Contents

[第 I 条 Java基础 2](#_Toc531251985)

[节 1.01 概述 2](#_Toc531251986)

[节 1.02 基本概念 2](#_Toc531251987)

[节 1.03 运算符 3](#_Toc531251988)

[节 1.04 数据类型 5](#_Toc531251989)

[节 1.05 流程控制 8](#_Toc531251990)

[节 1.06 面向对象 9](#_Toc531251991)

[第 II 条 Java高级编程 16](#_Toc531251992)

[节 2.01 正则表达式 16](#_Toc531251993)

[节 2.02 GUI 16](#_Toc531251994)

[节 2.03 本地化和内部类 16](#_Toc531251995)

[(a) 本地化 16](#_Toc531251996)

[(b) 内部类 17](#_Toc531251997)

[节 2.04 异常机制 18](#_Toc531251998)

[节 2.05 泛型、容器(集合框架) 19](#_Toc531251999)

[(a) 泛型 19](#_Toc531252000)

[(b) 集合 20](#_Toc531252001)

[节 2.06 多线程 22](#_Toc531252002)

[节 2.07 I/O流 25](#_Toc531252003)

[节 2.08 JDBC 27](#_Toc531252004)

[节 2.09 网络编程 30](#_Toc531252005)

[节 2.10 类加载和反射、、、、 32](#_Toc531252006)

# Java基础

## 概述

特点(简单自然\面向对象\平台可移植性\支持函数式编程\网络技能JIt编译\更好的并发编程\高性能 动态性 多线程\健壮安全)

术语JDK(javac、java、javadoc、javap、javah、jar、jdb和 native2ascii)

JRE-API(java.lang、java.net、java.io、java.nio、java.util、java.time、java.text和java.sql)

JVM-HotSpot

执行方式

1. 编译 将源代码(.java)处理成字节码(.class) javac xxx.java

2. 解释 将字节码加载到JVM中执行（JIT编译）java xxx

程序结构

package 方便管理类、避免命名冲突

import 除java.lang包其他包中的内容需要先导入

class / interface / enum / @interface

被public修饰的类/接口/枚举/注解最多只有一个

建议：一个源文件中只写一个类/接口/枚举/注解

注释 行注释 - // 块注释（多行注释）- /\* ... \*/

文档注释 - /\*\* ... \*/

@author @since @param @return

@throws @see @deprecated

IDE(编译环境) Eclipse IntelliJ IDEA NetBeans

类型：控制台程序 图形化应用程序及applet应用程序

注:java对大小写敏感，调用方法通用语法 object.methd(pram)

## 基本概念

关键字(有特殊含义的单词)

标识符：给类型(类、接口、枚举、注解)、包、方法、属性、变量等起的名字

修饰符：给类型(类、接口、枚举)、方法、属性、变量等权限的限定

访问修饰符

public -- 公共的(在整个项目中都可以调用)

private -- 私有的(只能在本类中使用)

protected -- 受保护的(只能在本包或所有子类中使用)

default(friendly) -- 缺省(只能在本包中使用)

其他修饰符

static 该方法代表发给类的消息

abstract 该方法只有声明没有实现（方法体）

final 该方法在子类中不能被重写

native 该方法通过C/C++实现

synchronized 该方法是同步方法（只允许一个线程进入执行）

java类：它是java应用程序和applet的构建块，命名采用驼峰命名以及见名知义

保证数据私有、数据初始化、不宜过多使用基本类型(抽象类，职责进行分解)

常量(final)和变量：它用来表示数据在内存中的储存地址，申明—初始化---调用，申明变量(定义变量的注意事项):

1）先明确要表示什么及它的取值范围；

2）变量名称要有实际意义及见名知义；

3）语法结构: 数据类型 变量名(长度不限);

4）变量名命名规则:只以字母,下划线,$来开头,后面只是数字,字母,下划线,$,不使用特殊符号

5）变量名不能重复(同一范畴中)

6）使用范围的区分：局部和全局变量

7）变量声明的位置决定变量作用域,变量作用域确定可在程序中按变量名访问该变量的区域

成员变量和局部变量的区别

A 作用域不同（局部变量的作用域限于定义它的方法，而成员变量的在整个类内部）

B 初始值不同（Java会给成员变量一个初始值，不会给局部变量赋予初始值）

C 在同一个方法中，不允许有同名局部变量；在不同的方法中，可以有同名局部变量,它们同名时，局部变量具有更高的优先级

包package（分门别类，易于查找、管理，使用目录解决文件同名冲突问题）

创建包：包的声明放最前，以分号结尾 package cn.jbit.classandobject; //声明包

命名规范：由小写字母组成，不能以圆点开头或结尾，最好加上唯一的前缀，通常使用组织倒置的网络域名。如：域名javagroup.net，包名后续部分依不同机构内部的规范不同而不同

使用import关键字导入外部类：import 包名. 类名；

## 运算符

数学函数java.lang.Math

算术运算符 + - \*/ %(求余数) + （连接符）注如果其两边是数字则是加号

赋值运算符

简单赋值= 例如i=10;

复杂赋值+=,-=,\*=,/=,%=例:x+=3相当于 x=x+3；x/=3相当于 x=x/3;

比较运算符，返回布尔类型>,<,>=,<=,==,!=

逻辑运算符&& ||

&&只有两个true时候才会返回 true

|| 只有两个false时候才会返回 false

一元(一个数)运算符unary operator;

+（正数）,-（负数）, ++ --（自增自减）!（非）

按位运算符（二进制运算）

AND（&）只有两个都是1 才得1 其余都得0

OR（|）只有两个都是0 才得0 其余都得1

NOT （~）取反运算

XOR（^）异或运算，同零异一（即相同得0 不同得1）

移位运算

<<左移：乘以2的n次方n代表移动的位数

>>右移：除以2的n次方n代表移动的位数

如: int a=2;（先定义并进行赋值才能使用）

int a=56789; int res1=a/10000; System.out.println( "万位位置上的数是" +res1);

int res2=a/1000%10; System.out.println( "千位位置上的数是" +res2);

int res3=a/100%10; System.out.println( "百位位置上的数是" +res3);

int res4=a/10%10; System.out.println( "十位位置上的数是" +res4);

int res5=a%10; System.out.println( "个位位置上的数是" +res5);

int sum=res1+res2+res3+res4+res5; System.out.println( "各位位置上的数相加和是" +sum);

随机数Random ran=new Random();int nunber=ran.nextInt(10);

eg 随机产生不同的随机数

package com.niit.test;

public class GetRandom {

public static void main(String[] args) {

GetRandom gr=new GetRandom();

int num[]=gr.getNum(10,1,10);//括号内的参数：最大值,最小值和随机数的个数

for (int i = 0; i < num.length; i++) { System.out.print(num[i]+" "); }

}

public int[] getNum(int max,int min,int count){

int number[]=new int[count]; int num=0;//记录有效数据 boolean b=false;

while(num<count){

int r=(int)(Math.random()\*(max-min+1)+min);

if(count>max-min+1||min>max){

System.out.println("你所要求的数据范围有误!"); return null;}

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

if(r==number[i]){ b=true; break;}}

if(!b){number[num]=r;num++; }

}return number;}}

接收键盘输入、输出

输入异常判断 .hasNextXxx(),eg input.hasNextInt();

System.out.print()不会换行 System.out.println()换行

System.out.printf(“%8.2f”，xx)以指定格式输出

Scanner input=new Scanner(System.in);创建输入对象接收输入 in.nextXxx();接收并保存输入

Scanner(File f)或Scanner(String s)从指定文件或字符串读取数据

PrintWriter(String Filename)将数据写入指定文件

举个例子：接收两个数并计算他们的和

Scanner input=new Scanner(System.in);//创建输入对象

System.out.println( "请输入第一个数"); int num1=input.nextInt();

System.out.println( "请输入第二个数"); int num2=input.nextInt();

System.out.println( "这两个数的和是"+(num1+num2));

## 数据类型

基本类型

整型: byte[字节型]1 short[短整型]2 int[整型]4 long[长整型]8

浮点型(小数):float[单精度]4 double[双精度]8 eg 1.5 0.3

字符型: char eg '男' 'a' 'b' '1'它用utf-16编码描述一个代码单元

布尔类型:boolean true/false 真/假(整型值和布尔值之间不能进行相互转换)

大数值：BigInteger BigDemical valueOf()转为大数值

引用数据类型:

基本类型封装类：Byte Short Integer Long Float Double Boolean Character Void

字符串: String eg "男" "2" "abc"

对象类型 数组Array 枚举enum 自定义的类和接口

注：未初始化时数值型默认为零 布尔型为false 对象引用为null

类型转换

1）自动转换小转大且类型兼容 2）强制转换大转小（类型名）表达式

大数值

Java.Math中BigInteger和BigDecimal,使用静态的valueOf可将普通数值转换为大数值

Eg BigInteger bi= BigInteger.valueOf(100)

它们的算术方法不能直接用算术运算符进行运算，要调用对应的方法eg BigInteger c=a.add(b)

字符串

String类---字符串//声明字符串类,创建了String类类型变量将str1指向了变量的地址

String str1 = "abc";String str2 = "abc";String str3 = new String("abc");

str1==str2,返回结果为true str1==str3,返回结果为false

a.如果内存中没有"abc",则会产生一个对象,如果内存中有了"abc",则直接找到原"abc"所对应的地址,直接引用

b.无论内存中是否有"abc",只要看见new关键字,就会产生一个新的对象(在内存中开辟一个新的空间,即拥有不同的地址)

特殊串：空串与Null串

StringBuffer：String增强版🡪StringBuilder(StringBuffer发展而来 字符串构造器 两者API相同)

String str="abc";//不可修改 System.out.println(str);

String newstr=str.concat(str);//在末尾拼接字符串System.out.println(str);

