# 第一章：java开发入门

## 一．计算机的基本概念

计算机（Computer）：是一种能够按照程序运行，自动，高效处理海量数据的现代化智能电子设备。由硬件与软件组成，没有安装任何软件的的计算机称为裸机，常见的形式有台式计算机、笔记本计算机、大型计算机。

软件（Software）：是使用计算机过程中必不可少的东西，软件可以使计算机安照预定好的顺序完成特定的功能。

一台计算机由：主板、CPU、内存、电源、显卡、硬盘等组成

## 二．计算机语言的发展

### 1.机器语言：

用二进制代码指令表达的计算机语言，CPU可以直接识别，指令是0和1组成的一串代码，他们有一定的位数，并分成若干段，每段表示不同的含义

|  |
| --- |
| 0000,0000,000000010000 代表 加载（LOAD）A, 16  0000,0001,000000000001 代表 加载（LOAD）B, 1 |

### 2. 汇编语言：

汇编语言是使用一些特殊的符号来代替机器语言的二进制码，计算机不能直接识别，需要用一种软件将汇编语言翻译成机器语言。

### 3.高级语言：

高级语言是使用普通英语进行编写源代码，通过编译器将源代码翻译成计算机直接识别的机器语言，之后再由计算机执行。c语言，java，pascal，basic，perl，C++， C#，等等

### 4. Dos常用的命令

（1）.dir

无参数：查看当前所在目录的文件和文件夹。

/s：查看当前目录以及其所有子目录的文件和文件夹。

/a：查看包括隐含文件的所有文件。

/ah：只显示出隐含文件。

/w：以紧凑方式（一行显示5个文件）显示文件和文件夹。

/p：以分页方式（显示一页之后会自动暂停）显示。

（2）.cd 目录名 进入下一个目录

cd \ 退回到根目录

cd .. 退回到上一级目录

（3）.md 目录名 新建一个目录

（4）.rd 目录名 删除特定文件夹

（5）.cls 清楚屏幕

（6）. copy 路径\文件名 路径\文件名 ：把一个文件拷贝到另一个地方。

（7）. move 路径\文件名 路径\文件名 ：把一个文件移动（就是剪切+复制）到另一个地方。

（8）. del 文件名：删除一个文件。

del \*.\*：删除当前文件夹下所有文件。

（9）. deltree 删除文件夹和它下面的所有子文件夹还有文件

（10）. type 文本文件名：显示出文本文件的内容

（11）. ren 旧文件名 新文件名：改文件名

## 三．java语言的发展史

SUN公司于1995 年推出

1991 年Sun 公司的詹姆斯·高斯林（James Gosling）等人开始开发名称为Oak 的语言。希望用于控制嵌入在有线电视交换盒、PDA 等的微处理器

Internet 使Java 成为网上最流行的编程语言,Java 对Internet 的影响也意义深远

1994 年将Oak 语言更名为Java

Java 既安全、可移植，又可跨平台，而且人们发现它能够解决Internet 上的大型应用问题

1996 年获得1亿美金的投资

1998 年提出JDK1.2，更名为Java2

之后推出JDK1.3，JDK1.4

2005 年更名为JDK 5.0，是对以前版本最大改进

2007 年JDK6.0

2009年4月Oracle以74亿美元收购SUN

2011年7月由Oracle正式发布Java7

2014年3月18日发布Java8正式版

## 四．JDK下载安装及环境变量配置

JDK（Java Development Kit）Java开发工具包

JRE（Java Runtime Environment）Java运行环境

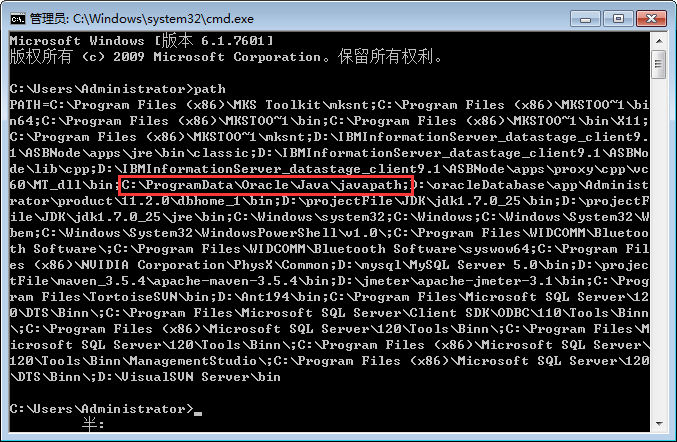
下载地址：<http://www.oracle.com/technetwork/java/archive-139210.html>

JDK的安装

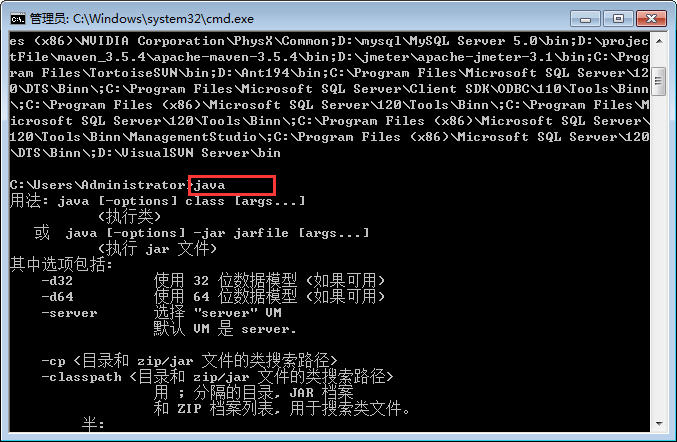
src.zip 为java的类库，其中lang文件里面的类是不用import引入的，如System



在JDK安装成功后，Oracle会默认在环境变量中pash变量下添加一个变量值，同时在相应的目录下添加了几个java简单的命令



现在java运行环境已经配置成功，java命令就可以使用



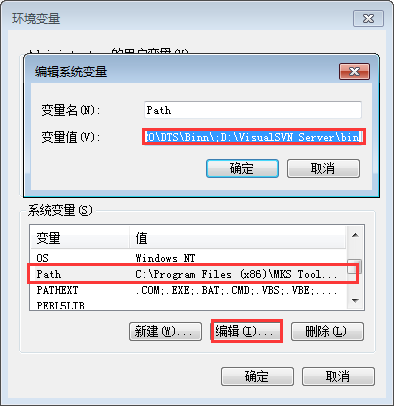
由于简单的命令完全不能满足使用，并且没有编译命令，所以需要将jdk中bin下的所有命令配置进去。复制bin所在目录，

如：D:\projectFile\JDK\jdk1.7.0\_25\bin

**我的电脑/计算机——右击属性——高级系统设置——高级——环境变量**

点击系统变量中的Path——编辑 添加到变量值中 注意后面要加分号

由于bin中包含java命令 所以Oracle生成的简单命令就可以删除了



## 五．Java体系与特点

Java SE：Java Platform，Standard Edition

标准版：各应用平台的基础，桌面开发和低端商务应用的解决方案。

Java EE：Java Platform，Enterprise Edition

企业版：以企业为环境而开发应用程序的解决方案

Java ME ：Java Platform, Micro Edition

微型版：致力于消费产品 和嵌入式设备的最佳解决方案

特点

一种纯面向对象的编程语言。

一种与平台无关（跨平台）的语言。（它提供了在不同平台下运行的解释环境）

一种健壮的语言，吸收了C/C++语言的优点。

有较高的安全性。(自动回收垃圾，强制类型检查，取消指针)

## 六．程序的开发步骤

第一步：编写程序 编写HelloWord.java文件

|  |
| --- |
| 注：在文本文档中编辑java文件，类名用public修饰时文件名称必须与类名保持一致，否则随意。 |

第二步：编译程序 通过编译器生成HelloWord.class文件

|  |
| --- |
| 在cmd中进入此文件所在的目录使用javac 命令进行编译：javac HelloWord.java  注：出现‘编码GBK的不可映射字符’时，javac -encoding  UTF-8 HelloWord.java |

第三步：运行程序 JVM(java运行平台)

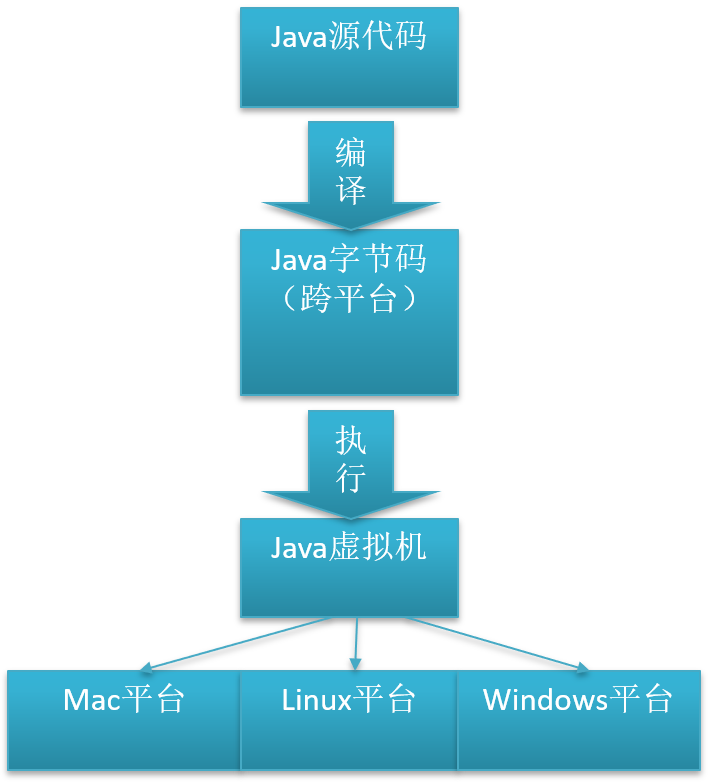
|  |
| --- |
| 在cmd中进入此文件所在的目录使用java命令进行运行：java HelloWord |

## 七．生成java类文档

|  |
| --- |
| 在cmd中进入此文件所在的目录使用javadoc命令进行运行：单个类文档javadoc HelloWord.java；所有类文档 javadoc \*.java。注意：类名必须被public修饰，否则回报‘找不到可以文档化的公共或受保护的类’的错误 |

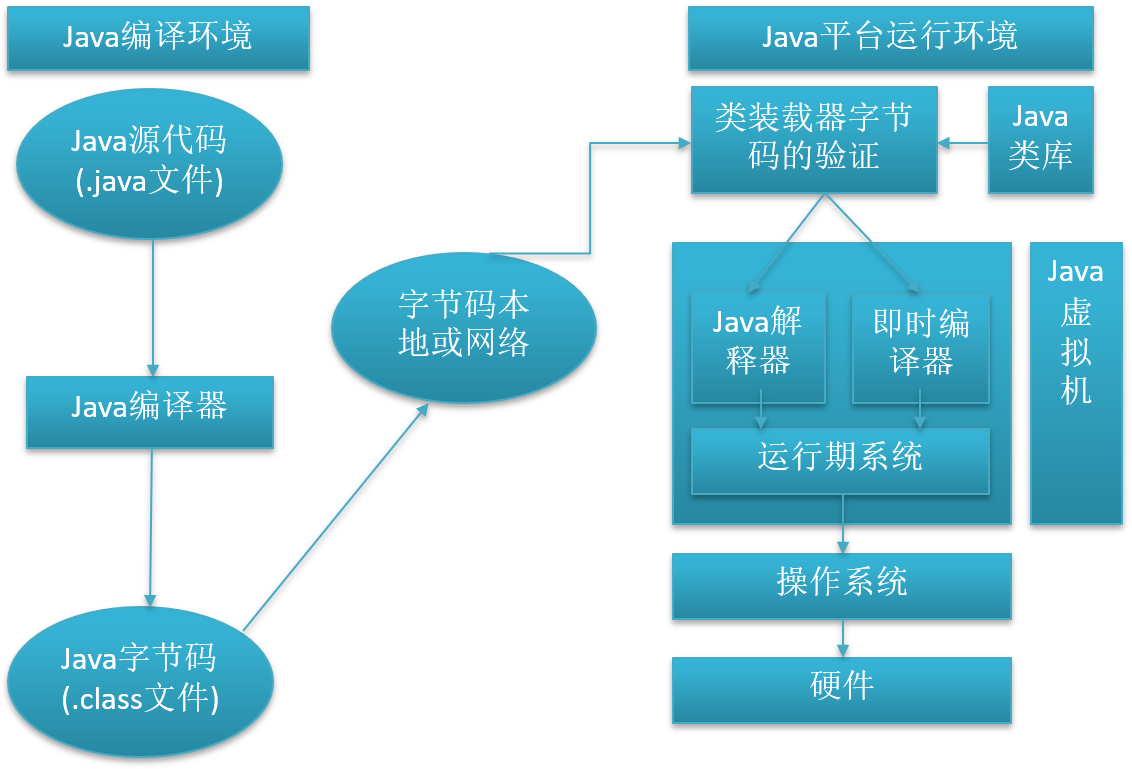
## 八．Java的跨平台原理

JRE（java运行环境）拥有window、Linux、Mac等不同平台的版本，在不同的平台安装对应的JRE，便可实现同一套class文件在不同平台上运行，从而解决了跨平台。



对于不同的运行平台，有不同的JVM。

JVM 屏蔽了底层运行平台的差别，实现了“一次编译，随处运行”



## 九．编辑器Notepad++的使用

第一步安装java编译运行插件NppExec

|  |
| --- |
| 在Notepad++软件中查看插件中有没有NppExec    如果没有点击Plugins Admin（插件管理），在avallable(可用)中找到，并安装，然后重新打开在插件中就可看到 |

第二步 配置编译运行

|  |
| --- |
| 点击‘插件’—‘NppExec’—‘Execute..’  Javac的配置信息为javac "$(FULL\_CURRENT\_PATH)"  Java的配置信息为java -cp "$(CURRENT\_DIRECTORY)" "$(NAME\_PART)"  将javac的配置信息放入command（命令）中，点击保存，输入script name(脚本名称) 再点击OK，java的配置同理 |
|  |

第三步 将执行的命令放入菜单

|  |
| --- |
| 点击‘插件’—‘ NppExec’—‘ Advanced Options…’(高级选项)  如图操作  最后点击OK  会出现提示must be restarted to apply some of the options（必须重新启动才能应用某些选项） |

注意：编译java文件时报‘编码GBK的不可映射字符’错误，需要将编码改为使用ANSI编码。如果在文本中已经有了代码，应点击转为ANSI编码

## 十．java反编译工具Cavaj Java Decompiler

|  |
| --- |
| 下载地址：http://cavaj-java-decompiler.en.softonic.com/ |

## 十一. 垃圾回收器（GC）

不再使用的内存空间应当进行回收——垃圾回收。在C/C++等语言中，由程序员负责回收无用内存。

垃圾回收器在Java 程序运行过程中自动启用，用于检查并释放那些可以被释放的存储空间，程序员无法精确控制和干预。

# 第二章：java编程基础

## 一．变量与标识符

1. 变量就是系统为程序分配的一块内存单元，用来存储各种类型的数据。根据所存储的数据类型的不同，有各种不同类型的变量。变量名代表这块内存中的数据 。

2变量分类

按所属的数据类型划分：

基本数据类型变量 引用数据类型变量

按被声明的位置划分：

局部变量：方法或语句块内部定义的变量

成员变量：方法外部、类的内部定义的变量

3. Java对包、类、方法、参数和变量等要素命名时使用的字符序列称为标识符。

4.标识符的命名规则

由字母、数字、下划线（\_）和美元符号（$）组成。

不能以数字开头。

区分大小。

长度无限制。

不能是Java中的关键字和保留关键字。

标识符命名习惯：驼峰命名法、见名知意。以单词或单词组合来命名

## 二．八种基本数据类型

1. 字节(Byte)是：计算机文件大小的基本计算单位。

1个字节(Byte) = 8个位（Bit)

单位换算：

1byte = 8bit

1KB = 1024byte

1MB = 1024KB

1GB = 1024MB

1TB=1024G

1PB=1024TB

**说明：**  
在内存计算过程中，字节少的数据运算速度更快;

在硬盘存储中，字节少的数据类型也可充分存入更多的数据

2.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 数据类型 | 关键字 | 占用字节 | 取值范围 | 默认值 |
| 布尔型 | boolean | 不定值 | true,false | false |
| 字节型 | byte | 1 | -128~127 | 0 |
| 短整型 | short | 2 | -32768~32767 | 0 |
| 整型 | int | 4 | -2147483648~2147483647 | 0 |
| 长整型 | long | 8 | -9223372036854775808~9223372036854775807 | 0 |
| 字符型 | char | 2 | 0~65535 | ‘\u0000’ |
| 单精度浮点型 | float | 4 | 1.4E-45~3.4028235E38 | 0.0F |
| 双精度浮点型 | double | 8 | 4.9E-324~1.7976931348623157E308 | 0.0D |

3.基本数据类型转换

**自动类型转换：**容量小的类型自动转换成容量大的数据类型，byte,short,int，它们三者在计算时会转换成int类型；如果把int值转换为float值，或者long转换为double值，不需要强制转换，但可能丢失精度.

**强制类型转换：**

容量大的类型转换成容量小的数据类型时，要加上强制转换符。

long n = 100L;

int i = (int)n;

有可能造成精度降低或数据溢出，使用时要小心。

boolean 类型不能转换成任何其它数据类型。

## 三．进制与转换

计算机进制通常有：十进制、二进制、八进制、十六进制

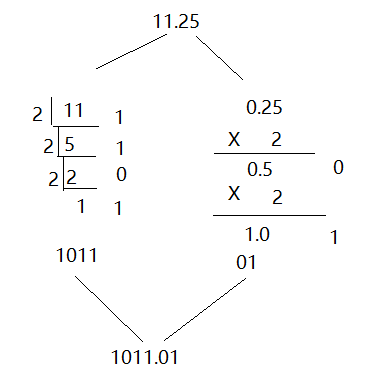
十进制转二进制

1011.01 = 1×23＋0×22＋1×21＋1×20＋0×2-1＋1×2-2 = 8＋0＋2＋1＋0＋0.25  
 =11.25

二进制转十进制

整数部分："除2取余，逆序排列"法

小数部分："乘2取整，顺序排列"法



## 四．关键字

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| abstract | assert | boolean | break | byte |
| case | catch | char | class | const |
| continue | default | do | double | else |
| enum | extends | final | finally | float |
| for | goto | if | implements | import |
| instanceof | int | interface | long | native |
| new | package | private | protected | public |
| return | strictfp | short | static | super |
| switch | synchronized | this | throw | throws |
| transient | try | void | volatile | while |

## 五．转义字符

|  |  |
| --- | --- |
| \n | 换行(LF) ，将当前位置移到下一行开头 |
| \r | 回车(CR) ，将当前位置移到本行开头 |
| \t | 水平制表(HT) （跳到下一个TAB位置） |
| \\ | 代表一个反斜线字符''\' |
| \' | 代表一个单引号（撇号）字符 |
| \" | 代表一个双引号字符 |

## 六．运算符与优先级

### 1.算数运算符：

|  |  |  |  |
| --- | --- | --- | --- |
| **运算符** | **描述** | **示例** | **结果** |
| + | 加法 | 5+5 | 10 |
| - | 减法 | 5-3 | 2 |
| \* | 乘法 | 2\*3 | 6 |
| / | 除法 | 10/3 | 3 |
| % | 取余（取模） | 10%3 | 1 |
| ++ | 自增（前，后） |  |  |
| -- | 自减（前，后） |  |  |

b = 3; a = b++; //先赋值后自增 a = 3; b = 4;

b = 3; a = ++b; //先自增后赋值 a = 4; b = 4;

b = 3; a = b--; //先赋值后自减 a = 3; b = 2;

b = 3; a = --b; //先自减后赋值 a = 2; b = 2;

### 2赋值运算符：

|  |  |  |  |
| --- | --- | --- | --- |
| **运算符** | **描述** | **示例** | **结果** |
| = | 赋值 | a=10 | a=10 |
| += | 加等于 | a=1, a+=3 | a=4 |
| -= | 减等于 | a=1, a-=3 | a=-2 |
| \*= | 乘等于 | a=1, a\*=3 | a=3 |
| /= | 除等于 | a=10, a/=3 | a=3 |
| %= | 模等于 | a=10, a%=3 | a=1 |

注意：a+=1 不等同与a = a + 1 这是涉及到一个数据类型的问题；当a为short类型时：a =（short）a+（int）1这里的1为int类型会报错，而a+=1就相当于a = （short）（a+1）不会报错，减、乘、除、模同理。

### 3.关系运算符：

|  |  |  |  |
| --- | --- | --- | --- |
| **运算符** | **描述** | **示例** | **结果** |
| == | 相等于 | a=1,a==10 | false |
| != | 不等于 | a=1, a!=3 | true |
| < | 小于 | a=1, a<3 | true |
| > | 大于 | a=1, a>3 | false |
| <= | 小于等于 | a=10, a<=3 | false |
| >= | 大于等于 | a=10, a>=3 | true |

### 4.逻辑运算符：

|  |  |  |  |
| --- | --- | --- | --- |
| **运算符** | **描述** | **示例** | **结果** |
| & | 与 | false & true | false |
| | | 或 | false & true | true |
| ^ | 异或 | true ^ false | true |
| ! | 非 | ! true | false |
| && | 短路与 | false && true | false |
| || | 短路或 | false || true | true |

&与&&的区别

&在逻辑运算时分别计算表达式两边的结果，在做&运算;&在做位数运算时，表示的是按位与。

&&短路与运算，先计算左边的表达式，如果是false，那么就不用计算右边的表达式，直接返回false；如果左边的表达式结果是true，在计算右边的表达式，如果右边的表达式为true，结果为true，否则为false。

### 5.位运算符：

对两个操作数中的每一个二进制位都进行运算

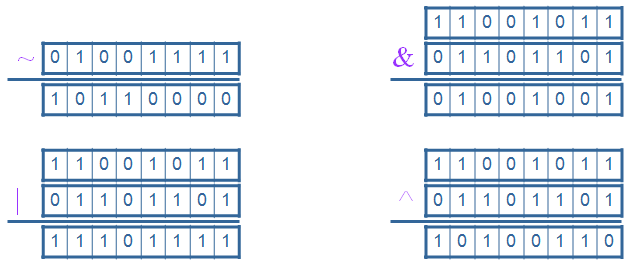
位运算符功能：

1、按位取反 ~

2、按位与 &

3、按位或 |

4、按位异或 ^



### 6.位移运算符：

左移："a<<b; "将二进制形式的a逐位左移b位，最低位空出的b位补0

带符号右移："a>>b; "将二进制形式的a逐位右移b位，最高位空出的b位补原来的符号位

无符号右移："a>>>b;"将二进制形式的a逐位右移b位，最高位空出的b位补0

位移运算符正数示例：

**100 00000000 00000000 00000000 01100100**

**100<<3 00000000 00000000 00000011 00100000**

**100>>3 00000000 00000000 00000000 00001100**

**100>>>3 00000000 00000000 00000000 00001100**

位移运算符负数示例：

**-100 11111111 11111111 11111111 10011100**

**-100<<3 11111111 11111111 11111100 11100000**

**-100>>3 11111111 11111111 11111111 11110011**

**-100>>>3 00011111 11111111 11111111 11110011**

面试题：以最快的速度计算8\*4结果？

00000000 00000000 00000000 00001000 8

10000000 00000000 00000000 00100000 32

使用位移的方法是最快的。

int result = 8<<2; （8\*2 ² 左移幂数2,也就是两位）

左移算法：位移数作为2的次幂与操作数相乘

10<<2=40 （与乘法相反）

右移算法：操作数除以位移数的2次幂

### 7.数据交换的方法

int a = 3;

int b = 5;

方法一：int c = a; a = b; b = c;

方法二：a = a + b - (b = a); 理解为：后面的（b = a）为一个表达式b的值变成了a, 程序就变成了a = a + b – a

方法三：a = a + b; b = a – b; a = a – b;

方法四：乘、除也可以

方法五：异或方法（a、b不能相同）a = a ^ b; b = a ^ b; a = a ^ b;

### 8.三目运算符

X ? Y : Z

### 9.运算符优先级



### 10.分支语句

#### （1.）if else

if(表达式){

执行语句;

}else{

执行语句;

}

**分支嵌套：**

if(表达式1){

执行语句1;

}else if(表达式2){

执行语句2;

}else if(表达式3){

执行语句3;

}else{//可有可无

执行语句4;

}

#### （2.）switch case default

**switch语句有关规则**

1、switch括号中的表达式的返回值必须是下述几种类型之一：int, byte, char, short,确定的结果，jdk1.7后支持String；

2、case子句中的取值必须是常量，且所有case子句中的取值应是不同的；

3、default子句是可选的；

4、break语句用来在执行完一个case分支后使程序跳出switch语句块；如果case后面没有写break则直接往下面执行！

5、case后面的执行体可写{ }也可以不写{ }

**语法：**

switch(表达式){

case 取值1: 语句块1;

break;

case 取值n: 语句块n;

break;

default: 语句块n+1;

break;

}

**Java 7 switch支持String的实现原理:**

这个新特性是在编译器这个层次上实现的，在编译的过程中，编译器会根据源代码的含义进行转换,不同的Java编译器可能采用不同的方式来转换比如：如果switch语句中只包含一个case语句，那么就可以简单的将其转换成一个if语句。

如果包含一个case和一个default语句，就可以转换成if-else语句。而对于复杂的情况（多个case语句），也可以转换成Java 7 之前的switch语句，只不过使用字符串的哈希值（整数）作为switch语句表达式的值

### 11.循环语句

#### （1.）while

**符合条件，循环继续执行；否则，循环退出**

**特点：先判断，再执行**

**语法：**

while(条件表达式){

//语句块；

}

#### （2.）do while

**先执行一遍循环操作，符合条件，循环继续执行；否则，循环退出**

**特点：先执行，再判断**

**语法：**

do {

循环操作

}while ( 循环条件 );

#### （3.）for

**语法：**

for(初始化参数;判断条件 ;更新循环变量){

循环体;

}

标准写法

for(int i = 0; i < 5; i++) {

System.out.println(i);

}

## 七．ASCII码

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **ASCII码表** | | | | | | | |
| 信息在计算机上是用二进制表示的，这种表示法让人理解就很困难。因此计算机上都配有输入和输出设备，这些设备的主要目的就是，以一种人类可阅读的形式将信息在这些设备上显示出来供人阅读理解。为保证人类和设备，设备和计算机之间能进行正确的信息交换，人们编制的统一的信息交换代码，这就是ASCII码表，它的全称是“美国信息交换标准代码”。 |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **八进制** | **十六进制** | **十进制** | **字符** | **八进制** | **十六进制** | **十进制** | **字符** |
| 00 | 00 | 0 | nul | 100 | 40 | 64 | @ |
| 01 | 01 | 1 | soh | 101 | 41 | 65 | A |
| 02 | 02 | 2 | stx | 102 | 42 | 66 | B |
| 03 | 03 | 3 | etx | 103 | 43 | 67 | C |
| 04 | 04 | 4 | eot | 104 | 44 | 68 | D |
| 05 | 05 | 5 | enq | 105 | 45 | 69 | E |
| 06 | 06 | 6 | ack | 106 | 46 | 70 | F |
| 07 | 07 | 7 | bel | 107 | 47 | 71 | G |
| 10 | 08 | 8 | bs | 110 | 48 | 72 | H |
| 11 | 09 | 9 | ht | 111 | 49 | 73 | I |
| 12 | 0a | 10 | nl | 112 | 4a | 74 | J |
| 13 | 0b | 11 | vt | 113 | 4b | 75 | K |
| 14 | 0c | 12 | ff | 114 | 4c | 76 | L |
| 15 | 0d | 13 | cr | 115 | 4d | 77 | M |
| 16 | 0e | 14 | so | 116 | 4e | 78 | N |
| 17 | 0f | 15 | si | 117 | 4f | 79 | O |
| 20 | 10 | 16 | dle | 120 | 50 | 80 | P |
| 21 | 11 | 17 | dc1 | 121 | 51 | 81 | Q |
| 22 | 12 | 18 | dc2 | 122 | 52 | 82 | R |
| 23 | 13 | 19 | dc3 | 123 | 53 | 83 | S |
| 24 | 14 | 20 | dc4 | 124 | 54 | 84 | T |
| 25 | 15 | 21 | nak | 125 | 55 | 85 | U |
| 26 | 16 | 22 | syn | 126 | 56 | 86 | V |
| 27 | 17 | 23 | etb | 127 | 57 | 87 | W |
| 30 | 18 | 24 | can | 130 | 58 | 88 | X |
| 31 | 19 | 25 | em | 131 | 59 | 89 | Y |
| 32 | 1a | 26 | sub | 132 | 5a | 90 | Z |
| 33 | 1b | 27 | esc | 133 | 5b | 91 | [ |
| 34 | 1c | 28 | fs | 134 | 5c | 92 | \ |
| 35 | 1d | 29 | gs | 135 | 5d | 93 | ] |
| 36 | 1e | 30 | re | 136 | 5e | 94 | ^ |
| 37 | 1f | 31 | us | 137 | 5f | 95 | \_ |
| 40 | 20 | 32 | sp | 140 | 60 | 96 | ' |
| 41 | 21 | 33 | ! | 141 | 61 | 97 | a |
| 42 | 22 | 34 | " | 142 | 62 | 98 | b |
| 43 | 23 | 35 | # | 143 | 63 | 99 | c |
| 44 | 24 | 36 | $ | 144 | 64 | 100 | d |
| 45 | 25 | 37 | % | 145 | 65 | 101 | e |
| 46 | 26 | 38 | & | 146 | 66 | 102 | f |
| 47 | 27 | 39 | ` | 147 | 67 | 103 | g |
| 50 | 28 | 40 | ( | 150 | 68 | 104 | h |
| 51 | 29 | 41 | ) | 151 | 69 | 105 | i |
| 52 | 2a | 42 | \* | 152 | 6a | 106 | j |
| 53 | 2b | 43 | + | 153 | 6b | 107 | k |
| 54 | 2c | 44 | , | 154 | 6c | 108 | l |
| 55 | 2d | 45 | - | 155 | 6d | 109 | m |
| 56 | 2e | 46 | . | 156 | 6e | 110 | n |
| 57 | 2f | 47 | / | 157 | 6f | 111 | o |
| 60 | 30 | 48 | 0 | 160 | 70 | 112 | p |
| 61 | 31 | 49 | 1 | 161 | 71 | 113 | q |
| 62 | 32 | 50 | 2 | 162 | 72 | 114 | r |
| 63 | 33 | 51 | 3 | 163 | 73 | 115 | s |
| 64 | 34 | 52 | 4 | 164 | 74 | 116 | t |
| 65 | 35 | 53 | 5 | 165 | 75 | 117 | u |
| 66 | 36 | 54 | 6 | 166 | 76 | 118 | v |
| 67 | 37 | 55 | 7 | 167 | 77 | 119 | w |
| 70 | 38 | 56 | 8 | 170 | 78 | 120 | x |
| 71 | 39 | 57 | 9 | 171 | 79 | 121 | y |
| 72 | 3a | 58 | : | 172 | 7a | 122 | z |
| 73 | 3b | 59 | ; | 173 | 7b | 123 | { |
| 74 | 3c | 60 | < | 174 | 7c | 124 | | |
| 75 | 3d | 61 | = | 175 | 7d | 125 | } |
| 76 | 3e | 62 | > | 176 | 7e | 126 | ~ |
| 77 | 3f | 63 | ? | 177 | 7f | 127 | del |

# 第三章：方法与数组

## 一．方法的定义

方法（又叫函数）就是一段特定功能的代码块。方法提高程序的复用性和可读性。

**方法的格式**

语法：

访问权限 修饰符 [其他的修饰符 如static] 返回值类型 方法名(参数类型1 形参1,参数类型2 形参2,...){ //形参列表

//方法体

return 返回值;

}

方法的返回值

返回值：就是功能的结果，由return带给调用者。

注意：

1、若当前方法没有返回值类型，即返回值类型是void，那么当前方法中可以不写return

2、return即表示结束一个方法，也可以将返回值返回给调用当前方法的调用者

3、return返回值时一次只能返回一个值，不可以返回多个值

4、一个方法中可以有多个return，但被执行的只能有一个，所以需要判断

## 二．方法的重载

方法重载：overloading method

在类中可以创建多个方法，它们具有相同的名字，但具有不同的参数和不同的定义；

返回值不能作为重载的条件。

如：

public void method(int a){…}

public void method(char c){…}

## 三．java数组

### 1.java数组的定义

数组：一组能够存储相同数据类型值的变量的集合。

当我们有一组相同类型的数据需要存储，如果此时使用单个变量来存储，我们将要定义若干个变量名，这样将会非常繁琐，并不利于维护。

### 2.数组的赋值方式

第一种：int [] scores = new int[3];

scores[0] = 23;

scores[1] = 34;

scores[2] = 45;

第二种：int [] scores;

scores = new int[3];

第三种int [] scores = new int[]{56,78,98};

第四种int [] scores = {56,78,98};

### 3.数组的遍历

第一种for循环

int[] arr = {1,2,3,4,5,6,7};

for(int i = 0; i < arr.length; i++) {

System.out.println(arr[i]);

}

第二种foreach循环

int[] arr = {1,2,3,4,5,6,7};

for(int num : arr) {

System.out.println(num);

}

### 4.可变参数

public static void main(String[] args) {

print(1,2,3,4,5,6,7);

}

public static void print(int ...arr) {

for(int num : arr) {

System.out.println(num);

}

}

### 5.数组要注意的问题

空指针异常（NullPointerException）：当变量为空时，调用该变量的属性和方法

数组越界异常（ArrayIndexOutOfBoundsException ）：输出的数组数量多余对象本有的数组数量。

数组内存结构分析：数组是引用类型，会存放在堆内存中

### 6.多维数组

int[] [] arrs = {{12,23,34},{23,56,67},{34,45,56}};

### 7.排序算法

#### （1.）冒泡排序算法

冒泡排序算法的运作如下：（从后往前）

比较相邻的元素。如果第一个比第二个大，就交换他们两个。

对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对。在这一点，最后的元素应该会是最大的数。

针对所有的元素重复以上的步骤，除了最后一个。

持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。

相同元素的前后顺序并没有改变，所以冒泡排序是一种稳定排序算法。

|  |
| --- |
| int[] arr = {100,54,83,3,6,9,2};  int len = arr.length;  for(int i = 0; i < len-1; i++) {  for(int j = 0; j < len-i-1; j++){  if(arr[j] > arr[j+1]) {  int c = arr[j];  arr[j] = arr[j+1];  arr[j+1] = c;  }  }    } |

#### （2.）选择排序算法

每一趟从待排序的数据元素中选出最小（或最大）的一个元素，顺序放在已排好序的数列的最后，直到全部待排序的数据元素排完。 选择排序是不稳定的排序方法。

|  |
| --- |
| int[] arr = {10,54,83,3,6,9,45};  int len = arr.length;  int c = 0;  for(int i = 0; i < len-1; i++) {  c = i;  for(int j = i+1; j < len; j++){  if(arr[c] > arr[j]) {  c = j;  }  }  if(c != i) {  arr[c] = arr[c] + arr[i];  arr[i] = arr[c] - arr[i];  arr[c] = arr[c] - arr[i];  }  } |

#### （3.）直接插入排序算法

（从后向前找到合适位置后插入）

基本思想：每步将一个待排序的记录，按其顺序码大小插入到前面已经排序的子序列的合适位置（从后向前找到合适位置后），直到全部插入排序完为止

|  |
| --- |
| int[] arr = {10,54,83,3,6,9,45};  int len = arr.length;  for(int i = 1; i < len; i++) {  int temp = arr[i];  int j = 0;  for(j = i-1; j >= 0; j--){  if(temp < arr[j]) {  arr[j+1] = arr[j];  } else {  break;  }  }  if(arr[j+1] != temp) {  arr[j+1] = temp;  }  } |

#### （4.）二分法查找（折半查找）：

前提是在已经排好序的数组中，通过将待查找的元素与中间索引值对应的元素进行比较，若大于中间索引值对应的元素，去右半部分查找，否则，去左半部分查找。依此类推。直到找到为止；找不到返回一个负数。

|  |
| --- |
| int[] arr = {10,20,30,40,50,60,70};  int key = 60;  int start = 0;  int end =arr.length-1;  while(start <= end) {  int mobile = (start + end)/2;  if(arr[mobile] > key) {  end = mobile - 1;  } else if(arr[mobile] < key) {  start = mobile + 1;  } else {  System.out.println(start+","+end+","+arr[mobile]);  break;  }  } |

### 8.Array 工具类

常用方法：

使用二分法查找

Arrays.binarySearch(int[] array, int value);

数组内容转成字符串的形式输出

Arrays.toString(int[] array);

数组排序

Arrays.sort(int[] array);

复制指定的数组

Arrays.copyOf(int[] array, int length);

Arrays.copyOf(int[] array, int from, int to);

System.arraycopy(Object src, int srcPos, Object dest, int destPos, int length)效率比较高

判断两个数组是否相等

Arrays.equels();

使用指定元素填充数组

Arrays.fill();

# 第四章：面向对象

## 一．面向对象的基本概念

1. **什么是面向对象：**

面向对象是一种编程思想。

面向对象是一种思考问题的思维方式。

## 二．类与对象

### 1、什么是类？

类是：分类、类别。

通过分类，我们可以区别不同的事物种类，在日常生活当中，我们常常这样做。

所以，类是一组具有相同特性（属性）与行为（方法）的事物集合

### 2、类与对象的关系

类表示一个共性的产物，是一个综合的特征，而对象，是一个个性的产物，是一个个体的特征。

类由属性和方法组成：

· 属性：就相当于一个个的特征

· 方法：就相当于人的一个个的行为，例如：说话、吃饭、唱歌、睡觉

### 3. 对象的定义：

一个类要想真正的进行操作，则必须依靠对象，对象的定义格式如下：

类名称 对象名称 = new 类名称() ;

按照以上的格式就可以产生对象了。

### 4.对象内存分析

new 关键字表示创建一个对象

new 关键字表示实例化对象

new 关键字表示申请内存空间

#### （1.）对象在内存中的结构：

Horse horse= null;

栈内存

horse

堆内存

null

horse = new Horse();

栈内存

horse（0x0011）

new

堆内存

name=null

age=0

horse.name = “小白”;

栈内存

horse（0x0011）

堆内存

name=“小白”

age=0

horse.age = 4;

栈内存

horse（0x0011）

堆内存

name=“小白”

age=4

#### （2.）在内存中创建多个对象：

Horse horse1=null

Horse horse2=null

栈内存

horse1

horse2

堆内存

null

null

horse1 = new Horse()

horse2 = new Horse()

new

栈内存

horse1（0x0011）

horse2（0x0022）

new

堆内存

name=null

age=0

name=null

age=0

horse1.name=“小白”

horse1.age=4

栈内存

horse1（0x0011）

horse2（0x0022）

堆内存

name=“小白”

age=4

name=null

age=0

horse2.name=“小黑”

horse2.age=5

栈内存

horse1（0x0011）

horse2（0x0022）

堆内存

name=“小白”

age=4

name=”小黑”

age=5

#### （3.）声明两个对象，一个实例化，一个没实例化

Horse horse1=null

Horse horse2=null

栈内存

horse1

horse2

堆内存

null

null

horse1 = new Horse()

null

栈内存

horse1（0x0011）

horse2

new

堆内存

name=null

age=0

horse1.name=“小白”

horse1.age=4

栈内存

horse1（0x0011）

horse2

堆内存

name=“小白”

age=4

null

horse2 = horse1

horse2.name=“小黑”

栈内存

horse1（0x0011）

horse2（0x0011）

堆内存

name=“小黑”

age=4

#### （4.）实例化两个对象，对象间进行赋值

Horse horse1=null

Horse horse2=null

栈内存

horse1

horse2

堆内存

null

null

horse1 = new Horse()

horse2 = new Horse()

new

栈内存

horse1（0x0011）

horse2（0x0022）

new

堆内存

name=null

age=0

name=null

age=0

horse1.name=“小白”

horse1.age=4

horse2.name=”小黑”

栈内存

horse1（0x0011）

horse2（0x0022）

堆内存

name=“小白”

age=4

name=“小黑”

age=0

horse2 = horse1

horse2.name=”黑黑”

栈内存

horse1（0x0011）

horse2（0x0011）

堆内存

name=“黑黑”

age=4

name=”小黑”

age=5

垃圾

## 三．封装性

**封装性的概念**

1、封装性是面向对象思想的三大特征之一。

2、封装就是隐藏实现细节，仅对外提供访问接口。

封装有：

属性的封装、方法的封装、类的封装、组件的封装、模块化封装、系统级封装…

**封装的好处**

1、模块化

2、信息隐藏

3、代码重用

4、插件化易于调试

5、具有安全性

**封装缺点：**

会影响执行效率

**封装之前：**

class Person{

String name;

int age;

}

**封装之后：**

class Person{

//属性是成员变量

private String name;

private int age;

//参数及方法内定义的变量是局部变量

public void setName(String name){

this.name = name;

}

public String getName(){

return name;

}

}

#### 1、成员变量和局部变量

1.1、在类中的位置不同

成员变量：在类中定义

局部变量：在方法中定义或者方法的参数

1.2、在内存中的位置不同

成员变量：在堆内存(成员变量属于对象，对象进堆内存)

局部变量：在栈内存(局部变量属于方法，方法进栈内存)

1.3、生命周期不同

成员变量：随着对象的创建而存在，随着对象的销毁而消失

局部变量：随着方法的调用而存在，随着方法的调用完毕而消失

1.4、初始化值不同

成员变量：有默认初始化值，引用类型默认为null

局部变量：没有默认初始化值，必须定义，赋值，然后才能使用

注意：

局部变量名称可以和成员变量名称一样，在方法中使用的时候，采用的是就近原则。

## 四．构造方法

什么是构造方法

1、构造方法就是类构造对象时调用的方法，用于对象的初始化工 作

2、构造方法是实例化一个类的对象时，也就是new 的时候，最先调用的方法。

构造方法的定义：

构造方法是在类中定义的，构造方法的定义格式：方法名称与类名称相同，无返回值类型的声明。

对象的实例化语法：

Dog dog = new Dog(); //new Dog后面有个括号，带括号表示调用了方法，此时调用的方法就是构造方法了

构造方法重载：

**无参构造方法：**

|  |
| --- |
| public Dog(){} |

**带一个参数的构造方法：**

|  |
| --- |
| public Dog(String name){  this.name = name;  } |

**带多个参数的构造方法：**

|  |
| --- |
| public Dog(String name,int age){  this.name = name;  this.age = age;  } |

（1）构造方法名称与类名相同，没有返回值声明（包括 void）

（2）构造方法用于初始化数据（属性）

（3）每一个类中都会有一个默认的无参的构造方法

（4）如果类中有显示的构造方法，那么默认构造方法将无效

（5）如果有显示的构造方法，还想保留默认构造 方法，需要显示的写出来。

（6）构造方法可以有多个，但参数不一样，称为构造方法的重载

（7）在构造方法中调用另一个构造方法，使用this(...)，该句代码必须在第一句。

|  |
| --- |
| public Dog(String name){  this.name = name;  }  public Dog(String name,int age){  this(name);  this.name = name;  this.age = age;  } |

（8）构造方法之间的调用，必须要有出口。

（9）给对象初始化数据可以使用构造方法或setter方法，通常情况下，两者都会保留。

（10）一个好的编程习惯是要保留默认的构造方法。（为了方便一些框架代码使用反射来创建对象）

（11）private Dog(){}，构造方法私有化，当我们的需求是为了 保正该类只有一个对象时。

什么时候一个类只需要一个对象？比如，工具类（没有属性的类，只有行为）并且该工具对象被频繁使用。权衡只用一个对象与产生多个对象的内存使用，来确定该类是否要定义为只需要一个对象。

## 五．this关键字

在Java基础中，this关键字是一个最重要的概念。使用this关键字可以完成以下的操作：

· 调用类中的属性

· 调用类中的方法或构造方法

|  |
| --- |
| 调用类中的方法：  class Hero{  private String name = "张三";  public String getName() {  return name;  }  public void input() {  System.out.println(this.getName());  }  }  构造方法  public Dog(String name,int age){  this(name);  this.name = name;  this.age = age;  } |

· 表示当前对象

|  |
| --- |
| class Wulin{  public static void main(String[] args) {  Hero hero = new Hero();  String name = hero.getName();  System.out.println(name);  }  }  class Hero{  private String name = "张三";  public String getName() {  return this.name;  }  } |

## 六．值传递与引用传递

示例一：值传递

|  |
| --- |
| public class ValueDemo{  public static void main(String[] args){  int x = 10;  method(x) ;  System.out.println(“x=”+x) ;  }  public static void method(int mx){  mx = 20;  }  } |

示例二：引用传递

|  |
| --- |
| public class RefDemo1{  public static void main(String[] args){  Duck d = new Duck();  method(d) ;  System.out.println(“Duck age = ”+d.age) ;  }  public static void method(Duck duck){  duck.age = 5;  }  }  class Duck{  int age = 2; //省略封装  } |

示例三：String传递

|  |
| --- |
| public class RefDemo2{  public static void main(String[] args){  String name = “小飞”;  method(name) ;  System.out.println(“name=”+name) ;  }  public static void method(String sname){  sname = “小备”;  }  } |

示例四：String传递

|  |
| --- |
| public class RefDemo3{  public static void main(String[] args){  Person p = new Person();  method(p) ;  System.out.println(“person name= ”+p.name) ;  }  public static void method(Person p ){  p.name = “备备”;  }  }  class Person{  String name = “飞飞”; //省略封装  } |

## 七．对象的一对一关系

案例

|  |
| --- |
| /\*\*  定义一个英雄类  \*/  class Hero{  private String name; //英雄名称  private int age; //英雄年龄  private Weapon weapon; //该英雄所使用的武器  }  /\*\*  定义一个武器类  \*/  class Weapon{  private String weaponName; //武器名称  private String source; //武器来源  private Hero hero; //该武器属于哪位英雄  } |

## 八．static关键字

### 1、static关键字的作用：

**1.1、使用static关键字修饰一个属性**

声明为static的变量实质上就是全局变量

**1.2、使用static关键字修饰一个方法**

通常，在一个类中定义一个方法为static，那就是说，无需本类的对象即可调用此方法

|  |
| --- |
| class Test{  public static void main(String[] args) {  Horse.input();  }  }  class Horse{  public static void input() {  System.out.println("可以直接使用！！");  }  } |

1.3、使用static关键字修饰一个类（内部类）

### 2.static的内存分析

|  |
| --- |
| class Horse{  private String name;  private int age;  private static String country;  } |

Horse horse1=null

Horse horse2=null

全局数据区

栈内存

horse1

horse2

堆内存

null

null

horse1 = new Horse()

horse2 = new Horse()

全局数据区

new

栈内存

horse1（0x0011）

horse2（0x0022）

new

堆内存

name=null

age=0

name=null

age=0

country=null

horse1.name=“小白” horse2.name=“小黑”

horse1.age=4 horse2.age=5

horse1.country=”中国” horse2.country=”中国”

全局数据区

栈内存

horse1（0x0011）

horse2（0x0022）

堆内存

name=“小白”

age=4

name=”小黑”

age=5

country=”中国”

注：任何一个修改全局区的数据，其他的都会修改。

全局数据区放的是static定义的数据，static定义的属性不受类的限制，static定义的方法不能调用非static定义的属性或方法，非static定义的方法可以任意调用所有的static属性或方法

