**第1章 Java多线程技能(Thread API)**

1.1 进程和多线程的概念及线程的优点

* **进程:**



一个正在操作系统上执行的exe程序。

* **线程**

进程中独立运行的子任务。

**单任务:**排队执行，即同步。CPU利用率大幅降低。

**1.2 使用多线程**

JVM创建调用main()的线程

1.2.1 继承Thread类

* Public class Thread implements Runnable {}
* CPU以不确定的方式(随机时间)调用线程中run方法。

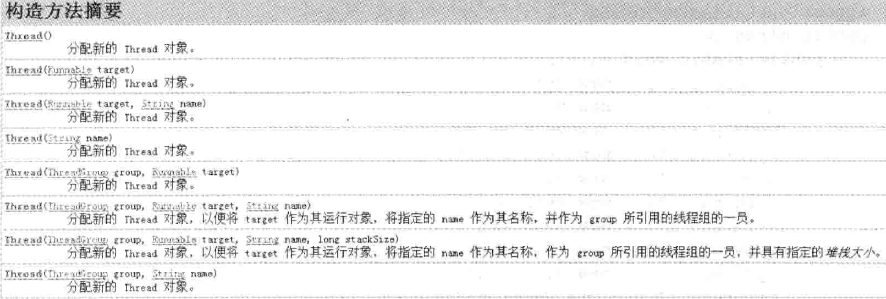
如多次调用start(),会出现Exception in thread “main” java.lang. IllegalThreadStateException

* start()通知”线程规划器”此线程已准备就绪，等待调用线程对象的run()方法，就是让系统安排时间来调用Thread中的run()；线程启动顺序与start()执行顺序无关。
* 如用thread.run()，是同步，此线程对象不交给”线程规划器”处理，而是由main线程调用run()

1.2.2 实现Runnable接口

如欲创建的线程类已有父类，就不能再继承Thread类。

* Thread构造函数：



* 例子:
* Thread(Runnable target, String name)
* Thread(Runnable target)：因Thread implements Runnable，因此也可传入Thread类对象

|  |
| --- |
| package com.thread.myrunable;  public class MyRunable implements Runnable {  @Override  public void run() {  System.out.println("MyRunable运行中!");  }  } |

|  |
| --- |
| package com.thread.myrunable;  public class Run {  public static void main(String[] args) {  MyRunable myRunable = new MyRunable();  Thread thread = new Thread(myRunable);//Thread(Runnable target)  thread.start();  System.out.println("main()运行结束！");  }  } |

结果：

|  |
| --- |
| main()运行结束！  MyRunable运行中! |

1.2.3 【补】实例变量与线程安全

1.2.4 【补】留意i——与System.out.println（）的异常

**1.3** 【补】**currentThread（）方法**

**1.4**【补】 **isAlive（）方法**

**1.5** 【补】**sleep（）方法**

**1.6** 【补】**getId（）方法**

**1.7 停止线程**

* 终止正在运行的线程:

1. run方法完成后线程正常终止
2. Thread.stop()：不安全，且已被废弃，最好不用。
3. Thread.interrupted()

1.7.1 停止不了的线程：interrupt ()

Thread.interrupt ()仅是在当前线程中打了一个停止标记，并不是真正停止线程。

1.7.8 使用return(+interrupt)停止线程

* 例子:

|  |
| --- |
| package com.thread.sleepinterupt;  public class MyThrrad extends Thread{  @Override  public void run() {  while(true) {  if(this.isInterrupted()) {  System.out.println("停止了！");  return;  }  System.out.println("timer=" + System.currentTimeMillis());  }  }  } |

|  |
| --- |
| package com.thread.sleepinterupt;  public class Run2 {  public static void main(String[] args) {  MyThrrad myThrrad = new MyThrrad();  myThrrad.start();    try {  Thread.sleep(300);  } catch (InterruptedException e) {  e.printStackTrace();  }    myThrrad.interrupt();  }  } |

结果:

|  |
| --- |
| timer=1503036078539  timer=1503036078539  timer=1503036078539  timer=1503036078539  timer=1503036078539  timer=1503036078539  timer=1503036078539  timer=1503036078539  timer=1503036078539  timer=1503036078539  timer=1503036078539  timer=1503036078539  timer=1503036078539  timer=1503036078539  timer=1503036078539  timer=1503036078539  timer=1503036078539  timer=1503036078539  停止了！ |

* 总结:

但还是建议用”抛异常”的方法停止线程，因在catch中还可将异常往上抛，使线程停止事件得到传播。

1.7.2 判断线程是否是停止状态

* Thread.interrupted()：测试当前线程是否已中断

|  |
| --- |
| public static boolean interrupted() {  return currentThread().isInterrupted(true);  } |

调用后，线程的中断状态由该方法清除。

* thread.isInterrupted()：测试线程Thread对象是否已中断

|  |
| --- |
| public boolean isInterrupted() {  return isInterrupted(false);  } |

调用后，未清除线程的中断状态。

1.7.3 【补】能停止的线程——异常法

1.7.4 sleep()状态下停止线程

* 例子

|  |
| --- |
| package com.thread.sleepinterupt;  public class MyThrrad extends Thread{  @Override  public void run() {  super.run();  try {  System.out.println("run begin");  Thread.sleep(200000);  System.out.println("run end");  } catch (InterruptedException e) {  System.out.println("在沉睡中停止！进入catch:" + this.isInterrupted());  e.printStackTrace();  }  }  } |

|  |
| --- |
| package com.thread.sleepinterupt;  public class Run2 {  public static void main(String[] args) {  try {  MyThrrad myThrrad = new MyThrrad();  myThrrad.start();  Thread.sleep(200);  myThrrad.interrupt();  } catch (InterruptedException e) {  System.out.println("main catch");  e.printStackTrace();  }  System.out.println("main end!");  }  } |

结果：

|  |
| --- |
| run begin  main end!  在沉睡中停止！进入catch:false  java.lang.InterruptedException: sleep interrupted  at java.lang.Thread.sleep(Native Method)  at com.thread.sleepinterupt.MyThrrad.run(MyThrrad.java:11) |

* 总结:

在sleep状态下停止某一线程，会抛异常，进入catch语句，且清除停止状态，变为false

1.7.5 【补】能停止的线程——暴力停止(stop())

