# 黑马程序员入学 Java 知识——精华总结

**J2SE 部分，Java 高新技术部分，7K 面试题部分等黑马入学要求的知识点总结！**

# 一、黑马程序员—java 概述与基础知识

## 1、何为编程？

编程就是让计算机为解决某个问题而使用某种程序设计语言编写程序代码，并最终得到结果的过程。

为了使计算机能够理解人的意图，**人类就必须要将需解决的问题的思路、方法、和手段通过计算机能够理解的形式告诉计算机，**使得计算机能够根据人的指令一步一步去工作，完成某种特定的任务。**这种人和计算机之间交流的过程就是编程。**

## 2、Java 语言概述，历史、特点

是 SUN(Stanford University Network，斯坦福大学网络公司)1995 年推出的一门高级语言。
是一种面向 Internet 的编程语言。
随着 Java 技术在 web 方面的不断成熟，已经成为 Web 应用程序的首选开发语言。
是**简单易学，完全面向对象，安全可靠，与平台无关**的编程语言。

**java5.0 之后的三大技术框架**

**J2EE(Java 2 Platform Enterprise Edition)企业版**

在 jdk5.0 版本后称为 JAVAEE,是为开发企业环境下的应用程序提供的一套解决方案。该技术体系中包含的技术如 Servlet Jsp 等，主要针对于 Web 应用程序开发。是传智播客就业班和黑马训练营的主要学习内容之一.。

**J2SE（Java 2 Platform Standard Edition）标准版**

在 jdk5.0 版本后称为 JAVASE,这是在 java 基础阶段主要学习的内容,也是 java 的基础,以后不管从事 Android 开发或者是物联网+云计算的开发,等是建立在 JSE 基础上的,因此该技术是 java 的最核心技术,是传智播客基础班的主要上课内容.。

**J2ME(Java 2 Platform Micro Edition)小型版**

在 jdk5.0 版本以后称为 JAVAME,该技术多应用于一些电子产品的嵌入式开发,以前在手机开发上应用的也比较多,但是随着智能手机的发展,现在手机应用程序(比如 Android 程序)的开发已经不再使用该技术。

## 3、什么是跨平台性？原理是什么？JVM

所谓跨平台性，是指 java 语言编写的程序，一次编译后，可以在多个系统平台上运行。

实现原理：Java 程序是通过 java 虚拟机在系统平台上运行的，只要该系统可以安装相应的 java 虚拟机，该系统就可以运行 java 程序。（注意不是能在所有的平台上运行，关键是该平台是否能安装相应的虚拟机）。

**我的总结：Java 程序之所以能够实现跨平台运行，是因为它根本就不直接运行在任何底层平台上，而是需要在哪里运行，就在哪里(如 Windows 平台)事先准备好自己的 Java 平台，而这只是仅仅是安装和配置一个软件而已！**

## 4、Jre 和 Jdk 的区别?

**JRE:**（Java Runtime Environment），**java 运行环境**。包括 Java 虚拟机(JVM Java Virtual Machine)和 Java 程序所需的核心类库等，如果想要运行一个开发好的 Java 程序，计算机中只需要安装 JRE 即可。

**JDK:**（Java Development Kit Java）**开发工具包**。JDK 是提供给 Java 开发人员使用的，其中包含了 java 的开发工具，也包括了 JRE。所以安装了 JDK，就不用在单独安装 JRE 了。

**其中的开发工具：编译工具(javac.exe) 打包工具(jar.exe)等**

简单而言：使用 JDK 开发完成的 java 程序，交给 JRE 去运行。

**我的总结：必须熟练的记忆，核心类库，开发工具！**

## 5、java 虚拟机 JVM

Java Virtual Machine ，简称 JVM;

它是运行所有 Java 程序的抽象计算机,是 Java 语言的运行环境，它是 Java 最具吸引力的特性之一，JVM 读取并处理编译过的与平台无关的字节码(class)文件。

Java 编译器针对 JVM 产生 class 文件，因此是独立于平台的。

Java 解释器负责将 JVM 的代码在特定的平台上运行。

Java 虚拟机是不跨平台的.

## 6、Java 程序运行机制


编译: javac 文件名.文件后缀名

运行: java 类名

**我的总结：Java 程序的组成：Java 源文件，字节码文件。**

## 7、学习 java 注意的地方

Java 语言拼写上**严格区分大小写**;

一个 Java 源文件里可以定义多个 Java 类，**但其中最多只能有一个类被定义成 public 类；**

**若源文件中包括了 public 类，源文件必须和该 public 类同名；**

一个源文件中包含 N 个 Java 类时，编译后会生成 N 份字节码文件，即每个类都会生成一份单独的 class 文件，且字节码文件名和

其对应的类名相同;

**我的总结：一个 Java 源文件只定义一个类,不同的类使用不同的源文件定义;**

**将每个源文件中单独定义的类都定义成 public 的;**

**保持 Java 源文件的主文件名与源文件中的类名一致;**

## 8、Java 语法格式

任何一种语言都有自己的语法规则,Java 也一样,既然是规则,那么知道其如何使用就可以了。
代码都定义在类中,类由 class 来定义，区分 public class 和 class;
代码严格区分大小写,如 main 和 Main 是不一样的;
Java 中的标识符与关键字;
注释;

main 方法的作用：
程序的入口
保证程序的独立运行
被 JVM 调用

## 9、代码注释：单行//，多行/* */，文档注释/** */

1.单行注释 //:
//后到本行结束的所有字符会被编译器忽略;
2.多行注释 /* */:
/* */之间的所有字符会被编译器忽略
3.文档注释 /** */:
在/** */之间的所有字符会被编译器忽略,java 特有的(用于生成文档);
**我的总结：多行和文档注释都不能嵌套使用。**

## 10、Java 中的标识符

可简单理解为在 Java 程序中为了增强阅读性自定义的名称。比如：类名，方法名，变量名等。
命名规则：

(1) 由字母、数字、下划线、$组成，不能以数字开头
注意:此处的字母还可以是中文,日文等;
(2) 大小写敏感
(3) 不得使用 java 中的关键字和保留字
(4) 别用 Java API 里面的类名作为自己的类名。

## 11、java 中的常量和变量

变量的概念：
占据着内存中的某一个存储区域;
该区域有自己的名称（变量名）和类型（数据类型）;
该区域的数据可以在同一类型范围内不断变化;

为什么要定义变量：

用来不断的存放同一类型的常量，并可以重复使用；
使用变量注意：
变量的作用范围, 初始化值
定义变量的格式：
**数据类型　　变量名　=　初始化值；**
注：格式是固定的，记住格式，以不变应万变。
作用范围：定义开始到定义它的代码块结束;
同一范围内，不允许多个个局部变量命名冲突

## 12、Java 成员变量和局部变量

局部变量：不是声明在类体括号里面的变量；
局部变量使用前必须初始化值;
局部变量没有默认初始化值;
局部变量的作用域是从定义开始到定义它的代码块结束;

成员变量:在方法体外,类体内声明的变量，又称字段(Field)或全局变量；（其实 Java 中没有全局变量，由于 Java 是面向对象语言，所有变量都是类成员）
成员变量的作用域是整个类中;
我的总结：注意成员变量和局部变量的区别

```
package reviewDemo;

public class Demo2 {
  public static void main(String[] args) {
        int i;
            System.out.println(i);
    }
}
```

备注：局部变量调用前没有初始化，所以编译的时候就会报错！

```
package reviewDemo;

public class Demo2 {
   static int i;
      public static void main(String[] args) {
        System.out.println(i);
    }
```

```
}
```

备注：此时 i 为全局变量，未初始化会被赋予默认的初始化值！程序正确！

**我的总结：**
**局部变量是定义在方法中的变量,,,出了该方法就不能访问该变量了....**
**成员变量是在类中定义,,,,并且在类的成员方法中都能访问的变量..**

## 13、基本数据类型

在数据类型中，最常用也是最基础的数据类型，被称作基本数据类型。可以使用这些类型的值来代表一些简单的状态。
Java 语言的基本数据类型总共有以下 8 种，下面是按照用途划分出的 4 个类别：

```
整数型：byte(字节型)、short(短整型)、int(整型)、long(长整型)
小数型：float(单精度浮点型)、double(双精度浮点型)
字符型：char(字符型)
布尔型：boolean（布尔型）
```

**定点类型：**
整数型是一类代表整数值的类型。当需要代表一个整数的值时，可以根据需要从 4 种类型中挑选合适的，如果没有特殊要求的话，一般选择 int 类型。4 种整数型区别主要在每个数据在内存中占用的空间大小和代表的数值的范围。

整数型参数表：

| 类型名称关键字 | 占用空间(字节) | 取值范围 | 默认值 |
|---|---|---|---|
| 字节型 byte | 1 | -2的7次方到2的7次方减一 (128) | 0 |
| 短整型 short | 2 | -2的15次方到2的15次方减一 (32768) | 0 |
| 整型 int | 4 | -2的31次方到2的31次方减一 (65535) | 0 |
| 长整型 long | 8 | -2的63次方到2的63次方减一 (太长可以忽略) | 0 |

**浮点类型：**
小数型是一类代表小数值的类型。当需要代表一个小数的值时，可以根据需要从以下 2 种类型中挑选合适的。如果没有特殊要求，一般选择 double 类型。
由于小数的存储方式和整数不同，所以小数都有一定的精度，所以在计算机中运算时不够精确。根据精度和存储区间的不同，设计了两种小数类型。

小数型参数表

| 类型名称关键字 | | 占用空间(字节) | 取值范围 | 默认值 |
|---|---|---|---|---|
| 单精度浮点型 | float | 4 | -3.4E+38—3.4E+38 | 0.0f |
| 双精度浮点型 | double | 8 | -1.7E+308—1.7E+308 | 0.0 |

**字符型：**
字符型代表特定的某个字符，按照前面介绍的知识，计算机中都是以字符集的形式来保存字符的，所以字符型的值实际只是字符集中的编号，而不是实际代表的字符，由计算机完成从编号转换成对应字符的工作。Java 语言中为了更加方便国际化，使用 Unicode 字符集作为默认的字符集，该字符集包含各种语言中常见的字符。在程序代码中，字符使用一对单引号

加上需要表达的字符来标识，例
如’A’、’a’等，当然也可以直接使用字符编码，也就是一个非负整数进行表示。

| 字符型参数表 | | | |
|---|---|---|---|
| 类型名称关键字 | 占用空间(字节) | 取值范围 | 默认值 |
| 字符型char | 2 | 0到2的16次方减一 | ’\u0000’（代表空格） |

**布尔型：**
布尔型代表逻辑中的成立和不成立。Java 语言中使用关键字 true 代表成立，false 代表不成立。布尔型是存储逻辑值的类型，其实很多程序中都有逻辑值的概念，Java 把逻辑的值用布尔型来进行表达。

| 布尔型参数表 | | | |
|---|---|---|---|
| 类型名称关键字 | 占用空间(字节) | 取值范围 | 默认值 |
| 布尔型boolean | 1 | true或false | false |

## 14、基本数据类型转换之向上转型和向下转换

向上转换：
整型，字符型，浮点型的数据在混合运算中相互转换，转换时遵循以下原则：
容量小的类型可**自动转换**为容量大的数据类型；
byte,short,char → int → long → float → double
byte，short，char 之间不会相互转换，他们在计算时首先会转换为 int 类型。
boolean 类型是不可以转换为其他基本数据类型。
Eg：
int i = 123;
long l = i;//自动转换，不需强转
float f = 3.14F;
double d = f;

向下转换：
整型，字符型，浮点型的数据在混合运算中相互转换，转换时遵循以下原则：
容量小的类型可自动转换为容量大的数据类型；
byte,short,char → int → long → float → double
byte，short，char 之间不会相互转换，他们在计算时首先会转换为 int 类型。
boolean 类型是不可以转换为其他基本数据类型。
Eg：
long l = 123L;
int i = (int) l;//必须强转
double d = 3.14;
float f = (float) d;

**我的总结：类型转化**
**小转大，自动！自动类型转换（也叫隐式类型转换）**
**大转小，强转！强制类型转换（也叫显式类型转换）**

## 15、Java 的运算符

运算符是一种特殊符号，用以表示数据的运算、赋值和比较等共分以下几种：

算术运算符（+ 、— 、* 、/ 、%）
赋值运算符（= 、+=、-=、*=、/=,、%=）
关系运算符（>、>=、<、<=、!=）
条件运算符（&&、‖、！ &、|、^）
位运算符（&、|、^、~、>>、<<、<<<、>>>）

**我的总结：**
**逻辑运算符用于连接布尔型表达式，在 Java 中不可以写成 3<x<6，应该写成 x>3 & x<6 。**
**“&”和“&&”的区别：单与时，左边无论真假，右边都进行运算；双与时，如果左边为真，右边参与运算，如果左边为假，那么右边不参与运算。**
**“|”和“||”的区别同理，双或时，左边为真，右边不参与运算。**
**“ ^ ”与“|”不同之处是：当左右都为 true 时，"^"结果为 false。**

## 16、表达式和三目运算符

是由数字、运算符、数字分组符号(括号)等以能求得数值的有意义排列的序列;
a + b
3.14 + a
(x + y) * z + 100
boolean b= i < 10 && (i%10 != 0)
表达式的类型和值：
对表达式中操作数进行运算得到的结果是表达式的值。
表达式值的数据类型即为表达式的类型。
表达式的运算顺序
应按照运算符的优先级从高到低的顺序进行;
优先级相同的运算符按照事先约定的结合方向进行;

“三目运算符”，语法格式：
**x ? y : z;**
其中 x 为 boolean 类型表达式，先计算 x 的值，若为 true,则整个三目运算符的结果是表达式 y 的值，否则就是 z 的值。

```
package reviewDemo;

public class Demo2 {
    public static void main(String[] args) {
        String  s = "lz";
        s = (s == "lz" ? "正确" : "错误");
```

```
        System.out.println(s);
    }
}
```
输出：正确

## 17、程序流程控制

顺序结构
分支结构（条件结构）
循环结构
控制循环结构

顺序结构：
如果代码里没有流程控制，程序是至上而下一行一行执行的，一条语句执行完之后继续执行下一条语句，直到程序的最后。

if 语句:
基本语法:if(表达式){方法体}else if(表达式){方法体}else(方法体)
几种形式:
三种格式：
if(条件表达式){执行语句；}
if(条件表达式){执行语句；}else{执行语句；}
if(条件表达式){执行语句；}else if (条件表达式){执行语句；}……else{执行语句；}

```
public class If3
{
    public static void main(String args[])
    {
        int i = 3;
        if (i > 5)
        {
            System.out.println("变量 i 大于 5");
        }
        else if (i > 4)
        {
            System.out.println("变量 i 小于 4");
        }
        else
        {
            System.out.println("其他");
        }
    }
}
```

switch 控制语句

格式：

```
switch(表达式)
{
    case  表达式可能的结果值 1:
            执行语句；
    break；
    case  表达式可能的结果值 2:
            执行语句；
    break；
        ...
    default:
            执行语句；
    break；//最后这个一般不写
}
```

备注：

case 之间与 default 没有顺序。先执行第一个 case，都没有匹配的 case 值执行 default。

结束 switch 语句的两种情况：遇到 break，switch 语句执行结束。

如果匹配的 case 或者 default 没有对应的 break，那么程序会从第一次匹配的 case 语句开始继续向下执行，运行可以执行的语句，直到遇到 break 或者 switch 结束为止。

```
class switch2
{
    public static void main(String[] args)
    {
        int a = 7,b = 5;
        switch(a-b)
        {
            case 3://表达式可能的值；
            {
                System.out.println("33!");
            }
            break;

            case 4:
            {
                System.out.println("44!");
                }
                break;
                default://都没有执行则  执行该语句!
                    System.out.println("其它的");

        }
    }
}
```

注意：switch 语句只能使用 byte、char、short、int 四种基本类型以及它们的包装类和枚举

## 18、三大循环结构:

用于处理需要重复执行的操作；
根据判断条件的成立与否，决定程序段落的执行次数，而这个程序段落我们称为循环体；
**while:事先不需要知道循环执行多少次；**
**do while:同上，只是至少要执行一次(先做，后判断)；**
**for:需要知道循环次数；**

**循环结构(while & do while)**
while 语句
格式：
```
while(条件表达式值为 true)
{
    执行语句；
}
```

do while 语句
格式：
```
do
{
    执行语句；
}
while(条件表达式值为 true);
```

**我的总结：do while 特点是无论条件是否满足，循环体至少被执行一次。**
**循环里的条件表达式不能直接写 false 或直接写结果是 false 的表达式,不过可以使用变量来传递 false 值;**

**循环结构(for)**
格式：
```
for(初始化表达式（1）；循环条件表达式（2）；循环后的操作表达式（3）)
{
        执行语句；（4）
}
```
**执行顺序：(1) → (2) → (4) → (3) → (2) → (4) → (3)**

备注：
1, for 里面的 3 个表达式运行的顺序，初始化表达式只读一次，判断循环条件，为真就执行循环体，然后再执行循环后的操作表达式，接着继续判断循环条件，重复找个过程，直到条件不满足为止。

2, while 与 for 可以互换，区别在于 for 为了循环而定义的变量在 for 循环结束就在内存中释放。而 while 循环使用的变量在循环结束后还可以继续使用。
3, 最简单无限循环格式：while(true) , for(;;),无限循环存在的原因是并不知道循环多少次，而是根据某些条件，来控制循环。

Eg：
用三种循环控制求出 100 以内前 5 个 3 的倍数的数字；

```
class while
{
	public static void main(String[] args)
	{
		int i = 1,k = 0;
		while(i<=100)
		{
			if(i%3==0)
			{
			if(k<5)
			System.out.println(i);
			k++;
			}
			i++;
		}
	}
}

class dowhile
{
	public static void main(String[] args)
	{
		int i = 1,k = 0;
		do{
			if(i%3==0)
			{
			if(k<5)
			System.out.println(i);
			k++;
			}
			i++;
		}
		while(i<=100);
	}
}
```

```
class for
{
    public static void main(String[] args)
    {
        int i = 1,k = 0;
        for(;i<100;i++)
        {
            if(i%3==0&&k<5)
            {
            System.out.println(i);
            k++;
            }
        }
    }
}
```

## 19、嵌套循环与流程控制

嵌套循环：循环里套循环
假设外循环的循环次数是 m 次,内循环的循环次数是 n 次，那么内层循环的循环次数需要 m * n 次。

Eg：利用 for 循环语句的嵌套打印出**乘法口诀表**

```
class break1
{
    public static void main(String[] args)
    {
        for(int i=1;i<=10;i++)//定义 i 与 j 的值必须在 for 循环里面，否则每次跳出循环重新执行就无法取值
        {
            for(int j=1;j<=10;j++)
            {
                if(j<=i)
                System.out.print(j+"*"+i+"="+i*j+" ");//小注意:print()括号里面必须传递参数,println()无限制!
            }
                System.out.println();
        }
    }
}
```

**流程控制**

break 语句、 continue 语句；

break：终止该层循环；
continue：跳过该层循环
注：
①：若这两个语句离开应用范围，存在是没有意义的。
②：这个两个语句后面都不能有语句，因为执行不到。
③：continue 语句是跳过本次循环，继续下次循环。
④：标签的出现，可以让这两个语句作用于指定的循环。

Eg：

```
package reviewDemo;

public class Demo2 {
    public static void main(String[] args) {
        int i = 1;
        for (; i < 10; i++) {
            System.out.println(i);
            if(i == 8){
                break;//流程控制，不会完全执行！
            }
        }
    }
}
```

# 二、黑马程序员—方法与数组

第二篇：

## 1、什么是方法（Method）？

**方法是一组为了实现特定功能的代码块的集合。**方法在语法上的功能主要有以下两个：
①：**结构化代码**

将代码按照功能进行组织，使代码的结构比较清晰，容易阅读和修改，也就是程序的可维护性强。

②：**减少代码重复**

一个固定的功能，可能会在程序中多次使用，在使用时只需要调用写好的方法，而不用重复书写对应的功能代码。

方法在书写时需要注意以下两点：

**①：逻辑严谨**

方法实现的一个完整的功能，所以在书写时要考虑到各种可能的情况，并对每种情况做出恰当的处理。

**②：通用性强（可重复利用）**

方法实现的是一种功能，在实际实现时，可以根据需要，使方法具备一定的**通用性**，除非必要，否则不要写专用的方法。在 Java 语言中，恰当的使用方法，将使程序更加优雅，便于阅读和使用。

**我的总结：**

**方法：一段可重复使用的代码段；**

**程序中完成独立功能的一段代码的集合。**

## 2、Java 中方法的格式：

```
[修饰符] 返回值类型 方法名([参数类型 形式参数 1，参数类型 形式参数 2，……])
{
    执行语句;    [return 返回值;]//需要的话
}
```

参数列表（参数的类型 ，参数的个数，参数的顺序）

**我的总结：只要上述有一个不一样，那么这个参数列表就不一样！对于方法而言，即使同名也不是同一个方法，也就是下面讲的方法签名。**

## 3、方法里的属性

访问控制符：访问控制符限定方法的可见范围，或者说是方法被调用的范围。方法的访问控制符有四种，按可见范围从大到小依次是：public、protected，无访问控制符,private。其中无访问控制符不书写关键字即可。具体的范围在后续有详细介绍。

形式参数：在方法被调用时用于接收外界输入的数据。

实参：调用方法时实际传给方法的数据。

返回值：方法在执行完毕后返还给调用它的环境的数据。

返回值类型：事先约定的返回值的数据类型，如无返回值，必须给出返回类型 void。

**方法签名：方法名和方法的参数列表(能区别方法)；//最关键的**

java 语言中调用方法：对象名.方法名（实参列表）。

实参的数目、数据类型和次序必须和所调用方法声明的形参列表匹配。

return 语句终止方法的运行并指定要返回的数据。

## 4、方法特点

它可以实现独立的功能;
必须定义在类里面;
它只有被调用才会执行;
它可以被重复使用;
方法结束后方法里的对象失去引用;

如何定义一个功能,并通过方法体现出来:

① 明确该功能运算后的结果.明确返回值类型;
② 明确在实现该功能过程中是否有未知内容参与运算，确定参数列表;

Eg:

```
package reviewDemo;
/**
 * player 的值分别是1,2,3或者其他值时，该方法返回“大明”,“二明”,“小明”,“我没有这个孩子”。
 */

public class Demo2 {
    public static void main(String[] args) {
        int player = 5;
        System.out.println("返回的结果是: "+show(player));
    }

    public static String show(int p){
        switch (p) {
        case 1:
            return "大明";
        case 2:
            return "二明";
        case 3:
            return "小明";
        default:
            return "我没有这个孩子！";
        }
    }
}
```

## 5、什么是方法的重载（Overload）?

概念:

**在同一个类中，允许存在一个以上的同名方法，只要它们的参数个数或者参数类型或参数顺序不同即可。**

存在的原因：

**屏蔽了一个对象的同一类方法由于参数不同所造成的差异。**

特点：

与返回值类型无关，只看参数列表。

以下是方法重载的示例：

**public void a(int a){}**
**public int a(){}**
**public void a(int a,String s){}**

Eg：

```
class Chongzai
{
    public static void main(String[] args)
    {
        int ret = name(3,4,5);
        System.out.println(ret);
    }
    public static int name(int a, int b)
    {
        int c = a+b;
        return c;
    }
    public static int name(int a, int b, int m)//这就叫做方法的重载,只要参数列表不一样。好处：用户使用时限制很小；
    {
        int c = a+b+m;
        return c;
    }
}
```

**我的总结：方法重载，方法名相同，但是参数列表不一致！（和方法覆写不一样，方法覆写的方法名和参数列表都必须一样），注意方法重载和方法覆写的不同！**

## 6、可变参数

从 java5 开始出现了可变参数，这是对 java 方法及数组的拓展！

方法中可以接受的参数不再是固定个数的，而是随着具体需求传递的多少来决定。

**定义格式： 返回值类型 方法名(参数类型 ... 形式参数){ }**

可变参数的**特点**：

**只能出现在参数列表的最后；**

... 位于变量类型和变量名之间，前后有无空格都可以;
调用可变参数的方法时，编译器为该可变参数隐含创建一个数组，在方法体中以数组的形式访问可变参数。

Eg：

```
package reviewDemo;

import java.util.Arrays;

public class Demo6 {
    public static void main(String[] args) {
        int []age = {1,2,3,69,36,636,459};
        show(age);
    }

    public static void show(int...a){//此时这里写的就是可变参数，和数组的作用一样!
        //把数组给打印出来!
        System.out.println(Arrays.toString(a));
    }
}
```

## 7、数组简介：

数组(Array)是 Java 语言中内置的一种基本数据存储结构，通俗的理解，就是一组数的集合，目的是用来一次存储多个数据。数组是程序中实现很多算法的基础，可以在一定程度上简化代码的书写。

备注：

1. 数组的好处：数组里的每个元素都有编号，编号从 0 开始，并且依次递增，方便操作这些元素；
2. 使用 Java 数组：必须先声明数组，再给该数组分配内存；
3. 数组对应在内存中一段连续空间。
4. 数组元素必须是相同数据类型，也可以是引用数据类型，但是同一个数组中的元素必须是同一类数据类型。

## 8、一维数组的声明与初始化：

一维数组：可以理解为一列多行、类型相同的数据，其中每个数据被称为数组元素；
一维数组的声明方式：

type varName[]; 或 type[] varName;(推荐)

Eg：int age[]; int []age;

数组的长度一旦确定,就不能改变,也就数组是定长的;

**我的总结：java 语言声明数组的时候不能指定其长度（元素的个数）**

Eg：int a[5]; //非法

**初始化：**

Java 中的数组必先初始化才可以使用，所谓初始化就是为数组的数组元素分配内存，并为每个数组元素赋值；

数组初始化的两种方式：

**静态初始化：**初始化时由我们自己指定每个数组元素的初始值，由系统决定需要的数组长度；

格式：数组名 = new 数组类型[]{元素 1,元素 2,元素 3...元素 n};

简化语法：数组名 = {元素 1,元素 2,元素 3...元素 n};

**动态初始化：**初始化时由我们指定数组的长度，由系统为数组元素分配初始值；

格式：数组名 = new 数组类型[数组长度];

**我的总结：**

**不能静态和动态初始化同时使用：**

**也就是说不能同时指定数组的长度和元素；**

Eg:

```
package reviewDemo;
/**
 * 声明数组及初始化
 */
public class Demo3 {
    public static void main(String[] args) {
        int []age = new int[10];
        //动态初始化
        for (int i = 0; i < age.length; i++) {
            age[i] = i;
            System.out.print(age[i]+"    ");
        }
    }
}
```

输出：0  1  2  3  4  5  6  7  8  9

## 9、数组的使用：

Java 语言的数组索引是从 0 开始的，也就是说数组里的第一个元素的索引是 0,第二个元素的索引是 1，依次可以类推。

**常见操作：**

给数组元素赋值

数组名[索引] = 数组类型的值 ；

访问数组元素
数组类型 变量 = 数组名[索引];
得到数组的长度
int len = 数组名.length; //length 是数组的属性
遍历数组（备注：length 属性和循环语句）
数组元素的索引范围（0，长度—1）

Eg：**判断数组是否重复**

```
package reviewDemo;
/**
 * 判断数组是否重复
 */
public class Demo3 {
    public static void main(String[] args) {
        int []age = {1,2,3,4,5,6,5};
        for (int i = 0; i < age.length-1; i++) {//双层循环，定住一个，
再考虑下一个！
            for (int j = i+1; j < age.length; j++) {
                if(age[i] == age[j]){
                    System.out.println("有重复的！"+i+"  "+j);
                    break;
                }
            }
        }
    }
}
```

**求最大值：**

```
package reviewDemo;

public class Demo4 {
    public static void main(String[] args) {
        int age[] = new int[] { 12, 26, 3, 60, 55, 6, 48, 4, 98 };
        int max = age[0];
        for (int i = 0; i < age.length; i++) {
            if (max < age[i]) {
                max = age[i];
            }
        }
        System.out.println(max);

    }
}
```

经典用法：冒泡法排序

```
class Bubblesort
{
    public static void main(String args[])
    {
        int [] arr={5,1,6,4,2,8,9};
        bubble(arr);
        printarray(arr);

    }
    public static void bubble(int[] arr)
    {
        for (int i=0;i<arr.length-1 ;i++ )
        {
            for (int y=0;y<arr.length-i-1 ; y++) //让每一次比较的元素减
少，-1是为了防止数组角标越界；
            {
                if(arr[y]>arr[y+1])  //相邻两元素相比
                {
                    int temp = 0;
                    temp = arr[y];
                    arr[y] = arr[y+1] ;
                    arr[y+1] = temp;
                }
            }
        }
    }
    public static void printarray(int[] arr)
    {

        for (int i=0;i<arr.length ;i++ )
        {
            if(i!=arr.length-1)
            System.out.print(arr[i]+",");
            else
                System.out.println(arr[i]);
        }
    }

}
//选择排序
public class Demo6 {
    public static void main(String[] args) {
        int []age = {1,2,36,363,56,95,12,32,1232,3263};
```

```
        for (int i = 0; i < age.length; i++) {
            for (int j = i+1; j <= age.length-1; j++) {
                if(age[i] > age[j]){
                    int temp = age[i];
                    age[i] = age[j];
                    age[j] = temp;
                }
            }
        }
        System.out.println(Arrays.toString(age));
    }
}
//输出为：[1, 2, 12, 32, 36, 56, 95, 363, 1232, 3263]
```

## 10、分配内存空间

分配内存空间
数组名=new 数据类型[数组长度];
**new 关键字用来实现为数组或对象分配内存 （堆内存）**
数组具有固定的长度。获取数组的长度： 数组名.length

定义数组+分配内存空间
数据类型[]数组名=new 数据类型[数组长度];
定义数组时不指定长度，分配空间时指定数组长度；如：String cities[] = new String[6];

数组元素：
数组名[下标值];
数组下标从 0 开始 scores[0]=75；

数组的内存分配 栈内存和堆内存
如定义一个数组 int[]scores 将在栈内存中为 scores 分配内存空间，其值是一个不确定的值。当执行语句 scores=new int[5]时，将在堆内存分配连续 5 个空间，每个空间 4 个字节，用于存放整型数据，其初始值为 0，然后将该段空间首地址，也就是第一个元素的地址，比如 0*3000，赋给 scores 变量。该地址相当于一个指针，指向堆内存中分配的空间。此时堆内存中分配的 5 个空间可以分别使用 scores[0],一直到 scores[4]来表示。当执行四个赋值语句时，分别用指定值填充到对应元素位置。如果此时将 null 值赋给 scores 时，scores 变量将不再指向任何位置，此时堆内存中分配的空间就变成了垃圾，由垃圾回收器在某一时间进行回收。

在方法中定义的变量，包括基本数据类型变量和引用数据类型变量，都将在栈内存中分配空间，当超过变量作用范围后，自动回收

**我的总结：初始化=定义数组+分配空间+赋值**

## 11、二维数组：

**二维数组：(其实是一个一维数组，它的每一个元素又是一个一维数组),**
可以看做是一张表格。
初始化：

**动态初始化**

int[ ][ ]  arr = new int[3][2];
定义了一个二维数组，其中有 3 个一维数组，每一个一维数组中有 2 个元素

**静态初始化**

int[ ][ ]  arr = new int[][]{{1,2},{3,4},{5,6}};
int[ ][ ]  arr = {{1,2},{3,4},{5,6}};

Eg：

```
public class Demo3 {
    public static void main(String[] args) {
        int age[][] = new int[][]{{1,2},{3,4},{5,6,7}};
        System.out.println(age[0].length);//2
        System.out.println(age[2].length);//3
    }
}
```

## 12、操作数组的工具类-Arrays

static int binarySearch(type[] a, type key) 使用二分搜索法来搜索 key 元素在数组中的索引；若 a 数组不包括 key，返回负数。（该方法必须已按升序排列后调用）。
static int binarySearch(type[] a, int fromIndex, int toIndex, type key) 使用二分搜索法来搜索 key 元素在数组中从 fromIndex 到 toIndex 的索引；若 a 数组不包括 key，返回负数。（该方法必须已按升序排列后调用）。
static boolean[] copyOf(type[] original, int newLength) 复制指定的数组见下面备注
static byte[] copyOfRange(type[] original, int from, int to) 将数组的指定范围复制到一个新数组。

static boolean equals(type[] a, type[] a2) 如果两个数组长度相等和元素一一相等，则返回 true
static void fill(type[] a, type val) 将 a 数组所有元素都赋为 val。
static void fill(type[] a, int fromIndex, int toIndex, type val) 将 a 数组从 formIndex 到 tiondex 索引之间的元素都赋为 val。
static void sort(type[] a) //sort(int[] arr)对指定的数组按数字升序进行排序。
static void sort(type[] a, int fromIndex, int toIndex) 对指定数组的从 formIndex 到 tiondex 索引之间的元素按数字升序进行排序。
static String toString(type[] a) 返回指定数组内容的字符串表示形式。多个数组元素之间用英文逗号或空格隔开。

**我的总结：使用数组工具类可以节省时间，提高效率，注意常查阅 api；**

Eg:

```
package reviewDemo;

import java.util.Arrays;

//使用Arrays类
public class Demo4 {
    public static void main(String[] args) {
        int[] age = new int[] { 12, 26, 3, 60, 55, 6, 48, 4, 98 };
        System.out.println(Arrays.toString(age));//直接打印出数组的方
法

        int []age2 = {1,2,3,4,5,6,98,65,23};
        int i = Arrays.binarySearch(age2, 98);
        System.out.println(i);
    }
}
```

## 13、Java5 新特性对数组的支持：

增强 for 循环 → for-each

```
for (参数类型 参数名 : 数组名) {
        代码块
}
```

Eg:

```
package reviewDemo;

public class Demo6 {
    public static void main(String[] args) {
        int []age = {1,2,3,69,36,636,459};
        for (int i : age) {
            System.out.println(i);
        }
    }
}
```

这样就用 for-each 把数组遍历输出！

**我的总结：for-each 用于数组和 Iterable 对象！在以后的集合学习中还会经常用到 for-each 循环。**

# 三、黑马程序员—面向对象（1）

第三篇：

## 1、什么叫面向对象？

面向对象(Object-Oriented，简称 OO)就是一种常见的程序结构设计方法。
面向对象思想的基础是**将相关的数据和方法放在一起**，组合成一种新的复合数据类型，然后使用新创建的复合数据类型作为项目的基础。

面向对象是一个很抽象的概念，它相对面向过程而言。
过程与对象都是一种解决问题的思想。
**面向过程：强调的是功能行为,一种过程,先干啥,再干啥;**
**面向对象：将功能封装到对象里，强调的是具备某功能的对象;**
按照面向对象的思想，可以把任何的东西看做对象！

面向对象的三个特征：
**封装(Encapsulation);**
**继承(Inheritance);**
**多态(Polymorphism)。**

**我的总结：**
**面向过程：强调的是具体的功能实现；（执行者）**
**面向对象：强调的是具备功能的对象。（管理者）**

## 2、类

类(class)是 Java 语言的最小编程单位，也是设计和实现 Java 程序的基础，本部分将深入介绍类的相关知识。
**类的概念**
类是一组事物共有特征和功能的描述。类是对于一组事物的总体描述，是按照面向对象技术进行设计时最小的单位，也是组成项目的最基本的模块。类的概念是抽象的，类似于建筑设计中的图纸，是对于现实需要代表的具体内容的抽象。类只包含框架结构，而不包含具体的数据。所以类代表的是总体，而不代表某个特定的个体。

**我的总结：类是抽象的，对象是具体的，实实在在的！**

类的定义：
[修饰符] class　类名{
　　1~n 个构造方法；
　　0~n 个字段；
　　0~n 个方法
}

定义类，其实就是定义类里面的对象

对象包含：
**状态；（属性）**
**功能、行为；（方法）**
**通过类来描述对象；**
**状态--------成员变量；**
**功能、行为——方法；**

Eg：
```
class Person{
    //属性
    private String name;
    private int age;
    private int sal;
    //方法
    public void show(){
        System.out.println("个人情况： "+name+age+sal);
    }
}
```

## 3、构造方法

构造方法：用来构造类的实例（每一个类都默认有一个无参的构造方法，得使用 new 调用）
**字段：类或对象所包含的数据，对类状态的一种描述；**
**方法：类或对象的特征或行为**

**作用：**
**给类中的字段进行初始化，可以用来创建对象。**

特点：
方法名与类名相同
不用定义返回值类型
不需要写 return 语句

**我的总结：**
**注意：**
**默认构造方法的特点。**
**多个构造方法是以重载的形式存在的。**

构造方法的重载:**（需要哪个就去适配哪个，调用哪个）**
this([实参]);调用当前类的构造方法
注意: this([实参]);必须放在构造器的第一行;

对象的产生格式：

**类名称　对象名 = new　类名称();**

因为有(),所以是方法,实际上它就是构造方法，并且是非私有的构造方法。

如：CellPhone cp = new CellPhone();

Eg：

```
class Person{
    private String name;
    private int age;
    private int sal;

    public void show(){
        System.out.println("个人情况："+name+age+sal);
    }

    public Person(String name) {
        super();
        this.name = name;
    }

    public Person(String name, int age) {
        super();
        this.name = name;
        this.age = age;
    }

    public Person(String name, int age, int sal) {
        super();
        this.name = name;
        this.age = age;
        this.sal = sal;
    }
}
```

## 4、static 关键字

**特点：**

随着类的加载而加载

优先于对象存在

被所有对象所共享

可以直接被类名调用

**使用注意：**

静态方法只能访问静态成员

但是非静态成员可以访问静态成员;

静态方法中不可以使用 this，super 关键字

主方法（main）是静态的（可以利用类名去调用静态的 main 方法，很正常！但是会陷入死循环，导致内存溢出，jvm 自动停止！）

public static void main(String[] agrs){}

可修饰字段,方法。

**用 static 修饰的成员表示它属于这个类共有，而不是属于该类的单个实例。**

static 修饰的字段 == 类字段

static 修饰的方法 == 类方法

没使用 static 修饰的字段和方法，成员属于类的单个实例，

不属于类。

没有 static 修饰的字段 == 实例字段

没有 static 修饰的方法 == 实例方法

类和实例访问字段和方法的语法：

访问类成员： 类.字段 类.方法

访问实例成员： 实例.字段 实例.方法

**我的总结：**

**static 修饰的字段和方法，既可以通过类调用，也可以使用实例调用；**

**没 static 修饰的字段和方法，只能使用实例来调用（建议使用:类名来调用； 其实在底层，对象调用类成员,也会转换类名调用）**

**static 关键字不能与 this，super 同时连用！**

## 5、匿名对象

一个没有名字的对象， 创建了一个对象出来,没有赋给一个变量;

**特点：**

对方法或字段只进行一次调用时;

可作为实际参数进行传递;

只在堆里面开辟存储区域,

**只能使用一次， 使用完就被销毁了;**

**何时使用？只拿来用一次!!**

new Person();表示匿名对象,没有名字的对象

new Person().age = 17;//使用一次之后就被销毁了

## 6、this 关键字

特点：this 表示当前对象。

当前对象 ←→ 当前正在调用实例成员的对象
换言之：谁调用了方法，谁就是当前对象。

什么时候使用 this 关键字呢？
方法间的相互调用;
this.字段;
**构造器中相互调用，但是此时 this([参数])必须写在构造方法第一行。**
this 不能用在 static 修饰的方法里和 static 修饰的代码块里;

Eg：构造方法中的 this.name = name;

## 7、面向对象之封装

封装的两个含义：
1.把对象的状态和行为看成一个统一的整体，将二者存放在一个独立的模块中(类)；
2."信息隐藏", 把不需要让外界知道的信息隐藏起来,尽可能隐藏对象功能实现细节,字段;

**封装机制在程序中的体现是：把描述对象的状态用字段表示，描述对象的行为用方法表示，把字段和方法定义在一个类中，并保证外界不能任意更改其内部的字段值，也不允许任意调动其内部的功能方法。**

程序中的一种体现：通常将类中的成员变量私有化(private)，通过对外提供方法（setXxx,getXxx）,可对该变量(xxx)进行访问。
boolean 类型的变量没有 getXX,只有 isXX;

Eg：

```
class Person1{
    private String name;
    private int age;
    private int sal;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }
```

```
    public void setAge(int age) {
        this.age = age;
    }
}
```

## 8、访问修饰符

private 类访问权限:本类内部可以访问,不能继承到子类;
default 什么都不写,包访问权限:本类内部可以访问，同包其他类也可以访问,同包可继承;
protected 子类访问权限:本类内部可以访问，不同包的子类也可以访问，同包其他类也可以访问,能继承到子类;
public 公共访问权限:任何地方都可以访问,能继承到子类;

**我的总结：**

| 修饰符 | 类内部 | 同一个包 | 子类 | 任何地方 |
|---|---|---|---|---|
| private | √ | | | |
| default | √ | √ | | |
| protected | √ | √ | √ | |
| public | √ | √ | √ | √ |

## 9、类的设计分析

分析思路：
**根据要求写出类所包含的字段；**
**所有的字段都必须私有化；**
**封装之后的字段可通过 setter 和 getter 设值和取得；**
**按需求可添加若干构造方法；**
**根据需求可添加相应的方法；**
**类中的所有方法都不要直接处理(输出打印)，而是交给调用者去处理。**

## 10、面向对象之继承

首先有反映一般事物特性的类,然后在此基础上反映出特殊事物的类；
也就是说：**继承是一种从一般到特殊的关系；**

特点：
1、提高了代码的复用性。
2、让类与类之间产生关系，有了这个继承关系才有了多态的特性。
3、Java 语言中只支持单继承（有别于 C 语言）。
　　因为多继承容易带来安全隐患（父类多了， 功能相同的话，就会出现调用不确定性吗，覆写一个方法，到底覆写的谁的？）。

