# (一) Java基础语法

## (1.1)环境搭建（myeclipse16及jdk的安装）

|  |
| --- |
| 步骤1. |

## (1.2) Java概述

|  |
| --- |
| /\* 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?  JavaSE Java标准版，定位在桌面应用程序，同时是其余两个平台的基础。  JavaME Java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。  JavaEE Java企业版，定位在大中型企业级的应用。  \*/  /\* 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。  JDK与Java的关系及区别?  http://www.cnblogs.com/myitm/archive/2011/05/03/2035942.html  \*/    // Java应用程序的入口(即程序开始执行的地方)  // 未注释的地方即为源代码（以下为例）  **package** com;  **public** **class** Test {  **public** **static** **void** main(String[] args) {  System.***out***.println("Hello world");  }  } |

## (1.3) Java的八种基本数据类型

|  |
| --- |
| **public static void** main(String[] args) {  /\* 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,  java 整数默认情况下使用int类型(定义的变量没有超出int的范围，编译器自动向下转化，超出,编译器直接报错，亦可以进行强制转化)  \*/  (1)byte字节类型，占8位，取值范围-128~127（2的7次方，第一位是符号位）  **byte** b = 1;  System.*out*.println("byte字节类型的值是：" + b);  // (2)short短整型，占16位，取值范围-32768~32767（2的15次方）  **short** s = 10;  System.*out*.println("short短整型的值是：" + s);  // (3)int整型，占32位（2的31次方）  **int** i = 100;  System.*out*.println("int整型的值是：" + i);  // (4)long长整型，占64位（2的63次方）  **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,  Java浮点数默认使用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或者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);  } |

## (1.4)变量与常量

|  |
| --- |
| **public static void** 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。  **int** a = 2;  **int** b = 3;  //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);  // 常量使用关键字final定义，表示恒定的量，不能被修改。  /\* 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。若final int a = 2;a=9;则编译器报错  \*/  } |

## (1.5)基本数据类型之间的转换

|  |
| --- |
| **public static void** 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);  } |

## (1.6) 运算符

### (1.6.1)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); |

### (1.6.2)算数运算符

|  |
| --- |
| //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); //取余数 |

### (1.6.3)比较运算符

|  |
| --- |
| // 比较运算符，结果一定是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); |

### (1.6.4)逻辑运算符

|  |
| --- |
| // 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**); |

### (1.6.5)自增自减运算符

|  |
| --- |
| // 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++)); |

### (1.6.6)三目运算符

|  |
| --- |
| // 三目运算符?:如果a>b，则将a赋值给max否则将b赋值给max  **int** a = 2;  **int** b = 3;  **int** max = (a > b) ? a : b;  System.*out*.println(max);  // 例子：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); |

## (1.7)流程控制语句

### (1.7.1)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");  }  // 例子：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");  }  // 例子：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);  } |

### (1.7.2)switch分支语句

|  |
| --- |
| /\* jdk1.7之前switch分支语句中的类型必须是：byte、short、int、char及  其对应的封装类或者枚举Enum类型,枚举在讲面相对象的时候讲到\*/  // 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!");  }  }  // jdk1.7之后也支持String类型（String不是基本数据类型，是类类型）  String i = "优秀";  **switch** (i) {  **case** "优秀": {  System.***out***.println("优");  **break**;  }  **case** "良好": {  System.***out***.println("良");  **break**;  }  **default**: {  System.***out***.println("不及格!");  }  } |

### (1.7.3)for循环语句（简单循环在讲解对象的时候再提）

|  |
| --- |
| /\*for(<初始化>; <[条件表达式](https://www.baidu.com/s?wd=%E6%9D%A1%E4%BB%B6%E8%A1%A8%E8%BE%BE%E5%BC%8F&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d9PjwBujKhuHbsm16zPhcz0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnHfLPH6dnjRY)>; <增量>){};执行顺序：  初始化->条件表达式成立->语句块->增量->条件表达式成立->语句块…  若其中条件表达式不成立，则跳出当前循环  \*/  // 例子：求1 + 3 + 5的和  **int** s = 0;  **for** (**int** i = 1; i <= 5; i++) {  **if** (i == 2 || i == 4) {  **continue**;  }  s = s + i;  }  System.***out***.println(s);  /\*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(;;){}代表无限循环，for(int i=0;;i+=2)也是无限循环，  初始化只进行一次。i+=2表示每次的步数是增加2。  \*/  //无限循环的例子  **int** i = 0;  **for**(;;){  **if**(i>4){  **break**;  }**else**{  i++;  }  }  System.***out***.println(i); |

### (1.7.4)while与do-while循环语句

|  |
| --- |
| //对于第一次循环,while是先判断再执行,do-while是先执行,再判断。  //如下的while不会打印出任何信息  **int** i = 10;  **while** (i < 10) {  System.*out*.println(i);  i++;  }  //如下的do-while会打印结果  **int** i=10;  **do**{  System.***out***.println(i);  i++;  }**while**(i<10);  //打印结果的是10,i=11 |

## (1.8)数组

|  |
| --- |
| // Java中的类型有两种：一是基本数据类型。二是类类型。  // 一维数组  // (1)创建一维数组几种定义格式  **int**[] arr1=**new** **int**[10];//指定数组大小，大小0-9  **int**[] arr2={1,2,3,6}; //指定数组内容  **int**[] arr3=**new** **int**[]{1,2,3};//此种效果与上一个相同，会创建两个对象，不能指定int[]大小  // (2)通过下标存取  **int**[] arr = **new** **int**[3];  arr[0] = 1;  arr[1] = 2;  **int** x = arr[0];  System.***out***.println(x);  // (3) 遍历数组,length是数组的属性（此处暂且记住，讲到类的时候会提到）  **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) 创建二维数组几种定义格式  **int**[][] arr = { { 1, 2 }, { 3, 4 }, { 5 } };//定义的同时赋初始值  **int**[][] arr1 = **new** **int**[2][];//不规则数组  **int**[][] arr2 = **new** **int**[2][3];//定义的同时分配空间大小  // (2)下标存取  **int**[][] arr1 = **new** **int**[2][];  arr1[0]=**new** **int**[5];// 为第1行分配5列，也可以认为给第一个一维数组装进一个数组  arr1[0][4]=9;//一行五列值是9，也可认为是一维数组的第一个元素内的数组的第5个位置  System.***out***.println(arr1[0][4]);  (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();  } |

## (1.9)函数

|  |
| --- |
| // 等到下一章节讲到类的时候，就会涉及到函数，在面向过程开发被称为函数，在面向对象中被称为方法  // 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;}  // 如下例子:  /\*第一次被调用的时候m的值是2，然后返回到调用此函数的地方，即m1(a,b),  \* 第二次被调用m的值是2，然后返回到调用此函数的地方，即m1(m1(a, b), c)的值是2  \*/  **public** **static** **int** m1(**int** a, **int** b) {    **int** m = (a > b) ? b : a;  **return** m;  }  /\*此时a,b,c分别为2,3,5，程序会先调用内部的m1(a,b)，然后得到返回值再调用m1(m1(a,b)的返回值,c),经过两次m1函数的返回值，int m的值是2，然后将m的值返回到调用m2(int a, int b, int c)的地方，即int max的值是2，然后打印出max的值  \*/  **public** **static** **int** m2(**int** a, **int** b, **int** c) {    **int** m = *m1*(*m1*(a, b), c);  **return** m;  }  //此为入口程序  **public** **static** **void** main(String[] arr) {  /\*会调用方法名为m2，参数列表即m2括号中的2,3,5，类型分别为int,int,int,程序会自动寻找符合这样条件的函数，执行完m2函数后的返回值赋给max\*/  **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

|  |
| --- |
| /\*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。\*/  **package** com;  **public** **class** Test {  **private** String name = "张三";  **private** **int** age = 30;  **public** Test() {//构造方法，方法体是空的，则只是创建对象  }  **public** Test(**int** a) {//有参数的构造方法，也只是创建对象  }  // 对象：实例，实体。  //static后面会专门讲到  **public** **static** **void** run() {  System.***out***.println("跑步");  }  **public** **static** **void** eat() {  System.***out***.println("吃饭");  }  **public** **static** **void** 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();  Test test = **new** Test(4);  System.***out***.println(test.name);  System.***out***.println(test.age);  test.*eat*();  test.*run*();  }  } |

