需要使用数据库连接的方法中具体使用时才创建数据库链接，然后在方法调用完毕再释放这个连接。**比如下面这样：**

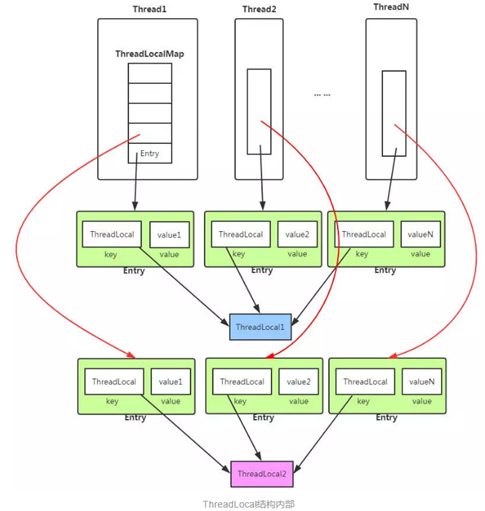




　　 这样处理确实也没有任何问题，由于每次都是在方法内部创建的连接，那么线程之间自然不存在线程安全问题。但是这样会有一个致命的影响：导致服务器压力非常大，并且严重影响程序执行性能。由于在方法中需要频繁地开启和关闭数据库连接，这样不尽严重影响程序执行效率，还可能导致服务器压力巨大。

　　那么这种情况下使用ThreadLocal是再适合不过的了，因为ThreadLocal在每个线程中对该变量会创建一个副本，即每个线程内部都会有一个该变量，且在线程内部任何地方都可以使用，线程之间互不影响，这样一来就不存在线程安全问题，也不会严重影响程序执行性能。

　　但是要注意，虽然ThreadLocal能够解决上面说的问题，但是由于在每个线程中都创建了副本，所以要考虑它对资源的消耗，比如内存的占用会比不使用ThreadLocal要大。



**从上面的结构图，我们已经窥见ThreadLocal的核心机制：**

　　每个Thread线程内部都有一个Map。

　　Map里面存储线程本地对象（key）和线程的变量副本（value）

　　但是，Thread内部的Map是由ThreadLocal维护的，由ThreadLocal负责向map获取和设置线程的变量值。

　　所以对于不同的线程，每次获取副本值时，别的线程并不能获取到当前线程的副本值，形成了副本的隔离，互不干扰。

**ThreadLocal类提供如下几个核心方法：**



　　l get()方法用于获取当前线程的副本变量值。

　　l set()方法用于保存当前线程的副本变量值。

　　l initialValue()为当前线程初始副本变量值。

　　l remove()方法移除当前前程的副本变量值。

**用法 ： 见代码 ThreadLocalTest1**

**7、程序、进程、线程的区别是什么? 举个现实的例子说明。**

　　 程序（Program）：是一个指令的集合。程序不能独立执行，只有被加载到内存中，系统为它分配资源后才能执行。

　　 进程（Process）：如上所述，一个执行中的程序称为进程。

　　 进程是系统分配资源的独立单位，每个进程占有特定的地址空间。

　　 程序是进程的静态文本描述，进程是程序在系统内顺序执行的动态活动。

　　 线程（Thread）：是进程的“单一的连续控制流程“。

　　 线程是CPU调度和分配的基本单位，是比进程更小的能独立运行的基本单位，也被称为轻量级的进程。

　　 线程不能独立存在，必须依附于某个进程。一个进程可以包括多个并行的线程，一个线程肯定属于一个进程。Java虚拟机允许应用程序并发地执行多个线程。

　　 举例：如一个车间是一个程序，一个正在进行生产任务的车间是一个进程，车间内每个从事不同工作的工人是一个线程。

**8、【上机】Java中通过哪些方式创建多线程类? 分别使用代码说明。并调用之。**

　　l 继承Thread类创建线程

　　l 实现Runnable接口创建线程

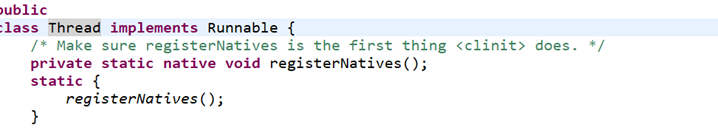
　　l 实现Callable接口通过FutureTask包装器来创建Thread线程

　　l 使用ExecutorService、Callable、Future实现有返回结果的线程

　　见代码ThreadCreateTest1-4

**9、Thread类有没有实现Runnable接口?**

　　 有实现。



**10、当调用一个线程对象的start方法后，线程马上进入运行状态吗?**

　　 不是，只是进入就绪（可运行）状态，等待分配CPU时间片。一旦得到CPU时间片，即进入运行状态。

一 面试中关于 synchronized 关键字的 5 连击

1.1 说一说自己对于 synchronized 关键字的了解

synchronized关键字解决的是多个线程之间访问资源的同步性，synchronized关键字可以保证被它修饰的方法或者代码块在任意时刻只能有一个线程执行。

另外，在 Java 早期版本中，synchronized属于重量级锁，效率低下，因为监视器锁（monitor）是依赖于底层的操作系统的 Mutex Lock 来实现的，Java 的线程是映射到操作系统的原生线程之上的。如果要挂起或者唤醒一个线程，都需要操作系统帮忙完成，而操作系统实现线程之间的切换时需要从用户态转换到内核态，这个状态之间的转换需要相对比较长的时间，时间成本相对较高，这也是为什么早期的 synchronized 效率低的原因。庆幸的是在 Java 6 之后 Java 官方对从 JVM 层面对synchronized 较大优化，所以现在的 synchronized 锁效率也优化得很不错了。JDK1.6对锁的实现引入了大量的优化，如自旋锁、适应性自旋锁、锁消除、锁粗化、偏向锁、轻量级锁等技术来减少锁操作的开销。

1.2 说说自己是怎么使用 synchronized 关键字，在项目中用到了吗

synchronized关键字最主要的三种使用方式：

修饰实例方法，作用于当前对象实例加锁，进入同步代码前要获得当前对象实例的锁

修饰静态方法，作用于当前类对象加锁，进入同步代码前要获得当前类对象的锁 。也就是给当前类加锁，会作用于类的所有对象实例，因为静态成员不属于任何一个实例对象，是类成员（ static 表明这是该类的一个静态资源，不管new了多少个对象，只有一份，所以对该类的所有对象都加了锁）。所以如果一个线程A调用一个实例对象的非静态 synchronized 方法，而线程B需要调用这个实例对象所属类的静态 synchronized 方法，是允许的，不会发生互斥现象，因为访问静态 synchronized 方法占用的锁是当前类的锁，而访问非静态 synchronized 方法占用的锁是当前实例对象锁。

修饰代码块，指定加锁对象，对给定对象加锁，进入同步代码库前要获得给定对象的锁。 和 synchronized 方法一样，synchronized(this)代码块也是锁定当前对象的。synchronized 关键字加到 static 静态方法和 synchronized(class)代码块上都是是给 Class 类上锁。这里再提一下：synchronized关键字加到非 static 静态方法上是给对象实例上锁。另外需要注意的是：尽量不要使用 synchronized(String a) 因为JVM中，字符串常量池具有缓冲功能！

下面我已一个常见的面试题为例讲解一下 synchronized 关键字的具体使用。

1.3 讲一下 synchronized 关键字的底层原理

synchronized 关键字底层原理属于 JVM 层面。

① synchronized 同步语句块的情况

public class SynchronizedDemo {

public void method() {

synchronized (this) {

System.out.println("synchronized 代码块");

}

}

}

1

2

3

4

5

6

7

8

通过 JDK 自带的 javap 命令查看 SynchronizedDemo 类的相关字节码信息：首先切换到类的对应目录执行 javac SynchronizedDemo.java 命令生成编译后的 .class 文件，然后执行javap -c -s -v -l SynchronizedDemo.class。

从上面我们可以看出：

synchronized 同步语句块的实现使用的是 monitorenter 和 monitorexit 指令，其中 monitorenter 指令指向同步代码块的开始位置，monitorexit 指令则指明同步代码块的结束位置。 当执行 monitorenter 指令时，线程试图获取锁也就是获取 monitor(monitor对象存在于每个Java对象的对象头中，synchronized 锁便是通过这种方式获取锁的，也是为什么Java中任意对象可以作为锁的原因) 的持有权.当计数器为0则可以成功获取，获取后将锁计数器设为1也就是加1。相应的在执行 monitorexit 指令后，将锁计数器设为0，表明锁被释放。如果获取对象锁失败，那当前线程就要阻塞等待，直到锁被另外一个线程释放为止。

② synchronized 修饰方法的的情况

public class SynchronizedDemo2 {

public synchronized void method() {

System.out.println("synchronized 方法");

}

}

1

2

3

4

5

6

synchronized 修饰的方法并没有 monitorenter 指令和 monitorexit 指令，取得代之的确实是 ACC\_SYNCHRONIZED 标识，该标识指明了该方法是一个同步方法，JVM 通过该 ACC\_SYNCHRONIZED 访问标志来辨别一个方法是否声明为同步方法，从而执行相应的同步调用。

1.4 说说 JDK1.6 之后的synchronized 关键字底层做了哪些优化，可以详细介绍一下这些优化吗

JDK1.6 对锁的实现引入了大量的优化，如偏向锁、轻量级锁、自旋锁、适应性自旋锁、锁消除、锁粗化等技术来减少锁操作的开销。

锁主要存在四中状态，依次是：无锁状态、偏向锁状态、轻量级锁状态、重量级锁状态，他们会随着竞争的激烈而逐渐升级。注意锁可以升级不可降级，这种策略是为了提高获得锁和释放锁的效率。