StringBuffer sb=new StringBuffer("abc");//字符串，可修改System.out.println(sb);

sb.append(sb); ////在末尾拼接字符串 System.out.println(sb);

输出结果如下：Abc abc abc abcabc

空串与NULL串

空串是一个Java对象，有自己的串长度0和内容(空)，而NULL表目前没有对象与该变量关联if(str.length==0||str.equals(“”)) if(str==null)

String类位于java.lang包中，具有丰富的方法

字符串的长度：length（）

比较字符串

equals（）:判断两个字符串的值是否一致

==:判断两个字符串在内存中的首地址，即判断是否是同一个字符串对象

equalsIgnoreCase()：不区分大小写比较两个字符串的内容

连接字符串

方法1：使用“+” 方法2：使用String类的concat()方法

提取字符串

搜索第一个出现的字符ch（或字符串value）

public int indexOf(int ch) public int indexOf(String value)

搜索最后一个出现的字符ch（或字符串value）

public int lastIndexOf(int ch) public int lastIndexOf(String value)

public String substring(int index) 提取从位置索引开始的字符串部分

public String substring(int begini, int endi) 提取beginindex和endindex之间的字符串部分

beginindex:从0开始算；endindex: 从1开始算，返回一个前后不含任何空格的字符串

public String trim()去掉字符串两边的空格

字符串拆分

String类提供split()方法，将一个字符串分割为子字符串，结果作为字符串数组返回

String words="长亭外古道边芳草碧连天晚风扶柳笛声残夕阳山外山";

printword=words.split(" ");

for(int i=0;i<printword.length;i++){System.out.println( printword[i] );}

.charAt(下标);--根据下标返回该下标所对应位置的字符，将字符串拆分为字符

eg String str="sdfgg"； char c[]=new char[str.length()]；

for(int i=0;i<str.length();i++){c[i]=str.charAt(i);}

将整数数组的整数转换为字符串数组的元素，整数->字符串

eg id[i]=Integer.toString String s=Integer.toString(n)

.toCharArray();--转化成字符数组

.contains("内容");--判断字符串是否包含该内容,返回true/false

.replace(旧内容,新内容);--将新内容替换掉旧内容

.startsWith("内容");--判断字符串是否以该内容开头

.endsWith("内容");--判断字符串是否以该内容结尾

使用toLowerCase()、toUpperCase( )方法//字符转大小写

键盘输入一个邮箱地址,判断这个邮箱地址是否合法

1.必须有@和. 2.@和.不能在一起 3.@后必须有.

4.只能有一个@ 5.邮箱长度为6-30 6.@和.不能在第一位或最后一位

7.邮箱地址除@和.之外只能以数字和字母组合

import java.util.Scanner;

public class TestString {

public static void main(String[] args) {//邮箱验证

//^([a-zA-Z0-9\_-])+@([a-zA-Z0-9\_-])+((\.[a-zA-Z0-9\_-]{2,3}){1,2})$/;

System.out.println("请输入邮箱地址：");

Scanner in = new Scanner(System.in);

String str = in.next();

if (!str.contains("@") || !str.contains(".")){System.out.println("邮箱地址须包含@和.");}

int a1 = str.indexOf("@"); int a2 = str.indexOf("."); int count = 0;

char c[] = new char[str.length()];

for (int i = 0; i < str.length(); i++) { c[i] = str.charAt(i); if (c[i] == '@') { count++; }}

if (count >= 2) {System.out.println("邮箱地址只能有一个@");}

if (a2 - a1 == 1 && a1 < a2) { System.out.println("@和.不能在一起");}

if (a1 > a2) { System.out.println("@后必须有.");}

if (str.length() > 30 || str.length() < 6) { System.out.println("邮箱长度为6-30");}

if (str.startsWith("@") || str.startsWith(".") || str.endsWith("@")|| str.endsWith(".")) {

System.out.println("@和.不能在第一位或最后一位");

}

System.out.println("请重新输入:");

}

}

数组：是一个变量，存储相同数据类型的一组数据，属于引用数据类型存放的是地址,数组的名称指向的是数组的首地址

注：引用数据类型（地址的传递，引用传递）和基础数据类型比较（独立变量，互不干扰）

判断两个引用数据类型是否相等，实质是判断其内存地址是否相同

数组结构及其要素：

A标示符即数组名称，用于区分不同的数组

B数组元素即数组中存放的数据

C元素下标对元素进行编号，数组长度是固定不变的，避免数组越界 length

常见异常：java.lang.ArrayIndexOutOfBoundsException

D元素类型即数组元素的数据类型

使用数组四步走

1声明数组 eg int[] a;

2分配空间 eg a=new int[5];1和2一步走 eg int[] a=new int[10];

3赋值 eg a[0]=0;a={0};1、2和3一步走 eg int a[]={1,2,3,4,5}(已经知道初值时)

4使用 eg a[0]=a[0]\*3;

错误写法 int age[]=new int[3];age={1,2,3};等价于int age[]=new int[3]{1,2,3};

数组排序和最值求取

A Array.copyOf(原数组,新数组长度)

若新数组长度大于原数组，那会根据数组中元素类型将新数组中的多余元素赋初值eg 0 false，若若新数组长度小于原数组，则只会复制最前面的数据

B Arrays.sort(数组名);

C 数组的最值求取，注意定义中间变量并赋数组中任意值，再比较

a[]={10,12,23,30,52,65,40},下标3的位置插入34// 数组插入运算eg在指定位置插入数据

int[] a = { 10, 12, 23, 30, 52, 65, 40 };int b[] = new int[a.length + 1];

for (int i = 0; i < b.length; i++) {if (i < 3) { b[i] =a[i] ;} else if (i == 3) { b[i] = 34; }else { b[i] = a[i - 1]; } }

for (int i = 0; i < b.length; i++) {System.out.println(b[i]); }

多维数组，数组中再装数组

eg int c[][]={{80,89,90,89},{89,90,99,88},{66,77,88,99,},{77,88,89,98}};//冒泡排序

int nun[] = { 66, 9, 10，99,88,77};int c = 0;

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

for (int j = i+1; j < nun.length - 1; j++) {

c++;if (nun[j] > nun[j + 1]) { int p = nun[j + 1]; nun[j + 1] = nun[j]; nun[j] = p; } } }

System.out.println("比较次数"+c);

案例：接收键盘输入，数组存储（一维和二维）

double score[] = new double[5];

System.out.println("请输入5个学生成绩："); Scanner in=new Scanner(System.in);

for(int i=0;i<score.length;i++){ score[i]=in.nextDouble();}

for(int i=0;i<score.length;i++){ System.out.println("第"+(i+1)+"条数据是"+score[i]);}

Arrays.sort(score);//数组排序，默认是升序排序

System.out.println("score中的最大数据是" + score[score.length - 1]);

//利用双重循环给二维数组动态键入数据

Scanner in=new Scanner(System.in);

String str[][]=new String[3][3];//1框外层数组长度，2框内层数组长度，通常不写

for(int i=0;i<str.length;i++){ System.out.println("第"+(i+1)+"小组：");

for(int j=0;j<str[i].length;j++){

System.out.println("请输入第"+(j+1)+"成员姓名"); str[i][j]=in.next(); }

System.out.println(); }

for(int i=0;i<number3.length;i++){ System.out.println(number3[i]);} // 遍历数组中的全部数据

// for-each循环: 执行顺序:①从数组中取出一个元素②把值赋值给 i ③使用i(打印i) ④重复前三步，直到数组打印完所有数据

for(int i : number3){System.out.println(i);}

Array.toString(a)//返回一个包含数组元素的字符串

枚举（类）：一些预定义的常量的集合(普通常量无法限制范围，而枚举类型则限制只能从枚举中选值)

A 类似于一个类（包含变量、方法、构造方法）B 常量 static final int i=10；不能再被赋值

enum Apple{//定义枚举

hongfushi(20), gg(30); //枚举中的构造方法，在定义枚举值时调用

private Apple(int price) {}}

enum Hour{one,two,three,four,five,six,seven,eight,nine,ten,eleven,twelve}

enum month{jan,feb,mar,apr,may,june,july,aug,sep,oct,nov,dem}

enum gender{male,female}

enum Week{

Mon,Tues,Wed,Thur,Fri,Sat,Sun;//枚举值的集合常量相当于publicstatic final xx==Mon/Tues/Wed等值

int i=12;//枚举中的变量

public int gethour(){returni;}//枚举中的方法

}

public static void main(String[] args) {//使用枚举

Week weekday= Week.Tues;System.out.println(weekday.getHour());

switch(week){

case Mon: System.out.println("今天是星期一"); break;

case Tues: System.out.println("今天是星期二"); break;}}

Apple[] values=Apple.values();//values方法返回一个对应类型的数组

for(Apple a:values){ System.out.println(a); }//遍历枚举

## 流程控制

分支结构

语法1： if（布尔值/布尔表达式）{ if代码块 }

若if中代码仅有一行，则可省大括号 若布尔值/布尔表达式等于true，则执行if代码块

语法2： if（布尔值/布尔表达式）{ if代码块

}else{ else代码块 }

若布尔值/布尔表达式等于true，则执行if代码块，否则执行else代码块

语法3：switch（字符串/字符串表达式/数字/数字表达式即char或int类型的结果）{

case 值1： break； case 值2： break； case 值3： break；default:.....}

循环结构

语法1： while（A布尔值或布尔表达式）{ B循环体 }

执行顺序：先判断A条件是否成立，成立则执行B，不成立则不执行

在B中执行到break语句是，立即跳出当前循环

否则当B执行完毕后，又返回判断A条件是否成立，依次类推

语法2： for（A任何语句；B布尔值或布尔表达式；C任何语句）{ D循环体 }

执行顺序：ABDC BDC BDC…直到break语句或B为false,A仅执行一次

语法3： do{A循环体 }while（B布尔值或布尔表达式）

执行顺序：BABA 注：可以简化if嵌套

语法4：for(A类型：A类型数组或集合){循环体}

死循环 while（true）{} for（；；）{}

注：关键字break和continue的应用

break：中止整个循环（一层循环），跳出循环

continue：结束本次循环，进入下一次循环，多用于筛选数据

练习:自己设计一个程序,要求有一个登录界面,账号(String) 密码(int)

