

## 章节31-章节40

### 章节31 课时 124 03105\_Eclipse基本使用（简介）

Eclipse简介：Eclipse如果翻译为中文指的是"日食"，指的是遮盖一切的光芒，这是一个针对于SUN很挑衅的一个名字，它是由IBM开发的。后来转送给了今天的Eclipse组织，进行开源项目的推广。

Java原本是sun公司开发的，但是结果使用最流行的开发工具不是SUN提供的。最早SUN提供了Java之后，他们认为Java可以完全的取代行业的主流变成语言的地位，所以曾经高调的宣布，我们不搞开发工具，交给其他公司去搞。

最早SUN的开发工具最得力的就是Borland公司的JBuilder（功能强大外表不好），可是在最早的时候都用它。可是后来在2004年的时候，SUN开始推出自己的开发工具——NetBeans，在2003年的时候有一个小小的工具开始默默的成长，就是今天的Eclipse。再后来JBuilder持续发展，到了2006年底的时候Borland公司倒闭了。因为此时全球开始席卷"开源"风潮，随后Borland公司的技术部单独成立出去继续进行JBuilder的后续版本开发（就是Eclipse插件）。此时的NetBeans继续很少人问津。而现在的开发几乎都以Eclipse为主了（伴随Eclipse一起成长的还有IDEA工具，Eclipse更容易上手，后来IDEA的逐步改善，从2014年开始，IDEA的使用者越来越多）。

<https://www.eclipse.org/home/>

在2004年之前在整个行业内有两套开发的架构：

解释顺序：操作系统+数据库+中间件+开发工具；

皇家级：AIX+DB2+WAS+WSAD；

杂牌级：UNIX/Linux+Oracle+BEAWebLogic+JBuilder；

从2006年开始出现了一下的一套架构：

免费级：Linux+MySQL+Tomcat+Eclipse

eclipse是一个免费的绿色版本软件，不需要安装。。。（貌似已改）

从历史上Eclipse来看，它几乎都会包含有一下的几个部分：JDT、JUNIT、CVS客户端、插件开发、GIT客户端。

Eclipse之中所有的项目都是以工作区为主的，一个工作区中可以包含有多个项目。Eclipse的所有配置都是以工作区为主的，也就是说每一个工作区都有自己独立的配置。如果发现某一个工作区坏了，只需要更换一个工作区即可恢复到原始状态。

### 章节31 课时 125 03106\_Eclipse基本使用（JDT使用）

首先建立一个java项目

项目建立完成后首先在项目的文件目录下生成两个目录：

src:保存所有的\*.java源文件

bin:保存所有生成的\*.class文件

如果想要调整所有的**字体选项**，要使用Window→首选项(Preferences)→General→Appearance→Colors and Fonts→Basic→Text Font

每当程序编写完成之后，一保存就会自动的进行编译。

在Eclipse里面最大的特点在于代码生成的功能，下面例如编写了如下代码：

```
class Book{
    private String title;
    private double price;
}
```

如果此时要编写简单Java类，则需要生成构造、setter、getter等方法，但是用Eclipse只需要点击Source选择Generate Getters and Setters就会自动生成。

要想输入初始化参数，那么必须进入到运行时的配置项。

之所以会使用Eclipse还有一个最为重要的功能，它可以利用一些快捷键方便的进行一些项目的开发。

**Alt+/** 进行代码的提示

cout或者sysout然后按Alt+/直接就输入了System.out.println();

主方法则输入main然后按

**ctrl+D** 删除当前行代码

**ctrl+shift+o** 组织导入，导入其他类的包

**ctrl+/** 使用单行注释，选定区域，按下。选区每行都变为单行注释，再按一次取消。

**ctrl+h** 强力搜索

**ctrl+alt+↓** 复制当前行代码

**ctrl+shift+L** 全部快捷键列表

**/\*\***再按回车可以出现注释

**ctrl+1** 为错误的代码给出纠正方案

```
package cn.mldn.demo;
```

```
class MyMath{
```

```
    public static int div(int x,int y)throws Exception{
        return x/y;
    }
```

```
}
```

```
}
```

```
public class Hello {
```

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

```
        MyMath.div(10,2);//调用出错，在此处按Ctrl+1，选择Surround with try/catch就会变成如下代码
```

```
    }
```

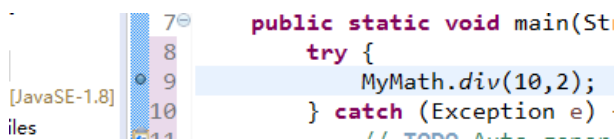
```
}
```

```

package cn.mldn.demo;
class MyMath{
    public static int div(int x,int y)throws Exception{
        return x/y;
    }
}
public class Hello {

    public static void main(String[] args) {
        try {
            MyMath.div(10,2);
        } catch (Exception e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}

```



在Eclipse里面还提供有debug(代码的跟踪调试)功能。设置好断点之后采用调试的方式运行程序。

单步进入 (F5) : 指的是进入到执行的方法之中观察方法的执行效果。

单步跳过 (F6) : 在当前代码的表面上执行;

单步返回 (F7) : 不再观察了, 而返回到进入处

单步执行 (F8) : 停止调试, 而直接正常执行完毕。

在调试之中可以清楚的知道方法中所有变量的数值的变化情况。

如果说现在某一个项目不再使用了, 那么可以进行删除, 而删除也分为两种形式;

从工作区里删除, 但是磁盘保留; 日后可以对项目进行重新导入;

从磁盘上彻底删除项目, 彻底消失。

Import 导入

Export 导出, 有些时候Eclipse导出的jar文件可能无法使用。最稳妥的办法还是用jar命令打包。

### 章节31 课时 126 03107\_Eclipse基本使用 (junit测试工具) 重点

junit是一个测试工具, 对于软件测试而言分为两种测试:

黑盒测试: 针对功能进行测试, 看不见里面的代码;

白盒测试: 针对性能进行测试, 算法的调整。

对于白盒测试一定是比黑盒测试更加的有技术要求。可是除了这2中测试之外, 还有一类测试——用例测试 (UseCase测试工程师)。这个职位要求在某一个行业的从业经验丰富, 要充分理解某一个行业的业务流程。

junit就是一个use case测试工具, 但是junit本身使用并不麻烦。

范例: 定义一个程序

```

package cn.mldn.util;
public class MyMath {
    public static int div(int x,int y)throws Exception{
        int temp=0;
        temp=x/y;
        return temp;
    }
}

```

首先选择好这个类, 而后选择建立junit测试工具类。

新建java→JUnit下面有两个:

JUnit Test Case 一个测试用例






JUnit Test Suite一组测试用例

选择针对div方法进行测试

## Test Methods

Select methods for which test method stubs

Available methods:

- ☒ ☒  MyMath
  - ☒  div(int, int)
- ☐ ☐  Object
  - ☐  Object()
  - ☐  getClass()

由于junit属于第三方一个开放包，所以使用前需要导入此包。

```
package cn.mldn.test;
import static org.junit.Assert.*;
import org.junit.Test;
import cn.mldn.util.MyMath;
import junit.framework.TestCase;
public class MyMathTest {
    @Test
    public void testDiv() {
        try {
            TestCase.assertNotSame(MyMath.div(10,2),1);
        } catch (Exception e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

对于junit的测试结果有两种：

GREEN BAR:测试通过；

RED : BAR: 测试失败；

Eclipse工具随着时间一定都可以掌握

对于junit先熟悉它的操作。

## 章节32 课时 127 03108\_可变参数

```
package cn.mldn.demo;
public class TestDemo {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        System.out.println(add(new int[] {1,2,3})); //传递三个整型数据
        System.out.println(add(new int[] {10,20})); //传递两个整型数据
    }
    /**
     * 实现任意多个整型数据的相加操作
     * @param data 由于要接收多个整型数据，所以使用数组完成接收。
     * @return 多个整型数据的累加结果
     */
    public static int add(int [] data){
        int sum=0;
        for(int x=0;x<data.length;x++){
            sum+=data[x];
        }
        return sum;
    }
}
```

以上的代码之所以使用数组，是因为多个参数方法上无法描述，所以利用数组整合多个参数，但是严格来讲这样的实现并不标准。要求是可以接收任意多个整型数据：

理想的调用形式：add(1,2,3)、add(10,20)、add(100,10,102,3,20,40,30...);

这一功能从JDK1.5之后正式的登陆到了Java之中，它主要是在方法上使用，定义其形式：

```
[public] [protected] [private] [static] [final] [abstract] 返回值类型 方法名称(参数类型...变量){
    [return] [返回值];
}
```

此时给出的参数不再是一个内容，而是多个内容。但是尽管参数的定义形式变了，可是参数的访问却没有改变，也就是说在进行参数访问的时候按照数组的形式进行操作。

```

package cn.mldn.demo;
public class TestDemo {
    public static void main(String[] args) {
        // 可变参数支持接收数组
        System.out.println(add(new int[] {1,2,3})); // 传递三个整型数据
        System.out.println(add(new int[] {10,20})); // 传递两个整型数据
        // 或者使用", " 区分不同的参数, 接收的时候还是数组
        System.out.println(add(1,2,3)); // 传递3个参数
        System.out.println(add(10,20)); // 传递2个参数
        System.out.println(add()); // 不传递参数
    }
    /**
     * 实现任意多个整型数据的相加操作
     * @param data 由于要接收多个整型数据, 所以使用数组完成接收。
     * @return 多个整型数据的累加结果
     */
    public static int add(int...data){
        int sum=0;
        for(int x=0;x<data.length;x++){
            sum+=data[x];
        }
        return sum;
    }
}

```

在大部分的咖啡情况下, 应该要求参数的个数是准确的, 所以对于这样的开发往往不会应用于应用型的开发, 可能用于一些程序相关系统类的设计使用上。

- 1、在设计一个类的视乎可变参数绝对不是优先的选择
- 2、可变参数属于数组的变形应用。

## 章节32 课时 128 03109\_foreach循环

foreach输出是有C#最早引入的概念, 其目的就是进行数组或者是集合数据的输出。在最早如果要进行数组输出肯定使用for循环, 而后利用下标进行输出。