## 九．main方法分析

主方法：

public static void main(String[] args){

//代码块

}

public：公有的，最大的访问权限

static：静态的，无需创建对象

void:：表示没有返回值，无需向JVM返回结果

main：方法名，固定的方法名

String[] args：表示参数为字符串数组，可以在调用方法时传入参数

## 十．单例设计模式

单例设计模式：保证一个类仅有一个实例，并提供一个访问它的全局访问点。

1、构造方法私有化

2、声明一个本类对象

3、给外部提供一个静态方法获取对象实例

两种实现方式：

### 饿汉式

生命周期：在类被加载后，对象被创建，到程序结束后释放，占用内存时间长，效率高。

|  |
| --- |
| class Test{  public static void main(String[] args) {  Singleton s = Singleton.getInstance();  s.print();  }  }  class Singleton {  private Singleton(){};  private static Singleton s = new Singleton();  public static Singleton getInstance() {  return s;  }  public void print() {  System.out.println("测试方法");  }  } |

### 懒汉式

生命周期：在第一次调用 getInstance()方法时，对象被创建，程序结束后释放，占用内存时间短，效率低。（建议使用）

|  |
| --- |
| class Test{  public static void main(String[] args) {  Singleton s = Singleton.getInstance();  s.print();  }  }  class Singleton {  private Singleton(){};  private static Singleton s;  public static Singleton getInstance() {  if(s == null) {  s = new Singleton();  }  return s;  }  public void print() {  System.out.println("测试方法");  }  } |

## 十一.对象数组案例

数组是一种线性数据结构

数组不适合做删除插入等操作，适合添加，查找，遍历

|  |
| --- |
| import java.util.Arrays;  class Test{  public static void main(String[] args) {  ManageChicken mc = new ManageChicken(5);  mc.add(new Chicken(1,"大米",12));  mc.add(new Chicken(2,"中米",8));  mc.add(new Chicken(3,"小米",6));  mc.add(new Chicken(4,"黄米",12));  mc.add(new Chicken(5,"绿米",8));  mc.add(new Chicken(6,"蓝米",6));  mc.update(new Chicken(6,"灰太狼",68));  mc.delete(2);  mc.findAll();  }  }  class ManageChicken{  private Chicken[] arr = null;  private int count = 0;  public ManageChicken(int size) {  if(size > 5) {  arr = new Chicken[size];  } else {  arr = new Chicken[5];  }  }  //添加数组  public void add(Chicken c) {  if(count >= arr.length) {//数组已满，需要扩充  //算法一：扩充原来的一半 arr.length\*3/3+1  //算法二：扩充原来的一倍 arr.length\*2  int newLength= arr.length\*2;  arr = Arrays.copyOf(arr,newLength);  }  arr[count] = c;  count++;  }  //修改数组  public void update(Chicken c) {  Chicken chicken = find(c.getId());  if(chicken != null) {  chicken.setName(c.getName());  chicken.setAge(c.getAge());  }  }  //删除数组  //原理：删除某一个，是将后面的统一往前提，最后将末尾的那个赋值为null,并将数组下标减一  public void delete(int id) {  for(int i = 0; i < count; i++) {  if(arr[i].getId() == id) {  for(int j = i; j < count-1; j++) {  arr[j] = arr[j+1];  }  arr[count-1] = null;  count--;  break;  }  }  }    //通过ID查找数组  public Chicken find(int id) {  for(Chicken chicken:arr) {  if(chicken.getId() == id) {  return chicken;  }  }  return null;  }  //打印所有对象  public void findAll() {  for(Chicken chicken:arr) {  System.out.println("姓名："+chicken.getName()+"年龄："+chicken.getAge());  }  }  }  class Chicken {  private int id;  private String name;  private int age;  public Chicken(int id,String name,int age) {  this.id = id;  this.name = name;  this.age = age;  }  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;  }  public int getAge() {  return age;  }  public void setAge(int age) {  this.age = age;  }  } |

## 十二.继承的基本概念

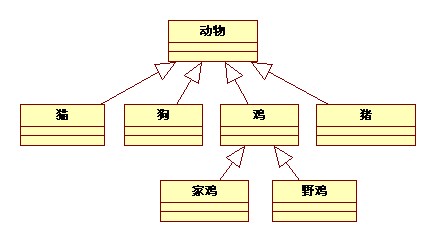
继承是从已有的类创建新类的过程。

1、继承是面向对象三大特征之一

2、被继承的类称为父类（超类），继承父类的类称为子类（派生类）

3、继承是指一个对象直接使用另一对象的属性和方法。

4、通过继承可以实现代码重用



语法：[访问权限] class 子类名 extends 父类名{

类体定义；

}

示例：

public class Dog{

private String name;

private String sex;

public void eat(){System.out.println(“吃饭”);}

}

public class HomeDog extends Dog{

//类的定义

}

public class HuskyDog extends Dog{

//类的定义

}

protected（受保护的访问权限修饰符，用于修饰属性和方法，使用protected修饰的属性和方法可以被子类继承）

**继承的限制约定：**

1、Java只能实现单继承，也就是一个类只能有一个父类

2、允许多层继承，即：一个子类可以有一个父类，一个父类还可以有其他的父类。

3、继承只能继承非私有的属性和方法。

4、构造方法不能被继承

**继承的好处：**

1、提高代码的复用性

2、提高代码的维护性

3、让类与类之间产生关系，是多态的前提

**继承的缺点：**

增强了类与类之间的耦合性

**开发原则：高内聚，低耦合**

## 十三.方法的重写

**方法重写(overriding method)**

在Java中，子类可继承父类中的方法，而不需要重新编写相同的方法。但有时子类并不想原封不动地继承父类的方法，而是想做一定的修改，这就需要采用方法的重写。方法重写又称方法覆盖。

在子类和父类中，重写方法后，在调用时，以创建的对象类型为准，会调用谁的方法。

**关于方法重写的一些特性：**

1、发生在子父类中，方法重写的两个方法返回值、方法名、参数列表必须完全一致（子类重写父类的方法）

2、子类抛出的异常不能超过父类相应方法抛出的异常（子类异常不能大于父类异常）

3、子类方法的访问级别不能低于父类相应方法的访问级别(子类访问级别不能低于父类访问级别)

4、父类中的方法若使用private、static、final任意修饰符修饰，那么，不能被子类重写。

**为什么要重写方法？或者方法重写的目的是什么？**

若子类从父类中继承过来的方法，不能满足子类特有的需求时，子类就需要重写父类中相应的方法，方法的重写也是程序扩展的体现。

## 十四.super关键字

**super可以完成以下的操作：**

1、使用super调用父类中的属性，可以从父类实例处获得信息。

2、使用super调用父类中的方法，可以委托父类对象帮助完成某件事情。

3、使用super调用父类中的构造方法（super(实参)形式），必须在子类构造方法的第一条语句，调用父类中相应的构造方法，若不显示的写出来，默认调用父类的无参构造方法，比如：super();

## 十五.final关键字

**使用final关键字完成以下的操作：**

1、使用final关键字声明一个常量

修饰属性或者修饰局部变量（最终变量），也称为常量。

2、使用final关键字声明一个方法

该方法为最终方法，且只能被子类继承，但是不能被子类重写。

3、使用final关键字声明一个类

该类就转变为最终类，没有子类的类，final修饰的类无法被继承。

4、在方法参数中使用final，在该方法内部不能修改参数的值（在内部类中详解）

**final应用：**

定义一个常量：

public static final int NUM = 10;

定义一个final方法（不常用）

定义一个final类：通常在常量类中使用

//常量类：在该类中只有常量，通常是应用程序中公共的常量或标记

public final class Constant{

public static final String SERVER\_ROOT\_URL = "http://www.baidu.com";

public static final String CACHE\_PATH = "data\_cache";

//....

}

## 十六.抽象类

**抽象类的基本概念**

（1）很多具有相同特征和行为的对象可以抽象为一个类；

（2）使用abstract关键字声明的类为抽象类。

|  |
| --- |
| 定义一个抽象类  abstract class Animal{  public abstract void move();  }  abstract class Person extends Animal{  private String name;  //...  public abstract void eat();//抽象方法  }  //具体类  class Man extends Person{  public void eat(){  System.out.println("我是男人，我爱吃肉");  }  public void move(){  System.out.println("我爱跑步");  }  }  class Women extends Person{  public void eat(){  System.out.println("我是女人，我爱吃香蕉");  }  public void move(){  System.out.println("我喜欢逛街");  }  } |

**抽象类的规则：**

1、抽象类可以没有抽象方法，有抽象方法的类必须是抽象类

2、非抽象类继承抽象类必须实现所有抽象方法

3、抽象类可以继承抽象类，可以不实现父类抽象方法。

4、抽象类可以有方法实现和属性

5、抽象类不能被实例化

6、抽象类不能声明为final

7、抽象类可以有构造方法

## 十七.接口

**接口的概念**

1、接口是一组行为的规范、定义，没有实现（JDK1.8默认方法）

2、使用接口，可以让我们的程序更加利于变化

3、接口是面向对象编程体系中的思想精髓之一

4、面向对象设计法则：基于接口编程

**接口的定义格式：**

interface 接口名称{

全局常量 ;

抽象方法 ;

}

**示列：**

|  |
| --- |
| interface IEat{  //public abstract void eat();  void eat();  //public static final int NUM = 10;  int NUM = 10;  }  interface ISleep extends IEat{  void sleep();  }  **//实现接口的类**  class Girl implements IEat,ISleep{  private String name;  public Girl(String name){  this.name = name;  }  public void eat(){  System.out.println("我是"+name+"的女票，我爱吃香蕉");  }  public void sleep(){  System.out.println("我爱睡觉");  }  } |

**接口的使用规则：**

（1）定义一个接口，使用interface关键字

（2）在一个接口中，只能定义常量、抽象方法，JDK1.8后可以定义默认的实现方法

（3）接口可以继承多个接口：extends xxx,xxx

（4）一个具体类实现接口使用implements关键字

（5）一个类可以实现多个接口

（6）抽象类实现接口可以不实现接口的方法

（7）在接口中定义的方法没有声明 访问修饰符，默认为public

（8）接口不能有构造方法

（9）接口不能被实例化

**面向对象设计原则：**

（1）对修改关闭，对扩展开放

（2）面向接口编程

## 十八.多态性

多态是面向对象三大特性之一

**什么是多态性？**

对象在运行过程中的多种形态。

多态性我们大概可以分为两类：

（1）方法的重载与重写

（2）对象的多态性

**例如：**

//用父类的引用指向子类对象（用大的类型去接受小的类型，向上转型、自动转换）

Chicken home = new HomeChicken();

**结论：**

在编程时针对抽象类型的编写代码，称为面向抽象编程（或面向接口编程）

父类通常都定义为抽象类、接口

**对象的多态性：**

对象多态性是从继承关系中的多个类而来，

**向上转型：**将子类实例转为父类引用

格式：父类 父类对象 = 子类实例 ；🡪 自动转换

以基本数据类型操作为例：int i = ‘a' ;

（因为char的容量比int小，所以可以自动完成）

**向下转型：**将父类实例转为子类实例

格式：子类 子类对象 = （子类）父类实例 ；强制转换

以基本数据类型操作为例：char c = (char)97;

因为整型是4个字节比char 2个字节要大，所以需要强制完成

## 十九.instanceof关键字

**instanceof** 是用于检查对象是否为指定的类型，通常在把父类引用强制转换为子类引用时要使用，以避免发生类型转换异常（ClassCastException）。

语法格式如下：

对象 instanceof 类 型 －－返回boolean类型值

示例：

if(homeChicken instanceof Chicken){

//...

}

该语句一般用于判断一个对象是否为某个类的实例，是返回true，否返回false

**父类的设计法则**

通过instanceof关键字，我们可以很方便 的检查对象的类型，但如果一个父类的子类过多，这样的判断还是显得很繁琐，那么如何去设计一个父类呢？

1、父类通常情况下都设计为抽象类或接口，其中优先考虑接口，如接口不能满足才考虑抽象类。

2、一个具体的类尽可能不去继承另一个具体类，这样的好处是无需检查对象是否为父类的对象。

## 二十.抽象类应用——模板方法模式

模板方法模式（Templete Method）：定义一个操作中的算法的骨架，而将一些可变部分的实现延迟到子类中。模版方法模式使得子类可以不改变一个算法的结构即可重新定义该算法的某些特定的步骤。

|  |
| --- |
| class Test{  public static void main(String[] args) {  UserManager um = new UserManager();  um.action("admin","add");  }  }  abstract class BaseManager{  public void action(String name,String method) {  if("admin".equals(name)) {  execute(method);  } else {  System.out.println("没有操作权限！！！");  }  }    public abstract void execute(String method);  }  class UserManager extends BaseManager{  public void execute(String method) {  if("add".equals(method)) {  System.out.println("调用了添加的方法");  } else if("del".equals(method)) {  System.out.println("调用了删除的方法");  }  }  }  class ClassManager{  public void execute(String method) {  if("add".equals(method)) {  System.out.println("调用了添加的方法");  } else if("del".equals(method)) {  System.out.println("调用了删除的方法");  }  }  } |

## 二十一.接口应用——策略模式

策略模式（Strategy Pattern），定义了一系列的算法，将每一种算法封装起来并可以相互替换使用，策略模式让算法独立于使用它的客户应用而独立变化。

OO设计原则：

1、面向接口编程（面向抽象编程）

2、封装变化

3、多用组合，少用继承

|  |
| --- |
| class Test{  public static void main(String[] args) {  BaveManager bm = new BaveManagerSon();  bm.setBave(new FileBave());  bm.validation("西游记，水浒传");  }  }  interface Bave{  public void bave(String name);  }  class FileBave implements Bave {  public void bave(String name) {  System.out.println("在文件中保存了"+name);  }  }  class NetBave implements Bave {  public void bave(String name) {  System.out.println("在网络上传了"+name);  }  }  abstract class BaveManager {  private Bave bave;  public void setBave(Bave bave) {  this.bave = bave;  }  public void validation(String name) {  System.out.println("正在进行验证！！！");  bave.bave(name);  System.out.println("保存成功！！！");  }  }  class BaveManagerSon extends BaveManager {    } |

二十二.object类

**Object类 是类层次结构的根类。**

每个类都使用 Object 作为超类。所有对象（包括数组）都实现这个类的方法

所有类都是Object类的子类。

public String **toString**()方法 返回该对象的字符串表示。

public boolean **equals**(Object obj) 指示其他某个对象是否与此对象“相等”。

重写equals方法，使实例化统一对象的值返回true

|  |
| --- |
| class Test{  public static void main(String[] args) {  Student s1 = new Student("小明", 32, 23);  Student s2 = new Student("小明", 12, 23);  System.out.println(s1.equals(s2));  }  }  class Student {  private String name;  private int sid;  private int age;  public Student(String name,int sid,int age) {  this.name = name;  this.sid = sid;  this.age = age;  }  public boolean equals(Object obj) {  if(this == obj) {  return true;  }  if(obj instanceof Student) {  Student s = (Student)obj;  if(!this.name.equals(s.name)) {  return false;  } else if (this.sid != s.sid) {  return false;  } else if (this.age != s.age) {  return false;  }  return true;  }  return false;  }  } |

**protected void finalize()throws Throwable** 当垃圾回收器确定不存在对该对象的更多引用时，由对象的垃圾回收器调用此方法

**public final Class<?> getClass()** 返回此 Object 的运行时类。

## 二十二.简单工程设计模式

简单工厂模式是由一个工厂对象决定创建出哪一种产品类的实例。简单工厂模式是工厂模式家族中最简单实用的模式。

|  |
| --- |
| class Test{  public static void main(String[] args) {  //使用者与被使用者之间耦合，产生了依赖，当被使用者改变时，会影响使用者。  //ProductTool p = new PhoneImpl();  //使用工厂模式来降低两者之间的依赖  ProductTool pt = ProductFactory.getProduct("tv");  if (null != pt) {  pt.product();  }  }  }  class ProductFactory {  public static ProductTool getProduct(String name) {  if ("phone".equals(name)) {  return new PhoneImpl();  } else if ("tv".equals(name)) {  return new TvImpl();  } else {  return null;  }  }  }  interface ProductTool {  public void product();  }  class PhoneImpl implements ProductTool {  public void product() {  System.out.println("我们生产了手机！！");  }  }  class TvImpl implements ProductTool {  public void product() {  System.out.println("我们生产了电视！！");  }  } |

## 二十三.静态代理模式

代理模式（Proxy）：为其他对象提供一种代理以控制对这个对象的访问。

代理模式说白了就是“真实对象”的代表，在访问对象时引入一定程度的间接性，因为这种间接性可以附加多种用途。

|  |
| --- |
| class Test{  public static void main(String[] args) {  ActionProxy ap = new ActionProxy(new UserActionImpl());  ap.doAction();  }  }  class ActionProxy implements Action {  private Action target; //被代理的对象  public ActionProxy (Action target) {  this.target = target;  }    public void doAction() {  long startTime = System.currentTimeMillis();  target.doAction();  long endTime = System.currentTimeMillis();  System.out.println(endTime-startTime+"毫秒");  }  }  interface Action {  public void doAction();  }  class UserActionImpl implements Action {  public void doAction() {  System.out.println("用户开始工作！！");  }  }  class AdminActionImpl implements Action {  public void doAction() {  System.out.println("管理员开始工作！！");  }  } |

## 二十四.适配器模式

适配器模式（ Adapter ）：将一个类的接口转换成客户希望的另外一个接口。适配器模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。

案例一：

|  |
| --- |
| class Test{  public static void main(String[] args) {  PowerA powera = new PowerAImpl();  //work(powera);  //适配器可以让work()这个方法不用重复写  PowerA adapter = new Adapter(new PowerBImpl());  work(adapter);  }  public static void work (PowerA powera) {  System.out.println("正在连接...");  powera.insert();  System.out.println("工作结束...");  }  }  //适配器  class Adapter implements PowerA {  private PowerB powerb;  public Adapter (PowerB powerb) {  this.powerb = powerb;  }  public void insert() {  powerb.connect();  }  }  /\*\*  电源A接口  \*/  interface PowerA {  public void insert();  }  class PowerAImpl implements PowerA {  public void insert() {  System.out.println("插入到了电源A口上");  }  }  /\*\*  电源B接口  \*/  interface PowerB {  public void connect();  }  class PowerBImpl implements PowerB {  public void connect() {  System.out.println("连接到了电源B口上");  }  } |

案例二.

|  |
| --- |
| interface Animal {  public void eat();  public void run();  public void sing();  public void jump();  public void swim();  }  //适配器  abstract class AnimalAdapter implements Animal {  public void eat() {}  public void run() {}  public void sing() {}  public void jump() {}  public void swim() {}  }  class Dog extends AnimalAdapter {  public void run() {  System.out.println("我要开始疯狂的跑了！！");  }  } |

## 二十五.内部类

内部类就是在一个类的内部定义的类。

成员内部类格式如下：

class Outer {

class Inner{}

}

编译上述代码会产生两个文件：

Outer.class和Outer$Inner.class。

### 成员内部类

内部类除了可以在外部类中产生实例化对象，也可以在外部类的外部来实例化。

那么，根据内部类生成的\*.class文件：Outer$Inner.class

“$” 符号在程序运行时将替换成“.”

所以内部类的访问：通过“外部类.内部类”的形式表示。

Outer out = new Outer() ;// 产生外部类实例

Outer.Inner in = null; // 声明内部类对象

in = out.new Inner() ; // 实例化内部类对象

|  |
| --- |
| class Test {  public static void main(String[] args) {  Outer outer = new Outer();  Outer.Inner inner = outer.new Inner();  inner.print();  }  }  class Outer{ （不建议使用）  public class Inner{  public void print() {  System.out.println("inner");  }  }  }  class Outer{ （建议使用）  public void innerPrint() {  Inner inner = new Inner();  inner.print();  }  private class Inner{  public void print() {  System.out.println("成员内部类");  }  }  } |

### 方法内部类

内部类可以作为一个类的成员外，还可以把类放在方法内定义。

|  |
| --- |
| class Outer {  public void doSomething(){  class Inner{  public void seeOuter(){}  }  }  } |

**注意：**

1.方法内部类只能在定义该内部类的方法内实例化，不可以在此方法外对其实例化。

|  |
| --- |
| class Outer{  public void doSomething(){  class Inner{  public void seeOuter(){  System.out.println("方法内部类！！");  }  }  Inner in = new Inner();  in.seeOuter();  }  } |

2.方法局部变量或方法参数要被内部类使用，必须是final类型（jdk1.8可不用写final，默认为final，以前的版本必须用final修饰）

|  |
| --- |
| class Outer{  public void doSomething(final int y){  final int x = 5;  class Inner{  public void seeOuter(){  System.out.println("方法内部类！！"+x);  }  }  Inner in = new Inner();  in.seeOuter();  }  } |

### 静态内部类

在一个类内部定义一个静态内部类：

静态的含义是该内部类可以像其他静态成员一样，没有外部类对象时，也能够访问它。静态嵌套类仅能访问外部类的静态成员和方法。

|  |
| --- |
| class Outer{  static class Inner{}  }  class Test {  public static void main(String[] args){  Outer.Inner n = new Outer.Inner();  }  } |

### 匿名内部类

匿名内部类就是没有名字的内部类。

匿名内部类的三种情况：

（1）继承式的匿名内部类

|  |
| --- |
| class Outer{  public void doSomething () {  Cat cat = new Cat() {  public void eat() {  System.out.println("继承式匿名内部类！！");  }  };  cat.eat();  }  }  abstract class Cat {  public abstract void eat();  } |

（2）接口式的匿名内部类

|  |
| --- |
| class Outer{  public void doSomething () {  Cat cat = new Cat() {  public void eat() {  System.out.println("接口式匿名内部类！！");  }  };  cat.eat();  }  }  interface Cat {  public void eat();  } |

（3）参数式的匿名内部类

|  |
| --- |
| class Test {  public static void main(String[] args) {  Outer outer = new Outer();  outer.doSomething(new Cat() {  public void eat() {  System.out.println("参数式匿名内部类");  }  });  }  }  class Outer{  public void doSomething (Cat cat) {  cat.eat();  }  }  interface Cat {  public void eat();  } |

**在使用匿名内部类时，要记住以下几个原则：**

（1）不能有构造方法，只能有一个实例。

（2）不能定义任何静态成员、静态方法。

（3）不能是public,protected,private,static。

（4）一定是在new的后面，用其隐含实现一个接口或继承一个类。

（5）匿名内部类为局部的，所以局部内部类的所有限制都对其生效。

**问题：局部内部类访问局部变量必须用final修饰，为什么?**

当调用这个方法时，局部变量如果没有用final修饰，他的生命周期和方法的生命周期是一样的，当方法被调用时会入栈，方法结束后即弹栈，这个局部变量也会消失，那么如果局部内部类对象还没有马上消失想用这个局部变量，显然已无法使用了，如果用final修饰会在类加载的时候进入常量池，即使方法弹栈，常量池的常量还在，也就可以继续使用了。

注意：在jdk1.8中取消了在局部内部类中使用的变量必须显示的使用final修饰，编译器默认会为这个变量加上final

**内部类的作用**

每个内部类都能独立地继承自一个（接口的）实现，所以无论外部类是否已经继承了某个（接口的）实现，对于内部类都没有影响。如果没有内部类提供的可以继承多个具体的或抽象的类的能力，一些设计与编程问题就很难解决。从这个角度看，内部类使得多重继承的解决方案变得完整。接口解决了部分问题，而内部类有效地实现了“多重继承”。

依赖外部类对象的：成员内部类，方法内部类，匿名内部类

静态内部类不依赖外部类的对象。所以，我们在项目中优先考虑选择静态内部类（不会产生内存泄露）

## 二十六.数据结构之链表

### 递归算法

在链表数据结构中，我们需要使用到递归算法。

递归算法是一种直接或者间接地调用自身算法的过程。在计算机编写程序中，递归算法对解决一大类问题是十分有效的，它往往使算法的描述简洁而且易于理解。

注意：1.递归必需要有出口；2.递归内存消耗大，容易发生内存溢出。（每一次调用方法都会入栈内存）建议不用

|  |
| --- |
| class Test {  public static void main(String[] args) {  int result = jiecheng(10);  System.out.println(result);  }  // 10\*9\*8\*7\*6\*5\*4\*3\*2  //递归算法，方法本身调用自己  public static int jiecheng(int num) {  if(num == 1) return 1;  return num \* jiecheng(num-1);  }  } |

**链表**(Linked list）一种常见的基础数据结构，是一种线性表，但是并不会按线性的顺序存储数据，而是在每一个节点里存的是下一个节点的指针(Pointer）。

链表与数组：线性数据结构

数组适合查找，遍历，固定长度。

链表适合插入，删除，不易过长，否则会导致遍历性能下降

|  |
| --- |
| class Test {  public static void main(String[] args) {  NodeManager nm = new NodeManager();  nm.add(5);  nm.add(4);  nm.add(3);  nm.add(2);  nm.add(1);  //System.out.println("查找情况："+nm.find(3));  //System.out.println("修改情况："+nm.update(4,6));  //System.out.println("删除情况："+nm.del(4));  System.out.println("前置插入情况："+nm.insert(20,6));  //System.out.println("总共的数量："+nm.count());  nm.print();  }  }  class NodeManager {  private Node root; //根节点  private int initIndex = 0;  private int count = 0;  //添加  public void add(int data) {  if(root == null) {  root = new Node(data);  } else {  root.addNode(data);  }  }  //删除  public boolean del(int data) {  if(root == null) return false;  if(root.getData() == data) {  root = root.next;  return true;  } else {  return root.delNode(data);  }  }  //打印所有  public void print() {  if(root != null) {  System.out.println(root.getData()+"-->");  root.printNode();  System.out.println();  }  }  //查找  public boolean find(int data) {  if(root == null) return false;  if (root.getData() == data) {  return true;  } else {  return root.findNode(data);  }  }  //修改  public boolean update(int oldData,int newData) {  if(root == null) return false;  if (root.getData() == oldData) {  root.setData(newData);  return true;  } else {  return root.updateNode(oldData, newData);  }  }  //插入  public boolean insert(int index, int data) {  System.out.println(count());  if(0 > index) return false;  if(index > count) return false;  initIndex = 0;  if (initIndex == index) {  Node node = new Node(data);  node.next = root;  root = node;  return true;  } else {  return root.insertNode(index, data);  }  }  //计算数量  public int count() {  if(root == null) {  return count;  } else {  count++;  return root.countNode();  }  }    private class Node {  private int data;  private Node next;  public Node(int data) {  this.data = data;  }  public void setData(int data) {  this.data = data;  }  public int getData() {  return data;  }  public void addNode(int data) {  if(this.next == null) {  this.next = new Node(data);  } else {  this.next.addNode(data);  }  }  public boolean delNode(int data) {  if(this.next == null) return false;  if (this.next.data == data) {  this.next = this.next.next;  return true;  } else {  return this.next.delNode(data);  }  }  public void printNode() {  if(this.next != null) {  System.out.println(this.next.data+"-->");  this.next.printNode();  System.out.println();  }  }  public boolean findNode(int data) {  if(this.next == null) return false;  if(this.next.data == data) {  return true;  } else {  return this.next.findNode(data);  }  }  public boolean updateNode(int oldData,int newData) {  if(this.next == null) return false;  if(this.next.data == oldData) {  this.next.data = newData;  return true;  } else {  return this.next.updateNode(oldData, newData);  }  }  public boolean insertNode(int index, int data) {  initIndex++;  if (initIndex == index) {  Node node = new Node(data);  node.next = this.next;  this.next = node;  return true;  } else {  return this.next.insertNode(index, data);  }  }  public int countNode() {  if(this.next == null) {  return count;  } else {  count++;  return this.next.countNode();  }  }  }  } |

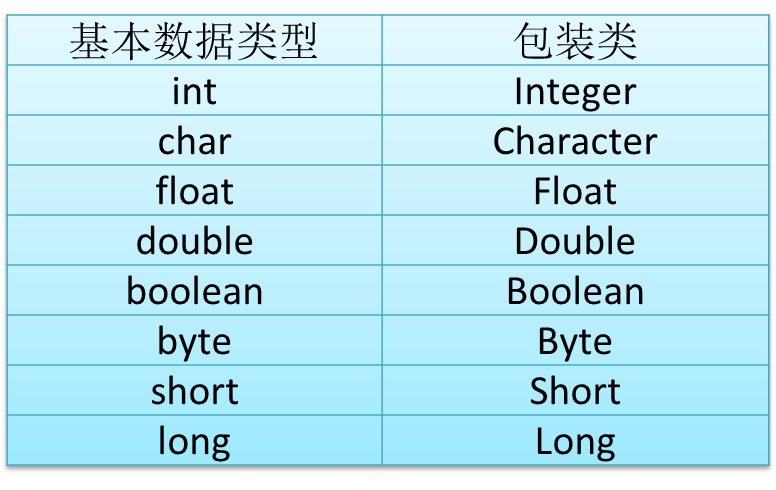
## 二十七.基本数据类型包装类

在Java中有一个设计的原则“一切皆对象”，Java中的基本数据类型就完全不符合这种设计思想，因为八种基本数据类型并不是引用数据类型，所以Java中为了解决这样的问题，JDK1.5以后引入了八种基本数据类型的包装类。

**八种包装类分为两大类型：**

· **Number**：Integer、Short、Long、Double、Float、Byte都是Number的子 类表示是一个数字。

· **Object**：Character、Boolean都是Object的直接子类。



**装箱及拆箱操作**

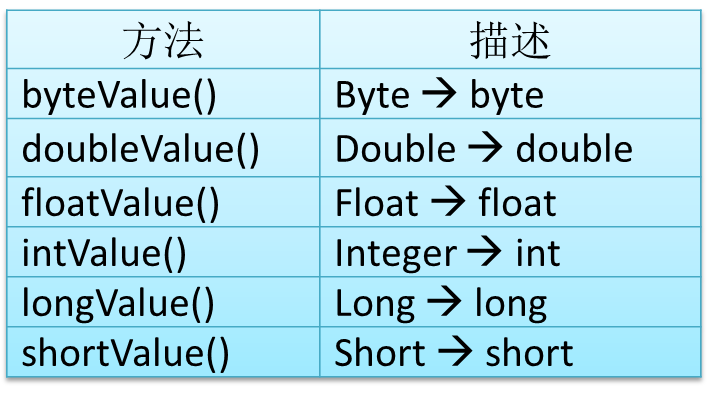
将一个基本数据类型转换为包装类，那么这样的操作称为装箱操作。将一个包装类转换为一个基本数据类型，这样的操作称为拆箱操作。

装箱：

方法一：Integer i1 = new Integer(10);

方法二：Integer i2 = 10; (建议使用)

拆箱：



**转型操作**

在包装类中，可以将一个字符串变为指定的基本数据类型，一般在输入数据时会使用较多。

在Integer类中将String变为int型数据：public static int parseInt(String s)

在Float类中将String变为float型数据：public static float parseFloat(String s)

注意：转型操作时，字符串必须由数字组成，否则会出现错误哦

**享元模式**（Flyweight Pattern）它使用共享对象，用来尽可能减少内存使用量以及分享资讯给尽可能多的相似对象；它适合用于当大量对象只是重复因而导致无法令人接受的使用大量内存。通常对象中的部分状态是可以分享。常见做法是把它们放在外部数据结构，当需要使用时再将它们传递给享元。

运用共享技术有效的支持大量细粒度的对象。

享元模式是指将一个字节127以内（包含127）的包装数据类型缓存到内存中，当Integer i1 = 10 与Integer i2 = 10 判断相等性的时候由于享元将他们指向的是同一块内存，所以 == 与 equals 都为true。而Integer i3 = 128 与 Integer i4 = 128判断相等性的时候由于不在享元的范围内所以需要重新创建对象 i3 = new Integer(128) 与i4 = new Integer(128) 所以他们在==下是false，在equals下是true。

## 二十八.包与访问修饰符

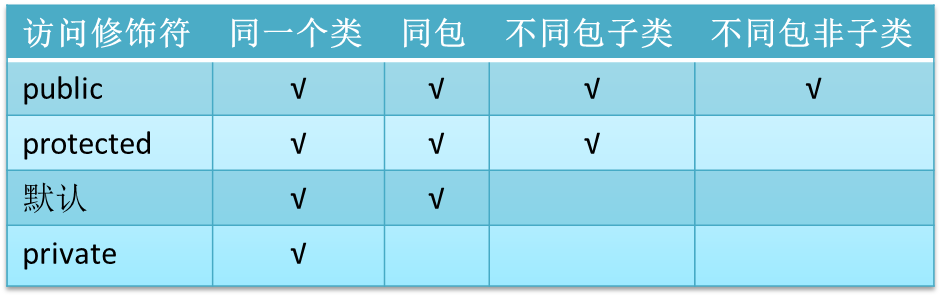
包用于对多个java源文件的管理，就像我们的文件目录一样。

定义一个包：

package com.vince;

该语句只能出现在代码中的第一句。

访问修饰符：



## 二十九.OO原则

**（1）开闭原则**

一个软件实体如类、模块和函数应该对扩展开放，对修改关闭。

**（2）合成/聚合复用原则**

新对象的某些功能在已创建好的对象里已实现，那么尽量用已有对象提供的功能，使之成为新对象的一部分，而不要再重新创建。

**（3）依赖倒置原则**

高层模块不应该依赖低层模块，二者都应该依赖其抽象；抽象不应该依赖细节；细节应该依赖抽象。

**（4）接口隔离原则**

客户端不应该依赖它不需要的接口；一个类对另一个类的依赖应该建立在最小的接口上。

**（5）迪米特法则**

一个对象应该对其他对象保持最少的了解

**（6）里氏替换原则**

所有引用基类的地方必须能透明地使用其子类的对象

**（7）单一职责原则**

不要存在多于一个导致类变更的原因，即一个类只负责一项职责。

# 第五章．Eclipse与异常处理

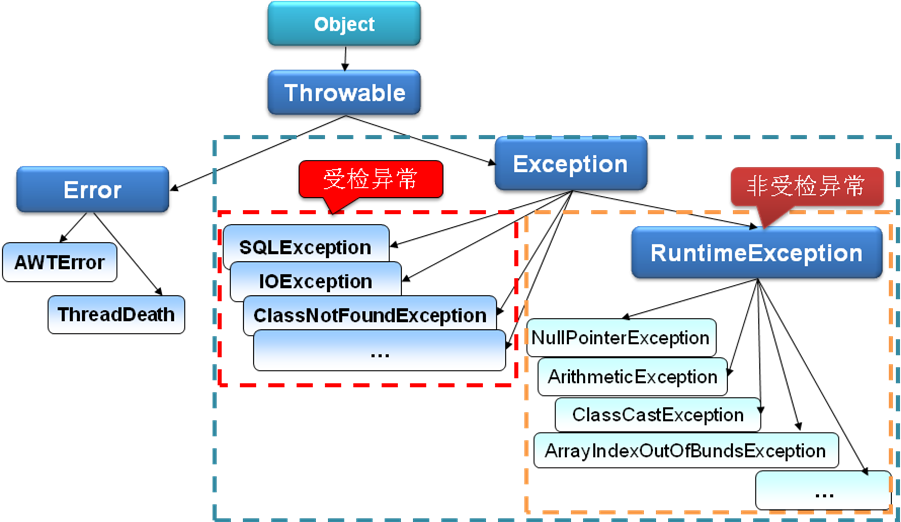
## 一．Eclipse开发环境使用入门

下载安装：http://www.eclipse.org/downloads/eclipse-packages/

## 二．异常处理

### 什么是异常

**异常**是阻止当前方法或作用域继续执行的问题，在程序中导致程序中断运行的一些指令。



### try与catch关键字

在程序中出现异常，就必须进行处理，处理格式如下：

|  |
| --- |
| **class** Test {  **public** **static** **void** main(String[] args) {  *div*(6,3);  }  **public** **static** **void** div(**int** num1, **int** num2) {  **int**[] arr = {1,2,3,4,5};  **try** {  //System.out.println(arr[5]);  arr = **null**;  System.***out***.println(arr.length);  **int** result = num1/num2;    System.***out***.println(result);  } **catch** (ArithmeticException e) {  System.***out***.println("除数不能为0");  } **catch** (ArrayIndexOutOfBoundsException e) {  System.***out***.println("索引越界异常");  } **catch** (NullPointerException e) {  System.***out***.println("空指针异常");  } **catch** (Exception e) {  System.***out***.println("捕捉想象不到的异常");  } **finally** {  System.***out***.println("程序执行完毕");  }  }  } |

### 异常处理过程分析：

（1）、一旦产生异常，则系统会自动产生一个异常类的实例化对象。

（2）、此时如果存在了try语句，则会自动找到匹配的catch语句执行，如果没有异常处理，则程序将退出，并由系统报告错误。

（3）、所有的catch根据方法的参数匹配异常类的实例化对象，如果匹配成功，则表示由此catch进行处理。

### finally关键字：

在进行异常的处理之后，在异常的处理格式中还有一个finally语句，那么此语句将作为异常的统一出口，不管是否产生了异常，最终都要执行此段代码。

### throw与throws关键字：

throws关键字主要在方法的声明上使用，表示方法中不处理异常，而交给调用处处理。实际上对于Java程序来讲，如果没有加入任何的异常处理，默认由JVM进行异常的处理操作。

throw关键字表示在程序中手动抛出一个异常，因为从异常处理机制来看，所有的异常一旦产生之后，实际上抛出的就是一个异常类的实例化对象，那么此对象也可以由throw直接抛出。

|  |
| --- |
| **public** **static** **void** main(String[] args) {  *div2*(6,0);  }  **public** **static** **void** div2(**int** num1, **int** num2) {  **int** result = num1/num2;  System.***out***.println(result);  }  报异常为  Exception in thread "main" java.lang.ArithmeticException: / by zero  at test.Test.div2(Test.java:21)  at test.Test.main(Test.java:7)  **使用 throws 与throw 后**  **public** **static** **void** main(String[] args) {  *div*(6,0);  }  **public** **static** **void** div(**int** num1, **int** num2) **throws** ArithmeticException {  **try** {  **int** result = num1/num2;  System.***out***.println(result);  } **catch** (ArithmeticException e) {  **throw** **new** ArithmeticException("除数不能为0");  }  }  报异常为  Exception in thread "main" java.lang.ArithmeticException: 除数不能为0  at test.Test.div(Test.java:16)  at test.Test.main(Test.java:6) |

return的变化 当使用throw时 return结束可以在try catch里面，不使用throw必须在try catch外面定义一个return。

|  |
| --- |
| public static int div(int num1, int num2) throws ArithmeticException {  try {  int result = num1/num2;  System.out.println(result);  return result;  } catch (ArithmeticException e) {  throw new ArithmeticException("除数不能为0");  }  }    public static int div2(int num1, int num2) {  int result = 0;  try {  result = num1/num2;  System.out.println(result);  } catch (ArithmeticException e) {  System.out.println("aaaa");  }  return result;  } |

**异常处理的语法规则：**  
    第一、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，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。  
    第二、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。  
    第三、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。

### 异常类

ArithmeticException 算数异常

ArrayIndexOutOfBoundsException 数组索引越界异常

NullPointerException 空指针异常

InputMismatchException 输入不匹配异常

RuntimeException 运行时异常

ClassNotFoundException 类找不到异常

DataFormatException 数据格式异常

ClassCastException 类转换异常

**受检异常：Exception （编译class文件时出现的异常，必须抛出的异常）**

定义方法时必须声明所有可能会抛出的exception； 在调用这个方法时，必须捕获它的checked exception，不然就得把它的exception传递下去；exception是从java.lang.Exception类衍生出来的。例如：IOException，SQLException就属于Exception

**非受检异常：RuntimeException （class文件运行时出现的异常）**

在定义方法时不需要声明会抛出runtime exception； 在调用这个方法时不需要捕获这个runtime exception； runtime exception是从java.lang.RuntimeException或java.lang.Error类衍生出来的。 例如：NullPointException，IndexOutOfBoundsException就属于runtime exception

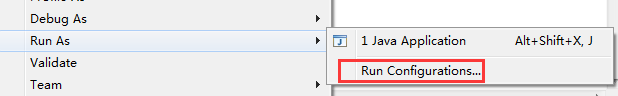
### 自定义异常

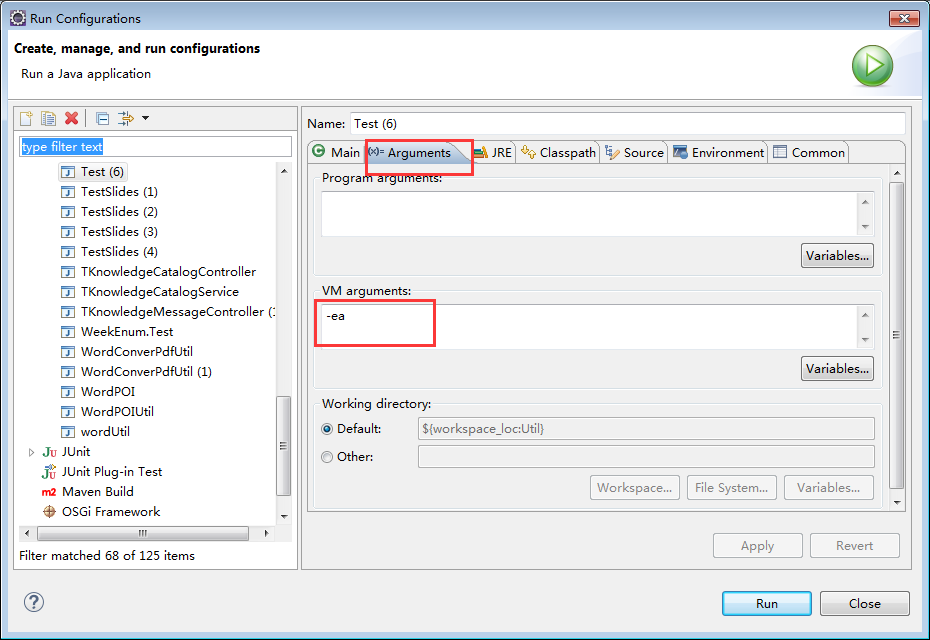
|  |
| --- |
| **class Test {**  **public static void main(String[] args) {**  **UserService userService = new UserService();**  **try {**  **User user = userService.login("admin", 12345);**  **System.out.println(user.toString());**  **} catch (MyException e) {**  **//打印异常**  **e.printStackTrace();**  **}**  **}**  **}**  **/\*\***  **\***  **\* 自定义异常通常都是继承一个类来实现**  **\* 1.Throwable**  **\* 2.Exception**  **\* 3.RuntimeException**  **\* 自定义异常的实现是提供构造方法**  **\* 异常对象本身是没有实现功能，只是一个有意义的标识**  **\* @author: schu**  **\* @date: 2019年1月18日 上午9:55:06**  **\*/**  **class MyException extends Exception{**  **private static final long serialVersionUID = 1L;**  **public MyException () {**  **super();**  **}**  **public MyException (String massage) {**  **super(massage);**  **}**  **}**  **class User {**  **private String name;**  **private int psw;**  **public User() {**  **super();**  **}**  **public User(String name, int psw) {**  **super();**  **this.name = name;**  **this.psw = psw;**  **}**  **public String getName() {**  **return name;**  **}**  **public void setName(String name) {**  **this.name = name;**  **}**  **public int getPsw() {**  **return psw;**  **}**  **public void setPsw(int psw) {**  **this.psw = psw;**  **}**  **@Override**  **public String toString() {**  **return "User [name=" + name + ", psw=" + psw + "]";**  **}**  **}**  **class UserService {**  **public User login(String name, int psw) throws MyException{**  **if (!"admin".equals(name)) {**  **throw new MyException("用户名错误！！");**  **}**  **if (12345 != psw) {**  **throw new MyException("密码错误！！");**  **}**  **User user = new User(name, psw);**  **return user;**  **}**  **}** |

### assert关键字，表示断言

当程序执行到某个固定位置的时候，程序中的某个变量的取值肯定是预期的结果，那么这种操作可以使用断言完成。

在eclipse中使用时，必须先设置，断言才会生效





|  |
| --- |
| public static void main(String[] args) {  int result = add(5, 5);  assert result == 11:"结果不正确";  }  public static int add(int a, int b) {  return a+b;  } |

### Jdk1.7新特性

|  |
| --- |
| public static void main(String[] args) {  //jdk1.5以后  Scanner scanner = new Scanner(System.in);  //jdk1.5以前 需要手动关闭IO流  BufferedReader bf = new BufferedReader(new InputStreamReader(System.in));  try {  bf.close();  } catch (IOException e1) {  // TODO Auto-generated catch block  e1.printStackTrace();  }  //jdk1.7新特性  //1.这样写 可以自动关闭IO流  //2.小异常从多个catch 可以并列写  try (BufferedReader bf2 = new BufferedReader(new InputStreamReader(System.in))) {  } catch (IOException|ArithmeticException|NullPointerException e) {  e.printStackTrace();  } catch (Exception e) {  } |

# 第六章．常用类库API

## 一．String类

String类实际是使用字符数组存储的。类被final修饰不可被继承，不可变。

String类的两种赋值方式：

（1）一种称为直接赋值：（建议使用）

String name = “小白”

（2）通过关键字new调用String的构造方法赋值

String name = new String(“小白”)

内存分析：

String s1 = “小白” String s2 = new String(“小白”) String s3 = “小白”

栈内存

S3[oxoo33]

字符串常量池

堆内存

小白

小白

new

S2[oxoo22]

s1[oxoo11]

String类编译期与运行期分析：

1、编译期和运行期

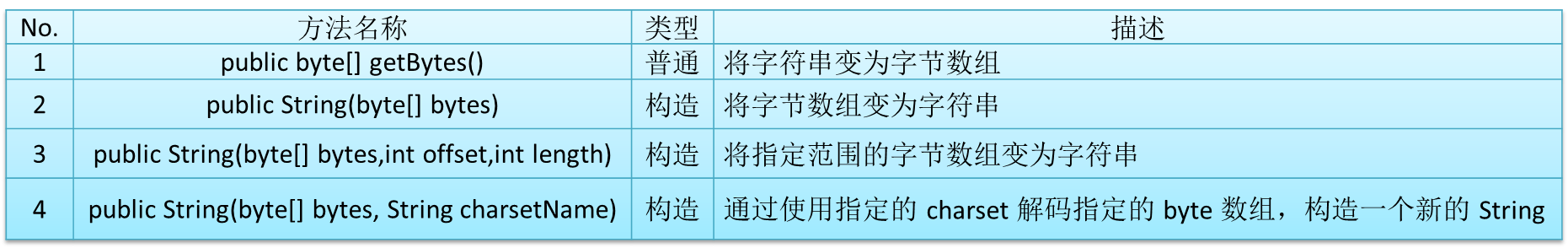
2、代码示例：4种情况分析

|  |
| --- |
| //情况一 a为运行期的变量 返回值为false  String a = "a";  String a1 = a+1;  String a2 = "a1";  System.out.println(a1 == a2);  栈内存  a[oxoo11]  a1=a[oxoo11]+1  a2[oxoo22]  堆内存  字符串常量池  “a”  “a1”  //情况二 final修饰后编译期b为常量 返回值为true  final String b = "b";  String b1 = b+1;  String b2 = "b1";  System.out.println(b1 == b2);  栈内存  final b[oxoo11]  b1[oxoo22]  b2[oxoo33]  堆内存  字符串常量池  “b”  “b1”  public static void main(String[] args) {  //情况三 c为运行期方法变量 返回值为false  String c = getC();  String c1 = c+1;  String c2 = "c1";  System.out.println(c1 == c2);  }  public static String getC() {  return "c";  }  栈内存  c[oxoo11]  c1=c[oxoo11]+1  c2[oxoo33]  堆内存  字符串常量池  “c1”  方法栈  getC()  public static void main(String[] args) {  //情况四 d为运行期方法变量 返回值为false  final String d = getD();  String d1 = d+1;  String d2 = "d1";  System.out.println(d1 == d2);  }  public static String getD() {  return "d";  } |

