**JAVA线程**

[1 JAVA线程机制 3](#_Toc376382449)

[1.1 Java线程启动方式 3](#_Toc376382450)

[1.1.1 扩展java.lang.Thread类 3](#_Toc376382451)

[1.1.2 实现java.lang.Runnable接口 5](#_Toc376382452)

[1.2 Java线程内存模型 8](#_Toc376382453)

[1.2.1 Java内存模型 8](#_Toc376382454)

[1.2.2 从JMM的角度审视synchronized关键字 9](#_Toc376382455)

[1.3 线程状态转换图 10](#_Toc376382456)

[1.4 线程的调度 12](#_Toc376382457)

[1.4.1 线程调度优先级setPriority() 12](#_Toc376382458)

[1.4.2 线程睡眠sleep() 13](#_Toc376382459)

[1.4.3 线程等待 wait() 13](#_Toc376382460)

[1.4.4 线程让步yield() 15](#_Toc376382461)

[1.4.5 线程加入join() 15](#_Toc376382462)

[1.4.6 线程唤醒 notify()/notifyAll() 15](#_Toc376382463)

[1.4.7 Thread中suspend()和resume() 19](#_Toc376382464)

[2 Java5线程新特征 19](#_Toc376382465)

[2.1 Java线程：新特征-线程池 19](#_Toc376382466)

[2.1.1 一、固定大小的线程池 20](#_Toc376382467)

[2.1.2 二、单任务线程池 21](#_Toc376382468)

[2.1.3 三、可变尺寸的线程池 22](#_Toc376382469)

[2.1.4 四、延迟连接池 22](#_Toc376382470)

[2.1.5 五、单任务延迟连接池 23](#_Toc376382471)

[2.1.6 六、自定义线程池 24](#_Toc376382472)

[2.1.7 ThreadPoolExecutor 25](#_Toc376382473)

[2.2 Java线程：新特征-有返回值的线程 26](#_Toc376382474)

[2.3 Java线程：新特征-锁（上） 28](#_Toc376382475)

[2.4 Java线程：新特征-锁（下） 31](#_Toc376382476)

[2.5 Java线程：新特征-信号量 35](#_Toc376382477)

[2.6 Java线程：新特征-阻塞队列 37](#_Toc376382478)

[2.7 Java线程：新特征-阻塞栈 39](#_Toc376382479)

[2.8 Java线程：新特征-条件变量 40](#_Toc376382480)

[2.9 Java线程：新特征-原子量 51](#_Toc376382481)

[2.10 Java线程：新特征-障碍器 55](#_Toc376382482)

[3 多线程编程注意事项 57](#_Toc376382483)

[3.1 Wait()必须放到循环内 57](#_Toc376382484)

[3.2 尽可能使用NotifyAll 59](#_Toc376382485)

[3.3 尽量不要使用ThreadGroup 59](#_Toc376382486)

[3.4 尽量不要在synchronized代码块中调用sleep方法 60](#_Toc376382487)

[4 JAVA线程中的设计模式 61](#_Toc376382488)

[4.1 Single Threaded Execution Pattern 61](#_Toc376382489)

[4.2 Immutable Pattern 61](#_Toc376382490)

[4.3 Guarded Suspension Pattern 61](#_Toc376382491)

[4.4 Balking Pattern 61](#_Toc376382492)

[4.5 Producer-Consumer Pattern 61](#_Toc376382493)

[4.6 Read-Write Lock Pattern 61](#_Toc376382494)

[4.7 Thread-Per-Message Pattern 61](#_Toc376382495)

[4.8 Worker Thread Pattern 61](#_Toc376382496)

[4.9 Future Pattern 61](#_Toc376382497)

[4.10 Two-Parse Termination Pattern 61](#_Toc376382498)

[4.11 Thread-Specific Storage Pattern 61](#_Toc376382499)

[4.12 Active Object Pattern 62](#_Toc376382500)

[5 其它点滴知识： 62](#_Toc376382501)

[5.1 守护线程 62](#_Toc376382502)

[5.2 Volatile使用说明 62](#_Toc376382503)

# JAVA线程机制

## Java线程启动方式

### 扩展java.lang.Thread类

/\*\*

 \* Readme:      通过扩展Thread类实现多线程

 \*/

|  |
| --- |
| public class TestMitiThread {      public static void main(String[] rags) {          System.out.println(Thread.currentThread().getName() + " 线程运行开始!");          new MitiSay("A").start();          new MitiSay("B").start();          System.out.println(Thread.currentThread().getName() + " 线程运行结束!");      }  }    class MitiSay extends Thread {      public MitiSay(String threadName) {          super(threadName);      }        public void run() {          System.out.println(getName() + " 线程运行开始!");          for (int i = 0; i < 10; i++) {              System.out.println(i + " " + getName());              try {                  sleep((int) Math.random() \* 10);              } catch (InterruptedException e) {                  e.printStackTrace();              }          }          System.out.println(getName() + " 线程运行结束!");      }  } |

运行结果：

main 线程运行开始!

main 线程运行结束!

A 线程运行开始!

0 A

1 A

B 线程运行开始!

2 A

0 B

3 A

4 A

1 B

5 A

6 A

7 A

8 A

9 A

A 线程运行结束!

2 B

3 B

4 B

5 B

6 B

7 B

8 B

9 B

B 线程运行结束!

说明：

程序启动运行main时候，java虚拟机启动一个进程，主线程main在main()调用时候被创建。随着调用MitiSay的两个对象的start方法，另外两个线程也启动了，这样，整个应用就在多线程下运行。

在一个方法中调用Thread.currentThread().getName()方法，可以获取当前线程的名字。在mian方法中调用该方法，获取的是主线程的名字。

 注意：start()方法的调用后并不是立即执行多线程代码，而是使得该线程变为可运行态（Runnable），什么时候运行是由操作系统决定的。

从程序运行的结果可以发现，多线程程序是乱序执行。因此，只有乱序执行的代码才有必要设计为多线程。

Thread.sleep()方法调用目的是不让当前线程独自霸占该进程所获取的CPU资源，以留出一定时间给其他线程执行的机会。

实际上所有的多线程代码执行顺序都是不确定的，每次执行的结果都是随机的。

### 实现java.lang.Runnable接口

 /\*\*

 \* 通过实现 Runnable 接口实现多线程

 \*/

|  |
| --- |
| public class TestMitiThread1 implements Runnable {      public static void main(String[] args) {          System.out.println(Thread.currentThread().getName() + " 线程运行开始!");          TestMitiThread1 test = new TestMitiThread1();          Thread thread1 = new Thread(test);          Thread thread2 = new Thread(test);          thread1.start();          thread2.start();          System.out.println(Thread.currentThread().getName() + " 线程运行结束!");      }        public void run() {          System.out.println(Thread.currentThread().getName() + " 线程运行开始!");          for (int i = 0; i < 10; i++) {              System.out.println(i + " " + Thread.currentThread().getName());              try {                  Thread.sleep((int) Math.random() \* 10);              } catch (InterruptedException e) {                  e.printStackTrace();              }          }          System.out.println(Thread.currentThread().getName() + " 线程运行结束!");      }  } |

运行结果：

main 线程运行开始!

Thread-0 线程运行开始!

main 线程运行结束!

0 Thread-0

Thread-1 线程运行开始!

0 Thread-1

1 Thread-1

1 Thread-0

2 Thread-0

2 Thread-1

3 Thread-0

3 Thread-1

4 Thread-0

4 Thread-1

5 Thread-0

6 Thread-0

5 Thread-1

7 Thread-0

8 Thread-0

6 Thread-1

9 Thread-0

7 Thread-1

Thread-0 线程运行结束!

8 Thread-1

9 Thread-1

Thread-1 线程运行结束!