```

package cn.mldn.demo;
public class TestDemo {
    public static void main(String[] args) {
        int data[]=new int[]{1,2,3,4,5};
        for(int x=0;x<data.length;x++){
            System.out.println(data[x]);
        }
    }
}

```

有人会觉得以上的输出需要使用索引比较麻烦。

从JDK1.5之后增加的foreach循环开式就可以取消掉索引的操作开式。语法如下:

```

for(类型 变量:数组|集合){
    //每一次循环会自动将数组的内容设置给变量
}

```

范例: 观察增强型for循环

```

package cn.mldn.demo;
public class TestDemo {
    public static void main(String[] args) {
        int data[]=new int[]{1,2,3,4,5};
        for(int x:data){ // 循环次数由长度决定
            // 每一次循环实际上都表示数组的脚标增长, 会取得每一个数组的内容, 并且将其设置给了x
            System.out.println(x); // x就是每一个数组元素的内容
        }
    }
}

```

用数组直接通过索引访问会比较麻烦, 而有了这样的开式代码就避免了索引的麻烦。foreach循环支持数组的直接访问, 避免了索引访问带来的麻烦。

## 章节32 课时 129 03110\_静态导入

// 如果一个类中定义的方法全部都属于static型的方法, 那么其他类要引用此类时必须使用"类名称.方法"进行调用。

// 范例: 传统的做法

```

package cn.mldn.util;
public class MyMath {
    public static int add(int x,int y){
        return x+y;
    }
    public static int div(int x,int y){
        return x/y;
    }
}
//此时MyMath类里面的方法都是static型的方法，随后在其他类使用这些方法。
//范例：基本使用形式
package cn.mldn.demo;
import cn.mldn.util.MyMath;
public class TestDemo {
    public static void main(String[] args) {
        System.out.println("加法操作"+MyMath.add(10,20));
        System.out.println("除法操作"+MyMath.div(10,2));
    }
}

```

如果再主类中定义的是static方法，那么可以直接调用static方法，而现在的MyMath类里面都是static方法，那么我们觉得前面加上类名称实在多余。于是从JDK1.5Z之后增加了静态导入。

范例：静态导入

```

package cn.mldn.demo;
//将MyMath类中的全部static方法导入，这些方法就好比在主类中定义的static方法一样。
import static cn.mldn.util.MyMath.*;
public class TestDemo {
    public static void main(String[] args) {
        //直接使用方法名称访问
        System.out.println("加法操作"+add(10,20));
        System.out.println("除法操作"+div(10,2));
    }
}

```

从道理上来讲，如果再前面加上了类名称，认为反而能够更加清楚的表示出具体方法属于哪个类。

### 章节33 课时 130 03111 泛型（问题引出）

1、泛型技术的产生背景

2、泛型操作的实现

3、通配符的使用

泛型的引出：

现在要求定义一个表示坐标的操作类（point），在这个类里面要求保存有以下几种坐标：

保存数字：x=10;y=20;

保存小数：x=10.2;y=20.3;

保存字符串：x=东经20度;y=北纬15度。

现在这个Point类设计的关键就在于x与y这两个变量的类型设计上。首先想到的一定是Object类型：

int：int自动装箱为Integer，Integer向上转型为Object；

double：double自动装箱为Double，Double向上转型为Object；

String：直接向上转型为Object

范例：初期设计如下：

```

package cn.mldn.demo;
class Point{
    private Object x;
    private Object y;
    public void setX(Object x) {
        this.x = x;
    }
    public void setY(Object y) {
        this.y = y;
    }
    public Object getX() {
        return x;
    }
    public Object getY() {
        return y;
    }
}

```

//下面重复的演示三个程序，分别使用各个不同的数据类型。

//范例：在Point类里面保存整型数据

```
public class TestDemo {
    public static void main(String[] args) {
        //第一步设置数据
        Point p=new Point();
        p.setX(10);
        p.setY(20);
        //第二部取出数据
        int x=(Integer) p.getX();
        int y=(Integer) p.getY();
        System.out.println("X坐标"+x+" , Y坐标"+y);
    }
}
```

范例：使用小数

```
public class TestDemo {
    public static void main(String[] args) {
        //第一步设置数据
        Point p=new Point();
        p.setX(10.1);
        p.setY(20.2);
        //第二部取出数据
        double x=(Double) p.getX();
        double y=(Double) p.getY();
        System.out.println("X坐标"+x+" , Y坐标"+y);
    }
}
```

范例：使用字符串

```
public class TestDemo {
    public static void main(String[] args) {
        //第一步设置数据
        Point p=new Point();
        p.setX("东经18°");
        p.setY("北纬100°");
        //第二部取出数据
        String x=(String) p.getX();
        String y=(String) p.getY();
        System.out.println("X坐标"+x+" , Y坐标"+y);
    }
}
```

此时的代码已经利用了Object数据类型解决了一切的开发问题，可是解决的关键靠的是Object，于是失败的关键就是Object。

范例：错误的程序

```
public class TestDemo {
    public static void main(String[] args) {
        //第一步设置数据
        Point p=new Point();
        p.setX("东经18°");
        p.setY(10);
        //第二部取出数据
        String x=(String) p.getX();
        String y=(String) p.getY();
        System.out.println("X坐标"+x+" , Y坐标"+y);
    }
}
```

这样的代码还不需要执行呢，就已经可以观察到本程序可能存在的问题，因为在设置的时候存放的是int（Integer），而取出的时候使用的是String，两个没有任何关系的类对象之间要发生强制转换，就一定会产生ClassCastException错误。

向上转型的核心目的在于统一操作的参数上，而向下转型的目的是操作子类定义的特殊功能，可是现在的问题发现，向下转型是一件非常不安全的操作，那么这一操作应该在代码运行之前就已经能够自动的排查出来这是最好的选择。可是之前的技术做不到。

从JDK1.5之后开始增加了泛型技术，而泛型技术的核心意义在于：类在定义的时候，可以使用一个标记，此标记就表示类中属性或方法参数的类型标记，在使用的时候才动态的设置类型。

package cn.mldn.demo;

//此时设置的T在Point类定义上只表示一个标记，在使用的时候需要为其设置具体的类型。

```
class Point<T>{//定义坐标，type=T,是一个类型
    private T x;//此属性的类型不知道，由Point使用时动态决定
    private T y;//此属性的类型不知道，由Point使用时动态决定
    public void setX(T x) {
        this.x = x;
    }
}
```

```

    public void setY(T y) {
        this.y = y;
    }
    public T getX() {
        return x;
    }
    public T getY() {
        return y;
    }
}
public class TestDemo {
    public static void main(String[] args) {
        //第一步设置数据
        Point<String> p=new Point<String>();
        //利用的就是包装类的自动装箱功能
        p.setX("东经120°");
        //如果设置的数据类型是错误的，那么在编译的时候就会自动的排查
        p.setY("北纬91°");
        //第二步取出数据，由于接收的类型就是String，所以不需要向下强制转换
        String x=p.getX();
        String y=p.getY();
        System.out.println("X坐标："+x+"，Y坐标："+y);
    }
}

```

发现使用了泛型之后，所有类中属性的类型都是动态设置的，而所有使用泛型标记的方法参数类型也都发生了改变，这样就相当于避免了向下转型的问题，从而解决了类转换的安全隐患。

但是需要特别说明的是，如果要想使用泛型，那么它能够采用的类型只能是类，即：不能是基本类型，只能是引用类型

对于泛型有两点说明：

1、如果再使用泛型类或者是接口的时候，没有设置泛型的具体类型，那么会出现编译时的警告，同时为了保证程序不出错，所有泛型都将使用Object表示。

```

public class TestDemo {
    public static void main(String[] args) {
        //第一步设置数据
        Point <Integer> p=new Point <Integer>();
        //利用的就是包装类的自动装箱功能
        p.setX(10);
        //如果设置的数据类型是错误的，那么在编译的时候就会自动的排查
        p.setY(20);
        //第二步取出数据，由于接收的类型就是String，所以不需要向下强制转换
        int x=p.getX();
        int y=p.getY();
        System.out.println("X坐标："+x+"，Y坐标："+y);
    }
}

```

---

```

public class TestDemo {
    public static void main(String[] args) {
        //第一步设置数据
        Point p=new Point();//如果两边都没有<Integer>，将使用Object类型来描述泛型
        //利用的就是包装类的自动装箱功能
        p.setX(10);
        //如果设置的数据类型是错误的，那么在编译的时候就会自动的排查
        p.setY(20);
        //第二步取出数据，由于接收的类型就是String，所以不需要向下强制转换
        int x=(Integer)p.getX();//相当于Object类向下转型为int
        int y=(Integer)p.getY();
        System.out.println("X坐标："+x+"，Y坐标："+y);
    }
}

```

2、从JDK1.7开始可以简化声明泛型

```

public class TestDemo {
    public static void main(String[] args) {
        //第一步设置数据
        Point <Integer> p=new Point <>();//JDK1.7之后，实例化的泛型可以省略，即等号右边的Integer可以省略
        //利用的就是包装类的自动装箱功能
        p.setX(10);
        //如果设置的数据类型是错误的，那么在编译的时候就会自动的排查
        p.setY(20);
    }
}

```

