# 1. 进程和线程概述

每个运行的程序相当于一个“进程”，现在的操作系统是多进程的。比如我们可以同时运行“网易云音乐”和浏览器，边听歌曲边上网。其实CPU在某个时刻只能运行一个程序，只不过是操作系统在安排CPU频繁的切换所运行的进程，由于速度很快，所以人们感觉多个程序是同时进行的。

进程是系统进行资源分配和调用的一个基本单位，多进程提高了CPU的利用率。

而在一个进程中，又可以运行多个线程，这也是通过抢占CPU时间来运行的。我们以前写的程序只有一个执行流程，就是单线程程序。一个程序中也能“同时”进行多个执行流程，这就是多线程程序。比如一个安全软件能同时进行病毒查杀和电脑清理，是两个流程在“同时”进行，因此应用了多线程。

线程是依赖于进程的。对于开发者，主要考虑的是本程序的执行流程，也就是多线程。比如编写一个GUI程序，当进行到某一步时，此步骤需要执行大量计算并且耗时，如果使用单线程，那么界面此时是无响应的，必须等此步骤执行完毕，界面才有响应。若采用多线程，可另外开辟一个线程专门用于计算，此时该线程和界面线程抢占CPU时间，由于切换速度极快，因此用户不会感到界面无响应。

多进程和多线程都是并发，多进程是操作系统层面的并发，多线程是当前进程的并发，既然有并发，必然需要考虑处理多线程的“隐患”，比如线程安全、同步、死锁等问题，这个后面会讲（建议了解操作系统知识）。而对于开发者，考虑的主要就是本进程中的多线程。

# 2. 编写多线程程序

对于单线程程序，由java命令运行程序，运行程序时就启动JVM进程，接着由该JVM进程创建一个主线程调用main()方法。

在程序中创建新线程需要使用Thread类，通过查看此类的API，可知有两种方式创建新线程。下面进行讲解。

## 2.1 方式1：自定义类继承Thread类

通过自定义类继承Thread类实现多线程，步骤如下：

（1）新建类继承自Thread；

（2）重写run()方法，此方法中的代码就是该新线程需要执行的代码。

（3）需要创建并运行此线程时，就创建此类对象，并调用start()方法开始线程。这样新线程一旦获得CPU时间就会执行run()中的代码。

例子：

（1）自定义类继承Thread并重写run()方法

|  |
| --- |
| **package** com.zhang.test;  **public class** MyThread **extends** Thread {  @Override  **public void** run() {  *// 执行1000次输出* **for**(**int** i = 0; i < 1000; i++) {  System.***out***.println(i);  }  } } |

（2）主类创建多个线程

|  |
| --- |
| **package** com.zhang.test;  **public class** Demo {  **public static void** main(String[] args) {  *// 创建两个MyThread线程，让他们运行看效果* MyThread t1 = **new** MyThread();  MyThread t2 = **new** MyThread();  t1.start();  t2.start();  } } |

运行结果发现，两遍0-999的数字很可能会交叉输出，这就是多线程。如果是单线程的话，必定是先输出完一遍0-999，再输出一遍0-999，而不会交叉输出。如果看不到交叉输出的效果，可把循环次数再变大，因为很可能CPU执行很快，在一个线程首次得到CPU时就把0-999全部输出完毕了。

注意启动线程调用的是start()方法，而不是run()方法，若调用run()方法，那么JVM认为这是一个普通的方法调用，是没有多线程效果的。注意，一个线程对象只能启用一次，即只能调用一次start()方法，若调用一次以上，就会抛出“IllegalThreadStateException”异常。

当然，可以直接用匿名内部类的方式创建并开始一个线程，例如：

|  |
| --- |
| **package** com.zhang.test;  **public class** Demo {  **public static void** main(String[] args) {  **new** Thread() {  @Override  **public void** run() {  System.***out***.println(**"我是一个新线程"**);  }  }.start();  } } |

## 2.2 方式2：实现Runnable接口

此方法的步骤如下：

（1）定义一个类实现Runnable接口。同样重写其中的run()方法。

（2）用Thread类的一个重载创建对象，将Runnable接口实现类传递进Thread构造中。

（3）调用Thread对象的start()方法。

接口实现多线程的好处：避免由于Java单继承带来的局限性，这样可以使Runnable的实现类继承其他的类。使用第一种方式的话，自定义类只能继承Thread类了。

例子：

（1）MyThread实现Runnable：

|  |
| --- |
| **package** com.zhang.test;  **public class** MyThread **implements** Runnable {  @Override  **public void** run() {  **for**(**int** i = 0; i < 1000; i++) {  System.***out***.println(i);  }  } } |

（2）主类

|  |
| --- |
| **package** com.zhang.test;  **public class** Demo {  **public static void** main(String[] args) {  MyThread mt = **new** MyThread();  Thread t1 = **new** Thread(mt); *// 将Runnable实现类传递进去* Thread t2 = **new** Thread(mt);  *// 启动* t1.start();  t2.start();  } } |

也可以通过匿名内部类实现：

|  |
| --- |
| **package** com.zhang.test;  **public class** Demo {  **public static void** main(String[] args) {  **new** Thread(**new** Runnable() {  @Override  **public void** run() {  System.***out***.println(**"我是一个新线程"**);  }  }).start();  } } |

那么就有个很有趣的问题，使用new Thread()方式，既能在括号中传递一个实现Runnable的匿名内部类，把new Thread看成对象，也能将new Thread()本身当做匿名内部类，重写run方法。即：

|  |
| --- |
| **package** com.zhang.test;  **public class** Demo {  **public static void** main(String[] args) {  **new** Thread(**new** Runnable() {  @Override  **public void** run() {  System.***out***.println(**"这是2"**);  }  }){  @Override  **public void** run() {  System.***out***.println(**"这是1"**);  }  }.start();  } } |

此时输出结果是“这是1”，因为start()还是被Thread的匿名内部类对象调用的，传递的Runnable匿名内部类没有什么作用。不过也没人用这种方式，只是好玩。

# 3. 线程常用方法

## 3.1 普通操作

下面是Thread类常用的方法：

（1）public static Thread currentThread()：静态方法，返回当前正在执行的线程对象。

（2）public final String getName()：获得线程名称。默认线程的名称是“Thread-序号”，序号是JVM定的。

（3）public final void setName(String name)：设置线程的名称。此方法应该在start()线程之前操作。

对于线程的名称，可在创建线程时就设置，因为有构造方法：

Thread()、Thread(Runnable target)、Thread(Runnable target, String name)、Thread(String name)。如果是自定义线程类，可以借用父类的Thread(String)构造。

例如：

|  |
| --- |
| **package** com.zhang.test;  **public class** MyThread **extends** Thread {  **public** MyThread(String name) {  **super**(name);  }  @Override  **public void** run() {  System.***out***.println(**"此线程名称"** + **this**.getName());  } } |

然后MyThread也能通过new MyThread(“名字”)设置线程名了。若想获得主线程名，自然是通过Thread.currentThread().getName()的方式，主线程的名字就是“main”。

如果上述代码采用的是实现Runnable接口，就没法用this.getName()了，那么就用Thread.currentThread().getName()。

（4）设置和获取线程的优先级：

public final int getPriority()：获得线程优先级，默认是5，

public final void setPriority(int new Priority)：设置线程优先级。优先级从低到高可设置的范围是1-10。优先级就是一种抢占式调度模型，优先让优先级高的线程使用 CPU。

## 3.2 操作线程方法

（1）线程休眠：public static void sleep(long millis)：在指定的毫秒数内让当前正在执行的线程休眠（暂停执行），注意这是静态方法。

（2）设置后台线程（守护线程）：public final void setDaemon(boolean on)：如果设置为 true，则将该线程标记为守护线程。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。

守护进程就是在后台默默运行的线程，当所有的非后台线程结束时，程序也就终止了，同时还会杀死进程中的所有后台线程，也就是说，只要有非后台线程还在运行，程序就不会终止，执行main方法的主线程就是一个非后台线程。

如果不设置后台线程，那么就当所有的线程执行完后程序才结束。比如一个GUI程序，可能有很多后台线程，那么当主界面关闭后，程序就会终止，后台线程也会随着主线程关闭而被结束。如果此程序中都是用户线程，那么即使用户关闭了主界面，很可能程序也不会被关闭，还有线程在执行。这就是setDaemon()的特点。

# 4. 线程同步和锁

在多个线程使用到同一个资源时，就很可能有线程同步问题，这时我们可以用锁来解决。

现在看一个例子：现在有1000张票，需要有3个窗口同时卖票。解决这个问题就需要使用多线程。线程的run()方法就用来处理每个线程的不断卖票。由于每个线程是一起卖1000张票，因此不能把票数定义在方法中，而是把票数定义为线程类的静态变量。