说明：

TestMitiThread1类通过实现Runnable接口，使得该类有了多线程类的特征。run（）方法是多线程程序的一个约定。所有的多线程代码都在run方法里面。Thread类实际上也是实现了Runnable接口的类。

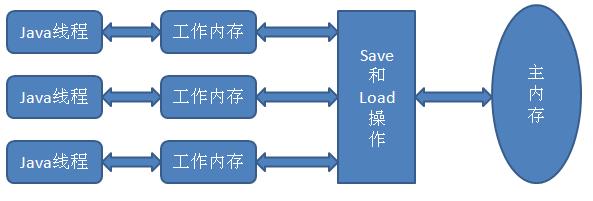
在启动的多线程的时候，需要先通过Thread类的构造方法Thread(Runnable target) 构造出对象，然后调用Thread对象的start()方法来运行多线程代码。

实际上所有的多线程代码都是通过运行Thread的start()方法来运行的。因此，不管是扩展Thread类还是实现Runnable接口来实现多线程，最终还是通过Thread的对象的API来控制线程的，熟悉Thread类的API是进行多线程编程的基础。

## Java线程内存模型

### Java内存模型

Java内存模型规定了所有变量都存储在主内存中，每条线程都有自己的工作内存，线程的工作内存保存了被该线程使用到变量的主内存副本拷贝，线程对变量的所有操作(读取,赋值等)都必须在工作内存中进行，而不能直接读写主内存中的变量。不同线程也不能直接访问对方工作内存中的变量，线程间变量值的传递均需要通过主内存来完成，线程，主内存，工作内存三者的交互关系如图所示。



      关于主内存与工作内存之间具体的交互协议，即一个变量如何从主内存拷贝到工作内存，如何从工作同步回主内存之类的实现细节，Java内存模型中定义了八种操作来完成：

* lock(锁定)：作用于主内存的变量它拔一个变量标识为一条线程独占的状态。
* unlock(解锁)：作用于主内存的变量，它把一个处于锁定状态变量释放出来，释放后的变量才可以被其他线程锁定。
* read(读取)：作用于主内的变量，把一个变量的值从主内存传输到线程的工作内存中，以便随后的load动作使用。
* load(载入)：作用于工作内存的变量，把read读取操作从主内存中得到的变量值放入工作内存的变量拷贝中。
* use(使用)：作用于工作内存的变量，把工作内存中一个变量的值传递给java虚拟机执行引擎，每当虚拟机遇到一个需要使用到变量值的字节码指令时将会执行该操作。
* assign(赋值)：作用于工作内存变量，把一个从执行引擎接收到的变量的值赋值给工作变量，每当虚拟机遇到一个给变量赋值的字节码时将会执行该操作。
* store(存储)：作用于工作内存的变量，把工作内存中一个变量的值传送到主内存中，以便随后的write操作使用。
* write(写入)：作用于主内存的变量，把store操作从工作内存中得到的变量值放入主内存的变量中。

    如果把一个变量从主内存复制到工作内存，那就需要顺序地执行read和load操作，如果要把变量从工作内存同步到主内存，就要顺序地执行store和write操作。注意Java内存模型只要求上述两个操作必须按照顺序执行，而没有保证必须是连续执行。也即是说read和load之间，store和write之间是可插入其他指令的，如对主内存中的变量a,b进行访问时，一种可能的顺序是read a,read b,load b,load a。除此之外，Java内存模型还规定了在执行上述八种基本操作时必须满足如下规则：

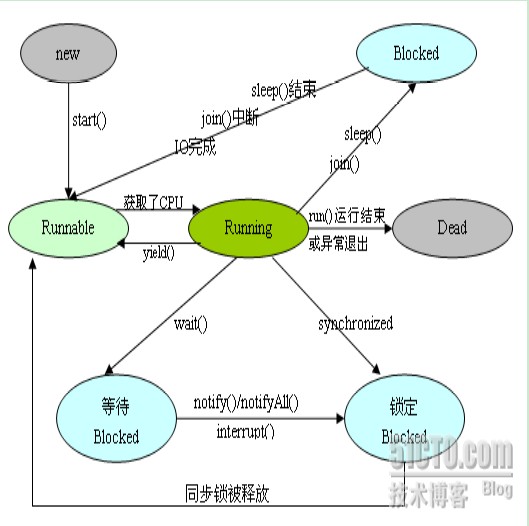
* 不允许read和load、store和write操作之一单独出现，即不允许一个变量从主内存读取可但工作内存不接受，或者从工作内存发起了回写但主内存不接受的情况出现。
* 不允许一个线程丢弃它的最近的assign赋值操作，即工作内存变量值改变之后必须同步回主内存。
* 只有发生过assign赋值操作的变量才需要从工作内存同步回主内存。
* 一个新变量只能在主内存中产生，不允许在工作内存中直接使用一个未被初始化(load或assign)的变量，即一个变量在进行use和store操作之前，必须先执行过assgin和load操作
* 一个变量在同一时刻只允许一条线程对其进行lock锁定操作，但是lock锁定可以被一条线程重复执行多次，多次执行lock之后，只有执行相同次数的unlock操作变量才会被解锁
* 如果对一个变量执行lock锁定操作，将会清空工作内存中该变量的值，在执行引擎使用这个变量前，需要重新执行load或assign操作初始化变量的值。
* 如果一个变量事先没有被lock锁定，则不允许对这个变量进行unlock解锁操作，也不允许对一个被别的线程锁定的变量进行unlock解锁。
* 一个变量进行unlock解锁操作之前，必须先把此变量同步回主内存中(执行store和write操作)。

### 从JMM的角度审视synchronized关键字

假设某条线程执行一个synchronized代码段，其间对某变量进行操作，JVM会依次执行如下动作：   
(1) 获取同步对象monitor (lock)   
(2) 从主存复制变量到当前工作内存 (read and load)   
(3) 执行代码，改变共享变量值 (use and assign)   
(4) 用工作内存数据刷新主存相关内容 (store and write)   
(5) 释放同步对象锁 (unlock)   
可见，synchronized的另外一个作用是保证主存内容和线程的工作内存中的数据的一致性。如果没有使用synchronized关键字，JVM不保证第2步和第4步会严格按照上述次序立即执行。因为根据JLS中的规定，线程的工作内存和主存之间的数据交换是松耦合的，什么时候需要刷新工作内存或者更新主内存内容，可以由具体的虚拟机实现自行决定。如果多个线程同时执行一段未经synchronized保护的代码段，很有可能某条线程已经改动了变量的值，但是其他线程却无法看到这个改动，依然在旧的变量值上进行运算，最终导致不可预料的运算结果

## 线程状态转换图

线程在一定条件下，状态会发生变化。线程变化的状态转换图如下：

http://lavasoft.blog.51cto.com/blank.gif

1、新建状态（New）：新创建了一个线程对象。

2、就绪状态（Runnable）：线程对象创建后，其他线程调用了该对象的start()方法。该状态的线程位于可运行线程池中，变得可运行，等待获取CPU的使用权。

3、运行状态（Running）：就绪状态的线程获取了CPU，执行程序代码。

4、阻塞状态（Blocked）：阻塞状态是线程因为某种原因放弃CPU使用权，暂时停止运行。直到线程进入就绪状态，才有机会转到运行状态。阻塞的情况分三种：

（一）、等待阻塞：运行的线程执行wait()方法，JVM会把该线程放入等待池中。

（二）、同步阻塞：运行的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则JVM会把该线程放入锁池中。

（三）、其他阻塞：运行的线程执行sleep()或join()方法，或者发出了I/O请求时，JVM会把该线程置为阻塞状态。当sleep()状态超时、join()等待线程终止或者超时、或者I/O处理完毕时，线程重新转入就绪状态。

5、死亡状态（Dead）：线程执行完了或者因异常退出了run()方法，该线程结束生命周期。

## 线程的调度

### 线程调度优先级setPriority()

调整线程优先级：Java线程有优先级，优先级高的线程会获得较多的运行机会。

Java线程的优先级用整数表示，取值范围是1~10，Thread类有以下三个静态常量：

static int MAX\_PRIORITY

          线程可以具有的最高优先级，取值为10。

static int MIN\_PRIORITY

          线程可以具有的最低优先级，取值为1。

static int NORM\_PRIORITY

          分配给线程的默认优先级，取值为5。

Thread类的setPriority()和getPriority()方法分别用来设置和获取线程的优先级

每个线程都有默认的优先级。主线程的默认优先级为Thread.NORM\_PRIORITY。

线程的优先级有继承关系，比如A线程中创建了B线程，那么B将和A具有相同的优先级。

JVM提供了10个线程优先级，但与常见的操作系统都不能很好的映射。如果希望程序能移植到各个操作系统中，应该仅仅使用Thread类有以下三个静态常量作为优先级，这样能保证同样的优先级采用了同样的调度方式。