**ps：接口可以实现多继承**
4、Java 支持多层继承，object 是每个类的超类，实现树形结构。

**我的总结：**
**继承是多态的前提。**
**对类而言，只支持单继承。接口可以实现多继承**

格式：
[修饰符] class SubClass extends SuperClass

按照这种关系，我们把 SuperClass 类称为父类或基类，把 SubClass 称为子类或派生类或拓展类；

**我的总结：**
**java.lang.Object 是所有类的父类，**
**Object 要么是直接父类要么是间接父类。**

Eg：
学生属于人的一种特殊情况，此时我把人的共性写在 Person 类里面，为了让学生拥有这些共性（别的比如老师也可以有这些共性），然后我就让学生来拓展 Person 类。

**我的总结：**
**子类与父类的关系：**
**子类拓展父类（子类是父类的一种特殊情况）**
**主要是以父类为基础，然后添加属于自己的字段和方法。**

父类的私有成员子类不能继承到;父类的构造方法不能被继承；
**Java 只支持单继承,不支持多继承;//不然的话，比如 show 方法，继承了多个，不知道到底调用那一个。**
一个类有且只有一个直接父类;
一个类没显示的继承其他的一个类的时候,默认的直接父类就是 Object 类;
Student 的直接父类是 Person,Object 类也是 Student 类的父类,但是是间接父类;
一旦一个类显示的继承了其他的一个类的时候,此时默认的直接父类 Object 就会被取消;
Java 里一个类只能有一个直接父类;java.lang.Object 是所有类的父类，Object 要么是直接父类要么是间接父类。

**子类对象实例化过程**
在继承操作中，对于子类对象的实例化：
子类对象在实例化之前必须首先调用父类中的构造方法之后再调用自身的构造方法。

## 11、子类访问父类和方法覆写

子类不能直接访问父类的私有成员；
但是子类可以调用父类中的**非私有方法来间接访问父类的私有成员。**
Person 类中有私有字段 name,Student 继承 Person
new Sudent().name;　　×
new Student().getName();　　√

子类拓展父类（子类是父类的一种特殊情况）
　　主要是以父类为基础，然后添加属于自己的字段和方法。

方法覆写产生原因：
当父类中某个方法不适合于子类时,子类出现父类一模一样的方法.
判断必杀技:**子类方法前加上@Override 能编译通过,表明是方法的覆写。**
**调用被覆盖的父类方法：使用 super.方法名(实参);**
**方法覆写时应遵循的原则(一同两小一大)：**
(一同):方法签名必须相同;
(两小):
　　子类方法的返回值类型比父类方法的返回值类型更小或相等
　　子类方法声明抛出的异常应比父类方法申明抛出的异常更小或相等;
(一大):子类方法的访问权限应比父类方法更大或相等;

**子类需要覆写父类方法。**
　　**当父类的某个方法不适合于子类本身的特征行为时就当覆写父类中应当改变的方法。**

## 12、super 关键字和调用父类构造方法

表示父类对象的默认引用
如果子类要调用父类被覆盖的实例方法，可用 super 作为调用者调用父类被覆盖的实例方法。
使用 super 调用父类方法
使用 super 调用父类的构造方法

调用构造方法
本类中调用另一个重载构造方法用 this(参数列表)
子类构造方法调用父类构造方法用 super(参数列表)
子类调用父类的构造方法时：
super 必须放在第一句
Java 在执行子类的构造方法前会先调用父类无参的构造方法，其目的是为了对继承自父类的成员做初始化操作。
子类在创建对象的时候，默认调用父类的无参构造方法，要是子类构造方法中显示指定调用父类其他构造方法，就调用指定的父类构造方法，取消调用父类无参构造方法。

Eg：
```
package reviewDemo;
```

```
class A{
    String name;
    A(){
        System.out.println("父类默认隐式的构造方法！");
    }

    A(String name){
        System.out.println("父类显式的构造方法！");
    }
}

class B extends A{

    B(){
        super(null);
        System.out.println("子类默认隐式的构造方法！");
    }
}

public class Demo10 {
    public static void main(String[] args) {
        new B();
    }
}
```

结果：父类显式的构造方法！
子类默认隐式的构造方法！

## 13、面向对象之多态

多态:指同一个实体同时具有多种形式
好比,你去面馆吃面,说我要吃面,那么;老板给我牛肉面,鸡蛋面等都可以,
这就是说"面"有多种形态,也就是说实体有多种形态;
**编译时的类型由声明该变量时使用的类型决定,运行时的类型由实际赋给变量的对象决定。**
**如果编译时类型和运行时类型不同，就出现多态。**

Eg:
前提:Student extends Person:

Person p = new Person();
Student s = new Student();
Person p = new Student();//多态

引用关系:父类变量指向子类实例对象

实现多态的机制:
父类的引用变量可以指向子类的实例对象,而程序调用的方法在运行期才动态绑定，就是引用变量所指向的真正实例对象的方法,也就是内存里正在运行的那个对象的方法，而不是引用变量的类型中定义的方法。

多态的作用：
把不同的子类对象都当作父类来看，可以屏蔽不同子类对象之间的差异，写出通用的代码，做出通用的编程，以适应需求的不断变化。
只修改方法的实现，不必修改方法的声明
**继承是多态产生的前提条件;**

分类:
编译时多态：方法重载
运行时多态：方法覆写

Eg:

```
package test;

class Dog{
    void eat(){
        System.out.println("一般的狗吃一般的狗粮！");
    }
}

class HashDog extends Dog{
    void eat(){
        System.out.println("哈士奇吃哈士奇的狗粮！");
    }
}

class ZangAoDog extends Dog{
    void eat(){
        System.out.println("藏獒吃藏獒的狗粮！");
    }
}

//定义一个动物园喂的方法
class Zoo{

    void feed(Dog d){
        d.eat();
    }
```

```
}

public class Demo11 {
    public static void main(String[] args) {

        Dog hd = new HashDog();

        Dog zd = new ZangAoDog();

        Zoo z = new Zoo();
        z.feed(hd);
        z.feed(zd);
    }
}
```

输出：
哈士奇吃哈士奇的狗粮！
藏獒吃藏獒的狗粮！

## 14、引用变量类型转换

向上转型（子类→父类）：(自动完成)
    父类名称 父类对象 = 子类实例 ；

向下转型（父类→子类）：(强制完成)
    子类名称 子类对象 = （子类名称）父类实例 ；

对象名 instanceof 类

**判断指定的变量名此时引用的真正类型是不是当前给出的类或子类；**
**我的总结：对象的类型和类必须有继承关系**

Eg：
class A extends B{}

B b = new A();
If(b instanceof A){ ...
}

# 四、黑马程序员—面向对象（2）

第四篇：

## 1、基本数据类型的包装类

引言：Java 提倡的万物皆对象，但是数据类型的划分出现了基本数据类型和引用数据类型，那么我们怎么能把基本数据类型称为对象呢？

| 基本数据类型 | 包装类 |
| --- | --- |
| byte | Byte |
| short | Short |
| int | Integer |
| long | Long |
| char | Character |
| float | Float |
| double | Double |
| boolean | Boolean |

除了 Integer 和 Character 定义的名称和对应的基本类型差异大，其他六种都是将首字母大写就可以了。
Integer,Byte,Float,Double,Short,Long 都是 **Number 类的子类**。(Number 类后面讲)；
Character 和 Boolean 都是 **Object 直接子类**；
8 个类都是 final 修饰的(不可被继承)。

## 2、基本数据类型和包装类相互转换

把基本数据类型 → 包装类：
通过对应包装类的构造方法实现
除了 Character 外，其他包装类都可以传入一个字符串参数构建包装类对象。
包装类 → 基本数据类型
包装类的实例方法 xxxValue();　　// xxx 表示包装类对应的基本数据类型

Eg：

```
boolean bool = false;
Boolean b2 = new Boolean(bool);
Integer i = new Integer(3);
int i2 = i.intValue();
Boolean b1 = new Boolean("TRue");//true
boolean b2 = b1.booleanValue();
Float f = new Float("3.14");//3.14
```

Integer i2 = new Integer("123s");//NumberFormatException

备注：
**自动装箱&自动拆箱**

jdk1.5 开始出现的特性：
**自动装箱：可把一个基本类型变量直接赋给对应的包装类对象或则 Object 对象**
**自动拆箱：允许把 包装类对象直接赋给对应的基本数据类型**

Eg：
```
Integer i = 3;//装箱
int i2 = i;//拆箱
Object flag = new Boolean(false);
if(flag instanceof Boolean){
    Boolean b = (Boolean)flag;
    boolean b2 = b;
}
```

**我的总结：对于基本数据类型和包装类之间的装换，我们可以直接的用，相互转换，因为 java5 之后的自动拆箱、装箱功能！即便不知道这个，其实使用中也不影响！**

## 3、基本类型和 String 之间的转换

String → 基本类型，除了 Character 外所有的包装类提供 parseXxx(String s)静态方法，用于把一个特定的字符串转换成基本类型变量；
基本类型 → String，String 类有静态方法 valueOf()，用于将基本类型的变量转换成 String 类型。

```
String str = "17";
int i = Integer.parseInt(str);//String    -->  基本类型
String s1 = String.valueOf(i);//基本类型  --> String
```

**我的总结：这个从后续的学习来看，用处不大，记住有这样的方法就行，查 api！**

## 4、Object 类

所有类的公共父类，一旦一个类没有显示地继承一个类则其直接父类一定是 Object。
一切数据类型都可用 Object 接收
class OOXX  extends Object{}等价于 class ooXX {}

**常见方法**
public boolean equals(Object obj)：对象比较
public int hashCode()：取得该对象的 Hash 码
public String toString()：对象描述

Object 类的 toString()方法：“对象的描述”
建议所有类都**覆写此方法**
直接打印输出对象时，会调用该对象的 toString()方法。//可以不写出来
打印对象的时候,实际调用的对象实际指向的类的自我描述；
全限定类名+@+十六进制的 hashCode 值，等价于
全限定类名+@+IntegertoHexString(该对象.hashCode)

equals 也是判断是否指向同一个对象
没有实际意义，有必要可以重写
public boolean equals(Object obj) {}
String 覆写了 Object 的 equals 方法：只比较字符的序列是否相同
==用于判断两个变量是否相等
基本类型：
引用类型：必须指向同一个对象，才 true
只能比较有父子或平级关系的两个对象
new String("1") == new String("1"); ？

## 5、代码块

代码块指的是使用"{}"括起来的一段代码，根据代码块存在的位置可以分为 4 种:
**普通代码块;**
**构造代码块;**
**静态代码块;**
**同步代码块(线程同步的时候讲解)。**
代码块里变量的作用域：
只在自己所在区域(前后的{})内有效；

**普通代码块：**
**普通代码块就是直接定义在方法或语句中定义的代码块：**

```
public void show(){
    普通代码块
}
```

**构造代码块：**
**直接写在类中的代码块：**
**优先于构造方法执行，每次实例化对象之前都会执行构造代码块。**

Eg:

```
public class Demo {
    {
        System.out.println("我是构造代码块");
    }
    public Demo(){
        System.out.println("我是构造方法");
    }
    public static void main(String[] args) {
        Demo d1  = new Demo();
        Demo d2  = new Demo();
    }
}
```

**静态代码块**

**使用 static 修饰的构造代码块:**

**优先于主方法执行，优先于构造代码块执行，不管有创建多少对象，静态代码块只执行一次，可用于给静态变量赋值；**

Eg:

```
package reviewDemo;
/**
 * 测试各代码块的优先级
 *  优先级顺序：静态代码块   >  构造代码块  >  普通代码块
 *  备注：无论创建几个对象，静态代码块只执行一次！
 */

public class Demo13 {
    Demo13(){
        System.out.println("我是构造方法！");
    }
    {
        System.out.println("我是构造代码块！");//实例化对象的时候才会去调用!
    }
    static{
        System.out.println("我是静态代码块！");
    }

    public static void main(String[] args) {
        new Demo13();
        new Demo13();//再次创建对象，证明无论创建几次对象，静态代码块都只执行一次
```

```
        System.out.println("我是普通代码块！");
    }
}
```

输出：

我是静态代码块！
我是构造代码块！
我是构造方法！
我是构造代码块！
我是构造方法！
我是普通代码块！

**我的总结：这个例子非常好！**

## 6、构造方法的私有化

有的时候我们为了避免外界创建某类的实例，就将某类的构造方法私有化，**即将它的构造方法用 private 修饰：**

外界如何用到？
提供 get 方法！不提供的话外界就没法创建对象！（对反射无效）

Eg：

```
package reviewDemo;

class Stu{
    //将构造方法私有化
    private Stu(){

    }
}

public class Demo15 {
    public static void main(String[] args) {
        Stu s = new Stu();
    }
}
```

## 7、Singleton 模式(单例模式) 饿汉式和懒汉式

目的：**整个应用中有且只有一个实例，所有指向该类型实例的引用都指向这个实例。**
好比一个国家就只有一个皇帝(XXX)，此时每个人叫的“皇帝”都是指叫的 XXX 本人;

常见单例模式类型：

饿汉式单例：直接将对象定义出来
懒汉式单例：只给出变量，并不将其初始化；

我的总结：
饿汉式，static 修饰，随着类的加载而加载，会损耗性能，但是方法相对简单
懒汉式　第一次用的时候相对较慢，因为需要加载！线程，不安全！

```
package reviewDemo;
//单例模式

//饿汉式,直接把对象构造出来
class SingleDemo{
    private static SingleDemo s1 = new SingleDemo();
    private SingleDemo(){
        //提供私有化的构造方法，那么外界就不能构造对象了！
    }

    public static SingleDemo getS1() {
        return s1;
    }
}

//懒汉式，先定义，但是不创建对象
class SingleDemo2{
    private static SingleDemo2 s3 ;

    private SingleDemo2(){
        //提供私有化的构造方法，那么外界就不能构造对象了！
    }

    public static SingleDemo2 getS3() {//这是一个方法，返回值为创建的对
象！
        if(s3 == null){
            s3 = new SingleDemo2();
        }//和饿汉式的区别，此时才来创建对象！
        return s3;
    }
}

public class Demo14 {
    public static void main(String[] args) {
        SingleDemo s1 = SingleDemo.getS1();
        SingleDemo s2 = SingleDemo.getS1();
```

```
        SingleDemo2 s3 = SingleDemo2.getS3();
        SingleDemo2 s4 = SingleDemo2.getS3();

        System.out.println(s1 == s2);
        System.out.println(s3 == s4);

    }
}
```

输出：true true

**备注：枚举更加安全些**

```
package reviewDemo;

enum Stu{
    jake;
    //将构造方法私有化起来，反射也不能创建对象，安全
    private Stu(){

    }
}

public class Demo15 {
    public static void main(String[] args) {
    }
}
```

## 8、final 关键字

① final 可以修饰类,方法,变量。
② final 修饰类不可以被继承，但是可以继承其他类。
③ final 修饰的方法不可以被覆写,但可以覆写父类方法。
④ final 修饰的变量称为常量，这些变量只能赋值一次。
⑤ 内部类在局部时，只可以访问被 final 修饰的局部变量。
⑥ final 修饰的引用类型变量,表示该变量的引用不能变,而不是该变量的值不能变;

Eg:

```
package reviewDemo;

final class Name{
}
```

```
class NewName extends Name{//ERROR，报错，因为Name有final修饰
}

public class Demo15 {
    public static void main(String[] args) {
    }
}
```

## 9、抽象类

当编写一个类时，我们往往会为该类定义一些方法，这些方法是用来描述该类的行为方式，那么这些方法都有具体的方法体。
**但是有的时候，某个父类只是知道子类应该包含怎么样的方法，但是无法准确知道子类如何实现这些方法。**

抽象方法的定义：通过 abstract 关键字来修饰的类称为抽象类；

**我的总结：**
**抽象类用 private 修饰，里面可以有用 private 修饰的方法（没有方法体），强制子类进行覆写；**
**可以理解为：具有某些公共方法的一个总结类。**

**可以定义被 abstract 修饰的抽象方法**
**抽象方法只有返回类型和方法签名，没有方法体。**

**备注：**
**抽象类可以含有普通方法**
**抽象类不能创建实例对象（不能 new）**
**需要子类覆盖掉所有的抽象方法后才可以创建子类对象，否则子类也必须作为抽象类**
**列举常见的几个抽象类：**
**流的四个基本父类**
**InputStream，OutputStream，Reader，Writer**

**我的总结：**
**抽象类是类的一种特殊情况:据有类的一切特点,但是不能实例化;一般的都得带有抽象方法。**
**抽象类不可以实例化，有时看到的近似实例化是多态机制的体现，并不是真正的实例化。**
Eg：

```
Socket s = new Socket();
OutputStream os = s.getOutputStream();
```

左边是 OutputStream 类型变量的声明，右边是获取抽象类 OutputStream 的一个实例对象！

```
package testDemo2;

abstract class Person{
}

class Student extends Person{
}

public class Demo2 {
    public static void main(String[] args) {
        Person p = new Student();//体现的是多态，父类声明实例化子类对象。而不是抽象类实例化
    }
}
```

**abstract 方法**

分析事物时，发现了共性内容，就出现向上抽取。会有这样一种特殊情况，就是功能声明相同，但功能主体不同。

那么这时也可以抽取，但只抽取方法声明，不抽取方法主体。那么此方法就是一个抽象方法。

abstract [非 private 访问修饰符] 返回值类型 方法名称(参数列表);
抽象方法要存放在抽象类中。
抽象方法也可以存在于接口中

Eg:

```
package reviewDemo;

abstract class Person3{
    abstract void show();
    abstract void inof();
    void turn(){
    }
}

class NewP extends Person3{
    @Override
    void show() {
```

```
    }

    @Override
    void inof() {
    }
    //不覆写的话会报错
}

public class Demo15 {
    public static void main(String[] args) {
        //new Person3();报错！因为抽象类不可以实例化
    }
}
```

## 10、抽象类的体现-模板模式

抽象类是多个具体子类抽象出来的父类,具有高层次的抽象性;以该抽象类作为子类的模板可以避免子类设计的随意性;

抽象类的体现主要就是模板模式设计,抽象类作为多个子类的通用模板,子类在抽象类的基础上进行拓展,但是子类在总体上大致保留抽象类的行为方式;
编写一个抽象父类,该父类提供了多个子类的通用方法,并把一个或多个抽象方法留给子类去实现,这就是模板设计模式;

**模板模式应用的简单规则:**

**1.抽象父类可以只定义需要使用的某些方法,其余留给子类去实现;**

**2.父类提供的方法只是定义了一个通用算法,其实现必须依赖子类的辅助;**

**我的总结:**

**如果父类的方法不想被子类覆写，那么可以在前面加上 final 关键字修饰。**

Eg:

```
package reviewDemo;
//模板模式

//抽象类中包含很多的抽象方法，子类必须去覆写！
abstract class Method{
    abstract double mul();//返回值类型如果是void的话，下面报错，因为没有返回值，无法引用！
    abstract double divid();
    void show(){
        System.out.println("面积是: "+mul());//周长
        System.out.println("面积是: "+divid());//面积
```

```
    }
}

class Square extends Method{
    double d;

    public Square(double d) {
        super();
        this.d = d;
    }

    @Override
    double mul() {
        return d * d;
    }

    @Override
    double divid() {
        return 4 * d;
    }
}

class Cirle extends Method{
    double r;

    public Cirle(double r) {
        super();
        this.r = r;
    }

    @Override
    double mul() {
        return 2 * 3.14 * r;
    }

    @Override
    double divid() {
        return 3.14 * r * r;
    }
}

public class Demo16 {
    public static void main(String[] args) {
        Square s = new Square(5);
```

```
        s.show();
        Cirle c = new Cirle(4);
        c.show();
    }
}
```

## 11、接口(interface)

引入：抽象类是从多个类中抽象出来的模板，若要将这种抽象进行得更彻底，就得用到一种特殊的“抽象类”→ 接口;

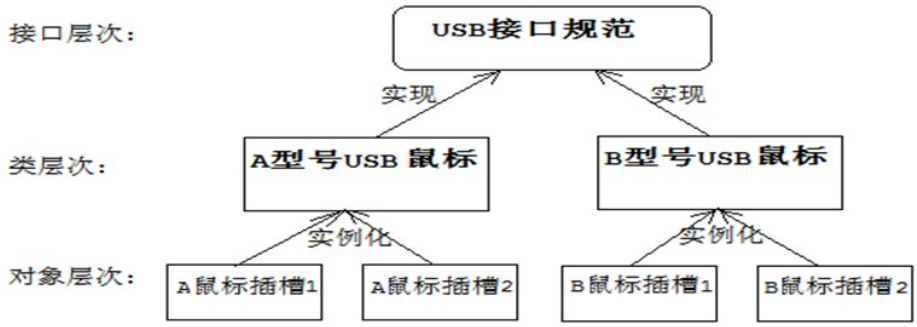
例子：
生活中听说过的 USB 接口其实并不是我们所看到的那些插槽,而是那些插槽所遵循的一种规范;而我们看到的那些插槽是根据 USB 规范设计出来的实例而已,也就说插槽是 USB 的实例;对应不同型号的 USB 设备而言,他们各自的 USB 插槽都需要遵循一个规范,遵守这个规范就可以保证插入插槽的设备能与主板正常通信;

对于同一种型号的主板上的多个 USB 插槽,他们有相同的数据交换方式,相同的实现细节,可认为他们都是同一个类的不同实例

**我的总结：**
**接口只定义了类应当遵循的规范,却不关心这些类的内部数据和其方法内的实现细节.**
**接口只规定了这些类里必须提供的方法;从而分离了规范和实现.增强了系统的可拓展性和维护性;**

使用接口的好处,拓展性,维护性更好,所以我们在开发中会经常用到接口.(相当于定义了一种标准)

**接口,类,对象示意图**




**interface 定义**

接口定义一种规范,规定一个类必须做什么,但它不管如何具体去做;

**[修饰符] interface 接口名 extends 父接口 1,父接口 2....**

**没有构造方法,不能实例化;**

**接口只能继承接口,不能继承类**

**接口里没有普通方法,方法全是抽象的;**

**接口里的方法默认修饰符是 public abstract;**

**接口里的字段全是全局常量,默认修饰符是 public static final;**

**接口里的成员包括(主要是前两个):**

全局常量

公共的抽象方法

内部类(包括内部类,内部接口,内部枚举类);

**我的总结:**

**接口没有构造方法,不能实例化!**

**接口里的方法全部是抽象的,没有普通方法,有默认的修饰符 public abstract,必须全部覆写!**

## 12、接口的使用

格式:public class SubImpl extends Super implements IA,IB

**接口可以多继承,但是只能继承接口,不能继承类。**

实现接口(支持多实现)

[修饰符] class 类名 implements 接口 1,接口 2...

接口的实现必须在 extends 之后;

实现接口的方法必须是 public 类型

接口不能创建实例,但是可以声明引用类型的变量。

此时,引用类型的变量必须指向到其实现类对象。

IStudent s = new String();//

IStudent s = new StudentImpl();//

**接口与类之间的关系:**

**实现关系或者说是继承关系.**

可以说类实现了接口的方法,也可以说类继承了接口的方法,不同情况下不同的理解!

## 13、面向接口编程之制定标准和简单工厂模式

制定一个标准,让别人去实现或者说满足它!

Eg:

interface USB{//定义 USB 标准
    void useUSB();//USB 有使用 USB 的行为
}

**简单工厂模式**

**构建一个工厂出来，在里面进行生产，用的时候直接拿**

**我的总结：**

**好处:屏蔽不同子类实现的差异，提高代码的可拓展性和可维护性;**

```
package reviewDemo;
//简单工厂模式

interface Phone{//制定标准，都要实现send()方法
    public void send();
}

class Iphone implements Phone{
    @Override
    public void send() {
        System.out.println("Iphone手机在发短信");
    }
}

class AndroidPhone implements Phone{
    @Override
    public void send() {
        System.out.println("AndroidPhone手机在发短信");
    }
}

class MyPhone implements Phone{
    @Override
    public void send() {
        System.out.println("MyPhone手机在发短信");
    }
}

class Factory{
    public static void show(String type){//传入参数，根据不同的类型个性化定制
        if(type.equals("")){//为空的情况，不用往下执行
            System.out.println("对不起，类型为空！,请重新输入！");
```

```
            return;
        }
        Phone p = null;
        if("Iphone".equals(type)){//判断类型
            p = new Iphone();
        }else if("AndroidPhone".equals(type)){
            p = new AndroidPhone();
        }else{
            p = new MyPhone();
        }
        p.send();
    }
}

public class FactoryDemo17 {
    public static void main(String[] args) {

        new Factory().show("Iphone");//调用方法
        new Factory().show("AndroidPhone");
        new Factory().show("MyPhone");
        new Factory().show("YourPhone");
        new Factory().show("");
    }
}
```

**输出：**

```
Iphone手机在发短信
AndroidPhone手机在发短信
MyPhone手机在发短信
MyPhone手机在发短信
对不起，类型为空！
```

## 14、面向接口编程之适配器模式

使用一个现成的类，但是它的接口不完全符合你的需求,我只想要它其中的一个方法，不想覆写其他的方法。

比如,窗体有变大,变小,关闭的行为,但是我现在只需要关闭行为;

```
package reviewDemo;
```

```
//适配器模式：只想用其中的某一个方法，用适配器作为中间的过渡

interface Windows{
    void max();
    void min();
    void close();
}

//适配器模式，实现接口所有的方法，但是不写方法体！
class AdapterWindows implements Windows{

    @Override
    public void max() {
    }

    @Override
    public void min() {
    }

    @Override
    public void close() {
    }

}

class MyWindows extends AdapterWindows{
    //覆写父类的方法
    public void close(){
        System.out.println("这个实现的是关闭功能！");
    }
}

public class Demo17 {
    public static void main(String[] args) {
        new MyWindows().close();
    }
}
```

## 15、接口和抽象类的比较

**相同点:**

**都位于继承的顶端,用于被其他实现或继承;**

都不能实例化;
都包含抽象方法,其子类都必须覆写这些抽象方法;

区别:
抽象类为部分方法提供实现,避免子类重复实现这些方法,提供代码重用性;接口只能包含抽象方法;
一个类只能继承一个直接父类(可能是抽象类),却可以实现多个接口;(接口弥补了 Java 的单继承)

二者的选用:
优先选用接口,尽量少用抽象类;
需要定义子类的行为,又要为子类提供共性功能时才选用抽象类;

我的总结:
接口不能有构造函数，抽象类是可以有构造函数的，
abstract 可以定义构造函数（包括带函数的构造函数），因为要保证其子类在创建的时候能够进行正确的初始化，但是 Abstract 类不能被实例化。

知识点：如果不可以或者没有创建对象，那么我们必须加上 static 修饰，不能用对象调用，就只好用类去调用。

| No. | 比较 | 抽象类 | 接口 |
|---|---|---|---|
| 1 | 关键字 | 使用 abstract class 声明 | 使用 interface 声明 |
| 2 | 定义 | 包含一个抽象方法的类 | 抽象方法和全局常量的集合 |
| 3 | 组成 | 属性、方法、构造、常量、抽象方法 | 全局常量、抽象方法 |
| 4 | 权限 | 抽象方法的权限可以任意 | 只能是 public 权限 |
| 5 | 使用 | 通过 extends 关键字继承抽象类 | 通过 implements 关键字实现接口 |
| 6 | 局限 | 抽象类存在单继承局限 | 没有此局限，一个子类可以实现多个接口 |
| 7 | 顺序 | 一个子类只能先继承抽象类再实现多个接口 | |
| 8 | 设计模式 | 模板设计 | 工厂设计、代理设计 |
| | | 两者联合可以完成一个适配器设计 | |
| 9 | 实际作用 | 只能做一个模板使用 | 作为标准、表示能力 |
| 10 | 使用 | 两者没有什么本质的区别，但是从实际上来看，如果一个程序中抽象类和接口都可以使用的话，则一定要优先考虑接口，因为接口可以避免单继承所带来的局限。 | |
| 11 | 实例化 | 都是依靠对象多态性，通过子类进行对象实例化的 | |

## 16、匿名内部类

适合只使用一次的类
不能是抽象类，因为系统在创建匿名内部类的时候，会立即创建匿名内部类的对象。
匿名内部类不能定义构造器，因为匿名内部类没有类名。

格式：
new 父类构造器([实参列表]) 或 接口()
{
    //匿名内部类的类体部分
}

## 17、枚举类

使用 enum 声明,默认直接继承了 java.lang.Enum 类，而不是 Object 类；
枚举类的对象是固定的,实例个数有限,不可以再 new( ),枚举对象后可以跟()。
枚举元素必须位于枚举类体中的最开始部分,枚举元素后要有分号与其他成员分隔。
枚举类的构造方法的权限修饰符默认是 private；
一旦枚举对象后面加上{},那么该对象实际是枚举匿名内部类对象;
所有枚举类都提供一个静态的 values()方法(返回该枚举类所有对象组成的数组),便于遍历所有枚举对象；
所有枚举类都提供一个静态的 valueOf(String name)方法, 返回枚举类中对象名等于 name 的对象。

**Eg：** public enum Color{
    RED(), GREEN(){}, BLUE{};
}

```
package reviewDemo;
//枚举

enum Color{
    Green,Blue,Yellow;

    @Override
    public String toString() {
        String ret = super.toString();
        switch (this) {
        case Green:
            ret = "绿色";
            break;

        case Blue:
            ret = "蓝色";
            break;

        case Yellow:
```

```
            ret = "黄色";
            break;

        default:
            break;
        }

        return ret;
    }

}

class Personp{
    Color c = Color.Blue;
    void show(){
        System.out.println(c);
    }
}

public class Demo18 {
    public static void main(String[] args) {
        Color []color = Color.values();
        for (Color c : color) {
            System.out.println(c);
        }
        new Personp().show();
    }
}
```

输出：

绿色
蓝色
黄色
蓝色

枚举类覆写接口抽象方法的两种方式：

在枚举类中实现接口的抽象方法；
在枚举匿名内部类中实现接口的抽象方法；

```
interface I{
    void show();
}

enum Color implements I{
```

```
    RED(){
    public void show(){
        }
    }, GREEN{
    public void show(){
        }
    }, BLUE{
    public void show(){
        }
    };
}

enum Color implements I{
    RED(), GREEN, BLUE;
    public void show() {
    }
}
```

**我的总结：**

**枚举不可以 new();即便是反射也不可以！**

**备注：一个类如果没有构造方法，那么一定有相对应的某个方法可以获取对象！**

# 五、黑马程序员—异常处理和常用类

第五篇

## 1、异常

异常：就是不正常，是指程序在运行时出现的不正常情况。其实就是程序中出现的问题。这个问题按照面向对象思想进行描述，并封装成了对象。因为问题的产生有产生的原因、有问题的名称、有问题的描述等多个属性信息存在。当出现多属性信息最方便的方式就是将这些信息进行封装。异常就是 java 按照面向对象的思想将问题进行对象封装。这样就方便于操作问题以及处理问题。

出现的问题有很多种，比如角标越界，空指针等都是。就对这些问题进行分类。而且这些问题都有共性内容比如：每一个问题都有名称，同时还有问题描述的信息，问题出现的位置，所以可以不断的向上抽取。形成了异常体系。

**异常的体系　Throwable**

Error

通常指 JVM 出现重大问题如：运行的类不存在或者内存溢出等。

不需要编写针对代码对其处理，程序无法处理。

Exception

在运行时运行出现的一些情况，可以通过 try,catch,finally 处理

**异常处理两种方式：**

**1、捕获异常：try catch　　直接处理可能出现的异常！**

**2、声明异常：throws　　声明告诉调用者可能的异常，暴露问题，调用者自己处理！**

**我的总结：**

**Exception 和 Error 的子类名大都是以父类名作为后缀。**

**Java 异常其实是对不正常情况的一种描述,并将其封装成对象;**

**Java 在 设计异常体系时，将容易出现的异常情况都封装成了对象。**

## 2、异常处理格式

异常处理的 5 个关键字

try ,catch,

finally

throw, throws

**我的总结：**

**捕获异常：先捕获小异常再捕获大异常。**

**程序是调出来的，不是写出来的；多测试是程序员的必修课。**

**异常处理后，程序不会因为出现异常而退出！**

**异常处理格式**

```
try{
        //可能出异常的代码
} catch(异常类　对象){
        //处理该异常类型的语句
}
[finally] {
        //一定会执行的代码
        //catch 块使用 System.exit(1);除外
}
```

备注：当 try 语句块出现异常，程序会自动跳到 catch 语句块去找匹配的异常类型，并执行异常处理语句，finally 语句块是异常的统一出口。

## 3、多异常处理

声明异常时尽可能声明具体异常类型,方便更好的处理;
方法声明几个异常就对应有几个 catch 块;
若多个 catch 块中的异常出现继承关系,父类异常 catch 块放在最后;
在 catch 语句块使用 Exception 类作为异常类型时：
所有子类实例都可以使用父类接收(向上转型)，即所有的异常对象都可以使用 Exception 接收;
**注：在 java 处理多异常时捕获小范围的异常必须放在大范围异常之前。**

java7 - 同时捕获多个异常类型
Java7 之前：

```
try {
      int a = Integer.parseInt("1");
      int b = Integer.parseInt("0");
      int c = a / b;
      System.out.println(c);
} catch (NumberFormatException e)
{
    e.printStackTrace();
} catch(ArithmeticException e)
{
      e.printStackTrace();
}
```

Java7：将多个异常写到了同一个 catch 代码块

```
try {
Integer a = Integer.parseInt("1");
Integer b = Integer.parseInt("0");

Integer c = a / b;

System.out.println(c);

} catch (NumberFormatException
      | ArithmeticException    e ) {
        e.printStackTrace();
}
```

## 4、异常的分类

异常分类:

编译时被检查异常;　　---> Checked 异常

在程序中必须使用 try...catch 处理;

编译时不被检测的异常;　　---> Runtime 异常

可以不使用 try...catch 处理，但一旦出现异常就将由 JVM 处理。

**异常的分类之 Runtime 异常**

RuntimeException(运行时异常)是指因设计或实现方式不当而导致的问题.
说白了,就是程序员造成的,程序员小心谨慎是完全可以避免的异常.比如,事先判断对象是否为 null 就可以避免 NullPointerException 异常,事先检查除数不为 0 就可以避免 ArithmeticException 异常;

特点:

这种异常Java编译器不会检查它,也就说程序中出现这类异常的时候,即使不处理也没有问题,但是一旦出现异常,程序将异常终止,若采用异常处理,则会被相应的程序执行处理.

**异常的分类之 Checked 异常**

除了 RuntimeException 以及子类,其他的 Exception 及其子类都是受检查异常,我们也可以称为非 RuntimeException 异常.
特点:

Java 编译器会检查它,也就说程序中一旦出现这类异常,要么是没有 try-catch 语句捕获,或 throws 语句没有声明抛出它,编译就不会通过,也就说这种异常,程序要求必须处理.

## 5、声明异常（throws）

在可能出现异常的方法上声明抛出可能出现异常的类型:

声明的时候尽可能声明具体的异常,方便更好的处理.

当前方法不知道如何处理这种异常，可将该异常交给上一级调用者来处理(非 RuntimeException 类型的异常)。

方法一旦使用 throws 声明抛出方法内可能出现的异常类型， 该方法就可以不再过问该异常了;

一个方法调用另一个使用 throws 声明抛出的方法，自己要么 try...catch , 要么也 throws;

**格式:**

```
public  返回值类型　方法名(参数列表...)
                throws  异常类 A,异常类 B... {
```

```
}
```

## 6、throw

自行抛出一个异常对象，抛出异常类的对象；
若 throw 抛出的是 Runtime 异常：
程序可以显示使用 try...catch 来捕获并处理，也可以不管，直接交给方法调用者处理；
若 throw 抛出 Checked 异常：
要么放在 try 里自己处理，要么放在一个 throws 声明的方法里面，交给调用者处理。

**Eg：**

```
public static void main(String[] args) {
    try {
        fn1(1);
    } catch (Exception e) { e.printStackTrace(); }
    fn2(2);
}
public static void fn1(int a) throws Exception{
    if(a >0) { throw new Exception("fn1 -- a 值不合法"); }
}
public static void fn2(int a) {
    if(a >0) { throw new RuntimeException("a 值不合法"); }
}
```

**throws & throw**

throws 用于在方法上声明该方法不需要处理的异常类型。
throw 用于抛出具体异常类的对象。

**throws 与 throw 的区别：**

thorws 用在方法上，后面跟异常类名,可以是多个异常类。
throw 用在方法内，后面跟异常对象,只能是一个。

**7、finally**

异常的统一出口：
不管 try 块程序是否异常，也不管哪个 catch 执行，finally 块总会执行。
try 语句块或会执行的 catch 语句块使用了 JVM 系统退出语句例外;//System.exit(1);
try 块必须和 catch 块或和 finally 同在，不能单独存在，二者必须出现一个。
不要在 finally 中使用 return 或 throw 语句，否则将会导致 try、catch 中的 return 或 throw 失效。

**我的总结：finally 代码块只在一种情况下不执行：System.exit(0)；**

**Eg：**

```
package reviewDemo;

public class Demo19 {
    public static void main(String[] args) {
        try{
            System.out.println(17/0);
        }catch(Exception e){
            //e.printStackTrace();
            System.out.println("程序错误，请修正！");
        }finally{
            System.out.println("这是finally代码块！");
        }
    }
}
```

**输出：**

程序错误，请修正！
这是finally代码块！

## 8、throw 和 catch 同时使用

当异常出现在当前方法中,程序只对异常进行部分处理,还有一些处理需要在方法的调用者中才能处理完成,此时还应该再次抛出异常,这样就可以让方法的调用者也能捕获到异常;

**Eg：**

```
public static void buy(String price) throws Exception {
    try {
        if(price != null)
            Double.parseDouble(price);
    } catch (Exception e) {
        e.printStackTrace();
        throw new Exception("价格不能只能是数字组成");
    }
}
public static void main(String[] args)  {
    try {
        buy(null);
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
}
```

## 9、常用类

**String**

String 表示字符串,所谓字符串,就是一连串的字符;

String 是不可变类，一旦 String 对象被创建，包含在对象中的字符序列(内容)是不可变的，直到对象被销毁；

常量池:JVM 中一块独立的区域存放字符串常量和基本类型常量（public static final）。

String 使用 private final char value[]来实现字符串的存储,也就是说 String 对象创建之后,就不能再修改此对象中存储的字符串内容,就是因为如此,才说 String 类型是不可变的.

String 对象比较:

单独使用""引号创建的字符串都是常量,编译期就已经确定存储到常量池中；

使用 new String("")创建的对象会存储到堆内存中,是运行期新创建的；

使用只包含常量的字符串连接符如"aa" + "bb"创建的也是常量,编译期就能确定,已经确定存储到常量池中；

使用包含变量的字符串连接符如"aa" + s1 创建的对象是运行期才创建的,存储在堆中；

Eg：

```
package string;

public class StringDemo {
	public static void main(String[] args) {
		/*
		 *   ""  其实表示的是   String 的匿名对象
		 */
		String s = "";
		/**
		 * String()       初始化一个新创建的  String  对象，使其表示一个空字符序列。
		 *   "",不是表示  null
		 *   String(String original)
		 *   String s = new String("asd");// s  的值就是  asd
		 */
		s = new String();//""
		String s1    = new String("11");
		String s2    = new String("11");
		System.out.println(s1 == s2);//false
		String s3 = "22";
		String s4 = "22";
```

```
        System.out.println(s3 == s4);//true
        System.out.println(s3 == s2);//false
        String  s5 = "2" + "2";//这句话其实在编译器编译后的 class 文件中 已经变成  "22"
        //s5 创建了几个对象?//回答：最多一个，如果常量池里面没有就是创建一个，如果常量池里本来就有就是创建零个！下面的 str 一样的道理。
        String str = "A" +"B"+ "C" +"D" +"E";//String str = "ABCDE";
        System.out.println(s3 == s5);//true
        /**
         * String 特点:    不可变的,  一个 String 对象的内容不能变,一旦内容变量该对象就变成一个新的 String 对象了
         *  String  str = "AAA";
         *  str = str+"12";
         */
        String str1 = "ABCDE";
        System.out.println(str == str1);//true
        String str2 = "ABC" + "DE";
        System.out.println(str == str2);//true
        String  ss = "AB";
        String str3 = ss + "CDE";//这里的 str3 在运行期才能确定内容值
        System.out.println(str3 == str2);//false
        String str4 = getStr() +"CDE";//在运行期才能确定 str4 的值
        System.out.println(str == str4);//false
        System.out.println(str3 == str4);//false
        /*
         *  String s = "cd";    //一个 String 对象
            s = s + "itcast";    //itcast 也是 String 对象，最后的 s 的值 cditcast,也是一个 String 对象
            System.out.print(s);
         * */
    }
    public static String getStr(){
        return "AB";
    }
}
```

Eg:

```
package reviewDemo;

public class Demo20 {
    public static void main(String[] args) {
        //s1,s2只新建了一个对象
        String s1 = "你好";
        String s2 = "你好";
```

```
        //s3,s4新建了两个对象
        String s3 = "你好";
        String s4 = "你好";
    }
}
```

## 10、String 方法

String():初始化一个新的 String 对象，使其表示一个空字符序列，并不是返回空(不等于 null)。
String(StringBuffer buffer):根据 StringBuffer 对象来创建 String 对象;
String(StringBuilder builder):同上
char charAt(int index):取字符串中指定位置的字符，index 从 0 开始计算。
String concat(String str):连接字符串，等同于 “+”;
boolean contentEquals(StringBuffer buffer):若二者包含的字符序列相同时就返回 true;
boolean equals(Object obj):将该字符串与指定对象比较，若二者包含的字符序列相等返回 true;
boolean equalsIgnoreCase(String anotherString) 将此 String 与另一个 String 比较，不考虑大小写;
byte[] getBytes():将该字符串转换成 byte 数组;

int indexOf(String str):找出 str 字符串在该字符串中第一次出现的位置;
int indexOf(String str, int fromIndex) 返回指定子字符串在此字符串中第一次出现处的索引，从指定的索引开始;
int lastIndexOf(String str) 返回指定子字符串在此字符串中最后一次出现处的索引;
int length():返回当前字符串长度;
String replace(char oldChar, char newChar) :返回一个新的字符串，它是通过用 newChar 替换此字符串中出现的所有 oldChar 得到的。
String replaceAll(String regex, String replacement) 使用给定的 字符串 replacement 替换此字符串所有的 regex 字符串;
boolean startsWith(String prefix) 测试此字符串是否以指定的前缀开始.

