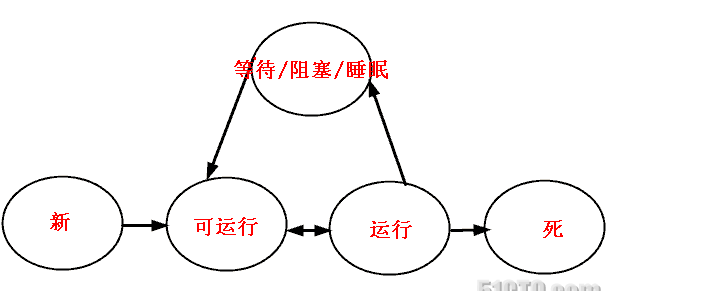
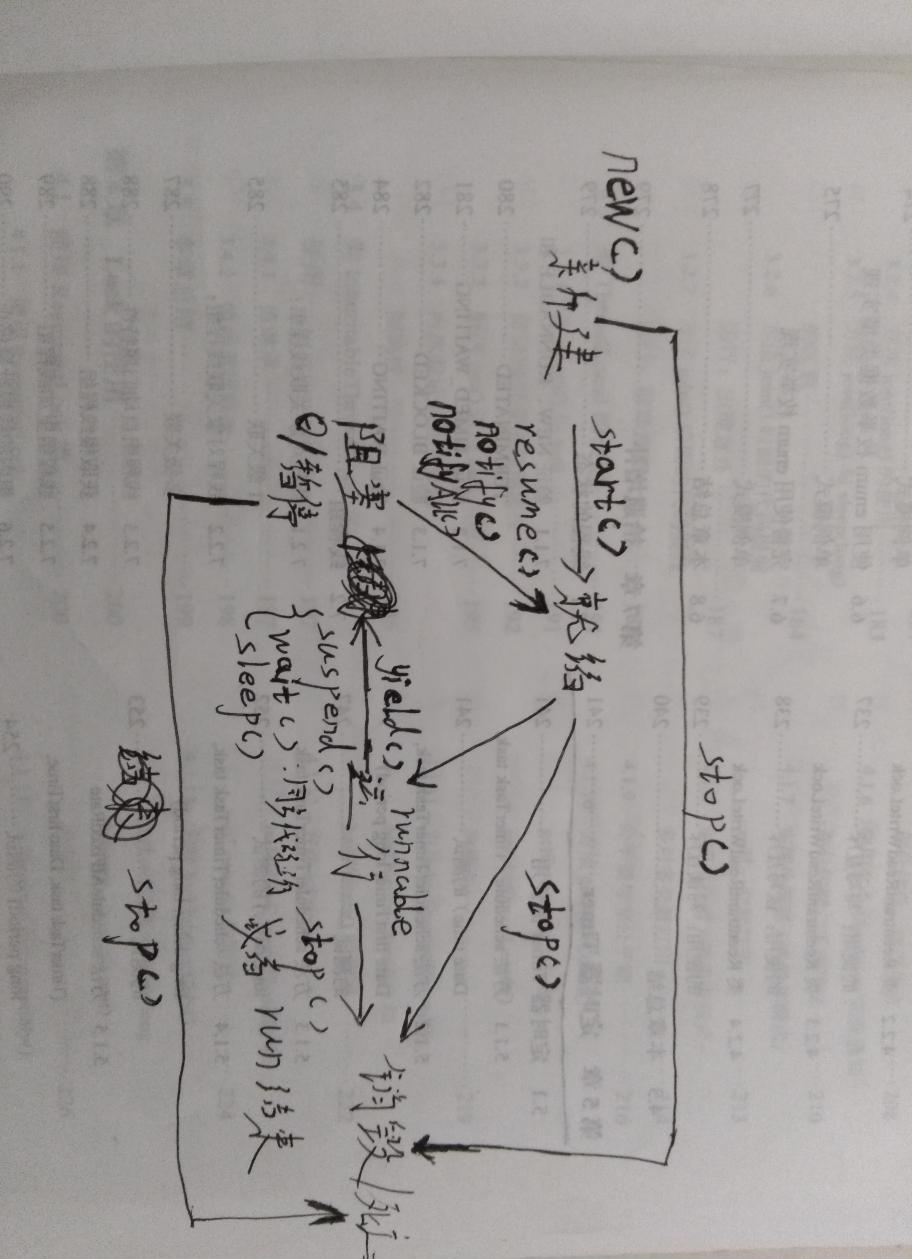
**窗体底端**

**[Java线程详解](http://www.cnblogs.com/riskyer/p/3263032.html)**



**Java线程：概念与原理**

**一、操作系统中线程和进程的概念**

**现在的操作系统是多任务操作系统。多线程是实现多任务的一种方式。**

**进程是指一个内存中运行的应用程序，每个进程都有自己独立的一块内存空间，一个进程中可以启动多个线程。比如在Windows系统中，一个运行的exe就是一个进程。**

**线程是指进程中的一个执行流程，一个进程中可以运行多个线程。比如java.exe进程中可以运行很多线程。线程总是属于某个进程，进程中的多个线程共享进程的内存。**

**“同时”执行是人的感觉，在线程之间实际上轮换执行。**

**二、Java中的线程**

**在Java中，“线程”指两件不同的事情：  
1、java.lang.Thread类的一个实例；**

**2、线程的执行。**

**使用java.lang.Thread类或者java.lang.Runnable接口编写代码来定义、实例化和启动新线程。**

**一个Thread类实例只是一个对象，像Java中的任何其他对象一样，具有变量和方法，生死于堆上。**

**Java中，每个线程都有一个调用栈，即使不在程序中创建任何新的线程，线程也在后台运行着。**

**一个Java应用总是从main()方法开始运行，mian()方法运行在一个线程内，它被称为主线程。**

**一旦创建一个新的线程，就产生一个新的调用栈。**

**线程总体分两类：用户线程和守候线程。**

**当所有用户线程执行完毕的时候，JVM自动关闭。但是守候线程却不独立于JVM，守候线程一般是由操作系统或者用户自己创建的**

**Java线程：创建与启动**

**一、定义线程**

**1、扩展java.lang.Thread类。**

**此类中有个run()方法，应该注意其用法：**

**public void run()**

**如果该线程是使用独立的Runnable运行对象构造的，则调用该Runnable对象的run方法；否则，该方法不执行任何操作并返回。**

**Thread的子类应该重写该方法。**

**2、实现java.lang.Runnable接口。**

**void run()**

**使用实现接口Runnable的对象创建一个线程时，启动该线程将导致在独立执行的线程中调用对象的run方法。**

**方法run的常规协定是，它可能执行任何所需的操作。**

**二、实例化线程**

**1、如果是扩展java.lang.Thread类的线程，则直接new即可。**

**2、如果是实现了java.lang.Runnable接口的类，则用Thread的构造方法：**

**Thread(Runnable target)   
Thread(Runnable target, String name)   
Thread(ThreadGroup group, Runnable target)   
Thread(ThreadGroup group, Runnable target, String name)   
Thread(ThreadGroup group, Runnable target, String name, long stackSize)**

**三、启动线程**

**在线程的Thread对象上调用start()方法，而不是run()或者别的方法。**

**在调用start()方法之前：线程处于新状态中，新状态指有一个Thread对象，但还没有一个真正的线程。**

**在调用start()方法之后：发生了一系列复杂的事情**

**启动新的执行线程（具有新的调用栈）；**

**该线程从新状态转移到可运行状态；**

**当该线程获得机会执行时，其目标run()方法将运行。**

**注意：对Java来说，run()方法没有任何特别之处。像main()方法一样，它只是新线程知道调用的方法名称(和签名)。因此，在Runnable上或者Thread上调用run方法是合法的。但并不启动新的线程。**

**四、例子**

**1、实现Runnable接口的多线程例子**

**/\*\*   
\* 实现Runnable接口的类  
\*   
\* @author leizhimin 2008-9-13 18:12:10   
\*/   
public class DoSomething implements  Runnable {  
    private String name;  
  
    public DoSomething(String name) {  
        this.name = name;  
    }   
  
    publicvoid run() {  
        for (int i = 0; i < 5; i++) {  
            for (long k = 0; k < 100000000; k++) ;  
            System.out.println(name + ": " + i);  
        }   
    }   
}**

**/\*\*   
\* 测试Runnable类实现的多线程程序  
\*   
\* @author leizhimin 2008-9-13 18:15:02   
\*/   
publicclass TestRunnable {  
    publicstaticvoid main(String[] args) {  
        DoSomething ds1 = new DoSomething("阿三");  
        DoSomething ds2 = new DoSomething("李四");  
  
        Thread t1 = new Thread(ds1);  
        Thread t2 = new Thread(ds2);  
  
        t1.start();   
        t2.start();   
    }   
}**

**执行结果：**

**李四: 0  
阿三: 0   
李四: 1   
阿三: 1   
李四: 2   
李四: 3   
阿三: 2   
李四: 4   
阿三: 3   
阿三: 4   
  
Process finished with exit code 0**

**2、扩展Thread类实现的多线程例子**

**/\*\*   
\* 测试扩展Thread类实现的多线程程序  
\*   
\* @author leizhimin 2008-9-13 18:22:13   
\*/   
public class TestThread extends Thread{   
    public TestThread(String name) {  
        super(name);  
    }   
  
    public void  run() {  
        for(int i = 0;i<5;i++){  
            for(long k= 0; k <100000000;k++);  
            System.out.println(this.getName()+" :"+i);  
        }   
    }   
  
    public static void  main(String[] args) {  
        Thread t1 = new TestThread("阿三");  
        Thread t2 = new TestThread("李四");  
        t1.start();   
        t2.start();   
    }   
}**

**执行结果：**

**阿三 :0  
李四 :0   
阿三 :1   
李四 :1   
阿三 :2   
李四 :2   
阿三 :3   
阿三 :4   
李四 :3   
李四 :4   
  
Process finished with exit code 0**

**对于上面的多线程程序代码来说，输出的结果是不确定的。其中的一条语句for(long k= 0; k <100000000;k++);是用来模拟一个非常耗时的操作的。**

**五、一些常见问题**

**1、线程的名字，一个运行中的线程总是有名字的，名字有两个来源，一个是虚拟机自己给的名字，一个是你自己的定的名字。在没有指定线程名字的情况下，虚拟机总会为线程指定名字，并且主线程的名字总是mian，非主线程的名字不确定。**

**2、线程都可以设置名字，也可以获取线程的名字，连主线程也不例外。**

**3、获取当前线程的对象的方法是：Thread.currentThread()；**

**4、在上面的代码中，只能保证：每个线程都将启动，每个线程都将运行直到完成。一系列线程以某种****顺序启动并不意味着将按该顺序执行。对于任何一组启动的线程来说，调度程序不能保证其执行次序，持续时间也无法保证。**

**5、当线程目标run()方法结束时该线程完成。**

**6、一旦线程启动，它就永远不能再重新启动。只有一个新的线程可以被启动，并且只能一次。一个可运行的线程或死线程可以被重新启动。**

**7、线程的调度是JVM的一部分，在一个CPU的机器上上，实际上一次只能运行一个线程。一次只有一个线程栈执行。JVM线程调度程序决定实际运行哪个处于可运行状态的线程。**

**众多可运行线程中的某一个会被选中做为当前线程。可运行线程被选择运行的顺序是没有保障的。**

**8、尽管通常采用队列形式，但这是没有保障的。队列形式是指当一个线程完成“一轮”时，它移到可运行队列的尾部等待，直到它最终排队到该队列的前端为止，它才能被再次选中。事实上，我们把它称为可运行池而不是一个可运行队列，目的是帮助认识线程并不都是以某种有保障的顺序排列唱呢个一个队列的事实。**