例如：

（1）线程类：

|  |
| --- |
| **package** com.zhang.test;  **public class** MyThread **extends** Thread {  **public** MyThread() {}   **public** MyThread(String name) {  **super**(name);  }   *// 定义票数* **private static int** *TICKET* = 1000;   @Override  **public void** run() {  *// 不停卖票* **while**(**true**) {  *// 存在余票就售卖，否则退出卖票的循环* **if**(*TICKET* > 0) {  System.***out***.println(**this**.getName() + **"正在卖第"** + *TICKET* + **"张票"**);  *TICKET*--;  } **else** {  **break**;  }  }  } } |

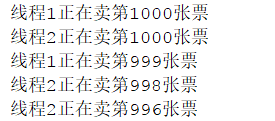
说明：为什么上述代码不直接用while(TICKET > 0)进行判断呢？这是为了模拟一直卖票的场景，窗口一直开启，只有无票时才结束。而且如果用while(TICKET > 0)的方式，就不好使用下面要讲的锁对象了，会导致一个线程把票卖完。

（2）主类：

|  |
| --- |
| **package** com.zhang.test;  **public class** Demo {  **public static void** main(String[] args) {  *// 新建三个线程，争抢卖票* **new** MyThread(**"线程1"**).start();  **new** MyThread(**"线程2"**).start();  **new** MyThread(**"线程3"**).start();  } } |

运行程序就会有三个线程在售卖1000张票。

但上述的程序是有问题的，有时会发现多个线程可能会售卖相同的票，例如有这样的情况：



如果在MyThread类中让线程“小睡”一会儿：

|  |
| --- |
| @Override **public void** run() {  *// 不停卖票* **while**(**true**) {  *// 存在余票就售卖，否则退出卖票的循环* **try** {  Thread.*sleep*(50); *// 睡眠50毫秒* } **catch** (InterruptedException e) {  e.printStackTrace();  }  **if**(*TICKET* > 0) {  System.***out***.println(**this**.getName() + **"正在卖第"** + *TICKET* + **"张票"**);  *TICKET*--;  } **else** {  **break**;  }  } } |

这样的错误就更明显了。为什么会出现这种情况呢？这是因为多个线程共享使用了同一个资源TICKET，就会出现线程安全问题。比如一开始的TICKET是1000，线程1先执行，它执行到打印输出“线程1正在卖第1000张票”时，突然操作系统将CPU的执行权交由了线程2执行。由于线程1的“TICKET--”还没有执行，此时TICKET还是1000，这样线程2进行输出时，也会输出“线程2正在卖第1000张票”，这样就产生了差错。同样，还可能出现三个窗口同时卖出第1000张票的情况。如果学习过操作系统相关知识，可能有对同步和线程安全问题有更深的理解。

上述展示的就是线程安全问题，当多线程共享资源时就可能出现这样的问题。如何解决呢？可用锁来解决这个问题，即用线程锁锁住使用的资源。下面详细讲。

## 4.1 synchronized关键字

synchronized关键字可以解决线程安全问题，synchronized关键字可以标记同步代码块和同步方法。

### 4.1.1 同步代码块

同步代码块的使用格式如下：

|  |
| --- |
| synchronized(锁对象) {  同步代码块  } |

这里需要一把锁，锁可以是任意对象，但要求：对于要同步资源的每个线程，锁对象必须是相同的，否则无法成功锁住共享资源，也就无法同步代码了（用锁解决线程安全问题就是使操作共享资源的代码同步执行，不会在某线程执行时被另一线程争抢到该资源。因为本线程获得了锁，别的线程就无法获得该锁进行执行，除非等到该线程的本次同步代码块中内容执行完毕并释放锁，其他线程才有机会拿到锁进行执行，从而使共享资源不被争抢）。

因此，可以用一个静态变量来定义一把锁，把对共享资源操作的代码放到同步代码块中。如果某个线程执行到同步代码块中，会立即获得锁（上锁），这时共享资源只能被该线程使用，即使CPU切换到别的线程，发现同步代码块已经被锁住了，那么线程也就不能再执行了，从而还是会执行原来获得锁的线程，除非原来的线程执行完同步代码块中内容，这时原来的线程会自动释放锁，接着这些线程会再争抢锁执行。

由于有了这样的过程，所以我们说使用线程安全的类会导致效率降低，但这在高并发时是不得不做的，否则数据将出错。

综上所述，我们可将MyThread类改进如下来解决线程安全问题：

|  |
| --- |
| **package** com.zhang.test;  **public class** MyThread **extends** Thread {  **public** MyThread() {}   **public** MyThread(String name) {  **super**(name);  }   *// 定义票数* **private static int** *TICKET* = 1000;  *// 定义锁* **private static final** Object ***LOCK*** = **new** Object();   @Override  **public void** run() {  *// 不停卖票* **while**(**true**) {  *// 在while中包含同步代码块* **synchronized** (***LOCK***) {  *// 存在余票就售卖，否则退出卖票的循环* **if**(*TICKET* > 0) {  System.***out***.println(**this**.getName() + **"正在卖第"** + *TICKET* + **"张票"**);  *TICKET*--;  } **else** {  **break**;  }  }  }  } } |

要看准synchronized同步代码块开始和结束的位置！

首先，同步代码块不能写在if里面，因为这样根本起不到线程安全的效果。因为if中已经用到了共享资源TICKET，如果多个线程同时通过了if判断，同样会出现线程安全问题！

其次，同步代码块也不能写在while外面，因为这样的话，程序会让一个线程卖完所有的票！因为一个线程获得锁之后，一直循环，直到把1000张票卖没了才停止，其他线程没有机会进行卖票，这是需要注意的。

### 4.1.2 同步方法

使用synchronized关键字修饰方法，使该方法成为一个同步方法，那么就不需要使用锁对象和同步代码块了。注意，synchronized只能修饰方法，不能修饰类和变量。

例如将代码改为同步方法：

|  |
| --- |
| **package** com.zhang.test;  **public class** MyThread **extends** Thread {  **public** MyThread() {}   **public** MyThread(String name) {  **super**(name);  }   *// 定义票数* **private static int** *TICKET* = 1000;    @Override  **public void** run() {  *// 不停卖票* **while**(**true**) {  *// 把卖票抽取成一个同步方法进行调用  // 无票则结束线程* **if** (!sellTicket())  **break**;  }  }   *// 卖票的方法，使用同步方法。  // 方法的返回值表示是否有剩余的票，用于结束线程的执行* **private synchronized boolean** sellTicket() {  **if**(*TICKET* > 0) {  **try** {  Thread.*sleep*(20);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.***out***.println(**this**.getName() + **"正在卖第"** + *TICKET* + **"张票"**);  *TICKET*--;  }  **return** *TICKET* > 0;  } } |

说明：为什么将卖票单独封装成一个同步方法？和之前一样的道理，不能在run方法上使用synchronized关键字把run方法变成一个同步方法。这是因为如果run是一个同步方法，那么所有的票都会被某一个线程卖光，因为一个线程执行run同步方法后，会一直循环卖票，其他线程卖不出票了。因此这里单独讲卖票的代码抽取成一个同步方法。

但是这样改进后，执行程序还是出现了线程安全问题，有可能会售卖相同的票，似乎同步方法并没有起到任何作用！这是因为：实例同步方法其实使用的是一个和本实例相关的内置锁（即对象锁）。sellTicket实例同步方法在MyThread类中，而我们每次都是新建一个MyThread对象来开启新线程的，因此导致每次执行实例同步方法时其内部使用的锁是不一样的（即是各自线程实例的对象锁）！

有两种方式解决这个问题：

方式一：使用类锁。也就是把实例同步方法改成静态同步方法，因为静态同步方法使用的锁是和本类相关的锁（类锁），即使新建多个本类对象，一个静态同步方法也会使用同一个类锁。因此可将同步方法改进如下：

|  |
| --- |
| *// 使用静态同步方法* **private static synchronized boolean** sellTicket() {  **if**(*TICKET* > 0) {  System.***out***.println(Thread.*currentThread*().getName() + **"正在卖第"** + *TICKET* + **"张票"**);  *TICKET*--;  }  **return** *TICKET* > 0; } |

方式二：MyThread线程类不使用继承Thread的方法，而是使用实现Runnable接口的方式。这样创建多个线程都可使用同一个MyThread对象（也即Runnable接口对象）。此时，TICKET也不用设置为静态常量，写为实例常量也可以，因为只需要用到一个MyThread对象。

具体代码如下：

（1）线程类：

|  |
| --- |
| **package** com.zhang.test;  **public class** MyThread **implements** Runnable {   *// 定义票数(可不用静态常量)* **private int ticket** = 1000;    @Override  **public void** run() {  *// 不停卖票* **while**(**true**) {  *// 调用实例同步方法即可* **if** (!sellTicket())  **break**;  }  }   *// 卖票的实例同步方法* **private synchronized boolean** sellTicket() {  **if**(**ticket** > 0) {  System.***out***.println(Thread.*currentThread*().getName() + **"正在卖第"** + **ticket** + **"张票"**);  **ticket**--;  }  **return ticket** > 0;  } } |

