高并发线程

# 1、并发和并行

1、并行：多个任务同时执行

2、并发：多个任务交替进行，肉眼观察和并行是一样的。

# 2、临界区：

表示一种公共的资源或者说是共享数据，可以被多个线程使用，但是每一次，只能有一个线程使用它，一旦临界区资源被占用，其他线程要是想使用这个资源，就必须等待。

# 3、线程的状态

### 1、所有的线程都在State 枚举中定义

|  |
| --- |
| **public** **enum** State {  *NEW*,  *RUNNABLE*,  *BLOCKED*,  *WAITING*,  *TIMED\_WAITING*,    *TERMINATED*;  } |

### 2、解释

new New表示新建的线程

# 4、终止线程

一般来说，线程执行完毕会自动结束，无需手动关闭，，但是凡事有例外，一些服务器后台线程可能会常驻系统，比如，它本身就是一个大大的无穷循环。

线程提供给了一个Stop方法，可以离职终止一个线程，但是会发现stop已经被弃用了。Thread.stop()方法在结束线程的时候，会直接终止线程，这个线程就会立即释放掉所有的锁，而这些锁恰恰是用来维护对象的一致性的，如果此时写到了一半，强行中止，那么对象就会被破坏，所以现在不使用了

## 1、错误的实例

|  |
| --- |
| **public** **class** StopThreadUnsafe {  **public** **static** User *user* = **new** User();  **public** **static** **class** User {  **private** **int** id;  **private** String name;  **public** **int** getId() {  **return** id;  }  **public** **void** setId(**int** id) {  **this**.id = id;  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** User() {  id = 0;  name = "0";  }  @Override  **public** String toString() {  **return** "User{" +  "name='" + name + '\'' +  ", id=" + id +  '}';  }  }  **public** **static** **class** ChangeObjectThread **extends** Thread {  **public** **void** run() {  **while** (**true**) {  **synchronized** (*user*) {  **int** v = (**int**) (System.*currentTimeMillis*() / 1000);  *user*.setId(v);  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  *user*.setName(v + "");  }  Thread.*yield*();  }  }  }  **public** **static** **class** ReadObjectThread **extends** Thread {  **public** **void** run() {  **while** (**true**) {  **synchronized** (*user*) {  **if** (*user*.getId() != Integer.*parseInt*(*user*.getName())) {  System.*out*.println(*user*.toString());  }  }  Thread.*yield*();  }  }  }  **public** **static** **void** main(String args[]) **throws** InterruptedException {  **new** ReadObjectThread().start();  **while** (**true**) {  Thread thread = **new** ChangeObjectThread();  thread.start();  Thread.*sleep*(150);  thread.~~stop~~();  }  }  } |

## 2、中止线程的正确方法为，添加一个标记变量，用于提示线程是否需要退出

## 3、Thread.yield( )

Java线程中有一个Thread.yield( )方法，很多人翻译成线程让步。顾名思义，使当前线程从执行状态（运行状态）变为可执行态（就绪状态）。就是说当一个线程使用了这个方法之后，它就会把自己CPU执行的时间让掉，让自己或者其它的线程运行。（也就是谁先抢到谁执行）

# 5、线程中断

严格 讲，线程中断不会使线程立即退出，而是给线程发送一个通知，告诉目标线程，有人希望你退出，后续目前线程接到通知怎么处理，是线程自己的事情了，但是如果中断后，线程立即无条件退出，又会遇到上面的Thread.stop()方法的问题

# 6、wait和notify（等待和通知）

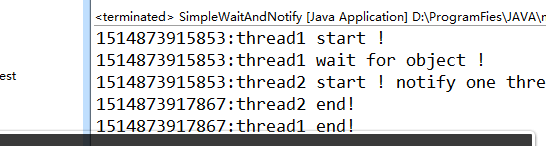
这两个方法都是Object的方法，当一个对象实例上调用了，wait()方法后，当前线程就会在这个对象上等待，比如，线程A调用了obj.wait()方法，那么线程A就会停止继续运行，而转化为等待状态，知道其他的线程比如 B调用了obj.nofify()方法，这个obj对象就会成为多个对象之间的通信手段