**9、尽管我们没有无法控制线程调度程序，但可以通过别的方式来影响线程调度的方式。**

**Java线程：线程栈模型与线程的变量**

**要理解线程调度的原理，以及线程执行过程，必须理解线程栈模型。**

**线程栈是指某时刻时内存中线程调度的栈信息，当前调用的方法总是位于栈顶。线程栈的内容是随着程序的运行动态变化的，因此研究线程栈必须选择一个运行的时刻（实际上指代码运行到什么地方)。**

**Java线程：线程状态的转换**

**一、线程状态**

**线程的状态转换是线程控制的基础。线程状态总的可分为五大状态：分别是生、死、可运行、运行、等待/阻塞。用一个图来描述如下：**

**1、新状态：线程对象已经创建，还没有在其上调用start()方法。**

**2、可运行状态：当线程有资格运行，但调度程序还没有把它选定为运行线程时线程所处的状态。当start()方法调用时，线程首先进入可运行状态。在线程运行之后或者从阻塞、等待或睡眠状态回来后，也返回到可运行状态。**

**3、运行状态：线程调度程序从可运行池中选择一个线程作为当前线程时线程所处的状态。这也是线程进入运行状态的唯一一种方式。**

**4、等待/阻塞/睡眠状态：这是线程有资格运行时它所处的状态。实际上这个三状态组合为一种，其共同点是：线程仍旧是活的，但是当前没有条件运行。换句话说，它是可运行的，但是如果某件事件出现，他可能返回到可运行状态。**

**5、死亡态：当线程的run()方法完成时就认为它死去。这个线程对象也许是活的，但是，它已经不是一个单独执行的线程。线程一旦死亡，就不能复生。如果在一个死去的线程上调用start()方法，会抛出java.lang.IllegalThreadStateException异常。**

**有关详细状态转换图可以参看本人的“[Java多线程编程总结](http://lavasoft.blog.51cto.com/62575/27069" \t "_blank)”中的图**

**二、阻止线程执行**

**对于线程的阻止，考虑一下三个方面，不考虑IO阻塞的情况：**

**睡眠；**

**等待；**

**因为需要一个对象的锁定而被阻塞。**

**1、睡眠**

**Thread.sleep(long millis)和Thread.sleep(long millis, int nanos)静态方法强制当前正在执行的线程休眠（暂停执行），以“减慢线程”。当线程睡眠时，它入睡在某个地方，在苏醒之前不会返回到可运行状态。当睡眠时间到期，则返回到可运行状态。**

**线程睡眠的原因：线程执行太快，或者需要强制进入下一轮，因为Java规范不保证合理的轮换。**

**睡眠的实现：调用静态方法。**

**try {  
            Thread.sleep(123);  
        } catch (InterruptedException e) {  
            e.printStackTrace();    
        }**

**睡眠的位置：为了让其他线程有机会执行，可以将Thread.sleep()的调用放线程run()之内。这样才能保证该线程执行过程中会睡眠。**

**例如，在前面的例子中，将一个耗时的操作改为睡眠，以减慢线程的执行。可以这么写：**

**public void run() {  
        for(int i = 0;i<5;i++){**

**// 很耗时的操作，用来减慢线程的执行  
//            for(long k= 0; k <100000000;k++);  
            try {  
                Thread.sleep(3);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  .  
            }  
            System.out.println(this.getName()+" :"+i);  
        }  
    }**

**运行结果：**

**阿三 :0  
李四 :0   
阿三 :1   
阿三 :2   
阿三 :3   
李四 :1   
李四 :2   
阿三 :4   
李四 :3   
李四 :4   
  
Process finished with exit code 0**

**这样，线程在每次执行过程中，总会睡眠3毫秒，睡眠了，其他的线程就有机会执行了。**

**注意：**

**1、线程睡眠是帮助所有线程获得运行机会的最好方法。**

**2、线程睡眠到期自动苏醒，并返回到可运行状态，不是运行状态。sleep()中指定的时间是线程不会运行的最短时间。因此，sleep()方法不能保证该线程睡眠到期后就开始执行。**

**3、sleep()是静态方法，只能控制当前正在运行的线程。**

**下面给个例子：**

**/\*\*   
\* 一个计数器，计数到100，在每个数字之间暂停1秒，每隔10个数字输出一个字符串  
\*   
\* @author leizhimin 2008-9-14 9:53:49   
\*/   
publicclass MyThreadextends Thread {  
  
    publicvoid run() {  
        for (int i = 0; i < 100; i++) {  
            if ((i) % 10 == 0) {  
                System.out.println("-------" + i);  
            }   
            System.out.print(i);   
            try {  
                Thread.sleep(1);   
                System.out.print("    线程睡眠1毫秒！\n");  
            } catch (InterruptedException e) {  
                e.printStackTrace();   
            }   
        }   
    }   
  
    publicstaticvoid main(String[] args) {  
        new MyThread().start();  
    }   
}**

**-------0   
0    线程睡眠1毫秒！  
1    线程睡眠1毫秒！  
2    线程睡眠1毫秒！  
3    线程睡眠1毫秒！  
4    线程睡眠1毫秒！  
5    线程睡眠1毫秒！  
6    线程睡眠1毫秒！  
7    线程睡眠1毫秒！  
8    线程睡眠1毫秒！  
9    线程睡眠1毫秒！  
-------10   
10    线程睡眠1毫秒！  
11    线程睡眠1毫秒！  
12    线程睡眠1毫秒！  
13    线程睡眠1毫秒！  
14    线程睡眠1毫秒！  
15    线程睡眠1毫秒！  
16    线程睡眠1毫秒！  
17    线程睡眠1毫秒！  
18    线程睡眠1毫秒！  
19    线程睡眠1毫秒！  
-------20   
20    线程睡眠1毫秒！  
21    线程睡眠1毫秒！  
22    线程睡眠1毫秒！  
23    线程睡眠1毫秒！  
24    线程睡眠1毫秒！  
25    线程睡眠1毫秒！  
26    线程睡眠1毫秒！  
27    线程睡眠1毫秒！  
28    线程睡眠1毫秒！  
29    线程睡眠1毫秒！  
-------30   
30    线程睡眠1毫秒！  
31    线程睡眠1毫秒！  
32    线程睡眠1毫秒！  
33    线程睡眠1毫秒！  
34    线程睡眠1毫秒！  
35    线程睡眠1毫秒！  
36    线程睡眠1毫秒！  
37    线程睡眠1毫秒！  
38    线程睡眠1毫秒！  
39    线程睡眠1毫秒！  
-------40   
40    线程睡眠1毫秒！  
41    线程睡眠1毫秒！  
42    线程睡眠1毫秒！  
43    线程睡眠1毫秒！  
44    线程睡眠1毫秒！  
45    线程睡眠1毫秒！  
46    线程睡眠1毫秒！  
47    线程睡眠1毫秒！  
48    线程睡眠1毫秒！  
49    线程睡眠1毫秒！  
-------50   
50    线程睡眠1毫秒！  
51    线程睡眠1毫秒！  
52    线程睡眠1毫秒！  
53    线程睡眠1毫秒！  
54    线程睡眠1毫秒！  
55    线程睡眠1毫秒！  
56    线程睡眠1毫秒！  
57    线程睡眠1毫秒！  
58    线程睡眠1毫秒！  
59    线程睡眠1毫秒！  
-------60   
60    线程睡眠1毫秒！  
61    线程睡眠1毫秒！  
62    线程睡眠1毫秒！  
63    线程睡眠1毫秒！  
64    线程睡眠1毫秒！  
65    线程睡眠1毫秒！  
66    线程睡眠1毫秒！  
67    线程睡眠1毫秒！  
68    线程睡眠1毫秒！  
69    线程睡眠1毫秒！  
-------70   
70    线程睡眠1毫秒！  
71    线程睡眠1毫秒！  
72    线程睡眠1毫秒！  
73    线程睡眠1毫秒！  
74    线程睡眠1毫秒！  
75    线程睡眠1毫秒！  
76    线程睡眠1毫秒！  
77    线程睡眠1毫秒！  
78    线程睡眠1毫秒！  
79    线程睡眠1毫秒！  
-------80   
80    线程睡眠1毫秒！  
81    线程睡眠1毫秒！  
82    线程睡眠1毫秒！  
83    线程睡眠1毫秒！  
84    线程睡眠1毫秒！  
85    线程睡眠1毫秒！  
86    线程睡眠1毫秒！  
87    线程睡眠1毫秒！  
88    线程睡眠1毫秒！  
89    线程睡眠1毫秒！  
-------90   
90    线程睡眠1毫秒！  
91    线程睡眠1毫秒！  
92    线程睡眠1毫秒！  
93    线程睡眠1毫秒！  
94    线程睡眠1毫秒！  
95    线程睡眠1毫秒！  
96    线程睡眠1毫秒！  
97    线程睡眠1毫秒！  
98    线程睡眠1毫秒！  
99    线程睡眠1毫秒！  
  
Process finished with exit code 0**

**2、线程的优先级和线程让步yield()**

**线程的让步是通过Thread.yield()来实现的。yield()方法的作用是：暂停当前正在执行的线程对象，并执行其他线程。**

**要理解yield()，必须了解线程的优先级的概念。线程总是存在优先级，优先级范围在1~10之间。JVM线程调度程序是基于优先级的抢先调度机制。在大多数情况下，当前运行的线程优先级将大于或等于线程池中任何线程的优先级。但这仅仅是大多数情况。**

**注意：当设计多线程应用程序的时候，一定不要依赖于线程的优先级。因为线程调度优先级操作是没有保障的，只能把线程优先级作用作为一种提高程序效率的方法，但是要保证程序不依赖这种操作。**

**当线程池中线程都具有相同的优先级，调度程序的JVM实现自由选择它喜欢的线程。这时候调度程序的操作有两种可能：一是选择一个线程运行，直到它阻塞或者运行完成为止。二是时间分片，为池内的每个线程提供均等的运行机会。**

**设置线程的优先级：线程默认的优先级是创建它的执行线程的优先级。可以通过setPriority(int newPriority)更改线程的优先级。例如：**

**Thread t = new MyThread();  
        t.setPriority(8);  
        t.start();**

**线程默认优先级是5，Thread类中有三个常量，定义线程优先级范围：**

**static int MAX\_PRIORITY   
          线程可以具有的最高优先级。  
static int MIN\_PRIORITY   
          线程可以具有的最低优先级。  
static int NORM\_PRIORITY   
          分配给线程的默认优先级。**

**3、Thread.yield()方法**

**Thread.yield()方法作用是：暂停当前正在执行的线程对象，并执行其他线程。**

**yield()应该做的是让当前运行线程回到可运行状态，以允许具有相同优先级的其他线程获得运行机会。因此，使用yield()的目的是让相同优先级的线程之间能适当的轮转执行。但是，实际中无法保证yield()达到让步目的，因为让步的线程还有可能被线程调度程序再次选中。**

