# (一)Java基础语法部分

## (1)Java概述

|  |
| --- |
| // JDK是Java开发工具包，用于开发Java程序。  // JRE是Java运行时环境，用于运行Java程序，JRE是平台有关的。  // Java是什么?  // Java是一个跨平台的编程语言，Java是一个纯软件平台。  // Java有什么?  // JavaSEJava标准版，定位在桌面应用程序，同时是其余两个平台的基础。  // JavaMEJava缩微版，定位在移动设备电子产品，例如Android系统的应用和游戏。  // JavaEEJava企业版，定位在大中型企业级的应用。  // Java应用程序的入口  **publicstaticvoid** main(String[] arr) {  System.*out*.println("第一个Java应用程序");  } |

## (2)Java的八种基本数据类型

|  |
| --- |
| **publicstaticvoid** main(String[] args) {  // Java的八种基本数据类型  // 整数类型的有4种，按照存储范围由小到大依次为：byte、short、int、long  // 默认情况下使用int类型  // (1)byte字节类型，占8位  **byte** b = 1;  System.*out*.println("byte字节类型的值是：" + b);  // (2)short短整型，占16位  **short** s = 10;  System.*out*.println("short短整型的值是：" + s);  // (3)int整型，占32位  **int** i = 100;  System.*out*.println("int整型的值是：" + i);  // (4)long长整型，占64位  **long** g = 1000;  System.*out*.println("long长整型的值是：" + g);  // 浮点数类型有2种，按照存储范围由小到大依次为：float、double  // 默认使用double类型  // (5)float单精度浮点数，占32位，实际值比int大  // 注意：float的数值需要添加F或者f指定是float类型  **float** f = 1.5F;  System.*out*.println("float单精度浮点数的值是：" + f);  // (6)double双精度浮点数，占64位，实际值比long大  // 注意：D表示double类型  **double** d = 1.23D;  System.*out*.println("double双精度浮点数的值是：" + d);  // (7)char字符类型，单引号，只能存储单个数字、单个字母、单个汉子、占16位  **char** c = '你';  System.*out*.println("char字符类型的值是：" + c);  // (8)boolean布尔类型，只有true或者false两个值  **boolean** boo = **true**;  boo = **false**;  System.*out*.println("boolean布尔类型的值是：" + boo);  } |

## (3)变量与常量

|  |
| --- |
| **publicstaticvoid** main(String[] args) {  // Java的八种基本数据类型的变量可以理解为是一个容器，可以改变的量。  // 常量使用关键字final定义，表示恒定的量，不能被修改。  **int** a = 2;  **int** b = 3;  // **TODO**让两个变量中的值互换  **int** t = a;  a = b;  b = t;  System.*out*.println(a);  System.*out*.println(b);  // 在方法中定义的变量叫做局部变量，局部变量必须要声明而且初始化之后才能使用。  } |

## (4)基本数据类型之间的转换

|  |
| --- |
| **publicstaticvoid** main(String[] args) {  // (1)同类型的转换，低精度赋值给高精度没问题，高精度赋值给低精度的需要强制类型转换  **int** i = 10;  **short** s = (**short**) i;  System.*out*.println(s);  // (2)不同类型的转换，浮点数转换为整数的时候需要注意强制类型转换  **double** d = 1.34;  **long** g = (**long**) d;  System.*out*.println(g);  **double** dd = 10L;  // (3)int和char之间的转换  **int** v = 20320;  **char** c = (**char**) v;  System.*out*.println(c);  } |

## (5)运算符

### (5.1)赋值运算符

|  |
| --- |
| // (1)赋值运算符  **int** a = 2;  // a = a + 3;  a += 3;  System.*out*.println(a);  **short** s = 10;  // 当数据类型不一致进行计算时，Java会先将低精度的数值往高精度转换，然后再计算。  // s = (short)(s + a);  s += a;  System.*out*.println(s); |

### (5.2)算数运算符

|  |
| --- |
| // (2)算数运算符  System.*out*.println(3 + 2);  System.*out*.println(3 - 2);  System.*out*.println(3 \* 2);  System.*out*.println(3.0 / 2);  System.*out*.println(3 % 2);  // **TODO**练习：打印出下面五位数的的各个位数的值  **int** value = 12345;  System.*out*.println("个位：" + value % 10);  System.*out*.println("十位：" + value % 100 / 10);  System.*out*.println("百位：" + value / 100 % 10);  System.*out*.println("千位：" + value / 1000 % 10);  System.*out*.println("万位：" + value / 10000); |

### (5.3)比较运算符

|  |
| --- |
| // (3)比较运算符，结果一定boolean类型的true或者false  System.*out*.println(2 > 3);  System.*out*.println(2 >= 3);  System.*out*.println(2 < 3);  System.*out*.println(2 <= 3);  System.*out*.println(2 == 3);  System.*out*.println(2 != 3); |

### (5.4)逻辑运算符

|  |
| --- |
| // (4)逻辑运算符，true和false的运算  // 逻辑非，取反  // System.out.println(!true);  // 逻辑与，两边都为true结果为true，其余情况都为false  // System.out.println(true && false);  // System.out.println(false && true);  // System.out.println(true && true);  // System.out.println(false && false);  // 逻辑或，两边有一边为true结果为true，其余情况都为false  System.*out*.println(**true** || **false**);  System.*out*.println(**false** || **true**);  System.*out*.println(**true** || **true**);  System.*out*.println(**false** || **false**); |

### (5.5)自增自减运算符

|  |
| --- |
| // (5)自增自减运算符  **int** i = 3;  // ++i表示先自增1，然后再参与计算，--i同理  // i++表示先参与计算，然后再自增1，i--同理  System.*out*.println((3 == i++)); |

### (5.6)三目运算符

|  |
| --- |
| // (6)三目运算符?:  **int** a = 2;  **int** b = 3;  **int** max = (a > b) ? a : b;  System.*out*.println(max);  // **TODO**练习3：使用三目运算符找出3个变量中的最大值  **int** v1 = 2;  **int** v2 = 3;  **int** v3 = 5;  **int**m = (v1 > v2) ? v1 : v2;  m = (m> v3) ? m : v3;  System.*out*.println(m); |

