# Java多线程编程

## Java多线程技能

### com.thread.capther1.multiThreadSkill.demo1

继承Thread类实现线程

**package com.thread.capther1.multiThreadSkill.demo1**;  
  
**public class MyThread extends Thread** {  
 @Override  
 **public void** run() {  
 **System**.***out***.println("MyThread.run");  
 }  
}

**package com.thread.capther1.multiThreadSkill.demo1**;  
  
**public class Run** {  
 **public static void** main(**String**[] args) {  
 **MyThread** mt = **new** MyThread();  
 mt.start();  
 **System**.***out***.println("线程结束");  
 }  
}  
  
*/\*  
线程结束  
MyThread.run  
  
这里可以看出,  
在使用多线程技术时,代码的运行结果与代码执行顺序或调用顺序是无关的  
\*/*

### com.thread.capther1.multiThreadSkill.demo2

多个线程访问,并无特定的执行顺序

**package com.thread.capther1.multiThreadSkill.demo2**;  
  
**public class MyThread extends Thread** {  
 @Override  
 **public void** run() {  
 **try** {  
 **for** (**int** i = **0**; i < **10**; i++) {  
 **int** time = (**int**) (**Math**.random() \* **1000**);  
 **Thread**.sleep(time);  
 **System**.***out***.println("run=" + **Thread**.currentThread().getName());  
  
 }  
 }**catch** (**InterruptedException** e){  
  
 }  
 }  
}

**package com.thread.capther1.multiThreadSkill.demo2**;  
  
**import com.thread.capther1.multiThreadSkill.demo1.MyThread**;  
  
**public class Main** {  
 **public static void** main(**String**[] args) {  
 **try** {  
 **com.thread.capther1.multiThreadSkill.demo1.MyThread** mt = **new** MyThread();  
 mt.setName("myThread");  
 mt.start();  
 **for** (**int** i = **0**; i < **10**; i++) {  
 **int** time = (**int**) (**Math**.random() \* **1000**);  
 **Thread**.sleep(time);  
 **System**.***out***.println("Main=" + **Thread**.currentThread().getName());  
 }  
 } **catch** (**InterruptedException** e) {  
  
 }  
 }  
}  
  
  
*/\*  
\*  
Main=main  
Main=main  
run=myThread  
Main=main  
run=myThread  
Main=main  
Main=main  
run=myThread  
run=myThread  
run=myThread  
Main=main  
run=myThread  
Main=main  
run=myThread  
Main=main  
Main=main  
Main=main  
run=myThread  
run=myThread  
run=myThread  
  
随机数的形式来使线程得到挂起的效果,从而表现出CPU执行哪个线程具有不确定性  
\* \*/*

### com.thread.capther1.multiThreadSkill.demo3

实现runnable方法方式创建线程

**package com.thread.capther1.multiThreadSkill.demo3**;  
  
**public class MyRunnable implements Runnable** {  
 @Override  
 **public void** run() {  
 **System**.***out***.println("运行中...");  
 }  
}

**package com.thread.capther1.multiThreadSkill.demo3**;  
  
**public class Main** {  
 **public static void** main(**String**[] args) {  
  
 **MyRunnable** mr = **new** MyRunnable();  
 **Thread** t = **new** Thread(mr);  
 t.start();  
 **System**.***out***.println("运行结束...");  
  
 }  
}  
  
  
*/\*  
 public Thread(Runnable target) {}  
 单独的runnable方法是无法执行的,需要将runnable目标放在Thread对象中执行...  
  
  
 target也可以传入一个Thread类对象  
\*/*

### com.thread.capther1.multiThreadSkill.demo4

多个线程不共享同一个实例变量

**package com.thread.capther1.multiThreadSkill.demo4**;  
  
**public class MyThread extends Thread** {  
 **private int** *count* = **5**;  
  
 **public MyThread**(**String** name) {  
 **this**.setName(name);  
 }  
  
 @Override  
 **public void** run() {  
 **super**.run();  
 **while** (*count* > **0**) {  
 *count*--;  
 **System**.***out***.println("由" + **Thread**.currentThread().getName() + " 计算,count:" + *count*);  
 }  
 }  
}

**package com.thread.capther1.multiThreadSkill.demo4**;  
  
**public class Run** {  
 **public static void** main(**String**[] args) {  
 **MyThread** mt1 = **new** MyThread("A");  
 **MyThread** mt2 = **new** MyThread("B");  
 **MyThread** mt3 = **new** MyThread("C");  
  
 mt1.start();  
 mt2.start();  
 mt3.start();  
 }  
}  
  
*/\*  
  
 由C 计算,count:4  
 由B 计算,count:4  
 由A 计算,count:4  
 由A 计算,count:3  
 由A 计算,count:2  
 由B 计算,count:3  
 由B 计算,count:2  
 由B 计算,count:1  
 由B 计算,count:0  
 由C 计算,count:3  
 由A 计算,count:1  
 由A 计算,count:0  
 由C 计算,count:2  
 由C 计算,count:1  
 由C 计算,count:0  
  
每个线程独自计算各自的count变量,自己减少自己的count值  
  
这里的变量不共享  
不存在多线程访问同一个实例变量的情况  
  
\*/*

### com.thread.capther1.multiThreadSkill.demo5

多个线程共享同一个实例变量的情况,第一次出现了非线程安全问题

**package com.thread.capther1.multiThreadSkill.demo5**;  
  
**public class MyThread extends Thread** {  
 **private int** *count* = **5**;  
  
  
 @Override  
 **public void** run() {  
 **super**.run();  
 *count*--;  
  