**结论：yield()从未导致线程转到等待/睡眠/阻塞状态。在大多数情况下，yield()将导致线程从运行状态转到可运行状态，但有可能没有效果。**

**4、join()方法**

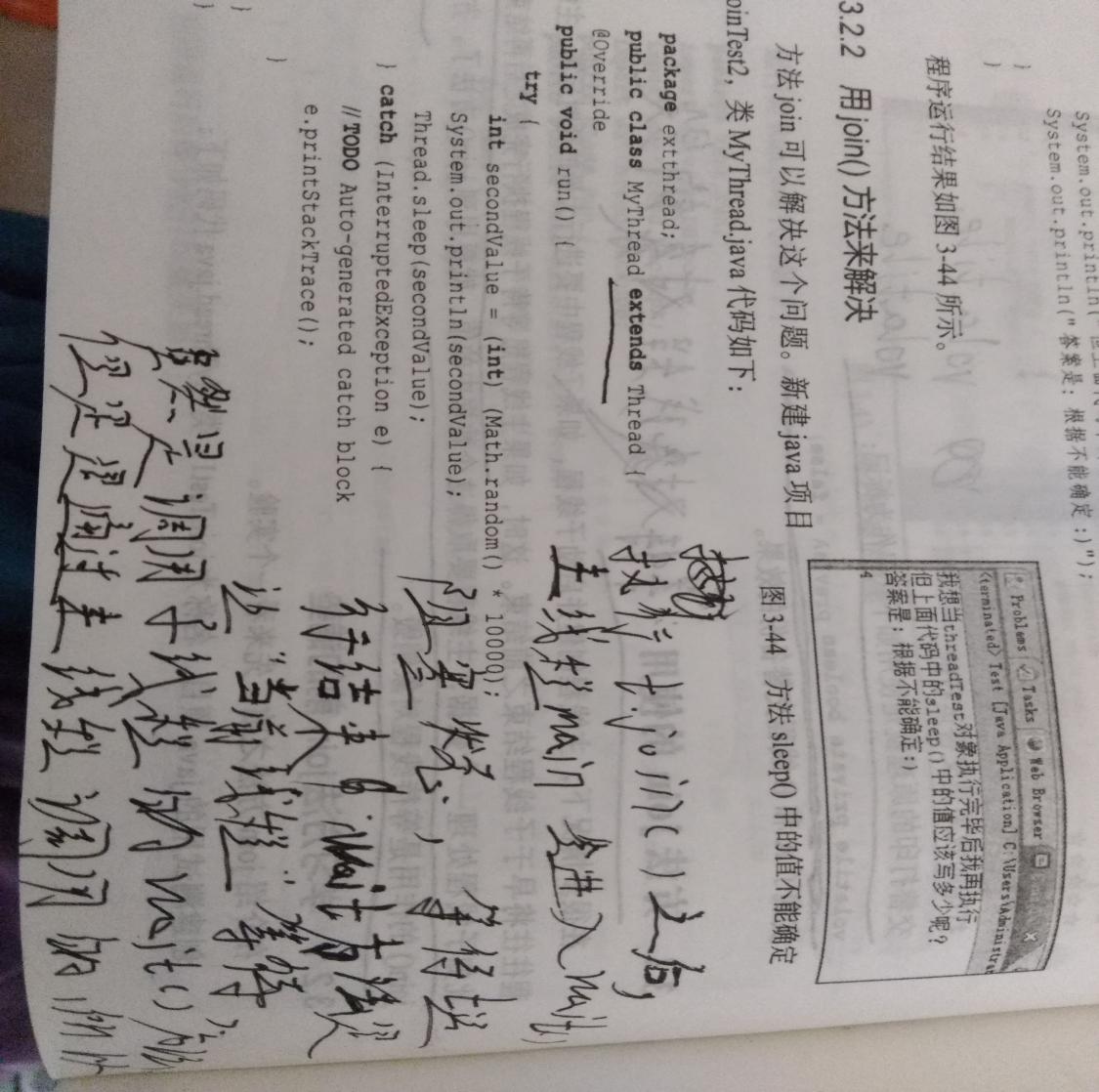
**Thread的非静态方法join()让一个线程B“加入”到另外一个线程A的尾部。在A执行完毕之前，B不能工作。例如：**

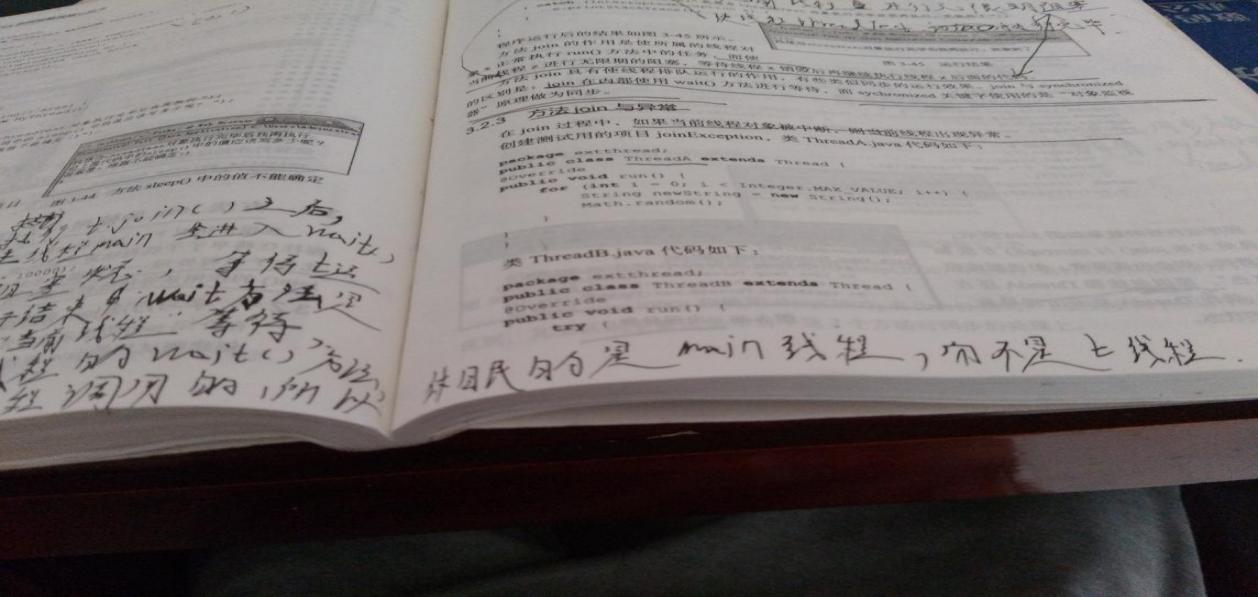
**Thread t = new MyThread();  
        t.start();  
        t.join();**

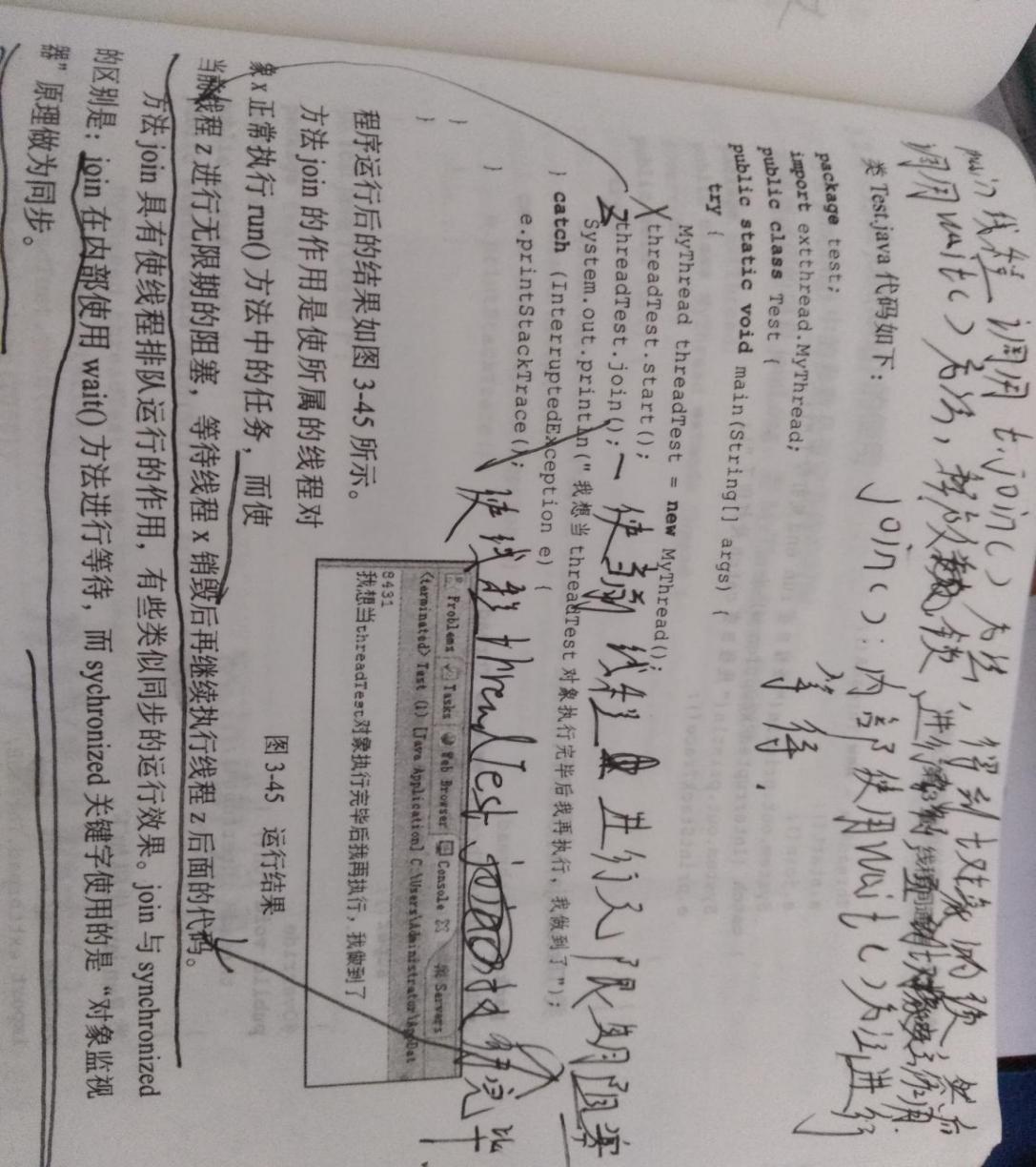
**另外，join()方法还有带超时限制的重载版本。例如t.join(5000);则让线程等待5000毫秒，如果超过这个时间，则停止等待，变为可运行状态。**