### 线程睡眠sleep()

线程睡眠：Thread.sleep(long millis)方法，使线程转到阻塞状态。millis参数设定睡眠的时间，以毫秒为单位。当睡眠结束后，就转为就绪（Runnable）状态。sleep()平台移植性好。

sleep()使当前线程进入停滞状态（阻塞当前线程），让出CUP的使用、目的是不让当前线程独自霸占该进程所获的CPU资源，以留一定时间给其他线程执行的机会;

sleep()是Thread类的Static(静态)的方法；因此他不能改变对象的机锁，所以当在一个Synchronized块中调用Sleep()方法是，线程虽然休眠了，但是对象的机锁并没有被释放，其他线程无法访问这个对象（即使睡着也持有对象锁）。  
　　在sleep()休眠时间期满后，该线程不一定会立即执行，这是因为其它线程可能正在运行而且没有被调度为放弃执行，除非此线程具有更高的优先级

### 线程等待 wait()

线程等待：Object类中的wait()方法，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() 唤醒方法。这个两个唤醒方法也是Object类中的方法，行为等价于调用 wait(0) 一样。当wait()方法被调用时，对象锁会被释放。

当前线程必须拥有此对象监视器 。该线程发布对此监视器的所有权并等待，直到其他线程通过调用 notify 方法，或 notifyAll 方法通知在此对象的监视器上等待的线程醒来。然后该线程将等到重新获得对监视器的所有权后才能继续执行。

对于某一个参数的版本，实现中断和虚假唤醒是可能的，而且此方法应始终在循环中使用：

|  |
| --- |
| synchronized (obj) {  while (<condition does not hold>)  obj.wait();  ... // Perform action appropriate to condition  } |

此方法只应由作为此对象监视器的所有者的线程来调用。

**抛出：**

[IllegalMonitorStateException](http://salever.iteye.com/java/lang/IllegalMonitorStateException.html) - 如果当前线程不是此对象监视器的所有者。

[InterruptedException](http://salever.iteye.com/java/lang/InterruptedException.html) - 如果在当前线程等待通知之前或者正在等待通知时，任何线程中断了当前线程。在抛出此异常时，当前线程的*中断状态* 被清除。

对于红色部分的内容，个人曾一直都不是很理解，什么叫做拥有此对象的监视器。下面我们看看代码：

|  |
| --- |
| DateFormat format = new SimpleDateFormat("yyyy-MM-dd:hh:mm:ss");  private String getTime(){  return format.format(Calendar.getInstance().getTime());  }  private Object monitor = new Object();  public void waitOnce(String thread, final long ms) {  Thread waitThread = new Thread() {  public void run() {  synchronized (monitor) {//获得对象监视器  try {  System.out.println("Thread " + Thread.currentThread().getName() + " Wait at " + getTime());  monitor.wait(ms);  System.out.println("Thread " + Thread.currentThread().getName() + " Waked at " + getTime());  } catch (InterruptedException e) {  }  }  };  };  waitThread.setName(thread);  waitThread.start();  } |

如果我们去掉synchronized(monitor) ，运行则会出现异常[IllegalMonitorStateException。](http://salever.iteye.com/java/lang/IllegalMonitorStateException.html)

WaitAndNotifyTest test = new WaitAndNotifyTest();

test.waitOnce("1", Long.MAX\_VALUE);

Exception in thread “1″ java.lang.IllegalMonitorStateException

而加上以后就没问题了。因此个人觉得使用synchronized关键字锁定对象，也就是获得了对象的监视器了

### 线程让步yield()

线程让步：Thread.yield() 方法，暂停当前正在执行的线程对象，把执行机会让给相同或者更高优先级的线程。

### 线程加入join()

线程加入：join()方法，等待其他线程终止。在当前线程中调用另一个线程的join()方法，则当前线程转入阻塞状态，直到另一个进程运行结束，当前线程再由阻塞转为就绪状态。

### 线程唤醒 notify()/notifyAll()

线程唤醒：Object类中的notify()方法，唤醒在此对象监视器上等待的单个线程。如果所有线程都在此对象上等待，则会选择唤醒其中一个线程。选择是任意性的，并在对实现做出决定时发生。线程通过调用其中一个 wait 方法，在对象的监视器上等待。 直到当前的线程放弃此对象上的锁定，才能继续执行被唤醒的线程。被唤醒的线程将以常规方式与在该对象上主动同步的其他所有线程进行竞争；例如，唤醒的线程在作为锁定此对象的下一个线程方面没有可靠的特权或劣势。类似的方法还有一个notifyAll()，唤醒在此对象监视器上等待的所有线程。

1. notify()

唤醒在此对象监视器上等待的单个线程。如果所有线程都在此对象上等待，则会选择唤醒其中一个线程 。选择是任意性 的，并在对实现做出决定时发生。线程通过调用其中一个 wait 方法，在对象的监视器上等待。

直到当前线程放弃此对象上的锁定，才能继续执行被唤醒的线程 。被唤醒的线程将以常规方式与在该对象上主动同步的其他所有线程进行竞争；例如，唤醒的线程在作为锁定此对象的下一个线程方面没有可靠的特权或劣势。

此方法只应由作为此对象监视器的所有者的线程来调用。通过以下三种方法之一，线程可以成为此对象监视器的所有者：

通过执行此对象的同步实例方法。  
通过执行在此对象上进行同步的 synchronized 语句的正文。  
对于 Class 类型的对象，可以通过执行该类的同步静态方法。

一次只能有一个线程拥有对象的监视器。

抛出：  
IllegalMonitorStateException – 如果当前线程不是此对象监视器的所有者。

首先理解一下获得对象的监视器，简单的说就是取得了当前对象的“加锁”使用权，最简单的就是使用synchronized关键字。另外使用 synchronized修饰的方法也行。

notify方法还有一个值得提出的是它会在当前线程释放了对象锁以后随机唤醒一个在该对象上等待的线程 。

看看一个例子：

|  |
| --- |
| public void awakeAndWait(String thread, final long ms) {  Thread notifyThread = new Thread() {  public void run() {  synchronized (monitor) {  monitor.notify();  System.out.println("Thread " + Thread.currentThread().getName() + " Notify at " + getTime());  //保持了对象锁的等待  try {  Thread.sleep(ms);  } catch (InterruptedException e)  {  }  }  //释放了对象锁之后的等待  try {  Thread.sleep(ms);  } catch (InterruptedException e) {  }  };  };  notifyThread.setName(thread);  notifyThread.start();  } |

这个方法会唤醒一个在对象上等待的线程，并在两次sleep后退出，注意的是一个sleep是在对象锁内，而另一次则是在释放了对象锁以后，这时候运行上面2个方法得到：

|  |
| --- |
| WaitAndNotifyTest test = new WaitAndNotifyTest();  test.waitOnce("1", Long.MAX\_VALUE);// 在对象上等待无限长  test.waitOnce("2", Long.MAX\_VALUE);// 在对象上等待无限长  test.waitOnce("3", Long.MAX\_VALUE);// 在对象上等待无限长  try {  // 延迟2s  Thread.sleep(2000);  } catch (InterruptedException e) {  }  // 在唤醒一个在对象上等待的线程，本身执行时间4s，2s是在对象锁内  //，2s是在释放了对象锁以后  test.awakeAndWait("3", 2000); |

执行结果为：

Thread 1 Wait at 2011-05-06:10:57:04  
Thread 2 Wait at 2011-05-06:10:57:04  
Thread 3 Wait at 2011-05-06:10:57:04  
Thread 3 Notify at 2011-05-06:10:57:06  
Thread 1 Waked at 2011-05-06:10:57:08

2秒后唤醒了线程1，尽管它自己执行花了4s，在释放了对象锁之后的2s不会影响线程1的执行

1. notifyAll()

唤醒在此对象监视器上等待的所有线程 。线程通过调用其中一个 wait 方法，在对象的监视器上等待。  
直到当前线程放弃此对象上的锁定，才能继续执行被唤醒的线程。被唤醒的线程将以常规方式与在该对象上主动同步的其他所有线程进行竞争；例如，唤醒的线程在作为锁定此对象的下一个线程方面没有可靠的特权或劣势。