用法必须在同步方法或者是同步块代码中，并且会释放锁，这样其他对象才能notify哦

## 1、测试代码

|  |
| --- |
| **public** **class** SimpleWaitAndNotify {  **final** **static** Object *object* = **new** Object();  **public** **static** **class** Thread1 **extends** Thread {  **public** **void** run() {  **synchronized** (*object*) {  System.*out*.println(System.*currentTimeMillis*() + ":thread1 start !");  **try** {  System.*out*.println(System.*currentTimeMillis*() + ":thread1 wait for object !");  *object*.wait();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.*out*.println(System.*currentTimeMillis*() + ":thread1 end!");  }  }  }  **public** **static** **class** Thread2 **extends** Thread {  **public** **void** run() {  **synchronized** (*object*) {  System.*out*.println(System.*currentTimeMillis*() + ":thread2 start ! notify one thread");  *object*.notify();  **try** {  Thread.*sleep*(2000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.*out*.println(System.*currentTimeMillis*() + ":thread2 end!");  }  }  }  **public** **static** **void** main(String args[]) {  Thread thread1 = **new** Thread1();  Thread thread2 = **new** Thread2();  thread1.start();  thread2.start();  }  //wait()方法会释放目标对象的锁,而Thread.sleep()方法不会释放任何资源.  } |

## 2、控制台（控制台结果并不一定是，因为不能控制线程1和线程2哪个先执行我们希望是这样，）



# 7、挂起（suspend）和继续执行（resume）线程

挂起和继续执行，是相反的操作，被挂起的线程必须等到resume之后才能继续执行，现在已经被标注为废弃方法，已经 不推荐使用了。

# 8、等待程序结束join和谦让yield

很多时候，一个线程的输入，非常依赖别的线程的输出，此时，需要等待线程执行完毕，才能继续执行，所以出现了join方法，有两个join方法

## 1、Join方法

**public** **final** **void** join() **throws** InterruptedException

**public** **final** **synchronized** **void** join(**long** millis)

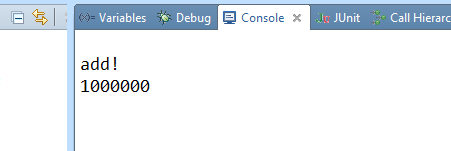
第一个奇偶是你为表示无限等待，它会一直阻塞当前线程，直到目标线程执行完毕

第二个给出了最大等待时间，如果超过给定的时间，目标线程还在执行，当前线程也会因为等不及了，而继续往下执行

## 2、测试

|  |
| --- |
| **public** **class** JoinMain {  **public** **volatile** **static** **int** *i* = 0;  **public** **static** **class** AddThread **extends** Thread {  **public** **void** run() {  System.*out*.println("add!");  **for** (*i* = 0; *i* < 1000000; *i*++) ;  **try** {  Thread.*sleep*(500);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  }  **public** **static** **void** main(String args[]) **throws** InterruptedException {  AddThread at = **new** AddThread();  at.start();  at.join();//使用join()方法后,主线程会等待AddThread执行完毕,i输出为1000000,如果没有这条语句,i输出为0  //可以查看join的底层代码,本质即让调用线程在当前线程对象实例上等待  System.*out*.println(*i*);  }  } |

