### 线程概述

#### 1.1概念

进程是计算机中程序关于某个数据集合上的一次运行活动，是操作系统进行资源分配的基本单位

线程是进程的一个执行单元，是CPU调度的基本单位

主线程与子线程

JVM启动时会创建一个主线程，对应执行的是main方法

java中的线程不是孤立的，如果在A线程中创建了B线程，那么，B线程就是A线程的子线程，A线程就是B线程的父线程

串行、并发（concurrent）与并行

这是操作系统中学的

#### 1.2创建线程的方式

1. 继承Thread父类
2. 实现Runnable接口
3. 实现callable接口

演示：1 继承Thread类

步骤：

1新建MyThread类继承Thread类

2在MyThread中重写run()方法

public class MyThread extends Thread{  
 @Override  
 public void run() {  
 System.*out*.println("子线程在运行");  
 }  
}

3新建一个类，创建main方法作为主线程，在里面创建子线程对象

4调用子线程对象的start()方法，开启子线程

public class Test {  
 public static void main(String[] args) {  
 System.*out*.println("mian方法执行主线程");  
 // 创建子线程对象  
 MyThread thread = new MyThread();  
 // 启动子线程  
 thread.start();

System.*out*.println("mian方法输出了一句话");  
 }  
}

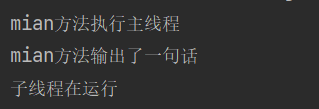
启动线程的实质就是请求JVM运行相应的线程，这个线程具体在什么时候运行由线程调度器Scheduler决定

也就是说，start()方法调用结束并不意味着子线程开始执行了

新开启的线程会执行run()方法

如果开启了多个线程，start()调用的顺序并不一定是线程启动的顺序

输出结果：



演示2：实现Runnable接口

1定义一个类实现Runnable接口

2重写run()方法

public class MyRunnable implements Runnable{  
 @Override  
 public void run() {  
 for (int i = 0; i < 100; i++){  
 System.*out*.println("subThread" + i);  
 }  
 }  
}

3创建一个类，其main方法作为主线程，在其中创建Runnable接口的实现类对象

4创建线程对象

5开启线程，使用start()方法

public class Test {  
 public static void main(String[] args) {  
 MyRunnable runnable = new MyRunnable();  
 Thread thread = new Thread(runnable);  
 thread.start();  
 for (int i = 0; i < 100; i++) {  
 System.*out*.println("main thread:" + i);  
 }  
 }  
}

文档中关于线程的构造方法：



对于Thread(Runnable target)的方式，我们可以使用匿名内部类的方式新建线程对象

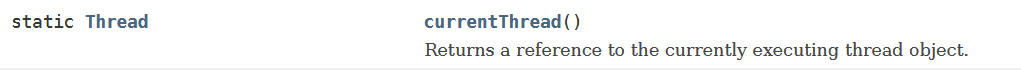
Thread thread2 = new Thread(new Runnable() {  
 @Override  
 public void run() {  
 for (int i = 0; i < 100; i++) {  
 System.*out*.println("subThread2:" + i);  
 }  
 }  
});  
thread2.start();

#### 1.3线程常用的方法

##### 1.3.1 currentThread()

Thread.currentThread()方法可以获得当前线程

这是一个静态方法：



在哪个线程中运行返回的就是哪个线程对象

示例：查看当前线程：

新建一个类继承Thread

public class SubThread01 extends Thread{  
 public SubThread01() {  
 System.*out*.println("sub1的构造方法");  
 }  
  
 @Override  
 public void run() {  
 System.*out*.println("sub1的run方法打印当前线程1名：" + Thread.*currentThread*().getName());  
 }  
}

再建一个类，作为主线程

public class MainThread {  
 public static void main(String[] args) {  
 System.*out*.println("main方法中打印当前线程:" + Thread.*currentThread*().getName());  
 SubThread01 t1 = new SubThread01();  
 t1.start();  
 }  
}

直接说结论：

在方法中只有使用start()方法才是开启子线程，调用run方法相当于是在主线程中运行这个方法体

##### 1.3.2 setName()/getName()

设置线程名称，有助于程序的调试，提高程序可读性

##### 1.3.3 isAlive()

查看线程是否存活

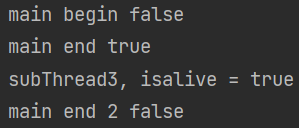
简单的演示：

public class SubThread3 extends Thread{  
 @Override  
 public void run() {  
 System.*out*.println("subThread3, isalive = " + this.isAlive());  
 }  
}

主线程中：

public class Test {  
 public static void main(String[] args) {  
 SubThread3 t3 = new SubThread3();  
 System.*out*.println("main begin " + t3.isAlive());  
 t3.start();  
 System.*out*.println("main end " + t3.isAlive());  
 try {  
 Thread.*sleep*(100);  
 System.*out*.println("main end 2 " + t3.isAlive());  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
}

输出结果：



通过结果我们可以查看线程t3是否存活

##### 1.3.4 Thread.sleep()

静态方法，让当前线程睡眠一定的毫秒数

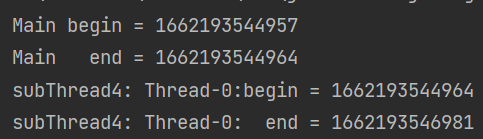
演示：子线程调用sleep方法，同时在睡眠前后打印当前系统时间

public class SubThread4 extends Thread {  
 @Override  
 public void run() {  
 System.*out*.println("subThread4: " + Thread.*currentThread*().getName() +   
 ":begin = " + System.*currentTimeMillis*());  
 try {  
 Thread.*sleep*(2000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 System.*out*.println("subThread4: " + Thread.*currentThread*().getName() +  
 ":end = " + System.*currentTimeMillis*());  
 }  
}

主线程开启子线程，并打印当前系统时间

public class Test {  
 public static void main(String[] args) {  
 SubThread4 t4 = new SubThread4();  
 System.*out*.println("Main begin = " + System.*currentTimeMillis*());  
 t4.start();  
 System.*out*.println("Main end = " + System.*currentTimeMillis*());  
 }  
}

执行结果：



值得一提的是，子类在run()方法中使用Thread.sleep()方法的时候需要对异常进行处理，由于父类Thread是没有对异常进行处理的

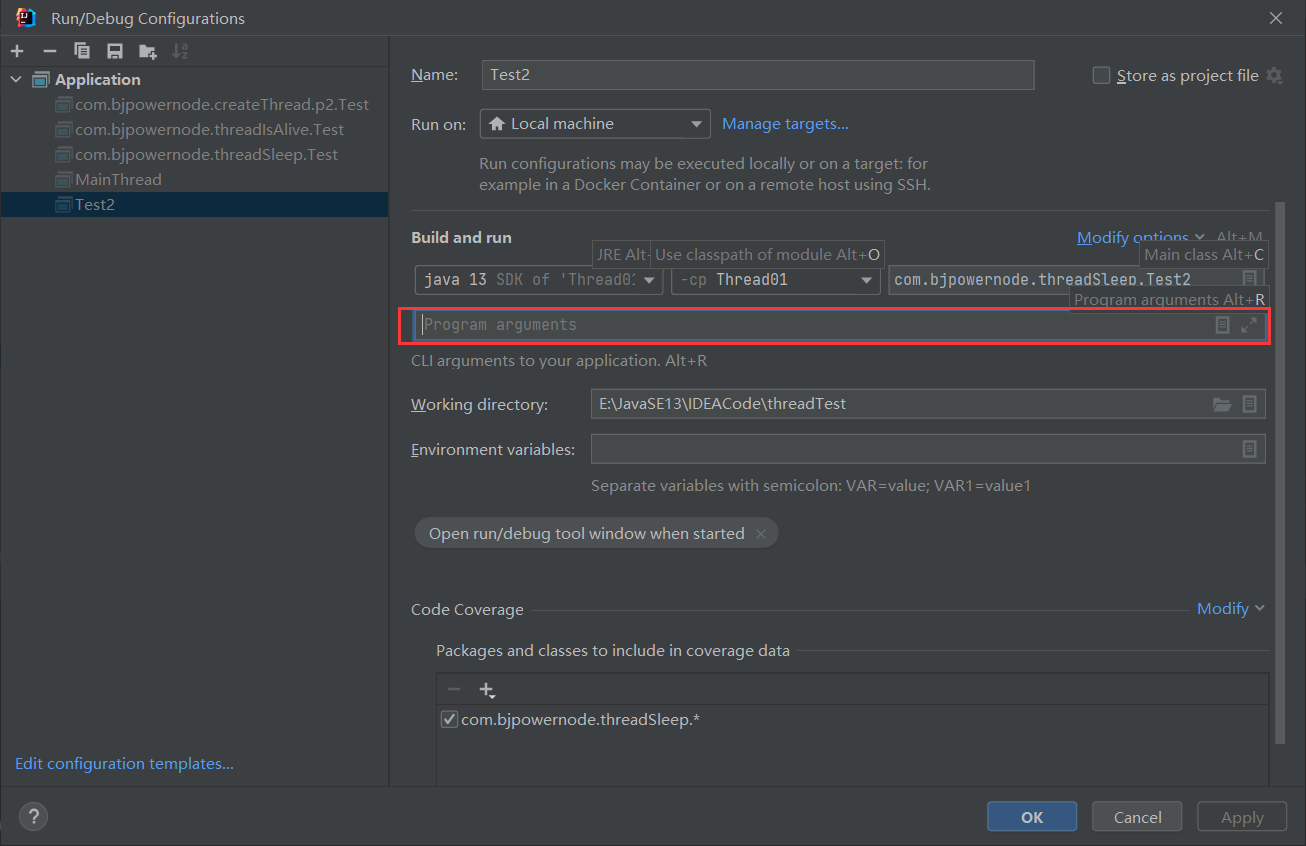
子类抛出的异常不能大于父类抛出的异常，所以run方法不能抛出异常，只能进行try catch捕获