## (6)流程控制语句

### (6.1)if分支语句

|  |
| --- |
| // (1)if单分支语句，特点是语句块有可能不执行  **if** (**true**) {  System.*out*.println("if单分支语句");  }  // (2)if两个分支语句，至少执行一个分支  **if** (**false**) {  System.*out*.println("if两个分支语句true");  } **else** {  System.*out*.println("if两个分支语句false");  }  // **TODO**练习4：求两个变量中的最大值  **int** a = 2;  **int** b = 5;  **if** (a > b) {  System.*out*.println(a);  } **else** {  System.*out*.println(b);  }  // (3)if多个分支语句，最多只能执行一个分支，而且只能执行一个分支  **if** (**true**) {  System.*out*.println("1");  } **elseif** (**true**) {  System.*out*.println("2");  } **else** {  System.*out*.println("3");  }  // **TODO**练习5：三个变量中找最小值  **int** x = 2;  **int** y = 5;  **int** z = 9;  **if** (x < y && x < z) {  System.*out*.println(x);  } **elseif** (y < x && y < z) {  System.*out*.println(y);  } **else** {  System.*out*.println(z);  } |

### (6.2)switch分支语句

|  |
| --- |
| // switch分支语句中的类型必须是：byte、short、int、char  // break关键字用于跳出switch语句  **int** i = 85;  **switch** (i / 10) {  **case** 9: {  System.*out*.println("优");  **break**;  }  **case** 8: {  System.*out*.println("良");  **break**;  }  **case** 7: {  System.*out*.println("中");  **break**;  }  **case** 6: {  System.*out*.println("差");  **break**;  }  **default**: {  System.*out*.println("不及格!");  }  } |

### (6.3)for循环语句

|  |
| --- |
| // for循环  // for (int i = 1; i <= 10; i++) {  // System.out.println(i);  // }  // **TODO**练习1：求1 + 2 + 3 + ... 10的和  // int s = 0;  // for (int i = 1; i <= 10; i++) {  // s = s + i;  // }  // System.out.println(s);  // **TODO**练习2：求2 \* 4 \* 6 \* 8的乘积  // int m = 1;  // for (int i = 2; i <= 8; i += 2) {  // m \*= i;  // }  // System.out.println(m);  // **TODO**练习3：求2 + 4 + 10 + 14的和  **int** s = 0;  **for** (**int** i = 2; i <= 10; i += 2) {  **if** (i == 6 || i == 8 || i == 12) {  **continue**;  }  s = s + i;  }  // 循环中的两个关键字：break关键字和continue关键字。  // break关键字：跳出当前循环、跳出switch语句。  // continue关键字：终止当前这次循环，直接执行下一次循环。  // for (int i = 1; i < 10; i++) {  // if (i == 3) {  // break;  // // continue;  // }  // System.out.println(i);  // } |

### (6.4)while循环语句

|  |
| --- |
| **int** i = 1;  **while** (i < 10) {  System.*out*.println(i);  i++;  } |

### (6.5)do while循环语句

|  |
| --- |
|  |

## (7)数组

|  |
| --- |
| // Java中的类型有两种：一是基本数据类型。二是类类型。  **publicstaticvoid** main(String[] args) {  // 一维数组  // (1)创建  // int[] arr1 = { 1, 2, 3 };  // int[] arr2 = new int[10];  // (2)通过下标存取  // int[] arr = new int[3];  // arr[0] = 1;  // arr[1] = 2;  // int x = arr[0];  // (3)遍历数组  // int[] arr = { 1, 2, 3 };  // for (int i = 0; i < arr.length; i++) {  // System.out.println(arr[i]);  // }  // 二维数组(用一维数组的方式来理解二维数组)  // (1)创建  // int[][] arr = { { 1, 2 }, { 3, 4 }, { 5 } };  // int[][] arr1 = new int[2][];  // int[][] arr2 = new int[2][3];  // (2)下标存取  // int[][] arr = new int[2][3];  // arr[0][0] = 5;  // int val = arr[0][0];  // (3)遍历二维数组  // int[][] arr = { { 1 }, { 4, 5 }, { 1, 2, 3 } };  // for (int i = 0; i < arr.length; i++) {  // for (int j = 0; j < arr[i].length; j++) {  // System.out.print(arr[i][j]);  // }  // System.out.println();  // }  // 二维数组练习题：打印杨辉三角  **int**[][] arr = **newint**[10][2];  **for** (**int** i = 0; i < arr.length; i++) {  arr[i] = **newint**[i + 1];  }  **for** (**int** i = 0; i < arr.length; i++) {  arr[i][0] = 1;  arr[i][i] = 1;  }  **for** (**int** i = 2; i < arr.length; i++) {  **for** (**int** j = 1; j < arr[i].length - 1; j++) {  arr[i][j] = arr[i - 1][j] + arr[i - 1][j - 1];  }  }  **for** (**int** i = 0; i < arr.length; i++) {  **for** (**int** j = 0; j < arr[i].length; j++) {  System.*out*.print(arr[i][j] + "\t");  }  System.*out*.println();  }  } |

## (8)函数

|  |
| --- |
| // 函数需要掌握：  // 调用函数：根据定义调用函数。注意：方法名和參數列表必須完全一致。  // 定义函数：根据需求写出函数定义。  // 现阶段的函数定义格式：  // (1)没有返回值的函数方法签名  // public static void 方法名(参数列表) { 方法体 }  // (2)有返回值的函数方法签名，return关键字返回到方法的调用处。  // public static 返回类型方法名(参数列表) { 方法体 return 返回值;}  **publicstaticint** m1(**int** a, **int** b) {  **int** m = (a > b) ? b : a;  **return** m;  }  **publicstaticint** m2(**int** a, **int** b, **int** c) {  **int** m = *m1*(*m1*(a, b), c);  **return** m;  }  **publicstaticvoid** main(String[] arr) {  **int** max = *m2*(2, 3, 5);  System.*out*.println(max);  } |

# (二)面向对象的分析和设计OOAD

## (2.1)类与对象

|  |
| --- |
| /\*\*  \*  \* (1)类与对象的关系  \*  \* **@author**赵阳  \*  \* 2014-5-30 上午10:29:17  \*/  **publicclass** A {  **private** String name = "张三";  **privateint**age = 30;  // 类：图纸，模板。  // 类中有类的成员变量和类的成员方法。  // 成员变量：在类中定义的变量叫做成员变量，用来表示类的属性和状态。  // 成员方法：在类总定义的方法叫做成员方法，用来表示类的动作和行为。  // 构造方法：方法名和类同名，没有返回值，也不加void关键字。  // 构造方法的作用：一是创建对象，二是初始化。  // 构造方法注意：如果不写构造方法，系统将提供一个默认的无参的构造方法。  **public** A() {  }  **public** A(**int** a) {  }  // 对象：实例，实体。  **publicstaticvoid** run() {  System.*out*.println("跑步");  }  **publicstaticvoid** eat() {  System.*out*.println("吃饭");  }  **publicstaticvoid** main(String[] args) {  // 创建对象的语法格式：  // 类的名字变量名字 = new关键字类的构造方法();  A a = **new** A(4);  System.*out*.println(a.name);  System.*out*.println(a.age);  a.*eat*();  a.*run*();  }  } |

