黑马程序员
itheima.com

非贷款，0元入学，不1万就业不给1分钱学费，我们已干四年了！

笔记总链接：http://bbs.itheima.com/thread-200600-1-1.html

5、多线程

5.3 线程间通信

5.3.1 线程间通信涉及的方法

多个线程在处理统一资源，但是任务却不同，这时候就需要线程间通信。

等待/唤醒机制涉及的方法：

1. wait()：让线程处于冻结状态，被wait的线程会被存储到线程池中。
2. notify()：唤醒线程池中的一个线程（任何一个都有可能）。
3. notifyAll()：唤醒线程池中的所有线程。

P.S.

1、这些方法都必须定义在同步中，因为这些方法是用于操作线程状态的方法。

2、必须要明确到底操作的是哪个锁上的线程！

3、wait和sleep区别？

1）wait可以指定时间也可以不指定。sleep必须指定时间。

2）在同步中时，对CPU的执行权和锁的处理不同。

wait：释放执行权，释放锁。

sleep：释放执行权，不释放锁。

为什么操作线程的方法wait、notify、notifyAll定义在了object类中，因为这些方法是监视器的方法，监视器其实就是锁。

锁可以是任意的对象，任意的对象调用的方式一定在object类中。

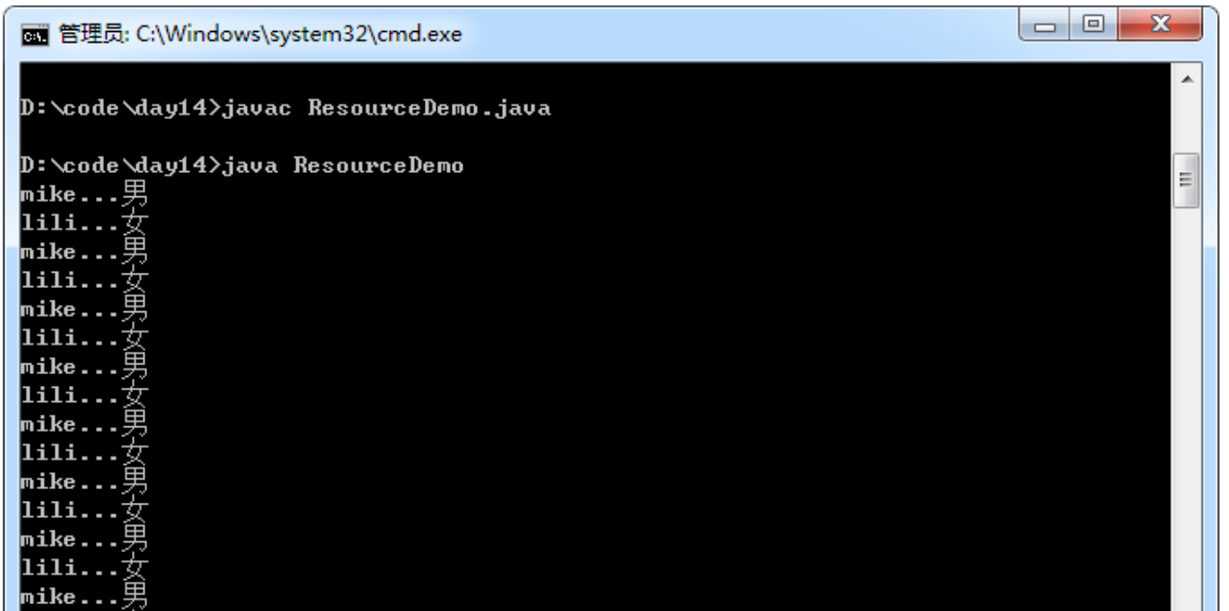
生产者-消费者问题：

```
class Resource{
        private String name ;
        private String sex ;
        private boolean flag = false;

        public synchronized void set(String name,String sex){
              if(flag )
                    try{
                          this.wait();
                    } catch(InterruptedException e){
                          e.printStackTrace();
                    }
              this.name = name;
              this.sex = sex;
              flag = true ;
              this.notify();
        }

        public synchronized void out(){
              if(!flag )
                    try{
                          this.wait();
                    } catch(InterruptedException e){
                          e.printStackTrace();
                    }
              System. out.println(name + "..." + sex);
              flag = false ;
              this.notify();
       }
}

//输入
class Input implements Runnable{
        Resource r;
        Input(Resource r){
              this.r = r;
        }

        public void run(){
              int x = 0;
              while(true ){
                    if(x == 0){
                          r.set( "mike","男" );
                    } else{
                          r.set( "lili","女" );
                    }
                    x = (x + 1)%2;
```

```
48.                 }
49.         }
50.   }
51.
52.   //输出
53.   class Output implements Runnable{
54.         Resource r;
55.
56.         Output(Resource r){
57.                this.r = r;
58.         }
59.
60.          public void run(){
61.                while(true ){
62.                      r.out();
63.                }
64.         }
65.   }
66.
67.   class ResourceDemo {
68.          public static void main(String[] args){
69.                //创建资源
70.                Resource r = new Resource();
71.                //创建任务
72.                Input in = new Input(r);
73.                Output out = new Output(r);
74.                //创建线程，执行路径
75.                Thread t1 = new Thread(in);
76.                Thread t2 = new Thread(out);
77.                //开启线程
78.                t1.start();
79.                t2.start();
80.         }
81.   }
```