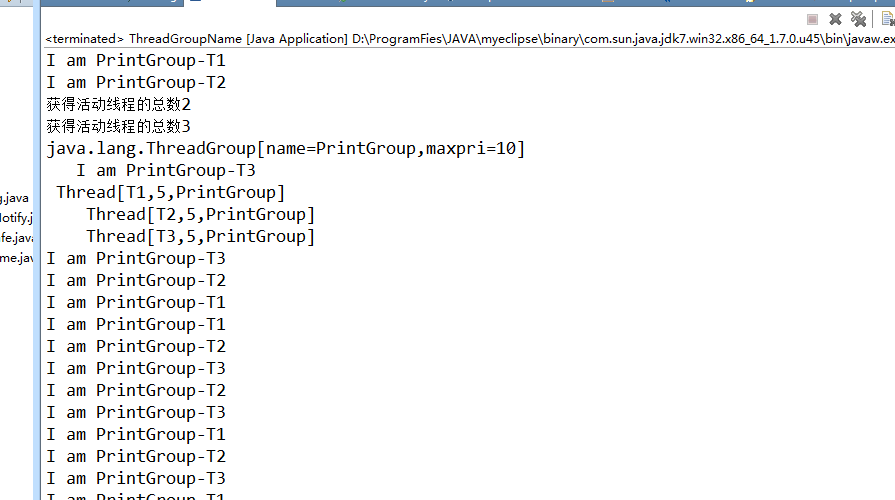
### 2.控制台



# 9、线程组ThreadGroup

在一个系统给中，如果线程数量很多，而且功能分配比较明确，可以将相同功能的放到一个线程组中，

|  |
| --- |
| **public** **class** ThreadGroupName **implements** Runnable {  @Override  **public** **void** run() {  String groupAndName = Thread.*currentThread*().getThreadGroup().getName() + "-" + Thread.*currentThread*().getName();  **while** (**true**) {  System.*out*.println("I am " + groupAndName);  **try** {  Thread.*sleep*(3000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  }  **public** **static** **void** main(String args[]) {  ThreadGroup tg = **new** ThreadGroup("PrintGroup");  Thread t1 = **new** Thread(tg, **new** ThreadGroupName(), "T1");  Thread t2 = **new** Thread(tg, **new** ThreadGroupName(), "T2");  t1.start();  t2.start();  System.*out*.println("获得活动线程的总数"+tg.activeCount()); //获得活动线程的总数  Thread t3 = **new** Thread(tg, **new** ThreadGroupName(), "T3");  t3.start();  System.*out*.println("获得活动线程的总数"+tg.activeCount());  tg.list(); //打印这个线程组中 所有的线程信息，对于调试有一定的帮助，  }  } |



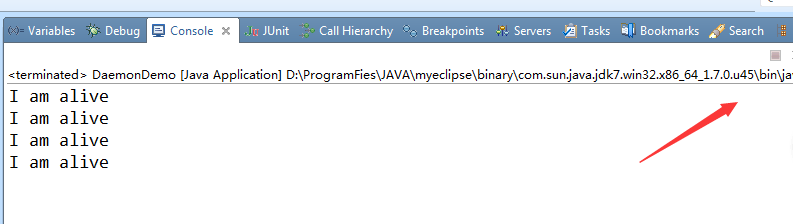
# 10、守护线程Daemon和用户线程

## 1、解释：

守护线程，是在后台默默的完成一些系统的服务，比如垃圾回收线程，与之相对于的就是用户线程，用户线程可以认为是系统的工作线程，如果用户线程全部结束，也就意味着，这个应用程序无事可做了，那么守护线程要守护的对象也就不存在了，整个应用程序也就结束了，必须在start之前设置 thread.setDaemon(**true**);

## 2、代码

|  |
| --- |
| **public** **class** DaemonDemo {  **public** **static** **class** DaemonT **extends** Thread {  **public** **void** run() {  **while** (**true**) {  System.*out*.println("I am alive");  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  }  }  /\*\*  \* thread被设置为守护线程,系统中只有主线程main为用户线程,因此在main线程休眠3秒退出时,整个程序也随之结束,如果不把线程thread设置为守护线程,  \* main线程结束后,t线程还会不停的打印,永远也不会结束.  \*  \* tip:当一个Java应用内,只有守护线程时,Java虚拟机就会退出.  \* \*/  **public** **static** **void** main(String args[]) **throws** InterruptedException {  Thread thread = **new** DaemonT();  thread.setDaemon(**true**);  thread.start();  Thread.*sleep*(3000);  }  } |