此方法只应由作为此对象监视器的所有者的线程来调用。有关线程能够成为监视器所有者的方法的描述，请参阅 notify 方法。

抛出：  
IllegalMonitorStateException – 如果当前线程不是此对象监视器的所有者。

与notify稍微有一点差别的是，它会唤醒所有的等待线程。

|  |
| --- |
| public void awakeAll(String thread) {  Thread notifyThread = new Thread() {  public void run() {  synchronized (monitor) {  monitor.notifyAll();  System.out.println("Thread "+ Thread.currentThread().getName()+ " Notify all at " + getTime());  }  };  };  notifyThread.setName(thread);  notifyThread.start();  } |

执行一下代码：

|  |
| --- |
| WaitAndNotifyTest test = new WaitAndNotifyTest();  test.waitOnce("1", Long.MAX\_VALUE);// 在对象上等待无限长  test.waitOnce("2", Long.MAX\_VALUE);// 在对象上等待无限长  test.waitOnce("3", Long.MAX\_VALUE);// 在对象上等待无限长  try {// 延迟2s  Thread.sleep(2000);  } catch (InterruptedException e) {  }  test.awakeAll("4"); |

结果为：

Thread 1 Wait at 2011-05-06:10:59:15  
Thread 3 Wait at 2011-05-06:10:59:15  
Thread 2 Wait at 2011-05-06:10:59:15  
Thread 4 Notify all at 2011-05-06:10:59:17  
Thread 2 Waked at 2011-05-06:10:59:17  
Thread 3 Waked at 2011-05-06:10:59:17  
Thread 1 Waked at 2011-05-06:10:59:17

全部唤醒了

### Thread中suspend()和resume()

注意：Thread中suspend()和resume()两个方法在JDK1.5中已经废除，不再介绍。因为有死锁倾向。

# Java5线程新特征

## **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-   
\*/   
public class Test {   
        public static void 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   
        public void 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-   
\*/   
public class Test {   
        public static void 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   
        public void 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   
\*/   
public class Test {   
        public static void 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 = new ThreadPoolExecutor(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   
        public void 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](mk:@MSITStore:F:\CHM\jdk150.ZH_cn.chm::/jdk150/api/java/util/concurrent/TimeUnit.html) unit,

[BlockingQueue](mk:@MSITStore:F:\CHM\jdk150.ZH_cn.chm::/jdk150/api/java/util/concurrent/BlockingQueue.html)<[Runnable](mk:@MSITStore:F:\CHM\jdk150.ZH_cn.chm::/jdk150/api/java/lang/Runnable.html)> 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](mk:@MSITStore:F:\CHM\jdk150.ZH_cn.chm::/jdk150/api/java/util/concurrent/Executors.html) 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](mk:@MSITStore:F:\CHM\jdk150.ZH_cn.chm::/jdk150/api/java/lang/IllegalArgumentException.html) - 如果 corePoolSize 或 keepAliveTime 小于零，或者 maximumPoolSize 小于或等于零，或者 corePoolSize 大于 maximumPoolSize。

[NullPointerException](mk:@MSITStore:F:\CHM\jdk150.ZH_cn.chm::/jdk150/api/java/lang/NullPointerException.html) - 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

## 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   
\*/   
public class Test {   
        public static void 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 MyCallable implements 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**](mk:@MSITStore:F:\CHM\jdk150.ZH_cn.chm::/jdk150/api/java/util/concurrent/locks/Condition.html) | Condition 将 Object 监视器方法（[wait](mk:@MSITStore:F:\CHM\jdk150.ZH_cn.chm::/jdk150/api/java/lang/Object.html#wait())、[notify](mk:@MSITStore:F:\CHM\jdk150.ZH_cn.chm::/jdk150/api/java/lang/Object.html#notify()) 和 [notifyAll](mk:@MSITStore:F:\CHM\jdk150.ZH_cn.chm::/jdk150/api/java/lang/Object.html#notifyAll())）分解成截然不同的对象，以便通过将这些对象与任意 [Lock](mk:@MSITStore:F:\CHM\jdk150.ZH_cn.chm::/jdk150/api/java/util/concurrent/locks/Lock.html) 实现组合使用，为每个对象提供多个等待 set （wait-set）。 |
| [**Lock**](mk:@MSITStore:F:\CHM\jdk150.ZH_cn.chm::/jdk150/api/java/util/concurrent/locks/Lock.html) | 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**](mk:@MSITStore:F:\CHM\jdk150.ZH_cn.chm::/jdk150/api/java/util/concurrent/locks/ReadWriteLock.html) | ReadWriteLock 维护了一对相关的[锁定](mk:@MSITStore:F:\CHM\jdk150.ZH_cn.chm::/jdk150/api/java/util/concurrent/locks/Lock.html)，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   
\*/   
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：锁   
\*/   
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

## Java线程：新特征-信号量

Java的信号量实际上是一个功能完毕的计数器，对控制一定资源的消费与回收有着很重要的意义，信号量常常用于多线程的代码中，并能监控有多少数目的线程等待获取资源，并且通过信号量可以得知可用资源的数目等等，这里总是在强调“数目”二字，但不能指出来有哪些在等待，哪些资源可用。

因此，本人认为，这个信号量类如果能返回数目，还能知道哪些对象在等待，哪些资源可使用，就非常完美了，仅仅拿到这些概括性的数字，对精确控制意义不是很大。目前还没想到更好的用法。

下面是一个简单例子：

import java.util.concurrent.ExecutorService;   
import java.util.concurrent.Executors;   
import java.util.concurrent.Semaphore;   
  
/\*\*   
\* Java线程：新特征-信号量   
\*/   
public class Test {   
        public static void main(String[] args) {   
                MyPool myPool = new MyPool(20);   
                //创建线程池   
                ExecutorService threadPool = Executors.newFixedThreadPool(2);   
                MyThread t1 = new MyThread("任务A", myPool, 3);   
                MyThread t2 = new MyThread("任务B", myPool, 12);   
                MyThread t3 = new MyThread("任务C", myPool, 7);   
                //在线程池中执行任务   
                threadPool.execute(t1);   
                threadPool.execute(t2);   
                threadPool.execute(t3);   
                //关闭池   
                threadPool.shutdown();   
        }   
}   
  
/\*\*   
\* 一个池   
\*/   
class MyPool {   
        private Semaphore sp;     //池相关的信号量   
  
        /\*\*   
         \* 池的大小，这个大小会传递给信号量   
         \*   
         \* @param size 池的大小   
         \*/   
        MyPool(int size) {   
                this.sp = new Semaphore(size);   
        }   
  
        public Semaphore getSp() {   
                return sp;   
        }   
  
        public void setSp(Semaphore sp) {   
                this.sp = sp;   
        }   
}   
  
class MyThread extends Thread {   
        private String threadname;            //线程的名称   
        private MyPool pool;                        //自定义池   
        private int x;                                    //申请信号量的大小   
  
        MyThread(String threadname, MyPool pool, int x) {   
                this.threadname = threadname;   
                this.pool = pool;   
                this.x = x;   
        }   
  
        public void run() {   
                try {   
                        //从此信号量获取给定数目的许可   
                        pool.getSp().acquire(x);   
                        //todo：也许这里可以做更复杂的业务   
                        System.out.println(threadname + "成功获取了" + x + "个许可！");   
                } catch (InterruptedException e) {   
                        e.printStackTrace();   
                } finally {   
                        //释放给定数目的许可，将其返回到信号量。   
                        pool.getSp().release(x);   
                        System.out.println(threadname + "释放了" + x + "个许可！");   
                }   
        }   
}

任务B成功获取了12个许可！   
任务B释放了12个许可！   
任务A成功获取了3个许可！   
任务C成功获取了7个许可！   
任务C释放了7个许可！   
任务A释放了3个许可！   
  