**线程的加入join()对线程栈导致的结果是线程栈发生了变化，当然这些变化都是瞬时的。**

**join方法实现原理：**

****

****

****

**小结**

**到目前位置，介绍了线程离开运行状态的3种方法：**

**1、调用Thread.sleep()：使当前线程睡眠至少多少毫秒（尽管它可能在指定的时间之前被中断）。**

**2、调用Thread.yield()：不能保障太多事情，尽管通常它会让当前运行线程回到可运行性状态，使得有相同优先级的线程有机会执行。**

**3、调用join()方法：保证当前线程停止执行，直到该线程所加入的线程完成为止。然而，如果它加入的线程没有存活，则当前线程不需要停止。**

**除了以上三种方式外，还有下面几种特殊情况可能使线程离开运行状态：**

**1、线程的run()方法完成。**

**2、在对象上调用wait()方法（不是在线程上调用）。**

**3、线程不能在对象上获得锁定，它正试图运行该对象的方法代码。**

**4、线程调度程序可以决定将当前运行状态移动到可运行状态，以便让另一个线程获得运行机会，而不需要任何理由。**

**Java线程：线程的同步与锁**

**一、同步问题提出**

**线程的同步是为了防止多个线程访问一个数据对象时，对数据造成的破坏。**

**例如：两个线程ThreadA、ThreadB都操作同一个对象Foo对象，并修改Foo对象上的数据。**

**publicclass Foo {  
    privateint x = 100;  
  
    publicint getX() {  
        return x;  
    }   
  
    publicint fix(int y) {  
        x = x - y;   
        return x;  
    }   
}**

**publicclass MyRunnableimplements Runnable {  
    private Foo foo =new Foo();   
  
    publicstaticvoid main(String[] args) {  
        MyRunnable r = new MyRunnable();  
        Thread ta = new Thread(r,"Thread-A");   
        Thread tb = new Thread(r,"Thread-B");   
        ta.start();   
        tb.start();   
    }   
  
    publicvoid run() {  
        for (int i = 0; i < 3; i++) {  
            this.fix(30);  
            try {  
                Thread.sleep(1);   
            } catch (InterruptedException e) {  
                e.printStackTrace();   
            }   
            System.out.println(Thread.currentThread().getName() + " :当前foo对象的x值= " + foo.getX());  
        }   
    }   
  
    publicint fix(int y) {  
        return foo.fix(y);  
    }   
}**

**运行结果：**

**Thread-A : 当前foo对象的x值= 40  
Thread-B : 当前foo对象的x值= 40  
Thread-B : 当前foo对象的x值= -20  
Thread-A : 当前foo对象的x值= -50  
Thread-A : 当前foo对象的x值= -80  
Thread-B : 当前foo对象的x值= -80  
  
Process finished with exit code 0**

**从结果发现，这样的输出值明显是不合理的。原因是两个线程不加控制的访问Foo对象并修改其数据所致。**

**如果要保持结果的合理性，只需要达到一个目的，就是将对Foo的访问加以限制，每次只能有一个线程在访问。这样就能保证Foo对象中数据的合理性了。**

**在具体的Java代码中需要完成一下两个操作：**

**把竞争访问的资源类Foo变量x标识为private；**

**同步哪些修改变量的代码，使用synchronized关键字同步方法或代码。**

**二、同步和锁定**

**1、锁的原理**

**Java中每个对象都有一个内置锁**

**当****程序运行到非静态的synchronized同步方法上时，自动获得与正在执行代码类的当前实例（this实例）有关的锁。获得一个对象的锁也称为获取锁、锁定对象、在对象上锁定或在对象上同步。**

**当程序运行到synchronized同步方法或代码块时才该对象锁才起作用。**

**一个对象只有一个锁。所以，如果一个线程获得该锁，就没有其他线程可以获得锁，直到第一个线程释放（或返回）锁。这也意味着任何其他线程都不能进入该对象上的synchronized方法或代码块，直到该锁被释放。**

**释放锁是指持锁线程退出了synchronized同步方法或代码块。**

**关于锁和同步，有一下几个要点：**

**1）、只能同步方法，而不能同步变量和类；**

**2）、每个对象只有一个锁；当提到同步时，应该清楚在什么上同步？也就是说，在哪个对象上同步？**

**3）、不必同步类中所有的方法，类可以同时拥有同步和非同步方法。**

**4）、如果两个线程要执行一个类中的synchronized方法，并且两个线程使用相同的实例来调用方法，那么一次只能有一个线程能够执行方法，另一个需要等待，直到锁被释放。也就是说：如果一个线程在对象上获得一个锁，就没有任何其他线程可以进入（该对象的）类中的任何一个同步方法。**

**5）、如果线程拥有同步和非同步方法，则非同步方法可以被多个线程自由访问而不受锁的限制。**

**6）、线程睡眠时，它所持的任何锁都不会释放。**

**7）、线程可以获得多个锁。比如，在一个对象的同步方法里面调用另外一个对象的同步方法，则获取了两个对象的同步锁。**

**8）、同步损害并发性，应该尽可能缩小同步范围。同步不但可以同步整个方法，还可以同步方法中一部分代码块。**

**9）、在使用同步代码块时候，应该指定在哪个对象上同步，也就是说要获取哪个对象的锁。例如：**

**public int fix(int y) {  
        synchronized (this) {  
            x = x - y;  
        }  
        return x;  
    }**

**当然，同步方法也可以改写为非同步方法，但功能完全一样的，例如：**

**public synchronized int getX() {  
        return x++;  
    }**

**与**

**public int getX() {  
        synchronized (this) {  
            return x;  
        }  
    }**

**效果是完全一样的。**

**三、静态方法同步**

**要同步静态方法，需要一个用于整个类对象的锁，这个对象是就是这个类（XXX.class)。**

**例如：**

**public static synchronized int setName(String name){**

**Xxx.name = name;**

**}**

**等价于  
public static int setName(String name){  
      synchronized(Xxx.class){  
            Xxx.name = name;  
      }  
}**

**静态同步方法和非静态同步方法将永远不会彼此阻塞，因为静态方法锁定在Class对象上，非静态方法锁定在该类的对象上。**

**对于同步代码块，要看清楚什么对象已经用于锁定（synchronized后面括号的内容）。在同一个对象上进行同步的线程将彼此阻塞，在不同对象上锁定的线程将永远不会彼此阻塞。**

**五、何时需要同步**

**在多个线程同时访问互斥（可交换）数据时，应该同步以保护数据，确保两个线程不会同时修改更改它。**

**对于非静态字段中可更改的数据，通常使用非静态方法访问。**

**对于静态字段中可更改的数据，通常使用静态方法访问。**

**六、线程安全类**

**当一个类已经很好的同步以保护它的数据时，这个类就称为“线程安全的”。**

**即使是线程安全类，也应该特别小心，因为操作的线程是间仍然不一定安全。**

**举个形象的例子，比如一个集合是线程安全的，有两个线程在操作同一个集合对象，当第一个线程查询集合非空后，删除集合中所有元素的时候。第二个线程也来执行与第一个线程相同的操作，也许在第一个线程查询后，第二个线程也查询出集合非空，但是当第一个执行清除后，第二个再执行删除显然是不对的，因为此时集合已经为空了。**

**看个代码：**

**publicclass NameList {  
    private  List nameList = Collections.synchronizedList(new LinkedList());  
  
    publicvoid add(String name) {  
        nameList.add(name);   
    }   
  
    public String removeFirst() {  
        if (nameList.size() > 0) {  
            return (String) nameList.remove(0);  
        } else {  
            returnnull;  
        }   
    }   
}**

**publicclass Test {  
    publicstaticvoid main(String[] args) {  
        final NameList nl =new NameList();   
        nl.add("aaa");**

**class NameDropper extends Thread{   
            publicvoid run(){  
                String name = nl.removeFirst();   
                System.out.println(name);   
            }   
        }   
  
        Thread t1 = new NameDropper();  
        Thread t2 = new NameDropper();  
        t1.start();   
        t2.start();   
    }   
}**

**虽然集合对象**

**private List nameList = Collections.synchronizedList(new LinkedList());  
是同步的，但是程序还不是线程安全的。**

**出现这种事件的原因是，上例中一个线程操作列表过程中无法阻止另外一个线程对列表的其他操作。**

**解决上面问题的办法是，在操作集合对象的NameList上面做一个同步。改写后的代码如下：**

**publicclass NameList {  
    private List nameList = Collections.synchronizedList(new LinkedList());  
  
    public synchronized void add(String name) {  
        nameList.add(name);   
    }   
  
    public synchronized  String removeFirst() {  
        if (nameList.size() > 0) {  
            return (String) nameList.remove(0);  
        } else {  
            returnnull;  
        }   
    }   
}**

**这样，当一个线程访问其中一个同步方法时，其他线程只有等待。**

**Java线程：线程的交互**

**一、线程交互的基础知识**

**SCJP所要求的线程交互知识点需要从java.lang.Object的类的三个方法来学习：**

**void notify()   
          唤醒在此对象监视器上等待的单个线程。  
 void notifyAll()   
          唤醒在此对象监视器上等待的所有线程。  
 void wait()   
          导致当前的线程等待，直到其他线程调用此对象的 notify()方法或 notifyAll()方法。**

**当然，wait()还有另外两个重载方法：**

**void wait(long timeout)   
          导致当前的线程等待，直到其他线程调用此对象的 notify()方法或 notifyAll()方法，或者超过指定的时间量。   
 void wait(long timeout, int nanos)   
          导致当前的线程等待，直到其他线程调用此对象的 notify()方法或 notifyAll()方法，或者其他某个线程中断当前线程，或者已超过某个实际时间量。**

**以上这些方法是帮助线程传递线程关心的时间状态。**

**关于等待/通知，要记住的关键点是：**

**必须从同步环境内调用wait()、notify()、notifyAll()方法。线程不能调用对象上等待或通知的方法，除非它拥有那个对象的锁。**

**wait()、notify()、notifyAll()都是Object的实例方法。**

**下面看个例子就明白了：**

**/\*\*   
\* 计算输出其他线程锁计算的数据  
\*   
\* @author leizhimin 2008-9-15 13:20:38   
\*/   
publicclass ThreadA {  
    publicstaticvoid main(String[] args) {  
        ThreadB b = new ThreadB();  
        //启动计算线程  
        b.start();   
        //线程A拥有b对象上的锁。线程为了调用wait()或notify()方法，该线程必须是那个对象锁的拥有者  
        synchronized (b) {  
            try {  
                System.out.println("等待对象b完成计算。。。");  
                //当前线程A等待  
                b.wait();   
            } catch (InterruptedException e) {  
                e.printStackTrace();   
            }   
            System.out.println("b对象计算的总和是：" + b.total);  
        }   
    }   
}**

**/\*\*   
\* 计算1+2+3 ... +100的和  
\*   
\* @author leizhimin 2008-9-15 13:20:49   
\*/   
publicclass ThreadBextends Thread {  
    int total;   
  
    publicvoid run() {  
        synchronized (this) {  
            for (int i = 0; i < 101; i++) {  
                total += i;   
            }   
            //（完成计算了）唤醒在此对象监视器上等待的单个线程，在本例中线程A被唤醒  
            notify();   
        }   
    }   
}**

**等待对象b完成计算。。。  
b对象计算的总和是：5050   
  
Process finished with exit code 0**

**千万注意：**

**当在对象上调用wait()方法时，执行该代码的线程立即放弃它在对象上的锁。然而调用notify()时，并不意味着这时线程会放弃其锁。如果线程荣然在完成同步代码，则线程在移出之前不会放弃锁。因此，只要调用notify()并不意味着这时该锁变得可用。**

**二、多个线程在等待一个对象锁时候使用notifyAll()**

**/\*\*   
\* 计算线程   
\*   
\* @author leizhimin 2008-9-20 11:15:46   
\*/   
publicclass Calculator extends Thread {  
        int total;  
        publicvoid run() {  
                synchronized (this) {  
                        for (int i = 0; i < 101; i++) {  
                                total += i;   
                        }   
                }   
                //通知所有在此对象上等待的线程  
                notifyAll();   
        }   
}**

