Java SE编程

**2013/1/1 Tuesday**

徐振康 整理

Java SE 学习的摘要

**Java SE编程**

徐振康 整理

**目录**

[说明 10](#_Toc438295121)

[1.Java概述 10](#_Toc438295122)

[1.1 Java起源 10](#_Toc438295123)

[1.2 Java的发展 11](#_Toc438295124)

[1.3 Java的特点 11](#_Toc438295125)

[(一)、简单易学 11](#_Toc438295126)

[(二)、安全性高 11](#_Toc438295127)

[(三)、跨平台 12](#_Toc438295128)

[(四)、多线程的 12](#_Toc438295129)

[1.4 Java的应用领域 12](#_Toc438295130)

[1.5 参考资料: 12](#_Toc438295131)

[1.6 Java关键字 12](#_Toc438295132)

[1.7 学习目标 13](#_Toc438295133)

[2.Java的安装和卸载 13](#_Toc438295134)

[2.1 安装 13](#_Toc438295135)

[2.2 卸载 15](#_Toc438295136)

[3.环境变量的设置 16](#_Toc438295137)

[3.1 为什么要设置path? 16](#_Toc438295138)

[3.2 path的设置 17](#_Toc438295139)

[3.3 有关classpath设置的问题 17](#_Toc438295140)

[4.Java基础知识 17](#_Toc438295141)

[4.1预备知识 17](#_Toc438295142)

[DOS命令 17](#_Toc438295143)

[运行时注意的问题 17](#_Toc438295144)

[Java中的注释 18](#_Toc438295145)

[Java语言的基本要素——标识符 18](#_Toc438295146)

[4.2 数据类型 19](#_Toc438295147)

[Java数据类型的划分 19](#_Toc438295148)

[输出数据的格式控制 19](#_Toc438295149)

[常量……整型常量 20](#_Toc438295150)

[常量……浮点常量 20](#_Toc438295151)

[常量……字符常量 21](#_Toc438295152)

[字符常量的’\uxxxx’表示 21](#_Toc438295153)

[常量……布尔类型 21](#_Toc438295154)

[不同类型变量存储范围 22](#_Toc438295155)

[数据类型的转化 22](#_Toc438295156)

[4.3 运算符 23](#_Toc438295157)

[Java中的运算符 23](#_Toc438295158)

[算术运算符(+) 23](#_Toc438295159)

[除法运算符(/) 24](#_Toc438295160)

[取余运算符(%) 24](#_Toc438295161)

[逻辑与逻辑或 24](#_Toc438295162)

[存储单位介绍 25](#_Toc438295163)

[位运算符(1) 25](#_Toc438295164)

[位运算符 示例 25](#_Toc438295165)

[位运算符(2) 26](#_Toc438295166)

[运算符的优先级 26](#_Toc438295167)

[4.4 流程控制 26](#_Toc438295168)

[顺序 26](#_Toc438295169)

[选择 26](#_Toc438295170)

[循环 27](#_Toc438295171)

[4.5 函数的重载 27](#_Toc438295172)

[概念 27](#_Toc438295173)

[函数重载的要求 27](#_Toc438295174)

[5. 面向对象编程(上) 28](#_Toc438295175)

[5.1 什么叫面向对象 28](#_Toc438295176)

[什么叫做面向过程 28](#_Toc438295177)

[面向过程：自顶向下的设计模式 28](#_Toc438295178)

[面向过程设计思想 29](#_Toc438295179)

[面向过程设计的优点 29](#_Toc438295180)

[面向过程设计的缺点 29](#_Toc438295181)

[面向对象 30](#_Toc438295182)

[面向对象的设计思想 30](#_Toc438295183)

[举例观察面向对象与面向过程的区别 30](#_Toc438295184)

[举例 30](#_Toc438295185)

[面向过程的实现方式 30](#_Toc438295186)

[面向对象的实现方式 31](#_Toc438295187)

[5.2 什么叫类 33](#_Toc438295188)

[为什么要使用类？ 33](#_Toc438295189)

[什么是类？ 33](#_Toc438295190)

[什么是对象？ 33](#_Toc438295191)

[类与对象 33](#_Toc438295192)

[类的定义 34](#_Toc438295193)

[程序执行过程 34](#_Toc438295194)

[单个对象实例1 34](#_Toc438295195)

[类对象示例2 35](#_Toc438295196)

[类对象示例3 36](#_Toc438295197)

[5.3 访问控制符 37](#_Toc438295198)

[类的访问控制符 37](#_Toc438295199)

[类的访问控制符有四种 37](#_Toc438295200)

[外部访问包括两种方式： 37](#_Toc438295201)

[public 38](#_Toc438295202)

[private 38](#_Toc438295203)

[private修饰符(重点) 38](#_Toc438295204)

[5.4 构造函数 38](#_Toc438295205)

[类的构造函数 38](#_Toc438295206)

[构造函数的返回值问题 39](#_Toc438295207)

[构造函数数据成员的赋值问题 41](#_Toc438295208)

[多个构造函数可能带来的冲突 43](#_Toc438295209)

[5.5 关键字this 44](#_Toc438295210)

[5.6 关键字static 46](#_Toc438295211)

[static属性 46](#_Toc438295212)

[static方法 47](#_Toc438295213)

[6. 面向对象编程(下) 51](#_Toc438295214)

[6.1 继承 51](#_Toc438295215)

[继承的概念 51](#_Toc438295216)

[同包继承权限问题(重点) 53](#_Toc438295217)

[继承的原则 54](#_Toc438295218)

[不同访问修饰符 55](#_Toc438295219)

[使用继承时应该注意 55](#_Toc438295220)

[super的使用 55](#_Toc438295221)

[方法重写 58](#_Toc438295222)

[重写的意义 61](#_Toc438295223)

[6.2 多态 63](#_Toc438295224)

[多态的概念 63](#_Toc438295225)

[多态的优点 65](#_Toc438295226)

[举例 65](#_Toc438295227)

[多态注意事项(难点) 66](#_Toc438295228)

[7 抽象类和接口 69](#_Toc438295229)

[7.1 抽象类 69](#_Toc438295230)

[概念 69](#_Toc438295231)

[抽象类的由来 69](#_Toc438295232)

[抽象方法与抽象类 70](#_Toc438295233)

[抽象方法 70](#_Toc438295234)

[抽象类 70](#_Toc438295235)

[抽象类实现多台的示例 72](#_Toc438295236)

[7.2 Final关键字的使用 72](#_Toc438295237)

[Final 72](#_Toc438295238)

[Final修饰整个类 73](#_Toc438295239)

[final修饰类表示类不能被继承 73](#_Toc438295240)

[Final修饰类中的若干属性 73](#_Toc438295241)

[Final修饰类中的若干个方法 75](#_Toc438295242)

[7.3 接口 75](#_Toc438295243)

[接口(interface) 76](#_Toc438295244)

[接口的定义 76](#_Toc438295245)

[接口的格式 76](#_Toc438295246)

[实现的接口类(1) 78](#_Toc438295247)

[实现的接口类(2) 79](#_Toc438295248)

[接口的多态 79](#_Toc438295249)

[接口的作用 80](#_Toc438295250)

[接口与抽象类区别 80](#_Toc438295251)

[8 包 81](#_Toc438295252)

[8.1 package 的使用 81](#_Toc438295253)

[java zhangsan.lisi.TestPackage 解析 82](#_Toc438295254)

[8.2 同包不同类的相互访问 82](#_Toc438295255)

[8.3 不同包类的相互访问 83](#_Toc438295256)

[使用不同包中类的第一种方式 84](#_Toc438295257)

[使用不同包中类的第二种方式 84](#_Toc438295258)

[使用不同包中类的第三种方式 84](#_Toc438295259)

[8.4 归档工具jar 85](#_Toc438295260)

[jar使用举例如何使用jar包中的类 85](#_Toc438295261)

[8.5 不同的访问修饰符 86](#_Toc438295262)

[9 异常 88](#_Toc438295263)

[9.1为什么需要异常 88](#_Toc438295264)

[9.2 异常的处理机制 90](#_Toc438295265)

[9.3 常见的一些异常 91](#_Toc438295266)

[空指针异常 91](#_Toc438295267)

[下表越界异常 92](#_Toc438295268)

[算术异常 92](#_Toc438295269)

[异常的分类 93](#_Toc438295270)

[9.4 异常处理步骤 94](#_Toc438295271)

[异常处理步骤 94](#_Toc438295272)

[捕捉处理异常长注意问题 95](#_Toc438295273)

[finally的作用 96](#_Toc438295274)

[throw 98](#_Toc438295275)

[throws 98](#_Toc438295276)

[注意的问题 100](#_Toc438295277)

[9.5 自定义异常 102](#_Toc438295278)

[9.6 异常的优缺点 104](#_Toc438295279)

[异常的优点 104](#_Toc438295280)

[优点 106](#_Toc438295281)

[注意 106](#_Toc438295282)

[10. 字符串的使用 106](#_Toc438295283)

[10.1 toString方法 106](#_Toc438295284)

[System.out.println(类对象名);示例1 106](#_Toc438295285)

[System.out.println(类对象名);示例2 107](#_Toc438295286)

[toString方法总结 108](#_Toc438295287)

[10.2 String 与StringBuffer 108](#_Toc438295288)

[Object类的equals方法 108](#_Toc438295289)

[何时需要重写equals方法 109](#_Toc438295290)

[String类 111](#_Toc438295291)

[String类示例内存示意图 115](#_Toc438295292)

[String类常用方法 115](#_Toc438295293)

[String举例 116](#_Toc438295294)

[10.3 StringBuffer类 118](#_Toc438295295)

[StringBuffer类的由来 118](#_Toc438295296)

[StringBuffer类的构造函数 119](#_Toc438295297)

[StringBuffer常用方法 120](#_Toc438295298)

[StringBuffer举例 120](#_Toc438295299)

[11. 线程 121](#_Toc438295300)

[11.1 刚要 121](#_Toc438295301)

[线程的定义 121](#_Toc438295302)

[如何创建一个线程 121](#_Toc438295303)

[线程的控制 121](#_Toc438295304)

[线程同步 121](#_Toc438295305)

[11.2 初学者要注意四个问题 122](#_Toc438295306)

[11.3什么叫程序 122](#_Toc438295307)

[11.4 单道程序设计环境中程序特点 122](#_Toc438295308)

[11.5 多道程序设计环境中程序特点 122](#_Toc438295309)

[11.6 进程的由来 123](#_Toc438295310)

[11.7 什么事进程 123](#_Toc438295311)

[11.8 线程的定义 123](#_Toc438295312)

[11.9 创建一个线程的第一种方法 124](#_Toc438295313)

[11.10 创建一个新线程的第二种方法 129](#_Toc438295314)

[11.12 Thread的常用方法 130](#_Toc438295315)

[11.13 线程的控制 132](#_Toc438295316)

[线程状态的切换 132](#_Toc438295317)

[线程控制的基本方法 132](#_Toc438295318)

[线程的优先级 132](#_Toc438295319)

[线程的休眠 134](#_Toc438295320)

[run方法不可以抛出异常 135](#_Toc438295321)

[线程的让步 135](#_Toc438295322)

[线程的串行化 137](#_Toc438295323)

[线程的挂起和恢复 138](#_Toc438295324)

[声明周期控制 139](#_Toc438295325)

[11.14 线程的同步 140](#_Toc438295326)

[买票程序 140](#_Toc438295327)

[生产消费程序 146](#_Toc438295328)

[Synchromized关键字 150](#_Toc438295329)

[synchronized修饰代码块 150](#_Toc438295330)

[synchronized修饰方法 151](#_Toc438295331)

[同步概念 151](#_Toc438295332)

[notify和wait方法 153](#_Toc438295333)

[生产消费——经典问题 159](#_Toc438295334)

[12 GUI 159](#_Toc438295335)

[12.1 GUI基础 159](#_Toc438295336)

[组件 159](#_Toc438295337)

[容器 160](#_Toc438295338)

[Frame常用的方法 160](#_Toc438295339)

[Panel 165](#_Toc438295340)

[12.2 布局管理器 167](#_Toc438295341)

[GUI默认布局管理器 167](#_Toc438295342)

[FlowLayout布局管理器 167](#_Toc438295343)

[BorderLayout布局管理器 169](#_Toc438295344)

[GridLayout布局管理器 171](#_Toc438295345)

[布局管理器总结 172](#_Toc438295346)

[布局管理器练习题 172](#_Toc438295347)

[12.3 事件处理 174](#_Toc438295348)

[事件处理相关概念 174](#_Toc438295349)

[事件处理步骤 175](#_Toc438295350)

[遗留问题 175](#_Toc438295351)

[Button监听器 176](#_Toc438295352)

[TextField监听器 177](#_Toc438295353)

[Window监听器 181](#_Toc438295354)

[12.4 内部类 185](#_Toc438295355)

[内部类定义 185](#_Toc438295356)

[内部类访问原则 185](#_Toc438295357)

[内部类的优点 185](#_Toc438295358)

[何时使用内部类 185](#_Toc438295359)

[内部类举例 186](#_Toc438295360)

[12.5 匿名类 191](#_Toc438295361)

[创建匿名类的三种方式 192](#_Toc438295362)

[创建匿名类之实现接口 192](#_Toc438295363)

[创建匿名类之实现抽象类 193](#_Toc438295364)

[创建匿名类之继承父类 195](#_Toc438295365)

[匿名类的优缺点 197](#_Toc438295366)

[练习题：设计一个计算器预备知识 197](#_Toc438295367)

[练习题：设计一个计算器 202](#_Toc438295368)

[12.6 重绘 206](#_Toc438295369)

[附：可运行jar包生成步骤 208](#_Toc438295370)

[13 流 208](#_Toc438295371)

[13.1 流概述 208](#_Toc438295372)

[流与类的关系 208](#_Toc438295373)

[流的定义 209](#_Toc438295374)

[流的分类和使用 209](#_Toc438295375)

[13.2 四大基本抽象流 210](#_Toc438295376)

[InputStream流中常用的方法 211](#_Toc438295377)

[OutputStream流中常用的方法 211](#_Toc438295378)

[Reader流的常用方法 212](#_Toc438295379)

[Writer流中的常用方法 212](#_Toc438295380)

[13.3 文件流 213](#_Toc438295381)

[FileInputStream的使用 215](#_Toc438295382)

[FileReader的使用 216](#_Toc438295383)

[字节流与字符流区别 216](#_Toc438295384)

[实例 216](#_Toc438295385)

[13.4 缓冲流 219](#_Toc438295386)

[BufferedOutputStream和BufferedInputStream 220](#_Toc438295387)

[BufferedReader和BufferedWriter 223](#_Toc438295388)

[13.5 数据流 225](#_Toc438295389)

[DataInputStream 225](#_Toc438295390)

[DataOutputStream 226](#_Toc438295391)

[数据流实例 226](#_Toc438295392)

[13.6 转换流 229](#_Toc438295393)

[readLine()与回车符的问题 230](#_Toc438295394)

[readLine()与回车符的问题示例 231](#_Toc438295395)

[13.7 Print流 231](#_Toc438295396)

[Print流的由来 231](#_Toc438295397)

[PrintStream 232](#_Toc438295398)

[PrintWriter 233](#_Toc438295399)

[标准输入输出的重定向 233](#_Toc438295400)

[13.8 对象的序列化 235](#_Toc438295401)

[附：IO流一览表 238](#_Toc438295402)

[14 容器 239](#_Toc438295403)

[14.1 为什么需要容器 239](#_Toc438295404)

[容器与现实的对应关系 239](#_Toc438295405)

[容器的框架图 239](#_Toc438295406)

[14.2 容器的分类和使用 240](#_Toc438295407)

[Collection接口 240](#_Toc438295408)

[Collection接口中的方法介绍 240](#_Toc438295409)

[Collection接口的子接口——List接口 243](#_Toc438295410)

[Collection接口的子接口——Set接口 243](#_Toc438295411)

[Map接口 243](#_Toc438295412)

[预备知识：toString方法 244](#_Toc438295413)

[Collection接口实现类示例 245](#_Toc438295414)

[ArrayList与LinkedList的比较 247](#_Toc438295415)

[14.3 Collections类 248](#_Toc438295416)

[Collections类常用算法 248](#_Toc438295417)

[Collections类举例 248](#_Toc438295418)

[Collections.fill()用法 249](#_Toc438295419)

[14.4 Comparale接口 250](#_Toc438295420)

[为什么要使用Comparable接口 250](#_Toc438295421)

[Comparable接口介绍 252](#_Toc438295422)

[14.5 Set接口 254](#_Toc438295423)

[HashSet类 254](#_Toc438295424)

[14.6 equals()和hashCode()方法 254](#_Toc438295425)

[什么类必须得重写equals()和hashCode()方法 254](#_Toc438295426)

[为什么要重写equals()和hashCode()方法 258](#_Toc438295427)

[怎样重写equals()和hashCode()方法 261](#_Toc438295428)

[举例 261](#_Toc438295429)

[14.7 Iterator接口 263](#_Toc438295430)

[Iterator方法介绍 263](#_Toc438295431)

[TreeSet类 265](#_Toc438295432)

[示例 265](#_Toc438295433)

[HashSet和TreeSet的比较 269](#_Toc438295434)

[14.8 Map 269](#_Toc438295435)

[预备知识：哈希表 269](#_Toc438295436)

[哈希表的定义 269](#_Toc438295437)

[哈希表 269](#_Toc438295438)

[哈希表注意事项 269](#_Toc438295439)

[Map接口 269](#_Toc438295440)

[Map示例 270](#_Toc438295441)

[14.9 泛型 273](#_Toc438295442)

[15 网络编程 277](#_Toc438295443)

[15.1 基础知识 277](#_Toc438295444)

[基本概念 277](#_Toc438295445)

[网络程序 277](#_Toc438295446)

[IP 277](#_Toc438295447)

[端口号 277](#_Toc438295448)

[协议 278](#_Toc438295449)

[常见协议 278](#_Toc438295450)

[套接字（socket）的引入 278](#_Toc438295451)

[15.2 基于UDP的socket编程 278](#_Toc438295452)

[基于UDP的socket编程步骤 278](#_Toc438295453)

[基于UDP的socket编程应注意的问题 282](#_Toc438295454)

[15.3 基于TCP的socket编程 283](#_Toc438295455)

[基于TCP的socket编程步骤 283](#_Toc438295456)

[基于TCP的socket编程应注意的问题 284](#_Toc438295457)

[16 枚举 284](#_Toc438295458)

[16.1 原始的接口定义常量 284](#_Toc438295459)

[16.2 语法（定义） 284](#_Toc438295460)

[16.3 遍历、switch 等常用操作 285](#_Toc438295461)

[16.4 enum 对象的常用方法介绍 286](#_Toc438295462)

[16.5 给 enum 自定义属性和方法 287](#_Toc438295463)

[16.6 EnumSet，EnumMap 的应用 288](#_Toc438295464)

[16.7 enum原理分析 288](#_Toc438295465)

[16.8 总结 289](#_Toc438295466)

[17 注解(Annotation) 289](#_Toc438295467)

[17.1 注解基本概念 289](#_Toc438295468)

[什么是注解 289](#_Toc438295469)

[什么是metadata（元数据） 290](#_Toc438295470)

[Annotation和Annotation类型 290](#_Toc438295471)

[注解的分类 290](#_Toc438295472)

[17.2系统内置标准注解 291](#_Toc438295473)

[@Override，限定重写父类方法 291](#_Toc438295474)

[@Deprecated，标记已过时 292](#_Toc438295475)

[@SuppressWarnnings，抑制编译器警告 293](#_Toc438295476)

[17.3 元注解 294](#_Toc438295477)

[@Target 294](#_Toc438295478)

[@Retention 295](#_Toc438295479)

[@Documented 296](#_Toc438295480)

[@Inherited 296](#_Toc438295481)

[17.4 自定义注解 297](#_Toc438295482)

[17.5 注解元素的默认值 299](#_Toc438295483)

[17.6注解处理器 300](#_Toc438295484)

[注解处理器类库(java.lang.reflect.AnnotatedElement) 300](#_Toc438295485)

# 说明

本笔记的结构：由概念到注意事项再到具体的例子(大多用代码实现)

# 1.Java概述

## 1.1 Java起源

Java来自于sun公司的一个叫Green的项目，其原先的目的是为家用消费电子产品开发一个分布式代码系统，这样我们可以把E-mail发给电冰箱、电视机等家用电器，对它们进行控制，和他们进行信息交流。开始，准备采用C++，但是C++太复杂，安全性差，最后基于C++开发了一种新的语言Oak(Java的前身)，Oak是一种用于网络的精巧而安全的语言，Sun公司曾以此投标一个交互电视项目，但结果被SGI打败。可怜的Oak几乎无家可归，恰巧这时Mark Ardreesen开发的Mosaic和Netscape启发了Oak项目组成员，他们用Java编制了HotJava浏览器，得到了Sun公司首席执行官Scott McNealy的支持，触发了Java进军Internet。Java的取名也有一则趣闻，有一天，几位Java成员组的会员正在讨论给这个新的语言取什么名字，当时他们正在咖啡馆里喝着Java(爪哇)咖啡，有一个人灵机一动说就叫Java怎样，得到了其他人的赞赏，于是，Java这个名字就这样传开了。

## 1.2 Java的发展

1996年1月23日发布了JDK1.0

1997年2月18日JDK1.1

1998年12月4日JDK1.2

2000年5月8日 JDK1.3

2002年2月13日JDK1.4

历史上最为成熟的版本，性能得到了极大提高

2004年10月JDK1.5

易用，增加了诸如泛型、增强的for语句、可变数目参数、注释(Annotations)、自动拆箱(unboxing)和装箱等功能。

## 1.3 Java的特点

### (一)、简单易学

Java最初是为对家用电器进行集成控制而设计的一种语言，因此它必须简单明了。Java语言的简单性主要体现在四个方面：

1、Java的风格类似于C++，因而C++程序员初次接触Java语言，就会感到很熟悉。从某种意义上讲，Java语言是C和C++语言的一个变种，因此，C++程序员可以很快的掌握Java编程技术。

2、Java摒弃了C\C++中容易引发程序错误并且难以掌握的一些特性，如指针、结构、以及内村管理等。

3、Java提供了丰富的类库，可以帮助我们很方便的开发Java程序。

4、Java是完全面向对象的语言，所以它支持继承、重载、多态等面向对象的特性。C++语言是面向过程和面向对象混合的语言，C是纯面向过程的语言。

### (二)、安全性高

1、Java是一种强类型的语言，其类型检查比C/C++还要严格。类型检查帮助我们检查出很多开发早期出现的错误

2、Java提供了垃圾回收机制，有效的避免了C/C++中最头疼的内存泄露问题。

3、Java禁止非法内存访问，在没有授权的情况下是不能访问内存的。所有这些措施，使Java程序员不用再担心内存的崩溃。

总结：**Java去掉C和C++中影响程序健壮性的部分使程序更安全，例如指针、内存申请和释放**。

### (三)、跨平台

Java作为一种网络语言，其源代码被编译成一种结构中立的中间文件格式。只要有Java运行系统的机器都能执行这种中间代码。Java源程序被编译成一种与机器无关的字节码格式，在Java虚拟机上运行。

### (四)、多线程的

Java语言的一个重要特性就是在语言级支持多线程的程序设计。多线程就好像我们做一张桌子，如果你一个人完成这张桌子就好像单线程，那么你需要先做桌面，做完桌面后，在做4个桌子腿，如果现在有5个人来做桌子——在这个程序中开辟5个线程，其中一个人做桌面，另外4个人分别做4个桌子腿，那么这两种方式效率的高低，相信大家都能区分出来。

使得一个程序能够同时执行多个任务。

## 1.4 Java的应用领域

JAVA分为三个部分

一、J2SE，主要用来开发桌面应用软件的。

二、J2ME嵌入式开发，像手机里的软件、掌上电脑等等。

三、J2EE属于网络编程，JSP等等，就是做网站用到的编程。

## 1.5 参考资料:

《JAVA就业培训》张孝祥

《JAVA学习笔记》林信良

《JAVA核心编程》机械工业出版社

《JAVA编程思想》机械工业出版社

## 1.6 Java关键字

Java中一些赋以特定的含义，用作专门用途的字符串称为关键字(keyword)。

大多数编辑器会将关键字用特殊的方式标出

所有Java关键字都是小写英文。

goto和const虽然从未使用，但也被作为Java关键字保留。

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

表1-6-1 Java关键字

注：带\*号的关键字现在已不使用。

## 1.7 学习目标

了解程序语言级发展历史

熟练掌握JAVA语言的语法规则

掌握JAVA常用类的使用

掌握编程逻辑思维能力

能看懂程序

会调试程序

理解并应用面向对象的设计思想

为将来学习J2EE做准备

# 2.Java的安装和卸载

## 2.1 安装

下载java的安装文件，并且安装。



图2-1-1 安装Java(一)

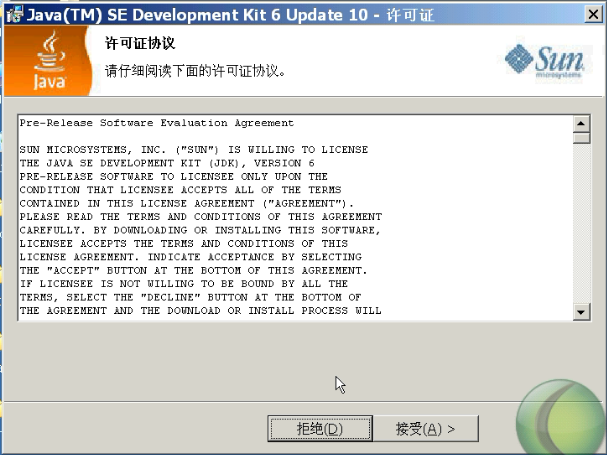


图2-1-2 安装Java(二)



图2-1-3 安装Java(三)



图2-1-4 安装Java(四)

## 2.2 卸载

到控制面板->添加或删除程序

找到



图2-2-1 卸载Java(一)

双击该项



图2-2-2 卸载Java(二)

# 3.环境变量的设置

## 3.1 为什么要设置path?

在dos的任何目录下我们都可以运行操作系统自带的命令

要想在dos下运行用户自己的程序，则必须进入到该程序的当前目录下方可运行

如果希望在dos的任何目录下都可以运行自己创建的程序，则需要我们自己手动设置操作系统自带的环境变量path。

## 3.2 path的设置

操作系统是利用path变量来寻找当前程序所存放的路径，并且以最先找到的为准

路径和路径之间用分号(;)隔开

set path=%path%;c:\Intel;

%path%;c:\Intel;表示在当前path变量中再添加一个路径为C:\Intel;并且该路径是添加在末尾的

set path

表示查看当前path变量中的内容。

## 3.3 有关classpath设置的问题

操作系统是利用classpath变量来寻找当前后缀为class的字节码文件所存放的路径，并且以最先找到的为准

从JDK1.6开始，classpath不需要再手动设置

初学者可以暂时不去理会classpath的设置

如果要导入第三方的jar包，则需要手动设置classpath

# 4.Java基础知识

## 4.1预备知识

### DOS命令

cd \

表示返回当前根目录

cd A\B\C

表示进入当前目录下的A文件夹的B文件夹下的C文件夹下面

E:

进入E盘根目录

### 运行时注意的问题

/\*假设该文件的名字是Test.java\*/

**class** Test {

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

System.*out*.printf("欢迎来到Java");

}

}

javac Test.java

java A

不能写java Test，编译时必须写文件名，运行时写的是文件中的类名。

### Java中的注释

三种注释：// /\*\*/ /\*\* \*/

单行注解，不需要配对

/\*\*/多行注解，需要配对

/\*\* \*/用于将注释变为说明文档，用javadoc实现。

### Java语言的基本要素——标识符

程序员对程序中的各个元素加以命名时使用的命名记号称为标识符(identifier)

包括：类名、变量名、常量名、方法名、……

Java语言中，标识符是以字母，下划线(\_)，美元符($)开始的一个字符序列，后面可以跟字母，下划线，美元符，数字。

合法的标识符

identifier username User\_Name

\_sys\_value $change

非法的标识符

2mail room# class

## 4.2 数据类型

### Java数据类型的划分

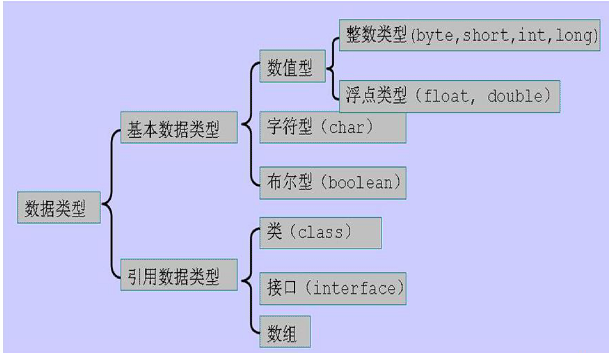


图4-2-1 Java数据类型的划分

### 输出数据的格式控制

|  |  |
| --- | --- |
| 输出控制符 | 针对的数据类型 |
| %d | int, long int, short, byte |
| %x, %#x, %X, %#X | int, long int |
| %c | char |
| %f | float, double |
| %s | string |

表4-2-1 输出控制符对应的数据类型

注意：

在Java中没有%ld和%lf，比如float和double都用%f输出，这是与C/C++不同的。

代码：

TestHex.java

**class** TestHex {

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

**int** i = 47;

System.*out*.printf("%d\n", i);

System.*out*.printf("%#x\n", i);

System.*out*.printf("%#X\n", i);

}

}

### 常量……整型常量

整型常量

十进制， 十六进制， 八进制

一个常量整数默认是int类型，如果数字过大，则必须得在末尾加L，否则会出错！

示例：

代码：

LongData.java

**class** LongData {

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

**long** i = 9223372036854775807L;

System.*out*.printf("%d\n", i);

**if** (1 > 0) {

System.*out*.printf("哈哈");

}

}

}

### 常量……浮点常量

浮点常量

一个实数默认是double类型，如果希望一个实数是float类型，可以再数字后面加f(F)

将一个double类型数值赋给float类型变量，编译时会报错

例子：

代码：

TestData.java

**class** TestData

{

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

{

**byte** b = 10; //1个字节

**int** i = 6; //4个字节

i = b; //OK

//b = i; //error 会丢失精度

b = (**byte**)i;//OK 强制类型转换

//b = i; //本语句错误，说明(byte)i并不会改变i本身的类型

**float** x;

//x = 10 \* 0.2;//error 10 \* 0.2；是double类型

x = 10 \* 0.2f;//OK

x = (**float**)(10 \* 0.2);//OK

System.*out*.printf(b + ", "+ i + ", " + x);

}

}

### 常量……字符常量

字符常量

**必须用单引号括起来**

Java中字符和字符串都用Unicode编码表示

在Unicode编码中一个字符占两个字节

‘a’ ‘B’ ‘n’ ‘\u0030’

特殊字符的常量表示法

反斜线(Backslash) ‘\\’

退格(Backspace) ‘\b’

回车(Carriage) ‘\r’

制表符(Form feed) ‘\t’

换行(New line) ‘\n’

单引号(Single quote) ‘\’’

### 字符常量的’\uxxxx’表示

代码：

A.java

**public** **class** A {

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

**char** ch1 = '中';

**char** ch2 = '\u4e2d';

System.*out*.printf("%c %c\n", ch1, ch2);

}

}

**‘\uxxxx’表示十六进制的xxxx所对应的Unicode编码下的字符**

### 常量……布尔类型

用boolean表示，不能写成bool

布尔类型数据只有两个值true和false，且它们不对应于任何整数值。

布尔类型变量的定义如：

boolean b – true;

布尔类型数据只能参与逻辑关系运算：

&& || != !

注意：

if while for 中进行真假判断时只能使用逻辑表达式

### 不同类型变量存储范围

byte占用**一个字节**，8bit，数字大小为~

short 占用**两个字节**，16bit，数字大小为~

int占用**四个字节**，32bit，数字大小为~

long占用**八个字节**，64bit，数字大小为~

float占用**四个字节**，数字大小为1.4E-45 ~ 3.4E+38， -1.4E-45 ~ 3.4E+38。用二进制的指数形式表示一个浮点数的格式，如：, 。

double占用**八个字节**，数字大小为4.9E-324 ~ 1.7E+308， -4.9E-324 ~ -1.7E+308。

char占用**两个字节**，数字大小为0~，是unicode编码。

boolean占用**一个字节**，其取值只有两个，true和false。

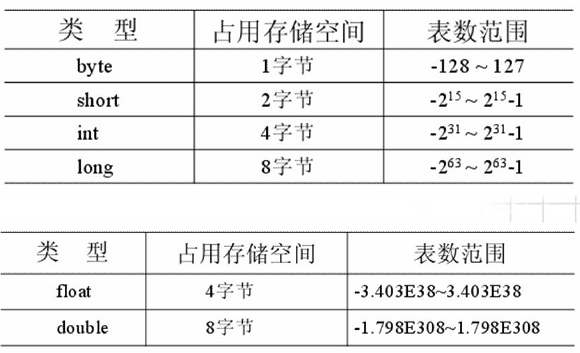


图4-2-2 Java数据类型占用空间和表数范围的情况

### 数据类型的转化

不能对boolean类型进行类型转化

在把容量大的类型转换为容量小的类型时必须使用强制类型转换。

整型、实型、字符型数据可以混合运算。运算中，不同类型的数据先转化为同一类型，然后进行运算，转换从低级到高级：

低…………………………………高

byte→char→int→long→float→double

例子：

代码：

TestData.java

**class** TestData

{

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

{

**byte** b = 10; //1个字节

**int** i = 6; //4个字节

i = b; //OK

//b = i; //error 会丢失精度

b = (**byte**)i;//OK 强制类型转换

//b = i; //本语句错误，说明(byte)i并不会改变i本身的类型

**float** x;

//x = 10 \* 0.2;//error 10 \* 0.2；是double类型

x = 10 \* 0.2f;//OK

x = (**float**)(10 \* 0.2);//OK

System.*out*.printf(b + ", "+ i + ", " + x);

}

}

## 4.3 运算符

### Java中的运算符

算术运算符

+ - \* / %

关系运算符

> >= < <= != ==

逻辑运算符

! && ||

赋值运算符

= += \*= /= %=

位运算符

& | ~ ^

### 算术运算符(+)

+可以表示数值的相加

+可以表示字符串的连接， 如：“123”+“abc”的结果是“123abc”

+还能把非字符串换成字符串， 如：“x”+123;的结果是“x123”

分析System.out.println(‘a’ + 1)与System.out.println(“” + ‘a’ + 1)的区别

### 除法运算符(/)

除法/的运算结果和运算对象的数据类型有关，若两个数都是int，则商就是int，若商有小数，则截取小数部分；若两个对象中有一个或两个都是浮点型数据，则商也是浮点型，不截取小数部分

如：

16 / 5 == 3 16 / 5.0 == 3.20000 -13 / 4 == -4 -13 / -3 == 4 3 / 5 == 0 5 / 3 == 1

测试源代码如下：

代码：

A.java

**public** **class** A

{

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

{

**double** x = 1 / 2.0; // 考虑改为1/2 1.0/2 1.0/2.0 输出结果会怎样？

System.*out*.printf("%f\n", x);

}

}

结果：0.500000

### 取余运算符(%)

Java中允许取余运算符的被除数和除数是实数(这与C/C++是不同的)，但所得余数的正负只和被除数相同

例子

double x = 1 % -0.3;

System.out.println(x);

输出结果：0.10000000000000003

### 逻辑与逻辑或

当&&左边表达式为假时，&&右边的表达式是不执行的。因为无论&&右边的表达式为真还是为假，整个表达式都为假

同理，当||表达式左边为真时，||右边的表达式是不会执行的

### 存储单位介绍

一个0或一个1就代表一位(bit)

一位(bit)只能存储一个0或一个1

8位(bit)代表一个字节(byte)

位用bit表示

字节用byte表示

### 位运算符(1)

&(按位与)

把两个数字所有的二进制位相与

1 & 1 = 1 1 & 0 = 0 0 & 0 = 0 0 & 1 = 0

|(按位或)

把两个数字所有的二进制位相或

1 | 1 = 1 1 | 0 = 1 0 | 0 = 0 0 | 1 = 1

~(按位取反)

把一个数字的所有二进制位取反

~1 = 0 ~0 = 1

^(按位异或)

把两个数字的所有二进制位异或

1 ^ 1 = 0 1 ^ 0 = 1 0 ^ 0 = 0 0 ^ 1 = 1

### 位运算符 示例

代码：

A.java

**class** A {

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

**int** i = 11;

**int** j = -2;

System.*out*.printf("%d\n", i & j);

}

}

执行结果：10

### 位运算符(2)

右移>>

在java中右移没有任何歧义， 这与C/C++不同。对于有符号数，在右移时，符号位将随同移动。当为正数时，最高位0，最高位补0，而为负数时， 最高位为1，最高位补1.

