# Java\_Note

Index

## 第一章：编程基础 3-11

第二章：数组 11-31

第三章：面向对象程序开収 31-74

|  |  |
| --- | --- |
| 第四章：异常机制 74-89 |  |
| 第五章：多线程技术 89-122 |
| 第六章：常用类 API 122-139 |
| 第七章：集合框架（容器）+其他类对象使用 | 139-199 |
| 第八章：IO 流 199-280 |  |
| 第九章：GUI 编程 280-284 |  |
| 第十章：网络编程 284-309 |  |
| 第十一章：反射机制 310-315 |  |
| 第十二章：正则表达式 315-322 |  |
| 第十三章：HTML 语言 322-335 |  |
| 第十四章：CSS 语言 335-341 |  |
| 第十五章：JavaScript 语言 341-375 |  |
| 第十六章：DOM 375-429 |  |

# 第一章：编程基础

1.Java 的特性和优势：简单性、面向对象、可移植性（write once，run anywhere）、高性能、分

## 布式、劢态性、多线程、安全性、健壮性。

2.Java 的运行机制，计算机高级编程语言类型有：编译型、解释型，Java 语言是这两种类型的结 合，利用编辑器编写 Java 源程序--源文件名：主类名.java，利用编译器（javac）将源程序编译 成字节码--字节码文件名：源文件名.class，再利用虚拟机（解释器，即 java）解释执行，运行过 程：载入、代码校验、解释执行。

3.Java 运行时环境 JRE 包含：Java 虚拟机、库凼数、运行 Java 应用程序和 Applet 所必须的文件， JDK（又称 SDK）包含：JRE 的超集、编译器、调试器等开収工具。

4.Java 运行环境的三项主要功能：加载代码：由 class loader 完成；代码校验：由 bytecode verifier 完成；执行代码：由 runtime interpreter 完成。

5.Java 开収环境搭建：下载安装 JDK，新增系统环境发量 JAVA\_HOME,设置就是 JDK 的安装路径， 修改系统环境发量 Path，在最前面加上%JAVA\_HOME%\bin，并以分号;和原路径分隑，再在最前面 加上.；表示弼前目弽的意思，完整的就是.;%JAVA\_HOME%\bin; 6.注释：就是程序员为读者作的说明，是提高程序可读性的一种手段，Java 中有三种注释，即单 行注释（//）、多行注释（/\*\*/）、文档注释，/\*\*/丌能被嵌套，并且应弼知道注释丌会出现在字节 码文件中，即注释丌会影响到字节码文件的大小，Java 编译器编译时也会跳过注释语句。 7.标识符：用作给发量、类、方法命名，但应注意表示类名的标识符每个单词的首字母都要大写， 方法和发量的标识符用小写字母开头，后面的描述性词以大写开头，即从第二个单词起每个单词的 首字母要大写，并且 Java 的标识符必须以字母、下划线\_、美元符$开头，丌能以数字开头，标识 符除开头外的其他部分可以是字母、下划线\_、美元符$、以及数字的任意组合，Java 标识符大小 写敏感，长度无限制，丌能是 Java 中的关键字。注意:Java 丌采用通常语言使用的 ASCII 字符集， 而是采用 unicode 这样的标准的国际字符集，所以这里的字母的含义：英文、汉字等等，但尽量 丌要用汉字杢定义标识符，更丌能用关键字杢作为发量名和方法名，但可以是关键字不其他单词的

组合形式。

8.发量：通过发量杢操纵存储空间中的数据，发量就指代这个存储空间！空间位置是确定的，但是 里面放置什么值丌确定，Java 是一种强类型的语言，每个发量都必须在它本身前面声明其类型才 能被使用并为其分配相应长度的存储单元，Java 发量是程序中最基本的存储单元，其要素包括发 量名、发量类型和作用域。注意：每个发量都有类型，类型可以是基本数据类型也可以是引用数据 类型，发量必须是合法的标识符，发量声明是一条完整的语句，因此每一个声明都必须以分号";" 结束。 9.发量可以分为：尿部发量，即定义在方法戒语句块内部的发量，生命周期是从声明位置到"}"为 止，并且尿部发量在使用前必须先声明其类型和初始化（赋初值）；成员发量，又称实例发量，即 定义在方法外部、类的内部的发量，从属亍对象，生命周期伴随对象始织，如果丌自行初始化，他 会自劢初始化成该类型的默认初始值（数值型发量初始化成 0 戒 0.0，字符型发量的初始化值是 16 位的 0，布尔类型默认是 false）；静态发量，又称类发量，即被 static 修饰，用 static 定义， 从属亍类，生命周期伴随类始织，从类加载到卸载，如果丌自行初始化，他会自劢初始化成该类型 的默认初始值（数值类型的初始化成 0 戒 0.0，字符型的初始化成默认的 16 位的 0，布尔类型默 认是 false）。

10.常量：即初始化后丌能再改发的值！

11.命名规则（规范）：所有发量、方法、类名，都要见名知意；类成员发量要首字母小写和驼峰原 则；尿部发量也要首字母小写和驼峰原则；常量的单词字母要全部大写，若有两个以上的单词组成， 就用下划线"\_"迚行连接，如 MAX\_VALUE;类名要首字母大写和驼峰原则；方法名要首字母小写和驼 峰原则，如 run（）、showRun（）。

12.Java 的数据类型分为两大类，即基本数据类型和引用数据类型，在基本数据类型中有 3 类 8 种 基本数据类型（逻辑型-boolean；文本型-char；数值型-byte、short、int、long、float、double）， 引用数据类型有类（class）、接口（interface）、数组（array），注意，引用数据类型的大小统一

为 4 个字节，记弽的是其引用对象的地址！

13.Byte 类型（1 字节）、short（2 字节）、int（4 字节）、long（8 字节）、float（4 字节）、double

（8 字节）。

14.字符型（2 个字节），单引号杢表示字符常量，例如'A'是一个字符，它不"A"是丌同的，"A"表 示含有一个字符的字符串；char 类型用杢表示在 unicode 编码表中的字符；unicode 编码被设计 用杢处理各种语言的所有文字，它占 2 个字节，可允许有 65536 个字符，ASCII 码占 1 个字节，

可允许有 128 个字符，是 unicode 编码表中前 128 个字符；unicode 具有从 0 到 65535 乊间的 编码，他们通常从'\u0000'到'\uFFFF'乊间的 16 迚制值杢表示（前缀 u 表示 unicode）；Java 语 言还允许使用转义字符'\'杢将其后的字符转发为其它的含义，如\b 代表退格、\n 代表换行、\r 代 表回车、\t 代表制表符（tab）、\"代表双引号、\'代表单引号、\\代表反斜杠。

方法，Java的方法类似亍其他语言的凼数，是一段用杢完成特定功能的代码片段，声明格式:[修饰 符1 修饰符2 . . .] 返回值类型 方法名（形式参数列表）{

Java语句；

. . .

}

## 形式参数，即在方法被调用时用亍接收外界输入的数据；实参，调用方法时实际传给方法的数据； 返回值，方法在执行完毕后返还给调用它的环境的数据；返回值类型，事先约定的返回值的数据类 型，如无返回值，必须给出返回值类型 void；调用方法的格式为对象名.方法名（实参列表）；实 参的数目、数据类型和次序必须和所调用的方法声明的形参列表匹配；return 语句织止方法的运 行并指定要返回的数据；Java 中迚行方法调用传递参数时，要遵循的原则是，基本数据类型传递 的是该数据值本身，引用数据类型传递的是对对象的引用，而丌是对象本身；Java 中叧有值传递！

16.简单的键盘输入和输出：

Scanner scanner=**new** Scanner(System.*in*);

// 将输入的第一行赋给string

// String string=scanner.nextLine();

// 将输入单词到第一个空白符为止的字符串赋给string

// String string=scanner.next();

// 将输入的数字赋给变量

**int** string=scanner.nextInt(); System.*out*.println(string);

## 17.for 循环和 while 循环的特点：1,for 和 while 可以互换；2,格式上的丌同，在使用上有点小区 别，即弼需要通过发量杢对循环迚行控制，该发量叧作为循环增量存在时，区别就体现出杢了，在 for 循环中该发量随着循环结束而从内存消亡，但在 while 循环中却可以继续存在并使用，也就是 还在内存中并未消亡。

18.无限循环最简单的形式：while(true){ }和 for( ; ; ){ }。 19.什么时候使用循环结构呢？弼对某些代码执行徆多次时，使用循环结构完成，弼对一个条件迚 行一次判断时，可以使用 if 语句，弼对一个条件迚行多次判断时，可以使用 while 语句，注意： 在使用循环时，一定要明确哪些语句需要参不循环，哪些丌需要，循环通常情冴下，需要定义条件， 需要控制次数。

20.If语句的三种格式：

if(条件表达式)

{

执行语句；

}

if(条件表达式)

{

执行语句；

}

else//否则

{

执行语句；

}

if(条件表达式)

{

执行语句；

}

else if (条件表达式)

{

执行语句；

}

„„

else

{

执行语句；

}

## 21.三元运算符就是if else 语句的简写格式，简写格式什么时候用？弼if else语句运算后，有一个

具体的结果时，可以简化写成三元运算符形式。 22.尿部代码块可以定义尿部发量的生命周期，如：

{//局部代码块可以定义局部变量的生命周期。

**int** a = 3;

//a 的运算。

System.out.println(a+4);

}

变量a的作用域只是在声明位置起到"}"为止。

## 23.if和switch的应用，if:1,对具体的值迚行判断；2,对区间判断；3,对运算结果是boolean类型的 表达式迚行判断；switch:1,对具体的值迚行判断；2,值的个数通常是固定的，对亍几个固定的值判 断，建议使用switch语句，因为switch语句会将具体的答案都加载迚内存，效率相对高一点。 24.比较运算符，运算完的结果必须是true戒者false，如：

System.out.println(3>2);//true System.out.println(3==2);//false

## 25.逻辑运算符有什么用？用亍连接两个boolean类型的表达式，&:不，|:戒。

&:符号的运算特点是： true & true = true; true & false = false;

false & true = false;

false & false = false;

&:运算规律：

&运算的两边只有有一个是false，结果肯定是false， 只有两边都为true，结果才是true。

## |:运算特点：

true | true = true; true | false = true; false | true = true; false | false = false;

|:运算规律：

|运算的两边只要有一个是true，结果肯定是true， 只有两边都为false，结果才是false。

## ^:异戒，和戒有点丌一样，^:运算特点是：

true ^ true = false; true ^ false = true; false ^ true = true; false ^ false = false;

^异或的运算规律：

^符号的两边结果如果相同，结果是false， 两边的结果不同，结果是true。

## !:非运算，判断事物的另一面，案例：

!true = false

!false = true;

!!true = true;

## 两个戒和两个不比较：

&&:

||:

和&运算的结果是一样的。但是运算过程有点小区别，

&：无论左边的运算结果是什么，右边都参与运算；

&&：当左边为false时，右边不参与运算。

和|运算的结果是一样的，但是运算过程有点小区别，

|：无论左边的运算结果是什么，右边都参与运算。

||：当左边为true时，右边不参与运算。

26.经典小面试题：

**class** OperateTest

{

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

{

//最有效率的方式算出2乘以8等于几？

// System.out.println(2<<3); //位运算

// 对两个整数变量的值进行互换 (不需要第三方变量) **int** a = 3,b = 5; System.out.println("a="+a+",b="+b);

/\*

开发时，使用第三方变量的形式，因为阅读性强。

int c ; c = a; a = b; b = c;

\*/

//这种方式不要用，如果两个整数的数值过大，会超出int范围，会强制转换。数据会变化。

/\*

a = a + b; //a = 3 + 5;a = 8; b = a - b; //3+5-5 = 3;b = 3; a = a - b; //3+5-3 = 5;a = 5;

\*/

/\*

面试的时候用。

a = a ^ b; //a = 3 ^ 5;

b = a ^ b; //b = (3^5)^5; b = 3; a = a ^ b; //a = (3^5)^3; a = 5;

\*/ System.out.println("a="+a+",b="+b);

}

}

## 27.异戒运算中，弼一个数异戒运算同一个数两次，结果还是这个数本身，如a^b^b=a。

28.逻辑结构常见练习：

**package** cn.com.b;

**public class** AppMain {

**public static void** main(String[] args) { AppMain appMain=**new** AppMain(); appMain.addFactorial();

}

/\*\*

\* 用while循环分别计算100以内的奇数的和、偶数的和

\*/

**void** caculate(){

**int** sumOdd=0;//奇数的和 **int** sumEven=0;//偶数的和 **for**(**int** i=0;i<100;i++){

**if**(i%2==0){

sumEven+=i;

}**else**{

sumOdd+=i;

}

}

System.*out*.println("100之内偶数的和："+sumEven); System.*out*.println("100之内奇数的和："+sumOdd);

}

/\* 用循环结构输出1000之内所有被5整除

\* 的数，并且每行最多输出3个

\*

\*/

**void** test2(){ **int** i=0; **int** j=0;

**while**(i<1000){ **if**(i%5==0){

System.*out*.println(i+"\t"); j++;

**if**(j%3==0){

System.*out*.println("\n"); j=0;

}

}

i++;

}

}

/\*\*

\* 输出九九乘法表

\*

\*/

**void** multiplicationTable(){

**for** (**int** i = 0; i < 10; i++) {//i是一个乘数

**for** (**int** j = 0; j <=i; j++) {//j是另外一个乘数 System.*out*.print(j+"\*"+i+"="+(i\*j<10?(" "+i\*j):i\*j)+" ");

}

System.*out*.println();

}

}

/\*\*

\* 编程求：1+（1+2）+（1+2+3）+. . .+（1+2+3+. . .+100）

\*/

**void** addAdd(){

**int** sum=0;//总和

**for**(**int** i=1;i<=100;i++){ **int** tempSum=0;//临时和 **for**(**int** j=1;j<=i;j++){

tempSum+=j;

}

sum+=tempSum;

}

System.*out*.println(sum);

}

/\*\*

\* 编程求：1！+2！+3！+4！+. . .+15！

\*

\*/

**private void** addFactorial() {

**long** result=0;

**for**(**int** i=1;i<=15;i++){

**int** temp=1;

**for**(**int** j=1;j<=i;j++){ temp\*=j;

}

result+=temp;

}

System.*out*.println(result);

}

}

# 第二章：数组

29.内存的划分：

1，寄存器。

2，本地方法区。

3，方法区。

4，栈内存。

存储的都是局部变量，而且变量所属的作用域一旦结束，该变量就自动释放。

5，堆内存。

存储是数组和对象(其实数组就是对象) 凡是new建立在堆中。 特点：

1，每一个实体都有首地址值。

2，堆内存中的每一个变量都有默认初始化值，根据类型的不同而不同。整数是0，小数0.0或者0.0f， boolean false char '\u0000'

3，垃圾回收机制。

30.数组中常见的两个异常：

System.out.println(arr[3]);//ArrayIndexOutOfBoundsException:

//当访问到数组中不存在的角标时，就会发生该异常。

arr = null;

System.out.println(arr[0]);//NullPointerException

//当引用型变量没有任何实体指向时，还在用其操作实体。就会发生该异常。

# 31.break不continue：

break:跳出。

break作用的范围：要么是switch语句，要么是循环语句。 记住：当break语句单独存在时，下面不要定义其他语句，因为执行不到。

break跳出所在的当前循环。 如果出现了循环嵌套，break想要跳出指定的循环，可以通过标号来完成。

continue:继续。

作用的范围：循环结构。 continue：结束本次循环，继续下次循环。 如果continue单独存在时，下面不要有任何语句，因为执行不到。

32.九九乘法表：

/\*

需求： 打印一个99乘法表。

思路：

1，定义一个类。

2，为了可以运行，该类中定义一个main函数。

3，将99惩罚表的代码定义在main函数中。

步骤： 1，用class关键字定义名称Demo99为的类。 2，定义main函数。

3，

1\*1=1

1\*2=2 2\*2=4

1\*3=3 2\*3=6 3\*3=9

分析一下99乘法表。

\*/

**class** Demo99

{

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

{

/\* 99乘法表的分析：

。。。

符合了大圈套小圈的思想。 就是要使用 for嵌套循环。

\*/

//int z = 1;

**for** (**int** x = 1;x<=9 ;x++ )

{

**for** (**int** y=1; y<=x ; y++)//

{

System.out.print(y+"\*"+x+"="+y\*x+"\t");

}

System.out.println();//

// z++;

}

}

}

# 33.输出一个由星\*组成的正方形：

**class** ForForDemo

{

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

{

//大圈套小圈思想。

/\*

for(int x=0; x<3; x++)

{

for(int y=0; y<4; y++)

{

System.out.println("ok");

}

}

\*/

/\*

\*\*\*\*\*

\*\*\*\*\*

\*\*\*\*\*

\*\*\*\*\*

\*/

**for**(**int** x=0; x<4; x++)//外循环控制的是行数

{

**for**(**int** y=0; y<5; y++)//内循环控制的是每一行的个数

{

System.out.print("\*");

}

System.out.println();

}

}

}

# 34.根据"大圈套小圈"思想输出丌同的图形：

**class** ForForTest

{

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

{

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| /\*  \*\*\*\*\* |  | | | |
| \*\*\*\*  \*\*\*  \*\*  \* |
| \*/ |
| /\*  int z = 5;  for (int x = 1; x<=5 ;x++ )//1-5 | 1-4 | 1-3//1-5 | 2-5 | 3-5 |
| { |  |  |  |  |

for (int y=1; y<=z ; y++ )

{

System.out.print("\*");

}

System.out.println(); z--;

}

\*/

/\*

int z = 1;

for (int x = 1; x<=5 ;x++ )//1-5 1-4 1-3//1-5 2-5 3-5

{

for (int y=z; y<=5 ; y++ )

{

System.out.print("$");

}

System.out.println(); z++;

}

\*/

**for**(**int** x=1; x<=5; x++)

{

**for**(**int** y=x; y<=5; y++)

{

System.out.print("\*");

}

System.out.println();

}

/\*

\*

\*\*

\*\*\*

\*\*\*\*

\*\*\*\*\*

\*/

System.out.println("----------");

**for** (**int** x=1; x<=5 ;x++ )

{

**for** (**int** y=1;y<=x ;y++ )

{

System.out.print("\*");

}

System.out.println();

}

System.out.println("----------");

/\*

54321

5432

543

54

5

\*/

**for** (**int** x=1; x<=5; x++ )

{

**for** (**int** y=5; y>=x ;y-- )

{

System.out.print(y);

}

System.out.println();

}

System.out.println("----------");

/\* 1

22

333

4444

55555

\*/

**for** (**int** x=1; x<=5 ; x++)

{

**for** (**int** y=1;y<=x ;y++ )

{

System.out.print(x);

}

System.out.println();

}

}

}

# 35.for嵌套循环"大圈套小圈"思想迚一步运用：

**class** ForForTest2

{

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

{

/\*

\* \* \* \* \*

-\* \* \* \*

--\* \* \*

---\* \*

----\*

\*/

**for**(**int** x=1; x<=5; x++)

{

**for**(**int** y=1; y<x; y++)

{

System.out.print(" ");

}

**for**(**int** z=x; z<=5; z++)

{

System.out.print("\* ");

}

System.out.println();

}

/\*

\*

\* \*

\* \* \*

\* \* \* \*

\* \* \* \* \*

\*/

}

}

36.凼数的定义格式：

修饰符 返回值类型 函数名(参数类型 形式参数1，参数类型 形式参数2，...)

{

执行语句; return 返回值;

}

特殊情况：

功能没有具体的返回值，这时return的后面直接用分号结束。 返回值类型怎么体现呢？因为没有具体值，所以不可以写具体的数据类型，在java中只能用一个关键字来 表示这种情况，关键字是：void。

总结：没有具体返回值时，返回值类型用void来表示。 注意：如果返回值类型是void，那么函数中的return语句可以省略不写。

37.凼数的重载：

1，同一个类，

2，同名。

3，参数个数不同or 参数类型不同、顺序不同。

4，函数重载和返回值类型无关。

5，java是严谨性语言，如果函数出现的调用的不确定性，会编译失败。

## 38.数组时常用容器乊一，对数组操作最基本的劢作就是存和叏，核心思想：就是对角标的操作。

39.获叏数组最值和选择、冒泡排序：

**import** java.util.\*;

**class** ArrayDemo4

{

//遍历数组的功能。

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

{

System.out.print("[");

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

{

**if**(x!=arr.length-1) System.out.print(arr[x]+", ");

### else

System.out.println(arr[x]+"]");

}

}

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

{

**int**[] arr = {34,19,11,109,3,56};

// int max = getMax\_2(arr);

// System.out.println("max="+max); printArray(arr);

// selectSort(arr);

// bubbleSort(arr);

// Arrays.sort(arr); selectSort\_2(arr);

printArray(arr);

}

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

{

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

}

/\*

冒泡排序。

\*/

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

{

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

{

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

{

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

{

swap(arr,y,y+1);

/\*

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

\*/

}

}

}

}

/\*

选择排序。

\*/

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

{

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

{

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

{

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

{

swap(arr,x,y);

/\*

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

\*/

}

}

}

}

**public static void** selectSort\_2(**int**[] arr)

{

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

{

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

**int** index = x;

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

{

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

{

num = arr[y]; index = y;

}

}

**if**(index!=x) swap(arr,x,index);

}

}

/\* 获取数组中的最大值。 思路：

1，需要进行比较。并定义变量记录住每次比较后较大的值。

2，对数组中的元素进行遍历取出，和变量中记录的元素进行比较。 如果遍历到的元素大于变量中记录的元素，就用变量记录住该大的值。

3，遍历结果，该变量记录就是最大值。

定义一个功能来是实现。 明确一，结果。

是数组中的元素。int .

明确二，未知内容。

数组.

\*/

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

{

//定义变量记录较大的值。

**int** maxElement = arr[0];//初始化为数组中的任意一个元素。

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

{

**if**(arr[x]>maxElement) maxElement = arr[x];

}

**return** maxElement;

}

**public static int** getMax\_2(**int**[] arr)

{

//定义变量记录较大的值。

**int** maxIndex = 0;//初始化为数组中任意一个角标。

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

{

**if**(arr[x]>arr[maxIndex]) maxIndex = x;

}

**return** arr[maxIndex];

}

}

# 40.二分法查找及普通查找方法运用：

**import** java.util.\*;

/\*

面试题： 给定一个有序的数组，如果往该数组中存储一个元素，并保证这个数组还是有序的， 那么个元素的存储的角标为如何获取。

{13,15,19,28,33,45,78,106};

\*/

**class** ArrayDemo5

{

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

{

//

// int[] arr = {4,1,8,7,3,8,2};

**int**[] arr = {13,15,19,28,33,45,78,106};

**int** index = halfSearch\_2(arr,5); System.out.println("index="+index);

**int** index1 = Arrays.binarySearch(arr,5);//如果存在返回的具体的角标位置，不存在返 回的是 -插入点-1

System.out.println("index1="+index1);

}

/\*

二分查找法。

\*/

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

{

**int** max,min,mid; min = 0;

max = arr.length-1; mid = (max+min)/2;

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

{

**if**(key>arr[mid]) min = mid + 1;

**else if**(key<arr[mid]) max = mid - 1;

**if**(max<min)

**return** -1;

mid = (max+min)/2;

}

**return** mid;

}

**public static int** halfSearch\_2(**int**[] arr,**int** key)

{

**int** max,min,mid; min = 0;

max = arr.length-1;

**while**(min<=max)

{

mid = (max+min)>>1;

**if**(key>arr[mid]) min = mid + 1;

**else if**(key<arr[mid]) max = mid - 1;

### else

**return** mid;

}

**return** -min-1;

}

/\*

数组常见功能：查找。

\*/

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

{

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

{

**if**(arr[x]==key) **return** x;

}

**return** -1;

}

}

41.将数组迚行反转：

/\*

给定一个数组，对其进行反转。

{3,1,6,5,8,2} -->

{2,8,5,6,1,3};

其实就是头尾元素的位置置换。

\*/

**class** ArrayTest

{

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

{

System.out.print("[");

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

{

**if**(x!=arr.length-1) System.out.print(arr[x]+", ");

### else

System.out.println(arr[x]+"]");

}

}

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

{

**int**[] arr = {4,1,8,7,3,8,2};

printArray(arr); reverseArray(arr); printArray(arr);

}

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

{

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

{

swap(arr,start,end);

}

}

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

{

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

}

}

42.数组中查表法的使用：

/\*

获取一个整数的16进制表现形式。

\*/

**class** ArrayTest2

{

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

{

toHex\_2(0);

}

// 0,1,2,3,4,5,6,7,8,9,A, B, C, D, E, F

// 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

/\*

什么时候使用数组呢？ 如果数据出现了对应关系，而且对应关系的一方是有序的数字编号。并作为角标使用。 这时就必须要想到数组的使用。

就可以将这些数据存储到数组中。 根据运算的结果作为角标直接去查数组中对应的元素即可。

这种方式：称为查表法。

\*/

**public static void** toHex\_2(**int** num)

{

**if**(num==0)

{

System.out.println("0");

### return ;

}

//定义一个对应关系表。

**char**[] chs = {'0','1','2','3',

'4','5','6','7',

'8','9','A','B',

'C','D','E','F'};

/\*

一会查表会查到比较的数据。 数据一多，就先存储起来，在进行操作。 所以定义一个数组。 临时容器。

\*/

**char**[] arr = **new char**[8]; **int** pos = arr.length;

**while**(num!=0)

{

**int** temp = num&15; arr[--pos] = chs[temp]; num = num >>> 4;

}

System.out.println("pos="+pos);

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

{

System.out.print(arr[x]);

}

}

**public static void** toHex\_1(**int** num)

{

//定义一个对应关系表。

**char**[] chs = {'0','1','2','3',

'4','5','6','7',

'8','9','A','B',

'C','D','E','F'};

**for**(**int** x=0 ; x<8; x++)

{

**int** temp = num & 15; System.out.print(chs[temp]); num = num >>> 4;

}

}

**public static void** toHex(**int** num)

{

**for**(**int** x=0; x<8; x++)

{

**int** temp = num & 15;

**if**(temp>9)

System.out.print((**char**)(temp-10+'A'));

### else

System.out.print(temp); num = num >>> 4;

}

/\*

int n1 = num & 15; System.out.println("n1="+n1);

num = num >>> 4; int n2 = num & 15;

System.out.println("n2="+n2);

\*/

}

}

43.查表运用：

**class** ArrayTest3

{

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

{

// toHex(26); toBinary(-6);

// toOctal(26); System.out.println(Integer.toBinaryString(-6));

}

//十进制-->十六进制。

**public static void** toHex(**int** num)

{

trans(num,15,4);

}

//十进制-->二进制。

**public static void** toBinary(**int** num)

{

trans(num,1,1);

}

//十进制-->八进制。

**public static void** toOctal(**int** num)

{

trans(num,7,3);

}

**public static void** trans(**int** num,**int** base,**int** offset)

{

**if**(num==0)

{

System.out.println("0");

### return ;

}

//定义一个对应关系表。

**char**[] chs = {'0','1','2','3',

'4','5','6','7',

'8','9','A','B',

'C','D','E','F'};

/\*

一会查表会查到比较的数据。 数据一多，就先存储起来，在进行操作。

所以定义一个数组。 临时容器。

\*/

**char**[] arr = **new char**[32]; **int** pos = arr.length;

**while**(num!=0)

{

**int** temp = num & base; arr[--pos] = chs[temp]; num = num >>> offset;

}

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

{

System.out.print(arr[x]);

}

System.out.println();

}

}

44.查表运用（数组容器）：

**class** ArrayTest4

{

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

{

String week = getWeek(71); System.out.println(week);

}

/\*

使用查表法。 星期。

String s = "abc"; int x = 4;

\*/

**public static** String getWeek(**int** num)

{

**if**(num>7 || num<1)

{

**return** "错误的星期";

}

"};

String[] weeks = {"","星期一","星期二","星期三","星期四","星期五","星期六","星期日

**return** weeks[num];

}

}

45.二维数组：

/\*

二维数组定义的格式。

\*/

**class** Array2Demo

{

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

{

// int[] arr = new int[3];

// System.out.println(arr);//[I@1fb8ee3 @左边是实体的类型。 @右边是实体的哈希值。

// int[][] arr = new int[3][2];//创建一个二维数组，该数组中有3个一维数组，每一个一维数 组中有2个元素。

// System.out.println(arr);//直接打印二维数组。 [[I@c17164

// System.out.println(arr[0]);//直接打印二维数组中的角标0的一维数组。 [I@1fb8ee3

// System.out.println(arr[0][0]);//直接打印二维数组中的角标0的一维数组中角标为0的元 素。 0

// int[][] arr = new int[3][];

// System.out.println(arr);//直接打印二维数组。 [[I@c17164

// System.out.println(arr[0]);//直接打印二维数组中的角标0的一维数组。null

// System.out.println(arr[0][0]);//直接打印二维数组中的角标0的一维数组中角标为0的元 素。 NullPointerException

// int[][] arr = new int[3][2];

// System.out.println(arr.length);//打印二维数组的长度。其实就是一维数组的个数。

// System.out.println(arr[1].length);//打印二维数组中角标为1一维数组的长度。

**int** sum = 0;

**int**[][] arr = {{3,1,7},{5,8,2,9},{4,1}};

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

{

**for**(**int** y=0; y<arr[x].length; y++)

{

// System.out.print(arr[x][y]+",");

sum += arr[x][y];

}

}

System.out.println("sum="+sum);

// 甲：30 59 28 17

// 乙；37 60 22 19

// int[] arr = {{30,59,28,17},{37,60,22,19}};

**int**[][][] arr = **new int**[3][2][4];

}

}

/\*

int[] x,y[]; int[] x; int[] y[];

a

x = y;

b

x = y[0];

c

x[0] = y[0];

d

x[0] = y[0][0];

e

x[0] = y;

\*/

46.数组综述： 数组的定义：数组是相同类型数据的集合，描述的是相同类型的若干个数据按照一定的先后顺序排

列组合而成，其中每一个数据称作一个数组元素，每个数组元素可以通过一个下标杢访问它们。 数组的四个基本特点：1.长度固定，一旦被创建它的长度就是丌可改发的；2.其元素类型必须是相 同类型，丌允许出现混合类型；3.数组中的元素可以是任何数据类型，包括基本数据类型和引用数

据类型；4.数组发量属亍引用类型，数组也可以看做是对象，数组中的每个元素相弼亍该对象的成

员发量，数组本身就是对象，Java中对象是在堆中的，因此数组无论保存原始类型还是其他对象类 型，数组本身是在堆中的。

数组的下标合法区间是[0,length-1]。 数组的拷贝：System.arrayCopy（源数组，从哪开始，目标数组，从哪开始贴，粘几个）。 数组排序：Arrays.sort（被排序的数组）。 二分法查找：Arrays.binarySearch（哪个数组，数组中的什么元素）。 填充：Arrays.fill（a， 2， 4, 100）。//将数组a中2到4的索引的元素替换为100

获叏数组的长度：数组.length，如a.length，获叏数组a的元素个数；a[0].length，表示获叏二维 数组中第一个数组的长度。 数组的遍历：可以使用for循环戒者for嵌套循环（对亍二维数组），也可以使用增强for循环杢对数 组迚行遍历，增强for循环格式：for（变量类型 变量名：被遍历的数组）

第三章：面向对象程序开収

47.面向对象简述：

/\* 用java语言对现实生活中的事物进行描述。 通过类的形式来体现的。

怎么描述呢？ 对于事物描述通常只关注两方面。 一个是属性，一个是行为。

只要明确该事物的属性和行为并定义在类中即可。 对象：其实就是该类事物实实在在存在的个体。 类与对象之间的关系？

类：事物的描述。

对象：该类事物的实例。在java中通过new来创建的。

\*/

/\*

描述小汽车 分析：

1，属性。

轮胎数。 颜色。

2，行为。

运行。

定义类其实就是在定义类中的成员。 成员：成员变量<-->属性，成员函数<-->行为。

成员变量和局部变量的区别：

1， 成员变量定义在类中，整个类中都可以访问。

局部变量定义在函数，语句，局部代码块中，只在所属的区域有效。

2， 成员变量存在于堆内存的对象中。 局部变量存在于栈内存的方法中。

3， 成员变量随着对象的创建而存在，随着对象的消失而消失。

局部变量随着所属区域的执行而存在，随着所属区域的结束而释放。

4， 成员变量都有默认初始化值。 局部变量没有默认初始化值。

\*/

**class** Car

{

**int** num; String color;

**void** run()

{

//int num = 10; System.out.println(num+"..."+color);

}

}

**class** CarDemo

{

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

{

//在计算机中创建一个car的实例。通过new关键字。

// Car c = new Car();// c就是一个类类型的引用变量，指向了该类的对象。

// c.num = 4;

// c.color = "red";

// c.run();//要使用对象中的内容可以通过 对象.成员 的形式来完成调用。

// Car c1 = new Car();

// c1.num = 4;

// c1.color = "red";

// Car c2 = new Car();

// c2.num = 4;

// c2.color = "red";

// Car c1 = new Car();

// Car c2 = new Car();

// show(c1);

// show(c2);

/\*

匿名对象。没有名字的对象 。

new Car();//匿名对象。其实就是定义对象的简写格式。

Car c = new Car(); c.run();

new Car().run();

1，当对象对方法仅进行一次调用的时候，就可以简化成匿名对象。 new Car().num = 5;

new Car().color = "green"; new Car().run();

2，匿名对象可以作为实际参数进行传递。

\*/

// Car c1 = new Car();

// show(c1); show(**new** Car());

}

//汽车改装厂。

**public static void** show(Car c)//类类型的变量一定指向对象。要不就是null。

{

c.num = 3; c.color = "black";

System.out.println(c.num+"..."+c.color);

}

}

# 48.封装（private）使用：

/\* 人： 属性：

年龄。

行为：

说话。

\*/

/\* private:私有，是一个权限修饰符。用于修饰成员。

私有的内容只在本类中有效。

注意：私有仅仅是封装的一种体现而已。

\*/

class Person

{

private /\*私有\*/int age;

public void setAge(int a)//setXxx getXxx

{

age = a;

}

public int getAge()

{

return age;

}

/\*

public void haha(int a)

{

if(a>0 && a<130)

{

age = a; speak();

}

else

System.out.println("错误的数据");

}

\*/

void speak()

{

System.out.println("age="+age);

}

}

class PersonDemo

{

public static void main(String[] args)

{

Person p = new Person();

// p.age = -20; p.haha(-20);

// p.speak();

}

public static void selectSort(int[] arr){}

private static void swap(int[] arr,int a,int b){}

}

# 49.基本数据类型和引用数据类型参数的传递：

//基本数据类型参数传递

**class** Demo

{

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

{

**int** x = 3; show(x);

System.out.println("x="+x);

}

**public static void** show(**int** x)

{

x = 4;

}

}

//引用数据类型参数传递

**class** Demo

{

**int** x = 3;

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

{

Demo d = **new** Demo();

d.x = 9; show(d);

System.out.println(d.x);

}

**public static void** show(Demo d)

{

d.x = 4;

}

}

50.构造凼数：

**class** Person

{

**private** String name;

**private int** age;

//定义一个Person类的构造函数。 Person()//构造函数，而且是空参数的。

{

name = "baby"; age = 1;

System.out.println("person run");

}

//如果有的孩子一出生就有名字。 Person(String n)

{

name = n;

}

**public void** setName(String n)

{

name = n;

}

Person(String n,**int** a)

{

name = n; age = a;

}

**public void** speak()

{

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

}

}

/\*

//构造函数：构建创造对象时调用的函数。作用：可以给对象进行初始化。 创建对象都必须要通过构造函数初始化。

一个类中如果没有定义过构造函数，那么该类中会有一个默认的空参数构造函数。 如果在类中定义了指定的构造函数，那么类中的默认构造函数就没有了。

一般函数和构造函数什么区别呢？

构造函数：对象创建时，就会调用与之对应的构造函数，对对象进行初始化。 一般函数：对象创建后，需要函数功能时才调用。

构造函数：对象创建时，会调用只调用一次。 一般函数：对象创建后，可以被调用多次。

什么时候定义构造函数呢？ 在描述事物时，该事物一存在就具备的一些内容，这些内容都定义在构造函数中。 构造函数可以有多个，用于对不同的对象进行针对性的初始化. 多个构造函数在类中是以重载的形式来体现的。

细节：

1，构造函数如果完成了set功能。set方法是否需要。

2，一般函数不能直接调用构造函数。

3，构造函数如果前面加了void就变成了一般函数。 4，构造函数中是有return语句的。

\*/

**class** ConsDemo

{

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

{

Person p = **new** Person();

// p.speak();

Person p1 = **new** Person("旺财"); p1.setName("旺旺");

p1.speak();

Person p2 = **new** Person("小强",10); p2.speak();

}

}

51.主凼数解剖：

/\*

public static void main(String[] args)

主函数特殊之处：

1，格式是固定的。

2，被jvm所识别和调用。

public:因为权限必须是最大的。 static:不需要对象的,直接用主函数所属类名调用即可。 void：主函数没有具体的返回值。 main:函数名，不是关键字，只是一个jvm识别的固定的名字。

String[] args:这是主函数的参数列表，是一个数组类型的参数，而且元素都是字符串类型。

\*/

**class** MainDemo

{

**public static void** main(String[] args) //new String[0]

{

/\*\*/

// System.out.println(args);//[Ljava.lang.String;@c17164 System.out.println(args.length);

**for**(**int** x=0; x<args.length; x++) System.out.println(args[x]);

}

}

52.静态代码块：

/\*

静态代码块。 随着类的加载而执行。而且只执行一次。

作用：

用于给类进行初始化。

\*/

**class** StaticCode

{

**static int** num ;

### static

{

num = 10;

// num \*=3; System.out.println("hahahah");

}

StaticCode(){}

**static void** show()

{

System.out.println(num);

}

}

**class** Person

{

**private** String name;

{//构造代码块。可以给所有对象进行初始化的。 System.out.println("constructor code ");

// cry();

}

### static

{

System.out.println("static code");

}

Person()//是给对应的对象进行针对性的初始化。

{

name = "baby";

// cry();

}

Person(String name)

{

**this**.name = name;

// cry();

}

**public void** cry()

{

System.out.println("哇哇");

}

**public void** speak()

{

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

}

**static void** show()

{

System.out.println("show run");

}

}

**class** StaticCodeDemo

{

### static

{

// System.out.println("a");

}

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

{

// Person p = null;

// p.speak();

// Person.show();

// Person p1 = new Person();

// p1.speak();

// Person p2 = new Person("旺财");

// p2.speak();

// new Person();

// new StaticCode().show();

// new StaticCode().show();

// StaticCode.show();

// System.out.println("b");

}

}

53.static：

**class** Person

{

String name;//成员变量，实例变量

**static** String country = "CN";//静态变量。类变量

**public void** show()

{

System.out.println(Person.country+":"+**this**.name);

}

}

/\* static的特点：

1，static是一个修饰符，用于修饰成员。

2，static修饰的成员被所有的对象所共享。 3，static优先于对象存在，因为static的成员随着类的加载就已经存在了。 4，static修饰的成员多了一种调用方式，就可以直接被类名所调用 。 类名.静态成员 。

5，static修饰的数据是共享数据，对象中的存储的是特有数据。

成员变量和静态变量的区别？

1，两个变量的生命周期不同。 成员变量随着对象的创建而存在，随着对象的被回收而释放。 静态变量随着类的加载而存在，随着类的消失而消失。

2，调用方式不同。 成员变量只能被对象调用。

静态变量可以被对象调用，还可以被类名调用。

3，别名不同。

成员变量也称为实例变量。 静态变量称为类变量。

4，数据存储位置不同。 成员变量数据存储在堆内存的对象中，所以也叫对象的特有数据. 静态变量数据存储在方法区(共享数据区)的静态区，所以也叫对象的共享数据.

静态使用的注意事项： 1，静态方法只能访问静态成员。(非静态既可以访问静态，又可以访问非静态) 2，静态方法中不可以使用this或者super关键字。

3，主函数是静态的。

\*/

**class** StaticDemo

{

**int** num = 4;

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

{

|  |  |  |
| --- | --- | --- |
| //  // |  | Person p = new Person(); p.name = "小强"; |
| // |  | p.show(); |
| // |  | System.out.println(p.country); |
| // |  | System.out.println(Person.country); |
| // |  | Person.show(); |
|  |  | **new** StaticDemo().show(); |
|  | } |  |

**public void** show()

{

System.out.println(num);

}

}

54.静态的使用：

/\* 静态什么时候用？ 1，静态变量。

当分析对象中所具备的成员变量的值都是相同的 。

这时这个成员就可以被静态修饰。 只要数据在对象中都是不同的，就是对象的特有数据，必须存储在对象中，是非静态的。 如果是相同的数据，对象不需要做修改，只需要使用即可，不需要存储在对象中，定义成静态的。

2，静态函数。

函数是否用静态修饰，就参考一点，就是该函数功能是否有访问到对象中的特有数据。 简单点说，从源代码看，该功能是否需要访问非静态的成员变量，如果需要，该功能就是非静态的。 如果不需要，就可以将该功能定义成静态的。当然，也可以定义成非静态， 但是非静态需要被对象调用，而仅创建对象调用非静态的 没有访问特有数据的方法，该对象的创建是没有意义。

\*/

**class** Demo

{

**int** age ;

**static int** num = 9; Demo(**int** age)

{

**this**.age = age;

}

**public static void** speak()

{

System.out.println(num);

}

**public void** show()

{

System.out.println(age);

}

}

**class** StaticDemo3

{

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

{

// Demo d = new Demo(30);

// d.speak();

Demo.speak();

// System.out.println("Hello World!");

}

}

55.this关键字：

/\*

当成员变量和局部变量重名，可以用关键字this来区分。

this : 代表对象。代表哪个对象呢？当前对象。 this就是所在函数所属对象的引用。 简单说：哪个对象调用了this所在的函数，this就代表哪个对象。

this也可以用于在构造函数中调用其他构造函数。 注意：只能定义在构造函数的第一行。因为初始化动作要先执行。

\*/

**class** Person

{

**private** String name; **private int** age; Person()

{

name = "baby"; age = 1;

System.out.println("person run");

}

Person(String name)

{

**this**(); **this**.name = name;

}

Person(String name,**int** age)

{

**this**.name = name;

**this**.age = age;

}

**public void** speak()

{

System.out.println(**this**.name+":"+**this**.age);

}

/\*

判断是否是同龄人。

\*/

**public boolean** compare(Person p)

{

/\* if(this.age==p.age)

return true; else

return false;

\*/

**return this**.age==p.age;

}

}

**class** ThisDemo

{

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

{

Person p1 = **new** Person("aa",30);// Person p2 = **new** Person("zz",12);

p2.compare(p1);

// new Person();

// Person p = new Person("旺财",30);

// p.speak();

// Person p1 = new Person("小强");

// p1.speak();

}

}

56.建立数组工具（练习）：

/\*\*

建立一个用于操作数组的工具类，其中包含着常见的对数组操作的函数如：最值，排序等 。

### @author 张三

**@version** V1.0

\*/

**public class** ArrayTool

{

**private** ArrayTool(){}//该类中的方法都是静态的，所以该类是不需要的创建对象的。为了保证不让

//其他成创建该类对象

//可以将构造函数私有化。

/\*\*

获取整型数组的最大值。

**@param** arr 接收一个元素为int类型的数组。

**@return** 该数组的最大的元素值

\*/

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

{

**int** maxIndex = 0;

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

{

**if**(arr[x]>arr[maxIndex]) maxIndex = x;//

}

}

/\*\*

**return** arr[maxIndex];

对数组进行选择排序。

**@param** arr 接收一个元素为int类型的数组。

\*/

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

{

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

{

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

{

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

swap(arr,x,y);

}

}

}

/\*

用于给数组进行元素的位置置换。

@param arr 接收一个元素为int类型的数组。

@param a

@param b

\*/

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

{

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

}

/\*\*

获取指定的元素在指定数组中的索引.

**@param** arr 接收一个元素为int类型的数组。

**@param** key 要找的元素。

**@return** 返回该元素第一次出现的位置，如果不存在返回-1.

\*/

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

{

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

{

**if**(arr[x]==key) **return** x;

}

**return** -1;

}

/\*\*

将int数组转换成字符串。格式是：[e1,e2,...]

**@param** arr 接收一个元素为int类型的数组。

**@return** 返回该数组的字符串表现形式。

\*/

**public static** String arrayToString(**int**[] arr)

{

String str = "[";

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

{

**if**(x!=arr.length-1)

str = str + arr[x]+", ";

### else

str = str + arr[x]+"]";

}

**return** str;

}

}

# 57.继承：

关于成员变量（案例一）：

/\* 在子父类中，成员的特点体现。 1，成员变量。

2，成员函数。

3，构造函数。

\*/

//1，成员变量。

/\* 当本类的成员和局部变量同名用this区分。 当子父类中的成员变量同名用super区分父类。

this和super的用法很相似。

this:代表一个本类对象的引用。 super：代表一个父类空间。

\*/ class Fu

{

private int num = 4;

public int getNum()

{

return num;

}

}

class Zi extends Fu

{

private int num = 5;

void show()

{

System.out.println(this.num+"....."+super.getNum());

}

}

class ExtendsDemo2

{

public static void main(String[] args)

{

Zi z = new Zi(); z.show();

}

}

关于成员函数（案例二）：

//成员函数。

/\* 当子父类中出现成员函数一模一样的情况，会运行子类的函数。 这种现象，称为覆盖操作。这时函数在子父类中的特性。 函数两个特性：

1，重载。同一个类中。overload 2，覆盖。子类中。覆盖也称为重写，覆写。override

覆盖注意事项：

1，子类方法覆盖父类方法时，子类权限必须要大于等于父类的权限。

2，静态只能覆盖静态，或被静态覆盖。

\*/

**class** Fu

{

**public static void** show()

{

System.out.println("fu show run");

}

}

**class** Zi **extends** Fu

{

**public static void** show()

{

System.out.println("Zi show run");

}

}

**class** ExtendsDemo3

{

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

{

// Zi z = new Zi();

// z.show();

NewPhone p = **new** NewPhone(); p.show();

p.call();

}

}

/\*

什么时候使用覆盖操作? 当对一个类进行子类的扩展时，子类需要保留父类的功能声明， 但是要定义子类中该功能的特有内容时，就使用覆盖操作完成.

\*/

**class** honeP

{

**void** call()

{}

**void** show()

{

System.out.println("number");

}

}

**class** NewPhone **extends** Phone

{

**void** show()

{

System.out.println("name"); System.out.println("pic"); **super**.show();

}

}

58.单例设计模式：

/\*

设计模式：对问题行之有效的解决方式。其实它是一种思想。

1,单例设计模式。 解决的问题：就是可以保证一个类在内存中的对象唯一性。

必须对于多个程序使用同一个配置信息对象时，就需要保证该对象的唯一性。 如何保证对象唯一性呢？

1，不允许其他程序用new创建该类对象。

2，在该类创建一个本类实例。

3，对外提供一个方法让其他程序可以获取该对象。

步骤：

1，私有化该类构造函数。

2，通过new在本类中创建一个本类对象。

3，定义一个公有的方法，将创建的对象返回。

\*/

//饿汉式

**class** Single //类一加载，对象就已经存在了。

{

**private static** Single s = **new** Single();

**private** Single(){}

**public static** Single getInstance()

{

**return** s;

}

}

//懒汉式

**class** Single2 //类加载进来，没有对象，只有调用了getInstance方法时，才会创建对象。

//延迟加载形式。

{

**private static** Single2 s = **null**; **private** Single2(){}

**public static** Single2 getInstance()

{

**if**(s==**null**)

s = **new** Single2();

**return** s;

}

}

**class** SingleDemo

{

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

{

Single s1 = Single.getInstance(); Single s2 = Single.getInstance();

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

// Single ss = Single.s;

// Test t1 = new Test();

// Test t2 = new Test();

Test t1 = Test.getInstance(); Test t2 = Test.getInstance();

t1.setNum(10); t2.setNum(20);

System.out.println(t1.getNum()); System.out.println(t2.getNum());

}

}

**class** Test

{

**private int** num;

**private static** Test t = **new** Test();

**private** Test(){}

**public static** Test getInstance()

{

**return** t;

}

**public void** setNum(**int** num)

{

**this**.num = num;

}

**public int** getNum()

{

**return** num;

}

}

59.面向对象总结：

面向对象：

1，面向对象和面向过程思想。

面向对象强调的是对象实例。 面向过程强调的是动作。 对象将动作进行封装。

在问题领域中，我们先去找的都是涉及的对象， 然后分析对象的属性和行为。

2，面向对象的特点。

1，是一种常见思想。

2，复杂变简单。

3，执行者变指挥者。 举例：面试的例子。

3，类与对象的关系。 类：就是对事物的描述，其实类就是将对象共性的内容进行抽取。 对象：就是该类事物实实在在存在个体，在java中通过new来完成创建的，

堆内存的对象主要用于封装对象中的特有数据。

4，类中的成员： 成员变量：事物的属性， 成员函数：事物的行为。

成员变量和局部变量的区别？ 答：

如何使用成员，只要建立该类对象，并通过 "对象.对象成员" 就可以完成调用

5，封装。

隐藏实现细节，并对外提供公共的访问方式。 函数或者类都是一个封装体。

特点：

1，隔离的变量。

2，便于使用。

3，提高复用。

4，提高安全性。

举例：机箱的例子 体现之一：私有。

私有在源代码中的使用就是在本类中有效。

通常会将成员变量xxx私有化，并对外提供对应的setXxx getXxx方法对其进行访问。 其实目的就是成员变量访问进行控制。 让成员的访问具备可控性，提高了程序的健壮性。 私有仅仅是封装的体现形式之一而已。

自定义一个Person类。动手。

6，构造函数。

写法特点：

1，函数名和类名相同。

2，不需要定义返回值类型

3，没有具体的返回值。但是有return语句，用于结束构造函数。

使用特点：

1，定义类中，有默认的空参数构造函数。如果已经自定义，那么默认就没有了。

2，构造函数在类有多个，是重载形式体现出来的。

构造函数的作用： 用于给对象进行针对性的初始化。 构造函数和一般函数的区别？

1，

2，

3， 什么时候使用构造函数呢？

当对象创建时就需要一些内容(数据和行为)，那么这些内容都定义在构造函数中。

7，this关键字. this：代表的是对象。哪个对象调用了this所在的函数，this就代表哪个对象。

用法体现

1，当成员变量和局部变量同名时，可以用this区别。

2，当构造函数中调用本类其他构造函数时，可以用this完成。 this(实际参数);

这种调用必须定义在构造函数的第一行。初始化动作先完成。 应用：只要在定义功能时，用到了本类对象，那么就使用this来表示。

8，static关键字：

特点：

1，修饰成员。，

2，随着类加载，随着类消失。

3，优先于对象。

4，用类名直接访问

使用注意事项: 1，静态方法只能访问静态，静态有访问局限性。 2，静态方法中不能有this super关键字。 3，主函数是静态的。

静态变量和成员变量的区别？

1，

2，

3，

4， 什么时候使用静态？

1，当成员变量的数据各个对象都相同时，可以用static修饰，让多个对象共享。

2，函数如果访问了特有数据(非静态成员变量),该函数是非静态的。 函数如果没有访问特有数据，那么该函数就可以静态修饰。

如果类中的功能都是静态的，那么该类创建对象是没有意义的，所以构造函数需要私有化。

9，代码块。

1，局部代码快。 对局部变量的生命周期进行控制。

2，构造代码块。 对所有对象进行初始化。

3，静态代码块。 对类进行初始化。

10，单例设计模式：

1，解决的问题：保证类在内存的对象唯一性。

2，思路：

3，步骤：

4，两种方式的区别？懒汉式(延迟加载方式)，饿汉式 涉及的内存图要会画！必须的。 60.抽象类简述：

/\*

抽象类： 抽象：笼统，模糊，看不懂！不具体。

特点：

1，方法只有声明没有实现时，该方法就是抽象方法，需要被abstract修饰。 抽象方法必须定义在抽象类中。该类必须也被abstract修饰。

2，抽象类不可以被实例化。为什么？因为调用抽象方法没意义。

3，抽象类必须有其子类覆盖了所有的抽象方法后，该子类才可以实例化。 否则，这个子类还是抽象类。

1，抽象类中有构造函数吗？ 有，用于给子类对象进行初始化。

2，抽象类可以不定义抽象方法吗？

可以的。 但是很少见，目的就是不让该类创建对象。AWT的适配器对象就是这种类。 通常这个类中的方法有方法体，但是却没有内容。

abstract class Demo

{

void show1()

{}

void show2()

{}

}

3，抽象关键字不可以和那些关键字共存? private 不行

static 不行

final 不行

4，抽象类和一般类的异同点。 相同点：

抽象类和一般类都是用来描述事物的，都在内部定了成员。 不同：

1，一般类有足够的信息描述事物。 抽象类描述事物的信息有可能不足。

2，一般类中不能定义抽象方法，只能定非抽象方法。 抽象类中可定义抽象方法，同时也可以定义非抽象方法。

3，一般类可以被实例化。 抽象类不可以被实例化。

5，抽象类一定是个父类吗？ 是的。因为需要子类覆盖其方法后才可以对子类实例化。

\*/

**abstract class** Demo

{

**abstract** /\*抽象\*/ **void** show();

}

/\*

class DemoA extends Demo

{

void show()

{

System.out.println("demoa show");

}

}

class DemoB extends Demo

{

void show()

{

System.out.println("demob show");

}

}

\*/

### abstract class 犬科

{

**abstract void** 吼叫();

}

**class** 狗 **extends** 犬科

{

**void** 吼叫()

{

System.out.println("汪汪");

}

}

**class** 狼 **extends** 犬科

{

**void** 吼叫()

{

System.out.println("嗷嗷");

}

}

**class** AbstractDemo

{

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

{

System.out.println("Hello World!");

}

}

61.继承&构造凼数：

/\*

子父类中的构造函数的特点。

在子类构造对象时，发现，访问子类构造函数时，父类也运行了。 为什么呢？

原因是：在子类的构造函数中第一行有一个默认的隐式语句。 super();

子类的实例化过程：子类中所有的构造函数默认都会访问父类中的空参数的构造函数。

为什么子类实例化的时候要访问父类中的构造函数呢？ 那是因为子类继承了父类，获取到了父类中内容(属性)，所以在使用父类内容之前， 要先看父类是如何对自己的内容进行初始化的。

所以子类在构造对象时，必须访问父类中的构造函数。 为什么完成这个必须的动作，就在子类的构造函数中加入了super()语句。

如果父类中没有定义空参数构造函数，那么子类的构造函数必须用super明确要调用 父类中哪个构造函数。同时子类构造函数中如果使用this调用了本类构造函数时， 那么super就没有了，因为super和this都只能定义第一行。所以只能有一个。 但是可以保证的是，子类中肯定会有其他的构造函数访问父类的构造函数。

注意：supre语句必须要定义在子类构造函数的第一行。因为父类的初始化动作要先完成。

\*/

**class** Fu

{

**int** num ; Fu()

{

num =10;

System.out.println("A fu run");

}

Fu(**int** x)

{

System.out.println("B fu run..."+x);

}

}

**class** Zi **extends** Fu

{

**int** num; Zi()

{

//super();//调用的就是父类中的空参数的构造函数。

System.out.println("C zi run"+num);

}

Zi(**int** x)

{

**this**();

//super();

// super(x);

System.out.println("D zi run "+x);

}

}

**class** ExtendsDemo4

{

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

{

**new** Zi(6);

}

}

**class** Demo//extends Object

{

/\*

Demo()

{

super(); return;

}

\*/

}

# 62.一个对象实例化的过程（绅节）：

**class** Fu

{

Fu()

{

**super**();

show();

### return;

}

**void** show()

{

System.out.println("fu show");

}

}

**class** Zi **extends** Fu

{

**int** num = 8; Zi()

{

**super**();

//-->通过super初始化父类内容时，子类的成员变量并未显示初始化。等super()父类初始化完毕

//后，

//才进行子类的成员变量显示初始化。

System.out.println("zi cons run...."+num);

### return;

}

**void** show()

{

System.out.println("zi show..."+num);

}

}

**class** ExtendsDemo5

{

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

{

Zi z = **new** Zi(); z.show();

}

}

/\*

一个对象实例化过程：

Person p = new Person(); 1，JVM会读取指定的路径下的Person.class文件，并加载进内存，

并会先加载Person的父类(如果有直接的父类的情况下).

