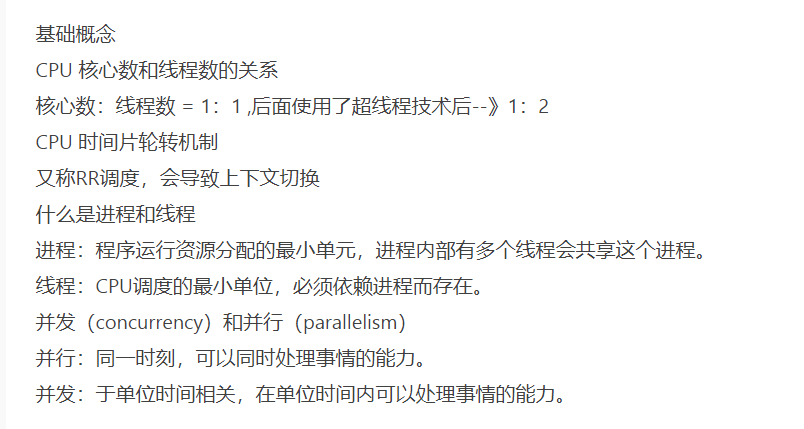
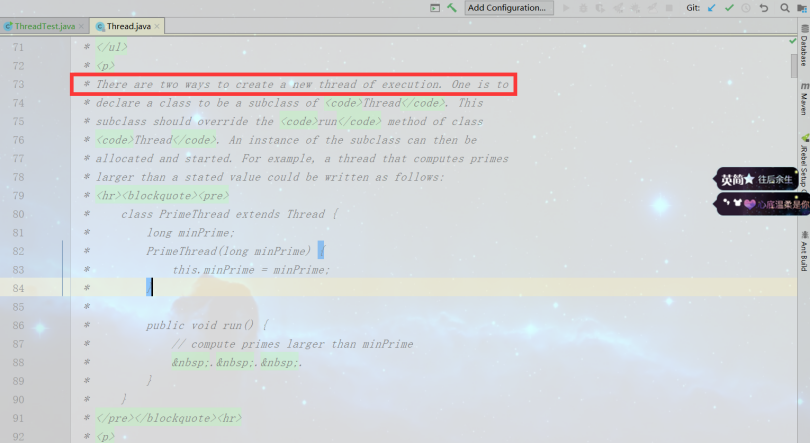
**并发编程**

# 线程与进程



# 创建线程的几种方式

1. 通过thread
2. 通过Runnable



## Thread和Runnable的区别

Thread是对线程的抽象，Runnable是对任务的抽象

## 关闭线程的方法

### Stop()

不建议使用，可能会到导致线程占用的锁不会正常释放，最终可能发生死锁问题。

### Interrupt,interrupted,isInterrupted的区别

Interrupt()是对线程进行中断，只是设置了一个标志位，线程并不会马上停止

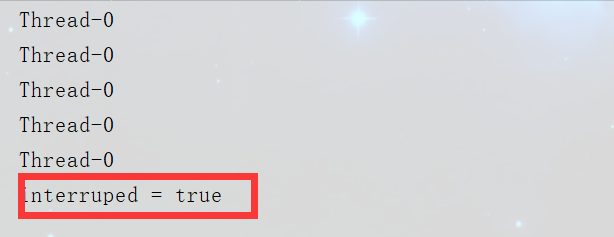
|  |
| --- |
| **public class** EndThread **implements** Runnable {  @Override  **public void** run() {  **while** (**true**) {  System.***out***.println(Thread.*currentThread*().getName());  }  }   **public static void** main(String[] args) {  EndThread endThread = **new** EndThread();  Thread thread = **new** Thread(endThread);  thread.start();  thread.interrupt();  } } |

结果：虽然调用了中断方法，但是线程没有理会，一直运行下去

Interrupted()判断当前线程是否被中断

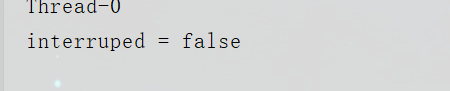
|  |
| --- |
| **public class** EndThread **extends** Thread {  @Override  **public void** run() {  **while** (!isInterrupted()) {  System.***out***.println(Thread.*currentThread*().getName());  }  }   **public static void** main(String[] args) **throws** InterruptedException {  EndThread endThread = **new** EndThread();  endThread.start();   Thread.*sleep*(1000);  endThread.interrupt();  } } |

结果:当线程被中断的时候，就不会运行下去



isInterrupted()，是静态方法，是属于类的，不不属于每个对象的，所以在中断后，会将标志设置为true，否则其他线程就不能执行了

|  |
| --- |
| **public class** EndThread **extends** Thread {  @Override  **public void** run() {  **while** (!*interrupted*()) {  System.***out***.println(Thread.*currentThread*().getName());  }  System.***out***.println(**"interruped = "** + isInterrupted());  }   **public static void** main(String[] args) **throws** InterruptedException {  EndThread endThread = **new** EndThread();  endThread.start();   Thread.*sleep*(1000);  endThread.interrupt();  } } |



## Sleep()

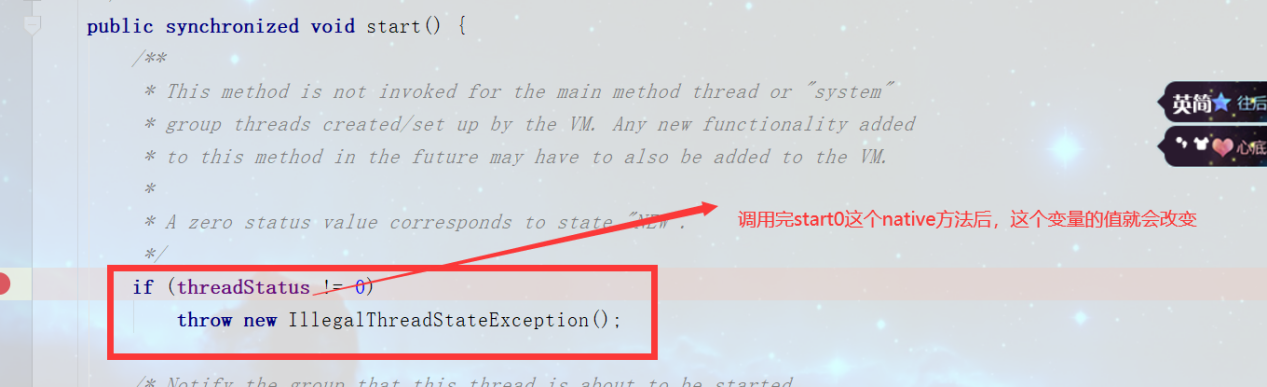
线程sleep的过程中不会失去锁

|  |
| --- |
| **public class** EndThread **extends** Thread {  @Override  **public void** run() {  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  System.***out***.println(**"interrupted = "** + *interrupted*());  e.printStackTrace();  interrupt();  System.***out***.println(**"interrupted = "** + *interrupted*());  }  }    **public static void** main(String[] args) **throws** InterruptedException {  EndThread endThread = **new** EndThread();  endThread.start();  endThread.interrupt();  } } |



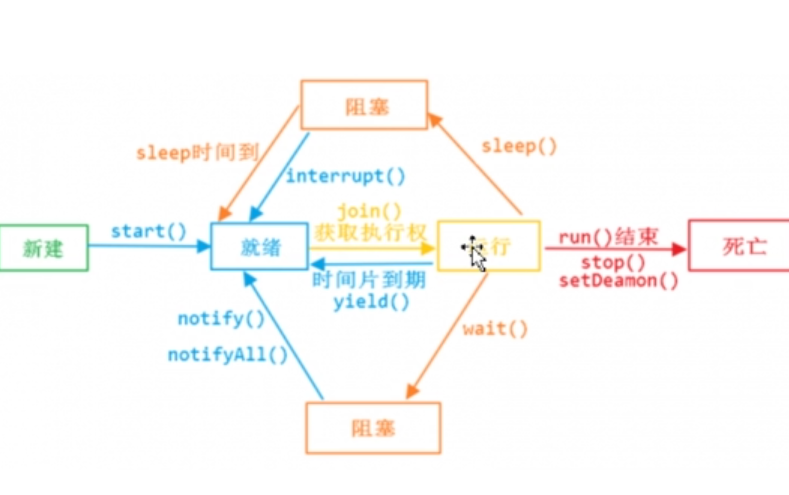
## Run()和start()

调用start()才是真正意义上启动了一个线程，才和操作系统联系起来，调用两次start方法会报错，一开始调用会判断是否是初始状态。



Run方法和其他方法没有什么区别，放线程所要做的事情，就是一个普通方法。

## 线程的状态



## Yield方法

让出cpu的执行权，不会让出锁。

## Join方法

### 面试题:如何让两个线程顺序的执行

使用join方法会使线程顺序执行

|  |
| --- |
| **public class** EndThread **extends** Thread {  @Override  **public void** run() {  System.***out***.println(Thread.*currentThread*().getName());  System.***out***.println(Thread.*currentThread*().getName() + **"========================"**);  }    **public static void** main(String[] args) **throws** InterruptedException {  EndThread endThread = **new** EndThread();  endThread.start();  endThread.join();   EndThread endThread2 = **new** EndThread();  endThread2.start();  } } |

