# 什么是死锁，发生原因是什么，如何解决和避免产生死锁？

死锁是两个或者以上的线程，由于资源竞争以及彼此之间的通信造成的一种阻塞现象。通常产生死锁有四个必要条件：

1. 互斥使用，一个资源只能由一个线程占用。
2. 不可抢占，资源请求的一方只能等待资源拥有的一样主动释放才能使用。
3. 请求和保持，请求方请求其他资源的同时持有自己的资源。
4. 循环等待，多个线程拥有的资源形成了一个环，比如P1拥有P2的资源，P2拥有P1的资源。

死锁预防：只要破坏其中一个条件就能够打破死锁了，预防死锁的发生。

首先是互斥使用，我们不能改变资源是否互斥，这是由资源的特性所决定的，但是我们能够减少互斥，减少竞争。比如降低锁的粒度，另外还要尽量避免线程同时获取几个锁。

然后不可抢占，当进程发生死锁的时候，我们可以直接中断当前的进程，强制其释放资源，比如当线程被wait,join方法阻塞的时候，我们可以使用interrupt方法中断它，使其退出阻塞的状态，从而可能打破死锁。

然后是请求和保持，如果我们请求不到想要的资源时，就不要保持自己由的资源了，比如tryLock,wait,notify方法等都有超时返回，如果超过一定时间就放弃已经拥有的锁。

最后是循环产生死锁只能我们在设计的时候尽量避免了，比如尽量以固定的顺序进行。

**死锁避免：** 进程在每次申请资源时判断这些操作是否安全。（银行家算法）

**死锁检测：** 判断系统是否属于死锁的状态，如果是，则执行死锁解除策略。

**死锁解除：** 将某进程所占资源进行强制回收，然后分配给其他进程。（与死锁检测结合使用的）

# 什么是原子操作，Java中的原子操作是什么？

原子操作不能被中断的一系列操作。一个原子操作在中途是不会被其他线程打断的。原子操作不需要同步，因为同步的目的就是防止被打断，而原子操作已经不会被打断了，所以没有必要同步了。

i++不是原子操作，同时也是线程不安全的。i++分为了3步，定义一个临时变量等于当前变量，临时变量+1，赋值回当前变量。由于CPU的运算速度是很快的，在赋值回之前，可能又有其他线程改变了i的值，那么此时赋值回去，i的值就不正确了。

# Java中的volatile关键字是什么作用？在Java中它跟synchronized方法有什么不同？

**vaolatile**关键字保证了所有线程对变量访问的可见性和有序性，但是不保证原子性。

有序性：我们编程中的顺序，到最后执行就不一定是这个顺序了，因为编译器或处理器出于优化考虑，可能会对指令进行重排序，在单线程中指令的重排序不会对程序结果造成影响，但是在多线程中可能就会造成线程不安全，比如单例模式中，new对象的原本顺序是分配内存，初始化，指向分配的内存，但是重排后，可能是分配内存，指向分配后的内存，最后是初始化，那么就可能出现其他的线程拿到没有初始化的对象。因此，volatile关键字修饰的变量能够通过内存屏障来禁止编译器和处理器的重排序，当然禁止不是完全不让他重排序，不让优化，而是有一定的规则，对可能导致线程不安全的重排序禁止。

可见性：处理器为了效率，在多线程的情况下不会直接到主存里面去读取共享变量，而是每个线程都有本地的内存，在本地的内存中有共享内存的缓存，读的话就读取本地缓存的值，这样读取的速度就会快很多。而写的时候，也是先写本地变量，更新成功后再写到主内存中去，那么就可能会导致，在本地内存中更新成功了，还没及时刷回到主存中去，另外一个线程从没有更新的主存中读取了变量，因此这里也可见性也就是说多线程中见不到其他线程的更新，线程之间是隔离的，线程之间不可见。那么volatile是怎么解决的呢？首先是如果光是读变量肯定不会有不一致的情况，主要是写变量。volatil在写的时候首先会**锁总线或者锁缓存**，从使其他线程中的缓存无效，并通过消息总线通知他们从主存中去读内存，相当于写的期间，加了锁使其他的线程不能够读，然后写完之后让其他的线程马上更新，从而保证数据的可见性。 。

# synchronized关键字

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。因此能够保证程序的原子性和可见性，但是不能够保证有序性。

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。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。

