# 3、具体内容

从多线程开始，Java正式进入到应用部分，而对于多线程的开发，从Java EE上表现的并不是特别多，但是在Android开发之中使用较多，并且需要提醒的是，笔试或面试的过程之中，多线程所问到的问题是最多的。

## 3.1、多线程的基本概念

如果要想解释多线程，那么首先应该从单进程开始讲起，最早的DOS系统有一个最大的特征：一旦电脑出现了病毒，电脑会立刻死机，因为传统DOS系统属于单进程的处理方式，即：在同一个时间段上只能有一个程序执行。后来到了windows时代，电脑即使（非致命）存在了病毒，那么也可以正常使用，只是慢一些而已，因为windows属于多进程的处理操作，但是这个时候的资源依然只有一块，所以在同一个时间段上会有多个程序共同执行，而在一个时间点上只能有一个程序在执行，多线程是在一个进程基础之上的进一步划分，因为进程的启动所消耗的时间是非常长的，所以在进程之上的进一步的划分就变得非常重要，而且性能也会有所提高。

所有的线程一定要依附于进程才能够存在，那么进程一旦消失了，线程也一定会消失，但是反过来不一定。而Java是为数不多的支持多线程的开发语言之一。

## 3.2、多线程的实现（重点）

在Java之中，如果要想实现多线程的程序，那么就必须依靠一个线程的主体类（就好比主类的概念一样，表示的是一个线程的主类），但是这个线程的主体类在定义的时候也需要有一些特殊的要求，这个类可以**继承Thread类或实现Runnable接口**来完成定义。

### 3.2.1、继承Thread类实现多线程

java.lang.Thread是一个负责线程操作的类，任何的类只需要继承了Thread类就可以成为一个线程的主类，但是既然是主类必须有它的使用方法，而线程启动的主方法是需要覆写Thread类中的run()方法才可以。

**范例：**定义一个线程的主体类

|  |
| --- |
| **class** MyThread **extends** Thread { // 线程的主体类  **private** String title;  **public** MyThread(String title) {  **this**.title = title;  }  @Override  **public** **void** run() { // 线程的主方法  **for** (**int** x = 0; x < 50; x++) {  System.*out*.println(**this**.title + "运行，x = " + x);  }  }  } |

现在按照道理来讲，已经出现了线程类，并且里面也存在了相应的操作方法，那么就应该产生对象并调用里面的方法，自然下面编写出了下的程序。

|  |
| --- |
| **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  MyThread mt1 = **new** MyThread("线程A") ;  MyThread mt2 = **new** MyThread("线程B") ;  MyThread mt3 = **new** MyThread("线程C") ;  mt1.run() ;  mt2.run() ;  mt3.run() ;  }  } |

但是以上的操作实话而言并没有真正的启动多线程，因为多个线程彼此之间的执行一定是交替的方式运行，而此时是顺序执行，即：每一个对象的代码执行完之后才向下继续执行。如果要想在程序之中真正的启动多线程，必须依靠Thread类的一个方法：public void start()，表示真正启动多线程，调用此方法后会间接调用run()方法。

|  |
| --- |
| **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  MyThread mt1 = **new** MyThread("线程A") ;  MyThread mt2 = **new** MyThread("线程B") ;  MyThread mt3 = **new** MyThread("线程C") ;  mt1.start() ;  mt2.start() ;  mt3.start() ;  }  } |

此时可以发现，多个线程之间彼此交替执行，但是每次的执行结果肯定是不一样的。通过以上的代码就可以得出结论：**要想启动线程必须依靠Thread类的start()方法执行，线程启动之后会默认调用了run()方法。**

**疑问？**为什么线程启动的时候必须调用start()而不是直接调用run()？

发现调用了start()之后，实际上它执行的还是覆写后的run()方法，那为什么不直接调用run()方法呢？那么为了解释此问题，下面打开Thread类的源代码，观察一下start()方法的定义。

|  |
| --- |
| **public** **synchronized** **void** start() {  **if** (threadStatus != 0)  **throw** **new** IllegalThreadStateException();  group.add(**this**);  **boolean** started = **false**;  **try** {  **start0();**  started = **true**;  } **finally** {  **try** {  **if** (!started) {  group.threadStartFailed(**this**);  }  } **catch** (Throwable ignore) {  }  }  }  **private** **native** **void** start0(); |