可以使用sleep方法做一个计时器。

通过main方法传参：在下图的Program arguments中输入参数

public class Test2 {  
 public static void main(String[] args) {  
 int remaining = 10;  
 if(args.length == 1){  
 remaining = Integer.*parseInt*(args[0]);  
 }  
 while (remaining > 0){  
 System.*out*.println("Remaining : " +remaining);  
 remaining--;  
 try {  
 Thread.*sleep*(1000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 System.*out*.println("Done !");  
 }  
}

默认计数10秒，若想根据main方法的参数倒计时，就按如下图方式打开设置，从菜单栏中的Run找到下图的窗口



##### 1.3.5 getId()方法

获得当前运行的线程的唯一编号

注意：当某个线程运行结束之后，当前编号可能被后续创建的线程使用

重启JVM之后，同一个线程的编号可能不同

##### 1.3.6 yeild()方法

静态方法，Thread.yeild()——放弃当前的CPU资源

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，注意是让自己或者其他线程运行，并不是单纯的让给其他线程

即回到就绪状态，还是可以参与抢夺时间片的

##### 1.3.7 setPriority()

设置线程的优先级，参数大小为1~10，超出范围抛出异常IllegalArgumentException

在操作系统中，优先级越高的线程获得的CPU资源越多

优先级只是给线程调度器一个提示信息，不是说优先级高的线程一定就会先运行

优先级设置不当可能导致某些线程永远得不到运行，产生线程饥饿

优先级的设置不是越高越好，一般使用默认优先级即可

线程的优先级具有继承性，在A线程中创建了B线程，则B线程的优先级和A线程是一样的

##### 1.3.8 interrupt()方法

给子线程打一个中断的标志，并不会真正的中断子线程

要想让线程真正的退出，可以通过在子线程中判断this.interrupt() == true,然后直接return达到退出的效果

##### 1.3.9 setDaemon()

java中的线程分为用户线程和守护线程

守护线程是为其他线程提供服务的线程，如垃圾回收器

守护线程不能单独运行，当JVM中没有其他的用户线程，只有守护线程的时候，守护线程会自动销毁，JVM退出

需要在start方法之前设置守护线程

#### 1.4 线程的生命周期

线程的生命周期可以通过getState()方法获得

线程的状态是Thread.State枚举类型定义的，有以下几种

NEW新建状态，创建了线程对象，在调用start()方法之前的状态

RUNNABLE，可运行状态，它是一个复合状态，包含：READY和RUNNING两个状态

（1）READY状态下，线程可以被调度器调度，从而变为RUNNING状态，RUNNING状态表示线程正在执行

（2）Thread.yeild()方法可以将线程由RUNNING状态转为READY状态

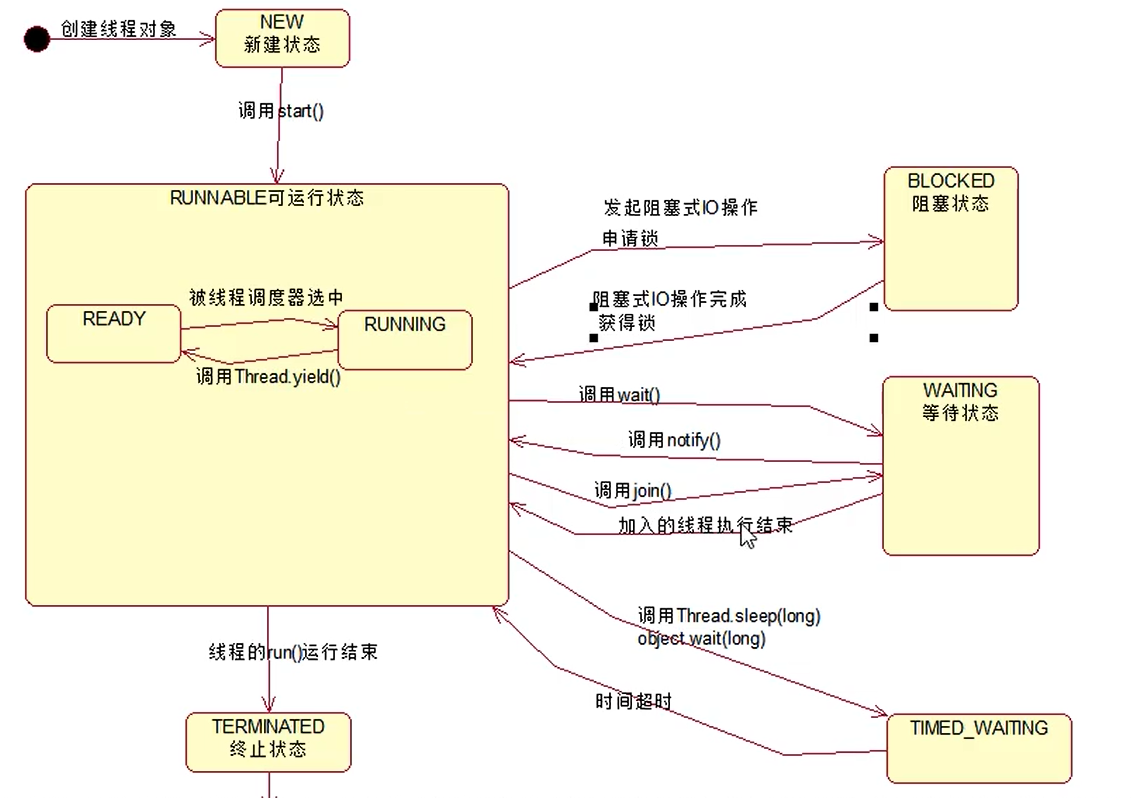
BLOCKED阻塞状态，当线程发起IO操作，或申请由其他线程正在独占的资源，线程会变为BLOCKED阻塞状态。当IO操作结束或线程获得了其申请的资源，线程转为RUNNABLE状态

WAITING等待状态，线程执行了object.wait()，thread.join()方法会进入WAITING方法，之类object.notify()方法，或者加入的线程执行完毕，当前线程会转换为RUNNABLE状态

TIMED\_WAITING状态，与WAITING状态类似，都是等待状态，区别在于处于该状态的线程不会无限的等待，如果线程没有在指定的时间范围内完成期望的操作，该线程自动转为RUNNABLE

TERMINATER终止状态，线程结束就处于终止状态

转换图：



#### 1.5多线程的优势

1. 提高系统的吞吐量（Throughout），多线程编程可以使一个进程有多个并发(concurrent,即同时进行的操作)
2. 提高响应性（Responsiveness）。Web服务会采用一些专门的线程负责用户的请求处理，缩短用户的等待时间
3. 充分利用多核（Multicore）处理器资源，通过多线程可以充分利用CPU资源

#### 1.6 多线程存在的风险

1) 线程安全问题，多线程共享数据时，如果没有采用正确的并发访问控制措施，可能产生数据一致性问题，如读取脏数据（过期的数据），如丢失更新

2) 线程活性问题，由于程序自身的缺陷或者由资源稀缺性导致线程一直处于非RUNNABLE状态，这就是线程的活性问题，常见的活性故障有一下几种：

(1)死锁 Deadlock

(2)锁死 Lockout

(3)活锁 Livelock

(4)饥饿 Starvation

3) 上下文切换，处理器从执行一个线程切换到执行另外一个线程

4) 可靠性，可能会由一个线程导致JVM意外终止，其他线程也无法执行

### 线程安全问题

非线程安全主要是指多个线程对同一个对象的实例变量进行操作时，出现值被更改，值不同步的情况

线程安全问题表现为三个方面：原子性，可见性，有序性

#### 2.1原子性

原子：Atomic 不可分割的意思

原子操作的不可分割有两层含义

1)访问(读，写)某个共享变量的操作要么已经执行完毕，要么尚未发生。即其他线程看不到当前操作的中间过程

2)访问同一组共享变量的原子操作不能交叉的

Java有两种方式实现原子性：一种是使用锁；另一种是利用处理器的CAS（Compare and Swap）指令

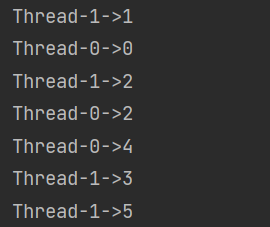
锁具有排他性，保证共享变量在某一时刻只能被一个线程访问

CAS指令2直接在硬件（处理器和内存）层次上实现，看作是硬件锁

演示：

public class Test01 {  
 public static void main(String[] args){  
 //启动两个线程，不断的调用静态内部类的getNum方法  
 MyInt myInt = new MyInt();  
 for (int i = 0; i < 2; i++) {  
 new Thread(new Runnable() {  
 @Override  
 public void run() {  
 while (true){  
 System.*out*.println(Thread.*currentThread*().getName() + "->" + myInt.getNum());  
 try {  
 Thread.*sleep*(200);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 }).start();  
 }  
 }  
  
 static class MyInt{  
 int num;  
 public int getNum(){  
 return num++;  
 }  
 }  
}