## (2.2)引用与实例

|  |
| --- |
| /\*\*  \*  \* (2)引用与实例的关系  \*  \* **@author**赵阳  \*  \* 2014-5-30 上午11:19:26  \*/  **publicclass** B {  **publicstaticvoid** main(String[] args) {  // Java中的类型有2种：八种基本数据类型，类类型(引用类型)。  // 含义：创建一个类的对象，返回对象的内存地址存储到变量中，因此类类型的变量存储的是对象的引用。  // B b = null;  // b = new B();// 此对象会被java的垃圾回收机制收回GC  // b = new B();    B b1 = **new** B();  B b2 = b1;  B b3 = b1;  // 引用与实例的关系：  // 一个引用可以引用零个或一个对象。  // 一个对象可以同时被多个变量引用。  }  } |

## (2.3)值传递

|  |
| --- |
| **publicclass** C {  **publicstaticvoid** m1(**int**[] arr) {  // arr = new int[3];  arr[0] = 4;  }  **publicstaticvoid** main(String[] args) {  **int**[] arr = { 1, 2, 3 };  *m1*(arr);  System.*out*.println(arr[0]);  // Java中的值传递：  // 基本数据类型传递的是实实在在的值。  // 类类型传递的是地址的值。  }  } |

## (2.4)成员变量和局部变量

|  |
| --- |
| /\*\*  \* **@author**赵阳  \*  \* 2014-6-3 上午9:19:35  \*  \* 成员变量和局部变量  \*  \* 成员变量：在类中定义的变量，作用域在类中都可见。  \* 系统会默认赋值。  \* 局部变量：在方法或语句块中定义的变量，作用域在方法中都可见。  \* 必须初始化才能使用。  \*  \*/  **publicclass** F {  **int**age;  {  **int** c = age;  System.*out*.println(age);  }  **publicstaticvoid** main(String[] args) {  **int** a = 2;  System.*out*.println(a);  }  } |

## (2.5)关键字static

|  |
| --- |
| // static静态关键字  // 被static修饰的成员属于类，称为类成员静态成员，因此通过“类.成员”的方式来调用。  // 不被static修饰的成员属于对象，称为对象成员非静态成员，因此通过“对象.成员”的方式来调用。  // 静态成员被所有对象共享。  **publicclass** A {    **staticint***a* = 20;    **int**b = 30;    **staticvoid** m1() {    }    **void** m2() {    }    **publicstaticvoid** main(String[] args) {    System.*out*.println(A.*a*);  A.*m1*();    A test = **new** A();  System.*out*.println(test.b);  test.m2();    System.*out*.println(test.*a*);  test.*m1*();    }  }  // 类的初始化顺序：加载静态-》加载非静态-》调用构造方法  // 静态不能包含非静态的内容。  **publicclass** A {  **publicvoid** m(**int** a, **int** b) {  }  **publicstaticvoid** main(String[] args) {  // m(1, 2);  // A.m(1, 2);    A a = **new** A();  a.m(1, 2);  }  } |

## (2.6)关键字final

|  |
| --- |
| // final常量关键字  // (1)被final修饰的变量是常量，不能被修改。  //注意：被final修饰的变量必须在声明的同时初始化，因为final变量在编译器加载。  //问题：为何static和final通常连用?  // (2)被final修饰的方法不能被覆写。  // (3)被final修饰的类不能被继承。  **publicclass** B {  **finalstaticint***b* = 3;  **publicstaticvoid** main(String[] args) {  **finalint** a = 2;  }  } |

## (2.7)关键字this和super

|  |
| --- |
| // this关键字表示当前类的对象  // this的作用：用于区分成员变量和局部变量。用于构造方法之间的调用，入口一致。  **publicclass** Person {  **private** String name;  **privateint**age;  **public** Person() {  **this**("");  }  **public** Person(String name) {  **this**(name, 0);  }  **public** Person(String name, **int** age) {  **this**.name = name;  **this**.age = age;  }  **public** String getName() {  **return**name;  }  **publicvoid** setName(String name) {  **this**.name = name;  }  **publicint** getAge() {  **return**age;  }  **publicvoid** setAge(**int** age) {  **this**.age = age;  }  **public** String toString() {  **returnthis**.name + "\t" + **this**.age;  }  **publicstaticvoid** main(String[] args) {  // 重点是OOAP  Person p1 = **new** Person();  p1.setName("张三");  p1.setAge(20);  System.*out*.println(p1);    Person p2 = **new** Person("李四");  p2.setAge(30);  System.*out*.println(p2);    Person p3 = **new** Person("王五", 40);  System.*out*.println(p3);  }  }  // super概念：super关键字表示父类的引用  // super作用：调用父类的构造方法，调用父类的被覆写的方法。  // 注意：  // 如果一个类不写构造方法，系统将提供一个默认公共的无参构造方法。  // 子类的构造方法第一句会默认调用super()  // 好的习惯：最好每个类中都写一个无参的构造方法。 |

## (2.7)封装

|  |
| --- |
| // Java是一个面向对象的语言，它三个特征  // (1)封装(2)继承(3)多态  // 封装的理解：(1)Java将相关的变量和方法封装到类中。  //(2)Java通过可见性来实现封装。  // 可见性有4个范围3个关键字  // private关键字：当前类可见。  // 默认：当前类可见，本包可见。  // protected关键字：当前类可见，本包可见，外包子类可见。  // public关键字：都可见。  // 修饰class的可见性要么是public，要么是本包可见  **publicclass** B {  **protectedint**age = 30;  **publicstaticvoid** main(String[] args) {  (**new** B()).age = 10;  }  } |