打开此方法的实现代码首先可以发现方法会抛出一个“IllegalThreadStateException”异常。按照之前所学习的方式来讲，如果一个方法之中使用了throw抛出一个异常对象，那么这个异常应该使用try…catch捕获，或者是方法的声明上使用throws抛出，但是这块都没有，因为这个异常类是属于运行时异常（RuntimeException）的子类。

|  |
| --- |
| java.lang.Object  |- java.lang.Throwable  |- java.lang.Exception  |- java.lang.**RuntimeException**  |- java.lang.IllegalArgumentException  |- java.lang.IllegalThreadStateException |

当一个线程对象被重复启动之后会抛出此异常，即：一个线程对象只能启动唯一的一次。在start()方法之中有一个最为关键的部分就是start0()方法，而且这个方法上使用了一个native关键字的定义。

native关键字指的是Java本地接口调用（Java Native Interface），即：是使用Java调用本机操作系统的函数功能完成一些特殊的操作，而这样的代码开发在Java之中几乎很少出现，因为Java的最大特点是可移植性，如果一个程序只能在固定的操作系统上使用，那么可移植性就将彻底的丧失，所以，此操作一般只作为兴趣使用。

多线程的实现一定需要操作系统的支持，那么以上的start0()方法实际上就和抽象方法很类似没有方法体，而这个方法体交给JVM去实现，即：在windows下的JVM可能使用A方法实现了start0()，而在linux下的JVM可能使用了B方法实现了start0()，但是在调用的时候并不会去关心具体是何方式实现了start0()方法，只会关心最终的操作结果，交给JVM去匹配了不同的操作系统。

所以在多线程操作之中，使用start()方法启动多线程的操作是需要进行操作系统函数调用的。

### 3.2.2、实现Runnable接口实现多线程

使用Thread类的确是可以方便的进行多线程的实现，但是这种方式最大的缺点就是单继承的问题，为此，在java之中也可以利用Runnable接口来实现多线程，而这个接口的定义如下：

|  |
| --- |
| **public** **interface** Runnable {  **public** **void** run();  } |

**分享：**如何区分新老接口？

在JDK之中，由于其发展的时间较长，那么会出现一些新的接口和老的接口，这两者有一个最大的明显特征：所有最早提供的接口方法里面都不加上public，所有的新接口里面都有public。

**范例：**通过Runnable接口实现多线程

|  |
| --- |
| **class** MyThread **implements** Runnable { // 线程的主体类  **private** String title;  **public** MyThread(String title) {  **this**.title = title;  }  @Override  **public** **void** run() { // 线程的主方法  **for** (**int** x = 0; x < 50; x++) {  System.*out*.println(**this**.title + "运行，x = " + x);  }  }  } |

这个时候和之前的继承Thread类区别不大，但是唯一的好处就是避免了单继承局限，不过现在问题也就来了。刚刚解释过，如果要想启动多线程依靠Thread类的start()方法完成，之前继承Thread类的时候可以将此方法直接继承过来使用，但现在实现的是Runable接口，没有这个方法可以继承了，为了解决这个问题，还是需要依靠Thread类完成，在Thread类中定义了一个构造方法：public Thread(Runnable target)，接收Runnable接口对象。

**范例：**利用Thread类启动多线程

|  |
| --- |
| **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  MyThread mt1 = **new** MyThread("线程A");  MyThread mt2 = **new** MyThread("线程B");  MyThread mt3 = **new** MyThread("线程C");  **new** Thread(mt1).start();  **new** Thread(mt2).start();  **new** Thread(mt3).start();  }  } |

这个时候就实现了多线程的启动，而且没有了单继承局限。

### 3.2.3、Thread类和Runnable接口实现多线程的区别（面试题）

现在Thread类和Runnable接口都可以做为同一功能的方式来实现多线程，那么这两者如果从Java的实际开发而言，肯定使用Runnable接口，因为可以有效的避免单继承的局限，那么除了这些之外，这两种方式是否还有其他联系呢？

