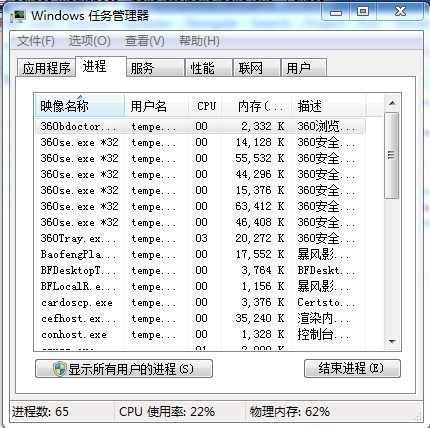
**线程与进程**

在操作系统的定义中，进程值得是一次程序的完整运行，这个运行的过程中内存、处理器、IO等资源操作都要为这个进程进行服务。



在同一个时间端上，会有多进程轮流抢占资源，但是某一个时间点上只会有一个进程运行。

线程是在进程基础上进一步的划分结果。即：一个进程上可以同时创建多个线程。

线程是比进程更快的处理单元，而且所占的资源也小。那么多线程的应用也就是性能最高的应用。

线程的存在离不开进程。进程如果消失后线程一定会消失，反之如果线程消失后进程未必会消失。

**多线程的实现**

如果要想在Java中实现多线程有两种途径：

·继承Thread类；

·实现Runnable接口（Callbale接口）；

**3.1、继承Thread类**

Thread类是一个支持多线程的功能类，只要有一个自雷就可以实现多线程的支持。

|  |
| --- |
| **class**MyThread**extends** Thread{//这是一个多线程的操作类    } |

所有程序的起点是main()方法，但是所有线程也一定要有一个自己的起点，那么这个起点就是run(0方法，也就是说在多线程的每个主体类之中都必须覆写Thread类中的run方法。

|  |
| --- |
| public void run(){ } |

这个方法上没有返回值，那么也就表示了线程一旦开始就要一直执行，不能都返回内容。

|  |
| --- |
| **class**MyThread**extends** Thread{//这是一个多线程的操作类  **private** String name;  **public**MyThread(String name) {  **this**.name = name;  }  @Override  **publicvoid** run() {  // 覆写run()方法，作为线程的主题操作方法  **for** (**int**i = 0; i< 200; i++) {  System.***out***.println(**this**.name + "--->" + i);  }  }  } |

本线程类的功能是进行循环输出的操作。所有的线程与进程是一样的，都必须轮流去抢占资源，所以多线程的执行应该是多个线程彼此交替执行。也就是说如果直接调用了run()方法，那么并不能够启动多线程，多线程启动的唯一方法就是Thread类中的start()方法：public void start()（调用此方法执行的方法体是run()方法定义的）

|  |
| --- |
| **publicclass**TestDemo {    **publicstaticvoid** main(String[] args) {  MyThreadmta = **new**MyThread("线程A");  MyThreadmtb = **new**MyThread("线程B");  MyThreadmtc = **new**MyThread("线程C");  mta.start();  mtb.start();  mtc.start();  }  } |

此时每一个线程交替执行。

疑问？为什么多线程启动不是调用run()而必须调用start()?

打开Java的源码，观察下start方法：

|  |
| --- |
| **publicsynchronizedvoid** start() {  **if** (threadStatus != 0)  **thrownew**IllegalThreadStateException();  group.add(**this**);  **boolean**started = **false**;  **try** {  start0();  started = **true**;  } **finally** {  **try** {  **if** (!started) {  group.threadStartFailed(**this**);  }  } **catch** (Throwableignore) {  }  }  }  **privatenativevoid**start0(); |

首先方法在Thread类的start()方法里面存在有一个“IllegalThreadStateException”异常抛出。本方法里面使用了throw抛出异常，按照道理来讲应该使用try…catch处理，或者在start()方法声明上使用throws声明，但是此处并没有这样的代码，以为磁异常属于RuntimeException的自雷，属于选择性处理。如果某一个线程对象重复进行了启动，那么就会抛出此异常。

发现在start()方法中药调用一个start()方法，而且此方法的机构与抽象方法类似。使用了native声明，在Java开发里面有一门技术称为JNI技术（Java Native Interface），这门技术的特点：是使用Java调用本机操作系统提供的函数。但是这样的技术有一个缺点，不能够离开特定的操作系统。

如果要想线程能够执行，需要操作系统来进行资源的分配，所以此操作严格来讲主要是由JVM负责根据不同的操作系统而实现的。

**即:使用Thread类的stat()方法不仅仅需要启动多线程的执行代码，还要去根据不同的操作系统进行资源的分配。**

**3.2、实现Runnable接口**

虽然Thread类可以实现多线程的主体类定义，但是它有一个问题，Java具有单继承局限，正因为如此在任何情况下针对类的继承都应该回避的问题，那么多线程也一样，为了解决单继承的限制，Java里面专门提供了Runnable接口。此接口定义如下：

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

在接口里面任何的方法都是public权限，不存在默认权限

那么只需要让一个雷实现Runnable接口即可，并且也需要覆写run()方法。

|  |
| --- |
| **class**MyThread**implements**Runnable{//这是一个多线程的操作类  **private** String name;  **public**MyThread(String name) {  **this**.name = name;  }  @Override  **publicvoid**run() {  // 覆写run()方法，作为线程的主题操作方法  **for** (**int**i = 0; i< 200; i++) {  System.***out***.println(**this**.name + "--->" + i);  }  }  } |

与继承Thread雷相比，此时的MyThread类在结构上与之前没有区别的，但是有一点是由严重区别的，如果此时继承了Thread类，那么可以直接继承start()方法，但是如果实现的是Runnable接口并没有start()方法可以被继承。

不管何种情况下，如果要想启动多线程一定一开Thread类完成，在Thread类里面定义有一下的构造方法：

·构造方法:public Thread(Runnable target),接受的是Runnable接口对象；

范例：启动多线程

|  |
| --- |
| **publicclass**TestDemo {    **publicstaticvoid** main(String[] args) {  MyThreadmta = **new**MyThread("线程A");  MyThreadmtb = **new**MyThread("线程B");  MyThreadmtc = **new**MyThread("线程C");  **new** Thread(mta).start();  **new** Thread(mtb).start();  **new** Thread(mtc).start();  }  } |