 **System**.***out***.println("由" + **Thread**.currentThread().getName() + " 计算,count:" + *count*);  
 }  
}

**package com.thread.capther1.multiThreadSkill.demo5**;  
  
**public class Run** {  
 **public static void** main(**String**[] args) {  
  
 **MyThread** m = **new** MyThread();  
 **Thread** a = **new** Thread(m, "A");  
 **Thread** b = **new** Thread(m, "B");  
 **Thread** c = **new** Thread(m, "C");  
 **Thread** d = **new** Thread(m, "D");  
 **Thread** e = **new** Thread(m, "E");  
  
 a.start();  
 b.start();  
 c.start();  
 d.start();  
 e.start();  
 }  
}  
  
*/\*  
由B 计算,count:3  
由A 计算,count:0  
由D 计算,count:1  
由E 计算,count:3  
由C 计算,count:2  
  
从结果中可以看出有两个线程打印了同一个值,  
产生了"非线程安全问题"  
  
  
 public Thread(Runnable target, String name) {  
 init(null, target, name, 0);  
 }  
Thread类有这个一个构造方法  
可以同享一个runnable对象  
\*/*

### com.thread.capther1.multiThreadSkill.demo6

用synchronized解决线程安全问题

非线程安全?  
  
主要是指多个线程对同一个对象中的同一个实例变量进行操作时会出现值被更改,  
值不同步的情况,进而影响程序的执行流程

**package com.thread.capther1.multiThreadSkill.demo6**;  
  
**public class MyThread extends Thread** {  
 **private int** *count* = **5**;  
  
  
 @Override  
 **synchronized public void** run() {  
 **super**.run();  
 *count*--;  
  
 **System**.***out***.println("由" + **Thread**.currentThread().getName() + " 计算,count:" + *count*);  
 }  
}

**package com.thread.capther1.multiThreadSkill.demo6**;  
  
**public class Run** {  
 **public static void** main(**String**[] args) {  
  
 **MyThread** m = **new** MyThread();  
 **Thread** a = **new** Thread(m, "A");  
 **Thread** b = **new** Thread(m, "B");  
 **Thread** c = **new** Thread(m, "C");  
 **Thread** d = **new** Thread(m, "D");  
 **Thread** e = **new** Thread(m, "E");  
  
 a.start();  
 b.start();  
 c.start();  
 d.start();  
 e.start();  
 }  
}  
  
*/\*  
由A 计算,count:4  
由E 计算,count:3  
由D 计算,count:2  
由B 计算,count:1  
由C 计算,count:0  
  
使用synchronized解决了线程占用的问题,  
达到了count按顺序减少的效果.  
  
syn可以在任意对象及方法上加锁,而加锁的这段代码称为"互斥区"或"临界区"  
\*/*

### com.thread.capther1.multiThreadSkill.demo7

用synchronized解决线程安全问题

**package com.thread.capther1.multiThreadSkill.demo7**;  
  
**public class MyThread extends Thread** {  
 **private int** *count* = **5**;  
  
  
 @Override  
 **synchronized public void** run() {  
 **super**.run();  
 *count*--;  
  
 **System**.***out***.println("由" + **Thread**.currentThread().getName() + " 计算,count:" + *count*);  
 }  
}

**package com.thread.capther1.multiThreadSkill.demo7**;  
  
**public class Run** {  
 **public static void** main(**String**[] args) {  
  
 **MyThread** m = **new** MyThread();  
 **Thread** a = **new** Thread(m, "A");  
 **Thread** b = **new** Thread(m, "B");  
 **Thread** c = **new** Thread(m, "C");  
 **Thread** d = **new** Thread(m, "D");  
 **Thread** e = **new** Thread(m, "E");  
  
 a.start();  
 b.start();  
 c.start();  
 d.start();  
 e.start();  
 }  
}  
  
*/\*  
由A 计算,count:4  
由E 计算,count:3  
由D 计算,count:2  
由B 计算,count:1  
由C 计算,count:0  
  
使用synchronized解决了线程占用的问题,  
达到了count按顺序减少的效果.  
  
syn可以在任意对象及方法上加锁,而加锁的这段代码称为"互斥区"或"临界区"  
\*/*

## 多线程及变量的并发访问

### com.thread.capther2.concurrentAccess.demo1

多个对象多个锁

**package com.thread.capther2.concurrentAccess.demo1**;  
  
**public class HasSelfPrivateNum** {  
 **private int** *num* = **0**;  
  
 **synchronized public void** addI(**String** userName) {  
 **try** {  
 **if** (userName.equals("a")) {  
 *num* = **100**;  
 **System**.***out***.println("a set over!");  
 **Thread**.sleep(**1000**);  
 } **else** {  
 *num* = **200**;  
 **System**.***out***.println("b set over!");  
 }  
 **System**.***out***.println(userName + " num =" + *num*);  
  
 } **catch** (**InterruptedException** e) {  
  
 }  
 }  
  
}

**package com.thread.capther2.concurrentAccess.demo1**;  
  
**public class ThreadA extends Thread** {  
 **private HasSelfPrivateNum** *numRef*;  
  
 **public ThreadA**(**HasSelfPrivateNum** numRef) {  
 **this**.*numRef* = numRef;  
 }  
  
 @Override  
 **public void** run() {  
 **super**.run();  
 *numRef*.addI("a");  
 }  
}

**package com.thread.capther2.concurrentAccess.demo1**;  
  
**public class ThreadB extends Thread** {  
 **private HasSelfPrivateNum** *numRef*;  
  
 **public ThreadB**(**HasSelfPrivateNum** numRef) {  
 **this**.*numRef* = numRef;  
 }  
  
 @Override  
 **public void** run() {  
 **super**.run();  
 *numRef*.addI("b");  
 }  
}

**package com.thread.capther2.concurrentAccess.demo1**;  
  
**public class Run** {  
 **public static void** main(**String**[] args) {  
  
 **HasSelfPrivateNum** hspn1 = **new** HasSelfPrivateNum();  
 **HasSelfPrivateNum** hspn2 = **new** HasSelfPrivateNum();  
  
 **ThreadA** ta = **new** ThreadA(hspn1);  
 **ThreadB** tb = **new** ThreadB(hspn2);  
  
 ta.start();  
 tb.start();  
 }  
}  
  
*/\*  
  
a set over!  
b set over!  
b num =200  
a num =100  
  
创建了2个对象,也就是JVM创建了2个锁  
  
\*/*

### com.thread.capther2.concurrentAccess.demo2

什么叫脏读?  
  