>>>

无论最高位是0还是1，左边移空的改为都补0

<<和<<<同理

### 运算符的优先级

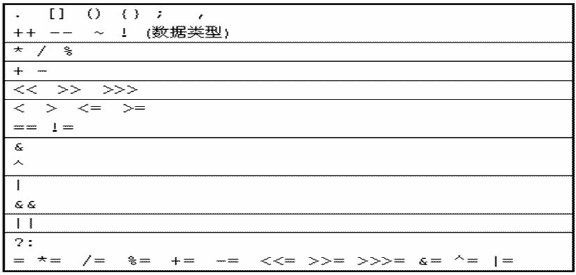


图4-3-1 运算符的优先级

运算表达式书写规则：

1.不要写过与复杂的表达式

2.为了程序的可读性，可以适当的使用括号

## 4.4 流程控制

### 顺序

### 选择

if …… else

switch

### 循环

for

while

do …… while

注意：continue break 选择 循环的混合使用

## 4.5 函数的重载

### 概念

同名的函数通过不同的形参做类似的事情，这就叫**函数的重载**。

### 函数重载的要求

函数的形参**个数**

函数的形参**顺序**

函数的形参**数据类型**

\*这三个至少有一个是不一样的

**如果两个函数只是函数的返回值不一样，其他都一样，这构不成函数的重载，并且编译时会报错！**

代码：

TestOver.java

**class** TestOver

{

**static** **int** add(**int** a, **int** b)

{

**return** a + b;

}

**static** **double** add(**double** x, **double** y)

{

**return** x + y;

}

**static** **int** add(**int** a, **int** b, **int** c)

{

**return** a + b + c;

}

**static** **int** add(**int** a, **int** b, **int** c, **int** d)

{

**return** a + b + c + d;

}

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

{

System.*out*.printf("(整型)a + b = %d\n", *add*(1, 2));

System.*out*.printf("(整型)a + b + c = %d\n", *add*(1, 2, 3));

System.*out*.printf("(整型)a + b + c + d = %d\n", *add*(1, 2, 3, 4));

System.*out*.printf("(双精度浮点型)a + b = %f\n", *add*(1.1, 2.2));

}

}

/\*

执行结果

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

(整型)a + b = 3

(整型)a + b + c = 6

(整型)a + b + c + d = 10

(双精度浮点型)a + b = 3.300000

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

\*/

# 5. 面向对象编程(上)

## 5.1 什么叫面向对象

### 什么叫做面向过程

#### 面向过程：自顶向下的设计模式

main

SubFunction1

SubFunction2

SubFunction3

Sub1\_Sub1

Sub3\_Sub1

Sub3\_Sub2

………

图5-1-1 面向过程

#### 面向过程设计思想

1.分析出解决问题所需要的步骤，然后用函数把这些步骤一步一步实现

2.以算法为核心

3.自顶向下设计，要求一开始必须对问题有很深的了解

4.将大问题转化为若干小问题来求解

5.表现形式：用函数来作为划分程序的基本单位

6.直接面向问题

#### 面向过程设计的优点

1.易于掌握与理解，符合人们的思维习惯

2.对于需求明确、规模不大、变动较小的问题非常合适

#### 面向过程设计的缺点

1.数据与操作分离开，对数据与操作的修改变得很困难

2.数据的安全性得不到保证

3.程序架构的依赖关系不合理

main函数依赖于子函数，子函数有依赖于更小的子函数；而子函数往往是细节的实现，这些实现是经常变化的，造成的结构就是：程序的核心逻辑依赖于外延的细节，一个细节上的小改动，会引起一系列的变动。

4.对于“需求不明确、变动较大、规模很大的问题”，显得力不从心

### 面向对象

面向对象的三大特点：封装，继承，多态

#### 面向对象的设计思想

1.确定该问题由哪些事物组成！先用类模拟出该事物

2.通过类间接的解决问题

3.自下而上设计，从问题的一部分着手，一点一点地构建出整个程序

4.表现形式：用类来作为划分程序的基本单位

5.对于需求不明确、变动较大、规模很大的问题非常适合

6.对于“需求明确、规模不大、变动较小的问题”则显得十分累赘

### 举例观察面向对象与面向过程的区别

#### 举例

编程实现求三角形的长度和面积

#### 面向过程的实现方式

1.定义三个可以存储三角形三边的变量

2.定义两个函数，分别实现求周长和求面积

3.将三个变量发送给这两个函数，求出周长和面积

缺点：

1.可以任意修改三边的值

2.可以把这三边发送给其他的函数

3.两个函数可能接受到不是边长的垃圾数字

4.三边和两个函数不是一个有机整体， 组合度太低

代码：

TestTriangle\_1.java

**public** **class** TestTriangle\_1

{

**public** **static** **int** zhouchang(**int** i, **int** j, **int** k)

{

**return** (i + j + k);

}

**public** **static** **double** area(**int** i, **int** j, **int** k)

{

**double** p = 1.0 \* (i+j+k) / 2;

**return** Math.*sqrt*(p \* (p-i) \* (p-j) \* (p-k));

}

**public** **static** **int** f(**int** i, **int** j, **int** k)

{

**return** i \* j \* k \* 3 \* 2 - 1;

}

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

{

**int** m, n, r;

m = 3;

n = 4;

r = 5;

System.*out*.printf("三角形的面积是: %f\n", *area*(m, n, r));

System.*out*.printf("三角形的周尝试: %d\n" , *zhouchang*(m, n, r));

*f*(m, n, r);

}

}

#### 面向对象的实现方式

1. 设计一个新的数据类型，名字可以叫做Triangle

2. Triangle内部含有两部分

三角形的三条边，用变量表示

两个专门针对这三条边的函数，分别用于求长度和面积

3.用设计好的Triangle生成一个三角形变量

4.如果用户想执行什么操作，直接调用该变量中的函数即可！

5.优点：

待处理的数据和对待处理数据的操作是个有机整体，外部无法修改三边的值，这两个函数也不会处理外部的垃圾数据。

数据的安全性得到了极大的增强

更符合人类的思考方式，编程简单化

代码：

TestTriangle\_2.java

**class** Triangle

{

**private** **int** a, b, c;

//设置三边长度

**public** **void** setTriangle(**int** i, **int** j, **int** k)

{

a = i;

b = j;

c = k;

}

//求周长

**public** **int** zhouchang()

{

**return** (a+b+c);

}

//求面积

**public** **double** area()

{

**double** p = 1.0 \* (a+b+c)/2;

**return** Math.*sqrt*(p \* (p-a) \* (p-b) \* (p-c));

}

}

**public** **class** TestTriangle\_2

{

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

{

Triangle t = **new** Triangle();

t.setTriangle(3, 4, 5);

System.*out*.printf("三角形的面积是: %f\n", t.area());

System.*out*.printf("三角形的周尝试: %d\n" , t.zhouchang());

}

}

## 5.2 什么叫类

### 为什么要使用类？

1.采用简单数据类型表示现实世界中概念存在很大的局限性。例如：采用int型数据表示一个日期概念，需要使用3个变量：

int day, month, year;

2.如果要表示2个人的生日，就要使用6个变量

3.并且在使用中必须时刻注意三者的联系和约束关系

4.同时在使用日期概念时要同时对三个变量进行访问

**使用类可以把现实问题中的对象映射为程序中的一个整体——类。**

### 什么是类？

1.把一类事物的静态属性和动态可以执行的操作组合在一起所得的这个概念就是**类**

2.类是抽象的，用来模拟一类事物，是一个概念

3.一旦被定义，类的概念就永远存在了

### 什么是对象？

1.类的一个个体

2.具体的，实实在在存在的事物

3.生命周期是短暂的，会生成和消亡

### 类与对象

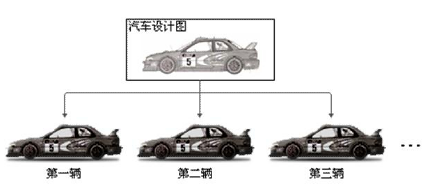


图5-2-1 类和对象

如果将对象比作汽车，那么类就是汽车的设计图纸

### 类的定义

代码：(无启动类)

**class** Person

{

**int** age;

**void** shout()

{

System.*out*.println("Oh,my god! I am " + age);

}

}

age是**类的属性**，也叫**类数据成员**，也叫**字段**，也叫**域**。

shout是**方法**，也叫**类的成员函数**。

shout方法可以直接访问同一个类中的age变量。

### 程序执行过程

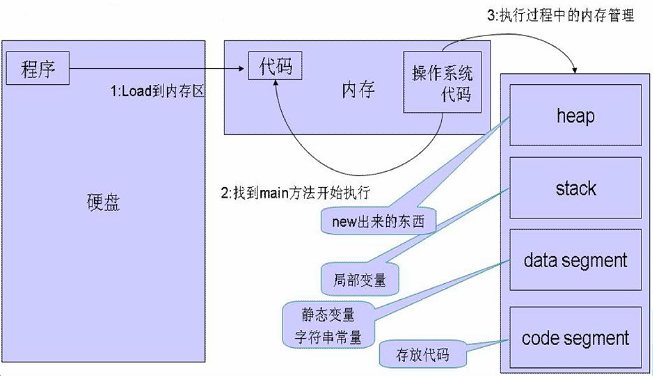


图5-2-2 程序执行过程

#### 单个对象实例1

代码：

TCS.java

**class** A {

**int** i = 10;

**int** j = 20;

}

**public** **class** TCS {

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

A aa1 = **new** A();

}

}

堆空间

栈空间

aa1:XXXX

i :10

j : 20

图5-2-3 单个对象实例1

在Java中如果一个引用变量指向了某块对内存，那么我们就可以认为这个引用变量表示的就是该变量指向的那块对内存。

#### 类对象示例2

代码：

TCS.java

**class** A {

**int** i = 10;

**int** j = 20;

}

**public** **class** TCS {

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

A aa1 = **new** A();

A aa2 = **new** A();

}

}

aa1:XXXX

i = 10

j = 20

aa2:XXXX

i = 10

j = 20

栈空间

堆空间

图5-2-4 单个对象实例2

#### 类对象示例3

代码：

TCS.java

**class** A {

**int** i = 10;

**int** j = 20;

}

**public** **class** TCS {

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

A aa1 = **new** A();

A aa2 = aa1;

}

}

aa1:XXXX

aa2:XXXX

i = 10

j = 20

分为以sic= new A();id main()gsdfgsdfgdfgdfgsdfg分为以sic= new A();id main()gsdfgsdfgdfgdfgsdfg

栈空间

堆空间

图5-2-5 单个对象实例3

## 5.3 访问控制符

### 类的访问控制符

#### 类的访问控制符有四种

1.public

2.protected

3.默认(即不加任何修饰符)

4.private

**在一个类的内部，所有的成员可以相互访问，访问控制符是透明的；访问控制符是针对外部访问而言的**

#### 外部访问包括两种方式：

**1.通过类名访问类内部的成员**

**2.通过类对象名访问类内部成员**

#### public

可以通过外部访问方式访问类内部的public成员

#### private

不可以通过外部访问方式访问类内部的private成员

其他内容在学完继承和包之后介绍

### private修饰符(重点)

在一个类的内部，所有的成员可以相互访问，访问控制符是透明的

在一个类的外部：通过 **类对象名.私有成员名** 的方式是无法访问该对象中的私有成员的，这样写编译时会出错！

## 5.4 构造函数

### 类的构造函数

特点：

1.函数名与类名是相同的

2.不能有返回值

3.可以有参数，也可以没有参数

4.可以有多个

说明：

声明一个对象，系统首先为该对象分配内存，然后立即自动调用该对象的构造函数

注意：

1.任何一个类对象被生成时一定会调用该类的构造函数

2.无论一个类有多少构造函数，生成一个类对象时一定只会调用其中的某一个构造函数！

代码：

TestConst.java

**class** A

{

**public** **int** i;

**public** **int** j;

//构造函数

**public** A()

{

System.*out*.printf("无参构造函数被调用了！");

}

}

**class** TestConst

{

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

{

A aa = **new** A();

}

}

/\*

执行结果：

////////////////////////////////////

无参构造函数被调用了！

////////////////////////////////////

分析：

只有当main中生成一个A类， 7~10行才会被执行。构造了A类之后， 马上就会执行7~10行。

\*/

### 构造函数的返回值问题

1.在一个类中可以定义多个函数名与类名相同但却有返回值的函数，返回值为void、int或double都可以，这些有返回值的函数只要能满足重载特点，就可以同时存在一个类中，不过要注意：这些有返回值的函数(包括返回值为void的函数)都不是构造函数，都不会被类对象自动调用

2.当然也可以定义多个没有任何返回值的函数，注意连void都不可以加，这些函数才是构造函数，无返回值的函数才会被对象自动调用

3.如果定义了有返回值并且方法名与类名相同的方法，程序并不会报错，但容易产生歧义，所以**强烈建议不要在构造方法前面加返回值**

代码：

TestConst\_1.java

/\*

函数名与类名相同的函数用法

\*/

**class** A

{

**private** **int** i;

//这不是构造函数，创建类对象时是不会自动被调用的,与16行的A函数构成重载

**public** **void** A(**int** m) //6行

{

i = 11111;

System.*out*.println("调用了有参的函数");

}

**public** A() //将本函数注释掉 则输出结果是 ： i = 0, 如果本函数不被注释掉， 则输出结果是: i = 10

{

**this**.i = 10;

}

//这不是构造函数，创建类对象时是不会自动被调用的,与6行的A函数构成重载

**public** **int** A() //16行

{

i = 22222;

System.*out*.println("调用了无参的函数");

**return** 888;

}

**public** **void** shout()

{

System.*out*.println("i = " + i);

}

}

**class** TestConst\_1

{

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

{

A ra = **new** A();

ra.shout();

ra.A();

ra.A(1);

}

}

/\*

在JDK 1.6中的运行结果是：

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

i = 10

调用了无参的函数

调用了有参的函数

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

总结：

在一个类中可以定义多个函数名与类名相同但却有返回值的函数，

返回值为void 或int 或 double都可以，这些有返回值的函数只要

能满足重载特点，就可以同时存在一个类中,不过要注意：这些有返回值的

函数(包括返回值为void的函数)都不是构造函数，都不会被类对象自动调用

当然也可以定义多个没有任何返回值的函数，注意连void都不可以加，

这些函数才是构造函数

\*/

### 构造函数数据成员的赋值问题

当一个对象呗创建时，会对其中各种类型的成员变量自动进行初始化赋值。

|  |  |
| --- | --- |
| 成员变量类型 | 初始值 |
| byte | 0 |
| short | 0 |
| int | 0 |
| long | 0L |
| float | 0.0F |
| double | 0.0D |
| char | ‘\u0000’(表示为空) |
| boolean | false |
| All reference type | null |

表5-4-1 各种变量类型的初始化

1.如果在定义的时候不初始化，则它的值是系统自动分配好的默认值！

2.如果在定义的同时赋初值，则是可以的，也就是说该值是生效的。注意C++中则不可以，在C++占用一个类的数据成员不能在定义的同时初始化，它只能在构造函数中初始化

3.如果在定义的同时赋初值，当然生效，但如果在构造函数中又改变了定义时赋的初值，则该数据成员最终的值就是构造函数中修改之后的那个值，因为：

**系统会先执行定义时赋的初值，然后再执行构造函数中赋的初值**

代码：

TestConst\_3.java

/\*

2013年2月4日17:41:29

一个类中的数据成员

1、 如果在定义的时候不初始化，则它的值是系统自动分配好的默认值！ 如int型为零 boolean型是false

如本程序的A类对象就是这样

2、 如果在定义的同时赋初值， 则是可以的，也就是说该值是生效的.注意在C++中则不可以，在C++中一个类的数据成员不能在定义的同时初始化，它只能在构造函数中初始化

如本程序的A类对象就是这样

3、 如果在定义的同时赋初值，当然生效，但如果在构造函数中又改变了定义时赋的初值，

则该数据成员最终的值就是构造函数中修改之后的那个值,因为：

系统会先执行定义时赋的初值，然后再执行构造函数中赋的初值

如本程序的B类对象就是这样

\*/

**class** A

{

**int** i;

**int** j = 10;

**boolean** flag;

**void** show()

{

System.*out*.println("i = " + i);

System.*out*.println("j = " + j);

System.*out*.println("flag = " + flag);

}

}

**class** B

{

**int** i;

**int** j = 10;

**boolean** flag;

**public** B()

{

System.*out*.println("以前的值是 " + i + " " + j + " " + flag);

i = 88;

j = 88;

flag = **true**;

}

**void** show()

{

System.*out*.println("i = " + i);

System.*out*.println("j = " + j);

System.*out*.println("flag = " + flag);

}

}

**class** TestConst\_3

{

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

{

A aa1 = **new** A();

aa1.show();

B bb1 = **new** B();

bb1.show();

}

}

/\*

在j2sdk1.4.2\_16中的运行结果是：

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

i = 0

j = 10

flag = false

以前的值是 0 10 false

i = 88

j = 88

flag = true

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

\*/

### 多个构造函数可能带来的冲突

1.在Java每个类里都至少有一个构造方法，如果程序员没有在一个类里定义构造方法，系统会自动为这个类产生一个默认的构造方法，这个默认构造方法没有参数，在其方法体中也没有任何代码，即什么也不做。

2.由于系统提供的默认构造方法往往不能满足编程者的需求，我们可以自己定义类的构造方法，来满足我们的需要

3.**一旦编程者为类定义了构造方法，系统就不再提供默认的构造方法了**。

代码：

TestConst\_4.java

**class** A {

**private** **int** i;

// 如果把6到8行代码注释掉，则程序会报错！

**public** A()// 6行

{

}

**public** A(**int** j) {

i = j;

}

}

**public** **class** TestConst\_4 {

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

A aa1 = **new** A();

A aa2 = **new** A(1);

}

}

## 5.5 关键字this

this 是一个系统隐含的指针被自动附加在非静态的成员函数参数列表中

当前时刻，哪个对象调用该函数，那么this就指向当前调用该函数对象，系统会自动在该函数的参数列表中添加一个隐藏的this指针，并且把调用该函数的对象的地址赋给this指针，这样一来，在函数的内部通过this就可以访问当前正在调用该函数的对象的成员。

静态函数内部，没有this指针

关键字this代表当前正在调用该方法的那个对象，具体可以分为两种情况：

1.在普通方法中，关键字this代表方法的调用者、即本次调用了该方法的对象；

2.在构造方法中，关键字this代表了该方法本次运行所创建的那个新对象。

代码：

TestThis1.java

**class** A

{

**public** **int** i;

**public** A(**int** j)

{

i = j;

}

**public** **void** show()

{

System.*out*.println("i = " + i);

}

}

**class** TestThis1

{

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

{

A aa1 = **new** A(5);

A aa2 = **new** A(8);

aa1.show();

aa2.show();

}

}

程序分析：

aa1和aa2在内存中分别有各自的数据成员i，但是aa1和aa2公用show()方法，show方法如何知道输出的i应该是哪个对象中的i？

实际上**每个非static方法中都隐含这一个this指针，这项当前正在调用该方法的对象**

代码：

TestThis.java

**class** A

{

**private** **int** i;

**public** A(**int** i)

{

**this**.i = i; //将形参 i 赋给该构造方法本次运行所创建的那个新对象的i数据成员

}

**public** **void** show(){

System.*out*.println("i = " + **this**.i);

//this表示当前时刻正在调用show方法的对象

//this可以省略

}

}

**public** **class** TestThis

{

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

A aa1 = **new** A(100);

aa1.show();

A aa2 = **new** A(200);

aa2.show();

}

}

## 5.6 关键字static

### static属性

静态成员属于类本身的，而不是属于对象，被类的所有对象所共有

**即便不创建对象，也可以使用类本身的静态成员**

静态成员分为

1.静态数据成员

2.静态方法成员

使用静态成员的两种方法

1.类名.静态成员名

2.类对象名.静态成员名

编写使用静态变量统计一个类产生的实例对象的个数的程序

代码：

TestStatic\_1.java

**class** Student

{

**public** **static** **int** *cnt* = 0;

**private** String sname;

**private** **int** sage;

**public** Student()

{

*cnt*++;

}

**public** Student(String name, **int** age)

{

**this**.sname = name;

**this**.sage = age;

*cnt*++;

}

}

**public** **class** TestStatic\_1

{

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

{

System.*out*.println("Student.cnt = " + Student.*cnt* );

Student st1 = **new** Student("zhangsan", 20);

Student st2 = **new** Student("lisi", 30);

System.*out*.printf("Student类总共构造了%d个对象!\n", Student.*cnt*);

}

}

/\*在开辟内存时不需要每个都开辟，只需开辟一段static的内存空间\*/

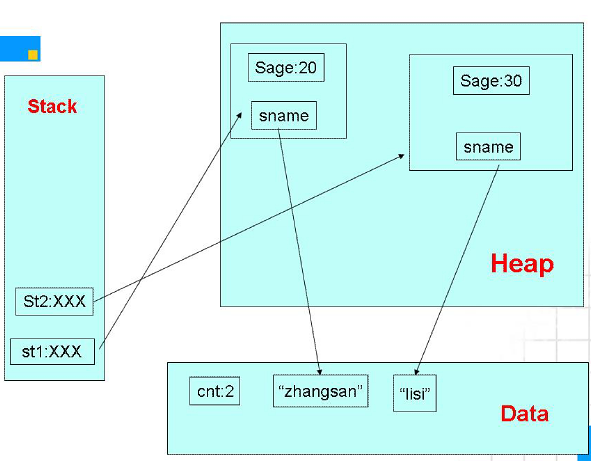


图5-6-1

### static方法

**在静态方法里只能直接调用同类中其他的静态成员(包括变量和方法)，而不能直接访问类中的非静态成员。**这是因为，对于非静态的方法和变量，需要先创建类的实例对象后才能使用，而静态方法在使用前不用创建任何对象。

**静态方法不能以任何方式引用this和super关键字**(super关键字在下一章讲解)**。**与上面的道理一样，因为静态方法在使用前不用创建任何实例对象，当静态方法被调用时，this所引用的对象好根本就没有产生。

静态方法只能访问类的静态成员，但**非静态方法却值可以访问类中所有成员，包括静态成员。**

代码：

TestStatic\_1.java

/\*

2013年2月6日22:04:30

本程序证明了:

1.static属性i是属于类本身，或者讲：没有对象我们仍然可以直接通过类名的方式访问该类内部的static属性。

2.static方法f同理

3.static属性和方法不仅属于类本身，不仅可以通过类名方式访问，而且static属性和方法很明显也属于类对象，当然也可以通过类对象名的方式来访问。

4.只有非private的static成员可以通过类名的方式访问，static只是表明了该成员具有了可以通过类名的潜在特征，但是是否可以通过类名方式访问，还必须满足一个条件：该成员必须是非private

5.非静态方法可以访问静态成员，而静态方法不能访问非静态成员。(原因是，只有造出了对象之后才能访问非静态方法，那么，利用非静态方法访问静态成员时，说明成员已经造好，所以可以访问静态成员；同理，利用静态方法访问非静态成员时，对象并不一定造好，而非静态成员一定要在对象造好之后才能通过对象访问，所以，不能用静态成员访问访问非静态成员。)

\*/

**class** A

{

**public** **static** **int** *i* = 10;

**public** **static** **void** f()

{

//g(); // error静态方法不能访问非静态成员

//j = 22; // error理由同上

System.*out*.printf("FFFF\n");

System.*out*.printf("2013年2月6日22:09:16");

}

**public** **void** g()

{

//f(); //OK说明非静态方法可以访问静态成员

System.*out*.printf("GGGG\n");

System.*out*.printf("%d\n", *i*);

}

}

**class** TestStatic\_1

{

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

{

System.*out*.printf("%d\n", A.*i*);

A.*f*();

A aa = **new** A();

aa.*f*();

}

}

代码：

TestGetCnt.java

/\*

功能：记录生成对象的个数

\*/

**class** A

{

**private** **int** i;

**private** **static** **int** *cnt* = 0;

**public** A()

{

*cnt*++;

}

**public** A(**int** i)

{

**this**.i = i;

*cnt*++;

}

**public** **static** **int** getCnt()

{

**return** *cnt*;

}

}

**class** TestGetCnt

{

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

{

A aa1 = **new** A();

A aa2 = **new** A();

A aa3 = **new** A();

A aa4 = **new** A(5);//有参以及无参的创建对象，都可以计数

System.*out*.printf("%d\n", A.*getCnt*());

}

}

代码：

TestStaticOnlyOneObject.java

/\*

2013年2月7日22:28:41

一个类，只能生成一个对象。

封装方法

\*/

**class** A

{

**private** **static** A *aa* = **new** A();//10行

**private** A() //避免外部调用

{

}

**public** **static** A getA() // 必须是static， 否则外部无法调用

{

**return** *aa*;

}

}

/\*

一个类自己可以含有自己，第10行。类的属性可以是一个对象。

可以这样理解：这个类，包含一个属性i，和一个指向该类的指针aa。

类似链表，数据域，和指向结点的指针。

\*/

**public** **class** TestStaticOnlyOneObject

{

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

{

A aa = A.*getA*();

// A aa1 = new A();

// A aa2 = new A(2);

}

}

/\*

以上属于封装的一些知识。把类进行修饰，即添加条件。

\*/

# 6. 面向对象编程(下)

## 6.1 继承

### 继承的概念

一个新类从已有的类那里获得其已有的属性和方法，这种现象叫**类的继承**。

这个新类被称为**子类**，也叫**派生类**，已有的那个类叫做**父类**，也叫做**基类**。

继承的好处

1.代码得到极大的重用

2.形成一种类的层次体系结构

3.为多态创造条件

继承的实现方式

class SubClass extends SuperClass

{

……

}

利用继承可以较好的模拟出现实世界事物之间的联系

人

男人

女人

男学生

女学生

男大学生

男中学生

男小学生

图6-1-1 继承来模拟人这类事物

代码：

TestExtends\_1.java

**class** Human

{

//人笑

**public** **void** laugh()

{

System.*out*.println("笑!");

}

//人哭

**public** **void** cry()

{

System.*out*.println("哭!");

}

}

**class** Student **extends** Human

{

//做作业

**public** **void** doHomework()

{

System.*out*.println("做作业!");

}

}

**public** **class** TestExtends\_1

{

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

{

Student s = **new** Student();

s.laugh();

s.cry();

s.doHomework();

}

}

/\*

在JDK 1.6中的运行结果是：

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

笑!

哭!

做作业!

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

\*/

### 同包继承权限问题(重点)

子类的所有方法内部都可以访问父类除私有成员以外的所有成员，所谓子类的所有方法也包括子类的私有方法。

通过子类对象名可以访问

1.父类除私有成员外的所有成员

2.子类本身的除私有成员外的所有成员

附注：私有成员包括私有属性和私有方法

子类可以继承父类除私有成员以外的所有成员

**父类的私有成员不可以被子类继承，其他的成员都可以被子类继承**

不同包继承问题等讲到包时再说！

注意：其实， 在继承时， 父类的private成员已经被子类继承，只是程序员无法访问它而已。因此，继承必须慎重，否则会浪费内存!

代码：

TestExtends\_2.java

/\*

2013年2月6日20:41:40

同包继承权限问题

\*/

**class** A

{

**private** **int** s;

**int** m;

**public** **int** g;

**protected** **int** b;

**private** **void** fs()

{

}

**public** **void** fg()

{

}

**void** fm()

{

}

**protected** **void** fb()

{

}

}

**class** B **extends** A

{

//子类的所有方法内部都可以访问父类除私有成员以外的所有成员

**private** **void** g()

{

m = g = 10;

fg();

fm();

fb();

//s = 10; //error

//fs(); //error

}

}

**public** **class** TestExtends\_2

{

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

{

B bb = **new** B();

//通过子类对象名可以访问父类除私有成员外的所有成员

bb.g = bb.b = bb.m = 22;

//bb.s = 11; //error

//bb.fs(); //error

bb.fg();

bb.fm();

bb.fb();

System.*out*.println("2013年2月6日20:41:50");

}

}

### 继承的原则

何时选择继承？

一个很好的经验：“B是一个A吗？”

如果是则让B做A的子类

常犯的错误A有一个B吗？例如让汽车轮子称为汽车的子类是错误的

其实，继承可以理解为现实世界中的**一般到特殊**的一种关系。(**泛化**)

### 不同访问修饰符

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | public | protected | default | private |
| 同包同类 | √ | √ | √ | √ |
| 访问同包不同类型 | √ | √ | √ |  |
| 同包不同类继承 | √ | √ | √ |  |
| 不同包继承 | √ | √ |  |  |
| 访问不同包无任何关系的类 | √ |  |  |  |

表6-1-1

### 使用继承时应该注意

Java只支持单继承，不允许多重继承。(C++允许多继承，C#不允许多继承， 如果允许多继承，会出现**钻石环**的问题。)

单继承就是一个类只能有一个父类

多继承就是一个类可以有多个父类

可以有多层继承，即一个类可以继承其一个类的子类，如类B继承了类A，类C又可以继承类B，那么类C也间接继承了类A。

子类可以继承父类所有的成员变量和成员方法，但**子类永远无法继承父类的构造方法**。在子类的构造方法中可以使用语句super(参数列表)调用父类的构造方法。

### super的使用

调用父类的构造函数的语句必须借助于super(实参列表)，不能直接写父类的类名，这与C++不同

使用super函数时，必须保证位置在**构造函数**中的**第一条语句**。

每个子类**构造方法的第一条语句，都是隐含地调用super()**

如果显示的写出super();语句，则必须保证该语句是第一条语句，并且必须保证父类有无参的构造函数，否则会出错

即可以显示写super();前提是父类必须有无参的构造函数也可以显示写super(实参);前提是父类必须有带参的构造函数

一个子类的任何一个构造函数中都可只能出现一个super(实参列表)

代码：

TestSuper\_1.java

/\*

2013年2月6日20:39:40

super的使用

\*/

**class** A

{

A()

{

System.*out*.println("AAAA");

}

A(**int** i)

{

}

}

**class** B **extends** A

{

B()

{

**super**(2); //如果把该语句注释掉的，则编译器默认的是自动隐藏调用super(); 但如果父类没有无参的构造函数，则会报错

//一个子类的构造函数中只能出现一个 super(....)

System.*out*.println("BBBB");

}

}

**class** C **extends** B

{

C()

{

//int k = 10; //如果该语句生效 则会出错，因为会导致super()语句不是构造函数的第一条语句

**super**(); //35行 每个子类构造方法的第一条语句，都是隐含地调用super()，如果父类没有无参的构造函数，那么在编译的时候就会报错。

//super();语句可以写，也可以不写，不写的化，系统会自动调用的

//如果显示写出来的话，编译器要求该语句前面不能加任何语句，也就是说该语句必须保证是第一条语句

// super()也可以改为super(2); 但前提是父类必须有带一个参数的构造函数，否则也会报错

//如果把35行改为 super(2); 编译时就会报错!

System.*out*.println("CCCC");

}

}

**class** TestSuper\_1

{

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

{

C cc = **new** C();

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

}

}

/\*

在j2sdk1.4.2\_16中的运行结果是：

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

BBBB

CCCC

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

总结：

1、每个子类构造方法的第一条语句，都是隐含地调用super()，如果父类没有这种形式的

构造函数，那么在编译的时候就会报错。

2、如果显示的写出super();语句，则必须保证该语句是第一条语句，否则会出错

3、super();如果不写，则编译器会自动添加,所以此时如果父类没有无参的构造函数就会出错

4、既可以显示写super();前提是父类必须有无参的构造函数

也可以显示写super(实参); 前提是父类必须有带参的构造函数

5、调用父类的构造函数的语句必须借助于super,不能直接写父类的类名，这与C++不同，

6、一个构造函数中不能写多个super(参数列表)语句；原因可以用1、解释。

\*/

代码：

TestSuper\_2.java

class A

{

public int i;

A(int i)

{

this.i = i;

}

}

class B extends A

{

public B()

{

super(2); //如果将本语句注释掉 则会出错，因为 “派生类构造函数中如果不写super(...); 则默认的会添加super();”

}

}

class TestSuper\_2

{

public static void main(String[] args)

{

A aa = new A(5);

B bb = new B();

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

}

}

### 方法重写

方法重写

指在子类中重新定义父类中已有的方法

**重写方法必须和被重写方法具有相同的方法名称、参数列表和返回值类型**

**子类中不允许出现与父类同名同参但不同返回类型的方法，如果出现了，编译时会报错**

**覆盖方法时，不能使用比父亲中被覆盖的方法更严格的访问权限**(原因等我们讲到多态时再讲)

代码：

TestOverWrite\_1.java

**class** A

{

**public** **int** f(**int** i)

{

System.*out*.println("AAAA");

**return** 10;

}

}

**class** B **extends** A

{ // 把int改为long编译时会报错！

**public** **int** f(**int** i)

{

System.*out*.println("BBBB");

**return** 10;

}

}

**public** **class** TestOverWrite\_1

{

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

{

B bb = **new** B();

bb.f(10); // 编译器无法确定f方法是父类的还是父类

}

}

子类中不允许出现与父类同名参但不同返回类型的方法，如果出现了，编译时会报错

代码：

TestOverWrite\_2.java

**class** A {

**public** **void** f() {

}

}

**class** B **extends** A {

// 把public改为protected就会报错， 重写方法的权限不能低于被重写方法的访问权限

**public** **void** f() {

}

}

**public** **class** TestOverWrite\_2 {

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

}

}

重写方法时，不能使用比父类中被重写的方法更严格的访问权限

代码：

TestOverWrite\_3.java

**class** A

{

**public** **void** f()

{

System.*out*.printf("AAAA\n");

}

/\*重载A.f()方法\*/

**public** **void** f(**int** i)

{

System.*out*.printf("哈哈哈");

}

}

**class** B **extends** A

{

**public** **void** f()

{

**super**.f();//引用父类方法(非构造方法)

f(10);//此处可以引用父类中的方法(重载的那个)， 但是一定是有参的那个方法，此时有参的那个方法就相当于父类中的另一个与本方法(B.f())不同的一个方法，因为本方法(B.f())是无参的方法。

System.*out*.printf("BBBB\n");

}

}

**public** **class** TestOverWrite\_3

{

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

{

B bb = **new** B();

bb.f();

}

}

### 重写的意义

如果子类对继承自父类的方法不满意，就应该重写父类的方法

代码：

TestOverWrite\_4.java

**class** Person

{

**private** String name;

**private** **int** age;

**public** Person()

{

}

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

{

**this**.name = name;

**this**.age = age;

}

**public** **void** setName(String name)

{

**this**.name = name;

}

**public** **void** setAge(**int** age)

{

**this**.age = age;

}

**public** String getName()

{

**return** name;

}

**public** **int** getAge()

{

**return** age;

}

**public** String getInfor()

{

**return** "name: " + name + ", age: " + age;

}

}

**class** Student **extends** Person

{

**private** String school;

**public** Student()

{

}

**public** Student(String name, **int** age, String school)

{

**super**(name, age);

**this**.school = school;

}

**public** **void** setSchool(String school)

{

**this**.school = school;

}

**public** String getInfor()

{

String strInfor = "name: " + getName() + ", age:" + getAge() +

", school:" + school;

**return** strInfor;

}

}

**public** **class** TestOverWrite\_4

{

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

{

Person p1 = **new** Person("zhangsan", 30);

System.*out*.println(p1.getInfor());

Student s1 = **new** Student();

s1.setName("lisi");

s1.setAge(88);

s1.setSchool("ruide");

System.*out*.println(s1.getInfor());

}

}

## 6.2 多态

### 多态的概念

一个父类的引用类型变量它既可以指向父类对象也可以指向子类对象，它可以根据当前时刻指向的不同，自动调用不同对象的方法，这就是多态。

代码：

TestPoly.java

**class** A //父类

{

**public** **void** f()

{

System.*out*.printf("AAAA\n");

}

}

**class** B **extends** A // 子类

{

**public** **void** f()

{

System.*out*.printf("BBBB\n");

}

}

**public** **class** TestPoly

{

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

{

A aa = **new** A();

B bb = **new** B();

aa.f();

bb.f();

aa = bb; /\*把bb当作aa看了，就是说：如果aa是动物，bb是狗,

动物是狗的父类，狗是动物的子类。那么，

狗可以继承动物的所有属性(因为狗就是动物)，

所以狗可以看作动物。但是，动物不能看作狗。

子类可以当作父类看待。父类不可以当作子类来看待。

所以可以将狗赋给动物。但是，动物不可以赋给狗。\*/

//bb = aa; // error 把aa当作bb来看了

/\*在这里aa是父类的引用，bb是子类的引用。(这里说的引用就等价于C语言里的指针变量)

如果把子类的引用发送给父类的引用，然后通过父类的引用调用方法的时候，就相当于调用子类的方法\*/

aa.f();

/\*这就是多态，同一段代码，做不同的事情，原理是引用(指针)的指向发生了改变。

可以举这样的例子：给不同对象同一个刺激，但是，不同对象会发生不同的反应。

即：在一个学校的操场上，当铃声响起之后，同学们进入不同的教室。\*/

/\*看似一个小小的技巧，其实它的功能很强大！！！\*/

}

}

代码：

TestPoly\_1.java

**class** A {

**public** **void** f() {

System.*out*.println("AAAA");

}

}

**class** B **extends** A {

**public** **void** f() {

System.*out*.println("BBBB");

}

}

**public** **class** TestPoly\_1 {

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

A aa = **new** B();

aa.f();

aa = **new** A();

aa.f();

}

}

aa可以根据它自己当前时刻指向的是A类对象还是A子类对象，而自动决定调用哪个对象的f方法，这就是多态。

### 多态的优点

利用多态可以实现：

同一段代码做不同事情

如：

假设A派生出B，B派生出C

试着编写一个函数实现调用整个A类族所有对象f方法

所谓A类族就是A累及其A的子孙类所形成的一个族群

### 举例

代码：

TestPoly\_2.java

**class** A

{

**public** **void** f()

{

System.*out*.println("AAAA");

}

}

**class** B **extends** A

{

**public** **void** f()

{

System.*out*.println("BBBB");

}

}

**class** C **extends** B

{

**public** **void** f()

{

System.*out*.println("CCCC");

}

}

**public** **class** TestPoly\_2

{

//m函数可以实现调用整个A类族的所有对象f方法的功能

**public** **static** **void** m(A aa)

{

aa.f();

}

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

{

A aa = **new** A();

B bb = **new** B();

C cc = **new** C();

*m*(aa);

*m*(bb);

*m*(cc);

}

}

### 多态注意事项(难点)

**子类对象可以直接赋给父类引用，但父类对象在任何情况下都不可以直接赋给子类引用，**因为子类是父类的一种，但父类不可以是子类的一种，或者讲“子类可以当做父类看待，但父类不可以当做子类看待”，“狗可以当做动物看待，但动物不可以当做狗来看待”

通过父类引用只能访问子类对象从父类继承过来的成员

通过父类引用不能访问子类对象所有特有的成员

代码：

TestPoly\_3.java

**class** A

{

**public** **void** f()

{

System.*out*.printf("AAAA\n");

}

}

**class** B **extends** A

{

**public** **void** f()

{

System.*out*.printf("BBBB\n");

}

**public** **void** g() //g方法为子类B所特有的成员

{

System.*out*.printf("GGGG\n");

}

}

**public** **class** TestPoly\_3

{

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

{

A aa = **new** A();

B bb = **new** B();

aa = bb;

// aa.g(); // error 通过父类引用不能访问子类对象所有特有的成员

}

}

父类引用永远不可能直接赋给子类引用

1.**只有在父类引用本身指向的就是一个子类对象时，才可以把父类引用强制转化为子类引用**

2.其他情况下不允许把父类引用强制转化为子类引用，否则运行时会出错。

代码：

TestPoly\_4.java

/\*

2013年2月6日20:49:40

父类引用永远不可能直接赋给子类引用

注意：

只有在父类引用本身指向的就是一个子类对象时，才可以把父类引用强制转化为子类引用

其他情况下不允许把父类引用强制转化为子类引用

\*/

**class** A

{

}

**class** B **extends** A

{

}

**public** **class** TestPoly\_4

{

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

{

A aa = **new** A();

B bb = **new** B();

//bb = aa; //error

//bb = (B)aa; //24行 编译没有错误，但运行时出错！ 因为aa指向的是父类对象

A aa2 = **new** B();

//bb = aa2; //error 永远不可以把父类引用直接赋给子类引用

bb = (B)aa2; //OK 因为aa2 本身指向的就是一个B类对象 所以可以进行强制转化，注意与24行的区别 如果父类引用指向的是个子类对象，则可以通过强制类型转化把父类引用强制转化为子类引用，注意必须强制转化，在Java中无论如何绝对不可能直接把父类引用赋给子类引用的

}

}

代码：

TestPoly\_5.java

**class** A

{

**public** **void** f()

{

System.*out*.println("AAAA");

}

}

**class** B **extends** A

{

**public** **void** f()

{

System.*out*.println("BBBB");

}

**public** **void** g()

{

System.*out*.println("哈哈");

}

}

**public** **class** TestPoly\_5

{

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

{

A aa = **new** B();//等价于 A aa; aa = new B();

aa.f();

//aa.g(); //error

}

}

# 7 抽象类和接口

## 7.1 抽象类

### 概念

#### 抽象类的由来

利用抽象类是为了更好的对类加以分类，就如同人类不但给各种具体植物取了名字还发明了“植物”这个抽象名词对所有具体植物进行归类一样

Java用来模拟现实世界，所以也存在抽象类

抽象类通常用来作为具体一个类族的最顶层的父亲，用最底层的类表示现实中的具体事物，用最顶层的类表示类族所有事物的共性。

图7-1-1 抽象类的由来

生物

动物

植物

高级动物

低级动物

男人

女人

通常最上面的若干层类都是抽象的

### 抽象方法与抽象类

#### 抽象方法

在定义Java方法时可以只给出方法头，而不给出方法内部实现代码，这样的方法称为抽象方法。

凡是没有方法体的方法必须得使用关键字abstract修饰为抽象方法

凡是含有抽象方法的类都必须得声明为抽象类

#### 抽象类

用abstract关键字来修饰一个类时，该类叫做**抽象类**。包含抽象方法的类必须声明为抽象类。但是一个抽象类中却可以不包含任何抽象方法，尽管比较少见

抽象类不一定有抽象方法

有抽象方法的一定是抽象类

代码：

TestAbsClass\_1.java

**abstract** **class** A //如果f方法没有方法体，则必须的在class前加 abstract 如果f方法含有方法体，则class前面可加也可不加abstract ，因为“一个抽象类中是可以不包含任何抽象方法，尽管比较少见”

{

**private** **int** i;

**public** **abstract** **void** f(); //如果f方法没有方法体，则f方法必须的声明为抽象方法，即必须的在前面加abstract

}

**class** TestAbsClass\_1

{

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

{

System.*out*.println("今天我很高兴!");

}

}

不能new出抽象类对象，但可以定义一个抽象类的引用

我们可以把一个子类对象的地址赋给抽象类的引用，然后通过抽象类的引用调用子类从父类继承过来的方法，即**抽象类也可以实现多态**【重点】

假设A是抽象类，B是A的子类且完全实现了A的所有抽象方法，则

A aa = new A();//error

A aa = new B();//OK

代码：

TestAbsClass\_2.java

/\*

不能new 出抽象类对象，但可以用抽象类的引用指向抽象类的子类对象，即抽象类也可以实现多态

\*/

**abstract** **class** A

{

**abstract** **public** **void** f();

}

**class** B **extends** A

{

**public** **void** f()

{

System.*out*.println("BBB");

}

}

**public** **class** TestAbsClass\_2

{

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

{

//A aa = new A(); //error 不能new 出抽象类对象

A aa = **new** B(); //OK

aa.f(); //多态

}

}

### 抽象类实现多台的示例

代码：

TestAbs\_1.java

**abstract** **class** A

{

**public** **abstract** **void** f();

}

**class** B **extends** A

{

**public** **void** f()

{

System.*out*.printf("哈哈");

}

}

**public** **class** TestAbs\_1

{

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

{

//A aa1 = new A(); // error

A aa2 = **null**; // 可以定义一个抽象类的引用

aa2 = **new** B(); // 将子类对象的地址赋给抽象类的引用

aa2.f(); // 用抽象类的引用访问子类的方法， 这就是多态

}

}

## 7.2 Final关键字的使用

### Final

final可以修饰

1.整个类

2.类中的若干属性

3.类中的若干个方法

### Final修饰整个类

表示该类不能被继承

如果认为一个类已经很完美且不需要定义子类来继承它时，可以使用它

格式：

public final class A{…………}

public 和 final可以互换

### final修饰类表示类不能被继承

代码：

TestFinal\_1.java

**class** A //如果在class前面加final编译时程序就会报错

{

}

**class** B **extends** A

{

}

**public** **class** TestFinal\_1

{

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

{

}

}

### Final修饰类中的若干属性

**final 修饰类中的若干属性表示该属性必须被赋值并且只能被赋一次值(注意默认值不算真正的赋值)**

初始化方式有两种：(只能选择其中的一种)

1. 在定义成员变量的同时初始化

2. 在类中所有的构造函数中初始化

注意：

一个类的所有普通方法内部都不可以修改final修饰过的成员变量的值

代码：

TestFinalField.java

/\*

final类型修饰的属性必须在定义的同时初始化

如果final类型修饰的属性没有在定义的同时初始化，则必须在该类中的所有构造函数中完成初始化

不过这两种初始化只能存在一种，否则编译时会报错！因为final表示常量的意思，常变量当然不能被初始化两次了

\*/

**class** Point

{

**int** x;

**final** **double** PI = 1.9; //10 final定义的变量必须在定义的同时初始化 如果这里只定义不初始化则我们就必须在所有的构造函数中完成对final变量的初始化 不过这两种方式只能选择其中的一种

Point()

{

//PI = 1.234; //只要10行对PI进行了初始化，本语句就必须的被注释掉，否则编译时会报错!