String[] split(String regex)： 把字符串按指定的字符串分隔开。
String substring(int beginIndex) 返回一个新的字符串，从 beginIndex 开始截取，它是此字符串的一个子字符串;
String substring(int beginIndex, int endIndex) 返回一个新字符串，它是此字符串的一个子字符串;[begin,end)
char[] toCharArray() 将此字符串转换为一个新的字符数组。
String toLowerCase() 将此 String 中的所有字符都转换为小写;
String toUpperCase()转成大写;
static String valueOf(基本类型 obj):把基本类型值转成字符串;
String trim() ： 忽略字符串前导空白和尾部空白。

**String 小练习**

**判断字符串是否由数字组成：**

```
package reviewDemo;

public class Demo20 {
    public static void main(String[] args) {

        String s1 = "123456789";
        String s2 = "12345  6789";
        System.out.print(isnum(s1));
        System.out.print(isnum(s2));
    }

    public static boolean isnum(String s){
        char []ch = s.toCharArray();
        for (char c : ch) {
            if(!(c > '0' && c <= '9')){
                return false;
            }
        }
        return true;
    }
}
```

输出：

true
false

**判断字符串是否为空：**

分析：

字符串的空有两种情况：1、null；2、"";

```
package reviewDemo;

public class Demo21 {
    public static void main(String[] args) {

        String s1 = "";
        System.out.println(isimpty(s1));
```

```
    }

    public static String isimpty(String s){

        if(s != null & !"".equals(s)){
            return "不为空";
        }
        return "为空！";
    }
}
```

## 11、StringBuffer 与 StringBuilder

String 是不可变类，一旦 String 对象被创建，包含在对象中的字符序列是不可变的，直到对象被销毁；

StringBuffer 与 StringBuilder 对象则是可变的！

举例说明这两个的好处：（不用每次新建对象，效率高！）

```
package reviewDemo;

public class Demo22 {
    public static void main(String[] args) {

        String s = "";
        long begintime = System.currentTimeMillis();
        for(int i = 1;i <= 100000;i++){
            s += i;
        }


        long endtime = System.currentTimeMillis();
        long time = endtime - begintime;
        System.out.println("运行时间为："+time);

        StringBuffer s1 = new StringBuffer();
        s = "";
        begintime = System.currentTimeMillis();
        for(int i = 1;i <= 100000;i++){
            s = ""+i;
            s1 = new StringBuffer(s);
        }
```

```
        endtime = System.currentTimeMillis();
        time = endtime - begintime;
        System.out.println("运行时间为："+time);

    }
}
```

输出：
    运行时间为：82922
    运行时间为：15
可以看得出：StringBuffer 这两个效率相当高！

StringBuffer：　是线程安全的；
StringBuilder：　是线程不安全的，性能高点，推荐使 StringBuilder；(jdk1.5 出现)
StringBuffer 的字符序列是可变的(通过 append 等方法操作)
StringBuffer　和　String 之间的转换；
String toString()　返回此序列中数据的字符串表示形式。
StringBuffer(String str):以指定的字符串创建 StringBuffer 对象。

StringBuffer 方法

public StringBuilder()构造一个不带任何字符的 StringBuilder 对象。
StringBuffer(String str) ：构造一个字符串缓冲区，并将其内容初始化为指定的字符串内容。
StringBuffer append(Object o) ：将指定的任意类型对象追加到此 StringBuffer 对象。
StringBuffer insert(int offset, Object o) :将任意类型参数的字符串表示形式插入此序列中。
StringBuffer delete(int start, int end) :移除此序列的子字符串中的字符。
StringBuffer deleteCharAt(int index): 移除此序列指定位置的 char。

Eg：

```
package reviewDemo;

/**
 * 用StringBuilder 或StringBuffer:
   把字符串“ABCDE”;
   转变成字符串“A,B,C,D”
 */

public class Demo23 {
    public static void main(String[] args) {
        //第一种方法：往里面插入；
        StringBuilder sb = new StringBuilder("ABCDE");
```

```
        sb.deleteCharAt(sb.length()-1);
        System.out.println(sb);
        for (int i = 0; i < sb.length(); i+=2) {
            sb.insert(i, ",");
        }
        sb.deleteCharAt(0);

        System.out.println(sb);


        //第二种方法：往里面追加，要追加必须遍历，必须换为数组！
        sb = new StringBuilder("ABCDE");
        sb.deleteCharAt(sb.length()-1);
        System.out.println(sb);
        char []cs = sb.toString().toCharArray();
        StringBuilder sb1 = new StringBuilder();
        for (char c : cs) {
            sb1.append(c).append(",");
        }
        sb1.deleteCharAt(sb1.length()-1);
        System.out.println(sb1);
    }
}
```

**编写一个程序，这个程序把一个整数数组中的每个元素用逗号连接成一个字符串，例如，根据内容为{1，2，3}的数组形成内容为"1,2,3"的字符串。**

```
package reviewDemo;
/**
 * 编写一个程序，
 * 这个程序把一个整数数组中的每个元素用逗号连接成一个字符串，
 * 例如，根据内容为{1，2，3}的数组形成内容为"1,2,3"的字符串。
 */

public class Demo24 {
    public static void main(String[] args) {
        int []i = {1,2,3,4,5};
        StringBuilder sb = new StringBuilder();
        for (int j : i) {
            sb.append(j).append(",");
        }
        sb.deleteCharAt(sb.length()-1);
        System.out.println(sb);
```

```
    }
}
```

## 12、Math 和 Random 和 UUID

Math 类

public final class Math extends Object

以下 X 表示 double,float,int, long

abs(X x):求绝对值

max(X x1,X x2):求最大值

min(X x1,X x2)：求最小值

public static double random()：返回带正号的 double 值，该值大于等于 0.0 且小于 1.0。和使用 new java.util.Random 一样

Math.PI;

Random 类

负责生成伪随机数；

Random() 创建一个新的随机数生成器。

int nextInt() 返回下一个伪随机数，它是此随机数生成器的序列中均匀分布的 int 值。

int nextInt(int n) 返回一个伪随机数，它是取自此随机数生成器序列的、在 0（包括）和指定值 n（不包括）之间均匀分布的 int 值。

```
package reviewDemo;

public class Demo26 {
    public static void main(String[] args) {
        System.out.println(Math.E);//2.718281828459045
        int a = 12;
        int b = 25;

        System.out.println(Math.max(a,b));//这里可以使用静态导入（导入Math类中的方法，这样的话前面就不用写上Math.）
        System.out.println(Math.min(a,b));
    }
}
```

```
package reviewDemo;

import java.util.UUID;
```

```
public class Demo27 {
    public static void main(String[] args) {
        UUID u = UUID.randomUUID();
        String s = u.toString();
        System.out.println(s);//此时是随机生成的，肯定每次都不一样，全网唯
一!

        u = new UUID(1222222222, 12);//根据构造方法来
        s = u.toString();
        System.out.println(s);//这一个的UUID是固定的。
    }
}
```

**我的总结：UUID（用来标示文件名等（免得文件上传因为名字可能一样而被覆盖），可以保证全网唯一!）**

UUID 类:用唯一标识符 (UUID) 的类。 UUID 表示一个 128 位的值。
UUID(Universally Unique Identifier)全局唯一标识符,是指在一台机器上生成的数字，它保证对在同一时空中的所有机器都是唯一的。按照开放软件基金会(OSF)制定的标准计算，用到了以太网卡地址、纳秒级时间、芯片 ID 码和许多可能的数字。由以下几部分的组合：当前日期和时间(UUID 的第一个部分与时间有关，如果你在生成一个 UUID 之后，过几秒又生成一个 UUID，则第一个部分不同，其余相同)，时钟序列，全局唯一的 IEEE 机器识别号（如果有网卡，从网卡获得，没有网卡以其他方式获得），UUID 的唯一缺陷在于生成的结果串会比较长。
标准的 UUID 格式为：xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxx (8-4-4-4-12),其中每个 x 是 0-9 或 a-f 范围内的一个十六进制的数字;
UUID uuid = UUID.randomUUID();
String uid = uuid.toString();

## 13、Date 和 Calendar

处理日期，时间;
大部分的方法已过时，不推荐使用，但是你使用过时的方法也不会报错。
Date() 分配 Date 对象并初始化此对象，以表示分配它的时间（精确到毫秒）。
Date(long date) 分配 Date 对象并初始化此对象，以表示自从标准基准时间（称为“历元（epoch）”，即 1970 年 1 月 1 日 00:00:00 GMT）以来的指定毫秒数。

SimpleDateFormat

java.text.SimpleDateFormat

SimpleDateFormat 是一个与语言环境有关的方式来格式化和解析日期的具体类。它允许进行格式化（日期 -> 文本）、解析（文本 -> 日期）和规范化。
SimpleDateFormat(String pattern)　用给定的模式和默认语言环境的日期格式符号构造 SimpleDateFormat。
public final String format(Date date)将一个 Date 格式化为日期/时间字符串。
public Date parse(String source) throws ParseException：把字符串 source 表示的时间按 source 的格式转成 Date 对象。

**练习：String 与 Date 的相互转化**

**按某种时间格式来输出指定的 String 类型时间**
**date2String**
**format**

**把某种时间格式的 String 时间转成 Date 类型时间**
**string2Date**
**parse**

```
package reviewDemo;

import java.text.SimpleDateFormat;
import java.util.Date;

public class Demo28 {
    public static void main(String[] args) throws Exception {

        Date d = new Date();
        System.out.println(d);//获取当前时间

        //格式化
        SimpleDateFormat sd = new SimpleDateFormat("yyyy-M-d
HH:mm:ss E");

        String s = sd.format(d);//这个方法继承于SimpleDateFormat的父类
DateFormat类!
        System.out.println(s);

        //反格式化
        d = sd.parse(s);
        System.out.println(d);
    }
}
```

**Calendar**

推荐使用处理日期和时间的类 Calendar;
是抽象类，不能实例化，通过
static Calendar getInstance() 获得一个 Calendar 对象。
int get(int field):返回指定日历字段值
静态常量：
YEAR 表示年的字段数字。
MONTH 表示月份字段数字,月份范围是[0,11]。
DATE 表示一个月中的某天。
DAY_OF_MONTH 表示一个月中的某天。
DAY_OF_WEEK 表示一个星期中的某天。
HOUR_OF_DAY / HOUR 表示第几小时
MINUTE 表示第几分钟
SECOND 表示第几秒

Date getTime() 返回一个表示此 Calendar 时间值的 Date 对象。
void set(int year, int month, int date, int hour, int minute, int second) 设置字段 YEAR、MONTH、DAY_OF_MONTH、HOUR、MINUTE 和 SECOND 的值。
abstract void add(int field, int amount) 根据日历的规则，为给定的日历字段添加或减去指定的时间量。
若 amount 为负数，则减去一天，
若 amount 为正数 ，则加上一天

例子

```
package reviewDemo;

import java.util.Calendar;
import java.util.Date;

public class Demo29 {
    public static void main(String[] args) {
        Calendar c = Calendar.getInstance();
        System.out.println(c);

        Date d = c.getTime();
        System.out.println(d);

        /**
         * void set(int year, int month, int date, int hour, int minute,
int second)
         * 设置字段 YEAR、MONTH、DAY_OF_MONTH、HOUR、MINUTE 和 SECOND 的
值。
```

```
         */

        c.set(2015, 07, 15, 12, 24, 55);
        d = c.getTime();
        System.out.println(d);

        c.add(Calendar.DAY_OF_MONTH, 3);
        d = c.getTime();
        System.out.println(d);
    }
}
```

**我的总结：注意常查看 api！**

Eg:“查询距今最近三天内的记录”，这里的‘三天内’是什么时间段?
将当前日期时间转换为 只获取当天的 00:00:00
如: 2011-08-08 12:12:12 转化为 2011-08-08 00:00:00、

```
package reviewDemo;

import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.Date;

public class Demo30 {
    public static void main(String[] args) {

        Calendar c = Calendar.getInstance();
        Date d = c.getTime();
        System.out.println(d);

        SimpleDateFormat sd = new SimpleDateFormat("yyyy-M-d HH:mm:ss
E");

        String s = sd.format(d);
        System.out.println(s);

        c.set(2015, 8, 23, 13, 24, 15);
        d = c.getTime();
        System.out.println(sd.format(d));
        c.add(Calendar.DAY_OF_MONTH, -3);
        d = c.getTime();
        System.out.println(sd.format(d));
```

```
    }
}
```

输出：

```
    Thu Jul 04 08:56:51 CST 2013
    2013-7-4 08:56:51 星期四
    2015-9-23 13:24:15 星期三
    2015-9-20 13:24:15 星期日
```

## 14、System

System 类包含一些与系统相关的类字段和方法。它不能被实例化，类中所有属性和方法都是 static,可直接被 System 调用。

常用方法：
static void exit(int status) 终止虚拟机的运行.对于发生了异常情况而想终止虚拟机的运行,传递一个非 0 数值,对于正常情况下退出系统传递 0 值;
该方法实际调用的是 Runtime.getRuntime().exit(int status)；
static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length) 数组拷贝
static long currentTimeMillis() 返回以毫秒为单位的当前时间。
String getenv(String name) 获得指定的环境变量;
static void gc() 运行垃圾回收器。
实际上调用了 Runtime 中的 gc()方法;
Runtime.getRuntime().exec("notepad ");
static Properties getProperties() 取得当前的系统属性。
static String getProperty(String key) 取得指定键指示的系统属性。
static String getProperty(String key, String def) 获取用指定键描述的系统属性,def 表示默认信息。

Eg：

```
package reviewDemo627;

import java.util.Properties;

public class Demo32 {
    public static void main(String[] args) {
        Properties p = System.getProperties();
        System.out.println(p);

        System.out.println(System.getenv());
    }
}
```

# 六、黑马程序员—线程技术

第六篇

## 1、进程和线程

进程是指一个内存中运行的应用程序，每个进程都有自己独立的一块内存空间，一个进程中可以有多个线程。比如在 Windows 系统中，一个运行的 xx.exe 就是一个进程。

**Java 程序的进程里有几个线程:主线程, 垃圾回收线程(后台线程)**

线程是指进程中的一个执行任务(控制单元)，一个进程中可以运行多个线程,多个线程可共享数据。

多进程:操作系统中同时运行的多个程序;

多线程:在同一个进程中同时运行的多个任务;

一个进程至少有一个线程，为了提高效率，可以在一个进程中开启多个控制单元。

并发运行。如：多线程下载软件。

可以完成同时运行，但是通过程序运行的结果发现，虽然同时运行，但是每一次结果都不一致。

因为多线程存在一个特性：随机性。

**造成的原因：CPU 在瞬间不断切换去处理各个线程而导致的。**

可以理解成多个线程在抢 cpu 资源。

**我的总结：**

**多线程下载:此时线程可以理解为下载的通道，一个线程就是一个文件的下载通道，多线程也就是同时开起好几个下载通道.当服务器提供下载服务时，使用下载者是共享带宽的，在优先级相同的情况下，总服务器会对总下载线程进行平均分配。不难理解，如果你线程多的话，那下载的越快。现流行的下载软件都支持多线程。**

**多线程是为了同步完成多项任务,不是为了提供运行效率,通过提高资源使用效率来提高系统的效率.**

**线程是在同一时间需要完成多项任务的时候实现的.**

**线程与进程的比较**

线程具有许多传统进程所具有的特征，故又称为轻型进程(Light—Weight Process)或进程元；而把传统的进程称为重型进程(Heavy—Weight Process)，它相当于只有一个线程的任务。在引入了线程的操作系统中，通常一个进程都有若干个线程，至少需要一个线程。

进程与线程的区别：

1.进程有独立的进程空间，进程中的数据存放空间（堆空间和栈空间）是独立的。

2.线程的堆空间是共享的，栈空间是独立的，线程消耗的资源也比进程小，相互之间可以影响的。

## 2、创建线程方式

### 1、继承 Thread 类

子类覆写父类中的 run 方法，将线程运行的代码存放在 run 中。

建立子类对象的同时线程也被创建。

通过调用 start 方法开启线程。

### 2、实现 Runnable 接口

子类覆盖接口中的 run 方法。

通过 Thread 类创建线程，并将实现了 Runnable 接口的子类对象作为参数传递给 Thread 类的构造函数。

Thread 类对象调用 start 方法开启线程。

可使用匿名内部类来写

Eg:

```
package july7;
//线程的两种方法

class MyThread extends Thread{
    private String name;

    public MyThread(String name) {
        super();
        this.name = name;
    }

    public void run(){
        System.out.println(name+"启动！");
    }
}

class YourThread implements Runnable{
    private String name;
    public YourThread(String name) {
        this.name = name;
    }

    @Override
    public void run() {
        for (int i = 0; i < 3; i++) {
            System.out.println(Thread.currentThread().getName()+" 第
"+i+"次启动！");
        }
    }
```

```
}

public class Demo1 {
    public static void main(String[] args) {
        for (int i = 0; i < 100; i++) {
            if(i == 50){
                new MyThread("刘昭").start();
                new Thread(new YourThread(""),"章泽天").start();
            }
        }
    }
}
```

**我的总结：**

**Thread 类中 run()和 start()方法的区别如下：**

**run()方法:在本线程内调用该 Runnable 对象的 run()方法，可以重复多次调用；**

**start()方法:启动一个线程，调用该 Runnable 对象的 run()方法，不能多次启动一个线程；**

## 3、两种进程创建方式比较

A extends Thread：

简单

不能再继承其他类了(Java 单继承)

同份资源不共享

A implements Runnable:(推荐)

多个线程共享一个目标资源，适合多线程处理同一份资源。

该类还可以继承其他类，也可以实现其他接口。

**我的总结：**

**实现方式，因为避免了单继承的局限性，所以创建线程建议使用第二种方式。**

Eg：

```
package july7;
//线程卖票的例子

class SellTicket extends Thread{
    private String name;
    private int num = 50;
    public SellTicket(String name) {
        super();
        this.name = name;
    }
```

```
    public void run(){
        for (int i = 1; i <= num; i++) {
            System.out.println(name+"卖出了第"+i+"张票！");
        }
    }
}

class MySell implements Runnable{
    private int num = 50;

    @Override
    public void run() {
        for (int i = 1; i <= num; i++) {
            System.out.println(Thread.currentThread().getName()+" 卖
出了第"+i+"张票！");
        }
    }
}

public class Demo2 {
    public static void main(String[] args) throws Exception {

        new SellTicket("A").start();
        new SellTicket("B").start();
        new SellTicket("C").start();

        new Thread(new MySell(),"D").start();
        new Thread(new MySell(),"E").start();
        new Thread(new MySell(),"F").start();

        for (int i = 10; i > 0; i--) {
            System.out.println(i);
            Thread.sleep(1000);
        }
    }
}
```

**我的总结：**

**为什么要覆盖 run 方法呢？**

**Thread 类用于描述线程。该类就定义了一个功能，用于存储线程要运行的代码。该存储功能就是 run 方法.**

**也就是说 Thread 类中的 run 方法，用于存储线程要运行的代码。**

## 4、线程的生命周期

Thread 类内部有个 public 的枚举 Thread.State，里边将线程的状态分为：

NEW-------新建状态，至今尚未启动的线程处于这种状态。

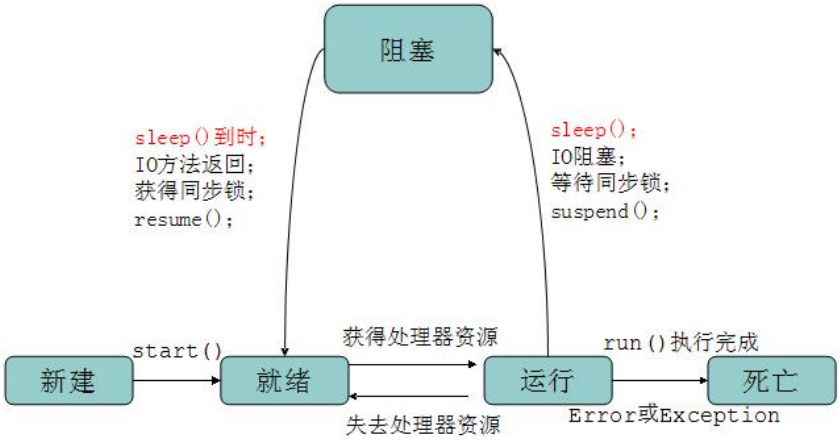
RUNNABLE-------运行状态，正在 Java 虚拟机中执行的线程处于这种状态。

BLOCKED-------阻塞状态，受阻塞并等待某个监视器锁的线程处于这种状态。

WAITING-------冻结状态，无限期地等待另一个线程来执行某一特定操作的线程处于这种状态。

TIMED_WAITING-------等待状态，等待另一个线程来执行取决于指定等待时间的操作的线程处于这种状态。

TERMINATED-------已退出的线程处于这种状态。




**我的总结：**

**如何停止线程？**

**只有一种，run 方法结束。 开启多线程运行，运行代码通常是循环结构。 只要控制住循环，就可以让 run 方法结束，也就是线程结束。**

## 5、控制线程

join 方法：调用 join 方法的线程对象强制运行，该线程强制运行期间，其他线程无法运行，必须等到该线程结束后其他线程才可以运行。

有人也把这种方式成为联合线程

join 方法的重载方法:

join(long millis):

join(long millis,int nanos):

通常很少使用第三个方法:

程序无须精确到一纳秒;

计算机硬件和操作系统也无法精确到一纳秒;

Eg：

```
package july7;

class MyThreadDemo implements Runnable{

    @Override
    public void run() {
        for (int i = 0; i < 50; i++) {
            System.out.println(Thread.currentThread().getName()+" 正
在运行！"+i);
            if(i == 25){
                try {
                    new Thread(new MyThreadDemo(),"刘昭").join();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}

public class DemoRe10 {
    public static void main(String[] args) {

        new Thread(new MyThreadDemo(),"刘昭").start();
        new Thread(new MyThreadDemo(),"章泽天").start();
    }
}
```

Daemon

后台线程：处于后台运行，任务是为其他线程提供服务。也称为“守护线程”或“精灵线程”。
JVM 的垃圾回收就是典型的后台线程。
特点：若所有的前台线程都死亡，后台线程自动死亡。
设置后台线程：Thread 对象 setDaemon(true);
setDaemon(true)必须在 start()调用前。否则出现 IllegalThreadStateException 异常；
前台线程创建的线程默认是前台线程;
判断是否是后台线程：使用 Thread 对象的 isDaemon()方法；
并且当且仅当创建线程是后台线程时，新线程才是后台线程。

sleep
线程休眠:
让执行的线程暂停一段时间，进入阻塞状态。
sleep(long milllis) throws InterruptedException：毫秒

sleep(long millis,int nanos)
throws InterruptedException:毫秒，纳秒
调用 sleep()后，在指定时间段之内，该线程不会获得执行的机会。

控制线程之优先级

每个线程都有优先级，优先级的高低只和线程获得执行机会的次数多少有关。
并非线程优先级越高的就一定先执行，哪个线程的先运行取决于 CPU 的调度;
默认情况下 main 线程具有普通的优先级，而它创建的线程也具有普通优先级。
Thread 对象的 setPriority(int x)和 getPriority()来设置和获得优先级。
MAX_PRIORITY : 值是 10
MIN_PRIORITY : 值是 1
NORM_PRIORITY : 值是 5(主方法默认优先级)

yield

线程礼让:
暂停当前正在执行的线程对象，并执行其他线程;
Thread 的静态方法，可以是当前线程暂停，但是不会阻塞该线程，而是进入就绪状态。所以完全有可能：某个线程调用了 yield()之后，线程调度器又把他调度出来重新执行。

我的总结：用到时查询 api！

## 6、多线程安全问题

导致安全问题的出现的原因：
多个线程访问出现延迟。
线程随机性。
注：线程安全问题在理想状态下，不容易出现，但一旦出现对软件的影响是非常大的。
我们可以通过 Thread.sleep(long time)方法来简单模拟延迟情况。

**我的总结：**
**当多条语句在操作同一个线程共享数据时，一个线程对多条语句只执行了一部分，还没有执行完，另一个线程参与进来执行。导致共享数据的错误。**

**解决办法：**
**对多条操作共享数据的语句，只能让一个线程都执行完。在执行过程中，其他线程不可以参与执行。**

Eg：在前面的卖票例子上，在每卖票的前面加上模拟延时的语句！

```
package july7;
```

```
class SellDemo implements Runnable{
    private int num = 50;
    @Override
    public void run() {
        for (int i = 0; i < 200; i++) {

                if(num > 0){
                    try {
                    //因为它不可以直接调用getName()方法，所以必须要获取当前线
程。
                        Thread.sleep(10);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }

    System.out.println(Thread.currentThread().getName()+" 卖 出 第
"+num--+"张票！ ");
            }
        }
    }
}

public class Demo3 {
    public static void main(String[] args) {
        SellDemo s = new SellDemo();
        new Thread(s,"A").start();
        new Thread(s,"B").start();
        new Thread(s,"C").start();
    }
}
```

输出：这样的话，会出现买了第 0，甚至-1 张票的情况！

## 7、多线程安全问题的解决方法

**三种方法：**

**同步代码块：**

```
synchronized(obj)
{
    //obj 表示同步监视器,是同一个同步对象
```

```
        /**.....
            TODO SOMETHING
        */
    }
```

**同步方法**

格式：

在方法上加上 synchronized 修饰符即可。（一般不直接在 run 方法上加！）

```
synchronized  返回值类型  方法名(参数列表)
{
    /**.....
        TODO SOMETHING
    */
}
```

同步方法的同步监听器其实的是 this

**静态方法的同步**

同步方法

同步代码块

static 不能和 this 连用

**静态方法的默认同步锁是当前方法所在类的.class 对象**

**同步锁**

jkd1.5 后的另一种同步机制：

通过显示定义同步锁对象来实现同步，这种机制，同步锁应该使用 Lock 对象充当。

在实现线程安全控制中，通常使用 ReentrantLock(可重入锁)。使用该对象可以显示地加锁和解锁。

具有与使用 synchronized 方法和语句所访问的隐式监视器锁相同的一些基本行为和语义，但功能更强大。

```
public class X {
    private final ReentrantLock lock = new ReentrantLock();
    //定义需要保证线程安全的方法
    public void    m(){
        //加锁
        lock.lock();
        try{
            //... method body
        }finally{
            //在 finally 释放锁
            lock.unlock();
```

```
        }
    }
}
```

修改后的例子：

//同步代码块

```
package july7;

class SellDemo implements Runnable{
    private int num = 50;
    @Override
    public void run() {
        for (int i = 0; i < 200; i++) {
            synchronized (this) {
                if(num > 0){
                    try {
                    //因为它不可以直接调用getName()方法，所以必须要获取当前线
程。
                        Thread.sleep(10);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }

    System.out.println(Thread.currentThread().getName()+" 卖 出 第
"+num--+"张票！ ");
                }
            }
        }
    }
}

public class Demo3 {
    public static void main(String[] args) {
        SellDemo s = new SellDemo();
        new Thread(s,"A").start();
        new Thread(s,"B").start();
        new Thread(s,"C").start();
    }
}
```

//同步方法

```
package july7;
```

```
//同步方法

class FinalDemo1 implements Runnable {
    private int num = 50;

    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            gen();
        }
    }

    public synchronized void gen() {
        for (int i = 0; i < 100; i++) {
            if (num > 0) {
                try {
                    Thread.sleep(10);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println(Thread.currentThread().getName() +
"卖出了第"
                        + num-- + "张票! ");
            }
        }
    }
}

public class Demo6 {
    public static void main(String[] args) {
        FinalDemo1 f = new FinalDemo1();

        new Thread(f, "A").start();
        new Thread(f, "B").start();
        new Thread(f, "C").start();

    }
}


//线程同步锁
package july7;

import java.util.concurrent.locks.ReentrantLock;
```

```
//同步锁

class FinalDemo2 implements Runnable {
    private int num = 50;
    private final ReentrantLock lock = new ReentrantLock();

    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            gen();
        }
    }

    public void gen() {
        lock.lock();
        try{
            //for (int i = 0; i < 100; i++) {
                if (num > 0) {
                    try {
                        Thread.sleep(10);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }

    System.out.println(Thread.currentThread().getName() + "卖出了第"
                            + num-- + "张票! ");
                }
            //}
        }finally{
            lock.unlock();
        }
    }
}

public class Demo7 {
    public static void main(String[] args) {
        FinalDemo2 f = new FinalDemo2();

        new Thread(f, "A").start();
        new Thread(f, "B").start();
        new Thread(f, "C").start();

    }
```

}

## 8、线程通信

有一个数据存储空间,划分为两部分,一部分用于存储人的姓名,另一部分用于存储人的性别;
我们的应用包含两个线程,一个线程不停向数据存储空间添加数据(生产者),另一个线程从数据空间取出数据(消费者);
因为线程的不确定性，存在于以下两种情况：
若生产者线程刚向存储空间添加了人的姓名还没添加人的性别，CPU 就切换到了消费者线程，消费者线程把姓名和上一个人的性别联系到一起；
生产者放了若干数据，消费者才开始取数据，或者是消费者取完一个数据，还没等到生产者放入新的数据，又重复的取出已取过的数据；

生产者和消费者

wait():让当前线程放弃监视器进入等待，直到其他线程调用同一个监视器并调用 notify()或 notifyAll()为止。
notify():唤醒在同一对象监听器中调用 wait 方法的第一个线程。
notifyAll():唤醒在同一对象监听器中调用 wait 方法的所有线程。

这三个方法只能让同步监听器调用:
在同步方法中:　　谁调用
在同步代码块中:　谁调用

wait()、notify()、notifyAll()，这三个方法属于 Object 不属于 Thread，这三个方法必须由同步监视对象来调用，两种情况：

1.synchronized 修饰的方法，因为该类的默认实例(this)就是同步监视器，所以可以在同步方法中调用这三个方法;

2.synchronized 修饰的同步代码块，同步监视器是括号里的对象，所以必须使用该对象调用这三个方法;

可要是我们使用的是 Lock 对象来保证同步的,系统中不存在隐式的同步监视器对象,那么就不能使用者三个方法了,那该咋办呢?

此时,Lock 代替了同步方法或同步代码块,Condition 代替了同步监视器的功能;

Condition 对象通过 Lock 对象的 newCondition()方法创建;

里面方法包括:

await():　等价于同步监听器的 wait()方法;

signal(): 等价于同步监听器的 notify()方法;

signalAll(): 等价于同步监听器的 notifyAll()方法;

例子：设置属性
容易出现的问题是：
　　名字和性别不对应！

线程通信，很好！

```
package july7;

class Person{
    private String name;
    private String sex;
    private Boolean isimpty = Boolean.TRUE;//内存区为空!
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getSex() {
        return sex;
    }
    public void setSex(String sex) {
        this.sex = sex;
    }

    public void set(String name,String sex){
        synchronized (this) {
            while(!isimpty.equals(Boolean.TRUE)){//不为空的话等待消费者
消费!
                try {
                    this.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            this.name = name;//为空的话生产者创造!
            this.sex = sex;
            isimpty = Boolean.FALSE;//创造结束后修改属性!
            this.notifyAll();
        }
    }

    public void get(){
        synchronized (this) {
            while(!isimpty.equals(Boolean.FALSE)){
                try {
                    this.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
```

```
                }
            }
            System.out.println(" 姓 名 "+getName()+   ",     "+" 性 别
"+getSex());
            isimpty = Boolean.TRUE;
            this.notifyAll();
        }
    }
}

class Producer implements Runnable{
    private Person p;

    public Producer(Person p) {
        super();
        this.p = p;
    }

    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            if( i % 2 == 0){
                p.set("刘昭", "男");
            }else{
                p.set("章泽天", "女");
            }
        }
    }
}

class Consumer implements Runnable{
    private Person p;

    public Consumer(Person p) {
        super();
        this.p = p;
    }

    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            p.get();
        }
    }
```

```
}

public class Demo9 {
    public static void main(String[] args) {
        Person p = new Person();

        new Thread(new Producer(p)).start();
        new Thread(new Consumer(p)).start();
    }
}
```

# 七、黑马程序员—集合框架（1）

第七篇　集合框架（1）

## 1、集合类 &　容器

为什么出现集合类?

面向对象语言对事物的体现都是以对象的形式，所以为了方便对多个对象的操作，就要对对象进行存储，集合就是存储对象最常用的一种方式。

数组和集合类同是容器，有何不同?

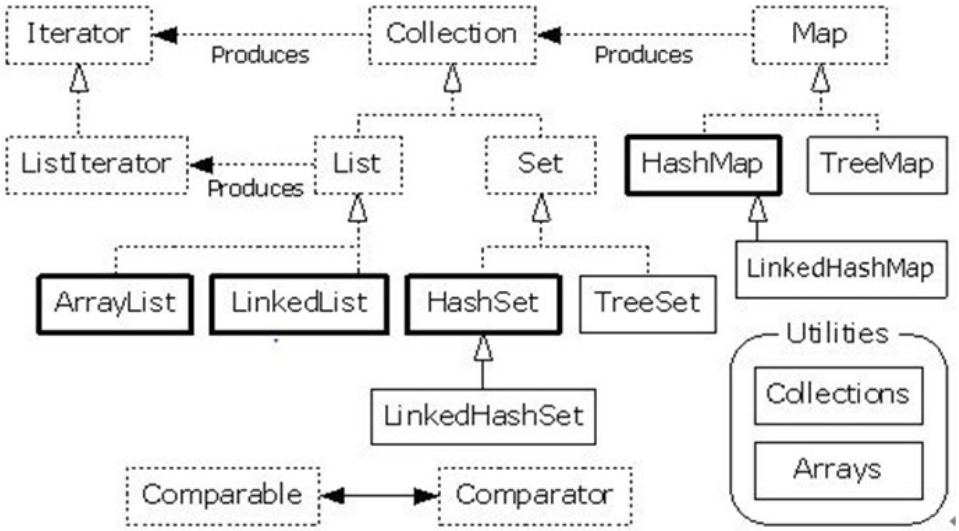
数组虽然也可以存储对象，但长度是固定的；集合长度是可变的。数组中可以存储任意数据类型，集合只能存储对象。

集合类的特点

集合只用于存储对象，集合长度是可变的，集合可以存储不同类型的对象。

**我的总结：集合类比数组强大，类似于包装类与基本数据类型！**

**集合框架的构成及分类**




**两大接口**

Java 集合类主要由两个接口派生出来：

**Collection**

Set ：不能存放重复对象

List ：可存放重复对象，有序

Queue ：队列

SortedSet：可对集合数据排序

**Map**

SortedMap ：可对集合数据排序

## 2、Collection 接口

方法：

boolean add(Object o):该方法用于向集合里面添加一个元素，若集合对象被添加操作改变了，返回 true.

boolean addAll(Collection c):把集合 c 里面的所有元素添加到指定集合里面去，如果集合对象被添加操作改变了返回 true.

void clear():清除集合里面的所有元素，将集合长度变为 0。

boolean contains(Object o):返回集合里是否包含指定的元素。

boolean containsAll(Collection c):返回集合里是否包含集合 c 内所有的元素。

boolean isEmpty()：返回集合是否为空（长度是否为 0）。

Iterator iterator():返回一个 Iterator 对象，用于遍历集合里的元素。

boolean remove(Object o)：删除集合中指定元素 o。

boolean removeAll(Collection c):从集合中删除集合 c 里面的元素。若删除一个或以上返回 true。

boolean retainAll(Collection c):从集合中删除集合 c 里不包含的元素。

int size()：得到集合元素的个数。

Object[] toArray():把集合转成一个数组，所有集合元素编程数组元素。

Eg：

```
public class Demo10 {
    public static void main(String[] args) {
        Collection<String> c = new ArrayList<String>();

        c.add("A");
        c.add("A");
        c.add("A");
        c.add("A");
        //c.add(new Date());上面写了泛型，这个语句就是错误的了，因为泛型规定了加进来的类型！
        System.out.println(c);
    }
}
```

## 3、Iterator 接口

**Iterator 主要遍历 Collection 集合中的元素，也有称为迭代器或迭代精灵。**

boolean hasNext()：若被迭代的集合元素还没有被遍历，返回 true.

