多线程

笔记本:	JAVA		
创建时间:	2018/8/12 9:48	**更新时间:**	2018/10/23 22:03
作者:	Debao Xu		

进程:
就是应用程序在内存中分配的空间，即正在进行中的程序(直译)。

线程:
举例：运动场、鸟巢、360.
跑步的，扔标枪的，跳远的，.....
“体检”，“杀毒”，“清理”，.....等同时执行
jvm中的多线程：
至少两个线程：一个是负责自定义代码运行的（这个从main方法开始执行的线程称之为主线程）；一个是负责垃圾回收的。

多线程**好处**：解决了多部分同时运行的问题，合理的使用CPU的资源
多线程的**弊端**：多线程的运行根据CPU的切换完成的，至于怎么切换由CPU说了算，所以多线程运行有一个随机性（CPU的快速切换造成的）。CPU以**时间片**方式访问，线程“开多了”，线程之间的访问间隔时间就会过长，从而使效率下降了。

每个线程都有运行的代码内容，这个称之为线程的任务，之所以创建一个线程就是为了去运行制定的任务代码。而线程的任务都封装在特定的区域中，比如：主线程运行的任务都定义在main方法中，**垃圾回收线程在回收垃圾时都会运行finalize方法。**

创建线程
如何创建一个线程呢?
创建线程方式一：继承Thread类。
步骤:
1，定义一个类继承Thread类。
2，覆盖Thread类中的run方法。
3，直接**创建Thread的子类对象创建线程**。
4，调用start方法开启线程并调用线程的任务run方法执行。【可以通过Thread的getName获取线程的名称 Thread-编号(从0开始)】

```
class Demo extends Thread //继承类Thread
{
	private String name;
	Demo(String name)
	{
		this.name = name;
	}
	public void run()  //覆盖Thread中的run方法
	{
		for(int i = 0;i<10;i++)
		{
			System.out.println(name+"..."+i);
		}
	}
}
public class threadDemo
{
	public static void main(String[] args)
	{
		Demo d1 = new Demo("zhang");
		Demo d2 = new Demo("xu");
		d1.start(); //调用start方法开启线程并调用线程的任务run方法执行。
		d2.start(); //调用start方法开启线程并调用线程的任务run方法执行。
```

```
            for(int i = 0;i<20;i++) // 主线程
            {
                  System.out.println("main....."+i);
            }
      }
}
```

第一次运行结果：

```
main.....0
zhang...0
zhang...1
xu...0
zhang...2
zhang...3
zhang...4
zhang...5
zhang...6
zhang...7
zhang...8
zhang...9
main.....1
main.....2
xu...1
main.....3
xu...2
xu...3
xu...4
xu...5
xu...6
xu...7
main.....4
main.....5
main.....6
main.....7
main.....8
main.....9
main.....10
main.....11
main.....12
main.....13
xu...8
xu...9
main.....14
main.....15
main.....16
main.....17
main.....18
main.....19
```

第二次运行结果

```
main.....0
main.....1
main.....2
xu...0
xu...1
xu...2
zhang...0
xu...3
zhang...1
zhang...2
zhang...3
zhang...4
zhang...5
zhang...6
zhang...7
main.....3
main.....4
main.....5
main.....6
main.....7
main.....8
main.....9
zhang...8
zhang...9
xu...4
xu...5
xu...6
xu...7
xu...8
main.....10
main.....11
main.....12
xu...9
main.....13
main.....14
main.....15
main.....16
main.....17
main.....18
main.....19
```

上面的代码中开启了三条线程，分别是d1、d2、主线程，每次打印结果不一样，体现了多线程的随机性

说明

创建线程的目的是为了开启一条执行路径，去运行指定的代码和其他代码实现同时运行。而运行的指定代码就是这个执行路径的任务。

jvm创建的主线程的任务都定义在了主函数中。而自定义的线程它的任务在哪儿呢?
这个任务就通过Thread类中的run方法来体现。也就是说，run方法就是封装自定义线程运行任务的函数。**run方法中定义就是线程要运行的任务代码**。开启线程是为了运行指定代码，所以只有继承Thread类，并复写run方法（将运行的代码定义在run方法中即可）。

获取线程的名称

对于非主线程，可以直接用getName()来获取线程的名称；而对于主线程，由于没有继承Thread类，因此不能直接用getName()方法，我们可以先获取到当前线程，而在Thread类中就有可以获取到当前线程的方法，currentThread的作用是：哪一个线程运行了，就能获取到哪一个线程，然后再通过调用线程的getName方法就能获取到线程的名称。此外，currentThread是静态方法，因此可以不创建对象就能访问（直接用类名调用）。即"Thread.*currentThread*().getName()" **是先返回得到当前线程，再得到当前线程的名称**。

```
class Demo extends Thread //继承类Thread
{
      private String name;
      Demo(String name)
      {
            this.name = name;
      }
      public void run()  //覆盖Thread中的run方法
      {
            for(int i = 0;i<10;i++)
            {
                  //此处的getName()方法就是Thread中的方法，作用是：返回线程的名称
```