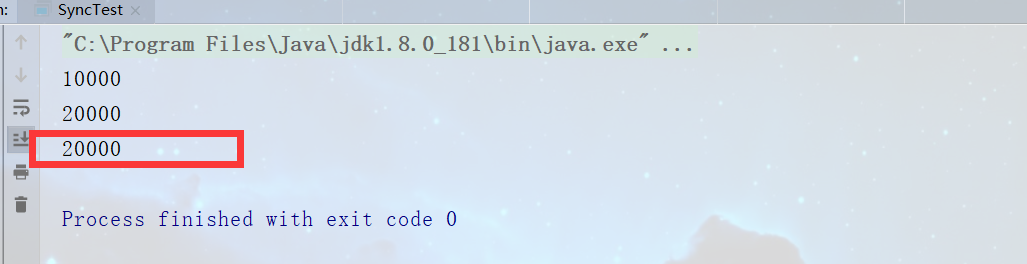
## 线程的同步

|  |
| --- |
| **public class** SyncTest{   **private static int** *count* = 0;   **public static void** main(String[] args) **throws** InterruptedException {  CountTest countTest = **new** CountTest();   Thread t1 = **new** Thread(countTest);  t1.start();   Thread t2 = **new** Thread(countTest);  t2.start();   Thread.*sleep*(2000);  System.***out***.println(*count*);  }   **private static class** CountTest **implements** Runnable {   @Override  **public void** run() {  count();  }   **private void** count() {  **for** (**int** i = 0; i < 10000; i++) {  *count*++;  }  System.***out***.println(*count*);  }  }  } |



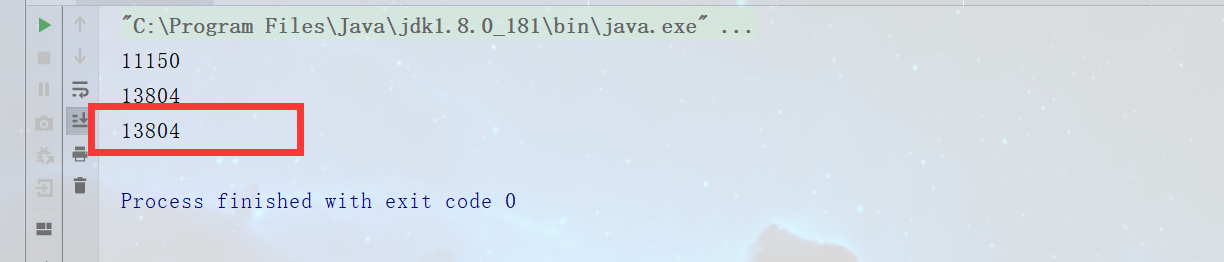
解决方法：可以在方法上加上synchronized，但是只是针对同一个对象的，不能锁住不同的对象

|  |
| --- |
| **public class** SyncTest{   **private static int** *count* = 0;   **public static void** main(String[] args) **throws** InterruptedException {  CountTest countTest = **new** CountTest();   Thread t1 = **new** Thread(countTest);  t1.start();   Thread t2 = **new** Thread(countTest);  t2.start();   Thread.*sleep*(2000);  System.***out***.println(*count*);  }   **private static class** CountTest **implements** Runnable {   @Override  **public void** run() {  count();  }   **private synchronized void** count() {  **for** (**int** i = 0; i < 10000; i++) {  *count*++;  }  System.***out***.println(*count*);  }  }  } |

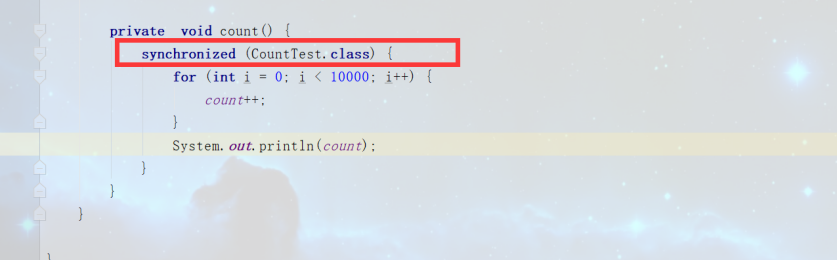


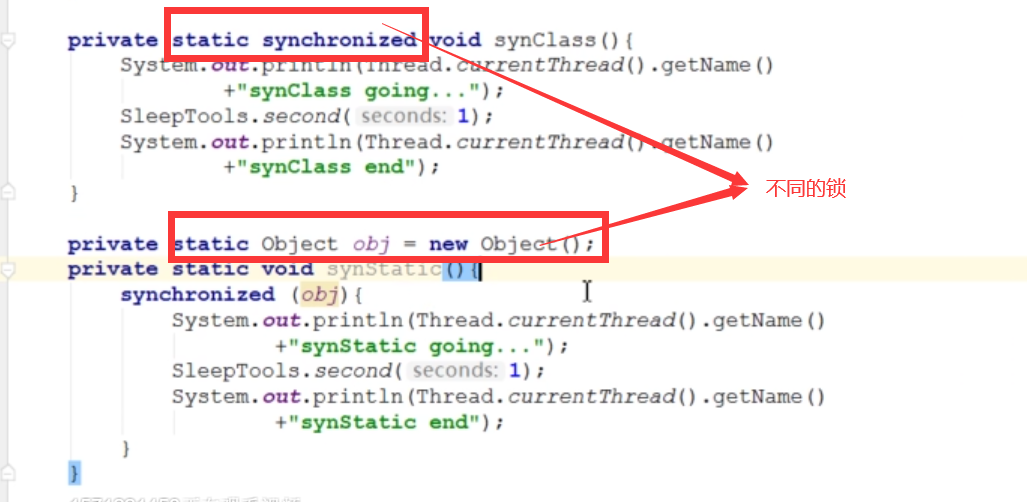
### 执行不同对象的同一个方法

|  |
| --- |
| **public class** SyncTest{   **private static int** *count* = 0;   **public static void** main(String[] args) **throws** InterruptedException {  CountTest countTest = **new** CountTest();   Thread t1 = **new** Thread(countTest);  t1.start();   CountTest countTest2 = **new** CountTest();  Thread t2 = **new** Thread(countTest2);  t2.start();   Thread.*sleep*(2000);  System.***out***.println(*count*);  }   **private static class** CountTest **implements** Runnable {   @Override  **public void** run() {  count();  }   **private synchronized void** count() {  **for** (**int** i = 0; i < 10000; i++) {  *count*++;  }  System.***out***.println(*count*);  }  }  } |



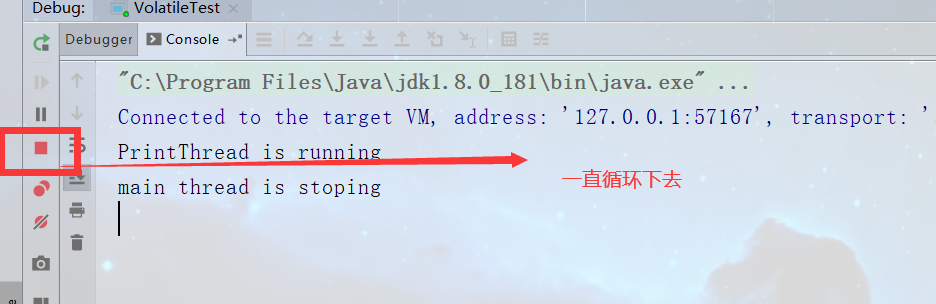
解决方法：锁住整个类



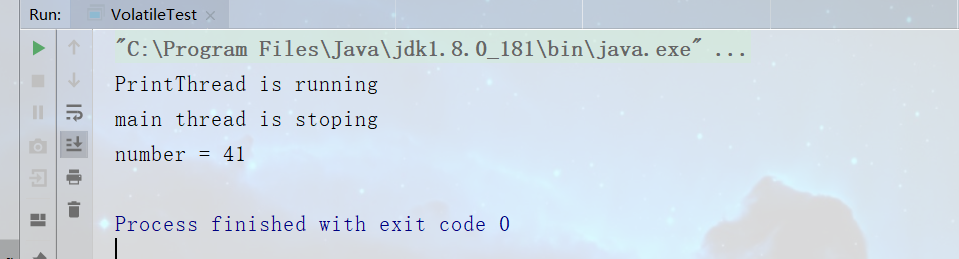


## Volatile

|  |
| --- |
| **public class** VolatileTest{   **private static boolean** *ready* = **false**;  **private static int** *number*;   **public static class** PrintThread **extends** Thread{  @Override  **public void** run() {  System.***out***.println(**"PrintThread is running"**);   **while** (!*ready*) { // 无限循环  *// System.out.println("ready = " + ready);* }   System.***out***.println(**"number = "** + *number*);  }  }   **public static void** main(String[] args) **throws** InterruptedException {  PrintThread printThread = **new** PrintThread();  printThread.start();   Thread.*sleep*(1000);   *ready* = **true**;  *number* = 41;   System.***out***.println(**"main thread is stoping"**);  }  } |

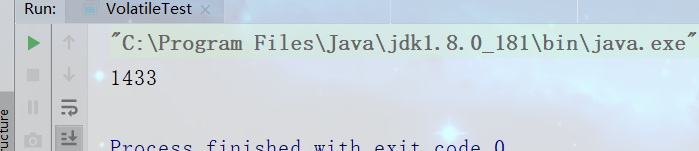


在变量上加了volatile，就会停止循环，立即从主内存中读取最新的值



### 能够保证线程安全吗

|  |
| --- |
| **public class** VolatileTest{   **private static int** *number* = 0;   **public static class** PrintThread **extends** Thread{  @Override  **public void** run() {  **for** (**int** i = 0; i < 1000; i++) {  *number*++;  }  }  }   **public static void** main(String[] args) **throws** InterruptedException {  PrintThread printThread = **new** PrintThread();  printThread.start();   PrintThread printThread2 = **new** PrintThread();  printThread2.start();   Thread.*sleep*(1000);   System.***out***.println(*number*);  }  } |