1.7.6 stop()与java.lang.ThreadDeath异常

* 例子:

|  |
| --- |
| package com.thread.sleepinterupt;  public class MyThrrad extends Thread{  @Override  public void run() {  try {  this.~~stop();~~  } catch (ThreadDeath e) {  System.out.println("进入了catch方法！");  e.printStackTrace();  }  }  } |

调用代码(略),结果：

|  |
| --- |
| 进入了catch方法！  java.lang.ThreadDeath  at java.lang.Thread.stop(Thread.java:836)  at com.thread.sleepinterupt.MyThrrad.run(MyThrrad.java:8) |

* 总结:
* 调用stop()会抛出java.lang.ThreadDeath异常，但通常不用显式捕捉。

//public class ThreadDeath extends Error

* stop()已被废弃，因强制停止：

可能使一些清理性工作得不到完成。

另一情况是对锁定对象”解锁”,导致数据得不到同步处理，导致数据不一样问题。

1.7.7【补】 释放锁的不良后果

**1.8 暂停线程（**suspend- resume**）**

1.8.1 【补】suspend与resume方法的使用

suspend(停止)[已废弃] –resume(恢复)

1.8.2 【补】suspend与resume方法的缺点—独占

使用不当（如获得锁对象后suspend），易造成公共同步对象的独占，其它线程无法访问公共同步对象。

1.8.3 suspend与resume方法的缺点—不同步

**1.9 yield方法**

作用:放弃当前CPU资源，让给其它任务区去占用CPU执行时间。

但放弃的时间不确定，可能刚放弃，马上又获得。

**1.10线程的优先级：**setPriority()

public final void setPriority(int newPriority)

优先级较高的线程得到CPU资源较多。

newPriority：1-10, 不在此范围抛出异常:throw new IllegalArgumentException()

|  |
| --- |
| myThrrad.setPriority(Thread.MIN\_PRIORITY);  3个常量:  public final static int MIN\_PRIORITY = 1;  public final static int NORM\_PRIORITY = 5;  public final static int MAX\_PRIORITY = 10; |

1.10.1 【代码补】线程优先级的继承特性

如A线程启动B线程，B的线程优先级同A

1.10.2 优先级具有规则性

高优先级的线程总是大部分先执行完。

当线程优先级差距很大，低优先级先调用，却不一定先执行完

即:线程优先级与代码执行顺序无关，CPU尽量将执行资源让给优先级较高的线程。

1.10.3 优先级具有随机性和不确定性

即优先级较高的线程不一定每次都先执行完。

1.10.4 看谁运行得快

**1.11 守护线程**

|  |
| --- |
| 用户线程  守护线程 |

守护线程：

仅当进程中不存在非守护线程，则守护线程自动销毁。

作用：为其它线程的运行提供便利服务

典型:垃圾回收线程

|  |
| --- |
| MyThrrad myThrrad = new MyThrrad();  myThrrad.setDaemon(true); |

**第2章 对象及变量的并发访问**

**2.1 synchronized同步方法**

“线程安全”:获得的实例变量的值经过了同步处理，不会出现脏读。

2.1.1 方法内的变量为线程安全

“非线程安全”存在于实例变量中，方法内部的私有变量，不存在该问题。

2.1.2 【可写代码】实例变量非线程安全

如多个线程共同访问1个对象中的实例变量，可能出现非线程安全问题

用线程访问的对象中如有多个实例变量，结果可能交叉。

如对象只有一个实例变量，可能覆盖

2.1.3 多个对象多个锁

Synchronized取得的是对象锁，不是把一段代码或方法(函数)当作锁。

如多个线程访问多个对象，JVM会创建多个锁。

2.1.4 synchronized方法与锁对象

1个对象一个锁

* 结论:

1. A线程先持有object对象的Lock锁，B线程可异步方式调用object对象中的非

Synchronized类型方法。

1. A线程先持有object对象的Lock锁，B线程若在此时调用object对象的Synchronized方法则需等待，即同步

2.1.5 脏读

2.1.4中已实现多个线程调用同一方法时，用Synchronized进行同步。虽在赋值时同步，取值时可能出意外，即脏读。

脏读：在读取实例变量时，此值已被其它线程更改了。

解决方法：在取值(getValue())方法前加上synchronized



2.1.6 synchronized锁重入(代码P65)

当一个线程得到一个对象锁，再次请求此对象锁是可得到的。

* 锁重入:自己可再次获得自己的内部锁。

如不可锁重入，会造成死锁。

* 锁重入也支持在父子类继承的环境中(代码P67)。

即子类(的同步方法)可通过“可重入锁”调用父类的同步方法。

2.1.7 出现异常，锁自动释放

当一个线程执行代码出现异常，其持有的锁会自动释放。

2.1.8 同步不可继承(看代码P69)

父类的方法A是synchronized的，子类重写该方法，不能继承synchronized，需在子类方法中添加synchronized

**2.2 synchronized同步语句块**

2.2.1 synchronized方法的弊端(看代码P74)

弊端:运行时间长

2.2.2 synchronized同步代码块的使用(synchronized(this))

2.2.3 用同步代码块解决同步方法的弊端(看代码P76)

缩短时间，加快运行效率。

2.2.4 一半异步，一半同步

不在synchronized块中的是异步执行，在synchronized块中就是同步执行。

2.2.5 synchronized代码块间的同步性

当一个线程访问object的一个synchronized(this)同步代码块时，其它线程对同一个object中所有其它synchronized(this)同步代码块的访问将被阻塞，说明synchronized使用的”对象监视器”是同一个。

2.2.6 synchronized（this）同步代码块锁定当前对象(看代码P82)

* 多个线程调用同一对象中不同名称的synchronized方法或synchronized同步代码块，调用效果是按顺序执行，是同步的，阻塞的。即锁定当前对象的。
* synchronized方法或synchronized(this):

1. 对其它同步方法或synchronized(this)同步代码块呈阻塞状态
2. 同一时间只有一个线程可执行同步方法/同步代码块

2.2.7将任意对象作为对象监视器

* 任意对象:实例变量及方法的参数

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执行synchronized(非this对象x)同步代码块中代码。

(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执行synchronized(非this对象x)同步代码块中代码。

* 优点

如一个类中有很多synchronized方法，虽能实现同步，但会阻塞，影响效率。

synchronized(非this对象x)代码块中程序与同步方法是异步的，不与其他锁this方法争抢this锁，提高效率。

|  |
| --- |
| 自己的理解:  synchronized(this)与synchronized方法的对象监视器都是当前对象，相同。  synchronized(非this对象x)的对象监视器与它们不同。  持有不同对象监视器是异步效果。 |