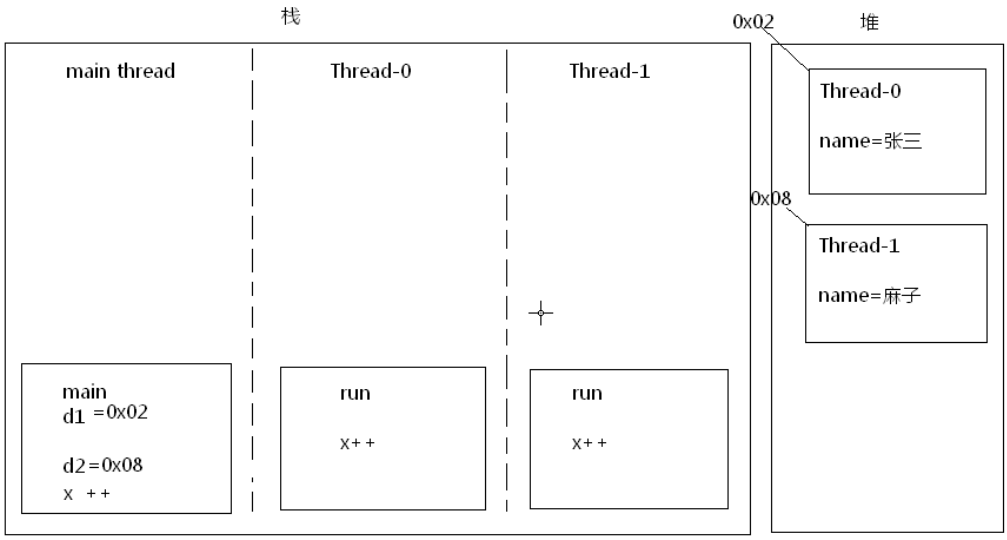
```
                    System.out.println(getName()+"..."+name+"..."+i);
            }
        }
}
public class threadDemo
{
        public static void main(String[] args)
        {
                Demo d1 = new Demo("zhang");
                Demo d2 = new Demo("xu");
                d1.start(); //start方法的两个作用：1、开启线程，2、调用run方法
                d2.start(); //调用start方法开启线程并调用线程的任务run方法执行。

                /* 如果将上面的
                   d1.start();
                   d2.start();
                   改为:
                   d1.run();
                   d2.run();
                   问有什么区别?
                   答：调用start会开启线程，并让开启的线程去执行run方法中的线程任务；而直接调用run方 法，则线程并未开启，去执行run方法的只有主线程
                   */
                for(int i = 0;i<20;i++) // 主线程
                {
                        //主线程可以用Thread中的currentThread来获取到
                        System.out.println(Thread.currentThread().getName()+"......"+i);
                }
        }
}
```

运行结果：

```
Thread-1...xu...0
Thread-0...zhang...0
Thread-0...zhang...1
Thread-1...xu...1
main.....1
Thread-1...xu...2
Thread-1...xu...3
Thread-0...zhang...2
Thread-1...xu...4
Thread-1...xu...5
Thread-1...xu...6
main.....2
Thread-1...xu...7
Thread-1...xu...8
Thread-1...xu...9
Thread-0...zhang...3
main.....3
main.....4
main.....5
main.....6
main.....7
main.....8
main.....9
main.....10
main.....11
main.....12
main.....13
main.....14
Thread-0...zhang...4
main.....15
main.....16
main.....17
main.....18
main.....19
Thread-0...zhang...5
Thread-0...zhang...6
Thread-0...zhang...7
Thread-0...zhang...8
Thread-0...zhang...9
```

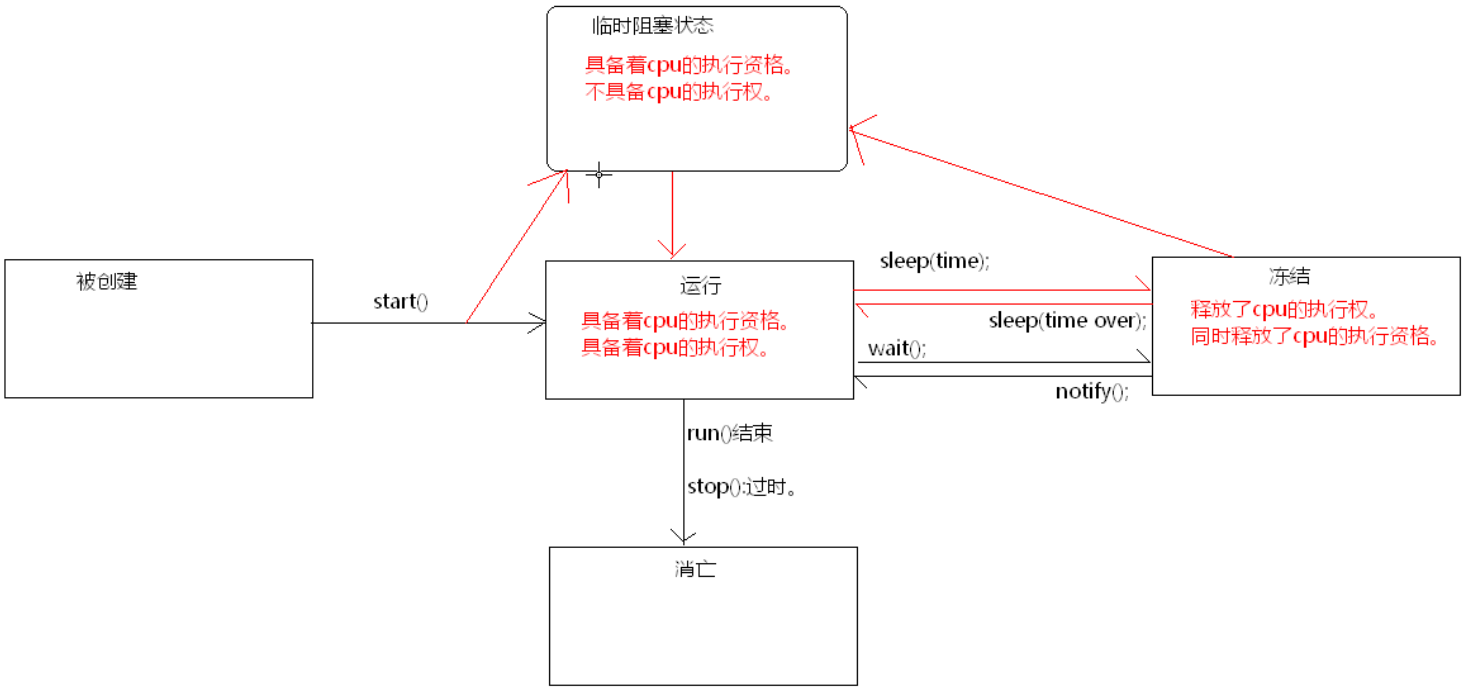
多线程内存图

```
class Demo extends Thread
{
   public void run()
   {
      for(int x=1; x<=40; x++)
      {
        sop(this.getName());
        sop(Thread.currentThread().getName());
      }
   }

}
class  ThreadDemo
{
    public static void main(String[] args)
    {
        Demo d1 = new Demo("张三");
        Demo d2 = new Demo("麻子");
        d1.start();
        d2.start();
        for(int x=1; x<40; x++)
        {

        }
    }
}
            java ThreadDemo
```



