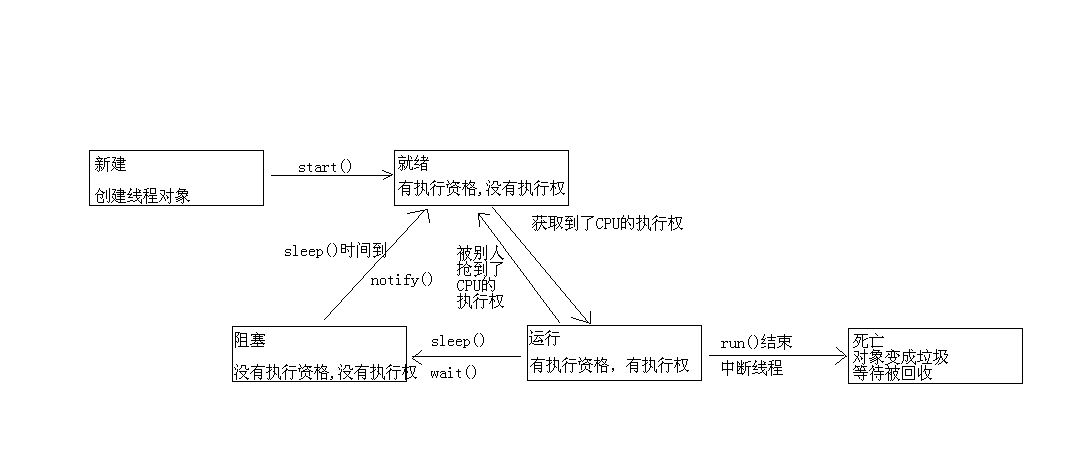
# 线程

创建线程的常用方式为：继承Thread类和实现Runnable接口。



## 线程中断

Java中有三种方法可以终止正在运行的线程：

1. 使用退出标志，是线程正常退出，也就是当run方法完成后线程终止。
2. 使用stop方法强行终止线程，但是不推荐使用这个方法，该方法和suspend，resume方法一样，都是作废过期的方法，使用它们可能产生不可预料的结果。
3. 使用interrupt方法终端线程，该方法不会终止一个正在运行的线程。

判断线程是否停止状态：在java中Thread类提供了两种方法来判断线程是否为状态

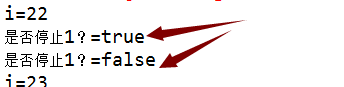
（1）this.interrupted()：测试当前线程是否已经中断,当前线程指的是运行该方法的线程。

（2）this.isInterrupted()：测试线程是否已经终端。

|  |
| --- |
| **public** **class** MyThread **extends** Thread  {  @Override  **public** **void** run() {  **super**.run();  **for**(**int** i =0 ;i<500000;i++)  {  System.***out***.println("i="+(i+1));  **try** {  Thread.*sleep*(100);  } **catch** (Interrupted Exception e) {  e.printStackTrace();  }  }  }  **public** **static** **void** main(String[] args) **throws** Interrupted Exception {  Thread t = **new** Thread(**new** MyThread()) ;  t.start();  Thread.*sleep*(1000);  t.interrupt();  //Thread.currentThread().interrupt();  System.***out***.println("是否停止1？="+t.*interrupted*());  System.***out***.println("是否停止1？="+t.*interrupted*());  /\*System.out.println("是否停止1？="+t.isInterrupted());  System.out.println("是否停止1？="+t.isInterrupted());\*/  }  } |



从运行结果可以看出这个“当前线程”是main线程，所以输入两个false



当main线程调用interrupt方式后，输出结果为一个true和false，这是为啥呢？我们可以从API看出，如果两次调用this.interrupted()方法，则第二次返回true，在第二次调用该方法后可以清楚线程的中断状态。所以interrupted具有清楚状态的功能。

调用isInterrupted()方法运行结果：



**我们可以总结出**：this.interrupted()测试当前线程是否已经是中断状态，执行后具有将状态标志清除为false的功能，而this.isInterrupted()方法是测试Thread对象是否已经是中断状态，但不清楚状态标志。

## 守护线程

Java中有两种线程，一种是：用户线程，另一种是：守护线程。

守护线程是一种特殊的线程，当进程中不存在非守护线程时，则守护线程自动销毁。Java中典型的守护线程时GC线程，当线程中没有非守护线程时，则垃圾回收线程就没有存在的意义了，任何一个守护线程都是JVM中的“保姆”，只有当前JVM实例中存在任何一个非守护线程没有运行结束，守护线程就继续工作，只有当最后一个非守护线程运行结束，守护线程才会随着JVM一同结束。

|  |
| --- |
| **public** **class** MyThread **extends** Thread  {  **private** **int** i = 0 ;  @Override  **public** **void** run() {  **while**(**true**) {  i++;  System.***out***.println("i="+i);  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  }  **public** **static** **void** main(String[] args) **throws** InterruptedException {  Thread t = **new** Thread(**new** MyThread()) ;  //设置守护线程  t.setDaemon(**true**);  t.start();  Thread.*sleep*(5000);  System.***out***.println("----end---");  }  } |