# 11、synchronize

1、指定加锁对象，对给定对象加锁

2、直接作用于实例方法，相当于对当前实例加锁（this）

3、作用于静态方法，相当于对当前类加锁

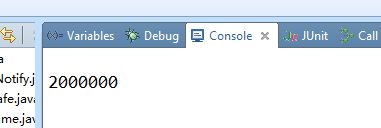
# 12、重入锁ReentrantLock

## 1、解释： synchronize obj.wait(),obj.nofiy()的替代品，

### 1、代码

|  |
| --- |
| **public** **class** ReenterLock **implements** Runnable {  **public** **static** ReentrantLock *lock* = **new** ReentrantLock();  **public** **static** **int** *i* = 0;  @Override  **public** **void** run() {  **for** (**int** j = 0; j < 1000000; j++) {  *lock*.lock();  **try** {  *i*++;  } **finally** {  *lock*.unlock();  }  }  }  **public** **static** **void** main(String args[]) **throws** InterruptedException {  ReenterLock reenterLock = **new** ReenterLock();  Thread thread1 = **new** Thread(reenterLock);  Thread thread2 = **new** Thread(reenterLock);  thread1.start();  thread2.start();  thread1.join();  thread2.join();  System.*out*.println(*i*);  }  } |

### 2、控制台



### 3、详解

开发人员必须手动选择何时加锁，何时释放锁，冲入锁的灵活性，远远高原synchronized，但是一定要记得，退出临界区是，一定要释放锁，否则其他线程没有机会再使用临界区（多个线程交替使用，不像synchronize，线程单独霸占）

## 2、中断锁（避免死锁）

### 解释：

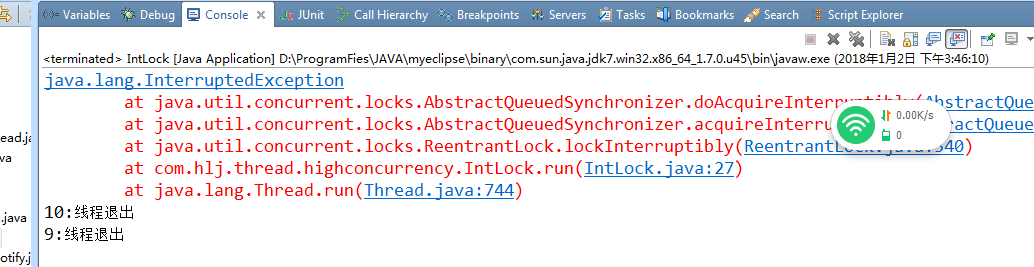
下面的t1和t2启动后，t1先占用lock1，在占用lock2，t2县占用lock2，再占用lock1，这样很容易形成死锁，这里统一使用了lockInterruptibly获得锁，但优先响应中断 这个是一个可以对中断响应的申请动作，即在等待锁的过程中可以响应中断（然后就会抛出异常），isHeldByCurrentThread检查当前线程是否拥有该锁，

如果main处于休眠，这个时候应该是正好这两个线程处于死锁状态了，后来t2中断了因此，t2放弃了对于lock1的申请，同事释放已经获得的lock2，这样t1就可以顺利的执行下去，通过控制台可以看到，两个线程双双退出，但是真正执行完工作的只有t1，t2线程则放弃了任务直接退出

### 1、代码

|  |
| --- |
| **public** **class** IntLock **implements** Runnable {  **public** **static** ReentrantLock *lock1* = **new** ReentrantLock();  **public** **static** ReentrantLock *lock2* = **new** ReentrantLock();  **int** lock;  **public** IntLock(**int** lock) {  **this**.lock = lock;  }  @Override  **public** **void** run() {  **try** {  **if** (lock == 1) {  *lock1*.lockInterruptibly();  Thread.*sleep*(500);  *lock2*.lockInterruptibly();  } **else** {  *lock2*.lockInterruptibly();  Thread.*sleep*(500);  *lock1*.lockInterruptibly();  }  } **catch** (InterruptedException e) {  e.printStackTrace();  } **finally** {  **if** (*lock1*.isHeldByCurrentThread()) {  *lock1*.unlock();  }  **if** (*lock2*.isHeldByCurrentThread()) {  *lock2*.unlock();  }  System.*out*.println(Thread.*currentThread*().getId() + ":线程退出");  }  }  /\*\*  \* 中断响应lockInterruptibly  \*  \* **@param** args  \* **@throws** InterruptedException  \*/  **public** **static** **void** main(String args[]) **throws** InterruptedException {  IntLock r1 = **new** IntLock(1);  IntLock r2 = **new** IntLock(2);  Thread thread1 = **new** Thread(r1);  Thread thread2 = **new** Thread(r2);  thread1.start();  thread2.start();  Thread.*sleep*(1000);  thread2.interrupt();  }  } |

