Java 高并发

哈喽大家好，我是厨子，这是我整理的面经PDF，并且对其进行了分类整理，应该能够对你有所帮助，阅读过程中，有任何问题大家都可以联系我，进行反馈，我的微信是 chefyuan105，备注你的问题即可。

这是我的两个 Github：PDF内容会实时在这里进行更新，大家可以 star 收藏一下。

https://github.com/chefyuan/interview-base

https://github.com/chefyuan/algorithm-base

另外我还建了一个秋招交流群，大家一起打卡，寒假时，还会有一个每日打卡活动，如果你需要的话，可以联系我，备注秋招即可。

秋招小队简介：http://www.chengxuchu.com/#/Exchange/README?id=%e7%a7%8b%e6%8b%9b%e5%b0%8f%e9%98%9f

入队方式

方式一

扫描下方二维码，关注公众号，点击求职交流，选择社招/校招

方式二

扫码我的二维码，注明来意 校招/社招 + 个人技术栈 + 个人简单介绍，我会尽快同意你的申请，拉你进群。

希望大家都可以拿到满意的 offer 。

1.基础知识

1.0 interrupt()

该函数代表的是，中断状态，当某个线程被其他线程执行interrupt()时，则会给线程一个通知，并将线程的状态变为中断，有人希望你中断啦，怎么做就是你的事啦。是否释放锁由我们自己决定，我们可以通过函数 isinterrupt() 来获得中断状态，

1.1 runnable和 callable 有什么不同?

两者都是调用接口，都需要使用start方法启动。

- callable的核心是 `call()` 方法，允许返回值， `runnable` 的核心是 `run()` 方法，没有返回值
- `call()` 方法可以抛出异常，但是 `run()` 方法不行
- `callable` 和 `runnable` 都可以应用于 `executors` ， `thread` 类只支持 `runnable`

1.2 线程的 run 和 start 有什么区别?

我们创建好线程对象之后，调用run方法，仅仅是调用某个方法，不会创建线程，但是start会开一个新的线程。

另外start只可以启动一次，而run可以启动多次。

1.3 为什么调用 start 而不是 run?

`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， 这时此线程处于就绪（可运行）状态，并没有运行，一旦得到cpu时间片，就开始执行`run()`方法 ，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，此线程随即终止。

`run()` 方法只是类的一个普通方法而已，如果直接调用 `run` 方法，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，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，这样就没有达到写线程的目的。

调用 `start()` 方法可以开启一个线程，而 `run()` 方法只是thread类中的一个普通方法，直接调用 `run()` 方法还是在主线程中执行的。

2.线程

2.0 多线程的优缺点

优点：当一个线程进入等待状态或者阻塞时，CPU可以先去执行其他线程，提高CPU的利用率。

缺点：

资源限制：因为计算机资源有限，所以有可能让我们想要并发运行的部分，依旧是串行，这样就程序就会执行很慢，因为程序并发需要上下文切换和资源调度。

上下文切换：因为有可能造成上下文切换。

死锁：有可能因为竞争出现死锁

2.1 线程的上下文切换

因为我们线程是调度的基本单位，但是线程共用进程的虚拟地址空间，所以线程调度速度更快。

单核的CPU也是支持多线程，多是通过分配时间片来进行线程的调度，时间片是每个线程的执行时间，所以CPU需要不断的切换线程。

CPU会通过时间片分配算法来循环执行任务，当前任务执行完一个时间片后会切换到下一个任务，但切换前会保存上一个任务的状态，因为下次切换回这个任务时还要加载这个任务的状态继续执行，从任务保存到再加载的过程就是一次上下文切换。

2.2 守护线程和用户线程的区别?