Process finished with exit code 0

从结果可以看出，信号量仅仅是对池资源进行监控，但不保证线程的安全，因此，在使用时候，应该自己控制线程的安全访问池资源

## Java线程：新特征-阻塞队列

阻塞队列是Java5线程新特征中的内容，Java定义了阻塞队列的接口java.util.concurrent.BlockingQueue，阻塞队列的概念是，一个指定长度的队列，如果队列满了，添加新元素的操作会被阻塞等待，直到有空位为止。同样，当队列为空时候，请求队列元素的操作同样会阻塞等待，直到有可用元素为止。

有了这样的功能，就为多线程的排队等候的模型实现开辟了便捷通道，非常有用。

java.util.concurrent.BlockingQueue继承了java.util.Queue接口，可以参看API文档。

下面给出一个简单应用的例子：

import java.util.concurrent.BlockingQueue;   
import java.util.concurrent.ArrayBlockingQueue;   
  
/\*\*   
\* Java线程：新特征-阻塞队列   
\*/   
public class Test {   
        public static void main(String[] args) throws InterruptedException {   
                BlockingQueue bqueue = new ArrayBlockingQueue(20);   
                for (int i = 0; i < 30; i++) {   
                        //将指定元素添加到此队列中，如果没有可用空间，将一直等待（如果有必要）。   
                        bqueue.put(i);   
                        System.out.println("向阻塞队列中添加了元素:" + i);   
                }   
                System.out.println("程序到此运行结束，即将退出----");   
        }   
}

输出结果：

向阻塞队列中添加了元素:0   
向阻塞队列中添加了元素:1   
向阻塞队列中添加了元素:2   
向阻塞队列中添加了元素:3   
向阻塞队列中添加了元素:4   
向阻塞队列中添加了元素:5   
向阻塞队列中添加了元素:6   
向阻塞队列中添加了元素:7   
向阻塞队列中添加了元素:8   
向阻塞队列中添加了元素:9   
向阻塞队列中添加了元素:10   
向阻塞队列中添加了元素:11   
向阻塞队列中添加了元素:12   
向阻塞队列中添加了元素:13   
向阻塞队列中添加了元素:14   
向阻塞队列中添加了元素:15   
向阻塞队列中添加了元素:16   
向阻塞队列中添加了元素:17   
向阻塞队列中添加了元素:18   
向阻塞队列中添加了元素:19

可以看出，输出到元素19时候，就一直处于等待状态，因为队列满了，程序阻塞了。

这里没有用多线程来演示，没有这个必要。

另外，阻塞队列还有更多实现类，用来满足各种复杂的需求：ArrayBlockingQueue, DelayQueue, LinkedBlockingQueue, PriorityBlockingQueue, SynchronousQueue ，具体的API差别也很小

## Java线程：新特征-阻塞栈

对于阻塞栈，与阻塞队列相似。不同点在于栈是“后入先出”的结构，每次操作的是栈顶，而队列是“先进先出”的结构，每次操作的是队列头。

这里要特别说明一点的是，阻塞栈是Java6的新特征。、

Java为阻塞栈定义了接口：java.util.concurrent.BlockingDeque，其实现类也比较多，具体可以查看JavaAPI文档。

下面看一个简单例子：

import java.util.concurrent.BlockingDeque;   
import java.util.concurrent.LinkedBlockingDeque;   
  
/\*\*   
\* Java线程：新特征-阻塞栈   
\*/   
public class Test {   
        public static void main(String[] args) throws InterruptedException {   
                BlockingDeque bDeque = new LinkedBlockingDeque(20);   
                for (int i = 0; i < 30; i++) {   
                        //将指定元素添加到此阻塞栈中，如果没有可用空间，将一直等待（如果有必要）。   
                        bDeque.putFirst(i);   
                        System.out.println("向阻塞栈中添加了元素:" + i);   
                }   
                System.out.println("程序到此运行结束，即将退出----");   
        }   
}

输出结果：

向阻塞栈中添加了元素:0   
向阻塞栈中添加了元素:1   
向阻塞栈中添加了元素:2   
向阻塞栈中添加了元素:3   
向阻塞栈中添加了元素:4   
向阻塞栈中添加了元素:5   
向阻塞栈中添加了元素:6   
向阻塞栈中添加了元素:7   
向阻塞栈中添加了元素:8   
向阻塞栈中添加了元素:9   
向阻塞栈中添加了元素:10   
向阻塞栈中添加了元素:11   
向阻塞栈中添加了元素:12   
向阻塞栈中添加了元素:13   
向阻塞栈中添加了元素:14   
向阻塞栈中添加了元素:15   
向阻塞栈中添加了元素:16   
向阻塞栈中添加了元素:17   
向阻塞栈中添加了元素:18   
向阻塞栈中添加了元素:19

从上面结果可以看到，程序并没结束，二是阻塞住了，原因是栈已经满了，后面追加元素的操作都被阻塞了

## Java线程：新特征-条件变量

条件变量是Java5线程中很重要的一个概念，顾名思义，条件变量就是表示条件的一种变量。但是必须说明，这里的条件是没有实际含义的，仅仅是个标记而已，并且条件的含义往往通过代码来赋予其含义。

这里的条件和普通意义上的条件表达式有着天壤之别。

条件变量都实现了java.util.concurrent.locks.Condition接口，条件变量的实例化是通过一个Lock对象上调用newCondition()方法来获取的，这样，条件就和一个锁对象绑定起来了。因此，Java中的条件变量只能和锁配合使用，来控制并发程序访问竞争资源的安全。

条件变量的出现是为了更精细控制线程等待与唤醒，在Java5之前，线程的等待与唤醒依靠的是Object对象的wait()和notify()/notifyAll()方法，这样的处理不够精细。

而在Java5中，一个锁可以有多个条件，每个条件上可以有多个线程等待，通过调用await()方法，可以让线程在该条件下等待。当调用signalAll()方法，又可以唤醒该条件下的等待的线程。有关Condition接口的API可以具体参考JavaAPI文档。

条件变量比较抽象，原因是他不是自然语言中的条件概念，而是程序控制的一种手段。

下面以一个银行存取款的模拟程序为例来揭盖Java多线程条件变量的神秘面纱：

有一个账户，多个用户（线程）在同时操作这个账户，有的存款有的取款，存款随便存，取款有限制，不能透支，任何试图透支的操作都将等待里面有足够存款才执行操作。

import java.util.concurrent.ExecutorService;   
import java.util.concurrent.Executors;   
import java.util.concurrent.locks.Condition;   
import java.util.concurrent.locks.Lock;   
import java.util.concurrent.locks.ReentrantLock;   
  
/\*\*   
\* Java线程：条件变量   
\*/   
public class Test {   
        public static void main(String[] args) {   
                //创建并发访问的账户   
                MyCount myCount = new MyCount("95599200901215522", 10000);   
                //创建一个线程池   
                ExecutorService pool = Executors.newFixedThreadPool(2);   
                Thread t1 = new SaveThread("张三", myCount, 2000);   
                Thread t2 = new SaveThread("李四", myCount, 3600);   
                Thread t3 = new DrawThread("王五", myCount, 2700);   
                Thread t4 = new SaveThread("老张", myCount, 600);   
                Thread t5 = new DrawThread("老牛", myCount, 1300);   
                Thread t6 = new DrawThread("胖子", myCount, 800);   
                //执行各个线程   
                pool.execute(t1);   
                pool.execute(t2);   
                pool.execute(t3);   
                pool.execute(t4);   
                pool.execute(t5);   
                pool.execute(t6);   
                //关闭线程池   
                pool.shutdown();   
        }   
}   
  
/\*\*   
\* 存款线程类   
\*/   
class SaveThread extends Thread {   
        private String name;                //操作人   
        private MyCount myCount;        //账户   
        private int x;                            //存款金额   
  
        SaveThread(String name, MyCount myCount, int x) {   
                this.name = name;   
                this.myCount = myCount;   
                this.x = x;   
        }   
  
        public void run() {   
                myCount.saving(x, name);   
        }   
}   
  
/\*\*   
\* 取款线程类   
\*/   
class DrawThread extends Thread {   
        private String name;                //操作人   
        private MyCount myCount;        //账户   
        private int x;                            //存款金额   
  
        DrawThread(String name, MyCount myCount, int x) {   
                this.name = name;   
                this.myCount = myCount;   
                this.x = x;   
        }   
  
        public void run() {   
                myCount.drawing(x, name);   
        }   
}   
  