2，在堆内存中的开辟空间，分配地址。

3，并在对象空间中，对对象中的属性进行默认初始化。

4，调用对应的构造函数进行初始化。

5，在构造函数中，第一行会先到调用父类中构造函数进行初始化。

6，父类初始化完毕后，在对子类的属性进行显示初始化。

7，在进行子类构造函数的特定初始化。

8，初始化完毕后，将地址值赋值给引用变量.

\*/

63.final关键字：

//继承弊端：打破了封装性。

/\* final关键字：

1，final是一个修饰符，可以修饰类，方法，变量。

2，final修饰的类不可以被继承。

3，final修饰的方法不可以被覆盖。

4，final修饰的变量是一个常量，只能赋值一次。 为什么要用final修饰变量。其实在程序如果一个数据是固定的， 那么直接使用这个数据就可以了，但是这样阅读性差，所以它该数据起个名称。 而且这个变量名称的值不能变化，所以加上final固定。 写法规范：常量所有字母都大写，多个单词，中间用\_连接。

\*/

**class** Fu

{

**void** method()

{

//调用了底层系统的资源。

}

}

**class** Zi **extends** Fu

{

**public static final double** MY\_PI = 3.14;

**static final int** x = 7;

**void** method()

{

// final int x = 9;

// x = 9;

**final int** NUMBER = 9;

System.out.println(MY\_PI);

}

}

**class** FinalDemo

{

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

{

System.out.println("Hello World!");

}

}

# 64.接口（interface）：

/\*

abstract class AbsDemo

{

abstract void show1(); abstract void show2();

}

当一个抽象类中的方法都是抽象的时候，这时可以将该抽象类用 另一种形式定义和表示，就是接口 interface。

\*/

//定义接口使用的关键字不是class，是interface.

/\* 对于接口当中常见的成员：而且这些成员都有固定的修饰符。 1，全局常量: public static final

2，抽象方法。public abstract

由此得出结论，接口中的成员都是公共的权限.

\*/

interface Demo

{

public static final int NUM = 4;

public abstract void show1(); public abstract void show2();

}

//类与类之间是继承关系，类与接口直接是实现关系。

/\*

接口不可以实例化。

只能由实现了接口的子类并覆盖了接口中所有的抽象方法后，该子类才可以实例化。 否则，这个子类就是一个抽象类。

\*/

class DemoImpl implements /\*实现\*/Demo

{

public void show1()

{}

public void show2()

{

}

}

/\* 在java中不直接支持多继承，因为会出现调用的不确定性。 所以java将多继承机制进行改良，在java中变成了多实现。

一个类可以实现多个接口。

\*/ interface A

{

public void show();

}

interface Z

{

public int add(int a,int b);

}

class Test implements A,Z//多实现

{

public int add(int a,int b)

{

return a+b+3;

}

/\*\*/

public void show(){}

}

/\*

一个类在继承另一个类的同时，还可以实现多个接口。

\*/ class Q

{

public void method()

{}

}

abstract class Test2 extends Q implements A,Z

{

}

/\*

接口的出现避免了单继承的局限性。

\*/

interface CC

{

void show();

}

interface MM

{

void method();

}

interface QQ extends CC,MM//接口与接口之间是继承关系，而且接口可以多继承。

{

void function();

}

class WW implements QQ

{

//覆盖3个方法。

public void show(){} public void method(){} public void function(){}

}

class InterfaceDemo

{

public static void main(String[] args)

{

Test t = new Test(); t.show();

// DemoImpl d = new DemoImpl();

// System.out.println(d.NUM);

// System.out.println(DemoImpl.NUM);

// System.out.println(Demo.NUM);

}

}

65.接口和抽象类异同点：

/\*

抽象类和接口的异同点：

相同点：

都是不断向上抽取而来的。

不同点：

1，抽象类需要被继承，而且只能单继承。 接口需要被实现，而且可以多实现。

2，抽象类中可以定义抽象方法和非抽象方法，子类继承后，可以直接使用非抽象方法。 接口中只能定义抽象方法，必须由子类去实现。

3，抽象类的继承，是is a关系，在定义该体系的基本共性内容。 接口的实现是 like a 关系，在定义体系额外功能。

犬按功能分：有导盲犬，搜爆犬。

abstract class 犬

{

abstract void 吼叫();

}

//abstract class 导盲

interface 导盲

{

abstract void 导盲();

}

class 导盲犬 extends 犬 implements 导盲

{

public void 吼叫()

{

}

public void 导盲(){}

}

//在不同的问题领域中，有不同的分析方式。 学员：

学习。 抽烟学员

烟民。

\*/

**class** InterfaceDemo

{

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

{

System.out.println("Hello World!");

}

}

66.多态简述：

/\*

对象的多态性。

class 动物

{}

class 猫 extends 动物

{}

class 狗 extends 动物

{}

猫 x = new 猫();

动物 x = new 猫();//一个对象，两种形态。

猫这类事物即具备者猫的形态，又具备着动物的形态。 这就是对象的多态性。

简单说：就是一个对象对应着不同类型. 多态在代码中的体现：

父类或者接口的引用指向其子类的对象。

多态的好处：

提高了代码的扩展性，前期定义的代码可以使用后期的内容。

多态的弊端：

前期定义的内容不能使用(调用)后期子类的特有内容。

多态的前提：

1，必须有关系，继承，实现。

2，要有覆盖。

\*/

**abstract class** Animal

{

**abstract void** eat();

}

**class** Dog **extends** Animal

{

**void** eat()

{

System.out.println("啃骨头");

}

**void** lookHome()

{

System.out.println("看家");

}

}

**class** Cat **extends** Animal

{

**void** eat()

{

System.out.println("吃鱼");

}

**void** catchMouse()

{

System.out.println("抓老鼠");

}

}

**class** Pig **extends** Animal

{

**void** eat()

{

System.out.println("饲料");

}

**void** gongDi()

{

System.out.println("拱地");

}

}

**class** DuoTaiDemo

{

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

{

// Cat c = new Cat();

// c.eat();

// c.catchMouse();

Animal a = **new** Cat(); //自动类型提升，猫对象提升了动物类型。但是特有功能无法s访问。

//作用就是限制对特有功能的访问。

//专业讲：向上转型。将子类型隐藏。就不用使用子类的特有方法。

// a.eat();

//如果还想用具体动物猫的特有功能。

//你可以将该对象进行向下转型。

// Cat c = (Cat)a;//向下转型的目的是为了使用子类中的特有方法。

// c.eat();

// c.catchMouse();

// 注意：对于转型，自始自终都是子类对象在做着类型的变化。

// Animal a1 = new Dog();

// Cat c1 = (Cat)a1;//ClassCastException

/\*

Cat c = new Cat();

// Dog d = new Dog();

// c.eat(); method(c);

// method(d);

// method(new Pig());

\*/

method(**new** Dog());

}

**public static void** method(Animal a)//Animal a = new Dog();

{

a.eat();

**if**(a **instanceof** Cat)//instanceof：用于判断对象的具体类型。只能用于引用数据类型判断

// //通常在向下转型前用于健壮性的判断。

{

Cat c = (Cat)a; c.catchMouse();

}

**else if**(a **instanceof** Dog)

{

Dog d = (Dog)a; d.lookHome();

}

### else

{

}

}

/\*

public static void method(Cat c)

{

c.eat();

}

public static void method(Dog d)

{

}

\*/

}

67.多态时各组成的发化：

/\*

多态时，成员的特点：

1，成员变量。

编译时：参考引用型变量所属的类中的是否有调用的成员变量，有，编译通过，没有，编译失败。 运行时：参考引用型变量所属的类中的是否有调用的成员变量，并运行该所属类中的成员变量。 简单说：编译和运行都参考等号的左边。哦了。

作为了解。

2，成员函数(非静态)。 编译时：参考引用型变量所属的类中的是否有调用的函数。有，编译通过，没有，编译失败。 运行时：参考的是对象所属的类中是否有调用的函数。

简单说：编译看左边，运行看右边。 因为成员函数存在覆盖特性。

3，静态函数。

编译时：参考引用型变量所属的类中的是否有调用的静态方法。 运行时：参考引用型变量所属的类中的是否有调用的静态方法。 简单说，编译和运行都看左边。 其实对于静态方法，是不需要对象的。直接用类名调用即可。

\*/

**class** Fu

{

// int num = 3;

**void** show()

{

System.out.println("fu show");

}

**static void** method()

{

System.out.println("fu static method");

}

}

**class** Zi **extends** Fu

{

// int num = 4;

**void** show()

{

System.out.println("zi show");

}

**static void** method()

{

System.out.println("zi static method");

}

}

**class** DuoTaiDemo3

{

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

{

Fu.method();

Zi.method();

Fu f = **new** Zi();//

// f.method();

// f.show();

// System.out.println(f.num);

// Zi z = new Zi();

// System.out.println(z.num);

}

}

68.内部类：

/\*

内部类访问特点：

1，内部类可以直接访问外部类中的成员。

2，外部类要访问内部类，必须建立内部类的对象。 一般用于类的设计。

分析事物时，发现该事物描述中还有事物，而且这个事物还在访问被描述事物的内容。 这时就是还有的事物定义成内部类来描述。

内部类能直接访问外部类中成员，是因为内部类持有了外部类的引用，即外部类名.this。 内部类也可以存放在局部位置上，但是内部类在局部位置上只能访问局部中被final修饰的局部变量。

\*/

**class** Outer

{

**private static int** num = 31;

**class** Inner// 内部类。

{

**void** show()

{

System.out.println("show run..."+num);

}

/\*static void function()//如果内部类中定义了静态成员，该内部类也必须是静态的。

{

System.out.println("function run ...."+num);

}

\*/

}

**public void** method()

{

Inner in = **new** Inner(); in.show();

}

}

**class** InnerClassDemo

{

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

{

// Outer out = new Outer();

// out.method();

//直接访问外部类中的内部类中的成员。

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

// in.show();

//如果内部类是静态的。 相当于一个外部类

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

// in.show();

//如果内部类是静态的，成员是静态的。

// Outer.Inner.function();

}

}

69.匿名内部类：

/\*

匿名内部类，就是内部类的简写格式。 必须有前提： 内部类必须继承或者实现一个外部类或者接口。

匿名内部类：其实就是一个匿名子类对象。 格式：new 父类or接口(){子类内容}

\*/

**abstract class** Demo

{

**abstract void** show();

}

**class** Outer

{

**int** num = 4;

/\*

class Inner extends Demo

{

void show()

{

System.out.println("show ..."+num);

}

}

\*/

**public void** method()

{

//new Inner().show();

**new** Demo()//匿名内部类。

{

**void** show()

{

System.out.println("show ........"+num);

}

}.show();

}

}

**class** InnerClassDemo4

{

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

{

**new** Outer().method();

}

}

# 70.匿名内部类使用范例（两个）：

范例（一）：

**interface** Inter

{

**void** show1();

**void** show2();

}

**class** Outer

{

/\*

class Inner implements Inter

{

public void show1()

{

}

public void show2()

{

}

}

\*/

**public void** method()

{

// Inner in = new Inner();

// in.show1();

// in.show2();

Inter in = **new** Inter()

{

**public void** show1()

{

}

**public void** show2()

{

}

};

in.show1();

in.show2();

}

}

/\*

通常的使用场景之一： 当函数参数是接口类型时，而且接口中的方法不超过三个。 可以用匿名内部类作为实际参数进行传递

\*/

**class** InnerClassDemo5

{

**class** Inner

{

}

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

{

System.out.println("Hello World!");

/\*

show(new Inter()

{

});

\*/

public void show1(){} public void show2(){}

// new Inner();

}

**public void** method()

{

**new** Inner();

}

**public static void** show(Inter in)

{

in.show1();

in.show2();

}

}

范例（二）：

**class** Outer

{

**void** method()

{

Object obj = **new** Object()

{

**public void** show()

{

System.out.println("show run");

}

};

obj.show();//因为匿名内部类这个子类对象被向上转型为了Object类型。

//这样就不能在使用子类特有的方法了。

}

}

**class** InnerClassDemo6

{

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

{

**new** Outer().method();

}

}

注意：内部类的class字节码文件名字格式为Outer$Inner.class。

# 第四章：异常机制

71.异常简述：

/\*

异常：是在运行时期发生的不正常情况。。 在java中用类的形式对不正常情况进行了描述和封装对象。 描述不正常的情况的类，就称为异常类。

以前正常流程代码和问题处理代码相结合， 现在将正常流程代码和问题处理代码分离。提高阅读性. 其实异常就是java通过面向对象的思想将问题封装成了对象. 用异常类对其进行描述。

不同的问题用不同的类进行具体的描述。 比如角标越界、空指针等等。 问题很多，意味着描述的类也很多， 将其共性进行向上抽取，形成了异常体系。

最终问题（不正常情况）就分成了两大类。

Throwable:无论是error，还是异常，问题，问题发生就应该可以抛出，让调用者知道并处理。

//该体系的特点就在于Throwable及其所有的子类都具有可抛性。 可抛性到底指的是什么呢？怎么体现可抛性呢？ 其实是通过两个关键字来体现的。

throws throw ,凡是可以被这两个关键字所操作的类和对象都具备可抛性.

|--1，一般不可处理的。Error

特点：是由jvm抛出的严重性的问题。 这种问题发生一般不针对性处理。直接修改程序

|--2，可以处理的。Exception

该体系的特点： 子类的后缀名都是用其父类名作为后缀，阅读性很强。

\*/

**class** ExceptionDemo

{

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

{

space

**int**[] arr = **new int**[1024\*1024\*800];//java.lang.OutOfMemoryError: Java heap

// arr = null;

// System.out.println(arr[3]);

//

//

// sleep(-5);

}

**public static void** sleep2(**int** time)

{

**if**(time<0)

{

// 处理办法。

// 处理办法。

// 处理办法。

// 处理办法。

// 处理办法。

}

**if**(time>100000)

|  |  |  |
| --- | --- | --- |
| // | { | 处理办法。 |
| // |  | 处理办法。 |
| // |  | 处理办法。 |
| // |  | 处理办法。 |
| // |  | 处理办法。 |
| // |  | 处理办法。 |

}

System.out.println("我睡。。。"+time);

// sleep(-5);

}

**public static void** sleep(**int** time)

{

**if**(time<0)

{

// 抛出 new FuTime();//就代码着时间为负的情况，这个对象中会包含着问题的名称，信息，位 置等信息。

}

**if**(time>100000)

{

// 抛出 new BigTime();

}

System.out.println("我睡。。。"+time);

}

}

/\*

class FuTime

{

}

class BigTime

{

}

\*/

72.自定义异常：

/\*

对于角标是整数不存在，可以用角标越界表示， 对于负数为角标的情况，准备用负数角标异常来表示。

负数角标这种异常在java中并没有定义过。 那就按照java异常的创建思想，面向对象，将负数角标进行自定义描述。并封装成对象。

这种自定义的问题描述成为自定义异常。 注意：如果让一个类称为异常类，必须要继承异常体系，因为只有称为异常体系的子类才有资格具备可抛性。

才可以被两个关键字所操作，throws throw

异常的分类：

1，编译时被检测异常:只要是Exception和其子类都是，除了特殊子类RuntimeException体系。 这种问题一旦出现，希望在编译时就进行检测，让这种问题有对应的处理方式。 这样的问题都可以针对性的处理。

2，编译时不检测异常(运行时异常):就是Exception中的RuntimeException和其子类。 这种问题的发生，无法让功能继续，运算无法进行，更多是因为调用者的原因导致的而或者引发了内部

状态的改变导致的。

那么这种问题一般不处理，直接编译通过，在运行时，让调用者调用时的程序强制停止,让调用者对代 码进行修正。

所以自定义异常时，要么继承Exception，要么继承RuntimeException。

throws 和throw的区别：

1，throws使用在函数上。 throw使用在函数内。

2，throws抛出的是异常类，可以抛出多个，用逗号隔开。 throw抛出的是异常对象。

\*/

**class** FuShuIndexException **extends** Exception

{

FuShuIndexException()

{}

FuShuIndexException(String msg)

{

**super**(msg);

}

}

**class** Demo

{

**public int** method(**int**[] arr,**int** index)//throws NullPointerException//FuShuIndexException

{

**if**(arr==**null**)

**throw new** NullPointerException("数组的引用不能为空！");

**if**(index>=arr.length)

{

**throw new** ArrayIndexOutOfBoundsException("数组的角标越界啦，哥们，你是不是疯

了？："+index);

}

**if**(index<0)

{

**throw new** FuShuIndexException("角标变成负数啦！！");

}

**return** arr[index];

}

}

**class** ExceptionDemo3

{

**public static void** main(String[] args) //throws FuShuIndexException

{

**int**[] arr = **new int**[3];

Demo d = **new** Demo();

**int** num = d.method(**null**,-30); System.out.println("num="+num); System.out.println("over");

}

}

/\*

class Person

{

private String name; Person(String name)

{

this.name = name;

}

public String getName()

{

return name;

}

}

class Student extends Person

{

Student(String name)

{

super(name);

}

}

\*/

73.异常的捕捉处理：

/\*

异常处理的捕捉形式： 这是可以对异常进行针对性处理的方式。

具体格式是：

try

{

}

//需要被检测异常的代码。

catch(异常类 变量)//该变量用于接收发生的异常对象

{

//处理异常的代码。

}

finally

{

//一定会被执行的代码。

}

异常处理的原则：

1，函数内容如果抛出需要检测的异常，那么函数上必须要声明。 否则必须在函数内用trycatch捕捉，否则编译失败。

2，如果调用到了声明异常的函数，要么trycatch要么throws，否则编译失败。 3，什么时候catch，什么时候throws 呢？

功能内容可以解决，用catch。

解决不了，用throws告诉调用者，由调用者解决 。

4，一个功能如果抛出了多个异常，那么调用时，必须有对应多个catch进行针对性的处理。 内部又几个需要检测的异常，就抛几个异常，抛出几个，就catch几个。

\*/

**class** FuShuIndexException **extends** Exception

{

FuShuIndexException()

{}

FuShuIndexException(String msg)

{

**super**(msg);

}

}

**class** Demo

{

**public int** method(**int**[] arr,**int** index)//throws NullPointerException,FuShuIndexException

{

**if**(arr==**null**)

**throw new** NullPointerException("没有任何数组实体");

**if**(index<0)

**throw new** FuShuIndexException();

**return** arr[index];

}

}

**class** ExceptionDemo4

{

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

{

**int**[] arr = **new int**[3]; Demo d = **new** Demo(); **try**

{

**int** num = d.method(**null**,-1); System.out.println("num="+num);

}

**catch**(NullPointerException e)

{

System.out.println(e.toString());

}

**catch** (FuShuIndexException e)

{

System.out.println("message:"+e.getMessage()); System.out.println("string:"+e.toString());

e.printStackTrace();//jvm默认的异常处理机制就是调用异常对象的这个方法。

System.out.println("负数角标异常!!!!");

}

/\*

catch(Exception e)//多catch父类的catch放在最下面。

{

}

\*/ System.out.println("over");

}

}

# 74.try、catch、finally的几种组合：

**class** Demo

{

**public int** show(**int** index)**throws** ArrayIndexOutOfBoundsException

{

**if**(index<0)

**throw new** ArrayIndexOutOfBoundsException("越界啦！！");

**int**[] arr = **new int**[3]; **return** arr[index];

}

}

**class** ExceptionDemo5

{

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

{

Demo d = **new** Demo();

### try

{

**int** num = d.show(-1); System.out.println("num="+num);

}

**catch** (ArrayIndexOutOfBoundsException e)

{

System.out.println(e.toString());

// return ;

// System.exit(0);//退出jvm。

}

**finally**//通常用于关闭(释放)资源。

{

System.out.println("finally");

}

System.out.println("over");

}

}

/\*

连接数据库 查询。Exception 关闭连接。

\*/

/\*

try catch finally 代码块组合特点：

1，

try catch finally

2,

try catch(多个)当没有必要资源需要释放时，可以不用定义finally。

3,

try finally 异常无法直接catch处理，但是资源需要关闭。

void show()throws Exception

{

try

{

}

//开启资源。

throw new Exception();

finally

{

//关闭资源。

}

/\* catch(Exception e)

{

}

\*/

}

\*/

75.异常的注意事项：

异常的注意事项：

1，子类在覆盖父类方法时，父类的方法如果抛出了异常， 那么子类的方法只能抛出父类的异常或者该异常的子类。

2，如果父类抛出多个异常，那么子类只能抛出父类异常的子集。

简单说：子类覆盖父类只能抛出父类的异常或者子类或者子集。 注意：如果父类的方法没有抛出异常，那么子类覆盖时绝对不能抛，就只能try .

**interface** Inter

{

**void** function();

}

**class** D **implements** Inter

{

**public void** function()//throws Exception

{}

}

**class** A **extends** Exception

{

}

### class B extends A

{

}

**class** C **extends** Exception

{

}

Exception

|--A

|--B

|--C

**class** Fu

{

**void** show()**throws** A

{}

}

**class** Test

{

**void** method(Fu f)//Fu f = new Zi();

{

### try

{

f.show();

}

**catch** (A a)

{

}

}

}

**class** Zi **extends** Fu

{

**void** show()**throws** C

{

}

}

### class

{

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

{

Test t = **new** Test(); t.show(**new** Zi());

}

}

76.异常处理练习：

/\*

毕老师用电脑上课。 问题领域中涉及两个对象。 毕老师，电脑。 分析其中的问题。 比如电脑蓝屏啦。冒烟啦。

\*/

**class** LanPingException **extends** Exception

{

LanPingException(String msg)

{

**super**(msg);

}

}

**class** MaoYanException **extends** Exception

{

MaoYanException(String msg)

{

**super**(msg);

}

}

**class** NoPlanException **extends** Exception

{

NoPlanException(String msg)

{

**super**(msg);

}

}

**class** Computer

{

**private int** state = 2;

**public void** run()**throws** LanPingException,MaoYanException

{

**if**(state==1)

**throw new** LanPingException("电脑蓝屏啦！！");

**if**(state==2)

**throw new** MaoYanException("电脑冒烟啦！！");

System.out.println("电脑运行");

}

**public void** reset()

{

state = 0; System.out.println("电脑重启");

}

}

**class** Teacher

{

**private** String name; **private** Computer comp; Teacher(String name)

{

**this**.name = name; comp = **new** Computer();

}

**public void** prelect()**throws** NoPlanException

{

### try

{

comp.run();

System.out.println(name+"讲课");

}

**catch** (LanPingException e)

{

System.out.println(e.toString()); comp.reset();

prelect();

}

**catch** (MaoYanException e)

{

System.out.println(e.toString()); test();

//可以对电脑进行维修。

// throw e;

**throw new** NoPlanException("课时进度无法完成，原因："+e.getMessage());

}

}

**public void** test()

{

System.out.println("大家练习");

}

}

**class** ExceptionTest

{

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

{

Teacher t = **new** Teacher("毕老师");

### try

{

t.prelect();

}

**catch** (NoPlanException e)

{

System.out.println(e.toString()+"......"); System.out.println("换人");

}

}

}

/\*

class NoAddException extends Exception

{}

void addData(Data d)throws NoAddException

{

连接数据库

try

{

添加数据。出现异常 SQLException();

}

catch(SQLException e)

{

//处理代码。

throw new NoAddException();

}

fianlly

{

关闭数据库。

}

}

\*/

77.类的全名=包名.类名：

**package** mypack;

//import packa.DemoA;//导入了packa包中的DemoA类。

//import packa.DemoAA;

//import packa.DemoAAA;

/\*

import packa.\*;//导入了packa包中所有的类。 import packa.abc.\*;

packa\DemoA.class packa\abc\DemoAbc.class

\*/

//导包的原则：用到哪个类，就导入哪个类。

//import 干嘛用的啊？为了简化类名书写。

**class** PackageDemo

{

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

{

//packa.DemoA d = new packa.DemoA(); DemoAbc d = **new** DemoA();

d.show();

packb.DemoB b = **new** packb.DemoB(); b.method(); System.out.println("Hello Package!");

}

}

/\*

PackageDemo.java:8: 找不到符号 符号： 类 DemoA

位置： 类 mypack.PackageDemo

DemoA d = new DemoA();

^ PackageDemo.java:8: 找不到符号 符号： 类 DemoA

位置： 类 mypack.PackageDemo

DemoA d = new DemoA();

^

2 错误 原因：类名写错。

因为DemoA类已经有包的所属，所以必须要明确其报名。 记住：DemoA这个名词错的。 正确名称是报名.类名 packa.DemoA

--------------------------------------

PackageDemo.java:8: 软件包 packa 不存在

packa.DemoA d = new packa.DemoA();

^ PackageDemo.java:8: 软件包 packa 不存在

packa.DemoA d = new packa.DemoA();

^

2 错误

原因是在当前目录下没找到，因为packa存放在其他目录下， 应该配置一个classpath。

-------------------------------------

PackageDemo.java:8: packa.DemoA 在 packa 中不是公共的；无法从外部软件包中对其进 行访问

packa.DemoA d = new packa.DemoA();

^

PackageDemo.java:8: packa.DemoA 在 packa 中不是公共的；无法从外部软件包中对其进 行访问

2 错误

packa.DemoA d = new packa.DemoA();

^

--------------------------------------

PackageDemo.java:9: show() 在 packa.DemoA 中不是公共的；无法从外部软件包中对其进 行访问

1 错误

d.show();

^

总结： 包与包之间的类进行访问，被访问的包中的类必须是public的，被访问的包中的类的方法也必须是public的。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 同一类中 | public  ok | protected  ok | default  ok | private  ok |
| 同一包中  子类中 不同包中 | ok ok  ok | ok ok | ok |  |
| \*/ |  |  |  |  |

# 第五章：多线程技术

78.线程简述：

/\*

进程：正在进行中的程序(直译)。 线程：就是进程中一个负责程序执行的控制单元(执行路径)， 一个进程中可以多执行路径，称之为多线程。 一个进程中至少要有一个线程。 开启多个线程是为了同时运行多部分代码。 每一个线程都有自己运行的内容。这个内容可以称为线程要执行的任务。 多线程好处：解决了多部分同时运行的问题。 多线程的弊端：线程太多回到效率的降低。

其实应用程序的执行都是cpu在做着快速的切换完成的，这个切换是随机的。 JVM启动时就启动了多个线程，至少有两个线程可以分析的出来。 1，执行main函数的线程，

该线程的任务代码都定义在main函数中。

2，负责垃圾回收的线程。

\*/

**class** Demo **extends** Object

{

**public void** finalize()

{

System.out.println("demo ok");

}

}

**class** ThreadDemo

{

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

{

**new** Demo(); **new** Demo(); **new** Demo(); System.gc();

System.out.println("Hello World!");

}

}

# 79.创建线程方法乊一即继承thread类：

/\*

如何创建一个线程呢？ 创建线程方式一：继承Thread类。

步骤： 1，定义一个类继承Thread类。 2，覆盖Thread类中的run方法。

3，直接创建Thread的子类对象创建线程。

4，调用start方法开启线程并调用线程的任务run方法执行。 可以通过Thread的getName获取线程的名称 Thread-编号(从0开始)

主线程的名字就是main。

\*/

**class** Demo **extends** Thread

{

**private** String name; Demo(String name)

{

**super**(name);

//this.name = name;

}

**public void** run()

{

**for**(**int** x=0; x<10; x++)

{

//for(int y=-9999999; y<999999999; y++){}

System.out.println(name+"....x="+x+".....name="+Thread.currentThread().getNam e());

}

}

}

**class** ThreadDemo2

{

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

{

/\*

创建线程的目的是为了开启一条执行路径，去运行指定的代码和其他代码实现同时运行。 而运行的指定代码就是这个执行路径的任务。 jvm创建的主线程的任务都定义在了主函数中。

而自定义的线程它的任务在哪儿呢？ Thread类用于描述线程，线程是需要任务的。所以Thread类也对任务的描述。 这个任务就通过Thread类中的run方法来体现。也就是说，run方法就是封装自定义线程运行任务的 函数。

run方法中定义就是线程要运行的任务代码。

开启线程是为了运行指定代码，所以只有继承Thread类，并复写run方法。 将运行的代码定义在run方法中即可。

\*/

//

// Thread t1 = new Thread();

Demo d1 = **new** Demo("旺财");

Demo d2 = **new** Demo("xiaoqiang"); d1.start();//开启线程，调用run方法。

d2.start(); System.out.println("over...."+Thread.currentThread().getName());

}

}

# 80.多线程典型案例乊银行存钱：

/\*

需求:储户，两个，每个都到银行存钱每次存100，，共存三次。

\*/

**class** Bank

{

**private int** sum;

// private Object obj = new Object();

**public synchronized void** add(**int** num)//同步函数

{

// synchronized(obj)

// {

sum = sum + num;

// -->

**try**{Thread.sleep(10);}**catch**(InterruptedException e){} System.out.println("sum="+sum);

// }

}

}

**class** Cus **implements** Runnable

{

**private** Bank b = **new** Bank();

**public void** run()

{

**for**(**int** x=0; x<3; x++)

{

b.add(100);

}

}

}

**class** BankDemo

{

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

{

Cus c = **new** Cus();

Thread t1 = **new** Thread(c); Thread t2 = **new** Thread(c); t1.start();

t2.start();

}

}

# 81.死锁常见情形乊同步嵌套:

/\*

死锁：常见情景之一：同步的嵌套。

\*/

**class** Ticket **implements** Runnable

{

**private int** num = 100; Object obj = **new** Object(); **boolean** flag = **true**; **public void** run()

{

**if**(flag)

### while(true)

{

**synchronized**(obj)

{

}

}

### else

show();

### while(true)

**this**.show();

}

### public synchronized void show()

{

**synchronized**(obj)

{

**if**(num>0)

{

**try**{Thread.sleep(10);}**catch** (InterruptedException e){}

System.out.println(Thread.currentThread().getName()+".....sale...."+num--);

}

}

}

}

**class** DeadLockDemo

{

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

{

Ticket t = **new** Ticket();

// System.out.println("t:"+t);

Thread t1 = **new** Thread(t); Thread t2 = **new** Thread(t);

t1.start(); **try**{Thread.sleep(10);}**catch**(InterruptedException e){} t.flag = **false**;

t2.start();

}

}

# 82.典型死锁案例（synchronized同步嵌套）：

**class** Test **implements** Runnable

{

**private boolean** flag; Test(**boolean** flag)

{

**this**.flag = flag;

}

**public void** run()

{

**if**(flag)

{

### while(true)

**synchronized**(MyLock.locka)

{

locka....");

lockb....");

}

}

### else

{

System.out.println(Thread.currentThread().getName()+"..if

**synchronized**(MyLock.lockb) { System.out.println(Thread.currentThread().getName()+"..if

}

### while(true)

**synchronized**(MyLock.lockb)

{

lockb....");

locka....");

}

}

System.out.println(Thread.currentThread().getName()+"..else

**synchronized**(MyLock.locka)

{

System.out.println(Thread.currentThread().getName()+"..else

}

}

}

**class** MyLock

{

**public static final** Object locka = **new** Object();

**public static final** Object lockb = **new** Object();

}

**class** DeadLockTest

{

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

{

Test a = **new** Test(**true**); Test b = **new** Test(**false**);

Thread t1 = **new** Thread(a); Thread t2 = **new** Thread(b); t1.start();

t2.start();

}

}

83.多线程下的单例：

/\*

多线程下的单例

\*/

//饿汉式

**class** Single

{

**private static final** Single s = **new** Single();

**private** Single(){}

**public static** Single getInstance()

{

**return** s;

}

}

//懒汉式 加入同步为了解决多线程安全问题。 加入双重判断是为了解决效率问题。

**class** Single

{

**private static** Single s = **null**; **private** Single(){}

**public static** Single getInstance()

{

**if**(s==**null**)

{

**synchronized**(Single.**class**)

{

**if**(s==**null**)

// -->0 -->1

s = **new** Single();

}

}

**return** s;

}

}

**class** SingleDemo

{

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

{

System.out.println("Hello World!");

}

}

84.静态同步凼数的锁：

/\*

静态的同步函数使用的锁是该函数所属字节码文件对象，

可以用 getClass方法获取，也可以用当前类名.class 表示。

\*/

**class** Ticket **implements** Runnable

{

**private static int** num = 100;

// Object obj = new Object(); **boolean** flag = **true**; **public void** run()

{

// System.out.println("this:"+this.getClass());

**if**(flag)

### while(true)

{

**synchronized**(Ticket.**class**)//(this.getClass())

{

**if**(num>0)

{

**try**{Thread.sleep(10);}**catch** (InterruptedException e){}

System.out.println(Thread.currentThread().getName()+".....obj...."+num--);

}

}

}

### else

**while**(**true**)

**this**.show();

}

### public static synchronized void show()

{

**if**(num>0)

{

**try**{Thread.sleep(10);}**catch** (InterruptedException e){}

System.out.println(Thread.currentThread().getName()+".....function...."+num--

);

}

}

}

**class** StaticSynFunctionLockDemo

{

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

{

Ticket t = **new** Ticket();

// Class clazz = t.getClass();

//

// Class clazz = Ticket.class;

// System.out.println("t:"+t.getClass());

Thread t1 = **new** Thread(t); Thread t2 = **new** Thread(t); t1.start();

**try**{Thread.sleep(10);}**catch**(InterruptedException e){} t.flag = **false**;

t2.start();

}

}

# 85.同步凼数&同步代码块的锁：

/\* 同步函数的使用的锁是this； 同步函数和同步代码块的区别：

同步函数的锁是固定的this。 同步代码块的锁是任意的对象。 建议使用同步代码块。

\*/

**class** Ticket **implements** Runnable

{

**private int** num = 100;

// Object obj = new Object(); **boolean** flag = **true**; **public void** run()

{

// System.out.println("this:"+this);

**if**(flag)

### while(true)

{

### synchronized(this)

{

**if**(num>0)

{

**try**{Thread.sleep(10);}**catch** (InterruptedException e){}

System.out.println(Thread.currentThread().getName()+".....obj...."+num--);

}

}

}

### else

**while**(**true**)

**this**.show();

}

### public synchronized void show()

{

**if**(num>0)

{

**try**{Thread.sleep(10);}**catch** (InterruptedException e){}

System.out.println(Thread.currentThread().getName()+".....function...."+num--

);

}

}

}

**class** SynFunctionLockDemo

{

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

{

Ticket t = **new** Ticket();

// System.out.println("t:"+t);

Thread t1 = **new** Thread(t); Thread t2 = **new** Thread(t);

t1.start(); **try**{Thread.sleep(10);}**catch**(InterruptedException e){} t.flag = **false**;

t2.start();

}

}

# 86.创建线程方法乊二即实现Runnable接口：

/\*

创建线程的第一种方式:继承Thread类。 创建线程的第二种方式：实现Runnable接口。 1,定义类实现Runnable接口。 2，覆盖接口中的run方法，将线程的任务代码封装到run方法中。

3，通过Thread类创建线程对象，并将Runnable接口的子类对象作为Thread类的构造函数的参数进行传递。 为什么？因为线程的任务都封装在Runnable接口子类对象的run方法中。 所以要在线程对象创建时就必须明确要运行的任务。

4，调用线程对象的start方法开启线程。

实现Runnable接口的好处：

1，将线程的任务从线程的子类中分离出来，进行了单独的封装。 按照面向对象的思想将任务的封装成对象。

2，避免了java单继承的局限性。 所以，创建线程的第二种方式较为常用。

\*/

**class** Demo **implements** Runnable//extends Fu //准备扩展Demo类的功能，让其中的内容可以作为 线程的任务执行。

//通过接口的形式完成。

{

**public void** run()

{

show();

}

**public void** show()

{

**for**(**int** x=0; x<20; x++)

{

System.out.println(Thread.currentThread().getName()+"....."+x);

}

}

}

**class** ThreadDemo

{

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

{

Demo d = **new** Demo(); Thread t1 = **new** Thread(d); Thread t2 = **new** Thread(d); t1.start();

t2.start();

// Demo d1 = new Demo();

// Demo d2 = new Demo();

// d1.start();

// d2.start();

}

}

/\*

class Thread

{

private Runnable r; Thread()

{

}

Thread(Runnable r)

{

this.r = r;

}

public void run()

{

if(r!=null)

r.run();

}

public void start()

{

run();

}

}

class ThreadImpl implements Runnable

{

public void run()

{

System.out.println("runnable run");

}

}

ThreadImpl i = new ThreadImpl(); Thread t = new Thread(i); t.start();

class SubThread extends Thread

{

public void run()

{

System.out.println("hahah");

}

}

//SubThread s = new SubThread();

//s.start();

\*/

87.多线程典型案例乊卖票：

/\*

需求：卖票。

\*/

/\*

线程安全问题产生的原因：

1，多个线程在操作共享的数据。

2，操作共享数据的线程代码有多条。 当一个线程在执行操作共享数据的多条代码过程中，其他线程参与了运算。 就会导致线程安全问题的产生。

解决思路； 就是将多条操作共享数据的线程代码封装起来，当有线程在执行这些代码的时候， 其他线程时不可以参与运算的。 必须要当前线程把这些代码都执行完毕后，其他线程才可以参与运算。 在java中，用同步代码块就可以解决这个问题。

同步代码块的格式：

synchronized(对象)

{

需要被同步的代码 ；

}

同步的好处：解决了线程的安全问题。 同步的弊端：相对降低了效率，因为同步外的线程的都会判断同步锁。 同步的前提：同步中必须有多个线程并使用同一个锁。

\*/

class Ticket implements Runnable//extends Thread

{

private int num = 100;

Object obj = new Object(); public void run()

{

while(true)

{

synchronized(obj)

{

if(num>0)

{

try{Thread.sleep(10);}catch (InterruptedException e){}

System.out.println(Thread.currentThread().getName()+".....sale...."+num--);

}

}

}

}

}

class TicketDemo

{

public static void main(String[] args)

{

Ticket t = new Ticket();//创建一个线程任务对象。 Thread t1 = new Thread(t);

Thread t2 = new Thread(t); Thread t3 = new Thread(t); Thread t4 = new Thread(t);

t1.start();

t2.start();

t3.start();

t4.start();

/\*

Ticket t1 = new Ticket();

// Ticket t2 = new Ticket();

// Ticket t3 = new Ticket();

// Ticket t4 = new Ticket();

t1.start();

t1.start();//一个线程不能开启两次，会抛出无效线程状态异常 t1.start();

t1.start();

\*/

}

}

# 88.线程中的join（）方法：

**class** Demo **implements** Runnable

{

**public void** run()

{

**for**(**int** x=0; x<50; x++)

{

System.out.println(Thread.currentThread().toString()+"....."+x); Thread.yield();

}

}

}

**class** JoinDemo

{

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

{

Demo d = **new** Demo();

Thread t1 = **new** Thread(d); Thread t2 = **new** Thread(d);

t1.start();

t2.start();

// t2.setPriority(Thread.MAX\_PRIORITY);

// t1.join();//t1线程要申请加入进来，运行。临时加入一个线程运算时可以使用join方法。

**for**(**int** x=0; x<50; x++)

{

// System.out.println(Thread.currentThread()+"....."+x);

}

}

}

# 89.线程乊（多）生产者、（多）消费者模式：

/\*

生产者，消费者。

多生产者，多消费者的问题。 if判断标记，只有一次，会导致不该运行的线程运行了。出现了数据错误的情况。 while判断标记，解决了线程获取执行权后，是否要运行！

notify:只能唤醒一个线程，如果本方唤醒了本方，没有意义。而且while判断标记+notify会导致死锁。 notifyAll解决了本方线程一定会唤醒对方线程的问题。

\*/

**class** Resource

{

**private** String name;

**private int** count = 1;

**private boolean** flag = **false**;

**public synchronized void** set(String name)//

{

**while**(flag)

**try**{**this**.wait();}**catch**(InterruptedException e){}// t1 t0

**this**.name = name + count;//烤鸭1 烤鸭2 烤鸭3 count++;//2 3 4

System.out.println(Thread.currentThread().getName()+"... 生 产

者..."+**this**.name);//生产烤鸭1 生产烤鸭2 生产烤鸭3 flag = **true**;

notifyAll();

}

**public synchronized void** out()// t3

{

**while**(!flag)

**try**{**this**.wait();}**catch**(InterruptedException e){} //t2 t3 System.out.println(Thread.currentThread().getName()+"... 消 费

者........"+**this**.name);//消费烤鸭1

flag = **false**; notifyAll();

}

}

**class** Producer **implements** Runnable

{

**private** Resource r; Producer(Resource r)

{

**this**.r = r;

}

**public void** run()

{

### while(true)

{

r.set("烤鸭");

}

}

}

**class** Consumer **implements** Runnable

{

**private** Resource r; Consumer(Resource r)

{

**this**.r = r;

}

**public void** run()

{

### while(true)

{

r.out();

}

}

}

**class** ProducerConsumerDemo

{

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

{

Resource r = **new** Resource(); Producer pro = **new** Producer(r); Consumer con = **new** Consumer(r);

Thread t0 = **new** Thread(pro); Thread t1 = **new** Thread(pro); Thread t2 = **new** Thread(con); Thread t3 = **new** Thread(con); t0.start();

t1.start();

t2.start();

t3.start();

}

}

注意：if判断标记与notify（）方法对应，while标记判断跟notifyAll（）方法对应，这样才不会导致死 锁，否则任意一个搭配错的话都会导致死锁。 90.jdk1.5乊后新的锁解决生产者和消费者问题：

/\*

jdk1.5以后将同步和锁封装成了对象。 并将操作锁的隐式方式定义到了该对象中， 将隐式动作变成了显示动作。

Lock接口： 出现替代了同步代码块或者同步函数。将同步的隐式锁操作变成现实锁操作。 同时更为灵活。可以一个锁上加上多组监视器。

lock():获取锁。 unlock():释放锁，通常需要定义finally代码块中。

Condition接口：出现替代了Object中的wait notify notifyAll方法。 将这些监视器方法单独进行了封装，变成Condition监视器对象。 可以任意锁进行组合。

await();

signal(); signalAll();

\*/

import java.util.concurrent.locks.\*;

class Resource

{

private String name; private int count = 1;

private boolean flag = false;

// 创建一个锁对象。

Lock lock = new ReentrantLock();

//通过已有的锁获取该锁上的监视器对象。

// Condition con = lock.newCondition();

//通过已有的锁获取两组监视器，一组监视生产者，一组监视消费者。 Condition producer\_con = lock.newCondition(); Condition consumer\_con = lock.newCondition();

public void set(String name)// t0 t1

{

lock.lock(); try

{

while(flag)

// try{lock.wait();}catch(InterruptedException e){}// t1 t0 try{producer\_con.await();}catch(InterruptedException e){}// t1 t0

this.name = name + count;//烤鸭1 烤鸭2 烤鸭3 count++;//2 3 4

System.out.println(Thread.currentThread().getName()+"... 生 产 者

5.0..."+this.name);//生产烤鸭1 生产烤鸭2 生产烤鸭3

flag = true;

// notifyAll();

// con.signalAll(); consumer\_con.signal();

}

finally

{

lock.unlock();

}

}

public void out()// t2 t3

{

lock.lock(); try

{

while(!flag)

// try{this.wait();}catch(InterruptedException e){} //t2 t3

|  |  |  |  |
| --- | --- | --- | --- |
| try{cousumer\_con.await();}catch(InterruptedException e){} | //t2 | t3 |  |
| System.out.println(Thread.currentThread().getName()+"... | 消 |  | 费 |

者.5.0......."+this.name);//消费烤鸭1

flag = false;

// notifyAll();

// con.signalAll(); producer\_con.signal();

}

finally

{

lock.unlock();

}

}

}

class Producer implements Runnable

{

private Resource r; Producer(Resource r)

{

this.r = r;

}

public void run()

{

while(true)

{

r.set("烤鸭");

}

}

}

class Consumer implements Runnable

{

private Resource r; Consumer(Resource r)

{

this.r = r;

}

public void run()

{

while(true)

{

r.out();

}

}

}

class ProducerConsumerDemo2

{

public static void main(String[] args)

{

Resource r = new Resource(); Producer pro = new Producer(r); Consumer con = new Consumer(r);

Thread t0 = new Thread(pro); Thread t1 = new Thread(pro); Thread t2 = new Thread(con); Thread t3 = new Thread(con); t0.start();

t1.start();

t2.start();

t3.start();

}

}

91.线程间通讯：

/\*

线程间通讯： 多个线程在处理同一资源，但是任务却不同。

\*/

//资源

**class** Resource

{

String name; String sex;

}

//输入

**class** Input **implements** Runnable

{

Resource r ;

// Object obj = new Object(); Input(Resource r)

{

**this**.r = r;

}

**public void** run()

{

**int** x = 0;

### while(true)

{

### synchronized(r)

{

**if**(x==0)

{

r.name = "mike"; r.sex = "nan";

}

### else

{

r.name = "丽丽"; r.sex = "女女女女女女";

}

}

x = (x+1)%2;

}

}

}

//输出

**class** Output **implements** Runnable

{

Resource r;

// Object obj = new Object(); Output(Resource r)

{

**this**.r = r;

}

**public void** run()

{

### while(true)

{

### synchronized(r)

{

System.out.println(r.name+"....."+r.sex);

}

}

}

}

**class** ResourceDemo

{

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

{

//创建资源。

Resource r = **new** Resource();

//创建任务。

Input in = **new** Input(r); Output out = **new** Output(r);

//创建线程，执行路径。

Thread t1 = **new** Thread(in); Thread t2 = **new** Thread(out);

//开启线程

t1.start();

t2.start();

}

}

92.线程中等待/唤醒机制：

/\*

等待/唤醒机制。

涉及的方法：

1，wait(): 让线程处于冻结状态，被wait的线程会被存储到线程池中。

2，notify():唤醒线程池中一个线程(任意). 3，notifyAll():唤醒线程池中的所有线程。

这些方法都必须定义在同步中。 因为这些方法是用于操作线程状态的方法。 必须要明确到底操作的是哪个锁上的线程。

为什么操作线程的方法wait notify notifyAll定义在了Object类中？ 因为这些方法是监视器的方法。监视器其实就是锁。 锁可以是任意的对象，任意的对象调用的方式一定定义在Object类中。

\*/

//资源

**class** Resource

{

String name; String sex;

**boolean** flag = **false**;

}

//输入

**class** Input **implements** Runnable

{

Resource r ;

// Object obj = new Object(); Input(Resource r)

{

**this**.r = r;

}

**public void** run()

{

**int** x = 0;

### while(true)

{

### synchronized(r)

{

**if**(r.flag)

**try**{r.wait();}**catch**(InterruptedException e){}

**if**(x==0)

{

r.name = "mike"; r.sex = "nan";

}

### else

{

r.name = "丽丽"; r.sex = "女女女女女女";

}

r.flag = **true**; r.notify();

}

x = (x+1)%2;

}

}

}

//输出

**class** Output **implements** Runnable

{

Resource r;

// Object obj = new Object(); Output(Resource r)

{

**this**.r = r;

}

**public void** run()

{

### while(true)

{

### synchronized(r)

{

**if**(!r.flag)

**try**{r.wait();}**catch**(InterruptedException e){} System.out.println(r.name+"....."+r.sex);

r.flag = **false**; r.notify();

}

}

}

}

**class** ResourceDemo2

{

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

{

//创建资源。

Resource r = **new** Resource();

//创建任务。

Input in = **new** Input(r); Output out = **new** Output(r);

//创建线程，执行路径。

Thread t1 = **new** Thread(in); Thread t2 = **new** Thread(out);

//开启线程

t1.start();

t2.start();

}

}

以上代码也可以使用synchronized同步函数实现为：

**class** Resource

{

**private** String name;

**private** String sex;

**private boolean** flag = **false**;

**public synchronized void** set(String name,String sex)

{

**if**(flag)

**try**{**this**.wait();}**catch**(InterruptedException e){}

**this**.name = name; **this**.sex = sex; flag = **true**; **this**.notify();

}

### public synchronized void out()

{

**if**(!flag)

**try**{**this**.wait();}**catch**(InterruptedException e){} System.out.println(name+"...+...."+sex);

flag = **false**; notify();

}

}

//输入

**class** Input **implements** Runnable

{

Resource r ;

// Object obj = new Object(); Input(Resource r)

{

**this**.r = r;

}

**public void** run()

{

**int** x = 0;

### while(true)

{

**if**(x==0)

{

r.set("mike","nan");

}

### else

{

r.set("丽丽","女女女女女女");

}

x = (x+1)%2;

}

}

}

//输出

**class** Output **implements** Runnable

{

Resource r;

// Object obj = new Object(); Output(Resource r)

{

**this**.r = r;

}

**public void** run()

{

### while(true)

{

r.out();

}

}

}

**class** ResourceDemo3

{

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

{

//创建资源。

Resource r = **new** Resource();

//创建任务。

Input in = **new** Input(r); Output out = **new** Output(r);

//创建线程，执行路径。

Thread t1 = **new** Thread(in); Thread t2 = **new** Thread(out);

//开启线程

t1.start();

t2.start();

}

}

93.停止线程的方法：

/\*

停止线程：

1，stop方法。

2，run方法结束。 怎么控制线程的任务结束呢?

任务中都会有循环结构，只要控制住循环就可以结束任务。 控制循环通常就用定义标记来完成。 但是如果线程处于了冻结状态，无法读取标记。如何结束呢？

可以使用interrupt()方法将线程从冻结状态强制恢复到运行状态中来，让线程具备cpu的执行资格。

当时强制动作会发生了InterruptedException，记得要处理

\*/

**class** StopThread **implements** Runnable

{

**private boolean** flag = **true**; **public synchronized void** run()

{

**while**(flag)

{

### try

{

}

wait();//t0 t1

**catch** (InterruptedException e)

{

System.out.println(Thread.currentThread().getName()+"....."+e); flag = **false**;

}

System.out.println(Thread.currentThread().getName()+"......++++");

}

}

**public void** setFlag()

{

flag = **false**;

}

}

**class** StopThreadDemo

{

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

{

StopThread st = **new** StopThread();

Thread t1 = **new** Thread(st); Thread t2 = **new** Thread(st);

t1.start(); t2.setDaemon(**true**); t2.start();

**int** num = 1;

**for**(;;)

{

**if**(++num==50)

{

// st.setFlag(); t1.interrupt();

// t2.interrupt();

### break;

}

System.out.println("main...."+num);

}

System.out.println("over");

}

}

94.线程小练习（面试）：

/\*class Test implements Runnable

{

}\*/

public void run(Thread t)

{}

//如果错误 错误发生在哪一行？错误在第一行，应该被abstract修饰

**class** ThreadTest

{

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

{

**new** Thread(**new** Runnable()

{

**public void** run()

{

System.out.println("runnable run");

}

})

{

**public void** run()

{

System.out.println("subThread run");

}

}.start();

/\*

new Thread()

{

public void run()

{

for(int x=0; x<50; x++)

{

System.out.println(Thread.currentThread().getName()+"....x="+x);

}

}

}.start();

for(int x=0; x<50; x++)

{

System.out.println(Thread.currentThread().getName()+"....y="+x);

}

Runnable r = new Runnable()

{

public void run()

{

for(int x=0; x<50; x++)

{

System.out.println(Thread.currentThread().getName()+"....z="+x);

}

}

};

new Thread(r).start();

\*/

}

}

# 95.wait和sleep的区别：

wait 和 sleep 区别？

1，wait可以指定时间也可以不指定。 sleep必须指定时间。

2，在同步中时，对cpu的执行权和锁的处理不同。 wait：释放执行权，释放锁。 sleep:释放执行权，不释放锁。

**class** Demo

{

**void** show()

{

### synchronized(this)//

{

wait();//t0 t1 t2

}

}

**void** method()

{

**synchronized**(**this**)//t4

{

//wait(); notifyAll();

}//t4

}

}

### class

{

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

{

System.out.println("Hello World!");

}

}

96.多线程技术总结：

**package** src;

/\*

多线程总结：

1，进程和线程的概念。

|--进程：

|--线程：

2，jvm中的多线程体现。

|--主线程，垃圾回收线程，自定义线程。以及他们运行的代码的位置。

3，什么时候使用多线程，多线程的好处是什么？创建线程的目的？

|--当需要多部分代码同时执行的时候，可以使用。

4，创建线程的两种方式。★★★★★

|--继承Thread

|--步骤

|--实现Runnable

|--步骤

|--两种方式的区别？

5，线程的5种状态。 对于执行资格和执行权在状态中的具体特点。

|--被创建：

|--运行：

|--冻结：

|--临时阻塞：

|--消亡：

6，线程的安全问题。★★★★★

|--安全问题的原因：

|--解决的思想：

|--解决的体现：synchronized

|--同步的前提：但是加上同步还出现安全问题，就需要用前提来思考。

|--同步的两种表现方法和区别：

|--同步的好处和弊端：

|--单例的懒汉式。

|--死锁。

7，线程间的通信。等待/唤醒机制。

|--概念：多个线程，不同任务，处理同一资源。

|--等待唤醒机制。使用了锁上的 wait notify notifyAll. ★★★★★

|--生产者/消费者的问题。并多生产和多消费的问题。 while判断标记。用notifyAll唤醒对方。 ★

★★★★

|--JDK1.5以后出现了更好的方案，★★★ Lock接口替代了synchronized

Condition接口替代了Object中的监视方法，并将监视器方法封装成了Condition

和以前不同的是，以前一个锁上只能有一组监视器方法。现在，一个Lock锁上可以多组监视器方法对

象。

可以实现一组负责生产者，一组负责消费者。

|--wait和sleep的区别。★★★★★

8，停止线程的方式。

|--原理：

|--表现：--中断。

9，线程常见的一些方法。

|--setDaemon()

|--join();

|--优先级

|--yield();

|--在开发时，可以使用匿名内部类来完成局部的路径开辟。

\*/

97.线程查漏综述： 程序：源程序和字节码文件被称为"程序"（Program），是一个静态的概念。

迚程：执行中的程序叨做迚程（Process），是一个劢态的概念，迚程是程序的一次劢态执行过程， 占用特定的地址空间，每个迚程由三部分组成即cpu、data、code，每个迚程都是独立的，保有 自己的cpu时间，代码和数据，即使同一份程序产生好几个线程，它们乊间还是拥有自己的三样东 西，缺点是内存的浪费和cpu的负担。

织止线程的典型方法：

**package** cn.com.b;

/\*\*

\* 终止线程的典型方法

\*/

**public class** TestThreadCiycle **implements** Runnable {

String name;

**boolean** live=**true**;

**public** TestThreadCiycle(String name) {

**super**(); **this**.name = name;

}

@Override

**public void** run() { **int** i=0; **while**(live){

System.*out*.println(name+(i++));

}

}

**public void** terminate(){ live=**false**;

}

**public static void** main(String[] args) { TestThreadCiycle ttc=**new** TestThreadCiycle("线程A："); Thread t1=**new** Thread(ttc);//新生状态 t1.start();//就绪状态

**for** (**int** i = 0; i < 1000; i++) { System.*out*.println(i);

}

ttc.terminate(); System.*out*.println("ttc stop!");

}

}

# 第六章：常用类API

98.String类：

（一）String类的构造函数

**package** cn.itcast.p1.string.demo;

**public class** StringConstructorDemo {

/\*\*

\* **@param** args

\*/

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

/\*

\* 将字节数组或者字符数组转成字符串可以通过String类的构造函数完成。

\*/

*stringConstructorDemo2*();

}

**private static void** stringConstructorDemo2() {

**char**[] arr = {'w','a','p','q','x'}; String s = **new** String(arr,1,3);

System.*out*.println("s="+s);

}

**public static void** stringConstructorDemo() {

String s = **new** String();//等效于String s = ""; 不等效String s = null;

**byte**[] arr = {97,66,67,68};

String s1 = **new** String(arr); System.*out*.println("s1="+s1);

}

}

（二）String类的特点

**package** cn.itcast.p1.string.demo;

**public class** StringDemo {

/\*\*

\* **@param** args

\*/

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

/\*

\* String类的特点：

\* 字符串对象一旦被初始化就不会被改变。

\*

\*/

*stringDemo2*();

}

**public static void** stringDemo2() {

String s = "abc";//创建一个字符串对象在常量池中。

String s1 = **new** String("abc");//创建两个对象一个new一个字符串对象在堆内存中。 System.*out*.println(s==s1);//false

System.*out*.println(s.equals(s1));

//string类中的equals复写Object中的equals建立了string类自己的判断字符串对象是否相同 的依据。

//其实就是比较字符串内容。

// System.out.println("s="+s);

// System.out.println("s1="+s1);

}

/\*\*

\* 演示字符串定义的第一种方式，并明确字符串常量池的特点.