我们可以理解成守护线程是服务与用户线程的，保证用户线程的平稳运行，我们可以通过SetDaemon来设置成守护线程，不过守护线程要设置成 start（）之前。不然会抛出异常，比如我们的垃圾回收线程就是用户线程。

用户线程：平常使用到的线程均为用户线程

守护线程：垃圾回收线程

主要区别就是JVM虚拟机是否存活

用户线程：当任何一个用户线程未结束，Java虚拟机是不会结束的。

守护线程：如何只剩守护线程未结束，Java虚拟机结束。

2.3 死锁，活锁和饥饿有什么区别

死锁是因为线程竞争资源而产生的，请求和保持，互斥，不可剥夺，等待。

活锁：线程间互相礼让，同时让其他线程先获取，然后造成都无法获取。

`死锁和活锁的区别：`

- 活锁是在不断地尝试、死锁是在一直等待。
- 活锁有可能自行解开、死锁无法自行解开

`饥饿：`

饥饿就是某些线程一直得不到资源，进行运行，比如我们的使用的短作业优先的调度方式的话，则有可能造成长作业一直得不到运行，如果一直有短作业来的话。

另外优先级调度的情况，也有可能造成饥饿的情况。

`产生饥饿的原因：`

- 高优先级的线程占用了低优先级线程的CPU时间
- 线程被永久堵塞在一个等待进入同步块的状态，因为其他线程总是能在它之前持续地对该同步块进行访问。
- 线程在等待一个本身也处于永久等待完成的对象(比如调用这个对象的 `wait()` 方法)，因为其他线程总是被持续地获得唤醒。

2.4 线程同步和线程调度相关的方法有哪些?

`wait方法` :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方法：使一个线程暂停执行，但是不会释放锁。

notify()：唤醒，wait线程，但是不确定是哪一个，由JVM控制。

notifyall()：唤醒所有线程

interrupt()：发送中断命令，告知其有人希望你中断执行，具体如何执行还是要看你自己。我们可以根据中断标识位进行修改。

join()： 与 `sleep()` 方法一样，是一个可中断的方法，在一个线程中调用另一个线程的 `join()` 方法，会使得当前的线程挂起，知直到执行 `join()` 方法的线程结束。例如在B线程中调用A线程的 `join()` 方法，B线程进入阻塞状态，直到A线程结束或者到达指定的时间。

yield()：提醒调度器，该线程愿意放弃CPU资源，从运行态，到就绪态。

2.5 sleep和yield的异同

sleep仅仅是暂停住，暂停指定的时间，并且不会释放锁，没有消耗时间片。

`yield()` 只是对CPU进行提示，如果CPU没有忽略这个提示，会使得线程上下文的切换，进入到就绪状态。

sleep一定会执行成功暂停指定的时间，yeild不一定。

2.6 sleep 和 wait 的异同

相同点：

sleep和wait都会发生状态的转化，转换成阻塞状态。

不同点：

wait()是Object方法，sleep是Thread方法

sleep不会释放锁，而wait会释放锁

sleep到相应时间之后，会退出堵塞，但是wait需要其他线程唤醒才可以。

sleep可以通过interrupt进行停止。

2.7 如何唤醒一个阻塞的线程

我们需要具体情况，具体分析

`wait:` 我们可以使用 notify 和 notifyall 来进行唤醒

`sleep():` 调用该方法，会使其在指定状态进行进入阻塞状态，等到指定时间过去，线程再次获取到 CPU 时间片而被再次唤醒。

`join()` ： 当前线程A调用另一个线程B的 join() 方法，当线程B运行完毕之后，线程则会从阻塞转为可执行状态。

`yield()`: 使得当前线程放弃CPU时间片，但随时可能再次得到时间片而激活。

2.8 如何实现两个线程之间的通信和协作?