运行结果：设置线程t为守护线程，当main线程执行结束后，则线程t也随着结束了。

## 线程同步

非线程安全是多个线程对同一个对象中的实例变量进行并发访问时，可能产生“脏读”现象，也就是取到的数据其实是被更改过的，线程安全就是获得的实例变量的值是经过同步处理的，不会产生脏读的现象。

非线程安全问题存在于对实例变量的并发访问，如果方法内部有私有变量，则不存在非线程安全问题，方法内部的变量时线程私有所造成。

### 脏读

脏读就是获取不正确的数据。

|  |  |
| --- | --- |
| **public** **class** PublicVar {  **private** String username="A";  **private** String password="AA";  **public** **synchronized** **void** setValue(String un ,String pwd) {  **try** {  **this**.username=un;  Thread.*sleep*(5000);  **this**.password=pwd;  System.***out***.println("setValue threadName="+  Thread.*currentThread*().getName()+  " username="+username+" password="+password);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  **public** **void** getValue() {  System.***out***.println("getValue threadName="+  Thread.*currentThread*().getName()+  " username="+username+" password="+password);  }  **public** **static** **void** main(String[] args) **throws** InterruptedException {  PublicVar pv = **new** PublicVar();  ThreadC tc = **new** ThreadC(pv);  tc.start();  Thread.*sleep*(200);  pv.getValue();  }  }  **class** ThreadC **extends** Thread{  **private** PublicVar pv ;  **public** ThreadC(PublicVar pv) {  **this**.pv=pv ;  }  @Override  **public** **void** run() {  pv.setValue("B", "BB");  }  } |  |

从程序运行结果可以看出，程序发生了脏读，在读取实例变量时，实例变量已经被其他线程修改。

|  |  |
| --- | --- |
| **public** **class** Run  {  **public** **static** **void** main(String[] args){  MyOneList list = **new** MyOneList();  MyThread1 thread1 = **new** MyThread1(list);  thread1.setName("A");  thread1.start();  MyThread2 thread2 = **new** MyThread2(list);  thread2.setName("B");  thread2.start();  Thread.*sleep*(6000);  System.***out***.println("list size="+list.getSize());  }  }  **class** MyOneList  {  **private** List list = **new** ArrayList();  **public** **synchronized** **void** add(String data)  {  list.add(data);  }  **public** **synchronized** **int** getSize()  {  **return** list.size();  }  }  **class** MyService  {  **public** MyOneList addServiceMethod(MyOneList list ,String data)  {  **try**  {  **if**(list.getSize()<1)  {  Thread.*sleep*(2000);  list.add(data);  }  }  **catch**(Exception e)  {  e.printStackTrace();  }    **return** list ;  }  }  **class** MyThread1 **extends** Thread  {  **private** MyOneList list ;  **public** MyThread1(MyOneList list)  {  **this**.list = list;  }  @Override  **public** **void** run()  {  MyService ms = **new** MyService() ;  ms.addServiceMethod(list, "A");  }  }  **class** MyThread2 **extends** Thread  {  **private** MyOneList list ;  **public** MyThread2(MyOneList list)  {  **this**.list = list;  }  @Override  **public** **void** run()  {  MyService ms = **new** MyService() ;  ms.addServiceMethod(list, "B");  }  } | **返回的结果是:list size=2**  **这个结果并不是我们想要的结果。**  由于线程的执行方法的顺序不确定，所以在多线程执行带有分支判断的方法时，容易出现逻辑上的错误，有肯能出现脏读。 |

### Synchronized同步

当我们操作某个方法对非线程安全的变量操作时，为了避免发生“脏读”，我们一般对操作的方法添加关键字synchronized，即对方法进行加锁，只有我们获得锁对象才能执行该方法。

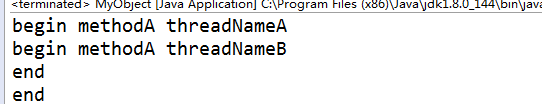
关键字synchronized取得的锁都是对象锁，而不是把一段代码或方法当作锁。

关键字synchronized修改非静态方法，锁对象是当前类的实例。

关键字synchronized修改静态方法，锁对象是该类的字节码。

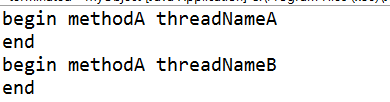
#### Synchronized与锁对象

|  |
| --- |
| **public** **class** MyObject {  **public** /\*synchronized\*/ **void** methodA(){  System.***out***.println("begin methodA threadName"+Thread.*currentThread*().getName());  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.***out***.println("end");  }  **public** **static** **void** main(String[] args) {  MyObject mo = **new** MyObject();  ThreadA a = **new** ThreadA(mo) ;  a.setName("A");  a.start();  ThreadA b = **new** ThreadA(mo) ;  b.setName("B");  b.start();  }  }  **class** ThreadA **extends** Thread{  **private** MyObject object ;  **public** ThreadA(MyObject object) {  **this**.object = object ;  }  @Override  **public** **void** run() {  **super**.run();  object.methodA();  }  }  **class** ThreadB **extends** Thread{  **private** MyObject object ;  **public** ThreadB(MyObject object) {  **this**.object = object ;  }  @Override  **public** **void** run() {  **super**.run();  object.methodA();  }  } |