## (2.8)String

|  |
| --- |
| **publicstaticvoid** main(String[] args) {  //http://www.cnblogs.com/ydpvictor/archive/2012/09/09/2677260.html 字符串到底有几个对象  //http://www.cnblogs.com/A\_ming/archive/2010/04/13/1711395.html StringBuffer,StringBuilder,String  // 字符串的创建  // String s1 = "你好";// 建议使用字面值  // String s2 = new String("你好");  // 字符串的连接  // String s = "a";  // s += "b";  // s += "c";  // System.out.println(s);  // 连接频繁的时候使用StringBuffer  // StringBuffer s = new StringBuffer();  // s.append("a");  // s.append("b");  // s.append("c");  // System.out.println(s);  // 字符串的本质是字符数组  // String s = "123456789";  // for (int i = 0; i < s.length(); i++) {  // System.out.println(s.charAt(i));  // }  // 要求至少记住String中的10个方法  String str = "abc";  // str.length()字符串长度  // str.charAt(i)获取字符  // str.toLowerCase()小写  // str.toUpperCase()大写  // str.trim()去除两边空格  // str.substring(1, 2)截取字符串  // str.startsWith("a")以开头  // str.endsWith("c")以结尾  // str.equals("abc")判断值相等  // str.indexOf("bc")获取下标  }  // Java中的类型有两种：8种基本数据类型、类类型  // String字符串类，它的使用频率比8种基本数据类型还要高。非常重要。  // 创建字符串对象方式  // String str1 = "abc";// 强烈推荐采用此种方式创建字符串  // String str2 = new String("abc");  // 第一种方式采用的是"字面值"，每次使用Java会先检查内容中是否有此字符串，如果有直接引用。  // 第二种每次都会创建新的对象。  // String a = "abc";  // String b = new String("abc");  // System.out.println(a == b);  // String str1 = "abc";  // String str2 = "abc";  // (1)equals方法：比较两个字符串的值是否相等  // System.out.println(str1.equals(str2));  // (2)substring方法：截取方法  // 字符串的本质是char[]，因此下标从0开始  // 一个参数：值>=i  // 两个参数：i <= 值< j  // String s = "abcde";  // System.out.println(s.substring(4));  // System.out.println(s.substring(1, 3));  // (3)toUpperCase方法转换大写  // toLowerCase方法转换小写  // String s = "aBcD";  // System.out.println(s.toUpperCase());  // System.out.println(s.toLowerCase());  // (4)startsWith判断是否以某字符串开头，是返回true，不是返回false  // endsWith判断是否以某字符串结尾，是返回true，不是返回false  // String s = "abc";  // System.out.println(s.startsWith("a"));  // System.out.println(s.endsWith("bc"));  // (5)charAt方法根据下标返回对应的字符  // String s = "abcdef";  // char c = s.charAt(2);  // System.out.println(c);  // **TODO**打印出给定字符串的各个字符  // length()方法返回的是字符串的长度  // for (int i = 0; i < s.length(); i++) {  // System.out.println(s.charAt(i));  // }  // (6)trim方法可以去除字符串两端的空格  // String s = " abc ";  // s = s.trim();  // System.out.println(s.length());  // System.out.println(s);  // (7)indexOf方法搜索字符串，找得到则返回起始下标，找不到返回-1  // String s = "abcde";  // int index = s.indexOf("bd");  // System.out.println(index); |
|  |

## (2.9)抽象类

|  |
| --- |
| // (1)抽象类：被abstract关键字修饰的类叫做抽象类。  **publicabstractclass** A {  // (4)构造方法：抽象类中有构造方法。  // 作用：抽象类不能创建对象，所以构造方法的作用就是初始化。  // 初始化谁？子类  **public** A() {  }  // (2)抽象类中的成员变量和类中的成员变量相同。  **protectedstaticfinalint***a* = 2;  // (3)抽象类中的成员方法：  // 抽象方法：被abstract关键字修饰的并且没有方法体的方法叫做抽象方法。  // 类中方法和抽象类中方法的区别：  // 类中的方法的特点是：都是实现的方法！！！！！  // 抽象类中的方法的特点是：部分实现!!!!!!!!!!!!  **publicabstractvoid** m1();  **publicvoid** m2(String a, **double** d) {  }  **publicstaticvoid** main(String[] args) {  //A a = new A();  }    }  // 抽象类概念：被abstract关键字修饰的class叫做抽象类  **publicabstractclass** A {  // 成员变量：abstract class中的成员变量和class中相同。  // 抽象方法：被abstract关键字修饰并且没有方法体的方法叫做抽象方法。  **publicabstractvoid** m1();  // 实现方法：有方法体的叫做实现方法。  **publicvoid** m2() {  // 空实现  }  // 类和抽象类中方法的区别总结：  // 类中的方法必须都是实现的方法。  // 抽象类中的方法可以是抽象方法，也可以是实现方法。  // 包含抽象方法的类一定是抽象类，抽象类中不一定包含抽象方法。  // 抽象类中有构造方法，但是不能创建对象。  **public** A() {  }  // Java是一个面向对象的语言，而抽象类还不能创建对象，要抽象类干嘛??!!  // 抽象类的目的：设计(继承)  // 类的目的：实现  } |

## (2.9)继承

|  |
| --- |
| **// 当定义一个类时，该类会默认继承Object，Java中的单根继承**  **publicstaticvoid** main(String[] args) {  // (1)创建子类的对象  // Child c = new Child();  // System.out.println(c.getName());  // c.m();  // (2)创建父类的对象  // Father f = new Father();  // System.out.println(f.getName());  // f.m();  // (3)重点：建议使用父类声明变量，来应用子类对象。  // 为什么?  // Java思想：面向抽象类做开发的思想。  // Father f2 = new Child2();  // f2.m();  // (4)类和抽象类是可以相互继承的  // 重点：类继承抽象类。  // (5)继承中的初始化顺序  // 父类静态: 子类静态->父类非静态->父类构造->子类非静态->子类构造  **new** C();    // (6)抽象类和继承的关系    }  // 继承：父类有的子类都有。  // 继承原则：满足“是一个”的原则。  // Java中所有的类都是直接或者间接的继承Object类，Java中是单根继承，Object是顶级父类。  // 如果不指定父类，则新定义的类会默认继承Object。  // 继承的讨论：类和抽象类之间可以相互继承，因为都是class类。  // (1)class可以继承class  // (2)class可以继承abstract class!!!开发中的最常用!!!  // (3)abstract class可以继承class  // (4)abstract class可以继承abstract class |