关于这几种优化的详细信息可以查看：synchronized 关键字使用、底层原理、JDK1.6 之后的底层优化以及 和ReenTrantLock 的对比

1.5 谈谈 synchronized和ReenTrantLock 的区别

① 两者都是可重入锁

两者都是可重入锁。“可重入锁”概念是：自己可以再次获取自己的内部锁。比如一个线程获得了某个对象的锁，此时这个对象锁还没有释放，当其再次想要获取这个对象的锁的时候还是可以获取的，如果不可锁重入的话，就会造成死锁。同一个线程每次获取锁，锁的计数器都自增1，所以要等到锁的计数器下降为0时才能释放锁。

② synchronized 依赖于 JVM 而 ReenTrantLock 依赖于 API

synchronized 是依赖于 JVM 实现的，前面我们也讲到了 虚拟机团队在 JDK1.6 为 synchronized 关键字进行了很多优化，但是这些优化都是在虚拟机层面实现的，并没有直接暴露给我们。ReenTrantLock 是 JDK 层面实现的（也就是 API 层面，需要 lock() 和 unlock 方法配合 try/finally 语句块来完成），所以我们可以通过查看它的源代码，来看它是如何实现的。

③ ReenTrantLock 比 synchronized 增加了一些高级功能

相比synchronized，ReenTrantLock增加了一些高级功能。主要来说主要有三点：①等待可中断；②可实现公平锁；③可实现选择性通知（锁可以绑定多个条件）

ReenTrantLock提供了一种能够中断等待锁的线程的机制，通过lock.lockInterruptibly()来实现这个机制。也就是说正在等待的线程可以选择放弃等待，改为处理其他事情。

ReenTrantLock可以指定是公平锁还是非公平锁。而synchronized只能是非公平锁。所谓的公平锁就是先等待的线程先获得锁。 ReenTrantLock默认情况是非公平的，可以通过 ReenTrantLock类的ReentrantLock(boolean fair)构造方法来制定是否是公平的。

synchronized关键字与wait()和notify/notifyAll()方法相结合可以实现等待/通知机制，ReentrantLock类当然也可以实现，但是需要借助于Condition接口与newCondition() 方法。Condition是JDK1.5之后才有的，它具有很好的灵活性，比如可以实现多路通知功能也就是在一个Lock对象中可以创建多个Condition实例（即对象监视器），线程对象可以注册在指定的Condition中，从而可以有选择性的进行线程通知，在调度线程上更加灵活。 在使用notify/notifyAll()方法进行通知时，被通知的线程是由 JVM 选择的，用ReentrantLock类结合Condition实例可以实现“选择性通知” ，这个功能非常重要，而且是Condition接口默认提供的。而synchronized关键字就相当于整个Lock对象中只有一个Condition实例，所有的线程都注册在它一个身上。如果执行notifyAll()方法的话就会通知所有处于等待状态的线程这样会造成很大的效率问题，而Condition实例的signalAll()方法 只会唤醒注册在该Condition实例中的所有等待线程。

如果你想使用上述功能，那么选择ReenTrantLock是一个不错的选择。

④ 性能已不是选择标准

二 面试中关于线程池的 4 连击

2.1 讲一下Java内存模型

在 JDK1.2 之前，Java的内存模型实现总是从主存（即共享内存）读取变量，是不需要进行特别的注意的。而在当前的 Java 内存模型下，线程可以把变量保存本地内存（比如机器的寄存器）中，而不是直接在主存中进行读写。这就可能造成一个线程在主存中修改了一个变量的值，而另外一个线程还继续使用它在寄存器中的变量值的拷贝，造成数据的不一致。

要解决这个问题，就需要把变量声明为 volatile，这就指示 JVM，这个变量是不稳定的，每次使用它都到主存中进行读取。

说白了， volatile 关键字的主要作用就是保证变量的可见性然后还有一个作用是防止指令重排序。

2.2 说说 synchronized 关键字和 volatile 关键字的区别

synchronized关键字和volatile关键字比较

volatile关键字是线程同步的轻量级实现，所以volatile性能肯定比synchronized关键字要好。但是volatile关键字只能用于变量而synchronized关键字可以修饰方法以及代码块。synchronized关键字在JavaSE1.6之后进行了主要包括为了减少获得锁和释放锁带来的性能消耗而引入的偏向锁和轻量级锁以及其它各种优化之后执行效率有了显著提升，实际开发中使用 synchronized 关键字的场景还是更多一些。

多线程访问volatile关键字不会发生阻塞，而synchronized关键字可能会发生阻塞

volatile关键字能保证数据的可见性，但不能保证数据的原子性。synchronized关键字两者都能保证。

volatile关键字主要用于解决变量在多个线程之间的可见性，而 synchronized关键字解决的是多个线程之间访问资源的同步性。

三 面试中关于 线程池的 2 连击

3.1 为什么要用线程池？

线程池提供了一种限制和管理资源（包括执行一个任务）。 每个线程池还维护一些基本统计信息，例如已完成任务的数量。

这里借用《Java并发编程的艺术》提到的来说一下使用线程池的好处：

降低资源消耗。 通过重复利用已创建的线程降低线程创建和销毁造成的消耗。

提高响应速度。 当任务到达时，任务可以不需要的等到线程创建就能立即执行。

提高线程的可管理性。 线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。

3.2 实现Runnable接口和Callable接口的区别

如果想让线程池执行任务的话需要实现的Runnable接口或Callable接口。 Runnable接口或Callable接口实现类都可以被ThreadPoolExecutor或ScheduledThreadPoolExecutor执行。两者的区别在于 Runnable 接口不会返回结果但是 Callable 接口可以返回结果。

备注： 工具类Executors可以实现Runnable对象和Callable对象之间的相互转换。（Executors.callable（Runnable task）或Executors.callable（Runnable task，Object resule））。

3.3 执行execute()方法和submit()方法的区别是什么呢？

1)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与否；

2)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，而使用 get（long timeout，TimeUnit unit）方法则会阻塞当前线程一段时间后立即返回，这时候有可能任务没有执行完。

3.4 如何创建线程池

《阿里巴巴Java开发手册》中强制线程池不允许使用 Executors 去创建，而是通过 ThreadPoolExecutor 的方式，这样的处理方式让写的同学更加明确线程池的运行规则，规避资源耗尽的风险\*\*

Executors 返回线程池对象的弊端如下：

FixedThreadPool 和 SingleThreadExecutor ： 允许请求的队列长度为 Integer.MAX\_VALUE,可能堆积大量的请求，从而导致OOM。

CachedThreadPool 和 ScheduledThreadPool ： 允许创建的线程数量为 Integer.MAX\_VALUE ，可能会创建大量线程，从而导致OOM。

方式一：通过构造方法实现

方式二：通过Executor 框架的工具类Executors来实现

我们可以创建三种类型的ThreadPoolExecutor：

FixedThreadPool ： 该方法返回一个固定线程数量的线程池。该线程池中的线程数量始终不变。当有一个新的任务提交时，线程池中若有空闲线程，则立即执行。若没有，则新的任务会被暂存在一个任务队列中，待有线程空闲时，便处理在任务队列中的任务。

SingleThreadExecutor： 方法返回一个只有一个线程的线程池。若多余一个任务被提交到该线程池，任务会被保存在一个任务队列中，待线程空闲，按先入先出的顺序执行队列中的任务。

CachedThreadPool： 该方法返回一个可根据实际情况调整线程数量的线程池。线程池的线程数量不确定，但若有空闲线程可以复用，则会优先使用可复用的线程。若所有线程均在工作，又有新的任务提交，则会创建新的线程处理任务。所有线程在当前任务执行完毕后，将返回线程池进行复用。

对应Executors工具类中的方法如图所示：

四 面试中关于 Atomic 原子类的 4 连击

4.1 介绍一下Atomic 原子类

Atomic 翻译成中文是原子的意思。在化学上，我们知道原子是构成一般物质的最小单位，在化学反应中是不可分割的。在我们这里 Atomic 是指一个操作是不可中断的。即使是在多个线程一起执行的时候，一个操作一旦开始，就不会被其他线程干扰。

所以，所谓原子类说简单点就是具有原子/原子操作特征的类。

并发包 java.util.concurrent 的原子类都存放在java.util.concurrent.atomic下,如下图所示。

4.2 JUC 包中的原子类是哪4类?

基本类型

使用原子的方式更新基本类型

AtomicInteger：整形原子类

AtomicLong：长整型原子类

AtomicBoolean ：布尔型原子类

数组类型

使用原子的方式更新数组里的某个元素

AtomicIntegerArray：整形数组原子类

AtomicLongArray：长整形数组原子类

AtomicReferenceArray ：引用类型数组原子类

引用类型

AtomicReference：引用类型原子类

AtomicStampedRerence：原子更新引用类型里的字段原子类

AtomicMarkableReference ：原子更新带有标记位的引用类型

对象的属性修改类型

AtomicIntegerFieldUpdater:原子更新整形字段的更新器

AtomicLongFieldUpdater：原子更新长整形字段的更新器

AtomicStampedReference ：原子更新带有版本号的引用类型。该类将整数值与引用关联起来，可用于解决原子的更新数据和数据的版本号，可以解决使用 CAS 进行原子更新时可能出现的 ABA 问题。