```

//第二步取出数据，由于接收的类型就是String，所以不需要向下强制转换
int x=p.getX();
int y=p.getY();
System.out.println("X坐标："+x+"，Y坐标："+y);
}
}

```

### 章节33 课时 131 03112\_泛型（通配符）

为了更好的理解通配符的作用，下面首先来观察一个程序。

```

package cn.mldn.demo;
class Message <T>{
    private T msg;
    public void setMsg(T msg) {
        this.msg = msg;
    }
    public T getMsg() {
        return msg;
    }
}
public class TestDemo {
    public static void main(String[] args) {
        Message <String> m=new Message <String>();
        m.setMsg("Hello world!");
        fun(m);//引用传递
    }
    public static void fun(Message<String> temp){
        System.out.println(temp.getMsg());
    }
}

```

以上的代码为Message类设置的是一个String型的泛型类型，但是如果说现在设置的是其他类型呢。如果改为Integer，fun()方法里面接收的“Message<String>”那么就不能够使用了，并且fun()方法不能够针对不同的泛型进行重载，因为方法的重载认的只是参数的类型，与泛型无关。

解决方法一：不设置方法参数的泛型

```

public class TestDemo {
    public static void main(String[] args) {
        Message <Integer> m1=new Message <Integer>();
        Message <String> m2=new Message <String>();
        m1.setMsg(100);
        m2.setMsg("Hello world!");
        fun(m1);//引用传递
        fun(m2);//引用传递
    }
    public static void fun(Message temp){
        System.out.println(temp.getMsg());
    }
}

```

发现可以输出100 和Hello World！但是真的解决问题了吗

此时在fun()方法上依然存在有警告信息，因为只要不设置泛型，就一定会有警告信息。可能存在的问题：本来m1为Integer型，因为fun()方法里面未设置泛型，默认为Object型，所以设置了String型也不会报错

```

public class TestDemo {
    public static void main(String[] args) {
        Message <Integer> m1=new Message <Integer>();
        m1.setMsg(100);
        fun(m1);//引用传递
    }
    public static void fun(Message temp){
        temp.setMsg("Hello");//设置为String型
        System.out.println(temp.getMsg());
    }
}

```

所以现在最需要解决的是，需要有一种方式可以接收一个类的任意的泛型类型，但是不可以修改只能取出，那么就可以使用“?”来进行描述。

```

public class TestDemo {
    public static void main(String[] args) {
        Message <Integer> m1=new Message <Integer>();
        m1.setMsg(100);
        fun(m1);//引用传递
    }
}

```



```

    }
    public static void fun(Message<?> temp){//不能够设置，但是可以取出。
        System.out.println(temp.getMsg());
    }
}

```

在"?"通配符基础上还会有两个子的通配符：

**?extends**类：设置泛型上限，可以在声明上和方法参数上使用；

?extends Number.意味着可以设置Number或者是Number的子类(Integer、Double...)

**?super**类：设置泛型下限，方法参数上使用；

?super String :意味着只能设置String或者是它的父类Object;

范例：设置泛型的上限

```

package cn.mldn.demo;
class Message <T extends Number>{
    private T msg;
    public void setMsg(T msg) {
        this.msg = msg;
    }
    public T getMsg() {
        return msg;
    }
}
public class TestDemo {
    public static void main(String[] args) {
        Message <Integer> m1=new Message <Integer>();
        m1.setMsg(100);
        fun(m1);//引用传递
    }
    public static void fun(Message<? extends Number> temp){//不能够设置，但是可以取出。
        System.out.println(temp.getMsg());
    }
}

```

如果设置了非Number或者是其子类的话，那么将出现语法错误。

范例：设置泛型的下限

```

public class TestDemo {
    public static void main(String[] args) {
        Message <String> m1=new Message <String>();//如果还是Integer就会报错
        m1.setMsg("Hello");
        fun(m1);//引用传递
    }
    public static void fun(Message<? super String> temp){//不能够设置，但是可以取出。
        System.out.println(temp.getMsg());
    }
}

```

对于这些概念要求能够看懂就行了，至于说具体的代码你们可以编写的情况并不多。

### 章节33 课时 132 03113\_泛型（泛型接口）

之前都是讲泛型定义在了一个类里面，那么泛型也可以在接口上声明，成为泛型接口。

范例：定义泛型接口

形式一：在子类设置泛型

```

package cn.mldn.demo;
//如果是接口在前面加上字母I，例如IMessage
//如果是抽象类在前面加上Abstract，例如AbstractMessage
//如果是普通类直接编写，例如Message
interface IMessage <T>{//设置泛型接口
    public void print(T t);
}
//子类也继续使用泛型，并且父接口使用和子类同样的泛型标记。
class MessageImpl<T> implements IMessage<T>{
    public void print(T t) {
        System.out.println(t);
    }
}
public class TestDemo {
    public static void main(String[] args) {
        IMessage <String> msg=new MessageImpl <String>();
    }
}

```

```

        msg.print("Hello World!");
    }
}

```

形式二：在子类不设置泛型，而为父接口明确的定义一个泛型类型。

```

package cn.mldn.demo;
//如果是接口在前面加上字母I，例如IMessage
//如果是抽象类在前面加上Abstract，例如AbstractMessage
//如果是普通类直接编写，例如Message
interface IMessage <T>{//设置泛型接口
    public void print(T t);
}
class MessageImpl implements IMessage<String>{
    public void print(String t) {
        System.out.println(t);
    }
}
public class TestDemo {
    public static void main(String[] args) {
        IMessage <String> msg=new MessageImpl ();
        msg.print("Hello World!");
    }
}

```

这两类代码要求一定可以弄明白。

### 章节33 课时 133 03114\_泛型（泛型方法）

泛型方法不一定非要定义在支持泛型的类里面。在之前所编写的所有存在有泛型的方法都是在泛型支持类里面定义的。

范例：泛型方法定义

```

package cn.mldn.demo;
public class TestDemo {
    public static void main(String[] args) {
        String str=fun("Hello");
        System.out.println(str.length());
    }
    //T的类型由传入的参数类型决定
    public static <T> T fun(T t){
        return t;
    }
}

```

能够看懂泛型方法的标记就行了。

总结：

- 1、泛型解决的是向下转型所带来的安全隐患，其核心的组成就是在声明类或接口的时候不设置参数的属性或类型。
- 2、“?”可以接收任意的泛型类型，只能够取出，但是不能够修改。

### 章节34 课时 134 03115\_枚举（枚举简介）

枚举能用就用，不能用就别用，我不用。。。。

1、枚举的基本用法

2、枚举的定义组成

多例设计模式：构造方法私有化，而后再类的内部提供有若干个实例化对象，并且通过static方法返回。

范例：定义一个表示颜色基色的多例

```

package cn.mldn.demo;
class Color{
    public static final Color RED=new Color("红色");
    public static final Color GREEN=new Color("绿色");
    public static final Color BLUE=new Color("蓝色");
    private String title;
    private Color(String title) {
        this.title = title;
    }
    public static Color getInstance(int ch){
        switch(ch){
            case 1:
                return RED;
            case 2:
                return GREEN;

```

```

        case 3:
            return BLUE;
        default:
            return null;
    }
}
public String toString(){
    return this.title;
}
}
public class TestDemo {
    public static void main(String[] args) {
        Color red=Color.getInstance(1);
        System.out.println(red);
    }
}

```

枚举严格来讲，各个语言都支持（除了2005年之前的Java），也就是说在2005年之前，要是想定义枚举都会采用如上的开式代码完成。从2005年之后，增加了枚举的概念，同时设置了一个新的enum的关键字。

现在定义枚举：

```

package cn.mldn.demo;
enum Color{//定义好了枚举
    RED, GREEN, BLUE;//大写表示此处为实例化对象
}
public class TestDemo {
    public static void main(String[] args) {
        Color red=Color.RED;
        System.out.println(red);
    }
}

```

使用枚举之后可以完全简化的替代多例设计模式。

但是需要说明的是，严格来讲枚举并不是一个新的功能，在Java里面虽然使用了enum关键字定义了枚举，但是使用enum定义的枚举就相当于是一个类继承了Enum类而已。

```
public abstract class Enum<E extends Enum<E>>
```

```
extends Object
```

```
implements Comparable<E>, Serializable
```

Enum是一个抽象类，里面定义的构造如下：

```
protected Enum(String name, int ordinal)
```

Enum类的构造方法依然是被封装的，所以也属于构造方法私有化的应用范畴，所有的多例设计模式前提是构造方法私有化。

在Enum类定义了两个方法：

取得枚举的索引：public final int ordinal()

取得枚举的名字：public final String name()

除了以上支持的方法之外，使用enum关键字定义的枚举类里面还有一个values()方法，这个方法将枚举对象以对象数组的开式全部返回。

```

package cn.mldn.demo;
enum Color{//定义好了枚举
    RED, GREEN, BLUE;//大写表示此处为实例化对象
}
public class TestDemo {
    public static void main(String[] args) {
        for(Color c:Color.values()){//foreach循环
            System.out.println(c.ordinal()+"-"+c.name());
        }
    }
}

```

执行结果为：

0-RED

1-GREEN

2-BLUE

表示的确实继承了Enum类

面试题：请解释enum和Enum的区别？

enum是一个关键字，而Enum是一个抽象类；

使用enum定义的枚举就相当于一个类继承了Enum这个抽象类；

## 章节34 课时 135 03116\_枚举（定义其它结构）