此时就避免了单继承局限，也就是说在实际的工作中使用Runnable是最合适的。

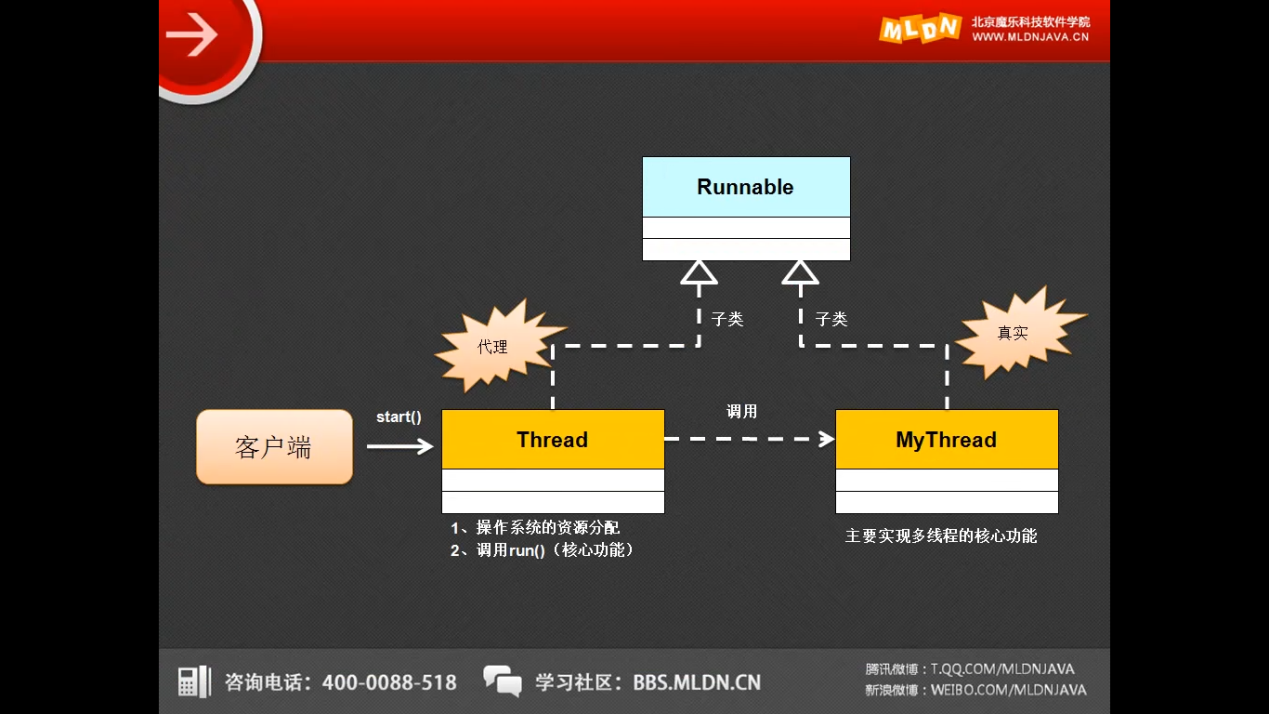
**3.3、多线程两种实现方式的区别（面试题）**

首先一定要明确的是，使用Runnable接口与Thread类相比，解决了单继承的定义局限，所以不管后面的区别与联系是什么，至少这一点上就已经下了死定义----如果要使用一定使用Runnable接口。

首先观察下Thread类的定义：

|  |
| --- |
| public class **Thread**  extends [Object](file:///D:\jdk-8u144-docs-all\docs\api\java\lang\Object.html)  implements [Runnable](file:///D:\jdk-8u144-docs-all\docs\api\java\lang\Runnable.html) |

发现Thread类实现了Runnable接口，那么这样一来程序就变为了以下的形式。



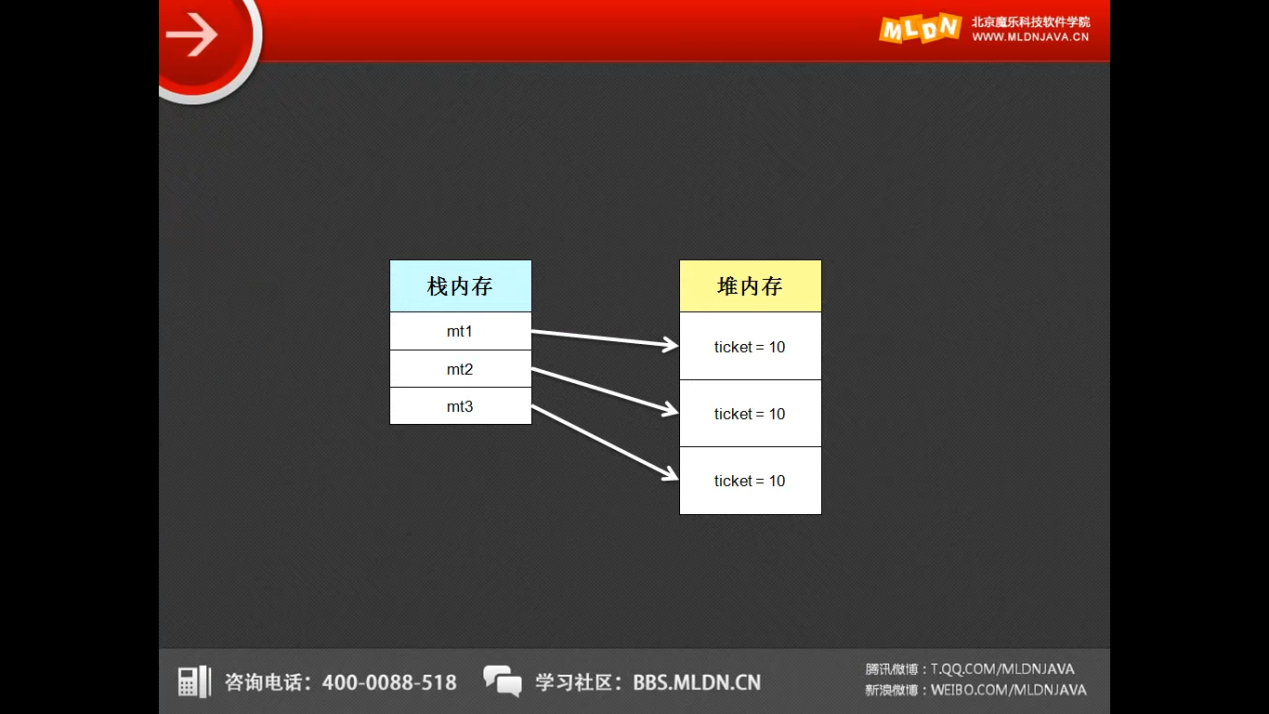
此时，整个的定义结构看起来非常想代理设计模式。如果是代理设计模式，客户端调用的代理类的方法也应该是接口里提供的方法，那么也应该是run()才对。

除了以上的联系之外，还有一点：使用Runnable接口可以比Thread类能够更好的描述出数据共享这一概念。此时的数据共享指的是多个线程访问统一资源的操作。

范例:观察代码（每一个线程对象都必须通过start()启动）

|  |
| --- |
| **package**cn.mldn.demo;  **class**MyThread**extends** Thread{  **privateint**ticket = 10;  **publicvoid** run(){  **for**(**int**x = 0;x<100 ; x++){  **if**(**this**.ticket>0){  System.***out***.println("卖票,ticket = " + **this**.ticket --);  }  }  }  }  **publicclass**TestDemo {  **publicstaticvoid** main(String[] args) {  MyThreadmt1 = **new**MyThread();  MyThreadmt2 = **new**MyThread();  MyThreadmt3 = **new**MyThread();  mt1.start();  mt2.start();  mt3.start();  }  } |