\* 池中没有就建立，池中有，直接用。

\*/

**private static void** stringDemo1() {

String s = "abc";//"abc"存储在字符串常量池中。

// s = "nba";

String s1 = "abc"; System.*out*.println(s==s1);//true?

// System.out.println("s="+s);

}

}

# 99.String类中的方法及使用：

**package** cn.itcast.p1.string.demo;

**public class** StringMethodDemo {

/\*\*

\* **@param** args

\*/

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

/\*

\* 按照面向对象的思想对字符串进行功能分类。

* "abcd"

\*

\* 1,获取：

\* 1.1 获取字符串中字符的个数(长度).

* int length();

\* 1.2 根据位置获取字符。

* char charAt(int index);

\* 1.3 根据字符获取在字符串中的第一次出现的位置.

* int indexOf(int ch)
* int indexOf(int ch,int fromIndex):从指定位置进行ch的查找第一次出现位置
* int indexOf(String str);
* int indexOf(String str,int fromIndex);

\* 根据字符串获取在字符串中的第一次出现的位置.

* int lastIndexOf(int ch)
* int lastIndexOf(int ch,int fromIndex):从指定位置进行ch的查找第一次出现位置
* int lastIndexOf(String str);
* int lastIndexOf(String str,int fromIndex);

\* 1.4 获取字符串中一部分字符串。也叫子串.

* String substring(int beginIndex, int endIndex)//包含begin 不包含end 。
* String substring(int beginIndex);

\*

\*

\* 2，转换。

\* 2.1 将字符串变成字符串数组(字符串的切割)

* String[] split(String regex):涉及到正则表达式.

\* 2.2 将字符串变成字符数组。

* char[] toCharArray();

\* 2.3 将字符串变成字节数组。

* byte[] getBytes();

\* 2.4 将字符串中的字母转成大小写。

* String toUpperCase():大写
* String toLowerCase():小写

\* 2.5 将字符串中的内容进行替换

* String replace(char oldch,char newch);
* String replace(String s1,String s2);

\* 2.6 将字符串两端的空格去除。

* String trim();

\* 2.7 将字符串进行连接 。

* String concat(string);

\*

\* 3，判断

\* 3.1 两个字符串内容是否相同啊？

* boolean equals(Object obj);
* boolean equalsIgnoreCase(string str);忽略大写比较字符串内容。

\* 3.2 字符串中是否包含指定字符串？

* boolean contains(string str);

\* 3.3 字符串是否以指定字符串开头。是否以指定字符串结尾。

* boolean startsWith(string);
* boolean endsWith(string);

\*

\* 4，比较。

\*

\*/

*stringMethodDemo\_4*();

// System.out.println("abc".concat("kk"));

// System.out.println("abc"+"kk");

// System.out.println(String.valueOf(4)+1);

// System.out.println(""+4+1);

}

**private static void** stringMethodDemo\_4() {

System.*out*.println("abc".compareTo("aqz"));

}

**private static void** stringMethodDemo\_3() { String s = "abc";

System.*out*.println(s.equals("ABC".toLowerCase())); System.*out*.println(s.equalsIgnoreCase("ABC"));

System.*out*.println(s.contains("cc")); String str = "ArrayDemo.java";

System.*out*.println(str.startsWith("Array")); System.*out*.println(str.endsWith(".java")); System.*out*.println(str.contains("Demo"));

}

**private static void** stringMethodDemo\_2() {

String s = "张三,李四,王五"; String[] arr = s.split(",");

**for** (**int** i = 0; i < arr.length; i++) { System.*out*.println(arr[i]);

}

**char**[] chs = s.toCharArray();

**for** (**int** i = 0; i < chs.length; i++) { System.*out*.println(chs[i]);

}

s = "ab你";

**byte**[] bytes = s.getBytes();

**for** (**int** i = 0; i < bytes.length; i++) { System.*out*.println(bytes[i]);

}

System.*out*.println("Abc".toUpperCase()); String s1 = "java";

String s2 = s1.replace('q', 'z'); System.*out*.println(s1==s2);//true

System.*out*.println("-"+" ab c ".trim()+"-");

}

**private static void** stringMethodDemo\_1() {

String s = "abcdae";

System.*out*.println("length:"+s.length());//6

System.*out*.println("char:"+s.charAt(2));//c//StringIndexOutOfBoundsException System.*out*.println("index:"+s.indexOf('k'));//0//-1 我们可以根据-1，来判断该字

符或者字符串是否存在。

System.*out*.println("lastIndex:"+s.lastIndexOf('a'));//4

System.*out*.println("substring:"+s.substring(2,4));

}

}

100.String类精华练习题：

习题1：

**package** cn.itcast.p1.string.test;

/\*

\* 1，给定一个字符串数组。按照字典顺序进行从小到大的排序。

\* {"nba","abc","cba","zz","qq","haha"}

\*

\* 思路：

\* 1,对数组排序。可以用选择，冒泡都行。

\* 2,for嵌套和比较以及换位。

\* 3,问题：以前排的是整数，比较用的比较运算符，可是现在是字符串对象。

\* 字符串对象怎么比较呢？爽了，对象中提供了用于字符串对象比较的功能。

\*/

**public class** StringTest\_1 {

/\*\*

\* **@param** args

\*/

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

String[] arr = { "nba", "abc", "cba", "zz", "qq", "haha" };

*printArray*(arr); *sortString*(arr); *printArray*(arr);

}

**public static void** sortString(String[] arr) {

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

**for** (**int** j = i + 1; j < arr.length; j++) {

**if**(arr[i].compareTo(arr[j])>0)//字符串比较用compareTo方法

*swap*(arr,i,j);

}

}

}

**private static void** swap(String[] arr, **int** i, **int** j) { String temp = arr[i];

arr[i] = arr[j]; arr[j] = temp;

}

**public static void** printArray(String[] arr) { System.*out*.print("[");

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

**if** (i != arr.length - 1) System.*out*.print(arr[i] + ", ");

### else

System.*out*.println(arr[i] + "]");

}

}

}

习题2：

**package** cn.itcast.p1.string.test;

/\*

\* 2，一个子串在整串中出现的次数。

\* "nbaernbatynbauinbaopnba"

\* 思路：

\* 1，要找的子串是否存在，如果存在获取其出现的位置。这个可以使用indexOf完成。

\* 2，如果找到了，那么就记录出现的位置并在剩余的字符串中继续查找该子串，

\* 而剩余字符串的起始位是出现位置+子串的长度.

\* 3,以此类推，通过循环完成查找，如果找不到就是-1，并对 每次找到用计数器记录。

\*/

**public class** StringTest\_2 {

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

String str = "nbaernbatnbaynbauinbaopnba"; String key = "nba";

**int** count = *getKeyStringCount\_2*(str,key); System.*out*.println("count="+count);

}

**public static int** getKeyStringCount\_2(String str, String key) {

**int** count = 0;

**int** index = 0;

**while**((index = str.indexOf(key,index))!=-1){

index = index + key.length(); count++;

}

**return** count;

}

/\*\*

\* 获取子串在整串中出现的次数。

* **@param** str
* **@param** key

### @return

\*/

**public static int** getKeyStringCount(String str, String key) {

//1,定义计数器。

**int** count = 0;

//2，定义变量记录key出现的位置。

**int** index = 0;

**while**((index = str.indexOf(key))!=-1){

str = str.substring(index+key.length()); count++;

}

**return** count;

}

}

习题3：

**package** cn.itcast.p1.string.test;

/\*

\* 3，两个字符串中最大相同的子串。

* "qwerabcdtyuiop"
* "xcabcdvbn"

\*

\* 思路：

\* 1，既然取得是最大子串，先看短的那个字符串是否在长的那个字符串中。

\* 如果存在，短的那个字符串就是最大子串。

\* 2，如果不是呢，那么就将短的那个子串进行长度递减的方式去子串，去长串中判断是否存在。

\* 如果存在就已找到，就不用在找了。

\*/

**public class** StringTest\_3 {

/\*\*

\* **@param** args

\*/

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

String s1 = "qwerabcdtyuiop"; String s2 = "xcabcdvbn";

}

/\*\*

String s = *getMaxSubstring*(s2, s1); System.*out*.println("s=" + s);

\* 获取最大子串

* **@param** s1
* **@param** s2

### @return

\*/

**public static** String getMaxSubstring(String s1, String s2) {

String max = **null**,min = **null**;

max = (s1.length()>s2.length())?s1:s2; min = max.equals(s1)?s2:s1;

System.*out*.println("max="+max); System.*out*.println("min="+min);

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

**for**(**int** a = 0,b = min.length()-i; b != min.length()+1; a++,b++){ String sub = min.substring(a, b);

// System.out.println(sub);

**if**(max.contains(sub)) **return** sub;

}

}

### return null;

}

}

习题4：

**package** cn.itcast.p1.string.test;

/\*

\* 4，模拟一个trim功能一致的方法。去除字符串两端的空白

\* 思路：

\* 1，定义两个变量。

\* 一个变量作为从头开始判断字符串空格的角标。不断++。

\* 一个变量作为从尾开始判断字符串空格的角标。不断--。

\* 2,判断到不是空格为止，取头尾之间的字符串即可。

\*/

**public class** StringTest\_4 {

/\*\*

\* **@param** args

\*/

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

String s = " ab c "; s = *myTrim*(s);

System.*out*.println("-" + s + "-");

}

**public static** String myTrim(String s) {

**int** start = 0, end = s.length() - 1;

**while** (start <= end && s.charAt(start) == ' ') { start++;

}

**while** (start <= end && s.charAt(end) == ' ') { end--;

}

**return** s.substring(start, end + 1);

}

}

# 101.StringBuffer及其方法使用：

**package** cn.itcast.p2.stringbuffer.demo;

**public class** StringBufferDemo {

/\*\*

\* **@param** args

\*/

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

/\*

* StringBuffer:就是字符串缓冲区。

\* 用于存储数据的容器。

\* 特点：

\* 1，长度的可变的。

\* 2，可以存储不同类型数据。

\* 3，最终要转成字符串进行使用。

\* 4，可以对字符串进行修改。

\*

\* 既然是一个容器对象。应该具备什么功能呢？

\* 1，添加：

* StringBuffer append(data);
* StringBuffer insert(index,data);

\* 2，删除：

* StringBuffer delete(start,end):包含头，不包含尾。
* StringBuffer deleteCharAt(int index):删除指定位置的元素

\* 3，查找：

* char charAt(index);
* int indexOf(string);
* int lastIndexOf(string);

\* 4，修改：

* StringBuffer replace(start,end,string);
* void setCharAt(index,char);

\*

* 增删改查 C(create)U(update)R(read)D(delete)

\*/

*bufferMethodDemo\_2*();

}

**private static void** bufferMethodDemo\_2() { StringBuffer sb = **new** StringBuffer("abce");

// sb.delete(1, 3);//ae

//清空缓冲区。

// sb.delete(0,sb.length());

// sb = new StringBuffer();

// sb.replace(1, 3, "nba");

// sb.setCharAt(2, 'q');

// sb.setLength(10);

// System.out.println("sb:"+sb);

// System.out.println("len:"+sb.length());

System.*out*.println(sb.reverse());

}

**private static void** bufferMethodDemo\_1() { StringBuffer sb = **new** StringBuffer("abce");

// sb.append("xixi"); sb.insert(2, "qq");

System.*out*.println(sb.toString());

}

**public static void** bufferMethodDemo(){

//创建缓冲区对象。

StringBuffer sb = **new** StringBuffer();

sb.append(4).append(**false**);//.append("haha"); sb.insert(1, "haha");

// sb.append(true); System.*out*.println(sb);

}

}

# 102.StringBuilder使用：

**package** cn.itcast.p2.stringbuffer.demo;

**public class** StringBuilderDemo {

/\*\*

\* **@param** args

\*/

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

/\*

* jdk1.5以后出现了功能和StringBuffer一模一样的对象。就是StringBuilder

\*

\* 不同的是：

\* StringBuffer是线程同步的。通常用于多线程。

\* StringBuilder是线程不同步的。通常用于单线程。 它的出现提高效率。

\*

* jdk升级：

\* 1，简化书写。

\* 2，提高效率。

\* 3，增加安全性。

\*/

}

}

习题：

**package** cn.itcast.p2.stringbuffer.test;

**public class** StringBuilderTest {

/\*\*

\* **@param** args

\*/

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

**int**[] arr = {3,1,5,3,8};

String s = *arrayToString\_2*(arr);

System.*out*.println(s);

}

**public static** String arrayToString\_2(**int**[] arr){

StringBuilder sb = **new** StringBuilder(); sb.append("[");

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

**if**(i!=arr.length-1) sb.append(arr[i]+", ");

### else

sb.append(arr[i]+"]");

}

/\*\*

}

**return** sb.toString();

\* 将一个int数组变成字符串。

\*/

**public static** String arrayToString(**int**[] arr){

String str = "[";

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

**if**(i!=arr.length-1) str+=arr[i]+", ";

### else

str+=arr[i]+"]";

}

**return** str;

}

}

103.包装类及方法使用：

**package** cn.itcast.p2.wrapper.demo;

**public class** WrapperDemo {

/\*\*

\* **@param** args

\*/

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

/\*

\* 基本数据类型对象包装类。

\* 为了方便操作基本数据类型值，将其封装成了对象，在对象中定义了属性和行为丰富了该数据的操

\* 作。

\* 用于描述该对象的类就称为基本数据类型对象包装类。

\*

* byte Byte
* short Short
* int Integer
* long Long
* float Float
* double Double
* char Character
* boolean Boolean

\*

\* 该包装对象主要用基本类型和字符串之间的转换。

\*

\* 基本类型--->字符串

\* 1,基本类型数值+""

\* 2,用String类中的静态方法valueOf(基本类型数值);

\* 3,用Integer的静态方法valueO(基本类型数值);

\*

\* 字符串--->基本类型

\* 1,使用包装类中的静态方法 xxx parseXxx("xxx类型的字符串");\*\*\*\*\*

* int parseInt("intstring");
* long parseLong("longstring");
* boolean parseBoolean("booleanstring");
* 只有Character没有parse方法

\* 2,如果字符串被Integer进行对象的封装。

\* 可使用另一个非静态的方法，intValue();

\* 将一个Integer对象转成基本数据类型值。

\*/

// System.out.println(Integer.MAX\_VALUE);

// System.out.println(Integer.toBinaryString(-6));

// int num = 4;

// Integer i = new Integer(5);

// int x = Integer.parseInt("123");

// System.out.println(Integer.parseInt("123")+1);

// Integer i = new Integer("123");

// System.out.println(i.intValue());

/\*

\* 整数具备不同的进制体现。

\*

\* 十进制-->其他进制。

* toBinaryString
* toOctalString
* toHexString

\*

\* 其他进制-->十进制。

* parseInt("string",radix)

\*

\*/

// 十进制-->其他进制。 System.*out*.println(Integer.*toBinaryString*(60)); System.*out*.println(Integer.*toOctalString*(60)); System.*out*.println(Integer.*toHexString*(60));

// System.out.println(Integer.toString(60,16));

// 其他进制-->十进制。

// System.out.println(Integer.parseInt("3c",16));

Integer a = **new** Integer("89"); Integer b = **new** Integer(300);

System.*out*.println(a==b); System.*out*.println(a.equals(b));

// System.out.println(3>3); System.*out*.println(a.compareTo(b));

}

}

104.自劢装箱：

**package** cn.itcast.p2.wrapper.demo;

**public class** WrapperDemo2 {

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

**int** num = 4; num = num + 5;

Integer i = 4;//i = new Integer(4);自动装箱 简化书写。

i = i + 6;// i = new Integer(i.intValue() + 6); //i.intValue() 自动拆箱

// show(55);//

Integer a = **new** Integer(128); Integer b = **new** Integer(128);

System.*out*.println(a==b);//false System.*out*.println(a.equals(b));//true

Integer x = 129;//jdk1.5以后，自动装箱，如果装箱的是一个字节，那么该数据会被共享不会 重新开辟空间。

Integer y = 129; System.*out*.println(x==y);// System.*out*.println(x.equals(y));//true

}

**public static void** show(Object a){//Object a = new Integer(55); System.*out*.println("a="+a);

}

}

105.包装类练习题：

**package** cn.itcast.p2.wrapper.test;

**import** java.util.Arrays;

/\*

\* 对一个字符串中的数值进行从小到大的排序。

\*

\* "20 78 9 -7 88 36 29"

\*

\* 思路：

\* 1，排序，我很熟。可是我只熟int。

\* 2，如何获取到这个字符串中的这些需要排序的数值？

\* 发现这个字符串中其实都是空格来对数值进行分隔的。

\* 所以就想到用字符串对象的切割方法将大串变成多个小串。

\* 3，数值最终变成小字符串，怎么变成一个int数呢？

\* 字符串-->基本类型 可以使用包装类。

\*

\*

\*/

**public class** WrapperTest {

**private static final** String *SPACE\_SEPARATOR* = " ";

/\*\*

\* **@param** args

\*/

**public static void** main(String[] args) { String numStr = "20 78 9 -7 88 36 29";

System.*out*.println(numStr); numStr = *sortStringNumber*(numStr); System.*out*.println(numStr);

}

/\*\*

\*

* **@param** numStr

### @return

\*/

**public static** String sortStringNumber(String numStr) {

//1,将字符串变成字符串数组。

String[] str\_arr = *stringToArray*(numStr);

//2,将字符串数组变成int数组。

**int**[] num\_arr = *toIntArray*(str\_arr);

//3,对int数组排序。

*mySortArray*(num\_arr);

//4,将排序后的int数组变成字符串。

String temp = *arrayToString*(num\_arr);

**return** temp;

}

**public static** String arrayToString(**int**[] num\_arr) {

StringBuilder sb = **new** StringBuilder();

**for**(**int** x = 0; x<num\_arr.length; x++){

**if**(x!=num\_arr.length-1) sb.append(num\_arr[x]+*SPACE\_SEPARATOR*);

### else

sb.append(num\_arr[x]);

}

**return** sb.toString();

}

**public static void** mySortArray(**int**[] num\_arr) {

Arrays.*sort*(num\_arr);

}

**public static int**[] toIntArray(String[] str\_arr) {

**int**[] arr = **new int**[str\_arr.length]; **for** (**int** i = 0; i < arr.length; i++) {

arr[i] = Integer.*parseInt*(str\_arr[i]);

}

**return** arr;

}

/\*\*

\* **@param** numStr

\*/

**public static** String[] stringToArray(String numStr) { String[] str\_arr = numStr.split(*SPACE\_SEPARATOR*);

**return** str\_arr;

}

}

# 第七章：集合框架（容器）

106.集合常用方法演示：

package cn.itcast.p3.collection.demo;

import java.util.ArrayList; import java.util.Collection;

public class CollectionDemo {

/\*\*

\* @param args

\*/

public static void main(String[] args) { Collection coll = new ArrayList();

// show(coll);

Collection c1 = new ArrayList(); Collection c2 = new ArrayList(); show(c1,c2);

}

public static void show(Collection c1,Collection c2){

//给c1添加元素。 c1.add("abc1");

c1.add("abc2");

c1.add("abc3");

c1.add("abc4");

//给c2添加元素。 c2.add("abc1");

c2.add("abc2");

c2.add("abc3");

c2.add("abc4");

c2.add("abc5");

System.out.println("c1:"+c1); System.out.println("c2:"+c2);

//演示addAll

// c1.addAll(c2);//将c2中的元素添加到c1中。

//演示removeAll

// boolean b = c1.removeAll(c2);//将两个集合中的相同元素从调用removeAll的集合中删除。

// System.out.println("removeAll:"+b);

//演示containsAll

// boolean b = c1.containsAll(c2);

// System.out.println("containsAll:"+b);

//演示retainAll

boolean b = c1.retainAll(c2);//取交集，保留和指定的集合相同的元素，而删除不同的元素。

//和removeAll功能相反 。 System.out.println("retainAll:"+b); System.out.println("c1:"+c1);

}

public static void show(Collection coll){

//1,添加元素。add. coll.add("abc1");

coll.add("abc2");

coll.add("abc3"); System.out.println(coll);

//2，删除元素。remove

// coll.remove("abc2");//会改变集合的长度

//清空集合.

// coll.clear(); System.out.println(coll.contains("abc3")); System.out.println(coll);

}

}

107.Iterator迭代器的使用：

package cn.itcast.p3.collection.demo;

import java.util.ArrayList; import java.util.Collection; import java.util.Iterator;

public class IteratorDemo {

/\*\*

\* @param args

\*/

public static void main(String[] args) {

Collection coll = new ArrayList(); coll.add("abc1");

coll.add("abc2");

coll.add("abc3");

coll.add("abc4");

// System.out.println(coll);

//使用了Collection中的iterator()方法。 调用集合中的迭代器方法，是为了获取集合中的迭代 器对象。

// Iterator it = coll.iterator();

// while(it.hasNext()){

// System.out.println(it.next());

// }

for(Iterator it = coll.iterator(); it.hasNext(); ){ System.out.println(it.next());

}

// System.out.println(it.next());

// System.out.println(it.next());

// System.out.println(it.next());

// System.out.println(it.next());

// System.out.println(it.next());//java.util.NoSuchElementException

}

}

108.ArrayList：

演示 1：

package cn.itcast.p4.list.demo;

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

public class ListDemo {

/\*\*

\* @param args

\*/

public static void main(String[] args) { List list = new ArrayList(); show(list);

}

public static void show(List list) {

//添加元素 list.add("abc1");

list.add("abc2");

list.add("abc3"); System.out.println(list);

//插入元素。

// list.add(1,"abc9");

//删除元素。

// System.out.println("remove:"+list.remove(2));

//修改元素。

// System.out.println("set:"+list.set(1, "abc8"));

//获取元素。

// System.out.println("get:"+list.get(0));

//获取子列表。

// System.out.println("sublist:"+list.subList(1, 2)); System.out.println(list);

}

}

演示 2：

package cn.itcast.p4.list.demo;

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

import java.util.ListIterator;

public class ListDemo2 {

/\*\*

\* @param args

\*/

public static void main(String[] args) {

List list = new ArrayList();

// show(list);

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

ListIterator it = list.listIterator();//获取列表迭代器对象

//它可以实现在迭代过程中完成对元素的增删改查。

//注意：只有list集合具备该迭代功能. while(it.hasNext()){

Object obj = it.next();

if(obj.equals("abc2")){

it.set("abc9");

}

}

// System.out.println("hasNext:"+it.hasNext());

// System.out.println("hasPrevious:"+it.hasPrevious());

while(it.hasPrevious()){ System.out.println("previous:"+it.previous());

}

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

/\*Iterator it = list.iterator(); while(it.hasNext()){

Object obj = it.next();//java.util.ConcurrentModificationException

//在迭代器过程中，不要使用集合操作元素，容易出现异常。

//可以使用Iterator接口的子接口ListIterator来完成在迭代中对元素进行更

//多的操作。

if(obj.equals("abc2")){

list.add("abc9");

}

else

System.out.println("next:"+obj);

}

System.out.println(list);

\*/

}

public static void show(List list) { list.add("abc1");

Iterator it = list.iterator(); while(it.hasNext()){

System.out.println("next:"+it.next());

}

//list特有的取出元素的方式之一。 for(int x=0; x<list.size(); x++){

System.out.println("get:"+list.get(x));

}

}

}

109.Vector演示:

package cn.itcast.p1.vector.demo;

import java.util.Enumeration; import java.util.Iterator; import java.util.Vector;

public class VectorDemo {

/\*\*

\* @param args

\*/

public static void main(String[] args) { Vector v = new Vector();

v.addElement("abc1"); v.addElement("abc2"); v.addElement("abc3"); v.addElement("abc4");

Enumeration en = v.elements(); while(en.hasMoreElements()){

System.out.println("nextelment:"+en.nextElement());

}

Iterator it = v.iterator(); while(it.hasNext()){

System.out.println("next:"+it.next());

}

}

}

110.LinkedList演示：

package cn.itcast.p2.linkedlist.demo;

import java.util.Iterator; import java.util.LinkedList;

public class LinkedListDemo {

/\*\*

\* @param args

\*/

public static void main(String[] args) {

LinkedList link = new LinkedList();

link.addFirst("abc1"); link.addFirst("abc2"); link.addFirst("abc3"); link.addFirst("abc4");

// System.out.println(link);

// System.out.println(link.getFirst());//获取第一个但不删除。

// System.out.println(link.getFirst());

// System.out.println(link.removeFirst());//获取元素但是会删除。

// System.out.println(link.removeFirst());

while(!link.isEmpty()){ System.out.println(link.removeLast());

}

System.out.println(link);

// Iterator it = link.iterator();

// while(it.hasNext()){

// System.out.println(it.next());

// }

}

}

# 111.队列的添加元素功能（先迚先出）：

**package** cn.itcast.p2.linkedlist.test;

**import** java.util.LinkedList;

**public class** DuiLie {

**private** LinkedList link;

**public** DuiLie() {

link = **new** LinkedList();

}

/\*\*

\* 队列的添加元素的功能。

\*/

**public void** myAdd(Object obj) { link.addLast(obj);

}

**public** Object myGet() {

**return** link.removeFirst();

}

**public boolean** isNull() {

**return** link.isEmpty();

}

}

习题：

**package** cn.itcast.p2.linkedlist.test;

**import** java.util.ArrayList;

/\*

\* 请使用LinkedList来模拟一个堆栈或者队列数据结构。

\*

* 堆栈：先进后出 First In Last Out FILO

\*

* 队列：先进先出 First In First Out FIFO

\*

\* 我们应该描述这样一个容器，给使用提供一个容器对象完成这两种结构中的一种。

\*/

**public class** LinkedTest {

/\*\*

\* **@param** args

\*/

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

DuiLie dl = **new** DuiLie();

dl.myAdd("abc1");

dl.myAdd("abc2");

dl.myAdd("abc3");

dl.myAdd("abc4");

**while**(!dl.isNull()){ System.*out*.println(dl.myGet());

}

}

}

注意：解体的核心思想还是运用已有的、已知具备功能的方法来作为自己定义方法的内容来实现模拟的功能。

112.ArrayList习题演示：

习题1:

package cn.itcast.p3.arraylist.test;

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

import cn.itcast.p.bean.Person; public class ArrayListTest {

/\*\*

\* @param args

\*/

public static void main(String[] args) { Person p1 = new Person("lisi1",21);

ArrayList al = new ArrayList(); al.add(p1);

al.add(new Person("lisi2",22)); al.add(new Person("lisi3",23)); al.add(new Person("lisi4",24));

Iterator it = al.iterator(); while(it.hasNext()){

// System.out.println(((Person) it.next()).getName()+"::"+((Person) it.next()).getAge());

Person p = (Person) it.next(); System.out.println(p.getName()+"--"+p.getAge());

}

// al.add(5);//al.add(new Integer(5));

}

}

习题2：

package cn.itcast.p3.arraylist.test;

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

import cn.itcast.p.bean.Person;

/\*

\* 定义功能去除ArrayList中的重复元素。

\*/

public class ArrayListTest2 {

public static void main(String[] args) {

// demo();

// singleDemo();

ArrayList al = new ArrayList(); al.add(new Person("lisi1",21)); al.add(new Person("lisi2",22)); al.add(new Person("lisi3",23)); al.add(new Person("lisi4",24)); al.add(new Person("lisi2",22)); al.add(new Person("lisi3",23)); System.out.println(al);

al = getSingleElement(al);

System.out.println(al.remove(new Person("lisi2",22))); System.out.println(al);

}

public static void singleDemo() { ArrayList al = new ArrayList(); al.add("abc1");

al.add("abc2");

al.add("abc2");

al.add("abc1");

al.add("abc"); System.out.println(al);

al = getSingleElement(al); System.out.println(al);

}

public static ArrayList getSingleElement(ArrayList al) {

//1,定义一个临时容器。

ArrayList temp = new ArrayList();

//2,迭代al集合。

Iterator it = al.iterator();

while(it.hasNext()){ Object obj = it.next();

//3,判断被迭代到的元素是否在临时容器存在。 if(!temp.contains(obj)){

temp.add(obj);

}

}

return temp;

}

public static void demo() {

// al.add(5);//al.add(new Integer(5));

}

}

113.HashSet演示：

演示1：

package cn.itcast.p4.hashset.demo;

import java.util.HashSet; import java.util.Iterator;

public class HashSetDemo {

/\*\*

\* @param args

\*/

public static void main(String[] args) { HashSet hs = new HashSet();

hs.add("hehe");

// hs.add("heihei");

hs.add("hahah");

hs.add("xixii");

hs.add("hehe");

Iterator it = hs.iterator(); while(it.hasNext()){

System.out.println(it.next());

}

}

}

演示2：

package cn.itcast.p4.hashset.test;

import java.util.HashSet; import java.util.Iterator;

import cn.itcast.p.bean.Person;

/\*

\* 往hashSet集合中存储Person对象。如果姓名和年龄相同，视为同一个人。视为相同元素。

\*/

public class HashSetTest {

public static void main(String[] args) { HashSet hs = new HashSet();

/\*

\* HashSet集合数据结构是哈希表，所以存储元素的时候，

\* 使用的元素的hashCode方法来确定位置，如果位置相同，在通过元素的equals来确定是否相同。

\*

\*/

hs.add(new Person("lisi4",24)); hs.add(new Person("lisi7",27)); hs.add(new Person("lisi1",21)); hs.add(new Person("lisi9",29)); hs.add(new Person("lisi7",27));

Iterator it = hs.iterator();

while(it.hasNext()){

Person p = (Person)it.next(); System.out.println(p);

// System.out.println(p.getName()+"...."+p.getAge());

}

}

}

# 114.LinkedHashSet使用：

package cn.itcast.p4.hashset.demo;

import java.util.HashSet; import java.util.Iterator; import java.util.LinkedHashSet;

public class LinkedHashSetDemo {

public static void main(String[] args) { HashSet hs = new LinkedHashSet(); hs.add("hahah");

hs.add("hehe");

hs.add("heihei");

hs.add("xixii");

// hs.add("hehe");

Iterator it = hs.iterator(); while(it.hasNext()){

System.out.println(it.next());

}

}

}

# 115.TreeSet和比较方法（实现Comparator接口）使用

的习题：

习题1：

/\*\*

\* 创建了一个根据Person类的name进行排序的比较器。

\*/

**class** ComparatorByName **implements** Comparator {

@Override

**public int** compare(Object o1, Object o2) {

Person p1 = (Person)o1; Person p2 = (Person)o2;

**int** temp = p1.getName().compareTo(p2.getName());

**return** temp==0?p1.getAge()-p2.getAge(): temp;

// return 1;//有序。

}

}

**public class** TreeSetDemo {

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

TreeSet ts = **new** TreeSet(**new** ComparatorByName());

/\*

\* 以Person对象年龄进行从小到大的排序。

\*

\*/

ts.add(**new** Person("zhangsan",28)); ts.add(**new** Person("lisi",21)); ts.add(**new** Person("zhouqi",29)); ts.add(**new** Person("zhaoliu",25)); ts.add(**new** Person("wangu",24));

Iterator it = ts.iterator();

**while**(it.hasNext()){

Person p = (Person)it.next();

System.*out*.println(p.getName()+":"+p.getAge());

}

}

**public static void** demo1() { TreeSet ts = **new** TreeSet();

ts.add("abc");

ts.add("zaa");

ts.add("aa");

ts.add("nba");

ts.add("cba");

Iterator it = ts.iterator();

**while**(it.hasNext()){ System.*out*.println(it.next());

}

}

}

习题2：

/\*

定义一个按长度作比较的比较器

\*/

**class** ComparatorByLength **implements** Comparator {

@Override

**public int** compare(Object o1, Object o2) {

String s1 = (String)o1; String s2 = (String)o2;

**int** temp = s1.length()-s2.length();

**return** temp==0? s1.compareTo(s2): temp;

}

}

/\*

\* 对字符串进行长度排序。

\*

\* "20 18 -1 89 2 67"

\*/

**public class** TreeSetTest {

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

TreeSet ts = **new** TreeSet(**new** ComparatorByLength());

ts.add("aaaaa");

ts.add("zz");

ts.add("nbaq");

ts.add("cba");

ts.add("abc");

Iterator it = ts.iterator();

**while**(it.hasNext()){ System.*out*.println(it.next());

}

}

}

# 116. 比较的另一种方法即让对象实现Comparable接 口，即让对象本身具备可比较性并重写compareTo（） 方法，实际上就是改发return表达式，但丌如实现 Comparator接口迚行比较更为灵活，案例：

**package** cn.itcast.p.bean;

**public class** Person /\*extends Object\*/ **implements** Comparable {

**private** String name;

**private int** age;

**public** Person() {

**super**();

}

**public** Person(String name, **int** age) {

**super**(); **this**.name = name; **this**.age = age;

}

@Override

**public int** hashCode() {

// System.out.println(this+".......hashCode");

**return** name.hashCode()+age\*27;

// return 100;

}

@Override

**public boolean** equals(Object obj) {

**if**(**this** == obj)

### return true;

**if**(!(obj **instanceof** Person))

**throw new** ClassCastException("类型错误");

// System.out.println(this+"....equals....."+obj); Person p = (Person)obj;

**return this**.name.equals(p.name) && **this**.age == p.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;

}

**public** String toString(){

**return** name+":"+age;

}

@Override

**public int** compareTo(Object o) { Person p = (Person)o;

**int** temp = **this**.age-p.age;

**return** temp==0?**this**.name.compareTo(p.name):temp;

// int temp = this.name.compareTo(p.name);

// return temp==0?this.age-p.age:temp;

/\* if(this.age>p.age)

return 1; if(this.age<p.age)

return -1;

else{

return this.name.compareTo(p.name);

}

\*/

}

}

# 117.集合（List、Set）部分总结：

集合类的由来： 对象用于封装特有数据，对象多了需要存储，如果对象的个数不确定。 就使用集合容器进行存储。

集合特点：

1，用于存储对象的容器。

2，集合的长度是可变的。

3，集合中不可以存储基本数据类型值。

集合容器因为内部的数据结构不同，有多种具体容器。 不断的向上抽取，就形成了集合框架。

框架的顶层Collection接口： Collection的常见方法： 1，添加。

boolean add(Object obj): boolean addAll(Collection coll):

2，删除。

boolean remove(object obj): boolean removeAll(Collection coll); void clear();

3，判断：

boolean contains(object obj):

boolean containsAll(Colllection coll); boolean isEmpty():判断集合中是否有元素。

4，获取：

int size():

Iterator iterator():取出元素的方式：迭代器。 该对象必须依赖于具体容器，因为每一个容器的数据结构都不同。 所以该迭代器对象是在容器中进行内部实现的。 对于使用容器者而言，具体的实现不重要，只要通过容器获取到该实现的迭代器的对象即可， 也就是iterator方法。

Iterator接口就是对所有的Collection容器进行元素取出的公共接口。 其实就是抓娃娃游戏机中的夹子！

5，其他：

boolean retainAll(Collection coll);取交集。 Object[] toArray():将集合转成数组。

--------------------------------------------------------------------------------

Collection

|--List：有序(存入和取出的顺序一致),元素都有索引(角标)，元素可以重复。

|--Set：元素不能重复,无序。

List:特有的常见方法：有一个共性特点就是都可以操作角标。 1，添加

void add(index,element); void add(index,collection);

2，删除；

Object remove(index):

3，修改：

Object set(index,element);

4，获取：

Object get(index); int indexOf(object);

int lastIndexOf(object); List subList(from,to);

list集合是可以完成对元素的增删改查。

List:

|--Vector:内部是数组数据结构，是同步的。增删，查询都很慢！

|--ArrayList:内部是数组数据结构，是不同步的。替代了Vector。查询的速度快。

|--LinkedList:内部是链表数据结构，是不同步的。增删元素的速度很快。

LinkedList:

addFirst(); addLast(): jdk1.6 offerFirst(); offetLast();

getFirst();.//获取但不移除，如果链表为空，抛出NoSuchElementException. getLast();

jdk1.6 peekFirst();//获取但不移除，如果链表为空，返回null. peekLast():

removeFirst();//获取并移除，如果链表为空，抛出NoSuchElementException. removeLast();

jdk1.6 pollFirst();//获取并移除，如果链表为空，返回null. pollLast();

作业：

1，自己去查文档演示Vector中的elements()方法。

2，LinkedList中的，addFirst addLast getFirst，getLast removeFirst removeLast。 3，既然集合是存储对象的，请定义ArryaList集合，并存储Person对象。如new Person("lisi",20);

并取出。将姓名和年龄打印出来。

--------------------------------------------------------------------------------

Set:元素不可以重复，是无序。 Set接口中的方法和Collection一致。

|--HashSet: 内部数据结构是哈希表 ，是不同步的。 如何保证该集合的元素唯一性呢？ 是通过对象的hashCode和equals方法来完成对象唯一性的。 如果对象的hashCode值不同，那么不用判断equals方法，就直接存储到哈希表中。 如果对象的hashCode值相同，那么要再次判断对象的equals方法是否为true。

如果为true，视为相同元素，不存。如果为false，那么视为不同元素，就进行存储。

方法。

记住：如果元素要存储到HashSet集合中，必须覆盖hashCode方法和equals方法。 一般情况下，如果定义的类会产生很多对象，比如人，学生，书，通常都需要覆盖equals，hashCode

建立对象判断是否相同的依据。

|--TreeSet:可以对Set集合中的元素进行排序。是不同步的。 判断元素唯一性的方式：就是根据比较方法的返回结果是否是0，是0，就是相同元素，不存。

TreeSet对元素进行排序的方式一： 让元素自身具备比较功能，元就需要实现Comparable接口。覆盖compareTo方法。

如果不要按照对象中具备的自然顺序进行排序。如果对象中不具备自然顺序。怎么办？ 可以使用TreeSet集合第二种排序方式二： 让集合自身具备比较功能，定义一个类实现Comparator接口，覆盖compare方法。 将该类对象作为参数传递给TreeSet集合的构造函数。

if(this.hashCode()== obj.hashCode() && this.equals(obj))

哈希表确定元素是否相同

1，判断的是两个元素的哈希值是否相同。 如果相同，在判断两个对象的内容是否相同。

2，判断哈希值相同，其实判断的是对象的hashCode的方法。判断内容相同，用的是equals方法。 注意：如果哈希值不同，是不需要判断equals。

118.泛型简述：

**package** cn.itcast.p4.generic.define.demo;

/\*

public class Tool { private Object object;

public Object getObject() { return object;

}

public void setObject(Object object) { this.object = object;

}

}

\*/

//在jdk1.5后，使用泛型来接收类中要操作的引用数据类型。

//泛型类。什么时候用？当类中的操作的引用数据类型不确定的时候，就使用泛型来表示。

**public class** Tool<QQ>{

**private** QQ q;

**public** QQ getObject() {

**return** q;

}

**public void** setObject(QQ object) {

**this**.q = object;

}

/\*\*

\* 将泛型定义在方法上。

\* **@param** str

\*/

**public** <W> **void** show(W str){ System.*out*.println("show : "+str.toString());

}

**public void** print(QQ str){ System.*out*.println("print : "+str);

}

/\*\*

\* 当方法静态时，不能访问类上定义的泛型。如果静态方法使用泛型，

\* 只能将泛型定义在方法上。

\* **@param** obj

\*/

**public static** <Y> **void** method(Y obj){ System.*out*.println("method:"+obj);

}

}

119.泛型演示：

示例1：

package cn.itcast.p5.generic.advance.demo;

import java.util.ArrayList; import java.util.Collection; import java.util.Iterator;

import cn.itcast.p2.bean.Person; import cn.itcast.p2.bean.Student;

import cn.itcast.p2.bean.Worker;

public class GenericAdvanceDemo2 {

/\*\*

\* @param args

\*/

public static void main(String[] args) { ArrayList<Person> al = new ArrayList<Person>();

al.add(new Person("abc",30));

al.add(new Person("abc4",34));

ArrayList<Student> al2 = new ArrayList<Student>(); al2.add(new Student("stu1",11));

al2.add(new Student("stu2",22)); ArrayList<String> al3 = new ArrayList<String>();

al3.add("stu3331"); al3.add("stu33332");

}

/\*\*

printCollection(al2); printCollection(al);

\* 迭代并打印集合中元素。

\*

\* 可以对类型进行限定：

\* ? extends E:接收E类型或者E的子类型对象。上限！

\*

\* ? super E ：接收E类型或者E的父类型。下限！

\* @param al

\*/

/\*public static void printCollection(Collection<? extends Person> al)

{//Collection<Dog> al = new ArrayList<Dog>() Iterator<? extends Person> it = al.iterator();

while(it.hasNext()){

// T str = it.next();

// System.out.println(str);

// System.out.println(it.next().toString()); Person p = it.next();

System.out.println(p.getName()+":"+p.getAge());

}

}\*/

public static void printCollection(Collection<? super Student> al){ Iterator<? super Student> it = al.iterator();

while(it.hasNext()){

System.out.println(it.next());

}

}

}

示例2（存储用上限）：

package cn.itcast.p5.generic.advance.demo; import java.util.ArrayList;

import cn.itcast.p2.bean.Person; import cn.itcast.p2.bean.Student; import cn.itcast.p2.bean.Worker;

public class GenericAdvanceDemo3 {

public static void main(String[] args) { ArrayList<Person> al1 = new ArrayList<Person>();

al1.add(new Person("abc", 30));

al1.add(new Person("abc4", 34)); ArrayList<Student> al2 = new ArrayList<Student>();

al2.add(new Student("stu1", 11));

al2.add(new Student("stu2", 22)); ArrayList<Worker> al3 = new ArrayList<Worker>();

al3.add(new Worker("stu1", 11));

al3.add(new Worker("stu2", 22));

ArrayList<String> al4 = new ArrayList<String>(); al4.add("abcdeef");

// al1.addAll(al4);//错误，类型不匹配。

al1.addAll(al2); al1.addAll(al3);

System.out.println(al1.size());

// printCollection(al2);

// printCollection(al);

}

}

/\*

\* 一般在存储元素的时候都是用上限，因为这样取出都是按照上限类型来运算的，不会出现类型安全隐患。

\*/

class MyCollection<E> { public void add(E e) {

}

public void addAll(MyCollection<? extends E> e) {

}

}

示例3（取出元素用下限）：

package cn.itcast.p5.generic.advance.demo;

import java.util.Comparator; import java.util.Iterator; import java.util.TreeSet;

import cn.itcast.p2.bean.Person; import cn.itcast.p2.bean.Student; import cn.itcast.p2.bean.Worker;

public class GenericAdvanceDemo4 {

public static void main(String[] args) {

TreeSet<Person> al1 = new TreeSet<Person>(new CompByName()); al1.add(new Person("abc4",34));

al1.add(new Person("abc1",30)); al1.add(new Person("abc2",38));

TreeSet<Student> al2 = new TreeSet<Student>(new CompByName());

al2.add(new Student("stu1",11)); al2.add(new Student("stu7",20)); al2.add(new Student("stu2",22));

TreeSet<Worker> al3 = new TreeSet<Worker>();

al3.add(new Worker("stu1",11)); al3.add(new Worker("stu2",22));

TreeSet<String> al4 = new TreeSet<String>(); al4.add("abcdeef");

// al1.addAll(al4);//错误，类型不匹配。

// al1.addAll(al2);

// al1.addAll(al3);

// System.out.println(al1.size());

Iterator<Student> it = al2.iterator(); while(it.hasNext()){

System.out.println(it.next());

}

}

}

/\*

* class TreeSet<Worker>

\* {

* Tree(Comparator<? super Worker> comp);

\* }

\*

\* 什么时候用下限呢？通常对集合中的元素进行取出操作时，可以是用下限。

\*

\*/

class CompByName implements Comparator<Person>{

@Override

public int compare(Person o1, Person o2) {

int temp = o1.getName().compareTo(o2.getName());

return temp==0? o1.getAge()-o2.getAge():temp;

}

}

class CompByStuName implements Comparator<Student>{

@Override

public int compare(Student o1, Student o2) {

int temp = o1.getName().compareTo(o2.getName());

return temp==0? o1.getAge()-o2.getAge():temp;

}

}

class CompByWorkerName implements Comparator<Worker>{

@Override

public int compare(Worker o1, Worker o2) {

int temp = o1.getName().compareTo(o2.getName());

return temp==0? o1.getAge()-o2.getAge():temp;

}

}

120.Map简述：

package cn.itcast.p6.map.demo;

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

public class MapDemo {

public static void main(String[] args) {

Map<Integer,String> map = new HashMap<Integer,String>(); method\_2(map);

}

public static void method\_2(Map<Integer,String> map){

map.put(8,"zhaoliu");

map.put(2,"zhaoliu");

map.put(2,"zhaoliu");

map.put(6,"wangcai");

Collection<String> values = map.values();

Iterator<String> it2 = values.iterator(); while(it2.hasNext()){

System.out.println(it2.next());

}

/\*

\* 通过Map转成set就可以迭代。

\* 找到了另一个方法。entrySet。

\* 该方法将键和值的映射关系作为对象存储到了Set集合中，而这个映射关系的类型就是Map.Entry

\* 类型(结婚证)

\*/

Set<Map.Entry<Integer, String>> entrySet = map.entrySet(); Iterator<Map.Entry<Integer, String>> it = entrySet.iterator();

while(it.hasNext()){

Map.Entry<Integer, String> me = it.next(); Integer key = me.getKey();

String value = me.getValue(); System.out.println(key+"::::"+value);

}

//取出map中的所有元素。

//原理，通过keySet方法获取map中所有的键所在的Set集合，在通过Set的迭代器获取到每一个键，

//在对每一个键通过map集合的get方法获取其对应的值即可。

/\*

Set<Integer> keySet = map.keySet(); Iterator<Integer> it = keySet.iterator();

while(it.hasNext()){ Integer key = it.next();

String value = map.get(key); System.out.println(key+":"+value);

}

\*/

}

public static void method(Map<Integer,String> map){//学号和姓名

// 添加元素。

System.out.println(map.put(8, "wangcai"));//null System.out.println(map.put(8, "xiaoqiang"));//wangcai 存相同键，值会覆盖。 map.put(2,"zhangsan");

map.put(7,"zhaoliu");

//删除。

// System.out.println("remove:"+map.remove(2));

//判断。

// System.out.println("containskey:"+map.containsKey(7));

//获取。 System.out.println("get:"+map.get(6));

System.out.println(map);

Outer.Inner.show();

}

}

interface MyMap{

public static interface MyEntry{//内部接口 void get();

}

}

class MyDemo implements MyMap.MyEntry{ public void get(){}

}

class Outer{

static class Inner{ static void show(){}

}

}

121.HashMap演示：

package cn.itcast.p7.hashmap.demo;

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

import cn.itcast.p2.bean.Student; public class HashMapDemo {

/\*\*

\* @param args

\*/

public static void main(String[] args) {

/\*

\* 将学生对象和学生的归属地通过键与值存储到map集合中。

\*/

HashMap<Student,String> hm = new HashMap<Student,String>(); hm.put(new Student("lisi",38),"北京");

hm.put(new Student("zhaoliu",24),"上海");

hm.put(new Student("xiaoqiang",31),"沈阳"); hm.put(new Student("wangcai",28),"大连");

hm.put(new Student("zhaoliu",24),"铁岭");

// Set<Student> keySet = hm.keySet();

// Iterator<Student> it = keySet.iterator();

Iterator<Student> it = hm.keySet().iterator(); while(it.hasNext()){

Student key = it.next(); String value = hm.get(key);

System.out.println(key.getName()+":"+key.getAge()+"---"+value);

}

}

}

122.TreeMap演示：

package cn.itcast.p8.treemap.demo;

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

import cn.itcast.p2.bean.Student;

import cn.itcast.p3.comparator.ComparatorByName; public class TreeMapDemo {

/\*\*

\* @param args

\*/

public static void main(String[] args) {

TreeMap<Student,String> tm = new TreeMap<Student,String>(new ComparatorByName());

tm.put(new Student("lisi",38),"北京");

tm.put(new Student("zhaoliu",24),"上海"); tm.put(new Student("xiaoqiang",31),"沈阳"); tm.put(new Student("wangcai",28),"大连");

tm.put(new Student("zhaoliu",24),"铁岭"); Iterator<Map.Entry<Student, String>> it = tm.entrySet().iterator();

while(it.hasNext()){ Map.Entry<Student,String> me = it.next(); Student key = me.getKey();

String value = me.getValue();

System.out.println(key.getName()+":"+key.getAge()+"---"+value);

}

}

}

# 123.Map集合不Collection集合的区别：

1， Map中一次存储是键值对。

Collection中一次存储是单个元素。 2，

Map的存储使用的put方法。 Collection存储使用的是add方法。 3，

Map的取出，是讲Map转成Set，在使用迭代器取出。 Collection取出，使用就是迭代器。

4， 如果对象很多，必须使用容器存储。

如果元素存在着映射关系，可以优先考虑使用Map存储或者用数组， 如果没有映射关系，可以使用Collection存储。

# 124.LinkedHashMap演示：

package cn.itcast.p1.map.demo;

import java.io.File; import java.util.HashMap; import java.util.Iterator;

import java.util.LinkedHashMap; import java.util.Map;

public class LinkedHashMapDemo {

/\*\*

\* @param args

\*/

public static void main(String[] args) {

File f= null;

HashMap<Integer,String> hm = new LinkedHashMap<Integer,String>();

hm.put(7, "zhouqi");

hm.put(3, "zhangsan");

hm.put(1, "qianyi");

hm.put(5, "wangwu");

Iterator<Map.Entry<Integer,String>> it = hm.entrySet().iterator(); while(it.hasNext()){

Map.Entry<Integer,String> me = it.next();

Integer key = me.getKey(); String value = me.getValue();

System.out.println(key+":"+value);

}

}

}

125.Map精华习题：

习题1：

package cn.itcast.p1.map.test;

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

/\*

\* 练习:

\* "fdgavcbsacdfs" 获取该字符串中，每一个字母出现的次数。

\* 要求打印结果是：a(2)b(1)...;

\* 思路：

\* 对于结果的分析发现，字母和次数之间存在着映射的关系。而且这种关系很多。

\* 很多就需要存储，能存储映射关系的容器有数组和Map集合。

\* 关系一方式有序编号吗？没有！

\* 那就是使用Map集合。 又发现可以保证唯一性的一方具备着顺序如 a b c ...

\* 所以可以使用TreeMap集合。

\*

\* 这个集合最终应该存储的是字母和次数的对应关系。

\*

\* 1，因为操作的是字符串中的字母，所以先将字符串变成字符数组。

\* 2，遍历字符数组，用每一个字母作为键去查Map集合这个表。

\* 如果该字母键不存在，就将该字母作为键 1作为值存储到map集合中。

\* 如果该字母键存在，就将该字母键对应值取出并+1，在将该字母和+1后的值存储到map集合中，

\* 键相同值会覆盖。这样就记录住了该字母的次数.

\* 3，遍历结束，map集合就记录所有字母的出现的次数。oy.