## (2.10)接口

|  |
| --- |
| // 接口概念：使用interface关键字定义的叫做接口。  **publicinterface** A {  // 接口中的成员变量：接口中的变量默认被public static final修饰。  **publicstaticfinalint***age* = 20;  // 接口中的成员方法：必须都是抽象方法，默认被public abstract修饰。  **publicabstractvoid** m();  // 类、抽象类、接口方法区别：  // 类中方法：必须都是实现方法。  // 抽象类中方法：可以是实现方法，也可以是抽象方法。只有抽象类可以部分方法实现!!!  // 接口中方法：必须都是抽象方法。  // 接口中没有构造方法，不能创建对象。  // 接口目的：设计、更纯粹的设计。  // 类和抽象类之间的关系：继承extends。注意：Java中类的继承每个类只能有一个直接父类。  // 类抽象类和接口之间的关系：实现implements。注意：类可以同时实现多个接口。  // 接口和接口之间的关系：继承extends。注意：Java中接口的继承可以多继承。  }  补充：  // 开发中常见的使用方式一：设计一个类继承一个父类同时实现多个接口。  **publicclass** C **extends** B **implements** A, E, F {  @Override  **publicvoid** ma() {  System.*out*.println("实现接口中的方法");  }  @Override  **publicvoid** mb() {  System.*out*.println("实现抽象类中的方法");  }  }  补充：  **publicinterface** A {  **publicabstractvoid** m1();  **publicabstractvoid** m2();  }  **publicabstractclass** B **implements** A {  @Override  **publicvoid** m1() {  System.*out*.println("实现方法一");  }  @Override  **publicabstractvoid** m2();  }  **publicclass** C **extends** B {  @Override  **publicvoid** m2() {  System.*out*.println("实现方法二");  }  } |

## (2.11)多态

|  |
| --- |
| //多态：  //(1)类内部多态(重载):  // 在一个类的内部，如果有多个方法的方法名相同参数列表不同，叫做重载。  // 典型的方法就是构造方法、String.valueOf方法  //(2)继承中多态(覆写):  // 在父类和子类中，如果父类和子类的方法的方法签名完全相同，将会调用子类的方法，叫做覆写。  // 典型的方法就是Object中的toString方法  补充1：  **publicstaticvoid** main(String[] args) {  // 当父类和子类中定义了相同的变量  // (1)直接访问同名变量：此时会调用类型中的变量的值。  // B b1 = new B();  // System.out.println(b1.name);  //  // A b2 = new B();  // System.out.println(b2.name);  // (2)通过方法访问同名变量：此时会调用方法所在的变量的值。  // B b = new B();  // System.out.println(b.getName());  }  补充2：  // 当发生覆写时，子类方法的可见性必须>= 父类的可见性 |

# (三)JDK API

## (3.1)集合

|  |
| --- |
| /\*\*  \*  \* 数组缺点：每次创建都要指定长度。  \*  \* Java中的集合构成：6个接口和一个工具类组成，都在java.util包下。  \*  \*  \* Collection：集合顶级interface、定义了一个集合所具备的通用抽象方法。  \* List：标准是有序。ArrayList实现类。  \* Set：标准是不重复。HashSet实现类。  \* SortedSet：标准是不重复，排序。TreeSet实现类。  \*  \* Map  \* SortedMap  \*  \* Collections集合工具类可以对集合排序  \* java.lang.Comparable  \*  \* **@author**赵阳  \*  \* 2014-6-13 上午9:28:17  \*/  **publicclass** A {  **publicstaticvoid** main(String[] args) {    // List接口    // 创建了一个ArrayList集合对象  // 泛型：可以指定集合中存储的类型，在编译器检查，强烈建议使用泛型。  List<String> list = **new** ArrayList<String>();    // 添加元素  list.add("a");  list.add("b");  list.add("c");    // 删除元素  // list.remove("b");  // list.remove("c");    // 更新元素  //list.set(1, "你好");    // ArrayList低层是通过数组实现的，因此获取是通过下标获取的。  // System.out.println(list);  // String s1 = (String)list.get(0);  // System.out.println(s1);    // 遍历List集合    // 第一种遍历方法通过下标遍历  **for** (**int** i=0; i < list.size(); i++) {  System.*out*.println(list.get(i));  }  **for**(**int** i=0;i<list.size();i++){    list.remove(i);  --i;  }  // 第二种遍历方法简单遍历  **for** (String s:list) {  System.*out*.println(s);  }    // 第三种遍历方法：迭代器遍历，所有集合通用的遍历方法，建议采用此种遍历。  // 将List集合返回到一个迭代器中(设计模式之一：迭代器设计模式)  Iterator<String> iterator = list.iterator();  **while** (iterator.hasNext()) {  String s = iterator.next();  System.*out*.println(s);  }      }  }  **publicstaticvoid** main(String[] args) {  // Set接口  // 创建一个Set集合  Set<String> set = **new** HashSet<String>();  // 添加  set.add("a");  set.add("b");  set.add("c");  // 删除  set.remove("a");  System.*out*.println(set);  // 遍历Set集合方法  Iterator<String> iterator = set.iterator();  **while** (iterator.hasNext()) {  String s = iterator.next();  System.*out*.println(s);  }  }  **publicstaticvoid** main(String[] args) {  // SortedSet接口  SortedSet<String> set = **new** TreeSet<String>();  set.add("d");  set.add("b");  set.add("a");  set.add("a");  set.add("a");  set.add("c");  System.*out*.println(set);  Iterator<String> iterator = set.iterator();  **while** (iterator.hasNext()) {  String s = iterator.next();  System.*out*.println(s);  }  }  **publicstaticvoid** main(String[] args) {  // Map接口  // Map<String, String> map = new HashMap<String, String>();  SortedMap<String, String> map = **new** TreeMap<String, String>();  // 添加一组键值对(key-value)  map.put("2", "张三");  map.put("1", "李四");  map.put("3", "王五");  // 根据key获取value  String name = (String) map.get("1");  System.*out*.println(name);  // 获取key的集合  // Set<String> set = map.keySet();  // System.out.println(set);  // 遍历map集合方法一  // for (String key : set) {  // System.out.println(key + "\t" + map.get(key));  // }  // 遍历map集合方法二  Set<Entry<String, String>> set = map.entrySet();  **for** (Entry<String, String> entry : set) {  String key = entry.getKey();  String value = entry.getValue();  System.*out*.println(key + "\t" + value);  }  }  **publicstaticvoid** main(String[] args) {  // 数组和集合的区别：  // 数组是静态的必须是定长，集合是动态的。  // 数组中既可以存储基本数据类型，也可以存储类类型。  // 集合只能存储类类型。  // Java的八种基本数据类型都有对应的类类型  // byte->Byte  // short->Short  // int->Integer  // long->Long  // float->Float  // double->Double  // boolean->Boolean  // char->Character  List<Integer> list = **new** ArrayList<Integer>();  // 自动装箱过程  **int**a = 100;  list.add(a);  list.add(200);  list.add(3);  System.*out*.println(list);  // 自动拆箱过程  **for** (**int** value : list) {  System.*out*.println(value);  }  }  集合排序  **publicint** compareTo(Object o) {  Member member = (Member) o;  // return (int) (this.salary - member.salary);  **returnthis**.name.compareTo(member.name);  } |