虽然在赋值时进行了同步,但在取值时有可能出现一些意向不到的意外,这种情况就是脏读(dirtyRead)  
发生脏读的情况是在读取实例变量时,此值已经被其他线程更改过了.

**package com.thread.capther2.concurrentAccess.demo2**;  
  
**public class PublicVar** {  
 **public String** *username* = "A";  
 **public String** *password* = "AA";  
  
 **synchronized public void** setValue(**String** username, **String** password) {  
 **try** {  
 **this**.*username* = username;  
 **Thread**.sleep(**2000**);  
 **this**.*password* = password;  
 **System**.***out***.println("SetValue thread name:" + **Thread**.currentThread().getName() + " username:" + username + " password:" + password);  
  
 } **catch** (**InterruptedException** e) {  
  
 }  
 }  
  
 **public void** getValue() {  
 **System**.***out***.println("getValue thread name:" + **Thread**.currentThread().getName() + " username:" + *username* + " password:" + *password*);  
  
 }  
}

**package com.thread.capther2.concurrentAccess.demo2**;  
  
**public class ThreadA extends Thread** {  
 **private PublicVar** *pv*;  
  
 **public ThreadA**(**PublicVar** pv) {  
 **this**.*pv* = pv;  
 }  
  
 @Override  
 **public void** run() {  
 **super**.run();  
 *pv*.setValue("B", "BB");  
 }  
}

**package com.thread.capther2.concurrentAccess.demo2**;  
  
**public class Test** {  
  
 **public static void** main(**String**[] args) {  
 **try** {  
 **PublicVar** pv = **new** PublicVar();  
 **ThreadA** t = **new** ThreadA(pv);  
  
 t.start();  
 **Thread**.sleep(**200**);  
 pv.getValue();  
  
 } **catch** (**InterruptedException** e) {  
  
 }  
 }  
}  
  
*/\*  
  
 getValue thread name:main username:B password:AA  
 SetValue thread name:Thread-0 username:B password:BB  
  
  
结果读取的数据并不一致,  
出现脏读是因为public void getValue()方法并不是同步的,所以可以在任意时候进行调用,  
解决办法当然是加上同步synchronized  
\*/*

### com.thread.capther2.concurrentAccess.demo3

什么叫锁重入?  
关键字synchronized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,再次请求此对象的锁时,是可以再次得到该对象的锁的.  
  
这也证明在一个synchronized方法/块的内部调用本类的其他synchonized方法/块时,是永远可以得到锁的.  
  
  
\*\*\*  
当一个线程执行代码出现异常时,其所持有的锁会自动释放.  
  
\*\*\*  
同步不可以继承

**package com.thread.capther2.concurrentAccess.demo3**;  
  
**public class Service** {  
  
 **synchronized public void** service1() {  
 **System**.***out***.println("service1");  
 service2();  
 }  
  
 **synchronized public void** service2() {  
 **System**.***out***.println("service2");  
 service3();  
 }  
  
 **synchronized public void** service3() {  
 **System**.***out***.println("service3");  
 }  
  
}

**package com.thread.capther2.concurrentAccess.demo3**;  
  
**public class MyThread extends Thread** {  
 @Override  
 **public void** run() {  
 **super**.run();  
 **Service** s = **new** Service();  
 s.service1();  
 }  
  
  
  
 **public static void** main(**String**[] args) {  
  
 **MyThread** m = **new** MyThread();  
 m.start();  
 }  
}  
  
*/\*  
service1  
service2  
service3  
  
可重入锁的概念是:自己可以再次获取自己的内部锁  
\*/*

### com.thread.capther2.concurrentAccess.demo4

关键字volatile的主要作用是是变量在多个线程间可见

在使用同步synchronized(this)代码块时需要注意的是,当一个线程访问object的一个synchronized(this)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使用的"对象监视器"是一个.  
  
  
  
和synchronized一样,synchronized(this)代码块也是锁定当前对象的.  
  
  
  
  
  
Java还支持"任意对象"作为"对象监视器"来实现同步功能,这个"任意对象"大多数是实例变量及方法的参数,  
 使用格式为  
 synchronized(非this对象)  
  
  
synchronized(非this对象)格式的作用只有一种:synchronized(非this对象x)同步代码块:  
1)在多个线程持有"对象监视器"为同一个对象的前提下,同一时间只有一个线程可以执行synchronized(非this对象x)同步代码块中的代码;  
2)当持有"对象监视器"为同一个对象的前提下,同一时间只有一个线程可以执行synchronized(非this对象x)同步代码块中的代码;  
  
  
  
锁非this队友具有一定的优点:  
 如果在一个类中有很多歌synchronized方法,这时虽然能实现同步,但会阻塞,所以影响效率;  
 但如果使用同步代码块非this对象,则与同步方法是异步的,不与其他锁this同步方法争抢this锁,则可以大大提高效率;  
  
  
关键字synchronized还可以应用在static静态方法上,如果这样写,那是对当前的\*.java文件对应的Class类进行持锁.  
  
syn关键字加到static静态方法上市给Class类上锁,而syn关键字加到非static静态方法上是给对象上锁.  
Class锁可以对类的所有对象实例起作用!  
  
