# 怎么认为一个类是线程安全？线程安全的定义是什么？Java有多少个关键字进行同步？ .

## 判断一个类是否线程安全：

多个线程访问某个类时，不管运行时环境采用何种调度方式或者这些线程将如何交替执行，并且在主调代码中不需要任何额外的同步或者协同，这个类都能表现出正确的行为，那么就称这个类是线程安全的。

## 线程安全的定义：

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。

关键字：synchronized。

# 多个线程如何指定执行顺序？如何让三个线程顺序执行？

应用到了生产者与消费者模式。详见生产者与消费者模式面经整理。

# 怎么使用Lamda表达式创建线程？

# Java多线程---JUC包下的常见类？

JUC是JDK5才引入的并发类库。它的基础就是AbstractQueuedSynchronizer抽象类。JUC在同步器的设计上有获取操作和释放操作，AbstractQueuedSynchronizer（AQS）就是一种同步器的实现。

需要满足以上两个操作，需要以下3点来支持：

1、原子操作同步状态：CAS操作是原子的，进行状态量的改变。  
2、阻塞或者唤醒一个线程：通过LockSupport.park阻塞线程，LockSupport.unpark唤醒线程，它们都是用到Native方法调用底层。

3、内部应该维护一个队列：采用CLH（三个人名）队列，Node节点，而线程Thread为Node节点的一个元素。

## CountDownLatch

CountDownLatch的主要作用是利用计数来保证线程的执行顺序，有点像倒计时，当计数为0时某个线程才能开始执行。

CountDownLatch的主要方法很简单易用，包括：

* CountDownLatch(int count) ： 构造方法，需要传入计数的初始值
* void await() ： 调用者线程会被挂起，直到计数为0时才能执行
* boolean await(long timeout, TimeUnit unit) ： 同上，但是加入了超时参数，如果超时了计数还不为0，也会照样执行，避免了一直阻塞
* void countDown() ： 计数减一

这个类是一个同步计数器，主要用于线程间的控制，当CountDownLatch的count计数>0时，await()会造成阻塞，直到count变为0，await()结束阻塞。使用countDown()会让count减1。CountDownLatch的构造函数可以设置count值，当count=1时，它的作用类似于wait()和notify()的作用。如果想让其他线程执行完指定程序，其他所有程序都执行结束后我再执行，这时可以用CountDownLatch，但计数无法被重置，如果需要重置计数，考虑使用 CyclicBarrier 。

## CyclicBarrier

CyclicBarrier的意思是可循环使用的屏障，作用是让一组线程同时到达某个时间节点。提供的重要方法如下：

* CyclicBarrier(int parties) ： 构造方法，传入线程组的数量
* CyclicBarrier(int parties, Runnable barrierAction) ： 构造方法，传入线程组的数量和当线程达到时间节点后要做的操作（由其中的某一个线程去执行）
* int await() : 挂起当前线程，直到所有线程组中的线程都完成后继续执行，返回当前线程到达的次序
* int await(long timeout, TimeUnit unit) ： 加了一个超时参数

该类从字面理解为循环屏障，它可以协同多个线程，让多个线程在这个屏障前等待，直到所有线程都到达了这个屏障时，再一起执行后面的操作。假如每个线程各有一个await，任何一个线程运行到await方法时就阻塞，直到最后一个线程运行到await时才同时返回。和之前的CountDownLatch相比，它只有await方法，而CountDownLatch是使用countDown()方法将计数器减到0，它创建的参数就是countDown的数量；CyclicBarrier创建时的int参数是await的数量。

## Semaphore

Semaphore的意思是信号量，其作用是提供一个许可范围，只有获得了许可才能继续执行。

* Semaphore(int permits)：构造方法，需要传入许可数
* Semaphore(int permits, boolean fair)：fair为true时使用公平锁，false时使用非公平锁，具体在介绍AQS时讲解
* void acquire()：获得许可，可中断
* void acquireUninterruptibly()：不可中断的获取
* void release()：释放许可

该类用于控制信号量的个数，构造时传入个数。总数就是控制并发的数量。假如是5，程序执行前用acquire()方法获得信号，则可用信号变为4，程序执行完通过release()方法归还信号量，可用信号又变为5.如果可用信号为0，acquire就会造成阻塞，等待release释放信号。acquire和release方法可以不在同一个线程使用。单个信号量的Semaphore对象可以实现互斥锁的功能，并且可以是由一个线程获得了“锁”，再由另一个线程释放“锁”，这可应用于死锁恢复的一些场合。