## (3.2)异常

|  |
| --- |
| // 异常：  // (1)异常的5个关键字  // 捕获(程序本身的问题就捕获)：try(包含异常代码)、catch(捕获异常从小到大)、finally(最终执行)  // 抛出(调用者问题就抛出)：throw(在方法内部抛出)、throws(方法声明处抛出)  // (2)异常体系结构  // Throwable类 -Error错误类(非检查型)  // (异常顶级父类) -Exception异常类(检查型) -RuntimeException运行时异常类(非检查型)  // Java中的异常分两种：检查型异常(必须处理)、非检查型异常(非必须处理)  // (3)自定义异常  // 自定义异常必须继承Exception异常类  **publicclass** A {  **publicstaticvoid** main(String[] args) {  // 认识一下常见的异常有哪些:  // (1)算数异常ArithmeticException  **try** {  System.*out*.println("111");  // 发生异常时，程序会立刻终止，然后抛出异常  String s = **null**;  System.*out*.println(s.equals(""));  System.*out*.println("222");  } **catch** (ArithmeticException e1) {  System.*out*.println("算数异常");  } **catch** (ArrayIndexOutOfBoundsException e2) {  System.*out*.println("下标越界异常");  } **catch** (RuntimeException e3) {  System.*out*.println("空指针异常");  } **finally** {  System.*out*.println("fainll语句块中的代码一定会被执行！！！");  }  // (2)数组小标越界异常ArrayIndexOutOfBoundsException  // int[] arr = { 1, 2, 3 };  // System.out.println(arr[3]);  // (3)空指针异常NullPointerException  // String str = null;  // System.out.println(str.equals("abc"));  }  }  // 自定义异常类的写法  **publicclass**MyException**extends** Exception {  **public** MyException(String str) {  **super**(str);  }  } |

## (3.3)封装类

|  |
| --- |
| **publicstaticvoid** main(String[] args) {  // 8种基本数据类型都有对应的封装类  // Byte、Short、Integer、Long  // Float、Double  // Boolean、Character  // 作用：(1)用于集合存储。(2)将String转换为基本数据类型。  // List<Integer> list = new ArrayList<Integer>();  // list.add(100);  // int value = list.get(0);  // 基本数据类型和字符串之间的相互转换:  // 基本数据类型->String通过重载方法valueOf即可  // String->基本数据类型通过基本类型对应的封装类即可  // byte b = 10;  // String s = String.valueOf(b);  // b = Byte.parseByte(s);  // int b = 100;  // String s = String.valueOf(b);  // b = Integer.parseInt(s);  //  // double d = 1.35;  // String ss = String.valueOf(d);  // d = Double.parseDouble(ss);  // 注意：java.lang.NumberFormatException数字格式化异常  // int i = Integer.parseInt("你abc");  // char[]之间String相互转换  **char**[] arr = { 'a', 'b', 'c' };  String str = String.*valueOf*(arr);  arr = str.toCharArray();  } |

## (3.4)日期

|  |
| --- |
| /\*\*  \* 必须养成查看Java API文档的习惯，从现在起，每讲一个新的API都要先看下API文档。  \*  \*  \*/  **publicstaticvoid** main(String[] args) {  // 创建一个当前日期时间的对象  // Date date = new Date();  // System.out.println(date);  // Date类型和String类型之间的相互转换都要通过SimpleDateFormat类  // 创建对象需要指定一个日期格式  // // Date->String  // SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");  // String str = sdf.format(date);  // System.out.println(str);  // String->Date  // String str = "06-16:2014年";  // SimpleDateFormat sdf = new SimpleDateFormat("MM-dd:yyyy年");  // Date date = null;  // try {  // date = sdf.parse(str);  // System.out.println(date);  // } catch (ParseException e) {  // // ParseException解析异常  // e.printStackTrace();  // }  // System.out.println(date);  // Date类不是按照年月日时分秒来存储的，存储的是距离历史一个时间点的毫秒数。  // 在实际开发中Date用于日期的计算  // Date date = new Date();  // 返回毫秒数  // long time = date.getTime();  // System.out.println(time);  // 练习题一：求100天之后是几月几号  Date date = **new** Date();  date.setTime(date.getTime() + 100 \* 24 \* 60 \* 60 \* 1000L);  SimpleDateFormat sdf = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss");  System.*out*.println(sdf.format(date));  // 练习题二：求距离2016年10月1日相差多少天  } |

## (3.5)I/O

|  |
| --- |
| // Java的I/O体系：装饰设计模式  // 字节输入流java.io.InputStream  // 字节输出流java.io.OutputStream  // 字符输入流java.io.Reader  // 字符输出流java.io.Writer  //http://blog.csdn.net/yczz/article/details/38761237 I/O流概括图  **publicstaticvoid** main(String[] args) {  // Java的I/O  FileReader fr = **null**;  BufferedReader in = **null**;  **try** {  // 创建一个文件输入流对象  fr = **new** FileReader("f:\\test.txt");  // 创建  in = **new** BufferedReader(fr);  String str = "";  **while** ((str = in.readLine()) != **null**) {  System.*out*.println(str);  }  } **catch** (FileNotFoundException e) {  e.printStackTrace();  } **catch** (IOException e) {  e.printStackTrace();  } **finally** {  // 进行Java的IO操作时要注意：  // 有打开一定有关闭，关闭顺序和打开顺序相反。  **try** {  in.close();  fr.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  }  **publicstaticvoid** main(String[] args) {  FileWriter fw = **null**;  BufferedWriter bw = **null**;  **try** {  fw = **new** FileWriter("f:\\abc.txt", **true**);  bw = **new** BufferedWriter(fw);  **for** (**int** i = 0; i < 10; i++) {  bw.write("顺口溜豆腐加两块豆腐");  bw.write("\r\n");  }  } **catch** (IOException e) {  e.printStackTrace();  } **finally** {  **try** {  bw.close();  fw.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  } |
|  |