复制代码

运行结果：



```
lili...女
mike...男
lili...女
```

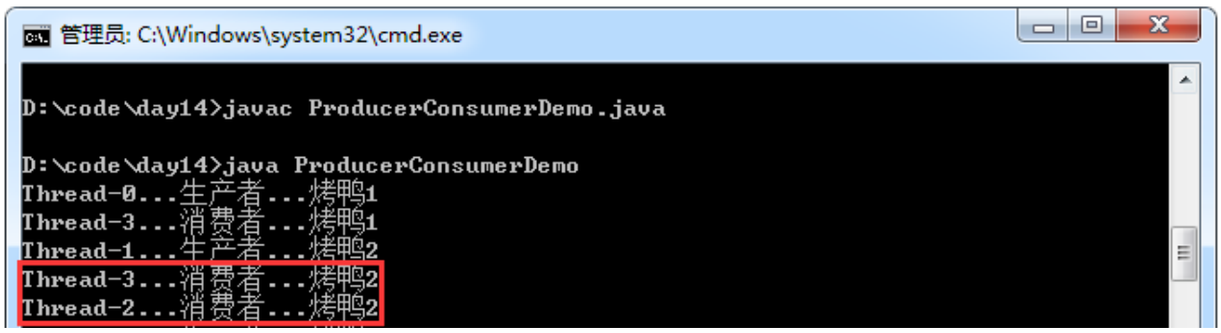
多生产者-多消费者问题：

```
class Resource{
        private String name ;
        private int count = 1;
        private boolean flag = false;

        public synchronized void set(String name){
              if(flag )
                    try{
                          wait();
                    } catch(InterruptedException e){
                          e.printStackTrace();
                    }
              this.name = name + count;
              count++;
              System.out.println(Thread.currentThread().getName() + "...生产者..." + this. name);
              flag = true ;
             notify();
        }

        public synchronized void out(){
             if(!flag )
                    try{
                          wait();
                    } catch(InterruptedException e){
                          e.printStackTrace();
                    }
             flag = false ;
             notify();
             System.out.println(Thread.currentThread().getName() + "...消费者..." + this. name);
       }
}

class Producer implements Runnable{
       private Resource r ;
       Producer(Resource r){
              this.r = r;
       }
        public void run(){
              while(true ){
                    r.set( "烤鸭");
             }
       }
}
```

```
class Consumer implements Runnable{
        private Resource r ;
        Consumer(Resource r){
               this.r = r;
        }
         public void run(){
               while(true ){
                     r.out();
              }
        }
}

class ProducerConsumerDemo {
         public static void main(String[] args){
              Resource r = new Resource();
              Producer pro = new Producer(r);
              Consumer con = new Consumer(r);

              Thread t0 = new Thread(pro);
              Thread t1 = new Thread(pro);
              Thread t2 = new Thread(con);
              Thread t3 = new Thread(con);
              t0.start();
              t1.start();
              t2.start();
              t3.start();
        }
}
```

复制代码

运行结果：以上代码存在安全问题。



原因分析：

得到以上结果的过程分析如下：

1. 线程Thread-0获取到CPU执行权及锁，生产了烤鸭3298，将flag设置为true。然后，Thread-0又重新获取到CPU执行权，由于flag为true，故执行wait方法，阻塞。Thread-1接着获取到CPU执行权，由于flag为true，故执行wait方法，也阻塞。

```
public synchronized void set(String name){
        if (flag)
                try {
                        wait();
                } catch (InterruptedException e){
```

```
                e.printStackTrace();
            }
        this .name = name + count;
        count ++;
        System. out .println(Thread.currentThread().getName() + "...生产者..." + this. name );
        flag = true ;
        notify();
}
```

2. 线程Thread-3获取到CPU执行权及锁，消费了烤鸭3298，将flag设置为false。然后，线程Thread-0被唤醒，但是并没有获取到锁，而是线程Thread-3接着获取到CPU执行权及锁，然而此时flag为false，所以Thread-3阻塞。下面线程Thread-2接着获取到CPU执行权及锁，然而此时flag为false，所以Thread-2也阻塞。

```
public synchronized void out(){
        if (!flag)
            try {
                wait();
            } catch (InterruptedException e){
                e.printStackTrace();
            }
        flag = false ;
        notify();
        System. out .println(Thread.currentThread().getName() + "...消费者..." + this. name );
}
```

3. 线程Thread-0获取到CPU执行权及锁，不需要if语句判断，直接生产烤鸭3299，然后又唤醒线程Thread-1获取到CPU执行权及锁，不需要if语句判断，直接生产烤鸭3300。从而造成了烤鸭3299还没有被消费，就直接生产了烤鸭3300的情况。

由于if判断标记，只有一次，会导致不该运行的线程运行了，出现了数据错误的情况。故修改成while判断标记，线程获取CPU执行权及锁后，将重新判断是否具备运行条件。

notify方法只能唤醒一个线程，如果本方唤醒了本方，没有意义。而且while判断标记+notify会导致死锁。notifyAll解决了本方线程一定会唤醒对方线程的问题。

P.S.

while判断标记+notify会导致死锁的示例：

如果将上面的代码中的if判断标记修改成wile判断标记，就会出现死锁的现象，前2步与原来是一致的。第3步如下：

3. 线程Thread-0获取到CPU执行权及锁，通过了while语句判断，直接生产烤鸭3299，将flag设置为true。然后又唤醒线程Thread-1获取到CPU执行权及锁，没有通过while语句判断，阻塞。线程Thread-0又获取到CPU执行权及锁，通不过while语句判断，也阻塞，此时Thread-0、1、2、3都阻塞，故死锁。