（2）主类

|  |
| --- |
| **package** com.zhang.test;  **public class** Demo {  **public static void** main(String[] args) {  *// 注意：使用同一个Runnable对象* Runnable runnable = **new** MyThread();  *// 用Runnable方式新建三个线程。* **new** Thread(runnable, **"线程1"**).start();  **new** Thread(runnable, **"线程2"**).start();  **new** Thread(runnable, **"线程3"**).start();  } } |

## 4.2 同步锁（Lock）

JDK5中提供了一个新的方式解决线程安全问题，就是使用同步锁Lock。同步锁是一个显式的锁，Lock接口中提供了lock和unlock方法，分别用于实现加锁和释放锁。使用这些API就能灵活地控制在哪里加锁和在哪里释放锁。当然，使用灵活也意味着存在一些风险，例如加锁后忘记释放锁，这是非常严重的，将会导致其他线程无法获得锁，也就无法执行了。

因此注意：推荐将unlock释放锁的操作放在finally代码块中，确保会释放锁。

Lock是一个接口，我们使用它的一个实现类：ReentrantLock（音标[ri:'entrənt]）来进行同步锁的操作。

现在，用同步锁的方式编写卖票程序：

|  |
| --- |
| **package** com.zhang.test;  **import** java.util.concurrent.locks.Lock; **import** java.util.concurrent.locks.ReentrantLock;  **public class** MyThread **implements** Runnable {   *// 定义票数(可不用静态常量)* **private int ticket** = 1000;  *// 创建锁对象(由于使用的是Runnable方式，因此可不用静态变量)* **private** Lock **lock** = **new** ReentrantLock();    @Override  **public void** run() {  *// 不停卖票* **while**(**true**) {  **lock**.lock(); *// 加锁* **try** {  **if**(**ticket** > 0) {  System.***out***.println(Thread.*currentThread*().getName() + **"正在卖第"** + **ticket** + **"张票"**);  **ticket**--;  } **else** {  **break**;  }  } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  **lock**.unlock(); *// finally中确保释放锁* }  }  } } |

再次提醒：加锁操作是在while里面、if的外面，否则一个线程将卖完所有的票。

# 5. 死锁

死锁是因线程间争夺资源产生的一种互相等待现象。比如线程A锁定了M资源，此时切换到线程B执行，线程B锁定了N资源，但同时线程B又需要M资源，由于M资源被A锁定，因此B线程等待A执行完释放资源。但是A执行过程中，同样需要N资源，此时N资源被锁，同样等待。至此，AB线程互相等待，产生死锁。如果出现了同步嵌套，就容易产生死锁问题。

例子（类似于哲学家就餐）：

|  |
| --- |
| **package** com.zhang.test;  **public class** Demo {  **public static void** main(String[] args) {  *// 假定中国人和美国人吃饭都需要“刀叉”和“筷子”。两种人吃饭是两个线程  // 但是只有一个刀叉和一双筷子   // 中国人吃饭线程* **new** Thread(**new** Runnable() {  @Override  **public void** run() {  **synchronized** (**"刀叉"**) {  System.***out***.println(**"我要先拿住刀叉，不给别人"**);  **synchronized** (**"筷子"**) {  System.***out***.println(**"我又拿到了筷子。吃完了就释放两个资源给别人使用"**);  }  }  }  }, **"中国人吃饭"**).start();   *// 美国人吃饭线程* **new** Thread(**new** Runnable() {  @Override  **public void** run() {  **synchronized** (**"筷子"**) {  System.***out***.println(**"我要先拿到筷子，不给别人"**);  **synchronized** (**"刀叉"**) {  System.***out***.println(**"我又拿到了刀叉。吃完了就释放两个资源给别人使用"**);  }  }  }  }, **"美国人吃饭"**).start();   } } |

那么上述的程序可能会出现死锁，只是可能，不是一定。如果“中国”线程刚拿到刀叉，就切换到“美国”线程，那么两个线程就会各自拿着刀叉和筷子不放，但是双方想要吃饭必须得到两个资源，这是线程死锁，程序一直等待，就出现下面的死锁输出：

|  |
| --- |
| 我要先拿到筷子，不给别人  我要先拿住刀叉，不给别人 |

有时没有出现死锁，则正常进行程序，要么中国人先吃，要么美国人先吃，比如这样的输出情况：

|  |
| --- |
| 我要先拿住刀叉，不给别人  我又拿到了筷子。吃完了就释放两个资源给别人使用  我要先拿到筷子，不给别人  我又拿到了刀叉。吃完了就释放两个资源给别人使用 |

# 6. 线程的通信

Object对象提供了等待和唤醒的方法，当该对象为锁对象时才能调用：

wait()：若执行了wait方法，则当前线程进入等待状态（即临时阻塞状态），即当前线程进入到以此锁对象为标识符的线程池中等待。

notify()：若执行了notify方法，会唤醒以此锁对象为标识符的线程池中等待线程中的一个线程。而notifyAll()方法就是唤醒该线程池中所有等待的方法。

上述方法就是线程通信需要用到的。

## 6.1 等待唤醒机制

等待唤醒机制就需要使用到线程的通信。等待唤醒机制中，一个经典的案例就是“生产者和消费者模式”，这里就使用“生产者和消费者模式”作为案例。

生产者和消费者模式是这样的：生产者负责生产产品，消费者负责消费产品。每个生产者或消费者都是一个线程，在应用中，可能同时有多个生产者和消费者线程在运行。此外，还有一个缓冲区用于暂存数据（例如缓冲区是10个长度），生产者将生产的数据放到缓冲区中，消费者从缓冲区中取数据。但要做到：

如果缓冲区满了，则应阻塞生产者线程，只让消费者进行消费；

如果缓冲区为空，则应阻塞消费者线程，只让生产者进行生产。

如此在应用中不断地进行下去（并不控制一定要生产满缓冲区才可消费数据，可以达到“随产随用”目的即可）。

## 6.2 消费者和生产者模式

编写代码，应该编写缓冲区类、生产者线程类和消费者线程类。由于生产者和消费者线程都要操作缓冲区，因此可以把缓冲区在构造函数中传递。

如果不采用等待唤醒机制的话，那么代码如下：

（1）缓冲区类。缓冲区类提供了向缓冲区添加数据和取出缓冲区数据的方法。

注意：需要考虑线程安全问题，因为缓冲区的数据会被多个消费者和生产者线程共享操作。下面采用同步代码块进行处理。

代码如下：

|  |
| --- |
| **package** com.zhang.test;  **import** java.util.LinkedList; **import** java.util.List;  *// 缓冲区类* **public class** MyBuffer {  *// 缓冲区，用一个集合表示。一开始集合为空，表示“无产品”* **private** List<Object> **buffer** = **new** LinkedList<>();  *// 定义缓冲区大小（最大长度）* **private int bufferSize**;   *// 传递缓冲区大小的构造方法* **public** MyBuffer(**int** bufferSize) {  **this**.**bufferSize** = bufferSize;  }   */\*\*  \* 提供给缓冲区添加对象的方法。  \** ***@param obj*** *加入的对象  \*/* **public void** addObject(Object obj) {  *// 下面需要操作共享的缓冲区，因此使用同步代码块。  // 就以缓冲区对象作为锁，因为生产者和消费者都会使用该对象，  // 因此缓冲区对象可作为相同的一把锁。* **synchronized** (**this**.**buffer**) {  **if** (**this**.**buffer**.size() >= **this**.**bufferSize**) {  *// 缓冲区满了* System.***out***.println(**"缓冲区已满"**);  } **else** {  *// 进行生产(即添加产品)* **this**.**buffer**.add(obj);  }  }  }   */\*\*  \* 提供获取缓冲区的一个对象的方法。  \** ***@return*** *获得的对象  \*/* **public** Object removeObject() {  *// 还是用this.buffer当做锁。* **synchronized** (**this**.**buffer**) {  Object retValue = **null**; *// 记录返回值。* **if** (**this**.**buffer**.size() > 0) {  retValue = **this**.**buffer**.remove(0);  }  **return** retValue;  }  }   *// 获得当前缓冲区对象数目* **public int** getBufferObjectCount() {  **return this**.**buffer**.size();  } } |