为了解释这两种方式的联系，下面可以打开Thread类的定义：

|  |
| --- |
| public class Thread extends Object implements **Runnable** |

发现Thread类也是Runnable接口的子类，而如果真的是这样，那么之前程序的结构就变为了以下形式。

这个时候所表现出来的代码模式非常类似于代理设计模式，但是它并不是严格意义上代理设计模式，因为从严格来讲代理设计模式之中，代理主题所能够使用的方法依然是接口中定义的run()方法，而此处代理主题调用的是start()方法，所以只能够说形式上类似于代理设计模式，但本质上还是有差别的。

但是除了以上的联系之外，对于Runnable和Thread类还有一个不太好区分的区别：**使用Runnable接口可以更加方便的表示出数据共享的概念。**

**范例：**通过继承Thread类实现卖票程序

|  |
| --- |
| **package** cn.mldn.demo;  **class** MyThread **extends** Thread { // 线程的主体类  **private** **int** ticket = 5; // 一共5张票  @Override  **public** **void** run() { // 线程的主方法  **for** (**int** x = 0; x < 50; x++) {  **if** (**this**.ticket > 0) {  System.*out*.println("卖票，ticket = " + **this**.ticket --);  }  }  }  }  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  MyThread mt1 = **new** MyThread();  MyThread mt2 = **new** MyThread();  MyThread mt3 = **new** MyThread();  mt1.start() ;  mt2.start() ;  mt3.start() ;  }  } |

现在的结果是一共买出了15张票，等于是每一个线程对象各自卖各自的5张票，这个时候的内存关系图如下：

**范例：**利用Runnable来实现多线程

|  |
| --- |
| **package** cn.mldn.demo;  **class** MyThread **implements** Runnable { // 线程的主体类  **private** **int** ticket = 5; // 一共5张票  @Override  **public** **void** run() { // 线程的主方法  **for** (**int** x = 0; x < 50; x++) {  **if** (**this**.ticket > 0) {  System.*out*.println("卖票，ticket = " + **this**.ticket--);  }  }  }  }  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  MyThread mt = **new** MyThread();  **new** Thread(mt).start();  **new** Thread(mt).start();  **new** Thread(mt).start();  }  } |

现在使用继承Thread类也可以实现同样的功能。

|  |
| --- |
| **package** cn.mldn.demo;  **class** MyThread **extends** Thread { // 线程的主体类  **private** **int** ticket = 5; // 一共5张票  @Override  **public** **void** run() { // 线程的主方法  **for** (**int** x = 0; x < 50; x++) {  **if** (**this**.ticket > 0) {  System.*out*.println("卖票，ticket = " + **this**.ticket--);  }  }  }  }  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  MyThread mt = **new** MyThread();  **new** Thread(mt).start();  **new** Thread(mt).start();  **new** Thread(mt).start();  }  } |

**面试题：**请解释多线程的两种实现方式及区别？分别编写程序以验证两种实现方式。

· 多线程的两种实现方式都需要一个线程的主类，而这个类可以实现Runnable接口或继承Thread类，不管使用何种方式都必须在子类之中覆写run()方法，此方法为线程的主方法；

· Thread类是Runnable接口的子类，而且使用Runnable接口可以避免单继承局限，以及更加方便的实现数据共享的概念。

· 程序实现：

|  |  |
| --- | --- |
| **Runnable接口：** | **Thread类：** |
| **class** MyThread **implements** Runnable { @Override  **public** **void** run() { // 线程的主方法  **// 线程操作方法**  }  } | **class** MyThread **extends** Thread {  @Override  **public** **void** run() { // 线程的主方法  **// 线程操作方法**  }  } |
| MyThread mt = **new** MyThread();  **new** Thread(mt).start(); | MyThread mt = **new** MyThread();  mt.start(); |