Object next():返回集合的下一个元素.

void remove():删除集合上一次 next()方法返回的元素。(若集合中有多个相同的元素，都可以删掉)

iterator 对于集合才能用，for 不同，只要是循环都可用。

**迭代是取出集合中元素的一种方式。**

因为 Collection 中有 iterator 方法，所以每一个子类集合对象都具备迭代器。

迭代器在 Collcection 接口中是通用的，它替代了 Vector 类中的 Enumeration(枚举)。迭代器的 next 方法是自动向下取元素，要避免出现 NoSuchElementException。

迭代器的 next 方法返回值类型是 Object，所以要记得类型转换。(学到泛型就可以消除强转！)

Eg：Iterator iter = l.iterator();

```
while(iter.hasNext()){
    System.out.println(iter.next());
}
```

## 4、集合遍历输出方式

Iterator：迭代输出

一旦操作集合的遍历输出，首选 Iterator 接口;

ListIterator:Iterator 子接口，专门输出 List 中的元素;

Enumeration:古老的输出方式，迭代 Vector 元素，被 Iterator 取代;

foreach:可输出数组和 Iterable 对象;

**我的总结：**
**集合遍历输出方式：**
**1. Iterable 接口（迭代遍历） 注：优先选择 Iterator 接口，遍历 Collection 里所有元素，也称为迭代器和迭代精灵；迭代是取出集合中元素的一种推荐方式。**
**2. Foreach 循环 注：可以直接用，使用场合：数组和 Iterable 对象！**
**3. For 循环 注：在（）内实例化 Iterable 对象，进行遍历！**
**4. 先用 toArray 方法输出成为数组，再用 Foreach 循环！**
**第一个和第三个很类似，第二个和第四个很类似！**

很好
Eg：

```
package july7;
//遍历输出的四种方式

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class Demo11 {
    public static void main(String[] args) {

        List<String> l = new ArrayList();

        l.add("1");
        l.add("2");
        l.add("3");
        l.add("4");

        System.out.println(l);
        /**
         * 遍历输出：四种方式
         */

        // 1、推荐使用的Iterator对象，迭代输出！
        Iterator it = l.iterator();
        while (it.hasNext()) {
            System.out.println("迭代输出： " + it.next());
        }

        // 2、对集合进行fore循环！
        for (String str : l) {
            System.out.println("fore集合迭代输出： " + str);
```

```
        }

        // 3、for循环迭代，在for方法体内部实例化Iterator对象!
        int i = 0;// for方法体内定义项不能出现多种不同类型
        for (Iterator iter = l.iterator(); i < l.size(); i++) {
            System.out.println("for循环迭代实例化输出： " + iter.next());
        }

        // 4、先将集合转换为数组，再利用数组的遍历输出;
        Object[] o = l.toArray();
        for (Object object : o) {
            System.out.println("转换数组迭代输出： " + object);
        }
    }
}
```

## 5、Set 接口（元素不可以重复）

Set 是 Collection 子接口;
Set 和 Collection 基本上一样，一点除外：
Set 无法记住添加的顺序，不允许包含重复的元素。
　　当试图添加两个相同元素进 Set 集合，添加操作失败，add()方法返回 false。
Set 判断两个对象是否相等用 equals,而不是使用==。
　　也就是说两个对象 equals 比较返回 true，Set 集合是不会接受这个两个对象的。
常用子类：
　　HashSet:散列存放
　　TreeSet:有序存放

**hashCode 方法对于 HashSet 的作用**

HashSet 类是 Set 接口最常用的实现类,采用 hash 算法存储数据,具有良好的存储和查找功能。
散列存储:不记录添加顺序;排列顺序时，**顺序有可能发生变化;**
线程不安全的，多个线程访问一个 HashSet 要使用同步代码;
HashSet 集合元素值允许是 null,但是最多只能有一个;//因为 Set 集合就不可以装重复的对象!
hash(翻译为哈希,或散列)算法的功能:
保证通过一个对象快速找到另一个对象;
其算法价值体现在速度,可以保证查询快速执行;
当从 HashSet 中访问元素时,HashSet 先计算该元素的 hashCode(也就是该对象的 hashCode 方法返回值),然后直接到该 HashCode 对应的位置取出该元素;
在这里对象的 hashCode 就好比是数组里的索引,但是不是索引;

**HashSet 元素添加**

当向 HashSet 集合中存入一个元素时，HashSet 会调用该对象的 hashCode()方法来得到该对象的 hashCode 值,判断已经存储在集合中的对象的 hashCode 值是否与添加的对象的 hashCode 值一致:若不一致:直接添加进去;若一致,再进行 equals 方法比较,equals 方法如果返回 true,表明对象已经添加进去了,就不会再添加新的对象了,否则添加进去;
如果我们重写了 equals 方法,也要重写 hashCode 方法,反之亦然;。
HashSet 集合判断两个元素相等的标准是两个对象通过 equals 方法比较相等，并且两个对象的 hashCode 方法返回值也相等。
**如果需要某个类的对象保存到 HashSet 集合中，覆写该类的 equals()和 hashCode()方法，应该尽量保证两个对象通过 equals 比较返回 true 时,他们的 hashCode 返回也相等。**

**我的总结：**
**很重要的一点：理解！往 HashSet 集合里面存入数据，要先后调用两个方法：hashCode 方法和 equals 方法!!!**
**备注：使用 eclipse 添加这两个方法。**

Eg:

```
package july7;
//hashset方法和equals方法判断输入的对象是否重复！

import java.util.HashSet;
import java.util.Set;

class PersonDemo{
    private String name;

    public PersonDemo(String name) {
        super();
        this.name = name;
    }

    @Override
    public String toString() {
        return "name= " + name ;
    }
    //没有覆写hashcode和equals方法前，显示三次（一样的）。覆写后，只剩下一个了！说明覆写后方法起作用了，重复的输入不进去！
    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + ((name == null) ? 0 : name.hashCode());
        return result;
    }
```

```
    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        PersonDemo other = (PersonDemo) obj;
        if (name == null) {
            if (other.name != null)
                return false;
        } else if (!name.equals(other.name))
            return false;
        return true;
    }
}

public class Demo12 {
    public static void main(String[] args) {

        Set s = new HashSet();

        s.add(new PersonDemo("章泽天"));
        s.add(new PersonDemo("章泽天"));
        s.add(new PersonDemo("章泽天"));
        System.out.println(s);
    }
}
```

## 6、TreeSet

TreeSet 是 SortedSet 接口唯一的实现，与 HashSet 相比额外的方法有：
Comparator comparator():返回当前 Set 使用的 Comparator,若返回 null,表示以自然顺序排序。
Object first() 返回此 set 中当前第一个(最低)元素。
Object last() 返回此 set 中当前最后一个(最高)元素。
SortedSet subSet(Object fromElement, E toElement) 返回此 set 的部子集，其元素从 fromElement（包括）到 toElement（不包括）。
SortedSet headSet(Object toElement)返回此 set 的部分子集，其元素严格小于 toElement。
SortedSet tailSet(Object fromElement) 返回此 set 的部分子集，其元素大于等于 fromElement。

**TreeSet 的排序之自然排序**

TreeSet 会调用元素的 compareTo(Object o)方法来比较元素之间的大小关系,然后将集合里的元素按升序排列.此时需要排序元素的类**必须实现 Compareble 接口,并覆写其 int compareTo(Object o)方法;**
该方法用于比较对象,若:obj1,compareTo(obj2),返回 0,表示两个对象相等,若返回一个正整数,表示 obj1 大于 obj2,若返回一个负整数,表示 obj1 小于 obj2;
对于 TreeSet 集合而言,判断两个对象相等的标准是:
compareTo()方法比较返回 0;

```
package july7;
//TreeSet可以自动进行排序！最简单的情况

import java.util.Set;
import java.util.TreeSet;

public class Demo13 {
    public static void main(String[] args) {

        Set<Integer> s = new TreeSet<Integer>();
        s.add(1);
        s.add(192);
        s.add(123);

        s.add(56);
        s.add(13);
        s.add(96);
        System.out.println(s);//[1, 13, 56, 96, 123, 192]
    }
}
```

稍复杂点的

```
package july7;
//TreeSet的自然排序，升序

import java.util.Set;
import java.util.TreeSet;

class Student implements Comparable{//必须实现接口
    private Integer age;

    public Student(Integer age) {
```

```
        super();
        this.age = age;
    }

    @Override
    public int compareTo(Object o) {//比较的规则，运用泛型可以消除强转！
        if(o instanceof Student){
            Student s = (Student)o;
            return this.age.compareTo(s.age);
        }
        return 0;
    }

    @Override
    public String toString() {
        return age+"" ;
    }
}

public class Demo14 {
    public static void main(String[] args) {

        Set<Student> s = new TreeSet();
        s.add(new Student(140));
        s.add(new Student(15));
        s.add(new Student(11));
        s.add(new Student(63));
        s.add(new Student(96));
        System.out.println(s);//[11, 15, 63, 96, 140]
    }
}
```

**TreeSet 的排序之定制排序**

TreeSet 的自然排序是根据元素的大小进行升序排序的,若想自己定制排序,比如降序排序,就可以使用 **Comparator 接口了:**
该接口包含 int compare(Object o1,Object o2)方法,用于比较两个对象的大小,比较结果和 compareTo 方法一致;
**要实现定制排序,需要在创建 TreeSet 集合对象时,提供一个一个 Comparator 对象,该对象里负责集合元素的排序逻辑;**
TreeSet(Comparator comparator)

Eg:

```java
package july7;
//定制排序的话，必须在创建TreeSet集合对象的时候提供一个Comparator方法

import java.util.Comparator;
import java.util.Set;
import java.util.TreeSet;

class Student1{
    private Integer age;

    public Student1(Integer age) {
        super();
        this.age = age;
    }

    public Integer getAge() {
        return age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }

    @Override
    public String toString() {
        return age + "";
    }
}

class MyComparator implements Comparator{

    @Override
    public int compare(Object o1, Object o2) {
        if(o1 instanceof Student1 & o2 instanceof Student1){
            Student1 s1 = (Student1)o1;
            Student1 s2 = (Student1)o2;
            if(s1.getAge() > s2.getAge()){
                return -1;
            }else if(s1.getAge() < s2.getAge()){
                return 1;
            }
        }
        return 0;
    }
```

```
}

public class Demo15 {
    public static void main(String[] args) {
        Set<Student1> s = new TreeSet(new MyComparator());
        /**
         * 要实现定制排序，需要在创建TreeSet集合对象时，提供一个一个Comparator对象，
         * 该对象里负责集合元素的排序逻辑；
         */
        s.add(new Student1(140));
        s.add(new Student1(15));
        s.add(new Student1(11));
        s.add(new Student1(63));
        s.add(new Student1(96));

        System.out.println(s);
    }
}
```

## 7、List 接口

Collection 子接口；
List 是有序的集合，集合中每个元素都有对应的顺序序列。List 集合可使用重复元素，可以通过索引来访问指定位置的集合元素（顺序索引从 0 开始）,List 集合默认按元素的添加顺序设置元素的索引,比如第一个元素的索引就是 0,好似数组。
List 作为 Collection 子接口当然拥有其所有方法，同时也有自己的方法：

void add(int index,Object e):将元素 e 添加到 List 集合中的 index 处；
boolean addAll(int index,Collection c):将集合 c 所包含的所有元素都插入在 List 集合的 index 处；
Object get(int index):返回集合 index 索引处的元素；
int indexOf(Object o):返回对象 o 在 List 集合中第一次出现位置的索引；
int lastIndexOf(object o):返回对象 o 在 List 集合中最后一次出现的位置索引；
Object remove(int index)：删除并返回 index 索引处的元素；
Object set(int index,Object e):把集合 index 处的元素替换为 e 对象，返回以前在指定位置的元素；
List subList(int fromIndex,int toIndex):返回从所有 fromIndex(包括)到 toIndex(不包括)处所有集合元素的子集合。

ListIterator

Iterator 的子接口，专门用于操作 List 集合的输出；

List 自己还有一个 listIterator()方法，该方法返回 ListIterator 对象，ListIterator 继承了 Iterator 接口，提供了专门操作 List 的方法。在 Iterator 上额外增加的方法：
支持双向输出：
boolean hasPrevious():返回该迭代器关联集合是否还有上一个元素；
Object previous():返回该迭代器的上一个元素；

**我的总结：这是相对更加特殊的一个接口，只用于 List 集合，可以完成逆序输出！**

Eg：

```
package july7;

import java.util.ArrayList;
import java.util.List;
import java.util.ListIterator;

public class Demo16 {
    public static void main(String[] args) {
        List<Integer> l = new ArrayList<Integer>();
        l.add(12);
        l.add(24);
        l.add(36);
        l.add(23);
        l.add(37);

        System.out.println(l);

        ListIterator<Integer> it = l.listIterator();
        while(it.hasNext()){
            System.out.println("正序: "+it.next());
        }
        System.out.println("==================================");
        //使用向前迭代前必须将游标（指针）移动到后边！
        while(it.hasPrevious()){
            System.out.println("逆序: "+it.previous());
        }
    }
}
```

输出：

```
    [12, 24, 36, 23, 37]
    正序: 12
    正序: 24
    正序: 36
    正序: 23
```

```
正序：37
==================================
逆序：37
逆序：23
逆序：36
逆序：24
逆序：12
```

## 8、List 接口中常用类

**Vector：线程安全，但速度慢，已被 ArrayList 替代。**
**ArrayList：线程不安全，查询速度快。**
**LinkedList：链表结构，增删速度快。取出 List 集合中元素的方式：**
get(int index)：通过脚标获取元素。
iterator()：通过迭代方法获取迭代器对象。
ArrayList 和 Vector 类都是基于数组实现的 List 类,Vector 比较古老,被 ArrayList 取代了;
ArrayList 是线程不安全的,而 Vector 是线程安全的,但是即使这样,也不推荐使用 Vector,因为 Collections 有方法可以得到线程安全的 ArrayList 对象;
Collections 类: static List synchronizedList(List list) 返回指定列表支持的同步（线程安全的）列表。

**Queue 接口**

继承 Collection 接口
模拟队列：先进先出(FIFO);
void add(Object e):将 e 插入到队列尾部；
Object element():获取队列头部的元素；
boolean offer(Object e):将 e 插入到队列的尾部，当使用有容量限制的队列时，此方法比 add(Object e)方法更好。
Object peek():获取队列头部的元素。如果此双端队列为空，则返回 null。
Object poll():获取并删除队列头部的元素。如果此双端队列为空，则返回 null。
Object remove():获取并删除队列头部的元素。

Eg：**练习：List 的使用**

设计 Department 和 Employee 对象类(按照开发的样式没有写在一个文件里面)
//一个员工只可以有一个部门，所以按照普通的写法
//一个部门可以有许多员工，所以按照集合的写法；

//员工

```java
package re538.Department;

public class Emp {
    private Dept dept;//一个员工只可以有一个部门，所以按照普通的写法
    private String name;
    public Dept getDept() {
        return dept;
    }
    public void setDept(Dept dept) {
        this.dept = dept;
    }
    public Emp(String name) {
        super();
        this.name = name;
    }
    public String toString(){//覆写toString方法
        return this.name+"";
    }
}
```

```java
//部门
package re538.Department;

import java.util.List;
public class Dept {
    private String name;
    private List emp;//一个部门可以有许多员工，所以按照集合的写法；部门里的属性，员工！
    public Dept(String name) {
        super();
        this.name = name;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public List getEmp() {
        return emp;
    }
    public void setEmp(List emp) {
        this.emp = emp;
```

```
    }
    private Dept parent;//一个部门只有一个直接主管部门，所以普通写法
    private List children;//但是一个部门可以管理很多部门，充当很多部门的直接主管部门，所以使用集合
    public Dept getParent() {
        return parent;
    }
    public void setParent(Dept parent) {
        this.parent = parent;
    }
    public List getChildren() {
        return children;
    }
    public void setChildren(List children) {
        this.children = children;
    }
    public String toString(){
        return this.name.toString();
    }
}

//测试类，主方法！

package re538.Department;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class TestDemo {

    public static void main(String[] args) {
        Emp e1 = new Emp("刘亦菲");
        Emp e2 = new Emp("章泽天");
        Emp e3 = new Emp("张月娟");
        Emp e4 = new Emp("王二五");
        Emp e5 = new Emp("刘昭");
        List list = new ArrayList();//把员工丢尽集合去
        list.add(e1);
        list.add(e2);
        list.add(e3);
        list.add(e4);
        list.add(e5);
        for (Object object : list) {//把员工给遍历出来，使用的是直接的
```

```
foreach（四种方法之一）
            System.out.println("----->"+object);
        }
        System.out.println("--------------------");
        Dept d = new Dept("部门");//定义出一个部门来接收员工
        d.setEmp(list);//把员工这个集合丢到部门去。
        Dept d1 = new Dept("itcast");
        Dept cditcast = new Dept("成都传智播客");
        Dept gzitcast = new Dept("广州传智播客");
        Dept d4 = new Dept("CSDN");
        System.out.println(d4);
        List children = new ArrayList();
        children.add(cditcast);
        children.add(gzitcast);
        d1.setChildren(children);//定义直接被主管部门（集合）
        d1.setParent(d4);//定义直接主管部门
        Iterator it = children.iterator();//运用到了迭代遍历，四种之一！
        while(it.hasNext()){
            System.out.println(it.next());
        }
    }
}
```

## 9、Map 接口

**映射关系,也有人称为字典,Map 集合里存在两组值，一组是 key,一组是 value。Map 里的 key 不允许重复。通过 key 总能找到唯一的 value 与之对应。**

Map 里的 key 集存储方式和对应的 Set 集合中的元素存储方式一致;

学生都有一个学号，我们能点学号就能找到某个学生，好比这个学号就是 key，学生就是 value。

Map.Entry 是 Map 接口的内部接口，专门用来保存 key-value 内容：

Map 常用方法

void clear():删除该 Map 对象中所有的 key-value 对。也就是清理该集合；

boolean containsKey(Object key):查询 Map 中是否包含指定的 key；

boolean containsValue(Object value):查询 Map 中是否包含至少一个 value;

Set entrySet():返回 Map 所包含的 key-value 对所组成的 Set 集合，每个集合元素都是 Map.Entry 对象(Entry 是 Map 内部类)；

Object get(Object key):返回指定 key 所对应的 value,若此 Map 中不包含该 key,返回 null;

boolean isEmpty():判断 Map 集合是否为空；

Set keySet():返回该 Map 中所有 key 所组成的 Set 集合；

Object put(Object key,Object value):添加一个 key-value 对，若 Map 中已有与 key 相等的 key-value 对，则新的 key-value 对覆盖原来的 key-value 对；
void putAll(Map m):将 m 中的 key-value 赋值到调用该方法的 Map 对象中；
Object remove(Object key):删除指定 key 所对应的 key-value 对，返回本删除 key 所关联的 value,若 key 不存在，返回 null;
int size():返回该 Map 里面 key-value 对的个数；
Collection values():返回 Map 里所有 value 组成的 Collection。

Eg：

```
package july7;

import java.util.Collection;
import java.util.HashMap;
import java.util.Map;
import java.util.Set;

public class Demo17 {
    public static void main(String[] args) {

        Map m = new HashMap();

        m.put(1, "jack");
        m.put(2, "rose");
        m.put(3, "lucy");

        System.out.println(m);//{1=jack, 2=rose, 3=lucy}

        Set s = m.entrySet();
        System.out.println(s);//[1=jack, 2=rose, 3=lucy]

        s = m.keySet();
        System.out.println(s);//[1, 2, 3]

        for (Object key : s) {
            System.out.println(key);//Key
            System.out.println(m.get(key));//Value
        }

        Collection c = m.values();
        System.out.println(c);
    }
}
```

**Map.Entry**

Entry 是 Map 接口里面的一个内部接口.
该接口用于封装 key- value,有 3 个方法:
Object getKey();返回 Entry 里包含的 key 值
Object getValue();返回 Entry 里包含的 value 值
Object setValue(Object value):设置 Entry 里包含的 value 值,并返回新设置的 value 值;

Map 集合的输出

按照最正统的做法，所有的 Map 集合的内容都要依靠 Iterator 输出，以上虽然是完成了输出，但是完成的不标准，Map 集合本身并不能直接为 Iterator 实例化，如果此时非要使用 Iterator 输出 Map 集合中内容的话，则要采用如下的步骤：

**方法一：**

**1.通过 entrySet 方法变成 Set 对象**
**2.调用 Set 的 Iterator 方法，此时每个 Iterator 对象是 Map.Entry 对象**
**3.对 Map.Entry 分离出 key - value**

**方法二：**

**1.通过 keySet 得到 Map 集合多有 key 的 Set 集合**
**2.调用 Set 的 Iterator 方法，此时每个 Iterator 对象是 key 值**
**3.通过 Map 的 getValue(key)得到 value 值**

Eg:

```
package july7;
//把Map中的元素取出来

import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Map.Entry;
import java.util.Set;

public class Demo18 {
    public static void main(String[] args) {

        Map<Integer, String> m = new HashMap<Integer, String>();

        m.put(1, "jack");
        m.put(2, "rose");
        m.put(3, "lucy");

        //第一种方法
        Set s = m.entrySet();
        Iterator it = s.iterator();
```

```
        while(it.hasNext()){
            Map.Entry me = (Entry) it.next();
            System.out.println(me.getKey()+"  "+me.getValue());
        }

        //第二种方法

        s = m.keySet();//得到的是key的集合
        it = s.iterator();//然后将key迭代出来
        while(it.hasNext()){
            int i = (int)it.next();
            System.out.println(i+"   "+m.get(i));
        }
    }
}
```

# 八、黑马程序员—集合框架（2）

第八篇　集合框架（2）

## 1、Collections 类

操作集合的工具类：
static void reverse(List list):反转指定 List 集合中的顺序；（和 ListItertor 的逆序排列一样！）
static void shuffle(List list):对集合元素随机排序
static void sort(List list):自然升序排序
static vois swap(List list,int i, int j):将指定的 List 集合 i 处元素和 j 处元素进行交换；
static void rotate(List list, int distance):
　　若 distance 为正数，将 list 集合后的 distance 个元素移到前面；
　　当 distance 为负数，将 list 集合前的 distance 个元素移到后面；
static　int binarySearch(List list, Object key)　使用二分搜索法搜索指定列表，以获得指定对象。
调用之前　必须调用　Collections.sort(List list)(完成自然排序);

static Object max(Collection coll)　根据元素的自然顺序，返回给定　collection　的最大元素。
static Object min(Collection coll)　根据元素的自然顺序，返回给定　collection　的最小元素。

static　void fill(List list, Object obj)　使用指定元素替换指定列表中的所有元素。
static int frequency(Collection c, Object o)　返回指定　collection　中等于指定对象的元素数。
static int indexOfSubList(Listsource, List target)　返回指定源列表中第一次出现指定目标列表的起始位置；如果没有出现这样的列表，则返回　-1。

static int lastIndexOfSubList(List source, List target)　返回指定源列表中最后一次出现指定目标列表的起始位置；如果没有出现这样的列表，则返回 -1。
static boolean　replaceAll(List list, Object oldVal, Object newVal)　使用另一个值替换列表中出现的所有某一指定值。
同步集合

Eg：

```
import static java.util.Collections.binarySearch;
import static java.util.Collections.shuffle;
import static java.util.Collections.sort;

import java.util.ArrayList;
import java.util.List;

public class CollectionsDemo {
    public static void main(String[] args) {
        /**
         * static int binarySearch(List list, Object ele) 使用二分搜索法搜索指定列表，以获得指定对象。
         *
         * static void sort(List<T> list) 根据元素的自然顺序 对指定列表按升序进行排序。
         */
        List list = new ArrayList();
        list.add(1);
        list.add(-3);
        list.add(5);
        list.add(-99);

        System.out.println(list);
        sort(list);
        System.out.println(list);
        int index = binarySearch(list, 0);
        System.out.println(index);

        /*
         * static void copy(List dest, List src) 将所有元素从一个列表复制到另一个列表。
         *
         * static void fill(List list, Object obj) 使用指定元素替换指定列表中的所有元素。
         *
         * static boolean replaceAll(List list, Object oldVal, Object newVal)
```

```
         * 使用另一个值替换列表中出现的所有某一指定值。
         *
         * static void shuffle(List list) 使用默认随机源对指定列表进行置
换。
         */
        System.out.println("好的顺序" + list);
        shuffle(list);
        System.out.println("随机" + list);

        /*
         * static void swap(List list, int i, int j) 在指定列表的指定位
置处交换元素。
         */

    }
}
```

## 2、Arrays

public static　List asList(Object... a)返回一个受指定数组支持的固定大小的列表(返回的是不可变的 List(长度固定))。
（对返回列表的更改会“直接写”到数组。）此方法同 Collection.toArray() 一起，充当了基于数组的 API 与基于 collection 的 API 之间的桥梁。返回的列表是可序列化的，并且实现了 RandomAccess。

此方法还提供了一个创建固定长度的列表的便捷方法，该列表被初始化为包含多个元素：

List<String> list= Arrays.asList("Larry", "Moe", "Curly");
list.add("Will");×

Eg:

```
import java.util.Arrays;
import java.util.List;


public class ArraysDemo {
    public static void main(String[] args) {

        /*
         * static  List  asList(Object... a)
                    返回一个固定长度的列表。
         * */
        List list = Arrays.asList("will","Lucy","小强");
```

```
        System.out.println(list);
        list.set(0, "你好");
        //list.add("22");//错误，返回一个受指定数组支持的固定大小的列表。不可以再添加！
        //list.remove(0);
        System.out.println(list);
    }
}
```

## 3、泛型(Generic)

引入：我想使用 List 集合装公司的员工，却装了一条狗进来；
使用 TreeSet 的时候,只能存放同一种数据类型,可惜存了不同的数据类型,依然没有报错,可是运行时出错.

**泛型定义：**

java5 开始出现的一种对 Java 语言类型的一种拓展,以支持创建可以按类型进行参数化的类.可以把类型参数看作是使用参数类型时指定的类型占位符,就好比方法的形式参数是实际参数的占位符一样.

**泛型能保证大型应用程序的类型安全和良好的维护性；**

**使用泛型的优势:**

类型安全,使编译器对泛型定义的类型做判断限制.如保证 TreeSet 里的元素类型必须一致;消除强制类型的转换,如,使用 Comparable 比较时每次都需要类型强转；

## 4、泛型的使用

### 泛型类

**在类声明时通过一个标识符表示类中某个字段的类型或者某个方法的返回值或参数的类型，这样在类声明或实例化的时候只要指定自己需要的类型就 ok。**

**声明带泛型的类:**

```
class 类名<泛型类型 1,泛型类型 2……>{
    泛型类型  变量名;
    泛型类型  方法名(){}
    返回值类型 方法名(泛型类型 变量名){}
}
```

**使用带泛型的类:**

```
类名<具体类> 对象名 = new 类名<具体类>();
```

类型参数规范：推荐使用规范-常见的泛型,泛型只保存在源文件中,class 文件中不存在;也就是说在编译阶段就会丢失,基本数据类型不能作为泛型类型;

K　键，比如映射的键　key 的类型

V　值，比如 Map 的值　value 类型

E　元素，比如 Set<E>　Element 表示元素,元素的类型

T　泛型，Type 的意思

我的总结：泛型好处：限定添加类型和消除强转转换的麻烦！

泛型使用

```
public class Point<Q> {//声明任意符号的标识符
    private Q x; //变量类型由外部组成
    private Q y;

    public Q getX() {
        return x;
    }
    public void setX(Q x) {//类型由外部决定
        this.x = x;
    }
    //.........................
}

.................main.........
{
    Point<Double> p = new Point<Double>();  //定义具体类型
    p.setX(1.1);
    p.setY(2.2);
}
```

练习例子

需求：设计一个表示点的类 Point,该类有两个字段，一个是横坐标 x,一个纵坐标 y,要求坐标有 3 种表达形式（Integer，Double，String）：

如果不使用泛型的话可以新建多个类，但是内部方法体只有参数类型不一样，所以用泛型的话更加简单，给定一个占位符，并不明确表示到底是什么类型，在实际运用的时候才确定类型!!

很好的例子!

```
package generic;

class Point<T>{
```

```
    private T t1;
    private T t2;
    public T getT1() {
        return t1;
    }
    public void setT1(T t1) {
        this.t1 = t1;
    }
    public T getT2() {
        return t2;
    }
    public void setT2(T t2) {
        this.t2 = t2;
    }
}

public class GenericDemo {
    public static void main(String[] args) {
        //String 类型的
        Point<String> p = new Point<String>();
        p.setT1("2");
        p.setT2("3");
        System.out.println(p.getT1());
        System.out.println(p.getT2());
        //Integer 类型的
        Point<Integer> p2 = new Point<Integer>();
        p2.setT1(23);
        p2.setT2(24);
        System.out.println(p2.getT1());
        System.out.println(p2.getT2());
        //Double 类型的
        Point<Double> p3 = new Point<Double>();
        p3.setT1(23.00);
        p3.setT2(24.00);
        System.out.println(p3.getT1());
        System.out.println(p3.getT2());
//============================
        Set<String> s = new HashSet<String>();//创建一个容器对象,应该在创建的时候就明确是装什么的
        s.add("a");
        //s.add(1);//此时就加不进去了，因为已经限制了容器内参数类型！
        //此时就能保证集合里元素类型一致,
        Set<Integer> treeSet = new  TreeSet<Integer>();
        //规定key只能是String,value是Date
```

```
        Map<String,Date> map = new HashMap<String,Date>();
        // V put(K key, V value)
        Date v = map.put("", new Date());//和上面定义的类型一样
        //V get(Object key)
        Date val = map.get("");
    }
}
```

运行结果

```
2
3
23
24
23.0
24.0
```

**这样的话借助泛型一个类就可以表达多个不同类型的参数！**

要求

消除强制类型的转换,如,使用 Comparable 比较时每次都需要类型强转;

1、没有加上泛型，最初的需要强制类型转换

```
package generic;

import java.util.Set;
import java.util.TreeSet;

class Person implements Comparable{//需要进行排序的类要实现Comparable

    private Integer age;
    public Person(Integer age) {
        super();
        this.age = age;
    }
    @Override
    public int compareTo(Object o) {
        Person p = (Person)o;//强制类型转换
        return this.age.compareTo(p.age);
    }
    public String toString(){
        return this.age.toString();
    }
}
```

```
public class GenericDemo2 {
    public static void main(String[] args) {
        Set set = new TreeSet();
        set.add(new Person(15));
        set.add(new Person(12));
        set.add(new Person(19));
        set.add(new Person(53));
        set.add(new Person(62));
        System.out.println(set);
    }
}
```

**第二步：加上泛型，不再需要强转（因为类型已经固定了）!**

```
package generic;

import java.util.Set;
import java.util.TreeSet;

class Person implements Comparable<Person>{//
    private Integer age;
    public Person(Integer age) {
        super();
        this.age = age;
    }
    @Override
        public int compareTo(Person o) {
            return this.age.compareTo(o.age);//按照什么排序
        }
    public String toString(){
        return this.age.toString();
    }
}

public class GenericDemo2 {
    public static void main(String[] args) {
        Set<Person> set = new TreeSet<Person>();
        set.add(new Person(15));
        set.add(new Person(12));
        set.add(new Person(19));
        set.add(new Person(53));
        set.add(new Person(62));
        System.out.println(set);
    }
}
```

## 5、声明多个泛型类型和通配符

若一个类中多个字段需要不同的泛型声明，则在声明类的时候指定多个泛型类型即可；

格式：

```
public interface IDAO<PK, T> {

    PK add(T t);

    void remove(PK id);

    void update(PK id, T t);

    T get(PK id);
}
```

在进行引用传递的时候泛型类型必须匹配才可以传递，否则编译不通过；
使用 ？ ，表示未知类型的泛型对象：
List<?> 表示未知元素的 List 集合；
这种带通配符的 List 仅表示各种泛型 List 的父类，并不能把元素添加入集合中；
List<?> list = new ArrayList<>(); list.add(1);//ERROR
public void show(List<?> list){}
//表示可接受任意类型的 List 集合

## 6、泛型的上限与下限

设置泛型对象的上限使用 extends,表示参数类型只能是该类型或该类型的子类：
声明对象：类名<? extends 类> 对象名
定义类：类名<泛型标签 extends 类>{}
设置泛型对象的下限使用 super,表示参数类型只能是该类型或该类型的父类：
声明对象：类名<? super 类> 对象名称
定义类：类名<泛型标签 extends 类>{}

```
public static void    show(List<? extends Number> l){

}
public static void    show(List<? super String> l){

}
public static void show(List<? extends Number> l){}
public static void show(List<? super String> l){}
```

**泛型的上限**

```
public static void main(String[] args) {
        Person<Integer> p1 = new Person<>();
        p1.setVal(99);
        Person<Double> p2 = new Person<>();
        p2.setVal(3.14);
        Person<String> p3 = new Person<>();
        p3.setVal("007");
        show(p1);//√
        show(p2);//√
        show(p3);//×
    }
    public static void show(Person<? extends Number> p){//此处限定了Person的参数类型只能是Number或者是其子类，而String并不属于Number。
        System.out.println(p.getVal());
    }
```

**泛型的下限**

```
    public static void main(String[] args) {
        Person<Integer> p1 = new Person<>();
        p1.setVal(99);//Integer
        Person<Double> p2 = new Person<>();
        p2.setVal(3.14);//Double

        Person<String> p3 = new Person<>();
        p3.setVal("007");//String
        Person<Object> p4 = new Person<>();
        p4.setVal(new Object());//Object

        show(p1);//×
        show(p2);//×
        show(p3);//√
        show(p4);//√
    }
    public static void show(Person<? super String> p){
        System.out.println(p.getVal());
    }
```

**很好的例子！**

```
package generic;
```

```java
import java.util.ArrayList;
import java.util.List;

public class GenericDemo3 {
    public static void main(String[] args) {
        //因为show方法是用List<?>通配符接收的，所以可以是任意类型！
        List<String> l1 = new ArrayList<>();//new ArrayList<String>()
        show(l1);
        List<Double> l2 = new ArrayList<>();
        show(l2);
        List<Number> l3 = new ArrayList<>();
        show(l3);
        List<Object> l4 = new ArrayList<>();
        show(l4);
        //使用up方法的话接收类型为Number或者其子类
        //up(l1);//错误，因为up方法接收类型为Number或者其子类，l1（String）不符合！
        up(l2);
        up(l3);
        //使用down方法的话接收类型为Number或者其父类
        //down(l2);error
        down(l3);
        down(l4);
    }
public static void down(List<? super Number> l){

        for (Object object : l) {
            System.out.println(object);
        }
    }
public static void up(List<? extends Number> l){

        for (Object object : l) {
            System.out.println(object);
        }
    }
    public static void show(List<?> l){

        for (Object object : l) {
            System.out.println(object);
        }
    }
}
```

## 7、泛型接口和方法

java5 后，可以声明泛型接口，声明方式和声明泛型类是一样的。
public interface IDAO<T>{}
泛型接口子类有两种方式：
直接在子类后申明泛型；
在子类实现的接口中给出具体的泛型类型
public class DaoImpl<T> implements IDAO<T>{}
public class DaoImpl implements IDAO<String>{}

### 泛型方法

方法中可定义泛型参数，形参的参数类型就是实参的类型。
格式：
**<泛型标签> 返回值类型** 方法名([泛型标签 参数]...)

```
public static <T extends Number> List<T> show(T……t){
    return null;
}
.....main.....{
    //Show(new Object[]{});不可以，因为方法参数类型的限定
    Show(new Number[]{});
    Show(new Integer[]{});
}
```

## 8、泛型的嵌套（map 例子）

```
package july7;
//泛型加Map的输出!

import java.util.Iterator;
import java.util.Map;
import java.util.Map.Entry;
import java.util.Set;
import java.util.TreeMap;


public class Demo20 {
    public static void main(String[] args) {
```

```
        Map<Integer, String> m = new TreeMap<Integer, String>();

        m.put(1, "周冬雨");
        m.put(2, "章子怡");
        m.put(3, "章泽天");

        System.out.println(m);

        //第一种方式，用到了entrySet方法
        Set<Entry<Integer, String>> s = m.entrySet();
        Iterator<Entry<Integer, String>> it = s.iterator();
        while(it.hasNext()){
            Entry<Integer, String> e = it.next();//用到了泛型，这里消除了 强转!
            System.out.println(e.getKey()+"  "+e.getValue());
        }


        //第二种方法，用到了keySet方法
        Set<Integer> set = m.keySet();
        Iterator<Integer> iter = set.iterator();
        while(iter.hasNext()){
            Integer i = iter.next();//直接写到输出语句，出错!
            System.out.println(i+"  "+m.get(i));
        }
    }
}
```

# 九、黑马程序员—IO 操作（1）

第九篇　IO 文件操作（1）

## 1、IO 的概述和 File 方法

IO 流用来处理设备之间的数据传输

Java 对数据的操作是通过流的方式

Java 用于操作流的对象都在 IO 包中

File 类在整个 IO 包中与文件本身有关的操作类，所有的与文件本身有关指的是创建、删除文件等操作。在 java.io 包中的 File 类本身是一个跨平台的文件操作类，所以在操作中要更多的考虑到各个操作系统的区别。

**File 即指文件也指文件夹。**

**File 类构造方法和字段摘要**

static String pathSeparator　路径分隔符，window 下是";"。
static char pathSeparatorChar　路径分隔符，window 下是";"。
static String separator　路径分隔符，window 下是"\"。
static char separatorChar　路径分隔符，window 下是"\"。
File(File parent, String child) 根据 parent 抽象路径名和 child 路径名字符串创建一个新 File 实例。
File(String pathname)　通过将给定路径名字符串转换为抽象路径名来创建一个新 File 实例。
File(String parent, String child) 根据 parent 路径名字符串和 child 路径名字符串创建一个新 File 实例。
File(URI uri)　通过将给定的 file: /URI 转换为一个抽象路径名来创建一个新的 File 实例。

**File 的相关方法**

String getName():返回文件名或路径名（若是路径，返回最后一级子路径名）
String getPath():返回对象对应的路径名
File　getAbsoluteFile():返回绝对路径
String getAbsolutePath():返回对象对应的绝对路径
String getParent():返回文件目录的上一级目录名
boolean renameTo(File newName):重命名此 File 对象对应的文件或目录，若重命名成功返回 true;
boolean exists():判断对象对应的文件或目录是否存在；
boolean canWrite():判断对象对应文件或目录是否可写；
boolean canRead():判断对象对应文件或目录是否可读；
boolean isFile():判断对象是文件，不是目录；
boolean isDirectory()　判断对象的文件是否是一个目录；
boolean isAbsolute() 判断对象对应文件或目录是否为绝对路径名；

boolean createNewFile() 当且仅当不存在，该方法创建一个该 File 对象所指定的新文件，创建成功返回 true。
boolean delete():删除 File 对象所对应的文件或路径；
boolean mkdir() 创建 File 对象所对应的目录，调用该方法的 File 对象必须对应路径，而不是文件。
String[] list():列出 File 对象的所有子文件名和路径名。
File[] listFiles():列出 File 对象的所有子文件和路径。
static File[] listRoots():列出系统所有的根路径；

**我的总结：IO 这一章节最应该记住的关键字：读进来，写进去！**

Eg：

```java
package july7file;

import java.io.File;
import java.io.IOException;

public class Demo1 {
    public static void main(String[] args) throws IOException {

        File f = new File("E:/你好.txt");
        System.out.println(f.createNewFile());
        System.out.println(f.getName());
        System.out.println(f.getParent());
        System.out.println(f.length());
    }
}
```

输出：

```
false
你好.txt
E:\
6905
```

## 2、递归(Recursion)

现在要求输出一个给定目录中的全部文件的路径。
本程序肯定只能依靠递归的操作完成，因为在一个给定的路径下有可能还是文件夹，那么如果是文件夹的话则肯定要继续列出，重复判断。
**递归：程序调用自身的编程技巧**
递归就是在方法里调用自身；
在使用递归时，必须有一个明确的递归结束条件，称为递归出口。

**练习：列出文件夹下所有文件（包含子文件夹内）**

```java
package july7file;

//利用递归遍历输出

import java.io.File;

public class Demo2 {
    public static void main(String[] args) {
        File f = new File("D:/V5");
```

```
        mylist(f);
    }

    public static void mylist(File f) {
        System.out.println(f);// 先输出一下，因为不能确定接受来的文件是否是文件夹!
        if (f.isDirectory()) {
            File[] file = f.listFiles();
            for (File file2 : file) {
                mylist(file2);
            }
        }
    }
}
```

**练习：删除一个目录（注意：要删除目录必须删除目录下的文件和子目录)**

```
package july7file;

import java.io.File;

public class Demo11 {
    public static void main(String[] args) {

        File f = new File("D:/V5");
        deleter(f);

        System.out.println("删除成功 ！");
    }

    public static void deleter(File f){//程序简陋，就没有判断空引用!
        if(f.isFile()){
            f.delete();
        }else if(f.isDirectory()){
            File []file = f.listFiles();
            for (File file2 : file) {
                deleter(file2);//调用自身，递归!
                file2.delete();//删除子文件夹（内部没有文件的时候可以删除），如果这里写上f.delete();那么V5这个文件夹也没有了
            }
        }
    }
}
```

## 3、文件过滤器 java.io.FilenameFilter

File 类里有方法： String[] list(FilenameFilter filter)　　返回一个字符串数组，这些字符串指定此抽象路径名表示的目录中满足指定过滤器的文件和目录。
FilenameFilter(文件过滤器)该接口里包含 accept(File dir,String name)方法，该方法依次对指定 File 的所有子目录，子文件夹进行迭代。
dir - 被找到的文件所在的目录。
name - 文件的名称。
当且仅当该名称应该包含在文件列表中时返回 true；否则返回 false

Eg：

```
package july7file;

//构造过滤器，只输出需要的文件!

import java.io.File;
import java.io.FilenameFilter;

class MyFilter implements FilenameFilter {
    private String ext;

    public MyFilter(String ext) {
        super();
        this.ext = ext;
    }

    @Override
    public boolean accept(File dir, String name) {
        return name.endsWith(ext);// 真正起作用的还是这里的ext
    }
}

public class Demo3 {
    public static void main(String[] args) {

        File f = new File("D:/V5/牛/水牛");

        File[] file = f.listFiles(new MyFilter(".txt"));
        for (File file2 : file) {
            System.out.println(file2);
        }
    }
}
```

## 4、流

数据流是一串连续不断的数据的集合,就像水管里的水流,在水管的一端一点一点地供水,而在水管的另一端看到的是一股连续不断的水流.
数据写入程序可以使一段一段地向数据流管道中写入数据,这些数据段会按先后顺序形成一个长的数据流.

在程序中所有的数据都是以流的方法进行传输和保存的。
Java 的 IO 是实现输入和输出的基础。
Java 把所有传统的流类型(类或抽象类)都放在 java.io 包中，用以实现输入输出功能。
输入和输出是一个相对的概念，我们一般站在程序的角度来分析和处理问题的。
**程序需要数据 --> 读进来 --> 输入**
**程序保存数据 --> 写出去 --> 输出**
水流

**我的总结：最重要的：从程序的角度出发，读进来，写出去！（在储存数据的时候是把数据写出去，这时候数据就储存在了文件里面，在需要调用数据的时候就把数据读进来，这样数据就又到了程序中！）**

**流的分类（面试常考）**

从不同角度分类：
按**流动方向**的不同可以分为**输入流和输出流；**
按**处理数据的单位**不同分为**字节流和字符流；**
按**功能的不同**可分为**节点流和处理流；**
**节点流：直接操作目标设备，例如：磁盘或一块内存区域。**
**处理流：通过操作节点流,从而间接完成输入或输出功能的流。处理流是的存在是建立在一个已经存在的输入流或输出流的基础之上的。**

**所有流都继承于以下四种抽象流类型的某一种：（抽象流）**

| | 字节流 | 字符流 |
|---|---|---|
| 输入流 | InputStream | Reader |
| 输出流 | OutputStream | Writer |

## 5、操作流的步骤（重点）

**File 类本身是与文件操作有关，但是如果要想操作内容则必须使用字节流或字符流完成，**但是不管是使用何种的输入输出流，其基本的操作原理是一样的（以文件流为准）：

**一、使用 File 类找到一个文件对象，得到 IO 操作的源或目标**

**二、通过字节流或字符流的子类创建对象，(得到 IO 操作的通道)**

**三、进行读或写的操作，(IO 操作)**

**四、关闭输入/输出，(打完收工，注意节约资源，关掉)**

**由于流的操作属于资源操作，所以在操作的最后一定要关闭以释放资源。**

其实上面的流操作步骤可以联系生活中的例子:比如想把水井里的水弄到家里的大水缸去,怎么搞呢?

**1.找到水井在哪里;2.找根管子一头接在水井里,一头接在家里的大水缸里;3.打开管子上的龙头,放水;4.水放满了,关掉水龙头.**

计算机访问外部设备,要比直接访问内存慢得多,若我们每一次 write 方法调用都是直接写到外部设备(比如磁盘上的一个文件),CPU 就要花费更多的时间去等待外部设备;我们可以开辟一个内存**缓冲区**,程序每一次的 write 方法都是写到这个内存缓冲区中,只有这个缓冲区装满了之后,系统才将这个缓冲区的内容一次集中写到外部设备.