我们可以使用wait的 `notify / notifyAll` 来进行。

ReentrantLock类加锁的线程的 `Condition类的await和 signal/signAll()` 来实现

2.9 什么是线程同步，什么是线程互斥，他们如何实现。

我们可以通过wait 和 notify来实现

volatile实现同步

同步方法

整体含义就是让保证线程间的通信。

2.10 在Java中有那些方法可以保证线程安全

线程安全主要是，原子性，可见性，有序性

原子性：Atomic、synchronized、Lock来实现。

可见性：volatile、synhronized、Lock

有序性：Happens-Before

推荐阅读：https://blog.csdn.net/qq_32273417/article/details/109148693

3.几种关键字

Volatile

3.0 并发编程的三个概念

原子性：要干就干完，要不干就不干

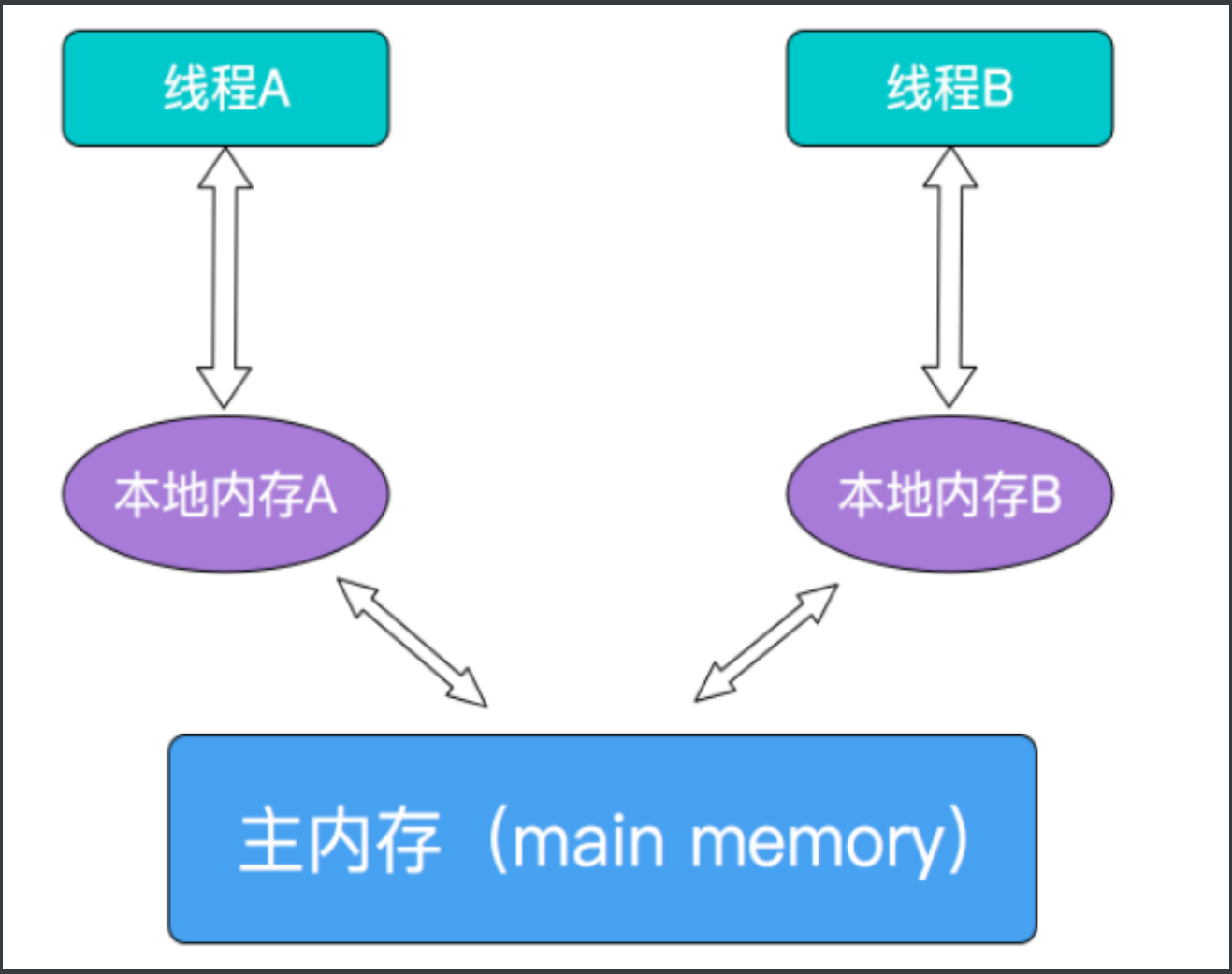
有序性：即程序执行的顺序按照代码的先后顺序执行。

Java内存模型中的有序性可以总结为：`如果在本线程内观察，所有操作都是有序的；如果在一个线程中观察另一个线程，所有操作都是无序的`

可见性：指当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。

3.1 共享变量的可见性

JMM决定一个线程对共享变量的写入何时对另一个线程可见，JMM定义了线程和主内存之间的抽象关系：共享变量存储在主内存(Main Memory)中，每个线程都有一个私有的本地内存（Local Memory），本地内存保存了被该线程使用到的主内存的副本拷贝，线程对变量的所有操作都必须在工作内存中进行，而不能直接读写主内存中的变量。