* 解决脏读问题【可写代码】

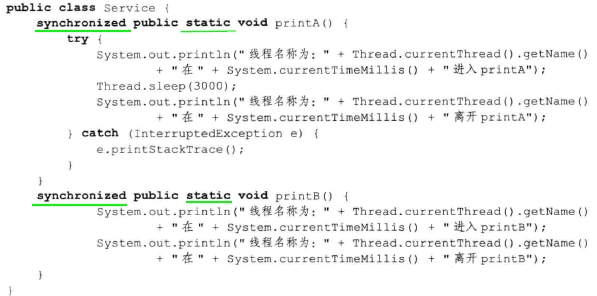
2.2.8 细化验证3个结论

synchronized(非this对象x)将x对象作为”对象监视器”：

1. 多个线程同时执行(同一x对象的)synchronized(x)同步代码块，同步效果
2. 【可写代码】当其它线程执行x对象中synchronized方法或synchronized(this)代码块，同步效果。(**自己理解**:调用synchronized方法或synchronized(this)代码块的那个当前对象是x对象)

注意: 其它线程调用不加synchronized的方法，异步。

2.2.9 静态同步synchronized方法与synchronized(class)代码块



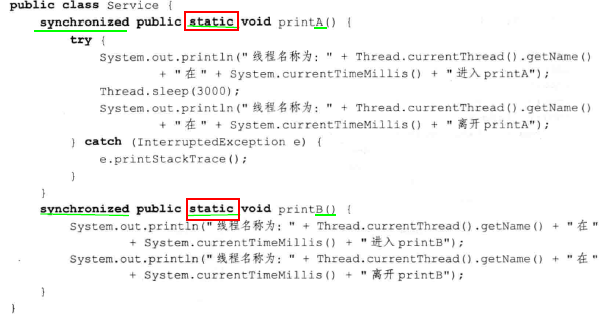
* Synchronized用在静态方法上，是对当前\*.java对应的Class类持锁。

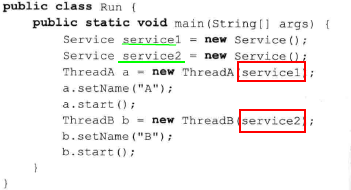
|  |
| --- |
| 比较: Synchronized用在非静态方法上，是给调用的对象上锁。 |



A与B同步，与C异步。因持有不同锁，A和B是Class锁，C是对象锁。

* Class锁可对类的所有实例起作用





结果:同步

* synchronized(class)代码块效果与synchronized static相同

如:synchronized(Service.class)

2.2.10 【可看代码】String的常量池特性

|  |
| --- |
| String a = "a";  String b = "a";  System.out.println(a==b); //true |

String常量池缓存

因此一般synchronized代码块不用String作为锁对象，而用例如new Object()

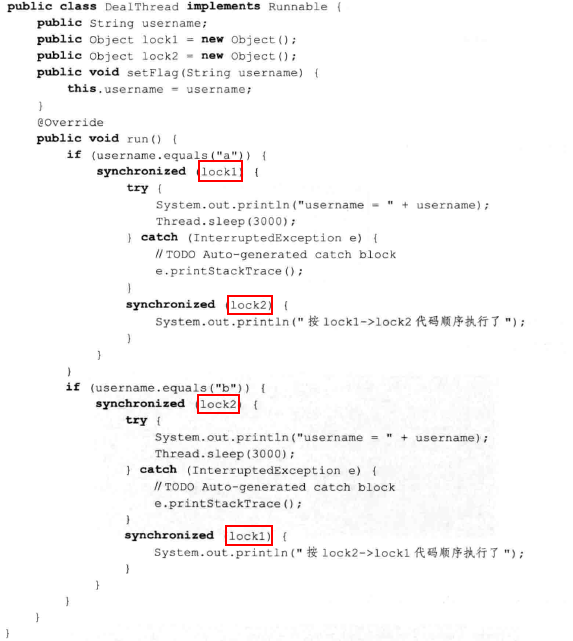
2.2.11 同步synchronized方法无限等待与解决【可看代码】

解决:使非静态方法的锁对象不一样。

2.2.12 多线程的死锁

不同的线程都在等待不可能被释放的锁，导致所有任务都无法继续完成，会造成线程”假死”,必须避免。

例如:



可用JDK自带工具检测是否有死锁。

2.2.13 内置类与静态内置类【可看代码】

2.2.14 内置类与同步：实验1(异步)

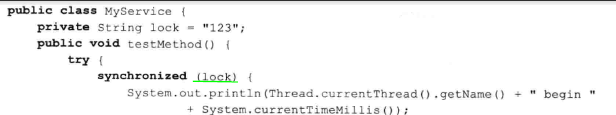
内置类中有两同步方法，但用的不同的锁(不通对象监视器)，结果异步。

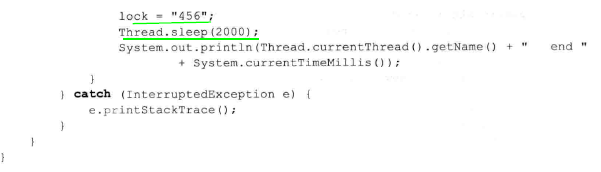
2.2.15 内置类与同步：实验2【代码没太明白】

2.2.16 锁对象的改变【代码没太明白】

只要对象不变，即使对象的属性被改变，依旧同步。

如:



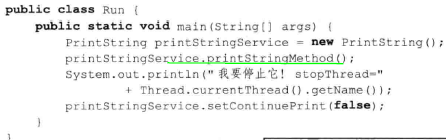


**2.3 volatile关键字**

作用:使变量在多个线程间可见。

**2.3.1死循环**



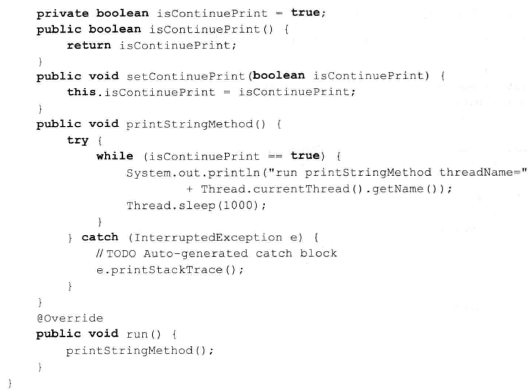


程序陷入死循环，原因:main线程一直在处理while()循环,程序不能执行后续代码。

**2.3.2解决同步死循环(但会出现异步死循环)**

解决思路:用多线程技术



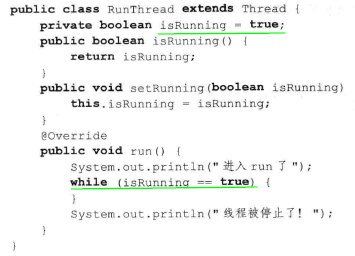




运行在-server服务器模式中64bit的JVM上时，会出现死循环.