提示,请输入账号和密码,然后进行比较,

如果账号和密码都正确,提示登陆成功(在输入密码的时候要先进行判断是否为整型)

如果有账号或者密码有一个不正确,则提示账号或密码有误,登录失败.

uname1="admin1" password1=123456; 初始账号:

uname2="admin2" password2=654321;

public static void main(String[] args) {

Scanner input = new Scanner(System.in);// 创建输入对象

System.out.println("请输入用户名"); String name = input.next();

System.out.println("请输入密码");

if (input.hasNextInt()) {int pwd = input.nextInt();

if ((name.equals("admin1") && pwd == 123456)|| name.equals("admin2") && pwd == 654321) {

System.out.println("登录成功"); }

else {System.out.println("账号或密码有误");} }

## 面向对象

面向对象（OO）的优点

封装实现了模块化和信息隐藏，封装使得在对象外部不能随意访问对象的属性和方法，提高了程序的可维护性和安全性：一个类可以创建多个对象实例，提高了程序的可重用性

对象：用来描述客观事物的一个实体，由一组属性和方法构成，对象同时具有属性和方法两项特性，它们通常被封装在一起，共同体现事物的特性，二者相辅相承，不能分割

对象的特征

属性（静态）—对象具有的各种特征，每个对象的每个属性都拥有特定值

方法（动态）—对象执行的操作

类（从对象抽象出类）：具有相同属性和方法的一组对象的集合，类是模子，确定对象将会拥有的特征（属性）和行为（方法），它是构造对象的模板或蓝图，在一个源文件中，只能拥有一个公有类(public),但可有任意数目的非公有类，而且源文件的名字须与public类的名字匹配

创建和使用对象：类名对象名 = new 类名();

引用：对象名.属性/对象名.方法名()

形参：起表示作用，没有值，任何类型，定义方法时写入的

实参：在调用方法时传入的值，类型要与相应的形参相配

数据参数传递采用的是值传递

类之间的关系：依赖(use-a) 聚合(has-a) 继承(is-a)

常用java预定义类 Math Date GregorianCalendar Calender(日历)

定义一个类的步骤

1、定义类名

2、编写类的属性//类的成员，成员属性，全局变量

3、编写类的方法//类的成员，成员方法

public class 类名 {

属性1的类型属性1; //定义属性部分

属性2的类型属性2;

属性n的类型属性n;

方法1;public void 方法名（）{} //定义方法部分

方法2 …

方法m;}

类的方法即定义类的某种行为（或功能）并返回结果,方法之间允许相互调用，不需要知道方法的具体实现，提高了效率，设计原则：高度内聚

定义方法：[访问修饰符] [其他修饰符] 返回类型 方法名([参数列表]) [异常声明] { ... }

不定个参数的方法 调用：t.add(); t.add(11,13); t.add(2,3,4,5);

Public int add(int …nun){ int sum=0; for(int i=0;i<nun.length;i++){sum+=nun[i];} return sum;}

注意：

方法不能返回多个值！

多个方法不能相互嵌套定义！

不能在方法外部直接写程序逻辑代码！

同一个类中不能有多个相同的方法（相同：参数列表相同且方法名相同，与返回值无关）

调用方法时不能指定实参类型！

调用方法后没有对返回值作任何处理

方法的调用:

1.在同一个类中,直接调用:方法名(参数列表);

2.在不同类中调用方法,先实例化对象:对象.方法名(参数列表);

方法参数

Java程序设计语言总是采用按值调用即方法得到的是所有参数值的拷贝，方法不能修改传递给它的任何参数变量的内容。

一个方法不能修改一个基本数据类型的参数（数值型和布尔值）

一个方法可改变一个对象参数的状态

一个方法不能让对象参数引用一个新的对象

面向对象的三大特性:封装,继承,多态

***封装:***JavaBean—可重用组件，对一个类的封装，即把属性和方法封装成类或把复杂的业务逻辑封装成方法 注意：this--本类对象 super—父类对象

A 类须提供一个公开的无参的构造器

B 所有属性私有（private）

C 所有属性提供对应get/set 方法来访问

set方法://初始化属性，批量化操作/单个操作

public void set属性名(和属性数据类型一样的参数){this.属性 = 参数对象; }

get方法(获取属性)：public 属性类型 get属性名(){return 属性;}

类初始化：构造方法 初始化块{} 静态域初始化块static{}，若没有显式设置则采用默认设置，所有的静态初始化块或语句将按照类定义的顺序执行

隐式和显式参数：显式参数是明显地列在方法声明中的，而隐式的参数则没有，在每一个方法中，关键字this表示隐式参数

Eg public void raiseSl（double per）{double raise=this.sal\*per/100; this.sal+=raise}

基于类的访问权限：类中方法可访问所属类的私有特性，而不仅限于访问隐式参数的私有特性eg class Emp{ spublic Boolean equal(Emp ep){ return name.equals(ep.name); }}

构造方法（构造器，类和枚举都可以有构造方法）作用:用来初始化对象，没有返回值

语法：修饰符 类名(参数列表){}

1.当程序new一个对象的时候,就会隐式的调用不带参的构造方法

2.当类中显示的自定义了带参构造方法,则类中默认的空的不带参的构造方法消失掉，Java会调用自定义的构造方法

静态:static修饰变量和方法，优点:调用方便缺点:占用内存 静态方法不能使用this关键字

如果类中的成员变量定义为静态的,则所有对象都共享该属性,其生命周期贯穿，对象的所有的"副本" 共享是同一个，也是单独存放，不实例化也可以直接使用

静态变量须用类名来访问，也称为类变量

静态方法须用类名来访问，也称为类方法

静态方法调用如下：类名.属性名 类名.方法名(参数列表);

静态的类里面只能调用静态的方法,除此之外只能用new对象调用

静态的方法里面只能调用静态的属性,除此之外只能用new对象调用

使用静态方法的两种情况：

一个方法不需要访问对象状态，其所需参数都是通过显式参数提供

一个方法只需要访问类的静态域

Eg工厂方法和main()方法(程序入口或进行单元测试)

public class A{

eg static private int age;// private 创建set和get方法其他类才能调用

private String name;

public void p(){ System.out.println(A.age); }//可用类名来直接访问静态的方法和变量

public static void say() {

System.out.println(new DogA().name);// 静态方法new调用非静态变量

System.out.println(age);}}// 静态方法直接调用静态变量

final--最终的

1.修饰类---这个类是最终类,不能被继承

2.修饰方法---这个方法是最终方法,不能被重写

3.修饰静态方法--没什么用

4.修饰全局变量--全局常量 不能被修改

5.修饰局部变量/方法参数--局部常量

***继承:***

public class Person {

private String name; private int age;//set/get方法

public Person(int age) { System.out.println("父类Person的构造方法"+age); }

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

public void sleep(){System.out.println("睡觉"); }

}

public class Student extends Person{

private int sid;//学号，子类自己的属性

public void havingClass(){}//子类自己的方法

public Student() {

super(22);//显示调用父类构造方法

System.out.println("子类Student的构造方法");

}

public void sleep() {//子类重写父类的方法

System.out.println("子类学生午睡,时间为1个小时");

}

public void sleep(int a) {

super.sleep();//显示调用父类的方法

System.out.println("子类学生午睡,时间为”+a+”个小时");

}

public static void main(String[] args) {

Student s=new Student();

s.setAge(30);//不能写s.age(age是父类私有的，子类不能直接调用)

System.out.println(s.getAge());

s.eat();

s.sleep();//子类若重写父类的方法，则调用子类重写后的方法，否则调用父类的方法

s.sleep(2);//子类调用重写后的方法，

}结果为父类Person的构造方法22，子类Student的构造方法

}结果为子类学生午睡,时间为1个小时，父类睡觉，子类学生午睡,时间为2个小时

1.提高代码的复用性，子类可直接访问父类非私有的属性和方法

2.单一性(单继承)一个子类只能拥有一个父类,但一个父类可以拥有多个子类

3.传递性: A extends B B extends C 则A也拥有C的属性和方法

父类,超类,====object类，它是所有类的父类

如果某个类没有明确的显示继承某个类,则默认继承object类,必然会调用Object类的构造器

子类中当new时会先调用父类的构造方法，再调用子类的构造方法

在书写构造器时,记得要调用父类的构造器，如果没写,则默认调用父类无参的构造器

1显示调用父类带参的构造方法：super(值...)（须写在当前构造器中的第一行）

2.调用父类的属性和方法：super.属性 super.方法

对象类型之间的类型转换,

必要条件：子类和父类之间extends或实现类和接口之间implements

向上类型转换：父类变量可以引用子类对象，但子类变量不能引用父类对象

向下类型转换：还原必须是经过向上类型转换之后才能向下类型转换

Animal a=new Dog();√new Dog创建一个Dog类的实例，Animal a申明一个Animal类型的变量a，指向Dog类的实例，a其实是Dog类型

Dog d=new Animal();×

Dog d=(Dog)a;//还原，从父类转换为子类

public class Animal {}// 父类

class Dog extends Animalimplements F{ public void wang(){System.out.println("汪");}}// 子类狗

class Cat extends Animalimplements F { public void miao(){System.out.println("喵");}}// 子类猫

interface F{}//接口

class Test {

public static void main(String[] args) {Animal a = new Dog();Test t = new Test();