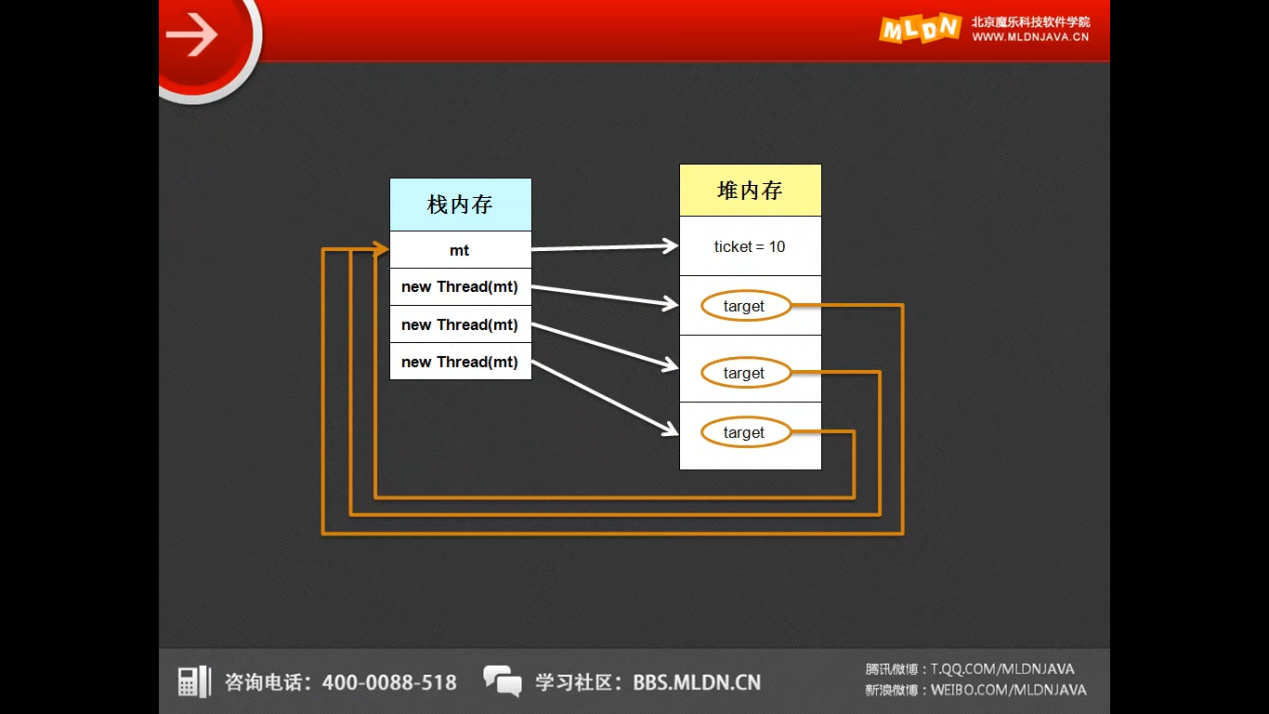
本程序声明了三个MyThread类的对象，并且分别调用可三次start()方法，启动线程对象。但是发现最终的结果是，每一个线程对象都在卖鸽子的10张票，因为此时的内存关系如下。



此时并不存在有数据共享这一概念

范例：利用Runnable来实现

|  |
| --- |
| **package**cn.mldn.demo;  **class**MyThread**implements** Runnable{  **privateint**ticket = 10;  **publicvoid** run(){  **for**(**int**x = 0;x<100 ; x++){  **if**(**this**.ticket>0){  System.***out***.println("卖票,ticket = " + **this**.ticket --);  }  }  }  }  **publicclass**TestDemo {  **publicstaticvoid** main(String[] args) {  MyThreadmt1 = **new**MyThread();  **new** Thread(mt1).start();  **new** Thread(mt1).start();  **new** Thread(mt1).start();  }  } |



此时也属于三个线程对象，可是唯一的区别是，这三个线程对象都直接占用用一个MyThread对象的引用，也就是说这三个线程对象都直接访问同一个数据资源。

面试题：请解释Thread类与Runnable接口实现多线程的区别？（面试题：请解释多线程两种实现方式的区别）

·Thread类是Runnable接口的子类，使用Runnable接口实现多线程可以表面单继承局限；

·Runnable接口实现的多线程可以比Thread类实现的多线程更加清楚的描述数据共享的概念。

面试题：请斜侧多线程两种实现操作。

把Thread类继承的方式和Runnable接口实现的方式都写出来

**3.4、第三种实现方式（理解）**

使用Runnable接口实现的多线程可以表面单继承局限，但是有一个问题，Runnable接口中的run()方法不能返回操作结果。为了解决这样的矛盾，提供了一个新的接口java.util.concurrent.Callable接口。

|  |
| --- |
| **publicinterface**Callable<V>{  **public** V call()  **throws** Exception;  } |

Call()方法执行完线程的主题功能之后可以返回一个结果，而返回结果的类型由Callable接口上的泛型决定

范例：定义一个线程主题类

|  |
| --- |
| **import**java.util.concurrent.Callable;  **class**MyThread**implements** Callable<String>{  **privateint**ticket = 10;  **public**String call() **throws** Exception{  **for**(**int**x = 0;x<100 ; x++){  **if**(**this**.ticket>0){  System.***out***.println("卖票,ticket = " + **this**.ticket --);  }  }  **return**"票已卖光";  }  } |

此时观察Thread类里面返现并没有直接支持Callable接口的多线程应用。

从JDK1.5开始提供有java.util.concurrent.FutureTask<V>类。这个类主要是负责Callable接口对象操作的。这个接口的定义结构是：

|  |
| --- |
| public class FutureTask<V>  extends Object  implements RunnableFuture<V> |
| public interface RunnableFuture<V>  extends Runnable, Future<V> |

在FutureTask里面定义有如下构造方法：publicFutureTask(Callable<V> callable)

接收的目的只有一个，那么就是取得call()方法的返回结果。

|  |
| --- |
| **publicclass**TestDemo {  **publicstaticvoid** main(String[] args) **throws**InterruptedException, ExecutionException {  MyThreadmt1 = **new**MyThread();  MyThreadmt2 = **new**MyThread();  FutureTask<String>task1 = **new**FutureTask<String>(mt1);//目的是为了取得call()的返回结果  FutureTask<String>task2 = **new**FutureTask<String>(mt2);//目的是为了取得call()的返回结果  //FutureTask是Runnable接口子类，所以可以使用Thread类的构造老接收task对象  **new** Thread(task1).start(); //启动多线程  **new** Thread(task2).start(); //启动多线程  //多线程执行完毕后可以取得内容，依靠FutureTask的父接口Future中的get()方法完成  System.***out***.println("task1 result:" + task1.get());  System.***out***.println("task1 result:" + task2.get());  }  } |

最麻烦的问题在于需要接收返回值信息，并且又要与原始的多线程实现靠拢（想Thread类靠拢）

**3.1、多线程的常用操作方法（命名和取得）**

所有的线程程序的执行，每一次都是不同的运行结果，因为它会根据自己的情况进行资源抢占，所以如果要想区分每一个线程，那么久必须依靠线程的名字。对于线程名字一般而言会在其启动之前进行定义，不建议对已经启动的线程进行更改名称，或者是为不同的线程设置重名的情况。