多线程的运行状态

多线程有5种状态：被创建、运行、临时阻塞、冻结、消亡



创建线程的第二种方式——实现Runnable接口　【注】：第一种方式——继承Thread类

1、定义类实现Runnable接口。
2、覆盖接口中的run方法，将线程的任务代码封装到run方法中。
3、通过Thread类创建线程对象，并将Runnable接口的子类对象作为Thread类的构造函数的参数进行传递。为什么？因为线程的任务都封装在Runnable接口子类对象的run方法中。所以要在线程对象创建时就必须明确要运行的任务。
4、调用线程对象的start方法开启线程

```
class saletickets implements Runnable  //实现接口的类
{
       private int ticket = 100;
```

```
        public void run() //方法覆盖，要执行的线程任务
        {
                while(true)
                {
                        if(ticket>0)
                                System.out.println(Thread.currentThread().getName()+"..."+ticket--);
                }
        }
}
public class ticketDemo
{
        public static void main(String[] args)
        {
                saletickets t = new saletickets();//线程任务对象，这句代码也可以用接口的多态形式表示：
Runnable t = new saletickets();

                //那么怎么能够让线程获得任务呢？答：通过如下的构造函数传递
                Thread t1 = new Thread(t);//参数传递
                Thread t2 = new Thread(t);
                Thread t3 = new Thread(t);
                Thread t4 = new Thread(t);
                t1.start();
                t2.start();
                t3.start();
                t4.start();
        }
}
//代码说明：创建线程的另一种方法是声明实现 Runnable 接口的类。该类然后在类中覆盖 run 方法。然后可以分配该类的实例，在创建 Thread 时作为一个参数来传递并启动。
```

实现Runnable接口的好处：
1、将线程的任务（run方法）从线程的子类中分离出来，进行了单独的封装。（按照面向对象的思想将任务的封装成对象）。
2、避免了继承Thread类的单继承的局限性。
3、Runnable接口的出现，降低了线程对象和线程任务的耦合性
所以，创建线程的第二种方式较为常用。

线程安全问题

产生的原因：
1、多个线程在操作共享的数据。
2、操作共享数据的线程任务的代码有多条。
当一个线程在执行操作共享数据的多条代码过程中，其他线程参与了运算。就会导致线程安全问题的产生。
解决思路：
就是将多条操作共享数据的线程代码封装起来，当有线程在执行这些代码的时候，其他线程时不可以参与运算的。必须要当前线程把这些代码都执行完毕后，其他线程才可以参与运算。在java中，用同步代码块就可以解决这个问题。

同步代码块的格式：【注】：同步机制，也即“锁机制”，**火车上的卫生间**

```
synchronized(对象)
{
    需要被同步的代码；
}
```

同步的**好处**：解决了线程的安全问题。
同步的**弊端**：相对降低了效率，因为同步外的线程的都会判断同步锁，即**同步化会强制线程进行排队等候执行方法**。另外，同步可能会产生“**死锁**”。

有可能出现这样一种情况：多线程出现安全问题之后，加入了同步机制，然而安全问题仍然存在，怎么办？这时肯定是同步机制出了问题。因此，**同步的前提**：同步中必须有**多个线程**并使用**同一个锁**。

```
class saletickets implements Runnable
{
	private int ticket = 100;
	Object obj = new Object();
	public void run()
	{
		synchronized(obj) //同步代码块
		{
			while(true)
			{
				if(ticket>0)
					System.out.println(Thread.currentThread().getName()+"..."+ticket--);
			}
		}
	}
}
public class ticketDemo
{
	public static void main(String[] args)
	{
		saletickets t = new saletickets();//线程任务对象

		//那么怎么能够让线程获得任务呢？答：通过构造函数传递，如下
		Thread t1 = new Thread(t);
		Thread t2 = new Thread(t);
		Thread t3 = new Thread(t);
		Thread t4 = new Thread(t);
		t1.start();
		t2.start();
		t3.start();
		t4.start();
	}
}
//由上面的代码我们可以知道，要让对象在线程上有足够的安全性，就要判断哪些指令不能被分割执行，然后对不能分割执行的代码进行同步化。
```