对于普通的共享变量来讲，线程A将其修改为某个值发生在线程A的本地内存中，此时还未同步到主内存中去；而线程B已经缓存了该变量的旧值，所以就导致了共享变量值的不一致。解决这种共享变量在多线程模型中的不可见性问题，较粗暴的方式自然就是加锁，但是此处使用synchronized或者Lock这些方式太重量级了，比较合理的方式其实就是volatile。

3.2 volatile的特性

volatile 保证可见性，有序性，但是不能保证原子性

可见性

volatile 关键字会将修改后的变量强制刷新到主内存中

然后使其他本地内存中的共享变量拷贝副本失效。其他线程需要获取该值时，则只能从主内存中获取，进而保证了可见性。

有序性

重排序是为了优化程序的性能，但是重排序后不会影响程序的执行结果

重排序的准则是这样。

（1）重排序操作不会对存在数据依赖关系的操作进行重排序。

a=1;b=a; 这个指令序列，由于第二个操作依赖于第一个操作，所以第二个应该发生在第一个的后面。

（2）重排序是为了优化性能，但是不管怎么重排序，单线程下程序的执行结果不能被改变

比如：a=1;b=2;c=a+b这三个操作，`第一步（a=1)和第二步(b=2)由于不存在数据依赖关系， 所以可能会发生重排序`，但是c=a+b这个操作是不会被重排序的，因为需要保证最终的结果一定是c=a+b=3。

单线程下如果使用如果在满足两个原则的情况下进行重排序是没有问题的，但是在多线程的情况下则会出现问题。

`使用volatile关键字修饰共享变量，便可以禁止这种重排序，修饰共享变量时，则会在指令序列中插入内存屏障来禁止特定类型的重排序`。主要是根据以下原则。

a.执行到 volatile 变量的读操作或者写操作时，在其前面的操作肯定全部已经进行，并且对后面的操作可见，在其后面的操作肯定还没有进行

b.不能将 volatile 前面的语句放到后面执行，也不能将后面的放到前面执行。

也就是执行到volatile时，前面所有的语句都执行完了，后面的所有语句都未执行，且前面的结果对后面可见。

3.3 volatile原理

volatile可以保证线程可见性且提供了一定的有序性，但是无法保证原子性。在JVM底层volatile是采用“内存屏障”来实现的。观察加入volatile关键字和没有加入volatile关键字时所生成的汇编代码发现，加入volatile关键字时，会多出一个lock前缀指令，lock前缀指令实际上相当于一个内存屏障（也成内存栅栏），内存屏障会提供3个功能：

（1）它确保指令重排序时不会把其后面的指令排到内存屏障之前的位置，也不会把前面的指令排到内存屏障的后面；即在执行到内存屏障这句指令时，在它前面的操作已经全部完成；

（2）它会强制将对缓存的修改操作立即写入主存；

（3）如果是写操作，它会导致其他CPU中对应的缓存行无效。

推荐阅读：https://www.cnblogs.com/dolphin0520/p/3920373.html

3.4 为什么volatile不支持原子性

1.线程读取i

2.temp = i + 1

3.i = temp 当 i=5 的时候A,B两个线程同时读入了 i 的值， 然后A线程执行了 temp = i + 1的操作， 要注意，此时的 i 的值还没有变化，然后B线程也执行了 temp = i + 1的操作，注意，此时A，B两个线程保存的 i 的值都是5，temp 的值都是6， 然后A线程执行了 i = temp （6）的操作，此时i的值会立即刷新到主存并通知其他线程保存的 i 值失效， 此时B线程需要重新读取 i 的值那么此时B线程保存的 i 就是6，同时B线程保存的 temp 还仍然是6， 然后B线程执行 i=temp （6），所以导致了计算结果比预期少了1