其中一段输出结果如下：

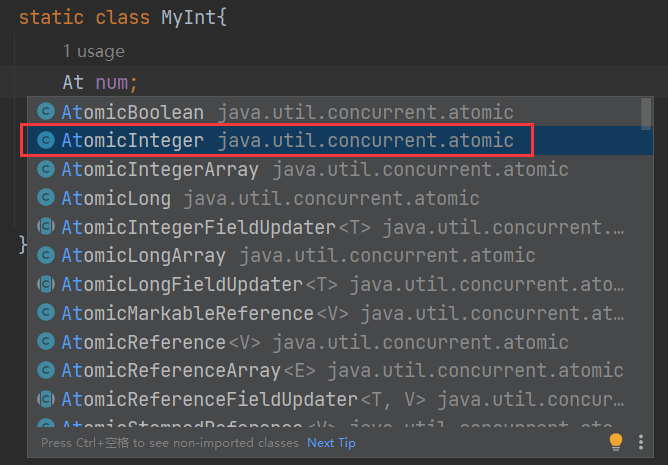


其中有t1获取了num，没来得及修改，t2线程也获取了num，然后他们输出相同数字的情况。

本来应该是t1拿到num，+1之后，t2再拿到num，此时num已经被加过了，t2在原来加的基础上再+1才是理想的结果。

这样是不安全的。

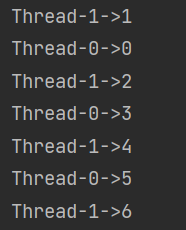
java中提供了一个AtomicInteger类，保证了操作的原子性



将前面的静态内部类中num的类型改为AtomicInteger 类型即可

static class MyInt{  
 AtomicInteger num = new AtomicInteger();  
 public int getNum(){  
 return num.getAndIncrement();  
 }  
}

多线程运行结果：



输出可能顺序不对，但原子性是保证的

#### 2.2 可见性

在多线程环境中，一个线程对某个共享变量进行更新之后，后续其他线程可能无法立即读到这个更新的结果，这就是线程安全问题的另一种形式：可见性

如果一个线程对共享变量更新后，后续访问该变量的其他线程可以读到更新的结果，称这个线程对共享变量的更新对其他线程可见，否则称这个线程对共享变量的更新对其他线程不可见

多线程程序因为可见性问题可能会导致其他线程读到旧数据（脏数据）

以下是这么一个例子，MyTask 类中的run方法中在不停的执行某个任务，这个任务就是doSomething方法，主线程中会调用cancel();方法，来改变toCancel的值，当toCancel为true的时候，任务就停止执行

public class Test02 {  
 public static void main(String[] args) throws InterruptedException {  
 MyTask myTask = new MyTask();  
 new Thread(myTask).start();  
  
 Thread.*sleep*(1000);  
 myTask.cancel();  
 }  
  
 static class MyTask implements Runnable{  
 private boolean toCancel = false;  
 @Override  
 public void run() {  
 while (!toCancel){  
 if(doSomething()){} //break;  
 }  
// if(toCancel) System.out.println("任务被取消");  
// else System.out.println("任务执行结束");  
 }  
  
 private boolean doSomething(){  
// System.out.println("执行任务中....");  
 try {  
 Thread.*sleep*(new Random().nextInt(1000)); // 模拟执行任务所花费的时间  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 return true;  
 }  
  
 public void cancel(){  
 toCancel = true;  
// System.out.println("收到消息 取消线程的执行");  
 }  
 }  
}

那么在程序执行的时候，可能遇到以下的情况，主线程中调用了task.cancel()方法，但是子线程可能看不到main线程对toCancel的修改，其原因可能是：

1. JIT即时编译器可能会对run方法中while循环进行优化为：

if(toCancel()){

while(true){

if(doSomething()){}

}

}

编译器为了提高效率，而作的优化

这种属于编译器的自作聪明导致的死循环

1. 可能与计算机的存储系统有关，假设两个线程分别运行在两个不同的CPU内核上，运行子线程的CPU内核无法立刻获取运行主线程内核中的数据

#### 2.3有序性

有序性Ordering是指在什么情况下一个处理器上运行的一个线程所执行的内存访问操作在另一个处理器上运行的其他线程看来是乱序的 Out of Order

乱序是指内存访问操作的顺序看起来发生了变化

##### 2.3.1 重排序介绍

重排序：在多核处理器的环境下，编写的顺序结构，这种操作执行的顺序可能是没有保障的——

编译器可能会改变两个操作的先后顺序

处理器可能不会按目标代码的顺序执行

这种一个处理器上执行的多个操作，在其他处理器看来它的顺序与目标代码指定的顺序可能不一样，这种现象称为重排序

重排序是对内存访问有序操作的一种优化，可以在不影响单线程程序正确的情况下提升程序的性能，但是可能对多线程程序的正确性产生影响，即可能导致线程安全问题

重排序问题不是必然出现的

这里介绍一下与内存操作顺序有关的几个概念：

源码顺序，就是源码中指定的内存访问顺序

程序顺序，处理器上运行的目标代码所指定的内存访问顺序

执行顺序，内存访问操作在处理器上的实际执行顺

感知顺序，给定处理器所感知到的该处理器及其他处理器的内存访问操 作顺序

重排序分为指令重排序与存储子系统重排序两种

指令重排序主要由JIT编译器、处理器引起的，指程序的顺序和执行顺序不一样

存储子系统重排序是由高速缓存，写缓冲器引起的，感知顺序与执行顺不一致

##### 2.3.2 指令重排序

当源码顺序与程序顺序不一致，或 程序顺与执行顺序不一致，我们就说发生了指令重排序

指令重排序是一种动作，对指令的顺序做了调整

javac编译器一般不会执行指令重排序，而JIT编译器可能会执行指令重排序

（JIT是JVM虚拟机中的编译器）

处理器也可能执行指令重排序，使执行顺序和程序的编写顺序不一样

##### 2.3.3 存储子系统重排序

存储子系统包括了写缓存器与高速缓存

高速缓存（Cache 一级缓存 二级缓存这种夹在CPU和内存之间的东西）是为了匹配CPU与内存处理速度不匹配而设计的

写缓冲器（Store buffer, Write buffer）用来提高写高速缓存操作的效率

存储子系统重排序和指令重排序的区别：

存储子系统并没有真正的修改指令执行的顺序，而是由于不同硬件的处理速度不同而导致从其他核的CPU角度来看，当前CPU操作顺序看起来发生了变化

存储子系统重排序对象是内存操作的结果

从处理器的角度来看，读内存就是从指定的RAM地址中加载数据到寄存器，成为Load操作

写内存就是把数据存储到指定的地址表示的RAM存储单元中，成为Store

内存重排序有一下四种可能：

LoadLoad重排序，一个处理器先后执行两个读操作L1和L2,在其他处理器感知的顺序是先L2再L1

StoreStore重排序，一个处理器先后执行两个写操作W1和W2,在其他处理器感知的顺序是先W2再W1

LoadStore重排序，一个处理器先执行读内存操作L1再执行写内存操作W1，在其他处理器感知的顺序是先W1再L1

StoreLoad重排序，一个处理器先执行写内存操作W1再执行读内存操作L1，在其他处理器感知的顺序是先L1再W1

内存重排序与处理器的内存架构有关，不同架构的处理器所允许的内存重排序不同

内存重排序可能会导致线程安全问题

##### 2.3.4 貌似串行语义

JIT编译器、处理器、存储子系统是按照一定的规则对指令内存操作的结果进行重排序，给单线程程序造成一种假象，让其认为指令是按照源码的顺序执行的，这种假象叫做貌似串行语义

在多线程环境下就可能出问题

为了保证串行语义，有数据依赖关系的语义不会被重排序，只有不存在数据依赖关系的语句才会被重排序

——什么是数据依赖关系？Data dependency

如两个操作访问同一个变量，一个操作的结果依赖上一个操作的结果，比如一个操作是写操作的时候，这两个操作之间就存在数据依赖关系

举例：int x = 1; int y = x + 1;

y的操作数包含前一条语句的执行结果，这就存在了数据依赖关系

举例：int y = x; int x = 1;

先读x，再更新x，也是存在依赖关系的

举例：x = 1; x = 2;

两个语句同时对一个变量进行写操作，也是存在依赖关系的

如果不存在数据依赖关系，则可能重排序

double p = 40.1;

int quantity = 10;

double sum = p \* quantity;