**我的总结：**

**好处:1.有效提高了 CPU 的使用率;2.write 方法并没有马上真正写入到外部设备,我们还有机会回滚部分写入的数据;**

Eg:

```
package july7file;
//构建输入流，读进来，输出到控制台！

import java.io.FileInputStream;
import java.io.InputStream;

public class Demo4 {
    public static void main(String[] args) throws Exception {
        //第一步：创建源！
        String filename = "6.4";//这个文件是在工作空间里面，没有后缀名！

        //第二步：创建管道！
        InputStream ips = new FileInputStream(filename);

        //第三步：操作！

        byte []buff = new byte[1024];
        int len;//定义缓冲区
        while((len = ips.read(buff)) != -1){
            System.out.println(new String(buff,0,buff.length));//输出到控制台！此时的输出流就是打印流！
```

```
    System.out.println("==========================================
");//打印下，看哪里在1024。1024的地方被隔开了
        }

        //第四步：关闭资源(字符流必须关闭资源，因为它中间有缓冲区！对于字节流可以不用关闭，但是还是建议写上，习惯！)
        ips.close();
    }
}
```

输出：就将文件 6.4 中的数据打印到了控制台！

## 6、字节流和字符流

二者仅仅是**操作单位**不一样。
InputStream 和 Reader 是**所有输入流的基类**，他们都是抽象类，本身不能创建实例，但是他们是所有输入流的模板。
一般来说处理字符或字符串时使用字符流，处理字节或二进制对象时应使用字节流；
备注：字符流必须关闭资源，因为它中间有缓冲区！而字节流不需要！但是一般都会（最后）关闭资源！

**字节流**
**字节流主要是操作 byte(字节)的类型数据:**
**字节输出流：OutputStream**
**字节输入流：InputStream**

**字符流**
Java 中的字符是 Unicode 编码,是双字节的,1 个字符 等于 2 个字节;
**使用字节来处理字符文本就不太方便了,此时可以考虑使用字符流;**
字符流主要是操作 **char** 的类型数据:
**字符输出流：Writer**
**字符输入流：Reader**

**字节流和字符流的区别**

字节流和字符流在使用上的代码结构都是非常类似的，但是其内部本身也是有区别的，因为**在进行字符流操作的时候会使用到缓冲区（内存中），而字节流操作的时候是不会使用到缓冲区的。**
**在输出的时候，`OutputStream` 类即使最后没有关闭内容也可以输出。但是如果是 `Writer` 的话，则如果不关闭，最后一条内容是无法输出的，因为所有的内容都是保存在了缓冲区**

之中，每当调用了 close() 方法就意味着清空缓冲区了。那么可以证明字符流确实使用了缓冲区：
字节流：程序 → 文件
字符流：程序 → 缓冲区（内存中） → 文件

如果现在字符流即使不关闭也可以完成输出的话，则必须强制性清空缓冲区：
方法：public void flush() throws IOException

我的总结：
两者相比，肯定使用字节流更加的方便，而且在程序中像图片、MP3 等都是采用字节的方式的保存，那么肯定字节流会比字符流使用的更广泛。
但是需要说明的是，但是如果要是想操作中文的话，字符流肯定是最好使的。（字节流的话可能会出现乱码（一个汉字分成了两份）!）

Eg：

```
package july7file;
//字符流读出来，这时候就不会出现乱码的情况，在进行文字操作的时候最好使用字符流！

import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.io.Reader;

public class Demo6 {
    public static void main(String[] args) {

        File src = new File("6.4");
        read(src);
    }

    public static void read(File src){
        Reader r = null;
        try {
            r = new FileReader(src);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }

        char []c = new char[1024];
        int len;
        try {
            while((len = r.read(c)) != -1){
                System.out.println(new String(c,0,c.length));//打印到
```

控制台

```
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        try {
            r.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

## 7、文件拷贝

需求：源和目标！

那么我们需要源文件和目标文件！

构建管道的时候就需要两个：输出流和输入流管道！

Eg:

```
package july7file;

//java7开始的自动关闭资源

import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

public class Demo8 {
    public static void main(String[] args) throws IOException {

        File src = new File("E:/自荐信.doc");
        File tar = new File("E:/自荐信1.doc");

        copy(src, tar);
        System.out.println("Well done ! ");
    }

    public static void copy(File src, File tar) throws IOException {
```

```
        try (InputStream is = new FileInputStream(src);
                OutputStream os = new FileOutputStream(tar);) {
            byte[] b = new byte[1024];
            int len;
            while ((len = is.read(b)) != -1) {
                os.write(b);
            }

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

题目：复制图片！

```
package july7file;
/**
 * 文件的复制！对于本题而言，因为是图片，所以要想读出来，必须使用字节流！
 * 字符流必须关闭资源，而字节流可以不关闭资源！但是还是建议全部的关闭，因为也不会出错，这是关闭资源的习惯！
 * 另外：最常用的是字节流，因为字节流在内存中不需要缓冲区，图片，mp3 等都是字节流！但是对于文字的话还是字符流比较好；
 * 因为字符流可以避免在字节流操作文字时出现的乱码现象（正好读取到了自定义缓冲区的分割处）；
 */

import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.io.OutputStream;


public class Demo7 {
    public static void main(String[] args) throws Exception {
        File src = new File("D:/java.jpg");

        File tar = new File("D:/meinv.jpg");

        copy(src,tar);
        System.out.println("复制完成！");
    }
    public static void copy(File src,File tar) throws Exception{
/*      Reader r = new FileReader(src);
```

```
        Writer w = new FileWriter(tar);*/
        /*if(!src.exists()){
            throw new Exception("对不起，源文件不存在！");
        }*/
        InputStream in = new FileInputStream(src);
        OutputStream os = new FileOutputStream(tar);

        byte []c = new byte[1024];
        int len;
        while((len = in.read(c)) != -1){
            os.write(c);
        }

    /*  w.close();
        r.close();*/
    }
}
```

**我的总结：对于图片的复制，可以使用字符流，但是这样的话文件可以复制成功但是无法读取！**

## 8、字节→字符转换流

OutputStreamWriter:把字节输出流对象转成字符输出流对象
InputStreamReader:把字节输入流对象转成字符输入流对象
FileWriter 和 FileReader 分别是 OutputStreamWriter 和 InputStreamReader 的直接子类，而不是 Writer 和 Reader 的直接子类，区别于 FileInputStream 和 InputStream。

**我的总结：无论使用字节流还是字符流实际上在内存中最终都是通过字节的形式来操作流的。**
**所以并没有字符流转换字节流。**

Eg:

```
//构建一个字节输出流对象
OutputStream out = new FileOutputStream("");
//把字节输出流转成字符输出流
Writer w    = new OutputStreamWriter(out);
//然后的操作和使用字符输出流的操作一样
```

---

//构建一个字节输入流对象
InputStream is = new FileInputStream("");
//把字节输入流转成字符输入流
Reader r = new InputStreamReader(is);
//然后的操作和使用字符输入流的操作一样

## 9、自动关闭资源的 try 语句

Java 7 简化资源清理(try-with-resources)自动关闭资源的 try 语句

**自动关闭资源格式：**
**try( )//此处多了圆括号，（）圆括号内写打开资源的代码，在这里创建的对象必须实现 Autocloseable 接口**
**{**
    **IO 操作**
**}**
**catch（）{**
    **处理异常的代码**
**}**

Eg: **package** july7file;

```
//java7开始的自动关闭资源

import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

public class Demo8 {
    public static void main(String[] args) throws IOException {

        File src = new File("E:/自荐信.doc");
        File tar = new File("E:/自荐信1.doc");

        copy(src, tar);
        System.out.println("Well done ! ");
    }

    public static void copy(File src, File tar) throws IOException {
```

```
        try (InputStream is = new FileInputStream(src);
                OutputStream os = new FileOutputStream(tar);) //圆括号内写打开资源的操作
            {
            byte[] b = new byte[1024];
            int len;
            while ((len = is.read(b)) != -1) {
                os.write(b);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

# 十、IO 文件操作（2）

## 1、内存操作流

操作内存流的时候(从读取出来,注意一定要把**真正的数据**用 **toByteArray 或者 toCharArray 将数据读出来）**

**之前的文件操作流是以文件的输入输出为主的，当输出的位置变成了内存，那么就称为内存操作流。此时要使用内存流完成内存的输入和输出操作。**

**如果程序运行过程中要产生一些临时文件，可采用虚拟文件方式实现；**

**直接操作磁盘的文件很耗性能,使用内存流可以提升性能**;jdk 里提供了内存流可实现类似于内存虚拟文件的功能。

**ByteArrayInputStream：将内容写到内存中　　CharArrayReader**

**ByteArrayOutputStream：将内存中的数据写出　　CharArrayWriter**

ByteArrayInputStream：构造方法：

public ByteArrayInputStream(byte[] buf):全部内容

public ByteArrayInputStream(byte[] buf,int offset,int length)：指定范围的内容

ByteArrayOutputStream:

public ByteArrayOutputStream()

**我的总结：内存流操作：先把数据写到内存中去，然后再从内存中读取出来！提升了性能！**

例子

//此处用的是内存字节流（写入加输出）

```
package june6D;
```

```
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;

public class ByteArrayDemo7 {
    public static void main(String[] args) throws IOException {
        String s = "java is a good language";
        ByteArrayOutputStream bos = new ByteArrayOutputStream();//输
出流
        bos.write(s.getBytes());
        // 已经把信息写到了内存中

        byte[] bys = bos.toByteArray();// 得到真正的数据
        ByteArrayInputStream bis = new ByteArrayInputStream(bys);//
输入流，需要源。
        byte[] b = new byte[1024];
        int len;
        while ((len = bis.read(b)) != -1) {
            String data = new String(b, 0, len);
            System.out.println(data);
        }
    }
}
```

```
package june6D;

import java.io.CharArrayReader;
import java.io.CharArrayWriter;

public class CharArrayDemo8 {

    public static void main(String[] args) throws Exception {
        // 用内存字符流先把数据保存到内存中，然后从内存中取出数据
        String s = "河南城建学院";
        CharArrayWriter cw = new CharArrayWriter();
        cw.write(s);// 数据写到了内存中
        char[] ch = cw.toCharArray();
        CharArrayReader cr = new CharArrayReader(ch);
        char[] b = new char[1024];
        int len;
        while ((len = cr.read(b)) != -1) {
            String data = new String(b, 0, len);
```

```
            System.out.println(data);
        }
    }
}
```

## 2、打印流

（只有两个，PrintWriter 和 PrintStream）

思考：如果现在要想完成一个**字符串或者是 boolean 型或者是字符型的数据输出**使用 OutputStream 是否方便？

肯定是不方便的，因为 OutputStream 中只能操作字节数据，所以其他的数据类型很难操作，**那么在 Java 的 IO 包中为了解决这种问题增加了两种类：PrintStream、PrintWriter。**

打印流有非常好的打印功能，可以打印**任何的数据类型**。如，整数，小数，字符串等。

观察 PrintStream 类的构造：

public PrintStream(File file) throws FileNotFoundException

public PrintStream(OutputStream out)

虽然 PrintStream 是 OutputStream 的子类，但是在实例化的时候依然需要一个 OutputStream 的对象。

**PrintWriter 和 PrintStream 都属于输出流，分别针对字符和字节。**

PrintWriter 和 PrintStream 重载的 print()和 println()用于多种数据类型的输出。

**print()里的参数不能为空；println()可以**

PrintWriter 和 PrintStream 输出操作不抛出异常

**PrintStream 调用 println 方法有自动 flush 功能;**

Eg:

```
package june6D;

import java.io.FileWriter;
import java.io.PrintStream;
import java.io.PrintWriter;

public class PrintDemo9 {
    public static void main(String[] args) throws Exception {

        PrintStream ps = new PrintStream("out.txt");
        // ps.write(12);
        ps.println(10086);
        ps.println(false);
        // ps.print();针对print而言，不可以出现这样的（无参）
        ps.println();// 此时就可以
```

```
        //备注：System.out.println();想当于ps = System.out;

        ps = System.out;
        ps.println("您好！");// 控制台操作，注意上一句
        // 字符打印流
        PrintWriter pr = new PrintWriter("out2.txt");
        // PrintWriter(OutputStream out, boolean autoFlush) 通过现有的 OutputStream，创建新的 PrintWriter。（构造方法）

        pr = new PrintWriter(new FileWriter("out2.txt"), true);// 自动刷新,否则的话需要关闭资源!
        // 与PrintStream不同，若PrintWriter使用了自动刷新方法，那么必须调用println，print，format这些方法的其中一个才可以实现操作
        pr.println("呼哈哈哈哈");
        pr.println(false);

        pr = new PrintWriter(System.out, true);// 打印在控制台上
        pr.println(false);
        pr.println("河南城建学院");
        pr.println(376430645);
        // pr.close();//因为使用了自动刷新。
    }
}
```

## 3、格式化输出

Java5 后，PrintStream 类多了 printf()方法用于格式化输出操作。但是格式化输出的时候必须指定输出数据的类型：

**（构造方法）**
**PrintStream format(String fo, Object... args) 使用指定格式字符串和参数将格式化字符串写入此输出流中。**

| 字符 | 描述 |
| --- | --- |
| %s | 表示内容是字符串 |
| %d | 表示内容是整数 |
| %f | 表示内容是小数 |
| %c | 表示内容是字符 |

**备注：当然你也可以全部使用“%s”来表示所有的数据类型！**

格式：
需要格式 % 占位符

Eg：

```
package july7file;

import java.io.FileWriter;
import java.io.PrintStream;
import java.io.PrintWriter;

public class Demo9 {
    public static void main(String[] args) throws Exception {

        PrintStream ps = new PrintStream("E:/name.txt");

        ps.write("123456".getBytes());

        ps.println(false);
        ps.print(false);

        PrintWriter pw = new PrintWriter(new
FileWriter("E:/name1.txt"),true);

        pw.println("我们的未来！");//write不可以

        pw = new PrintWriter(System.out,true);
        pw.println("我们的未来！");//打印在控制台上

        String name = "刘昭";
        int age = 13;
        char score = 'A';

        String format = "姓名=%s,年龄=%d,成绩=%c";

        System.out.printf(format,name,age,score);

        int i = System.in.read();//流的重定向
        System.out.println((char)i);
    }
}
```

## 4、标准流

**标准输入流：System.in 默认表示的是键盘录入**
**标准输出流：System.out 默认表示的是屏幕输出**

Eg：

```
package june6D;

import java.io.FileInputStream;
import java.io.IOException;
import java.io.PrintStream;

public class SyetemDemo {
    public static void main(String[] args) throws IOException {
        /**
         *  标准流：标准输入流：System.in 默认表示的是键盘录入
            标准输出流：System.out 默认表示的是屏幕输出
         *
         * 若现在我输出不想打印在屏幕上,怎么办?
         * 若现在我不想通过键盘来录入数据,有怎么办?
         *
         * 流的重定向：static void setOut(PrintStream out) 重新分配"标准"输出流。 static void
         * setIn(InputStream in) 重新分配"标准"输入流。
         */
        System.out.println("123456789");//最普通的打印
        System.err.println("123652");// 同样是一种输出流,打印出的是123652

        /**
         * static void setOut(PrintStream out) 重新分配"标准"输出流。
         */

        // 输出流就被重新定义到了文件，而不是屏幕
        System.setOut(new PrintStream("68Demo.txt"));// 只有在定义过后才生效，所以上面的还是会输出到控制台
        System.out.println("AAAA");
        System.out.println("BBBB");//就输出了在文件里

        int i = System.in.read();// 接受从键盘输入的数据，写在最前面（没有任何重新分配）才有效
        System.out.println("i所对应的是: " + (char) i);// 把它转换为字符型
```

```
        System.out.println(i);
        /**
         * 重新分配，由标准键盘输入变为由文件输入
         */
        System.setIn(new FileInputStream("copy6.4"));//备注：打印的是文件copy6.4中的数据，而且打印在文件68Demo.txt中。
        byte[] buff = new byte[1024];
        int len = 0;
        while ((len = System.in.read(buff)) != -1) {
            System.out.println(new String(buff, 0, len));// 此时不会打印在控制台上，因为上面已经重新定义了由打印到屏幕上转换为打印到文件里
        }
    }
}
```

## 5、Scanner(简单文本扫描器)

Scanner(File source) 构造一个新的 Scanner，它生成的值是从指定文件扫描的。
备注：实现了Iterable接口

```
package june6D;

import java.io.File;
import java.io.PrintStream;
import java.util.Scanner;

public class ScannerDemo18 {
    public static void main(String[] args) throws Exception {

        // 参照api
        Scanner sc = new Scanner(System.in);// 从键盘输入
        // int i = System.in.read();
        /*  连着进行了三次，没有完成的话不停止运行
            System.out.println(sc.next());
            System.out.println(sc.next());
         *  System.out.println(sc.next());
         */
        System.setOut(new PrintStream("Scanner.txt"));// 流的重定向 打印到哪里）
        sc = new Scanner(new File("copy6.4"));// 扫描位置
        int line = 1;
        while (sc.hasNextLine()) {// 是否有下一行 //一行一行的读取，这样的话格式好看
```

```
            System.out.println(line + " " + sc.nextLine());// 读取下一
行
            line++;
        }
    }
}
```

例子：
猜数字游戏：
1．系统随机生成一个数字[1,100];
2．从键盘录入一个数字,[1,100]
3． 判断输入的数字和随机数比较：
    随机数 > 输入数:你输入太小了
    随机数 < 输入数:输入太大了
    随机数 = 输入数: 恭喜哦

思考：先想怎么生成一个随机数；然后怎么在键盘中录入一个随机数，第三步就是比较了；但是注意的是：需要确保输入的数据为数字，而且有范围的限制！

```
package july7file;

import java.util.Random;
import java.util.Scanner;

public class Demo12 {
    public static void main(String[] args) {

        guess();
    }

    public static void guess(){
        int i = new Random().nextInt(100)+1;

        System.out.println(i);
        System.out.println("请输入随机数来匹配，您有五次机会！");
        Scanner sc = new Scanner(System.in);//键盘录入

        for (int j = 0; j < 5; j++) {
            String s = sc.nextLine();

            if(!s.matches("\\d+")){
                System.out.println("请确认您输入的是数字");
                break;
```

```
            }
            Integer in = new Integer(s);
            if(in > 100 | in < 0){
                System.out.println("请确认您输入的数字在0到100之间！");
            }

            switch (in.compareTo(i)) {
            case 1:
                System.out.println("您输入的数字过大！");
                System.out.println("请输入:");
                break;

            case -1:
                System.out.println("您输入的数字过小！");
                System.out.println("请输入:");
                break;

            default:
                System.out.println("恭喜您，您输入的数字正好匹配！");
                return;
            }
        }
    }
}
```

## 6、缓冲流

缓冲流要“套接”在相应的节点流之上，对读写的数据提供了缓冲的功能，提高了读写效率，同时增加了一些新的方法。

四种缓冲流

BufferedReader(Reader in)

BufferedReader(Reader in,int sz)//sz 表示自定义缓冲区大小

BufferedWriter(Writer out)

BufferedWriter(Writer out,int sz)

BufferedInputStream(InputStream in)

BufferedInputStream(InputStream in,int sz)

BufferedOutputStream(OutputStream out)

BufferedOutputStream(OutputStream out,int sz)

BufferedReader 提供 readLine 方法用于读取一行字符串。

BufferedWriter 提供了 newLine 方法用于写入一个行分隔符。等价于//.writer("\r\n");

对于输出的缓冲流，写出的数据会先在内存中缓冲，使用 flush 方法将会使内存中的数据立刻写出。

Eg:

```
package june6D;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

//用缓冲流，性能相对高些
public class BufferedInputStreamDemo22 {
    public static void main(String[] args) throws IOException {
        /*
         * BufferedInputStream bis = new BufferedInputStream(new
         * FileInputStream("68.txt")); BufferedOutputStream bos = new
         * BufferedOutputStream(new FileOutputStream("buffer.txt"));
         *
         * int len = 0;
         * while((len = bis.read()) != -1){
         * bos.write(len);
         * }
         * bos.close();
         * bis.close();
         */

        try (
    BufferedReader br = new BufferedReader(new FileReader("68.txt"));
    BufferedWriter bw = new BufferedWriter(new FileWriter(
                        "bufferWriter.txt")))
{//java7新特性，自动关闭资源
            String Line = null;
            while ((Line = br.readLine()) != null) {
                bw.write(Line);
                bw.newLine();//此时必须加上换行操作，注意这是个新用法（方法）
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

**我的总结：在后续的学习中，缓冲流用到的会很多，因为效率高！**

## 7、合并流（SequenceInputStream）

需要两个源文件，还有输出的目标文件

SequenceInputStream：
将两个文件的内容合并成一个文件
该类提供的方法：
SequenceInputStream(InputStream s1, InputStream s2) ：根据两个字节输入流对象来创建合并流对象。

备注：谁放在前面，谁就先打印出来

Eg：

```
package june6D;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.io.SequenceInputStream;

//和并两个文件的合并流
public class SequenceInputStreamDemo24 {
    public static void main(String[] args) throws IOException {

        SequenceInputStream si = new SequenceInputStream(
                new FileInputStream("6.4"),
                new FileInputStream("hello.java"));

        OutputStream os = new FileOutputStream("sequence.txt");
        int len;
        byte []b = new byte[1024];
        while((len = si.read(b)) != -1){
            os.write(b, 0, len);
        }
    }
}
```

# 十一、黑马程序员—网络编程

第 11 天：网络编程（1）

## 1、什么是网络编程？

网络编程的本质是两个设备之间的数据交换，当然，在计算机网络中，设备主要指计算机。数据传递本身没有多大的难度，不就是把一个设备中的数据发送给两外一个设备，然后接受另外一个设备反馈的数据。现在的网络编程基本上都是基于请求/响应方式的，也就是一个设备发送请求数据给另外一个，然后接收另一个设备的反馈。在网络编程中，发起连接程序，也就是发送第一次请求的程序，被称作客户端(Client)，等待其他程序连接的程序被称作服务器(Server)。客户端程序可以在需要的时候启动，而服务器为了能够时刻相应连接，则需要一直启动。

例如以打电话为例，首先拨号的人类似于客户端，接听电话的人必须保持电话畅通类似于服务器。连接一旦建立以后，就客户端和服务器端就可以进行数据传递了，而且两者的身份是等价的。在一些程序中，程序既有客户端功能也有服务器端功能，最常见的软件就是 BT、emule 这类软件了。

网络模型

OSI 参考模型

TCP/IP 参考模型


## 2、网络通讯要素

IP 地址

端口号

传输协议

IP 地址:InetAddress

网络中设备的标识

不易记忆，可用主机名

本地回环地址：127.0.0.1　主机名：localhost

端口号

用于标识进程的逻辑地址，不同进程的标识

有效端口：0~65535，其中 0~1024 系统使用或保留端口。

备注：不是所谓的物理端口！

传输协议

通讯的规则

常见协议：TCP，UDP

## 3、TCP 和 UDP 的联系和用途

一.区别

二者都是有用的和常用的，如果纯粹从概念上区分二者就比较费解了，我们直接从功能上进行区分，简单明了：

这两种传输协议也就是合于适配不同的业务和不同的硬件终端。

在使用中，类似于图像、声音等对可靠性要求没有那么高的业务可以用 UDP，他们不需要准确存储对准确性无要求但要求速度快。

类似于文本、程序、文件等要求可靠的数据最好就用 TCP，但会牺牲一些速度。

对系统资源的要求：CP 较多，UDP 少。

程序结构：UDP 程序结构较简单，TCP 复杂。

流模式与数据报模式:TCP 保证数据正确性，UDP 可能丢包; TCP 保证数据顺序，UDP 不保证

二.用途

TCP 是面向连接的，有比较高的可靠性，一些要求比较高的服务一般使用这个协议，如 FTP、Telnet、SMTP、HTTP、POP3 等，而 UDP 是面向无连接的，使用这个协议的常见服务有 DNS、SNMP、QQ 等。对于 QQ 必须另外说明一下，QQ2003 以前是只使用 UDP 协议的，其服务器 使用 8000 端口，侦听是否有信息传来，客户端使用 4000 端口，向外发送信息(这也就不难理解在一般的显 IP 的 QQ 版本中显示好友的 IP 地址信息中端口 常为 4000 或其后续端口的原因了)，即 QQ 程序既接受服务又提供服务，在以后的 QQ 版本中也支持使用 TCP 协议了。

Udp 是一种面向无连接的通信协议，该协议使得数据传输的速度得到大幅度的提高。视频聊天语音聊天基本都是用 UPD 协议。

**我的总结：**

**一、UDP:**

**1、将数据源和目的地封装到数据包中，不需要建立连接**

**2、每个数据包的大小限制在 64k 以内**

**3、因无连接，是不可靠协议**
**4、不需要建立连接，速度快**

**例子：聊天、对讲机就是 UDP 的，面向无连接（不管在不在，知不知道，只管发送，求速度），丢数据也不管。速度快。数据被分成包**

**二、TCP:**

**1、建立连接，形成传输数据的通道**
**2、在连接中进行大量数据的传输**
**3、通过三次握手完成连接、是可靠协议**
**4、必须建立连接，效率会稍低**

**例子：电话通话，必须连接，对方同意才可以发送数据（不然就等待），不能丢失数据。**

## 4、InetAddress 与 Socket

InetAddress：**构造方法私有，不能直接创建对象。**
InetAddress getByName(String host):在给定主机名的情况下确定主机的 ip 地址。
InetAddress getLocalHost()：返回本地主机。
InetAddress[] getAllByName(String host)
ip.getHostAddress(),
ip.getHostName()

Eg:

```
package july76net;

import java.net.InetAddress;

public class Demo1 {
    public static void main(String[] args) throws Exception {

        InetAddress i = InetAddress.getLocalHost();
        System.out.println(i);

        i = InetAddress.getByName("www.baidu.com");
        System.out.println(i);
        System.out.println(i.getHostAddress());

        System.out.println(i.getHostName());
    }
```

```
}
```

输出：

```
XP-201304252326/10.6.147.2
www.baidu.com/61.135.169.105
61.135.169.105
www.baidu.com
```

**Socket**

Socket 就是为网络服务提供的一种机制。

通信的两端都有 Socket。

网络通信其实就是 Socket 间的通信。

数据在两个 Socket 间通过 IO 传输。

## 5、UDP 传输

①：只要是网络传输，必须有 socket 。

②：数据一定要封装到数据包中，数据包中包括目的地址、端口、数据等信息。

直接操作 udp 不可能，对于 java 语言应该将 udp 封装成对象，易于我们的使用，这个对象就是 **DatagramSocket**. 封装了 udp 传输协议的 socket 对象。

因为数据包中包含的信息较多，为了操作这些信息方便，也一样会将其封装成对象。这个数据包对象就是：DatagramPacket.通过这个对象中的方法，就可以获取到数据包中的各种信息。

DatagramSocket 具备**发送和接受**功能，在进行 udp 传输时，需要明确一个是发送端，一个是接收端。

**udp 的发送端：**

**①：建立 udp 的 socket 服务，创建对象时如果没有明确端口，系统会自动分配一个未被使用的端口。**

**②：明确要发送的具体数据。**

**③：将数据封装成了数据包。**

**④：用 socket 服务的 send 方法将数据包发送出去。**

**⑤：关闭资源。**

**udp 的接收端：**

**①：创建 udp 的 socket 服务，必须要明确一个端口，作用在于，只有发送到这个端口的数据才是这个接收端可以处理的数据。**

**②：定义数据包，用于存储接收到数据。**

**③：通过 socket 服务的接收方法将收到的数据存储到数据包中。**

**④：通过数据包的方法获取数据包中的具体数据内容，比如 ip、端口、数据等等。**

⑤：关闭资源。

Eg：
发送端（客户端）

```
import java.net.*;
class  UdpSend{
        public static void main(String[] args)throws Exception {
                // 1，建立 udp 的 socket 服务。
                DatagramSocket ds = new DatagramSocket(8888);//指定发送端口，这个可以不指定，系统会随机分配。
                // 2，明确要发送的具体数据。
                String text = "udp 传输演示  哥们来了";
                byte[] buf = text.getBytes();
                // 3，将数据封装成了数据包。
                DatagramPacket dp = new DatagramPacket(buf,buf.length,InetAddress.getByName("10.1.31.127"),10000);
                // 4，用 socket 服务的 send 方法将数据包发送出去。
                ds.send(dp);
                // 5，关闭资源。
                ds.close();
        }
}
```

接收端（服务器端）

```
import java.net.*;
class UdpRece {
        public static void main(String[] args) throws Exception{
                // 1，创建 udp 的 socket 服务。
                DatagramSocket ds = new DatagramSocket(10000);//必须指定，并且和上面的端口号一样！
                // 2，定义数据包，用于存储接收到数据。先定义字节数组，数据包会把数据存储到字节数组中。
                byte[] buf = new byte[1024];
                DatagramPacket dp = new DatagramPacket(buf,buf.length);
                // 3，通过 socket 服务的接收方法将收到的数据存储到数据包中。
                ds.receive(dp);//该方法是阻塞式方法。
                // 4，通过数据包的方法获取数据包中的具体数据内容，比如 ip，端口，数据等等。
                String ip = dp.getAddress().getHostAddress();
                int port = dp.getPort();
                String text = new String(dp.getData(),0,dp.getLength());// 将字节数组中的有效部分转成字符串。
```

```
                System.out.println(ip+":"+port+"--"+text);
                // 5，关闭资源。
                ds.close();
        }
}
```

练习：

通过键盘录入获取要发送的信息。

将发送和接收分别封装到两个线程中。

```
package july76net;
//一个聊天的例子，利用UDP传输协议

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;

//客户端，发送端
class Send implements Runnable {
    private DatagramSocket ds;

    public Send(DatagramSocket ds) {
        super();
        this.ds = ds;
    }

    @Override
    public void run() {
        try {
            BufferedReader     br     =     new     BufferedReader(new
InputStreamReader(
                    System.in));//数据源是键盘录入
            String line;
            while ((line = br.readLine()) != null) {
                byte[] buf = line.getBytes();
                DatagramPacket dp = new DatagramPacket(buf, buf.length,
                        InetAddress.getByName("localhost"), 10225);

                ds.send(dp);
            }
```

```
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

// 服务器端，接收端
class Rece implements Runnable {
    private DatagramSocket ds;

    public Rece(DatagramSocket ds) {
        super();
        this.ds = ds;
    }

    @Override
    public void run() {
        try {
            while (true) {
                byte[] buf = new byte[1024];

                DatagramPacket dp = new DatagramPacket(buf, 0,
buf.length);
                ds.receive(dp);

                String ip = dp.getAddress().getHostAddress();
                String data = new String(dp.getData(), 0,
dp.getLength());

                System.out.println(ip + "    " + data);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

public class Demo6 {
    public static void main(String[] args) throws Exception {
        DatagramSocket sendDs = new DatagramSocket();
        DatagramSocket receDs = new DatagramSocket(10225);
        new Thread(new Send(sendDs)).start();
        new Thread(new Rece(receDs)).start();
    }
```

```
}
```

输出：

```
你好
127.0.0.1      你好
你好
127.0.0.1      你好
```

## 6、TCP 传输

两个端点的建立连接后会有一个传输数据的通道，这通道称为流，而且是建立在网络基础上的流，称之为 socket 流。**该流中既有读取，也有写入。**

tcp 的两个端点：一个是客户端，一个是服务端。
客户端：对应的对象，Socket
服务端：对应的对象，ServerSocket

TCP 客户端：

**①：建立 tcp 的 socket 服务，最好明确具体的地址和端口。这个对象在创建时，就已经可以对指定 ip 和端口进行连接(三次握手)。**

**②：如果连接成功，就意味着通道建立了，socket 流就已经产生了。只要获取到 socket 流中的读取流和写入流即可，只要通过 getInputStream 和 getOutputStream 就可以获取两个流对象。**

**③：关闭资源。**

```
import java.net.*;
import java.io.*;
//需求：客户端给服务器端发送一个数据。
class   TcpClient{
        public static void main(String[] args) throws Exception{
                Socket s = new Socket("10.1.31.69",10002);
                OutputStream out = s.getOutputStream();//获取了 socket 流中的输出流对象。
                out.write("tcp 演示，哥们又来了!".getBytes());
                s.close();
    }
}
```

TCP 服务端：

**①：创建服务端 socket 服务，并监听一个端口。**

**②：服务端为了给客户端提供服务，获取客户端的内容，可以通过 accept 方法获取连接过来的客户端对象。**

**③：可以通过获取到的 socket 对象中的 socket 流和具体的客户端进行通讯。**
**④：如果通讯结束，关闭资源。注意：要先关客户端，再关服务端。**

**我的总结：对于 UDP 和 TCP，既可以定义输出流也可以创建输入流，具体情况根据需要构建；比如：我们需要客户端给服务器端发送数据，服务器端再给客户端反馈数据；那么就要在客户端和服务器端分别多加一个输入流和输出流！否则，发不出去，收不到！**

```
class   TcpServer{
        public static void main(String[] args) throws Exception{
                ServerSocket ss = new ServerSocket(10002);//建立服务端的 socket 服务
                Socket s = ss.accept();//获取客户端对象
                String ip = s.getInetAddress().getHostAddress();
                System.out.println(ip+".....connected");//打印下作为连接上的标志

                // 可以通过获取到的 socket 对象中的 socket 流和具体的客户端进行通讯。
                InputStream in = s.getInputStream();//读取客户端的数据，使用客户端对象的 socket 读取流
                byte[] buf = new byte[1024];
                int len = in.read(buf);
                String text = new String(buf,0,len);
                System.out.println(text);
                // 如果通讯结束，关闭资源。注意：要先关客户端，在关服务端。
                s.close();
                ss.close();
        }
}
```

备注:这个例子只是单方面的输入！

Eg：双向对话

客户端:

```
package july76net;
//TCP双向对话

import java.io.InputStream;
import java.io.OutputStream;
import java.net.Socket;

public class Demo10 {
    public static void main(String[] args) throws Exception {
        Socket s = new Socket("localhost",10036);

        OutputStream out = s.getOutputStream();
```

```
        out.write("你好，服务器！我是刘昭！".getBytes());
        s.shutdownOutput();//注意!!! 关闭标签
        InputStream is = s.getInputStream();
        byte []buf = new byte[1024];
        int len = is.read(buf);
            System.out.println(new String(buf,0,len));
        s.close();
    }
}
```

服务器端

```
package july76net;
//TCP双向输入输出

import java.io.InputStream;
import java.io.OutputStream;
import java.net.ServerSocket;
import java.net.Socket;

public class Demo11 {
    public static void main(String[] args) throws Exception {
        ServerSocket ss = new ServerSocket(10036);
        Socket s = ss.accept();

        String ip = s.getInetAddress().getHostAddress();
        System.out.println(ip+".........connected!");

        InputStream in = s.getInputStream();
        byte[] buf = new byte[1024];
        /*int len = in.read(buf);
        System.out.println(new String(buf,0,len));*/
        int len;
        while((len = in.read(buf)) != -1){
            System.out.println(new String(buf,0,len));
        }
        OutputStream os = s.getOutputStream();

        os.write("刘昭你好！我是服务器！".getBytes());

        s.close();
        ss.close();
    }
```

```
}
```

7、利用 TCP 上传文件

从客户端上传到服务器端，其实本质上也就是复制！

```
package july76net;
//上传文件(文本)

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;

public class Demo14 {
    public static void main(String[] args) throws Exception {
        Socket s = new Socket("localhost",12362);

        BufferedReader br  = new BufferedReader(new FileReader("E:/
你好.txt"));
        PrintWriter pw = new PrintWriter(s.getOutputStream(),true);
        String line;
        while((line = br.readLine()) != null){
            pw.println(line);
        }
        s.shutdownOutput();//阻塞式方法的应对，否则会一直等待!

        BufferedReader     br2     =     new     BufferedReader(new
InputStreamReader(s.getInputStream()));
        String str = br2.readLine();//服务器端反馈的数据
        System.out.println(str);

        br.close();
        s.close();
    }
}
```

服务器端

```
package july76net;

import java.io.BufferedReader;
import java.io.FileWriter;
import java.io.InputStreamReader;
```

```
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;

public class Demo15 {
    public static void main(String[] args) throws Exception {
        ServerSocket ss = new ServerSocket(12362);
        Socket s = ss.accept();

        String ip = s.getInetAddress().getHostAddress();
        System.out.println(ip+".....connected!");

        BufferedReader br = new BufferedReader(new
InputStreamReader(s.getInputStream()));
        PrintWriter pw = new PrintWriter(new FileWriter("E:/ 你好
2.txt"),true);//建议打印时都用打印流
        String line;
        while((line = br.readLine()) != null){
            pw.println(line);//不可以写成是write();
        }

        PrintWriter out = new PrintWriter(s.getOutputStream(),true);
        out.println("上传成功！");
        s.close();
        ss.close();
    }
}
```

上传图片（注意图片是 2 进制文件，必须是字节流，否则读取不出来！）；

客户端

```
package july76net;

import java.io.FileInputStream;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.Socket;

public class Demo16 {
    public static void main(String[] args) throws Exception {
        Socket s = new Socket("localhost",10256);
        FileInputStream fis = new FileInputStream("E:/DSC_2451.jpg");
```

```
        OutputStream os = s.getOutputStream();

        byte []buf = new byte[1024];
        int len;
        while((len = fis.read(buf)) != -1){
            os.write(buf);
        }
        s.shutdownOutput();
        InputStream in = s.getInputStream();
        byte []b = new byte[1024];
        int i = in.read(b);
        System.out.println(new String(b,0,i));
        fis.close();
        s.close();
    }
}
```

服务器端

```
package july76net;

import java.io.FileOutputStream;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.ServerSocket;
import java.net.Socket;

public class Demo17 {
    public static void main(String[] args) throws Exception {

        ServerSocket ss = new ServerSocket(10256);
        Socket s = ss.accept();

        String ip = s.getInetAddress().getHostAddress();
        System.out.println(ip+".....connected");

        InputStream is = s.getInputStream();
        FileOutputStream fos = new FileOutputStream("E:/我的照片
0.jpg");

        byte []b = new byte[1024];
        int len;
        while((len = is.read(b)) != -1){
            fos.write(b);
        }
```

```
        OutputStream os = s.getOutputStream();
        os.write("上传成功！".getBytes());

        s.close();
        ss.close();
    }
}
```

**我的总结：**

**对于网络编程而言，重要的是理解其步骤，按照步骤的需要，一步步搭建根基！**
**客户端和服务端需要交互，那么就要构建相对应的流，供其输入和输出！**
**对于阻塞式方法，一定要注意，提供停止标签！**
**对于 PrintWriter ,记得用 println 而不是 write;不要忘了加上 true，自动刷新！**

## 8、小例子

**题目：上传文件，多客户端上传，并且保证不会因为文件的名称而重复！**

客户端：

```
package july76net;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;

public class Demo22 {
    public static void main(String[] args) throws Exception {
        Socket s = new Socket("localhost", 12036);

        BufferedReader br = new BufferedReader(new FileReader("E:/你好.txt"));
        PrintWriter pw = new PrintWriter(s.getOutputStream(),true);

        BufferedReader br2 = new BufferedReader(new InputStreamReader(s.getInputStream()));


        String line;
        while((line = br.readLine()) != null){
```

```
            pw.println(line);
        }
        s.shutdownOutput();

        String str = br2.readLine();
        System.out.println(str);
        s.close();
    }
}
```

服务器端：

```
package july76net;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileWriter;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;


class MyUpdate implements Runnable{
    private Socket s;

    public MyUpdate(Socket s) {
        super();
        this.s = s;
    }

    @Override
    public void run() {

        String ip = s.getInetAddress().getHostAddress();
        System.out.println(ip+".........connected!");
        int count = 0;
        try {
            BufferedReader br = new BufferedReader(new
InputStreamReader(s.getInputStream()));
            File file = new File("E:/");
            File f = new File(file,"你好"+count+".txt");
            while(f.exists()){//如果写成if，就不可以!
                f = new File(file,"你好"+(++count)+".txt");
            }
```

```
            PrintWriter pw = new PrintWriter(new FileWriter(f),true);
            PrintWriter             pw2             =             new
PrintWriter(s.getOutputStream(),true);

            String line;
            while((line = br.readLine()) != null){
                pw.println(line);
            }

            pw2.println("恭喜您，上传成功！");
            s.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

public class Demo23 {
    public static void main(String[] args) throws Exception {

        ServerSocket ss = new ServerSocket(12036);
        while(true){
            Socket s = ss.accept();
            new Thread(new MyUpdate(s)).start();
        }
    }
}
```

# 十二、黑马程序员—反射机制

## 1、反射的概念

**反射的引入：**

Object obj = new Student();

若程序运行时接收到外部传入的一个对象，该对象的编译类型是 Object，但程序又需要调用该对象运行类型的方法：

1.若编译和运行类型都知道，使用 instanceof 判断后，强转。

2.编译时根本无法预知该对象属于什么类，程序只能依靠运行时信息来发现对象的真实信息，这时就必须使用反射了。

**3.要是想得到对象真正的类型，就得使用反射。**

**什么是反射机制？**

简单的来说，反射机制指的是程序在运行时能够获取自身的信息。在 java 中，只要给定类的名字，那么就可以通过反射机制来获得类的所有信息。

**反射机制的优点与缺点：**

为什么要用反射机制？直接创建对象不就可以了吗，这就涉及到了动态与静态的概念，

静态编译：在编译时确定类型，绑定对象,即通过。

动态编译：运行时确定类型，绑定对象。动态编译最大限度发挥了 java 的灵活性，体现了多态的应用，有以降低类之间的藕合性。

一句话，反射机制的优点就是可以实现动态创建对象和编译，体现出很大的灵活性，特别是在 J2EE 的开发。

它的缺点是对性能有影响。使用反射基本上是一种解释操作，我们可以告诉 JVM，我们希望做什么并且它满足我们的要求。这类操作总是慢于只直接执行相同的操作。

**Class 类和 Class 类实例**

Java 程序中的各个 Java 类属于同一类事物,描述这类事物的 Java 类就是 Class 类。

对比提问：众多的人用一个什么类表示？众多的 Java 类用一个什么类表示？

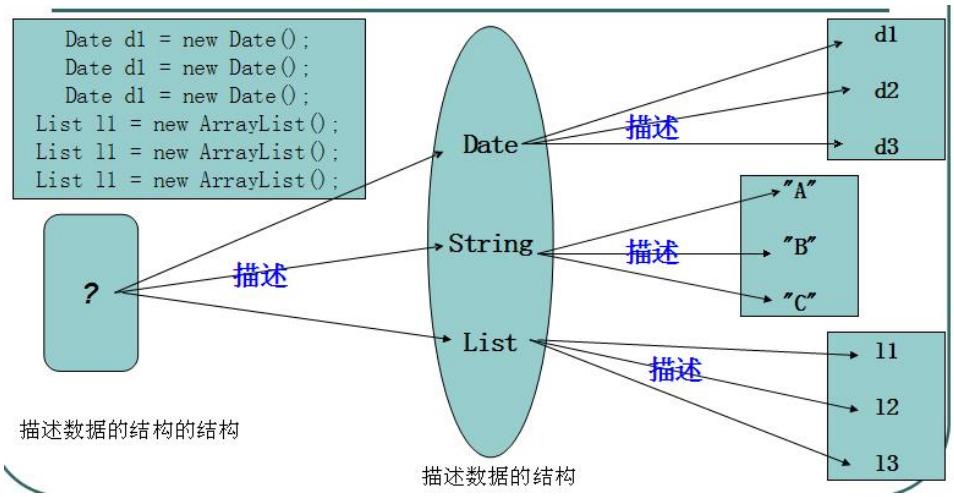
人　　Person

Java 类　　Class

对比提问：　Person 类代表人，它的实例对象就是张三，李四这样一个个具体的人，Class 类代表 Java 类，它的各个实例对象又分别对应什么呢？

对应各个类在内存中的字节码，例如，Person 类的字节码，ArrayList 类的字节码，等等；

**一个类被类加载器加载到内存中，占用一片存储空间，这个空间里面的内容就是类的字节码，不同的类的字节码是不同的，所以它们在内存中的内容是不同的；**




用类来描述对象，类：描述数据的结构

用元数据来描述 Class，MetaData(元数据):描述数据结构的结构;

**反射就是得到元数据的行为;**

**备注：一个类在虚拟机中只有一份字节码；**

## 2、获得 Class 对象

如何得到各个字节码对应的实例对象？
每个类被加载后，系统会为该类生成对应的 Class 对象，通过 Class 对象可以访问到 JVM 中的这个类，
3 种方式：

1、调用某个类的 class 属性获取 Class 对象，如 Date.class 会返回 Date 类对应的 Class 对象(其实就是得到一个类的一份字节码文件)；

2、使用 Class 类的 forName(String className)静态方法，className 表示**全限定名**；如 String 的全限定名:java.lang.String;