String类字符与字符串操作方法：



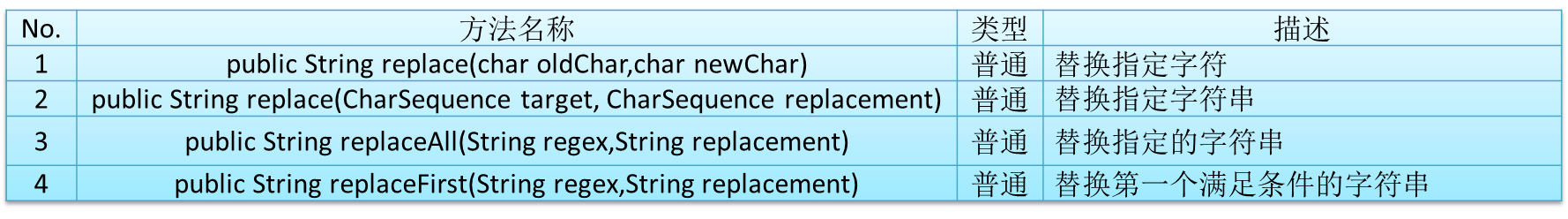
String类字节与字符串操作方法：



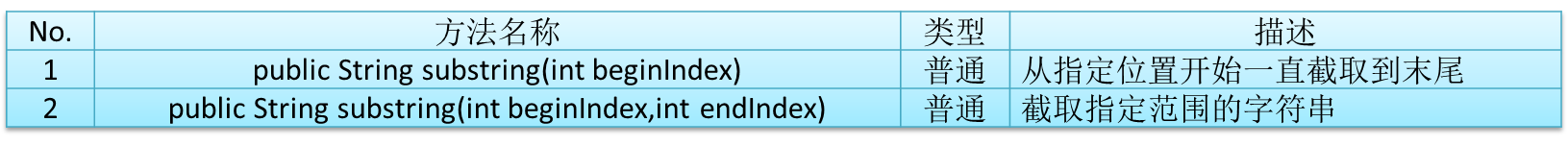
String类判断是否以指定内容开头或结尾：



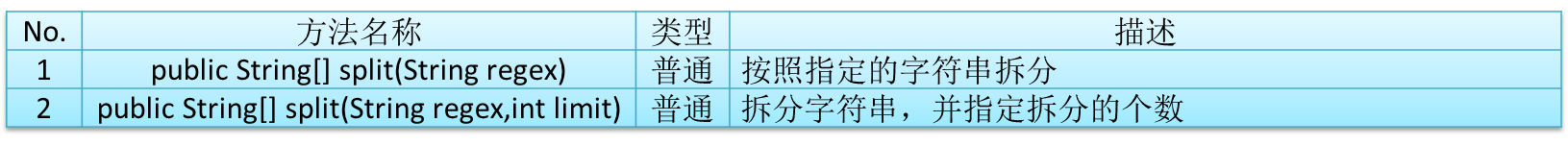
String类替换操作：



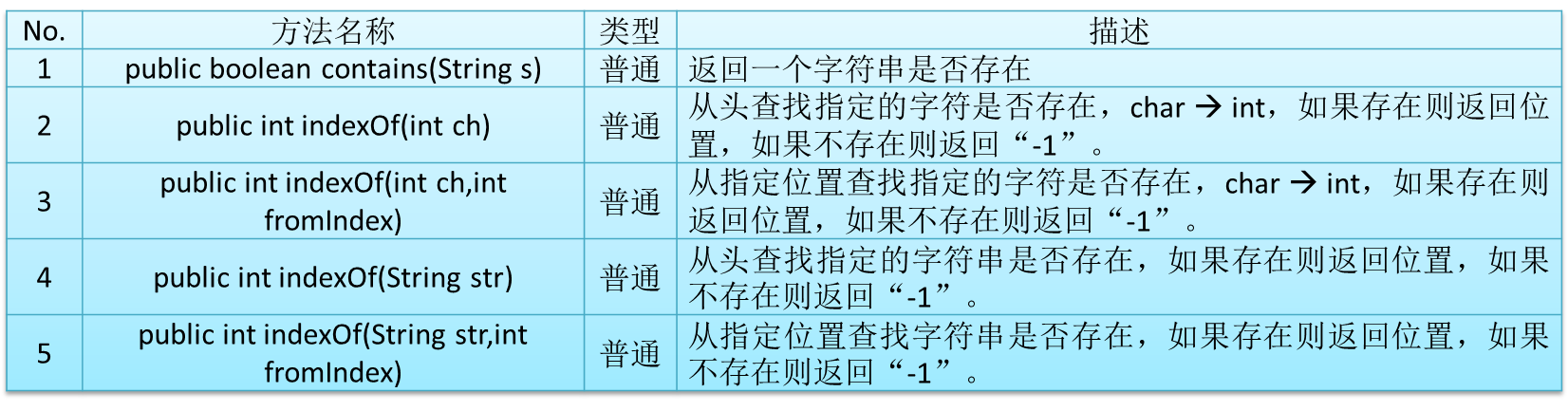
String类字符串截取操作：

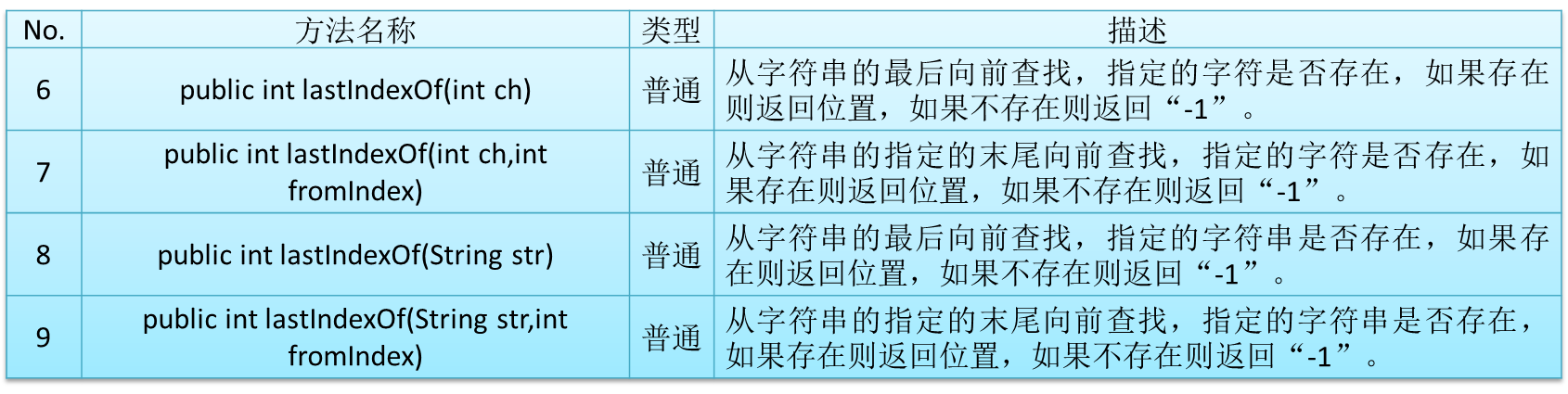


String类字符串拆分操作：

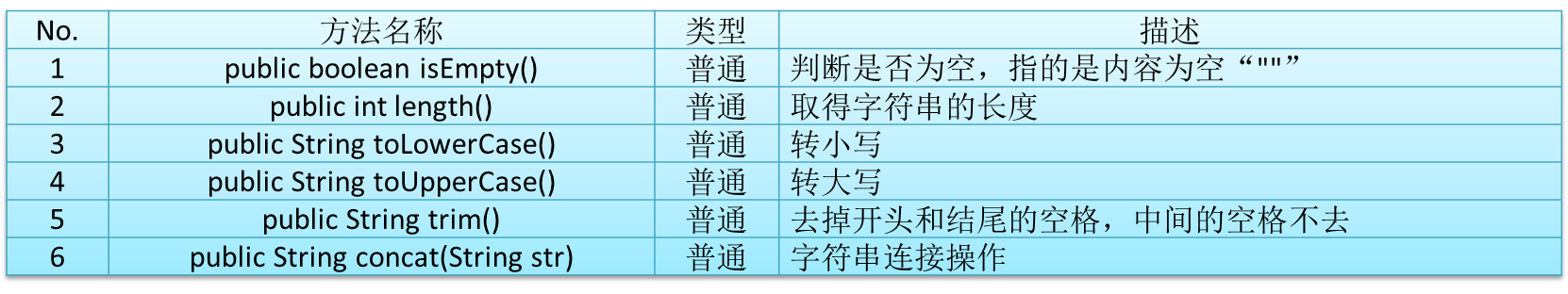


String类字符串查找操作：





String类其它操作方法：



## 二．StringBuffer类

在实际开发当中，我们经常会使用到字符串连接的操作，如果用String来操作，则使用“+”号完成字符串的连接操作。

使用String连接字符串，代码性能会非常低，因为String的内容不可改变。

内存分析 （比较占内存）

String a = “a”;

String b = “b”;

String c = a+b+1;

栈内存

堆内存

字符串常量池

“a”

a

b

1

“b”

a+b

a+b+1

解决这个问题的方法是使用StringBuffer。

StringBuffer sb = new StringBuffer(16);（建议这样使用，数组长度为16+16）

1. StringBuffer内部实现采用字符数组，默认数组长度为16，超过数组大小时，动态扩容的算法为原来长度\*2+2。
2. 线程安全的，性能较低

内存分析：

String a = "a"; String b = "b";

StringBuffer sb = new StringBuffer();

sb.append(a).append(b).append(1);

栈内存

sb

堆内存

b

a

字符串常量池

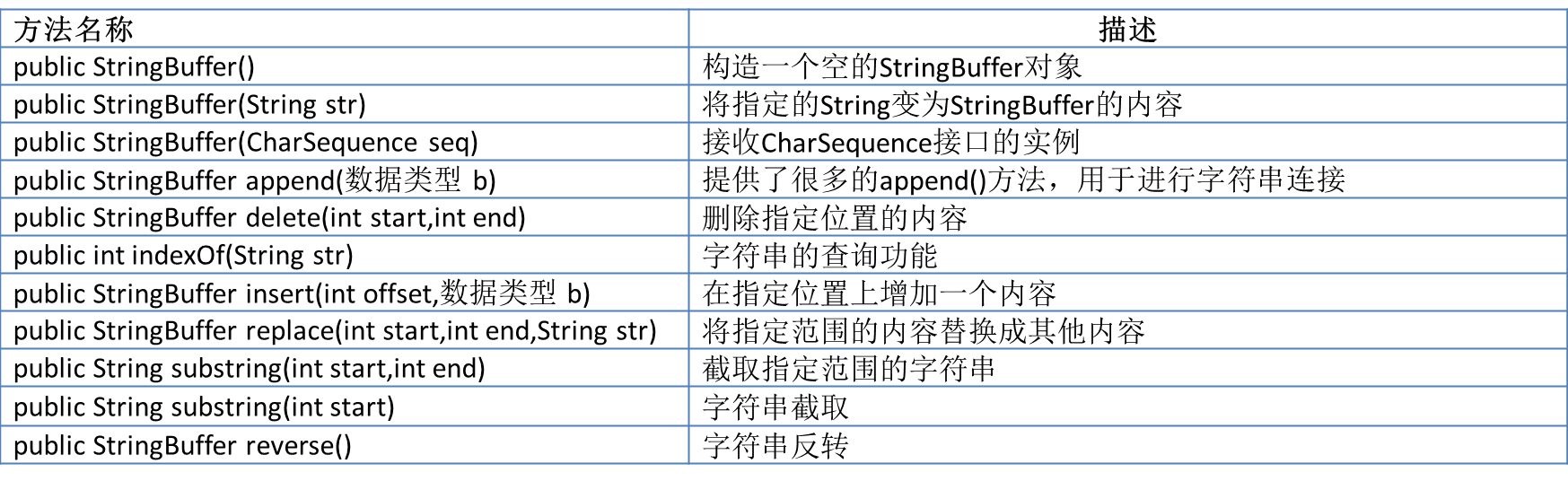
“a”

“b”

new

a,b,1

StringBuffer常用操作方法



## 三．StringBuilder类

**StringBuffer的兄弟StringBuilder：**

一个可变的字符序列。此类提供一个与 StringBuffer 兼容的 API，但不保证同步。该类被设计用作 StringBuffer 的一个简易替换，用在字符串缓冲区被单个线程使用的时候（这种情况很普遍）。如果可能，建议优先采用该类，因为在大多数实现中，它比 StringBuffer 要快。

StringBuilder就相当于String中的相加是线程不安全的，但性能比较高。

## 四．程序国际化

**1、对国际化程序的理解**

Internationalization：国际化程序可以这样理解：

同一套程序代码可以在各个语言环境下进行使用。

各个语言环境下，只是语言显示的不同，那么具体的程序操作本身都是一样的，那么国际化程序完成的就是这样的一个功能。

**2、Locale类**

Locale 对象表示了特定的地理、政治和文化地区。需要 Locale 来执行其任务的操作称为语言环境敏感的操作，它使用 Locale 为用户量身定制信息。例如，显示一个数值就是语言环境敏感的操作，应该根据用户的国家、地区或文化的风俗/传统来格式化该数值。

使用此类中的构造方法来创建 Locale：

Locale(String language)

Locale(String language, String country)

通过静态方法创建Locale：

getDefault()

**3、ResourceBundle类**

国际化的实现核心在于显示的语言上，通常的做法是将其定义成若干个属性文件（文件后缀是\*.properties），属性文件中的格式采用“key=value”的格式进行操作。

ResourceBundle类表示的是一个资源文件的读取操作，所有的资源文件需要使用ResourceBundle进行读取，读取的时候不需要加上文件的后缀。

getBundle(String baseName)

getBundle(String baseName,Locale locale)

getString(String key)

创建properties文件规则：

|  |
| --- |
| 文件名称创建格式：Info\_en\_US.properties  其中info可以自己定义,下划线不可省略，en为语言编码，US为地区编码  Key 一般为英文 value为utf格式存放 |

|  |
| --- |
| Info\_en-US.properties 英文属性文件  system.name = EMP Manager System  input.username = Input Name  input.password = Input Password  login.success = Login Success!  login.error = Login Error!  Info\_zh-CN.properties 中文属性文件  system.name = \u901a\u77e5\u516c\u544a  input.username = \u901a\u77e5\u6807\u9898  input.password = \u53d1\u5e03\u5904\u5ba4  login.success = \u53d1\u5e03\u5904\u5ba4  login.error = \u53d1\u5e03\u65f6\u95f4  ----------  import java.util.Locale;  import java.util.ResourceBundle;  class Test {  public static void main(String[] args) {  Locale.getDefault(); //默认获取操作系统语言环境  Locale locale\_cn = new Locale("ch", "CN");  Locale locale\_us = new Locale("en", "US");  //用于绑定属性文件的工具类（参数为属性文件的基本名，就是properties文件前缀）  ResourceBundle r = ResourceBundle.getBundle("Info",locale\_cn);  System.out.println(r.getString("system.name"));  }  } |

**4、处理动态文本**

前面的示例读取的内容都是固定的，如果现在假设要想打印这样的信息“欢迎你，**XXX**！”，具体的名字不是固定的，那么就要使用动态文本进行程序的处理。

进行动态的文本处理，必须使用java.text.MessageFormat类完成。这个类是java.text.Format的子类。

|  |
| --- |
| Info\_en-US.properties 英文属性文件  system.name = EMP Manager System{0}  input.username = Input Name  input.password = Input Password  login.success = Login Success!  login.error = Login Error!  Info\_zh-CN.properties 中文属性文件  system.name = \u901a\u77e5\u516c\u544a{0}  input.username = \u901a\u77e5\u6807\u9898  input.password = \u53d1\u5e03\u5904\u5ba4  login.success = \u53d1\u5e03\u5904\u5ba4  login.error = \u53d1\u5e03\u65f6\u95f4  ----------  import java.util.Locale;  import java.util.ResourceBundle;  import java.text.MessageFormat;  class Test {  public static void main(String[] args) {  Locale.getDefault(); //默认获取操作系统语言环境  Locale locale\_cn = new Locale("ch", "CN");  Locale locale\_us = new Locale("en", "US");  //用于绑定属性文件的工具类（参数为属性文件的基本名，就是properties文件前缀）  ResourceBundle r = ResourceBundle.getBundle("Info",locale\_us);  String username = r.getString("input.username");  String sys = r.getString("system.name");  sys = MessageFormat.format(sys, username);  System.out.println(sys); }  } |

## 五．Math与Random类

**1、Math类**

Math 类包含用于执行基本数学运算的方法，如初等指数、对数、平方根和三角函数。

使用Math类可以有两种方式：

(1)、直接使用（Math所在的包java.lang为默认引入的包）

(2)、使用 **import static java.lang.Math.*abs; 静态导入***

|  |  |
| --- | --- |
| static double PI | 比任何其他值都更接近 *pi*的 double 值 |
| **abs**(double a) | 返回 double 值的绝对值 |
| **random**() | 返回带正号的 double 值，该值大于等于 0.0 且小于 1.0 |
| **round**(double a) | 返回最接近参数并等于某一整数的 double 值 |
| **sqrt**(double a) | 返回正确舍入的 double 值的正平方根 |

**2、Random类**

Random：此类的实例用于生成伪随机数流

|  |  |
| --- | --- |
| nextLong() | 返回下一个伪随机数的long 值 |
| nextBoolean() | 返回下一个伪随机数boolean 值 |
| nextDouble() | 返回下一个伪随机数，在 0.0 和 1.0 之间的 double 值 |
| nextFloat() | 返回下一个伪随机数，在 0.0 和 1.0 之间的 float 值 |
| nextInt() | 返回下一个伪随机数，int 值 |
| nextInt(int n) | 返回一个伪随机数，在 0（包括）和指定值分布的 int 值 |

## 六．日期操作类

1、Date类

类 Date 表示特定的瞬间，精确到毫秒，也就是程序运行时的当前时间。

Date date = **new** Date(); // 实例化Date对象，表示当前时间

2、Calendar类

Calendar，日历类，使用此类可以将时间精确到毫秒显示。

//两种实例化方式

Calendar c = Calendar.getInstance();

Calendar c = new GregorianCalendar();

|  |
| --- |
| public static void main(String[] args) {  Calendar c1 = Calendar.getInstance();  Calendar c2 = new GregorianCalendar();  System.out.println("年："+c1.get(Calendar.YEAR));  System.out.println("月："+c1.get(Calendar.MONTH));  System.out.println("日："+c1.get(Calendar.DAY\_OF\_MONTH));  System.out.println("时："+c1.get(Calendar.HOUR\_OF\_DAY));  System.out.println("分："+c1.get(Calendar.MINUTE));  System.out.println("秒："+c1.get(Calendar.SECOND));  System.out.println("毫秒："+c1.get(Calendar.MILLISECOND));  } |

3、 DateFormat类及子类SimpleDateFormat

|  |
| --- |
| import java.text.DateFormat;  import java.text.SimpleDateFormat;  import java.util.Date;  class Test {  public static void main(String[] args) {  DateFormat df = new SimpleDateFormat("yyyy年MM月dd日 HH:mm:ss SSS");  String newDate = df.format(new Date());  System.out.println(newDate);  }  } |

## 七．对象比较器

对两个或多个数据项进行比较，以确定它们是否相等，或确定它们之间的大小关系及排列顺序称为比较。

前面我学习过Arrays.sort方法可实现对象的排序操作：

public static void **sort**(Object[] a)

|  |
| --- |
| import java.util.Arrays;  class Test {  public static void main(String[] args) {  int[] arr = {3, 8,4,2,9};  Arrays.sort(arr);  System.out.println(Arrays.toString(arr));  }  }  结果  [2, 3, 4, 8, 9] |

**（1）Comparable接口：**

此接口强行对实现它的每个类的对象进行整体排序。这种排序被称为类的自然排序，类的 compareTo 方法被称为它的自然比较方法。

|  |
| --- |
| import java.util.Arrays;  class Test {  public static void main(String[] args) {  Cat[] catArr = {new Cat("小明",2),new Cat("小黄",1),new Cat("小俊",3)};  Arrays.sort(catArr);  System.out.println(Arrays.toString(catArr));  }  }  /\*\*  自定义对象排序，实现Comparable接口，并定义compareTo()方法排序规则 \*/  class Cat implements Comparable<Cat> {  private String name;  private int age;  public Cat(){}  public Cat(String name, int age) {  this.name = name;  this.age = age;  }    public int compareTo(Cat o) {  return this.age - o.age;  }  } |

**（2）Comparator接口：**

Comparable接口是要求自定义类去实现，按照OO原则：对修改关闭，对扩展开放。

那么如果这个类已经定义好了，不想再去修改它，那如何实现比较呢？

Comparator接口：强行对某个对象collection进行整体排序的比较。

|  |
| --- |
| import java.util.Comparator;  import java.util.Arrays;  class Test {  public static void main(String[] args) {  Cat[] catArr = {new Cat("小明",2),new Cat("小黄",1),new Cat("小俊",3)};  Arrays.sort(catArr,new CatComparator());  System.out.println(Arrays.toString(catArr));  }  }  class Cat {  private String name;  private int age;  public Cat(){}  public Cat(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;  }  }  class CatComparator implements Comparator<Cat> {  public int compare(Cat c1, Cat c2) {  return c1.getAge() - c2.getAge();  }  } |

## 八．对象的克隆

将一个对象复制一份，称为对象的克隆技术。

在Object类中存在一个clone()方法：

protected Object clone() throws CloneNotSupportedException

如果某个类的对象要想被克隆，则对象所在的类必须实现Cloneable接口。此接口没有定义任何方法，是一个标记接口。

|  |
| --- |
| class Test {  public static void main(String[] args) {  Cat cat = new Cat("咪咪", 2);  try {  Cat newCat = (Cat)cat.clone();  System.out.println("Cat:"+cat);  System.out.println("newCat:"+newCat);  } catch (CloneNotSupportedException e){  e.printStackTrace();  }    }  }  /\*\*  克隆一个对象需要实现接口Cloneable,和重写方法clone() \*/  class Cat implements Cloneable{  private String name;  private int age;  public Cat(){super();}  public Cat(String name, int age) {  super();  this.name = name;  this.age = age;  }  public String toString() {  return "Cat [name="+ name +", age="+ age +"]";  }  //重写Object中的clone方法  protected Object clone() throws CloneNotSupportedException {  return super.clone();  }  } |

## 九．System与Runtime类

System类代表系统，系统级的很多属性和控制方法都放置在该类的内部。该类位于java.lang包。

**1、成员变量**

System类内部包含in、out和err三个成员变量，分别代表标准输入流(键盘输入)，标准输出流(显示器)和标准错误输出流。

**2、成员方法**

System类中提供了一些系统级的操作方法

（1）public static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)

该方法的作用是数组拷贝，也就是将一个数组中的内容复制到另外一个数组中的指定位置，由于该方法是native方法，所以性能上比使用循环高效。

|  |
| --- |
| import java.util.Arrays;  class Test {  public static void main(String[] args) {  int[] arr = {1,2,3,4,5,6,7,8};  int[] arr2 = new int[arr.length];  //参数（源数组、源数组的起始位置、目标数组、目标数组的起始位置、长度）  System.arraycopy(arr,0,arr2,0,arr.length);  System.out.println(Arrays.toString(arr2));  }  } |

（2）public static long currentTimeMillis()

该方法的作用是返回当前的计算机时间，时间的表达格式为当前计算机时间和GMT时间(格林威治时间)1970年1月1号0时0分0秒所差的毫秒数。

|  |
| --- |
| import java.util.Date;  import java.text.DateFormat;  import java.text.SimpleDateFormat;  class Test {  public static void main(String[] args) {  System.out.println(System.currentTimeMillis());  Date newDate = new Date(System.currentTimeMillis());  DateFormat df= new SimpleDateFormat("HH:mm:ss SSS");  System.out.println(df.format(newDate));  }  } |

（3）public static void exit(int status)

该方法的作用是退出程序。其中status的值为0代表正常退出，非零代表异常退出。使用该方法可以在图形界面编程中实现程序的退出功能等。

|  |
| --- |
| class Test {  public static void main(String[] args) {  //退出JVM  System.exit(0);  }  } |

（4）public static void gc()

该方法的作用是请求系统进行垃圾回收。至于系统是否立刻回收，则取决于系统中垃圾回收算法的实现以及系统执行时的情况。

（5）public static String getProperty(String key)

该方法的作用是获得系统中属性名为key的属性对应的值。

|  |
| --- |
| public static void main(String[] args) {  System.out.println("Java 运行时环境版本："+System.getProperty("java.version"));  System.out.println("Java 安装目录:"+System.getProperty("java.home"));  System.out.println("操作系统的名称:"+System.getProperty("os.name"));  System.out.println("操作系统的版本:"+System.getProperty("os.version"));  System.out.println("用户的账户名称:"+System.getProperty("user.name"));  System.out.println("用户的主目录:"+System.getProperty("user.home"));  System.out.println("用户的当前工作目录:"+System.getProperty("user.dir"));  } |

Runtime类：每个 Java 应用程序都有一个 Runtime 类实例，使应用程序能够与其运行的环境相连接。

|  |
| --- |
| import java.io.IOException;  class Test {  public static void main(String[] args) {  //获取Java运行时相关的运行时对象  Runtime rt = Runtime.getRuntime();  System.out.println("处理器数量：" + rt.availableProcessors()+" 个");  System.out.println("Jvm总内存数 ："+ rt.totalMemory()+" byte");  System.out.println("Jvm空闲内存数： "+ rt.freeMemory()+" byte");  System.out.println("Jvm可用最大内存数： "+ rt.maxMemory()+" byte");  //在单独的进程中执行指定的字符串命令。  try {  rt.exec("notepad"); //打开记事本  } catch (IOException e){  e.printStackTrace();  }  }  } |

## 十．数字处理工具

BigInteger：可以让超过Integer范围内的数据进行运算。

构造方法：public BigInteger(String val)

常用方法：

public BigInteger add(BigInteger val)

public BigInteger subtract(BigInteger val)

public BigInteger multiply(BigInteger val)

public BigInteger divide(BigInteger val)

public BigInteger[] divideAndRemainder(BigInteger val)

|  |
| --- |
| import java.math.BigInteger;  import java.util.Arrays;  class Test {  public static void main(String[] args) {  //大整数运算  String val1 = "8456772356784748385";  String val2 = "5434536454657543444";  BigInteger b1 = new BigInteger(val1);  BigInteger b2 = new BigInteger(val2);  System.out.println("加法运算+：" + b1.add(b2));  System.out.println("减法运算-：" + b1.subtract(b2));  System.out.println("乘法运算\*：" + b1.multiply(b2));  System.out.println("除法运算/：" + b1.divide(b2));  System.out.println("取余运算%：" + b1.remainder(b2));  System.out.println("除法并且取余运算/and%：" + Arrays.toString(b1.divideAndRemainder(b2)));  }  } |

BigDecimal：由于在运算的时候，float类型和double很容易丢失精度，为了能精确的表示、计算浮点数，Java提供了BigDecimal，不可变的、任意精度的有符号十进制数。

构造方法：public BigDecimal(String val)

常用方法：

public BigDecimal add(BigDecimal augend)

public BigDecimal subtract(BigDecimal subtrahend)

public BigDecimal multiply(BigDecimal multiplicand)

public BigDecimal divide(BigDecimal divisor)

|  |
| --- |
| import java.math.BigDecimal;  class Test {  public static void main(String[] args) {  //大小数运算  String val1 = "8456.772356784748385";  String val2 = "5";  BigDecimal b1 = new BigDecimal(val1);  BigDecimal b2 = new BigDecimal(val2);  System.out.println("加法运算+：" + b1.add(b2));  System.out.println("减法运算-：" + b1.subtract(b2));  System.out.println("乘法运算\*：" + b1.multiply(b2));  System.out.println("除法运算/：" + b1.divide(b2)); //注意：除不尽的问题出现  }  } |

DecimalFormat：Java 提供 DecimalFormat类，帮你用最快的速度将数字格式化为你需要的样子。

例如，取2位小数。

|  |
| --- |
| import java.text.DecimalFormat;  class Test {  public static void main(String[] args) {  double pi = 3.1415927;  //取一位整数，结果：3  System.out.println(new DecimalFormat("0").format(pi));  //取一位整数和两位小数，结果3.14  System.out.println(new DecimalFormat("0.00").format(pi));  //取两位整数和三位小数，整数不足部分以0填补，结果：03.142  System.out.println(new DecimalFormat("00.000").format(pi));  //取所有整数部分，结果：3  System.out.println(new DecimalFormat("#").format(pi));  //以百分比方式计数，并取两位小数，结果：314.16%  System.out.println(new DecimalFormat("#.##%").format(pi));  long l = 345678934;  System.out.println(new DecimalFormat("###,###").format(l));  }  } |

## 十一.MD5工具类

MD5的全称是Message-Digest Algorithm 5（信息-摘要算法）

//确定计算方法

MessageDigest md5=MessageDigest.getInstance("MD5");

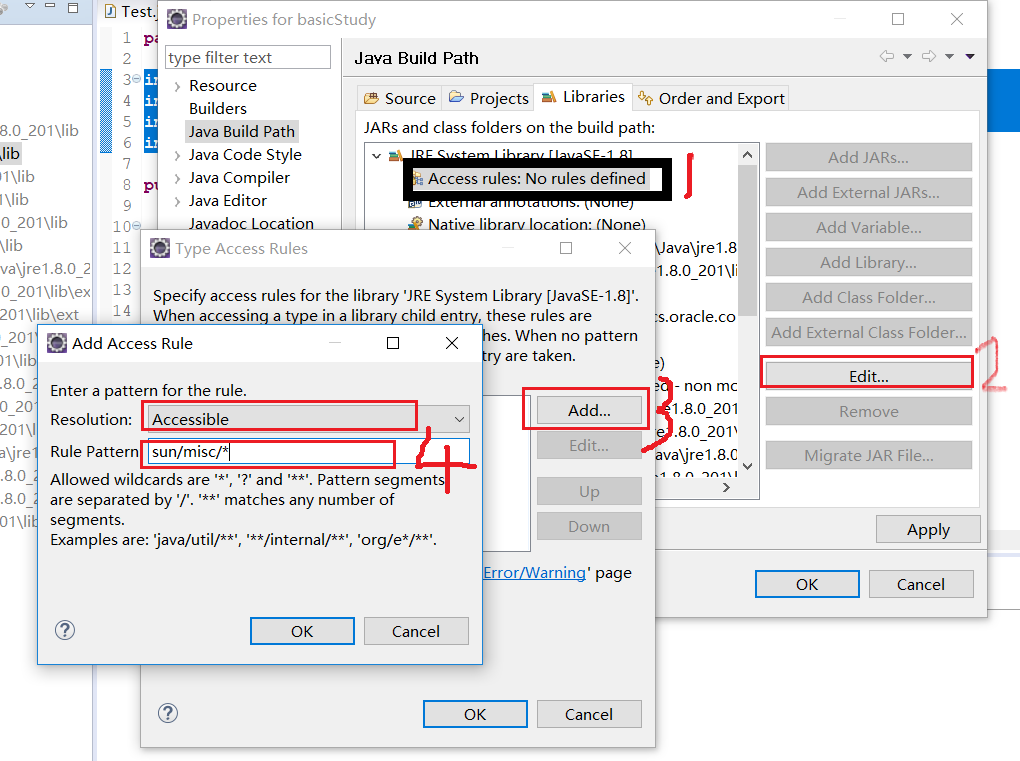
//JDK1.8新增Base64

String newstr = Base64.getEncoder().encodeToString(md5.digest(str.getBytes("utf-8")));

|  |
| --- |
| import java.io.UnsupportedEncodingException;  import java.security.MessageDigest;  import java.security.NoSuchAlgorithmException;  import java.util.Base64;  class Test {  public static void main(String[] args) {  try {  String password = "admin123456";  //设置md5算法 (还有SHA-1算法)  MessageDigest md = MessageDigest.getInstance("md5");  //通过MD5计算摘要  byte[] digest = md.digest(password.getBytes("UTF-8"));  //使用a-z A-Z 0-9 / \*(BASE64编码)进行表示  //1.8版本  String newpassword = Base64.getEncoder().encodeToString(digest);  System.out.println(newpassword);  //结果：pmq7VoTEWWLYh1ZPCDRujQ==  } catch (NoSuchAlgorithmException e) {  // TODO Auto-generated catch block  e.printStackTrace();  } catch (UnsupportedEncodingException e) {  // TODO Auto-generated catch block  e.printStackTrace();  }  }  } |

//1.8之前使用sun.misc.BASE64Encoder(此类没有访问权限，在rt.jar中添加访问权限：sun/misc/\*)

Eclipse配置访问权限



配置完访问权限后使用

|  |
| --- |
| import java.io.UnsupportedEncodingException;  import java.security.MessageDigest;  import java.security.NoSuchAlgorithmException;  import sun.misc.BASE64Encoder;  class Test {  public static void main(String[] args) {  try {  String password = "admin123456";  //设置md5算法 (还有SHA-1算法)  MessageDigest md = MessageDigest.getInstance("md5");  //通过MD5计算摘要  byte[] digest = md.digest(password.getBytes("UTF-8"));  //使用a-z A-Z 0-9 / \*(BASE64编码)进行表示  //1.8版本之前  BASE64Encoder base64 = new BASE64Encoder();  String newpassword = base64.encode(digest);  System.out.println(newpassword);  //结果：pmq7VoTEWWLYh1ZPCDRujQ==  } catch (NoSuchAlgorithmException e) {  // TODO Auto-generated catch block  e.printStackTrace();  } catch (UnsupportedEncodingException e) {  // TODO Auto-generated catch block  e.printStackTrace();  }  }  } |

## 十二.数据结构之二叉树实现

树是一种重要的非线性数据结构，直观地看，它是数据元素（在树中称为结点）按分支关系组织起来的结构。二叉树（Binary Tree）是每个节点最多有两个子树的有序树。通常子树被称作“左子树” 和“右子树”。

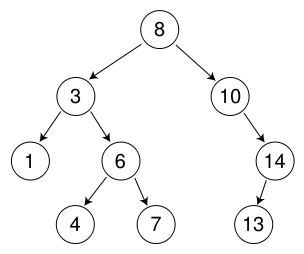
**二叉树算法的排序规则：**

1、选择第一个元素作为根节点

2、之后如果元素大于根节点放在右子树，如果元素小于根节点，则放在左子树

3、最后按照中序遍历的方式进行输出，则可以得到排序的结果（左🡪根🡪右）

8、3、10、1、6、14、4、7、13



|  |
| --- |
| **class Test {**  **public static void main(String[] args) {**  **BinaryTree bt = new BinaryTree();**  **bt.add(8);**  **bt.add(3);**  **bt.add(10);**  **bt.add(1);**  **bt.add(6);**  **bt.add(14);**  **bt.add(4);**  **bt.add(7);**  **bt.add(13);**  **bt.print();**  **}**  **}**  **class BinaryTree {**  **private Node node;**  **public void add (int data) {**  **if (this.node == null) {**  **this.node = new Node(data);**  **} else {**  **this.node.addNode(data);**  **}**  **}**  **public void print() {**  **this.node.printNode();**  **}**  **private class Node {**  **private int data;**  **private Node left;**  **private Node right;**  **public Node (int data) {**  **this.data = data;**  **}**  **public void addNode(int data) {**  **if (this.data >= data ) {**  **if (this.left == null) {**  **this.left = new Node(data);**  **} else {**  **this.left.addNode(data);**  **}**  **} else {**  **if (this.right == null) {**  **this.right = new Node(data);**  **} else {**  **this.right.addNode(data);**  **}**  **}**  **}**  **public void printNode() {**  **if (this.left != null) {**  **this.left.printNode();**  **}**  **System.out.print(this.data+"->");**  **if (this.right != null) {**  **this.right.printNode();**  **}**  **}**  **}**  **}** |

## 十三. JDK1.8新特性-Lambda表达式

**1、Lambda表达式**

Lambda表达式（也称为闭包）是整个Java 8发行版中最受期待的在Java语言层面上的改变，Lambda允许把函数作为一个方法的参数（函数作为参数传递进方法中），或者把代码看成数据。Lambda表达式用于简化JAVA中接口式的匿名内部类。被称为函数式接口的概念。函数式接口就是一个具有一个方法的普通接口。像这样的接口，可以被隐式转换为lambda表达式。

语法：

(参数1,参数2...) -> { ... }

1. 没有参数时使用Lambda表达式

|  |
| --- |
| class Test {  public static void main(String[] args) {  IEat eat = new IEatImpl();  eat.eat();  IEat eat2 = new IEat() {  public void eat() {  System.out.println("eat bnana");  }  };  eat2.eat();  //lambda表达式  //1.代码简洁，2不会单独生成class文件  //限制 接口中只能有一个方法  //IEat eat3 = () -> {System.out.println("eat apple bnana");};  IEat eat3 = () -> System.out.println("eat apple bnana");  eat3.eat();  }  }  interface IEat {  public void eat();  }  class IEatImpl implements IEat {  public void eat() {  System.out.println("eat apple");  }  } |

1. 带参数时使用Lambda表达式

|  |
| --- |
| class Test {  public static void main(String[] args) {  IEat eat = (String name) -> System.out.println(name);  eat.eat("eat apple");  }  }  interface IEat {  public void eat(String name);  } |

1. 代码块中只一句代码时使用Lambda表达式

|  |
| --- |
| IEat eat = (String name) -> System.out.println(name); |

1. 代码块中有多句代码时使用Lambda表达式

|  |
| --- |
| IEat eat = (String name) -> {  name += "好吃";  System.out.println(name);  }; |

1. 有返回值的代码块

|  |
| --- |
| IEat eat2 = (String name) -> 5;  或  IEat eat = (String name) -> {  return 5;  }; |

1. 参数中使用final关键字

|  |
| --- |
| IEat eat = (final String name) -> name == null ? 0:1; |

**2、接口中的默认方法和静态方法**

interface A{

public **default** void print(){}

public **static** void method(){}

}

默认方法与静态方法并不影响函数式接口的契约，可以直接使用

|  |
| --- |
| interface IEat {  public int eat(final String name);  public default void print(){  System.out.println("我是默认的");  }  public static void method(){  System.out.println("我是静态的");  }  } |

# 第七章 文件与IO

## 一．File类

**1、File类的基本概念**

File类：表示文件和目录路径名的抽象表示形式。

File类可以实现文件的创建、删除、重命名、得到路径、创建时间等等，是唯一与文件本身有关的操作类。

**2、File类的操作方法**

|  |  |
| --- | --- |
| public static final String separator | 表示路径分隔符“\” |
| public File(String pathname) | 构造File类实例，要传入路径 |
| public boolean createNewFile() | 创建新文件 |
| public boolean delete() | 删除文件 |
| public boolean isDirectory() | 判断给定的路径是否是文件夹 |
| public boolean isFile() | 判断给定的路径是否是文件 |
| public String[] list() | 列出文件夹中的文件 |
| public File[] listFiles() | 列出文件夹中的所有文件 |
| public boolean mkdir() | 创建新的文件夹 |
| public boolean renameTo(File dest) | 为文件重命名 |
| public long length() | 返回文件大小 |
| String getPath() | 路径名字符串。 |

|  |
| --- |
| import java.io.File;  import java.io.IOException;  class Test {  public static void main(String[] args) {  // String path = "F:/test/test.txt";  // String path = "F:\\test\\test.txt";  String path = "F:"+File.separator+"test"+File.separator+"test.txt";  File file = new File(path);  if(!file.exists()) { //文件是否存在  try {  boolean isSuccessful = file.createNewFile(); //创建文件  if(isSuccessful) {  System.out.println("文件创建成功！");  }  } catch (IOException e) {  e.printStackTrace();  }  }  System.out.println("文件相对路径："+file.getPath());  System.out.println("文件绝对路径："+file.getAbsolutePath());  System.out.println("是否是文件："+file.isFile());  System.out.println("是否是文件夹："+file.isDirectory());  System.out.println("删除文件和空文件夹："+file.delete());  //当前目录下所有的文件名  String[] names = new File("F:\\test").list();  //获取当前目录下所有文件  File[] files = new File("F:\\test").listFiles();    }  } |

## 二．字节流

**1、IO流概述**

IO流：输入输出流（Input/Output）

流是一组有顺序的，有起点和终点的字节集合，是对数据传输的总称或抽象。即数据在两设备间的传输称为流

**流的本质是数据传输，根据数据传输特性将流抽象为各种类，方便更直观的进行数据操作。**

**IO流的分类**

根据处理数据类型的不同分为：字符流和字节流

根据数据流向不同分为：输入流和输出流

**2、字节输出流**

OutputStream类定义

public abstract class OutputStream extends Object implements Closeable, Flushable

此抽象类是表示输出字节流的所有类的超类。输出流接受输出字节并将这些字节发送到InputStream 类某个接收器要向文件中输出，使用FileOutputStream类

**3、字节输入流**

定义：

public abstract class InputStream extends Object implements Closeable

此抽象类是表示字节输入流的所有类的超类。

FileInputStream 从文件系统中的某个文件中获得输入字节。

|  |
| --- |
| import java.io.File;  import java.io.OutputStream;  import java.io.FileOutputStream;  import java.io.InputStream;  import java.io.FileInputStream;  import java.io.IOException;  import java.io.FileNotFoundException;  /\*\*  输出流：超类OutputStream, 对文件输出流使用子类FileOutputStream  输入流：超类InputStream, 对文件的输入流使用子类FileInputStream  每次操作一个字节  每次直接写入文件，没有缓存。  \*/  class Test {  public static void main(String[] args) {  out();  in();  }  //输出到文件中  public static void out() {  File file = new File("F:\\test\\test.txt");  try {  // OutputStream out = new FileOutputStream(file);  OutputStream out = new FileOutputStream(file, true); //append为true表示追加内容  String info = "小河流水哗啦啦！！";  out.write(info.getBytes());  out.close();  } catch (FileNotFoundException e) {  e.printStackTrace();  } catch (IOException e) {  e.printStackTrace();  }  }  //从文件中输入到程序  public static void in() {  File file = new File("F:\\test\\test.txt");  try {  InputStream in = new FileInputStream(file);  byte[] bytes = new byte[1024];  StringBuilder buf = new StringBuilder();  int len = -1;  while((len = in.read(bytes)) != -1) {  //将读取的字节数组转换成字符串内容  buf.append(new String(bytes, 0, len));  }  System.out.println(buf);  in.close();  } catch (FileNotFoundException e) {  e.printStackTrace();  } catch (IOException e) {  e.printStackTrace();  }  }  } |

## 三．字符流

**Writer**

写入字符流的抽象类。子类必须实现的方法仅有 write(char[], int, int)、flush() 和 close()。但是，多数子类将重写此处定义的一些方法，以提供更高的效率和/或其他功能。

与OutputStream一样，对文件的操作使用：FileWriter类完成。

**Reader**用于读取字符流的抽象类。子类必须实现的方法只有 read(char[], int, int) 和 close()。但是，多数子类将重写此处定义的一些方法，以提供更高的效率和/或其他功能。

使用FileReader类进行实例化操作。

|  |
| --- |
| import java.io.File;  import java.io.Writer;  import java.io.FileReader;  import java.io.Reader;  import java.io.FileWriter;  import java.io.IOException;  import java.io.FileNotFoundException;  /\*\*  字符输出流：Writer, 对文件输出流使用子类FileWriter  字符输入流：Reader, 对文件的输入流使用子类FileReader  每次操作的单位是一个字符  文件字符操作流会自带缓存，默认大小为1024字节 当缓存满后、或手动刷新缓存、或关闭流时会把数据写入文件。  \*/  class Test {  public static void main(String[] args) {  out();  in();  }  //输出到文件中  public static void out() {  File file = new File("F:\\test\\test.txt");  try {  Writer out = new FileWriter(file);  // Writer out = new FileWriter(file, true); //append为true表示追加内容  String info = "村姑到我家";  out.write(info);  out.close();  } catch (FileNotFoundException e) {  e.printStackTrace();  } catch (IOException e) {  e.printStackTrace();  }  }  //从文件中输入到程序  public static void in() {  File file = new File("F:\\test\\test.txt");  try {  Reader in = new FileReader(file);  char[] cs = new char[1];  StringBuilder buf = new StringBuilder();  int len = -1;  while((len = in.read(cs)) != -1) {  //将读取的字符数组转换成字符串内容  buf.append(new String(cs, 0, len));  }  System.out.println(buf);  in.close();  } catch (FileNotFoundException e) {  e.printStackTrace();  } catch (IOException e) {  e.printStackTrace();  }  }  } |

**字节流与字符流的区别**

在所有的流操作里。字节永远是最基础的。任何基于字节的操作都是正确的。无论你是文本文件还是二进制的文件。 如果确认流里面只有可打印的字符，包括英文的和各种国家的文字，也包括中文，那么可以考虑用字符流。 由于编码不同，多字节的字符可能占用多个字节。比如GBK的汉字就占用2个字节，而UTF-8的汉字就占用3个字节。 所以，字符流是根据指定的编码，将1个或多个字节转化为java里面的unicode的字符，然后进行操作。 字符操作一般使用Writer,Reader等， 字节操作一般都是InputStream, OutputStream 以及各种包装类，比如BufferedInputStream和BufferedOutputStream等。   
  
总结：如果你确认你要处理的流是可打印的字符，那么使用字符流会看上去简单一点。如果不确认，那么用字节流总是不会错的。

指定一个盘符下的文件，把该文件复制到指定的目录下。

|  |
| --- |
| public static void copy() {  File sourceFile = new File("F:\\test\\test.txt");  File newFile = new File("F:\\test\\test2.txt");  InputStream in = null;  OutputStream out = null;  try {  in = new FileInputStream(sourceFile);  out = new FileOutputStream(newFile);  byte[] bytes = new byte[1024];  int len = -1;  while((len = in.read(bytes)) != -1) {  out.write(bytes, 0, len);  }  out.close();  } catch (FileNotFoundException e) {  e.printStackTrace();  } catch (IOException e) {  e.printStackTrace();  } finally {  try {  if(in != null) in.close();  if(out != null) out.close();  } catch (IOException e) {  e.printStackTrace();  }  }  } |

## 四．字节字符转换流

转换流，可以将一个字节流转换为字符流，也可以将一个字符流转换为字节流。

OutputStreamWriter：可以将输出的字符流转换为字节流的输出形式

InputStreamReader：将输入的字节流转换为字符流输入形式

|  |
| --- |
| //输出到文件中  public static void wrile (){  try {  OutputStream out = new FileOutputStream("F:\\test\\test.txt");  Writer writer= new OutputStreamWriter(out, Charset.defaultCharset());  String info = "小河流水哗啦啦！！";  writer.write(info);  writer.close();  out.close();  } catch (FileNotFoundException e) {  e.printStackTrace();  } catch (IOException e) {  e.printStackTrace();  }  }  //从文件中输入到程序  public static void read() {  try {  InputStream in = new FileInputStream("F:\\test\\test.txt");  Reader reader = new InputStreamReader(in,Charset.defaultCharset());  byte[] bytes = new byte[1024];  StringBuilder buf = new StringBuilder();  int len = -1;  while((len = in.read(bytes)) != -1) {    buf.append(new String(bytes, 0, len));  }  System.out.println(buf);  reader.close();  in.close();  } catch (FileNotFoundException e) {  e.printStackTrace();  } catch (IOException e) {  e.printStackTrace();  }  } |