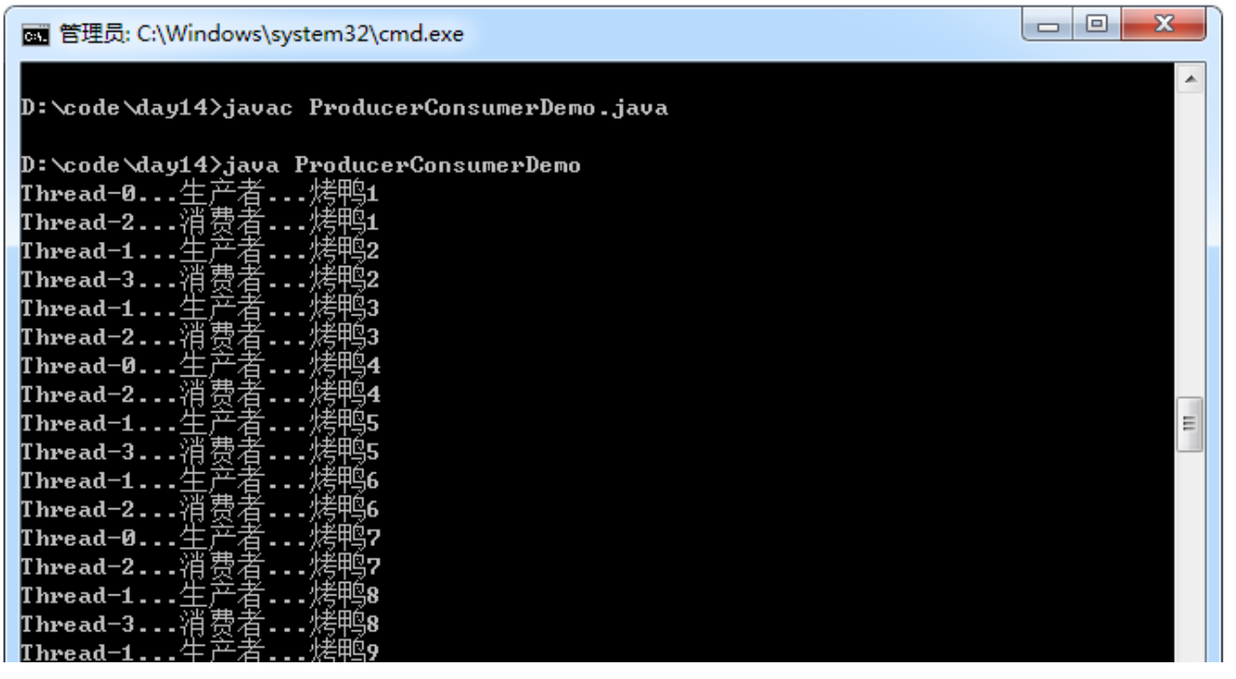
代码：

```
class Resource{
       private String name ;
       private int count = 1;
       private boolean flag = false;

       public synchronized void set(String name){
            while(flag )
                  try{
                        this.wait();
                 } catch(InterruptedException e){
                       e.printStackTrace();
                 }
             this.name = name + count;
             count++;
             System.out.println(Thread.currentThread().getName() + "...生产者..." + this. name);
             flag = true ;
             notifyAll();
       }

       public synchronized void out(){
             while(!flag )
                  try{
                        this.wait();
                 } catch(InterruptedException e){
                       e.printStackTrace();
                 }
            flag = false ;
            notifyAll();
            System.out.println(Thread.currentThread().getName() + "...消费者..." + this. name);
       }
}

class Producer implements Runnable{
       private Resource r ;
       Producer(Resource r){
             this.r = r;
       }
       public void run(){
             while(true ){
                  r.set( "烤鸭");
            }
       }
}

class Consumer implements Runnable{
       private Resource r ;
       Consumer(Resource r){
             this.r = r;
       }
       public void run(){
             while(true ){
```

```
                    r.out();
                }
        }
    }

    class ProducerConsumerDemo {
        public static void main(String[] args){
            Resource r = new Resource();
            Producer pro = new Producer(r);
            Consumer con = new Consumer(r);

            Thread t0 = new Thread(pro);
            Thread t1 = new Thread(pro);
            Thread t2 = new Thread(con);
            Thread t3 = new Thread(con);
            t0.start();
            t1.start();
            t2.start();
            t3.start();
        }
    }
```

复制代码

运行结果：



5.3.2 JDK1.5新特性

同步代码块就是对于锁的操作是隐式的。

JDK1.5以后将同步和锁封装成了对象，并将操作锁的隐式方式定义到了该对象中，将隐式动作变成了显示动作。

Lock接口：出现替代了同步代码块或者同步函数，将同步的隐式操作变成显示锁操作。同时更为灵活，可以一个锁上加上多组监视器。

lock()：获取锁。

unlock()：释放锁，为了防止异常出现，导致锁无法被关闭，所以锁的关闭动作要放在finally中。

Condition接口：出现替代了Object中的wait、notify、notifyAll方法。将这些监视器方法单独进行了封装，变成Condition监视器对象，可以任意锁进行组合。