```
package cn.mldn.demo;
```

```
enum Color{//定义好了枚举
    //大写表示此处为实例化对象,枚举对象必须要放在首行,随后才可以定义属性、构造、普通方法。
    RED("红色"),GREEN("绿色"),BLUE("蓝色");//由于构造方法有参数,所以这里要传递参数。
    private String title;//属性
    private Color(String title){
        this.title=title;
    }
    public String toString(){//覆写toString()方法
        return this.title;
    }
}
public class TestDemo {
    public static void main(String[] args) {
        for(Color c:Color.values()){
            System.out.println(c);//直接输出对象调用toString()
        }
    }
}
```

此时与之前定义的多例设计模式操作方式完全相同,而且代码更加的简单。枚举还可以实现接口。

范例:枚举实现接口

```
package cn.mldn.demo;
interface Message{
    public String getTitle();
}
enum Color implements Message{
    RED("红色"),GREEN("绿色"),BLUE("蓝色");
    private String title;
    private Color(String title){
        this.title=title;
    }
    public String toString(){
        return this.title;
    }
    public String getTitle() {
        return this.title;
    }
}
public class TestDemo {
    public static void main(String[] args) {
        Message msg=Color.RED;
        System.out.println(msg.getTitle());
    }
}
```

枚举里面最变态的一种做法是可以在每一个对象后面使用匿名内部类的形式实现抽象方法。

范例:另外一种接口的实现

```
package cn.mldn.demo;
interface Message{
    public String getTitle();
}
enum Color implements Message{//定义好了枚举
    //大写表示此处为实例化对象,枚举对象必须要放在首行,随后才可以定义属性、构造、普通方法。
    RED("红色"){
        public String getTitle(){
            return "自己的"+this;
        }
    },GREEN("绿色"){
        public String getTitle(){
            return "自己的"+this;
        }
    },BLUE("蓝色"){
        public String getTitle(){
            return "自己的"+this;
        }
    };
    //由于构造方法有参数,所以这里要传递参数。
    private String title;//属性
    private Color(String title){
        this.title=title;
    }
}
```

```

    public String toString(){//覆写toString()方法
        return this.title;
    }
}
public class TestDemo {
    public static void main(String[] args) {
        Message msg=Color.RED;
        System.out.println(msg.getTitle());
    }
}

```

更变态的是，在枚举里面还能够直接定义抽象方法。此时每一个枚举对象必须分别覆写抽象方法；

范例：定义抽象方法并覆写

```

package cn.mldn.demo;
enum Color{
    RED("红色"){
        public String getTitle(){
            return "自己的"+this;
        }
    },GREEN("绿色"){
        public String getTitle(){
            return "自己的"+this;
        }
    },BLUE("蓝色"){
        public String getTitle(){
            return "自己的"+this;
        }
    };
    private String title;
    private Color(String title){
        this.title=title;
    }
    public String toString(){
        return this.title;
    }
    public abstract String getTitle();
}
public class TestDemo {
    public static void main(String[] args) {
        System.out.println(Color.RED.getTitle());
    }
}

```

#### 章节34 课时 136 03117\_枚举（枚举应用）

```

package cn.mldn.demo;
enum Color{
    RED, GREEN, BLUE;
}
public class TestDemo {
    public static void main(String[] args) {
        Color c=Color.RED;
        switch(c){//支持枚举判断
            case RED:
                System.out.println("这是红色");
                break;
            case GREEN:
                System.out.println("这是绿色");
                break;
            case BLUE:
                System.out.println("这是蓝色");
                break;
        }
    }
}

```

下面编写一个程序，来试图努力用一下枚举。

范例：定义程序

```

package cn.mldn.demo;
enum Sex{
    MALE("男"), FEMALE("女");
}

```

```

private String title;
private Sex(String title){
    this.title=title;
}
public String toString(){
    return this.title;
}
}
class Person{
    private String name;
    private int age;
    private Sex sex;
    public Person(String name,int age,Sex sex){
        this.name=name;
        this.age=age;
        this.sex=sex;
    }
    public String toString(){
        return "名字："+this.name+"，年龄："+this.age+"，性别："+this.sex;
    }
}
public class TestDemo {
    public static void main(String[] args) {
        System.out.println(new Person("阿佑",20,Sex.FEMALE));
    }
}

```

不适用枚举也同样可以实现上述的功能。

总结：

- 1、枚举属于高级的多例设计模式；
- 2、如果你没习惯于使用枚举，那么就别习惯了。如果你已经习惯了，那么继续。

### 章节35 课时 137 03118\_Annotation (简介)

- 1、Annotation的作用；
- 2、三种内置的Annotation的使用

具体内容：

对于软件程序的开发实际上经过了三个发展过程：

第一个过程：将所有与配置相关的内容直接写到代码之中；

第二个过程：将配置与程序代码独立，即：程序运行的时候根据配置文件进行操作；

最严重的问题：一个项目里面配置文件过多，根本就无法查询错误；

第三个过程：配置信息对于用户而言无用，而且胡乱的修改还会导致程序错误，所以可以将配置信息写回到程序里面，但是利用一些明显的标记来区分配置信息与程序。

Annotation是JDK1.5最大的特殊，利用朱姐的句式来实现程序的不同功能实现。

在JavaSE里面支持自动以Annotation的开发，并且提供了三个最为常用的基础Annotation：@Override、@Deprecated、@SuppressWarnings

### 章节35 课时 138 03119\_Annotation (准确覆写)

准确的覆写：**@Override**

如果再输出对象的时候希望其可以返回需要的内容，那么一定要进行toString()方法的覆写。

范例：有没有可能在开发中出现以下的开式

```

package cn.mldn.demo;
class Book{
    public String toString(){
        return "这是一本书。";
    }
}
public class TestDemo {
    public static void main(String[] args) {
        System.out.println(new Book());
    }
}

```

这个时候很明显并没有成功的进行toString()方法的覆写。但是这个错误无法在编译的时候发现，只能在程序运行的时候发现。

所以此时为了告诉编译器，toString()应该是覆写的方法，那么就可以加上@Override,明确的告诉编译器，这个方法应该是覆写来的，如果不是报错。

```

package cn.mldn.demo;
class Book{
    @Override//只要正确的覆写，那么就不会出现编译的语法错误。
    public String toString(){
        return "这是一本书。";
    }
}

```

```

    public String toString(){//报错The method toString() of type Book must override or implement a supertype method
        return "这是一本书。";
    }
}
public class TestDemo {
    public static void main(String[] args) {
        System.out.println(new Book());
    }
}

```

其实覆写的时候可以不用手动覆写，在类中输入to然后按下alt+/就会自动覆写如下：

```

package cn.mldn.demo;
class Book{
    @Override
    public String toString() {
        // TODO Auto-generated method stub
        return super.toString();
    }
}
public class TestDemo {
    public static void main(String[] args) {
        System.out.println(new Book());
    }
}

```

### 章节35 课时 139 03120\_Annotation ( 过期声明 )

#### @Deprecated

如果说现在有一个专门负责完成某些功能的工具包，里面有一个Hello类，在Hello类里面有一个fun()方法，在所有项目最初的发展阶段，fun()方法非常的完善，但是后来随着开发技术的不断加强，发现fun()这个方法的功能不足，这个时候对于开发者有两个选择：

选择1：直接在新版本的工具包里面取消掉fun()方法，同时直接给出fun2()方法；

选择2：在新版本的开发包里保存fun()方法，但是通过某种途径告诉新的开发者fun()方法有问题，并且提供fun2()这个新的方法提供开发者使用。

很明显，第二种会比较合适，因为第二种的作法可以兼顾已使用项目的情况。

这个时候我就可以使用@Deprecated来进行声明。

范例：声明过期操作

```

1 package cn.mldn.demo;
2 class Book{
3     @Deprecated
4     public void fun(){
5     }
6 }
7 public class TestDemo {
8     public static void main(String[] args) {
9         Book book=new Book();
10        book.fun();
11    }
12 }

```

利用此操作可以很好的实现方法功能的新旧交替。

### 章节35 课时 140 03121\_Annotation ( 压制警告 )

#### @SuppressWarnings

可以压制多个警告

```

package cn.mldn.demo;
class Book<T>{
    private T title;
    public void setTitle(T title) {
        this.title = title;
    }
}
public class TestDemo {
    public static void main(String[] args) {
        Book book=new Book();
        book.setTitle("Hello");
    }
}

```

此时出现多个警告

那么如果说现在是在开发者故意留下的警告信息，但是又不希望其总是提示警告。就可以选择压制警告。

范例：压制警告

现在相当于取消了警告的提示信息。

```
8 public class TestDemo {
9     public static void main(String[] a
10         Book book=new Book();
11         book.setTitle("Hello");
12     }
```

光标在图中划线部分按住ctrl+1即可弹出压制信息，选择即可出现下例红色代码。

package cn.mldn.demo;

```
class Book<T>{
    private T title;
    public void setTitle(T title) {
        this.title = title;
    }
    public T getTitle() {
        return title;
    }
}

public class TestDemo {
    @SuppressWarnings({ "rawtypes", "unchecked" })
    public static void main(String[] args) {
        Book book=new Book();
        book.setTitle("Hello");
    }
}
```

总结：

- 1、可以发现如果使用了开发工具，所有的Annotation实际上都不需要我们去编写；
- 2、清楚三个Java的Annotation作用。

## 章节36 课时 141 03122\_接口定义增强

使用default与static定义接口方法

从Java发展之初到今天已经经过20年的时间了，在这20年底额时间里面所有的开发者都知道java的接口由全局常量和抽象方法组成，但是从JDK1.8的时代这一组成改变了。

如果说现在有某一个接口，这个接口锁着时间的发展已经产生了2万个子类。突然有一天发现，这个接口里面的方法不足，应该再增加一个方法，而针对于所有不同的子类，这个方法的功能实现是完全一样的。按照我们最初的做法应该再每一个子类上都覆写这个新的方法，那么就要修改2万个子类。

所以为了解决这样的问题，允许在接口里面定义普通方法了。但是如果定义普通方法就必须明确的使用default来进行定义。

package cn.mldn.demo;