举例：安全问题处理（用同步函数的方式来实现），

```
class bank
{
	private int sum;
	private Object obj = new Object();
	public void add(int x)
	{
		synchronized(obj)
		{
			sum = sum+x;
			System.out.println("sum = "+sum);
		}
	}
}
/* 使用同步函数方式来实现同步
class bank
{
	private int sum;
	public synchronized void add(int x)
	{
		sum = sum+x;
```

```
                System.out.println("sum = "+sum);
        }
}
*/

class customer implements Runnable
{
        private bank b = new bank();
        public void run()
        {
                for(int i=0;i<3;i++)
                {
                        b.add(100);
                }
        }
}
public class saveMoney
{
        public static void main(String[] args)
        {
                customer c = new customer();//线程任务对象

                Thread t1 = new Thread(c);
                Thread t2 = new Thread(c);
                Thread t3 = new Thread(c);
                t1.start();
                t2.start();
                t3.start();
        }
}
```

同步函数和同步代码块的区别：
同步代码块使用任意的对象作为锁。同步函数只能使用this作为锁。如果说，**一个类中只需要一个锁**，这时可以考虑同步函数，使用this锁，书写简单。但是，**一个类中如果需要多个锁，还有多个类中使用同一个锁**，这时只能使用同步代码块。
综上：建议使用同步代码块

多线程的死锁
场景一：同步嵌套

```
class SaleTicket implements Runnable
{
        private int tickets = 100;
        boolean flag = true;
        Object obj = new Object();
        public void run()
        {
                if(flag)
                {
                        while(true)
                        {
                                synchronized (obj) //同步代码块,obj锁
                                {
                                        sale(); //同步函数，this锁
                                }
                        }
                }
                else
```

```
                    while(true)
                            sale();
        }
        public synchronized void sale() //同步函数，this锁
        {
                synchronized (obj) //同步代码块,obj锁
                {
                        if(tickets>0)
                        {
                                try {Thread.sleep(10);} catch(InterruptedException e) {}
                                System.out.println(Thread.currentThread().getName()+"...fun..."+tickets-
-);
                        }
                }
        }
}
public class ThisLockDemo
{
        public static void main(String[] args) throws InterruptedException
        {
                SaleTicket t = new SaleTicket();

                Thread t1 = new Thread(t);
                Thread t2 = new Thread(t);
                t1.start();
                Thread.sleep(10);
                t.flag = false;
                t2.start();
        }
}
```

因此，在编程时，**要避免“同步嵌套”**。

下面写一个“死锁”程序（通过同步嵌套【即两层synchronized代码块】）实现）：

```
class DeathLock implements Runnable
{
        private boolean flag;
        DeathLock(boolean flag)
        {
                this.flag = flag;
        }

        public void run()
        {
                if(flag)
                {
                        synchronized (MyLock.LOCKA) //LOCKA锁
                        {
                                System.out.println("if....LOCKA");
                                synchronized (MyLock.LOCKB) //LOCKB锁
                                {
                                        System.out.println("if....LOCKB");
                                }
                        }
                }
                else
                {
                        synchronized (MyLock.LOCKB) //LOCKB锁
                        {
                                System.out.println("else....LOCKB");
```

```
                        synchronized (MyLock.LOCKA) //LOCKA锁
                        {
                                System.out.println("else....LOCKA");
                        }
                }
        }
    }
}
class MyLock
{
    public static final Object LOCKA = new Object();
    public static final Object LOCKB = new Object();
}
public class DeathLockDemo
{
    public static void main(String[] args)
    {
        DeathLock d1 = new DeathLock(true); // 创建第一个线程任务
        DeathLock d2 = new DeathLock(false); //创建第二个线程任务
        new Thread(d1).start(); //开启一个线程
        new Thread(d2).start(); //再开一个线程
    }
}
```

//最终的**运行结果**会是打印出两句话，并且**处于"卡死"状态**。

分析：

本程序创建了两个线程任务，并开启了两个线程，当其中一个线程执行第一个线程任务时，flag = true，于是它拿到了LOCKA锁，此时它**如果继续**执行就会拿到LOCKB锁；

然而这个时候CPU可能已经切到了另一个线程，这个线程在执行第二个线程任务，flag = false，它拿到的就是LOCKB锁，接着它**如果继续**执行就想要拿到LOCKA锁，而此时的LOCKA锁正在被第一个线程持有，所以它肯定拿不到。

这样就形成了一个局面：第一个线程已经拿到了LOCKA锁，想要继续拿LOCKB锁，而另一个已经拿到了LOCKB锁的线程又想要拿LOCKA锁，这样两个线程都想要拿到对方正在持有的锁，这样就形成了“死锁”。

//上面假设一种情况，就是“**如果继续**”，但是如果不继续执行的话，有可能“锁不住”，怎么能一定锁住呢，就是让程序循环运行，即用while循环，见下面一个代码：

使用了**while循环的死锁**，这样就一定可以锁的住了，如下：

```
class DeathLock implements Runnable
{
    private boolean flag;
    DeathLock(boolean flag)
    {
        this.flag = flag;
    }

    public void run()
    {
        if(flag)
        {
            while(true) //采用while循环，让程序一直运行，保证能够“锁住”
            {
                synchronized (MyLock.LOCKA) //LOCKA锁
                {
                    System.out.println("if....LOCKA");
                    synchronized (MyLock.LOCKB) //LOCKB锁
                    {
                        System.out.println("if....LOCKB");
                    }
                }
            }
```

```
            }
            else
            {
                    while(true)//采用while循环，让程序一直运行，保证能够“锁住”
                    {
                            synchronized (MyLock.LOCKB) //LOCKB锁
                            {
                                    System.out.println("else....LOCKB");
                                    synchronized (MyLock.LOCKA) //LOCKA锁
                                    {
                                            System.out.println("else....LOCKA");
                                    }
                            }
                    }
            }
        }
}
class MyLock
{
        public static final Object LOCKA = new Object();
        public static final Object LOCKB = new Object();
}
public class DeathLockDemo
{
        public static void main(String[] args)
        {
                DeathLock d1 = new DeathLock(true);
                DeathLock d2 = new DeathLock(false);
                new Thread(d1).start();
                new Thread(d2).start();
        }
}
```