Condition接口中的await方法对应于Object中的wait方法。

Condition接口中的signal方法对应于Object中的notify方法。

Condition接口中的signalAll方法对应于Object中的notifyAll方法。

使用一个Lock、一个Condition修改上面的多生产者-多消费者问题。

代码：

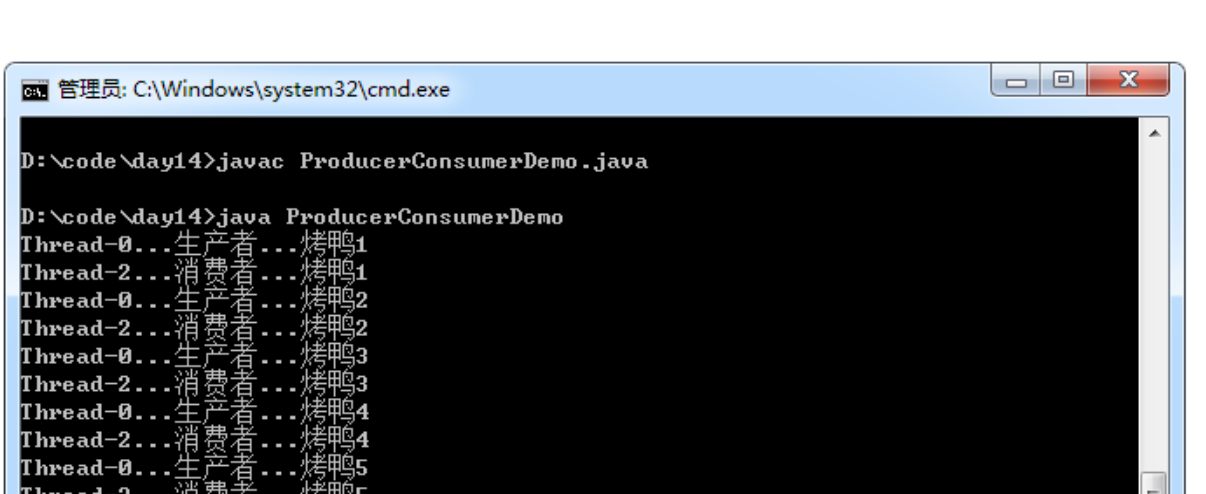
```
import java.util.concurrent.locks.*;
class Resource{
    private String name ;
    private int count = 1;
    private boolean flag = false;

   //创建一个锁对象
   Lock lock = new ReentrantLock();

    //通过已有的锁获取该锁上的监视器对象
   Condition con = lock .newCondition();

    public void set(String name){
        lock.lock();
        try{
            while(flag )
                try{
                    con.await();
                } catch(InterruptedException e){
                    e.printStackTrace();
                }
            this.name = name + count;
            count++;
            System.out.println(Thread.currentThread().getName() + "...生产者..." + this.name);
            flag = true ;
            con.signalAll();
        }finally{
            lock.unlock();
        }
    }

    public void out(){
```

```
            lock.lock();
             try{
                 while(!flag )
                     try{
                         con.await();
                     } catch(InterruptedException e){
                         e.printStackTrace();
                     }
                 flag = false ;
                 con.signalAll();
                 System.out.println(Thread.currentThread().getName() + "...消费者..." + this.name);
            }finally{
                 lock.unlock();
            }
        }
    }

    class Producer implements Runnable{
        private Resource r ;
        Producer(Resource r){
            this.r = r;
        }
        public void run(){
            while(true ){
                r.set( "烤鸭");
            }
        }
    }

    class Consumer implements Runnable{
        private Resource r ;
        Consumer(Resource r){
            this.r = r;
        }
        public void run(){
            while(true ){
                r.out();
            }
        }
    }

    class ProducerConsumerDemo {
        public static void main(String[] args){
            Resource r = new Resource();
            Producer pro = new Producer(r);
            Consumer con = new Consumer(r);

            Thread t0 = new Thread(pro);
```

```
        Thread t1 = new Thread(pro);
        Thread t2 = new Thread(con);
        Thread t3 = new Thread(con);
        t0.start();
        t1.start();
        t2.start();
        t3.start();
    }
}
```