## Exchanger

这个类用于交换数据，只能用于两个线程。当一个线程运行到exchange()方法时会阻塞，另一个线程运行到exchange()时，二者交换数据，然后执行后面的程序。

# CountDownLatch和CyclicBarrier的区别

## 1.CountDownLatch的使用

### CountDownLatch定义

一个同步辅助类，在完成一组正在其他线程中执行的操作之前，它允许一个或多个线程一直等待。用给定的计数初始化 CountDownLatch。由于调用了 countDown() 方法，所以在当前计数到达零之前，await 方法会一直受阻塞。之后，会释放所有等待的线程，await 的所有后续调用都将立即返回。这种现象只出现一次——计数无法被重置。 一个线程(或者多个)， 等待另外N个线程完成某个事情之后才能执行。

### 分析CountDownLatch：

运行程序我们会发现当我们在t1调用CountDownLatch的await()方法时，就好比我们调用了wait()方法，当前线程会处于阻塞状态，直到等到t2和t3完全执行完毕并且调用countDown()方法时，我们才能唤醒t1继续进行执行，CountDownLatch就好比一个计时器，我们可以让当前线程调用CountDownLatch中的await()方法进行等待，如果想让当前线程继续执行，我们必须让CountDownLatch获得初始化时候传入的构造参数个countDown()方法，我们才能继续执行。

### CountDownLatch的使用场景：

在一些应用场合中，需要等待某个条件达到要求后才能做后面的事情；同时当线程都完成后也会触发事件，以便进行后面的操作。 这个时候就可以使用CountDownLatch。CountDownLatch最重要的方法是countDown()和await()，前者主要是倒数一次，后者是等待倒数到0，如果没有到达0，就只有阻塞等待了。

## 2. CyclicBarrier的使用

### CyclicBarrier的定义

CyclicBarrier 的意思是可循环使用的屏障。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。

### CyclicBarrier 分析结果

上述程序我们创建了一个线程池，这个线程池中有三个线程，每个线程都传递了一个相同的CyclicBarrier 对象和运动员的名字，我们Runner类中的run方法使每一个进来的运动员都休眠0-5秒的时间，然后调用await()方法，就是说每个线程进来都需要进行等待，直到所有的CyclicBarrier 都处于准备好了的状态，所有线程才能统一开始执行！

### CyclicBarrier 使用场景

CyclicBarrier可以用于多线程计算数据，最后合并计算结果的应用场景。比如我们用一个Excel保存了用户所有银行流水，每个Sheet保存一个帐户近一年的每笔银行流水，现在需要统计用户的日均银行流水，先用多线程处理每个sheet里的银行流水，都执行完之后，得到每个sheet的日均银行流水，最后，再用barrierAction用这些线程的计算结果，计算出整个Excel的日均银行流水。

## 3. cyclicBarrier和CountDownLatch的区别

1、CountDownLatch简单的说就是一个线程等待，直到他所等待的其他线程都执行完成并且调用countDown()方法发出通知后，当前线程才可以继续执行。

2、cyclicBarrier是所有线程都进行等待，直到所有线程都准备好进入await()方法之后，所有线程同时开始执行！

3、CountDownLatch的计数器只能使用一次。而CyclicBarrier的计数器可以使用reset() 方法重置。所以CyclicBarrier能处理更为复杂的业务场景，比如如果计算发生错误，可以重置计数器，并让线程们重新执行一次。

4,、CyclicBarrier还提供其他有用的方法，比如getNumberWaiting方法可以获得CyclicBarrier阻塞的线程数量。isBroken方法用来知道阻塞的线程是否被中断。如果被中断返回true，否则返回false。

# volatile关键字的作用？为什么使用AtomicLong而不使用Long?AtomicLong的底层是怎么实现的？

作用：保证被volatile关键字修饰的变量对所有的线程的可见性；禁止指令重排序优化。

在32位的操作系统中，对于64位的数据类型（long和double），JMM定义了一条相对宽松的规定：允许虚拟机将没有被volatile修饰的64位的数据的读写操作划分为两次32位的操作来进行，即允许虚拟机实现选择可以不保证64位数据类型的load、store、read和write这四个操作的原子性。这点就是所谓的long和double的非原子性协定。原始的数据类型long和doubler，没有实现同步，在多线程的情况下会出现错误。

在32位操作系统中，64位的long 和 double 变量由于会被JVM当作两个分离的32位来进行操作，所以不具有原子性。而使用AtomicLong能让long的操作保持原子型。

## AtomicLong的底层实现原理

AtomicLong是一个可以原子更新的long值，主要方法有：

get()

set(long newValue)

lazySet(long newValue)

getAndSet(long newValue)

boolean compareAndSet(long expect, long update)

long getAndIncrement()

long getAndAdd(long delta)

long incrementAndGet()

long decrmentAndGet()

...