其实，也可使用同步方法，就不需要把代码放在同步代码块中了。因为实例同步方法的锁就是本类对象，本类对象作为锁也是没有问题的。

（2）生产者类：

|  |
| --- |
| **package** com.zhang.test;  *// 生产者线程类* **public class** Producer **implements** Runnable {   *// 用于接收缓冲区对象* **private** MyBuffer **buff**;   *// 构造方法接收缓冲区对象* **public** Producer(MyBuffer buff) {  **this**.**buff** = buff;  }   @Override  **public void** run() {  *// 循环生产，这里设置为进行50次循环* **for** (**int** i = 0; i < 50; i++) {  **buff**.addObject(**"产品"**);  String threadName = Thread.*currentThread*().getName();  System.***out***.println(threadName + **"生产了产品，缓冲区产品数："** + **buff**.getBufferObjectCount());  }  } } |

（3）消费者类：

|  |
| --- |
| **package** com.zhang.test;  **public class** Consumer **implements** Runnable {   *// 同样也要维护缓冲区* **private** MyBuffer **buff**;   *// 构造方法传递缓冲区* **public** Consumer(MyBuffer buff) {  **this**.**buff** = buff;  }   @Override  **public void** run() {  *// 下述代码和生产者类似* **for** (**int** i = 0; i < 50; i++) {  Object o = **buff**.removeObject();  String threadName = Thread.*currentThread*().getName();  System.***out***.println(threadName + **"消费了："** + o + **"，缓冲区产品数："** + **buff**.getBufferObjectCount());  }  } } |

（4）主类代码：

|  |
| --- |
| **package** com.zhang.test;  **public class** Demo {  **public static void** main(String[] args) {  *// 创建缓冲对象* MyBuffer myBuffer = **new** MyBuffer(100);  *// 创建生产者和消费者对应的Runnable* Runnable consumer = **new** Consumer(myBuffer);  Runnable producer = **new** Producer(myBuffer);  *// 开启线程* **new** Thread(consumer, **"消费者1"**).start();  **new** Thread(producer, **"生产者1"**).start();  } } |

说明：上述程序把添加对象（生产）和取出对象（消费）的方法放到了缓冲区类中，这样消费者类和生产者类只要调用缓冲区提供的操作方法即可，减轻了生产者和消费者的工作，使得缓冲区类更加通用。其实很多缓存框架都是这样做的，让缓存框架自己提供操作的API。

上述程序虽然可以正常运行，但是会出现这样的问题：当缓冲区已满（或为空）的时候，生产者（或消费者）还是继续争抢执行。实际上我们想要的是当缓冲区已满时，生产者线程就等待，让消费者线程执行，而缓冲区为空时，消费者线程就等待，而让生产者执行。因此，我们需要使用等待唤醒机制解决这个问题。

在实际开发中，我们管创建和添加数据的线程叫生产者线程，把销毁和删除数据的线程叫消费者线程。不能让一方线程过快，否则另一方来不及处理，因此用到等待唤醒机制。

下面就使用等待唤醒机制，我们只要改进MyBuffer类即可：

|  |
| --- |
| **package** com.zhang.test;  **import** java.util.LinkedList; **import** java.util.List;  *// 缓冲区类* **public class** MyBuffer {  *// 缓冲区，用一个集合表示。一开始集合为空，表示“无产品”* **private** List<Object> **buffer** = **new** LinkedList<>();  *// 定义缓冲区大小（最大长度）* **private int bufferSize**;   *// 传递缓冲区大小的构造方法* **public** MyBuffer(**int** bufferSize) {  **this**.**bufferSize** = bufferSize;  }   */\*\*  \* 提供给缓冲区添加对象的方法。如果缓冲区已满，则本方法所在线程会等待。  \** ***@param obj*** *加入的对象  \*/* **public void** addObject(Object obj) {  *// 下面需要操作共享的缓冲区，因此需要使用同步代码块。  // 就以缓冲区对象作为锁，因为生产者和消费者都会使用该对象，  // 因此缓冲区对象可作为相同的一把锁。* **synchronized** (**this**.**buffer**) {  *// 下面进行生产操作。如果缓冲区满了，就停止本次调用该方法的生产者线程，否则进行生产* **if** (**this**.**buffer**.size() >= **this**.**bufferSize**) {  *// 缓冲区满了* **try** {  **this**.**buffer**.wait(); *// wait当前线程，等待消费者使用。wait需要处理异常。* } **catch** (InterruptedException e) {  e.printStackTrace();  }  } **else** {  *// 进行生产(即添加产品)* **this**.**buffer**.add(obj);  *// 生产好了就唤醒其他线程，进行争抢执行  // 如果下面方法被首次调用，可能线程池中并没有等待唤醒的线程，不过不影响，只是没有被唤醒的。* **this**.**buffer**.notifyAll();  }  }  }   */\*\*  \* 提供获取缓冲区的一个对象的方法。如果缓冲区为空，则本方法所在线程会等待。  \** ***@return*** *获得的对象  \*/* **public** Object removeObject() {  *// 还是用this.buffer当做锁。* **synchronized** (**this**.**buffer**) {  Object retValue = **null**; *// 记录返回值。  // 同步代码块中，如果缓冲区为空则本次调用该方法的消费者线程等待，否则进行消费* **if** (**this**.**buffer**.size() <= 0) {  **try** {  **this**.**buffer**.wait();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  } **else** {  retValue = **this**.**buffer**.remove(0); *// 取出产品  // 消费完之后也要唤醒其他线程* **this**.**buffer**.notifyAll();  }  **return** retValue;  }  }   *// 获得当前缓冲区对象数目* **public int** getBufferObjectCount() {  **return this**.**buffer**.size();  } } |

再提醒一下：调用wait()和notify()的对象一定要是锁对象，且要在同步代码块或者同步方法中调用，否则程序运行会出现异常。

其实程序可以用同步方法改为如下：

|  |
| --- |
| **package** com.zhang.test;  **import** java.util.LinkedList; **import** java.util.List;  *// 缓冲区类* **public class** MyBuffer {  *// 缓冲区，用一个集合表示。一开始集合为空，表示“无产品”* **private** List<Object> **buffer** = **new** LinkedList<>();  *// 定义缓冲区大小（最大长度）* **private int bufferSize**;   *// 传递缓冲区大小的构造方法* **public** MyBuffer(**int** bufferSize) {  **this**.**bufferSize** = bufferSize;  }   */\*\*  \* 提供给缓冲区添加对象的方法。如果缓冲区已满，则本方法所在线程会等待。  \** ***@param obj*** *加入的对象  \*/* **public synchronized void** addObject(Object obj) {  *// 下面进行生产操作。如果缓冲区满了，就停止本次调用该方法的生产者线程，否则进行生产* **if** (**this**.**buffer**.size() >= **this**.**bufferSize**) {  *// 缓冲区满了* **try** {  *// 直接使用this当前对象调用wait等方法，因为实例方法的锁就是本类对象* **this**.wait(); *// wait当前线程，等待消费者使用。wait需要处理异常。* } **catch** (InterruptedException e) {  e.printStackTrace();  }  } **else** {  *// 进行生产(即添加产品)* **this**.**buffer**.add(obj);  *// 生产好了就唤醒其他线程，进行争抢执行  // 如果下面方法被首次调用，可能线程池中并没有等待唤醒的线程，不过不影响，只是没有被唤醒的。* **this**.notifyAll();  }  }   */\*\*  \* 提供获取缓冲区的一个对象的方法。如果缓冲区为空，则本方法所在线程会等待。  \** ***@return*** *获得的对象  \*/* **public synchronized** Object removeObject() {  *// 还是用this.buffer当做锁。* Object retValue = **null**; *// 记录返回值。  // 同步代码块中，如果缓冲区为空则本次调用该方法的消费者线程等待，否则进行消费* **if** (**this**.**buffer**.size() <= 0) {  **try** {  **this**.wait();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  } **else** {  retValue = **this**.**buffer**.remove(0); *// 取出产品  // 消费完之后也要唤醒其他线程* **this**.notifyAll();  }  **return** retValue;  }   *// 获得当前缓冲区对象数目* **public int** getBufferObjectCount() {  **return this**.**buffer**.size();  } } |

但是这样的程序还是有问题的，通过设置缓冲区大小为1（即主类new MyBuffer(1)即可，缓冲区最多存一个对象，即运行时消费者和生产者会交替执行），且循环为50次（或者更大就更容易出现问题），问题就出现了，会发现程序有时无法停止！现进行分析：

例如此时生产者线程执行到i=48的循环，而消费者执行到i=49的循环（一共都是50次），且此时缓冲区对象为空。这时，消费者线程争抢到了CPU执行权，但由于缓冲区对象为空，因此执行到removeObject方法中的this.wait();语句时，消费者线程会等待，让生产者执行。随后生产者线程执行完添加操作，并执行唤醒后，缓冲区对象就有了一个对象，此时生产者循环的i变为49了，然后生产者等待，消费者再执行。而消费者从this.wait()后开始执行，不会执行else语句内容，因此此时返回的结果只能是null。这时消费者执行完i为50，意味着消费者线程全部执行结束了，此时系统只有生产者线程了。而生产者线程的i=49将会有最后一次执行机会，而这时生产者执行时，发现缓冲区是满的，因此生产者线程又等待了，这将导致该线程等待后无人唤醒，因此程序就无法结束了。

