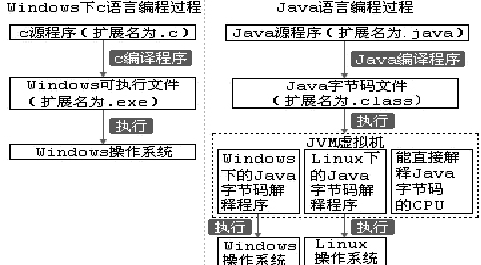
### Java虚拟机（JVM）

Java Virtual Machine 一个软件模拟的计算机。Java源程序想象成我们的C源程序，Java源程序编译后生成的字节码（.class）就相当于C源程序编译后的二进制可执行文件。JVM虚拟机相当于计算机系统（操作系统+硬件环境），Java解释器相当于CPU。

.exe文件直接在windows操作系统下运行，在CPU上运行的是机器码（.exe文件）

.class字节码文件是运行在JVM虚拟机下的，在Java解释器上运行的是Java字节码。

Java解释器相当于运行Java字节码的“CPU”，但该“CPU”不是通过硬件实现的，而是用软件实现的。Java解释器实际上就是特定的平台下的一 个应用程序。只要实现了特定平台下的解释器程序，Java字节码就能通过解释器程序在该平台下运行，这是Java跨平台的根本。当前，并不是在所有的平台 下都有相应Java解释器程序，这也是Java并不能在所有的平台下都能运行的原因，它只能在已实现了Java解释器程序的平台下运行。





**3.char 类型**

char类型表示的是单个字符类型，任何数据使用单引号括起来的都是表示字符。字符只能有一个字符，比如：普通的老百姓穿上军装就是军人。

**注意：**特殊字符的转义序列：转义字符

转义字符的概述:

特殊字符使用”\”把其转化成字符的本身输出，那么使用”\”的字符称作为转移字符。

需求：使用输出语句，打印出带引号的信息例如输出。

System.out.println("teacher said"java is fun"");编译是无法正常通过的。语法有错误，编译器读到第二个引号就认为是字符串的结束，剩余的不知道怎么处理。如何解决这个问题：java中使用转义字符来表示特殊的字符。一个转义字符以反斜杠（\）开始。

**问题：**想要打印带引号的字符串怎么办，就可以使用反斜杠（\）后跟字符，这个反斜杠就是转义字符。

|  |  |  |
| --- | --- | --- |
| **转义字符** | **名称** | **Unicode** |
| **\b** | **Backspace （退格键）** | **\u0008** |
| **\t** | **Tab （Tab键盘）** | **\u0009** |
| **\n** | **Linefeed （换行）** | **\u000A** |
| **\r** | **Carriage Return（回车）** | **\u000D** |
| **\\** | **Backslash （反斜杠）** | **\u005C** |
| **\'** | **Single Quote （单引号）** | **\u0027** |
| **\"** | **Double Quote （双引号）** | **\u0022** |

\r 表示接受键盘输入，相当于按下回车。

\n 表示换行。

\t 制表符，相当于Table键

\b 退格键，相当于Back Space

\’ 单引号

\’’ 双引号

\\ 表示一个斜跨

上述问问题解决：System.*out*.println("teacher said\"java is fun\"");

注意：换行符就是另起一行，回车符就是回到一行的开头，所以我们平时编写文件的回车符应该确切来说叫做回车换行符

# 进制的转换

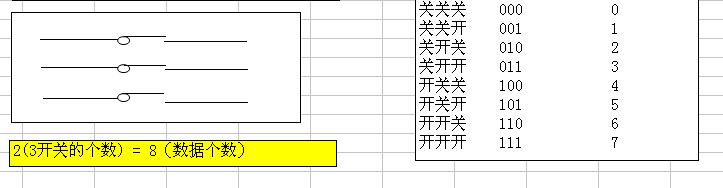
进制：进制是一种记数方式 ，可以用有限的[数字](http://baike.baidu.com/view/37407.htm" \t "_blank)符号代表所有的数值。由特定的数值组成。

## 整型的表现形式

1. 十进制： 都是以0-9这九个数字组成，不能以0开头。
2. 二进制： 由0和1两个数字组成。
3. 八进制： 由0-7数字组成，为了区分与其他进制的数字区别，开头都是以0开始。
4. 十六进制：由0-9和A-F组成。为了区分于其他数字的区别，开头都是以ox开始。

## 进制的由来

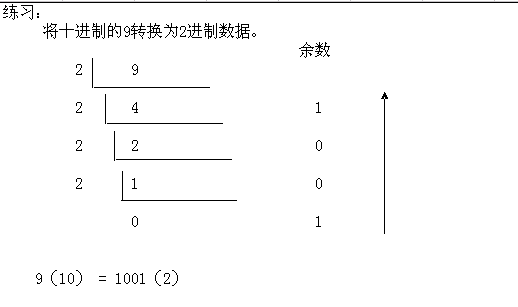
几乎每个民族最早都使用都十进制计数法，这是因为人类计数时自然而然地首先使用的是十个手指。 但是这不等于说只有十进制计数法一种计数方法。例如，世界各国在计算年月日时不约而同地使用“十二进制”12个月为一年又如：我国过去16两才算为一斤，这就是“十六进计数法”,一个星期七天，这个就是”七进制计算法”。 计算机是由[逻辑电路](http://baike.baidu.com/view/23361.htm" \t "_blank)组成，逻辑电路通常只有两个状态，开关的接通与断开，这两种状态正好可以用“1”和“0”表示。

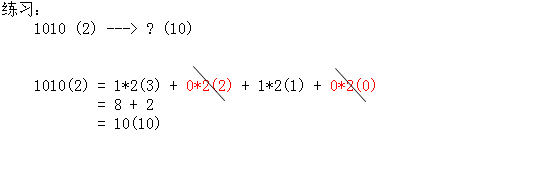


如果要在计算机里面保存十进制的7.

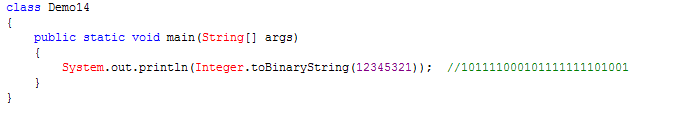
### 十进制与二进制之间的转换

十进制转二进制的转换原理：除以2，反向取余数，直到商为0终止。

二进制转十进制的转换原理:就是用二进制的每一个乘以2的n次方，n从0开始，每次递增1。然后得出来的每个数相加



存在问题：书写特别长，不方便记忆。



### 十进制与八进制之间转换

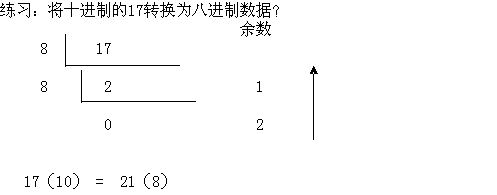
1. **八进制的由来**

二进位制在计算机内部使用是再自然不过的。但在人机交流上，二进位制有致命的弱点——数字的书写特别冗长。例如，十进位制的100000写成二进位制成为11000011010100000。为了解决这个问题，在计算机的理论和应用中还使用两种辅助的进位制——八进位制和十六进位制。二进位制的三个数位正好记为八进位制的一个数位，这样，数字长度就只有二进位制的三分之一，与十进位制记的数长度相差不多。例如，十进位制的100000写成八进位制就是303240。十六进位制的一个数位可以代表二进位制的四个数位，这样，一个字节正好是十六进位制的两个数位。十六进位制要求使用十六个不同的符号，除了0—9十个符号外，常用A、B、C、D、E、F六个符号分别代表（十进位制的）10、11、12、13、14、15。这样，十进位制的100000写成十六进位制就是186A0。

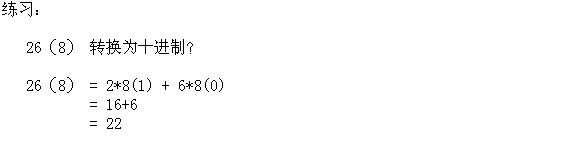
1. **八进制的特点**

由数字0-7组成。即使用三个开关表示一个八进制数。

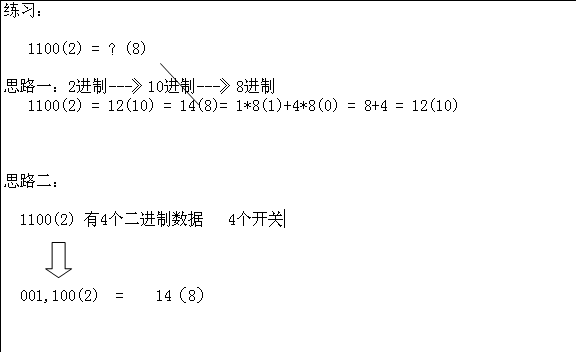
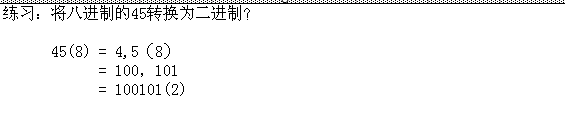
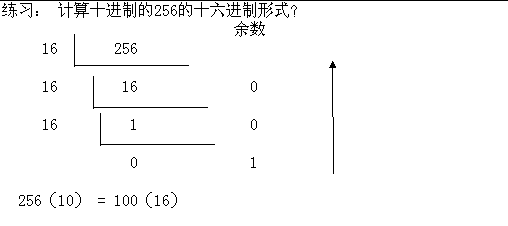
10进制转换8进制原理：就是用十进制的数字不断除于8，取余数。



八进制转十进制原理： 用把进制的数不断乘以8的n次方，n从0开始，每次递增1。



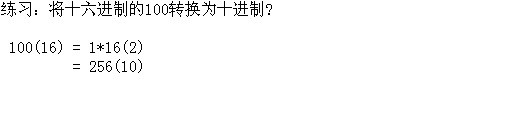
除了这种方法之外，我们还有另一种方法，因为三个开关表示一个八进制数。

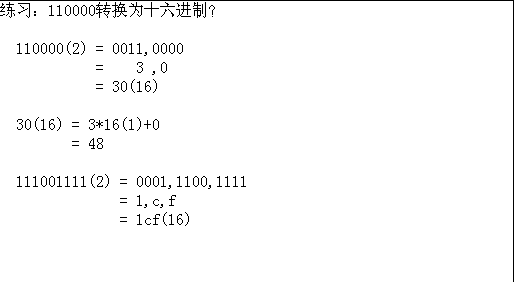


十进制与十六进制之间的转换

十六进制特点：由0~9 a(10) b(11) c(12) d(13) e(14) f(15)组成。

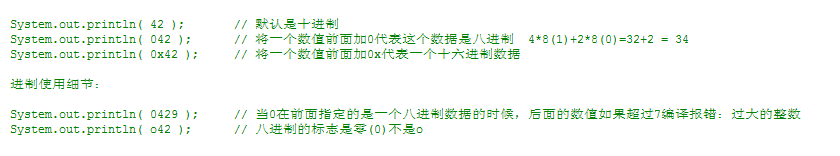
十进制转十六进制原理：就是不断除以16，取余数。



代码体现：

人使用的十进制 、 计算机底层处理的数据是二进制、八进制、十六进制，

那么如果给计算机输入不同的进制数据呢？



**1. 整型**

byte 代表一个字节的大小 8bit 2(8) -128~127 256

short 代表两个字节的大小 16bit 2(16) -2(15)~2(15)-1

int 代表四个字节的大小 32bit 2(32) -2(31)~2(31)-1

long 代表八个字节的大小 64bit 2(64) -2(63)~2(63)-1

**如果一个数值没有采取特殊的处理，那么该整数默认的类型是int。**

**可以使用数值后面添加L或小写L改变默认的整数类型。**

**2. 浮点型**

float 代表四个字节的大小 32bit

double 代表八个字节的大小 64bit

java程序中所有的小数默认的类型是double类型，所以需要使用特殊的符号改变默认的小数类型。

**3. 字符型**

char 代表两个字节的大小 16bit 2(16)

原理：将字符映射为码表中对应的十进制数据加以存储。

**4. 布尔型**

boolean 占一个字节。只有true与false两个值。

判断闰年

1：什么是闰年？可以被4整除不能被100整除，或者可以被400整除，那么这一年就是闰年（leap year）

|  |
| --- |
| **public** **static** **void** main(String[] args) {  Scanner sc = **new** Scanner(System.*in*);  System.*out*.println("请输入年份：");  **int** year = sc.nextInt();  // 判断年份能否被4整除  **boolean** isLeapYear = (year % 4 == 0);  // 年份能被4整除，并且不能被100整除并且使用&&（and）  isLeapYear = isLeapYear && (year % 100 != 0);  // 年份或者能够被400整除  isLeapYear = isLeapYear || (year % 400 == 0);  **if** (isLeapYear) {  System.*out*.println(year + "是闰年！");  }  // 简写格式；  **if** (year % 4 == 0 && year % 100 != 0 || year % 400 == 0) {  System.*out*.println(year + "是闰年！");  }  } |

switch语句

格式：

|  |
| --- |
| switch(表达式)  {  case 取值1:  执行语句；  break；  case 取值2:  执行语句；  break；  …...  default:  执行语句；  break；  } |

switch语句特点：

1,switch语句选择的类型只有四种：byte，short，int ， char。

2,case之间与default没有顺序。先判断所有的case，没有匹配的case执行

default。

3,switch语句停止的条件是遇到了break关键字或者结束switch语句的大括号。

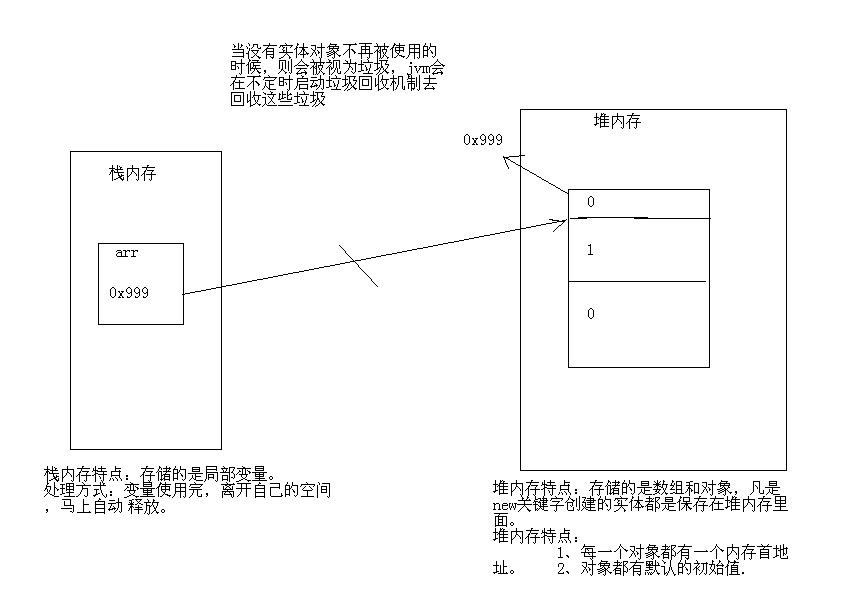
4,如果匹配的case或者default没有对应的break，那么程序会继续向下执行，运

行可以执行的语句，直到遇到break或者switch结尾结束。

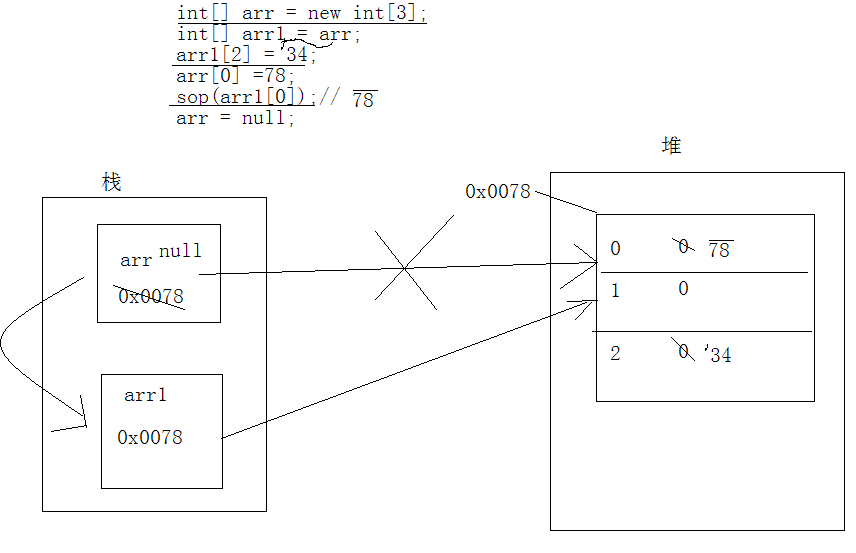
5，switch case中的值必须要与switch表达式的值具有相同的数据类型。而且case后跟的值必须是常量，不能跟变量。

### 数组的内存分析

案例分析一：



案例分析二：



### 数组的常见操作

### 6.1案例一个数组取出最大值

/\*定义一个获取最大值的功能:

1、确定结果 ：返回值类型 int

2、未知内容：要获取哪个数组的最大值没有确定，则是数组没有确定

思路：

1、定义一个变量，记录住数组的比较大的元素。

2、遍历整个数组，让数组的每一个元素都和该变量进行对比即可。

3、当变量遇到比它大的元素，则让该变量记录该元素的值，当循环结束时，最大 值产生了

\*/

**public** **static** **int** getMax(**int**[] arr)

{

//定义变量记录较大的值，初始化为数组中的任意一个元素。

**int** max = arr[0];

**for**(**int** x=1; x<arr.length; x++)

{

**if**(arr[x]>max)

max = arr[x];

}

**return** max;

}

### 6.2直接排序

案例二：使用直接排序对数组进行排序：

/\*

选择排序。

以一个角标的元素和其他元素进行比较。

在内循环第一次结束，最值出现的头角标位置上。

\*/

**public** **static** **void** selectSort(**int**[] arr)

{

**for**(**int** x=0; x<arr.length-1; x++)

{

**for**(**int** y=x+1; y<arr.length; y++)//为什么y的初始化值是 x+1？ 因为每一次比较，

//都用x角标上的元素和下一个元素进 行比较。

{

**if**(arr[x]>arr[y])

{

**int** temp = arr[x];

arr[x] = arr[y];

arr[y] = temp;

}

}

}

}

### 6.3冒泡排序

案例三：冒泡排序

/\*

冒泡排序。

比较方式：相邻两个元素进行比较。如果满足条件就进行位置置换。

原理：内循环结束一次，最值出现在尾角标位置。

\*/

**public** **static** **void** bubbleSort(**int**[] arr)

{

**for**(**int** x=0; x<arr.length-1; x++)

{

**for**(**int** y=0; y<arr.length-x-1; y++)//-x:让每次参与比较的元减。

//-1:避免角标越界。

{

**if**(arr[y]>arr[y+1])

{

**int** temp = arr[y];

arr[y] = arr[y+1];

arr[y+1] = temp;

}

}

}

}

### 6.4折半查找(二分法)

案例四：

/\*

为了提高查找效率，可使用折半查找的方式，注意：这种查找只对有序的数组有效。

这种方式也成为二分查找法。

\*/

**public** **static** **int** halfSeach(**int**[] arr,**int** key)

{

**int** min,mid,max;

min = 0;

max = arr.length-1;

mid = (max+min)/2;

**while**(arr[mid]!=key)

{

**if**(key>arr[mid])

min = mid + 1;

**else** **if**(key<arr[mid])

max = mid - 1;

**if**(min>max)

**return** -1;

mid = (max+min)/2;

}

**return** mid;

}

案例五：数组翻转

/\*

反转其实就是头角标和尾角标的元素进行位置的置换，

然后在让头角标自增。尾角标自减。

当头角标<尾角标时，可以进行置换的动作。

\*/

**public** **static** **void** reverseArray(**int**[] arr)

{

**for**(**int** start=0,end=arr.length-1; start<end; start++,end--)

{

swap(arr,start,end);

}

}

//对数组的元素进行位置的置换。

**public** **static** **void** swap(**int**[] arr,**int** a,**int** b)

{

**int** temp = arr[a];

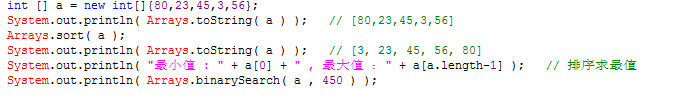
arr[a] = arr[b];

arr[b] = temp;

}

# 二维数组

**Arrays的使用**



**遍历**： toString() 将数组的元素以字符串的形式返回

**排序：** sort() 将数组按照升序排列

**查找：** binarySearch()在指定数组中查找指定元素，返回元素的索引，如果没有找到返回（-插入点-1） 注意：使用查找的功能的时候，数组一定要先排序。

**二维数组:**

吸烟：

没钱 零买 1根 一个变量

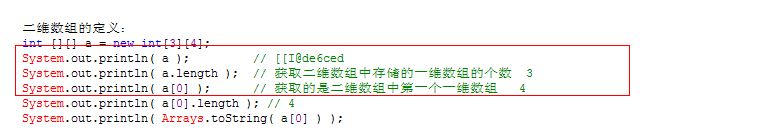
稍微有钱 一包 一维数组 20根变量

很有钱 一条 10包(二维数组) 二维数组

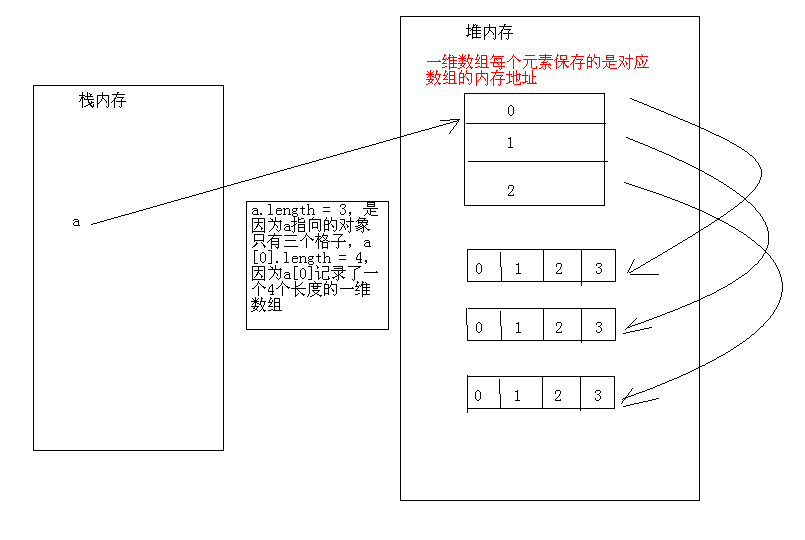
**二维数组：**实质就是存储是一维数组。

**数组定义：**

**数组类型[][] 数组名 = new 数组类型[一维数组的个数][每一个一维数组中元素的个数];**



**疑问： 为什么a.length = 3, a[0].length = 4?**

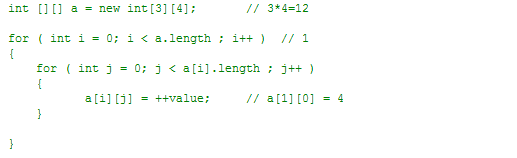


数组的初始化：

静态初始化:

int [][] a = new int[][]{ {12,34,45,89},{34,56,78,10},{1,3,6,4} };

动态初始化：



**二维数组常见的操作:**

1. 遍历二维数组
2. 对二维数组求和

|  |
| --- |
| **class** Demo3  {  // 定义一个遍历二维数组的功能函数  **public** **static** **void** printArr2( **int** [][] a ){  // 1. 拆开二维数组  **for** ( **int** i = 0 ; i < a.length ; i++ )  {  // 2. 拆开一维数组获取数据  **for** ( **int** j = 0 ; j < a[i].length ; j++ )  {  System.*out*.print( a[i][j]+" ," );  }  }    }  // 定义一个函数计算二维数组中的元素的累加和  **public** **static** **long** getSum( **int** [][] a ){  // 0. 定义一个结果变量  **long** sum = 0L;  // 1. 拆开二维数组  **for** ( **int** i = 0 ; i < a.length ; i++ )  {  // 2. 拆开一维数组获取数据  **for** ( **int** j = 0 ; j < a[i].length ; j++ )  {  sum+=a[i][j];  }  }  **return** sum;  }    // 统计二维数组中元素的个数  **public** **static** **int** getDataCount( **int** [][] a ){  // 0. 记录元素个数  **int** count = 0;  // 1. 拆开二维数组  **for** ( **int** i = 0 ; i < a.length ; i++ )  {  // 2. 拆开一维数组获取数据  **for** ( **int** j = 0 ; j < a[i].length ; j++ )  {  count++;  }  }  **return** count;  }  **public** **static** **void** main(String[] args)  {    **int** [][] a = **new** **int**[][]{ {23,4,5},{2},{4,5,78,56,90} };  *printArr2*( a );  System.*out*.println();  System.*out*.println("累加和是： "+*getSum*( a ) );  System.*out*.println("统计元素个数： "+*getDataCount*( a ) );  System.*out*.println("Hello World!");      }  } |

**成员变量与局部变量的区别：**

1. 应用范围
   * + 1. 成员变量在整个类内都有效
       2. 局部变量只在其声明的方法内有效
2. 生命周期
3. 成员变量: 它属于对象，它随着对象的创建而创建，随着对象的消失而消失
4. 局部变量: 使用完马上释放空间。

void show(int id){

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

for(int j=0;j<10;j++){

System.out.println(id);

}

}

}

这时候 id,i,j者是在方法内声明的，全是局部变量

j当里层for循环执行它的生命周期开始，当里层for结束，j消失

i当外层for循环执行它的生命周期开始，当外层for结束，j消失

id在方法被调用时开始，方法结束时，id消失.

1. 存储位置 成员变量属于对象，它存储在堆内，堆内的实体，当没有引用指向其时，才垃圾回收清理 局部变量存在栈内存中，当不在使用时，马上就会被释放。
2. 初始值

成员变量它存储在堆中，如果没有赋初值，它有默认值。

整数byte、short、int、long =0；

char='\uoooo'；

boolean =flase；

String =null;

类类型 =null；

数组 =null；

局部变量，如果要想使用必须手动初始化.

方法中，参数列表中，语句中。

必须给初始化值，没有初始值，不能使用

在栈内存中

## 内存分析

案例一：

//汽车

**class** Car {

//汽车应该具备的属性

**int** num;

//汽车具备的颜色

String color;

//汽车跑的行为

**public** **void** run(){

System.*out*.println(num+"轮子的汽车跑起来啦");

}

}

**public** **class** CarDemo{

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

{ //创建实体，并且给该实体起一个名字

Car c = **new** Car();

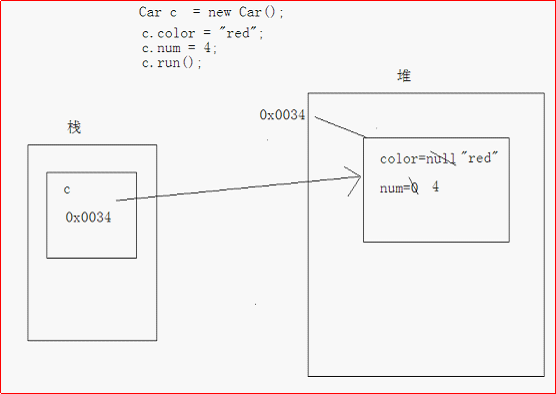
c.color = "red";

c.num = 4;

c.run();//指挥车进行运行。调用格式：对象.对象成员

}

}



(图1 )

案例二分析：

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

{ //创建实体，并且给该实体起一个名字

Car c = **new** Car();

Car c1 = **new** Car();

c.color = "red";

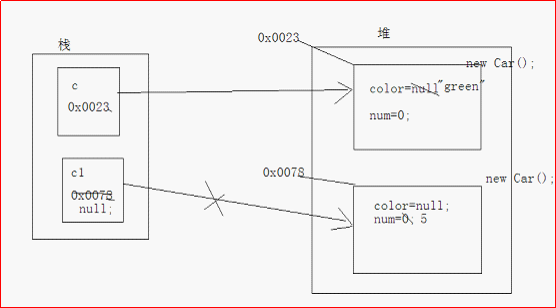
c1.num = 4;

System.*out*.println(c1.color);

c.run();//指挥车进行运行。调用格式：对象.对象成员

}

内存图：



(图二)

案例三

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

{ //创建实体，并且给该实体起一个名字

Car c = **new** Car();

Car c1 = c;

c.color = "red";

c1.num = 4;

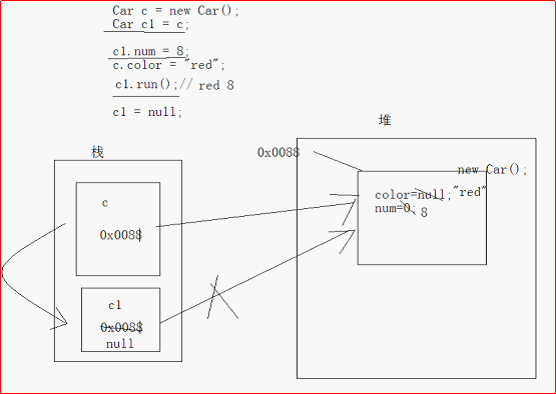
c1.color = "green";

System.*out*.println(c1.color);

c.run();//指挥车进行运行。调用格式：对象.对象成员

}

内存图三



# (图三) static关键字

1：如果没有static会怎样？

1：定义Person类

1：姓名、年龄、国籍，说话行为

2：多个构造，重载形式体现

2：中国人的国籍都是确定的

1：国籍可以进行显示初始化

|  |
| --- |
| **class** Person {  String name;  **int** age;  String gender;  String country = "CN";  Person() {  }  Person(String name, **int** age, String gender, String country) {  **this**.name = name;  **this**.age = age;  **this**.gender = gender;  **this**.country = country;  }  **void** speak() {  System.*out*.println("国籍:" + country + " 姓名：" + name + " 性别：" + gender + " 年龄：" + age + " 哈哈！！！");  }  } |

3：new Person 对象

1：分析内存

2：每个对象都维护实例变量国籍也是。

|  |
| --- |
| **public** **class** PersonDemo {  **public** **static** **void** main(String[] args) {  Person p1 = **new** Person("jack", 20, "男");  p1.speak();  Person p2 = **new** Person("rose", 18, "女");  p2.speak();  }  } |

4：内存分析

1：栈，堆、共享区

2：Demo.class加载近共享区

1：Demo类的main方法进栈

2：Person p1=new Person();

1：Person.class 加载进方法区

2：堆内存开辟空间，实例变量进行默认初始化，显示初始化。

3：内存地址传给变量p1，栈和堆建立连接

3：person p2=new Person();

1：堆内存开辟空间，实例变量进行默认初始化，显示初始化。

2：内存地址传给变量p2，栈和堆建立连接

4：如果建立多个Person对象发现问题

1：每个对象都维护有国籍。

5：解决问题，内存优化

1：为了让所有Person对象都共享一个country ，可以尝试将country放入共享区。

2：country变量如何放入共享区？对象如何访问？

1：使用static

2：static

1：为了实现对象之间重复属性的数据共享

3：static使用

1：主要用于修饰类的成员

1：成员变量

1：非静态成员变量：需要创建对象来访问

2：静态成员变量：使用类名直接调用，也可以通过对象访问

|  |
| --- |
| **public** **static** **void** main(String[] args) {    //访问静态成员  //直接通过类名来调用  String country=Person.*country*;  System.*out*.println(country);    //通过对象.成员的形式访问  Person p1 = **new** Person("jack", 20, "男");  p1.*country*="US";  p1.speak();  }  **class** Person {  String name;  **int** age;  String gender;  //static 修饰成员变量  **static** String *country* = "CN";  Person() {  }  Person(String name, **int** age, String gender) {  **this**.name = name;  **this**.age = age;  **this**.gender = gender;  }  **void** speak() {  System.*out*.println("国籍:" + *country* + " 姓名：" + name + " 性别：" + gender  + " 年龄：" + age + " 哈哈！！！");  }  } |

2：成员方法

可以使用类名直接调用

1：静态函数：

1：静态函数中不能访问非静态成员变量，只能访问静态变量。

2：静态方法不可以定义this,super关键字.

3：因为静态优先于对象存在.静态方法中更不可以出现this

2：非静态函数：非静态函数中可以访问静态成员变量

|  |
| --- |
| **class** Person {  String name;  **int** age;  String gender;  //static 修饰成员变量  **static** String *country* = "CN";  Person() {  }  Person(String name, **int** age, String gender) {  **this**.name = name;  **this**.age = age;  **this**.gender = gender;  }  //非静态方法  **void** speak() {  //非静态方法可以访问静态成员  System.*out*.println("国籍:" + *country* );    System.*out*.println("国籍:" + *country* + " 姓名：" + name + " 性别：" + gender  + " 年龄：" + age + " 哈哈！！！");    }  //静态方法  **static** **void** run(){  //静态方法只能访问静态成员变量。  System.*out*.println("国籍:"+*country*);    //静态方法访问非静态成员变量，编译报错。  *System.out.println(" 姓名：" + name);*    //静态方法中不可以出现this,编译报错  ***this****.speak();*  }  } |

2：细节：

1：静态函数中不能使用非静态变量

2：非静态函数可以访问静态变量

3：为什么静态函数中不能访问非静态成员

1：static修饰的成员在共享区中。优先于对象存在

2：验证

1：使用静态代码块验证

1：静态代码块

static{

静态代码块执行语句;

}

1：静态代码块特点

随着类的加载而加载。只执行一次，优先于主函数。用于给类进行初始化。

|  |
| --- |
| **public** **class** PersonDemo {  **public** **static** **void** main(String[] args) {  // 访问静态成员  // 直接通过类名来调用  String country = Person.*country*;  System.*out*.println(country);  // 通过对象.成员的形式访问  Person p1 = **new** Person("jack", 20, "男");  p1.*country* = "US";  p1.speak();  }  }  **class** Person {  String name;  **int** age;  String gender;  // static 修饰成员变量  **static** String *country* = "CN";  **static** {  System.*out*.println("这是静态代码块");  }  {  System.*out*.println("这是构造代码块");  }  Person() {  System.*out*.println("无参数构造");  }  Person(String name, **int** age, String gender) {  **this**.name = name;  **this**.age = age;  **this**.gender = gender;  System.*out*.println(" 有参数构造");  }  // 非静态方法  **void** speak() {  // 非静态方法可以访问静态成员  System.*out*.println("国籍:" + *country*);  System.*out*.println("国籍:" + *country* + " 姓名：" + name + " 性别：" + gender  + " 年龄：" + age + " 哈哈！！！");  // 非静态方法可以调用静态方法。  *run*();  }  // 静态方法  **static** **void** run() {  // 静态方法只能访问静态成员变量。  System.*out*.println("国籍:" + *country*);  }  } |

4：static特点

1 随着类的加载而加载，静态会随着类的加载而加载，随着类的消失而消失。说明它的生命周期很长。

2 优先于对象存在。-->静态是先存在，对象是后存在。

3 被所有实例(对象)所共享。

4 可以直接被类名调用

5：静态变量（类变量）和实例变量的区别:

1存放位置

1：类变量随着类的加载而加载存在于方法区中.

2：实例变量随着对象的建立而存在于堆内存中.

2生命周期

1：类变量生命周期最长,随着类的消失而消失.

2：实例变量生命周期随着对象的消失而消失.

6：静态优缺点

1： 优点:对对象的共享数据进行单独空间的存储,节省空间 例如Person 都有

国籍。该数据可以共享可以被类名调

2：缺点：生命周期过长

访问出现局限性。（静态只能访问静态）

7: 什么时候定义静态变量

1:静态变量（类变量）当对象中出现共享数据

例如：学生的学校名称。学校名称可以共享

对象的数据要定义为非静态的存放在对内存中（学生的姓名，学生的年龄）

8：什么时候定义静态函数

如果功能内部没有访问到非静态数据（对象的特有数据。那么该功能就可以定义为静态）

9：静态的应用

自定义数组工具类

|  |
| --- |
| /\*  定义数组工具类  1:定义一个遍历数组的函数  2:定义一个求数组和的功能函数 1. 遍历 2. 两两相加  3:定义一个获取数组最大值的功能函数  4:定义一个获取数组最大值角标的功能函数  5:定义一个返回指定数在指定数组中包含的角标的功能函数  6:定义一个可以用于排序int数组的函数  1:冒泡  2:选择    定义自己的工具类    \*/  **class** Arrays {  **private** Arrays() {  }  // 1:定义一个遍历数组的函数  **public** **static** **void** print(**int**[] arr) {  **for** (**int** x = 0; x < arr.length; x++) {  **if** (x != (arr.length - 1)) {  System.*out*.print(arr[x] + ",");  } **else** {  System.*out*.print(arr[x]);  }  }  }  // 2:定义一个求数组和的功能函数  **public** **static** **int** getSum(**int**[] arr) {  **int** sum = 0;  **for** (**int** x = 0; x < arr.length; x++) {  sum += arr[x];  }  **return** sum;  }  // 3:定义一个获取数组最大值的功能函数  **public** **static** **int** getMax(**int**[] arr) {  **int** max = 0;  **for** (**int** x = 0; x < arr.length; x++) {  **if** (arr[max] < arr[x]) {  max = x;  }  }  **return** arr[max];  }  // 4:定义一个获取数组最大值角标的功能函数  **public** **static** **int** getIndexMax(**int**[] arr) {  **int** max = 0;  **for** (**int** x = 0; x < arr.length; x++) {  **if** (arr[max] < arr[x]) {  max = x;  }  }  **return** max;  }  // 5:定义一个返回 指定数在指定数组中包含的角标的功能函数  **public** **static** **int** getIndex(**int**[] arr, **int** src) {  **int** index = -1;  **for** (**int** x = 0; x < arr.length; x++) {  **if** (arr[x] == src) {  index = x;  }  }  **return** index;  }  // 冒泡  **public** **static** **void** test(**int**[] arr) {  **for** (**int** x = 0; x < arr.length - 1; x++) {  **if** (arr[x] > arr[x + 1]) {  **int** temp = arr[x + 1];  arr[x + 1] = arr[x];  arr[x] = temp;  }  }  }  // 选择排序  **public** **static** **void** selectSort(**int**[] arr) {  **for** (**int** x = 0; x < arr.length - 1; x++) {  **for** (**int** y = 1 + x; y < arr.length; y++) {  **if** (arr[x] > arr[y]) {  **int** temp = arr[y];  arr[y] = arr[x];  arr[x] = temp;  }  }  }  }  // 7：定义一个可以将整数数组进行反序的功能函数。  **public** **static** **void** reverseSort(**int**[] arr) {  **int** start = 0;  **int** end = arr.length - 1;  **for** (**int** x = 0; x < arr.length; x++) {  **if** (start < end) {  **int** tem = arr[start];  arr[start] = arr[end];  arr[end] = tem;  }  start++;  end--;  }  }  // 折半查找  **public** **static** **int** halfSearch(**int** key, **int**[] arr) {  **int** min = 0;  **int** max = arr.length - 1;  **int** mid = 0;  **while** (min < max) {  mid = (min + max) / 2;  **if** (key > arr[mid]) {  min = mid + 1;  } **else** **if** (key < arr[mid]) {  max = mid - 1;  } **else** {  **return** mid;  }  }  **return** -1;  }  }  **class** Demo6 {  **public** **static** **void** main(String[] args) {  **int**[] arr = { 3, 4, 5, 2, 3, 7, 4 };  Arrays.*print*(arr);  System.*out*.println();  Arrays.*selectSort*(arr);  Arrays.*print*(arr);  }  } |

# 单例设计模式

一些人总结出来用来解决特定问题的固定的解决方案。

解决一个类在内存中只存在一个对象，想要保证对象的唯一。

1 为了避免其他程序过多的建立该类对象。禁止其他程序建立该类对象。

2 为了其他程序可以访问该类对象，在本类中自定义一个对象。

3 方便其他程序对自定义类的对象的访问，对外提供一些访问方式。

代码：

1将构造函数私有化

2在类中创建一个私有的本类对象

3提供一个用类名调用的公有方法获取该对象。

|  |
| --- |
| **class** Single {  **private** **static** Single *s* = **new** Single(); // 恶汉式  **private** Single() {  }  **public** **static** Single getInstance() {  **return** *s*;  }  }  **class** Single2 {  **private** **static** Single2 *s* = **null**; // 懒汉  **private** Single2() {  }  **public** **static** Single2 getInstance() {  **if** (*s* == **null**) {  *s* = **new** Single2();  }  **return** *s*;  }  } |

## instanceof 关键字

1：快速演示instanceof

|  |
| --- |
| Person p=**new** Person();  System.*out*.println( p **instanceof** Person); |

2：instanceof是什么？

1：属于比较运算符：

2：instanceof关键字：该关键字用来判断一个对象是否是指定类的对象。

3：用法：

对象 instanceof 类;

该表达式是一个比较运算符，返回的结果是boolea类型 true|false

注意：使用instanceof关键字做判断时，两个类之间必须有关系。

3：案例

定义一个功能表函数，根据传递进来的对象的做不同的事情，如果是狗让其看家，如果是猫让其抓老鼠

1：定义动物类

2：定义狗类继承动物类

3：定义猫类继承动物类

4：定义功能根据传入的动物，执行具体的功能

5：instanceof好处

1：可以判断对象是否是某一个类的实例

|  |
| --- |
| **package** oop01;  /\*  instanceof  比较运算符  检查是否是类的对象  1：可以判断对象是否是某一个类的实例  用法  对象 instanceof 类;    案例  定义一个功能函数，根据传递进来的对象的做不同的事情  如果是狗让其看家，如果是猫让其抓老鼠  1：定义动物类  2：定义狗类继承动物类  3：定义猫类继承动物类  4：定义功能根据传入的动物，执行具体的功能  \*/  **class** Animal {  String name;  **void** eat() {  System.*out*.println("吃东西");  }  **void** shout() {  System.*out*.println("我是动物");  }  }  **class** Dog **extends** Animal {  **void** eat() {  System.*out*.println("啃骨头");  }  **void** shout() {  System.*out*.println("旺旺");  }  }  **class** Cat **extends** Animal {  **void** eat() {  System.*out*.println("吃老鼠");  }  **void** shout() {  System.*out*.println("喵喵");  }  }  **class** Demo11 {  **public** **static** **void** main(String[] args) {  Demo11 d = **new** Demo11();  // 对象 instanceof 类;  System.*out*.println(d **instanceof** Demo11);  d.doSomething(new Dog());  d.doSomething(**new** Cat());  }  // 定义一个功能函数，根据传递进来的对象的做不同的事情  // 如果是狗让其看家，如果是猫让其抓老鼠  // 对象 instanceof 类;  **void** doSomething(Animal a) {  **if** (a **instanceof** Dog) {  a.eat();  a.shout();  System.*out*.println("小狗看家");  } **else** **if** (a **instanceof** Cat) {  a.eat();  a.shout();  System.*out*.println("抓老鼠");  }  }  } |

## final关键字

1：定义静态方法求圆的面积

2：定义静态方法求圆的周长

3：发现方法中有重复的代码，就是PI，圆周率。

1：如果需要提高计算精度，就需要修改每个方法中圆周率。

4：描述一个变量

1：方法都是静态的，静态只能访问静态，所以变量也定义为静态的。

public static double PI=3.14;

1：如果定义为public后，新的问题，类名.PI=300; 改变了PI的值。

2：修改为private，修改为private后进行了封装，需要getset公共访问方法。

3：现有的知识不能解决这样的问题了。可以使用final

|  |
| --- |
| **class** Demo12 {  **public** **static** **final** **double** *PI* = 3.14; // 静态常量  **public** **static** **double** getArea(**double** r) {  **return** *PI* \* r \* r;  }  **public** **static** **double** getLength(**double** r) {  **return** *PI* \* r \* 2;  }  **public** **static** **void** main(String[] args) {  // Demo12.PI=300; 无法为最终变量 PI 指定值  System.*out*.println(Demo12.*PI*);  }  } |

5：使用final

1：final关键字主要用于修饰类、类成员、方法、以及方法的形参。

2：final修饰成员属性：

1：说明该成员属性是常量，不能被修改。

public static final double PI=3.14;

1：public ：访问权限最大

2：static ：内存中只有一份

3：final ：是一个常量

4：常量名大写

5：必须初赋值。

2：使用类名.成员。修改该成员的值，报错。--常量不能被修改

1：基本数据类型，final使值不变

2：对象引用，final使其引用恒定不变，无法让其指向一个新的对象，但是对象自身却可以被修改。

3：该关键字一般和static关键字结合使用

1：常量可以优先加载，不必等到创建对象的时候再初始化。

4：final和static可以互换位置

5：常量一般被修饰为final

3：fianl修饰类：

1：该类是最终类，不能被继承。

1：将父类加final修饰，子类继承，就会报错。

2：查看api文档发现String类是final的。Integer类也是final的

1：为了防止代码功能被重写

2：该类没有必要进行扩展

4：final修饰方法：

1：该方法是最终方法，不能被重写

2：当一个类被继承，那么所有的非私有函数都将被继承，如果函数不想被子类继承并重写可以将该函数final修饰

3：当一个类中的函数都被修饰为final时，可以将类定义为final的。

|  |
| --- |
| **class** Father2{  **final** **void** eat(){  System.*out*.println("eating....");  }  }  **class** Son2 **extends** Father2{  //该方法是最终方法，不能被重写  **void** eat(){  System.*out*.println("eating....");  }  }  **class** Demo12 {    **public** **static** **void** main(String[] args) {  // Demo12.PI=300; 无法为最终变量 PI 指定值  System.*out*.println(Demo12.*PI*);  Son2 s=**new** Son2();  s.eat();  } |

5：final关键字修饰形参

1：当形参被修饰为final,那么该形参所属的方法中不能被篡改。

2： 项目中主要用于一些只用来遍历未知数据的函数。将未知变量声明为final的。增强数据的安全性。

|  |
| --- |
| **class** Demo14 {  **public** **static** **void** main(String[] args) {  System.*out*.println();  String[] arr = { "think in java", "java就业教程", "java核心技术" };  *print*(arr);  }  // 该方法，打印书名。  **public** **static** **void** print(**final** String[] arr) {  //arr = null; ,无法重新赋值  **for** (**int** x = 0; x < arr.length; x++) {  System.*out*.println(arr[x]);  }  }  } |