## (3.6)设计模式

|  |
| --- |
| /\*\*  \*  \* 设计模式之：单例模式一  \*  \* 要求设计一个类A，对外只提供一个对象。  \*  \*/  **publicclass** A {  **privatestatic** A *a* = **new** A();  **private** A() {  }  **publicstatic** A m() {  **return***a*;  }  }  /\*\*  \*  \* 单例模式二  \*/  **publicclass** B {  **privatestatic** B *b*;  **private** B() {  }  **publicstatic** B m() {  **if** (*b* == **null**) {  *b* = **new** B();  }  **return***b*;  }  }  **publicclass** Client {  **publicstaticvoid** main(String[] args) {  // 使用者，调用端，客户端  // (1)类声明变量  Audi car1 = **new** Audi();  car1.driver();  // (2)接口声明变量  Car car2 = **new** Audi();  car2.driver();  // (3)工厂模式  Car car3 = CarFactory.*getCar*("1");  car3.driver();  // (4)Spring IOC  }  }  // 汽车工厂类  **publicclass** CarFactory {  **publicstatic**Car getCar(String name) {  **if** (name.equals("1")) {  **returnnew** Audi();  } **elseif** (name.equals("2")) {  **returnnew** Benz();  }  **returnnull**;  }  } |

## (3.7)eclipse调试

|  |
| --- |
| F5：跟踪到方法内部  F6：跟踪当前代码  F8：跟踪到下一个断点，或者执行完 |

## (3.8)内部类和匿名类

|  |
| --- |
| // 一个文件中可以定义多个类，但是注意文件名必须和public修饰的类名相同。  // 嵌套类概念：可以在一个类中定义另外一个类。  // (1)静态嵌套类(开发中用的非常少)  // (2)非静态嵌套类(内部类)：成员内部类、局部内部类、匿名类。  **publicclass** A {  **staticclass** B {  }  **class** C {  }  **publicvoid** m() {  **class**D {  }  }  }  **publicstaticvoid** main(String[] args) {  // 创建静态嵌套类的对象  // A.B b = new A.B();  // System.out.println(b);  // 创建内部类的对象  A a = **new** A();  A.C c = a.**new** C();  System.*out*.println(c);  }  **publicstaticvoid** main(String[] args) {  // 匿名类的使用  // 实现接口 + 创建对象  E e = **new** E() {  **publicvoid** m() {  System.*out*.println("方法的实现!");  }  };  e.m();  } |

## (3.9)dom4j解析XML

|  |
| --- |
| dom4j是java语言的文档对象模型的实现，用来解析XML文件。  XML文件的概念：   1. XML文件的首行必须声明部分，而且不能有空格。 2. XML文件只能有一个根节点。 3. XML文件只有三类节点：元素节点、属性节点、文本节点。 4. 元素节点有打开，必须有关闭。 5. 属性节点的值必须使用双引号。 |

|  |
| --- |
| **publicstaticvoid** main(String[] args) {  // 编码的理解  String str = "你好";  **try** {  // 参数指定按照什么编码返回字节数组  // UTF-8编码：一个汉字3个字节，国际化通用  // GBK编码：一个汉字2个字节，中文编码  **byte**[] arr = str.getBytes("GBK");  **for** (**byte** b : arr) {  System.*out*.println(b);  }  } **catch** (UnsupportedEncodingException e) {  e.printStackTrace();  }  } |

|  |
| --- |
| **publicstaticvoid** main(String[] args) {  // 第一部分：获取一个Document(文档对象模型)对象的方法：  // (1)从一个文件获取一个document对象  **try** {  // (1)创建一个文件对象  // 绝对路径：  // File file = new File("f:\\members.xml");  // 相对路径：相对于工程的路径!!!  File file = **new** File("members.xml");  // (2)创建一个解析器对象  SAXReader reader = **new** SAXReader();  // (3)将xml文件转化为document对象  Document document = reader.read(file);  System.*out*.println(document);  } **catch** (DocumentException e) {  e.printStackTrace();  }  // (2)通过一个xml片段转化为一个document对象  // try {  // String xmlstr = "<members></members>";  // Document document = DocumentHelper.parseText(xmlstr);  // System.out.println(document);  // } catch (DocumentException e) {  // e.printStackTrace();  // }  // (3)创建一个空的document对象  // Document document = DocumentHelper.createDocument();  // System.out.println(document);  }  **publicstaticvoid** main(String[] args) {  // Document对应XML文件  // Element对应元素节点  // Attribute对应属性节点  // 第二部分：怎样获取元素节点、属性节点、文本节点。  **try** {  File file = **new** File("company\\members.xml");  SAXReader reader = **new** SAXReader();  Document document = reader.read(file);  // (1)获取根节点  Element root = document.getRootElement();  // (2)获取多个元素节点的集合  List<Element> list = root.elements("member");  **for** (Element e : list) {  // (3)获取属性节点  Attribute att = e.attribute("name");  System.*out*.println("名字：" + att.getText());  // (4)获取单个元素节点的对象  Element ele = e.element("age");  System.*out*.println("年龄：" + ele.getText());  }  } **catch** (DocumentException e) {  e.printStackTrace();  }  }  **publicstaticvoid** main(String[] args) {  **try** {  // (1)读：将xml文件转化为内存中的document对象  File file = **new** File("company/members.xml");  SAXReader reader = **new** SAXReader();  Document document = reader.read(file);  // **TODO**: 从document中将张三的元素节点删除  // **TODO**: 从document中将李四的年龄改为50  // **TODO**: 从document中将王五的名字改为赵六  // (2)写：将内存中的document对象转换到xml文件中  OutputFormat of = OutputFormat.*createPrettyPrint*();  of.setEncoding("GBK");// 设置编码方式  XMLWriter w = **new** XMLWriter(**new** FileWriter("company/out.xml"), of);  w.write(document);  w.close();  } **catch** (DocumentException e) {  e.printStackTrace();  } **catch** (IOException e) {  e.printStackTrace();  }  } |