多线程间的通信

多个线程都在处理同一个资源，但是处理的任务却不一样。举例：生产者，消费者

通过“同步”，解决了“还没生产就会产生消费”的问题，但是出现了连续的生产却没有消费的情况，这和实际中的情况不一样。这就要使用“**等待唤醒机制**”（wait()、notify()、notifyall()）。

wait()：该方法可以让线程处于**冻结状态**，并将线程临时存储到线程池中。

notify()：**唤醒**指定线程池中的**任意一个**线程。

notifyAll()：**唤醒**指定线程池中的**所有**线程。

此外，**这些方法必须使用在同步中**，因为它们是用来操作同步锁上的线程的状态的。在使用这些方法时，必须标识它们所属的锁。标识方法就是：锁对象.wait()、锁对象.notify()、锁对象.notifyAll()。**相同锁的notify()，可以获取相同锁的wait()**。

线程池

线程池是一种多线程处理形式，处理过程中将任务添加到队列，然后在创建线程后自动启动这些任务。**多线程技术**主要解决处理器单元内多个线程执行的问题，它可以显著减少处理器单元的闲置时间，增加处理器单元的吞吐能力。

```
class Res //描述资源
{
        private String name; //商品名称
        private int count;  //商品数量

        private boolean flag;//定义标记

        //提供给商品赋值的方法
        public synchronized void set(String name)
        {
```

```
            if(flag) //判断标记为true，则执行wait等待，为false，就生产。
                  try{wait();}catch(InterruptedException e) {} //本行属于if代码块

            this.name = name+"..."+count;
            count++;
            System.out.println(Thread.currentThread().getName()+"...生产者..."+this.name);

            //生产完毕，将标记改为true.
            flag = true;

            //唤醒消费者
            notify();
      }

      //提一个获取商品的方法
      public synchronized void get()
      {
            if(!flag)
                  try{wait();}catch(InterruptedException e) {}
            System.out.println(Thread.currentThread().getName()+"...消费者..."+this.name);

            //将标记改为false
            flag = false;

            //唤醒生产者
            notify();
      }
}
class Producer implements Runnable
{
      private Res r;
      Producer(Res r)
      {
            this.r = r;
      }
      public void run()
      {
            while(true)
            r.set("面包");
      }
}
class Consumer implements Runnable
{
      private Res r;
      Consumer(Res r)
      {
            this.r = r;
      }
      public void run()
      {
            while(true)
            r.get();
      }
}
public class ProducerConsumerDemo
{
      public static void main(String[] args)
      {
            // 1、创建资源
            Res r = new Res();
```

```
            //2、创建两个任务
            Producer pro = new Producer(r);
            Consumer con = new Consumer(r);

            //3、创建线程
            Thread t1 = new Thread(pro);
            Thread t2 = new Thread(con);

            t1.start();
            t2.start();
        }
}
```

运行结果：

```
Thread-1...消费者...面包...44329
Thread-0...生产者...面包...44330
Thread-1...消费者...面包...44330
Thread-0...生产者...面包...44331
Thread-1...消费者...面包...44331
Thread-0...生产者...面包...44332
Thread-1...消费者...面包...44332
Thread-0...生产者...面包...44333
Thread-1...消费者...面包...44333
Thread-0...生产者...面包...44334
Thread-1...消费者...面包...44334
Thread-0...生产者...面包...44335
Thread-1...消费者...面包...44335
.............................
```

从运行结果可以看出：Thread-0“生产一次”，Thread-1“消费一次”，如此交替执行.....，这样就能够符合实际情况了。

多生产多消费

问题1：重复生产，重复消费。
原因：经过分析，发现被唤醒的线程没有经过判断标记就开始工作了。从而导致了重复的生产和消费。
解决办法：被唤醒的线程必须判断标记，可以用“while循环”来替换“if判断”，

问题2：死锁了，所有线程都处于冻结状态。
原因：本方线程在唤醒时，又一次唤醒了本方线程，而本方线程循环判断标记，又继续等待，从而导致了所有的线程都等待了。
解决办法：如果本方线程唤醒了对方线程就可以解决问题了，这个可以使用notifyAll()方法实现。但是这样不就把所有的线程都唤醒了吗？是的，然而在本方被唤醒之后，它会去判断标记从而继续等待。这样对方就有线程可以执行了。

```
class Res //描述资源
{
        private String name;
        private int count;

        private boolean flag;//定义标记

        //生产商品的方法
        public synchronized void set(String name)
        {
                while(flag) //循环判断标记为true，则执行wait等待，为false，就生产。
                        try{wait();}catch(InterruptedException e) {}

                this.name = name+"..."+count;
                count++;
                System.out.println(Thread.currentThread().getName()+"...生产者..."+this.name);

                //生产完毕，将标记改为true.
```