10：思考

为什么子类一定要访问父类的构造函数呢

1：子类继承了父类的属性，如果要使用父类的属性必须初始化，创建子类对象，必须先初始化父类属性

必须调用父类的构造方法。

2：为什么调用父类无参的构造函数

设计java语言之时，只知道编译器会默认添加无参的构造函数，有参的无法确定。

但是可以通过super关键字显式调用父类指定构造函数。

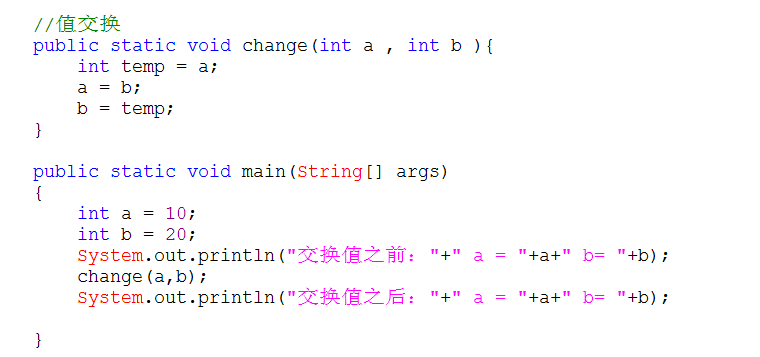
3：为什么super()this()语句要放在构造函数的第一行

子类可能会用到父类的属性，所以必须先初始化父类。

# 值交换

案例： 定义交换数值的功能函数，基本类型数据, 数组,实例对象, String。

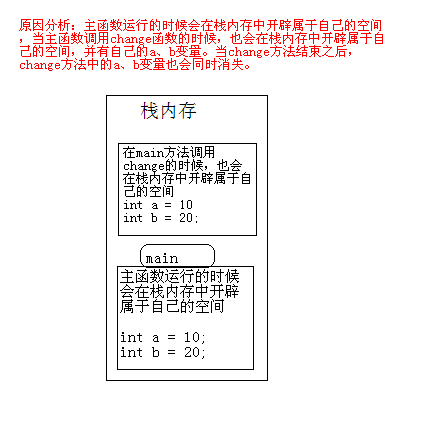
**基本数据类型交换**



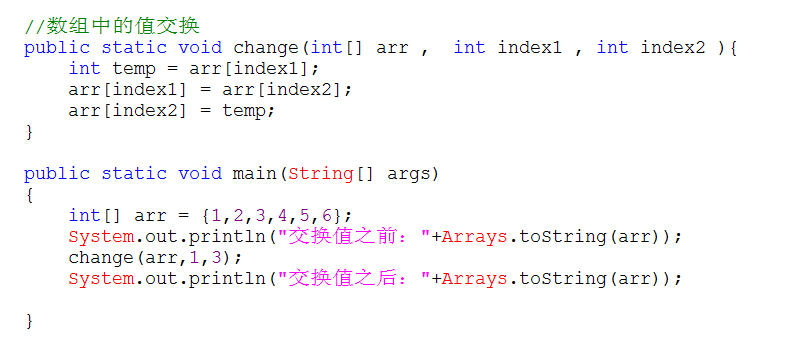


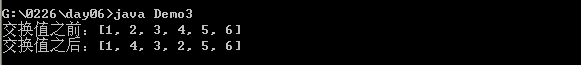
结果：发现交换值前后没有变量的值发生变化。

原因分析：

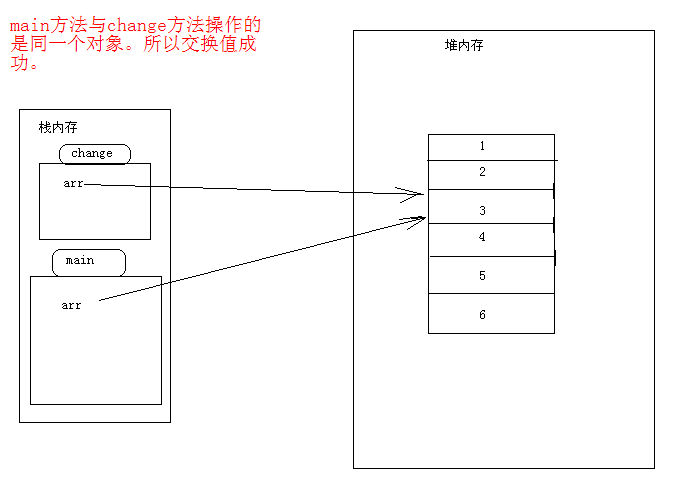


**数组类型交换**



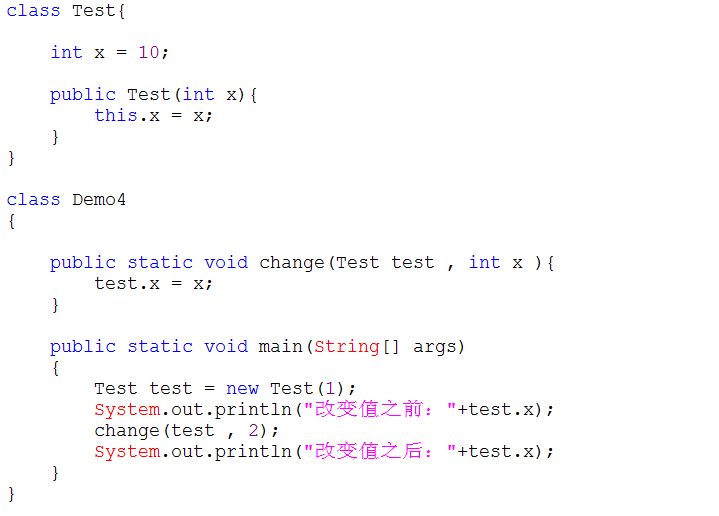


**结果：交换值成功。**



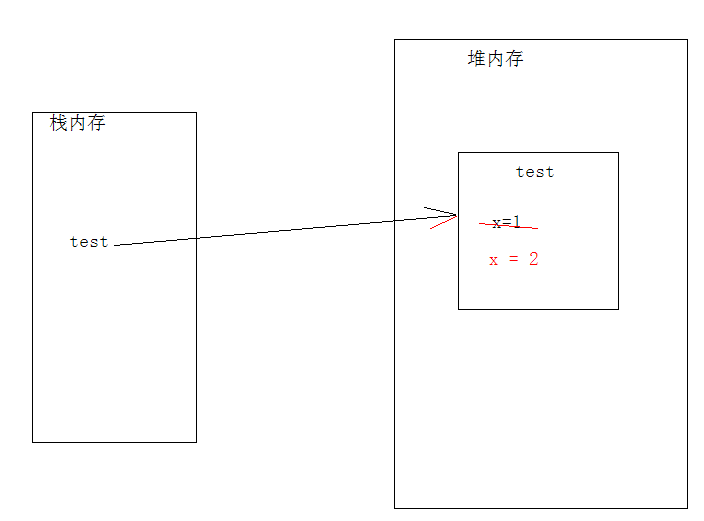
原因分析：操作的是同一个数组对象。

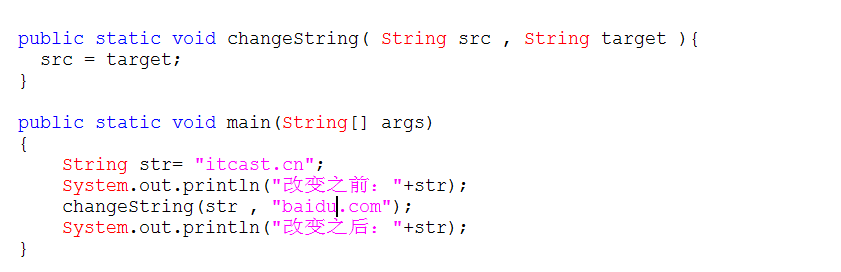
**对象的值交换：**



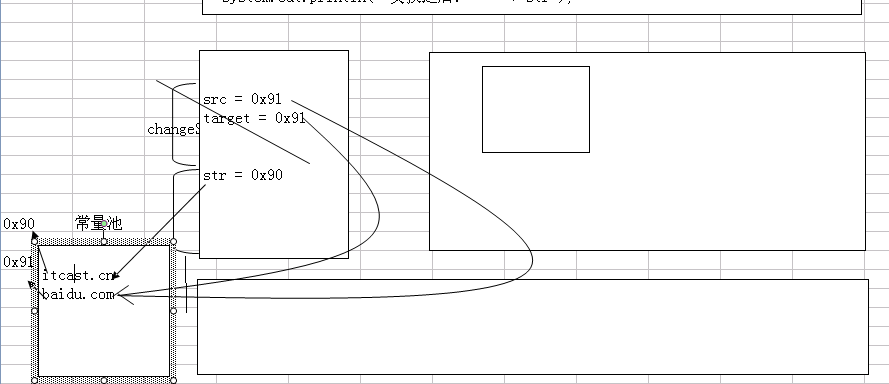


结果：交换值成功。



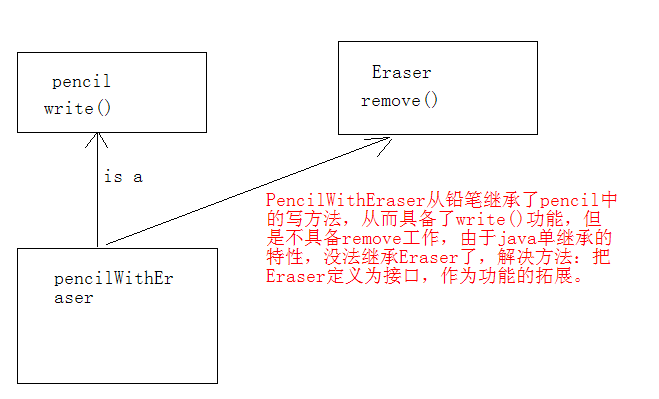
**字符串的值交换:**

**交换值失败。**



# 接口

## 接口的概述



接口(interface)：usb接口，主要是使用来拓展笔记本的功能，那么在java中的接口主要是使用来拓展定义类的功能，可以弥补java中单继承的缺点。

|  |
| --- |
| **class** Pencil {  String name;  Pencil() {  }  Pencil(String name) {  **this**.name = name;  }  **void** write() {  System.*out*.println("写字");  }  }  **interface** Eraser {  **public** **static** **final** String *color* = "白色";  **public** **abstract** **void** clean();  }  // 1：带橡皮的铅笔类继承铅笔类实现橡皮接口  **class** PencilWithEraser **extends** Pencil **implements** Eraser {  PencilWithEraser() {  }  PencilWithEraser(String name) {  **super**(name);  }  **void** write() {  System.*out*.println(name + ":考试专用");  }  **public** **void** clean() {  System.*out*.println(**super**.name + ":带橡皮的铅笔，就是好用");  }  }  **class** Demo6 {  **public** **static** **void** main(String[] args) {  PencilWithEraser pe = **new** PencilWithEraser("中华2B");  pe.write();  pe.clean();  System.*out*.println(pe.*color*);  System.*out*.println(PencilWithEraser.*color*);  }  } |

接口的定义格式：

|  |
| --- |
| interface 接口名{  属性  抽象方法  } |

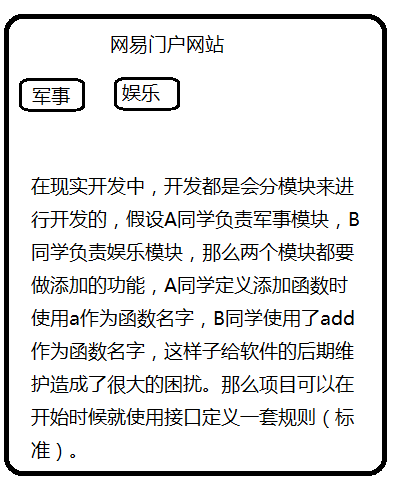
接口的体验

|  |
| --- |
| interface Inter  {  int num = 6; 可以定义属性与方法。  void show();  } |

注意：可以通过javap命令查看.

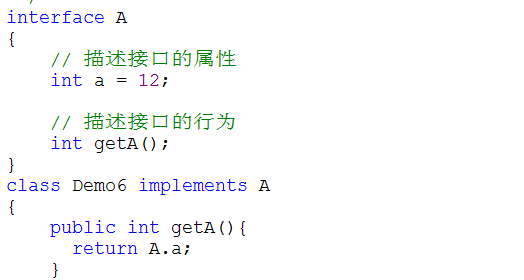
1. 接口中的所有属性 默认的修饰符是 public static final。
2. 接口中的所有方法 默认的修饰符是 public abstract。

疑惑：干嘛不在PencilWithEraser添加remove功能函数，而要通过接口？



## 接口的特点

1. 类实现接口可以通过implements实现，实现接口的时候必须把接口中的所有方法实现,一个类可以实现多个接口。
2. 接口中定义的所有的属性默认是public static final的，即静态常量既然是常量，那么定义的时候必须赋值。
3. 接口中定义的方法不能有方法体。接口中定义的方法默认添加public abstract
4. 有抽象函数的不一定是抽象类，也可以是接口类。
5. 由于接口中的方法默认都是抽象的，所以不能被实例化。
6. 对于接口而言，可以使用子类来实现接口中未被实现的功能函数。
7. 如果实现类中要访问接口中的成员，不能使用super关键字。因为两者之间没有显示的继承关系，况且接口中的成员成员属性是静态的。可以使用接口名直接访问。
8. 接口没有构造方法。



## 接口与类、接口之间的关系

1. 大家之前都知道类与类之间的关系继承，那么接口与类之间又是怎样子的关系呢？**接口与类之间是实现关系**。非抽象类实现接口时，必须把接口里面的所有方法实现。类实现接口用关键字implments，类与接口之间是可以多实现的(即一个类可以实现多个接口)。

|  |
| --- |
| **interface** Eraser {  **public** **static** **final** String *color* = "白色";  **public** **abstract** **void** clean();  }  **class** Pencil **implements** Eraser {  String name;  Pencil() {  }  Pencil(String name) {  **this**.name = name;  }  **void** write() {  System.*out*.println("写字");  }  @Override  **public** **void** clean() {  System.*out*.println("涂改...");  }  } |

分析：

原本铅笔没有涂改功能的，但是一旦实现了Eraser接口做了实现，那么就具备了涂改功能，那么接口的作用则是拓展功能。

**2. 接口与接口之间的关系式继承**。

|  |
| --- |
| **interface** A{  **public** **void** show();  }  **interface** B{  **public** **void** print();  }  **interface** C **extends** A,B{    } |

接口与接口之间的关系是继承，接口可以多继承接口.

**练习：**在现实生活中有部分同学在学校期间只会学习，但是有部分学生除了学习外还会赚钱。

# 多态

## 多态的概述

1：什么是多态

一个对象的多种状态

（老师）（员工）（儿子）

教师 a =老钟;

员工 b= 老钟;

2：多态体现

1：Father类

1：非静态成员变量x

2：静态成员变量y

3：非静态方法eat,方法体输出父类信息

4：静态方法speak();方法体输出父类信息

2：Son类

1：非静态成员变量x

2：静态成员变量y

3：非静态方法eat，方法体输出子类信息

4：静态方法speak();方法体输出子类信息

|  |
| --- |
| **class** Father {  **int** x = 1;  **static** **int** *y* = 2;  **void** eat() {  System.*out*.println("开吃");  }  **static** **void** speak() {  System.*out*.println("小头爸爸");  }  }  **class** Son **extends** Father {  **int** x = 3;  **static** **int** *y* = 4;  **void** eat() {  System.*out*.println("大头儿子很能吃");  }  **static** **void** speak() {  System.*out*.println("大头儿子。");  }  }  **class** Demo10 {  **public** **static** **void** main(String[] args) {  Father f = **new** Son(); // 父类引用指向了子类对象。  System.*out*.println(f.x); // 1  System.*out*.println(f.*y*); // 2  f.eat(); // 输出的是子类的。    f.*speak*(); // 输出的是父类    }  } |

3：Son类继承父类

1：创建Father f=new Son();

1：这就是父类引用指向了子类对象。

2：问f.x=?（非静态）

3：问f.y=?（静态）

4：问f.eat()输出的是子类还是父类信息？（非静态）

5：问f.speak()输出的是子类还是父类信息？（静态）

4：总结

1：当父类和子类具有相同的非静态成员变量，那么在多态下访问的是父类的成员变量

2：当父类和子类具有相同的静态成员变量，那么在多态下访问的是父类的静态成员变量

所以：父类和子类有相同的成员变量，多态下访问的是父类的成员变量。

3：当父类和子类具有相同的非静态方法（就是子类重写父类方法），多态下访问的是子类的成员方法。

4：当父类和子类具有相同的静态方法（就是子类重写父类静态方法），多态下访问的是父类的静态方法。

2：多态体现

1：父类引用变量指向了子类的对象

2：父类引用也可以接受自己的子类对象

3：多态前提

1：类与类之间有关系，继承或者实现

4：多态弊端

1：提高扩展性，但是只能使用父类引用指向父类成员。

5：多态特点

非静态

1：编译时期，参考引用型变量所属的类是否有调用的方法，如果有编译通过。没有编译失败

2：运行时期，参考对象所属类中是否有调用的方法。

3：总之成员函数在多态调用时，编译看左边，运行看右边。

在多态中，成员变量的特点，无论编译和运行参考左边（引用型变量所属的类）。

在多态中，静态成员函数特点，无论编译和运行都参考左边

6：多态练习

1：多态可以作为形参，接受范围更广的对象，避免函数重载过度使用。

1：定义功能，根据输出任何图形的面积和周长。

1：定义抽象类abstract MyShape

1：定义抽象方法public abstract double getArea();

2：定义抽象方法public abstract double getLen();

2：定义Rect类继承MyShape

1：定义长和宽成员变量，double width height;

2：无参构造，有参构造。

3：实现父类方法。

3：定义Cricle类继承MyShape

1：定义半径成员变量，和PI常量

2：无参构造，有参构造

3：实现父类方法。

4：定义静态方法计算任意图形的面积和周长

1：未知内容参与运算，不能确定用户传入何种图形，使用多态。

1：形参定义为 MyShape my

2：调用计算面积方法，和计算周长方法。并打印

2：使用多态特性，子类重写了父类非静态方法，会执行子类的方法。

|  |
| --- |
| /\*  多态练习  1：多态可以作为形参，接受范围更广的对象，避免函数重载过度使用。  1：定义功能，根据输出任何图形的面积和周长。  子类重写了父类的抽象方法，多态下，会执行子类的非静态方法。  2：多态可以作为返回值类型。  获取任意一辆车对象  3：抽象类和接口都可以作为多态中的父类引用类型。  \*/  **abstract** **class** MyShape{  **public** **abstract** **double** getArea();  **public** **abstract** **double** getLen();  }  **class** Rect **extends** MyShape{  **double** width ;  **double** height;  Rect(){    }  Rect(**double** width ,**double** height){  **this**.width=width;  **this**.height=height;  }  **public** **double** getArea(){  **return** width\*height;  }  **public** **double** getLen(){  **return** 2\*(width+height);  }    }  **class** Circle **extends** MyShape{  **double** r;  **public** **static** **final** **double** *PI*=3.14;    Circle(){    }    Circle(**double** r){  **this**.r=r;  }  **public** **double** getLen(){  **return** 2\**PI*\*r;  }    **public** **double** getArea(){  **return** *PI*\*r\*r;  }  }  **class** Demo11{  **public** **static** **void** main(String[] args){  System.*out*.println();    *print*(**new** Rect(3,4)); //MyShape m =new Rect(3,4);    *print*(**new** Circle(3));    }    //根据用户传入的图形对象，计算出该图形的面积和周长  //1：多态可以作为形参，接受范围更广的对象，避免函数重载过度使用。  **public** **static** **void** print(MyShape m){  System.*out*.println(m.getLen());  System.*out*.println(m.getArea());  }  } |

2：多态可以作为返回值类型。

获取任意一辆车对象

1：定义汽车类，有名字和颜色，提供有参和无参构造，有运行的行为。

2：定义Bmw类，继承Car类，提供无参构造和有参构造（super父类构造），重写父类运行行为。

3：定义Benz类，继承Car类，提供无参构造和有参构造（super父类构造），重写父类运行行为。

4：定义Bsj类，继承Car类，提供无参构造和有参构造（super父类构造），重写父类运行行为。

5：定义静态方法，汽车工厂，随机生产汽车。使用多态定义方法返回值类型。

1：使用(int)Math.round(Math.random()\*2); 生成0-2之间随机数。

2：使用if else 判断，指定，0,1,2 new 不同汽车 并返回。

6：调用该方法，发现多态的好处。

|  |
| --- |
| \*  2：多态可以作为返回值类型。  获取任意一辆车对象  1：定义汽车类，有名字和颜色，提供有参和无参构造，有运行的行为。  2：定义Bmw类，继承Car类，提供无参构造和有参构造（super父类构造），重写父类运行行为。  3：定义Benz类，继承Car类，提供无参构造和有参构造（super父类构造），重写父类运行行为。  4：定义Bsj类，继承Car类，提供无参构造和有参构造（super父类构造），重写父类运行行为。  5：定义静态方法，汽车工厂，随机生产汽车。使用多态定义方法返回值类型。  1：使用(int)Math.round(Math.random()\*2); 生成0-2之间随机数。  Math 类  2：使用if else 判断，指定，0,1,2 new 不同汽车 并返回。  6：调用该方法，发现多态的好处。  \*/  **class** Car {  String name;  String color;  Car() {  }  Car(String name, String color) {  **this**.name = name;  **this**.color = color;  }  **void** run() {  System.*out*.println("跑跑。。。。");  }  }  **class** Bmw **extends** Car {  Bmw() {  }  Bmw(String name, String color) {  **super**(name, color);  }  **void** run() {  System.*out*.println("宝马很拉风。。。。");  }  }  **class** Benz **extends** Car {  Benz() {  }  Benz(String name, String color) {  **super**(name, color);  }  **void** run() {  System.*out*.println("奔驰商务首选。。。。");  }  }  **class** Bsj **extends** Car {  Bsj() {  }  Bsj(String name, String color) {  **super**(name, color);  }  **void** run() {  System.*out*.println("泡妞首选。。。。");  }  }  **class** Demo12 {  **public** **static** **void** main(String[] args) {  **int** x = 0;  **while** (x < 100) {  Car c = *CarFactory*();  c.run();  x++;  }  }  // 定义静态方法，汽车工厂，随机生产汽车。使用多态定义方法返回值类型。  // 使用随机数，0.1.2 if 0 bsj 1 bmw 2 bc  **public** **static** Car CarFactory() {  **int** x = (**int**) Math.*round*(Math.*random*() \* 2);  **if** (0 == x) {  **return** **new** Bmw("宝马x6", "红色");  } **else** **if** (1 == x) {  **return** **new** Benz("奔驰", "黑色");  } **else** **if** (2 == x) {  **return** **new** Bsj("保时捷", "棕色");  } **else** {  **return** **new** Benz("Smart", "红色");  }  }  } |

3：抽象类和接口都可以作为多态中的父类引用类型。

1：sun Arrays

6：多态之类型转型

1：案例定义Father类

1：定义method1和method2方法

2：定义Son类继承Father类

1：定义method1（重写父类method1）和method2方法

3：创建Father f=new Son();

1： f.method1() 调用的子类或者父类？

2： f.method2() 编译和运行是否通过？

3： f.method3() 编译和运行是否通过？（编译报错）

4：如何在多态下，使用父类引用调用子类特有方法。

1：基本类型转换：

1：自动：小->大

2：强制：大->小

2：类类型转换

前提：继承，必须有关系

1：自动：子类转父类

2：强转：父类转子类

3：类型转换

1：Son s=(Son)f

2：s.method3();

|  |
| --- |
| /\*  如何在多态下，使用父类引用调用子类特有方法。  1：基本类型转换：  1：自动：小->大 int x=1 double d=x;  2：强制：大->小 int y=(int)d;  2：类类型转换  前提：继承，必须有关系  1：自动：子类转父类 Father f=new Son();  2：强转：父类转子类 Son s=(Son)f;  1：类型转换  1：Son s=(Son)f  2：s.method3();  \*/  **class** Father {  **void** method1() {  System.*out*.println("这是父类1");  }  **void** method2() {  System.*out*.println("这是父类2");  }  }  **class** Son **extends** Father {  **void** method1() {  System.*out*.println("这是子类1");  }  **void** method3() {  System.*out*.println("这是子类3");  }  }  **class** Demo14 {  **public** **static** **void** main(String[] args) {  Father f = **new** Son();  f.method1(); // 这是子类1  f.method2(); // 这是父类2  // f.method3(); //编译报错。  // 多态弊端，只能使用父类引用指向父类成员。  // 类类型转换  Son s = (Son) f;  s.method3();  System.*out*.println();  }  } |

5：案例：

1：定义Animal类颜色成员变量，无参构造，有参构造，run方法

2：定义Dog类，继承Animal,定义无参构造，有参构造（使用super调用父类有参构造），Dog的特有方法ProtectHome

3：定义Fish类，继承Animal，定义无参构造，有参构造（使用super调用父类有参构造），Fish特有方法swim

4：定义Bird类，继承Animal，定义无参构造，有参构造（使用super调用父类有参构造），Bird特有方法fly

5：使用多态，Animal a=new Dog();

6：调用Dog的特有方法，ProtectHome

1：类类型转换，Dog d=(Dog)a;

2：d.protectHome

7：非多态

1：Animal a=new Animal();

2：类类型转换

Dog d=(Dog)a;

d.protectHome();

3：编译通过，运行出现异常

1：ClassCastException

8：多态例外

1：Animal a=new Dog();

2：类类型转换

1：Fish f=(Fish)a;

2：f.fish();

3：编译通过，运行异常

1：ClassCastException

4：虽然是多态，但是鸟不能转为狗，狗不能转为鱼，他们之间没有关系。

|  |
| --- |
| **class** Animal {  String color;  Animal() {  }  Animal(String color) {  **this**.color = color;  }  **void** run() {  System.*out*.println("跑跑");  }  }  **class** Dog **extends** Animal {  Dog() {  }  Dog(String color) {  **super**(color);  }  **void** run() {  System.*out*.println("狗儿跑跑");  }  **void** protectHome() {  System.*out*.println("旺旺，看家");  }  }  **class** Fish **extends** Animal {  Fish() {  }  Fish(String color) {  **super**(color);  }  **void** run() {  System.*out*.println("鱼儿水中游");  }  **void** swim() {  System.*out*.println("鱼儿游泳");  }  }  **class** Demo15 {  **public** **static** **void** main(String[] args) {  Animal ani = **new** Dog();  // ani.protectHome();  // 正常转换  Dog d = (Dog) ani;  d.protectHome();  // 多态例外  Animal an = **new** Animal();  // ClassCastException  // Dog d=(Dog)an  // 多态例外  Animal dog = **new** Dog();  // ClassCastException  // Fish f = (Fish) dog;  }  } |

6：案例2

1：定义一功能，接收用户传入动物，根据用于传入的具体动物，执行该动物特有的方法

2：使用多态，方法形参，不能确定用户传入的是那种动物

3：使用instanceof 关键字，判断具体是何种动物，

4：类转换，执行该动物的特有方法。

|  |
| --- |
| **package** oop04;  /\*  案例2  1：定义一功能，接收用户传入动物，根据用于传入的具体动物，执行该动物特有的方法  2：使用多态，方法形参，不能确定用户传入的是那种动物  3：使用instanceof 关键字，判断具体是何种动物，  4：类转换，执行该动物的特有方法。  \*/  **class** Animal {  String color;  Animal() {  }  Animal(String color) {  **this**.color = color;  }  **void** run() {  System.*out*.println("跑跑");  }  }  **class** Dog **extends** Animal {  Dog() {  }  Dog(String color) {  **super**(color);  }  **void** run() {  System.*out*.println("狗儿跑跑");  }  **void** protectHome() {  System.*out*.println("旺旺，看家");  }  }  **class** Fish **extends** Animal {  Fish() {  }  Fish(String color) {  **super**(color);  }  **void** run() {  System.*out*.println("鱼儿水中游");  }  **void** swim() {  System.*out*.println("鱼儿游泳");  }  }  **class** Bird **extends** Animal {  Bird() {  }  Bird(String color) {  **super**(color);  }  **void** run() {  System.*out*.println("鸟儿空中飞");  }  **void** fly() {  System.*out*.println("我是一只小小鸟。。。。");  }  }  **class** Demo16 {  **public** **static** **void** main(String[] args) {  System.*out*.println();  *doSomething*(**new** Dog());  *doSomething*(**new** Bird());  *doSomething*(**new** Fish());  }  // 定义一功能，接收用户传入动物，根据用于传入的具体动物，执行该动物特有的方法  **public** **static** **void** doSomething(Animal a) {  **if** (a **instanceof** Dog) {  Dog d = (Dog) a;  d.protectHome();  } **else** **if** (a **instanceof** Fish) {  Fish f = (Fish) a;  f.swim();  } **else** **if** (a **instanceof** Bird) {  Bird b = (Bird) a;  b.fly();  } **else** {  System.*out*.println("over");  }  }  } |

# 部类

## 内部类的概述

将类定义在另一个类的内部则成为内部类。其实就是类定义的位置发生了变化。

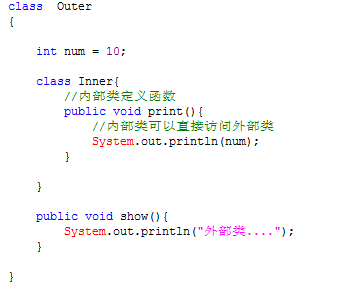
在一个类中，定义在类中的叫成员变量，定义在函数中的叫成员函数，那么根据类定义的位置也可以分为**成员内部类**和**局部内部类**。

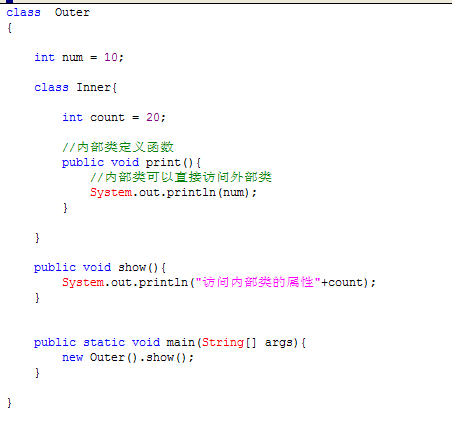
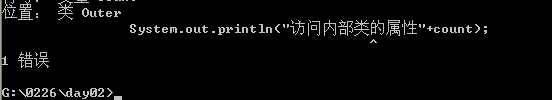
备注：内部类生产的class文件为 “外部类$内部类”，为了标明该内部类是属于具体哪个外部类的。

## 成员内部类的访问方式

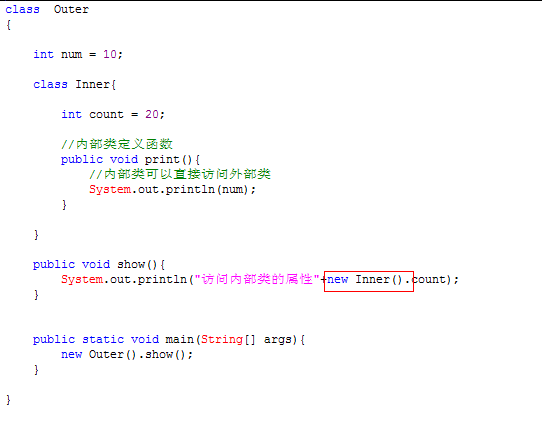
1. 内部类可以直接访问外部类的成员属性。(孙悟空相当于内部类飞到牛魔王的肚子里面去)。
2. 外部类需要访问内部类的成员属性时需要创建内部类的对象。
   * + 1. 在外部类的成员函数中创建内部类的对象，通过内部类对象对象直接访问内部类的成员。
       2. 在其他类中直接创建内部类的对象。

Outer.Inner inner = new Outer().new Inner();

外部类访问内部类的属性



编译异常分析：外部类需要访问内部类的属性时，需要创建内部类的对象访问。



有A类和B类，当A类想要直接访问B类中的成员，而B类又需要建立A类的对象来A类中的成员。这时，就将A类定义成B类的内部类。比喻：孙悟空和铁扇公主。孙悟空到了公主肚子中，就成了内部类（其实是持有外部类的对象引用）。

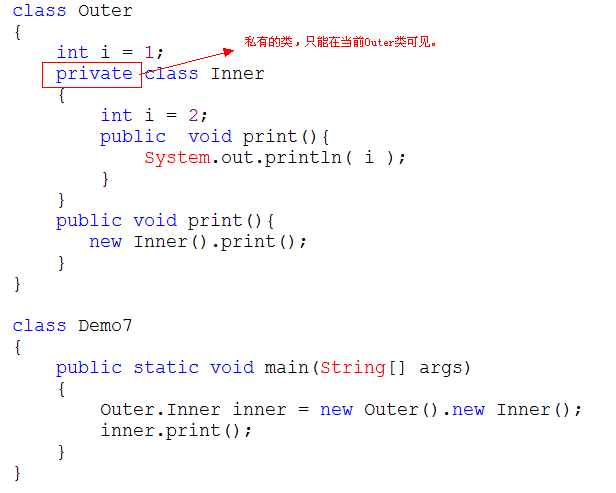
**疑问： 什么时候使用内部类呢？**

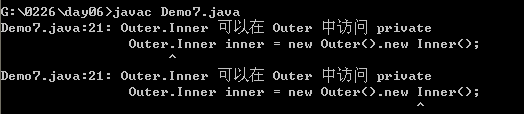
当我们分析事物时，发现事物的内部还有具体的事物，这时则应该定义内部类了。

比如人体是一个类，人体有心脏，心脏的功能在直接访问人体的其他内容。这时就将心脏定义在人体类中，作为内部类存在。

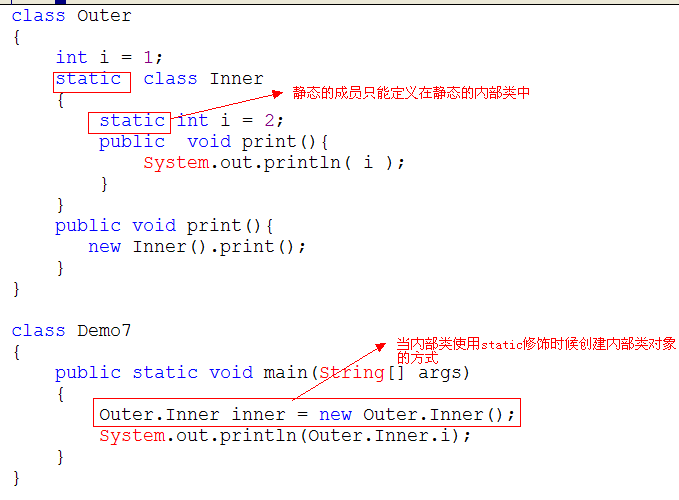
**内部类的优势：成员内部类作为外部类的成员，那么可以访问外部类的任意成员。**

## 成员内部类访问细节





私有的成员内部类不能在其他类中直接创建内部类对象来访问。



如果内部类中包含有静态成员，那么java规定内部类必须声明为静态的访问静态内部类的形式：Outer.Inner in = new Outer.Inner();

**总结：成员内部类(成员属性、成员方法)特点:**

1. **私有的成员内部类**

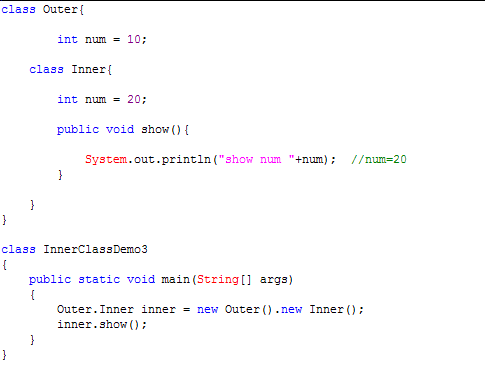
特点：不能在其他类中直接创建内部类对象来访问

1. **静态的成员内部类**

特点：如果内部类中包含有静态成员，那么java规定内部类必须声明为静的

访问静态内部类的形式：

Outer.Inner in = new Outer.Inner();

疑问： 目前打印的num是20，如果想打印10的话，应该怎么做？

解答：这时候其实在show方法中已经存在了两个this对象，一个是外部类对象，一个是内部类对象，所以要在this前面加上类名标明对应的this。

## 局部内部类

**局部内部类概述：**包含在外部类的函数中的内部类称之为局部内部类。

**访问：**可以在包含局部内部类的方法中直接创建局部内部类的对象调用局部内部类的成员。

**注意：**局部内部类只能访问所在函数的fanal属性。



## 匿名内部类

匿名内部类：就是没有类名字的内部类。

匿名内部类作用：简化内部类书写。

匿名内部类的前提:必须继承一个父类或者是实现一个接口。

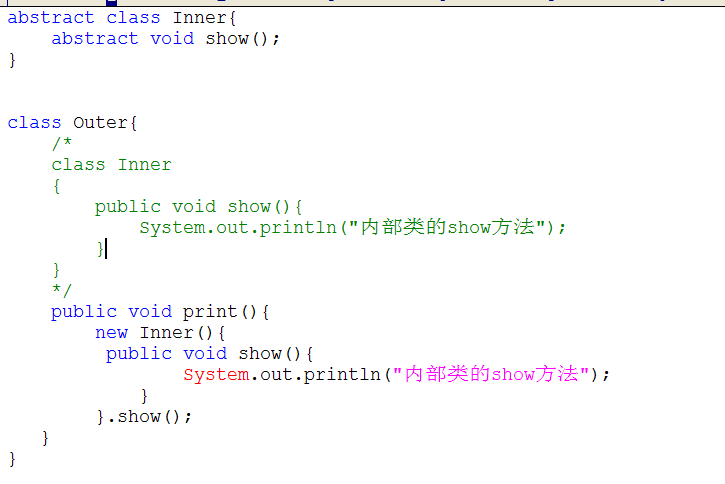
匿名内部类的格式：

new 父类或者接口(){ 执行代码….};

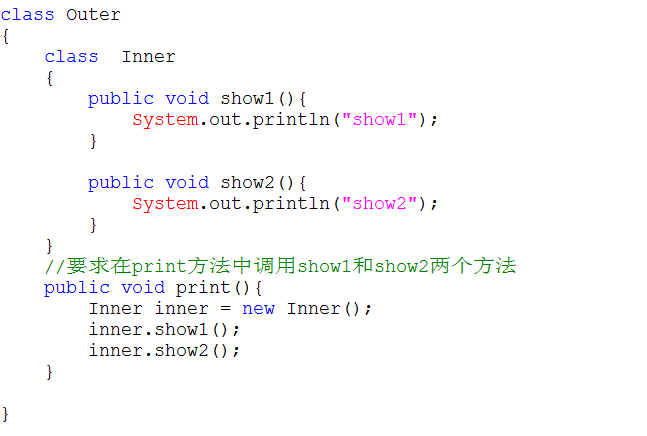
内部类的写法：

|  |
| --- |
| class Outer{  class Inner  {  public void show(){  System.out.println("内部类的show方法");  }  }  public void print(){  new Inner().show();  }  } |

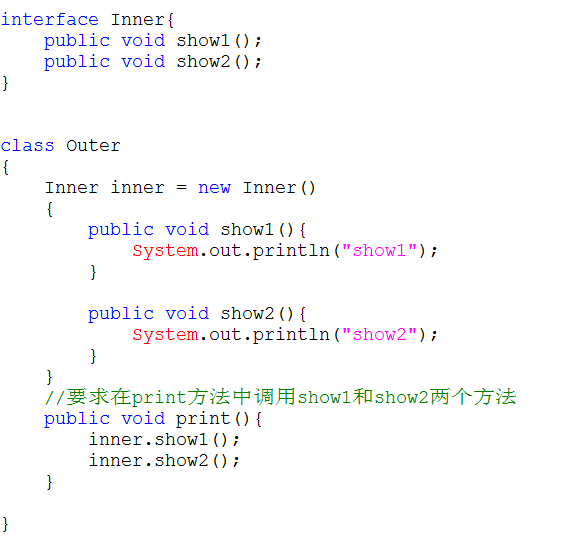
匿名内部类调用show方法:



案例：在外部类调用show1、show2方法。内部类的实现方法/



使用匿名内部类实现：



注意细节：

1.使用匿名内部类时，如果需要调用匿名内部类的两个方法或者两个方法以上。可以使用变量指向该对象。

# 异常

## 现实生活的病

现实生活中万物在发展和变化会出现各种各样不正常的现象。

1：例如：人的成长过程中会生病。

|——病

|——不可治愈(癌症晚期)

|——可治愈

|——小病自行解决(上火,牙痛)

|——去医院(感冒,发烧)

## java异常体系图

现实生活中的很多病况从面向对象的角度考虑也是一类事物，可以定义为类。

java中可以通过类对这一类不正常的现象进行描述，并封装为对象。

1. java的异常体系包含在java.lang这个包默认不需要导入。
2. java异常体系

|——Throwable （实现类描述java的错误和异常）

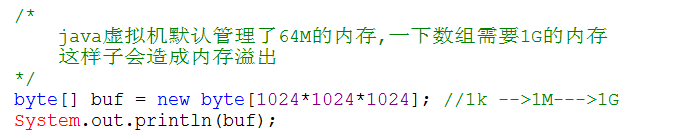
|——Error （错误）一般不通过代码去处理。

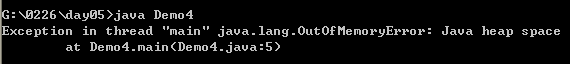
|——Exceprion （异常）

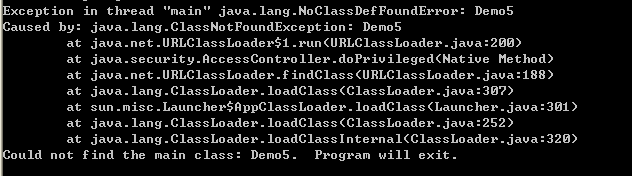
|——RuntimeException （运行时异常）

|——非运行时异常

常见的Error



 错误原因：内存溢出。需要的内存已经超出了java虚拟机管理的内存范围。

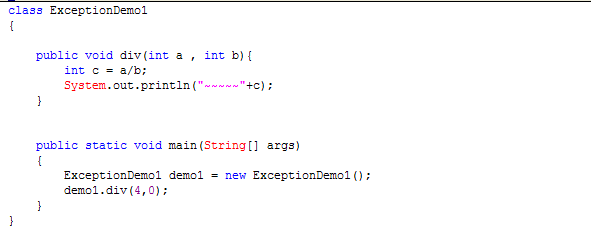


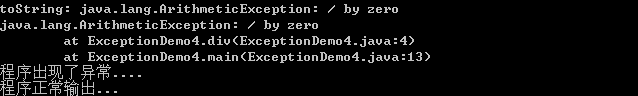
错误原因：找不到类文件。

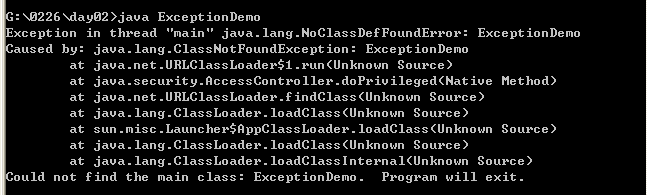
错误(Error):

它指的是一个合理的应用程序不能截获的严重的问题。大多数都是反常的情况。错误是JVM的一个故障(虽然它可以是任何系统级的服务)。所以，错误是很难处理的，一般的开发人员(当然不是你)是无法处理这些错误的。比如内存溢出.

1. 异常体系图的对应







## Throwable类

1. toString() 输出该异常的类名。
2. getMessage() 输出异常的信息，需要通过构造方法传入异常信息（例如病态信息）。
3. printStackTrace() 打印栈信息。

人生病：流鼻涕，感冒，呼吸道感染，肺炎。。。最后体现的是肺炎。

医生要处理需要获知这些信息。从外到里处理。最后找病源

|  |
| --- |
| /\*  Throwable类  printStackTrace() 打印栈信息  肺炎  上呼吸道感染  发烧  流鼻涕感冒  小感冒  \*/  **class** Demo6 {  **public** **static** **void** main(String[] args) {  // Throwable able=new Throwable();  Throwable able = **new** Throwable("想吐。。。");  System.*out*.println(able.toString()); // 输出该异常的类名  System.*out*.println(able.getMessage()); // 输出异常的信息  able.printStackTrace(); // 打印栈信息  }  } |

## 程序中的异常处理

1. 当除数是非0，除法运算完毕，程序继续执行。
2. 当除数是0，程序发生异常，并且除法运算之后的代码停止运行。因为程序发生异常需要进行处理。

|  |
| --- |
| **class** Demo7 {  **public** **static** **void** main(String[] args) {  *div*(2, 0);  System.*out*.println("over");  }  **public** **static** **void** div(**int** x, **int** y) {  //该行代码的y值可能是0,程序会出现异常并停止  System.*out*.println(x / y);  System.*out*.println("除法运算");  }  }  //ArithmeticException |

疑问：  出现异常如何处理？

### 自行处理

1. try{//可能发生异常的代码 }catch(异常类 变量名){//处理}。
2. 案例除法运算的异常处理。
3. 如果没有进行try catch处理，出现异常程序就停止。进行处理后，程序会继续执行。

|  |
| --- |
| **class** Demo7 {  **public** **static** **void** main(String[] args) {  *div*(2, 0);  System.*out*.println("over");  }  **public** **static** **void** div(**int** x, **int** y) {  **try** {  System.*out*.println(x / y); // 可能出现异常的语句，放入try中。  } **catch** (ArithmeticException e) { // 进行异常匹配，  //异常信息  System.*out*.println(e.toString());  System.*out*.println(e.getMessage());  e.printStackTrace();  System.*out*.println("除数不能为0");  }  System.*out*.println("除法运算");  }  } |

**多个异常**

1. 案例print方法，形参中增加数组。
2. 在print方法中操作数组会发生新的异常

ArrayIndexOutOfBoundsException，NullPointerException），如何处理？

* + - 1. 使用将可能发生异常的代码放入try语句中，添加多个catch语句即可。
      2. 可以处理多种异常，但是同时只能处理一种异常。
      3. try中除0异常和数组角标越界同时出现，只会处理一种。

|  |
| --- |
| **public** **class** Demo8 {  **public** **static** **void** main(String[] args) {  System.*out*.println();  **int**[] arr = { 1, 2 };  arr = **null**;  // print (1, 0, arr);  print (1, 2, arr);  System.*out*.println("over");  }  **public** **static** **void** print(**int** x, **int** y, **int**[] arr) {  **try** {  System.*out*.println(arr[1]);  System.*out*.println(x / y);  } **catch** (ArithmeticException e) {  e.toString();  e.getMessage();  e.printStackTrace();  System.*out*.println("算术异常。。。");  } **catch** (ArrayIndexOutOfBoundsException e) {  e.toString();  e.getMessage();  e.printStackTrace();  System.*out*.println("数组角标越界。。。");  } **catch** (NullPointerException e) {  e.toString();  e.getMessage();  e.printStackTrace();  System.*out*.println("空指针异常。。。");  }  System.*out*.println("函数执行完毕");  }  } |

**总结**

1. 程序中有多个语句可能发生异常，可以都放在try语句中。并匹配对个catch语句处理。
2. 如果异常被catch匹配上，接着执行try{}catch(){} 后的语句。没有匹配上程序停止。
3. try中多个异常同时出现，只会处理第一条出现异常的一句，剩余的异常不再处理。
4. 使用多态机制处理异常。
   * + 1. 程序中多态语句出现不同异常，出现了多个catch语句。简化处理（相当于急诊）。
       2. 使用多态，使用这些异常的父类进行接收。（父类引用接收子类对象）

|  |
| --- |
| **public** **static** **void** div(**int** x, **int** y, **int**[] arr, Father f) {  **try** {  System.*out*.println(arr[1]); // 数组越界  System.*out*.println(x / y); // 除零  Son s = (Son) f; // 类型转换  } **catch** (Exception e) {  e.toString();  e.getMessage();  e.printStackTrace();  System.*out*.println("出错啦");  }  System.*out*.println("函数执行完毕");  } |

多个catch语句之间的执行顺序。

* + - 1. 是进行顺序执行，从上到下。
      2. 如果多个catch 内的异常有子父类关系。

子类异常在上，父类在最下。编译通过运行没有问题

父类异常在上，子类在下，编译不通过。(因为父类可以将子类的异常处理，子类的catch处理不到)。

多个异常要按照子类和父类顺序进行catch

|  |
| --- |
| **class** Father {  }  **class** Son **extends** Father {  }  **public** **class** Demo8 {  **public** **static** **void** main(String[] args) {  System.*out*.println();  **int**[] arr = { 1, 2 };  arr = **null**;  Father f = **new** Father();  *div*(1, 0, arr, f);    System.*out*.println("over");  }  **public** **static** **void** div(**int** x, **int** y, **int**[] arr, Father f) {  **try** {  System.*out*.println(arr[1]);  System.*out*.println(x / y);  Son s = (Son) f;  } **catch** (ArithmeticException e) {  e.toString();  e.getMessage();  e.printStackTrace();  System.*out*.println("算术异常。。。");  } **catch** (ArrayIndexOutOfBoundsException e) {  e.toString();  e.getMessage();  e.printStackTrace();  System.*out*.println("数组角标越界。。。");  } **catch** (NullPointerException e) {  e.toString();  e.getMessage();  e.printStackTrace();  System.*out*.println("空指针异常。。。");  } **catch** (Exception e) {  e.toString();  e.getMessage();  e.printStackTrace();  System.*out*.println("出错啦");  }  System.*out*.println("函数执行完毕");  }  } |

**总结**

处理异常应该catch异常具体的子类，可以处理的更具体，不要为了简化代码使用异常的父类。

疑惑：感觉异常没有作用.