## (2.2)引用与实例

|  |
| --- |
| /\*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地址，即引用。  Test b = null;声明了一个对象（B的引用），但jvm并没有开辟内存放入一个对象,  所以b内没有引用，暂时理解null就是什么也没有  b = new Test();// 此对象会被java的垃圾回收机制收回GC  b = new Test();  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。\*/  **public** **class** Test {    **public** **static** **void** main(String args){  //最后b1,b2,b3都会存有同一个指向new Test()这个对象的引用  Test b1 = **new** Test();  Test b2 = b1;  Test b3 = b1;  }  } |

## (2.3)值传递和引用传递

|  |
| --- |
| /\*值传递：  方法调用时，实际参数把它的值传递给对应的形式参数，方法接收的是原始值的一个copy，此时内存中存在两个相等的基本类型，即实际参数和形式参数，后面方法中的操作都是对形参这个值的修改，不影响实际参数的值。  引用传递：  也称为传地址。方法调用时，实际参数的引用(地址，而不是参数的值)被传递给方法中相对应的形式参数，方法接收的是原始值的内存地址；  在方法执行中，形参和实参内容相同，指向同一块内存地址，方法执行中对引用的操作将会影响到实际对象。\*  总结：  值传递是8种基本类型值的传递，除了基本类型其他都是类类型，是引用传递，传递的是地址  注意：这里要特殊考虑String，以及Integer、Double等几个基本类型包装类，它们都是immutable类型（即不可变类型），可以认为是和基本数据类型相似，传值操作。  \*/  //值传递，并没有改变原始值  **public** **class** Test {  **int** m;  **public** **static** **void** main(String[] args){  **int** k = 9;  *m1*(9);  System.***out***.println(k);  }  **public** **static** **void** m1(**int** m ) {  m = 4;  }  }  //引用传递，改变引用的实际对象的值  **public** **class** Test {  **public** **static** **void** main(String[] args){  **int**[] arr = { 1, 2, 3 };  *m1*(arr);  System.***out***.println(arr[0]);  }  **public** **static** **void** m1(**int**[] m ) {  m[0] = 4;  }  } |

## (2.4)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。\*/  **public** **class** Test {    **int** age; //成员变量，类初始化时，若不赋值，会默认赋值  **public** **static** **void** main(String[] args) {    **int** k = 0; //局部变量，若此处不赋值，则调用k的地方会报错，编译不会通过  System.***out***.println(k);    }  }  /\*8种基本类型默认赋值： byte、short、int、long默认都是0，float、double是0.0  Boolean：false,char:空字符  类类型默认赋值：null  \*/ |

## (2.5)关键字

|  |
| --- |
| /\*关键字：被Java语言赋予了特殊含义，用做专门用途的字符串（单词）,java的关键字都是小写的，java共有51个关键字，2个保留字（现有版本没用，以后的版本可能会用）。定义变量时，避免与这53个单词发生冲突。  51个关键字分别为：  ①访问修饰符的关键字(共3个)：  public、private、protected  ②定义类、接口、抽象类和实现接口、继承类的关键字、实例化对象（共6个）：  class、interface、abstract、implements、extends、new  ③包的关键字（共2个）：  import、package  ④数据类型的关键字（共12个）：  byte、char、boolean、short、int 、float、long、double、void、null、true、false  ⑤条件循环（流程控制）（共12个）：  if、else、while、for、switch、case、default、do、break、continue、return、instanceof  ⑥修饰方法、类、属性和变量（共9个）：  static、final、super、this、native、strictfp、synchronized、transient、volatile  ⑦错误的处理（共5个）：  catch、try、finally、throw、throws  ⑧枚举、断言（共2个）：  enum、assert  2个保留字为：  const、goto  ------------------------------------------------------------------  这些关键字有些已经学过，以后一点一点都会接触到。  ------------------------------------------------------------------ |

### (2.5.1)关键字static

|  |
| --- |
| static 关键字：  ①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。  ②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。  ③静态成员被所有对象共享。\*/  **public** **class** Test {  **static** **int** *a* = 20;    **int** b = 30;    **static** **void** m1() {  System.***out***.println("访问静态方法m1");  }    **void** m2() {  System.***out***.println("访问非静态方法m2");  }    **public** **static** **void** main(String[] args) {    //static int m = 6;//这样定义是错误的，static只能修饰类的成员变量或方法    System.***out***.println(Test.*a*);//类直接访问自身的静态属性  Test.*m1*();//类直接调用自身的静态方法    Test test = **new** Test();  System.***out***.println(test.b);//对象访问非静态成员变量  System.***out***.println(test.*a*);//对象访问静态成员变量也可以，但是有黄线警告，//建议使用类直接访问  test.m2();//对象访问非静态方法    System.***out***.println(test.*a*);  test.*m1*();//对象访问静态方法，有黄线警告，但并不影响程序运行，建议最好使用//Test调用静态方法  //Test.m2();//此处编译报错，因为m2没有被static修饰  //m2();//此处编译报错，静态方法不能访问非静态方法  //b=50;//此处编译报错，静态方法不能访问非静态成员变量  }  }  /\*①类的初始化：创建对象时调用（目前可认为是new的时候），可以用于完成初始化属性值（在构造方法中初始化）、加载其他的类的功能。  ②类的初始化顺序：加载静态-》加载非静态-》调用构造方法（目前可以这么理解，以后会提到父类的加载）  ③静态不能包含非静态的内容：静态早于非静态初始化，所以无法访问。\*/  **public** **class** Test {    // 静态变量  **public** **static** String *staticField* = "静态变量";  //暂时不用理解这个字符串数组，只为证明static是初始化一次  **public** **static** String[] *strArr* = {"1","2"};  // 变量  **public** String field = "变量";  // 静态初始化块：一般在里面放一些需要提前加载又不需要多次加载的东西（目前不需要  // 太明白，一般用于servlet或者框架的启动加载）  **static** {  System.***out***.println(*staticField*);  System.***out***.println("静态初始化块");  m1();  //看控制台结果，静态只初始化一次，非静态创建几个对象就初始化几次  main(*strArr*);  }  // 初始化块：创建类的时候会调用.  //作用:①类似于构造方法的初始化工作。  // ②用于匿名类的初始化，匿名类无构造方法。  {  System.***out***.println(field);  System.***out***.println("初始化块");  m2();  }  // 构造器  **public** Test() {  System.***out***.println("构造器");  }  **public** **static** **void** m1(){  System.***out***.println("静态方法m1");  }  **public** **void** m2(){  System.***out***.println("非静态方法m2");  }  **public** **static** **void** main(String[] args) {  **new** Test();  }  } |