Dog dd = new Dog(); t.print(dd);//汪 Cat cc = new Cat(); t.print(cc);//喵

F f1=new Dog();

Dog dog=new Dog(); t.print1(dog);//汪 Cat cat=new Cat(); t.print1(cat);//喵}

public void print(Animal a) {

if (a instanceof Dog) {Dog dd = (Dog) a; dd.wang();}// 还原，从父类转换为子类

else if (a instanceof Cat) {Cat cc = (Cat) a; cc.miao();}}// 还原，从父类转换为子类

public void print1(F f) {

if (f instanceof Dog) {Dog dog = (Dog) f; dog.wang();// 还原，从父类转换为子类

} else if (f instanceof Cat) {Cat cat = (Cat) f; cat.miao();}}}//还原，从父类转换为子类

***多态(多种形态):***

1.方法的重载 2.方法重写 3.抽象方法 4.接口

方法的重载：在同一个类中满足以下条件（同类同名不同参数列表）

A方法的名称相同

B方法的参数列表不同（参数的个数、类型、顺序）

C与修饰符、返回类型无关

方法的覆盖(重写):

1.两个类必须有继承关系

2.除了方法体不同，其他的包括返回值、类型和方法名都一样

子类去重写父类的方法的几个注意点：

a.final修饰的方法不能被重写

b.private修身的方法不能被重写

c.重写之后的方法访问说明符的权限>= 父类的方法的访问说明符的权限

public> protected > default > private

抽象类abstract（抽象方法+普通类和方法）包含一个或多个抽象方法的类，不能new，

abstract class Xiao {

public static void main(String[] args) {

// Xiao xx=new Xiao();//抽象类不能实例化，只能子类继承实现

}

public abstract void add(int n1, int n2); // 抽象方法，须放在抽象类中

public void sleep() { System.out.println("普通方法");}

}

A可直接使用子类实现抽象类中的抽象方法，

B也可使子类变成抽象类继承，继续继承子类实现即可

1.抽象方法只能是方法的申明,没有方法体

2.abstract可以用来修饰方法和类

3.抽象方法必须放在抽象类中,抽象类可以没有抽象方法

4.抽象类不是静态类,密封(最终)类

接口:关键词interface一种对象类型“某种标准”如：插座/插线板国家标准

语法: public interface 接口名字{主体}

1.在接口中,所有的方法只能申明,不能有方法体即默认仅能定义公开的抽象方法，静态方法

2.接口中的变量都是 static 和 final的也可以不写static，final

3.接口不能被实例化,只能得到它的实现类(多态)implements ,多个接口实现用逗号隔开

A实现某个接口必须实现接口中的所有方法，B接口也可以继承接口或抽象类实现

4.接口是一种规范,必须要满足规范以后才可以实现接口

public interface Music { void play(); void make(); void sale();}

interface f2 {void f2();} interface f1 extends f2 {void f1();}//接口也可以继承接口

class classmusic implements Music, f2 {// 类实现接口,实现接口中所有的方法

public void play() {} public void make() {} public void sale() {} public void f2() {}}

抽象类和接口的区别:

1.抽象类:子类和父类要有关联,接口:和实现类可以毫不相关

2.继承只能单继承,接口可以多实现

1）练习：设计一个父类为图形类，派生出圆形和矩形，要求使用多态，通过调用父类的计算面积的方法分别计算圆形和矩形的面积

**figure**double height; double width Area()

**Rectangle**double height;double width Area()

**Rotundity**double width Area()

2）继承练习

设计一个手机类和一个照相机类，手机有型号、颜色、可以打电话，照相；照相机有型号、颜色、可以照相、摄像。类中还具备哪些成员可以自己思考，要求画出UML类图

实现UML类图的代码，在测试代码中：

定义一个能够装10个电子设备的数组，

用Random对象实现随机地放入手机或照相机。

遍历每一个电子设备，如果是手机就打电话和照相，如果是照相机，就只照相

设计模式(23种)

常用的设计模式

单例模式(创建型模式 为了创建某个对象的实例)

目的：保证一个类仅有一个实例

步骤：私有化构造方法

创建静态、私有的单一实例

实现静态、公开的一个方法返回单例

分类：懒汉式、饿汉式、登入式(静态内部类)

private static Singleton2 ints;//懒汉式,不会直接实例化单例

private Singleton2() { }

public static Singleton2 getInts() {

if (ints == null) {ints = new Singleton2(); }

return ints;}

public class Singleton1 {//饿汉式

public int i = 10;

// 一 静态static 全局变量，内存中仅一个,保证只有一个实例

private static Singleton1 ints = new Singleton1();

// 二 构造函数/方法私有化，private确保在外部无法创建其他实例

private Singleton1() {}

// 三 公开公共方法以便外部可以访问单一实例

public static Singleton1 getInts() { return ints; }

public void print() { System.out.println(i++); }}

public static void main(String[] args) {需求:仅创建一个实例

Singleton1 s1=Singleton1.getInts(); s1.print();

Singleton1 s2=Singleton1.getInts(); s2.print();}

private Singleton3() {}// 登入式（静态内部类式）

private static class SingletonClass {

private static final Singleton3 ints = new Singleton3();}

public static final Singleton3 getInts() {return SingletonClass.ints;}

工厂模式 创建型模式

public interface Phone { public void call();}

public class Oppo implements Phone {public void call() {System.out.println("OP");}}

public class HW implements Phone {public void call() {System.out.println("华为");}}

public class PFactory {

public Phone getPhone(String name) {

if ("hw".equals(name)) {return new HW();}

if ("op".equals(name)) {return new Oppo(); }

else { return null;}

}}

class Test1 {// 父类和子类 接口和其实现类的类型可自动转换

public static void main(String[] args) {

PFactory fa = new PFactory();// 创建工厂

Phone hw = fa.getPhone("hw"); hw.call();

Phone op = fa.getPhone("op"); op.call();

}

}

抽象工厂模式：工厂的工厂

命令模式

public class Stock {// 股票类充当请求类

private String code;

private int number;

public String getCode() {return code;}

public void setCode(String code) { this.code = code; }

public int getNumber() { return number; }

public void setNumber(int number) {this.number = number; }

public void buy(){System.out.println("买入"+code+number+"股"); }

public void sell(){System.out.println("卖出"+code+number+"股");}

}

public interface Order { public void excute();//买卖股票命令操作}

public class BuyS implements Order {

private Stock ss; public BuyS(Stock ss) {this.ss=ss;} public void excute() { ss.buy();}}

public class SellS implements Order {

private Stock ss; public SellS(Stock ss) {this.ss=ss;} public void excute() {ss.sell();}}

public class Broker {// 操作员 被委托人 接收并执行买、卖的命令

private List<Order> ol = new ArrayList();

public void takeOrder(Order order) {

ol.add(order);// 接收命令并存入命令列表

if (order instanceof BuyS) {System.out.println("成功接收买入命令"); }

if (order instanceof SellS) {System.out.println("成功接收卖出命令");}}

public void execOrder() { for (Order or : ol) { or.excute();}}// 执行命令

}

public class Boss {//请求者，委托人

public static void main(String[] args) {

Stock s1 = new Stock();//创建股票

s1.setCode("顺丰快递"); s1.setNumber(2000);

BuyS buy = new BuyS(s1);//创建命令

SellS sell=new SellS(s1);

Broker bb = new Broker();//创建操作员

bb.takeOrder(buy);//操作员接收买入命令

bb.takeOrder(sell);//操作员接收买入命令

bb.execOrder();//操作员执行买入、卖出命令

}}

# Java高级编程

## 正则表达式

字符串（查询或替换大的字符串的条件）

1模式Patterm 指定一个正则表达式，当做查询或替换的条件

2 匹配Matcher对象 指定被查询或替换的目标字符串，并执行查询或替换

正则表达式的语法

A [ ]匹配一个字符 . 任意一位字符 [abc] 匹配a b c中的一个

[^abc] 匹配不是a b c的一个 [a-zA-z] 匹配一个字母

[b-e[n-q]] 匹配b-e或n-q中的一个 [a-z&&[abc]] 匹配 a-z且a b c中的一个

B 量词:控制字符出现的次数

? 字符出现零次或一次 \* 字符出现零次或n次

+ 字符出现一次或n次 {m,n} {m, }{ ,n} 出现m次到n次

Eg Pattern p2 = Pattern.compile("a?"); Matcher m2=p2.matcher("aa");

System.out.println(m2.matches());FALSE

Pattern p5 = Pattern.compile("a?"); Matcher m5 = p5.matcher("aacbads");

while (m5.find()) {System.out.println("找到" + m5.group() + ",起始" + m5.start() + ",结束"+ m5.end());}

Patternp3=Pattern.compile("a\*"); Matcher m3=p3.matcher("aa");

System.out.println(m3.matches());TRUE

Pattern p4 = Pattern.compile("a+"); Matcher m4=p4.matcher("aa");

System.out.println(m4.matches());TRUE

C限定符 目标字符串xfooxxxxxxfoo

Greedy贪婪模式/最长匹配模式 然后再回溯eg则 .\*foo匹配结果为0-13

Relutant 懒惰模式 最短模式 eg .\*?foo匹配结果为xfoo 0-4 xxxxxxfoo 4-13

Possessive 占有模式先匹配最长模式 然后不会回溯 .\*+foo匹配结果为匹配不到

## [GUI](GUI（图形用户界面）笔记.docx)

## 本地化和内部类

### 本地化

将程序特定为当地语言环境和文化的过程

public class Localdate {

public static void main(String[] args) {

Date date = new Date();// 本地化日期

System.out.println(date);

// locale确定国家和语言，传入国家代码集合语言代码集

Locale ll = new Locale("de", "DE");// Locale ll = new Locale("zh", "cn");