}

Point(**int** m, **int** n)

{

//PI = 2.33; //只要10行对PI进行了初始化，本语句就必须的被注释掉，否则编译时会报错!

}

**void** output()

{

//PI = 2; //error 因为“一个类的所有方法内部都不可以修改final修饰过的成员变量的”

System.*out*.printf("%d %f\n", x, PI);

}

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

{

Point pt = **new** Point();

pt.output();

}

}

/\*

在JDK 1.6中的运行结果是：

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

0 1.900000

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

\*/

### Final修饰类中的若干个方法

表示该方法可以被子类继承，但不可以被子类重写

代码：

TestFinal\_2.java

**class** A

{

**public** **void** f() //如果public前面加final，则编译时就会报错

{

System.*out*.printf("AAAA\n");

}

}

**class** B **extends** A

{

**public** **void** f() // 重写父类的f方法

{

System.*out*.printf("BBBB\n");

}

}

**public** **class** TestFinal\_2

{

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

{

}

}

## 7.3 接口

### 接口(interface)

#### 接口的定义

就是抽象方法和常量值的集合。从本质上讲，接口是一种特殊的抽象类

#### 接口的格式

[public ] interface interfaceName [extends SuperInterfaceList]

{

……//常量定义和方法

}

接口中定义的属性必须是public static final，而接口中定义的方法则必须是public abstract 的，因此这些修饰符可以部分或全部省略

接口中定义的属性的值在实现类中不能被更改

代码：

TestInter\_1.java

/\*

接口中定义的属性的值在实现类中不能被更改

接口的属性是public static final 类型

\*/

//接口中是不可以定义构造函数的

**interface** It

{

**int** *i* = 10; //不能改为 int i;

}

**class** A **implements** It

{

**public** A(**int** j)

{

//this.i = j; // 接口It中的属性i 是public static final 类型，不可以在实现类中被改变

}

}

**class** TestInter\_1

{

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

{

A aa = **new** A(10);

System.*out*.println("aa.i = " + aa.*i*);

System.*out*.println("aa.i = " + A.*i*);

}

}

一个类只能实现某个接口，不能继承某个接口。但接口不但可以继承接口，而且可以继承多个接口，即接口允许多继承

一个文件的名字必须得和改文件名中public接口的名字一样

如果一个类只实现了一个接口的部分方法，则该类必须的声明为抽象类

一个类可以在继承一个父类的同时实现一个或多个接口，但extends关键字必须得在implements之前

代码：

TestInter\_2.java

**class** A

{

}

**interface** It1

{

}

**interface** It2 {

}

//接口可以多重继承，即一个接口可以有多个父接口，但是一个类只能有一个父类，Java类不允许多继承，接口却允许多继承

**interface** It3 **extends** It1, It2

{

}

**interface** It4

{

**int** *i* = 20;

}

//一个类可以在继承一个父类的同时实现一个或多个接口，但extends关键字必须的在implements 之前

**class** T **extends** A **implements** It4,It3

{

}

**class** TestInter\_2

{

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

{

System.*out*.println("嘿嘿!");

}

}

不可以new接口对象，但可以定义一个接口引用类型的变量并将其指向实现接口的对象，达到多态的目的！

### 实现的接口类(1)

代码：

TestInter\_5.java

**interface** It1

{

**void** f();

**void** g();

}

**abstract** **class** A **implements** It1 //去掉了abstract就会报错！

{

**public** **void** f()

{

System.*out*.printf("AAAA\n");

}

}

**public** **class** TestInter\_5

{

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

{

System.*out*.printf("哈哈\n");

}

}

如果一个类只实现了一个接口的部分方法，则该类必须得声明为抽象类

### 实现的接口类(2)

代码：

M.java

**interface** It

{

**void** f();

}

**class** A **implements** It

{

//public 不能丢。也不能改为其他修饰符

**public** **void** f()

{

System.*out*.printf("AAAA\n");

}

}

**class** M

{

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

{

//It it = new It(); // error

It it = **new** A();

it.f();

}

}

一个类要想实现某接口中的方法时，必须得在方法返回值前加上public

### 接口的多态

代码：

TestInter\_6.java

**interface** It

{

**void** f();

}

**class** A **implements** It

{

**public** **void** f()

{

System.*out*.printf("AAAA\n");

}

}

**class** TestInter\_6

{

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

{

//It tt = new It(); // error

It tt = **new** A();

tt.f();

}

}

不可以new接口对象，但可以定义一个接口引用类型的变量并将其指向实现接口的对象，达到多态的目的！

### 接口的作用

通过接口可以实现不相关类的相同行为

如：Java规定所有可以完成自我赋值功能的类都必须得实现java.lang.Colneable接口，但该接口却是空的，该接口中没有任何内容，目的只是起个标志作用

接口提供了不同对象进行协作的平台

如事件处理

接口可以实现多继承，从一定程序上弥补类只能单继承的缺陷

接口是我们了解一个类功能的重要途径

如：Java整个容器框架就是以接口的方式建立起来的，实现不同接口的类完成的是不同的功能，接口是我们了解一个类功能的重要途径

### 接口与抽象类区别

接口中的方法不允许有方法体，但抽象类却允许

Java不允许多继承，接口却允许多继承

接口可以实现多继承，即一个接口可以有多个父类，但Java类只允许但继承，即一个类只能有一个父亲

# 8 包

## 8.1 package 的使用

代码：

TestPackage.java

**package** zhangsan.lisi;

**public** **class** TestPackage

{

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

{

**new** A().print();

}

}

**class** A

{

**public** **void** print()

{

System.*out*.println("AAAA");

}

}

1.package语句必须得是第一条语句

2.package zhangsan.lisi表示：把该文件中所有的类放入zhangsan.lisi这个包中，并且该文件中所有的类真正名字将是包名和类名的组合。如：类TestPackage的名字将变成zhangsan.lisi.TestPackage，而不再是TestPackage

3.编译时建议用javac –d . TestPackage.java，尽量不要使用javac TestPackage.java。因为后者要自己手动建立包目录。如果不在当前路径下运行程序则必须保证class文件的最上层目录的父目录位于classpath下。

命令：javac –d . A.java

-d表示：自动生成包层

.表示：这个包层是在当前目录下建立

### java zhangsan.lisi.TestPackage 解析

先检测当前目录下是否有zhangsan/lisi这个包(包即文件夹)，如果有，再检测该包下是否有zhangsan.lisi.TestPackage这个类，如果没有，编译器将再去classpath设置的路径中依次查找。如果都查找失败，则运行时出错。

## 8.2 同包不同类的相互访问

代码：A.java

//A.java文件

**class** A

{

**public** **void** f()

{

System.*out*.println("AAAA");

}

}

代码：B.java

//B.java文件

**class** B

{

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

{

A aa = **new** A();

aa.f();

}

}

命令：

javac A.java B.java

java B

结果：

AAAAA

说明：**因为类A和类B默认是在同一个无名的包中所以彼此可以相互访问，只要是非私有成员都可以被同一个包的另一个类访问。**

## 8.3 不同包类的相互访问

代码：A.java

//文件名 A.java

**package** zhangsan.lisi;

**public** **class** A

{

**public** **void** ma()

{

System.*out*.println("AAAAA");

}

}

代码：B.java

//文件夹 B.java

**package** com.rulde;

**public** **class** B

{

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

{

zhangsan.lisi.A aa = **new** zhangsan.lisi.A();

aa.ma();

}

}

说明：单独编译时必须先编译A.java，后编译B.java，否则会报错，建议两个文件一起编译。

命令：

法一：

javac -d . A.java B.java或javac -d . B.java A.java

java com.rulde.B

法二：

javac -d . A.java

javac -d . B.java

java com.rulde.B

结果：

AAAAA

### 使用不同包中类的第一种方式

**package** com.rulde;

**public** **class** B

{

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

{

zhangsan.lisi.A aa = **new** zhangsan.lisi.A();

aa.ma();

}

}

使用一个类的全名

### 使用不同包中类的第二种方式

**package** com.rulde;

**import** zhangsan.lisi.\*;

**public** **class** B

{

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

{

zhangsan.lisi.A aa = **new** zhangsan.lisi.A();

aa.ma();

}

}

使用import语句导入一个包中所有的类

### 使用不同包中类的第三种方式

**package** com.rulde;

**import** zhangsan.lisi.\*;

**public** **class** B

{

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

{

A aa = **new** A();

aa.ma();

}

}

说明：使用import语句导入一个包中特定的类

注意：导入父包的类并不会自动导入子包的类。如：import zhangsan.\*;只会导入zhangsan这个包下的所有类，并不会导入zhangsan子包lisi这个包中的类

考虑一下：import java.awt.\*;和import java.awt.event.\*;的区别

## 8.4 归档工具jar

JAVA归档工具是JDK中提供的一种多用途的存档及压缩工具，可以将多个文件或目录合并压缩为单个的Java归档文件

jar文件的主要作用：

1.发布和使用类库

2.便于资源的组合和管理

### jar使用举例如何使用jar包中的类

**使用举例**

格式：

jar –cvf 要生成的包名.jar \*

举例：

jar –cvf c.jar \*

功能：把当前路径下所有的文件即文件夹下所有的内容打包成c.jar

jar –tf c.jar

功能：在dos下显示c.jar这个包解压后的文件内容

jar –xf d:\1\c.jar

功能：把d:\1\c.jar这个文件中的内容解压到当前目录下

**使用jar包中的类的方法**

假设现在有一个T.jar包，要想在任何目录下都可以访问T.jar包中的类，则设置classpath时，必须把包名T.jar也设置进去，因为T.jar也相当于一个目录。如：在d:\share\java下有一个T.jar，则classpath必须设置为d:\share\java\T，否则在非当前目录下是无法访问T.jar包中的类的。

## 8.5 不同的访问修饰符

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | public | protected | default | private |
| 同包同类 | √ | √ | √ | √ |
| 访问同包不同类型 | √ | √ | √ |  |
| 同包不同类继承 | √ | √ | √ |  |
| 不同包继承 | √ | √ |  |  |
| 访问不同包无任何关系的类 | √ |  |  |  |

表8-5-1

**不同访问修饰符(重点)**

**1.在同一个包中只有私有的不能被另一个类访问，也只有私有的不能被继承**

**2.在不同包没有任何关系的两个类，只有public类的public成员才可以被另一个包中的类访问**

**3.在不同包中有继承关系的两个类，只有public类的public成员和public类的protected成员可以被另一个包中的子类在内部使用，但是在子类的外部，通过子类对象名只能访问父类的public成员。**

代码：

A.java

**package** com.bao;

//可以把protected改为public，但却不能改为private，也不能在void前面什么都不写。

**public** **class** A

{

**protected** **void** f()

{

System.*out*.println("AAAA");

}

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

{

A aa = **new** A();

aa.f();

}

}

代码：

B.java

**package** zhangsan.lisi;

**import** com.bao.\*;

**class** B **extends** A

{

**public** **void** g()

{

f();//OK

System.*out*.println("GGGG\n");

}

}

**class** TestB

{

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

{

B bb = **new** B();

bb.g();//OK

//bb.f();//error

}

}

本程序证明了：

**在不同包中有继承关系的两个类，只有public类的public成员和public类的protected成员可以被另一个包中的子类在内部使用，但是在子类的外部，通过子类对象名只能访问父类的public成员。**

代码：

A.java

**package** com.rulde;

**public** **class** A //public既不能去掉，也不能改为protected，也不能改为private

{

**public** **void** f()

{

System.*out*.println("AAAA");

}

}

代码：

B.java

**package** zhangsan.lisi;

**import** com.rulde;

**public** **class** B

{

**public** **void** g()

{

A aa = **new** A();//只有com.rulde.A是public时，才可以定义com.rulde.A对象

aa.f();//只有com.rulde.A是public并且f方法是public时本语句 才会正确

}

}

**class** M

{

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

{

B bb = **new** B();

bb.g();

}

}

本程序证明了：**如果一个类A是public，但该类的方法成员和域成员是非public，则我们仍然可以在其他包中的类中定义类A的对象aa，但却无法通过类对象aa调用aa中的非public方法成员和非public域成员。**

# 9 异常

## 9.1为什么需要异常

示例：编程实现把键盘输入的数字付给整形变量

代码：

TestInput\_1.java

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

**public** **class** TestInput\_1

{

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

{

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

**int** i = sc.nextInt();

System.*out*.println("i = " + i);

}

}

输入12 结果如下：



图9-1-1

如果输入的不是合法数字，则结果如下：

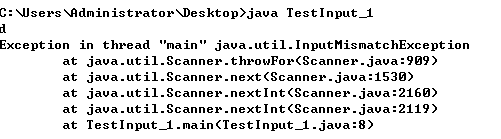


图9-1-2

读写错误的异常的处理

代码：

TestInput.java

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

**public** **class** TestInput

{

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

{

Scanner sc = **null**;

**try**

{

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

**int** i = sc.nextInt();

System.*out*.println(i);

}

**catch**(InputMismatchException e)

{

System.*out*.println("读取错误，程序终止");

}

}

}

本程序出现的问题是无法通过逻辑判断来解决的，Java提供的异常处理机制可以很好的解决这个问题。

## 9.2 异常的处理机制

什么事异常

异常(Exception)是程序运行过程中发生的事件，该事件可以中断程序指令的正常执行流程。

代码：

Excep\_1.java

**class** A

{

**int** divide(**int** a, **int** b)

{

**return** a / b;

}

**int** add(**int** a, **int** b)

{

**return** a + b;

}

}

**class** Excep\_1

{

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

{

A aa = **new** A();

**try**

{

**int** i = aa.divide(3, 0);

System.*out*.println(i);

}

**catch** (ArithmeticException e)

{

System.*out*.println("除数不能为零!");

}

**int** j = aa.add(2, 4);

System.*out*.println("j = " + j);

}

}

异常的处理机制(重点)

**1. 当Java程序运行时出现问题时，系统会自动检测到该错误，并立即生成一个与该错误对应的异常现象。**

**2. 然后把该异常对象提交给Java虚拟机**

**3. Java虚拟机会自动寻找相应的处理代码来处理这个异常，如果没有找到，则程序终止**

**4.程序员可以自己编写代码来捕捉可能出现的异常，并编写代码来处理相应的异常。**

## 9.3 常见的一些异常

### 空指针异常

代码：

TestNullPointerException.java

**class** Person

{

**public** **int** age;

}

**public** **class** TestNullPointerException

{

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

{

Person p = **null**;

System.out.println(p.age);

}

}

执行结果：

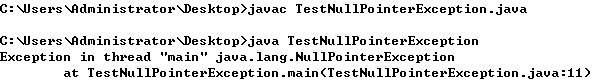


图9-3-1

### 下表越界异常

代码：

TestIndexOutOf.java

**public** **class** TestIndexOutOf

{

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

{

String friends[] = {"Lisa", "Bily", "Kessy"};

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

{

System.*out*.println(friends[i]);

}

System.*out*.println("\nThis is the end");

}

}

执行结果：

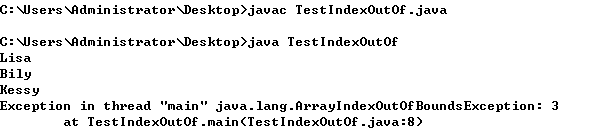


图9-3-2

### 算术异常

代码：

TestArithExcep.java

**class** A

{

**int** divide(**int** a, **int** b)

{

**return** a / b;

}

}

**public** **class** TestArithExcep

{

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

{

A aa = **new** A();

**int** i = aa.divide(3, 0);

System.*out*.println(i);

}

}

执行结果：

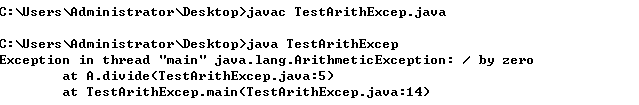


图9-3-3

### 异常的分类

Throwable

Error

Exception

RuntimeException

图9-3-4

**1.Error是系统错误，程序员无法处理这些异常**

**2.Exception是程序员可以捕获并处理的异常**

**3.RuntimeException的子类异常是可以处理也可以不处理的异常**

**4.凡是继承自Exception但又不是RuntimeException子类的异常我们都必须得捕获并进行处理。**

Error：

由Java虚拟机生成并抛出，包括动态链接失败、虚拟机错误等，Java程序无法对此错误进行处理。

RuntimeException：

Java虚拟机在运行时声称的异常，如被0除等系统错误、数组下标超出范围等，其产生比较频繁，处理麻烦，对程序可读性和运行效率影响太大。因此由系统检测，用户可以不作处理，系统将它们交给缺省的异常处理程序（当然，必要时，用户可以对其处理）。

Exception：

一般程序中可以预知的问题，其产生的异常可能会带来意想不到的结果，因此Java编译器要求Java程序必须捕获或声明所有的非运行时异常。

## 9.4 异常处理步骤

### 异常处理步骤

try

{

可能出现异常的代码块

}

catch(ExceptionName1 e)

{

当产生ExceptionName1异常时的处理措施

}

catch(ExceptionName2 e)

{

当产生ExceptionName2异常时的处理措施

}

……

finally

{

无论是否捕捉到异常都必须处理的代码

}

### 捕捉处理异常长注意问题

→捕捉SomeException2时：

try{

语句1;

语句2;

}

catch(SomeException1 e)

{……}

catch(SomeException2 e)

{……}

finally{……}

后面的语句;

图9-4-1

一旦语句1抛出异常，则语句2永远都得不到执行

→没有捕捉到异常时：

try{

语句1;

语句2;

}

catch (SomeException1 e)

{……}

catch (SomeException2 e)

{……}

finally{……}

后面的语句;

图9-4-2

无论try中代码是否抛出异常，finally中代码都会得到执行

### finally的作用

无论try所指定的程序块中是否抛出异常，也无论catch语句的异常类型是否与所抛弃的异常的类型一致，finally中的代码一定会得到执行

finally语句为异常处理提供一个统一的出口，使得在控制流程转到程序的其他部分以前，能够对程序的状态做统一的管理

通常在finally语句中可以进行资源的清除工作，如关闭打开的文件、删除临时文件等

代码：

ChatServer.java

**import** java.awt.\*;

**import** java.awt.event.\*;

**import** java.net.\*;

**import** java.io.\*;

**public** **class** ChatServer

{

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

{

**boolean** started = **false**; // 判断服务器是否启动了，先假定没有启动

ServerSocket ss = **null**;

DataInputStream dis = **null**;

**try**

{

ss = **new** ServerSocket(8888);

started = **true**; // 服务器已经启动了，设置started为true

**while** (started)

{ //当服务器端启动时。。。。

**boolean** bConnect = **false**; //判断服务器和客户端的连接是否已经建立， 没有为false，连接成功返回true

Socket s = ss.accept(); //accept()与readUTF()方法一样，都是一个阻塞式方法，如果没有收到连接请求，则一直等待。。。。。

bConnect = **true**; //连接成功，设置bConnect为true

System.out.println("一个连接已经建立!"); // --------------------------

dis = **new** DataInputStream(s.getInputStream());

**while** (bConnect)

{

String str = **null**;

str = dis.readUTF(); //如果客户端突然断开连接,该语句就会抛出异常EOFException，所以我们必须得对dis.readUTF();进行异常处理

//readUTF()是阻塞式方法，如果得不到数据，则继续等待，直到读取到数据为止

System.out.println("从客户端接受的数据是:" + str);

}

}

}

**catch** (BindException e)

{

System.out.println("端口已被占用，其使用其他端口");

System.out.println("请关闭相关程序，重新运行!");

System.exit(-1);

}

**catch** (EOFException e)

{

System.out.println("客户端已经断开连接!");

}

**catch** (IOException e)

{

//e.printStackTrace();

System.out.println("服务器端读写错误!");

}

**finally**

{

**try**

{

**if** (**null** != ss) //如果监听程序打开了，则关闭网络监听程序

{

ss.close();

ss = **null**;

}

**if** (**null** != dis) //如果输入流打开了，则关闭输入流

{

dis.close();

dis = **null**;

}

}

**catch** (Exception e)

{

}

}

}

}

### throw

throw用来抛出异常

格式：

throw new 异常名(参数);

假设f方法抛出了A异常，则f方法有两种方式来处理A异常

1.throws A

谁调用f方法，谁处理A异常，f方法本身不处理A异常

2.try{……} catch(){……}

f方法本身自己来处理A异常

要抛出才异常必须得是Throwable的子类

要抛出的异常必须得是throwable的子类的实例：

代码：

TestExcep.java

//去掉了extends Throwable程序就会报错

**class** A **extends** Throwable

{

}

**class** M

{

**public** **void** f() **throws** A

{

**throw** **new** A();//要抛出的异常必须得是Throwable的子类

}

}

**public** **class** TestExcep

{

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

{

}

}

### throws

void f() throws A

{

……

……

}

throws A 表示调用f方法时f方法可能会抛出A类异常，建议您调用f方法时最好对f方法可能抛出的A类异常进行捕捉

**throws A不表示f方法一定会抛出A类异常**

throws A，f方法也可以不抛出A类异常

**throws A不表示调用f方法时，必须得对A异常进行捕捉**

假设A是RuntimeException子类异常

由于RuntimeException子类异常可以处理也可以不处理，所以编译器允许你调用f方法时，对f方法抛出的RuntimeException子类异常不进行处理

强烈建议您

**对throws出的所有异常进行处理**

**如果一个方法内部已经对A异常进行了处理，则就不要再throws A**

示例：

代码：

M.java

**class** ER **extends** RuntimeException

{

}

**class** A

{

**public** **void** f() **throws** ER

{

System.out.println("AAAAA");

}

}

**class** M

{

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

{

A aa = **new** A();

aa.f();

}

}

执行结果：

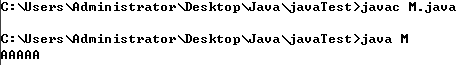


图9-4-5

本程序证明了：一个f方法throws A f方法可以不抛出A异常，调用f方法的方法也可以不处理A类异常

### 注意的问题

所有的catch只能有一个被执行

有可能所有的catch都没有执行

**先catch子类异常在catch父类异常**

如果先catch父类异常在catch子类异常，则编译时会报错

**重写方法抛出异常的范围不能大于重写方法排除的异常范围**

代码：

TestExtendExcep.java

/\*

2013年4月24日19:44:40

子类覆盖了基类方法时，

子类方法抛出异常的范围不能大于基类方法抛出的异常范围

子类方法可以不抛出异常，也可以只抛出基类方法的部分异常

但不可以抛出基类方法以外的异常

\*/

//自定义异常A

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

{

}

//自定义异常B

**class** B **extends** Exception

{

}

//自定义异常C

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

{

}

**class** M

{

**void** f() **throws** A, B

{

}

}

**class** N **extends** M

{

**void** f() **throws** A,B //可以throws A或B,也可以throws A,B 也可以不throws，但不可以throws C 即"子类覆盖了基类方法时，子类方法抛出异常的范围不能大于基类方法抛出的异常范围"

{

}

}

**class** Test

{

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

{

M m = **new** M();

N n = **new** N();

System.out.println("1111");

}

}

/\*

在JDK 1.6中的运行结果是：

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

1111

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

\*/

代码：

TestTryCatch.java

/\*

2013年4月24日19:46:42

先catch子类异常再catch父类异常

如果先catch父类异常再catch子类异常，则编译时会报错

\*/

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

{

}

**class** B **extends** A

{

}

**class** C **extends** B

{

}

**class** M

{

**public** **void** compare(**int** i, **int** j) **throws** A,B

{

**if** (i > j)

**throw** **new** A();

**else**

**throw** **new** B();

}

}

**public** **class** TestTryCatch

{

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

{

M mm = **new** M();

**try**

{

mm.compare(-4, 1);

}

**catch** (B bb)

{

System.out.println("左边不能小于右边");

}

**catch** (A aa)

{

System.out.println("左边不能大于右边");

}

}

}

## 9.5 自定义异常

public Exception(String message)含义

异常与重写的关系

代码：

**import** java.io.\*;

**class** DivisorIsZeroException **extends** Exception // DivisorIsZeroException这个异常类是自己定义的，意思是除数为零异常

{

**public** DivisorIsZeroException(String name)//写一个构造方法

{

**super**(name);

}

}

**class** A

{

**public** **int** divide(**int** a, **int** b) **throws** DivisorIsZeroException//必须对这个异常进行补货处理以便抛出

{

**int** m = 0;

**if** (0 == b)

{

**throw** **new** DivisorIsZeroException("除数不能为零！");//异常信息为："除数不能为零！"

}

**else**

{

m = a / b;

}

**return** m;

}

}

**public** **class** UserDefineTestExcep

{

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

{

A aa = **new** A();

**try**

{

aa.divide(6, 0);

}

**catch**(Exception e)

{

e.printStackTrace();

}

}

}

/\*

自定义异常处理

\*/

## 9.6 异常的优缺点

### 异常的优点

没有错误处理的程序：

{

openTheFile;

determine its size;

allocate that much memory;

read-file;

closeTheFile;

}

以常规方法处理错误：

openFiles;

if (theFilesOpen)

{

determine the length of the file;

if (gotTheFileLength)

{

allocate that much memory;

if (gotEnoughMemory)

{

read the file into memory;

if (readFailed) errorCode = -1;

else errorCode = -2;

}

else errorCode = -3;

}

else errorCode = -4;

}

else errorCode = -5;

以常规方法处理错误存在的问题

观察前面的程序，你会发现大部分精力花在出错处理上了，只把能够想到的错误考虑到，对以外的情况无法处理，程序的可读性差，打量的错误处理代码混杂在程序中，出错返回信息量太少，无法更确切的了解错误状况或原因

用异常的形式处理错误：

{

try

{

openTheFile;

determine its size;

allocate that much memory;

read-File;

closeTheFile;

}

catch (fileopenFailed)

{

dosomething;

}

catch (sizeDetermineFailed)

{

dosomething;

}

catch (memoryAllocateFailed)

{

dosomething;

}

catch (readFailed)

{

dosomething;

}

catch (fileCloseFailed)

{

dosomething;

}

finally

{

dosomething;

}

}

### 优点

强制程序员考虑程序的安全性与健壮性

增强了程序员对程序的可控性

有利于代码的调试

把错误代码处理从常规代码中分离出来

### 注意

异常并不一定能够使程序的逻辑更清晰

因为有时我们必须得编写代码捕捉异常，所以可能会导致程序的逻辑非常混乱

异常并不能解决所有问题

# 10. 字符串的使用

## 10.1 toString方法

### System.out.println(类对象名);示例1

代码：TestToString\_1.java

**class** A

{

}

**public** **class** TestToString\_1{

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

{

A aa = **new** A();

System.out.println(aa);

}

}

执行结果：



图10-1-1

代码：TestToString\_2.java

**class** A{

**public** String toString(){

**return** "哈哈";

}

}

**public** **class** TestToString\_2{

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

A aa = **new** A();

System.*out*.println(aa);

}

}

执行结果：



图10-1-2

### System.out.println(类对象名);示例2

代码：TestToString\_3.java

**class** Point{

**private** **int** i, j;

**public** Point(**int** i, **int** j)

{

**this**.i = i;

**this**.j = j;

}

//如果将9行到12行的代码注释掉，则输出结果解释乱码

**public** String toString()//9行

{

**return** "[" + i + "，" + j + "]";

}//12行

}

**public** **class** TestToString\_3{

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

Point p1 = **new** Point(1, 1);

Point p2 = **new** Point(2, 2);

System.out.println(p1);

System.out.println(p2);

}

}

执行结果：

如果没有注释掉9行到12行的代码结果如下：



图10-1-3

如果注释掉9行到12行的代码结果如下：



图10-1-4

### toString方法总结

所有的类都默认自动继承了Object类

Object类中的toString方法返回的是类的名字和该对象**哈希码**组成的一个字符串

System.out.println(类对象名);

实际输出的是该对象的toString()方法所返回的字符串

为了实际需要，建议子类重写父类Object继承toString方法

## 10.2 String 与StringBuffer

### Object类的equals方法

**所有类都从Object类中继承了equals方法**

Object类中的equals方法源代码如下：

public Boolean equals(Object obj)

{

return this == obj;

}

Object类中equals方法是直接判断this和obj本身的值是否相等，即用来判断调用equals的对象和形参obj所引用的对象是否是同一对象，所谓同一对象就是指内存中同一块存储单元，如果this和obj指向的是同一块内存对象，则返回true，如果this和obj指向的不是同一块内存，则返回false

**如果是同一块内存，则Object中的equals方法返回true，如果是不同的内存，则返回false**

### 何时需要重写equals方法

用一个类构造出来的不同内存的两个对象，如果内存中的值相等，我们一般情况下也应该认为这两个对象相等，很明显Object中的equals()无法完成这样的重任，Object中的equals()方法只有在两个对象是同一块内存时，才返回true，这时候我们就有必要重写父类Object中的equals方法

代码：

TestObjectEquals.java

**class** A

{

**public** **int** i = 10;

}

**public** **class** TestObjectEquals

{

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

{

A aa1 = **new** A();

A aa2 = **new** A();

**if** (aa1.equals(aa2))

{

System.*out*.println("aa1 == aa2");

}

**else**

{

System.*out*.println("aa1 != aa2"); //aa1和aa2分配的是两块不同的内存，但这两块内存的值都是一样的，尽管如此，还是返回false，这从本语句会输出，既可以判断出

}

aa1 = aa2;

**if** (aa1.equals(aa2))

{

System.*out*.println("aa1 == aa2");

}

**else**

{

System.*out*.println("aa1 != aa2");

}

}

}

/\*

在JDK 1.6中的运行结果是：

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

aa1 != aa2

aa1 == aa2

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

总结:

Object中的public boolean equals(Object obj);

即Object中的equals方法是直接判断this和obj本身的值是否相等，即用来判断调用

equals的对象和形参obj所引用的对象是否是同一对象，所谓同一对象就是指是内存中同

一块存储单元

Object中的equals方法功能：如果this和obj指向的是同一块内存对象，则返回true，

如果this和obj指向的不是同一块内存，则返回false,注意：即便是内容完全相等的两块

不同的内存对象，也会返回false，本程序就是这样的例子，aa1和aa2占用不同的内存，

尽管aa1和aa2所占不同内存的内容都是一样的，照样返回false

很明显，这样设计是不太合理的，假设现在有两个A类对象aa1 和 aa2，它们分别

占用不同的内存，但是它们的内容都是一样的， 很多情况下，我们也应该认为aa1和aa2

是相等的， 即：用同一个类构造的两个占用不同内存的对象，如果这两个对象虽然占用

不同的内存但是内存中的内值是一样的，则我们也应该认为这两个对象是相等的：

即同类型不同内存的两个对象，如果内存中的值相等，我们一般应该也认为这两个对象

相等，很明显Object中的equals()无法完成这样的重任，Object中的equals()方法只有

在两个对象是同一块内存时，才返回true。

所以我们自己定义对象时，如果要使用equals方法，则通常都需要重写超父类的

Object中的equals方法，

public boolean equals(Object obj)

{

return this == obj;

}

\*/

代码：

Test.java

**class** A

{

**public** **int** i;

**public** A(**int** i)

{

**this**.i = i;

}

}

**public** **class** Test

{

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

{

A aa1 = **new** A(1);

A aa2 = **new** A(1);

System.out.println(aa1 == aa2);

System.out.println(aa1.equals(aa2));

}

}

执行结果：



图10-2-1

### String类

java.lang.String类对象表示不可修改的Unicode编码字符串。

在Java中双引号括起来的字符串也被当作String对象

System.out.println(“abc”.length());//输出3

System.out.println(“abc”.equals(“abc”));//输出true

String类的equals方法

String类已经重写了Object中的equals方法

例子

假设：

假设str1和str2都是String对象

str1.equals(str2);是用来比较str1变量本身所占内存的值所指向的对象和str2变量本身所占内存的值所指向的对象的内存是否相等，如果想等则返回true，否则返回false

String类的equals方法是用来判断两个对象的内容是否相等的，Object类的equals方法是用来判断两个对象是否是同一个对象

一定要注意 == 和 equals的区别

代码：

TestString\_1.java

/\*

== 与 equals()区别

\*/

**public** **class** TestString\_1

{

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

{

String str1 = "hello";

String str2 = "hello"; //str1 和 str2 都指向了匿名对象"hello"

**if** (str1 == str2) //判断str1 和 str2本身的内容是否相等

{

System.*out*.println("str1 == str2");

}

**else**

{

System.*out*.println("str1 != str2");

}

String str3 = **new** String("hello"); //str3 和 str4 很明显指向的是不同的对象

String str4 = **new** String("hello");

**if** (str3 == str4)

{

System.*out*.println("str3 == str4");

}

**else**

{

System.*out*.println("str3 != str4");

}

**if** (str3.equals(str4)) //判断str3 与 str4所引用的对象是否相等 很明显是true

{

System.*out*.println("str3.equals(str4) == true");

}

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

}

}

/\*

在JDK 1.6中的运行结果是：

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

str1 == str2

str3 != str4

str3.equals(str4) == true

Hello World!

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

总结：

str1 == str2

是用来比较str1变量本身所占内存的值和str2变量本身所占内存的值

是否相等

str1.equals(str2)

是用来比较str1变量本身所占内存的值所指向的对象和str2变量本身

所占内存 的值所指向的对象的内容是否相等

也就是说如果str1和str2占用不同的内存，但是这两个不同内存的值是一样的，

Stirng.equals()方法也会返回true.但是Object中的equals()方法却是返回

false,具体可参见“【测试Object类中的Equals方法】TestObjectEquals.java”

Object中就含有equals()方法，不过String类重写了Object中的equals()方法

API：

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

equals

public boolean equals(Object anObject)将此字符串与指定的对象比较。当且仅当该参数不为 null，并且是与此对象表示相同字符序列的 String 对象时，结果才为 true。

覆盖：

类 Object 中的 equals

参数：

anObject - 与此 String 进行比较的对象。

返回：

如果给定对象表示的 String 与此 String 相等，则返回 true；否则返回 false。

另请参见：

compareTo(String), equalsIgnoreCase(String)

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

\*/

代码：

TestString\_2.java

**public** **class** TestString\_2

{

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

{

String str1 = **new** String("China");

String str2 = **new** String("China");

System.out.println(str1.equals(str2)); // 输出true

System.out.println(str1 == str2); // 输出false

String str3 = "China";

String str4 = "China";

System.out.println(str3.equals(str4)); // 输出true

System.out.println(str3 == str4); // 输出true

}

}

执行结果：

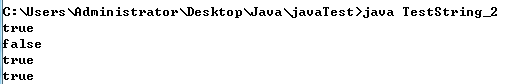


图10-2-2

### String类示例内存示意图

str1

str1

str1

str1

“China”

“China”

“China”

堆区

栈区

数据区

图2-2-3

### String类常用方法

**public** **char** charAt(**int** index)//返回字符串中的第index个字符

**public** **int** length()//返回字符串的长度

**public** **int** indexof(String str)//返回字符串中出现str的第一个位置

**public** **int** indexof(String str, **int** fromIndex)//返回字符串中从fromIndex开始出现str的第一个位置

**public** **boolean** equalsIgnoreCase(String anther)//比较字符串与anther是否一样（忽略大小写）

**public** String replace(**char** oldChar, **char** newChar)//在字符串中用newChar字符串替换oldChar字符

**public** **boolean** starsWith(String prefix)//判断字符串是否以prefix字符串开头

**public** **boolean** endsWith(String suffix)//判断字符串是否已suffix结尾

**public** String toUpperCase()//返回一个字符串为该字符串的大写形式

**public** String toLowerCase()//返回一个字符串为该字符串的小写形式

**public** String substring(**int** beginIndex)//返回该字符串从beginIndex开始到结尾的子字符串

**public** String substring(**int** beginIndex, **int** endsIndex)//返回该字符串从beginIndex开始到endIndex结尾的子字符串

静态重载方法

public static String valueOf(…)可以将基本类型数据转换为字符串：

例如：

public static String valueOf(double d)

public static String valueOf(int i)

… … …

方法public String[] split(String regex)可以将一个字符串按照指定的分隔符分隔，返回分隔后的字符串数组。

### String举例

举例一：

代码：

TestString\_3.java

**public** **class** TestString\_3 {

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

String s1 = "sun java", s2 = "Sun Java";

System.*out*.println(s1.charAt(1));//u

System.*out*.println(s2.length());//8

System.*out*.println(s1.indexOf("java"));//4

System.*out*.println(s1.indexOf("Java"));//-1

System.*out*.println(s1.equals(s2));//false

System.*out*.println(s1.equalsIgnoreCase(s2));//true

String s = "我是程序员，我在学习Java";

String sr = s.replace('我', '你');

System.*out*.println(sr);//你是程序员，你在学习Java

}

}

执行结果：



图2-2-4

举例二：

代码：

TestString\_4.java

**public** **class** TestString\_4

{

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

{

String s = "Welcome to Java World!";

String s1 = " sun java ";

System.*out*.println(s.startsWith("Welcome"));//true

System.*out*.println(s.endsWith("World"));//false

String sL = s.toLowerCase();

String sU = s.toUpperCase();

System.*out*.println(sL);//welcome to java world!

System.*out*.println(sU);//WELCOME TO JAVA SORLD!

String subS = s.substring(11);

System.*out*.println(subS);//Java World!