### (2.5.2)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。  (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。  (4)final不能用于修饰构造方法  (5) static final一起使用修饰成员变量或方法时，可以理解为“全局常量”，均可通过类名直接访问  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(否则IDE报错)。\*/  **public** **class** Test {  **final** **static** **int** ***b*** = 3;  **public** **static** **void** main(String[] args) {  //final int a = 2;//局与变量也可以定义final但是并没有什么意义  System.***out***.println(***b***);  }  } |

### (2.5.3)关键字this

|  |
| --- |
| /\*this关键字表示当前类的对象  this的作用：①用于区分成员变量和局部变量。  ②用于构造方法之间的调用，入口一致\*/  **public** **class** Test {  **private** String name;  **private** **int** age;  //使用new Test()会调用这个无参数的构造方法  **public** Test() {  **this**("");//会调用当前对象参数是(String)的构造方法  }  //使用new Test("参数值")会调用这个只有一个String参数的构造方法  **public** Test(String name) {  **this**(name, 0);//会调用当前对象参数是(String，String)的构造方法  System.***out***.println(name);//打印看看是如何调用  }  //使用new Test("参数值","参数值")会调用这个有两个String参数的构造方法  **public** Test(String name, **int** age) {  **this**.name = name;//this代表当前类的对象  **this**.age = age;  System.***out***.println(name+"+++"+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** **this**.name + "\t" + **this**.age;  }  **public** **static** **void** main(String[] args) {  Test p1 = **new** Test();  p1.setName("张三");  p1.setAge(20);  System.***out***.println(p1);    Test p2 = **new** Test("李四");  p2.setAge(30);  System.***out***.println(p2);    Test p3 = **new** Test("王五", 40);  System.***out***.println(p3);  }  } |

### (2.5.4)关键字null

|  |
| --- |
| /\*①null是关键字，像public、static、final。它是大小写敏感的，你不能将null写成Null或NULL，编译器将不能识别它们然后报错。  ②null既不是对象也不是一种类型，它仅是一种特殊的值，你可以将其赋予任何引用类型，你也可以将null转化成任何类型  ③null可以赋值给引用变量，你不能将null赋给基本类型变量  ④你可以使用==或者!=操作来比较null值，但是不能使用其他算法或者逻辑操作，例如小于或者大于。在Java中null==null将返回true\*/  String str = **null**;  Integer i = **null**;//8种基本类型的封装类可以  //int i = null;//不能将null赋值给8种基本类型  Double d = **null**;  String myStr = (String) **null**;  Integer myI = (Integer) **null**;  Double myD = (Double) **null**;  System.***out***.println(myStr); |

### (2.5.5)关键字super(后面讲到继承的时候会讲到)

|  |
| --- |
| /\*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。\*/ |

## (2.6)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:①Java将相关的变量和方法封装到类中。  ②Java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\*/ |

## (2.7)String

|  |
| --- |
| /\*String是Java中最常用的引用类型，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。\*/  **public** **class** Test {    **public** **static** **void** main(String[] args){  // 字符串的创建  // 第一种方式采用的是"字面值"，每次使用Java会去字符串常量池检查是否有此字符//串，如果有直接引用，没有则创建并放入字符串常量池中。  String s1 = "你好";  /\*每次都会创建新的对象，建议使用上面的方式创建，此种方式创建了两个对象,new与“你好”,若果常量池中有“你好”，则创建一个对象\*/  String s2 = **new** String("你好");  //编译器会自动编译成你好hello,还是两个对象  String s3 = **new** String("你好"+"hello");  // 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 strbuff = **new** StringBuffer();  strbuff.append("a");  strbuff.append("b");  strbuff.append("c");  System.***out***.println(strbuff);  // 字符串的本质是字符数组  String strArr = "123456789";  **for** (**int** i = 0; i < strArr.length(); i++) {  System.***out***.println(strArr.charAt(i));  }  // 要求至少记住String中的11个方法  String str = "ab,c ";  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,不包含截止点，即在此不包含2  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  str.split(",");//以,号开始分割字符串(也可以以别的做分割标识)  }  } |

### (2.7.1)equals与==的区别

|  |
| --- |
| /\*都是判断相等。  Equals：用于比较两个独立对象的内容是否相同。  ==：①用于基本数据类型的比较。②判断引用是否指向堆内存的同一块地址。  \*/  **public** **static** **void** main(String[] args){  // String str1 = new String("str");  // String str2 = new String("str");  // //false,str1与str2引用的对象的地址不一样，故false  // System.out.println(str1==str2);  // //true,值都是str，所以相等  // System.out.println(str1.equals(str2));    String s1 = **new** String("str");  String s2 = s1;  //true,s1与s2的地址相同  System.***out***.println(s1==s2);  //true,值相同  System.***out***.println(s1.equals(s2));  }  //例子（理解常量池）  **public** **static** **void** main(String[] args){  String s1="Hello";    String s2="Hello";    String s3=**new** String("Hello");    System.***out***.println("s1和s2 引用地址是否相同："+(s1 == s2));  System.***out***.println("s1和s2 值是否相同："+s1.equals(s2));    System.***out***.println("s1和s3 引用地址是否相同："+(s1 == s3));  System.***out***.println("s1和s3 值是否相同："+s1.equals(s3));  } |

### (2.7.2)String与StringBuilder,StringBuffer的区别

|  |
| --- |
| /\*①String:由于String是被final修饰的，所以它是字符串常量。以下代码打印出的结果是abc1,为什么改变值了？实际上s的值改变的时候s的引用是又创建了一个对象s+1,原来的abc仍然在，只不过现在又增加了一个新的对象abc1\*/  String s = "abc";  s = s+1;  System.***out***.print(s);  /\*②连接频繁的时候使用StringBuffer或者StringBuilder\*/  StringBuffer strbuff = **new** StringBuffer();  strbuff.append("a");  strbuff.append("b");  strbuff.append("c");  System.***out***.println(strbuff);  StringBuilder strBuilder = **new** StringBuilder();  strBuilder.append("a");  strBuilder.append("b");  strBuilder.append("c");  System.***out***.println(strBuilder);  /\*③总结：String：字符串常量，StringBuffer：字符创变量;线程非安全的，StringBuilder：\* 字符创变量;线程安全的，  \* 如果要操作少量的数据用 = String,  \* 单线程操作字符串缓冲区下操作大量数据 = StringBuilder,  \* 多线程操作字符串缓冲区下操作大量数据 = StringBuffer  \* \*/ |

## (2.8)继承

|  |
| --- |
| /\*(1) java8以前：当定义一个类时，该类会默认继承Object（顶级父类），Java中的单根继承（java8是2014年发布的，至今使用不广泛，2017年oracle发布java9,目前学习java7即可）, 父类有的子类都有(属性和方法，为了复用)，如果不指定父类，则新定义的类会默认继承Object \*/  **public** **class** Test **extends** Father{    **private** **int** b=4;    **public** Test(**int** a) {  **super**(a);//调用父类的构造方法，必须在第一行，否则编译出错  **this**.b=9;  }    **public** Test() {  /\*调用父类的无参数构造方法，若父类中只定义了有参数的（没有定义无参数的构造方法），此处会编译错误\*/  **this**.b=5;  }    **public** **static** **void** main(String[] args){    System.***out***.println((**new** Test(10)).getA());    }  }  /\*这是父类\*/  **public** **class** Father {    **private** **int** a;    **protected** Father(**int** a){  **this**.a=a;  }    **protected** Father(){    }  **public** **int** getA() {  **return** a;  }  **public** **void** setA(**int** a) {  **this**.a = a;  }    } |