1）getAndIncrement（）

在获取数据的同时，进行自增操作。

public final long getAndIncrement() {

// 死循环保证并发性

while (true) {

long current = get(); // 获取当前值

long next = current + 1; // 获取下一个值

if (compareAndSet(current, next)) // 利用原子CAS更新值

return current;

}

}

在采用CAS之前，如果在这之前有其他线程更新了这个值，那么当前值和预期值current就不相同，这将导致CAS失败，也就不会更新值为next。然后进入下一次循环，重新进行操作直到CAS操作成功。  
如果在这之前没有其他线程更新这个值，那么当前值和预期值current相同，更新值为next，然后返回值current。

这里采用了compareAndSet函数保证了原子性和并发性，来看看这个函数的实现，

public final boolean compareAndSet(long expect, long update) {

return unsafe.compareAndSwapLong(this, valueOffset, expect, update);

}

unsafe.compareAndSwapLong(this, valueOffSet, expect,  update)如何理解呢？

第一个参数，表示该属性（需要进行CAS操作的属性）所在的对象。

第二个参数，表示该属性的相对偏移量。这里的相对偏移量是指：属性所在对象的地址是A，属性的地址是B，则相对地址偏移量是B – A。

第三个参数，表示该属性预期的值。

第四个参数，表示需要更新的值。

this指向了AtomicLong的对象。而valueOffSet在类进行加载的时候就已经计算好了，因为它是一个static属性。所以对于所有的AtomicLong对象，需要更新的这个值的偏移量都是相同的

static {

try {

valueOffset = unsafe.objectFieldOffset

(AtomicLong.class.getDeclaredField("value"));

} catch (Exception ex) { throw new Error(ex); }

}

上面的静态初始块，对valueOffSet进行了设置。其调用unsafe.objectFiledOffSet（）。

2）getAndDecrement（）

获得数据的同时，进行自减操作。

3）getAndAdd（long delta）

将当前值加上delta，其实现和getAndIncrement基本一致，在一定程度上可以将后者理解为前者的特殊。

4）incrementAndGet（）

获得数据的同时，进行自增操作。和getAndIncrement不同的是，其返回的值是自增之后的值。如果要用java原始的操作符解释的话，incrementAndGet可以理解为 ++i，而getAndIncrement可以立即为 i++。

5）decrementAndGet（）

获得数据的同时，进行自减操作。和getAndDecrement不同的是，其返回的值是自减之后的值。如果要用java原始的操作符解释的话，decrementAndGet可以理解为 --i，而getAndDecrement可以立即为 i--。

6）addAndGet（long delta）

和getAndAdd相类似，其不同点和上面的几个不同点一样。

# Java5中的LongAdder和AtomicLong的区别

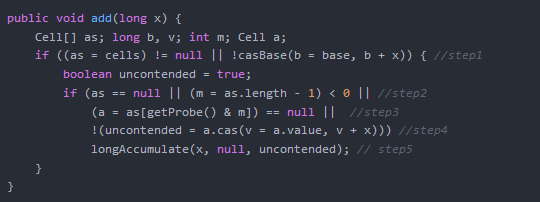
## 基本原理和思想

AtomicLong是通过无限循环不停的采取CAS的方法去设置value，直到成功为止。那么当并发数比较多或出现更新热点时，就会导致CAS的失败机率变高，重试次数更多，越多的线程重试，CAS失败的机率越高，形成恶性循环，从而降低了效率。而LongAdder的原理就是降低对value更新的并发数，也就是将对单一value的变更压力分散到多个value值上，降低单个value的“热度”。 这样，线程数再多也会分担到多个value上去更新，只需要增加value就可以降低 value的 “热度”  。