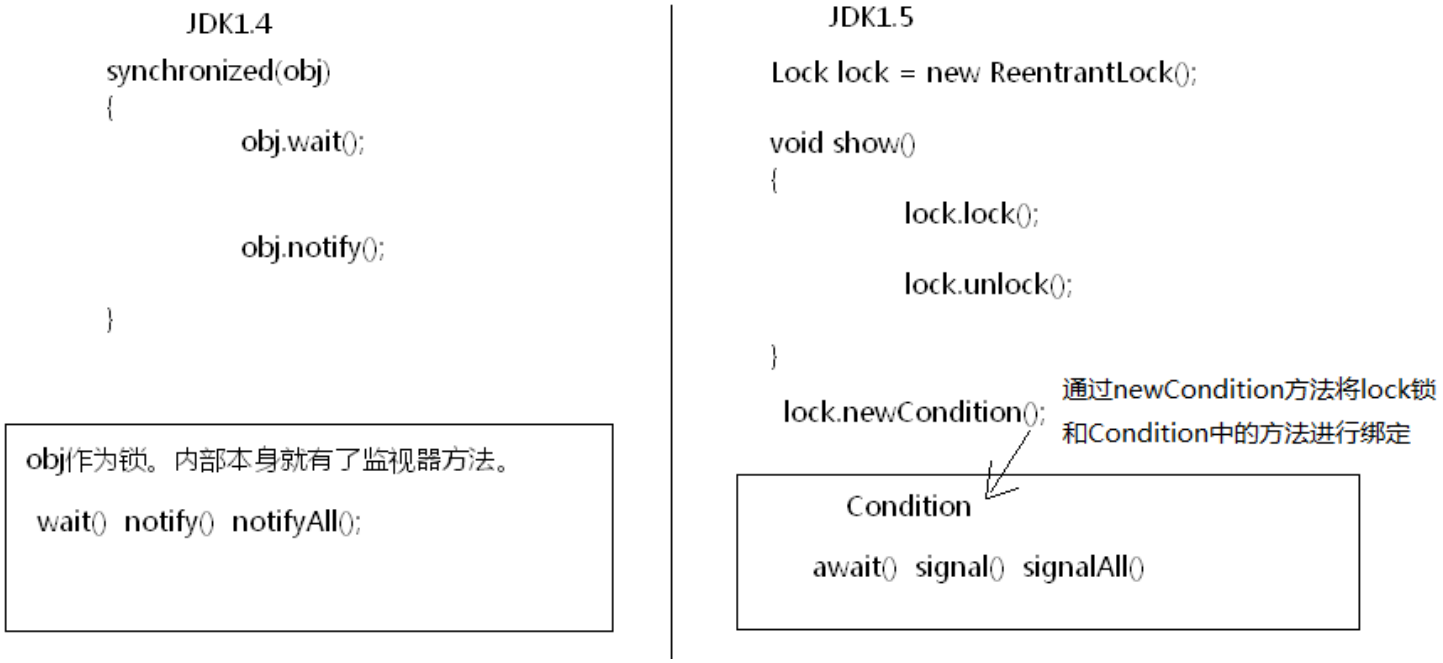
```
                flag = true;

                //唤醒所有消费者
                notifyAll();
        }

        //消费商品的方法
        public synchronized void get()
        {
                while(!flag) //循环判断
                        try{wait();}catch(InterruptedException e) {}
                System.out.println(Thread.currentThread().getName()+"...消费者..."+this.name);

                //将标记改为false
                flag = false;

                //唤醒所有生产者
                notifyAll();
        }
}
class Producer implements Runnable
{
        private Res r;
        Producer(Res r)
        {
                this.r = r;
        }
        public void run()
        {
                while(true)
                r.set("面包");
        }
}
class Consumer implements Runnable
{
        private Res r;
        Consumer(Res r)
        {
                this.r = r;
        }
        public void run()
        {
                while(true)
                r.get();
        }
}
public class ProducerConsumerDemo
{
        public static void main(String[] args)
        {
                // 1、创建资源
                Res r = new Res();

                //2、创建两个任务
                Producer pro = new Producer(r);
                Consumer con = new Consumer(r);

                //3、创建线程,多生产，多消费
                Thread t0 = new Thread(pro);  //生产
                Thread t1 = new Thread(pro);  //生产
```

```
            Thread t2 = new Thread(con);  //消费
            Thread t3 = new Thread(con);  //消费

            t0.start();
            t1.start();
            t2.start();
            t3.start();
        }
}
运行结果：
Thread-3...消费者...面包...52922
Thread-0...生产者...面包...52923
Thread-2...消费者...面包...52923
Thread-1...生产者...面包...52924
Thread-3...消费者...面包...52924
Thread-0...生产者...面包...52925
Thread-2...消费者...面包...52925
Thread-1...生产者...面包...52926
Thread-3...消费者...面包...52926
Thread-0...生产者...面包...52927
Thread-2...消费者...面包...52927
Thread-1...生产者...面包...52928
Thread-3...消费者...面包...52928
Thread-0...生产者...面包...52929
Thread-2...消费者...面包...52929
................................
```

上面的这个程序已经实现了“多生产多消费”，但是还是有点小问题，就是“效率有点低”，原因是因为notify()也唤醒了本方，做了不必要的判断。

jdk1.5的锁

使用了JDK1.5中 java.util.concurrent.locks包中的对象。即Lock接口：它的出现比synchronized有更多的操作。
Lock接口： 出现替代了同步代码块或者同步函数。将同步的隐式锁操作变成现实锁操作。同时更为灵活。可以一个锁上加上多组监视器。
同步函数或者同步代码块的锁操作是隐式的。而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():获取锁。
unlock():释放锁，通常需要定义finally代码块中。
Lock接口就是同步的替代，将线程中的同步更换为Lock接口的形式，然而当我们替换完毕发现“运行还是失败了”，这是因为wait没有了同步区域，没有了所属的同步锁。【注】：wait()、notify()、notifyAll()，这些方法都是放在“同步”中的。
同步升级了，其中锁已经不再是任意对象，而是Lock类型的对象。那么和任意对象绑定的监视器方法（wait()、notify()、notifyAll()）是不是也应该升级了，即有了专门和Lock类型锁绑定的监视器方法呢？查阅API可以发现，Condition接口就代替了Object中的监视器方法。
以前监视器方法封装到每一个对象中，现在将监视器方法封装到Condition对象中，方法名分别为：await，signal，signalAll
那么监视器对象Condition如何和Lock绑定呢？可以通过Lock接口的newCondition()方法来完成。下面，用图来展示jdk1.4中的锁及其方法和jdk1.5中的锁及其对应的方法，