复制代码

运行结果：

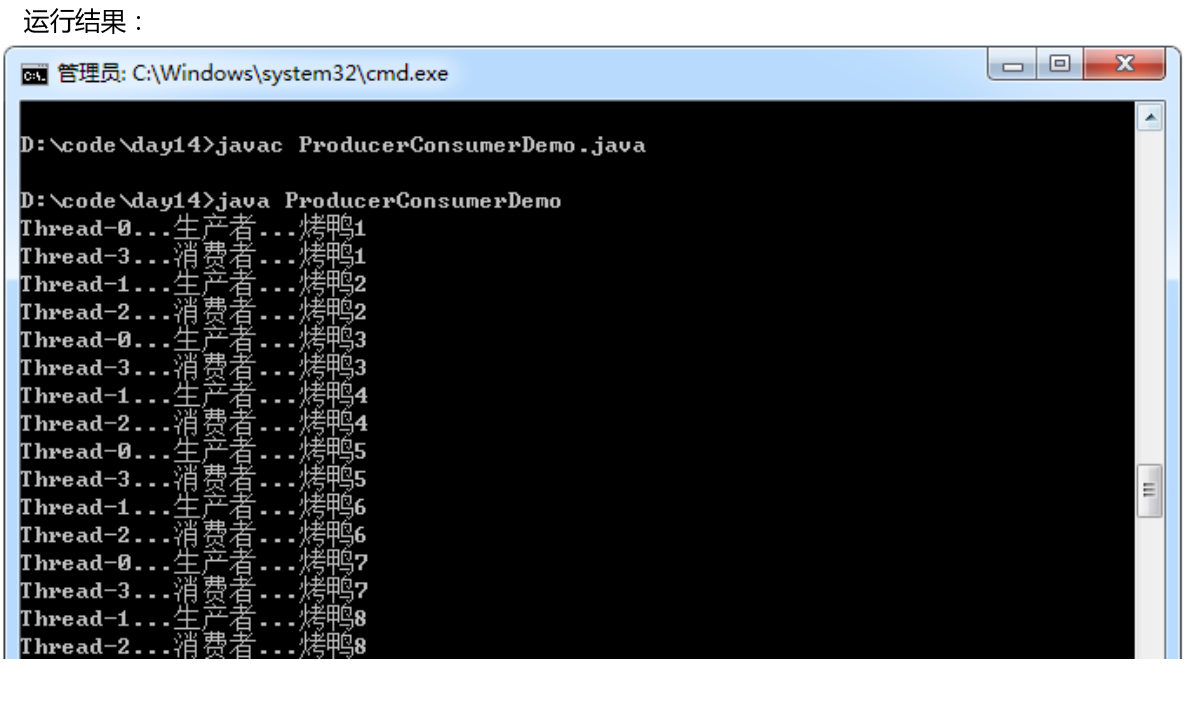


使用一个Lock、两个Condition修改上面的多生产者-多消费者问题。

代码：

```java
import java.util.concurrent.locks.*;
class Resource{
       private String name ;
       private int count = 1;
       private boolean flag = false;

       //创建一个锁对象
       Lock lock = new ReentrantLock();

       //通过已有的锁获取该锁上的监视器对象
       Condition con = lock .newCondition();

       //通过已有的锁获取两组监视器，一组监视生产者，一组监视消费者
       Condition producer_con = lock .newCondition();
       Condition consumer_con = lock .newCondition();

       public void set(String name){
             lock.lock();
             try{
                   while(flag )
                         try{
                              producer_con.await();
                        } catch(InterruptedException e){
                              e.printStackTrace();
                        }
                   this.name = name + count;
                   count++;
                   System.out.println(Thread.currentThread().getName() + "...生产者..." + this. name);
                   flag = true ;
                   consumer_con.signal();
            } finally{
                   lock.unlock();
            }
      }

       public void out(){
             lock.lock();
             try{
                   while(!flag )
                         try{
                               consumer_con.await();
                        } catch(InterruptedException e){
                              e.printStackTrace();
                        }
                   flag = false ;
                   producer_con.signal();
                   System.out.println(Thread.currentThread().getName() + "...消费者..." + this. name);
            } finally{
                   lock.unlock();
            }
      }
}

class Producer implements Runnable{
       private Resource r ;
       Producer(Resource r){
             this.r = r;
       }
       public void run(){
             while(true ){
                   r.set( "烤鸭");
             }
      }
}

class Consumer implements Runnable{
       private Resource r ;
       Consumer(Resource r){
             this.r = r;
       }
       public void run(){
             while(true ){
                   r.out();
             }
      }
}

class ProducerConsumerDemo {
       public static void main(String[] args){
            Resource r = new Resource();
            Producer pro = new Producer(r);
            Consumer con = new Consumer(r);

            Thread t0 = new Thread(pro);
            Thread t1 = new Thread(pro);
            Thread t2 = new Thread(con);
            Thread t3 = new Thread(con);
            t0.start();
            t1.start();
            t2.start();
            t3.start();
      }
}
```
复制代码

运行结果：



利用数组解决生产者-消费者问题。

代码：

```java
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

class BoundedBuffer {
   final Lock lock = new ReentrantLock();
   final Condition notFull  = lock .newCondition();
   final Condition notEmpty = lock .newCondition();

   final Object[] items = new Object[100];
   int putptr, takeptr , count ;

   public void put(Object x) throws InterruptedException {
     lock.lock();
     try {
       while (count == items.length)
         notFull.await();
         items[putptr ] = x;
         if (++putptr == items.length) putptr = 0;
         ++ count;
         notEmpty.signal();
      } finally {
        lock.unlock();
     }
   }

   public Object take() throws InterruptedException {
      lock.lock();
      try {
       while (count == 0)
         notEmpty.await();
       Object x = items[takeptr];
       if (++takeptr == items.length) takeptr = 0;
       -- count;
       notFull.signal();
       return x;
     } finally {
       lock.unlock();
     }
   }
```
复制代码

5.3.3 停止线程

怎么控制线程的任务结束呢？

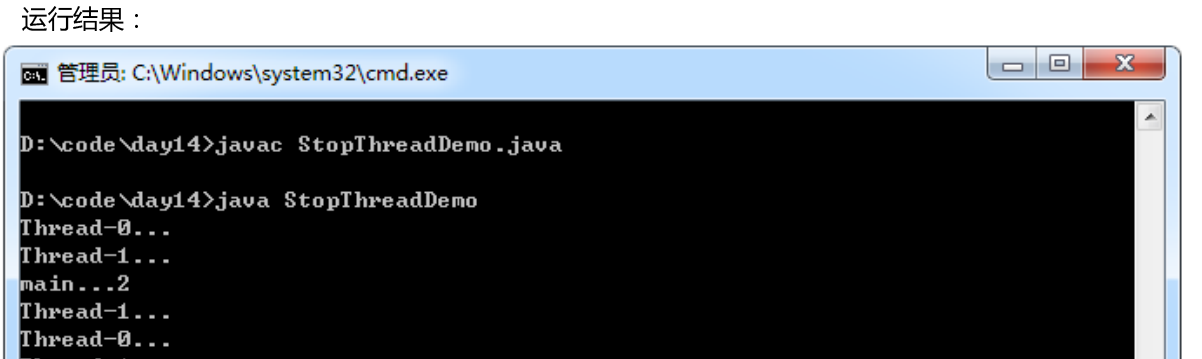
任务中都会有循环结构，只要控制住循环就可以结束任务。

控制循环通常就用定义标记来完成。

示例：

```java
class StopThread implements Runnable{
       private boolean flag = true;
       public void run(){
             while(flag ){
                  System. out.println(Thread.currentThread().getName() + "...");
            }
      }
       public void setFlag(){
             flag = false ;
      }
}

class StopThreadDemo{
       public static void main(String[] args){
            StopThread st = new StopThread();

            Thread t1 = new Thread(st);
            Thread t2 = new Thread(st);

            t1.start();
            t2.start();

             int num = 1;
             for(;;){
                   if(++num == 50){
                        st.setFlag();
                         break;
                  }
                  System. out.println("main..." + num);
            }
             System. out.println("over" );
      }
}
```
复制代码

运行结果：



......

P.S.