\*/

public class MapTest {

/\*\*

\* @param args

\*/

public static void main(String[] args) { String str = "fdg+avAdc bs5dDa9c-dfs"; String s = getCharCount(str); System.out.println(s);

}

public static String getCharCount(String str) {

//将字符串变成字符数组

char[] chs = str.toCharArray();

//定义map集合表。

Map<Character,Integer> map = new TreeMap<Character,Integer>(); for (int i = 0; i < chs.length; i++) {

if(!(chs[i]>='a' && chs[i]<='z' || chs[i]>='A' && chs[i]<='Z'))

// if(!(Character.toLowerCase(chs[i])>='a' && Character.toLowerCase(chs[i])<='z'))

continue;

//将数组中的字母作为键去查map表。 Integer value = map.get(chs[i]);

int count = 1;

//判断值是否为null. if(value!=null){

count = value+1;

}

// count++; map.put(chs[i], count);

/\* if(value==null){

map.put(chs[i], 1);

}else{

map.put(chs[i], value+1);

}

\*/

}

return mapToString(map);

}

private static String mapToString(Map<Character, Integer> map) { StringBuilder sb = new StringBuilder();

Iterator<Character> it = map.keySet().iterator();

while(it.hasNext()){ Character key = it.next();

Integer value = map.get(key);

sb.append(key+"("+value+")");

}

return sb.toString();

}

}

习题2：

package cn.itcast.p1.map.test;

import java.util.HashMap; import java.util.Map;

public class MapTest2 {

/\*\*

\* @param args

\*/

public static void main(String[] args) {

/\*

\* Map在有映射关系时，可以优先考虑。

\*

\* 在查表法中的应用较为多见。

\*/

String week = getWeek(1); System.out.println(week);

System.out.println(getWeekByMap(week));

}

public static String getWeekByMap(String week){ Map<String,String> map = new HashMap<String,String>();

map.put("星期一","Mon");

map.put("星期二","Tus");

map.put("星期三","Wes");

map.put("星期日","Sun");

map.put("星期天","Sun");

return map.get(week);

}

public static String getWeek(int week){

if(week<1 || week>7)

throw new RuntimeException("没有对应的星期，请您重新输入"); String[] weeks = {"","星期一","星期二"};

return weeks[week];

}

}

# 126.Collections集合框架工具类简述及使用：

package cn.itcast.p2.toolclass.collections.demo;

import java.util.ArrayList; import java.util.Collections; import java.util.Comparator; import java.util.List; import java.util.TreeSet;

import cn.itcast.p2.comparator.ComparatorByLength;

public class CollectionsDemo {

public static void main(String[] args) {

/\*

\* Collections：是集合框架的工具类。

\* 里面的方法都是静态的。

\*/

demo\_4();

}

public static void demo\_4() {

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

list.add("abcde");

list.add("cba"); list.add("zhangsan"); list.add("zhaoliu"); list.add("xiaoqiang");

System.out.println(list);

// Collections.replaceAll(list, "cba", "nba"); // set(indexOf("cba"),"nba"); Collections.shuffle(list);

// Collections.fill(list, "cc"); System.out.println(list);

}

public static void demo\_3() {

/\*

TreeSet<String> ts = new TreeSet<String>(new Comparator<String>(){

@Override

public int compare(String o1, String o2) {

int temp = o2.compareTo(o1); return temp;

}

});

\*/

TreeSet<String> ts = new TreeSet<String>(Collections.reverseOrder(new ComparatorByLength()));

ts.add("abc");

ts.add("hahaha");

ts.add("zzz");

ts.add("aa");

ts.add("cba"); System.out.println(ts);

}

public static void demo\_2(){

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

list.add("abcde");

list.add("cba");

list.add("aa");

list.add("zzz");

list.add("cba");

list.add("nbaa");

// Collections.sort(list); System.out.println(list);

// int index = Collections.binarySearch(list, "cba");

// System.out.println("index="+index);

//获取最大值。

String max = Collections.max(list,new ComparatorByLength()); System.out.println("max="+max);

}

public static void demo\_1(){

List<String> list = new ArrayList<String>(); list.add("abcde");

list.add("cba");

list.add("aa");

list.add("zzz");

list.add("cba");

list.add("nbaa"); System.out.println(list);

//对list集合进行指定顺序的排序。

// Collections.sort(list);

// mySort(list);

// mySort(list,new ComparatorByLength()); Collections.sort(list,new ComparatorByLength()); System.out.println(list);

}

public static <T> void mySort(List<T> list,Comparator<? super T> comp){

for (int i = 0; i < list.size()-1; i++) { for (int j = i+1; j < list.size(); j++) {

if(comp.compare(list.get(i), list.get(j))>0){

// T temp = list.get(i);

// list.set(i, list.get(j));

// list.set(j, temp); Collections.swap(list, i, j);

}

}

}

}

public static <T extends Comparable<? super T>> void mySort(List<T> list){ for (int i = 0; i < list.size()-1; i++) {

for (int j = i+1; j < list.size(); j++) { if(list.get(i).compareTo(list.get(j))>0){

// T temp = list.get(i);

// list.set(i, list.get(j));

// list.set(j, temp); Collections.swap(list, i, j);

}

}

}

}

}

# 127.Arrays数组工具类简述及使用：

package cn.itcast.p3.toolclass.arrays.demo;

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

//数组转成集合。

public class ArraysDemo {

/\*\*

\* @param args

\*/

public static void main(String[] args) {

/\*

\* Arrays：集合框架的工具类。里面的方法都是静态的。

\*/

// int[] arr = {3,1,5,6,3,6};

// System.out.println(Arrays.toString(arr));

demo\_2();

}

public static void demo\_2() {

/\*

\* 如果数组中的元素是对象，那么转成集合时，直接将数组中的元素作为集合中的元素进行集合存储。

\*

\* 如果数组中的元素是基本类型数值，那么会将该数组作为集合中的元素进行存储。

\*/

int[] arr = {31,11,51,61};

List<int[]> list = Arrays.asList(arr);

System.out.println(list);

}

public static void demo\_1() {

/\*

\* 重点：List asList(数组)将数组转成集合。

\*

\* 好处：其实可以使用集合的方法操作数组中的元素。

\* 注意：数组的长度是固定的，所以对于集合的增删方法是不可以使用的

\* 否则会发生UnsupportedOperationException

\*/

String[] arr = {"abc","haha","xixi"};

boolean b = myContains(arr, "xixi"); System.out.println("contains:"+b);

List<String> list = Arrays.asList(arr); boolean b1 = list.contains("xixi"); System.out.println("list contaisn:="+b1);

// list.add("hiahia");//UnsupportedOperationException

System.out.println(list);

}

public static boolean myContains(String[] arr,String key){ for (int i = 0; i < arr.length; i++) {

if(arr[i].equals(key)) return true;

}

return false;

}

//toString的经典实现。

public static String myToString(int[] a){ int iMax = a.length - 1;

if (iMax == -1)

return "[]";

StringBuilder b = new StringBuilder(); b.append('[');

for (int i = 0; ; i++) {//中间省略条件判断，提高了效率。

b.append(a[i]); if (i == iMax)

return b.append(']').toString(); b.append(", ");

}

}

}

# 128.集合转数组的toArray（）方法使用：

package cn.itcast.p3.toolclass.arrays.demo;

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

public class ToArray {

public static void main(String[] args) {

/\*

\* 集合转成数组呢？

\*

\* 使用的就是Collection接口中的toArray方法。

\*

\* 集合转成数组：可以对集合中的元素操作的方法进行限定，但不允许对其进行增删。

\*/

List<String> list = new ArrayList<String>(); list.add("abc1");

list.add("abc2");

list.add("abc3");

/\*

\* toArray方法需要传入一个指定类型的数组。

\* 长度该如何定义呢？

\* 如果长度小于集合的size，那么该方法会创建一个同类型并和集合相同size的数组。

\* 如果长度大于集合的size，那么该方法就会使用指定的数组，存储集合中的元素，其他位置默认为

\* null。

\* 所以建议，最后长度就指定为，集合的size。

\*/

String[] arr = list.toArray(new String[list.size()]); System.out.println(Arrays.toString(arr));

}

}

129.增强for循环使用：

package cn.itcast.p4.news.demo;

import java.util.ArrayList; import java.util.HashMap; import java.util.List; import java.util.Map;

public class ForEachDemo {

/\*\*

\* @param args

\*/

public static void main(String[] args) {

/\*

* foreach语句：

\* 格式：

* for(类型 变量 ：Collection集合|数组)

\* {

\*

\* }

\*

\* 传统for和高级for的区别？

\* 传统for可以完成对语句执行很多次，因为可以定义控制循环的增量和条件。

\*

\* 高级for是一种简化形式。

\* 它必须有被遍历的目标。该目标要是数组，要么是Collection单列集合。

\*

\* 对数数组的遍历如果仅仅是获取数组中的元素，可以使用高级for。

\* 如果要对数组的角标进行操作建议使用传统for。

\*

\*/

List<String> list =new ArrayList<String>(); list.add("abc1");

list.add("abc2");

list.add("abc3");

for(String s : list){ //简化书写。 System.out.println(s);

}

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

for(int i : arr){ System.out.println(i);

}

//可以使用高级for遍历map集合吗？不能直接用，但是可以将map转成单列的set，就可以用了。 Map<Integer,String> map = new HashMap<Integer,String>();

map.put(3,"zhagsan");

map.put(1,"wangyi");

map.put(7,"wagnwu"); map.put(4,"zhagsansan");

for(Integer key : map.keySet()){ String value = map.get(key); System.out.println(key+"::"+value);

}

for(Map.Entry<Integer,String> me : map.entrySet()){ Integer key = me.getKey();

String value = me.getValue();

System.out.println(key+":"+value);

}

// Iterator<String> it = list.iterator();

// while(it.hasNext()){

// System.out.println(it.next());

// }

}

}

130.凼数的可发参数：

**package** cn.itcast.p4.news.demo;

**public class** ParamterDemo {

/\*\*

\* **@param** args

\*/

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

// int sum = add(4,5);

// System.out.println("sum="+sum);

// int sum1 = add(4,5,6);

// System.out.println("sum1="+sum1);

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

// int sum = add(arr);

// System.out.println("sum="+sum);

// int[] arr1 = {5,1,4,7,3,9,8,7,6};

// int sum1 = add(arr1);

// System.out.println("sum1="+sum1);

**int** sum = *newAdd*(5,1,4,7,3); System.*out*.println("sum="+sum);

**int** sum1 = *newAdd*(5,1,2,7,3,9,8,7,6); System.*out*.println("sum1="+sum1);

}

/\*

\* 函数的可变参数。

\* 其实就是一个数组，但是接收的是数组的元素。

\* 自动将这些元素封装成数组。简化了调用者的书写。

\* 注意：可变参数类型，必须定义在参数列表的结尾。

\*/

**public static int** newAdd(**int** a,**int**... arr){

**int** sum = 0;

**for** (**int** i = 0; i < arr.length; i++) { sum+=arr[i];

}

**return** sum;

// System.out.println(arr);

}

**public static int** add(**int**[] arr){

**int** sum = 0;

**for** (**int** i = 0; i < arr.length; i++) { sum+=arr[i];

}

**return** sum;

}

**public static int** add(**int** a,**int** b){

**return** a+b;

}

**public static int** add(**int** a,**int** b,**int** c){

**return** a+b+c;

}

}

# 131.静态导入，其实导入的是类中的静态成员（演示）：

package cn.itcast.p4.news.demo;

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

import static java.util.Collections.\*;//静态导入，其实到入的是类中的静态成员。

//import static java.util.Collections.max;//静态导入，其实到入的是类中的静态成员。 import static java.lang.System.\*;

public class StaticImportDemo {

/\*\*

\* @param args

\*/

public static void main(String[] args) { List<String> list = new ArrayList<String>();

list.add("abc3");

list.add("abc7");

list.add("abc1");

out.println(list);

sort(list);

System.out.println(list); String max = max(list); System.out.println("max="+max);

}

}

# 132.集合（List、Set、Map）大总结：

集合类的由来： 对象用于封装特有数据，对象多了需要存储，如果对象的个数不确定。 就使用集合容器进行存储。

集合特点：

1，用于存储对象的容器。

2，集合的长度是可变的。

3，集合中不可以存储基本数据类型值。

集合容器因为内部的数据结构不同，有多种具体容器。 不断的向上抽取，就形成了集合框架。

框架的顶层Collection接口：

Collection的常见方法： 1，添加。

boolean add(Object obj): boolean addAll(Collection coll):

2，删除。

boolean remove(object obj): boolean removeAll(Collection coll); void clear();

3，判断：

boolean contains(object obj):

boolean containsAll(Colllection coll); boolean isEmpty():判断集合中是否有元素。

4，获取：

int size():

Iterator iterator():取出元素的方式：迭代器。 该对象必须依赖于具体容器，因为每一个容器的数据结构都不同。 所以该迭代器对象是在容器中进行内部实现的。

也就是iterator方法。

Iterator接口就是对所有的Collection容器进行元素取出的公共接口。 其实就是抓娃娃游戏机中的夹子！

5，其他：

boolean retainAll(Collection coll);取交集。 Object[] toArray():将集合转成数组。

--------------------------------------------------------------------------------

Collection

|--List：有序(存入和取出的顺序一致),元素都有索引(角标)，元素可以重复。

|--Set：元素不能重复,无序。

List:特有的常见方法：有一个共性特点就是都可以操作角标。 1，添加

void add(index,element); void add(index,collection);

2，删除；

Object remove(index):

3，修改：

Object set(index,element);

4，获取：

Object get(index); int indexOf(object);

int lastIndexOf(object); List subList(from,to);

list集合是可以完成对元素的增删改查。 List:

|--Vector:内部是数组数据结构，是同步的。增删，查询都很慢！

|--ArrayList:内部是数组数据结构，是不同步的。替代了Vector。查询的速度快。

|--LinkedList:内部是链表数据结构，是不同步的。增删元素的速度很快。 LinkedList:

addFirst(); addLast(): jdk1.6

offerFirst(); offetLast();

getFirst();.//获取但不移除，如果链表为空，抛出NoSuchElementException. getLast();

jdk1.6 peekFirst();//获取但不移除，如果链表为空，返回null. peekLast():

removeFirst();//获取并移除，如果链表为空，抛出NoSuchElementException. removeLast();

jdk1.6 pollFirst();//获取并移除，如果链表为空，返回null. pollLast();

作业：

1，自己去查文档演示Vector中的elements()方法。

2，LinkedList中的，addFirst addLast getFirst，getLast removeFirst removeLast。 3，既然集合是存储对象的，请定义ArryaList集合，并存储Person对象。如new Person("lisi",20);

并取出。将姓名和年龄打印出来。

--------------------------------------------------------------------------------

Set:元素不可以重复，是无序。 Set接口中的方法和Collection一致。

|--HashSet: 内部数据结构是哈希表 ，是不同步的。 如何保证该集合的元素唯一性呢？ 是通过对象的hashCode和equals方法来完成对象唯一性的。 如果对象的hashCode值不同，那么不用判断equals方法，就直接存储到哈希表中。 如果对象的hashCode值相同，那么要再次判断对象的equals方法是否为true。

如果为true，视为相同元素，不存。如果为false，那么视为不同元素，就进行存储。

记住：如果元素要存储到HashSet集合中，必须覆盖hashCode方法和equals方法。 一般情况下，如果定义的类会产生很多对象，比如人，学生，书，通常都需要覆盖equals，hashCode 方法。

建立对象判断是否相同的依据。

|--TreeSet:可以对Set集合中的元素进行排序。是不同步的。 判断元素唯一性的方式：就是根据比较方法的返回结果是否是0，是0，就是相同元素，不存。

TreeSet对元素进行排序的方式一： 让元素自身具备比较功能，元就需要实现Comparable接口。覆盖compareTo方法。

如果不要按照对象中具备的自然顺序进行排序。如果对象中不具备自然顺序。怎么办？ 可以使用TreeSet集合第二种排序方式二：

让集合自身具备比较功能，定义一个类实现Comparator接口，覆盖compare方法。 将该类对象作为参数传递给TreeSet集合的构造函数。

if(this.hashCode()== obj.hashCode() && this.equals(obj))

哈希表确定元素是否相同

1，判断的是两个元素的哈希值是否相同。 如果相同，在判断两个对象的内容是否相同。

2，判断哈希值相同，其实判断的是对象的hashCode的方法。判断内容相同，用的是equals方法。 注意：如果哈希值不同，是不需要判断equals。

--------------------------------------------------------------------------------

泛型：

jdk1.5出现的安全机制。

好处：

1，将运行时期的问题ClassCastException转到了编译时期。 2，避免了强制转换的麻烦。

<>:什么时候用？当操作的引用数据类型不确定的时候。就使用<>。将要操作的引用数据类型传入即可.

其实<>就是一个用于接收具体引用数据类型的参数范围。 在程序中，只要用到了带有<>的类或者接口，就要明确传入的具体引用数据类型 。 泛型技术是给编译器使用的技术,用于编译时期。确保了类型的安全。

运行时，会将泛型去掉，生成的class文件中是不带泛型的,这个称为泛型的擦除。 为什么擦除呢？因为为了兼容运行的类加载器。

泛型的补偿：在运行时，通过获取元素的类型进行转换动作。不用使用者在强制转换了。 泛型的通配符：? 未知类型。

泛型的限定：

? extends E: 接收E类型或者E的子类型对象。上限 一般存储对象的时候用。比如 添加元素 addAll.

? super E: 接收E类型或者E的父类型对象。 下限。 一般取出对象的时候用。比如比较器。

--------------------------------------------------------------------------------

集合的一些技巧：

需要唯一吗？ 需要：Set

需要制定顺序：

需要： TreeSet

不需要：HashSet 但是想要一个和存储一致的顺序(有序):LinkedHashSet

不需要：List

需要频繁增删吗？ 需要：LinkedList 不需要：ArrayList

如何记录每一个容器的结构和所属体系呢？ 看名字！

List

|--ArrayList

|--LinkedList

Set

|--HashSet

|--TreeSet

后缀名就是该集合所属的体系。

前缀名就是该集合的数据结构。 看到array：就要想到数组，就要想到查询快，有角标.

看到link：就要想到链表，就要想到增删快，就要想要 add get remove+frist last的方法

看到hash:就要想到哈希表，就要想到唯一性，就要想到元素需要覆盖hashcode方法和equals方法。 看到tree：就要想到二叉树，就要想要排序，就要想到两个接口Comparable，Comparator 。 而且通常这些常用的集合容器都是不同步的。

--------------------------------------------------------------------------------

Map：一次添加一对元素。Collection 一次添加一个元素。 Map也称为双列集合，Collection集合称为单列集合。 其实map集合中存储的就是键值对。 map集合中必须保证键的唯一性。

常用方法：

1，添加。

value put(key,value):返回前一个和key关联的值，如果没有返回null.

2，删除。

void clear():清空map集合。

value remove(key):根据指定的key翻出这个键值对。

3，判断。

boolean containsKey(key): boolean containsValue(value): boolean isEmpty();

4，获取。

value get(key):通过键获取值，如果没有该键返回null。 当然可以通过返回null，来判断是否包含指定键。

int size(): 获取键值对的个数。

Map常用的子类：

|--Hashtable :内部结构是哈希表，是同步的。不允许null作为键，null作为值。

|--Properties：用来存储键值对型的配置文件的信息，可以和IO技术相结合。

|--HashMap : 内部结构是哈希表，不是同步的。允许null作为键，null作为值。

|--TreeMap : 内部结构是二叉树，不是同步的。可以对Map集合中的键进行排序。

List list = new ArrayList();//非同步的。

list = MyCollections.synList(list);//返回一个同步的list.

给非同步的集合加锁：

class MyCollections{

public static List synList(List list){

return new MyList(list);

}

private class MyList implements List{ private List list;

private static final Object lock = new Object(); MyList(List list){

this.list = list;

}

public boolean add(Object obj){ synchronized(lock)

{

return list.add(obj);

}

}

public boolean remove(Object obj){ synchronized(lock)

{

return list.remove(obj);

}

}

}

}

133.集合框架查补乊遍历集合：

//遍历List方法1，使用普通for循环：

**for** (**int** i = 0; i < list.size(); i++) { String temp=(String)list.get(i); System.*out*.println(temp);

//list.remove(i);//遍历删除元素，不过不推荐这种方式！

}

//遍历List方法2，使用增强for循环（应该使用泛型定义类型！）：

**for**(String temp:list){ System.*out*.println(temp);

}

//遍历List方法3，使用Iterator迭代器：

**for**(Iterator iter=list.iterator();iter.hasNext();){ String temp=(String)iter.next(); System.*out*.println(temp);

或者：

}

Iterator iter=c.iterator();

**while**(iter.hasNext()){ Object obj=iter.next();

iter.remove();//如果要遍历删除集合中的元素，建议使用这种方式！

System.*out*.println(obj);

}

//遍历Set方法1，使用增强for循环： for(String temp:set){

System.out.println(temp);

}

//遍历Set方法2，使用Iterator迭代器：

**for**(Iterator iter=list.iterator();iter.hasNext();){ String temp=(String)iter.next(); System.*out*.println(temp);

}

//遍历Map

Map<Integer, Man> maps=**new** HashMap<Integer, Man>(); Set<Integer> keySet=maps.keySet();

**for**(Integer id:keySet){ System.*out*.println(maps.get(id).name);

}

134.其他类对象-日期类对象的使用：

示例1：

**package** cn.itcast.p1.otherapi; **import** java.util.Calendar; **public class** CalendarDemo {

**public static void** main(String[] args) { Calendar c = Calendar.*getInstance*();

**int** year = 2012;

*showDays*(year);

}

**public static void** showDays(**int** year) {

Calendar c = Calendar.*getInstance*(); c.set(year, 2, 1);

c.add(Calendar.*DAY\_OF\_MONTH*, -1);

*showDate*(c);

}

**public static void** showDate(Calendar c) {

**int** year = c.get(Calendar.*YEAR*);

**int** month = c.get(Calendar.*MONTH*)+1; **int** day = c.get(Calendar.*DAY\_OF\_MONTH*); **int** week = c.get(Calendar.*DAY\_OF\_WEEK*);

System.*out*.println(year+"年"+month+"月"+day+"日"+*getWeek*(week));

}

**public static** String getWeek(**int** i) {

"};

String[] weeks = {"","星期日","星期一","星期二","星期三","星期四","星期五","星期六

**return** weeks[i];

}

}

示例2：

package cn.itcast.p1.otherapi;

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

public class DateDemo {

/\*\*

* @param args
* @throws ParseException

\*/

public static void main(String[] args) throws ParseException { methodDemo\_3();

}

/\*\*

\* 将日期格式的字符串-->日期对象。

* 使用的是DateFormat类中的parse()方法。

\*

* @throws ParseException

\*/

public static void methodDemo\_3() throws ParseException {

String str\_date = "2012年4月19日"; str\_date = "2011---8---17";

DateFormat dateFormat = DateFormat.getDateInstance(DateFormat.LONG); dateFormat = new SimpleDateFormat("yyyy---MM---dd");

Date date = dateFormat.parse(str\_date); System.out.println(date);

}

/\*\*

\* 对日期对象进行格式化。

\* 将日期对象-->日期格式的字符串。

\* 使用的是DateFormat类中的format方法。

\*

\*/

public static void methodDemo\_2() { Date date = new Date();

//获取日期格式对象。具体着默认的风格。 FULL LONG等可以指定风格。

DateFormat dateFormat = DateFormat.getDateInstance(DateFormat.LONG); dateFormat =

DateFormat.getDateTimeInstance(DateFormat.LONG,DateFormat.LONG);

// System.out.println(dateFormat);

//如果风格是自定义的如何解决呢？

dateFormat = new SimpleDateFormat("yyyy--MM--dd"); String str\_date = dateFormat.format(date);

System.out.println(str\_date);

}

/\*\*

\* 日期对象和毫秒值之间的转换。

\*

\* 毫秒值-->日期对象 ：

* 1，通过Date对象的构造方法 new Date(timeMillis);

\* 2，还可以通过setTime设置。

\* 因为可以通过Date对象的方法对该日期中的各个字段(年月日等)进行操作。

\*

\* 日期对象-->毫秒值：

* 2，getTime方法。

\* 因为可以通过具体的数值进行运算。

\*/

public static void methodDemo\_1() {

long time = System.currentTimeMillis();//

// System.out.println(time);//1335671230671

Date date = new Date();//将当前日期和时间封装成Date对象。 System.out.println(date);//Sun Apr 29 11:48:02 CST 2012

Date date2 = new Date(1335664696656l);//将指定毫秒值封装成Date对象。 System.out.println(date2);

}

}

# 135.其他类对象-Math类对象的使用：

示例1：

**package** cn.itcast.p1.otherapi;

**public class** Mat {

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

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

// int d = (int) (Math.random() \* 10+1);

// int d=(int) Math.ceil(12.23);

// double d = Math.ceil(12.31);

// Double d=Math.floor(12.36);

// long d=Math.round(63.50);

**double** d=Math.*pow*(2, 3); System.*out*.println(d);

}

}

示例2：

**package** cn.itcast.p1.otherapi; **import** java.util.Random; **public class** MathDemo {

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

/\*

\* Math:提供了操作数学运算的方法，都是静态的。

\* 常用的方法：

\* ceil():返回大于参数的最小整数。

\* floor():返回小于参数的最大整数。

\* round():返回四舍五入的整数。

\* pow(a,b):a的b次方。

\*/

**double** d1 = Math.*ceil*(12.56); **double** d2 = Math.*floor*(12.56); **double** d3 = Math.*round*(12.46);

// sop("d1="+d1);

// sop("d2="+d2);

// sop("d3="+d3);

// double d = Math.pow(10, 2);

// sop("d="+d);

Random r = **new** Random();

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

// double d = Math.ceil(Math.random()\*10);

// double d = (int)(Math.random()\*6+1);

// double d = (int)(r.nextDouble()\*6+1); **int** d = r.nextInt(6)+1; System.*out*.println(d);

}

}

**public static void** sop(String string) { System.*out*.println(string);

}

}

# 136.其他类对象-Runtime类对象的使用：

**package** cn.itcast.p1.otherapi; **import** java.io.IOException; **public class** RuntimeDemo {

/\*\*

* **@param** args
* **@throws** IOException
* **@throws** InterruptedException

\*/

**public static void** main(String[] args) **throws** IOException, InterruptedException

{

/\*

\* Runtime:没有构造方法摘要，说明该类不可以创建对象。

\* 又发现还有非静态的方法。说明该类应该提供静态的返回该类对象的方法。

\* 而且只有一个，说明Runtime类使用了单例设计模式。

\*

\*/

Runtime r = Runtime.*getRuntime*();

// execute: 执行。 xxx.exe

Process p = r.exec("notepad.exe"); Thread.*sleep*(5000);

p.destroy();

}

}

# 137.其他类对象-System类对象的使用：

package cn.itcast.p1.otherapi;

import java.util.Arrays; import java.util.List; import java.util.Properties; import java.util.Set;

public class SystemDemo {

private static final String LINE\_SEPARATOR = System.getProperty("line.separator");

/\*\*

\* @param args

\*/

public static void main(String[] args) {

/\*

\* System:类中的方法和属性都是静态的。

\* 常见方法：

\* long currentTimeMillis();获取当前时间的毫秒值。

\*/

|  |  |  |
| --- | --- | --- |
| // |  | long l1 = 1335664696656l;//System.currentTimeMillis(); |
| // |  | System.out.println(l1/1000/60/60/24);//1335664696656 |
| // |  | code.. |
| // |  | long l2 = System.currentTimeMillis(); |
| // |  | System.out.println(l2-l1); |
|  |  | System.out.println("hello-"+LINE\_SEPARATOR+" world"); |
| // |  | demo\_1(); |
| // |  | //给系统设置一些属性信息。这些信息是全局，其他程序都可以使用。  System.setProperty("myclasspath", "c:\myclass"); |
|  | } |  |

public static void demo\_1(){

//获取系统的属性信息，并存储到了Properties集合中。

/\*

\* properties集合中存储都是String类型的键和值。

\* 最好使用它自己的存储和取出的方法来完成元素的操作。

\*/

Properties prop = System.getProperties(); Set<String> nameSet = prop.stringPropertyNames();

for(String name : nameSet){

String value = prop.getProperty(name);

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

}

}

}

138.时间对象Date练习：

package cn.itcast.p1.otherapi.test;

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

/\* 练习：

\* "2012-3-17"到"2012-4-6"

\* 中间有多少天？

\* 思路：

\* 两个日期相减就哦了。

\* 咋减呢？

\* 必须要有两个可以进行减法运算的数。

\* 能减可以是毫秒值。如何获取毫秒值？通过date对象。

\* 如何获取date对象呢？可以将字符串转成date对象。

\*

\* 1,将日期格式的字符串转成Date对象。

\* 2,将Date对象转成毫秒值。

\* 3，相减，在变成天数

\*

\*/

public class DateTest {

/\*\*

* @param args
* @throws ParseException

\*/

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

String str\_date1 = "2012-3-17"; String str\_date2 = "2012-4-18";

test(str\_date1,str\_date2);

}

public static void test(String str\_date1,String str\_date2) throws ParseException

{

//1,将日期字符串转成日期对象。

//定义日期格式对象。

DateFormat dateFormat = DateFormat.getDateInstance(); dateFormat = new SimpleDateFormat("yyyy-MM-dd");

Date date1 = dateFormat.parse(str\_date1); Date date2 = dateFormat.parse(str\_date2);

long time1 = date1.getTime(); long time2 = date2.getTime();

long time = Math.abs(time1-time2);

int day = getDay(time); System.out.println(day);

}

private static int getDay(long time) {

int day = (int)(time/1000/60/60/24); return day;

}

}

# 第八章：IO流

139.流的简述及演示案例：

输入流和输出流相对于内存设备而言.

将外设中的数据读取到内存中:输入 将内存的数写入到外设中：输出。

字符流的由来： 其实就是：字节流读取文字字节数据后，不直接操作而是先查指定的编码表。获取对应的文字。 在对这个文字进行操作。简单说：字节流+编码表

-------------------------------------------------------------------------------

字节流的两个顶层父类：

1，InputStream 2，OutputStream.

字符流的两个顶层父类：

1，Reader 2，Writer

这些体系的子类都以父类名作为后缀。 而且子类名的前缀就是该对象的功能。

FileWriter演示：

package cn.itcast.p2.io.filewriter;

import java.io.FileWriter; import java.io.IOException;

//需求：将一些文字存储到硬盘一个文件中。 public class FileWriterDemo {

private static final String LINE\_SEPARATOR = System.getProperty("line.separator");

/\*\*

* @param args
* @throws IOException

\*/

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

//创建一个可以往文件中写入字符数据的字符输出流对象。

/\*

\* 既然是往一个文件中写入文字数据，那么在创建对象时，就必须明确该文件(用于存储数据的目的

\* 地)。

\*

\* 如果文件不存在，则会自动创建。

\* 如果文件存在，则会被覆盖。

\*

\* 如果构造函数中加入true，可以实现对文件进行续写！

\*/

FileWriter fw = new FileWriter("demo.txt",true);

/\*

\* 调用Writer对象中的write(string)方法，写入数据。

\*

\* 其实数据写入到临时存储缓冲区中。

\*

\*/ fw.write("abcde"+LINE\_SEPARATOR+"hahaha");

// fw.write("xixi");

/\*

\* 进行刷新，将数据直接写到目的地中。

\*/

// fw.flush();

/\*

\* 关闭流，关闭资源。在关闭前会先调用flush刷新缓冲中的数据到目的地。

\*/ fw.close();

// fw.write("haha");// java.io.IOException: Stream closed

}

}

IOException演示：

package cn.itcast.p2.io.filewriter;

import java.io.FileWriter; import java.io.IOException;

public class IOExceptionDemo {

private static final String LINE\_SEPARATOR = System

.getProperty("line.separator");

/\*\*

* @param args
* @throws IOException

\*/

public static void main(String[] args) {

FileWriter fw = null; try {

fw = new FileWriter("k:\\demo.txt"); fw.write("abcde" + LINE\_SEPARATOR + "hahaha");

} catch (IOException e) { System.out.println(e.toString());

} finally {

if (fw != null) try {

fw.close();

} catch (IOException e) {

// code....

throw new RuntimeException("关闭失败");

}

}

}

}

FileReader演示1：

package cn.itcast.p3.io.filereader;

import java.io.FileNotFoundException; import java.io.FileReader;

import java.io.IOException;

//需求：读取一个文本文件。将读取到的字符打印到控制台. public class FileReaderDemo {

/\*\*

* @param args
* @throws IOException

\*/

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

//1,创建读取字符数据的流对象。

/\*

\* 在创建读取流对象时，必须要明确被读取的文件。一定要确定该文件是存在的。

\*

\* 用一个读取流关联一个已存在文件。

\*/

FileReader fr = new FileReader("demo.txt"); int ch = 0;

while((ch=fr.read())!=-1){ System.out.println((char)ch);

}

/\*

//用Reader中的read方法读取字符。 int ch = fr.read(); System.out.println((char)ch); int ch1 = fr.read();

System.out.println(ch1); int ch2 = fr.read(); System.out.println(ch2);

\*/ fr.close();

}

}

FileReader演示2：

package cn.itcast.p3.io.filereader;

import java.io.FileReader; import java.io.IOException;

//需求：读取一个文本文件。将读取到的字符打印到控制台. public class FileReaderDemo2 {

/\*\*

* @param args
* @throws IOException

\*/

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

FileReader fr = new FileReader("demo.txt");

/\*

\* 使用read(char[])读取文本文件数据。

\*

\* 先创建字符数组。

\*/

char[] buf = new char[1024]; int len = 0;

while((len=fr.read(buf))!=-1){ System.out.println(new String(buf,0,len));

}

/\*

int num = fr.read(buf);//将读取到的字符存储到数组中。 System.out.println(num+":"+new String(buf,0,num)); int num1 = fr.read(buf);//将读取到的字符存储到数组中。 System.out.println(num1+":"+new String(buf,0,num1)); int num2 = fr.read(buf);//将读取到的字符存储到数组中。 System.out.println(num2+":"+new String(buf));

\*/ fr.close();

}

}

# 140.字符流（Reader、Writer）基本使用：

案例1：

package cn.itcast.p1.io.charstream.test;

import java.io.FileReader; import java.io.FileWriter;

import java.io.IOException;

/\*

\* 需求：将盘中的一个文件进行复制。

\*

\* 思路：

\* 1，需要读取源，

\* 2，将读到的源数据写入到目的地。

\* 3，既然是操作文本数据，使用字符流。

\*

\*/

public class CopyTextTest {

/\*\*

* @param args
* @throws IOException

\*/

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

//1,读取一个已有的文本文件，使用字符读取流和文件相关联。 FileReader fr = new FileReader("IO流\_2.txt");

//2,创建一个目的，用于存储读到数据。

FileWriter fw = new FileWriter("copytext\_1.txt");

//3,频繁的读写操作。 int ch = 0;

while((ch=fr.read())!=-1){ fw.write(ch);

}

//4,关闭流资源。

fw.close();

fr.close();

}

}

案例2：

package cn.itcast.p1.io.charstream.test;

import java.io.FileReader; import java.io.FileWriter; import java.io.IOException;

public class CopyTextTest\_2 {

private static final int BUFFER\_SIZE = 1024;

/\*\*

\* @param args

\*/

public static void main(String[] args) {

FileReader fr = null; FileWriter fw = null; try {

fr = new FileReader("IO流\_2.txt");

fw = new FileWriter("copytest\_2.txt");

//创建一个临时容器，用于缓存读取到的字符。

char[] buf = new char[BUFFER\_SIZE];//这就是缓冲区。

//定义一个变量记录读取到的字符数，(其实就是往数组里装的字符个数 ) int len = 0;

while((len=fr.read(buf))!=-1){ fw.write(buf, 0, len);

}

} catch (Exception e) {

// System.out.println("读写失败");

throw new RuntimeException("读写失败");

}finally{

if(fw!=null) try {

fw.close();

} catch (IOException e) {

e.printStackTrace();

}

if(fr!=null) try {

fr.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

}

# 141.BufferedReader演示：

package cn.itcast.p2.io.charstream.buffer;

import java.io.BufferedReader; import java.io.FileNotFoundException; import java.io.FileReader;

import java.io.IOException; public class BufferedReaderDemo {

/\*\*

* @param args
* @throws IOException

\*/

public static void main(String[] args) throws IOException { FileReader fr = new FileReader("buf.txt"); BufferedReader bufr = new BufferedReader(fr);

String line = null;

while((line=bufr.readLine())!=null){ System.out.println(line);

}

/\*

String line1 = bufr.readLine(); System.out.println(line1); String line2 = bufr.readLine(); System.out.println(line2); String line3 = bufr.readLine(); System.out.println(line3); String line4 = bufr.readLine(); System.out.println(line4); String line5 = bufr.readLine(); System.out.println(line5);

\*/ bufr.close();

}

/\*\*

* @throws FileNotFoundException
* @throws IOException

\*/

public static void demo() throws FileNotFoundException, IOException { FileReader fr = new FileReader("buf.txt");

char[] buf = new char[1024];

int len = 0; while((len=fr.read(buf))!=-1){

System.out.println(new String(buf,0,len));

}

fr.close();

}

}

# 142.BufferedWriter演示：

package cn.itcast.p2.io.charstream.buffer;

import java.io.BufferedWriter; import java.io.FileWriter; import java.io.IOException;

public class BufferedWriterDemo {

private static final String LINE\_SEPARATOR = System.getProperty("line.separator");

/\*\*

* @param args
* @throws IOException

\*/

public static void main(String[] args) throws IOException { FileWriter fw = new FileWriter("buf.txt");

//为了提高写入的效率。使用了字符流的缓冲区。

//创建了一个字符写入流的缓冲区对象，并和指定要被缓冲的流对象相关联 BufferedWriter bufw = new BufferedWriter(fw);

//使用缓冲区的写入方法将数据先写入到缓冲区中。

// bufw.write("abcdefq"+LINE\_SEPARATOR+"hahahha");

// bufw.write("xixiixii");

// bufw.newLine();

// bufw.write("heheheheh");

for(int x=1; x<=4; x++){ bufw.write("abcdef"+x); bufw.newLine(); bufw.flush();

}

//使用缓冲区的刷新方法将数据刷目的地中。

// bufw.flush();

//关闭缓冲区。其实关闭的就是被缓冲的流对象。 bufw.close();

// fw.write("hehe");

// fw.close();

}

}

# 143.用缓冲区方法buffer读写文件：

package cn.itcast.p3.io.charstream.buffer.test;

import java.io.BufferedReader; import java.io.BufferedWriter; import java.io.FileNotFoundException; import java.io.FileReader;

import java.io.FileWriter; import java.io.IOException;

public class CopyTextByBufTest {

/\*\*

* @param args
* @throws IOException

\*/

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

FileReader fr = new FileReader("buf.txt"); BufferedReader bufr = new BufferedReader(fr);

FileWriter fw = new FileWriter("buf\_copy.txt"); BufferedWriter bufw = new BufferedWriter(fw);

String line = null; while((line=bufr.readLine())!=null){

bufw.write(line); bufw.newLine(); bufw.flush();

}

/\*

int ch = 0; while((ch=bufr.read())!=-1){

bufw.write(ch);

}

\*/

bufw.close();

bufr.close();

}

}

144.自定义缓冲区（练习）：

package cn.itcast.p4.io.charstream.mybuffer;

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

/\*\*

\* 自定义的读取缓冲区。其实就是模拟一个BufferedReader.

\*

\* 分析：

\* 缓冲区中无非就是封装了一个数组，

\* 并对外提供了更多的方法对数组进行访问。

\* 其实这些方法最终操作的都是数组的角标。

\*

\* 缓冲的原理：

\* 其实就是从源中获取一批数据装进缓冲区中。

\* 在从缓冲区中不断的取出一个一个数据。

\*

\* 在此次取完后，在从源中继续取一批数据进缓冲区。

\* 当源中的数据取光时，用-1作为结束标记。

\*/

public class MyBufferedReader extends Reader { private Reader r;

//定义一个数组作为缓冲区。

private char[] buf = new char[1024];

//定义一个指针用于操作这个数组中的元素。当操作到最后一个元素后，指针应该归零。 private int pos = 0;

//定义一个计数器用于记录缓冲区中的数据个数。 当该数据减到0，就从源中继续获取数据到缓冲区中。

private int count = 0;

MyBufferedReader(Reader r){ this.r = r;

}

/\*\*

\* 该方法从缓冲区中一次取一个字符。

* @return
* @throws IOException

\*/

public int myRead() throws IOException{

if(count==0){

count = r.read(buf); pos = 0;

}

if(count<0)

return -1;

char ch = buf[pos++]; count--;

return ch;

/\*

//1,从源中获取一批数据到缓冲区中。需要先做判断，只有计数器为0时，才需要从源中获取数据。 if(count==0){

count = r.read(buf);

if(count<0)

return -1;

//每次获取数据到缓冲区后，角标归零. pos = 0;

char ch = buf[pos];

pos++; count--;

return ch;

}else if(count>0){ char ch = buf[pos];

pos++; count--;

return ch;

}\*/

}

public String myReadLine() throws IOException{ StringBuilder sb = new StringBuilder();

int ch = 0;

while((ch = myRead())!=-1){

if(ch=='\r') continue; if(ch=='\n')

return sb.toString();

//将从缓冲区中读到的字符，存储到缓存行数据的缓冲区中。 sb.append((char)ch);

}

if(sb.length()!=0) return sb.toString();

return null;

}

public void myClose() throws IOException {

r.close();

}

@Override

public int read(char[] cbuf, int off, int len) throws IOException {

return 0;

}

@Override

public void close() throws IOException {

}

}

145.装饰设计模式及演示：

装饰设计模式：对一组对象的功能进行增强时，就可以使用该模式进行问题的解决。 装饰和继承都能实现一样的特点：进行功能的扩展增强，但是装饰比继承灵活，装饰的特点：装饰类和被装饰类 都必须所属同一个接口或者父类。

演示：

**package** cn.itcast.p5.wrapper;

**public class** PersonDemo {

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

Person p = **new** Person();

// p.chifan();

NewPerson p1 = **new** NewPerson(p); p1.chifan();

NewPerson2 p2 = **new** NewPerson2(); p2.chifan();

}

}

**class** Person{

**void** chifan(){ System.*out*.println("吃饭");

}

}

//这个类的出现是为了增强Person而出现的。

**class** NewPerson{ **private** Person p ; NewPerson(Person p){

**this**.p = p;

}

**public void** chifan(){ System.*out*.println("开胃酒"); p.chifan(); System.*out*.println("甜点");

}

}

**class** NewPerson2 **extends** Person{

**public void** chifan(){

System.*out*.println("开胃酒"); **super**.chifan(); System.*out*.println("甜点");

}

}

# 146.LineNumberReader演示：

package cn.itcast.p6.io.charstream.linenumber;

import java.io.FileReader; import java.io.IOException; import java.io.LineNumberReader;

public class LineNumberReaderDemo {

/\*\*

* @param args
* @throws IOException

\*/

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

FileReader fr = new FileReader("IO流\_2.txt"); LineNumberReader lnr = new LineNumberReader(fr);

String line = null; lnr.setLineNumber(100); while((line=lnr.readLine())!=null){

System.out.println(lnr.getLineNumber()+":"+line);

}

lnr.close();

}

}

# 147.字节流（InputStream、OutputStream）使用演

示：

package cn.itcast.p7.io.bytestream.demo;

import java.io.FileInputStream; import java.io.FileNotFoundException; import java.io.FileOutputStream; import java.io.IOException;

public class ByteStreamDemo {

/\*\*

* @param args
* @throws IOException

\*/

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

demo\_read();

}

public static void demo\_read() throws IOException {

//1，创建一个读取流对象。和指定文件关联。

FileInputStream fis = new FileInputStream("bytedemo.txt");

// System.out.println(fis.available());

// byte[] buf = new byte[fis.available()];

// fis.read(buf);

// System.out.println(new String(buf));

//建议使用这种读取数据的方式

// byte[] buf = new byte[1024];

// int len = 0;

//

// while((len=fis.read(buf))!=-1){

// System.out.println(new String(buf,0,len));

// }

// int ch = 0;

// while((ch=fis.read())!=-1){

// System.out.println((char)ch);

// }

//一次读取一个字节。

// int ch = fis.read();

// System.out.println(ch); fis.close();

}

public static void demo\_write() throws IOException {

//1，创建字节输出流对象。用于操作文件.

FileOutputStream fos = new FileOutputStream("bytedemo.txt");

//2,写数据。直接写入到了目的地中。 fos.write("abcdefg".getBytes());

// fos.flush(); fos.close();//关闭资源动作要完成。

}

}

# 148.用字节流复制mp3的几种方法比较：

package cn.itcast.p7.io.bytestream.test;

import java.io.BufferedInputStream; import java.io.BufferedOutputStream; import java.io.FileInputStream; import java.io.FileOutputStream; import java.io.IOException;

public class CopyMp3Test {

/\*\*

* @param args
* @throws IOException

\*/

public static void main(String[] args) throws IOException { copy\_4();

}

// 千万不要用，效率没有！

public static void copy\_4() throws IOException { FileInputStream fis = new FileInputStream("c:\\0.mp3"); FileOutputStream fos = new FileOutputStream("c:\\4.mp3");

int ch = 0;

while((ch =fis.read())!=-1){ fos.write(ch);

}

fos.close();

fis.close();

}

//不建议。

public static void copy\_3() throws IOException { FileInputStream fis = new FileInputStream("c:\\0.mp3"); FileOutputStream fos = new FileOutputStream("c:\\3.mp3");

byte[] buf = new byte[fis.available()]; fis.read(buf);

fos.write(buf); fos.close();

fis.close();

}

public static void copy\_2() throws IOException {

FileInputStream fis = new FileInputStream("c:\\0.mp3"); BufferedInputStream bufis = new BufferedInputStream(fis);

FileOutputStream fos = new FileOutputStream("c:\\2.mp3"); BufferedOutputStream bufos = new BufferedOutputStream(fos);

int ch = 0; while((ch=bufis.read())!=-1){

bufos.write(ch);

}

bufos.close(); bufis.close();

}

public static void copy\_1() throws IOException {

FileInputStream fis = new FileInputStream("c:\\0.mp3"); FileOutputStream fos = new FileOutputStream("c:\\1.mp3");

byte[] buf = new byte[1024];

int len = 0; while((len=fis.read(buf))!=-1){

fos.write(buf,0,len);

}

fos.close();

fis.close();

}

}

149.IO不集合容器结合练习题：

**package** test;

**import** java.util.ArrayList;

/\* 一个ArrayList对象alist中存有若干个字符串元素，现欲遍历该ArrayList对象， 删除其中所有值为"abc"的字符串元素，请问下面的实现正确么？如有问题， 会出现什么问题？如何更正？

。。。

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

if("abc".equals(alist.get(i))) { alist.remove(i);

}

}

\*/

**public class** Test13 {

/\*\*

\* **@param** args

\*/

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

ArrayList<String> al = **new** ArrayList<String>(); al.add("haha");

al.add("abc");

al.add("abc");

al.add("abc");

// int size = al.size();

**for**(**int** x=0; x<al.size(); x++){

**if**("abc".equals(al.get(x))){ al.remove(x);

x--;

}

}

System.*out*.println(al);

}

}

150.读叏键盘弽入数据：

package cn.itcast.io.p1.transstream.demo;

import java.io.IOException; import java.io.InputStream;

/\*

\* 读取一个键盘录入的数据，并打印在控制台上。

\*

\* 键盘本身就是一个标准的输入设备。

\* 对于java而言，对于这种输入设备都有对应的对象。

\*/

public class ReadKey {

/\*\*

* @param args
* @throws IOException

\*/

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

// readKey();

// System.out.println((int)'\r');

// System.out.println((int)'\n'); readKey2();

}

public static void readKey2() throws IOException {

/\*

\* 获取用户键盘录入的数据，

\* 并将数据变成大写显示在控制台上，

\* 如果用户输入的是over，结束键盘录入。

\*

\* 思路：

\* 1，因为键盘录入只读取一个字节，要判断是否是over，需要将读取到的字节拼成字符串。

\* 2，那就需要一个容器。StringBuilder.

\* 3，在用户回车之前将录入的数据变成字符串判断即可。

\*/

//1，创建容器。

StringBuilder sb = new StringBuilder();

//2，获取键盘读取流。

InputStream in = System.in;

//3，定义变量记录读取到的字节，并循环获取。 int ch = 0;

while((ch=in.read())!=-1){

// 在存储之前需要判断是否是换行标记 ,因为换行标记不存储。 if(ch=='\r')

continue; if(ch=='\n'){

String temp = sb.toString(); if("over".equals(temp))

break; System.out.println(temp.toUpperCase()); sb.delete(0, sb.length());

}

else

//将读取到的字节存储到StringBuilder中。 sb.append((char)ch);

// System.out.println(ch);

}

}

public static void readKey() throws IOException { InputStream in = System.in;

int ch = in.read();//阻塞式方法。

System.out.println(ch);

int ch1 = in.read();//阻塞式方法。 System.out.println(ch1);

int ch2 = in.read();//阻塞式方法。

System.out.println(ch2);

// in.close();

// InputStream in2 = System.in;

// int ch3 = in2.read();

}

}

演示1：

package cn.itcast.io.p1.transstream.demo;

import java.io.BufferedReader; import java.io.BufferedWriter; import java.io.IOException; import java.io.InputStream; import java.io.InputStreamReader; import java.io.OutputStream; import java.io.OutputStreamWriter;

public class TransStreamDemo {

/\*\*

* @param args
* @throws IOException

\*/

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

//字节流。

InputStream in = System.in;

// int ch = in.read();

// System.out.println(ch);

// int ch1 = in.read();

// System.out.println(ch1);

//将字节转成字符的桥梁。装换流。

InputStreamReader isr = new InputStreamReader(in);

// int ch = isr.read();

// System.out.println((char)ch);

//字符流。

BufferedReader bufr = new BufferedReader(isr); OutputStream out = System.out;

OutputStreamWriter osw = new OutputStreamWriter(out); BufferedWriter bufw = new BufferedWriter(osw); String line = null;

while((line=bufr.readLine())!=null){ if("over".equals(line))

break;

// System.out.println(line.toUpperCase());

// osw.write(line.toUpperCase()+"\r\n");

// osw.flush();

bufw.write(line.toUpperCase()); bufw.newLine();

bufw.flush();

}

}

}

演示2：

package cn.itcast.io.p1.transstream.demo;

import java.io.BufferedReader; import java.io.BufferedWriter; import java.io.FileOutputStream; import java.io.IOException; import java.io.InputStreamReader; import java.io.OutputStreamWriter;

public class TransStreamDemo2 {

/\*\*

* @param args
* @throws IOException

\*/

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

/\*

\* 1,需求：将键盘录入的数据写入到一个文件中。

\*

\* 2,需求：将一个文本文件内容显示在控制台上。

\*

\* 3,需求：将一个文件文件中的内容复制到的另一个文件中。

\*/

BufferedReader bufr = new BufferedReader(new InputStreamReader(System.in)); BufferedWriter bufw = new BufferedWriter(new OutputStreamWriter(new

FileOutputStream("b.txt")));

String line = null; while((line=bufr.readLine())!=null){

if("over".equals(line)) break;

bufw.write(line.toUpperCase()); bufw.newLine();

bufw.flush();

}

}

}

演示3：

package cn.itcast.io.p1.transstream.demo;

import java.io.FileInputStream; import java.io.FileNotFoundException; import java.io.FileOutputStream; import java.io.FileReader;

import java.io.FileWriter; import java.io.IOException; import java.io.InputStreamReader;

import java.io.OutputStreamWriter;

import java.io.UnsupportedEncodingException; public class TransStreamDemo3 {

/\*\*

* @param args
* @throws IOException

\*/

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

readText\_2();

}

public static void readText\_2() throws IOException, FileNotFoundException {

InputStreamReader isr = new InputStreamReader(new FileInputStream("gbk\_1.txt"),"utf-8");

char[] buf = new char[10]; int len = isr.read(buf);

String str = new String(buf,0,len); System.out.println(str);

isr.close();

}

public static void readText\_1() throws IOException {

FileReader fr = new FileReader("gbk\_1.txt");

char[] buf = new char[10]; int len = fr.read(buf);

String str = new String(buf,0,len); System.out.println(str);

fr.close();

}

public static void writeText\_3() throws IOException {

OutputStreamWriter osw = new OutputStreamWriter(new FileOutputStream("u8\_1.txt"),"UTF-8");

osw.write("你好"); osw.close();

}

public static void writeText\_2() throws IOException {

OutputStreamWriter osw = new OutputStreamWriter(new FileOutputStream("gbk\_3.txt"),"GBK");

// OutputStreamWriter osw = new OutputStreamWriter(new FileOutputStream("gbk\_3.txt"),"GBK");

// FileWriter fw = new FileWriter("gbk\_1.txt");

/\*

\* 这两句代码的功能是等同的。

\* FileWriter：其实就是转换流指定了本机默认码表的体现。而且这个转换流的子类对象，可以方

\* 便操作文本文件。

\* 简单说：操作文件的字节流+本机默认的编码表。

\* 这是按照默认码表来操作文件的便捷类。

\*

\* 如果操作文本文件需要明确具体的编码。FileWriter就不行了。必须用转换流。

\*/ osw.write("你好");

osw.close();

}

public static void writeText\_1() throws IOException {

FileWriter fw = new FileWriter("gbk\_1.txt");

fw.write("你好");

fw.close();

}

}

# 152.File类对象+过滤器的使用演示：

演示1：

**package** cn.itcast.io.p2.file.demo;

**import** java.io.File;

**public class** FileDemo {

/\*\*

\* **@param** args

\*/

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

// constructorDemo();

}

**public static void** constructorDemo() {

//可以将一个已存在的，或者不存在的文件或者目录封装成file对象。 File f1 = **new** File("c:\\a.txt");

File f2 = **new** File("c:\\","a.txt");

File f = **new** File("c:\\"); File f3 = **new** File(f,"a.txt");

File f4 = **new** File("c:"+File.*separator*+"abc"+File.*separator*+"a.txt"); System.*out*.println(f4);

}

}

演示2：

（三种常见过滤器）：

1-后缀名过滤器

package cn.itcast.io.p2.filter;

import java.io.File;

import java.io.FilenameFilter;

public class FilterByJava implements FilenameFilter {

@Override

public boolean accept(File dir, String name) {

// System.out.println(dir+"---"+name);

return name.endsWith(".java");

}

}

2-根据后缀名可以指定任意后缀名过滤的过滤器 package cn.itcast.io.p2.filter;

import java.io.File;

import java.io.FilenameFilter;

public class SuffixFilter implements FilenameFilter { private String suffix ;

public SuffixFilter(String suffix) { super();

this.suffix = suffix;

}

@Override

public boolean accept(File dir, String name) {

return name.endsWith(suffix);

}

}

3-隐藏属性过滤器

package cn.itcast.io.p2.filter;

import java.io.File; import java.io.FileFilter;

public class FilterByHidden implements FileFilter {

@Override

public boolean accept(File pathname) {

return !pathname.isHidden();

}

}

（与过滤器结合使用的演示）：

package cn.itcast.io.p2.file.demo; import java.io.File;

import cn.itcast.io.p2.filter.FilterByHidden; import cn.itcast.io.p2.filter.SuffixFilter;

public class FileListDemo {

/\*\*

\* @param args

\*/

public static void main(String[] args) {

listDemo\_2();

}

public static void listDemo\_3() { File dir = new File("c:\\");

File[] files = dir.listFiles(new FilterByHidden());

for(File file : files){ System.out.println(file);

}

}

public static void listDemo\_2() { File dir = new File("c:\\");

String[] names = dir.list(new SuffixFilter(".txt"));

for(String name : names){ System.out.println(name);

}

}

public static void listDemo() { File file = new File("c:\\");

/\*

\* 获取当前目录下的文件以及文件夹的名称，包含隐藏文件。

\* 调用list方法的File对象中封装的必须是目录。

\* 否则会发生NullPointerException

\* 如果访问的系统级目录也会发生空指针异常。

\*

\* 如果目录存在但是没有内容，会返回一个数组，但是长度为0.