将上一个例子的代码（jdk1.4版）用新的形式表现出来（jdk1.5版），如下：

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
class Res
{
	private String name;
	private int count;

	//创建新锁Lock
	private Lock lock = new ReentrantLock(); //多态

	//创建和Lock绑定的监视器对象
	private Condition con = lock.newCondition();

	private boolean flag;//定义标记

	//提供给商品赋值的方法
	public void set(String name)
	{
		lock.lock(); //获取锁

		try {
			while(flag) //判断标记为true，则执行wait等待，为false，就生产。
				try{con.await();}catch(InterruptedException e) {}

			this.name = name+"..."+count;
			count++;
			System.out.println(Thread.currentThread().getName()+"...生产者..."+this.name);

			//生产完毕，将标记改为true.
			flag = true;

			//唤醒所有线程
			con.signalAll();
		}
		finally
```

```
		{
			//释放锁
			lock.unlock();
		}
	}

	//提一个获取商品的方法
	public synchronized void get()
	{
		lock.lock(); //获取锁

		try {
		while(!flag)
			try{con.await();}catch(InterruptedException e) {}
		System.out.println(Thread.currentThread().getName()+"...消费者..."+this.name);

		//将标记改为false
		flag = false;

		//唤醒所有线程
		con.signalAll();
		}
		finally
		{
			//释放锁
			lock.unlock();
		}
	}
}
class Producer implements Runnable
{
	private Res r;
	Producer(Res r)
	{
		this.r = r;
	}
	public void run()
	{
		while(true)
		r.set("面包");
	}
}
class Consumer implements Runnable
{
	private Res r;
	Consumer(Res r)
	{
		this.r = r;
	}
	public void run()
	{
		while(true)
		r.get();
	}
}
public class ProducerConsumerDemo
{
	public static void main(String[] args)
	{
		// 1、创建资源
		Res r = new Res();
```

```
		//2、创建两个任务
		Producer pro = new Producer(r);
		Consumer con = new Consumer(r);

		//3、创建线程,多生产，多消费
		Thread t0 = new Thread(pro);  //生产
		Thread t1 = new Thread(pro);  //生产
		Thread t2 = new Thread(con);  //消费
		Thread t3 = new Thread(con);  //消费

		t0.start();
		t1.start();
		t2.start();
		t3.start();
	}
}
```

到了这里，“效率有点低”的问题依然没有被解决，原因是因为jdk1.5中的signalAll()依然唤醒了本方和对方的所有线程，还是做了不必要的判断，那么如何解决这个问题呢？使用“Lock绑定多个监视器”的功能，让Lock创建两个监视器，一个用于监视生产者，一个用于监视消费者，而生产者和消费者分别带有各自的监视方法（await，signal，signalAll）。这样既不会产生“死锁”，在唤醒时，也只会唤醒对方的线程，从而提高了效率。

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
class Res
{
	private String name;
	private int count;

	//创建新锁Lock
	private Lock lock = new ReentrantLock(); //多态

	//创建和Lock绑定的监视器对象,创建两个
	private Condition producer_con = lock.newCondition();//生产者监视器
	private Condition consumer_con = lock.newCondition();//消费者监视器

	private boolean flag;//定义标记

	//提供给商品赋值的方法
	public void set(String name)
	{
		lock.lock(); //获取锁

		try {
			while(flag) //判断标记为true，则执行wait等待，为false，就生产。
				try{producer_con.await();}catch(InterruptedException e) {}

			this.name = name+"..."+count;
			count++;
			System.out.println(Thread.currentThread().getName()+"...生产者..."+this.name);

			//生产完毕，将标记改为true.
			flag = true;

			//生产完毕，唤醒消费者，（只唤醒了对方）
			consumer_con.signalAll();
```

```
			}
			finally
			{
				//释放锁
				lock.unlock();
			}
		}

		//提一个获取商品的方法
		public synchronized void get()
		{
			lock.lock(); //获取锁

			try {
			while(!flag)
				try{consumer_con.await();}catch(InterruptedException e) {}
			System.out.println(Thread.currentThread().getName()+"...消费者..."+this.name);

			//将标记改为false
			flag = false;

			//消费完毕，唤醒生产者（只唤醒了对方）
			producer_con.signalAll();
			}
			finally
			{
				//释放锁
				lock.unlock();
			}
		}
}
class Producer implements Runnable
{
		private Res r;
		Producer(Res r)
		{
			this.r = r;
		}
		public void run()
		{
			while(true)
			r.set("面包");
		}
}
class Consumer implements Runnable
{
		private Res r;
		Consumer(Res r)
		{
			this.r = r;
		}
		public void run()
		{
			while(true)
			r.get();
		}
}
public class ProducerConsumerDemo
{
		public static void main(String[] args)
		{
```

```
            // 1、创建资源
            Res r = new Res();

            //2、创建两个任务
            Producer pro = new Producer(r);
            Consumer con = new Consumer(r);

            //3、创建线程,多生产，多消费
            Thread t0 = new Thread(pro);  //生产
            Thread t1 = new Thread(pro);  //生产
            Thread t2 = new Thread(con);  //消费
            Thread t3 = new Thread(con);  //消费

            t0.start();
            t1.start();
            t2.start();
            t3.start();
        }
}
```
运行结果同上。