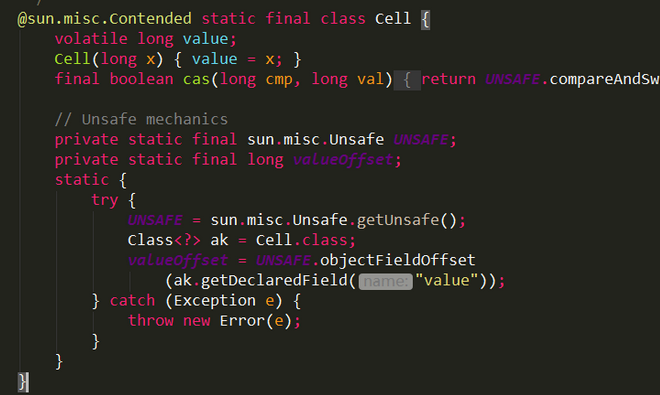
## 底层实现原理：

LongAdder中的方法和AtomicLong的方法类似，同样有增加和减少的操作。那么如何实现原子的增加呢?

### Add操作



LongAdder中包含了一个Cell 数组，Cell是Striped64的一个内部类，顾名思义，Cell 代表了一个最小单元。Cell内部有一个非常重要的value变量，并且提供了一个cas更新其值的方法。



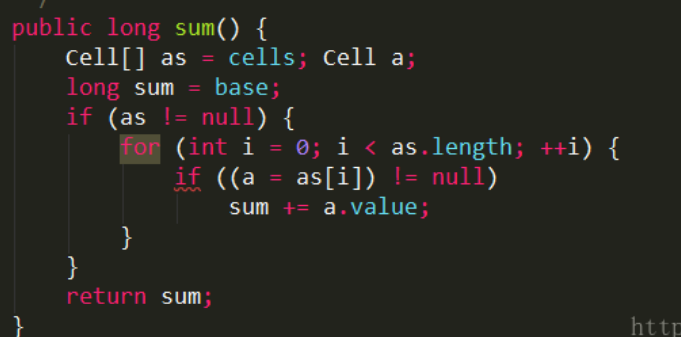
回到add方法，

如果cell数组不为空，那么就尝试更新base元素，如果更新成功，那么就直接返回。base元素在这里可以保障的是在低并发的时候和AtomicLong一样的直接对基础元素进行更新。 而如果cell为空或者更新base失败，我们看接下来的那个if判断，即如果as不为空并且成功更新对应节点的数据，则返回，否则就会进入longAccumulate()方法。

//进入longAccumulate()方法后，

longAccumulate()方法代码主要有三个分支:   
 1. 如果数组不为空   
 2. 数据为空，则初始化   
 3. 前面都更新失败了，尝试更新base数据   
  cellBusy是一个标示元素，只有当修改cell数组大小或者插入元素的时候才会修改。分支二、分支三都很简单，我们来重点分析一下分支一。   
   当要更新的位置没有元素的时候，首先cas标志位，防止扩容以及插入元素，然后插入数据。如果成功直接返回，否则标示发生了冲突，然后重试。如果对应的位置有元素则更新，如果更新失败，进行判断是否数组的大小已经超过了cpu的核数，如果大于的话，则意味着扩容没有意义。直接重试。否则进行扩容，扩容完成后，重新设置要更新的位置，尽可能保证下一次更新成功。//

如何统计计数:



当计数的时候，将base和各个cell元素里面的值进行叠加，从而得到计算总数的目的。这里的问题是在计数的同时如果修改cell元素，有可能导致计数的结果不准确。

总结:

  LongAdder在AtomicLong的基础上将单点的更新压力分散到各个节点，在低并发的时候通过对base的直接更新可以很好的保障和AtomicLong的性能基本保持一致，而在高并发的时候通过分散提高了性能。   
  缺点是LongAdder在统计的时候如果有并发更新，可能导致统计的数据有误差。

# CAS的底层实现原理？

## 什么是CAS：

CAS的全称为Compare-And-Swap，直译就是对比交换。是一条CPU的原子指令。 CAS是乐观锁技术，当多个线程尝试使用CAS同时更新同一个变量时，只有其中一个线程能更新变量的值，而其它线程都失败，失败的线程并不会被挂起，而是被告知这次竞争中失败，并可以再次尝试。