4.3 讲讲 AtomicInteger 的使用

AtomicInteger 类常用方法

public final int get() //获取当前的值

public final int getAndSet(int newValue)//获取当前的值，并设置新的值

public final int getAndIncrement()//获取当前的值，并自增

public final int getAndDecrement() //获取当前的值，并自减

public final int getAndAdd(int delta) //获取当前的值，并加上预期的值

boolean compareAndSet(int expect, int update) //如果输入的数值等于预期值，则以原子方式将该值设置为输入值（update）

public final void lazySet(int newValue)//最终设置为newValue,使用 lazySet 设置之后可能导致其他线程在之后的一小段时间内还是可以读到旧的值。

1

2

3

4

5

6

7

AtomicInteger 类的使用示例

使用 AtomicInteger 之后，不用对 increment() 方法加锁也可以保证线程安全。

class AtomicIntegerTest {

private AtomicInteger count = new AtomicInteger();

//使用AtomicInteger之后，不需要对该方法加锁，也可以实现线程安全。

public void increment() {

count.incrementAndGet();

}

public int getCount() {

return count.get();

}

}

1

2

3

4

5

6

7

8

9

10

11

12

4.4 能不能给我简单介绍一下 AtomicInteger 类的原理

AtomicInteger 线程安全原理简单分析

AtomicInteger 类的部分源码：

// setup to use Unsafe.compareAndSwapInt for updates（更新操作时提供“比较并替换”的作用）

private static final Unsafe unsafe = Unsafe.getUnsafe();

private static final long valueOffset;

static {

try {

valueOffset = unsafe.objectFieldOffset

(AtomicInteger.class.getDeclaredField("value"));

} catch (Exception ex) { throw new Error(ex); }

}

private volatile int value;

1

2

3

4

5

6

7

8

9

10

11

12

AtomicInteger 类主要利用 CAS (compare and swap) + volatile 和 native 方法来保证原子操作，从而避免 synchronized 的高开销，执行效率大为提升。

CAS的原理是拿期望的值和原本的一个值作比较，如果相同则更新成新的值。UnSafe 类的 objectFieldOffset() 方法是一个本地方法，这个方法是用来拿到“原来的值”的内存地址，返回值是 valueOffset。另外 value 是一个volatile变量，在内存中可见，因此 JVM 可以保证任何时刻任何线程总能拿到该变量的最新值。

关于 Atomic 原子类这部分更多内容可以查看我的这篇文章：并发编程面试必备：JUC 中的 Atomic 原子类总结

五 AQS

5.1 AQS 介绍

AQS的全称为（AbstractQueuedSynchronizer），这个类在java.util.concurrent.locks包下面。

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。当然，我们自己也能利用AQS非常轻松容易地构造出符合我们自己需求的同步器。

5.2 AQS 原理分析

AQS 原理这部分参考了部分博客，在5.2节末尾放了链接。

在面试中被问到并发知识的时候，大多都会被问到“请你说一下自己对于AQS原理的理解”。下面给大家一个示例供大家参加，面试不是背题，大家一定要假如自己的思想，即使加入不了自己的思想也要保证自己能够通俗的讲出来而不是背出来。

下面大部分内容其实在AQS类注释上已经给出了，不过是英语看着比较吃力一点，感兴趣的话可以看看源码。

5.2.1 AQS 原理概览

AQS核心思想是，如果被请求的共享资源空闲，则将当前请求资源的线程设置为有效的工作线程，并且将共享资源设置为锁定状态。如果被请求的共享资源被占用，那么就需要一套线程阻塞等待以及被唤醒时锁分配的机制，这个机制AQS是用CLH队列锁实现的，即将暂时获取不到锁的线程加入到队列中。

CLH(Craig,Landin,and Hagersten)队列是一个虚拟的双向队列（虚拟的双向队列即不存在队列实例，仅存在结点之间的关联关系）。AQS是将每条请求共享资源的线程封装成一个CLH锁队列的一个结点（Node）来实现锁的分配。

看个AQS(AbstractQueuedSynchronizer)原理图：

AQS使用一个int成员变量来表示同步状态，通过内置的FIFO队列来完成获取资源线程的排队工作。AQS使用CAS对该同步状态进行原子操作实现对其值的修改。

private volatile int state;//共享变量，使用volatile修饰保证线程可见性

1

状态信息通过procted类型的getState，setState，compareAndSetState进行操作

//返回同步状态的当前值

protected final int getState() {

return state;

}

// 设置同步状态的值

protected final void setState(int newState) {

state = newState;

}

//原子地（CAS操作）将同步状态值设置为给定值update如果当前同步状态的值等于expect（期望值）

protected final boolean compareAndSetState(int expect, int update) {

return unsafe.compareAndSwapInt(this, stateOffset, expect, update);

}

1

2

3

4

5

6

7

8

9

10

11

12

13

5.2.2 AQS 对资源的共享方式

AQS定义两种资源共享方式

Exclusive（独占）：只有一个线程能执行，如ReentrantLock。又可分为公平锁和非公平锁：

公平锁：按照线程在队列中的排队顺序，先到者先拿到锁

非公平锁：当线程要获取锁时，无视队列顺序直接去抢锁，谁抢到就是谁的

Share（共享）：多个线程可同时执行，如Semaphore/CountDownLatch。Semaphore、CountDownLatCh、 CyclicBarrier、ReadWriteLock 我们都会在后面讲到。

ReentrantReadWriteLock 可以看成是组合式，因为ReentrantReadWriteLock也就是读写锁允许多个线程同时对某一资源进行读。

不同的自定义同步器争用共享资源的方式也不同。自定义同步器在实现时只需要实现共享资源 state 的获取与释放方式即可，至于具体线程等待队列的维护（如获取资源失败入队/唤醒出队等），AQS已经在顶层实现好了。

5.2.3 AQS底层使用了模板方法模式

同步器的设计是基于模板方法模式的，如果需要自定义同步器一般的方式是这样（模板方法模式很经典的一个应用）：

使用者继承AbstractQueuedSynchronizer并重写指定的方法。（这些重写方法很简单，无非是对于共享资源state的获取和释放）

将AQS组合在自定义同步组件的实现中，并调用其模板方法，而这些模板方法会调用使用者重写的方法。

这和我们以往通过实现接口的方式有很大区别，这是模板方法模式很经典的一个运用。

AQS使用了模板方法模式，自定义同步器时需要重写下面几个AQS提供的模板方法：

isHeldExclusively()//该线程是否正在独占资源。只有用到condition才需要去实现它。

tryAcquire(int)//独占方式。尝试获取资源，成功则返回true，失败则返回false。

tryRelease(int)//独占方式。尝试释放资源，成功则返回true，失败则返回false。

tryAcquireShared(int)//共享方式。尝试获取资源。负数表示失败；0表示成功，但没有剩余可用资源；正数表示成功，且有剩余资源。

tryReleaseShared(int)//共享方式。尝试释放资源，成功则返回true，失败则返回false。

1

2

3

4

5

6

默认情况下，每个方法都抛出 UnsupportedOperationException。 这些方法的实现必须是内部线程安全的，并且通常应该简短而不是阻塞。AQS类中的其他方法都是final ，所以无法被其他类使用，只有这几个方法可以被其他类使用。

以ReentrantLock为例，state初始化为0，表示未锁定状态。A线程lock()时，会调用tryAcquire()独占该锁并将state+1。此后，其他线程再tryAcquire()时就会失败，直到A线程unlock()到state=0（即释放锁）为止，其它线程才有机会获取该锁。当然，释放锁之前，A线程自己是可以重复获取此锁的（state会累加），这就是可重入的概念。但要注意，获取多少次就要释放多么次，这样才能保证state是能回到零态的。

再以CountDownLatch以例，任务分为N个子线程去执行，state也初始化为N（注意N要与线程个数一致）。这N个子线程是并行执行的，每个子线程执行完后countDown()一次，state会CAS(Compare and Swap)减1。等到所有子线程都执行完后(即state=0)，会unpark()主调用线程，然后主调用线程就会从await()函数返回，继续后余动作。

一般来说，自定义同步器要么是独占方法，要么是共享方式，他们也只需实现tryAcquire-tryRelease、tryAcquireShared-tryReleaseShared中的一种即可。但AQS也支持自定义同步器同时实现独占和共享两种方式，如ReentrantReadWriteLock。

推荐两篇 AQS 原理和相关源码分析的文章：

http://www.cnblogs.com/waterystone/p/4920797.html

https://www.cnblogs.com/chengxiao/archive/2017/07/24/7141160.html

5.3 AQS 组件总结

Semaphore(信号量)-允许多个线程同时访问： synchronized 和 ReentrantLock 都是一次只允许一个线程访问某个资源，Semaphore(信号量)可以指定多个线程同时访问某个资源。

CountDownLatch （倒计时器）： CountDownLatch是一个同步工具类，用来协调多个线程之间的同步。这个工具通常用来控制线程等待，它可以让某一个线程等待直到倒计时结束，再开始执行。