String sp = s1.trim();

System.*out*.println(sp);//sun java

}

}

执行结果：

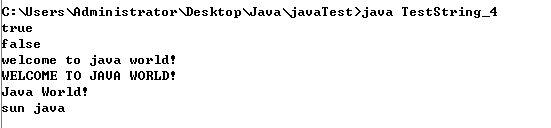


图2-2-5

举例三：

代码：

TestString\_5.java

**public** **class** TestString\_5

{

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

{

**int** j = 1234567;

String sNumber = String.valueOf(j);

System.out.println("j是" + sNumber.length() + "位数。");

String s = "Marry,F,1976";

String[] sPlit = s.split(",");

**for** (**int** i = 0; i < sPlit.length; i++)

{

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

}

}

}

执行结果：



图2-2-6

练习：

统计一个String对象中大写字母 小写字母 非字母 各自出现的个数

统计一个字符串在另一个字符串中出现的次数

## 10.3 StringBuffer类

### StringBuffer类的由来

String类对象一旦创建就不可以更改

如果经常对字符串内容进行修改，则使用StringBuffer。

如果经常对字符串内容进行修改而使用String的话，就会导致既耗时间又耗空间！

例子：

String s1 = “abasdmasc”;

String str2 = “123”;

String s1 = s1 + s2;

删除str1中的字母d

StringBuffer对象的内容是可以改变的

因此String类中没有修改字符串的方法，但是StringBuffer类中却有大量修改字符串的方法。

代码：

TestStringBuffer\_1.java

/\*

本程序说明了: StringBuffer 类的重要性

String s1 = "zhangsan";

只使用String类来删除s1对象中的第一个n这个字母的方法

\*/

**public** **class** TestStringBuffer\_1

{

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

{

String s1 = "zhangsan";

String s2 = s1.substring(0, 3);

String s3 = s1.substring(4, s1.length());

s1 = s2 + s3;

System.out.println("s1 = " + s1);

}

}

/\*

在JDK 1.6中的运行结果是：

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

s1 = zhagsan

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

\*/

执行结果：



图10-3-1

### StringBuffer类的构造函数

**public** StringBuffer()

创建一个空的没有任何字符的StringBuffer对象

**public** StringBuffer(**int** capacity)

创建一个不带字符，但具有指定初始容量的字符串缓冲区。

**public** StringBuffer(String str)

创建一个StringBuffer对象， 包含与str对象相同的字符序列

### StringBuffer常用方法

重载方法public StringBuffer append(…)可以为该StringBuffer对象添加字符序列，返回添加后的该StringBuffer对象引用，例如：

**public** StringBuffer append(String str)

**public** StringBuffer append(StringBuffer sbuf)

**public** StringBuffer append(**char**[] str)

**public** StringBuffer append(**char**[] str, **int** offset, **int** len)

**public** StringBuffer append(**double** d)

**public** StringBuffer append(Object obj)

… … …

重载方法

public StringBuffer insert(…)可以为该StringBuffer对象在指定位置插入序列，返回修改后的该StringBuffer对象引用，例如：

**public** StringBuffer insert(**int** offset, String str)

**public** StringBuffer insert(**int** offset, **double** d)

… … …

方法public StringBuffer delete(int start, int end)可以删除从start开始到end-1位置的一段字符序列，返回修改后的该StringBuffer对象引用

### StringBuffer举例

代码：

TestStringBuffer\_2.java

**public** **class** TestStringBuffer\_2

{

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

{

StringBuffer sb = **new** StringBuffer();

sb.append("abc");

sb.append("123");

System.*out*.println("sb = " + sb); // sb = abc123

sb.insert(3, "--");

System.*out*.println("sb = " + sb); // sb = abc--123

sb.delete(2, 6);// 把下标从2开始到6-1结束的字符删除

System.*out*.println("sb = " + sb); // sb = ab23

sb.reverse();

System.*out*.println("sb = " + sb); // sb = 32ba

String str = sb.toString();

System.*out*.println("str = " + str); // str = 32ba

}

}

执行结果：

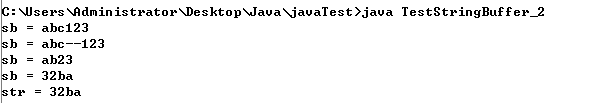


图10-3-2

# 11. 线程

## 11.1 刚要

### 线程的定义

### 如何创建一个线程

### 线程的控制

优先级控制

线程的休眠、让步、挂起和修复

线程的串行化

线程生命周期控制

### 线程同步

线程的死锁

例子：

买票

生产和消费问题

## 11.2 初学者要注意四个问题

什么事进程（尽量理解）

为什么需要进程（理解）

什么是线程（必须掌握）

为什么需要线程（不知道也没关系）

## 11.3什么叫程序

所谓“程序”，是一个严格有序的指令集合。程序规定了完成某一任务时，计算机需做的各种操作，以及这些操作的执行顺序。

## 11.4 单道程序设计环境中程序特点

单道程序设计环境指：计算机中除了操作系统之外， 只存在一个用户程序，即用户程序独享整个计算机资源

单道程序有如下特点：

1.资源的独占性：任何时候，位于内存中的程序可以使用系统中的一切资源，不可能有其他程序与之竞争。

2.执行的顺序性：内存中每次只有一个程序，各个程序是按次序执行的，即做完一个在做下一个。

3.结果的再现性：只要执行环境和初始条件相同，重复执行一个程序，获得的结果总是一样的。

## 11.5 多道程序设计环境中程序特点

多道程序设计是指：计算机中除了操作系统之外，存在多个用户程序，这些程序同时运行

多道程序设计有如下特点：

1.间断性：由于资源共享和合作，并发程序间相互制约，造成合作执行间断。

2.失去封闭性：程序执行受外界影响。

3.不可再现性：重复执行时，可能得到不同的结果。

## 11.6 进程的由来

一方面为了保持程序“是一个在时间上严格有序的指令集合，是静态的保存在存储介质上”这个概念的缘由含义，另一方面为了刻画多个程序共同运行时呈现出的这些特征，在操作系统中，以“程序”为基础，有引入了“进程”这一新的概念！

通俗点说，为了不破坏“程序”这个词原有的含义，而又能刻画多个程序共同运行时呈现出的新特征，所以引入了进程这一概念，按照大多数教材的说法：“为了使程序能并发执行，且为了对并发执行的程序加以描述，所以人们引入了进程”

## 11.7 什么是进程

至今没有统一的说法。

程序只是一组指令的有序集合，它本身没有任何运行的含义，它只是一个静态的实体

进程是程序在某个数据集上的执行。

进程是一个动态的实体，它有自己的生命周期。它因创建而产生，因调度而运行，因等待资源或事件而被处于等待状态，因完成任务而被撤销。

## 11.8 线程的定义

线程是一个程序里的不同执行路径

以前所编写的程序，每个程序都有一个入口、一个出口以及一个顺序执行的序列，在程序执行过程中的任何指定时刻，都只有一个单独的执行点。

事实上，在单个程序内部是可以在统一时刻进行多种运算的，这就是所谓的**多线程**。

程序、进程、线程的异同参见《操作系统》，这些并不影响我们对Java的学习。

代码：

ThreadDef\_1.java

**public** **class** ThreadDef\_1

{

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

{

A aa = **new** A();

aa.run();

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

{

System.out.println("BBBBBBB");

}

}

}

**class** A **extends** ThreadDef\_1

{

**public** **void** run()

{

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

{

System.out.println("AAAAA");

}

}

}

执行结果：

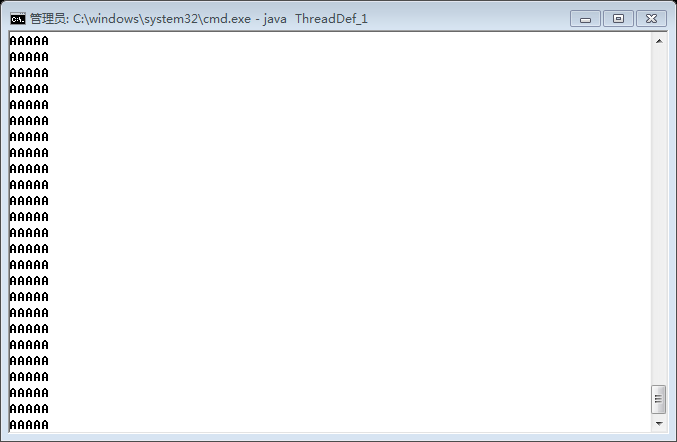


图11-8-1

多线程的优势：

多线程编程简单，效率高（能直接共享数据和资源，多进程不能）

适合于开发服务程序（如Web服务，聊天服务等）

## 11.9 创建一个线程的第一种方法

1.创建一个继承Thread的类（假定类名为A），并重写Thread中的run方法

2.构造一个A类对象，假定对象名为aa

3.调用aa的start方法【start方法是从Thread继承过来的】

代码：

ThreadDef\_2.java

**public** **class** ThreadDef\_2 {

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

A aa = **new** A(); // 构造一个A类对象

aa.start(); // 调用aa的start方法，start方法会创建一个新的线程，并自动调用aa对象的run()方法

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

System.*out*.println("BBBBB");

}

}

}

**class** A **extends** Thread // 创建一个继承Thread的类

{

**public** **void** run() // 重写Thread中的run方法

{

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

System.*out*.println("AAAAAAAAAA");

}

}

}

注意问题：

Thread中start()方法的功能就是创建一个新的线程，并自动调用该线程的run()方法，直接调用run()方法不会创建一个新的线程的

执行一个线程实际就是执行该线程run方法中的代码

执行完aa.start();后并不表示aa所对应的线程就一定会立即得到了执行，aa.start();执行完后只是 表示aa线程具有了可以立即被CPU执行的资格，但由于想抢占CPU执行的线程很多，CPU并不一定会立即去执行aa所对应的线程

代码：

TestThreadStart\_1.java

**public** **class** TestThreadStart\_1

{

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

{

A aa = **new** A();

aa.start(); // 执行完本语句后并不表示说CPU一定会立即开启一个线程，转去执行18到24行的代码

System.out.println("哈哈");

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

{

System.out.println("1111111111");

}

}

}

**class** A **extends** Thread // 创建一个继承Thread的类

{

**public** **void** run() // 18行 重写Thread中的run方法

{

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

{

System.out.println("2222222222");

}

} // 24行

}

/\*

执行完aa.start();后并不表示aa所对象的线程就一定会立即得到了执行，

aa.start();执行完后只是表示t线程具有了可以立即被CPU执行的资格，

但由于想抢占CPU执行的线程很多，CPU并不一定会立即去执行t所对应的线程

\*/

执行结果：

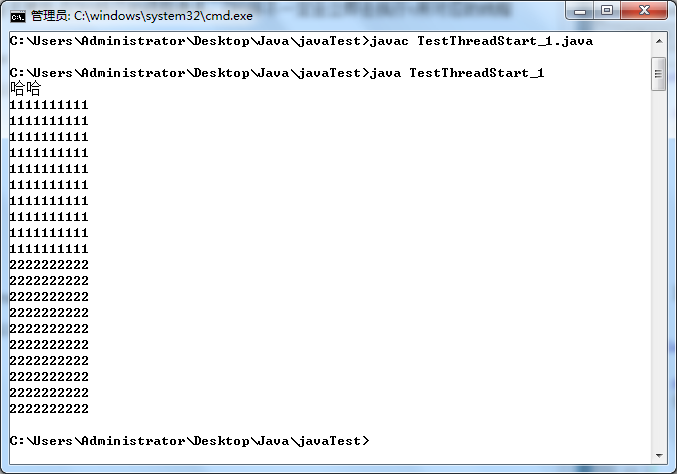


图11-9-1

一个Thread对象能且只能代表一个线程，一个Thread对象不能调用两次start()方法，否则会抛出java.lang.IllegalThreadStateException异常

代码：

TestThreadStart\_2.java

**public** **class** TestThreadStart\_2

{

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

{

A aa = **new** A();

aa.start();

**try**

{

aa.start(); // 一定要明白这是主线程中的语句，这是主线程中可能会抛出的异常，此时很可能aa所对应的线程已经在执行......

}

**catch** (java.lang.IllegalThreadStateException e)

{

System.out.println("一个线程不能启动多次");

System.out.println("一个线程不能启动多次");

System.out.println("一个线程不能启动多次");

System.exit(-1);

}

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

{

System.out.println("BBBBBBBBBBB");

}

}

}

**class** A **extends** Thread // 创建一个继承Thread的类

{

**public** **void** run() // 重写Thread中的run方法

{

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

{

System.out.println("AAAAAAAAAAAAAA");

}

}

}

/\*

一个Thread对象只能代表一个线程并且只能代表一个线程，

一个Thread对象不能调用两次start()f方法，

否则会抛出java.lang.IllegalThreadStateException异常

\*/

执行结果：

某一次的执行结果如下：



图11-9-1

执行结果不唯一

## 11.10 创建一个新线程的第二种方法

1. 定义一个实现了Runnable借口的类，假定为A

2. 创建A类对象aa，代码如下：

A aa = new A();

3. 利用aa构造一个Thread对象tt，代码如下：

Thread tt = new Thread(aa);

4. 调用tt中的start方法，代码如下：

tt.start();

代码：

ThreadTest.java

**public** **class** ThreadTest

{

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

{

A aa = **new** A(); // 创建A类对象aa

Thread tt = **new** Thread(aa); // 利用aa构造一个Thread对象tt

tt.start(); // 调用tt中的start方法

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

{

System.out.println("BBBBBB");

}

}

}

**class** A **implements** Runnable // 定义了一个实现了Runnable接口的类A

{

**public** **void** run()

{

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

{

System.out.println("AAAAAAAAA");

}

}

}

## 11.11 创建一个新线程的第三种方法

实现Callable接口，定义返回值类型，重写call方法；启动方法为，将实例放入FutureTask实例中去，再将FutureTask实例放入一个Thread对象中去，使用Thread的start方法启动。好处：具有返回值

例子：

**import** java.util.concurrent.Callable;

**public** **class** MethodThree **implements** Callable<Integer> {

**private** **int** i;

@Override

**public** Integer call() **throws** Exception {

**for** (; i < 20; i++) {

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

}

**return** i;

}

}

启动方法：

**new** Thread(**new** FutureTask<Integer>(**new** MethodThree()), "创建方法3 ").start();

Callable和FutureTask，它俩很有意思的，一个产生结果，一个拿到结果。Callable接口类似于Runnable，从名字就可以看出来了，但是Runnable不会返回结果，并且无法抛出返回结果的异常，而Callable功能更强大一些，被线程执行后，可以返回值，这个返回值可以被FutureTask拿到，也就是说，FutureTask可以拿到异步执行任务的返回值，

## 11.12 Thread的常用方法

Thread.setName(String name)

设置当前线程的名字

Thread.currentThread()

返回对当前正在执行的线程对象的引用。

Thread.getName()

返回当前线程名字

代码：

TestThreadAPI.java

**public** **class** TestThreadAPI

{

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

{

T t = **new** T();

t.setName("康康"); // 不能写成setName("康康");

t.start();

**for** (**int** i=0; i<30; i++)

{

System.out.printf("嘿嘿");

System.out.printf("%s线程被调用了\n", Thread.currentThread().getName());

}

}

}

**class** T **extends** Thread

{

**public** **void** run()

{

//setName("康康");

**for** (**int** i=0; i<30; i++)

{

System.out.printf("哈哈");

System.out.printf("%s线程被调用了\n", Thread.currentThread().getName());

}

}

}

/\*

2013年5月16日19:55:52

Thread.setName(String name)

设置当前线程的名字

Thread.currentThread()

返回当前线程的引用

Thread.getName()

返回当前线程的名字

\*/

## 11.13 线程的控制

### 线程状态的切换

start();

就绪状态

阻塞状态

运行状态

调度

接触阻塞

导致阻塞的事件

创建

终止

图11-13-1

执行完aa.start();后并不表示aa所对象的线程就一定会立即得到了执行，aa.start();执行完后只是表示t线程具有了可以立即被CPU执行的资格，但由于想抢占CPU执行的线程很多，CPU并不一定会立即执行t所对应的线程。

### 线程控制的基本方法

|  |  |
| --- | --- |
| 方法 | 功能 |
| isAlive() | 判断线程是否还“活”着，即线程是否还未终止 |
| getPriority() | 获得线程的优先级数值 |
| setPriority() | 设置线程的优先级数值 |
| Thread.sleep() | 将当前线程睡眠指定毫秒数 |
| join() | 调用某线程的该方法，将当前线程与该线程“合并”，即等待该线程结束，在恢复当前线程的运行。 |
| yield() | 让出CPU，当前线程进入就绪队列等待调度。 |
| wait() | 当前线程进入对象的wait pool |
| notify()/notifyAll() | 唤醒对象的wait pool中的一个或所有等待线程 |

表11-13-1

### 线程的优先级

线程的优先级用数字来表示，范围从1到10

主线程的缺省优先级是5，子线程的优先级默认与其父线程相同。

Java提供一个线程调度器来监控程序中启动后进入就绪状态的所有线程。线程调度器按照线程的优先级决定应调度哪个线程来执行

线程的优先级用数字表示，范围从1到10，一个线程的缺省优先级是5.

Thread.MIN\_PRIORITY = 1;

Thread.MAX\_PRIORITY = 10;

Thread.NORM\_PRIORITY = 5;

使用下述线程方法获得或设置线程对象的优先级

int getPrority();

void setPriority(int newPriority);

通常高优先级的线程将先于低优先级的执行，但并不总是这样，因此实际开发中不可依赖优先级来决定线程运行次序。

代码：

TestPriority.java

**public** **class** TestPriority

{

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

{

Thread t1 = **new** Thread(**new** T1());

Thread t2 = **new** Thread(**new** T2());

t1.setPriority(Thread.NORM\_PRIORITY + 3); // 考虑把本语句注释掉后会怎样

t1.start();

t2.start();

}

}

**class** T1 **implements** Runnable

{

**public** **void** run()

{

**for** (**int** i=0; i<100; i++)

{

System.out.println("T1:" + i);

}

}

}

**class** T2 **implements** Runnable

{

**public** **void** run()

{

**for** (**int** i=0; i<100; i++)

{

System.out.println("----------T2:" + i);

}

}

}

### 线程的休眠

线程休眠——暂停执行当前运行中的线程，使之进入阻塞状态，待经过指定的“延迟时间”后再醒来并转入到就绪状态。

Thread类提供的相关方法：

public static void sleep(long millis)

public static void sleep(long millis, int nanos)

由于是静态方法，可以由Thread直接调用

sleep()方法会抛出InterruptedException异常，我们必须得对其进行捕捉

代码：

TestSleep.java

**public** **class** TestSleep

{

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

{

A aa = **new** A();

Thread tt = **new** Thread(aa);

tt.start();

}

}

**class** A **implements** Runnable

{

**public** **void** run() // 13行， 这里添加throws Exception是错误的

{

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

{

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

**try**

{

Thread.sleep(1000); // 21行

}

**catch** (Exception e)

{

}

}

}

}

//21行的Thread.sleep(1000)会抛出异常，异常必须得进行捕捉，不能在13行的后面天剑throws Exception

执行结果：

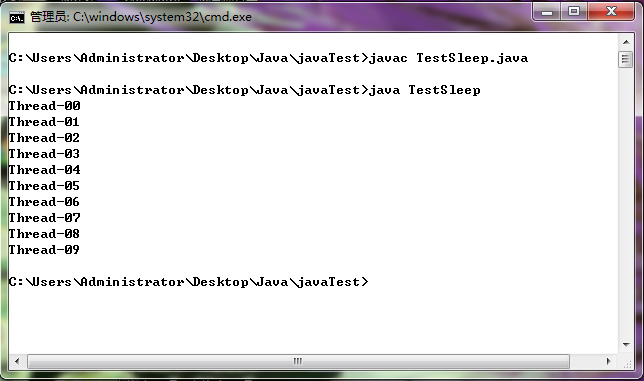


图11-13-2

### run方法不可以抛出异常

**class** A **implements** Runnable

{

**public** **void** run() // 这里不可以写throws Exception

{

}

}

原因：

重写方法抛出异常的范围不能大于被重写方法排除的异常范围

### 线程的让步

让出CPU，给其他线程执行的机会

让运行中的线程主动放弃当前获得的CPU处理机会，但不是使改程序阻塞，而是使之转入就绪状态。

public static void yield()

代码：

TestYield.java

**public** **class** TestYield

{

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

{

MyThread mt = **new** MyThread();

Thread t1 = **new** Thread(mt);

Thread t2 = **new** Thread(mt);

t1.setName("线程A");

t2.setName("线程B");

t1.start();

t2.start();

}

}

**class** MyThread **implements** Runnable

{

**public** **void** run()

{

**for** (**int** i=1; i<=100; i++)

{

System.out.print(Thread.currentThread().getName() + ":" + i + " ");

**if** (0 == i % 10)

{

Thread.yield();

}

}

}

}

执行结果：

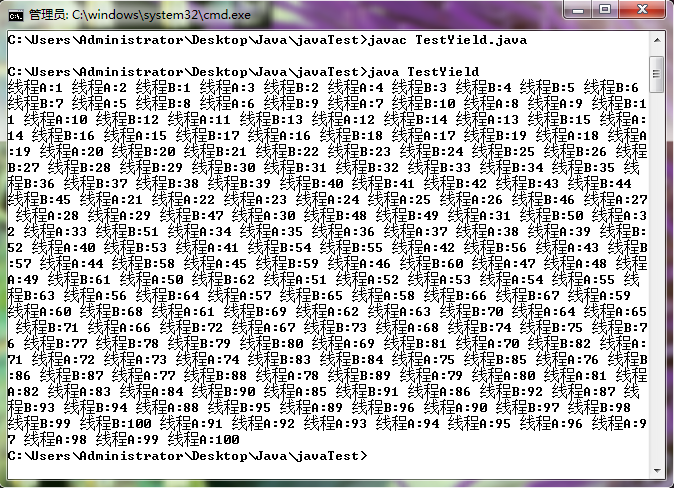


图11-13-3

### 线程的串行化

在多多线程程序中，如果在一个线程运行的过程中要用到另一个线程的运行结果，则可进行线程的串行化处理。

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

代码：

TestJoin.java

**public** **class** TestJoin

{

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

{

MyRunner r = **new** MyRunner();

Thread t = **new** Thread(r);

t.start();

**try**

{

t.join(); // 10行 暂停当前正在执行t.join();的线程，直到t所对应的线程运行终止之后，当前线程才会获得继续执行的机会

}

**catch** (InterruptedException e)

{

e.printStackTrace();

}

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

{

System.*out*.print("主线程：" + i + " ");

}

}

}

**class** MyRunner **implements** Runnable

{

**public** **void** run()

{

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

{

System.*out*.print("子线程：" + i + " ");

}

}

}

执行结果：



图11-13-4

### 线程的挂起和恢复

线程挂起——暂停当前运行中的线程，使之转入阻塞状态，并且不会自动恢复运行。

线程恢复——使得一个已挂起的线程恢复运行。

Thread类提供的相关方法：

public final void suspend()

public final void resume()

suspend()方法挂起线程时并不释放其锁定的资源。这可能会影响其他线程的运行，且容易导致线程的死锁，已不提倡使用。

### 声明周期控制

如何结束一个线程

代码：

TestShutThread.java

**public** **class** TestShutThread

{

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

{

A aa = **new** A();

Thread tt = **new** Thread(aa);

tt.start();

**try**

{

Thread.sleep(5000);

}

**catch** (Exception e)

{

e.printStackTrace();

}

aa.shutDown();

}

}

**class** A **implements** Runnable

{

**private** **boolean** flag = **true**;

**public** **void** run()

{

**while** (flag)

{

System.out.println("AAAA"); //本线程只会有一行代码

}

}

**public** **void** shutDown()

{

**this**.flag = **false**;

}

}

执行结果：



图11-13-5

本程序执行5秒钟之后会停止。

## 11.14 线程的同步

通过两个例子说明

### 买票程序

常犯错误

代码：

TestSync1.java

/\*

2013年5月28日20:34:03

String 对象所带来的线程同步问题

\*/

**public** **class** TestSync1

{

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

{

A aa = **new** A();

Thread t1 = **new** Thread(aa);

Thread t2 = **new** Thread(aa);

t1.start();

Thread.sleep(10);

t2.start();

}

}

**class** A **implements** Runnable

{

**private** **int** tickets = 100;

String str = **new** String("123");

**public** **void** run()

{

**synchronized** (str)

{

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

{

**if** (tickets > 0)

{

System.out.printf("%s线程正在卖出第%d张票!\n",

Thread.currentThread().getName(), tickets);

--tickets;

}

**else**

{

**break**;

}

}

}

}

}

执行结果：

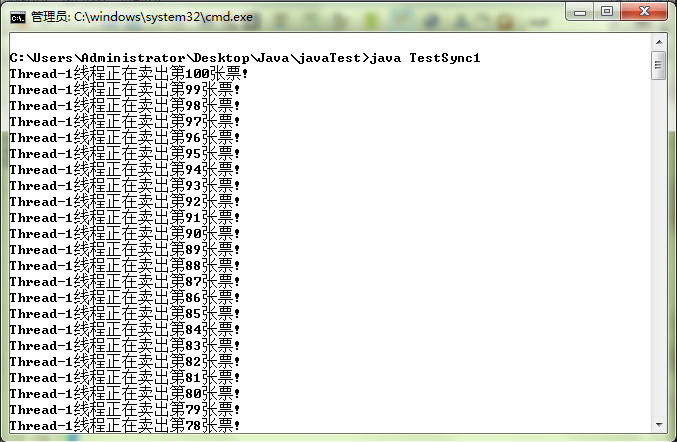


图11-14-1

代码：

TestSync2.java

/\*

2013年5月28日20:36:03

String 对象所带来的线程同步问题

\*/

**public** **class** TestSync2

{

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

{

A aa = **new** A();

Thread t1 = **new** Thread(aa);

Thread t2 = **new** Thread(aa);

Thread t3 = **new** Thread(aa);

Thread t4 = **new** Thread(aa);

t1.start();

t2.start();

t3.start();

t4.start();

}

}

**class** A **implements** Runnable

{

**private** **int** tickets = 100;

**public** **void** run()

{

String str = **new** String("123");

**synchronized** (str)

{

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

{

**if** (tickets > 0)

{

System.out.printf("%s线程正在卖出第%d张票!\n",

Thread.currentThread().getName(), tickets);

--tickets;

}

**else**

{

**break**;

}

}

}

}

}

执行结果：

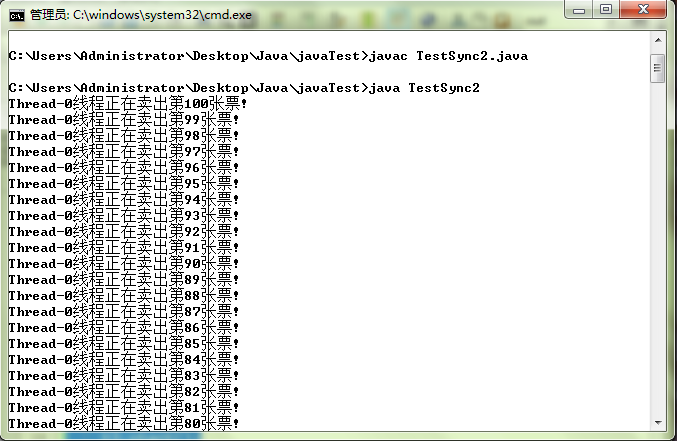


图11-14-2

代码：

TestSync3.java

/\*

正确但很容易写错的买票程序

\*/

**public** **class** TestSync3

{

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

{

ThreadRun tr = **new** ThreadRun();

**new** Thread(tr).start();

**new** Thread(tr).start();

**new** Thread(tr).start();

**new** Thread(tr).start();

**new** Thread(tr).start();

**new** Thread(tr).start();

}

}

**class** ThreadRun **implements** Runnable

{

**private** **int** tickets = 100;

**public** **void** run()

{

**synchronized** (**this**)

{

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

{

**if** (tickets > 0)

{

System.*out*.printf("%s线程正在卖出第%d张票!\n",

Thread.*currentThread*().getName(), tickets);

--tickets;

}

**else**

{

**break**;

}

}

}

}

}

执行结果：

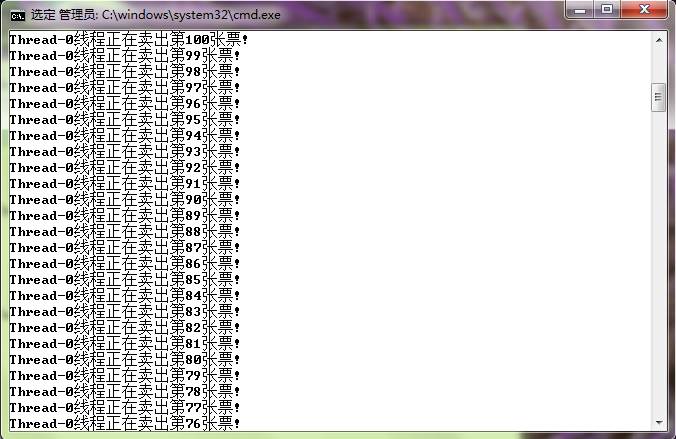


图11-14-3

代码：

TestSync4.java

**public** **class** TestSync4

{

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

{

ThreadRun tr = **new** ThreadRun();

**new** Thread(tr).start();

**new** Thread(tr).start();

**new** Thread(tr).start();

**new** Thread(tr).start();

**new** Thread(tr).start();

**new** Thread(tr).start();

}

}

**class** ThreadRun **implements** Runnable

{

**private** **int** tickets = 100;

**public** **synchronized** **void** run()

{

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

{

**if** (tickets > 0)

{

System.*out*.printf("%s线程正在卖出第%d张票!\n",

Thread.*currentThread*().getName(), tickets);

--tickets;

}

**else**

{

**break**;

}

}

}

}

执行结果：

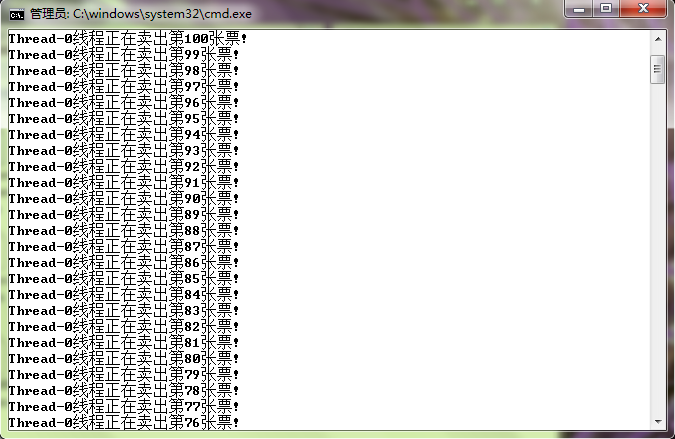


图11-14-4

### 生产消费程序

问题见 [生产消费——经典问题](#_生产消费——经典问题)。

代码：

ProducerConsumer.java

**public** **class** ProducerConsumer

{

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

{

SyncStack ss = **new** SyncStack();

Producer p = **new** Producer(ss);

Consumer c = **new** Consumer(ss);

Thread t1 = **new** Thread(p);

Thread t2 = **new** Thread(c);

t1.start();

t2.start();

}

}

**class** SyncStack

{

**int** cnt = 0;

**char**[] data = **new** **char**[6];

**public** **synchronized** **void** push(**char** ch)

{

**while** (cnt == data.length)

{

**try**

{

**this**.wait(); //wait是Object 类中的方法，不是Thread中的方法，Thread中wait也是继承自Object,

//this.wait();不是让当前对象wait，而是让当前锁定this对象的线程wait，同时释放对this的锁定。

//注意：如果该对象没有被锁定，则调用wait方法就会报错！即只有在同步方法或者同步代码块中才可以调用wait方法，notify同理

}

**catch** (Exception e)

{

}

}

**this**.notify(); //如果注释掉了本语句，可能会导致消费线程陷入阻塞(如果消费线程本身执行很慢的话，则消费线程永远不会wait，即永远不会阻塞)，因为消费线程陷入阻塞， 所以生产线程因此不停生产产品达到6个后也陷入阻塞，最后显示的肯定是“容器中现在共有6个字符!”

//this.notify();叫醒一个现在正在wait this对象的一个线程，如果有多个线程正在wait this对象，通常是叫醒最先wait this对象的线程,但具体是叫醒哪一个，这是由系统调度器控制，程序员无法控制

// nority 和 notifyAll 都是Object 类中的方法

data[cnt] = ch;

cnt++;

System.*out*.printf("生产了: %c\n", ch);

System.*out*.printf("容器中现在共有%d个字符!\n\n", cnt);

}

**public** **synchronized** **char** pop()

{

**char** ch;

**while** (0 == cnt)

{

**try**

{

**this**.wait();

}

**catch** (Exception e)

{

}

}

**this**.notify(); //如果注释掉了本语句，可能会导致生产线程陷入阻塞(如果生产线程本身执行很慢的话，则生产线程永远不会wait，即永远不会阻塞)，因为生产线程陷入阻塞，消费线程因此不停取出产品，当容器中再也没有产品时消费线程也陷入阻塞，最后显示的肯定是“容器中现在共有0个字符!”

ch = data[cnt-1];

--cnt;

System.*out*.printf("取出: %c\n", ch);

System.*out*.printf("容器中现在共有%d个字符!\n\n", cnt);

**return** ch;

}

}

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

{

SyncStack ss = **null**;

**public** Producer(SyncStack ss)

{

**this**.ss = ss;

}

**public** **void** run()

{

**char** ch;

//总共生产20个产品

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

{

ch = (**char**)('a'+i);

ss.push(ch);

// try

// {

// Thread.sleep(500);

// }

// catch (Exception e)

// {

// }

}

}

}

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

{

SyncStack ss = **null**;

**public** Consumer(SyncStack ss)

{

**this**.ss = ss;

}

//总共消费20个产品

**public** **void** run()

{

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

{

ss.pop();

**try**

{

Thread.*sleep*(500);

}

**catch** (Exception e)

{

}

}

}

}

执行结果：

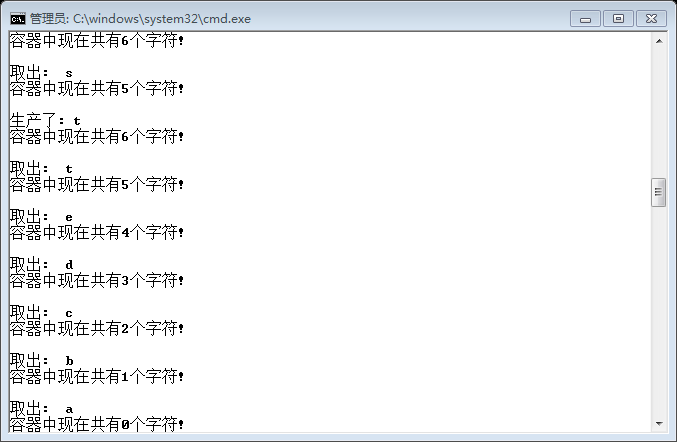


图11-14-5

### Synchromized关键字

synchronized可以用来修饰：

1. 一个方法
2. 一个方法内部的代码块

#### synchronized修饰代码块

格式：

synchronized (类对象名aa) // 1行

{

同步代码块 //3行

}

功能：

synchronized (类对象名aa)的含义是：判断aa是否已经被其他线程霸占，如果发现已经被其他线程霸占，则当前线程陷入等待中，如果发现aa没有被其他线程霸占，则当前县城霸占住aa对象，并执行3行的代码(因为当前线程已经霸占了aa对象)，当前线程执行完3行的代码后，会自动释放对aa对象的霸占，此时其他线程会相互竞争对aa的霸占，最终CPU会选择其中的某一个线程执行。

最终导致的结果是：一个线程正在操作某资源的时候，将不允许其他线程操作该资源，即一次只允许一个线程处理该资源

#### synchronized修饰方法

synchronized修饰一个方法时，实际霸占的是该方法的this指针指向的对象

即synchronized修饰一个方法时，实际霸占的正在调用该方法的对象

附注：

霸占的专业术语叫做：锁定

霸占住的那个对象专业术语叫做：监听器

### 同步概念

通常，一些同时运行的线程需要共享数据。在这种时候，每个线程就必须要考虑与其他一起共享数据的线程的状态与行为，否则的话就不能保证共享数据的一致性，从而也就不能保证程序的正确性。

class Stack {

int index = 0;

char[] data = new char[6];

public void push(char c) {

data[index] = c;

index++;

}

public char pop() {

index--;

resturn data[index];

}

}

当有两个线程A和B同时使用了Stack类的一个对象时，现在我们要求：先把r存入Stack中，再将r去出来

下面的步骤详细演示了AB线程不同步所带来的问题

1. 操作之前，堆栈中有两个字符：

data = |a|c| | | | | index = 2

1. A执行push中的第一条语句data[index] = ‘r’:

data = |a|c|r| | | | index = 2 //2还没有被存入

1. A还没有执行index++语句，A被B中断，B执行pop()方法，返回’c’:

data = |a|c|r| | | | index = 1 //取出的是’C’却不是’r’

1. A继续执行index++语句：

data = |a|c|r| | | | index = 2

最终结果是：’r’没有被存入，取出的是’C’而不是’r’

**public** **class** SyncStack {

**private** **int** index = 0;

**private** **char**[] data = **new** **char**[6];

**public** **synchronized** **void** push(**char** c) {

**while** (index == data.length) {

**try** {

**this**.wait(); // 7行

} **catch** (InterruptedException e) {}

}

**this**.notify(); // 10行

data[index] = c;

index++;

System.*out*.println("produced:" + c);

}

**public** **synchronized** **char** pop() {

**while** (index == 0) {

**try** {

**this**.wait();

} **catch** (InterruptedException e) {}

}

**this**.notify(); // 20行

index--;

System.*out*.println("消费:" + data[index]);

**return** data[index]; // 23行

}

}

假设现在有两个线程P（生产）和C（消费），P（生产）已满，执行7行代码陷入阻塞状态，同时释放P线程对this的锁定，这时候C线程会得到this对象的标志位开始运行。

另外C线程执行完20行代码后，程序并不会立即转到P线程开始运行，因为C执行notify，只是叫醒P，让P从因为wait this对象而陷入阻塞的状态进入就绪状态，记住：一个线程notify，该线程并不会释放对this的锁定，只有C执行完23行的代码后，C才会释放对this的争夺对this的锁定，具体执行哪个由系统决定

### notify和wait方法

代码：

TestNotify.java

/\*

2013年7月28日12:12:43

this.notify();

功能:

不是叫醒正在执行this.notify();的当前线程

而是叫醒一个现在正在wait this对象的其他线程，如果有多个线程正在wait this对象，

通常是叫醒最先wait this对象的线程,但具体是叫醒哪一个，

这是由系统调度器控制，程序员无法控制

但是要注意：

假设现在有T1 T2 T3 T4 四个线程

在T4线程中调用了

aa.notify()

即便此时T1 T2 T3 没有一个线程因为wait aa对象而陷入阻塞状态，

T4线程中执行aa.notify方法时也不会有任何错误

本程序就证明了这一点

\*/

**public** **class**

{

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

{

A aa = **new** A();

Thread tt = **new** Thread(aa);

tt.start();

System.*out*.println("哈哈!");

ThreadRun tr = **new** ThreadRun();

**new** Thread(tr).start();

**new** Thread(tr).start();

**new** Thread(tr).start();

**new** Thread(tr).start();

**new** Thread(tr).start();

**new** Thread(tr).start();

}

}

**class** A **implements** Runnable

{

**public** **synchronized** **void** run() //如果去掉了synchronized 则会报错，只有在同步代码块或同步方法中才可以使用notify方法

{

**int** cnt = 0;

**while** (cnt < 20)

{

cnt++;

System.*out*.println("AAAA" + cnt);

**try**

{

**this**.notify(); //如果此时有很多线程因为 wait this对象而陷入阻塞状态，则叫醒其中的一个,至于具体是叫醒哪一个，则有系统调度器确定，程序员无法控制

//本程序证明了，即便此时没有任何线程因为 wait this对象而陷入阻塞状态，也可以调用notify方法

}

**catch** (IllegalMonitorStateException e)

{

System.*out*.println("在非同步方法 非同不代码块中调用了notify方法");

System.*exit*(-1);

}

}

}

}

**class** ThreadRun **implements** Runnable

{

**private** **int** tickets = 100;

**public** **void** run()

{

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

{

**synchronized** (**this**)

{

**if** (tickets > 0)

{

System.*out*.printf("%s线程在卖出第%d张票!\n",

Thread.*currentThread*().getName(), tickets);

--tickets;

}

**else**

{

**break**;

}

}

}

}

}

执行结果：

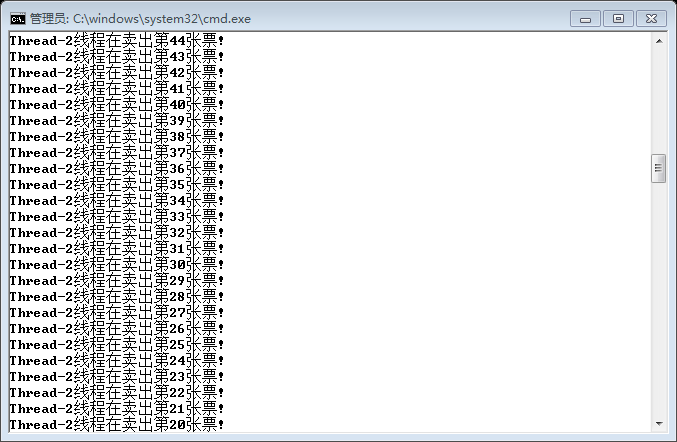


图11-14-6

代码：

SyncTest.java

**public** **class** SyncTest

{

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

{

SyncStack stack = **new** SyncStack();

Runnable p=**new** Producer(stack);

Runnable c = **new** Consumer(stack);

Thread t1 = **new** Thread(p);

Thread t2 = **new** Thread(c);

t1.start();

Thread.*sleep*(1000);

t2.start();

// Thread t3 = new Thread(p);

// t3.start();

// Thread t4 = new Thread(c);

// t4.start();

}

}

**class** SyncStack{ //支持多线程同步操作的堆栈的实现

**private** **int** index = 0;

**private** **char** []data = **new** **char**[6];

**public** **synchronized** **void** push(**char** c)

{

**while**(index == data.length)

{

**try**

{

**this**.wait();

}**catch**(InterruptedException e)

{

e.printStackTrace();

System.*exit*(-1);

}

}

**this**.notify();

data[index] = c;

index++;

System.*out*.println("生产: "+c);

System.*out*.printf("容器里有%d个字母\n\n", index);

}

**public** **synchronized** **char** pop()

{

**while**(index == 0)

{

**try**

{

**this**.wait();

}

**catch**(InterruptedException e)

{

e.printStackTrace();

System.*exit*(-1);

}

}

**this**.notify();

index--;

System.*out*.println("取出："+data[index]);

System.*out*.printf("容器里还有%d个字母!\n\n", index);

**return** data[index];

}

}

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

{

SyncStack stack;

**public** Producer(SyncStack s)

{

stack = s;

}

**public** **void** run()

{

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

{

**char** c =(**char**)('a'+i);

stack.push(c);

**try**

{

Thread.*sleep*((**int**)(Math.*random*()\*20));

}**catch**(InterruptedException e)

{

}

}

}

}

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

{

SyncStack stack;

**public** Consumer(SyncStack s)

{

stack = s;

}

**public** **void** run()

{

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

{

**char** c = stack.pop();

**try**

{

Thread.*sleep*((**int**)(Math.*random*()\*1000));