运行结果：

从结果我们可以看出，当两个线程对同一个非synchronized声明的方法是非同步的。

使用synchronized修饰方法：



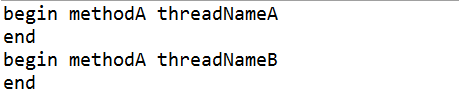
Synchronized声明的方法的调用是同步的。

**总结：**关键字synchronized声明的方法是排队运行的，所以我们可以得到对共享资源的读写访问才需要同步化，如果不是共享资源是不要同步的。

如果我们在类MyObject中添加一个同步方法，代码如下：

|  |
| --- |
| **public** **synchronized** **void** methodB()  {  System.***out***.println("begin methodA threadName"+Thread.*currentThread*().getName());  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.***out***.println("end");  } |

对类ThreadB的run方法调用methodB方法

运行结果：

两个线程调用同一个对象的不同的synchronized方法，当一个线程首先获得锁对象，则其他线程必须等待，直到锁对象被释放。

**总结：**（1）A线程先持有object对象的Lock锁，B线程可以异步的方式调用object对象的非synchronized方法。

（2）A线程先持有object对象的Lock锁，B线程如果调用object对象的synchronized方法，则必须等到A线程释放Lock锁，才能调用。

#### 锁重入

关键字synchronized拥有锁重入的功能，也就是在使用synchronized时，当一个线程得到一个对象锁后，再次请求此对象锁时是可以再次得到该对象的锁。

|  |  |
| --- | --- |
| **public** **class** Service {  **public** **synchronized** **void** service(){  System.***out***.println("service1");  service2();  }  **public** **synchronized** **void** service2() {  System.***out***.println("service2");  service3() ;  }  **public** **synchronized** **void** service3() {  System.***out***.println("service3");  }  **public** **static** **void** main(String[] args) {  Thread t = **new** Thread(**new** ServiceThread()) ;  t.start();  }  }  **class** ServiceThread **extends** Thread{  @Override  **public** **void** run() {  **super**.run();  Service s = **new** Service();  s.service();  }  } |  |

可重入锁：自己可以再次获得自己内部锁。

可重入锁也支持父子类继承的环境。当存在父子类继承关系时，子类是完全可以通过“可重入锁”调用父类的同步方法。

|  |  |
| --- | --- |
| **public** **class** Main **extends** Thread  {  @Override  **public** **void** run() {  Sub sub = **new** Sub();  sub.operateSubMethod();  }  **public** **static** **void** main(String[] args) {  Main m = **new** Main();  m.start();  }  }  **class** Parent  {  **public** **int** i = 10 ;  **public** **synchronized** **void** operateParentMethod()  {  **try** {  i--;  System.***out***.println("parent print i="+i);  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  }  **class** Sub **extends** Parent  {  **public** **synchronized** **void** operateSubMethod() {  **try** {  **while**(i>0) {  i-- ;  System.***out***.println("sub print i="+i);  Thread.*sleep*(1000);  **this**.operateParentMethod();  }  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  } | **运行结果：** |

#### 同步代码块

synchronized方法的弊端：在A线程获得对象锁后，则其他线程必须等待A线程释放锁，才能执行，如果A线程不释放其他线程必须继续等待，这需要耗费大量的时间。

|  |  |
| --- | --- |
| **public** **class** CommonUtils {  **public** **static** **long** *beginTime1*;  **public** **static** **long** *endTime1*;  **public** **static** **long** *beginTime2*;  **public** **static** **long** *endTime2*;  } | **public** **class** Task {  **private** String getData1;  **private** String getData2;  **public** **synchronized** **void** doLongTimeTask() {  **try** {  System.***out***.println("begin task");  Thread.*sleep*(3000);  getData1 = "长时间处理任务后从远程返回的值 1 ThreadName="+  Thread.*currentThread*().getName();  getData2 = "长时间处理任务后从远程返回的值 2 ThreadName="+  Thread.*currentThread*().getName();  System.***out***.println(getData1);  System.***out***.println(getData2);  System.***out***.println("end");  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  } |
| **public** **class** MyThreadA **extends** Thread{  **private** Task task;  **public** MyThreadA (Task task) {  **this**.task = task;  }  @Override  **public** **void** run() {  **super**.run();  CommonUtils.*beginTime1* = System.*currentTimeMillis*();  task.doLongTimeTask();  CommonUtils.*endTime1* = System.*currentTimeMillis*();  } | **public** **class** MyThreadB **extends** Thread{  **private** Task task;  **public** MyThreadB (Task task) {  **this**.task = task;  }  @Override  **public** **void** run() {  **super**.run();  CommonUtils.*beginTime2* = System.*currentTimeMillis*();  task.doLongTimeTask();  CommonUtils.*endTime2* = System.*currentTimeMillis*();  } |
| **public** **class** Test {  **public** **static** **void** main(String[] args){  Task task = **new** Task();  MyThreadA myThreadA = **new** MyThreadA(task);  myThreadA.start();  MyThreadB myThreadB = **new** MyThreadB(task);  myThreadB.start();  **try** {  Thread.*sleep*(10000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **long** beginTime = (CommonUtils.*beginTime2* <CommonUtils.*beginTime1*?  CommonUtils.*beginTime2*:CommonUtils.*beginTime1*);  **long** endTime = (CommonUtils.*endTime2*> CommonUtils.*endTime1*?  CommonUtils.*endTime2*:CommonUtils.*endTime1*);  System.***out***.println("耗时"+ (endTime- beginTime)/1000);  }  } |  |