**/\*\*   
\* 获取计算结果并输出   
\*   
\* @author leizhimin 2008-9-20 11:15:22   
\*/   
publicclass ReaderResultextends Thread {  
        Calculator c;   
  
        public ReaderResult(Calculator c) {  
                this.c = c;  
        }   
  
        public void run() {  
                synchronized (c) {  
                        try {  
                                System.out.println(Thread.currentThread() + "等待计算结果。。。");  
                                c.wait();   
                        } catch (InterruptedException e) {  
                                e.printStackTrace();   
                        }   
                        System.out.println(Thread.currentThread() + "计算结果为：" + c.total);  
                }   
        }   
  
        publicstaticvoid main(String[] args) {  
                Calculator calculator = new Calculator();  
  
                //启动三个线程，分别获取计算结果  
                new ReaderResult(calculator).start();  
                new ReaderResult(calculator).start();  
                new ReaderResult(calculator).start();  
                //启动计算线程  
                calculator.start();   
        }   
}**

**运行结果：**

**Thread[Thread-1,5,main]等待计算结果。。。  
Thread[Thread-2,5,main]等待计算结果。。。  
Thread[Thread-3,5,main]等待计算结果。。。  
Exception in thread "Thread-0" java.lang.IllegalMonitorStateException: current thread not owner  
  at java.lang.Object.notifyAll(Native Method)   
  at threadtest.Calculator.run(Calculator.java:18)   
Thread[Thread-1,5,main]计算结果为：5050  
Thread[Thread-2,5,main]计算结果为：5050  
Thread[Thread-3,5,main]计算结果为：5050  
  
Process finished with exit code 0**

**运行结果表明，程序中有异常，并且多次运行结果可能有多种输出结果。这就是说明，这个****多线程的交互程序还存在问题。究竟是出了什么问题，需要深入的分析和思考，下面将做具体分析。**

**实际上，上面这个代码中，我们期望的是读取结果的线程在计算线程调用notifyAll()之前等待即可。但是，如果计算线程先执行，并在读取结果线程等待之前调用了notify()方法，那么又会发生什么呢？这种情况是可能发生的。因为无法保证线程的不同部分将按照什么顺序来执行。幸运的是当读取线程运行时，它只能马上进入等待状态----它没有做任何事情来检查等待的事件是否已经发生。  ----因此，如果计算线程已经调用了notifyAll()方法，那么它就不会再次调用notifyAll()，----并且等待的读取线程将永远保持等待。这当然是开发者所不愿意看到的问题。**

**因此，当等待的事件发生时，需要能够检查notifyAll()通知事件是否已经发生。**

**通常，解决上面问题的最佳方式是将**

**Java线程：线程的调度-休眠**

**Java线程调度是Java多线程的核心，只有良好的调度，才能充分发挥系统的性能，提高程序的执行效率。**

**这里要明确的一点，不管程序员怎么编写调度，只能最大限度的影响线程执行的次序，而不能做到精准控制。**

**线程休眠的目的是使线程让出CPU的最简单的做法之一，线程休眠时候，会将CPU资源交给其他线程，以便能轮换执行，当休眠一定时间后，线程会苏醒，进入准备状态等待执行。**

**线程休眠的方法是Thread.sleep(long millis)和Thread.sleep(long millis, int nanos)，均为静态方法，那调用sleep休眠的哪个线程呢？简单说，哪个线程调用sleep，就休眠哪个线程。**

**/\*\*   
\* Java线程：线程的调度-休眠  
\*   
\* @author leizhimin 2009-11-4 9:02:40   
\*/   
publicclass Test {  
        publicstaticvoid main(String[] args) {  
                Thread t1 = new MyThread1();  
                Thread t2 = new Thread(new MyRunnable());  
                t1.start();   
                t2.start();   
        }   
}   
  
class MyThread1 extends Thread {   
        publicvoid run() {  
                for (int i = 0; i < 3; i++) {  
                        System.out.println("线程1第" + i + "次执行！");  
                        try {  
                                Thread.sleep(50);   
                        } catch (InterruptedException e) {  
                                e.printStackTrace();   
                        }   
                }   
        }   
}   
  
class MyRunnable implements  Runnable {  
        publicvoid run() {  
                for (int i = 0; i < 3; i++) {  
                        System.out.println("线程2第" + i + "次执行！");  
                        try {  
                                Thread.sleep(50);   
                        } catch (InterruptedException e) {  
                                e.printStackTrace();   
                        }   
                }   
        }   
}**

**线程2第0次执行！  
线程1第0次执行！  
线程1第1次执行！  
线程2第1次执行！  
线程1第2次执行！  
线程2第2次执行！  
  
Process finished with exit code 0**

**从上面的结果输出可以看出，无法精准保证线程执行次序。**

**Java线程：线程的调度-优先级**

**与线程休眠类似，线程的优先级仍然无法保障线程的执行次序。只不过，优先级高的线程获取CPU资源的概率较大，优先级低的并非没机会执行。**

**线程的优先级用1-10之间的整数表示，数值越大优先级越高，默认的优先级为5。**

**在一个线程中开启另外一个新线程，则新开线程称为该线程的子线程，子线程初始优先级与父线程相同。**

**/\*\*   
\* Java线程：线程的调度-优先级  
\*   
\* @author leizhimin 2009-11-4 9:02:40   
\*/   
publicclass Test {  
        publicstaticvoid main(String[] args) {  
                Thread t1 = new MyThread1();  
                Thread t2 = new Thread(new MyRunnable());  
                t1.setPriority(10);   
                t2.setPriority(1);   
  
                t2.start();   
                t1.start();   
        }   
}   
  
class MyThread1 extends Thread {   
        publicvoid run() {  
                for (int i = 0; i < 10; i++) {  
                        System.out.println("线程1第" + i + "次执行！");  
                        try {  
                                Thread.sleep(100);   
                        } catch (InterruptedException e) {  
                                e.printStackTrace();   
                        }   
                }   
        }   
}   
  
class MyRunnableimplements Runnable {  
        publicvoid run() {  
                for (int i = 0; i < 10; i++) {  
                        System.out.println("线程2第" + i + "次执行！");  
                        try {  
                                Thread.sleep(100);   
                        } catch (InterruptedException e) {  
                                e.printStackTrace();   
                        }   
                }   
        }   
}**

**线程1第0次执行！  
线程2第0次执行！  
线程2第1次执行！  
线程1第1次执行！  
线程2第2次执行！  
线程1第2次执行！  
线程1第3次执行！  
线程2第3次执行！  
线程2第4次执行！  
线程1第4次执行！  
线程1第5次执行！  
线程2第5次执行！  
线程1第6次执行！  
线程2第6次执行！  
线程1第7次执行！  
线程2第7次执行！  
线程1第8次执行！  
线程2第8次执行！  
线程1第9次执行！  
线程2第9次执行！  
  
Process finished with exit code 0**

**Java线程：线程的调度-让步**

**Thread.yield()：线程的让步含义就是使当前运行着线程让出CPU资源，但是然给谁不知道，仅仅是让出，线程状态回到可运行状态。**

**线程的让步使用Thread.yield()方法，yield()为静态方法，功能是暂停当前正在执行的线程对象，并执行其他线程。**

**/\*\*   
\* Java线程：线程的调度-让步  
\*   
\* @author leizhimin 2009-11-4 9:02:40   
\*/   
publicclass Test {  
        publicstaticvoid main(String[] args) {  
                Thread t1 = new MyThread1();  
                Thread t2 = new Thread(new MyRunnable());  
  
                t2.start();   
                t1.start();   
        }   
}   
  
class MyThread1 extends Thread {   
        publicvoid run() {  
                for (int i = 0; i < 10; i++) {  
                        System.out.println("线程1第" + i + "次执行！");  
                }   
        }   
}   
  
class MyRunnableimplements Runnable {  
        public void run() {  
                for (int i = 0; i < 10; i++) {  
                        System.out.println("线程2第" + i + "次执行！");  
                        Thread.yield();   
                }   
        }   
}**

**线程2第0次执行！  
线程2第1次执行！  
线程2第2次执行！  
线程2第3次执行！  
线程1第0次执行！  
线程1第1次执行！  
线程1第2次执行！  
线程1第3次执行！  
线程1第4次执行！  
线程1第5次执行！  
线程1第6次执行！  
线程1第7次执行！  
线程1第8次执行！  
线程1第9次执行！  
线程2第4次执行！  
线程2第5次执行！  
线程2第6次执行！  
线程2第7次执行！  
线程2第8次执行！  
线程2第9次执行！  
  
Process finished with exit code 0**

**Java线程：线程的调度-合并**

**线程的合并的含义就是将几个并行线程的线程合并为一个单线程执行，应用场景是当一个线程必须等待另一个线程执行完毕才能执行时可以使用join方法。**

**join为非静态方法，定义如下：**

**void join()      
    等待该线程终止。      
void join(long millis)      
    等待该线程终止的时间最长为 millis毫秒。      
void join(long millis,int nanos)      
    等待该线程终止的时间最长为 millis毫秒 + nanos 纳秒。**

**/\*\*   
\* Java线程：线程的调度-合并  
\*   
\* @author leizhimin 2009-11-4 9:02:40   
\*/   
publicclass Test {  
        publicstaticvoid main(String[] args) {  
                Thread t1 = new MyThread1();  
                t1.start();   
  
                for (int i = 0; i < 20; i++) {  
                        System.out.println("主线程第" + i +"次执行！");  
                        if (i > 2)try {   
                                //t1线程合并到主线程中，主线程停止执行过程，转而执行t1线程，直到t1执行完毕后继续。  
                                t1.join();   
                        } catch (InterruptedException e) {  
                                e.printStackTrace();   
                        }   
                }   
        }   
}   
  
class MyThread1 extends Thread {   
        publicvoid run() {  
                for (int i = 0; i < 10; i++) {  
                        System.out.println("线程1第" + i + "次执行！");  
                }   
        }   
}**

**主线程第0次执行！  
主线程第1次执行！  
主线程第2次执行！  
线程1第0次执行！  
主线程第3次执行！  
线程1第1次执行！  
线程1第2次执行！  
线程1第3次执行！  
线程1第4次执行！  
线程1第5次执行！  
线程1第6次执行！  
线程1第7次执行！  
线程1第8次执行！  
线程1第9次执行！  
主线程第4次执行！  
主线程第5次执行！  
主线程第6次执行！  
主线程第7次执行！  
主线程第8次执行！  
主线程第9次执行！  
主线程第10次执行！  
主线程第11次执行！  
主线程第12次执行！  
主线程第13次执行！  
主线程第14次执行！  
主线程第15次执行！  
主线程第16次执行！  
主线程第17次执行！  
主线程第18次执行！  
主线程第19次执行！  
  
Process finished with exit code 0**

**Java线程：线程的调度-守护线程**

**守护线程与普通线程写法上基本么啥区别，调用线程对象的方法setDaemon(true)，则可以将其设置为守护线程。**

**守护线程使用的情况较少，但并非无用，举例来说，JVM的垃圾回收、内存管理等线程都是守护线程。还有就是在做数据库应用时候，使用的数据库连接池，连接池本身也包含着很多后台线程，监控连接个数、超时时间、状态等等。**

**setDaemon方法的详细说明：**

**public final void setDaemon(boolean on)将该线程标记为守护线程或用户线程。当正在运行的线程都是守护线程时，Java虚拟机退出。      
  该方法必须在启动线程前调用。      
  
  该方法首先调用该线程的 checkAccess方法，且不带任何参数。这可能抛出 SecurityException（在当前线程中）。      
  
  
  参数：  
    on - 如果为true，则将该线程标记为守护线程。      
  抛出：      
    IllegalThreadStateException - 如果该线程处于活动状态。      
    SecurityException - 如果当前线程无法修改该线程。  
  另请参见：  
    isDaemon(), checkAccess()**