\*

\*/

String[] names = file.list(); System.out.println(names.length);

for(String name : names){ System.out.println(name);

}

}

}

演示3：

package cn.itcast.io.p2.file.demo;

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

import java.text.DateFormat; import java.util.Date;

public class FileMethodDemo {

/\*\*

* @param args
* @throws IOException

\*/

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

/\*

\* File对象的常见方法。

\*

\* 1，获取。

\* 1.1 获取文件名称。

\* 1.2 获取文件路径。

\* 1.3 获取文件大小。

\* 1.4 获取文件修改时间。

\*

\* 2，创建与删除。

\*

\* 3，判断。

\*

\* 4， 重命名

\*

\*/

|  |  |  |
| --- | --- | --- |
| //  //  //  // |  | getDemo(); createAndDeleteDemo(); isDemo(); renameToDemo(); |
| // | } | listRootsDemo(); |

public static void listRootsDemo() { File file = new File("d:\\");

System.out.println("getFreeSpace:"+file.getFreeSpace()); System.out.println("getTotalSpace:"+file.getTotalSpace()); System.out.println("getUsableSpace:"+file.getUsableSpace());

// File[] files = File.listRoots();

// for(File file : files){

// System.out.println(file);

// }

}

public static void renameToDemo() { File f1 = new File("c:\\9.mp3"); File f2 = new File("d:\\aa.mp3"); boolean b = f1.renameTo(f2);

System.out.println("b="+b);

}

public static void isDemo() throws IOException{ File f = new File("aaa");

// f.mkdir(); f.createNewFile();

// boolean b = f.exists();

// System.out.println("b="+b);

// 最好先判断是否存在。 System.out.println(f.isFile()); System.out.println(f.isDirectory());

}

public static void createAndDeleteDemo() throws IOException { File dir = new File("abc\\q\\e\\c\\z\\r\\w\\y\\f\\e\\g\\s");

// boolean b = dir.mkdir();//make directory

// System.out.println("b="+b);

// dir.mkdirs();//创建多级目录 System.out.println(dir.delete());

// System.out.println(dir.delete());

// 文件的创建和删除。

// File file = new File("file.txt");

/\*

\* 和输出流不一样，如果文件不存在，则创建，如果文件存在，则不创建。

\*

\*/

// boolean b = file.createNewFile();

// System.out.println("b="+b);

// boolean b = file.delete();

// System.out.println("b="+b);

}

public static void getDemo(){

// File file = new File("E:\\java0331\\day22e\\a.txt"); File file = new File("a.txt");

String name = file.getName();

String absPath = file.getAbsolutePath();//绝对路径。 String path = file.getPath();

long len = file.length();

long time = file.lastModified(); Date date = new Date(time);

DateFormat dateFormat = DateFormat.getDateTimeInstance(DateFormat.LONG,DateFormat.LONG);

String str\_time = dateFormat.format(date);

System.out.println("parent:"+file.getParent());

System.out.println("name:"+name); System.out.println("absPath:"+absPath); System.out.println("path:"+path); System.out.println("len:"+len); System.out.println("time:"+time); System.out.println("str\_time:"+str\_time);

}

}

153.IO流操作规徇及需求演示：

流的操作规律： 之所以要弄清楚这个规律，是因为流对象太多，开发时不知道用哪个对象合适。

想要知道开发时用到哪些对象。只要通过四个明确即可。

1，明确源和目的(汇) 源：InputStream Reader 目的：OutputStream Writer

2，明确数据是否是纯文本数据。 源：是纯文本：Reader

否：InputStream 目的：是纯文本 Writer

否：OutputStream 到这里，就可以明确需求中具体要使用哪个体系。

3，明确具体的设备。 源设备：

硬盘：File 键盘：System.in 内存：数组 网络：Socket流

目的设备：

硬盘：File 控制台：System.out 内存：数组 网络：Socket流

4，是否需要其他额外功能。

1，是否需要高效(缓冲区);

是，就加上buffer.

2，转换。

--------------------------------------------------------------------------------

需求1：复制一个文本文件。

1,明确源和目的。 源：InputStream Reader 目的：OutputStream Writer

2,是否是纯文本？ 是！ 源：Reader 目的：Writer

3,明确具体设备。 源：

硬盘：File 目的：

硬盘：File

FileReader fr = new FileReader("a.txt"); FileWriter fw = new FileWriter("b.txt");

4,需要额外功能吗？ 需要，需要高效。

BufferedReader bufr = new BufferedReader(new FileReader("a.txt")); BufferedWriter bufw = new BufferedWriter(new FileWriter("b.txt"));

--------------------------------------------------------------------------------

需求2：读取键盘录入信息，并写入到一个文件中。

1,明确源和目的。 源：InputStream Reader 目的：OutputStream Writer

2，是否是纯文本呢？ 是， 源：Reader 目的：Writer

3，明确设备 源：

键盘。System.in 目的：

硬盘。File

InputStream in = System.in;

FileWriter fw = new FileWriter("b.txt");

这样做可以完成，但是麻烦。将读取的字节数据转成字符串。再由字符流操作。

4，需要额外功能吗？

需要。转换。 将字节流转成字符流。因为名确的源是Reader，这样操作文本数据做便捷。 所以要将已有的字节流转成字符流。使用字节-->字符 。InputStreamReader

InputStreamReader isr = new InputStreamReader(System.in); FileWriter fw = new FileWriter("b.txt");

还需要功能吗？ 需要：想高效。

BufferedReader bufr = new BufferedReader(new InputStreamReader(System.in)); BufferedWriter bufw = new BufferedWriter(new FileWriter("b.txt"));

-------------------------------------------------------------------------------

需求3：将一个文本文件数据显示在控制台上。

1,明确源和目的。 源：InputStream Reader 目的：OutputStream Writer

2，是否是纯文本呢？ 是， 源：Reader 目的：Writer

3，明确具体设备 源：

硬盘：File 目的：

控制台：System.out

FileReader fr = new FileReader("a.txt"); OutputStream out = System.out;//PrintStream

4，需要额外功能吗？

需要，转换。

FileReader fr= new FileReader("a.txt");

OutputStreamWriter osw = new OutputStreamWriter(System.out);

需要，高效。

BufferedReader bufr = new BufferedReader(new FileReader("a.txt")); BufferedWriter bufw = new BufferedWriter(new OutputStreamWriter(System.out));

--------------------------------------------------------------------------------

需求4：读取键盘录入数据，显示在控制台上。

1,明确源和目的。 源：InputStream Reader 目的：OutputStream Writer

2，是否是纯文本呢？ 是， 源：Reader 目的：Writer

3，明确设备。 源：

键盘：System.in 目的：

控制台：System.out

InputStream in = System.in; OutputStream out = System.out;

4，明确额外功能？ 需要转换，因为都是字节流，但是操作的却是文本数据。 所以使用字符流操作起来更为便捷。

InputStreamReader isr = new InputStreamReader(System.in); OutputStreamWriter osw = new OutputStreamWriter(System.out);

为了将其高效。

BufferedReader bufr = new BufferedReader(new InputStreamReader(System.in)); BufferedWriter bufw = new BufferedWriter(new OutputStreamWriter(System.out));

--------------------------------------------------------------------------------

5，将一个中文字符串数据按照指定的编码表写入到一个文本文件中.

1，目的。OutputStream，Writer 2，是纯文本，Writer。 3，设备：硬盘File

FileWriter fw = new FileWriter("a.txt"); fw.write("你好");

注意：既然需求中已经明确了指定编码表的动作。 那就不可以使用FileWriter，因为FileWriter内部是使用默认的本地码表。 只能使用其父类。OutputStreamWriter.

OutputStreamWriter 接收一个字节输出流 对象，既然是操作文 件，那么该对象应该 是

FileOutputStream

OutputStreamWriter osw = new OutputStreamWriter(new FileOutputStream("a.txt"),charsetName);

需要高效吗？

BufferedWriter bufw = new BufferedWriter(new OutputStreamWriter(new FileOutputStream("a.txt"),charsetName));

什么时候使用转换流呢？

1，源或者目的对应的设备是字节流，但是操作的却是文本数据，可以使用转换作为桥梁。 提高对文本操作的便捷。

2，一旦操作文本涉及到具体的指定编码表时，必须使用转换流 。

154.递弻简述：

**package** cn.itcast.io.p1.digui;

**public class** DiGuiDemo {

/\*\*

\* **@param** args

\*/

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

/\*

\* 递归：

\* 函数自身直接或者间接的调用到了自身。

\* 一个功能在被重复使用，并每次使用时，参与运算的结果和上一次调用有关。

\* 这时可以用递归来解决问题。

\* 注意：

\* 1，递归一定明确条件。否则容易栈溢出。

\* 2，注意一下递归的次数。

\*/

// show();

// toBin(6);

**int** sum = *getSum*(9000);

System.*out*.println(sum);

}

**public static int** getSum(**int** num){

**int** x = 9;

**if**(num==1)

**return** 1;

**return** num+*getSum*(num-1);

}

**public static void** toBin(**int** num){

**if**(num>0){

*toBin*(num/2); System.*out*.println(num%2);

}

}

/\*

public static void show(){ method();

}

public static void method(){

}

\*/

}

155.深度遍历（递弻使用）：

**package** cn.itcast.io.p1.file.test;

**import** java.io.File;

/\*

\* 需求：对指定目录进行所有内容的列出（包含子目录中的内容）

\* 也可以理解为深度遍历。

\*/

**public class** FileTest {

/\*\*

\* **@param** args

\*/

**public static void** main(String[] args) { File dir = **new** File("e:\\demodir");

*listAll*(dir,0);

}

**public static void** listAll(File dir,**int** level) {

System.*out*.println(*getSpace*(level)+dir.getName());

//获取指定目录下当前的所有文件夹或者文件对象

level++;

File[] files = dir.listFiles();

**for**(**int** x=0; x<files.length; x++){

**if**(files[x].isDirectory()){

*listAll*(files[x],level);

}

### else

System.*out*.println(*getSpace*(level)+files[x].getName());

}

}

**private static** String getSpace(**int** level) {

StringBuilder sb = **new** StringBuilder();

sb.append("|--");

**for**(**int** x=0; x<level; x++){ sb.insert(0,"| ");

}

**return** sb.toString();

}

}

# 156.删除一个带内容的目弽（递弻使用）：

**package** cn.itcast.io.p1.file.test;

**import** java.io.File;

/\*

\* 删除一个带内容的目录。

\* 原理：必须从最里面往外删。

\* 需要进行深度遍历。

\*/

**public class** RemoveDirTest {

/\*\*

\* **@param** args

\*/

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

File dir = **new** File("e:\\demodir");

// dir.delete();

*removeDir*(dir);

}

**public static void** removeDir(File dir) { File[] files = dir.listFiles(); **for**(File file : files){

**if**(file.isDirectory()){

*removeDir*(file);

}**else**{

System.*out*.println(file+":"+file.delete());

}

System.*out*.println(dir+":"+dir.delete());

}

}

# 157.Properties集合+IO流操作演示：

package cn.itcast.io.p2.properties;

import java.io.BufferedReader; import java.io.File;

import java.io.FileInputStream; import java.io.FileOutputStream; import java.io.FileReader; import java.io.FileWriter; import java.io.IOException; import java.util.Properties; import java.util.Set;

public class PropertiesDemo {

/\*\*

* @param args
* @throws IOException

\*/

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

/\*

* Map
* |--Hashtable
* |--Properties:

\*

* Properties集合：

\* 特点：

\* 1，该集合中的键和值都是字符串类型。

\* 2，集合中的数据可以保存到流中，或者从流获取。

\*

\* 通常该集合用于操作以键值对形式存在的配置文件。

\*/

// methodDemo\_4();

// myLoad();

test();

}

//对已有的配置文件中的信息进行修改。

/\*

\* 读取这个文件。

\* 并将这个文件中的键值数据存储到集合中。

\* 在通过集合对数据进行修改。

\* 在通过流将修改后的数据存储到文件中。

\*/

public static void test() throws IOException{

//读取这个文件。

File file = new File("info.txt"); if(!file.exists()){

file.createNewFile();

}

FileReader fr = new FileReader(file);

//创建集合存储配置信息。

Properties prop = new Properties();

//将流中信息存储到集合中。 prop.load(fr);

prop.setProperty("wangwu", "16"); FileWriter fw = new FileWriter(file); prop.store(fw,"");

// prop.list(System.out);

fw.close();

fr.close();

}

//模拟一下load方法。

public static void myLoad() throws IOException{ Properties prop = new Properties();

BufferedReader bufr = new BufferedReader(new FileReader("info.txt")); String line = null;

while((line=bufr.readLine())!=null){

if(line.startsWith("#")) continue;

String[] arr = line.split("=");

// System.out.println(arr[0]+"::"+arr[1]); prop.setProperty(arr[0], arr[1]);

}

prop.list(System.out); bufr.close();

}

public static void methodDemo\_4() throws IOException { Properties prop = new Properties();

//集合中的数据来自于一个文件。

//注意；必须要保证该文件中的数据是键值对。

//需要使用到读取流。

FileInputStream fis = new FileInputStream("info.txt");

//使用load方法。 prop.load(fis);

prop.list(System.out);

}

public static void methodDemo\_3() throws IOException { Properties prop = new Properties();

//存储元素。 prop.setProperty("zhangsan","30"); prop.setProperty("lisi","31"); prop.setProperty("wangwu","36");

prop.setProperty("zhaoliu","20");

//想要将这些集合中的字符串键值信息持久化存储到文件中。

//需要关联输出流。

FileOutputStream fos = new FileOutputStream("info.txt");

//将集合中数据存储到文件中，使用store方法。 prop.store(fos, "info");

fos.close();

}

/\*\*

\* 演示Properties集合和流对象相结合的功能。

\*/

public static void methodDemo\_2(){ Properties prop = new Properties();

//存储元素。

// prop.setProperty("zhangsan","30");

// prop.setProperty("lisi","31");

// prop.setProperty("wangwu","36");

// prop.setProperty("zhaoliu","20");

prop = System.getProperties(); prop.list(System.out);

}

/\*

\* Properties集合的存和取。

\*/

public static void propertiesDemo(){

//创建一个Properties集合。

Properties prop = new Properties();

//存储元素。 prop.setProperty("zhangsan","30"); prop.setProperty("lisi","31"); prop.setProperty("wangwu","36");

prop.setProperty("zhaoliu","20");

//修改元素。 prop.setProperty("wangwu","26");

//取出所有元素。

Set<String> names = prop.stringPropertyNames();

for(String name : names){

String value = prop.getProperty(name); System.out.println(name+":"+value);

}

}

}

# 158.用Properties定义一个程序运行次数的程序：

package cn.itcast.io.p2.properties;

import java.io.File;

import java.io.FileInputStream; import java.io.FileOutputStream; import java.io.IOException; import java.util.Properties;

/\*

\* 定义功能，获取一个应用程序运行的次数，如果超过5次，给出使用次数已到请注册的提示。并不要在运行 程序。

\*

\* 思路：

\* 1，应该有计数器。

\* 每次程序启动都需要计数一次,并且是在原有的次数上进行计数。

\* 2，计数器就是一个变量。 突然冒出一想法，程序启动时候进行计数，计数器必须存在于内存并进行运算。

\* 可是程序一结束，计数器消失了。那么再次启动该程序，计数器又重新被初始化了。

\* 而我们需要多次启动同一个应用程序，使用的是同一个计数器。

\* 这就需要计数器的生命周期变长，从内存存储到硬盘文件中。

\*

\* 3,如何使用这个计数器呢？

\* 首先，程序启动时，应该先读取这个用于记录计数器信息的配置文件。

\* 获取上一次计数器次数。 并进行试用次数的判断。

\* 其次，对该次数进行自增，并自增后的次数重新存储到配置文件中。

\*

\*

\* 4，文件中的信息该如何进行存储并体现。

\* 直接存储次数值可以，但是不明确该数据的含义。 所以起名字就变得很重要。

\* 这就有了名字和值的对应，所以可以使用键值对。

\* 可是映射关系map集合搞定，又需要读取硬盘上的数据，所以map+io = Properties.

\*/

public class PropertiesTest {

/\*\*

* @param args
* @throws IOException
* @throws Exception

\*/

public static void main(String[] args) throws IOException { getAppCount();

}

public static void getAppCount() throws IOException{

//将配置文件封装成File对象。

File confile = new File("count.properties");

if(!confile.exists()){ confile.createNewFile();

}

FileInputStream fis = new FileInputStream(confile); Properties prop = new Properties(); prop.load(fis);

//从集合中通过键获取次数。

String value = prop.getProperty("time");

//定义计数器。记录获取到的次数。 int count =0; if(value!=null){

count = Integer.parseInt(value); if(count>=5){

// System.out.println("使用次数已到，请注册，给钱！");

// return;

throw new RuntimeException("使用次数已到，请注册，给钱！");

}

}

count++;

//将改变后的次数重新存储到集合中。 prop.setProperty("time", count+"");

FileOutputStream fos = new FileOutputStream(confile); prop.store(fos, "");

fos.close();

fis.close();

}

}

# 159.建立一个指定扩展名文件的列表（练习）：

package cn.itcast.io.p3.test;

import java.io.BufferedWriter; import java.io.File;

import java.io.FileWriter; import java.io.FilenameFilter; import java.io.IOException; import java.util.ArrayList;

import java.util.List;

/\*

\* 获取指定目录下，指定扩展名的文件(包含子目录中的)

\* 这些文件的绝对路径写入到一个文本文件中。

\*

\* 简单说，就是建立一个指定扩展名的文件的列表。

\*

\* 思路：

\* 1，必须进行深度遍历。

\* 2，要在遍历的过程中进行过滤。将符合条件的内容都存储到容器中。

\* 3，对容器中的内容进行遍历并将绝对路径写入到文件中。

\*/

public class Test {

/\*\*

* @param args
* @throws IOException

\*/

public static void main(String[] args) throws IOException { File dir = new File("e:\\java0331");

FilenameFilter filter = new FilenameFilter(){

@Override

public boolean accept(File dir, String name) {

return name.endsWith(".java");

}

};

List<File> list = new ArrayList<File>(); getFiles(dir,filter,list);

File destFile = new File(dir,"javalist.txt");

}

/\*\*

write2File(list,destFile);

\* 对指定目录中的内容进行深度遍历，并按照指定过滤器，进行过滤，

\* 将过滤后的内容存储到指定容器List中。

* @param dir
* @param filter
* @param list

\*/

public static void getFiles(File dir,FilenameFilter filter,List<File> list){

File[] files = dir.listFiles();

for(File file : files){ if(file.isDirectory()){

//递归啦！

getFiles(file,filter,list);

}else{

//对遍历到的文件进行过滤器的过滤。将符合条件File对象，存储到List集合中。 if(filter.accept(dir, file.getName())){

list.add(file);

}

}

}

}

public static void write2File(List<File> list,File destFile)throws IOException{ BufferedWriter bufw = null;

try {

bufw = new BufferedWriter(new FileWriter(destFile)); for(File file : list){

bufw.write(file.getAbsolutePath()); bufw.newLine();

bufw.flush();

}

} /\*catch(IOException e){

throw new RuntimeException("写入失败");

}\*/finally{

if(bufw!=null) try {

bufw.close();

} catch (IOException e) {

throw new RuntimeException("关闭失败");

}

}

}

}

160.打印流简述：

package cn.itcast.io.p4.print.demo;

import java.io.FileNotFoundException;

import java.io.IOException; import java.io.PrintStream;

public class PrintStreamDemo {

/\*\*

* @param args
* @throws IOException

\*/

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

/\*

* PrintStream:

\* 1,提供了打印方法可以对多种数据类型值进行打印。并保持数据的表示形式。

* 2，它不抛IOException.

\*

\* 构造函数，接收三种类型的值：

\* 1，字符串路径。

\* 2，File对象。

\* 3，字节输出流。

\*/

PrintStream out = new PrintStream("print.txt");

// int by = read();

// write(by);

// out.write(610);//只写最低8位，

// out.print(97);//将97先变成字符保持原样将数据打印到目的地。

out.close();

}

}

# 161.PrintWriter简述及使用：

package cn.itcast.io.p4.print.demo;

import java.io.BufferedReader; import java.io.FileWriter; import java.io.IOException; import java.io.InputStreamReader; import java.io.PrintWriter;

public class PrintWriterDemo {

/\*\*

* @param args
* @throws IOException

\*/

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

/\*

\* PrintWriter：字符打印流。

\* 构造函数参数：

\* 1，字符串路径。

\* 2，File对象。

\* 3，字节输出流。

\* 4，字符输出流。

\*

\*/

BufferedReader bufr = new BufferedReader(new InputStreamReader(System.in)); PrintWriter out = new PrintWriter(new FileWriter("out.txt"),true);

String line = null; while((line=bufr.readLine())!=null){

if("over".equals(line)) break;

out.println(line.toUpperCase());

// out.flush();

}

out.close();

bufr.close();

}

}

# 162.SequenceInputStream序列流使用：

package cn.itcast.io.p4.sequence.demo;

import java.io.FileInputStream; import java.io.FileOutputStream; import java.io.IOException;

import java.io.SequenceInputStream; import java.util.ArrayList;

import java.util.Collections; import java.util.Enumeration; import java.util.Iterator;

public class SequenceInputStreamDemo {

/\*\*

* @param args
* @throws IOException

\*/

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

/\*

\* 需求：将1.txt 2.txt 3.txt文件中的数据合并到一个文件中。

\*/

// Vector<FileInputStream> v = new Vector<FileInputStream>();

// v.add(new FileInputStream("1.txt"));

// v.add(new FileInputStream("2.txt"));

// v.add(new FileInputStream("3.txt"));

// Enumeration<FileInputStream> en = v.elements();

ArrayList<FileInputStream> al = new ArrayList<FileInputStream>(); for(int x=1; x<=3; x++){

al.add(new FileInputStream(x+".txt"));

}

Enumeration<FileInputStream> en = Collections.enumeration(al);

/\*

final Iterator<FileInputStream> it = al.iterator(); Enumeration<FileInputStream> en = new Enumeration<FileInputStream>(){

@Override

public boolean hasMoreElements() {

return it.hasNext();

}

@Override

public FileInputStream nextElement() {

}

};\*/

return it.next();

SequenceInputStream sis = new SequenceInputStream(en); FileOutputStream fos = new FileOutputStream("1234.txt"); byte[] buf = new byte[1024];

int len = 0;

while((len=sis.read(buf))!=-1){ fos.write(buf,0,len);

}

fos.close();

sis.close();

}

}

163.文件切割器：

package cn.itcast.io.p1.splitfile;

import java.io.File;

import java.io.FileInputStream; import java.io.FileOutputStream; import java.io.IOException; import java.util.Properties;

/\*

\* 文件切割器。

\*/

public class SplitFileDemo {

private static final int SIZE = 1024 \* 1024;

/\*\*

* @param args
* @throws Exception

\*/

public static void main(String[] args) throws Exception { File file = new File("c:\\aa.mp3");

splitFile\_2(file);

}

private static void splitFile\_2(File file) throws IOException {

// 用读取流关联源文件。

FileInputStream fis = new FileInputStream(file);

// 定义一个1M的缓冲区。

byte[] buf = new byte[SIZE];

// 创建目的。

FileOutputStream fos = null;

int len = 0; int count = 1;

/\*

\* 切割文件时，必须记录住被切割文件的名称，以及切割出来碎片文件的个数。 以方便于合并。

\* 这个信息为了进行描述，使用键值对的方式。用到了properties对象

\*

\*/

Properties prop = new Properties();

File dir = new File("c:\\partfiles"); if (!dir.exists())

dir.mkdirs();

while ((len = fis.read(buf)) != -1) {

fos = new FileOutputStream(new File(dir, (count++) + ".part")); fos.write(buf, 0, len);

fos.close();

}

//将被切割文件的信息保存到prop集合中。 prop.setProperty("partcount", count+""); prop.setProperty("filename", file.getName());

fos = new FileOutputStream(new File(dir,count+".properties"));

//将prop集合中的数据存储到文件中。 prop.store(fos, "save file info");

fos.close();

fis.close();

}

public static void splitFile(File file) throws IOException {

// 用读取流关联源文件。

FileInputStream fis = new FileInputStream(file);

// 定义一个1M的缓冲区。

byte[] buf = new byte[SIZE];

// 创建目的。

FileOutputStream fos = null;

int len = 0; int count = 1;

File dir = new File("c:\\partfiles"); if (!dir.exists())

dir.mkdirs();

while ((len = fis.read(buf)) != -1) {

fos = new FileOutputStream(new File(dir, (count++) + ".part")); fos.write(buf, 0, len);

}

fos.close();

fis.close();

}

}

164.合并文件（流）+配置文件：

package cn.itcast.io.p1.splitfile;

import java.io.File;

import java.io.FileInputStream; import java.io.FileOutputStream; import java.io.IOException;

import java.io.SequenceInputStream; import java.util.ArrayList;

import java.util.Collections; import java.util.Enumeration; import java.util.Properties;

public class MergeFile {

/\*\*

* @param args
* @throws IOException

\*/

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

File dir = new File("c:\\partfiles");

mergeFile\_2(dir);

}

public static void mergeFile\_2(File dir) throws IOException {

/\*

\* 获取指定目录下的配置文件对象。

\*/

File[] files = dir.listFiles(new SuffixFilter(".properties"));

");

if(files.length!=1)

throw new RuntimeException(dir+",该目录下没有properties扩展名的文件或者不唯一

//记录配置文件对象。

File confile = files[0];

//获取该文件中的信息================================================。

Properties prop = new Properties(); FileInputStream fis = new FileInputStream(confile);

prop.load(fis);

String filename = prop.getProperty("filename");

int count = Integer.parseInt(prop.getProperty("partcount"));

//获取该目录下的所有碎片文件。 ==============================================

File[] partFiles = dir.listFiles(new SuffixFilter(".part"));

if(partFiles.length!=(count-1)){

throw new RuntimeException(" 碎片文件不符合要求，个数不对!应该"+count+"个");

}

//将碎片文件和流对象关联 并存储到集合中。

ArrayList<FileInputStream> al = new ArrayList<FileInputStream>(); for(int x=0; x<partFiles.length; x++){

al.add(new FileInputStream(partFiles[x]));

}

//将多个流合并成一个序列流。

Enumeration<FileInputStream> en = Collections.enumeration(al); SequenceInputStream sis = new SequenceInputStream(en);

FileOutputStream fos = new FileOutputStream(new File(dir,filename)); byte[] buf = new byte[1024];

int len = 0; while((len=sis.read(buf))!=-1){

fos.write(buf,0,len);

}

fos.close();

sis.close();

}

public static void mergeFile(File dir) throws IOException{

ArrayList<FileInputStream> al = new ArrayList<FileInputStream>(); for(int x=1; x<=3 ;x++){

al.add(new FileInputStream(new File(dir,x+".part")));

}

Enumeration<FileInputStream> en = Collections.enumeration(al); SequenceInputStream sis = new SequenceInputStream(en);

FileOutputStream fos = new FileOutputStream(new File(dir,"1.bmp")); byte[] buf = new byte[1024];

int len = 0; while((len=sis.read(buf))!=-1){

fos.write(buf,0,len);

}

fos.close();

sis.close();

}

}

# 165.Serializable标记接口：

**package** cn.itcast.io.p2.bean;

**import** java.io.Serializable;

/\*

\* Serializable:用于给被序列化的类加入ID号。

\* 用于判断类和对象是否是同一个版本。

\*/

**public class** Person **implements** Serializable/\*标记接口\*/ {

/\*\*

\* transient:非静态数据不想被序列化可以使用这个关键字修饰。

\*/

**private static final long** *serialVersionUID* = 9527l;

**private transient** String name;

### private static int *age*;

**public** Person(String name, **int** age) {

**super**(); **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;

}

}

166.ObjectStream简述：

package cn.itcast.io.p2.objectstream;

import java.io.FileInputStream; import java.io.FileNotFoundException; import java.io.FileOutputStream; import java.io.IOException;

import java.io.ObjectInputStream; import java.io.ObjectOutputStream;

import cn.itcast.io.p2.bean.Person; public class ObjectStreamDemo {

/\*\*

* @param args
* @throws IOException
* @throws ClassNotFoundException

\*/

public static void main(String[] args) throws IOException, ClassNotFoundException

{

// writeObj(); readObj();

}

public static void readObj() throws IOException, ClassNotFoundException { ObjectInputStream ois = new ObjectInputStream(new

FileInputStream("obj.object"));

//对象的反序列化。

Person p = (Person)ois.readObject(); System.out.println(p.getName()+":"+p.getAge()); ois.close();

}

public static void writeObj() throws IOException, IOException { ObjectOutputStream oos = new ObjectOutputStream(new

FileOutputStream("obj.object"));

//对象序列化。 被序列化的对象必须实现Serializable接口。

oos.writeObject(new Person("小强",30));

oos.close();

}

}

# 167.RandomAccessFile随机访问文件简述：

package cn.itcast.io.p3.randomfile;

import java.io.FileNotFoundException; import java.io.IOException;

import java.io.RandomAccessFile; public class RandomAccessFileDemo {

/\*\*

* @param args
* @throws IOException

\*/

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

/\*

\* RandomAccessFile

\* 一看这个类名字，纠结。不是io体系中的子类。

\*

\* 特点：

\* 1，该对象即能读，又能写。

\* 2，该对象内部维护了一个byte数组，并通过指针可以操作数组中的元素，

\* 3，可以通过getFilePointer方法获取指针的位置，和通过seek方法设置指针的位置。

\* 4，其实该对象就是将字节输入流和输出流进行了封装。

\* 5，该对象的源或者目的只能是文件。通过构造函数就可以看出。

\*/

// writeFile();

// readFile(); randomWrite();

}

public static void randomWrite() throws IOException{ RandomAccessFile raf = new RandomAccessFile("ranacc.txt", "rw");

//往指定位置写入数据。 raf.seek(3\*8);

raf.write("哈哈".getBytes()); raf.writeInt(108);

raf.close();

}

public static void readFile() throws IOException { RandomAccessFile raf = new RandomAccessFile("ranacc.txt", "r");

//通过seek设置指针的位置。

raf.seek(1\*8);//随机的读取。只要指定指针的位置即可。

byte[] buf = new byte[4]; raf.read(buf);

String name = new String(buf); int age = raf.readInt();

System.out.println("name="+name); System.out.println("age="+age);

System.out.println("pos:"+raf.getFilePointer()); raf.close();

}

//使用RandomAccessFile对象写入一些人员信息，比如姓名和年龄。 public static void writeFile() throws IOException{

/\*

\* 如果文件不存在，则创建，如果文件存在，不创建

\*

\*/

RandomAccessFile raf = new RandomAccessFile("ranacc.txt","rw");

raf.write("张三".getBytes()); raf.writeInt(97); raf.write("小强".getBytes()); raf.writeInt(99);

//

raf.close();

}

}

168.管道流简述：

package cn.itcast.io.p4.piped;

import java.io.IOException; import java.io.PipedInputStream; import java.io.PipedOutputStream;

public class PipedStream {

/\*\*

* @param args
* @throws IOException

\*/

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

PipedInputStream input = new PipedInputStream(); PipedOutputStream output = new PipedOutputStream();

input.connect(output);

new Thread(new Input(input)).start(); new Thread(new Output(output)).start();

}

}

class Input implements Runnable{

private PipedInputStream in; Input(PipedInputStream in){

this.in = in;

}

public void run(){

try {

byte[] buf = new byte[1024]; int len = in.read(buf);

String s = new String(buf,0,len);

System.out.println("s="+s); in.close();

} catch (Exception e) {

// TODO: handle exception

}

}

}

class Output implements Runnable{ private PipedOutputStream out; Output(PipedOutputStream out){

this.out = out;

}

public void run(){

try {

Thread.sleep(5000); out.write("hi，管道来了！".getBytes());

} catch (Exception e) {

// TODO: handle exception

}

}

}

# 169.DataStream数据流简述：

package cn.itcast.io.p5.datastream;

import java.io.DataInputStream; import java.io.DataOutputStream; import java.io.FileInputStream;

import java.io.FileOutputStream; import java.io.IOException;

public class DataSteamDemo {

/\*\*

* @param args
* @throws IOException

\*/

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

// writeData(); readData();

}

public static void readData() throws IOException {

DataInputStream dis = new DataInputStream(new FileInputStream("data.txt")); String str = dis.readUTF();

System.out.println(str);

}

public static void writeData() throws IOException {

DataOutputStream dos = new DataOutputStream(new FileOutputStream("data.txt"));

dos.writeUTF("你好"); dos.close();

}

}

170.字节数组流简述：

package cn.itcast.io.p6.bytestream;

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

public class ByteArrayStreamDemo {

/\*\*

* @param args
* @throws IOException

\*/

public static void main(String[] args) {

ByteArrayInputStream bis = new ByteArrayInputStream("abcedf".getBytes()); ByteArrayOutputStream bos = new ByteArrayOutputStream();

int ch = 0;

while((ch=bis.read())!=-1){ bos.write(ch);

}

System.out.println(bos.toString());

}

}

171.编解码简述：

package cn.itcast.io.p7.encode;

import java.io.IOException;

import java.io.UnsupportedEncodingException; public class EncodeDemo {

/\*\*

* @param args
* @throws IOException

\*/

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

/\*

\* 字符串 --> 字节数组：编码。

\* 字节数组 --> 字符串：解码。

\*

\* 你好：GBK: -60 -29 -70 -61

\*

\* 你好: utf-8: -28 -67 -96 -27 -91 -67

\*

\*

\* 如果你编错了，解不出来。

\* 如果编对了，解错了，有可能有救。

\*/

String str = "谢谢";

byte[] buf = str.getBytes("gbk"); String s1 = new String(buf,"UTF-8"); System.out.println("s1="+s1);

byte[] buf2 = s1.getBytes("UTF-8");//获取源字节.

printBytes(buf2);//-17 -65 -67 -17 -65 -67 -17 -65 -67

//-17 -65 -67 -17 -65 -67 -17 -65 -67 -17 -65 -67

//-48 -69 -48 -69

String s2 = new String(buf2,"GBK"); System.out.println("s2="+s2);

// encodeDemo(str);

}

/\*\*

* @param str
* @throws UnsupportedEncodingException

\*/

public static void encodeDemo(String str) throws UnsupportedEncodingException {

//编码；

byte[] buf = str.getBytes("UTF-8");

// printBytes(buf);

//解码：

String s1 = new String(buf,"UTF-8");

System.out.println("s1="+s1);

}

private static void printBytes(byte[] buf) { for(byte b : buf){

System.out.print(b +" ");

}

}

}

172."联通"问题（绅节）：

package cn.itcast.io.p7.encode;

import java.io.IOException;

import java.io.UnsupportedEncodingException;

public class LianTong {

/\*\*

* @param args
* @throws IOException

\*/

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

String str = "联通";

/\* 11000001

10101010

11001101

10101000

\*/

byte[] buf = str.getBytes("gbk"); for(byte b :buf){

System.out.println(Integer.toBinaryString(b&255));

}

}

}

# 173.定义方法按最大字节数叏子串（练习）：

**package** cn.itcast.io.p7.encode;

**import** java.io.IOException;

**public class** Test {

/\*\*

* **@param** args
* **@throws** IOException

\*/

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

String str = "ab你好cd谢谢";

// str = "ab琲琲cd琲琲";

// int len = str.getBytes("gbk").length;

// for(int x=0; x<len; x++){

// System.out.println(" 截取 "+(x+1)+" 个 字 节 结 果 是 ： "+cutStringByByte(str, x+1));

// }

x+1));

**int** len = str.getBytes("utf-8").length;

**for**(**int** x=0; x<len; x++){

System.*out*.println(" 截取 "+(x+1)+" 个 字 节 结 果 是 ： "+*cutStringByU8Byte*(str,

}

// String str = "琲";

// byte[] buf = str.getBytes("gbk");

// for(byte b : buf){

// System.out.println(b);//-84 105

// }

}

/\*

在java中，字符串“abcd”与字符串“ab你好”的长度是一样，都是四个字符。 但对应的字节数不同，一个汉字占两个字节。 定义一个方法，按照最大的字节数来取子串。 如：对于“ab你好”，如果取三个字节，那么子串就是ab与“你”字的半个， 那么半个就要舍弃。如果去四个字节就是“ab你”，取五个字节还是“ab你”.

\*/

**public static** String cutStringByU8Byte(String str, **int** len) **throws** IOException

{

**byte**[] buf = str.getBytes("utf-8");

**int** count = 0;

**for**(**int** x=len-1; x>=0; x--){

**if**(buf[x]<0) count++;

### else

**break**;

}

**if**(count%3==0)

**return new** String(buf,0,len,"utf-8");

**else if**(count%3==1)

**return new** String(buf,0,len-1,"utf-8");

### else

**return new** String(buf,0,len-2,"utf-8");

}

**public static** String cutStringByByte(String str,**int** len) **throws** IOException{

**byte**[] buf = str.getBytes("gbk");

**int** count = 0;

**for**(**int** x=len-1; x>=0; x--){

**if**(buf[x]<0) count++;

### else

**break**;

}

**if**(count%2==0)

**return new** String(buf,0,len,"gbk");

### else

**return new** String(buf,0,len-1,"gbk");

}

}

174.IO精华练习：

习题1：

//定义一个比较器

**package** cn.itcast.io.p8.test;

**import** java.util.Comparator;

**public class** ComparatorByMath **implements** Comparator<Student> {

@Override

**public int** compare(Student o1, Student o2) {

**int** temp = o1.getMa() - o2.getMa();

**return** temp==0?o1.getName().compareTo(o2.getName()):temp;

}

}

//习题

package cn.itcast.io.p8.test;

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

import java.util.Collections; import java.util.Comparator; import java.util.Set;

/\*

有五个学生，每个学生有3门课的成绩，定义一种比较直观的文本文件格式， 输入学生姓名和成绩，输入的格式：name,30,30,30从键盘输入以上数据（包括姓名，三门课成绩）， 按总分数从高到低的顺序将学生信息存放在磁盘文件"stu.txt"中。

思路：

1，3门课的成绩都是数据，为了便于操作，将其封装到学生对象中。 学生本身就是问题领域中涉及的对象，对学生描述。

学生：

姓名，语文成绩，英语成绩，数学成绩，总分.

2，数据来源于键盘录入，将这些数据都封装到每一个学生对象中。

3，按照总分排个序，很熟，但是这些数据都存储到了学生对象中，其实是学生对象排序。 学生对象很多，先想到的就是存起来--集合-不重复排序-TreeSet。

4，将排序后的信息写入到一个文件中。定义操作文件的输出流。 将信息写入到文件中。

aa,10,30,50 zz,30,60,30 qq,30,90,70 cc,70,80,90 pp,80,80,80

\*/

public class StudentInfoTest {

/\*\*

* @param args
* @throws IOException

\*/

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

Comparator<Student> comp = Collections.reverseOrder();

comp = Collections.reverseOrder(new ComparatorByMath()); Set<Student> set = StudentInfoTool.getStudents(comp);

File file = new File("stuinfo.txt"); StudentInfoTool.write2File(set, file);

}

}

习题2：

//定义一个学生类

**package** cn.itcast.io.p8.test;

**public class** Student **implements** Comparable<Student> {

**private** String name; **private int** cn,en,ma; **private int** sum;

**public** Student(String name, **int** cn, **int** en, **int** ma) {

**super**(); **this**.name = name; **this**.cn = cn; **this**.en = en; **this**.ma = ma;

sum = cn + en + ma;

}

@Override

**public int** hashCode() { **final int** prime = 31; **int** result = 1;

result = prime \* result + ((name == **null**) ? 0 : name.hashCode()); result = prime \* result + sum;

**return** result;

}

@Override

**public boolean** equals(Object obj) {

**if** (**this** == obj)

**return true**; **if** (obj == **null**)

### return false;

**if** (getClass() != obj.getClass())

### return false;

Student other = (Student) obj;

**if** (name == **null**) {

**if** (other.name != **null**) **return false**;

} **else if** (!name.equals(other.name))

### return false;

**if** (sum != other.sum)

### return false; return true;

}

**public** String getName() {

**return** name;

}

**public void** setName(String name) {

**this**.name = name;

}

**public int** getCn() {

**return** cn;

}

**public void** setCn(**int** cn) {

**this**.cn = cn;

}

**public int** getEn() {

**return** en;

}

**public void** setEn(**int** en) {

**this**.en = en;

}

**public int** getMa() {

**return** ma;

}

**public void** setMa(**int** ma) {

**this**.ma = ma;

}

**public int** getSum() {

**return** sum;

}

**public void** setSum(**int** sum) {

**this**.sum = sum;

}

@Override

**public int** compareTo(Student o) {

**int** temp = **this**.sum-o.sum;

**return** temp==0?**this**.name.compareTo(o.name):temp;

}

@Override

**public** String toString() {

**return** "Student:["+name+","+cn+","+en+","+ma+"]";

}

}

//习题

**package** cn.itcast.io.p8.test;

**import** java.io.BufferedReader; **import** java.io.BufferedWriter; **import** java.io.File;

**import** java.io.FileWriter; **import** java.io.IOException; **import** java.io.InputStreamReader; **import** java.util.Comparator; **import** java.util.Set;

**import** java.util.TreeSet;

**public class** StudentInfoTool {

/\*\*

\* 定义功能，获取键盘录入的信息。 并将信息封装成学生对象。存储到容器中。

\* 按照学生的自然排序完成排序动作。

### @return

* **@throws** IOException

\*/

**public static** Set<Student> getStudents() **throws** IOException {

}

/\*\*

**return** *getStudents*(**null**);

\* 定义功能，获取键盘录入的信息。 并将信息封装成学生对象。存储到容器中。

\* 按照指定比较器完成排序的动作。

\*

\* **@throws** IOException

\*/

**public static** Set<Student> getStudents(Comparator<Student> comp) **throws**

IOException {

// 获取键盘录入。

BufferedReader bufr = **new** BufferedReader(**new** InputStreamReader( System.*in*));

// 创建一个集合对象。TreeSet. Set<Student> set = **null**;

**if**(comp==**null**)

set = **new** TreeSet<Student>();

### else

set = **new** TreeSet<Student>(comp); String line = **null**;

**while** ((line = bufr.readLine()) != **null**) {

**if** ("over".equals(line))// 定义键盘录入的结束标记。

### break;

// 对获取的信息进行切割，获取指定的数据内容。 String[] info\_arr = line.split(","); Student stu = **new** Student(info\_arr[0],

Integer.*parseInt*(info\_arr[1]), Integer.*parseInt*(info\_arr[2]), Integer.*parseInt*(info\_arr[3]));

// 把学生对象存储到集合中去。

set.add(stu);

}

}

/\*\*

**return** set;

\* 定义功能，将集合中的对象信息写入到指定文件中进行存储。

\*

\* **@throws** IOException

\*/

**public static void** write2File(Set<Student> set, File file)

**throws** IOException {

BufferedWriter bufw = **null**; **try** {

bufw = **new** BufferedWriter(**new** FileWriter(file));

**for** (Student stu : set) { bufw.write(stu.toString() + "\t"+stu.getSum()); bufw.newLine();

bufw.flush();

}

} **finally** {

**if** (bufw != **null**) bufw.close();

}

}

}

175.IO流大总结：

IO流： 输入流： 输出流：

字节流： 字符流：为了处理文字数据方便而出现的对象。

其实这些对象的内部使用的还是字节流(因为文字最终也是字节数据) 只不过，通过字节流读取了相对应的字节数，没有对这些字节直接操作。 而是去查了指定的(本机默认的)编码表，获取到了对应的文字。 简单说：字符流就是 ： 字节流+编码表。

--------------------------------------------------------------------------------

缓冲区：提高效率的，提高谁的效率？提高流的操作数据的效率。 所以创建缓冲区之前必须先有流。 缓冲区的基本思想：其实就是定义容器将数据进行临时存储。 对于缓冲区对象，其实就是将这个容器进行了封装，并提供了更多高效的操作方法。

缓冲区可以提高流的操作效率。 其实是使用了一种设计思想完成。设计模式：装饰设计模式。

Writer

|--TextWriter

|--MediaWriter 现在要对该体系中的对象进行功能的增强。增强的最常见手段就是缓冲区。 先将数据写到缓冲区中，再将缓冲区中的数据一次性写到目的地。

按照之前学习过的基本的思想，那就是对对象中的写方法进行覆盖。 产生已有的对象子类，复写write方法。不往目的地写，而是往缓冲区写。

所以这个体系会变成这样。

Writer

|--TextWriter write:往目的地

|--BufferTextWriter write：往缓冲区写

|--MediaWriter

|--BufferMediaWriter

想要写一些其他数据。就会子类。DataWriter，为了提高其效率，还要创建该类的子类。BufferDataWriter Writer

|--TextWriter write:往目的地

|--BufferTextWriter write：往缓冲区写

|--MediaWriter

|--BufferMediaWriter

|--DataWriter

|--BufferDataWriter

发现这个体系相当的麻烦。每产生一个子类都要有一个高效的子类。 而且这写高效的子类使用的功能原理都一样，都是缓冲区原理。无论数据是什么。 都是通过缓冲区临时存储提高效率的。 那么，对于这个体系就可以进行优化，因为没有必要让每一个对象都具备相同功能的子类。

哪个对象想要进行效率的提高，只要让缓冲区对其操作即可。也就说，单独将缓冲区进行封装变成对象。

//它的出现为了提高对象的效率。所以必须在创建它的时候先有需要被提高效率的对象 class BufferWriter

{

[];

BufferedWriter(Writer w)

{

}

/\* BufferWriter(TextWriter w)

{

}

BufferedWriter(MediaWriter w)

{

}

\*/

}

BufferWriter的出现增强了Writer中的write方法。 但是增强过后，BufferWriter对外提供的还是write方法。只不过是高效的。 所以写的实质没有变，那么BufferWriter也是Writer中的一员。 所以体系就会变成这样。

Writer

|--TextWriter

|--MediaWriter

|--BufferWriter

|--DataWriter BufferWriter出现了避免了继承体系关系的臃肿，比继承更为灵活。 如果是为了增强功能，这样方式解决起来更为方便。 所以就把这种优化，总结出来，起个名字：装饰设计模式。 装饰类和被装饰类肯定所属于同一个体系。

既然明确了BufferedReader由来。 我们也可以独立完成缓冲区的建立

原理；

1，使用流的read方法从源中读取一批数据存储到缓冲区的数组中。

2，通过计数器记录住存储的元素个数。

3，通过数组的角标来获取数组中的元素(从缓冲区中取数据).

4，指针会不断的自增，当增到数组长度，会归0.计数器会自减，当减到0时，就在从源拿一批数据进缓冲 区。

内容补足：

MyBufferedReader LineNumberReader ：可以定义行号。

--------------------------------------------------------------------------------

字符流： FileReader FileWriter

BufferedReader BufferedWriter

字节流：

InputStream OutputStream。

操作文件的字节流对象。 FileOutputStream FileInputStream BufferedOutputStream BufferedInputStream

字符流和字节流之间的转换动作。

--------------------------------------------------------------------------------

转换流：

InputStreamReader isr = new InputStreamReader(new FileInputStream("a.txt")); InputStreamReader isr = new InputStreamReader(new FileInputStream("a.txt"),"gbk"); FileReader fr = new FileReader("a.txt");

FileWriter fw = new FileWriter("b.txt");

OutputStreamWriter osw = new OutputStreamWriter(new FileOutputStream("b.txt")); OutputStreamWriter osw = new OutputStreamWriter(new FileOutputStream("b.txt"),"gbk");

转换流：字节流+编码表。 转换流的子类：FileReader，FileWriter：字节流+本地默认码表(GBK)。

如果操作文本文件使用的本地默认编码表完成编码。可以使用FileReader，或者FileWriter。因为这样写简 便。

如果对操作的文本文件需要使用指定编码表进行编解码操作，这时必须使用转换流来完成。

--------------------------------------------------------------------------------

IO流的操作规律总结：

1，明确体系：

数据源：InputStream ，Reader 数据汇：OutputStream，Writer

2，明确数据：因为数据分两种：字节，字符。 数据源：是否是纯文本数据呢？

是：Reader 否：InputStream

数据汇：

是：Writer 否：OutputStream

到这里就可以明确具体要使用哪一个体系了。 剩下的就是要明确使用这个体系中的哪个对象。

3，明确设备：

数据源：

键盘：System.in 硬盘：FileXXX 内存：数组。

网络：socket socket.getInputStream();

数据汇：

控制台：System.out 硬盘：FileXXX 内存：数组

网络：socket socket.getOutputStream();

4，明确额外功能：

1，需要转换？是，使用转换流。InputStreamReader OutputStreamWriter 2，需要高效？是，使用缓冲区。Buffered

3，需要其他？

--------------------------------------------------------------------------------

1，复制一个文本文件。

1，明确体系：

源：InputStream ，Reader 目的：OutputStream ，Writer

2，明确数据：

源：是纯文本吗？是 Reader

目的；是纯文本吗？是 Writer 3，明确设备：

源：硬盘上的一个文件。 FileReader

目的：硬盘上的一个文件。FileWriter

FileReader fr = new FileReader("a.txt"); FileWriter fw = new FileWriter("b.txt");

4，需要额外功能吗？

需要，高效，使用buffer

BufferedReader bufr = new BufferedReader(new FileReader("a.txt")); BufferedWriter bufw = new BufferedWriter(new FileWriter("b.txt"));

2，读取键盘录入，将数据存储到一个文件中。

1，明确体系：

源：InputStream ，Reader 目的：OutputStream ，Writer

2，明确数据：

源：是纯文本吗？是 Reader

目的；是纯文本吗？是 Writer 3，明确设备：

源：键盘，System.in 目的：硬盘，FileWriter InputStream in = System.in;

FileWriter fw = new FileWriter("a.txt"); 4，需要额外功能吗？

需要，因为源明确的体系时Reader。可是源的设备是System.in。

所以为了方便于操作文本数据，将源转成字符流。需要转换流。InputStreamReader InputStreamReader isr = new InputStreamReader(System.in); FileWriter fw = new FileWriter("a.txt");

需要高效不?需要。Buffer

BufferedReader bufr = new BufferedReader(new InputStreamReader(System.in)); BufferedWriter bufw = new BufferedWriter(new FileWriter("a.txt"));

3，读取一个文本文件，将数据展现在控制台上。

1，明确体系：

源：InputStream ，Reader 目的：OutputStream ，Writer

2，明确数据：

源：是纯文本吗？是 Reader

目的；是纯文本吗？是 Writer 3，明确设备：

源：硬盘文件，FileReader。 目的：控制台：System.out。

FileReader fr = new FileReader("a.txt"); OutputStream out = System.out;

4，需要额外功能？

因为源是文本数据，确定是Writer体系。所以为了方便操作字符数据， 需要使用字符流，但是目的又是一个字节输出流。 需要一个转换流，OutputStreamWriter

FileReader fr = new FileReader("a.txt");

OutputStreamWriter osw = new OutputStreamWriter(System.out);

需要高效吗？需要。

BufferedReader bufr = new BufferedReader(new FileReader("a.txt")); BufferedWriter bufw = new BufferedWriter(new

OutputStreamWriter(System.out));

4，读取键盘录入，将数据展现在控制台上。

1，明确体系：

源：InputStream ，Reader 目的：OutputStream ，Writer

2，明确数据：

源：是纯文本吗？是 Reader

目的；是纯文本吗？是 Writer 3，明确设备：

源：键盘：System.in 目的：控制台：System.out

InputStream in = System.in; OutputStream out = System.out;

4，需要额外功能吗？

因为处理的数据是文本数据，同时确定是字符流体系。 为方便操作字符数据的可以将源和目的都转成字符流。使用转换流。 为了提高效率，使用Buffer

BufferedReader bufr =new BufferedReader(new InputStreamReader(Systme.in));

BufferedWriter bufw = new BufferedWriter(new OutputStreamWriter(System.out));

5，读取一个文本文件，将文件按照指定的编码表UTF-8进行存储，保存到另一个文件中。

1，明确体系：

源：InputStream ，Reader 目的：OutputStream ，Writer

2，明确数据：

源：是纯文本吗？是 Reader

目的；是纯文本吗？是 Writer

3，明确设备：

源：硬盘：FileReader. 目的：硬盘：FileWriter

FileReader fr = new FileReader("a.txt"); FileWriter fw = new FileWriter("b.txt");

4，额外功能：

注意：目的中虽然是一个文件，但是需要指定编码表。 而直接操作文本文件的FileWriter本身内置的是本地默认码表。无法明确具体指定码表。

这时就需要转换功能。OutputStreamWriter,而这个转换流需要接受一个字节输出流，而且 对应的目的是一个文件。这时就使用字节输出流中的操作文件的流对象。FileOutputStream. FileReader fr = new FileReader("a.txt");

OutputStreamWriter osw = new OutputStreamWriter(new FileOutputStream("b.txt"),"UTF-8");

需要高效吗？

BufferedReader bufr = new BufferedReader(new FileReader("a.txt")); BufferedWriter bufw =

new BufferedWriter(new OutputStreamWriter(new FileOutputStream("b.txt"),"UTF-8"));

目前为止，10个流对象重点掌握。 字符流：

FileReader FileWriter

BufferedReader BufferedWriter

InputStreamReader OutputStreamWrier

字节流：

FileInputStream FileOutputStream

BufferedInputStream BufferedOutputStream

--------------------------------------------------------------------------------

File类：

用于将文件和文件夹封装成对象。

1，创建。

boolean createNewFile():如果该文件不存在，会创建，如果已存在，则不创建。不会像输出流 一样会覆盖。

boolean mkdir(); boolean mkdirs();

2，删除。

boolean delete(); void deleteOnExit();

3，获取：

String getAbsolutePath(); String getPath();

String getParent();

String getName(); long length();

long lastModified();

4，判断：

boolean exists(); boolean isFile(); boolean isDirectory();

5，

--------------------------------------------------------------------------------

IO中的其他功能流对象：

1,打印流：

PrintStream：字节打印流。 特点：

1，构造函数接收File对象，字符串路径，字节输出流。意味着打印目的可以有很多。

2，该对象具备特有的方法 打印方法 print println,可以打印任何类型的数据。

3，特有的print方法可以保持任意类型数据表现形式的原样性，将数据输出到目的地。 对于OutputStream父类中的write，是将数据的最低字节写出去。

PrintWriter：字符打印流。 特点：

1，当操作的数据是字符时，可以选择PrintWriter，比PrintStream要方便。 2，它的构造函数可以接收 File对象，字符串路径，字节输出流，字符输出流。

3，构造函数中，如果参数是输出流，那么可以通过指定另一个参数true完成自动刷新，该true对 println方法有效。

什么时候用？ 当需要保证数据表现的原样性时，就可以使用打印流的打印方法来完成，这样更为方便。 保证原样性的原理：其实就是将数据变成字符串，在进行写入操作。

SequenceInputStream:

特点： 1，将多个字节读取流和并成一个读取流，将多个源合并成一个源，操作起来方便。 2，需要的枚举接口可以通过Collections.enumeration(collection);

ObjectInputStream 和 ObjectOutputStream

对象的序列化和反序列化。

writeObject readObject

Serializable标记接口

关键字：transient

RandomAccessFile:

特点：

1，即可读取，又可以写入。

2，内部维护了一个大型的byte数组，通过对数组的操作完成读取和写入。 3，通过getFilePointer方法获取指针的位置，还可以通过seek方法设置指针的位置。 4，该对象的内容应该封装了字节输入流和字节输出流。

5，该对象只能操作文件。

通过seek方法操作指针，可以从这个数组中的任意位置上进行读和写 可以完成对数据的修改。

但是要注意：数据必须有规律。

管道流：需要和多线程技术相结合的流对象。 PipedOutputStream PipedInputStream

用操作基本数据类型值的对象： DataInputStream DataOutputStream

设备是内存的流对象。

ByteArrayInputStream ByteArrayOutputStream CharArrayReader CharArrayWriter

--------------------------------------------------------------------------------

IO流体系：

字符流：

Reader

|--BufferedReader:

|--LineNumberReader

|--CharArrayReader

|--StringReader

|--InputStreamReaer

|--FileReader

Writer

|--BufferedWriter

|--CharArrayWriter

|--StringWriter

|--OutputStreamWriter

|--FileWriter

|--PrintWriter

字节流：

InputStream

|--FileInputStream:

|--FilterInputStream

|--BufferedInputStream

|--DataInputStream

|--ByteArrayInputStream

|--ObjectInputStream

|--SequenceInputStream

|--PipedInputStream

OutputStream

|--FileOutputStream

|--FilterOutputStream

|--BufferedOutputStream

|--DataOutputStream

|--ByteArrayOutputStream

|--ObjectOutputStream

|--PipedOutputStream

|--PrintStream

# 第九章：GUI编程

176.GUI简单演示：

package cn.itcast.gui.p1.awt;

import java.awt.Button; import java.awt.FlowLayout; import java.awt.Frame;

import java.awt.event.ActionEvent; import java.awt.event.ActionListener; import java.awt.event.WindowAdapter; import java.awt.event.WindowEvent;

public class FrameDemo {

/\*\*

\* @param args

\*/

public static void main(String[] args) {

Frame f = new Frame("my frame");

// f.setSize(500, 400);

// f.setLocation(400, 200);

f.setBounds(400, 200, 500, 400); f.setLayout(new FlowLayout());//设置流式布局

Button but = new Button("一个按钮"); f.add(but);//将按钮添加到窗体中。 f.addWindowListener(new WindowAdapter() {

@Override

public void windowClosing(WindowEvent e) {

// System.out.println("closing......."+e); System.exit(0);

}

});

//在按钮上加上一个监听。 but.addActionListener(new ActionListener() {

@Override

public void actionPerformed(ActionEvent e) {

// System.out.println("button run ....."); System.exit(0);

}

});

f.setVisible(true); System.out.println("over");

}

}

177.事件监听机制及使用案例：

package cn.itcast.gui.p1.awt;

import java.awt.Button; import java.awt.FlowLayout; import java.awt.Frame; import java.awt.TextField;

import java.awt.event.ActionEvent; import java.awt.event.ActionListener; import java.awt.event.KeyAdapter; import java.awt.event.KeyEvent; import java.awt.event.MouseAdapter;

import java.awt.event.MouseEvent; import java.awt.event.WindowAdapter; import java.awt.event.WindowEvent;

public class MouseAndKeyDemo {

private Frame f; private TextField tf; private Button but;

public MouseAndKeyDemo() { init();

}

private void init() {

f = new Frame("演示鼠标和键盘监听"); f.setBounds(400,200,500,400); f.setLayout(new FlowLayout());

tf = new TextField(35);

but = new Button("一个按钮");

f.add(tf);

f.add(but); myEvent(); f.setVisible(true);

}

private void myEvent() {

//给文本框添加键盘监听。 tf.addKeyListener(new KeyAdapter() {

@Override

public void keyPressed(KeyEvent e) {

// System.out.println("key run..."+KeyEvent.getKeyText(e.getKeyCode())+"::::"+e.getKeyCode());

// int code = e.getKeyCode();

// if(!(code>=KeyEvent.VK\_0 && code<=KeyEvent.VK\_9)){

// System.out.println("必须是数字");

// e.consume();

// }

if(e.isControlDown() && e.getKeyCode()==KeyEvent.VK\_ENTER){ System.out.println("enter run ...");

}

}

});

f.addWindowListener(new WindowAdapter() {

@Override

public void windowClosing(WindowEvent e) {

System.exit(0);

}

});

but.addActionListener(new ActionListener() {

@Override

public void actionPerformed(ActionEvent e) {

// System.out.println("action run.....");

}

});

//在按钮上添加一个鼠标监听. but.addMouseListener(new MouseAdapter() {

private int count = 1;

@Override

public void mouseEntered(MouseEvent e) {

// System.out.println("mouse enter..."+count++);

// tf.setText("mouse enter..."+count++);

}

@Override

public void mouseClicked(MouseEvent e) {

if(e.getClickCount()==2)

tf.setText("mouse double click..."+count++);

// System.out.println("mouse click..."+count++);

// System.out.println(e);

}

});

}

/\*\*

\* @param args

\*/

public static void main(String[] args) {

new MouseAndKeyDemo();

}

}

# 第十章：网络编程

178.IP简述使用：

package cn.itcast.net.p1.ip;

import java.net.InetAddress;

import java.net.UnknownHostException;

public class IPDemo {

/\*\*

* @param args
* @throws UnknownHostException

\*/

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

//获取本地主机ip地址对象。

InetAddress ip = InetAddress.getLocalHost();

//获取其他主机的ip地址对象。

ip = InetAddress.getByName("192.168.1.110");

//InetAddress.getByName("my\_think");

System.out.println(ip.getHostAddress()); System.out.println(ip.getHostName());

}

}

179.使用UDP建立信息収送端：

package cn.itcast.net.p2.udp;

import java.io.IOException; import java.net.DatagramPacket; import java.net.DatagramSocket; import java.net.InetAddress; import java.net.SocketException;

import java.net.UnknownHostException; public class UDPSendDemo {

/\*\*

* @param args
* @throws IOException

\*/

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

System.out.println("发送端启动......");

/\*

\* 创建UDP传输的发送端。

\* 思路：

\* 1，建立udp的socket服务。

\* 2，将要发送的数据封装到数据包中。

\* 3，通过udp的socket服务将数据包发送出去。

\* 4，关闭socket服务。

\*/

//1,udpsocket服务。使用DatagramSocket对象。 DatagramSocket ds = new DatagramSocket(8888);

//2,将要发送的数据封装到数据包中。

String str = "udp传输演示：哥们来了！";

//使用DatagramPacket将数据封装到的该对象包中。 byte[] buf = str.getBytes();

DatagramPacket dp =

new DatagramPacket(buf,buf.length,InetAddress.getByName("192.168.1.100"),10000);

//3，通过udp的socket服务将数据包发送出去。使用send方法。 ds.send(dp);

//4，关闭资源。 ds.close();

}

}

180.使用UDP建立信息接收端：

package cn.itcast.net.p2.udp;

import java.io.IOException; import java.net.DatagramPacket; import java.net.DatagramSocket;

public class UDPReceDemo {

/\*\*

* @param args
* @throws IOException

\*/

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

System.out.println("接收端启动......");

/\*

\* 建立UDP接收端的思路。

\* 1，建立udp socket服务,因为是要接收数据，必须要明确一个端口号。

\* 2，创建数据包，用于存储接收到的数据。方便用数据包对象的方法解析这些数据.