**sync锁方法：**

普通方法：锁实例对象,可以访问同个对象中非sync方法。不同对象之间无影响

**synchronized** **public  void  doWork(){**

需要同步操作的代码

**}**

**静态方法：锁类对象**

**synchronized** **public  static void  doWork(){**需要同步操作的代码**}**

sync锁同步方法块：锁的括号里面的对象

**synchronized**  **public static (this){**

**}**

synchronized 关键字底层原理属于 JVM 层面。

首先我们知道每个对象都有对象头，然后锁的实现都是靠对象头中的监视器monitor实现的，只有拿到了对象的monitor，才能对对象进行操作，否则就进入该对象monitor中的等待队列。

synchronized 当作用在同步语句块的情况：synchronized 同步语句块的实现使用的是 monitorenter 和 monitorexit 指令，其中 monitorenter，当到达monitorenter语句时，会尝试获取对象头的monitor，如果获取不成功就进入等待队列，线程阻塞。如果成功就获取到锁，执行代码，然后到monitorexit处释放锁。另外monitor锁也是可重入的，内部维护这一个计数器，重入一次计数器+1，退出一次-1,到0的时候再释放，和AQS框架是类似的。

synchronized 修饰方法的的情况：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。实现也是与锁同步块类似，使用monitorenter和monitereixt来实现的，

# java synchronized锁升级过程

synchronized在jdk1.6之前，一直都是重量级锁，通过monitorEnter和monitorExit来控制的是锁的资源是通过操作系统去申请 ，所以比较重量级。

在jdk1.6之后，jvm对synchronized进行了升级，对锁的粒度进行了细化，分为无锁，偏向锁，轻量级锁，重量级锁。性能得到了很大的提升。

在没有进入sync代码块之前，都是无锁的状态，当进入代码块之后，默认升级为偏向锁。如果存在资源的竞争，那么会撤销偏向锁(不是马上撤销，需要在全局安全点，比较影响性能)，然后由偏向锁升级为轻量级锁。如果竞争比较激烈，多次CAS操作也不能获取到锁，就会升级为重量级锁。

由无锁升级为偏向锁，然后利用CAS进行资源获取，如果CAS到一定次数后还没有获取到资源，即竞争紧张，会升级为轻量级锁，如果并发上来，升级为重量级锁，

# 说说 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。

# CAS 的缺点是什么，怎么解决 ABA 问题

CAS，它的全称是Compare And Swap，比较并交换，它的功能是判断当前值和预期值是否相等，如果相更改为新的值，这个过程是原子的。CAS的优点就是他是在用户态进行操作的，并不需要加锁也能够保证原子性，减少了锁资源申请的开销。缺点就是cas操作如果竞争激烈的话，会产生无意义的原地自旋操作，浪费cpu的资源。

ABA问题就是线程1记录了预期值为a，然后线程2对数据进行了更改，由a变为b，然后又改为a,这时候线程1进行cas操作的时候发现当前值为a,预期值为a,可以进行操作。但是实际上a已经发生了改变，并不能进行操作。这就是aba问题。

解决aba问题，可以对数据加一个版本号或者时间戳，每次更改就将版本号+1或者更新时间戳，从而解决aba的问题，这种方式和数据库种的乐观锁的思想是一样的。

# 什么是不可变对象final，它对写并发应用有什么帮助？

不可变对象就是一旦被创建之后，状态就不可以被改变了。

对于修饰变量，必须在一开始就赋值或者在构造函数中赋值，且赋值之后就不可改变，只能一次赋值。

对于修方法，方法不能被重写。对于修饰类，类不能被继承。

final对象通过禁止指令重排，保证能够读取到正确的初始化好的值。而普通变量在多线程的情况下可能会读取到没有初始化的值。由于final修饰的变量只能一次赋值，后面就只能读了。因此它本身就是线程安全的，而不需要额外的锁机制，从而减少了应用的开销。并且final对象是存储在方法区的，可以重复使用，比如String类就是，而不需要重复创建。

# ****线程优先级****

**java中通过setPriority()来设置线程优先级。默认为5。优先级高的线程会分配到更多的时间片资源，从而获取到更多的执行时间。但是java的优先级在操作系统中不一定会生效。因此不能依靠优先级来保证程序的正确性。因此一般情况下不要设置程序的优先级。**

# 为什么要使用多线程

1. 更多的处理器核心

随着时代的发展，现在的处理器通常拥有着多核，使用多线程也是对资源的不浪费，一个单线程的程序在运行时只能使用一个处理器核心，那么再多的核心，比如现在常有8核，16核的处理器和单核的运行速度是没有任何区别的。