3.5 Happen-Before规则

- 一个程序内保证语义的串行性
- volatile 变量的写先于读发生
- 加锁先于解锁
- 传递性
- Start()第一个执行
- 终结最后发生
- 中断先于被中断

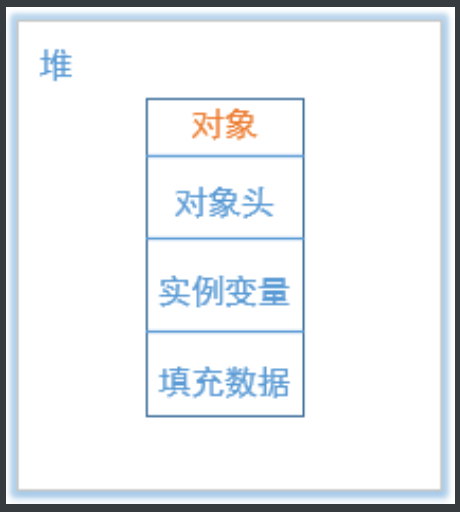
synchronized

3.6 synchronized 底层原理

几种用法

- 修饰类
- 修饰实例方法
- 修饰代码块

锁是加在对象上的，无论是类对象还是实例对象，他们都是由对象头，实例变量，填充数据三部分组成



synchronized使用的锁对象是存储在Java对象头里的，其主要结构是由Mark Word 和 Class Metadata Address 组成，然后Mark Word 的话存储着对象的信息，HashCode、分代年龄、锁标记等。

Mark Word 的存储状态

锁状态	25bit	4bit	1bit是否是偏向锁	2bit 锁标志位
无锁状态	对象HashCode	对象分代年龄	0	01

Mark Word 是一个非固定的数据结构，这是为了存储更多的数据，然后他会根据对象的状态来复用自己的空间。

<table>
<tr><td rowspan="2">锁状态</td><td colspan="2">25 bit</td><td rowspan="2">4bit</td><td>1bit</td><td>2bit</td></tr>
<tr><td>23bit</td><td>2bit</td><td>是否是偏向锁</td><td>锁标志位</td></tr>
<tr><td>轻量级锁</td><td colspan="3">指向栈中锁记录的指针</td><td colspan="2">00</td></tr>
<tr><td>重量级锁</td><td colspan="3">指向互斥量（重量级锁）的指针</td><td colspan="2">10</td></tr>
<tr><td>GC标记</td><td colspan="3">空</td><td colspan="2">11</td></tr>
<tr><td>偏向锁</td><td>线程ID</td><td>Epoch</td><td>对象分代年龄</td><td>1</td><td>01</td></tr>
</table>

synchronized 是一个重量级锁，所以其标志位是 10，每个对象都与一个 monitor 相连，然后他可以随着对象创建或销毁，然后也可以随着对象加锁后而被创建。

monitor是由，monitorObject 实现的，然后他里面的几个重要的值我们来进行解释一下

- count用来记录线程进入加锁代码的次数，
- owner记录当前持有锁的线程,即持有ObjectMonitor对象的线程。（目前被谁加锁了）
- EntryList是想要持有锁的线程的集合。（还有谁加锁了）
- WaitSet 是加锁对象调用wait（）方法后，等待被唤醒的线程的集合。（谁加了之后，释放了，然后等待被唤醒）

我个人是这样理解的，这个 monitor 就相当于是对象的管家，用来记录该对象被谁用了，然后后面还有谁想用。

另外需要注意的是 wait、notify、notifyAll 是synchronized 专有的，synchronized是可重入锁，再次获取锁时不会生阻塞，这也就是那个 count ++ 的原因

推荐阅读：https://zhuanlan.zhihu.com/p/377423211

3.7 Synchronized 锁升级

无锁状态

偏向锁： 因为大部分情况是不存在锁竞争的，大部分还是同一线程获取同一个锁，因此如果每次都要竞争锁会增大很多没有必要付出的代价，为了降低获取锁的代价，才引入的偏向锁。

偏向锁升级，某个线程获取对象的锁的时候，先查看对象的 threadid 是否和自己的id 相同，相同情况下说明自己拥有该锁，则无需重新通过CAS加锁，如果不一致，则查看锁住对象的那个线程是否还存活，如果不存活，则将对象设置成可竞争状态，其他线程可以来竞争锁。如果不存活，则查看该线程的祯栈来观察该线程是否还需要继续持有锁，如果不需要则，当前线程获取该锁，如果需要继续持有则 升级成轻量级锁 。