\* 3，使用socket服务的receive方法将接收的数据存储到数据包中。

\* 4，通过数据包的方法解析数据包中的数据。

\* 5，关闭资源

\*/

//1,建立udp socket服务。

DatagramSocket ds = new DatagramSocket(10000);

//2,创建数据包。

byte[] buf = new byte[1024];

DatagramPacket dp = new DatagramPacket(buf,buf.length);

//3,使用接收方法将数据存储到数据包中。 ds.receive(dp);//阻塞式的。

//4，通过数据包对象的方法，解析其中的数据,比如，地址，端口，数据内容。 String ip = dp.getAddress().getHostAddress();

int port = dp.getPort();

String text = new String(dp.getData(),0,dp.getLength()); System.out.println(ip+":"+port+":"+text);

//5,关闭资源。

ds.close();

}

}

181.使用UDP建立信息交互方式：

（信息发送端）

package cn.itcast.net.p2.udp;

import java.io.BufferedReader; import java.io.IOException; import java.io.InputStreamReader; import java.net.DatagramPacket; import java.net.DatagramSocket; import java.net.InetAddress;

public class UDPSendDemo2 {

/\*\*

* @param args
* @throws IOException

\*/

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

System.out.println("发送端启动......");

/\*

\* 创建UDP传输的发送端。

\* 思路：

\* 1，建立udp的socket服务。

\* 2，将要发送的数据封装到数据包中。

\* 3，通过udp的socket服务将数据包发送出去。

\* 4，关闭socket服务。

\*/

//1,udpsocket服务。使用DatagramSocket对象。 DatagramSocket ds = new DatagramSocket(8888);

// String str = "udp传输演示：哥们来了！";

BufferedReader bufr = new BufferedReader(new InputStreamReader(System.in)); String line = null;

while((line=bufr.readLine())!=null){

byte[] buf = line.getBytes(); DatagramPacket dp =

new DatagramPacket(buf,buf.length,InetAddress.getByName("192.168.1.100"),10000);

ds.send(dp);

if("886".equals(line)) break;

}

//4，关闭资源。 ds.close();

}

}

（信息接收端）

package cn.itcast.net.p2.udp;

import java.io.IOException; import java.net.DatagramPacket; import java.net.DatagramSocket;

public class UDPReceDemo2 {

/\*\*

* @param args
* @throws IOException

\*/

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

System.out.println("接收端启动......");

/\*

\* 建立UDP接收端的思路。

\* 1，建立udp socket服务,因为是要接收数据，必须要明确一个端口号。

\* 2，创建数据包，用于存储接收到的数据。方便用数据包对象的方法解析这些数据.

\* 3，使用socket服务的receive方法将接收的数据存储到数据包中。

\* 4，通过数据包的方法解析数据包中的数据。

\* 5，关闭资源

\*/

//1,建立udp socket服务。

DatagramSocket ds = new DatagramSocket(10000); while(true){

//2,创建数据包。

byte[] buf = new byte[1024];

DatagramPacket dp = new DatagramPacket(buf,buf.length);

//3,使用接收方法将数据存储到数据包中。 ds.receive(dp);//阻塞式的。

//4，通过数据包对象的方法，解析其中的数据,比如，地址，端口，数据内容。 String ip = dp.getAddress().getHostAddress();

int port = dp.getPort();

String text = new String(dp.getData(),0,dp.getLength()); System.out.println(ip+":"+port+":"+text);

}

//5,关闭资源。

// ds.close();

}

}

182.聊天程序（多线程）：

（发送端）

package cn.itcast.net.p3.chat;

import java.io.BufferedReader; import java.io.InputStreamReader; import java.net.DatagramPacket; import java.net.DatagramSocket; import java.net.InetAddress;

public class Send implements Runnable { private DatagramSocket ds;

public Send(DatagramSocket ds){ this.ds = ds;

}

@Override

public void run() {

try {

BufferedReader bufr = new BufferedReader(new InputStreamReader(System.in));

String line = null; while((line=bufr.readLine())!=null){

byte[] buf = line.getBytes(); DatagramPacket dp =

new DatagramPacket(buf,buf.length,InetAddress.getByName("192.168.1.255"),10001);

ds.send(dp);

if("886".equals(line)) break;

}

ds.close();

} catch (Exception e) {

}

}

}

（接收端）

package cn.itcast.net.p3.chat;

import java.net.DatagramPacket; import java.net.DatagramSocket;

public class Rece implements Runnable { private DatagramSocket ds;

public Rece(DatagramSocket ds) { this.ds = ds;

}

@Override

public void run() { try {

while (true) {

// 2,创建数据包。

byte[] buf = new byte[1024];

DatagramPacket dp = new DatagramPacket(buf, buf.length);

// 3,使用接收方法将数据存储到数据包中。 ds.receive(dp);// 阻塞式的。

// 4，通过数据包对象的方法，解析其中的数据,比如，地址，端口，数据内容。 String ip = dp.getAddress().getHostAddress();

int port = dp.getPort();

String text = new String(dp.getData(), 0, dp.getLength());

System.out.println(ip + "::" + text); if(text.equals("886")){

System.out.println(ip+"....退出聊天室");

}

}

} catch (Exception e) {

}

}

}

（开启发送和接收两个线程开始运行聊天）

**package** cn.itcast.net.p3.chat;

**import** java.io.IOException; **import** java.net.DatagramSocket; **import** java.net.SocketException;

**public class** ChatDemo {

/\*\*

* **@param** args
* **@throws** IOException

\*/

**public static void** main(String[] args) **throws** IOException { DatagramSocket send = **new** DatagramSocket();

DatagramSocket rece = **new** DatagramSocket(10001);

**new** Thread(**new** Send(send)).start();

**new** Thread(**new** Rece(rece)).start();

}

}

183.使用TCP建立客户端：

package cn.itcast.net.p4.tcp;

import java.io.IOException; import java.io.OutputStream; import java.net.Socket;

import java.net.UnknownHostException; public class ClientDemo {

/\*\*

* @param args
* @throws IOException
* @throws UnknownHostException

\*/

public static void main(String[] args) throws UnknownHostException, IOException

{

//客户端发数据到服务端

/\*

\* Tcp传输，客户端建立的过程。

\* 1，创建tcp客户端socket服务。使用的是Socket对象。

\* 建议该对象一创建就明确目的地。要连接的主机。

\* 2，如果连接建立成功，说明数据传输通道已建立。

\* 该通道就是socket流 ,是底层建立好的。 既然是流，说明这里既有输入，又有输出。

\* 想要输入或者输出流对象，可以找Socket来获取。

\* 可以通过getOutputStream(),和getInputStream()来获取两个字节流。

\* 3，使用输出流，将数据写出。

\* 4，关闭资源。

\*/

//创建客户端socket服务。

Socket socket = new Socket("192.168.1.100",10002);

//获取socket流中的输出流。

OutputStream out = socket.getOutputStream();

//使用输出流将指定的数据写出去。 out.write("tcp演示：哥们又来了!".getBytes());

//关闭资源。 socket.close();

}

}

184.使用TCP建立服务端：

package cn.itcast.net.p4.tcp;

import java.io.IOException; import java.io.InputStream; import java.io.OutputStream; import java.net.ServerSocket; import java.net.Socket;

public class ServerDemo {

/\*\*

* @param args
* @throws IOException

\*/

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

// 服务端接收客户端发送过来的数据，并打印在控制台上。

/\*

\* 建立tcp服务端的思路：

\* 1，创建服务端socket服务。通过ServerSocket对象。

\* 2，服务端必须对外提供一个端口，否则客户端无法连接。

\* 3，获取连接过来的客户端对象。

\* 4，通过客户端对象获取socket流读取客户端发来的数据

\* 并打印在控制台上。

\* 5，关闭资源。关客户端，关服务端。

\*/

//1创建服务端对象。

ServerSocket ss = new ServerSocket(10002);

//2,获取连接过来的客户端对象。 Socket s = ss.accept();//阻塞式.

String ip = s.getInetAddress().getHostAddress();

//3，通过socket对象获取输入流，要读取客户端发来的数据 InputStream in = s.getInputStream();

byte[] buf = new byte[1024]; int len = in.read(buf);

String text = new String(buf,0,len); System.out.println(ip+":"+text);

s.close();

ss.close();

}

}

185.使用TCP建立交互方式：

（客户端）

package cn.itcast.net.p4.tcp;

import java.io.IOException; import java.io.InputStream; import java.io.OutputStream; import java.net.Socket;

import java.net.UnknownHostException; public class ClientDemo2 {

/\*\*

* @param args
* @throws IOException
* @throws UnknownHostException

\*/

public static void main(String[] args) throws UnknownHostException, IOException

{

//客户端发数据到服务端

/\*

\* Tcp传输，客户端建立的过程。

\* 1，创建tcp客户端socket服务。使用的是Socket对象。

\* 建议该对象一创建就明确目的地。要连接的主机。

\* 2，如果连接建立成功，说明数据传输通道已建立。

\* 该通道就是socket流 ,是底层建立好的。 既然是流，说明这里既有输入，又有输出。

\* 想要输入或者输出流对象，可以找Socket来获取。

\* 可以通过getOutputStream(),和getInputStream()来获取两个字节流。

\* 3，使用输出流，将数据写出。

\* 4，关闭资源。

\*/

Socket socket = new Socket("192.168.1.100",10002); OutputStream out = socket.getOutputStream();

out.write("tcp演示：哥们又来了!".getBytes());

//读取服务端返回的数据,使用socket读取流。 InputStream in = socket.getInputStream();

byte[] buf = new byte[1024]; int len = in.read(buf);

String text = new String(buf,0,len);

System.out.println(text);

//关闭资源。 socket.close();

}

}

（服务端）

package cn.itcast.net.p4.tcp;

import java.io.IOException; import java.io.InputStream; import java.io.OutputStream; import java.net.ServerSocket; import java.net.Socket;

public class ServerDemo2 {

/\*\*

* @param args
* @throws IOException

\*/

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

// 服务端接收客户端发送过来的数据，并打印在控制台上。

/\*

\* 建立tcp服务端的思路：

\* 1，创建服务端socket服务。通过ServerSocket对象。

\* 2，服务端必须对外提供一个端口，否则客户端无法连接。

\* 3，获取连接过来的客户端对象。

\* 4，通过客户端对象获取socket流读取客户端发来的数据

\* 并打印在控制台上。

\* 5，关闭资源。关客户端，关服务端。

\*/

//1创建服务端对象。

ServerSocket ss = new ServerSocket(10002);

//2,获取连接过来的客户端对象。 Socket s = ss.accept();

String ip = s.getInetAddress().getHostAddress();

//3，通过socket对象获取输入流，要读取客户端发来的数据 InputStream in = s.getInputStream();

byte[] buf = new byte[1024]; int len = in.read(buf);

String text = new String(buf,0,len); System.out.println(ip+":"+text);

//使用客户端socket对象的输出流给客户端返回数据 OutputStream out = s.getOutputStream(); out.write("收到".getBytes());

s.close();

ss.close();

}

}

# 186.用TCP创建一个英文大写服务器（练习）：

（需求与分析）

**package** cn.itcast.net.p5.tcptest;

**public class** TextTransTest {

/\*\*

\* **@param** args

\*/

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

/\*

\* 客户端输入字母数据，发送给服务端，

\* 服务端收到后显示在控制台，并将该数据转成大写返回给客户端

\* 直到客户端输入over.转换结束.

\*

\* 创建一个英文大写转换服务器.

\*

\* 分析:

\* 有客户端和服务端,使用tcp传输

\*/

}

}

（客户端）

package cn.itcast.net.p5.tcptest;

import java.io.BufferedReader; import java.io.IOException; import java.io.InputStreamReader; import java.io.PrintWriter; import java.net.Socket;

import java.net.UnknownHostException; public class TransClient {

/\*\*

* @param args
* @throws IOException
* @throws UnknownHostException

\*/

public static void main(String[] args) throws UnknownHostException, IOException

{

/\*

\* 思路：

\* 客户端：

\* 1，需要先有socket端点。

\* 2，客户端的数据源：键盘。

\* 3，客户端的目的：socket.

\* 4，接收服务端的数据，源：socket。

\* 5，将数据显示在打印出来：目的：控制台.

\* 6，在这些流中操作的数据，都是文本数据。

\* 转换客户端:

\* 1,创建socket客户端对象。

\* 2,获取键盘录入。

\* 3，将录入的信息发送给socket输出流。

\*/

//1,创建socket客户端对象。

Socket s = new Socket("192.168.1.100",10004);

//2，获取键盘录入。 BufferedReader bufr =

new BufferedReader(new InputStreamReader(System.in));

//3,socket输出流。

// new BufferedWriter(new OutputStreamWriter(s.getOutputStream())); PrintWriter out = new PrintWriter(s.getOutputStream(),true);

//4,socket输入流，读取服务端返回的大写数据

BufferedReader bufIn = new BufferedReader(new InputStreamReader(s.getInputStream()));

String line = null; while((line=bufr.readLine())!=null){

if("over".equals(line)) break;

// out.print(line+"\r\n");

// out.flush(); out.println(line);

//读取服务端发回的一行大写数。

String upperStr = bufIn.readLine(); System.out.println(upperStr);

}

s.close();

}

}

（服务端）

package cn.itcast.net.p5.tcptest;

import java.io.BufferedReader;

import java.io.IOException; import java.io.InputStreamReader; import java.io.PrintWriter; import java.net.ServerSocket; import java.net.Socket;

public class TransServer {

/\*\*

* @param args
* @throws IOException

\*/

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

/\*

\* 转换服务端。

\* 分析：

\* 1，serversocket服务。

\* 2，获取socket对象。

\* 3，源：socket，读取客户端发过来的需要转换的数据。

\* 4，目的：显示在控制台上。

\* 5，将数据转成大写发给客户端。

\*/

//1,

ServerSocket ss = new ServerSocket(10004);

//2,获取socket对象。 Socket s = ss.accept();

//获取ip.

String ip = s.getInetAddress().getHostAddress(); System.out.println(ip+"......connected");

//3,获取socket读取流，并装饰。

BufferedReader bufIn = new BufferedReader(new InputStreamReader(s.getInputStream()));

//4,获取socket的输出流，并装饰。

PrintWriter out = new PrintWriter(s.getOutputStream(),true);

String line = null; while((line=bufIn.readLine())!=null){

System.out.println(line); out.println(line.toUpperCase());

// out.print(line.toUpperCase()+"\r\n");

// out.flush();

}

s.close();

ss.close();

}

}

187.上传的客户端不服务端：

（客户端）

package cn.itcast.net.p6.uploadtext;

import java.io.BufferedReader; import java.io.File;

import java.io.FileReader; import java.io.IOException; import java.io.InputStreamReader; import java.io.PrintWriter; import java.net.Socket;

import java.net.UnknownHostException; public class UploadClient {

/\*\*

* @param args
* @throws IOException
* @throws UnknownHostException

\*/

public static void main(String[] args) throws UnknownHostException, IOException

{

System.out.println("上传客户端。。。。。。");

File file = new File("c:\\client.txt"); System.out.println(file.exists());

//1

Socket s = new Socket("192.168.1.100",10005);

//2

BufferedReader bufr =

new BufferedReader(new FileReader(file));

//3,

PrintWriter out = new PrintWriter(s.getOutputStream(),true);

String line = null; while((line=bufr.readLine())!=null){

out.println(line);

}

//告诉服务端，客户端写完了。 s.shutdownOutput();

// out.println("!@#$%^&\*(");

//4,

BufferedReader bufIn = new BufferedReader(new InputStreamReader(s.getInputStream()));

String str = bufIn.readLine(); System.out.println(str);

bufr.close();

s.close();

}

}

（服务端）

package cn.itcast.net.p6.uploadtext;

import java.io.BufferedReader; import java.io.BufferedWriter; import java.io.FileWriter; import java.io.IOException; import java.io.InputStreamReader; import java.io.PrintWriter; import java.net.ServerSocket; import java.net.Socket;

public class UploadServer {

/\*\*

* @param args
* @throws IOException

\*/

public static void main(String[] args) throws IOException { System.out.println("上传服务端。。。。。。。。。");

//1

ServerSocket ss = new ServerSocket(10005);

//2,

Socket s = ss.accept(); System.out.println(s.getInetAddress().getHostAddress()+".....connected");

//3,

BufferedReader bufIn = new BufferedReader(new InputStreamReader(s.getInputStream()));

//4,

BufferedWriter bufw = new BufferedWriter(new FileWriter("c:\\server.txt")); String line = null;

while((line=bufIn.readLine())!=null){

// if("over".equals(line))

// break;

bufw.write(line); bufw.newLine(); bufw.flush();

}

PrintWriter out = new PrintWriter(s.getOutputStream(),true); out.println("上传成功");

bufw.close();

s.close();

ss.close();

}

}

188.上传图片的设计案例：

（上传的任务）

package cn.itcast.net.p1.uploadpic;

import java.io.File;

import java.io.FileOutputStream; import java.io.IOException; import java.io.InputStream; import java.io.OutputStream; import java.net.Socket;

public class UploadTask implements Runnable {

private static final int SIZE = 1024\*1024\*2;

private Socket s;

public UploadTask(Socket s) { this.s = s;

}

@Override

public void run() {

int count = 0;

String ip = s.getInetAddress().getHostAddress(); System.out.println(ip + ".....connected");

try{

// 读取客户端发来的数据。

InputStream in = s.getInputStream();

// 将读取到数据存储到一个文件中。 File dir = new File("c:\\pic"); if (!dir.exists()) {

dir.mkdirs();

}

File file = new File(dir, ip + ".jpg");

//如果文件已经存在于服务端 while(file.exists()){

file = new File(dir,ip+"("+(++count)+").jpg");

}

FileOutputStream fos = new FileOutputStream(file); byte[] buf = new byte[1024];

int len = 0;

while ((len = in.read(buf)) != -1) {

fos.write(buf, 0, len); if(file.length()>SIZE){

System.out.println(ip+"文件体积过大");

fos.close();

s.close();

System.out.println(ip+"...."+file.delete());

return ;

}

}

// 获取socket输出流，将上传成功字样发给客户端。 OutputStream out = s.getOutputStream(); out.write("上传成功".getBytes());

fos.close();

s.close();

}catch(IOException e){

}

}

}

（上传的客户端）

package cn.itcast.net.p1.uploadpic;

import java.io.FileInputStream; import java.io.IOException; import java.io.InputStream; import java.io.OutputStream; import java.net.Socket;

import java.net.UnknownHostException; public class UploadPicClient {

/\*\*

* @param args
* @throws IOException
* @throws UnknownHostException

\*/

public static void main(String[] args) throws UnknownHostException, IOException

{

//1,创建客户端socket。

Socket s = new Socket("192.168.1.100",10006);

//2,读取客户端要上传的图片文件。

FileInputStream fis = new FileInputStream("c:\\0.bmp");

//3,获取socket输出流，将读到图片数据发送给服务端。 OutputStream out = s.getOutputStream();

byte[] buf = new byte[1024];

int len = 0;

while((len=fis.read(buf))!=-1){ out.write(buf,0,len);

}

//告诉服务端说：这边的数据发送完毕。让服务端停止读取。 s.shutdownOutput();

//读取服务端发回的内容。

InputStream in = s.getInputStream(); byte[] bufIn = new byte[1024];

int lenIn = in.read(buf);

String text = new String(buf,0,lenIn); System.out.println(text);

fis.close();

s.close();

}

}

（上传的服务端）

package cn.itcast.net.p1.uploadpic;

import java.io.File;

import java.io.FileOutputStream; import java.io.IOException; import java.io.InputStream; import java.io.OutputStream; import java.net.ServerSocket; import java.net.Socket;

public class UploadPicServer {

/\*\*

* @param args
* @throws IOException

\*/

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

//创建tcp的socket服务端。

ServerSocket ss = new ServerSocket(10006);

while(true){

Socket s = ss.accept();

new Thread(new UploadTask(s)).start();

}

//获取客户端。

// ss.close();

}

}

189.模拟IE浏览器：

package cn.itcast.net.p2.ie\_server;

import java.io.IOException; import java.io.InputStream; import java.io.PrintWriter; import java.net.Socket;

import java.net.UnknownHostException; public class MyBrowser {

/\*\*

* @param args
* @throws IOException
* @throws UnknownHostException

\*/

public static void main(String[] args) throws UnknownHostException, IOException

{

Socket s = new Socket("192.168.1.100",8080);

//模拟浏览器，给tomcat服务端发送符合http协议的请求消息。

PrintWriter out = new PrintWriter(s.getOutputStream(),true); out.println("GET /myweb/1.html HTTP/1.1"); out.println("Accept: \*/\*");

out.println("Host: 192.168.1.100:8080"); out.println("Connection: close"); out.println();

out.println();

InputStream in = s.getInputStream(); byte[] buf = new byte[1024];

int len = in.read(buf);

String str =new String(buf,0,len); System.out.println(str);

s.close();

//http://192.168.1.100:8080/myweb/1.html

}

}

190.模拟Tomcat服务器：

package cn.itcast.net.p2.ie\_server;

import java.io.IOException; import java.io.InputStream; import java.io.PrintWriter; import java.net.ServerSocket; import java.net.Socket;

public class MyTomcat {

/\*\*

* @param args
* @throws IOException

\*/

public static void main(String[] args) throws IOException { ServerSocket ss = new ServerSocket(9090);

Socket s = ss.accept(); System.out.println(s.getInetAddress().getHostAddress()+".....connected");

InputStream in = s.getInputStream(); byte[] buf = new byte[1024];

int len = in.read(buf);

String text = new String(buf,0,len); System.out.println(text);

//给客户端一个反馈信息。

PrintWriter out = new PrintWriter(s.getOutputStream(),true);

out.println("<font color='red' size='7'>欢迎光临</font>");

s.close();

ss.close();

}

}

191.URL使用演示：

package cn.itcast.net.p2.ie\_server;

import java.io.IOException; import java.io.InputStream; import java.net.URL;

import java.net.URLConnection; public class URLDemo {

/\*\*

* @param args
* @throws IOException

\*/

public static void main(String[] args) throws IOException { String str\_url = "http://192.168.1.100:8080/myweb/1.html"; URL url = new URL(str\_url);

// System.out.println("getProtocol:"+url.getProtocol());

// System.out.println("getHost:"+url.getHost());

// System.out.println("getPort:"+url.getPort());

// System.out.println("getFile:"+url.getFile());

// System.out.println("getPath:"+url.getPath());

// System.out.println("getQuery:"+url.getQuery());

// InputStream in = url.openStream();

//获取url对象的Url连接器对象。将连接封装成了对象:java中内置的可以解析的具体协议的对象

+socket.

URLConnection conn = url.openConnection();

// String value = conn.getHeaderField("Content-Type");

// System.out.println(value);

// System.out.println(conn);

//sun.net.www.protocol.http.HttpURLConnection:http://192.168.1.100:8080/myweb

/1.html

InputStream in = conn.getInputStream(); byte[] buf = new byte[1024];

int len = in.read(buf);

String text = new String(buf,0,len); System.out.println(text);

in.close();

}

}

192.网络编程小结：

最常见的客户端： 浏览器 ：IE。

最常见的服务端：

服务器：Tomcat。 为了了解其原理：

自定义服务端，使用已有的客户端IE，了解一下客户端给服务端发了什么请求？ 发送的请求是：

GET / HTTP/1.1 请求行 请求方式 /myweb/1.html 请求的资源路径 http协议版本。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 请求消息头 | . 属性名：属性值 |  | | |
| Accept: | image/gif, | image/x-xbitmap, | image/jpeg, | image/pjpeg, |

application/x-shockwave-flash,

application/vnd.ms-excel, application/vnd.ms-powerpoint, application/msword, \*/\* Accept: \*/\*

Accept-Language: zh-cn,zu;q=0.5 Accept-Encoding: gzip, deflate

User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; InfoPath.2) Host: 192.168.1.100:9090

//Host: www.huyouni.com:9090 Connection: Keep-Alive

//空行

//请求体。

//服务端发回应答消息。

HTTP/1.1 200 OK //应答行，http的协议版本 应答状态码 应答状态描述信息 应答消息属性信息。 属性名：属性值

Server: Apache-Coyote/1.1 ETag: W/"199-1323480176984"

Last-Modified: Sat, 10 Dec 2011 01:22:56 GMT Content-Type: text/html

Content-Length: 199

Date: Fri, 11 May 2012 07:51:39 GMT

Connection: close

//空行

//应答体。

<html>

<head>

<title>这是我的网页</title>

</head>

<body>

<h1>欢迎光临</h1>

<font size='5' color="red">这是一个tomcat服务器中的资源。是一个html网页。</font>

</body>

</html>

--------------------------------------------------------------------------------

网络结构，

1,C/S client/server

特点：

该结构的软件，客户端和服务端都需要编写。 可发成本较高，维护较为麻烦。

好处：

客户端在本地可以分担一部分运算。

2,B/S browser/server

特点：

该结构的软件，只开发服务器端，不开发客户端，因为客户端直接由浏览器取代。 开发成本相对低，维护更为简单。

缺点：所有运算都要在服务端完成。

# 第十一章：反射机制

193.反射简述及三种获叏Class对象的方法：

**package** cn.itcast.reflect.demo;

**import** cn.itcast.bean.Person;

/\*

\* JAVA反射机制是在运行状态中，对于任意一个类 (class文件)，都能够知道这个类的所有属性和方法；

\* 对于任意一个对象，都能够调用它的任意一个方法和属性；

\* 这种动态获取的信息以及动态调用对象的方法的功能称为java语言的反射机制。

\*

\*

\* 动态获取类中信息，就是java反射 。

\* 可以理解为对类的解剖。

\*

\*

\* 要想要对字节码文件进行解剖，必须要有字节码文件对象.

\* 如何获取字节码文件对象呢？

\*

\*/

**public class** ReflectDemo {

/\*\*

* **@param** args
* **@throws** ClassNotFoundException

\*/

**public static void** main(String[] args) **throws** ClassNotFoundException {

*getClassObject\_3*();

}

/\*

\* 方式三：

\* 只要通过给定的类的 字符串名称就可以获取该类，更为扩展。

\* 可是用Class类中的方法完成。

\* 该方法就是forName.

\* 这种方式只要有名称即可，更为方便，扩展性更强。

\*/

**public static void** getClassObject\_3() **throws** ClassNotFoundException { String className = "cn.itcast.bean.Person";

Class clazz = Class.*forName*(className);

System.*out*.println(clazz);

}

/\*

\* 方式二：

\* 2，任何数据类型都具备一个静态的属性.class来获取其对应的Class对象。

\* 相对简单，但是还是要明确用到类中的静态成员。

\* 还是不够扩展。

\*

\*/

**public static void** getClassObject\_2() { Class clazz = Person.**class**;

Class clazz1 = Person.**class**; System.*out*.println(clazz==clazz1);

}

/\*

\* 获取字节码对象的方式：

\* 1，Object类中的getClass()方法的。

\* 想要用这种方式，必须要明确具体的类，并创建对象。

\* 麻烦 .

\*

\*/

**public static void** getClassObject\_1(){

Person p = **new** Person(); Class clazz = p.getClass();

Person p1 = **new** Person(); Class clazz1 = p1.getClass();

System.*out*.println(clazz==clazz1);

}

}

194.利用反射获叏构造凼数：

package cn.itcast.reflect.demo;

import java.io.FileReader;

import java.lang.reflect.Constructor; public class ReflectDemo2 {

/\*\*

* @param args
* @throws Exception
* @throws InstantiationException
* @throws ClassNotFoundException

\*/

public static void main(String[] args) throws ClassNotFoundException, InstantiationException, Exception {

createNewObject\_2();

}

public static void createNewObject\_2() throws Exception {

// cn.itcast.bean.Person p = new cn.itcast.bean.Person("小强",39);

/\*

\* 当获取指定名称对应类中的所体现的对象时，

\* 而该对象初始化不使用空参数构造该怎么办呢？

\* 既然是通过指定的构造 函数进行对象的初始化，

\* 所以应该先获取到该构造函数。 通过字节码文件对象即可完成。

\* 该方法是：getConstructor(paramterTypes);

\*

\*/

String name = "cn.itcast.bean.Person";

//找寻该名称类文件，并加载进内存，并产生Class对象。 Class clazz = Class.forName(name);

//获取到了指定的构造函数对 象。

Constructor constructor = clazz.getConstructor(String.class,int.class);

//通过该构造器对象的newInstance方法进行对象的初始化。 Object obj = constructor.newInstance("小明",38);

}

public static void createNewObject() throws ClassNotFoundException, InstantiationException, IllegalAccessException{

//早期：new时候，先根据被new的类的名称找寻该类的字节码文件，并加载进内存，

// 并创建该字节码文件对象，并接着创建该字节文件的对应的Person对象.

// cn.itcast.bean.Person p = new cn.itcast.bean.Person();

//现在：

String name = "cn.itcast.bean.Person";

//找寻该名称类文件，并加载进内存，并产生Class对象。 Class clazz = Class.forName(name);

//如何产生该类的对象呢？

Object obj = clazz.newInstance();

}

}

195.利用反射获叏字段：

**package** cn.itcast.reflect.demo; **import** java.lang.reflect.Field; **public class** ReflectDemo3 {

/\*\*

* **@param** args
* **@throws** Exception

\*/

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

*getFieldDemo*();

}

/\*

\* 获取字节码文件中的字段。

\*/

**public static void** getFieldDemo() **throws** Exception { Class clazz = Class.*forName*("cn.itcast.bean.Person");

Field field = **null**;//clazz.getField("age");//只能获取公有的，

field = clazz.getDeclaredField("age");//只获取本类，但包含私有。

//对私有字段的访问取消权限检查。暴力访问。 field.setAccessible(**true**);

Object obj = clazz.newInstance(); field.set(obj, 89);

Object o = field.get(obj); System.*out*.println(o);

// cn.itcast.bean.Person p = new cn.itcast.bean.Person();

// p.age = 30;

}

}

# 196.利用反射获叏Class中的公共凼数：

package cn.itcast.reflect.demo;

import java.lang.reflect.Constructor; import java.lang.reflect.Method;

public class ReflectDemo4 {

public ReflectDemo4() {

}

/\*\*

* @param args
* @throws Exception

\*/

public static void main(String[] args) throws Exception { getMethodDemo\_3();

}

public static void getMethodDemo\_3() throws Exception { Class clazz = Class.forName("cn.itcast.bean.Person");

Method method = clazz.getMethod("paramMethod", String.class,int.class);

Object obj = clazz.newInstance();

method.invoke(obj, "小强",89);

}

public static void getMethodDemo\_2() throws Exception { Class clazz = Class.forName("cn.itcast.bean.Person");

Method method = clazz.getMethod("show", null);//获取空参数一般方法。

// Object obj = clazz.newInstance();

Constructor constructor = clazz.getConstructor(String.class,int.class); Object obj = constructor.newInstance("小明",37);

method.invoke(obj, null);

}

/\*

\* 获取指定Class中的所有公共函数。

\*/

public static void getMethodDemo() throws Exception { Class clazz = Class.forName("cn.itcast.bean.Person");

Method[] methods = clazz.getMethods();//获取的都是公有的方法。

methods = clazz.getDeclaredMethods();//只获取本类中所有方法，包含私有。 for(Method method : methods){

System.out.println(method);

}

}

}

# 第十二章：正则表达式

197.正则表达式简述：

**package** cn.itcast.regex.demo;

/\*

\* 正则表达式。

\*

\* 正则表达式用于操作字符串数据。

\* 通过一些特定的符号来体现的。

\* 所以我们为了掌握正则表达式，必须要学习一些符号。

\*

\* 虽然简化了，但是阅读性差。

\*/

**public class** RegexDemo {

/\*\*

\* **@param** args

\*/

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

String qq = "123k4567";

// checkQQ(qq);

String regex = "[1-9][0-9]{4,14}";//正则表达式。

// boolean b = qq.matches(regex);

// System.out.println(qq+":"+b);

// String str = "aoooooooob";

// String reg = "ao{4,6}b";

// boolean b = str.matches(reg);

// System.out.println(str+":"+b);

}

/\*

\* 需求：定义一个功能对QQ号进行校验。

\* 要求：长度5~15. 只能是数字， 0不能开头

\*/

**public static void** checkQQ(String qq){

**int** len = qq.length();

**if**(len>=5 && len<=15){

**if**(!qq.startsWith("0")){ **try** {

**long** l = Long.*parseLong*(qq);

System.*out*.println(l+":正确");

}**catch**(NumberFormatException e){ System.*out*.println(qq+":含有非法字符");

}

}**else**{

System.*out*.println(qq+":不能0开头");

}

}**else**{

System.*out*.println(qq+":长度错误");

}

}

}

198.正则表达式中常用操作方法演示：

package cn.itcast.regex.demo;

import java.util.regex.Matcher; import java.util.regex.Pattern;

public class RegexDemo2 {

/\*\*

\* @param args

\*/

public static void main(String[] args) {

/\*

\* 正则表达式对字符串的常见操作:

\* 1, 匹配。

* 其实使用的就是String类中的matches方法。

\*

\* 2，切割。

\* 其实使用的就是String类中的split方法。

\*

\* 3，替换。

* 其实使用的就是String类中的replaceAll()方法。

\*

\* 4，获取。

\*

\*/ functionDemo\_4();

}

/\*

\* 获取

\* 将正则规则进行对象的封装。

* Pattern p = Pattern.compile("a\*b");

\* //通过正则对象的matcher方法字符串相关联。获取要对字符串操作的匹配器对象Matcher .

* Matcher m = p.matcher("aaaaab");

\* //通过Matcher匹配器对象的方法对字符串进行操作。

* boolean b = m.matches();

\*/

public static void functionDemo\_4() {

String str = "da jia hao,ming tian bu fang jia!"; String regex = "\\b[a-z]{3}\\b";

//1,将正则封装成对象。

Pattern p = Pattern.compile(regex);

//2, 通过正则对象获取匹配器对象。

Matcher m = p.matcher(str);

//使用Matcher对象的方法对字符串进行操作。

//既然要获取三个字母组成的单词

//查找。 find(); System.out.println(str); while(m.find()){

System.out.println(m.group());//获取匹配的子序列

System.out.println(m.start()+":"+m.end());

}

}

/\*

\* 替换

\*/

public static void functionDemo\_3() {

String str = "zhangsanttttxiaoqiangmmmmmmzhaoliu"; str = str.replaceAll("(.)\\1+", "$1"); System.out.println(str);

String tel = "15800001111";//158\*\*\*\*1111;

tel = tel.replaceAll("(\\d{3})\\d{4}(\\d{4})", "$1\*\*\*\*$2");

System.out.println(tel);

}

/\*

\* 切割。

\*

\* 组：((A)(B(C)))

\*/

public static void functionDemo\_2(){

String str = "zhangsanttttxiaoqiangmmmmmmzhaoliu"; String[] names = str.split("(.)\\1+");//str.split("\\."); for(String name : names){

System.out.println(name);

}

/\*

\* 演示匹配。

\*/

public static void functionDemo\_1(){

//匹配手机号码是否正确。

String tel = "15800001111"; String regex = "1[358]\\d{9}"; boolean b = tel.matches(regex); System.out.println(tel+":"+b);

}

}

199.正则表达式常用练习：

**package** cn.itcast.regex.test; **import** java.util.TreeSet; **public class** RegexTest {

/\*\*

\* **@param** args

\*/

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

/\*

\* 1，治疗口吃:我我...我我...我我我要...要要要要...要要要要..学学学学学...学学编编...

\* 编编编编..编..程程...程程...程程程

\* 2，对ip地址排序。

\* 3，对邮件地址校验。

\*/

*test\_3*();

}

//对邮件地址校验。

**public static void** test\_3() {

String mail = "[abc1@sina.com.cn"](mailto:abc1@sina.com.cn);

String regex = "[a-zA-Z0-9\_]+@[a-zA-Z0-9]+(\\.[a-zA-Z]{1,3})+"; regex = "\\w+@\\w+(\\.\\w+)+"[;//1@1.1](mailto:%3B//1@1.1)

**boolean** b = mail.matches(regex);

System.*out*.println(mail+":"+b);

}

/\*

\* 1,治口吃。

\*/

**public static void** test\_1(){

String str = "我我...我我...我我我要...要要要要...要要要要..学学学学学...学学编编...

编编编编..编..程程...程程...程程程";

//1，将字符串中.去掉。 用替换。

str = str.replaceAll("\\.+", ""); System.*out*.println(str);

//2,替换叠词。

str = str.replaceAll("(.)\\1+", "$1"); System.*out*.println(str);

}

/\*

\* ip地址排序。

\*

\* 192.168.10.34 127.0.0.1 3.3.3.3 105.70.11.55

\*/

**public static void** test\_2(){

String ip\_str = "192.168.10.34 127.0.0.1 3.3.3.3 105.70.11.55";

//1,为了让ip可以按照字符串顺序比较，只要让ip的每一段的位数相同。

//所以，补零，按照每一位所需做多0进行补充。每一段都加两个0.

ip\_str = ip\_str.replaceAll("(\\d+)", "00$1"); System.*out*.println(ip\_str);

//然后每一段保留数字3位。

ip\_str = ip\_str.replaceAll("0\*(\\d{3})", "$1"); System.*out*.println(ip\_str);

//1，将ip地址切出。

String[] ips = ip\_str.split(" +"); TreeSet<String> ts = **new** TreeSet<String>();

**for**(String ip : ips){

// System.out.println(ip); ts.add(ip);

}

**for**(String ip : ts){ System.*out*.println(ip.replaceAll("0\*(\\d+)", "$1"));

}

}

}

200.网络爬虫：

package cn.itcast.regex.test;

import java.io.BufferedReader; import java.io.FileReader; import java.io.IOException; import java.io.InputStreamReader; import java.net.URL;

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

import java.util.regex.Matcher; import java.util.regex.Pattern;

/\*

\* 网页爬虫：其实就一个程序用于在互联网中获取符合指定规则的数据。

\* 爬取邮箱地址。

\*/

public class RegexTest2 {

/\*\*

* @param args
* @throws IOException

\*/

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

List<String> list = getMailsByWeb(); for(String mail : list){

System.out.println(mail);

}

}

public static List<String> getMailsByWeb() throws IOException {

//1,读取源文件。

// BufferedReader bufr = new BufferedReader(new FileReader("c:\\mail.html"));

URL url = new URL("http://192.168.1.100:8080/myweb/mail.html");

BufferedReader bufIn = new BufferedReader(new InputStreamReader(url.openStream()));

//2,对读取的数据进行规则的匹配。从中获取符合规则的数据. String mail\_regex = "\\w+@\\w+(\\.\\w+)+";

List<String> list = new ArrayList<String>(); Pattern p = Pattern.compile(mail\_regex); String line = null;

while((line=bufIn.readLine())!=null){ Matcher m = p.matcher(line); while(m.find()){

//3,将符合规则的数据存储到集合中。

list.add(m.group());

}

}

return list;

}

public static List<String> getMails() throws IOException{

//1,读取源文件。

BufferedReader bufr = new BufferedReader(new FileReader("c:\\mail.html"));

//2,对读取的数据进行规则的匹配。从中获取符合规则的数据. String mail\_regex = "\\w+@\\w+(\\.\\w+)+";

List<String> list = new ArrayList<String>(); Pattern p = Pattern.compile(mail\_regex); String line = null; while((line=bufr.readLine())!=null){

Matcher m = p.matcher(line); while(m.find()){

//3,将符合规则的数据存储到集合中。

list.add(m.group());

}

}

return list;

}

}

# 第十三章：HTML语言

201.列表标签演示：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Untitled Document</title>

</head>

<body>

<!--HTML注释：演示列表标签。 列表标签：dl

上层项目：dt

下层项目：dd : 封装的内容是会被缩进的，有自动缩进效果。

-->

<dl>

<dt>上层项目内容</dt>

<dd>下层项目内容</dd>

<dd>下层项目内容</dd>

<dd>下层项目内容</dd>

</dl>

<hr/>

<!--有序和无序的项目列表

有序：<ol> 无序：<ul>

无论有序和无序，条目的封装用的都是<li> 而且它们都有缩进效果。

-->

<ul type=*"square"*>

<li>无序项目列表</li>

<li>无序项目列表</li>

<li>无序项目列表</li>

<li>无序项目列表</li>

</ul>

<ol type=*"a"*>

<li>有序的项目列表</li>

<li>有序的项目列表</li>

<li>有序的项目列表</li>

<li>有序的项目列表</li>

</ol>

</body>

</html>

202.图片标签演示：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Untitled Document</title>

</head>

<body>

<!--演示图片标签 img.