因此，上述程序可能会导致程序无法结束，且消费者可能拿到null对象。运行程序这种情况很常见。如何解决这个问题？此时应该把else语句及其大括号去掉，让else语句每次都一定会执行唤醒！这样的话，就不会出现最后一个线程没有人唤醒了。而且这样做逻辑上也是没问题的，被唤醒后，生产者再添加对象或者消费者取出对象总会被执行，这也是应该的，被唤醒本就说明可添加对象（或取出对象）。

现在运行基本没什么问题了，但实际上还有问题，如果我们再添加一对生产者和消费者的话：

|  |
| --- |
| **package** com.zhang.test;  **public class** Demo {  **public static void** main(String[] args) {  *// 创建缓冲对象* MyBuffer myBuffer = **new** MyBuffer(1);  *// 创建生产者和消费者对应的Runnable* Runnable consumer = **new** Consumer(myBuffer);  Runnable producer = **new** Producer(myBuffer);  *// 开启线程* **new** Thread(consumer, **"消费者1"**).start();  **new** Thread(producer, **"生产者1"**).start();   **new** Thread(consumer, **"消费者2"**).start();  **new** Thread(producer, **"生产者2"**).start();  } } |

运行程序就报错了，会出现越界的情况，这是因为有可能两个消费者线程都进入了removeObject方法，且此时缓冲区为空，那么两个线程都会wait。当生产者生产一个后又被消费者线程抢占执行了，那么两个消费者都先后取出元素，就会产生错误！

因此上述说明的都是虚假唤醒的问题！怎么解决呢？实际上Java的API中已经为我们指明了方向，在Object的wait方法中这样说明的：

|  |
| --- |
| As in the one argument version, interrupts and spurious wakeups are possible, and this method should always be used in a loop:  synchronized (obj) {  while (<condition does not hold>)  obj.wait();  ... // Perform action appropriate to condition  } |

意思是“中断和虚假唤醒是可能的”，因此wait一般都是在循环中使用。因此我们只要把if再改成while即可，这样的话唤醒后会再判断一次条件，满足了才会添加或取出对象。

因此MyBuffer的最终版本为：

|  |
| --- |
| **package** com.zhang.test;  **import** java.util.LinkedList; **import** java.util.List;  *// 缓冲区类* **public class** MyBuffer {  *// 缓冲区，用一个集合表示。一开始集合为空，表示“无产品”* **private** List<Object> **buffer** = **new** LinkedList<>();  *// 定义缓冲区大小（最大长度）* **private int bufferSize**;   *// 传递缓冲区大小的构造方法* **public** MyBuffer(**int** bufferSize) {  **this**.**bufferSize** = bufferSize;  }   */\*\*  \* 提供给缓冲区添加对象的方法。如果缓冲区已满，则本方法所在线程会等待。  \** ***@param obj*** *加入的对象  \*/* **public synchronized void** addObject(Object obj) {  *// 下面进行生产操作。如果缓冲区满了，就停止本次调用该方法的生产者线程，否则进行生产* **while** (**this**.**buffer**.size() >= **this**.**bufferSize**) {  *// 缓冲区满了* **try** {  *// 直接使用this当前对象调用wait等方法，因为实例方法的锁就是本类对象* **this**.wait(); *// wait当前线程，等待消费者使用。wait需要处理异常。* } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  *// 进行生产(即添加产品)* **this**.**buffer**.add(obj);  *// 生产好了就唤醒其他线程，进行争抢执行  // 如果下面方法被首次调用，可能线程池中并没有等待唤醒的线程，不过不影响，只是没有被唤醒的。* **this**.notifyAll();   }   */\*\*  \* 提供获取缓冲区的一个对象的方法。如果缓冲区为空，则本方法所在线程会等待。  \** ***@return*** *获得的对象  \*/* **public synchronized** Object removeObject() {  *// 还是用this.buffer当做锁。* Object retValue = **null**; *// 记录返回值。  // 同步代码块中，如果缓冲区为空则本次调用该方法的消费者线程等待，否则进行消费* **while** (**this**.**buffer**.size() <= 0) {  **try** {  **this**.wait();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  retValue = **this**.**buffer**.remove(0); *// 取出产品  // 消费完之后也要唤醒其他线程* **this**.notifyAll();   **return** retValue;  }   *// 获得当前缓冲区对象数目* **public int** getBufferObjectCount() {  **return this**.**buffer**.size();  } } |

说明：上述程序会在消费者和生产者数目且循环次数相等的情况下正常运行和结束，如果它们不相等的话，是无法结束的，因为某一方会提前结束而剩下的一方等待后无法被唤醒！这个也不用去解决，本来就是不断地生产和消费。应该吧？如果要解决的话，那么程序终止的条件是一方线程都已完全结束（更严苛的话就是缓存区中不剩余对象），后续可考虑实现，实现不了就算。

## 6.3 Condition控制线程通信

JDK1.5后提供了同步锁Lock，同步锁也能使用等待唤醒机制，只要通过Lock对象的newCondition()方法创建一个Condition对象，通过Condition对象来控制线程通信即可。

Condition中提供了await()、signal()和signalAll()方法，它们分别相当于wait()、notify()和notifyAll()方法。因此，如果将上述的MyBuffer.java改成用Condition控制的话，则代码为：

|  |
| --- |
| **package** com.zhang.test;  **import** java.util.LinkedList; **import** java.util.List; **import** java.util.concurrent.locks.Condition; **import** java.util.concurrent.locks.Lock; **import** java.util.concurrent.locks.ReentrantLock;  *// 缓冲区类* **public class** MyBuffer {  *// 缓冲区，用一个集合表示。一开始集合为空，表示“无产品”* **private** List<Object> **buffer** = **new** LinkedList<>();  *// 定义缓冲区大小（最大长度）* **private int bufferSize**;   **private** Lock **lock** = **new** ReentrantLock(); *// 锁对象* **private** Condition **condition** = **lock**.newCondition(); *// condition对象   // 传递缓冲区大小的构造方法* **public** MyBuffer(**int** bufferSize) {  **this**.**bufferSize** = bufferSize;  }   */\*\*  \* 提供给缓冲区添加对象的方法。如果缓冲区已满，则本方法所在线程会等待。  \** ***@param obj*** *加入的对象  \*/* **public void** addObject(Object obj) {  **lock**.lock();  **try** {  **while** (**this**.**buffer**.size() >= **this**.**bufferSize**) {  *// 缓冲区满了* **try** {  **condition**.await();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  *// 进行生产(即添加产品)* **this**.**buffer**.add(obj);  **condition**.signalAll();  } **finally** {  **lock**.unlock();  }  }   */\*\*  \* 提供获取缓冲区的一个对象的方法。如果缓冲区为空，则本方法所在线程会等待。  \** ***@return*** *获得的对象  \*/* **public** Object removeObject() {  **lock**.lock();  **try** {  Object retValue = **null**; *// 记录返回值。* **while** (**this**.**buffer**.size() <= 0) {  **try** {  **condition**.await();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  retValue = **this**.**buffer**.remove(0); *// 取出产品  // 消费完之后也要唤醒其他线程* **condition**.signalAll();  **return** retValue;  } **finally** {  **lock**.unlock();  }  }   *// 获得当前缓冲区对象数目* **public int** getBufferObjectCount() {  **return this**.**buffer**.size();  } } |

## 6.4 案例：控制线程按序交替执行

案例：编写一个程序，开启 3 个线程，这三个线程的ID分别为A、 B和 C。每个线程将自己的ID在屏幕上打印10遍，要求输出的结果必须按顺序显示。如：ABCABCABC……。依次递归。

这里，我们做个“更难”的，案例中要求ABC打印10轮，现在我们要做的是“AAAAABBBCCC”打印10轮，即每轮A打印5次，B打印3次，C打印3次。

思路：这里要控制3个线程，先在AlternateDemo中维护一个变量number，用于标记当前正在操作的线程。要对线程进行控制（线程通信），需要有一个锁和相应的三个Condition（控制三个线程要有三个Condition）。