CyclicBarrier(循环栅栏)： CyclicBarrier 和 CountDownLatch 非常类似，它也可以实现线程间的技术等待，但是它的功能比 CountDownLatch 更加复杂和强大。主要应用场景和 CountDownLatch 类似。CyclicBarrier 的字面意思是可循环使用（Cyclic）的屏障（Barrier）。它要做的事情是，让一组线程到达一个屏障（也可以叫同步点）时被阻塞，直到最后一个线程到达屏障时，屏障才会开门，所有被屏障拦截的线程才会继续干活。CyclicBarrier默认的构造方法是 CyclicBarrier(int parties)，其参数表示屏障拦截的线程数量，每个线程调用await方法告诉 CyclicBarrier 我已经到达了屏障，然后当前线程被阻塞。

关于AQS这部分的更多内容可以查看我的这篇文章:并发编程面试必备：AQS 原理以及 AQS 同步组件总结

上文博客地址：<https://blog.csdn.net/qq_34337272/article/details/83655409>

### 1.线程的sleep()方法和yield()方法有什么区别？

答： ① sleep()方法给其他线程运行机会时不考虑线程的优先级，因此会给低优先级的线程以运行的机会；yield()方法只会给相同优先级或更高优先级的线程以运行的机会； ② 线程执行sleep()方法后转入阻塞（blocked）状态，而执行yield()方法后转入就绪（ready）状态； ③ sleep()方法声明抛出InterruptedException，而yield()方法没有声明任何异常； ④ sleep()方法比yield()方法（跟操作系统CPU调度相关）具有更好的可移植性。

### 2.请说出与线程同步以及线程调度相关的方法。

答：

* wait()：使一个线程处于等待（阻塞）状态，并且释放所持有的对象的锁；
* sleep()：使一个正在运行的线程处于睡眠状态，是一个静态方法，调用此方法要处理InterruptedException异常；
* notify()：唤醒一个处于等待状态的线程，当然在调用此方法的时候，并不能确切的唤醒某一个等待状态的线程，而是由JVM确定唤醒哪个线程，而且与优先级无关；
* notityAll()：唤醒所有处于等待状态的线程，该方法并不是将对象的锁给所有线程，而是让它们竞争，只有获得锁的线程才能进入就绪状态；

### 3.举例说明同步和异步。

答：如果系统中存在临界资源（资源数量少于竞争资源的线程数量的资源），例如正在写的数据以后可能被另一个线程读到，或者正在读的数据可能已经被另一个线程写过了，那么这些数据就必须进行同步存取（数据库操作中的排他锁就是最好的例子）。当应用程序在对象上调用了一个需要花费很长时间来执行的方法，并且不希望让程序等待方法的返回时，就应该使用异步编程，在很多情况下采用异步途径往往更有效率。事实上，所谓的同步就是指阻塞式操作，而异步就是非阻塞式操作。

### 4.不使用stop停止线程？

当run() 或者 call() 方法执行完的时候线程会自动结束,如果要手动结束一个线程，你可以用volatile 布尔变量来退出run()方法的循环或者是取消任务来中断线程。

使用自定义的标志位决定线程的执行情况

public class SafeStopThread implements Runnable{

private volatile boolean stop=false;//此变量必须加上volatile

int a=0;

@Override

public void run() {

// TODO Auto-generated method stub

while(!stop){

synchronized ("") {

a++;

try {

Thread.sleep(100);

} catch (Exception e) {

// TODO: handle exception

}

a--;

String tn=Thread.currentThread().getName();

System.out.println(tn+":a="+a);

}

}

//线程终止

public void terminate(){

stop=true;

}

public static void main(String[] args) {

SafeStopThread t=new SafeStopThread();

Thread t1=new Thread(t);

t1.start();

for(int i=0;i<5;i++){

new Thread(t).start();

}

t.terminate();

}

}

### 5.Java中如何实现线程？各有什么优缺点,比较常用的是那种,为什么？

在语言层面有两种方式。java.lang.Thread 类的实例就是一个线程但是它需要调用java.lang.Runnable接口来执行，由于线程类本身就是调用的Runnable接口所以你可以继承java.lang.Thread 类或者直接调用Runnable接口来重写run()方法实现线程。

Java不支持类的多重继承，但允许你调用多个接口。所以如果你要继承其他类，当然是调用Runnable接口好了。

### 6.如何控制某个方法允许并发访问线程的大小？

Semaphore两个重要的方法就是semaphore.acquire() 请求一个信号量，这时候的信号量个数-1（一旦没有可使用的信号量，也即信号量个数变为负数时，再次请求的时候就会阻塞，直到其他线程释放了信号量）semaphore.release()释放一个信号量，此时信号量个数+1

public class SemaphoreTest {

private Semaphore mSemaphore = new Semaphore(5);

public void run(){

for(int i=0; i< 100; i++){

new Thread(new Runnable() {

@Override

public void run() {

test();

}

}).start();

}

}

private void test(){

try {

mSemaphore.acquire();

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println(Thread.currentThread().getName() + " 进来了");

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println(Thread.currentThread().getName() + " 出去了");

mSemaphore.release();

}

}

### 7.在Java中什么是线程调度？

线程调度是指系统为线程分配处理器使用权的过程。 主要调度方式有两种，分别是协同式线程调度和抢占式线程调度。

协同式线程调度：线程的执行时间由线程本身控制，当线程把自己的工作执行完了之后，主动通知系统切换到另一个线程上。

* 好处是切换操作对于线程自己是可知的，没什么线程同步问题。
* 坏处是线程执行时间不可控，可能会一直阻塞然后系统崩溃。

抢占式线程调度：每个线程由系统分配执行时间，不由线程本身决定。线程的执行时间是系统可控的，不会有一直阻塞的问题。

**Java使用抢占式调度**

### 8.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。

### 9.线程类的构造方法、静态块是被哪个线程调用的？

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。

### 10.在实现Runnable的接口中怎么样访问当前线程对象,比如拿到当前线程的名字？

Thread t = Thread.currentThread();

String name = t.getName();

System.out.println("name=" + name);

### 11.什么是线程池？为什么要使用它？为什么使用Executor框架比使用应用创建和管理线程好？

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。

Executor框架让你可以创建不同的线程池。比如单线程池，每次处理一个任务；数目固定的线程池或者是缓存线程池（一个适合很多生存期短的任务的程序的可扩展线程池）。

### 12常用的线程池模式以及不同线程池的使用场景？

以下是Java自带的几种线程池： 1、newFixedThreadPool 创建一个指定工作线程数量的线程池。 每当提交一个任务就创建一个工作线程，如果工作线程数量达到线程池初始的最大数，则将提交的任务存入到池队列中。

2、newCachedThreadPool 创建一个可缓存的线程池。 这种类型的线程池特点是：

* 1).工作线程的创建数量几乎没有限制(其实也有限制的,数目为Interger. MAX\_VALUE),这样可灵活的往线程池中添加线程。
* 2).如果长时间没有往线程池中提交任务，即如果工作线程空闲了指定的时间(默认为1分钟)，则该工作线程将自动终止。终止后，如果你又提交了新的任务，则线程池重新创建一个工作线程。

3、newSingleThreadExecutor创建一个单线程化的Executor，即只创建唯一的工作者线程来执行任务，如果这个线程异常结束，会有另一个取代它，保证顺序执行(我觉得这点是它的特色)。

单工作线程最大的特点是可保证顺序地执行各个任务，并且在任意给定的时间不会有多个线程是活动的。

4、newScheduleThreadPool 创建一个定长的线程池，而且支持定时的以及周期性的任务执行，类似于Timer。

### 13.在Java中Executor、ExecutorService、Executors的区别？

Executor 和 ExecutorService 这两个接口主要的区别是：

* ExecutorService 接口继承了 Executor 接口，是 Executor 的子接口
* Executor 和 ExecutorService 第二个区别是：Executor 接口定义了 execute()方法用来接收一个Runnable接口的对象，而 ExecutorService 接口中的 submit()方法可以接受Runnable和Callable接口的对象。
* Executor 和 ExecutorService 接口第三个区别是 Executor 中的 execute() 方法不返回任何结果，而 ExecutorService 中的 submit()方法可以通过一个 Future 对象返回运算结果。
* Executor 和 ExecutorService 接口第四个区别是除了允许客户端提交一个任务，ExecutorService 还提供用来控制线程池的方法。比如：调用 shutDown() 方法终止线程池。

Executors 类提供工厂方法用来创建不同类型的线程池。

比如: newSingleThreadExecutor() 创建一个只有一个线程的线程池，newFixedThreadPool(int numOfThreads)来创建固定线程数的线程池，newCachedThreadPool()可以根据需要创建新的线程，但如果已有线程是空闲的会重用已有线程。

### 14.如何创建一个Java线程池？

Java通过Executors提供四种线程池，分别为：

newCachedThreadPool创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。

newFixedThreadPool 创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。

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

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，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。

### 15.Thread 类中的start() 和 run() 方法有什么区别？

start()方法被用来启动新创建的线程，而且start()内部调用了run()方法，这和直接调用run()方法的效果不一样。

当你调用run()方法的时候，只会是在原来的线程中调用，没有新的线程启动，start()方法才会启动新线程。

### 16.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。

### 17.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。

当有线程调用了对象的 notifyAll()方法（唤醒所有 wait 线程）或 notify()方法（只随机唤醒一个 wait 线程），被唤醒的的线程便会进入该对象的锁池中，锁池中的线程会去竞争该对象锁。也就是说，调用了notify后只要一个线程会由等待池进入锁池，而notifyAll会将该对象等待池内的所有线程移动到锁池中，等待锁竞争