也可以使用stop方法停止线程，不过已经过时，不再使用。

但是如果线程处于了冻结状态，无法读取标记，如何结束呢？

可以使用interrupt()方法将线程从冻结状态强制恢复到运行状态中来，让线程具备CPU的执行资格。强制动作会发生InterruptedException，一定要记得处理。

API文档叙述如下：

interrupt

public void interrupt()

中断线程。

如果当前线程没有中断它自己（这在任何情况下都是允许的），则该线程的 checkAccess 方法就会被调用，这可能抛出 SecurityException。

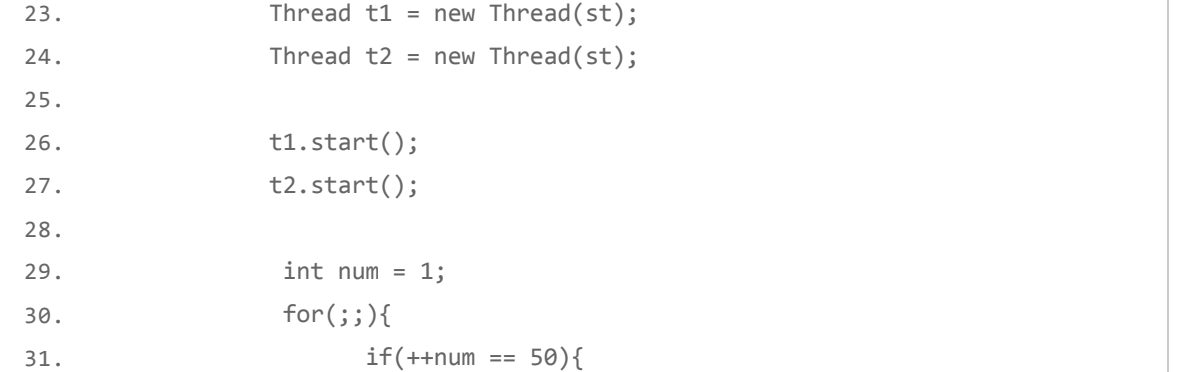
如果线程在调用 Object 类的 wait()、wait(long) 或 wait(long, int) 方法，或者该类的 join()、join(long)、join(long, int)、sleep(long) 或 sleep(long, int) 方法过程中受阻，则其中断状态将被清除，它还将收到一个 InterruptedException。

如果该线程在可中断的通道上的 I/O 操作中受阻，则该通道将被关闭，该线程的中断状态将被设置并且该线程将收到一个 ClosedByInterruptException。

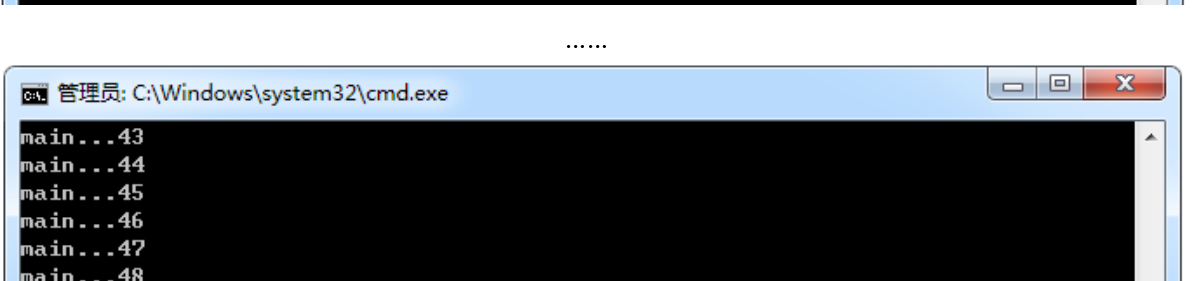
示例：

```java
class StopThread implements Runnable{
       private boolean flag = true;
       public synchronized void run(){
             while(flag){
                   try{
                        wait();
                  } catch(InterruptedException e){
                        System.out.println(Thread.currentThread().getName() + "..." + e);
                        flag = false;
                  }
                  System.out.println(Thread.currentThread().getName() + "......");
            }
      }
       public void setFlag(){
            flag = false;
      }
}

class StopThreadDemo{
       public static void main(String[] args){
            StopThread st = new StopThread();

            Thread t1 = new Thread(st);
            Thread t2 = new Thread(st);

            t1.start();
            t2.start();

             int num = 1;
             for(;;){
                   if(++num == 50){
                        t1.interrupt();
                        t2.interrupt();
                         break;
                  }
                  System.out.println( "main..." + num);
            }
            System.out.println( "over");
      }
}
```
复制代码

运行结果：



......