如果要想进行线程名称的操作，可以使用Thread类的如下方法：

·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()

对于线程名字的操作会出现一个问题，这些方法是属于Thread类里面的，可是如果换回到线程类（Runnable）子类，这个类里面并没有继承Thread类，如果要想取得我们的线程名字，那么能够取得的就是当前执行本方法的线程名字。所以在Thread里面提供有一个方法：

·取得当前线程对象：public static Thread currentThread()

范例：观察线程的命名

|  |
| --- |
| **package**cn.mldn.demo;  **class**MyThread**implements** Runnable{//这是一个多线程的操作类  @Override  **publicvoid** run() {  // 覆写run()方法，作为线程的主题操作方法  System.***out***.println(Thread.*currentThread*().getName());  }  }  **publicclass**TestDemo {    **publicstaticvoid** main(String[] args) {  MyThreadmt = **new**MyThread();  **new** Thread(mt).start();  **new** Thread(mt).start();  **new** Thread(mt).start();  }  } |

如果在实例化Thread类对象的时候没有为其设置名字，那么会自动的进行编号命名，也就是说保证线程对象的名字不重复。

范例：设置名字

|  |
| --- |
| **package**cn.mldn.demo;  **class**MyThread**implements** Runnable{//这是一个多线程的操作类  @Override  **publicvoid** run() {  // 覆写run()方法，作为线程的主题操作方法  System.***out***.println(Thread.*currentThread*().getName());  }  }  **publicclass**TestDemo {    **publicstaticvoid** main(String[] args) {  MyThreadmt = **new**MyThread();  **new** Thread(mt,"xiancheng A").start();  **new** Thread(mt).start();  **new** Thread(mt,"xiancheng b").start();  **new** Thread(mt).start();  **new** Thread(mt).start();  }  } |

观察如下的代码执行：

|  |
| --- |
| **package**cn.mldn.demo;  **class**MyThread**implements** Runnable{//这是一个多线程的操作类  @Override  **publicvoid** run() {  // 覆写run()方法，作为线程的主题操作方法  System.***out***.println(Thread.*currentThread*().getName());  }  }  **publicclass**TestDemo {    **publicstaticvoid** main(String[] args) {  MyThreadmt = **new**MyThread();  **new** Thread(mt,"xiancheng A").start();  mt.run();//直接调用run方法，main  }  } |
| main  xiancheng A |

原来主方法就是一个线程（mian线程），那么所有在主方法桑创建的线程实际上都可以将其表示为子线程。

通过以上的代码可以发现，线程实际上一直都存在（主方法就是主线程），可是进程去哪里了呢？

每当使用java命令去解释一个程序类的时候，对于操作系统而言，都相当于启动了一个新的进程，而main只是这个新进程上的一个子线程而已。

提问：每一个JVM进程启动的时候至少启动几个线程？

·main线程：程序的主要执行，以及启动子线程

·gc线程：负责垃圾收集。

**3.2、多线程的常用操作方法（休眠）**

所谓的线程休眠指的就是让线程的执行速度稍微变慢一点。休眠的方法：

·public static void sleep(long millis) throws InterruptedException

范例：观察休眠的特点

|  |
| --- |
| **package**cn.mldn.demo;  **class**MyThread**implements** Runnable{//这是一个多线程的操作类  @Override  **publicvoid** run() {  // 覆写run()方法，作为线程的主题操作方法  **for**(**int**i = 0;i<100;i++) {  **try** {  Thread.*sleep*(500);  } **catch** (InterruptedExceptione) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  System.***out***.println(Thread.*currentThread*().getName()+ ",i = " + i);  }    }  }  **publicclass**TestDemo {    **publicstaticvoid** main(String[] args) {  MyThreadmt = **new**MyThread();  **new** Thread(mt,"xiancheng A").start();  }  } |

因为每一次执行到run（）方法的线程对象都必须进行休眠，所以执行的速度就会变慢。

默认情况下，在休眠的时候如果设置了多个线程对象，那么所有的线程对象将一起进入到run()方法（所谓的一起进入实际上是因为先后顺序实在是太短了，但实际上有区别。）

**3.3、多线程的常用操作方法（线程的优先级）**

所谓的优先级指的是越高的优先级，越有可能先执行。在Thread类里面提供有以下的两个方法进行优先级操作

·设置优先级：public final void setPriority(intnewPriority)

·取得优先级：public final intgetPriority()

发现设置和取得优先级都是使用了int数据类型，对此内容有三种取值

·最高优先级：public static final int MAX\_PRIORITY

·中等优先级：public static final int NORM\_PRIORITY

·最低优先级：public static final int MIN\_PRIORITY

范例：测试

|  |
| --- |
| **package**cn.mldn.demo;  **class**MyThread**implements** Runnable{//这是一个多线程的操作类  @Override  **publicvoid** run() {  // 覆写run()方法，作为线程的主题操作方法  **for**(**int**i = 0;i<10;i++) {  **try** {  Thread.*sleep*(500);  } **catch** (InterruptedExceptione) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  System.***out***.println(Thread.*currentThread*().getName()+ ",i = " + i);  }    }  }  **publicclass**TestDemo {    **publicstaticvoid** main(String[] args) {  MyThreadmt = **new**MyThread();  Thread t1 = **new** Thread(mt,"xiancheng A");  Thread t2 = **new** Thread(mt,"xiancheng B");  Thread t3 = **new** Thread(mt,"xiancheng C");  t1.setPriority(Thread.***MAX\_PRIORITY***);  t2.setPriority(Thread.***MIN\_PRIORITY***);  t3.setPriority(Thread.***MIN\_PRIORITY***);  t1.start();  t2.start();  t3.start();  }  } |