### 2、控制台



## 3、锁申请等待限时

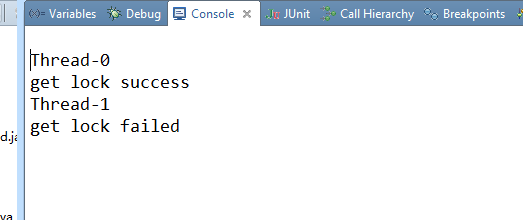
### 解释：

限时等待，比如，约好朋友，他如果2个小时没来，那么，就不等了

### 1、代码

|  |
| --- |
| **public** **class** TimeLock **implements** Runnable {  **public** **static** ReentrantLock *lock* = **new** ReentrantLock();  @Override  **public** **void** run() {  **try** {//等待时长，计时单位  **if** (*lock*.tryLock(5, TimeUnit.*SECONDS*)) {  System.out.println(Thread.currentThread().getName());  System.*out*.println("get lock success");  Thread.*sleep*(6000); //占用线程6秒，哈哈，肯定大于5秒了，所以两个  } **else** {  System.*out*.println(Thread.*currentThread*().getName());  System.*out*.println("get lock failed");  }  } **catch** (InterruptedException e) {  e.printStackTrace();  } **finally** {  **if** (*lock*.isHeldByCurrentThread()) {  *lock*.unlock();  }  }  }  /\*\*  \* 锁申请等待限时tryLock  \* **@param** args  \*/  **public** **static** **void** main(String args[]) {  TimeLock timeLock = **new** TimeLock();  Thread thread1 = **new** Thread(timeLock);  Thread thread2 = **new** Thread(timeLock);  thread1.start();  thread2.start();  }  } |

### 2、控制台



## 4、锁申请不限时l*ock*.tryLock()) {

上面的ReentratLock.trylock()可以不带参数运行，这种情况下，当前线程会尝试获得锁，如果所没有被其他线程使用，则申请锁会成功，并立即返回true，如果被其他线程扎弄，再额立即返回false，这种模式不会引起线程等待，因此不会产生死锁

### 1、代码

|  |
| --- |
| **public** **class** TryLock **implements** Runnable {  **public** **static** ReentrantLock *lock1* = **new** ReentrantLock();  **public** **static** ReentrantLock *lock2* = **new** ReentrantLock();  **int** lock;  **public** TryLock(**int** lock) {  **this**.lock = lock;  }  @Override  **public** **void** run() {  **if** (lock == 1) {  **while** (**true**) {  **if** (*lock1*.tryLock()) {  **try** {  **try** {  Thread.*sleep*(500);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **if** (*lock2*.tryLock()) {  **try** {  System.*out*.println(Thread.*currentThread*().getId() + ":My Job done;");  **return**;  } **finally** {  *lock2*.unlock();  }  }  } **finally** {  *lock1*.unlock();  }  }  }  } **else** {  **while** (**true**) {  **if** (*lock2*.tryLock()) {  **try** {  **try** {  Thread.*sleep*(500);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **if** (*lock1*.tryLock()) {  **try** {  System.*out*.println(Thread.*currentThread*().getId() + ":My Job done;");  **return**;  } **finally** {  *lock1*.unlock();  }  }  } **finally** {  *lock2*.unlock();  }  }  }  }  }  /\*\*  \* 上面代码中采用了非常容易死锁的加锁顺序,导致thread1和thread2由于锁的竞争而互相等待从而引起死锁  \*  \* 使用了tryLock后,线程不会一直等待而是不停的尝试去获得锁资源,只需要等待一定的时间,线程最终会获得所需要的资源  \*  \* **@param** args  \*/  **public** **static** **void** main(String args[]) {  TryLock r1 = **new** TryLock(1);  TryLock r2 = **new** TryLock(2);  Thread thread1 = **new** Thread(r1);  Thread thread2 = **new** Thread(r2);  thread1.start();  thread2.start();  }  } |