### 抛出处理

定义一个功能，进行除法运算例如（div(int x,int y)）如果除数为0，进行处理。

功能内部不想处理，或者处理不了。就抛出使用throw new Exception("除数不能为0"); 进行抛出。抛出后需要在函数上进行声明，告知调用函数者，我有异常，你需要处理如果函数上不进行throws 声明，编译会报错。例如：未报告的异常 java.lang.Exception；必须对其进行捕捉或声明以便抛出throw new Exception("除数不能为0");

|  |
| --- |
| **public** **static** **void** div(**int** x, **int** y) **throws** Exception { // 声明异常，通知方法调用者。  **if** (y == 0) {  **throw** **new** Exception("除数为0"); // throw关键字后面接受的是具体的异常的对象  }  System.*out*.println(x / y);  System.*out*.println("除法运算");  } |

5：main方法中调用除法功能

调用到了一个可能会出现异常的函数，需要进行处理。

1：如果调用者没有处理会编译失败。

如何处理声明了异常的函数。

1：try{}catch(){}

|  |
| --- |
| **public** **static** **void** main(String[] args) {  **try** {  *div*(2, 0);  } **catch** (Exception e) {  e.printStackTrace();  }  System.*out*.println("over");  }  **public** **static** **void** div(**int** x, **int** y) **throws** Exception { // 声明异常，通知方法调用者。  **if** (y == 0) {  **throw** **new** Exception("除数为0"); // throw关键字后面接受的是具体的异常的对象  }  System.*out*.println(x / y);  System.*out*.println("除法运算");  }  } |

2：继续抛出throws

|  |
| --- |
| **class** Demo9 {  **public** **static** **void** main(String[] args) **throws** Exception {  *div*(2, 0);  System.*out*.println("over");  }  **public** **static** **void** div(**int** x, **int** y) **throws** Exception { // 声明异常，通知方法调用者。  **if** (y == 0) {  **throw** **new** Exception("除数为0"); // throw关键字后面接受的是具体的异常的对象  }  System.*out*.println(x / y);  System.*out*.println("除法运算");  }  } |

throw和throws的区别

1. 相同：都是用于做异常的抛出处理的。
2. 不同点：
3. 使用的位置: throws 使用在函数上，throw使用在函数内
4. 后面接受的内容的个数不同:
   * + 1. throws 后跟的是异常类，可以跟多个，用逗号隔开。
       2. throw 后跟异常对象。

|  |
| --- |
| **//throws 处理**  **public** **static** **void** main(String[] args) **throws** InterruptedException {  Object obj = **new** Object();  obj.wait();  } |

|  |
| --- |
| **public** **static** **void** main(String[] args) {  //try catch 处理  Object obj = **new** Object();  **try** {  obj.wait();  } **catch** (InterruptedException e) {    e.printStackTrace();  }  } |

**总结**

* + - 1. try语句不能单独存在，可以和catch、finally组成 try...catch...finally、try...catch、try...finally三种结构。
      2. catch语句可以有一个或多个，finally语句最多一个，try、catch、finally这三个关键字均不能单独使用。
      3. try、catch、finally三个代码块中变量的作用域分别独立而不能相互访问。如果要在三个块中都可以访问，则需要将变量定义到这些块的外面。
      4. 多个catch块时候，Java虚拟机会匹配其中一个异常类或其子类，就执行这个catch块，而不会再执行别的catch块。（子类在上，父类在下）。
      5. throw语句后不允许有紧跟其他语句，因为这些没有机会执行。
      6. 如果一个方法调用了另外一个声明抛出异常的方法，那么这个方法要么处理异常，要么声明抛出。

### 自定义异常

问题：现实中会出现新的病，就需要新的描述。

分析： java的面向对象思想将程序中出现的特有问题进行封装。

案例: 定义功能模拟凌波登录。(例如：lb(String ip))需要接收ip地址

1. 当没有ip地址时，需要进行异常处理。

1. 当ip地址为null是需要throw new Exception("无法获取ip");

2. 但Exception是个上层父类，这里应该抛出更具体的子类。

3. 可以自定义异常

1. 自定义描述没有IP地址的异常（NoIpException）。

1. 和sun的异常体系产生关系。继承Exception类，自定义异常类名也要规范，结尾加上Exception,便于阅读

|  |
| --- |
| /\*  自定义异常  \*/  **class** NoIpException **extends** Exception {  NoIpException() {  }  NoIpException(String message) {  **super**(message);  }  }  **class** Demo10 {  **public** **static** **void** main(String[] args) **throws** NoIpException {  System.*out*.println();  String ip = "192.168.10.252";  ip = **null**;  **try** {  *Lb*(ip);  } **catch** (NoIpException e) {  System.*out*.println("程序结束");  }  }  /\*  \*  \* 凌波教学  \*/  **public** **static** **void** Lb(String ip) **throws** NoIpException {  **if** (ip == **null**) {  // throw new Exception("没插网线吧，小白");  **throw** **new** NoIpException("没插网线吧，小白");  }  System.*out*.println("醒醒了，开始上课了。");  }  } |

案例：模拟吃饭没带钱的问题

1. 定义吃饭功能，需要钱。（例如：eat(double money)）
2. 如果钱不够是不能吃放，有异常。
3. 自定义NoMoneyException();继承Exception 提供有参无参构造，调用父类有参构造初始化。at 方法进行判断，小于10块，throw NoMoneyException("钱不够");
4. eat 方法进行声明，throws NoMoneyException
5. 如果钱不够老板要处理。调用者进行处理。try{}catch(){} 。

|  |
| --- |
| **class** NoMoneyException **extends** Exception {  NoMoneyException() {  }  NoMoneyException(String message) {  **super**(message);  }  }  **class** Demo11 {  **public** **static** **void** main(String[] args) {  System.*out*.println();  **try** {  *eat*(0);  } **catch** (NoMoneyException e) {  System.*out*.println("跟我干活吧。");  }  }    **public** **static** **void** eat(**double** money) **throws** NoMoneyException {  **if** (money < 10) {  **throw** **new** NoMoneyException("钱不够");  }  System.*out*.println("吃桂林米粉");  }  } |

## 运行时异常和非运行时异常

### RuntimeException

RunntimeException的子类：

ClassCastException

多态中，可以使用Instanceof 判断，进行规避

ArithmeticException

进行if判断，如果除数为0，进行return

NullPointerException

进行if判断，是否为null

ArrayIndexOutOfBoundsException

使用数组length属性，避免越界

这些异常时可以通过程序员的良好编程习惯进行避免的

1：遇到运行时异常无需进行处理，直接找到出现问题的代码，进行规避。

2：就像人上火一样牙疼一样，找到原因，自行解决即可

3：该种异常编译器不会检查程序员是否处理该异常

4：如果是运行时异常，那么没有必要在函数上进行声明。

6：案例

1：除法运算功能（div(int x,int y)）

2：if判断如果除数为0，throw new ArithmeticException();

3：函数声明throws ArithmeticException

4：main方法调用div,不进行处理

5：编译通过，运行正常

6：如果除数为0，报异常，程序停止。

7：如果是运行时异常，那么没有必要在函数上进行声明。

1：Object类中的wait()方法，内部throw了2个异常 IllegalMonitorStateException InterruptedException

1：只声明了一个(throws) IllegalMonitorStateException是运行是异常没有声明。

|  |
| --- |
| **class** Demo12 {  **public** **static** **void** main(String[] args){  *div*(2, 1);  }  **public** **static** **void** div(**int** x, **int** y) {  **if** (y == 0) {  **throw** **new** ArithmeticException();  }  System.*out*.println(x / y);  }  } |

### 非运行时异常(受检异常)

如果出现了非运行时异常必须进行处理throw或者try{}catch(){}处理，否则编译器报错。

1；IOException 使用要导入包import java.io.IOException;

2：ClassNotFoundException

2；例如人食物中毒，必须进行处理，要去医院进行处理。

3：案例

1：定义一测试方法抛出并声明ClassNotFoundException（test()）

2：main方法调用test

3：编译报错

1：未报告的异常 java.lang.ClassNotFoundException；必须对其进行捕捉或声明以便抛出

|  |
| --- |
| **public void isFile(String path){**  **try**  **{**  **/\***  **根据文件的路径生成一个文件对象，如果根据该路径找不到相应的文件，**  **则没法生成文件对象。**  **\*/**  **File file = new File(path);**  **//读取文件的输入流**  **FileInputStream input = new FileInputStream(file);**  **//读取文件**  **input.read();**  **}**  **catch (NullPointerException e)**  **{**  **System.out.println("读取默认的文件路径..");**  **}**    **}** |

4：Sun 的API文档中的函数上声明异常，那么该异常是非运行是异常，

调用者必须处理。

5：自定义异常一般情况下声明为非运行时异常

2：函数的重写和异常

1：运行时异常

1：案例定义Father类，定义功能抛出运行是异常，例如（test() throw

ClassCastException）

2：定义Son类，继承Father类，定义test方法，没有声明异常

3：使用多态创建子类对象，调用test方法

4：执行子类方法

1：函数发生了重写，因为是运行时异常，在父类的test方法中，可以声明throws 也可以不声明throws

|  |
| --- |
| **class** Father {  **void** test() **throws** ClassCastException { // 运行时异常  System.*out*.println("父类");  **throw** **new** ClassCastException();  }  }  **class** Son **extends** Father {  **void** test() {  System.*out*.println("子类");  }  }  **class** Demo14 {  **public** **static** **void** main(String[] args) {  Father f = **new** Son();  f.test();  }  } |

2：非运行时异常

1：定义父类的test2方法，抛出非运行时异常，例如抛出ClassNotFoundException

1：此时父类test2方法必须声明异常，因为是非运行时异常

2：Son类定义test2 方法，抛出和父类一样的异常，声明异常

3：使用多态创建子类对象，调用test方法，调用test2方法，

1：声明非运行时异常的方法，在调用时需要处理，所以在main方法调用时throws

2：实现了重写，执行子类的test2方法

3：总结子类重写父类方法可以抛出和父类一样的异常，或

者不抛出异常。

|  |
| --- |
| // 1 子类覆盖父类方法父类方法抛出异常，子类的覆盖方法可以不抛出异常  **class** Father {  **void** test() **throws** ClassNotFoundException { // 非运行时异常  System.*out*.println("父类");  **throw** **new** ClassNotFoundException();  }  }  **class** Son **extends** Father {  **void** test() {  System.*out*.println("子类");  // 父类方法有异常，子类没有。  }  }  **class** Demo14 {  **public** **static** **void** main(String[] args) **throws** ClassNotFoundException {  Father f = **new** Son();  f.test();  }  } |

4：子类抛出并声明比父类大的异常例如子类test2方法抛出Exception

1：编译失败，无法覆盖

2：子类不能抛出父类异常的父类。

3：总结子类不能抛出比父类的异常更大的异常。

|  |
| --- |
| //2：子类覆盖父类方法不能比父类抛出更大异常  **class** Father {  **void** test() **throws** Exception {  // 非运行时异常  System.*out*.println("父类");  **throw** **new** Exception();  }  }  **class** Son **extends** Father {  **void** test() **throws** ClassNotFoundException { // 非运行时异常  System.*out*.println("子类");  **throw** **new** ClassNotFoundException();  }  }  **class** Demo14 {  **public** **static** **void** main(String[] args) **throws** Exception {  Father f = **new** Son();  f.test();  }  } |

3：总结

1：子类覆盖父类方法是，父类方法抛出异常，子类的覆盖方法可以不抛

出异常，或者抛出父类方法的异常，或者该父类方法异常的子类。

2：父类方法抛出了多个异常，子类覆盖方法时，只能抛出父类异常的子

集

3：父类没有抛出异常子类不可抛出异常

1：子类发生非运行时异常，需要进行try{}catch的(){}处理，不能

抛出。

4：子类不能比父类抛出更多的异常

## finally

1: 实现方式一：

try{ // 可能发生异常的代码 } catch( 异常类的类型 e ){ // 当发生指定异常的时候的处理代码 }catch...

比较适合用于专门的处理异常的代码，不适合释放资源的代码。

2：实现方式二：

try{ } catch(){} finally{ // 释放资源的代码 }

finally块是程序在正常情况下或异常情况下都会运行的。

比较适合用于既要处理异常又有资源释放的代码

3：实现方式三

try{ }finally{ // 释放资源 }

比较适合处理的都是运行时异常且有资源释放的代码。

4：finally:关键字主要用于释放系统资源。

1：在处理异常的时候该语句块只能有一个。

2：无论程序正常还是异常，都执行finally。

5：finally是否永远都执行？

1：只有一种情况，但是如果JVM退出了System.exit(0)，finally就不执行。

2：return都不能停止finally的执行过程。

6：案例使用流

1：使用FileInputStream加载文件。

导包import java.io.FileInputStream;

2：FileNotFoundException

导入包import java.io.FileNotFoundException;

3：IOException

import java.io.IOException;

|  |
| --- |
| **public** **class** FinallyDemo {  // 本例子使用finally 关闭系统资源。  **public** **static** **void** main(String[] args) {  FileInputStream fin = **null**;  **try** {  System.*out*.println("1创建io流可能出现异常");  fin = **new** FileInputStream("aabc.txt"); // 加载硬盘的文本文件到内存，通过流  // System.out.println(fin);  } **catch** (FileNotFoundException e) {  System.*out*.println("2没有找到abc.txt 文件");  System.*out*.println("3catch 了");  // System.exit(0);  // return;  }  // finally  **finally** {  System.*out*.println("4fianlly执行");  **if** (fin != **null**) { // 如果流对象为null 流对象就不存在，没有必要关闭资源  **try** {  fin.close();  } **catch** (IOException e) {  e.printStackTrace();  System.*out*.println("close 异常");  }  }  System.*out*.println("5finally over");  }  System.*out*.println("6mainover");  }  }  // 2：无论程序正常还是异常，都执行finally。 但是遇到System.exit(0); jvm退出。  // finally用于必须执行的代码， try{} catch(){}finally{}  // try{}finally{} |

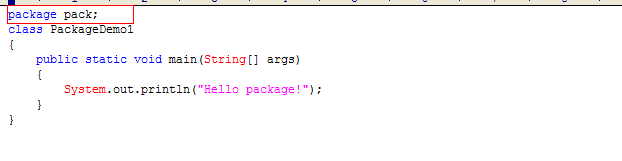
# 包机制

问题：当定义了多个类的时候，可能会发生类名的重复问题。

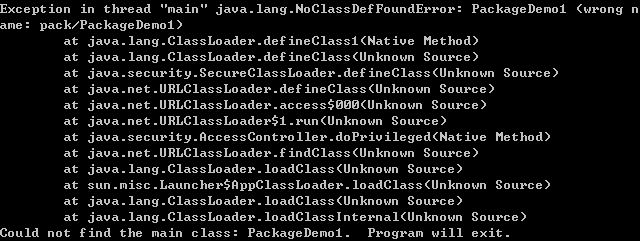
在java中采用包机制处理开发者定义的类名冲突问题。

怎么使用java的包机制呢？

1. 使用package 关键字。
2. package 包名。

问题：

1. javac PackDemo1.java编译没有问题。
2. java PackDemo1 运行出错。



错误原因分析：

在当前目录下找不到有pack目录，更加找不到pack目录下面的PackageDemo1.java文件。

解决办法：

1. 自己在当前目录下新建一个pack目录。
2. 执行Java pagk.PackageDemo1命令。(包其实就是文件夹).

存在的问题：使用包机制的话，我们是否每次都要自己创建一个文件夹呢？

解决：

在编译的时候则可以指定类文件存放的文件夹了。

javac -d . PackageDemo1.java -d 后面跟着就是包名，指定包存放的路径。

**包的优点**

1. 防止类文件冲突。
2. 使源文件与类文件分离，便已软件最终发布。

**注意细节**

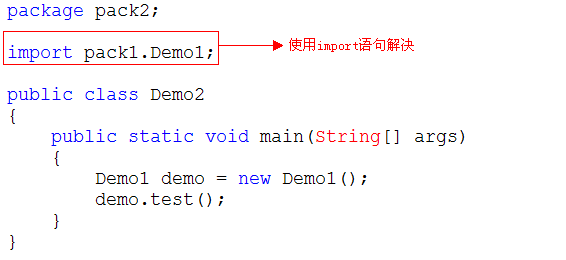
1. 一个java类只能定义在一个包中。
2. 包语句肯定是描述类的第一条语句。

**包机制引发的问题**

有了包之后访问类每次都需要把包名和类名写全。

**解决:**使用import语句。

**格式：** import 包名.类名 ;



**注意细节：**

1. **如果想使用一个包中的许多类时，这时不需要多条的导入语句，使用 “\*”号通配符代表所有的类。**
2. **使用\*时不能导入包中的子类包的class文件。**
3. **import语句可以是多条.**

# 访问修饰符

访问修饰符是用来控制类、属性、方法的可见性的关键字称之为访问修饰符。



1. public 一个类中，同一包中，子类中，不同包中
2. protected 一个类中，同一包中，子类中
3. default 一个类中，同一包中
4. private 一个类中
5. (修饰类成员)类成员
6. 成员使用private修饰只在本类中使用。
7. 如果一个成员没有使用任何修饰符，就是default，该成员可以被包中的其
8. 他类访问。
9. protected成员被protected修饰可以被包中其他类访问，并且位于不同
10. 包中的子类也可以访问。
11. public修饰的成员可以被所有类访问。
12. (修饰类)类
13. 类只有两种public和默认(成员内部类可以使用private))
14. 父类不可以是private和protected，子类无法继承
15. public类可以被所有类访问
16. 默认类只能被同一个包中的类访问

# Jar包

1：**jar**就是打包文件

jar文件时一种打包文件java active File,与zip兼容，称之为jar包

开发了很多类，需要将类提供给别人使用，通常以jar包形式提供.当项目写完之后，需要及将class字节码文件打包部署给客户。如何打包？可以使用jar命令.

2：jar命令

1：jar工具存放于jdk的bin目录中(jar.exe)

2：jar工具：主要用于对class文件进行打包(压缩)

3：dos中输入jar查看帮助

3：案例使用jar命令  
将day10中的cn文件打包为名字为test.jar文件(cn 文件是使用javac -d 编译带包的class文件夹)

**jar cvf test.jar cn**

**详细命令:**

1：jar cf test.jar cn 在当前目录生成test.jar 文件，没有显示执行过程

2：jar cvf test.jar cn 显示打包中的详细信息

3：jar tf test.jar 显示jar文件中包含的所有目录和文件名

4：jar tvf test.jar 显示jar文件中包含的所有目录和文件名大小，创建时间详

细信息

5：jar xf test.jar 解压test.jar到当前目录，不显示信息

6：jar xvf test.jar 解压test.jar到当前目录，显示详细信息

7：可以使用WinRaR进行jar解压

8；将两个类文件归档到一个名为 test2.jar 的归档文件中：

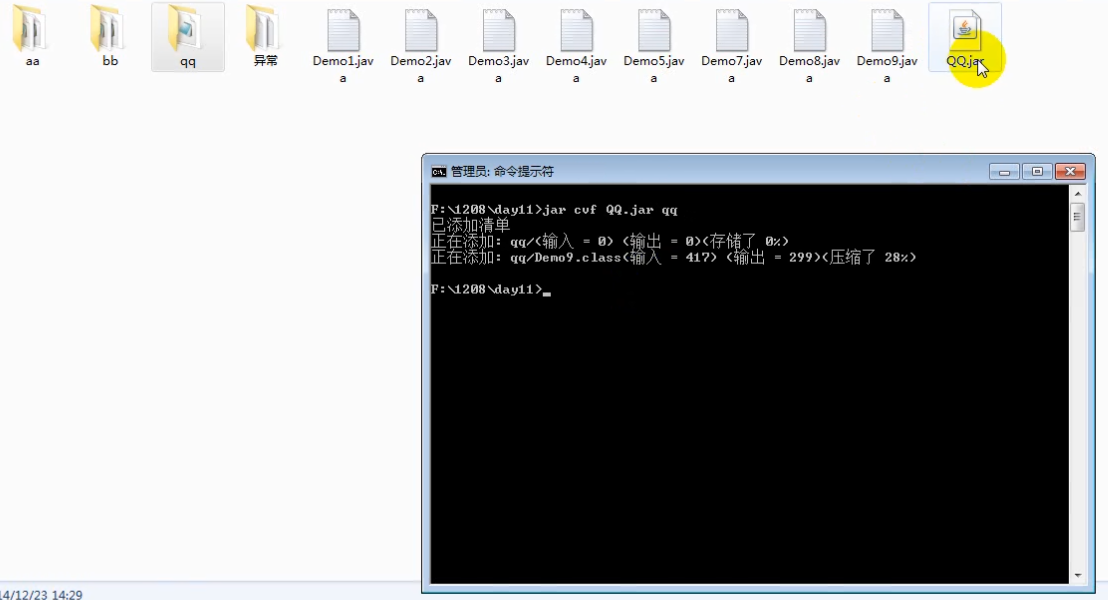
jar cvf test2.jar Demo3.class Demo4.class

9：重定向

1：tvf可以查看jar文件内容，jar文件大，包含内容多，dos看不全。

2：查看jdk中的rt.jar 文件 jar tvf rt.jar

3：jar tvf rt.jar>d:\rt.txt



# Object对象

面向对象的核心思想:“找合适的对象，做适合的事情”。

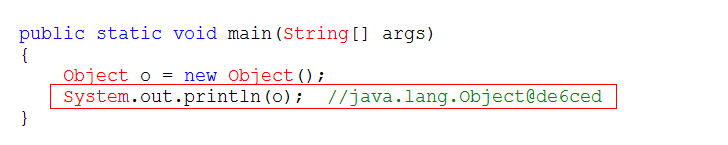
合适的对象：

1. 自己描述类，自己创建对象。
2. sun已经描述了好多常用的类，可以使用这些类创建对象。

API（Application Program Interface）

sun定义的那么多类的终极父类是Object。Object描述的是所有类的通用属性与方法。

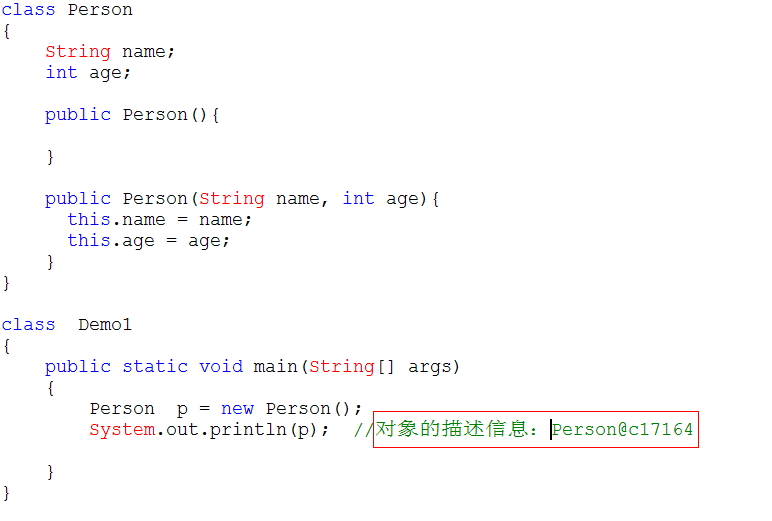
## toString方法



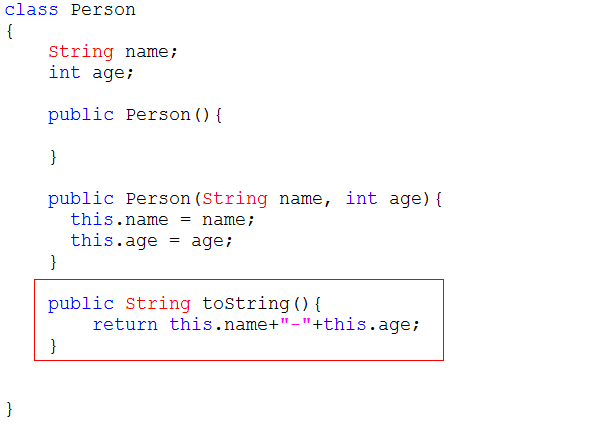
toString() 返回对象的描述信息 java.lang.Object@de6ced 类名@哈希码值的十六进制形式。

直接输入一个对象的时候，会调用对象的toString方法。

练习：自定义一个Person类，打印该对象的描述信息，要求描述信息为：姓名 — 年龄



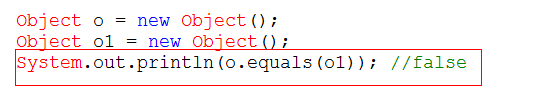
问题：调用p的toString方法时，打印出来的信息是类名+内存地址值。不符合要求。根据我们之前学的继承，假如父类的指定的功能不能满足要求，那么子类可以复写父类的功能函数。那么该对象再调用toString()方法时，则会调用子类复写的toString方法。

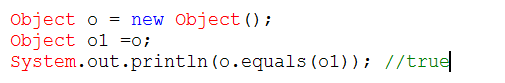


**编程习惯：**开发者要对自定义的类重写toString()，对对象做详细的说明

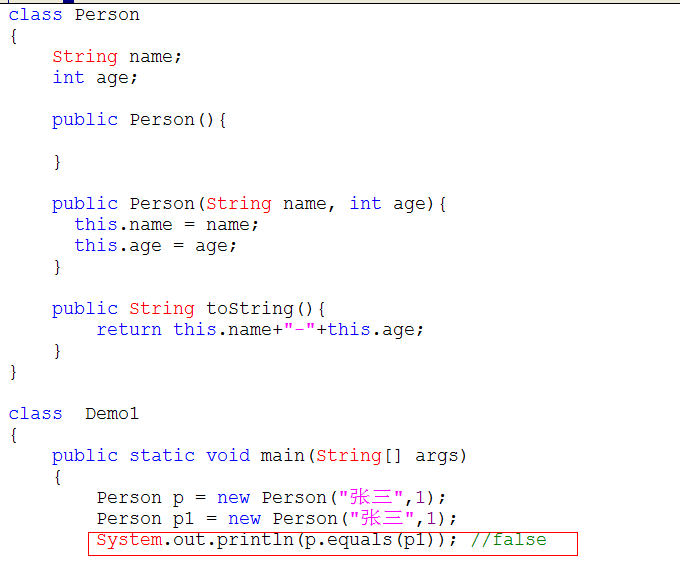
## equals方法

**equals()** 返回的是比较的结果 如果相等返回true，否则false，比较的是对象的内存地址值。



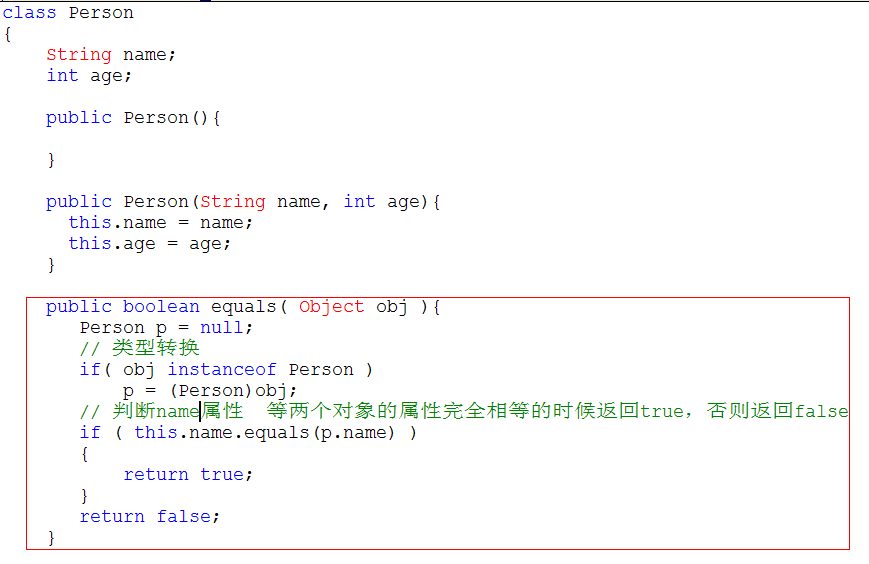


问题：比较两个人是否是同一个人，根据两个人的名字判断。



**问题：**如果根据名字去作为判断两个人是否是同一个时，明显p与p1是同一个人，但是程序输入却不是同一个人。不符合我们现实生活的要求。

**解决：**根据我们学的继承中的函数复写，如果父类的函数不能满足我们目前的要求，那么就可以在子类把该功能复写，达到复合我们的要求。



**编程习惯：**开发者要对自定义的类重写equals()，使得比较两个对象的时候比较对象的属性是否相等，而不是内存地址。

## hashCode方法

**hashCode()** 返回该对象的哈希码值： 采用操作系统底层实现的哈希算法。 同一个对象的哈希码值是唯一的。

java规定如果两个对象equals返回true，那么这两个对象的hashCode码必须一致。



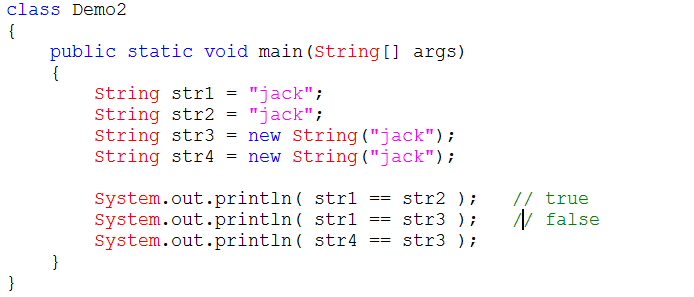
# String类

String类描述的是文本字符串序列。 留言 QQ 写日志。

**创建String类的对象的两种方式：**

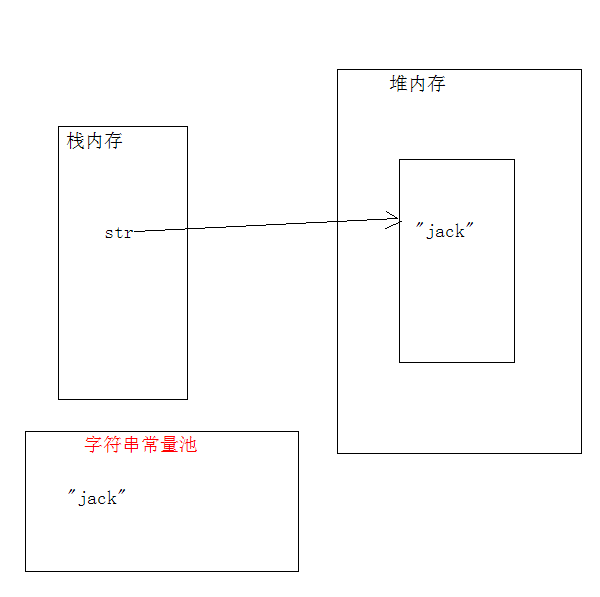
1. ""直接赋值法
2. new关键字法

## 字符串对象的比较



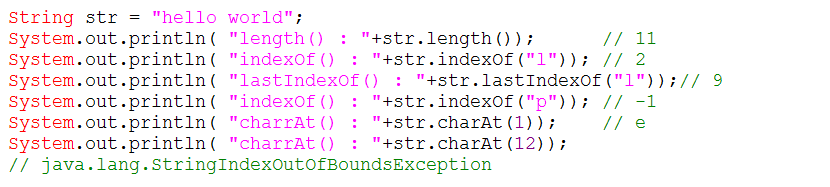
String Str = “jack”这个语句会先检查字符串常量池是否存放这个”jack1”这个字符串对象，如果没有存在，那么就会在字符串常量池中创建这个字符串对象，如果存在直接返回该字符串的内存地址值。

String str3 = new String(“jack”) 该语句会创建两个对象,首先会先检查字符串常量池中存不存在jack这个字符串对象，如果不存在就会创建，如果存在就返回内存地址值。创建了出来之后，new String这个语句就会在堆内存中开辟一个字符串对象。总共两个对象。



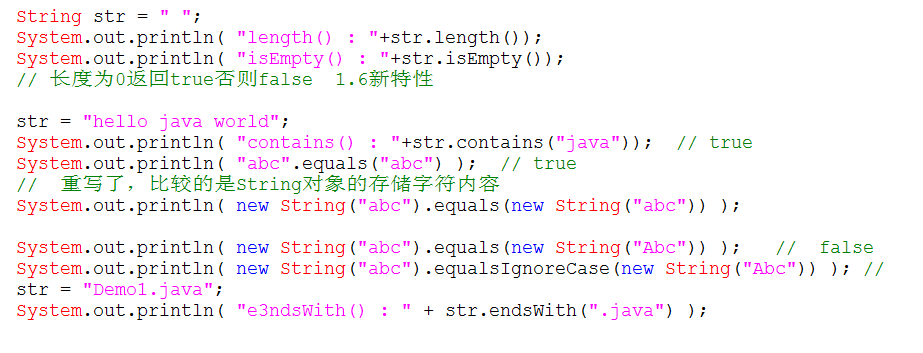
## 获取方法

|  |
| --- |
| int length() 获取字符串的长度  char charAt(int index) 获取特定位置的字符 (角标越界)  int indexOf(String str) 获取特定字符的位置(overload)  int lastIndexOf(int ch) 获取最后一个字符的位置 |



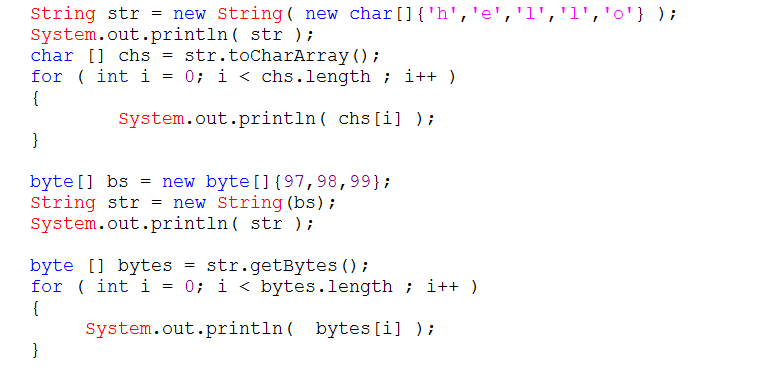
## 判断方法

|  |
| --- |
| boolean endsWith(String str) 是否以指定字符结束  boolean isEmpty()是否长度为0 如：“” null V1.6  boolean contains(CharSequences) 是否包含指定序列 应用：搜索  boolean equals(Object anObject) 是否相等  boolean equalsIgnoreCase(String anotherString) 忽略大小写是否相等 |



## 转换方法

|  |
| --- |
| String(char[] value) 将字符数组转换为字符串  String(char[] value, int offset, int count)  Static String valueOf(char[] data)  static String valueOf(char[] data, int offset, int count)  char[] toCharArray() 将字符串转换为字符数组 |



## 其他方法

|  |
| --- |
| String replace(char oldChar, char newChar) 替换  String[] split(String regex) 切割  String substring(int beginIndex)  String substring(int beginIndex, int endIndex)截取字串  String toUpperCase() 转大写  String toLowerCase() 转小写  String trim() 去除空格 |

## 练习

1. 去除字符串两边空格的函数。

|  |
| --- |
| **public** **class** Demo1 {  // 定义一个祛除字符串两边空格的函数  **public** **static** String trim( String str ){    // 0、定义求字串需要的起始索引变量  **int** start = 0;  **int** end = str.length()-1;  // 1. for循环遍历字符串对象的每一个字符  **for** (**int** i = 0; i<str.length() ; i++ )  {  **if** ( str.charAt(i) == ' ' )  {  start++;  }**else**{    **break**;  }  }  System.out.println( start );  **for** (; end<str.length() && end >= 0; )  {  **if** ( str.charAt(end) == ' ' )  {  end--;  }**else**{  **break**;  }  }  System.out.println( end );  // 2. 求子串  **if**( start < end ){    **return** str.substring( start , (end+1) );  }**else**{    **return** "\_";  } |

1. 获取上传文件名 "D:\\20120512\\day12\\Demo1.java"。

|  |
| --- |
| **public** **static** String getFileName2( String path ){  **return** path.substring( path.lastIndexOf("\\") + 1 );  }  } |

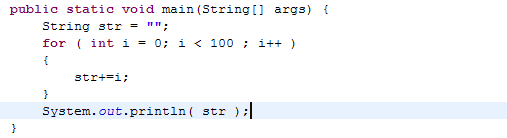
1. 将字符串对象中存储的字符反序。

|  |
| --- |
| // 将字符串对象中存储的字符反序  **public** **static** String reaverseString( String src ){    // 1. 将字符串转换为字符数组  **char** chs[] = src.toCharArray();  // 2. 循环交换  **for** ( **int** start = 0 , end = chs.length - 1; start < end ; start++,end-- )  {  // 3. 数据交换  **char** temp = chs[end];  chs[end] = chs[start];  chs[start] = temp;  }  // 4. 将字符数组转换为字符串  **return** **new** String( chs );  } |

4. 求一个子串在整串中出现的次数

|  |
| --- |
| **public** **static** **int** getCount( String src , String tag ){  // 0. 定义索引变量和统计个数的变量  **int** index = 0;  **int** count = 0;  // 1. 写循环判断  **while** ( ( index = src.indexOf(tag) ) != -1 ) // jackjava  {  // 2. 求字串  System.*out*.println( src );  src = src.substring( index + tag.length() ); // index 4 + 4 = 8  System.*out*.print( src.length() + " : " + index + " : " + tag.length() );  // 3. 累加  count++;  }  **return** count;  } |

# StringBuffer



**StringBuffer** : 由于String是不可变的，所以导致String对象泛滥，在频繁改变字符串对象的应用中，需要使用可变的字符串缓冲区类。

特点：

1. 默认缓冲区的容量是16。
2. StringBuffer ： 线程安全的所有的缓冲区操作方法都是同步的。效率很低。

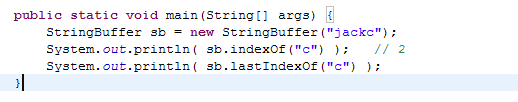
## 添加方法

|  |
| --- |
| StringBuffer("jack") 在创建对象的时候赋值  append() 在缓冲区的尾部添加新的文本对象  insert() 在指定的下标位置添加新的文本对象 |

|  |
| --- |
| StringBuffer sb = **new** StringBuffer("jack");  sb.append(**true**);  sb.append('a');  sb.append(97).append(34.0).append(**new** **char**[]{'o','o'}); // 链式编程  System.*out*.println( sb.toString() ); // 输出缓冲区的中文本数据  sb = **new** StringBuffer("jack");  sb.insert( 2, "java" ); // jajavack  System.*out*.println( sb.toString() ); |

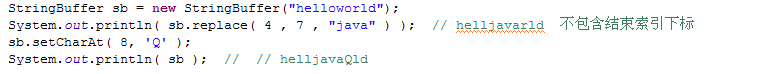
## 查看

|  |
| --- |
| toString() 返回这个容器的字符串  indexOf(String str) 返回第一次出现的指定子字符串在该字符串中的索引。  substring(int start) 从开始的位置开始截取字符串 |

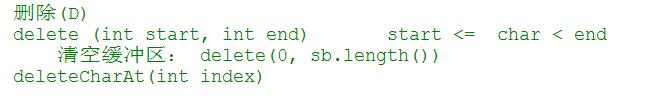


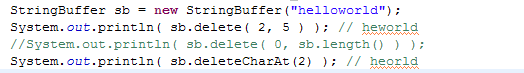
## 修改(U)

|  |
| --- |
| replace(int start int endString str) 使用给定 String 中的字符替换此序列的子字符串中的字符。该子字符串从指定的 start 处开始，一直到索引 end - 1 处的字符  setCharAt(int index char ch) 指定索引位置替换一个字符 |



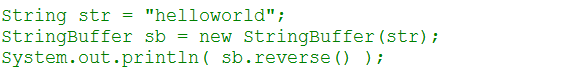
## 删除(D)





## 反序

reverse() 把字符串反序输出。



# StringBuilder

StringBuilder 是JDK1.5之后提出的，线程不安全，但是效率要高。用法与StringBuffer类似。

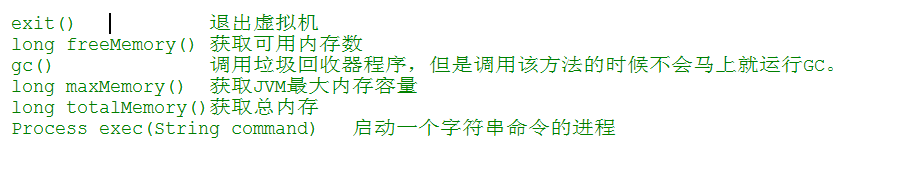
# System

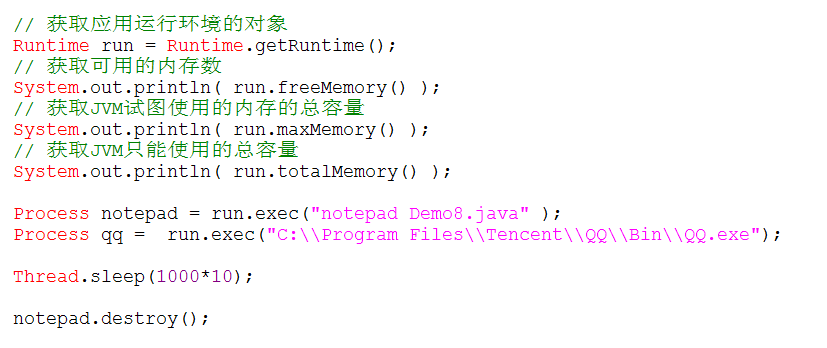
System 可以获取系统的属性。



# Runtime

Runtime 类主要描述的是应用程序运行的环境。

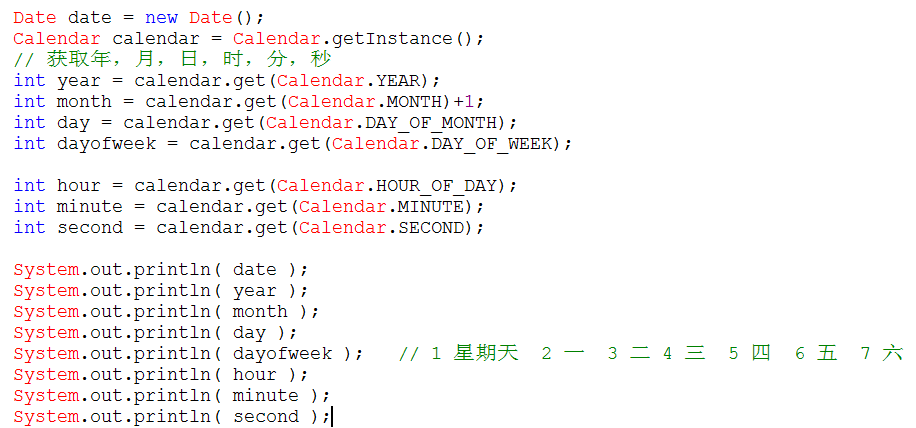




# Date

Date 类封装的是系统的当前时间.。但是Date已经过时了，sun推荐使用Calendar类。

Calendar: 该类是一个日历的类，封装了年月日时分秒时区。

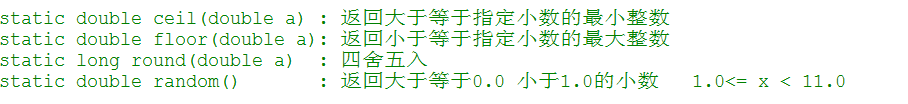


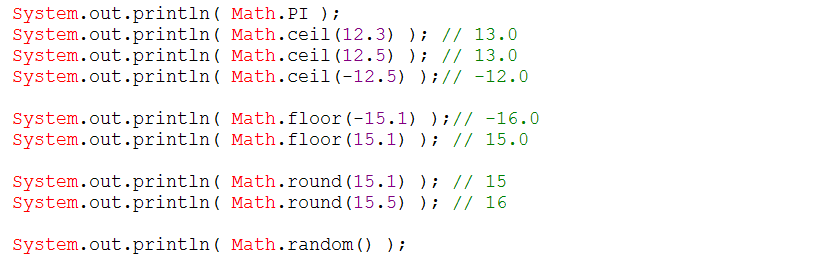
日期格式化类：SimpleDateFormat



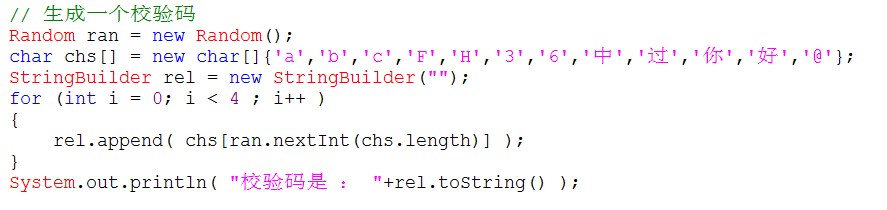
# Math

Math：类封装了很多数学的功能。





练习：生成一个随机码



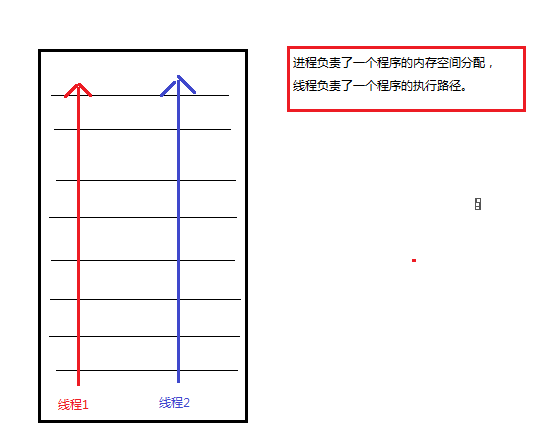
# 线程的概述

进程：正在运行的程序，负责了这个程序的内存空间分配，代表了内存中的执行区域。

线程：就是在一个进程中负责一个执行路径。

多线程：就是在一个进程中多个执行路径同时执行。

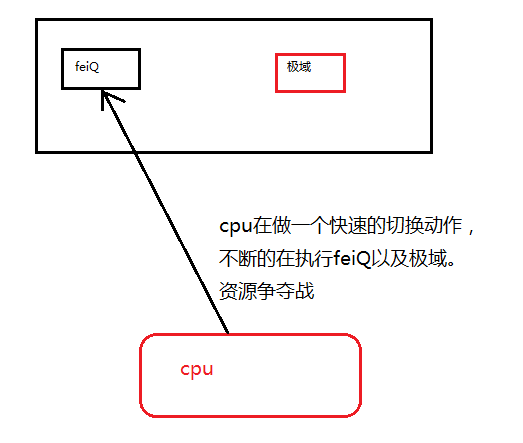




图上的一键优化与垃圾清除同时在运行，在一个进程中同时在执行了多个任务。

**假象：**

电脑上的程序同时在运行。“多任务”操作系统能同时运行多个进程（程序）——但实际是由于CPU分时机制的作用，使每个进程都能循环获得自己的CPU时间片。但由于轮换速度非常快，使得所有程序好象是在“同时”运行一样。



**多线程的好处：**

1. 解决了一个进程里面可以同时运行多个任务（执行路径）。
2. 提供资源的利用率，而不是提供效率。

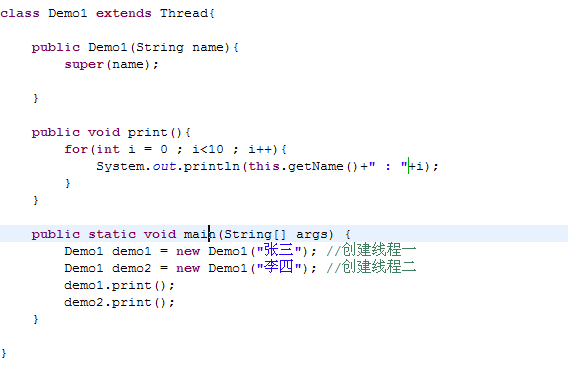
**多线程的弊端:**

1. 降低了一个进程里面的线程的执行频率。
2. 对线程进行管理要求额外的 CPU开销。线程的使用会给系统带来上下文切换的额外负担。
3. 公有变量的同时读或写。当多个线程需要对公有变量进行写操作时,后一个线程往往会修改掉前一个线程存放的数据，发生线程安全问题。
4. 线程的死锁。即较长时间的等待或资源竞争以及死锁等多线程症状。

# 创建线程的方式

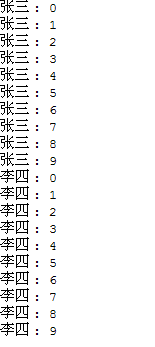
## 创建线程的方式一

1. **继承Thread类**



getName()是获取线程的名字。

执行后的效果：

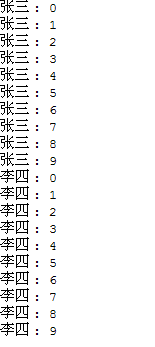


问题： 先按照顺序运行完了张三，然后接着再按照顺序运行完李四，我们想要的效果是张三和李四做资源的争夺战，也就是先是张三然后李四，没有顺序的执行。这就证明多线程没有起到效果。

1. **需要复写run方法，把要执行的任务放在run方法中。**



**运行效果：**

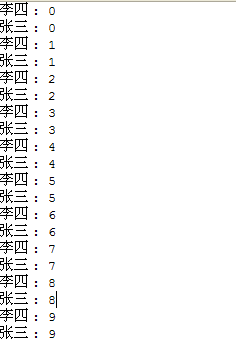


问题： 先按照顺序运行完了张三，然后接着再按照顺序运行完李四，我们想要的效果是张三和李四做资源的争夺战，也就是先是张三然后李四，没有顺序的执行。这就证明多线程没有起到效果。

1. **调用start()方法启动线程**



效果：

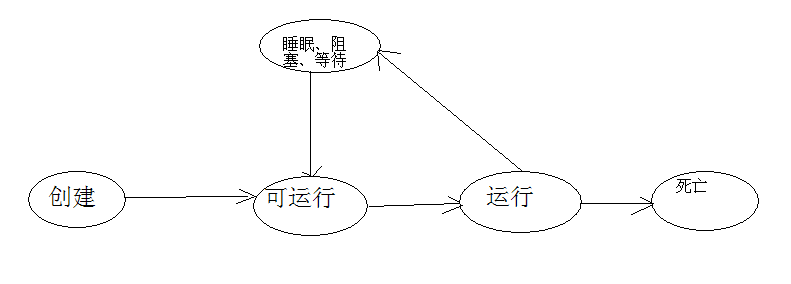


达到了我们预期的效果。

**线程的使用细节：**

1. 线程的启动使用父类的start()方法
2. 如果线程对象直接调用run()，那么JVN不会当作线程来运行，会认为是普通的方法调用。
3. 线程的启动只能由一次，否则抛出异常
4. 可以直接创建Thread类的对象并启动该线程，但是如果没有重写run()，什么也不执行。
5. 匿名内部类的线程实现方式

## 线程的状态



创建：新创建了一个线程对象。

可运行：线程对象创建后，其他线程调用了该对象的start()方法。该状态的线程位于可运行线程池中，变得可运行，等待获取[cpu](http://product.it168.com/list/b/0217_1.shtml" \o "CPU" \t "_blank)的执行权。

运行：就绪状态的线程获取了CPU执行权，执行程序代码。

阻临时塞: 阻塞状态是线程因为某种原因放弃CPU使用权，暂时停止运行。直到线程进入就绪状态，才有机会转到运行状态。

死亡：线程执行完它的任务时。

## 常见线程的方法

Thread(String name) 初始化线程的名字

getName() 返回线程的名字

setName(String name) 设置线程对象名

sleep() 线程睡眠指定的毫秒数。

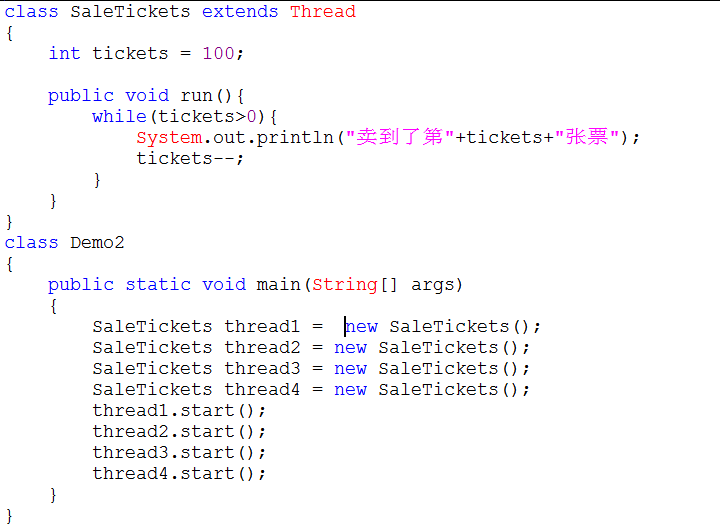
getPriority() 返回当前线程对象的优先级 默认线程的优先级是5

setPriority(int newPriority) 设置线程的优先级 虽然设置了线程的优先级，但是具体的实现取决于底层的操作系统的实现（最大的优先级是10 ，最小的1 ， 默认是5）。

currentThread() 返回CPU正在执行的线程的对象

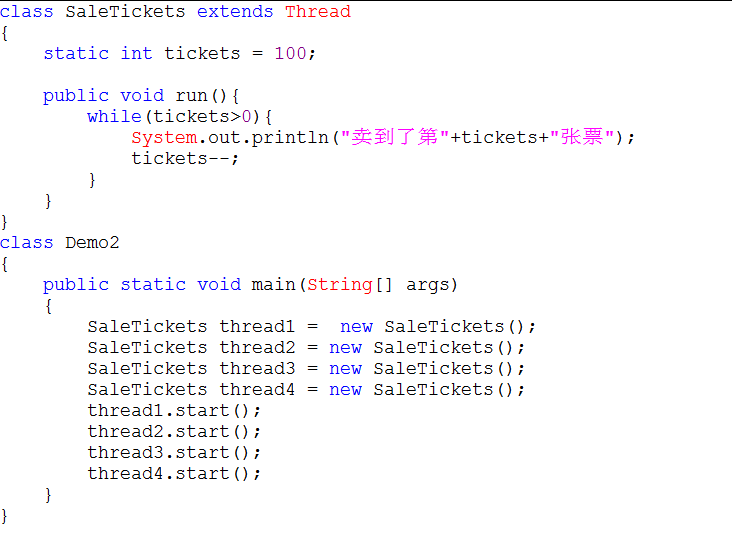
|  |
| --- |
| **class** ThreadDemo1 **extends** Thread  {  **public** ThreadDemo1(){    }  **public** ThreadDemo1( String name ){  **super**( name );  }    **public** **void** run(){  **int** i = 0;  **while**(i < 30){  i++;  System.*out*.println( **this**.getName() + " "+ " : i = " + i);  System.*out*.println( Thread.*currentThread*().getName() + " "+ " : i = " + i);  System.*out*.println( Thread.*currentThread*() == **this** );  System.*out*.println( "getId()" + " "+ " : id = " + **super**.getId() );  System.*out*.println( "getPriority()" + " "+ " : Priority = " + **super**.getPriority() );  }  }  }  **class** Demo3  {  **public** **static** **void** main(String[] args)  {  ThreadDemo1 th1 = **new** ThreadDemo1("线程1");  ThreadDemo1 th2 = **new** ThreadDemo1("线程2");  // 设置线程名  th1.setName( "th1" );  th2.setName( "th2" );  // 设置线程优先级 1 ~ 10  th1.setPriority( 10 );  th2.setPriority( 7 );  // 查看SUN定义的线程优先级范围  System.*out*.println("max : " + Thread.*MAX\_PRIORITY* );  System.*out*.println("min : " + Thread.*MIN\_PRIORITY* );  System.*out*.println("nor : " + Thread.*NORM\_PRIORITY* );  th1.start();  th2.start();  System.*out*.println("Hello World!");  }  } |

练习：模拟卖票



**存在问题：**这时候启动了四个线程，那么tickets是一个成员变量，也就是在一个线程对象中都维护了属于自己的tickets属性，那么就总共存在了四份。

**解决方案一：**tickets使用staitc修饰，使每个线程对象都是共享一份属性。



**解决方案2：**编写一个类实现Runnable接口。

## 创建线程的方式二

创建线程的第二种方式.使用Runnable接口.