```
interface IMessage{//定义了一个接口
    public void print();//接口里原本定义的抽象方法
    default void fun(){//接口里新增的普通方法
        System.out.println("毁三观的方法出现了！");
    }
}

class IMessageImpl implements IMessage{
    @Override
    public void print() {
        // TODO Auto-generated method stub
    }
}

public class TestDemo {
    public static void main(String[] args) {
        IMessage msg=new IMessageImpl();
        msg.fun();
    }
}
```

除了使用default定义方法之外，还可以使用static定义方法，一旦使用static定义方法意味着这个方法只可以由类名称调用。

范例：定义static方法

package cn.mldn.demo;

```
interface IMessage{//定义了一个接口
    public void print();//接口里原本定义的抽象方法
    default void fun(){//接口里新增的普通方法
        System.out.println("毁三观的方法出现了！");
    }
}
```



```

static void get(){
    System.out.println("直接由接口调用");
}
}
class IMessageImpl implements IMessage{
    @Override
    public void print() {
        // TODO Auto-generated method stub
    }
}
public class TestDemo {
    public static void main(String[] args) {
        IMessage msg=new IMessageImpl();
        msg.fun();
        IMessage.get();
    }
}

```

在JDK1.8有一个最重要的概念：内部类访问方法参数的时候可以不加final关键字，所有出现的这些新特性，完全打破了Java已有的代码组成形式。

接口里面使用default或static定义方法的意义是避免子类重复实现同样的代码；

接口的使用还应刻以抽象方法为主。

### 章节36 课时 142 03123\_Lamda表达式

1、分析函数式编程的产生原因

2、掌握函数编程的语法

具体内容

Lamda属于函数式编程的概念，那么为什么需要函数式的编程呢？

如果想清楚函数式编程的产生目的，那么必须通过匿名内部类来分析。

范例：传统的匿名内部类

匿名内部类的来源见102课：定义一个接口，只能通过子类实例化，但是假如子类只用一次，所以定义为匿名内部类，匿名内部类语法麻烦，所以用Lamda表达式。

```

package cn.mldn.demo;
interface IMessage{//定义了一个接口
    public void print();
}
public class TestDemo {
    public static void main(String[] args) {
        fun(new IMessage(){
            public void print(){
                System.out.println("Hello World!");
            }
        });
    }
    public static void fun(IMessage msg){
        msg.print();
    }
}

```

实际上整个代码之中，如果是fun()方法，最终需要的只是一个输出而已，但是由于java开发的结构性完整性的要求，所以不得不在这个核心的语句上嵌套更多的内容。

以上的做法要求的实在是过于严谨了，所以在JDK1.8时代引入了函数式的编程，可以简化以上的代码。

范例：使用Lamda表达式

```

package cn.mldn.demo;
interface IMessage{//定义了一个接口
    public void print();
}
public class TestDemo {
    public static void main(String[] args) {
        fun()->System.out.println("Hello World!");
    }
    public static void fun(IMessage msg){
        msg.print();
    }
}

```

整个操作里面匿名内部类只是进行一行语句的输出，所以此时使用Lamda表达式可以非常轻松的实现输出要求。

对于Lamda语法有三种形式：

(参数) -> 单行语句;  
(参数) -> {单行语句};  
(参数) -> 表达式

范例：观察有参的单行语句

```
package cn.mldn.demo;
interface IMessage{//定义了一个接口
    public void print(String str);
}
public class TestDemo {
    public static void main(String[] args) {
        //首先要定义此处表达式里面需要接收变量，单行语句直接输出。
        fun((s)->System.out.println(s));
    }
    public static void fun(IMessage msg){
        msg.print("Hello World!");//设置参数内容
    }
}
```

范例：多行语句

```
package cn.mldn.demo;
interface IMessage{//定义了一个接口
    public void print(String str);
}
public class TestDemo {
    public static void main(String[] args) {
        //首先要定义此处表达式里面需要接收变量，单行语句直接输出。
        fun((s)->{
            s=s.toUpperCase();//转大写
            System.out.println(s);
        });
    }
    public static void fun(IMessage msg){
        msg.print("Hello World!");//设置参数内容
    }
}
```

如果说现在代码里面只是一个简单的计算表达式，那么操作也可以很容易。

```
package cn.mldn.demo;
interface IMessage{//定义了一个接口
    public int add(int x,int y);
}
public class TestDemo {
    public static void main(String[] args) {
        //首先要定义此处表达式里面需要接收变量，单行语句直接输出。
        fun((s1,s2)->s1+s2);
    }
    public static void fun(IMessage msg){
        System.out.println(msg.add(10,20));//设置参数内容
    }
}
```

如果现在只是一个表达式，那么进行操作的返回，还是写return比较合适，是多行的时候才可以考虑写上return。

总结：

利用Lambda表达式最终解决的问题：避免了匿名内部类定义过多无用的操作。

## 章节36 课时 143 03124\_方法引用

掌握四种方法引用的使用

具体内容：

一直以来都只是在对象上能够发现引用的身影，而对象引用的特点：不同的对象可以操作同一块内容，所谓的方法引用就是指为一个方法设置别名，相当于一个方法定义了不同的名字。

方法引用在Java8之中一共定义了四种形式：

引用静态方法：        类名称    ::    static方法名称  
引用某个对象的方法： 实例化对象:: 普通方法  
引用特定类型的方法： 特定类    ::    普通方法  
引用构造方法：        类名称    ::    new

范例：引用静态方法

```
在String类里面有一个valueOf()方法 public static String valueOf(int x);
package cn.mldn.demo;
/**
```

```

* 实现方法的引用接口
* @param <P>引用方法的参数类型
* @param <R>引用方法的返回类型
*/
interface IMessage<P,R>{
    public R zhuanhuan(P p);
}
public class TestDemo {
    public static void main(String[] args) {
        //将String.valueOf()方法变为了IMessage接口里的zhuanhuan()方法
        IMessage<Integer,String> msg=String::valueOf;
        String str=msg.zhuanhuan(1000);
        System.out.println(str.replaceAll("0","9"));
    }
}

```

范例：普通方法引用

```

package cn.mldn.demo;
interface IMessage<R>{
    public R upper();
}
public class TestDemo {
    //String类的toUpperCase()定义：public String toUpperCase()
    //这个方法没有参数，但是有返回值，并且这个方法一定要在有实例化对象的情况下才可以调用
    //“Hello”字符串是String类的实例化对象，所以可以直接调用toUpperCase()方法
    //将toUpperCase()函数应用交给了IMessage接口
    public static void main(String[] args) {
        IMessage<String> msg="Hello"::toUpperCase;
        String str=msg.upper();//相当于"Hello"::toUpperCase
        System.out.println(str.toUpperCase());
    }
}

```

上例入如果在接口里多定义一个方法就不能调用执行了。

所以为了保证被引用接口里面只能有一个方法，那么就需要增加一个注解的声明。@FunctionalInterface

```

package cn.mldn.demo;
@FunctionalInterface//此为函数式接口，只能定义一个方法
interface IMessage<R>{
    public R upper();
}
public class TestDemo {
    //String类的toUpperCase()定义：public String toUpperCase()
    //这个方法没有参数，但是有返回值，并且这个方法一定要在有实例化对象的情况下才可以调用
    //“Hello”字符串是String类的实例化对象，所以可以直接调用toUpperCase()方法
    //将toUpperCase()函数应用交给了IMessage接口
    public static void main(String[] args) {
        IMessage<String> msg="Hello"::toUpperCase;
        String str=msg.upper();
        System.out.println(str.toUpperCase());
    }
}

```

在进行方法引用的过程里面还有另外一种形式的引用（它需要特定类的对象支持），正常情况下如果使用了“类:方法”，引用的一定是类中的静态方法，但是这种形式也可以引用普通方法。

例如：在String类里面有个方法public int compareTo(String anotherString)

如果要进行比较的话，比较的形式：字符串1对象.compareTo(字符串2对象);也就是说如果真要引用这个方法就需要准备从两个参数。

范例：引用特定类的方法

```

package cn.mldn.demo;
@FunctionalInterface//此为函数式接口，只能定义一个方法
interface IMessage<P>{
    public int compare(P p1,P p2);
}
public class TestDemo {
    public static void main(String[] args) {
        IMessage <String> msg=String::compareTo;
        System.out.println(msg.compare("A","B"));
    }
}

```

与之前相比，方法引用前不再需要定义对象，而是可以理解为将对象定义在了参数上。

范例：引用构造方法

```
package cn.mldn.demo;
@FunctionalInterface//此为函数式接口，只能定义一个方法
interface IMessage<C>{
    public C create(String t,double p);
}
class Book{
    private String title;
    private double price;
    public Book(String title,double price){
        this.title=title;
        this.price=price;
    }
    @Override
    public String toString(){
        return "书名："+this.title+"，价格："+this.price;
    }
}
public class TestDemo {
    public static void main(String[] args) {
        IMessage<Book> msg=Book :: new;//引用构造方法
        //调用的虽然是create()，但是这个方法引用的是Book类的构造
        Book book=msg.create("Java开发",20.2);
        System.out.println(book);
    }
}
```

四种方法引用基本的形式如上，但是是否真的去使用，现在给你的答案：NO

### 章节36 课时 144 03125 内建函数式接口

观察在JDK1.8中提供的新的函数式接口包以及提供的四个函数式接口。

具体内容：

对于方法的引用，严格来讲都需要定义一个接口，可是不管如何操作，实际上有可能操作的接口只有四种，在JDK1.8里面提供了一个包：

java.util.function，提供有以下四个核心接口：

- 1、**功能型接口**（Function）：**public interface Function<T,R>{public R apply(T t);}**  
此接口需要接收一个参数，并且返回一个处理结果；
- 2、**消费型接口**（Consumer）：**public interface Consumer<T>{public void accept(T t);}**  
此接口只是负责接收数据（引用数据是不需要返回），并且不返回处理结果。
- 3、**供给型接口**（Supplier）：**public interface Supplier<T>{public T get();}**  
此接口不接收参数，但是可以返回结果；
- 4、**断言型接口**（Predicate）：**public interface Predicate<T>{public boolean test(T t);}**  
进行判断操作使用

所有在JDK1.8之中由于存在有以上的四个功能性的接口，所以一般很少会由用户去定义新的函数式接口。

范例：观察函数式接口——接收参数并且返回一个处理结果，String类有一个方法：public boolean startsWith(String str)。