具体代码见下。

（1）AlternateDemo代码：

|  |
| --- |
| **public class** AlternateDemo {  **private int number** = 1; *// 标记当前正在执行的线程（1表示A，2表示B，3表示C）  // 一个锁对象和对应三个Condition* **private** Lock **lock** = **new** ReentrantLock();  **private** Condition **condition1** = **lock**.newCondition();  **private** Condition **condition2** = **lock**.newCondition();  **private** Condition **condition3** = **lock**.newCondition();   *// 提供输出A的方法。参数表示进行的是第几轮。* **public void** loopA(**int** loop) {  **lock**.lock();  **try** {  *// 1. 判断是否可以输出。为1表示是线程1，可以输出A，否则当前线程等待，通过condition1控制。* **if** (**number** != 1) {  **try** {  **condition1**.await();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  *// 2. 进行输出A，每轮输出5次，因此这里循环5次。* **for** (**int** i = 1; i <= 5; i++) {  *// 输出线程名、当前A循环的次数和这是第几轮* System.***out***.println(Thread.*currentThread*().getName() + **"\t"** + i + **"\t"** + loop);  }   *// 3. 唤醒线程2进行B的输出* **number** = 2;  **condition2**.signal();  } **finally** {  **lock**.unlock();  }  }   *// 同理，提供输出B的方法。参数表示进行的是第几轮。* **public void** loopB(**int** loop) {  **lock**.lock();  **try** {  *// 1. 判断* **if** (**number** != 2) {  **try** {  *// 通过condition2控制等待* **condition2**.await();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }   *// 2. 输出，循环输出3次即可* **for** (**int** i = 1; i <= 3; i++) {  System.***out***.println(Thread.*currentThread*().getName() + **"\t"** + i + **"\t"** + loop);  }   *// 3. 唤醒线程3输出C* **number** = 3;  **condition3**.signal();   } **finally** {  **lock**.unlock();  }  }   *// 同理，提供输出C的方法。参数表示进行的是第几轮。* **public void** loopC(**int** loop) {  **lock**.lock();  **try** {  *// 1. 判断* **if** (**number** != 3) {  **try** {  *// 通过condition2控制等待* **condition3**.await();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }   *// 2. 输出，循环输出3次即可* **for** (**int** i = 1; i <= 3; i++) {  System.***out***.println(Thread.*currentThread*().getName() + **"\t"** + i + **"\t"** + loop);  }   *// 3. 唤醒，再唤醒线程1输出A即可，就是这样循环的* **number** = 1;  **condition1**.signal();   } **finally** {  **lock**.unlock();  }  } } |

（2）主类代码：

|  |
| --- |
| **public class** TestABCAlternate {   **public static void** main(String[] args) {  *// 三个线程需要调用同一个AlternateDemo对象的不同方法进行输出。* AlternateDemo ad = **new** AlternateDemo();   *// 下述开启三个线程，循环10次调用对应的输出方法即可实现功能* **new** Thread(**new** Runnable() {  @Override  **public void** run() {  **for** (**int** i = 1; i <= 10; i++) {  ad.loopA(i);  }  }  }, **"A"**).start();   **new** Thread(**new** Runnable() {  @Override  **public void** run() {  **for** (**int** i = 1; i <= 10; i++) {  ad.loopB(i);  }  }  }, **"B"**).start();   **new** Thread(**new** Runnable() {  @Override  **public void** run() {  **for** (**int** i = 1; i <= 10; i++) {  ad.loopC(i);  }  }  }, **"C"**).start();  } } |

当然，如果要每轮只输出一次ABC，把loopA、loopB和loopC里面的循环只用一次即可。

说明：为什么上述代码能实现线程交替执行的控制呢？因为当一个线程被condition调用了await()方法等待时，只有该condition对象再调用signal()或signalAll()方法时才会被唤醒。因此使用condition进行控制更加灵活，因为能得到一个锁的多个condition对象，而锁对象的wait和notify不易实现。

# 7. 如何停止一个线程

线程中执行的任务一般是有循环的复杂任务，简单的任务没必要使用多线程。当一个线程的任务结束之后，该线程自然结束，但是如果一个线程中不停的执行循环任务，如何在外界进行有条件的停止呢？

比如下面例子中，线程是个死循环，我们想在main方法中停止该线程，即main中的for循环i达到80的时候，就停止线程。可以使用线程提供的stop方法，虽然过时了，但是可以用。即：

（1）线程类：

|  |
| --- |
| **package** com.zhang.test;  **public class** MyThread **extends** Thread {  @Override  **public void** run() {  **int** i = 0;  **while**(**true**) {  System.***out***.println(**"MyThread线程输出:"** + i++);  }  } } |

（2）主类

|  |
| --- |
| **package** com.zhang.test;  **public class** MainDemo {  **public static void** main(String[] args) {  MyThread t = **new** MyThread();  t.start();  **for**(**int** i = 0; i < 100; i++) {  System.***out***.println(**"主线程："** + i);  **if**(i == 80) {  *// 想要停止线程* t.stop();  }  }  } } |

既然stop()方法已经过时了，那么有什么方法替代呢？

解决办法之一就是使用变量来控制线程循环的执行。可以在线程类中维护一个变量，然后外界操作此变量，线程类的循环条件就是该变量。即改成：

（1）线程类：

|  |
| --- |
| **package** com.zhang.test;  **public class** MyThread **extends** Thread {  **public boolean flag** = **true**; *// 初始标记为true* @Override  **public void** run() {  **int** i = 0;  *// 这时循环条件是flag* **while**(**flag**) {  System.***out***.println(**"MyThread线程输出:"** + i++);  }  } } |

（2）主类：

|  |
| --- |
| **package** com.zhang.test;  **public class** MainDemo {  **public static void** main(String[] args) {  MyThread t = **new** MyThread();  t.start();  **for**(**int** i = 0; i < 100; i++) {  System.***out***.println(**"主线程："** + i);  **if**(i == 80) {  *// 设置线程flag为false来结束线程* t.**flag** = **false**;  }  }  } } |

但是有时线程会执行锁的wait()方法，这样线程一直阻塞，程序很可能不能正常停止，因此外界最好先调用notify()唤醒一下。由于这里用的是同步方法（wait()和notify()一定要用同步），wait()方法是由this对象调用的，所以外界也是通过线程对象来调用notify()方法。即：

（1）线程类：

|  |
| --- |
| **package** com.zhang.test;  **public class** MyThread **extends** Thread {  **public boolean flag** = **true**; *// 初始标记为true* @Override  **public synchronized void** run() { *// 加上synchronized关键字* **int** i = 0;  *// 这时循环条件是flag* **while**(**flag**) {  **try** {  **this**.wait();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.***out***.println(**"MyThread线程输出:"** + i++);  }  } } |

（2）主类：

|  |
| --- |
| **package** com.zhang.test;  **public class** MainDemo {  **public static void** main(String[] args) {  MyThread t = **new** MyThread();  t.start();  **for**(**int** i = 0; i < 100; i++) {  System.***out***.println(**"主线程："** + i);  **if**(i == 80) {  *// 设置线程flag为false来结束线程* t.**flag** = **false**;  *// 唤醒也要在同步代码块中，这里的锁与线程中锁要一致，都是线程对象* **synchronized** (t) {  t.notify();  }  }  }  } } |

如果不唤醒，很可能先执行到MyThread线程的话，程序就一直无法结束。

除了用notify()方法，这里还能调用interrupt()方法来唤醒线程。interrupt()方法的解释是中断线程，实际上此方法就是清除暂时阻塞线程的状态，让线程可执行。

interrupt()与notify()的区别在于：

（1）interrupt()可以直接执行，不需要同步代码块，而notify()需要。

（2）interrupt()唤醒的是指定的该线程，而notify()唤醒的线程是不确定的（如果有多个线程等待在此锁标识符的线程池中的话）。

（3）interrupt()方法会使此线程的wait()方法得到一个InterruptException异常，所以wait()方法需要处理异常。

即主类改为：（线程类无需改变）

|  |
| --- |
| **package** com.zhang.test;  **public class** MainDemo {  **public static void** main(String[] args) {  MyThread t = **new** MyThread();  t.start();  **for**(**int** i = 0; i < 100; i++) {  System.***out***.println(**"主线程："** + i);  **if**(i == 80) {  *// 设置线程flag为false来结束线程* t.**flag** = **false**;  *// 直接调用interrupt()方法* t.interrupt();  }  }  } } |

# 8. 线程组（ThreadGroup）

线程组可以对一批线程进行分类管理。默认情况下，线程是属于主线程组的。可以用Thread类的实例方法getThreadGroup得到一个线程的线程组对象：

public final ThreadGroup getThreadGroup()。

可以新建一个线程组对象，然后在新创建线程时，将线程设置到该线程组中以便统一管理。比如可用构造方法：

public Thread(ThreadGroup group, Runnable target, String name)，当然也有其他的构造方法。如果使用继承Thread类的方法，那么自定义线程类需要提供构造，调用Thread的构造方法来加入线程组。

下面是使用Runnable的例子，可以一起设置线程组的线程为后台线程（当然也有其他方法）。比如：

（1）Runnable实现类：

|  |
| --- |
| **package** com.zhang.test;  **public class** MyThread **implements** Runnable {  @Override  **public void** run() {  System.***out***.println(**"线程："** + Thread.*currentThread*().getName());  } } |