## (2.9)多态

|  |
| --- |
| /\*多态：  (1)类内部多态(重载, **Overload**):  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, **Override**):  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  \*/  **public** **class** Test **extends** Father{  **public** **int** a = 100;    **public** Test(**int** a) {  **super**(a);//调用父类的构造方法  }    **public** Test() {    }    **public** **static** **void** main(String[] args){    //System.out.println((new Test(10)).getA());  /\*父类的引用指向子类对象，向上转型（向下转型与向上转型在接下来的在下一节会讲）\*/  Father father = **new** Test();  // 通过方法访问同名变量：此时会调用方法所在类的变量的值。  System.***out***.println((**new** Father()).getA());  // 此时会发生覆写，调用的是子类的方法  //System.out.println(father.getA());    }  **public** **int** getA() {  **return** a;  }  **public** **void** setA(**int** a) {  **this**.a = a;  }    }  /\*这是父类\*/  **public** **class** Father {    **private** **int** a=9;  //这里的有参数方法与下面的没参数方法是重载方法（参数列表不同）。  **protected** Father(**int** a){  **this**.a=a;  }    **protected** Father(){    }  **public** **int** getA() {  **return** a;  }  **public** **void** setA(**int** a) {  **this**.a = a;  }    } |

### (2.9.1)向上转型与向下转型

|  |
| --- |
| /\*转型：  转型是在继承的基础上而言的，继承是面向对象语言中，代码复用的一种机制，通过继承，子类可以复用父类的功能，如果父类不能满足当前子类的需求，则子类可以重写父类中的方法来加以扩展。  ①向上转型：子类引用的对象转换为父类类型称为向上转型。通俗地说就是是将子类对象转为父类对象。此处父类对象可以是接口（接口在接下来会讲）  前者是一个向上转型，Father father 引用指向new Test();子类对象当成父类对象，只能调用父类的成员，  如果子类重写了父类的方法就根据这个引用指向调用子类重写的这个方法（这个方法就是覆盖override）。这个调用过程就称为“动态绑定”。  父类指向子类引用对象会遗失除与父类对象共有的其他方法，也就是在转型过程中，子类独有的方法都会遗失掉，在编译时，系统会提供找不到方法的错误。  ②向下转型：父类引用的对象转换为子类类型称为向下转型(强制转型)。  ③向上转型的作用：减少重复代码，方法中父类做为参数，调用此方法时可以传入多个不同子类的对象。这样使代码变得简洁。体现了JAVA的抽象编程思想  \*/  **public** **class** Test **extends** Father{  **public** **int** a = 100;    **public** Test(**int** a) {  **super**(a);//调用父类的构造方法  }    **public** Test() {    }    **public** **static** **void** main(String[] args){    //System.out.println((new Test(10)).getA());  //父类的引用指向子类对象（向上转型）  Father father = **new** Test();  // 通过方法访问同名变量：此时会调用方法所在类的变量的值。  System.***out***.println((**new** Father()).getA());  //向下转型(强制类型转换)  Test test = (Test)father;  test.getA();  // 此时会发生覆写，调用的是子类的方法  //System.out.println(father.getA());    }  **public** **int** getA() {  **return** a;  }  **public** **void** setA(**int** a) {  **this**.a = a;  }    }  /\*向上转型好处的例子\*/  //父类  **public** **class** Father {    **private** **int** a=9;    **protected** Father(**int** a){  **this**.a=a;  }    **protected** Father(){    }  **public** **int** getA() {  **return** a;  }  **public** **void** setA(**int** a) {  **this**.a = a;  }    }  //子类A  **public** **class** ChildA **extends** Father{    **public** **int** getA() {  **return** 111;  }  }  //子类B  **public** **class** ChildB **extends** Father {  **public** **int** getA() {  **return** 222;  }  }  //测试类C  **public** **class** C {    **public** **static** **void** main(String[] args){  //可以 new不同的子类A或B,这样transferClass方法只需写一个  System.***out***.println((**new** C().transferClass(**new** ChildB())));  }  **public** **int** transferClass(Father father){    **return** father.getA();  }  } |

## (2.10)抽象类

|  |
| --- |
| /\* (1)抽象类：被abstract关键字修饰的类叫做抽象类。  \* (2)抽象类中的成员变量和类中的成员变量相同。  \* (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。  \* (4)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  \* (5)构造方法：抽象类中有构造方法,但是不能创建对象,所以构造方法的作用就是  \* 初始化（一般初始化子类的属性，子类大部分的属性都会封装在抽象类中）.  \* (6)抽象类存在的意义：  \* 设计规范（子类必须实现抽象方法）与继承（实现方法让子类继承）  \*/  //抽象类  **public** **abstract** **class** Father {    **private** **int** temp;  //不能被final修饰，被final修饰的类不能有子类  //abstract final void study();    **public** **abstract** **void** study();    **public** **void** sleep()  {  System.***out***.println("睡觉");  }    **public** Father(){  **this**.temp=100;  }    **public** **int** getTemp() {  **return** temp;  }  **public** **void** setTemp(**int** temp) {  **this**.temp = temp;  }    }  //子类A  **public** **class** ChildA **extends** Father{    //子类A,必须实现抽象类中的抽象方法，并且继承了抽象类中的属性与实现方法  @Override  **public** **void** study() {    System.***out***.println("我是子类A的实现方法");  }  }  //子类B  **public** **class** ChildB **extends** Father {  @Override  **public** **void** study() {  System.***out***.println("我是子类B的实现方法！");  }  }  //测试类C  **public** **class** C {    **public** **static** **void** main(String[] args){  //向上转型  (**new** C()).transferClass(**new** ChildB());    }  /\*\*  \* 向上转型  \* **@param** father  \*/  **public** **void** transferClass(Father father){  //调用了被子类覆写的方法  father.study();  //继承了父类的sleep方法  father.sleep();  //被抽象类初始化的属性值  System.***out***.println(father.getTemp());  }  } |

## (2.11)接口

|  |
| --- |
| /\* 接口概念：使用interface关键字定义的叫做接口。  \* 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,更不能初始化。  \* 接口目的：设计规范、更纯粹的设计规范。  \* 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。  \*/  **public** **interface** A {  // 接口中的成员变量：接口中的变量默认被public static final修饰。  **int** ***age*** = 20;  //与下面的效果一致  //public static final int 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。  **void** m();  //与下面的效果一致  //public abstract void m();  }  //抽象类与接口结合起来=====================start======================  **public interface** A {  **public abstract void** m1();  **public abstract void** m2();  }  **public abstract class** B **implements** A {  @Override  **Public void** m1() {  System.*out*.println("实现方法一");  }  //抽象类中可以不用实现抽象方法  @Override  **public abstract void** m2();  }  **public class** C **extends** B {    //必须实现抽象方法  @Override  **Public void** m2() {  System.*out*.println("实现方法二");  }  }  //抽象类与接口结合起来=====================end======================  // 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。========start=======  **public class** M **extends** N **implements** O, P, Q {  @Override  **public void** ma() {  System.*out*.println("实现接口中的方法");  }  @Override  **public void** mb() {  System.*out*.println("实现抽象类B中的方法");  }  }  // 开发中常见的使用方式：设计一个类继承一个父类同时实现多个接口。========end======= |