```
package cn.mldn.demo;
import java.util.function.Function;
public class TestDemo {
    public static void main(String[] args) {
        Function<String,Boolean> fun="##Hello" ::startsWith;
        System.out.println(fun.apply("##"));
    }
}
```

范例：消费型接口

```
package cn.mldn.demo;
import java.util.function.Consumer;
class MyDemo{
    public void print(String str){//此方法没有返回值但是有参数
        System.out.println(str);
    }
}
public class TestDemo {
    public static void main(String[] args) {
        Consumer<String>cons=new MyDemo()::print;
        cons.accept("Hello World!");
    }
}
```

范例：供给型接口

引用String类中的toUpperCase()方法：public String toUpperCase();  
package cn.mldn.demo;

```
import java.util.function.Supplier;
public class TestDemo {
    public static void main(String[] args) {
        Supplier <String> sup="Hello"::toUpperCase;
        System.out.println(sup.get());
    }
}
```

范例：断言型接口

String类里面有一个equalsIgnoreCase()方法

```
package cn.mldn.demo;
```

```
import java.util.function.Predicate;
```

```
public class TestDemo {
```

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

```
        Predicate <String> pre="Hello"::equalsIgnoreCase;
```

```
        System.out.println(pre.test("hello"));//这是一个不区分比较，所以结果是true
```

```
    }
```

```
}
```

这几个接口包含了所有可能出现的方法引用，也是函数式接口的代表，但是又许多的接口与它类似。

有了这几个函数式接口，那么开发中就不再需要了。

所有讲解的一切都要为最后的数据流准备。

### 章节37 课时 145 04001\_线程与进程

Java是一门为数不多的多线程支持的编程语言。

如果要想解释多线程之前首先需要知道什么叫进程？在操作系统的定义中，进程指的是一次程序的完整运行。在这个运行的过程之中内存、处理器、IO等资源操作都要为这个进程进行服务。

在最早的DOS的时代，有一个特点：如果你的电脑病毒发作了，那么你的电脑几乎就不能动了。因为所有的资源都被病毒软件所占用，其他的程序无法抢占这个资源。但是后来到了Windows时代，这一情况彻底发生了改变，因为电脑即使有病毒了，那么电脑也可以运行（就是慢点）。

Windows属于多进程的操作系统。但是有一个问题出现了，每一个进程都需要有资源的支持，那么这么多个进程怎么分配资源呢？

在同一个时间段上，会有多进程轮流去抢占资源，但是在某一个时间点上只有一个进程在运行。

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。进程如果消失了线程一定会消失，反之如果线程消失了，进程未必消失。

### 章节37 课时 146 04002\_多线程的实现（Thread类实现）

掌握java中三种多线程的实现方式（JDK1.5之后增加了第三种）

具体内容：

如果要想在Java之中实现了多线程有两种途径：

继承Thread类；

实现Runnable接口（Callable接口）

继承Thread类

Thread类是一个支持多线程的功能类，只要有一个子类它就可以实现多线程的支持。

```
package cn.mldn.demo;
```

```
class MyThread extends Thread{//这就是一个多线程的操作类
```

```
}
```

```
public class TestDemo {
```

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

```
    }
```

```
}
```

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()方法，也就是说在每个多线程的主体类之中都必须覆写Thread类中提供的run()方法。

```
public void run(){} 
```

这个方法上没有返回值，也就表示了线程一旦开始就要一直执行，不能够返回内容。

```
package cn.mldn.demo;
```

```
//线程操作主类
```

```
class MyThread extends Thread{//这就是一个多线程的操作类
```

```
    private String name;//定义类中的属性
```

```
    public MyThread (String name){//定义构造方法
```

```
        this.name=name;
```

```
    }
```

```
    @Override
```

```
    public void run() { //覆写run()方法，作为线程的主体操作方法
```

```
        for(int x=0;x<200;x++){
```

```
            System.out.println(this.name+"→"+x);
```

```
        }
```

```

        super.run();
    }
}
public class TestDemo {
    public static void main(String[] args) {
        MyThread mt1=new MyThread("线程1");
        MyThread mt2=new MyThread("线程2");
        MyThread mt3=new MyThread("线程3");
        mt1.run();
        mt2.run();
        mt3.run();
    }
}

```

}//结果是按顺序执行的，输出完线程1的1-199才执行下一个，并不是多线程。

本线程类的功能是进行循环的输出操作，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()定义的）

```

package cn.mldn.demo;
//线程操作主类
class MyThread extends Thread{//这就是一个多线程的操作类
    private String name;//定义类中的属性
    public MyThread (String name){//定义构造方法
        this.name=name;
    }
    @Override
    public void run() {//覆写run()方法，作为线程的主题操作方法
        for(int x=0;x<200;x++){
            System.out.println(this.name+"→"+x);
        }
        super.run();
    }
}
public class TestDemo {
    public static void main(String[] args) {
        MyThread mt1=new MyThread("线程1");
        MyThread mt2=new MyThread("线程2");
        MyThread mt3=new MyThread("线程3");
        mt1.start();
        mt2.start();
        mt3.start();
    }
}
}
//结果是交替执行，抢资源执行的。

```

此时每一个线程对象交替执行。

疑问？为什么多线程启动不是调用run()而**必须调用start()**？

打开Java的源代码，来观察一下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) {
            /* do nothing. If start0 threw a Throwable then
               it will be passed up the call stack */
        }
    }
}
}
private native void start0();

```

首先发现在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，按照道理来讲应该是使用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，因为此异常属于RuntimeException子类，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()方法里面要调用一个start0()方法，而且此方法与结构与抽象方法类似，使用了native声明，在Java开发里面有一门技术成为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分配，所以此操作严格来讲主要是有JVM负责根据不同的操作系统而实现的。

即：使用Thread类的start()方法不仅仅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。

### 章节37 课时 147 04003\_多线程的实现（Runnable接口实现）

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，针对于类的继承都应该是回避的问题，那么多线程也一样，为了解决单继承的限制，在Java里面专门提供了Runnable接口，此接口定义如下：

@FunctionalInterface

public interface Runnable{//函数式接口的最大特点是一个接口只能定义一个方法

public void run();//在接口里面任何的方法都是public定义的权限，不存在默认的权限

}

那么只需要让一个类实现Runnable接口即可，并且需要覆写run()方法。

与基础Thread类相比，此时的MyThread类在结构上与之前是没有区别的，但是有一点是有严重区别的，如果此时继承了Thread类，那么可以直接继承了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要依靠Thread类完成，在Thread类里面定义有以下的构造方法：

public Thread(Runnable target),接收的是Runnable接口对象；

范例：启动多线程

package cn.mldn.demo;

//线程操作主类

class MyThread implements Runnable{//这就是一个多线程的操作类

private String name;//定义类中的属性

public MyThread (String name){//定义构造方法

this.name=name;

}

@Override

public void run() { //覆写run()方法，作为线程的主题操作方法

for(int x=0;x<200;x++){

System.out.println(this.name+"→"+x);

}

}

}

public class TestDemo {

public static void main(String[] args) {

MyThread mt1=new MyThread("线程1");

MyThread mt2=new MyThread("线程2");

MyThread mt3=new MyThread("线程3");

new Thread(mt1).start();

new Thread(mt2).start();

new Thread(mt3).start();

}

}

此时就避免了单继承局限，那么也就是说实际工作中使用Runnable接口是最合适的。

### 章节37 课时 148 04004 多线程的实现（两种实现方式的区别）

多线程两种实现方式的区别？（面试题）

通过讲解已经清楚了多线程的两种实现方式，那么这两种方式有哪些区别呢？

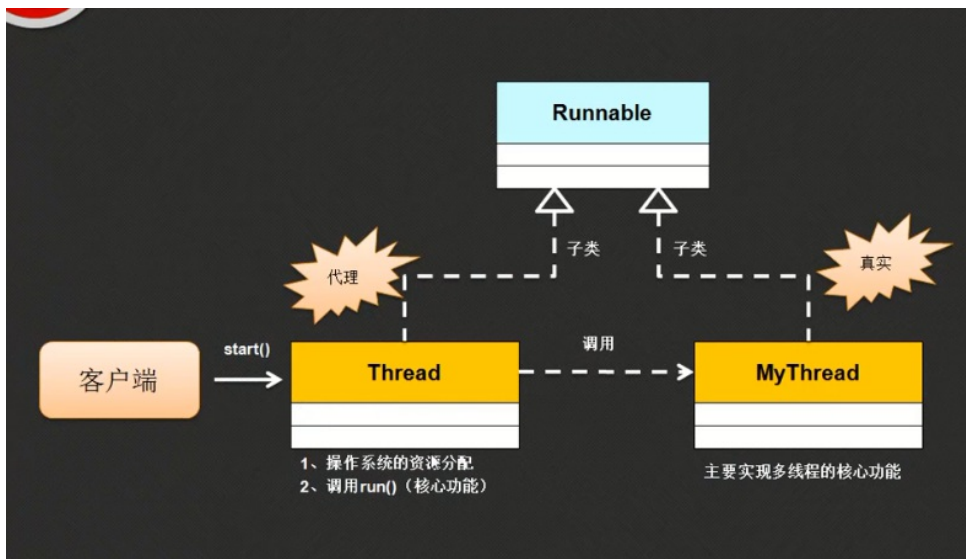
首先一定要明确的是，使用Runnable接口与Thread类相比，解决了单继承的定义局限，所以不管后面的联系与区别是什么，至少这一点已经下了死定义——如果要使用一定使用Runnable接口。