像p的赋值和quantity的赋值不存在数据依赖关系，就有可能会被重排

存在**控制依赖关系（Control Dependency）**的语句允许重排

一条语句的执行结果可能会决定另一条语句能否被执行，这两条语句就存在控制依赖关系。

如在id语句中允许重排，可能存在处理器先执行if代码块，再判断if条件是否成立

##### 2.3.5保证内存访问顺序性

如何保证重排序要么不发生，要么发生了不会影响多线程的执行顺序？

java中可以使用volatile关键字，synchronize关键字实现有序性

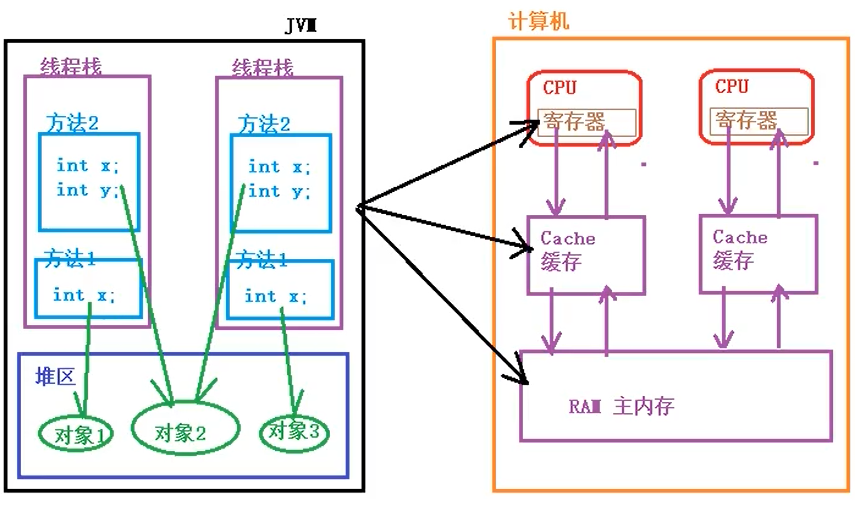
#### 2.4 Java内存模型

java中，堆内存和元空间是不同线程之间共享的

而虚拟机栈、本地方法栈、程序计数器是线程之间相互独立的

每个线程都有独立的栈空间，每个线程都可以访问堆内存

计算机CPU不直接从主内存中读取数据，CPU读取数据时，先把主内存的数据读到Cache缓存中，把Cache中的数据再读到Register寄存器中

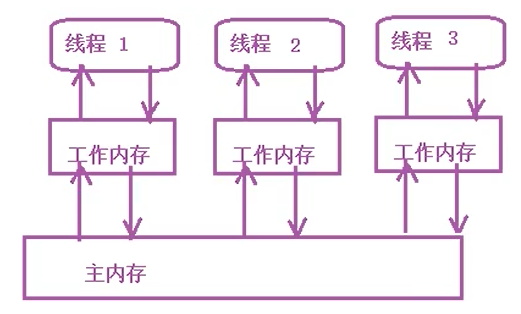


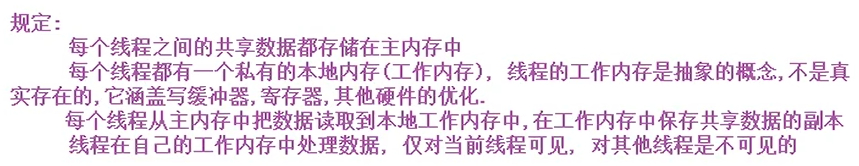
JVM中共享的数据可能会被分配到寄存器中，而一个CPU是不能读取其他CPU寄存器中的内容的，如果两个线程分别运行在不同的CPU上，而共享数据被分配到寄存器上，就会产生可见性问题

当JVM中的数据被分配到主内存中，也不能保证可见性，CPU不直接对主内存访问，而是通过Cache高速缓存进行的，一个处理器上运行的线程对数据的更新可能只是更新到处理器的写缓冲器（Store Buffer），还没达到Cache缓存，根本用说寄存器了。因此，另一个CPU又读不到该处理器写缓冲器上的内存，所以另一个CPU运行的线程看不到这个CPU对数据的更新

一个处理器不能读取另一个处理器的Cache，但是一个处理器可以通过缓存一致性协议读取其他处理器缓存中的数据，并将读取的数据更新到该处理器的Cache中，这个过程称为缓存同步。缓存同步使得一个处理器上运行的线程可以读取到另外一个处理器上运行的线程对共享数据的所做的更新,即保障了可见性.为了保障可见性,必须使一个处理器对共享数据的更新最终被写入该处理器的Cache,这个过程称为冲刷处理器缓存.

将不同CPU之间的寄存器，缓冲器等考虑起来，可以将Java内存模型抽象为下图：





### 3线程同步

#### 3.1线程同步机制

线程同步机制是一套用于协调线程之间的数据访问机制，该机制可以保障线程安全

Java提供了线程同步机制包括：锁，volatile关键字，final关键字，static关键字，以及相关的API，如Object.wait() Object.notify()

#### 3.2 锁概述

线程安全问题产生的前提是多个线程并发访问共享数据

锁的保障线程安全的思路：将多个线程对共享数据的并发访问转换为串行访问，即一个共享数据一次只能被一个线程访问

一个线程，只有持有了锁，才能对共享资源进行访问，一个资源的锁在同一时间只能有1个，访问资源结束之后，要释放锁

获得锁的线程被称为锁的持有线程

临界区：锁的持有线程在获得锁到释放锁的这一段被称为临界区

锁具有排他性，即一个锁只能同时被一个线程持有，这样的锁被称为排它锁或互斥锁

JVM把锁分为

内部锁——通过Synchronize关键字实现

显示锁——通过java.concurrent.locks.Lock接口的实现类实现

##### 3.2.1 锁的作用

实现共享数据的安全访问，保障线程的原子性，可见性与有序性

——通过互斥保障原子性，一个锁同一时间只能被一个线程持有，于是临界区的代码一次只能被一个线程执行，从而临界区就不可分割，就有了原子性

——可见性的保障则是通过写线程冲刷处理器的缓存和读线程刷新处理器缓存 这两个动作实现的。在java平台中，锁的获得隐含着刷新处理器缓存的动作，锁的释放隐含着冲刷处理器缓存的操作

——有序性的保障即写线程在临界区所执行的操作，在读线程的临界区中看起来就是按照源码的顺序执行的

这些线程在访问共享数据时必须使用同一个锁，不论是读或写，都要使用同步锁

##### 3.2.2 锁的相关概念

**（1）可重入性**

可重入性(Reentrancy)描述这样一个问题：一个线程持有该锁的时候，能再次申请该锁

如一个线程执行了两个方法，A,B

A方法申请了一个锁，并调用了B方法

B方法也申请了同一个锁

public void A(){

申请a锁;

B();

释放a锁;

}

public void B(){

申请a锁;

释放a锁;

}

如果能申请成功，称为该锁是可重入的，否则是不可重入的

**（2）锁的竞争和调度**

java平台中内部锁属于非公平锁，显示锁Lock即支持公平锁又支持非公平锁

##### 锁的粒度

一个锁可以保护的共享数据量的大小叫做锁的粒度

一个锁保护的数据量越大，该锁的粒度越粗，反之约细

锁的粒度过粗，会导致申请锁时不必要的等待，锁的粒度过细会导致锁调度不必要的开销

如银行开户——粒度粗：开户销户取钱都在一个窗口

粒度细：开户、销户、取钱开3个窗口

#### 3.3内部锁Synchronize

Java中每个对象都有一个与之关联的内部锁（Intrinsic lock）。这种锁也称为监视器Monitor，这种内部锁是一种排它锁，可以保障原子性，可见性，有序性

内部锁通过Synchronize关键字实现。Synchronize关键字修饰代码块，修饰方法

语法：

synchronize(对象锁){

同步代码块

}

修饰实例方法就称为同步实例方法

修饰静态方法就称为同步静态方法

##### 3.3.1 synchronize同步代码块

不加锁时的演示：

public class Test01 {  
 public static void main(String[] args) {  
 Test01 test01 = new Test01();  
 new Thread(new Runnable() {  
 @Override  
 public void run() {  
 test01.add();  
 }  
 }).start();  
  
 new Thread(new Runnable() {  
 @Override  
 public void run() {  
 test01.add();  
 }  
 }).start();  
 }  
   
 // 打印字符串方法  
 public void add(){  
 for (int i = 0; i < 100; i++) {  
 System.*out*.println(Thread.*currentThread*().getName() + "->" + i);  
 }  
 }  
}

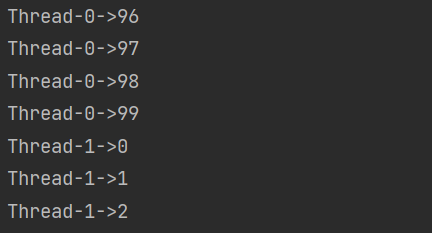
如何而加锁？

使用synchronize代码块

// 打印字符串方法  
public void add(){  
 synchronized (this){  
 for (int i = 0; i < 100; i++) {  
 System.*out*.println(Thread.*currentThread*().getName() + "->" + i);  
 }  
 }  
}

我们使用this作为锁对象，即test01对象就是锁对象

执行结果如下：Thread-0执行完了，释放test01对象，Thread-1才执行



解释：

假设Thread-0先获得CPU执行权，调用test01对象执行add()方法，执行方法时，先获得了锁对象，test01，然后开始执行循环

然后Thread-1获得了CPU执行权，它也调用test01对象的add()方法，要执行synchronize代码块，要先获得test01的锁，但是此时锁被Thread-0持有，所以Thread-1拿不到，所以Thread-1进入等待区等待

当Thread-0重新获得执行权，执行完毕，释放test01的锁，Thread-1就可以从等待区出来获得锁并执行代码。

**也可以使用常量作为锁对象**

##### 3.3.2同步实例方法

同步实例方法默认使用this作为锁对象

public synchronized void add(){  
 for (int i = 0; i < 100; i++) {  
 System.*out*.println(Thread.*currentThread*().getName() + "->" + i);  
 }  
}

一个线程调add方法，一个线程调add2方法，他们锁对象都是this，所以他们也能同步。代码如下：

public class Test03 {  
 public static void main(String[] args) {  
 Test03 test01 = new Test03();  
 new Thread(new Runnable() {  
 @Override  
 public void run() {  
 test01.add();  
 }  
 }).start();  
  
 new Thread(new Runnable() {  
 @Override  
 public void run() {  
 test01.add2();  
 }  
 }).start();  
 }  
  
 // 打印字符串方法  
 public synchronized void add(){  
 for (int i = 0; i < 100; i++) {  
 System.*out*.println(Thread.*currentThread*().getName() + "->" + i);  
 }  
 }  
  
 public void add2(){  
 synchronized (this){  
 for (int i = 0; i < 100; i++) {  
 System.*out*.println(Thread.*currentThread*().getName() + "->" + i);  
 }  
 }  
 }  
}