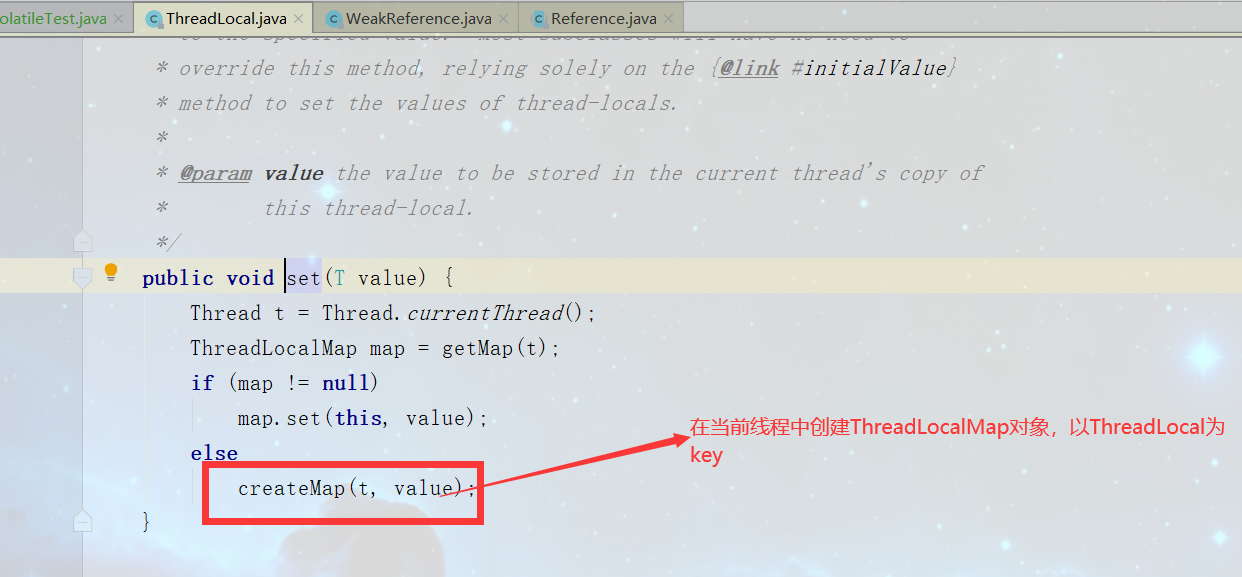


结果应该是2000，但是最终结果是1433，所以volotile不能保证线程的安全性，只能保证可见性和禁止指令重排序。

## ThreadLocal

将变量放到线程内部，不会被其他线程所共享，属于线程局部变量，实现线程隔离的，可以保证变量的安全性，不会被其他线程修改，spring的事物用到了ThreadLocal保存当前线程所用的Connection

|  |
| --- |
| **public class** VolatileTest{   **private static int** *number* = 0;   **public static class** PrintThread **extends** Thread{  **private** ThreadLocal<String> **threadLocal** = **new** ThreadLocal<>();  @Override  **public void** run() {  **threadLocal**.set(Thread.*currentThread*().getName());  **try** {  Thread.*sleep*(1000);  System.***out***.println(**"当前线程:"** + Thread.*currentThread*().getName() + **":"** + **threadLocal**.get());  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  }   **public static void** main(String[] args) **throws** InterruptedException {  PrintThread printThread = **new** PrintThread();  printThread.start();   PrintThread printThread2 = **new** PrintThread();  printThread2.start();  }  } |



### 内存泄露问题

强引用

软引用

弱引用

虚引用

JDK中的线程是协作式的

# Yield,sleep,wait,notify的区别

Yield()方法只是让出cpu执行权，不会释放锁

Sleep方法让出cpu，不会释放锁

Wait方法让出cpu，同时释放锁

Notify不会释放锁，等到同步代码快执行完了以后，才会释放锁

# 线程通信

|  |
| --- |
| */\*\*  \* 线程通信  \*/* **public class** ThreadCommunication {   **private** LinkedList<Integer> **linkedList** = **new** LinkedList<>();   **private int initSize** = 10;   **public void** add(**int** i) {  **synchronized** (**linkedList**) {  **while** (**linkedList**.size() == **initSize**) {  **try** {  System.***out***.println(**"容器已经满了"**);  **linkedList**.wait();  } **catch** (InterruptedException e) {  System.***out***.println(**"线程被中断"**);  }  }   **linkedList**.addLast(i);  **linkedList**.notifyAll();  }  }   **public void** get() {  **synchronized** (**linkedList**) {  **while** (**linkedList**.size() == 0) {  System.***out***.println(**"容器中已经为空，等待放入元素"**);  **try** {  **linkedList**.wait();  } **catch** (InterruptedException e) {  System.***out***.println(**"线程被中断"**);  }  }  System.***out***.println(**"获取到的元素为:"** + **linkedList**.removeFirst());  **linkedList**.notifyAll();  }  }   **private static class** Producer **implements** Runnable {   **private** ThreadCommunication **threadCommunication**;   **public** Producer(ThreadCommunication threadCommunication) {  **this**.**threadCommunication** = threadCommunication;  }   @Override  **public void** run() {  **for** (**int** i = 0; i < 50; i++) {  **threadCommunication**.add(i);  }  }  }   **private static class** Consumer **implements** Runnable {  **private** ThreadCommunication **threadCommunication**;   **public** Consumer(ThreadCommunication threadCommunication) {  **this**.**threadCommunication** = threadCommunication;  }   @Override  **public void** run() {  **for** (**int** i = 0; i < 20; i++) {  **this**.**threadCommunication**.get();  }  }  }   **public static void** main(String[] args) {  ThreadCommunication threadCommunication = **new** ThreadCommunication();  *// 生产者* Producer producer = **new** Producer(threadCommunication);  *// 消费者* Consumer consumer = **new** Consumer(threadCommunication);   Thread t1 = **new** Thread(producer);  Thread t2 = **new** Thread(consumer);   t1.start();  t2.start();  }  } |

# CountDownLatch和CyclicBarrier

## CountDownLatch

|  |
| --- |
| **public class** CountDownLatchTest {     **private static** CountDownLatch *countDownLatch* = **new** CountDownLatch(2);   **private static class** Worker **implements** Runnable {   **private** String **name**;   **public** Worker(String name) {  **this**.**name** = name;  }   @Override  **public void** run() {  System.***out***.println(**name** +**"已经完成了工作"**);  *countDownLatch*.countDown();  }  }   **private static class** Boss **implements** Runnable {   **private** String **name**;   **public** Boss(String name) {  **this**.**name** = name;  }   @Override  **public void** run() {  **try** {  *countDownLatch*.await();  System.***out***.println(**name** + **"开始检查工作"**);  } **catch** (InterruptedException e) {  System.***out***.println(**"老板被中断"**);  }  }  }   **public static void** main(String[] args) {  Worker worker = **new** Worker(**"张三"**);  Worker worker2 = **new** Worker(**"李四"**);  Boss boss = **new** Boss(**"王五"**);   Thread t1 = **new** Thread(worker);  Thread t2 = **new** Thread(worker2);  Thread t3 = **new** Thread(boss);   t1.start();  t2.start();  t3.start();    }  } |

一个线程等待其他线程完成任务之后再去执行。

## CyclicBarrier

|  |
| --- |
| **public class** CyclicBarrierTest {   **private static** CyclicBarrier *cyclicBarrier* = **new** CyclicBarrier(3);   **public static class** Sporter **implements** Runnable {   **private** String **name**;   **public** Sporter(String name) {  **this**.**name** = name;  }   @Override  **public void** run() {  System.***out***.println(**name** + **"已经准备好"**);  **try** {  *cyclicBarrier*.await();   System.***out***.println(**name** + **"开始跑"**);  } **catch** (InterruptedException e) {  e.printStackTrace();  } **catch** (BrokenBarrierException e) {  e.printStackTrace();  }  }  }   **public static void** main(String[] args) {  **for** (**int** i = 0; i <3; i++) {  **new** Thread(**new** Sporter(**"运动员"** + i)).start();  }  }  } |

// 可以实现汇总

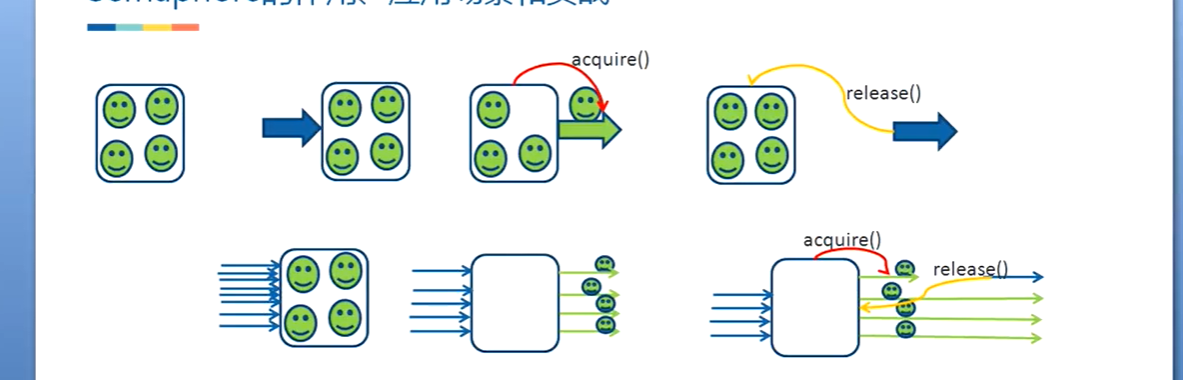
|  |
| --- |
| **public class** CyclicBarrierTest {   **private static** CyclicBarrier *cyclicBarrier* = **new** CyclicBarrier(3, **new** SummaryThred());   **private static** CopyOnWriteArrayList *writeArrayList* = **new** CopyOnWriteArrayList();   **public static class** Sporter **implements** Runnable {   **private** String **name**;   **public** Sporter(String name) {  **this**.**name** = name;  }   @Override  **public void** run() {  System.***out***.println(**name** + **"已经准备好"**);  *writeArrayList*.add(**name**);  **try** {  *cyclicBarrier*.await();   System.***out***.println(**name** + **"开始跑"**);   *// 计数器可以重复使用  cyclicBarrier*.await();  } **catch** (InterruptedException e) {  e.printStackTrace();  } **catch** (BrokenBarrierException e) {  e.printStackTrace();  }  }  }   **private static class** SummaryThred **extends** Thread {  @Override  **public void** run() {  System.***out***.println(**"汇总线程"** + *writeArrayList*);  }  }   **public static void** main(String[] args) {  **for** (**int** i = 0; i <3; i++) {  **new** Thread(**new** Sporter(**"运动员"** + i)).start();  }  }  } |