## 五．缓冲流

首先要明确一个概念：

对文件或其它目标频繁的读写操作，效率低，性能差。

使用缓冲流的好处是，能够更高效的读写信息，原理是将数据先缓冲起来，然后一起写入或者读取出来。

**BufferedInputStream：** 为另一个输入流添加一些功能，在创建 BufferedInputStream 时，会创建一个内部缓冲区数组，用于缓冲数据。

**BufferedOutputStream：**通过设置这种输出流，应用程序就可以将各个字节写入底层输出流中，而不必针对每次字节写入调用底层系统。

**BufferedReader：**从字符输入流中读取文本，缓冲各个字符，从而实现字符、数组和行的高效读取。

**BufferedWriter：**将文本写入字符输出流，缓冲各个字符，从而提供单个字符、数组和字符串的高效写入。

|  |
| --- |
| 字节缓冲流  import java.io.BufferedOutputStream;  import java.io.OutputStream;  import java.io.FileOutputStream;  import java.io.InputStream;  import java.io.FileInputStream;  import java.io.BufferedInputStream;  import java.io.IOException;  import java.io.FileNotFoundException;  import java.nio.charset.Charset;  /\*\*  缓存的目的  解决写入文件操作时，频繁的操作文件所带来的性能降低的问题  默认缓冲大小8K  \*/  class Test {  public static void main(String[] args) {  wrile();  read();  }    //输出到文件中  public static void wrile (){  try {  OutputStream out = new FileOutputStream("F:\\test\\test.txt");  BufferedOutputStream bos = new BufferedOutputStream(out);  String info = "小河流水继续哗啦啦！！";  bos.write(info.getBytes());  bos.close();  } catch (FileNotFoundException e) {  e.printStackTrace();  } catch (IOException e) {  e.printStackTrace();  }  }  //从文件中输入到程序  public static void read() {  try {  InputStream in = new FileInputStream("F:\\test\\test.txt");  BufferedInputStream bis = new BufferedInputStream(in);  byte[] bytes = new byte[1024];  StringBuilder buf = new StringBuilder();  int len = -1;  while((len = bis.read(bytes)) != -1) {    buf.append(new String(bytes, 0, len));  }  System.out.println(buf);  bis.close();  } catch (FileNotFoundException e) {  e.printStackTrace();  } catch (IOException e) {  e.printStackTrace();  }  }  } |

|  |
| --- |
| 字符缓冲流  import java.io.FileWriter;  import java.io.Writer;  import java.io.BufferedWriter;  import java.io.Reader;  import java.io.FileReader;  import java.io.BufferedReader;  import java.io.IOException;  import java.io.FileNotFoundException;  import java.nio.charset.Charset;  /\*\*  字符流  1.加入字符缓存流，增强读取功能（readLine）  2.  缓存的目的  解决写入文件操作时，频繁的操作文件所带来的性能降低的问题  默认缓冲大小8kb  \*/  class Test {  public static void main(String[] args) {  wrile();  read();  }    //输出到文件中  public static void wrile (){  try {  Writer writer = new FileWriter("F:\\test\\test.txt");  BufferedWriter bw = new BufferedWriter(writer);  String info = "小河流水在继续哗啦啦！！";  bw.write(info);  bw.close();  } catch (FileNotFoundException e) {  e.printStackTrace();  } catch (IOException e) {  e.printStackTrace();  }  }  //从文件中输入到程序  public static void read() {  try {  Reader in = new FileReader("F:\\test\\test.txt");  BufferedReader br = new BufferedReader(in);  char[] c = new char[1];  StringBuilder buf = new StringBuilder();  int len = -1;  while((len = br.read(c)) != -1) {  buf.append(new String(c, 0, len));  }  System.out.println(buf);  br.close();  } catch (FileNotFoundException e) {  e.printStackTrace();  } catch (IOException e) {  e.printStackTrace();  }  }  } |

## 六．打印流

打印流的主要功能是用于输出，在整个IO包中打印流分为两种类型：

字节打印流：PrintStream

字符打印流：PrintWriter

打印流可以很方便的进行输出

|  |
| --- |
| 字节打印流  public static void printStream (){  try {  OutputStream out = new FileOutputStream("F:\\test\\test.txt");  BufferedOutputStream bos = new BufferedOutputStream(out);  PrintStream ps = new PrintStream(bos);  String info = "小河流水继续哗啦啦！！";  ps.println(info);  ps.close();  bos.close();  } catch (FileNotFoundException e) {  e.printStackTrace();  } catch (IOException e) {  e.printStackTrace();  }  }  字符打印流  public static void printWriter (){  try {  Writer out = new FileWriter("F:\\test\\test.txt");  BufferedWriter bw = new BufferedWriter(out);  PrintWriter pw = new PrintWriter(bw);  String info = "小河流水天天哗啦啦！！";  pw.println(info);  pw.close();  bw.close();  } catch (FileNotFoundException e) {  e.printStackTrace();  } catch (IOException e) {  e.printStackTrace();  }  } |

## 七．对象流

**对象流的两个类：**

ObjectOutputStream 将 Java 对象的基本数据类型和图形写入 OutputStream

ObjectInputStream 对以前使用 ObjectOutputStream 写入的基本数据和对象进行反序列化。

**序列化一组对象：**

在序列化操作中，同时序列化多个对象时，反序列化也必须按顺序操作，如果想要序列化一组对象该如何操作呢？

序列化一组对象可采用：对象数组的形式，因为对象数组可以向Object进行转型操作。

**transient关键字：**

如果用transient声明一个实例变量，当对象存储时，它的值不需要维持。

|  |
| --- |
| import java.io.Serializable;  import java.io.OutputStream;  import java.io.FileOutputStream;  import java.io.ObjectOutputStream;  import java.io.InputStream;  import java.io.FileInputStream;  import java.io.ObjectInputStream;  import java.io.FileNotFoundException;  import java.io.IOException;  class Test {  public static void main(String[] args) {  // writeObject();  readObject();  }  /\*\*  对象序列化  把对象写入文件：实际写入的是类名、属性名、属性类型、属性的值等  \*/  private static void writeObject (){  Dog dog = new Dog("旺旺", 18, "小伞");  try {  OutputStream out = new FileOutputStream("F:\\test\\test.txt");  ObjectOutputStream oos = new ObjectOutputStream(out);  oos.writeObject(dog);  oos.close();  } catch (FileNotFoundException e) {  e.printStackTrace();  } catch (IOException e) {  e.printStackTrace();  }  }  /\*\*  对象反序列化  从文件中把对象的内容读取出来，还原成对象  \*/  private static void readObject (){  try {  InputStream in = new FileInputStream("F:\\test\\test.txt");  ObjectInputStream ois = new ObjectInputStream(in);  Dog dog= (Dog)ois.readObject();  ois.close();  System.out.println(dog);  } catch (FileNotFoundException e) {  e.printStackTrace();  } catch (IOException e) {  e.printStackTrace();  } catch (ClassNotFoundException e) {  e.printStackTrace();  }  }  }  /\*\*  如果一个类创建的时候需要被序列化，那么该类必须实现Serializable接口  Serializable是一个标记接口，没有任何定义，为了告诉JVM该类对象可以被序列化  什么时候对象需要被序列化呢？  1.把对象保存到文件中（存储到物理介质）  2.对象需要在网络上传播  \*/  class Dog implements Serializable {  //被transient修饰说明此属性在序列化中被忽略  private transient int id;  private String name;  private int age;  private String set;  public Dog() {  super();  }  public Dog(String name, int age, String set) {  super();  this.name = name;  this.age = age;  this.set = set;  }  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 getSet() {  return set;  }  public void setSet(String set) {  this.set = set;  }  @Override  public String toString() {  return "Dog [name=" + name + ", age=" + age + ", set=" + set + "]";  }  } |

## 八．字节数组流

**ByteArrayInputStream**

包含一个内部缓冲区，该缓冲区包含从流中读取的字节。内部计数器跟踪 read 方法要提供的下一个字节。 关闭 ByteArrayInputStream 无效。此类中的方法在关闭此流后仍可被调用，而不会产生任何 IOException。

**ByteArrayOutputStream**

此类实现了一个输出流，其中的数据被写入一个 byte 数组。缓冲区会随着数据的不断写入而自动增长。可使用 toByteArray() 和 toString() 获取数据。 关闭 ByteArrayOutputStream 无效。此类中的方法在关闭此流后仍可被调用，而不会产生任何 IOException。

|  |
| --- |
| import java.io.ByteArrayInputStream;  import java.io.ByteArrayOutputStream;  /\*\*  字节数组流：  基于内存操作，内部维护着一个字节数组，我们可以利用流的读取机制来处理字符串。  无需关闭。  \*/  class Test {  public static void main(String[] args) {  String str = "sgfdg45674SRT(\*^redfvfHJUVM%$#";  ByteArrayInputStream bais = new ByteArrayInputStream(str.getBytes());  ByteArrayOutputStream baos = new ByteArrayOutputStream();    int curr = -1;  while ((curr = bais.read()) != -1) {  //ASCII码 65~90代表a~z 97~122代表A~Z  if ((curr >= 65 && curr <= 90) || (curr >= 97 && curr <= 122)) {  baos.write(curr);  }  }  //此时无需关闭，因为字节数组流是基于内存的操作流  System.out.println(baos.toString());  //结果：sgfdgSRTredfvfHJUVM  }  } |

## 九．数据流

**DataInputStream：**

数据输入流允许应用程序以与机器无关方式从底层输入流中读取基本 Java 数据类型。应用程序可以使用数据输出流写入稍后由数据输入流读取的数据。 DataInputStream 对于多线程访问不一定是安全的。 线程安全是可选的，它由此类方法的使用者负责。

**DataOutputStream：**

数据输出流允许应用程序以适当方式将基本 Java 数据类型写入输出流中。然后，应用程序可以使用数据输入流将数据读入。

|  |
| --- |
| import java.io.File;  import java.io.OutputStream;  import java.io.FileOutputStream;  import java.io.BufferedOutputStream;  import java.io.DataOutputStream;  import java.io.InputStream;  import java.io.FileInputStream;  import java.io.BufferedInputStream;  import java.io.DataInputStream;  import java.io.FileNotFoundException;  import java.io.IOException;  class Test {  public static void main(String[] args) {  write ();  read ();  }  private static void write (){  File file = new File("F:/test/test.txt");  try {  OutputStream out = new FileOutputStream(file);  BufferedOutputStream bos = new BufferedOutputStream(out);  DataOutputStream dos = new DataOutputStream(bos);  dos.writeInt(10);  dos.writeByte(1);  dos.writeUTF("中");    dos.close();  bos.close();  } catch (FileNotFoundException e) {  e.printStackTrace();  } catch (IOException e) {  e.printStackTrace();  }  }  private static void read (){  File file = new File("F:/test/test.txt");  try {  InputStream in = new FileInputStream(file);  BufferedInputStream bis = new BufferedInputStream(in);  DataInputStream dis = new DataInputStream(bis);  int i = dis.readInt();  byte b = dis.readByte();  String u = dis.readUTF();  System.out.println(i+","+b+","+u);    dis.close();  bis.close();  } catch (FileNotFoundException e) {  e.printStackTrace();  } catch (IOException e) {  e.printStackTrace();  }  }  } |

## 十．实现文件分割合并

|  |
| --- |
| //文件分割  import java.io.File;  import java.io.OutputStream;  import java.io.FileOutputStream;  import java.io.BufferedOutputStream;  import java.io.DataOutputStream;  import java.io.InputStream;  import java.io.FileInputStream;  import java.io.BufferedInputStream;  import java.io.DataInputStream;  import java.io.FileNotFoundException;  import java.io.IOException;  class Test {  public static void main(String[] args) {  File file = new File("F:/test/IO\_14.avi");  division(file, 1024\*1024\*20);  }  /\*\*  文件分割  targetFile 要分割的目标文件  cutSize 每个文件大小  \*/  public static void division (File targetFile, long cutSize) {  if(targetFile == null) return;  int num = targetFile.length()%cutSize == 0  ? (int)(targetFile.length()/cutSize)  : (int)(targetFile.length()/cutSize+1);  try {  //构造文件输入流  BufferedInputStream in = new BufferedInputStream(new FileInputStream(targetFile));  BufferedOutputStream out = null;    byte[] bytes = null;//每次要读取的字节数  int len = -1;  int count = 0;//每一个文件要读取的次数  for(int i = 0; i <= num; i++) {  out = new BufferedOutputStream(new FileOutputStream("F:/test/切割/"+(i+1)+".avi"));  if(cutSize <= 1024) {  bytes = new byte[(int)cutSize];  count = 1;  } else {  bytes = new byte[1024];  count = (int)cutSize/1024;  }  while (count > 0 && (len = in.read(bytes)) != -1) {  out.write(bytes, 0, len);  out.flush();  count--;  }  if(cutSize%1024 != 0) {  bytes = new byte[(int)cutSize%1024];  len = in.read(bytes);  out.write(bytes, 0, len);  out.flush();  out.close();  }  }  in.close();  } catch (FileNotFoundException e) {  e.printStackTrace();  } catch (IOException e) {  e.printStackTrace();  }  }  } |

|  |
| --- |
| //文件合并  import java.io.File;  import java.io.OutputStream;  import java.io.FileOutputStream;  import java.io.BufferedOutputStream;  import java.io.DataOutputStream;  import java.io.InputStream;  import java.io.FileInputStream;  import java.io.BufferedInputStream;  import java.io.DataInputStream;  import java.io.SequenceInputStream;  import java.io.FileNotFoundException;  import java.io.IOException;  import java.util.Enumeration;  import java.util.Vector;  class Test {  public static void main(String[] args) {  // File file = new File("F:/test/IO\_14.avi");  // division(file, 1024\*1024\*20);  try {  InputStream is1 = new FileInputStream(new File("F:\\test\\切割\\1.avi"));  InputStream is2 = new FileInputStream(new File("F:\\test\\切割\\2.avi"));  InputStream is3 = new FileInputStream(new File("F:\\test\\切割\\3.avi"));  InputStream is4 = new FileInputStream(new File("F:\\test\\切割\\4.avi"));    Vector<InputStream> vc = new Vector<InputStream>();  vc.add(is1);  vc.add(is2);  vc.add(is3);  vc.add(is4);    Enumeration<InputStream> es = vc.elements();  merge(es);  } catch (FileNotFoundException e) {  e.printStackTrace();  }  }  /\*\*  文件合并  \*/  public static void merge (Enumeration<InputStream> es) {  SequenceInputStream sis = new SequenceInputStream(es);  try {  BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream("F:\\test\\切割\\合并后.avi"));    byte[] b = new byte[1024];  int len = -1;  while ((len = sis.read(b)) != -1){  bos.write(b, 0, len);  bos.flush();  }  bos.close();  sis.close();  } catch (FileNotFoundException e) {  e.printStackTrace();  } catch (IOException e) {  e.printStackTrace();  }  }  } |

## 十一. 合并流、字符串流、管道流

**一、合并流：**

**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。

**二、字符串流**

**1、StringReader**

其源为一个字符串的字符流。

|  |
| --- |
| import java.io.StringReader;  import java.io.StreamTokenizer;  /\*\*  字符串流：以一个字符串为数据源，来构造一个字符流  作用：在WEB开发中，我们经常要从服务器上获取数据，数据的返回格式通常是一个字符串（xml、json），  我们需要把这个字符串构造成一个字符流，然后再用第三方的数据解析器来解析数据  \*/  public static void stringReader () {  String info = "good good study day day up";  StringReader sr = new StringReader(info);  //流标记器  StreamTokenizer st = new StreamTokenizer(sr);    int count = 0;  //循环字符类型不到结尾  while (st.ttype != StreamTokenizer.TT\_EOF) {  try {  //判断下一个节点是否为一个单词  if (st.nextToken() == StreamTokenizer.TT\_WORD) {  count++;  }  } catch (IOException e) {  e.printStackTrace();  }  }  sr.close();  System.out.print(count);  } |

**2、 StringWriter**

一个字符流，可以用其回收在字符串缓冲区中的输出来构造字符串。 关闭 StringWriter 无效。此类中的方法在关闭该流后仍可被调用，而不会产生任何 IOException。

**三、管道流**

管道输入流应该连接到管道输出流；管道输入流提供要写入管道输出流的所有数据字节。通常，数据由某个线程从 PipedInputStream 对象读取，并由其他线程将其写入到相应的 PipedOutputStream。不建议对这两个对象尝试使用单个线程，因为这样可能死锁线程。管道输入流包含一个缓冲区，可在缓冲区限定的范围内将读操作和写操作分离开。如果向连接管道输出流提供数据字节的线程不再存在，则认为该管道已损坏。

|  |
| --- |
| /\*\*  一个线程写入，一个线程读取  \*/  import java.io.IOException;  import java.io.PipedInputStream;  import java.io.PipedOutputStream;  public class Test {  public static void main(String[] args) {  PipedInputStream pipedInputStream = new PipedInputStream();  PipedOutputStream pipedOutputStream = new PipedOutputStream();  try {  //两个管道进行连接  pipedInputStream.connect(pipedOutputStream);  } catch (IOException e) {  e.printStackTrace();  }  ReadThread readThread = new ReadThread(pipedInputStream);  WriteThread writeThread = new WriteThread(pipedOutputStream);  new Thread(readThread).start();  new Thread(writeThread).start();  }  }  //读取数据的线程  class ReadThread implements Runnable {  private PipedInputStream pipedInputStream; //输入管道    public ReadThread(PipedInputStream pipedInputStream) {  this.pipedInputStream = pipedInputStream;  }  @Override  public void run() {  try {  byte[] bytes = new byte[1024];  int len = pipedInputStream.read(bytes);  String str = new String(bytes, 0, len);  System.out.println("读到："+str);  pipedInputStream.close();  } catch (IOException e) {  e.printStackTrace();  }  }  }  //写入数据的线程  class WriteThread implements Runnable {  private PipedOutputStream pipedOutputStream; //输出管道    public WriteThread(PipedOutputStream pipedOutputStream) {  this.pipedOutputStream = pipedOutputStream;  }  @Override  public void run() {  try {  pipedOutputStream.write("一个美女...".getBytes());  pipedOutputStream.close();  } catch (IOException e) {  e.printStackTrace();  }  }  } |

## 十二. RandomAccessFile

RandomAccessFile是IO包的类，从Object直接继承而来。 只可以对文件进行操作，可以对文件进行读取和写入。  
当模式为r是，当文件不存在时会报异常，当模式为rw时，当文件不存在时，会自己动创建文件，当文件已经存在时 不会对原有文件进行覆盖。  
RandomAccessFile有强大的文件读写功能，其内部是大型 byte[]，可以通过seek(),getFilePointer()等方法操作的指针，方便对数据进行写入与读取。还可以对基本数据类型进行直接的读和写操作。

RandomAccessFile的绝大多数功能，已经被JDK 1.4的nio的“内存映射文件(memory-mapped files)”给取代了，你该考虑一下是不是用“内存映射文件”来代替RandomAccessFile了。

|  |
| --- |
| import java.io.FileNotFoundException;  import java.io.IOException;  import java.io.RandomAccessFile;  class Test {  /\*\*  \* r 代表以只读方式打开指定文件  \* rw 以读写方式打开指定文件  \* rws 读写方式打开，并对内容或元数据都同步写入底层存储设备  \* rwd 读写方式打开，对文件内容的更新同步更新至底层存储设备  \*\*/  public static void main(String[] args) {  // randomWrite();  randomRed("F:/test/test.txt",0);  }  /\*\*  \* 写入数据  \*\*/  public static void randomWrite(){  try {  //实例化RandomAccessFile对象  RandomAccessFile raf = new RandomAccessFile("F:/test/test.txt", "rw");  //打开文件时指针位置在最前，即0  System.out.println(raf.getFilePointer());  //写入数据  raf.write("小河流水哗啦啦！！".getBytes());  raf.close();  } catch (FileNotFoundException e) {  e.printStackTrace();  } catch (IOException e) {  e.printStackTrace();  }  }  /\*\*  \* 获取数据  \* @param path 文件路径  \* @param pointe 指针位置  \* \*\*/  public static void randomRed(String path,int pointe){  try{  RandomAccessFile raf = new RandomAccessFile(path, "r");  //获取RandomAccessFile对象文件指针的位置，初始位置是0  System.out.println("RandomAccessFile文件指针的初始位置:"+raf.getFilePointer());  raf.seek(0);//移动文件指针位置  byte[] bytes = new byte[1024];  //用于保存实际读取的字节数  int len=0;  //循环读取  while((len = raf.read(bytes)) != -1){  //打印读取的内容,并将字节转为字符串输入  System.out.println(new String(bytes,0,len));    }  raf.close();  }catch(Exception e){  e.printStackTrace();  }  }  } |

## 十三. Properties文件操作

Properties（Java.util.Properties），主要用于读取Java的配置文件，各种语言都有自己所支持的配置文件，配置文件中很多变量是经常改变的，这样做也是为了方便用户，让用户能够脱离程序本身去修改相关的变量设置。

它提供了几个主要的方法：

**1、getProperty ( String key)**，用指定的键在此属性列表中搜索属性。也就是通过参数 key ，得到 key 所对应的 value。

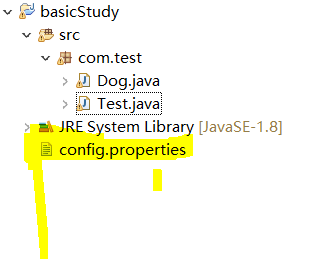
**2、 load ( InputStream inStream)**，从输入流中读取属性列表（键和元素对）。通过对指定的文件（比如说上面的 test.properties 文件）进行装载来获取该文件中的所有键 - 值对。以供 getProperty ( String key) 来搜索。

**3、setProperty ( String key, String value)** ，调用 Hashtable 的方法 put 。他通过调用基类的put方法来设置 键 - 值对。

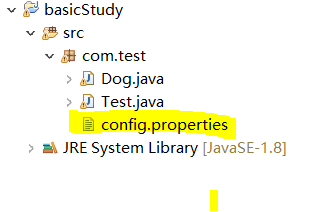
**4、store ( OutputStream out, String comments)**，以适合使用 load 方法加载到 Properties 表中的格式，将此 Properties 表中的属性列表（键和元素对）写入输出流。与 load 方法相反，该方法将键 - 值对写入到指定的文件中去。

5、 **clear ()**，清除所有装载的 键 - 值对。该方法在基类中提供。

|  |
| --- |
| **import** java.io.FileNotFoundException;  **import** java.io.FileOutputStream;  **import** java.io.IOException;  **import** java.io.InputStream;  **import** java.io.OutputStream;  **import** java.util.Properties;  **public** **class** Test {  **private** **static** String *version* = "";  **private** **static** String *username* = "";  **private** **static** String *password* = "";  //静态代码块 只会执行一次  **static** {  // readConfig ();  }  **public** **static** **void** main(String[] args) {  // writeConfig ();  *readConfig* ();  System.***out***.println(*version*);  System.***out***.println(*username*);  System.***out***.println(*password*);  }  /\*\*  \* 写入properties文件  \*/  **private** **static** **void** writeConfig () {  Properties properties = **new** Properties();  properties.put("app.version", "2");  properties.put("db.username", "xiaohong");  properties.put("db.password", "345678");  **try** {  OutputStream out = **new** FileOutputStream("config.properties");  properties.store(out, "update config");  out.close();  } **catch** (FileNotFoundException e) {  e.printStackTrace();  } **catch** (IOException e) {  e.printStackTrace();  }  }  /\*\*  \* 读取properties文件  \*/  **private** **static** **void** readConfig () {  Properties properties = **new** Properties();  **try** {  //通过当前线程的类加载器对象，来加载指定包下的配置文件  InputStream is = Thread.*currentThread*().getContextClassLoader().getResourceAsStream("com/test/config.properties");  // InputStream is = new FileInputStream("config.properties");  properties.load(is);  *version* = properties.getProperty("app.version");  *username* = properties.getProperty("db.username");  *password* = properties.getProperty("db.password");  is.close();  } **catch** (FileNotFoundException e) {  e.printStackTrace();  } **catch** (IOException e) {  e.printStackTrace();  }  }  } |



|  |
| --- |
| InputStream is = new FileInputStream("config.properties"); |



|  |
| --- |
| InputStream is = Thread.currentThread().getContextClassLoader().getResourceAsStream("com/test/config.properties"); |

## 十四.文件压缩与解压缩

java中实现zip的压缩与解压缩

**ZipOutputStream**

实现文件的压缩

ZipOutputStream(OutputStream out)

创建新的 ZIP 输出流。

void putNextEntry(ZipEntry e)

开始写入新的 ZIP 文件条目并将流定位到条目数据的开始处。

ZipEntry(String name) //test/mm.jpg /test/a.txt

使用指定名称创建新的 ZIP 条目。

**ZipInputStream**

实现文件的解压

ZipInputStream(InputStream in)

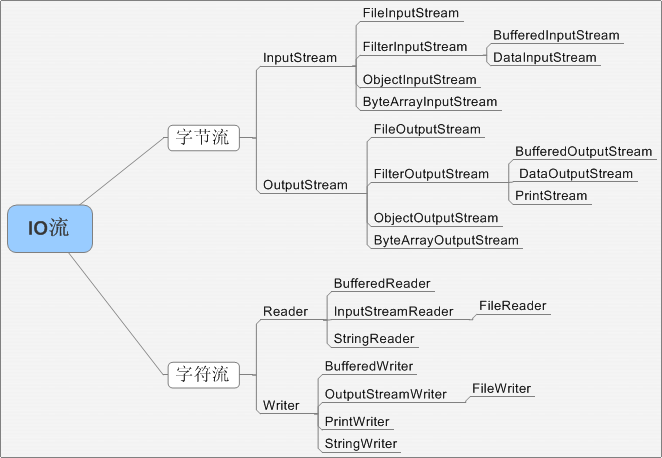
创建新的 ZIP 输入流。

ZipEntry getNextEntry()

读取下一个 ZIP 文件条目并将流定位到该条目数据的开始处。

|  |
| --- |
| **import** java.io.BufferedInputStream;  **import** java.io.BufferedOutputStream;  **import** java.io.File;  **import** java.io.FileInputStream;  **import** java.io.FileNotFoundException;  **import** java.io.FileOutputStream;  **import** java.io.IOException;  **import** java.io.InputStream;  **import** java.io.OutputStream;  **import** java.util.zip.ZipEntry;  **import** java.util.zip.ZipInputStream;  **import** java.util.zip.ZipOutputStream;  **public** **class** Test {  **public** **static** **void** main(String[] args) {  // compression ("F:/test/test.zip", new File("F:/test"));  *decompression* ("F:/test/test.zip", "F:/test/aaa/");  }  /\*\*  \* 压缩文件  \* zipName :压缩文件名称  \* targetFile：要压缩的文件夹  \*/  **public** **static** **void** compression (String zipName, File targetFile) {  System.***out***.println("正在压缩文件....");  **try** {  //要生成的压缩文件  ZipOutputStream zOut = **new** ZipOutputStream(**new** FileOutputStream(zipName));  BufferedOutputStream bos = **new** BufferedOutputStream(zOut);  *zip*(zOut, targetFile, targetFile.getName(), bos);  bos.close();  System.***out***.println("压缩完成!");  } **catch** (FileNotFoundException e) {  e.printStackTrace();  } **catch** (IOException e) {  e.printStackTrace();  }  }  //zip  **private** **static** **void** zip(ZipOutputStream zOut, File targetFile, String name, BufferedOutputStream bos) **throws** IOException {  //如果是个目录  **if** (targetFile.isDirectory()) {  File[] listFiles = targetFile.listFiles();  **if** (listFiles.length == 0) { //空文件夹  zOut.putNextEntry(**new** ZipEntry(name + "/")); //处理空目录  }  **for** (File file : listFiles) {  //递归处理  *zip*(zOut, file, name + "/" + file.getName(), bos);  }  } **else** {  zOut.putNextEntry(**new** ZipEntry(name));  InputStream in = **new** FileInputStream(targetFile);  BufferedInputStream bis = **new** BufferedInputStream(in);  **byte**[] bytes = **new** **byte**[1024];  **int** len = -1;  **while** ((len = bis.read(bytes)) != -1) {  bos.write(bytes, 0, len);  }  bis.close();  }  }  /\*\*  \* 解压压缩文件  \* targetFileName:目标文件  \* parent：上级目录  \*/  **public** **static** **void** decompression (String targetFileName, String parent) {  System.***out***.println("正在解压文件....");  **try** {  ZipInputStream zIn = **new** ZipInputStream(**new** FileInputStream(targetFileName));  ZipEntry entry = **null**;  File file = **null**;  **while** ((entry = zIn.getNextEntry()) != **null** && !entry.isDirectory()) {  file = **new** File(parent, entry.getName());  **if** (!file.exists()) {  **new** File(file.getParent()).mkdirs();//创建此文件的上级目录  }  OutputStream out = **new** FileOutputStream(file);  BufferedOutputStream bos = **new** BufferedOutputStream(out);  **byte**[] bytes = **new** **byte**[1024];  **int** len = -1;  **while** ((len = zIn.read(bytes)) != -1) {  bos.write(bytes, 0, len);  }  bos.close();  }  System.***out***.println("解压完成!");  } **catch** (FileNotFoundException e) {  e.printStackTrace();  } **catch** (IOException e) {  e.printStackTrace();  }  }  } |

## 十五. IO常用类体系

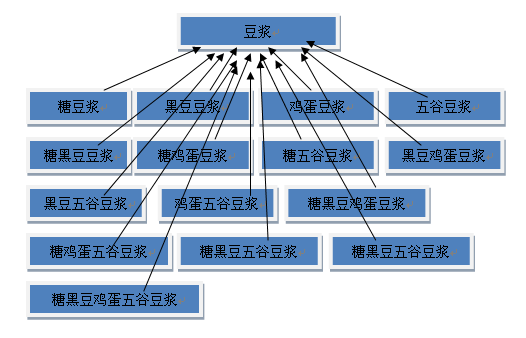


## 十六.装饰者模式

主料：豆浆

配料：糖 黑豆 五谷 鸡蛋…





**意图**：

动态地给一个对象添加一些额外的职责。就增加功能来说，Decorator模式相比生成子类更为灵活。该模式以对客户端透明的方式扩展对象的功能。

**适用环境**

在不影响其他对象的情况下，以动态、透明的方式给单个对象添加职责。

处理那些可以撤消的职责。

当不能采用生成子类的方法进行扩充时。一种情况是，可能有大量独立的扩展，为支持每一种组合将产生大量的子类，使得子类数目呈爆炸性增长。另一种情况可能是因为类定义被隐藏，或类定义不能用于生成子类。

**类图：**

**Component（被装饰对象基类）**

定义对象的接口，可以给这些对象动态增加职责；

**ConcreteComponent（具体被装饰对象）**

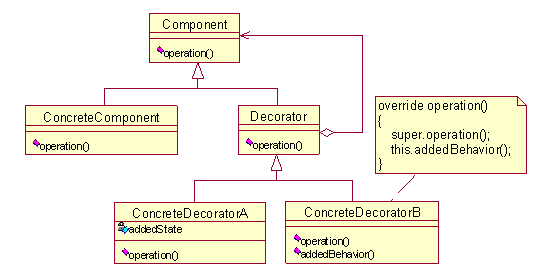
定义具体的对象，Decorator可以给它增加额外的职责；

**Decorator（装饰者抽象类）**

维护指向Component实例的引用，定义与Component一致的接口；

**ConcreteDecorator（具体装饰者）**

具体的装饰对象，给内部持有的具体被装饰对象增加具体的职责；



**涉及角色**

抽象组件：定义一个抽象接口，来规范准备附加功能的类。

具体组件：将要被附加功能的类，实现抽象构件角色接口。

抽象装饰者：持有对具体构件角色的引用并定义与抽象构件角色一致的接口。

具体装饰：实现抽象装饰者角色，负责为具体构件添加额外功能。

**代码实现：**

Drink.java 被装饰者对象的接口

SoyaBeanMilk.java 具体的被装饰者对象

EggDecorator.java 具体装饰者对象

SugarDecorator.java 具体装饰者对象

BlackBeanDecorator.java 具体装饰者对象

Decorator.java 装饰者基类

Test.java 测试

**装饰者模式小结：**

OO原则：动态地将责任附加到对象上。想要扩展功能，装饰者提供有别于继承的另一种选择。

**要点：**

1、继承属于扩展形式之一，但不见得是达到弹性设计的最佳方案。

2、在我们的设计中，应该允许行为可以被扩展，而不须修改现有的代码。

3、组合和委托可用于在运行时动态地加上新的行为。

4、除了继承，装饰者模式也可以让我们扩展行为。

5、装饰者模式意味着一群装饰者类，这些类用来包装具体组件。

6、装饰者类反映出被装饰的组件类型（实际上，他们具有相同的类型，

都经过接口或继承实现）。

7、装饰者可以在被装饰者的行为前面与/或后面加上自己的行为，

甚至将被装饰者的行为整个取代掉，而达到特定的目的。

8、你可以有无数个装饰者包装一个组件。

9、装饰者一般对组建的客户是透明的，

除非客户程序依赖于组件的具体类型。

|  |
| --- |
| **public** **class** Test {  **public** **static** **void** main(String[] args) {  Drink drink = **new** SoyaBeanMilk();  SugarDecorator sugar = **new** SugarDecorator(drink);  BlackBeanDecorator blackBean = **new** BlackBeanDecorator(sugar);  EggDecorator egg = **new** EggDecorator(blackBean);  System.***out***.println("你下单的豆浆是：" + egg.description());  System.***out***.println("一共花了" + egg.cost() + "元");  }  }  //被装饰者的接口  **interface** Drink {  **float** cost(); //计算价格  String description(); //描述  }  /\*\*  \* 具体被装饰的类  \* 豆浆  \*/  **class** SoyaBeanMilk **implements** Drink {  @Override  **public** **float** cost() {  **return** 10f;  }  @Override  **public** String description() {  **return** "纯豆浆";  }  }  /\*\*  \* 装饰者的基类  \*/  **abstract** **class** Decorator **implements** Drink {  **private** Drink drink;  **public** Decorator(Drink drink) {  **this**.drink = drink;  }  @Override  **public** **float** cost() {  **return** drink.cost();  }  @Override  **public** String description() {  **return** drink.description();  }  }  /\*\*  \* 具体的装饰类  \* 糖  \*/  **class** SugarDecorator **extends** Decorator {  **public** SugarDecorator(Drink drink) {  **super**(drink);  }  @Override  **public** **float** cost() {  **return** **super**.cost()+1.0f;  }  @Override  **public** String description() {  **return** **super**.description()+"+糖";  }  }  /\*\*  \* 具体的装饰类  \* 黑豆  \*/  **class** BlackBeanDecorator **extends** Decorator {  **public** BlackBeanDecorator(Drink drink) {  **super**(drink);  }  @Override  **public** **float** cost() {  **return** **super**.cost()+2.0f;  }  @Override  **public** String description() {  **return** **super**.description()+"+黑豆";  }  }  /\*\*  \* 具体的装饰类  \* 鸡蛋  \*/  **class** EggDecorator **extends** Decorator {  **public** EggDecorator(Drink drink) {  **super**(drink);  }  @Override  **public** **float** cost() {  **return** **super**.cost()+5.0f;  }  @Override  **public** String description() {  **return** **super**.description()+"+鸡蛋";  }  } |

## 十七.常见字符编码

在程序中如果没有处理好字符的编码，就有可能出现乱码问题。下面我们一起为大家介绍在开发中常见的编码有

哪些。

在计算机世界里，任何的文字都是以指定的编码方式存在的。

常见编码有：ISO8859-1、GBK/GB2312、unicode、UTF。

**iso8859-1：**

编码属于单字节编码，最多只能表示0——255的字符范围，主要在英文上应用。

**GBK/GB2312：**

中文的国际编码，专门用来表示汉字，是双字节编码

**unicode：**

java中就是使用此编码方式，也是最标准的一种编码，是使用16进制表示的编码。但此编码不兼容iso8859-1编码。

**UTF：**

由于unicode不支持iso8859-1编码，而且容易占用更多的空间，而且对于英文母也需要使用两个字节编码，这样使

用unicode不便于传输和储存，因此产生了utf编码，utf编码兼容了iso8859-1编码，也可以用来表示所有语言字符，

不过utf是不定长编码，每个字符的长度从1－6个字节不等，一般在中文网页中使用此编码，因为这样可以节省空

间。

**造成乱码的根本原因：**

1、程序使用的编码与本机的编码不统一

2、在网络中，客户端与服务端编码不统一（WEB开发中出现的乱码情况）

|  |
| --- |
| **import** java.io.UnsupportedEncodingException;  **public** **static** **void** main(String[] args) {  String info = "小河流水哗啦啦！！";  **try** {  //编码转化  String newInfo = **new** String(info.getBytes("GBK"), "iso8859-1");  System.***out***.println(newInfo); // --?????÷????????????  String newInfo2 = **new** String(newInfo.getBytes("iso8859-1"), "GBK");  System.***out***.println(newInfo2); // --小河流水哗啦啦！！  } **catch** (UnsupportedEncodingException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  } |

## 十八.New IO

**1、为什么要使用 NIO?**

NIO是JDK1.4加入的新包，NIO 的创建目的是为了让 Java 程序员可以实现高速 I/O 而无需编写自定义的本机代码。NIO 将最耗时的 I/O 操作（即填充和提取缓冲区）转移回操作系统，因而可以极大地提高速度。

**流与块的比较**

原来的 I/O 库(在 java.io.\*中) 与 NIO 最重要的区别是数据打包和传输的方式，原来的 I/O 以流的方式处理数据，而 NIO 以块的方式处理数据。

面向流 的 I/O 系统一次一个字节地处理数据。一个输入流产生一个字节的数据，一个输出流消费一个字节的数据。不利的一面是，面向流的 I/O 通常相当慢。

一个 面向块 的 I/O 系统以块的形式处理数据。每一个操作都在一步中产生或者消费一个数据块。按块处理数据比按(流式的)字节处理数据要快得多。但是面向块的 I/O 缺少一些面向流的 I/O 所具有的优雅性和简单性。

**缓冲区**

在 NIO 库中，所有数据都是用缓冲区处理的。在读取数据时，它是直接读到缓冲区中的。在写入数据时，它是写入到缓冲区中的。任何时候访问 NIO 中的数据，您都是将它放到缓冲区中。

缓冲区实质上是一个数组。通常它是一个字节数组，但是也可以使用其他种类的数组。但是一个缓冲区不 仅仅 是一个数组。缓冲区提供了对数据的结构化访问，而且还可以跟踪系统的读/写进程。

**缓冲区类型**

最常用的缓冲区类型是 ByteBuffer。一个 ByteBuffer 可以在其底层字节数组上进行 get/set 操作（即字节的获取和设置）。ByteBuffer 不是 NIO 中唯一的缓冲区类型。事实上，对于每一种基本 Java 类型都有一种缓冲区类型：

ByteBuffer

CharBuffer

ShortBuffer

IntBuffer

LongBuffer

FloatBuffer

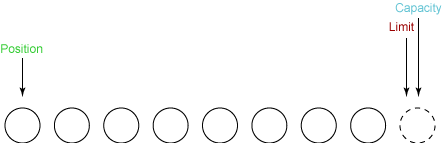
DoubleBuffer

**缓冲区内部细节：**

**状态变量**

可以用三个值指定缓冲区在任意时刻的状态：

**position limit capacity**



|  |
| --- |
| **import java.nio.ByteBuffer;**  **public class Test {**  **public static void main(String[] args) {**  **//创建一个字节缓冲区，申请内存空间为8个字节**  **ByteBuffer buf = ByteBuffer.allocate(8);**  **System.out.println("position=" + buf.position()); //下标位置 结果：0**  **System.out.println("limit=" + buf.limit()); //字节数 结果：8**  **System.out.println("capacity=" + buf.capacity()); //总字节数 结果：8**  **//向缓冲区中写入数据**  **buf.put((byte)10);**  **buf.put((byte)20);**  **buf.put((byte)30);**  **buf.put((byte)40);**  **System.out.println("position=" + buf.position()); //下标位置 结果：4**  **System.out.println("limit=" + buf.limit()); //字节数 结果：8**  **System.out.println("capacity=" + buf.capacity()); //总字节数 结果：8**  **//缓冲区反转**  **buf.flip();**  **System.out.println("position=" + buf.position()); //下标位置 结果：0**  **System.out.println("limit=" + buf.limit()); //字节数 结果：4**  **System.out.println("capacity=" + buf.capacity()); //总字节数 结果：8**  **//判断在当前位置和限制之间是否有元素**  **if (buf.hasRemaining()) {**  **//返回当前位置与限制位置之间的元素数。**  **for (int i = 0; i < buf.remaining(); i++) {**  **byte b = buf.get(i);**  **System.out.println(b);**  **}**  **}**  **}**  **}** |

**通道：** **Channel**

Channel 是一个对象，可以通过它读取和写入数据。拿 NIO 与原来的 I/O 做个比较，通道就像是流。

正如前面提到的，所有数据都通过 Buffer 对象来处理。您永远不会将字节直接写入通道中，相反，您是将数据写入包含一个或者多个字节的缓冲区。同样，您不会直接从通道中读取字节，而是将数据从通道读入缓冲区，再从缓冲区获取这个字节。

**使用通道读写文件示例**

|  |
| --- |
| import java.io.FileInputStream;  import java.io.FileNotFoundException;  import java.io.FileOutputStream;  import java.io.IOException;  import java.io.RandomAccessFile;  import java.nio.ByteBuffer;  import java.nio.MappedByteBuffer;  import java.nio.channels.FileChannel;  import java.nio.channels.FileChannel.MapMode;  class Test {  public static void main(String[] args) {  randomAccessFileCopy();  }  //通过文件通道实现文件的复制  private static void randomAccessFileCopy() {  try {  FileChannel in = new RandomAccessFile("F:/test/test.txt", "r").getChannel();  FileChannel out = new RandomAccessFile("F:/test/切割/test.txt", "rw").getChannel();  long size = in.size();//输入流的字节大小  //输入流的缓冲区  MappedByteBuffer inBuf = in.map(MapMode.READ\_ONLY, 0, size);  //输出流的缓冲区  MappedByteBuffer outBuf = out.map(MapMode.READ\_WRITE, 0, size);    for (int i = 0; i < size; i++) {  outBuf.put(inBuf.get(i));  }  out.close();  in.close();  } catch (FileNotFoundException e) {  e.printStackTrace();  } catch (IOException e) {  e.printStackTrace();  }  }  /\*\*  \* 通过文件通道实现文件的复制  \*/  private static void copyFile () {  //创建一个输入文件的通道  try {  FileChannel fcIn = new FileInputStream("F:/test/test.txt").getChannel();  FileChannel fcOut = new FileOutputStream("F:/test/切割/test.txt").getChannel();  ByteBuffer buf = ByteBuffer.allocate(1024);  while (fcIn.read(buf) != -1) {  buf.flip();  fcOut.write(buf);  buf.clear();  }  fcOut.close();  fcOut.close();  } catch (FileNotFoundException e) {  e.printStackTrace();  } catch (IOException e) {  e.printStackTrace();  }  }  } |

JDK1.7引入了新的IO操作类， java.nio.file包下，Java NIO Path接口和Files类

**Path接口**

1、Path表示的是一个目录名序列，其后还可以跟着一个文件名，路径中第一个部件是根部件时就是绝对路径，例如 / 或 C:\ ，而允许访问的根部件取决于文件系统；

2、以根部件开始的路径是绝对路径，否则就是相对路径；

3、静态的Paths.get方法接受一个或多个字符串，字符串之间自动使用默认文件系统的路径分隔符连接起来（Unix是 /，Windows是 \ ），这就解决了跨平台的问题，接着解析连接起来的结果，如果不是合法路径就抛出InvalidPathException异常，否则就返回一个Path对象；

|  |
| --- |
| import java.io.File;  import java.nio.file.FileSystems;  import java.nio.file.Path;  import java.nio.file.Paths;  public class Test {  public static void main(String[] args) {  Path p1 = Paths.get("F:/test","test.txt");  System.out.println(p1);  Path p2 = Paths.get("F:/test/test.txt");  System.out.println(p2);  File file = new File("F:/test/test.txt");  Path p3 = file.toPath();  System.out.println(p3);  Path p4 = FileSystems.getDefault().getPath("F:/test","test.txt");  System.out.println(p4);  }  } |

**Files工具类**

**1、读写文件**

static path write(Path path, byte[] bytes, OpenOption... options) 写入文件

static byte[] readAllBytes(Path path) 读取文件中的所有字节。

**2、复制、剪切、删除**

static path copy(Path source, Path target, CopyOption... options)

static path move(Path source, Path target, CopyOption... options)

static void delete(Path path) //如果path不存在文件将抛出异常，此时调用下面的比较好

static boolean deleteIfExists(Path path)

**3、创建文件和目录**

//创建新目录，除了最后一个部件，其他必须是已存在的

Files.createDirectory(path);

//创建路径中的中间目录，能创建不存在的中间部件

Files.createDirectories(path);

//创建一个空文件，检查文件存在，如果已存在则抛出异常而检查文件存在是原子性的，

//因此在此过程中无法执行文件创建操作

Files.createFile(path);

//添加前/后缀创建临时文件或临时目录

Path newPath = Files.createTempFile(dir, prefix, suffix);

Path newPath = Files.createTempDirectory(dir, prefix);

|  |
| --- |
| import java.io.IOException;  import java.io.UnsupportedEncodingException;  import java.nio.file.Files;  import java.nio.file.Path;  import java.nio.file.Paths;  import java.nio.file.StandardCopyOption;  import java.nio.file.StandardOpenOption;  public class Test {  public static void main(String[] args) {  Path path = Paths.get("F:/test/test.txt");  try {  //向文件中追加数据  Files.write(path, "村花到我家".getBytes("gb2312"), StandardOpenOption.APPEND);  //从文件中读取数据  byte[] bates = Files.readAllBytes(path);  System.out.println(new String(bates));  //复制文件  Files.copy(path, Paths.get("F:/test/切割/test.txt"), StandardCopyOption.REPLACE\_EXISTING);  //移动文件  Files.move(path, Paths.get("F:/test/切割/test.txt"), StandardCopyOption.REPLACE\_EXISTING);  //删除文件  Files.delete(path);  //创建新目录，除了最后一个部件，其他必须是已存在的  Files.createDirectory(Paths.get("F:/test/test"));  //创建多级不存在目录  Files.createDirectories(Paths.get("F:/test/test/test/test"));  //创建文件  Files.createFile(Paths.get("F:/test/aaa.txt"));  //添加前/后缀创建临时文件或临时目录  Files.createTempFile(Paths.get("F:/test"), "aaa", "txt");  Files.createTempDirectory(Paths.get("F:/test"), "test");  } catch (UnsupportedEncodingException e) {  e.printStackTrace();  } catch (IOException e) {  e.printStackTrace();  }  }  } |

# 第八章 集合

## 一．集合框架概述

**1、集合框架的作用**

在实际开发中，我们经常会对一组相同类型的数据进行统一管理操作。到目前为止，我们可以使用数组结构，链表结构，二叉树结构来实现。

数组的最大问题在于数组中的元素个数是固定的，要实现动态数组，必竟还是比较麻烦，自己实现链表或二叉树结构来管理对象更是不方便。

在JDK1.2版本后，JAVA完整的提供了类集合的概念，封装了一组强大的、非常方便的集合框架API，让我们在开发中大大的提高了效率。

集合中分为三大接口：

Collection、Map、Iterator

集合框架的接口和类在java.util包中

2、集合框架结构图

Comparable

Comparator

Utilities

Collections

Arrays

Iterator

Collection

Map

ListIterator

List

Set

HashMap

TreeMap

LinkedHashMap

HashSet

TreeSet

LinkedHashSet

ArrayList

LinkedList

Vector

3、Collection接口

*Collection 层次结构* 中的根接口。Collection 表示一组对象，这些对象也称为 collection 的*元素*。一些 collection 允许有重复的元素，而另一些则不允许。一些 collection 是有序的，而另一些则是无序的。JDK 不提供此接口的任何*直接* 实现：它提供更具体的子接口（如 Set 和 List）实现。此接口通常用来传递 collection，并在需要最大普遍性的地方操作这些 collection。