**2.3.3 解决异步死循环**

* 引出问题：



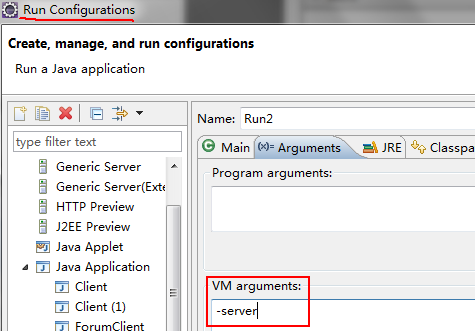


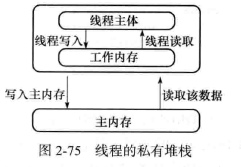
* 在win7结合JDK64bit环境中，用Eclipse运行，无死循环。
* 但当JVM设为server服务时，出现死循环。
* **原因:**

启动RunThread.java线程时，private boolean isRunning=true;存在于公共堆栈和线程私有堆栈中。

JVM设为-server时，为了线程运行效率，线程一直从私有堆栈中取isRunning值，为true.

thread.setRunning(false)更新的是公共堆栈中的isRunning

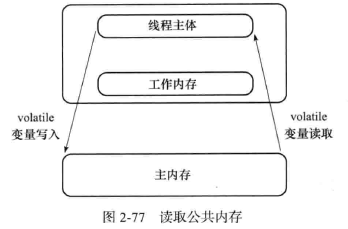




* **实质**:私有堆栈和公有堆栈中的值不同步。

**解决方法**:volatile关键字。当线程访问isRunning,强制从公有堆栈中取值。

修改: 



Volatile增加了实例变量在多个线程间的可见性，但不支持原子性

2.3.3.1 synchronized与volatile比较

1.volatile是线程同步的轻量级实现，性能更好。

2. volatile只能修饰变量，synchronized可修饰方法及代码块。

3.多线程访问volatile不会阻塞，synchronized会阻塞。

4. volatile可保证数据可见性，不能保证原子性。

Synchronized可间接保证可见性，可保证原子性。因它会将私有内存和公有内存中数据同步。(理解:可免于再用volatile)

5. volatile解决变量在多个线程间的可见性

Synchronized解决多个线程访问资源的同步性(原子性)。

注:线程安全—原子性(同步性)+可见性

2.3.4 Volatile非原子的特性

* volatile不具备同步性，也就不具备原子性。

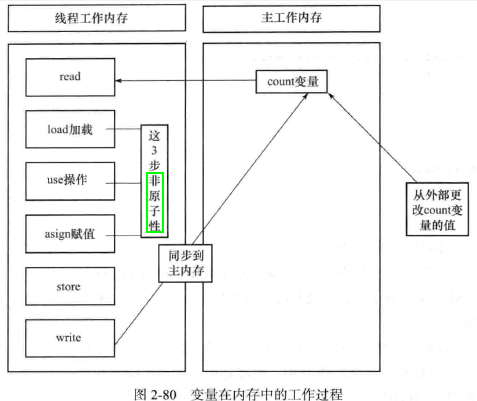
volatile提示线程从共享内存中读取变量

* 例:**i++(非原子操作)，非线程安全**

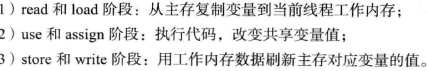
|  |
| --- |
| 1. 从内存中取出i的值 2. 计算i的值 3. 将i的值写到内存 |

假如在(2)步时，另一个线程也修改i的值，就会出现脏数据。需加Synchronized解决。

因此，Volatile本身不处理数据的原子性，而是强制对数据的读写及时影响到主内存。



结论:



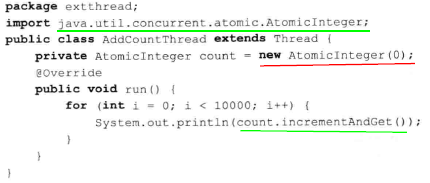
多线程时，use和assign多次出现，但并不是原子性。

在read和load之后，如主内存count变量发生修改，线程工作内存中的值由于已加载，不会感知变化，此时私有和公有内存中变量不同步了。

即用volatile修饰的变量，JVM虚拟机只保证从主内存加载到线程工作内存的值最新。

**2.3.5 使用原子类(AtomicInteger)进行i++操作**

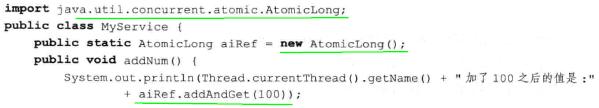
原子操作不可分割，没有锁时依旧线程安全。



**2.3.6 原子类也不完全安全**

在具有逻辑性的情况下也不是线程安全的，因多个原子方法组合后调用为非原子性。

例子:





解决方案: 

**2.3.7** Synchronized代码块有volatile的功能

Synchronized(互斥性+可见性)可使多个线程访问同一资源有同步性。

还会将私有内存和公有内存中变量同步。(理解:可免于再用volatile)

**2.4 本章总结**

**第3章 线程间通信**

线程间通信使线程在大大提高CPU利用率的同时，还使程序员对各线程任务处理过程进行把控和监督。

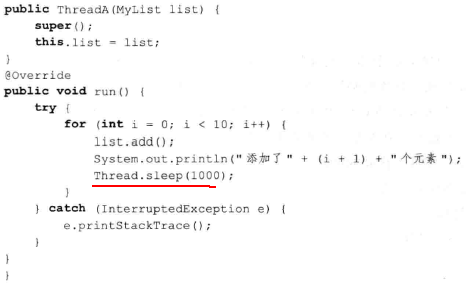
|  |
| --- |
| Wait/notify  生产者/消费者模式  Join  ThreadLocal |