### 3.2.4、线程的操作状态（了解）

每一个线程对象实际上都拥有属于自己的运行状态，那么下面分别说明线程的每种运行状态的特点：

1、 所有的线程对象都必须通过关键字new进行创建；

2、 线程如果要进行启动则一定会调用Thread类的start()方法，但是代码可能会分先后顺序：

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

以上是启动了三个线程，虽然在代码上有先后调用start()方法的顺序，可是对于JVM而言，都表示着所有的线程将同时进入到就绪状态，等待执行。

3、 进入到就绪状态之后，将等待着CPU进行资源的抢占，抢占到了资源之后，线程会进如到运行状态，开始执行run()方法体之中所定义的代码，；

4、 每一个线程执行run()方法到一定的时间的时候会让出CPU资源，进入到阻塞状态，而后重新回到就绪状态等待下次资源调度并继续执行run()方法中的代码；

5、 如果全部方法执行完毕之后，将进入到线程的终止状态，并且不会再进入到就绪状态，直接结束。

## 3.3、线程的主要操作方法（理解）

线程对象的也是可以进行若干种操作的，而且所有的线程操作方法都在Thread类中定义，如果要想学习完整的方法，可以参考本人的书《java开发实战经典》，但是下面只讲解核心的几个方法。

### 3.3.1、线程的命名和取得

线程本身是属于不可见的运行状态的，即：每次操作的时候是无法预料的，所以如果要想在程序之中操作线程，唯一依靠的就是线程名称，而要想取得和设置线程的名称可以使用如下的方法：

· **构造方法：**public Thread(Runnable target, String name)；

· **设置名字：**public final void setName(String name)；

· **取得名字：**public final String getName()。

但是由于线程的状态不确定，所以每次可以操作的都是正在执行run()方法的线程，那么取得当前线程对象的方法：public static Thread currentThread()。

**范例：**线程的命名和取得

|  |
| --- |
| **package** cn.mldn.demo;  **class** MyThread **implements** Runnable { // 线程的主体类  @Override  **public** **void** run() { // 线程的主方法  System.*out*.println(Thread.*currentThread*().getName());  }  }  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  MyThread mt = **new** MyThread();  **new** Thread(mt).start(); // Thread-0  **new** Thread(mt).start(); // Thread-1  **new** Thread(mt).start(); // Thread-2  **new** Thread(mt,"线程A").start(); // 线程A  **new** Thread(mt,"线程B").start(); // 线程B  }  } |

如果说现在为线程设置名字的话，那么会使用用户定义的名字，而如果没有设置线程名称，会自动的为其分配一个名称，这一点操作和之前讲解的static命名类似。

**范例：**观察如下代码

|  |
| --- |
| **package** cn.mldn.demo;  **class** MyThread **implements** Runnable { // 线程的主体类  @Override  **public** **void** run() { // 线程的主方法  System.*out*.println(Thread.*currentThread*().getName());  }  }  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  MyThread mt = **new** MyThread();  **new** Thread(mt, "线程").start(); // 启动线程，并设置名字，线程  mt.run(); // 在主方法之中，直接调用类的run()方法，main  }  } |

通过如上的程序可以发现，原来一个主方法也是一个线程，但是有一个问题出现了，现在一直是讲解了多线程，但是线程一定要依附于进程，进程在那里啊？

每一次使用java命令执行一个类的时候就表示启动了一个JVM的进程，而主方法是这个进程上的一个线程而已。

**问题：**一个JVM进程启动的时候至少启动几个线程呢？

两个线程：main、gc。

### 3.3.2、线程的休眠

线程的休眠指的是让程序的执行速度变慢一些，方法：public static void sleep(long millis) throws InterruptedException，设置的休眠单位是毫秒。

|  |
| --- |
| **package** cn.mldn.demo;  **class** MyThread **implements** Runnable { // 线程的主体类  @Override  **public** **void** run() { // 线程的主方法  **for** (**int** x = 0; x < 100; x++) {  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.*out*.println(Thread.*currentThread*().getName() + "，x = " + x);  }  }  }  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  MyThread mt = **new** MyThread();  **new** Thread(mt, "线程A").start();  **new** Thread(mt, "线程B").start();  **new** Thread(mt, "线程C").start();  **new** Thread(mt, "线程D").start();  **new** Thread(mt, "线程E").start();  }  } |

