Java多线程详解

线程池的优势：

1.降低线程的创建和销毁的开销

2.提升响应速度

3.线程池可以实现限流的效果（设置线程数）

多线程：指的是这个程序（一个进程）运行时产生了不止一个线程

线程对象是可以产生线程的对象。比如在Java平台中Thread对象，Runnable对象。线程，是指正在执行的一个指点令序列。在java平台上是指从一个线程对象的start()开始，运行run方法体中的那一段相对独立的过程。相比于多进程，多线程的优势有：

（1）进程之间不能共享数据，线程可以；

（2）系统创建进程需要为该进程重新分配系统资源，故创建线程代价比较小；

（3）Java语言内置了多线程功能支持，简化了java多线程编程。

1 永远在synchronized的方法或对象里使用wait、notify和notifyAll，不然Java虚拟机会生成IllegalMonitorStateException。

2 永远在while循环里而不是if语句下使用wait。这样，循环会在线程睡眠前后都检查wait的条件，并在条件实际上并未改变的情况下处理唤醒通知。

3 永远在多线程间共享的对象（在生产者消费者模型里即缓冲区队列）上使用wait。

在 Java 中可以用 wait、notify 和 notifyAll 来实现线程间的通信。线程在运行的时候，如果发现某些条件没有被满足，可以调用wait方法暂停自己的执行，并且放弃已经获得的锁，然后进入等待状态。当该线程被其他线程唤醒并获得锁后，可以沿着之前暂停的地方继续向后执行，而不是再次从同步代码块开始的地方开始执行。但是需要注意的一点是，对线程等待的条件的判断要使用while而不是if来进行判断。这样在线程被唤醒后，会再次判断条件是否正真满足。

一、创建线程和启动

（1）继承Thread类创建线程类

通过继承Thread类创建线程类的具体步骤和具体代码如下：

- 定义一个继承Thread类的子类，并重写该类的run()方法；
- 创建Thread子类的实例，即创建了线程对象；
- 调用该线程对象的start()方法启动线程。

复制代码

```
class SomeThead extends Thraad   {
    public void run()   {
     //do something here
    }
}

public static void main(String[] args){
 SomeThread oneThread = new SomeThread();
  步骤3：启动线程：
 oneThread.start();
}
```

复制代码

（2）实现Runnable接口创建线程类

通过实现Runnable接口创建线程类的具体步骤和具体代码如下：

- 定义Runnable接口的实现类，并重写该接口的run()方法；
- 创建Runnable实现类的实例，并以此实例作为Thread的target对象，即该Thread对象才是真正的线程对象。

复制代码

```
class SomeRunnable implements Runnable   {
  public void run()   {
  //do something here
  }
}
Runnable oneRunnable = new SomeRunnable();
Thread oneThread = new Thread(oneRunnable);
oneThread.start();
```

复制代码

（3）通过Callable和Future创建线程

通过Callable和Future创建线程的具体步骤和具体代码如下：

- 创建Callable接口的实现类，并实现call()方法，该call()方法将作为线程执行体，并且有返回值。
- 创建Callable实现类的实例，使用FutureTask类来包装Callable对象，该FutureTask对象封装了该Callable对象的call()方法的返回值。
- 使用FutureTask对象作为Thread对象的target创建并启动新线程。
- 调用FutureTask对象的get()方法来获得子线程执行结束后的返回值其中，Callable接口(也只有一个方法)定义如下：

复制代码

```
public interface Callable   {
  V call() throws Exception;
 }
```

步骤1：创建实现Callable接口的类SomeCallable(略)；

步骤2：创建一个类对象：

```
     Callable oneCallable = new SomeCallable();
```

步骤3：由Callable创建一个FutureTask对象：

```
    FutureTask oneTask = new FutureTask(oneCallable);
```

注释： FutureTask是一个包装器，它通过接受Callable来创建，它同时实现了 Future和Runnable接口。

步骤4：由FutureTask创建一个Thread对象：

```
    Thread oneThread = new Thread(oneTask);
```

步骤5：启动线程：

```
    oneThread.start();
```

复制代码

二、线程的生命周期

1、新建状态

用new关键字和Thread类或其子类建立一个线程对象后，该线程对象就处于新生状态。处于新生状态的线程有自己的内存空间，通过调用start方法进入就绪状态（runnable）。