该类中的代码就是对线程要执行的任务的定义.

1：定义了实现Runnable接口

2：重写Runnable接口中的run方法，就是将线程运行的代码放入在run方法中

3：通过Thread类建立线程对象

4：将Runnable接口的子类对象作为实际参数，传递给Thread类构造方法

5：调用Thread类的start方法开启线程，并调用Runable接口子类run方法

为什么要将Runnable接口的子类对象传递给Thread的构造函数，因为自定义的run方法所属对象是Runnable接口的子类对象，所以要让线程去执行指定对象的run方法

|  |
| --- |
| **package** cn.itcast.gz.runnable;  **public** **class** Demo1 {  **public** **static** **void** main(String[] args) {  MyRun my = **new** MyRun();  Thread t1 = **new** Thread(my);  t1.start();  **for** (**int** i = 0; i < 200; i++) {  System.*out*.println("main:" + i);  }  }  }  **class** MyRun **implements** Runnable {  **public** **void** run() {  **for** (**int** i = 0; i < 200; i++) {  System.*err*.println("MyRun:" + i);  }  }  } |

理解Runnable:

Thread类可以理解为一个工人,而Runnable的实现类的对象就是这个工人的工作(通过构造方法传递).Runnable接口中只有一个方法run方法,该方法中定义的事会被新线程执行的代码.当我们把Runnable的子类对象传递给Thread的构造时,实际上就是让给Thread取得run方法,就是给了Thread一项任务.

**买票例子使用Runnable接口实现**

在上面的代码中故意照成线程执行完后，执行Thread.sleep（100），以让cpu让给别的线程，该方法会出现非运行时异常需要处理，这里必须进行try{}catch（）{}，因为子类不能比父类抛出更多的异常，接口定义中没有异常，实现类也不能抛出异常。

运行发现票号出现了负数，显示了同一张票被卖了4次的情况。

出现了同样的问题。如何解决？

|  |
| --- |
| **class** MyTicket **implements** Runnable {  **int** tickets = 100;  **public** **void** run() {  **while** (**true**) {  **if** (tickets > 0) {  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.*out*.println(Thread.*currentThread*().getName() + "窗口@销售："  + tickets + "号票");  tickets--;    } **else** {  System.*out*.println("票已卖完。。。");  **break**;  }  }  }  }  **public** **class** Demo6 {  **public** **static** **void** main(String[] args) {  MyTicket mt = **new** MyTicket();  Thread t1 = **new** Thread(mt);  Thread t2 = **new** Thread(mt);  Thread t3 = **new** Thread(mt);  Thread t4 = **new** Thread(mt);  t1.start();  t2.start();  t3.start();  t4.start();  }  } |

# 锁对象

什么是锁对象?

每个java对象都有一个锁对象.而且只有一把钥匙.

如何创建锁对象：

可以使用this关键字作为锁对象，也可以使用所在类的字节码文件对应的Class对象作为锁对象

1. 类名.class

2. 对象.getClass()

Java中的每个对象都有一个内置锁，只有当对象具有同步方法代码时，内置锁才会起作用，当进入一个同步的非静态方法时，就会自动获得与类的当前实例（this）相关的锁，该类的代码就是正在执行的代码。获得一个对象的锁也成为获取锁、锁定对象也可以称之为监视器来指我们正在获取的锁对象。

因为一个对象只有一个锁，所有如果一个线程获得了这个锁，其他线程就不能获得了，直到这个线程释放（或者返回）锁。也就是说在锁释放之前，任何其他线程都不能进入同步代码（不可以进入该对象的任何同步方法）。释放锁指的是持有该锁的线程退出同步方法，此时，其他线程可以进入该对象上的同步方法。

1：只能同步方法（代码块），不能同步变量或者类

2：每个对象只有一个锁

3：不必同步类中的所有方法，类可以同时具有同步方法和非同步方法

4：如果两个线程要执行一个类中的一个同步方法，并且他们使用的是了类的同一个实例（对象）来调用方法，那么一次只有一个线程能够执行该方法，另一个线程需要等待，直到第一个线程完成方法调用，总结就是：一个线程获得了对象的锁，其他线程不可以进入该对象的同步方法。

5：如果类同时具有同步方法和非同步方法，那么多个线程仍然可以访问该类的非同步方法。

同步会影响性能（甚至死锁），优先考虑同步代码块。

6：如果线程进入sleep（） 睡眠状态，该线程会继续持有锁，不会释放。

# 死锁



经典的“哲学家就餐问题”，5个哲学家吃中餐，坐在圆卓子旁。每人有5根筷子（不是5双），每两个人中间放一根，哲学家时而思考，时而进餐。每个人都需要一双筷子才能吃到东西，吃完后将筷子放回原处继续思考，如果每个人都立刻抓住自己左边的筷子，然后等待右边的筷子空出来，同时又不放下已经拿到的筷子，这样每个人都无法得到1双筷子，无法吃饭都会饿死，这种情况就会产生死锁：每个人都拥有其他人需要的资源，同时又等待其他人拥有的资源，并且每个人在获得所有需要的资源之前都不会放弃已经拥有的资源。

当多个线程完成功能需要同时获取多个共享资源的时候可能会导致死锁。

1：两个任务以相反的顺序申请两个锁，死锁就可能出现

2：线程T1获得锁L1，线程T2获得锁L2，然后T1申请获得锁L2，同时T2申请获得锁L1，此时两个线程将要永久阻塞，死锁出现

如果一个类可能发生死锁，那么并不意味着每次都会发生死锁，只是表示有可能。要避免程序中出现死锁。

例如，某个程序需要访问两个文件，当进程中的两个线程分别各锁住了一个文件，那它们都在等待对方解锁另一个文件，而这永远不会发生。

3：要避免死锁

|  |
| --- |
| **public** **class** DeadLock {  **public** **static** **void** main(String[] args) {  **new** Thread(**new** Runnable() { // 创建线程, 代表中国人  **public** **void** run() {  **synchronized** ("刀叉") { // 中国人拿到了刀叉  System.*out*.println(Thread.*currentThread*().getName()  + ": 你不给我筷子, 我就不给你刀叉");  **try** {  Thread.*sleep*(10);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **synchronized** ("筷子") {  System.*out*.println(Thread.*currentThread*()  .getName() + ": 给你刀叉");  }  }  }  }, "中国人").start();  **new** Thread(**new** Runnable() { // 美国人  **public** **void** run() {  **synchronized** ("筷子") { // 美国人拿到了筷子  System.*out*.println(Thread.*currentThread*().getName()  + ": 你先给我刀叉, 我再给你筷子");  **try** {  Thread.*sleep*(10);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **synchronized** ("刀叉") {  System.*out*.println(Thread.*currentThread*()  .getName() + ": 好吧, 把筷子给你.");  }  }  }  }, "美国人").start();  }  } |

# 线程的通讯

线程间通信其实就是多个线程在操作同一个资源，但操作动作不同

生产者消费者

如果有多个生产者和消费者，一定要使用while循环判断标记，然后在使用notifyAll唤醒，否者容易只用notify容易出现只唤醒本方线程情况，导致程序中的所有线程都在等待。

例如：有一个数据存储空间，划分为两个部分，一部分存储人的姓名，一部分存储性别，我们开启一个线程，不停地想其中存储姓名和性别（生产者），开启另一个线程从数据存储空间中取出数据（消费者）。

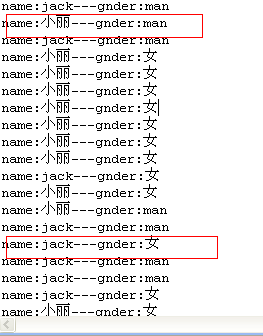
由于是多线程的，就需要考虑，假如生产者刚向数据存储空间中添加了一个人名，还没有来得及添加性别，cpu就切换到了消费者的线程，消费者就会将这个人的姓名和上一个人的性别进行了输出。

还有一种情况是生产者生产了若干次数据，消费者才开始取数据，或者消费者取出数据后，没有等到消费者放入新的数据，消费者又重复的取出自己已经去过的数据。

|  |
| --- |
| **public** **class** Demo10 {  **public** **static** **void** main(String[] args) {  Person p = **new** Person();  Producer pro = **new** Producer(p);  Consumer con = **new** Consumer(p);  Thread t1 = **new** Thread(pro, "生产者");  Thread t2 = **new** Thread(con, "消费者");  t1.start();  t2.start();  }  }  // 使用Person作为数据存储空间  **class** Person {  String name;  String gender;  }  // 生产者  **class** Producer **implements** Runnable {  Person p;  **public** Producer() {  }  **public** Producer(Person p) {  **this**.p = p;  }  @Override  **public** **void** run() {  **int** i = 0;  **while** (**true**) {  **if** (i % 2 == 0) {  p.name = "jack";  p.gender = "man";  } **else** {  p.name = "小丽";  p.gender = "女";  }  i++;  }  }  }  // 消费者  **class** Consumer **implements** Runnable {  Person p;  **public** Consumer() {  }  **public** Consumer(Person p) {  **this**.p = p;  }  @Override  **public** **void** run() {  **while** (**true**) {  System.*out*.println("name:" + p.name + "---gnder:" + p.gender);  }  }  } |

在上述代码中，Producer和Consumer 类的内部都维护了一个Person类型的p成员变量，通过构造函数进行赋值，在man方法中创建了一个Person对象，将其同时传递给Producer和Consumer对象，所以Producer和Consumer访问的是同一个Person对象。并启动了两个线程。

输出：



显然屏幕输出了小丽 man 这样的结果是出现了线程安全问题。所以需要使用synchronized来解决该问题。

|  |
| --- |
| **package** cn.itcast.gz.runnable;  **public** **class** Demo10 {  **public** **static** **void** main(String[] args) {  Person p = **new** Person();  Producer pro = **new** Producer(p);  Consumer con = **new** Consumer(p);  Thread t1 = **new** Thread(pro, "生产者");  Thread t2 = **new** Thread(con, "消费者");  t1.start();  t2.start();  }  }  // 使用Person作为数据存储空间  **class** Person {  String name;  String gender;  }  // 生产者  **class** Producer **implements** Runnable {  Person p;  **public** Producer() {  }  **public** Producer(Person p) {  **this**.p = p;  }  @Override  **public** **void** run() {  **int** i = 0;  **while** (**true**) {  **synchronized** (p) {  **if** (i % 2 == 0) {  p.name = "jack";  p.gender = "man";  } **else** {  p.name = "小丽";  p.gender = "女";  }  i++;  }  }  }  }  // 消费者  **class** Consumer **implements** Runnable {  Person p;  **public** Consumer() {  }  **public** Consumer(Person p) {  **this**.p = p;  }  @Override  **public** **void** run() {  **while** (**true**) {  **synchronized** (p) {  System.*out*.println("name:" + p.name + "---gnder:" + p.gender);  }  }  }  } |

编译运行：屏幕没有再输出jack –女 或者小丽- man 这种情况了。说明我们解决了线程同步问题，但是仔细观察，生产者生产了若干次数据，消费者才开始取数据，或者消费者取出数据后，没有等到消费者放入新的数据，消费者又重复的取出自己已经去过的数据。这个问题依然存在。

升级：在Person类中添加两个方法，set和read方法并设置为synchronized的，让生产者和消费者调用这两个方法。

|  |
| --- |
| **public** **class** Demo10 {  **public** **static** **void** main(String[] args) {  Person p = **new** Person();  Producer pro = **new** Producer(p);  Consumer con = **new** Consumer(p);  Thread t1 = **new** Thread(pro, "生产者");  Thread t2 = **new** Thread(con, "消费者");  t1.start();  t2.start();  }  }  // 使用Person作为数据存储空间  **class** Person {  String name;  String gender;    **public** **synchronized** **void** set(String name, String gender) {  **this**.name = name;  **this**.gender = gender;  }  **public** **synchronized** **void** read() {  System.*out*.println("name:" + **this**.name + "----gender:" + **this**.gender);  }  }  // 生产者  **class** Producer **implements** Runnable {  Person p;  **public** Producer() {  }  **public** Producer(Person p) {  **this**.p = p;  }  @Override  **public** **void** run() {  **int** i = 0;  **while** (**true**) {  **if** (i % 2 == 0) {  p.set("jack", "man");  } **else** {  p.set("小丽", "女");  }  i++;  }  }  }  // 消费者  **class** Consumer **implements** Runnable {  Person p;  **public** Consumer() {  }  **public** Consumer(Person p) {  **this**.p = p;  }  @Override  **public** **void** run() {  **while** (**true**) {  p.read();  }  }  } |

需求：我们需要生产者生产一次，消费者就消费一次。然后这样有序的循环。

这就需要使用线程间的通信了。Java通过Object类的wait，notify，notifyAll这几个方法实现线程间的通信。

### 等待唤醒机制

wait：告诉当前线程放弃执行权，并放弃监视器（锁）并进入阻塞状态，直到其他线程持有获得执行权，并持有了相同的监视器（锁）并调用notify为止。

notify：唤醒持有同一个监视器（锁）中调用wait的第一个线程，例如，餐馆有空位置后，等候就餐最久的顾客最先入座。注意：被唤醒的线程是进入了可运行状态。等待cpu执行权。

notifyAll：唤醒持有同一监视器中调用wait的所有的线程。

如何解决生产者和消费者的问题？

可以通过设置一个标记，表示数据的（存储空间的状态）例如，当消费者读取了（消费了一次）一次数据之后可以将标记改为false，当生产者生产了一个数据，将标记改为true。

，也就是只有标记为true的时候，消费者才能取走数据，标记为false时候生产者才生产数据。

代码实现：

|  |
| --- |
| **package** cn.itcast.gz.runnable;  **public** **class** Demo10 {  **public** **static** **void** main(String[] args) {  Person p = **new** Person();  Producer pro = **new** Producer(p);  Consumer con = **new** Consumer(p);  Thread t1 = **new** Thread(pro, "生产者");  Thread t2 = **new** Thread(con, "消费者");  t1.start();  t2.start();  }  }  // 使用Person作为数据存储空间  **class** Person {  String name;  String gender;  **boolean** flag = **false**;  **public** **synchronized** **void** set(String name, String gender) {  **if** (flag) {  **try** {  wait();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  **this**.name = name;  **this**.gender = gender;  flag = **true**;  notify();  }  **public** **synchronized** **void** read() {  **if** (!flag) {  **try** {  wait();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  System.*out*.println("name:" + **this**.name + "----gender:" + **this**.gender);  flag = **false**;  notify();  }  }  // 生产者  **class** Producer **implements** Runnable {  Person p;  **public** Producer() {  }  **public** Producer(Person p) {  **this**.p = p;  }  @Override  **public** **void** run() {  **int** i = 0;  **while** (**true**) {  **if** (i % 2 == 0) {  p.set("jack", "man");  } **else** {  p.set("小丽", "女");  }  i++;  }  }  }  // 消费者  **class** Consumer **implements** Runnable {  Person p;  **public** Consumer() {  }  **public** Consumer(Person p) {  **this**.p = p;  }  @Override  **public** **void** run() {  **while** (**true**) {  p.read();  }  }  } |

线程间通信其实就是多个线程在操作同一个资源，但操作动作不同，wait，notify（），notifyAll()都使用在同步中，因为要对持有监视器（锁）的线程操作，所以要使用在同步中，因为只有同步才具有锁。

为什么这些方法定义在Object类中

因为这些方法在操作线程时，都必须要标识他们所操作线程持有的锁，只有同一个锁上的被等待线程，可以被统一锁上notify唤醒，不可以对不同锁中的线程进行唤醒，就是等待和唤醒必须是同一个锁。而锁由于可以使任意对象，所以可以被任意对象调用的方法定义在Object类中

wait() 和 sleep()有什么区别？

wait():释放资源，释放锁。是Object的方法

sleep():释放资源，不释放锁。是Thread的方法

定义了notify为什么还要定义notifyAll，因为只用notify容易出现只唤醒本方线程情况，导致程序中的所有线程都在等待。

# 线程生命周期

任何事物都是生命周期，线程也是，

1. 正常终止 当线程的run()执行完毕，线程死亡。

2. 使用标记停止线程

注意：Stop方法已过时，就不能再使用这个方法。

如何使用标记停止线程停止线程。

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，线程就结束。

|  |
| --- |
| **class** StopThread **implements** Runnable {  **public** **boolean** tag = **true**;  @Override  **public** **void** run() {  **int** i = 0;  **while** (tag) {  i++;  System.*out*.println(Thread.*currentThread*().getName() + "i:" + i);  }  }  }  **public** **class** Demo8 {  **public** **static** **void** main(String[] args) {  StopThread st = **new** StopThread();  Thread th = **new** Thread(st, "线程1");  th.start();  **for** (**int** i = 0; i < 100; i++) {  **if** (i == 50) {  System.*out*.println("main i:" + i);  st.tag = **false**;  }  }  }  } |

上述案例中定义了一个计数器i，用来控制main方法（主线程）的循环打印次数，在i到50这段时间内，两个线程交替执行，当计数器变为50，程序将标记改为false，也就是终止了线程1的while循环，run方法结束，线程1也随之结束。注意：当计数器i变为50的，将标记改为false的时候，cpu不一定马上回到线程1，所以线程1并不会马上终止。

# 后台线程

后台线程：就是隐藏起来一直在默默运行的线程，直到进程结束。

实现：

setDaemon(boolean on)

特点：

当所有的非后台线程结束时，程序也就终止了同时还会杀死进程中的所有后台线程，也就是说，只要有非后台线程还在运行，程序就不会终止，执行main方法的主线程就是一个非后台线程。

必须在启动线程之前（调用start方法之前）调用setDaemon（true）方法，才可以把该线程设置为后台线程。

一旦main（）执行完毕，那么程序就会终止，JVM也就退出了。

可以使用isDaemon() 测试该线程是否为后台线程（守护线程）。

该案例：开启了一个qq检测升级的后台线程，通过while真循环进行不停检测，当计数器变为100的时候，表示检测完毕，提示是否更新，线程同时结束。

为了验证，当非后台线程结束时，后台线程是否终止，故意让该后台线程睡眠一会。发现只要main线程执行完毕，后台线程也就随之消亡了。

|  |
| --- |
| **class** QQUpdate **implements** Runnable {  **int** i = 0;  @Override  **public** **void** run() {  **while** (**true**) {  System.*out*.println(Thread.*currentThread*().getName() + " 检测是否有可用更新");  i++;  **try** {  Thread.*sleep*(10);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **if** (i == 100) {  System.*out*.println("有可用更新，是否升级？");  **break**;  }  }  }  }  **public** **class** Demo9 {  **public** **static** **void** main(String[] args) {  QQUpdate qq = **new** QQUpdate();  Thread th = **new** Thread(qq, "qqupdate");  th.setDaemon(**true**);  th.start();  System.*out*.println(th.isDaemon());  System.*out*.println("hello world");  }  } |

Thread的join方法

当A线程执行到了B线程Join方法时A就会等待，等B线程都执行完A才会执行，Join可以用来临时加入线程执行

本案例，启动了一个JoinThread线程，main（主线程）进行for循环，当计数器为50时，让JoinThread，通过join方法，加入到主线程中，发现只有JoinThread线程执行完，主线程才会执行完毕.

可以刻意让JoinThread线程sleep，如果JoinThread没有调用join方法，那么肯定是主线程执行完毕，但是由于JoinThread线程加入到了main线程，必须等JoinThread执行完毕主线程才能继续执行。

|  |
| --- |
| **class** JoinThread **implements** Runnable {  @Override  **public** **void** run() {  **int** i = 0;  **while** (i < 300) {  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.*out*.println(Thread.*currentThread*().getName() + " i:" + i);  i++;  }  }  }  **public** **class** Demo10 {  **public** **static** **void** main(String[] args) **throws** InterruptedException {  JoinThread jt = **new** JoinThread();  Thread th = **new** Thread(jt, "one");  th.start();  **int** i = 0;  **while** (i < 200) {  **if** (i == 100) {  th.join();  }  System.*err*.println(Thread.*currentThread*().getName() + " i:" + i);  i++;  }  }  } |

上述程序用到了Thread类中的join方法，即th.join语句，作用是将th对应的线程合并到嗲用th.join语句的线程中，main方法的线程中计数器到达100之前，main线程和one线程是交替执行的。在main线程中的计数器到达100后，只有one线程执行，也就是one线程此时被加进了mian线程中，one线程不执行完，main线程会一直等待

带参数的join方法是指定合并时间，有纳秒和毫秒级别。

# 集合

## 什么是集合

存储对象的容器，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，存储对象，集合是存储对象最常用的一种方式。

集合的出现就是为了持有对象。集合中可以存储任意类型的对象, 而且长度可变。在程序中有可能无法预先知道需要多少个对象, 那么用数组来装对象的话, 长度不好定义, 而集合解决了这样的问题。

## 集合和数组的区别

数组和集合类都是容器

数组长度是固定的，集合长度是可变的。数组中可以存储基本数据类型，集合只能存储对象数组中存储数据类型是单一的，集合中可以存储任意类型的对象。

集合类的特点

用于存储对象，长度是可变的，可以存储不同类型的对象。

### 数组的缺点

存储类型单一的数据容器，操作复杂(数组一旦声明好不可变)CRUD

## 集合的分类

集合做什么

1：将对象添加到集合

2：从集合中删除对象

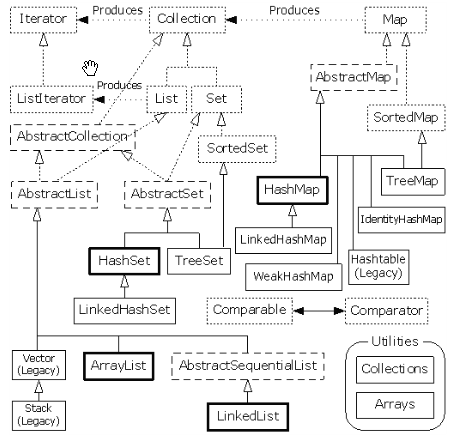
3: 从集合中查找一个对象

4：从集合中修改一个对象就是增删改查

注意：集合和数组中存放的都是对象的引用而非对象本身

Java工程师对不同的容器进行了定义,虽然容器不同,但是还是有一些共性可以抽取最后抽取了一个顶层接口,那么就形成了一个集合框架。如何学习呢?当然是从顶层学起,顶层里边具有最共性,最基本的行为。具体的使用,就要选择具体的容器了。为什么? 因为不断向上抽取的东西有可能是不能创建对象的.抽象的可能性很大，并且子类对象的方法更多一些. 所以是看顶层,创建底层。那么集合的顶层是什么呢 叫做Collection

集合框架体系



|  |
| --- |
| ---|Collection: 单列集合  ---|List: 有存储顺序, 可重复  ---|ArrayList: 数组实现, 查找快, 增删慢  由于是数组实现, 在增和删的时候会牵扯到数组  增容, 以及拷贝元素. 所以慢。数组是可以直接  按索引查找, 所以查找时较快  ---|LinkedList: 链表实现, 增删快, 查找慢  由于链表实现, 增加时只要让前一个元素记住自  己就可以, 删除时让前一个元素记住后一个元  素, 后一个元素记住前一个元素. 这样的增删效  率较高但查询时需要一个一个的遍历, 所以效率  较低  ---|Vector: 和ArrayList原理相同, 但线程安全, 效率略低  和ArrayList实现方式相同, 但考虑了线程安全问  题, 所以效率略低  ---|Set: 无存储顺序, 不可重复  ---|HashSet  ---|TreeSet  ---|LinkedHashSet  ---| Map: 键值对  ---|HashMap  ---|TreeMap  ---|HashTable  ---|LinkedHashMap |

为什么出现这么多集合容器，因为每一个容器对数据的存储方式不同，这种存储方式称之为数据结构（data structure）

注意 集合和数组中存放的都是对象的引用。

## 什么时候该使用什么样的集合

|  |  |
| --- | --- |
| Collection | 我们需要保存若干个对象的时候使用集合。 |
| List | 如果我们需要保留存储顺序, 并且保留重复元素, 使用List.  如果查询较多, 那么使用ArrayList  如果存取较多, 那么使用LinkedList  如果需要线程安全, 那么使用Vector |
| Set | 如果我们不需要保留存储顺序, 并且需要去掉重复元素, 使用Set.  如果我们需要将元素排序, 那么使用TreeSet  如果我们不需要排序, 使用HashSet, HashSet比  TreeSet效率高.  如果我们需要保留存储顺序, 又要过滤重复元素, 那么使用LinkedHashSet |

# 集合类(Collection)

Collection接口有两个子接口：

List(链表|线性表)

Set(集)

特点：

Collection中描述的是集合共有的功能(CRUD)

List可存放重复元素，元素存取是有序的

Set不可以存放重复元素，元素存取是无序的

|  |
| --- |
| java.util.Collection  ---| Collection 描述所有接口的共性  ----| List接口 可以有重复元素的集合  ----| Set 接口 不可以有重复元素的集合 |

2：学习集合对象

学习Collection中的共性方法，多个容器在不断向上抽取就出现了该体系。发现Collection接口中具有所有容器都具备的共性方法。查阅API时，就可以直接看该接口中的方法。并创建其子类对象对集合进行基本应用。当要使用集合对象中特有的方法，在查看子类具体内容。

查看api 文档Collection在在java.util 中（注意是大写Collection）

注意在现阶段遇到的 E T 之类的类型,需要暂时理解为object 因为涉及到了泛型.

3：创建集合对象,使用Collection中的List的具体实现类ArrayList

1：Collection coll=new Arraylist();

## Collection接口的共性方法

|  |
| --- |
| 增加：  1：add() 将指定对象存储到容器中  add 方法的参数类型是Object 便于接收任意对象  2：addAll() 将指定集合中的元素添加到调用该方法和集合中  删除：  3：remove() 将指定的对象从集合中删除  4：removeAll() 将指定集合中的元素删除  修改  5：clear() 清空集合中的所有元素  判断  6：isEmpty() 判断集合是否为空  7：contains() 判断集合何中是否包含指定对象    8：containsAll() 判断集合中是否包含指定集合  使用equals()判断两个对象是否相等  获取: 9：int size() 返回集合容器的大小  转成数组10： toArray() 集合转换数组 |

### 增加：

|  |
| --- |
| **public** **static** **void** main(String[] args) {  Collection list = **new** ArrayList();  // 增加：add() 将指定对象存储到容器中  list.add("计算机网络");  list.add("现代操作系统");  list.add("java编程思想");  System.*out*.println(list);  // [计算机网络, 现代操作系统, java编程思想]    // 增加2 将list容器元素添加到list2容器中  Collection list2 = **new** ArrayList();  list2.add("java核心技术");  list2.addAll(list);  list2.add("java语言程序设计");  System.*out*.println(list2);  // [java核心技术, 计算机网络, 现代操作系统, java编程思想, java语言程序设计]  } |

### 删除：

|  |
| --- |
| // 删除1 remove  **boolean** remove = list2.remove("java核心技术");  System.*out*.println(remove); // true  System.*out*.println(list2); //  //删除2 removeAll() 将list中的元素删除  **boolean** removeAll = list2.removeAll(list);  System.*out*.println(removeAll);//true  System.*out*.println(list2);//[java语言程序设计] |

### 修改：

|  |
| --- |
| **public** **static** **void** main(String[] args) {  Collection list = **new** ArrayList();  // 增加：add() 将指定对象存储到容器中  list.add("计算机网络");  list.add("现代操作系统");  list.add("java编程思想");  list.add("java核心技术");  list.add("java语言程序设计");  System.*out*.println(list);  // 修改 clear() 清空集合中的所有元素  list.clear();  System.*out*.println(list); //[]  } |

### 判断：

|  |
| --- |
| **public** **static** **void** main(String[] args) {  Collection list = **new** ArrayList();  // 增加：add() 将指定对象存储到容器中  list.add("计算机网络");  list.add("现代操作系统");  list.add("java编程思想");  list.add("java核心技术");  list.add("java语言程序设计");  System.*out*.println(list);    **boolean** empty = list.isEmpty();  System.*out*.println(empty);// false  **boolean** contains = list.contains("java编程思想");  System.*out*.println(contains);// true  Collection list2 = **new** ArrayList();  list2.add("水许传");  **boolean** containsAll = list.containsAll(list2);  System.*out*.println(containsAll);// false  } |

### 获取：

|  |
| --- |
| **public** **static** **void** main(String[] args) {  Collection list = **new** ArrayList();  // 增加：add() 将指定对象存储到容器中  list.add("计算机网络");  list.add("现代操作系统");  list.add("java编程思想");  list.add("java核心技术");  list.add("java语言程序设计");  System.*out*.println(list);  // 获取 集合容器的大小  **int** size = list.size();  System.*out*.println(size);  } |

### 练习：集合中添加自定义对象

该案例要求完成使用集合:

|  |
| --- |
| **public** **static** **void** main(String[] args) {  // 创建集合对象  Collection coll = **new** ArrayList();  // 创建Person对象  Person p1 = **new** Person("jack", 25);  Person p2 = **new** Person("rose", 22);  Person p3 = **new** Person("lucy", 20);  Person p4 = **new** Person("jack", 25);  // 集合中添加一些Perosn  // 删除指定Person  // 删除所有Person  // 判断容器中是否还有Person  // 判断容器中是否包含指定Person  // 获取容器中Person的个数  // 将容器变为数组,遍历除所有Person  } |

分析:

1：Person类

1：姓名和年龄

2：重写hashCode和equals方法

1：如果不重写，调用Object类的equals方法，判断内存地址，为false

1：如果是Person类对象，并且姓名和年龄相同就返回true

2：如果不重写，调用父类hashCode方法

1：如果equals方法相同，那么hashCode也要相同，需要重写hashCode方法

3：重写toString方法

1：不重写，直接调用Object类的toString方法，打印该对象的内存地址

Person类

|  |
| --- |
| **class** Person {  **private** String name;  **private** **int** age;  **public** Person() {  }  **public** Person(String name, **int** age) {  **this**.name = name;  **this**.age = age;  }  @Override  **public** **int** hashCode() {  **return** **this**.name.hashCode() + age;  }  @Override  **public** **boolean** equals(Object obj) {  **if** (!(obj **instanceof** Person)) {  **return** **false**;  }  Person p = (Person) obj;  **return** **this**.name.equals(p.name) && **this**.age == p.age;  }  @Override  **public** String toString() {  **return** "Person :name=" + name + ", age=" + age;  }  } |

|  |
| --- |
| **public** **static** **void** main(String[] args) {  Person p1 = **new** Person("张三", 19);  Person p2 = **new** Person("李四", 20);  Person p3 = **new** Person("王五", 18);  Collection list = **new** ArrayList();  list.add(p1);  list.add(p2);  list.add(p3);  // isEmpty() 判断集合是否为空  **boolean** empty = list.isEmpty();  System.*out*.println(empty);  // 返回集合容器的大小  **int** size = list.size();  System.*out*.println(size);  // contains()判断集合何中是否包含指定对象  **boolean** contains = list.contains(p1);  System.*out*.println(contains);  // remove(); 将指定的对象从集合中删除  list.remove(p1);    // clear() 清空集合中的所有元素  list.clear();  System.*out*.println(list);  } |
| **//使用集合存储自定义对象2**  **class** Book {  **private** String name;  **private** **double** price;  **public** Book() {  }  **public** Book(String name, **double** price) {  **this**.name = name;  **this**.price = price;  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** **double** getPrice() {  **return** price;  }  **public** **void** setPrice(**double** price) {  **this**.price = price;  }  @Override  **public** **int** hashCode() {  **return** (**int**) (**this**.name.hashCode() + price);  }  @Override  **public** **boolean** equals(Object obj) {  **if** (!(obj **instanceof** Book)) {  **return** **false**;  }  Book book = (Book) obj;  **return** **this**.name.equals(book.name) && **this**.price == book.price;  }  @Override  **public** String toString() {  **return** "book:@ name:" + **this**.name + ", price:" + **this**.price;  }  }  **public** **class** Demo1 {  **public** **static** **void** main(String[] args) {  Collection col = **new** ArrayList();  col.add(**new** Book("think in java", 100));  col.add(**new** Book("core java", 200)); System.*out*.println(col);  }  } |

## List

|  |
| --- |
| ---| Iterable 接口  Iterator iterator()  ----| Collection 接口  ------| List 接口 元素可以重复，允许在指定位置插入元素，并通过索  引来访问元素 |

### List集合特有方法

|  |
| --- |
| 1：增加  void add(int index, E element) 指定位置添加元素  boolean addAll(int index, Collection c) 指定位置添加集合  2：删除  E remove(int index) 删除指定位置元素  3：修改  E set(int index, E element) 返回的是需要替换的集合中的元素  4：查找：  E get(int index) 注意： IndexOutOfBoundsException  int indexOf(Object o) // 找不到返回-1  lastIndexOf(Object o)  5：求子集合  List<E> subList(int fromIndex, int toIndex) // 不包含toIndex |

#### 增加

|  |
| --- |
| **public** **static** **void** main(String[] args) {  List list = **new** ArrayList();  // 增加：add() 将指定对象存储到容器中  list.add("计算机网络");  list.add("现代操作系统");  list.add("java编程思想");  list.add("java核心技术");  list.add("java语言程序设计");  System.*out*.println(list);  // add,在0角标位置添加一本书  list.add(0, "舒克和贝塔");  System.*out*.println(list);  // 在list2集合的1角标位置添加list集合元素  List list2 = **new** ArrayList();  list2.add("史记");  list2.add("资治通鉴");  list2.add("全球通史");  **boolean** addAll = list2.addAll(1, list);  System.*out*.println(addAll); //true System.*out*.println(list2);  } |

#### 删除

|  |
| --- |
| **public** **static** **void** main(String[] args) {  List list = **new** ArrayList();  // 增加：add() 将指定对象存储到容器中  list.add("计算机网络");  list.add("现代操作系统");  list.add("java编程思想");  list.add("java核心技术");  list.add("java语言程序设计");  System.*out*.println(list);  // 删除0角标元素  Object remove = list.remove(0);  System.*out*.println(remove);  } |

#### 修改：

|  |
| --- |
| **public** **static** **void** main(String[] args) {  List list = **new** ArrayList();  // 增加：add() 将指定对象存储到容器中  list.add("计算机网络");  list.add("现代操作系统");  list.add("java编程思想");  list.add("java核心技术");  list.add("java语言程序设计");  System.*out*.println(list);  // 修改2角标位置的书，返回的原来2角标位置的书  Object set = list.set(2, "边城");  System.*out*.println(set); //java编程思想  System.*out*.println(list);    } |

查找

|  |
| --- |
| List list = **new** ArrayList();  // 增加：add() 将指定对象存储到容器中  list.add("计算机网络");  list.add("现代操作系统");  list.add("java编程思想");  list.add("java核心技术");  list.add("java语言程序设计");  System.*out*.println(list);  // 查找： E get(int index) 注意角标越界  Object set = list.get(list.size() - 1);  System.*out*.println(set); // java语言程序设计  System.*out*.println(list);  list.get(list.size()); //IndexOutOfBoundsException  } |

#### 查找

|  |
| --- |
| **public** **static** **void** main(String[] args) {  List list = **new** ArrayList();  // 增加：add() 将指定对象存储到容器中  list.add("计算机网络");  list.add("现代操作系统");  list.add("java编程思想");  list.add("java核心技术");  list.add("java语言程序设计");  list.add("java编程思想");  System.*out*.println(list);  // 查找： E get(int index) 注意角标越界  Object set = list.get(list.size() - 1);  System.*out*.println(set); // java语言程序设计  System.*out*.println(list);  // list.get(list.size()); //IndexOutOfBoundsException  // indexOf(Object o) 返回第一次出现的指定元素的角标  **int** indexOf = list.indexOf("java编程思想");  System.*out*.println(indexOf); // 2  // 没有找到，返回-1  **int** indexOf2 = list.indexOf("三国志");  System.*out*.println(indexOf2); // -1  // lastIndexOf 返回最后出现的指定元素的角标  **int** lastIndexOf = list.lastIndexOf("java编程思想");  System.*out*.println(lastIndexOf); // 5  } |

### ArrayList

|  |
| --- |
| --| Iterable  ----| Collection  ------| List  ---------| ArrayList 底层采用数组实现，默认10。每次增长  60%,（(oldCapacity \* 3)/2 + 1） 查询快，增删慢。  ---------| LinkedList |

ArrayList:实现原理：

数组实现, 查找快, 增删慢

数组为什么是查询快?因为数组的内存空间地址是连续的.

ArrayList底层维护了一个Object[] 用于存储对象，默认数组的长度是10。可以通过 new ArrayList(20)显式的指定用于存储对象的数组的长度。

当默认的或者指定的容量不够存储对象的时候，容量自动增长为原来的容量的1.5倍。

由于ArrayList是数组实现, 在增和删的时候会牵扯到数组增容, 以及拷贝元素. 所以慢。数组是可以直接按索引查找, 所以查找时较快

可以考虑,假设向数组的0角标未知添加元素,那么原来的角标位置的元素需要整体往后移,并且数组可能还要增容,一旦增容,就需要要将老数组的内容拷贝到新数组中.所以数组的增删的效率是很低的.

练习：去除ArrayList集合中重复元素

1：存入字符串元素

2：存入自定义对象元素（如Perosn对象）

原理：

循环遍历该集合，每取出一个放置在新的集合中，放置之前先判断新的集合是否以包含了新的元素。

|  |
| --- |
| 思路:  存入人的对象.  1先定义person 类  2将该类的实例存入集合  3 将对象元素进行操作. 注意:自定义对象要进行复写toString 和 equals 方法.  为什么? 因为object 是person 的父类,object 中的toString 返回的是哈希值,object 类中equals  方法比较的是对象的地址值.  思路  1存入字符串对象  2存入自定义对象 如person  2创建容器,用于存储非重复元素  3对原容器进行遍历,在遍历过程中进行判断遍历到的元素是否在容器中存在.(contains)  4如果存在,就不存入,否则存入.  5 返回新容器 |

|  |
| --- |
| **public** **class** Demo6 {  **public** **static** **void** main(String[] args) {  ArrayList arr = **new** ArrayList();  Person p1 = **new** Person("jack", 20);  Person p2 = **new** Person("rose", 18);  Person p3 = **new** Person("rose", 18);  arr.add(p1);  arr.add(p2);  arr.add(p3);  System.*out*.println(arr);  ArrayList arr2 = **new** ArrayList();  **for** (**int** i = 0; i < arr.size(); i++) {  Object obj = arr.get(i);  Person p = (Person) obj;  **if** (!(arr2.contains(p))) {  arr2.add(p);  }  }  System.*out*.println(arr2);  }  }  **class** Person {  **private** String name;  **private** **int** age;  **public** Person() {  }  **public** Person(String name, **int** age) {  **this**.name = name;  **this**.age = age;  }  **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;  }  @Override  **public** **int** hashCode() {  **return** **this**.name.hashCode() + age \* 37;  }  @Override  **public** **boolean** equals(Object obj) {  **if** (!(obj **instanceof** Person)) {  **return** **false**;  }  Person p = (Person) obj;  **return** **this**.name.equals(p.name) && **this**.age == p.age;  }  @Override  **public** String toString() {  **return** "Person@name:" + **this**.name + " age:" + **this**.age;  }  } |

在实际的开发中ArrayList是使用频率最高的一个集合。

### LinkedList

|  |
| --- |
| --| Iterable  ----| Collection  ------| List  ---------| ArrayList 底层采用数组实现，默认10。每次增长  60%,（(oldCapacity \* 3)/2 + 1） 查询快，增删慢。  ---------| LinkedList 底层采用链表实现，增删快，查询慢。 |

LinkedList:链表实现, 增删快, 查找慢

由于LinkedList:在内存中的地址不连续,需要让上一个元素记住下一个元素.所以每个元素中保存的有下一个元素的位置.虽然也有角标,但是查找的时候,需要从头往下找,显然是没有数组查找快的.但是,链表在插入新元素的时候,只需要让前一个元素记住新元素,让新元素记住下一个元素就可以了.所以插入很快.