## 两者的区别

* CyclicBarrier的计数器可以反复使用，CountDownLatch不能
* CyclicBarrier之间线程的协调是由工作线程本身进行协调的，而CountDownLatch是由外面的线程协调的。
* CyclicBarrier的构造函数计数器的值和线程是密切相关的，如果计数器的值大于线程数的值，那么线程会一直阻塞下去，CountDownLatch中计数器的值和线程数没有多大的关系。
* CyclicBarrier可以汇总结果，CountDownLatch不可以。

# Semaphore

## 图解



## 用途

可以做流控使用

## 代码示例

|  |
| --- |
| **public class** SemaphoreTest {  **private** LinkedList<Integer> **linkedList** = **new** LinkedList<>();   **private int initSize** = 10;   **private** Semaphore **useful** = **new** Semaphore(10);  **private** Semaphore **useless** = **new** Semaphore(0);   **public void** add(**int** i) {  **try** {  **useful**.acquire();  } **catch** (InterruptedException e) {  e.printStackTrace();  }   **linkedList**.addLast(i);  System.***out***.println(**"放入元素"** + i);  **useless**.release();  }   **public void** get() {  **try** {  **useless**.acquire();  System.***out***.println(**"获取到的元素为:"** + **linkedList**.removeFirst());  **useful**.release();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }   **private static class** Producer **implements** Runnable {   **private** SemaphoreTest **semaphoreTest**;   **public** Producer(SemaphoreTest semaphoreTest) {  **this**.**semaphoreTest** = semaphoreTest;  }   @Override  **public void** run() {  **for** (**int** i = 0; i < 20; i++) {  **semaphoreTest**.add(i);  }  }  }   **private static class** Consumer **implements** Runnable {  **private** SemaphoreTest **semaphoreTest**;   **public** Consumer(SemaphoreTest semaphoreTest) {  **this**.**semaphoreTest** = semaphoreTest;  }   @Override  **public void** run() {  **for** (**int** i = 0; i < 20; i++) {  **this**.**semaphoreTest**.get();  }  }  }   **public static void** main(String[] args) {  SemaphoreTest semaphoreTest = **new** SemaphoreTest();  *// 生产者* Producer producer = **new** Producer(semaphoreTest);  *// 消费者* Consumer consumer = **new** Consumer(semaphoreTest);   Thread t1 = **new** Thread(producer);  Thread t2 = **new** Thread(consumer);   t1.start();  t2.start();  } } |

# ReentrantReadWriteLock

## 概述

　ReentrantReadWriteLock是Lock的另一种实现方式，我们已经知道了ReentrantLock是一个排他锁，同一时间只允许一个线程访问，而ReentrantReadWriteLock允许多个读线程同时访问，但不允许写线程和读线程、写线程和写线程同时访问。相对于排他锁，提高了并发性。在实际应用中，大部分情况下对共享数据（如缓存）的访问都是读操作远多于写操作，这时ReentrantReadWriteLock能够提供比排他锁更好的并发性和吞吐量。

　　读写锁内部维护了两个锁，一个用于读操作，一个用于写操作。所有 ReadWriteLock实现都必须保证 writeLock操作的内存同步效果也要保持与相关 readLock的联系。也就是说，成功获取读锁的线程会看到写入锁之前版本所做的所有更新。

　　ReentrantReadWriteLock支持以下功能：

　　　　1）支持公平和非公平的获取锁的方式；

　　　　2）支持可重入。读线程在获取了读锁后还可以获取读锁；写线程在获取了写锁之后既可以再次获取写锁又可以获取读锁；

　　　　3）还允许从写入锁降级为读取锁，其实现方式是：先获取写入锁，然后获取读取锁，最后释放写入锁。但是，从读取锁升级到写入锁是不允许的；

　　　　4）读取锁和写入锁都支持锁获取期间的中断；

　　　　5）Condition支持。仅写入锁提供了一个 Conditon 实现；读取锁不支持 Conditon ，readLock().newCondition() 会抛出 UnsupportedOperationException。

# 为什么需要锁



# ReentrantLock

## 例子

|  |
| --- |
| ReentrantLock **reentrantLock** = **new** ReentrantLock();  **public void** test2() {  **for** (**int** i = 0; i < 3; i++) {  **reentrantLock**.lock();  }  **// 加锁三次，解锁两次，最终还是锁着的，另一个线程不会获取到锁**  **for** (**int** i = 0; i < 2; i++) {  **try** {   } **finally** {  **reentrantLock**.unlock();  }  } }  @Test **public void** test1() {  Thread t1 = **new** Thread(**new** Runnable() {  @Override  **public void** run() {  test2();  }  });   Thread t2 = **new** Thread(**new** Runnable() {  @Override  **public void** run() {  test2();  System.***out***.println(**"================="**);  }  });   t1.start();  t2.start(); } |

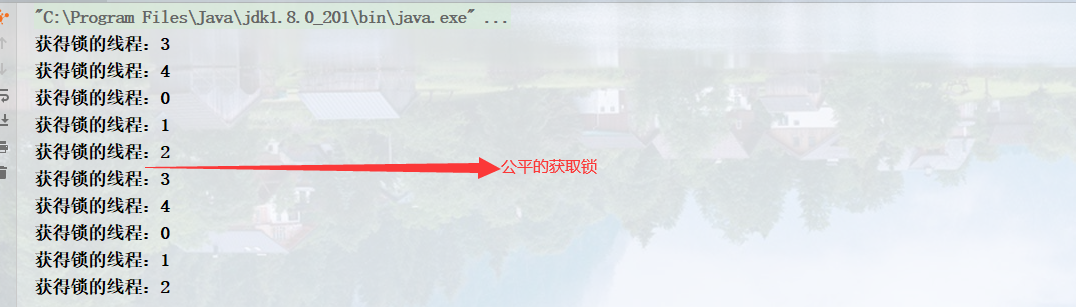
* ReentrantLock和synchronized都是独占锁,只允许线程互斥的访问临界区。但是实现上两者不同:synchronized加锁解锁的过程是隐式的,用户不用手动操作,优点是操作简单，但显得不够灵活。一般并发场景使用synchronized的就够了；ReentrantLock需要手动加锁和解锁,且解锁的操作尽量要放在finally代码块中,保证线程正确释放锁。ReentrantLock操作较为复杂，但是因为可以手动控制加锁和解锁过程,在复杂的并发场景中能派上用场。
* ReentrantLock和synchronized都是可重入的。synchronized因为可重入因此可以放在被递归执行的方法上,且不用担心线程最后能否正确释放锁；而ReentrantLock在重入时要却确保重复获取锁的次数必须和重复释放锁的次数一样，否则可能导致其他线程无法获得该锁。

## ReentrantLock相比synchronized的额外功能

### ReentrantLock可以实现公平锁。

公平锁是指当锁可用时,在锁上等待时间最长的线程将获得锁的使用权。而非公平锁则随机分配这种使用权。和synchronized一样，默认的ReentrantLock实现是非公平锁,因为相比公平锁，非公平锁性能更好。当然公平锁能防止饥饿,某些情况下也很有用。在创建ReentrantLock的时候通过传进参数true创建公平锁,如果传入的是false或没传参数则创建的是非公平锁。

|  |
| --- |
| **static ReentrantLock *reentrantLock* = new ReentrantLock(true);   public static void main(String[] args) throws InterruptedException {   for(int i=0;i<5;i++){  new Thread(new ThreadDemo(i)).start();  }   }   static class ThreadDemo implements Runnable {  Integer id;   public ThreadDemo(Integer id) {  this.id = id;  }   @Override  public void run() {  try {  TimeUnit.*MILLISECONDS*.sleep(10);  } catch (InterruptedException e) {  e.printStackTrace();  }  for(int i = 0;i < 2; i++){  *reentrantLock*.lock();  System.*out*.println("获得锁的线程："+id);  *reentrantLock*.unlock();  }  }  }** |



改为false



线程会重复获取锁。如果申请获取锁的线程足够多,那么可能会造成某些线程长时间得不到锁。这就是非公平锁的“饥饿”问题。

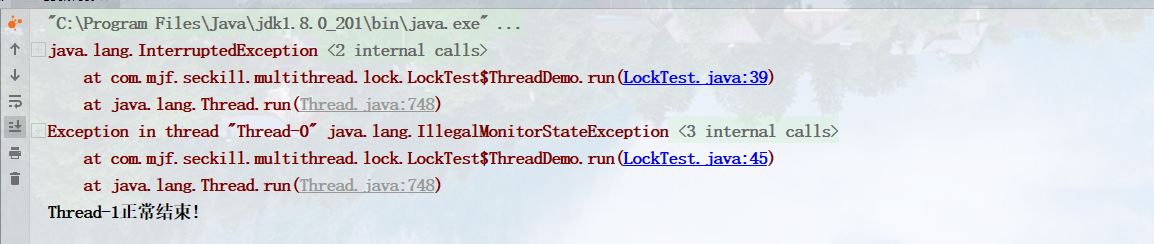
### 公平锁和非公平锁该如何选择

大部分情况下我们使用非公平锁，因为其性能比公平锁好很多。但是公平锁能够避免线程饥饿，某些情况下也很有用。

## ReentrantLock可响应中断

当使用synchronized实现锁时,阻塞在锁上的线程除非获得锁否则将一直等待下去，也就是说这种无限等待获取锁的行为无法被中断。而ReentrantLock给我们提供了一个可以响应中断的获取锁的方法lockInterruptibly()。该方法可以用来解决死锁问题。