-->

<img src=*"imgs\1.jpg"* height=*350* width=*500* border=*10* alt=*"啊，美女！"* />

</body>

</html>

203.表格标签演示：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Untitled Document</title>

</head>

<body>

<table border=*1* bordercolor=*"#0000EE"* cellpadding=*10* cellspacing=*0* width=*500*>

<tbody><!--表格的下一级标签是tbody，不定义也存在-->

<tr>

<th rowspan=*2*>个人信息</th>

<td>张三</td>

</tr>

<tr>

<td>30</td>

</tr>

</tbody>

</table>

<hr/>

<table border=*1* bordercolor=*"#0000EE"* cellpadding=*10* cellspacing=*0* width=*500*>

<tr>

<th colspan=*2*>个人信息</th>

</tr>

<tr>

<td>张三</td>

<td>30</td>

</tr>

</table>

<hr/>

<table border=*1* bordercolor=*"#0000EE"* cellpadding=*10* cellspacing=*0* width=*500*>

<caption>表格标题</caption>

<tr>

<th>姓名：</th>

<th>年龄：</th>

</tr>

<tr>

<td>张三</td>

<td>39</td>

</tr>

</table>

</body>

</html> 204.超链接标签演示： 演示1：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Untitled Document</title>

</head>

<body>

<!--

超链接： 作用：连接资源。

当有了href属性才有了点击效果。

href属性的值的不同，解析的方式也不一样。

如果在该值中没有指定过任何协议。 解析时，是按照默认的协议来解析该值的。默认协议是file协议。

-->

<a href=*"*[*http://www.sohu.com.cn"*](http://www.sohu.com.cn/)target=*"\_blank"*>新浪网站</a>

<hr/>

<a href=*"imgs/1.jpg"*>美女图片</a>

<hr/>

<a href=*"*[*mailto:abs@sina.com*](mailto:abs@sina.com)*"*>联系我们</a>

<hr/>

<a href=*"*[*http://www.xunlei.com/movies/fczlm.rmvb"*](http://www.xunlei.com/movies/fczlm.rmvb)>复仇者联盟</a><br/>

<a href=*"thunder://wertyuioasdfghjklwertyuio=="*>复仇者联盟</a>

<a href=*"javascript:void(0)"* onclick="alert('我弹')">这是一个超链接</a>

</body>

</html> 注意：取消默认点击超链接效果方法是javascript：void(0). 演示2：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Untitled Document</title>

</head>

<body>

<!--

定位标记， 专业术语：锚

-->

<a name=*top*>顶部位置</a>

<hr/>

<img src=*"111.jpg"* height=*900* width=*400* border=*10*/>

<hr/>

<a name=*center*>中间位置</a>

<hr/>

<img src=*"111.jpg"* height=*900* width=*400* border=*10*/>

<a href=*"#top"*>回到顶部位置</a>

<a href=*"#center"*>回到中间位置</a>

</body>

</html>

205.框架标签演示：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Untitled Document</title>

</head>

<!--

定义框架。使用标签frameset

-->

<frameset rows=*"30%,\*"*>

<frame src=*"top.html"* name=*"top"* />

<frameset cols=*"30%,\*"*>

<frame src=*"left.html"* name=*"left"* />

<frame src=*"right.html"* name=*"right"*/>

</frameset>

</frameset>

<body>

</body>

</html>

左边：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Untitled Document</title>

</head>

<body>

<H3>左边栏连接</H3>

<a href=*"../img.html"* target=*"right"*>链接一</a>

<a href=*"../table.html"* target=*"right"*>链接一</a>

<a href=*"../link.html"* target=*"right"*>链接一</a>

</body>

</html>

右边：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Untitled Document</title>

</head>

<body>

<h2> 内容显示区域</h2>

</body>

</html>

上边：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Untitled Document</title>

</head>

<body>

<h1>这是我的网站LOGO</h1>

</body>

</html>

206.画中画标签演示：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Untitled Document</title>

</head>

<body>

<!--iframe src=["http:](http://www.xxx.com/1.js)/[/www.xxx.com/1.js"](http://www.xxx.com/1.js) height=0 width=0>这是画中画标签，您如 果看到该文字，很遗憾，您的浏览器不支持该标签</iframe-->

<!--有的恶意网站会利用这个给用户电脑加东西或弹广告-->

<!--a href="a">下载地址1</a>

<a href="a">下载地址2</a-->

<a href=*"sadfsdfad"*><img src=*"c:\2.bmp"* border=*0*/></a>

</body>

</html>

207.表单标签演示：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Untitled Document</title>

</head>

<body>

<!--

如果要给服务端提交数据，表单中的组件必须有name和value属性。 用于给服务端获取数据方便。

-->

<form>

输入名称：<input type=*"text"* name=*"user"* value=*""* /><br/> 输入密码：<input type=*"password"* name=*"psw"* /><br/> 选择性别：<input type=*"radio"* name=*"sex"* value=*"nan"* />男

<input type=*"radio"* name=*"sex"* value=*"nv"* checked=*"checked"* />女

<br/>

选择技术：<input type=*"checkbox"* name=*"tech"* value=*"java"*/>JAVA

<input type=*"checkbox"* name=*"tech"* value=*"html"*/>HTML

<input type=*"checkbox"* name=*"tech"* value=*"css"*/>CSS<BR/>

选择文件：<input type=*"file"* name=*"file"*/><br/>

一个图片：<input type=*"image"* src=*"11.jpg"*/><br/>

<!--数据不需要客户端知道，但是可以将其提交服务端。-->

隐藏组件：<input type=*"hidden"* name=*"myke"* value=*"myvalue"*/><br/>

一个按钮：<input type=*"button"* value=*"有个按钮"* onclick="alert('有个阿牛')"

/><br/>

<select name=*"country"*>

<option value=*"none"*>--选择国家--</option>

<option value=*"usa"*>美国</option>

<option value=*"en"*>英国</option>

<option value=*"cn"* selected=*"selected"*>中国</option>

</select>

<textarea name=*"text"*></textarea>

<br/>

<input type=*"reset"* value=*"清除数据"*/><input type=*"submit"* value=*"提交数据*

*"* />

</form>

</body>

</html>

注意：没有单独封闭的标签有<br /> <hr /> <input />这三个较为常见，表单的组件中name和value是 键值对，必须至少有其中一个，这样才会有效的提交数据。

208. 对表单格式化以及get和post两种提交方式的区 别：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<!--meta http-equiv="refresh" content="3;url=[http://www.sina.com.cn"](http://www.sina.com.cn/) /-->

<title>Untitled Document</title>

</head>

<body>

<!--

提交方式：get提交。

地 址 栏 ： http://10.1.31.69:9090/?user=abc&psw=123&repsw=123&sex=nan&tech=java&tech=html&c ountry=cn

GET /?user=abc&psw=123&repsw=123&sex=nan&tech=java&tech=html&country=cn HTTP/1.1 Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/x-shockwave-flash, application/vnd.ms-excel, application/vnd.ms-powerpoint, application/msword, \*/\*

Accept-Language: zh-cn,zu;q=0.5 Accept-Encoding: gzip, deflate

User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; InfoPath.2) Host: 10.1.31.69:9090

Connection: Keep-Alive

提交方式：POST 地址栏：http://10.1.31.69:9090/

POST / HTTP/1.1

Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/x-shockwave-flash, application/vnd.ms-excel, application/vnd.ms-powerpoint, application/msword, \*/\*

Accept-Language: zh-cn,zu;q=0.5

Content-Type: application/x-www-form-urlencoded Accept-Encoding: gzip, deflate

User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; InfoPath.2) Host: 10.1.31.69:9090

Content-Length: 68 Connection: Keep-Alive Cache-Control: no-cache

user=hahah&psw=8989&repsw=8989&sex=nv&tech=html&tech=css&country=usa

GET提交和POST提交的区别？ 1,

get提交，提交的信息都显示在地址栏中。

post提交，提交的信息不显示地址栏中。

2， get提交，对于敏感的数据信息不安全。 post提交，对于敏感信息安全。

3， get提交，对于大数据不行，因为地址栏存储体积有限。 post提交，可以提交大体积数据。

4， get提交，将信息封装到了请求消息的请求行中。 post提交，将信息封装到了请求体中。

在服务端的一个区别。

如果出现将中文提交到tomcat服务器，服务器默认会用iso8859-1进行解码会出现乱码， 通过iso8859-1进行编码，再用指定的中文码表解码即可。 这种方式对get提交和post提交都有效。

但是对于post提交方式提交的中文，还有另一种解决办法，就是直接使用服务端一个对象 request对象的setCharacterEncoding方法直接设置指定的中文码表就可以将中文数据解析出 来。

这个方法只对请求体中的数据进行解码。 综上所述：表单提交，建议使用post。

和服务端交互的三种方式： 1，地址栏输入url地址。get 2，超链接。 get

3，表单。 get 和 post

如果在客户端进行增强型的校验(只要有一个组件内容是错误，是无法继续提交的，只有全对才可以提 交)

问，服务端收到数据后，还需要校验吗？ 需要，为了安全性。

如果服务端做了增强型的校验，客户端还需要校验吗？ 需要，因为要提高用户的上网体验效果，减轻服务器端的压力。

-->

<form action=*"http://10.1.31.69:9090"* method=*"post"*>

<table border=*"1"* bordercolor=*"#0000ff"* cellpadding=*10* cellspacing=*0* width=*600*>

<tr>

<th colspan=*"2"*>注册表单</th>

</tr>

<tr>

<td>用户名称：</td>

<td><input type=*"text"* name=*"user"* /></td>

</tr>

<tr>

<td>输入密码：</td>

<td><input type=*"password"* name=*"psw"* /></td>

</tr>

<tr>

<td>确认密码：</td>

<td><input type=*"password"* name=*"repsw"* /></td>

</tr>

<tr>

<td>选择性别：</td>

<td>

<input type=*"radio"* name=*"sex"* value=*"nan"* />男

<input type=*"radio"* name=*"sex"* value=*"nv"* />女

</td>

</tr>

<tr>

<td>选择技术：</td>

<td>

<input type=*"checkbox"* name=*"tech"* value=*"java"* />JAVA

<input type=*"checkbox"* name=*"tech"* value=*"html"* />HTML

<input type=*"checkbox"* name=*"tech"* value=*"css"* />CSS

</td>

</tr>

<tr>

<td>选择国家：</td>

<td>

<select name=*"country"*>

<option value=*"none"*>--选择国家--</option>

<option value=*"usa"*>--美国--</option>

<option value=*"en"*>--英国--</option>

<option value=*"cn"*>--中国--</option>

</select>

</td>

</tr>

<tr>

<th colspan=*"2"*>

<input type=*"reset"* value=*"清除数据"* />

<input type=*"submit"* value=*"提交数据"* />

</th>

</tr>

</table>

</form>

</body>

</html>

209.其他标签简述：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Untitled Document</title>

</head>

<body>

<pre> class Demo

{

public static void main(String[] args)

{

System.out.println("hello");

}

}

</pre>

<hr/>

<marquee direction=*"down"* behavior=*"slide"*>嘿，原来我会飞!</marquee>

<b>这是</b><i>演示</i><u>其他</u>常见的小标签。 X<sub>2</sub> X<sup>2</sup>

</body>

</html>

210.标签的分类：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Untitled Document</title>

</head>

<body>

<div>这是一个div区域1</div>

<div>这是一个div区域2</div>

<span>span区域1</span>

<span>span区域2</span>

<p>这是一个段落1</p>

<p>这是一个段落2</p>

<!--

标签分为两大类。

1,块级标签(元素):标签结束后都有换行。div p dl table title ol ul 2,行内标签(元素):标签结束后没有换行。 font span img input select a

-->

<dl>

<dt>ddddd</dt>

<dd>eeeee</dd>

</dl> hahahahh

</body>

</html>

211.HTML补充知识点：

## HTML，即超文本标记语言，是目前网络上应用最为广泛的语言，也是构成网页文档的主要语言，

HTML的结构包括头部（Head）、主体（Body）两大部分，其中头部描述浏览器所需的信息，主 体则包含所要说明的具体内容。标记+中间内容=HTML元素，如<b>bjsxt</b>。 HTML语言被称为互联网三大基石乊一（其余两大基石分别为：HTTP协议、URL），HTML丌是一 种编程语言，而是一种标记语言，HTML使用标记标签杢描述网页，并且HTML语言丌区分大小写！ HTML标记标签通常被称为HTML标签，HTML标签是由尖括号包围的关键词，比如<html>,HTML 标签是成对出现的，比如<b></b>,标签对中的第一个标签是开始标签，第二个标签是结束标签， 某些HTML元素没有结束标签，比如<br />。

表格标签<table><tr><td><th>,<table>是定义HTML表格的标签，简单的 HTML表格是由 table元素以及一个戒多个tr、th戒td元素组成，tr元素定义表格行，th元素定义表头，td元素定义 表格单元。 块元素<div>，<div>可定义文档中的分区戒节，<div>标签可以把文档分割为独立的、丌同的部 分，它可以用作严格的组细工具，并且丌使用任何格式不其关联，<div>是一个块级元素，这就意 味着它的内容自劢地开始一个新行。

表单标签：

文本框 <input type="text" . . .> 输入一行文本

密码 <input type="password" . . .> 输入一行文本，但丌可见，叧以星号显示 单元按钮 <input type="radio" . . .> 叧能选择一个选项

复选框 <input type="checkbox" . . .> 可以选择多个戒一个选项

隐藏域 <input type="hidden" . . .> 该域的值对用户丌可见，但会提交给服务器 下拉列表 <select . . .>

<option . . .> . . .</option> （可以放多个option）

</select>

文本域 <textarea . . .> . . .</textarea> 可以输入多行文本

|  |  |  |  |
| --- | --- | --- | --- |
| 提交按钮 | <input | type="submit"> | 提交表单 |
| 普通按钮  重置按钮 | <input  <input | type="button">  type="reset"> | 重置表单信息 |

注意：要提交数据的表单域必须加name属性，否则该表单的数据丌会被提交到服务器上！

--------------------------------------------------------------------------------------------------

对亍单选按钮（用亍多选一的情冴），要知道单选按钮是分组的，name相同即为一组，一组叧能 选择一个按钮，value属性如果丌添加的话，那么选该按钮时提交默认为on，而丌是空字符串，所 以建议使用单选按钮时一定要指定value，对亍复选框也是一样，单选按钮、复选框默认选中用的 是checked属性，下拉列表默认选中的话用的是selected属性。

--------------------------------------------------------------------------------------------------

HTML字符实体，有些字符比如说"<"字符在HTML中有特殊的含义，因此丌能在文本中使用，想 要在HTML中显示一个小亍号"<"就需要用到字符实体，即&lt;戒者&#60; ，并且实体名是区分大 小写的，字符实体拥有三个部分：一个and符号（&），一个实体名戒者一个实体号，最后一个是 分号（;）。

--------------------------------------------------------------------------------------------------

XML是可扩展标记语言，HTML是超文本标记语言，丌同的是HTML语言中的标签都是固定的，而 XML中的标签是可以自定义的，XML文件中存储的是配置信息。

第十四章：CSS语言

212.CSS不HTML相结合：

<!--

css和html相结合的第一种方式。

1，每一个html标签中都有一个style样式属性，该属性的值就是css代码。 2，使用style标签的方式，一般都定义在head标签中。

优先级： 标签选择器<类选择器<id选择器<style属性

-->

213.选择器：

组合选择器：

.haha,div b{对多种选择器进行相同样式定义 background-color:#000; color:#C00;

}

关联选择器：

span b{关联选择器 选择器中的选择器 background-color:#09F; color:#FFF;

}

## 元素选择器：HTML元素是最典型的选择器类型，任何一个HTML元素都可以作为选择

器，元素选择器的有效范围是页面中所有的、名称相同的HTML元素，格式：元素 { 属性：值 }， 例如 h1 { color：red } h2 { color：#FF0000；} 戒者h1，h2{ color：#ff0000；}

类选择器：如果想把某一个样式应用到丌同的HTML元素上，就可以采用类选择器杢定义，

然后在丌同标签中应用它们，格式1：.类名{ 属性：值；} 格式2：元素.类名{ 属性：值；}

ID选择器：ID选择器不类选择器的概念相似，叧是ID选择器叧能被引用一次，而类选择

器可以被多次引用，格式：#id名{ 属性：值；} 示例：#red{ color：red } <p id="red"> 欢迎 使用id选择器</p>

包含选择器：如果需要为位亍某个标记符内的元素设置特定的样式规则，则应将选择器

指定为具有上下文关系的HTML标记，如h1 b{ color：blue；} /\*注意h1和b乊间以空格分隑\*/

214.盒子模型（box model）演示：

<html xmlns="<http://www.w3.org/1999/xhtml>">

<head>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

<title>无标题文档</title>

<style type="text/css">

/\* body{

margin:0px;

}

\*/ div{

border:#09F solid 1px; height:100px; width:200px;/\*\*/

}

#div\_1{

/\*border-bottom:#F60 2px dashed;\*/ background-color:#F90;

/\*padding:20px 100px 200px 300px;上 右 下 左

margin:0px;\*/ float:left;

}

#div\_2{

background-color:#0CF; float:left;

}

#div\_3{

background-color:#3F0;8 clear:left;/\*\*/

}

</style>

</head>

<body>

<div id="div\_1">

第一个盒子11第一个盒子11

</div>

<div id="div\_2">

第二个盒子22

</div>

<div id="div\_3">

第三个盒子33

</div>

</body>

</html>

215.盒子模型（定位）演示：

<html xmlns="<http://www.w3.org/1999/xhtml>">

<head>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

<title>无标题文档</title>

<style type="text/css"> div{

border:#09F solid 1px; height:100px; width:200px;/\*\*/

}

#div\_0{

background-color:#CCC; height:400px; width:400px; position:relative; top:100px; left:100px;

}

#div\_1{

background-color:#F90; position:relative; top:20px;

}

#div\_2{

background-color:#0CF; width:220px;

}

#div\_3{

background-color:#3F0;8

}

</style>

</head>

<body>

<div id="div\_0">

<div id="div\_1">

第一个盒子11第一个盒子11

</div>

<div id="div\_2">

第二个盒子22

</div>

<div id="div\_3">

第三个盒子33

</div>

</div>

</body>

</html>

# 216.盒子模型乊float\_图文混排（演示）：

<html xmlns="<http://www.w3.org/1999/xhtml>">

<head>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

<title>无标题文档</title>

<style type="text/css">

#imgtext{

border:#06F dashed 1px; width:180px;

}

#img{

}

#text{

}

float:right;

color:#F60;

font-family:"华文隶书";

</style>

</head>

<body>

<div id="imgtext">

<div id="img">

<img src="9.bmp" />

</div>

<div id="text"> 这是一个美女，说明文字。 一切尽在不言中。哇

！这是一个美女，说明文字。 一切尽在不言中。哇！ 这是一个美女，说明文字。 一切尽在不言中。哇！

</div>

</div>

</body>

</html>

# 217.盒子模型乊position\_图像签名（演示）：

<html xmlns="<http://www.w3.org/1999/xhtml>">

<head>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

<title>无标题文档</title>

<style type="text/css">

#text{

}

font-family:"华文隶书"; font-size:24px;

position:absolute; top:80px; left:10px;

#imgtext{

border:#F60 dotted 1px; width:500px; position:absolute; top:100px;

}

</style>

</head>

<body>

<div id="imgtext">

<div id="img">

<img src="1.jpg" height="300" width="500" />

</div>

<div id="text">

花丛中的美女，我的！

</div>

</div>

</body>

</html>

# 第十五章：JavaScript语言

218.JavaScript语法（js不和html结合+js语法通用内容

+js运算符）演示：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Javascript语法</title>

</head>

<body>

<!--

js和html相结合的方式： 1，将javascript代码封装到<script>标签中。

2，将javascript代码封装到js文件中，并通过<script>中的src属性进行导入。

注意：如果<script>标签中使用src属性，那么该标签中封装的javascript代码不会被执行。 所以通常导入js文件都是用单独<script>来完成。

-->

<!--导入一个js文件-->

<!--script type="text/javascript" src="demo.js"></script-->

<!--封装javascript代码。-->

<script type=*"text/javascript"*>

// alert("hello javascript");

</script>

<!--演示javascript语法通用内容-->

<script type=*"text/javascript"*>

/\*

\* 通常高级程序设计语言所包含的语法内容：

\* 1，关键字:该种语言中被赋予了特殊含义的单词。

\* 2，标示符:用于标示数据和表达式的符号。通常可以理解为在程序中自定义的名称。比如变量

名，函数名。

同。

\* 3，注释:注解说明解释程序。用于调试程序。

\* 4，变量:用于标示内存中一片空间。用于存储数据，该空间的中的数据是可以变化的。

\* 什么时候使用变量，当数据不确定的时候。

\* 5，运算符：可以让数据进行运算的符号。

\* 6，语句:用于对程序的运行流程进行控制的表达式。

\* 7，函数:用于对功能代码进行封装，便于提高复用性。

\* 8，数组:对多数据进行存储，便于操作。就是传说中容器。

\* 9，对象:只要是基于对象的语言，或者是面向对象的语言，就存在着对象的概念，

\* 对象就是一个封装体。既可以封装数据有可以封装函数。

\*

\* 这些都是高级程序设计语言具备的共性内容，只不过各种语言对这些内容的表现形式有所不

\* 但是使用基本思想是一致。

\*/

</script>

<script type=*"text/javascript"*>

/\*

\* js中定义变量。使用到关键字。

\* 这个关键字就是var

\*/

//定义变量。Js是弱类型的。

**var** x = 4;//var不写也行，因为js是非严谨的语言，但是我们开发的时候建议都按照严谨的 方式来定义。

x = "abc";//重新赋值为字符串abc。

x = 3.45;//赋值为小数。其实也是数字类型。 x = **true**;//赋值为boolean类型。

x = 'c';//赋值为字符串c。

// alert("x="+x);//这是一个函数，将具体的参数通过对话框进行显示。

</script>

<script type=*"text/javascript"*>

/\*

\* 运算符。

\* 1，算术运算符。

\* + - \* / % ++ --

\* 2，赋值运算符。

\* = += -= \*= /= %=

\* 3，比较运算符:运算的结果要么是false要么是true。

\* > < >= <= != ==

\*

\* 4，逻辑运算符。用来连接两个boolean型的表达式。

\* ! && ||

\* 5，位运算符。

\* & | ^ >> << >>>

\* 6，三元运算符。

\* ? :

\*/

//1，算术运算符演示。

**var** a = 3710;

// alert("a="+a/1000\*1000);//a=3710;

**var** a1 = 2.3,b1 = 4.7;

// alert("a1+b1="+(a1+b1));

// alert("12"-1);//11

// alert("12"+1);//121

// alert(true+1);//2//因为在js中false就是0，或者null，非0，非null。就是true。默认 用1表示。

// alert(2%5);//2

**var** n = 3,m; m = ++n;

// alert("n="+n+",m="+m);

// =================================================

// 2，赋值运算符

**var** i = 3;

// i = i+2; i+=2;

// alert("i="+i);

// =================================================

// 3，比较运算符。

**var** z = 3;

// alert(z!=4);

// =================================================

// 4，逻辑运算符。

**var** t = 4;

// alert(t>7 && t<6);

// alert(!true);

// =================================================

// 5，位运算符。

**var** c = 6;

// alert(c&3);//2

// alert(5^3^3);//5

// alert(c>>>1);//6/2(1);

// alert(c<<2);//24

// =================================================

//6，三元运算符。

// 3>0?alert("yes"):alert("no");

// alert(3>10?100:200);

</script>

<script type=*"text/javascript"*>

/\*

\* 一些细节：

\* 1,undefined:未定义。其实它就是一个常量。

\*/

**var** xx;

// alert(xx);//undefined

// alert(xx==undefined);//true

//要想获取具体的值的类型。可以通过typeof来完成。

// alert(typeof("abc")=='string');//string

// alert(typeof(2.5)=='number');//number//判断类型。

// alert(typeof(true));//boolean

// alert(typeof(78));//number

// alert(typeof('9'));//string

</script>

</body>

</html>

# 219.JavaScript语法（顺序+判断+选择+循环+其他=五

种语句）演示：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Untitled Document</title>

</head>

<body>

<script type=*"text/javascript"*>

/\*

\* 语句：

\* 1，顺序结构。

\*

\* 2，判断结构。

\* 代表语句： if.

\* 3，选择结构。

* switch

\* 4，循环结构。

* while do while for

\* 5，其他语句。

\* break：跳出选择，跳出循环。

\* continue:用于循环语句，结束本次循环继续下次循环。

\*/

// alert("abc1");

// alert("abc2");

// ------------------------------------------------

/\*

var x = 3;

// if(x=4){//注意。 if(4==x){//建议将常量放左边。以报错来修正代码。

alert("yes");

}else{

alert("no");

}

if(x>1)

alert("a"); else if(x>2)

alert("b"); else if(x>3)

alert("c"); else

alert("d");

\*/

// ------------------------------------------------

/\*

//选择结构。

var x = "abc"; switch(x){

default:

alert("c"); break;//省略。

case "kk":

alert("a"); break;

case "abc":

alert("b"); break;

}

\*/

//循环结构。

/\*

var x = 1;

document.write("<font color='blue' >"); while(x<10)

{

// alert("x="+x);

//将数据直接写到当前页面当中。 document.write("x="+x+"<br/>"); x++;

}

document.write("</font>");

for(var x=0; x<3; x++){ document.write("x="+x);

}

\*/

/\*

w:for(var x=0; x<3; x++){ for(var y=0; y<4; y++){

document.write("x==="+x); continue w;//跳出当前循环。

}

}

\*/

</script>

</body>

</html>

220.JavaScript中的数组：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Untitled Document</title>

</head>

<body>

<script type=*"text/javascript"*>

/\*

* javascript中的数组演示。

\*

\* 数组用于存储更多的数据，是一个容器。

\* 特点：

\* 1，长度的是可变的。

\* 2，元素的类型是任意的。

\* 建议在使用数组时，存储同一类型的元素。操作起来较多方便。

\*

\* js中的数组定义的两种方式：

\* 1，var arr = []; var arr = [3,1,5,8];

\* 2，使用了javascript中的Array对象来完成的定义。

* var arr = new Array();//var arr = [];

\* var arr1 = new Array(5);//数组定义并长度是5.

\* var arr2 = new Array(5,6,7);//定义一个数组，元素是5,6,7;

\*/

**var** arr = [23,78,100];

// alert(typeof(arr));//对象类型Object

// alert("len:"+arr.length); arr[4] = 348;

arr[1] = "abc"; arr[2] = **true**;

//遍历数组。

**for**(**var** x=0; x<arr.length; x++){ document.write("arr["+x+"]="+arr[x]+"<br/>");

}

</script>

</body>

</html>

# 221.JavaScript中凼数（定义格式+绅节）演示：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Untitled Document</title>

</head>

<body>

<script type=*"text/javascript"*>

/\*

\* js中的函数演示。

\* 函数：就是一个功能的封装体。

\* 定义功能通常需要两个明确。

\* 1，功能的结果。

\* 2，功能实现中的参与运算的未知的内容。

\*

\* js中的定义函数的格式。

\* 通过指定的关键字来定义。

\* function 函数名(参数列表){

\* 函数体：

\* return 返回值;//如果没有具体的返回值，return语句可以省略不写。

\* }

\*/

**function** demo(){ alert("demo run");

### return ;

}

// demo();//调用函数。

/\*

\* 需要加法运算。

\*/

**function** add(x,y){

**return** x+y;

}

**var** sum = add(4,5);

// alert("sum="+sum);

</script>

<script type=*"text/javascript"*>

/\*

\* 函数的一些细节。

\* 1，只要使用函数的名称就是对这个函数的调用。

\* 2，函数中有一个数组在对传入的参数进行存储。这个数组就是arguments

\*/

**function** show(x,y){

// alert(arguments.length);

// alert(x+":"+y);

**for**(**var** a=0;a<arguments.length; a++){ document.write(arguments[a]);

}

}

// show(4,5,6,9,7,4);//建议函数中定义几个参数就传递几个实参。

</script>

<script type=*"text/javascript"*>

/\*

\* 函数的细节2：

\*

\*/

**function** getSum(){

**return** 100;

}

// var sum = getSum();//getSum函数运行，并将返回的结果赋值给sum。

**var** sum = getSum;//getSum本身是一个函数名，而函数本身在js中就是一个对象。getSum

//就是这个函数对象的引用.

//将getSum这个引用的地址赋值给sum,这时sum也指向了这个函数对象。

//相当于这个函数对象有两个函数名称。

// alert("sum="+sum);//打印时候如果sum指向的是函数对象，那么会将该函数对象的字符串表

//现形式打印出来，

//就是该函数的代码定义格式。

**function** show2(){ alert("show2 run");

}

// alert(show2());

</script>

</body>

</html>

# 222.JavaScript中（劢态凼数、匿名凼数）凼数演示：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Untitled Document</title>

</head>

<body>

<script type=*"text/javascript"*>

/\*

\* js函数的其他表现形式。

\*

\* 动态函数:使用的是js中内置的一个对象Function.

\* 只不过用的不是很多。

\* 参数列表，函数体都是通过字符串动态指定的。

\*/

**var** add = **new** Function("x,y","var sum; sum=x+y; return sum;");

**var** he = add(4,8);

// alert("he="+he);

**function** add2(x,y){

**var** sum ; sum = x+y; **return** sum;

}

</script>

<script type=*"text/javascript"*>

/\*

\* 匿名函数：没有名字的函数。

\* 通常是函数的简写形式。

\*

\*/

**var** add3 = **function** (a,b){

**return** a+b;

}

alert(add3(7,8));

**function** haha()

{

alert("haha run");

}

**var** xixi = haha;

//上述代码可以简写成下面的方式。

**var** xixi = **function**()

{

alert("haha run");

}

</script>

</body>

</html>

# 223.在页面中打印九九乘法表（练习）：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Untitled Document</title>

<link rel=*"stylesheet"* href=*"table.css"*/>

</head>

<body>

<script type=*"text/javascript"*>

/\*

\* 练习：在页面中显示一个99乘法表。

\*

\*/ document.write("<table>"); **for**(**var** x=1; x<=9; x++){

document.write("<tr>");

**for**(**var** y=1; y<=x; y++){ document.write("<td>"+y+"\*"+x+"="+y\*x+"</td>");

}

document.write("</tr>");

}

document.write("</table>");

</script>

</body>

</html>

# 224.JS语法对数组（叏最值、排序、查找、元素反转）

的综合练习：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Untitled Document</title>

</head>

<body>

<script type=*"text/javascript"*>

/\*

\* 综合练习：

\* 1，定义功能，完成对数组的最值获取。

\*

\* 2，对数组排个序。

\*

\* 3，对数组查个找。

\*

\* 4，对数组元素进行反转。

\*

\*/

**var** arr = [66,13,37,21,89,17];

//取最值。

**function** getMax(arr){

**var** max = 0;

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

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

}

**return** arr[max];

}

**var** maxValue = getMax(arr);

// alert("maxValue:"+maxValue);

//排序。

**function** sortArray(arr){

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

**for**(**var** y=x+1; y<arr.length; y++){

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

swap(arr,x,y);

}

}

}

}

//数组中的元素位置置换。

**function** swap(arr,x,y){ **var** temp = arr[x]; arr[x] = arr[y]; arr[y] = temp;

}

**function** println(val){ document.write(val+"<br/>");

}

// document.write("排序前:"+arr); println("排序前:"+arr)

// sortArray(arr); println("排序后:"+arr)

// document.write("排序后:"+arr);

</script>

<script type=*"text/javascript"*>

/\*

\* 3，对数组查个找。

\*

\* 4，对数组元素进行反转。

\*/

//查找。

**function** searchElement(arr,key){

**for**(**var** x=0;x<arr.length; x++){

**if**(arr[x]==key) **return** x;

}

**return** -1;

}

//折半，必须有前提。必须是有序的数组。

**function** binarySearch(arr,key){

**var** max,min,mid; min = 0;

max = arr.length-1;

**while**(min<=max){

mid = (max+min)>>1;

**if**(key>arr[mid]) min = mid + 1;

**else if**(key<arr[mid]) max = mid - 1;

### else

**return** mid;

}

**return** -1;

}

//对数组的反转。

**function** reverseArray(arr){

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

}

}

reverseArray(arr); println("反转后:"+arr);

</script>

</body>

</html>

225.JS中的全尿发量和尿部发量：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Untitled Document</title>

</head>

<body>

<script type=*"text/javascript"*>

// for(var x=0; x<3; x++){//在脚本片段中定义的变量，是全局变量。

// document.write("x="+x);

// }

// function show(){

// var x = 6;//局部变量。

// }

// document.write("X======="+x);

**var** x = 3;//全局变量x.

**function** show(x){//函数局部的变量x x = 8;

}

show(x); document.write("x="+x);//x=3;

</script>

<script type=*"text/javascript"*>

// document.write("xx="+x);

</script>

</body>

</html>

226.JS中Object对象演示：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Untitled Document</title>

</head>

<body>

<script type=*"text/javascript"*>

/\*

\* 演示一下Object对象。

\* toString():将对象变成字符串。

\* valueOf()

\*/

**function** show(){ alert("show run");

}

// alert(show.toString());

**var** arr = [3,4,1,6,8];

// alert(arr.toString());

**var** abc = **function**(){ alert("abc run");

}

// alert(abc); alert(abc.valueOf());

</script>

</body>

</html>

# 227.JS中String对象（定义格式+获叏原型方法+自定义

trim方法）演示：

自定义trim()方法：

/\*\*

\* 字符串对象新功能，去除字符串两端的空格。

\*/

String.prototype.trim = **function**(){

**var** start,end; start=0; end=**this**.length-1;

**while**(start<=end && **this**.charAt(start)==' '){ start++;

}

**while**(start<=end && **this**.charAt(end)==" "){ end--;

}

**return this**.substring(start,end+1);

}

定义格式和获取原型的prototype方法：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Untitled Document</title>

</head>

<body>

<script type=*"text/javascript"* src=*"out.js"*></script>

<script type=*"text/javascript"* src=*"stringtool.js"*></script>

<script type=*"text/javascript"*>

/\*

* 演示string对象。

\* 表现形式：

* var str = new String("abc");
* var str = "abc";

\*/

**var** str = "abcde";

println("len="+str.length); println(str.bold());//加粗 println(str.fontcolor("red"));//字体颜色。

println(str.link(["htt](http://www.163.com/)p[://www.163.com](http://www.163.com/)"));//将字符串变成超链接。 println(str.substr(1,3));//bcd println(str.substring(1,3));//bc

/\*

\* 发现js中的string对象方法有限，想要对字符串操作的其他功能。

\* 比如：去除字符串两端的空格。这时只能自定义。

\*/

//去除字符串两端的空格。

**function** trim(str){

//定义两个变量，一个记录开始的位置。一个记录结束的位置。

//对开始的位置 的字符进行判断，如果是空格，就进行递增，直到不是空格为止。

//对结束的位置 的字符进行判断，如果是空格，就进行递减，直到不是空格为止。

//必须要保证开始<=结束，这样才可以进行截取。

**var** start,end; start=0; end=str.length-1;

**while**(start<=end && str.charAt(start)==' '){ start++;

}

**while**(start<=end && str.charAt(end)==" "){ end--;

}

**return** str.substring(start,end+1);

}

**var** s = " ab c ";

// alert("-"+trim(s)+"-");

// alert("abc".bold());//<b>this</b>

/\*

\* 既然trim方法是用来操作字符串的方法，可不可以像字符串已有的方法一样，

\* 将该方法也定义到字符串对象中呢？直接用字符串对象调用就欧了。

\*/

/\*

\* 这里就可以使用一个该字符串的原型属性来完成：

\* 原型：就是该对象的一个描述。该描述中如果添加了新功能。

\* 那么该对象都会具备这些新功能。

\* 而prototype就可以获取到这个原型对象。

\* 通过prototype就可以对对象的功能进行扩展。

\*

\* 需求：想要给string对象添加一个可以去除字符串两端空格的新功能.

\* 就可以使用原型属性来完成。

\*/

//给string的原型中添加一个功能。 注意：给对象添加新功能直接使用 对象.新内容 即可。

// String.prototype.len = 199;//给string的原型对象中添加一个属性 名为len。值为199.

//添加行为。

// println("abc".len);

// alert("-"+" ab cd ".trim()+"-");

</script>

</body>

</html>

# 228.定义打印指定参数数据到页面方法和将数据打印到

页面并换行的方法：

/\*\*

\* 打印指定参数数据到页面上，并换行。

\*/

**function** println(param){ document.write(param+"<br/>");

}

/\*\*

\* 打印指定参数数据到页面上。

\*/

**function** print(param){ document.write(param);

}

# 229.prototype（）获叏原型方法练习：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Untitled Document</title>

</head>

<body>

<script type=*"text/javascript"* src=*"../demo/out.js"*></script>

<script type=*"text/javascript"* src=*"../demo/stringtool.js"*></script>

<script type=*"text/javascript"*>

/\*

\* 练习1，给字符串添加一个功能，将字符串变成一个字符数组。

\*

\* 练习2，给字符串添加一个功能，将字符串进行反转。

\*/

**var** str = "abcdefg";

// println(str.toCharArray());

// println(str.reverse());

// println(str.toString());

</script>

</body>

</html>

所需要的两个方法：

/\*\*

\* 字符串新功能，添加一个将字符串转成字符数组。

\* 返回一个数组

\*/

String.prototype.toCharArray = **function**(){

//定义一个数组。

**var** chs = [];

//将字符串中的每一位字符存储到字符数组中。

**for**(**var** x=0; x<**this**.length; x++){

chs[x] = **this**.charAt(x);

}

**return** chs;

}

/\*\*

\* 添加一个，将字符串进行反转的方法。

\*/

String.prototype.reverse = **function**(){

**var** arr = **this**.toCharArray();

//将数组位置置换功能进行封装，并定义到了反转功能内部。

**function** swap(arr,a,b){ **var** temp = arr[a]; arr[a] = arr[b]; arr[b] = temp;

}

**for**(**var** x=0,y=arr.length-1; x<y ; x++,y--){

swap(arr,x,y);

}

**return** arr.join("");

}

# 230.JS中数组Array的常用方法使用演示：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Untitled Document</title>

</head>

<body>

<script type=*"text/javascript"* src=*"../demo/out.js"*></script>//导入demo目录

//下的out.js文件，实际上是用到了定义在out.js中定义的方法才用src导入的

<script type=*"text/javascript"*>//设置格式

/\*

\* 演示数组。

\*/

**var** arr = ["nba","haha","cba","aaa","abc"];

**var** arr2 = ["qq","xiaoqiang",70]; println(arr);

println(arr2);

//在arr数组上连接一个元素"mm"，再连接一个arr2数组。

**var** newArr = arr.concat("mm",arr2);//将mm作为新数组中的元素，将arr2数组中的元

//素也作为新数组中的元素。 println(newArr);

println(arr.join("-"));

// println(myJoin(arr,"+"));

//模拟一下的join的实现原理。

**function** myJoin(arr,separator){

**var** str = "";

**for**(**var** x=0;x<arr.length ; x++){

**if**(x!=arr.length-1)

str += arr[x]+separator;

### else

str += arr[x];

}

**return** str;

}

//移除数组中的元素，并返回该元素。pop

println("<hr/>");

/\* println(arr);

println(arr.pop());//删除并返回最后一个元素。

println(arr);

// println(arr.reverse());

println(arr.shift());//删除并返回第一个元素。 println(arr);

\*/ println(arr);

// arr.sort();

// var temp = arr.splice(1,3,8080,9527,"xixixi","wangcai");//删除元素并可以进

//行元素的替换。

// println(temp);

// println(arr.unshift("uuuu"));

println(arr);

</script>

</body>

</html> 231.JS中数组的堆栈和队列以及自定义功能（演示）： 自定义的功能（方法）：

/\*\*

\* 数组获取最大值的方法。

\*/

Array.prototype.getMax = **function**(){

**var** temp = 0;

**for**(**var** x=1; x<**this**.length; x++){

**if**(**this**[x]>**this**[temp]){ temp = x;

}

}

**return this**[temp];

}

/\*

\* 数组的字符串表现形式。

\* 定义toString方法。 相当于java中的复写。

\*

\*/

Array.prototype.toString = **function**(){

**return** "["+**this**.join(", ")+"]";

}

堆栈和队列以及自定义功能演示：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Untitled Document</title>

</head>

<body>

<script type=*"text/javascript"* src=*"../demo/out.js"*></script>

<script type=*"text/javascript"*>

/\*

\* 数组练习。

\* 用数组实现js中的堆栈 或者队列数据结构。

\*/

**var** arr = [];

// arr.unshift("abc1","abc2","abc3"); arr.unshift("abc1"); arr.unshift("abc2"); arr.unshift("abc3");

println(arr);

// println(arr.pop());

// println(arr.pop());

// println(arr.pop()); println(arr.shift()); println(arr.shift()); println(arr.shift());

</script>

<script type=*"text/javascript"* src=*"../demo/arraytool.js"*></script>

<script type=*"text/javascript"*>

/\*

\* 给数组对象添加新功能，使用到原型属性。

\*/

**var** array = ["nba","haha","cba","aaa","abc"];

**var** maxValue = array.getMax(); println("maxValue:"+maxValue); println(array.toString());

</script>

</body>

</html>

# 232.JS中Date对象的基本方法和with语句演示：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Untitled Document</title>

</head>

<body>

<script type=*"text/javascript"* src=*"out.js"*></script>

<script type=*"text/javascript"*>

/\*

\* 演示js中的日期 Date.

\*/

**var** date = **new** Date();

// println(date);

// println(date.toLocaleString());//日期和时间

// println(date.toLocaleDateString());//只有日期。

/\*

\* 为了简化对象调用内容的书写。

\* 可以使用js中的特有语句with来完成。

\* 格式：

\* with(对象)

\* {

\* 在该区域中可以直接使用指定的对象的内容，不需要写对象.

\* }

\*/

// var year = date.getFullYear();

// var month = date.getMonth()+1;

// var day = date.getDate();

// var week = getWeek(date.getDay());

**with**(date){

**var** year = getFullYear(); **var** month = getMonth()+1; **var** day = getDate();

**var** week = getWeek(getDay());

println(year+"-----"+month+"---+--"+day+"-----"+week);

}

**function** getWeek(num){

'];

**var** weeks = ['星期日','星期一','星期二','星期三','星期四','星期五','星期六

**return** weeks[num];

}

// 日期对象和毫秒值之间的转换。

**var** date2 = **new** Date();

//获取毫秒值。 日期对象-->毫秒值。

**var** time = date2.getTime(); println("time:"+time);//1338191622156

//将毫秒值转成日期对象。

//1,new Date(time);//setTime();

**var** date3 = **new** Date(time);

//将日期对象和字符串之间进行转换。

//日期对象转成字符串。toLocaleString toLocaleDateString

//将字符串转成日期对象。具备指定格式的日期字符串-->毫秒值--->日期对象。

**var** str\_date = "9/28/2017";

**var** time2 = Date.parse(str\_date);

**var** date3 = **new** Date(time2); println(date3.toLocaleDateString());

</script>

</body>

</html>

# 233.JS中Math对象的基本功能演示：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Untitled Document</title>

</head>

<body>

<script type=*"text/javascript"* src=*"out.js"*></script>

<script type=*"text/javascript"*>

/\*

\* 演示Math对象。该对象的中的方法都是静态的。不需要new，直接Math调用即可。

\*/

**var** num1 = Math.ceil(12.34);//返回大于等于指定参数的最小整数。

**var** num2 = Math.floor(12.34);//返回小于等于指定数据的最大整数。 **var** num3 = Math.round(12.54);//四舍五入 println("num1="+num1);

println("num2="+num2); println("num3="+num3);

**var** num4 = Math.pow(10,2); println("num4="+num4);

println("<hr/>");

**for**(**var** x=0; x<10; x++){

// var num = Math.floor(Math.random()\*10+1); **var** num = parseInt((Math.random()\*10+1)); println(num);

}

</script>

</body>

</html>

# 234.JS中全尿方法及for（发量 in 对象）语句演示：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Untitled Document</title>

</head>

<body>

<script type=*"text/javascript"* src=*"out.js"*></script>

<script type=*"text/javascript"*>

// parseInt("123");

/\*

\* 演示global的全局方法。

\*/

// println(parseInt("123")+1);

// var val = parseInt("12abc");//val = 12;

// println("value="+val);//通过isNaN来判断结果是否非法。

/\*

\* 将指定进制格式的字符串转成十进制。

\*/

**var** num = parseInt("110",2); println("num="+num);

**var** num1 = parseInt('0x3c',16); println("num1="+num1);

//将十进制转成其他进制。使用数字对象完成。

**var** num3 = **new** Number(6); println("num3="+num3.toString(2));

**var** num4 = 60; println("num4="+num4.toString(16));

/\*

\* js中的特有语句 for in

\*

\* 格式：

\* for(变量 in 对象)//对对象进行遍历的语句。

\* {

\* }

\*/

**var** arr = [32,80,65];

**for**(i **in** arr){ println("i="+arr[i]);

}

println("<hr/>");

**for**(x **in** numObj){ println(x);

}

</script>

</body>

</html>

# 235.JS中自定义对象（类的定义方式）和集合容器演示：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Untitled Document</title>

</head>

<body>

<script type=*"text/javascript"* src=*"out.js"*></script>

<script type=*"text/javascript"*>

/\*

\* js自定义对象。

\*

\* 如果想要自定义对象，应该先对对象进行描述。

\* js是基于对象，不是面向对象的。不具备描述事物的能力。

\* 我们还想按照面向对象的思想编写js。

\* 就要先描述，在js中，可以用函数来模拟面对对象的中的描述。

\*/

/\*

//用js来描述人

function Person(){//相当于构造器。

// alert("person run");

}

//通过描述进行对象的建立。 new. var p = new Person();

//动态给p对象添加属性。直接使用p.属性名即可。

p.name = "zhangsan"; p.age = 29;

//如果定义的p对象的属性赋值为一个函数，即是给p对象添加一个方法。

p.show = function(){

alert("show :"+this.name+":"+this.age);

}

// p.show();

var obj = new Object();

obj.name = "god father"; obj.age = 2012;

// alert(obj.name+":"+obj.age);

\*/

**function** Person(name,age){

//在给Person对象添加了两个属性。

**this**.name = name;

**this**.age = age;

**this**.setName = **function**(name){ **this**.name = name;

}

**this**.getName = **function**(){ **return this**.name;

}

}

**var** p = **new** Person("旺财",20);

**for**(x **in** p){ println(x+":"+p[x]);

}

p.setName("小强");

// alert(p.name);

// alert(p.getName());

//直接使用{}定义属性和值的键值对方式，键值间通过:连接，键与键之间用逗号隔开。

**var** pp = {

//定义一些成员。 "name":"小明","age":38, "getName":**function**(){

**return this**.name;

}

}

//对象调用成员有两种方式：对象.属性名 对象["属性名"]

// alert(pp["age"]+":"+pp.name);

//用js实现键值对映射关系的集合容器。

**var** oMap = { 8:"小强",3:"旺财",7:"小明"

}

// var val1 = oMap[8];

// alert("val1:"+val1);

**var** val2 = get(7);

// alert("val2:"+val2);

**function** get(key){

**return** oMap[key];

}

// name="lisi";

// age = 30;

**var** myobj = { myname:"lisisi",myage:30

}

// alert(myobj.myname+":"+myobj["myage"]);

**var** myobj2 = { "myname2":"hahahah","myage2":48

}

// alert(myobj2.myname2+":"+myobj2["myage2"]);

**var** myMap = {

// names:["lsii1","lisi2","lisi4"],nums:[34,10,48]

names:[{name1:"zhangsan"},{myname:"hahahah"}]

}

// alert(myMap.names[1]);

// alert(myMap.names[0].name1);

</script>

</body>

</html>

## 236.JS前后知识串接： JavaScript是一种基亍对象，具有相对安全性的客户端脚本语言，是一种劢态、弱类型、基亍原型

的语言，内置支持类，不Java丌同的是JavaScript是基亍对象的语言而丌是面向对象的语言，在浏 览器客户端可以直接解释执行，而Java语言先经过编译再解释执行。

--------------------------------------------------------------------------------------------------

在HTML文档中揑入脚本语言可以使用<javascript>标记，并且可以将脚本语言单独写在扩展名 为.js的脚本文件中，然后在<javascript>标记中设置src属性值为脚本文件的位置和名称。

--------------------------------------------------------------------------------------------------

<javascript>标记使用要点：<javascript>标记可以置亍页面任何位置！<javascript>定义凼数一 般置亍<head>中；<javascript>可以直接写js语句。

--------------------------------------------------------------------------------------------------

发量的声明及赋值：JavaScript是一种弱类型的脚本语言；可以通过var关键字杢声明一个发量， 即在发量使用乊前必须加var声明，丌能使用未经声明的发量。

--------------------------------------------------------------------------------------------------

全尿发量：在方法外部声明的发量和在方法内部并没有加var关键字声明的发量都是全尿发量。 尿部发量：在方法内部，使用var声明的发量。 Javascript：void（0）；用亍叏消默认超链接的点击效果，Javascript伪协议。

--------------------------------------------------------------------------------------------------

发量的命名规则：发量命名必须以英文字母戒是下划线"\_"戒是美元符"$"开头；发量名的长度丌能 超过255个字符；发量名中丌允许使用空格；丌能使用脚本语言中保留的关键字及保留符号作为发 量名；发量名区分大小写（javascript是区分大小写的语言）。

--------------------------------------------------------------------------------------------------

数据类型：发量是没有类型的，数据是有类型的，常用的6种类型有一，undefined未定义（声明 了发量但从未赋值）；二，null空（赋值为空）；三，boolean布尔型（true、false）；四，字符串 String（单双引号均可）；五，数值number（包括整数和小数，NaN（Not a Number），Infinity，

-Infinity）；六，对象Object。

--------------------------------------------------------------------------------------------------

数 据 类 型 的 自 劢 转 化 ： 布 尔 型 <--> 字 符 串 <--> 数 字 ， 即 true<-->"true"<-->1 ，

false<-->"false"<-->0；数字<-->字符串，即1234<-->"1234"；null<-->布尔型<-->数字<--> 字符串，即<-->false<-->0<-->"null" ； undefined<--> 布尔型<--> 数字<--> 字符串， 即 undefined<-->false<-->0<-->"undefined"。

--------------------------------------------------------------------------------------------------

表达式不运算符：===是等同符，即丌会収生类型的自劢转化；==是等值符，即会収生类型的自 劢转化，自劢匹配。 赋值运算符：基本的赋值运算符为"="，用亍将右侧的操作数戒表达式的值赋给"="左侧的发量， 在程序中还会经常用到的扩展赋值运算符，如 +=，-=，\*=，/=，%=等。 字符串运算符：字符串运算符主要是字符串运算符"+"和它的发形形式即"+="，数字类型和字符串 迚行"+"操作时，实际是迚行了字符串的连接操作。 条件运算符：即（条件表达式1）？表达式2：表达式3，执行此表达式时，先判断条件即逻辑表达 式1的值，若为true，则整个三目运算的结果为表达式2的值，否则整个运算结果为表达式3的值。

--------------------------------------------------------------------------------------------------

分支语句：

一，if. . .else语句：if（条件）{

执行语句

}else{ 执行语句

}

二，switch语句可以根据一个发量的丌同叏值而采叏丌同的处理方法，在程序设计中提供丌同的分 支，即switch（expression）{

case const1：

语句块1

break； case const2：

语句块2 break；

. . .

default： 语句块N

}

弼同时可以使用switch和if语句时，建议使用switch语句，因为switch语句可以将答案一次性的全 部加载迚内存，效率更高。

三，for循环：for（初始化；条件；更新）{

语句块

}

四，while循环：while（条件）{

语句块；

}

五，do-while循环：do{

语句块；

}while（条件）； break语句是结束弼前的循环，并把程序的执行权交给循环的下一条语句，JS中是结束循环，循环 到此为止；continue语句是结束弼前的某一次循环，但是并没有跳出整个循环，在JS中是结束一 次循环，整个循环还在迚行。

--------------------------------------------------------------------------------------------------

数组：js数组更加类似java的容器，长度可发，元素类型也可以丌同！js数组长度随时可发，随时 可以修改，可以通过修改数组的长度清空数组，可以通过array.length获叏数组长度。

--------------------------------------------------------------------------------------------------

凼数及深化：一个典型的javascript凼数定义格式：

Function 凼数名（参数表）{

凼数执行部分

}

注意：参数列表直接写形参名即可，丌用写var。 return语句：return返回凼数的返回值并结束凼数运行，return语法为 return 表达式，弼使用 return语句的时候，要得到凼数的返回值，叧要利用凼数名对一个发量赋值就可以了，凼数在js中 也是对象！凼数的另一个定义方式即凼数可以作为参数传递。

--------------------------------------------------------------------------------------------------

内部凼数：使用eval（String str）杢执行javascript代码，eval参数字符串是和eval凼数上下文相 关的；parseInt（String str），parseFloat（String str）；isNaN（）；isFinite（）；escape（）， unescape（）。

--------------------------------------------------------------------------------------------------

Event对象：event代表事件的状态，例如触収event对象的元素、鼠标的位置及状态、按下的键等 等，event对象叧在事件収生的过程中才有效，event的某些属性叧对特定的事件有意义，比如 fromElement和toElement属性叧对onmouseover和onmouseout事件有意义。

--------------------------------------------------------------------------------------------------

对话框：警告框即alert，常用；询问框，返回提示框中的值；确认框，返回true/false。

--------------------------------------------------------------------------------------------------

常见内置对象乊时间对象：使用前必须先声明，即var curr=new Date（）；主要方法有getyear、

getmonth、setyear、setmonth、toLocaleString（）。

--------------------------------------------------------------------------------------------------

常见内置对象乊Math对象：内置的Math对象可以用杢处理各种数学运算，可以直接调用的方法 即Math.数学凼数（参数），求随机数方法即Math.random（），产生[0,1)范围的一个任意数。

--------------------------------------------------------------------------------------------------

常见内置对象乊String对象：一般利用String对象提供的方法杢处理字符串，String对字符串的处 理主要方法有charAt（index）：返回指定位置处的字符；indexOf（char）：返回指定子字符串的 位置，从左到右，找丌到就返回-1；substr（m，n）：返回给定字符串中从m位置开始，叏n个字 符，如果参数n省略，则意味着叏到字符串末尾；substring（m，n）：返回给定字符串中从m位 置开始，到n位置结束，如果参数n省略则意味着叏到字符串末尾；toLowerCase（）：将字符串中 的字符全部转化成小写；toUpperCase（）：将字符串中的字符全部转化成大写；length：属性， 丌是方法，返回字符串的长度。

--------------------------------------------------------------------------------------------------

JS中类的典型定义方式：function Person（name，age）{

this.name=name； this.age=age； this.study=function（）{

alert（"好好学习--我是："+this.name）；

}

}

var c=new Person（"aa"，18）；

--------------------------------------------------------------------------------------------------

继承的实现：JS中继承的实现即prototype原型继承

第十六章：DOM

237.DOM简述：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Untitled Document</title>

</head>

<body>

<!--

DOM:Document Object Model(文档对象模型)

用来将标记型文档封装成对象，并将标记型文档中的所有的内容(标签，文本，属性等)都封装成 对象。

封装成对象的目的是为了更为方便的操作这些文档以及文档中的所有内容。 因为对象的出现就可以有属性和行为被调用。

文档对象模型 文档：标记型文档。

对象：封装了属性和行为的实例，可以被直接调用。 模型：所有标记型文档都具备一些共性特征的一个体现。

标记型文档(标签，属性，标签中封装的数据). 只要是标记型文档，DOM这种技术都可以对其进行操作。 常见的标记型文档：html xml

DOM这种技术是如何对标记型文档机型操作的呢？ 要操作标记型文档必须对其进行解析。

DOM技术的解析方式：将标记型文档解析一棵DOM树，并将树中的内容都封装成节点对象。 注意：这个DOM解析方式的好处：可以对树中的节点进行任意操作，比如：增删改查。

弊端：这种解析需要将整个标记型文档加载进内存。意味着如果标记型文档的体积很大，较为浪费内存 空间。

简介另一种解析方式：SAX：是由一些组织定义的一种民间常用的解析方式，并不是w3c标准，而DOM 是W3C的标准。

SAX解析的方式：基于事件驱动的解析。获取数据的速度很快，但是不能对标记进行增删改。 DOM模型有三种：

DOM level 1：将html文档封装成对象。

DOM level 2：在leve 1基础上加入了新功能，比如解析名称空间。 DOM level 3：将xml文档封装成了对象。

DHTML:动态的HTML。不是一门语言：是多项技术综合体的简称。 其中包含了HTML,CSS,DOM,Javascript。

这四个技术在动态html页面效果定义时，都处于什么样角色呢？负责什么样的职责呢？

HTML:负责提供标签，对数据进行封装，目的是便于对该标签中的数据进行操作。 简单说：用标签封装数据。

CSS:负责提供样式属性，对标签中的数据进行样式的定义。 简单说：对数据进行样式定义

DOM:负责将标签型文档以及文档中的所有内容进行解析，并封装成对象，在对象中定义了更多的属性 和行为，便于对对象操作。

简单说：将文档和标签以及其他内容变成对象。

JS:负责提供程序设计语言，对页面中的对象进行逻辑操作。 简单说：负责页面的行为定义，就是页面的动态效果。

所以javascript是动态效果的主力编程语言。

DHTML+XMLhttpRequest = AJAX

BOM:Browser Object Model 浏览器对象模型。 这个模型方便于操作浏览器。

浏览器对应的对象就是window对象。这个可以通过查阅dhtml api获得。

-->

<!--html xmlns=["http:](http://www.sina.com.cn/myhtml/2)/[/www.sina.com.cn/myhtml/2](http://www.sina.com.cn/myhtml/2)">

<table></table>

</html>

<html xmln=["http://ww](http://www.huyouni.com/)w[.huyouni.com](http://www.huyouni.com/)">

<table></table>

</html-->

</body>

</html>

238.window对象演示：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Untitled Document</title>

</head>

<body>

<script type=*"text/javascript"* src=*"out.js"*></script>

<!--

为了演示方便，定义一个事件源。通过对事件源的触发，获取想要的结果。

比如让用户通过点击按钮就可以知道浏览器的一些信息。

-->

<!--

定义按钮 onclick事件的处理方式。

-->

<script type=*"text/javascript"*>

//定义一函数。

**function** windowObjDemo(){

//想要知道这个浏览器的信息,就需要使用window对象中的navigator

**var** name = navigator.appName;

**var** version = navigator.appVersion;

println(name+":"+version);

}

//演示location

**function** windowObjDemo2(){

// var pro = location.protocol;

// var text = location.href;

// alert(text);

//给location的href属性设置一个值。 改变了地址栏的值，并对其值进行了解析如果是

//http，还进行连接访问。

location.href = "[http://www.sina.com.cn"](http://www.sina.com.cn/);

}

</script>

<!--定义事件源,注册事件关联的动作-->

<input type=*"button"* value=*"演示window中的对象"* onclick="windowObjDemo2()" />

</body>

</html>

# 239.window对象中的方法（演示）：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Untitled Document</title>

</head>

<body>

<!--

演示window中的常见方法。

-->

<script type=*"text/javascript"* src=*"out.js"*></script>

<script type=*"text/javascript"*>

**var** timeid;

**function** windowMethodDemo(){

// var b = confirm("你真的确定要点击吗？");

// alert("b="+b);

// setTimeout("alert('time out run')",4000);

timeid = setInterval("alert('interval run')",3000);

}

**function** stopTime(){ clearInterval(timeid);

}

**function** windowMove(){

// moveBy(10,10);

// moveTo(40,40);

**for** (**var** x = 0; x < 700; x++){ moveBy(20, 0);

moveBy(0, 20);

moveBy(-20,0);

moveBy(0,-20);

}

}

**function** windowOpen(){

/\* open("ad.html","\_blank","height=400,width=400,status=no,toolbar=no,menubar=no, location=no");

\*/ c8lose();

}

</script>

<input type=*"button"* value=*"演示window对象的方法"* onclick="windowOpen()"/>

<input type=*"button"* value=*"停止"* onclick="stopTime()"/>

</body>

</html>

--------------------------------------------------------------------------------

Window中弹出的广告示例：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Untitled Document</title>

</head>

<body>

<h1>特价大优惠！</h1>

<h2>什么东西便宜了呢！！</h2>

<h1>你猜！</h1>

<script type=*"text/javascript"*>

// setTimeout("close()",3000);

// onunload = function(){

//

open("../demo/ad.html","\_blank","height=400,width=400,status=no,toolbar=no,me nubar=no,location=no");

// }

// setInterval("focus()",1000);

</script>

</body>

</html>

240.window常见事件：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Untitled Document</title>

</head>

<body>

<script type=*"text/javascript"*>

/\*

onunload = function(){ alert("onunload run");

}

onload = function(){ alert("onload run");

}

onbeforeunload = function(){ alert("onbeforeunload run");

}

\*/

onload = **function**(){

window.status = "啊！，加载完毕啦";

}

</script>

</body>

</html>

241.window演示弹窗广告：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Untitled Document</title>

</head>

<body>

<script type=*"text/javascript"*>

/\*

\* 演示弹窗广告效果。

\*

\* 页面一加载完就执行。

\*

\* 在当前页面定义脚本，可以在onload事件中完成广告的弹窗。

\*/

onload = **function**(){

open("../demo/ad.html","\_blank","height=400,width=400,status=no,toolbar=no,me nubar=no,location=no");

}

</script>

</body>

</html>

弹出的广告：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Untitled Document</title>

</head>

<body>

<h1>特价大优惠！</h1>

<h2>什么东西便宜了呢！！</h2>

<h1>你猜！</h1>

<script type=*"text/javascript"*>

// setTimeout("close()",3000);

// onunload = function(){

//

open("../demo/ad.html","\_blank","height=400,width=400,status=no,toolbar=no,me nubar=no,location=no");

// }

// setInterval("focus()",1000);

</script>

</body>

</html>

# 242.document对象+获叏节点三种方法演示：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Untitled Document</title>

</head>

<body>

<!--

document对象的演示。

该对象将标记型文档进行封装。 该对象的作用，是可以对标记型文档进行操作。

最常见的操作就是，想要实现动态效果，需要对节点操作，那么要先获取到这个节点。 要想获取节点，必须要先获取到节点所属的文档对象document。 所以document对象最常见的操作就是获取页面中的节点。

获取节点的方法体现： 1，getElementById():通过标签的id属性值获取该标签节点。返回该标签节点。 2，getElementsByName(): 通过标签的name属性获取节点，因为name有相同，所以返回的一个数 组。

3，getElementsByTagName(): 通过标签名获取节点。因为标签名会重复，所以返回的是一个数组。

凡是带s返回的都是数组。

-->

<script type=*"text/javascript"*>

**function** getNodeDemo(){

/\*

\* 需求：获取页面中的div节点。

\* 思路：

\* 通过docment对象完成。因为div节点有id属性。 所以可以通过id属性来完成获取。

\*

\*/

**var** divNode = document.getElementById("divid");

//节点都有三个必备的属性。节点名称，节点类型，节点值。

// alert(divNode.nodeName+":"+divNode.nodeType+":"+divNode.nodeValue);

/\*

\* 常见节点有三种：

\* 1，标签型节点：类型 1

\* 2，属性节点： 类型 2

\* 3，文本节点。 类型 3

\*

\* 标签型节点是没有值的，属性和文本节点是可以有值的。

\*/

//获取div节点中的文本。

// var text = divNode.innerHTML;

// alert(text);

//改变div中的文本。

divNode.innerHTML = "哈哈，文本被我改掉了".fontcolor("red");

}

//获取文本框节点演示getElementsByName方法

**function** getNodeDemo2(){

// var nodes = document.getElementsByName("user");

// alert(nodes[0].value);

**var** userNode = document.getElementsByName("user")[0]; alert(userNode.value);

}

//获取超链接节点对象。演示getElementsByTagName方法。

**function** getNodeDemo3(){

//直接用标签名获取。

**var** nodes = document.getElementsByTagName("a");

// alert(nodes.length);

// alert(nodes[0].innerHTML);

**for**(**var** x=0; x<nodes.length; x++){

// alert(nodes[x].innerHTML); nodes[x].target = "\_blank";

}

}

/\*

\* 对于页面中的超链接，新闻链接通过新窗口打开，门户网站链接在当前页面打开。

\* 当然是要获取其中被操作的超链接对象啊。

\* 可是通过document获取超链接，拿到的是页面中所有的超链接节点。

\* 只想获取一部分该如何办呢？

\* 只要获取到被操作的超链接所属的节点即可。

\*

\* 在通过这个节点获取到它里面所有的超链接节点。

\*/

**function** getNodeDemo4(){

//获取超链接所属的div节点

**var** divNode = document.getElementById("newslink");

//通过对div对象方法的查找发现它也具备getElementsByTagName方法。

//记住：所有的容器型标签都具备这个方法。在该标签范围内获取指定名称的标签。

**var** aNodes = divNode.getElementsByTagName("a");

**for**(**var** x=0;x<aNodes.length; x++){

// alert(aNodes[x].innerHTML); aNodes[x].target = "\_blank";

}

}

</script>

<input type=*"button"* value=*"演示document对象获取节点"* onclick="getNodeDemo4()"

/>

<div id=*"divid"*>这是一个div区域</div>

<input type=*"text"* name=*"user"* />

<a href=*"*[*http://www.sina.com.cn"*](http://www.sina.com.cn/)>新浪网站11</a>

<a href=*"*[*http://www.SOHU.com.cn"*](http://www.SOHU.com.cn/)>新浪网站22</a>

<div id=*"newslink"*>

<a href=*"*[*http://www.163.com.cn*](http://www.163.com.cn/)*"*>新闻链接1</a>

<a href=*"*[*http://www.164.com.cn*](http://www.164.com.cn/)*"*>新闻链接2</a>

<a href=*"*[*http://www.itcast.com.cn"*](http://www.itcast.com.cn/)>新闻链接3</a>

</div>

</body>

</html>

# 243.document对象通过节点的局次关系获叏节点对象

的方法演示：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Untitled Document</title>

</head>

<body>

<!--

通过节点的层次关系获取节点对象。 关系：

1，父节点: parentNode:对应一个节点对象。 2，子节点：childNodes:对应一个节点集合。 3，兄弟节点：

上一个兄弟节点：previousSibling 下一个兄弟节点：nextSibling

-->

<script type=*"text/javascript"*>

**function** getNodeByLevel(){

//获取页面中的表格节点。

**var** tabNode = document.getElementById("tabid");

//获取父节点。parentNode

// var node = tabNode.parentNode;

// alert(node.nodeName);//body

//获取子节点。childNodes

// var nodes = tabNode.childNodes;

// alert(nodes[0].childNodes[0].nodeName);

//获取兄弟节点。

//上一个。

// var node = tabNode.previousSibling;

// alert(node.nodeName);

// var node = tabNode.nextSibling.nextSibling;

// alert(node.nodeName);

// 尽量少用兄弟节点，因为在解析的时候会出现浏览器不同解析不一致，

// 会解析出标签间的空白文本节点

}

</script>

<input type=*"button"* value=*"通过节点层次关系获取节点"* onclick="getNodeByLevel()"

/>

<div>div区域</div>

<span>span区域</span>

<table id=*"tabid"*>

<tr>

<td>单元格一</td>

<td>单元格二</td>

</tr>

</table>

<span>span区域11</span>

<dl>

<dt>上层项目</dt>

<dd>下层项目</dd>

</dl>

<a href=*''*>一个超链接</a>

</body>

</html>

# 244.document对象对节点元素的增删改查（演示）：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Untitled Document</title>

<style type=*"text/css"*> div{

border:*#00ccff 1px solid*; width:*200px*; padding:*30px*; margin:*10px*;

}

#div\_1{

background-color:*#00ccff*;

}

#div\_2{

background-color:*#FFccff*;

}

#div\_3{

background-color:*#cc00ff*;

}

#div\_4{

background-color:*#00FF00*;

}

</style>

</head>

<body>

<script type=*"text/javascript"*>

//创建并添加节点。

//需求：在div\_1节点中添加一个文本节点。

**function** crtAndAdd(){

/\*

\* 1，创建一个文本节点。

\* 2，获取div\_1节点。

\* 3，将文本节点添加到div\_1节点中。

\*/

//1, 创建节点。使用document中的createTextNode方法。

**var** oTextNode = document.createTextNode("这是一个新的文本 ");

//2，获取div\_1节点。

**var** oDivNode = document.getElementById("div\_1");

//3，将文本节点添加到div\_1节点中。 oDivNode.appendChild(oTextNode);

}

//需求：在div\_1中创建并添加一个按钮节点。

**function** crtAndAdd2(){

//1,创建一个按钮节点。

**var** oButNode = document.createElement("input"); oButNode.type = "button";

oButNode.value = "一个新按钮";

//2，获取div\_1节点。

**var** oDivNode = document.getElementById("div\_1");

//3，将按钮节点添加到div\_1节点中。 oDivNode.appendChild(oButNode);

}

//通过另一种方式完成添加节点。

**function** crtAndAdd3(){

/\*

\* 其实是使用了容器型标签中的一个属性。innerHTML.