由于链表实现, 增加时只要让前一个元素记住自己就可以, 删除时让前一个元素记住后一个元素, 后一个元素记住前一个元素. 这样的增删效率较高。

但查询时需要一个一个的遍历, 所以效率较低。

特有方法

|  |
| --- |
| 1：方法介绍  addFirst(E e)  addLast(E e)  getFirst()  getLast()  removeFirst()  removeLast()  如果集合中没有元素，获取或者删除元  素抛：NoSuchElementException  2：数据结构  1：栈 （1.6）  先进后出  push()  pop()  2：队列（双端队列1.5）  先进先出  offer()  poll()  3：返回逆序的迭代器对象  descendingIterator() 返回逆序的迭代器对象 |

基本方法

|  |
| --- |
| **import** java.util.Iterator;  **import** java.util.LinkedList;  **public** **class** Demo3 {  **public** **static** **void** main(String[] args) {  LinkedList list = **new** LinkedList();  list.add("西游记");  list.add("三国演义");  list.add("石头记");  list.add("水浒传");  list.add("全球通史");  list.addFirst("史记");  list.addLast("呐喊");  // list.addFirst(null);  // list.addLast(null);  System.*out*.println(list);  // 获取指定位置处的元素。  String str = (String) list.get(0);  // 返回此列表的第一个元素。  String str2 = (String) list.getFirst();  System.*out*.println(str.equals(str2));  // 获取指定位置处的元素。  String str3 = (String) list.get(list.size() - 1);  // 返回此列表的最后一个元素。  String str4 = (String) list.getLast();  System.*out*.println(str3.equals(str4));  // 获取但不移除此列表的头（第一个元素）。  Object element = list.element();  System.*out*.println(element);  **int** size = list.size();  System.*out*.println(size);  } |

迭代

|  |
| --- |
| **mport** java.util.Iterator;  **import** java.util.LinkedList;  **public** **class** Demo3 {  **public** **static** **void** main(String[] args) {  LinkedList list = **new** LinkedList();  list.add("西游记");  list.add("三国演义");  list.add("石头记");  list.add("水浒传");  list.add("全球通史");  Iterator it = list.iterator();  **while** (it.hasNext()) {  String next = (String) it.next();  System.*out*.println(next);  }  }  } |

逆序迭代

|  |
| --- |
| **import** java.util.Iterator;  **import** java.util.LinkedList;  **public** **class** Demo6 {  **public** **static** **void** main(String[] args) {  LinkedList list = **new** LinkedList();  list.add("aa");  list.add("bb");  list.add("cc");  Iterator dit = list.descendingIterator();  **while** (dit.hasNext()) {  System.*out*.println(dit.next());  }  }  } |

注意：可以使用该集合去模拟出队列(先进先出) 或者堆栈(后进先出) 数据结构。

堆栈(后进先出)

|  |
| --- |
| **//**堆栈(后进先出) 数据结构  **public** **class** Demo3 {  **public** **static** **void** main(String[] args) {  LinkedList list = **new** LinkedList();  // 压栈，先进后出  list.push("西游记");  list.push("三国演义");  list.push("石头记");  list.push("水浒传");  System.*out*.println(list);  // 弹栈  String str1 = (String) list.pop();  System.*out*.println(str1);  String str2 = (String) list.pop();  System.*out*.println(str2);  String str3 = (String) list.pop();  System.*out*.println(str3);  String str4 = (String) list.pop();  System.*out*.println(str4);  System.*out*.println(list.size());// 0  System.*out*.println(list); //[]  }  } |

队列，先进先出

|  |
| --- |
| **import** java.util.LinkedList;  **public** **class** Demo3 {  **public** **static** **void** main(String[] args) {  LinkedList list = **new** LinkedList();  // 队列，先进先出  list.offer("西游记");  list.offer("三国演义");  list.offer("石头记");  list.offer("水浒传");  System.*out*.println(list);  // 出队列  System.*out*.println(list.poll());  System.*out*.println(list.poll());  System.*out*.println(list.poll());  System.*out*.println(list.poll());  System.*out*.println(list.size());  System.*out*.println(list.peek()); // 获取队列的头元素，但是不删除  System.*out*.println(list.peekFirst()); // 获取队列的头元素，但是不删除  System.*out*.println(list.peekLast()); // 获取队列的最后一个元素但是不删除  }  } |

ArrayList 和 LinkedList的存储查找的优缺点：

1、ArrayList 是采用动态数组来存储元素的，它允许直接用下标号来直接查找对应的元素。但是，但是插入元素要涉及数组元素移动及内存的操作。总结：查找速度快，插入操作慢。

2、LinkedList 是采用双向链表实现存储，按序号索引数据需要进行前向或后向遍历，但是插入数据时只需要记录本项的前后项即可，所以插入速度较快

问题：有一批数据要存储，要求存储这批数据不能出现重复数据，ArrayList、LinkedList都没法满足需求。解决办法：使用 set集合。

### Vector

Vector: 描述的是一个线程安全的ArrayList。

ArrayList： 单线程效率高

Vector ： 多线程安全的，所以效率低

|  |
| --- |
| 特有的方法：  void addElement(E obj) 在集合末尾添加元素  E elementAt( int index) 返回指定角标的元素  Enumeration elements() 返回集合中的所有元素，封装到Enumeration对象中  Enumeration 接口：  boolean hasMoreElements()  测试此枚举是否包含更多的元素。  E nextElement()  如果此枚举对象至少还有一个可提供的元素，则返回此枚举的下一个元素。 |

|  |
| --- |
| **public** **static** **void** main(String[] args)  {  Vector v = **new** Vector();  v.addElement("aaa");  v.addElement("bbb");  v.addElement("ccc");  System.*out*.println( v );  System.*out*.println( v.elementAt(2) ); // ccc  // 遍历Vector遍历  Enumeration ens = v.elements();  **while** ( ens.hasMoreElements() )  {  System.*out*.println( ens.nextElement() );  }  } |

## 迭代器

为了方便的处理集合中的元素,Java中出现了一个对象,该对象提供了一些方法专门处理集合中的元素.例如删除和获取集合中的元素.该对象就叫做迭代器(Iterator).

对 Collection 进行迭代的类，称其为迭代器。还是面向对象的思想，专业对象做专业的事情，迭代器就是专门取出集合元素的对象。但是该对象比较特殊，不能直接创建对象（通过new），该对象是以内部类的形式存在于每个集合类的内部。

如何获取迭代器？Collection接口中定义了获取集合类迭代器的方法（iterator（）），所以所有的Collection体系集合都可以获取自身的迭代器。

正是由于每一个容器都有取出元素的功能。这些功能定义都一样，只不过实现的具体方式不同（因为每一个容器的数据结构不一样）所以对共性的取出功能进行了抽取，从而出现了Iterator接口。而每一个容器都在其内部对该接口进行了内部类的实现。也就是将取出方式的细节进行封装。

### Iterable

Jdk1.5之后添加的新接口, Collection的父接口. 实现了Iterable的类就是可迭代的.并且支持增强for循环。该接口只有一个方法即获取迭代器的方法iterator（）可以获取每个容器自身的迭代器Iterator。（Collection）集合容器都需要获取迭代器（Iterator）于是在5.0后又进行了抽取将获取容器迭代器的iterator（）方法放入到了Iterable接口中。Collection接口进程了Iterable，所以Collection体系都具备获取自身迭代器的方法，只不过每个子类集合都进行了重写（因为数据结构不同）

### Iterator

Iterator iterator() 返回该集合的迭代器对象

|  |
| --- |
| 该类主要用于遍历集合对象，该类描述了遍历集合的常见方法  1：java.lang. Itreable  ---| Itreable 接口 实现该接口可以使用增强for循环  ---| Collection 描述所有集合共性的接口  ---| List接口 可以有重复元素的集合  ---| Set接口 不可以有重复元素的集合 |

public interface Iterable<T>

Itreable 该接口仅有一个方法，用于返回集合迭代器对象。

1： Iterator<T> iterator() 返回集合的迭代器对象

Iterator接口定义的方法

|  |
| --- |
| Itreator 该接口是集合的迭代器接口类，定义了常见的迭代方法  1：boolean hasNext()  判断集合中是否有元素，如果有元素可以迭代，就返回true。  2： E next()  返回迭代的下一个元素，注意： 如果没有下一个元素时，调用  next元素会抛出NoSuchElementException  3： void remove()  从迭代器指向的集合中移除迭代器返回的最后一个元素（可选操  作）。 |

思考：为什么next方法的返回类型是Object的呢？ 为了可以接收任意类型的对象,那么返回的时候,不知道是什么类型的就定义为object

### 迭代器遍历

1：while循环

|  |
| --- |
| **public** **static** **void** main(String[] args) {  ArrayList list = **new** ArrayList();  // 增加：add() 将指定对象存储到容器中  list.add("计算机网络");  list.add("现代操作系统");  list.add("java编程思想");  list.add("java核心技术");  list.add("java语言程序设计");  System.*out*.println(list);  Iterator it = list.iterator();  **while** (it.hasNext()) {  String next = (String) it.next();  System.*out*.println(next);  }  } |

2：for循环

|  |
| --- |
| **import** java.util.ArrayList;  **import** java.util.Iterator;  **public** **class** Demo2 {  **public** **static** **void** main(String[] args) {  ArrayList list = **new** ArrayList();  // 增加：add() 将指定对象存储到容器中  list.add("计算机网络");  list.add("现代操作系统");  list.add("java编程思想");  list.add("java核心技术");  list.add("java语言程序设计");  System.*out*.println(list);  **for** (Iterator it = list.iterator(); it.hasNext();) {  //迭代器的next方法返回值类型是Object，所以要记得类型转换。  String next = (String) it.next();  System.*out*.println(next);  }  }  } |

需要取出所有元素时，可以通过循环，java 建议使用for 循环。因为可以对内存进行一下优化。

3：使用迭代器清空集合

|  |
| --- |
| **public** **class** Demo1 {  **public** **static** **void** main(String[] args) {  Collection coll = **new** ArrayList();  coll.add("aaa");  coll.add("bbb");  coll.add("ccc");  coll.add("ddd");  System.*out*.println(coll);  Iterator it = coll.iterator();  **while** (it.hasNext()) {  it.next();  it.remove();  }  System.*out*.println(coll);  }  } |

细节一：

如果迭代器的指针已经指向了集合的末尾，那么如果再调用next()会返回NoSuchElementException异常

|  |
| --- |
| **import** java.util.ArrayList;  **import** java.util.Iterator;  **public** **class** Demo2 {  **public** **static** **void** main(String[] args) {  ArrayList list = **new** ArrayList();  // 增加：add() 将指定对象存储到容器中  list.add("计算机网络");  list.add("现代操作系统");  list.add("java编程思想");  list.add("java核心技术");  list.add("java语言程序设计");  System.*out*.println(list);  Iterator it = list.iterator();  **while** (it.hasNext()) {  String next = (String) it.next();  System.*out*.println(next);  }  // 迭代器的指针已经指向了集合的末尾  // String next = (String) it.next();  // java.util.NoSuchElementException  }  } |

细节二：

如果调用remove之前没有调用next是不合法的，会抛出IllegalStateException

|  |
| --- |
| **import** java.util.ArrayList;  **import** java.util.Iterator;  **public** **class** Demo2 {  **public** **static** **void** main(String[] args) {  ArrayList list = **new** ArrayList();  // 增加：add() 将指定对象存储到容器中  list.add("计算机网络");  list.add("现代操作系统");  list.add("java编程思想");  list.add("java核心技术");  list.add("java语言程序设计");  System.*out*.println(list);  Iterator it = list.iterator();  **while** (it.hasNext()) {  // 调用remove之前没有调用next是不合法的  // it.remove();  // java.lang.IllegalStateException  String next = (String) it.next();  System.*out*.println(next);  }  }  } |

4：迭代器原理

查看ArrayList源码

|  |
| --- |
| **private** **class** Itr **implements** Iterator<E> {  **int** cursor = 0;  **int** lastRet = -1;  **int** expectedModCount = modCount;  **public** **boolean** hasNext() {  **return** cursor != size();  }  **public** E next() {  checkForComodification();  **try** {  E next = get(cursor);  lastRet = cursor++;  **return** next;  } **catch** (IndexOutOfBoundsException e) {  checkForComodification();  **throw** **new** NoSuchElementException();  }  }  **public** **void** remove() {  **if** (lastRet == -1)  **throw** **new** IllegalStateException();  checkForComodification();  **try** {  AbstractList.**this**.remove(lastRet);  **if** (lastRet < cursor)  cursor--;  lastRet = -1;  expectedModCount = modCount;  } **catch** (IndexOutOfBoundsException e) {  **throw** **new** ConcurrentModificationException();  }  }    } |

5：注意在对集合进行迭代过程中，不允许出现迭代器以外的对元素的操作，因为这样会产生安全隐患，java会抛出异常并发修改异常（ConcurrentModificationException），普通迭代器只支持在迭代过程中的删除动作。

注意： ConcurrentModificationException: 当一个集合在循环中即使用引用变量操作集合又使用迭代器操作集合对象， 会抛出该异常。

|  |
| --- |
| **import** java.util.ArrayList;  **import** java.util.Collection;  **import** java.util.Iterator;  **public** **class** Demo1 {  **public** **static** **void** main(String[] args) {  Collection coll = **new** ArrayList();  coll.add("aaa");  coll.add("bbb");  coll.add("ccc");  coll.add("ddd");  System.*out*.println(coll);  Iterator it = coll.iterator();  **while** (it.hasNext()) {  it.next();  it.remove();  coll.add("abc"); // 出现了迭代器以外的对元素的操作  }  System.*out*.println(coll);  }  } |

如果是List集合，想要在迭代中操作元素可以使用List集合的特有迭代器ListIterator，该迭代器支持在迭代过程中，添加元素和修改元素。

### List特有的迭代器ListIterator

public interface ListIterator extends Iterator

ListIterator<E> listIterator()

|  |
| --- |
| ---| Iterator  hasNext()  next()  remove()  ------| ListIterator Iterator子接口 List专属的迭代器  add(E e) 将指定的元素插入列表（可选操作）。该元素直接插入到 next 返回的下一个元素的前面（如果有）  void set(E o) 用指定元素替换 next 或 previous 返回的最后一个元素  hasPrevious() 逆向遍历列表，列表迭代器有多个元素，则返回 true。  previous() 返回列表中的前一个元素。 |

Iterator在迭代时，只能对元素进行获取(next())和删除(remove())的操作。

对于 Iterator 的子接口ListIterator 在迭代list 集合时，还可以对元素进行添加

(add(obj))，修改set(obj)的操作。

|  |
| --- |
| **import** java.util.ArrayList;  **import** java.util.ListIterator;  **public** **class** Demo2 {  **public** **static** **void** main(String[] args) {  ArrayList list = **new** ArrayList();  // 增加：add() 将指定对象存储到容器中  list.add("计算机网络");  list.add("现代操作系统");  list.add("java编程思想");  list.add("java核心技术");  list.add("java语言程序设计");  System.*out*.println(list);  // 获取List专属的迭代器  ListIterator lit = list.listIterator();  **while** (lit.hasNext()) {  String next = (String) lit.next();  System.*out*.println(next);  }  }  } |

倒序遍历

|  |
| --- |
| **import** java.util.ArrayList;  **import** java.util.ListIterator;  **public** **class** Demo2 {  **public** **static** **void** main(String[] args) {  ArrayList list = **new** ArrayList();  // 增加：add() 将指定对象存储到容器中  list.add("计算机网络");  list.add("现代操作系统");  list.add("java编程思想");  list.add("java核心技术");  list.add("java语言程序设计");  System.*out*.println(list);  // 获取List专属的迭代器  ListIterator lit = list.listIterator();  **while** (lit.hasNext()) {  String next = (String) lit.next();  System.*out*.println(next);  }  System.*out*.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  **while** (lit.hasPrevious()) {  String next = (String) lit.previous();  System.*out*.println(next);  }  }  } |

Set方法：用指定元素替换 next 或 previous 返回的最后一个元素

|  |
| --- |
| **import** java.util.ArrayList;  **import** java.util.ListIterator;  **public** **class** Demo2 {  **public** **static** **void** main(String[] args) {  ArrayList list = **new** ArrayList();  // 增加：add() 将指定对象存储到容器中  list.add("计算机网络");  list.add("现代操作系统");  list.add("java编程思想");  list.add("java核心技术");  list.add("java语言程序设计");  System.*out*.println(list);  ListIterator lit = list.listIterator();  lit.next(); // 计算机网络  lit.next(); // 现代操作系统  System.*out*.println(lit.next()); // java编程思想  //用指定元素替换 next 或 previous 返回的最后一个元素  lit.set("平凡的世界");// 将java编程思想替换为平凡的世界  System.*out*.println(list);  }  } |

add方法将指定的元素插入列表，该元素直接插入到 next 返回的元素的后

|  |
| --- |
| **public** **class** Demo2 {  **public** **static** **void** main(String[] args) {  ArrayList list = **new** ArrayList();  // 增加：add() 将指定对象存储到容器中  list.add("计算机网络");  list.add("现代操作系统");  list.add("java编程思想");  list.add("java核心技术");  list.add("java语言程序设计");  System.*out*.println(list);  ListIterator lit = list.listIterator();  lit.next(); // 计算机网络  lit.next(); // 现代操作系统  System.*out*.println(lit.next()); // java编程思想  // 将指定的元素插入列表，该元素直接插入到 next 返回的元素的后  lit.add("平凡的世界");// 在java编程思想后添加平凡的世界  System.*out*.println(list);  }  } |

## Set

Set:注重独一无二的性质,该体系集合可以知道某物是否已近存在于集合中,不会存储重复的元素

用于存储无序(存入和取出的顺序不一定相同)元素，值不能重复。

对象的相等性

引用到堆上同一个对象的两个引用是相等的。如果对两个引用调用hashCode方法，会得到相同的结果，如果对象所属的类没有覆盖Object的hashCode方法的话，hashCode会返回每个对象特有的序号（java是依据对象的内存地址计算出的此序号），所以两个不同的对象的hashCode值是不可能相等的。

如果想要让两个不同的Person对象视为相等的，就必须覆盖Object继下来的hashCode方法和equals方法，因为Object hashCode方法返回的是该对象的内存地址，所以必须重写hashCode方法，才能保证两个不同的对象具有相同的hashCode，同时也需要两个不同对象比较equals方法会返回true

该集合中没有特有的方法，直接继承自Collection。

|  |
| --- |
| ---| Itreable 接口 实现该接口可以使用增强for循环  ---| Collection 描述所有集合共性的接口  ---| List接口 可以有重复元素的集合  ---| ArrayList  ---| LinkedList  ---| Set接口 不可以有重复元素的集合 |

案例：set集合添加元素并使用迭代器迭代元素。

|  |
| --- |
| **import** java.util.HashSet;  **import** java.util.Iterator;  **import** java.util.Set;  **public** **class** Demo4 {  **public** **static** **void** main(String[] args) {  //Set 集合存和取的顺序不一致。  Set hs = **new** HashSet();  hs.add("世界军事");  hs.add("兵器知识");  hs.add("舰船知识");  hs.add("汉和防务");  System.*out*.println(hs);  // [舰船知识, 世界军事, 兵器知识, 汉和防务]  Iterator it = hs.iterator();  **while** (it.hasNext()) {  System.*out*.println(it.next());  }  }  } |

### HashSet

|  |
| --- |
| ---| Itreable 接口 实现该接口可以使用增强for循环  ---| Collection 描述所有集合共性的接口  ---| List接口 可以有重复元素的集合  ---| ArrayList  ---| LinkedList  ---| Set接口 不可以有重复元素的集合  ---| HashSet 线程不安全，存取速度快。底层是以哈希表实现的。 |

HashSet

哈希表边存放的是哈希值。HashSet存储元素的顺序并不是按照存入时的顺序（和List显然不同） 是按照哈希值来存的所以取数据也是按照哈希值取得。

HashSet不存入重复元素的规则.使用hashcode和equals

由于Set集合是不能存入重复元素的集合。那么HashSet也是具备这一特性的。HashSet如何检查重复？HashSet会通过元素的hashcode（）和equals方法进行判断元素师否重复。

当你试图把对象加入HashSet时，HashSet会使用对象的hashCode来判断对象加入的位置。同时也会与其他已经加入的对象的hashCode进行比较，如果没有相等的hashCode，HashSet就会假设对象没有重复出现。

简单一句话，如果对象的hashCode值是不同的，那么HashSet会认为对象是不可能相等的。

因此我们自定义类的时候需要重写hashCode，来确保对象具有相同的hashCode值。

如果元素(对象)的hashCode值相同,是不是就无法存入HashSet中了? 当然不是,会继续使用equals 进行比较.如果 equals为true 那么HashSet认为新加入的对象重复了,所以加入失败。如果equals 为false那么HashSet 认为新加入的对象没有重复.新元素可以存入.

总结：

元素的哈希值是通过元素的hashcode方法 来获取的, HashSet首先判断两个元素的哈希值，如果哈希值一样，接着会比较equals方法 如果 equls结果为true ，HashSet就视为同一个元素。如果equals 为false就不是同一个元素。

哈希值相同equals为false的元素是怎么存储呢,就是在同样的哈希值下顺延（可以认为哈希值相同的元素放在一个哈希桶中）。也就是哈希一样的存一列。

hashtable

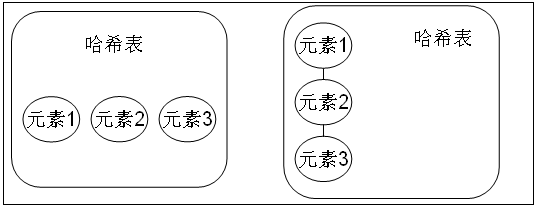


图1：hashCode值不相同的情况

图2：hashCode值相同，但equals不相同的情况。

HashSet：通过hashCode值来确定元素在内存中的位置。一个hashCode位置上可以存放多个元素。

当hashcode() 值相同equals() 返回为true 时,hashset 集合认为这两个元素是相同的元素.只存储一个（重复元素无法放入）。调用原理:先判断hashcode 方法的值,如果相同才会去判断equals 如果不相同,是不会调用equals方法的。

HashSet到底是如何判断两个元素重复。

通过hashCode方法和equals方法来保证元素的唯一性，add()返回的是boolean类型

判断两个元素是否相同，先要判断元素的hashCode值是否一致，只有在该值一致的情况下，才会判断equals方法，如果存储在HashSet中的两个对象hashCode方法的值相同equals方法返回的结果是true，那么HashSet认为这两个元素是相同元素，只存储一个（重复元素无法存入）。

注意：HashSet集合在判断元素是否相同先判断hashCode方法，如果相同才会判断equals。如果不相同，是不会调用equals方法的。

HashSet 和ArrayList集合都有判断元素是否相同的方法，

boolean contains(Object o)

HashSet使用hashCode和equals方法，ArrayList使用了equals方法

练习：使用HashSet存储字符串，并尝试添加重复字符串

回顾String类的equals()、hashCode()两个方法。

|  |
| --- |
| **import** java.util.HashSet;  **import** java.util.Iterator;  **import** java.util.Set;    **public** **class** Demo4 {  **public** **static** **void** main(String[] args) {  // Set 集合存和取的顺序不一致。  Set hs = **new** HashSet();  hs.add("世界军事");  hs.add("兵器知识");  hs.add("舰船知识");  hs.add("汉和防务");  // 返回此 set 中的元素的数量  System.*out*.println(hs.size()); // 4  // 如果此 set 尚未包含指定元素，则返回 true  **boolean** add = hs.add("世界军事"); // false  System.*out*.println(add);  // 返回此 set 中的元素的数量  System.*out*.println(hs.size());// 4  Iterator it = hs.iterator();  **while** (it.hasNext()) {  System.*out*.println(it.next());  }  }  } |

使用HashSet存储自定义对象，并尝试添加重复对象（对象的重复的判定）

|  |
| --- |
| **package** cn.itcast.gz.map;  **import** java.util.HashSet;  **import** java.util.Iterator;  **public** **class** Demo4 {  **public** **static** **void** main(String[] args) {  HashSet hs = **new** HashSet();  hs.add(**new** Person("jack", 20));  hs.add(**new** Person("rose", 20));  hs.add(**new** Person("hmm", 20));  hs.add(**new** Person("lilei", 20));  hs.add(**new** Person("jack", 20));  Iterator it = hs.iterator();  **while** (it.hasNext()) {  Object next = it.next();  System.*out*.println(next);  }  }  }  **class** Person {  **private** String name;  **private** **int** age;  Person() {  }  **public** Person(String name, **int** age) {  **this**.name = name;  **this**.age = age;  }  **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;  }  @Override  **public** **int** hashCode() {  System.*out*.println("hashCode:" + **this**.name);  **return** **this**.name.hashCode() + age \* 37;  }  @Override  **public** **boolean** equals(Object obj) {  System.*out*.println(**this** + "---equals---" + obj);  **if** (obj **instanceof** Person) {  Person p = (Person) obj;  **return** **this**.name.equals(p.name) && **this**.age == p.age;  } **else** {  **return** **false**;  }  }  @Override  **public** String toString() {  **return** "Person@name:" + **this**.name + " age:" + **this**.age;  }  } |

问题：现在有一批数据，要求不能重复存储元素，而且要排序。ArrayList 、 LinkedList不能去除重复数据。HashSet可以去除重复，但是是无序。

### TreeSet

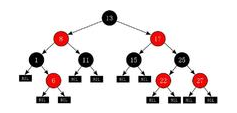
案例：使用TreeSet集合存储字符串元素，并遍历

|  |
| --- |
| **import** java.util.TreeSet;  **public** **class** Demo5 {  **public** **static** **void** main(String[] args) {  TreeSet ts = **new** TreeSet();  ts.add("ccc");  ts.add("aaa");  ts.add("ddd");  ts.add("bbb");  System.*out*.println(ts); // [aaa, bbb, ccc, ddd]  }  } |

|  |
| --- |
| ---| Itreable 接口 实现该接口可以使用增强for循环  ---| Collection 描述所有集合共性的接口  ---| List接口 有序，可以重复，有角标的集合  ---| ArrayList  ---| LinkedList  ---| Set接口 无序，不可以重复的集合  ---| HashSet 线程不安全，存取速度快。底层是以hash表实现的。  ---| TreeSet 红-黑树的数据结构，默认对元素进行自然排序（String）。如果在比较的时候两个对象返回值为0，那么元素重复。 |

红-黑树

红黑树是一种特定类型的二叉树



红黑树算法的规则: 左小右大。

既然TreeSet可以自然排序,那么TreeSet必定是有排序规则的。

1:让存入的元素自定义比较规则。

2:给TreeSet指定排序规则。

方式一：元素自身具备比较性

元素自身具备比较性，需要元素实现Comparable接口，重写compareTo方法，也就是让元素自身具备比较性，这种方式叫做元素的自然排序也叫做默认排序。

方式二：容器具备比较性

当元素自身不具备比较性，或者自身具备的比较性不是所需要的。那么此时可以让容器自身具备。需要定义一个类实现接口Comparator，重写compare方法，并将该接口的子类实例对象作为参数传递给TreeMap集合的构造方法。

注意：当Comparable比较方式和Comparator比较方式同时存在时，以Comparator的比较方式为主；

注意：在重写compareTo或者compare方法时，必须要明确比较的主要条件相等时要比较次要条件。（假设姓名和年龄一直的人为相同的人，如果想要对人按照年龄的大小来排序，如果年龄相同的人，需要如何处理？不能直接return 0，因为可能姓名不同（年龄相同姓名不同的人是不同的人）。此时就需要进行次要条件判断（需要判断姓名），只有姓名和年龄同时相等的才可以返回0.）

通过return 0来判断唯一性。

问题:为什么使用TreeSet存入字符串,字符串默认输出是按升序排列的?因为字符串实现了一个接口,叫做Comparable 接口.字符串重写了该接口的compareTo 方法,所以String对象具备了比较性.那么同样道理,我的自定义元素(例如Person类,Book类)想要存入TreeSet集合,就需要实现该接口,也就是要让自定义对象具备比较性.

存入TreeSet集合中的元素要具备比较性.

比较性要实现Comparable接口，重写该接口的compareTo方法

TreeSet属于Set集合，该集合的元素是不能重复的，TreeSet如何保证元素的唯一性

通过compareTo或者compare方法中的来保证元素的唯一性。

添加的元素必须要实现Comparable接口。当compareTo()函数返回值为0时，说明两个对象相等，此时该对象不会添加进来。

比较器接口

|  |
| --- |
| ----| Comparable  compareTo(Object o) 元素自身具备比较性  ----| Comparator  compare( Object o1, Object o2 ) 给容器传入比较器 |

TreeSet集合排序的两种方式：

一，让元素自身具备比较性。

也就是元素需要实现Comparable接口，覆盖compareTo 方法。

这种方式也作为元素的自然排序，也可称为默认排序。

年龄按照搜要条件，年龄相同再比姓名。

|  |
| --- |
| **import** java.util.TreeSet;  **public** **class** Demo4 {  **public** **static** **void** main(String[] args) {  TreeSet ts = **new** TreeSet();  ts.add(**new** Person("aa", 20, "男"));  ts.add(**new** Person("bb", 18, "女"));  ts.add(**new** Person("cc", 17, "男"));  ts.add(**new** Person("dd", 17, "女"));  ts.add(**new** Person("dd", 15, "女"));  ts.add(**new** Person("dd", 15, "女"));  System.*out*.println(ts);  System.*out*.println(ts.size()); // 5  }  }  **class** Person **implements** Comparable {  **private** String name;  **private** **int** age;  **private** String gender;  **public** Person() {  }  **public** Person(String name, **int** age, String gender) {  **this**.name = name;  **this**.age = age;  **this**.gender = gender;  }  **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;  }  **public** String getGender() {  **return** gender;  }  **public** **void** setGender(String gender) {  **this**.gender = gender;  }  @Override  **public** **int** hashCode() {  **return** name.hashCode() + age \* 37;  }  **public** **boolean** equals(Object obj) {  System.*err*.println(**this** + "equals :" + obj);  **if** (!(obj **instanceof** Person)) {  **return** **false**;  }  Person p = (Person) obj;  **return** **this**.name.equals(p.name) && **this**.age == p.age;  }  **public** String toString() {  **return** "Person [name=" + name + ", age=" + age + ", gender=" + gender  + "]";  }  @Override  **public** **int** compareTo(Object obj) {    Person p = (Person) obj;  System.*out*.println(**this**+" compareTo:"+p);  **if** (**this**.age > p.age) {  **return** 1;  }  **if** (**this**.age < p.age) {  **return** -1;  }  **return** **this**.name.compareTo(p.name);  }  } |

二，让容器自身具备比较性，自定义比较器。

需求：当元素自身不具备比较性，或者元素自身具备的比较性不是所需的。

那么这时只能让容器自身具备。

定义一个类实现Comparator 接口，覆盖compare方法。

并将该接口的子类对象作为参数传递给TreeSet集合的构造函数。

当Comparable比较方式，及Comparator比较方式同时存在，以Comparator

比较方式为主。

|  |
| --- |
| **import** java.util.Comparator;  **import** java.util.TreeSet;  **public** **class** Demo5 {  **public** **static** **void** main(String[] args) {  TreeSet ts = **new** TreeSet(**new** MyComparator());  ts.add(**new** Book("think in java", 100));  ts.add(**new** Book("java 核心技术", 75));  ts.add(**new** Book("现代操作系统", 50));  ts.add(**new** Book("java就业教程", 35));  ts.add(**new** Book("think in java", 100));  ts.add(**new** Book("ccc in java", 100));  System.*out*.println(ts);  }  }  **class** MyComparator **implements** Comparator {  **public** **int** compare(Object o1, Object o2) {  Book b1 = (Book) o1;  Book b2 = (Book) o2;  System.*out*.println(b1+" comparator "+b2);  **if** (b1.getPrice() > b2.getPrice()) {  **return** 1;  }  **if** (b1.getPrice() < b2.getPrice()) {  **return** -1;  }  **return** b1.getName().compareTo(b2.getName());  }  }  **class** Book {  **private** String name;  **private** **double** price;  **public** Book() {  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** **double** getPrice() {  **return** price;  }  **public** **void** setPrice(**double** price) {  **this**.price = price;  }  **public** Book(String name, **double** price) {  **this**.name = name;  **this**.price = price;  }  @Override  **public** String toString() {  **return** "Book [name=" + name + ", price=" + price + "]";  }  } |

TreeSet练习

将字符串中的数值进行排序。

例如String str="8 10 15 5 2 7"; 2,5,7,8,10,15

使用 TreeSet完成。

思路:1，将字符串切割。

2，可以将这些对象存入TreeSet集合。

因为TreeSet自身具备排序功能。

|  |
| --- |
| **public** **class** Demo5 {  **public** **static** **void** main(String[] args) {  String str = "8 10 15 5 2 7";  String[] strs = str.split(" ");  TreeSet ts = **new** TreeSet();  **for** (**int** x = 0; x < strs.length; x++) {  **int** y = Integer.*parseInt*(strs[x]);  ts.add(y);  }  System.*out*.println(ts);  }  } |

### LinkedHashSet

会保存插入的顺序。

看到array，就要想到角标。

看到link，就要想到first，last。

看到hash，就要想到hashCode,equals.

看到tree，就要想到两个接口。Comparable，Comparator。

练习:去除数组中的重复元素.

# Map

如果程序中存储了几百万个学生，而且经常需要使用学号来搜索某个学生，那么这个需求有效的数据结构就是Map。Map是一种依照键（key）存储元素的容器，键（key）很像下标，在List中下标是整数。在Map中键（key）可以使任意类型的对象。Map中不能有重复的键（Key），每个键（key）都有一个对应的值（value）。一个键（key）和它对应的值构成map集合中的一个元素。

Map中的元素是两个对象，一个对象作为键，一个对象作为值。键不可以重复，但是值可以重复。

看顶层共性方法找子类特有对象.

Map与Collection在集合框架中属并列存在

Map存储的是键值对

Map存储元素使用put方法，Collection使用add方法

Map集合没有直接取出元素的方法，而是先转成Set集合，在通过迭代获取元素

Map集合中键要保证唯一性

也就是Collection是单列集合, Map 是双列集合。

总结:

Map一次存一对元素, Collection 一次存一个。Map 的键不能重复，保证唯一。

Map 一次存入一对元素,是以键值对的形式存在.键与值存在映射关系.一定要保证键的唯一性.

查看api文档:

interface Map<K,V>

K - 此映射所维护的键的类型

V - 映射值的类型

概念

将键映射到值的对象。一个映射不能包含重复的键；每个键最多只能映射到一个值。

特点

Key和Value是1对1的关系，如：门牌号 ：家 老公:老婆

双列集合

|  |
| --- |
| Map学习体系：  ---| Map 接口 将键映射到值的对象。一个映射不能包含重复的键；每个键最多只能映射到一个值。  ---| HashMap 采用哈希表实现，所以无序  ---| TreeMap 可以对健进行排序 |

|  |
| --- |
| ---|Hashtable:  底层是哈希表数据结构，线程是同步的，不可以存入null键，null值。  效率较低，被HashMap 替代。  ---|HashMap:  底层是哈希表数据结构，线程是不同步的，可以存入null键，null值。  要保证键的唯一性，需要覆盖hashCode方法，和equals方法。  ---| LinkedHashMap：  该子类基于哈希表又融入了链表。可以Map集合进行增删提高效率。  ---|TreeMap:  底层是二叉树数据结构。可以对map集合中的键进行排序。需要使用Comparable或者Comparator 进行比较排序。return 0，来判断键的唯一性。 |

常见方法

|  |
| --- |
| 1、添加：  1、V put(K key, V value) （可以相同的key值，但是添加的value值会覆  盖前面的，返回值是前一个，如果没有就返回null）  2、putAll(Map<? extends K,? extends V> m) 从指定映射中将所有映射关  系复制到此映射中（可选操作）。  2、删除  1、remove() 删除关联对象，指定key对象  2、clear() 清空集合对象  3、获取  1：value get(key); 可以用于判断键是否存在的情况。当指定的键不存在的时候，返  回的是null。  3、判断：  1、boolean isEmpty() 长度为0返回true否则false  2、boolean containsKey(Object key) 判断集合中是否包含指定的key  3、boolean containsValue(Object value) 判断集合中是否包含指定的value  4、长度：  Int size（） |

添加：

该案例使用了HashMap，建立了学生姓名和年龄之间的映射关系。并试图添加重复的键。

|  |
| --- |
| **import** java.util.HashMap;  **import** java.util.Map;  **public** **class** Demo1 {  **public** **static** **void** main(String[] args) {  // 定义一个Map的容器对象  Map<String, Integer > map1 = **new** HashMap<String, Integer >();  map1.put("jack", 20);  map1.put("rose", 18);  map1.put("lucy", 17);  map1.put("java", 25);  System.*out*.println(map1);  // 添加重复的键值（值不同）,会返回集合中原有（重复键）的值， System.*out*.println(map1.put("jack", 30)); //20    Map<String, Integer> map2 = **new** HashMap<String, Integer>();  map2.put("张三丰", 100);  map2.put("虚竹", 20);  System.*out*.println("map2:" + map2);  // 从指定映射中将所有映射关系复制到此映射中。  map1.putAll(map2);  System.*out*.println("map1:" + map1);  //  }  } |

删除:

|  |
| --- |
| // 删除：  // remove() 删除关联对象，指定key对象  // clear() 清空集合对象  Map<String, Integer> map1 = **new** HashMap<String, Integer>();  map1.put("jack", 20);  map1.put("rose", 18);  map1.put("lucy", 17);  map1.put("java", 25);  System.*out*.println(map1);  // 指定key，返回删除的键值对映射的值。  System.*out*.println("value:" + map1.remove("java"));  map1.clear();  System.*out*.println("map1:" + map1); |

获取:

|  |
| --- |
| // 获取：  // V get(Object key) 通过指定的key对象获取value对象  // int size() 获取容器的大小  Map<String, Integer> map1 = **new** HashMap<String, Integer>();  map1.put("jack", 20);  map1.put("rose", 18);  map1.put("lucy", 17);  map1.put("java", 25);  System.*out*.println(map1);  // V get(Object key) 通过指定的key对象获取value对象  // int size() 获取容器的大小  System.*out*.println("value:" + map1.get("jack"));  System.*out*.println("map.size:" + map1.size()); |

判断:

|  |
| --- |
| // 判断：  // boolean isEmpty() 长度为0返回true否则false  // boolean containsKey(Object key) 判断集合中是否包含指定的key  // boolean containsValue(Object value)  Map<String, Integer> map1 = **new** HashMap<String, Integer>();  map1.put("jack", 20);  map1.put("rose", 18);  map1.put("lucy", 17);  map1.put("java", 25);  System.*out*.println(map1);  System.*out*.println("isEmpty:" + map1.isEmpty());  System.*out*.println("containskey:" + map1.containsKey("jack"));  System.*out*.println("containsvalues:" + map1.containsValue(100)); |

遍历Map的方式：

|  |
| --- |
| 1、将map 集合中所有的键取出存入set集合。  Set<K> keySet() 返回所有的key对象的Set集合  再通过get方法获取键对应的值。  2、 values() ，获取所有的值.  Collection<V> values()不能获取到key对象  3、 Map.Entry对象 推荐使用 重点  Set<Map.Entry<k,v>> entrySet()  将map 集合中的键值映射关系打包成一个对象  Map.Entry对象通过Map.Entry 对象的getKey，  getValue获取其键和值。 |
|  |

第一种方式:使用keySet

将Map转成Set集合（keySet()），通过Set的迭代器取出Set集合中的每一个元素（Iterator）就是Map集合中的所有的键，再通过get方法获取键对应的值。

|  |
| --- |
| **import** java.util.HashMap;  **import** java.util.Iterator;  **import** java.util.Map;  **import** java.util.Set;  **public** **class** Demo2 {  **public** **static** **void** main(String[] args) {  Map<Integer, String> map = **new** HashMap<Integer, String>();  map.put(1, "aaaa");  map.put(2, "bbbb");  map.put(3, "cccc");  System.*out*.println(map);  //  // 获取方法：  // 第一种方式: 使用keySet  // 需要分别获取key和value，没有面向对象的思想  // Set<K> keySet() 返回所有的key对象的Set集合  Set<Integer> ks = map.keySet();  Iterator<Integer> it = ks.iterator();  **while** (it.hasNext()) {  Integer key = it.next();  String value = map.get(key);  System.*out*.println("key=" + key + " value=" + value);  }  }  } |

第二种方式: 通过values 获取所有值,不能获取到key对象

|  |
| --- |
| **public** **static** **void** main(String[] args) {  Map<Integer, String> map = **new** HashMap<Integer, String>();  map.put(1, "aaaa");  map.put(2, "bbbb");  map.put(3, "cccc");  System.*out*.println(map);  // 第二种方式：  // 通过values 获取所有值,不能获取到key对象  // Collection<V> values()  Collection<String> vs = map.values();  Iterator<String> it = vs.iterator();  **while** (it.hasNext()) {  String value = it.next();  System.*out*.println(" value=" + value);  }  } |

第三种方式: Map.Entry

public static interface Map.Entry<K,V>

通过Map中的entrySet()方法获取存放Map.Entry<K,V>对象的Set集合。

Set<Map.Entry<K,V>> entrySet()

面向对象的思想将map集合中的键和值映射关系打包为一个对象，就是Map.Entry

，将该对象存入Set集合，Map.Entry是一个对象，那么该对象具备的getKey，getValue获得键和值。

|  |
| --- |
| **public** **static** **void** main(String[] args) {  Map<Integer, String> map = **new** HashMap<Integer, String>();  map.put(1, "aaaa");  map.put(2, "bbbb");  map.put(3, "cccc");  System.*out*.println(map);  // 第三种方式： Map.Entry对象 推荐使用 重点  // Set<Map.Entry<K,V>> entrySet()    // 返回的Map.Entry对象的Set集合 Map.Entry包含了key和value对象  Set<Map.Entry<Integer, String>> es = map.entrySet();  Iterator<Map.Entry<Integer, String>> it = es.iterator();  **while** (it.hasNext()) {    // 返回的是封装了key和value对象的Map.Entry对象  Map.Entry<Integer, String> en = it.next();  // 获取Map.Entry对象中封装的key和value对象  Integer key = en.getKey();  String value = en.getValue();  System.*out*.println("key=" + key + " value=" + value);  }  } |

## HashMap

底层是哈希表数据结构，线程是不同步的，可以存入null键，null值。要保证键的唯一性，需要覆盖hashCode方法，和equals方法。

案例：自定义对象作为Map的键。

|  |
| --- |
| **package** cn.itcast.gz.map;  **import** java.util.HashMap;  **import** java.util.Iterator;  **import** java.util.Map.Entry;  **import** java.util.Set;  **public** **class** Demo3 {  **public** **static** **void** main(String[] args) {  HashMap<Person, String> hm = **new** HashMap<Person, String>();  hm.put(**new** Person("jack", 20), "1001");  hm.put(**new** Person("rose", 18), "1002");  hm.put(**new** Person("lucy", 19), "1003");  hm.put(**new** Person("hmm", 17), "1004");  hm.put(**new** Person("ll", 25), "1005");  System.*out*.println(hm);  System.*out*.println(hm.put(**new** Person("rose", 18), "1006"));  Set<Entry<Person, String>> entrySet = hm.entrySet();  Iterator<Entry<Person, String>> it = entrySet.iterator();  **while** (it.hasNext()) {  Entry<Person, String> next = it.next();  Person key = next.getKey();  String value = next.getValue();  System.*out*.println(key + " = " + value);  }  }  }  **class** Person {  **private** String name;  **private** **int** age;  Person() {  }  **public** Person(String name, **int** age) {  **this**.name = name;  **this**.age = age;  }  **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;  }  @Override  **public** **int** hashCode() {  **return** **this**.name.hashCode() + age \* 37;  }  @Override  **public** **boolean** equals(Object obj) {  **if** (obj **instanceof** Person) {  Person p = (Person) obj;  **return** **this**.name.equals(p.name) && **this**.age == p.age;  } **else** {  **return** **false**;  }  }  @Override  **public** String toString() {  **return** "Person@name:" + **this**.name + " age:" + **this**.age;  }  }  } |

## TreeMap

TreeMap的排序，TreeMap可以对集合中的键进行排序。如何实现键的排序？

方式一：元素自身具备比较性

和TreeSet一样原理，需要让存储在键位置的对象实现Comparable接口，重写compareTo方法，也就是让元素自身具备比较性，这种方式叫做元素的自然排序也叫做默认排序。

方式二：容器具备比较性

当元素自身不具备比较性，或者自身具备的比较性不是所需要的。那么此时可以让容器自身具备。需要定义一个类实现接口Comparator，重写compare方法，并将该接口的子类实例对象作为参数传递给TreeMap集合的构造方法。

注意：当Comparable比较方式和Comparator比较方式同时存在时，以Comparator的比较方式为主；

注意：在重写compareTo或者compare方法时，必须要明确比较的主要条件相等时要比较次要条件。（假设姓名和年龄一直的人为相同的人，如果想要对人按照年龄的大小来排序，如果年龄相同的人，需要如何处理？不能直接return 0，以为可能姓名不同（年龄相同姓名不同的人是不同的人）。此时就需要进行次要条件判断（需要判断姓名），只有姓名和年龄同时相等的才可以返回0.）

通过return 0来判断唯一性。

|  |
| --- |
| **import** java.util.TreeMap;  **public** **class** Demo4 {  **public** **static** **void** main(String[] args) {  TreeMap<String, Integer> tree = **new** TreeMap<String, Integer>();  tree.put("张三", 19);  tree.put("李四", 20);  tree.put("王五", 21);  tree.put("赵六", 22);  tree.put("周七", 23);  tree.put("张三", 24);  System.*out*.println(tree);  System.*out*.println("张三".compareTo("李四"));//-2094  }  } |

自定义元素排序

|  |
| --- |
| **package** cn.itcast.gz.map;  **import** java.util.Comparator;  **import** java.util.Iterator;  **import** java.util.Map.Entry;  **import** java.util.Set;  **import** java.util.TreeMap;  **public** **class** Demo3 {  **public** **static** **void** main(String[] args) {  TreeMap<Person, String> hm = **new** TreeMap<Person, String>(  **new** MyComparator());  hm.put(**new** Person("jack", 20), "1001");  hm.put(**new** Person("rose", 18), "1002");  hm.put(**new** Person("lucy", 19), "1003");  hm.put(**new** Person("hmm", 17), "1004");  hm.put(**new** Person("ll", 25), "1005");  System.*out*.println(hm);  System.*out*.println(hm.put(**new** Person("rose", 18), "1006"));  Set<Entry<Person, String>> entrySet = hm.entrySet();  Iterator<Entry<Person, String>> it = entrySet.iterator();  **while** (it.hasNext()) {  Entry<Person, String> next = it.next();  Person key = next.getKey();  String value = next.getValue();  System.*out*.println(key + " = " + value);  }  }  }  **class** MyComparator **implements** Comparator<Person> {  @Override  **public** **int** compare(Person p1, Person p2) {  **if** (p1.getAge() > p2.getAge()) {  **return** -1;  } **else** **if** (p1.getAge() < p2.getAge()) {  **return** 1;  }  **return** p1.getName().compareTo(p2.getName());  }  }  **class** Person **implements** Comparable<Person> {  **private** String name;  **private** **int** age;  Person() {  }  **public** Person(String name, **int** age) {  **this**.name = name;  **this**.age = age;  }  **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;  }  @Override  **public** **int** hashCode() {  **return** **this**.name.hashCode() + age \* 37;  }  @Override  **public** **boolean** equals(Object obj) {  **if** (obj **instanceof** Person) {  Person p = (Person) obj;  **return** **this**.name.equals(p.name) && **this**.age == p.age;  } **else** {  **return** **false**;  }  }  @Override  **public** String toString() {  **return** "Person@name:" + **this**.name + " age:" + **this**.age;  }  @Override  **public** **int** compareTo(Person p) {  **if** (**this**.age > p.age) {  **return** 1;  } **else** **if** (**this**.age < p.age) {  **return** -1;  }  **return** **this**.name.compareTo(p.name);  }  } |

注意：Set的元素不可重复，Map的键不可重复，如果存入重复元素如何处理

Set元素重复元素不能存入add方法返回false

Map的重复健将覆盖旧键，将旧值返回。

# Collections与Arrays

集合框架中的工具类：特点：该工具类中的方法都是静态的。

|  |
| --- |
| **Collections**：常见方法：   1. 对list进行二分查找：   前提该集合一定要有序。  int binarySearch(list,key);  //必须根据元素自然顺序对列表进行升级排序  //要求list 集合中的元素都是Comparable 的子类。  int binarySearch(list,key,Comparator);  2，对list集合进行排序。  sort(list);  //对list进行排序,其实使用的事list容器中的对象的compareTo方法  sort(list,comaprator);  //按照指定比较器进行排序  3，对集合取最大值或者最小值。  max(Collection)  max(Collection,comparator)  min(Collection)  min(Collection,comparator) 4，对list集合进行反转。  reverse(list);  5，对比较方式进行强行逆转。  Comparator reverseOrder();  Comparator reverseOrder(Comparator);  6，对list集合中的元素进行位置的置换。  swap(list,x,y);  7，对list集合进行元素的替换。如果被替换的元素不存在，那么原集合不变。  replaceAll(list,old,new);  8，可以将不同步的集合变成同步的集合。  Set synchronizedSet(Set<T> s)  Map synchronizedMap(Map<K,V> m)  List synchronizedList(List<T> list)  9. 如果想要将集合变数组：  可以使用Collection 中的toArray 方法。注意：是Collection不是Collections工具类  传入指定的类型数组即可，该数组的长度最好为集合的size。 |

Arrays:用于对数组操作的工具类

|  |
| --- |
| 1，二分查找,数组需要有序  binarySearch(int[])  binarySearch(double[])  2，数组排序  sort(int[])  sort(char[])……   1. 将数组变成字符串。   toString(int[])   1. 复制数组。  copyOf(); 2. 复制部分数组。   copyOfRange():   1. 比较两个数组是否相同。   equals(int[],int[]);   1. 将数组变成集合。   List asList(T[]);  这样可以通过集合的操作来操作数组中元素，  但是不可以使用增删方法，add，remove。因为数组长度是固定的，会出现  UnsupportOperationExcetion。  可以使用的方法：contains，indexOf。。。  如果数组中存入的基本数据类型，那么asList会将数组实体作为集合中的元素。  如果数组中的存入的引用数据类型，那么asList会将数组中的元素作为集合中  的元素。 |

|  |
| --- |
| **import** java.util.ArrayList;  **import** java.util.Collections;  **import** java.util.Arrays;  **import** java.util.List;  **class** Demo1  {  **public** **static** **void** main(String[] args)  {  ArrayList<Integer> list = **new** ArrayList<Integer>();  list.add(4);  list.add(3);  list.add(1);  list.add(2);  list.add(3);  // 排序  Collections.*sort*(list);  // 折半查找的前提是排序好的元素  System.*out*.println( Collections.*binarySearch*( list , 8 ) ); // 找不到返回-插入点-1  // 反序集合输出  Collections.*reverse*( list );  System.*out*.println( list );  // 求最值  System.*out*.println( Collections.*max*( list ) ); // 4  // fill() 使用指定的元素替换指定集合中的所有元素  // Collections.fill( list, 5 );  System.*out*.println( list );  // 将数组转换为集合  Integer is[] = **new** Integer[]{6,7,8};  List<Integer> list2 = Arrays.*asList*(is);  list.addAll( list2 );  System.*out*.println( list );  // 将List转换为数组  Object [] ins = list.toArray();  System.*out*.println( Arrays.*toString*( ins ) );    }  } |

集合的练习

问题： 定义一个Person数组，将Person数组中的重复对象剔除？

思路:

1. 描述一个Person类

2. 将数组转换为Arrays.asList() List

3. Set addAll( list )

4. hashCode()且equals()

|  |
| --- |
| import java.util.Arrays;  import java.util.Set;  import java.util.List;  import java.util.HashSet;  // 1. 描述Person类  class Person {  public String name;  public int age;  public Person() {  }  public Person(String name, int age) {  this.name = name;  this.age = age;  }  public String toString() {  return getClass().getName() + " : name=" + this.name + " age="  + this.age;  }  // 4. 重写hashCode和equals()  public int hashCode() {  return this.age;  }  public boolean equals(Object o) {  Person p = null;  if (o instanceof Person)  p = (Person) o;  return this.name.equals(p.name) && (this.age == p.age);  }  }  class Demo2 {  public static void main(String[] args) {  Person[] ps = new Person[] { new Person("jack", 34),  new Person("lucy", 20), new Person("lili", 10),  new Person("jack", 34) };  // 遍历数组  System.out.println(Arrays.toString(ps));  // 2. 将自定义对象数组转换为List集合  List<Person> list = Arrays.asList(ps);  // 3. 将List转换为Set  Set<Person> set = new HashSet<Person>();  set.addAll(list);  System.out.println(set);  }  } |

# 泛型(Generic)

当集合中存储的对象类型不同时，那么会导致程序在运行的时候的转型异常

|  |
| --- |
| **import** java.util.ArrayList;  **import** java.util.Iterator;  **public** **class** Demo5 {  **public** **static** **void** main(String[] args) {  ArrayList arr = **new** ArrayList();  arr.add(**new** Tiger("华南虎"));  arr.add(**new** Tiger("东北虎"));  arr.add(**new** Sheep("喜羊羊"));  System.*out*.println(arr);  Iterator it = arr.iterator();  **while** (it.hasNext()) {  Object next = it.next();  Tiger t = (Tiger) next;  t.eat();  }  }  }  **class** Tiger {  String name;  **public** Tiger() {  }  **public** Tiger(String name) {  **this**.name = name;  }  @Override  **public** String toString() {  **return** "Tiger@name:" + **this**.name;  }  **public** **void** eat() {  System.*out*.println(**this**.name + "吃羊");  }  }  **class** Sheep {  String name;  **public** Sheep() {  }  **public** Sheep(String name) {  **this**.name = name;  }  @Override  **public** String toString() {  **return** "Sheep@name:" + **this**.name;  }  **public** **void** eat() {  System.*out*.println(**this**.name + "吃青草");  }  } |

原因 :发现虽然集合可以存储任意对象,但是如果需要使用对象的特有方法,那么就需要类型转换,如果集合中存入的对象不同,可能引发类型转换异常.

|  |
| --- |
| [Tiger@name:华南虎, Tiger@name:东北虎, Sheep@name:喜羊羊]  华南虎吃羊  东北虎吃羊  Exception in thread "main" java.lang.ClassCastException: cn.itcast.gz.map.Sheep cannot be cast to cn.itcast.gz.map.Tiger  at cn.itcast.gz.map.Demo5.main(Demo5.java:17) |

出现问题:

存入的是特定的对象,取出的时候是Object对象,需要强制类型转换,可能诱发类型转换异常.