范例：主线程的优先级是多少

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

主线程的优先级是中等优先级。

**3.1、线程的同步与死锁（同步问题引出）**

实际上所谓的同步指的就是多个线程访问同意资源是所需要考虑到的问题。

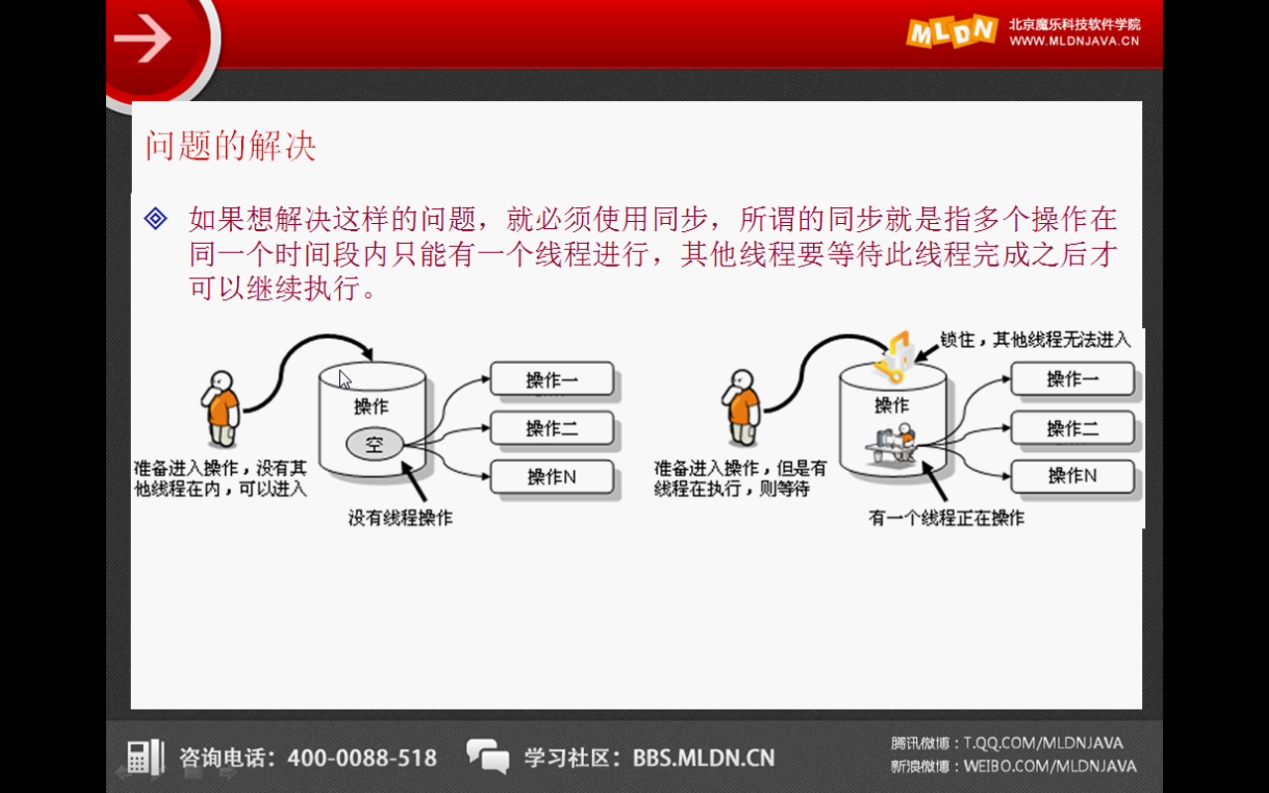
范例：观察费同步情况下的操作

|  |
| --- |
| **package**cn.mldn.demo;  **class**MyThread**implements** Runnable{//这是一个多线程的操作类  **privateint**ticket = 5;  @Override  **publicvoid** run() {  // 覆写run()方法，作为线程的主题操作方法  **for**(**int**i = 0;i<20;i++) {  **if**(**this**.ticket> 0){  System.***out***.println(Thread.*currentThread*().getName()+ "买票 = " + **this**.ticket --);  }    }    }  }  **publicclass**TestDemo {    **publicstaticvoid** main(String[] args) {  MyThreadmt = **new**MyThread();  **new** Thread(mt,"xiancheng A").start();  **new** Thread(mt,"xiancheng B").start();  **new** Thread(mt,"xiancheng C").start();  **new** Thread(mt,"xiancheng D").start();  **new** Thread(mt,"xiancheng E").start();  }  } |
| xiancheng A买票 = 5  xiancheng E买票 = 2  xiancheng C买票 = 3  xiancheng D买票 = 4  xiancheng B买票 = 5  xiancheng A买票 = 1 |

执行结果中出现了两个5，那么这就是不同步的情况，

**3.2、线程的同步与死锁（同步操作）**

通过观察可以发现以上程序锁带来的最大问题在：判断和修改数据是分开完成的，即：某几个线程可一同事执行这些功能。



在Java里面如果要想实现线程的同步可以使用synchronized关键字。而这个关键字可以通过两种方式使用:

·一种是同步代码块；

·另一种是同步方法。

在java里面有四种代码块：普通代码块、构造块、静态块、同步块

范例：观察同步块

|  |
| --- |
| **package**cn.mldn.demo;  **class**MyThread**implements** Runnable{//这是一个多线程的操作类  **privateint**ticket = 5;  @Override  **publicvoid** run() {  // 覆写run()方法，作为线程的主题操作方法  **for**(**int**i = 0;i<20;i++) {  **synchronized**(**this**) {//当前操作每次只允许一个对象进入  **if**(**this**.ticket> 0){  System.***out***.println(Thread.*currentThread*().getName()+ "买票 = " + **this**.ticket --);  }  }  }    }  }  **publicclass**TestDemo {    **publicstaticvoid** main(String[] args) {  MyThreadmt = **new**MyThread();  **new** Thread(mt,"xiancheng A").start();  **new** Thread(mt,"xiancheng B").start();  **new** Thread(mt,"xiancheng C").start();  **new** Thread(mt,"xiancheng D").start();  **new** Thread(mt,"xiancheng E").start();  }  } |

范例：使用同步方法解决

|  |
| --- |
| **package**cn.mldn.demo;  **class**MyThread**implements** Runnable{//这是一个多线程的操作类  **privateint**ticket = 50;  @Override  **publicvoid** run() {  // 覆写run()方法，作为线程的主题操作方法  **for**(**int**i = 0;i<200;i++) {  **this**.sale();//调用同步方法  }    }  **publicsynchronizedvoid** sale() {//同步方法  **if**(**this**.ticket> 0){  System.***out***.println(Thread.*currentThread*().getName()+ "买票 = " + **this**.ticket --);  }  }  }  **publicclass**TestDemo {    **publicstaticvoid** main(String[] args) {  MyThreadmt = **new**MyThread();  **new** Thread(mt,"xiancheng A").start();  **new** Thread(mt,"xiancheng B").start();  **new** Thread(mt,"xiancheng C").start();  **new** Thread(mt,"xiancheng D").start();  **new** Thread(mt,"xiancheng E").start();  }  } |