}

**catch**(InterruptedException e)

{

}

}

}

}

执行结果：

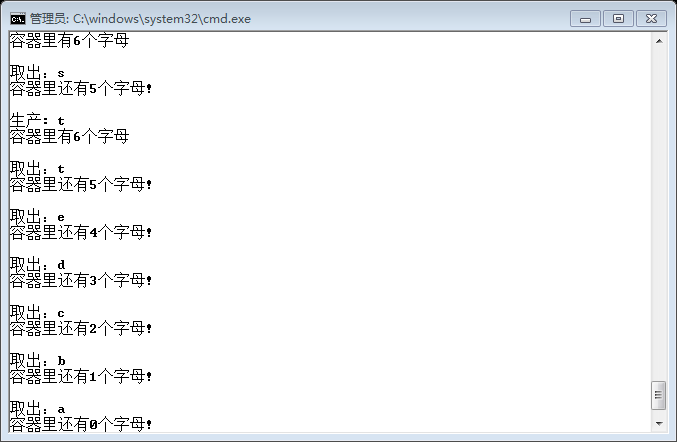


图11-14-7

this.notify();

功能：

不是叫醒正在执行this.notify();的当前进程，而是叫醒一个现在正在wait this对象的其他线程，如果有多个线程正在wait this对象，通常是叫醒最先wait this对象的线程，但具体是叫醒哪一个，这是由系统调度器控制，程序员无法控制。

假设现在有T1, T2, T3, T4四个线程，我们在T4线程中执行了aa.notify()语句，则即便此时T1 T2 T3没有一个线程因为wait aa对象而陷入阻塞状态，T4线程中执行aa.notify方法时也不会有任何错误，本程序就证明了这一点，执行aa.notify方法时如果一个线程都没有叫醒，这是可以的

notify和wait方法总结

aa.wait()

将执行aa.wait()的当前线程转入阻塞状态，让出CPU的控制权

释放对aa的锁定

aa.notify()

假设执行aa.notify()的当前线程为T1

如果当前时刻有其他线程因为执行了aa.wait()而陷入阻塞状态，则叫醒其中的一个。

所谓叫醒某个线程就是令该线程从因为wal而陷入阻塞的状态转入就绪状态

aa.notifyAll()

叫醒其他所有的因为执行了aa.wait()而陷入阻塞状态的线程

### 生产消费——经典问题

一个仓库最多容纳6个产品，制造商现在制造20件产品存入仓库，消费者要从仓库取出这20件产品来消费

制造商制造产品和消费者取出产品的速度很可能不是一样的，编程实现两者的同步

代码：见[生产消费程序->ProducerConsumer.java](#生产消费代码)

# 12 GUI

## 12.1 GUI基础

### 组件

组件（Component）是图形用户界面的基本组成元素，凡是能够以图形化显示在屏幕上并且能够与用户进行交互的对象均为组件，如菜单、按钮、标签、文本框、滚动条等。

组件分类如下：

java.awt.Component

java.awt.MenuComponent

说明：抽象类java.awt.Component是除菜单相关组件之外所有javaAWT组件类的根父类，该类规定了GUI组件的基本特性，如尺寸、位置和颜色效果等，并实现了作为一个GUI组件所应具备的基本功能。

### 容器

组件通常不能独立的显示出来，必须将组件放在一定的容器中才可以显示出来。

有一类特殊的组件是专门用来包含其他组件的，这类组件叫做容器，java.awt.Container是所有容器的父类，java.awt.Container继承自java.awt.Component

容器类对象本身也是一个组件，具有组件的所有性质，但反过来组件却不一定是容器。

容器分布图

Component

Container

Window

Frame

Dialog

Panel

Applet

BorderLayout

FlowLayout

图12-1-1 GUI默认布局管理器

### Frame常用的方法

**public** **void** setBounds(**int** x, **int** y, **int** width, **int** height)

// 设置窗体的位置和大小，x和y表示窗体左上角距离屏幕的水平距离和垂直距离，with和height是窗体本身的高宽度和高度

**public** **void** setSize(**int** width, **int** height)

// 设置窗体的大小，width和height是窗体本身的高度和宽度

**public** **void** setVisible(**boolean** flag)

// 设置窗体是否可见，true表示可见，false表示不可见

**public** **void** setBackground(Color c)

// 设置窗体的背景色

代码：

TestFrame\_1.java

/\*

有关Frame类中方法的使用\_1

\*/

**import** java.awt.\*;

**public** **class** TestFrame\_1

{

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

Frame f = **new** Frame("My First Test"); //这句话只是在内存中产生了一个对象，你要想让窗口在显示器上显示出来，则必须得调用f.setVisible(true);

//f.setLocation(800, 300); //第一个参数表示新产生的窗口距离屏幕(这个屏幕不是指当前屏幕，而是指整个Windows系统的屏幕)最左边有几个像素 第二个参数表示新产生的窗口距离屏幕最上边有几个像素

f.setSize(300, 600); //第一个参数表示窗口宽度 第二个参数表示窗口的高度

f.setBackground( Color.*PINK*); //设置背景色

//f.setResizable(false); //设置是否可以调整窗口大小，false表示不可以调整大小，true表示可以调整大小

f.setVisible( **true**); //设置窗口是否可见 true表示可见，false表示不可见

}

}

执行结果：

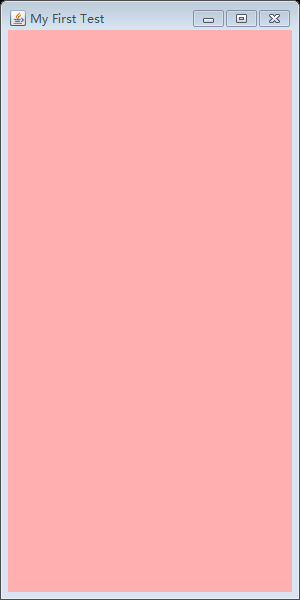


图12-1-2

代码：

TestFrame\_2.java

/\*

有关Frame类中方法的使用

\*/

**import** java.awt.Frame;

**public** **class** TestFrame\_2

{

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

{

Frame f = **new** Frame("哈哈");

f.setSize(200, 200);

f.setVisible(**true**); //先在距离显示器（0，0）的位置显示窗口

**try**

{

Thread.*sleep*(2000);

}

**catch** (InterruptedException e)

{

System.*out*.println(e.getMessage());

}

f.setLocation(200, 200); //过2秒后窗口移动到距离显示器（200， 200）的位置显示

**try**

{

Thread.*sleep*(2000);

}

**catch** (InterruptedException e)

{

System.*out*.println(e.getMessage());

}

f.setVisible(**false**); //再过一秒钟，窗口消失

}

}

执行结果：

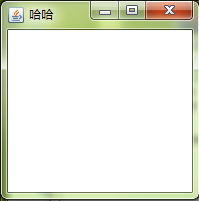


图12-1-3

代码：

TestFrame\_3.java

**import** java.awt.\*;

**public** **class** TestFrame\_3

{

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

MyFrame f1 =

**new** MyFrame(100,100,200,200,Color.*BLUE*); //一个类A中定义的属性的类型可以就是类A本身

MyFrame f2 =

**new** MyFrame(300,100,200,200,Color.*YELLOW*);

MyFrame f3 =

**new** MyFrame(100,300,200,200,Color.*GREEN*);

MyFrame f4 =

**new** MyFrame(300,300,200,200,Color.*MAGENTA*);

}

}

**class** MyFrame **extends** Frame

{

**public** **static** **int** *id* = 0;

MyFrame(**int** x,**int** y,**int** w,**int** h,Color color)

{

**super**("MyFrame " + (++*id*));

setBackground(color);

setLayout(**null**);

setBounds(x,y,w,h);

setVisible(**true**);

}

}

执行结果：

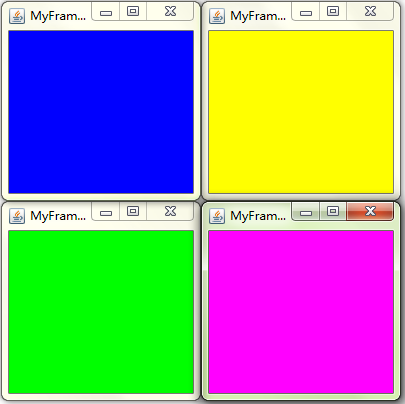


图12-1-4

### Panel

Panel是容纳其他组件的组件，是容器。

Panel不能单独存在，必须得被添加到其他容器中去

Panel类拥有从其父类继承来的以下方法：

**public** **void** setBounds(**int** x, **int** y, **int** width, **int** height)

// 设置窗体的位置和大小，x和y表示窗体左上角距离屏幕的水平距离和垂直距离，with和height是窗体本身的高宽度和高度

**public** **void** setSize(**int** width, **int** height)

// 设置窗体的大小，width和height是窗体本身的高度和宽度

**public** **void** setVisible(**boolean** flag)

// 设置窗体是否可见，true表示可见，false表示不可见

**public** **void** setBackground(Color c)

// 设置窗体的背景

**public** **void** setLayout(LayoutManager mgr)

// 设置此容器的布局管理器

Panel()的构造方法为：

Panel()使用的默认的FlowLayout类布局管理器初始化

Panel(LayoutManager layout) 使用指定的布局管理器初始化。

代码：

TestPanel\_1.java

**import** java.awt.\*;

**public** **class** TestPanel\_1

{

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

{

Frame f = **new** Frame("Java Frame with Panel");

Panel p = **new** Panel();

f.setLayout(**null**);

f.setBounds(300,300,500,500); //Frame的设置是相对于整个屏幕而言的

f.setBackground(**new** Color(100,100,102)); //三个颜色分量都分别使用一个字节表示

p.setBounds(300/2, 300/2, 500/2, 500/2); //Panel的设置是相对于Panel所在的父窗口而言的

p.setBackground(**new** Color(204,204,255));

f.add(p); // 这是java.awt.Container类中的一个方法， 把当前Panel对象添加到当前Frame对象中

f.setVisible(**true**);

}

}

执行结果：

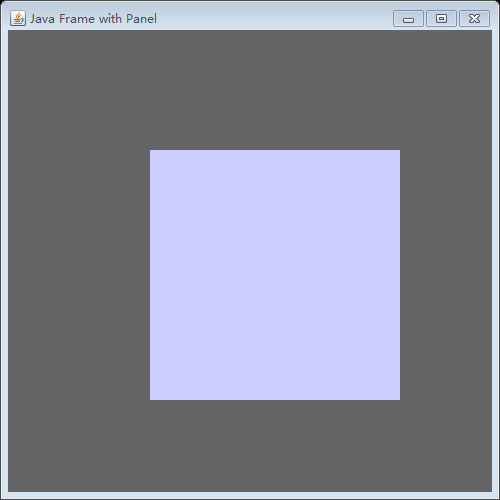


图12-1-5

## 12.2 布局管理器

容器对其中所包含组件的排列方式，包括组件的位置和大小设定，被称为容器的布局（Layout）。

为了使图形用户界面具有良好的平台无关性，Java语言提供了布局管理器来管理容器的布局，而不建议直接设置组件在容器中的位置和尺寸。

每个容器都由一个默认的布局管理器，当容器需要对某个组件进行定位或判断其大小尺寸时，就会自动调用其对应的布局管理器。

在AWT中，常见的布局管理器有：

BorderLayout

FlowLayout

GridLayout

### GUI默认布局管理器

图见 [GUI默认布局管理器](#OLE_LINK1)。

### FlowLayout布局管理器

FlowLayout是Panel类的默认布局管理器。

FlowLayout布局管理器对组件逐行定位，行内从左到右，一行排满后换行。

不改变组件的大小，按组件原有尺寸显示组件，可设置不同的组件间距，行距以及对齐方式。

FlowLayout布局管理器默认的对齐方式是居中。

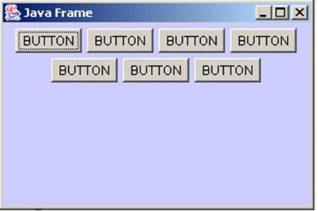


图12-2-1

**new** FlowLayout(FlowLayout.*RIGHT*, 20, 40);

// 右对齐，组件之间水平间距20个像素，垂直间距40个像素。

**new** FlowLayout(FlowLayout.*LEFT*);

// 左对齐，水平和垂直间距为缺省值（5）。

**new** FlowLayout();

// 使用缺省的居中对齐方式，水平和垂直间距为缺省值（5）。

代码：

TestFlowLayout\_1.java

**import** java.awt.\*;

**public** **class** TestFlowLayout\_1 {

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

Frame f = **new** Frame("Flow Layout");

Button button1 = **new** Button("Ok");

Button button2 = **new** Button("Open");

Button button3 = **new** Button("Close");

f.setLayout(**new** FlowLayout(FlowLayout.*CENTER*)); //设置f是FlowLayout布局，并且是右对齐

f.add(button1);

f.add(button2);

f.add(button3);

f.setSize(300,400);

f.setVisible(**true**);

}

}

执行结果：

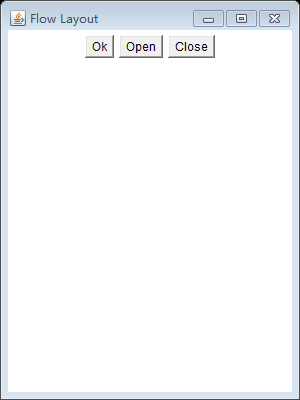


图12-2-2

代码：

TestFlowLayout\_2.java

**import** java.awt.\*;

**public** **class** TestFlowLayout\_2 {

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

Frame f = **new** Frame("Java Frame");

FlowLayout l =

**new** FlowLayout(FlowLayout.*CENTER*, 30, 30); //第二个参数400表示的是f容器中的组件之间的行间距，即左右间距； 第三个参数表示的是f容器中的组件之间的垂直间距，即上下距离的间距

f.setLayout(l);

f.setLocation(300,400);

f.setSize(300,100); //如果尺寸设置太小，则7个按钮就显示不全，不过可以利用鼠标拉大窗口，就会看到完整的7个按钮!

f.setBackground(**new** Color(255,222,255));

**for**(**int** i = 1; i<=7; i++){

f.add(**new** Button("BUTTON"));

}

f.setVisible(**true**);

}

}

执行结果：



图12-2-3

### BorderLayout布局管理器

BorderLayout布局管理器

BorderLayout是Frame类的默认布局管理器。

东（EAST）

西（WEST）

南（SOUTH）

北（NORTH）

中（CENTER）

五个区域，组件只能被添加到指定的区域。

如果不指定组件的加入部位，则默认加入到CENTER区。

每个区域只能加入一个组件，如加入多个，则先前加入的会被覆盖。

BorderLayout型布局容器尺寸缩放原则：

北、南两个区域在水平方向缩放。

东、西两个区域放在垂直方向缩放。

中部可在两个方向上缩放。

代码：

TestBorderLayout.java

**import** java.awt.\*;

**public** **class** TestBorderLayout

{

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

{

Frame f;

f = **new** Frame("Border Layout");

Button bn = **new** Button("BN");

Button bs = **new** Button("BS");

Button bw = **new** Button("BW");

Button be = **new** Button("BE");

Button bc = **new** Button("BC");

f.add(bn, "North");

f.add(bs, "South");

f.add(bw, "West");

f.add(be, "East");

f.add(bc, "Center");

// 也可使用下述语句

/\*

f.add(bn, BorderLayout.NORTH);

f.add(bs, BorderLayout.SOUTH);

f.add(bw, BorderLayout.WEST);

f.add(be, BorderLayout.EAST);

f.add(bc, BorderLayout.CENTER);

\*/

f.setSize(200,200);

f.setVisible(**true**);

}

}

执行结果：

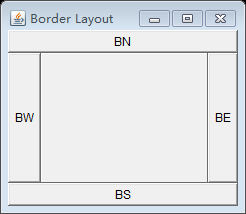


图12-2-4

### GridLayout布局管理器

GridLayout型布局管理器将空间划分成规则的矩形网格，每个单元格区域大小相等。组件被添加到每个单元格中，先从左到右填满一行后换行，在从上到下。

在GridLayout构造方法中指定分割的行数和列数：

如：GridLayout(3, 4)

GridLayout是以行数为准的。

代码：

TestGridLayout.java

/\*

2013年7月29日19:21:23

GridLayout的构造函数

public GriLayout(int, int);

的用法

\*/

**import** java.awt.\*;

**public** **class** TestGridLayout

{

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

{

Frame f = **new** Frame("GridLayout Example");

Button b1 = **new** Button("b1");

Button b2 = **new** Button("b2");

Button b3 = **new** Button("b3");

Button b4 = **new** Button("b4");

Button b5 = **new** Button("b5");

//Button b6 = new Button("b6");

f.setLayout (**new** GridLayout(2,10)); //第一个参数是多少，则输出的就是多少行，列数是编译器根据行数自动计算出来的

f.add(b1);

f.add(b2);

f.add(b3);

f.add(b4);

f.add(b5);

//f.add(b6);

f.pack(); //如果把该语句注释掉，则输出结果就变成一个只有标题栏的小窗口，摘自API：“pack() - 类 java.awt.Window 中的方法， 调整此窗口的大小，以适合其子组件的首选大小和布局” ，郝斌：也就是把子组件包括起来，会根据它所容纳的子组件的大小自动调整自身的大小 ，把子组件包括起来

f.setVisible(**true**);

}

}

执行结果：

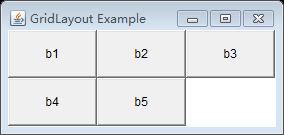


图12-2-5

### 布局管理器总结

Frame是一个顶级窗口，Frame的缺省布局管理器为BorderLayout。

Panel无法单独显示，必须添加到某个容器中。Panel的缺省布局管理器为FlowLayout。

当把Panel作为一个组件添加到某个容器中后，该Panel仍然可以有自己的布局管理器。

使用布局管理器时，布局管理器负责各个组件的大小和位置，因此用户无法再这种情况下设置组件大小和位置属性，如果试图使用Java语言提供的setLocation(), setSize(), setBounds()等方法，则都会被布局管理器覆盖。

如果用户确实需要亲自设置组件大小或位置，则应取消该容器的布局管理器，方法为：setLayout(null)

### 布局管理器练习题

请编程设计出如下的图形

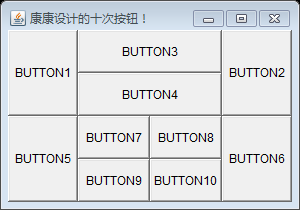


图12-2-6

答案如下

代码：

LayoutPractice.java

**import** java.awt.\*;

**public** **class** LayoutPractice **extends** Frame {

**private** Button[] Button = **new** Button[10];

**private** String[] buttonString = {"BUTTON1", "BUTTON2","BUTTON3","BUTTON4","BUTTON5","BUTTON6","BUTTON7","BUTTON8", "BUTTON9", "BUTTON10"};

**private** Panel p1 = **null**;

**private** Panel p2 = **null**;

**private** Panel p3 = **null**;

**private** Panel p4 = **null**;

**public** LayoutPractice() {

/\*创建组件\*/

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

Button[i] = **new** Button(buttonString[i]);

}

p1 = **new** Panel();

p2 = **new** Panel();

p3 = **new** Panel();

p4 = **new** Panel();

/\*设置布局管理器\*/

**this**.setLayout(**new** GridLayout(2, 1));

p1.setLayout(**new** BorderLayout());

p2.setLayout(**new** BorderLayout());

p3.setLayout(**new** GridLayout(2, 1));

p4.setLayout(**new** GridLayout(2, 2));

/\*添加组件\*/

p1.add(Button[0], BorderLayout.*WEST*);

p1.add(Button[1], BorderLayout.*EAST*);

p1.add(Button[2], BorderLayout.*NORTH*);

p1.add(Button[3], BorderLayout.*SOUTH*);

p1.add(p3, BorderLayout.*CENTER*);

p2.add(Button[4], BorderLayout.*WEST*);

p2.add(Button[5], BorderLayout.*EAST*);

p2.add(p4, BorderLayout.*CENTER*);

p3.add(Button[2]);

p3.add(Button[3]);

p4.add(Button[6]);

p4.add(Button[7]);

p4.add(Button[8]);

p4.add(Button[9]);

**this**.add(p1);

**this**.add(p2);

/\*设置窗口\*/

**this**.setTitle("康康设计的十次按钮！");

**this**.setSize(300, 210);

**this**.setLocationRelativeTo(**null**);

**this**.setVisible(**true**);

}

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

**new** LayoutPractice();

}

}

## 12.3 事件处理

### 事件处理相关概念

事件（Event）

用户对组件的一个操作，称之为一个事件

事件源（Event Source）

能够产生事件的GUI组件对象，如按钮、文本框等。

事件处理方法（Event Handler）

能够接收、解析和处理事件类对象，实现与用户交互功能的方法。

事件监听器（Event Listener）

可以处理事件的一个类。

默认情况下事件源不会自动长生任何事件，程序员需要做两件事：

告诉事件源可以自动产生哪类事件，即：向事件源注册某种事件的事件监听器对象

设计好可以处理这种事件的事件监听器

一旦完成了这两步操作，当用户对事件源进行操作时，事件源会自动产生事件，事件源就会自动把产生的事件封装成一个事件对象，事件源就会自动把封装好的事件对象传递给事件监听器

事件监听器收到事件源发送过来的事件时，事件监听器就会自动调用相应的事件处理方法来对该事件进行相应的处理。

### 事件处理步骤

假设事件为XXXX

1 向事件注册某种事件的监听器对象

addXXXXListener(……);

2 设计好可以处理这种事件的事件监听器

class 类名 implements XXXXListener

{

重写XXXXListener接口中的方法

}

说明：要想设计出能够处理XXXX事件的监听器，只需要编写出实现了XXXXListener接口的类就OK了，因为XXXXListener接口中已经定义了可以处理XXXX事件的方法。

### 遗留问题

事件有哪些？

java.awt.event包中含有所有的事件，常用的事件有：ActionEvent:激活组件时发生的事件 KeyEvent:操作鼠标时发生 MouseEvent:操作鼠标时发生 WindowEvent:操作窗口时发生的事件，如最大化或最小化某一窗口。

一个事件源可以自动产生哪些事件？

第三方软件会自动显示，不需要特殊记忆。

事件处理示意图

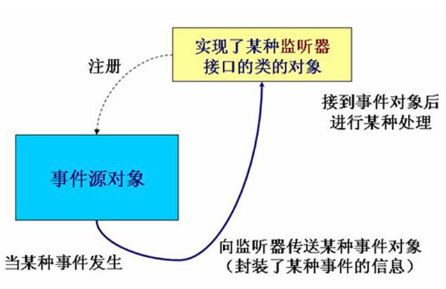


图12-3-1

注册和实现某种监听器接口的类对象这两步是由程序员完成的，其他步骤由编译器自动完成。

### Button监听器

代码：

TestActionEvent.java

**import** java.awt.event.ActionEvent;

**import** java.awt.event.ActionListener;

**import** java.awt.Frame;

**import** java.awt.Button;

**class** MyMonitor **implements** ActionListener

{

@Override

**public** **void** actionPerformed(ActionEvent e)

{

System.*out*.println("按钮被按下!");

}

}

**public** **class** TestActionEvent

{

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

{

Frame f = **new** Frame("Test");

Button b = **new** Button("Press Me!");

MyMonitor mm = **new** MyMonitor();

b.addActionListener(mm);

f.add(b);

f.pack();

f.setVisible(**true**);

}

}

执行结果：

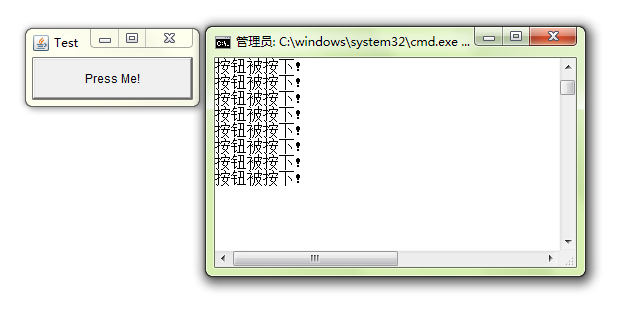


图12-3-2

### TextField监听器

TextField对象可能发生Action（光标在文本框内敲回车）事件。与该事件对应的事件类是java.awt.event.ActionEvent。

用来处理ActionEvent事件是实现了java.awt.event.ActionListener接口的类的对象。ActionListener接口定义有方法：**public** **void** actionPerformed(ActionEvent e)

实现该接口的类要在该方法中添加处理该事件（Action）的语句。

使用addActionListener(ActionListener l)方法为TextField对象注册一个ActionListener对象，当TextField对象发生Action事件时，会生成一个ActionEvent对象，该对象作为参数传递给ActionListener对象的actionPerformed方法在方法中可以获取该对象的信息，并做相应的处理。

代码：

TFActionEvent.java

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

**import** java.awt.\*;

**import** java.awt.event.\*;

**public** **class** TFActionEvent

{

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

{

Frame f = **new** Frame();

TextField tf = **new** TextField(30);

tf.addActionListener(**new** MyMonitor());

f.add(tf);

f.pack();

f.setVisible(**true**);

}

}

**class** MyMonitor **implements** ActionListener

{

**public** **void** actionPerformed(ActionEvent e)

{

TextField tf = (TextField)e.getSource();

System.*out*.println(tf.getText());

tf.setText("");

}

}

执行结果：

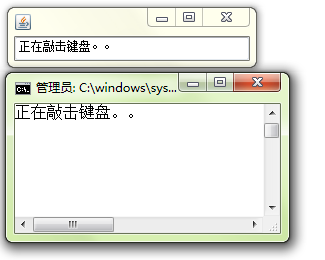


图12-3-3

代码：

TFPassword.java

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

**import** java.awt.\*;

**import** java.awt.event.\*;

**public** **class** TFPassword

{

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

{

Frame f = **new** Frame();

TextField tf = **new** TextField(30);

tf.addActionListener(**new** MyMonitor());

f.add(tf);

tf.setEchoChar('\*'); ////setEchoChar()方法是TextFile类中的一个方法， void setEchoChar(char c) ，当用户从键盘上向文本框输入字母时，在文本框显示的不是用户输入的那个字符而是实参c代表的字符

f.pack();

f.setVisible(**true**);

}

}

**class** MyMonitor **implements** ActionListener

{

**public** **void** actionPerformed(ActionEvent e)

{

TextField tf = (TextField)e.getSource();

String strRec = tf.getText();

**if** (strRec.equals("kangkang"))

{

System.*out*.println("验证通过!");

}

**else**

{

System.*out*.println("密码错误，请重新输入!");

}

tf.setText("");

}

}

执行结果：

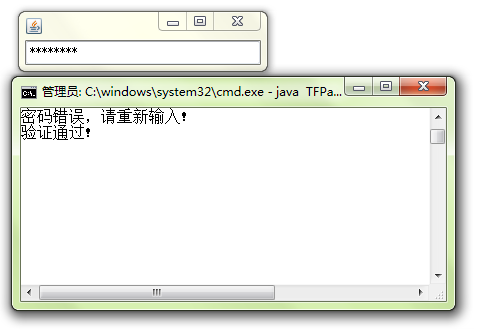


图12-3-4

在一个类中访问另一个类中属性的示例

代码：

Test.java

**class** A

{

**public** **int** i = 3;

**public** **int** j = 4;

}

**class** B

{

**private** A aa = **null**;

**public** B(A ma) {

**this**.aa = ma;

}

**public** **void** f() {

System.*out*.printf("%d, %d\n", aa.i, aa.j);

}

}

**public** **class** Test

{

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

A a1 = **new** A();

B bb = **new** B(a1);

bb.f();

}

}

执行结果：

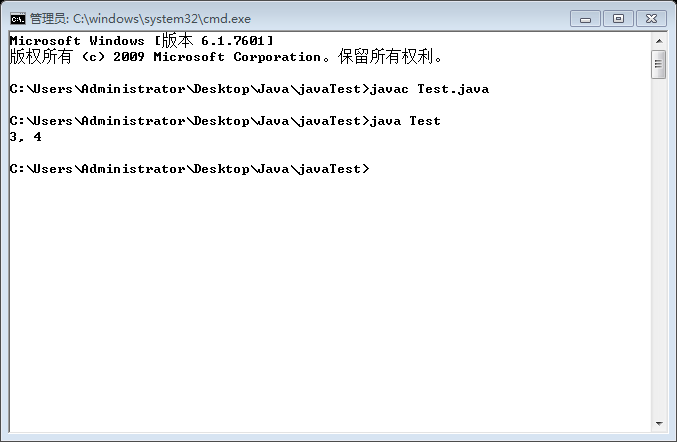


图12-3-5

示意图

bb : XXXX

a1 : XXXX

aa : XXXX

i : 3

j : 4

图12-3-6

### Window监听器

Window事件所对应的事件类为WindowEvent，所对应的事件监听接口为WindowListener。

WindowListener定义的方法：

**public** **void** windowOpened(WindowEvent e)

**public** **void** windowClosing(WindowEvent e)

**public** **void** windowClosed(WindowEvent e)

**public** **void** windowIconified(WindowEvent e)

**public** **void** windowDeiconified(WindowEvent e)

**public** **void** windowActived(WindowEvent e)

**public** **void** windowDeactivated(WindowEvent e)

与WindowListener对应的适配器为WindowAdapter。

代码：

TestWindowClosing\_1.java

/\*

2013年7月30日11:09:25

创建一个Windows窗口，点击关闭按钮时可以关闭它

利用外部监听器类来实现 方法一

\*/

**import** java.awt.\*;

**import** java.awt.event.\*;

**public** **class** TestWindowClosing\_1

{

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

{

Frame f = **new** Frame();

f.setBounds(100, 100, 200, 200);

f.setBackground(Color.*GREEN*);

f.addWindowListener(**new** MyWindowMonitor());

f.setVisible(**true**);

}

}

**class** MyWindowMonitor **extends** WindowAdapter

{

@Override

**public** **void** windowClosing(WindowEvent e)

{

Frame f = (Frame)e.getSource();

f.setVisible(**false**); //不能直接写 e.setVisible(false); 因为WindowEvent 是个事件类，事件类中肯定是不会有setVisible()方法的， 另外可以参见“E:\编程\Java\java心得\GUI\ActionEvent 和 WindowEvent 区别.java\ActionEvent 和 WindowEvent 区别.java”

System.*exit*(-1);

}

}

执行结果：

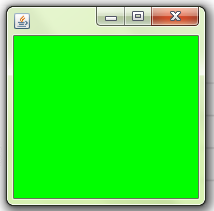


图12-3-7

代码：

TestWindowClosing\_2.java

**import** java.awt.\*;

**import** java.awt.event.\*;

**public** **class** TestWindowClosing\_2

{

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

{

Frame f = **new** Frame(); //13行

f.setBounds(100, 100, 200, 200);

f.setBackground(Color.*GREEN*);

f.addWindowListener(**new** MyWindowMonitor());

f.setVisible(**true**);

}

**static** **class** MyWindowMonitor **implements** WindowListener //去掉了static 编译时就会报错, 因为“要想在一个类的静态方法中访问该类的内部类，则该内部类必须的被标记为static”, 另外 implements WindowListener 改为 extends WindowAdapter更能减少代码的编写

{

@Override

**public** **void** windowClosing(WindowEvent e)

{

Frame f = (Frame)e.getSource();

f.setVisible(**false**);

System.*exit*(-1);

}

**public** **void** windowOpened(WindowEvent e){}; //{}; public 都不能省

**public** **void** windowClosed(WindowEvent e){};

**public** **void** windowIconified(WindowEvent e){};

**public** **void** windowDeiconified(WindowEvent e){};

**public** **void** windowActivated(WindowEvent e){};

**public** **void** windowDeactivated(WindowEvent e){};

}

}

执行结果：

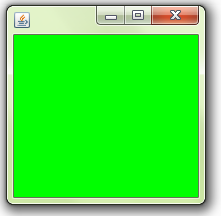


图12-3-8

代码：

TestWindowClosing\_3.java

/\*

2013年7月30日11:14:46

创建一个Windows窗口，点击关闭按钮时可以关闭它

利用方法内部的匿名类来实现 方法三

\*/

**import** java.awt.\*;

**import** java.awt.event.\*;

**public** **class** TestWindowClosing\_3

{

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

{

Frame f = **new** Frame(); //13行

f.setBounds(100, 100, 200, 200);

f.setBackground(Color.*GREEN*);

f.addWindowListener(**new** WindowAdapter()

{

**public** **void** windowClosing(WindowEvent e)

{

Frame f = (Frame)e.getSource(); //22行

//第一： 22行的代码不能改为 e.setVisible(false); 编译时会报错 因为WindowEvent 是个事件类，事件类中肯定是不会有setVisible()方法的， 另外可以参见“E:\编程\Java\java心得\GUI\ActionEvent 和 WindowEvent 区别.java\ActionEvent 和 WindowEvent 区别.java”

f.setVisible(**false**);

System.*exit*(-1);

}

}

); //16行

f.setVisible(**true**);

}

}

执行结果：

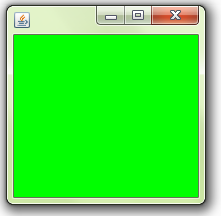


图12-3-9

## 12.4 内部类

### 内部类定义

在A类的内部但是所有方法的外部定义了一个B类，则B类就是A类的内部类，A是B的外部类

### 内部类访问原则

内部类的方法可以访问外部类所有的成员

外部类的方法不可以直接访问内部类的成员

### 内部类的优点

可以让一个类方便的访问另一个类中的所有成员

增加程序的安全性，有效避免其他不相关类对该类的访问

### 何时使用内部类

如果一个A类要使用B类的所有成员，并且A类不需要被除B类以外的其他类访问，则我们应当把A类定义为B类的内部类

代码：

Test.java

**class** A{

**private** **int** i = 1;

**private** **void** f(){

System.*out*.println("哈哈");

}

//B是A的内部类

**class** B{

**public** **int** bi = 10;

//非静态的内部类可以访问外部类所有的成员

**public** **void** g(){

System.*out*.printf("i = %d\n", i);

f();

}

}

**public** **void** k(){

//g(); //error 外部类的方法不可以直接访问内部类的成员

//System.out.println(bi); //error 外部类的方法不可以直接访问内部类的成员

B bb = **new** B();

bb.g();

}

}

**public** **class** Test{

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

A aa = **new** A();

aa.k();

}

}

执行结果：



图12-4-1

本程序证明了：

外部类的方法不可以直接访问内部类的成员

内部类的方法却可以访问外部类的所有成员

### 内部类举例

编程实现：两个文本框内容相加所得结果赋给第三个文本框



图12-4-2

代码：

TestTextField\_1.java

/\*

2013年7月30日11:27:11

文本框内容相加方法一：

主函数代码过多

程序逻辑混乱

不推荐

\*/

**import** java.awt.\*;

**import** java.awt.event.\*;

**public** **class** TestTextField\_1

{

**public** **static** TextField *tf1*, *tf2*, *tf3*;

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

{

Frame f = **new** Frame();

*tf1* = **new** TextField(30);

*tf2* = **new** TextField(30);

*tf3* = **new** TextField(30);

Button bn = **new** Button("=");

Label lb = **new** Label("+");

f.setLayout(**new** FlowLayout());

f.add(*tf1*);

f.add(lb);

f.add(*tf2*);

f.add(bn);

f.add(*tf3*);

bn.addActionListener(**new** MyMonitor());

f.pack();

f.setVisible(**true**);

}

}

**class** MyMonitor **implements** ActionListener

{

**public** **void** actionPerformed(ActionEvent e)

{

**int** num1 = Integer.*parseInt*(TestTextField\_1.*tf1*.getText());

**int** num2 = Integer.*parseInt*(TestTextField\_1.*tf2*.getText());

**int** num3 = num1 + num2;

TestTextField\_1.*tf3*.setText(num3+"");

}

}

代码：

TestTextField\_2.java

/\*

2013年7月30日11:31:22

文本框内容相加方法二：

通过在B类中定义A类的属性，就可以达到在B类访问A类成员的目的

但是通过这种方式无法访问A类私有成员

本方式既繁琐又有局限性，不推荐

\*/

**import** java.awt.\*;

**import** java.awt.event.\*;

**public** **class** TestTextField\_2

{

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

{

**new** TF().launch();

}

}

**class** TF

{

**public** TextField tf1, tf2, tf3;

**public** **void** launch()

{

Frame f = **new** Frame();

tf1 = **new** TextField(30);

tf2 = **new** TextField(30);

tf3 = **new** TextField(30);

Button bn = **new** Button("=");

Label lb = **new** Label("+");

f.setLayout(**new** FlowLayout());

f.add(tf1);

f.add(lb);

f.add(tf2);

f.add(bn);

f.add(tf3);

bn.addActionListener(**new** MyMonitor(**this**));

f.pack();

f.setVisible(**true**);

}

}

**class** MyMonitor **implements** ActionListener

{

**public** TF tf = **null**;

**public** MyMonitor()

{

}

**public** MyMonitor(TF tf)

{

**this**.tf = tf;

}

**public** **void** actionPerformed(ActionEvent e)

{

**int** num1 = Integer.*parseInt*(tf.tf1.getText());

**int** num2 = Integer.*parseInt*(tf.tf2.getText());

**int** num3 = num1 + num2;

tf.tf3.setText(num3 + "");

}

}

代码：

TestTextField\_3.java

/\*

2013年7月30日11:33:31

文本框内容相加方法三:

把B当做A的内部类，则我们就可以在B类中轻松访问A类的所有成员，当然包括私有成员

推荐

\*/

**import** java.awt.\*;

**import** java.awt.event.\*;

**public** **class** TestTextField\_3

{

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

{

**new** TF().launch();

}

}

**class** TF

{

**public** TextField tf1, tf2, tf3;

**public** **void** launch()

{

Frame f = **new** Frame();

tf1 = **new** TextField(30);

tf2 = **new** TextField(30);

tf3 = **new** TextField(30);

Button bn = **new** Button("=");

Label lb = **new** Label("+");

f.setLayout(**new** FlowLayout());

f.add(tf1);

f.add(lb);

f.add(tf2);

f.add(bn);

f.add(tf3);

bn.addActionListener(**new** MyMonitor());

f.pack();

f.setVisible(**true**);

}

**class** MyMonitor **implements** ActionListener

{

**public** **void** actionPerformed(ActionEvent e)

{

**int** num1 = Integer.*parseInt*(tf1.getText());

**int** num2 = Integer.*parseInt*(tf2.getText());

**int** num3 = num1 + num2;

tf3.setText(num3 + "");

}

}

}

## 12.5 匿名类

匿名类是一种特殊的内部类

如果在一个方法内部定义了一个匿名类，则该匿名类可以访问：

1 外部类的所有成员

2 包裹本匿名类的方法中的所有final类型的局部变量，非fianl类型的局部变量无法被匿名类访问

代码：

TestWindow.java

**import** java.awt.\*;

**import** java.awt.event.\*;

**public** **class** TestWindow

{

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

{

//如果把10行的final去掉，本程序在18行就会报错,因为匿名类只能访问包裹它的方法中的final类型的局部变量

**final** Frame f = **new** Frame(); //10行

f.setSize(200, 200);

f.addWindowListener(

**new** WindowAdapter()

{

**public** **void** windowClosing(WindowEvent e)

{

f.setVisible(**false**);//18行

System.*exit*(-1);

}

}

);

f.setVisible(**true**);

}

}

执行结果：

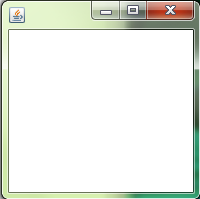


图12-5-1

### 创建匿名类的三种方式

实现接口

实现抽象类

继承父类

#### 创建匿名类之实现接口

假设A是接口名

格式：

new A()

{

实现接口中方法的代码

}

功能：

生成了一个实现A接口的匿名类

代码：

TestAnonyClass\_1.java

/\*

假设Ａ是接口名

new A()

{

实现接口中方法的代码

};

功能：生成一个实现了A接口的匿名类

\*/

**interface** It

{

**void** f();

//void g(); // 如果该语句生效的话，21行到27行的代码中就必须的实现g方法

}

**public** **class** TestAnonyClass\_1

{

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

{

It it = **new** It()

{ //21行

**public** **void** f()

{

System.*out*.println("哈哈");

}

}; //27行

//error

// It it = new It1()

// ( //是{ 不是(

// public void f()

// {

// System.out.println("哈哈");

// }

// ); // 是} 不是)

//error

// It it = new It()

// (

// {

// public void f()

// {

// System.out.println("哈哈");

// }

// }

// ;

it.f();

}

}

执行结果：



图12-5-2

#### 创建匿名类之实现抽象类

假设A是抽象类

格式：

new A()

{

实现了A类的所有抽象类的方法代码

添加自己的方法或属性代码【不建议，因为没有实际意义】

}

功能：

生成一个匿名类，该匿名类必须得实现了A类的所有抽象方法，当然该匿名类也可以定义自己的属性和方法

代码：

TestAnonyClass\_2.java

/\*

假设A是抽象类

new A()

{

实现了A类的所有抽象类的方法代码

添加自己的方法或属性代码【不建议，因为没有实际意义】

}

功能： 生成一个匿名类，该匿名类必须得实现了A类的所有抽象方法，

当然该匿名类也可以定义自己的属性和方法

\*/

**abstract** **class** A

{

**abstract** **public** **void** f();

**public** **void** g()

{

}

}

**public** **class** TestAnonyClass\_2

{

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

{

A aa = **new** A()