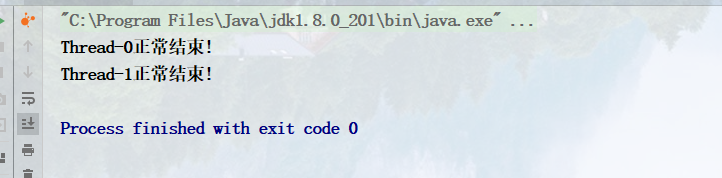
|  |
| --- |
| **static** Lock *lock1* = **new** ReentrantLock(); **static** Lock *lock2* = **new** ReentrantLock(); **public static void** main(String[] args) **throws** InterruptedException {   Thread thread = **new** Thread(**new** ThreadDemo(*lock1*, *lock2*));*//该线程先获取锁1,再获取锁2* Thread thread1 = **new** Thread(**new** ThreadDemo(*lock2*, *lock1*));*//该线程先获取锁2,再获取锁1* thread.start();  thread1.start();  thread.interrupt();*//是第一个线程中断* }  **static class** ThreadDemo **implements** Runnable {  Lock **firstLock**;  Lock **secondLock**;  **public** ThreadDemo(Lock firstLock, Lock secondLock) {  **this**.**firstLock** = firstLock;  **this**.**secondLock** = secondLock;  }  @Override  **public void** run() {  **try** {  **firstLock**.lockInterruptibly();  TimeUnit.***MILLISECONDS***.sleep(10);*//更好的触发死锁* **secondLock**.lockInterruptibly();  } **catch** (InterruptedException e) {  e.printStackTrace();  } **finally** {  **firstLock**.unlock();  **secondLock**.unlock();  System.***out***.println(Thread.*currentThread*().getName()+**"正常结束!"**);  }  } } |



## 获取锁时限时等待

ReentrantLock还给我们提供了获取锁限时等待的方法tryLock(),可以选择传入时间参数,表示等待指定的时间,无参则表示立即返回锁申请的结果:true表示获取锁成功,false表示获取锁失败。我们可以使用该方法配合失败重试机制来更好的解决死锁问题。

|  |
| --- |
| **static** Lock *lock1* = **new** ReentrantLock(); **static** Lock *lock2* = **new** ReentrantLock(); **public static void** main(String[] args) **throws** InterruptedException {   Thread thread = **new** Thread(**new** ThreadDemo(*lock1*, *lock2*));*//该线程先获取锁1,再获取锁2* Thread thread1 = **new** Thread(**new** ThreadDemo(*lock2*, *lock1*));*//该线程先获取锁2,再获取锁1* thread.start();  thread1.start(); }  **static class** ThreadDemo **implements** Runnable {  Lock **firstLock**;  Lock **secondLock**;  **public** ThreadDemo(Lock firstLock, Lock secondLock) {  **this**.**firstLock** = firstLock;  **this**.**secondLock** = secondLock;  }  @Override  **public void** run() {  **try** {  **while**(!*lock1*.tryLock()){  TimeUnit.***MILLISECONDS***.sleep(10);  }  **while**(!*lock2*.tryLock()){  *lock1*.unlock();  TimeUnit.***MILLISECONDS***.sleep(10);  }  } **catch** (InterruptedException e) {  e.printStackTrace();  } **finally** {  **firstLock**.unlock();  **secondLock**.unlock();  System.***out***.println(Thread.*currentThread*().getName()+**"正常结束!"**);  }  } } |



线程通过调用tryLock()方法获取锁,第一次获取锁失败时会休眠10毫秒,然后重新获取，直到获取成功。第二次获取失败时,首先会释放第一把锁,再休眠10毫秒,然后重试直到成功为止。线程获取第二把锁失败时将会释放第一把锁，这是解决死锁问题的关键,避免了两个线程分别持有一把锁然后相互请求另一把锁。

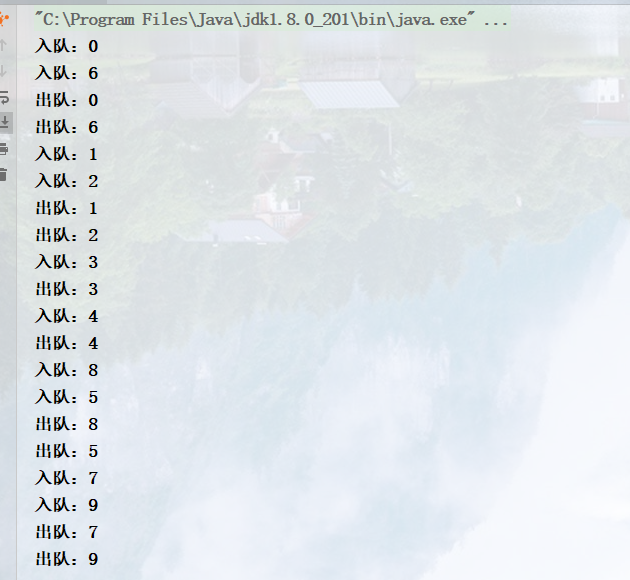
## 使用Condition实现简单的阻塞队列

阻塞队列是一种特殊的先进先出队列,它有以下几个特点

1.入队和出队线程安全

2.当队列满时,入队线程会被阻塞;当队列为空时,出队线程会被阻塞。

|  |
| --- |
| **int size**;*//阻塞队列最大容量* ReentrantLock **lock** = **new** ReentrantLock();  LinkedList<E> **list**=**new** LinkedList<>();*//队列底层实现* Condition **notFull** = **lock**.newCondition();*//队列满时的等待条件* Condition **notEmpty** = **lock**.newCondition();*//队列空时的等待条件* **public** LockTest(**int** size) {  **this**.**size** = size; }  **public void** enqueue(E e) **throws** InterruptedException {  **lock**.lock();  **try** {  **while** (**list**.size() ==**size**)*//队列已满,在notFull条件上等待* **notFull**.await();  **list**.add(e);*//入队:加入链表末尾* System.***out***.println(**"入队："** +e);  **notEmpty**.signal(); *//通知在notEmpty条件上等待的线程* } **finally** {  **lock**.unlock();  } }  **public** E dequeue() **throws** InterruptedException {  E e;  **lock**.lock();  **try** {  **while** (**list**.size() == 0)*//队列为空,在notEmpty条件上等待* **notEmpty**.await();  e = **list**.removeFirst();*//出队:移除链表首元素* System.***out***.println(**"出队："**+e);  **notFull**.signal();*//通知在notFull条件上等待的线程* **return** e;  } **finally** {  **lock**.unlock();  } }  **public static void** main(String[] args) {  LockTest<Integer> queue = **new** LockTest<>(2);  **for** (**int** i = 0; i < 10; i++) {  **int** data = i;  **new** Thread(**new** Runnable() {  @Override  **public void** run() {  **try** {  queue.enqueue(data);  } **catch** (InterruptedException e) {   }  }  }).start();   }  **for**(**int** i=0;i<10;i++){  **new** Thread(**new** Runnable() {  @Override  **public void** run() {  **try** {  Integer data = queue.dequeue();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  }).start();  } } |

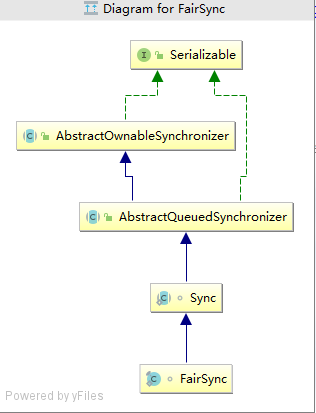


## 总结

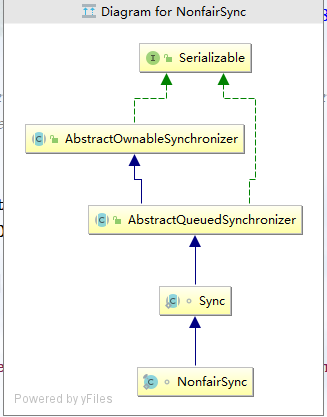
ReentrantLock是可重入的独占锁。比起synchronized功能更加丰富，支持公平锁实现，支持中断响应以及限时等待等等。可以配合一个或多个Condition条件方便的实现等待通知机制。

## 源码角度

### FairSync

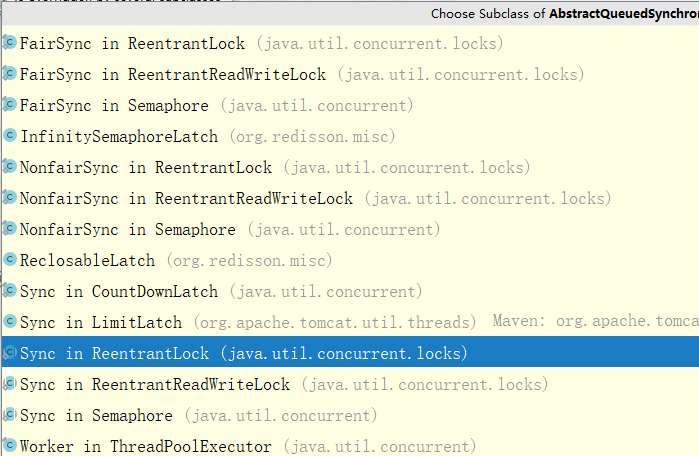


### NonfairSync



### AbstractQueuedSynchronizer介绍

AbstractQueuedSynchronizer,简称AQS,为构建不同的同步组件(重入锁,读写锁,CountDownLatch等)提供了可扩展的基础框架



AQS以模板方法模式在内部定义了获取和释放同步状态的模板方法,并留下钩子函数供子类继承时进行扩展,由子类决定在获取和释放同步状态时的细节,从而实现满足自身功能特性的需求。除此之外,AQS通过内部的同步队列管理获取同步状态失败的线程,向实现者屏蔽了线程阻塞和唤醒的细节。