  
在将任何数据类型作为同步锁时,需要注意的是,是否有多个线程同时持有锁对象,  
如果同时持有相同的锁对象,则这些线程之间就是同步的,  
如果分别获得锁对象,这些线程之间就是异步的!

**package com.thread.capther2.concurrentAccess.demo4**;  
  
**public class ObjectService** {  
 **public void** serviceMethod() {  
  
 **try** {  
 **synchronized** (**this**) {  
 **System**.***out***.println("begin time = " + **System**.currentTimeMillis());  
 **Thread**.sleep(**2000**);  
 **System**.***out***.println("end time = " + **System**.currentTimeMillis());  
 }  
 } **catch** (**InterruptedException** e) {  
  
 }  
 }  
}

**package com.thread.capther2.concurrentAccess.demo4**;  
  
**public class ThreadA extends Thread** {  
  
 **private ObjectService** *service*;  
  
 **public ThreadA**(**ObjectService** service) {  
 **this**.*service* = service;  
 }  
  
 @Override  
 **public void** run() {  
 **super**.run();  
 *service*.serviceMethod();  
 }  
}

**package com.thread.capther2.concurrentAccess.demo4**;  
  
**public class ThreadB extends Thread** {  
 **private ObjectService** *service*;  
  
 **public ThreadB**(**ObjectService** service) {  
 **this**.*service* = service;  
 }  
  
 @Override  
 **public void** run() {  
 **super**.run();  
 *service*.serviceMethod();  
 }  
}

**package com.thread.capther2.concurrentAccess.demo4**;  
  
**public class Run** {  
 **public static void** main(**String**[] args) {  
  
 **ObjectService** os = **new** ObjectService();  
 **ThreadA** ta = **new** ThreadA(os);  
 ta.setName("a");  
 ta.start();  
  
 **ThreadB** tb = **new** ThreadB(os);  
 tb.setName("b");  
 tb.start();  
 }  
}  
  
  
//使用使用了同步代码块,但是效率并没有提高

### com.thread.capther2.concurrentAccess.demo5

syn和volatile的比较  
  
使用volatile关键字增加了实例变量在多个线程之间的可见性.但volatile关键字最致命的缺点是不支持原子性.  
  
下面将关键字syn和volatile进行一下比较:  
1)关键字volatile是线程同步的轻量级实现,所以volatile性能肯定比synchronized要好,并且volatile只能修饰于变量,而syn可以修饰方法,以及代码块.  
随着JDK新版本的发布,syn关键字在执行效率上得到很大的提升,在开发中使用synchronized关键字的比率还是比较大的;  
  
2)多线程访问volatile不会发生阻塞,而syn会出现阻塞;  
  
3)volatile能保证数据的可见性,但不能保证原子性;而syn可以保证原子性,也可以间接保证可见性,因为他会将私有内存和公共内存中的数据做同步,  
  
4)再次重申一下,关键字volatile解决的是变量在多个线程之间的可见性,而syn关键字解决的是多个线程之间访问资源的同步性;

什么叫线程安全?  
  
线程安全就是以获得实例变量的值是经过同步处理的,不会出现脏读的现象.  
  
  
非线程安全问题存在于"实例变量"中,如果是方法内部的私有变量,则不存在非线程安全问题,所得结果也就是 "线程安全的"了  
并且永远都是线程安全的!  
这是方法内部的变量是私有的特性造成的!

## 线程间通信

### com.thread.capther3.threadsCommunication.demo1

**package com.thread.capther3.threadsCommunication.demo1**;  
  
**public class Test** {  
  
 **public static void** main(**String**[] args) {  
  
 **try** {  
 **String** lock = **new** String();  
 **System**.***out***.println("syn上面");  
 **synchronized** (lock) {  
 **System**.***out***.println("syn里的第一行");  
 lock.wait();  
 **System**.***out***.println("wait下面的代码");  
 }  
 **System**.***out***.println("syn外面的代码");  
 } **catch** (**InterruptedException** e) {  
  
 }  
 }  
}  
  
*/\*  
  
 syn上面  
 syn里的第一行  
  
  
 程序走到这里就不走了,但是程序并没有退出!  
  
\*/*

### com.thread.capther3.threadsCommunication.demo2

**package com.thread.capther3.threadsCommunication.demo2**;  
  
**public class MyThread1 extends Thread** {  
  
 **private Object** *lock*;  
  
 **public MyThread1**(**Object** lock) {  
 **this**.*lock* = lock;  
 }  
  
 @Override  
 **public void** run() {  
 **super**.run();  
 **try** {  
 **synchronized** (*lock*) {  
 **System**.***out***.println("wait开始时间:" + **System**.currentTimeMillis());  
 *lock*.wait();  
 **System**.***out***.println("wait结束时间:" + **System**.currentTimeMillis());  
 }  
  
 } **catch** (**InterruptedException** e) {  
  
 }  
 }  
}

**package com.thread.capther3.threadsCommunication.demo2**;  
  
**public class MyThread2 extends Thread** {  
  
 **private Object** *lock*;  
  
 **public MyThread2**(**Object** lock) {  
 **this**.*lock* = lock;  
 }  
  
 @Override  
 **public void** run() {  
  
 **synchronized** (*lock*) {  
 **System**.***out***.println("notify开始时间:" + **System**.currentTimeMillis());  
 *lock*.notify();  
 **System**.***out***.println("notify结束时间:" + **System**.currentTimeMillis());  
 }  
  
 }  
}

**package com.thread.capther3.threadsCommunication.demo2**;  
  
**public class Test** {  
 **public static void** main(**String**[] args) {  
 **try** {  
 **Object** lock = **new** Object();  
 **MyThread1** m1 = **new** MyThread1(lock);  
  
 m1.start();  
 **Thread**.sleep(**2000**);  
 **MyThread2** m2 = **new** MyThread2(lock);  
 m2.start();  
  