1. 更快的运行速度

通常来说程序在数据一致性不是很强的情况下，使用多线程的技术能够明显的减少相应时间，让用户能够有更好的体验。

1. 即使在单核处理器的情况下，**虽然多线程的程序由于频繁的上下文切换回导致运行速度还不如单线程**，但是多线程的模型有时候能使系统更加的健壮。比如单线程如果阻塞了，比如说读取文件，那么可能会使整个系统崩溃，但是多线程就能够预防这种情况，并且多线程情况下也不会影响其他任务的执行。
2. 更好的建立编程模型

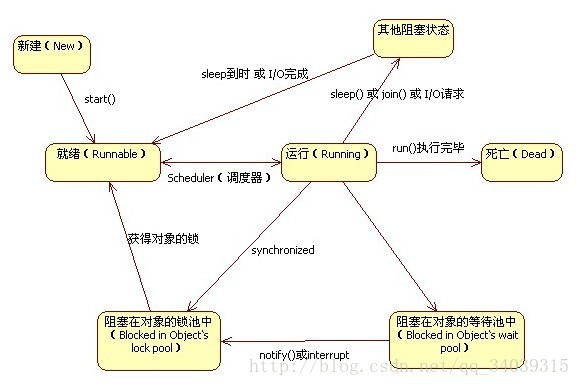
使用多线程可以把一个很大的任务A,分解成小的任务B,C,D这样可能回事我们的模型更加简单。

# 并行和并发

**并行**是在一个时间点上，多个进程同时运行，这要求有多个CPU，因为一个CPU在统一时间点只能有一个进程运行。

**并发**是指在一段时间内，多个程序同时运行，但是这并不是真正意义上的同时运行，只不过CPU通过时间片分时调度让他们看起来同时运行。

# 线程间的几种状态

1. 新建状态（NEW）： 用**new语句**的线程处于新建状态，和其他的对象一样，仅在堆内分配了内存。
2. 臀型状态(RUNNABLE)：java线程将操作系统中的就绪和运行都笼统地称作’运行中’
3. 阻塞状态(BLOCKED)：线程没有获取到锁，被阻塞。
4. 等待状态（WAITING）：因为线程因为某些原因放弃了CPU，暂停了运行，主要原因有3种：
   1. 调用了某个对象的wait()方法，就会持有该对象的锁，然后进入该对象的等待池中，直到满足了某个条件，比如读取文件完毕。
   2. 其他阻塞状态：比如该线程调用了sleep方法，或者调用了其他线程的join方法，实际也是wait()方法，还有就是I/0请求，都会阻塞当前线程。
5. 超时等待(TIME\_WAITING)：超时等待状态，和等待状态不同地是，超时后会返回
6. 死亡/终止状态(DEAD/TERMNATED):线程的run方法执行完毕了，从run方法中退出，自此生命周期就完了

# ****中断interrupt，interrupted与isInterrupted方法的区别? 如何停止一个正在运行的线程.****

中断是线程地一个标志位。表示线程是否被其他线程中断。中断只是对线程的一个通知，中断标志改变后线程可以响应，也可以不响应。

intertupt:对线程进行中断操作。

interupted:对线程中断标志位进行复位。

isInterupted:判断当前线程是否被中断，如果被中断，可以对中断进行响应。

对于正常运行的线程，完全可以不理会中断操作，也可以通过isInterupted对中断进行响应。

对于join,wait等阻塞方法，当调用interrupt之后，线程会被中断，并且抛出异常，因为他们响应了中断。

sync锁不可以中断，lock也不可中断，tryLock和LockInterruptibly()可以响应中断。

# ****等待/通知机制****

wait()/wait(long):调用wait()方法的线程进入WAITINT状态，知道其他线程通知它或者中断才返回，并且会释放掉持有该对象的锁。就是某些条件没有完成，先把锁释放掉，让其他线程先完成任务，完成后对它进行通知，然后再继续运行下去。

notify()/notifyAll():对等待该对象的线程进行通知。

范式：

synchronized（对象）{

while(条件不满足){

对象.wait()