上面的代码，在生产者生产时，都只生产了“一个”，而更符合实际情况的是：应该可以生产“**多个**”，并设定了一个上限。对于生产者来说，只要生产的数目没有达到上限，就一直生产；对于消费者来说，只要还有东西，就可以一直消费，直到为空。下面的代码取自API，用数组来存放生产出的多个东西，

```
class BoundedBuffer {
          final Lock lock = new ReentrantLock();
          final Condition notFull  = lock.newCondition();
          final Condition notEmpty = lock.newCondition();
          final Object[] items = new Object[100];//用来存储100个对象的数组
          int putptr, takeptr, count; //putprt（存），takeptr（取），count（计数）
          //存储对象 x 的方法put
          public void put(Object x) throws InterruptedException {
            lock.lock();
            try {
              while (count == items.length)  //判断计数器是否已满，已满则等待
                notFull.await();
              items[putptr] = x;  //未满，则存储一次
              if (++putptr == items.length) putptr = 0;//角标自增一次，并判断是否已满，已满则清零
              ++count;//计数
              notEmpty.signal();//唤醒对方
            } finally {
              lock.unlock();
            }
          }
          public Object take() throws InterruptedException {
            lock.lock();
            try {
              while (count == 0)  //判断是否有东西，若没有东西，则等待
                notEmpty.await();
              Object x = items[takeptr]; //若有东西，则继续“取出”
              if (++takeptr == items.length) takeptr = 0;
              --count; //自减一次
              notFull.signal();//唤醒对方
              return x;
            } finally {
              lock.unlock();
            }
```

```
        }
    }
```

wait和sleep的区别：

1、wait可以指定等待时间也可以不指定。sleep必须指定等待时间。
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。

会看异常提示

首先要看异常发生在哪个线程上（main线程 or Thread-0 or .....），再看具体的异常。

如何停止线程?

原理：让run方法结束就可以停止线程了。线程任务通常都有循环，因为开启线程就是为了执行需要一段时间的代码。只要控制住循环，就可以结束run方法，从而停止线程。控制循环，弄个标记即可。
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。当时强制动作会发生了InterruptedException，记得要处理。

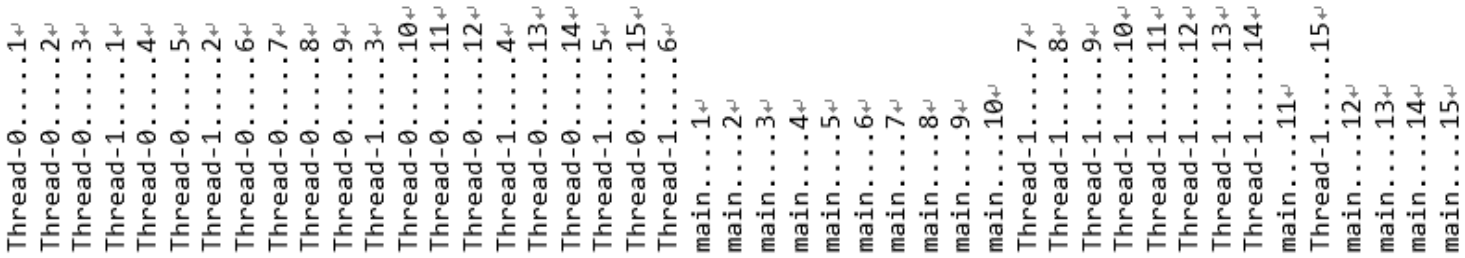
Join方法

join方法什么时候用？在执行某一个运算时，需要插入一个临时的运算，就可以单独开启一个线程把这个运算插进来，并写一个join方法把这个运算执行完毕再接着向下执行。

```
class Join implements Runnable
{
	public void run()
	{
		for(int x = 1;x<=15;x++)
		{
			System.out.println(Thread.currentThread().getName()+"....."+x);
		}
	}
}
public class JoinDemo
{
	public static void main(String[] args) throws InterruptedException
	{
		Join j = new Join();

		Thread t1 = new Thread(j);
		Thread t2 = new Thread(j);
		t1.start();
		t2.start();
		t1.join(); //等待该线程终止

		for(int x = 1;x<=15;x++)
		{
			System.out.println("main...."+x);
		}
	}
```

```
}
//在上面的代码中，开启了t1、t2线程，让t1、t2都有了执行资格，接着执行了t1.join，于是t1与t2相互切换执行，并且由于执行了t1.join，所以主线程必须要等t1执行完毕才可以执行。
```

运行结果：



多线程中常见的写法（开启线程）

在开发时，我们有时需要new一个任务并启动线程跑起来，下面给出两种**开启线程的便捷方式**。

```
public class ThreadTest
{
	public static void main(String[] args)
	{
		//开启线程的第一种便捷的方式
		new Thread()
		{
			public void run()
			{
				for(int i = 0;i<=150;i++)
				{
					System.out.println("x="+i);
				}
			}
		}.start();

		//开启线程的第二种便捷的方式
		Runnable r = new Runnable()
		{
			public void run()
			{
				for(int i = 0;i<=150;i++)
				{
					System.out.println("y="+i);
				}
			}
		};
		Thread t = new Thread(r);
		t.start();
		//t.join();

		//主线程
		for(int i = 0;i<=150;i++)
		{
			System.out.println("z="+i);
		}
	}
}
```

以上开启了三个线程，分别是两种“便捷式线程”和一个主线程。