 } **catch** (**InterruptedException** e) {  
  
 }  
 }  
}  
  
  
*/\*  
  
 wait开始时间:1536310955451  
 notify开始时间:1536310957452  
 notify结束时间:1536310957452  
 wait结束时间:1536310957453  
  
2s后线程被notify通知唤醒;  
\*/*

### com.thread.capther3.threadsCommunication.demo3

**package com.thread.capther3.threadsCommunication.demo3**;  
  
**import java.util.ArrayList**;  
**import java.util.List**;  
  
**public class MyList** {  
 **private static List** *list* = **new** ArrayList();  
  
 **public static void** add() {  
 *list*.add("anyString");  
 }  
  
 **public static int** size() {  
 **return** *list*.size();  
 }  
}

**package com.thread.capther3.threadsCommunication.demo3**;  
  
**public class ThreadA extends Thread** {  
  
 **private Object** *lock*;  
  
 **public ThreadA**(**Object** lock) {  
 **this**.*lock* = lock;  
 }  
  
  
 @Override  
 **public void** run() {  
 **try** {  
 **synchronized** (*lock*) {  
 **if** (**MyList**.size() != **5**) {  
 **System**.***out***.println("wait开始时间:" + **System**.currentTimeMillis());  
 *lock*.wait();  
 **System**.***out***.println("wait结束时间:" + **System**.currentTimeMillis());  
 }  
 }  
  
 } **catch** (**InterruptedException** e) {  
  
 }  
  
 }  
}

**package com.thread.capther3.threadsCommunication.demo3**;  
  
**public class ThreadB extends Thread** {  
 **private Object** *lock*;  
  
 **public ThreadB**(**Object** lock) {  
 **this**.*lock* = lock;  
 }  
  
 @Override  
 **public void** run() {  
 **synchronized** (*lock*) {  
 **try** {  
 **for** (**int** i = **0**; i < **10**; i++) {  
 **MyList**.add();  
 **if** (**MyList**.size() == **5**) {  
 *lock*.notify();  
 **System**.***out***.println("已发出通知!");  
 }  
 **System**.***out***.println("添加了" + (i + **1**) + "个元素");  
 **Thread**.sleep(**1000**);  
 }  
  
 } **catch** (**InterruptedException** e) {  
  
 }  
 }  
  
 }  
}

**package com.thread.capther3.threadsCommunication.demo3**;  
  
**public class Test** {  
 **public static void** main(**String**[] args) {  
  
 **try** {  
 **Object** lock = **new** Object();  
  
 **ThreadA** ta = **new** ThreadA(lock);  
 ta.start();  
 **Thread**.sleep(**50**);  
  
 **ThreadB** tb = **new** ThreadB(lock);  
 tb.start();  
  
 } **catch** (**InterruptedException** e) {  
 }  
 }  
}  
  
  
*/\*  
  
wait开始时间:1536312457291  
 添加了1个元素  
 添加了2个元素  
 添加了3个元素  
 添加了4个元素  
 已发出通知!  
 添加了5个元素  
 添加了6个元素  
 添加了7个元素  
 添加了8个元素  
 添加了9个元素  
 添加了10个元素  
wait结束时间:1536312467345  
  
  
wait end在最后输出,说明notify()方法执行并不立即释放锁!!!  
\*/*

### com.thread.capther3.threadsCommunication.demo4

调用方法notify()一次只随机通知一个线程进行唤醒!  
  
为了唤醒全部的线程,可使用notifyAll()方法;  
  
带一个参数的wait(long)方法的功能:  
 等待某一时间内是否有线程对锁进行唤醒,如果超过这个时间则自动唤醒!

**package com.thread.capther3.threadsCommunication.demo4**;  
  
**public class NotifyMethodThread extends Thread** {  
 **private Object** *lock*;  
  
 **public NotifyMethodThread**(**Object** lock) {  
 **this**.*lock* = lock;  
 }  
  
 @Override  
 **public void** run() {  
 **Service** service = **new** Service();  
 service.synNotifyMethod(*lock*);  
 }  
}

**package com.thread.capther3.threadsCommunication.demo4**;  
  
**public class synNotifyMethodThread extends Thread** {  
 **private Object** *lock*;  
  
 **public synNotifyMethodThread**(**Object** lock) {  
 **this**.*lock* = lock;  
 }  
  
 @Override  
 **public void** run() {  
 **Service** service = **new** Service();  
 service.synNotifyMethod(*lock*);  
  
 }  
}

**package com.thread.capther3.threadsCommunication.demo4**;  
  
**public class Service** {  
 **public void** testMethod(**Object** obj) {  
 **try** {  
 **synchronized** (obj) {  
 **System**.***out***.println("wait() 开始的线程名称:"+**Thread**.currentThread().getName());  
 obj.wait();  
 **System**.***out***.println("wait() 结束的线程名称:"+**Thread**.currentThread().getName());  
 }  
 } **catch** (**InterruptedException** e) {  
  
 }  
 }  
  
 **public void** synNotifyMethod(**Object** lock){  
 **try** {  
 **synchronized** (lock){  
 **System**.***out***.println("notify() 开始的线程名称:"+**Thread**.currentThread().getName()+" "+**System**.currentTimeMillis());  
 lock.notify();  
 **Thread**.sleep(**4000**);  
 **System**.***out***.println("notify() 结束的线程名称:"+**Thread**.currentThread().getName()+" "+**System**.currentTimeMillis());  
 }  
 }**catch** (**InterruptedException** e){  
  