注意：不能对已经启动的线程再次调用start()方法，否则会出现Java.lang.IllegalThreadStateException异常。

2、就绪状态

处于就绪状态的线程已经具备了运行条件，但还没有分配到CPU，处于线程就绪队列（尽管是采用队列形式，事实上，把它称为可运行池而不是可运行队列。因为cpu的调度不

一定是按照先进先出的顺序来调度的），等待系统为其分配CPU。等待状态并不是执行状态，当系统选定一个等待执行的Thread对象后，它就会从等待执行状态进入执行状态，系统挑选的动作称之为“cpu调度”。一旦获得CPU，线程就进入运行状态并自动调用自己的run方法。

提示：如果希望子线程调用start()方法后立即执行，可以使用Thread.sleep()方式使主线程睡眠一伙儿，转去执行子线程。

3、运行状态

处于运行状态的线程最为复杂，它可以变为阻塞状态、就绪状态和死亡状态。

处于就绪状态的线程，如果获得了cpu的调度，就会从就绪状态变为运行状态，执行run()方法中的任务。如果该线程失去了cpu资源，就会又从运行状态变为就绪状态。重新等待系统分配资源。也可以对在运行状态的线程调用yield()方法，它就会让出cpu资源，再次变为就绪状态。

注： 当发生如下情况是，线程会从运行状态变为阻塞状态：

①、线程调用sleep方法主动放弃所占用的系统资源

②、线程调用一个阻塞式IO方法，在该方法返回之前，该线程被阻塞

③、线程试图获得一个同步监视器，但更改同步监视器正被其他线程所持有

④、线程在等待某个通知（notify）

⑤、程序调用了线程的suspend方法将线程挂起。不过该方法容易导致死锁，所以程序应该尽量避免使用该方法。

当线程的run()方法执行完，或者被强制性地终止，例如出现异常，或者调用了stop()、desyory()方法等等，就会从运行状态转变为死亡状态。

4、阻塞状态

处于运行状态的线程在某些情况下，如执行了sleep（睡眠）方法，或等待I/O设备等资源，将让出CPU并暂时停止自己的运行，进入阻塞状态。

在阻塞状态的线程不能进入就绪队列。只有当引起阻塞的原因消除时，如睡眠时间已到，或等待的I/O设备空闲下来，线程便转入就绪状态，重新到就绪队列中排队等待，被系统选中后从原来停止的位置开始继续运行。有三种方法可以暂停Threads执行：

5、死亡状态

当线程的run()方法执行完，或者被强制性地终止，就认为它死去。这个线程对象也许是活的，但是，它已经不是一个单独执行的线程。线程一旦死亡，就不能复生。 如果在一个死去的线程上调用start()方法，会抛出java.lang.IllegalThreadStateException异常。