同步代码块运行的时间：修改Task代码

|  |  |
| --- | --- |
| **public** **class** Test {  **public** **static** **void** main(String[] args){  Task task = **new** Task();  MyThreadA myThreadA = **new** MyThreadA(task);  myThreadA.start();  MyThreadB myThreadB = **new** MyThreadB(task);  myThreadB.start();  **try** {  Thread.*sleep*(10000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **long** beginTime = (CommonUtils.*beginTime2* <CommonUtils.*beginTime1*?  CommonUtils.*beginTime2*:CommonUtils.*beginTime1*);  **long** endTime = (CommonUtils.*endTime2*> CommonUtils.*endTime1*?  CommonUtils.*endTime2*:CommonUtils.*endTime1*);  System.***out***.println("耗时"+ (endTime- beginTime)/1000);  }  } |  |

程序运行的时间缩短了原来的一半。

Synchronized代码块间的同步性：在使用同步synchronized(this)代码块时，当一个线程访问object的一个synchronized(this)同步代码块时，其他线程对同一个object中所有其他synchronized(this)同步代码块的访问将被阻塞，这说明synchronized使用同一个“对象监听器”

多个线程调用同一个对象中的不同名称的synchronized同步方法或synchronized(this)同步代码块，调用的效果就是按顺序执行，也就是同步的，阻塞的。

Synchronized同步方法：（1）同一时间只要一个 线程可以执行synchronized同步方法中的代码块。

（2）对其他synchronized同步方法或synchronized(this)同步代码块调用呈堵塞状态。

Synchronized(this)同步方法：（1）对其他synchronized同步方法或synchronized(this)同步代码块调用呈堵塞状态。

（2）同一时间只要一个 线程可以执行synchronized(this)同步方法中的代码块。

使用synchronized(非this对象x)同步代码块格式进行同步操作时，对象监听器必须是同一个

对象，如果不是同一个对象监听器，运行的结果就是异步调用了，就会交叉运行。

### Volatile

|  |
| --- |
| **public** **class** PrintString3  {  **public** **static** **void** main(String[] args) **throws** InterruptedException {  RunThread rt = **new** RunThread();  rt.start();  Thread.*sleep*(2000);  rt.setRunning(**false**);  System.***out***.println("已经被赋值为false");  }  }  **class** RunThread **extends** Thread  {  **private** /\*volatile \*/ **boolean** isRunning = **true** ;  **public** **boolean** isRunning() {  **return** isRunning ;  }  **public** **void** setRunning(**boolean** isRunning) {  **this**.isRunning = isRunning;  }  @Override  **public** **void** run() {  System.***out***.println("进入run");  **while**(isRunning == **true**) {}  System.***out***.println("线程被停止了");  }  } |

在JVM被设置为-server模式下，线程一直从私有堆栈中去isRunning的值时true，而代码中rt.setRunning(true)虽然执行了，但是更新的一直是公共堆栈中的isRunning变量值为false，所以程序一直就是死循环的转态。

关键字volatile的作用：强制从公共堆栈中取得变量的值，而不是从线程私有数据栈中取得变量的值。

Volatile关键字增加了实例变量在多个线程之间的可见性，但是volatile不支持原子性。

Volatile和synchronized的比较：（1）volatile是线程同步轻量级实现，所以volatile的性能比synchronized高，并且volatile只能修饰变量，而synchronized可以修饰方式，以及代码块。

（2）多线程访问volatile不会发生阻塞，而synchronized会发生堵塞。

（3）volatile保证了数据的可见性，但不能保证原子性，而synchronized保证原子性，也可

以间接保证可见性。

|  |
| --- |
| **public** **class** Run5  {  **public** **static** **void** main(String[] args) {  MyThread[] ts = **new** MyThread[100];  **for**(**int** i =0;i<100;i++) {  ts[i] = **new** MyThread() ;  }  **for**(**int** i =0;i<100;i++) {  ts[i].start(); ;  }  }  }  **class** MyThread **extends** Thread  {  **public** **volatile** **static** **int** *count* ;  **private** /\*synchronized\*/ **static** **void** addCount()  {  **for**(**int** i =0;i<100;i++) {  *count*++;  }  System.***out***.println("count="+*count*);  }  @Override  **public** **void** run() {  *addCount*();  }  } |
| **Volatile非原子性操作** |