  
/\*\*   
\* 普通银行账户，不可透支   
\*/   
class MyCount {   
        private String oid;                         //账号   
        private int cash;                             //账户余额   
        private Lock lock = new ReentrantLock();                //账户锁   
        private Condition \_save = lock.newCondition();    //存款条件   
        private Condition \_draw = lock.newCondition();    //取款条件   
  
        MyCount(String oid, int cash) {   
                this.oid = oid;   
                this.cash = cash;   
        }   
  
        /\*\*   
         \* 存款   
         \*   
         \* @param x        操作金额   
         \* @param name 操作人   
         \*/   
        public void saving(int x, String name) {   
                lock.lock();                        //获取锁   
                if (x > 0) {   
                        cash += x;                    //存款   
                        System.out.println(name + "存款" + x + "，当前余额为" + cash);   
                }   
                \_draw.signalAll();            //唤醒所有等待线程。   
                lock.unlock();                    //释放锁   
        }   
  
        /\*\*   
         \* 取款   
         \*   
         \* @param x        操作金额   
         \* @param name 操作人   
         \*/   
        public void drawing(int x, String name) {   
                lock.lock();                                 //获取锁   
                try {   
                        if (cash - x < 0) {   
                                \_draw.await();             //阻塞取款操作   
                        } else {   
                                cash -= x;                     //取款   
                                System.out.println(name + "取款" + x + "，当前余额为" + cash);   
                        }   
                        \_save.signalAll();             //唤醒所有存款操作   
                } catch (InterruptedException e) {   
                        e.printStackTrace();   
                } finally {   
                        lock.unlock();                     //释放锁   
                }   
        }   
}

李四存款3600，当前余额为13600   
张三存款2000，当前余额为15600   
老张存款600，当前余额为16200   
老牛取款1300，当前余额为14900   
胖子取款800，当前余额为14100   
王五取款2700，当前余额为11400   
  
Process finished with exit code 0

假如我们不用锁和条件变量，如何实现此功能呢？下面是实现代码：

import java.util.concurrent.ExecutorService;   
import java.util.concurrent.Executors;   
  
/\*\*   
\* Java线程：不用条件变量   
\*/   
public class Test {   
        public static void main(String[] args) {   
                //创建并发访问的账户   
                MyCount myCount = new MyCount("95599200901215522", 10000);   
                //创建一个线程池   
                ExecutorService pool = Executors.newFixedThreadPool(2);   
                Thread t1 = new SaveThread("张三", myCount, 2000);   
                Thread t2 = new SaveThread("李四", myCount, 3600);   
                Thread t3 = new DrawThread("王五", myCount, 2700);   
                Thread t4 = new SaveThread("老张", myCount, 600);   
                Thread t5 = new DrawThread("老牛", myCount, 1300);   
                Thread t6 = new DrawThread("胖子", myCount, 800);   
                //执行各个线程   
                pool.execute(t1);   
                pool.execute(t2);   
                pool.execute(t3);   
                pool.execute(t4);   
                pool.execute(t5);   
                pool.execute(t6);   
                //关闭线程池   
                pool.shutdown();   
        }   
}   
  
/\*\*   
\* 存款线程类   
\*/   
class SaveThread extends Thread {   
        private String name;                //操作人   
        private MyCount myCount;        //账户   
        private int x;                            //存款金额   
  
        SaveThread(String name, MyCount myCount, int x) {   
                this.name = name;   
                this.myCount = myCount;   
                this.x = x;   
        }   
  
        public void run() {   
                myCount.saving(x, name);   
        }   
}   
  
/\*\*   
\* 取款线程类   
\*/   
class DrawThread extends Thread {   
        private String name;                //操作人   
        private MyCount myCount;        //账户   
        private int x;                            //存款金额   
  
        DrawThread(String name, MyCount myCount, int x) {   
                this.name = name;   
                this.myCount = myCount;   
                this.x = x;   
        }   
  
        public void run() {   
                myCount.drawing(x, name);   
        }   
}   
  
  
/\*\*   
\* 普通银行账户，不可透支   
\*/   
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;   
        }   
  
        /\*\*   
         \* 存款   
         \*   
         \* @param x        操作金额   
         \* @param name 操作人   
         \*/   
        public synchronized void saving(int x, String name) {   
                if (x > 0) {   
                        cash += x;                    //存款   
                        System.out.println(name + "存款" + x + "，当前余额为" + cash);   
                }   
                notifyAll();            //唤醒所有等待线程。   
        }   
  
        /\*\*   
         \* 取款   
         \*   
         \* @param x        操作金额   
         \* @param name 操作人   
         \*/   
        public synchronized void drawing(int x, String name) {   
                if (cash - x < 0) {   
                        try {   
                                wait();   
                        } catch (InterruptedException e1) {   
                                e1.printStackTrace();   
                        }   
                } else {   
                        cash -= x;                     //取款   
                        System.out.println(name + "取款" + x + "，当前余额为" + cash);   
                }   
                notifyAll();             //唤醒所有存款操作   
        }   
}

输出结果为：

李四存款3600，当前余额为13600   
王五取款2700，当前余额为10900   
老张存款600，当前余额为11500   
老牛取款1300，当前余额为10200   
胖子取款800，当前余额为9400   
张三存款2000，当前余额为11400   
  
Process finished with exit code 0

结合先前同步代码知识，举一反三，将此例改为同步代码块来实现，代码如下：

import java.util.concurrent.ExecutorService;   
import java.util.concurrent.Executors;   
  
/\*\*   
\* Java线程：改为同步代码块   
\*/   
public class Test {   
        public static void main(String[] args) {   
                //创建并发访问的账户   
                MyCount myCount = new MyCount("95599200901215522", 10000);   
                //创建一个线程池   
                ExecutorService pool = Executors.newFixedThreadPool(2);   
                Thread t1 = new SaveThread("张三", myCount, 2000);   
                Thread t2 = new SaveThread("李四", myCount, 3600);   
                Thread t3 = new DrawThread("王五", myCount, 2700);   
                Thread t4 = new SaveThread("老张", myCount, 600);   
                Thread t5 = new DrawThread("老牛", myCount, 1300);   
                Thread t6 = new DrawThread("胖子", myCount, 800);   
                //执行各个线程   
                pool.execute(t1);   
                pool.execute(t2);   
                pool.execute(t3);   
                pool.execute(t4);   
                pool.execute(t5);   
                pool.execute(t6);   
                //关闭线程池   
                pool.shutdown();   
        }   
}   
  
/\*\*   
\* 存款线程类   
\*/   
class SaveThread extends Thread {   
        private String name;                //操作人   
        private MyCount myCount;        //账户   
        private int x;                            //存款金额   
  
        SaveThread(String name, MyCount myCount, int x) {   
                this.name = name;   
                this.myCount = myCount;   
                this.x = x;   
        }   
  
        public void run() {   
                myCount.saving(x, name);   
        }   
}   
  
/\*\*   
\* 取款线程类   
\*/   
class DrawThread extends Thread {   
        private String name;                //操作人   
        private MyCount myCount;        //账户   
        private int x;                            //存款金额   
  
        DrawThread(String name, MyCount myCount, int x) {   
                this.name = name;   
                this.myCount = myCount;   
                this.x = x;   
        }   
  
        public void run() {   
                myCount.drawing(x, name);   
        }   
}   
  
  
/\*\*   
\* 普通银行账户，不可透支   
\*/   
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;   
        }   
  
        /\*\*   
         \* 存款   
         \*   
         \* @param x        操作金额   
         \* @param name 操作人   
         \*/   
        public void saving(int x, String name) {   
                if (x > 0) {   
                        synchronized (this) {   
                                cash += x;                    //存款   
                                System.out.println(name + "存款" + x + "，当前余额为" + cash);   
                                notifyAll();            //唤醒所有等待线程。   
                        }   
                }   
        }   
  
        /\*\*   
         \* 取款   
         \*   
         \* @param x        操作金额   
         \* @param name 操作人   
         \*/   
        public synchronized void drawing(int x, String name) {   
                synchronized (this) {   
                        if (cash - x < 0) {   
                                try {   
                                        wait();   
                                } catch (InterruptedException e1) {   
                                        e1.printStackTrace();   
                                }   
                        } else {   
                                cash -= x;                     //取款   
                                System.out.println(name + "取款" + x + "，当前余额为" + cash);   
                        }   
                }   
                notifyAll();             //唤醒所有存款操作   
        }   
}