CAS 操作中包含三个操作数 —— 需要读写的内存位置（V）、进行比较的预期原值（A）和拟写入的新值(B)。如果内存位置V的值与预期原值A相匹配，那么处理器会自动将该位置值更新为新值B。否则处理器不做任何操作。无论哪种情况，它都会在 CAS 指令之前返回该位置的值。在JDK1.5 中新增 java.util.concurrent (J.U.C)就是建立在CAS之上的。CAS操作是原子性的，所以多线程并发使用CAS更新数据时，可以不使用锁相对于对于 synchronized 这种阻塞算法，CAS是非阻塞算法的一种常见实现。

## CAS缺点：

1.  **ABA问题**。因为CAS需要在操作值的时候检查下值有没有发生变化，如果没有发生变化则更新，但是如果一个值原来是A，变成了B，又变成了A，那么使用CAS进行检查时会发现它的值没有发生变化，但是实际上却变化了。ABA问题的解决思路就是使用版本号。在变量前面追加上版本号，每次变量更新的时候把版本号加一，那么A－B－A 就会变成1A-2B－3A。

**从Java1**.5开始JDK的atomic包里提供了一个类AtomicStampedReference来解决ABA问题。这个类的compareAndSet方法作用是首先检查当前引用是否等于预期引用，并且当前标志是否等于预期标志，如果全部相等，则以原子方式将该引用和该标志的值设置为给定的更新值。

**2. 循环时间长开销大**。自旋CAS如果长时间不成功，会给CPU带来非常大的执行开销。如果JVM能支持处理器提供的pause指令那么效率会有一定的提升，pause指令有两个作用，第一它可以延迟流水线执行指令（de-pipeline）,使CPU不会消耗过多的执行资源，延迟的时间取决于具体实现的版本，在一些处理器上延迟时间是零。第二它可以避免在退出循环的时候因内存顺序冲突（memory order violation）而引起CPU流水线被清空（CPU pipeline flush），从而提高CPU的执行效率。

**3. 只能保证一个共享变量的原子操作**。当对一个共享变量执行操作时，我们可以使用循环CAS的方式来保证原子操作，但是对多个共享变量操作时，循环CAS就无法保证操作的原子性，这个时候就可以用锁，或者有一个取巧的办法，就是把多个共享变量合并成一个共享变量来操作。比如有两个共享变量i＝2,j=a，合并一下ij=2a，然后用CAS来操作ij。从Java1.5开始JDK提供了**AtomicReference类来保证引用对象之间的原子性，你可以把多个变量放在一个对象里来进行CAS操作。**

# Java中Semaphore(信号量)的使用

在一种场景下，一个资源有多个副本可供同时使用，比如打印机房有多个打印机、厕所有多个坑可供同时使用，这种情况下，Java提供了另外的并发访问控制--资源的多副本的并发访问控制，信号量Semaphore即是其中的一种。

Semaphore的实现原理：

一个信号量有且仅有3种操作，且它们全部是原子的：初始化、增加和减少 。增加可以为一个进程解除阻塞； 减少可以让一个进程进入阻塞。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。

# java为什么要使用多线程？线程是不是越多效率越快？

## 1.更多的处理器核心

随着处理器上的核心数量越来越多，以及超线程技术的广泛应用，现在处理器性能提升的方式也从更高的主频向更多的核心发展。要利用好处理器上的多个核心。

一个进程可以创建多个线程，一个线程在同一时刻只能使用一个处理器核心。

## 2.更快的响应时间

有时会编写一些比较复杂的代码（复杂的业务逻辑），使用多线程技术，将数据一致性不强的操作派发给其他线程处理。

## 3.更好的编程模型

java为多线程编程提供了良好、考究并且一致的编程模型。使开发人员能够更加专注于问题的解决。

# 如何控制线程尽可能的减少上下文的切换？有没有一种方式达到了多线程的效果，但是避免了上下文的切换？

上下文切换：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，可以在加载这个任务的状态。所以任务从保存到再加载的过程就是一次上下文切换。

## 1.什么情况下会触发上下文的切换？

1. 中断请求：在中断处理中，其他程序”打断”了当前正在运行的程序。当CPU接收到中断请求时，会在正在运行的程序和发起中断请求的程序之间进行一次上下文切换。
2. 多任务执行（时间分片切换）：在多任务处理中，CPU会在不同程序之间来回切换，每个程序都有相应的处理时间片，CPU在两个时间片的间隔中进行上下文切换。
3. 用户态切换：对于一些操作系统，当进行用户态切换时也会进行一次上下文切换，虽然这不是必须的。