##### 3.3.3 同步静态方法

使用synchronize修饰静态方法，就是同步静态方法，默认运行时类作为锁对象

可以理解为把Test类的字节码作为锁对象

也有人称之为类锁

演示如下：

//同步静态方法的演示  
public class Test04 {  
 public static void main(String[] args) {  
 Test04 test01 = new Test04();  
 new Thread(new Runnable() {  
 @Override  
 public void run() {  
 Test04.*add*();  
 }  
 }).start();  
  
 new Thread(new Runnable() {  
 @Override  
 public void run() {  
 test01.add2();  
 }  
 }).start();  
 }  
  
 // 打印字符串方法  
 public static synchronized void add(){  
 for (int i = 0; i < 100; i++) {  
 System.*out*.println(Thread.*currentThread*().getName() + "->" + i);  
 }  
 }  
  
 public void add2(){  
 synchronized (Test04.class){  
 for (int i = 0; i < 100; i++) {  
 System.*out*.println(Thread.*currentThread*().getName() + "->" + i);  
 }  
 }  
 }  
}

#### 3.3.4 同步方法与同步代码块如何选择？

例子：

使用同步代码块

子线程执行同步代码块之前，先准备3秒，两个子线程可以同时准备这3秒，被锁住的只有循环的部分

public void task(){  
 try {  
 Thread.*sleep*(3000);  
 synchronized (this){  
 for (int i = 0; i < 100; i++) {  
 System.*out*.println(Thread.*currentThread*().getName() + "->" + i);  
 }  
 }  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
}

使用同步方法

方法中的全部内容都会被synchronize锁住，所以准备3秒部分的代码也是被锁住的

public synchronized void task(){  
 try {  
 Thread.*sleep*(3000);  
 for (int i = 0; i < 100; i++) {  
 System.*out*.println(Thread.*currentThread*().getName() + "->" + i);  
 }  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
}

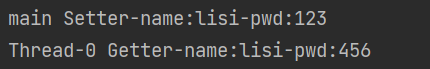
**所以，同步方法锁的粒度粗，并发效率低。同步代码块的并发效率高**

##### 3.3.5 脏读

如下代码

// 脏读  
public class Test05 {  
 public static void main(String[] args) throws InterruptedException {  
 PublicValue publicValue = new PublicValue();  
 SubThread t1 = new SubThread(publicValue);  
 t1.start();  
 Thread.*sleep*(100);  
 publicValue.getValue();  
 }  
 // 定义一个线程类  
 static class SubThread extends Thread{  
 private PublicValue publicValue;  
 public SubThread(PublicValue publicValue) {  
 this.publicValue = publicValue;  
 }  
 @Override  
 public void run() {  
 publicValue.setValue("lisi","456");  
 }  
 }  
  
 // 定义一个静态内部类  
 static class PublicValue{  
 private String name = "zhangsan";  
 private String pwd = "123";  
  
 public void getValue(){  
 System.*out*.println(Thread.*currentThread*().getName() + " Setter-name:" + name + "-pwd:" + pwd);  
 }  
  
 public void setValue(String name, String pwd){  
 this.name = name;  
 // 模拟操作时间  
 try {  
 Thread.*sleep*(1000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 this.pwd = pwd;  
 System.*out*.println(Thread.*currentThread*().getName() + " Getter-name:" + name + "-pwd:" + pwd);  
 }  
 }  
}

读出来的数据是：



lisi被改了，但是由于操作需要时间，对应的密码没来的及改就被读出来了，结果导致读出了一个修改到一半的数据，这就是脏读

所以，要保证操作原子性

##### 3.3.6 线程出现异常的时候会自动释放锁

如线程0和线程1使用同一个共享变量，这个变量被synchronize锁住，当线程0发生异常，会发生什么？是一直锁住还是释放锁

同步过程中出现异常会自动释放锁对象

##### 3.3.7 死锁

多线程时，使用多个锁时，获得锁的顺序不同可能导致死锁

public void run() {  
 if("a".equals(Thread.*currentThread*().getName())){  
 synchronized (*lock1*){  
 System.*out*.println(Thread.*currentThread*().getName() + "获得了lock1");  
 synchronized (*lock2*){  
 System.*out*.println(Thread.*currentThread*().getName() + "获得了lock2");  
 }  
 }  
 }else if("b".equals(Thread.*currentThread*().getName())){  
 synchronized (*lock2*){  
 System.*out*.println(Thread.*currentThread*().getName() + "获得了lock2");  
 synchronized (*lock1*){  
 System.*out*.println(Thread.*currentThread*().getName() + "获得了lock1");  
 }  
 }  
 }  
}

如何避免死锁？当需要获得多个锁的时候，让获取锁的顺序保持一致

#### 3.4轻量级同步机制

##### 3.4.1 volatile的作用

volatile的作用是使变量在多个线程之间可见

volatile可以强制线程从公共内存中行读取变量的值，而不是从工作内存中读取

volatile与synchronize的比较

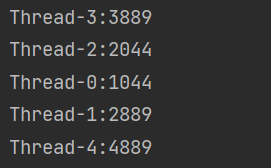
1. volatile是线程同步的轻量级实现，所以性能比synchronize好
2. volatile只能修饰变量，而synchronize可以修饰代码块，方法，随着JDK新版本的发布，synchronize的执行效率也有较大的提升，所以synchronize的使用还是比较多的
3. 多线程访问volatile变量不会发生阻塞，而synchronize可能会导致阻塞
4. volatile能保证数据的可见性，但不能保证原子性；而synchronize可以保证原子性也可以保证可见性
5. 关键字volatile解决的是变量在多个线程之间的可见性，而synchronize关键字解决的是多个线程之间访问公共资源的同步性

##### 3.4.2 volatile不具备原子性

例

// Volatile不具备原子性  
public class VolatileTest3 {  
 public static void main(String[] args) {  
 for (int i = 0; i < 5; i++) {  
 MyThread myThread = new MyThread();  
 myThread.start();  
 }  
 }  
 static class MyThread extends Thread{  
 volatile public static int *count*;  
 public static void addCount(){  
 for (int i = 0; i < 1000; i++) {  
 *count*++;  
 }  
 System.*out*.println(Thread.*currentThread*().getName() + ":" + *count*);  
 }  
 @Override  
 public void run() {  
 *addCount*();  
 }  
 }  
}

输出结果并不是1000的整倍数



以上例子说明了volatile不具备原子性

要想让其同步，还是要使用synchronize修饰addCount()方法，同时，volatile就不再需要了。因为synchronize同时保证了可见性和原子性

##### 3.4.3 常用原子类进行自增自减操作

AtomicInteger——原子类，可以保证原子性

为什么它们能保证操作的原子性？因为它们底层是CAS实现的

多个线程用时对一个AtomicInteger进行加操作

*count*.getAndIncrement();

不会出现脏读

#### 3.5 CAS

CAS Compare And Swap 是由硬件实现的

CAS可以将read modify write 这类的操作转为原子操作

以i++为例，这个操作包含了三个子操作

从主内存读取i的变量值

对i的值+1

把i的值保存到主内存

CAS的原理：在把数据更新到主内存时，再次读取主内存变量的值，如果当前读到的值与期望的值一样，就执行更新操作

CAS概念都知道了，图我就不画了

##### 3.5.1 CAS实现线程安全的计数器

class CASCounter{  
 volatile private long value;  
  
 public long getValue() {  
 return value;  
 }  
  
 // 定义CAS方法  
 private boolean compareAndSwap(long exceptedValue, long newValue){  
 if(value == exceptedValue){  
 value = newValue;  
 return true;  
 }else {  
 return false;  
 }  
 }  
 public long incrementAndGet(){  
 long oldValue;  
 long newValue;  
 do {  
 oldValue = value;  
 newValue = oldValue + 1;  
 }while ( !compareAndSwap(oldValue, newValue));  
 return newValue;  
 }  
}

然后在多个线程中调用下面对象的增加方法，最后的结果是安全的

CASCounter casCounter = new CASCounter();  
System.*out*.println(casCounter.incrementAndGet());

##### 3.5.2 ABA问题

CAS实际上实现原子操作是有一个假设的：共享变量的当前值与当前线程提供的期望值相同，就认为这个变量没有被其他线程修改过

但是如count初始为0

A线程：count = 10

B线程：count = 20

C线程：count = 0

经过多次修改，当前线程看到的还是0

这就CAS中的ABA问题

有时能容忍ABA问题，但有时ABA问题会导致问题

要想规避ABA问题，可以为共享变量引入一个修订号（或时间戳），每次修改共享变量，修订号+1

#### 3.6 原子变量类

原子变量类是基于CAS实现的，当对共享变量进行read modify write操作的时候没听过原子变量类类可以保障操作的原子性与可见性。对变量的 读写改 操作不是一个简单的赋值，而是变量的新值依赖变量的旧值，如自增i++

由于volatile只能保证可见性，无法保证原子性，原子变量的内部就是借助volatile变量，再保障 读写改 操作的原子性 。

有时又把原子变量类当做是增强的volatile变量

原子变量类有12个：

基础数据型原子变量类：AtomicInteger、AtomicLong、AtomicBoolean

数组原子变量类：AtomicIntegerArray, AtomicLongArray, AtomicReferenceArray

引用型原子变量类：AtomicReference, AtomicMarkableReference, AtomicStampedReference

更新器类：AtomicIntegerFieldUpdate, AtomicLongFieldUpdate, AtomicReferenceFieldUpdate

演示：每种演示一个

##### 3.6.1 AtomicLong

演示看具体的代码

##### 3.6.2 AtomicIntegerArray

构造方法AtomicIntegerArray a = new AtomicIntegerArray(10);