这个时候由于电脑的执行速度原因，所有的线程先后顺序并不容易发现，但是可以发现休眠了之后，程序运行速度变慢了。

### 3.3.3、线程的优先级

从理论上讲，线程的优先级越高，越有**可能**先执行。如果要想操作线程的优先级有如下两个方法：

· **设置线程的优先级：**public final void setPriority(int newPriority)；

· **取得线程的优先级：**public final int getPriority()；

发现设置和取得优先级的时候都是利用了一个int型数据的操作，而这个int型数据有三种取值：

· **最高优先级：**public static final int MAX\_PRIORITY，10；

· **中等优先级：**public static final int NORM\_PRIORITY，5；

· **最低优先级：**public static final int MIN\_PRIORITY，1；

**范例：**设置优先级

|  |
| --- |
| **package** cn.mldn.demo;  **class** MyThread **implements** Runnable { // 线程的主体类  @Override  **public** **void** run() { // 线程的主方法  **for** (**int** x = 0; x < 10; x++) {  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.*out*.println(Thread.*currentThread*().getName() + "，x = " + x);  }  }  }  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  MyThread mt = **new** MyThread();  Thread t1 = **new** Thread(mt,"线程A") ;  Thread t2 = **new** Thread(mt,"线程B") ;  Thread t3 = **new** Thread(mt,"线程C") ;  t3.setPriority(Thread.*MAX\_PRIORITY*) ;  t1.setPriority(Thread.*MIN\_PRIORITY*) ;  t2.setPriority(Thread.*MIN\_PRIORITY*) ;  t1.start() ;  t2.start() ;  t3.start() ;  }  } |

**问题：**主线程的优先级是什么呢？

|  |
| --- |
| **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  System.*out*.println(Thread.*currentThread*().getPriority());  }  } |

发现主线程的优先级是5，是中等级别。

## 3.4、线程的同步与死锁（理解）

### 3.4.1、同步问题

所谓的同步问题指的是多个线程操作同一资源时所带来的信息的安全性问题，例如，下面模拟一个简单的卖票程序，要求有5个线程，卖6张票。

|  |
| --- |
| **package** cn.mldn.demo;  **class** MyThread **implements** Runnable { // 线程的主体类  **private** **int** ticket = 6;  @Override  **public** **void** run() { // 线程的主方法  **for** (**int** x = 0; x < 10; x++) {  **if** (**this**.ticket > 0) {  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.*out*.println(Thread.*currentThread*().getName()  + "卖票，ticket = " + **this**.ticket--);  }  }  }  }  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  MyThread mt = **new** MyThread();  **new** Thread(mt, "票贩子A").start();  **new** Thread(mt, "票贩子B").start();  **new** Thread(mt, "票贩子C").start();  **new** Thread(mt, "票贩子D").start();  **new** Thread(mt, "票贩子E").start();  }  } |

这个时候发现操作的数据之中出现了负数，这个就可以理解为不同步问题。

如果说现在只剩下最后一张票了，一个线程判断条件满足，但是再它还没有修改票数之后，其他线程也同时通过了if判断，所以最终修改票数的时候就变成了负数。

如果现在要想增加这个锁，在程序之中就可以通过两种方式完成：一种是同步代码块，另外一种就是同步方法。

**实现一：**同步代码块，使用synchronized关键字定义的代码块就称为同步代码块，但是在进行同步的操作之中必须设置一个要同步的对象，而这个对象应该理解为当前对象：this。

|  |
| --- |
| **package** cn.mldn.demo;  **class** MyThread **implements** Runnable { // 线程的主体类  **private** **int** ticket = 6;  @Override  **public** **void** run() { // 线程的主方法  **for** (**int** x = 0; x < 10; x++) {  **synchronized** (**this**) { // 同步代码块  **if** (**this**.ticket > 0) {  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.*out*.println(Thread.*currentThread*().getName()  + "卖票，ticket = " + **this**.ticket--);  }  }  }  }  }  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  MyThread mt = **new** MyThread();  **new** Thread(mt, "票贩子A").start();  **new** Thread(mt, "票贩子B").start();  **new** Thread(mt, "票贩子C").start();  **new** Thread(mt, "票贩子D").start();  **new** Thread(mt, "票贩子E").start();  }  } |