轻量级锁： 轻量级锁适用于请求锁的线程不多，然后持有锁时间不长的情况，因为线程等待需要从，用户态转换到内核态，如果出现刚调入内核态等待，然后锁就释放了，这一来一回造成的开销是很大的，所以就让线程进行自旋，等待锁的释放。

线程1获取轻量级锁时会先把锁对象的**对象头MarkWord复制一份到线程1的栈帧中创建的用于存储锁记录的空间**（称为DisplacedMarkWord），然后**使用CAS把对象头中的内容替换为线程1存储的锁记录（DisplacedMarkWord**）的地址，如果此时有别的线程也来获取锁的话，发现已经被线程 1 给更换了，那么则修改失败，自旋等待线程 1 释放锁，等待若干时间之后，发现还没有释放锁，则升级为重量级锁。

重量级锁： 加锁后，如果其他线程想要获取锁，则会被堵塞。

注意锁不能降级，但是偏向锁可以被重置为无锁，

锁状态	优点	缺点	适用场景
偏向锁	加锁解锁无需额外的消耗，和非同步方法时间相差纳秒级别	如果竞争的线程多，那么会带来额外的锁撤销的消耗	基本没有线程竞争锁的同步场景
轻量级锁	竞争的线程不会阻塞，使用自旋，提高程序响应速度	如果一直不能获取锁，长时间的自旋会造成CPU消耗	适用于少量线程竞争锁对象，且线程持有锁的时间不长，追求响应速度的场景
重量级锁	线程竞争不适用CPU自旋，不会导致CPU空转消耗CPU资源	线程阻塞，响应时间长	很多线程竞争锁，且锁持有的时间长，追求吞吐量的场景

锁粗化

就是将多个频繁加锁解锁的小东西合成一个大东西，给锁起来，好比很多需要加锁的小箱子，给放到大箱子里进行加锁

`锁消除`

去除不可能存在共享资源竞争的锁

3.8 ReentrantLock底层原理

重入锁可以完全替代synchronized ， 我们将其理解为锁的一种即可，然后可重入锁的一些特性，我们来对其进行改造，我们知道 synchronized 需要我们手动加锁，手动解锁。

另外当synchronized 需要请求锁的时候，有两种可能，要么获得该锁，要么继续等待。但是可重入锁有第三种情况，`中断等待，或者到时放弃等待，trylock()。`

与可重入锁配合的关键字 await 和 signal(),一个进入等待状态，并释放锁，signal 将其唤醒。

底层原理就是 CAS 和 AQS

假设我们一个线程获取锁，发现该锁已经被其他线程占用，则会wait，加入到AQS中，

然后我们再说一下公平锁和非公平锁，

1. 非公平锁，如果有其他线程尝试lock()，有可能被其他刚好申请锁的线程抢占。
2. 公平锁，只有在CLH队列头的线程才可以获取锁，新来的线程只能插入到队尾。

默认是非公平锁，因为这样可以避免维护一个有序队列。

通过大量CAS保证线程安全

如果通过CAS修改了值，则说明该线程获得了锁，然后一个线程也可以重复加锁，每加一次执行 state++（也就是可重入）。如果没有获得锁，则会加入AQS队列，通过自旋和CAS保证能够进入队列。写入队列之后，则需要挂起当前线程。

释放锁时 state—，直至为0之后，才释放该锁。

我们可以通过这个例子来进行解释，村里有一个口井，然后每个人都想在井里打水，但是这个井每次只能有一个打水，所以如果该井被使用的话，新的则需到队伍后面排队（公平锁），然后如果这个人打完了水（释放锁），有的人想利用换人的间隙，去抢打水的位置，（非公平锁），另外就是，如果某个人想连续打为 5 桶水，则需要占用 5 次打水的位置，当他打完 5 次之后，别人才能打（可重入）。