优先级高的线程竞争到对象锁的概率大，假若某线程没有竞争到该对象锁，它还会留在锁池中，唯有线程再次调用 wait()方法，它才会重新回到等待池中。

### 18.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。

如果线程需要等待某些锁那么调用对象中的wait()方法就有意义了。如果wait()方法定义在Thread类中，线程正在等待的是哪个锁就不明显了。

简单的说，由于wait，notify和notifyAll都是锁级别的操作，所以把他们定义在Object类中因为锁属于对象。

### 19.为什么wait和notify方法要在同步块中调用？

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

最主要的原因是为了防止以下这种情况

// 等待者(Thread1)

while (condition != true) { // step.1

lock.wait() // step.4

}

// 唤醒者(Thread2)

condition = true; // step.2

lock.notify(); // step.3

在对之前的代码去掉 synchronized 块之后，如果在等待者判断 condition != true 之后而调用 wait() 之前，唤醒者\*\*将 condition 修改成了 true 同时调用了 notify() \*\*的话，那么等待者在调用了 wait() 之后就没有机会被唤醒了。

### 20.讲下join,yield方法的作用,以及什么场合用它们？

join() 的作用：让“主线程”等待“子线程”结束之后才能继续运行。

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

### 21.sleep方法有什么作用,一般用来做什么？

sleep()方法（休眠）是线程类（Thread）的静态方法，调用此方法会让当前线程暂停执行指定的时间，**将执行机会（CPU）让给其他线程**，但是对象的锁依然保持，因此休眠时间结束后会自动恢复。注意这里的恢复并不是恢复到执行的状态，而是恢复到可运行状态中等待CPU的宠幸。

### 22.Java多线程中调用wait() 和 sleep()方法有什么不同？

Java程序中wait和sleep都会造成某种形式的暂停，它们可以满足不同的需要。

* wait存在于Object类中；sleep存在于Thread类中。
* wait会让出CPU资源以及释放锁；sleep只会释放CPU资源。
* wait只能在同步块中使用；sleep没这限制。
* wait需要notify（或 notifyAll）唤醒，进入等锁状态；sleep到指定时间便会自动恢复到运行状态。

### 23.为什么Thread里面的大部分方法都是final的？

不能被重写，线程的很多方法都是由系统调用的，不能通过子类覆写去改变他们的行为。

### 24.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。

该代码只有在某个A线程执行时会被执行，这种情况下通知某个B线程yield是无意义的（因为B线程本来就没在执行）。因此只有当前线程执行yield才是有意义的。通过使该方法为static，你将不会浪费时间尝试yield 其他线程。

只能给自己喂安眠药，不能给别人喂安眠药。

### 25.什么是阻塞式方法？

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。

### 26.如何强制启动一个线程？

在Java里面没有办法强制启动一个线程，它是被线程调度器控制着

### 27.一个线程运行时发生异常会怎样？

简单的说，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。

### 28.在线程中你怎么处理不可控制异常？

在Java中有两种异常。

非运行时异常（Checked Exception）：这种异常必须在方法声明的throws语句指定，或者在方法体内捕获。例如：IOException和ClassNotFoundException。

运行时异常（Unchecked Exception）：这种异常不必在方法声明中指定，也不需要在方法体中捕获。例如，NumberFormatException。

因为run()方法不支持throws语句，所以当线程对象的run()方法抛出非运行异常时，我们必须捕获并且处理它们。当运行时异常从run()方法中抛出时，默认行为是在控制台输出堆栈记录并且退出程序。

好在，java提供给我们一种在线程对象里捕获和处理运行时异常的一种机制。实现用来处理运行时异常的类，这个类实现UncaughtExceptionHandler接口并且实现这个接口的uncaughtException()方法。示例：

package concurrency;

import java.lang.Thread.UncaughtExceptionHandler;

public class Main2 {

public static void main(String[] args) {

Task task = new Task();

Thread thread = new Thread(task);

thread.setUncaughtExceptionHandler(new ExceptionHandler());

thread.start();

}

}

class Task implements Runnable{

@Override

public void run() {

int numero = Integer.parseInt("TTT");

}

}

class ExceptionHandler implements UncaughtExceptionHandler{

@Override

public void uncaughtException(Thread t, Throwable e) {

System.out.printf("An exception has been captured\n");

System.out.printf("Thread: %s\n", t.getId());

System.out.printf("Exception: %s: %s\n", e.getClass().getName(),e.getMessage());

System.out.printf("Stack Trace: \n");

e.printStackTrace(System.out);

System.out.printf("Thread status: %s\n",t.getState());

}

}

当一个线程抛出了异常并且没有被捕获时（这种情况只可能是运行时异常），JVM检查这个线程是否被预置了未捕获异常处理器。如果找到，JVM将调用线程对象的这个方法，并将线程对象和异常作为传入参数。

Thread类还有另一个方法可以处理未捕获到的异常，即静态方法setDefaultUncaughtExceptionHandler()。这个方法在应用程序中为所有的线程对象创建了一个异常处理器。

当线程抛出一个未捕获到的异常时，JVM将为异常寻找以下三种可能的处理器。

* 首先，它查找线程对象的未捕获异常处理器。
* 如果找不到，JVM继续查找线程对象所在的线程组（ThreadGroup）的未捕获异常处理器。
* 如果还是找不到，如同本节所讲的，JVM将继续查找默认的未捕获异常处理器。
* 如果没有一个处理器存在，JVM则将堆栈异常记录打印到控制台，并退出程序。

### 29.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。

1、一般来说，wait肯定是在某个条件调用的，不是if就是while 2、放在while里面，是防止出于waiting的对象被别的原因调用了唤醒方法，但是while里面的条件并没有满足（也可能当时满足了，但是由于别的线程操作后，又不满足了），就需要再次调用wait将其挂起。 3、其实还有一点，就是while最好也被同步，这样不会导致错失信号。

while(condition){

wait();

}

### 30.多线程中的忙循环是什么?

忙循环就是程序员用循环让一个线程等待，不像传统方法wait()、 sleep() 或 yield()，它们都放弃了CPU控制，而忙循环不会放弃CPU，它就是在运行一个空循环。

这么做的目的是为了保留CPU缓存，在多核系统中，一个等待线程醒来的时候可能会在另一个内核运行，这样会重建缓存。为了避免重建缓存和减少等待重建的时间就可以使用它了。

### 31.什么是自旋锁？

没有获得锁的线程一直循环在那里看是否该锁的保持者已经释放了锁，这就是自旋锁。

### 32.什么是互斥锁？

互斥锁：从等待到解锁过程，线程会从sleep状态变为running状态，过程中有线程上下文的切换，抢占CPU等开销。

### 33.自旋锁的优缺点？

自旋锁不会引起调用者休眠，如果自旋锁已经被别的线程保持，调用者就一直循环在那里看是否该自旋锁的保持者释放了锁。由于自旋锁不会引起调用者休眠，所以自旋锁的效率远高于互斥锁。

虽然自旋锁效率比互斥锁高，但它会存在下面两个问题： 1、自旋锁一直占用CPU，在未获得锁的情况下，一直运行，如果不能在很短的时间内获得锁，会导致CPU效率降低。 2、试图递归地获得自旋锁会引起死锁。递归程序决不能在持有自旋锁时调用它自己，也决不能在递归调用时试图获得相同的自旋锁。

由此可见，我们要慎重的使用自旋锁，自旋锁适合于锁使用者保持锁时间比较短并且锁竞争不激烈的情况。正是由于自旋锁使用者一般保持锁时间非常短，因此选择自旋而不是睡眠是非常必要的，自旋锁的效率远高于互斥锁。

### 34.如何在两个线程间共享数据？

同一个Runnable，使用全局变量。

第一种：将共享数据封装到一个对象中，把这个共享数据所在的对象传递给不同的Runnable

第二种：将这些Runnable对象作为某一个类的内部类，共享的数据作为外部类的成员变量，对共享数据的操作分配给外部类的方法来完成，以此实现对操作共享数据的互斥和通信，作为内部类的Runnable来操作外部类的方法，实现对数据的操作

class ShareData {

private int x = 0;

public synchronized void addx(){

x++;

System.out.println("x++ : "+x);

}

public synchronized void subx(){

x--;

System.out.println("x-- : "+x);

}

}

public class ThreadsVisitData {

public static ShareData share = new ShareData();

public static void main(String[] args) {

//final ShareData share = new ShareData();

new Thread(new Runnable() {

public void run() {

for(int i = 0;i<100;i++){

share.addx();

}

}

}).start();

new Thread(new Runnable() {

public void run() {

for(int i = 0;i<100;i++){

share.subx();

}

}

}).start();

}

}

### 35Java中Runnable和Callable有什么不同？

Runnable和Callable都是接口, 不同之处： 1.Callable可以返回一个类型V，而Runnable不可以 2.Callable能够抛出checked exception,而Runnable不可以。 3.Runnable是自从java1.1就有了，而Callable是1.5之后才加上去的 4.Callable和Runnable都可以应用于executors。而Thread类只支持Runnable.

import java.util.concurrent.Callable;

import java.util.concurrent.ExecutionException;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.Future;

public class ThreadTestB {

public static void main(String[] args) {

ExecutorService e=Executors.newFixedThreadPool(10);

Future f1=e.submit(new MyCallableA());

Future f2=e.submit(new MyCallableA());

Future f3=e.submit(new MyCallableA());