**/\*\*   
\* Java线程：线程的调度-守护线程  
\*   
\* @author leizhimin 2009-11-4 9:02:40   
\*/   
publicclass Test {  
        publicstaticvoid main(String[] args) {  
                Thread t1 = new MyCommon();  
                Thread t2 = new Thread(new MyDaemon());  
                t2.setDaemon(true);        //设置为守护线程  
  
                t2.start();   
                t1.start();   
        }   
}   
  
class MyCommon extends Thread {   
        public void run() {  
                for (int i = 0; i < 5; i++) {  
                        System.out.println("线程1第" + i + "次执行！");  
                        try {  
                                Thread.sleep(7);   
                        } catch (InterruptedException e) {  
                                e.printStackTrace();   
                        }   
                }   
        }   
}   
  
class MyDaemon implements Runnable {   
        publicvoid run() {  
                for (long i = 0; i < 9999999L; i++) {  
                        System.out.println("后台线程第" + i +"次执行！");  
                        try {  
                                Thread.sleep(7);   
                        } catch (InterruptedException e) {  
                                e.printStackTrace();   
                        }   
                }   
        }   
}**

**后台线程第0次执行！  
线程1第0次执行！  
线程1第1次执行！  
后台线程第1次执行！  
后台线程第2次执行！  
线程1第2次执行！  
线程1第3次执行！  
后台线程第3次执行！  
线程1第4次执行！  
后台线程第4次执行！  
后台线程第5次执行！  
后台线程第6次执行！  
后台线程第7次执行！  
  
Process finished with exit code 0**

**从上面的执行结果可以看出：**

**前台线程是保证执行完毕的，后台线程还没有执行完毕就退出了。**

**实际上：JRE判断程序是否执行结束的标准是所有的前台执线程行完毕了，而不管后台线程的状态，因此，在使用后台线程时候一定要注意这个问题。**

**Java线程：线程的同步-同步方法**

**在本文之前，请参阅《[Java线程：线程的同步与锁](http://lavasoft.blog.51cto.com/62575/99155" \t "_blank)》，本文是在此基础上所写的。**

**对于同步，在具体的Java代码中需要完成一下两个操作：**

**把竞争访问的资源标识为private；**

**同步哪些修改变量的代码，使用synchronized关键字同步方法或代码。**

***当然这不是唯一控制并发安全的途径。***

**。**

**为了演示同步方法的使用，构建了一个信用卡账户，起初信用额为100w，然后模拟透支、存款等多个操作。显然银行账户User对象是个竞争资源，而多个并发操作的是账户方法oper(int x)，当然应该在此方法上加上同步，并将账户的余额设为私有变量，禁止直接访问。**

**/\*\*   
\* Java线程：线程的同步   
\*   
\* @author leizhimin 2009-11-4 11:23:32   
\*/   
publicclass Test {  
        publicstaticvoid main(String[] args) {  
                User u = new User("张三", 100);  
                MyThread t1 = new MyThread("线程A", u, 20);  
                MyThread t2 = new MyThread("线程B", u, -60);  
                MyThread t3 = new MyThread("线程C", u, -80);  
                MyThread t4 = new MyThread("线程D", u, -30);  
                MyThread t5 = new MyThread("线程E", u, 32);  
                MyThread t6 = new MyThread("线程F", u, 21);  
  
                t1.start();   
                t2.start();   
                t3.start();   
                t4.start();   
                t5.start();   
                t6.start();   
        }   
}   
  
class MyThread extends Thread {   
        private User u;  
        private int y = 0;  
  
        MyThread(String name, User u, int y) {  
                super(name);  
                this.u = u;  
                this.y = y;  
        }   
  
        publicvoid run() {  
                u.oper(y);   
        }   
}   
  
class User {   
        private String code;  
        private int cash;  
  
        User(String code, int cash) {  
                this.code = code;  
                this.cash = cash;  
        }   
  
        public String getCode() {  
                return code;  
        }   
  
        publicvoid setCode(String code) {  
                this.code = code;  
        }   
  
        /\*\*   
         \* 业务方法  
         \* @param x 添加x万元  
         \*/   
        public synchronized void oper(int x) {  
                try {  
                        Thread.sleep(10L);   
                        this.cash += x;  
                        System.out.println(Thread.currentThread().getName() + "运行结束，增加“" + x +"”，当前用户账户余额为：" + cash);  
                        Thread.sleep(10L);   
                } catch (InterruptedException e) {  
                        e.printStackTrace();   
                }   
        }   
  
        @Override   
        public String toString() {  
                return"User{" +  
                                "code='" + code + '\'' +  
                                ", cash=" + cash +  
                                '}';   
        }   
}**

**输出结果：**

**线程A运行结束，增加“20”，当前用户账户余额为：120  
线程F运行结束，增加“21”，当前用户账户余额为：141  
线程E运行结束，增加“32”，当前用户账户余额为：173  
线程C运行结束，增加“-80”，当前用户账户余额为：93  
线程B运行结束，增加“-60”，当前用户账户余额为：33  
线程D运行结束，增加“-30”，当前用户账户余额为：3  
  
Process finished with exit code 0**

**反面教材，不同步的情况，也就是去掉oper(int x)方法的synchronized修饰符，然后运行程序，结果如下：**

**线程A运行结束，增加“20”，当前用户账户余额为：61  
线程D运行结束，增加“-30”，当前用户账户余额为：63  
线程B运行结束，增加“-60”，当前用户账户余额为：3  
线程F运行结束，增加“21”，当前用户账户余额为：61  
线程E运行结束，增加“32”，当前用户账户余额为：93  
线程C运行结束，增加“-80”，当前用户账户余额为：61  
  
Process finished with exit code 0**

**很显然，上面的结果是错误的，导致错误的原因是多个线程并发访问了竞争资源u，并对u的属性做了改动。**

**可见同步的重要性。**

**注意：**

**通过前文可知，线程退出同步方法时将释放掉方法所属对象的锁，但还应该注意的是，同步方法中还可以使用特定的方法对线程进行调度。这些方法来自于java.lang.Object类。**

**void notify()      
                    唤醒在此对象监视器上等待的单个线程。      
void notifyAll()      
                    唤醒在此对象监视器上等待的所有线程。      
void wait()      
                    导致当前的线程等待，直到其他线程调用此对象的 notify()方法或 notifyAll()方法。**

**void wait(long timeout)      
                    导致当前的线程等待，直到其他线程调用此对象的 notify()方法或 notifyAll()方法，或者超过指定的时间量。      
void wait(long timeout,int nanos)      
                    导致当前的线程等待，直到其他线程调用此对象的 notify()方法或 notifyAll()方法，或者其他某个线程中断当前线程，或者已超过某个实际时间量。**

**结合以上方法，处理多线程同步与互斥问题非常重要，著名的生产者-消费者例子就是一个经典的例子，任何语言多线程必学的例子。**

**Java线程：线程的同步-同步块**

**对于同步，除了同步方法外，还可以使用同步代码块，有时候同步代码块会带来比同步方法更好的效果。**

**注意：**

**在使用synchronized关键字时候，应该尽可能避免在synchronized方法或synchronized块中使用sleep或者yield方法，因为synchronized程序块占有着对象锁，你休息那么其他的线程只能一边等着你醒来执行完了才能执行。不但严重影响效率，也不合逻辑。**

**同样，在同步程序块内调用yeild方法让出CPU资源也没有意义，因为你占用着锁，其他互斥线程还是无法访问同步程序块。当然与同步程序块无关的线程可以获得更多的执行时间。**

**sleep（）和yeild（）方法占用锁，其他互斥线程无法访问同步代码块，但是与同步代码块无关的线程不影响。**

**Java线程：并发协作-死锁**

**线程发生死锁可能性很小，即使看似可能发生死锁的代码，在运行时发生死锁的可能性也是小之又小。**

**发生死锁的原因一般是两个对象的锁相互等待造成的。**

**在《Java线程：线程的同步与锁》一文中，简述死锁的概念与简单例子，但是所给的例子是不完整的，这里给出一个完整的例子。**

**/\*\*   
\* Java线程：并发协作-死锁  
\*   
\* @author Administrator 2009-11-4 22:06:13   
\*/   
public class Test {  
        public static void main(String[] args) {  
                DeadlockRisk dead = new  DeadlockRisk();  
                MyThread t1 = new MyThread(dead, 1, 2);  
                MyThread t2 = new MyThread(dead, 3, 4);  
                MyThread t3 = new MyThread(dead, 5, 6);  
                MyThread t4 = new MyThread(dead, 7, 8);  
  
                t1.start();   
                t2.start();   
                t3.start();   
                t4.start();   
        }   
  
}   
  
class MyThread extends Thread {   
        private DeadlockRisk dead;  
        private int  a, b;  
  
  
        MyThread(DeadlockRisk dead, int a,int b) {   
                this.dead = dead;  
                this.a = a;  
                this.b = b;  
        }   
  
        @Override   
        public void run() {  
                dead.read();   
                dead.write(a, b);   
        }   
}**

**class DeadlockRisk {  
        private static class Resource {  
                public int value;  
        }   
  
        private Resource resourceA =new Resource();   
        private Resource resourceB =new Resource();   
  
        public int read() {  
                synchronized (resourceA) {  
                        System.out.println("read():" + Thread.currentThread().getName() +"获取了resourceA的锁！");  
                        synchronized (resourceB) {  
                                System.out.println("read():" + Thread.currentThread().getName() +"获取了resourceB的锁！");  
                                return resourceB.value + resourceA.value;  
                        }   
                }   
        }   
  
        public void write(int a,int b) {   
                synchronized (resourceB) {  
                        System.out.println("write():" + Thread.currentThread().getName() +"获取了resourceA的锁！");  
                        synchronized (resourceA) {  
                                System.out.println("write():" + Thread.currentThread().getName() +"获取了resourceB的锁！");  
                                resourceA.value = a;   
                                resourceB.value = b;   
                        }   
                }   
        }   
}**

**下面死锁的情况发生了，真是难得一见啊：**

**Java线程：新特征-线程池**

**Sun在Java5中，对Java线程的类库做了大量的扩展，其中线程池就是Java5的新特征之一，除了线程池之外，还有很多多线程相关的内容，为多线程的编程带来了极大便利。为了编写高效稳定可靠的多线程程序，线程部分的新增内容显得尤为重要。**

**有关Java5线程新特征的内容全部在java.util.concurrent下面，里面包含数目众多的接口和类，熟悉这部分API特征是一项艰难的学习过程。目前有关这方面的资料和书籍都少之又少，大所属介绍线程方面书籍还停留在java5之前的知识层面上。**

**当然新特征对做多线程程序没有必须的关系，在java5之前通用可以写出很优秀的多线程程序。只是代价不一样而已。**

**线程池的基本思想还是一种对象池的思想，开辟一块内存空间，里面存放了众多（未死亡）的线程，池中线程执行调度由池管理器来处理。当有线程任务时，从池中取一个，执行完成后线程对象归池，这样可以避免反复创建线程对象所带来的性能开销，节省了系统的资源。**

