# 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。

# 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？

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waita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。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？在java中wait和sleep方法的不同？

# 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。

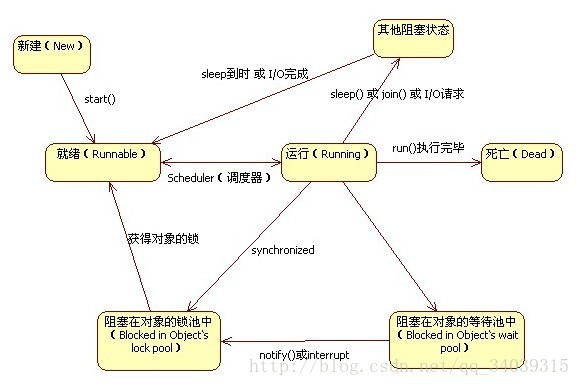
**并发**是指在一段时间内，多个程序同时运行，但是这并不是真正意义上的同时运行，只不过CPU通过时间片分时调度让他们看起来同时运行。

# 线程和进程的区别

进程是系统进行资源分配和调度的一个独立单位，而线程是CPU调度的最小单位. 一个程序至少拥有一个进程，一个进程至少有一个线程。形象点可以把进程想象为一个公司，线程就是公司里面的员工。每个公司都有它自身的业务，并且每个公司都有许多资源，当任务来的时候，就会分配资源给员工，让员工去完成任务，这里的员工就相当于线程。

# 进程间的几种状态

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. 就绪状态(Runnable)：当**线程被调用start**的时候就处于就绪状态，java虚拟机会为它创建栈和程序计数器，等待获取CPU的使用权。
3. 运行状态(Running)：当成功获取了CPU的使用权时，线程就处于运行状态，执行程序代码。
4. 阻塞状态(Blocking)：阻塞状态是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. 由于线程同步的原因，当尝试获取某个锁时，但是该锁被其他对象占用，那么Java虚拟机就会把该线程放入对象的等待队列中。
   3. 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. 死亡状态(Dead):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



# 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。

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，锁资源进行只锁自己锁需要的资源而不是锁住整个对象。比如HashTable，我们只需要访问其中部分数据，但是整个表都会被锁住，这样就会加大竞争，而改良后的concurrentHashMap就很好的解决的这个问题，把数据分为了多个segment,每次锁只锁对应的segment，这样即提高了效率。又降低了死锁发生的可能性。

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。

还有一个银行家算法也能够有效避免死锁的发生：我们每次分配资源之前都进行检查，如果可能产生死锁的话就不进行资源的分配，从而保证死锁的发生。

# 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。

# 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();  
}

# 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锁保证了数据的原子性和可见性，但是不保证有序性。

原子性是一系列不会被打断的操作。加了sync的代码块，方法只会有一个线程进入操作，从而保证了原子性和可见性。

# synchronized关键字

**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){**

**}**

# ThreadLocal

# 用Java实现阻塞队列。

# BlockingQueue介绍：

# 用Java写代码来解决生产者——消费者问题。

# 什么是竞争条件(race condition)？你怎样发现和解决的？

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

# java线程的状态转换介绍

# Java中你怎样唤醒一个阻塞的线程？

# 在Java中CycliBarriar和CountdownLatch有什么区别？

# 什么是不可变对象，它对写并发应用有什么帮助？

# 你在多线程环境中遇到的常见的问题是什么？你是怎么解决它的？

# ****ThreadLocal有什么用****

# CAS 的缺点是什么，怎么解决 ABA 问题

# 僵尸进程和孤儿进程

# 悲观锁和乐观锁

# Java 怎么保证多线程的安全

# Synchronized 的底层实现

# Java 锁机制，lock 实现

# Java 里面如何去关闭一个线程

# Java 线程池的原理和实现，一些机制