# Java多线程

## 1、实现多线程的两种方式

### 1）继承自Thread类

（1）Thread类的常用方法：

currentThread()：获取当前线程对象；

yield()：主动放弃当前线程所占的CPU资源；

join()：阻塞等待另一个线程结束（有A、B两个线程，在A线程中调用B.join()，会导致A线程阻塞直到B线程结束）；

sleep(long millis)：释放CPU资源并且让当前线程睡眠一段时间，在此期间不在争夺CPU资源；

start()：开启一个线程；

getName()：获取当前线程的名称；

setDaemon(boolean on)：on为true时将该线程设置为后台线程（注：该方法必须在start() 前调用；当一个进程中没有非后台线程时会直接退出）；

getPriority()：获取当前线程的优先级；

setPriority(int newPriority)：设置当前线程的优先级（注：MIN\_PRIORITY = 1、NORM\_PRIORITY = 5，该值为缺省值、MAX\_PRIORITY = 10；不是必须在调用start() 前才能设置，线程开启后也可以随时设置；优先级高只是抢占到CPU的几率较大，并不是一直占用CPU）。

（2）示例代码：

**public class Test**

**{**

**public static void main(String[] args)**

**{**

**new MyThread().start();**

**int i = 100;**

**while (i-- > 0)**

**System.out.println(Thread.currentThread().getName());**

**}**

**}**

**class MyThread extends Thread**

**{**

**@Override**

**public void run()**

**{**

**int i = 100;**

**while (i-- > 0)**

**System.out.println(getName());**

**}**

**}**

### 2）实现Runnable接口

示例代码：

**public class Test**

**{**

**public static void main(String[] args)**

**{**

**MyThread mt = new MyThread();**

**new Thread(mt).start();**

**int i = 100;**

**while (i-- > 0)**

**System.out.println(Thread.currentThread().getName());**

**}**

**}**

**class MyThread implements Runnable**

**{**

**@Override**

**public void run()**

**{**

**int i = 100;**

**while (i-- > 0)**

**System.out.println(Thread.currentThread().getName());**

**}**

**}**

### 3）建议

（1）在不需要修改除run() 方法之外的其他方法时最好采用实现Runnable接口的方法，有两个好处：

① 自定义的类还可以继承自其他类；

② 方便线程间使用共享资源。

（2）示例代码：

**public class Test**

**{**

**public static void main(String[] args)**

**{**

**MyThread mt = new MyThread();**

**new Thread(mt).start();**

**new Thread(mt).start();**

**new Thread(mt).start();**

**}**

**}**

**class MyThread implements Runnable**

**{**

**int i = 1;**

**@Override**

**public void run()**

**{**

**while (true)**

**System.out.println(i++);**

**}**

**}**

### 4）使用内部类实现多线程

（1）这种方式同样具有实现Runnable接口的两个好处。

（2）示例代码：

**public class Test**

**{**

**public static void main(String[] args)**

**{**

**MyThread mt = new MyThread();**

**mt.getThread().start();**

**mt.getThread().start();**

**mt.getThread().start();**

**}**

**}**

**class MyThread**

**{**

**int i = 1;**

**private class InnerThread extends Thread**

**{**

**@Override**

**public void run()**

**{**

**while (true)**

**System.out.println(i++);**

**}**

**}**

**Thread getThread()**

**{**

**return new InnerThread();**

**}**

**}**

## 2、线程的同步（并发）

### 1）模拟售票系统

**public class Test**

**{**

**public static void main(String[] args)**

**{**

**SellThread st = new SellThread();**

**new Thread(st).start();**

**new Thread(st).start();**

**new Thread(st).start();**

**new Thread(st).start();**

**new Thread(st).start();**

**}**

**}**

**class SellThread implements Runnable**

**{**

**private int tickets = 100;**

**@Override**

**public void run()**

**{**

**while (true)**

**{**

**if (tickets > 0)**

**{**

**try**

**{**

**Thread.sleep(10); //让出CPU资源**

**}**

**catch (InterruptedException e)**

**{**

**e.printStackTrace();**

**}**

**System.out.println(Thread.currentThread().getName() + " sell ticket:" + tickets--);**

**}**

**else**

**break;**

**}**

**}**

**}**

在以上代码中，由于不同的线程访问了共享资源：tickets，而没有进行同步控制，导致了对共享资源的访问混乱，出现了0、-1、-2等不合适的值。

### 2）同步块

（1）原理：每个对象都有一个锁，通过加锁解锁操作完成同步。在使用同步块来完成线程的同步时，只需要对一个所有线程都共享的对象完成加锁解锁操作即可对临界区（对共享变量进行访问、修改等操作的区域）完成同步控制。在Java中，使用sychronized关键字来完成加锁解锁操作。