5.3.4 线程类的其他方法

setDaemon方法：

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。

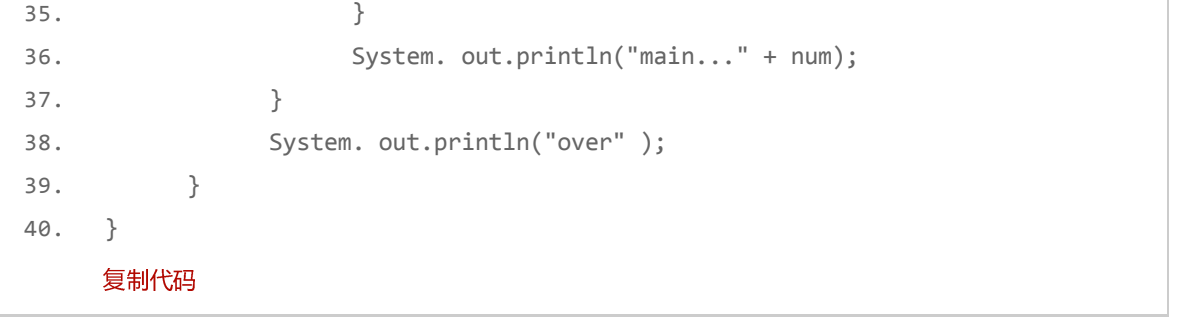
该方法必须在启动线程前调用。

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）。

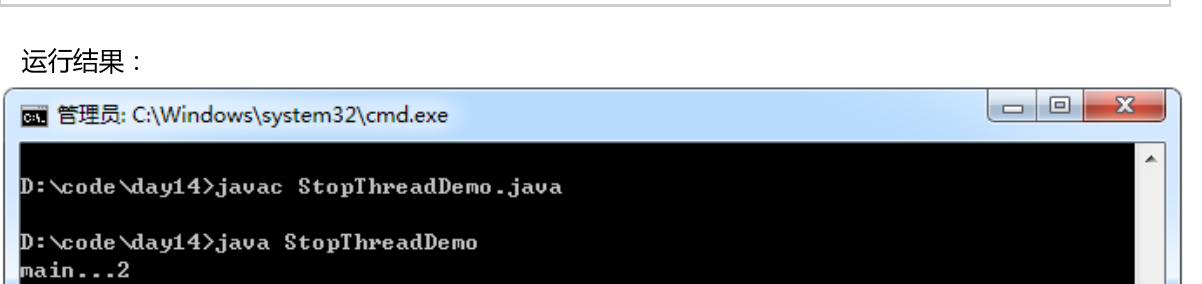
示例：

```java
class StopThread implements Runnable{
       private boolean flag = true;
       public synchronid void run(){
             while(flag ){
                   try{
                        wait();
                  } catch(InterruptedException e){
                        System. out.println(Thread.currentThread().getName() + "..." + e);
                         flag = false ;
                  }
                  System. out.println(Thread.currentThread().getName() + "......");
            }
      }
       public void setFlag(){
             flag = false ;
      }
}

class StopThreadDemo{
       public static void main(String[] args){
            StopThread st = new StopThread();

            Thread t1 = new Thread(st);
            Thread t2 = new Thread(st);

            t1.start();
            t2.setDaemon( true);
            t2.start();

             int num = 1;
             for(;;){
                   if(++num == 50){
                        t1.interrupt();
                         break;
                  }
                  System. out.println("main..." + num);
            }
            System. out.println("over" );
      }
}
```
复制代码

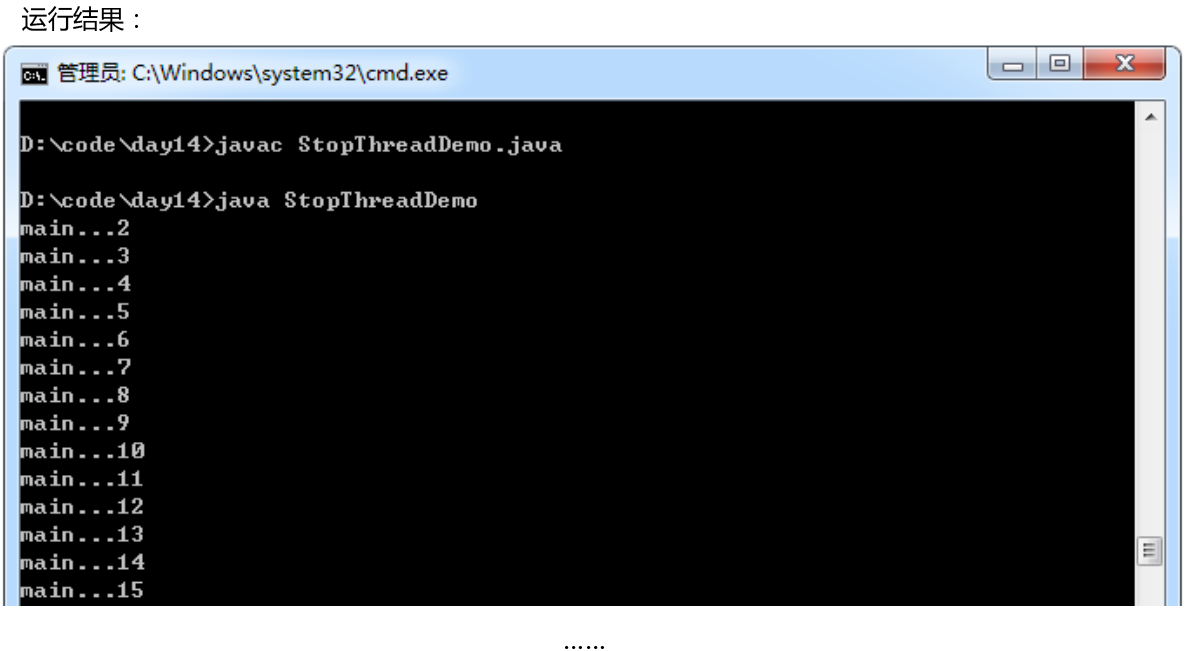
运行结果：



......