首先来观察一下Thread类的定义

public class Thread extends Object implements Runnable

发现Thread类实现了Runnable接口，这样一来





此时整个的定义结构看起来非常像代理设计模式，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，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。

数据共享是指多个线程访问同一资源的操作。

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;
```

```
//线程操作主类
```

```
class MyThread extends Thread{//这就是一个多线程的操作类
```

```
    private String name;//定义类中的属性
```

```
    private int ticket=10;
```

```
    @Override
```

```
    public void run() { //覆写run()方法，作为线程的主题操作方法
```

```
        for(int x=0;x<200;x++){
```

```
            if(this.ticket>0){
```

```
                System.out.println("卖票ticket"+this.ticket--);
```

```
            }
```

```
        }
```

```
    }
```

```
}
```

```
public class TestDemo {
```

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

```
        //由于MyThread类有start()方法，所以每一个MyThread类对象就是一个线程对象，可以直接启动
```

```
        MyThread mt1=new MyThread();
```

```
        MyThread mt2=new MyThread();
```

```
        MyThread mt3=new MyThread();
```

```
        mt1.start();
```

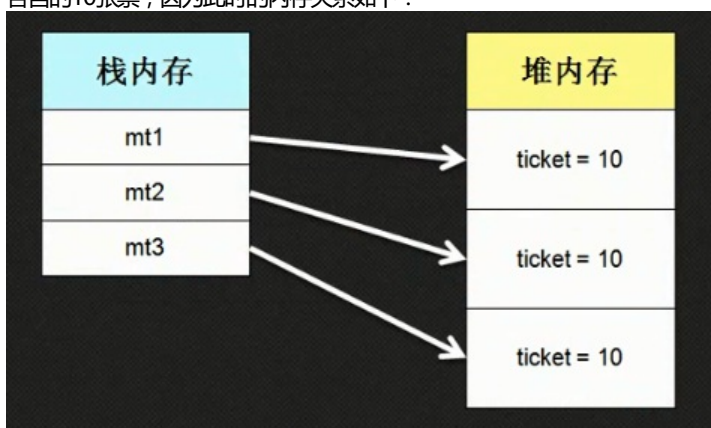
```
        mt2.start();
```

```
        mt3.start();
```

```
    }
```

```
}
```

本程序声明了三个MyThread类的对象，并且分别调用了三次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各自的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;
```



//线程操作主类

```
class MyThread implements Runnable{//这就是一个多线程的操作类
```

```
    private String name;//定义类中的属性
```

```
    private int ticket=10;
```

```
    @Override
```

```
    public void run() { //覆写run()方法，作为线程的主体操作方法
```

```
        for(int x=0;x<200;x++){
```

```
            if(this.ticket>0){
```

System.out.println("卖票ticket"+this.ticket--); //有个问题，如果“卖票”改为“卖票”，10张票各执行一次，但是还原之后如果不保存直接执行并保存的第一次，会随机多卖一张，之后就都正常了。

```
            }
```

```
        }
```

```
    }
```

```
}
```

```
public class TestDemo {
```

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

```
        MyThread mt=new MyThread();
```

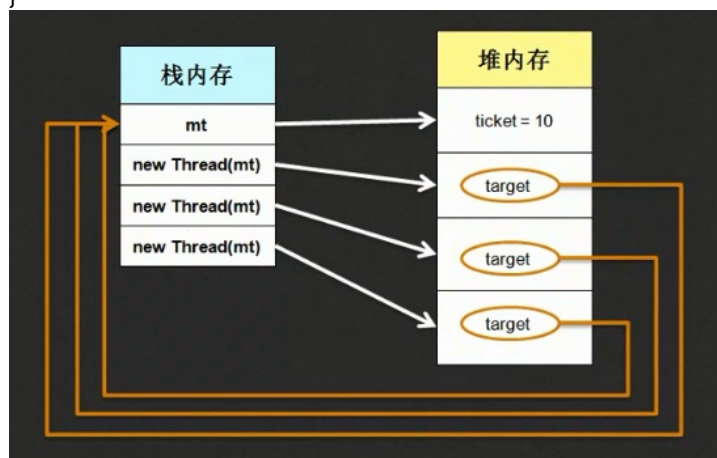
```
        new Thread(mt).start();
```

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

```
        new Thread(mt).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了同一个MyThread类的对象引用，也就是说着三个线程对象都直接访问同一个数据资源。

注：上述代码用Thread也可以实现，但是MyThread继承了Thread类之后本身就有start()方法，还要调用Thread类的start()方法，并不合适。而用Runnable就好了，Runnable并没有start()方法。

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

答：Thread类是Runnable接口的子类，使用Runnable接实现多线程可以避免单继承局限；

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

面试题：请写出多线程两种实现操作

把Thread类递减的方式和Runnable接口实现的方式代码都写出来。

### 章节37 课时 149 04005\_多线程的实现（Callable接口）

```
java.util.concurrent
```

使用Runnable接口实现的多线程可以避免单继承局限，的确很好，但是有一个问题，Runnable接口里面的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Callable接口。

```
@FunctionalInterface
```

```
public interface Callable<V>{
```

```
    public V call() throws Exception;
```

```
}
```

call()方法执行完线程的主体功能之后可以返回一个结果，而返回结果的类型由Callable接口上的泛型来决定。

范例：定义一个线程主体类

```
class MyThread implements Callable<String>{//点中类名称按ctrl+1就会覆写继承类中的方法
```

```
    private int ticket=10;
```

```
    @Override
```

```
    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:

```
public class FutureTask<V> extends Object implements RunnableFuture<V>
```

```
public interface RunnableFuture<V> extends Runnable, Future<V>
```

在FutureTask类里面有如下的构造方法:

```
FutureTask(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。

```
package cn.mldn.demo;
```

```
import java.util.concurrent.Callable;
```

```
import java.util.concurrent.ExecutionException;
```

```
import java.util.concurrent.FutureTask;
```

```
//线程操作主类
```

```
class MyThread implements Callable<String>{//点中类名称按ctrl+1就会覆写继承类中的方法
```

```
    private int ticket=10;
```

```
    @Override
```

```
    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!";
```

```
    }
```

```
}
```

```
public class TestDemo {
```

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

```
        MyThread mt1=new MyThread();
```

```
        MyThread mt2=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类的构造来接收task对象
```

```
        new Thread(task1).start();//启动多线程
```

```
        new Thread(task2).start();
```

```
        //多线程执行完毕后取得内容, 依靠FutureTask的父接口Future中的get()方法完成
```

```
        System.out.println("A线程的返回结果: "+task1.get());
```

```
        System.out.println("B线程的返回结果: "+task2.get());
```

```
    }
```

```
}
```

总结:

最麻烦的问题在与需要接收返回值信息, 并且又要与原始的多线程的实现靠拢 (Thread类靠拢)。

1、对于多线程的实现, 重点在于Runnable接口与Thread类启动的配合上;

2、对于JDK1.5新特性, 了解就行了, 知道区别就在于返回结果上。

### 章节38 课时 150 04006 多线程的常用操作方法 (命名和取得)

多线程里面有很多的方法定义, 但是大部分的方法都是在Thread类里面定义, 强调几个与我们开发有关的方法:

1、线程的命名与取得

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, 那么就必须要依靠线程的名字。对于线程的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。

如果要想进程线程名称的操作, 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)//使用final表示不能被覆写

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类里面提供一个方法:

取得当前线程对象: public static Thread currentThread()//用static声明的, 表示可以用类名称Thread直接调用。

范例: 观察线程的命名

```
package cn.mldn.demo;
```

```
class MyThread implements Runnable{
```

```
    @Override
```

```
    public void run() {
```

```
        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();
        new Thread(mt).start();
        new Thread(mt).start();
    }
}

```

输出结果：

Thread-0

Thread-1

Thread-2

得出结论：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{
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName());
    }
}
public class TestDemo {
    public static void main(String[] args) throws Exception {
        MyThread mt=new MyThread();
        new Thread(mt,"自己的线程A").start();
        new Thread(mt).start();
        new Thread(mt,"自己的线程B").start();
        new Thread(mt).start();
        new Thread(mt).start();
    }
}

```

执行结果：

自己的线程A

Thread-0

Thread-1

自己的线程B

Thread-2

下面来观察如下的一个程序执行：

```

package cn.mldn.demo;
class MyThread implements Runnable{
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName());
    }
}
public class TestDemo {
    public static void main(String[] args) throws Exception {
        MyThread mt=new MyThread();
        new Thread(mt,"自己的线程对象").start();
        mt.run();//直接调用run()方法
    }
}

```

执行结果：

main

自己的线程对象

得出结论：原来主方法就是一个线程（main线程），那么所有在主方法上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，线程实际上一一直都存在（主方法就是主线程），可是进程都去哪里了呢？

```

public class TestDemo{
    public static void main(String args[]){
        for(int x=0;x<2000000000L;x++){
            ;
        }
    }
}

```

打开系统任务管理器，执行java TestDemo

Groove 音乐	0%	0.3 MB
Host Process for Setting Syn...	0%	0.5 MB
Java(TM) Platform SE binary	0%	8.8 MB
Lenovo Drivers Management...	0%	1.5 MB