{

//f方法不可以被注释掉, 因为f是抽象方法，匿名类必须的实现

**public** **void** f()

{

System.*out*.println("FFFF");

}

//g方法可以被注释掉

**public** **void** g()

{

System.*out*.println("GGGG");

}

**public** **void** k()

{

}

};

aa.f();

aa.g();

//aa.k();

}

}

/\*

在JDK 1.6中的运行结果是：

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

FFFF

GGGG

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

\*/

执行结果：



图12-5-3

#### 创建匿名类之继承父类

假设A是个类名

格式：

new A()

{

重写了A类的方法代码

添加自己的属性和方法【不建议，因为没有实际意义】

}

功能：

生成一个A类的子类对象，该匿名类对象继承了A的所有非private成员

代码：

TestAnonyClass\_3.java

/\*

假设A是个类名

new A()

{

重写了A类的方法代码

自己添加的属性和方法【【不建议，因为没有实际意义】】

}

功能： 生成一个A类的子类对象，该匿名类对象继承了A的所有非private成员

\*/

**class** A

{

**public** **void** f()

{

System.*out*.println("哈哈");

}

}

**public** **class** TestAnonyClass\_3

{

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

{

A aa = **new** A()

{

//重写了父类A的方法

**public** **void** f()

{

System.*out*.println("嘿嘿");

}

//可以定义自己的方法，但没有任何实际意义

**public** **void** g()

{

System.*out*.println("GGGG");

}

};

aa.f();

//aa.g(); //error

}

}

/\*

在JDK 1.6中的运行结果是：

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

FFFF

GGGG

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

\*/

执行结果：



图12-5-4

### 匿名类的优缺点

如果一个类的语句比较少，逻辑比较简单，而且不经常变动，这个时候可以使用匿名类！

如果一个雷包含了很重要的逻辑，将来要经常修改，则这个类就不应该做匿名类来使用，匿名类会导致代码的混乱。

### 练习题：设计一个计算器预备知识

假设bn是个Button对象，单击bn按钮时，bn对象会自动产生ActionEvent对象，并把ActionEvent对象发送给监听器中。

void actionPerformed(ActionEvent e) 方法的形参e

注意：bn会自动把它自己的标签字符串的信息也封装到ActionEvent对象中，但是如果程序员修改了bn中的属性”actionCommand”的值，则bn就不再把它自己标签字符串的信息封装到ActionEvent对象中，而是把属性”actionCommand”的值封装到ActionEvent对象中。

代码：

TestMonitor\_1.java

/\*

假设bn 是个Button对象

单击bn按钮时，bn对象会自动产生ActionEvent对象， 并把ActionEvent对象发送给监听器中

void actionPerformed(ActionEvent e)

方法的形参e

注意：bn会自动把它自己标签字符串的信息也封装到ActionEvent对象中

但是如果程序员修改了bn中属性"actionCommand"的值，则bn就不再把它自己标签字符串的信息

封装到ActionEvent对象中，而是把属性"actionCommand"的值封装到ActionEvent对象中

\*/

**import** java.awt.\*;

**import** java.awt.event.\*;

**public** **class** TestMonitor\_1

{

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

{

Frame f = **new** Frame();

f.setLayout(**new** BorderLayout());

Button bn1 = **new** Button("Start!");

bn1.addActionListener( **new** MyMonitor() ) ;

Button bn2 = **new** Button("End!");

bn2.setActionCommand("Game Over!");

bn2.addActionListener(**new** MyMonitor());

f.add(bn1, BorderLayout.*NORTH*);

f.add(bn2, BorderLayout.*SOUTH*);

f.pack();

f.setVisible(**true**);

}

}

**class** MyMonitor **implements** ActionListener

{

**public** **void** actionPerformed(ActionEvent e)

{

System.*out*.println(e.getActionCommand()); //e.getActionCommand() 返回的是e事件所对应的事件源按钮的actionCommand 的值，但如果事件源按钮的actionCommand 的值为null，则返回事件源按钮的标签字符串的值,事件源按钮的标签字符串的值可以通过下面两行代码的道

//获取事件源按钮的标签字符串的值

Button bn = (Button)e.getSource();

System.*out*.println(bn.getLabel()); //不能直接写e.getLabel(); 因为ActionEvent类中是没有getLabel()方法的

}

}

执行结果：

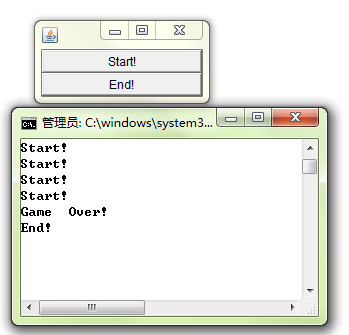


图12-5-5

Button类中有如下方法：

**public** String getLabel()

// getLabel()返回的一定是我们在显示器上能够看到的按钮上的信息，即返回的是标签字符串的信息，如果按钮的标签字符串为空，则，getLabel()返回null

**public** String getActionCommand()

// 返回此按钮激发的动作事件的命令名称。如果没有以前通过setActionCommand()方法设置动作事件的命令名称，则本方法返回按钮的标签字符串的信息。

**public** **void** setActionCommand(String command)

// 设置此按钮激发的动作事件的命令名称。如果没有设置动作事件的命令名称，则动作事件的命令名称默认值与按钮标签字符串的值一样。

代码：

TestButton\_1.java

/\*

Button类

1、 getLabel() 返回的一定是我们在显示器上能够可看到的按钮上的信息,

即返回的是标签字符串的信息，如果按钮的标间字符串为空，则getLabel()返回null

2、 getActionCommand()返回的是ActionCommand属性的值，如果ActionCommand属性值为null，

则返回按钮标签字符串的信息

附注： 假设bn 是个Button对象

单击bn按钮时，bn对象会自动产生ActionEvent对象， 并把ActionEvent对象发送给监听器中

void actionPerformed(ActionEvent e)

方法的形参e

注意：bn会自动把它自己标签字符串的信息也封装到ActionEvent对象中,但是如果程序员修改了bn中属性"actionCommand"的值，则bn就不再把它自己标签字符串的信息封装到ActionEvent对象中，而是把属性"actionCommand"的值封装到ActionEvent对象中

\*/

**import** java.awt.\*;

**public** **class** TestButton\_1

{

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

{

Frame f = **new** Frame();

Button bn = **new** Button("哈哈");

String str = bn.getLabel();

System.*out*.println(str);

str = bn.getActionCommand();

System.*out*.println(str);

bn.setActionCommand("嘻嘻");

str = bn.getActionCommand();

System.*out*.println(str);

str = bn.getLabel();

System.*out*.println(str);

}

}

执行结果：



图12-5-6

代码：

TestButton\_2.java

/\*

按钮Button对象中有getActionCommand()方法，

ActionEvent对象中也有getActionCommand()方法，

两者都可以获得按钮激发的动作事件的命令名称

\*/

**import** java.awt.\*;

**import** java.awt.event.\*;

**class** MyMonitor **implements** ActionListener

{

@Override

**public** **void** actionPerformed(ActionEvent e){

System.*out*.println( e.getActionCommand() + ": 通过对Button执行操作的的事件对象获取的" );

}

}

**public** **class** TestButton\_2{

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

Frame f = **new** Frame();

Button bn = **new** Button("点我啊");

bn.addActionListener(**new** MyMonitor());

System.*out*.println(bn.getActionCommand() + ": 通过Button对象获取的");

f.add(bn);

f.pack();

f.setVisible(**true**);

}

}

执行结果：

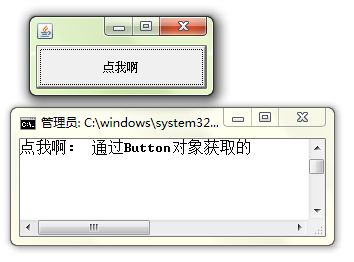


图12-5-7

### 练习题：设计一个计算器

变成设计出如下的计算器，并可参与计算

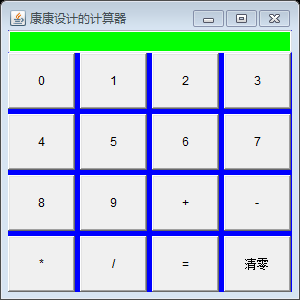


图12-5-8

代码：

TestCalculate.java

**import** java.awt.\*;

**import** java.awt.event.\*;

**public** **class** TestCalculate

{

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

{

**new** Calculate().launch();

}

}

**class** Calculate **extends** Frame

{

**private** **boolean** firstFlag = **true**;

**private** String str1 = "";

**private** String str2 = "";

**private** TextField tf = **null**;

**private** String strOper = **null**; //18行 strOper用来记录到底是+还是-还是\*还是/操作,因为我们点击"="按钮时是无法通过该事件得到上次我们执行的到底是+还是-还是\*还是/操作，所以我们需要定义一个属性strOper来记录单击"="之前的上次执行的操作

**public** **void** launch()

{

setTitle("康康设计的计算器");

tf = **new** TextField(30);

tf.setBackground(Color.*GREEN*);

//setLayout(new BorderLayout()); //Frame默认的布局管理器就是BorderLayout

add(tf, BorderLayout.*NORTH*);

Panel p = **new** Panel(**new** GridLayout(4, 4, 5, 5));

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

{

Button bn = **new** Button("" + i);

bn.setActionCommand("数字");

p.add(bn);

bn.addActionListener(**new** MyMonitor());

}

Button bnAdd = **new** Button("+");

p.add(bnAdd);

bnAdd.setActionCommand("算术操作");

bnAdd.addActionListener(**new** MyMonitor());

Button bnSub = **new** Button("-");

p.add(bnSub);

bnSub.setActionCommand("算术操作");

bnSub.addActionListener(**new** MyMonitor());

Button bnMult = **new** Button("\*");

bnMult.addActionListener(**new** MyMonitor());

p.add(bnMult);

bnMult.setActionCommand("算术操作");

Button bnDiv = **new** Button("/");

p.add(bnDiv);

bnDiv.setActionCommand("算术操作");

bnDiv.addActionListener(**new** MyMonitor());

Button bnEq = **new** Button("=");

p.add(bnEq);

bnEq.setActionCommand("=");

bnEq.addActionListener(**new** MyMonitor());

Button bnClear = **new** Button("清零");

p.add(bnClear);

bnClear.setActionCommand("清零");

bnClear.addActionListener(**new** MyMonitor());

p.add(bnAdd);

p.add(bnSub);

p.add(bnMult);

p.add(bnDiv);

p.add(bnEq);

p.add(bnClear);

add(p, BorderLayout.*CENTER*);

setBounds(300, 300,300, 300);

setBackground(Color.*BLUE*);

setVisible(**true**);

addWindowListener( **new** WindowAdapter()

{

@Override

**public** **void** windowClosing(WindowEvent e)

{

System.*exit*(-1);

}

}

);

}

**class** MyMonitor **implements** ActionListener

{

**public** **void** actionPerformed(ActionEvent e)

{

//如果是对=按钮的点击操作，则我们需要得到上次动作命令名称的信息，因为本次事件中只能得到按钮标签的信息= 和 按钮动作命令名称=， 无法得到上次执行的到底是+还是-还是\*还是/操作，所以我们需要单独定义一个属性用来记录动作命令名称的信息，参见18行代码

/\*actionPerformed()方法算法如下：

第一： 我们必须先得通过e得到按钮的动作命令名称strAc和按钮标签字符串的

信息strLab

第二： 然后通过判断按钮的动作命令名称strAc来执行不同的操作，即：

如果strAc是"数字"，即如果是对0-9按钮的单击操作， 则更新str1 或者 str2的值

如果strAc是"算术操作"，即如果是对"+" "-" "\*" "/"按钮的单击操作，则我们必须得把按钮标签字符窜strLb的值赋给属性strOper, 注意不能把strAc赋给strOper, 因为此时strAc的值是"算术操作"

如果strAc是"="， 则。。。。。。略

如果strAc是"清零",则。。。。。。略

\*/

String strAc = e.getActionCommand();

Button bn = (Button)e.getSource();

String strLb = bn.getLabel();

**if** (strAc.equals("数字"))

{

**if** (firstFlag)

{

str1 = str1 + strLb; //103

tf.setText(str1);

}

**else**

{

str2 = str2 + strLb; //109

tf.setText(str2);

}

}

**else** **if** (strAc.equals("算术操作")) //如果是+ - \* / 操作

{

strOper = strLb; //118 千万不要写成了 strOper = strAc; 因为本语句要被执行的话，此时strAc的值一定是"算术操作", 我们需要的是+ - \* / 即我们需要的是按钮标签字符窜的信息

firstFlag = **false**;

}

**else** **if** (strAc.equals("="))

{

result();

}

**else** **if** (strAc.equals("清零"))

{

str1 = str2 = "";

tf.setText("");

firstFlag = **true**;

}

}

}

**public** **void** result()

{

**double** num1 = Double.*parseDouble*(str1);

**double** num2 = Double.*parseDouble*(str2);

**double** num3 = 0;

**if** (strOper.equals("+"))

{

num3 = num1 + num2;

}

**else** **if** (strOper.equals("-"))

{

num3 = num1 - num2;

}

**else** **if** (strOper.equals("\*"))

{

num3 = num1 \* num2;

}

**else** **if** (strOper.equals("/"))

{

num3 = num1 / num2;

}

tf.setText("" + num3);

str1 = String.*valueOf*(num3); //不能写成了 str1 = str2;

str2 = "";

//firstFlag = false; //本行代码可以省略

}

}

## 12.6 重绘

我们在窗口中画的图形是不会立即在窗口显示的，除非整个窗口得到了重绘

整个窗口被改变时，系统会自动调用**public** **void** paint(Graphics g)方法，窗口改变包括：整个窗口第一次显示在显示器上，窗口大小被改变，如最大最小化，窗口被遮盖后又被显示。

代码：

TestPaint.java

**import** java.awt.\*;

**public** **class** TestPaint {

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

**new** PaintFrame().launchFrame();

}

}

**class** PaintFrame **extends** Frame {

**public** **void** launchFrame() {

setBounds(200,200,640,480);

setVisible(**true**);

}

//当Frame需要重画时，paint方法就会被自动调用, 重画就是重新显示时，重新画出来

//所谓重画包括：Ｆｒａｍｅ对象第一次在显示器上被显示出来的时候，Ｆｒａｍｅ窗口被其他窗口盖住但又变成当前窗口的时候，改变窗口大小的时候

**public** **void** paint(Graphics g) { //该方法的原型是: public void paint(Graphics g); paint()是Container类的方法，当容器对象被重画时，paint方法就会被容器对象自动被调用, Frame对象建立之后，它内部就有一个画笔了，一旦Frame对象所代表的窗口被重画，Frame对象就会自动的把它自己的画笔发送给paint()方法的形参, 我们所要做的就是重写paint()方法就可以了

//形参 (Graphics g) 千万不要生，否则就不是重写Frame的paint方法了

Color c = g.getColor(); //18行 g就是画笔，g.getColor()就是得到当前画笔的颜色

g.setColor(Color.*RED*); //设置画笔为红色

g.fillOval(50, 50, 500, 30); //填充实心的椭圆 前两个参数表示左上顶点的坐标， 后两个参数表示宽度和高度，这4个参数合起来表示由这4个参数所确定的长方形中的最大内置椭圆, 当然如果后两个参数的值一样的化，那就变成了一个内置圆了，要知道圆只不过是椭圆的一种特殊形式而已嘛 O(∩\_∩)O哈哈~

//drawOval() 绘制空心椭圆 fillOval()绘制实心椭圆,实心椭圆的颜色是由g.getColor()确定的

g.setColor(Color.*green*);

g.fillRect(80,80,40,40); //前两个参数表示左上顶点的坐标

g.drawLine(300, 300, 400, 400); //这是一根很细的线，

g.fillRect(100, 400, 400, 5); //把实心矩形的高度设低一点就变成了一根粗线了

g.setColor(c); //24行 18行：这只画笔在使用之前就已经有颜色了，所以在修改颜色之前最好把以前的颜色保存起来 24行： 再恢复到以前的颜色 这是良好的编程

}

}

执行结果：

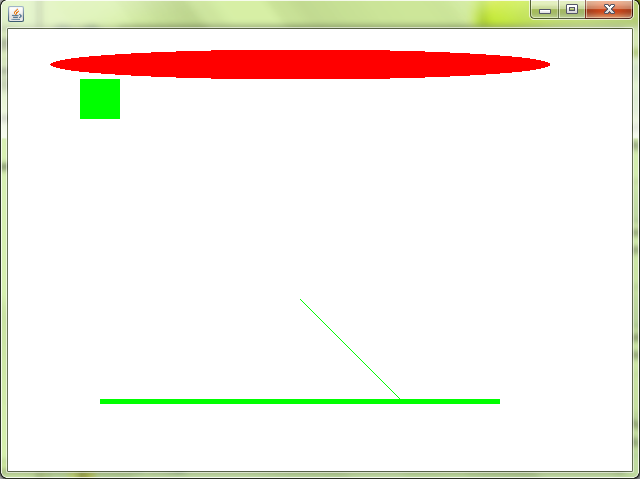


图12-6-1

## 附：可运行jar包生成步骤

1. 新建一个记事本文件，假设1.txt，文件内容：

Main-Class:可运行类的名字

附注：记得敲回车

1. dos命令：

jar cvfm haha.jar 1.txt \*.class

记住：只有GUI程序生成class文件才可以作为main-class

# 13 流

## 13.1 流概述

### 流与类的关系

如果一个类是用作输入输出，则这个类有一个新的名字叫流。流一定是类，但类不一定是流

原始流与包裹流关系

### 流的定义

流就是程序和设备之间嫁接起来的一根用于数据传输的管道，这个管道上有很多按钮，不同的按钮可以实现不同的功能，这根管道就是**流**。

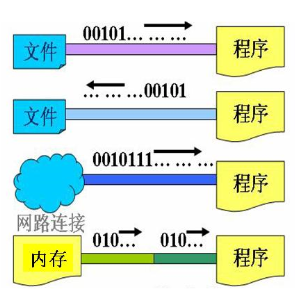


图13-1-1

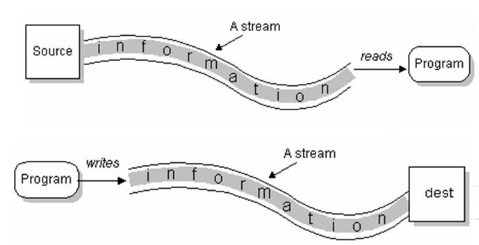


图13-1-2

输入时，程序在源（文件，网络，内存）上打开一个stream，然后如图一个一个顺序读。写也是一样。

### 流的分类和使用

1 四大基本抽象流

java.io包中定义了多个流类型（类或抽象类）来实现输入或输出功能：可以从不同的角度对其进行分类：

按数据流的方向不同可以分为输入流和输出流

按处理数据单位不同可以分为字节流和字符流

按照功能不同可以分为节点流和处理流

J2SDK所提供的所有流类型位于包java.io内都分别继承自以下四种抽象流类型。

|  |  |  |
| --- | --- | --- |
|  | 字节流 | 字符流 |
| 输入流 | InputStream | Reader |
| 输出流 | OutputStream | Writer |

表13-1-1

2 节点流和处理流

节点流为可以从一个特定的数据源（节点）读写数据（如：文件，内存），节点流也称原始流。

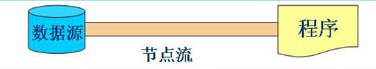


图13-1-3

处理流是“连接”在已存在的流（节点流或处理流）之上，通过对数据的处理为程序提供更为强大的读写功能。处理流也称包裹流

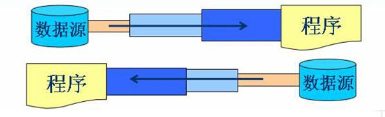


图13-1-4

文件流

缓冲流

转换流

数据流

Print流

Object流

## 13.2 四大基本抽象流

InputStream和OutputStream读写数据的单位是一个字节。

Reader和Writer读写数据的单位是一个字符

在Java中一个字符占两个字节

见 [流表格](#流表格)。

InputStream OutputStream Reader Writer这四个类都是抽象类

凡是以Stream结尾的都是字节流

### InputStream流中常用的方法

**public** **int** read() **throws** IOException

// 读取一个字节并以整数形式返回

// 如果读取到输入流的末尾则返回-1

**public** **int** read(**byte**[] b) **throws** IOException

// 从输入流中读取一定数量的字节，并将其存储在缓冲区数组b中。以整数形式返回实际读取的字节书

// 如果b的长度为0，则不读取任何字节并返回0；如果因为流位于文件末尾而没有可用的字节，则返回-1；

// 例子：

FileInputStream fis = **new** FileInputStream("D:\\share\\errorlog.txt");

len = fis.read(buf);

// 从fis流所关联的D:\share\errorlog.txt文件中读取数据，并将读取出来的数据写入buf数组中，返回值是实际写入buf数组的字节个数，如果读取到文件的结尾，则返回-1.

**public** **int** read(**byte**[] b, **int** off, **int** len) **throws** IOException

// 从输入流中最多读取len个字节的数据并存入byte数组中

// b表示读取的数据要存入的数组的名字

// off表示第一个读出的数据要存入的位置，是下标

// len表示最多能读取的字节数

// 将从输入流所关联到的设备中读取的第一个字节存储在元素b[off]中，下一个字节存储在b[off+1]中，但读取的字节也可能小于该值。以整数形式返回实际读取的字节数。

// 如果读取到了文件的末尾，则返回-1

**void** close() **throws** IOException

// 关闭此输入流并释放与该流关联的所有系统资源

**long** skip(**long** n) **throws** IOException

// 跳过和丢弃此输入流中数据的n个字节。[这个用的很少]

### OutputStream流中常用的方法

// 向输出流中写入一个字节数据，该字节数据为参数b的低8位

**void** write(**int** b) **throws** IOException

// 将一个字节类型的数组中的数据写入输出流

**void** write(**byte**[] b) **throws** IOException

// 将一个字节类型的数组中的从指定位置（off）开始的len歌字节写入到输出流

**void** write(**byte**[] b, **int** off, **int** len) **throws** IOException

// 关闭流释放内存资源

**void** close() **throws** IOException

// 将输出流中缓冲的数据全部写出到目的地

**void** flush() **throws** IOException

### Reader流的常用方法

// 读取一个字符并以整数的形式返回（0~255），如果返回-1表示已到输入流的末尾。

**int** read() **throws** IOException

// 读取一系列字符并存储到一个数组buffer，返回实际读取的字符数，如果读取前已到输入流的末尾返回-1.

**int** read(**char**[] cbuf) **throws** IOException

// 最多读取length个字符，并存储到一个数组buffer，从length位置开始，返回实际读取的字符数，如果读取前已到输入流的末尾返回-1.

**int** read(**char**[] cbuf, **int** offset, **int** length) **throws** IOException

// 关闭流释放内存资源

**void** close() **throws** IOException

// 跳过n歌字符不读，返回实际跳过的字节数

**long** skip(**long** n) **throws** IOException

### Writer流中的常用方法

// 向输出流中写入一个字符数据，该字节数据为参数b的低16位

**void** write(**int** c) **throws** IOException

// 将一个字符类型的数组中的数据写入输出流

**void** write(**char**[] cbuf) **throws** IOException

// 将一个字符类型的数组中的从指定位置（offset）开始的length个字符写入到输出流

**void** write(**char**[] cbuf, **int** offset, **int** length) **throws** IOException

// 将一个字符串中的字符写入到输出流

**void** write(String string) **throws** IOException

// 将一个字符串从offset开始的length个字符写入到输出流

**void** write(String string, **int** offset, **int** length) **throws** IOException

// 关闭流释放内存资源

**void** close() **throws** IOException

// 将输出流中缓冲的数据全部写出到目的地

**void** flush() **throws** IOException

## 13.3 文件流

文件流包括

FileInputStream FileOutputStream --字节流

FileReader FileWriter --字符流

例子：读取一个文件的内容并将其输出到显示器上，并统计读取出来的字节的个数

代码：

TestFileReader.java

/\*

2013年7月30日20:42:48

利用FileReader流来读取一个文件中的数据，并在显示器上输出!

\*/

**import** java.io.\*;

**public** **class** TestFileReader

{

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

{

FileReader fr = **null**;

**try**

{

fr = **new** FileReader("C:\\Users\\Administrator\\Desktop\\TestFileReader.java");

**int** cnt = 0;

**int** ch;

**while** (-1 != (ch=fr.read())) //20行

{

System.*out*.print((**char**)ch); // System.out.print(int ch); 这是在显示器上输出ch的整数值，所以必须的进行类型转化，我们需要输出的是ch所代表的整数对应的字符

++cnt;

}

System.*out*.printf("总共从TestFileReader.java文件中读取了%d个字符", cnt);

}

**catch** (FileNotFoundException e)

{

System.*out*.println("找不到文件!");

System.*exit*(-1);

}

**catch** (IOException e)

{

System.*out*.println("文件读取失败!");

System.*exit*(-1);

}

}

}

执行结果：

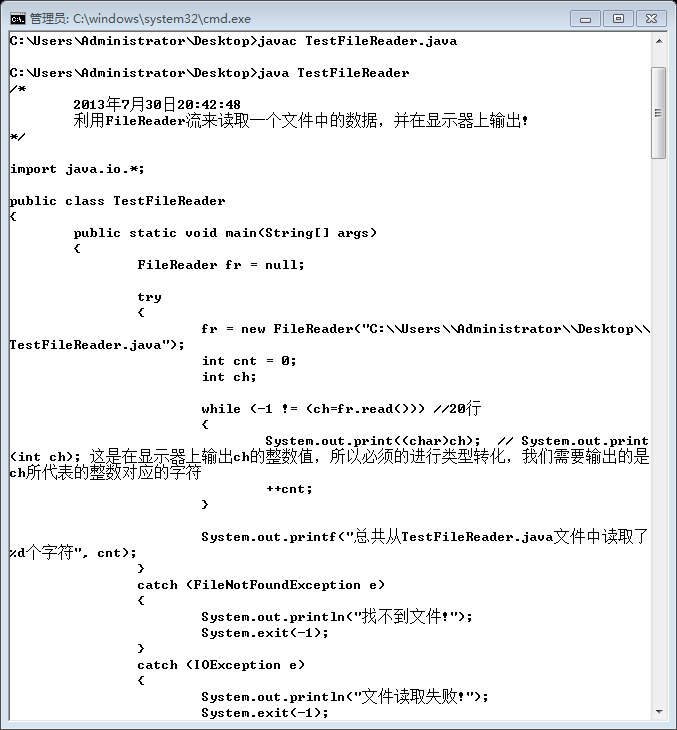


图13-3-1

### FileInputStream的使用

InputStream是用来读取字节的，是个抽象类，我们通常使用的是该类的子类

FileInputStream是InputStream的子类，利用FileInputStream可以将一个文件的内容按字节为单位读取出来

FileInputStream有一个很常用的构造函数**public** FileInputStream(String fileName) **throws** FileNotFoundException，利用该构造函数可以实现将输入流连接到某个文件的功能，必须对本构造函数抛出的异常进行捕捉，如果用字符串来表示操作系统的文件路径时，我们可以使用\\和/两种方式来做为文件夹的路径分隔符

FileOutputStream同理，不再赘述

### FileReader的使用

Reader是用来读取字符的，是个抽象类，我们通常使用的是该类的子类

FileReader是Reader的子类，利用FileReader可以将一个文件的内容以字符为单位读取出来

FileReader有一个很常用的构造函数 **public** FileReader(String fileName) **throws** FileNotFoundException利用该构造函数可以实现将输入流连接到某个文件的功能，必须对本构造函数抛出的异常进行捕捉，如果用字符串来表示操作系统的文件路径时，我们可以使用\\和/两种方式来做为文件夹的路径分隔符

FileWriter同理，不再赘述

### 字节流与字符流区别

FileInputStream和FileOutputStream可以完成所有格式文件的复制

FileReader和FileWriter只可以完成文本文件的复制，却无法完成视频格式文件的复制

因为字节是不需要解码和编码的，将字节转化为字符才存在编码和解码的问题

字节流可以从所有格式的设备中读写数据，但字符流只能从文本格式的设备中读写数据

### 实例

编程实现文件的复制

代码：

TestFileInputStreamOutputStreamCopy.java

/\*

2013年7月30日21:03:48

利用FileInputStream 和 FileOutputStream 可以完成所有格式文件的赋值

因为字节是不需要解码和编码的，将字节转化为字符才存在解码的问题

本程序完成了视频文件的复制

\*/

**import** java.io.\*;

**public** **class** TestFileInputStreamOutputStreamCopy

{

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

{

FileInputStream fi = **null**;

FileOutputStream fo = **null**;

**try**

{

fi = **new** FileInputStream("E:\\综艺\\歌曲\\妹妹来看我.mp3");

fo = **new** FileOutputStream("d:/share/Output.txt"); //Output.txt用暴风影音打开

**int** ch;

**while** (-1 != (ch=fi.read()))

{

fo.write(ch);

}

}

**catch** (FileNotFoundException e)

{

System.*out*.println("文件没有找到!");

System.*exit*(-1);

}

**catch**(IOException e)

{

System.*out*.println("文件读写错误!");

System.*exit*(-1);

}

**finally**

{

**try**

{

**if** (**null** != fi)

{

fi.close();

fi = **null**;

}

**if** (**null** != fo)

{

fo.close();

fo = **null**;

}

}

**catch** (Exception e)

{

e.printStackTrace();

System.*exit*(-1);

}

}

System.*out*.println("文件复制成功!");

}

}

代码：

TestFileReaderWriterCopy.java

/\*

2013年7月30日21:06:04

本程序证明了 FileReader 和 FileWriter 只可以完成文本文件的复制，

却无法完成视频格式文件的复制

\*/

**import** java.io.\*;

**public** **class** TestFileReaderWriterCopy

{

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

{

FileReader fi = **null**;

FileWriter fo = **null**;

**try**

{

fi = **new** FileReader("E:\\综艺\\歌曲\\妹妹来看我.mp3");

fo = **new** FileWriter("d:/share/Output.txt"); //Output.txt用暴风影音打开失败！ 本程序证明了FileWriter 和 FileReader 无法完成视频文件的赋值，实际上FileWriter 和 FileReader 只能完成文本文件的复制

**int** ch;

**while** (-1 != (ch=fi.read()))

{

fo.write(ch);

}

}

**catch** (FileNotFoundException e)

{

System.*out*.println("文件没有找到!");

System.*exit*(-1);

}

**catch**(IOException e)

{

System.*out*.println("文件读写错误!");

System.*exit*(-1);

}

**finally**

{

**try**

{

**if** (**null** != fi)

{

fi.close();

fi = **null**;

}

**if** (**null** != fo)

{

fo.close();

fo = **null**;

}

}

**catch** (Exception e)

{

e.printStackTrace();

System.*exit*(-1);

}

}

System.*out*.println("文件复制成功!");

}

}

## 13.4 缓冲流

缓冲流就是带有缓冲区的输入输出流

缓冲流可以显著的减少我们对IO访问的次数，保护我们的硬盘！！

缓冲流本身就是处理流（处理流也叫包裹流），缓冲流必须得依附于节点流（节点流也叫原始流）

处理流是包裹在原始节点流上的流，相当于包裹在管道上的管道。

缓冲流要“套接”在相应的节点流之上，对读写的数据提供了缓冲的功能，提高了读写的效率，同时增加了一些新的方法。

J2SDK提供了四种缓冲流，其常用的构造方法为：

BufferedReader(Reader in)

BufferedReader(Reader in, **int** sz) // sz为自定义缓冲区的大小

BufferedWriter(Writer out)

BufferedWriter(Writer out, **int** sz)

BufferedInputStream(InputStream in)

BufferedInputStream(InputStream in, **int** size)

BufferedOutputStream(OutputStream out)

BufferedOutputStream(OutputStream out, **int** size)

缓冲输入流支持其父类的mark和reset方法。

BufferedReader提供了readLine方法用于读取一行字符串（以\r或\n分隔）。

BufferedWriter提供了newLine用于写入一个行分隔符。

对于输出的缓冲流，写出的数据会先在内存中缓存，使用flush方法将会使内存中的数据立刻写出。

### BufferedOutputStream和BufferedInputStream

BufferedOutputStream：带缓冲的输出流，允许一次想硬盘写入多个字节的数据。

BufferedInputStream：带缓冲的输入流，允许一次想程序中读入多个字节的数据。

BufferedOutputStream和BufferedInputStream都是包裹流，必须得依附于OutputStream和InputStream。

例子：利用BufferedOutputStream和BufferedInputStream完成大容量文件的复制，这远比单纯利用FileInputStream和FileOutputStream要快得多。

代码：

TestBufferedInputStreamOutputStreamCopy\_1.java

/\*

2013年7月31日9:22:07

利用BufferedOutputStream 和 BufferedInputStream完成大容量文件的复制

这远比单纯利用 FileInputStream 和 FileOutputStream 要快得多

BufferedOutputStream 和 BufferedInputStream 都是包裹流，必须的依附于

OutputStream 和 OutputStream

\*/

**import** java.io.\*;

**public** **class** TestBufferedInputStreamOutputStreamCopy\_1

{

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

{

BufferedOutputStream bos = **null**;

BufferedInputStream bis = **null**;

**try**

{

bos = **new** BufferedOutputStream (

**new** FileOutputStream("e:/OutputView.txt")

); //bos 输出流有个默认的缓冲区，大小为32个字节

bis = **new** BufferedInputStream(

**new** FileInputStream("c:\\[高清在线www.66ys.cn]海底总动员DVD中英字幕.rmvb")

); //bis 输入流有个默认的缓冲区，大小为32个字节

**byte**[] buf = **new** **byte**[1024];

**int** len = bis.read(buf, 0, 1024); //一定要注意，这不是从buf中读数据，而是从bis所关联到的D:\\综艺\\电影\\猫和老鼠\\CD4.rmvb文件中读取数据，并将读取的数据写入bis自己的默认缓冲区中，然后再将缓冲区的内容写入buf数组中，每次最多向buf数组中写入1024个字节，返回实际写入buf数组的字节个数，如果读到了文件的末尾，无法再向buf数组中写入数据，则返回-1

**while** (-1 != len)

{

bos.write(buf, 0, len); //不是写入buf数组，而是将buf数组中下标从0开始的到len-1为止的所有数据写入bos所关联到的"d:/share/OutputView.txt"文件中

len = bis.read(buf); //bis.read(buf); 等价于 bis.read(buf, 0, buf.length);

}

bos.flush();

bis.close();

bos.close();

}

**catch** (FileNotFoundException e)

{

System.out.println("没有找到文件!");

System.exit(-1);

}

**catch** (IOException e)

{

System.out.println("文件读写错误!");

System.exit(-1);

}

System.out.println("文件复制成功!");

}

}

BufferedInputStream流中有public int read(byte[] b)方法用来把从当前流关联到的设备中读取出来的数据存入一个byte数组中

BufferedOutputStream流中有public int write(byte[] b)方法用来把byte数组中的数据输出到当前流所关联到的设备中

如果我们希望用BufferedInputStream和BufferedOutputStream完成“将一个设备中的数据导入另一个设备中”，我们就应该定义一个临时的byte类型的数组，用这个临时数组作为输入流与输出流进行交互的中转枢纽。

BufferedInputStream BufferedOutputStream要比FileInputStream FileOutputStream读写数据的速度快

代码：

TestBufferedInputStreamOutputStreamCopy\_2.java

/\*

本程序读写速度要慢于 "TestBufferedInputStreamOutputStreamCopy.java" 程序

即：

利用BufferedOutputStream 和 BufferedInputStream完成大容量文件的复制

这远比单纯利用 FileInputStream 和 FileOutputStream 要快得多

BufferedOutputStream 和 BufferedInputStream 都是包裹流，必须的依附于

OutputStream 和 OutputStream

\*/

**import** java.io.\*;

**public** **class** TestBufferedInputStreamOutputStreamCopy\_2

{

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

{

FileOutputStream bos = **null**;

FileInputStream bis = **null**;

**try**

{

bos = **new** FileOutputStream("e:/OutputView.txt");

bis = **new** FileInputStream("c:\\[高清在线www.66ys.cn]海底总动员DVD中英字幕.rmvb");

**byte**[] buf = **new** **byte**[1024];

**int** len = bis.read(buf, 0, 1024);

**while** (-1 != len)

{

bos.write(buf, 0, len);

len = bis.read(buf);

}

bos.flush();

bis.close();

bos.close();

}

**catch** (FileNotFoundException e)

{

System.*out*.println("没有找到文件!");

System.*exit*(-1);

}

**catch** (IOException e)

{

System.*out*.println("文件读写错误!");

System.*exit*(-1);

}

System.*out*.println("文件复制成功!");

}

}

### BufferedReader和BufferedWriter

我们只有

BufferedInputStream

BufferedOutputStream

BufferedWriter

BufferedReader

没有

BufferedStream

BufferedFileInputStream（但是有FileOutputStream）

BufferedFileOutputStream（但是有FileInputStream）

BufferedFileReader（但有FileReader）

BufferedFileWriter（但是有FileWriter）

所以前四个流都是包裹流。

Reader FileReader InputStream FileInputStream BufferedInputStream流中都没有readLine方法

DataInputStream流中有readLine方法，但已经被标记为过时

BufferedReader流中有readLine方法，并且该方法是可以正确被使用的

BufferedReader流中有readLine方法，BufferedReader是个要经常使用的流

利用BufferedReader和BufferedWriter可以提高读写文本文件内容的速度

示例：利用BufferedReader和BufferedWriter完成文本文件的复制

代码：

TestBufferedReaderWriterCopy.java

/\*

利用 BufferedReader 和 BufferedWriter 完成文本文件的复制

\*/

**import** java.io.\*;

**public** **class** TestBufferedReaderWriterCopy

{

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

{

BufferedReader br = **null**;

BufferedWriter bw = **null**;

**try**

{

br = **new** BufferedReader(

**new** FileReader("C:\\Documents and Settings\\others\\桌面\\java\\TestBufferedReaderWriterCopy.java")

);

bw = **new** BufferedWriter(

**new** FileWriter("d:/share/Writer.txt")

);

String str = **null**;

**while** (**null** != (str=br.readLine())) //br.readLine()读取一行字符，但会将读取的换行符自动丢弃,即返回的String对象中并不包括换行符

{

bw.write(str);

bw.newLine(); //写入一个换行符 这行不能省

}

bw.flush();

}

**catch** (FileNotFoundException e)

{

e.printStackTrace();

System.*exit*(-1);

}

**catch** (IOException e)

{

e.printStackTrace();

System.*exit*(-1);

}

**finally**

{

**try**

{

bw.close();

br.close();

}

**catch** (IOException e)

{

e.printStackTrace();

System.*exit*(-1);

}

}

}

}

## 13.5 数据流

### DataInputStream

DataInputStream能够以一种与机器无关的方式，直接从底层字节输入流读取Java基本类型和String类型的数据，常用方法包括：

**public** DataInputStream(InputStream in)

**public** **final** **boolean** readBoolean()

**public** **final** **byte** readByte()

**public** **final** **char** readChar()

**public** **final** **double** readDouble()

**public** **final** **float** readFloat()

**public** **final** **int** readInt()

**public** **final** **long** readLong()

**public** **final** **short** readShort()

**public** **final** String readUTF()

DataInputStream是包裹流，必须依附于InputStream

### DataOutputStream

DataOutputStream能够以一种与机器无关的方式，直接将Java基本类型和String类型数据写出到其他的字节输出流。常见的方法有：

**public** DataOutputStream(OutputStream in)

**public** **final** **boolean** writeBoolean()

**public** **final** **byte** writeByte()

**public** **final** **char** writeChar()

**public** **final** **double** writeDouble()

**public** **final** **float** writeFloat()

**public** **final** **int** writeInt()

**public** **final** **long** writeLong()

**public** **final** **short** writeShort()

**public** **final** String writeUTF()

DataOutputStream是包裹流，必须依附于OutputStream

### 数据流实例

例子：

编程实现将long类型数据写入byte数组，然后再从byte数组中把该数据读出来

附注：

这是Socket编程中经常要完成的功能，因为网络编程中经常要把数值型数据存入byte数组中，然后把byte数组打包成数据包（即DatagramPacket），再把数据包经过网络传输到目的机，目的机再从byte数组中把原数值型数据还原回来