**3.3、线程的同步与死锁（死锁）**

实际上通过分析可以发现，所谓的同步指的就是一个线程对象等待另外一个线程对象执行完毕后的操作形式。线程同步过多就有可能造成死锁。

面试题：请解释多个线程访问同意资源时需要考虑到哪些情况？有可能带来哪些问题？

·多个线程访问同一个资源时要处理好同步，可以使用同步代码块或同步方法解决

|-同步代码块：synchronized(锁定对象){代码}

|-同步方法:public synchronized 返回值方法名称(){代码}

·但是过多的使用同步，有可能造成死锁。

**3.1、综合实战：生产者和消费者（问题引出）**

生产者和消费者指的是两个不同的线程类对象，操作同一个资源的情况。具体的操作流程如下：

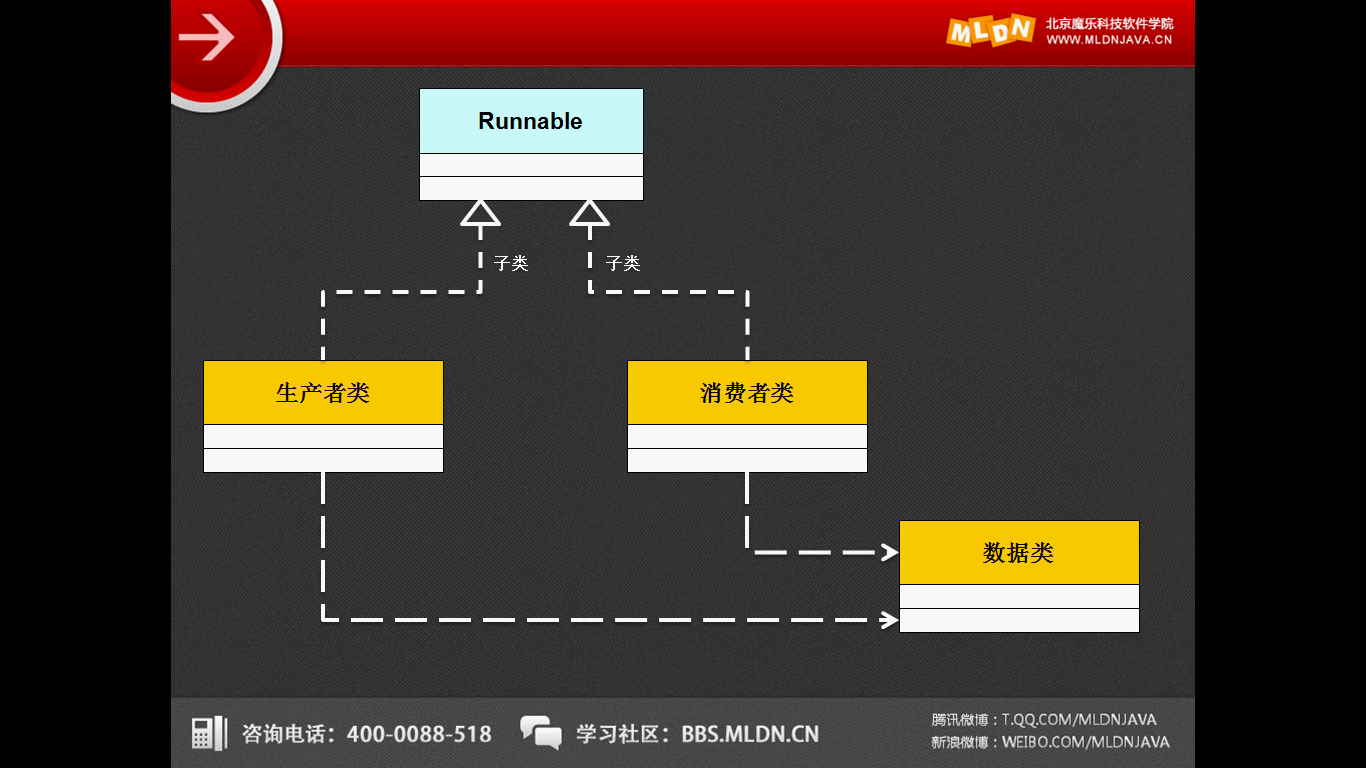
·生产者负责生产数据，消费者负责取走数据；

·生产者没生产完一组数据之后，消费者就要取走一组数据。

那么现在假设要生产的数据如下：

·第一组数据：title=王惊雷，content=好学生一枚；

·第二组数据：title=动物，content=草泥马。



范例：程序基本模型

|  |
| --- |
| **package**cn.mldn.demo;  **class** Info{  **private** String title;  **private** String content;  **public** String getContent() {  **return**content;  }  **publicvoid**setContent(String content) {  **this**.content = content;  }  **public** String getTitle() {  **return**title;  }  **publicvoid**setTitle(String title) {  **this**.title = title;  }  }  **class**Productor**implements** Runnable{  **private** Info info;  **public**Productor(Info info) {  **this**.info = info;  }  @Override  **publicvoid** run() {  **for**(**int**i = 0;i<100 ;i++){  **if**(i%2==0){  info.setTitle("王惊雷");  **try** {  Thread.*sleep*(500);  } **catch** (InterruptedExceptione) {  // **TODO**自动生成的 catch 块  e.printStackTrace();  }  info.setContent("好学生一枚");  }**else**{  info.setTitle("动物");  **try** {  Thread.*sleep*(500);  } **catch** (InterruptedExceptione) {  // **TODO**自动生成的 catch 块  e.printStackTrace();  }  info.setContent("草泥马");  }  }    }  }  **class**Customer**implements** Runnable{  **private** Info info;  **public** Customer(Info info) {  **this**.info = info;  }  @Override  **publicvoid** run() {  **for**(**int**i = 0;i<100;i++){  **try** {  Thread.*sleep*(500);  } **catch** (InterruptedExceptione) {  // **TODO**自动生成的 catch 块  e.printStackTrace();  }  System.***out***.println(**this**.info.getTitle() + " - " + **this**.info.getContent());  }    }  }  **publicclass**TestDemo {  **publicstaticvoid** main(String[] args) {  Info info = **new** Info();  **new** Thread(**new**Productor(info)).start();  **new** Thread(**new**Customer(info)).start();  }  } |

现在实际上通过以上的代码可以返现两个严重问题；

·数据错位，返现不在是一个所需要的完整数据；

·数据重复取出，数据重复生产。

**3.2、解决数据错乱问题**

数据的错位完全是因为非同步的操作所造成的，所以应该使用同步处理。因为取和设置是两个不同的操作，所以要想进行同步控制，那么就需要将其定义在一个类里面完成。

|  |
| --- |
| **package**cn.mldn.demo;  **class** Info{  **private** String title;  **private** String content;  **publicsynchronizedvoid** set(String title,Stringcontent){  **this**.title = title;  **try** {  Thread.*sleep*(500);  } **catch** (InterruptedExceptione) {  // **TODO**自动生成的 catch 块  e.printStackTrace();  }  **this**.content = content;  }  **publicsynchronizedvoid** get(){  **try** {  Thread.*sleep*(500);  } **catch** (InterruptedExceptione) {  // **TODO**自动生成的 catch 块  e.printStackTrace();  }  System.***out***.println(**this**.title + " - " + **this**.content);  }  }  **class**Productor**implements** Runnable{  **private** Info info;  **public**Productor(Info info) {  **this**.info = info;  }  @Override  **publicvoid** run() {  **for**(**int**i = 0;i<100 ;i++){  **if**(i%2==0){  info.set("王惊雷","好学生一枚");  }**else**{  info.set("动物","草泥马");  }  }    }  }  **class** Customer **implements** Runnable{  **private** Info info;  **public** Customer(Info info) {  **this**.info = info;  }  @Override  **publicvoid** run() {  **for**(**int**i = 0;i<100;i++){  **this**.info.get();  }    }  }  **publicclass**TestDemo {  **publicstaticvoid** main(String[] args) {  Info info = **new** Info();  **new** Thread(**new**Productor(info)).start();  **new** Thread(**new** Customer(info)).start();  }  } |

**3.3、解决数据重复的问题**

如果要想实现整个代码的操作，必须加入等待与唤醒机制，在Object类里面提供有专门的处理方法。

·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，那个优先级高先执行哪个：public final void notifyAll()