## 线程通信

### 等待/通知机制实现

方法wait()的作用是使当前执行的线程进行等待，wait()方法是Object类的方法，该方法用来将当前线程置入“预执行队列”中，并且在wait()所在的代码行处停止执行，直到接到通知或被中断为止。在调用wait()方法之前，线程必须获得该对象的对象级别锁，即只能在同步方法或同步代码块中调用wait()方法，在执行wait()方法后，当前线程释放锁。在从wait()方法返回前，线程与其他线程竞争重新获得锁。如果调用wait()方法 没有持有适当的锁，则抛出异常。

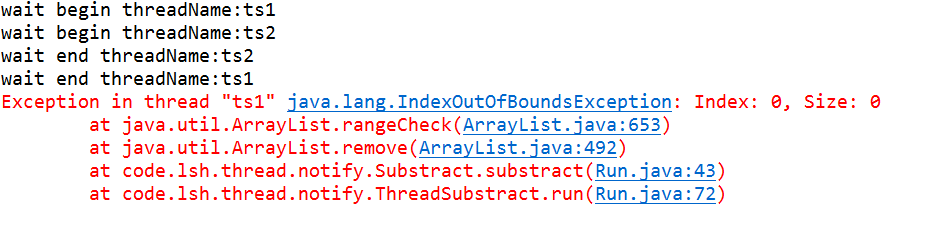
方法notify()在同步方法或同步块中调用，即在调用前，线程也必须获得该对象的对象级别锁。该方法用来通知那些可能等待对象锁的其他线程，如果多个线程等待，则有线程规划器随机挑选出其中一个呈wait状态的线程，对其发出通知notify，并使它等待获取该对象的对象锁，在执行notify方法后，当前线程不会马上释放该对象锁，呈wait状态的线程也不会马上获取该对象锁。

|  |  |
| --- | --- |
| **public** **class** MyList  {  **private** **static** List *list* = **new** ArrayList();  **public** **static** **void** add()  {  *list*.add("anyString");  }  **public** **static** **int** size()  {  **return** *list*.size();  }  **public** **static** **void** main(String[] args)**throws** Exception  {  Object obj = **new** Object();  ThreadA ta = **new** ThreadA(obj) ;  ta.start();  Thread.*sleep*(50);  ThreadB tb = **new** ThreadB(obj) ;  tb.start();  }  }  **class** ThreadA **extends** Thread  {  **private** Object object ;  **public** ThreadA(Object obj)  {  **this**.object = obj;  }  @Override  **public** **void** run() {  **try**  {  **synchronized**(object)  {  **if**(MyList.*size*()!=5)  {  System.***out***.println("wait begin "+System.*currentTimeMillis*());  object.wait();  System.***out***.println("wait end "+System.*currentTimeMillis*());  }  }  }  **catch**(Exception e) {e.getMessage();}  }  }  **class** ThreadB **extends** Thread  {  **private** Object object ;  **public** ThreadB(Object obj)  {  **this**.object = obj;  }  @Override  **public** **void** run() {  **try**  {  **synchronized**(object)  {  **for**(**int** i = 0 ;i<10;i++)  {  MyList.*add*();  **if**(MyList.*size*() == 5)  {  object.notify();  System.***out***.println("已发出通知");  }  System.***out***.println("添加了"+(i+1)+"个元素");  Thread.*sleep*(1000);  }  }  }  **catch**(Exception e) {e.getMessage();}  }  } | **Wait():**调用该方法的线程释放共享资源的锁，然后从运行转态退出，进入等待队列，直到被再次唤醒。  **notify():**可以随机唤醒等待队列中等待同一共享资源的一个线程，并使该线程退出等待队列，进入可运行转态。  **notifyAll()；**可以使所有等待队列中等待同一资源共享资源的“全部”线程从等待状态退出，进入可运行转态。 |

**线程运行转态转化图：**

**等待wait的条件发生变化：**

|  |
| --- |
| package code.lsh.thread.notify;  import java.util.ArrayList;  import java.util.List;  class Add  {  private String lock ;  public Add(String lock)  {  this.lock = lock ;  }  public void add()  {  synchronized(lock)  {  ValueObject.list.add("anyString") ;  lock.notifyAll();  }  }  }  class Substract  {  private String lock ;  public Substract(String lock)  {  this.lock = lock ;  }  public void substract()  {  synchronized(lock)  {  if/\*while\*/(ValueObject.list.size()==0)  {  System.out.println("wait begin threadName:"+Thread.currentThread().getName());  try {  lock.wait();  } catch (InterruptedException e) {  e.printStackTrace();  }  System.out.println("wait end threadName:"+Thread.currentThread().getName());  }  ValueObject.list.remove(0) ;  }  }  }  class ValueObject  {  public static List list = new ArrayList();  }  class ThreadAdd extends Thread  {  private Add add;  public ThreadAdd(Add add)  {  this.add = add ;  }  @Override  public void run() {  add.add();  }  }  class ThreadSubstract extends Thread  {  private Substract sub;  public ThreadSubstract(Substract sub)  {  this.sub = sub ;  }  @Override  public void run() {  sub.substract();  }  }  public class Run  {  public static void main(String[] args) throws Exception{  String lock = new String("");  Add add =new Add(lock);  Substract sub = new Substract(lock);  ThreadSubstract ts =new ThreadSubstract(sub);  ts.setName("ts1");  ts.start();  ThreadSubstract ts2 =new ThreadSubstract(sub);  ts2.setName("ts2");  ts2.start();  Thread.sleep(1000);  ThreadAdd ta = new ThreadAdd(add) ;  ta.setName("ta");  ta.start();  }  } |