#### AQS的内部结构(ReentrantLock的语境下)

AQS的内部结构主要由同步等待队列构成

##### 同步等待队列

AQS中同步等待队列的实现是一个带头尾指针(这里用指针表示引用是为了后面讲解源码时可以更直观形象,况且引用本身是一种受限的指针)且不带哨兵结点(后文中的头结点表示队列首元素结点,不是指哨兵结点)的双向链表。

|  |
| --- |
| */\*\*  \* Head of the wait queue, lazily initialized. Except for  \* initialization, it is modified only via method setHead. Note:  \* If head exists, its waitStatus is guaranteed not to be  \* CANCELLED.  \*/* **private transient volatile** Node **head**;  */\*\*  \* Tail of the wait queue, lazily initialized. Modified only via  \* method enq to add new wait node.  \*/* **private transient volatile** Node **tail**; |

head是头指针,指向队列的首元素;tail是尾指针,指向队列的尾元素。而队列的元素结点Node定义在AQS内部,主要有如下几个成员变量

volatile Node prev; //指向前一个结点的指针

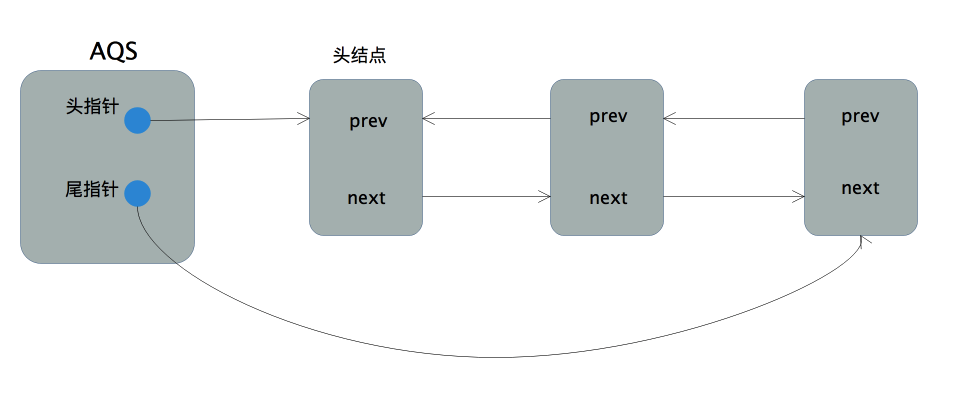
volatile Node next; //指向后一个结点的指针

volatile Thread thread; //当前结点代表的线程

volatile int waitStatus; //等待状态

* prev:指向前一个结点的指针
* next:指向后一个结点的指针
* thread:当前结点表示的线程,因为同步队列中的结点内部封装了之前竞争锁失败的线程,故而结点内部必然有一个对应线程实例的引用
* waitStatus:对于重入锁而言,主要有3个值。0:初始化状态；-1(SIGNAL):当前结点表示的线程在释放锁后需要唤醒后续节点的线程;1(CANCELLED):在同步队列中等待的线程等待超时或者被中断,取消继续等待。

**同步队列的结构如下图所示**

****

##### AQS中的其他数据结构(ReentrantLock的语境下)

（1）同步状态变量

|  |
| --- |
| /\*\*  \* The synchronization state.  \*/  private volatile int state; |

这是一个带volatile前缀的int值,是一个类似计数器的东西。在不同的同步组件中有不同的含义。以ReentrantLock为例,state可以用来表示该锁被线程重入的次数。当state为0表示该锁不被任何线程持有;当state为1表示线程恰好持有该锁1次(未重入);当state大于1则表示锁被线程重入state次。因为这是一个会被并发访问的量,为了防止出现可见性问题要用volatile进行修饰。

1. 持有同步状态的线程标志

|  |
| --- |
| /\*\*  \* The current owner of exclusive mode synchronization.  \*/  private transient Thread exclusiveOwnerThread; |

如注释所言,这是在独占同步模式下标记持有同步状态线程的。ReentrantLock就是典型的独占同步模式,该变量用来标识锁被哪个线程持有。

#### 获取锁和加锁的事项

|  |
| --- |
| **private static final** Unsafe ***unsafe*** = Unsafe.*getUnsafe*(); **private static final long *stateOffset***; **private static final long *headOffset***; **private static final long *tailOffset***; **private static final long *waitStatusOffset***; **private static final long *nextOffset***;  **static** {  **try** {  **// 获取某个属性的地址偏移值**  ***stateOffset*** = ***unsafe***.objectFieldOffset  (AbstractQueuedSynchronizer.**class**.getDeclaredField(**"state"**));  ***headOffset*** = ***unsafe***.objectFieldOffset  (AbstractQueuedSynchronizer.**class**.getDeclaredField(**"head"**));  ***tailOffset*** = ***unsafe***.objectFieldOffset  (AbstractQueuedSynchronizer.**class**.getDeclaredField(**"tail"**));  ***waitStatusOffset*** = ***unsafe***.objectFieldOffset  (Node.**class**.getDeclaredField(**"waitStatus"**));  ***nextOffset*** = ***unsafe***.objectFieldOffset  (Node.**class**.getDeclaredField(**"next"**));   } **catch** (Exception ex) { **throw new** Error(ex); } } |

#### 获取锁失败的线程如何安全的加入同步队列:addWaiter()

|  |
| --- |
| ***/\*\*  \* Creates and enqueues node for current thread and given mode.  \*  \* @param mode Node.EXCLUSIVE for exclusive, Node.SHARED for shared  \* @return the new node  \*/* private Node addWaiter(Node mode) {  Node node = new Node(Thread.*currentThread*(), mode);  *// Try the fast path of enq; backup to full enq on failure* Node pred = tail;  if (pred != null) {  node.prev = pred;  if (compareAndSetTail(pred, node)) {  pred.next = node;  return node;  }  }  enq(node);  return node; }**  **private** Node enq(**final** Node node) {  **for** (;;) {  Node t = **tail**;  **if** (t == **null**) { *// Must initialize*  ***// 头节点不存储有线程的信息,只是一个标志而已*** **if** (compareAndSetHead(**new** Node()))  **tail** = **head**;  } **else** {  node.**prev** = t;  **// 不断修改tail所指向的节点来插入新的节点**  **if** (compareAndSetTail(t, node)) {  t.**next** = node;  **return** t;  }  }  } } |

##### 线程加入同步队列后会做什么:acquireQueued()

|  |
| --- |
| **final boolean acquireQueued(final Node node, int arg) {  boolean failed = true;  try {  boolean interrupted = false;  for (;;) {  final Node p = node.predecessor();  if (p == head && tryAcquire(arg)) {  setHead(node);  p.next = null; *// help GC* failed = false;  return interrupted;  }  if (*shouldParkAfterFailedAcquire*(p, node) &&  parkAndCheckInterrupt())  interrupted = true;  }  } finally {  if (failed)  cancelAcquire(node);  } }** |

这段代码主要的内容都在for循环中,这是一个死循环,主要有两个if分句构成。第一个if分句中,当前线程首先会判断前驱结点是否是头结点,如果是则尝试获取锁,获取锁成功则会设置当前结点为头结点(更新头指针)。为什么必须前驱结点为头结点才尝试去获取锁？因为头结点表示当前正占有锁的线程,正常情况下该线程释放锁后会通知后面结点中阻塞的线程,阻塞线程被唤醒后去获取锁,这是我们希望看到的。然而还有一种情况,就是前驱结点取消了等待,此时当前线程也会被唤醒,这时候就不应该去获取锁,而是往前回溯一直找到一个没有取消等待的结点,然后将自身连接在它后面。一旦我们成功获取了锁并成功将自身设置为头结点,就会跳出for循环。否则就会执行第二个if分句:确保前驱结点的状态为SIGNAL,然后阻塞当前线程。

先来看shouldParkAfterFailedAcquire(p, node)，从方法名上我们可以大概猜出这是判断是否要阻塞当前线程的,方法内容如下

|  |
| --- |
| **private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {  int ws = pred.waitStatus;  if (ws == Node.*SIGNAL*)  */\*  \* This node has already set status asking a release  \* to signal it, so it can safely park.  \*/* return true;  if (ws > 0) {  */\*  \* Predecessor was cancelled. Skip over predecessors and  \* indicate retry.  \*/* do {  node.prev = pred = pred.prev;  } while (pred.waitStatus > 0);  pred.next = node;  } else {  */\*  \* waitStatus must be 0 or PROPAGATE. Indicate that we  \* need a signal, but don't park yet. Caller will need to  \* retry to make sure it cannot acquire before parking.  \*/  compareAndSetWaitStatus*(pred, ws, Node.*SIGNAL*);  }  return false; }** |

可以看到针对前驱结点pred的状态会进行不同的处理

1.pred状态为SIGNAL,则返回true,表示要阻塞当前线程。

2.pred状态为CANCELLED,则一直往队列头部回溯直到找到一个状态不为CANCELLED的结点,将当前节点node挂在这个结点的后面。

3.pred的状态为初始化状态,此时通过compareAndSetWaitStatus(pred, ws, Node.SIGNAL)方法将pred的状态改为SIGNAL。