李四存款3600，当前余额为13600   
王五取款2700，当前余额为10900   
老张存款600，当前余额为11500   
老牛取款1300，当前余额为10200   
胖子取款800，当前余额为9400   
张三存款2000，当前余额为11400   
  
Process finished with exit code 0

对比以上三种方式，从控制角度上讲，第一种最灵活，第二种代码最简单，第三种容易犯错

## Java线程：新特征-原子量

所谓的原子量即操作变量的操作是“原子的”，该操作不可再分，因此是线程安全的。

为何要使用原子变量呢，原因是多个线程对单个变量操作也会引起一些问题。在Java5之前，可以通过volatile、synchronized关键字来解决并发访问的安全问题，但这样太麻烦。

Java5之后，专门提供了用来进行单变量多线程并发安全访问的工具包java.util.concurrent.atomic，其中的类也很简单。

下面给出一个反面例子（切勿模仿）：

import java.util.concurrent.ExecutorService;   
import java.util.concurrent.Executors;   
import java.util.concurrent.atomic.AtomicLong;   
  
/\*\*   
\* Java线程：新特征-原子量   
\*/   
public class Test {   
        public static void main(String[] args) {   
                ExecutorService pool = Executors.newFixedThreadPool(2);   
                Runnable t1 = new MyRunnable("张三", 2000);   
                Runnable t2 = new MyRunnable("李四", 3600);   
                Runnable t3 = new MyRunnable("王五", 2700);   
                Runnable t4 = new MyRunnable("老张", 600);   
                Runnable t5 = new MyRunnable("老牛", 1300);   
                Runnable t6 = new MyRunnable("胖子", 800);   
                //执行各个线程   
                pool.execute(t1);   
                pool.execute(t2);   
                pool.execute(t3);   
                pool.execute(t4);   
                pool.execute(t5);   
                pool.execute(t6);   
                //关闭线程池   
                pool.shutdown();   
        }   
}   
  
class MyRunnable implements Runnable {   
        private static AtomicLong aLong = new AtomicLong(10000);        //原子量，每个线程都可以自由操作   
        private String name;                //操作人   
        private int x;                            //操作数额   
  
        MyRunnable(String name, int x) {   
                this.name = name;   
                this.x = x;   
        }   
  
        public void run() {   
                System.out.println(name + "执行了" + x + "，当前余额：" + aLong.addAndGet(x));   
        }   
}

运行结果：

李四执行了3600，当前余额：13600   
王五执行了2700，当前余额：16300   
老张执行了600，当前余额：16900   
老牛执行了1300，当前余额：18200   
胖子执行了800，当前余额：19000   
张三执行了2000，当前余额：21000   
  
Process finished with exit code 0

张三执行了2000，当前余额：12000   
王五执行了2700，当前余额：18300   
老张执行了600，当前余额：18900   
老牛执行了1300，当前余额：20200   
胖子执行了800，当前余额：21000   
李四执行了3600，当前余额：15600   
  
Process finished with exit code 0

张三执行了2000，当前余额：12000   
李四执行了3600，当前余额：15600   
老张执行了600，当前余额：18900   
老牛执行了1300，当前余额：20200   
胖子执行了800，当前余额：21000   
王五执行了2700，当前余额：18300   
  
Process finished with exit code 0

从运行结果可以看出，虽然使用了原子量，但是程序并发访问还是有问题，那究竟问题出在哪里了？

这里要注意的一点是，原子量虽然可以保证单个变量在某一个操作过程的安全，但无法保证你整个代码块，或者整个程序的安全性。因此，通常还应该使用锁等同步机制来控制整个程序的安全性。

下面是对这个错误修正：

import java.util.concurrent.ExecutorService;   
import java.util.concurrent.Executors;   
import java.util.concurrent.locks.Lock;   
import java.util.concurrent.locks.ReentrantLock;   
import java.util.concurrent.atomic.AtomicLong;   
  
/\*\*   
\* Java线程：新特征-原子量   
\*/   
public class Test {   
        public static void main(String[] args) {   
                ExecutorService pool = Executors.newFixedThreadPool(2);   
                Lock lock = new ReentrantLock(false);   
                Runnable t1 = new MyRunnable("张三", 2000,lock);   
                Runnable t2 = new MyRunnable("李四", 3600,lock);   
                Runnable t3 = new MyRunnable("王五", 2700,lock);   
                Runnable t4 = new MyRunnable("老张", 600,lock);   
                Runnable t5 = new MyRunnable("老牛", 1300,lock);   
                Runnable t6 = new MyRunnable("胖子", 800,lock);   
                //执行各个线程   
                pool.execute(t1);   
                pool.execute(t2);   
                pool.execute(t3);   
                pool.execute(t4);   
                pool.execute(t5);   
                pool.execute(t6);   
                //关闭线程池   
                pool.shutdown();   
        }   
}   
  
class MyRunnable implements Runnable {   
        private static AtomicLong aLong = new AtomicLong(10000);        //原子量，每个线程都可以自由操作   
        private String name;                //操作人   
        private int x;                            //操作数额   
        private Lock lock;   
  
        MyRunnable(String name, int x,Lock lock) {   
                this.name = name;   
                this.x = x;   
                this.lock = lock;   
        }   
  
        public void run() {   
                lock.lock();   
                System.out.println(name + "执行了" + x + "，当前余额：" + aLong.addAndGet(x));   
                lock.unlock();   
        }   
}

执行结果：

张三执行了2000，当前余额：12000   
王五执行了2700，当前余额：14700   
老张执行了600，当前余额：15300   
老牛执行了1300，当前余额：16600   
胖子执行了800，当前余额：17400   
李四执行了3600，当前余额：21000   
  
Process finished with exit code 0

这里使用了一个对象锁，来控制对并发代码的访问。不管运行多少次，执行次序如何，最终余额均为21000，这个结果是正确的。

有关原子量的用法很简单，关键是对原子量的认识，原子仅仅是保证变量操作的原子性，但整个程序还需要考虑线程安全的

## Java线程：新特征-障碍器

Java5中，添加了障碍器类，为了适应一种新的设计需求，比如一个大型的任务，常常需要分配好多子任务去执行，只有当所有子任务都执行完成时候，才能执行主任务，这时候，就可以选择障碍器了。

障碍器是多线程并发控制的一种手段，用法很简单。下面给个例子：

import java.util.concurrent.BrokenBarrierException;   
import java.util.concurrent.CyclicBarrier;   
  
/\*\*   
\* Java线程：新特征-障碍器   
\*/   
public class Test {   
        public static void main(String[] args) {   
                //创建障碍器，并设置MainTask为所有定数量的线程都达到障碍点时候所要执行的任务(Runnable)   
                CyclicBarrier cb = new CyclicBarrier(7, new MainTask());   
                new SubTask("A", cb).start();   
                new SubTask("B", cb).start();   
                new SubTask("C", cb).start();   
                new SubTask("D", cb).start();   
                new SubTask("E", cb).start();   
                new SubTask("F", cb).start();   
                new SubTask("G", cb).start();   
        }   
}   
  
/\*\*   
\* 主任务   
\*/   
class MainTask implements Runnable {   
        public void run() {   
                System.out.println(">>>>主任务执行了！<<<<");   
        }   
}   
  
/\*\*   
\* 子任务   
\*/   
class SubTask extends Thread {   
        private String name;   
        private CyclicBarrier cb;   
  
        SubTask(String name, CyclicBarrier cb) {   
                this.name = name;   
                this.cb = cb;   
        }   
  
        public void run() {   
                System.out.println("[子任务" + name + "]开始执行了！");   
                for (int i = 0; i < 999999; i++) ;    //模拟耗时的任务   
                System.out.println("[子任务" + name + "]开始执行完成了，并通知障碍器已经完成！");   
                try {   
                        //通知障碍器已经完成   
                        cb.await();   
                } catch (InterruptedException e) {   
                        e.printStackTrace();   
                } catch (BrokenBarrierException e) {   
                        e.printStackTrace();   
                }   
        }   
}