**3.1 等待/通知(**Wait/notify**)机制**

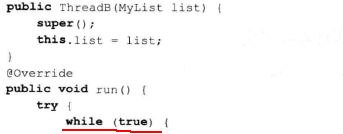
3.1.1 不使用等待/通知机制实现线程间通信:

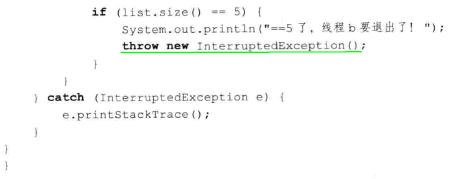
sleep() + [while(true)死循环 + 某条件下抛异常终止线程]

(1)ThreadA



(2) ThreadB





弊端: ThreadB不停通过while死循环检测某一条件，浪费CPU资源。轮询间隔越小，越浪费资源。

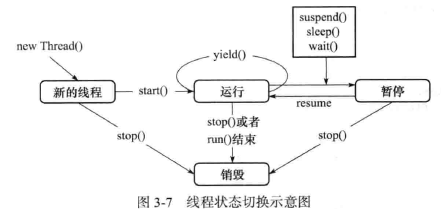
3.1.2 什么是等待/通知机制

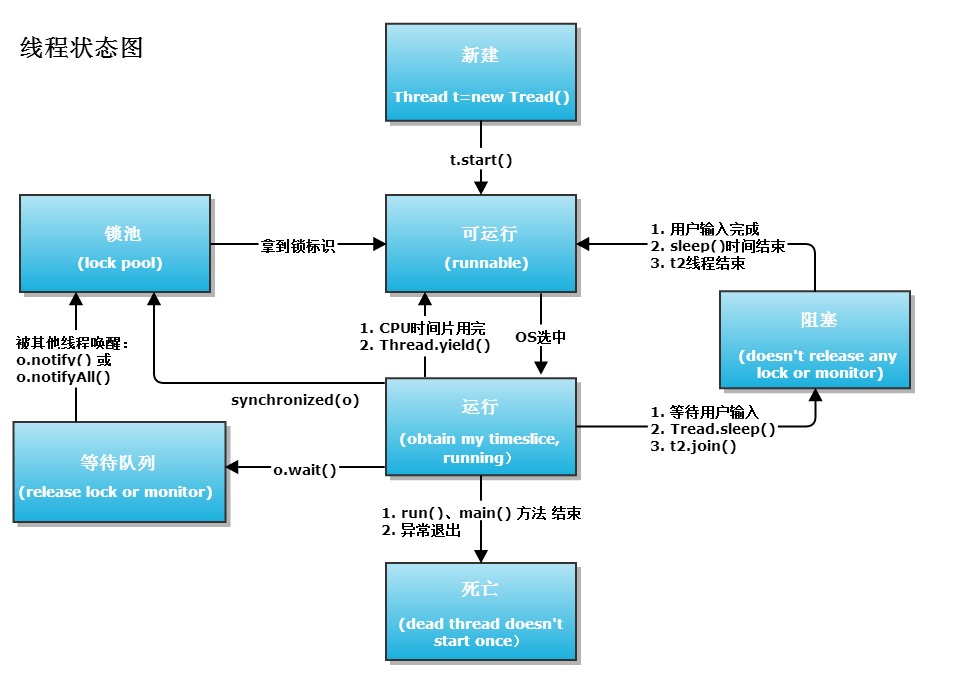
3.1.3 等待/通知机制的实现

* **wait()**
* wait()将当前线程置入”预执行队列”中，且在wait()所在的代码处停止执行。知道接到通知或被中断为止。
* 只能在同步方法或同步块中调用wait()。
* 执行wait()后，当前线程释放锁。
* 如调用wait()时，没持有锁，抛出IllegalMonitorStateException的运行时异常。
* **notify()**
* 也要在同步方法或同步块中调用
* 如调用notify ()时，没持有锁，抛出IllegalMonitorStateException的运行时异常。
* 如有多个线程等待，由线程规划器随机挑选其中一个wait状态的线程，对其发出notify，并使它等待获取该对象锁。
* 某线程执行完notify()后，不会马上释放锁，将程序执行完(即执行完synchronized代码块后)，该线程才释放锁。

因此，呈wait状态的线程也不会马上获取该对象锁。

* 当第一个获取该对象锁的wait线程运行完毕后，会释放掉该对象锁。此时，如该对象没有再次使用notify语句，则即使该对象已空闲，其它wait状态的线程由于收不到通知，还会继续阻塞在wait状态【理解:wait状态只能被唤醒】，直到该对象发出notify或notifyAll。
* 线程状态图:





1. 新建一个线程对象后，调用其start()，系统会为该线程分配CPU资源，使其处于Runable(可运行)状态。如线程抢到CPU资源，就处于Running(运行)状态
2. Runable与Running状态可相互切换。如运行一段时间被高优先级线程抢占CPU资源。
3. 线程进入Runable状态

|  |
| --- |
| 调用sleep()后时间到了  线程调用的阻塞IO已返回，阻塞方法执行完毕  线程获得了试图同步的监视器  线程等待通知，其它线程发出了通知。  处于挂起状态的线程调用了resume() |

1. Blocked（阻塞-暂停）状态

|  |
| --- |
| * 调用sleep(),主动放弃占用的CPU资源 * 调用了阻塞式IO方法，方法返回前，线程被阻塞 * 线程试图获取锁对象，但该锁对象正被其它线程持有。 * 线程等待某个通知 * 程序调用了suspend()将该线程挂起，易导致死锁，尽量避免。 |

每个锁对象都有两个队列:

|  |
| --- |
| 就绪队列: 存储将要获得锁的线程  阻塞队列：被阻塞的线程 |

一个线程被唤醒后，才会进入就绪队列，等待CPU调度

一个线程被wait后，就会进入阻塞队列，等待被唤醒

3.1.4 方法wait()锁释放与notify()锁不释放

3.1.5 当interrupt方法遇到wait方法

wait状态下调用interrupt(),会出现InterruptedException异常。

|  |
| --- |
| 执行完同步代码块会释放对象的锁；  执行同步代码块过程中，遇到异常导致线程终止，锁会被释放  执行同步代码块过程中，执行wait(),线程会释放对象锁，此线程对象进入线程等待池，等待被唤醒。 |