由此可得出：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。

### 章节38 课时 151 04007\_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//long声明日期时间，声明文件大小

范例：观察休眠特点

package cn.mldn.demo;

class MyThread implements Runnable{

    @Override

    public void run() {

        for(int x=0;x<10000;x++){

            try {

                Thread.sleep(1000);

            } catch (InterruptedException e) {

                // TODO Auto-generated catch block

                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,"自己的线程对象A").start();

        new Thread(mt,"自己的线程对象B").start();

        new Thread(mt,"自己的线程对象C").start();

        new Thread(mt,"自己的线程对象D").start();

        new Thread(mt,"自己的线程对象E").start();

    }

}

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。）

### 章节38 课时 152 04008\_多线程的常用操作方法（线程优先级）

所谓的优先级指的是越高的优先级，越有可能先执行。在Thread类里面提供有以下两种优先级：

设置优先级：//[pri:ˈɒrəti]

public final void setPriority(int newPriority)

取得优先级：

public final int getPriority()

发现设置和取得 优先级都是使用了int属于类型，对于此内容有三种取值：

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

package cn.mldn.demo;

class MyThread implements Runnable{

    @Override

    public void run() {

        for(int x=0;x<10000;x++){

            try {

                Thread.sleep(2000);

            } catch (InterruptedException e) { //终端异常

                // TODO Auto-generated catch block

                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,"自己的线程对象A");
        Thread t2=new Thread(mt,"自己的线程对象B");
        Thread t3=new Thread(mt,"自己的线程对象C");
        t1.setPriority(Thread.MAX_PRIORITY);
        t2.setPriority(Thread.MIN_PRIORITY);
        t3.setPriority(Thread.MIN_PRIORITY);
        t1.start();
        t2.start();
        t3.start();
    }
}

```

范例：主线程优先级是多少？

```

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

```

输出结果为5，主线程属于中等优先级。

总结：

- 1、Thread.currentThread()可以取得当前线程类对象；
- 2、Thread.sleep()主要是休眠，感觉是一起休眠，但实际上是有先后顺序的；
- 3、优先级越高的线程对象越有可能先执行。

### 章节39 课时 153 04009 线程的同步与死锁（同步问题引出）

- 1、线程的同步产生原因；
- 2、线程的同步处理操作；
- 3、线程的死锁情况。

具体内容（了解）：

#### 1、同步问题的引出

实际上所谓的同步指的就是多个线程访问同一资源时所需要考虑的问题。

```
package cn.mldn.demo;
```

```

class MyThread implements Runnable{
    private int ticket=5;//一共有五张票
    @Override
    public void run() {
        for(int x=0;x<20;x++){
            if(this.ticket>0){
                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();
    }
}

```

现在四个线程对象应该一起销售出5张票。此时没有出现是因为在一个JVM进程下运行，并且没有收到任何影响，那么如果要想观察到问题，可以加入一个延迟来看。

```
package cn.mldn.demo;
```

```

class MyThread implements Runnable{
    private int ticket=5;//一共有五张票
    @Override
    public void run() {
        for(int x=0;x<20;x++){
            if(this.ticket>0){
                try {
                    Thread.sleep(100);
                }
            }
        }
    }
}

```



```

        new Thread(mt,"票贩子D").start();
    }
}
范例：使用同步方法
package cn.mldn.demo;
class MyThread implements Runnable{
    private int ticket=60;//一共有五张票
    @Override
    public void run() {
        for(int x=0;x<200;x++){
            this.sale();
        }
    }
    public synchronized void sale(){//同步方法
        if(this.ticket>0){
            try {
                Thread.sleep(100);
            } 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();
    }
}

```

同步操作与异步操作相比，异步操作的执行速度要高于同步操作，但是同步操作时数据的安全性较高，属于安全的线程操作。

### 章节39 课时 155 04011 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。下面编写一个程序，本程序没有任何实际意义，目的只是让大家看一下死锁的具体情况。

```

package cn.mldn.demo;
class A{
    public synchronized void say(B b){
        System.out.println("我说：把你的本给我，我把笔给你，否则不给。");
    }
    public synchronized void get(){
        System.out.println("我：得到了本，失去了笔，还是什么都干不了。");
    }
}
class B{
    public synchronized void say(A a){
        System.out.println("姜说：把你的笔给我，我把本给你，否则不给。");
    }
    public synchronized void get(){
        System.out.println("姜：得到了笔，失去了本，还是什么都干不了。");
    }
}
public class TestDemo implements Runnable{//主类
    private static A a=new A();
    private static B b=new B();
    public static void main(String[] args) throws Exception {
        new TestDemo();
    }
    public TestDemo(){
        new Thread(this).start();
        b.say(a);
    }
    public void run() {

```

```

    a.say(b);
}
}

```

以上的代码只是为了说明死锁而举的一个最无用的例子，本程序没有任何的可参考性。  
死锁是程序开发之中由于某种逻辑上的错误所造成问题，并且不是简单的就会出现。

面试题：请解释多个线程访问同一资源时需要考虑哪些情况？有可能带来哪些问题？

多个线程访问同一资源时一定要处理好同步，可以使用同步代码块或同步方法来解决；

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。

总结：

- 1、最简单的理解同步和异步的操作那么就可以通过synchronized来实现；
- 2、死锁是一种不定的状态。

#### 章节40 课时 156 04012\_综合实战：生产者与消费者（问题引出）

知识点：

- 1、生产者和消费者问题的产生
- 2、Object类对多线程的支持

具体内容：

生产者和消费者指的是不同的线程类对象，操作同一资源的情况，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；

生产者每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。

package cn.mldn.demo;

```

class Info{
    private String title;
    private String content;
    public void setTitle(String title) {
        this.title = title;
    }
    public String getTitle() {
        return title;
    }
    public void setContent(String content) {
        this.content = content;
    }
    public String getContent() {
        return content;
    }
}

class Productor implements Runnable{
    private Info info;
    public Productor(Info info){
        this.info=info;
    }
    @Override
    public void run() {
        for(int x=0;x<100;x++){
            if(x%2==0){//偶数
                this.info.setTitle("王惊雷");
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
                this.info.setContent("好学生一枚");
            }else{
                this.info.setTitle("萌动物");
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
        }
    }
}

```



```

        this.info.setContent("草泥马");
    }
}

}

class Customer implements Runnable{
    private Info info;
    public Customer(Info info){
        this.info=info;
    }
    @Override
    public void run() {
        for(int x=0;x<100;x++){
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
            System.out.println(this.info.getTitle()+"-"+this.info.getContent());
        }
    }
}

public class TestDemo{//主类
    public static void main(String[] args) throws Exception {
        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发现两个严重问题：

数据错位，发现不再是一个所需要的完整数据；

数据重复取出，数据重复设置。

#### 章节40 课时 157 04013\_综合实战：生产者与消费者（同步处理）

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;
    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);
    }
    public void setTitle(String title) {
        this.title = title;
    }
    public String getTitle() {
        return title;
    }
    public void setContent(String content) {

```

```

        this.content = content;
    }
    public String getContent() {
        return content;
    }
}
class Productor implements Runnable{
    private Info info;
    public Productor(Info info){
        this.info=info;
    }
    @Override
    public void run() {
        for(int x=0;x<100;x++){
            if(x%2==0){//偶数
                this.info.set("王惊雷","好学生一枚");
            }else{
                this.info.set("萌动物","草泥马");
            }
        }
    }
}
class Customer implements Runnable{
    private Info info;
    public Customer(Info info){
        this.info=info;
    }
    @Override
    public void run() {
        for(int x=0;x<100;x++){
            this.info.get();
        }
    }
}
public class TestDemo{//主类
    public static void main(String[] args) throws Exception {
        Info info=new Info();
        new Thread(new Productor(info)).start();
        new Thread(new Customer(info)).start();
    }
}

```

此时数据的错位问题很好的得到了解决，但是重复操作问题更加严重了。

#### 章节40 课时 158 04014\_综合实战：生产者与消费者（Object类支持）

解决重复的问题：

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(long timeout)throws InterruptedException

唤醒第一个等待线程：

唤醒全部等待线程，哪个优先级高就先执行：public final void notifyAll()

```

package cn.mldn.demo;
class Info{
    private String title;
    private String content;
    private boolean flag=true;
    //flag=true 表示可以生产，但是不可以取走
    //flag=false 表示可以取走，但是不可以生产
    public synchronized void set(String title,String content){
        //重复进入到了set()方法里面，发现不能够生产，所以要等待
        if(this.flag==false){
            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;//修改生产标记
        super.notify();//唤醒其他等待线程
    }

    public synchronized void get(){
        if(this.flag==true){//还没生产呢
            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;
        super.notify();
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getTitle() {
        return title;
    }

    public void setContent(String content) {
        this.content = content;
    }

    public String getContent() {
        return content;
    }
}

class Productor implements Runnable{
    private Info info;
    public Productor(Info info){
        this.info=info;
    }
    @Override
    public void run() {
        for(int x=0;x<100;x++){
            if(x%2==0){//偶数
                this.info.set("王惊雷","好学生一枚");
            }else{
                this.info.set("萌动物","草泥马");
            }
        }
    }
}

class Customer implements Runnable{
    private Info info;
    public Customer(Info info){
        this.info=info;
    }
    @Override
    public void run() {
        for(int x=0;x<100;x++){
            this.info.get();
        }
    }
}

```

```
    }  
  }  
}  
public class TestDemo{//主类  
    public static void main(String[] args) throws Exception {  
        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()进行唤醒。

总结：

这是一个非常经典的多线程的处理模型。所以掌握它属于你个人能力的额一个提升，同时可以更加理解Object类的作用。