接口的定义：

public interface **Collection<E>**

extends Iterable<E>

## 二．集合框架List接口

**1、 List接口**

public interface **List<E>** extends Collection<E>

有序的 collection（也称为*序列*）。此接口的用户可以对列表中每个元素的插入位置进行精确地控制。用户可以根据元素的整数索引（在列表中的位置）访问元素，并搜索列表中的元素。

**2、ArrayList**

public class **ArrayList<E>** extends AbstractList<E>

implements List<E>, RandomAccess, Cloneable, Serializable

List 接口的大小可变数组的实现。实现了所有可选列表操作，并允许包括 null 在内的所有元素。除了实现 List 接口外，此类还提供一些方法来操作内部用来存储列表的数组的大小。

**3、Vector**

public class **Vector<E>** extends AbstractList<E>

implements List<E>, RandomAccess, Cloneable, Serializable

Vector 类可以实现可增长的对象数组。与数组一样，它包含可以使用整数索引进行访问的组件。但是，Vector 的大小可以根据需要增大或缩小，以适应创建 Vector 后进行添加或移除项的操作。

**4、LinkedList**

public class **LinkedList<E>** extends AbstractSequentialList<E>

implements List<E>, Deque<E>, Cloneable, Serializable

List 接口的链接列表实现。实现所有可选的列表操作，并且允许所有元素（包括 null）。除了实现 List 接口外，LinkedList 类还为在列表的开头及结尾 get、remove 和 insert 元素提供了统一的命名方法。

|  |
| --- |
| import java.util.List;  import java.util.ArrayList;  import java.util.Vector;  import java.util.LinkedList;  /\*\*  Collection接口：由于存储单个对象的集合  List接口：  1、有序的，可以重复  2、允许多个null元素  3、具体实现常用的有：ArrayList，Vector，LinkedList  在实际开发中，我们如何选择list具体实现？  1.安全性问题  2.是否频繁插入，删除操作（LinkedList）  3.是否是存储后遍历  \*/  class Test {  /\*\*  ArrayList  1.实现原理：采用动态数组实现，默认构造方法创建了一个空数组  2.第一次添加元素，扩展容量为10，之后的扩充算法：原来数组大小+原来数组的一半  3.不适合进行删除和插入操作  4.为了防止数组动态扩充次数过多，建议创建ArrayList时，给定初始容量  5.多线程中使用不安全，适合在单线程访问使用，效率较高  JDK1.2开始  \*/  private static void arrayList () {  //使用集合来存储多个不同类型的元素（对象），那么在处理时会比较麻烦，在实际开发中，  //不建议这样使用，我们应该在一个集合中存储相同的类型对象。  //List list = new ArrayList();  List<String> list= new ArrayList<String>();  list.add("张老师");  list.add("李老师");  list.add("王老师");  int size = list.size();  for (int i = 0; i < size; i++) {  System.out.println(list.get(i));  }  System.out.println(list.contains("张老师")); //list中是否包含“张老师”  // list.remove("王老师"); //删除集合中的元素  list.set(2, "杭老师"); //替换集合中下标为2的数据  String[] array = list.toArray(new String[]{}); //集合转为数组  for(String arr : array) {  System.out.println(arr);  }  }  /\*\*  Vector  1.实现原理，采用动态数组实现，默认构造方法创建了一个大小为10的对象数组  2.扩充算法：当增量大于0时，扩充为原来大小+增量  3.不适合进行删除和插入操作  4.为了防止数组动态扩充次数过多，建议创建vector时，给定初始容量  5.多线程中使用安全，适合在多线程访问使用，在单线程中使用效率较低  \*/  private static void vector () {  Vector<String> v= new Vector<String>();  v.add("张老师");  v.add("李老师");  v.add("王老师");  for(int i = 0; i < v.size(); i++) {  System.out.println(v.get(i));  }  }  /\*\*  LinkedList  1.实现原理,采用双向链表结构实现  2.适合插入删除操作  \*/  private static void linkedList () {  List<String> list = new LinkedList<String>();  list.add("张老师");  list.add("李老师");  list.add("王老师");  for(int i = 0; i < list.size(); i++) {  System.out.println(list.get(i));  }  }  public static void main(String[] args) {  // arrayList ();  // vector ();  linkedList ();  }  } |

## 三．集合框架Set接口

**1、Set接口**

public interface **Set<E>** extends Collection<E>

一个不包含重复元素的 collection。更确切地讲，set 不包含满足 e1.equals(e2) 的元素对 e1 和 e2，并且最多包含一个 null 元素。正如其名称所暗示的，此接口模仿了数学上的 *set* 抽象。

**2、HashSet**

public class **HashSet<E>** extends AbstractSet<E> implements Set<E>, Cloneable, Serializable

类实现 Set 接口，由哈希表（实际上是一个 HashMap 实例）支持。它不保证 set 的迭代顺序；特别是它不保证该顺序恒久不变。此类允许使用 null 元素。

hashCode深入分析

**hashcode() 方法，在object类中定义如下：**

**public native int hashCode();**

hashCode是本地方法，它的实现是根据本地机器相关，当然我们可以在自己写的类中覆盖hashcode()方法，比如String、Integer、Double。。。。等等这些类都是覆盖了hashcode()方法的。

**在java的集合中，判断两个对象是否相等的规则是：**

（1）判断两个对象的hashCode是否相等

如果不相等，认为两个对象也不相等，结束

如果相等，转入2

（2）判断两个对象用equals运算是否相等

如果不相等，认为两个对象也不相等

如果相等，认为两个对象相等

（equals()是判断两个对象是否相等的关键）

**3、TreeSet**

public class **TreeSet<E>** extends AbstractSet<E> implements NavigableSet<E>, Cloneable, Serializable

基于 TreeMap 的 NavigableSet 实现。使用元素的自然顺序对元素进行排序，或者根据创建 set 时提供的 Comparator 进行排序，具体取决于使用的构造方法。

**4、LinkedHashSet**

public class **LinkedHashSet<E>** extends HashSet<E> implements Set<E>, Cloneable, Serializable

具有可预知迭代顺序的 Set 接口的哈希表和链接列表实现。此实现与 HashSet 的不同之外在于，后者维护着一个运行于所有条目的双重链接列表。此链接列表定义了迭代顺序，即按照将元素插入到 set 中的顺序（插入顺序）进行迭代。注意，插入顺序不 受在 set 中重新插入的 元素的影响。（如果在 s.contains(e) 返回 true 后立即调用 s.add(e)，则元素 e 会被重新插入到 set s 中。）

|  |
| --- |
| import java.util.Set;  import java.util.HashSet;  import java.util.Vector;  import java.util.LinkedList;  import java.util.TreeSet;  import java.util.LinkedHashSet;  import java.util.Comparator;  /\*\*  Collection接口：由于存储单个对象的集合  set接口：  1、无序的，不允许重复  2、具体实现常用的有：HashSet，TreeSet，LinkedHashList  在使用中选择  (1)如果要排序，选择treeSet  (2)如果不要排序，也不用保证顺序,选择HashSet  (3)如果不要排序，要保证顺序，选择LinkedHashSet  \*/  class Test {  /\*\*  HashSet  1.实现原理：基于哈希表（HashMap）实现  2.不允许重复，可以有一个null元素  3.不保证顺序恒久不变  4.添加元素时把元素作为HashMap的key存储，HashMap的value使用一个固定的object对象  5.排除重复元素是通过equals来检查对象是否相同  6.判断两个对象是否相同，先判断两个对象的hashcode是否相同  （如果两个对象的hashcode相同，不一定是同一个对象，如果不同，那一定不是同一个对象）  如果不同，则两个对象不是同一个对象，如果相同，还要进行equals判断，equals相同则是同一个对象  不同则不是同一个对象。  7.自定义对象要认为属性值都相同时为同一个对象，有这种需求时，那么我们要重写对象所在类的hashcode和equals方法  小结：  （1）哈希表的存储结构：数组+链表，数组里面的每个元素以链表的形式存储  （2）如何把对象存储到哈希表中，先计算对象的hashcode值，再对数组的长度求余数，来决定对象要存储在数组的哪个位置  （3）解决hashset中的重复值使用的方式是，参考6    \*/  private static void hashSet () {  Set<String> set= new HashSet<String>();  set.add("张老师");  set.add("李老师");  set.add("王老师");  String[] array = set.toArray(new String[]{}); //集合转为数组  for(String arr : array) {  System.out.println(arr);  }  Dog d1 = new Dog("wangwang", 3);  Dog d2 = new Dog("huahua", 4);  Dog d3 = new Dog("bandian", 5);  Dog d4 = new Dog("wangwang", 3);    Set<Dog> dogSet = new HashSet<Dog>();  dogSet.add(d1);  dogSet.add(d2);  dogSet.add(d3);  dogSet.add(d4);  for (Dog dog : dogSet) {  System.out.println(dog+"hashcode="+dog.hashCode());  }  }  /\*\*  TreeSet  1.有序的，基于TreeMap(二叉树数据结构)，对象需要比较大小，通过对象比较器来实现，  对象比较器还可以用来去除重复元素，如果自定义的数据类，没有实现比较器接口，将无法添加到TreeSet集体中。  \*/  private static void treeSet () {  Dog d1 = new Dog("wangwang", 3);  Dog d2 = new Dog("huahua", 4);  Dog d3 = new Dog("bandian", 5);  Dog d4 = new Dog("wangwang", 3);  TreeSet<Dog> set = new TreeSet<Dog>(new Dog());  set.add(d1);  set.add(d2);  set.add(d3);  set.add(d4);  for (Dog dog : set) {  System.out.println(dog);  }  }  /\*\*  LinkedHashSet  1.哈希表和链接列表实现  2.维护者一个运行于所有条目的双重链接列表，此链接列表定义了迭代顺序，  即按照将元素插入都set中的顺序（插入顺序）进行迭代。  \*/  private static void linkedHashSet () {  Dog d1 = new Dog("wangwang", 3);  Dog d2 = new Dog("huahua", 4);  Dog d3 = new Dog("bandian", 5);  Dog d4 = new Dog("wangwang", 3);  Set<Dog> set = new LinkedHashSet<Dog>();  set.add(d1);  set.add(d2);  set.add(d3);  set.add(d4);  for (Dog dog : set) {  System.out.println(dog);  }  }  public static void main(String[] args) {  // hashSet ();  // treeSet ();  linkedHashSet ();  }  }  class Dog implements Comparator<Dog> {  private int id;  private String name;  private int age;  public Dog() {  super();  }  public Dog(String name, int age) {  super();  this.name = name;  this.age = age;  }  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;  }  public int getAge() {  return age;  }  public void setAge(int age) {  this.age = age;  }  @Override  public String toString() {  return "Dog [id=" + id + ", name=" + name + ", age=" + age + "]";  }  @Override  public int hashCode() {  final int prime = 31;  int result = 1;  result = prime \* result + age;  result = prime \* result + id;  result = prime \* result + ((name == null) ? 0 : name.hashCode());  return result;  }  @Override  public boolean equals(Object obj) {  if (this == obj)  return true;  if (obj == null)  return false;  if (getClass() != obj.getClass())  return false;  Dog other = (Dog) obj;  if (age != other.age)  return false;  if (id != other.id)  return false;  if (name == null) {  if (other.name != null)  return false;  } else if (!name.equals(other.name))  return false;  return true;  }  @Override  public int compare(Dog o1, Dog o2) {  return o1.age - o2.age;  }  } |

## 四．集合框架Iterator接口

### 1、集合输出

前面我们已经学习了集合的基本操作，很多情况下，我们需要把集合的内容进行输出，也就是遍历集合。

遍历集合的方式有以下几种：

**1、 Iterator**

2、 ListIterator

3、 Enumeration

**4、 foreach**

其中Iterator的使用率最高，在JDK1.5后新增的foreach也被大量使用。

### 2、Iterator

public interface **Iterator<E>**

对 collection 进行迭代的迭代器。迭代器取代了 Java Collections Framework 中的 Enumeration。

|  |  |
| --- | --- |
| boolean hasNext() | 如果仍有元素可以迭代，则返回 true。 |
| E next() | 返回迭代的下一个元素。 |
| void remove() | 从迭代器指向的 collection 中移除迭代器返回的最后一个元素。 |

### 3、ListIterator

public interface **ListIterator<E>**

extends Iterator<E>

系列表迭代器，允许程序员按任一方向遍历列表、迭代期间修改列表，并获得迭代器在列表中的当前位置。

|  |  |
| --- | --- |
| void add(E e) | 增加元素 |
| boolean hasPrevious() | 判断是否有前一个元素 |
| E previous() | 取出前一个元素 |
| void set(E e) | 修改元素的内容 |
| int previousIndex() | 前一个索引位置 |
| int nextIndex() | 下一个索引位置 |

### 4、Enumeration

public interface **Enumeration<E>**

实现 Enumeration 接口的对象，它生成一系列元素，一次生成一个。连续调用 nextElement 方法将返回一系列的连续元素。

**注：**此接口的功能与 Iterator 接口的功能是重复的。此外，Iterator 接口添加了一个可选的移除操作，并使用较短的方法名。新的实现应该优先考虑使用 Iterator 接口而不是 Enumeration 接口。

|  |  |
| --- | --- |
| boolean hasMoreElements() | 判断是否有下一个元素 |
| E nextElement() | 取出当前元素 |

### 5、foreach

在前面的知识讲解中，我们使用foreach来输出数组的内容，那么也可以输出集合中的内容。在使用foreach输出的时候一定要注意的是，创建集合时要指定操作泛型的类型。

|  |
| --- |
| import java.util.Collection;  import java.util.List;  import java.util.ArrayList;  import java.util.Iterator;  import java.util.Enumeration;  import java.util.Vector;  class Test {  //foreach (1.5后)  private static void foreach (Collection<String> list) {  for (String str : list) {  System.out.println(str);  }  }  //Iterator (1.5前)  private static void iterator (Collection<String> list) {  Iterator<String> iterator= list.iterator();  while(iterator.hasNext()) {  System.out.println(iterator.next());  }  }  //Enumeration (1.1)  private static void enumeration () {  Vector<String> v = new Vector<String>();  v.add("张老师");  v.add("黄老师");  v.add("李老师");  Enumeration<String> elements = v.elements();  while(elements.hasMoreElements()) {  System.out.println(elements.nextElement());  }  }  public static void main(String[] args) {  List<String> list = new ArrayList<String>();  list.add("张老师");  list.add("黄老师");  list.add("李老师");  // foreach (list);  // iterator (list);  enumeration ();  }  } |

## 五．JDK1.8新特性

### 1. forEach的使用

List<Integer> numbers = new ArrayList<>();

//no.1

numbers.forEach((Integer integer) -> {System.out.println(integer);});

//no.2

numbers.forEach(integer -> {System.out.println(integer);});

//no.3

numbers.forEach(integer -> System.out.println(integer));

//no.4

numbers.forEach(System.out::println);

//no.5

numbers.forEach(new MyConsumer());

|  |
| --- |
| import java.util.List;  import java.util.ArrayList;  class Test {  public static void main(String[] args) {  List<String> list = new ArrayList<String>();  list.add("张老师");  list.add("黄老师");  list.add("李老师");  //写法1  list.forEach((String s) -> {System.out.println(s);});  //写法2  list.forEach(s -> {System.out.println(s);});  //写法3  list.forEach(s -> System.out.println(s));  //写法4  list.forEach(System.out::println);  }  } |

### 2. Consumer、Function、Supplier、Predicate接口的使用

Consumer<T>接口 消费者接口

Function<T,R> 接口 表示接受一个参数并产生结果的函数。

Supplier<T>接口 代表结果供应商。

Predicate<T>接口 断言接口

|  |
| --- |
| **import** java.util.ArrayList;  **import** java.util.Arrays;  **import** java.util.List;  **import** java.util.function.Consumer;  **import** java.util.function.Function;  **import** java.util.function.Predicate;  **import** java.util.function.Supplier;  **public** **class** Test {  //Consumer 消费者接口  **private** **static** **void** consumerTest() {  List<String> result = **new** ArrayList<String>();  *print*("雄安", (str) -> result.add(str));  result.forEach(System.***out***::println);  }  **private** **static** **void** print(String str, Consumer<String> con) {  con.accept(str);  }  //Function 表示接受一个参数并产生结果的函数  **private** **static** **void** fuctionTest() {  String ss = *strToUpp*("sss", (str) -> str.toUpperCase());  System.***out***.println(ss);  }  **private** **static** String strToUpp(String str, Function<String, String> f) {  **return** f.apply(str);  }  //Supplier 代表结果供应商  **private** **static** **void** supplierTest() {  List<Integer> nums = *getNums*(10, () -> (**int**)(Math.*random*()\*100));  nums.forEach(System.***out***::println);  }  **private** **static** List<Integer> getNums(**int** num, Supplier<Integer> sup) {  List<Integer> result = **new** ArrayList<Integer>();  **for** (**int** i = 0; i < num; i++) {  result.add(sup.get());  }  **return** result;  }  //Predicate 断言接口  **private** **static** **void** pridicateTest() {  List<String> list = Arrays.*asList*("asd", "ert", "jkl", "qwu");  List<String> resultList = *filter*(list, (s) -> s.contains("d"));  resultList.forEach(System.***out***::println);  }  **private** **static** List<String> filter(List<String> list, Predicate<String> p) {  List<String> result = **new** ArrayList<String>();  list.forEach((s) -> {**if**(p.test(s)) result.add(s);});  **return** result;  }  **public** **static** **void** main(String[] args) {  *consumerTest*();  // pridicateTest();  // supplierTest();  // fuctionTest();  }  } |

### Stream的使用

什么是Stream？

Stream是元素的集合，这点让Stream看起来用些类似Iterator；

可以支持顺序和并行的对原Stream进行汇聚的操作；

我们可以把Stream当成一个高级版本的Iterator。原始版本的Iterator，用户只能一个一个的遍历元素并对其执行某些操作；高级版本的Stream，用户只要给出需要对其包含的元素执行什么操作，比如“过滤掉长度大于10的字符串”、“获取每个字符串的首字母”等，具体这些操作如何应用到每个元素上，就给Stream就好了！

Stream的常见操作有：

|  |  |  |
| --- | --- | --- |
| <R> Stream<R> | map(Function<? super T,? extends R> mapper) | 返回由给定函数应用于此流的元素的结果组成的流。 |
| <R> Stream<R> | flatMap(Function<? super T,? extends Stream<? extends R>> mapper) | 返回由通过将提供的映射函数应用于每个元素而产生的映射流的内容来替换该流的每个元素的结果的流。 |
| Stream<T> | filter(Predicate<? super T> predicate) | 返回由与此给定谓词匹配的此流的元素组成的流。 |
| void | forEach(Consumer<? super T> action) | 对此流的每个元素执行操作。 |
| Optional<T> | findFirst() | 返回描述此流的第一个元素的Optional如果流为空，则返回一个空的Optional 。 |
| Optional<T> | reduce(BinaryOperator<T> accumulator) | 使用 associative累积函数对此流的元素执行 reduction ，并返回描述减小值的 Optional （如果有）。 |
| Stream<T> | peek(Consumer<? super T> action) | 返回由该流的元素组成的流，另外在从生成的流中消耗元素时对每个元素执行提供的操作。 |
| Stream<T> | sorted() | 返回由此流的元素组成的流，根据自然顺序排序。 |

|  |
| --- |
| import java.util.Arrays;  import java.util.List;  import java.util.Optional;  import java.util.stream.Collector;  import java.util.stream.Collectors;  import java.util.stream.Stream;  /\*\*  \* Stream接口：不是存储数据结构，数据源可以是一个集合，为了函数式编程创造  \* 懒式执行，数据只能被消费一次  \* 两种类型的操作方法  \* 1、中间操作(生成一个Stream)  \* 2、结束操作（执行计算操作）  \*/  public class Test {  public static void main(String[] args) {  Stream<String> stream = Stream.of("Good", "Good", "Study", "Day", "Day", "up");  //foreach方法  // stream.forEach(System.out::println);  //filter  // stream.filter((s) -> s.length() > 3).forEach(System.out::println);  //distinct  // stream.distinct().forEach(System.out::println);  //map  // stream.map(s -> s.toUpperCase()).forEach(System.out::println);  //flatMap  // Stream<List<Integer>> ss = Stream.of(Arrays.asList(1, 2, 3), Arrays.asList(4, 5));  // ss.flatMap(list -> list.stream()).forEach(System.out::println);  //reduce  // Optional<String> reduce = stream.reduce((s1, s2) -> s1.length() > s2.length() ? s1 : s2 );  // System.out.println(reduce.get());  //collect  List<String> list = stream.collect(Collectors.toList());  list.forEach(System.out :: println);  //:: 方法引用  //引用静态方法 Integer :: valueOf  //引用对象的方法 list :: add  //引用构造方法 ArrayList :: new  }  } |

## 六．集合框架Map接口

**1、Map接口**

public interface **Map<K,V>**

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。

|  |  |
| --- | --- |
| void clear() | 清空Map集合中的内容 |
| boolean containsKey(Object key) | 判断集合中是否存在指定的key |
| boolean containsValue(Object value) | 判断集合中是否存在指定的value |
| Set<Map.Entry<K,V>> entrySet() | 将Map接口变为Set集合 |
| V get(Object key) | 根据key找到其对应的value |
| boolean isEmpty() | 判断是否为空 |
| Set<K> keySet() | 将全部的key变为Set集合 |
| Collection<V> values() | 将全部的value变为Collection集合 |
| V put(K key,V value) | 向集合中增加内容 |
| void putAll(Map<? extends K,? extends V> m) | 增加一组集合 |
| V remove(Object key) | 根据key删除内容 |

|  |
| --- |
| **import** java.util.Collection;  **import** java.util.HashMap;  **import** java.util.Map;  **import** java.util.Map.Entry;  **import** java.util.Set;  /\*\*  \* Map接口：  \* 1、键值对存储一组对象  \* 2、key不能重复（唯一），value可以重复  \* 3、具体的实现类：HashMap TreeMap Hashtable LinkedHashMap  \*/  **public** **class** Test {  **public** **static** **void** main(String[] args) {  Map<Integer, String> map = **new** HashMap<Integer, String>();  map.put(1, "Tom");  map.put(2, "Jack");  map.put(3, "Vince");  map.put(4, "BIn");  System.***out***.println(map.size());  System.***out***.println(map.containsKey(3));  //取值  System.***out***.println(map.get(1));  //map的遍历1  Set<Entry<Integer, String>> entrySet = map.entrySet();  **for** (Entry<Integer, String> set : entrySet) {  System.***out***.println(set.getKey() + "->" + set.getValue());  }  //map的遍历2（遍历键）  Set<Integer> keys = map.keySet();  **for** (Integer key : keys) {  System.***out***.println(key + "->" + map.get(key));  }  //map的遍历3（遍历值）  Collection<String> values = map.values();  **for** (String str : values) {  System.***out***.println(str);  }  //foreach  map.forEach((key, value) -> System.***out***.println(key + "->" + value));  }  } |

**2、HashMap**

public class **HashMap<K,V>** extends AbstractMap<K,V>

implements Map<K,V>, Cloneable, Serializable

基于哈希表的 Map 接口的实现。此实现提供所有可选的映射操作，并允许使用 null 值和 null 键。（除了非同步和允许使用 null 之外，HashMap 类与 Hashtable 大致相同。）此类不保证映射的顺序，特别是它不保证该顺序恒久不变。

**3、Hashtable**

public class **Hashtable<K,V>** extends Dictionary<K,V>

implements Map<K,V>, Cloneable, Serializable

此类实现一个哈希表，该哈希表将键映射到相应的值。任何非 null 对象都可以用作键或值。

为了成功地在哈希表中存储和获取对象，用作键的对象必须实现 hashCode 方法和 equals 方法。

**4、LinkedHashMap**

public class **LinkedHashMap<K,V>**

extends HashMap<K,V> implements Map<K,V>

Map 接口的哈希表和链接列表实现，具有可预知的迭代顺序。此实现与 HashMap 的不同之处在于，后者维护着一个运行于所有条目的双重链接列表。

**5、TreeMap**

public class **TreeMap<K,V>** extends AbstractMap<K,V>

implements NavigableMap<K,V>, Cloneable, Serializable

基于红黑树（Red-Black tree）的 NavigableMap 实现。该映射根据其键的自然顺序进行排序，或者根据创建映射时提供的 Comparator 进行排序，具体取决于使用的构造方法。

|  |
| --- |
| **import** java.util.HashMap;  **import** java.util.Hashtable;  **import** java.util.LinkedHashMap;  **import** java.util.Map;  **import** java.util.TreeMap;  /\*\*  \* Map接口：  \* 1、键值对存储一组对象  \* 2、key不能重复（唯一），value可以重复  \* 3、具体的实现类：HashMap TreeMap Hashtable LinkedHashMap  \* 4、HashMap 与 Hashtable的区别  \* 5、如何选择是使用哪个  \* 6、数据结构：数组、链表、二叉树（红黑树）、哈希表（数组 +链表）、栈、队列  \*/  **public** **class** Test {  /\*\*  \* HashMap的实现原理：  \* 1、基于哈希表（数组+链表+二叉树（红黑树））1.8JDK  \* 2、默认加载因子为0.75，默认数组大小是16  \* 3、把对象存储到哈希表中，如何存储？  \* 把key对象通过hash()方法计算hash值，然后用这个hash值对数组长度取余数（默认16），  \* 来决定该对KEY对象在数组中存储的位置，当这个位置有多个对象时，以链表结构存储，  \* JDK1.8后，当链表长度大于8时，链表将转换为红黑树结构存储。  \* 这样的目的是为了取值更快，存储的数据量越大，性能越明显。  \* 4、扩充原理：当数组的容量超过了75%，那么表示该数组需要扩充，如何扩充？  \* 扩充的算法：当前的数组容量<<1（相当于是乘2），扩大一倍，扩充次数过多，会影响性能，  \* 每次扩充表示哈希表重新散列（重新计算每个对象的存储位置），我们在开发中尽量要减少扩充  \* 次数带来的性能问题。  \* 5、线程不安全，适合在单线程中使用。  \*/  **public** **static** **void** hashMap () {  Map<Integer, String> map = **new** HashMap<Integer, String>();  }  /\*\*  \* JDK1.0开始  \* 基于哈希表实现（数组 + 链表）  \* 默认数组大小为11，加载因子0.75  \* 扩充方式：原数组大小<<1（\*2）+1  \* 线程安全的，用在多线程访问时  \*/  **public** **static** **void** hashtable () {  Map<Integer, String> map = **new** Hashtable<Integer, String>();  }  /\*\*  \* LinkedHashMap是HashMap的子类，由于HashMap不能保证顺序恒久不变，此类使用一个双重链表来维护  \* 元素添加的顺序  \*/  **public** **static** **void** linkedHashMap () {  Map<Integer, String> map = **new** LinkedHashMap<Integer, String>();  }  /\*\*  \* 基于二叉树的红黑树实现  \* 与TreeSet实现原理相同，应用对象时，对象需要实现Comparable接口，  \* 重写compareTo方法来实现数据排序  \*/  **public** **static** **void** treeMap () {  Map<Integer, String> map = **new** TreeMap<Integer, String>();  }  } |

**1.8 Map接口的新方法：**

在JDK8中Map接口提供了一些新的便利的方法。因为在本文中我所提到的所有Map方法都是以默认值方法的方式实现的，所以现有的Map接口的实现可以直接拥有这些在默认值方法中定义的默认行为，而不需要新增一行代码。

getOrDefault(Object, V)

putIfAbsent(K,V)

remove(Object key, Object value)

replace(K,V)

replace(K,V,V)

compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)

computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)

merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction)

|  |
| --- |
| **import** java.util.HashMap;  **import** java.util.Map;  **public** **class** Test {  **public** **static** **void** main(String[] args) {  Map<Integer, String> map = **new** HashMap<Integer, String>();  map.put(1, "Jack");  map.put(2, "Tom");  map.put(3, "Bin");  map.put(4, "Vice");  //当为空的时候设置输出默认值  System.***out***.println(map.getOrDefault(5, "ddd")); //结果 ddd  //只会添加不存在相同key的值（避免出现值被替换）  map.putIfAbsent(4, "bbb");  //根据键和值都匹配时才删除  map.remove(4, "Vice");  //替换值  map.replace(3, "Vice");  //根据键和值都匹配时才替换  map.replace(3, "Vice", "Potato");  //替换val值  map.compute(1, (k, v) -> v + "Test");  //当value不为null时替换值  map.computeIfPresent(2, (k, v) -> v+"Test");  //当value为null时put值  map.computeIfAbsent(5, (val) -> "Vice");  //将旧值替换为新值  map.merge(4, "Test", (oldVal, newVal) -> oldVal.concat(newVal));  map.forEach((k, v) -> System.***out***.println(k + "->" + v));  }  } |

## 七．Collections工具类

**Collections类**

Collections工具类提供了大量针对Collection/Map的操作，总体可分为四类，都为静态（static）方法：

**1. 排序操作（主要针对List接口相关）**

reverse(List list)：反转指定List集合中元素的顺序

shuffle(List list)：对List中的元素进行随机排序（洗牌）

sort(List list)：对List里的元素根据自然升序排序

sort(List list, Comparator c)：自定义比较器进行排序

swap(List list, int i, int j)：将指定List集合中i处元素和j出元素进行交换

rotate(List list, int distance)：将所有元素向右移位指定长度，如果distance等于size那么结果不变

**2. 查找和替换（主要针对Collection接口相关）**

binarySearch(List list, Object key)：使用二分搜索法，以获得指定对象在List中的索引，前提是集合已经排序

max(Collection coll)：返回最大元素

max(Collection coll, Comparator comp)：根据自定义比较器，返回最大元素

min(Collection coll)：返回最小元素

min(Collection coll, Comparator comp)：根据自定义比较器，返回最小元素

fill(List list, Object obj)：使用指定对象填充

frequency(Collection Object o)：返回指定集合中指定对象出现的次数

replaceAll(List list, Object old, Object new)：替换

**3. 同步控制**

Collections工具类中提供了多个synchronizedXxx方法，该方法返回指定集合对象对应的同步对象，从而解决多线程并发访问集合时线程的安全问题。HashSet、ArrayList、HashMap都是线程不安全的，如果需要考虑同步，则使用这些方法。这些方法主要有：synchronizedSet、synchronizedSortedSet、synchronizedList、synchronizedMap、synchronizedSortedMap。

**特别需要指出的是，在使用迭代方法遍历集合时需要手工同步返回的集合。**

**4. 设置不可变集合**

Collections有三类方法可返回一个不可变集合：

emptyXxx()：返回一个空的不可变的集合对象

singletonXxx()：返回一个只包含指定对象的，不可变的集合对象。

unmodifiableXxx()：返回指定集合对象的不可变视图

**5. 其它**

disjoint(Collection<?> c1, Collection<?> c2) - 如果两个指定 collection 中没有相同的元素，则返回 true。

addAll(Collection<? super T> c, T... a) - 一种方便的方式，将所有指定元素添加到指定 collection 中。

Comparator<T> reverseOrder(Comparator<T> cmp) - 返回一个比较器，它强行反转指定比较器的顺序。如果指定比较器为 null，则此方法等同于 reverseOrder()（换句话说，它返回一个比较器，该比较器将强行反转实现 Comparable 接口那些对象 collection 上的自然顺序）。

|  |
| --- |
| **import** java.util.ArrayList;  **import** java.util.Collections;  **import** java.util.List;  **public** **class** Test {  **public** **static** **void** main(String[] args) {  List<String> list= **new** ArrayList<String>();  list.add("Jack");  list.add("Tom");  list.add("Bin");  list.add("Vice");  //反转指定List集合中元素的顺序  // Collections.reverse(list);  //对List中的元素进行随机排序（洗牌）  // Collections.shuffle(list);  //对List里的元素根据自然升序排序  // Collections.sort(list);  //自定义比较器进行排序  // Collections.sort(list, c);  //将指定List集合中i处元素和j出元素进行交换  // Collections.swap(list, 0, 2);  //将所有元素向右移位指定长度，如果distance等于size那么结果不变  // Collections.rotate(list, 3);  //使用二分搜索法，以获得指定对象在List中的索引，前提是集合已经排序  // System.out.println(Collections.binarySearch(list, "Bin"));  //返回最大元素  // System.out.println(Collections.max(list));  //根据自定义比较器，返回最大元素  // Collections.max(Collection coll, Comparator comp);  //返回最小元素  // System.out.println(Collections.min(list));  //根据自定义比较器，返回最小元素  // Collections.min(Collection coll, Comparator comp);  //使用指定对象填充  // Collections.fill(list, "bin");  //返回指定集合中指定对象出现的次数  // System.out.println(Collections.frequency(list,"bin"));  //替换  // Collections.replaceAll(list, "Tom", "Tom2");  // System.out.println(list);  //同步控制  List<String> syncList = Collections.*synchronizedList*(**new** ArrayList<String>());  //返回一个空的不可变的集合对象,size=0不同于List<String> = null;  List<String> emptyList = Collections.*emptyList*();  //返回一个只包含指定对象的，不可变的集合对象。  List<String> singletonList = Collections.*singletonList*("Tom");  //返回指定集合对象的不可变视图  List<String> unmodifiableList = Collections.*unmodifiableList*(list);  System.***out***.println(unmodifiableList);  }  } |

## 八．Optional容器类（JDK1.8）

**Optional容器类（JDK1.8）**

这是一个可以为null的容器对象。如果值存在则isPresent()方法会返回true，调用get()方法会返回该对象。

**of：**

为非null的值创建一个Optional。

**ofNullable：**

为指定的值创建一个Optional，如果指定的值为null，则返回一个空的Optional。

**isPresent：**

如果值存在返回true，否则返回false。

**get：**

如果Optional有值则将其返回，否则抛出NoSuchElementException。

**ifPresent：**

如果Optional实例有值则为其调用consumer，否则不做处理

**orElse：**

如果有值则将其返回，否则返回指定的其它值。

**orElseGet：**

orElseGet与orElse方法类似，区别在于得到的默认值。orElse方法将传入的字符串作为默认值，orElseGet方法可以接受Supplier接口的实现用来生成默认值。

**orElseThrow：**

如果有值则将其返回，否则抛出supplier接口创建的异常。

**map：**

如果有值，则对其执行调用mapping函数得到返回值。如果返回值不为null，则创建包含mapping返回值的Optional作为map方法返回值，否则返回空Optional。

**flatMap：**

如果有值，为其执行mapping函数返回Optional类型返回值，否则返回空Optional。flatMap与map（Funtion）方法类似，区别在于flatMap中的mapper返回值必须是Optional。调用结束时，flatMap不会对结果用Optional封装。

**filter：**

如果有值并且满足断言条件返回包含该值的Optional，否则返回空Optional。

|  |
| --- |
| **import** java.util.Optional;  /\*\*  \* 使用情景：一个方法要使用另一个方法的返回值时，判断返回值是否为空使用  \*/  **public** **class** Test {  **public** **static** **void** main(String[] args) {  //创建Optional对象  //方法一：创建非空的Optional对象  Optional<String> optional = Optional.*of*("bin");  //方法二：创建可空的Optional对象（当值为null时返回Optional.empty）  Optional<String> optional2 = Optional.*ofNullable*(**null**);  //方法三：创建空的Optional对象  Optional<String> optional3 = Optional.*empty*();    //如果值存在返回true，否则返回false。  System.***out***.println(optional2.isPresent());  //如果Optional有值则将其返回，否则抛出NoSuchElementException。  System.***out***.println(optional.get());  //如果Optional实例有值则为其调用consumer，否则不做处理  optional.ifPresent((val) -> System.***out***.println(val));  //如果有值则将其返回，否则返回指定的其它值  System.***out***.println(optional2.orElse("default"));  //如果有值则将其返回，否则返回指定的其它值  System.***out***.println(optional2.orElse("default"));  //与orElse方法类似,接受Supplier接口的实现用来生成默认值  System.***out***.println(optional2.orElseGet(() -> "default"));  //如果有值则将其返回，否则抛出supplier接口创建的异常  //() -> new NullPointerException() = NullPointerException :: new  **try** {  System.***out***.println(optional.orElseThrow(NullPointerException :: **new**));  } **catch** (Exception e) {  e.printStackTrace();  }  //如果有值 返回function后的Optional对象，否则返回 空的Optional对象  Optional<String> optional4 = optional.map((val) -> val.toUpperCase());  System.***out***.println(optional4.orElse("no found"));  //如果有值 返回function后的Optional对象，否则返回 空的Optional对象  //(与map区别function中的方法体必须返回Optional对象)  Optional<Object> optional5 = optional2.flatMap((val) -> Optional.*of*(val.toUpperCase()+"\_flatMap"));  System.***out***.println(optional5.orElse("no found"));  //如果有值并且满足断言条件返回包含该值的Optional，否则返回空Optional。  Optional<String> optional6 = optional2.filter((val) -> val.length()>2);  System.***out***.println(optional6.orElse("长度小于2"));  }  } |

## 九．队列（Queue）、双端对列（Deque）接口、堆栈（Stack）类

1、队列（Queue）是一种特殊的线性表，是一种先进先出（FIFO）的数据结构。它只允许在表的前端（front）进行删除操作，而在表的后端（rear）进行插入操作。进行插入操作的端称为队尾，进行删除操作的端称为队头。队列中没有元素时，称为空队列。

**LinkedList是Queue接口的实现类**

**boolean add(E e) ：**将指定的元素插入此队列（如果立即可行且不会违反容量限制），在成功时返回 true，如果当前没有可用的空间，则抛出 IllegalStateException。

**E element() ：**获取，但是不移除此队列的头。

**boolean offer(E e) ：**将指定的元素插入此队列（如果立即可行且不会违反容量限制），当使用有容量限制的队列时，此方法通常要优于 add(E)，后者可能无法插入元素，而只是抛出一个异常。

**E peek() ：**获取但不移除此队列的头；如果此队列为空，则返回 null。

**E poll() ：**获取并移除此队列的头，如果此队列为空，则返回 null。

**E remove() ：**获取并移除此队列的头。

2、双端对列（Deque）：一个线性 collection，支持在两端插入和移除元素。

此接口既支持有容量限制的双端队列，也支持没有固定大小限制的双端队列。

接口定义在双端队列两端访问元素的方法。提供插入、移除和检查元素的方法。

3、堆栈（Stack）:支持先进后出形式移除数据。

|  |
| --- |
| import java.util.Deque;  import java.util.LinkedList;  import java.util.Queue;  import java.util.Stack;  public class Test {  /\*\*  \* Queue接口：队列，是一种先进先出的线性数据结构（排队）  \* LinkedList类实现了queue接口  \*/  public static void queue() {  Queue<String> queue = new LinkedList<String>();  //如果当前没有可用的空间，则抛出 IllegalStateException。  queue.add("张老师");  //如果当前没有可用的空间，此方法通常要优于 add(E)  queue.offer("李老师");  queue.offer("王老师");  queue.offer("薛老师");  System.out.println(queue.size()); //4  //获取此队列的头，如果此队列为空，则返回 null。  System.out.println(queue.peek());  //获取此队列的头，如果此队列为空时报异常NoSuchElementException  System.out.println(queue.element());  //获取并移除此队列的头，如果此队列为空，则返回 null。  System.out.println(queue.poll());  //获取并移除此队列的头，如果此队列为空报异常NoSuchElementException  System.out.println(queue.remove());  System.out.println(queue.size()); //2  }  /\*\*  \* Deque接口：双端队列，一个线性 collection，支持在两端插入和移除元素  \* LinkedList类实现了Deque接口  \*/  public static void deque() {  Deque<String> deque = new LinkedList<String>();  Deque<String> deque2 = new LinkedList<String>();  //如果当前没有可用的空间，则抛出 IllegalStateException。  deque.add("张老师");  //如果当前没有可用的空间，此方法通常要优于 add(E)  deque.offer("李老师");  deque.offer("苏老师");  deque.offer("庞老师");  //将数据插入到头部  deque.offerFirst("王老师");  //将数据插入到尾部  deque.offerLast("薛老师");  System.out.println(deque.size()); //6  //获取此队列的头，如果此队列为空，返回null  System.out.println(deque.peek());  System.out.println(deque.peekFirst());  //获取此队列的尾，如果此队列为空，返回null  System.out.println(deque.peekLast());  //获取此队列的头，如果此队列为空NoSuchElementException  System.out.println(deque.element());  System.out.println(deque.getFirst());  //获取此队列的尾，如果此队列为空NoSuchElementException  System.out.println(deque.getLast());  //获取并移除此队列的头，如果此队列为空，则返回 null。  System.out.println(deque.poll());  System.out.println(deque.pollFirst());  //获取并移除此队列的尾，如果此队列为空，则返回 null。  System.out.println(deque.pollLast());  //获取并移除此队列的头，如果此队列为空报异常NoSuchElementException  System.out.println(deque.remove());  System.out.println(deque.removeFirst());  //获取并移除此队列的尾，如果此队列为空报异常NoSuchElementException  System.out.println(deque.removeLast());  System.out.println(deque.size()); //0  }  /\*\*  \* Stack:堆栈：先进后出  \*/  public static void stack() {  Stack<String> stack = new Stack<String>();  stack.push("李老师");  stack.push("王老师");  stack.push("薛老师");  System.out.println(stack.size());//3  //获取此队列的尾，如果此队列为空，返回null  System.out.println(stack.peek());  //获取并移除此队列的尾，如果此队列为空，则返回 null。  System.out.println(stack.pop());  System.out.println(stack.size());//2    }  public static void main(String[] args) {  // queue();  // deque();  stack();  }  } |

## 十．对象一对多与多对多关系

|  |
| --- |
| **import** java.util.HashSet;  **import** java.util.Set;  **public** **class** Test {  //一对多  **public** **static** **void** oneToMany() {  Teacher t = **new** Teacher(1,"张老师",35);  Student s1 = **new** Student(1, "tom", 12);  Student s2 = **new** Student(2, "Lili", 13);  Student s3 = **new** Student(3, "schu", 15);  Student s4 = **new** Student(4, "jack", 14);  t.getStudent().add(s1);  t.getStudent().add(s2);  t.getStudent().add(s3);  t.getStudent().add(s4);  s1.setTeacher(t);  s2.setTeacher(t);  s3.setTeacher(t);  s4.setTeacher(t);  System.***out***.println(t);  t.getStudent().forEach((s) -> System.***out***.println(s));  }    **public** **static** **void** main(String[] args) {  *oneToMany*();  }  }  //老师  **class** Teacher {  **private** **int** id;  **private** String name;  **private** **int** age;  **private** Set<Student> student = **new** HashSet<Student>();  **public** Teacher() {  **super**();  }  **public** Teacher(**int** id, String name, **int** age) {  **super**();  **this**.id = id;  **this**.name = name;  **this**.age = age;  }  **public** Set<Student> getStudent() {  **return** student;  }  **public** **void** setStudent(Set<Student> student) {  **this**.student = 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;  }  **public** **int** getAge() {  **return** age;  }  **public** **void** setAge(**int** age) {  **this**.age = age;  }  @Override  **public** String toString() {  **return** "Teacher [id=" + id + ", name=" + name + ", age=" + age + "]";  }  }  //学生  **class** Student {  **private** **int** id;  **private** String name;  **private** **int** age;  **private** Teacher teacher;  **private** Set<StudentAndCourse> sc = **new** HashSet<StudentAndCourse>();  **public** Student() {  **super**();  }  **public** Student(**int** id, String name, **int** age) {  **super**();  **this**.id = id;  **this**.name = name;  **this**.age = age;  }  **public** Set<StudentAndCourse> getSc() {  **return** sc;  }  **public** **void** setSc(Set<StudentAndCourse> sc) {  **this**.sc = sc;  }  **public** Teacher getTeacher() {  **return** teacher;  }  **public** **void** setTeacher(Teacher teacher) {  **this**.teacher = teacher;  }  **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;  }  **public** **int** getAge() {  **return** age;  }  **public** **void** setAge(**int** age) {  **this**.age = age;  }  @Override  **public** String toString() {  **return** "Student [id=" + id + ", name=" + name + ", age=" + age + "]";  }  }  //课程  **class** Course{  **private** **int** id;  **private** String name;  **private** Set<StudentAndCourse> sc = **new** HashSet<StudentAndCourse>();  **public** Course() {  **super**();  }  **public** Course(**int** id, String name) {  **super**();  **this**.id = id;  **this**.name = name;  }  **public** Set<StudentAndCourse> getSc() {  **return** sc;  }  **public** **void** setSc(Set<StudentAndCourse> sc) {  **this**.sc = sc;  }  **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** "Course [id=" + id + ", name=" + name + "]";  }  }  //学生与课程的关系  **class** StudentAndCourse {  **private** **int** id;  **private** **int** sid;  **private** **int** cid;  **public** StudentAndCourse() {  **super**();  }  **public** StudentAndCourse(**int** id, **int** sid, **int** cid) {  **super**();  **this**.id = id;  **this**.sid = sid;  **this**.cid = cid;  }  **public** **int** getId() {  **return** id;  }  **public** **void** setId(**int** id) {  **this**.id = id;  }  **public** **int** getSid() {  **return** sid;  }  **public** **void** setSid(**int** sid) {  **this**.sid = sid;  }  **public** **int** getCid() {  **return** cid;  }  **public** **void** setCid(**int** cid) {  **this**.cid = cid;  }  } |

## 十一.迭代器设计模式

提供一个方法按顺序遍历一个集合内的元素，而又不需要暴露该对象的内部表示。

**应用场景**

1、访问一个聚合的对象，而不需要暴露对象的内部表示

2、支持对聚合对象的多种遍历

3、对遍历不同的对象，**提供统一的接口**。

**迭代器模式的角色构成**

(1)迭代器角色（Iterator）:定义遍历元素所需要的方法，一般来说会有这么三个方法：取得下一个元素的方法next()，判断是否遍历结束的方法hasNext()），移出当前对象的方法remove(),

(2)具体迭代器角色（Concrete Iterator）：实现迭代器接口中定义的方法，完成集合的迭代。

(3)容器角色(Aggregate): 一般是一个接口，提供一个iterator()方法，例如java中的Collection接口，List接口，Set接口等

(4)具体容器角色（ConcreteAggregate）：就是抽象容器的具体实现类，比如List接口的有序列表实现ArrayList，List接口的链表实现LinkedList，Set接口的哈希列表的实现HashSet等。

|  |
| --- |
| **public** **class** Test {  **public** **static** **void** main(String[] args) {  MyList list = **new** ConcreteAggregate();  list.add("张老师");  list.add("李老师");  list.add("王老师");  System.***out***.println(list.getSize());  System.***out***.println(list.get(0));  Iterator iterator = list.iterator();  **while**(iterator.hasNext()) {  String str = iterator.next().toString();  System.***out***.println(str);  }  }  }  /\*\*  \* 迭代器的接口  \*/  **interface** Iterator {  **public** **boolean** hasNext();  **public** Object next();  }  /\*\*  \* 迭代器接口具体实现类  \*/  **class** ConcreteIterator **implements** Iterator {  **private** MyList list = **null**;  **private** **int** index;  **public** ConcreteIterator(MyList list) {  **this**.list = list;  }  @Override  **public** **boolean** hasNext() {  **if**(index >= list.getSize()) {  **return** **false**;  } **else** {  **return** **true**;  }  }  @Override  **public** Object next() {  Object obj = list.get(index);  index++;  **return** obj;  }    }  /\*\*  \* 容器接口  \*/  **interface** MyList {  **void** add(Object obj);  Object get(**int** index);  Iterator iterator();  **int** getSize();  }  /\*\*  \* 容器接口实现类  \*/  **class** ConcreteAggregate **implements** MyList {  **private** Object[] elements;  **private** **int** size;  **private** **int** index;  **public** ConcreteAggregate() {  elements = **new** Object[100];  }  @Override  **public** **void** add(Object e) {  elements[index++] = e;  size++;  }  @Override  **public** Object get(**int** index) {  **return** elements[index];  }  @Override  **public** Iterator iterator() {  **return** **new** ConcreteIterator(**this**);  }  @Override  **public** **int** getSize() {  **return** **this**.size;  }  } |