}

其他代码

}

# 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？（Join）

通过thread.jion()方法，这个方法的作用是阻塞调用此方法的线程，比如主线程调用t1.join(),那么调用之后必须等待t1执行完毕，主线程才会继续执行，那么只要把t2.start放在t1.join之后，t3.start放在t1.join之后，就能够保证了。他原理就是通过Object.wait()方法。主线程调用t1.join实际上是调用了t1.wait，调用了wait方法的线程（主线程）就会进入WAITING状态，直到t1线程执行完毕后主线程再继续执行，关键是通过两段代码，while(this.isAlive()){this.wait()},即t1还存活，没有执行完毕，主线程就一直等待。最后是通过在jvm源码中会notifyAll,从而让主线程继续跑下去。

# ThreadLocal

ThreadLocal即线程变量，ThreadLocal修饰的对象会为使用该变量的线程提供独立的变量副本。由于变量在线程之间是隔离的，互相不影响，从而保证了线程安全。

ThreadLocal和线程同步机制相比：他们的用处不同，同步机制是为了解决线程之间共享变量线程安全的问题。ThreadLoacl是为了实现线程之间的数据隔离。

实现原理：ThreadLocalMap,其中key为每个线程的ThreadLoac,value为每个线程独立的值。从而实现了每个线程都有它独立的值。

应用场景：数据的作用域是线程并且不同的线程是有不同的数据副本的时候。

# 内存泄漏

内存泄漏指开辟的空间在使用完毕之后一致没有释放，导致一致占用着该内存，造成了内存的浪费。

ThreadLocal的key为弱引用，就是为了解决内存泄漏的问题。但是为弱引用的时候，还是会存在内存泄漏，当gc的时候，因为key是弱引用，key被垃圾回收了，但是value还没有被回收掉。从而造成key为null，的键值对。造成内存泄漏。虽然map中也回对key为Null的进行检测回收，但是不是每时每刻的。因此最好的是如果不适用当前的对象了，就直接remove掉。

# ****线程池？Java 线程池有哪些参数？阻塞队列有几种？拒绝策略有几种？线程池的工作机制？（非大厂会问：有哪些线程池）****

在没有线程池的时候，在使用多线程每次新建任务都需要去创建新的线程，这样频繁的创建和销毁线程，会造成系统频繁的上下文切换，无故地增加了系统地负担，造成了资源的浪费。

而线程池就是为了解决这一个问题的。线程池它预先创建了若干个线程，当需要用到线程的时候，不需要自己去创建，而是使用线程池早已经创建好的线程。从而提高线程的复用，减少了线程创建和销毁的开销，提高了系统的效率。

线程池有以下几个重要的参数：

corePoolSize:核心的线程数，就是线程池中至少有这么多个线程。

maxPoolSize:最多创建的线程数，主要是为了应对在并发很高的时候，可以多创建几个线程来应对。

keepAliveTime:线程空余时间。就是针对非核心线程的，如果空闲时间达到了，说明当前并发量并不高了，那么就对非核心线程进行回收。

workQueue:阻塞队列：java提供了7种，ArrayBlockingQueue，LinkedBlockingQueue，PriorityBlockingQueue，DelayQueue，LinkedBlockingD eque

threadFactory（可选）

Handler（可选）:任务拒绝策略：AbortPolicy（默认）：丢弃任务并抛出RejectedExecutionException异常。CallerRunsPolicy：由调用线程处理该任务。丢弃任务，但是不抛出异常。可以配合这种模式进行自定义的处理方式。

流程：首先判断运行的线程数是否削与corePoolSize()；如果小于，就直接创建新的线程执行任务，如果大于，则判断workQueue阻塞队列有没有满，如果没有满，则放到阻塞队中去，如果已经满了，说明并发比较高了，则判断当前线程数是否已经大于maxPoolSize(),如果小于的话，就创建新的线程，然后执行任务。如果大于了，就执行拒绝策略，具体的拒绝策略就是自己实现的handler。

关闭线程池：shutDown():拒绝新的任务提交，待正在执行的任务执行后关闭。shutDownNow():给所有正在执行的任务interrupt中断，强行关闭。

# AQS原理

**AQS是一个用来构建锁和同步器的框架**。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。

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。

状态信息通过protected类型的getState，setState，compareAndSetState进行操作。

**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：

Share（共享）：共享锁和独占锁的主要区别是同一时刻是否能有多个线程同时获取到同步状态。

读锁就是一种共享锁。写锁就是一种独占锁。

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。

# ****ReetrantLock实现方式****

锁的获取过程：

通过cas操作来修改state状态，表示争抢锁的操作，如果能够获取到锁，设置当前获得锁状态的线程。compareAndSetState(0, 1)