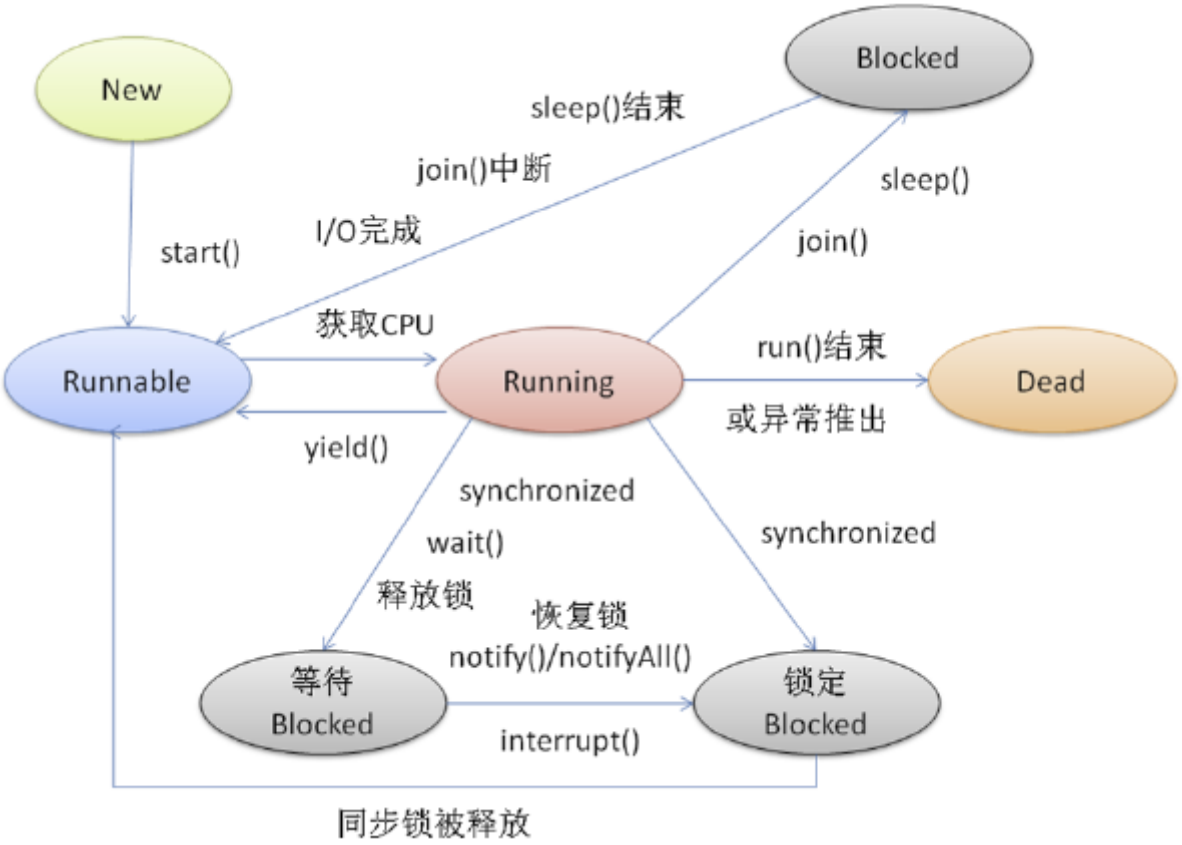
两张图：

public static enum **Thread.State**
extends Enum<Thread.State>

A thread state. A thread can be in one of the following states:

- NEW
  A thread that has not yet started is in this state.
- RUNNABLE
  A thread executing in the Java virtual machine is in this state.
- BLOCKED
  A thread that is blocked waiting for a monitor lock is in this state.
- WAITING
  A thread that is waiting indefinitely for another thread to perform a particular action is in this state.
- TIMED_WAITING
  A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state.
- TERMINATED
  A thread that has exited is in this state.

A thread can be in only one state at a given point in time. These states are virtual machine states which do not reflect any operating system thread states.



各种状态一目了然，值得一提的是"blocked"这个状态：

线程在Running的过程中可能会遇到阻塞(Blocked)情况

1. 调用join()和sleep()方法，sleep()时间结束或被打断，join()中断,IO完成都会回到Runnable状态，等待JVM的调度。
2. 调用wait()，使该线程处于等待池(wait blocked pool),直到notify()/notifyAll()，线程被唤醒被放到锁定池(lock blocked pool)，释放同步锁使线程回到可运行状态（Runnable）
3. 对Running状态的线程加同步锁(Synchronized)使其进入(lock blocked pool),同步锁被释放进入可运行状态(Runnable)。

此外，在runnable状态的线程是处于被调度的线程，此时的调度顺序是不一定的。Thread类中的yield方法可以让一个running状态的线程转入runnable。

三、线程管理

Java提供了一些便捷的方法用于会线程状态的控制。具体如下：

1、线程睡眠——sleep

如果我们需要让当前正在执行的线程暂停一段时间，并进入阻塞状态，则可以通过调用Thread的sleep方法。

注：

（1）sleep是静态方法，最好不要用Thread的实例对象调用它，因为它睡眠的始终是当前正在运行的线程，而不是调用它的线程对象，它只对正在运行状态的线程对象有效。如下面的例子：

复制代码

```
public class Test1 {
    public static void main(String[] args) throws InterruptedException {
        System.out.println(Thread.currentThread().getName());
        MyThread myThread=new MyThread();
        myThread.start();
        myThread.sleep(1000);//这里sleep的就是main线程，而非myThread线程
        Thread.sleep(10);
        for(int i=0;i<100;i++){
            System.out.println("main"+i);
        }
    }
}
```