（2）示例代码：

**public class Test**

**{**

**public static void main(String[] args)**

**{**

**SellThread st = new SellThread();**

**new Thread(st).start();**

**new Thread(st).start();**

**new Thread(st).start();**

**new Thread(st).start();**

**new Thread(st).start();**

**}**

**}**

**class SellThread implements Runnable**

**{**

**private int tickets = 100;**

**private Object obj = new Object();**

**@Override**

**public void run()**

**{**

**while (true)**

**{**

**synchronized (obj)**

**{**

**if (tickets > 0)**

**{**

**try**

**{**

**Thread.sleep(10); //让出CPU资源**

**}**

**catch (InterruptedException e)**

**{**

**e.printStackTrace();**

**}**

**System.out.println(Thread.currentThread().getName() + " sell ticket:" + tickets--);**

**}**

**else**

**break;**

**}**

**}**

**}**

**}**

### 3）同步方法

（1）原理：将临界区代码提取成一个同步方法，在该方法中是通过this对象的锁来实现同步控制，同样使用到sychronized关键字。

（2）示例代码：

**public class Test**

**{**

**public static void main(String[] args)**

**{**

**SellThread st = new SellThread();**

**new Thread(st).start();**

**new Thread(st).start();**

**new Thread(st).start();**

**new Thread(st).start();**

**new Thread(st).start();**

**}**

**}**

**class SellThread implements Runnable**

**{**

**private int tickets = 100;**

**@Override**

**public void run()**

**{**

**while (true)**

**{**

**sell();**

**}**

**}**

**public synchronized void sell()**

**{**

**if (tickets > 0)**

**{**

**try**

**{**

**Thread.sleep(10); //让出CPU资源**

**}**

**catch (InterruptedException e)**

**{**

**e.printStackTrace();**

**}**

**System.out.println(Thread.currentThread().getName() + " sell ticket:" + tickets--);**

**}**

**}**

**}**

（3）对于同步静态方法，使用的是静态方法所在类的Class对象的锁。

### 4）线程的死锁

（1）线程死锁发生的原因：当一号线程占有A对象的锁，等待B对象的锁，而二号线程占有B对象的锁，等待A对象的锁时就会发送死锁。

（2）示例代码：

**public class Test**

**{**

**public static void main(String[] args) throws InterruptedException**

**{**

**SellThread st = new SellThread();**

**new Thread(st).start();**

**Thread.sleep(1);**

**st.flag = false;**

**new Thread(st).start();**

**}**

**}**

**class SellThread implements Runnable**

**{**

**private int tickets = 100;**

**private Object obj = new Object();**

**boolean flag = true;**

**@Override**

**public void run()**

**{**

**if (flag)**

**{**

**while (true)**

**{**

**synchronized (obj) //占有obj对象的锁**

**{**

**try**

**{**

**Thread.sleep(10);**

**}**

**catch (InterruptedException e1)**

**{**

**// TODO Auto-generated catch block**

**e1.printStackTrace();**

**}**

**synchronized (this) //等待this对象的锁**

**{**

**if (tickets > 0)**

**{**

**System.out.println(Thread.currentThread().getName() + " sell ticket:" + tickets--);**

**}**

**}**

**}**

**}**

**}**

**else**

**{**

**while (true)**

**{**

**sell();**

**}**

**}**

**}**

**public synchronized void sell() //占有this对象的锁**

**{**

**synchronized (obj) //等待obj对象的锁**

**{**

**if (tickets > 0)**

**{**

**System.out.println(Thread.currentThread().getName() + " sell ticket:" + tickets--);**

**}**

**}**

**}**

**}**

### 5）生产者消费者问题

（1）原理：每一个对象除了有一个锁之外，还有一个等待队列（wait set），当一个对象刚创建的时候，它的对待队列是空的。在当前线程锁住对象的锁后，去调用该对象的wait() 方法，此时，当前线程会进入等待状态，并释放锁。当调用对象的notify() 方法时，将从该对象的等待队列中删除一个任意选择的线程，这个线程将再次成为可运行的线程。当调用对象的notifyAll() 方法时，将从该对象的等待队列中删除所有等待的线程，这些线程将成为可运行的线程。