新建一个长度为10的数组

常用的方法：set(int a, int b);

将下标为a的元素的值设置为b

getAndSet(1,11)先获得下标为1的元素的值，然后再对下标为1的元素重新赋值

addAndGet(0,22)先给下标为0的元素加22，再读加后的值

getAndAdd(1,32)先读取值然后再做加法

compareAndSet(0,32,222)CAS操作，先比较数组下标为0的元素是不是32，是则赋值其为222

##### 3.6.3 AtomicIntegerFieldUpdate

AtomicIntegerFieldUpdate可以对原子整数型字段进行更新，要求

1. 字符必须使用volatile修饰
2. 只能是实例变量，不能是静态变量，也不能用final修饰

演示看IDEA中的代码

##### 3.6.4 AtomicReference

演示看IDEA中的代码

AtomicReference存在ABA问题

可以使用AtomicStampedReference解决ABA问题

##### 3.6.5 AtomicStampedReference

AtomicStampedReference中有一个整数标记值stamp，每次执行CAS操作的时候，会对比版本

新建对象的时候除了赋值，还要给一个初始版本号

static AtomicStampedReference<String> *atomicReference* = new AtomicStampedReference<>("abc", 0);

### 4 线程之间的通信

#### 4.1 等待/通知机制

##### 4.1.1什么是等待通知机制？

在单线程中，某一操作要满足某条件下才能执行，可以把这个操作放到if中

在多线程当中，B线程修改了某个变量，A线程的某个条件就得到了满足。可以将A线程暂停,等变量修改之后，A的条件满足了再唤醒A线程

伪代码：

atomics{

while(条件不成立){

等待

}

唤醒线程，继续执行后续操作

}

##### 4.1.2 等待/通知机制的实现（面试常考）

Object类中的wait()方法可以使执行当前代码的线程等待，暂停执行，直到接收到通知或被中断为止

注意：

1. wait()方法只能在同步代码块中由锁对象调用
2. 调用wait()方法之后，当前线程会释放锁对象

伪代码：

synchronize(锁对象){

while(条件不成立){

// 通过锁对象调用 wait()方法暂停线程

锁对象.wait();

}

// 线程的条件满足了，就继续向下执行

}

1.Object类的notify()可以唤醒线程，该方法也要在同步代码块中由锁对象使用，没有使用锁对象调用wait()/notify()会抛出illegalMonitorStateException异常

2.如果有多个等待的线程，notify()方法只会唤醒其中的一个，具体唤醒哪一个，不确定

3.同步代码块中调用notify()执行之后，并不会立即释放锁对象，需要等待同步代码块执行完之后才会释放锁对象，所以一般将notify()放在同步代码块的最后

伪代码：

synchronize(锁对象){

// 其他代码

// 唤醒

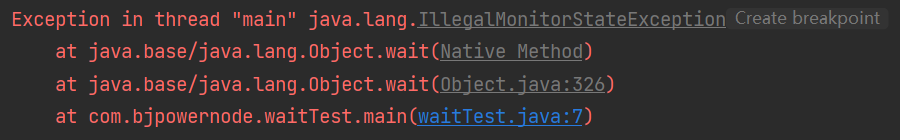
锁对象.notify();

}

演示：

没在同步代码块中使用锁对象.wait()：

public class waitTest {  
 public static void main(String[] args) {  
 String test = "zhangsan";  
 try {  
 test.wait();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
}



正确的使用wait和notify的演示

public class WaitTest3 {  
 public static void main(String[] args) {  
 String text = "zhangsan";  
 Thread t1 = new Thread(new Runnable() {  
 @Override  
 public void run() {  
 synchronized (text){  
 System.*out*.println("线程1开始等待:" + System.*currentTimeMillis*());  
 try {  
 text.wait();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 System.*out*.println("线程1结束等待:" + System.*currentTimeMillis*());  
 }  
 }  
 });  
  
 // 线程1负责唤醒线程1  
 Thread t2 = new Thread(new Runnable() {  
 @Override  
 public void run() {  
 synchronized (text){  
 System.*out*.println("线程2开始唤醒:" + System.*currentTimeMillis*());  
 text.notify();  
 System.*out*.println("线程2结束唤醒:" + System.*currentTimeMillis*());  
 }  
 }  
 });  
  
 t1.start();  
 try {  
 Thread.*sleep*(1000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 t2.start();  
 }  
}

线程等待会导致线程释放锁对象，进入等待状态

##### 4.1.3 notify()方法执行执行不会立即释放锁对象

代码演示略

当前的同步代码块全部执行完之后，才会释放锁对象

##### 4.1.4 interrupt()方法会中断wait()

当前线程处于wait()等待状态的时候，调用线程对象的interrupt()方法会中断线程的等待状态，会产生InterruptedException异常

**4.1.5 Interrupt()**

Interrupt()会中断线程的wait()等待

原来锁对象需要等待同步代码块执行完才会释放、

执行程序代码块过程中，发生了异常，导致线程终止，也会释放锁对象

调用wait()方法也会释放锁对象

**4.1.6 notify()与notifyAll()**

notify()一次只能唤醒一个线程，如果有多个等待的线程，只能随机唤醒其中的一个

要想唤醒所有线程，就要使用notifyAll()

演示：

public class Test02 {  
 public static void main(String[] args) throws InterruptedException {  
 Object lock = new Object();  
 SubThread t1 = new SubThread(lock);  
 SubThread t2 = new SubThread(lock);  
 SubThread t3 = new SubThread(lock);  
 t1.setName("t1"); t2.setName("t2"); t3.setName("t3");  
 t1.start(); t2.start(); t3.start();  
 Thread.*sleep*(1000);  
 synchronized (lock){  
 // lock.notify();  
 lock.notifyAll();  
 }  
 }  
  
 static class SubThread extends Thread{  
 private Object lock;  
 public SubThread(Object lock) {  
 this.lock = lock;  
 }  
 @Override  
 public void run() {  
 synchronized (lock){  
 try {  
 System.*out*.println(Thread.*currentThread*().getName()+"begin wait");  
 lock.wait();  
 System.*out*.println(Thread.*currentThread*().getName()+"end wait");  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 }  
}

**4.1.7 wait(long)的使用**

wait有一个重载，就是wait(long)

如果在参数指定的时间没有被唤醒，超时后就会自动唤醒

**4.1.8 通知过早问题**

线程wiat()等待之后，可以调用notify()唤醒线程。如果notify()唤醒的时机过早，在等待之前就调用了notify()方法，可能会打乱程序正常的运行逻辑

**4.1.9 wait等待的条件发送了变化**

使用wait/notify时，wait条件发生变化，可能会造成逻辑混乱

**4.1.10 生产者消费者模式**

在java中负责产生数据的模块是生产者，负责使用数据的模块是消费者

即先有数据然后才能使用，如果没有数据，消费者就要等待

1. **生产-消费-操作数据**

一个负责产生数据和取数据的方法

public class ValueOperator {  
 private String value = "";  
 // 定义方法修改value的值  
  
 public void setValue() {  
 synchronized (this){  
 if( !value.equalsIgnoreCase("")){  
 try {  
 this.wait();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 String value = System.*currentTimeMillis*() + "-" + System.*nanoTime*();  
 System.*out*.println("set的值：" + value);  
 this.value = value;  
 this.notify();  
 }  
 }  
  
 public void getValue(){  
 synchronized (this){  
 if(value.equalsIgnoreCase("")){  
 try {  
 this.wait();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 System.*out*.println("get的值：" + this.value);  
 this.value = "";  
 this.notify();  
  
 }  
 }  
}

生产者和消费者通过调用以上的set和get方法进行生产和消费

// 生产者  
public class MyProducer extends Thread{  
 private ValueOperator operator = new ValueOperator();  
  
 public MyProducer(ValueOperator operator) {  
 this.operator = operator;  
 }  
  
 @Override  
 public void run() {  
 while (true){  
 operator.setValue();  
 }  
 }  
}

// 消费者  
public class MyConsumer extends Thread{  
 private ValueOperator operator = new ValueOperator();  
  
 public MyConsumer(ValueOperator operator) {  
 this.operator = operator;  
 }  
  
 @Override  
 public void run() {  
 while (true){  
 operator.getValue();  
 }  
 }  
}

然后主线程中开启生产者线程和消费者线程

当只有一个生产者和一个消费者的时候，程序可以顺利的执行

**当有多个生产者消费者的时候**，就可能出现取值取不到的情况

为什么？

因为notify()唤醒的线程是随机的，如t2生产者唤醒了t4消费者，t4消费者取出数据，又唤醒了t5消费者，那我生产者没被唤醒，却唤醒了消费者，这就出问题了

这时就应该修改getValue的代码

synchronized (this){  
 while(value.equalsIgnoreCase("")){  
 try {  
 this.wait();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 System.*out*.println("get的值：" + this.value);  
 this.value = "";  
 this.notify();  
}

setValue方法同理

但此时又会出现新的问题——**假死**

全部的线程都进入了等待状态

——生产者1生产之后唤醒生产者2，生产者2发现有数据的，所以while(条件满足)，所以进入等待，释放了CPU执行权，然后生产者3得到执行权，但立马也进入等待状态，然后又到生产者1得到CPU，然后又发现有数据，所以不生产，进入等待，结果是所有生产者都在等待，没有人唤醒它就无法继续运行