// DateFormat类实现日期的格式化

DateFormat df = DateFormat.getDateInstance(DateFormat.LONG, ll);

String re = df.format(date); System.out.println(re);

// 本地化时间

DateFormat tf = DateFormat.getTimeInstance(DateFormat.LONG, ll);

String te = tf.format(date); System.out.println(te);

// 本地化货币

NumberFormat nf = NumberFormat.getCurrencyInstance(ll);

String nn = nf.format(11111.8888); System.out.println(nn);

// 本地化文本资源束 资源文件

ResourceBundle rb1 = ResourceBundle.getBundle("Message", ll);

String m1 = rb1.getString("message"); System.out.println(m1); }}

### 内部类

属于类成员

A正则内部类/规则内部类 注;须先实例化外部类，再实例化内部类，外部类不可以访问内部类的变量和方法，内部类可以访问外部类的变量和方法

public int i = 100; // 外部类的变量和方法

public void print() { System.out.println("外部类的方法");}

class Inner {// 普通内部类

public int i = 101; // 内部类的变量和方法

public void print() { int j=1002; System.out.println("内部类的方法");}

}

public static void main(String[] args) {

Gp gp = new Gp(); System.out.println(gp.i);

Inner in=gp.new Inner();//外部类实例化对象.new 内部类 System.out.println(in.i); }

B静态内部类//它不能问外部类的非静态变量、方法，非静态变量、方法须先实例化才能用

public int m = 2; public static int n = 3;

public static class Inners {// 静态内部类

public void print() {// System.out.printl(m);

System.out.println(n);}}// 静态变量、方法则可直接使用

public static void main(String[] args) { Gp gp = new Gp(); System.out.println(gp.i);

// 静态内部类不需例化外部类,可直接实例化内部类

// 静态内部类只能访外部类的静态的变量和方法

Gp.Inners ins = new Gp.Inners(); System.out.println(); }

C方法局部内部类：只在方法内部有效

public int k = 100;

public void prins() {

int a=10;

class Innerf {int mm = 1; public void pin() { System.out.println("方法局部内部类");}}

Innerf inf=new Innerf();

System.out.println(inf.mm); inf.pin();//调用其变量和方法

//System.out.println(inf.a);//不能访问超出其类范围的变量}

D匿名内部类

public void ppap(){ new animal(){ public void tst() {}};}

interface animal{ public void tst();}

## 异常机制

1 什么叫异常？运行时的错误

2 异常的危害性：使程序异常中断

3 java中把不同类型异常都封装成一个类， java中所有异常类父类都是Exception类

例如:ArithmeticException类表示算术运算相关的异常

4 如何来处理异常

try 语句块包含可能出现异常的代码,语法 try{// 可能出异常的代码}

catch 1.语句块，捕获异常进行处理

2.需要传参数，也就是要捕获的异常的类型

3.一个try语句可以跟多个catch

4.多个异常可以合并为一个catch

例如：catch(ArithmeticException | ClassCastException e)

finally 语句块，出异常时候还可以继续执行，无论是否出异常，都会执行的代码

throws 使用在方法的签名中抛出异常，目的：在方法中不进行异常处理，抛出到调用方法的地方去处理

throw 手动抛出异常，自定义异常，例如: throw new ArithmeticException();

Scanner sc =new Scanner(System.in);

int i=10; int j=sc.nextInt();

try{ System.out.println(i/j);

}catch(ArithmeticException|ClassCastException e){//e 存放异常的具体信息,异常合并

e.printStackTrace(); System.out.println("输入异常，计算失败！");

}catch(){System.out.println("类型转换错误");//多个异常

}finally{System.out.println("无论是否出现异常都会出现的代码");

}

Test t = new Test();

t.add(100, 200); //调用add，但add抛出异常，所以这里需try catch finally处理

public void add(int n1,int n2)throws ArithmeticException{

int i=11;int j=0; System.out.println(i/j); }

## 泛型、容器(集合框架)

### 泛型

类型参数（将类型作为参数，提高代码的复用性，更灵活，使得程序具有更好的可读性和安全性）

泛型的适用范围

A 泛型类(具有一个或多个类型变量的类，看做是普通类的工厂) 类型参数<T>

public class GenericClass<T> {

private T t;

public void setT(T t) {this.t = t;}

public T getT() {return t; }

public static void main(String[] args) {

GenericClass<String> gcs = new GenericClass<String>(); gcs.setT("字符"); GenericClass<Integer> gci = new GenericClass<Integer>(); gci.setT(11);}

}

B 泛型方法

a 在泛型类中使用类型参数的方法自动成为泛型方法

b 在非泛型类中也可定义泛型方法

语法：public<类型参数> [返回值][方法名](参数列表)

public class Comm {

public <T> T Gener(T t) { return t;}

public static void main(String[] args) {

Comm cm=new Comm();

System.out.println(cm.Gener(3.3)); System.out.println(cm.Gener("任何类型"));

System.out.println(cm.Gener(true)); System.out.println(cm.Gener(123));

System.out.println(cm.Gener(new Date())); }}

使用泛型的类型推断：编译器根据上下文推断出类型自变量的值，则允许写一个（前一个写）

GenericC<Integer> gci = new GenericC<>(); gci.setT(11);

类型限定(类或方法需对类型变量加以约束)

限定类型用&分隔，类型变量用逗号分隔,可有多个接口超类型，但至多只有一个类且在限定列表第一个

Public class Pair<T extends Student,U extends Comprable & Serializable>{}

泛型代码和虚拟机

虚拟机没有泛型类型对象即所有对象都属于普通类，所以无论何时定义一个泛型类型，都自动提供一个相应的原始类型：擦除类型变量替换为限定类型，类型变量有限定时原始类型为第一个限定的类型变量，而无限定时为Object

eg Public class Pair<T extends Student>{} 为Pair<Student>

Public class Pair<T>{} Pair<Object>

桥方法被合成来保持多态，为保持类型安全性，必要时插入强制类型转换

虚拟机编译器翻译泛型表达式：

调用原始的方法或属性(去掉类型变量)-🡪object类

将返回的object强制转换回对应的传入的类型

约束与局限性

不能用基本类型实例化类型参数eg 没有Pair<double>,只有Pair<Double>

运行时类型查询只适用于类型eg stringPair.getClass()===doublePair.getClass()

不能创建参数化类型的数组,但用来声明变量可以,只是不能实例化类型变量

eg Pair<String>[] ta=new Pair<String>[10]//error

泛型类的静态上下文中类型变量无效，所以禁止使用带有类型变量的静态域和方法

Eg public class Sing<T>{ private static T sing;} //error

不能抛出或捕获泛型类的实例 eg？？？

泛型类型的继承规则

泛型类可扩展或实现其他的泛型类

Eg ArrayList<Man>可转换为List<Man>,但不可转为ArrayList<Emp>或List<Emp>,尽管Man继承与Emp

解决办法：通配符

？ 所有的类

？ extends 类名n1 所有父类n1的子类，也包括子类的子类

？ super类名n2 所有子类n2的父类，也包括父类的父类

通配符不是类型变量，因此不能在编码用？作为一种类型 eg ？ t=p.getIn()

public class GeC<T> {

public static void main(String[] args) {

//GeC<Person> w=new GeC<Stu>();//泛型中不允许子类作为父类的子类型传递

GeC<?> w1 = new GeC<Stu>();

GeC<? extends Person> w2 = new GeC<IStu>();

GeC<? super IStu> w3 = new GeC<Person>();

GeC<? super IStu> w4 = new GeC<Stu>();

}

}

class Person {} class Stu extends Person {} class IStu extends Stu {} class Teacher extends Person {}

### 集合

集合---高级数组

数组的特点:数据类型一致,有固定长度,有序

集合的特点:数据类型可以不一样,长度不限定

List---接口特点:数据不唯一,有序{

ArrayList类 相当于可变长的数组

LinkedList类 同ArrayList操作基本一样 双向链表

Vector类 同ArrayList操作基本一样 线程安全(同步、排队)

public static void main(String[] args) {

ArrayList list1 = new ArrayList();// 如何声明ArrayList

List list2 = new ArrayList(); List<Integer> list3 = new ArrayList<Integer>();

list1.add("abc");// 添加数据 调用add方法 list3.add(10);

System.out.println("长度1;" + list3.size()); // 获取list的长度，调用size()方法

System.out.println("第二个元素" + list3.get(1)); // 获取数据 调用get方法

for (int i = 0; i < list3.size(); i++) { System.out.print(list3.get(i) + " "); }

for (int i : list3) { System.out.println(i);} //遍历list中的所有数据

list3.remove(2);// 按下标删除System.out.println("删除之后长度变为：" + list3.size());

// list3.clear();// 清空 // System.out.println("清空之后的长度："+list3.size());

list1.add(30); list1.addLast(50);// 末尾追加 list1.addFirst(11);// 前面添加

list1.peek();// 从头，返回第一个元素list1.pop();// 返回并删除元素

list1.push(11);// 添加数据到头部，相当于addFirst()

Vector<Integer> v1 = new Vector<>();// 迭代器

v1.add(11); v1.add(12); v1.add(13);

ListIterator li = v1.listIterator();// 遍历数据

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

Set---接口特点:数据唯一的,不允许重复,无序 Eg HahSet类 TreeSet类(基本操作与HashSet一致 它将数据自动排序的一个Set) HahSet比TreeSet查询慢，添加或删除快

public static void main(String[] args) {

HashSet hs1 = new HashSet();// 声明

Set hs2 = new HashSet(); Set<Integer> hs3 = new HashSet<>();

hs3.add(11);// 添加数据 hs3.add(14); hs3.add(12); hs3.add(11);