复制代码

（2）Java线程调度是Java多线程的核心，只有良好的调度，才能充分发挥系统的性能，提高程序的执行效率。但是不管程序员怎么编写调度，只能最大限度的影响线程执行的次序，而不能做到精准控制。因为使用sleep方法之后，线程是进入阻塞状态的，只有当睡眠的时间结束，才会重新进入到就绪状态，而就绪状态进入到运行状态，是由系统控制的，我们不可能精准的去干涉它，所以如果调用Thread.sleep(1000)使得线程睡眠1秒，可能结果会大于1秒。

2、线程让步——yield

yield()方法和sleep()方法有点相似，它也是Thread类提供的一个静态的方法，它也可以让当前正在执行的线程暂停，让出cpu资源给其他的线程。但是和sleep()方法不同的是，它不会进入到阻塞状态，而是进入到就绪状态。yield()方法只是让当前线程暂停一下，重新进入就绪的线程池中，让系统的线程调度器重新调度器重新调度一次，完全可能出

现这样的情况：当某个线程调用yield()方法之后，线程调度器又将其调度出来重新进入到运行状态执行。

实际上，当某个线程调用了yield()方法暂停之后，优先级与当前线程相同，或者优先级比当前线程更高的就绪状态的线程更有可能获得执行的机会，当然，只是有可能，因为我们不可能精确的干涉cpu调度线程。用法如下：

复制代码

```
public class Test1 {
    public static void main(String[] args) throws InterruptedException {
        new MyThread("低级", 1).start();
        new MyThread("中级", 5).start();
        new MyThread("高级", 10).start();
    }
}

class MyThread extends Thread {
    public MyThread(String name, int pro) {
        super(name);// 设置线程的名称
        this.setPriority(pro);// 设置优先级
    }

    @Override
    public void run() {
        for (int i = 0; i < 30; i++) {
            System.out.println(this.getName() + "线程第" + i + "次执行！");
            if (i % 5 == 0)
                Thread.yield();
        }
    }
}
```

复制代码

注：关于sleep()方法和yield()方的区别如下：

①、sleep方法暂停当前线程后，会进入阻塞状态，只有当睡眠时间到了，才会转入就绪状态。而yield方法调用后 ，是直接进入就绪状态，所以有可能刚进入就绪状态，又被调度到运行状态。

②、sleep方法声明抛出了InterruptedException，所以调用sleep方法的时候要捕获该异常，或者显示声明抛出该异常。而yield方法则没有声明抛出任务异常。

③、sleep方法比yield方法有更好的可移植性，通常不要依靠yield方法来控制并发线程的执行。

3、线程合并——join

线程的合并的含义就是将几个并行线程的线程合并为一个单线程执行，应用场景是当一个线程必须等待另一个线程执行完毕才能执行时，Thread类提供了join方法来完成这个功能，注意，它不是静态方法。

从上面的方法的列表可以看到，它有3个重载的方法：

`void join()`

当前线程等该加入该线程后面，等待该线程终止。

`void join(long millis)`

当前线程等待该线程终止的时间最长为 `millis` 毫秒。 如果在`millis`时间内，该线程没有执行完，那么当前线程进入就绪状态，重新等待`cpu`调度

`void join(long millis,int nanos)`

等待该线程终止的时间最长为 `millis` 毫秒 + `nanos` 纳秒。如果在`millis`时间内，该线程没有执行完，那么当前线程进入就绪状态，重新等待`cpu`调度

4、设置线程的优先级

每个线程执行时都有一个优先级的属性，优先级高的线程可以获得较多的执行机会，而优先级低的线程则获得较少的执行机会。与线程休眠类似，线程的优先级仍然无法保障线程的执行次序。只不过，优先级高的线程获取CPU资源的概率较大，优先级低的也并非没机会执行。

每个线程默认的优先级都与创建它的父线程具有相同的优先级，在默认情况下，main线程具有普通优先级。

注：Thread类提供了setPriority(int newPriority)和getPriority()方法来设置和返回一个指定线程的优先级，其中setPriority方法的参数是一个整数，范围是1~·0之间，也可以使用Thread类提供的三个静态常量：

```
MAX_PRIORITY   =10

MIN_PRIORITY   =1

NORM_PRIORITY   =5
```