## 2.如何减少上下文的切换：

减少上下文切换的方法有无锁并发编程、CAS算法、使用最少的线程和使用协程。

1. 无锁并发编程。多线程竞争时，会引起上下文切换，所以多线程处理数据时，可以用一些办法来避免使用锁，将数据Id按hash算法取模来分段，不同的线程处理不同段的数据。
2. CAS算法。Java的atomic包使用CAS算法来更新数据，而不需要加锁。
3. 使用最少的线程。避免创建不需要的线程。若任务少，但创建了很多线程来处理，这样会造成大量的线程处于等等状态。
4. 协程：在单线程里实现多任务的调度，并在单线程里维持多个任务间的切换。

# 多线程是怎么通信的？多线程的哪些资源可以共享？

## 多线程之间的通信机制：

### 1.等待通知机制：

wait()、notify()、notifyAll()都是Object类中的方法，方法为本地方法，为final方法，无法重写。

* 调用某个对象的wait()方法能让当前线程阻塞，并且在调用之前当前线程必须拥有此对象的锁
* 在执行wait()方法之后当前线程释放锁调用某个对象的notify()方法能够唤醒其中一个正在等待这个对象锁的线程，notifyAll()唤醒所有
* 调用wait()、notify()都方法必须在同步块或者同步方法中进行
* notify()和notifyAll()只是唤醒正在等待该对象monitor的线程，并不决定哪个线程能够获取到monitor（一个线程被唤醒不等于立刻获取了对象的monitor,只有等调完notify()或者notifyAll()执行完并退出synchronized块后，释放对象锁后才可以收到）
* 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。
* 当线程呈wait()状态，调用该线程对象的interrupt()方法会出现InterruptedException异常

//为什么wait, notify 和 notifyAll这些方法不在thread类里面：

JAVA提供的锁是对象级的而不是线程级的，每个对象都有锁，通过线程获得,由于wait，notify和notifyAll都是锁级别的操作，所以把他们定义在Object类中因为锁属于对象。//

### 2.通过管道进行线程间的通信

管道流（pipeStream)是一种特殊的流用于在不同线程间直接传送数据，一个线程发送数据到输出管道，另一个线程从输入管道中读数据。

JDK中提供了4个类来使线程间可以进行通信：

PipedInputStream和PipedOutStream；

PipedReader和PipedWriter。

## 多线程可共享的资源：

a. 堆

 由于堆是在进程空间中开辟出来的，所以它是理所当然地被共享的；因此new出来的都是共享的（16位平台上分全局堆和局部堆，局部堆是独享的）；

b. 全局变量

它是与具体某一函数无关的，所以也与特定线程无关；因此也是共享的；

c. 静态变量

虽然对于局部变量来说，它在代码中是“放”在某一函数中的，但是其存放位置和全局变量一样，存于堆中开辟的.bss和.data段，是共享的；

d. 文件等公用资源

 这个是共享的，使用这些公共资源的线程必须同步。Win32 提供了几种同步资源的方式，包括信号、临界区、事件和互斥体。

## 多线程各自独享的资源：

1.线程ID  
      每个线程都有自己的线程ID，这个ID在本进程中是唯一的。进程用此来标识线程。

 2.寄存器组的值  
      由于线程间是并发运行的，每个线程有自己不同的运行线索，当从一个线程切换到另一个线程上 时，必须将原有的线程的寄存器集合的状态保存，以便将来该线程在被重新切换到时能得以恢复。

   3.线程的栈  
      栈是保证线程独立运行所必须的线程函数可以调用函数，而被调用函数中又是可以层层嵌套的，所以线程必须拥有自己的函数堆栈， 使得函数调用可以正常执行，不受其他线程的影响。  
   4.错误返回码  
      由于同一个进程中有很多个线程在同时运行，可能某个线程进行系统调用后设置了errno值，而在该 线程还没有处理这个错误，另外一个线程就在此时被调度器投入运行，这样错误值就有可能被修改。所以，不同的线程应该拥有自己的错误返回码变量。  
   5.线程的信号屏蔽码  
      由于每个线程所感兴趣的信号不同，所以线程的信号屏蔽码应该由线程自己管理。但所有的线程都 共享同样的信号处理器。  
   6.线程的优先级  
      由于线程需要像进程那样能够被调度，那么就必须要有可供调度使用的参数，这个参数就是线程的 优先级。