然后消费者同理，消费者1读取了数据之后，数据就为空了，然后它唤醒消费者2，消费者2进入等待。消费者3得到CPU，一运行，也进入等待。CPU又到了消费者1的手上，消费者1一运行，也等待了。

好家伙，全都等待了，没人唤醒

解决方法：将notify()换成notigyAll();

1. **操作栈**

生产者将数据存入list()集合中，消费者从list()集合中取数据

一生产一消费

一生产多消费

多生产多消费

**4.2 通过管道实现线程间的通信**

在java.io中的PipeStream管道流用于在线程之间传送数据，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取数据。相关的类包括：PipedInputStream和PipedOutputStream，PipedReader和PipedWriter

举例：

public class Test {  
 public static void main(String[] args) throws IOException {  
 PipedInputStream inputStream = new PipedInputStream();  
 PipedOutputStream outputStream = new PipedOutputStream();  
 inputStream.connect(outputStream);  
  
 new Thread(new Runnable() {  
 @Override  
 public void run() {  
 Test.*writeData*(outputStream);  
 }  
 }).start();  
  
 new Thread(new Runnable() {  
 @Override  
 public void run() {  
 Test.*readDate*(inputStream);  
 }  
 }).start();  
 }  
 // 向管道流中写数据  
 public static void writeData(PipedOutputStream out){  
 try {  
 for (int i = 0; i < 100; i++) {  
 String data = ""+i;  
 out.write(data.getBytes());  
 }  
 out.close();// 关闭流  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
  
 // 向管道流中读取数据  
 public static void readDate(PipedInputStream in){  
 byte[] bytes = new byte[1024];  
 try {  
 int len = in.read(bytes);  
 while (len != -1){  
 System.*out*.println(new String(bytes,0,len));  
 len = in.read(bytes);  
 }  
 } catch (IOException e) {  
 e.printStackTrace();  
 } finally {  
 try {  
 in.close();  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}

**4.3 Join()**

**4.4 ThreadLocal的使用**

除了控制资源的访问，还可以通过增加资源来保证线程安全

ThreadLocal主要解决为每个线程绑定自己的值

这么解释：我现在有1只笔，有100个小朋友要用，这样就会导致每个小朋友都使用笔都要抢，就会导致异常的发生

使用ThreadLocal，相当于每个小朋友都给了一支笔，这样就不会发生异常

**4.4.1 ThreadLocal指定初始值**

如果不设置，默认的ThreadLocal的初始值为null

为ThreadLocal赋初值挺麻烦的，一种方案是：

1. 定义ThreadLocal的子类
2. 重写initialValue方法，设置初始值

public class Test3 {  
 static class SubThreadLocal extends ThreadLocal<Date>{  
 @Override  
 protected Date initialValue() {  
// return new Date();  
 return new Date(System.*currentTimeMillis*() - 15 \* 60 \* 1000);// 15分钟前  
 }  
 }  
}

**5、Lock显示锁**

jdk5中增加了Lock锁接口，有ReentrantLock实现类，ReentrantLock锁称为可重入锁，功能比synchronized多

**5.1 锁的可重入性**

当一个线程获得一个对象锁后，再次请求给对象锁时，是可以获得该对象锁的

演示：

public class Test {  
 public synchronized void m1(){  
 System.*out*.println("同步方法1");  
 m2();  
 }  
  
 public synchronized void m2(){  
 System.*out*.println("同步方法2");  
 m3();  
 }  
  
 public synchronized void m3(){  
 System.*out*.println("同步方法3");  
 }  
  
 public static void main(String[] args) {  
 Test test = new Test();  
 new Thread(new Runnable() {  
 @Override  
 public void run() {  
 test.m1();  
 }  
 }).start();  
 }  
}

线程m1方法锁的是this，m2也会锁this，m3也会锁this。一层一层的都锁着，他们拿的是同一个锁，但是程序能正常执行，这就是锁的可重入性。如果锁不能重入的话，就会造成死锁

Reentrantlock就是可重入的锁

**5.2 Reentrantlock**

**5.2.1 Reentrantlock的基本使用方法**

Reentrantlock使用lock()获得锁，使用unlock()方法释放锁

如：

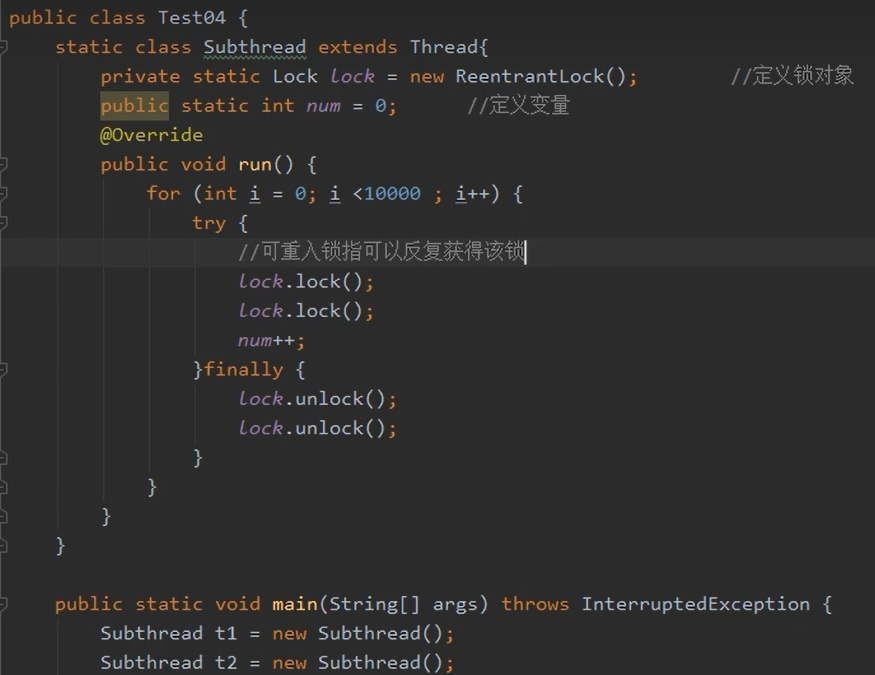
public class Test2 {  
 // 创建显示锁  
 static Lock *Lock* = new ReentrantLock();  
 public static void m1(){  
 *Lock*.lock(); // 获得锁对象  
 for (int i = 0; i < 100; i++) {  
 System.*out*.println(Thread.*currentThread*().getName() + " :" + i);  
 }  
 *Lock*.unlock();// 释放锁对象  
 }  
  
 public static void main(String[] args) {  
 Runnable r = new Runnable() {  
 @Override  
 public void run() {  
 *m1*();  
 }  
 };  
 // 启动3个线程  
 Thread t1 = new Thread(r);  
 Thread t2 = new Thread(r);  
 Thread t3 = new Thread(r);  
   
 t1.start();  
 t2.start();  
 t3.start();  
 }  
}

这种方式，t1先获得锁，当t1执行完之后，下一个线程才能继续执行，类似于同步代码块的功能

不管在哪个方法中使用锁，只要锁对象是同一个，就可以使用锁进行同步

演示详见IDEA代码

可重入锁的演示：懒得敲了，直接看图：

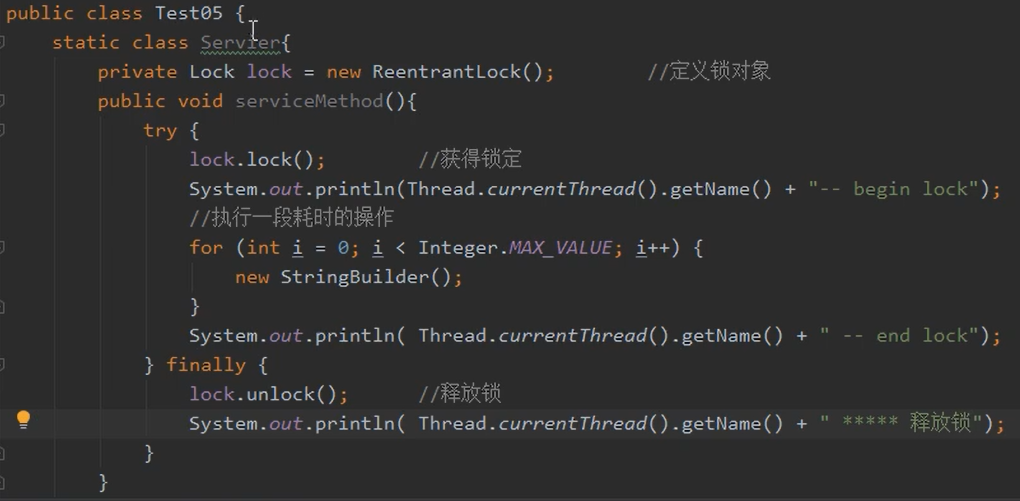


一个线程获得了Lock还可以再次获得这个锁，但是相应的，你得释放2次这个Lock锁

**5.2.2 lockInterruptibly()**

lockInterruptibly()方法的作用：如果当前线程未被中断，则获得锁，如果当前线程被中断则出现异常

演示：



在main方法中新建线程调用serviceMethod方法

并在t2线程开始后50ms给t2线程打上一个中断标志



结果：t2没有被中断

现在修改代码：

lock.lock()改为了lock.lockInterruptibly();