出现异常的原因是因为两个实现删除remove()操作的线程，它们在Thead.sleep(1000);之前都执行wait方法，呈等待状态，当加操作的线程在1秒之后被运行后时，通知了所有呈wait等待状态的减操作的线程，第一个实现减操作的线程能正确执行减操作，然而第二个实现减操作的线程则出现索引异常，因为list中只有一个数据，也只能删除一个数据，没有第二个数据可以删除。我们可以使用while循环解决这种问题，当加操作执行完毕并释放锁后，唤所有等待的线程，实现减操作线程会先获得锁，然后对list集合大小进行判断如果list的大小小于0，则继续等待，否则执行减操作。

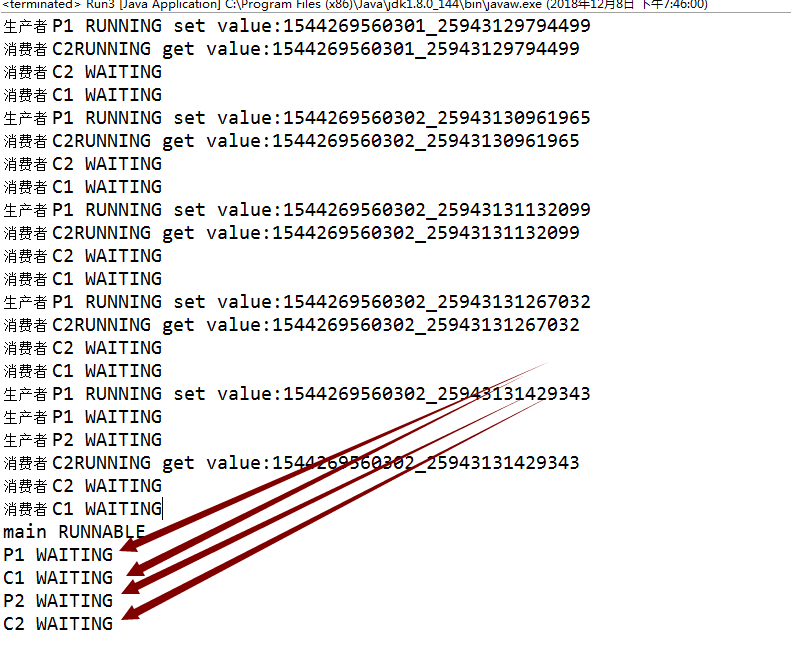
#### 消费者/生产者

##### 一生产与一消费

|  |  |
| --- | --- |
| **public** **class** P  {  **private** String lock ;  **public** P(String lock)  {  **this**.lock = lock ;  }  **public** **void** setValue()  {  **synchronized**(lock)  {  **if**(!ValueObject2.*value*.equals(""))  {  **try** {  lock.wait();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  String value = System.*currentTimeMillis*()+"\_"+System.*nanoTime*();  System.***out***.println("set value:"+value);  ValueObject2.*value*=value;  lock.notify();  }  }  } |  |
| **public** **class** C  {  **private** String lock ;  **public** C(String lock)  {  **this**.lock = lock ;  }  **public** **void** getValue()  {  **synchronized**(lock)  {  **if**(ValueObject2.*value*.equals(""))  {  **try** {  lock.wait();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  System.***out***.println("get value:"+ValueObject2.*value*);  ValueObject2.*value*="";  lock.notify();  }  }  } |  |
| **public** **class** Run2  {  **public** **static** **void** main(String[] args)  {  String lock = **new** String();  P p = **new** P(lock);  C c = **new** C(lock);  ThreadP tp = **new** ThreadP(p);  ThreadC tc = **new** ThreadC(c);  tp.start();  tc.start();  }  }  **class** ThreadP **extends** Thread  {  **private** P p ;  **public** ThreadP(P p)  {  **this**.p = p;  }  @Override  **public** **void** run() {  **while**(**true**)  {  p.setValue();  }  }  }  **class** ThreadC **extends** Thread  {  **private** C c ;  **public** ThreadC(C c)  {  **this**.c = c ;  }  @Override  **public** **void** run() {  **while**(**true**)  {  c.getValue();  }  }  } |  |