## 十二.guava对集合的支持

版本下载：https://repo1.maven.org/maven2/com/google/guava/guava/

Guava工程包含了若干被Google的 Java项目广泛依赖 的核心库，例如：集合 [collections] 、缓存 [caching] 、原生类型支持 [primitives support] 、并发库 [concurrency libraries] 、通用注解 [common annotations] 、字符串处理 [string processing] 、I/O 等等。 所有这些工具每天都在被Google的工程师应用在产品服务中。

Guava对JDK集合的扩展，这是Guava最成熟和为人所知的部分。

1、 不可变集合：用不变的集合进行防御性编程和性能提升。

2 、新集合类型： multisets, multimaps, tables,等

3、 强大的集合工具类：提供java.util.Collections中没有的集合工具

4 、扩展工具类：让实现和扩展集合类变得更容易，比如创建Collection的装饰器，或实现迭代器

1、只读设置

2、函数式编程:过滤器

3、函数式编程:转换

4、组合式函数编程

5、加入约束：非空、长度验证

6、集合操作：交集、差集、并集

7、Multiset：无序可重复

8、Multimap key可以重复

9、BiMap：双向Map(bidirectional Map) 键与值不能重复

10、双键的Map -->Table --->rowKey+columnKye+value

|  |
| --- |
| **import** java.text.SimpleDateFormat;  **import** java.util.Collection;  **import** java.util.List;  **import** java.util.Set;  **import** org.junit.Test;  **import** com.google.common.base.Function;  **import** com.google.common.base.Functions;  **import** com.google.common.collect.ArrayListMultimap;  **import** com.google.common.collect.BiMap;  **import** com.google.common.collect.Collections2;  **import** com.google.common.collect.HashBasedTable;  **import** com.google.common.collect.HashBiMap;  **import** com.google.common.collect.HashMultiset;  **import** com.google.common.collect.ImmutableList;  **import** com.google.common.collect.Lists;  **import** com.google.common.collect.Multimap;  **import** com.google.common.collect.Sets;  **import** com.google.common.collect.Sets.SetView;  **import** com.google.common.collect.Table;  **import** com.google.common.collect.Table.Cell;  **public** **class** KnowledgeTest {  /\*\*  \* 设置只读  \*/  @Test  **public** **void** onlyRead() {  System.***out***.println("-----只读-------");  ImmutableList<String> list = ImmutableList.*of*("tom", "jack", "limi", "alun", "yahu");  list.forEach((str) -> System.***out***.println(str));  }  /\*\*  \* 过滤器  \*/  @Test  **public** **void** fllter() {  System.***out***.println("-----过滤器-------");  List<String> list = Lists.*newArrayList*("tom", "jack", "limi", "jlun", "yahu");  Collection<String> filter = Collections2.*filter*(list, (e) -> e.startsWith("j"));  filter.forEach(System.***out***::println);  }  /\*\*  \* 转换  \*/  @Test  **public** **void** conversion() {  System.***out***.println("-----转换-------");  Set<Long> set = Sets.*newHashSet*(18732305L, 17670823L, 20042604L);  Collection<String> timeCollection = Collections2.*transform*(set, (e) -> **new** SimpleDateFormat("yyyy-MM-dd").format(e));  timeCollection.forEach(System.***out***::println);  }  /\*\*  \* 组合式函数  \*/  @Test  **public** **void** combination() {  System.***out***.println("-----组合式函数-------");  List<String> list = Lists.*newArrayList*("tom", "jack", "limi", "jlun", "yahutt");  Function<String, String> f1 = **new** Function<String, String>() {  @Override  **public** String apply(String t) {  **return** t.length() > 4 ? t.substring(0, 4) : t;  }  };  Function<String, String> f2 = **new** Function<String, String>() {  @Override  **public** String apply(String t) {  **return** t.toUpperCase();  }  };  Function<String, String> compose = Functions.*compose*(f1, f2);  Collection<String> fcollection = Collections2.*transform*(list, compose);  fcollection.forEach(System.***out***::println);  }  /\*\*  \* 加入约束：非空、长度验证  \*/  @Test  **public** **void** constraint () {  Set<Object> hs = Sets.*newHashSet*();  //1.4版本可用  // Constraint<String> constraint = new Constraint<>() {  // public String chekElement (String element) {  //  // }  // }  // Preconditions.checkArgument(expression);  // Preconditions.checkNotNull(reference);  }  /\*\*  \* 集合操作：交集、差集、并集  \*/  @Test  **public** **void** setOperations() {  Set<Integer> set1 = Sets.*newHashSet*(1, 2, 3);  Set<Integer> set2 = Sets.*newHashSet*(3, 4, 5);  //交集(两个中共有的)  SetView<Integer> v1 = Sets.*intersection*(set1, set2);  v1.forEach(System.***out***::println);  //差集(set1中不包含set2的)  SetView<Integer> v2 = Sets.*difference*(set1, set2);  v2.forEach(System.***out***::println);  //并集(两个中的全部 共有的只出现一次)  SetView<Integer> v3 = Sets.*union*(set1, set2);  v3.forEach(System.***out***::println);  }  /\*\*  \* Multiset：无序可重复  \*/  @Test  **public** **void** multiset() {  String s = "good good study day day up";  String[] split = s.split(" ");  HashMultiset<String> set = HashMultiset.*create*();  **for** (String string : split) {  set.add(string);  }  Set<String> elementSet = set.elementSet();  **for** (String str : elementSet) {  System.***out***.println(str);  }  }  /\*\*  \* Multimap key可以重复  \*/  @Test  **public** **void** multimap() {  Multimap<String, String> map = ArrayListMultimap.*create*();  map.put("名著", "西游记");  map.put("名著", "水浒传");  map.put("名著", "三国演义");  map.put("小说", "三体");  map.put("小说", "笑看人生");  Set<String> ks = map.keySet();  **for** (String key : ks) {  Collection<String> vals = map.get(key);  System.***out***.println(key + "->" + vals);  }  //打印：名著->[西游记, 水浒传, 三国演义]  // 小说->[三体, 笑看人生]  }  /\*\*  \* BiMap：双向Map(bidirectional Map) 键与值不能重复  \*/  @Test  **public** **void** biMap() {  BiMap<String, String> map = HashBiMap.*create*();  map.put("bin\_test", "1111");  map.put("vinc\_test", "2222");  //map反转取值  String name = map.inverse().get("1111");  System.***out***.println(name);//结果：bin\_test  System.***out***.println(map.inverse().inverse() == map); //结果：true  }  /\*\*  \* 双键的Map -->Table --->rowKey+columnKye+value  \*/  @Test  **public** **void** table() {  Table<String, String, Integer> table = HashBasedTable.*create*();  table.put("小明", "数学", 100);  table.put("小红", "语文", 46);  table.put("小花", "英语", 78);  Set<Cell<String, String, Integer>> cells = table.cellSet();  **for** (Cell<String, String, Integer> cell : cells) {  System.***out***.println(cell.getRowKey() + "->" + cell.getColumnKey() + "->" + cell.getValue());  }  }  } |

# 第九章 多线程与并发

## 一．进程与线程

**1、什么是进程**

程序是指令和数据的有序集合，其本身没有任何运行的含义，是一个静态的概念。而进程是程序在处理机上的一次执行过程，它是一个动态的概念。

进程是一个具有一定独立功能的程序，一个实体，每一个进程都有它自己的地址空间。

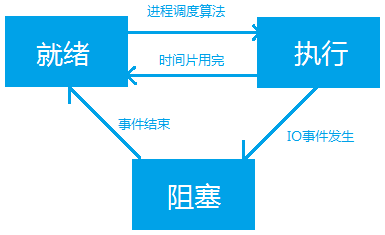
**2、进程的状态**

进程执行时的间断性，决定了进程可能具有多种状态。事实上，运行中的进程具有以下三种基本状态。

1）就绪状态（Ready）

2）运行状态（Running）

**3）**阻塞状态（Blocked）



**3、线程**

线程实际上是在进程基础之上的进一步划分，一个进程启动之后，里面的若干程序又可以划分成若干个线程。

线程：是进程中的一个执行路径，共享一个内存空间，线程之间可以自由切换，并发执行，一个进程最少有一个线程（单线程程序）

一个程序可以同时执行多个任务，来提高效率。

例如：

（1）、同时下载多个电影

（2）、同时与多人聊天

并行：就是两个任务同时运行(多个CPU)

并发：是指两个任务同时请求运行，而处理器一次只能接受一个任务，就会把两个任务安排轮流执行，由于CPU时间片运行时间较短,就会感觉两个任务在同时执行

## 二．线程的基本使用

|  |
| --- |
| package com.vince;  public class ThreadDemo {  public static void main(String[] args) {  MyThread mt = new MyThread();  mt.start(); //启动线程   //推荐使用  MyRunnable mr = new MyRunnable();  Thread tr = new Thread(mr);  tr.start();  } }  */\*\*  \* 实现线程的第一种方式，继承Thread类  \*/* class MyThread extends Thread {  @Override  public void run() {  for (int i = 0; i < 50; i++) {  System.*out*.println(Thread.*currentThread*().getName()+"\_"+i);  }  } }  */\*\*  \* 实现线程的第二种方式，实现runnable接口  \*/* class MyRunnable implements Runnable {  @Override  public void run() {  for (int i = 0; i < 50; i++) {  System.*out*.println(Thread.*currentThread*().getName()+"\_"+i);  }  } } |

## 三．线程休眠

public static void sleep(long millis) throws InterruptedException使当前正在执行的线程以指定的毫秒数暂停（暂时停止执行），释放CPU的时间片，具体取决于系统定时器和调度程序的精度和准确性。 线程不会丢失任何显示器的所有权。

参数

millis - 以毫秒为单位的睡眠时间长度

异常

IllegalArgumentException - 如果 millis值为负数

InterruptedException - 如果任何线程中断当前线程。 当抛出此异常时，当前线程的中断状态将被清除。

public static void sleep(long millis,int nanos) throws InterruptedException 毫秒，纳秒

static Thread currentThread() 返回对当前正在执行的线程对象的引用。

|  |
| --- |
| */\*\*  \* 实现线程的第二种方式，实现runnable接口  \* 线程休眠  \* 在当前线程的执行中，暂停指定的毫秒数，释放CPU的时间片  \*/* class MyRunnable implements Runnable {  @Override  public void run() {  for (int i = 0; i < 50; i++) {  System.*out*.println(Thread.*currentThread*().getName()+"\_"+i);  try {  Thread.*sleep*(500); //休眠半秒  } catch (InterruptedException e) {  e.printStackTrace();  }  }  } } |

## 四．Join与中断线程

**public final void join() throws InterruptedException**

等待这个线程死亡。

调用此方法的行为方式与调用完全相同

join (0)

异常 InterruptedException - 如果任何线程中断当前线程。 当抛出此异常时，当前线程的中断状态将被清除。

|  |
| --- |
| package com.vince;  public class ThreadDemo {  */\*\*  \* join方法  \* 加入线程，让调用的线程先执行指定时间或执行完毕  \** ***@param*** *args  \*/* public static void main(String[] args) {  MyRunnable mr = new MyRunnable();  Thread tr = new Thread(mr);  tr.start();   for (int i = 0; i < 50; i++) {  System.*out*.println(Thread.*currentThread*().getName()+"\_"+i);  try {  Thread.*sleep*(300); //休眠半秒  if (i == 20) {  tr.join(); //让tr线程执行完毕,在执行主线程  }  } catch (InterruptedException e) {  e.printStackTrace();  }  }  } }  class MyRunnable implements Runnable {  @Override  public void run() {  for (int i = 0; i < 50; i++) {  System.*out*.println(Thread.*currentThread*().getName()+"\_"+i);  try {  Thread.*sleep*(300);  } catch (InterruptedException e) {  e.printStackTrace();  }  }  } } |

**public void interrupt()**

中断这个线程。

除非当前线程中断自身，这是始终允许的

**public static boolean interrupted()**

测试当前线程是否中断。 该方法可以清除线程的中断状态 。 换句话说，如果这个方法被连续调用两次，那么第二个调用将返回false（除非当前线程再次中断，在第一个调用已经清除其中断状态之后，在第二个调用之前已经检查过）。

忽略线程中断，因为线程在中断时不存在将被该方法返回false所反映。

中断线程第一种方法：

|  |
| --- |
| package com.vince;  public class ThreadDemo {  */\*\*  \* 中断线程第一种方法  \* 使用interrupt方法来中断线程，设置一个中断状态（标记）*  *\*/* public static void main(String[] args) {  MyRunnable mr = new MyRunnable();  Thread tr = new Thread(mr);  tr.start();   for (int i = 0; i < 50; i++) {  System.*out*.println(Thread.*currentThread*().getName()+"\_"+i);  try {  Thread.*sleep*(300);   if (i == 20) {  tr.interrupt();//中断线程，只是做一个中断标记  }  } catch (InterruptedException e) {  e.printStackTrace();  }  }  } }  class MyRunnable implements Runnable {  @Override  public void run() {  for (int i = 0; i < 50; i++) {  if (Thread.*interrupted*()) { //测试中断状态，此方法会把中断状态清除  break;  }  System.*out*.println(Thread.*currentThread*().getName()+"\_"+i);  try {  Thread.*sleep*(300);  } catch (InterruptedException e) {  e.printStackTrace();  //由于睡眠方法抛出InterruptedException异常时，中断状态会消除，  //所以需要再次设置中断线程  Thread.*currentThread*().interrupt();  }  }  } } |

中断线程第二种方法：**自定义标记中断线程**

|  |
| --- |
| package com.vince;  public class ThreadDemo {  */\*\*  \* 中断线程第二种方法：  \* 自定义标记中断线程  \*/* public static void main(String[] args) {  MyRunnable mr = new MyRunnable();  Thread tr = new Thread(mr);  tr.start();   for (int i = 0; i < 50; i++) {  System.*out*.println(Thread.*currentThread*().getName()+"\_"+i);  try {  Thread.*sleep*(300);  if (i == 20) {  mr.flag = false;  }  } catch (InterruptedException e) {  e.printStackTrace();  }  }  } }  class MyRunnable implements Runnable {  public Boolean flag = true;   public MyRunnable() {  flag = true;  }  @Override  public void run() {  int i = 0;  while (flag) {  System.*out*.println(Thread.*currentThread*().getName()+"\_"+(i++));  try {  Thread.*sleep*(300);  } catch (InterruptedException e) {  e.printStackTrace();  }  }  } } |

## 五．守护线程与yield

**public final void setDaemon(boolean on)**

将此线程标记为daemon线程或用户线程。 当运行的唯一线程都是守护进程线程时，Java虚拟机将退出。

**public final boolean isDaemon()**

测试这个线程是否是守护线程。

**public static void yield()**

暂停当前正在执行的线程对象，并执行其他线程。（了解）

|  |
| --- |
| package com.vince;  public class ThreadDemo {  public static void main(String[] args) {  MyRunnable mr = new MyRunnable();  Thread tr = new Thread(mr);  //线程可以分成守护线程和用户线程，等用户中没有用户线程时，JVM会退出  tr.setDaemon(true);//把线程设置为守护线程  tr.start();   for (int i = 0; i < 50; i++) {  System.*out*.println(Thread.*currentThread*().getName()+"\_"+i);  try {  Thread.*sleep*(300);  } catch (InterruptedException e) {  e.printStackTrace();  }  }  } }  class MyRunnable implements Runnable {  @Override  public void run() {  for (int i = 0; i < 100; i++) {  System.*out*.println(Thread.*currentThread*().getName()+"\_"+i);  try {  Thread.*sleep*(300);  } catch (InterruptedException e) {  e.printStackTrace();  }  if (i == 5) {  Thread.*yield*(); //让出本次CPU执行的时间片  }  }  } } |

## 六．其它方法与优先级

long getId()   
          返回该线程的标识符。

String getName()   
          返回该线程的名称。

void setName(String name)   
          改变线程名称，使之与参数 name 相同。

boolean isAlive()   
          测试线程是否处于活动状态。

void setPriority(int newPriority)   
          更改线程的优先级。

static int MAX\_PRIORITY   
          线程可以具有的最高优先级。

static int MIN\_PRIORITY   
          线程可以具有的最低优先级。

static int NORM\_PRIORITY   
          分配给线程的默认优先级。

|  |
| --- |
| package com.vince;  public class ThreadDemo {  public static void main(String[] args) {  MyRunnable mr = new MyRunnable();  Thread tr = new Thread(mr);  tr.setName("Tread-tr"); //设置线程名称  //优先级高可以提高该线程抢占CPU时间片的概率大  tr.setPriority(Thread.*MAX\_PRIORITY*);//设置线程的优先级  System.*out*.println(tr.isAlive()); //测试线程活动状态 结果：false  tr.start();  System.*out*.println(tr.isAlive()); //测试线程活动状态 结果：true   for (int i = 0; i < 50; i++) {  System.*out*.println(Thread.*currentThread*().getName()+"\_"+i);  try {  Thread.*sleep*(300);  } catch (InterruptedException e) {  e.printStackTrace();  }  }  } }  class MyRunnable implements Runnable {  @Override  public void run() {  for (int i = 0; i < 50; i++) {  System.*out*.println(Thread.*currentThread*().getName()+"\_"+i);  try {  Thread.*sleep*(300);  } catch (InterruptedException e) {  e.printStackTrace();  }  }  } } |

## 七．线程同步

**1、多线程共享数据**

在多线程的操作中，多个线程有可能同时处理同一个资源，这就是多线程中的共享数据。

**2、线程同步**

解决数据共享问题，必须使用同步，所谓同步就是指 多个线程在同一个时间段内只能有一个线程执行指定代码，其他线程要等待此线程完成之后才可以继续执行。

线程进行同步，有以下三种方法：

**（1）同步代码块**

synchronized(要同步的对象){

要同步的操作 ;

}

**（2）同步方法**

public synchronized void method(){

要同步的操作 ;

}

**（3）Lock（ReentrantLock）**

**3、同步准则**

当编写 synchronized 块时，有几个简单的准则可以遵循，这些准则在避免死锁和性能危险的风险方面大有帮助：

（1）使代码块保持简短。把不随线程变化的预处理和后处理移出synchronized 块。

（2）不要阻塞。如InputStream.read()。

（3）在持有锁的时候，不要对其它对象调用方法

第一种使用同步代码块：

|  |
| --- |
| package com.vince;  public class ThreadDemo {  */\*\*  \* 1、多线程共享数据时，会发生线程不安全的情况  \* 2、多线程共享数据必须使用同步  \** ***@param*** *args  \*/* public static void main(String[] args) {  MyRunnable mr = new MyRunnable();  Thread tr = new Thread(mr);  Thread tr2 = new Thread(mr);  tr.start();  tr2.start();   } }  class MyRunnable implements Runnable {  private int ticket = 10; //售票  private Object obj = new Object();  @Override  public void run() {  for (int i = 0; i < 300; i++) {  synchronized (obj) {//同步锁 括号中可以使用obj或this  if (ticket > 0) {  ticket--;  try {  Thread.*sleep*(1000);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.*out*.println("您购买的票数剩余"+ticket+"张");  }  }  }  } } |

第二种使用同步方法：

|  |
| --- |
| package com.vince;  public class ThreadDemo {  */\*\*  \* 1、多线程共享数据时，会发生线程不安全的情况  \* 2、多线程共享数据必须使用同步  \** ***@param*** *args  \*/* public static void main(String[] args) {  MyRunnable mr = new MyRunnable();  Thread tr = new Thread(mr);  Thread tr2 = new Thread(mr);  tr.start();  tr2.start();   } }  class MyRunnable implements Runnable {  private int ticket = 10; //售票  private Object obj = new Object();  @Override  public void run() {  for (int i = 0; i < 300; i++) {  method();  }  }  //同步方法  private synchronized void method () {  if (ticket > 0) {  ticket--;  try {  Thread.*sleep*(1000);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.*out*.println("您购买的票数剩余"+ticket+"张");  }  } } |

第二种使用lock（更灵活的代码控制）

|  |
| --- |
| package com.vince;  import java.util.concurrent.locks.ReentrantLock;  public class ThreadDemo {  */\*\*  \* 1、多线程共享数据时，会发生线程不安全的情况  \* 2、多线程共享数据必须使用同步  \** ***@param*** *args  \*/* public static void main(String[] args) {  MyRunnable mr = new MyRunnable();  Thread tr = new Thread(mr);  Thread tr2 = new Thread(mr);  tr.start();  tr2.start();   } }  class MyRunnable implements Runnable {  private int ticket = 10; //售票  private Object obj = new Object();  @Override  public void run() {  for (int i = 0; i < 300; i++) {  method();  }  }  //Lock实现同步  ReentrantLock lock = new ReentrantLock();  private void method () {  lock.lock(); // 上锁  try {  if (ticket > 0) {  ticket--;  try {  Thread.*sleep*(1000);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.*out*.println("您购买的票数剩余"+ticket+"张");  }  } finally {  lock.unlock(); //解锁  }  } } |

## 八．死锁

过多的同步有可能出现死锁，死锁的操作一般是在程序运行的时候才有可能出现。

**多线程中要进行资源的共享，就需要同步，但同步过多，就可能造成死锁。**

|  |
| --- |
| package com.vince;  */\*\*  \* 线程死锁：在一个同步方法中调用了另一个对象的同步方法，可能产生死锁  \*/* public class DeadThreadDemo {  public static void main(String[] args) {  new DeadThead();  } }  //死锁线程 class DeadThead implements Runnable {  Waiter w = new Waiter();  Customer c = new Customer();  public DeadThead () {  new Thread(this).start();  w.say(c);  }  @Override  public void run() {  c.say(w);  } }  class Waiter {  public synchronized void say (Customer c) {  System.*out*.println("服务员说：先买单在吃饭！");  c.doService();  }  public synchronized void doService () {  System.*out*.println("同意了，吃完饭再买单！");  } }  class Customer{  public synchronized void say (Waiter w) {  System.*out*.println("顾客说：先吃饭在买单！");  w.doService();  }  public synchronized void doService () {  System.*out*.println("同意了，买完单在吃饭！");  } } |

## 九．生产者与消费者应用案例

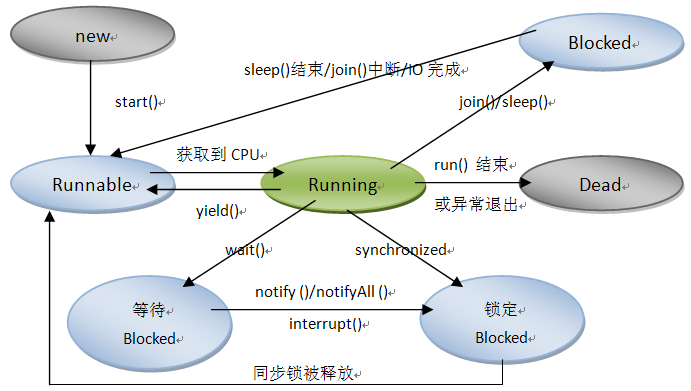
多线程的开发中有一个最经典的操作案例，就是生产者-消费者，生产者不断生产产品，消费者不断取走产品。

例如：饭店里的有一个厨师和一个服务员，这个服务员必须等待厨师准备好膳食。当厨师准备好时，他会通知服务员，之后服务员上菜，然后返回继续等待。这是一个任务协作的示例，厨师代表生产者，而服务员代表消费者。

|  |
| --- |
| package com.vince;  */\*\*  \* 两个线程协调工作，先生产，再消费  \* sleep与wait的区别  \* sleep：让线程进入休眠状态，让出CPU的时间片，不释放对象监视器的所有权（对象锁）  \* wait:让线程进入等待状态，让出CPU的时间片，并释放对象监视器的所有权，  \* 等待其他线程通过notify方法来唤醒  \*/* public class ProducterCustomerDemo {  public static void main(String[] args) {  Food food = new Food();  Thread tp = new Thread(new Producter(food));  Thread tc = new Thread(new Customers(food));  tp.start();  tc.start();  } }  */\*\*  \* 食物  \*/* class Food {  private String name;  private String desc;  private boolean flag = true; //true 可以生产，false 可以消费   */\*\*  \* 生产产品  \*/* public synchronized void setFood(String name, String desc) {  if (!flag) {//不能生产  try {  this.wait(); //线程进入等待状态，释放监视器的所有权（对象锁）  } catch (InterruptedException e) {  e.printStackTrace();  }  }  this.setName(name);  try {  Thread.*sleep*(500);  } catch (InterruptedException e) {  e.printStackTrace();  }  this.setDesc(desc);  System.*out*.println("生产：" + name +"->"+ desc);  flag = false;  this.notify(); //唤醒等待的线程（随机一个），notifyAll();唤醒所有线程  }   */\*\*  \* 消费产品  \*/* public synchronized void getFood () {  if (flag) {//不能消费  try {  this.wait(); //线程进入等待状态，释放监视器的所有权（对象锁）  } catch (InterruptedException e) {  e.printStackTrace();  }  }  try {  Thread.*sleep*(500);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.*out*.println("消费：" + this.getName() +"->"+ this.getDesc());  flag = true;  this.notify(); //唤醒等待的线程（随机一个），notifyAll();唤醒所有线程  }  public String getName() {  return name;  }   public void setName(String name) {  this.name = name;  }   public String getDesc() {  return desc;  }   public void setDesc(String desc) {  this.desc = desc;  }   @Override  public String toString() {  return "Food{" +  "name='" + name + '\'' +  ", desc='" + desc + '\'' +  '}';  }   public Food(String name, String desc) {  this.name = name;  this.desc = desc;  }   public Food() {  } }  */\*\*  \* 生产者  \*/* class Producter implements Runnable {  private Food food;  public Producter (Food food) {  this.food = food;  }  @Override  public void run() {  for (int i = 0; i < 20; i++) {  if (i%2 == 0) {  food.setFood("锅包肉", "酸甜口味，霜");  } else {  food.setFood("佛跳墙", "滋阴补阳");  }  }  } } */\*\*  \* 消费者  \*/* class Customers implements Runnable {  private Food food;  public Customers (Food food) {  this.food = food;  }  @Override  public void run() {  for (int i = 0; i < 20; i++) {  if (i%2 == 0) {  food.getFood();  }  }  } } |

## 十. 线程生命周期

线程生命周期图：



## 十一. 线程池

线程池是预先创建线程的一种技术。线程池在还没有任务到来之前，创建一定数量的线程，放入空闲队列中，然后对这些资源进行复用。减少频繁的创建和销毁对象。

jdk1.5版本以上提供了现成的线程池。

Java里面线程池的顶级接口是Executor，是一个执行线程的工具。

线程池接口是ExecutorService。

java.util.concurrent 包：并发编程中很常用的实用工具类

**Executor 接口：**执行已提交的 Runnable 任务的对象。

**ExecutorService 接口：** Executor 提供了管理终止的方法，以及可为跟踪一个或多个异步任务执行状况而生成 Future 的方法。

**Executors 类：**此包中所定义的Executor、ExecutorService等的工厂和实用方法。

在Executors类里面提供了一些静态工厂，生成一些常用的线程池。

**newSingleThreadExecutor**：创建一个单线程的线程池。这个线程池只有一个线程在工作，也就是相当于单线程串行执行所有任务。如果这个唯一的线程因为异常结束，那么会有一个新的线程来替代它。此线程池保证所有任务的执行顺序按照任务的提交顺序执行。

**newFixedThreadPool**：创建固定大小的线程池。每次提交一个任务就创建一个线程，直到线程达到线程池的最大大小。线程池的大小一旦达到最大值就会保持不变，如果某个线程因为执行异常而结束，

那么线程池会补充一个新线程。

**newCachedThreadPool**：创建一个可缓存的线程池。如果线程池的大小超过了处理任务所需要的线程，那么就会回收部分空闲（60秒不执行任务）的线程，当任务数增加时，此线程池又可以智能的添加新线程来处理任务。此线程池不会对线程池大小做限制，线程池大小完全依赖于操作系统（或者说JVM）能够创建的最大线程大小。

**newScheduledThreadPool**：创建一个大小无限的线程池。此线程池支持定时以及周期性执行任务的需求。

|  |
| --- |
| package com.vince;  import java.util.concurrent.Executors; import java.util.concurrent.ScheduledExecutorService; import java.util.concurrent.TimeUnit;  public class ThreadPoolDemo {  public static void main(String[] args) {  //1、创建一个单线程的线程池 // ExecutorService es = Executors.newSingleThreadExecutor(); // es.execute(new MyRunnable2()); // es.execute(new MyRunnable2());  //2、创建一个固定大小的线程池 // ExecutorService es = Executors.newFixedThreadPool(2); // es.execute(new MyRunnable2()); // es.execute(new MyRunnable2());  //3、创建一个不固定大小的线程池，线程数随任务的变化而变化 // ExecutorService es = Executors.newCachedThreadPool(); // es.execute(new MyRunnable2()); // es.execute(new MyRunnable2());  //4、创建一个大小无限的线程池。此线程池支持定时以及周期性执行任务的需求  ScheduledExecutorService es = Executors.*newScheduledThreadPool*(2);  es.schedule(new MyRunnable2(), 3000, TimeUnit.*MILLISECONDS*);  es.shutdown(); //结束  } }  class MyRunnable2 implements Runnable {   @Override  public void run() {  for (int i = 0; i < 10; i++) {  System.*out*.println(Thread.*currentThread*().getName()+ "->" +i);  try {  Thread.*sleep*(300);  } catch (InterruptedException e) {  e.printStackTrace();  }  }  } } |

# 第十章 网络编程

## 一. 网络编程基本概念

**1、什么是计算机网络**

把分布在不同地理区域的计算机与专门的外部设备用通信线路互连成一个规模大、功能强的网络系统，从而使众多的计算机可以方便地互相传递信息，共享硬件、软件、数据信息等资源。

**2、计算机网络的主要功能**

资源共享

信息传输与集中处理

均衡负荷与分布处理

综合信息服务(www/综合业务数字网络ISDN)等

**3、网络通信协议**

要使计算机连成的网络能够互通信息，需要对数据传输速率、传输代码、代码结构、传输控制步骤、出错控制等制定一组标准，这一组共同遵守的通信标准就是网络通信协议，不同的计算机之间必须使用相同的通讯协议才能进行通信。

**网络通信接口**

为了使两个结点之间能进行对话，必须在它们之间建立通信工具(即接口)，使彼此之间能进行信息交换。接口包括两部分：

（1）硬件装置：实现结点之间的信息传送

（2）软件装置：规定双方进行通信的约定协议

**4、TCP/IP**

TCP/IP：传输控制协议/因特网互联协议，又叫网络通讯协议，这个协议是Internet最基本的协议、Internet国际互联网络的基础，简单地说，就是由网络层的IP协议和传输层的TCP协议组成的。

IP地址：网络中每台计算机的一个标识号，本地IP：127.0.0.1 localhost

端口号(PORT)：端口号的范围：0~65535之间，0~1023之间的端口数是用于一些知名的网络服务和应用



**5、程序开发结构**

**网络编程主要是指完成C/S程序的开发，程序的开发结构有两种：**

· C/S（客户端/服务器）

开发两套程序，两套程序需要同时维护，例如：QQ。CS程序一般比较稳定

· B/S（浏览器/服务器）

开发一套程序，客户端使用浏览器进行访问，例如：各个论坛。BS程序一般稳定性较差，而且安全性较差。

但是，C/S的程序开发在实际的Java应用中毕竟很少了，而且整个java基本上都是以B/S为主。

**C/S程序主要可以完成以下两种程序的开发：**

· TCP：（Transmission Control Protocol）传输控制协议，采用三方握手的方式，保证准确的连接操作。

· UDP：（User Datagram Protocol）数据报协议，发送数据报，例如：手机短信或者是QQ消息。

 TCP、UDP的数据帧格式简单图例：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **协议类型** | **源IP** | **目标IP** | **源端口** | **目标端口** | **帧序号** | **帧数据** |

其中协议类型用于区分TCP、UDP

## 二．网络编程TCP协议

**1、TCP程序概述**

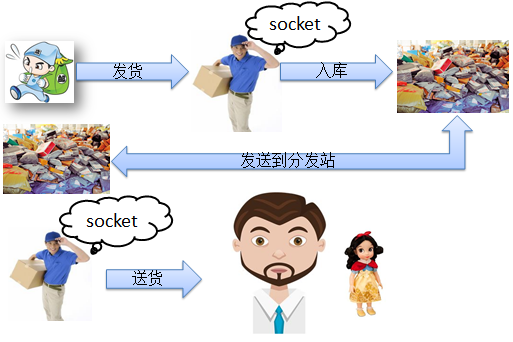
TCP是一个可靠的协议，面向连接的协议。

实现TCP程序，需要编写服务器端和客户端，Java API为我们提供了**java.net** 包，为实现网络应用程序提供类。

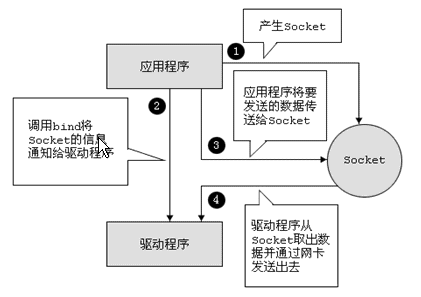
**ServerSocket** ：此类实现服务器套接字。

**Socket** ：此类实现客户端套接字（也可以就叫“套接字”）。

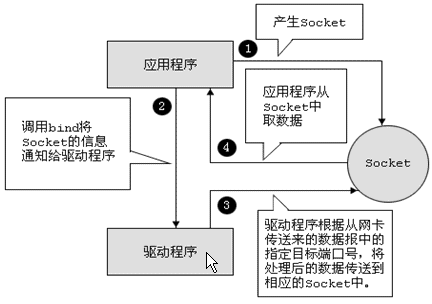
**Socket是网络驱动层提供给应用程序编程的接口和一种机制。**



**2、数据发送过程**



**3、数据接收过程**



**4、实现服务器端与客户端程序**

**服务器端：**

public class ServerSocket extends Object

此类实现服务器套接字。服务器套接字等待请求通过网络传入。它基于该请求执行某些操作，然后可能向请求者返回结果。

ServerSocket(int port) 创建绑定到特定端口的服务器套接字。

 void setSoTimeout(int timeout) 通过指定超时值启用/禁用 SO\_TIMEOUT，以毫秒为单位。

 InetAddress getInetAddress()  返回此服务器套接字的本地地址。

 Socket accept() 侦听并接受到此套接字的连接。

**客户端：**

public class Socket extends Object

此类实现客户端套接字（也可以就叫“套接字”）。套接字是两台机器间通信的端点。

Socket(String host, int port) 创建一个流套接字并将其连接到指定主机上的指定端口号。

InputStream getInputStream() 返回此套接字的输入流。

OutputStream getOutputStream()  返回此套接字的输出流。

 void setSoTimeout(int timeout) 启用/禁用带有指定超时值的 SO\_TIMEOUT，以毫秒为单位。

|  |
| --- |
| 服务端编码  package com.vince2;  import java.io.\*; import java.net.ServerSocket; import java.net.Socket;  public class EchoServerDemo {  public static void main(String[] args) {  try {  //创建一个服务器端的socket(端口号1024-65535)  ServerSocket server = new ServerSocket(6666);  System.*out*.println("服务器已启动，正在等待客户端的连接...");  //等待客户端的连接，造成阻塞，如果有客户端连接成功，立即返回一个Socket对象  Socket socket = server.accept();  System.*out*.println("客户端连接成功"+ socket.getInetAddress().getHostAddress());  BufferedReader br = new BufferedReader(  new InputStreamReader(socket.getInputStream()));  //通过输入流读取网络数据,如果没有数据，一直会阻塞  String info = br.readLine();  System.*out*.println(info);  PrintStream ps = new PrintStream(new BufferedOutputStream((socket.getOutputStream())));  ps.println("echo:"+info);  ps.flush();  //关闭  ps.close();  br.close();  } catch (IOException e) {  e.printStackTrace();  }  } }  客户端编码  package com.vince2;  import java.io.\*; import java.net.Socket;  public class EchoClientDemo {  public static void main(String[] args) {  try {  //创建一个socket对象，指定要连接的服务器  Socket socket = new Socket("localhost", 6666);  PrintStream ps = new PrintStream(new BufferedOutputStream(socket.getOutputStream()));  BufferedReader br = new BufferedReader(new InputStreamReader(socket.getInputStream()));  ps.println("hello, my name is Bin");  ps.flush();  //读取服务器端返回的数据  String info = br.readLine();  System.*out*.println(info);  ps.close();  br.close();  } catch (IOException e) {  e.printStackTrace();  }   } }  测试时需要先启动服务端Run As 在启动客户端Run As(服务端不会停) |

## 三．服务器与多客户端通信

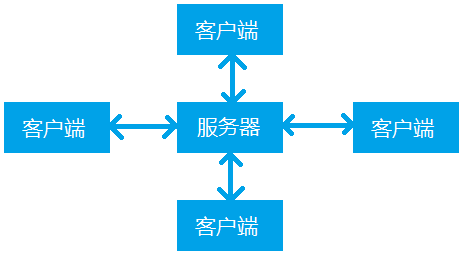
目前为止我们编写的程序中，服务器只能同时处理一个客户端连接，要想服务器同时支持多个客户端的连接，就必须加入多线程的处理机制，将每一个连接的客户端都创建一个新的线程对象。

|  |
| --- |
| 服务端代码  package com.vince2;  import java.io.\*; import java.net.ServerSocket; import java.net.Socket; import java.util.concurrent.ExecutorService; import java.util.concurrent.Executors;  */\*\*  \* 处理多个客户端  \* 主线程用于监听客户端的连接，每次有连接成功，开启一个线程来处理该客户端的消息  \*/* public class MutilServerDemo {  public static void main(String[] args) {  ExecutorService es = Executors.*newFixedThreadPool*(3);  try {  //创建一个服务器端的socket(端口号1024-65535)  ServerSocket server = new ServerSocket(6666);  System.*out*.println("服务器已启动，正在等待客户端的连接...");  while (true) {  //等待客户端的连接，造成阻塞，如果有客户端连接成功，立即返回一个Socket对象  Socket socket = server.accept();  System.*out*.println("客户端连接成功"+ socket.getInetAddress().getHostAddress());  es.execute(new UserThread(socket));  }    } catch (IOException e) {  e.printStackTrace();  }  } }  */\*\*  \* 用来处理客户端请求的任务  \*/* class UserThread implements Runnable {  private Socket socket;  public UserThread (Socket socket) {  this.socket = socket;  }  @Override  public void run() {  try {  BufferedReader br = new BufferedReader(new InputStreamReader(socket.getInputStream()));  //通过输入流读取网络数据,如果没有数据，一直会阻塞  String info = br.readLine();  System.*out*.println(info);  PrintStream ps = new PrintStream(new BufferedOutputStream((socket.getOutputStream())));  ps.println("echo:"+info);  ps.flush();  //关闭  ps.close();  br.close();  } catch (IOException e) {  e.printStackTrace();  }  } }  客户端代码  package com.vince2;  import java.io.\*; import java.net.Socket; import java.util.Scanner;  public class MutilClientDemo {  public static void main(String[] args) {  Scanner sc = new Scanner(System.*in*);  try {  //创建一个socket对象，指定要连接的服务器  Socket socket = new Socket("127.0.0.1", 6666);  PrintStream ps = new PrintStream(new BufferedOutputStream(socket.getOutputStream()));  BufferedReader br = new BufferedReader(new InputStreamReader(socket.getInputStream()));  System.*out*.println("请输入:");  String info = sc.nextLine();  ps.println(info);  ps.flush();  //读取服务器端返回的数据  info = br.readLine();  System.*out*.println(info);  ps.close();  br.close();  } catch (IOException e) {  e.printStackTrace();  }   } }  测试时需要先启动服务端Run As 在多次启动客户端Run As |

## 四、多客户端之间的通信

服务器可以与多个客户端实现通信了，那我们真正的目的是要实现多个客户端之间的通信，使用TCP协议实现的方案是：

客户端的数据包通过服务器中转，发送到另一个客户端，如下图所示：



|  |
| --- |
| 消息类型  package com.communication;  public class MessageType {  public static final int *TYPE\_LOGIN* = 0x1; //登录的消息类型  public static final int *TYPE\_SEND* = 0X2; //发送的消息类型 }  消息  package com.communication;  import java.io.Serializable;  */\*\*  \* 消息包  \*/* public class Message implements Serializable {  private String from; // 发送者  private String to; //接受者  private int type; //消息类型  private String info; //消息   public String getFrom() {  return from;  }   public void setFrom(String from) {  this.from = from;  }   public String getTo() {  return to;  }   public void setTo(String to) {  this.to = to;  }   public int getType() {  return type;  }   public void setType(int type) {  this.type = type;  }   public String getInfo() {  return info;  }   public void setInfo(String info) {  this.info = info;  }   public Message() {  }   public Message(String from, String to, int type, String info) {  this.from = from;  this.to = to;  this.type = type;  this.info = info;  } }  服务器端  package com.communication;  import java.io.\*; import java.net.ServerSocket; import java.net.Socket; import java.util.Vector; import java.util.concurrent.ExecutorService; import java.util.concurrent.Executors;  public class Server {  public static void main(String[] args) {  Vector<UserThread> vector = new Vector<UserThread>();  ExecutorService es = Executors.*newFixedThreadPool*(5);  try {  //创建一个服务器端的socket(端口号1024-65535)  ServerSocket server = new ServerSocket(8888);  System.*out*.println("服务器已启动，正在等待客户端的连接...");  while (true) {  //等待客户端的连接，造成阻塞，如果有客户端连接成功，立即返回一个Socket对象  Socket socket = server.accept();  System.*out*.println("客户端连接成功"+ socket.getInetAddress().getHostAddress());  es.execute(new UserThread(socket, vector));  }  } catch (IOException e) {  e.printStackTrace();  }  } } */\*\*  \* 用来处理客户端请求的任务  \*/* class UserThread implements Runnable {  private String name; //客户端用户名称（唯一）  private Socket socket;  private Vector<UserThread> vector;  private ObjectInputStream ois;  private ObjectOutputStream oos;  private boolean flag = true;  public UserThread (Socket socket, Vector<UserThread> vector) {  this.vector = vector;  this.socket = socket;  vector.add(this);  }  @Override  public void run() {  try {  System.*out*.println("客户端"+socket.getInetAddress().getHostAddress()+"已连接");  ois = new ObjectInputStream(socket.getInputStream());  oos = new ObjectOutputStream(socket.getOutputStream());  while (flag) {  Message msg = (Message) ois.readObject();  int type = msg.getType();  switch (type) {  case MessageType.*TYPE\_LOGIN*:  name = msg.getFrom();  msg.setInfo("欢迎你：");  oos.writeObject(msg);  break;  case MessageType.*TYPE\_SEND*:  String to = msg.getTo();  UserThread ut;  int size = vector.size();  for (int i = 0; i < size; i++) {  ut = vector.get(i);  if (to.equals(ut.name) && ut != this) {  ut.oos.writeObject(msg);  break;  }  }  break;  }   }  ois.close();  oos.close();  } catch (IOException | ClassNotFoundException e) {  e.printStackTrace();  }  } }  客户端  package com.communication;  import java.io.\*; import java.net.Socket; import java.util.Scanner; import java.util.concurrent.ExecutorService; import java.util.concurrent.Executors;  public class Client {  public static void main(String[] args) {  Scanner sc = new Scanner(System.*in*);  ExecutorService es = Executors.*newSingleThreadExecutor*();  try {  //创建一个socket对象，指定要连接的服务器  Socket socket = new Socket("localhost", 8888);  System.*out*.println("服务器连接成功");  ObjectOutputStream oos = new ObjectOutputStream(socket.getOutputStream());  ObjectInputStream ois = new ObjectInputStream(socket.getInputStream());  //向服务器发送登录信息  System.*out*.println("请输入名称:");  String name = sc.nextLine();  Message msg = new Message(name, null, MessageType.*TYPE\_LOGIN*, null);  oos.writeObject(msg);  msg = (Message) ois.readObject();  System.*out*.println(msg.getInfo() + msg.getFrom());  //启动读取消息的线程  es.execute(new ReadInfoThread(ois));  //使用主线程来发送消息  boolean falg = true;  while (falg) {  msg = new Message();  System.*out*.println("To:");  msg.setTo(sc.nextLine());  msg.setFrom(name);  msg.setType(MessageType.*TYPE\_SEND*);  System.*out*.println("Info:");  msg.setInfo(sc.nextLine());  oos.writeObject(msg);  }  } catch (IOException | ClassNotFoundException e) {  e.printStackTrace();  }   } }  //读取消息 class ReadInfoThread implements Runnable {  private ObjectInputStream in;  private boolean flag = true;  public ReadInfoThread (ObjectInputStream in) {  this.in = in;  }  public void setFlag(boolean flag) {  this.flag = flag;  }   @Override  public void run() {  try {  while (flag) {  Message msg = (Message) in.readObject();  System.*out*.println("[" + msg.getFrom() + "]对我说：" + msg.getInfo());  }  if (in != null) {  in.close();  }  } catch (IOException | ClassNotFoundException e) {  e.printStackTrace();  }  } }  启动服务端，启动多个客户端来进行交流测试 |

## 五．网络编程UDP协议

**1、UDP协议概述**

**UDP**是**User Datagram Protocol**的简称，是一种无连接的协议，每个数据报都是一个独立的信息，包括完整的源地址或目的地址，它在网络上以任何可能的路径传往目的地，因此能否到达目的地，到达目的地的时间以及内容的正确性都是不能被保证的，每个被传输的数据报必须限定在64KB之内。

主要使用以下的两个类：

DatagramPacket：此类表示数据报包。

DatagramSocket：此类表示用来发送和接收数据报包的套接字

**2、UDP服务器与客户端程序**

**服务器端：**

String info = “….”