3、调用某个对象的 getClass()方法。该方法属于 Object 类；

Class<?> clz = new Date().getClass();

Eg：

```
package junereflect624;

public class ClassDemo1 {
    public static void main(String[] args) throws Exception {
        //获得Class对象的方法（三种）
        //一：调用属性
        Class<String> c = String.class;
        System.out.println(c);// 打印结果： class java.lang.String
                String.class就表示JVM中一份表示String类的字节码
        Class<String> c2 = String.class;
        System.out.println(c == c2);//true都是String类的字节码
    一个类在虚拟机中只有一份字节码；

        //二：使用forName()方法
        //Class cla = Class.forName("String");//ERROR,
        Class<String> cla =
            (Class<String>)Class.forName("java.lang.String");// 必须用上全限定名，否则报错
        System.out.println(c == cla);//true

        //三：利用对象调用Object的getClass方法;
        Class c3 = new String().getClass();
        System.out.println(c == c3);//ture
    }
}
```

**我的总结：获取 Class 对象最常用的是利用属性的方法！**

## 3、九个预定义 Class 对象

基本的 Java 类型（boolean、byte、char、short、int、long、float 和 double）和关键字 void 通过 class 属性也表示为 Class 对象；
Class 类中 boolean isPrimitive() :判定指定的 Class 对象是否表示一个基本类型。
包装类和 Void 类的静态 TYPE 字段；
Integer.TYPE == int.class ;
Integer.class == int.class;
数组类型的 Class 实例对象：
Class<String[]> clz = String[].class;
数组的 Class 对象如何比较是否相等? 数组的维数和数组的类型;
Class 类中 boolean isArray() ：判定此 Class 对象是否表示一个数组类型。

```
package junereflect624;

public class PreClassDemo2 {
    public static void main(String[] args) {

        Class<?> in = int.class;
        System.out.println(in);//int
        Class<?> in2 = Integer.class;
        //包装类都有一个常量TYPE，用来表示其基本数据类型的字节码
        Class<?> in3 = Integer.TYPE;

        System.out.println(in2);//class java.lang.Integer
        System.out.println(in3);//int
        System.out.println(in3 == in);//true 包装类都有一个常量TYPE，用来表示其基本数据类型的字节码，所以这里会相等!
        System.out.println(in3 == in2);//false
        Class<String[]> s = String [].class;
        Class<int[]> i = int [].class;
        //System.out.println(i ==s);//编译根本就通过不了，一个是int，一个是String
    }
    //这两个自定义的方法是可以的，一个int，一个Integer//包装类与基本数据类型的字节码是不一样的
    public void show(int i){}
    public void show(Integer i){}
}
```

## 4、利用 Class 获取类的属性信息

```
package junereflect624;

import java.lang.reflect.Modifier;

class A {
}
interface B{
}
interface C{
}

public class BaseDemo3 extends A implements B,C{

    //内部类
    public class C{}
    public interface D{}
    public static void main(String[] args) {
        //类可以，接口也可以
        Class<BaseDemo3> c = BaseDemo3.class;
        System.out.println(c);//class junereflect624.BaseDemo3

        //得到包名
        System.out.println(c.getPackage());//package junereflect624

        //得到全限定名
        System.out.println(c.getName());//junereflect624.BaseDemo3

        //得到类的简称
        System.out.println(c.getSimpleName());//BaseDemo3

        //得到父类
        /**
         * Class<? super T> getSuperclass() 此处super表示下限
               返回表示此 Class 所表示的实体（类、接口、基本类型或 void）的超
类的 Class。
         */
        System.out.println(c.getSuperclass().getSimpleName());//A,
先获取父类，再获取父类的简称

        //得到接口
```

```
    System.out.println(c.getInterfaces());//[Ljava.lang.Class;@1b6
0280
        Class[] arr = c.getInterfaces();
        for (Class cla : arr) {
            System.out.println(cla);//interface     junereflect624.B
interface junereflect624.C
        }

        //获得public修饰的类
        /**
         * Class<?>[] getClasses()
                    返回一个包含某些 Class 对象的数组，这些对象表示属于此
Class 对象所表示的类的成员的所有公共类和接口。（如果内部类前面没有加上public
的话那么得不到!）
         */
        Class[] cl = c.getClasses();
        System.out.println(cl.length);//在内部类没有加上public修饰的时
候长度为0，加上就是2(获取的是公共的)
        for (Class class1 : cl) {
            System.out.println(class1);
        }

        //获得修饰符
        int i = c.getModifiers();
        System.out.println(i);//常量值1表示public
        System.out.println(Modifier.toString(i));//直接打印出public
    }
}
```

## 5、Class 中得到构造方法 Constructor、方法 Method、字段 Field

常用方法：
Constructor 类用于描述类中的构造方法：
Constructor<T> getConstructor(Class<?>... parameterTypes)
返回该 Class 对象表示类的指定的 public 构造方法;
Constructor<?>[] getConstructors()
返回该 Class 对象表示类的所有 public 构造方法;
Constructor<T> getDeclaredConstructor(Class<?>... parameterTypes)
返回该 Class 对象表示类的指定的构造方法,和访问权限无关;
Constructor<?>[] getDeclaredConstructors()
返回该 Class 对象表示类的所有构造方法,和访问权限无关;

Method 类用于描述类中的方法：
Method getMethod(String name, Class<?> ... parameterTypes)
返回该 Class 对象表示类和其父类的指定的 public 方法;
Method[] getMethods()：
返回该 Class 对象表示类和其父类的所有 public 方法;
Method getDeclaredMethod(String name, Class<?>... parameterTypes)
返回该 Class 对象表示类的指定的方法。和访问权限无关，但不包括继承的方法;
Method[] getDeclaredMethods(): 获得类所有的方法，包括公共、保护、默认（包）访问和私有方法，但不包括继承的方法;

Eg：

```
package junereflect624;

import java.lang.reflect.Constructor;

class Emp{
    private String name;
    private int age;
    private Emp() {
    }
    Emp(String name){
    }
    public Emp(String name,int age){
    }
}

public class ConstructorDemo4 {
    public static void main(String[] args) throws Exception {
        //得到所有的构造器（先得到类）
        Class<Emp> c = Emp.class;
        /**
         * Constructor<?>[] getConstructors()
               返回一个包含某些 Constructor 对象的数组，这些对象反映此 Class 对象所表示的类的所有公共构造方法。
         */
        Constructor[] con = c.getConstructors();//前面的修饰符必须是public才可以在这个方法下获取到
        for (Constructor cons : con) {
            System.out.println("c.getConstructors()"+cons);//如果上面的某构造器public去掉，则显示不出
            /**打印
               public junereflect624.Emp(java.lang.String,int)
             */
```

```
        }

        //得到指定的构造器,也是必须public
        Constructor c1 = c.getConstructor(String.class,int.class);
        System.out.println(c1);//public
junereflect624.Emp(java.lang.String,int)

        System.out.println("====================================");
        //现在想获得不受public影响的,getDeclaredConstructors(),暴力反射

        con = c.getDeclaredConstructors();
        for (Constructor cons : con) {

        System.out.println("c.getDeclaredConstructors()=="+cons);//
    此时不受修饰符的影响
            /**打印
             *  public junereflect624.Emp()
                public junereflect624.Emp(java.lang.String)
                public junereflect624.Emp(java.lang.String,int)
             */
        }
    }
}


package junereflect624;

import java.lang.annotation.Annotation;
import java.lang.reflect.Field;
import java.lang.reflect.Method;

class AB{
    protected String name;
    protected String id;
}
@Deprecated
public class MethodDemo5 extends AB{
    void show(){}
    public void say(){}
    private int age;
    public char c;
    private boolean b;
    public static void main(String[] args) throws Exception {
        Class<MethodDemo5> c = MethodDemo5.class;
```

```
        //获取所有的（包含父类的方法）public修饰的方法
        Method[] m = c.getMethods();
        for (Method method : m) {
            System.out.println(method);
        }
        //总结：4个方法，获取全部，获取特定；不受修饰符影响的全部，不受修饰符影响的特定；（前两个都还是受限制的）

        //获取指定的方法
        Method me = c.getMethod("main", String[].class);
        System.out.println("main  "+me);//main  public  static  void junereflect624.MethodDemo5.main(java.lang.String[])          throws java.lang.Exception

        //访问所有方法，不受访问权限影响
        m = c.getDeclaredMethods();
        for (Method method : m) {
            System.out.println("不受影响的："+method);
        }

        me = c.getDeclaredMethod("show");
        System.out.println(me);//void junereflect624.MethodDemo.show()

        me = c.getMethod("toString");
        System.out.println(me);//public             java.lang.String java.lang.Object.toString()

        /**
         * Method[] getDeclaredMethods()
                返回 Method 对象的一个数组，这些对象反映此 Class 对象表示的类或接口声明的所有方法，
                包括公共、保护、默认（包）访问和私有方法，但不包括继承的方法，只可以对当前类有效
         */
        /*me                                                                =
c.getDeclaredMethod("toString");//ERROR,c.getDeclaredMethod() 不能得到继承的方法
        System.out.println(me);//public             java.lang.String java.lang.Object.toString()
         */
        //得到字段
        Field[] f = c.getFields();
        for (Field field : f) {//只得到了public的
```

```
            System.out.println("字段"+field);
        }

        //特定字段
        Field fi = c.getField("c");//""里面是名称
        System.out.println(fi);//public                                    char
junereflect624.MethodDemo.c

        //得到不受限定名限定的全部字段
        f = c.getDeclaredFields();
        for (Field field : f) {//得到不受修饰符限定的字段，但是只对当前类有
效
            System.out.println("全部字段: "+field);
            /**
             *  全部字段: private int junereflect624.MethodDemo.age
                全部字段: public char junereflect624.MethodDemo.c
                全部字段: private boolean junereflect624.MethodDemo.b
             */
        }
        //注释  Annotation
         Annotation[] a = c.getAnnotations();
         System.out.println(a.length);
         for (Annotation annotation : a) {
            System.out.println(annotation);
        }

         //特定注解
         Deprecated d = c.getAnnotation(Deprecated.class);
         System.out.println(d);
    }
}
```

获取当前对象的字段：

```
package july78javaEnhance;

import java.lang.reflect.Field;

class Stu{
    public String name;
    public String sex;
    public int age;

    public Stu(String name, String sex, int age) {
        super();
```

```
        this.name = name;
        this.sex = sex;
        this.age = age;
    }
}

public class ReflectDemo6 {
    public static void main(String[] args) throws Exception {
        Stu s = new Stu("刘昭", "男", 12);

        Class<Stu> c = Stu.class;

        Field f = c.getField("name");
        System.out.println(f.get(s));////从哪个对象身上取！此时显示刘昭！
        //  修改对象的值
        /**
        Field f = c.getField("name");
        f.set(s,"章泽天");
        System.out.println(f.get(s));//从哪个对象身上取！//此时显示章泽天
        */
    }
}
```

**我的总结：对于方法，字段，构造方法之类用类获取记住四个：获取全部，获取特定，暴力获取全部，暴力获取特定！**

## 6、利用反射创建对象

创建对象：

1、使用 Class 对象的 newInstance()方法创建该 Class 对象的实例，**此时该 Class 对象必须要有无参数的构造方法。**

2、使用 Class 对象获取指定的 Constructor 对象，再调用 Constructor 的 newInstance（）方法创建对象类的实例，此时可以选择使用某个构造方法。如果这个构造方法被私有化起来，那么必须先申请访问，将可以访问设置为 true；

Eg：

最简单的：

```
package junereflect624;

class User{
    /*private User(){//将默认的构造方法私有化的话就不可以再创建对象，两种方法都是这样
```

```
    }*/
    public String toString() {
        return "User对象创建成功！";
    }
}

public class NewInstanceDemo6 {
    public static void main(String[] args) throws Exception {
        //传统方式创建对象
        System.out.println(new User());

        //使用反射的方式
        Class<User> c = User.class;
        User u = c.newInstance();（直接newInstance的话必须保证默认的构造方法正常存在，也就是没有被私有化！这是前提条件）
        System.out.println(u);
    }
}
```

复杂点的：更强大的第二种：

使用指定构造方法来创建对象:
获取该类的 Class 对象。
利用 Class 对象的 getConstructor()方法来获取指定的构造方法。
调用 Constructor 的 newInstance()方法创建对象。

AccessibleObject 对象的 setAccessible(boolean flag)方法,当 flag 为 true 的时候，**就会忽略访问权限(可访问私有的成员)。**
其子类有 Field, Method, Constructor;
**若要访问对象 private 的成员？**
**在调用之前使用 setAccessible(true),**
**Xxx x = getDeclaredXxxx();//才能得到私有的类字段.**

总结步骤：

1. **获取该类的 Class 对象。**
2. **利用 Class 对象的 getConstructor()方法来获取指定的构造方法。**
3. **申请访问（设置为可访问）**
4. **调用 Constructor（构造方法）的 newInstance()方法创建对象。**

例子

```
package junereflect624;
```

```
import java.lang.reflect.Constructor;

class Per{
    private String name;
    private int age;
    private Per(){
    }
    private Per(String name){
    }
    public String toString() {
        return "对象!!! ";
    }
}

public class NewInstanceDemo7 {
    public static void main(String[] args) throws Exception {
        Class<Per> c = Per.class;
        //System.out.println(c.newInstance());;//证明利用无参的可以

        ////先获得需要被调用的构造器(private 修饰的构造方法)
        Constructor<Per> con = c.getDeclaredConstructor();//调用默认
的，什么都不要写
        System.out.println(con);//private junereflect624.Per()
        /*con = c.getDeclaredConstructor(String.class);获取指定的构造
方法
        System.out.println(con);//private
junereflect624.Per(java.lang.String)*/
        //现在只需要执行这个构造器，
        /**
         *  T newInstance(Object... initargs)
               使用此 Constructor 对象表示的构造方法来创建该构造方法的声明
类的新实例，并用指定的初始化参数初始化该实例。
         */

        //私有的成员是受保护的,不能直接访问
        //若要访问私有的成员,得先申请一下
        con.setAccessible(true);//允许访问
        Per p = con.newInstance();//成功，通过私有的受保护的构造方法创建了
对象
        System.out.println("无参构造方法"+p);

        con = c.getDeclaredConstructor(String.class);
        System.out.println(con);//private
junereflect624.Per(java.lang.String)
```

```
        con.setAccessible(true);//允许访问
        p = con.newInstance("liuzhao");//成功，通过私有的受保护的构造方法创建了对象
        System.out.println("String构造方法"+p);
    }
}
```

**备注：对于此时的话，单例模式就不再安全了！反射可破之!!**

验证：对于枚举而言，反射依然没有办法重新创建对象
对于枚举，安全！

```
package junereflect624;

import java.lang.reflect.Constructor;

enum Color{
    RED,BLUE,GREEN;
    private Color(){
    }
}

public class EnumDemo8 {
    public static void main(String[] args) throws Exception {
        Class<Color> c = Color.class;

        Constructor<Color> con = c.getDeclaredConstructor();//（错误在这一行发生，就是说对枚举而言这种方法连构造器都获得不了，）编译可以通过，但是运行就通不过了！
        Color co = (Color) con.newInstance();
        System.out.println(co);//失败，证明对枚举而言不行，所以枚举的单例模式更加安全
        System.out.println(c.isEnum());//true是枚举
    }
}
```

## 7、使用反射调用方法

每个 Method 的对象对应一个具体的底层方法。获得 Method 对象后，程序可以使用 Method 里面的 invoke 方法来执行该底层方法。

Object invoke(Object obj,Object ... args):obj 表示调用底层方法的对象，后面的 args 表示传递的实际参数。
如果底层方法是静态的，那么可以忽略指定的 obj 参数。该参数可以为 null,想想为什么?
如果底层方法所需的形参个数为 0，则所提供的 args 数组长度可以为 0 或 null。
不写,null,或 new Object[]{}
若底层方法返回的是数组类型,invoke 方法返回的不是底层方法的值,而是底层方法的返回类型;

```
package junereflect624;

import java.lang.reflect.Method;

class Dept{
    public String show(String name){//用反射的方法来调用正常的方法
        return name+", 您好！";
    }

    private void privateshow(){//用反射来实现对私有化方法的调用
        System.out.println("privateshow");
    }

    public static void staticshow(){
        System.out.println("staticshow");
    }
}

public class InvokeDemo9 {
    public static void main(String[] args) throws Exception {

        /* 传统方式:
        String name = new Dept().show("刘昭");
        System.out.println(name);*/

        /**
         * Method getMethod(String name, Class<?>... parameterTypes)
                返回一个 Method 对象，它反映此 Class 对象所表示的类或接口的指
            定公共成员方法。
            name - 方法名
            parameterTypes - 参数列表
         */
        //想要通过反射来调用Dept中的方法
        Class<Dept> c = Dept.class;
        Method m = c.getMethod("show", String.class);
        Object o = m.invoke(c.newInstance(), "刘昭");
```

```
        System.out.println(o);

        //私有化的方法
        m = c.getDeclaredMethod("privateshow");//无参方法
        m.setAccessible(true);
        o = m.invoke(c.newInstance());

        //静态方法的调用
        m = c.getMethod("staticshow");
        m.invoke(null);//staticshow为静态方法，不需创建对象，所以这里会是
null
    }
}
```

打印

```
刘昭，您好!
privateshow
staticshow
```

## 8、使用反射调用可变参数方法

**使用反射操作对象-调用可变参数方法**

要把可变参数都当做是其对应的数组类型参数;
如 show(XX... is)作为 show(XX[] is)调用;

若可变参数元素类型是引用类型:

JDK 内部接收到参数之后,会自动拆包取出参数再分配给该底层方法,为此我们需要把这个数组实参先包装成一个 Object 对象或把实际参数作为一个 Object 一维数组的元素再传递。

若可变参数元素类型是基本类型:

JDK 内部接收到参数之后,不会拆包,所以可以不必再封装.不过封装了也不会错.所以建议,不管基本类型还是引用类型都使用 Object[]封装一层,保证无误.

例子

```
package junereflect624;
//可变参数的方法调用

import java.lang.reflect.Method;

class VaryMethod{
    public static void show(int ...args){
        System.out.println("基本数据类型传递过来了！ ");
```

```
    }
    /*public static void show(int[] args){//这是一样的
    }*/
    public static void show(String ...args){
        System.out.println("引用数据类型传递过来了！");
    }
}

public class InvokeVaryDemo10 {
    public static void main(String[] args) throws Exception{
        Class<VaryMethod> c = VaryMethod.class;

        Method m = c.getMethod("show",int[].class);
        m.invoke(null,new int[]{1,2,3});
        m = c.getMethod("show",String[].class);
        //m.invoke(null,new String[]{"A","B","C"});//ERROR
        m.invoke(null,(Object)new String[]{"A","B","C"});//YES,强转为Object类型
        m.invoke(null,new Object[]{new String[]{"A","B","C"}});//推荐写法
    }
}
```

## 9、使用反射操作字段

Field 提供两组方法操作字段：
xxx getXxx(Object obj):获取 obj 对象该 Field 的字段值，此处的 xxx 表示 8 个基本数据类型。若该字段的类型是引用数据类型则使用，Object get(Object obj);
void setXxx(Object obj,xxx val):将 obj 对象的该 Field 字段设置成 val 值，此处的 xxx 表示 8 个基本数据类型。若该字段的类型是引用数据类型则使用，void set(Object obj, Object value);

```
package junereflect624;
//获取字符，并且赋值，然后再取出来（对应的去查看api，比如这个是Field，别的比如Constructor，Method）
```

步骤：

1. 获取类
2. 获取字段
3. 赋值(set(c.newInstance(),""));{如果为私有的话设置可接受}

```
import java.lang.reflect.Field;

class Cat{
    private String name;
```

```
    public int age;
    private String color;
}

public class FieldDemo12 {
    public static void main(String[] args) throws Exception {
        Class<Cat> clz = Cat.class;
        Field[] f = clz.getDeclaredFields();

        for (Field field : f) {
            System.out.println(field);
        }

        Field fi = clz.getDeclaredField("name");
        System.out.println(fi);

        System.out.println(fi.getName());//name

        //核心开始
        /**
         *  void set(Object obj, Object value)
                将指定对象变量上此 Field 对象表示的字段设置为指定的新值。
         */
        Cat c = clz.newInstance();
        fi.setAccessible(true);
        fi.set(c, "刘昭");//赋值成功
        Object o = fi.get(c);
        System.out.println(o);//取出成功

        fi = clz.getDeclaredField("age");
        fi.setAccessible(true);
        fi.set(c, 21);
        int i = fi.getInt(c);//左边的接受类型已经写成了int，右边的返回类型就也必须是int
        System.out.println(i);//获取成功
    }
}

打印
private java.lang.String junereflect624.Cat.name
public int junereflect624.Cat.age
private java.lang.String junereflect624.Cat.color
private java.lang.String junereflect624.Cat.name
name
```

刘昭
21

## 10、反射和泛型-反射来获取泛型信息

通过指定对应的 Class 对象，程序可以获得该类里面所有的 Field，不管该 Field 使用 private 方法 public。获得 Field 对象后都可以使用 getType()来获取其类型。
Class<?> type = f.getType();//获得字段的类型
但此方法只对普通 Field 有效,若该 Field 有泛型修饰,则不能准确得到该 Field 的泛型参数,如 Map<String,Integer>;
为了获得指定 Field 的泛型类型，我们采用：
Type gType = f.getGenericType();得到泛型类型
然后将 Type 对象强转为 ParameterizedType,其表示增加泛型后的类型
Type getRawType()//返回被泛型限制的类型;
Type[]  getActualTypeArguments()//返回泛型参数类型;

**利用反射来获取泛型的类型（泛型信息）**
**步骤：**

1. **获取当前类**
2. **获取目标字段**
3. **获取包含泛型类型的类型 `getGenericType()`**
4. **强转至子类 `ParameterizedType`　　因为 `Type` 没有任何对应的方法**
5. **获得泛型真正的类型 `getActualTypeArguments()`**

例子

```
package junereflect624;

import java.lang.reflect.Field;
import java.lang.reflect.ParameterizedType;
import java.lang.reflect.Type;
import java.util.HashMap;
import java.util.Map;

public class GetGenericTypeDemo14 {
    Map<String,Integer> map = new HashMap<String,Integer>();

    public static void main(String[] args) throws Exception {
        Class c = GetGenericTypeDemo14.class;
        Field f = c.getDeclaredField("map");
        System.out.println(f);
        System.out.println(f.getName());//map
```

```
        // Class<?> getType()  返回一个 Class 对象，它标识了此 Field 对象所表示字段的声明类型。
        Class cl = f.getType();
        System.out.println("获得其类型："+cl);
        //获得其类型：interface java.util.Map

        /**
         *  Type getGenericType() 返回一个 Type 对象，它表示此 Field 对象所表示字段的声明类型。
         *  Type是Class的接口;
         */
        Type t = f.getGenericType();//包含泛型的类型
        System.out.println(t);
        //java.util.Map<java.lang.String, java.lang.Integer>


        /**
         * Type这个类里面没有任何的方法，所以需要调用子类的方法，那么大的类型转到小的类型，需要强转!
         */
        ParameterizedType pt = (ParameterizedType)t;//强转到其子类
        /**
         *  Type[] getActualTypeArguments()
                    返回表示此类型实际类型参数的 Type对象的数组。
             Type getOwnerType()
                    返回 Type 对象，表示此类型是其成员之一的类型。
             Type getRawType()
                    返回 Type 对象，表示声明此类型的类或接口。
         */

        t = pt.getRawType();//类型的类或接口
        System.out.println(t);

        Type[] ts = pt.getActualTypeArguments();
        for (Type type : ts) {
            System.out.println(type);
            /**
             *  class java.lang.String
                class java.lang.Integer
             */
        }
    }
}
```

打印：

```
java.util.Map junereflect624.GetGenericTypeDemo14.map
map
获得其类型：interface java.util.Map
java.util.Map<java.lang.String, java.lang.Integer>
interface java.util.Map
class java.lang.String
class java.lang.Integer
```

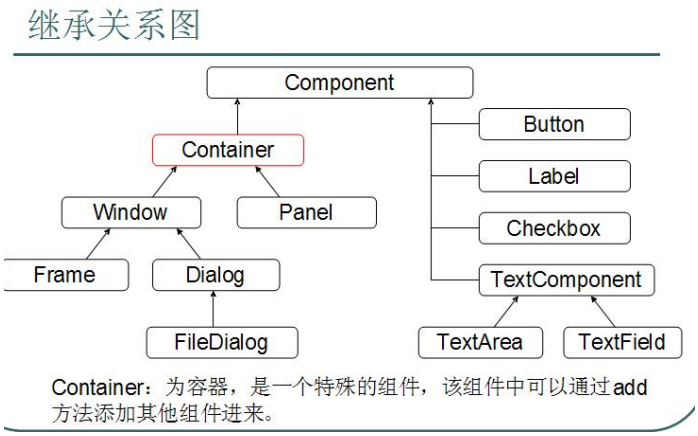
**我的总结：多查找 api，参考 api 中方法使用的限制，比如是否静态、返回值类型等。**

# 十三、黑马程序员—反射机制

## 1、GUI(图形用户界面)

GUI
Graphical User Interface(图形用户接口)。
用图形的方式，来显示计算机操作的界面，这样更方便更直观。
CLI
Command line User Interface (命令行用户接口）
就是常见的 Dos 命令行操作。
需要记忆一些常用的命令，操作不直观。
举例：
比如：创建文件夹，或者删除文件夹等
Java 为 GUI 提供的对象都存在 java.Awt 和 javax.Swing 两个包中。

Act 和 Swing
Awt 与 Swing
java.Awt：Abstract Window ToolKit (抽象窗口工具包)，需要调用本地系统方法实现功能。属重量级控件。
javax.Swing：在 AWT 的基础上，建立的一套图形界面系统，其中提供了更多的组件，而且完全由 Java 实现。增强了移植性，属轻量级控件。




## 2、布局管理器

容器中的组件的排放方式，就是布局。
常见的布局管理器：
FlowLayout（流式布局管理器）**//目前最常用的**
从左到右的顺序排列。
Panel 默认的布局管理器。
BorderLayout（边界布局管理器）
东，南，西，北，中
Frame 默认的布局管理器。
GridLayout（网格布局管理器）
规则的矩阵
CardLayout（卡片布局管理器）
选项卡
GridBagLayout（网格包布局管理器）
非规则的矩阵

**使用方法：new frame().setLayout(new FlowLayout());**

**创建图形化界面：**
**1，创建 frame 窗体。**
**2，对窗体进行基本设置。　　比如大小，位置，布局。**
**3，定义组件。**
**4，将组件通过窗体的 add 方法添加到窗体中。**
**5，让窗体显示，通过 setVisible(true)**

Eg：一个简单的窗体

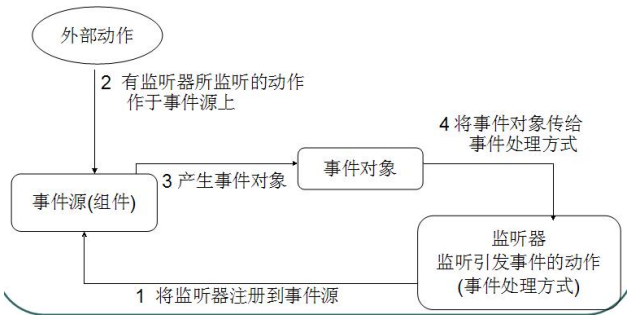
```
import java.awt.*;
import java.awt.event.*;
```

```
import javax.swing.*;
import javax.swing.event.*;

class  SwingDemo
{
    public static void main(String[] args)
    {
        JFrame f = new JFrame();

        f.setBounds(300,100,500,400);

        f.setLayout(new FlowLayout());

        JButton but = new JButton("我是一个按钮");

        f.add(but);

        f.addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent e)
            {
                System.exit(0);
            }
        });
        f.setVisible(true);
    }
}
```

## 3、事件监听机制组成

事件源（组件）
事件（Event）
监听器（Listener）
事件处理（引发事件后处理方式）

事件监听机制流程图




**务必记牢：**
**确定事件源（容器或组件）**
**通过事件源对象的 addXXXListener()方法将侦听器注册到该事件源上。**
**该方法中接收 XXXListener 的子类对象，或者 XXXListener 的子类 XXXAdapter 的子类对象。**
**一般用匿名内部类来表示。**
**在覆盖方法的时候，方法的参数一般是 XXXEvent 类型的变量接收。**
**事件触发后会把事件打包成对象传递给该变量。（其中包括事件源对象。通过 getSource() 或者，getComponent()获取。）**

事件源：就是 awt 包或者 swing 包中的那些图形界面组件。
事件：每一个事件源都有自己特有的对应事件和共性事件。
监听器：将可以触发某一个事件的动作（不只一个动作）都已经封装到了监听器中。
以上三者，在 java 中都已经定义好了。直接获取其对象来用就可以了。
我们要做的事情是，就是对产生的动作进行处理。

Eg:

```
package june610;

import java.awt.Button;
import java.awt.FlowLayout;
import java.awt.Frame;
import java.awt.TextField;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.KeyAdapter;
import java.awt.event.KeyEvent;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

class MyWin extends WindowAdapter{
    public void windowClosing(WindowEvent e){
        System.out.println("hahahha");
```

```
        System.exit(0);
    }
}

public class FrameDemo {//如果写成内部类的形式那么前面必须加上public static，因为主方法是静态的，不能调用动态类或者方法
    public static void main(String[] args) {
        //设置窗体
        Frame f = new Frame("窗体");
        f.setSize(400, 300);
        f.setLocation(500, 300);//距离左侧，距离上面
        /**可以用这个方法一次性设置
         * setBounds(int x, int y, int width, int height) 移动组件并调整其大小。
         */
        Button b = new Button("按钮");
        Button b2 = new Button("按钮2");
        TextField tf = new TextField(20);
        f.add(b);//把按钮添加到窗体上
        f.add(b2);//把按钮添加到窗体上
        f.add(tf);//在窗体上的的顺序按照添加的顺序

        f.setLayout(new FlowLayout());//设置容器的布局管理器
        //f.addWindowListener(new MyWin());

        b.addActionListener(new ActionListener(){//通过匿名内部类，方便添加动作监听器

            public void actionPerformed(ActionEvent e) {
                System.out.println("按钮把界面关闭了");
                //System.exit(0);
            }
        });

        //鼠标
        b.addMouseListener(new MouseAdapter() {//鼠标动作监听器
            int count = 1;
            public void mouseEntered(MouseEvent e){
                System.out.println("鼠标进入"+(count++)+"次！");
            }
        });

        b.addMouseListener(new MouseAdapter(){//和上面的一样，可以写在一起
```

```
        int clickCount = 1;
        public void mouseClicked(MouseEvent e){
            if(e.getClickCount() == 2){
            System.out.println("双击动作"+clickCount++);
            }
        }
    });

    /*
    f.addWindowListener(new WindowAdapter()//匿名内部类的写法
    {
        public void windowClosing(WindowEvent e)
        {
            System.out.println("我关");
            System.exit(0);
        }
        public void windowActivated(WindowEvent e)
        {
            System.out.println("我活了。");

        }

        public void windowOpened(WindowEvent e)
        {
            System.out.println("我被打开了,hahahhahah");
        }

    });*/

    //键盘:
    b2.addKeyListener(new KeyAdapter() {

        public void keyPressed(KeyEvent e)
        {
            System.out.println("键盘的作用");//用鼠标按没反应,

System.out.println(e.getKeyChar()+"---"+e.getKeyCode());//f---
70等、
            if(e.getKeyCode() == 27){//按住esc键退出
                System.out.println("ESC键把我关闭了！");
                System.exit(0);
            }
            //组合键去关闭 CTRL + ENTER
            if(e.isControlDown()     &&     e.getKeyCode()     ==
```

```
KeyEvent.VK_ENTER){
                    System.out.println("CTRL + ENTER组合键把我关闭了！ ");
                    System.exit(0);
                }
            }
        });
        //文本框
        tf.addKeyListener(new KeyAdapter() {
            public void keyPressed(KeyEvent e){
                if(!(e.getKeyCode()>=KeyEvent.VK_0                    &&
e.getKeyCode()<=KeyEvent.VK_9)){
                    System.out.println(e.getKeyChar()+"不符合是数字！");
                }
            }
        });

        f.setVisible(true);//设置可见性

    }

}
```

Eg://列出文件夹内全部的文件

```
package june610;

import java.awt.Button;
import java.awt.FlowLayout;
import java.awt.Frame;
import java.awt.TextArea;
import java.awt.TextField;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.KeyAdapter;
import java.awt.event.KeyEvent;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.io.File;

class MyFrame {
    private Frame f;
    private Button b;
    private TextField tf;
    private TextArea ta;
```

```
MyFrame() {
    init();
}

void init() {
    f = new Frame("我的电脑");
    f.setBounds(300, 100, 600, 500);
    f.setLayout(new FlowLayout());
    b = new Button("转到");
    tf = new TextField(60);
    ta = new TextArea(25, 70);

    f.add(tf);
    f.add(b);
    f.add(ta);
    f.setVisible(true);
    action();
}

//窗体上的操作
void action() {
    f.addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
            System.exit(0);
        }
    });

    buttonAction();
    keyAction();
}
void keyAction(){
    //设置键盘监听器，当输入enter键的时候实现和点击鼠标同样的功能!
    b.addKeyListener(new KeyAdapter() {
        public void keyPressed(KeyEvent e){
            //if(e.getKeyCode() == 10){
                //buttonAction();
            //}

            String dirPath = tf.getText();// 获取文本（我们想验证的是路径），接下来获取文件
            File file = new File(dirPath);// 获取文件
            if (file.exists() && file.isDirectory()) {// 判断，存在否以及是否是文件夹
                ta.setText("");// 如果符合条件的话，清空以前的数据；
```

```
                String[] names = file.list();
                for (String name : names) {
                    ta.append(name + "\r\n");
                }
                System.out.println("=======");
            } else {
                ta.setText("");
                ta.append("对不起，请确认您输入的是路径！");
            }

            System.out.println(e.getKeyCode());
        }
    });

    tf.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            String dirPath = tf.getText();// 获取文本（我们想验证的是路径），接下来获取文件
            File file = new File(dirPath);// 获取文件
            if (file.exists() && file.isDirectory()) {// 判断，存在否以及是否是文件夹
                ta.setText("");// 如果符合条件的话，清空以前的数据；
                String[] names = file.list();
                for (String name : names) {
                    ta.append(name + "\r\n");
                }
                System.out.println("=======");
            } else {
                ta.setText("");
                ta.append("对不起，请确认您输入的是路径！");
            }
        }
    });
}

void buttonAction() {
    b.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            String dirPath = tf.getText();// 获取文本（我们想验证的是路径），接下来获取文件
            File file = new File(dirPath);// 获取文件
            if (file.exists() && file.isDirectory()) {// 判断，存在否以及是否是文件夹
                ta.setText("");// 如果符合条件的话，清空以前的数据；
```

```
                    String[] names = file.list();
                    for (String name : names) {
                        ta.append(name + "\r\n");
                    }
                    System.out.println("=======");
                } else {
                    ta.setText("");
                    ta.append("对不起，请确认您输入的是路径！");
                }
            }
        });
    }
}

public class FrameDemo3 {
    public static void main(String[] args) {
        new MyFrame();
    }
}
```

## 4、Dialog

Dialog构造方法

* Dialog(Frame owner, String title, boolean modal)

构造一个最初不可见的 Dialog，它带有指定的所有者 Frame、标题和模式。

**备注：Dialog的模式区别在于：**

**true的话对话框依附于窗体，不取消Dialog不可以操作窗体，**

**false 的话，不取消 Dialog 可以操作窗体！**

```
package june610;

import java.awt.Button;
import java.awt.Dialog;
import java.awt.FlowLayout;
import java.awt.Frame;
import java.awt.Label;
import java.awt.TextArea;
import java.awt.TextField;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.KeyAdapter;
import java.awt.event.KeyEvent;
import java.awt.event.WindowAdapter;
```

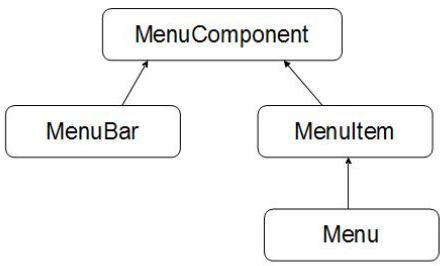
```
import java.awt.event.WindowEvent;
import java.io.File;

public class FrameDemo4 {
    public static void main(String[] args) {
        final Frame f = new Frame("我的电脑");
        f.setBounds(300, 100, 600, 500);
        f.setLayout(new FlowLayout());
        Button b = new Button("转到");
        Button okBut = new Button("确定");
        final TextField tf = new TextField(60);
        final TextArea ta = new TextArea(25, 70);

        f.add(tf);
        f.add(b);
        f.add(ta);
        f.setVisible(true);

        final Dialog d = new Dialog(f,"提示信息",true);
        final Label lab = new Label();//没有给出内容，用到的时候再给出！
        d.add(lab);//label标签加到Dialog上去！
        d.setBounds(400, 200, 240, 150);
        d.setLayout(new FlowLayout());
        d.add(okBut);

        okBut.addKeyListener(new KeyAdapter() {
            public void keyPressed(KeyEvent e){
                d.setVisible(false);
            }
        });

        //只可以对鼠标有作用！
        okBut.addActionListener(new ActionListener(){

            public void actionPerformed(ActionEvent e) {
                d.setVisible(false);
            }
        });

        d.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                d.setVisible(false);//对话框不显示
            }
        });
```

```
        //窗体上的操作
        f.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });

        //设置键盘监听器，当输入enter键的时候实现和点击鼠标同样的功能!
        /*tf.addKeyListener(new KeyAdapter() {
            public void keyPressed(KeyEvent e) {
                if (e.getKeyCode() == 10) {
                    // buttonAction();
                    run(tf,ta,f,d,lab);
                }
                System.out.println(e.getKeyCode());
            }
        });*/

        //和上面被注释的代码实现的是同样的功能，也是键盘控制，不过不能设定哪个键，只有enter!
        tf.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                run(tf,ta,f,d,lab);
                // System.out.println(text);
            }
        });

        //给转到添加键盘和鼠标双控制
        b.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                run(tf,ta,f,d,lab);
                // System.out.println(text);
            }
        });

        b.addKeyListener(new KeyAdapter() {
            public void keyPressed(KeyEvent e){
                run(tf,ta,f,d,lab);
            }
        });
    }

    //封装这一方法，为了方便使用(注意这个时候传递参数太多了，尽量避免这种情况的
```

```
发生！
    //在一个方法内部创建的对象只有在自己方法体里面才可以直接调用，而在外部方法必须传递参数）
    public static void run(TextField tf,TextArea ta,Frame f,Dialog d,Label lab){

        String dirPath = tf.getText();// 获取文本（我们想验证的是路径），接下来获取文件
        File file = new File(dirPath);// 获取文件
        if (file.exists() && file.isDirectory()) {// 判断，存在否以及是否是文件夹
            ta.setText("");// 如果符合条件的话，清空以前的数据；
            String[] names = file.list();
            for (String name : names) {
                ta.append(name + "\r\n");
            }
        } else {
            //备注：应该在这里构建对话框，为了内存的优化，用到的时候才创建对象，用不到就不创建！
            String info = "您输入的信息："+dirPath+"有误，请重新输入！";

            lab.setText(info);
            d.setVisible(true);

            /**可以这样写，但是不专业，现在弹出对话框！
             * ta.setText("");
               ta.append("对不起，请确认您输入的是路径！");
             */
        }
    }
}
```

## 5、菜单

继承体系



**MenuBar,Menu,MenuItem 之间的关系：**
**先创建菜单条，再创建菜单，每一个菜单中建立菜单项。**
**也可以菜单添加到菜单中，作为子菜单。**
**通过 setMenuBar()方法，将菜单添加到 Frame 中。**

```
package june610;

import java.awt.FileDialog;
import java.awt.Frame;
import java.awt.Menu;
import java.awt.MenuBar;
import java.awt.MenuItem;
import java.awt.TextArea;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

//最后导出jar包，用eclipse，必须要指定main函数!! 不能直接点击finished!!!

/**
 * 菜单的结构MenuBar(相当于一个环境，f.set、、)
 *  -->Menu(菜单，m.add、、可以添加自己，也可以添加Item)
 *  -->MenuItem(条目 )
 */
class MyMenuDemo{
    private Frame f;//首先声明对象的好处，全局可以调用!
    private MenuBar mb;
    private Menu m,subm;
    private MenuItem mi,close,save,open;
    private FileDialog openDia,saveDia;
    private TextArea ta;
    private File file;

    MyMenuDemo(){
        f = new Frame("我的电脑");
        f.setBounds(400, 150, 500, 500);
        //备注：此时没有设置布局管理器类型（因为不设置的话下面的TextArea会很
```

```
爽!)
        mb = new MenuBar();
        m = new Menu("文件");
        save = new MenuItem("保存");
        open = new MenuItem("打开");
        subm = new Menu("子菜单");
        close = new MenuItem("退出");
        mi = new MenuItem("子菜单2");
        openDia = new FileDialog(f, "我的打开", FileDialog.LOAD);//加
载
        saveDia = new FileDialog(f, "我的保存", FileDialog.SAVE);//保
存
        ta = new TextArea();

        f.setMenuBar(mb);//添加
        mb.add(m);
        subm.add(mi);
        m.add(subm);
        m.add(open);
        m.add(save);
        m.add(close);
        f.add(ta);

        f.setVisible(true);


        init();
    }

    public  void init(){

        //打开文件，弹出对话框
        open.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e) {
                openDia.setVisible(true);
                String path = openDia.getDirectory();
                String name = openDia.getFile();
                //ta.append(path+"-----"+name+"\n");
                if(path==null || name==null)
                    return ;

                ta.setText("");//每次开始都清空

                file = new File(path, name);
```

```
                try {
                    BufferedReader br = new BufferedReader(new
FileReader(file));//缓冲流
                    String line = null;
                    while((line = br.readLine()) != null){
                        ta.append(line+"\n");
                    }
                    br.close();
                } catch (IOException e1) {
                    e1.printStackTrace();
                }

            }
        });

        //保存文件，弹出对话框!
        save.addActionListener(new ActionListener() {

            public void actionPerformed(ActionEvent e) {
                //注意问题：只有文件第一次保存（不存在）的时候才需要你去弹出对
话框，以后只保存，不弹出!
                if(file == null){
                    saveDia.setVisible(true);//不存在才弹，创建文件
                    String path = saveDia.getDirectory();
                    String name = saveDia.getFile();

                    if(name == null || path == null)
                        return;

                    //目录和名字正确，但是文件不存在，就新建一个文件!
                    file = new File(path,name);
                }

                try {
                    BufferedWriter bw = new BufferedWriter(new
FileWriter(file));

                    String s = ta.getText();
                    bw.write(s);
                    bw.close();

                } catch (IOException e1) {
                    e1.printStackTrace();
                }
```

```
            }
        });

        f.addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent e){
                System.exit(0);
            }
        });

        close.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e) {
                System.exit(0);
            }
        });
    }
}

public class MenuDemo {
    public static void main(String[] args) {
        new MyMenuDemo();
    }
}
```