复制代码

```
    public class Test1 {
        public static void main(String[] args) throws InterruptedException {
            new MyThread("高级", 10).start();
            new MyThread("低级", 1).start();
        }
    }
```

```
    class MyThread extends Thread {
        public MyThread(String name,int pro) {
            super(name);//设置线程的名称
            setPriority(pro);//设置线程的优先级
        }
        @Override
        public void run() {
            for (int i = 0; i < 100; i++) {
                System.out.println(this.getName() + "线程第" + i + "次执行！");
            }
        }
    }
```

复制代码

注：虽然Java提供了10个优先级别，但这些优先级别需要操作系统的支持。不同的操作系统的优先级并不相同，而且也不能很好的和Java的10个优先级别对应。所以我们应该使用MAX_PRIORITY、MIN_PRIORITY和NORM_PRIORITY三个静态常量来设定优先级，这样才能保证程序最好的可移植性。

5、后台（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。调用线程对象的方法setDaemon(true)，则可以将其设置为守护线程。守护线程的用途为：

- 守护线程通常用于执行一些后台作业，例如在你的应用程序运行时播放背景音乐，在文字编辑器里做自动语法检查、自动保存等功能。
- Java的垃圾回收也是一个守护线程。守护线的好处就是你不需要关心它的结束问题。例如你在你的应用程序运行的时候希望播放背景音乐，如果将这个播放背景音乐的线程设定为非守护线程，那么在用户请求退出的时候，不仅要退出主线程，还要通知播放背景音乐的线程退出；如果设定为守护线程则不需要了。

setDaemon方法的详细说明：

复制代码

`public final void setDaemon(boolean on)` 将该线程标记为守护线程或用户线程。当正在运行的线程都是守护线程时，`Java` 虚拟机退出。

该方法必须在启动线程前调用。 该方法首先调用该线程的 `checkAccess` 方法，且不带任何参数。这可能抛出 `SecurityException`（在当前线程中）。

参数：

```
    on - 如果为 true，则将该线程标记为守护线程。
```

```
抛出：
```

```
    IllegalThreadStateException - 如果该线程处于活动状态。
    SecurityException - 如果当前线程无法修改该线程。
```

复制代码

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

6、正确结束线程

Thread.stop()、Thread.suspend、Thread.resume、Runtime.runFinalizersOnExit这些终止线程运行的方法已经被废弃了，使用它们是极端不安全的！想要安全有效的结束一个线程，可以使用下面的方法：

- 正常执行完run方法，然后结束掉；
- 控制循环条件和判断条件的标识符来结束掉线程。

复制代码

```
class MyThread extends Thread {
    int i=0;
    boolean next=true;
    @Override
    public void run() {
        while (next) {
            if(i==10)
                next=false;
            i++;
            System.out.println(i);
        }
    }
}
```

复制代码

四、线程同步

java允许多线程并发控制，当多个线程同时操作一个可共享的资源变量时（如数据的增删改查），将会导致数据不准确，相互之间产生冲突，因此加入同步锁以避免在该线程没有完成操作之前，被其他线程的调用，从而保证了该变量的唯一性和准确性。

1、同步方法

即有synchronized关键字修饰的方法。由于java的每个对象都有一个内置锁，当用此关键字修饰方法时，内置锁会保护整个方法。在调用该方法前，需要获得内置锁，否则就处于阻塞状态。

```
1 public synchronized vo
```

注： synchronized关键字也可以修饰静态方法，此时如果调用该静态方法，将会锁住整个类

2、同步代码块

即有synchronized关键字修饰的语句块。被该关键字修饰的语句块会自动被加上内置锁，从而实现同步。

复制代码

```
    public class Bank {

        private int count =0;//账户余额

        //存钱
        public   void addMoney(int money){

            synchronized (this) {
                count +=money;
            }
            System.out.println(System.currentTimeMillis()+"存进："+money);
        }

        //取钱
        public   void subMoney(int money){

            synchronized (this) {
                if(count-money < 0){
                    System.out.println("余额不足");
                    return;
                }
                count -=money;
            }
            System.out.println(+System.currentTimeMillis()+"取出："+money);
        }
```

```
        //查询
        public void lookMoney(){
            System.out.println("账户余额："+count);
        }
    }
```