无法控制存入的是什么类型的对象,取出对象的时候进行强转时可能诱发异常.而且在编译时期无法发现问题.

虽然可以再类型转换的时候通过if语句进行类型检查(instanceof),但是效率较低.(例如吃饭的时候,还需要判断米饭里有没有沙子,吃饭效率低).可以通过给容器加限定的形式规定容器只能存储一种类型的对象.

就像给容器贴标签说明该容器中只能存储什么样类型的对象。

所以在jdk5.0后出现了泛型

泛型应用:

格式

1. 集合类<类类型> 变量名 = new 集合类<类类型>();

|  |
| --- |
| **public** **class** Demo5 {  **public** **static** **void** main(String[] args) {  // 使用泛型后,规定该集合只能放羊,老虎就进不来了.  ArrayList<Sheep> arr = **new** ArrayList<Sheep>();  arr.add(**new** Sheep("美羊羊"));  arr.add(**new** Sheep("懒洋洋"));  arr.add(**new** Sheep("喜羊羊"));  // 编译失败  // arr.add(new Tiger("东北虎"));  System.*out*.println(arr);  Iterator<Sheep> it = arr.iterator();  **while** (it.hasNext()) {  // 使用泛型后,不需要强制类型转换了  Sheep next = it.next();  next.eat();  }  }  } |

1. 将运行时的异常提前至编译时发生。

2. 获取元素的时候无需强转类型，就避免了类型转换的异常问题

格式 通过<> 来指定容器中元素的类型.

什么时候使用泛型:当类中操作的引用数据类型不确定的时候,就可以使用泛型类.

JDK5.0之前的Comparable

|  |
| --- |
| **package java.lang;**  **public** **interface** Comparable {  **public** **int** compareTo(Object o);  } |

JDK5.0之后的Comparable

|  |
| --- |
| **package java.lang;**  **public** **interface** Comparable<T> {  **public** **int** compareTo(T o);  } |

这里的<T>表示泛型类型,随后可以传入具体的类型来替换它.

细节一

声明好泛型类型之后,集合中只能存放特定类型元素

|  |
| --- |
| **public** **class** Demo6 {  **public** **static** **void** main(String[] args) {  //创建一个存储字符串的list  ArrayList<String> arr=**new** ArrayList<String>();  arr.add("gz");  arr.add("itcast");  //存储非字符串编译报错.  arr.add(1);  }  } |

细节二:

泛型类型必须是引用类型

|  |
| --- |
| **public** **class** Demo6 {  **public** **static** **void** main(String[] args) {  // 泛型类型必须是引用类型,也就是说集合不能存储基本数据类型  // ArrayList<int> arr2=new ArrayList<int>();  // 使用基本数据类型的包装类  ArrayList<Integer> arr2 = **new** ArrayList<Integer>();      }  } |

细节三: 使用泛型后取出元素不需要类型转换.

|  |
| --- |
| **public** **class** Demo6 {  **public** **static** **void** main(String[] args) {  ArrayList<String> arr = **new** ArrayList<String>();  arr.add("gzitcast");  arr.add("cditcast");  arr.add("bjitcast");  //使用泛型后取出元素不需要类型转换.  String str=arr.get(0);  System.*out*.println();  }  } |

## 泛型方法

需求：写一个函数，调用者传递什么类型的变量，该函数就返回什么类型的变量？

实现一:

由于无法确定具体传递什么类型的数据.那么方法的形参就定义为Object类型.返回值也就是Object类型.但是使用该函数时需要强制类型转换.

|  |
| --- |
| **private** Object getDate(Object obj) {  **return** obj;  } |

当不进行强制类型转换能否写出该功能.?

目前所学的知识无法解决该问题

就需要使用泛型类解决

使用的泛型的自定义来解决以上问题。

泛型： 就是将类型当作变量处理。规范泛型的定义一般是一个大写的任意字母。

|  |
| --- |
| 1. 函数上的泛型定义  当函数中使用了一个不明确的数据类型，那么在函数上就可以进行泛型的定义。  public <泛型的声明> 返回值类型 函数名( 泛型 变量名 ){      } |

|  |
| --- |
| **public** **static** **void** main(String[] args) {  **int**[] arr = { 1, 2, 3, 4, 5 };    new Demo6().*getData*(5);  }  **public** <T> T getData(T data) {  **return** data;  } |

细节：

使用泛型方法前需要进行泛型声明，使用一对尖括号 <泛型>，声明的位置在static后返回值类型前。

当一个类中有多个函数声明了泛型，那么该泛型的声明可以声明在类上。

## 泛型类

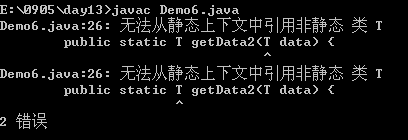
格式

|  |
| --- |
| 2. 类上的泛型声明  修饰符 class 类名<泛型>{    } |

|  |
| --- |
| **import** java.util.Arrays;  **public** **class** Demo6<T> {  **public** **static** **void** main(String[] args) {  // 使用泛型类，创建对象的时候需要指定具体的类型  **new** Demo6<Integer>().getData(5);  }  **public** T getData(T data) {  **return** data;  }  // 反序任意类型数组  **public** **void** reverse(T[] arr) {  **int** start = 0;  **int** end = arr.length - 1;  **for** (**int** i = 0; i < arr.length; i++) {  **if** (start < end) {  T temp = arr[start];  arr[start] = arr[end];  arr[end] = temp;  }  }    } |

在泛型类中定义一个静态方法

|  |
| --- |
| public class Demo6<T> {  public static void main(String[] args) {  System.out.println(getData2(100));  }  public T getData(T data) {  return data;  }  //静态方法  public static T getData2(T data) {  return data;  }  } |



注意：静态方法不可以使用类中定义的泛型

因为类中的泛型需要在对象初始化时指定具体的类型，而静态优先于对象存在。那么类中的静态方法就需要单独进行泛型声明，声明泛型一定要写在static后，返回值类型之前

泛型类细节：

|  |
| --- |
| 1、创建对象的时候要指定泛型的具体类型  2、创建对象时可以不指定泛型的具体类型(和创建集合对象一眼)。默认是Object，例如我们使用集合存储元素的时候没有使用泛型就是那么参数的类型就是Object  3、类上面声明的泛型只能应用于非静态成员函数，如果静态函数需要使用泛型，那么  需要在函数上独立声明。  4、如果建立对象后指定了泛型的具体类型，那么该对象操作方法时，这些方法只能操作一种数据类型。  5、所以既可以在类上的泛型声明，也可以在同时在该类的方法中声明泛型。 |

泛型练习：

定义泛型成员

|  |
| --- |
| **public** **class** Demo7 {  **public** **static** **void** main(String[] args) {  Father<String> f = **new** Father<String>("jack");  System.*out*.println(f.getT());  Father<Integer> f2 = **new** Father<Integer>(20);  System.*out*.println(f2.getT());  }  }  **class** Father<T> {  **private** T t;  **public** Father() {  }  **public** Father(T t) {  **super**();  **this**.t = t;  }  **public** T getT() {  **return** t;  }  **public** **void** setT(T t) {  **this**.t = t;  }  } |

如果Father类有子类，子类该如何实现

|  |
| --- |
| **public** **class** Demo7 {  **public** **static** **void** main(String[] args) {  Father<String> f = **new** Father<String>("jack");  System.*out*.println(f.getT());  Father<Integer> f2 = **new** Father<Integer>(20);  System.*out*.println(f2.getT());  }  }  **class** Father<T> {  **private** T t;  **public** Father() {  }  **public** Father(T t) {  **super**();  **this**.t = t;  }  **public** T getT() {  **return** t;  }  **public** **void** setT(T t) {  **this**.t = t;  }  }  //子类指定了具体的类型  **class** Son **extends** Father<String>{    }  //子类也需要使用泛型  **class** Son3<T> **extends** Father<T>{    }  //错误写法，父类上定义有泛型需要进行处理  **class** Son2 **extends** Father<T>{    } |

## 泛型接口

|  |
| --- |
| **public** **class** Demo8 {  **public** **static** **void** main(String[] args) {  MyInter<String> my = **new** MyInter<String>();  my.print("泛型");  MyInter2 my2 = **new** MyInter2();  my.print("只能传字符串");  }  }  **interface** Inter<T> {  **void** print(T t);  }  // 实现不知为何类型时可以这样定义  **class** MyInter<T> **implements** Inter<T> {  **public** **void** print(T t) {  System.*out*.println("myprint:" + t);  }  }  //使用接口时明确具体类型。  **class** MyInter2 **implements** Inter<String> {  @Override  **public** **void** print(String t) {  System.*out*.println("myprint:" + t);  }  } |

## 泛型通配符

需求：

定义一个方法，接收一个集合对象(该集合有泛型)，并打印出集合中的所有元素。

例如集合对象如下格式:

|  |
| --- |
| Collection<Person> coll = **new** ArrayList<Person>();  coll.add(**new** Person("jack", 20));  coll.add(**new** Person("rose", 18));  Collection<Object> coll2 = **new** ArrayList<Object>();  coll2.add(**new** Object());  coll2.add(**new** Object());  coll2.add(**new** Object());  Collection<String> coll3 = **new** ArrayList<String>();  coll3.add("abc");  coll3.add("ddd");  coll3.add("eee"); |

分析,集合对象中的元素的类型是变化的,方法的形参的那么泛型类型就只能定义为Object类型.

|  |
| --- |
| **import** java.util.ArrayList;  **import** java.util.Collection;  **import** java.util.HashSet;  **import** java.util.Iterator;  **public** **class** Demo9 {  **public** **static** **void** main(String[] args) {  ArrayList<Object> arr = **new** ArrayList<Object>();  arr.add(**new** Object());  arr.add("String");  *print*(arr);    //将集合的泛型设置类String类型，是Object子类  HashSet<String> hs = **new** HashSet<String>();  hs.add("hello");  hs.add("jack");  //由于print方法接收的集合进行了元素限定，只接受限定为Object类型的集合,编译不通过  //print(hs);  }  **public** **static** **void** print(Collection<Object> coll) {  Iterator<Object> it = coll.iterator();  **while** (it.hasNext()) {  Object next = it.next();  System.*out*.println(next);  }  }  } |

但是,由于print方法接收的集合进行了元素限定，只接受限定为Object类型的集合,编译不通过该问题如何解决？

可以把方法的形参的泛型去掉,那么方法中就把集合中的元素当做Object类型处理.

也可以使用使用泛型通配符

|  |
| --- |
| **public** **class** Demo9 {  **public** **static** **void** main(String[] args) {  ArrayList<Object> arr = **new** ArrayList<Object>();  arr.add(**new** Object());  arr.add("String");  *print*(arr);  // 将集合的泛型设置类String类型，是Object子类  HashSet<String> hs = **new** HashSet<String>();  hs.add("hello");  hs.add("jack");  // 使用泛型通配符，编译通过。  *print*(hs);  }  **public** **static** **void** print(Collection<?> coll) {  Iterator<?> it = coll.iterator();  **while** (it.hasNext()) {  Object next = it.next();  System.*out*.println(next);  }  }  } |

上述就使用了泛型通配符

|  |
| --- |
| 通配符：？  public void show(List<?> list)  {  }  可以对类型进行限定范围。  ？extends E: 接收E类型或者E的子类型。  ? super E: 接收E类型或者E的父类型。 |

限定泛型通配符的边界

限定通配符的上边界：

extends

接收**Number 类型或者Number的子类型**

|  |
| --- |
| **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

super

接收Integer 或者Integer的父类型

|  |
| --- |
| 正确：Vector<? super Integer> x = new Vector<Number>();  错误：Vector<? super Integer> x = new Vector<Byte>(); |

总结：

JDK5中的泛型允许程序员在编写集合代码时，就限制集合的处理类型，从而把原来程序运行时可能发生问题，转变为编译时的问题，以此提高程序的可读性和稳定

注意：泛型是提供给javac编译器使用的，它用于限定集合的输入类型，让编译器在源代码级别上，即挡住向集合中插入非法数据。但编译器编译完带有泛形的java程序后，生成的class文件中将不再带有泛形信息，以此使程序运行效率不受到影响，这个过程称之为“擦除”。

泛型的基本术语，以ArrayList<E>为例：<>念着typeof

ArrayList<E>中的E称为类型参数变量

ArrayList<Integer>中的Integer称为实际类型参数

整个称为ArrayList<E>泛型类型

整个ArrayList<Integer>称为参数化的类型ParameterizedType

最后:

关于数据结构可以查看如下网站:

http://www.cs.armstrong.edu/liang/animation/index.html 数据结构

# File类

## File类说明

存储在变量,数组和对象中的数据是暂时的,当程序终止时他们就会丢失.为了能够永

久的保存程序中创建的数据,需要将他们存储到硬盘或光盘的文件中.这些文件可以移动,传送,亦可以被其他程序使用.由于数据存储在文件中,所以我们需要学习一个和文件有密切关系的类,叫做File类,将要掌握获取文件的属性以及删除和重命名文件.最终如何向文件中写入数据和从文件中读取数据.

那么File类关心的是在磁盘上文件的存储.

File类描述的是一个文件或文件夹。（文件夹也可以称为目录）

该类的出现是对文件系统的中的文件以及文件夹进行对象的封装。可以通过对象的思想来操作文件以及文件夹。

可以用面向对象的处理问题，通过该对象的方法，可以得到文件或文件夹的信息方便了对文件与文件夹的属性信息进行操作。

文件包含很多的信息:如文件名、创建修改时间、大小、可读可写属性等。

## 体验File类

检验指定路径下是否存在指定的目录或者文件.

|  |
| --- |
| // 检验指定路径下是否存在指定的目录或者文件.  File file = **new** File("c:\\a.txt");  System.*out*.println(file.exists());  // File对象是否是目录  System.*out*.println(file.isDirectory());  // 对象是否是文件  System.*out*.println(file.isFile()); |

结论：File对象也可以表示不存在的文件。其实代表了一个抽象路径

构建一个File类的实例并不会在机器上创建一个文件.不管文件是否存在都可以创建任意文件名的File实例,可以调用File实例的exists方法判断文件或目录是否存在

## 构造一个File类实例：

|  |
| --- |
| new File(String pathname);  通过将给定路径来创建一个新File实例。  new File(String parent, String child);  根据parent路径名字符串和child路径名创建一个新File实例。  parent是指上级目录的路径，完整的路径为parent+child.  new File(File parent, String child);  根据parent抽象路径名和child路径名创建一个新File实例。  parent是指上级目录的路径，完整的路径为parent.getPath()+child.  说明：  如果指定的路径不存在（没有这个文件或是文件夹），不会抛异常，这时file.exists()返回false。 |

新建File对象File file=new File();

|  |
| --- |
| **public** **static** **void** main(String[] args) {  File file = **new** File();  } |

1:创建File对象需要导包, import java.io.File

2:File对象没有无参数构造.创建对象需要传参.

根据API文档提示,传入一个文件的字符串路径. String path="c:/a.txt";

(a.txt 文件在c盘下已经存在)

|  |
| --- |
| //file 是一个文件对象  String path = "c:/a.txt";  File file = **new** File(path); |

File类的对象，既可以代表文件也可以代表文件夹。

|  |
| --- |
| **public** **static** **void** main(String[] args) {  //file 是一个文件夹  String path = "c:/test";  File file = **new** File(path);  } |

## 路径：

路径就是文件或文件夹所在的位置。

### 路径分割符：

上下级文件夹之间使用分隔符分开：

在Windows中分隔符为'\'，在Unix/Linux中分隔符为'/'。

跨平台的目录分隔符

更专业的做法是使用File.separatorChar，这个值就会根据系统得到的相应的分割符。

例：new File("c:" + File.separatorChar + "a.txt");

注意，如果是使用"\"，则需要进行转义，写为"\\"才可以，如果是两个"\"，则写为"\\\\"。

实验：

|  |
| --- |
| 在以下代码的path处写不同的路径试一下，并观察输出结果。  File file = new File(path);  System.out.println(file.getAbsolutePath()); |

### 绝对路径与相对路径：

对于UNIX平台，绝对路径名的前缀是"/"。相对路径名没有前缀。

对于Windows平台，绝对路径名的前缀由驱动器号和一个":"组成，例"c:\\..."。相对路径没有盘符前缀。

相对路径：

相对路径是指相对于某位置的路径，是指相对于当前目录。

在执行Java程序时，相对路径为执行java命令时当前所在的目录。

实验：

|  |
| --- |
| 在不同的路径下执行java命令运行以下程序，并观察输出结果。  File file = new File("a.txt");  System.out.println(file.getAbsolutePath());  一般在使用时，建议用绝对路径，因为相对路径容易出问题，不好确定到底在什么地方。  //相对路径  //File file = new File("src/a.txt"); |

## File类中常用的方法：

创建：

|  |
| --- |
| createNewFile() 在指定位置创建一个空文件，成功就返回true，如果已存在就不创建然后返回false  mkdir() 在指定位置创建目录，这只会创建最后一级目录，如果上级目录不存在就抛异常。  mkdirs() 在指定位置创建目录，这会创建路径中所有不存在的目录。  renameTo(File dest) 重命名文件或文件夹，也可以操作非空的文件夹，文件不同时相当于文件的剪切,剪切时候不能操作非空的文件夹。移动/重命名成功则返回true，失败则返回false。 |

删除：

|  |
| --- |
| delete() 删除文件或一个空文件夹，如果是文件夹且不为空，则不能删除，成功返回true，失败返回false。  deleteOnExit() 在虚拟机终止时，请求删除此抽象路径名表示的文件或目录，保证程序异常时创建的临时文件也可以被删除 |

判断：

|  |
| --- |
| exists() 文件或文件夹是否存在。  isFile() 是否是一个文件，如果不存在，则始终为false。  isDirectory() 是否是一个目录，如果不存在，则始终为false。  isHidden() 是否是一个隐藏的文件或是否是隐藏的目录。  isAbsolute() 测试此抽象路径名是否为绝对路径名。 |

获取：

|  |
| --- |
| getName() 获取文件或文件夹的名称，不包含上级路径。  getPath() 返回绝对路径，可以是相对路径，但是目录要指定  getAbsolutePath() 获取文件的绝对路径，与文件是否存在没关系  length() 获取文件的大小（字节数），如果文件不存在则返回0L，如果是文件夹也返回0L。  getParent() 返回此抽象路径名父目录的路径名字符串；如果此路径名没有指定父目录，则返回null。  lastModified() 获取最后一次被修改的时间。  文件夹相关：  staic File[] listRoots() 列出所有的根目录（Window中就是所有系统的盘符）  list() 返回目录下的文件或者目录名，包含隐藏文件。对于文件这样操作会返回null。  list(FilenameFilter filter) 返回指定当前目录中符合过滤条件的子文件或子目录。对于文件这样操作会返回null。  listFiles() 返回目录下的文件或者目录对象（File类实例），包含隐藏文件。对于文件这样操作会返回null。  listFiles(FilenameFilter filter) 返回指定当前目录中符合过滤条件的子文件或子目录。对于文件这样操作会返回null。 |

## 案例：

1，列出指定目录中所有的子文件名与所有的子目录名。

2，列出指定目录中所有的子文件名与所有的子目录名，要求目录名与文件名分开列出，格式如下：

子目录：

...

...

子文件：

...

...

3，列出指定目录中所有扩展名为.java的文件。

4，列出指定目录中所有扩展名为.class的文件。

5，思考第3与第4题，代码是不是重复呢，如果想要列出其中的所有.txt文件，是不是要再写一个类呢？

a, 请自行设计一个工具方法，可以传递指定的扩展名，工具方法会过列出指定目录中指定扩展名的所有子文件与子文件夹。

b, 请利用FilenameFilter接口写一个工具类，可以传递指定的过滤规则。

6，列出指定目录中所有的子孙文件与子孙目录名，只需要列出名称即可。

解题: 列出指定目录中所有的子文件名与所有的子目录名。

|  |
| --- |
| 需求1：获取所有的c:/test 即test目录下的所有文件和文件夹  解题思路:  代码需要封装,就需要创建方法,并在main方法中调用和测试. 方法名要有意义: listAllFilesAndDirs  第一步：创建File对象  第二步：查找File类中可用的方法，想要获取该目录下的所有子文件和子目录  第三步：显示这些文件和文件夹的名称 |

实现:

|  |
| --- |
| /\*\*  \* 列出指定目录中所有包含的子文件与子目录  \*/  **public** **static** **void** listAllFilesAndDirs(String path) {  // 1.创建File对象，表示这个目录  File dir = **new** File(path);  // 2.通过list方法得到所包含的所有子目录与子文件名称  String[] names = dir.list();  // 3显示这些名称  **for** (**int** i = 0; i < names.length; i++) {  System.*out*.println(names[i]);  }  } |

解题: 列出指定目录中所有的子文件名与所有的子目录名，要求目录名与文件名分开列出

案例1把文件和文件夹都列了出来，但是无法区分文件和文件夹.File类有判断文件和文件夹的方法，但是list方法返回的是String数组，这个list() 方法无法满足我们的需求.继续查找File的方法.查看api找到 File[] listFiles() 发现该方法返回的是一个File数组。

|  |
| --- |
| 思路:  第一步:创建listAllFilesAndDirs(String path)方法,接受路径  第二步: 创建File对象表示这个目录  第三步: 通过listFiles方法得到所包含的所有子目录与子文件名称  第四步: 得到所有的文件名集合，与所有的文件夹名集合  第五步: 分别显示文件名与文件夹名 |

实现

|  |
| --- |
| **public** **static** **void** listAllFilesAndDirs2(String path) {  // 1.创建File对象，表示这个目录  File dir = **new** File(path);  // 2通过listFiles方法得到所包含的所有子目录与子文件名称  File[] names = dir.listFiles();  // 3,分别显示文件名与文件夹名  **for** (**int** i = 0; i < names.length; i++) {  File file = names[i];  **if** (file.isFile()) {  System.*out*.println(("子文件："));  System.*out*.println("\t" + file.getName());  } **else** **if** (file.isDirectory()) {  System.*out*.println(("子目录："));  System.*out*.println("\t" + file.getName());  }  }  } |

实现二:

|  |
| --- |
| **public** **static** **void** listAllFilesAndDirs(String path) {  //1创建File对象表示这个目录  File dir = **new** File(path);    //2通过listFiles方法得到所包含的所有子目录与子文件名称  File[] names = dir.listFiles();    //3,得到所有的文件名集合，与所有的文件夹名集合  List<File> filesList = **new** ArrayList<File>();  List<File> dirsList = **new** ArrayList<File>();  **for** (**int** i = 0; i < names.length; i++) {  File file = names[i];  **if** (file.isFile()) {  filesList.add(file);  } **else** **if** (file.isDirectory()) {  dirsList.add(file);  }  }    //4，分别显示文件名与文件夹名  System.*out*.println("子文件：");  **for** (**int** i = 0; i < filesList.size(); i++) {  System.*out*.println("\t" + filesList.get(i).getName());  }  System.*out*.println("子目录：");  **for** (**int** i = 0; i < dirsList.size(); i++) {  System.*out*.println("\t" + dirsList.get(i).getName());  }  } |

练习3，列出指定目录中所有扩展名为.java的文件。

|  |
| --- |
| 需求: 从指定目录中找到指定扩展名的文件，并列出来  思路  第一步: 创建lsitAllFiles方法,接受路径和文件后缀名  第二步: 获取所有的子文件和子文件夹  第三步: 从中找出符合条件的文件并显示出来 |

注意：不同系统对于路径的 windows系统使用斜线作为路径分隔符 "\" linux 系统使用反斜线作为路径分隔符"/"java是跨平台的语言，java程序如果部署到linux系统上，如果程序中有File对象， 可以使用File类*separatorChar* (字段)

|  |
| --- |
| **public** **class** FileTest2 {  **public** **static** **void** main(String[] args) {  String path = "c:" + File.*separatorChar* + "test";  File file = **new** File(path);  *listtAllFiles*(file, "java");  }  /\*\*  \* 从指定目录中找到指定扩展名的文件，并列出来  \*  \*/  **public** **static** **void** listtAllFiles(File dir, String extension) {  // 1.获取所有的子文件和子文件夹  File[] files = dir.listFiles();  // 2.从中找出符合条件的文件并显示出来  **for** (**int** i = 0; i < files.length; i++) {  File file = files[i];  // 3.需要以指定文件后缀结尾才算符合条件  **if** (file.getName().endsWith(extension)) {  System.*out*.println(file.getName());  }  }  }  } |

练习4:

|  |
| --- |
| **public** **class** FileTest2 {  **public** **static** **void** main(String[] args) {  String path = "c:" + File.*separatorChar* + "test";  File file = **new** File(path);  *listtAllFiles2*(file, "txt");  }  /\*\*  \* FilenameFilter接口写一个工具类，可以传递指定的过滤规则。  \* 从指定目录中找到指定扩展名的文件，并列出来  \*  \* \*/  **public** **static** **void** listtAllFiles2(File dir, String name) {  // 1.获取所有的子文件和子文件夹  String[] files = dir.list(**new** DirFilter("txt"));  // 2显示名称  **for** (**int** i = 0; i < files.length; i++) {  System.*out*.println(files[i]);  }  }  }  **class** DirFilter **implements** FilenameFilter {  **private** String extension;  **public** DirFilter() {  }  **public** DirFilter(String extension) {  **this**.extension = extension;  }  @Override  **public** **boolean** accept(File dir, String name) {  **return** name.endsWith(extension);  }  } |

注意: DirFilter 就是实现了accept 方法.提供给File类的list方法使用.

# IO流体验与简介

File对象可以表示存在的文件或文件夹，也可以表示不存在的。

我们想要得到文件的内容怎么办，File只是操作文件，文件的内容如何处理就需要使用io流技术了。

例如在C盘下有一个名称为a.txt的文本文件.想要通过Java程序读出来文件中的内容,需要使用IO流技术.同样想要将程序中的数据,保存到硬盘的文件中,也需要IO流技术.

读和写文件文件示例:

|  |
| --- |
| **public** **class** IoTest {  **public** **static** **void** main(String[] args) **throws** FileNotFoundException,  IOException {  *writFileTest*();  *readFileTest*();  }  **private** **static** **void** writFileTest() **throws** FileNotFoundException,  IOException {  // 创建文件对象  File file = **new** File("c:\\a.txt");  // 创建文件输出流  FileOutputStream fos = **new** FileOutputStream(file);  fos.write('g');  fos.write('z');  fos.write('i');  fos.write('t');  fos.write('c');  fos.write('a');  fos.write('s');  fos.write('t');  fos.close();  }  **private** **static** **void** readFileTest() **throws** FileNotFoundException,  IOException {  // 创建文件对象  File file = **new** File("c:\\a.txt");  // 创建文件输入流  FileInputStream fis = **new** FileInputStream(file);  // 有对多长，就读多少字节。  **for** (**int** i = 0; i < file.length(); i++) {  System.*out*.print((**char**) fis.read());  }  fis.close();  }  } |

当完成流的读写时,应该通过调用close方法来关闭它,这个方法会释放掉十分有限的操作系统资源.如果一个应用程序打开了过多的流而没有关闭它们,那么系统资源将被耗尽.

IO流简介：（Input/Output）

I/O类库中使用“流”这个抽象概念。Java对设备中数据的操作是通过流的方式。

表示任何有能力产出数据的数据源对象，或者是有能力接受数据的接收端对象。“流”屏蔽了实际的I/O设备中处理数据的细节。IO流用来处理设备之间的数据传输。设备是指硬盘、内存、键盘录入、网络等。

Java用于操作流的对象都在IO包中。IO流技术主要用来处理设备之间的数据传输。

由于Java用于操作流的对象都在IO包中。所以使用IO流需要导包如：import java.io.\*;

IO流的分类

流按操作数据类型的不同分为两种：字节流与字符流。

流按流向分为：输入流，输出流（以程序为参照物，输入到程序，或是从程序输出）

# 字节流

什么是字节流

计算机中都是二进制数据,一个字节是8个2进制位.字节可以表示所有的数据,比如文本,音频,视频.图片,都是作为字节存在的.也就是说字节流处理的数据非常多。

在文本文件中存储的数据是以我们能读懂的方式表示的。而在二进制文件中存储的数据是用二进制形式表示的。我们是读不懂二进制文件的，因为二进制文件是为了让程序来读取而设计的。例如，Java的源程序(.java源文件)存储在文本文件中,可以使用文本编辑器阅读,但是Java的类(字节码文件)存储在二进制文件中,可以被Java虚拟机阅读。二进制文件的优势在于它的处理效率比文本文件高。

我们已经知道File对象封装的是文件或者路径属性，但是不包含向（从）文件读（写）数据的方法。为了实现对文件的读和写操作需要学会正确的使用Java的IO创建对象。

字节流的抽象基类：

输入流：java.io.InputStream

输出流：java.io.OutputStream

特点：

字节流的抽象基类派生出来的子类名称都是以其父类名作为子类名的后缀。

如：FileInputStream, ByteArrayInputStream等。

说明：

字节流处理的单元是一个字节，用于操作二进制文件（计算机中所有文件都是二进制文件）

## InputStream

案例：读取"c:/a.txt"文件中的所有内容并在控制台显示出来。

注意：事先准备一个a.txt并放到c:/下，不要保存中文。

a, 使用read()方法实现。

b, 使用int read(byte[] b)方法实现。

写代码读取"c:/a.txt"文件中的所有的内容并在控制台显示出来

实现：

查看api文档(自己一定要动手)

InputStream 有read方法，一次读取一个字节，OutputStream的write方法一次写一个int。发现这两个类都是抽象类。意味着不能创建对象，那么需要找到具体的子类来使用。

通过查看api文档，找到了FileInputStream类，该类正是我们体验Io流的一个输入流。

|  |
| --- |
| 实现；显示指定文件内容。  明确使用流，使用哪一类流？使用输入流，FileInputStream  第一步：  1：打开流（即创建流）  第二步：  2：通过流读取内容  第三步：  3：用完后，关闭流资源 |

显然流是Java中的一类对象，要打开流其实就是创建具体流的对象，由于是读取硬盘上的文件，应该使用输入流。所以找到了InputStream类，但是InputStream是抽象类，需要使用它的具体实现类来创建对象就是FileInputStream。通过new 调用FileInputStream 的构造方法来创建对象。发现FileInputStream的构造方法需要指定文件的来源。查看构造方法，可以接受字符串也可以接受File对象。我们通过构建File对象指定文件路径。

使用流就像使用水管一样，要打开就要关闭。所以打开流和关闭流的动作是比不可少的。如何关闭流？使用close方法即可，当完成流的读写时,应该通过调用close方法来关闭它,这个方法会释放掉十分有限的操作系统资源.如果一个应用程序打开了过多的流而没有关闭它们,那么系统资源将被耗尽.

如何通过流读取内容？

查找api文档通过read方法，查看该方法，发现有返回值，并且是int类型的，该方法一次读取一个字节（byte）

### 输入流读取方式1：

read方法（）

一次读取一个字节,读到文件末尾返回-1.

仔细查看api文档发现read方法如果读到文件的末尾会返回-1。那么就可以通过read方法的返回值是否是-1来控制我们的循环读取。

|  |
| --- |
| /\*\*  \* 根据read方法返回值的特性，如果独到文件的末尾返回-1，如果不为-1就继续向下读。  \* \*/  **private** **static** **void** showContent(String path) **throws** IOException {  // 打开流  FileInputStream fis = **new** FileInputStream(path);  **int** len = fis.read();  **while** (len != -1) {  System.*out*.print((**char**)len);  len = fis.read();  }  // 使用完关闭流  fis.close();  } |

我们习惯这样写：

|  |
| --- |
| /\*\*  \* 根据read方法返回值的特性，如果独到文件的末尾返回-1，如果不为-1就继续向下读。  \* \*/  **private** **static** **void** showContent(String path) **throws** IOException {  // 打开流  FileInputStream fis = **new** FileInputStream(path);  **int** len;  **while** ((len = fis.read()) != -1) {  System.*out*.print((**char**) len);  }  // 使用完关闭流  fis.close();  } |

### 输入流读取方式2：

使用read(byte[] b) 方法。使用缓冲区(关键是缓冲区大小的确定)

使用read方法的时候，流需要读一次就处理一次，可以将读到的数据装入到字节数组中，一次性的操作数组，可以提高效率。

**问题1:缓冲区大小**

那么字节数组如何定义？定义多大？

可以尝试初始化长度为5的byte数组。通过read方法，往byte数组中存内容

那么该read方法返回的是往数组中存了多少字节。

|  |
| --- |
| /\*\*  \* 使用字节数组存储读到的数据  \* \*/  **private** **static** **void** showContent2(String path) **throws** IOException {  // 打开流  FileInputStream fis = **new** FileInputStream(path);  // 通过流读取内容  **byte**[] byt = **new** **byte**[5];  **int** len = fis.read(byt);  **for** (**int** i = 0; i < byt.length; i++) {  System.*out*.print((**char**) byt[i]);  }  // 使用完关闭流  fis.close();  } |

**问题1: 缓冲区太小**:

数据读取不完.

测试发现问题，由于数组太小，只装了5个字节。而文本的字节大于数组的长度。那么很显然可以将数组的长度定义大一些。例如1024个。

|  |
| --- |
| /\*\*  \* 使用字节数组存储读到的数据  \* \*/  **private** **static** **void** showContent2(String path) **throws** IOException {  // 打开流  FileInputStream fis = **new** FileInputStream(path);  // 通过流读取内容  **byte**[] byt = **new** **byte**[1024];  **int** len = fis.read(byt);  **for** (**int** i = 0; i < byt.length; i++) {  System.*out*.print(byt[i]);  }    // 使用完关闭流  fis.close();  } |

**问题三:缓冲区有默认值**.

测试,打印的效果打印出了很多0，因为数组数组有默认初始化值，所以，我们将数组的数据全部都遍历和出来.现在需要的是取出数组中的部分数据.需要将循环条件修改仔细查看api文档。发现该方法read(byte[] b)返回的是往数组中存入了多少个字节。就是数组实际存储的数据个数。

|  |
| --- |
| /\*\*  \* 使用字节数组存储读到的数据  \* \*/  **private** **static** **void** showContent2(String path) **throws** IOException {  // 打开流  FileInputStream fis = **new** FileInputStream(path);  // 通过流读取内容  **byte**[] byt = **new** **byte**[1024];  **int** len = fis.read(byt);  **for** (**int** i = 0; i <len; i++) {  System.*out*.print(byt[i]);  }    // 使用完关闭流  fis.close();  } |

总结：

问题一：为什么打印的不是字母而是数字，

是字母对应的码值。

如何显示字符，强转为char即可

问题二：注意：回车和换行的问题。

windows的换车和换行是"\r\n" 对应码表是13和10 。

### 输入流读取方式3：

使用read(byte[] b,int off,int len)

查看api文档，

b显然是一个byte类型数组，当做容器来使用

off，是指定从数组的什么位置开始存字节

len，希望读多少个

其实就是把数组的一部分当做流的容器来使用。告诉容器，从什么地方开始装要装多少。

|  |
| --- |
| /\*\*  \* 把数组的一部分当做流的容器来使用  \* read(byte[] b,int off,int len)  \*/  **private** **static** **void** showContent3(String path) **throws** IOException {  // 打开流  FileInputStream fis = **new** FileInputStream(path);  // 通过流读取内容  **byte**[] byt = **new** **byte**[1024];  // 从什么地方开始存读到的数据  **int** start = 5;    // 希望最多读多少个(如果是流的末尾，流中没有足够数据)  **int** maxLen = 6;  // 实际存放了多少个  **int** len = fis.read(byt, start, maxLen);  **for** (**int** i = start; i < start + maxLen; i++) {  System.*out*.print((**char**) byt[i]);  }  // 使用完关闭流  fis.close();  } |

需求2：测试skip方法

通过Io流,读取"c:/a.txt"文件中的第9个字节到最后所有的内容并在控制台显示出来。

分析：其实就是要跳过文件中的一部分字节，需要查找API文档。可以使用skip方法skip(long n)，参数跟的是要跳过的字节数。

我们要从第9个开始读，那么要跳过前8个即可。

|  |
| --- |
| /\*\*  \* skip方法  \*  \* \*/  **private** **static** **void** showContent4(String path) **throws** IOException {  // 打开流  FileInputStream fis = **new** FileInputStream(path);  // 通过流读取内容  **byte**[] byt = **new** **byte**[1024];  fis.skip(8);  **int** len = fis.read(byt);  System.*out*.println(len);  System.*out*.println("\*\*\*\*\*\*\*\*\*\*");  **for** (**int** i = 0; i < len; i++) {  System.*out*.println((**char**) byt[i]);  }  // 使用完关闭流  fis.close();  } |

### 输入流读取方式4：

使用缓冲(提高效率),并循环读取(读完所有内容).

总结：读完文件的所有内容。很显然可以使用普通的read方法，一次读一个字节直到读到文件末尾。为了提高效率可以使用read(byte[] byt);方法就是所谓的使用缓冲提高效率。我们可以读取大文本数据测试(大于1K的文本文件.)

|  |
| --- |
| /\*\*  \* 使用字节数组当缓冲  \* \*/  **private** **static** **void** showContent5(String path) **throws** IOException {  FileInputStream fis = **new** FileInputStream(path);  **byte**[] byt = **new** **byte**[1024];  **int** len = fis.read(byt);  System.*out*.println(len);  String buffer = **new** String(byt, 0, len);  System.*out*.print(buffer);  } |

注意：如何将字节数组转成字符串？ 可以通过创建字符串对象即可。

发现:一旦数据超过1024个字节，数组就存储不下。

如何将文件的剩余内容读完？

我们可以通过通过循环保证文件读取完。

|  |
| --- |
| /\*\*  \* 使用字节数组当缓冲  \* \*/  **private** **static** **void** showContent7(String path) **throws** IOException {  FileInputStream fis = **new** FileInputStream(path);  **byte**[] byt = **new** **byte**[1024];  **int** len = 0;  **while** ((len = fis.read(byt)) != -1) {  System.*out*.println(**new** String(byt, 0, len));  }  } |

## OutputStream

字节输出流

案例：

1，写代码实现把"Hello World!"写到"c:/a.txt"文件中。

a, c:/a.txt不存在时，测试一下。

b, c:/a.txt存在时，也测试一下。

要写两个版本：

a, 使用write(int b) 实现。

b, 使用write(byte[] b) 实现。

2，在已存在的c:/a.txt文本文件中追加内容：“Java IO”。

显然此时需要向指定文件中写入数据。

使用的就是可以操作文件的字节流对象。OutputStream。该类是抽象类，需要使用具体的实现类来创建对象查看API文档，找到了OutputStream的实现类FileOutputStream 创建FileOutputStream 流对象,必须指定数据要存放的目的地。通过构造函数的形式。创建流对象时，调用了系统底层的资源。在指定位置建立了数据存放的目的文件。

|  |
| --- |
| 流程：  1：打开文件输出流，流的目的地是指定的文件  2：通过流向文件写数据  3: 用完流后关闭流 |

### 输出流写出方式1：

使用write(int b)方法,一次写出一个字节.

在C盘下创建a.txt文本文件

|  |
| --- |
| **import** java.io.FileOutputStream;  **import** java.io.IOException;  **public** **class** IoTest2 {  **public** **static** **void** main(String[] args) **throws** IOException {  String path = "c:\\a.txt";  *writeTxtFile*(path);  }  **private** **static** **void** writeTxtFile(String path) **throws** IOException {  // 1：打开文件输出流，流的目的地是指定的文件  FileOutputStream fos = **new** FileOutputStream(path);  // 2：通过流向文件写数据  fos.write('j');  fos.write('a');  fos.write('v');  fos.write('a');  // 3:用完流后关闭流  fos.close();  }  } |

当c盘下的a.txt不存在会怎么样？

测试：将c盘下的a.txt文件删除，发现当文件不存在时，会自动创建一个，但是创建不了多级目录。

注意：使用write(int b)方法，虽然接收的是int类型参数，但是write 的常规协定是：向输出流写入一个字节。要写入的字节是参数 b 的八个低位。b 的 24 个高位将被忽略。

### 输出流写出方式2：

使用write(byte[] b),就是使用缓冲.提高效率.

上述案例中的使用了OutputStram 的write方法，一次只能写一个字节。成功的向文件中写入了内容。但是并不高效，如和提高效率呢？是否应该使用缓冲，根据字节输入流的缓冲原理，是否可以将数据保存中字节数组中。通过操作字节数组来提高效率。查找API文档，在OutputStram类中找到了write(byte[] b)方法，将 b.length 个字节从指定的 byte 数组写入此输出流中。

如何将字节数据保存在字节数组中，以字符串为例，”hello , world” 如何转为字节数组。显然通过字符串的getBytes方法即可。

|  |
| --- |
| **public** **class** IoTest2 {  **public** **static** **void** main(String[] args) **throws** IOException {  String path = "c:\\a.txt";  *writeTxtFile*(path);  }  **private** **static** **void** writeTxtFile(String path) **throws** IOException {  // 1：打开文件输出流，流的目的地是指定的文件  FileOutputStream fos = **new** FileOutputStream(path);  // 2：通过流向文件写数据  **byte**[] byt = "java".getBytes();  fos.write(byt);  // 3:用完流后关闭流  fos.close();  }  } |

仔细查看a.txt文本文件发现上述程序每运行一次，老的内容就会被覆盖掉。，那么如何不覆盖已有信息，能够往a.txt里追加信息呢。查看API文档，发现FileOutputStream类中的构造方法中有一个构造可以实现追加的功能FileOutputStream(File file, boolean append) 第二个参数，append - 如果为 true，则将字节写入文件末尾处，而不是写入文件开始处

|  |
| --- |
| **private** **static** **void** writeTxtFile(String path) **throws** IOException {  // 1：打开文件输出流，流的目的地是指定的文件  FileOutputStream fos = **new** FileOutputStream(path,**true**);  // 2：通过流向文件写数据  **byte**[] byt = "java".getBytes();  fos.write(byt);  // 3:用完流后关闭流  fos.close();  } |

## 字节流文件拷贝

### 字节输入输出流综合使用

通过字节输出流向文件中写入一些信息，并使用字节输入流把文件中的信息显示到控制台上。

|  |
| --- |
| **public** **class** IoTest3 {  **public** **static** **void** main(String[] args) **throws** IOException {  String path = "c:\\b.txt";  String content = "hello java";  *writeFile*(path, content);  *readFile*(path);  }  **public** **static** **void** writeFile(String path, String content)  **throws** IOException {  // 打开文件输出流  FileOutputStream fos = **new** FileOutputStream(path);  **byte**[] buffer = content.getBytes();  // 向文件中写入内容  fos.write(buffer);  // 关闭流  fos.close();  }  **public** **static** **void** readFile(String path) **throws** IOException {  FileInputStream fis = **new** FileInputStream(path);  **byte**[] byt = **new** **byte**[1024];  **int** len = 0;  **while** ((len = fis.read(byt)) != -1) {  System.*out*.println(**new** String(byt, 0, len));  }  // 关闭流  fos.close();  }  } |

注意输出流的细节：

这个输出流显然只适合小数据的写入，如果有大数据想要写入，我们的byte数组，该如何定义？

上述案例中我们将输入流和输出流进行和综合使用,如果尝试进输出流换成文本文件就可以实现文件的拷贝了.

什么是文件拷贝？很显然，先开一个输入流，将文件加载到流中，再开一个输出流，将流中数据写到文件中。就实现了文件的拷贝。

|  |
| --- |
| 分析：  第一步：需要打开输入流和输出流  第二步：读取数据并写出数据  第三步：关闭流  **public** **class** IoTest3 {  **public** **static** **void** main(String[] args) **throws** IOException {  String srcPath = "c:\\a.txt";  String destPath = "d:\\a.txt";  *copyFile*(srcPath, destPath);  }  **public** **static** **void** copyFile(String srcPath, String destPath)  **throws** IOException {  }  } |

### 字节流拷贝文件实现1

读一个字节写一个字节read 和write

|  |
| --- |
| **public** **class** IoTest3 {  **public** **static** **void** main(String[] args) **throws** IOException {  String srcPath = "c:\\a.txt";  String destPath = "d:\\a.txt";  *copyFile*(srcPath, destPath);  }  **public** **static** **void** copyFile(String srcPath, String destPath)  **throws** IOException {  // 打开输入流，输出流  FileInputStream fis = **new** FileInputStream(srcPath);  FileOutputStream fos = **new** FileOutputStream(destPath);  // 读取和写入信息  **int** len = 0;  **while** ((len = fis.read()) != -1) {  fos.write(len);  }  // 关闭流  fis.close();  fos.close();  }  } |

文本文件在计算机中是以二进制形式存在的,可以通过io流来拷贝,那么图片能不能拷贝呢?视频呢？音频呢？

|  |
| --- |
| **public** **class** IoTest3 {  **public** **static** **void** main(String[] args) **throws** IOException {  String srcPath = "c:\\秋.jpg";  String destPath = "d:\\秋.jpg";  *copyFile*(srcPath, destPath);  }  **public** **static** **void** copyFile(String srcPath, String destPath)  **throws** IOException {  // 打开输入流，输出流  FileInputStream fis = **new** FileInputStream(srcPath);  FileOutputStream fos = **new** FileOutputStream(destPath);  // 读取和写入信息  **int** len = 0;  **while** ((len = fis.read()) != -1) {  fos.write(len);  }  // 关闭流  fis.close();  fos.close();  }  } |

测试统统通过，所以字节流可以操作所有的文件。只是发现程序很慢，需要很长时间。特别是拷贝音频和视频文件时。

为什么？因为每次读一个字节再写一个字节效率很低。很显然这样效率低下的操作不是我们想要的。有没有更快更好的方法呢，是否可以使用缓冲区来提高程序的效率呢。

### 字节流拷贝文件实现2；

使用字节数组作为缓冲区

|  |
| --- |
| **public** **static** **void** copyFile2(String srcPath, String destPath)  **throws** IOException {  // 打开输入流，输出流  FileInputStream fis = **new** FileInputStream(srcPath);  FileOutputStream fos = **new** FileOutputStream(destPath);  // 读取和写入信息  **int** len = 0;  // 使用字节数组，当做缓冲区  **byte**[] byt = **new** **byte**[1024];  **while** ((len = fis.read(byt)) != -1) {  fos.write(byt);  }  // 关闭流  fis.close();  fos.close();  } |

问题1: 使用缓冲(字节数组)拷贝数据,拷贝后的文件大于源文件.