# 多个进程是如何通信的？

### 1、管道( pipe )

既可在程序中使用，也可在shell中使用。

管道是一种**半双工**的通信方式，数据只能**单向**流动。

管道的问题在于他们没有名字，只能在具有**亲缘关系（父子进程间）**的进程间使用。

**扩展：**   
管道由pipe函数创建，提供一个单向数据流。当需要一个双向数据流时，我们必须创建两个管道，每个方向一个。这也就是全双工管道的实现原理：由两个半双工管道构成。

### 2、命名管道 (named pipe) ：即FIFO

命名管道也是**半双工**的通信方式。提供单向数据流。

克服了管道没有名字的限制，因此允许**无亲缘关系**的进程间通信，解决了管道的上述问题。

**扩展：**   
管道和FIFO都是使用通常的read和write函数访问。

FIFO由mkfifo函数创建。FIFO不同于管道的是，**每个FIFO有一个路径名与之关联**，从而允许无亲缘关系的进程访问同一FIFO。

FIFO的真正优势表现在服务器可以是一个长期运行的进程（如守护进程），而且与其客户可以无亲缘关系。

### 3、信号量( semophore )

**主要作为进程间以及同一进程内不同线程之间的同步手段。**

进程间通信**处理同步互斥的机制**。信号量是一个计数器，可以用来控制多个进程对共享资源的访问。它常作为一种**锁机制**，防止某进程正在访问共享资源时，其他进程也访问该资源。

### 4、信号 ( sinal )

是一种**处理异步事件的方式**。

信号是一种比较复杂的通信方式，用于通知接收进程某个事件已经发生。除了用于进程外，还可以发送信号给进程本身。

信号和信号量是不同的，他们虽然都可用来实现同步和互斥，但前者是使用信号处理器来进行的，后者是使用P,V操作来实现的。

### 5、消息队列( message queue )

消息队列是消息的链表，包括Posix消息队列systemV消息队列。

消息队列克服了信号承载信息量少，管道只能承载无格式字节流以及缓冲区大小受限等缺点。

### 6、共享内存( shared memory )

是由一个进程创建，但多个进程都可以访问的同一块内存空间。是最快的可用IPC形式（因为共享内存区中的单个数据副本对于共享该内存的所有线程或进程都是可用的）。

是针对其他通信机制运行效率较低而设计的。往往与其它通信机制，如信号量结合使用，来达到进程间的同步和通信。

一般IPC形式（管道、FIFO、消息队列）的问题在于，两个进程要交换信息，这些信息必须经由内核传递。**而进程间共享内存时，交换数据就不再涉及内核。**这些进程必须协调或同步对该共享内存区的使用。

### 7、套接字( socket )

**更为一般的进程间通信机制，可用于不同机器之间的进程间通信。**

然后让我详细说一下用管道是怎么实现的

# 线程的中断

中断表示一个运行中的线程是否被其他线程进行了中断操作（通过调用其他线程的interrupt()方法对其进行中断操作）。

可以通过isInterrupted()来进行判断是否被中断，但要注意许多声明抛出InterruptedException的方法（例如sleep）这些方法在抛出异常之前，会先将

该线程的中断标识位清除。

Thread中中断标志位检查方法：

this.interrupted()：测试当前线程是否已经是中断状态，执行后具有将标志位置

清除为false的功能（连续两次调用，第二次返回false）。

this.isInterrupted()：测试线程Thread对象是否已经是中断状态，但不清除状

态标志。

单独调用它可以使得处于阻塞状态的线程抛出一个异常，它可以中断一个正处于

阻塞状态的线程，但不能中断处于非租塞状态的线程，通过interrupt方法和

isInterrupt()方法来停止正在运行的线程。

守护线程与用户线程的区别：守护线程依赖于创建它的线程，而用户线程则不依

赖。举个简单的例子：如果在main线程中创建了一个守护线程，当main方法运

行完毕之后，守护线程也会随着消亡。而用户线程则不会，用户线程会一直运行

直到其运行完毕。在JVM中，像垃圾收集器线程就是守护线程。

# sleep()，wait()，yield()和join()方法的区别

sleep()