System.out.println("--Future.get()....");

try {

System.out.println(f1.get());

System.out.println(f2.get());

System.out.println(f3.get());

} catch (InterruptedException e1) {

e1.printStackTrace();

} catch (ExecutionException e1) {

e1.printStackTrace();

}

e.shutdown();

}

}

class MyCallableA implements Callable<String>{

public String call() throws Exception {

System.out.println("开始执行Callable");

String[] ss={"zhangsan","lisi"};

long[] num=new long[2];

for(int i=0;i<1000000;i++){

num[(int)(Math.random()\*2)]++;

}

if(num[0]>num[1]){

return ss[0];

}else if(num[0]<num[1]){

throw new Exception("弃权!");

}else{

return ss[1];

}

}

}

### 36.Java中CyclicBarrier 和 CountDownLatch有什么不同？

CountDownLatch和CyclicBarrier都能够实现线程之间的等待，只不过它们侧重点不同：

* CountDownLatch一般用于某个线程A等待若干个其他线程执行完任务之后，它才执行；
* 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；
* 另外，CountDownLatch是不能够重用的，而CyclicBarrier是可以重用的。

CountDownLatch的用法:

public class Test {

public static void main(String[] args) {

final CountDownLatch latch = new CountDownLatch(2);

new Thread(){

public void run() {

try {

System.out.println("子线程"+Thread.currentThread().getName()+"正在执行");

Thread.sleep(3000);

System.out.println("子线程"+Thread.currentThread().getName()+"执行完毕");

latch.countDown();

} catch (InterruptedException e) {

e.printStackTrace();

}

};

}.start();

new Thread(){

public void run() {

try {

System.out.println("子线程"+Thread.currentThread().getName()+"正在执行");

Thread.sleep(3000);

System.out.println("子线程"+Thread.currentThread().getName()+"执行完毕");

latch.countDown();

} catch (InterruptedException e) {

e.printStackTrace();

}

};

}.start();

try {

System.out.println("等待2个子线程执行完毕...");

latch.await();

System.out.println("2个子线程已经执行完毕");

System.out.println("继续执行主线程");

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

CyclicBarrier用法:

public class Test {

public static void main(String[] args) {

int N = 4;

CyclicBarrier barrier = new CyclicBarrier(N,new Runnable() {

@Override

public void run() {

System.out.println("当前线程"+Thread.currentThread().getName());

}

});

for(int i=0;i<N;i++)

new Writer(barrier).start();

}

static class Writer extends Thread{

private CyclicBarrier cyclicBarrier;

public Writer(CyclicBarrier cyclicBarrier) {

this.cyclicBarrier = cyclicBarrier;

}

@Override

public void run() {

System.out.println("线程"+Thread.currentThread().getName()+"正在写入数据...");

try {

Thread.sleep(5000); //以睡眠来模拟写入数据操作

System.out.println("线程"+Thread.currentThread().getName()+"写入数据完毕，等待其他线程写入完毕");

cyclicBarrier.await();

} catch (InterruptedException e) {

e.printStackTrace();

}catch(BrokenBarrierException e){

e.printStackTrace();

}

System.out.println("所有线程写入完毕，继续处理其他任务...");

}

}

}

### 37.Java中interrupted和isInterruptedd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。

isInterrupted 只是简单的查询中断状态，不会对状态进行修改。

### 38.concurrentHashMap的源码理解以及内部实现原理，为什么他是同步的且效率高

#### ConcurrentHashMap 分析

ConcurrentHashMap的结构是比较复杂的，都深究去本质，其实也就是数组和链表而已。我们由浅入深慢慢的分析其结构。

先简单分析一下，ConcurrentHashMap 的成员变量中，包含了一个 Segment 的数组（final Segment<K,V>[] segments;），而 Segment 是 ConcurrentHashMap 的内部类，然后在 Segment 这个类中，包含了一个 HashEntry 的数组（transient volatile HashEntry<K,V>[] table;）。而 HashEntry 也是ConcurrentHashMap 的内部类。HashEntry 中，包含了 key 和 value 以及 next 指针（类似于 HashMap 中 Entry），所以 HashEntry 可以构成一个链表。

所以通俗的讲，ConcurrentHashMap 数据结构为一个 Segment 数组，Segment 的数据结构为 HashEntry 的数组，而 HashEntry 存的是我们的键值对，可以构成链表。

首先，我们看一下 HashEntry 类。

#### HashEntry

HashEntry 用来封装散列映射表中的键值对。在 HashEntry 类中，key，hash 和 next 域都被声明为 final 型，value 域被声明为 volatile 型。其类的定义为：

static final class HashEntry<K,V> {

final int hash;

final K key;

volatile V value;

volatile HashEntry<K,V> next;

HashEntry(int hash, K key, V value, HashEntry<K,V> next) {

this.hash = hash;

this.key = key;

this.value = value;

this.next = next;

}

...

...

}

HashEntry 的学习可以类比着 HashMap 中的 Entry。我们的存储键值对的过程中，散列的时候如果发生“碰撞”，将采用“分离链表法”来处理碰撞：把碰撞的 HashEntry 对象链接成一个链表。

如下图，我们在一个空桶中插入 A、B、C 两个 HashEntry 对象后的结构图（其实应该为键值对，在这进行了简化以方便更容易理解）：

https://ask.qcloudimg.com/http-save/5473698/s3hw20ojyt.png?imageView2/2/w/1620

#### Segment

Segment 的类定义为static final class Segment<K,V> extends ReentrantLock implements Serializable。其继承于 ReentrantLock 类，从而使得 Segment 对象可以充当锁的角色。Segment 中包含HashEntry 的数组，其可以守护其包含的若干个桶（HashEntry的数组）。Segment 在某些意义上有点类似于 HashMap了，都是包含了一个数组，而数组中的元素可以是一个链表。

table:table 是由 HashEntry 对象组成的数组如果散列时发生碰撞，碰撞的 HashEntry 对象就以链表的形式链接成一个链表table数组的数组成员代表散列映射表的一个桶每个 table 守护整个 ConcurrentHashMap 包含桶总数的一部分如果并发级别为 16，table 则守护 ConcurrentHashMap 包含的桶总数的 1/16。

count 变量是计算器，表示每个 Segment 对象管理的 table 数组（若干个 HashEntry 的链表）包含的HashEntry 对象的个数。之所以在每个Segment对象中包含一个 count 计数器，而不在 ConcurrentHashMap 中使用全局的计数器，是为了避免出现“热点域”而影响并发性。

/\*\*

\* Segments are specialized versions of hash tables. This

\* subclasses from ReentrantLock opportunistically, just to

\* simplify some locking and avoid separate construction.

\*/

static final class Segment<K,V> extends ReentrantLock implements Serializable {

/\*\*

\* The per-segment table. Elements are accessed via

\* entryAt/setEntryAt providing volatile semantics.

\*/

transient volatile HashEntry<K,V>[] table;

/\*\*

\* The number of elements. Accessed only either within locks

\* or among other volatile reads that maintain visibility.

\*/

transient int count;

transient int modCount;

/\*\*

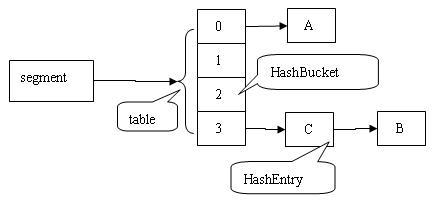
\* 装载因子

\*/

final float loadFactor;

}

我们通过下图来展示一下插入 ABC 三个节点后，Segment 的示意图：

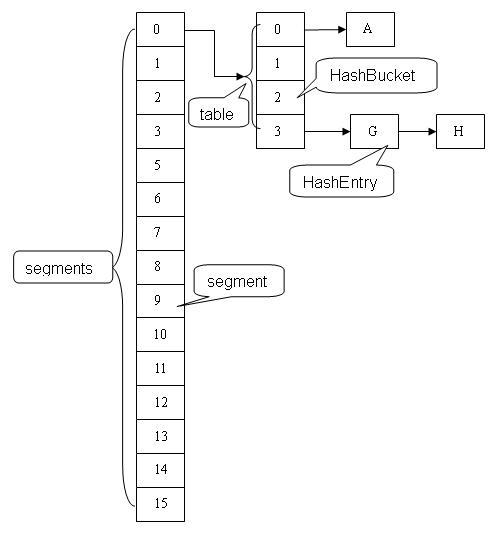


其实从我个人角度来说，Segment结构是与HashMap很像的。

#### ConcurrentHashMap

ConcurrentHashMap 的结构中包含的 Segment 的数组，在默认的并发级别会创建包含 16 个 Segment 对象的数组。通过我们上面的知识，我们知道每个 Segment 又包含若干个散列表的桶，每个桶是由 HashEntry 链接起来的一个链表。如果 key 能够均匀散列，每个 Segment 大约守护整个散列表桶总数的 1/16。

下面我们还有通过一个图来演示一下 ConcurrentHashMap 的结构：



#### 并发写操作

在 ConcurrentHashMap 中，当执行 put 方法的时候，会需要加锁来完成。我们通过代码来解释一下具体过程： 当我们 new 一个 ConcurrentHashMap 对象，并且执行put操作的时候，首先会执行 ConcurrentHashMap 类中的 put 方法，该方法源码为：

/\*\*

\* Maps the specified key to the specified value in this table.

\* Neither the key nor the value can be null.