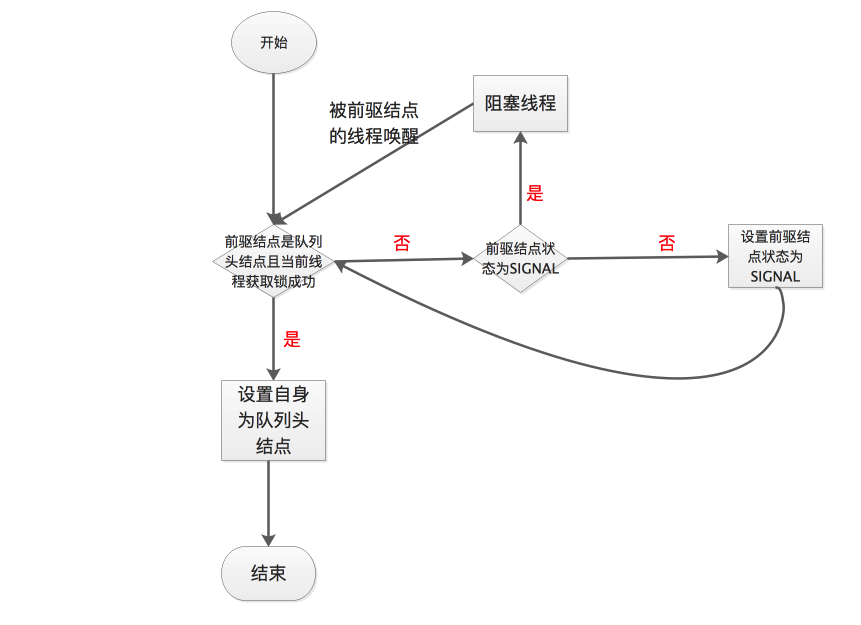
其实这个方法的含义很简单,就是确保当前结点的前驱结点的状态为SIGNAL,SIGNAL意味着线程释放锁后会唤醒后面阻塞的线程。毕竟,只有确保能够被唤醒，当前线程才能放心的阻塞。

但是要注意只有在前驱结点已经是SIGNAL状态后才会执行后面的方法立即阻塞,对应上面的第一种情况。其他两种情况则因为返回false而重新执行一遍for循环。这种延迟阻塞其实也是一种高并发场景下的优化,试想我如果在重新执行循环的时候成功获取了锁,是不是线程阻塞唤醒的开销就省了呢？

最后我们来看看阻塞线程的方法parkAndCheckInterrupt

shouldParkAfterFailedAcquire返回true表示应该阻塞当前线程,则会执行parkAndCheckInterrupt方法,这个方法比较简单,底层调用了LockSupport来阻塞当前线程,源码如下:

|  |
| --- |
| **private final boolean parkAndCheckInterrupt() {  LockSupport.*park*(this);  return Thread.*interrupted*(); }** |



概括的说,线程在同步队列中会尝试获取锁,失败则被阻塞,被唤醒后会不停的重复这个过程,直到线程真正持有了锁,并将自身结点置于队列头部。

### 非公平模式解锁流程

### 为什么非公平锁性能好

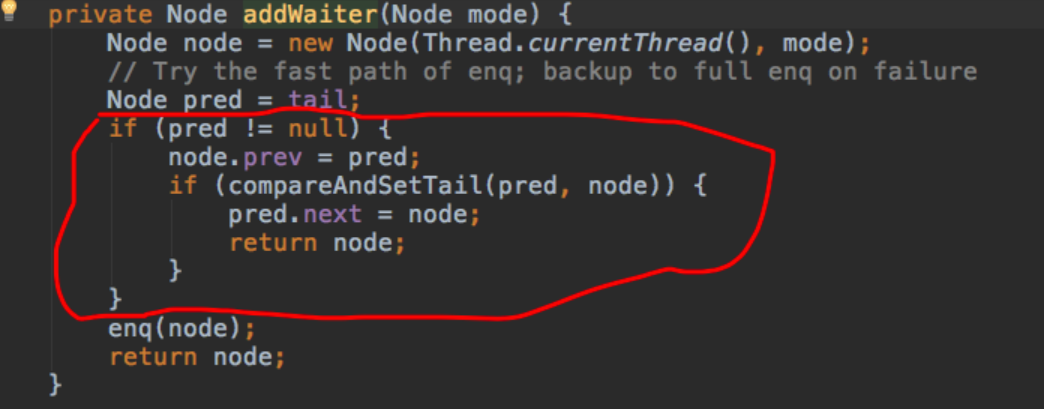
非公平锁对锁的竞争是抢占式的(队列中线程除外),线程在进入等待队列前可以进行两次尝试,这大大增加了获取锁的机会。这种好处体现在两个方面:

1.线程不必加入等待队列就可以获得锁,不仅免去了构造结点并加入队列的繁琐操作,同时也节省了线程阻塞唤醒的开销,线程阻塞和唤醒涉及到线程上下文的切换和操作系统的系统调用,是非常耗时的。在高并发情况下,如果线程持有锁的时间非常短,短到线程入队阻塞的过程超过线程持有并释放锁的时间开销,那么这种抢占式特性对并发性能的提升会更加明显。

2.减少CAS竞争。如果线程必须要加入阻塞队列才能获取锁,那入队时CAS竞争将变得异常激烈,CAS操作虽然不会导致失败线程挂起,但不断失败重试导致的对CPU的浪费也不能忽视。除此之外,加锁流程中至少有两处通过将某些特殊情况提前来减少CAS操作的竞争,增加并发情况下的性能。一处就是获取锁时将非重入的情况提前,如下图所示



另一处就是入队的操作,将同步队列非空的情况提前处理



这两部分的代码在之后的通用逻辑处理中都有,很显然属于重复代码,但因为避免了执行无意义的流程代码,比如for循环,获取同步状态等,高并发场景下也能减少CAS竞争失败的可能。

# JMM

## 参考链接

<https://www.jianshu.com/p/8a58d8335270>

## 线程之间的通信

线程的通信是指线程之间以何种机制来交换信息。在编程中，线程之间的通信机制有两种，共享内存和消息传递。  
在共享内存的并发模型里，线程之间共享程序的公共状态，线程之间通过写-读内存中的公共状态来隐式进行通信，典型的共享内存通信方式就是通过共享对象进行通信。  
在消息传递的并发模型里，线程之间没有公共状态，线程之间必须通过明确的发送消息来显式进行通信，在java中典型的消息传递方式就是wait()和notify()。

## 线程间的同步

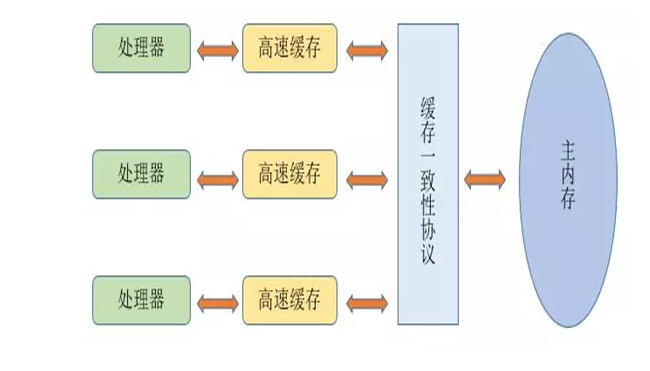
同步是指程序用于控制不同线程之间操作发生相对顺序的机制。  
在共享内存并发模型里，同步是显式进行的。程序员必须显式指定某个方法或某段代码需要在线程之间互斥执行。  
在消息传递的并发模型里，由于消息的发送必须在消息的接收之前，因此同步是隐式进行的。

## 现代计算机的内存模型

物理计算机中的并发问题，物理机遇到的并发问题与虚拟机中的情况有不少相似之处，物理机对并发的处理方案对于虚拟机的实现也有相当大的参考意义。

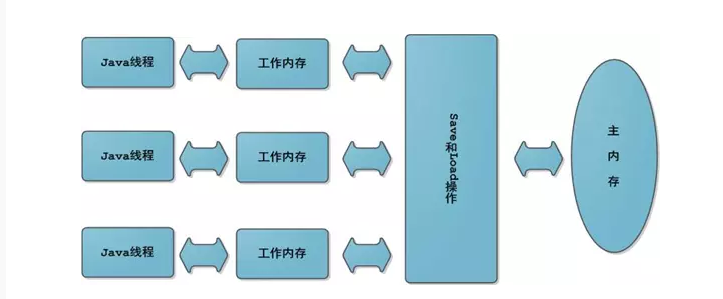
其中一个重要的复杂性来源是绝大多数的运算任务都不可能只靠处理器“计算”就能完成，处理器至少要与内存交互，如读取运算数据、存储运算结果等，这个I/O操作是很难消除的（无法仅靠寄存器来完成所有运算任务）。早期计算机中cpu和内存的速度是差不多的，但在现代计算机中，cpu的指令速度远超内存的存取速度,由于计算机的存储设备与处理器的运算速度有几个数量级的差距，所以现代计算机系统都不得不加入一层读写速度尽可能接近处理器运算速度的高速缓存（Cache）来作为内存与处理器之间的缓冲：将运算需要使用到的数据复制到缓存中，让运算能快速进行，当运算结束后再从缓存同步回内存之中，这样处理器就无须等待缓慢的内存读写了。

