**什么是线程？**

**线程是操作系统能够进行运算调度最小的单位，它可被包含在进程之中，是进程中的实际运作单位，可以使用多线程对进行运算提速**

**线程安全和线程不安全**

**通俗点就是加锁的就是线程安全的，不加锁的就是线程不安全的**

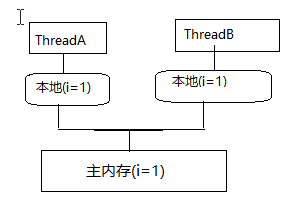
1. **线程安全就是多线程访问时，采用了加锁机制，当一个线程访问该类的某个数据时，进行保护，其他线程不能访问，直到该线程读取完，其他线程才能访问，不会出现数据不一致或者数据污染**
   1. **Vector是用同步方法来实现线程安全的，而和它相同的ArrayLIst不是线程安全的**
2. **线程不安全：就是不提供数据访问保护，有可能出现多个线程先后修改数据造成所得到的数据是脏数据**

**自旋锁**

**自旋锁是SMP架构中的一种low-level的同步机制，当线程A想要获取一把自旋锁而该所又被其它线程持有时，A会在一个循环中自旋以检测锁是不是已经可用了**

**注意：自旋时不释放cpu因此其它线程尽快释放锁，否则一直自旋浪费cpu**

**开发中不免会遇到需要所有子线程执行完毕通知主线程处理某些逻辑的场景，或者是线程1执行到某个条件通知线程2执行某个操作。可以通过以下几种方法实现：**

1. **等待通知机制**
   1. **等待通知机制是java中比较经典的线程通信方式**
   2. **两个线程通过对同一对象调用等待wait()和通知notify()进行通讯**
   3. **需要注意的是wait() notify() notifyAll() 调用的前提都是获得了对象的锁（对象监视器）**
      1. **调用wait()后进程会释放锁，进入waiting状态，该线程也会被移动到等待队列中**
      2. **调用notify()会将等待队列中的线程移动到同步队列中，线程状态也会更新为blocked**
      3. **从wait()返回的前提是调用notify()的线程释放锁，wait()的线程获得锁**
   4. **等待通知有着一个经典范式：**
      1. **线程1作为消费者：获得对象的锁。进入while(条件)，并调用wait()，当条件满足跳出循环执行具体处理逻辑**
      2. **线程2作为生产者：获得对象锁。更改与线程1共用的判断条件调用notify()**
2. **Volatile共享内存**
   1. **Java采用共享内存的方式进行线程通信的，**
   2. **Java内存模型（JMM）规定，所有的变量都存放在主内存中，每个线程都有着自己的工作内存（高速缓存）**
   3. **线程在工作时，需要将主内存中的数据copy到工作内存中。这样对数据的任何操作都是基于工作内存（效率提高），并且不能直接操作主内存以及其他线程工作内存中的数据，之后将更新之后的数据刷新到主内存中**
      1. **这里提到的主内存可以简单的认为是堆内存，工作内存可以认为是栈内存。如下图**
      2. 
      3. **所以在并发运行的时候可能会出现线程B所读取的数据是线程A更新之前的数据，所以就需要volatile了**
         1. **当一个变量被volatile修饰，任何线程对他的读写操作都会立即刷新到主内存中，并且会强制让缓存了该变量的线程中的数据清空，必须从主内存中重新读取最新数据**
         2. **Volatile修饰之后并不是让线程直接从主内存中获得数据，依然需要将变量copy到工作内存中**
   4. **内存可见性的应用**
      1. **当我们需要在两个线程间依据主内存通讯时，通信的那个变量必须是被volatile修饰**
      2. **但是这里强调的是volatile并不能保证线程安全性**
      3. **比如：虽然volatile保证了内存可见性，每个线程拿到的都是新值，但是i++这种操作并不是原子的，这里面涉及到获取值、自增、赋值的操作并不能同时完成。**
         1. **所以想要达到线程安全可以使多个线程串行执行（那这样就是变成单线程了，这就不好了）**
         2. **也可以使用synchronize或者是锁的方式来保证原子性**
         3. **还可以使用Atomic包中AtomicInterger来替换int，他利用了CAS算法来保证了原子性**
   5. **指令重排**
      1. **内存可见性只是volatile的其中一个语义，它还可以防止JVM进行指令重排**
      2. **比如一段代码，理想的执行顺序是1》2》3，但是经过JVM在保证单线程中最终结果不变的情况下优化之后可能就变成了2》1》3，当加上volatile之后可以防止这样的重排优化，保证业务的正确性**
      3. **指令重排的应用：双重懒加载的单例模式**
         1. **public class Singleton{**

**private static volatile Singleton singleton;**

**private Singleton(){}**

**public static Singleton getInstince(){**

**if(singleton = = null){**

**synchronized(Singleton.class){**

**if(singleton = = null){**

**singleton=new Singleton();**

**}**

**}**

**}**

**return singleton;**

**}**

**}**

**这里的volatile关键字主要是为了防止指令重排**

**所以执行顺序是：**

1. **分配内存空间**
2. **初始化对象**
3. **将singleton对象指向分配的内存地址**

**iv．总结：volatile 在java中并发中用的很多，比如像Atomic包中的value、以及AbstractQueueLongSynchronized中的**

**state都是被定义为volatile 来用于保证内存可见性**

**f) CountDownLatch并发工具**

**1、CountDownLatch也是基于AQS(AbstractQueuedSynchronizer)实现的**

**2、初始化一个CountDownLatch时告诉并发的线程，然后再每个线程处理完毕之后调用countDown()，该方法会将**

**AQS内置的一个state状态-1,最终将会在主线程调用await()，时他会阻塞直到state = = 0的时候返回**

**g) 管道通信**