## //线程中常用的方法

sleep()方法需要指定等待的时间，它可以让当前正在执行的线程在指定的时间内暂停执行，进入阻塞状态，该方法既可以让其他同优先级或者高优先级的线程得到执行的机会，也可以让低优先级的线程得到执行机会。但是sleep()方法不会释放“锁标志”，也就是说如果有synchronized同步块，其他线程仍然不能访问共享数据。是Thread类的方法。

wait()方法需要和notify()及notifyAll()两个方法一起介绍，这三个方法用于协调多个线程对共享数据的存取，所以必须在synchronized语句块内使用，也就是说，调用wait()，notify()和notifyAll()的任务在调用这些方法前必须拥有对象的锁。注意，它们都是Object类的方法，而不是Thread类的方法。

wait()  
　　wait()方法与sleep()方法的不同之处在于，wait()方法会释放对象的“锁标志”。当调用某一对象的wait()方法后，会使当前线程暂停执行，并将当前线程放入对象等待池中，直到调用了notify()方法后，将从对象等待池中移出任意一个线程并放入锁标志等待池中，只有锁标志等待池中的线程可以获取锁标志，它们随时准备争夺锁的拥有权。当调用了某个对象的notifyAll()方法，会将对象等待池中的所有线程都移动到该对象的锁标志等待池。   
　　除了使用notify()和notifyAll()方法，还可以使用带毫秒参数的wait(long timeout)方法，效果是在延迟timeout毫秒后，被暂停的线程将被恢复到锁标志等待池。   
　　此外，wait()，notify()及notifyAll()只能在synchronized语句中使用，但是如果使用的是ReenTrantLock实现同步，该如何达到这三个方法的效果呢？解决方法是使用ReenTrantLock.newCondition()获取一个Condition类对象，然后Condition的await()，signal()以及signalAll()分别对应上面的三个方法。

## ****yield()****

yield()方法和sleep()方法类似，也不会释放“锁标志”，区别在于，它没有参数，即yield()方法只是使当前线程重新回到可执行状态，所以执行yield()的线程有可能在进入到可执行状态后马上又被执行，另外yield()方法只能使同优先级或者高优先级的线程得到执行机会，这也和sleep()方法不同。

## join()

join()方法会使当前线程等待调用join()方法的线程结束后才能继续执行。

上述四种方法中，sleep()、join() 、yeild()三种方法是Thread中的方法，wait()是Object中的方法。

# 把所有认识熟用的JUC下的类写出来，讲下使用，然后讲下原生的线程操作

Java原生多线程并发方法归纳：

***Object.wait:***   
该方法只能在持有锁的情况下调用，也就是在同步块中,如下所示，调用该方法后，当前线程会被放入obj对象的waiting 池中， 同时当前线程会释放持有的obj对象的锁。   
synchronized(obj){   
…   
obj.wait();   
…   
}

***\*obj.notify():\****   
唤醒其他阻塞在obj.wait上的线程, 该方法只唤醒一个被阻塞的线程（如果有多个线程调用了obj.wait), 至于是哪一个线程被唤醒，jvm按照某种规则选取，此处不关注这点。obj.notify()不会释放对象锁，被唤醒的线程被放入obj的锁等待池中，去竞争obj的锁，如果获得 锁，就进入runnable状态。

**obj.notifyAll():**   
唤醒所有阻塞在obj.wait上的线程，被唤醒的所有线程都被放入obj的锁等待池中，去竞争锁，获得锁的进程就进入runnable状态。

**Thread.sleep():**   
当前线程进入阻塞状态，但不会释放锁持有的锁。

**volatile:**   
关键字，就是告知任何对该变量的访问均需要从共享内存中获取，而对它的改变必须同步刷新会共享内存。(多线程中，虽然变量分配的内存在共享内存中，但是每个执行的线程还是可以拥有一份拷贝，这样做的目的是加速程序的执行，这是现代多核处理器的一个显著特性). 编译器一般会对代码做优化，加上volatile后，就是告诉编译器，对该变量的访问不要做优化，对他的访问都是对内存的访问，不要做对寄存器的访问。

**ThredLocal:**   
使每个线程都可以有自己的变量副本，理解这一点的关键是所有的线程都使用同一个对象作为线程体(也就是所有的线程用同一个Worker的对象)。

i++,线程A：i++,线程B：i--，在非线程安全的情况下，i有几种取值，采用什么方法使得i线程安全

i++是不是原子操作。

原子类和CAS

AQS原理

说下多线程，我们什么时候需要分析线程数，怎么分析，分析什么因素