本程序要使用到

ByteArrayInputStream

ByteArrayOutputStream

DataInputStream

DataoutputStream

代码：

TestByteArrayOutputStream1.java

/\*

功能：把一个long类型的数据写入byte数组中,然后再从byte数组中读取出

这个long类型的数据

因为网络编程中经常要把数值型数据存入byte数组中然后打包成

DatagramPacket经过网络传输到目的机，目的机再从byte数组中

把原数值型数据还原回来

目的: ByteArrayOutputStream DataOutputStream ByteInputStream DataInputStream 流的使用

记住: DataOutputStream流中的writeLong(long n)是把n变量在内存

中的二进制代码写入该流所连接到的设备中

注意：查API文档得知：

构造 ByteArrayOutputStream 对象时不需要也不能指定缓冲数组，因为缓冲数组默认已经内置好了

构造 ByteArrayInputStream 对象时必须的指定缓冲数组是谁！

\*/

**import** java.io.\*;

**public** **class** TestByteArrayOutputStream1

{

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

{

**long** n = 9876543210L;

ByteArrayOutputStream baos = **new** ByteArrayOutputStream(); //9行 API:"public ByteArrayOutputStream(): 创建一个新的 byte 数组输出流。缓冲区的容量最初是 32 字节，如有必要可增加其大小。 "

//9行代码一旦执行完毕，意味着两点: 1、在内存中生成了一个大小为32个字节的byte数组 2、有一根叫做baos的管道已链接到了该byte数组中，并且可以通过这个管道向该byte数组中写入数据

//虽然此时可以通过baos向baos所连接到的在内存中分配好的byte数组中写入数据，但是ByteArrayOutputStream流并没有提供可以直接把long类型数据直接写入ByteArrayOutputStream流所连接到的byte数组中的方法, 简单说我们没法通过baos向baos所连接到的byte数组中写入long类型的数据, 查API文档可以发现: ByteArrayOutputStream流中并没有类似writeLong()这样的方法，但是DataOutputStream流中却有writeLong() writeFloat()等方法

DataOutputStream dos = **new** DataOutputStream(baos);

dos.writeLong(n); //把n变量所代表的10000L在内存中的二进制代码写入dos所依附的baos管道所连接到的内存中的大小为32字节的byte数组中，由运行结果来看，这是二进制写入，既不是把10000L转化为字符'1' '0' '0' '0' '0'写入byte数组中，而是把10000L在内存中的总共8个字节的二进制代码写入byte数组中

dos.flush();

**byte**[] buf = baos.toByteArray(); //DataOutputStream 流中并没有toByteArray()方法，但是ByteArrayOutputStream 流中却有toByteArray()方法, 所以不可以把baos 改为dos，否则编译时会出错! ByteArrayOutputStream流中toByteArray()方法的含义，摘自API“创建一个新分配的 byte 数组。其大小是此输出流的当前大小，并且缓冲区的有效内容已复制到该数组中”

//利用ByteArrayInputStream 和 DataInputStream 可以从byte数组中得到原long类型的数值10000L

ByteArrayInputStream bais = **new** ByteArrayInputStream(buf);

DataInputStream dis = **new** DataInputStream(bais);

**long** l = dis.readLong();

System.*out*.println("l = " + l);

dos.close();

}

}

/\*

在JDK 1.6中的运行结果是：

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

l = 9876543210

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

\*/

代码：

TestByteArrayOutputStream2.java

/\*

功能:

将long类型数据写入byte数组，然后在从byte数组中把该数据读出来

\*/

**import** java.io.\*;

**public** **class** TestByteArrayOutputStream2

{

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

{

**long** n = 1234567;

ByteArrayOutputStream baos = **new** ByteArrayOutputStream();

DataOutputStream dos = **new** DataOutputStream(baos);

dos.writeLong(n);

**byte**[] buf = baos.toByteArray();

ByteArrayInputStream bis = **new** ByteArrayInputStream(buf);

DataInputStream dis = **new** DataInputStream(bis);

**long** n2 = dis.readLong();

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

dos.close();

dis.close();

}

}

## 13.6 转换流

OutputStreamWriter流是把OutputStream流转化成Writer流的流

InputStreamReader是把Inputstream转化成Reader

OutputStreamWriter和InputStreamReader都是包裹流

编程：如何将键盘输入的字符组成字符串直接赋值给String对象

代码：

TestStringInput.java

/\*

2013年7月31日10:22:10

如何将键盘输入的字符组成字符串直接赋给String 对象

预备知识:

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

Reader FileReader InputStream FileInputStream BufferedInputStream

流中都没有 readLine 方法

DataInputStream 流中有 readLine方法，但已经 被标记为过时

BufferedReader 流中有readLine方法，并且该方法是可以正确被使用的

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

\*/

**import** java.io.\*;

**public** **class** TestStringInput

{

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

{

String str = **null**;

BufferedReader br = **new** BufferedReader ( //21行

**new** InputStreamReader(System.*in*)

); //23行 查API：从21行到23行的代码是不会抛出任何异常的

**try**

{

str = br.readLine(); //会抛出IOException异常

}

**catch** (IOException e)

{

e.printStackTrace();

System.*exit*(-1);

}

System.*out*.println("str = " + str);

**try**

{

br.close(); //会抛出IOException异常

}

**catch** (IOException e)

{

e.printStackTrace();

System.*exit*(-1);

}

}

}

/\*

在JDK 1.6中的运行结果是：

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

sadd行政村123Asd?asd撒旦

str = sadd行政村123Asd?asd撒旦

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

\*/

### readLine()与回车符的问题

String str = "123";

BufferedReader br = **new** BufferedReader(**new** InputStreamReader(System.in));

str = br.readLine();

如果直接输入回车的话，则

1 br.readLine()会丢弃回车符，而不是返回回车符，即br.readLine()遇到回车符时终止读取，并且会把读取到的回车符自动丢弃掉

2 br.readLine()返回的是””而不是null，””表示空字符串,null表示空指针，空指针就是空地址，空地址就是不知向任何存储单元的意思。

### readLine()与回车符的问题示例

**import** java.io.\*;

**public** **class** A

{

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

{

String str = "zhangsan";

System.*out*.println("str = " + str);

BufferedReader br = **new** BufferedReader(**new** InputStreamReader(System.*in*));

str = br.readLine();

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

System.*out*.println(str + "123");

System.*out*.println(str.equals("")); // true

// System.out.println(str == null)); // false

}

}

## 13.7 Print流

Print流只输出，不输入

分类：

PrintWriter输出字符

PrintStream输出字节

### Print流的由来

Writer和write方法可以写入：

一个字符；一个字符数组；一个字符数组的一部分；一个字符串；一个字符的一部分。

OutputStream的write方法可以写入：

一个字节；一个字节数组；一个字节数组的一部分。

DataOutputStream流可以写入：

一个字节；一个字节数组（继承自OutputStream）；一个字节数组的一部分；所有的基本类型数据的二进制代码（如：writeDouble(8, 8);写入的8.8的二进制代码，共占8个字节）

PrintStream流的print可以写入：

所有基本类型数据的字符串形式表示（如：print(8.8);写入的是’8’’.’’8’这三个字符，共占3个字节）

### PrintStream

PrintStream在OutputStream基础上提供了增强的功能，即可以方便的输出各种类型的数据（而不仅限于byte类型）的格式化表示形式。

PrintStream重载了print和println方法，用于各种不同类型的数据的格式化输出。

格式化输出是指将一个数据勇气字符串格式输出。

DataOutputStream中的writeXXX(data)方法是把data在内存中的二进制数据写入文件。

PrintStream中的println(data)写出的是该数据格式化后的字符串。

代码：

TestPrintStream\_1.java

/\*

DataOutputStream 中的 writeXXX(data)方法

与

PrintStream 中的 println(data)的区别

总结:

DataOutputStream 中的 writeXXX(data)方法是把data在内存中的二进制数据写入文件

PrintStream 中的 println(data)写出的是该数据的格式化后的字符串

\*/

**import** java.io.\*;

**public** **class** TestPrintStream\_1

{

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

{

DataOutputStream dos = **new** DataOutputStream(**new** FileOutputStream("d:/share/kk.txt"));

dos.writeLong(12345); //实际写入文件的是00 00 00 00 00 00 30 39

dos.close();

System.*out*.printf("%#X\n", 12345);

PrintStream ps = **new** PrintStream(**new** FileOutputStream("d:/share/kk2.txt"), **true**);

ps.println(12345); //实际写入文件的是'1' '2' '3' '4' '5'

ps.close();

}

}

### PrintWriter

PrintWriter提供了PrintStream的所有打印方法，其方法也从不抛出IOException。

与PrintStream的区别：

作为处理流使用时，PrintStream只能封装OutputStream类型的字节流，而PrintWriter既可以封装OutputStream，还能够封装Writer。PrintWriter中的println()方法具有更好的跨平台性。

### 标准输入输出的重定向

例子：编程实现将键盘输入的数据输入A文件中，如果输入有误，则把出错信息输出到B文件中

代码：

TestSetSystemOut.java

**import** java.io.\*;

**public** **class** TestSetSystemOut

{

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

{

PrintStream ps\_out = **null**;

**try**

{

ps\_out = **new** PrintStream(**new** FileOutputStream("d:/share/ww.txt"));

System.*setOut*(ps\_out); //将System.out的值重新设置为ps\_out,即System.out不在关联到显示器，而是关联到"d:/share/ww.txt"文件

System.*out*.println(12); //这实际上是把12输出到了System.out所关联的d:/share/ww.txt中

System.*out*.println(55.5); //同上

}

**catch** (Exception e)

{

e.printStackTrace();

}

**finally**

{

**try**

{

ps\_out.close();

}

**catch** (Exception e)

{

e.printStackTrace();

}

}

}

}

代码：

TestSetOutErr.java

**import** java.io.\*;

**public** **class** TestSetSystemOut

{

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

{

PrintStream ps\_out = **null**;

**try**

{

ps\_out = **new** PrintStream(**new** FileOutputStream("d:/share/ww.txt"));

System.*setOut*(ps\_out); //将System.out的值重新设置为ps\_out,即System.out不在关联到显示器，而是关联到"d:/share/ww.txt"文件

System.*out*.println(12); //这实际上是把12输出到了System.out所关联的d:/share/ww.txt中

System.*out*.println(55.5); //同上

}

**catch** (Exception e)

{

e.printStackTrace();

}

**finally**

{

**try**

{

ps\_out.close();

}

**catch** (Exception e)

{

e.printStackTrace();

}

}

}

}

## 13.8 对象的序列化

所谓序列化是指：把一个Object对象直接转化为字节流，然后把这个字节流直接写入本地硬盘或网络中

如果要想把某个对象序列化，则必须得实现Serializable接口。

Serializable接口中并没有任何方法，该接口被称为标记接口，如果一个类实现了Serializable接口，潜在含义就是告诉编译器这个类是允许被序列化的，如果程序中存在序列该对象的代码，编译器就会自动进行相应的处理以完成对象的序列化，如果该对象没有实现Serializable接口，程序中却存在该对象被序列化的代码，编译器编译时就会报错！！

在Java中transient修饰的成员变量在对象序列化时不被序列化。

代码：

TestObjectIO.java

**import** java.io.\*;

**public** **class** TestObjectIO

{

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

{

ObjectOutputStream oos = **null**;

ObjectInputStream ois = **null**;

Student ss = **new** Student("zhansan", 1000, 88.8f); //注意88.8f不能改为88.8

Student ss2 = **null**;

**try**

{

FileOutputStream fos = **new** FileOutputStream("d:/share/java/ObjectOut.txt");

oos = **new** ObjectOutputStream(fos);

oos.writeObject(ss);

ois = **new** ObjectInputStream(**new** FileInputStream("d:/share/java/ObjectOut.txt"));

ss2 = (Student)ois.readObject(); //(Student)不能省 ois.readObject();如果ois中的某个成员是transient,则该成员是不会被读取的，因为该成员不会被保存，何来读取之说？！

System.*out*.println("ss2.sname = " + ss2.sname);

System.*out*.println("ss2.sid = " + ss2.sid);

System.*out*.println("ss2.sscore = " + ss2.sscore);

}

**catch** (FileNotFoundException e)

{

System.*out*.println("文件没有找到!");

System.*exit*(-1);

}

**catch** (Exception e)

{

e.printStackTrace();

System.*exit*(-1);

}

**finally**

{

**try**

{

oos.close();

ois.close();

}

**catch** (Exception e)

{

e.printStackTrace();

System.*exit*(-1);

}

}

}

}

**class** Student **implements** Serializable //如果将implements Serializable 注释掉，则程序编译时就会报错

{

**public** String sname = **null**;

**public** **int** sid = 0;

**transient** **public** **float** sscore = 0; //表示sscore成员不能被序列化，所谓不能被序列化就是指：“该成员调用ObjectOutputStream 的writeOnbject()时不会被保存，调用ObjectInputStream的readObject()方法时不会被读取”

**public** Student(String name, **int** id, **float** score)

{

**this**.sname = name;

**this**.sid = id;

**this**.sscore = score;

}

}

## 附：IO流一览表

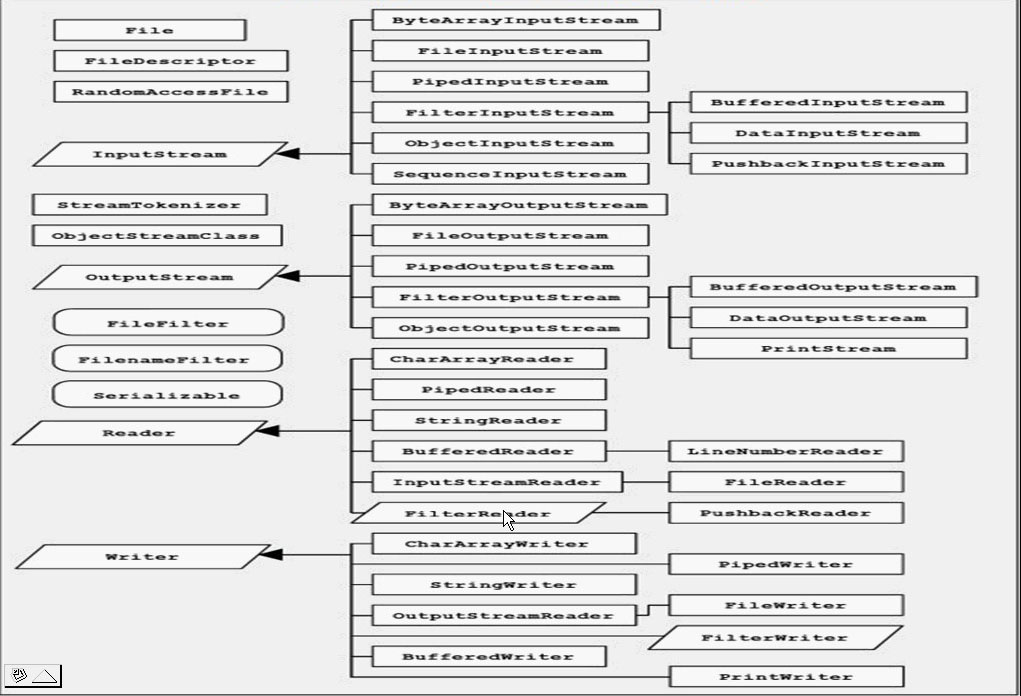


图13-附-1

# 14 容器

## 14.1 为什么需要容器

数组存在两个缺陷：

数组长度难以扩充

数组中元素类型必须相同

容器可以弥补数组的这两个缺陷

举例：

假设A是个类名，A[] arr = new A[10]; 表示分配了一个数组，数组的每个元素都是A类对象的一个引用，但是如果想扩充数组的长度，比如希望数组的长度变成15，我们是不能直接在原数组内存的后面追加内存的，必须得另外分配长度为15的内存空间，然后利用System.arrayCopy()方法来把原数组的内容拷贝到新内存中，很明显，这既耗时又耗内存！

所以一旦数组内存已分配，你想改变数组的长度，效率就会变得很低。

### 容器与现实的对应关系

集合就是将若干用途、性质相同或相近的“数据”组合而成一个整体。

数学上，集合类型可以归纳为三种：

集（Set）：Set集合中不区分元素的顺序，不允许出现重复元素。

列表（List）：List集合区分元素的顺序，且允许包含重复元素。

映射（Map）：映射中保存成对的“键-值”（Key-Value）信息，映射中不能包含重复的键，每个键最多只能映射一个值。

Java设计了三个接口来对应数学上的三种集合类型，这三个接口名字分别是Set List Map

### 容器的框架图

容器API

J2SDK所提供的容器API位于java.util包内。容器API的类图结构如下图所示：

<<interface>>

Collection

<<interface>>

Set

<<interface>>

Map

<<interface>>

List

HashSet

LinkedList

ArrayList

HashMap

图14-1-1

Collection接口已定义了存取一组对象的方法，其子接口Set和List分别定义了存储方式。

Set中的数据对象没有顺序且不可以重复。

List中的数据对象有顺序且可以重复。

Map接口定义了存储“键（Key）-值（Value）”映射对的方法。

## 14.2 容器的分类和使用

### Collection接口

#### Collection接口中的方法介绍

// 返回此collection中的元素数

**int** size();

// 判断是否为空

**boolean** isEmpty();

// 判断形参c所指向的集合中所有的元素是不是已经全部包含在了当前集合中，是，返回true，否则返回false

**boolean** containesAll(Collection c);

// 返回能够遍历当前集合所有元素的迭代器

Iterator iterator();

// 容器不是数组，不能通过下标的方式访问容器中的元素，只有数组才可以通过下标来访问，访问一个包含此collection中所有元素的数组

Object[] toArray();

代码：

TestArrayListToArray.java

/\*

ArrayList对象中的toArray()

size()

的使用

\*/

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

**public** **class** TestArrayListToArray

{

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

{

ArrayList a1 = **new** ArrayList();

a1.add(**new** Point(1,1));

a1.add(**new** Point(2,2));

a1.add(**new** Point(3,3));

System.*out*.println(a1);

System.*out*.println("a1.get(2) = " + a1.get(2)); //a1.get(2)返回第三个元素，注意不能写成a1[2],因为a1是ArrayList类型的对象， 而不是普通数组

Object [] obs = a1.toArray(); //toArray()返回的是Object类型的数组

**for** (**int** i=0; i<obs.length; ++i) // obs.length和a1.size()是一样的效果，但要注意不能混合写成obs.size() 或 a1.length() 这都是错的!

{

System.*out*.println(obs[i]);

}

}

}

**class** Point

{

**int** x;

**int** y;

Point(**int** i, **int** j)

{

x = i;

y = j;

}

**public** String toString()

{

**return** "x=" + x + ",y=" + y;

}

}

/\*

在j2sdk1.4.2\_16中的运行结果是：

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

[x=1,y=1, x=2,y=2, x=3,y=3]

a1.get(2) = x=3,y=3

x=1,y=1

x=2,y=2

x=3,y=3

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

总结：

ArrayList可变数组不能使用 [], 必须使用get(int index);

普通数组可以使用[], 并且还可以使用字段length,但不能使用get(int index);

\*/

// 添加e到当前集合中

**boolean** add(Object e);

// 删除元素o

**boolean** remove(Object o);

// 把c中所有的元素添加到当前集合中

**boolean** addAll(Collection c);

// 把c中所有元素删除

**boolean** removeAll(Collection c);

// 把当前容器中的所有元素清除

**void** clear();

**boolean** equals(Object o);

**int** hashCode();

#### Collection接口的子接口——List接口

List接口是Collection的字接口，实现List接口的容器类中的元素是有顺序的，而且可以重复。

List容器中的元素都应对一个整数型的序号记载其在容器中的位置，可以根据序号存取容器中的元素。

J2SDK所提供的List容器类有ArrayList，LinkedList等。

Object get(**int** index);

Object set(**int** index, Object element);

**void** add(**int** index, Object element);

Object remove(**int** index);

**int** indexOf(Object o);

**int** lastIndexOf(Object o);

#### Collection接口的子接口——Set接口

因为Set和List都是继承自Collection接口，所以Set和List中有很多方法是一样的

List接口中有add，set，indexOf方法，但是Set接口中却只有add方法，没有set，indexOf方法，因为Set是无序不能重复的，不存在某元素具体位置这个概念。

在后面的[Set接口](#_14.5_Set接口)还会详细介绍。

### Map接口

java.util.Map接口描述了映射结构，Map结构允许以键集、值集合或键-值映射关系集的形式查看某个映射的内容。

主要方法：

Object put(Object key, Object value);

Object get(Object key); // 注意Map接口中并没有Object get(Object value)

**boolean** isEmpty();

**void** clear();

**int** size();

**boolean** containsKey(Object key);

**boolean** containsValue(Object value);

### 预备知识：toString方法

对任意类A，假设aa是该A类的一个实例System.*out*.println(aa); // 等价于System.out.println(aa.toString())等价于System.*out*.println(aa.toString());

Object类中含有toString方法

Java中所有的类默认都继承了Object的toString方法

建议所有的子类都重写toString方法。

代码：

TestToString.java

**public** **class** TestToString

{

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

{

Point p1 = **new** Point(1, 1);

System.*out*.println(p1);

}

}

**class** Point

{

**private** **int** i, j;

**public** Point()

{

}

**public** Point(**int** i, **int** j)

{

**this**.i = i;

**this**.j = j;

}

**public** String toString() //24行

{

**return** i + ", " + j;

} //27行

}

/\*

在JDK 1.6中的运行结果是：

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

1 , 1

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

如果把24行到27行全部注释到，则输出结果就是乱码

所以“建议所有的类都应该重写toString方法”

\*/

### Collection接口实现类示例

代码:

TestCollection\_2.java

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

**public** **class** TestCollection\_2

{

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

{

Collection c = **new** ArrayList();

c.add("zhangsan");

c.add(123);

c.add(66.6);

System.*out*.println(c);

}

}

执行结果：

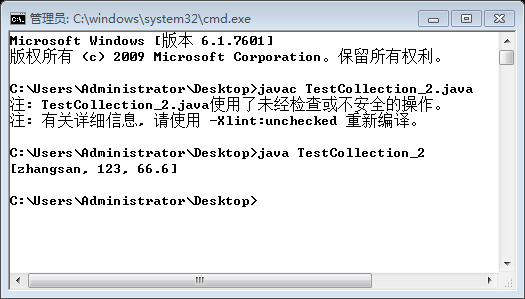


图14-2-1

总结：

Collection c = new ArrayList();

……

……

System.out.println(c);

本语句等价于：

[调用第一个元素的toString方法，调用第二个元素的toString方法，调用第三个元素的toString方法……]

因此强烈建议：所有添加到Collection容器中的对象都应该重写父类Object的toString方法

代码：

TestCollection.java

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

**public** **class** TestCollection {

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

Collection c = **new** LinkedList();

c.add(**new** Student("zhangsan", 80));

c.add(66.6);

System.*out*.println(c);

}

}

**class** Student {

**private** String name;

**private** **int** age;

**public** Student(String name, **int** age) {

**this**.name = name;

**this**.age = age;

}

// 如果把toString方法注释掉了，则程序输出结果会有乱码

**public** String toString() {

**return** name + "" + age;

}

}

执行结果：

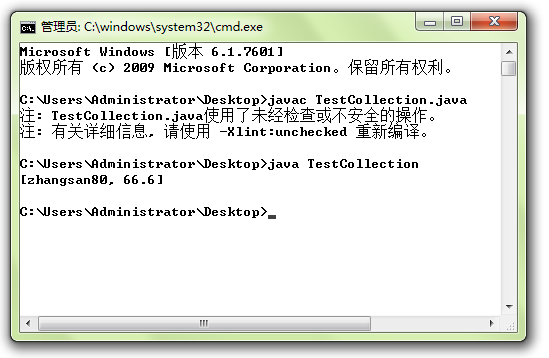


图14-2-2

强烈建议：所有添加到Collection容器中的对象都应该重写父类Object的toString方法

### ArrayList与LinkedList的比较

ArrayList和LinkedList都实现了List接口中的方法，但两者内部实现不同

ArrayList底层采用数组完成，而LinkedList则是以一般的双向链表（double-linked list）完成，其内每个对象除了数据本身外，还有两个引用，分别指向前一个元素和后一个元素。

如果我们经常在List的开始处增加元素，或者在List中进行插入和删除操作，我们应该使用LinkedList，否则的话，使用ArrayList将更加快速。

ArrayList存取速度快，插入删除慢；

LinkedList存取速度慢，插入删除速度快。

## 14.3 Collections类

Collection接口的实现类，如ArrayList、LinkedList本身并没有提供排序，倒置，查找等方法，这些方法是由Collections类来实现的，该类有很多public static 方法，可以直接对Collection接口的实现类进行操作。

### Collections类常用算法

类java.util.Collections提供了一些静态方法实现了基于List容器的一些常用算法。

**void** sort(List) // 对List容器内的元素排序

**void** shuffle(List) // 对List容器内的对象进行随机排列

**void** reverse(List) // 对List容器内的对象进行逆序排列

**void** fill(List, Object) // 用一个特定的对象重写整个List容器

**void** copy(List dest, List src) // 将src List容器内拷贝到dest List容器

**int** binarySearch(List, Object) // 对于顺序的List容器，采用折半查找的方法查找特定对象。

### Collections类举例

代码：

TestCollections.java

/\*

2013年8月1日11:36:32

测试Collections类的使用

\*/

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

**public** **class** TestCollections

{

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

{

List lt = **new** LinkedList();

**for** (**int** i=0; i<7; ++i)

{

lt.add("a" + i);

}

System.*out*.println(lt);

Collections.*shuffle*(lt); //记住LinkedList中是没有shuffle方法的，因此需要通过Collections类的相关方法来实现

System.*out*.println(lt);

Collections.*sort*(lt); //默认升序排序，要降序很简单，先调用Collections.sort(); 再调用Collections.reverse()

System.*out*.println(lt);

Collections.*reverse*(lt); //倒置

System.*out*.println("倒置之后: " + lt);

System.*out*.println(Collections.*binarySearch*(lt, "a5")); //因为lt默认不是升序排序的，所以调用Collections.binarySearch()方法是不会成功的

Collections.*sort*(lt);

System.*out*.println("重新排序之后: " + lt);

System.*out*.println(Collections.*binarySearch*(lt, "a5")); //记住，使用binarySearch()方法的前提是该容器已升序排序

}

}

/\*

在JDK 1.6中的运行结果是：

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

[a0, a1, a2, a3, a4, a5, a6]

[a5, a3, a6, a4, a2, a0, a1]

[a0, a1, a2, a3, a4, a5, a6]

倒置之后: [a6, a5, a4, a3, a2, a1, a0]

-8

重新排序之后: [a0, a1, a2, a3, a4, a5, a6]

5

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

\*/

### Collections.fill()用法

代码：

Test

**import** java.util.Collections;

**import** java.util.LinkedList;

**import** java.util.List;

**public** **class** Test

{

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

List lt = **new** LinkedList();

lt.add("111");

lt.add("222");

lt.add("333");

System.*out*.println(lt);

Collections.*fill*(lt, "888");

System.*out*.println(lt);

}

}

执行结果：



图14-3-1

## 14.4 Comparale接口

### 为什么要使用Comparable接口

基本类型数据和String类型数据，它们彼此的比较标准Java语言本身已经提供好了。用户自定义类对象之间比较的标准Java语言本身是没有提供的，所以如果一个容器中含有用户自定义类型的数据，并且我们需要对容器中元素进行排序，或查找某一元素时，我们就必须得知道制定容器中元素与元素之间比较的标准。

凡是需要进行对象比较/排序的场合均可考虑实现Comparable接口。

代码：

TestComparableQuestion.java

/\*

为何要使用Comparable接口示例

\*/

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

// 如果把implements Comparable注释掉了，则程序编译时没有错误，但运行时会报错！！

**class** Person **implements** Comparable {

**private** **int** id;

**private** String name;

**public** Person(**int** id, String name) {

**this**.id = id;

**this**.name = name;

}

**public** String toString() {

**return** "id:" + id + ", name:" + name;

}

@Override

**public** **int** compareTo(Object o) {

// **TODO** Auto-generated method stub

Person p = (Person) o;

**return** **this**.id - p.id;

}

}

**public** **class** TestComparableQuestion {

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

ArrayList c = **new** ArrayList();

c.add(**new** Person(1000, "zhangsan"));

c.add(**new** Person(1002, "wangwu"));

java.util.Collections.*sort*(c);

}

}

### Comparable接口介绍

问题：上面的算法根据什么确定容器中对象的“大小”顺序？

所有可以“排序”的类都实现了java.lang.Comparable接口，Comparable接口中只有一个方法public int comparator(Object obj);该方法：返回0表示this == obj，返回证书表示 this > obj 返回负数表示this < obj

实现了Comparable接口的类通过实现comparaTo方法从而确定该类对象的排序方式。

编写一个程序：把所有人的信息存入电脑，在按编号排序输出。

代码：

TestComparable.java

/\*

所有可以排序的类都应该实现Comparable接口

\*/

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

**import** java.util.Collections;

**class** Person **implements** Comparable

{

**private** **int** id;

**private** String name;

**public** Person()

{

}

**public** Person(**int** id, String name)

{

**this**.id = id;

**this**.name = name;

}

**public** String toString()

{

**return** "id: " + id + ", name: " + name ;

}

@Override

**public** **int** compareTo(Object o)

{

Person p = (Person)o;

**return** **this**.id - p.id;

}

}

**public** **class** TestComparable

{

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

{

ArrayList c = **new** ArrayList(); //不能写成：Collection c = new ArrayList(); 这会导致49行出错，因为Collection.sort()接受的形参必须的是List接口类型的变量,Collection.sort()不能接受Collection类型的变量，这语法上编译时是通不过的

c.add(**new** Person(1000, "zhangsan"));

c.add(**new** Person(1002, "wangwu"));

c.add(**new** Person(1001, "lisi"));

c.add(**new** Person(1008, "xiaojuan"));

c.add(**new** Person(1005, "asd"));

**for** (**int** i=0; i<c.size(); ++i)

{

System.*out*.println(c.get(i));

}

java.util.Collections.*sort*(c); //49行

System.*out*.println("排序之后的内容是:");

Iterator it = c.iterator();

**while** (it.hasNext())

{

System.*out*.println(it.next());

}

}

}

/\*

在JDK 1.6中的运行结果是：

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

id: 1000, name: zhangsan

id: 1002, name: wangwu

id: 1001, name: lisi

id: 1008, name: xiaojuan

id: 1005, name: asd

排序之后的内容是:

id: 1000, name: zhangsan

id: 1001, name: lisi

id: 1002, name: wangwu

id: 1005, name: asd

id: 1008, name: xiaojuan

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

\*/

## 14.5 Set接口

Set接口是Collection的子接口，Set接口没有提供额外的方法，但实现Set接口的容器类中的元素是没有顺序的，而且不可以重复。

Set容器可以与数学中“集合”的概念相对应。

J2SDK API中所提供的Set容器类有HashSet，TreeSet等。

### HashSet类

HashSet类实现了Set接口，HashSet容器中的元素是不能重复，无顺序的；存放入HashSet容器中的类必须要实现equals()和hashCode方法。

具体原因下面会有介绍。

### 14.6 equals()和hashCode()方法

#### 什么类必须得重写equals()和hashCode()方法

HashTable HashSet HashMap都必须得同时实现equals()方法和hashCode()方法，TreeSet和TreeMap则不需要实现equals()方法和hashCode()方法。

添加到HashMap和Hashtable容器中的键必须得同时实现equals()方法和hashCode()方法，否则很可能导致容器中出现重复的映射，所谓重复映射是指同一个键值映射在容器中出现了多次，参见“TestHash.java”代码

添加到HashSet容器中的对象也必须得同时实现equals()方法和hashCode()方法，否则很可能导致容器中出现重复的对象，参见“TestHashSet.java”代码。

代码：

TestHash.java

**import** java.util.Set;

**import** java.util.TreeMap;

**import** java.util.Iterator;

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

**public** **class** TestHash

{

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

Map hm = **new** Hashtable(); //针对HashMap而言：只要没有重写键类MyKey中的hashCode 和 equasl方法中的任何一个方法，都会导致HashMap中出现重复映射，即会出现多个"1001 Nancy 49"

//针对Hashtable而言：只要没有重写键类MyKey中的hashCode 和 equasl方法中的任何一个方法，都会导致HashMap中出现重复映射，即会出现多个"1001 Nancy 49"

//但是针对TreeMap而言，至少在本程序中, 是否重写键类MyKey中的hashCode 和 equasl方法,是重写一个还是重写两个还是两个都不重写，整个程序的输出结果没有任何影响，我个人估计是因为TreeMap容器已经要求所有的键类都要实现Comparable接口,这时候可能hashCode方法和equasl方法都用不到了，当然这只是我个人猜测而已!

hm.put(**new** MyKey(1003),**new** Person("Tom",15));

hm.put(**new** MyKey(1008),**new** Person("Billy",25));

hm.put(**new** MyKey(1005),**new** Person("Kity",73));

hm.put(**new** MyKey(1001),**new** Person("Nancy",49));

hm.put(**new** MyKey(1001),**new** Person("Nancy",49));

hm.put(**new** MyKey(1001),**new** Person("Nancy",49));

hm.put(**new** MyKey(1001),**new** Person("Nancy",49));

System.*out*.println("----检索单个元素----");

Person p = (Person)hm.get(**new** MyKey(1008));

System.*out*.println(p);

System.*out*.println("----遍历所有元素----");

Set names = hm.keySet();

Iterator it = names.iterator();

**while**(it.hasNext()){

MyKey key = (MyKey)it.next();

Person value = (Person)hm.get(key);

System.*out*.println(key.getId() + "\t" + value);

}

System.*out*.println("hm.size() = " + hm.size());

}

}

/\*

在JDK 1.6中的运行结果是：

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

----检索单个元素----

Name: Billy Age: 25

----遍历所有元素----

1008 Name: Billy Age: 25

1005 Name: Kity Age: 73

1003 Name: Tom Age: 15

1001 Name: Nancy Age: 49

hm.size() = 4

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

\*/

代码：

TestHashSet.java

/\*

2013年8月1日16:54:11

Student类必须同时实现equals方法 和 hashCode方法 才可以保证在装入HashSet类时不会出现“重复”装入的情况

重新实现了equals方法 和 hashCode 方法 的正确的程序

\*/

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

**public** **class** TestHashSet

{

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

{

Collection c = **new** HashSet();

c.add(**new** Student(1001, "张三"));

c.add(**new** Student(1002, "李四"));

c.add(**new** Student(1003, "王五")); //10行

c.add(**new** Student(1003, "王五"));

c.add(**new** Student(1003, "王五"));

c.add(**new** Student(1003, "王五"));

c.add(**new** Student(1003, "王五")); //14行

Iterator i = c.iterator();

**while** (i.hasNext())

{

System.*out*.println(i.next());

}

}

}

**class** Student

{

**private** **int** num;

**private** String name;

**public** Student()

{

}

**public** Student(**int** num, String name)

{

**this**.num = num;

**this**.name = name;

}

**public** String toString()

{

**return** "学号: " + **this**.num + ", 姓名: " + name;

}

**public** **boolean** equals(Object o)

{

Student s = (Student)o;

**return** **this**.num==s.num && **this**.name.equals(s.name);

}

**public** **int** hashCode()

{

//return num; //如果你设定的学生信息中学号是唯一的，则可以直接用num来作为哈希码

**return** num \* **this**.name.hashCode();

}

}

/\*

在JDK 1.6中的运行结果是：

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

学号: 1002, 姓名: 李四

学号: 1003, 姓名: 王五

学号: 1001, 姓名: 张三

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

总结:

必须同时实现equals()方法 和 hashCode() 方法,只要有任意一个方法没有实现

装入时就会出现重复元素

\*/

添加到TreeSet中的对象和添加到TreeMap中的键都可以不重写equals()方法和hashCode()方法，更准确的讲：添加到TreeSet中的对象和添加到TreeMap中的键是否重写equals()方法和hashCode()方法，以及怎样重写equals()方法和hashCode()方法，对程序没有任何影响，可以参见 [TreeSet类](#_TreeSet类) 和

#### 为什么要重写equals()和hashCode()方法

预备知识：

散列码：Object中的hashCode方法会返回该对象的内存真是地址的证书化表示，这个形象的不是真正地址的整数值就是哈希码。

向HashSet中添加对象时，HashSet先通过该对象的hashCode()计算出相应的桶，然后再根据equals()方法找到相应的对象。如果容器中已经存在该对象则不再添加，如果不存在，则添加进去！！！

代码：

TestHashCode\_1.java

/\*

2013年8月1日16:41:26

String 和 Integer 这些Java自带的类都重写了hashCode方法，如果String 和 Integer new出来的

对象的内容是一样的，则这些对象的hashCode返回值也是一样的，虽然这些对象占用的是不同的内存

不过用户自定义类型则不同，如本程序的A类，即便是两个内容一模一样的A类对象，它们返回的

hashCode值也是不一样的，但是两个内容一模一样的Integer类对象或者String类对象返回的

hashCode值却是一样的，因为系统自带的String 和 Integer 类都已经重写了Object的hashCode方法嘛！

如果程序员希望自己定义的类对象，占用不同内存空间但内容却是一样的对象调用hashCode方法

返回值是一样的，则程序员就必须自己重写hashCode方法,如本程序的B类

\*/

**class** A

{

**private** **int** id;

**public** A(**int** id)

{

**this**.id = id;

}

}

**class** B

{

**private** **int** id;

**public** B(**int** id)

{

**this**.id = id;

}

@Override

**public** **int** hashCode()

{

**return** **new** Integer(id).hashCode();

}

}

**public** **class** TestHashCode\_1{

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

System.*out*.println("new A(1).hashCode() = " + **new** A(1).hashCode());

System.*out*.println("new A(1).hashCode() = " + **new** A(1).hashCode());

System.*out*.println();

System.*out*.println("new Integer(1).hashCode() = " + **new** Integer(1).hashCode());

System.*out*.println("new Integer(1).hashCode() = " + **new** Integer(1).hashCode());

System.*out*.println();

System.*out*.println("new String(\"haha\").hashCode() = " + **new** String("haha").hashCode());

System.*out*.println("new String(\"haha\").hashCode() = " + **new** String("haha").hashCode());

System.*out*.println();

System.*out*.println("\"haha\".hashCode() = " + "haha".hashCode());

System.*out*.println("\"haha\".hashCode() = " + "haha".hashCode());

System.*out*.println();

System.*out*.println("new B(1).hashCode() = " + **new** B(1).hashCode());

System.*out*.println("new B(1).hashCode() = " + **new** B(1).hashCode());

Integer it1 = **new** Integer(1); Integer it2 = **new** Integer(1);

System.*out*.println(it1 == it2);

System.*out*.println(it1.equals(it2));

System.*out*.println(it1.hashCode() == it2.hashCode());

}

}

/\*

在JDK 1.6中的运行结果是：

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

new A(1).hashCode() = 14576877

new A(1).hashCode() = 12677476

new Integer(1).hashCode() = 1

new Integer(1).hashCode() = 1

new String("haha").hashCode() = 3194802

new String("haha").hashCode() = 3194802

"haha".hashCode() = 3194802

"haha".hashCode() = 3194802

new B(1).hashCode() = 1

new B(1).hashCode() = 1

false

true

true

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

\*/

#### 怎样重写equals()和hashCode()方法

如何重写equals()

public Boolean equals(Object obj)

{

如果this和Obj的内容是一模一样的

返回true

否则

返回false

}

如何重写hashCode()方法

public int hashCode()

{

return 当前类中基本类型数据对象的hashCode()方法

}

#### 举例

把学生信息存入HashSet容器中，要求不能出现重复的学生信息，最后把学生信息全部输出。附注：HashSet类中并没有public Object get(int index)方法，因此要想输出HashSet容器所有的元素，可以借助iterator()方法来实现。

代码：

TestHashSet.java

/\*

Student类必须同时实现equals方法 和 hashCode方法 才可以保证在装入HashSet类时不会出现“重复”装入的情况

重新实现了equals方法 和 hashCode 方法 的正确的程序

\*/

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

**public** **class** TestHashSet3

{

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

{

Collection c = **new** HashSet();

c.add(**new** Student(1001, "张三"));

c.add(**new** Student(1002, "李四"));

c.add(**new** Student(1003, "王五")); //10行

c.add(**new** Student(1003, "王五"));

c.add(**new** Student(1003, "王五"));

c.add(**new** Student(1003, "王五"));

c.add(**new** Student(1003, "王五")); //14行

Iterator i = c.iterator();

**while** (i.hasNext())

{

System.out.println(i.next());

}

}

}

**class** Student

{

**private** **int** num;

**private** String name;

**public** Student()

{

}

**public** Student(**int** num, String name)

{

**this**.num = num;

**this**.name = name;

}

**public** String toString()

{

**return** "学号: " + **this**.num + ", 姓名: " + name;

}

**public** **boolean** equals(Object o)

{

Student s = (Student)o;

**return** **this**.num==s.num && **this**.name.equals(s.name);

}

**public** **int** hashCode()

{

//return num; //如果你设定的学生信息中学号是唯一的，则可以直接用num来作为哈希码

**return** num \* **this**.name.hashCode();

}

}

/\*

在JDK 1.6中的运行结果是：

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

学号: 1002, 姓名: 李四

学号: 1003, 姓名: 王五

学号: 1001, 姓名: 张三

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

总结:

必须同时实现equals()方法 和 hashCode() 方法,只要有任意一个方法没有实现

装入时就会出现重复元素

\*/

## 14.7 Iterator接口

Iterator接口用来以统一的方式对集合中的各个元素进行遍历，Iterator接口的对象有称为迭代器，利用该对象可以方便的遍历容器中的元素。所有实现了Collection接口的容器类都有一个iterator()方法，该方法返回一个实现了Iterator接口的对象。

### Iterator方法介绍

示意图

Next()

游标

元素

图14-7-1

**boolean** hasNext(); // 是用来判断当前游标的后面还是否存在元素，如果存在返回true，否则返回false

Object next(); // 先返回当前游标右边的元素，然后游标后移一个位置

**void** remove(); // 删除最近返回的元素，在调用remove之前，我们至少保证先调用一次next方法，而且调用next之后只能调用一次remove方法。remove方法不推荐使用

代码：

TestIterator\_1.java

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

**public** **class** TestIterator\_1

{

//可以遍历所有Collection接口的实现类

**public** **static** **void** showCollection(Collection c)

{

Iterator it = c.iterator();

**while** (it.hasNext())

{

System.*out*.println(it.next());

}

}

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

{

ArrayList al = **new** ArrayList();

al.add("one");

al.add(22);

al.add(**new** Point(1,1));

System.*out*.println("a1 容器的内容是:");

*showCollection*(al);

HashSet hs = **new** HashSet();

hs.add("one");

hs.add(22);

hs.add(**new** Point(1,1));

System.*out*.println("hs容器的内容是:");

*showCollection*(hs);

}

}

**class** Point

{

**private** **int** i, j;

**public** Point(**int** i, **int** j)

{

**this**.i = i;

**this**.j = j;

}

**public** String toString()

{

**return** "[" + i + ", " + j + "]";

}

}

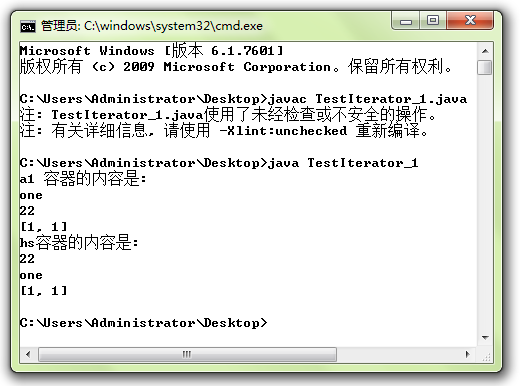


图14-7-2

### TreeSet类

TreeSet类实现了Set接口，TreeSet是一个有序集合，TreeSet中元素将按照升序排列，缺省是按照自然顺序进行排列，因此TreeSet中元素要实现Comparable接口。

记住：所有可以进行排序的类都应该实现Comparable接口。

### 示例

代码：

Test.java

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

**public** **class** Test {

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

Collection c = **new** TreeSet();

c.add("123");

c.add("456");

c.add("234");

c.add("111");

c.add("678");

Iterator i = c.iterator();

**while** (i.hasNext()) {

System.*out*.println(i.next());

}

}

}

执行结果：

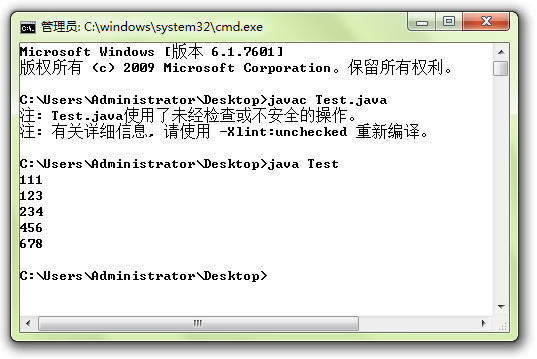


图14-7-3

代码：

TestTreeSet.java

/\*

2013年8月1日16:06:17

总结:

向TreeSet容器中添加的对象必须得是实现Comparable接口的类对象 ，否则运行程序会报错

向TreeSet容器中添加的对象可以不重写hashCode方法和equasl方法.