（2）示例代码：

**public class Test**

**{**

**public static void main(String[] args)**

**{**

**Queue q = new Queue();**

**new Producer(q).start();**

**new Consumer(q).start();**

**}**

**}**

**class Queue**

**{**

**private int value;**

**private boolean hasValue = false;**

**//生产者放值**

**public synchronized void put(int value)**

**{**

**if (!hasValue)**

**{**

**this.value = value;**

**hasValue = true;**

**notify();**

**}**

**try**

**{**

**wait();**

**}**

**catch (InterruptedException e)**

**{**

**e.printStackTrace();**

**}**

**}**

**//消费者取值**

**public synchronized int get()**

**{**

**if (!hasValue)**

**{**

**try**

**{**

**wait();**

**}**

**catch (InterruptedException e)**

**{**

**e.printStackTrace();**

**}**

**}**

**int value = this.value;**

**hasValue = false;**

**notify();**

**return value;**

**}**

**}**

**class Producer extends Thread //生产者**

**{**

**private Queue q;**

**public Producer(Queue q)**

**{**

**this.q = q;**

**}**

**@Override**

**public void run()**

**{**

**for (int i = 0; i < 10; i++)**

**{**

**q.put(i);**

**System.out.println("producer put:" + i);**

**}**

**}**

**}**

**class Consumer extends Thread**

**{**

**private Queue q;**

**public Consumer(Queue q)**

**{**

**this.q = q;**

**}**

**@Override**

**public void run()**

**{**

**while (true)**

**{**

**System.out.println("consumer get:" + q.get());**

**}**

**}**

**}**

## 3、线程的状态

### 1）五种状态

新建状态：刚创建出线程对象还未启动（new Thread()），当调用start() 方法后进入就绪状态；

就绪状态：已具备运行所需的除CPU资源之外的所有资源，处于系统就绪队列，正在等待系统分配CPU资源；

运行状态：正在占有CPU资源，执行run() 方法；

阻塞状态：在等待除CPU资源外的其他资源，如执行了sleep() 方法，或者正在等待IO；

死亡状态：线程死亡的方法有两个：一是完成了它的全部工作正常退出，二是被强制终止，如通过执行stop() 或者destroy() 方法可以终止一个线程（不推荐使用这两个方法，前者会产生异常，后者不会释放锁）。

### 2）终止线程

（1）通过使用一个标志位来控制线程结束。

（2）示例代码：

**public class Test**

**{**

**public static void main(String[] args) throws InterruptedException**

**{**

**MyThread mt = new MyThread();**

**mt.start();**

**Thread.sleep(2000);**

**mt.stopThread();**

**}**

**}**

**class MyThread extends Thread**

**{**

**private boolean flag = true;**

**public void stopThread()**

**{**

**flag = false;**

**}**

**@Override**

**public void run()**

**{**

**while (flag)**

**{**

**System.out.println(getName());**

**}**

**System.out.println("MyThread finish...");**

**}**

**}**

## 4、定时器Timer

### 1）作用

用于实现类似于闹钟的功能，定时或者周期性的触发一个线程。当创建一个Timer类的对象时，会自动开启一个线程，该线程用来按指定方式调用指定任务线程。

### 2）常用方法

schedule(TimerTask task, long delay)：指定时间后触发task任务线程（单位ms）；

schedule(TimerTask task, Date time)：指定时间触发task任务线程；

schedule(TimerTask task, long delay, long period)：指定时间后触发task任务线程，并开始周期性的触发该任务；

schedule(TimerTask task, Date firstTime, long period)：指定时间触发task任务线程，并开始周期性的触发该任务；

cancel()：取消此定时器。

（注：TimerTask类是实现了Runnable接口抽象类；一个Timer对象可以启动任意多个任务线程。）

### 3）示例代码

**public class Test**

**{**

**public static void main(String[] args) throws InterruptedException**

**{**

**Timer timer = new Timer();**

**timer.schedule(new MyTask1(), 1000);**

**timer.schedule(new MyTask2(), 2000, 500);**

**Thread.sleep(4000);**

**timer.cancel();**

**}**

**}**

**class MyTask1 extends TimerTask**

**{**

**@Override**

**public void run()**

**{**

**System.out.println("Task1 is running...");**

**}**

**}**

**class MyTask2 extends TimerTask**

**{**

**@Override**

**public void run()**

**{**

**System.out.println("Task2 is running...");**

**}**

**}**