**在Java5之前，要实现一个线程池是相当有难度的，现在Java5为我们做好了一切，我们只需要按照提供的API来使用，即可享受线程池带来的极大便利。**

**Java5的线程池分好多种：固定尺寸的线程池、可变尺寸连接池、。**

**在使用线程池之前，必须知道如何去创建一个线程池，在Java5中，需要了解的是java.util.concurrent.Executors类的API，这个类提供大量创建连接池的静态方法，是必须掌握的。**

**一、固定大小的线程池**

**import java.util.concurrent.Executors;  
import java.util.concurrent.ExecutorService;  
  
/\*\*   
\* Java线程：线程池-   
\*   
\* @author Administrator 2009-11-4 23:30:44   
\*/   
publicclass Test {  
        publicstaticvoid main(String[] args) {  
                //创建一个可重用固定线程数的线程池  
                ExecutorService pool = Executors.newFixedThreadPool(2);   
                //创建实现了Runnable接口对象，Thread对象当然也实现了Runnable接口  
                Thread t1 = new MyThread();  
                Thread t2 = new MyThread();  
                Thread t3 = new MyThread();  
                Thread t4 = new MyThread();  
                Thread t5 = new MyThread();  
                //将线程放入池中进行执行  
                pool.execute(t1);   
                pool.execute(t2);   
                pool.execute(t3);   
                pool.execute(t4);   
                pool.execute(t5);   
                //关闭线程池  
                pool.shutdown();   
        }   
}   
  
class MyThread extends Thread{   
        @Override   
        publicvoid run() {  
                System.out.println(Thread.currentThread().getName()+"正在执行。。。");  
        }   
}**

**pool-1-thread-1正在执行。。。  
pool-1-thread-1正在执行。。。  
pool-1-thread-1正在执行。。。  
pool-1-thread-1正在执行。。。  
pool-1-thread-2正在执行。。。  
  
Process finished with exit code 0**

**二、单任务线程池**

**在上例的基础上改一行创建pool对象的代码为：**

**//创建一个使用单个 worker线程的 Executor，以无界队列方式来运行该线程。  
                ExecutorService pool = Executors.newSingleThreadExecutor();**

**输出结果为：**

**pool-1-thread-1正在执行。。。  
pool-1-thread-1正在执行。。。  
pool-1-thread-1正在执行。。。  
pool-1-thread-1正在执行。。。  
pool-1-thread-1正在执行。。。  
  
Process finished with exit code 0**

**对于以上两种连接池，大小都是固定的，当要加入的池的线程（或者任务）超过池最大尺寸时候，则入此线程池需要排队等待。**

**一旦池中有线程完毕，则排队等待的某个线程会入池执行。**

**三、可变尺寸的线程池**

**与上面的类似，只是改动下pool的创建方式：**

**//创建一个可根据需要创建新线程的线程池，但是在以前构造的线程可用时将重用它们。  
                ExecutorService pool = Executors.newCachedThreadPool();**

**pool-1-thread-5正在执行。。。  
pool-1-thread-1正在执行。。。  
pool-1-thread-4正在执行。。。  
pool-1-thread-3正在执行。。。  
pool-1-thread-2正在执行。。。  
  
Process finished with exit code 0**

**四、延迟连接池**

**import java.util.concurrent.Executors;  
import java.util.concurrent.ScheduledExecutorService;  
import java.util.concurrent.TimeUnit;  
  
/\*\*   
\* Java线程：线程池-   
\*   
\* @author Administrator 2009-11-4 23:30:44   
\*/   
publicclass Test {  
        publicstaticvoid main(String[] args) {  
                //创建一个线程池，它可安排在给定延迟后运行命令或者定期地执行  
                ScheduledExecutorService pool = Executors.newScheduledThreadPool(2);  
                //创建实现了Runnable接口对象，Thread对象当然也实现了Runnable接口  
                Thread t1 = new MyThread();  
                Thread t2 = new MyThread();  
                Thread t3 = new MyThread();  
                Thread t4 = new MyThread();  
                Thread t5 = new MyThread();  
                //将线程放入池中进行执行  
                pool.execute(t1);   
                pool.execute(t2);   
                pool.execute(t3);   
                //使用延迟执行风格的方法  
                pool.schedule(t4, 10, TimeUnit.MILLISECONDS);   
                pool.schedule(t5, 10, TimeUnit.MILLISECONDS);   
                //关闭线程池  
                pool.shutdown();   
        }   
}   
  
class MyThread extends Thread {   
        @Override   
        publicvoid run() {  
                System.out.println(Thread.currentThread().getName() + "正在执行。。。");  
        }   
}**

**pool-1-thread-1正在执行。。。  
pool-1-thread-2正在执行。。。  
pool-1-thread-1正在执行。。。  
pool-1-thread-1正在执行。。。  
pool-1-thread-2正在执行。。。  
  
Process finished with exit code 0**

**五、单任务延迟连接池**

**在四代码基础上，做改动**

**//创建一个单线程执行程序，它可安排在给定延迟后运行命令或者定期地执行。  
                ScheduledExecutorService pool = Executors.newSingleThreadScheduledExecutor();**

**pool-1-thread-1正在执行。。。  
pool-1-thread-1正在执行。。。  
pool-1-thread-1正在执行。。。  
pool-1-thread-1正在执行。。。  
pool-1-thread-1正在执行。。。  
  
Process finished with exit code 0**

**六、自定义线程池**

**import java.util.concurrent.ArrayBlockingQueue;  
import java.util.concurrent.BlockingQueue;  
import java.util.concurrent.ThreadPoolExecutor;  
import java.util.concurrent.TimeUnit;  
  
/\*\*   
\* Java线程：线程池-自定义线程池  
\*   
\* @author Administrator 2009-11-4 23:30:44   
\*/   
public class Test {  
        publicstaticvoid main(String[] args) {  
                //创建等待队列  
                BlockingQueue<Runnable> bqueue = new ArrayBlockingQueue<Runnable>(20);**

**//创建一个单线程执行程序，它可安排在给定延迟后运行命令或者定期地执行。  
                ThreadPoolExecutor pool = newThreadPoolExecutor(2,3,2,TimeUnit.MILLISECONDS,bqueue);  
                //创建实现了Runnable接口对象，Thread对象当然也实现了Runnable接口  
                Thread t1 = new MyThread();  
                Thread t2 = new MyThread();  
                Thread t3 = new MyThread();  
                Thread t4 = new MyThread();  
                Thread t5 = new MyThread();  
                Thread t6 = new MyThread();  
                Thread t7 = new MyThread();  
                //将线程放入池中进行执行  
                pool.execute(t1);   
                pool.execute(t2);   
                pool.execute(t3);   
                pool.execute(t4);   
                pool.execute(t5);   
                pool.execute(t6);   
                pool.execute(t7);   
                //关闭线程池  
                pool.shutdown();   
        }   
}   
  
class MyThread extends Thread {   
        @Override   
        publicvoid run() {  
                System.out.println(Thread.currentThread().getName() + "正在执行。。。");  
                try {  
                        Thread.sleep(100L);   
                } catch (InterruptedException e) {  
                        e.printStackTrace();   
                }   
        }   
}**

**pool-1-thread-1正在执行。。。  
pool-1-thread-2正在执行。。。  
pool-1-thread-2正在执行。。。  
pool-1-thread-1正在执行。。。  
pool-1-thread-2正在执行。。。  
pool-1-thread-1正在执行。。。  
pool-1-thread-2正在执行。。。  
  
Process finished with exit code 0**

**创建自定义线程池的构造方法很多，本例中参数的含义如下：**

**ThreadPoolExecutor**

**public ThreadPoolExecutor(int corePoolSize,**

**int maximumPoolSize,**

**long keepAliveTime,**

**[TimeUnit](http://www.cnblogs.com/riskyer/admin/" \t "_blank" \o "java.util.concurrent) unit,**

**[BlockingQueue](http://www.cnblogs.com/riskyer/admin/" \t "_blank" \o "java.util.concurrent)<[Runnable](http://www.cnblogs.com/riskyer/admin/" \t "_blank" \o "java.lang)> workQueue)**

**用给定的初始参数和默认的线程工厂及处理程序创建新的ThreadPoolExecutor。使用[Executors](http://www.cnblogs.com/riskyer/admin/" \t "_blank" \o "java.util.concurrent)工厂方法之一比使用此通用构造方法方便得多。**

**参数：**

**corePoolSize -池中所保存的线程数，包括空闲线程。**

**maximumPoolSize -池中允许的最大线程数。**

**keepAliveTime -当线程数大于核心时，此为终止前多余的空闲线程等待新任务的最长时间。**

**unit - keepAliveTime参数的时间单位。**

**workQueue -执行前用于保持任务的队列。此队列仅保持由execute方法提交的Runnable任务。**

**抛出：**

**[IllegalArgumentException](http://www.cnblogs.com/riskyer/admin/" \t "_blank" \o "java.lang) -如果 corePoolSize或 keepAliveTime小于零，或者 maximumPoolSize小于或等于零，或者 corePoolSize大于 maximumPoolSize。**

**[NullPointerException](http://www.cnblogs.com/riskyer/admin/" \t "_blank" \o "java.lang) -如果workQueue为 null**

**自定义连接池稍微麻烦些，不过通过创建的ThreadPoolExecutor线程池对象，可以获取到当前线程池的尺寸、正在执行任务的线程数、工作队列等等。**

**有关Java5线程池的内容到此就没有了，更多的内容还需要研读API来获取。**

**Java线程：新特征-有返回值的线程**

**在Java5之前，线程是没有返回值的，常常为了“有”返回值，破费周折，而且代码很不好写。或者干脆绕过这道坎，走别的路了。**

**现在Java终于有可返回值的任务（也可以叫做线程）了。**

**可返回值的任务必须实现Callable接口，类似的，无返回值的任务必须Runnable接口。**

**执行Callable任务后，可以获取一个Future的对象，在该对象上调用get就可以获取到Callable任务返回的Object了。**

**下面是个很简单的例子：**

**import java.util.concurrent.\*;  
  
/\*\*   
\* Java线程：有返回值的线程  
\*   
\* @author Administrator 2009-11-5 0:41:50   
\*/   
publicclass Test {  
        publicstaticvoid main(String[] args)throws ExecutionException, InterruptedException {  
                //创建一个线程池  
                ExecutorService pool = Executors.newFixedThreadPool(2);   
                //创建两个有返回值的任务  
                Callable c1 = new MyCallable("A");  
                Callable c2 = new MyCallable("B");  
                //执行任务并获取Future对象  
                Future f1 = pool.submit(c1);   
                Future f2 = pool.submit(c2);   
                //从Future对象上获取任务的返回值，并输出到控制台  
                System.out.println(">>>"+f1.get().toString());  
                System.out.println(">>>"+f2.get().toString());  
                //关闭线程池  
                pool.shutdown();   
        }   
}   
  
class MyCallableimplements Callable{  
        private String oid;  
  
        MyCallable(String oid) {   
                this.oid = oid;  
        }   
  
        @Override   
        public Object call()throws Exception {  
                return oid+"任务返回的内容";  
        }   
}**