// 将信息封装成数据报

byte[] bytes = info.getBytes();

DatagramPacket dp = **new** DatagramPacket(bytes, 0, bytes

.length(), InetAddress.*getByName*("localhost"), 5000);// 客户端在5000端口监听

DatagramSocket server = **new** DatagramSocket(3000);// 服务器的端口

server.send(dp);// 发送数据报

server.close();

**客户端：**

**byte** b[] = **new** **byte**[1024];// 接收内容

DatagramPacket dp = **new** DatagramPacket(b, b.length);// 接收内容

// 客户端在5000端口等待

DatagramSocket client = **new** DatagramSocket(5000);

client.receive(dp); // 接收内容

System.*out*.println(**new** String(dp.getData(), 0, dp.getLength()));

client.close();

|  |
| --- |
| 服务端  package com.vince2;  import java.io.IOException; import java.net.\*;  public class UDPServerDemo {  public static void main(String[] args) {  String info = "good good 学习，天天 up";  byte[] bytes = info.getBytes();  try {  */\*\*  \* 封装一个数据报包  \* buf - 数据包数据  \* offset - 分组数据偏移量  \* length - 分组数据长度  \* address - 目的地址  \* port - 目的端口号  \*/* DatagramPacket dp = new DatagramPacket(  bytes,  0,  bytes.length,  InetAddress.*getByName*("127.0.0.1"),  8000  );  //本程序的端口  DatagramSocket socket = new DatagramSocket(9000);  socket.send(dp);  } catch (UnknownHostException | SocketException e) {  e.printStackTrace();  } catch (IOException e) {  e.printStackTrace();  }  } }  客户端  package com.vince2;  import java.io.IOException; import java.net.DatagramPacket; import java.net.DatagramSocket; import java.net.SocketException;  public class UDPClientDemo {  public static void main(String[] args) {  byte[] bytes = new byte[1024];  DatagramPacket dp = new DatagramPacket(bytes, bytes.length);  try {  DatagramSocket socket = new DatagramSocket(8000);  System.*out*.println("正在接受数据中.....");  socket.receive(dp);//接受数据  String s = new String(dp.getData(), 0, dp.getLength());  System.*out*.println(s);  socket.close();  } catch (SocketException e) {  e.printStackTrace();  } catch (IOException e) {  e.printStackTrace();  }  } }  需要先启动客户端等待接受数据，再启动服务端 |

## 六．MINA框架

1、什么是MINA？ 一句话就是：**一个简洁易用的基于 TCP/IP 通信的 JAVA框架**。

2、下载地址：http://mina.apache.org/

3、一个简单的网络程序需要的最少jar包：mina-core-2.0.16.jar、slf4j-api-1.7.21.jar



4、开发一个 Mina 应用，简单的说，就是**创建连接，设定过滤规则，编写自己的消息处理器**

|  |
| --- |
| 编写服务端代码  package com.mina;  import org.apache.mina.core.filterchain.DefaultIoFilterChainBuilder; import org.apache.mina.filter.codec.ProtocolCodecFilter; import org.apache.mina.filter.codec.textline.TextLineCodecFactory; import org.apache.mina.transport.socket.SocketAcceptor; import org.apache.mina.transport.socket.nio.NioSocketAcceptor;  import java.io.IOException; import java.net.InetSocketAddress;  public class Server {  public static void main(String[] args) {  //创建一个非阻塞的Server端Socket,用NIO  SocketAcceptor acceptor = new NioSocketAcceptor(); //创建接收数据的过滤器  DefaultIoFilterChainBuilder chain = acceptor.getFilterChain();  //设定这个过滤器将一行一行(/r/n)的读取数据  chain.addLast("myChin", new ProtocolCodecFilter(new TextLineCodecFactory()));  //设定服务器端的消息处理器:一个SampleMinaServerHandler对象  acceptor.setHandler(new SampleMinaServerHandler());  int bindPort = 9999;  try {  //绑定端口,启动服务器(不会阻塞，立即返回)  acceptor.bind(new InetSocketAddress(bindPort));  } catch (IOException e) {  e.printStackTrace();  }  System.*out*.println("Mina Server is Listing on:= " + bindPort);   } }  编写服务端的消息处理器  package com.mina;  import org.apache.mina.core.service.IoHandlerAdapter; import org.apache.mina.core.session.IoSession;  */\*\*  \* 服务器端的消息处理器  \*/* public class SampleMinaServerHandler extends IoHandlerAdapter {  //一次会话被打开  @Override  public void sessionOpened(IoSession session) throws Exception {  super.sessionOpened(session);  //打印客户端地址  System.*out*.println("welcome client" + session.getRemoteAddress());  }  //会话关闭  @Override  public void sessionClosed(IoSession session) throws Exception {  super.sessionClosed(session);  System.*out*.println("client close");  }  //接受消息  @Override  public void messageReceived(IoSession session, Object message) throws Exception {  super.messageReceived(session, message);  //我们己设定了服务器解析消息的规则是一行一行读取,这里就可转为String:  String s=(String)message;  System.*out*.println("收到客户机发来的消息: "+s);  //向客户端发送消息对象  session.write("echo:"+s);  } }  **使用telnet测试：**telnet localhost 9999  查看本机Telnet客户端是否打开：    启动服务端Run，打开cmd输入telnet localhost 9999 然后出现一个空白页，在这里面输入信息回车，查看服务端控制台是否打印出来。  **客户端代码**  package com.mina;  import org.apache.mina.core.filterchain.DefaultIoFilterChainBuilder; import org.apache.mina.core.future.ConnectFuture; import org.apache.mina.filter.codec.ProtocolCodecFilter; import org.apache.mina.filter.codec.textline.TextLineCodecFactory; import org.apache.mina.transport.socket.nio.NioSocketConnector;  import java.net.InetSocketAddress; import java.util.Scanner;  public class Client {  public static void main(String[] args) {  NioSocketConnector connector = new NioSocketConnector(); // 创建接收数据的过滤器  DefaultIoFilterChainBuilder chain = connector.getFilterChain();  //设定这个过滤器将一行一行(/r/n)的读取数据  chain.addLast("myChin", new ProtocolCodecFilter(new TextLineCodecFactory()));  //设定服务器端的消息处理器:一个 SamplMinaServerHandler 对象  connector.setHandler(new SampleMinaClientHandler());  connector.setConnectTimeoutMillis(10000);// Set connect timeout.  //连接到服务器:  ConnectFuture cf = connector.connect(new InetSocketAddress("localhost", 9999));  cf.awaitUninterruptibly(); // 等待连接成功  Scanner sc = new Scanner(System.*in*);  while (true) {  System.*out*.println("请输入：");  String info = sc.nextLine();  //发送消息  cf.getSession().write(info);  }  //等待服务器连接关闭，结束长连接  //cf.getSession().getCloseFuture().awaitUninterruptibly();  //connector.dispose(); //关闭连接  } }  客户端的消息处理器  package com.mina;  import org.apache.mina.core.service.IoHandlerAdapter; import org.apache.mina.core.session.IoSession;  */\*\*  \* 客户端的消息处理器  \*/* public class SampleMinaClientHandler extends IoHandlerAdapter {  //当一个客端端连结进入时  @Override  public void sessionOpened(IoSession session) throws Exception {  super.sessionOpened(session);  System.*out*.println("incomming client :"+session.getRemoteAddress());   }  //当一个客户端关闭时  @Override  public void sessionClosed(IoSession session) throws Exception {  super.sessionClosed(session);  System.*out*.println("one Clinet Disconnect !");  }  //当客户端发送的消息到达时:  @Override  public void messageReceived(IoSession session, Object message) throws Exception {  super.messageReceived(session, message);  //我们己设定了服务器解析消息的规则是一行一行读取,这里就可转为 String:  String s=(String)message;  System.*out*.println("服务器发来的收到消息: "+s);  } } |

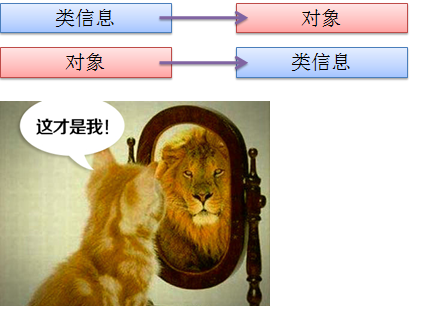
**5、使用 Mina 直接传送对象**

|  |
| --- |
| Message对象  package com.mina;  import java.io.Serializable;  */\*\*  \* 消息包  \*/* public class Message implements Serializable {  private String from; // 发送者  private String to; //接受者  private String type; //消息类型  private String info; //消息   public String getFrom() {  return from;  }   public void setFrom(String from) {  this.from = from;  }   public String getTo() {  return to;  }   public void setTo(String to) {  this.to = to;  }   public String getType() {  return type;  }   public void setType(String type) {  this.type = type;  }   public String getInfo() {  return info;  }   public void setInfo(String info) {  this.info = info;  }   public Message() {  }   public Message(String from, String to, String type, String info) {  this.from = from;  this.to = to;  this.type = type;  this.info = info;  }   @Override  public String toString() {  return "Message{" +  "from='" + from + '\'' +  ", to='" + to + '\'' +  ", type=" + type +  ", info='" + info + '\'' +  '}';  } }  服务器端编码  package com.mina;  import org.apache.mina.core.filterchain.DefaultIoFilterChainBuilder; import org.apache.mina.filter.codec.ProtocolCodecFilter; import org.apache.mina.filter.codec.serialization.ObjectSerializationCodecFactory; import org.apache.mina.transport.socket.SocketAcceptor; import org.apache.mina.transport.socket.nio.NioSocketAcceptor;  import java.io.IOException; import java.net.InetSocketAddress;  public class Server {  public static void main(String[] args) {  //创建一个非阻塞的Server端Socket,用NIO  SocketAcceptor acceptor = new NioSocketAcceptor(); //创建接收数据的过滤器  DefaultIoFilterChainBuilder chain = acceptor.getFilterChain();  //设定过滤器以对象为单位读取数据  chain.addLast("objectFilter", new ProtocolCodecFilter(new ObjectSerializationCodecFactory()));  //设定服务器端的消息处理器:一个SampleMinaServerHandler对象  acceptor.setHandler(new SampleMinaServerHandler());  int bindPort = 9999;  try {  //绑定端口,启动服务器(不会阻塞，立即返回)  acceptor.bind(new InetSocketAddress(bindPort));  } catch (IOException e) {  e.printStackTrace();  }  System.*out*.println("Mina Server is Listing on:= " + bindPort);   } }  服务器端的消息处理器  package com.mina;  import org.apache.mina.core.service.IoHandlerAdapter; import org.apache.mina.core.session.IoSession;  */\*\*  \* 服务器端的消息处理器  \*/* public class SampleMinaServerHandler extends IoHandlerAdapter {  //一次会话被打开  @Override  public void sessionOpened(IoSession session) throws Exception {  super.sessionOpened(session);  //打印客户端地址  System.*out*.println("welcome client" + session.getRemoteAddress());  }  //会话关闭  @Override  public void sessionClosed(IoSession session) throws Exception {  super.sessionClosed(session);  System.*out*.println("client close");  }  //接受消息  @Override  public void messageReceived(IoSession session, Object message) throws Exception {  super.messageReceived(session, message);  Message msg = (Message) message;  System.*out*.println("收到客户机发来的消息: "+msg);  //向客户端发送消息对象  session.write(msg);  } }  客户端代码  package com.mina;  import org.apache.mina.core.filterchain.DefaultIoFilterChainBuilder; import org.apache.mina.core.future.ConnectFuture; import org.apache.mina.filter.codec.ProtocolCodecFilter; import org.apache.mina.filter.codec.serialization.ObjectSerializationCodecFactory; import org.apache.mina.transport.socket.nio.NioSocketConnector;  import java.net.InetSocketAddress; import java.util.Scanner;  public class Client {  public static void main(String[] args) {  NioSocketConnector connector = new NioSocketConnector(); // 创建接收数据的过滤器  DefaultIoFilterChainBuilder chain = connector.getFilterChain();  //设定过滤器以对象为单位读取数据  chain.addLast("objectFilter", new ProtocolCodecFilter(new ObjectSerializationCodecFactory()));  //设定服务器端的消息处理器:一个 SamplMinaServerHandler 对象  connector.setHandler(new SampleMinaClientHandler());  connector.setConnectTimeoutMillis(10000);// Set connect timeout.  //连接到服务器:  ConnectFuture cf = connector.connect(new InetSocketAddress("localhost", 9999));  cf.awaitUninterruptibly(); // 等待连接成功  Scanner sc = new Scanner(System.*in*);  while (true) {  //以对象的方式传输数据  Message msg = new Message();  System.*out*.println("from：");  msg.setFrom(sc.nextLine());  System.*out*.println("to：");  msg.setTo(sc.nextLine());  System.*out*.println("info：");  msg.setInfo(sc.nextLine());  msg.setType("send");  //发送消息  cf.getSession().write(msg);  }  //等待服务器连接关闭，结束长连接  //cf.getSession().getCloseFuture().awaitUninterruptibly();  //connector.dispose(); //关闭连接  } }  客户端消息处理器  package com.mina;  import org.apache.mina.core.service.IoHandlerAdapter; import org.apache.mina.core.session.IoSession;  */\*\*  \* 客户端的消息处理器  \*/* public class SampleMinaClientHandler extends IoHandlerAdapter {  //当一个客端端连结进入时  @Override  public void sessionOpened(IoSession session) throws Exception {  super.sessionOpened(session);  System.*out*.println("incomming client :"+session.getRemoteAddress());   }  //当一个客户端关闭时  @Override  public void sessionClosed(IoSession session) throws Exception {  super.sessionClosed(session);  System.*out*.println("one Clinet Disconnect !");  }  //当客户端发送的消息到达时:  @Override  public void messageReceived(IoSession session, Object message) throws Exception {  super.messageReceived(session, message);  Message msg = (Message) message;  System.*out*.println("服务器发来的收到消息: "+msg);  } }  启动服务端Run 再启动客户端Run测试 |

# 第十一章 反射与内省

## 什么是反射

反射： Reflection



## Class类

Class类是一切的反射根源。

### 1、Class类表示什么？

很多的人——可以定义一个Person类（有年龄，性别，姓名等）

很多的车——可以定义一个Car类（有发动机，颜色，车轮等）

很多的类——Class类（类名，构造方法，属性，方法）/

### 2、得到Class类的对象有三种方式：

第一种形式：Object类中的getClass()方法

第二种形式：类.class

第三种形式：通过Class类的forName方法

|  |
| --- |
| */\*\*  \* 获取Class对象的三种形式  \*/* @Test public void test() {  //第一种方式：通过对象的getClass()方法  Dog dog = new Dog("wangwang", 4, "黄色");  Class aClass = dog.getClass();  //第二种方式：直接通过 类.class  Class dogClass = Dog.class;  //第三种方式：通过Class.forName方法  try {  Class fClass = Class.*forName*("com.usClass.Dog");  } catch (ClassNotFoundException e) {  e.printStackTrace();  } } |

### 3、使用Class类进行对象的实例化操作

调用无参构造进行实例化

public T newInstance() throws InstantiationException,IllegalAccessException

调用有参构造进行实例化

public Constructor<?>[] getConstructors() throws SecurityException

|  |
| --- |
| */\*\*  \* 第一种：通过反射来实例化对象  \*/* @Test public void test2() {  Class dogClass = Dog.class;  try {  //通过class对象实例化类对象，调用了默认无参的构造方法  Dog dog = (Dog) dogClass.newInstance();  } catch (InstantiationException e) {  e.printStackTrace();  } catch (IllegalAccessException e) {  e.printStackTrace();  }  }  */\*\*  \* 第二种：获取所有构造方法  \*/* @Test public void test3() {  Class dogClass = Dog.class;  //获取所有构造方法  Constructor[] constructors = dogClass.getConstructors();  for (int i = 0; i < constructors.length; i++) {  System.*out*.println(constructors[i].getName());//获取构造方法的名称  System.*out*.println(constructors[i].getParameterCount());//获取构造方法的参数数量  }  //根据参数类型获取指定的构造方法  try {  Constructor<Dog> constructor = dogClass.getConstructor(String.class, int.class, String.class);  Dog dog = constructor.newInstance("小白", 12, "白色");  } catch (SecurityException e) {  e.printStackTrace();  } catch (NoSuchMethodException e) {  e.printStackTrace();  } catch (InstantiationException e) {  e.printStackTrace();  } catch (IllegalAccessException e) {  e.printStackTrace();  } catch (InvocationTargetException e) {  e.printStackTrace();  } } |

### 4、取得类所在的包

public Package getPackage() //得到一个类所在的包

public String getName() //得到名字

|  |
| --- |
| */\*\*  \* 获取类所在的包  \*/* @Test public void test5() {  Class dogClass = Dog.class;  //获取类的包名  Package aPackage = dogClass.getPackage();  System.*out*.println(aPackage.getName()); } |

### 5、取得一个类中的全部方法

public Method[] getMethods()

public int getModifiers() //Modifier.*toString*(mod); // 还原修饰符

public Class<?> getReturnType()

public Class<?>[] getParameterTypes()

public Class<?>[] getExceptionTypes()

public static String toString(int mod)

### 6、取得一个类中的全部属性

public Field[] getFields()

public Field[] getDeclaredFields()

public Class<?> getType()

public int getModifiers()

public String getName()

|  |
| --- |
| */\*\*  \* 获取类中的所有属性  \*/* @Test public void test4() {  Class dogClass = Dog.class;  //getFields方法获非私有的属性  Field[] fields = dogClass.getFields();  //getDeclaredFields方法获取所有属性包括私有属性  Field[] declaredFields = dogClass.getDeclaredFields();  for (int i = 0; i < declaredFields.length; i++) {  int modifiers = declaredFields[i].getModifiers();  //获取属性的修饰符 public/private  System.*out*.println(Modifier.*toString*(modifiers));  //获取属性的数据类型 int/String  System.*out*.println(declaredFields[i].getType());  //获取属性的名称 name/age  System.*out*.println(declaredFields[i].getName());  } } |

### 7、调用类中的方法

调用类中的方法，传入实例化对象，以及具体的参数内容

public Object invoke(Object obj,Object... args)

|  |
| --- |
| */\*\*  \* 调用类中的方法  \*/* @Test public void test5() {  Dog dog = new Dog("小白", 5, "白色");  Class dogClass = Dog.class;  //获取公共的方法，包括继承的公有方法  Method[] methods = dogClass.getMethods();  for (int i = 0; i < methods.length; i++) {  if (methods[i].getName().equals("toString")) {  try {  //调用此方法  methods[i].invoke(dog);  } catch (IllegalAccessException | InvocationTargetException e) {  e.printStackTrace();  }  }  }  //获取到本类中定义的所有方法（包含私有方法，不包含父类的方法）  Method[] declaredMethods = dogClass.getDeclaredMethods();  for (int i = 0; i < declaredMethods.length; i++) {  //set 为私有的方法  if (declaredMethods[i].getName().equals("set")) {  try {  //设置方法可以被访问  declaredMethods[i].setAccessible(true);  //调用此方法  declaredMethods[i].invoke(dog);  } catch (IllegalAccessException | InvocationTargetException e) {  e.printStackTrace();  }  }  }  } |

### 8、直接调用属性

取得属性

public Object get(Object obj)

//设置属性，等同于使用“=”完成操作

public void set(Object obj,Object value)

//让属性对外部可见

 public void setAccessible(boolean flag)

## 动态代理

所谓动态代理，即通过代理类：Proxy的代理，接口和实现类之间可以不直接发生联系，而可以在运行期（Runtime）实现动态关联。

java动态代理主要是使用java.lang.reflect包中的两个类。

**InvocationHandler类**

public Object invoke(Object obj,Method method,Object[] obs)

其中第一个参数 obj 指的是代理类，method是被代理的方法，obs是指被代理的方法的参数组。此方法由代理类来实现。

**Proxy类**

protected Proxy(InvocationHandler h);

static Class getProxyClass(ClassLoader loader,Class[] interfaces);

static Object newProxyInstance(ClassLoader loader,Class[]interfaces,InvocationHandler h);

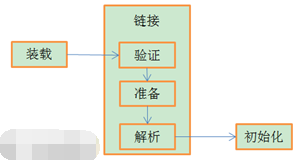
动态代理其实是在运行时生成class，所以，我们必须提供一组interface，然后告诉他class已经实现了这些interface，而且在生成Proxy的时候，必须给他提供一个handler，让他来接管实际的工作。

|  |
| --- |
| 创建接口  package com.proxy;  public interface Subject {   public void shopping(); }  实现接口  package com.proxy;  public class Person implements Subject{  @Override  public void shopping() {  System.*out*.println("付款，买到心仪的衣服");  } }  创建代理类  package com.proxy;  import java.lang.reflect.InvocationHandler; import java.lang.reflect.Method; import java.lang.reflect.Proxy;  */\*\*  \* 用于动态生成一个代理对象  \*/* public class CreateProxy implements InvocationHandler {  private Object target; //被代理的对象   //用于创建代理对象的方法  public Object create (Object target) {  this.target = target;  Object proxy = Proxy.*newProxyInstance*(target.getClass().getClassLoader(),  target.getClass().getInterfaces(), this);  return proxy;  }   */\*\*  \* 代理对象要执行的方法  \** ***@param*** *proxy 代理类对象  \** ***@param*** *method 本代理对象的方法  \** ***@param*** *args 被代理对象方法的参数  \** ***@return*** *\** ***@throws*** *Throwable  \*/* @Override  public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  System.*out*.println("上海外寻找客户需要的产品.....");  System.*out*.println("给客户确认物品");  method.invoke(target, args);  System.*out*.println("完成本次购物");  return null;  } }  测试类  package com.proxy;  import org.junit.Test;  public class TestDemo {    @Test  public void proxyTest () {  CreateProxy cp = new CreateProxy();  Subject person = new Person();  Subject proxy = (Subject) cp.create(person);  proxy.shopping(); //invoke  } } |

## 四、类加载器原理分析

**1、类的加载过程**

JVM将类加载过程分为三个步骤：装载（Load），链接（Link）和初始化(Initialize)链接又分为三个步骤，如下图所示：



1) 装载：查找并加载类的二进制数据；

2)链接：

验证：确保被加载类的正确性；

准备：为类的静态变量分配内存，并将其初始化为默认值；

解析：把类中的符号引用转换为直接引用；

3)初始化：为类的静态变量赋予正确的初始值；

**2、类的初始化，类什么时候才被初始化：**

1）创建类的实例，也就是new一个对象

2）访问某个类或接口的静态变量，或者对该静态变量赋值

3）调用类的静态方法

4）反射（Class.forName("com.vince.Dog")）

5）初始化一个类的子类（会首先初始化子类的父类）

6）JVM启动时标明的启动类，即文件名和类名相同的那个类

**3、类的加载：**

指的是将类的.class文件中的二进制数据读入到内存中，将其放在运行时数据区的方法区内，然后在堆区创建一个这个类的Java.lang.Class对象，用来封装类在方法区类的对象。

## JavaBean的概念

**什么是 JavaBean？**

Bean理解为组件意思， JavaBean就是Java组件，在广泛的理解就是一个类，对于组件来说，关键在于要具有“能够被IDE构建工具侦测其属性和事件”的能力，通常在Java中。

**一个JavaBean要具备这样的命名规则：**

**1、**对于一个名称为xxx的属性，通常你要写两个方法：getXxx()和setXxx()。任何浏览这些方法的工具，都会把get或set后面的第一个字母自动转换为小写。

**2、**对于布尔型属性，可以使用以上get和set的方式，不过也可以把get替换成is。

**3、**Bean的普通方法不必遵循以上的命名规则，不过它们必须是public的。

**4、**对于事件，要使用Swing中处理监听器的方式。如addWindowListener，removeWindowListener

|  |
| --- |
| package com.javaBean;  public class Emp {  private String name;  private int age;  private int salary;   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 int getSalary() {  return salary;  }   public void setSalary(int salary) {  this.salary = salary;  } } |

BeanUtils工具类：http://apache.org/

需要jar



|  |
| --- |
| package com.javaBean;  import org.apache.commons.beanutils.BeanUtils; import org.junit.Test; import java.lang.reflect.InvocationTargetException;  public class BeanTest {   @Test  public void test () {  //从客户端获取到的数据是这样的  String name = "bin";  String age = "18";  String salary = "20000";  Emp emp = new Emp();  //使用BeanUtils给emp对象赋值，优点在于不需要类型转换  try {  BeanUtils.*setProperty*(emp, "name", name);  BeanUtils.*setProperty*(emp, "age", age);  BeanUtils.*setProperty*(emp, "salary", salary);  System.*out*.println(emp);  } catch (IllegalAccessException e) {  e.printStackTrace();  } catch (InvocationTargetException e) {  e.printStackTrace();  }  } } |

## 内省基本概念

内省(Introspector)是Java 语言对 Bean 类属性、事件的一种缺省处理方法。例如类 A 中有属性 name, 那我们可以通过 getName,setName 来得到其值或者设置新的值。

通过 getName/setName 来访问 name 属性，这就是默认的规则。

Java 中提供了一套 API 用来访问某个属性的 getter/setter 方法，通过这些 API 可以使你不需要了解这个规则，这些 API 存放于包 java.beans 中，一般的做法是通过类 Introspector 的 getBeanInfo方法 来获取某个对象的 BeanInfo 信息，然后通过 BeanInfo 来获取属性的描述器(PropertyDescriptor)，通过这个属性描述器就可以获取某个属性对应的 getter/setter 方法，然后我们就可以通过反射机制来调用这些方法。

## Introspector相关API

1、Introspector类：

Introspector 类为通过工具学习有关受目标 Java Bean 支持的属性、事件和方法的知识提供了一个标准方法。

static BeanInfo getBeanInfo(Class<?> beanClass)   
          在 Java Bean 上进行内省，了解其所有属性、公开的方法和事件。

2、BeanInfo类：

该类实现此 BeanInfo 接口并提供有关其 bean 的方法、属性、事件等显式信息。

MethodDescriptor[] getMethodDescriptors()   
          获得 beans MethodDescriptor。

PropertyDescriptor[] getPropertyDescriptors()   
          获得 beans PropertyDescriptor。

Properties 属性文件工具类的使用

3、PropertyDescriptor 类：

PropertyDescriptor 描述 Java Bean 通过一对存储器方法导出的一个属性。

 Method getReadMethod()   
          获得应该用于读取属性值的方法。

Method getWriteMethod()   
          获得应该用于写入属性值的方法。

4、MethodDescriptor 类：

MethodDescriptor 描述了一种特殊方法，

即 Java Bean 支持从其他组件对其进行外部访问。

 Method getMethod()   
          获得此 MethodDescriptor 封装的方法。

|  |
| --- |
| Config javabean  package com.introspector;  public class Config {  private String username;  private String password;  private String url;   public Config() {  }   public Config(String username, String password, String url) {  this.username = username;  this.password = password;  this.url = url;  }   public String getUsername() {  return username;  }   public void setUsername(String username) {  this.username = username;  }   public String getPassword() {  return password;  }   public void setPassword(String password) {  this.password = password;  }   public String getUrl() {  return url;  }   public void setUrl(String url) {  this.url = url;  }   @Override  public String toString() {  return "Config{" +  "username='" + username + '\'' +  ", password='" + password + '\'' +  ", url='" + url + '\'' +  '}';  } }  config.properties  bean.name=com.introspector.Config bean.username=admin bean.password=123 bean.url=http://www.baidu.com  BeanFactory.java  package com.introspector;  import java.beans.BeanInfo; import java.beans.IntrospectionException; import java.beans.Introspector; import java.beans.PropertyDescriptor; import java.io.IOException; import java.io.InputStream; import java.lang.reflect.InvocationTargetException; import java.lang.reflect.Method; import java.util.Properties;  */\*\*  \* 通过内省的API来装配一个Bean对象，Bean对象的值是通过配置文件中来获取  \* 目的是为了提高维护性  \*/* public class BeanFactory {  private static Properties *pro* = new Properties();  static {  try {  InputStream is = Thread.*currentThread*().getContextClassLoader()  .getResourceAsStream("com/introspector/config.properties");  *pro*.load(is);  } catch (IOException e) {  e.printStackTrace();  }  }  */\*\*  \* 获取一个Bean  \*/* public static Object getBean (String name) {  Object bean = null;  try {  String beanName = *pro*.getProperty(name);  Class<?> aClass = Class.*forName*(beanName);  bean = aClass.newInstance();  //通过类信息获取javaBean的描述信息  BeanInfo beanInfo = Introspector.*getBeanInfo*(aClass);  //通过javaBean的描述信息，获取该类的所有属性描述器  PropertyDescriptor[] propertyDescriptors = beanInfo.getPropertyDescriptors();  for (int i = 0; i < propertyDescriptors.length; i++) {  String propertyName = propertyDescriptors[i].getName();  Method writeMethod = propertyDescriptors[i].getWriteMethod();  if ("username".equals(propertyName)) {  //调用属性的set方法  writeMethod.invoke(bean, *pro*.getProperty("bean.username"));  } else if ("password".equals(propertyName)) {  writeMethod.invoke(bean, *pro*.getProperty("bean.password"));  } else if ("url".equals(propertyName)) {  writeMethod.invoke(bean, *pro*.getProperty("bean.url"));  }  }  } catch (ClassNotFoundException e) {  e.printStackTrace();  } catch (IntrospectionException e) {  e.printStackTrace();  } catch (IllegalAccessException e) {  e.printStackTrace();  } catch (InstantiationException e) {  e.printStackTrace();  } catch (InvocationTargetException e) {  e.printStackTrace();  }  return bean;  } }  测试  package com.introspector;  import org.junit.Test;  public class BeanTest {   @Test  public void getBeanTest () {  Config config = (Config)BeanFactory.*getBean*("bean.name");  System.*out*.println(config);  } } |

## 理解可配置的AOP框架

**补充知识：**

1、AOP的概念：Aspect Oriented Programming（面向切面编程）

2、可配置 AOP框架实现

**AOP使用场景**

**AOP用来封装横切关注点，具体可以在下面的场景中使用:**

权限

缓存

错误处理

调试

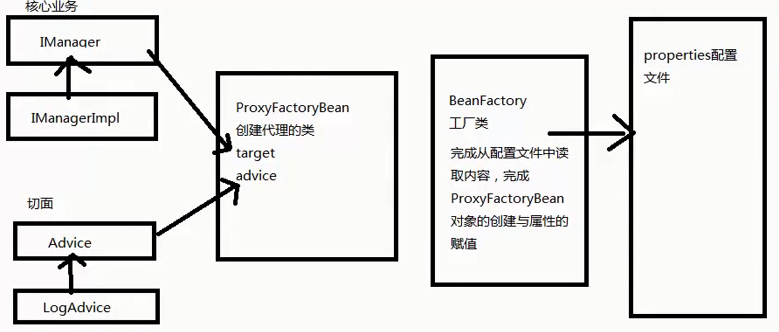
记录跟踪

持久化

同步

事务

等等。。



|  |
| --- |
| Manager接口  package com.aop;  public interface IManager {  public void add(String item); }  ManagerImpl—Manager接口的实现类  package com.aop;  import java.util.ArrayList; import java.util.List;  public class IManagerImpl implements IManager {  private List<String> list = new ArrayList<>();  @Override  public void add(String item) {  list.add(item);  System.*out*.println(item);  } }  Advice切面接口  package com.aop;  public interface Advice {   public void beforeAdvice();  public void afterAdvice(); }  LogAdvice –Advice切面接口的实现类  package com.aop;  */\*\*  \* 切面实现类  \*/* public class LogAdvice implements Advice {   @Override  public void beforeAdvice() {  System.*out*.println("start time:" + System.*currentTimeMillis*());  }  @Override  public void afterAdvice() {  System.*out*.println("end time:" + System.*currentTimeMillis*());  } }  beens.properties  bean.target=com.aop.IManagerImpl bean.advice=com.aop.LogAdvice bean=com.aop.ProxyFactorBean  ProxyFactorBean 代理工厂类  package com.aop;  import java.lang.reflect.InvocationHandler; import java.lang.reflect.Method; import java.lang.reflect.Proxy;  public class ProxyFactorBean implements InvocationHandler {  private Object target; //被代理的对象  private Advice advice;   public Object getProxy () {  Object proxy = Proxy.*newProxyInstance*(target.getClass().getClassLoader(),  target.getClass().getInterfaces(), this);  return proxy;  }   @Override  public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  advice.beforeAdvice();  Object obj = method.invoke(target, args);  advice.afterAdvice();  return obj;  }   public Object getTarget() {  return target;  }   public void setTarget(Object target) {  this.target = target;  }   public Advice getAdvice() {  return advice;  }   public void setAdvice(Advice advice) {  this.advice = advice;  } }  工厂类  package com.aop;  import java.beans.BeanInfo; import java.beans.IntrospectionException; import java.beans.Introspector; import java.beans.PropertyDescriptor; import java.io.IOException; import java.io.InputStream; import java.lang.reflect.InvocationTargetException; import java.lang.reflect.Method; import java.util.Properties;  */\*\*  \* AOP框架的简单实现  \*/* public class BeanFactory {   Properties pro = new Properties();   public BeanFactory(InputStream in) {  try {  pro.load(in);  } catch (IOException e) {  e.printStackTrace();  }  }  */\*\*  \* 获取一个bean实例对象  \*/* public Object getBean (String name) {  Object bean = null;  try {  Class<?> aClass = Class.*forName*(pro.getProperty(name));  bean = aClass.newInstance();  Object target = Class.*forName*(pro.getProperty(name + ".target")).newInstance();  Object advice = Class.*forName*(pro.getProperty(name + ".advice")).newInstance();  //通过类信息获取javaBean的描述信息  BeanInfo beanInfo = Introspector.*getBeanInfo*(aClass);  //通过javaBean的描述信息，获取该类的所有属性描述器  PropertyDescriptor[] propertyDescriptors = beanInfo.getPropertyDescriptors();  for (PropertyDescriptor pd : propertyDescriptors) {  String propertyName = pd.getName();  Method writeMethod = pd.getWriteMethod();  if ("target".equals(propertyName)) {  //调用属性的set方法  writeMethod.invoke(bean, target);  } else if ("advice".equals(propertyName)) {  writeMethod.invoke(bean, advice);  }  }  } catch (InstantiationException e) {  e.printStackTrace();  } catch (IllegalAccessException e) {  e.printStackTrace();  } catch (ClassNotFoundException e) {  e.printStackTrace();  } catch (IntrospectionException e) {  e.printStackTrace();  } catch (InvocationTargetException e) {  e.printStackTrace();  }  return bean;  } }  测试  package com.aop;  import org.junit.Test; import java.io.InputStream;  public class AOPTest {   @Test  public void test () {  //1、读取配置文件  InputStream is = Thread.*currentThread*().getContextClassLoader()  .getResourceAsStream("com/aop/beans.properties");  //2、创建Bean的工厂对象  BeanFactory beanFactory = new BeanFactory(is);  //3、获取代理对象  ProxyFactorBean proxyFactorBean = (ProxyFactorBean)beanFactory.getBean("bean");  IManager iManager = (IManager)proxyFactorBean.getProxy();  iManager.add("我是一只猫");  } } |

## 九、单例模式优化

1、使用同步保正线程安全 synchronized

2、使用volatile关键字

volatile提醒编译器它后面所定义的变量随时都有可能改变，因此编译后的程序每次需要存储或读取这个变量的时候，都会直接从变量地址中读取数据。如果没有volatile关键字，则编译器可能优化读取和存储，可能暂时使用寄存器中的值，如果这个变量由别的程序更新了的话，将出现不一致的现象。

3、防止反射调用私有构造方法

4、让单例类序例化安全

|  |
| --- |
| package com.vince;  import java.io.Serializable;  */\*\*  \* 1、多线程访问的安全问题 synchronized  \* 2、加上volatile关键字保证变量的一致性  \* 3、防止反射调用私有构造方法  \* 4、让单例类序例化安全  \*/* //4 public class Singleton implements Serializable {  //2  public volatile static Singleton *singleton* = null;   public Singleton() {  //3  if (*singleton* != null) {  throw new RuntimeException("此类对象为单例模式，已经被实例化了");  }  }   public static Singleton getInstance() {  //1  if (*singleton* == null) {  synchronized (Singleton.class) {  if (*singleton* == null) {  *singleton* = new Singleton();  }  }  }  return *singleton*;  } } |

# 第十二章 泛型

## 一、为什么需要泛型

public void test1 () {  
 //List中可以添加任意类型，因为参数是Object，  
 List list = new ArrayList();  
 list.add("vnice");  
 list.add(123);  
 list.add(new Object());  
  
 for (int i = 0; i < list.size(); i++) {  
 //如果我们不能确定集合中的元素类型，那么我们需要在处理元素时  
 //要判断元素的类型，才能做相应操作  
 }  
}

**以上操作主要存在两个问题：**

1.当我们将一个对象放入集合中，集合不会记住此对象的类型，当再次从集合中取出此对象时，改对象的编译类型变成了Object类型，但其运行时类型依然为其本身类型。

2.因此，//取出集合元素时需要人为的强制类型转化到具体的目标类型，且很容易出现“java.lang.ClassCastException”异常。

那么有没有什么办法可以使集合能够记住集合内元素各类型，且能够达到只要编译时不出现问题，运行时就不会出现“java.lang.ClassCastException”异常呢？答案就是使用泛型。

## 二、什么是泛型

JDK 1.5之后出现了新的技术 —— 泛型（Generic），此技术的最大特点是类中的属性的类型可以由外部决定。泛型，即“参数化类型”。一提到参数，最熟悉的就是定义方法时有形参，然后调用此方法时传递实参。那么参数化类型怎么理解呢？顾名思义，就是将类型由原来的具体的类型参数化，类似于方法中的变量参数，此时类型也定义成参数形式（可以称之为类型形参），然后在使用/调用时传入具体的类型（类型实参）。

List<String> list = new ArrayList<String>();//... = new ArrayList<>(); //... = new ArrayList();

list.add("vince");

list.add("bin");

//list.add(100); // 提示编译错误

for (int i = 0; i < list.size(); i++) {

String name = list.get(i); // 2

System.out.println("name:" + name);

}

## 三、自定义泛型接口、泛型类

泛型类和方法定义：

package com.vice;  
*/\*\*  
 \* 泛型类  
 \* T:参数化类型，在实际使用时才会指定具体的类型  
 \* 泛型只作用于编译期检查，在编译后，会被擦除  
 \*/*public class Node<T> {  
 private T data;  
 public Node () {}  
 public Node (T data) {  
 this.data = data;  
 }  
  
 public T getData() {  
 return data;  
 }  
  
 public void setData(T data) {  
 this.data = data;  
 }  
}

package com.vice;  
  
*/\*\*  
 \* 泛型接口  
 \** ***@param*** <*T*>  
 *\*/*public interface Shopping<T> {  
 public void shopping(T name);  
}

在泛型接口、泛型类和泛型方法的定义过程中，我们常见的如T、E、K、V等形式的参数常用于表示泛型形参，由于接收来自外部使用时候传入的类型实参，从编码的角度也称为参数化类型了。

泛型只是作用于代码编译阶段，在编译过程中，对于正确检验泛型结果后，会将泛型的相关信息擦除，也就是说，成功编译过后的class文件中是不包含任何泛型信息的。泛型信息不会进入到运行时阶段。

## 四、通配符

问题：

public void test2 () {  
 Node<Number> c1 = new Node<Number>();  
 Node<Integer> c2 = new Node<Integer>();  
 *getData*(c1); //可以执行  
 //getData(c2); //报错 无法执行  
}  
public static void getData(Node<Number> data) {  
 System.*out*.println("data :" + data.getData());  
}

**此时，我们可以使用通配符来解决**

“?”表示的是可以接收任意的泛型类型，但是只是接收输出，并不能修改。

public void test2 () {  
 Node<Number> c1 = new Node<Number>();  
 Node<Integer> c2 = new Node<Integer>();  
 *getData*(c1); //可以执行  
 *getData*(c2); //可以执行  
}  
public static void getData(Node<?> data) {  
 System.*out*.println("data :" + data.getData());  
}

泛型上限就指一个的操作泛型最大的操作父类，例如，现在最大的上限设置成“Number”类型，那么此时，所能够接收到的类型只能是Number及其子类（Integer）。

泛型的上限通过以下的语法完成：

? extends 类

*/\*\*  
 \* 只能是Number类及其子类调用此方法   
 \* 如：  
 \* 1、Node<Number> c1 = new Node<Number>();  
 \* getData(c1);  
 \* 2、Node<Integer> c2 = new Node<Integer>();  
 \* getData(c2);  
 \** ***@param*** *data  
 \*/*public static void getData(Node<? extends Number> data) {  
 System.*out*.println("data :" + data.getData());  
}

泛型的下限指的是只能设置其具体的类或者父类。设置的语法如下：

? super 类

## 五、泛型方法

泛型除了在类中定义之外，还可以在方法上定义，而且在方法上使用泛型，此方法所在的类不一定是泛型的操作类。

**定义一个方法，实现任意类型数组中两个位置值的调换**

@Test

public void test4 () {  
 String[] arrays = {"vince", "jack", "Tom", "liLi"};  
 String[] strs = *func*(arrays, 0, 1);  
 System.*out*.println(Arrays.*toString*(strs));  
}  
public static <T> T[] func(T[] array,int i,int t){  
 T temp = array[i];  
 array[i] = array[t];  
 array[t] = temp;  
 return array;  
}

## 六、泛型的嵌套使用

在使用集合Map的时候，我们可以这样遍历：

Set<Entry<Integer, String>> entrys = map.entrySet();

# 第十三章 正则表达式

## 一、认识正则

通过一个程序来简单了解一下正则有那些用处：

**问题：判断一个字符串是否由数字组成**

*/\*\*  
 \* 没有使用正则表达式来检查字符串是否由数字组成  
 \*/*public void test1 () {  
 String s = "253801";  
 char[] chars = s.toCharArray();  
 boolean flag = true;  
 for (int i = 0; i < chars.length; i++) {  
 if (chars[i]<'0' || chars[i]>'9') {  
 flag = false;  
 break;  
 }  
 }  
 if (flag) {  
 System.*out*.println("是由数字组成");  
 } else {  
 System.*out*.println("不是由数字组成");  
 }  
}

## 二、正则表达式

正则表达式（Regular Expression）

正则表达式使用单个字符串来描述、匹配一系列符合某个句法规则的字符串。

则表达式通常被用来检索、替换那些符合某个模式的文本

java.util.regex包中提供以下两个类对正则表达式的支持：

Matcher 类

通过解释 Pattern 对 character sequence 执行匹配操作的引擎。

Pattern 类

正则表达式的编译表示形式。

|  |
| --- |
| 字符  x 字符 x  \\ 反斜线字符  \0n 带有八进制值 0 的字符 n (0 <= n <= 7)  \0nn 带有八进制值 0 的字符 nn (0 <= n <= 7)  \0mnn 带有八进制值 0 的字符 mnn（0 <= m <= 3、0 <= n <= 7）  \xhh 带有十六进制值 0x 的字符 hh  \uhhhh 带有十六进制值 0x 的字符 hhhh  \t 制表符 ('\u0009')  \n 新行（换行）符 ('\u000A')  \r 回车符 ('\u000D')  \f 换页符 ('\u000C')  \a 报警 (bell) 符 ('\u0007')  \e 转义符 ('\u001B')  \cx 对应于 x 的控制符    字符类  [abc] a、b 或 c（简单类）  [^abc] 任何字符，除了 a、b 或 c（否定）  [a-zA-Z] a 到 z 或 A 到 Z，两头的字母包括在内（范围）  [a-d[m-p]] a 到 d 或 m 到 p：[a-dm-p]（并集）  [a-z&&[def]] d、e 或 f（交集）  [a-z&&[^bc]] a 到 z，除了 b 和 c：[ad-z]（减去）  [a-z&&[^m-p]] a 到 z，而非 m 到 p：[a-lq-z]（减去）    预定义字符类  . 任何字符（与行结束符可能匹配也可能不匹配）  \d 数字：[0-9]  \D 非数字： [^0-9]  \s 空白字符：[ \t\n\x0B\f\r]  \S 非空白字符：[^\s]  \w 单词字符：[a-zA-Z\_0-9]  \W 非单词字符：[^\w]    POSIX 字符类（仅 US-ASCII）  \p{Lower} 小写字母字符：[a-z]  \p{Upper} 大写字母字符：[A-Z]  \p{ASCII} 所有 ASCII：[\x00-\x7F]  \p{Alpha} 字母字符：[\p{Lower}\p{Upper}]  \p{Digit} 十进制数字：[0-9]  \p{Alnum} 字母数字字符：[\p{Alpha}\p{Digit}]  \p{Punct} 标点符号：!"#$%&'()\*+,-./:;<=>?@[\]^\_`{|}~  \p{Graph} 可见字符：[\p{Alnum}\p{Punct}]  \p{Print} 可打印字符：[\p{Graph}\x20]  \p{Blank} 空格或制表符：[ \t]  \p{Cntrl} 控制字符：[\x00-\x1F\x7F]  \p{XDigit} 十六进制数字：[0-9a-fA-F]  \p{Space} 空白字符：[ \t\n\x0B\f\r]    java.lang.Character 类（简单的 java 字符类型）  \p{javaLowerCase} 等效于 java.lang.Character.isLowerCase()  \p{javaUpperCase} 等效于 java.lang.Character.isUpperCase()  \p{javaWhitespace} 等效于 java.lang.Character.isWhitespace()  \p{javaMirrored} 等效于 java.lang.Character.isMirrored()    Unicode 块和类别的类  \p{InGreek} Greek 块（简单块）中的字符  \p{Lu} 大写字母（简单类别）  \p{Sc} 货币符号  \P{InGreek} 所有字符，Greek 块中的除外（否定）  [\p{L}&&[^\p{Lu}]] 所有字母，大写字母除外（减去）    边界匹配器  ^ 行的开头  $ 行的结尾  \b 单词边界  \B 非单词边界  \A 输入的开头  \G 上一个匹配的结尾  \Z 输入的结尾，仅用于最后的结束符（如果有的话）  \z 输入的结尾    Greedy 数量词  X? X，一次或一次也没有  X\* X，零次或多次  X+ X，一次或多次  X{n} X，恰好 n 次  X{n,} X，至少 n 次  X{n,m} X，至少 n 次，但是不超过 m 次    Reluctant 数量词  X?? X，一次或一次也没有  X\*? X，零次或多次  X+? X，一次或多次  X{n}? X，恰好 n 次  X{n,}? X，至少 n 次  X{n,m}? X，至少 n 次，但是不超过 m 次    Possessive 数量词  X?+ X，一次或一次也没有  X\*+ X，零次或多次  X++ X，一次或多次  X{n}+ X，恰好 n 次  X{n,}+ X，至少 n 次  X{n,m}+ X，至少 n 次，但是不超过 m 次    Logical 运算符  XY X 后跟 Y  X|Y X 或 Y  (X) X，作为捕获组    Back 引用  \n 任何匹配的 nth 捕获组    引用  \ Nothing，但是引用以下字符  \Q Nothing，但是引用所有字符，直到 \E  \E Nothing，但是结束从 \Q 开始的引用    特殊构造（非捕获）  (?:X) X，作为非捕获组  (?idmsux-idmsux) Nothing，但是将匹配标志i d m s u x on - off  (?idmsux-idmsux:X) X，作为带有给定标志 i d m s u x on - off  (?=X) X，通过零宽度的正 lookahead  (?!X) X，通过零宽度的负 lookahead  (?<=X) X，通过零宽度的正 lookbehind  (?<!X) X，通过零宽度的负 lookbehind  (?>X) X，作为独立的非捕获组 |

## 三、Pattern类

public final class Pattern extends Objectimplements Serializable

正则表达式的编译表示形式。指定为字符串的正则表达式必须首先被编译为此类的实例。

典型的调用顺序是

public void test2 () {  
 //创建一个匹配模板  
 Pattern p = Pattern.*compile*("a\*b");  
 Matcher m = p.matcher("aaaaab");  
 boolean b = m.matches(); //匹配  
}

## 四、Matcher类

public final class Matcher extends Object implements MatchResult

Matcher类的主要功能是用于进行正则的匹配，通过Pattern类中定义完的正则，再使用Matcher类进行验证或者替换。

常用方法：

boolean matches() 尝试将整个区域与模式匹配。

String replaceAll(String replacement) 替换模式与给定替换字符串相匹配的输入序列的每个子序列。

String replaceFirst(String replacement) 替换模式与给定替换字符串匹配的输入序列的第一个子序列。

## 五、String类对正则的支持

在JDK 1.4之后加入了正则，随后又更新了String的操作类，因为在使用正则中，所有的内容通过字符串表示的比较多。在String类中有以下的方法可以完成对正则的支持：

boolean **matches**(String regex) 告知此字符串是否匹配给定的正则表达式。

String **replaceAll**(String regex, String replacement) 使用给定的 replacement 替换此字符串所有匹配给定的正则表达式的子字符串。

String **replaceFirst**(String regex, String replacement) 使用给定的 replacement 替换此字符串匹配给定的正则表达式的第一个子字符串。

String[] **split**(String regex) 根据给定正则表达式的匹配拆分此字符串。

## 六、示例

public void test3 () {  
 //匹配电话号码  
 String phoneNumber = "010-38389438";  
 boolean b = phoneNumber.matches("\\d{3,4}-\\{7,8}");  
 //匹配手机号  
 String phone = "13876543839";  
 boolean b1 = phone.matches("[1][3-9]\\d{9}");  
 //验证用户名，字母开头的数字、字母或下划线的组合  
 String username = "abc1314";  
 boolean b2 = phone.matches("[a-zA-Z]+[\\w|\_]\*");  
 //验证IP地址  
 String ip = "20.10.20.123";  
 boolean b3 = phone.matches("\\d{1,3}.\\d{1,3}.\\d{1,3}.\\d{1,3}");  
 //验证网址  
 String addr = "http://www.baidu.com";  
 boolean b4 = phone.matches("http://\\w+.\\w+.\\S\*");  
 //验证年龄  
 String age = "18";  
 boolean b5 = phone.matches("\\d{1,3}");  
 //验证金额  
 String price = "19.8";  
 boolean b6 = phone.matches("\\d+.+\\d+");  
}

# 第十四章 枚举

## 一、认识枚举

枚举就是要让某个类型的变量的取值只能为若干个固定值中的一个，否则编译器就会报错，枚举可以让编译器在编译时就可以控制源程序赋给的非法值，使用普通变量的方式在开发阶段无法实现这一目标。

在JDK 1.5之后，使用关键字enum定义一种新的类型，称为枚举类型。

|  |
| --- |
| package com.vice;  public enum Color {  *RED*,*GREEN*,*BlUE*; }  package com.vice;  public class EnumDemo {   public static final int *RED* = 0x1;  public static final int *GREEN* = 0x2;  public static final int *BlUE* = 0x3;   */\*\*  \* 普通的方法  \*/* public int color;  public void test1 () {  //这里给color赋值可以为任意值  //目标color的值只能为RED、GREEN、BlUE中的一个  color = *RED*;  color = 4;  }   */\*\*  \* 使用枚举  \*/* public Color colorEnum;  public void test2 () {  //这里colorEnum的值只能为Color.RED、Color.GREEN、Color.BlUE  colorEnum = Color.*RED*;  } } |

## 二、Enum类与enum关键字

使用enum关键字定义的枚举类，实际上就相当于定义了一个类，此类继承了Enum类而已。

Enum类中定义了如下的方法：

|  |  |
| --- | --- |
| protected Enum(String name,int ordinal) | 此构造方法不能被外部直接调用，只能被其子类访问，此构造方法为自动调用。 |
| public final String name() | 枚举的名字 |
| public final int ordinal() | 枚举的序号 |

*/\*\*  
 \* 使用枚举  
 \*/*public Color colorEnum;  
public void test2 () {  
 colorEnum = Color.*RED*;  
 System.*out*.println(colorEnum); //=toString方法  
 System.*out*.println(colorEnum.name()); //名称  
 System.*out*.println(colorEnum.ordinal()); //序号 从0开始  
 Color[] vals = Color.*values*(); //获取所有  
}

## 三、集合对枚举的支持

在JDK 1.5之后，对于Set和Map接口而言又增加了两个新的子类：EnumSet、EnumMap两个类。

public void test3 () {  
 EnumSet<Color> set = EnumSet.*allOf*(Color.class);  
 EnumMap<Color, Object> map = new EnumMap<>(Color.class);  
 map.put(Color.*RED*, 0x1);  
}

## 四、带构造方法的枚举

public enum Color {  
 *RED*(10),*GREEN*(30),*BlUE*;  
 private int color;  
 private Color() {  
 System.*out*.println(color+"无参构造器");  
 }  
  
 private Color(int color){  
 this.color = color;  
 System.*out*.println(color+"有参构造器");  
 }  
}

## 五、让枚举实现接口

*/\*\*  
 \* 接口  
 \*/*public interface Info {  
 public String getColor();  
}

*/\*\*  
 \* 实现类  
 \*/*public enum Color implements Info {  
 *RED* {  
 public String getColor() {return "红色";}  
 },  
 *GREEN* {  
 public String getColor() {return "绿色";}  
 },  
 *BLUE* {  
 public String getColor() {return "蓝色";}  
 };  
  
}

## 六、在枚举中定义抽象方法

public enum Color {  
 *RED* {  
 public String getColor() {return "红色";}  
 },  
 *GREEN* {  
 public String getColor() {return "绿色";}  
 },  
 *BLUE* {  
 public String getColor() {return "蓝色";}  
 };  
 public abstract String getColor();  
}

## 七、枚举的单例模式

public enum Color {  
 *RED*;  
 public void method () {  
 System.*out*.println("method");  
 }  
}

@Test  
public void test4 () {  
 Color.*RED*.method();  
}

# 第十五章 注解

## 一、认识Annotation

JDK1.5开始，Java增加了对元数据（即类的组成单元数据）的支持，也就是（Annotation）注解，他是代码里做的特殊标记，这些标记可以在编译，类加载，运行时在不改变原有逻辑的情况下，被读取，并执行相应的处理，通过使用Annotation，程序员可以在源文件中嵌入一些补充的信息。代码分析工具，开发工具和部署工具可以通过这些补充信息进行验证或者进行部署。Annotation类似于修饰符一样被使用，可以用于包，类，构造方法，方法，成员变量，参数，局部变量的声明。

注意：

Annotation是一个接口

java.lang.Annotation接口.