## (3.10)log4j在App中的使用

|  |
| --- |
| // log4j是用Java程序的日志记录技术  **publicclass** A {  **privatefinalstatic** Logger *logger* = Logger.*getLogger*(A.**class**);  **publicstaticvoid** main(String[] args) {  // log4j的5个日志等级：  // debug > info > warn > error > fatal  *logger*.debug("debug!");  *logger*.info("info!");  *logger*.warn("warn!");  *logger*.error("error!");  *logger*.fatal("fatal!");  }  }  log4j.rootLogger=warn,stdout,R  log4j.appender.stdout=org.apache.log4j.ConsoleAppender  log4j.appender.stdout.layout=org.apache.log4j.PatternLayout  # Pattern to output the caller's file name and line number.  log4j.appender.stdout.layout.ConversionPattern=%d%5p[%t](%F:%L)-%m%n  log4j.appender.R=org.apache.log4j.RollingFileAppender  log4j.appender.R.File=zhaoyang.log  log4j.appender.R.MaxFileSize=1000KB  # Keep one backup file  log4j.appender.R.MaxBackupIndex=1  log4j.appender.R.layout=org.apache.log4j.PatternLayout  log4j.appender.R.layout.ConversionPattern=%d%5p[%t](%F:%L)-%m%n |

## (3.10)线程

|  |
| --- |
| // 并发：单处理器，某一个时刻只有一个线程执行。  // 并行：多处理器，某一个时刻可以有多个线程执行。  // (多核时代：充分利用CPU、省时、响应快、用户体验好。)  // 多线程的实现方法有两种：  // (1)继承Thread类来实现多线程。  // (2)实现Runnable接口实现多线程。  **publicstaticvoid** main(String[] args) {  // 我们之前写的程序都是单线程的程序  **while** (**true**) {  // 打印出当前线程的名字  System.*out*.println(Thread.*currentThread*().getName());  }  }  // 通过继承Thread类实现多线程  **publicclass** B **extends** Thread {  // run方法是多线程的业务方法  **publicvoid** run() {  **while** (**true**) {  System.*out*.println(Thread.*currentThread*().getName());  }  }  **publicstaticvoid** main(String[] args) {  // 调用方法：  B b1 = **new** B();  b1.start();// 启动一个线程  B b2 = **new** B();  b2.start();// 启动一个线程  }  }  // 通过实现Runnable接口实现多线程  **publicclass** C **implements** Runnable {  // 多线程的业务方法  **publicvoid** run() {  **while** (**true**) {  System.*out*.println(Thread.*currentThread*().getName());  }  }  **publicstaticvoid** main(String[] args) {  // 调用方法：  C c1 = **new** C();  Thread t1 = **new** Thread(c1);  t1.start();  C c2 = **new** C();  Thread t2 = **new** Thread(c2);  t2.start();  }  }  // 银行类  **publicclass** Bank {  **privateint**count;  **public** Bank(**int** count) {  **this**.count = count;  }  **publicvoid** getMoney(**int** money) {  Date date = **new** Date();  **synchronized** (date) {  **if** (money <count) {  **try** {  Thread.*sleep*(10000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  count -= money;  System.*out*.println("余额为：" + count);  } **else** {  System.*out*.println("余额不足!!!");  }  **if** (count< 0) {  System.*out*.println("银行存款出现负数：" + count);  System.*exit*(1);  }  }  }  }  // 线程死锁的场合以及怎样解决  // (1)建议run方法中不要使用同步方法或者同步块。  // (2)多个方法中同步块的锁定顺序应该一致。  **publicclass** A **extends** Thread {  **privatestaticfinal** String *str* = "你好";  **publicvoid** run() {  **synchronized** (**this**) {  **while** (**true**) {  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.*out*.println(Thread.*currentThread*().getName());  }  }  }  **publicstaticvoid** main(String[] args) {  A a1 = **new** A();  a1.start();  A a2 = **new** A();  a2.start();  }  }  **publicclass** B **implements** Runnable {  **privatestaticfinal** String *s1* = "abc";  **privatestaticfinal** String *s2* = "123";  **publicvoid** run() {  **while** (**true**) {  m1();  m2();  }  }  **publicvoid** m1() {  **synchronized** (*s1*) {  **synchronized** (*s2*) {  System.*out*.println("执行方法一!!!");  }  }  }  **publicvoid** m2() {  **synchronized** (*s1*) {  **synchronized** (*s2*) {  System.*out*.println("执行方法二!!!");  }  }  }  **publicstaticvoid** main(String[] args) {  (**new** Thread(**new** B())).start();  (**new** Thread(**new** B())).start();  }  } |

## (3.11)反射

|  |
| --- |
| **import**java.lang.Class;  **import**java.lang.reflect.Field;  **import**java.lang.reflect.Method;  **import**java.lang.reflect.Constructor;  // 反射：一种“看透”Java的能力，JDK中的反射API都在java.lang.reflect.\*  // Java中反射机制的核心API(Java思想一切皆对象)  // (1)Class表示类  // (2)Field表示成员变量  // (3)Method表示成员方法  // (4)Constructor表示构造方法  **publicclass** Member {  **private** String name;  **privateint**age;  **public** Member() {  **this**("匿名", 0);  }  **public** Member(String name, **int** age) {  **this**.name = name;  **this**.age = age;  }  **publicvoid** m() {  System.*out*.println("方法一");  }  **public** String toString() {  **returnthis**.name + "\t" + **this**.age;  }  **publicstaticvoid** main(String[] args) {  // 创建对象  // Member m1 = new Member();  // System.out.println(m1);  // 用Java的反射机制创建对象  // try {  // // 加载类，获取一个Class类型的对象  // Class c = Class.forName("com.a.Member");  // // 调用类中无参的构造方法创建对象  // Member m2 = (Member) (c.newInstance());  // System.out.println(m2);  // } catch (ClassNotFoundException e) {  // e.printStackTrace();  // } catch (InstantiationException e) {  // e.printStackTrace();  // } catch (IllegalAccessException e) {  // e.printStackTrace();  // }  // 思考：实现此方法  Object obj = *getMember*("member");  Member m3 = (Member) obj;  System.*out*.println(m3);  }  // Spring框架IOC实现原理：  // 读取xml文件，传入id属性的值，返回class属性对应的类的对象  **privatestatic** Object getMember(String id) {  **returnnull**;  }  } |

## (3.12)Spring IoC

|  |
| --- |
| Spring是一个轻量级的JavaEE框架。  Struts是一个web框架。  Hibernate是一个持久层框架。  MyBatis是一个持久层框架。  IoC：控制反转(思想)。是整个Spring框架的灵魂核心。  有了Spring的IoC之后，IoC容器控制应用程序所需要的外部资源，然后反转给应用程序。  DI：依赖注入(思想)。 |

## (3.13)Java整体概述

|  |
| --- |
| JVM的server与client关系及切换  http://blog.csdn.net/rainnnbow/article/details/52228773 |