如果没有获取到锁，尝试去获取锁。acquire(1)。

（1）通过tryAcquire尝试获取独占锁，如果成功返回true，失败返回false。如果是同一个线程来获得锁，则直接增加重入次数，并返回true。

（2）如果tryAcquire失败，则会通过addWaiter方法将当前线程封装成Node，添加到AQS队列尾部

（3）acquireQueued，将Node作为参数，通过自旋去尝试获取锁。（如果前驱为head才有资格进行锁的抢夺。）

（4）如果获取锁失败，则挂起线程。

锁的释放过程：

释放锁。

如果锁能够被其他线程获取，唤醒后继节点中的线程。一般情况下只要唤醒后继结点的线程就行了，但是后继结点可能已经取消等待，所以从队列尾部往前回溯，找到离头结点最近的正常结点，并唤醒其线程。

在获得同步锁时，同步器维护一个同步队列，获取状态失败的线程都会被加入到队列中并在队列中进行自旋；移出队列（或停止自旋）的条件是前驱节点为头节点且成功获取了同步状态。在释放同步状态时，同步器调用tryRelease(int arg)方法释放同步状态，然后唤醒头节点的后继节点。

# ****ReetrantLock 和 synchronized 的区别****

1. **ReetrantLock默认为非公平锁，也可以是公平锁；sync是非公平锁**
2. **ReetrantLock响应中断，sync不能中断。**

# CountDownLatch

CountDownLatch是AQS框架共享锁的实现：基本原理为， CountDownLatch在多线程并发编程中充当一个计时器的功能，并且维护一个count的变量，并且其操作都是原子操作，该类主要通过countDown()和await()两个方法实现功能的，首先通过建立CountDownLatch对象，并且传入参数即为count初始值。如果一个线程调用了await()方法，那么这个线程便进入阻塞状态，并进入阻塞队列。如果一个线程调用了countDown()方法，则会使count-1；当count的值为0时，这时候阻塞队列中调用await()方法的线程便会逐个被唤醒，从而进入后续的操作。

# Condition

对象的wait，notify是配合synchonrized使用的，而Condition是配合lock.lock使用的，Condition的强大之处在于他能有多个阻塞队列，唤醒的时候可以针对唤醒某一个阻塞队列。而Object得notify只有一个阻塞队列。

Object:wait,notify Condition:await,signal

# 在Java中Lock接口比synchronized块的优势是什么？

1. Lock接口更加灵活，提供了中断，公平锁，非公平锁，tryLock等接口。
2. synchronized是非公平锁，可重入，Lock接口可以是公平锁，也可以是非公平锁，可重入

# 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？

调用 start 方法方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。

# 说一下 runnable 和 callable 有什么区别？

相同点

1. 都是接口
2. 都可以编写多线程程序
3. 都采用Thread.start()启动线程

不同点

1. Runnable 接口 run 方法无返回值；Callable 接口 call 方法有返回值，是个泛型，和Future、FutureTask配合可以用来获取异步执行的结果
2. Runnable 接口 run 方法只能抛出运行时异常，且无法捕获处理；Callable 接口 call 方法允许抛出异常，可以获取异常信息

# ****Java 怎么实现线程？****

① 继承 Thread 类并重写 run 方法。实现简单，但不符合里氏替换原则，不可以继承其他类。

② 实现 Runnable 接口并重写 run 方法。避免了单继承局限性，实现解耦。

③实现 Callable 接口并重写 call 方法。可以获取线程执行结果的返回值，并且可以抛出异常。

# 用Java写一个会导致死锁的程序。

循环等待，多个线程拥有的资源形成了一个环，比如P1拥有P2的资源，P2拥有P1的资源。

public static void main(String[] args){  
 List<Integer> list = new LinkedList<>();  
 List<Integer> list2 = new LinkedList<>();  
 Thread t1 = new Thread(() -> {  
 synchronized (list){  
 list.add(1);  
 synchronized (list2){  
 list2.add(1);  
 }  
 }  
 System.*out*.println("t1");  
 });  
 Thread t2 = new Thread(() -> {  
 synchronized (list2){  
 list2.add(1);  
 synchronized (list){  
 list.add(1);  
 }  
 }  
 System.*out*.println("t2");  
 });  
 t1.start();  
 t2.start();  
}

# 用Java写代码来解决生产者——消费者问题