 }  
  
 }  
}

**package com.thread.capther3.threadsCommunication.demo4**;  
  
**public class TestMethodThead extends Thread** {  
 **private Object** *lock*;  
  
 **public TestMethodThead**(**Object** lock) {  
 **this**.*lock* = lock;  
 }  
  
 @Override  
 **public void** run() {  
 **Service** service = **new** Service();  
 service.testMethod(*lock*);  
 }  
}

**package com.thread.capther3.threadsCommunication.demo4**;  
  
**public class Test** {  
 **public static void** main(**String**[] args) {  
  
 **Object** lock = **new** Object();  
 **TestMethodThead** tt = **new** TestMethodThead(lock);  
 tt.start();  
  
 **NotifyMethodThread** nt = **new** NotifyMethodThread(lock);  
 nt.start();  
  
 **synNotifyMethodThread** st = **new** synNotifyMethodThread(lock);  
 st.start();  
  
 }  
}  
  
*/\*  
 wait() 开始的线程名称:Thread-0  
 notify() 开始的线程名称:Thread-2 1536543715315  
 notify() 结束的线程名称:Thread-2 1536543719315  
 notify() 开始的线程名称:Thread-1 1536543719315  
 notify() 结束的线程名称:Thread-1 1536543723315  
 wait() 结束的线程名称:Thread-0  
  
  
  
\*/*

### com.thread.capther3.threadsCommunication.demo5

join方法的产生原因:  
 在很多情况下,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,主线程往往将早于子线程结束之前结束;  
 这时,如果主线程想等待子线程执行完成之后再结束,比如子线程处理一个数据,主线程要取得这个数据中的值,就要用到join方法了;  
 方法join()的作用是等待线程对象的销毁;  
  
  
  
方法join的作用是使所属的线程对象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,等到线程x销毁后再继续执行线程z后面的代码;  
  
  
  
join与syn的区别:  
 join在内部使用wait()方法进行等待,  
 而syn关键字使用的是"对象监视器"原理做为同步!

**package com.thread.capther3.threadsCommunication.demo5**;  
  
**public class Test** {  
 **public static void** main(**String**[] args) {  
  
 **MyThread** mt = **new** MyThread();  
 mt.start();  
 **System**.***out***.println("我想得到上面线程的value之后再执行");  
  
 }  
}

**package com.thread.capther3.threadsCommunication.demo5**;  
  
**public class Test2** {  
 **public static void** main(**String**[] args) {  
 **try** {  
 **MyThread** mt = **new** MyThread();  
 mt.start();  
 mt.join();  
 **System**.***out***.println("我想得到上面线程的value之后再执行");  
 } **catch** (**InterruptedException** e) {  
  
 }  
 }  
}

**package com.thread.capther3.threadsCommunication.demo5**;  
  
**public class MyThread extends Thread** {  
  
 @Override  
 **public void** run() {  
 **try** {  
 **int** secondVaule = (**int**) (**Math**.random() \* **1000**);  
 **System**.***out***.println(secondVaule);  
 **Thread**.sleep(secondVaule);  
 } **catch** (**InterruptedException** e) {  
  
 }  
 }  
}

### com.thread.capther3.threadsCommunication.demo6

类ThreadLocal主要解决的就是每个线程都绑定自己的值,可以将ThreadLocal类比喻成全局存放数据的盒子,盒子可以存储每个线程的私有数据!

## ReentrantLock类的使用; ReentrantReadWriteLock类的使用;

### com.thread.capther4.Lock.demo1

ReentrantLock类可以实现线程之间的同步互斥,并且在扩展功能上更加强大,  
  
比如具有嗅探锁定,多路分支通知等功能,而且在使用上也比syn更加的灵活!

**package com.thread.capther4.Lock.demo1**;  
  
**public class MyThread extends Thread** {  
 **private MyService** *myService*;  
  
 **public MyThread**(**MyService** myService) {  
 **this**.*myService* = myService;  
 }  
  
 @Override  
 **public void** run() {  
 *myService*.testMethod();  
 }  
}

**package com.thread.capther4.Lock.demo1**;  
  
**import java.util.concurrent.locks.Lock**;  
**import java.util.concurrent.locks.ReentrantLock**;  
  
**public class MyService** {  
 **private Lock** *lock* = **new** ReentrantLock();  
  
  
 **public void** testMethod() {  
 *lock*.lock();  
 **for** (**int** i = **0**; i < **5**; i++) {  
 **System**.***out***.println("当前线程名字:" + **Thread**.currentThread().getName() + " " + (i + **1**));  
 }  
 *lock*.unlock();  
  
 }  
}

**package com.thread.capther4.Lock.demo1**;  
  
**public class Test** {  
 **public static void** main(**String**[] args) {  
 **MyService** ms = **new** MyService();  
 **MyThread** mt1 = **new** MyThread(ms);  
 **MyThread** mt2 = **new** MyThread(ms);  
 **MyThread** mt3 = **new** MyThread(ms);  
 **MyThread** mt4 = **new** MyThread(ms);  
 **MyThread** mt5 = **new** MyThread(ms);  
  
 mt1.start();  
 mt2.start();  
 mt3.start();  
 mt4.start();  
 mt5.start();  
  
 }  
}  
*/\*  
  
 当前线程名字:Thread-0 1  
 当前线程名字:Thread-0 2  
 当前线程名字:Thread-0 3  
 当前线程名字:Thread-0 4  
 当前线程名字:Thread-0 5  
 当前线程名字:Thread-3 1  
 当前线程名字:Thread-3 2  
 当前线程名字:Thread-3 3  
 当前线程名字:Thread-3 4  
 当前线程名字:Thread-3 5  
 当前线程名字:Thread-1 1  
 当前线程名字:Thread-1 2  
 当前线程名字:Thread-1 3  
 当前线程名字:Thread-1 4  
 当前线程名字:Thread-1 5  
 当前线程名字:Thread-4 1  
 当前线程名字:Thread-4 2  
 当前线程名字:Thread-4 3  
 当前线程名字:Thread-4 4  
 当前线程名字:Thread-4 5  
 当前线程名字:Thread-2 1  
 当前线程名字:Thread-2 2  
 当前线程名字:Thread-2 3  
 当前线程名字:Thread-2 4  
 当前线程名字:Thread-2 5  
  