## 二、系统定义的Annotation

在JDK 1.5之后，在系统中提供了三个Annotation，分别是：@Override、@Deprecated、@SuppressWarnings。

**@Override**

表示当前的方法定义将覆盖超类中的方法。如果你不小心拼写错误，或者方法签名对不上被覆盖的方法，编译器就会发出错误提示。

//用于验证方法是否覆盖父类中的方法  
@Override  
public String toString() {  
 return "AnnotaionDemo{" +  
 "name='" + name + '\'' +  
 ", age=" + age +  
 '}';  
}

**@Deprecated**

表示的是一个类或方法已经不再建议继续使用了，标记为已过时。

//用于标记方法已过时，不建议使用  
@Deprecated  
public String printInfo() {  
 return "AnnotaionDemo{" +  
 "name='" + name + '\'' +  
 ", age=" + age +  
 '}';  
}

**@SuppressWarnings**

表示关闭不当的编译器警告信息。可以放在类上或方法上

@SuppressWarnings(“unchecked”)//未检查的转化，如集合没有指定类型

@SuppressWarnings(“unused”) //未使用的变量

@SuppressWarnings(“resource”) //有泛型未指定类型

@SuppressWarnings(“path”) //在类路径，原文件路径中有不存在的路径

@SuppressWarnings("deprecation") //使用了某些不赞成使用的类和方法

@SuppressWarnings("fallthrough") //switch语句执行到底没有break关键字

@SuppressWarnings("serial")//某类实现Serializable 但是没有定义serialVersionUID 这个需要但是不必须的字段

@SuppressWarnings("rawtypes") //没有传递带有泛型的参数

@SuppressWarnings("all") //全部类型的警告

## 三、自定义Annotation

注解应用需要三个步骤：

（1）编写注解

（2）在类上应用注解

（3）对应用了注解的类进行反射操作的类

自定义Annotation的语法如下：

访问控制权限 @interface Annotation名称{}

例如：

**public** **@interface** MyAnnotation {}

|  |
| --- |
| 自定义注解类  package com.vince;  */\*\*  \* 自定义注解  \*/* public @interface MyAnnotation {  //定义变量  public String name();  public int age() default 2; //给变量设置默认值  public String[] like();//定义数组变量  public Color color(); //定义一个枚举类型的变量 }  枚举类  package com.vince;  public enum Color {  *Red*, *Green*, *Yellow*; }  自定义注解的使用  package com.vince; /\*\*  \*在类上使用自定义注解  \*/ @MyAnnotation(name = "vince", like = {"金鱼","鲤鱼"}, color = Color.*Green*) public class Cat {  private String name;  private int age;  private String[] like;  private Color color;   public Cat() {  }   public Cat(String name, int age, Color color) {  this.name = name;  this.age = age;  this.color = color;  }   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[] getLike() {  return like;  }   public void setLike(String[] like) {  this.like = like;  }  public Color getColor() {  return color;  }   public void setColor(Color color) {  this.color = color;  } } |

## 四、Retention和RetentionPolicy

Annotation要想决定其作用的范围，通过@Retention指定，而Retention指定的范围由RetentiontPolicy决定，RetentionPolicy指定了三种范围：

|  |  |
| --- | --- |
| 范围 | 描述 |
| public static final RetentionPolicy SOURCE | 在java源程序中存在 |
| public static final RetentionPolicy CLASS | 在java生成的class中存在 |
| public static final RetentionPolicy RUNTIME | 在java运行的时候存在 |

*/\*\*  
 \* 自定义注解  
 \*/*//表示该注解的作用范围在运行时存在  
@Retention(RetentionPolicy.*RUNTIME*)  
public @interface MyAnnotation {  
 //定义变量  
 public String name();  
 public int age() default 2; //给变量设置默认值  
 public String[] like();//定义数组变量  
 public Color color(); //定义一个枚举类型的变量  
}

## 五、反射与Annotation

一个Annotation真正起作用，必须结合反射机制，在反射中提供了以下的操作方法：java.lang.reflect.AccessibleObject

|  |  |
| --- | --- |
| 方法名称 | 描述 |
| public boolean isAnnotationPresent(Class<? extends Annotation> annotationClass) | 判断是否是指定的Annotation |
| public Annotation[] getAnnotations() | 得到全部的Annotation |

@Test  
public void test () {  
 Class<Cat> catClass = Cat.class;  
 //获取类上应用的自定义注解  
 MyAnnotation annotation = catClass.getAnnotation(MyAnnotation.class);  
 //获取注解上的变量值  
 String name = annotation.name();  
 int age = annotation.age();  
 String[] like = annotation.like();  
 Color color = annotation.color();  
 try {  
 Cat cat = catClass.newInstance();  
 cat.setName(name);  
 cat.setAge(age);  
 cat.setLike(like);  
 cat.setColor(color);  
 System.*out*.println(cat);  
 } catch (InstantiationException e) {  
 e.printStackTrace();  
 } catch (IllegalAccessException e) {  
 e.printStackTrace();  
 }  
}

## 六、@Documented注解

此注解表示的是文档化，可以在生成doc文档的时候添加注解。

@Documented

@Retention(value = RetentionPolicy.*RUNTIME*)

**public** **@interface** MyAnnotation {

**public** String name();

**public** String info();

}

可以增加一些DOC注释。

/\*\*

\* 这个方法是从Object类中覆写而来的

\*/

@MyAnnotation(name = “vince", info = “teacher")

**public** String toString() {

**return** "hello";

}

## 七、@Target注解

@Target注解表示的是一个Annotation的使用范围，例如：之前定义的MyAnnotation可以在任意的位置上使用。

|  |  |
| --- | --- |
| 范围 | 描述 |
| public static final ElementType TYPE | 只能在类或接口或枚举上使用 |
| public static final ElementType FIELD | 在成员变量使用 |
| public static final ElementType METHOD | 在方法中使用 |
| public static final ElementType PARAMETER | 在参数上使用 |
| public static final ElementType CONSTRUCTOR | 在构造中使用 |
| public static final ElementType LOCAL\_VARIABLE | 局部变量上使用 |
| public static final ElementType ANNOTATION\_TYPE | 只能在Annotation中使用 |
| public static final ElementType PACKAGE | 只能在包中使用 |

## 八、@Inherited注解

@Inherited表示一个Annotation是否允许被其子类继承下来。

示例

@Inherited

@Target(value = ElementType.TYPE)

@Retention(value = RetentionPolicy.RUNTIME)

public @interface MyAnnotation {

public String name();

public String info();

}

使用时允许被其子类所继承。

# 第十六章 XML与JSON

## 一、XML

### 1、什么是XML

XML（Extensible Markup Language 可扩展标记语言），XML是一个以文本来描述数据的文档。

|  |
| --- |
| **一个XML文档示例：**  <?xml version="1.0" encoding="UTF-8"?>  <people>  <person personid="E01">  <name>Tony</name>  <address>10 Downing Street, London, UK</address>  <tel>(061) 98765</tel>  <fax>(061) 98765</fax>  <email>tony@everywhere.com</email>  </person>  <person personid="E02">  <name>Bill</name>  <address>White House, USA</address>  <tel>(001) 6400 98765</tel>  <fax>(001) 6400 98765</fax>  <email>bill@everywhere.com</email>  </person>  </people> |

### 2、XML的用途

XML技术的用途：

（1）充当显示数据（以XML充当显示层）

（2）存储数据（存储层）的功能

（3）以XML描述数据，并在联系服务器与系统的其余部分之间传递。(传输数据的一样格式)

从某种角度讲，XML是数据封装和消息传递技术。

### 3、SAX解析XML

SAX是Simple API for XML的缩写

SAX 是读取和操作 XML 数据更快速、更轻量的方法。SAX 允许您在读取文档时处理它，从而不必等待整个文档被存储之后才采取操作。它不涉及 DOM 所必需的开销和概念跳跃。 SAX API是一个基于事件的API ，适用于处理数据流，即随着数据的流动而依次处理数据。SAX API 在其解析您的文档时发生一定事件的时候会通知您。在您对其响应时，您不作保存的数据将会被抛弃。

SAX API中主要有四种处理事件的接口，它们分别是ContentHandler，DTDHandler， EntityResolver 和 ErrorHandler 。实际上只要继承DefaultHandler 类就可以，DefaultHandler实现了这四个事件处理器接口，然后提供了每个抽象方法的默认实现。

|  |
| --- |
| Xml文件  <?xml version="1.0" encoding="UTF-8"?> <people>  <person personid="E01">  <name>Tony</name>  <address>10 Downing Street, London, UK</address>  <tel>(061) 98765</tel>  <fax>(061) 98765</fax>  <email>tony@everywhere.com</email>  </person>  <person personid="E02">  <name>Bill</name>  <address>White House, USA</address>  <tel>(001) 6400 98765</tel>  <fax>(001) 6400 98765</fax>  <email>bill@everywhere.com</email>  </person> </people>  POJO类  package com.vince2;  public class Person {  private String personid;  private String name;  private String address;  private String tel;  private String fax;  private String email; }  SAX解析类  package com.vince2;  import org.xml.sax.Attributes; import org.xml.sax.SAXException; import org.xml.sax.helpers.DefaultHandler;  import java.util.ArrayList; import java.util.List;  public class PersonHandler extends DefaultHandler {  private List<Person> persons;  private Person person;//当前正在解析的person  private String tag; //用于记录当前正在解析的标签   public List<Person> getPersons() {  return persons;  }   */\*\*  \* 开始解析文档时调用  \*/* @Override  public void startDocument() throws SAXException {  super.startDocument();  persons = new ArrayList<Person>();  System.*out*.println("开始解析文档...");  }  */\*\*  \* 在XML文档解析结束时调用  \*/* @Override  public void endDocument() throws SAXException {  super.endDocument();  System.*out*.println("文档解析结束...");  }   */\*\*  \* 解析开始元素时调用  \** ***@param*** *uri 命名空间  \** ***@param*** *localName 不带前缀的标签名 如1、<aa:person> 返回person 2、<person> 返回person  \** ***@param*** *qName 带前缀的标签名 如：1、<aa:person> 返回aa:person 2、<person> 返回person  \** ***@param*** *attributes 当前标签的属性集合  \** ***@throws*** *SAXException  \*/* @Override  public void startElement(String uri, String localName, String qName, Attributes attributes) throws SAXException {  super.startElement(uri, localName, qName, attributes);  if ("person".equals(qName)) {  person = new Person();  String personid = attributes.getValue("personid");  person.setPersonid(personid);  }  tag = qName;  }   */\*\*  \* 解析结束元素时调用  \** ***@param*** *uri 命名空间  \** ***@param*** *localName 不带前缀的标签名 如1、</aa:person> 返回person 2、</person> 返回person  \** ***@param*** *qName 带前缀的标签名 如：1、</aa:person> 返回aa:person 2、</person> 返回person  \** ***@throws*** *SAXException  \*/* @Override  public void endElement(String uri, String localName, String qName) throws SAXException {  super.endElement(uri, localName, qName);  if ("person".equals(qName)) {  persons.add(person);  }  tag = null;  }   */\*\*  \* 解析文本内容时调用  \** ***@param*** *ch 当前行切割成数组格式  \** ***@param*** *start 文本所在的开始值  \** ***@param*** *length 文本值的长度  \** ***@throws*** *SAXException  \*/* @Override  public void characters(char[] ch, int start, int length) throws SAXException {  super.characters(ch, start, length);  if (tag != null) {  if ("name".equals(tag)) {  person.setName(new String(ch, start, length));  } else if ("address".equals(tag)) {  person.setAddress(new String(ch, start, length));  } else if ("tel".equals(tag)) {  person.setTel(new String(ch, start, length));  } else if ("fax".equals(tag)) {  person.setFax(new String(ch, start, length));  } else if ("email".equals(tag)) {  person.setEmail(new String(ch, start, length));  }  }  } }  实现SAX解析方法  */\*\*  \* SAX解析的特点  \* 1、基于事件驱动  \* 2、顺序读取，速度快  \* 3、不能任意读取节点（灵活性差）  \* 4、解析时占用内存小  \* 5、SAX更适用于在性能要求更高的设备上使用（Android开发中）  \*/*  @Test  public void setParseXML() {  try {  //1、创建一个SAX解析器工厂对象  SAXParserFactory saxParserFactory = SAXParserFactory.*newInstance*();  //2、通过工程对象创建SAX解析器  SAXParser saxParser = saxParserFactory.newSAXParser();  //3、创建一个数据处理器（需要我们自己来填写）  PersonHandler personHandler = new PersonHandler();  //4、开始解析  InputStream is = Thread.*currentThread*().getContextClassLoader()  .getResourceAsStream("com/vince2/person.xml");  saxParser.parse(is, personHandler);  List<Person> persons = personHandler.getPersons();  for (Person p : persons) {  System.*out*.println(p);  }  } catch (ParserConfigurationException e) {  e.printStackTrace();  } catch (SAXException e) {  e.printStackTrace();  } catch (IOException e) {  e.printStackTrace();  }  } |

### 4、DOM解析XML

JAVA 解析 XML 通常有两种方式，**DOM 和 SAX**

DOM：Document Object Model（文档对象模型）

DOM的特性：

定义一组 Java 接口，基于对象，与语言和平台无关将 XML 文档表示为树，在内存中解析和存储 XML 文档，允许随机访问文档的不同部分。

**DOM解析XML**

DOM的优点，由于树在内存中是持久的，因此可以修改后更新。它还可以在任何时候在树中上下导航，API使用起来也较简单。

|  |
| --- |
| XML文件  <?xml version="1.0" encoding="UTF-8"?> <people>  <person personid="E01">  <name>Tony</name>  <address>10 Downing Street, London, UK</address>  <tel>(061) 98765</tel>  <fax>(061) 98765</fax>  <email>tony@everywhere.com</email>  </person>  <person personid="E02">  <name>Bill</name>  <address>White House, USA</address>  <tel>(001) 6400 98765</tel>  <fax>(001) 6400 98765</fax>  <email>bill@everywhere.com</email>  </person> </people>  POJO类  public class Person {  private String personid;  private String name;  private String address;  private String tel;  private String fax;  private String email;  实现DOM解析方法  */\*\*  \* DOM解析XML特点：  \* 基于树形结构，通过解析一次性把文档加载到内存中，所以会比较占用内存，  \* 可以随机访问，更加灵活，更适合在web开发中使用  \*/* @Test public void domParseXml() {  try {  //1、创建一个DOM解析器工厂对象  DocumentBuilderFactory builder = DocumentBuilderFactory.*newInstance*();  //2、通过工厂对象创建解析器对象  DocumentBuilder db = builder.newDocumentBuilder();  //3、解析文档  InputStream is = Thread.*currentThread*().getContextClassLoader()  .getResourceAsStream("com/vince2/person.xml");  //此代码完成后，整个XML文档已经加载到内存中，以树桩形式存储  Document doc = db.parse(is);  //4、从内存中读取数据  //获取节点名称为person的所有节点，返回节点集合  NodeList personNodeList = doc.getElementsByTagName("person");  List<Person> persons = new ArrayList<>();  Person person = null;  for (int i = 0; i < personNodeList.getLength(); i++) {  Node personNode = personNodeList.item(i);  person = new Person();  String personid = personNode.getAttributes().getNamedItem("personid").getNodeValue();  person.setPersonid(personid);  //获取当前节点的所有子节点  NodeList childNodes = personNode.getChildNodes();  for (int j = 0; j < childNodes.getLength(); j++) {  Node item = childNodes.item(j);  String nodeName = item.getNodeName();  if ("name".equals(nodeName)) {  person.setName(item.getFirstChild().getNodeValue());  } else if ("address".equals(nodeName)) {  person.setAddress(item.getFirstChild().getNodeValue());  } else if ("tel".equals(nodeName)) {  person.setTel(item.getFirstChild().getNodeValue());  } else if ("fax".equals(nodeName)) {  person.setFax(item.getFirstChild().getNodeValue());  } else if ("email".equals(nodeName)) {  person.setEmail(item.getFirstChild().getNodeValue());  }  }  persons.add(person);  System.*out*.println(person);  }   } catch (ParserConfigurationException e) {  e.printStackTrace();  } catch (SAXException e) {  e.printStackTrace();  } catch (IOException e) {  e.printStackTrace();  } } |

### 5、JDOM解析XML

JDOM是两位著名的 Java 开发人员兼作者，Brett Mclaughlin 和 Jason Hunter 的创作成果， 2000 年初在类似于Apache协议的许可下，JDOM作为一个开放源代码项目正式开始研发了。

JDOM 简化了与 XML 的交互并且比使用 DOM 实现更快，JDOM 与 DOM 主要有两方面不同。首先，JDOM 仅使用具体类而不使用接口。这在某些方面简化了 API，但是也限制了灵活性。第二，API 大量使用了 Collections 类，简化了那些已经熟悉这些类的 Java 开发者的使用。

下载地址：

<http://www.jdom.org/downloads/index.html>

导入jar包：jdom-2.0.6.jar

|  |
| --- |
| XML文件  <?xml version="1.0" encoding="UTF-8"?> <people>  <person personid="E01">  <name>Tony</name>  <address>10 Downing Street, London, UK</address>  <tel>(061) 98765</tel>  <fax>(061) 98765</fax>  <email>tony@everywhere.com</email>  </person>  <person personid="E02">  <name>Bill</name>  <address>White House, USA</address>  <tel>(001) 6400 98765</tel>  <fax>(001) 6400 98765</fax>  <email>bill@everywhere.com</email>  </person> </people>  POJO类  public class Person {  private String personid;  private String name;  private String address;  private String tel;  private String fax;  private String email;  实现JDOM解析方法  */\*\*  \* JDOM解析XML  \* 1、与DAO类似基于树形结构  \* 2、与DAO的区别  \* （1）第三方开源组件  \* （2）实现使用JAVA的Collection接口  \* （3）效率比DOM更快  \*/* @Test public void jdomParseXML() throws JDOMException, IOException {  //1、创建JDOM解析器  SAXBuilder builder = new SAXBuilder();  //2、解析文档  InputStream is = Thread.*currentThread*().getContextClassLoader()  .getResourceAsStream("com/vince2/person.xml");  org.jdom2.Document build = builder.build(is);  Element rootElement = build.getRootElement();  List<Person> persons = new ArrayList<>();  Person person = null;  List<Element> children = rootElement.getChildren();  for (Element element : children) {  person = new Person();  String personid = element.getAttributeValue("personid");  person.setPersonid(personid);  //获取当前节点的所有子节点  List<Element> children1 = element.getChildren();  for (Element e : children1) {  String tag = e.getName();  if ("name".equals(tag)) {  person.setName(e.getText());  } else if ("address".equals(tag)) {  person.setAddress(e.getText());  } else if ("tel".equals(tag)) {  person.setTel(e.getText());  } else if ("fax".equals(tag)) {  person.setFax(e.getText());  } else if ("email".equals(tag)) {  person.setEmail(e.getText());  }  }  persons.add(person);  System.*out*.println(person);  } } |

### 6、DOM4J解析XML

dom4j是一个非常非常优秀的Java XML API，具有性能优异、功能强大和极端易用使用的特点，同时它也是一个开放源代码的软件，可以在SourceForge上找到它。在对主流的Java XML API进行的性能、功能和易用性的评测，dom4j无论在那个方面都是非常出色的。如今你可以看到越来越多的Java软件都在使用dom4j来读写XML，特别值得一提的是连Sun的JAXM也在用dom4j。这是必须使用的jar包， Hibernate用它来读写配置文件。

下载地址：

<https://dom4j.github.io/>

导入jar包：dom4j-1.6.1.jar

|  |
| --- |
| XML文件  <?xml version="1.0" encoding="UTF-8"?> <people>  <person personid="E01">  <name>Tony</name>  <address>10 Downing Street, London, UK</address>  <tel>(061) 98765</tel>  <fax>(061) 98765</fax>  <email>tony@everywhere.com</email>  </person>  <person personid="E02">  <name>Bill</name>  <address>White House, USA</address>  <tel>(001) 6400 98765</tel>  <fax>(001) 6400 98765</fax>  <email>bill@everywhere.com</email>  </person> </people>  POJO类  public class Person {  private String personid;  private String name;  private String address;  private String tel;  private String fax;  private String email;  实现DOM4J解析方法  */\*\*  \* DOM4J解析XML  \* 1、基于树形结构 第三方组件  \* 2、解析速度快，效率更高，使用java中的迭代器实现数据读取，在web框架中使用较多（Hibernate）  \*/* @Test public void dom4jParseXML() throws DocumentException {  //1、创建DOM4J解析器  SAXReader reader = new SAXReader();  //2、解析文档  InputStream is = Thread.*currentThread*().getContextClassLoader()  .getResourceAsStream("com/vince2/person.xml");  org.dom4j.Document doc = reader.read(is);  org.dom4j.Element rootElement = doc.getRootElement();  Iterator<org.dom4j.Element> iterator = rootElement.elementIterator();  List<Person> persons = new ArrayList<>();  Person person = null;  while (iterator.hasNext()) {  person = new Person();  org.dom4j.Element element = iterator.next();  person.setPersonid(element.attributeValue("personid"));  //获取当前节点的所有子节点  Iterator<org.dom4j.Element> iterator1 = element.elementIterator();  while (iterator1.hasNext()) {  org.dom4j.Element e = iterator1.next();  String tag = e.getName();  if ("name".equals(tag)) {  person.setName(e.getText());  } else if ("address".equals(tag)) {  person.setAddress(e.getText());  } else if ("tel".equals(tag)) {  person.setTel(e.getText());  } else if ("fax".equals(tag)) {  person.setFax(e.getText());  } else if ("email".equals(tag)) {  person.setEmail(e.getText());  }  }  persons.add(person);  System.*out*.println(person);   } } |

### 7、各种解析方法比较

JDOM 和 DOM 在性能测试时表现不佳，在测试 10M 文档时内存溢出。

SAX表现较好，这要依赖于它特定的解析方式。一个 SAX 检测即将到来的XML流，但并没有载入到内存（当然当XML流被读入时，会有部分文档暂时隐藏在内存中）。   
  
DOM4J是这场测试的获胜者，目前许多开源项目中大量采用 DOM4J，例如大名鼎鼎的 Hibernate 也用 DOM4J 来读取 XML 配置文件。

### 8、通过对象生成XML文件

根据对象生成XML文档.

使用Java提供的java.beans.XMLEncoder和java.beans.XMLDecoder类。

这是JDK 1.4以后才出现的类

|  |
| --- |
| */\*\*  \* 把对象转成XML文件写入  \*/* @Test public void xmlEnCoder() throws FileNotFoundException {  BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream("test.xml"));  XMLEncoder xmlEncoder = new XMLEncoder(bos);  Person person = new Person();  person.setPersonid("1212");  person.setName("38");  person.setAddress("北京");  person.setTel("13838484589");  person.setFax("34677856");  person.setEmail("schu@cnpc.com.cn");  xmlEncoder.writeObject(person);  xmlEncoder.close(); } */\*\*  \* 把XML文件中读取对象  \*/* @Test public void xmlDnCoder() throws FileNotFoundException {  BufferedInputStream bis = new BufferedInputStream(new FileInputStream("test.xml"));  XMLDecoder xmlDecoder = new XMLDecoder(bis);  Person p = (Person)xmlDecoder.readObject();  System.*out*.println(p); } |

### 9、第三方组件xstream 实现XML的转换

需要jar包：

xpp3\_min-1.1.4c.jar

xstream-1.4.7.jar

|  |
| --- |
| */\*\*  \* 使用第三方xstream组件实现XML的解析与生成  \*/* @Test public void xStream() {  Person person = new Person();  person.setPersonid("1212");  person.setName("38");  person.setAddress("北京");  person.setTel("13838484589");  person.setFax("34677856");  person.setEmail("schu@cnpc.com.cn");  XStream xStream = new XStream(new Xpp3Driver());  xStream.alias("person", Person.class);//设置根节点别名  xStream.useAttributeFor(Person.class, "personid");//设置根节点属性  String xml = xStream.toXML(person);  System.*out*.println(xml);   //解析XML  Person p = (Person)xStream.fromXML(xml);  System.*out*.println(p); } |

## 二、JSON

### 1、什么是JSON

JSON(JavaScript Object Notation) 是一种轻量级的数据交换格式。 JSON 官方:http://www.json.org

JSON 数据格式的特点

JSON 建构于两种结构：

1、 “名称/值”对的集合

2.、 值的有序列表（数组）

JSON 表示名称 / 值对的方式 ：

{ "firstName": "vince", "lastName":"ma", "email": "finally\_m@foxmail.com" }

表示数组

{ "user": [

{ "firstName": "vince", "lastName":"ma", "email": "finally\_m@foxmail.com" },

{ "firstName": "lin", "lastName":"jacks", "email": “jacks@qq.com”}

]}

### 2、GSON组件的使用

**GSON是Google开发的Java API，用于转换Java对象和Json对象。**

下载地址：http://www.mvnrepository.com/artifact/com.google.code.gson/gson

需要jar包：gson-2.5.jar

#### 2.1. java对象与JSON对象之间的转换生成

|  |
| --- |
| name.json  {"firstName":"Vince", "lastName": "Ma", "email": "schu@cnpc.com"}  names.json  [  {"firstName":"Vince", "lastName": "Ma", "email": "schu@cnpc.com"},  {"firstName":"Jason", "lastName": "Hunter", "email": "json@cnpc.com"} ]  name对象  public class Name {  private String firstName;  private String lastName;  private String email;  案例1：把java对象数组转换成Json对象  */\*\*  \*把java对象数组转换成Json对象*  *\*/* @Test public void createJSON() {  List<Name> list = new ArrayList<>();  list.add(new Name("vince", "me", "123@cnpc.cn"));  list.add(new Name("jack", "wang", "jack@cnpc.cn"));  JsonArray jsonArray = new JsonArray();  for (Name name : list) {  JsonObject jsonObject = new JsonObject();  jsonObject.addProperty("firstName", name.getFirstName());  jsonObject.addProperty("lastName", name.getLastName());  jsonObject.addProperty("email", name.getEmail());  jsonArray.add(jsonObject);  }  System.*out*.println(jsonArray.toString()); }  案例2：把一个JSON对象转换成JAVA对象或把一个java对象转换成Json对象  */\*\*  \* 把一个JSON对象转换成JAVA对象  \* 或把一个java对象转换成Json对象  \*/* @Test public void gson1() {  Gson gson = new Gson();  InputStream is = Thread.*currentThread*().getContextClassLoader()  .getResourceAsStream("com/vince3/name.json");  InputStreamReader in = new InputStreamReader(is);  Name name = gson.fromJson(in, Name.class);  System.*out*.println(name);   String json = gson.toJson(name);  System.*out*.println(json); }  案例3：把一组JSON对象转换成一个JAVA对象集合或把一个java对象集合转换成Json数组  */\*\*  \* 使用GSON把一组JSON对象转换成一个JAVA对象集合  \* 或把一个java对象集合转换成Json数组  \*/* @Test public void gson2() {  Gson gson = new Gson();  InputStream is = Thread.*currentThread*().getContextClassLoader()  .getResourceAsStream("com/vince3/names.json");  InputStreamReader in = new InputStreamReader(is);  TypeToken<List<Name>> typeToken = new TypeToken<>();  List<Name> list = gson.fromJson(in, typeToken.getType());  System.*out*.println(list);   String json = gson.toJson(list, typeToken.getType());  System.*out*.println(json); }  class MyTypeToken extends TypeToken<List<Name>> {} |

#### 2.2.解析JSON数组

案例一（简单的）：

|  |
| --- |
| names.json  [  {"firstName":"Vince", "lastName": "Ma", "email": "schu@cnpc.com"},  {"firstName":"Jason", "lastName": "Hunter", "email": "json@cnpc.com"} ]  Name类对象  public class Name {  private String firstName;  private String lastName;  private String email;  解析JSON方法  import com.google.gson.stream.JsonReader; import com.google.gson.stream.JsonToken; import org.junit.Test;  import java.io.IOException; import java.io.InputStream; import java.io.InputStreamReader; import java.util.ArrayList; import java.util.Arrays; import java.util.List;  */\*\*  \* 解析一个JSON数组  \*/* @Test public void parseJsonNames() {  InputStream is = Thread.*currentThread*().getContextClassLoader()  .getResourceAsStream("com/vince3/names.json");  InputStreamReader isr = new InputStreamReader(is);  //Json的解析器  JsonReader reader = new JsonReader(isr);  List<Name> list = new ArrayList<>();  try {  //开始解析数组  reader.beginArray();  while (reader.hasNext()) {  list.add(parseName(reader));  }  //结束解析数组  reader.endArray();  } catch (IOException e) {  e.printStackTrace();  }  System.*out*.println(Arrays.*toString*(list.toArray())); } //解析name对象 private Name parseName(JsonReader jsonReader) {  Name name = new Name();  try {  //开始解析对象  jsonReader.beginObject();  while (jsonReader.hasNext()) {  String attrName = jsonReader.nextName();  if ("firstName".equals(attrName)) {  name.setFirstName(jsonReader.nextString());  } else if ("lastName".equals(attrName)) {  name.setLastName(jsonReader.nextString());  } else if ("email".equals(attrName)) {  name.setEmail(jsonReader.nextString());  }  }  //结束解析对象  jsonReader.endObject();  } catch (IOException e) {  e.printStackTrace();  }  return name; } |

案例二（复杂点的）:

|  |
| --- |
| Message.json  [  {  "id":1111111111,  "text": "How do I read a json",  "geo": **null**,  "user": {  "username": "json\_newb",  "followers\_count": 41  }  },  {  "id":3467982356,  "text": "json djfn sieown sklkdm",  "geo": [50.345722, 56.234733],  "user": {  "username": "jesse",  "followers\_count": 2  }  } ]  Message对象  public class Message {  private long id;  private String text;  private List<Double> geo;  private User user;  user对象  public class User {  private String username;  private int followers\_count;  解析json数组  import com.google.gson.stream.JsonReader; import com.google.gson.stream.JsonToken; import org.junit.Test;  import java.io.IOException; import java.io.InputStream; import java.io.InputStreamReader; import java.util.ArrayList; import java.util.Arrays; import java.util.List;  */\*\*  \* 解析一个复杂的JSON数组  \*/* @Test public void parseJsonMessage() {  InputStream is = Thread.*currentThread*().getContextClassLoader()  .getResourceAsStream("com/vince3/message.json");  InputStreamReader isr = new InputStreamReader(is);  //Json的解析器  JsonReader reader = new JsonReader(isr);  List<Message> list = readMessageArray(reader);   System.*out*.println(Arrays.*toString*(list.toArray())); } //解析json数组 private ArrayList<Message> readMessageArray(JsonReader jsonReader) {  ArrayList<Message> list = new ArrayList<>();  try {  //开始解析数组  jsonReader.beginArray();  while (jsonReader.hasNext()) {  list.add(readMessage(jsonReader));  }  //结束解析数组  jsonReader.endArray();  } catch (IOException e) {  e.printStackTrace();  }  return list; } //解析Message对象 private Message readMessage(JsonReader jsonReader) {  Message message = new Message();  try {  //开始解析对象  jsonReader.beginObject();  while (jsonReader.hasNext()) {  String attrName = jsonReader.nextName();  if ("id".equals(attrName)) {  message.setId(jsonReader.nextLong());  } else if ("text".equals(attrName)) {  message.setText(jsonReader.nextString());  } else if ("geo".equals(attrName) && jsonReader.peek() != JsonToken.*NULL*) {  message.setGeo(readGeo(jsonReader));  } else if ("user".equals(attrName)) {  message.setUser(readUser(jsonReader));  } else {  //跳过不处理的数据  jsonReader.skipValue();  }  }  //结束解析对象  jsonReader.endObject();  } catch (IOException e) {  e.printStackTrace();  }  return message; }  //解析geo private List<Double> readGeo(JsonReader jsonReader) {  List<Double> list = new ArrayList<>();  try {  //开始解析数组  jsonReader.beginArray();  while (jsonReader.hasNext()) {  list.add(jsonReader.nextDouble());  }  //结束解析数组  jsonReader.endArray();  } catch (IOException e) {  e.printStackTrace();  }  return list; }  //解析user对象 private User readUser(JsonReader jsonReader) {  User user = new User();  try {  //开始解析对象  jsonReader.beginObject();  while (jsonReader.hasNext()) {  String attrName = jsonReader.nextName();  if ("username".equals(attrName)) {  user.setUsername(jsonReader.nextString());  } else if ("followers\_count".equals(attrName)) {  user.setFollowers\_count(jsonReader.nextInt());  } else {  //跳过不处理的数据  jsonReader.skipValue();  }  }  //结束解析对象  jsonReader.endObject();  } catch (IOException e) {  e.printStackTrace();  }  return user; } |

## 三、XML与JSON的比较

**从以下6点比较JSON与XML：**

1、JSON和XML的数据可读性基本相同

2、JSON和XML同样拥有丰富的解析手段

3、JSON相对于XML来讲，数据的体积小

4、JSON与JavaScript的交互更加方便

5、JSON对数据的描述性比XML较差

6、JSON的速度要远远快于XML

适合的场景：

（1）数据传输：JSON要比XML更有优势

（2）存储数据：XML描述性更强

（3）XML通常用做配置文件（WEB课程中会有详细介绍）

# 第十七章 GUI事件处理

## 一、GUI组件介绍

GUI编程（Graphic User Interface，图形用户接口）

GUI的各种元素，如：容器、按钮、文本框等

1、Frame类

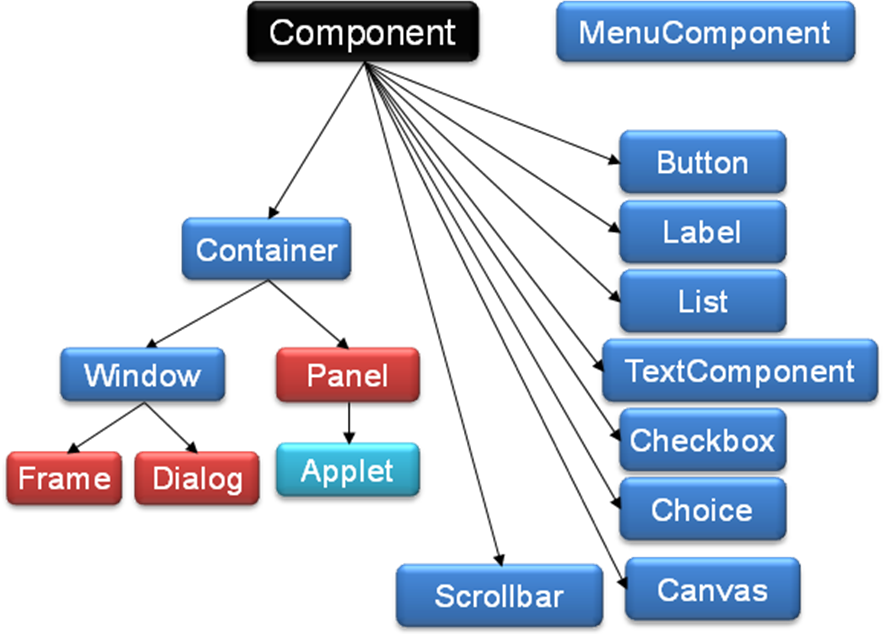
2、Button类

3、Panel类

5、Toolkit类

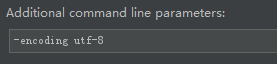
5、布局管理器

6、基本组件



设置窗体字体显示编码：(需要设置两个地方)

1、在idea中 file—Setting—Build…. ---Compiler---java Compiler



2、在idea中 file—Setting—Editor---File Encodings



|  |
| --- |
| package com.gui;  import java.awt.\*; import java.awt.event.ActionEvent; import java.awt.event.ActionListener; import java.awt.event.WindowAdapter; import java.awt.event.WindowEvent;  public class MyFram extends Frame implements ActionListener {   */\*\*  \* 初始化窗体的基本属性  \*/* public MyFram () {  this.setSize(600, 400);  this.setTitle("我的第一个GUI窗口");  //创建一个按钮  Button button = new Button("点我一下");  //给按钮添加单机事件  button.addActionListener(this);  //创建一个线性布局  FlowLayout flowLayout = new FlowLayout();  //把布局应用到窗体上  this.setLayout(flowLayout);  //给窗体添加关闭事件  this.addWindowListener(new WindowAdapter() {  @Override  public void windowClosing(WindowEvent e) {  super.windowClosing(e);  System.*exit*(0);  }  });  //把按钮添加到窗体上  this.add(button);  //让窗体显示出来  this.setVisible(true);  }   */\*\*  \* 单机事件处理方法  \** ***@param*** *e  \*/* @Override  public void actionPerformed(ActionEvent e) {  System.*out*.println("惊喜就在这里");  }  public static void main(String[] args) {  new MyFram();  } } |

## 二、事件处理

事件处理：

事件(Event)：用户对组件的一个操作，称之为一个事件

事件源(Event source) ：产生事件的对象

事件处理方法(Event handler) ： 能够接收、解析和处理事件类对象、实现和用户交互的方法 ，事件监听器。

为简化编程，针对大多数事件监听器接口定义了相应的实现类----事件适配器类，在适配器类中，实现了相应监听器接口中所有的方法，但不做任何事情。

|  |
| --- |
| Frame1.java  package com.gui;  import java.awt.\*; import java.awt.event.ActionEvent; import java.awt.event.ActionListener;  public class Frame1 extends Frame implements Frame2.MoneyListener {  //创建一个Label文本  private Label label = new Label("金额：");  //创建一个按钮  private Button button = new Button("购买");  public Frame1 () {  //设置窗体尺寸  this.setSize(400, 200);  //把布局应用到窗体上  this.setLayout(new FlowLayout());  //把文本添加到窗体上  this.add(label);  //把按钮添加到窗体上  this.add(button);  //给按钮添加事件  button.addActionListener(new ActionListener() {  @Override  public void actionPerformed(ActionEvent e) {  new Frame2().setMoneyListener(Frame1.this);  }  });  //让窗体显示出来  this.setVisible(true);  }   @Override  public void setMoney(String money) {  label.setText("金额："+money);  }   public static void main(String[] args) {  new Frame1();  } }  Frame2.java  package com.gui;  import java.awt.\*; import java.awt.event.ActionEvent; import java.awt.event.ActionListener;  */\*\*  \* 接口回调：  \* 当一个对象需要给外部对象提供数据时，  \* 我们可以定义一个内部接口把数据通过接口传递出去  \* 所有外部对象需要这个数据时，就实现这个接口，  \* 这样的好处是传递数据的对象不直接依赖接受数据的对象  \* （降低耦合度）  \*/* public class Frame2 extends Frame {  //创建一个输入框并设置长度  private TextField textField = new TextField(20);  //创建一个按钮  private Button button = new Button("付款");  public Frame2() {  //设置窗体尺寸  this.setSize(400, 200);  //设置为线性布局  this.setLayout(new FlowLayout());  //把输入框添加到窗体上  this.add(textField);  //把按钮添加到窗体上  this.add(button);  //给按钮添加事件  button.addActionListener(new ActionListener() {  @Override  public void actionPerformed(ActionEvent e) {  String money = textField.getText();  moneyListener.setMoney(money);  }  });  //让窗体显示出来  this.setVisible(true);  }   private MoneyListener moneyListener;   public void setMoneyListener(MoneyListener moneyListener) {  this.moneyListener = moneyListener;  }   public static interface MoneyListener {  public void setMoney(String money);  } } |

## 三、观察者模式

1、观察者模式原理

观察者模式定义：简单地说，观察者模式定义了一个一对多的依赖关系，让一个或多个观察者对象监察一个主题对象。这样一个主题对象在状态上的变化能够通知所有的依赖于此对象的那些观察者对象，使这些观察者对象能够自动更新。

**2、观察者模式实现**

**Subject（被观察的对象接口）**

规定ConcreteSubject的统一接口；

每个Subject可以有多个Observer；

**ConcreteSubject（具体被观察对象）**

维护对所有具体观察者的引用的列表；

状态发生变化时会发送通知给所有注册的观察者。

**Observer（观察者接口）**

规定ConcreteObserver的统一接口；

定义了一个update()方法，

在被观察对象状态改变时会被调用。

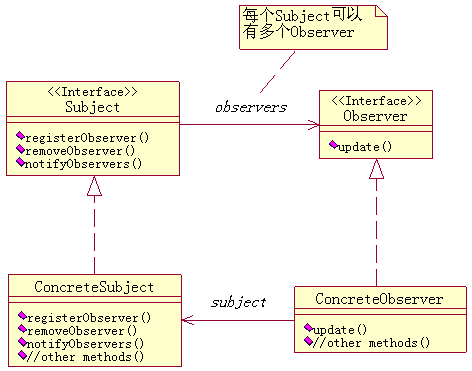
**ConcreteObserver（具体观察者）**

维护一个对ConcreteSubject的引用；

特定状态与ConcreteSubject同步；

实现Observer接口，通过update()方法

接收ConcreteSubject的通知。



3、观察者模式作用

观察者模式在被观察者和观察者之间建立一个抽象的耦合。被观察者角色所知道的只是一个具体观察者列表。

 由于被观察者和观察者没有紧密地耦合在一起，因此它们可以属于不同的抽象化层次。如果被观察者和观察者都被扔到一起，那么这个对象必然跨越抽象化和具体化层次。

 观察者模式支持广播通讯。被观察者会向所有的登记过的观察者发出通知。

|  |
| --- |
| 被观察者的接口  package com.abserver;  */\*\*  \* 被观察者的接口  \*/* public interface MessageSubject {  //注册观察者  public void registerObserver(Observer o);  //移除观察者  public void removeObserver(Observer o);  //通知所有观察者  public void notifyObservers(); }  观察者接口  package com.abserver;  */\*\*  \* 观察者接口  \*/* public interface Observer {  //更新消息  public void update(String message); }  被观察者实现类  package com.abserver;  import java.util.ArrayList; import java.util.List;  */\*\*  \* 具体的被观察者  \*/* public class Message implements MessageSubject {  //维护的观察者列表  List<Observer> observerlist = new ArrayList<Observer>();  //通知消息  private String message;   public void setMessage(String message) {  this.message = message;  notifyObservers();  }   @Override  public void registerObserver(Observer o) {  observerlist.add(o);  }   @Override  public void removeObserver(Observer o) {  observerlist.remove(o);  }   @Override  public void notifyObservers() {  for (Observer observer : observerlist) {  observer.update(message);  }  } }  观察者实现类  package com.abserver;  */\*\*  \* 具体的观察者  \*/* public class User implements Observer {  private String name;   public User(String name) {  this.name = name;  }   @Override  public void update(String message) {  System.*out*.println("[" + name + "]收到消息：" + message);  } }  测试类  package com.abserver;  import org.junit.Test;  public class TestDemo {   @Test  public void testObserver () {  Message message = new Message();  Observer lily = new User("lily");  Observer tom = new User("Tom");  Observer vince = new User("vince");  message.registerObserver(lily);  message.registerObserver(tom);  message.registerObserver(vince);  message.setMessage("亲们，么么哒");  } } |