\* 这个属性可以设置html文本。

\*/

//1，获取div\_1节点。

**var** oDivNode = document.getElementById("div\_1");

// oDivNode.innerHTML = "<input type='button' value='有个按钮'/>"; oDivNode.innerHTML = "<a href='http://www.163.com'>有个超链接</a>";

}

// =======================

//需求将div\_2节点删除。

**function** delNode(){

//1,获取div\_2节点。

**var** oDivNode = document.getElementById("div\_2");

//2,使用div节点的remvoeNode方法删除。

// oDivNode.removeNode(true);较少用。

//一般使用removeChild方法。 删除子节点。

//获取div\_2的父节点，然后在用父节点的removeChild，将div\_2删除。 oDivNode.parentNode.removeChild(oDivNode);

}

//需求：用div\_3节点替换div\_1节点。

**function** updateNode(){

/\*

\* 获取div-3和div-1.

\* 使用replaceNode进行替换。

\*/

**var** oDivNode\_1 = document.getElementById("div\_1");

**var** oDivNode\_3 = document.getElementById("div\_3");

// oDivNode\_1.replaceNode(oDivNode\_3);

// 建议使用replaceChild方法。

oDivNode\_1.parentNode.replaceChild(oDivNode\_3,oDivNode\_1);

}

//需求：希望用div\_3替换div-1，并保留div-3节点。

//其实就是对div\_3进行克隆

**function** cloneDemo(){

**var** oDivNode\_1 = document.getElementById("div\_1"); **var** oDivNode\_3 = document.getElementById("div\_3"); **var** oCopyDiv\_3 = oDivNode\_3.cloneNode(**true**);

// oDivNode\_1.replaceNode(oDivNode\_3);

// 建议使用replaceChild方法。

oDivNode\_1.parentNode.replaceChild(oCopyDiv\_3,oDivNode\_1);

}

// firstNode = childNodes[0];

// lastNode = childNodes[childeNodes.length-1];

</script>

<input type=*"button"* value=*"创建并添加节点"* onclick="crtAndAdd3()"/>

<input type=*"button"* value=*"删除节点"* onclick="delNode()"/>

<input type=*"button"* value=*"修改节点"* onclick="updateNode()"/>

<input type=*"button"* value=*"克隆节点"* onclick="cloneDemo()"/>

<hr/>

<div id=*"div\_1"*>

</div>

<div id=*"div\_2"*>

好好学习，day day up！

</div>

<div id=*"div\_3"*> div区域演示文字

</div>

<div id=*"div\_4"*>

节点的增删改查

</div>

</body>

</html>

245.DOM示例-新闻字体：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Untitled Document</title>

<style type=*"text/css"*>

/\*超链接访问前，访问后样式一致\*/ a:link,a:visited{

color:*#0044ff*;

text-decoration:*none*;

}

a:hover{

color:*#ff8800*;

}

#newstext{

width:*500px*; border:*#00ff00 1px solid*; padding:*10px*;

}

/\*预定义一些样式封装到选择器。一般使用类选择器。\*/

.norm{

color:*#000000*; font-size:*16px*;

background-color:*#cdd8d0*;

}

.max{

color:*#808080*; font-size:*28px*;

background-color:*#9ce9b4*;

}

.min{

color:*#ffffff*; font-size:*12px*;

background-color:*#f1b96d*;

}

</style>

</head>

<body>

<!--

需求：新闻字体的大中小样式改变。 思路： 1，先有新闻数据，并用标签封装。

2，定义一些页面样式。静态的样式和预定义的样式(用于动态效果，方便于事件处理时的样式加载。)

3，确定事件源和事件，以及处理方式中被处理的节点。 事件源：a标签，事件：onclick。

被处理的节点 div-newstext

既然要给超链接加入自定义的事件处理，就要先取消超链接的默认点击效果。 可以使用给href设置：javascript:void(0)来完成。

4，用js处理页面的行为。

-->

<script type=*"text/javascript"*>

//定义改变 字体的方法。

**function** changeFont2(size,clr){

/\*

\* 既然要对div newstext中的文本字体进行操作。

\* 必须要先获取div节点对象。

\*

\*/

**var** oNewsText = document.getElementById("newstext");

//获取oNewText节点的style样式属性对象。 oNewsText.style.fontSize = size; oNewsText.style.color = clr;

}

/\*

\* 如果根据用户点击所需要的效果不唯一。

\* 仅通过传递多个参数虽然可以实现效果，

\* 但是

\* 1，传参过多，阅读性差，

\* 2，js代码和css代码耦合性高。

\* 3，不利于扩展。

\*

\* 解决：

\* 将多个所需的样式进行封装。

\* 封装到选择器中，只要给指定的标签加载不同的而选择器就可以了。

\*/

**function** changeFont(selectorName){

**var** oNewsText = document.getElementById("newstext"); oNewsText.className = selectorName;

}

</script>

<h1>这是一个新闻标题</h1>

<hr/>

<a href=*"javascript:void(0)"* onclick="changeFont('max')">大字体</a>

<a href=*"javascript:void(0)"* onclick="changeFont('norm')">中字体</a>

<a href=*"javascript:void(0)"* onclick="changeFont('min')">小字体</a>

<div id=*"newstext"* class=*"norm"*> 这是新闻内容这是新闻内容这是新闻内容这是新闻内容这是新闻内容这是新闻内容 这是新闻内容这是新闻内容这是新闻内容这是新闻内容这是新闻内容这是新闻内容 这是新闻内容这是新闻内容这是新闻内容这是新闻内容这是新闻内容这是新闻内容 这是新闻内容这是新闻内容这是新闻内容这是新闻内容这是新闻内容这是新闻内容 这是新闻内容这是新闻内容这是新闻内容这是新闻内容这是新闻内容这是新闻内容 这是新闻内容这是新闻内容这是新闻内容这是新闻内容这是新闻内容这是新闻内容 这是新闻内容这是新闻内容这是新闻内容这是新闻内容这是新闻内容这是新闻内容 这是新闻内容这是新闻内容这是新闻内容这是新闻内容这是新闻内容这是新闻内容 这是新闻内容这是新闻内容这是新闻内容这是新闻内容这是新闻内容这是新闻内容 这是新闻内容这是新闻内容这是新闻内容这是新闻内容这是新闻内容这是新闻内容 这是新闻内容这是新闻内容这是新闻内容这是新闻内容这是新闻内容这是新闻内容 这是新闻内容这是新闻内容这是新闻内容这是新闻内容这是新闻内容这是新闻内容

</div>

</body>

</html>

246.DOM示例-展开闭合列表：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Untitled Document</title>

<style type=*"text/css"*> dl dd{

margin:*0px*;

}

dl{

}

/\*

height:*16px*;

预定义一些样式，类选择器，便于给标签进行样式的动态加载。

\*/

.open{

overflow:*visible*;;

}

.close{

overflow:*hidden*;

}

dl dt{

color:*#ff8000*;

}

</style>

</head>

<body>

<!--

思路： 1,标签封装数据，html 2,定义样式 css

3,明确事件源，事件，以及要处理节点，dom。

4,明确具体的操作方式，其实就是事件的处理内容。 js。

-->

<script type=*"text/javascript"*>

**var** flag = **true**;

//列表展开闭合效果。

**function** list2(){

/\*

\* 思路：

* 1,无非就是将dl属性的overflow的值改成visible即可。

\* 2,要先获取dl节点。

* 3,改变该节点的style.overflow的属性。

\*/

//1，获取dl节点。

**var** oDlNode = document.getElementsByTagName("dl")[0];

//2,修改style样式属性中的overflow属性值。

// oDlNode.style.overflow = "visible";

**if**(flag){

oDlNode.style.overflow = "visible"; flag = **false**;

}**else**{

oDlNode.style.overflow = "hidden"; flag = **true**;

}

}

//重新定义list。降低js和css的耦合性。

**function** list3(node){

// alert(node.nodeName);//dt

//1，获取dl节点。

// var oDlNode = document.getElementsByTagName("dl")[0];

**var** oDlNode = node.parentNode;

// alert(oDlNode.nodeName);

//2，设置dl的className属性值。

// oDlNode.className = "open";

**if**(oDlNode.className == "close"){ oDlNode.className = "open";

}**else**{

oDlNode.className = "close";

}

}

/\*

\* 在多个列表下，一次只需要展开一个列表其他列表都关闭。

\* 怎么保证开一个，而关其他呢？

\* 思路：

\* 1，获取所有的dl节点。

\* 2，遍历这些节点，只对当前的dl进行展开或者闭合操作，其他节点都进行闭合操作。

\*/

**function** list(node){

//获取对当前的dl节点。

**var** oDlNode = node.parentNode;

//获取所有的dl节点。

**var** collDlNodes = document.getElementsByTagName("dl");

//遍历所有dl。

**for**(**var** x=0; x<collDlNodes.length; x++){

**if**(collDlNodes[x]==oDlNode){ **if**(oDlNode.className == "close"){

oDlNode.className = "open";

}**else**{

oDlNode.className = "close";

}

}**else**{

collDlNodes[x].className = "close";

}

}

}

</script>

<dl class=*"close"*>

<dt onclick="list(this)">显示条目一</dt>

<dd>展开列表内容1</dd>

<dd>展开列表内容1</dd>

<dd>展开列表内容1</dd>

<dd>展开列表内容1</dd>

</dl>

<dl class=*"close"*>

<dt onclick="list(this)">显示条目二</dt>

<dd>展开列表内容2</dd>

<dd>展开列表内容2</dd>

<dd>展开列表内容2</dd>

<dd>展开列表内容2</dd>

</dl>

<dl class=*"close"*>

<dt onclick="list(this)">显示条目三</dt>

<dd>展开列表内容3</dd>

<dd>展开列表内容3</dd>

<dd>展开列表内容3</dd>

<dd>展开列表内容3</dd>

</dl>

</body>

</html>

247.DOM示例-好友列表：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Untitled Document</title>

<style type=*"text/css"*>

/\* 对表格中的ul进行样式定义， 1，去除无序列表的样式。

2，将列表的外边距取消。

\*/

table ul{

list-style:*none*; margin:*0px*;

background-color:*#26e356*; display:*none*;

/\*border:#ff0000 1px solid;\*/ padding:*0px*;

}

/\*

\* 对表格框线进行定义。

\* 以及单元格的框线进行定义。

\*/ table{

border:*#8080ff 1px solid*; width:*80px*;

}

table td{

border:*#8080ff 1px solid*; background-color:*#f3c98f*; padding:*0px*;

}

/\*

\* 单元格中的超链接样式。

\*/

table td a:link,table td a:visited{ color:*#1f6df1*;

text-decoration:*none*;

}

table td a:hover{ color:*#df4011*;

}

/\*预定一些样式\*/

.open{

display:*block*;

}

.close{

display:*none*;

}

</style>

</head>

<body>

<!--

完成一个好友菜单. 展开闭合效果。 而且要求一次开一个而关其他。

-->

<script type=*"text/javascript"*>

**function** list(node){

//获取被操作的节点ul。

/\*

\* 先通过事件源超链接标签获取其父节点td，然后在通过父节点获取ul节点。

\*/

**var** oTdNode = node.parentNode;

**var** oUlNode = oTdNode.getElementsByTagName("ul")[0];

//获取表格中所有的ul。

//先获取表格节点对象。

**var** oTabNode = document.getElementById("goodlist");

**var** collUlNodes = oTabNode.getElementsByTagName("ul");

**for**(**var** x=0; x<collUlNodes.length; x++){

**if**(collUlNodes[x]==oUlNode){

**if**(oUlNode.className == "open"){ oUlNode.className = "close";

}**else**{

oUlNode.className = "open";

}

}**else**{

collUlNodes[x].className = "close";

}

}

// alert(oUlNode.nodeName);//oUlNode.className

}

</script>

<table id=*"goodlist"*>

<tr>

<td>

<a href=*"javascript:void(0)"* onclick="list(this)">好友菜单</a>

<ul>

<li>一个好友</li>

<li>一个好友</li>

<li>一个好友</li>

<li>一个好友</li>

</ul>

</td>

</tr>

<tr>

<td>

<a href=*"javascript:void(0)"* onclick="list(this)">好友菜单</a>

<ul>

<li>一个好友</li>

<li>一个好友</li>

<li>一个好友</li>

<li>一个好友</li>

</ul>

</td>

</tr>

<tr>

<td>

<a href=*"javascript:void(0)"* onclick="list(this)">好友菜单</a>

<ul>

<li>一个好友</li>

<li>一个好友</li>

<li>一个好友</li>

<li>一个好友</li>

</ul>

</td>

</tr>

<tr>

<td>

<a href=*"javascript:void(0)"* onclick="list(this)">好友菜单</a>

<ul>

<li>一个好友</li>

<li>一个好友</li>

<li>一个好友</li>

<li>一个好友</li>

</ul>

</td>

</tr>

</table>

</body>

</html>

# 248.DOM示例-创建表格（增删行列）：

Table.css文件： table{

border:*#249bdb 1px solid*; width:*500px*;

border-collapse:*collapse*;

}

table td{

border:*#249bdb 1px solid*; padding:*10px*;

}

table th{

border:*#249bdb 1px solid*; padding:*10px*;

background-color: *rgb(200,200,200)*;

}

创建表格源码：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Untitled Document</title>

<link rel=*"stylesheet"* type=*"text/css"* href=*"table.css"* />

<style type=*"text/css"*>

/\*@import url("table.css"); @错误提示？？\*/

</style>

</head>

<body>

<!--

在页面中创建一个5行6列的表格。

1，事件源，比如按钮。

2，必须有一个生成的表格节点存储位置。

-->

<script type=*"text/javascript"*>

**function** crtTable2(){

/\*

\* 思路：

\* 可以通过之前学过的createElement创建元素的形式来完成。

\*/

//1,创建表格节点。

**var** oTabNode = document.createElement("table");

//2,创建tBody节点。

**var** oTbdNode = document.createElement("tbody");

//3，创建行节点tr。

**var** oTrNode = document.createElement("tr");

//4，创建单元格节点 td

**var** oTdNode = document.createElement("td");

oTdNode.innerHTML = "这是单元格";

//让这些节点有关系。进行指定层次的节点添加。 oTrNode.appendChild(oTdNode);

oTbdNode.appendChild(oTrNode); oTabNode.appendChild(oTbdNode);

document.getElementsByTagName('div')[0].appendChild(oTabNode);

}

/\*

\* 上面的方法很麻烦。

\* 既然操作的是表格，

\* 那么最方便的方式就是使用表格节点对象的方法来完成。

\*

\* 表格是由行组成。表格节点对象中 insertRow方法就完成了创建行并添加的动作。

\*

\* 行是由单元格组成。通过tr节点对象的insertCell来完成。

\*/

**function** crtTable(){

**var** oTabNode = document.createElement("table");

// oTabNode.id = "tabid"; oTabNode.setAttribute("id","tabid");

**var** rowVal = document.getElementsByName("rownum")[0].value;

**var** colVal = document.getElementsByName("colnum")[0].value;

**for** (**var** x = 1; x <= rowVal; x++) { **var** oTrNode = oTabNode.insertRow(); **for** (**var** y = 1; y <= colVal; y++) {

**var** oTdNode = oTrNode.insertCell(); oTdNode.innerHTML = x+"--"+y;

}

}

//将表格节点添加到div中。 document.getElementsByTagName('div')[0].appendChild(oTabNode);

document.getElementsByName("crtBut")[0].disabled = **true**;

}

//删除行。

**function** delRow(){

//获取表格对象。

**var** oTabNode = document.getElementById("tabid");

**if**(oTabNode==**null**){ alert("表格不存在！"); **return**;

}

//获取要删除的表格行数。

**var** rowVal = document.getElementsByName("delrow")[0].value;

**if**(rowVal>=1 && rowVal<=oTabNode.rows.length) oTabNode.deleteRow(rowVal-1);

### else

alert("要删除的行不存在！");

}

//删除列，其实就是在删除每一行中同一位置的单元格。

**function** delCol(){

**var** oTabNode = document.getElementById("tabid");

**if**(oTabNode==**null**){ alert("表格不存在！"); **return**;

}

**var** colVal = document.getElementsByName("delcol")[0].value;

**if** (colVal >= 1 && colVal <= oTabNode.rows[0].cells.length) {

**for** (**var** x = 0; x < oTabNode.rows.length; x++) { oTabNode.rows[x].deleteCell(colVal - 1);

}

}**else**{

alert("要删除的列不存在！");

}

}

</script>

行：<input type=*"text"* name=*"rownum"* />列：<input type=*"text"* name=*"colnum"* />

<input type=*"button"* value=*'创建表格'* name=*"crtBut"* onclick="crtTable()"/>

<hr/>

<input type=*"text"* name=*"delrow"* /><input type=*"button"* value=*"删除行"* onclick="delRow()"/><br/>

<input type=*"text"* name=*"delcol"* /><input type=*"button"* value=*"删除列"*

onclick="delCol()"/>

<hr/>

<div></div>

</body>

</html>

# 249.DOM示例-创建表格高亮背景色：

table.css文件： table{

border:*#249bdb 1px solid*; width:*500px*;

border-collapse:*collapse*;

}

table td{

border:*#249bdb 1px solid*; padding:*10px*;

}

table th{

border:*#249bdb 1px solid*; padding:*10px*;

background-color: *rgb(200,200,200)*;

}

表格设计源代码：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<link rel=*"stylesheet"* type=*"text/css"* href=*"table.css"*/>

<title>Untitled Document</title>

<style type=*"text/css"*>

.one{

background-color:*#e1e16a*;

}

.two{

background-color:*#75f094*;

}

.over{

background-color:*#f9360d*;

}

</style>

<script type=*"text/javascript"*>

**var** name;

**function** trColor(){

/\*

\* 思路：

\* 1，因为要操作行的样式，所以要先获取表格对象。

\* 2，获取所有被操作的行对象。

\* 3，遍历行并给每一行指定样式。

\*/

**var** oTabNode = document.getElementById("info");

// alert(oTabNode);

//表格中所有的行。

**var** collTrNodes = oTabNode.rows;

//遍历的时候从第二行遍历。

**for**(**var** x=1; x<collTrNodes.length; x++){

**if**(x%2==1){

collTrNodes[x].className = "one";

}**else**{

collTrNodes[x].className = "two";

}

// 给每一个行对象都添加两个事件。

collTrNodes[x].onmouseover = **function**(){ name = **this**.className; **this**.className = "over";

}

collTrNodes[x].onmouseout = **function**(){ **this**.className = name;

}

}

}

onload = **function**(){ trColor();

}

/\*

function over(node){

name = node.className; node.className = "over";

}

function outa(node){ node.className = name;

}

\*/

</script>

</head>

<body>

<table id=*"info"*>

<tr>

<th>姓名</th>

<th>年龄</th>

<th>地址</th>

</tr>

<tr>

<td>张三</td>

<td>27</td>

<td>上海</td>

</tr>

<tr>

<td>小强</td>

<td>26</td>

<td>北京</td>

</tr>

<tr>

<td>李四</td>

<td>29</td>

<td>广州</td>

</tr>

<tr>

<td>孙八</td>

<td>23</td>

<td>南京</td>

</tr>

<tr>

<td>二麻子</td>

<td>19</td>

<td>大连</td>

</tr>

<tr>

<td>犀利姐</td>

<td>32</td>

<td>青岛</td>

</tr>

<tr>

<td>旺财</td>

<td>19</td>

<td>深圳</td>

</tr>

<tr>

<td>周七</td>

<td>42</td>

<td>铁岭</td>

</tr>

<tr>

<td>毕姥爷</td>

<td>18</td>

<td>沈阳</td>

</tr>

</table>

</body>

</html>

250.DOM示例-创建表格并排序：

table.css文件： table{

border:*#249bdb 1px solid*; width:*500px*;

border-collapse:*collapse*;

}

table td{

border:*#249bdb 1px solid*; padding:*10px*;

}

table th{

border:*#249bdb 1px solid*; padding:*10px*;

background-color: *rgb(200,200,200)*;

}

表格排序源代码：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Untitled Document</title>

<link rel=*"stylesheet"* type=*"text/css"* href=*"table.css"*/>

<style type=*"text/css"*>

th a:link,th a:visited{ color:*#279bda*;

text-decoration:*none*;

}

</style>

<script type=*"text/javascript"*>

**var** flag = **true**; **function** sortTable(){

/\*

\* 思路：

\* 1，排序就需要数组。获取需要参与排序的行对象数组。

\* 2，对行数组中的每一个行的年龄单元格的数据进行比较，并完成行对象在数组中的位置

\* 置换。

\* 3，将排好序的数组重新添加回表格。

\*/

**var** oTabNode = document.getElementById("info");

**var** collTrNodes = oTabNode.rows;

//定义一个临时容器，存储需要排序行对象。

**var** trArr = [];

//遍历原行集合，并将需要排序的行对象存储到临时容器中。

**for**(**var** x=1; x<collTrNodes.length; x++){ trArr[x-1] = collTrNodes[x];

}

//对临时容器排个序。 mySort(trArr);

//将排完序的行对象添加会表格。

**if** (flag) {

**for** (**var** x = 0; x < trArr.length; x++) {

//oTabNode.childNodes[0].appendChild(trArr[x]);

// alert(trArr[x].cells[0].innerHTML); trArr[x].parentNode.appendChild(trArr[x]);

}

flag = **false**;

}**else**{

**for** (**var** x = trArr.length-1; x >=0; x--) { trArr[x].parentNode.appendChild(trArr[x]);

}

flag = **true**;

}

// alert("over");

}

**function** mySort(arr){

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

**for**(**var** y=x+1; y<arr.length; y++){

//按照年龄数值排序，并转成整数。

**if**(parseInt(arr[x].cells[1].innerHTML)>parseInt(arr[y].cells[1].innerHTML)){ **var** temp = arr[x];

arr[x] = arr[y]; arr[y] = temp;

// arr[x].swapNode(arr[y]);

}

}

}

}

</script>

</head>

<body>

<table id=*"info"*>

<tr>

<th>姓名</th>

<th><a href=*"javascript:void(0)"* onclick="sortTable()">年龄</a></th>

<th>地址</th>

</tr>

<tr>

<td>张三</td>

<td>27</td>

<td>上海</td>

</tr>

<tr>

<td>小强</td>

<td>6</td>

<td>北京</td>

</tr>

<tr>

<td>李四</td>

<td>29</td>

<td>广州</td>

</tr>

<tr>

<td>孙八</td>

<td>23</td>

<td>南京</td>

</tr>

<tr>

<td>二麻子</td>

<td>24</td>

<td>大连</td>

</tr>

<tr>

<td>犀利姐</td>

<td>32</td>

<td>青岛</td>

</tr>

<tr>

<td>旺财</td>

<td>19</td>

<td>深圳</td>

</tr>

<tr>

<td>周七</td>

<td>42</td>

<td>铁岭</td>

</tr>

<tr>

<td>毕姥爷</td>

<td>18</td>

<td>沈阳</td>

</tr>

</table>

</body>

</html>

251.DOM示例-全选商品列表：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Untitled Document</title>

</head>

<body>

<script type=*"text/javascript"*>

**function** getSum(){

/\*

\* 获取所有的名称为item的复选框。

\* 判断checked状态，为true表示被选中，获取该节点的value进行累加。

\*/

**var** sum = 0;

**var** collItemNodes = document.getElementsByName("item");

**for**(**var** x=0; x<collItemNodes.length; x++){

**if**(collItemNodes[x].checked){

sum += parseInt(collItemNodes[x].value);

}

}

**var** sSum = sum+"元";

document.getElementById("sumid").innerHTML = sSum.fontcolor('red');

}

//全选。

**function** checkAll(node){

/\*

\* 将全选的box的checked状态赋值给所有的itembox的checked。

\*/

**var** collItemNodes = document.getElementsByName("item");

**for**(**var** x=0; x<collItemNodes.length; x++){ collItemNodes[x].checked = node.checked;

}

}

</script>

<input type=*"checkbox"* name=*"allitem"* onclick="checkAll(this)" />全选<br/>

<input type=*"checkbox"* name=*"item"* value=*"4000"* />笔记本电脑：4000元<br/>

<input type=*"checkbox"* name=*"item"* value=*"4000"* />笔记本电脑：4000元<br/>

<input type=*"checkbox"* name=*"item"* value=*"4000"* />笔记本电脑：4000元<br/>

<input type=*"checkbox"* name=*"item"* value=*"4000"* />笔记本电脑：4000元<br/>

<input type=*"checkbox"* name=*"item"* value=*"4000"* />笔记本电脑：4000元<br/>

<input type=*"checkbox"* name=*"item"* value=*"4000"* />笔记本电脑：4000元<br/>

<input type=*"checkbox"* name=*"item"* value=*"4000"* />笔记本电脑：4000元<br/>

<input type=*"checkbox"* name=*"item"* value=*"4000"* />笔记本电脑：4000元<br/>

<input type=*"checkbox"* name=*"allitem"* onclick="checkAll(this)" />全选<br/>

<input type=*"button"* value=*" 总 金 额 是 "* onclick="getSum()"/><span id=*"sumid"*></span>

</body>

</html>

252.DOM示例-邮件列表：

table.css文件： table{

border:*#249bdb 1px solid*; width:*500px*;

border-collapse:*collapse*;

}

table td{

border:*#249bdb 1px solid*; padding:*5px*;

}

table th{

border:*#249bdb 1px solid*; padding:*5px*;

background-color: *rgb(200,200,200)*;

}

邮件列表设计代码：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Untitled Document</title>

<link rel=*"stylesheet"* type=*"text/css"* href=*"table.css"*/>

<style type=*"text/css"*>

.one{

background-color:*#9bf7d5*;

}

.two{

background-color:*#f3e99a*;

}

.over{

background-color:*#ef7125*;

}

</style>

<script type=*"text/javascript"*>

//行颜色间隔显示功能。

**var** name;

**function** trColor(){

//1,获取表格对象。

**var** oTabNode = document.getElementById("mailtable");

//2，获取行对象。

**var** collTrNodes = oTabNode.rows;

//3，对所有需要设置背景的行对象进行遍历。

**for**(**var** x=1; x<collTrNodes.length-1;x++){

**if**(x%2==1){

collTrNodes[x].className = "one";

}**else**{

collTrNodes[x].className = "two";

}

collTrNodes[x].onmouseover = **function**(){ name = **this**.className; **this**.className = "over";

}

collTrNodes[x].onmouseout = **function**(){ **this**.className = name;

}

}

}

onload = **function**(){ trColor();

}

//复选框的全选动作。

**function** checkAll(node){

//获取所有的mail复选框。

**var** collMailNodes = document.getElementsByName("mail");

**for**(**var** x=0; x<collMailNodes.length; x++){

collMailNodes[x].checked = node.checked;

}

}

//定义操作复选框按钮的方法。

**function** checkAllByBut(num){

**var** collMailNodes = document.getElementsByName("mail");

**for**(**var** x=0; x<collMailNodes.length; x++){

**if**(num>1)

collMailNodes[x].checked = !collMailNodes[x].checked;

### else

collMailNodes[x].checked = num;

}

}

//删除所选邮件。

**function** deleteMail(){

**if** (!confirm("你真的要删除所选的邮件吗？")){

### return;

}

//获取所有的mail节点

**var** collMailNodes = document.getElementsByName("mail");

**for** (**var** x = 0; x < collMailNodes.length; x++) {

**if** (collMailNodes[x].checked) {

**var** oTrNode = collMailNodes[x].parentNode.parentNode; oTrNode.parentNode.removeChild(oTrNode);

x--;

}

}

trColor();

}

</script>

</head>

<body>

<table id=*"mailtable"*>

<tr>

<th>

<input type=*"checkbox"* name=*"all"* onclick="checkAll(this)" />全选

</th>

<th>

发件人

</th>

<th>

邮件名称

</th>

<th>

邮件附属信息

</th>

</tr>

<tr>

<td>

<input type=*"checkbox"* name=*"mail"* />

</td>

<td>

张三11

</td>

<td>

我是邮件11

</td>

<td>

我是附属信息11

</td>

</tr>

<tr>

<td>

<input type=*"checkbox"* name=*"mail"* />

</td>

<td>

张三22

</td>

<td>

我是邮件22

</td>

<td>

我是附属信息22

</td>

</tr>

<tr>

<td>

<input type=*"checkbox"* name=*"mail"* />

</td>

<td>

张三33

</td>

<td>

我是邮件33

</td>

<td>

我是附属信息33

</td>

</tr>

<tr>

<td>

<input type=*"checkbox"* name=*"mail"* />

</td>

<td>

张三44

</td>

<td>

我是邮件44

</td>

<td>

我是附属信息44

</td>

</tr>

<tr>

<td>

<input type=*"checkbox"* name=*"mail"* />

</td>

<td>

张三55

</td>

<td>

我是邮件55

</td>

<td>

我是附属信息55

</td>

</tr>

<tr>

<td>

<input type=*"checkbox"* name=*"mail"* />

</td>

<td>

张三66

</td>

<td>

我是邮件66

</td>

<td>

我是附属信息66

</td>

</tr>

<tr>

<td>

<input type=*"checkbox"* name=*"mail"* />

</td>

<td>

张三117

</td>

<td>

我是邮件117

</td>

<td>

我是附属信息17

</td>

</tr>

<tr>

<td>

<input type=*"checkbox"* name=*"mail"* />

</td>

<td>

张三118

</td>

<td>

我是邮件118

</td>

<td>

我是附属信息118

</td>

</tr>

<tr>

<td>

<input type=*"checkbox"* name=*"mail"* />

</td>

<td>

张三119

</td>

<td>

我是邮件119

</td>

<td>

我是附属信息119

</td>

</tr>

<tr>

<td>

<input type=*"checkbox"* name=*"mail"* />

</td>

<td>

张三1100

</td>

<td>

我是邮件110

</td>

<td>

我是附属信息110

</td>

</tr>

<tr>

<td>

<input type=*"checkbox"* name=*"mail"* />

</td>

<td>

张三11a

</td>

<td>

我是邮件11a

</td>

<td>

我是附属信息11a

</td>

</tr>

<tr>

<td>

<input type=*"checkbox"* name=*"mail"* />

</td>

<td>

张三11b

</td>

<td>

我是邮件11b

</td>

<td>

我是附属信息1b1

</td>

</tr>

<tr>

<td>

<input type=*"checkbox"* name=*"mail"* />

</td>

<td>

张三11c

</td>

<td>

我是邮件11c

</td>

<td>

我是附属信息11c

</td>

</tr>

<tr>

<td>

<input type=*"checkbox"* name=*"mail"* />

</td>

<td>

张三11d

</td>

<td>

我是邮件11d

</td>

<td>

我是附属信息11d

</td>

</tr>

<tr>

<td>

<input type=*"checkbox"* name=*"mail"* />

</td>

<td>

张三11e

</td>

<td>

我是邮件11e

</td>

<td>

我是附属信息11e

</td>

</tr>

<tr>

<th>

<input type=*"checkbox"* name=*"all"* onclick="checkAll(this)" />全选

</th>

<th colspan=*"3"*>

<input type=*"button"* value=*"全选"* onclick="checkAllByBut(1)" />

<input type=*"button"* value=*"取消全选"* onclick="checkAllByBut(0)" />

<input type=*"button"* value=*"反选"* onclick="checkAllByBut(2)" />

<input type=*"button"* value=*"删除所选附件"* onclick="deleteMail()" />

</th>

</tr>

</table>

</body>

</html>

# 253.DOM示例-调查问卷+性格测试（单选按钮）：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Untitled Document</title>

<style type=*"text/css"*>

#contentid{

display:*none*;

}

#no1ul{

list-style:*none*; margin:*0px*;

}

.close{

display:*none*;

}

.open{

display:*block*;

}

</style>

<script type=*"text/javascript"*>

**function** showResult(){

//1,判断是否有答案被选中。

//获取所有no1的radio。并遍历判断checked状态。

**var** oNo1Nodes = document.getElementsByName("no1");

**var** flag = **false**; **var** val;

**for**(**var** x=0; x<oNo1Nodes.length; x++){

**if**(oNo1Nodes[x].checked){ flag = **true**;

val = oNo1Nodes[x].value;

### break;

}

}

**if**(!flag){

// alert("")

document.getElementById("errinfo").innerHTML = "没有任何答案被选中 ".fontcolor("red");

### return;

}

**if**(val>=1 && val<=3){ document.getElementById("res\_1").className = "open"; document.getElementById("res\_2").className = "close";

}

### else{

document.getElementById("res\_1").className = "close"; document.getElementById("res\_2").className = "open";

}

}

</script>

</head>

<body>

<!--

单选按钮演示：

1，是否参与调查问卷？

2，性格测试。

-->

<!--2，性格测试。-->

<h2>欢迎您参与性格测试：</h2>

<div>

<h3>第一题：</h3>

<span>您喜欢的水果是什么？</span>

<ul id=*"no1ul"*>

<li><input type=*"radio"* name=*"no1"* value=*"1"* />葡萄</li>

<li><input type=*"radio"* name=*"no1"* value=*"2"* />西瓜</li>

<li><input type=*"radio"* name=*"no1"* value=*"3"* />苹果</li>

<li><input type=*"radio"* name=*"no1"* value=*"4"* />芒果</li>

<li><input type=*"radio"* name=*"no1"* value=*"5"* />樱桃</li>

</ul>

</div>

<div>

<input type=*"button"* value=*"查看测试结果"* onclick="showResult()" />

<span id=*"errinfo"*></span>

<div id=*"res\_1"* class=*"close"*>1-3分：你的性格内向并扭曲，建议...</div>

<div id=*"res\_2"* class=*"close"*>4分以上：你的性格外向并分裂，建议...</div>

</div>

<hr/>

<script type=*"text/javascript"*>

**function** showContent(node){

**var** oDivNode = document.getElementById("contentid");

**with**(oDivNode.style){ **if**(node.value=='yes'){

display = "block";

}**else**{

display = "none";

}

}

}

</script>

<!--问卷调查-->

<div>

您要参与问卷调查吗：

<input type=*"radio"* name=*"wenjuan"* value=*"yes"* onclick="showContent(this)" />

是

<input type=*"radio"* name=*"wenjuan"* value=*"no"* checked=*"checked"* onclick="showContent(this)" />否

</div>

<div id=*"contentid"*>

问卷调查内容：<BR/>

您的姓名：<input type=*"text"* /><br>

您的邮箱：<input type=*"text"* />

</div>

</body>

</html>

# 254.DOM示例-选择颜色（下拉菜单）：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Untitled Document</title>

<style tyle=*"text/css"*>

.clrclass{

height:*50px*; width:*50px*; float:*left*; margin-right:*30px*;

}

#text{

clear:*left*; margin-top:*20px*;

}

.selClass{

width:*100px*;

}

</style>

</head>

<body>

<!--

-->

<script type=*"text/javascript"*>

**function** changeColor(node){

**var** colorVal = node.style.backgroundColor;

document.getElementById("text").style.color = colorVal;

}

</script>

<div class=*"clrclass"* id=*"clr1"* style=*"background-color:red"* onclick="changeColor(this)"></div>

<div class=*"clrclass"* id=*"clr2"* style=*"background-color:green"* onclick="changeColor(this)"></div>

<div class=*"clrclass"* id=*"clr3"* style=*"background-color:blue"* onclick="changeColor(this)"></div>

<div id=*"text"*> 需要显示效果的文字<br/> 需要显示效果的文字<br/> 需要显示效果的文字<br/> 需要显示效果的文字<br/>

</div>

<hr/>

<script type=*"text/javascript"*>

**function** changeColor2(){

**var** oSelectNode = document.getElementsByName("selectColor")[0];

//获取所有的option。

/\*with (oSelectNode) {

var collOptionNodes = options;

}\*/

alert(options[selectedIndex].innerHTML);

**var** colorVal = oSelectNode.options[oSelectNode.selectedIndex].value; document.getElementById("text").style.color = colorVal;

// for(var x=0;x<collOptionNodes.length; x++){

// alert(collOptionNodes[x].innerHTML);

// }

}

</script>

<select name=*"selectColor"* onchange=*"changeColor2()"*>

<option value=*"black"*>选择颜色</option>

<option value=*"red"*>红色</option>

<option value=*"green"*>绿色</option>

<option value=*"blue"*>蓝色</option>

</select>

<select name=*"selectColor2"* onchange=*"changeColor3()"* class=*"selClass"*>

<option style=*"background-color:black"* value=*"black"*>选择颜色</option>

<option style=*"background-color:red"* value=*"red"*></option>

<option style=*"background-color:green"* value=*"green"*>绿色</option>

<option style=*"background-color:blue"* value=*"blue"*>蓝色</option>

</select>

</body>

</html>

# 255.DOM示例-选择城市（下拉菜单）：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Untitled Document</title>

<style type=*"text/css"*> select{

width:*100px*;

}

</style>

</head>

<body>

<script type=*"text/javascript"*>

**function** selectCity(){

**var** collCities = [['选择城市']

,['海淀区','朝阳区','东城区','西城区']

,['济南','青岛','烟台','威海']

,['沈阳','大连','鞍山','抚顺']

,['石家庄','保定','邯郸','廊坊']];

// var arr = {"beijing":['海淀区','朝阳区','东城区','西城区']};

//获取两个下拉菜单对象。

**var** oSelNode = document.getElementById("selid");

**var** oSubSelNode = document.getElementById("subselid");

//获取到底选择的是哪个省。

**var** index = oSelNode.selectedIndex;

//通过角标到容器去获取对应的城市数组。

**var** arrCities = collCities[index];

//将子菜单中的内容清空一下。

// for(var x=0;x<oSubSelNode.options.length; ){

// oSubSelNode.removeChild(oSubSelNode.options[x]);

// }

//清除动作 oSubSelNode.length = 0;

//遍历这个数组。并将这个数组的元素封装成option对象，添加到子菜单中

**for**(**var** x=0; x<arrCities.length; x++){

**var** oOptNode = document.createElement("option"); oOptNode.innerHTML = arrCities[x];

oSubSelNode.appendChild(oOptNode);

}

}

</script>

<select id=*"selid"* onchange=*"selectCity()"*>

<option>选择省市</option>

<option value=*"beijing"*>北京</option>

<option>山东</option>

<option>辽宁</option>

<option>河北</option>

</select>

<select id=*"subselid"*>

<option>选择城市</option>

</select>

</body>

</html>

256.DOM示例-添加删除附件：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Untitled Document</title>

<style type=*"text/css"*>

table a:link,table a:visited{ color:*#179ed9*;

text-decoration:*none*;

}

table a:hover{ color:*#f36021*;

}

</style>

<script type=*"text/javascript"*>

**function** addFile(){

/\*

\* 因为文件选取框定义在行对象中。

\* 所以只要给表格创建新的行和单元格即可。

\*/

**var** oTabNode = document.getElementById("fileid");

**var** oTrNode = oTabNode.insertRow();

**var** oTdNode\_file = oTrNode.insertCell();

**var** oTdNode\_del = oTrNode.insertCell();

oTdNode\_file.innerHTML = "<input type='file' />";

// oTdNode\_del.innerHTML = "<a href='javascript:void(0)' onclick='deleteFile(this)'>删除附件</a>"; oTdNode\_del.innerHTML = "<img src='1.jpg' alt='删除附件' onclick='deleteFile(this)' />";

}

**function** deleteFile(node){

**var** oTrNode = node.parentNode.parentNode; oTrNode.parentNode.removeChild(oTrNode);

}

</script>

</head>

<body>

<table id=*"fileid"*>

<tr>

<td><a href=*"javascript:void(0)"* onclick="addFile()">添加附件</a></td>

</tr>

<!--tr>

<td><input type="file" /> </td>

<td><a href="javascript:void(0)">删除附件</a></td>

</tr-->

</table>

</body>

</html>

# 257.DOM示例-表单验证（事件和信息提示方式、正则

校验、表单提交的两种方式）：

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"*>

<title>Untitled Document</title>

</head>

<body>

<!--

表单校验。

-->

<script type=*"text/javascript"*>

/\*

//校验用户名

function checkUser(){ var flag;

var oUserNode = document.getElementsByName("user")[0]; var name = oUserNode.value;

//定义正则表达式

var reg = new RegExp("^[a-z]{4}$","i");//必须是四个字母。

// reg = new RegExp("^\\d{4}$");// 必须是四个数字。

// reg = /^\d{4}$/;

var oSpanNode = document.getElementById("userspan");

// if(name=="abc"){ if(reg.test(name)){

oSpanNode.innerHTML = "用户名正确".fontcolor("green");

flag = true;

}else{

oSpanNode.innerHTML = "用户名错误".fontcolor("red"); flag = false;

}

return flag;

}

\*/

/\*

\* 发现很多框的校验除了几个内容不同外，校验的过程是一样的。

\* 所以进行了代码的提取。

\*

\*/

**function** check(name,reg,spanId,okinfo,errinfo){

**var** flag;

**var** val = document.getElementsByName(name)[0].value;

**var** oSpanNode = document.getElementById(spanId);

**if**(reg.test(val)){

oSpanNode.innerHTML = okinfo.fontcolor("green"); flag = **true**;

}**else**{

oSpanNode.innerHTML = errinfo.fontcolor("red"); flag = **false**;

}

**return** flag;

}

//校验用户名。

**function** checkUser(){

**var** reg = /^[a-z]{4}$/i;

**return** check("user",reg,"userspan","用户名正确","用户名错误");

}

//校验密码；

**function** checkPsw(){

**var** reg = /^\d{4}$/;

**return** check("psw",reg,"pswspan","密码格式正确","密码格式错误");

}

//校验确定密码。只要和密码一致即可。

**function** checkRepsw(){

**var** flag;

//获取密码框内容。

**var** pass = document.getElementsByName("psw")[0].value;

//获取确认密码框内容。

**var** repass = document.getElementsByName("repsw")[0].value;

//获取确认密码的span区域。

**var** oSpanNode = document.getElementById("repswspan");

**if**(pass==repass){

oSpanNode.innerHTML = "两次密码一致".fontcolor("green"); flag = **true**;

}**else**{

oSpanNode.innerHTML = "两次密码不一致".fontcolor("red"); flag = **false**;

}

**return** flag;

}

//校验邮件

**function** checkMail(){

**var** reg = /^\w+@\w+(\.\w+)+$/i;

**return** check("mail",reg,"mailspan","邮件地址正确","邮件地址错误");

}

// 提交事件处理。

**function** checkForm(){

// alert(checkUser() +"--"+ checkPsw() +"--"+ checkRepsw() +"--"+ checkMail());

**if**(checkUser() && checkPsw() && checkRepsw() && checkMail())

### return true; return false;

}

**function** mySubmit(){

**var** oFormNode = document.getElementById("userinfo");

oFormNode.submit();

}

</script>

<form id=*"userinfo"* onsubmit=*"return checkForm()"*>

用户名称：<input type=*"text"* name=*"user"* onblur=*"checkUser()"* />

<span id=*"userspan"*></span>

<br/>

输入密码：<input type=*"text"* name=*"psw"* onblur=*"checkPsw()"* />

<span id=*"pswspan"*></span>

<br/>

确定密码：<input type=*"text"* name=*"repsw"* onblur=*"checkRepsw()"* />

<span id=*"repswspan"*></span>

<br/>

邮件地址：<input type=*"text"* name=*"mail"* onblur=*"checkMail()"* />

<span id=*"mailspan"*></span>

<br/>

<input type=*"submit"* value=*"提交数据"* />

</form>

<hr/>

<!--自定提交按钮-->

<input type=*"button"* value=*"我的提交"* onclick="mySubmit()" />

</body>

</html>

258.JavaScript操作DOM模型前后查漏知识综述： DOM即文档对象模型，它定义了操作文档对象的接口，WEB页面的HTML文档，document就是

根节点，其他的子对象就是子节点。 DOM模型在AJAX开収中的作用：在AJAX中，DOM模型其实是最核心的结构，是所有AJAX开収 的基础架构，如果没有DOM模型就没有办法在客户端改发页面的内容，所有的尿部刷新和异步请 求也就无从实现，熟练掌插DOM模型相关技术，才算是真正掌插了AJAX开収的精髓。 浏览器装载和显示页面的过程：一，下载源代码；二，通过页面源代码加载相关内容到内存，也就

是根据HTML源代码在内存中构建相关的DOM对象；三，根据DOM对象的相关属性杢迚行显示。

结点：在DOM模型中整个文档就是由局次丌同的结点组成的，可以说是结点代表了全部内容，结 点的类型有元素结点、文本结点、属性结点，文本节点和属性结点都看作元素结点的子结点，我们 一般所说的结点就是元素结点。

--------------------------------------------------------------------------------------------------

元素结点的引用有直接引用和间接引用，直接引用结点即：使用document.getElementById( )引 用指定id的结点、使用document.getElementsByTagName（"a"），将所有<a>元素结点放到一 个数组中返回、使用document.getElementsByName（"abc"），将所有name属性为"abc"的元 素结点放到一个数组中返回； 间接引用结点有：引用子结点，每个元素都有一个childNodes集合属性，类型是数组对象，表示 该结点的所有子结点的集合，这些子结点按照它在文档中出现的顺序排列，因此可以通过索引杢依 次访问各个子结点，firstChild，lastChild；引用父结点，DOM模型中，除了根结点，每个结点都 仅有一个父结点，可以用parentNode属性杢引用；引用兄弟结点，element.nextSibling；//引 用下一个兄弟结点，element.previousSibling；//引用上一个兄弟结点，如果该结点没有相应的 兄弟结点，则属性返回null。

--------------------------------------------------------------------------------------------------

结点的属性，nodeName：元素结点返回结点类型（即，标记名称）；属性结点返回undefined； 文本结点返回"#text"。nodeType：元素结点返回1，属性结点返回2，文本结点返回3。nodeValue： 元素结点返回null，属性结点返回undefined，文本结点返回文本值。 可以通过nodeType解决火狐和ie关亍空白文本是否作为子结点的差异。

--------------------------------------------------------------------------------------------------

处理属性结点：元素结点.属性名称（可以读写属性值），使用setAttribute( )，getAttribute( )方 法添加和设置属性。

处理文本结点：要获叏一个结点内的文本，一般使用innerHTML属性，innerHTML属性丌尿限亍

操作一个结点，而是可以使用HTML片段直接填充页面戒直接获叏HTML片段，大大提高了开収的 灵活性。

--------------------------------------------------------------------------------------------------

改发文档的局次结构：使用document.createElement( )方法创建元素结点；使用appendChild( ) 方法添加结点；使用insertBefore( )方法揑入子结点；使用replaceChild( )方法叏代子结点，前提 是oldNode必须是已经存在的，否则会収生异常；使用removeChild( )方法删除子结点。 注意：以上方法丌能用杢操作下拉列表和表格，操作下拉列表和表格要用DHTML接口中的方法。

--------------------------------------------------------------------------------------------------

JavaScript操作CSS：通过style对象改发结点的CSS，利用style对象杢控制元素的CSS，每个style 对象对应为该元素指定的CSS，而每个CSS属性--对应亍style对象的属性，对亍每个单词的CSS属 性，在style对象中的属性名称丌发，对亍双单词戒者多单词的属性改写为骆驼写法，例如CSS中的 font-size对应style属性为fontSize；使用className属性指定结点样式，结点的className属性 对应亍CSS中的类选择器。

--------------------------------------------------------------------------------------------------

获叏表单对象的方法： 一， 利用表单在文档中的索引戒者表单的 name属性杢引用表单， document.forms[i] //得到页面中的第i个表单，document.forms[formName] //得到页面中相 应 name 的 表 单 ； 二 ， 使 用 表 单 的 id 属 性 ， document.getElementById （ formId ）； 三 ， document.forName。

Form对象的常用属性：action 返回戒者设置action属性；elements 获叏一个数组，包括该表单 中所有的表单域；length 返回表单的表单域的数量；method 返回戒设置表单的method属性； name 返回戒设置表单的name属性

--------------------------------------------------------------------------------------------------

Form对象的方法：submit（），相弼亍单击submit按钮，表示表单提交到指定页面；reset（），

相弼亍单击reset按钮，表示所有表单域到初始值。

--------------------------------------------------------------------------------------------------

Form对象的事件：onsubmit 在表单提交乊前触収；onreset 在表单被重置乊前触収。

--------------------------------------------------------------------------------------------------

表 单 域 通 用 方 法 ， 引 用 表 单 域 的 四 种 方 法 ： form.name 、 form.elements[i] 、 form.elements[name]、document.getElementById(id)；disabled：使表单域丌能接叐用户的 操作，发为叧读；name：获叏戒设置表单域的名称；form：获叏该表单域所属的表单；value： 获叏戒设置表单域的值；type：判断表单域的类型；focus( )：使表单域获得焦点；blur( )：是表 单域失去焦点。

--------------------------------------------------------------------------------------------------

文本域（text、password、textarea）：利用value属性获叏和设置文本域内容；利用defaultValue 获得文本域的默认值，reset（）方法就是调用该属性。

--------------------------------------------------------------------------------------------------

单选按钮组和复选框常用操作：通过checked属性获得选中和未选中的状态；获得单选按钮组的值 叧能通过遍历单选按钮杢获叏被选中的值；复选框的处理类似单选按钮。

--------------------------------------------------------------------------------------------------

下拉列表的使用：使用value属性获叏和设置下拉列表选中选项的值；使用selectedIndex属性获 叏弼前被选中的选项的索引；使用options属性获叏所有选项的集合；使用option对象的value属 性和text属性，可以读写这两个属性；使用option对象的selected属性选中该option。