推荐阅读：https://www.cnblogs.com/heqiyoujing/p/11145146.html

3.9 可重入锁 和 synchronized 的区别。

- 可重入锁可以放弃等待，可以中断等待锁
- synchronized 是基于JVM的，由操作系统实现，可重入锁是jdk实现
- 可重入锁更加灵活，粒度小一些
- 可重入锁可以通过Condition 来实现分组唤醒，不像syn 要么唤醒一个，要么全部唤醒

3.10 锁优化

`无锁状态`

`偏向锁`

因为经过HotSpot的作者大量的研究发现，大多数时候是不存在锁竞争的，常常是一个线程多次获得同一个锁，因此如果每次都要竞争锁会增大很多没有必要付出的代价，为了降低获取锁的代价，才引入的偏向锁

`轻量级锁`

`重量级锁`

无锁

CAS就是无锁状态，乐观锁，默认是没有人占用我们的资源，通过比较来修改值，看起来比较复杂，但是这样不会出现死锁。无锁情况都是基于CAS原理

AtomicInteger

实现数字的原子性+/-，和 Redis 里的 INCR/DECR 一样，但是会出现ABA问题，因为比如我们给他 20 块钱，他又给花完了，让我们误以为还没有给他钱。

时间戳解决ABA问题

`AtomicStampedReference` 通过时间戳来解决ABA问题，出现ABA问题的原因就是，对象在修改过程中丢失了状态信息，所以我们可以引入时间戳的，对象值和时间戳都必须要满足期望值，写入才会成功。

无锁数组

AtomicIntegerArray数组无锁

无锁的概念是通过 Atomic 包来进行的，主要通过CAS

减少锁持有的时间

尽可能减少持有锁的时间，等完全思考好之后再进行写，

减少锁粒度

4.经典常考题目

4.0 介绍一下CAS

CAS是一种无锁算法。有3个操作数：内存值V（也就是要更新的变量）、旧的预期值A、要修改的新值B。当且仅当预期值A和内存值V相同时，将内存值V修改为B，否则什么都不做。

4.1 CAS的底层原理

CAS 全称为比较和交换，然后他的核心类是unsafe类，因为Java无法访问底层操作系统，所以借助unsafe来帮我们访问，相当于开了一个后门，他里面的方法可以直接操控内存，

```
public final native boolean compareAndSwapInt(Object o, long offset,
                                              int expected,
                                              int x);
```

CAS全称为Compare-And-Swap，它是一条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，如果是则更改为新的值，因为原语必须是连续的，这个过程是原子的，不会造成数据不一致的情况。

4.2 wait和sleep关键字的不同

sleep使线程进行暂停一段时间，并不会释放锁，当时间到了之后，则会自动继续执行。

另外wait关键字也是使线程暂停，然后他会释放锁，需要其他线程使用 notify进行唤醒。

4.3 为什么wait()、notify()、notifyAll()被定义在Object类中而不是在Thread类中?

因为这些方法在操作同步线程时，都必须要标识他们操作线程的锁，只有同一个锁上的被等待线程，可以被同一个锁上的 `notify()` 或 `notifyAll()` 唤醒，不可以对不同锁中的线程进行唤醒，也就是说等待和唤醒必须是同一锁。而锁可以是任意对象，所以可以被任意对象调用的方法是定义在 `Object` 类中。

如果把 `wait()` 、 `notify()` 、 `notifyAll()` 定义在Thread类中，则会出现一些难以解决的问题，例如如何让一个线程可以持有多把锁？如何确定线程等待的是哪把锁？既然是当前线程去等待某个对象的锁，则应通过操作对象来实现而不是操作线程，而Object类是所有对象的父类，所以将这三种方法定义在Object类中最合适

4.4 线程安全的类有哪些?

通过synchronized 关键字给方法加上内置锁来实现线程安全

如：Timer，TimerTask，Vector，Stack，HashTable，StringBuffer

原子类Atomicxxx 包装类的线程安全类

如AtomicLong，AtomicInteger等等

ConcurrentHashMap，和 ConcurrentSkipListSet和ConcurrentSkipListMap等等。

推荐阅读：https://blog.csdn.net/dgeek/article/details/70145555

4.5 线程安全容器

HashTable，Vector，ConcurrentHashMap 等