复制代码

注：同步是一种高开销的操作，因此应该尽量减少同步的内容。通常没有必要同步整个方法，使用synchronized代码块同步关键代码即可。

3、使用特殊域变量(volatile)实现线程同步

- volatile关键字为域变量的访问提供了一种免锁机制；
- 使用volatile修饰域相当于告诉虚拟机该域可能会被其他线程更新；
- 因此每次使用该域就要重新计算，而不是使用寄存器中的值；
- volatile不会提供任何原子操作，它也不能用来修饰final类型的变量。

复制代码

```
public class SynchronizedThread {

        class Bank {

            private volatile int account = 100;

            public int getAccount() {
                return account;
            }

            /**
             * 用同步方法实现
             *
             * @param money
             */
            public synchronized void save(int money) {
                account += money;
            }

            /**
             * 用同步代码块实现
             *
             * @param money
```

```
             */
            public void save1(int money) {
                synchronized (this) {
                    account += money;
                }
            }
        }

        class NewThread implements Runnable {
            private Bank bank;

            public NewThread(Bank bank) {
                this.bank = bank;
            }

            @Override
            public void run() {
                for (int i = 0; i < 10; i++) {
                    // bank.save1(10);
                    bank.save(10);
                    System.out.println(i + "账户余额为：" +bank.getAccount());
                }
            }

        }

        /**
         * 建立线程，调用内部类
         */
        public void useThread() {
            Bank bank = new Bank();
            NewThread new_thread = new NewThread(bank);
            System.out.println("线程1");
            Thread thread1 = new Thread(new_thread);
            thread1.start();
            System.out.println("线程2");
            Thread thread2 = new Thread(new_thread);
            thread2.start();
        }

        public static void main(String[] args) {
```

```
            SynchronizedThread st = new SynchronizedThread();
            st.useThread();
        }

```

复制代码

注：多线程中的非同步问题主要出现在对域的读写上，如果让域自身避免这个问题，则就不需要修改操作该域的方法。用final域，有锁保护的域和volatile域可以避免非同步的问题。

4、使用重入锁（Lock）实现线程同步

在Java SE5.0中新增了一个java.util.concurrent包来支持同步。ReentrantLock类是可重入、互斥、实现了Lock接口的锁，它与使用synchronized方法和快具有相同的基本行为和语义，并且扩展了其能力。ReenreantLock类的常用方法有：

`ReentrantLock()` : 创建一个ReentrantLock实例

`lock()` : 获得锁

`unlock()` : 释放锁

注：ReentrantLock()还有一个可以创建公平锁的构造方法，但由于能大幅度降低程序运行效率，不推荐使用

复制代码

```
//只给出要修改的代码，其余代码与上同
        class Bank {

            private int account = 100;
            //需要声明这个锁
            private Lock lock = new ReentrantLock();
            public int getAccount() {
                return account;
            }
            //这里不再需要synchronized
            public void save(int money) {
                lock.lock();
                try{
                    account += money;
                }finally{
                    lock.unlock();
                }

```

```
            }
        }
```

复制代码

五、线程通信

1、借助于Object类的wait()、notify()和notifyAll()实现通信

线程执行wait()后，就放弃了运行资格，处于冻结状态；

线程运行时，内存中会建立一个线程池，冻结状态的线程都存在于线程池中，notify()执行时唤醒的也是线程池中的线程，线程池中有多个线程时唤醒第一个被冻结的线程。

notifyall(), 唤醒线程池中所有线程。

注：（1）wait(), notify(),notifyall()都用在同步里面，因为这3个函数是对持有锁的线程进行操作，而只有同步才有锁，所以要使用在同步中；

（2）wait(),notify(),notifyall()，在使用时必须标识它们所操作的线程持有的锁，因为等待和唤醒必须是同一锁下的线程；而锁可以是任意对象，所以这3个方法都是Object类中的方法。