范例：解决程序问题

|  |
| --- |
| **package** cn.mldn.demo;  **class** Info{  **private** String title;  **private** String content;  **private** **boolean** flag = **true**;  **public** **synchronized** **void** set(String title,String content) {  **if**(flag == **false**) {  **try** {  **super**.wait();  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  }  **this**.title = title;  **try** {  Thread.*sleep*(200);  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  **this**.content = content;  flag = **false**;  **super**.notify();  }  **public** **synchronized** **void** get() {  **if**(flag == **true**) {  **try** {  **super**.wait();  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  }  **try** {  Thread.*sleep*(200);  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  System.***out***.println(**this**.title + " - " + **this**.content );  flag = **true**;  **super**.notify();  }  }  **class** Productor **implements** Runnable{  **private** Info info;  **public** Productor(Info info) {  **this**.info = info;  }  @Override  **public** **void** run() {  // **TODO** Auto-generated method stub  **for**(**int** i = 0;i<100;i++) {  **if**(i%2==0) {  info.set("啦啦啦","学生");  }**else** {  info.set("动物","草泥马");  }  }  }    }  **class** Customer **implements** Runnable{  **private** Info info;  **public** Customer(Info info) {  **this**.info = info;  }  @Override  **public** **void** run() {  // **TODO** Auto-generated method stub  **for**(**int** i = 0;i<100;i++) {  info.get();  }  }    }  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) {  Info info = **new** Info();  **new** Thread(**new** Productor(info)).start();  **new** Thread(**new** Customer(info)).start();  }  } |

面试题：请解释sleep()与wait()的区别？

·sleep()是Thread类定义的方法，wait()是Object类定义的方法；

·sleep()可以设置休眠时间，时间一到自动唤醒，而wait()需要等待notify()唤醒

**StringBuffer类**

回顾 String类的特点：

·String类对象有两种实例化方式：

|-直接复制：只开辟一块堆内存空间，可以自动入池

|-构造方法：开辟两块堆内存空间，不会自动入池，使用intern()手工入池；

·任何一个字符串都是String类的匿名对象；

·字符串一旦声明则不可改变，可以改变的只是String类对象的引用。

虽然在所有的项目里面，String类都一定要是用，可是String类有一个问题不得不进行重复，那么就是String类的内容不可改变。为此在Java里面提供有另外一个类—Stringuffer类（里面的内容可以修改）。

String类的对象可以使用“+”进行字符串的链接操作，但是在StringBuffer类里面必须使用append()方法进行追加。

·方法：public StringBuffer append(数据类型 变量)

范例：观察StringBuffer的基本使用

|  |
| --- |
| **package** cn.mldn.demo;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) {  StringBuffer buf = **new** StringBuffer();  buf.append("hello").append(" world").append(" !");  *change*(buf);  System.***out***.println(buf);  }  **public** **static** **void** change(StringBuffer buf) {  buf.append("\n").append("hello").append(" atos .");  }  } |

发现StringBuffer的内容是可以进行修改的，而String的内容是不可以进行修改的。

|  |  |
| --- | --- |
| String类： | StringBuffer类： |
| public final class String  extends Object  implements Serializable, Comparable<String>, CharSequence | public final class StringBuffer  extends Object  implements Serializable, CharSequence |

发现String与StringBuffer类都是CharSequence的子类。而在以后的开发之中，如果你看见某些方法的操作上出现的是CharSequence接口，那么应该立刻清楚，只要传递字符串就可以。

|  |
| --- |
| **package** cn.mldn.demo;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) {  CharSequence cs = "hello"; //向上转型  System.***out***.println(cs);//String类覆写的toString()  }  } |

虽然String与StringBuffer类有着共同的接口，但是这里两个雷对象之间如果要转换不能够直接转换。

1. 将String变为StringBuffer？

·方式一：利用StringBuffer类的构造方法实现：public StringBuffer(String str)

|  |
| --- |
| **package** cn.mldn.demo;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) {  StringBuffer buf = **new** StringBuffer("hello");  System.***out***.println(buf);  }  } |

·方式二：利用append()方法

|  |
| --- |
| **package** cn.mldn.demo;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) {  StringBuffer buf = **new** StringBuffer();  buf.append("hello");  System.***out***.println(buf);  }  } |

1. 将StringBuffer类变为String

·利用toString方法强StringBuffer变为String

|  |
| --- |
| **package** cn.mldn.demo;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) {  StringBuffer buf = **new** StringBuffer("hello");  String str = buf.toString();  System.***out***.println(str);  }  } |

也可以利用String的构造方法（public String(StringBuffer buffer)）完成SringBuffer到String的转换

在String类里面也存在一个与StringBuffer比较的方法：public boolean contentEquals(StringBuffer sb)

|  |
| --- |
| **package** cn.mldn.demo;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) {  StringBuffer buf = **new** StringBuffer("hello");  System.***out***.println("hello".contentEquals(buf));  }  } |

StringBuffer方法

1、字符串反转public StringBuffer reverse()

|  |
| --- |
| **package** cn.mldn.demo;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) {  StringBuffer buf = **new** StringBuffer("hello");  System.***out***.println(buf.reverse());  }  } |

2、在指定的索引位置增加数据：public StringBuffer insert(int offset,数据类型 变量)

|  |
| --- |
| **package** cn.mldn.demo;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) {  StringBuffer buf = **new** StringBuffer("hello");  buf.insert(0, "hi ").insert(0, "NIhao ");  System.***out***.println(buf);  }  } |

3、删除部分数据：public StringBuffer delete(int start,int end)

|  |
| --- |
| **package** cn.mldn.demo;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) {  StringBuffer buf = **new** StringBuffer("hello world yayayyaa");  buf.delete(5, 11);  System.***out***.println(buf);  }  } |

从JDK1.5之后增加了一个新的字符串操作类：StringBuilder类，这个类定义结构如下：

|  |
| --- |
| public final class StringBuilder  extends Object implements Serializable, CharSequence |

发现StringBuffer类与StringBuilder类在定义上非常的相似，几乎连方法也一样。

面试题：请解释String、StringBuffer、StringBuilder的区别？

·String的内容一旦声明则不可改变，而StringBuffer与StringBuilder声明的内容可以改变。

·StringBuffer类中提供的方法都是同步方法，属于安全的线程操作，而StringBuilder类中提供的方法都是一部方法，属于非线程安全的操作。

日后再开发之中，如果遇见字符串的应用，不需要思考95%使用的都是String类，只有在需要频繁修改的时候次啊会考虑 到使用StringBuffer或StringBuilder的操作。

**Runtime类**

在每一个JVM进程里面都会存在一个Runtime类的对象，这个类的主要功能是取得一些与运行时有关的环境属性或者创建新的进程等操作。

在Runtime类定义的时候他的构造方法已经被私有化了，这就属于单例设计的应用，因为要保证进程里面只有唯一的一个Runtime类对象。所以在Runtime类中提供一个static型的方法，这个方法可以取得Runtime类的实例化对象：public static Runtime getRuntime()。