测试该方法，拷贝文本文件，仔细观察发现和源文件不太一致。

打开文件发现拷贝后的文件和拷贝前的源文件不同，拷贝后的文件要比源文件多一些内容问题就在于我们使用的容器，这个容器我们是重复使用的，新的数据会覆盖掉老的数据，显然最后一次读文件的时候，容器并没有装满，出现了新老数据并存的情况。

所以最后一次把容器中数据写入到文件中就出现了问题。

如何避免？使用FileOutputStream 的write(byte[] b, int off, int len)

b 是容器，off是从数组的什么位置开始，len是获取的个数，容器用了多少就写出多少。

|  |
| --- |
| **public** **static** **void** copyFile2(String srcPath, String destPath)  **throws** IOException {  // 打开输入流，输出流  FileInputStream fis = **new** FileInputStream(srcPath);  FileOutputStream fos = **new** FileOutputStream(destPath);  // 读取和写入信息  **int** len = 0;  // 使用字节数组，当做缓冲区  **byte**[] byt = **new** **byte**[1024];  **while** ((len = fis.read(byt)) != -1) {  fos.write(byt, 0, len);  }  // 关闭流  fis.close();  fos.close();  } |

使用缓冲拷贝视频，可以根据拷贝的需求调整数组的大小，一般是1024的整数倍。发现使用缓冲后效率大大提高。

## 字节流的异常处理

上述案例中所有的异常都只是进行了抛出处理，这样是不合理的。所以上述代码并不完善,因为异常没有处理。

当我们打开流,读和写,关闭流的时候都会出现异常,异常出现后,后面的代码都不会执行了。假设打开和关闭流出现了异常，那么显然close方法就不会再执行。那么会对程序有什么影响？

案例：

|  |
| --- |
| **public** **class** IoTest4 {  **public** **static** **void** main(String[] args) **throws** IOException,  InterruptedException {  String path = "c:\\b.txt";  *readFile*(path);  }  **private** **static** **void** readFile(String path) **throws** IOException,  InterruptedException {  FileInputStream fis = **new** FileInputStream(path);  **byte**[] byt = **new** **byte**[1024];  **int** len = fis.read(byt);  System.*out*.println(**new** String(byt, 0, len));  // 让程序睡眠，无法执行到close方法。  Thread.*sleep*(1000 \* 10);  fis.close();  }  } |

在执行该程序的同时我们尝试去删除b.txt文件。如果在该程序没有睡醒的话，我们是无法删除b.txt 文件的。因为b.txt还被该程序占用着，这是很严重的问题，所以一定要关闭流。

目前我们是抛出处理，一旦出现了异常，close就没有执行，也就没有释放资源。那么为了保证close的执行该如何处理呢。

那么就需要使用try{} catch(){}finally{}语句。try中放入可能出现异常的语句，catch是捕获异常对象，fianlly是一定要执行的代码

|  |
| --- |
| **public** **class** IoTest4 {  **public** **static** **void** main(String[] args) **throws** IOException,  InterruptedException {  String path = "c:\\b.txt";  *readFile*(path);  }  **private** **static** **void** readFile(String path) {  FileInputStream fis = **null**;  **try** {  fis = **new** FileInputStream(path);  **byte**[] byt = **new** **byte**[1024];  **int** len = fis.read(byt);  System.*out*.println(**new** String(byt, 0, len));  } **catch** (IOException e) {  // 抛出运行时异常  **throw** **new** RuntimeException(e);  } **finally** {  // 把close方法放入finally中保证一定会执行  // 先判断是否空指针  **if** (fis != **null**) {  **try** {  fis.close();  } **catch** (Exception e) {  **throw** **new** RuntimeException(e);  }  }  }  }  } |

文件拷贝的异常处理：

|  |
| --- |
| **public** **static** **void** copyFile(String srcPath, String destPath) {  FileInputStream fis = **null**;  FileOutputStream fos = **null**;  **try** {  fis = **new** FileInputStream(srcPath);  fos = **new** FileOutputStream(destPath);  **byte**[] byt = **new** **byte**[1024 \* 1024];  **int** len = 0;  **while** ((len = fis.read(byt)) != -1) {  fos.write(byt, 0, len);  }  } **catch** (IOException e) {  **throw** **new** RuntimeException(e);  } **finally** {  **if** (fis != **null**) {  **try** {  fis.close();  } **catch** (IOException e) {  **throw** **new** RuntimeException(e);  }  }  **if** (fos != **null**) {  **try** {  fos.close();  } **catch** (IOException e) {  **throw** **new** RuntimeException(e);  }  }  }  } |

注意：

在最后的close代码中可能会有问题，两个close，如果第一个close方法出现了异常，并抛出了运行时异常，那么程序还是停止了。下面的close方法就没有执行到。

那么为了保证close的执行，将第二个放到fianlly中即可。

|  |
| --- |
| **public** **static** **void** copyFile(String srcPath, String destPath) {  FileInputStream fis = **null**;  FileOutputStream fos = **null**;  **try** {  fis = **new** FileInputStream(srcPath);  fos = **new** FileOutputStream(destPath);  **byte**[] byt = **new** **byte**[1024 \* 1024];  **int** len = 0;  **while** ((len = fis.read(byt)) != -1) {  fos.write(byt, 0, len);  }  } **catch** (IOException e) {  **throw** **new** RuntimeException(e);  } **finally** {  **try** {  **if** (fis != **null**) {  fis.close();  }  } **catch** (IOException e) {  **throw** **new** RuntimeException(e);  } **finally** {  **if** (fos != **null**) {  **try** {  fos.close();  } **catch** (IOException e) {  **throw** **new** RuntimeException(e);  }  }  }  }  } |

## 字节缓冲流

### 缓冲流

上述程序中我们为了提高流的使用效率,自定义了字节数组,作为缓冲区.Java其实提供了专门的字节流缓冲来提高效率.

BufferedInputStream和BufferedOutputStream

BufferedOutputStream和BufferedOutputStream类可以通过减少读写次数来提高输入和输出的速度。它们内部有一个缓冲区，用来提高处理效率。查看API文档，发现可以指定缓冲区的大小。其实内部也是封装了字节数组。没有指定缓冲区大小，默认的字节是8192。

显然缓冲区输入流和缓冲区输出流要配合使用。首先缓冲区输入流会将读取到的数据读入缓冲区，当缓冲区满时，或者调用flush方法，缓冲输出流会将数据写出。

注意：当然使用缓冲流来进行提高效率时，对于小文件可能看不到性能的提升。但是文件稍微大一些的话，就可以看到实质的性能提升了。

|  |
| --- |
| **public** **class** IoTest5 {  **public** **static** **void** main(String[] args) **throws** IOException {  String srcPath = "c:\\a.mp3";  String destPath = "d:\\copy.mp3";  *copyFile*(srcPath, destPath);  }  **public** **static** **void** copyFile(String srcPath, String destPath)  **throws** IOException {  // 打开输入流，输出流  FileInputStream fis = **new** FileInputStream(srcPath);  FileOutputStream fos = **new** FileOutputStream(destPath);  // 使用缓冲流  BufferedInputStream bis = **new** BufferedInputStream(fis);  BufferedOutputStream bos = **new** BufferedOutputStream(fos);  // 读取和写入信息  **int** len = 0;  **while** ((len = bis.read()) != -1) {  bos.write(len);  }  // 关闭流  bis.close();  bos.close();  }  } |

# 字符流

计算机并不区分二进制文件与文本文件。所有的文件都是以二进制形式来存储的，因此，从本质上说，所有的文件都是二进制文件。所以字符流是建立在字节流之上的，它能够提供字符层次的编码和解码。例如，在写入一个字符时，Java虚拟机会将字符转为文件指定的编码（默认是系统默认编码），在读取字符时，再将文件指定的编码转化为字符。

**常见的码表如下：**

ASCII： 美国标准信息交换码。用一个字节的7位可以表示。

ISO8859-1： 拉丁码表。欧洲码表，用一个字节的8位表示。又称Latin-1(拉丁编码)或“西欧语言”。ASCII码是包含的仅仅是英文字母，并且没有完全占满256个编码位置，所以它以ASCII为基础，在空置的0xA0-0xFF的范围内，加入192个字母及符号，

藉以供使用变音符号的拉丁字母语言使用。从而支持德文，法文等。因而它依然是一个单字节编码，只是比ASCII更全面。

GB2312： 英文占一个字节，中文占两个字节.中国的中文编码表。

GBK： 中国的中文编码表升级，融合了更多的中文文字符号。

Unicode： 国际标准码规范，融合了多种文字。所有文字都用两个字节来表示，Java语言使用的就是unicode。

UTF-8： 最多用三个字节来表示一个字符。

（我们以后接触最多的是iso8859-1、gbk、utf-8）

查看上述码表后，很显然中文的‘中’在iso8859-1中是没有对映的编码的。或者一个字符在2中码表中对应的编码不同，例如有一些字在不同的编码中是有交集的，例如bjg5 和gbk 中的汉字简体和繁体可能是一样的，就是有交集，但是在各自码表中的数字不一样。

例如

使用gbk 将中文保存在计算机中，

中 国

对映 100 200 如果使用big5 打开

可能 ？ ...

不同的编码对映的是不一样的。

很显然，我们使用什么样的编码写数据，就需要使用什么样的编码来对数据。

ISO8859-1：一个字节

GBK： 两个字节包含了英文字符和扩展的中文 ISO8859-1+中文字符

UTF-8 万国码，推行的。是1~3个字节不等长。英文存的是1个字节，中文存的是3个字节，是为了节省空间。

那么我们之前学习的流称之为字节流，以字节为单位进行操作之情的操作全是英文，如果想要操作中文呢？

测试：将指定位置的文件通过字节流读取到控制台

|  |
| --- |
| **public** **class** TestIo {  **public** **static** **void** main(String[] args) **throws** IOException {  String path = "c:\\a.txt";  *writFileTest*();  *readFileByInputStream*(path);  }  **private** **static** **void** readFileByInputStream(String path) **throws** IOException {  FileInputStream fis = **new** FileInputStream(path);  **int** len = 0;  **while** ((len = fis.read()) != -1) {  System.*out*.print((**char**) len);  }  }  **private** **static** **void** writFileTest() **throws** FileNotFoundException,  IOException {  // 创建文件对象  File file = **new** File("c:\\a.txt");  // 创建文件输出流  FileOutputStream fos = **new** FileOutputStream(file);  fos.write("中国".getBytes());  fos.close();  }  } |

发现控制台输出的信息：

???ú 是这样的东西，打开a.txt 文本发现汉字”中国”确实写入成功。

那么说明使用字节流处理中文有问题。

仔细分析，我们的FileInputStream输入流的read() 一次是读一个字节的，返回的是一个int显然进行了自动类型提升。那么我们来验证一下“中国”对应的字节是什么

使用："中国".getBytes() 即可得到字符串对应的字节数组。是[-42, -48, -71, -6]

同样，将read方法返回值直接强转为byte ，发现结果也是-42, -48, -71, -6 。

代码：

|  |
| --- |
| **public** **class** TestIo {  **public** **static** **void** main(String[] args) **throws** IOException {  String path = "c:\\a.txt";  *writFileTest*();  *readFileByInputStream*(path);  //查看中国对应的编码  System.*out*.println(Arrays.*toString*("中国".getBytes()));  }  **private** **static** **void** readFileByInputStream(String path) **throws** IOException {  FileInputStream fis = **new** FileInputStream(path);  **int** len = 0;  **while** ((len = fis.read()) != -1) {  System.*out*.println((**byte**)len);  }  }  **private** **static** **void** writFileTest() **throws** FileNotFoundException,  IOException {  // 创建文件对象  File file = **new** File("c:\\a.txt");  // 创建文件输出流  FileOutputStream fos = **new** FileOutputStream(file);  fos.write("中国\r\n".getBytes());  fos.close();  }  } |

那么中国 对应的是-42, -48, -71, -6是4个字节。 那就是一个中文占2个字节，（这个和编码是有关系的）

很显然，我们的中文就不能够再一个字节一个字节的读了。所以字节流处理字符信息时并不方便那么就出现了字符流。

字节流是 字符流是以字符为单位。

体验字符流：

|  |
| --- |
| **public** **static** **void** main(String[] args) **throws** IOException {    String path = "c:\\a.txt";  *readFileByReader*(path);  }  **private** **static** **void** readFileByReader(String path) **throws** IOException {  FileReader fr = **new** FileReader(path);  **int** len = 0;  **while** ((len = fr.read()) != -1) {  System.*out*.print((**char**) len);  }  } |

总结：字符流就是：字节流 + 编码表，为了更便于操作文字数据。字符流的抽象基类：

Reader ， Writer。

由这些类派生出来的子类名称都是以其父类名作为子类名的后缀，如FileReader、FileWriter。

## Reader

方法：

1，int read()：

读取一个字符。返回的是读到的那个字符。如果读到流的末尾，返回-1.

2，int read(char[])：

将读到的字符存入指定的数组中，返回的是读到的字符个数，也就是往数组里装的元素的个数。如果读到流的末尾，返回-1.

3，close()

读取字符其实用的是window系统的功能，就希望使用完毕后，进行资源的释放

由于Reader也是抽象类，所以想要使用字符输入流需要使用Reader的实现类。查看API文档。找到了FileReader。

1，用于读取文本文件的流对象。

2，用于关联文本文件。

构造函数：在读取流对象初始化的时候，必须要指定一个被读取的文件。

如果该文件不存在会发生FileNotFoundException.

|  |
| --- |
| **public** **class** IoTest1\_Reader {  **public** **static** **void** main(String[] args) **throws** Exception {  String path = "c:/a.txt";  // readFileByInputStream(path);  *readFileByReader*(path);  }  /\*\*  \* 使用字节流读取文件内容  \*  \* **@param** path  \*/  **public** **static** **void** readFileByInputStream(String path) **throws** Exception {  InputStream in = **new** FileInputStream(path);  **int** len = 0;  **while** ((len = in.read()) != -1) {  System.*out*.print((**char**) len);  }  in.close();  }  /\*\*  \* 使用字符流读取文件内容  \*/  **public** **static** **void** readFileByReader(String path) **throws** Exception {  Reader reader = **new** FileReader(path);  **int** len = 0;  **while** ((len = reader.read()) != -1) {  System.*out*.print((**char**) len);  }  reader.close();  }  } |

## Writer

Writer中的常见的方法：

1，write(ch): 将一个字符写入到流中。

2，write(char[]): 将一个字符数组写入到流中。

3，write(String): 将一个字符串写入到流中。

4，flush():刷新流，将流中的数据刷新到目的地中，流还存在。

5，close():关闭资源：在关闭前会先调用flush()，刷新流中的数据去目的地。然流关闭。

发现基本方法和OutputStream 类似，有write方法，功能更多一些。可以接收字符串。

同样道理Writer是抽象类无法创建对象。查阅API文档，找到了Writer的子类FileWriter

1：将文本数据存储到一个文件中。

|  |
| --- |
| **public** **class** IoTest2\_Writer {  **public** **static** **void** main(String[] args) **throws** Exception {  String path = "c:/ab.txt";  *writeToFile*(path);  }  /\*\*  \* 写指定数据到指定文件中  \*  \*/  **public** **static** **void** writeToFile(String path) **throws** Exception {  Writer writer = **new** FileWriter(path);  writer.write('中');  writer.write("世界".toCharArray());  writer.write("中国");  writer.close();  }  } |

2：追加文件：

默认的FileWriter方法新值会覆盖旧值，想要实现追加功能需要

使用如下构造函数创建输出流 append值为true即可。

FileWriter(String fileName, boolean append)

FileWriter(File file, boolean append)

3:flush方法

如果使用字符输出流,没有调用close方法,会发生什么?

|  |
| --- |
| **private** **static** **void** writeFileByWriter(File file) **throws** IOException {  FileWriter fw = **new** FileWriter(file);  fw.write('新');  fw.flush();  fw.write("中国".toCharArray());  fw.write("世界你好!!!".toCharArray());  fw.write("明天");  // 关闭流资源  //fw.close();  } |

程序执行完毕打开文件,发现没有内容写入.原来需要使用flush方法. 刷新该流的缓冲。

为什么只要指定claose方法就不用再flush方法,因为close也调用了flush方法.

## 字符流拷贝文件

一个文本文件中有中文有英文字母，有数字。想要把这个文件拷贝到别的目录中。

我们可以使用字节流进行拷贝，使用字符流呢？肯定也是可以的。

### 字符流拷贝文件实现1

|  |
| --- |
| **public** **static** **void** main(String[] args) **throws** Exception {  String path1 = "c:/a.txt";  String path2 = "c:/b.txt";  *copyFile(path1, path2);*  }  /\*\*  \* 使用字符流拷贝文件  \*/  **public** **static** **void** copyFile(String path1, String path2) **throws** Exception {  Reader reader = **new** FileReader(path1);  Writer writer = **new** FileWriter(path2);  **int** ch = -1;  **while** ((ch = reader.read()) != -1) {  writer.write(ch);  }  reader.close();  writer.close();  } |

但是这个一次读一个字符就写一个字符，效率不高。把读到的字符放到字符数组中，再一次性的写出。

### 字符流拷贝文件实现2

|  |
| --- |
| **public** **static** **void** main(String[] args) **throws** Exception {  String path1 = "c:/a.txt";  String path2 = "c:/b.txt";  *copyFile(path1, path2);*  }  **public** **static** **void** copyFile3(String path1, String path2) **throws** Exception {  Reader reader = **new** FileReader(path1);  Writer writer = **new** FileWriter(path2);  **int** ch = -1;  **char** [] arr=**new** **char**[1024];  **while** ((ch = reader.read(arr)) != -1) {  writer.write(arr,0,ch);  }  reader.close();  writer.close();  } |

字节流可以拷贝视频和音频等文件，那么字符流可以拷贝这些吗？

经过验证拷贝图片是不行的。发现丢失了信息，为什么呢？

计算机中的所有信息都是以二进制形式进行的存储（1010）图片中的也都是二进制

在读取文件的时候字符流自动对这些二进制按照码表进行了编码处理，但是图片本来就是二进制文件，不需要进行编码。有一些巧合在码表中有对应，就可以处理，并不是所有的二进制都可以找到对应的。信息就会丢失。所以字符流只能拷贝以字符为单位的文本文件

（以ASCII码为例是127个，并不是所有的二进制都可以找到对应的ASCII，有些对不上的，就会丢失信息。）

## 字符流的异常处理

|  |
| --- |
| **public** **static** **void** main(String[] args) **throws** Exception {  String path1 = "c:/a.txt";  String path2 = "c:/b.txt";  *copyFile2(path1, path2);*  }  /\*\*  \* 使用字符流拷贝文件，有完善的异常处理  \*/  **public** **static** **void** copyFile2(String path1, String path2) {  Reader reader = **null**;  Writer writer = **null**;  **try** {  // 打开流  reader = **new** FileReader(path1);  writer = **new** FileWriter(path2);  // 进行拷贝  **int** ch = -1;  **while** ((ch = reader.read()) != -1) {  writer.write(ch);  }  } **catch** (Exception e) {  **throw** **new** RuntimeException(e);  } **finally** {  // 关闭流，注意一定要能执行到close()方法，所以都要放到finally代码块中  **try** {  **if** (reader != **null**) {  reader.close();  }  } **catch** (Exception e) {  **throw** **new** RuntimeException(e);  } **finally** {  **try** {  **if** (writer != **null**) {  writer.close();  }  } **catch** (Exception e) {  **throw** **new** RuntimeException(e);  }  }  }  } |

## 字符流的缓冲区

查看Reader 发现Reader,操作的是字符,我们就不需要进行编码解码操作,由字符流读到二进制,自动进行解码得到字符,写入字符自动编码成二进制.

Reader有一个子类BufferedReader。子类继承父类显然子类可以重写父类的方法,也可以增加自己的新方法。例如一次读一行就是常用的操作.那么BufferedReader 类就提供了这个方法,可以查看readLine()方法具备 一次读取一个文本行的功能。很显然,该子类可以对功能进行增强。

体验BufferedReader

|  |
| --- |
| **public** **class** IoTest\_BufferedReader {  **public** **static** **void** main(String[] args) **throws** IOException {  *readFile*("c:\\a.txt");  }  **private** **static** **void** readFile(String path) **throws** IOException {  Reader read = **new** FileReader(path);  BufferedReader br = **new** BufferedReader(read);    String line = **null**;  **while** ((line = br.readLine()) != **null**) {  System.*out*.println(line);  }  }  } |

注意：

在使用缓冲区对象时，要明确，缓冲的存在是为了增强流的功能而存在，所以在建立缓冲区对象时，要先有流对象存在.

缓冲区的出现提高了对流的操作效率。原理：其实就是将数组进行封装。

使用字符流缓冲区拷贝文本文件.

|  |
| --- |
| **public** **class** Demo7 {  **public** **static** **void** main(String[] args) **throws** IOException {  // 关联源文件  File srcFile = **new** File("c:\\linux大纲.txt");  // 关联目标文件  File destFile = **new** File("d:\\linux大纲.txt");  // 实现拷贝  *copyFile*(srcFile, destFile);  }  **private** **static** **void** copyFile(File srcFile, File destFile)  **throws** IOException {  // 创建字符输入流  FileReader fr = **new** FileReader(srcFile);  // 创建字符输出流  FileWriter fw = **new** FileWriter(destFile);  // 字符输入流的缓冲流  BufferedReader br = **new** BufferedReader(fr);  // 字符输出流的缓冲流  BufferedWriter bw = **new** BufferedWriter(fw);  String line = **null**;  // 一次读取一行  **while** ((line = br.readLine()) != **null**) {  // 一次写出一行.  bw.write(line);  // 刷新缓冲  bw.flush();  // 进行换行,由于readLine方法默认没有换行.需要手动换行  bw.newLine();  }  // 关闭流  br.close();  bw.close();  }  } |

## 装饰器模式

需求：想要在读取的文件的每一行添加行号。

|  |
| --- |
| **public** **class** IoTest7\_BufferedReader {  **public** **static** **void** main(String[] args) **throws** IOException {  *readFile*("c:\\a.txt");  }  **private** **static** **void** readFile(String path) **throws** IOException {  Reader read = **new** FileReader(path);  BufferedReader br = **new** BufferedReader(read);  **int** count = 0;  String line = **null**;  **while** ((line = br.readLine()) != **null**) {  count++;  System.*out*.println(count+":"+line);  }  }  } |

很容易的就可以实现。如果每次使用BufferedReader 输出时都需要显示行号呢? 每次都加? 很显然,我们的BufferedReader继承了Reader 对父类进行了功能的增强,那么我们也可以继承BufferedReader 重写该类的readLine方法,进行功能的增强.

|  |
| --- |
| **public** **class** IoTest\_BufferedReader {  **public** **static** **void** main(String[] args) **throws** IOException {  *readFile*("c:\\a.txt");  }  **private** **static** **void** readFile(String path) **throws** IOException {  Reader read = **new** FileReader(path);  BufferedReader br = **new** MyBufferedReader(read);  String line = **null**;  **while** ((line = br.readLine()) != **null**) {  System.*out*.println(line);  }  }  }  **class** MyBufferedReader **extends** BufferedReader {  **public** MyBufferedReader(Reader read) {  **super**(read);  }  **int** count;  @Override  **public** String readLine() **throws** IOException {  String line = **super**.readLine();  **if** (line != **null**) {  count++;  **return** count + ":" + line;  } **else** {  **return** **null**;  }  }  } |

需求:

要在输出的一行前加上引号

可以再定义一个BufferedReader的子类,继承BufferedReader增强功能.

|  |
| --- |
| **public** **class** IoTest\_BufferedReader {  **public** **static** **void** main(String[] args) **throws** IOException {  *readFile*("c:\\a.txt");  }  **private** **static** **void** readFile(String path) **throws** IOException {  Reader read = **new** FileReader(path);  BufferedReader br = **new** MyQutoBufferedReader(read);  **int** count = 0;  String line = **null**;  **while** ((line = br.readLine()) != **null**) {  System.*out*.println(line);  count++;  }  }  }  // quotation 引号  **class** MyQutoBufferedReader **extends** BufferedReader {  **public** MyQutoBufferedReader(Reader reader) {  **super**(reader);  }  **public** String readLine() **throws** IOException {  String line = **super**.readLine();  **if** (line != **null**) {  **return** "\"" + line + "\"";  } **else** {  **return** **null**;  }  }  } |

需求三:

既想要显示行号又想要显示引号

发现,就需要再定义子类,发现这样比较麻烦，代码臃肿.而且代码重复.

可以换一种方式.如下:

其实就是一个新类要对原有类进行功能增强.

1. 在增强类中维护一个被增强的父类引用变量

2. 在增强类的构造函数中初始化1中的变量

3. 创建需要增强的方法，在刚方法中调用被被增强类的方法，并加以增强。

|  |
| --- |
| **public** **class** IoTest\_BufferedReader {  **public** **static** **void** main(String[] args) **throws** IOException {  *readFile*("c:\\a.txt");  }  **private** **static** **void** readFile(String path) **throws** IOException {  Reader read = **new** FileReader(path);  BufferedReader bufferedReader = **new** BufferedReader(read);  BufferedReader br = **new** MyQutoBufferedReader2(bufferedReader);  br = **new** MyLineBufferedReader2(br);  String line = **null**;  **while** ((line = br.readLine()) != **null**) {  System.*out*.println(line);  }  }  }  // quotation 引号  **class** MyQutoBufferedReader2 **extends** BufferedReader {  **private** BufferedReader bufferedReader;  **public** MyQutoBufferedReader2(BufferedReader bufferedReader) {  **super**(bufferedReader);  **this**.bufferedReader = bufferedReader;  }  **public** String readLine() **throws** IOException {  String line = **super**.readLine();  **if** (line != **null**) {  **return** "\"" + line + "\"";  } **else** {  **return** **null**;  }  }  }  **class** MyLineBufferedReader2 **extends** BufferedReader {  **private** BufferedReader bufferedReader;  **public** MyLineBufferedReader2(BufferedReader bufferedReader) {  **super**(bufferedReader);  **this**.bufferedReader = bufferedReader;  }  **int** count;  @Override  **public** String readLine() **throws** IOException {  String line = **super**.readLine();  **if** (line != **null**) {  count++;  **return** count + ":" + line;  } **else** {  **return** **null**;  }  }  } |

这就是装饰器模式

装饰器模式：

使用分层对象来动态透明的向单个对象中添加责任（功能）。

装饰器指定包装在最初的对象周围的所有对象都具有相同的基本接口。

某些对象是可装饰的，可以通过将其他类包装在这个可装饰对象的四周，来将功能分层。

装饰器必须具有和他所装饰的对象相同的接口。

JavaIO中的应用：

Java I/O类库需要多种不同的功能组合，所以使用了装饰器模式。

FilterXxx类是JavaIO提供的装饰器基类，即我们要想实现一个新的装饰器，就要继承这些类。

装饰器与继承：

问题：

修饰模式做的增强功能按照继承的特点也是可以实现的，为什么还要提出修饰设计模式呢？

继承实现的增强类和修饰模式实现的增强类有何区别？

继承实现的增强类：

优点：代码结构清晰，而且实现简单

缺点：对于每一个的需要增强的类都要创建具体的子类来帮助其增强，这样会导致继承体系过于庞大。

修饰模式实现的增强类：

优点：内部可以通过多态技术对多个需要增强的类进行增强

缺点：需要内部通过多态技术维护需要增强的类的实例。进而使得代码稍微复杂。

# 其他流

## 序列流

也称为合并流。

### SequenceInputStream

序列流，对多个流进行合并。

SequenceInputStream 表示其他输入流的逻辑串联。它从输入流的有序集合开始，并从第一个输入流开始读取，直到到达文件末尾，接着从第二个输入流读取，依次类推，直到到达包含的最后一个输入流的文件末尾为止。

注意:

|  |
| --- |
| 构造函数  SequenceInputStream(InputStream s1, InputStream s2)  SequenceInputStream(InputStream s1, InputStream s2) |

合并两个流

使用构造函数SequenceInputStream(InputStream s1, InputStream s2)

|  |
| --- |
| **private** **static** **void** testSequenceInputStream() **throws** IOException {  FileInputStream fis1 = **new** FileInputStream("c:\\a.txt");  FileInputStream fis2 = **new** FileInputStream("c:\\b.txt");  SequenceInputStream s1 = **new** SequenceInputStream(fis1, fis2);  **int** len = 0;  **byte**[] byt = **new** **byte**[1024];  FileOutputStream fos = **new** FileOutputStream("c:\\z.txt");  **while** ((len = s1.read(byt)) != -1) {  fos.write(byt, 0, len);  }  s1.close();  } |

合并多个流:

|  |
| --- |
| **public** **static** **void** testSequenceInputStream() **throws** Exception {  InputStream in1 = **new** FileInputStream("c:/a.txt");  InputStream in2 = **new** FileInputStream("c:/b.txt");  InputStream in3 = **new** FileInputStream("c:/c.txt");  LinkedHashSet<InputStream> set = **new** LinkedHashSet<InputStream>();  set.add(in1);  set.add(in2);  set.add(in3);  **final** Iterator<InputStream> iter = set.iterator();  SequenceInputStream sin = **new** SequenceInputStream(  **new** Enumeration<InputStream>() {  @Override  **public** **boolean** hasMoreElements() {  **return** iter.hasNext();  }  @Override  **public** InputStream nextElement() {  **return** iter.next();  }  });  FileOutputStream out = **new** FileOutputStream("c:/z.txt");  **for** (**int** b = -1; (b = sin.read()) != -1;) {  out.write(b);  }  sin.close();  out.close();  } |

案例:将map3歌曲文件进行切割拷贝,并合并.

|  |
| --- |
| **public** **class** Demo2 {  **public** **static** **void** main(String[] args) **throws** IOException {    *split*(**new** File("c:\\a.mp3"), 10, **new** File("c:\\"));  System.*out*.println("切割完毕");  LinkedHashSet<InputStream> hs = **new** LinkedHashSet<InputStream>();  hs.add(**new** FileInputStream(**new** File("c:\\part.1.mp3")));  hs.add(**new** FileInputStream(**new** File("c:\\part.2.mp3")));  hs.add(**new** FileInputStream(**new** File("c:\\part.3.mp3")));  hs.add(**new** FileInputStream(**new** File("c:\\part.4.mp3")));  *merage*(hs, **new** File("c:\\merage.mp3"));  System.*out*.println("合并完毕");  }  **private** **static** **void** merage(LinkedHashSet<InputStream> hs, File dest)  **throws** IOException {  **final** Iterator<InputStream> it = hs.iterator();  FileOutputStream fos = **new** FileOutputStream(dest);  SequenceInputStream seq = **new** SequenceInputStream(  **new** Enumeration<InputStream>() {  @Override  **public** **boolean** hasMoreElements() {  **return** it.hasNext();  }  @Override  **public** InputStream nextElement() {  **return** it.next();  }  });  **byte**[] byt = **new** **byte**[1024 \* 1024];  **int** len = 0;  **while** ((len = seq.read(byt)) != -1) {  fos.write(byt, 0, len);  }  seq.close();  fos.close();  }  // 1. 切割文件  /\*  \* 切割文件,切割份数, 切割后保存路径  \*/  **private** **static** **void** split(File src, **int** count, File dir) **throws** IOException {  FileInputStream fis = **new** FileInputStream(src);  FileOutputStream fos = **null**;  **byte**[] byt = **new** **byte**[1024 \* 1024];  **int** len = 0;  **for** (**int** i = 1; i <= count; i++) {  len = fis.read(byt);  **if** (len != -1) {  fos = **new** FileOutputStream(dir + "part." + i + ".mp3");  fos.write(byt, 0, len);  }  // fos.close();  }  fis.close();  }  } |

## 对象的序列化

当创建对象时,程序运行时它就会存在,但是程序停止时,对象也就消失了.但是如果希望对象在程序不运行的情况下仍能存在并保存其信息，将会非常有用，对象将被重建并且拥有与程序上次运行时拥有的信息相同。可以使用对象的序列化。

对象的序列化： 将内存中的对象直接写入到文件设备中

对象的反序列化： 将文件设备中持久化的数据转换为内存对象

基本的序列化由两个方法产生：一个方法用于序列化对象并将它们写入一个流，另一个方法用于读取流并反序列化对象。

|  |
| --- |
| ObjectOutput  writeObject(Object obj)  将对象写入底层存储或流。  ObjectInput  readObject()  读取并返回对象。 |

### ObjectOutputStream

### ObjectInputStream

由于上述ObjectOutput和ObjectInput是接口，所以需要使用具体实现类。

|  |
| --- |
| ObjectOutput  ObjectOutputStream被写入的对象必须实现一个接口:Serializable  否则会抛出：NotSerializableException  ObjectInput  ObjectInputStream 该方法抛出异常：ClassNotFountException |

ObjectOutputStream和ObjectInputStream 对象分别需要字节输出流和字节输入流对象来构建对象。也就是这两个流对象需要操作已有对象将对象进行本地持久化存储。

案例：

序列化和反序列化Cat对象。

|  |
| --- |
| **public** **class** Demo3 {  **public** **static** **void** main(String[] args) **throws** IOException,  ClassNotFoundException {  Cat cat = **new** Cat("tom", 3);  FileOutputStream fos = **new** FileOutputStream(**new** File("c:\\Cat.txt"));  ObjectOutputStream oos = **new** ObjectOutputStream(fos);  oos.writeObject(cat);  System.*out*.println(cat);  oos.close();  // 反序列化  FileInputStream fis = **new** FileInputStream(**new** File("c:\\Cat.txt"));  ObjectInputStream ois = **new** ObjectInputStream(fis);  Object readObject = ois.readObject();  Cat cat2 = (Cat) readObject;  System.*out*.println(cat2);  fis.close();  }  **class** Cat **implements** Serializable {  **public** String name;  **public** **int** age;  **public** Cat() {  }  **public** Cat(String name, **int** age) {  **this**.name = name;  **this**.age = age;  }  @Override  **public** String toString() {  **return** "Cat [name=" + name + ", age=" + age + "]";  }  } |

例子关键点：

1. 声明Cat类实现了Serializable接口。是一个标示器，没有要实现的方法。
2. 新建Cat对象。
3. 新建字节流对象（FileOutputStream）进序列化对象保存在本地文件中。
4. 新建ObjectOutputStream对象，调用writeObject方法序列化Cat对象。
5. writeObject方法会执行两个工作：序列化对象，然后将序列化的对象写入文件中。
6. 反序列化就是调用ObjectInputStream的readObject()方法。
7. 异常处理和流的关闭动作要执行。

### **Serializable**：

类通过实现 java.io.Serializable 接口以启用其序列化功能。未实现此接口的类将无法使其任何状态序列化或反序列化。可序列化类的所有子类型本身都是可序列化的。序列化接口没有方法或字段，仅用于标识可序列化的语义。

所以需要被序列化的类必须是实现Serializable接口，该接口中没有描述任何的属性和方法，称之为标记接口。

如果对象没有实现接口Serializable，在进行序列化时会抛出：NotSerializableException 异常。

注意：

保存一个对象的真正含义是什么？如果对象的实例变量都是基本数据类型，那么就非常简单。但是如果实例变量是包含对象的引用，会怎么样？保存的会是什么？很显然在Java中保存引用变量的实际值没有任何意义，因为Java引用的值是通过JVM的单一实例的上下文中才有意义。通过序列化后，尝试在JVM的另一个实例中恢复对象，是没有用处的。

如下：

首先建立一个Dog对象，也建立了一个Collar对象。Dog中包含了一个Collar(项圈)

现在想要保存Dog对象，但是Dog中有一个Collar，意味着保存Dog时也应该保存Collar。假如Collar也包含了其他对象的引用，那么会发生什么？意味着保存一个Dog对象需要清楚的知道Dog对象的内部结构。会是一件很麻烦的事情。

Java的序列化机制可以解决该类问题，当序列化一个对象时，Java的序列化机制会负责保存对象的所有关联的对象（就是对象图），反序列化时，也会恢复所有的相关内容。本例中：如果序列化Dog会自动序列化Collar。但是，只有实现了Serializable接口的类才可以序列化。如果只是Dog实现了该接口，而Collar没有实现该接口。会发生什么？

Dog类和Collar类

|  |
| --- |
| **import** java.io.Serializable;  **public** **class** Dog **implements** Serializable {  **private** Collar collar;  **private** String name;  **public** Dog(Collar collar, String name) {  **this**.collar = collar;  **this**.name = name;  }  **public** Collar getCollar() {  **return** collar;  }  }  **class** Collar {  **private** **int** size;  **public** **int** getSize() {  **return** size;  }  **public** Collar(**int** size) {  **this**.size = size;  }  } |

序列化

|  |
| --- |
| **import** java.io.File;  **import** java.io.FileOutputStream;  **import** java.io.IOException;  **import** java.io.ObjectOutputStream;  **public** **class** Demo4 {  **public** **static** **void** main(String[] args) **throws** IOException {  Collar coll = **new** Collar(10);  Dog dog = **new** Dog(coll, "旺财");  FileOutputStream fis = **new** FileOutputStream(**new** File("c:\\dog.txt"));  ObjectOutputStream os = **new** ObjectOutputStream(fis);  os.writeObject(dog);  }  } |

执行程序，出现了运行时异常。

|  |
| --- |
| Exception in thread "main" java.io.NotSerializableException: Collar |

所以我们也必须将Dog中使用的Collar序列化。但是如果我们无法访问Collar的源代码，或者无法使Collar可序列化，如何处理？

两种解决方法：

一：继承Collar类，使子类可序列化

但是：如果Collar是final类，就无法继承了。并且，如果Collar引用了其他非序列化对象，也无法解决该问题。

transient

此时就可以使用transient修饰符，可以将Dog类中的成员变量标识为transient

那么在序列化Dog对象时，序列化就会跳过Collar。

|  |
| --- |
| **public** **class** Demo4 {  **public** **static** **void** main(String[] args) **throws** IOException,  ClassNotFoundException {  Collar coll = **new** Collar(10);  Dog dog = **new** Dog(coll, "旺财");  System.*out*.println(dog.getCollar().getSize());  FileOutputStream fis = **new** FileOutputStream(**new** File("c:\\dog.txt"));  ObjectOutputStream os = **new** ObjectOutputStream(fis);  os.writeObject(dog);  // 反序列化  FileInputStream fos = **new** FileInputStream(**new** File("c:\\dog.txt"));  ObjectInputStream ois = **new** ObjectInputStream(fos);  Object readObject = ois.readObject();  Dog dog2 = (Dog) readObject;  // Collar未序列化。  dog2.getCollar().getSize();  }  } |

这样我们具有一个序列化的Dog和非序列化的Collar。

此时反序列化Dog后，访问Collar，就会出现运行时异常

|  |
| --- |
| 10  Exception in thread "main" java.lang.NullPointerException |

注意：序列化不适用于静态变量，因为静态变量并不属于对象的实例变量的一部分。静态变量随着类的加载而加载，是类变量。由于序列化只适用于对象。

基本数据类型可以被序列化

|  |
| --- |
| **public** **class** Demo5 {  **public** **static** **void** main(String[] args) **throws** IOException {  // 创建序列化流对象  FileOutputStream fis = **new** FileOutputStream(**new** File("c:\\basic.txt"));  ObjectOutputStream os = **new** ObjectOutputStream(fis);  // 序列化基本数据类型  os.writeDouble(3.14);  os.writeBoolean(**true**);  os.writeInt(100);  os.writeInt(200);  // 关闭流  os.close();  // 反序列化  FileInputStream fos = **new** FileInputStream(**new** File("c:\\basic.txt"));  ObjectInputStream ois = **new** ObjectInputStream(fos);  System.*out*.println(ois.readDouble());  System.*out*.println(ois.readBoolean());  System.*out*.println(ois.readInt());  System.*out*.println(ois.readInt());  fos.close();  }  } |

serialVersionUID

用于给类指定一个UID。该UID是通过类中的可序列化成员的数字签名运算出来的一个long型的值。

只要是这些成员没有变化，那么该值每次运算都一样。

该值用于判断被序列化的对象和类文件是否兼容。

如果被序列化的对象需要被不同的类版本所兼容。可以在类中自定义UID。

定义方式：static final long serialVersionUID = 42L;

## Properties.

可以和流相关联的集合对象Properties.

Map

|--Hashtable

|--Properties

Properties:该集合不需要泛型，因为该集合中的键值对都是String类型。

1，存入键值对：setProperty(key,value);

2，获取指定键对应的值：value getProperty(key);

3，获取集合中所有键元素：

Enumeration propertyNames();

在jdk1.6版本给该类提供一个新的方法。

Set<String> stringPropertyNames();

4，列出该集合中的所有键值对，可以通过参数打印流指定列出到的目的地。

list(PrintStream);

list(PrintWriter);

例：list(System.out):将集合中的键值对打印到控制台。

list(new PrintStream("prop.txt")):将集合中的键值对存储到prop.txt文件中。

5，可以将流中的规则数据加载进行集合，并称为键值对。

load(InputStream):

jdk1.6版本。提供了新的方法。

load(Reader):

注意：流中的数据要是"键=值" 的规则数据。

6，可以将集合中的数据进行指定目的的存储。

store(OutputStram,String comment)方法。

jdk1.6版本。提供了新的方法。

store(Writer ,String comment):

使用该方法存储时，会带着当时存储的时间。

注意：

Properties只加载key=value这样的键值对，与文件名无关，注释使用#

练习：记录一个程序运行的次数，当满足指定次数时，该程序就不可以再继续运行了。

通常可用于软件使用次数的限定。

|  |
| --- |
| **public** **static** **void** sysPropList() **throws** IOException {  Properties prop = System.*getProperties*();  // prop.list(System.out);// 目的是控制台。  // 需求是：将jvm的属性信息存储到一个文件中。  prop.list(**new** PrintStream("java.txt"));  }  **public** **static** **void** sysProp() {  Properties prop = System.*getProperties*();  Set<String> keys = prop.stringPropertyNames();  **for** (String key : keys) {  System.*out*.println(key + ":" + prop.getProperty(key));  }  } |

Properties类与配置文件

Map

|--Hashtable

|--Properties

注意：是一个Map集合，该集合中的键值对都是字符串。该集合通常用于对键值对形式的配置文件进行操作.

配置文件：将软件中可变的部分数据可以定义到一个文件中，方便以后更改，该文件称之为配置文件。

优势: 提高代码的维护性。

Properties: 该类是一个Map的子类，提供了可以快速操作配置文件的方法

load() : 将文件设备数据装载为Map集合数据

get(key): 获取Map中的数据

getProperty()获取Map中的数据特有方法

案例：

|  |
| --- |
| /\*  \* 将配置文件中的数据通过流加载到集合中。  \*/  **public** **static** **void** loadFile() **throws** IOException {  // 1,创建Properties(Map)对象  Properties prop = **new** Properties();  // 2.使用流加载配置文件。  FileInputStream fis = **new** FileInputStream("c:\\qq.txt");  // 3。使用Properties 对象的load方法将流中数据加载到集合中。  prop.load(fis);  // 遍历该集合  Set<Entry<Object, Object>> entrySet = prop.entrySet();  Iterator<Entry<Object, Object>> it = entrySet.iterator();  **while** (it.hasNext()) {  Entry<Object, Object> next = it.next();  Object key = next.getKey();  Object value = next.getValue();  }  // 通过键获取指定的值  Object object = prop.get("jack");  System.*out*.println(object);  // 通过键修改值  prop.setProperty("jack", "888888");  // 将集合中的数据写入到配置文件中。  FileOutputStream fos = **new** FileOutputStream("c:\\qq.txt");  // 注释：  prop.store(fos, "yes,qq");  fos.close();  fis.close();  } |

获取记录程序运行次数：

|  |
| --- |
| **public** **class** Demo6 {  **public** **static** **void** main(String[] args) **throws** IOException {  **int** count = 0;  Properties pro = **new** Properties();  File file = **new** File("c:\\count.ini");  FileInputStream fis = **null**;  **if** (!file.exists()) {  file.createNewFile();  }  fis = **new** FileInputStream(file);  pro.load(fis);  String str = pro.getProperty("count");  **if** (str != **null**) {  count = Integer.*parseInt*(str);  }  **if** (count == 3) {  System.*out*.println("使用次数已到，请付费");  System.*exit*(0);  }  count++;  System.*out*.println("欢迎使用本软件" + "你已经使用了：" + count + " 次");  pro.setProperty("count", count + "");  FileOutputStream fos = **new** FileOutputStream(**new** File("c:\\count.ini"));  pro.store(fos, "请保护知识产权");  fis.close();  fos.close();  }  } |

## 打印流

PrintStream可以接受文件和其他字节输出流，所以打印流是对普通字节输出流的增强，其中定义了很多的重载的print()和println()，方便输出各种类型的数据。

### PrintStream

PrintWriter

1，打印流。

PrintStream：

是一个字节打印流，System.out对应的类型就是PrintStream。

它的构造函数可以接收三种数据类型的值。

1，字符串路径。

2，File对象。

3，OutputStream。

|  |
| --- |
| **public** **static** **void** main(String[] args) **throws** IOException {  PrintStream ps = System.*out*;  // 普通write方法需要调用flush或者close方法才会在控制台显示  // ps.write(100);  // ps.close();  // 不换行打印  ps.print(100);  ps.print('a');  ps.print(100.5);  ps.print("世界");  ps.print(**new** Object());  System.*out*.println("--------------");  // 换行  ps.println(100);  ps.println('a');  ps.println(100.5);  ps.println("世界");  ps.println(**new** Object());  // 重定向打印流  PrintStream ps2 = **new** PrintStream(**new** File("c:\\a.txt"));  System.*setOut*(ps2);  // 换行  ps2.println(100);  ps2.println('a');  ps2.println(100.5);  ps2.println("世界");  ps2.println(**new** Object());  // printf(); 格式化  ps2.printf("%d,%f,%c,%s", 100, 3.14, '中', "世界你好!!!");  ps2.printf("%4s和%8s 打价格战", "京东", "苏宁");    } } |

注意: 打印流的三种方法

void print(数据类型 变量)

println(数据类型 变量)

printf(String format, Object... args)

可以自定数据格式

print 和println方法的区别在于,一个换行一个不换行

print 方法和write方法的却别在于,print提供自动刷新.

普通的write方法需要调用flush或者close方法才可以看到数据.

JDK1.5之后Java对PrintStream进行了扩展，增加了格式化输出方式，可以使用printf()重载方法直接格式化输出。但是在格式化输出的时候需要指定输出的数据类型格式。



### **PrintWriter**

是一个字符打印流。构造函数可以接收四种类型的值。

1，字符串路径。

2，File对象。

对于1，2类型的数据，还可以指定编码表。也就是字符集。

3，OutputStream

4，Writer

对于3，4类型的数据，可以指定自动刷新。

注意：该自动刷新值为true时，只有三个方法可以用：println,printf,format.