## 5、公平锁new ReentrantLock(true);

### 解释：

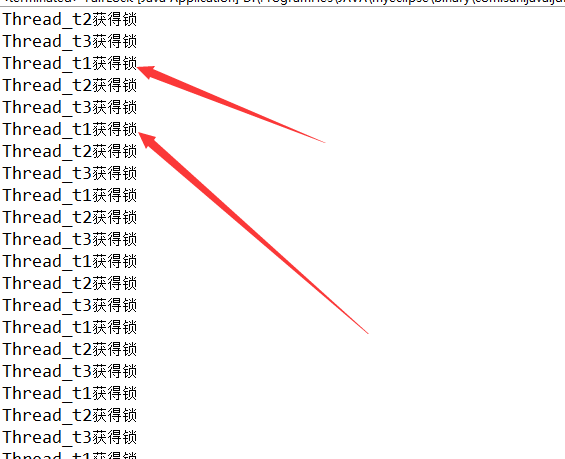
synchronized关键字进行的锁控制，是非公平的，冲入锁，允许我们进行公平性的设置，默认情况下，锁都是非公平的，如果没有特别需求，还是不用公平锁

**public** **static** ReentrantLock *fairLock* = **new** ReentrantLock(**true**);//设置true指定锁是公平的,也可以不设置,分别运行观察公平锁与非公平锁间的区别

### 1、代码

|  |
| --- |
| **public** **class** FairLock **implements** Runnable {  **public** **static** ReentrantLock *fairLock* = **new** ReentrantLock(**true**);//设置true指定锁是公平的,也可以不设置,分别运行观察公平锁与非公平锁间的区别  //public static ReentrantLock unfairLock = new ReentrantLock();  @Override  **public** **void** run() {  **while** (**true**) {  **try** {  *fairLock*.lock();  // unfairLock.lock();  System.*out*.println(Thread.*currentThread*().getName() + "获得锁");  } **finally** {  *fairLock*.unlock();  // unfairLock.unlock();  }  }  }  /\*\*  \* 公平锁的一个特点是:不会产生饥饿现象,只要排队最终都会得到资源.  \* <p/>  \* 但是实现公平锁要求系统维护一个有序队列,因此公平锁的实现成本较高,性能相对低下.  \*  \* **@param** args  \*/  **public** **static** **void** main(String args[]) {  FairLock r1 = **new** FairLock();  Thread thread1 = **new** Thread(r1, "Thread\_t1");  Thread thread2 = **new** Thread(r1, "Thread\_t2");  Thread thread3 = **new** Thread(r1, "Thread\_t3");  thread1.start();  thread2.start();  thread3.start();  }  } |

### 2、控制台，很清晰



## 6、Condition 条件

### 1、和wait和notify有点像

1、await()方法会让当前线程等待，同事释放当前锁，，当其他线程使用signal（或者signalAll（）方法时，先回重新获得锁，并继续执行

2、singal（）方法用于唤醒一个再线程中等待的线程，，这方法在使用的时候，也要求线程，先获得相关锁，然后唤醒线程，在singal之后，要释放掉相关锁，谦让给被唤醒的线程，让它可以继续执行。

|  |
| --- |
| **public** **class** ReenterLockCondition **implements** Runnable {  **public** **static** ReentrantLock *lock* = **new** ReentrantLock();  **public** **static** Condition *condition* = *lock*.newCondition();  @Override  **public** **void** run() {  **try** {  *lock*.lock();  *condition*.await();  System.*out*.println("Thread is going on");  } **catch** (InterruptedException e) {  e.printStackTrace();  } **finally** {  *lock*.unlock();  }  }  **public** **static** **void** main(String args[]) **throws** InterruptedException {  ReenterLockCondition reenterLockCondition = **new** ReenterLockCondition();  Thread thread1 = **new** Thread(reenterLockCondition);  thread1.start();  System.*out*.println("睡眠2秒钟");  Thread.*sleep*(2000);  *lock*.lock();  *condition*.signal();  *lock*.unlock();  }  } |