##### 多生产与多消费

|  |
| --- |
| **public** **class** P2  {  **private** String lock ;  **public** P2(String lock)  {  **this**.lock = lock ;  }  **public** **void** setValue()  {  **synchronized**(lock)  {  **while**(!ValueObject2.*value*.equals(""))  {  **try** {  System.***out***.println("生产者 "+Thread.*currentThread*().getName()+" WAITING");  lock.wait();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  String value = System.*currentTimeMillis*()+"\_"+System.*nanoTime*();  System.***out***.println("生产者 "+Thread.*currentThread*().getName()+" RUNNING "+"set value:"+value);  ValueObject2.*value*=value;  lock.notify();  }  }  } |
| **public** **class** Run3  {  **public** **static** **void** main(String[] args) **throws** InterruptedException  {  String lock = **new** String();  P2 p = **new** P2(lock);  C2 c = **new** C2(lock);  ThreadP2[] tp = **new** ThreadP2[2];  ThreadC2[] tc = **new** ThreadC2[2];  **for**(**int** i = 0 ;i<2 ;i++)  {  tp[i]= **new** ThreadP2(p);  tp[i].setName("P"+(i+1));  tc[i]=**new** ThreadC2(c);  tc[i].setName("C"+(i+1));  tp[i].start();  tc[i].start();  }  Thread.*sleep*(5000);  Thread[] ts = **new** Thread[Thread.*currentThread*().getThreadGroup().activeCount()];  Thread.*currentThread*().getThreadGroup().enumerate(ts);  **for**(**int** i = 0;i<ts.length;i++)  {  System.***out***.println(ts[i].getName()+" "+ts[i].getState());  }  }  }  **class** ThreadP2 **extends** Thread  {  **private** P2 p ;  **public** ThreadP2(P2 p)  {  **this**.p = p;  }  @Override  **public** **void** run() {  **while**(**true**)  {  p.setValue();  }  }  }  **class** ThreadC2 **extends** Thread  {  **private** C2 c ;  **public** ThreadC2(C2 c)  {  **this**.c = c ;  }  @Override  **public** **void** run() {  **while**(**true**)  {  c.getValue();  }  }  } |



从程序的运行结果上可以看出程序呈现死状态，所有的线程进入“WAITING”转态，这是一种“假死”状态。

**分析**：在代码中确实已经通过wait/notify进行通信，但不保证notify唤醒的是异类，也许是同类，比如“消费者”唤醒“消费者”，或“生产者”唤醒“生产者”，从而导致程序无法正产运行，最后程序呈现“假死”状态。

**“假死”出现的原因主要因为是可能连续唤醒了同类。**

解决“假死”状态很简单，只要使用notifyAll()代替notify();

##### 多生产与一消费

程序不会出现任何异常，正常运行

##### 一生产与多消费

易出现“假死”状态，对条件的判断，需要使用while。

### 管道进行线程通信

Java中管道流（pipeStream）是一种特殊的流，用于在不同线程间进行直接传送数据。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中提供了四个类使线程间可以通信：

1. PipedInputStream和PipedOutStream
2. PipedReader和PipedWriter

|  |
| --- |
| **public** **class** Run  {  **public** **static** **void** main(String[] args) **throws** Exception  {  WriteData wd = **new** WriteData();  ReadData rd = **new** ReadData();  PipedInputStream pis = **new** PipedInputStream();  PipedOutputStream pos = **new** PipedOutputStream();  pos.connect(pis);//输入流连接输出流  ThreadWrite tw = **new** ThreadWrite(wd, pos);  tw.start();  Thread.*sleep*(3000);  ThreadRead tr = **new** ThreadRead(rd, pis);  tr.start();  }  }  **class** WriteData  {  **public** **void** writeMethod(PipedOutputStream out)  {  System.***out***.println("write start");  **try** {  **for**(**int** i = 0;i<300;i++)  {  String data = ""+(i+1);  **try** {  out.write(data.getBytes());  } **catch** (IOException e) {  e.printStackTrace();  }  }  out.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  }  **class** ReadData  {  **public** **void** readMethod(PipedInputStream input)  {  System.***out***.println("read start");  **byte**[] byteArray = **new** **byte**[3];  **int** readLen ;  **try** {  **while**(-1!=(readLen=input.read(byteArray)))  {  String data = **new** String(byteArray,0,readLen);  System.***out***.println(data);  }  input.close();  } **catch** (IOException e) {  e.printStackTrace();  }    }  }  **class** ThreadWrite **extends** Thread  {  **private** WriteData wd ;  **private** PipedOutputStream pos ;  **public** ThreadWrite(WriteData wd , PipedOutputStream pos)  {  **this**.wd = wd ;  **this**.pos = pos ;  }  @Override  **public** **void** run()  {  wd.writeMethod(pos);  }  }  **class** ThreadRead **extends** Thread  {  **private** ReadData rd ;  **private** PipedInputStream pis ;  **public** ThreadRead(ReadData rd , PipedInputStream pis)  {  **this**.rd = rd ;  **this**.pis = pis ;  }  @Override  **public** **void** run()  {  rd.readMethod(pis);  }  } |