## 1、Eclipse 的使用技巧

eclipse 与 myeclipse 的关系(都属于 java 开发的工具)：
后者是前者的一个插件，后来为了方便使用，myeclipse 集合了 eclipse，后者是收费的。
可大部分人都是用 eclipse 等常用的免费软件。最好用的因为别人都在用。这就是市场。

eclipse 是 java 的一种 IDE 开发工具。

**我的总结：请注意一些英文简写的含义，对应的每个单词。平时积累！**

什么叫一个工程（project）：
我的看法是，一个独立的项目就是一个工程，一个工程包含多个源文件，运行环境，需要的资源。

IDE（Integrated Drive Electronics）开发工具都支持使用工程化方式管理一个项目的程序开发过程，一般来说一个相对独立的项目就是一个工程，一个项目中涉及的多个 java 文件，资源文件等用一个工程进行管理。(在这里可以看看以前工作间中的某个工程的结构)，在不使用工程管理的情况下，如果一个项目中包括多个 Java 源文件，编程人员需要精心维护这些源文件之间、以及源文件与其它文件的目录关系，需要逐一编译这些源文件，需要手工启动运行编译后的结果。如果将一个程序的所有源文件用一个工程来组织，开发工具能对所有源文件集中管理，记住每个源文件的位置和相互关系。 工程中有哪几个源文件、启动类是哪个、启动参数设置等配置信息在工程中都记录。

工作间（workspace）：

一个工作间包含多个工程，工作间与工作间之间可以独立配置信息

**（编译、运行的版本、快捷键之类的）必须注意：不同的 jre 对应的效果可能不一样，例如 java5 新特性你用之前的版本，就会报错！**

一个 workspace 可以包含多个 project，一个 workspace 保留了 eclipse 的一套环境选项的配置，例如，所使用的 javac 和 java 命令，等等，细节请查看 window->preferences。如果要为 eclispe 再配置一套环境选项，可以再创

建一个 workspace。Package explorer 视图窗口中的 filters 菜单项，可以显示空的父包（此功能默认是关闭的）。

设置快捷键、工作间。

快捷键使用技巧（经常出现软件之间快捷键冲突）：

快捷键的位置：window->preferences->General->keys，设置 alt+/键进行内容提示时，要注意解除 alt+/键原来的绑定关系，直接输入 alt+/就可以找到它的绑定关系，删除绑定关系时也可以使用 remove binding 这个按钮，课后必须教会大家在 eclipse 中。

代码模板的设置位置：java->editor->Templates

代码模板(template)的作用

**自己可以添加，自定义，例如设置 tryf 方便敲出 try{}finally{};**

多想一想：eclipse 工作台中的所有工程继承工作台的配置，其中某个工程也可以覆盖工作台的配置！这是不是 java 面向对象的思想啊？

设置单个工程的 javac 和 java

选择工程，右键->properties 可以设置 javac，右键->run asàopen run dialog 可以设置 java。

例子：

先用新的工作间，然后创建新工程，默认的语言即为 5.0。先使用 Integer x = 3;调整编译器的语法版本为 1.4，看到 eclipse 窗口报错了。然后将这个工程的语言设置为 6.0，马上又看到 bad version .class 运行错误了，这是因为 myeclise 自带的 java 为 1.5 。然后再将整个工作间的 javac 设置为 6.0（eclipse 自带的是 jdk1.5），然后看新建工程的 javac，也随之改成了 6.0，

运行则又报 bad version .class 错误。将工程的编译语言再单独改为 5.0，运行则没了问题。整个工作间的语言设置为 6.0 后，再将整个工作间的 java 也设置为自己安装的 java6。

Perspective 与 view（透视图与视图）
1）所谓的不同的透视图：不同的小窗口（view）构成的大窗口（perspective），便于进行特有的操作，通过 window 的 b 按钮选择不同的 view。
2）在 eclipse 中怎么调试？先双击最左边设置断点，然后回到代码区点右键出现 debug，出现的 debug 视图后选择需要调试的数据进行 watch。

关于工作间配置 javac（编译版本）和 java（运行版本），简单来说就是高版本运行同级或低版本编译的源文件。向下兼容。

怎么导入工程
找到需要导入的工程，然后粘贴到所需要的工作台，import 导入（看是否需要设置 jre）,buildpath 中设置。

## 2、静态导入（1.5 特性，好处是不用写类名）

**import 语句可以导入一个类或某个包中的所有类**
**import static 语句导入一个类中的某个静态方法或所有静态方法**
语法举例：
import static java.lang.Math.sin;
import static java.lang.Math.*;

例子：
原来的：

```
package reviewDemo76;

public class Demo12 {
    public static void main(String[] args) {
        System.out.println(Math.max(12, 15));
    }
}
```

使用静态导入的：

```
package reviewDemo76;
import static java.lang.Math.*;

public class Demo12 {
    public static void main(String[] args) {
        System.out.println(max(12, 15));
    }
```

}

## 3、可变参数

是为了解决一个简单的方法在参数不同时多个重载的问题，用带可变参数的一个函数就能解决。

可变参数的特点：

**1)只能出现在参数列表的最后；**

2)　...　位于变量类型和变量名之间，前后有无空格都可以；

3)调用可变参数的方法时，编译器为该可变参数隐含创建一个数组，在方法体中以数组的形式访问可变参数。

```
Public int add(int　x, int... args){//也可以直接（int..args）就是说传不传都可以
Int sum = x;
For(int i = 0; i<=args.lengrth;i++){
Sum+=args[i];
}
return sum;
}
```

## 4、增强 for 循环

语法：

　　for ( type　变量名：集合变量名 )　{ … }

注意事项：

　　迭代变量必须在( )中定义！

　集合变量可以是数组或实现了 Iterable 接口的集合类

举例：

```
    public static int add(int x,int ...args) {
          int sum = x;
          for(int arg:args) {
            sum += arg;
              }
              return sum;
    }
```

**我的总结：这些都是 java5 出现的新特性！**

## 5、基本数据类型的自动拆箱与装箱

——>　-128~127 之间的特殊性。为什么要这样设计，好处？

——>　享元模式（Flyweight Pattern）：享元模式的特点是，复用我们内存中已存在的对象，

降低系统创建对象实例。

自动装箱：
Integer num1 = 12;

自动拆箱：
System.out.println(num1 + 12);
基本数据类型的对象缓存：
Integer num1 = 12;
Integer num2 = 12;
System.out.println(num1 == num2);

Integer num3 = 129;
Integer num4 = 129;
System.out.println(num3 == num4);//false

Integer num5 = Integer.valueOf(12);
Integer num6 = Integer.valueOf(12);
System.out.println(num5 == num6);//false

**我的总结：对于享元模式，将最常用的封装以便于我们可以复用！**

## 6、枚举

为什么要有枚举？

问题：要定义星期几或性别的变量，该怎么定义？假设用 1-7 分别表示星期一到星期日，但有人可能会写成 int weekday = 0;或即使使用常量方式也无法阻止意外。
枚举就是要让某个类型的变量的取值只能为若干个固定值中的一个，否则，编译器就会报错。枚举可以让编译器在编译时就可以控制源程序中填写的非法值，普通变量的方式在开发阶段无法实现这一目标。

**用普通类如何实现枚举功能**，定义一个 Weekday 的类来模拟枚举功能。

1、私有的构造方法。

2、每个元素分别用一个公有的静态成员变量表示。

可以有若干公有方法或抽象方法。采用抽象方法定义 nextDay 就将大量的 if.else 语句转移成了一个个独立的类

Eg：
```
package july78javaEnhance;
//很好的一道题目。

enum TrafficLamp{
    /**
     * 对于这道题目而言，使用的其实是匿名对象内部类！
     * 枚举最简单的创建对象方法：RED，YELLOW，GREEN
```

```
     * 对于普通的类而言，创建对象：Person p = new Person();
     *
     */

    RED(30){

        @Override
        public TrafficLamp nextLamp() {
            return YELLOW;
        }

    },

    YELLOW(40){

        @Override
        public TrafficLamp nextLamp() {
            return GREEN;
        }

    },

    GREEN(45){

        @Override
        public TrafficLamp nextLamp() {
            return RED;
        }

    };
    public abstract TrafficLamp nextLamp();
    private int time;
    private TrafficLamp(int time){
        this.time = time;
    }
    public int getTime() {
        return time;
    }
    public void setTime(int time) {
        this.time = time;
    }

}
```

```
public class EnumDemo4 {
    public static void main(String[] args) {
        System.out.println(TrafficLamp.RED.nextLamp());
        System.out.println(TrafficLamp.RED.nextLamp().getTime());

    }
}
```

## 7、反射实现框架功能

框架与框架要解决的核心问题

我做房子卖给用户住，由用户自己安装门窗和空调，我做的房子就是框架，用户需要使用我的框架，把门窗插入进我提供的框架中。框架与工具类有区别，工具类被用户的类调用，而框架则是调用用户提供的类。

框架要解决的核心问题

我在写框架（房子）时，你这个用户可能还在上小学，还不会写程序呢？我写的框架程序怎样能调用到你以后写的类（门窗）呢？

因为在写才程序时无法知道要被调用的类名，所以，在程序中无法直接 new 某个类的实例对象了，而要用反射方式来做。

综合案例

先直接用 new 语句创建 ArrayList 和 HashSet 的实例对象，演示用 eclipse 自动生成 ReflectPoint 类的 equals 和 hashcode 方法，比较两个集合的运行结果差异。

然后改为采用配置文件加反射的方式创建 ArrayList 和 HashSet 的实例对象，比较观察运行结果差异。

引入了 elipse 对资源文件的管理方式的讲解。

Eg:

```
package july78javaEnhance;
//利用资源文件加上反射操作！

import java.io.FileInputStream;
import java.io.InputStream;
import java.util.Collection;
import java.util.Properties;

public class ReflectPropertiesDemo8 {
    public static void main(String[] args) throws Exception {
        //写出来的三个全部是可以的，但是相对又不一样！

        // 可以的绝对路径 InputStream ips = new FileInputStream("src/july78javaEnhance/MyPro.properties");// 将这个
```

文件加载进来！

```
        //      可      以      的      InputStream      ips      =
ReflectPropertiesDemo8.class.getResourceAsStream("MyPro.propertie
s");
        InputStream                          ips                          =
ReflectPropertiesDemo8.class.getClassLoader().getResourceAsStream
("july78javaEnhance/MyPro.properties");
        Properties pro = new Properties();
        pro.load(ips);
        ips.close();

        String className = pro.getProperty("className");

        Collection          col          =          (Collection)
Class.forName(className).newInstance();

        //Collection col = new TreeSet();//这样的做法也可以但是现在不想通
过在等号右边具体的新建名称来做！

        col.add(12);
        col.add(13);
        col.add(12);
        col.add(23);
        col.add(36);
        System.out.println(col);
    }
}
```

## 1、内省　了解 JavaBean

**JavaBean 是一种特殊的 Java 类，主要用于传递数据信息，这种 java 类中的方法主要用于访问私有的字段，且方法名符合某种命名规则。**

如果要在两个模块之间传递多个信息，可以将这些信息封装到一个 JavaBean 中，这种 JavaBean 的实例对象通常称之为值对象（Value Object，简称 VO）。这些信息在类中用私有字段来存储，如果读取或设置这些字段的值，则需要通过一些相应的方法来访问，大家觉得这些方法的名称叫什么好呢？JavaBean 的属性是根据其中的 setter 和 getter 方法来确定的，而不是根据其中的成员变量。如果方法名为 setId，中文意思即为设置 id，至于你把它存到哪个变量上，用管吗？如果方法名为 getId，中文意思即为获取 id，至于你从哪个变量上取，用管吗？去掉 set 前缀，剩余部分就是属性名，如果剩余部分的第二个字母是小写的，则把剩余部分的首字母改成小的。

setId()的属性名　id
isLast()的属性名　last
setCPU 的属性名是什么？　CPU
getUPS 的属性名是什么？　UPS

总之，一个类被当作 javaBean 使用时，JavaBean 的属性是根据方法名推断出来的，它根本看不到 java 类内部的成员变量。

一个符合 JavaBean 特点的类可以当作普通类一样进行使用，但把它当 JavaBean 用肯定需要带来一些额外的好处，我们才会去了解和应用 JavaBean！好处如下：

在 Java EE 开发中，经常要使用到 JavaBean。很多环境就要求按 JavaBean 方式进行操作，别人都这么用和要求这么做，那你就没什么挑选的余地！

JDK 中提供了对 JavaBean 进行操作的一些 API，这套 API 就称为内省。如果要你自己去通过 getX 方法来访问私有的 x，怎么做，有一定难度吧？用内省这套 api 操作 JavaBean 比用普通类的方式更方便。

## 2、内省综合案例和 Beanutils 工具包

演示用 eclipse 自动生成 ReflectPoint 类的 setter 和 getter 方法。

直接 new 一个 PropertyDescriptor 对象的方式来让大家了解 JavaBean API 的价值，先用一段代码读取 JavaBean 的属性，然后再用一段代码设置 JavaBean 的属性。

演示用 eclipse 将读取属性和设置属性的流水帐代码分别抽取成方法：

只要调用这个方法，并给这个方法传递了一个对象、属性名和设置值，它就能完成属性修改的功能。

得到 BeanInfo 最好采用“obj.getClass()”方式，而不要采用“类名.class”方式，这样程序更通用。

采用遍历 BeanInfo 的所有属性方式来查找和设置某个 RefectPoint 对象的 x 属性。在程序中把一个类当作 JavaBean 来看，就是调用 IntroSpector.getBeanInfo 方法，得到的 BeanInfo 对象封装了把这个类当作 JavaBean 看的结果信息。

演示用 eclipse 如何加入 jar 包，先只是引入 beanutils 包，等程序运行出错后再引入 logging 包。

在前面内省例子的基础上，用 BeanUtils 类先 get 原来设置好的属性，再将其 set 为一个新值。

get 属性时返回的结果为字符串，set 属性时可以接受任意类型的对象，通常使用字符串。

用 PropertyUtils 类先 get 原来设置好的属性，再将其 set 为一个新值。

get 属性时返回的结果为该属性本来的类型，set 属性时只接受该属性本来的类型。

演示去掉 JavaBean（ReflectPoint）的 public 修饰符时，BeanUtils 工具包访问 javabean 属性时出现的问题。

Eg：

```
package javaBean.cn.itcast;


import java.beans.BeanInfo;
import java.beans.IntrospectionException;
import java.beans.Introspector;
import java.beans.PropertyDescriptor;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

import org.apache.commons.beanutils.BeanUtils;

public class BeansTest {

    public static void main(String[] args) throws Exception {
        // TODO Auto-generated method stub
        Person p = new Person();
        p.setName("刘昭");

        String propertiesName = "name";
         String name = extracted(p, propertiesName);//演示了用eclipse
抽取方法
         System.out.println(name);

        String propertiesAge = "age";
        int age = 23;
        SetAge(p, propertiesAge, age);

        String  name1  =  BeanUtils.getProperty(p,  "name");// 使 用
beanUtils工具包进行获取和设置属性（尽管这些属性是私有的，可是有方法啊，是不是
很方便）
        System.out.println(BeanUtils.getProperty(p,
"name").getClass().getName());
        System.out.println(name1);

        BeanUtils.setProperty(p, "age", 19);
        System.out.println(p.getAge());

        /*打印结果
```

```
         * 刘昭
           23
           java.lang.String
                 刘昭
          19*/

    }

    private static void SetAge(Person p, String propertiesAge, int age)
            throws IntrospectionException, IllegalAccessException,
            InvocationTargetException {
        PropertyDescriptor bp1 = new PropertyDescriptor(propertiesAge,
p.getClass());
        Method methodSetAge = bp1.getWriteMethod();
        methodSetAge.invoke(p,age);
        System.out.println(p.getAge());

    }

    private static String  extracted(Object p, String propertiesName)
            throws IntrospectionException, IllegalAccessException,
            InvocationTargetException {
        /*PropertyDescriptor           bp           =           new
PropertyDescriptor(propertiesName, p.getClass());
        Method methodGetName = bp.getReadMethod();
        Object readVal = methodGetName.invoke(p);
        System.out.println(readVal);*/

        BeanInfo beanInfo =  Introspector.getBeanInfo(p.getClass());
        PropertyDescriptor[]                  pds                  =
beanInfo.getPropertyDescriptors();
        Object retVal = null;
        for(PropertyDescriptor pd : pds){
            if(pd.getName().equals(propertiesName))
            {
                Method methodGetX = pd.getReadMethod();
                retVal = (String)methodGetX.invoke(p);
                break;
            }
        }
        return (String) retVal;

    }
}
```

## 3、注解（Annotation）

**注解相当于一种标记，在程序中加了注解就等于为程序打上了某种标记，没加，则等于没有某种标记。**

以后，javac 编译器，开发工具和其他程序可以用反射来了解你的类及各种元素上有无何种标记，看你有什么标记，就去干相应的事。

标记可以加在包，类，字段，方法，方法的参数以及局部变量上。

一个注解相当于一个类。

看 java.lang 包，可看到 JDK 中提供的最基本的 annotation。
@SuppressWarning(”deprecation”)--->压制警告

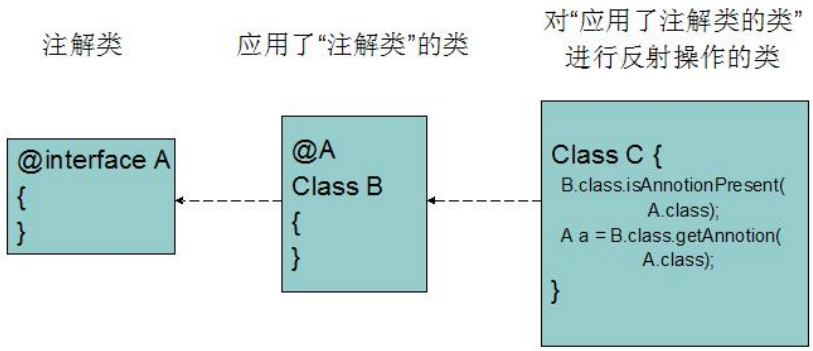
SupressWarning 是告知编译器或开发工具等提示指定的编译器警告；

“deprecation”是告知具体的信息即方法已过时。

@Deprecated（过时的方法，对于不再使用的方法，可能别人或别的地方有调用这个方法，不能删除完事）
直接在刚才的类中增加一个方法，并加上@Deprecated 标注，在另外一个类中调用这个方法。测试一下。
@SuppressWarnings("deprecation") （用这个可以告诉 程序说，我知道调用的方法过时了）
@Override--->提示覆盖（父类方法）
public boolean equals(Reflect other)方法与 HashSet 结合讲解

像 person 类，覆盖父类的 equals 和 hashCode 方法，人家接收的参数是 Object，人们习惯总是传入自己的对象，造成覆盖失败，变成重载！

注解的应用结构图



演示和讲解@Target 元注解

Target（告诉编译器，自定义的注解类可以用在方法还是类....），设置 Target 等于 ElementType.METHOD，原来加在类上的注解就报错了，改为用数组方式设置{ElementType.METHOD,ElementType.TYPE}就可以了。

元注解以及其枚举属性值不用记，只要会看 jdk 提供那几个基本注解的 API 帮助文档的定义或其源代码，按图索骥即可查到，或者直接看 java.lang.annotation 包下面的类。

（自定义注解）示例代码：
@Retention(RetentionPolicy.RUNTIME)//告诉程序说，这个注解要保存到运行时期
@Target({ElementType.METHOD,ElementType.TYPE})//告诉编译器，这个注解可以用在方法上，也可以用在类上

```
public @interface MyAnnotation {
    String color() default "yellow";//默认缺省值为yellow
    String value() ;//不指定
    int [] arrayAttr() default {1,2};//默认为{1,2}
    EnumTest.TrafficLamp lamp() default EnumTest.TrafficLamp.RED;//枚举类
    MetaAnnotation annotationAttr() default @MetaAnnotation("xxx");//属性中加注解，用@。可以在对别的类加注解时，改变值
}
```

**为注解增加基本属性**
**（可以是八种基本数据类型，String ，数组，枚举，注解，Class）**

什么是注解的属性?
一个注解相当于一个胸牌，如果你胸前贴了胸牌，就是传智播客的学生，否则，就不是。如果还想区分出是传智播客哪个班的学生，这时候可以为胸牌在增加一个属性来进行区分。加了属性的标记效果为：@MyAnnotation(color="red")

定义基本类型的属性和应用属性：
在注解类中增加 String color();
@MyAnnotation(color="red")
用反射方式获得注解对应的实例对象后，再通过该对象调用属性对应的方法
MyAnnotation a = (MyAnnotation)AnnotationTest.class.getAnnotation(MyAnnotation.class);
System.out.println(a.color());

可以认为上面这个@MyAnnotation 是 MyAnnotaion 类的一个实例对象
如果注解中有一个名称为 value 的属性，且你只想设置 value 属性（即其他属性都采用默认值或者你只有一个 value 属性），那么可以省略 value=部分，例如：@MyAnnotation("lhm")。

枚举和注解都是特殊的类，不能用 new 创建它们的实例对象，创建枚举的实例对象就是在其中增加元素。

**在程序中如何创建出一个注解的实例对象啊？直接用@放上一个标记即可**

Eg：

```
package july78javaEnhance;

import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Retention(RetentionPolicy.RUNTIME)//元注解：信息的信息就是元信息RUNTIME，保留到运行期间
/**
 * 指示注释类型的注释要保留多久。如果注释类型声明中不存在 Retention 注释，
 * 则保留策略默认为 RetentionPolicy.CLASS。
 */
//@Target(ElementType.METHOD)如果加上这句的话（只能作用于方法），那边就会报错！
public @interface AnnotationDemo12 {//注解

    String color() default "blue";//相当于构造方法一样,如果给了它默认的初值，可以不用再设置（有默认的）
    String value();
    int []age();//数组和枚举类型的注解
    //还有注解类型的注解，暂时没学会
    MetaAnnotation annotation();
    //上面的MetaAnnotation是自己定义的一个注解类型，这样的话对于应用了当前注解的AnnotationDemo11类，就必须写上注解类型的属性
}

package july78javaEnhance;

import java.util.Arrays;

//备注：应用注解后，它的属性你你没有写上，它会给予提醒！missing attribution
@AnnotationDemo12(color = "red",value = "j2ee",age = {1,2,3},annotation = @MetaAnnotation(sex = "男"))//注解加注解
public class AnnotationDemo11 {
    @SuppressWarnings("deprecation")//压缩注解，一个注解就是一个类，用到的一个注解就相当于是调用的实例对象
    @AnnotationDemo12(value = "jase",age = {1,2,3}, annotation = @MetaAnnotation(sex = "女"))//备注：如果别的属性有默认值，只有一个属性需要你设置，那么你就不需要写上全部的
    //赋值表达式，如上直接写上"jase"就行
    public static void main(String[] args) {
```

```
        System.runFinalizersOnExit(true);//表示已经过时的方法,开发工具会给它中间加上一天横线
        /**
         *  boolean isAnnotationPresent(Class<? extends Annotation> annotationClass)
            如果指定类型的注释存在于此元素上，则返回 true，否则返回 false。
         */

    if(AnnotationDemo11.class.isAnnotationPresent(AnnotationDemo12.class)){
            AnnotationDemo12 annocation = (AnnotationDemo12)AnnotationDemo11.class
                    .getAnnotation(AnnotationDemo12.class);//证明这里面有你的注解

            System.out.println(annocation.color());//调用属性

    System.out.println(Arrays.toString(annocation.age()));//将数组打印出来
            System.out.println(annocation.annotation().sex());//相当于调用属性的属性
        }
    }
}
```

## 4、泛型

Jdk 1.5 以前的集合类中存在什么问题
ArrayList collection = new ArrayList();
collection.add(1);
collection.add(1L);
collection.add("abc");
int i = (Integer) collection.get(1);//编译要强制类型转换且运行时出错！
Jdk 1.5 的集合类希望你在定义集合时，明确表示你要向集合中装哪种类型的数据，无法加入指定类型以外的数据
ArrayList<Integer> collection2 = new ArrayList<Integer>();
collection2.add(1);
/*collection2.add(1L);
collection2.add(“abc”);*///这两行代码编译时就报告了语法错误
int i2 = collection2.get(0);//不需要再进行类型转换

**泛型是提供给 javac 编译器使用的，**
**利用反射穿透泛型限制 （暴力反射）**
泛型能绝对保证集合中存入数据都是它限定的类型吗？先看下边的代码

```
package july78javaEnhance;

import java.util.ArrayList;

public class Demo23 {
    public static void main(String[] args) {
        ArrayList<Integer> collection2 = new ArrayList<Integer>();

    System.out.println(collection1.getClass()==collection2.getClas
s());
        collection2.add("真暴力"); //这句会报错
        collection2.getClass().getMethod("add",
Object.class).invoke(collection2, "真暴力");
        System.out.println(collection2.get(0)); //结果却为真暴力
        //已经限制集合中元素的类型为Integer，可用反射却能将String存入，为什
么？ 这是因为泛型是给编译器用的，运行时就没有这些泛型信息了，这叫做"去泛型化"，
所以可以通过反射，获取集合字节码加入非指定的类型。
    }
}
```

**泛型中的？通配符的扩展**
限定通配符的上边界：
正确：Vector<? extends Number> x = new Vector<Integer>();
错误：Vector<? extends Number> x = new Vector<String>();
限定通配符的下边界：
正确：Vector<? super Integer> x = new Vector<Number>();
错误：Vector<? super Integer> x = new Vector<Byte>();
提示：
限定通配符总是包括自己。
?只能用作引用，不能用它去给其他变量赋值
Vector<? extends Number> y = new Vector<Integer>();
Vector<Number> x = y;
上面的代码错误，原理与Vector<Object > x11 = new Vector<String>();相似，只能通过强制类型转换方式来赋值。

Eg：
泛型集合类的综合案例
能写出下面的代码即代表掌握了Java的泛型集合类：
HashMap<String,Integer> hm = new HashMap<String,Integer>();

```
  hm.put("zxx",19);
  hm.put("lis",18);

  Set<Map.Entry<String,Integer>> mes= hm.entrySet();
  for(Map.Entry<String,Integer> me : mes) {
   System.out.println(me.getKey() + ":" + me.getValue());
  }
```

由C++的模板函数引入自定义泛型

如下函数的结构很相似，仅类型不同：

```
int add(int x,int y) {
          return x+y;
    }
float add(float x,float y) {
          return x+y;
   }
double add(double x,double y) {
          return x+y;
   }
```

C++用模板函数解决，只写一个通用的方法，它可以适应各种类型，示意代码如下：

```
   template<class T>
   T add(T x,T y) {
          return (T) (x+y);
   }
```

类型参数的类型推断（花了张老师两天的时间总结）

l 编译器判断范型方法的实际类型参数的过程称为类型推断，类型推断是相对于知觉推断的，其实现方法是一种非常复杂的过程。

l 根据调用泛型方法时实际传递的参数类型或返回值的类型来推断，具体规则如下：

1．当某个类型变量只在整个参数列表中的所有参数和返回值中的一处被应用了，那么根据调用方法时该处的实际应用类型来确定，这很容易凭着感觉推断出来，即直接根据调用方法时传递的参数类型或返回值来决定泛型参数的类型，例如：

swap(new String[3],3,4)　à　static <E> void swap(E[] a, int i, int j)

2．当某个类型变量在整个参数列表中的所有参数和返回值中的多处被应用了，如果调用方法时这多处的实际应用类型都对应同一种类型来确定，这很容易凭着感觉推断出来，例如：

add(3,5)　à static <T> T add(T a, T b)

3．当某个类型变量在整个参数列表中的所有参数和返回值中的多处被应用了，如果调用方法时这多处的实际应用类型对应到了不同的类型，且没有使用返回值，这时候取多个参数中的最大交集类型，例如，下面语句实际对应的类型就是 Number 了，编译没问题，只是运行时出问题：

fill(new Integer[3],3.5f)　à static <T> void fill(T[] a, T v)

4．当某个类型变量在整个参数列表中的所有参数和返回值中的多处被应用了，如果调用方法时这多处的实际应用类型对应到了不同的类型， 并且使用返回值，这时候优先考虑返回值的类型，例如，下面语句实际对应的类型就是 Integer 了，编译将报告错误，将变量 x 的

类型改为 float，对比 eclipse 报告的错误提示，接着再将变量 x 类型改为 Number，则没有了错误：

int x =(3,3.5f)　　à static <T> T add(T a, T b)

5．参数类型的类型推断具有传递性，下面第一种情况推断实际参数类型为 Object，编译没有问题，而第二种情况则根据参数化的 Vector 类实例将类型变量直接确定为 String 类型，编译将出现问题：

copy(new Integer[5],new String[5]) à static <T> void copy(T[] a,T[]　b);

copy(new Vector<String>(), new Integer[5]) à static <T> void copy(Collection<T> a , T[] b);

定义泛型类型

如果类的实例对象中的多处都要用到同一个泛型参数，即这些地方引用的泛型类型要保持同一个实际类型时，这时候就要采用泛型类型的方式进行定义，也就是类级别的泛型，语法格式如下：

public class GenericDao<T> {

private T field1;

public void save(T obj){}

public T getById(int id){}

}

类级别的泛型是根据引用该类名时指定的类型信息来参数化类型变量的，例如，如下两种方式都可以：

GenericDao<String> dao = null;

new genericDao<String>();

注意：

在对泛型类型进行参数化时，类型参数的实例必须是引用类型，不能是基本类型。

当一个变量被声明为泛型时，只能被实例变量、方法和内部类调用，而不能被静态变量和静态方法调用。因为静态成员是被所有参数化的类所共享的，所以静态成员不应该有类级别的类型参数。

问题：类中只有一个方法需要使用泛型，是使用类级别的泛型，还是使用方法级别的泛型？

(Create)、查询(Retrieve)（重新得到数据）、更新(Update)和删除(Delete)几个单词的首字母简写

```
import java.util.Set;

//dao data access object（数据访问对象）--->crud
public class GenericDao<E>  {
    public void add(E x){
    }
    public E findById(int id){
        return null;
    }
```

```
    public void delete(E obj){
    }
    public void delete(int id){
    }
    public void update(E obj){
    }
    public static <E> void update2(E obj){
    }
    public E findByUserName(String name){
        return null;
    }
    public Set<E> findByConditions(String where){
        return null;
    }
}
```

# 1、类加载器

一、概述：

**1、定义：简单说，类加载器就是加载类的工具。**

当出现一个类，用到此类的时候，Java 虚拟机首先将类字节码加载进内存，通常字节码的原始信息放在硬盘上的 classpath 指定的目录下。

2、类加载器作用：将.class 文件中的内容加载进内存进行处理，处理完后的结果就是字节码。

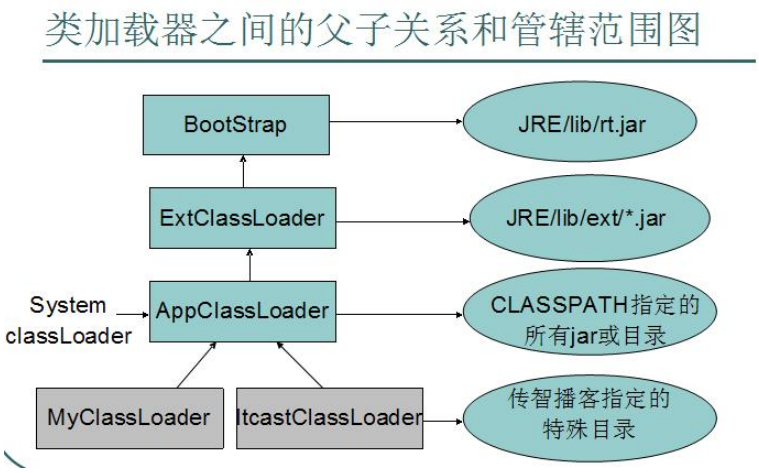
3、默认类加载器：

1）Java 虚拟机中可安装多个类加载器，系统默认的有三个主要的，每个类负责加载特定位置的类：BootStrap、ExtClassLoader、AppClassLoader

2）BootStrap--顶级类加载器：

类加载器本身也是 Java 类，因为它是 Java 类，本身也需要加载器加载，显然必须有第一个类加载器而不是 java 类的，这正是 BootStrap。它是嵌套在 Java 虚拟机内核中的，已启动即出现在虚拟机中，是用 c++写的一段二进制代码。所以不能通过 java 程序获取其名字，获得的只能是 null。

4、Java 虚拟机中的所有类加载器采用子父关系的树形结构进行组织，在实例化每个类加载器对象或默认采用系统类加载器作为其父级类加载器。




## 2、类加载器的委托机制

l 当 Java 虚拟机要加载一个类时，到底派出哪个类加载器去加载呢？
1．首先当前线程的类加载器去加载线程中的第一个类。
2．如果类 A 中引用了类 B，Java 虚拟机将使用加载类 A 的类装载器来加载类 B。
3．还可以直接调用 ClassLoader.loadClass()方法来指定某个类加载器去加载某个类。
l 每个类加载器加载类时，又先委托给其上级类加载器。
1．当所有祖宗类加载器没有加载到类，回到发起者类加载器，还加载不了，则抛 ClassNotFoundException，不是再去找发起者类加载器的儿子，因为没有 getChild 方法，即使有，那有多个儿子，找哪一个呢？
2.对着类加载器的层次结构图和委托加载原理,解释先前将 ClassLoaderTest 输出成 jre/lib/ext 目录下的 itcast.jar 包中后，运行结果为 ExtClassLoader 的原因。

每个 ClassLoader 本身只能分别加载特定位置和目录中的类，但它们可以委托其他的类装载器去加载类，这就是类加载器的委托模式。类装载器一级级委托到 BootStrap 类加载器，当 BootStrap 无法加载当前所要加载的类时，然后才一级级回退到子孙类装载器去进行真正的加载。当回退到最初的类装载器时，如果它自己也不能完成类的装载，那就应报告 ClassNotFoundException 异常。

有一道面试，能不能自己写个类叫 java.lang.System，为了不让我们写 System 类，类加载采用委托机制，这样可以保证爸爸们优先，也就是总是使用爸爸们能找到的类，这样总是使用 java 系统提供的 System。

把先前编写的类加入到 jdk 的 rt.jar 中，会有怎样的效果呢？不行!!!看来是不能随意将自己的 class 文件加入进 rt.jar 文件中的。

编写自己的类加载器(需要再仔细看一遍，不是太懂)

知识讲解：

自定义的类加载器的必须继承 ClassLoader
loadClass 方法（直接继承，省去委托机制的编写）与 findClass 方法（覆盖这个就行了）
defineClass 方法
编程步骤：

编写一个对文件内容进行简单加密的程序。

编写了一个自己的类装载器，可实现对加密过的类进行装载和解密。

编写一个程序调用类加载器加载类，在源程序中不能用该类名定义引用变量，因为编译器无法识别这个类。程序中可以除了使用 ClassLoader.load 方法之外，还可以使用设置线程的上下文类加载器或者系统类加载器，然后再使用 Class.forName。

实验步骤：

对不带包名的 class 文件进行加密，加密结果存放到另外一个目录，例如： java MyClassLoader MyTest.class F:\itcast

运行加载类的程序，结果能够被正常加载，但打印出来的类装载器名称为 AppClassLoader：java MyClassLoader MyTest F:\itcast

用加密后的类文件替换 CLASSPATH 环境下的类文件，再执行上一步操作就出问题了，错误说明是 AppClassLoader 类装载器装载失败。

删除 CLASSPATH 环境下的类文件，再执行上一步操作就没问题了。

```
import java.io.*;
import java.lang.reflect.*;
public class MyClassLoader extends ClassLoader
{
   private String path = null;
   public MyClassLoader(String path) throws Exception//检查文件是否存在
   {
      File f = new File(path);
      if(!f.isDirectory())
      {
         throw new RuntimeException(path + " is not a directory");
      }
      this.path = path;
   }

   public Class findClass(String name) //throws Exception //为什么不能抛出
   {
      try
      {
         File f = new File(path,name.substring(name.lastIndexOf('.')+1) + ".class");
         FileInputStream fis = new FileInputStream(f);
         ByteArrayOutputStream bos = new ByteArrayOutputStream();
         cypher(fis,bos);
```

```
            byte [] buf = bos.toByteArray();
            fis.close();
            bos.close();
            return defineClass(name,buf,0,buf.length);
        }catch(Exception e)
        {
            throw new ClassNotFoundException(name + " is not found!");
        }
        return null;
    }


    public static void cypher(InputStream istream,OutputStream
ostream) throws Exception
    {
        //下面这段代码可能遇到255的字节，当成byte就成了-1
        /*byte b = 0;
        while((b = (byte)istream.read()) != -1)
        {
            ostream.write(b ^ 0xff);
        }*/

        int b = 0;
        while((b = istream.read()) != -1)
        {
            ostream.write(((byte)b) ^ 0xff);
        }
    }

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

        //下面省略了错误检查
        if(!args[0].endsWith("class"))
        {
            ClassLoader loader = new MyClassLoader(args[1]);
            Class cls = loader.loadClass(args[0]);

            /*
            让自定义类继承Date类

System.out.println(cls.getClassLoader().getClass().getName());
                java.util.Date d = (java.util.Date)cls.newInstance();
                System.out.println(d.toString());
```

```
            */

            //Method m = cls.getMethod("test",null);//在jdk1.5中报警告，为什么？
            Method m = cls.getMethod("test");
            //m.invoke(cls.newInstance(),null);
            m.invoke(cls.newInstance());
            //((Test)cls.newInstance()).test();
            return;
        }
        else
        {
            FileInputStream fis = new FileInputStream(args[0]);
            File f = new File(args[1], new File(args[0]).getName());//不用检查目录最后是否有目录分割符
            FileOutputStream fos = new FileOutputStream(f);
            cypher(fis,fos);
            fis.close();
            fos.close();
        }
    }
}

//类加载器不能加载这种非public的类
/*
Exception in thread "main" java.lang.IllegalAccessException: Class MyClassLoader
 can not access a member of class MyTest with modifiers ""
*/
/*
class MyTest
{
    public void test()
    {
        System.out.println("hello,www.it315.org");
    }
}
*/
```

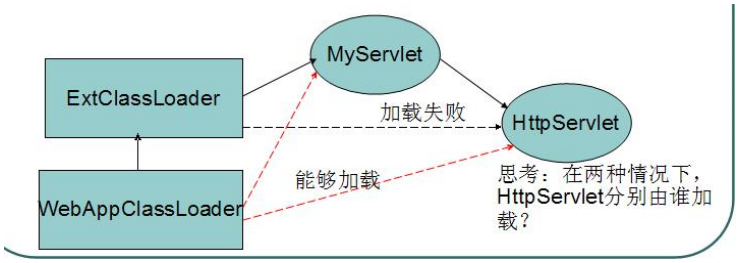
## 3、一个类加载器的高级问题分析

编写一个能打印出自己的类加载器名称和当前类加载器的父子结构关系链的 MyServlet，正常发布后，看到打印结果为 WebAppClassloader。

把 MyServlet.class 文件打 jar 包，放到 ext 目录中，重启 tomcat，发现找不到 HttpServlet 的错误。

把 servlet.jar 也放到 ext 目录中，问题解决了，打印的结果是 ExtclassLoader 。

父级类加载器加载的类无法引用只能被子级类加载器加载的类，原理如下图：




## 4、代理的概念与作用

1、引入:

为已存在的多个具有相同接口的目标类的各个方法增加一些系统功能，例如，异常处理、日志、计算方法的运行时间、事务管理、等等，你准备如何做？

编写一个与目标类具有相同接口的代理类，代理类的每个方法调用目标类的相同方法，并在调用方法时加上系统功能的代码。

2、代理类的优点：

如果采用工厂模式和配置文件的方式进行管理，则不需要修改客户端程序，在配置文件中配置是使用目标类、还是代理类，这样以后很容易切换，譬如，想要日志功能时就配置代理类，否则配置目标类，**这样，增加系统功能很容易，以后运行一段时间后，又想去掉系统功能也很容易。**

目标类：

```
class X{
   void sayHello(){
syso:Hello;
```

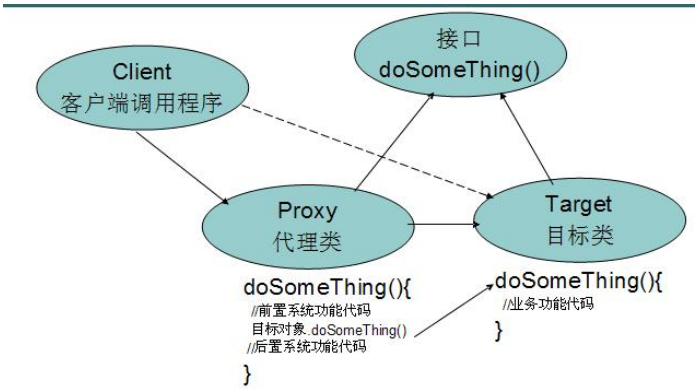
代理类：

```
Xproxy{
 void sayHello(){
startTime
```

}　　X. sayHello();

}　　**endTime;**}}

一般用接口来引用其子类，如：Collectioncoll = new ArrayList();




**AOP**

系统中存在交叉业务，一个交叉业务就是要切入到系统中的一个方面，如下所示：

安全　　事务　　日志

StudentService　------|----------|-----------|------------

CourseService　------|----------|-----------|------------

MiscService　------|----------|-----------|------------

用具体的程序代码描述交叉业务：

method1　　method2　　method3

{　　{　　{

----------切面

....　　....　　.....