# (三)JDK API

## (3.1)集合

|  |
| --- |
| /\* 集合：  \* Java中的集合构成：7个接口和一个工具类组成，都在java.util包下。  \* 在集合框架的类继承体系中，最顶层有两个接口：  \* Collection表示一组纯数据，Map表示一组key-value键值对  \* Collection:定义了一个集合所具备的通用抽象方法。  \* List:标准是有序可重复。ArrayList实现类。  \* Set:标准是不重复无序。HashSet实现类。  \* SortedSet:标准是不重复，有序。TreeSet实现类。  \* Queue:作为了解即可，先进先出的一种数据结构。  \* Stack:继承了Vector类，后进后出的一种数据结构，也作为了解。  \* Map：无序,key不重复。 HashMap实现类。  \* SortedMap:有序（用二叉树排序），key不重复。TreeMap实现类。  \* Collections：它包含有各种有关集合操作的静态多态方法。此类不能实例化，就像一个工具类。服务于Java的Collection框架。  \* 数组和集合的区别：  \* ①数组是静态的必须是定长（这是缺点），集合是动态的。  \* ②数组中既可以存储基本数据类型，也可以存储类类型。  \* ③集合只能存储类类型。  \*/  /\* 集合结构图\*/    ==============================List start============================  **public class** A {  **public static void** 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,删除序号或者对象，如果是数字，则默认为序号（list<Integer>）  // list.remove(0);  // 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);  }  }  }  ============================== List end============================  ============================== Set start============================  **public static void** 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);  }  }  ============================== Set end============================  ============================== SortedSet start============================  **public static void** 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);  }  }  ============================== SortedSet end ============================  ============================== Map、SortedSet start ====================  **public static void** main(String[] args) {  // Map与SortedMap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);  }  }  ============================== Map、SortedSet start =======================  ============================== Queue start（了解即可） =======================  **public** **static** **void** main(String[] args){  //先进先出  Queue<String> queue = **new** LinkedList<String>();  queue.offer("a");  queue.offer("b");  queue.offer("c");  queue.offer("d");  queue.offer("e");  queue.offer("a");  **for**(String q : queue){  System.***out***.println(q);  }  System.***out***.println("===");  //返回第一个元素，并在队列中删除  System.***out***.println("poll="+queue.poll());  **for**(String q : queue){  System.***out***.println(q);  }  System.***out***.println("===");  System.***out***.println("element="+queue.element()); //返回第一个元素  **for**(String q : queue){  System.***out***.println(q);  }  System.***out***.println("===");  System.***out***.println("peek="+queue.peek()); //返回第一个元素  **for**(String q : queue){  System.***out***.println(q);  }  }  ============================== Queue end =========================  Queue队列与Stack栈了解即可，Queue的数据结构是先进先出，Stack的数据结构是后进先出。  ============================== Collections start =========================  **public** **static** **void** main(String[] args){    List list = **new** ArrayList();  **double** array[] = { 112, 111, 23, 456, 231 };  **for** (**int** i = 0; i < array.length; i++) {  list.add(**new** Double(array[i]));  }  //对集合进行排序,还有其他许多功能  Collections.*sort*(list);  **for** (**int** i = 0; i < array.length; i++) {  System.***out***.println(list.get(i));  }  }  ============================== Collections end ========================= |

### (3.1.1)自动拆箱与装箱

|  |
| --- |
| /\*基本数据类型不是对象，也就是使用int、double、boolean等定义的变量、常量。  \* 当程序需要使用基本数据类型的对象时，就会自动装箱，将其变为对象  \* 当使用对象是个数据类型时，会自动将其拆箱，变为基本数据类型  \* 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  \*/  **public static void** main(String[] args) {    //list中存储的全是Integer类型的对象  List<Integer> list = **new** ArrayList<Integer>();  // 自动装箱过程  **int**a = 100;  // list添加内容是对象，所以将a给自动装箱变成对象  list.add(a);  list.add(200);  list.add(3);  System.*out*.println(list);  // 先将value自动装箱成对象进行for循环，然后再拆箱成基本数据类型打印  **for** (**int** value : list) {  System.*out*.println(value);  }  } |

### (3.1.2)String与基本数据类型的转换

|  |
| --- |
| /\*(1)8种基本数据类型都有对应的封装类有什么作用？  \* ①用于集合存储。②将String转换为基本数据类型（char除外）。  \* (2)基本数据类型和字符串之间的相互转换:  \* ①基本数据类型->String通过重载方法valueOf即可  \* ②String->基本数据类型通过基本类型对应的封装类即可（char除外）  \*/  **public** **static** **void** main(String[] args){  // byte b = 10;  // String s = String.valueOf(b);  // b = Byte.parseByte(s);    //short sho = 10;  //String strsho = String.valueOf(sho);  //sho = Short.parseShort(strsho);  // int b = 100;  // String s = String.valueOf(b);  // b = Integer.parseInt(s);    //long lon = 103339999999L;  //String strlon = String.valueOf(lon);  //lon = Long.parseLong(strlon);    //float flo = 10.3f;  //String strflo = String.valueOf(flo);  //flo = Float.parseFloat(strflo);    // 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");    //只有参数是true，m的值才是true,其他的都是false  //Boolean m = new Boolean("ddd");  //String strm = String.valueOf(m);  //m = Boolean.parseBoolean(strm)    //字符与字符串之间没必要相互转换,字符串本质上是字符数组  // char[]之间String相互转换,字符串转换为字符数组时，toCharArray  **char**[] arr = { 'a', 'b', 'c' };  String str = String.*valueOf*(arr);  arr = str.toCharArray();    } |