**方式二：**同步方法

|  |
| --- |
| **package** cn.mldn.demo;  **class** MyThread **implements** Runnable { // 线程的主体类  **private** **int** ticket = 6;  @Override  **public** **void** run() { // 线程的主方法  **for** (**int** x = 0; x < 10; x++) {  **this**.sale() ;  }  }  **public** **synchronized** **void** sale() {  **if** (**this**.ticket > 0) {  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.*out*.println(Thread.*currentThread*().getName()  + "卖票，ticket = " + **this**.ticket--);  }  }  }  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  MyThread mt = **new** MyThread();  **new** Thread(mt, "票贩子A").start();  **new** Thread(mt, "票贩子B").start();  **new** Thread(mt, "票贩子C").start();  **new** Thread(mt, "票贩子D").start();  **new** Thread(mt, "票贩子E").start();  }  } |

但是在此处需要说明的一个问题：加入同步之后明显比不加入同步慢许多，所以同步的代码性能会很低，但是数据的安全性会高。

### 3.4.2、死锁

同步就是指一个线程要等待另外一个线程执行完毕才会继续执行的一种操作形式，但是如果在一个操作之中都是在互相等着的话，那么就会出现死锁问题。

**范例：**下面简单的模拟一个死锁程序的样式

|  |
| --- |
| **package** cn.mldn.demo;  **class** YuShi {  **public** **synchronized** **void** say(FuXie f) {  System.*out*.println("玉史：给我30亿欧圆，放了你儿子。");  f.get() ;  }  **public** **synchronized** **void** get() {  System.*out*.println("玉史终于得到了赎金，放了儿子，为了下次继续绑架。");  }  }  **class** FuXie {  **public** **synchronized** **void** say(YuShi y) {  System.*out*.println("付谢：放了我儿子，我给你30亿欧圆，不见人不给钱。") ;  y.get() ;  }  **public** **synchronized** **void** get() {  System.*out*.println("付谢救回了自己的儿子，于是开始哭那30亿。");  }  }  **public** **class** DeadLock **implements** Runnable {  **static** YuShi *ys* = **new** YuShi() ;  **static** FuXie *fx* = **new** FuXie() ;  **public** **static** **void** main(String[] args) {  **new** DeadLock() ;  }  **public** DeadLock() {  **new** Thread(**this**).start() ;  *ys*.say(*fx*) ;  }  @Override  **public** **void** run() {  *fx*.say(*ys*) ;  }  } |

死锁是在日后多线程程序开发之中经常会遇见的问题，而以上的代码并没有任何的实际意义，大概可以理解死锁的操作形式就可以了，不用去研究程序。

**面试题：**请问多个线程操作同一资源的时候要考虑到那些，会带来那些问题？

多个线程访问同一资源的时候一定要考虑到同步的问题，但是过多的同步会带来死锁。

## 3.5、线程间的经典操作案例（理解）

在多线程的开发之中存在一种称为“生产者和消费者的程序”，这个程序的主要功能是生产者负责生产一些内容，每当生产完成之后，会由消费者取走全部内容，那么现在假设要生产的是如下两种数据：