Runtime类是直接与本地运行有关的所有相关属性的集合，所以在Runtime类，所以在Runtime类中定义有如下的方法：

·返回所有可用的内存空间：public long totalMemory()；

·返回最大内存空间：public long maxMemory()

·返回空余内存空间：public long freeMemory()

范例：观察内存大小

|  |
| --- |
| **package** cn.mldn.demo;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) {  Runtime run = Runtime.*getRuntime*();  System.***out***.println("1, MAX : " + run.maxMemory());  System.***out***.println("1, total : " + run.totalMemory());  System.***out***.println("1, free : " + run.freeMemory());  String str = "";  **for**(**int** i = 0 ; i<2000;i++) {  str += i;  }  System.***out***.println("2, MAX : " + run.maxMemory());  System.***out***.println("2, total : " + run.totalMemory());  System.***out***.println("2, free : " + run.freeMemory());  }  } |

如果一旦产生了过多的垃圾之后，那么久会改变可用的内存空间的大小。

可是在Runtime类里面有一个方法，可以释放掉垃圾空间：public void gc()

|  |
| --- |
| **package** cn.mldn.demo;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) {  Runtime run = Runtime.*getRuntime*();  System.***out***.println("1, MAX : " + run.maxMemory());  System.***out***.println("1, total : " + run.totalMemory());  System.***out***.println("1, free : " + run.freeMemory());  String str = "";  **for**(**int** i = 0 ; i<2000;i++) {  str += i;  }  System.***out***.println("2, MAX : " + run.maxMemory());  System.***out***.println("2, total : " + run.totalMemory());  System.***out***.println("2, free : " + run.freeMemory());  run.gc();  System.***out***.println("3, MAX : " + run.maxMemory());  System.***out***.println("3, total : " + run.totalMemory());  System.***out***.println("3, free : " + run.freeMemory());  }  } |

面试题：请解释什么叫GC？如何处理？

·GC(Garbage Collector) 垃圾收集器，指的是释放无用的内存空间；

·GC会由系统不定期进行自动的回收，或者调用Runtime类中的gc()方法手工回收。

实际上Runtime类还有一个更加有意思的功能，就是说它可以调用本机的可执行程序，并且创建进程。

·执行程序：public Process exec(String command) throws IOException

范例：执行进程

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.io.IOException;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** IOException, InterruptedException {  Runtime run = Runtime.*getRuntime*();  Process pro = run.exec("mspaint.exe");//调用本机程序  Thread.*sleep*(2000);  pro.destroy();//销毁进程  }  } |

**System类**

在System类里面之前使用过一个System.arraycopy()方法实现数组拷贝，而这个方法的真是定义如下：

·数组拷贝：public static void arraycopy(Object src,int srcPos,Object dest,int destPos,int length)

在System类里面定义有一个重要的方法：

·取得当前的系统时间：public static long currentTimeMillis()

范例：请统计出某项操作的执行时间

|  |
| --- |
| **package** cn.mldn.demo;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  **long** start = System.*currentTimeMillis*();  String str = "";  **for**(**int** i = 0 ; i< 3000; i++) {  str += i;  }  **long** end = System.*currentTimeMillis*();  System.***out***.println(end-start);  }  } |

如果要想统计出所花费的毫秒时间，就用long型数据直接进行数学计算后得来。

在System类里面定义了一个操作方法：public static void gc()，这个gc()方法并不是新定义的方法，而是间接调用了Runtime类中的gc()方法。

对象的产生一定会调用构造方法，可以进行一些处理操作，但是某一个对象如果要被回收了，那么连一个收尾的机会都没有。如果需要给兑现一个收尾的机会，那么久可以考虑覆写Object类中的finalize()方法。

·finalize()方法：protected void finalize() throws Throwable

|-在对象回收是就算抛出了任何异常，也不会影响到整个程序的正常运行。

|  |
| --- |
| **package** cn.mldn.demo;  **class** Member{  **public** Member() {  System.***out***.println("aaaaaa");  }  @Override  **protected** **void** finalize() **throws** Throwable {  System.***out***.println("bbbbb");  **throw** **new** Exception("cccccccc");  }  }  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Member m = **new** Member();  m = **null**;  System.*gc*();    }  } |

构造方法是留给对象初始化使用的，finalize()方法是留给对象回收钱使用的。

面试题：请解释final、finally、finalize的区别？

·final：关键字，定义不能被继承的类，不能被覆写的方法，常量；

·finally：关键字，异常的统一出口；

·finalize：方法，object类提供的方法（protected void finalize() throws Throwable），对象回收钱的收尾方法，及时出现了异常也不会导致程序中断；

**对象克隆**

对象克隆指的就是对象的复制操作，在Object类里面提供有一个专门的克隆方法。

·对象克隆：protected Object clone() throws CloneNotSupportedException

此方法上抛出一个“CloneNotSupportedException”异常，如果要使用对象克隆的类没有实现Cloneable接口，那么就会抛出此异常。但是Cloneable接口看不见方法，此为表示接口，表示一种操作能力。

|  |
| --- |
| **package** cn.mldn.demo;  **class** Book **implements** Cloneable{  **private** String title;  **private** **double** price;  **public** Book(String title,**double** price) {  **this**.title = title;  **this**.price = price;  }  **public** **void** setTitle(String title) {  **this**.title = title;  }  @Override  **public** String toString() {  // **TODO** Auto-generated method stub  **return** "title = " + **this**.title + ", price : " + **this**.price;  }  @Override  **protected** Object clone() **throws** CloneNotSupportedException {  // **TODO** Auto-generated method stub  **return** **super**.clone();  }  }  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Book boka = **new** Book("java",173);  Book bokb = (Book) boka.clone();  boka.setTitle("jsp");  System.***out***.println(boka);  System.***out***.println(bokb);  }  } |

**数学操作类（Math类）**

在Math类里面提供的一切方法都是static型的方法，因为Math类里面没有普通属性。

·四舍五入：public static long round(double a)

如果进行附属四舍五入的时候，操作的数据小数位大于0.5才进位，小于等于0.5不进位。整个小数位都进位

**数学操作类（Random类）**

这个类的主要功能是取得随机数的操作类。

范例：产生10个不大于100 的正整数（0~99）

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.util.Random;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Random rand = **new** Random();  **for**(**int** i = 0 ; i <10;i++){  System.***out***.print(rand.nextInt(100) + ",");  }  }  } |

36选7

|  |
| --- |
| **package** cn.mldn.demo;  **import** java.util.Random;  **public** **class** TestDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  Random rand = **new** Random();  **int** data[] = **new** **int**[7];  **int** foot = 0;  **while**(foot < 7) {  **int** t = rand.nextInt(37);  **if**(!*isRepeat*(data,t)) {  data[foot++] = t;  }  }  **for**(**int** i =0 ; i<data.length;i++) {  System.***out***.println(data[i]);  }  }    **public** **static** **boolean** isRepeat(**int** temp[],**int** num) {  **if**(num == 0) {  **return** **true**;  }  **for**(**int** i = 0;i<temp.length;i++) {  **if**(temp[i] == num) {  **return** **true**;  }  }  **return** **false**;  }  } |

**大整数操作类:BigInteger**