（2）主类

|  |
| --- |
| **package** com.zhang.test;  **public class** Demo {  **public static void** main(String[] args) {  *// 先看看主线程所在的线程组名称* System.***out***.println(Thread.*currentThread*().getThreadGroup().getName());*// 在主线程组main  // 创建线程组，指定名称* ThreadGroup group = **new** ThreadGroup(**"tg1"**);  *// 新创建线程加入到线程组* Thread t1 = **new** Thread(group, **new** MyThread(), **"线程1"**);  Thread t2 = **new** Thread(group, **new** MyThread(), **"线程2"**);  *// 同时设置为后台线程* group.setDaemon(**true**);  t1.start();  t2.start();  } } |

# 9. 线程池

## 9.1 线程池的概念和简介

我们在使用线程时，都是手动创建和开启线程，当线程执行结束该线程就会被JVM销毁。但是在应用中创建、开启和销毁线程是非常耗费资源的，尤其是在实际的大型项目中，频繁地创建销毁线程将会降低程序运行效率，因此，我们一般使用线程池来管理线程，通过线程池来创建线程。

JDK中本身就提供了线程池的实现（JDK1.5后提供的），和所有的“池”一样，都是为了提高效率，避免频繁地耗费资源而引入的，例如字符串池、以后将会学习的数据库连接池等。

线程池提供了一个线程队列，队列中保存着所有等待状态的线程（等待执行任务），只要给这些线程分配任务，线程就可执行。这样避免了创建与销毁线程的额外开销，提高了响应速度。因此线程池能提高效率，这也是我们使用的目的。

线程池的核心接口是Executor，它是维护线程的使用和调度的根接口。线程池的主要体系结构如下：

|  |
| --- |
| java.util.concurrent.Executor：负责线程的使用与调度的根接口  |-- ExecutorService：Executor的子接口，是线程池的主要接口  |-- ThreadPoolExecutor：ExecutorService的实现类，即线程池的实现类。  |--ScheduledExecutorService：ExecutorService的子接口，用于线程调度  |--ScheduledThreadPoolExecutor：ScheduledExecutorService的实现类。此类也继承了ThreadPoolExecutor。也就是说，ScheduledExecutorService既有线程池的功能，也能用于线程的调度。 |

但一般情况下，程序员不直接使用上面的实现类创建线程池对象来进行使用，因为JDK中把这部分工作也做好了。在ThreadPoolExecutor类的帮助文档中，第三段有这样一段话：

|  |
| --- |
| ......  However, programmers are urged to use the more convenient Executors factory methods Executors.newCachedThreadPool() (unbounded thread pool, with automatic thread reclamation), Executors.newFixedThreadPool(int) (fixed size thread pool) and Executors.newSingleThreadExecutor() (single background thread), that preconfigure settings for the most common usage scenarios. Otherwise, use the following guide when manually configuring and tuning this class:  ...... |

大意是说：强烈建议程序员使用更方便的“Exexutors”类中的工厂方法去创建线程池，例如使用Executors.newCachedThreadPool()等方法创建线程池。用这些方法创建的线程池都已经根据大部分通用的使用场景预先设置好了，否则如果想手动配置的话，则要看下面的指导......（这里就不继续了）

总之，我们要学会使用Executors工具类创建线程池并使用（JDK中很多工具类都是加s，例如Collections、Arrays和Files等）。常用Executors中的如下方法：

（1）public static ExecutorService newCachedThreadPool()：创建一个具有缓存功能的线程池（无界线程池）。线程池中的线程数不固定，可以根据需求自动变更数量。

（2）public static ExecutorService newFixedThreadPool(int nThreads) ：创建一个可重用的，具有固定线程数的线程池（固定线程池），此方法最常用；

（3）public static ExecutorService newSingleThreadExecutor()：创建一个只有一个线程的线程池（单个后台线程池），相当于将上个方法的参数设置为1。

上述方法的返回值都是ExecutorService对象，该对象就表示一个线程池。

## 9.2 使用线程池

使用线程池有如下三个步骤：

（1）创建线程池ExecutorService。利用上述的newFixedThreadPool等方法。

（2）为线程池中的线程分配任务。如果分配呢？ExecutorService中提供了一个submit方法，该方法就用来给线程池中的线程分配任务，即可用来执行线程任务。submit方法其中一个重载是：

Future<?> submit(Runnable task)，即此方法能执行Runnable实现类对象的线程任务。我们只要把线程对象交给submit，线程池就能执行分配的线程任务了。

（3）关闭线程池。调用ExecutorService的shutdown方法即可。shutdown方法会等待线程池中所有的任务执行结束后，才会关闭线程池。注意要将线程池关闭，如果不关闭线程池，程序将不会退出！ExecutorService还提供了一个shutdownNow方法，该方法会立即结束线程池，不管线程池中任务是否全部结束。

注意，一旦调用了shutdown或shutdownNow方法，就不能再用submit方法给线程池分配线程任务了。

例子：

（1）MyThread线程类：

|  |
| --- |
| **package** com.zhang.test;  **public class** MyThread **implements** Runnable {  @Override  **public void** run() {  System.***out***.println(Thread.*currentThread*().getName() + **"正在执行"**);  } } |

（2）主类：

|  |
| --- |
| **package** com.zhang.test;  **import** java.util.concurrent.ExecutorService; **import** java.util.concurrent.Executors;  **public class** ThreadPoolDemo {  **public static void** main(String[] args) {  *// 1. 创建线程数为3的线程池* ExecutorService pool = Executors.*newFixedThreadPool*(3);  *// 2. 给线程池中的线程分配线程任务，这里传递一个线程对象即可。* pool.submit(**new** MyThread());  *// 3. 关闭线程池* pool.shutdown();  } } |

运行上述程序后，输出结果为：

|  |
| --- |
| pool-1-thread-1正在执行 |

说明只要将实现Runnable接口的线程对象交给线程池，线程池自动开始任务，无需调用start方法。注意这里默认的线程名为“pool-1-thread-1”，表示这是这个“1”号线程池的“1”号线程。虽然我们创建的线程池的线程数是3，但由于我们只给了一个线程对象，即给了一个线程任务，因此只有一个线程被利用了。我们也可以在shutdown之前不断通过submit方法给线程池分配任务。例如这里改为分配10个MyThread任务：

|  |
| --- |
| **public static void** main(String[] args) {  *// 1. 创建线程数为3的线程池* ExecutorService pool = Executors.*newFixedThreadPool*(3);  *// 2. 分配10个MyThread任务（即10次）* MyThread myThread = **new** MyThread();  **for** (**int** i = 0; i < 10; i++) {  pool.submit(myThread); *// 当然也可以直接pool.submit(new MyThread());* }  *// 3. 关闭线程池* pool.shutdown(); } |

程序输出结果为：

|  |
| --- |
| pool-1-thread-1正在执行  pool-1-thread-3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-3正在执行  pool-1-thread-1正在执行  pool-1-thread-2正在执行  pool-1-thread-3正在执行 |

有10行输出说明确实执行了10个线程，但是线程名只有“pool-1-thread-1”、“pool-1-thread-2”和“pool-1-thread-3”，这是因为我们创建的线程池的线程数只有3，因此这些准备好的3个线程会被“重复利用”，直接任务全部结束才会关闭销毁。这个案例就体现了线程池的作用。

## 9.3 使用Callable

ExecutorService的submit方法还有重载：

<T> Future<T> submit(Callable<T> task)，说明submit也能执行Callable实现类对象的任务。这其实也算作创建线程的第三种方式：就是通过线程池执行Callable任务。

Callable也是接口，只不过重写的是call()方法，Callable可以当做Runnable使用。特点是：执行Callable任务是可以有返回值的，Callable有泛型，Callable使用什么样的泛型，call()方法就返回什么类型，并且这时submit的返回值Future也有泛型。Future中可以获得Callable任务得到的数据。

如果只是实现和Runnable一样的功能，直接call返回null即可，这里就不讲了，这里主要将获得Callable任务的返回数据。例子如下：

（1）MyCallable类：

|  |
| --- |
| **package** com.zhang.test;  **import** java.util.concurrent.Callable;  **public class** MyCallable **implements** Callable<String> {  @Override  **public** String call() **throws** Exception {  System.***out***.println(**"我返回的是字符串100"**);  **return "100"**;  } } |

（2）主类：

|  |
| --- |
| **package** com.zhang.test;  **import** java.util.concurrent.ExecutionException; **import** java.util.concurrent.ExecutorService; **import** java.util.concurrent.Executors; **import** java.util.concurrent.Future;  **public class** ThreadPoolDemo {  **public static void** main(String[] args) **throws** ExecutionException, InterruptedException {  *// 1. 创建线程池* ExecutorService pool = Executors.*newFixedThreadPool*(3);  *// 2. 分配MyCallable任务* Future<String> future = pool.submit(**new** MyCallable());  *// 3. 得到线程执行的结果* String res = future.get();  System.***out***.println(**"执行结果："** + res);  *// 4. 关闭线程池* pool.shutdown();  } } |