System.out.println(hs3.size());// 3个元素

for (int n : hs3) {System.out.println(n);}// 获取数据,结果为11 12 14

hs3.remove(11);// 删除数据 System.out.println(hs3.size());

Object[] a = hs3.toArray();// 转换为数组

if (hs3.contains(11)) {// 判断数据是否在Set集合中

System.out.println("在集合中");

} else {System.out.println("不在集合中");}

System.out.println(hs3.isEmpty());// 判断集合是否为空

// hs3.clear();//清空集合,慎用

Set<Integer> ts1 = new TreeSet<>();//TreeSet

ts1.add(11); ts1.add(14); ts1.add(13);

for (int n : ts1) {System.out.println(n); }// 结果为11 13 14(升序排序)

Set<String> ts2 = new TreeSet<>(); ts2.add("cc"); ts2.add("kk"); ts2.add("a");

for (String n : ts2) { System.out.println(n); }// 结果为a cc kk(升序排序)

}//中文是按照Unicode排序

Map---接口特点:键值对对象 K-V

HashMap类 key值唯一，允许有一个null的key值

TreeMap类 会将数据自动排序的一个Map key值不能为空

Hashtable类 同步、线程安全的，但不能有空对象(key值和value都不可以为空)

HashMap m1 = new HashMap();

Map<Integer, String> m2 = new HashMap<>();

m2.put(111, "aaa"); m2.put(111, "daa"); m2.put(112, "bbb");

m2.put(113, "ccc"); m2.put(114, "ddd");

System.out.println(m2.size());// 结果为4个

// 获取数据(key值唯一,也可为空 ,后边的会覆盖原来的 通过key值查找)

System.out.println(m2.get(111));// 结果为daa

// 获取到Map中的所有key值的集合（一个Set集合）

Set<Integer> ks = m2.keySet();

for (int k : ks) { System.out.println("key值:" + k + "value值:" + m2.get(k)); }

// 遍历方式二 Map中存放的数据是键值对(K-V) 一个键值对是一个Entry对象

// 则整个Map看做Set<Entry>

System.out.println();

Set<Entry<Integer, String>> ek = m2.entrySet();

for (Entry<Integer, String> e : ek) {

System.out.println("key值:" + e.getKey() + "value值:" + e.getValue()); }

// 通过key删除数据(整个键值对也没有了)，会返回被删除的键值对的value值，可实现备份

m2.remove(111);

System.out.println(m2.containsKey(111)); Map中是否有某一数据(通过key或value)

System.out.println(m2.containsValue("aaa"));

TreeMap mm1 = new TreeMap();

Map mm2 = new TreeMap(); Map<Integer, String> mm3 = new TreeMap<>();

mm3.put(1002, "qq"); mm3.put(1003, "sq"); mm3.put(1004, "dq");

// mm3.put(null,"dq");//TreeMap的key值不能为空

Set<Integer> ms = mm3.keySet();

for (int k : ms) {System.out.println("key值:" + k + "value值:" + mm3.get(k)); }

Hashtable ht = new Hashtable();

Map<Integer, String> ht2 = new Hashtable<>();

// ht2.put(null, "gfg");//key值不能为空 // ht2.put(1002, null);//value也不能为空

ArrayDeque队列 它实现了Deque接口(它允许从两端进行数据的添加或删除的操作)

ArrayDeque<Integer> ad1 = new ArrayDeque<>();// ArrayDeque 可变长的数组

ad1.add(11); ad1.addFirst(22); ad1.addLast(33);

for (Integer i : ad1) { System.out.println(i);}

System.out.println("队首数据" + ad1.getFirst()); //getLast());

ad1.removeFirst(); ad1.removeLast();

自定义集合的排序 Comparable接口和Comparator接口(可定义多条件排序)实现自定义规则

A实现接口,重写方法 B存入集合 C Collections.sort();方法 执行自定义排序

import java.util.Comparator;

public class Person implements Comparable {

private String name; private int age;//省略了set、get方法

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

public int compareTo(Object arg0) {//重写方法

Person p = (Person) arg0;// 将比较的对象转换为Person类型

if (this.age > p.age) { return 1;

} else if (this.age < p.age) { return -1;

} else { return 0; } }}

public static void main(String[] args) {

Person p1=new Person("sa",11); Person p2=new Person("ba",15);

ArrayList<Person> list=new ArrayList<>();

list.add(p1); list.add(p2);

System.out.println("没有自定义排序前");

for(Person p:list){System.out.println("名字："+p.getName()+"年龄："+p.getAge()); }

System.out.println("自定义排序后，"); Collections.sort(list);

for(Person p:list){System.out.println("名字："+p.getName()+"年龄："+p.getAge()); }}

## 多线程

线程（程序内的单序列控制流，分配系统资源的最小单位）

进程（它是计算机中的程序关于某数据集合上的一次运行活动，是系统进行资源分配和调度的基本单位，是操作系统结构的基础。进程是线程的容器。程序是指令、数据及其组织形式的描述，进程是程序的实体。）生命周期：创建(新建状态)—可运行，调用start方法并执行run方法---不可运行，sleep、interrupt等情况---死亡状态

A 创建线程类

1. 继承Thread类
2. 实现Runnable接口 只有抽象方法run方法 ，所以没办法自己启动，须借助Thread类

B 线程类的属性和方法 Thread类

Thread t = Thread.currentThread();//线程类的属性、方法

System.out.println("默认线程名：" + t.getName());

t.setName("主线程");// 设置线程名

System.out.println("默认优先级：" + t.getPriority());// 线程的优先级分1-10

t.setPriority(7); System.out.println("线程是否还活着:" + t.isAlive());

System.out.println("线程id:" + t.getId());

t.interrupt();// 中断 t.destroy();// 手动销毁线程

C 多线程的技术:多个线程，竞争资源、死锁、锁饥饿（缺点）；提高性能，提高CPU的利用率，降低资源消耗，简化代码（优点）

public class RGB extends JFrame implements Runnable {

JButton jbr, jbg, jbb; JLabel show;

public RGB() {//1实现红绿灯 三个方块 按时间切换 单线程

setSize(100, 250); setResizable(false); setLayout(null);

setDefaultCloseOperation(EXIT\_ON\_CLOSE);

getContentPane().setBackground(Color.black);

jbr = new JButton(); jbr.setBounds(10, 20, 30, 45); jbr.setBackground(Color.red);

jbg = new JButton(); jbg.setBounds(10, 70, 30, 45);jbg.setBackground(Color.green);

jbb = new JButton(); jbb.setBounds(10, 120, 30, 45);jbb.setBackground(Color.yellow);

add(jbr); add(jbg); add(jbb);

show=new JLabel(); show.setBounds(90, 160, 50, 80);

show.setForeground(Color.white); add(time); add(show); setVisible(true); }

public static void main(String[] args) {

RGB rgb = new RGB(); Thread t1 = new Thread(rgb); t1.start(); }

public void run() {

try { while (true) {// 无限循环 红灯先亮

jbr.setBackground(Color.red); jbg.setBackground(Color.gray);jbb.setBackground(Color.gray);

for (int i = 10; i >= 0; i--) {Thread.sleep(1000); show.setText(i+" ");}

onYellow();}}// 黄灯亮 // 绿灯亮

2setBounds方块移动通过判断线程名决定控制哪个方块 多线程

public class Fmove extends JFrame implements Runnable {

public static void main(String[] args) {

Fmove r1 = new Fmove();

Thread t1 = new Thread(r1); t1.setName("t1"); t1.start();

Thread t2 = new Thread(r1); t2.setName("t2"); t2.start();

Thread t3 = new Thread(r1); t3.setName("t3"); t3.start(); }

JPanel jp; JButton jbr, jbg, jbb;

public Fmove() {//界面设计参考红绿灯 }

public void run() { String name = Thread.currentThread().getName();

while (true) {

if ("t1".equals(name)) { for (int i = 0; i < 360; i++) {

jbr.setBounds(i, 20, 20, 20); try { Thread.sleep(50);

} catch (InterruptedException e) { e.printStackTrace(); }}

} else if ("t2".equals(name)) { for (int i = 0; i < 360; i++) {

jbg.setBounds(i, 50, 20, 20); try { Thread.sleep(100);

} catch (InterruptedException e) { e.printStackTrace(); }}

} else {for (int i = 0; i < 360; i++) {

jbb.setBounds(i, 80, 20, 20);try {Thread.sleep(5);

} catch (InterruptedException e) {e.printStackTrace();} }}}}}

解决多线程资源竞争，死锁，锁饥饿问题

同步(同一时间只有一个线程使用共享资源synchronized)

同步方法（加锁）每个对象都有一个监视程序，同一时间共享资源只能得到一个监视程序

同步语句块 同步某几行代码

public synchronized void run() {//操作共享资源的代码}

public class Syn implements Runnable {

public static int money = 10000;// 模拟共享资源

public static void main(String[] args) {

Syn s1 = new Syn();//对象一，

Thread t1 = new Thread(s1,"lily"); t1.start();

Thread t2 = new Thread(s1,"bob");t2.start();

Syn s2 = new Syn(); //对象二

Thread t3 = new Thread(s2,"hello");t3.start();}//不同对象会不同步

public synchronized void run() {//synchronized同步方法

for (int i = 0; i < 5; i++) { money += 2000;

System.out.println(Thread.currentThread().getName() + ",余额:"+money);

try { Thread.sleep(1000);} catch (InterruptedException e) { e.printStackTrace(); }}}

public void run() {// synchronized同步之后

//其他不需要同步的代码

synchronized (this) {//同步代码块

for (int i = 0; i < 5; i++) { money += 2000;

System.out.println(Thread.currentThread().getName() + ",余额:"+ money);

try { Thread.sleep(1000);} catch (InterruptedException e) {e.printStackTrace();}}}