## (3.2)异常

|  |
| --- |
| /\*(1)异常体系结构图\*/    /\*(1.1)异常体系概括： Thorwable类（表示可抛出）是所有异常和错误的超类 , 两个直接子类为Error和Exception，分别表示错误和异常。其中异常类Exception又分为运行时异常(RuntimeException)和非运行时异常，  这两种异常有很大的区别，也称之为不检查异常（Unchecked Exception）和检查异常（Checked Exception）。  (1.2) 异常体系详细分析：  ①Error与Exception：  Error是程序无法处理的错误，它是由JVM产生和抛出的，比如OutOfMemoryError、ThreadDeath等。这些异常发生时，Java虚拟机（JVM）一般会选择线程终止。  Exception是程序本身可以处理的异常，这种异常分两大类运行时异常和非运行时异常。程序中应当尽可能去处理这些异常。  ②运行时异常和非运行时异常：  运行时异常：  RuntimeException类及其子类异常，如NullPointerException、  IndexOutOfBoundsException等，这些异常是不检查异常，程序中可以选择捕获处理，也可以不处理。这些异常一般是由程序逻辑错误引起的，程序应该从逻辑角度尽可能避免这类异常的发生。  非运行时异常：  RuntimeException以外的异常，类型上都属于Exception类及其子类。从程序语法角度讲是必须进行处理的异常，如果不处理，程序就不能编译通过。如IOException、SQLException等以及用户自定义的Exception异常。  (1.3) 异常处理：  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自定义异常：  自定义异常必须继承Exception异常类。  \*/  **public** **static** **void** main(String[] args){    // 认识一下常见的异常有哪些:  //try语句块中的代码，如果产生异常，跳转到相应catch,异常代码后面代码不执行  **try** {  //String s = null;  //System.out.println(s.equals(""));  //System.out.println("222");    //int[] arr = { 1, 2, 3 };  //System.out.println(arr[3]);    System.***out***.println(9/0);  //catch语句块中的代码，是产生异常后需要做出相应处理的代码  } **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");  //无论是否发生异常，都会执行的代码  } **finally** {  System.***out***.println("finally语句块中的代码一定会被执行！！！");  }  //如果catch中没有继续throw异常以及没有产生新异常，则继续执行的代码  System.***out***.println("继续执行！！！");    }  /\*自定义异常的几种写法\*/  //①（抛出单个异常情况）============  //--------------------------自定义异常类 start-------------------------  **public** **class** WrongInputException **extends** Exception {  WrongInputException(String s) {  //调用父类构造，即Exception的构造方法  **super**(s);  }  }  //--------------------------自定义异常类 end--------------------------  //--------------------------抛出异常类 start--------------------------  **public** **class** Input {  **void** method() **throws** WrongInputException {  **throw** **new** WrongInputException("Wrong input"); // 抛出自定义的类  }  }  //--------------------------抛出异常类 end-----------------------------  //--------------------------测试类 start-----------------------------  **public** **class** TestInput {  **public** **static** **void** main(String[] args){    **try** {  **new** Input().method();  }  //捕获异常类，打印异常类信息  **catch**(WrongInputException wie) {  System.***out***.println(wie.getMessage());  }  }  }  //--------------------------测试类 end-------------------------------  //②（抛出多个异常情况）============  //--------------------------自定义异常类1 start-------------------------  **public** **class** MyException1 **extends** Exception {  **int** num;  MyException1(**int** a) {    num = a;  }  **public** String toString(){    **return** num+"<10! 值必须大于10";  }  }  //--------------------------自定义异常类1 end---------------------------  //--------------------------自定义异常类2 start-------------------------  **public** **class** MyException2 **extends** Exception{  **int** num;  MyException2(**int** a) {    num = a;  }  **public** String toString(){    **return** num+">100!/r/n值必须小于100";  }  }  //--------------------------自定义异常类2 end---------------------------  //--------------------------测试类 start---------------------------  **public** **class** MyExceptionTest {  **static** **void** makeException(**int** a) **throws** MyException1,MyException2{  **if**(a<10)  **throw** **new** MyException1(a);  **if**(a>100)  **throw** **new** MyException2(a);  System.***out***.println("No Exception");  }    **public** **static** **void** main(String[] args){  **int** a;  **try**{  a = 1;  *makeException*(a);  System.***out***.println("a="+a);  /\*MyException1不可以改成Exception, MyException2可以，也就是说  多个catch,下面的catch的容错范围一定要大于上面的catch\*/  }**catch** (MyException1 e){  System.***out***.println("MyException1:"+e);  }**catch**(MyException2 e){  System.***out***.println("MyException2:"+e);  }  }  }  //--------------------------测试类 end----------------------------- |

### (3.2.1)try,catch,finally与return关系

|  |
| --- |
| /\*try…catch…finally块中的finally语句不被执行的两种情况：  \* return语句在try之前，还没执行try..catch语句块就返回了，当然不执行。  \* try语句中用了system.exit(0)强制退出JVM，暴力退出当然也不会执行。  \* \*/  /\*①：try中有return的情况\*/  **public** **class** TryReturnFinally {  **public** **static** **void** main(String[] args) {  System.***out***.println(*test1*());  }  **public** **static** **int** test1() {  **int** b = 1;  **try** {  System.***out***.println("try block");  **return** ++b;  }  **catch** (Exception e) {  b = 10;  System.***out***.println("catch block");  }  **finally** {  ++b;  System.***out***.println("finally block");  }  System.***out***.println("last block");  **return** b;  }  }  /\*执行结果是：  try block  finally block  2  以上结果说明了一点：return语句执行了，然后再执行了finally语句，但是return语句并没有直接返回，而是待finally语句执行完了再返回。  \*/  /\*下面的例子是加深理解上面的观点\*/  **public** **class** TryReturnFinally {  **public** **static** **void** main(String[] args) {  System.***out***.println(*test2*());  }  **public** **static** String test2() {  **try** {  System.***out***.println("try block");  **return** *test3*();  }  **catch** (Exception e) {  System.***out***.println("catch block");  }  **finally** {  System.***out***.println("finally block");  }  System.***out***.println("last block");  **return** "last block";  }  **public** **static** String test3() {  System.***out***.println("return block");  **return** "output return block";  }  }  /\*执行结果是：  try block  return block  finally block  output return block\*/  /\*②：finally和catch中有return的情况\*/  **public** **class** TryReturnFinally {  **public** **static** **void** main(String[] args) {  System.***out***.println(*test1*());  }  **public** **static** **int** test1() {  **int** b = 1;  **try** {  b = b/0;  System.***out***.println("try block");  **return** ++b;  }  **catch** (Exception e) {  b = 10;  System.***out***.println("catch block");  **return** ++b;  }  **finally** {  b += 10;  System.***out***.println("finally block");  //return b;  }    }    }  /\*执行结果是：  catch block  finally block  11  与try中return是一样的原理，将finallya中的注释放开，则结果是：  catch block  finally block  21  可见，若finally中有return则最终会执行finally中的return  \*/ |

## (3.3)日期

|  |
| --- |
| /\*Date与Calendar都可以操作时间, Date类不便于实现国际化(不同国家不同语言),若无特殊要求，依然可以使用Date。 \*/  ---------------------------------Date操作 start---------------------------  /\*Date与String相互转化\*/  **public static void** 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(Date转化成String)  // SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");  // String str = sdf.format(date);  // System.out.println(str);  // String->Date(String转化成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);  // ParseException解析异常  // } catch (ParseException e)  // 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1970年1月1日0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));    }  ---------------------------------Date操作 end---------------------------  -----------------------------Calendar操作 start-------------------------  **public** **static** **void** main(String[] args){  //获得当前时间,直接打印不直观，可用getTime方法转换成Date方式查看  Calendar c = Calendar.*getInstance*();  //使用Calendar类代表指定的时间  //月份的值为实际的月份值减1,2009年3月9日23点59分58秒，0代表1月  //c.set(2009, 3 - 1, 9,23,59,58);    //set方法设置相应时间的值  //Calendar.YEAR——年份  //Calendar.MONTH——月份，0代表1月  //Calendar.DATE——日期  //Calendar.DAY\_OF\_MONTH——日期，和上面的字段完全相同  //Calendar.HOUR——12小时制的小时数  //Calendar.HOUR\_OF\_DAY——24小时制的小时数  //Calendar.MINUTE——分钟  //Calendar.SECOND——秒  //Calendar.DAY\_OF\_WEEK——星期几,1代表星期日,7星期六（当前星期）  //c.set(Calendar.DAY\_OF\_WEEK,3);    //获得1号是星期几  /\*c.set(Calendar.DATE,1);  int day = c.get(Calendar.DAY\_OF\_WEEK);  System.out.println(day);\*/    //get方法获得Calendar类中的信息  //System.out.println(c.get(Calendar.DAY\_OF\_WEEK));  //System.out.println(c.getTime());  //add方法  //计算一下当前时间2天以后的日期  /\*c.add(Calendar.DATE, 2);  System.out.println(c.getTime());  System.out.println(c.get(Calendar.DATE));\*/    //after方法  //判断c1的日期是否在c后面，若是则为true,否则为false  /\*c.set(2009, 10 - 1, 10);  Calendar c1 = Calendar.getInstance();  c1.set(2010, 10 - 1, 10);  boolean b = c.after(c1);  System.out.println(b);\*/    //计算两个日期之间相差的天数  /\*c.set(2017, 10 - 1, 11);  Calendar c1 = Calendar.getInstance();  c1.set(2017, 11 - 1, 13);  //转换为相对时间  long t1 = c.getTimeInMillis();  long t2 = c1.getTimeInMillis();  //计算天数  long days = (t2 - t1)/(24 \* 60 \* 60 \* 1000);  System.out.println(days);\*/  }  -----------------------------Calendar操作 end--------------------------- |