运行结果：

[子任务E]开始执行了！   
[子任务E]开始执行完成了，并通知障碍器已经完成！   
[子任务F]开始执行了！   
[子任务G]开始执行了！   
[子任务F]开始执行完成了，并通知障碍器已经完成！   
[子任务G]开始执行完成了，并通知障碍器已经完成！   
[子任务C]开始执行了！   
[子任务B]开始执行了！   
[子任务C]开始执行完成了，并通知障碍器已经完成！   
[子任务D]开始执行了！   
[子任务A]开始执行了！   
[子任务D]开始执行完成了，并通知障碍器已经完成！   
[子任务B]开始执行完成了，并通知障碍器已经完成！   
[子任务A]开始执行完成了，并通知障碍器已经完成！   
>>>>主任务执行了！<<<<   
  
Process finished with exit code 0

从执行结果可以看出，所有子任务完成的时候，主任务执行了，达到了控制的目标

# 多线程编程注意事项

## Wait()必须放到循环内

Wait()方法的作用是使一个线程等待某个条件成立。它一定是在一个同步区域中被调用，而且该同步区域锁住了被调用的对象。一般来说，使用wait方法的标准模式如下：

|  |
| --- |
| Synchronized(obj){  While ( <condition does not hold> )  {  obj.wait();  }  … // perform action appropriate to condition  } |

1、一个对象锁可能用于保护多个状态变量，当它们都需要wait-notify操作时，如果不将wait放到while中就有问题。例如，某对象锁obj保护两种状态变量a和b，当a的条件断言不成立时发生了wait操作，当b的条件断言不成立时也发生了wait操作，两个线程被加入到obj对应的条件队列中，现在若改变状态变量a的某操作发生，在obj上调用了notifyAll操作，obj对应的条件队列里的所有线程均被唤醒，之前等待a的某个或几个线程去判断a的条件断言可能成立了，但b对应的条件断言肯定仍不成立，而此时等待b的线程也被唤醒了，所以需要循环判断b的条件断言是否满足，如果不满足，继续wait。  
2、多个线程wait的同一个状态的条件断言。如BlockingQueue场景下，当前队列是空的，多个线程要从里面取元素，于是都wait了。此时另一个线程往里面添加了一个元素，调用了notifyAll操作，唤醒了所有线程，但只有一个线程能拿到那个新加进来的元素，继续走下去，其它的仍需等待。  
3、虚假唤醒。在没有被通知、中断或超时的情况下，线程自动苏醒了。虽然这种情况在实践中很少发生，但是必须通过循环检测条件是否满足的方式来防止其发生，如果不满足该条件，则继续等待。

扩展一下，上面说到的都是内置锁，内置条件队列，与之对应的，有显式锁（Lock），显式条件队列（Lock#newCondition()）  
Lock#newCondition()返回一个Condition对象，该对象上对应于操作条件队列的wait和notify方法为：await、signal。与内置锁一样，要调用Condition的await、signal，需要锁定创建该Condition的Lock。如下代码形式：

|  |
| --- |
| package com.ticmy.concurrency;  import java.util.concurrent.locks.Condition;  import java.util.concurrent.locks.Lock;  import java.util.concurrent.locks.ReentrantLock;  public class TestWatiNotifyMechanism {  private static Lock lock = new ReentrantLock();  private static Condition condition = lock.newCondition();  public static void main(String[] args) throws Exception {          //无意义，仅测试用，勿模仿          lock.lock();          try {              condition.await();          } finally {              lock.unlock();          }      }  } |

|  |
| --- |
|  |

除了这个对应关系不一样，也需要在循环中await，需要说到的是，它与内置锁、内置条件队列的区别：  
1、内置锁对象只有一个条件队列，而显式锁可以通过newCondition方法创建多个条件队列，这样就可以避免不同的条件断言关联同一个条件队列造成的问题。  
2、如同Lock比内置锁更灵活一样，显式的条件队列也提供了更多的方法供调用（如等待的时候不可被中断的awaitUninterruptibly方法），更多方法参见java.util.concurrent.locks.Condition的JAVA API。  
3、Condition也有wait、notify方法，它们从Object类继承而来，一般实际中不会调用这些方法（要调用这些方法必须持有Condition对象的锁，而不是Lock的锁定）以避免混淆。  
4、Condition可以继承Lock的公平策略。如new ReentrantLock的时候传入的公平策略参数。当公平策略为true的时候，signal的时候，Condition中的线程唤醒顺序是FIFO的。

至于是选择显式的条件队列还是内置的，如同[内置锁和Lock](http://www.ticmy.com/?p=192)一样，取决于应用是否需要使用内置条件队列无法提供而显式条件队列提供了的特性。如果已使用了Lock，那么使用Condition是自然而然的事情

## 尽可能使用NotifyAll

notify操作有两个方法可用，nofity和notifyAll，顾名思义，前者每次唤醒一个线程，后者唤醒所有线程。当唤醒所有线程的时候，会增加上下文切换、锁竞争。但很多时候，使用notify是有风险的，多个线程在同一个条件队列里等待不同的条件断言成立，极可能本该唤醒的线程没唤醒。那么什么时候才能用notify呢？牛人们已经总结好了，需要满足以下两个条件：  
1、该对象的条件队列只关联了一个条件断言，且线程被唤醒后执行的代码逻辑是相同的；  
2、单进单出。一次notify（这里不是指notify方法）能唤醒的线程至多一个。

比如要实现一个类似开/关锁存器（在构造CountDownLatch的时候传入1）的功能，所有线程调用await()操作，最终某一线程调用countDown()操作，该countDown()操作就需要唤醒所有wait的线程。在这种场景下，第二条是不满足的——使用notify方法，其它线程将无法唤醒。

## 尽量不要使用ThreadGroup

使用ThreadGroup的一个经常的考虑是统计active thread的数目，但是正如java doc中ThreadGroup.**activeCount**()描述的，它的统计并不精确，同样enumerate方法也是不精确的。

|  |
| --- |
| public int **activeCount**()  返回此线程组中活动线程的估计数。结果并不能反映并发活动，并且可能受某些系统线程的存在状态的影响。  由于结果所固有的不精确特性，建议只将此方法用于信息目的 |

ThreadGroup API自身不一定是线程安全的；ThreadGroup提供不了太多有用信息，基本不需要使用了。要表示一组线程时，可以使用Executor Service线程池代替

## 尽量不要在synchronized代码块中调用sleep方法

正如在1.3.2和1.3.3中讨论的sleep方法和wait方法的使用，sleep是不释放对象锁的，如果在synchronized代码块中调用sleep方法，那么当前线程会hold住对象锁，使其它在该对象锁上等待的线程不能调度而一直等待sleep退出。所以尽量不要在synchronized代码块中调用sleep方法。

# JAVA线程中的设计模式

## Single Threaded Execution Pattern

## Immutable Pattern

## Guarded Suspension Pattern

## Balking Pattern

## Producer-Consumer Pattern

## Read-Write Lock Pattern

## Thread-Per-Message Pattern

## Worker Thread Pattern

## Future Pattern

## Two-Parse Termination Pattern

## Thread-Specific Storage Pattern

## Active Object Pattern

# 其它点滴知识：

## 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）。

## Volatile使用说明

在Java内存模型中，有main memory，每个线程也有自己的memory (例如寄存器)。为了性能，一个线程会在自己的memory中保持要访问的变量的副本。这样就会出现同一个变量在某个瞬间，在一个线程的memory中的值可能与另外一个线程memory中的值，或者main memory中的值不一致的情况

volatile关键字用于声明简单类型变量，如int、float、 boolean等数据类型。如果这些简单数据类型声明为volatile，那么这些变量的值都直接使用main memory中的值。

使用volatile关键字时要慎重，并不是只要简单类型变量使用volatile修饰，对这个变量的所有操作都是原来操作，当变量的值由自身的上一个决定时，如n=n+1、n++ 等，volatile关键字将失效，只有当变量的值和自身上一个值无关时对该变量的操作才是原子级别的，如n = m + 1，这个就是原级别的。所以在使用volatile关键时一定要谨慎，如果自己没有把握，可以使用synchronized来代替volatile