· 数据一：title = 2012.12.21日，content = 世界末日；

· 数据二：title = 付谢，content = 打扫卫生迎接末日。

现在对于这样的程序，可以使用如下的一些基本模型实现。

|  |
| --- |
| **package** cn.mldn.demo;  **class** Message {  **private** String title ;  **private** String content ;  **public** **void** setTitle(String title) {  **this**.title = title;  }  **public** **void** setContent(String content) {  **this**.content = content;  }  **public** String getTitle() {  **return** title;  }  **public** String getContent() {  **return** content;  }  }  **class** Productor **implements** Runnable {  **private** Message msg = **null** ;  **public** Productor(Message msg) {  **this**.msg = msg ;  }  @Override  **public** **void** run() {  **for** (**int** x = 0; x < 50; x++) {  **if** (x % 2 == 0) {  **this**.msg.setTitle("2012.12.21") ;  **try** {  Thread.*sleep*(100) ;  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **this**.msg.setContent("世界末日") ;  } **else** {  **this**.msg.setTitle("付谢") ;  **try** {  Thread.*sleep*(100) ;  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **this**.msg.setContent("打扫卫生迎接末日") ;  }  }  }  }  **class** Customer **implements** Runnable {  **private** Message msg = **null** ;  **public** Customer(Message msg) {  **this**.msg = msg ;  }  @Override  **public** **void** run() {  **for** (**int** x = 0; x < 50; x++) {  **try** {  Thread.*sleep*(100) ;  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.*out*.println(**this**.msg.getTitle() + " --> " + **this**.msg.getContent());  }  }  }  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Message msg = **new** Message() ;  **new** Thread(**new** Productor(msg)).start() ;  **new** Thread(**new** Customer(msg)).start() ;  }  } |

但是，以上的代码模型出现了如下的两严重问题：

· 数据错位了；

· 出现了重复取出和重复设置的问题。

### 3.5.1、解决数据错位问题：依靠同步就可以解决

只要对设置和取得加上同步应用，就可以解决数据的错位的操作问题，下面，对代码进行修改。

|  |
| --- |
| **package** cn.mldn.demo;  **class** Message {  **private** String title ;  **private** String content ;  **public** **synchronized** **void** set(String title,String content) {  **this**.title = title ;  **try** {  Thread.*sleep*(200) ;  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **this**.content = content ;  }  **public** **synchronized** **void** get() {  **try** {  Thread.*sleep*(100) ;  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.*out*.println(**this**.title + " --> " + **this**.content);  }  }  **class** Productor **implements** Runnable {  **private** Message msg = **null** ;  **public** Productor(Message msg) {  **this**.msg = msg ;  }  @Override  **public** **void** run() {  **for** (**int** x = 0; x < 50; x++) {  **if** (x % 2 == 0) {  **this**.msg.set("2012.12.21", "世界末日");  } **else** {  **this**.msg.set("付谢", "打扫卫生迎接末日");  }  }  }  }  **class** Customer **implements** Runnable {  **private** Message msg = **null** ;  **public** Customer(Message msg) {  **this**.msg = msg ;  }  @Override  **public** **void** run() {  **for** (**int** x = 0; x < 50; x++) {  **this**.msg.get() ;  }  }  }  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Message msg = **new** Message() ;  **new** Thread(**new** Productor(msg)).start() ;  **new** Thread(**new** Customer(msg)).start() ;  }  } |

这个时候的确解决了数据的错位的问题，但同时新的问题又来了：发现数据的重复问题更严重了。

### 3.5.2、解决数据的重复设置和重复取出

要想解决重复的问题需要等待及唤醒机制，而这一机制的实现只能依靠Object类完成，在Object类之中定义了以下的三个方法完成线程的操作：

· 等待：public final void wait() throws InterruptedException；

· 唤醒第一个等待线程：public final void notify()；

· 唤醒全部等待线程：public final void notifyAll()。

对于唤醒的两个操作：notify()是按照等待顺序进行了唤醒，而使用了notifyAll()则表示所有等待的线程都会被唤醒，那个线程的优先级高，那个线程就先执行。

**范例：**修改Message类，解决数据的重复设置和重复取出的操作

|  |
| --- |
| **class** Message {  **private** String title ;  **private** String content ;  **private** **boolean** flag = **true** ;  // flag == true：表示可以生产，但是不能取走  // flag == false：表示可以取走，但是不能生产  **public** **synchronized** **void** set(String title,String content) {  **if** (**this**.flag == **false**) { // 已经生产过了，不能生产  **try** {  **super**.wait() ;  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  **this**.title = title ;  **try** {  Thread.*sleep*(200) ;  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **this**.content = content ;  **this**.flag = **false** ;  **super**.notify() ;  }  **public** **synchronized** **void** get() {  **if** (**this**.flag == **true**) { // 不能取走  **try** {  **super**.wait() ;  } **catch** (InterruptedException e) {  e.printStackTrace();  } }  **try** {  Thread.*sleep*(100) ;  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.*out*.println(**this**.title + " --> " + **this**.content);  **this**.flag = **true** ; // 已经取走了，可以继续生产  **super**.notify() ;  }  } |