单个消费者生产者例子如下：

复制代码

```
class Resource{   //生产者和消费者都要操作的资源
    private String name;
    private int count=1;
    private boolean flag=false;
    public synchronized void set(String name){
        if(flag)
            try{wait();}catch(Exception e){}
        this.name=name+"---"+count++;
        System.out.println(Thread.currentThread().getName()+"...生产者..."+this.name);
        flag=true;
        this.notify();
    }
    public synchronized void out(){
        if(!flag)
            try{wait();}catch(Exception e){}
        System.out.println(Thread.currentThread().getName()+"...消费者..."+this.name);
        flag=false;
        this.notify();
```

```
    }
}
class Producer implements Runnable{
    private Resource res;
    Producer(Resource res){
        this.res=res;
    }
    public void run(){
        while(true){
            res.set("商品");
        }
    }
}
class Consumer implements Runnable{
    private Resource res;
    Consumer(Resource res){
        this.res=res;
    }
    public void run(){
        while(true){
            res.out();
        }
    }
}
public class ProducerConsumerDemo{
    public static void main(String[] args){
        Resource r=new Resource();
        Producer pro=new Producer(r);
        Consumer con=new Consumer(r);
        Thread t1=new Thread(pro);
        Thread t2=new Thread(con);
        t1.start();
        t2.start();
    }
}//运行结果正常，生产者生产一个商品，紧接着消费者消费一个商品。
```

复制代码

但是如果有多个生产者和多个消费者，上面的代码是有问题，比如2个生产者，2个消费者，运行结果就可能出现生产的1个商品生产了一次而被消费了2次，或者连续生产2个商品而只有1个被消费，这是因为此时共有4个线程在操作Resource对象r,

而notify()唤醒的是线程池中第1个wait()的线程，所以生产者执行notify()时，唤醒的线程有可能是另1个生产者线程，这个生产者线程从wait()中醒来后不会再判断flag，而是直接向下运行打印出一个新的商品，这样就出现了连续生产2个商品。

为了避免这种情况，修改代码如下：

复制代码

```
    class Resource{
        private String name;
        private int count=1;
        private boolean flag=false;
        public synchronized void set(String name){
            while(flag) /*原先是if,现在改成while，这样生产者线程从冻结状态醒来时，还会再判断flag.*/
                try{wait();}catch(Exception e){}
            this.name=name+"---"+count++;
            System.out.println(Thread.currentThread().getName()+"...生产者..."+this.name);
            flag=true;
            this.notifyAll();/*原先是notity(), 现在改成notifyAll(),这样生产者线程生产完一个商品后可以将等待中的消费者线程唤醒，否则只将上面改成while后，可能出现所有生产者和消费者都在wait()的情况。*/
        }
        public synchronized void out(){
            while(!flag) /*原先是if,现在改成while，这样消费者线程从冻结状态醒来时，还会再判断flag.*/
                try{wait();}catch(Exception e){}
            System.out.println(Thread.currentThread().getName()+"...消费者..."+this.name);
            flag=false;
            this.notifyAll(); /*原先是notity(), 现在改成notifyAll(),这样消费者线程消费完一个商品后可以将等待中的生产者线程唤醒，否则只将上面改成while后，可能出现所有生产者和消费者都在wait()的情况。*/
        }
    }
    public class ProducerConsumerDemo{
        public static void main(String[] args){
            Resource r=new Resource();
            Producer pro=new Producer(r);
            Consumer con=new Consumer(r);
            Thread t1=new Thread(pro);
```

```
            Thread t2=new Thread(con);
            Thread t3=new Thread(pro);
            Thread t4=new Thread(con);
            t1.start();
            t2.start();
            t3.start();
            t4.start();
        }
    }
```

复制代码

七、死锁

产生死锁的四个必要条件如下。当下边的四个条件都满足时即产生死锁，即任意一个条件不满足既不会产生死锁。

（1）死锁的四个必要条件

- 互斥条件：资源不能被共享，只能被同一个进程使用
- 请求与保持条件：已经得到资源的进程可以申请新的资源
- 非剥夺条件：已经分配的资源不能从相应的进程中被强制剥夺
- 循环等待条件：系统中若干进程组成环路，该环路中每个进程都在等待相邻进程占用的资源

举个常见的死锁例子：进程A中包含资源A,进程B中包含资源B，A的下一步需要资源B，B的下一步需要资源A，所以它们就互相等待对方占有的资源释放，所以也就产生了一个循环等待死锁。

（2）处理死锁的方法

- 忽略该问题，也即鸵鸟算法。当发生了什么问题时，不管他，直接跳过，无视它；
- 检测死锁并恢复；
- 资源进行动态分配；
- 破除上面的四种死锁条件之一。