再次运行main方法：

产生异常，说明使用lockInterruptibly();之后，给线程打上中断标记，会立即中断线程

对于synchronized内部锁来说，如果一个线程在等待锁，只有2个结果：要么获得锁，然后继续执行，要么保持等待

而对于ReentrantLock可重入锁来说，提供另一种可能，在等待锁的过程中，程序可以根据需要取消对锁的请求

演示：

public class Test5 {  
 static class IntLock implements Runnable{  
 public static ReentrantLock *lock1* = new ReentrantLock();  
 public static ReentrantLock *lock2* = new ReentrantLock();  
 int lockNum;  
  
 public IntLock(int lockNum) {  
 this.lockNum = lockNum;  
 }  
  
 @Override  
 public void run() {  
 try {  
 if(lockNum % 2 == 1){  
 *lock1*.lockInterruptibly();  
 System.*out*.println(Thread.*currentThread*().getName() + "获得锁1，还缺锁2");  
 Thread.*sleep*(new Random().nextInt(500));  
 *lock2*.lockInterruptibly();  
 System.*out*.println(Thread.*currentThread*().getName() + "获得锁2，集齐2个锁");  
 }else {  
 *lock2*.lockInterruptibly();  
 System.*out*.println(Thread.*currentThread*().getName() + "获得锁2，还缺锁1");  
 Thread.*sleep*(new Random().nextInt(500));  
 *lock1*.lockInterruptibly();  
 System.*out*.println(Thread.*currentThread*().getName() + "获得锁1，集齐2个锁");  
 }  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 } finally {  
 if(*lock1*.isHeldByCurrentThread()) *lock1*.unlock();// 如果锁被当前线程持有，才释放，不是你拿的你也释放不了  
 if(*lock2*.isHeldByCurrentThread()) *lock2*.unlock();  
 System.*out*.println(Thread.*currentThread*().getName() + "线程退出");  
 }  
 }  
 }  
  
 public static void main(String[] args) throws InterruptedException {  
 IntLock intLock1 = new IntLock(11);  
 IntLock intLock2 = new IntLock(22);  
 Thread t1 = new Thread(intLock1);  
 Thread t2 = new Thread(intLock2);  
 t1.start();  
 t2.start();  
  
 Thread.*sleep*(3000);  
 // 唤醒两者之一即可解决死锁问题  
// if(t1.isAlive()) t1.interrupt();  
 if(t2.isAlive()) t2.interrupt();  
 }  
}

**5.2.3 tryLock()方法**

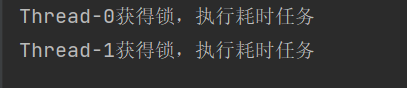
tryLock(long time, TimeUnit unit)方法的作用是在给定等待时长内锁没有被另外的线程持有，并且当前线程也没有被中断，则获得该锁

该方法可以实现锁对象的限时等待

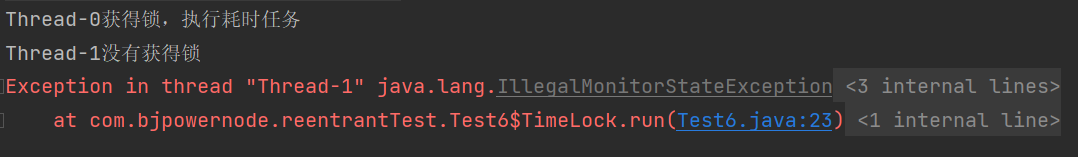
使用示例：

// tryLock示例  
public class Test6 {  
 static class TimeLock implements Runnable{  
 private static ReentrantLock *Lock* = new ReentrantLock();  
  
 @Override  
 public void run() {  
 try {  
 if(*Lock*.tryLock(3, TimeUnit.*SECONDS*)){  
 System.*out*.println(Thread.*currentThread*().getName() + "获得锁，执行耗时任务");  
 Thread.*sleep*(2000);// 模拟耗时任务  
 }else { // 没有获得锁  
 System.*out*.println(Thread.*currentThread*().getName() + "没有获得锁");  
 }  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 } finally {  
 *Lock*.unlock();  
 }  
 }  
 }  
  
 public static void main(String[] args) {  
 TimeLock timeLock = new TimeLock();  
 Thread t1 = new Thread(timeLock);  
 Thread t2 = new Thread(timeLock);  
 t1.start();  
 t2.start();  
 }  
}

如下，我sleep 2秒，在tryLock 3秒的接受时间内，所以我可以直接执行



当我把sleep的时间改为4秒，Thread-0获得锁，使用4秒执行完任务，所以占有了锁4秒，导致Thread-1的tryLock超时，于是Thread-1不再尝试获得锁，直接中断，打印异常



tryLock(long time, TimeUnit unit)有一个无参的重载方法，tryLock()，仅在调用时锁定未被其他线程持有的锁，如果调用方法时，锁对象被其他线程持有，不等待则放弃

拿到锁就返回true，没拿到返回false

**5.2.4 使用tryLock()方法可以避免死锁**

就是每次想要获得一个锁，先tryLock一下，演示略

**5.2.5 newCondition()方法**

关键字synchronized与wait()/notify()方法一起使用可以实现等待/通知模式

而Lock()锁的newCondition()方法返回Condition对象，Condition类也可以实现等待/通知模式

使用notify()通知时，java虚拟机会随机唤醒某个等待的线程

使用Condition类可以实现有选择的通知

Condition类中两个常用的方法：

await()方法，使当前线程等待，同时释放锁

当其他线程调用signal()时，线程会重新获得锁并继续执行——signal()用于唤醒一个等待的线程

注意：在调用Condition的await()和signal()方法前，也需要线程持有相关的Lock锁，调用await()后线程会释放这个锁，在signal()调用后会从当前condition对象的等待队列中唤醒一个线程，唤醒的线程尝试获得锁，一旦获得锁成功就继续执行

演示：

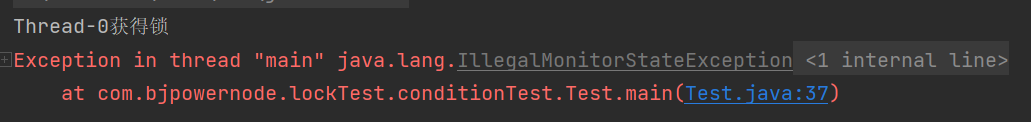
public class Test {  
 // 定义锁  
 static Lock *lock* = new ReentrantLock();  
 // 获得Condition对象  
 static Condition *condition* = *lock*.newCondition();  
  
 // 定义子线程类  
 static class SubThread extends Thread{  
 @Override  
 public void run() {  
 try {  
 *lock*.lock();  
 System.*out*.println(Thread.*currentThread*().getName() + "获得锁");  
 *condition*.await();  
 System.*out*.println("线程等待");  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 } finally {  
 *lock*.unlock();  
 System.*out*.println(Thread.*currentThread*().getName() + "释放锁");  
 }  
 }  
 }  
  
 public static void main(String[] args) {  
 SubThread subThread = new SubThread();  
 subThread.start();  
 }  
}

这个程序一运行，子线程就进入了等待状态

我们尝试唤醒线程：3秒后唤醒

public static void main(String[] args) throws InterruptedException {  
 SubThread t = new SubThread();  
 t.start();  
 Thread.*sleep*(3000);  
 *condition*.signal();  
}

但是，这么写是不行的，会爆异常：



之前我们也说了，使用signal()方法唤醒需要先获得锁对象

public static void main(String[] args) throws InterruptedException {  
 SubThread t = new SubThread();  
 t.start();  
 Thread.*sleep*(3000);  
 try {  
 *lock*.lock();  
 *condition*.signal();  
 } catch (Exception e) {  
 e.printStackTrace();  
 } finally {  
 *lock*.unlock();  
 }  
}

这样就行了

**5.2.6 多个Condition实现通知部分线程**

只需要创建多个Condition对象，然后用不同的condition控制不同的线程，就可以实现指定线程的唤醒了

**5.2.7 使用Condition实现两个线程交替打印**

非常简单，就是定义一个flag

线程1在flag为true的时候等待，为false的时候打印，线程2在flag为false的时候等待，为true的时候打印

详见IDEA代码，并尝试使用Synchronized实现交替打印

// Condition实现交替打印  
public class Test3 {  
 static class Service{  
 private Lock lock = new ReentrantLock();  
 Condition condition = lock.newCondition();  
 private boolean flag = true;  
  
 // 打印++++的方法  
 public void printOne(){  
 try {  
 lock.lock();  
 while (flag){  
 System.*out*.println(Thread.*currentThread*().getName() + " wait");  
 condition.await();  
 }  
 System.*out*.println("+++++++");  
 flag = true;  
 condition.signal();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 } finally {  
 lock.unlock();  
 }  
 }  
  
 // 打印-----的方法  
 public void printTwo(){  
 try {  
 lock.lock();  
 while (!flag){  
 System.*out*.println(Thread.*currentThread*().getName() + " wait");  
 condition.await();  
 }  
 System.*out*.println("--------");  
 flag = false;  
 condition.signal();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 } finally {  
 lock.unlock();  
 }  
 }  
 }  
  
 public static void main(String[] args) {  
 Service s = new Service();  
 Thread t1 = new Thread(new Runnable() {  
 @Override  
 public void run() {  
 while (true) s.printOne();  
 }  
 });  
  
 Thread t2 = new Thread(new Runnable() {  
 @Override  
 public void run() {  
 while (true) s.printTwo();  
 }  
 });  
  
 t1.start();  
 t2.start();  
 }  
}

多对多交替打印，如果使用以上的方法，有可能会出现假死现象

解决方法1：使用signalAll()

解决方法2：使用两个Condition对象分别负责处理打印+++++和打印----的多个线程