如果想要既有自动刷新，又可执行编码。如何完成流对象的包装？

PrintWrter pw =

new PrintWriter(new OutputSteamWriter(new FileOutputStream("a.txt"),"utf-8"),true);

如果想要提高效率。还要使用打印方法。

PrintWrter pw =

newPrintWriter(new BufferdWriter(new OutputSteamWriter(

newFileOutputStream("a.txt"),"utf-8")),true);

|  |
| --- |
| **public** **static** **void** testPrintWriter() **throws** Exception {  PrintWriter pw = **new** PrintWriter("c:/b.txt", "gbk");  // pw.append("xxx");  // pw.println(55);  // pw.println('c');  // pw.printf("%.1s与%4s打价格战, %c", "京东","苏宁", 'a');  pw.close();  } |

Scanner

|  |
| --- |
| **public** **static** **void** testScanner() **throws** Exception {  // Scanner scanner = new Scanner(new File("c:/test.txt"));  Scanner scanner = **new** Scanner(System.*in*);  System.*out*.println(scanner.nextInt());  System.*out*.println(scanner.nextBoolean());  scanner.close();  } |

## 操作数组的流对象

### 操作字节数组

ByteArrayInputStream

以及ByteArrayOutputStream

toByteArray();

toString();

writeTo(OutputStream);

|  |
| --- |
| **public** **static** **void** testByteArrayInputStream() **throws** Exception {  InputStream in = **new** ByteArrayInputStream(**new** **byte**[] { 65, 66, 67 });  ByteArrayOutputStream out = **new** ByteArrayOutputStream();  **for** (**int** b = -1; (b = in.read()) != -1;) {  out.write(b);  }  in.close();  out.close();  System.*out*.println(Arrays.*toString*(out.toByteArray()));  System.*out*.println(out);  } |

### 操作字符数组

CharArrayReader

CharArrayWriter

对于这些流，源是内存。目的也是内存。

而且这些流并未调用系统资源。使用的就是内存中的数组。

所以这些在使用的时候不需要close。

操作数组的读取流在构造时，必须要明确一个数据源。所以要传入相对应的数组。

对于操作数组的写入流，在构造函数可以使用空参数。因为它内置了一个可变长度数组作为缓冲区。

|  |
| --- |
| **public** **static** **void** testCharArrayReader() **throws** Exception {  CharArrayReader reader = **new** CharArrayReader(**new** **char**[] { 'A', 'b', 'c' });  CharArrayWriter writer = **new** CharArrayWriter();  **for** (**int** b = -1; (b = reader.read()) != -1;) {  writer.write(b);  }  reader.close();  writer.close();  System.*out*.println(writer.toCharArray());  } |

这几个流的出现其实就是通过流的读写思想在操作数组。

类似的对象同理：

StringReader

StringWriter。

|  |
| --- |
| **public** **static** **void** testStringReader() **throws** Exception {  StringReader reader = **new** StringReader("test 中国");  StringWriter writer = **new** StringWriter();  **for** (**int** b = -1; (b = reader.read()) != -1;) {  writer.write(b);  }  reader.close();  writer.close();  System.*out*.println(writer.toString());  } |

## 操作基本数据类型的流对象

### DataInputStream

以及DataOutputStream

查看API文档DataInputStream的信息。发现从底层输入流中读取基本 Java 数据类型。查看方法,有读一个字节,读一个char读一个double 的方法,

DataInputStream 从数据流读取字节，并将它们转换为正确的基本数据类型值或字符串。

该流有操作基本数据类型的方法.

有读的,那么必定有对应的写的就是DataOutputStream 将基本类型的值或字符串转换为字节，并且将字节输出到数据流。

DataInputStream类继承FilterInputStream类，并实现了DataInput接口。DataOutputStream

类继承FilterOutputStream 并实现了DataOutput 接口。

例如：

|  |
| --- |
| DataInputStream  操作基本数据类型的方法：  int readInt():一次读取四个字节，并将其转成int值。  boolean readBoolean():一次读取一个字节。  short readShort();  long readLong();  剩下的数据类型一样。  String readUTF():按照utf-8修改版读取字符。注意，它只能读writeUTF()写入的字符数据。  DataOutputStream  DataOutputStream(OutputStream):  操作基本数据类型的方法:  writeInt(int)：一次写入四个字节。  注意和write(int)不同。write(int)只将该整数的最低一个8位写入。剩余三个8位丢弃。  writeBoolean(boolean);  writeShort(short);  writeLong(long);  剩下是数据类型也也一样。  writeUTF(String):按照utf-8修改版将字符数据进行存储。只能通过readUTF读取。 |

测试: DataOutputStream

使用DataOutputStream写数据文件。

|  |
| --- |
| **public** **static** **void** testDataInputStream() **throws** Exception {  DataOutputStream out = **new** DataOutputStream(**new** FileOutputStream(  "c:/a.txt"));  out.writeBoolean(**true**);  out.writeByte(15); // 0x05 1 个字节  out.writeBytes("abc"); // 0x 0041 2个字节  out.writeChar('X'); // ??  out.writeChars("xyz");  out.writeLong(111);  out.writeUTF("中国");  out.close();  DataInputStream in = **new** DataInputStream(  **new** FileInputStream("c:/a.txt"));  System.*out*.println(in.readBoolean());  System.*out*.println(in.readByte());    System.*out*.println(in.readByte());  System.*out*.println(in.readByte());  System.*out*.println(in.readByte());    System.*out*.println(in.readChar());    System.*out*.println(in.readChar());  System.*out*.println(in.readChar());  System.*out*.println(in.readChar());    System.*out*.println(in.readLong());    System.*out*.println(in.readUTF());  in.close();  } |

# 编码

什么是编码？

计算机中存储的都是二进制，但是要显示的时候，就是我们看到的却可以有中国 ，a 1 等字符

计算机中是没有存储字符的，但是我们却看到了。计算机在存储这些信息的时候，根据一个有规则的编号，当用户输入a 有a对映的编号，就将这个编号存进计算机中这就是编码。

计算机只能识别二进制数据。

为了方便应用计算机，让它可以识别各个国家的文字。就将各个国家的文字用数字来表示，并一一对应，形成一张表，这就是编码表。

例如：汉字 中

有一种编码：

中字在utf 8中对映的编码

utf-8 -->100

在gbk中呢？有可能就不是100了

gbk --> 150

很显然同一个信息在不同的编码中对映的数字也不同，

不同的国家和地区使用的码表是不同的，

gbk 是中国大陆

bjg5 是台湾同胞中的繁体字。所以如果给big5一个简体字是不认识的。

还有ASCII 美国标准信息交换码

## **码表**

常见的码表如下：

ASCII： 美国标准信息交换码。用一个字节的7位可以表示。

ISO8859-1： 拉丁码表。欧洲码表，用一个字节的8位表示。又称Latin-1(拉丁编码)或“西欧语言”。ASCII码是包含的仅仅是英文字母，并且没有完全占满256个编码位置，所以它以ASCII为基础，在空置的0xA0-0xFF的范围内，加入192个字母及符号，

藉以供使用变音符号的拉丁字母语言使用。从而支持德文，法文等。因而它依然是一个单字节编码，只是比ASCII更全面。

GB2312： 中国的中文编码表。

GBK： 中国的中文编码表升级，融合了更多的中文文字符号。

Unicode： 国际标准码，融合了多种文字。所有文字都用两个字节来表示，Java语言使用的就是unicode。

UTF-8： 最多用三个字节来表示一个字符。

（我们以后接触最多的是iso8859-1、gbk、utf-8）

查看上述码表后，很显然中文的‘中’在iso8859-1中是没有对映的编码的。或者一个字符在2中码表中对应的编码不同，例如有一些字在不同的编码中是有交集的，例如bjg5 和gbk 中的汉字简体和繁体可能是一样的，就是有交集，但是在各自码表中的数字不一样。

例如

使用gbk 将中文保存在计算机中，

中 国

对映 100 200 如果使用big5 打开

可能 ？ ...

不同的编码对映的是不一样的。

很显然，我们使用什么样的编码写数据，就需要使用什么样的编码来对数据。

ISO8859-1：一个字节

GBK： 两个字节包含了英文字符和扩展的中文 ISO8859-1+中文字符

UTF-8 万国码，推行的。是1~3个字节不等长。英文存的是1个字节，中文存的是3个字节，是为了节省空间。

## **编码：**

字符串---》字节数组

String类的getBytes() 方法进行编码，将字符串，转为对映的二进制，并且这个方法可以指定编码表。如果没有指定码表，该方法会使用操作系统默认码表。

注意：中国大陆的Windows系统上默认的编码一般为GBK。在Java程序中可以使用System.getProperty("file.encoding")方式得到当前的默认编码。

## **解码：**

字节数组---》字符串

String类的构造函数完成。

String(byte[] bytes) 使用系统默认码表

String(byte[],charset)指定码表

注意：我们使用什么字符集（码表）进行编码，就应该使用什么字符集进行解码，否则很有可能出现乱码（兼容字符集不会）。

|  |
| --- |
| // 编码操作与解码操作。  **public** **static** **void** main(String[] args) **throws** Exception {  String value = System.*getProperty*("file.encoding");  System.*out*.println("系统默认的编码为 " + value);  String str = "中";  // 编码操作  **byte**[] bytes = str.getBytes();  **byte**[] bytes2 = str.getBytes("gbk");// d6d0  **byte**[] bytes3 = str.getBytes("utf-8");// e4b8ad  System.*out*.println(Arrays.*toString*(bytes)); // [-42, -48]  System.*out*.println(Arrays.*toString*(bytes2));// [-42, -48]  System.*out*.println(Arrays.*toString*(bytes3));// [-28, -72, -83]  // 解码操作  // 编码gbk,解码utf-8乱码。  String str2 = **new** String(bytes2, "utf-8");  System.*out*.println(str2);  // 编码utf-8 解码gbk，乱码  str2 = **new** String(bytes3, "gbk");  System.*out*.println(str2);  // gbk兼容gb2312所以，没有问题。  str = **new** String("中国".getBytes("gb2312"), "gbk");  System.*out*.println(str);  } |

存文件时可以使用各种编码，但是解码的时候要对映的采用相同的解码方式。

我们的字符流自动的做了编码和解码的工作，写一个中文，字符流进行了编码，存到了计算机中读到了一个字符，字符流进行了解码，我们可以看到字符。因为文件存的都是二进制。

但是拷贝图片时，是纯二进制，不是有意义的字符，所以码表无法转换。

字符流的弊端：

一：无法拷贝图片和视频。

二：拷贝文件使用字节流而不使用字符流，因为字符流读文件涉及到解码，会先解码，写文件的时候又涉及到编码，这些操作多余，而且读和写的码表不对应还容易引发问题。

例如FileReader 读文件，我们没有指定编码时，默认是按照系统编码gbk进行操作，如果读到utf-8的文件也是按照gbk编码进行解码，那就会出现问题。

## **字节流读取中文**

|  |
| --- |
| **public** **class** TestIo {  **public** **static** **void** main(String[] args) **throws** IOException {  *readFileByInputStream2*("c:\\a.txt");  }  **private** **static** **void** readFileByInputStream2(String path) **throws** IOException {  FileInputStream fis = **new** FileInputStream(path);  **int** len = 0;  **while** ((len = fis.read()) != -1) {  System.*out*.print((**char**) len);  }  }  } |

这个方法读取文本文件，中文是无法正确显示的。

很显然这些字节需要解码，可以将字节输入流读取的信息保存在字节数组中，指定对应的码表进行解码即可。

|  |
| --- |
| **public** **class** TestIo {  **public** **static** **void** main(String[] args) **throws** IOException {  *readFileByInputStream*("c:\\a.txt");  }  **private** **static** **void** readFileByInputStream(String path) **throws** IOException {  FileInputStream fis = **new** FileInputStream(path);  **int** len = 0;  **byte**[] buffer = **new** **byte**[1024];  **while** ((len = fis.read(buffer)) != -1) {  System.*out*.println(**new** String(buffer, 0, len, "gbk"));  }  }  } |

注意：如果指定的编码表和解码表不对应就会出现问题

|  |
| --- |
| **public** **class** TestIo {  **public** **static** **void** main(String[] args) **throws** IOException {  // 该文件默认是gbk编码  *readFileByInputStream*("c:\\a.txt");  }  **private** **static** **void** readFileByInputStream(String path) **throws** IOException {  FileInputStream fis = **new** FileInputStream(path);  **int** len = 0;  **byte**[] buffer = **new** **byte**[1024];  **while** ((len = fis.read(buffer)) != -1) {  // 使用utf-8 解码，解错。  System.*out*.println(**new** String(buffer, 0, len, "utf-8"));  }  }  } |

## **字节流写出中文**

需要编码，可以指定码表。就需要自己把字符串进行编码操作后，把得到的二进制内容通过字节流写入到文件中

使用String的getBytes方法，无参数的会使用系统默认的码表进行编码，也可以指定码表

系统默认编码

|  |
| --- |
| **public** **class** TestIo {  **public** **static** **void** main(String[] args) **throws** IOException {  String path = "c:\\test.txt";  *writeFileByOutputStream*(path, "世界你好");  *readFileByInputStream*(path);  }  **private** **static** **void** writeFileByOutputStream(String path, String content)  **throws** IOException {  FileOutputStream fos = **new** FileOutputStream(path);  // 把字符串进行编码操作，系统默认编码  **byte**[] bytes = content.getBytes();  // 内容通过字节流写入到文件中。  fos.write(bytes);  fos.close();  }  **private** **static** **void** readFileByInputStream(String path) **throws** IOException {  FileInputStream fis = **new** FileInputStream(path);  **int** len = 0;  **byte**[] buffer = **new** **byte**[1024];  **while** ((len = fis.read(buffer)) != -1) {  // 二进制解码，使用系统默认编码  System.*out*.println(**new** String(buffer, 0, len));  }  }  } |

使用utf-8进行编码

|  |
| --- |
| **public** **class** TestIo {  **public** **static** **void** main(String[] args) **throws** IOException {  String path = "c:\\test.txt";  *writeFileByOutputStream*(path, "世界你好");  *readFileByInputStream*(path);  }  **private** **static** **void** writeFileByOutputStream(String path, String content)  **throws** IOException {  FileOutputStream fos = **new** FileOutputStream(path);  // 把字符串进行编码操作  **byte**[] bytes = content.getBytes("utf-8");  // 内容通过字节流写入到文件中。  fos.write(bytes);  fos.close();  }  **private** **static** **void** readFileByInputStream(String path) **throws** IOException {  FileInputStream fis = **new** FileInputStream(path);  **int** len = 0;  **byte**[] buffer = **new** **byte**[1024];  **while** ((len = fis.read(buffer)) != -1) {  // 二进制解码，使用系统默认编码  System.*out*.println(**new** String(buffer, 0, len,"utf-8"));  }  }  } |

在明白了字节流也可以正确的处理中文字符之后，就应该明白字符流其实就是字节流在加上系统默认的码表。自动进行了编码和解码的操作。底层还是使用字节流读取文件。通过转换流的学习就可以明白这些道理。

## 转换流

InputStreamReader

查看API文档，发现是字节流通向字符流的桥梁。查看构造，可以传递字节流，可以指定编码，该流可以实现什么功能？很显然可以包装我们的字节流，自动的完成节流编码和解码的工作。该流是一个Reader的子类，是字符流的体系。所以将转换流称之为字节流和字符流之间的桥梁。

**InputStreamReader 是字节流通向字符流的桥梁**

测试InputStreamReader：

第一步: 需要专门新建以GBK编码的文本文件。为了便于标识，我们命名为gbk.txt

和以UFT-8编码的文本文件,命名为utf.txt  
 第二步: 分别写入汉字”中国”

第三步:编写测试方法,用InputStreamReader 分别使用系统默认编码,GBK,UTF-8编码读取文件.

|  |
| --- |
| **public** **class** Demo4 {  **public** **static** **void** main(String[] args) **throws** IOException {  File file = **new** File("c:\\a.txt");  File fileGBK = **new** File("c:\\gbk.txt");  File fileUTF = **new** File("c:\\utf.txt");  // 默认编码  *testReadFile*(file);  // 传入gbk编码文件,使用gbk解码  *testReadFile*(fileGBK, "gbk");  // 传入utf-8文件,使用utf-8解码  *testReadFile*(fileUTF, "utf-8");  }  // 该方法中nputStreamReader使用系统默认编码读取文件.  **private** **static** **void** testReadFile(File file) **throws**  IOException {  FileInputStream fis = **new** FileInputStream(file);  InputStreamReader ins = **new** InputStreamReader(fis);  **int** len = 0;  **while** ((len = ins.read()) != -1) {  System.*out*.print((**char**) len);  }  ins.close();  fis.close();  }  // 该方法使用指定编码读取文件  **private** **static** **void** testReadFile(File file, String encod)  **throws** IOException {  FileInputStream fis = **new** FileInputStream(file);  InputStreamReader ins = **new** InputStreamReader(fis, encod);  **int** len = 0;  **while** ((len = ins.read()) != -1) {  System.*out*.print((**char**) len);  }  ins.close();  }  } |

注意：码表不对应  
分别测试:

使用系统默认编码读取utf-8编码文件

使用utf-8编码读取gbk编码文件

使用"gbk”编码读取utf-8文件.

发现都会出现乱码的问题.

|  |
| --- |
| // 使用系统默认编码读取utf-8  *testReadFile*(fileUTF);  // 传入gbk编码文件,使用utf-8解码  *testReadFile*(fileGBK, "utf-8");  // 传入utf-8文件,使用"gbk解码  *testReadFile*(fileUTF, "gbk"); |

类 OutputStreamWriter

OutputStreamWriter

有了InputStreamReader 可以转换InputStream

那么其实还有OutputStreamWriter 可以转换OutputStream

**OutputStreamWriter 是字符流通向字节流的桥梁**

测试OutputStreamWriter

一: 分别使用OutputStreamWriter使用系统默认编码,GBK,UTF-8相对应的默认编码文件,GBK编码文件,UTF-8编码文件中写出汉字”中国”.

二: 在使用上述案例中的readFile方法传入相对应码表读取.

|  |
| --- |
| **public** **class** TestIo {  **public** **class** Demo4 {  **public** **static** **void** main(String[] args) **throws** IOException {  File file = **new** File("c:\\a.txt");  File fileGBK = **new** File("c:\\gbk.txt");  File fileUTF = **new** File("c:\\utf.txt");  // 写入  // 使用系统默认码表写入  *testWriteFile*(file);  // 使用gbk编码向gbk文件写入信息  *testWriteFile*(fileGBK, "gbk");  // 使用utf-8向utf-8文件中写入信息  *testWriteFile*(fileUTF, "utf-8");    // 读取  // 默认编码  *testReadFile*(file);  // 传入gbk编码文件,使用gbk解码  *testReadFile*(fileGBK, "gbk");  // 传入utf-8文件,使用utf-8解码  *testReadFile*(fileUTF, "utf-8");  }  // 使用系统码表将信息写入到文件中  **private** **static** **void** testWriteFile(File file) **throws** IOException {  FileOutputStream fos = **new** FileOutputStream(file);  OutputStreamWriter ops = **new** OutputStreamWriter(fos);  ops.write("中国");  ops.close();  }  // 使用指定码表,将信息写入到文件中  **private** **static** **void** testWriteFile(File file, String encod)  **throws** IOException {  FileOutputStream fos = **new** FileOutputStream(file);  OutputStreamWriter ops = **new** OutputStreamWriter(fos, encod);  ops.write("中国");  ops.close();  }  // 该方法中nputStreamReader使用系统默认编码读取文件.  **private** **static** **void** testReadFile(File file) **throws** IOException {  FileInputStream fis = **new** FileInputStream(file);  InputStreamReader ins = **new** InputStreamReader(fis);  **int** len = 0;  **while** ((len = ins.read()) != -1) {  System.*out*.print((**char**) len);  }  ins.close();    }  // 该方法适合用指定编码读取文件  **private** **static** **void** testReadFile(File file, String encod)  **throws** IOException {  FileInputStream fis = **new** FileInputStream(file);  InputStreamReader ins = **new** InputStreamReader(fis, encod);  **int** len = 0;  **while** ((len = ins.read()) != -1) {  System.*out*.print((**char**) len);  }    ins.close();  }  } |

注意: 码表不对应的问题

分别测试:

向GBK文件中写入utf-8编码的信息

向utf文件中写入gbk编码的信息

发现文件都有问题,无法正常的读取了.

|  |
| --- |
| **public** **static** **void** main(String[] args) **throws** IOException {  File file = **new** File("c:\\a.txt");  File fileGBK = **new** File("c:\\gbk.txt");  File fileUTF = **new** File("c:\\utf.txt");  // 写入  // // 使用系统默认码表写入  // testWriteFile(file);  // // 使用gbk编码向gbk文件写入信息  // testWriteFile(fileGBK, "gbk");  // // 使用utf-8向utf-8文件中写入信息  // testWriteFile(fileUTF, "utf-8");  *testWriteFile*(fileGBK);  // 向GBK文件中写入utf-8编码的信息  *testWriteFile*(fileGBK, "utf-8");  // 向utf文件中写入gbk编码的信息  *testWriteFile*(fileUTF, "gbk");  // 读取  // 默认编码  *testReadFile*(file);  // 传入gbk编码文件,使用gbk解码  *testReadFile*(fileGBK, "gbk");  // 传入utf-8文件,使用utf-8解码  *testReadFile*(fileUTF, "utf-8");  } |

InputStreamReader：字节到字符的桥梁。

OutputStreamWriter：字符到字节的桥梁。

它们有转换作用，而本身又是字符流。所以在构造的时候，需要传入字节流对象进来。

构造函数：

InputStreamReader(InputStream)

通过该构造函数初始化，使用的是本系统默认的编码表GBK。

InputStreamReader(InputStream,String charSet)

通过该构造函数初始化，可以指定编码表。

OutputStreamWriter(OutputStream)

通过该构造函数初始化，使用的是本系统默认的编码表GBK。

OutputStreamWriter(OutputStream,String charSet)

通过该构造函数初始化，可以指定编码表。

注意：

操作文件的字符流对象是转换流的子类。

|  |
| --- |
| Reader  |--InputStreamReader  |--FileReader  Writer  |--OutputStreamWriter  |--FileWriter |

注意：

在使用FileReader操作文本数据时，该对象使用的是默认的编码表。

如果要使用指定编码表时，必须使用转换流。

如果系统默认编码是GBK的：

FileReader fr = new FileReader("a.txt");//操作a.txt的中的数据使用的本系统默认的GBK。

操作a.txt中的数据使用的也是本系统默认的GBK。

InputStreamReader isr = new InputStreamReader(new FileInputStream("a.txt"));

这两句的代码的意义相同。

但是：如果a.txt中的文件中的字符数据是通过utf-8的形式编码。使用FileReader就无能为力，那么在读取时，就必须指定编码表。那么转换流必须使用。

InputStreamReader isr =

new InputStreamReader(new FileInputStream("a.txt"),"utf-8");

# 递归

递归做为一种算法在程序设计语言中广泛应用。是指函数/过程/子程序在运行过程中直接或间接调用自身而产生的重入现象。

（自己调用自己，有结束条件）

注意：递归时一定要明确结束条件。

数学中递归运算.

对于任何正整数N ,N! (读作N的阶乘)的值定义为1-N(包括N)的所有的整数的成绩.因此3! 就是 3!=3\*2\*1 =6;

5! 定义为5!=5\*4\*3\*2\*1=120

那么整数N 的阶乘 N! 可以表示为

1!=1

N!=N\*(N-1)! for N>1

若果N 等于1 那么1的继承就是1,其他所有N! =N\*(N-1)!,例如:50!=50\*49!

49!=49\*48! 48!=48\*47! 一直持续到1出现.

如何使用Java程序计算阶乘?

|  |
| --- |
| **public** **static** **long** recursion(**int** n) {  **if** (n == 1) {  **return** 1;  } **else** {  **return** n \* *recursion*(n - 1);  }  } |

## 案例：

1，列出指定目录中所有的子孙文件与子孙目录名，只需要列出名称即可。

2，列出指定目录中所有的子孙文件与子孙目录名，要求名称前面要有相应数量的空格：

第一级前面有0个，第二级前面有1个，第三级前面有2个...，以此类推。

3，列出指定目录中所有的子孙文件与子孙目录名，要求要是树状结构，效果如下所示：

|--src

| |--cn

| | |--itcast

| | | |--a\_helloworld

| | | | |--HelloWorld.java

| | | |--b\_for

| | | | |--ForTest.java

| | | |--c\_api

| | | | |--Student.java

|--bin

| |--cn

| | |--itcast

| | | |--i\_exception

| | | | |--ExceptionTest.class

| | | |--h\_linecount

| | | | |--LineCounter3.class

| | | | |--LineCounter2.class

| | | | |--LineCounter.class

|--lib

| |--commons-io.jar

答案:

案例一:

|  |
| --- |
| // 1，列出指定目录中所有的子孙文件与子孙目录名，只需要列出名称即可。  **private** **static** **void** listFile(File file) {  File[] listFiles = file.listFiles();  **for** (File f : listFiles) {  **if** (f.isFile()) {  System.*out*.println(f.getName());  } **else** **if** (f.isDirectory()) {  System.*out*.println(f.getName());  *listFile*(f);  }  }  }  **public** **static** **void** main(String[] args) {  File file = **new** File("c:\\abc");  listFile(file);  } |

案例二

|  |
| --- |
| // 2，列出指定目录中所有的子孙文件与子孙目录名，要求名称前面要有相应数量的空格：  **private** **static** **void** listFile2(File file, String str) {  File[] listFiles = file.listFiles();  **for** (**int** i = 0; i < listFiles.length; i++) {  File f = listFiles[i];  System.*out*.println(str + f.getName());  **if** (f.isDirectory()) {  *listFile2*(f, str + "-");  }  }  }  **public** **static** **void** main(String[] args) {  File file = **new** File("c:\\abc");  String str = "-";  listFile2(file, str); |

案例三:

|  |
| --- |
| // 列出指定目录中所有的子孙文件与子孙目录名，要求要是树状结构  **private** **static** **void** listFile3(File file, String str) {  File[] listFiles = file.listFiles();  **for** (File f : listFiles) {  System.*out*.println(str + f.getName());  **if** (f.isDirectory()) {  *listFile3*(f, "| " + str);  }  }  }  **public** **static** **void** main(String[] args) {  File file = **new** File("c:\\abc");  file = **new** File("c:\\day18ide");  file = **new** File("c:\\MyIo");  str = "|-";  listFile3(file, str); |

## 练习：

1，删除一个非空的目录。

2，移动一个非空的目录到另一个地方（剪切）。

3，把File类中的重要方法设计代码测试一遍。

|  |
| --- |
| // 1，删除一个非空的目录。并加强健壮性  **private** **static** **void** deleteFile(File file) {  **if** (!file.exists()) {  System.*out*.println("路径不存在");  **return**;  }  **if** (!file.isDirectory()) {  System.*out*.println("不是目录");  **return**;  }  // 如果当前目录中有子目录和文件,先删除子目录和文件  File[] listFiles = file.listFiles();  **for** (File f : listFiles) {  **if** (f.isFile()) {  f.delete();  } **else** **if** (f.isDirectory()) {  *deleteFile*(f);  }  }  // 删除当前目录  file.delete();  } |

0o

练习2:

使用File类的renameTo 方法和递归实现非空目录的剪切.

|  |
| --- |
| **public** **static** **void** main(String[] args) **throws** IOException {  // 重命名文件(成功)  // File src = new File("c:\\aaa.txt");  // File dest = new File("c:\\bbb.txt");  // src.renameTo(dest);  // //移动文件(成功)  // File src = new File("c:\\aaa.txt");  // File dest = new File("d:\\aaa.txt");  // src.renameTo(dest);  // 移动一个空目录(失败)  // File src = new File("c:\\aaa");  // File dest = new File("d:\\aaa");  // System.out.println(src.renameTo(dest));  // 使用File类和递归实现文件的剪切.  File src = **new** File("c:\\abc");  File dest = **new** File("d:\\");  *cutFile*(src, dest);  }  // 移动一个非空的目录到另一个地方（剪切）。  **private** **static** **void** cutFile(File srcDir, File dest) **throws** IOException {  **if** (!srcDir.exists() || !dest.exists()) {  System.*out*.println("指定的源目录或者目标目录不存在");  **return**;  }  **if** (!srcDir.isDirectory() || !dest.isDirectory()) {  System.*out*.println("指定的源目录或者目标目录不是目录");  **return**;  }  // 得到源目录名  String srcDirName = srcDir.getName(); // abc  // 根据源目录名创建新目录名  File destDir = **new** File(dest + srcDirName); // d:\\abc dest 为父路径  // srcDirName 为子路径  // 创建目标目录  destDir.mkdir();  // 遍历源目录  File[] listFiles = srcDir.listFiles();  **for** (File f : listFiles) {  // 如果是子源文件,使用renameTo方法,移动至目标目录中(该方法同时会删除源目录中的文件)  **if** (f.isFile()) {  f.renameTo(**new** File(destDir, f.getName())); // 指定目标文件的父目录,文件名(根据源文件名生成).  } **else** **if** (f.isDirectory()) {  // 如果是子目录,执行重复动作. 将源子目录 , 目标目录(父目录+//)  *cutFile*(f, **new** File(destDir, File.*separator*)); // 指定源目录,指定目的路径d:\\abc\\  }  }  // 删除源目录  srcDir.delete();  } |

# 批处理文件(bat)

简单的说，批处理的作用就是自动的连续执行多条命令 .编写bat处理文件可以使用记事本的方式:

常见批处理文件的命令：

echo 表示显示此命令后的字符

tiltle 设置窗口的标题。

echo off 表示在此语句后所有运行的命令都不显示命令行本身

color 设置窗体的字体颜色。

@与echo off相象，但它是加在每个命令行的最前面，表示运行时不显示这一行的命令行（只能影响当前行）。

pause 运行此句会暂停批处理的执行并在屏幕上显示Press any key to continue...的提示，等待用户按任意键后继续

rem 表示此命令后的字符为解释行（注释），不执行，只是给自己今后参考用的（相当于程序中的注释） 或者%注释的内容%

%[1-9]表示参数，参数是指在运行批处理文件时在文件名后加的以空格（或者Tab）分隔的字符串

# 对象拷贝

## 对象的浅拷贝

浅复制(浅克隆)被复制对象的所有变量都含有与原来对象相同的值，而所有的对其他对象的引用仍然只指向原来的对象，换言之，浅复制仅仅复制锁考虑的对象，而不复制它所引用的对象。

|  |
| --- |
| **public** **class** Student **implements** Cloneable{  String name;  **int** age;  Student(String name,**int** age){  **this**.name=name;  **this**.age=age;  }    **public** Object clone(){  Object o =**null**;  **try**{  o=**super**.clone();//Object中的clone()识别出你要复制的哪一个对象  }  **catch**(CloneNotSupportedException e){  System.*out*.println(e.toString());  }  **return** o;  }    **public** **static** **void** main(String[] args){  Student s1 = **new** Student("zhang",18);  Student s2 = (Student)s1.clone();  s2.name="li";  s2.age=20;  System.*out*.println("name="+s1.name+","+"age="+s1.age);//修改学生2后不影响学生1的值  }  } |

## 对象深拷贝

深复制(深克隆)被复制对象的所有变量都含有与原来的对象相同的值，除去那些引用其他对象的变量，那些引用其他对象的变量将指向被复制过的新对象，而不再试原有的那些被引用的对象，换言之，深复制把要复制的对象所引用的对象都复制了一遍。

把对象写到流里的过程是串行化（Serilization）过程，但是在Java程序师圈子里又非常形象地称为“冷冻”或者“腌咸菜（picking）”过程；而把对象从流中读出来的并行化（Deserialization）过程则叫做“解冻”或者“回鲜(depicking)”过程。应当指出的是，写在流里的是对象的一个拷贝，而原对象仍然存在于JVM里面，因此“腌成咸菜”的只是对象的一个拷贝，Java咸菜还可以回鲜。

在Java语言里深复制一个对象，常常可以先使对象实现Serializable接口，然后把对象（实际上只是对象的一个拷贝）写到一个流里（腌成咸菜），再从流里读出来（把咸菜回鲜），便可以重建对象。

|  |
| --- |
| **public** Object deepClone()  {  //将对象写到流里  ByteArrayOutoutStream bo=**new** ByteArrayOutputStream();  ObjectOutputStream oo=**new** ObjectOutputStream(bo);  oo.writeObject(**this**);  //从流里读出来  ByteArrayInputStream bi=**new** ByteArrayInputStream(bo.toByteArray());  ObjectInputStream oi=**new** ObjectInputStream(bi);  **return**(oi.readObject());  } |

### 内存溢出

由于Java具备自动的垃圾回收机制,当我们使用完对象之后,它们会被自动回收,是不是我们在Java程序中不需要再考虑内存管理了吗?

请看如下程序:

|  |
| --- |
| **class** Stack {  **private** Object[] elements;  // 初始化角标  **int** index = 0;  // 默认初始化容量  **private** **int** initialCapacity = 10;  **public** Stack() {  elements = **new** Object[initialCapacity];  }  // 压栈 push  **public** **void** push(Object e) {  ensureCapacity();  elements[index++] = e;  // System.out.println(index);  }  // 弹栈 pop  **public** Object pop() {  **if** (index == 0) {  **throw** **new** RuntimeException("没有元素");  }  **return** elements[--index];  }  **private** **void** ensureCapacity() {  **if** (index == elements.length) {  elements = Arrays.*copyOf*(elements, index \* 2 + 1);  }  }  } |

注意:从栈中弹出的对象不会作为垃圾回收,即使程序不再使用这些对象,因为栈内部继续维护着这些对象.最终可能会导致内存占用的不断增加,程序性能降低.这就是内存泄漏.

改进版本

|  |
| --- |
| **class** Stack {  **private** Object[] elements;  // 初始化角标  **int** index = 0;  // 默认初始化容量  **private** **int** initialCapacity = 10;  **public** Stack() {  elements = **new** Object[initialCapacity];  }  // 压栈 push  **public** **void** push(Object e) {  ensureCapacity();  elements[index++] = e;  // System.out.println(index);  }  // 弹栈 pop  **public** Object pop() {  **if** (index == 0) {  **throw** **new** RuntimeException("没有元素");  }  Object obj = elements[--index];  elements[index] = **null**;  **return** obj;  }  **private** **void** ensureCapacity() {  **if** (index == elements.length) {  elements = Arrays.*copyOf*(elements, index \* 2 + 1);  }  }  } |

## 设计模式

设计模式（Design pattern）是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性。

## 观察者模式

有时又被称为

发布-订阅<Publish/Subscribe>模式、

模型-视图<Model/View>模式、

源-收听者<Source/Listener>模式

或从属者<Dependents>模式）

这是软件设计模式的一种。

观察者模式（Observer）完美的将观察者和被观察的对象分离开。

此种模式中，一个目标物件管理所有相依于它的观察者物件，并且在它本身的状态改变时主动发出通知。

这通常透过呼叫各观察者所提供的方法来实现。

此种模式通常被用来实作事件处理系统。

有多个观察者时，不可以依赖特定的通知次序。

Swing大量使用观察者模式，许多GUI框架也是如此。

气象站：

|  |
| --- |
| **public** **class** WeatherStation {    **private** String weather;    String[] weathers = {"下雨","下雪","下冰雹","出太阳"};    **static** List<BookWeather> *list* = **new** ArrayList<BookWeather>();    Random random = **new** Random();      **public** **void** startWork(){    **new** Thread(){    @Override  **public** **void** run() {  **while**(**true**){  updateWeather();  **try** {  Thread.*sleep*(random.nextInt(1000)+500);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  }  }.start();  }  **public** **void** updateWeather(){  weather = weathers[random.nextInt(4)];  System.*out*.println("天气："+ weather);  }      **public** String getWeather() {  **return** weather;  } |

人：

|  |
| --- |
| **public** **class** Person **implements** BookWeather {    String name;    **public** Person(String name){  **this**.name = name;  }    **private** WeatherStation station ;    **public** Person(String name,WeatherStation station){  **this**(name);  **this**.station = station;  }    //下雨","下雪 ","下冰雹","出太阳"  @Override  **public** **void** notifyWeather() {  String weather = station.getWeather();  **if**("下雨".equals(weather)){  System.*out*.println(name+"打着雨伞上班");  }**else** **if**("下雪".equals(weather)){  System.*out*.println(name+"溜冰 上班");  }**else** **if**("下冰雹".equals(weather)){  System.*out*.println(name+"带着头盔 上班");  }**else** **if**("出太阳".equals(weather)){  System.*out*.println(name+"嗮着太阳 上班");  }  }    } |

测试类：

|  |
| --- |
| **public** **class** Test {    **public** **static** **void** main(String[] args) **throws** InterruptedException {  WeatherStation station = **new** WeatherStation();  station.startWork();    Person p1 = **new** Person("小明",station);  **while**(**true**){  p1.notifyWeather();  Thread.*sleep*(2000);  }  } |

问题：天气变化两三次，小明才知道一次。

解决方案 ：

|  |
| --- |
| package cn.itcast.test;  import java.util.List;  import java.util.ArrayList;  import java.util.Random;  public class WeatherStation {    private String weather;    String[] weathers = {"下雨","下雪","下冰雹","出太阳"};    private static List<BookWeather> list = new ArrayList<BookWeather>();    Random random = new Random();    public void addListaner(BookWeather e){  list.add(e);  }    public void startWork(){    new Thread(){    @Override  public void run() {  while(true){  updateWeather();  try {  Thread.sleep(random.nextInt(1000)+500);  } catch (InterruptedException e) {  e.printStackTrace();  }  }  }  }.start();  }  public void updateWeather(){  weather = weathers[random.nextInt(4)];  System.out.println("天气："+ weather);  for(BookWeather item : list){  item.notifyWeather(weather);  }  }      public String getWeather() {  return weather;  } |

人：

|  |
| --- |
| **public** **class** Person **implements** BookWeather {    String name;    **public** Person(String name){  **this**.name = name;  }    **private** WeatherStation station ;    **public** Person(String name,WeatherStation station){  **this**(name);  **this**.station = station;  }    //下雨","下雪 ","下冰雹","出太阳"  @Override  **public** **void** notifyWeather(String weather) {  **if**("下雨".equals(weather)){  System.*out*.println(name+"打着雨伞上班");  }**else** **if**("下雪".equals(weather)){  System.*out*.println(name+"溜冰 上班");  }**else** **if**("下冰雹".equals(weather)){  System.*out*.println(name+"带着头盔 上班");  }**else** **if**("出太阳".equals(weather)){  System.*out*.println(name+"嗮着太阳 上班");  }  } |

}

接口:

|  |
| --- |
| **public** **interface** BookWeather {    **public** **void** notifyWeather(String weather);  } |

|  |
| --- |
| **public** **class** Test {    **public** **static** **void** main(String[] args) **throws** InterruptedException {  WeatherStation station = **new** WeatherStation();  station.startWork();    Person p1 = **new** Person("小明");  Person p2 = **new** Person("小红");  Person p3 = **new** Person("小青 ");  station.addListaner(p1);  station.addListaner(p2);  station.addListaner(p3);    }  } |

### 单例

Singleton

是指只能拥有一个实例的类就是单例类。

私有构造方法。

获取方式

通过公共的静态方法创建单一的实例。

两种模式

懒汉模式 – 通常被称为延迟加载。注意存在线程安全问题.

饿汉模式

懒汉式的单例模式线程安全问题的解决方案：

|  |
| --- |
| **class** Single{    //声明本类的一个私有的成员变量  **private** **static** Single *single*;    //第一步 : 私有化构造方法  **private** Single(){    }  // 第三步：提供一个公共的方法获取该类的实例对象  **public** **static** Single getInstance(){  **if**(*single*==**null**){  **synchronized** (*single*) {  **if**(*single*==**null**){  *single* = **new** Single();  }  }  }  **return** *single*;  }  } |

## 反射

类字节码文件是在硬盘上存储的，是一个个的.class文件。我们在new一个对象时，JVM会先把字节码文件的信息读出来放到内存中，第二次用时，就不用在加载了，而是直接使用之前缓存的这个字节码信息。

字节码的信息包括：类名、声明的方法、声明的字段等信息。在Java中“万物皆对象”，这些信息当然也需要封装一个对象，这就是Class类、Method类、Field类。

通过Class类、Method类、Field类等等类可以得到这个类型的一些信息，甚至可以不用new关键字就创建一个实例，可以执行一个对象中的方法，设置或获取字段的值，**这就是反射技术**。

### Class类

#### 获取Class对象的三种方式

Java中有一个Class类用于代表某一个类的字节码。

Java提供了三种方式获取类的字节码

forName()。forName方法用于加载某个类的字节码到内存中，并使用class对象进行封装

类名.class

对象.getClass()

|  |
| --- |
| /\*\*  \* 加载类的字节码的3种方式  \* **@throws** Exception  \* \*/  **public** **void** test1() **throws** Exception {  // 方式一  Class clazz1 = Class.*forName*("cn.itcast.gz.reflect.Person");  // 方式二  Class clazz2 = Person.**class**;  // 方式三  Person p1 = **new** Person();  Class clazz3 = p1.getClass();  } |

#### 通过Class类获取类型的一些信息

1. getName()类的名称（全名，全限定名）

2 getSimpleName()类的的简单名称（不带包名）

3. getModifiers(); 类的的修饰符

4.创建对象

无参数构造创建对象

newInstance()

5. 获取指定参数的构造器对象，并可以使用Constructor对象创建一个实例

Constructor<T> getConstructor(Class<?>... parameterTypes)

|  |
| --- |
| /\*\*  \* 通过Class对象获取类的一些信息  \*  \* **@throws** Exception  \* \*/  **private** **static** **void** test2() **throws** Exception {  Class clazz1 = Class.*forName*("cn.itcast.gz.reflect.Person");  // 获取类的名称  String name = clazz1.getName();  System.*out*.println(name); // cn.itcast.gz.reflect.Person  // 获取类的简单名称  System.*out*.println(clazz1.getSimpleName()); // Person  // 获取类的修饰符  **int** modifiers = clazz1.getModifiers();  System.*out*.println(modifiers);  // 构建对象(默认调用无参数构造.)  Object ins = clazz1.newInstance();  Person p = (Person) ins;  System.*out*.println(p); // cn.itcast.gz.reflect.Person@c17164  // 获取指定参数的构造函数  Constructor<?> con = clazz1.getConstructor(String.**class**, **int**.**class**);  // 使用Constructor创建对象.  Object p1 = con.newInstance("jack", 28);  System.*out*.println(((Person) p1).getName());  } |

#### 通过Class类获取类型中的方法的信息

1.获取公共方法包括继承的父类的方法

getMethods()返回一个数组,元素类型是Method

2.获取指定参数的公共方法

getMethod("setName", String.**class**);

3.获得所有的方法,包括私有

Method[] getDeclaredMethods()

4.获得指定参数的方法,包括私有

Method getDeclaredMethod(String name, Class<?>... parameterTypes)

|  |
| --- |
| /\*\*  \* 获取公有方法.  \* **@throws** Exception  \* \*/  **private** **static** **void** test3() **throws** Exception {  Class clazz1 = Class.*forName*("cn.itcast.gz.reflect.Person");  // 1.获取非私用方法(包括父类继承的方法)  Method[] methods = clazz1.getMethods();  System.*out*.println(methods.length);  **for** (Method m : methods) {  // System.out.println(m.getName());  **if** ("eat".equals(m.getName())) {  m.invoke(clazz1.newInstance(), **null**);  }  }  } |

|  |
| --- |
| /\*\*  \* 获取指定方法签名的方法  \*  \* **@throws** Exception  \* \*/  **private** **static** **void** test4() **throws** Exception {  Class clazz1 = Class.*forName*("cn.itcast.gz.reflect.Person");  // 获取指定名称的函数  Method method1 = clazz1.getMethod("eat", **null**);  method1.invoke(**new** Person(), **null**);  } |

|  |
| --- |
| /\*\*  \* 获取指定方法名且有参数的方法  \*  \* **@throws** Exception  \* \*/  **private** **static** **void** test5() **throws** Exception {  Class clazz1 = Class.*forName*("cn.itcast.gz.reflect.Person");  Method method = clazz1.getMethod("eat", String.**class**);  method.invoke(**new** Person(), "包子");  }  /\*\*  \* 获取指定方法名,参数列表为空的方法.  \*  \* **@throws** Exception  \* \*/  **private** **static** **void** test4() **throws** Exception {  Class clazz1 = Class.*forName*("cn.itcast.gz.reflect.Person");  // 获取指定名称的函数  Method method1 = clazz1.getMethod("eat", **null**);  method1.invoke(**new** Person(), **null**);  } |

|  |
| --- |
| /\*\*  \* 反射静态方法  \* **@throws** Exception  \* \*/  **private** **static** **void** test7() **throws** Exception {  Class clazz1 = Class.*forName*("cn.itcast.gz.reflect.Person");  Method method = clazz1.getMethod("play", **null**);  method.invoke(**null**, **null**);  }  /\*\*  \* 访问私有方法 暴力反射  \* **@throws** Exception  \* \*/  **private** **static** **void** test6() **throws** Exception {  Class clazz1 = Class.*forName*("cn.itcast.gz.reflect.Person");  Method method = clazz1.getDeclaredMethod("movie", String.**class**);  method.setAccessible(**true**);  method.invoke(**new** Person(), "苍老师");  } |

#### 通过Class类获取类型中的字段的信息

1.获取公共字段

Field[] getFields()

2.获取指定参数的公共字段

Field getField(String name)

3.获取所有的字段

Field[] getDeclaredFields()

4.获取指定参数的字段,包括私用

Field getDeclaredField(String name)

|  |
| --- |
| /\*\*  \* 获取公有的字段  \* \*/  **private** **static** **void** test8() **throws** Exception {  Class clazz1 = Class.*forName*("cn.itcast.gz.reflect.Person");  Field[] fields = clazz1.getFields();  Person p = **new** Person();  System.*out*.println(fields.length);  **for** (Field f : fields) {  System.*out*.println(f.getName());  **if** ("name".equals(f.getName())) {  System.*out*.println(f.getType().getName());  f.set(p, "jack");  }  }  System.*out*.println(p.getName());  } |

|  |
| --- |
| /\*\*  \* 获取私有的字段  \* **@throws** Exception  \* \*/  **private** **static** **void** test9() **throws** Exception {  Class clazz1 = Class.*forName*("cn.itcast.gz.reflect.Person");  Field field = clazz1.getDeclaredField("age");  System.*out*.println(field.getName());  field.setAccessible(**true**);  Person p = **new** Person();  field.set(p, 100);  System.*out*.println(p.getAge());  } |

### 工厂模式

Factory

例如:汽车销售商场

该模式将创建对象的过程放在了一个静态方法中来实现.在实际编程中,如果需要大量的创建对象,该模式是比较理想的.

|  |
| --- |
| **public** **class** Demo1 {  **public** **static** **void** main(String[] args) {  System.*out*.println("买宝马");  Car bmw = *CarFactory*("BMW");  bmw.run();  System.*out*.println("买大奔");  Car benz = *CarFactory*("Benz");  benz.run();  }  **public** **static** Car CarFactory(String carName) {  **if** ("BMW".equals(carName)) {  **return** **new** BMW();  } **else** **if** ("Benz".equals(carName)) {  **return** **new** Benz();  } **else** {  **throw** **new** RuntimeException("车型有误");  }  }  }  **abstract** **class** Car {  **public** **abstract** **void** run();  }  **class** BMW **extends** Car {  @Override  **public** **void** run() {  System.*out*.println("BMW跑跑");  }  }  **class** Benz **extends** Car {  @Override  **public** **void** run() {  System.*out*.println("Benz跑跑");  }  } |

模拟spring工厂：

|  |
| --- |
| **import** java.io.BufferedReader;  **import** java.io.FileReader;  **import** java.io.IOException;  **import** java.lang.reflect.Constructor;  **import** java.lang.reflect.Field;  **class** Student{    **private** **int** id;    **private** String name;    **public** Student(**int** id , String name){  **this**.id = id;  **this**.name = name;  }    **public** Student(){    }  **public** **int** getId() {  **return** id;  }  **public** **void** setId(**int** id) {  **this**.id = id;  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }    @Override  **public** String toString() {  **return** **this**.id +"-"+**this**.name;  }    }  **class** Person{    **private** **int** age;    **public** Person(){    }          @Override  **public** String toString() {  **return** **this**.age+"";  }  }  **public** **class** Demo1 {    **public** **static** **void** main(String[] args) **throws** Exception {  Object o = *getInstance*();  System.*out*.println(o);    }    **public** **static** Object getInstance() **throws** Exception{  FileReader fileReader = **new** FileReader("src/info.txt");  BufferedReader bufferedReader = **new** BufferedReader(fileReader);  String line = bufferedReader.readLine();  Class clazz = Class.*forName*(line);  Constructor c = clazz.getConstructor(**null**);  Object c1 = c.newInstance(**null**);  **while**((line=bufferedReader.readLine())!=**null**){  String[] datas = line.split("=");  Field f = clazz.getDeclaredField(datas[0]);  f.setAccessible(**true**);  **if**(f.getType()==**int**.**class**){  f.set(c1, Integer.*parseInt*(datas[1]));  }**else**{  //f.setAccessible(true);  f.set(c1,datas[1]);  }  }  **return** c1;  }        } |