----------切面

}　　}　　}

交叉业务的编程问题即为面向方面的编程（Aspect oriented program ,简称 AOP），AOP 的目标就是要使交叉业务模块化。可以采用将切面代码移动到原始方法的周围，这与直接在方法中编写切面代码的运行效果是一样的，如下所示：

----------切面

func1　　func2　　func3

{　　{　　{

....　　....　　.....

}　　}　　}

----------切面

使用代理技术正好可以解决这种问题，代理是实现 AOP 功能的核心和关键技术。

**安全，事务，日志等功能要贯穿到好多个模块中，所以，它们就是交叉业务**
重要原则：不要把供货商暴露给你的客户

## 5、动态代理技术

1、要为系统中的各种接口的类增加代理功能，那将需要太多的代理类，全部采用静态代理方式，将是一件非常麻烦的事情！写成百上千个代理类，是不是太累！
2、JVM 可以在运行期动态生成出类的字节码，这种动态生成的类往往被用作代理类，即动态代理类。
3、JVM 生成的动态类必须实现一个或多个接口，所以，JVM 生成的动态类只能用作具有相同接口的目标类的代理。
4、CGLIB 库可以动态生成一个类的子类，一个类的子类也可以用作该类的代理，所以，如果要为一个没有实现接口的类生成动态代理类，那么可以使用 CGLIB 库。
5、代理类的各个方法中通常除了要调用目标的相应方法和对外返回目标返回的结果外，还可以在代理方法中的如下四个位置加上系统功能代码：

1）在调用目标方法之前
2）在调用目标方法之后
3）在调用目标方法前后
4）在处理目标方法异常的 catch 块中

## 6、分析 JVM 动态生成的类

总结思考：让 jvm 创建动态类及其实例对象，需要给它提供哪些信息？
三个方面：
1、生成的类中有哪些方法，通过让其实现哪些接口的方式进行告知；
2、产生的类字节码必须有个一个关联的类加载器对象；
3、生成的类中的方法的代码是怎样的，也得由我们提供。把我们的代码写在一个约定好了接口对象的方法中，把对象传给它，它调用我的方法，即相当于插入了我的代码。提供执行代码的对象就是那个 InvocationHandler 对象，它是在创建动态类的实例对象的构造方法时传递进去的。在上面的 InvocationHandler 对象的 invoke 方法中加一点代码，就可以看到这些代码被调用运行了。
用 newProxyInstance（）方法直接一步就创建出代理对象。

总结分析动态代理类的统计原理和结构：

1、怎样将目标传进去：

1）直接在 InvocationHandler 实现类中创建目标类的实例对象，可看运行效果和加入日志代码，但是毫无意义。

2）为 InvocationHandler 实现类注入目标的实例对象，不能采用匿名内部类的形式了。

3）让匿名内部类的 InvocationHandler 实现类访问外面的方法中的目标类实例对象的 final 类型的引用变量。

**2、动态代理的工作原理：**

1）Client(客户端)调用代理，代理的构造方法接受一个 InvocationHandler，client 调用代理的各个方法，代理的各个方法请求转发给刚才通过构造方法传入的 handler 对象，又把各请求分发给目标的相应的方法。就是将 handler 封装起来，其中 this 引用了当前的放(发来什么请求就接受哪个方法)。
猜想分析动态生成的类的内部代码：
1、动态生成的类实现了 Collection 接口（可以实现若干接口），生成的类有 Collection 接口中的所有方法和一个如下接受 InvocationHandler 参数的构造方法。
2、构造方法接受一个 InvocationHandler 对象，接受对象了要干什么用呢？该方法内部的代码会是怎样的呢？
实现 Collection 接口的动态类中的各个方法的代码又是怎样的呢？InvocationHandler 接口中定义的 invoke 方法接受的三个参数又是什么意思？图解说明如下：

分析为什么动态类的实例对象的 getClass()方法返回了正确结果呢？
为何动态类的实例对象的 getClass()方法返回了正确结果，而没调用 invoke 方法：
因为代理类从 Object 上继承了许多方法，其中只对三个方法（hashCode、equals 和 toString）进行开发，委托给 handler 去自行处理，对于它身上其他方法不会交给代理类去实现，所以对于 getClass()方法，还是由 Object 本身实现的。即 proxy3.getClass()，该是什么结果还是什么结果，并不会交给 invoke 方法处理。

## 7、综合示例代码（用到三个. java 文件，张老师的源码，自己加了注释，自己敲精简版的代理）：

```
package javaenhance.src.cn.itcast.day3;

import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import java.util.ArrayList;
import java.util.Collection;
public class ProxyTest {

    /**
     * @param args
     */
    public static void main(String[] args) throws Exception{
        // TODO Auto-generated method stub
```

```
        //接收两个参数，一个是后边参数的字节码的加载器，一个是所要实现代理的接口的字节码
        Class clazzProxy1 = Proxy.getProxyClass(Collection.class.getClassLoader(), Collection.class);
        System.out.println(clazzProxy1.getName());//打印代理的名字 $Proxy0

        System.out.println("----------begin constructors list----------");
        /*想打印出如下格式：
         * $Proxy0()
        $Proxy0(InvocationHandler,int)*/
        Constructor[] constructors = clazzProxy1.getConstructors();//获取代理的构造函数
        for(Constructor constructor : constructors){
            String name = constructor.getName();//获取代理的构造函数的name
            StringBuilder sBuilder = new StringBuilder(name);
            sBuilder.append('(');
            Class[] clazzParams = constructor.getParameterTypes();//获取代理的构造函数的参数
            for(Class clazzParam : clazzParams){
                sBuilder.append(clazzParam.getName()).append(',');
            }
            if(clazzParams!=null && clazzParams.length != 0)//稳妥的判断是否是一个参数，不是就删掉最后的，
                sBuilder.deleteCharAt(sBuilder.length()-1);
            sBuilder.append(')');
            System.out.println(sBuilder.toString());
//$Proxy0(java.lang.reflect.InvocationHandler)
        }

        System.out.println("----------begin methods list----------");
        /*$Proxy0()
        $Proxy0(InvocationHandler,int)*/
        Method[] methods = clazzProxy1.getMethods();//获取代理身上的方法
        for(Method method : methods){
            String name = method.getName();
            StringBuilder sBuilder = new StringBuilder(name);
            sBuilder.append('(');
            Class[] clazzParams = method.getParameterTypes();
```

```
            for(Class clazzParam : clazzParams){
                sBuilder.append(clazzParam.getName()).append(',');
            }
            if(clazzParams!=null && clazzParams.length != 0)
                sBuilder.deleteCharAt(sBuilder.length()-1);
            sBuilder.append(')');
            System.out.println(sBuilder.toString());
        }

        System.out.println("----------begin create instance
object----------");
        //Object obj = clazzProxy1.newInstance();
              //方式一：通过接口的子类创建对象
        Constructor constructor =
clazzProxy1.getConstructor(InvocationHandler.class);
        //获取代理身上的构造函数
        //创建内部类MyInvocationHandler1，目的是传递给代理的构造器
        class MyInvocationHandler1 implements InvocationHandler{

            public Object invoke(Object proxy, Method method, Object[]
args)
                    throws Throwable {
                // TODO Auto-generated method stub
                return null;
            }

        }
        //方式二：匿名内部类
        Collection proxy1 = (Collection)constructor.newInstance(new
MyInvocationHandler1());

        System.out.println(proxy1);//没有错误
        proxy1.clear();//没有错误
        //proxy1.size();报错，因为，代理调用size方法，其实是调用了
MyInvocationHandler1中的invoke，他的返回值是null
        //System.out.println("111111111111111");//调试用的

        //用了匿名内部类的方法实现
        Collection proxy2 = (Collection)constructor.newInstance(new
InvocationHandler(){

            public Object invoke(Object proxy, Method method, Object[]
args)
                    throws Throwable {
```

```
                return null;
            }

        });



        /*下边这部分代码非常重要和精辟*/
        final ArrayList target = new ArrayList();
        Collection    proxy3    =    (Collection)getProxy(target,new
MyAdvice());
        proxy3.add("zxx");
        proxy3.add("lhm");
        proxy3.add("bxd");
        System.out.println(proxy3.size());
        System.out.println(proxy3.getClass().getName());
    }

    private static Object getProxy(final Object target,final Advice
advice) {

        //方式三，newProxyInstance这个方法需要三个参数，可以直接创建target
的代理对象
        Object proxy3 = Proxy.newProxyInstance(
                target.getClass().getClassLoader(),
                /*new Class[]{Collection.class},*/

                //获取target上的接口
                target.getClass().getInterfaces(),
                new InvocationHandler(){

                    public Object invoke(Object proxy, Method method,
Object[] args)
                            throws Throwable {

                        /*long beginTime = System.currentTimeMillis();
                        Object retVal = method.invoke(target, args);
                        long endTime = System.currentTimeMillis();
                        System.out.println(method.getName() + " running
time of " + (endTime - beginTime));
                        return retVal;*/

                        //把上边的代码封装到一个类中，让后调用该类的方法，就实
```

现了方法的封装

```
                    advice.beforeMethod(method);
                    Object retVal = method.invoke(target, args);
                    advice.afterMethod(method);
                    return retVal;

                }
            }
            );
        return proxy3;
    }

}

/*$Proxy0
----------begin constructors list----------
$Proxy0(java.lang.reflect.InvocationHandler)
----------begin methods list----------
hashCode()
equals(java.lang.Object)
toString()
add(java.lang.Object)
contains(java.lang.Object)
isEmpty()
size()
toArray()
toArray([Ljava.lang.Object;)
addAll(java.util.Collection)
iterator()
remove(java.lang.Object)
clear()
containsAll(java.util.Collection)
removeAll(java.util.Collection)
retainAll(java.util.Collection)
getProxyClass(java.lang.ClassLoader,[Ljava.lang.Class;)
newProxyInstance(java.lang.ClassLoader,[Ljava.lang.Class;,java.la
ng.reflect.InvocationHandler)
isProxyClass(java.lang.Class)
getInvocationHandler(java.lang.Object)
getClass()
notify()
notifyAll()
wait(long)
wait(long,int)
```

```
wait()
----------begin create instance object----------
null
到传智播客来学习啦!
从传智播客毕业上班啦!
add running time of 0
到传智播客来学习啦!
从传智播客毕业上班啦!
add running time of 0
到传智播客来学习啦!
从传智播客毕业上班啦!
add running time of 0
到传智播客来学习啦!
从传智播客毕业上班啦!
size running time of 0
3
$Proxy1*/


-----------------------------------------------
import java.lang.reflect.Method;

public interface Advice {
    void beforeMethod(Method method);
    void afterMethod(Method method);
}

-----------------------------------------------
package javaenhance.src.cn.itcast.day3;

import java.lang.reflect.Method;
//创建实现Advice接口的子类
public class MyAdvice implements Advice {
    long beginTime = 0;
    public void afterMethod(Method method) {
        // TODO Auto-generated method stub
        System.out.println("从传智播客毕业上班啦! ");
        long endTime = System.currentTimeMillis();
        System.out.println(method.getName() + " running time of " +
(endTime - beginTime));

    }

    public void beforeMethod(Method method) {
```

```
        // TODO Auto-generated method stub
        System.out.println("到传智播客来学习啦！");
        beginTime = System.currentTimeMillis();
    }
}
```

## 8、实现类似 spring 的可配置的 AOP 框架

一、工厂类 BeanFactory：

1、工厂类 BeanFactory 负责创建目标类或代理类的实例对象，并通过配置文件实现切换。

2、getBean 方法根据参数字符串返回一个相应的实例对象，如果参数字符串在配置文件中对应的类名不是 ProxyFactoryBean，则直接返回该类的实例对象，否则返回该类示例对象的 getProxy 方法返回的对象。

3、BeanFactory 的构造方法接收代表配置文件的输入流对象的配置文件格式如下：

#xxx=java.util.ArrayList

xxx=cn.itcast.test3.aopframework.ProxyFactoryBean

xxx.advice=cn.itcast.test3.MyAdvice

xxx.target=java.util. ArrayList
注意：其中的#代表注释当前行。

4、**ProxyFactoryBean 充当封装成动态的工厂，**需为工厂提供的配置参数信息包括：

目标（target）

通告（advice）

5、BeanFactory 和 ProxyFactoryBean：

1）BeanFactory 是一个纯粹的 bean 工程，就是创建 bean 即相应的对象的工厂。

2）ProxyfactoryBean 是 BeanFactory 中的一个特殊的 Bean，是创建代理的工厂。

二、实现类似 spring 的可配置的 AOP 框架的思路：

1、创建 BeanFactory 类：

1）构造方法：接受一个配置文件，通过 Properties 对象加载 InputStream 流对象获得。

2）创建 getBean(String name)方法，接收 Bean 的名字，从上面加载后的对象获得。

3）通过其字节码对象创建实例对象 bean。

4）判断 bean 是否是特殊的 Bean 即 ProxyFactoryBean，如果是，就要创建代理类，并设置目标和通告，分别得到各自的实例对象，并返回代理类实例对象。如果不是在返回普通类的实例对象。

2、创建 ProxyFactoryBean(接口)，此处用类做测试，其中有一个 getProxy 方法，用于获得代理类对象。

3、对配置文件进行配置，如上面配置一样。

4、作一个测试类：AopFrameworkTest 进行测试。

```
//创建BeanFactory类
package cn.itcast.test3.aopframework;
import java.io.IOException;
import java.io.InputStream;
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import java.util.Properties;
public class BeanFactory {
    Properties prop = new Properties();
    //创建对象时需要传入一个配置文件中的数据，所以需要在构造方法中接受一个参数
    public BeanFactory(InputStream ips) {
        try {
            //将配置文件加载进来
            prop.load(ips);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    //创建getBean方法，通过配置文件中的名字获取bean对象
    public Object getBean(String name){
        //从配置文件中读取类名
        String className = prop.getProperty(name);
        Object bean = null;
        try {
            //由类的字节码获取对象
```

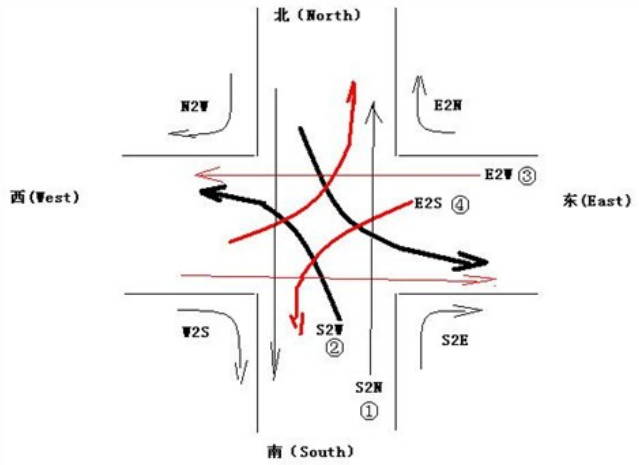
```
            Class clazz = Class.forName(className);
            bean = clazz.newInstance();
        } catch (Exception e) {
            e.printStackTrace();
        }
        //判断bean是特殊的bean即ProxyFactoryBean还是普通的bean
        if(bean instanceof ProxyFactoryBean){
            Object proxy = null;
            try {
                //是ProxyFactoryBean的话，强转，并获取目标和通告
                ProxyFactoryBean proxyFactoryBean =
(ProxyFactoryBean)bean;
                //获取advice和target
                Advice advice =
(Advice)Class.forName(prop.getProperty(name +
".advice")).newInstance();
                Object target = Class.forName(prop.getProperty(name +
".target")).newInstance();
                //设置目标和通告
                proxyFactoryBean.setAdvice(advice);
                proxyFactoryBean.setTarget(target);
                //通过类ProxyFactoryBean（开发中是作为接口存在）中获得proxy
对象
                proxy = proxyFactoryBean.getProxy();
            } catch (Exception e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
            //是ProxyFactoryBean的话，返回proxy对象
            return proxy;
        }
        //否则返回普通bean对象
        return bean;
    }
}

//创建ProxyFactoryBean类
package cn.itcast.test3.aopframework;
import java.lang.reflect.*;
import cn.itcast.test3.Advice;
public class ProxyFactoryBean {
    private Object target;
    private Advice advice;
    public Object getTarget() {
```

```
        return target;
    }
    public void setTarget(Object target) {
        this.target = target;
    }
    public Advice getAdvice() {
        return advice;
    }
    public void setAdvice(Advice advice) {
        this.advice = advice;
    }
    public Object getProxy() {
        Object proxy = Proxy.newProxyInstance(
                target.getClass().getClassLoader(),
                //这里的接口要和target实现相同的接口
                target.getClass().getInterfaces(),
                new InvocationHandler() {
                    public Object invoke(Object proxy, Method method,
Object[] args)
                            throws Throwable {
                        //通过契约，使用其方法--before和after方法
                        advice.beforeMethod(method);
                        Object value = method.invoke(target, args);
                        advice.afterMethod(method);
                        return value;
                    }
                }
                );
        return proxy;
    }
}
//创建测试类AopFrameworkTest
package cn.itcast.test3.aopframework;
import java.io.InputStream;
public class AopFramewrorkTest {
    public static void main(String[] args)throws Exception {
        //读取配置文件的数据
        InputStream ips =

AopFramewrorkTest.class.getResourceAsStream("config.property");
        //获取bean对象
        Object bean = new BeanFactory(ips).getBean("xxx");
        System.out.println(bean.getClass().getName());
    }
```

}

# 黑马程序员——面试题之交通灯管理系统

需求：
交通灯管理系统的项目需求
Ø 异步随机生成按照各个路线行驶的车辆。
例如：
由南向而来去往北向的车辆 ---- 直行车辆
由西向而来去往南向的车辆 ---- 右转车辆
由东向而来去往南向的车辆 ---- 左转车辆
。。。
Ø 信号灯忽略黄灯，只考虑红灯和绿灯。
Ø 应考虑左转车辆控制信号灯，右转车辆不受信号灯控制。
Ø 具体信号灯控制逻辑与现实生活中普通交通灯控制逻辑相同，不考虑特殊情况下的控制逻辑。
注：南北向车辆与东西向车辆交替放行，同方向等待车辆应先放行直行车辆而后放行左转车辆。
Ø 每辆车通过路口时间为 1 秒（提示：可通过线程 Sleep 的方式模拟）。
Ø 随机生成车辆时间间隔以及红绿灯交换时间间隔自定，可以设置。
Ø 不要求实现 GUI，只考虑系统逻辑实现，可通过 Log 方式展现程序运行结果。




## 1、面向对象的分析与设计

1、每条路线上都会出现多辆车，路线上要随机增加新的车，在灯绿期间还要每秒钟减少一辆车。

设计一个 Road 类来表示路线，每个 Road 对象代表一条路线，总共有 12 条路线，即系统中总共要产生 12 个 Road 实例对象。

每条路线上随机增加新的车辆，**增加到一个集合中保存。**

**每条路线每隔一秒都会检查控制本路线的灯是否为绿，是则将本路线保存车的集合中的第一辆车移除，即表示车穿过了路口。**

**每条路线每隔一秒都会检查控制本路线的灯是否为绿，一个灯由绿变红时，应该将下一个方向的灯变绿。**

2、设计一个 Lamp 类来表示一个交通灯，每个交通灯都维护一个状态：亮（绿）或不亮（红），每个交通灯要有变亮和变黑的方法，并且能返回自己的亮黑状态。

总共有 12 条路线，所以，系统中总共要产生 12 个交通灯。右拐弯的路线本来不受灯的控制，但是为了让程序采用统一的处理方式，故假设出有四个右拐弯的灯，只是这些灯为常亮状态，即永远不变黑。

3、除了右拐弯方向的其他 8 条路线的灯，它们是两两成对的，可以归为 4 组，所以，在编程处理时，只要从这 4 组中各取出一个灯，对这 4 个灯依次轮询变亮，与这 4 个灯方向对应的灯则随之一同变化，因此 Lamp 类中要有一个变量来记住自己相反方向的灯，在一个 Lamp 对象的变亮和变黑方法中，将对应方向的灯也变亮和变黑。每个灯变黑时，都伴随者下一个灯的变亮，Lamp 类中还用一个变量来记住自己的下一个灯。

4、无论在程序的什么地方去获得某个方向的灯时，每次获得的都是同一个实例对象，所以 Lamp 类改用枚举来做显然具有很大的方便性，永远都只有代表 12 个方向的灯的实例对象。

设计一个 LampController 类，它定时让当前的绿灯变红。

## 2、Road 类的编写

每个 Road 对象都有一个 name 成员变量来代表方向，有一个 vehicles 成员变量来代表方向上的车辆集合。

在 Road 对象的构造方法中启动一个线程每隔一个随机的时间向 vehicles 集合中增加一辆车（用一个“路线名_id”形式的字符串进行表示）。

在 Road 对象的构造方法中启动一个定时器，每隔一秒检查该方向上的灯是否为绿，是则打印车辆集合和将集合中的第一辆车移除掉

```
import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

/**
 * 每个Road对象代表一条路线，总共有12条路线，即系统中总共要产生12个Road实例对
```

```
象。
 * 每条路线上随机增加新的车辆，增加到一个集合中保存。
 * 每条路线每隔一秒都会检查控制本路线的灯是否为绿，是则将本路线保存车的集合中的
第一辆车移除，即表示车穿过了路口。
 * @author 张孝祥 www.it315.org
 *
 */
public class Road {
    private List<String> vechicles = new ArrayList<String>();

    private String name =null;
/*在这个构造函数中,传回哪个方向的车,
先开启一个线程池用于产生车辆,一个定时器用于观察交通灯状态*/
    public Road(String name){
        this.name = name;

        //模拟车辆不断随机上路的过程
       //使用线程池，通过产生单个线程的方法，创建一个线程池
        ExecutorService pool = Executors.newSingleThreadExecutor();
____//
        pool.execute(new Runnable(){
            public void run(){
                for(int i=1;i<1000;i++){
                    try {
                        Thread.sleep((new Random().nextInt(10) + 1) *
1000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    vechicles.add(Road.this.name + "_" + i);
                }
            }

        });

        //每隔一秒检查对应的灯是否为绿，是则放行一辆车
____ //产生一个单线程，创建定时器
        ScheduledExecutorService                timer                =
Executors.newScheduledThreadPool(1);
        timer.scheduleAtFixedRate(
                new Runnable(){
                    public void run(){
____ //判断路上是否有车，有则进行相应的操作
                        if(vechicles.size()>0){
```

```
                        boolean                lighted                =
Lamp.valueOf(Road.this.name).isLighted();
 //每隔1秒让车通行，通行前要先判断灯是否亮，亮了才能通行，即从集合中移除
                        if(lighted){
                            System.out.println(vechicles.remove(0)
+ " is traversing !");
                        }
                    }

                }
            },
            1,
            1,
            TimeUnit.SECONDS);

    }
}
```

## 3、Lamp 类的编写

系统中有 12 个方向上的灯，在程序的其他地方要根据灯的名称就可以获得对应的灯的实例对象，综合这些因素，**将 Lamp 类用 java5 中的枚举形式定义更为简单。**

1、**每个 Lamp 对象中的亮黑状态用 lighted 变量表示**，选用 S2N、S2W、E2W、E2N 这四个方向上的 Lamp 对象依次轮询变亮，Lamp 对象中还要有一个 oppositeLampName 变量来表示它们相反方向的灯，再用一个 nextLampName 变量来表示此灯变亮后的下一个变亮的灯。这三个变量用构造方法的形式进行赋值，因为枚举元素必须在定义之后引用，所以无法再构造方法中彼此相互引用，所以，相反方向和下一个方向的灯用字符串形式表示。

2、增加让 Lamp 变亮和变黑的方法：light 和 blackOut，对于 S2N、S2W、E2W、E2N 这四个方向上的 Lamp 对象，**这两个方法内部要让相反方向的灯随之变亮和变黑，blackOut 方法还要让下一个灯变亮。**

3、除了 S2N、S2W、E2W、E2N 这四个方向上的 Lamp 对象之外，其他方向上的 Lamp 对象的 nextLampName 和 oppositeLampName 属性设置为 null 即可，并且 S2N、S2W、E2W、E2N 这四个方向上的 Lamp 对象的 nextLampName 和 oppositeLampName 属性必须设置为 null，以便防止 light 和 blackOut 进入死循环。

代码：

```
/**
 * 每个Lamp元素代表一个方向上的灯，总共有12个方向，所有总共有12个Lamp元素。
 * 有如下一些方向上的灯,每两个形成一组，一组灯同时变绿或变红，所以，
 * 程序代码只需要控制每组灯中的一个灯即可：
 * s2n,n2s
 * s2w,n2e
 * e2w,w2e
```

```
 * e2s,w2n
 * s2e,n2w
 * e2n,w2s
 * 上面最后两行的灯是虚拟的，由于从南向东和从西向北、以及它们的对应方向不受红绿灯的控制，
 * 所以，可以假想它们总是绿灯。
 * @author 张孝祥 www.it315.org
 *
 */
/**/

public enum Lamp {
    /*每个枚举元素各表示一个方向的控制灯*/

S2N("N2S","S2W",false),S2W("N2E","E2W",false),E2W("W2E","E2S",false),E2S("W2N","S2N",false),
    /*下面元素表示与上面的元素的相反方向的灯，它们的"相反方向灯"和"下一个灯"应忽略不计！*/

N2S(null,null,false),N2E(null,null,false),W2E(null,null,false),W2N(null,null,false),
    /*由南向东和由西向北等右拐弯的灯不受红绿灯的控制，所以，可以假想它们总是绿灯*/

S2E(null,null,true),E2N(null,null,true),N2W(null,null,true),W2S(null,null,true);

    private Lamp(String opposite,String next,boolean lighted){
        this.opposite = opposite;
        this.next = next;
        this.lighted = lighted;
    }


    /*当前灯是否为绿*/
    private boolean lighted;
    /*与当前灯同时为绿的对应方向*/
    private String opposite;
    /*当前灯变红时下一个变绿的灯*/
    private String next;
    //灯的判断是否亮的方法
    public boolean isLighted(){
        return lighted;
    }
```

```
    /**让这个方向的等亮起来
     * 某个灯变绿时，它对应方向的灯也要变绿
     */
    public void light(){
        this.lighted = true;
        if(opposite != null){
            Lamp.valueOf(opposite).light();
        }
        System.out.println(name() + " lamp is green，下面总共应该有6个
方向能看到汽车穿过！");

    }

    /**
     * 某个灯变红时，对应方向的灯也要变红，并且下一个方向的灯要变绿
     * @return 下一个要变绿的灯
     */
    public Lamp blackOut(){
        this.lighted = false;
        if(opposite != null){
            Lamp.valueOf(opposite).blackOut();
        }

        Lamp nextLamp= null;
        if(next != null){
____ //当前灯变绿了，让对应的灯也变绿
            nextLamp = Lamp.valueOf(next);
            System.out.println("绿灯从" + name() + "-------->切换为" +
next);
            nextLamp.light();
        }
        return nextLamp;
    }
}
```

## 4、LampController 类的编写

```
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;
```

```
public class LampController {
    private Lamp currentLamp;

    public LampController(){
        //刚开始让由南向北的灯变绿;
        currentLamp = Lamp.S2N;
        currentLamp.light();

        /*每隔10秒将当前绿灯变为红灯，并让下一个方向的灯变绿*/
        ScheduledExecutorService timer =
Executors.newScheduledThreadPool(1);
        timer.scheduleAtFixedRate(
                new Runnable(){
                    public  void run(){
                        System.out.println("来啊");
                        currentLamp = currentLamp.blackOut();
                }
                },
                10,
                10,
                TimeUnit.SECONDS);
    }
```

## 5、MainClass 类的编写

**1、用 for 循环创建出代表 12 条路线的对象。**
**2、接着再获得 LampController 对象并调用其 start 方法。**

```
public class MainClass {

    /**
     * @param args
     */
    public static void main(String[] args) {

        /*产生12个方向的路线*/
        String [] directions = new String[]{

"S2N","S2W","E2W","E2S","N2S","N2E","W2E","W2N","S2E","E2N","N2W",
"W2S"
        };
        for(int i=0;i<directions.length;i++){
```

```
            new Road(directions[i]);
        }

        /*产生整个交通灯系统*/
        new LampController();
    }

}
```

打印结果：

```
/*N2S lamp is green-----有六个方向的车经过
    S2N lamp is green-----有六个方向的车经过
    N2S_1 is traversing !
    S2N_1 is traversing !
    N2W_1 is traversing !
    S2N_2 is traversing !
    E2N_1 is traversing !
    S2E_1 is traversing !
    S2N_3 is traversing !
    N2S_2 is traversing !
    S2N_4 is traversing !
    开始跑吧
    绿灯从S2N-------->切换为S2W
    N2E lamp is green-----有六个方向的车经过
    S2W lamp is green-----有六个方向的车经过
    W2S_1 is traversing !
    S2E_2 is traversing !
    N2W_2 is traversing !
    S2W_1 is traversing !
    W2S_2 is traversing !
    N2E_1 is traversing !
    N2E_2 is traversing !
    N2E_3 is traversing !
    S2W_2 is traversing !
    W2S_3 is traversing !
    E2N_2 is traversing !
    S2E_3 is traversing !
    E2N_3 is traversing !
    S2E_4 is traversing !
    开始跑吧
    绿灯从S2W-------->切换为E2W
    W2E lamp is green-----有六个方向的车经过
    E2W lamp is green-----有六个方向的车经过
    E2W_1 is traversing !
    N2W_3 is traversing !
```

```
E2W_2 is traversing !
W2E_1 is traversing !
E2W_3 is traversing !
W2E_2 is traversing !
W2S_4 is traversing !
E2W_4 is traversing !
N2W_4 is traversing !
W2E_3 is traversing !
S2E_5 is traversing !
E2N_4 is traversing !
E2W_5 is traversing !
W2S_5 is traversing !
N2W_5 is traversing !
W2E_4 is traversing !
*/
```

# 黑马程序员——面试题之银行业务系统

## 1、需求

模拟实现银行业务调度系统逻辑，具体需求如下：

- 银行内有 6 个业务窗口，1 - 4 号窗口为普通窗口，5 号窗口为快速窗口，6 号窗口为 VIP 窗口。
- 有三种对应类型的客户：VIP 客户，普通客户，快速客户（办理如交水电费、电话费之类业务的客户）。
- 异步随机生成各种类型的客户，生成各类型用户的概率比例为：
- VIP 客户 ：普通客户 ：快速客户 = 1 ：6 ：3。
- 客户办理业务所需时间有最大值和最小值，在该范围内随机设定每个 VIP 客户以及普通客户办理业务所需的时间，快速客户办理业务所需时间为最小值（提示：办理业务的过程可通过线程 Sleep 的方式模拟）。
- 各类型客户在其对应窗口按顺序依次办理业务。
- 当 VIP（6 号）窗口和快速业务（5 号）窗口没有客户等待办理业务的时候，这两个窗口可以处理普通客户的业务，而一旦有对应的客户等待办理业务的时候，则优先处理对应客户的业务。
- 随机生成客户时间间隔以及业务办理时间最大值和最小值自定，可以设置。
- 不要求实现 GUI，只考虑系统逻辑实现，可通过 Log 方式展现程序运行结果。

## 2、需求分析

面向对象的分析与设计

1、有三种对应类型的客户：VIP 客户，普通客户，快速客户 ，异步随机生成各种类型的客户，各类型客户在其对应窗口按顺序依次办理业务 。

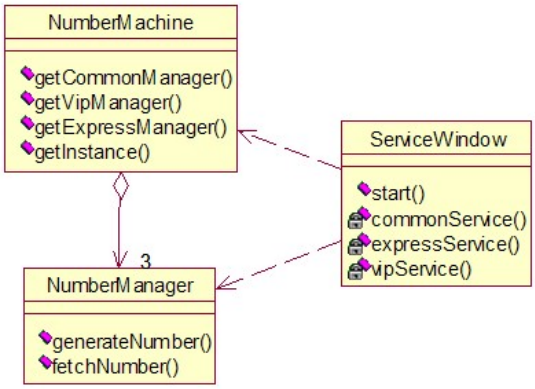
1）首先，每一个客户其实就是由银行的一个取号机器产生号码的方式来表示的。所以，我想到要有一个号码管理器对象，让这个对象不断地产生号码，就等于随机生成了客户。

2）由于有三类客户，每类客户的号码编排都是完全独立的，所以，我想到本系统一共要产生三个号码管理器对象，各自管理一类用户的排队号码。这三个号码管理器对象统一由一个号码机器进行管理，这个号码机器在整个系统中始终只能有一个，所以，它要被设计成单例。

2、各类型客户在其对应窗口按顺序依次办理业务 ，准确地说，应该是窗口依次叫号。

各个窗口怎么知道该叫哪一个号了呢？

它一定是问的相应的号码管理器，即服务窗口每次找号码管理器获取当前要被服务的号码。




## 3、NumberManager 类

1、定义一个用于存储上一个客户号码的成员变量和用于存储所有等待服务的客户号码的队列集合。

2、定义一个产生新号码的方法和获取马上要为之服务的号码的方法，这两个方法被不同的线程操作了相同的数据，所以，要进行同步。

```
import java.util.ArrayList;
import java.util.List;

public class NumberManager {
    private int lastNumber = 0;
    private List queueNumbers = new ArrayList();

    public synchronized Integer generateNewNumber(){//为客户服务,客户来了取走一个队号
```

```
        queueNumbers.add(++lastNumber);
        return lastNumber;
    }

    public synchronized Integer fetchNumber(){//服务于窗口,服务完一个叫
下一个
        if(queueNumbers.size()>0){
            return (Integer)queueNumbers.remove(0);
        }else{
            return null;
        }
    }
}
```

## 4、NumberMachine 类

定义三个成员变量分别指向三个 NumberManager 对象，分别表示普通、快速和 VIP 客户的号码管理器，定义三个对应的方法来返回这三个 NumberManager 对象。

将 NumberMachine 类设计成单例。

```
public class NumberMachine {

        private NumberMachine(){}
        private static NumberMachine instance = new NumberMachine();
        public static NumberMachine getInstance(){
            return instance;
        }

        private NumberManager commonManager = new NumberManager();
        private NumberManager expressManager = new NumberManager();
        private NumberManager vipManager = new NumberManager();
        public NumberManager getCommonManager() {
            return commonManager;
        }
        public NumberManager getExpressManager() {
            return expressManager;
        }
        public NumberManager getVipManager() {
            return vipManager;
        }

}
```

## 5、CustomerType 枚举类

系统中有三种类型的客户，所以用定义一个枚举类，其中定义三个成员分别表示三种类型的客户。

重写 toString 方法，返回类型的中文名称。这是在后面编码时重构出来的，刚开始不用考虑。

```
public enum CustomerType {
    COMMON,EXPRESS,VIP;
    public String toString(){
        String name = null;
        switch(this){
        case COMMON:
            name = "普通";
            break;
        case EXPRESS:
            name = "快速";
            break;
        case VIP:
            name = name();
            break;
        }
        return name;
    }
}
```

## 6、ServiceWindow 类

**定义一个 start 方法，内部启动一个线程，根据服务窗口的类别分别循环调用三个不同的方法。**

**定义三个方法分别对三种客户进行服务，为了观察运行效果，应详细打印出其中的细节信息。**

```
import java.util.Random;
import java.util.concurrent.Executors;
import java.util.logging.Logger;
```

```
/**
 * 没有把VIP窗口和快速窗口做成子类，是因为实际业务中的普通窗口可以随时被设置为VIP窗口和快速窗口。
 * */
public class ServiceWindow {
    private static Logger logger = Logger.getLogger("cn.itcast.bankqueue");
    private CustomerType type = CustomerType.COMMON;
    private int number = 1;

    public CustomerType getType() {
        return type;
    }

    public void setType(CustomerType type) {
        this.type = type;
    }

    public void setNumber(int number){
        this.number = number;
    }

    public void start(){
        Executors.newSingleThreadExecutor().execute(
                new Runnable(){
                    public void run(){
                        //下面这种写法的运行效率低，最好是把while放在case下面
                        while(true){
                            switch(type){
                                case COMMON:
                                    commonService();
                                    break;
                                case EXPRESS:
                                    expressService();
                                    break;
                                case VIP:
                                    vipService();
                                    break;
                            }
                        }
                    }
                }
        );
    }
```

```
    private void commonService(){
        String windowName = "第" + number + "号" + type + "窗口";
        System.out.println(windowName + "开始获取普通任务!");
        Integer serviceNumber = NumberMachine.getInstance().getCommonManager().fetchNumber();
        if(serviceNumber != null ){
            System.out.println(windowName + "开始为第" + serviceNumber + "号普通客户服务");
            int maxRandom = Constants.MAX_SERVICE_TIME - Constants.MIN_SERVICE_TIME;
            int serviceTime = new Random().nextInt(maxRandom)+1 + Constants.MIN_SERVICE_TIME;

            try {
                Thread.sleep(serviceTime);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(windowName + "完成为第" + serviceNumber + "号普通客户服务，总共耗时" + serviceTime/1000 + "秒");
        }else{
            System.out.println(windowName + "没有取到普通任务，正在空闲一秒");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    private void expressService(){
        Integer serviceNumber = NumberMachine.getInstance().getExpressManager().fetchNumber();
        String windowName = "第" + number + "号" + type + "窗口";
        System.out.println(windowName + "开始获取快速任务!");
        if(serviceNumber !=null){
            System.out.println(windowName + "开始为第" + serviceNumber + "号快速客户服务");
            int serviceTime = Constants.MIN_SERVICE_TIME;
            try {
                Thread.sleep(serviceTime);
            } catch (InterruptedException e) {
```

```
                e.printStackTrace();
            }
            System.out.println(windowName + "完成为第" + serviceNumber
+ "号快速客户服务，总共耗时" + serviceTime/1000 + "秒");
        }else{
            System.out.println(windowName + "没有取到快速任务！");
            commonService();
        }
    }

    private void vipService(){

        Integer                    serviceNumber                    =
NumberMachine.getInstance().getVipManager().fetchNumber();
        String windowName = "第" + number + "号" + type + "窗口";
        System.out.println(windowName + "开始获取VIP任务!");
        if(serviceNumber !=null){
            System.out.println(windowName + "开始为第" + serviceNumber
+ "号VIP客户服务");
            int    maxRandom    =    Constants.MAX_SERVICE_TIME    -
Constants.MIN_SERVICE_TIME;
            int serviceTime = new Random().nextInt(maxRandom)+1 +
Constants.MIN_SERVICE_TIME;
            try {
                Thread.sleep(serviceTime);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(windowName + "完成为第" + serviceNumber
+ "号VIP客户服务，总共耗时" + serviceTime/1000 + "秒");
        }else{
            System.out.println(windowName + "没有取到VIP任务！");
            commonService();
        }
    }
}
```

## 7、MainClass 类与 Constants 类

MainClass 类
用 for 循环创建出 4 个普通窗口，再创建出 1 个快速窗口和一个 VIP 窗口。

接着再创建三个定时器，分别定时去创建新的普通客户号码、新的快速客户号码、新的 VIP 客户号码。

Constants 类

定义三个常量：MAX_SERVICE_TIME、MIN_SERVICE_TIME、COMMON_CUSTOMER_INTERVAL_TIME

```
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
import java.util.logging.Logger;

public class MainClass {

    private static Logger logger =
Logger.getLogger("cn.itcast.bankqueue");


    public static void main(String[] args) {
        //产生4个普通窗口
        for(int i=1;i<5;i++){
            ServiceWindow window = new ServiceWindow();
            window.setNumber(i);
            window.start();
        }

        //产生1个快速窗口
        ServiceWindow expressWindow = new ServiceWindow();
        expressWindow.setType(CustomerType.EXPRESS);
        expressWindow.start();

        //产生1个VIP窗口
        ServiceWindow vipWindow = new ServiceWindow();
        vipWindow.setType(CustomerType.VIP);
        vipWindow.start();

        //普通客户拿号
        Executors.newScheduledThreadPool(1).scheduleAtFixedRate(
                new Runnable(){
                    public void run(){
                        Integer serviceNumber =
NumberMachine.getInstance().getCommonManager().generateNewNumber()
;
                        /**
                         * 采用logger方式，无法看到直观的运行效果，因为
logger.log方法内部并不是直接把内容打印出出来，
```

```
                         * 而是交给内部的一个线程去处理，所以，打印出来的结果在
时间顺序上看起来很混乱。
                         */
                        //logger.info("第" + serviceNumber + "号普通客户
正在等待服务！");
                        System.out.println("第" + serviceNumber + "号普
通客户正在等待服务！");
                    }
                },
                0,
                Constants.COMMON_CUSTOMER_INTERVAL_TIME,
                TimeUnit.SECONDS);

        //快速客户拿号
        Executors.newScheduledThreadPool(1).scheduleAtFixedRate(
                new Runnable(){
                    public void run(){
                        Integer             serviceNumber             =
NumberMachine.getInstance().getExpressManager().generateNewNumber
();
                        System.out.println("第" + serviceNumber + "号快
速客户正在等待服务！");
                    }
                },
                0,
                Constants.COMMON_CUSTOMER_INTERVAL_TIME * 2,
                TimeUnit.SECONDS);

        //VIP客户拿号
        Executors.newScheduledThreadPool(1).scheduleAtFixedRate(
                new Runnable(){
                    public void run(){
                        Integer             serviceNumber             =
NumberMachine.getInstance().getVipManager().generateNewNumber();
                        System.out.println("第" + serviceNumber + "号VIP
客户正在等待服务！");
                    }
                },
                0,
                Constants.COMMON_CUSTOMER_INTERVAL_TIME * 6,
                TimeUnit.SECONDS);
    }
}
```

加油，黑马人!!!