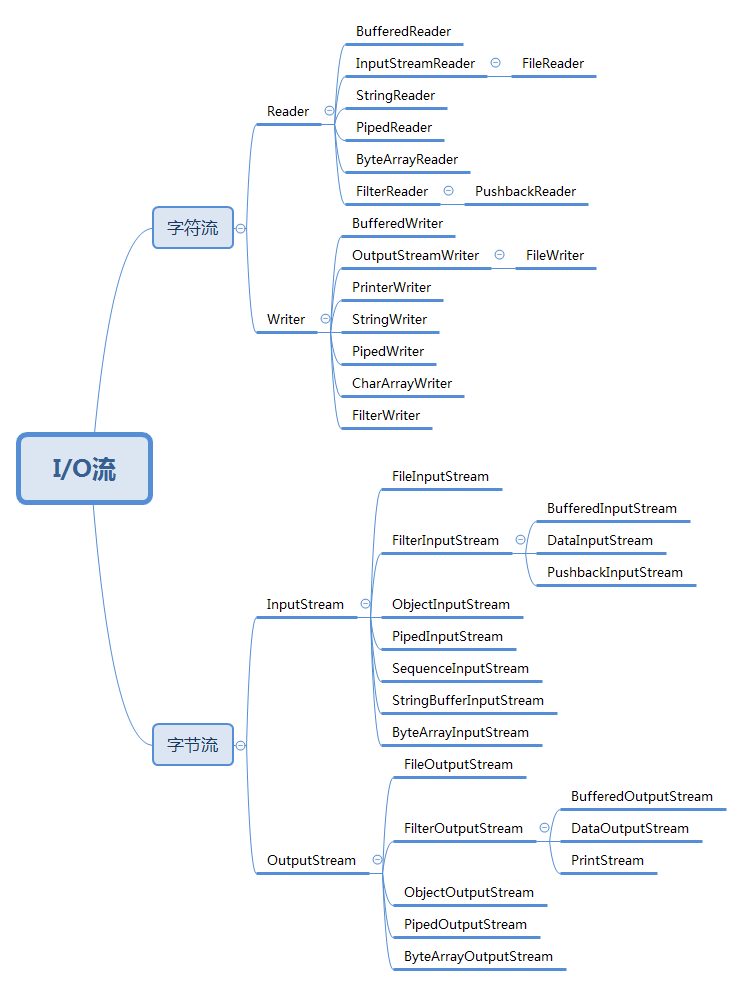
## (3.4)BigDecimal精确计算

|  |
| --- |
| /\*浮点数进行计算会有精度上的误差,看下面的例子\*/  **public** **static** **void** main(String[] args){  System.***out***.println(0.06+0.01);  System.***out***.println(1.0-0.42);  System.***out***.println(4.015\*100);  System.***out***.println(303.1/1000);  }  /\*运行的结果是：  0.06999999999999999  0.5800000000000001  401.49999999999994  0.30310000000000004  \*/  /\*原因在于我们的计算机是二进制的。浮点数没有办法是用二进制进行精确表示。我们的CPU表示浮点数由两个部分组成：指数和尾数，  \* 这样的表示方法一般都会失去一定的精确度，有些浮点数运算也会产生一定的误差。如：2.4的二进制表示并非就是精确的2.4。  \* 反而最为接近的二进制表示是 2.3999999999999999。举个例子（若不理解，知道结论即可）:  \* 二进制的1.11等于1\*2^0 + 1\*2^-1 + 1\*2^-2=1.75（十进制）  \* 表示0.2的浮点数：  \* 0.01 = 1/4 = 0.25 ,太大  \* 0.001 =1/8 = 0.125 , 又太小  \* 0.0011 = 1/8 + 1/16 = 0.1875 , 逼近0.2了  \* 0.00111 = 1/8 + 1/16 + 1/32 = 0.21875 , 又大了  \* 0.001101 = 1/8+ 1/16 + 1/64 = 0.203125 还是大  \* 0.0011001 = 1/8 + 1/16 + 1/128 = 0.1953125 这结果不错  \* 0.00110011 = 1/8+1/16+1/128+1/256 = 0.19921875  \* 已经很逼近了， 就这样吧。这就说明了用二进制小数没法精确表达10进制小数。  \* 在大多数的商业计算中，一般采用java.math.BigDecimal类来进行精确计算。  \*/  **public** **static** **void** main(String[] args){  //两种创建BigDecimal对象的方式：  //用构造方式创建BigDecimal对象,参数最好为字符串或者整数，否则又会丢失精度  BigDecimal b1 = **new** BigDecimal("0.2");  /\*此处的valueOf和java.lang.String.valueOf() 没有丝毫关系  这个静态方法把基本型构建成一个BigDecimal对象,不会丢失精度\*/  BigDecimal b2 = BigDecimal.*valueOf*(0.01);  //加法  System.***out***.println(b1.add(b2));  //减法  System.***out***.println(b1.subtract(b2));  //乘法  System.***out***.println(b1.multiply(b2));  //除法  System.***out***.println(b1.divide(b2));  }  /\*BigDecimal还可以有取绝对值，四舍五入、保留小数位等等功能，若涉及到浮点数计算，则使用BigDecimal即可  \*/ |

## (3.5)I/O

|  |
| --- |
| /\* 流是一组有顺序的，有起点和终点的字节集合，是对数据传输的总称或抽象。即数据在两设备间的传输称为流，流的本质是数据传输，根据数据传输特性将流抽象为各类，方便更直观的进行数据操作。I/O流的分类：①根据处理数据类型的不同分为：字符流和字节流  ②根据数据流向不同分为：输入流和输出流\*/ |