**>>>A任务返回的内容  
>>>B任务返回的内容   
  
Process finished with exit code 0**

**非常的简单，要深入了解还需要看Callable和Future接口的API啊。**

**Java线程：新特征-锁（上）**

在Java5中，专门提供了锁对象，**利用锁可以方便的实现资源的封锁，用来控制对竞争资源并发访问的控制**，这些内容主要集中在**java.util.concurrent.locks**包下面，里面有三个重要的接口**Condition、Lock、ReadWriteLock。**

|  |  |
| --- | --- |
| [Condition](http://blog.csdn.net/chbsrzf/article/details/48013681" \o "java.util.concurrent.locks" \t "http://blog.csdn.net/chbsrzf/article/details/_blank) | Condition 将 Object 监视器方法（[wait](http://blog.csdn.net/chbsrzf/article/details/48013681" \t "http://blog.csdn.net/chbsrzf/article/details/_blank)、[notify](http://blog.csdn.net/chbsrzf/article/details/48013681" \t "http://blog.csdn.net/chbsrzf/article/details/_blank) 和 [notifyAll](http://blog.csdn.net/chbsrzf/article/details/48013681" \t "http://blog.csdn.net/chbsrzf/article/details/_blank)）分解成截然不同的对象，以便通过将这些对象与任意 [Lock](http://blog.csdn.net/chbsrzf/article/details/48013681" \o "java.util.concurrent.locks" \t "http://blog.csdn.net/chbsrzf/article/details/_blank) 实现组合使用，为每个对象提供多个等待 set （wait-set）。 |
| [Lock](http://blog.csdn.net/chbsrzf/article/details/48013681" \o "java.util.concurrent.locks" \t "http://blog.csdn.net/chbsrzf/article/details/_blank) | Lock 实现提供了比使用 synchronized 方法和语句可获得的更广泛的锁定操作。 |
| [ReadWriteLock](http://blog.csdn.net/chbsrzf/article/details/48013681" \o "java.util.concurrent.locks" \t "http://blog.csdn.net/chbsrzf/article/details/_blank) | ReadWriteLock 维护了一对相关的[锁定](http://blog.csdn.net/chbsrzf/article/details/48013681" \o "java.util.concurrent.locks" \t "http://blog.csdn.net/chbsrzf/article/details/_blank)，一个用于只读操作，另一个用于写入操作。 |

有关锁的介绍,API文档解说很多，看得很烦，还是看个例子再看文档比较容易理解。

import java.util.concurrent.ExecutorService;   
import java.util.concurrent.Executors;   
import java.util.concurrent.locks.Lock;   
import java.util.concurrent.locks.ReentrantLock;   
  
/\*\*   
\* Java线程：锁   
\*   
\* @author chb  
\*/   
public class Test {   
        public static void main(String[] args) {   
                //创建并发访问的账户   
                MyCount myCount = new MyCount("95599200901215522", 10000);   
                //创建一个锁对象   
                Lock lock = new ReentrantLock();   
                //创建一个线程池   
                ExecutorService pool = Executors.newCachedThreadPool();   
                //创建一些并发访问用户，一个信用卡，存的存，取的取，好热闹啊   
                User u1 = new User("张三", myCount, -4000, lock);   
                User u2 = new User("张三他爹", myCount, 6000, lock);   
                User u3 = new User("张三他弟", myCount, -8000, lock);   
                User u4 = new User("张三", myCount, 800, lock);   
                //在线程池中执行各个用户的操作   
                pool.execute(u1);   
                pool.execute(u2);   
                pool.execute(u3);   
                pool.execute(u4);   
                //关闭线程池   
                pool.shutdown();   
        }   
}   
  
/\*\*   
\* 信用卡的用户   
\*/   
class User implements Runnable {   
        private String name;                //用户名   
        private MyCount myCount;        //所要操作的账户   
        private int iocash;                 //操作的金额，当然有正负之分了   
        private Lock myLock;                //执行操作所需的锁对象   
  
        User(String name, MyCount myCount, int iocash, Lock myLock) {   
                this.name = name;   
                this.myCount = myCount;   
                this.iocash = iocash;   
                this.myLock = myLock;   
        }   
  
        public void run() {   
                //获取锁   
                myLock.lock();   
                //执行现金业务   
                System.out.println(name + "正在操作" + myCount + "账户，金额为" + iocash + "，当前金额为"+ myCount.getCash());   
                myCount.setCash(myCount.getCash() + iocash);

                System.out.println(name + "操作" + myCount + "账户成功，金额为" + iocash + "，当前金额为"+ myCount.getCash());   
                //释放锁，否则别的线程没有机会执行了   
                myLock.unlock();   
        }   
}   
  
/\*\*   
\* 信用卡账户，可随意透支   
\*/   
class MyCount {   
        private String oid;         //账号   
        private int cash;             //账户余额   
  
        MyCount(String oid, int cash) {   
                this.oid = oid;   
                this.cash = cash;   
        }   
  
        public String getOid() {   
                return oid;   
        }   
  
        public void setOid(String oid) {   
                this.oid = oid;   
        }   
  
        public int getCash() {   
                return cash;   
        }   
  
        public void setCash(int cash) {   
                this.cash = cash;   
        }   
  
        @Override   
        public String toString() {   
                return "MyCount{" +   
                                "oid='" + oid + '\'' +   
                                ", cash=" + cash +   
                                '}';   
        }   
}

张三正在操作MyCount{oid='95599200901215522', cash=10000}账户，金额为-4000，当前金额为10000

张三操作MyCount{oid='95599200901215522', cash=6000}账户成功，金额为-4000，当前金额为6000   
张三他爹正在操作MyCount{oid='95599200901215522', cash=6000}账户，金额为6000，当前金额为6000   
张三他爹操作MyCount{oid='95599200901215522', cash=12000}账户成功，金额为6000，当前金额为12000   
张三他弟正在操作MyCount{oid='95599200901215522', cash=12000}账户，金额为-8000，当前金额为12000   
张三他弟操作MyCount{oid='95599200901215522', cash=4000}账户成功，金额为-8000，当前金额为4000   
张三正在操作MyCount{oid='95599200901215522', cash=4000}账户，金额为800，当前金额为4000   
张三操作MyCount{oid='95599200901215522', cash=4800}账户成功，金额为800，当前金额为4800   
  
Process finished with exit code 0

从上面的输出可以看到，利用锁对象太方便了，比直接在某个不知情的对象上用锁清晰多了。

但一定要注意的是，在获取了锁对象后，**用完后应该尽快释放锁，以便别的等待该锁的线程有机会去执行**。

**Java线程：新特征-锁（下）**

在上文中提到了Lock接口以及对象，使用它，很优雅的控制了竞争资源的安全访问，但是这种锁不区分读写，称这种锁为普通锁。为了提高性能，Java提供了**读写锁**，**在读的地方使用读锁，在写的地方使用写锁**，灵活控制，在一定程度上提高了程序的执行效率。

Java中读写锁有个接口**java.util.concurrent.locks.ReadWriteLock**，也有具体的实现ReentrantReadWriteLock，详细的API可以查看JavaAPI文档。

下面这个例子是在文例子的基础上，将普通锁改为读写锁，并添加账户余额查询的功能，代码如下：

import java.util.concurrent.ExecutorService;   
import java.util.concurrent.Executors;   
import java.util.concurrent.locks.ReadWriteLock;   
import java.util.concurrent.locks.ReentrantReadWriteLock;   
  
/\*\*   
\* Java线程：读写锁 ：读写互斥，写写互斥，读读多线程  
\*   
\* @author chb 2009-11-5 10:57:29   
\*/   
public class Test {   
        public static void main(String[] args) {   
                //创建并发访问的账户   
                MyCount myCount = new MyCount("95599200901215522", 10000);   
                //创建一个锁对象   
                ReadWriteLock lock = new ReentrantReadWriteLock(false);   
                //创建一个线程池   
                ExecutorService pool = Executors.newFixedThreadPool(2);   
                //创建一些并发访问用户，一个信用卡，存的存，取的取，好热闹啊   
                User u1 = new User("张三", myCount, -4000, lock, false);   
                User u2 = new User("张三他爹", myCount, 6000, lock, false);   
                User u3 = new User("张三他弟", myCount, -8000, lock, false);   
                User u4 = new User("张三", myCount, 800, lock, false);   
                User u5 = new User("张三他爹", myCount, 0, lock, true);   
                //在线程池中执行各个用户的操作   
                pool.execute(u1);   
                pool.execute(u2);   
                pool.execute(u3);   
                pool.execute(u4);   
                pool.execute(u5);   
                //关闭线程池   
                pool.shutdown();   
        }   
}   
  
/\*\*   
\* 信用卡的用户   
\*/   
class User implements Runnable {   
        private String name;                //用户名   
        private MyCount myCount;        //所要操作的账户   
        private int iocash;                 //操作的金额，当然有正负之分了   
        private ReadWriteLock myLock;                //执行操作所需的锁对象   
        private boolean ischeck;        //是否查询   
  
        User(String name, MyCount myCount, int iocash, ReadWriteLock myLock, boolean ischeck) {   
                this.name = name;   
                this.myCount = myCount;   
                this.iocash = iocash;   
                this.myLock = myLock;   
                this.ischeck = ischeck;   
        }   
  
        public void run() {   
                if (ischeck) {   
                        //获取读锁   
                        myLock.readLock().lock();   
                        System.out.println("读：" + name + "正在查询" + myCount + "账户，当前金额为" + myCount.getCash());   
                        //释放读锁   
                        myLock.readLock().unlock();   
                } else {   
                        //获取写锁   
                        myLock.writeLock().lock();   
                        //执行现金业务   
                        System.out.println("写：" + name + "正在操作" + myCount + "账户，金额为" + iocash +"，当前金额为" + myCount.getCash());   
                        myCount.setCash(myCount.getCash() + iocash);   
                        System.out.println("写：" + name + "操作" + myCount + "账户成功，金额为" + iocash +"，当前金额为" + myCount.getCash());   
                        //释放写锁   
                        myLock.writeLock().unlock();   
                }   
        }   
}   
  
/\*\*   
\* 信用卡账户，可随意透支   
\*/   
class MyCount {   
        private String oid;         //账号   
        private int cash;             //账户余额   
  
        MyCount(String oid, int cash) {   
                this.oid = oid;   
                this.cash = cash;   
        }   
  
        public String getOid() {   
                return oid;   
        }   
  
        public void setOid(String oid) {   
                this.oid = oid;   
        }   
  
        public int getCash() {   
                return cash;   
        }   
  
        public void setCash(int cash) {   
                this.cash = cash;   
        }   
  
        @Override   
        public String toString() {   
                return "MyCount{" +   
                                "oid='" + oid + '\'' +   
                                ", cash=" + cash +   
                                '}';   
        }   
}

写：张三正在操作MyCount{oid='95599200901215522', cash=10000}账户，金额为-4000，当前金额为10000   
写：张三操作MyCount{oid='95599200901215522', cash=6000}账户成功，金额为-4000，当前金额为6000   
写：张三他弟正在操作MyCount{oid='95599200901215522', cash=6000}账户，金额为-8000，当前金额为6000   
写：张三他弟操作MyCount{oid='95599200901215522', cash=-2000}账户成功，金额为-8000，当前金额为-2000   
写：张三正在操作MyCount{oid='95599200901215522', cash=-2000}账户，金额为800，当前金额为-2000   
写：张三操作MyCount{oid='95599200901215522', cash=-1200}账户成功，金额为800，当前金额为-1200   
读：张三他爹正在查询MyCount{oid='95599200901215522', cash=-1200}账户，当前金额为-1200   
写：张三他爹正在操作MyCount{oid='95599200901215522', cash=-1200}账户，金额为6000，当前金额为-1200   
写：张三他爹操作MyCount{oid='95599200901215522', cash=4800}账户成功，金额为6000，当前金额为4800   
  
Process finished with exit code 0

**在实际开发中，最好在能用读写锁的情况下使用读写锁，而不要用普通锁，以求更好的性能。**