//其他不需要同步的代码 }}

并发：多个任务同时执行concurrency

原子变量： 对于原子变量所做的任何更新都立即执行、完成，不会丢失

public class Concurrency implements Runnable {// 并行

final static Lock lock = new ReentrantLock();

public static void main(String[] args) {

AtomicInteger i = new AtomicInteger(22);它是在Java.utl.concurrency.atomic包中的一些类

System.out.println(i.get());// get()方法返回当前值 输出原子变量

i.getAndIncrement();//// 自增1 System.out.println(i.get());

i.getAndDecrement();// 自减1 System.out.println(i.get());

i.getAndAdd(5);// 自加5、-5 System.out.println(i.get());

i.compareAndSet(28, 100); // 如果i==28就把i的值更新为100，否则不做改变

System.out.println("比较之后的值：" + i.get());

Concurrency cc = new Concurrency();

Thread t1 = new Thread(cc, "t1"); t1.start();

Thread t2 = new Thread(cc, "t2"); t2.start();

Thread t3 = new Thread(cc, "t3"); t3.start(); }

public void run() { 锁（可重入锁）

do { try {// 抢锁，尝试着拥有这个锁，返回TRUE 则说明获得了锁

if (lock.tryLock(400, TimeUnit.MILLISECONDS)) {

System.out.println(Thread.currentThread().getName() + "正在使用锁....");

Thread.sleep(5000); lock.unlock(); System.out.println(Thread.currentThread().getName()+ "放开了这个锁");

} else {System.out.println(Thread.currentThread().getName()+ "没有获得，请等待");}

} catch (InterruptedException e) { e.printStackTrace(); }

} while (true); }}

执行器 执行一些并发的任务(task),任务分为Runnable任务(没有返回值)和Callable任务(有返回值) 1.Excutor 2. ExcutorService extends Excutor

线程：一旦调用start方法会立刻执行 任务：需要等待执行的条件

执行器一般使用大型线程池来执行Runnable或Callable任务 线程池对象比如acheThreadPool

实现分岔、汇合框架

// 找出 200个随机数字中的最大值// RecursiveTask 需要JDK高版本

public class MaxFind extends RecursiveTask<Integer> {

int [] data;// 接收要查找最大值的数组 int start; //开始下标 int end; // 结束下标

public MaxFind(int[] data ,int start,int end){

this.data = data; this.start = start; this.end = end; }

protected Integer compute() {// 真正的实现查找最大值的任务

System.out.println("开始:"+start+"---结束:"+end);

int length = end-start;// 总长

if(length <= 50){// 数组长度小于50，直接查找最大值

int max=-1; for(int i : data){if(i > max){max = i;} }return max;

}else{int mid = length/2;// mid当做分界点 对两段

MaxFind left = new MaxFind(data,start,start+mid);

left.fork();// 让left 使用同一个线程池中的其他线程 并行执行代码

MaxFind right = new MaxFind(data, start+mid, end);

return Math.max(right.compute(), left.join());} }

public static void main(String[] args) {

// ① 随机生成200个0-499之间的数据 存入data数组

int[] data = new int[200]; Random r = new Random();

for(int i=0;i<data.length;i++){ data[i]=r.nextInt(500); }// 随机产生

// ② 创建一个ForkJoinPool 本质上是一个线程池

ForkJoinPool pool = new ForkJoinPool(4);// 最多4个线程

// ③ 创建任务 Task 并调用线程池执行该任务

MaxFind max = new MaxFind(data, 0, data.length);//初始化任务

System.out.println("最大值是:"+pool.invoke(max)); }}

## I/O流

读取文件：使用输入流 和Reader类

FileInputStream ：read()方法读取每一个字节

byte [] buff=new byte[1024] read(buff) 返回读取的字节数 buff存放读取到的内容BufferedInputStream 自带buffer 缓冲区（需传入一个输入作为参数）次读取文件时会把内容存入缓冲区，以后再读取时先从缓冲区读取，若找不到再从文件读取(好处是缓冲区的速度要比直接从硬盘读取文件快

FileReader BufferedReader(带缓冲区的Reader) 按照读取字符的方式读取文件

写入文件：使用输出流 和Writer类

FileOututStream ：getByte()方法读取每一个字节

byte [] buff=new byte[1024] write(buff)

BufferedOutputStream 自带buffer 缓冲区（需传入一个输入作为参数）写入文件时会把内容存入缓冲区，再存入文件

FileWriter BufferedWriter (带缓冲区的Writer) 按照写入字符的方式写入文件

public class IO {

static FileInputStream fis = null;

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

try {fis = new FileInputStream("E:/io.txt"); int r = 0;// 读取文件

while ((r = fis.read()) != -1) { char c = (char) r;System.out.print(c);}

fis.close();

} catch (FileNotFoundException e) {e.printStackTrace();

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

readArray(); }

public static void readArray() {

try {FileInputStream fiss = new FileInputStream("E:/io.txt");

byte[] buff = new byte[1024];// 一次读取跟数组长度一样多的数据 返回值是字节数

System.out.println(fiss.read(buff)); fiss.close();

} catch (FileNotFoundException e) {e.printStackTrace();

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

public class BufferedReaderDemo {

public static void main(String[] args) {// 创建BufferedReader实例时需传Reader对象

try {FileReader fr = new FileReader("D:\\files\\niit.txt");

BufferedReader reader = new BufferedReader(fr);

String temp;// readLine每次读取一行数据 直到null 读取完毕

while((temp=reader.readLine())!=null){System.out.println(temp);}

reader.close();

} catch (FileNotFoundException e) {e.printStackTrace();

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

NIO:一些新的文件操作 Paths、Path（NIO要求必须创建Path对象）

创建类来实现FileVisitor接口 实现类执行walkFileTree()

public class Visitor implements FileVisitor<Path> {//遍历文件夹

public FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes attrs)throws IOException {// 访问目录之前

System.out.println("准备访问:" + dir);return FileVisitResult.CONTINUE; }

public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) throws IOException {// 在访问目录之时

System.out.println("正在访问:" + file);

System.out.println("是否是文件夹？" + Files.isDirectory(file));

return FileVisitResult.CONTINUE; }// 继续下一步操作

public FileVisitResult visitFileFailed(Path file, IOException exc)

throws IOException {// 访问目录失败时

System.out.println("准备访问" + file + "时失败");

return FileVisitResult.CONTINUE; }// 继续下一步操作

public FileVisitResult postVisitDirectory(Path dir, IOException exc)

throws IOException {// 访问目录之后

System.out.println(dir + "访问完毕"); return FileVisitResult.CONTINUE; }

public static void main(String[] args) {

Path ss=Paths.get("E:/nio.txt"); Visitor vv=new Visitor();

try { Files.walkFileTree(ss, vv); } catch (IOException e) {e.printStackTrace();}}}

## JDBC

java数据库连接 [(数据库基础)](数据库基础.docx)

驱动 jar包(不同的数据库使用不同的驱动 类似翻译) 导包

import java.sql.Connection; import java.sql.DriverManager; import java.sql.ResultSet;

import java.sql.SQLException; import java.sql.Statement;

java 连接数据库步骤：

加载驱动Class.forName("com.mysql.jdbc.Driver");

建立连接Connection conn= DriverManager.getConnection(url, user, password);

执行SQL语句

Statement stmt = conn.createStatement();//操作对象

ResultSet rs = stmt.executeQuery(sql);//查询操作，返回结果集

Int count = stmt.executeUpdate(sql);//更新操作，返回受影响的结果行数

Boolean b=stmt.execute(sql);//数据库操作（创建表或数据库等），返回是否操作成功

ResultSet 结果集 存放select语句的查询结果 结构类似于表

public class JDBC {

Connection conn = null;// 保存连接 ResultSet rs = null;// 结果集

Statement stmt = null;// 操作对象 int count = 0; String user = "root";

String url = "jdbc:mysql://localhost:3306/food";// 路径 String password = "123456";

public Connection getConnection() {return conn; }//返回数据库连接对象

public JDBC() {

try { Class.forName("com.mysql.jdbc.Driver");// 加载驱动

conn = DriverManager.getConnection(url, user, password);// 建立连接

} catch (ClassNotFoundException e) { e.printStackTrace();

} catch (SQLException e) { e.printStackTrace(); }}

public ResultSet executeQuery(String sql) {

try {// 创建Statement对象 执行sql语句 查询操作

stmt = conn.createStatement(); rs = stmt.executeQuery(sql);

} catch (SQLException e) { e.printStackTrace(); } return rs; }

public int excuteUpdate(String sql) {//更新操作（增删改）

try { stmt = conn.createStatement(); count = stmt.executeUpdate(sql);

} catch (SQLException e) { e.printStackTrace(); } return count; }}

public class Test {

public static void main(String[] args) { JDBC jd = new JDBC();

ResultSet rs = jd. executeQuery ("select \* from emp");//调用方法返回结果集

try {// rs.next()用来判断结果集中是否还有数据，若有则取出一条存入rs

int count = rs.getMetaData().getColumnCount();

while (rs.next()) {// 取值方式一

for (int i = 1; i <= count; i++) { System.out.print(rs.getObject(i));}

System.out.println(); // String s = "";//取值方式二

// s += rs.getInt(1);// 按照结果集的列数编列，不是照数据库中的列编列

// s += "\t" + rs.getString(2); // s += "\t" + rs.getInt("deptid");

// s += "\t" + rs.getInt("salary"); // s += "\t" + rs.getInt(5);

// s += "\t" + rs.getInt(6); // System.out.println(s); }

rs.close();

} catch (SQLException e) {e.printStackTrace();}

int count = jd.excuteUpdate("update emp set name='nainai'where id=1002");

if (count > 0) { System.out.println("更新成功！");}}}

PreparedStatement:带参数的sql语句的数据库操作对象

public class PreSta {

Connection conn = null; ResultSet rs = null; String user = "root"; int count = 0;