### 方法join的使用

在很多的情况下，主线程创建并启动子线程，如果子线程中要进行大量的耗时运算，主线程将早于子线程结束之前结束。如果主线程想等待子线程执行完毕后再结束，就要使用join()方法了。

|  |
| --- |
| **public** **class** MyThread **extends** Thread  {  @Override  **public** **void** run() {  **int** num = (**int**)(Math.*random*()\*10000);  System.***out***.println("num:"+num);  **try** {  Thread.*sleep*(num);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  **public** **static** **void** main(String[] args) **throws** InterruptedException {  MyThread mt = **new** MyThread();  mt.start();  mt.join();  Thread.*sleep*(1000);  System.***out***.println("haoahhh");  }  } |

方法join的作用：是所属的线程对象x正常执行run()方法中的任务，而使当前线程z进行无限期的阻塞，等待线程x销毁后再继续执行线程z后面的线程。

方法join具有是线程排队运行的作用，join在内部使用wait()方法进行等待。

**Join(long)和sleep(long)的区别：**

（1）从源码上看，join(long)释放当前线程的锁，而sleep(long)不释放锁。

# Lock

与使用synchronized方法或代码块相比，Lock的实现提供了更多扩展的锁操作，lock允许更灵活的结构，可以具有完全不同的属性，并且可以支持多个与之关联的Condition对象。

lock是控制多线程访问共享资源的工具，lock提供对共享资源独有的访问，在一个时刻只能有一个线程获得锁，对共享资源的访问必须先获得的锁，然而一些锁可以允许并发访问共享资源，例如：ReadWriteLock

使用synchronized方法或代码块提供了对与之关联对象的隐式监控器锁的访问，但强迫获取锁和释放锁均要在一个阻塞块中，当多个锁被请求，它们必须以相反的顺序释放，每个锁必须在获取它们的相同的语法范围内释放。

虽然synchronized方法或者代码块的范围机制使得使用监控锁编程更加容易，假如在你工作时刻，你需要以更加灵活的方式使用锁，例如，需要使用“手握手”方式或者“”来并发遍历某中数据结构。

Lock接口的实现比使用synchronized方法或代码块提供了额外的功能，通过提供一个非堵塞？？尝试获取锁，尝试获取一个可中断的锁或者尝试获取一个超时失效锁。

## ReentrantLock

一个可重入互斥Lock

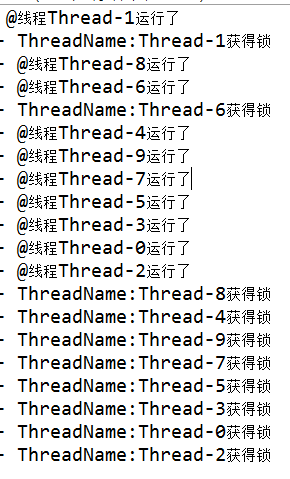
### 公平锁

表示线程获取锁的顺序是按照线程加锁的顺序来分配的，即先来先得FIFO先进先出顺序

|  |
| --- |
| class Server  {  private ReentrantLock lock ;  private Logger logger = Logger.getLogger(RunFair.class) ;  public Server(boolean isFair)  {  super();  lock = new ReentrantLock(isFair);  }  public void serviceMethod()  {  try  {  lock.lock();  logger.debug("ThreadName:"+Thread.currentThread().getName()+"获得锁");  }finally  {  lock.unlock();  }  }  }  public class RunFair  {  private static Logger logger = Logger.getLogger(RunFair.class) ;  public static void main(String[] args)  {  Server server = new Server(true) ;  Runnable runnable = new Runnable() {  @Override  public void run()  {  logger.debug("@线程"+Thread.currentThread().getName()+"运行了");  server.serviceMethod();  }  };  Thread[] threadArray = new Thread[10];  for(int i = 0;i<10;i++)  {  threadArray[i] = new Thread(runnable);  }  for(int i = 0;i<10;i++)  {  threadArray[i].start();;  }  }  } |

### 非公平锁

非公平锁就是一个获取锁的抢占机制，是随机获得锁的，和公平锁不一样的就是先来的不一定先得到锁，这种方式可能造成某些线程一直拿不到锁，结果就是不公平的。



# Condition

Condition将Object中的监听器方法分解成不同的对象，为了有效的给任意实现Lock接口的类相结合。

# Executor

执行提交的Runnable任务的对象，这个接口提供了一种将任务的提交和任务将如何运行的机制分离的方法，包括线程运行的细节和调度。通常使用Exceutor而不是显示的创建一个线程。

Exexutor接口并不要求执行的异步性，在最简单的情况下，executor可以立即执行调用者提交的任务。

## ExecutorService

# Future

Future代表了一个异步执行的结果，它提供了一个用于检测是否执行完成的方法，以等待方法执行完成，并且检索执行的结果。当执行完成后，才能使用get方法检测执行的结果。

# Callable