join方法：



join

public final void join()
        throws InterruptedException

等待该线程终止。

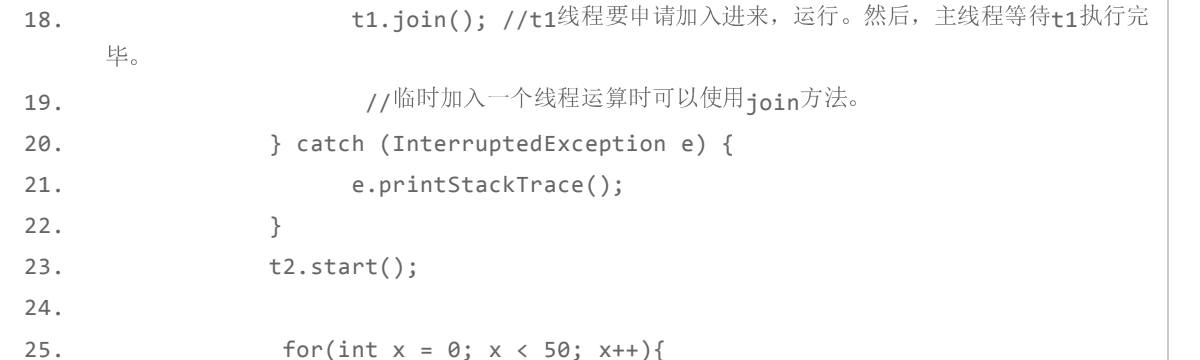
抛出：

InterruptedException - 如果任何线程中断了当前线程。当抛出该异常时，当前线程的*中断状态*被清除。

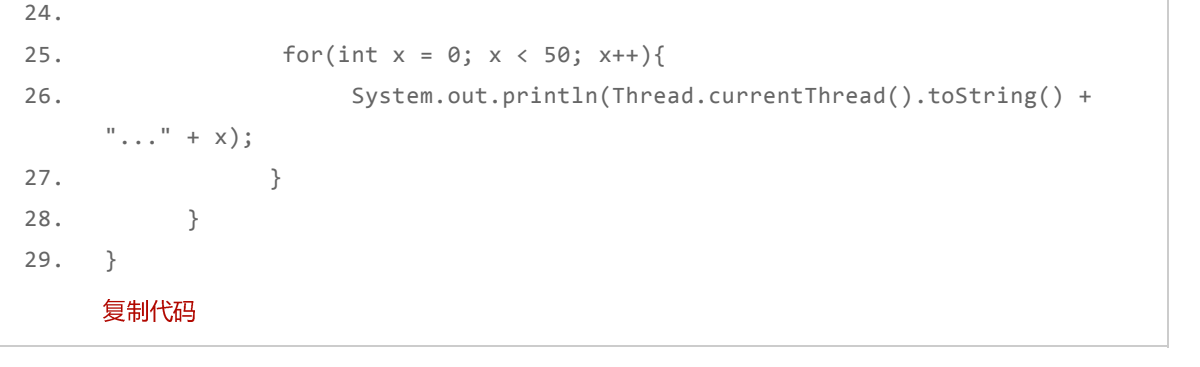
示例：

```java
class Demo implements Runnable{
       public void run(){
             for(int x = 0; x < 50; x++){
                  System. out.println(Thread.currentThread().getName() + "..." + x);
            }
      }
}

class JoinDemo{
       public static void main(String[] args){
            Demo d = new Demo();

            Thread t1 = new Thread(d);
            Thread t2 = new Thread(d);

            t1.start();
             try {
                  t1.join(); //t1线程要申请加入进来，运行。然后，主线程等待t1执行完毕。
                   //临时加入一个线程运算时可以使用join方法。
            } catch (InterruptedException e) {
                  e.printStackTrace();
            }
            t2.start();

             for(int x = 0; x < 50; x++){
                  System.out.println(Thread.currentThread().toString() + "..." + x);
            }
      }
}
```
复制代码

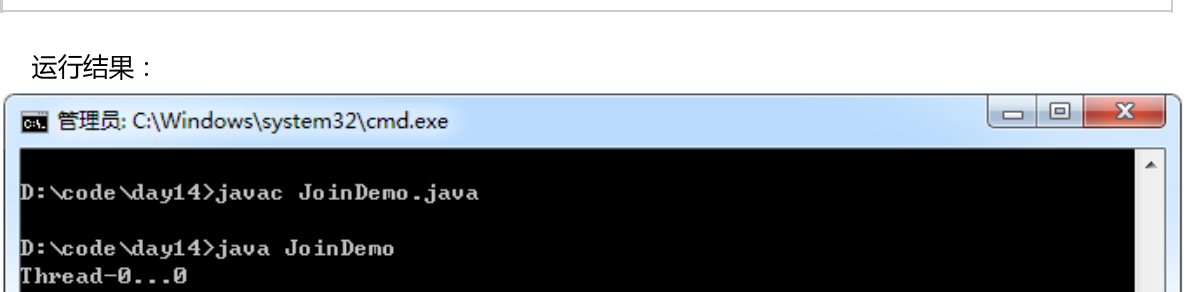
运行结果：



......



setPriority方法：



setPriority

public final void setPriority(int newPriority)

更改线程的优先级。

首先调用线程的 checkAccess 方法，且不带任何参数。这可能抛出 SecurityException。

在其他情况下，线程优先级被设定为指定的 newPriority 和该线程的线程组的最大允许优先级相比较小的一个。

参数：

newPriority - 要为线程设定的优先级

toString方法：

toString

public String **toString**()

返回该线程的字符串表示形式，包括线程名称、优先级和线程组。

覆盖：
类 Object 中的 toString

返回：
该线程的字符串表示形式。

yield方法：

yield

public static void **yield**()

暂停当前正在执行的线程对象，并执行其他线程。

示例：

```
class JoinDemo{
        public static void main(String[] args){
             Demo d = new Demo();

             Thread t1 = new Thread(d);
             Thread t2 = new Thread(d);

             t1.start();
             t2.start();
             t2.setPriority(Thread. MAX_PRIORITY);

              for(int x = 0; x < 50; x++){
                   System. out.println(Thread.currentThread().toString() + "..." + x);
             }
        }
}

class Demo implements Runnable{
        public void run(){
              for(int x = 0; x < 50; x++){
                   System. out.println(Thread.currentThread().getName() + "..." + x);
                   Thread. yield();//释放执行权
             }
        }
}
```

复制代码

运行结果：

```
管理员: C:\Windows\system32\cmd.exe

D:\code\day14>javac JoinDemo.java

D:\code\day14>java JoinDemo
Thread-0...0
Thread[main,5,main]...0
Thread-1...0
Thread[main,5,main]...1
Thread-0...1
Thread[main,5,main]...2
Thread-1...1
Thread[main,5,main]...3
Thread-0...2
Thread[main,5,main]...4
Thread-1...2
Thread[main,5,main]...5
Thread-0...3
Thread[main,5,main]...6
Thread-1...3
```

.....

```
管理员: C:\Windows\system32\cmd.exe
Thread-1...22
Thread[main,5,main]...45
Thread-0...23
Thread[main,5,main]...46
Thread-1...23
Thread[main,5,main]...47
Thread-0...24
Thread[main,5,main]...48
Thread-1...24
Thread[main,5,main]...49
Thread-0...25
Thread-1...25
Thread-0...26
Thread-1...26
Thread-0...27
```

.....

```
管理员: C:\Windows\system32\cmd.exe
Thread-0...44
Thread-1...44
Thread-0...45
Thread-1...45
Thread-0...46
Thread-1...46
Thread-0...47
Thread-1...47
Thread-0...48
Thread-1...48
Thread-0...49
Thread-1...49
```

~END~

Love
Shanghai
AURORA
Expo