\*

\* <p> The value can be retrieved by calling the <tt>get</tt> method

\* with a key that is equal to the original key.

\*

\* @param key key with which the specified value is to be associated

\* @param value value to be associated with the specified key

\* @return the previous value associated with <tt>key</tt>, or

\* <tt>null</tt> if there was no mapping for <tt>key</tt>

\* @throws NullPointerException if the specified key or value is null

\*/

@SuppressWarnings("unchecked")

public V put(K key, V value) {

Segment<K,V> s;

if (value == null)

throw new NullPointerException();

int hash = hash(key);

int j = (hash >>> segmentShift) & segmentMask;

if ((s = (Segment<K,V>)UNSAFE.getObject // nonvolatile; recheck

(segments, (j << SSHIFT) + SBASE)) == null) // in ensureSegment

s = ensureSegment(j);

return s.put(key, hash, value, false);

}

我们通过注释可以了解到，ConcurrentHashMap 不允许空值。该方法首先有一个 Segment 的引用 s，然后会通过 hash() 方法对 key 进行计算，得到哈希值；继而通过调用 Segment 的 put(K key, int hash, V value, boolean onlyIfAbsent)方法进行存储操作。该方法源码为：

final V put(K key, int hash, V value, boolean onlyIfAbsent) {

//加锁，这里是锁定的Segment而不是整个ConcurrentHashMap

HashEntry<K,V> node = tryLock() ? null :scanAndLockForPut(key, hash, value);

V oldValue;

try {

HashEntry<K,V>[] tab = table;

//得到hash对应的table中的索引index

int index = (tab.length - 1) & hash;

//找到hash对应的是具体的哪个桶，也就是哪个HashEntry链表

HashEntry<K,V> first = entryAt(tab, index);

for (HashEntry<K,V> e = first;;) {

if (e != null) {

K k;

if ((k = e.key) == key ||

(e.hash == hash && key.equals(k))) {

oldValue = e.value;

if (!onlyIfAbsent) {

e.value = value;

++modCount;

}

break;

}

e = e.next;

}

else {

if (node != null)

node.setNext(first);

else

node = new HashEntry<K,V>(hash, key, value, first);

int c = count + 1;

if (c > threshold && tab.length < MAXIMUM\_CAPACITY)

rehash(node);

else

setEntryAt(tab, index, node);

++modCount;

count = c;

oldValue = null;

break;

}

}

} finally {

//解锁

unlock();

}

return oldValue;

}

关于该方法的某些关键步骤，在源码上加上了注释。

需要注意的是：加锁操作是针对的 hash 值对应的某个 Segment，而不是整个 ConcurrentHashMap。因为 put 操作只是在这个 Segment 中完成，所以并不需要对整个 ConcurrentHashMap 加锁。所以，此时，其他的线程也可以对另外的 Segment 进行 put 操作，因为虽然该 Segment 被锁住了，但其他的 Segment 并没有加锁。同时，读线程并不会因为本线程的加锁而阻塞。

正是因为其内部的结构以及机制，所以 ConcurrentHashMap 在并发访问的性能上要比Hashtable和同步包装之后的HashMap的性能提高很多。在理想状态下，ConcurrentHashMap 可以支持 16 个线程执行并发写操作（如果并发级别设置为 16），及任意数量线程的读操作。

#### 总结

在实际的应用中，散列表一般的应用场景是：除了少数插入操作和删除操作外，绝大多数都是读取操作，而且读操作在大多数时候都是成功的。正是基于这个前提，ConcurrentHashMap 针对读操作做了大量的优化。通过 HashEntry 对象的不变性和用 volatile 型变量协调线程间的内存可见性，使得 大多数时候，读操作不需要加锁就可以正确获得值。这个特性使得 ConcurrentHashMap 的并发性能在分离锁的基础上又有了近一步的提高。

ConcurrentHashMap 是一个并发散列映射表的实现，它允许完全并发的读取，并且支持给定数量的并发更新。相比于 HashTable 和用同步包装器包装的 HashMap（Collections.synchronizedMap(new HashMap())），ConcurrentHashMap 拥有更高的并发性。在 HashTable 和由同步包装器包装的 HashMap 中，使用一个全局的锁来同步不同线程间的并发访问。同一时间点，只能有一个线程持有锁，也就是说在同一时间点，只能有一个线程能访问容器。这虽然保证多线程间的安全并发访问，但同时也导致对容器的访问变成串行化的了。

ConcurrentHashMap 的高并发性主要来自于三个方面：

* 用分离锁实现多个线程间的更深层次的共享访问。
* 用 HashEntery 对象的不变性来降低执行读操作的线程在遍历链表期间对加锁的需求。
* 通过对同一个 Volatile 变量的写 / 读访问，协调不同线程间读 / 写操作的内存可见性。

使用分离锁，减小了请求 同一个锁的频率。

通过 HashEntery 对象的不变性及对同一个 Volatile 变量的读 / 写来协调内存可见性，使得 读操作大多数时候不需要加锁就能成功获取到需要的值。由于散列映射表在实际应用中大多数操作都是成功的 读操作，所以 2 和 3 既可以减少请求同一个锁的频率，也可以有效减少持有锁的时间。通过减小请求同一个锁的频率和尽量减少持有锁的时间 ，使得 ConcurrentHashMap 的并发性相对于 HashTable 和用同步包装器包装的 HashMap有了质的提高。

### 39.BlockingQueue的使用？

#### BlockingQueue的原理

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。

#### BlockingQueue的核心方法：

1)add(E e): 添加元素,如果BlockingQueue可以容纳,则返回true,否则报异常

2)offer(E e): 添加元素,如果BlockingQueue可以容纳,则返回true,否则返回false.

3)put(E e): 添加元素,如果BlockQueue没有空间,则调用此方法的线程被阻断直到BlockingQueue里面有空间再继续.

4)poll(long timeout, TimeUnit timeUnit): 取走BlockingQueue里排在首位的对象,若不能立即取出,则可以等timeout参数规定的时间,取不到时返回null

5)take(): 取走BlockingQueue里排在首位的对象,若BlockingQueue为空,阻断进入等待状态直到Blocking有新的对象被加入为止

#### BlockingQueue常用实现类

1)ArrayBlockingQueue: 有界的先入先出顺序队列，构造方法确定队列的大小.

2)LinkedBlockingQueue: 无界的先入先出顺序队列，构造方法提供两种，一种初始化队列大小，队列即有界；第二种默认构造方法，队列无界（有界即Integer.MAX\_VALUE）

4)SynchronousQueue: 特殊的BlockingQueue,没有空间的队列，即必须有取的方法阻塞在这里的时候才能放入元素。

3)PriorityBlockingQueue: 支持优先级的阻塞队列 ，存入对象必须实现Comparator接口 （需要注意的是 队列不是在加入元素的时候进行排序，而是取出的时候，根据Comparator来决定优先级最高的）。

#### BlockingQueue<> 队列的作用

BlockingQueue 实现主要用于生产者-使用者队列，BlockingQueue 实现是线程安全的。所有排队方法都可以使用内部锁或其他形式的并发控制来自动达到它们的目的

这是一个生产者-使用者场景的一个用例。注意，BlockingQueue 可以安全地与多个生产者和多个使用者一起使用 此用例来自jdk文档

//这是一个生产者类

class Producer implements Runnable {

private final BlockingQueue queue;

Producer(BlockingQueue q) {

queue = q;

}

public void run() {

try {

while(true) {

queue.put(produce());

}

} catch (InterruptedException ex) {

... handle ...

}

}

Object produce() {

...

}

}

//这是一个消费者类

class Consumer implements Runnable {

private final BlockingQueue queue;

Consumer(BlockingQueue q) { queue = q; }

public void run() {

try {

while(true) {

consume(queue.take());

}

} catch (InterruptedException ex) {

... handle ...

}

}

void consume(Object x) {

...

}

}

//这是实现类

class Setup {

void main() {

//实例一个非阻塞队列

BlockingQueue q = new SomeQueueImplementation();

//将队列传入两个消费者和一个生产者中

Producer p = new Producer(q);

Consumer c1 = new Consumer(q);

Consumer c2 = new Consumer(q);

new Thread(p).start();

new Thread(c1).start();

new Thread(c2).start();

}

}

### 40.ThreadPool的深入考察？

#### 引言

合理利用线程池能够带来三个好处。第一：降低资源消耗。通过重复利用已创建的线程降低线程创建和销毁造成的消耗。第二：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。第三：提高线程的可管理性。线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。但是要做到合理的利用线程池，必须对其原理了如指掌。

#### 线程池的使用

我们可以通过ThreadPoolExecutor来创建一个线程池。

new ThreadPoolExecutor(corePoolSize, maximumPoolSize, keepAliveTime, milliseconds,runnableTaskQueue, handler);