String url = "jdbc:mysql://localhost:3306/food"; String password = "123456";

PreparedStatement ps=null;

public Connection getConnection() { return conn; }

public PreSta() {

try { Class.forName("com.mysql.jdbc.Driver");// 加载驱动

conn = DriverManager.getConnection(url, user, password);// 建立连接

} catch (ClassNotFoundException e) { e.printStackTrace();

} catch (SQLException e) { e.printStackTrace(); }}

public ResultSet ruselt(String sql, Object... params) {

try {// 创建PreStatement对象 执行sql语句 查询操作(带可变参数)

ps = conn.prepareStatement(sql);

for (int i = 0; i < params.length; i++) { ps.setObject(i + 1, params[i]); }

rs = ps.executeQuery();//执行查询操作

} catch (SQLException e) {e.printStackTrace();}

Return rs; }

public int update(String sql, Object... params) {

try { ps = conn.prepareStatement(sql);

for (int i = 0; i < params.length; i++) { ps.setObject(i + 1, params[i]); }

count = ps.executeUpdate();//执行更新操作

} catch (SQLException e) {e.printStackTrace(); } return count; }}

public class Test {

public static void main(String[] args) {

ResultSet rs = null; int count = 0; PreSta ps = new PreSta();

String mysql = "select \* from emp where id=?";

String mysql2 = "select \* from emp where id=? and name=?";

String mysql3 = "select \* from emp where id=? and name=? and deptid=?";

rs = ps.ruselt(mysql,1002);

//rs = ps.ruselt(mysql2, 1001, "浅浅");//中文出不来

// rs = ps.ruselt(mysql3, 1002, "nainai", 2001);

try { while (rs.next()) { System.out.println(rs.getInt("salary")); }rs.close();

} catch (SQLException e) {e.printStackTrace();}

String mysql4 = "update emp set name=?,deptid=? where id=1001";

count = ps.update(mysql4, "dabao", 2002);

if (count > 0) { System.out.println("更新成功");}}}

public class Batcher {批处理Batcher

public static void main(String[] args) {

PreSta ps = new PreSta(); Connection conn = ps.getConnection();

try { Statement stmt = conn.createStatement();// 一个批处理

stmt.addBatch("update emp set name='qian',deptid=2001 where id=1001");

stmt.addBatch("update emp set name='liang',deptid=2002 where id=1004");

stmt.addBatch("update emp set name='susu',deptid=2001 where id=1105");

int[] result = stmt.executeBatch();// 执行批处理

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

System.out.println(result[i]); }//值为1表成功，0表失败stmt.close(); conn.close();

} catch (SQLException e) {e.printStackTrace();}}}

public class ProcedureDemo {执行存储过程procedure

public static void main(String[] args) {

PreSta ps = new PreSta(); Connection conn = ps.getConnection();

try { CallableStatement csmt=conn.prepareCall("call pro1");//调用不带参数的

int count =csmt.executeUpdate();

if(count>0){System.out.println("更新成功！"); }

CallableStatement csmt1=conn.prepareCall("call pro2(?,?)");//调用带参数

csmt1.setInt(1,1002); csmt1.setInt(2,55556);

int count1 =csmt1.executeUpdate();

if(count1>0){ System.out.println("更新成功！");}

csmt.close(); conn.close();

} catch (SQLException e) { e.printStackTrace(); }}}

源数据MetaData 不是表中的数据，而是数据库或表本身的相关信息

结果集源数据：结果集ResultSet的结构信息

数据库源数据： 数据库DataBase的结构信息

public class MetaDatas {

public static void main(String[] args) {

JDBC ps = new JDBC(); ResultSet rs = ps.ruselt("select \* from emp");

Connection conn = ps.getConnection();

try { // 结果集源数据

ResultSetMetaData md1 = rs.getMetaData();

System.out.println("结果集的总列数" + md1.getColumnCount());

System.out.println("结果集的第一列的列名" + md1.getColumnLabel(1));

System.out.println("结果集的第一列的类型" + md1.getColumnType(1));

System.out.println("第一列是否自增长：" + md1.isAutoIncrement(1));

System.out.println("第一列是否可为空null" + md1.isNullable(1));

System.out.println("第一列是否只读" + md1.isReadOnly(1));

while (rs.next()) {

for (int i = 1; i <= md1.getColumnCount(); i++) {System.out.print(rs.getObject(i) );}

System.out.println();} rs.close();

} catch (SQLException e) {e.printStackTrace();}

try { DatabaseMetaData dmd = conn.getMetaData();// 数据库源数据

// 获取food数据库中emp表中的所有列的信息集合

//但并不知道结果集来自哪张表无法确定rs1的结构

ResultSet rs1 = dmd.getColumns("food", null, "emp", null);

//获取rs1结果集的源数据，以便打印rs1的内容

ResultSetMetaData rmd=rs1.getMetaData();

while (rs1.next()) {

for (int i = 1; i < rmd.getColumnCount(); i++) {

System.out.print(rmd.getColumnLabel(i)+" "+rs1.getObject(i)+ " ");

}System.out.println();}

} catch (SQLException e) {e.printStackTrace();}}

## 网络编程

Xml: 可扩展标记语言（标记<> 可扩展:自定义标记）

* 指令 说明文件的属性，须写在第一行 <?xml version="1.0" encoding="utf-8"?>
* 元素 描素和识别数据的基本单元 <A>数据</A>

根元素:最外层的元素，一个XML文件只能有一个根元素

* 数据 元素标记的中间部分

数据类型： 字符/数字类型 元素 混合内容

* 属性 对于元素的补充说明，位置在开始标签内 属性名亦可自定义

<stu class="java2"> </stu>

属性和元素比较：若数据必须显示，具体数据写元素，抽象数据写属性；

若数据频繁更新最好使用元素；若数据被频繁核对最好使用属性

* 注释 <!—这里写注释的内容 -->
* 实体（特殊写法 转义字符）若数据中包含 < > "" &等符号须转换

<即&lt; >即&gt; ""即&quot; &即&amp

Schema 规定XML格式,它是一个特殊的XML文件，扩展名为.xsd，所以它也有自己的语法规定

xmlns:xsd=<http://www.w3.org/2001/XMLSchema>//说明当前xsd文件遵循该文件的语法

targetNamespace="http://www.niit.com.cn/NIItSchema" 目标名称空间，其他的xml文件如果想按照我们写的xsd文件规定的语法需要在在他的xml文件中引用该名称空间

<?xml version="1.0" encoding="utf-8"?>

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"

targetNamespace="http://www.food.com.cn/FoodSchema"

xmlns="http://www.food.com.cn/FoodSchema">

<xsd:simpleType name="sidType"><!-- sidType是我们自己定义的类型的名字 -->

<xsd:restriction base="xsd:int"><!--规定sidType具体内容,限定取值范围-->

<xsd:minExclusive value="1000"></xsd:minExclusive>

<xsd:maxInclusive value="999999"></xsd:maxInclusive>

</xsd:restriction>

</xsd:simpleType>

<xsd:complexType name="studentType"><!-- 2 复杂数据类型 -->

<xsd:sequence><!-- sequece规定了 子元素的顺序 -->

<xsd:element name="sid" type="sidType"></xsd:element> <xsd:element name="name" type="xsd:string"></xsd:element>

<xsd:element name="age" type="xsd:int"></xsd:element>

</xsd:sequence>

</xsd:complexType>

<!-- 指定xml文件的根目录 -->

<xsd:element name="stu" type="studentType"></xsd:element>

</xsd:schema>

<?xml version="1.0" encoding="UTF-8"?>

<food:stu xmlns:food="http://www.food.com.cn/FoodSchema"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.food.com.cn/FoodSchema foodSchema.xsd">

<sid>222222</sid> <name>sss</name> <age>11</age>

</food:stu>

XML解析 运用第三方框架

两种方式：只读和修改源文件

package PXML;//导包

public class Pxmls {

public static void main(String[] args) {

Pxmls pp = new Pxmls(); pp.Readxml(); pp.Writexml();

}

public void Readxml() {

Document dom;// 文档对象，对应的是整个XML文件

SAXReader rr = new SAXReader();// 工具类 专门负责读取XML文件

try {dom = rr.read("src/stu.xml");// 读取整个文件存入dom对象

Element root = dom.getRootElement();// 获取根节点

List nodes = root.elements();// 获取全部子节点

for (Object o : nodes) {Element e = (Element) o;

System.out.println("元素名：" + e.getName()+"元素内容：" + e.getText());

if ("address".equals(e.getName())) { List cl = e.elements();

for (Object ob : cl) {

System.out.println("元素名：" + e.getName()+"元素内容：" + e.getText());}}}

} catch (DocumentException e) { e.printStackTrace(); } }

public void Writexml() {

Document dom; SAXReader rr = new SAXReader();

try {dom = rr.read("src/stu.xml"); Element root = dom.getRootElement();

Element sex = root.addElement("sex"); sex.setText("男");

root.addAttribute("class", "java1");

FileOutputStream fs = new FileOutputStream("src/stus.xml");

OutputStreamWriter os = new OutputStreamWriter(fs, "utf-8");

XMLWriter ww = new XMLWriter(os); ww.write(dom); ww.close();

} catch (Exception e) { e.printStackTrace();}}}

<?xml version="1.0" encoding="UTF-8"?>

<stu> <sid>222</sid> <name>ss</name> <age>18</age>

<address> <pro>贵州</pro> <city>贵阳</city> </address></stu>

<?xml version="1.0" encoding="UTF-8"?>

<stu class="java1"> <sid>222</sid> <name>ss</name> <age>18</age>

<address> <pro>贵州</pro> <city>贵阳</city> </address>

<sex>男</sex></stu>

## 类加载和反射、、、、