# 13、读写锁ReadWriteLock

解释：

读写锁可以有效的帮助减少竞争，以提高系统的性能，用锁分离的机制来提升性能，比如线程A1，A2，A3进行写操作，B1，B2,B3进行读操作，这样就能让B1，B2.B3,之间并行，如果系统中读操作次数远远大于写操作，这种方式旧很有用了

读读不互斥，读读不阻塞

写写互斥，写写阻塞，

读写互斥，读阻塞写，写阻塞读

## 1、代码

|  |
| --- |
| **public** **class** ReadWriteLockDemo {  **private** **static** Lock *lock* = **new** ReentrantLock();  **private** **static** ReentrantReadWriteLock *reentrantReadWriteLock* = **new** ReentrantReadWriteLock();  **private** **static** Lock *readLock* = *reentrantReadWriteLock*.readLock();  **private** **static** Lock *writeLock* = *reentrantReadWriteLock*.writeLock();  **private** **int** value;  **public** Object handleRead(Lock lock) **throws** InterruptedException {  **try** {  lock.lock();  Thread.*sleep*(1000);//模拟读操作  System.*out*.println("读操作:" + value);  **return** value;  } **finally** {  lock.unlock();  }  }  **public** **void** handleWrite(Lock lock, **int** index) **throws** InterruptedException {  **try** {  lock.lock();  Thread.*sleep*(1000);//模拟写操作  System.*out*.println("写操作:" + value);  value = index;  } **finally** {  lock.unlock();  }  }  **public** **static** **void** main(String args[]) {  **final** ReadWriteLockDemo demo = **new** ReadWriteLockDemo();  Runnable readRunnable = **new** Runnable() {  @Override  **public** **void** run() {  //分别使用两种锁来运行,性能差别很直观的就体现出来,使用读写锁后读操作可以并行,节省了大量时间  **try** {  demo.handleRead(*readLock*);  //demo.handleRead(lock);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  };  Runnable writeRunnable = **new** Runnable() {  @Override  **public** **void** run() {  //分别使用两种锁来运行,性能差别很直观的就体现出来  **try** {  demo.handleWrite(*writeLock*, **new** Random().nextInt(100));  //demo.handleWrite(lock, new Random().nextInt(100));  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  };  **for** (**int** i = 0; i < 18; i++) {  **new** Thread(readRunnable).start();  }  **for** (**int** i = 18; i < 20; i++) {  **new** Thread(writeRunnable).start();  }  }  } |

# 15、倒计时器CountDownLatch

## 解释：

把门锁起来，不让里面的线程跑出来，这个工具通常用来控制线程等待，它可以让某一个线程等待直到倒计时结束，再开始执行。

下面代码意思就是，等待上面的10个线程全部结束，再执行主线程代码

|  |
| --- |
| **public** **class** CountDownLatchDemo **implements** Runnable {  //计数数量为10，表示需要有10个线程完成任务等待在CountDownLatch 上的线程才能执行  **static** **final** CountDownLatch *end* = **new** CountDownLatch(10);  **static** **final** CountDownLatchDemo *demo* = **new** CountDownLatchDemo();  @Override  **public** **void** run() {  **try** {  System.*out*.println(Thread.*currentThread*().getName());  Thread.*sleep*(**new** Random().nextInt(3) \* 1000);  System.*out*.println("check complete");  *end*.countDown(); //通知CountDownLatch 一个线程已经完成了任务 倒计时可以减一了  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  **public** **static** **void** main(String args[]) **throws** InterruptedException {  ExecutorService executorService = Executors.*newFixedThreadPool*(10); //固定开启10个线程  **for** (**int** i = 0; i < 10; i++) {  executorService.submit(*demo*);  }  //等待检查  *end*.await(); //要求主线程等待所有的10个检查任务全部完成，主线程才能继续运行  //发射火箭  System.*out*.println("Fire!");  executorService.shutdown();  }  } |