\*/*

### com.thread.capther4.Lock.demo2

**package com.thread.capther4.Lock.demo2**;  
  
**public class ThreadA extends Thread** {  
 **private MyService** *myService*;  
  
 **public ThreadA**(**MyService** myService) {  
 **this**.*myService* = myService;  
 }  
  
 @Override  
 **public void** run() {  
  
 *myService*.methodA();  
  
 }  
}

**package com.thread.capther4.Lock.demo2**;  
  
**public class ThreadAA extends Thread** {  
 **private MyService** *myService*;  
  
 **public ThreadAA**(**MyService** myService) {  
 **this**.*myService* = myService;  
 }  
  
 @Override  
 **public void** run() {  
  
 *myService*.methodA();  
  
 }  
}

**package com.thread.capther4.Lock.demo2**;  
  
**public class ThreadB extends Thread** {  
 **private MyService** *myService*;  
  
 **public ThreadB**(**MyService** myService) {  
 **this**.*myService* = myService;  
 }  
  
 @Override  
 **public void** run() {  
  
 *myService*.methodB();  
 }  
}

**package com.thread.capther4.Lock.demo2**;  
  
**public class ThreadBB extends Thread** {  
 **private MyService** *myService*;  
  
 **public ThreadBB**(**MyService** myService) {  
 **this**.*myService* = myService;  
 }  
  
 @Override  
 **public void** run() {  
  
 *myService*.methodB();  
 }  
}

**package com.thread.capther4.Lock.demo2**;  
  
**import java.util.concurrent.locks.Lock**;  
**import java.util.concurrent.locks.ReentrantLock**;  
  
**public class MyService** {  
 **private Lock** *lock* = **new** ReentrantLock();  
  
 **public void** methodA() {  
 **try** {  
 *lock*.lock();  
 **System**.***out***.println("methodA 开始-线程名称:" + **Thread**.currentThread().getName() + " 开始时间:" + **System**.currentTimeMillis());  
 **Thread**.sleep(**5000**);  
 **System**.***out***.println("methodA 结束-线程名称:" + **Thread**.currentThread().getName() + " 结束时间:" + **System**.currentTimeMillis());  
  
  
 } **catch** (**InterruptedException** e) {  
  
 } **finally** {  
 *lock*.unlock();  
 }  
 }  
  
  
 **public void** methodB() {  
 **try** {  
 *lock*.lock();  
 **System**.***out***.println("methodB 开始-线程名称:" + **Thread**.currentThread().getName() + " 开始时间:" + **System**.currentTimeMillis());  
 **Thread**.sleep(**5000**);  
 **System**.***out***.println("methodB 结束-线程名称:" + **Thread**.currentThread().getName() + " 结束时间:" + **System**.currentTimeMillis());  
  
  
 } **catch** (**InterruptedException** e) {  
  
 } **finally** {  
 *lock*.unlock();  
 }  
 }  
  
}

**package com.thread.capther4.Lock.demo2**;  
  
**public class Test** {  
 **public static void** main(**String**[] args) **throws InterruptedException** {  
 **MyService** myService = **new** MyService();  
 **ThreadA** ta = **new** ThreadA(myService);  
 ta.setName("A");  
 ta.start();  
  
 **ThreadAA** taa = **new** ThreadAA(myService);  
 taa.setName("AA");  
 taa.start();  
  
 **ThreadB** tb = **new** ThreadB(myService);  
 tb.setName("B");  
 tb.start();  
  
 ThreadBB tbb = **new** ThreadBB(myService);  
 tbb.setName("BB");  
 tbb.start();  
 }  
}  
  
*/\*  
 methodA 开始-线程名称:A 开始时间:1536548341132  
 methodA 结束-线程名称:A 结束时间:1536548346132  
 methodA 开始-线程名称:AA 开始时间:1536548346132  
 methodA 结束-线程名称:AA 结束时间:1536548351132  
 methodB 开始-线程名称:B 开始时间:1536548351132  
 methodB 结束-线程名称:B 结束时间:1536548356133  
 methodB 开始-线程名称:BB 开始时间:1536548356133  
 methodB 结束-线程名称:BB 结束时间:1536548361134  
  
\*/*

### com.thread.capther4.Lock.demo3

syn与wait和notify与notifyAll方法相结合可以实现等待/通知模式,类ReentrantLock也可以实现同样的功能,但需要借助condition对象!  
  
  
Object.wait() === Condition.await()  
Object.wait(long) === Condition.await(long,unit);  
Object.notify() === Condition.signal();  
Object.notifyAll() === Condition.signalAll();

**package com.thread.capther4.Lock.demo3**;  
  
**public class MyThread extends Thread** {  
 **private MyService** *myService*;  
  
 **public MyThread**(**MyService** myService) {  
 **this**.*myService* = myService;  
 }  
  
 @Override  
 **public void** run() {  
 *myService*.await();  
 }  
}

**package com.thread.capther4.Lock.demo3**;  
  
**import java.util.concurrent.locks.Condition**;  
**import java.util.concurrent.locks.Lock**;  
**import java.util.concurrent.locks.ReentrantLock**;  
  
**public class MyService** {  
  
 **private Lock** *lock* = **new** ReentrantLock();  
 **private Condition** *condition* = *lock*.newCondition();  
  
 **public void** await() {  
 **try** {  
 *lock*.lock();//在condition.await()方法之前调用,获得同步监视器  
 **System**.***out***.println("await()时间:" + **System**.currentTimeMillis());  
 *condition*.await();  
  