创建一个线程池需要输入几个参数：

* corePoolSize（线程池的基本大小）：当提交一个任务到线程池时，线程池会创建一个线程来执行任务，即使其他空闲的基本线程能够执行新任务也会创建线程，等到需要执行的任务数大于线程池基本大小时就不再创建。如果调用了线程池的prestartAllCoreThreads方法，线程池会提前创建并启动所有基本线程。
* runnableTaskQueue（任务队列）：用于保存等待执行的任务的阻塞队列。 可以选择以下几个阻塞队列。
  + ArrayBlockingQueue：是一个基于数组结构的有界阻塞队列，此队列按 FIFO（先进先出）原则对元素进行排序。
  + LinkedBlockingQueue：一个基于链表结构的阻塞队列，此队列按FIFO （先进先出） 排序元素，吞吐量通常要高于ArrayBlockingQueue。静态工厂方法Executors.newFixedThreadPool()使用了这个队列。
  + SynchronousQueue：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，吞吐量通常要高于LinkedBlockingQueue，静态工厂方法Executors.newCachedThreadPool使用了这个队列。
  + PriorityBlockingQueue：一个具有优先级的无限阻塞队列。
* maximumPoolSize（线程池最大大小）：线程池允许创建的最大线程数。如果队列满了，并且已创建的线程数小于最大线程数，则线程池会再创建新的线程执行任务。值得注意的是如果使用了无界的任务队列这个参数就没什么效果。
* ThreadFactory：用于设置创建线程的工厂，可以通过线程工厂给每个创建出来的线程设置更有意义的名字。
* RejectedExecutionHandler（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处理提交的新任务。这个策略默认情况下是AbortPolicy，表示无法处理新任务时抛出异常。以下是JDK1.5提供的四种策略。
  + AbortPolicy：直接抛出异常。
  + CallerRunsPolicy：只用调用者所在线程来运行任务。
  + DiscardOldestPolicy：丢弃队列里最近的一个任务，并执行当前任务。
  + DiscardPolicy：不处理，丢弃掉。 当然也可以根据应用场景需要来实现RejectedExecutionHandler接口自定义策略。如记录日志或持久化不能处理的任务。
* keepAliveTime（线程活动保持时间）：线程池的工作线程空闲后，保持存活的时间。所以如果任务很多，并且每个任务执行的时间比较短，可以调大这个时间，提高线程的利用率。
* TimeUnit（线程活动保持时间的单位）：可选的单位有天（DAYS），小时（HOURS），分钟（MINUTES），毫秒(MILLISECONDS)，微秒(MICROSECONDS, 千分之一毫秒)和毫微秒(NANOSECONDS, 千分之一微秒)。

#### 向线程池提交任务

我们可以使用execute提交的任务，但是execute方法没有返回值，所以无法判断任务是否被线程池执行成功。通过以下代码可知execute方法输入的任务是一个Runnable类的实例。

threadsPool.execute(new Runnable() {

@Override

public void run() {

// TODO Auto-generated method stub

}

});

我们也可以使用submit 方法来提交任务，它会返回一个future,那么我们可以通过这个future来判断任务是否执行成功，通过future的get方法来获取返回值，get方法会阻塞住直到任务完成，而使用get(long timeout, TimeUnit unit)方法则会阻塞一段时间后立即返回，这时有可能任务没有执行完。

Future<Object> future = executor.submit(harReturnValuetask);

try {

Object s = future.get();

} catch (InterruptedException e) {

// 处理中断异常

} catch (ExecutionException e) {

// 处理无法执行任务异常

} finally {

// 关闭线程池

executor.shutdown();

}

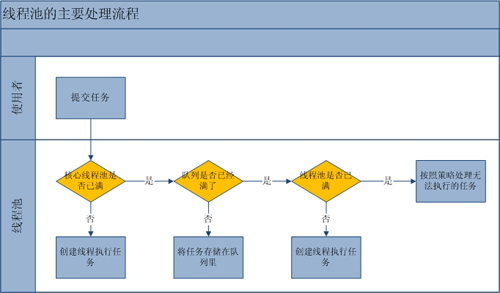
#### 线程池的关闭

我们可以通过调用线程池的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。但是它们存在一定的区别，shutdownNow首先将线程池的状态设置成STOP，然后尝试停止所有的正在执行或暂停任务的线程，并返回等待执行任务的列表，而shutdown只是将线程池的状态设置成SHUTDOWN状态，然后中断所有没有正在执行任务的线程。

只要调用了这两个关闭方法的其中一个，isShutdown方法就会返回true。当所有的任务都已关闭后,才表示线程池关闭成功，这时调用isTerminaed方法会返回true。至于我们应该调用哪一种方法来关闭线程池，应该由提交到线程池的任务特性决定，通常调用shutdown来关闭线程池，如果任务不一定要执行完，则可以调用shutdownNow。

#### 线程池的分析

流程分析：线程池的主要工作流程如下图：



从上图我们可以看出，当提交一个新任务到线程池时，线程池的处理流程如下：

1. 首先线程池判断基本线程池是否已满？没满，创建一个工作线程来执行任务。满了，则进入下个流程。
2. 其次线程池判断工作队列是否已满？没满，则将新提交的任务存储在工作队列里。满了，则进入下个流程。
3. 最后线程池判断整个线程池是否已满？没满，则创建一个新的工作线程来执行任务，满了，则交给饱和策略来处理这个任务。

#### 源码分析

上面的流程分析让我们很直观的了解了线程池的工作原理，让我们再通过源代码来看看是如何实现的。线程池执行任务的方法如下：

public void execute(Runnable command) {

if (command == null)

throw new NullPointerException();

//如果线程数小于基本线程数，则创建线程并执行当前任务

if (poolSize >= corePoolSize || !addIfUnderCorePoolSize(command)) {

//如线程数大于等于基本线程数或线程创建失败，则将当前任务放到工作队列中。

if (runState == RUNNING && workQueue.offer(command)) {

if (runState != RUNNING || poolSize == 0)

ensureQueuedTaskHandled(command);

}

//如果线程池不处于运行中或任务无法放入队列，并且当前线程数量小于最大允许的线程数量，

则创建一个线程执行任务。

else if (!addIfUnderMaximumPoolSize(command))

//抛出RejectedExecutionException异常

reject(command); // is shutdown or saturated

}

}

工作线程。线程池创建线程时，会将线程封装成工作线程Worker，Worker在执行完任务后，还会无限循环获取工作队列里的任务来执行。我们可以从Worker的run方法里看到这点：

public void run() {

try {

Runnable task = firstTask;

firstTask = null;

while (task != null || (task = getTask()) != null) {

runTask(task);

task = null;

}

} finally {

workerDone(this);

}

}

#### 合理的配置线程池

要想合理的配置线程池，就必须首先分析任务特性，可以从以下几个角度来进行分析：

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。

任务性质不同的任务可以用不同规模的线程池分开处理。CPU密集型任务配置尽可能小的线程，如配置Ncpu+1个线程的线程池。IO密集型任务则由于线程并不是一直在执行任务，则配置尽可能多的线程，如2\*Ncpu。混合型的任务，如果可以拆分，则将其拆分成一个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率要高于串行执行的吞吐率，如果这两个任务执行时间相差太大，则没必要进行分解。我们可以通过Runtime.getRuntime().availableProcessors()方法获得当前设备的CPU个数。

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得到执行，需要注意的是如果一直有优先级高的任务提交到队列里，那么优先级低的任务可能永远不能执行。

执行时间不同的任务可以交给不同规模的线程池来处理，或者也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，如果等待的时间越长CPU空闲时间就越长，那么线程数应该设置越大，这样才能更好的利用CPU。

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，可以根据需要设大一点，比如几千。有一次我们组使用的后台任务线程池的队列和线程池全满了，不断的抛出抛弃任务的异常，通过排查发现是数据库出现了问题，导致执行SQL变得非常缓慢，因为后台任务线程池里的任务全是需要向数据库查询和插入数据的，所以导致线程池里的工作线程全部阻塞住，任务积压在线程池里。如果当时我们设置成无界队列，线程池的队列就会越来越多，有可能会撑满内存，导致整个系统不可用，而不只是后台任务出现问题。当然我们的系统所有的任务是用的单独的服务器部署的，而我们使用不同规模的线程池跑不同类型的任务，但是出现这样问题时也会影响到其他任务。

#### 线程池的监控

通过线程池提供的参数进行监控。线程池里有一些属性在监控线程池的时候可以使用

* taskCount：线程池需要执行的任务数量。
* completedTaskCount：线程池在运行过程中已完成的任务数量。小于或等于taskCount。
* largestPoolSize：线程池曾经创建过的最大线程数量。通过这个数据可以知道线程池是否满过。如等于线程池的最大大小，则表示线程池曾经满了。
* getPoolSize:线程池的线程数量。如果线程池不销毁的话，池里的线程不会自动销毁，所以这个大小只增不+getActiveCount：获取活动的线程数。

通过扩展线程池进行监控。通过继承线程池并重写线程池的beforeExecute，afterExecute和terminated方法，我们可以在任务执行前，执行后和线程池关闭前干一些事情。如监控任务的平均执行时间，最大执行时间和最小执行时间等。这几个方法在线程池里是空方法。如：

protected void beforeExecute(Thread t, Runnable r) { }

### 41.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，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。

### 42.同步方法和同步代码块的区别是什么？

同步方法默认用this或者当前类class对象作为锁； 同步代码块可以选择以什么来加锁，比同步方法要更细颗粒度，我们可以选择只同步会发生同步问题的部分代码而不是整个方法； 同步方法使用关键字 synchronized修饰方法，而同步代码块主要是修饰需要进行同步的代码，用 synchronized（object）{代码内容}进行修饰；

### 43.如何确保N个线程可以访问N个资源同时又不导致死锁？

使用多线程的时候，一种非常简单的避免死锁的方式就是：指定获取锁的顺序，并强制线程按照指定的顺序获取锁。因此，如果所有的线程都是以同样的顺序加锁和释放锁，就不会出现死锁了。