3.1.6 只通知一个线程

notify()一次只随机通知一个线程进行唤醒。

3.1.7 唤醒所有线程(notifyAll)

3.1.8 wait(long)的使用

等待某一时间内是否有线程对锁进行唤醒，如超时，则自动唤醒。

3.1.9 通知过早

通知过早，会打乱程序正常的运行逻辑。可能仅执行notify方法。

3.1.10 等待wait的条件发生变化

此时也会造成程序逻辑混乱

3.1.11 生产者/消费者模式实现

3.1.1.11.1 一生产一消费者:操作值[代码]

|  |
| --- |
| 消费者:空，等待；不空，唤醒  生产者:空，唤醒；不空，等待 |

3.1.1.11.2 多生产与多消费者:假死[代码]

假死:全部线程进入waitting状态，程序不再执行任何业务功能

比如:有可能生产者唤醒生产者，消费者唤醒消费者，导致大家都在等待。

原因：有可能连续唤醒同类。

解决方法:同类异类一起唤醒(用notifyAll)

3.1.1.11.3多生产与多消费:操作值(用notifyAll)

3.1.1.11.4一生产与一消费:操作栈【代码没仔细看】

private List list = new ArrayList();

生产者向堆栈List对象中放入数据，消费者从中取出数据。List的最大容量是1.

3.1.1.11.5一生产与多消费—操作栈：解决wait条件改变与假死【代码没仔细看】

3.1.1.11.5多生产与一消费—操作栈【代码没仔细看】

3.1.1.11.6多生产与多消费—操作栈【good】【代码没仔细看】

3.1.12 通过管道进行线程间通信：字节流【代码没仔细看】

管道流:在不同线程间直接传送数据。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取数据。

使两个stream间产生通信链路。

|  |
| --- |
| PipedInputStream PipedOutputStream |

3.1.13 通过管道进行线程间通信：字符流

|  |
| --- |
| PipedReader PipedWriter |

3.1.14 实战：等待/通知之交叉备份【代码没仔细看】

**3.2 方法join的使用**

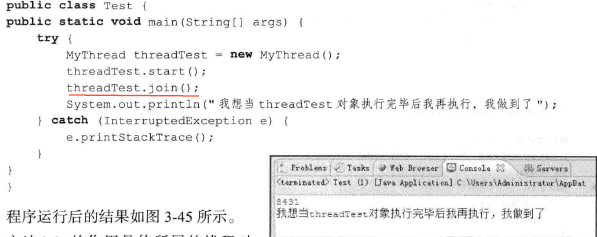
主线程创建并启动子线程，如子线程中要进行大量耗时运算，主线程将早于子线程结束。

如主线程想等子线程执行完后再结束(比如主线程想取得子线程处理好的某个数据)，用join()

Join:等待线程对象销毁

3.2.1 学习join前的铺垫

3.2.2 用join（）方法来解决



* 使所属线程对象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，等线程x销毁后再继续执行线程z后面的代码。
* Join具有使线程排队运行的作用。
* join与sychronized区别：

|  |
| --- |
| Join内部用wait()方法进行等待  Sychronized用对象监视器原理做同步 |

3.2.3 方法join与异常

join过程中，如当前线程对象被中断(interrupt())，则当前线程出现异常(InterruptedException)。

即join()遇到interrupt()，异常，但进程按钮还是“红色”，因被调用的线程还在继续运行。

|  |
| --- |
| 理解: Join内部用wait()方法进行等待,wait()与interrupt()一起用时，异常 |

3.2.4 方法join(long)的使用

如:join(2000) //只等2秒

3.2.5 方法join(long)与sleep(long)的区别

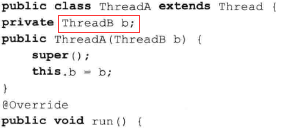
join(long)内部用wait(long)实现，所以会释放锁。

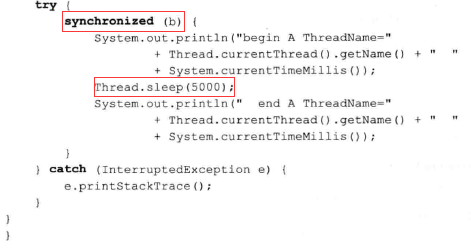
而Thread.sleep(long)不释放锁。

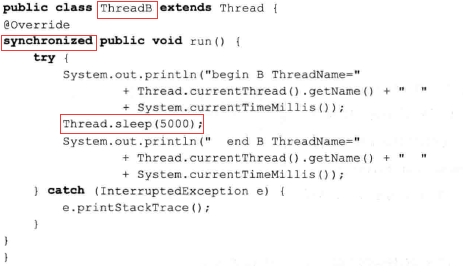
3.2.6 join()后的代码提前运行：出现意外[见代码]

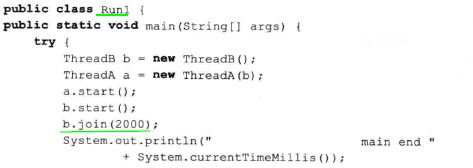
3.2.7 join()后的代码提前运行：解释意外

例子:

1.代码：  


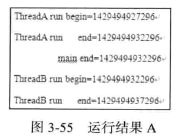


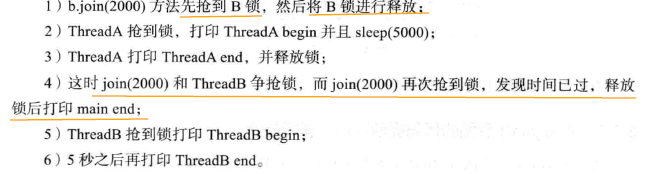




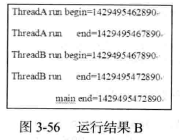
2.结果：

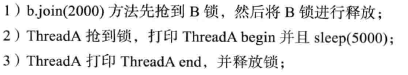
(1)结果A



解释：  


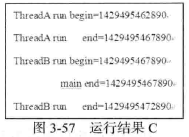
(2)结果B

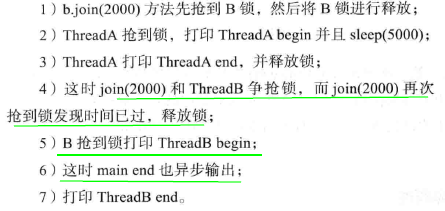


解释：  




(3)结果C



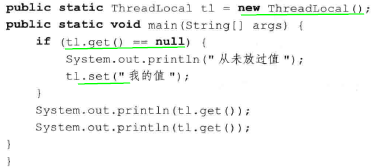
解释:  