 } **catch** (**InterruptedException** e) {  
  
 } **finally** {  
 *lock*.unlock();  
 }  
 }  
  
 **public void** Signal() {  
 **try** {  
 *lock*.lock();  
 **System**.***out***.println("signal()时间:" + **System**.currentTimeMillis());  
 *condition*.signal();  
 } **finally** {  
 *lock*.unlock();  
 }  
 }  
}

**package com.thread.capther4.Lock.demo3**;  
  
**public class Test** {  
 **public static void** main(**String**[] args) **throws InterruptedException** {  
  
 **MyService** myService = **new** MyService();  
 **MyThread** mt = **new** MyThread(myService);  
 mt.start();  
 **Thread**.sleep(**2000**);  
 myService.Signal();  
 }  
}  
  
  
*/\*  
 await()时间:1536549383856  
 signal()时间:1536549385856  
\*/*

### com.thread.capther4.Lock.demo4

锁Lock分为公平锁,非公平锁  
  
  
 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FIFO;  
  
 非公平锁就是  
 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,  
 这个方式可能造成某些线程一直拿不到锁,结果也就是不公平的了;

### com.thread.capther4.Lock.demo5

类ReentrantLock具有完全互斥排他的效果,即统一时间只有一个线程在执行ReentrantLock.lock()方法后面的任务;  
  
  
  
读写锁ReentrantReadWriteLock可以加快执行效率:  
 读写锁rw表示也有两个锁,  
 一个是读操作相关的锁,也称共享锁;  
 另一个是写操作相关的锁,也叫排他锁;  
  
 多个读之间不互斥;  
 读锁与写锁互斥,写锁与写锁互斥;  
  
 多个Thread可以同时进行读取操作,但是同一时刻只允许一个Thread进行写入操作;

**package com.thread.capther4.Lock.demo5**;  
  
**public class ThreadA extends Thread** {  
 **private Service** *service*;  
  
 **public ThreadA**(**Service** service) {  
 **this**.*service* = service;  
 }  
  
 @Override  
 **public void** run() {  
 *service*.read();  
 }  
}

**package com.thread.capther4.Lock.demo5**;  
  
**public class ThreadB extends Thread** {  
 **private Service** *service*;  
  
 **public ThreadB**(**Service** service) {  
 **this**.*service* = service;  
 }  
  
 @Override  
 **public void** run() {  
 *service*.read();  
 }  
}

**package com.thread.capther4.Lock.demo5**;  
  
**import java.util.concurrent.locks.ReentrantReadWriteLock**;  
  
**public class Service** {  
 **private ReentrantReadWriteLock** *lock* = **new** ReentrantReadWriteLock();  
  
 **public void** read() {  
 **try** {  
 *lock*.readLock().lock();  
 **System**.***out***.println("获得锁的线程名称:" + **Thread**.currentThread().getName() + " 时间:" + **System**.currentTimeMillis());  
 **Thread**.sleep(**1000**);  
 } **catch** (**InterruptedException** e) {  
  
 } **finally** {  
 *lock*.readLock().unlock();  
 }  
 }  
}

**package com.thread.capther4.Lock.demo5**;  
  
**public class Test** {  
 **public static void** main(**String**[] args) {  
 **Service** service = **new** Service();  
 **ThreadA** a = **new** ThreadA(service);  
 a.setName("a");  
 a.start();  
  
 **ThreadB** b = **new** ThreadB(service);  
 b.setName("b");  
 b.start();  
 }  
}  
  
  
  
*/\* 获得锁的线程名称:a 时间:1536629889020  
 获得锁的线程名称:b 时间:1536629889020  
  
 几乎同时获得读锁;  
\*/*

### com.thread.capther4.Lock.demo6

rw写写互斥

**package com.thread.capther4.Lock.demo6**;  
  
**public class ThreadA extends Thread** {  
 **private Service** *service*;  
  
 **public ThreadA**(**Service** service) {  
 **this**.*service* = service;  
 }  
  
 @Override  
 **public void** run() {  
 *service*.write();  
 }  
}

**package com.thread.capther4.Lock.demo6**;  
  
  
**public class ThreadB extends Thread** {  
 **private Service** *service*;  
  
 **public ThreadB**(**Service** service) {  
 **this**.*service* = service;  
 }  
  
 @Override  
 **public void** run() {  
 *service*.write();  
 }  
}

**package com.thread.capther4.Lock.demo6**;  
  
**import java.util.concurrent.locks.ReentrantReadWriteLock**;  
  
**public class Service** {  
 **private ReentrantReadWriteLock** *lock* = **new** ReentrantReadWriteLock();  
  
 **public void** write() {  
 **try** {  
 *lock*.writeLock().lock();  
 **System**.***out***.println("获得写锁的线程名称:" + **Thread**.currentThread().getName() + " 时间:" + **System**.currentTimeMillis());  
 **Thread**.sleep(**1000**);  
 } **catch** (**InterruptedException** e) {  
  
 } **finally** {  
 *lock*.writeLock().unlock();  
 }  
 }  
}

**package com.thread.capther4.Lock.demo6**;  
  
**public class Test1** {  
 **public static void** main(**String**[] args) {  
  
 **Service** service = **new** Service();  
 **ThreadA** a = **new** ThreadA(service);  
 a.setName("a");  
 a.start();  
  
 **ThreadB** b = **new** ThreadB(service);  
 b.setName("b");  
 b.start();  
  
  
 }  
}  
  
//获得写锁的线程名称:a 时间:1536630484205  
//获得写锁的线程名称:b 时间:1536630485206

## \*如何实现指定时间执行任务; \*如何实现指定周期执行任务;

com.thread.capther5.Timer