# 16、循环栅栏 CyclicBarrier

CyclicBarrier 是一种多线程并发控制工具，和CountDownLatch,相似，但是它更强大，它可以理解为循环栅栏，栅栏就是一种障碍物，比如通常在私人宅邸周围有栅栏，阻止闲杂人等进入，这里当然是用来阻止线程继续运行，要求线程在栅栏处等待前面Cyclec意为循环，也就是说这个计时器可以反复使用，比如，假设计数器设置为10，那么凑齐10个后，计数器归0，紧接着开始凑齐下一批10个线程

# 可见性与原子性

   可见性：一个线程对共享变量的修改，更够及时的被其他线程看到  
   原子性：即不可再分了，不能分为多步操作。比如赋值或者return。比如"a = 1;"和 "return a;"这样的操作都具有原子性。类似"a += b"这样的操作不具有原子性，在某些JVM中"a += b"可能要经过这样三个步骤：  
①　取出a和b  
②　计算a+b  
③　将计算结果写入内存

**(1)Synchronized：保证可见性和原子性**  
    Synchronized能够实现原子性和可见性；在Java内存模型中，synchronized规定，线程在加锁时，先清空工作内存→在主内存中拷贝最新变量的副本到工作内存→执行完代码→将更改后的共享变量的值刷新到主内存中→释放互斥锁。

**(2)Volatile：保证可见性，但不保证操作的原子性**

**[java]**  [copy](http://blog.csdn.net/guyuealian/article/details/52525724)

1. Private **int** Num=0;
2. Num++;//Num不是原子操作

Num不是原子操作，因为其可以分为：读取Num的值，将Num的值+1，写入最新的Num的值。  
    对于Num++;操作，线程1和线程2都执行一次，最后输出Num的值可能是：1或者2  
   【解释】

输出结果1的解释：

当线程1执行Num++;语句时，先是读入Num的值为0，倘若此时让出CPU执行权，线程获得执行，线程2会重新从主内存中，读入Num的值还是0，然后线程2执行+1操作，最后把Num=1刷新到主内存中； 线程2执行完后，线程1由开始执行，但之前已经读取的Num的值0，所以它还是在0的基础上执行+1操作，也就是还是等于1，并刷新到主内存中。所以最终的结果是1

    一般在多线程中使用volatile变量，为了安全，对变量的写入操作不能依赖当前变量的值：如Num++或者Num=Num\*5这些操作。

**(3)Synchronized和Volatile的比较**  
    1）Synchronized保证内存可见性和操作的原子性  
    2）Volatile只能保证内存可见性  
    3）Volatile不需要加锁，比Synchronized更轻量级，并不会阻塞线程（volatile不会造成线程的阻塞；synchronized可能会造成线程的阻塞。）  
    4）volatile标记的变量不会被编译器优化,而synchronized标记的变量可以被编译器优化（如编译器重排序的优化）.  
    5）volatile是变量修饰符，仅能用于变量，而synchronized是一个方法或块的修饰符。  
      volatile本质是在告诉JVM当前变量在寄存器中的值是不确定的，使用前，需要先从主存中读取，因此可以实现可见性。而对n=n+1,n++等操作时，volatile关键字将失效，不能起到像synchronized一样的线程同步（原子性）的效果。

### volatile关键字的作用

一个非常重要的问题，是每个学习、应用多线程的Java程序员都必须掌握的。理解volatile关键字的作用的前提是要理解Java内存模型，这里就不讲Java内存模型了，可以参见第31点，volatile关键字的作用主要有两个：

（1）多线程主要围绕可见性和原子性两个特性而展开，使用volatile关键字修饰的变量，保证了其在多线程之间的可见性，即每次读取到volatile变量，一定是最新的数据

         深入来说：通过加入内存屏障和禁止重排序优化来实现的。

* 对volatile变量执行写操作时，会在写操作后加入一条store屏障指令
* 对volatile变量执行读操作时，会在读操作前加入一条load屏障指令

         通俗地讲：volatile变量在每次被线程访问时，都强迫从主内存中重读该变量的值，而当该变量发生变化时，又会强迫线程将最新的值刷新到主内存。这样任何时刻，不同的线程总能看到该变量的最新值。