3.3 类ThreadLocal的使用

实现每个线程都有自己的私有数据。

ThreadLocal解决的是每个线程绑定自己的值。

3.3.1 方法get()与null

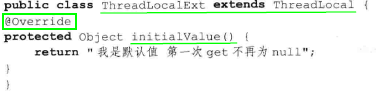




ThreadLocal解决的是变量在不同线程间的隔离性，就是不同线程有自己的值，可放入ThreadLocal中保存。

3.3.2 验证线程变量的隔离性

3.3.3 解决get()返回null问题



覆盖initialValue()方法使get()具有初始值。

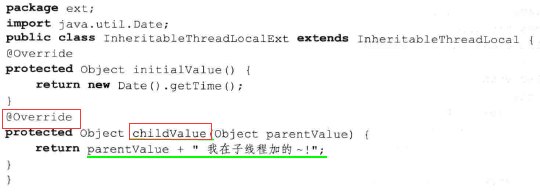
3.3.4 再次验证线程变量的隔离性[代码没仔细看]

3.4 类InheritableThreadLocal的使用

可在子线程中取得父线程继承下来的值。

3.4.1 值继承[代码没仔细看]

3.4.2 值继承再修改



如子线程取值时，主线程将InheritableThreadLocal中的值更改了，子线程取的还是旧值。

**第4章 Lock的使用**

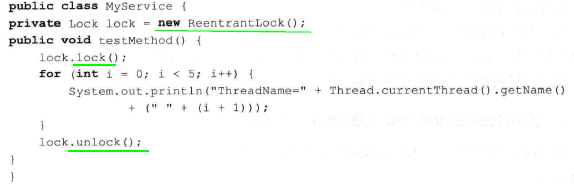
4.1 使用ReentrantLock类

JDK1.5中增加ReentrantLock类，可达到与synchronized同样的效果。

且在扩展功能上更强大，如嗅探锁定、多路分支通知等

使用上也比synchronized更灵活

4.1.1 使用ReentrantLock实现同步：测试1



4.1.2 使用ReentrantLock实现同步：测试2

调用lock.lock()代码的线程就持有了”对象监视器”，其它线程只能等待锁释放时再次争抢。

4.1.3 使用Condition实现等待/通知:错误用法与解决

1. Synchronized与wait()和notify()/notifyAll()结合，可实现等待/通知模式。

此时，被通知的线程是由JVM随机选择的。

相当于整个Lock对象中只有单一的condition对象。

2. ReentrantLock的condition+

Condition可实现多路通知，即在一个Lock对象中可创建多个condition(即对象监视器)实例，线程对象可注册在指定的condition中，从而可选择性的进行线程通知。

3．condition.await()之前必须调用lock.lock()获得同步监视器。

否则出现IllegalMonitorStateException异常。（与wait方法一样的情况）

4.1.4 正确使用Condition实现等待/通知

|  |
| --- |
| package com.thread.lock.reentrantlock;  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();//非公平锁  private Condition condition = lock.newCondition();    public void await() {  try {  lock.lock();  System.out.println("await时间为:" + System.currentTimeMillis());  condition.await();  } catch (InterruptedException e) {  e.printStackTrace();  } finally{  lock.unlock();  System.out.println("await锁释放了");  }  }    public void signal() {  try {  lock.lock();  System.out.println("signal时间为:" + System.currentTimeMillis());  condition.signal();  } catch (Exception e) {  } finally {  lock.unlock();  System.out.println("signal锁释放了");  }  }  } |

ThreadA.java

|  |
| --- |
| package com.thread.lock.reentrantlock;  public class ThreadA extends Thread{  private MyService service;  public ThreadA(MyService service) {  super();  this.service = service;  }  @Override  public void run() {  service.await();  }  } |

|  |
| --- |
| package com.thread.lock.reentrantlock;  public class Run {  public static void main(String[] args) {  MyService myService = new MyService();  ThreadA threadA = new ThreadA(myService);  threadA.start();  try {  Thread.sleep(3000);  } catch (InterruptedException e) {  e.printStackTrace();  }  myService.signal();  }  } |

结果：

|  |
| --- |
| await时间为:1503317900324  signal时间为:1503317903324  signal锁释放了  await锁释放了 |

* Object类中的wait()方法相当于condition类中的await()
* Object类中的wait(long timeout)方法相当于condition类中的await(long time, TimeUnit unit)
* Object类中的notify)方法相当于condition类中的signal()
* Object类中的notifyAll()方法相当于condition类中的signalAll()

4.1.5 使用多个Condition实现通知部分线程：错误用法

4.1.6 使用多个Condition实现通知部分线程：正确用法

4.1.7 实现生产者/消费者模式：一对一交替打印

4.1.8 实现生产者/消费者模式：多对多交替打印

4.1.9 公平锁与非公平锁

4.1.10 方法getHoldCount（）、getQueueLength（）和getWaitQueueLength（）的测试

4.1.11 方法hasQueuedThread（）、hasQueuedThreads（）和hasWaiters（）的测试

4.1.12 方法isFair（）、isHeldByCurrentThread（）和isLocked（）的测试

4.1.13 方法lockInterruptibly（）、tryLock（）和tryLock（long timeout，TimeUnit unit）的测试

4.1.14 方法awaitUninterruptibly（）的使用

4.1.15 方法awaitUntil（）的使用

4.1.16 使用Condition实现顺序执行

4.2 使用ReentrantReadWriteLock类

4.2.1 类ReentrantReadWriteLock的使用：读读共享

4.2.2 类ReentrantReadWriteLock的使用：写写互斥

4.2.3 类ReentrantReadWriteLock的使用：读写互斥

4.2.4 类ReentrantReadWriteLock的使用：写读互斥

4.3 本章总结

**第5章 定时器Timer**

5.1 定时器Timer的使用

5.1.1 方法schedule（TimerTask task， Date time）的测试

5.1.2 方法schedule（TimerTask task， Date firstTime， long period）的测试

5.1.3 方法schedule（TimerTask task， long delay）的测试

5.1.4 方法schedule（TimerTask task， long delay， long period）的测试

5.1.5 方法scheduleAtFixedRate（TimerTask task， Date firstTime， long period）的测试

5.2 本章总结

**第6章 单例模式与多线程**

6.1 立即加载/"饿汉模式"