**面试题：**请解释sleep()和wait()的区别？

· sleep()是Thread类定义的static方法，表示线程休眠，休眠到一定时间后自动唤醒；

· wait()是Object类定义的方法，表示线程等待，一直到执行了notify()或notifyAll()之后才结束等待。

# 4、总结

1、 多线程的两种实现方式及区别；

2、 理解同步与死锁的概念；

3、 Object类对多线程支持的三个方法。

# 5、预习任务

StringBuffer类、Date类、SimpleDateFormat、System、Runtime、Math、Random、BigInteger、BigDecimal、比较器、正则表达式。

# 6、学习问题解决

1、 对象比较假设类都是同一个类。主方法及输出结果如下

|  |  |
| --- | --- |
| public class Hello{  public static void main(String args[]){  Emp emp1 = new Emp(22,"李四","程序员") ;  Emp emp2 = new Emp(22,"李四","程序员") ;  if(emp1.equals(emp2)){  System.out.println("不是同一个人") ;  }else{  System.out.println("是同一个人") ;  }  } | public class Hello{  public static void main(String args[]){  Emp emp1 = new Emp(22,"李四","程序员") ;  Emp emp2 = new Emp(22,"李四","程序员") ;  if(emp1.equals(emp2)){  System.out.println("不是同一个人") ;  }else{  System.out.println("是同一个人") ;  }  } |
| 是同一个人 | 不是同一个人 |

不明白的是，既然是对象比较那么主方法中的两个输出语句位置应该无所谓先后。而且结果都是同一个人才对，那么所有的属性都是相同的情况下，为什么会出现两次输出结果不一样呢？

2、 数据类型的转换是转型吗？

对象多态性：向上转型和向下转型；

数据类型转换：String向基本型转换，或者是基本型向String转换，称为类型转换。

3、 public class Test{

public static void main(String args[]){

String str1 = "hello";

System.out.println("fun()方法调用之前："+str1);

fun(str1);

System.out.println("fun()方法调用之后："+str1);

}

public static void fun(String str){

str="mldn";

}

}

**为什么输出的是 hello。**

核心：如果看不懂图，把String基本数据类型那样，数值传递（就仿佛是数值拷贝一样）。

4、 关于 equals的问题：

|--equals()用于字符串比较：

String str1 = "Hello" ;

String str2 = new String("Hello") ;

System.out.println(str1.equals(str2)) ;--->true；

|--equals()用于对象比较：

Person per1 = new Person("张三",20) ;

Person per2 = new Person("张三",20) ;

System.out.println(per1.equals(per2)) ; --->false；

问题来了，String类既然也是继承自Object类，字符串比较之前也没有对Object的equals()进行覆写，怎么就相当于内容的比较了呢？？

String类覆写了Object类的equals()方法。

5、 匿名内部类不是很懂？

匿名内部类根本就没用，2个月之后用，匿名内部类就是指一个接口或抽象类的子类只使用一次的情况下所采用的技术。

|  |
| --- |
| **package** cn.mldn.demo;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  **new** Thread(**new** Runnable() **{**  **@Override**  **public void run() {**  **for (int x = 0; x < 10; x++) {**  **System.*out*.println(Thread.*currentThread*().getName()**  **+ "，x = " + x);**  **}**  **}**  **}**).start();  }  } |

复习：复习核心代码，并根据代码推导概念。

重点的概念认真复习，凡是明确给出日后再使用的技术，肯定到时候一用就会了。

之前也强调过了，自己编写的代码之中，现阶段是不去考虑内部类问题的。

# 7、测试题讲解

1、 请写出String类对象两种实例化方式的区别？

2、 请写出字符串比较的两种方式及区别？

3、 编写一个程序，判断一个字符串是否全部由数字所组成；、

4、 〖SQL〗查询出公司每个工资等级的人数、平均工资、最高和最低工资；

5、 〖SQL〗显示出emp表的5~10行记录。