|  |
| --- |
| /\* I/O结构图 \*/ |



### (3.5.1)字符流和字节流

|  |
| --- |
| /\*（1）字符流和字节流的区别： ①读写单位不同：字节流以字节（8bit）为单位，字符流以字符（16bit）为单位，根据码表映射字符。  ②处理对象不同：字节流能处理所有类型的数据（如：图片、音乐、视频等媒体介质），而字符流只能处理字符类型的数据(如纯文本)。  （2）字符流和字节流的使用场景：  ①设备上的数据无论是图片或者视频，文字，它们都以二进制存储的。二进制的最终都是以一个8位为数据单元进行体现，所以计算机中的最小数据单元就是字节。意味着，字节流可以处理设备上的所有数据，所以字节流一样可以处理字符数据。  ②建议：只要是处理纯文本数据，就优先考虑使用字符流。 除此之外都使用字节流。  \*/  ----------------------字符流操作读取写出start---------------------------  **public** **class** Test {    **public** **static** **void** main(String[] args){    //这是我要copy的log文件，位于D盘根目录  String srcFile = "D:\\engine.log";    //这是我要copy的目标路径  String targetFile = "D:\\htmlDir\\example.html";    **try** {  FileUtil.*copy*(srcFile, targetFile);  FileUtil.*close*();  } **catch** (IOException e) {  System.***out***.println("出现异常：");  e.printStackTrace();  }  }  }  **public** **class** FileUtil {  **private** **static** FileReader *fr* = **null**;  **private** **static** BufferedReader *bf* = **null**;  **private** **static** FileWriter *fw* = **null**;  **private** **static** BufferedWriter *bw* = **null**;  **private** **static** File *tempFile\_1* =**null**;    //字节流读取文件  **private** **static** FileInputStream *fin* = **null**;  //字节流转化字符流的类  **private** **static** InputStreamReader *instre* = **null**;  /\*\*  \* 文件复制  \* **@param** srcFile 源文件路径  \* **@param** targetFile 目标文件路径  \* **@throws** Exception 异常信息  \*/  **public** **static** **void** copy(String srcFile,String targetFile) **throws** IOException{    //如果目标路径的目录不存在，创建它 ，\\前面的\是转义字符  String tempStr = targetFile.substring(0, targetFile.lastIndexOf("\\")+1);  //输出结果是D:\htmlDir\  //System.out.println(tempStr);  *tempFile\_1* = **new** File(tempStr);  //输出文件路径不存在，会自动创建，但文件夹不行  **if**(!*tempFile\_1*.exists()){  *tempFile\_1*.mkdirs();  }    //初始化流  //此种方式读取文件使用字节流读取  *fin*= **new** FileInputStream(srcFile);  //将字节流转化为字符流,InputStreamReader转化字符流  *instre* = **new** InputStreamReader(*fin*);  //将字符流放入缓存流中，BufferedReader的缓冲区大小为8kb(8192字节)  *bf* = **new** BufferedReader(*instre*);  //此种方式是直接用字符流读取，与上面的字节流效果一致  /\*fr = new FileReader(srcFile);  bf = new BufferedReader(fr); \*/  *fw* = **new** FileWriter(targetFile);  *bw* = **new** BufferedWriter(*fw*);    //读取第一行  String str = *bf*.readLine();    //判断文件有没有读取完  **while**(str!=**null**){  *bw*.write(str);  *bw*.newLine();  *bw*.flush();  str = *bf*.readLine();  }    System.***out***.println("文件复制成功");  }    /\*\*  \* 关闭流，关闭规则，与打开顺序相反，即最后打开最先关闭.  \* 不关闭流的影响：  \* 1.关闭文件会浪费系统资源，与数据库连接资源类似。程序每执行一段代码都占用一部分内存，  \* 如果不关闭的话肯定会浪费内存的，而且如果有动态分配的内存，那还会造成内存泄露。  \* 2.每个系统打开文件数量是有一限制的，如果打开文件不关闭，当连接数满的时候就无法再打开文件了。  \* 一般一个操作系统能最多打开1024个文件，但这个不一定，与机器硬件配置和系统配置有关。老点的机器只能同时打开256个或更低。  \* 3.IO流也有独占模式，也就是锁的概念，打开一个文件流以后，如果是以读的模式打开那就是共享锁,其它程序也可访问读取;  \* 如果以写的模式打开一般为独占，其它程序不能访问.具体表现情况还要看操作系统或者应用程序是如何处理的,因为底层的程序和高层的程序不一样。  \*/  **public** **static** **void** close() **throws** IOException{  *bw*.flush();  *bw*.close();  *fw*.close();  *bf*.close();  //字符流需要关闭的资源  //fr.close();  //字节流需要关闭的资源  *instre*.close();  *fin*.close();  }  }  ----------------------字符流操作读取写出end---------------------------  ----------------------字节流操作读取写出start---------------------------  /\*\*  \* 此种方式为字节流方式进行复制，效果与字符流一致，当数据量大的时候，  \* 用字符流的效果就体现出来了,若使用字符流处理图片的复制，则复制完成的  \* 图片不会正常显示。所以，字符流只能处理纯文本，字节流可以可以处理一切  \* 类型数据（最好不要处理纯文本）。  \* 本类没有用到缓冲流，是因为文件较小，下一个案例用到缓冲流。  \*/  **public** **class** Test {    **public** **static** **void** main(String[] args){    Test test = **new** Test();  //电脑d盘中的cat.png 图片的路径  String srcFile= "D:\\engine.log" ;  //电脑e盘中的cat.png 图片的路径  String targetFile = "D:\\htmlDir\\example.html" ;  //复制文件  test.copyFile(srcFile, targetFile);  }  /\*\*  \* 文件复制  \* **@param** srcFile : 需要复制文件的路径  \* **@param** targetFile : 复制文件存放的路径  \*/  **public** **void** copyFile( String srcFile , String targetFile){  FileInputStream fis=**null** ;  FileOutputStream fout = **null** ;  File tempFile\_1 =**null**;  **try** {  //如果目标路径的目录不存在，创建它 ，\\前面的\是转义字符  String tempStr = targetFile.substring(0, targetFile.lastIndexOf("\\")+1);    tempFile\_1 = **new** File(tempStr);  //输出文件路径不存在，会自动创建，但文件夹不行  **if**(!tempFile\_1.exists()){  tempFile\_1.mkdirs();  }    //1.根据path路径实例化一个输入流的对象  fis = **new** FileInputStream( srcFile );  //2. 返回这个输入流中可以被读的剩下的bytes字节的估计值；  **int** size = fis.available() ;  //3. 根据输入流中的字节数创建byte数组；  **byte**[] array = **new** **byte**[size];  //4.把数据读取到数组中；  fis.read( array ) ;  //5、根据文件路径创建输出流  fout = **new** FileOutputStream( targetFile ) ;    //6、把byte数组输出；  fout.write( array );  } **catch** (FileNotFoundException e) {  e.printStackTrace();  }**catch** (IOException e) {  e.printStackTrace();  }**finally**{  **if** ( fout != **null** ) {  **try** {  //7、关闭资源  fout.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  **if** ( fis != **null**) {  **try** {  //7、关闭资源  fis.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }    }  }  }  ----------------------字节流操作读取写出end---------------------------  ----------------------字节流缓冲操作读取写出start---------------------------  /\*\*  \* 此种方式为字节流缓冲方式进行复制  \*  \*  \*/  **public** **class** Test {    **public** **static** **void** main(String[] args){    String srcFile="D:\\33.png";  String target="D:\\htmlDir\\105.png";  FileInputStream fis=**null**;  FileOutputStream fos=**null**;  BufferedInputStream bis=**null**;  BufferedOutputStream bos=**null**;    **byte**[] ch=**new** **byte**[1024];  **int** len=0;    **try** {  //创建流对象  fis=**new** FileInputStream(srcFile);  //创建缓冲流 ,缓冲大小8kb(8192字节),调用read进行读取，读完，再进行缓冲  bis=**new** BufferedInputStream(fis);    String tempStr = target.substring(0, target.lastIndexOf("\\")+1);  File tempFile\_1 = **new** File(tempStr);  //输出文件路径不存在，会自动创建，但文件夹不存在不可以  **if**(!tempFile\_1.exists()){  tempFile\_1.mkdirs();  }    fos=**new** FileOutputStream(**new** File(target));  bos=**new** BufferedOutputStream(fos);  **while**((len=bis.read(ch))!=-1){  System.***out***.println(len);  bos.write(ch,0,len);  }  //刷新缓冲区  bos.flush();    } **catch** (IOException e) {  e.printStackTrace();  }**finally**{    **if** ( bos != **null** ) {  **try** {  //7、关闭资源  bos.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  **if** ( fos != **null**) {  **try** {  //7、关闭资源  fos.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  **if** ( bis != **null**) {  **try** {  //7、关闭资源  bis.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  **if** ( fis != **null**) {  **try** {  //7、关闭资源  fis.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  }  System.***out***.println("复制完成");  }  }  ----------------------字节流缓冲操作读取写出end--------------------------- |

线程

http://blog.csdn.net/ygd266/article/details/43530417

### 