6.2 延迟加载/"懒汉模式"

6.3 使用静态内置类实现单例模式

6.4 序列化与反序列化的单例模式实现

6.5 使用static代码块实现单例模式

6.6 使用enum枚举数据类型实现单例模式

6.7 完善使用enum枚举实现单例模式

6.8 本章总结

**第7章 拾遗增补**

7.1 线程的状态

7.1.1 验证NEW、RUNNABLE和TERMINATED

7.1.2 验证TIMED\_WAITING

7.1.3 验证BLOCKED

7.1.4 验证WAITING

7.2 线程组

7.2.1 线程对象关联线程组：1级关联

7.2.2 线程对象关联线程组：多级关联

7.2.3 线程组自动归属特性

7.2.4 获取根线程组

7.2.5 线程组里加线程组

7.2.6 组内的线程批量停止

7.2.7 递归与非递归取得组内对象

7.3 使线程具有有序性

7.4 SimpleDateFormat非线程安全

7.4.1 出现异常

7.4.2 解决异常方法1

7.4.3 解决异常方法2

7.5 线程中出现异常的处理

7.6 线程组内处理异常

7.7 线程异常处理的传递

7.8 本章总结

第一章



**1.2.3 实例变量与线程安全**

**1.2.3.1 不共享数据**

(1)

|  |
| --- |
| package com.thread.www;  public class MyThread3 extends Thread{  private int count = 5;  public MyThread3(String name) {  super();  this.setName(name);//设置线程名称  }  @Override  public void run() {  super.run();  while(count > 0) {  count--;  System.out.println("由" + this.currentThread().getName() + "计算，count=" +  count );  }  }  } |

(2)

|  |
| --- |
| package test;  import com.thread.www.MyThread3;  public class RunMyThread3 {  public static void main(String[] args) {  MyThread3 a = new MyThread3("A");  MyThread3 b = new MyThread3("B");  MyThread3 c = new MyThread3("C");  a.start();  b.start();  c.start();  }  } |

(3)结果:一共创建了3线程，每个线程都有各自的count变量，自己减少自己的count变量。变量不共享

|  |
| --- |
| 由A计算，count=3  由A计算，count=2  由A计算，count=1  由A计算，count=0  由C计算，count=4  由B计算，count=4  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C计算，count=2  由C计算，count=1  由C计算，count=0 |

**1.2.3.2 共享数据**

1.2.3.2.1 例1:出现非线程安全问题

(1) MyThread4.java

|  |
| --- |
| package com.thread.www;  public class MyThread4 extends Thread {  private int count = 5;  @Override  public void run() {  super.run();  count--;  //此示例不要用for语句，因为使用同步后，其它线程就得不到线程运行的机会了，  //会一直由一个线程做减法运算  System.out.println("由" + this.currentThread().getName() + "计算count=" + count);  }  } |

(2) TestMyThread4.java

|  |
| --- |
| package test;  import com.thread.www.MyThread4;  public class TestMyThread4 {  public static void main(String[] args) {  MyThread4 myThread4 = new MyThread4();  Thread a = new Thread(myThread4, "a");  Thread b = new Thread(myThread4, "b");  Thread c = new Thread(myThread4, "c");  Thread d = new Thread(myThread4, "d");  Thread e = new Thread(myThread4, "e");  a.start();  b.start();  c.start();  d.start();  e.start();  }  } |

(3)结果: c和b线程同时对count进行处理，产生”非线程安全” 问题

|  |
| --- |
| 由a计算count=4  由c计算count=2  由b计算count=2  由e计算count=1  由d计算count=0 |

1.2.3.2.2 例2:

需要使多个线程间同步，即按顺序排队的方式进行减1操作

(1) 在例1基础上，更改 MyThread4.java代码如下：

|  |
| --- |
| package com.thread.www;  public class MyThread4 extends Thread {  private int count = 5;  @Override  synchronized public void run() {  super.run();  count--;  //此示例不要用for语句，因为使用同步后，其它线程就得不到线程运行的机会了，  //会一直由一个线程做减法运算  System.out.println("由" + this.currentThread().getName() + "计算count=" + count);  }  } |

(2)结果:

|  |
| --- |
| 由a计算count=4  由b计算count=3  由d计算count=2  由c计算count=1  由e计算count=0 |

synchronized使多个线程在执行run方法时，以排队方式处理。当一个线程调用run前，先判断run方法有没被上锁，若上锁，说明有其它线程在调用run方法，必须等其它线程对run方法调用结束后才可执行run方法。

Synchronized可在任意对象及方法前加锁，被加锁的代码称为”互斥区”或“临界区”。

如果不能拿到锁，该线程会不断的去尝试拿这把锁，直到能够拿到为止，且多个线程会同时争抢这把锁。

非线程安全:多个线程对同一个对象中的同一个实例变量进行操作时，会出现值更改、值不同步的情况，进而影响程序执行流程。

**1.2.4 留意i—与System.out.println()的异常**

1. MyThread5.java

|  |
| --- |
| package com.thread.www;  public class MyThread5 extends Thread {  private int i = 5;  @Override  public void run() {  System.out.println("i=" + (i--) + "threadname=" + Thread.currentThread().getName());  }  } |

2. TestMyThread5.java

|  |
| --- |
| package test;  import com.thread.www.MyThread5;  public class TestMyThread5 {  public static void main(String[] args) {  MyThread5 myThread5 = new MyThread5();  Thread a = new Thread(myThread5);  Thread b = new Thread(myThread5);  Thread c = new Thread(myThread5);  Thread d = new Thread(myThread5);  Thread e = new Thread(myThread5);  a.start();  b.start();  c.start();  d.start();  e.start();  }  } |

3.结果

|  |
| --- |
| i=5threadname=Thread-1  i=3threadname=Thread-4  i=4threadname=Thread-2  i=2threadname=Thread-3  i=1threadname=Thread-5 |

虽然println()方法在内部是同步的，但i--操作是在println之前发生的，所以有非线程同步的问题的概率。

|  |
| --- |
| public void println(String x) {  synchronized (this) {  print(x);  newLine();  }  } |

**1.3 currentThread()**

**3.3 ThreadLocal**

<http://blog.csdn.net/lufeng20/article/details/24314381>