如果执行的是多个任务，也是同样的方法。如果想先把Future对象的数据保存起来，可以用一个集合添加即可。

|  |
| --- |
| **package** com.zhang.test;  **import** java.util.LinkedList; **import** java.util.List; **import** java.util.concurrent.ExecutionException; **import** java.util.concurrent.ExecutorService; **import** java.util.concurrent.Executors; **import** java.util.concurrent.Future;  **public class** ThreadPoolDemo {  **public static void** main(String[] args) **throws** ExecutionException, InterruptedException {  *// 1. 创建线程池* ExecutorService pool = Executors.*newFixedThreadPool*(3);  *// 2. 分配MyCallable任务  // 集合用于存放多个Future对象* List<Future<String>> futureList = **new** LinkedList<>();  MyCallable myCallable = **new** MyCallable();  **for** (**int** i = 0; i < 10; i++) {  *// 把返回的future添加到集合中* futureList.add(pool.submit(myCallable));  }  *// 3. 得到线程执行的结果  // 遍历集合* **for** (Future<String> future : futureList) {  System.***out***.println(**"执行结果："** + future.get());  }  *// 4. 关闭线程池* pool.shutdown();  } } |

但是上述程序不能分清哪个结果对应的是哪个线程，线程返回的结果不同，无法知道是哪个线程的结果。可以这样解决：在MyCallable中用一个key属性作为线程的键，然后主类保存每个线程的Future时，不使用List集合，而是使用Map即可，Map集合的键就是MyCallable的key，Map集合的键值就是MyCallable对应的Future，这样就能分清了，自行实现吧。

上面只是得到任务的结果数据。还可以利用Callable实现类的构造函数接收传递进线程的数据，实现互相传递数据。

例子：

（1）MyCallable类：

|  |
| --- |
| **package** com.zhang.test;  **import** java.util.concurrent.Callable;  **public class** MyCallable **implements** Callable<String> {  **private** String **data**; *// 接收数据的变量  // 用构造函数接收数据* **public** MyCallable(String data) {  **this**.**data** = data;  }  @Override  **public** String call() **throws** Exception {  System.***out***.println(**"我把接收的数据再返回去"**);  **return data**;  } } |

（2）主类：

|  |
| --- |
| **package** com.zhang.test;  **import** java.util.concurrent.ExecutionException; **import** java.util.concurrent.ExecutorService; **import** java.util.concurrent.Executors; **import** java.util.concurrent.Future;  **public class** Demo {  **public static void** main(String[] args) **throws** ExecutionException, InterruptedException {  ExecutorService pool = Executors.*newFixedThreadPool*(2);  Future<String> f = pool.submit(**new** MyCallable(**"hello world"**));  System.***out***.println(f.get());  pool.shutdown();  } } |

这个只不过是构造方法的使用技巧。

# 10. 定时器

## 10.1 使用定时器

定时器可用于调度多个定时任务以后台线程的方式执行。在Java中，可以通过Timer和TimerTask类来实现定时调度的功能。

首先自定义类继承自TimerTask类，重写run()方法，方法里面就写定时需要执行的任务。并且类中需要接收一个Timer对象，因为Timer对象有cancel()方法，只有在run()方法中调用cancel()方法，此任务才会结束。

这个Timer就是外界传递过来的，并且就是这个外界的Timer对象调用schedual()方法来执行此定时任务。schedual方法有下面参数：

（1）public void schedule(TimerTask task, long delay)：延迟delay毫秒后结束任务。TimerTask子类需要接收Timer调用cancel()方法。

（2）public void schedule(TimerTask task, long delay, long period)：每隔一段时间，并延迟delay毫秒就执行task任务，这是是连续执行，所以不需要传递Timer进来调用cancel()方法。

当然，schedual重载还能传递Date对象，指定特定时间执行任务。

定时执行一次的例子：

（1）自定义类：

|  |
| --- |
| **package** com.zhang.test;  **import** java.util.Timer; **import** java.util.TimerTask;  **public class** MyTask **extends** TimerTask {  **private** Timer **timer**;  **public** MyTask(Timer timer) {  **this**.**timer** = timer;  }  @Override  **public void** run() {  System.***out***.println(**"这是自定义定时任务"**);  *// 完成，关闭* **timer**.cancel();  } } |

（2）主类：

|  |
| --- |
| **package** com.zhang.test;  **import** java.util.Timer;  **public class** Demo {  **public static void** main(String[] args) {  *// 创建Timer对象* Timer timer = **new** Timer();  *// 延迟1秒执行* timer.schedule(**new** MyTask(timer), 1000);  } } |

定时执行多次的例子：

（1）自定义类：

|  |
| --- |
| **package** com.zhang.test;  **import** java.util.TimerTask;  **public class** MyTask **extends** TimerTask {  @Override  **public void** run() {  System.***out***.println(**"这是自定义定时任务"**);  } } |

（2）主类：

|  |
| --- |
| **package** com.zhang.test;  **import** java.util.Timer;  **public class** Demo {  **public static void** main(String[] args) {  *// 创建Timer对象* Timer timer = **new** Timer();  *// 每隔1秒执行一次* timer.schedule(**new** MyTask(), 0, 1000);  } } |

## 10.2 利用线程池进行线程调度

在上面的第9节中说到，用线程池也能进行线程调度。具体用法如下。

Executors中还有一个方法：

public static ScheduledExecutorService newScheduledThreadPool()：该方法用于创建固定线程数的线程池，可以延时或定时地执行任务。方法返回的ScheduledExecutorService就是用于线程调度的线程池。

给ScheduledExecutorService分配线程任务时，使用schedule方法，可对分配的任务设置定时执行。schedule方法有三个参数：

public ScheduledFuture<?> schedule(Runnable command,long delay, TimeUnit unit);

public <V> ScheduledFuture<V> schedule(Callable<V> callable, long delay, TimeUnit unit);

参数1可接收Runnable或Callable，参数2是该线程任务延迟多长时间执行，参数3用于指明参数2的时间单位。

同样，返回值ScheduledFuture还是用于得到任务执行返回的数据。

演示案例：延迟1秒返回一个随机数。

|  |
| --- |
| **package** com.zhang.test;  **import** java.util.\*; **import** java.util.concurrent.\*;  **public class** ThreadPoolDemo {  **public static void** main(String[] args) **throws** ExecutionException, InterruptedException {  *// 1. 创建线程池* ScheduledExecutorService pool = Executors.*newScheduledThreadPool*(3);  *// 2. 分配任务(这里用匿名内部类)* Future<Integer> future = pool.schedule(**new** Callable<Integer>() {  @Override  **public** Integer call() {  *// 返回一个100以内的随机数* Random random = **new** Random();  **return** random.nextInt(100);  }  }, 1, TimeUnit.***SECONDS***); *// 延迟1秒执行  // 输出执行的结果* System.***out***.println(future.get());  *// 3. 关闭线程池* pool.shutdown();  } } |

上述可以改为执行多个任务并得到结果，只要把上面部分代码放到一个循环内即可。即：

|  |
| --- |
| **package** com.zhang.test;  **import** java.util.\*; **import** java.util.concurrent.\*;  **public class** ThreadPoolDemo {  **public static void** main(String[] args) **throws** ExecutionException, InterruptedException {  *// 1. 创建线程池* ScheduledExecutorService pool = Executors.*newScheduledThreadPool*(3);  *// 2. 分配任务(这里用匿名内部类)  // 5个任务* **for** (**int** i = 0; i < 5; i++) {  Future<Integer> future = pool.schedule(**new** Callable<Integer>() {  @Override  **public** Integer call() {  *// 返回一个100以内的随机数* Random random = **new** Random();  **return** random.nextInt(100);  }  }, 1, TimeUnit.***SECONDS***); *// 延迟1秒执行  // 输出执行的结果* System.***out***.println(future.get());  }  *// 3. 关闭线程池* pool.shutdown();  } } |

上述程序执行后，会每隔1秒输出一个随机数，因为定时任务规定每个任务时延迟1秒执行，因此输出5个随机数后程序结束。

另外，ScheduledExecutorService还有如下分配任务的方法，解释如下：

（1）public ScheduledFuture<?>

scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit)：设置线程任务以固定的频率执行，参数2设置首次延迟时间，参数3设置执行周期，参数4设置时间单位。

（2）public ScheduledFuture<?> scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit)：设置线程任务以固定的延迟执行。参数2是首次延迟时间，参数3设置每次延迟时间，参数4是时间单位。

通过上述方法，就能更加灵活地设置定时调度任务了。

实际上，Java有一个开源的调度框架Quartz，它的功能更加强大，能非常灵活的设置各种调度任务，有时间可以学习。