基于高速缓存的存储交互很好地解决了处理器与内存的速度矛盾，但是也为计算机系统带来更高的复杂度，因为它引入了一个新的问题：缓存一致性（Cache Coherence）。在多处理器系统中，每个处理器都有自己的高速缓存，而它们又共享同一主内存（MainMemory）。当多个处理器的运算任务都涉及同一块主内存区域时，将可能导致各自的缓存数据不一致，举例说明变量在多个CPU之间的共享。如果真的发生这种情况，那同步回到主内存时以谁的缓存数据为准呢？为了解决一致性的问题，需要各个处理器访问缓存时都遵循一些协议，在读写时要根据协议来进行操作，这类协议有MSI、MESI（Illinois Protocol）、MOSI、Synapse、Firefly及Dragon Protocol等。



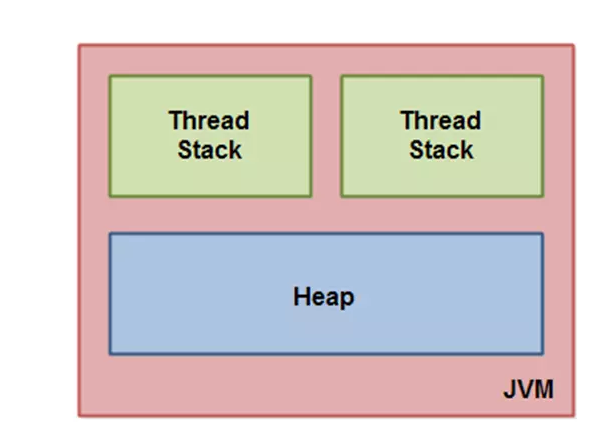
## Java内存模型（JMM）

JMM定义了Java 虚拟机(JVM)在计算机内存(RAM)中的工作方式。JVM是整个计算机虚拟模型，所以JMM是隶属于JVM的。从抽象的角度来看，JMM定义了线程和主内存之间的抽象关系：线程之间的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），本地内存中存储了该线程以读/写共享变量的副本。本地内存是JMM的一个抽象概念，并不真实存在。它涵盖了缓存、写缓冲区、寄存器以及其他的硬件和编译器优化。

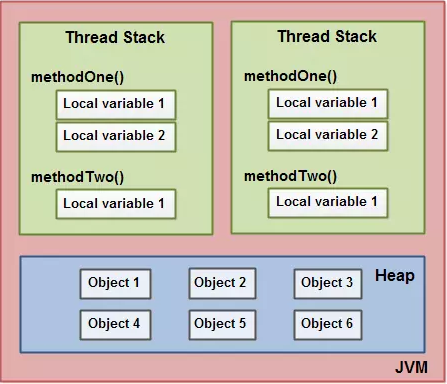


### JVM对Java内存模型的实现

在JVM内部，Java内存模型把内存分成了两部分：线程栈区和堆区  
JVM中运行的每个线程都拥有自己的线程栈，线程栈包含了当前线程执行的方法调用相关信息，我们也把它称作调用栈。随着代码的不断执行，调用栈会不断变化。



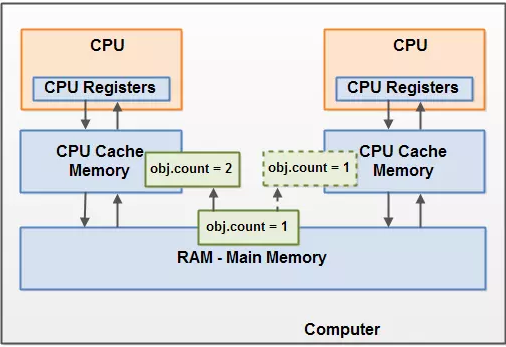
所有原始类型(boolean,byte,short,char,int,long,float,double)的局部变量都直接保存在线程栈当中，对于它们的值各个线程之间都是独立的。对于原始类型的局部变量，一个线程可以传递一个副本给另一个线程，当它们之间是无法共享的。  
堆区包含了Java应用创建的所有对象信息，不管对象是哪个线程创建的，其中的对象包括原始类型的封装类（如Byte、Integer、Long等等）。不管对象是属于一个成员变量还是方法中的局部变量，它都会被存储在堆区。  
一个局部变量如果是原始类型，那么它会被完全存储到栈区。 一个局部变量也有可能是一个对象的引用，这种情况下，这个本地引用会被存储到栈中，但是对象本身仍然存储在堆区。  
对于一个对象的成员方法，这些方法中包含局部变量，仍需要存储在栈区，即使它们所属的对象在堆区。 对于一个对象的成员变量，不管它是原始类型还是包装类型，都会被存储到堆区。Static类型的变量以及类本身相关信息都会随着类本身存储在堆区。



## Java内存模型带来的问题

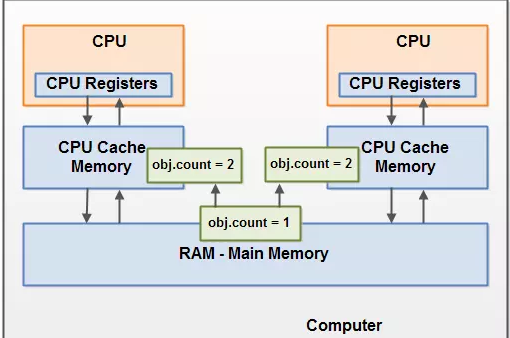
### 可见性问题

CPU中运行的线程从主存中拷贝共享对象obj到它的CPU缓存，把对象obj的count变量改为2。但这个变更对运行在右边CPU中的线程不可见，因为这个更改还没有flush到主存中：要解决共享对象可见性这个问题，我们可以使用java volatile关键字或者是加锁.



### 竞争现象

线程A和线程B共享一个对象obj。假设线程A从主存读取Obj.count变量到自己的CPU缓存，同时，线程B也读取了Obj.count变量到它的CPU缓存，并且这两个线程都对Obj.count做了加1操作。此时，Obj.count加1操作被执行了两次，不过都在不同的CPU缓存中。如果这两个加1操作是串行执行的，那么Obj.count变量便会在原始值上加2，最终主存中的Obj.count的值会是3。然而下图中两个加1操作是并行的，不管是线程A还是线程B先flush计算结果到主存，最终主存中的Obj.count只会增加1次变成2，尽管一共有两次加1操作。 要解决上面的问题我们可以使用java synchronized代码块。



## Java内存模型中的重排序

在执行程序时，为了提高性能，编译器和处理器常常会对指令做重排序。

### 重排序类型



· 1）编译器优化的重排序。编译器在不改变单线程程序语义的前提下，可以重新安排语句的执行顺序。

· 2）指令级并行的重排序。现代处理器采用了指令级并行技术（Instruction-LevelParallelism，ILP）来将多条指令重叠执行。如果不存在数据依赖性，处理器可以改变语句对应机器指令的执行顺序。

· 3）内存系统的重排序。由于处理器使用缓存和读/写缓冲区，这使得加载和存储操作看上去可能是在乱序执行。

## volatile的内存语义

volatile变量自身具有下列特性：

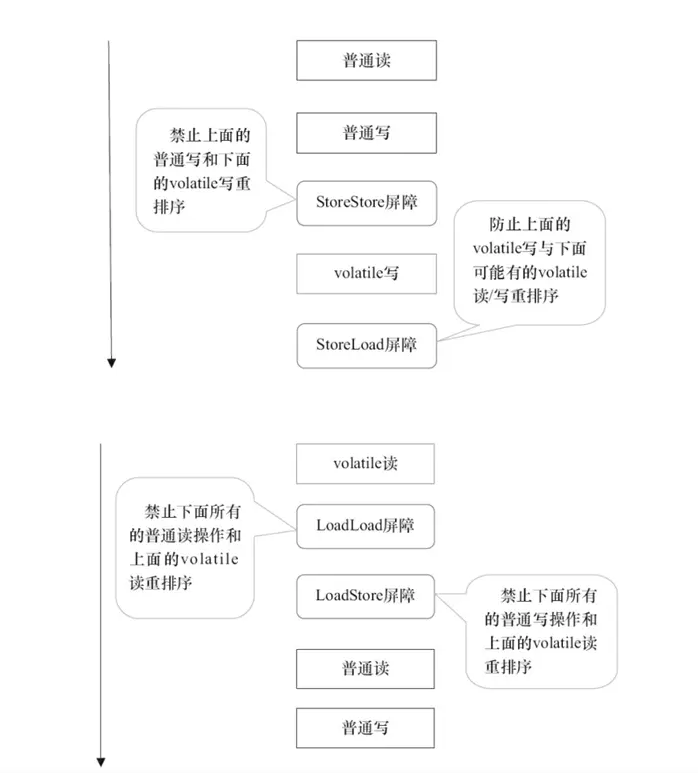
可见性。对一个volatile变量的读，总是能看到（任意线程）对这个volatile变量最后的写入。

原子性：对任意单个volatile变量的读/写具有原子性，但类似于volatile++这种复合操作不具有原子性。

volatile写的内存语义如下：当写一个volatile变量时，JMM会把该线程对应的本地内存中的共享变量值刷新到主内存。

volatile读的内存语义如下：当读一个volatile变量时，JMM会把该线程对应的本地内存置为无效。线程接下来将从主内存中读取共享变。

volatile内存语义的实现——JMM对volatile的内存屏障插入策略：  
在每个volatile写操作的前面插入一个StoreStore屏障。在每个volatile写操作的后面插入一个StoreLoad屏障。  
在每个volatile读操作的后面插入一个LoadLoad屏障。在每个volatile读操作的后面插入一个LoadStore屏障。



### volatile的实现原理

有volatile变量修饰的共享变量进行写操作的时候会使用CPU提供的Lock前缀指令：

* 将当前处理器缓存行的数据写回到系统内存
* 这个写回内存的操作会使在其他CPU里缓存了该内存地址的数据无效。