\*/

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

**public** **class** TestTreeSet

{

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

{

Collection c = **new** TreeSet();

c.add(**new** Student(1006, "张三"));

c.add(**new** Student(1003, "小娟")); //16行

c.add(**new** Student(1002, "李四"));

c.add(**new** Student(1008, "王刚")); //16行

c.add(**new** Student(1001, "王五")); //16行

Iterator i = c.iterator();

**while** (i.hasNext())

{

System.*out*.println(i.next());

}

System.*out*.println("哈哈嘿嘿!");

}

}

**class** Student **implements** java.lang.Comparable

{

**private** **final** **int** num;

**private** String name;

**public** Student(**int** num, String name)

{

**this**.num = num;

**this**.name = name;

}

**public** String toString()

{

**return** "学号: " + **this**.num + ", 姓名: " + name;

}

@Override

**public** **boolean** equals(Object o)

{

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

**if** (o **instanceof** Student)

{

**if** (**this**.num == ((Student)o).num)

flag = **true**;

}

**return** flag;

}

@Override

**public** **int** compareTo(Object o)

{

//Student s = (Student)o;

**return** **this**.num - ((Student)o).num;

}

}

/\*

在JDK 1.6中的运行结果是：

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

学号: 1001, 姓名: 王五

学号: 1002, 姓名: 李四

学号: 1003, 姓名: 小娟

学号: 1006, 姓名: 张三

学号: 1008, 姓名: 王刚

哈哈嘿嘿!

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

\*/

### HashSet和TreeSet的比较

HashSet是基于Hash算法实现的，其性能通常优于TreeSet。我们通常都应该使用HashSet，在我们需要排序的功能时，我们才使用TreeSet。

## 14.8 Map

### 预备知识：哈希表

#### 哈希表的定义

哈希表不是指存储需要保存的数据，而是既保存数据，也保存该数据的主键，实际是：先保存主键，然后哈希表会根据某种算法自动计算出以当前主键为主键的数据的存储位置，然后再把该数据保存进去

#### 哈希表

假设待保存的数据是val，val的主键是key。则哈希表是先存储key，然后哈希表会自动根据key计算出val的存储位置，并最终把val存储进去。

#### 哈希表注意事项

Hash即哈希表又称散列表

哈希表主要是为了提高数据的存储速度和查找速度而设计的

哈希表是人类的一种追求，人很难设计出完美的哈希表

几乎所有的哈希表都会产生哈希冲突

Java中是利用桶来解决哈希冲突的

### Map接口

Map（映射）是一种把键对象和值对象进行映射的集合，它的每一个元素都包含一对键和值对象。

向Map集合中加入元素时，必须提供一对键对象，从Map集合中检索元素时，只要给出键对象，就会返回对应的值对象。

接口Map提供了根据关键字找对应的值的方法get()，定义如下：Object get(Object key)

Map接口方法介绍

接口Map还提供了设置关键字和对应值的方法put(),定义如下：

Object put(Object key, Object value) // 根据key计算出存储位置，然后把value存入此位置，如果该位置原来已有元素，则覆盖它。该方法返回覆盖之前的元素对象的值，如果以前此位置为空，则返回null

**void** putAll(Map m) // 把m容器中所有的元素存入当前容器中。

**int** size()

**boolean** isEmpty()

**boolean** containsKey(Object key)

**boolean** containsValue(Object value)

Object remove(Object key)

**void** clear()

### Map示例

代码：

TestHashMap\_1.java

/\*

2013年8月1日16:25:09

有关HashMap类的实例

\*/

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

**public** **class** TestHashMap\_1

{

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

{

Map m1 = **new** HashMap();

m1.put("one", 1);

m1.put("two", 2);

m1.put("three", 3);

System.*out*.println("1-> " + m1);

System.*out*.println("2-> " + m1.size());

m1.put(66.6, 'm'); //Map中键和值的类型是任意的，这也是Map强大的重要表现

m1.put(123L, 34);

System.*out*.println("3-> " + m1.size());

System.*out*.println("4-> " + m1);

System.*out*.println(m1.containsKey("three")); //true ontainsKey 不要写成了containKey

System.*out*.println(m1.containsValue(34)); //true

System.*out*.println(m1.containsValue(123L)); //false

}

}

/\*

在JDK 1.6中的运行结果是：

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

1-> {two=2, one=1, three=3}

2-> 3

3-> 5

4-> {two=2, one=1, three=3, 123=34, 66.6=m}

true

true

false

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

\*/

代码：

TestHashMap\_2.java

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

**class** Student

{

**private** **int** id;

**private** String name;

**private** **int** age;

**public** Student()

{

}

**public** Student(**int** id, String name, **int** age)

{

**this**.id = id;

**this**.name = name;

**this**.age = age;

}

**public** **int** hashCode()

{

**return** **this**.name.hashCode()\*id\*age;

}

**public** **boolean** equals(Object o)

{

Student s = (Student)o;

**return** **this**.name.equals(s.name) && **this**.id==s.id && **this**.age==s.age;

}

**public** String toString()

{

**return** id + " " + name + " " + age;

}

}

**public** **class** TestHashMap\_2

{

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

{

HashMap hm = **new** HashMap();

hm.put(1001, **new** Student(1001, "zhangsan", 20));

hm.put(1003, **new** Student(1003, "lisi", 30));

hm.put(1004, **new** Student(1004, "wangwu", 10));

hm.put(1002, **new** Student(1002, "baichi", 20));

//遍历所有的元素

System.*out*.println("hm容器中所有的元素是:");

Set s = hm.keySet();

Iterator it = s.iterator();

**while** (it.hasNext())

{

**int** Key = (Integer)it.next(); // (Integer) 不能省

System.*out*.println(hm.get(Key));

}

System.*out*.println("直接查找某一元素");

System.*out*.println( hm.get(1003) );

System.*out*.println( hm.get(1005) ); //如果找不到 则直接返回null

}

}

执行结果：

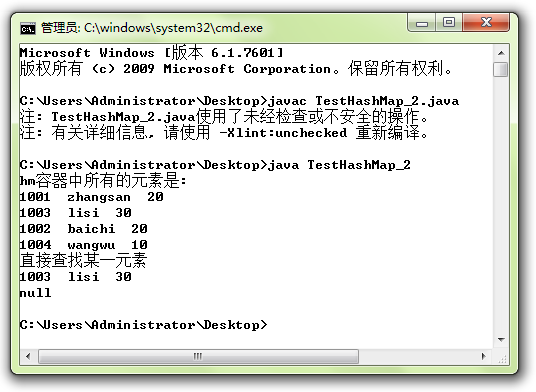


图14-8-1

## 14.9 泛型

泛型是用来限制传入容器、接口中的数据类型。

代码：

TestGenerics\_1.java

/\*

Comparable<T> 接口中有如下方法：

int compareTo(T o)

要想限制 传入Comparable接口的数据类型，可以利用泛型来实现，具体如下:

class Student implements Comparable<Student>

{

......

......

public int compareTo(Student ob) //注意Student不能写成了Object

{

}

\*/

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

**class** Student **implements** Comparable<Student>

{

**private** **int** id;

**private** String name;

**private** **int** age;

**public** Student()

{

}

**public** Student(**int** id, String name, **int** age)

{

**this**.id = id;

**this**.name = name;

**this**.age = age;

}

@Override

**public** String toString()

{

**return** id + " " + name + " " + age;

}

//形参不能写成Object ob 必须得写为 Student ob 否则编译时会报错，出错信息是：“Student 不是抽象的，并且未覆盖 java.lang.Comparable 中的抽象方法 compareTo(Student)”

// public int compareTo(Object ob) //本方法是错误的

// {

// Student s = (Student)ob;

// return this.id = s.id;

// }

//本方法是正确的

**public** **int** compareTo(Student s)

{

**return** **this**.id - s.id;

}

}

**public** **class** TestGenerics\_1

{

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

{

List<Student> li = **new** ArrayList<Student>();

li.add(**new** Student(1000, "zhangsan", 55));

li.add(**new** Student(1005, "lisi", 15));

li.add(**new** Student(1003, "wangwu", 32));

li.add(**new** Student(1001, "xiaojuan", 65));

System.*out*.println(li);

Collections.*sort*(li);

System.*out*.println(li);

}

}

代码：

TestGenerics\_2.java

/\*

2013年8月1日17:16:28

实现泛型之后的程序

\*/

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

**class** Student

{

**private** **int** id;

**private** String name;

**private** **int** age;

**public** Student()

{

}

**public** Student(**int** id, String name, **int** age)

{

**this**.id = id;

**this**.name = name;

**this**.age = age;

}

**public** **int** hashCode()

{

**return** **this**.name.hashCode()\*id\*age;

}

**public** **boolean** equals(Object o)

{

Student s = (Student)o;

**return** **this**.name.equals(s.name) && **this**.id==s.id && **this**.age==s.age;

}

**public** String toString()

{

**return** id + " " + name + " " + age;

}

}

**public** **class** TestGenerics\_2

{

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

{

HashMap<Integer, Student> hm = **new** HashMap<Integer, Student>();

hm.put(1001, **new** Student(1001, "zhangsan", 20));

hm.put(1003, **new** Student(1003, "lisi", 30));

hm.put(1004, **new** Student(1004, "wangwu", 10));

hm.put(1002, **new** Student(1002, "baichi", 20));

//遍历所有的元素

System.*out*.println("hm容器中所有的元素是:");

Set<Integer> s = hm.keySet();

Iterator<Integer> it = s.iterator();

**while** (it.hasNext())

{

**int** Key = (Integer)it.next(); // (Integer) 不能省

System.*out*.println(hm.get(Key));

}

System.*out*.println("直接查找某一元素");

System.*out*.println( hm.get(1003) );

System.*out*.println( hm.get(1005) ); //如果找不到 则直接返回null

}

}

# 15 网络编程

## 15.1 基础知识

### 基本概念

#### 网络程序

能够接受另一台计算机发送过来的数据或者能够向另一台计算机发送数据的程序叫做网络程序。

#### IP

能够在网络中唯一标示一台主机的编号就是IP；

网络中每台主机都必须有一个唯一的IP地址；

IP地址是一个逻辑地址；

因特网上的IP地址具有全球唯一性；

32位，4个字节，常用点分十进制的格式表示，例如：192.168.0.16

#### 端口号

一台计算机上可以同时运行多个网络程序；

一台计算机从网卡接受过来的数据到底应该交给本地的哪个网络程序来处理的，这是有端口号来决定的；

端口是用一个16位的数字来表示的，他的范围是0~65535，1024以下的端口号保留给预定义的服务。例如：80端口访问网页。25端口是用来邮件发送，oracle92默认的端口号是1521，tomcat默认的端口是8080。

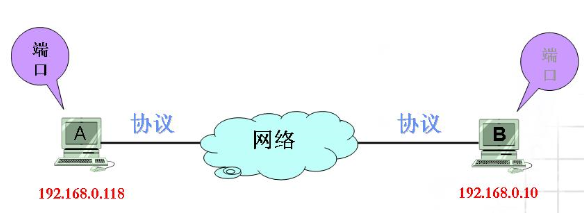


图15-1-1

#### 协议

为进行网络中的数据交换（通信）而建立的规则、标准或约定。

#### 常见协议

TCP：面向连接的可靠的传输协议。类似于打电话

UDP：是无连接的，不可靠的传输协议。类似于写信

### 套接字（socket）的引入

为了能够方便的开发网络应用软件，由美国伯克利大学在Unix上推出了一种应用程序访问通信协议的操作系统调用socket（套接字）。socket的出现，使程序员可以很方便地访问TCP/IP，从而开发各种网络应用的程序。

随着Unix的应用推广，套接字在编写网络软件中得到了极大的普及。后来，套接字又被引进了Windows等操作系统中。Java语言也引入了套接字编程模型。

## 15.2 基于UDP的socket编程

### 基于UDP的socket编程步骤

1. 定义码头

即：定义一个DatagramSocket对象ds

1. 定义可以用来接受或发送数据的集装箱

定义DatagramPacket对象dp

1. 在码头上用集装箱接受对方发送过来的数据（ds.receive(dp);）或者在码头上把集装箱中的数据发送给对方（ds.send(dp)）
2. 关闭码头（ds.close()）

代码：

TestUDPServer.java

**import** java.net.\*;

**import** java.io.\*;

**public** **class** TestUDPServer

{

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

{

//定义码头

DatagramSocket ds = **new** DatagramSocket(5678); //5678表示该码头占用的是5678这个编号，因为一台计算机可以有多个码头接收多个数据，这些码头用不同的编号来表示，这些编号的专业术语就是端口号

//定义可以用来接受数据的集装箱

**byte** buf[] = **new** **byte**[1024];

DatagramPacket dp = **new** DatagramPacket(buf, buf.length);

**try**

{

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

{

//在码头上用集装箱接受对方发送过来的数据

ds.receive(dp); //注意：本语句执行完毕就意味着，dp数据包中就已经含有了从客户端接收过来的数据

//从集装箱中取出对方发送过来的数据

ByteArrayInputStream bais = **new** ByteArrayInputStream(dp.getData()); //1、 ByteArrayInputStream的内核必须是个字节数组，并且是从该字节数组中读取数据 2、dp.getData()表示把dp集装箱中的数据转化为一个字节数组并返回该字节数组

DataInputStream dis = **new** DataInputStream(bais);

System.*out*.println(dis.readLong());

}

}

**catch** (Exception e)

{

e.printStackTrace();

ds.close(); //关闭码头

}

}

}

/\*

在JDK 1.6中的运行结果是：

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

10000

数据源自: 127.0.0.1 : 1464

10000

数据源自: 127.0.0.1 : 1471

10000

数据源自: 127.0.0.1 : 1474

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

\*/

代码：

TestUDPClient.java

**import** java.net.\*;

**import** java.io.\*;

**public** **class** TestUDPClient

{

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

{

//定义码头ds

DatagramSocket ds = **new** DatagramSocket();

//13行到23行完成的功能是： 定义可以发送数据的集装箱dp

**long** n = 10000L; //13行

ByteArrayOutputStream baos = **new** ByteArrayOutputStream(); //注意ByteArrayOutputStream的所有构造函数都没有byte[] buf这样的形参，即定义ByteArrayOutputStream流对象时是不能指定byte数组的，因为这个连接到的byte数组是由ByteArrayOutputStream自动生成的 9行 API:"public ByteArrayOutputStream(): 创建一个新的 byte 数组输出流。缓冲区的容量最初是 32 字节，如有必要可增加其大小。 "

//9行代码一旦执行完毕，意味着两点: 1、在内存中生成了一个大小为32个字节的byte数组 2、有一根叫做baos的管道已链接到了该byte数组中，并且可以通过这个管道向该数组中写入数据

//虽然此时可以通过baos向baos所连接到的在内存中分配好的byte数组中写入数据，但是ByteArrayOutputStream流并没有提供可以直接把long类型数据直接写入ByteArrayOutputStream流所连接到的byte数组中的方法, 简单说我们没法通过baos向baos所连接到的byte数组中写入long类型的数据, 查API文档可以发现: ByteArrayOutputStream流中并没有类似writeLong()这样的方法，但是DataOutputStream流中却有writeLong() writeFloat()等方法

DataOutputStream dos = **new** DataOutputStream(baos);

dos.writeLong(n); //把n变量所代表的10000L写入dos所依附的baos管道所连接到的内存中的大小为32字节的byte数组中

**byte**[] buf = baos.toByteArray(); //DataOutputStream 流中并没有toByteArray()方法，但是ByteArrayOutputStream 流中却有toByteArray()方法, 所以不可以把baos 改为dos，否则编译时会出错! ByteArrayOutputStream流中toByteArray()方法的含义，摘自API“创建一个新分配的 byte 数组。其大小是此输出流的当前大小，并且缓冲区的有效内容已复制到该数组中”

DatagramPacket dp = **new** DatagramPacket(buf, buf.length,

**new** InetSocketAddress("127.0.0.1", 5678)

); //23行

//在码头上把集装箱中的数据发送给对方

ds.send(dp);

//关闭码头

ds.close();

}

}

服务器端（Server）

DatagramSocket(int port)

DatagramPacket(byte[] buf, int length)

DataSocket.receive(DatagramPacket p)

DatagramSocket.close()

客户端（Client）

DatagramSocket()

DatagramPacket(byte[] buf, int length, InetAddress address, int port)

DataSocket.send(DatagramPacket p)

DatagramSocket.close()

图15-2-1

接收端程序编写

1. 调用DatagramSocket(int port)创建一个数据包套接字，并绑定到指定端口上；
2. 调用DatagramPacket(byte[] buf, int length)，建立一个字节数组以接收UDP包。
3. 调用DatagramSocket类的receive()，接收UDP包。
4. 最后关闭数据包套接字。

发送端程序编写

1. 调用DatagramSocket()创建一个数据包套接字；
2. 调用DatagramPacket(byte[] buf, int offset, int length, InetAddress address, int port)， 建立要发送的UDP包。
3. 调用DatagramSocket类的send()，发送UDP包。
4. 最后关闭数据包套接字。

### 基于UDP的socket编程应注意的问题

1. “127.0.0.1”是String，不是IP地址，IP地址在Java中是用java.net.InetAddress类表示
2. java.net.InetAddress并没有构造方法，因此要把“127.0.0.1”转化为IP地址

必须写成：

InetAddress.getByName(“127.0.0.1”);

不能写成：

new InetAddress(“127.0.0.1”);

1. DatagramPacket是个集装箱，既可以用来存放即将要发送到客户端的数据，也可以用来存放即将要接受的客户端数据
2. 如果UDP中两个网络程序要通信的话，这两个程序都必须得定义DatagramPacket包
3. 如果定义的DatagramPacket对象要用来接受客户端发送过来的数据，则不建议指定目的端口号和目的IP
4. 如果定义的DatagramPacket对象用来存储要发送到客户端的数据，则必须得指定目的端口号和目的IP
5. DatagramPacket所有的构造函数的第一个形参都是byte[] buf，，

即：所有的DatagramPacket对象的内核都是个byte数组

即：定义DatagramPacket对象时必须得借助byte数组来实现

## 15.3 基于TCP的socket编程

### 基于TCP的socket编程步骤

客户端（Client）

Socket(InetAddress address, int port)

OutputStream

Socket.getOutputStream()

InputStream

Socket.getInputStream()

Socket.close()

服务器端（Server）

ServerSocket(int port)

Socket accept()

OutputStream

Socket.getOutputStream()

InputStream

Socket.getInputStream()

Socket.close()

图15-3-1

服务器程序编写

1. 调用ServerSocket(int port)创建一个服务器端套接字，并绑定到指定端口上；
2. 调用accept()，监听连接请求，如果客户端请求连接，则接受连接，并返回和客户端匹配的套接字。
3. 调用Socket类的个体OutputStream()和getInputStream获取输出流和输入流，开始网络数据的发送和接收。
4. 最后关闭通信套接字。

客户端程序编写

1. 调用Socket()创建一个客户端套接字，该套接字会自动向服务器发送连接请求；
2. 调用Socket类的getOutputStream()和getInputStream获取输出流和输入流，开始网络数据的发送和接收。
3. 最后关闭通信套接字。

### 基于TCP的socket编程应注意的问题

TCP客户端

一旦new除了Socket对象，该对象就会自动向服务器端发送连接请求，如果连接不成功则程序立即终止

TCP服务器端

new出的ServerSocket对象并不会自动监听客户端的请求，只有调用了ServerSocket对象的accept方法时，才会监听客户端的请求。

# 16 枚举

## 16.1 原始的接口定义常量

**public** **interface** IConstants {

String *MON* = "Mon";

String *TUE* = "Tue";

String *WED* = "Wed";

String *THU* = "Thu";

String *FRI* = "Fri";

String *SAT* = "Sat";

String *SUN* = "Sun";

}

## 16.2 语法（定义）

创建枚举类型要使用 enum 关键字，隐含了所创建的类型都是 java.lang.Enum 类的子类（java.lang.Enum 是一个抽象类）。枚举类型符合通用模式 Class Enum<E extends Enum<E>>，而 E 表示枚举类型的名称。枚举类型的每一个值都将映射到 protected Enum(String name, int ordinal) 构造函数中，在这里，每个值的名称都被转换成一个字符串，并且序数设置表示了此设置被创建的顺序。

**package** com.hmw.test;

/\*\*

\* 枚举测试类

\* **@author** <a href="mailto:xuzhenkang@hotmail.com">徐振康</a>

\*/

**public** **enum** EnumTest {

MON, TUE, WED, THU, FRI, SAT, SUN;

}

这段代码实际上调用了7次 Enum(String name, int ordinal)：

**new** Enum<EnumTest>("MON",0);

**new** Enum<EnumTest>("TUE",1);

**new** Enum<EnumTest>("WED",2);

... ...

## 16.3 遍历、switch 等常用操作

对enum进行遍历和switch的操作示例代码：

**public** **class** Test {

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

**for** (EnumTest e : EnumTest.values()) {

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

}

System.*out*.println("----------------我是分隔线----------------");

EnumTest test = EnumTest.TUE;

**switch** (test) {

**case** MON:

System.*out*.println("今天是星期一");

**break**;

**case** TUE:

System.*out*.println("今天是星期二");

**break**;

// ... ...

**default**:

System.*out*.println(test);

**break**;

}

}

}

输出结果：

MON

TUE

WED

THU

FRI

SAT

SUN

----------------我是分隔线----------------

今天是星期二

## 16.4 enum 对象的常用方法介绍

**int** compareTo(E o)

比较此枚举与指定对象的顺序。

Class<E> getDeclaringClass()

返回与此枚举常量的枚举类型相对应的 Class 对象。

String name()

返回此枚举常量的名称，在其枚举声明中对其进行声明。

**int** ordinal()

返回枚举常量的序数（它在枚举声明中的位置，其中初始常量序数为零）。

String toString()

返回枚举常量的名称，它包含在声明中。

**static** <T **extends** Enum<T>> TvalueOf(Class<T> enumType, String name)

返回带指定名称的指定枚举类型的枚举常量。

**public** **class** Test {

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

EnumTest test = EnumTest.TUE;

//compareTo(E o)

**switch** (test.compareTo(EnumTest.MON)) {

**case** -1:

System.*out*.println("TUE 在 MON 之前");

**break**;

**case** 1:

System.*out*.println("TUE 在 MON 之后");

**break**;

**default**:

System.*out*.println("TUE 与 MON 在同一位置");

**break**;

}

//getDeclaringClass()

System.*out*.println("getDeclaringClass(): " + test.getDeclaringClass().getName());

//name() 和 toString()

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

System.*out*.println("toString(): " + test.toString());

//ordinal()， 返回值是从 0 开始

System.*out*.println("ordinal(): " + test.ordinal());

}

}

输出结果：

TUE 在 MON 之后

getDeclaringClass(): com.hmw.test.EnumTest

name(): TUE

toString(): TUE

ordinal(): 1

## 16.5 给 enum 自定义属性和方法

给 enum 对象加一下 value 的属性和 getValue() 的方法：

**package** com.xzk.test;

/\*\*

\* 枚举测试类

\*

\* **@author** <a href="mailto:xuzhenkang@hotmail.com">徐振康</a>

\*/

**public** **enum** EnumTest {

MON(1), TUE(2), WED(3), THU(4), FRI(5), SAT(6) {

@Override

**public** **boolean** isRest() {

**return** **true**;

}

},

SUN(0) {

@Override

**public** **boolean** isRest() {

**return** **true**;

}

};

**private** **int** value;

**private** EnumTest(**int** value) {

**this**.value = value;

}

**public** **int** getValue() {

**return** value;

}

**public** **boolean** isRest() {

**return** **false**;

}

}

**public** **class** Test {

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

System.out.println("EnumTest.FRI 的 value = " + EnumTest.FRI.getValue());

}

}

输出结果：

EnumTest.FRI 的 value = 5

## 16.6 EnumSet，EnumMap 的应用

**public** **class** Test {

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

// EnumSet的使用

EnumSet<EnumTest> weekSet = EnumSet.allOf(EnumTest.**class**);

**for** (EnumTest day : weekSet) {

System.*out*.println(day);

}

// EnumMap的使用

EnumMap<EnumTest, String> weekMap = **new** EnumMap(EnumTest.**class**);

weekMap.put(EnumTest.MON, "星期一");

weekMap.put(EnumTest.TUE, "星期二");

// ... ...

**for** (Iterator<Entry<EnumTest, String>> iter = weekMap.entrySet().iterator(); iter.hasNext();) {

Entry<EnumTest, String> entry = iter.next();

System.*out*.println(entry.getKey().name() + ":" + entry.getValue());

}

}

}

## 16.7 enum原理分析

enum 的语法结构尽管和 class 的语法不一样，但是经过编译器编译之后产生的是一个class文件。该class文件经过反编译可以看到实际上是生成了一个类，该类继承了java.lang.Enum<E>。EnumTest 经过反编译(javap com.xzk.test.EnumTest 命令)之后得到的内容如下：

**public** **class** com.xzk.test.EnumTest **extends** java.lang.Enum{

**public** **static** **final** com.xzk.test.EnumTest MON;

**public** **static** **final** com.xzk.test.EnumTest TUE;

**public** **static** **final** com.xzk.test.EnumTest WED;

**public** **static** **final** com.xzk.test.EnumTest THU;

**public** **static** **final** com.xzk.test.EnumTest FRI;

**public** **static** **final** com.xzk.test.EnumTest SAT;

**public** **static** **final** com.xzk.test.EnumTest SUN;

**static** {};

**public** **int** getValue();

**public** **boolean** isRest();

**public** **static** com.xzk.test.EnumTest[] values();

**public** **static** com.xzk.test.EnumTest valueOf(java.lang.String);

com.hmw.test.EnumTest(java.lang.String, **int**, **int**, com.xzk.test.EnumTest);

}

所以，实际上 enum 就是一个 class，只不过 java 编译器帮我们做了语法的解析和编译而已。

## 16.8 总结

可以把 enum 看成是一个普通的 class，它们都可以定义一些属性和方法，不同之处是：enum 不能使用 extends 关键字继承其他类，因为 enum 已经继承了 java.lang.Enum（java是单一继承）。

# 17 注解(Annotation)

## 17.1 注解基本概念

### 什么是注解

**Annotation（注解）就是Java提供了一种元程序中的元素关联任何信息和着任何元数据（metadata）的途径和方法。**Annotion(注解)是一个接口，程序可以通过反射来获取指定程序元素的Annotion对象，然后通过Annotion对象来获取注解里面的元数据。

　　Annotation(注解)是JDK5.0及以后版本引入的。它可以用于创建文档，跟踪代码中的依赖性，甚至执行基本编译时检查。从某些方面看，annotation就像修饰符一样被使用，并应用于包、类 型、构造方法、方法、成员变量、参数、本地变量的声明中。这些信息被存储在Annotation的“name=value”结构对中。

**Annotation的成员**在Annotation类型中以无参数的方法的形式被声明。其方法名和返回值定义了该成员的名字和类型。在此有一个特定的默认语法：允许声明任何Annotation成员的默认值：一个Annotation可以将name=value对作为没有定义默认值的Annotation成员的值，当然也可以使用name=value对来覆盖其它成员默认值。这一点有些近似类的继承特性，父类的构造函数可以作为子类的默认构造函数，但是也可以被子类覆盖。

　　Annotation能被用来为某个程序元素（类、方法、成员变量等）关联任何的信息。需要注意的是，这里存在着一个**基本的规则：Annotation不能影响程序代码的执行，无论增加、删除 Annotation，代码都始终如一的执行。**另外，尽管一些annotation通过java的反射api方法在运行时被访问，而java语言解释器在工作时忽略了这些annotation。正是由于java虚拟机忽略了Annotation，导致了annotation类型在代码中是“不起作用”的； 只有通过某种配套的工具才会对annotation类型中的信息进行访问和处理。本文中将涵盖标准的Annotation和meta-annotation类型，陪伴这些annotation类型的工具是java编译器（当然要以某种特殊的方式处理它们）。

### 什么是metadata（元数据）

　　元数据从metadata一词译来，就是“关于数据的数据”的意思。  
　　元数据的功能作用有很多，比如：你可能用过Javadoc的注释自动生成文档。这就是元数据功能的一种。总的来说，元数据可以用来创建文档，跟踪代码的依赖性，执行编译时格式检查，代替已有的配置文件。如果要对于元数据的作用进行分类，目前还没有明确的定义，不过我们可以根据它所起的作用，大致可分为三类：   
　　　　1. 编写文档：通过代码里标识的元数据生成文档  
　　　　2. 代码分析：通过代码里标识的元数据对代码进行分析  
　　　　3. 编译检查：通过代码里标识的元数据让编译器能实现基本的编译检查  
　　在Java中元数据以标签的形式存在于Java代码中，元数据标签的存在并不影响程序代码的编译和执行，它只是被用来生成其它的文件或针在运行时知道被运行代码的描述信息。  
　　综上所述：  
　　　　第一，元数据以标签的形式存在于Java代码中。  
　　　　第二，元数据描述的信息是类型安全的，即元数据内部的字段都是有明确类型的。  
　　　　第三，元数据需要编译器之外的工具额外的处理用来生成其它的程序部件。  
　　　　第四，元数据可以只存在于Java源代码级别，也可以存在于编译之后的Class文件内部。

### Annotation和Annotation类型

**Annotation：**

　　Annotation使用了在java5.0所带来的新语法，它的行为十分类似public、final这样的修饰符。每个Annotation具有一个名字和成员个数>=0。每个Annotation的成员具有被称为name=value对的名字和值（就像javabean一样），name=value装载了Annotation的信息。

**Annotation类型：**

Annotation类型定义了Annotation的名字、类型、成员默认值。一个Annotation类型可以说是一个特殊的java接口，它的成员变量是受限制的，而声明Annotation类型时需要使用新语法。当我们通过java反射api访问Annotation时，返回值将是一个实现了该 annotation类型接口的对象，通过访问这个对象我们能方便的访问到其Annotation成员。后面的章节将提到在java5.0的 java.lang包里包含的3个标准Annotation类型。

### 注解的分类

　　根据注解参数的个数，我们可以将注解分为三类：

　　　　1.标记注解:一个没有成员定义的Annotation类型被称为标记注解。这种Annotation类型仅使用自身的存在与否来为我们提供信息。比如后面的系统注解@Override;

　　　　2.单值注解

　　　　3.完整注解

　　根据注解使用方法和用途，我们可以将Annotation分为三类：

　　　　1.JDK内置系统注解

　　　　2.元注解

　　　　3.自定义注解

## 17.2系统内置标准注解

　　注解的语法比较简单，除了@符号的使用外，他基本与Java固有的语法一致，JavaSE中内置三个标准注解，定义在java.lang中：  
　　　　@Override：用于修饰此方法覆盖了父类的方法;  
　　　　@Deprecated：用于修饰已经过时的方法;  
　　　　@SuppressWarnnings:用于通知java编译器禁止特定的编译警告。

　　下面我们依次看看三个内置标准注解的作用和使用场景。

### @Override，限定重写父类方法

　　@Override 是一个标记注解类型，它被用作标注方法。它说明了被标注的方法重载了父类的方法，起到了断言的作用。如果我们使用了这种Annotation在一个没有覆盖父类方法的方法时，java编译器将以一个编译错误来警示。这个annotaton常常在我们试图覆盖父类方法而确又写错了方法名时发挥威力。使用方法极其简单：在使用此annotation时只要在被修饰的方法前面加上@Override即可。下面的代码是一个使用@Override修饰一个企图重载父类的displayName()方法，而又存在拼写错误的实例：

**public** **class** Fruit {

**public** **void** displayName(){

System.*out*.println("水果的名字是：\*\*\*\*\*");

}

}

**class** Orange **extends** Fruit {

@Override

**public** **void** displayName(){

System.*out*.println("水果的名字是：桔子");

}

}

**class** Apple **extends** Fruit {

@Override

**public** **void** displayname(){

System.*out*.println("水果的名字是：苹果");

}

}

Orange 类编译不会有任何问题，Apple 类在编译的时候会提示相应的错误。@Override注解只能用于方法，不能用于其他程序元素。

### @Deprecated，标记已过时

　　同 样Deprecated也是一个标记注解。当一个类型或者类型成员使用@Deprecated修饰的话，编译器将不鼓励使用这个被标注的程序元素。而且这种修饰具有一定的 “延续性”：如果我们在代码中通过继承或者覆盖的方式使用了这个过时的类型或者成员，虽然继承或者覆盖后的类型或者成员并不是被声明为 @Deprecated，但编译器仍然要报警。

　　值得注意，@Deprecated这个annotation类型和javadoc中的 @deprecated这个tag是有区别的：前者是java编译器识别的，而后者是被javadoc工具所识别用来生成文档（包含程序成员为什么已经过 时、它应当如何被禁止或者替代的描述）。

　　在java5.0，java编译器仍然象其从前版本那样寻找@deprecated这个javadoc tag，并使用它们产生警告信息。但是这种状况将在后续版本中改变，我们应在现在就开始使用@Deprecated来修饰过时的方法而不是 @deprecated javadoc tag。

　　下面一段程序中使用了@Deprecated注解标示方法过期，同时在方法注释中用@deprecated tag 标示该方法已经过时，代码如下：

**class** AppleService {

**public** **void** displayName(){

System.*out*.println("水果的名字是：苹果");

}

/\*\*

\* **@deprecated** 该方法已经过期，不推荐使用

\*/

@Deprecated

**public** **void** ~~showTaste~~(){

System.*out*.println("水果的苹果的口感是：脆甜");

}

**public** **void** showTaste(**int** typeId){

**if**(typeId==1){

System.*out*.println("水果的苹果的口感是：酸涩");

}

**else** **if**(typeId==2){

System.*out*.println("水果的苹果的口感是：绵甜");

}

**else**{

System.*out*.println("水果的苹果的口感是：脆甜");

}

}

}

**public** **class** FruitRun {

/\*\*

\* **@param** args

\*/

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

Apple apple=**new** Apple();

apple.displayName();

AppleService appleService=**new** AppleService();

appleService.~~showTaste~~();

appleService.showTaste(0);

appleService.showTaste(2);

}

}

AppleService类的showTaste() 方法被@Deprecated标注为过时方法，在FruitRun类中使用的时候，编译器会给出该方法已过期，不推荐使用的提示。

### @SuppressWarnnings，抑制编译器警告

　　@SuppressWarnings 被用于有选择的关闭编译器对类、方法、成员变量、变量初始化的警告。在java5.0，sun提供的javac编译器为我们提供了-Xlint选项来使编译器对合法的程序代码提出警告，此种警告从某种程度上代表了程序错误。例如当我们使用一个generic collection类而又没有提供它的类型时，编译器将提示出"unchecked warning"的警告。通常当这种情况发生时，我们就需要查找引起警告的代码。如果它真的表示错误，我们就需要纠正它。例如如果警告信息表明我们代码中的switch语句没有覆盖所有可能的case，那么我们就应增加一个默认的case来避免这种警告。

　　有时我们无法避免这种警告，例如，我们使用必须和非generic的旧代码交互的generic collection类时，我们不能避免这个unchecked warning。此时@SuppressWarning就要派上用场了，在调用的方法前增加@SuppressWarnings修饰，告诉编译器停止对此方法的警告。

　　SuppressWarning不是一个标记注解。它有一个类型为String[]的成员，这个成员的值为被禁止的警告名。对于javac编译器来讲，被-Xlint选项有效的警告 名也同样对@SuppressWarings有效，同时编译器忽略掉无法识别的警告名。

　　annotation语法允许在annotation名后跟括号，括号中是使用逗号分割的name=value对用于为annotation的成员赋值。实例如下：

**public** **class** FruitService {

@SuppressWarnings(value={ "rawtypes", "unchecked" })

**public** **static** List<Fruit> getFruitList(){

List<Fruit> fruitList=**new** ArrayList();

**return** fruitList;

}

@SuppressWarnings({ "rawtypes", "unchecked" })

**public** **static** List<Fruit> getFruit(){

List<Fruit> fruitList=**new** ArrayList();

**return** fruitList;

}

@SuppressWarnings("unused")

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

List<String> strList=**new** ArrayList<String>();

}

}

在这个例子中SuppressWarnings annotation类型只定义了一个单一的成员，所以只有一个简单的value={...}作为name=value对。又由于成员值是一个数组，故使用大括号来声明数组值。注意：我们可以在下面的情况中缩写annotation：当annotation只有单一成员，并成员命名为"value="。这时可以省去"value="。比如将上面方法getFruit()的SuppressWarnings annotation就是缩写的。

　 SuppressWarnings注解的常见参数值的简单说明：

　　　　1.deprecation：使用了不赞成使用的类或方法时的警告；

　　　　2.unchecked：执行了未检查的转换时的警告，例如当使用集合时没有用泛型 (Generics) 来指定集合保存的类型;

　　　　3.fallthrough：当 Switch 程序块直接通往下一种情况而没有 Break 时的警告;

　　　　4.path：在类路径、源文件路径等中有不存在的路径时的警告;

　　　　5.serial：当在可序列化的类上缺少 serialVersionUID 定义时的警告;

　　　　6.finally：任何 finally 子句不能正常完成时的警告;

　　　　7.all：关于以上所有情况的警告。

## 17.3 元注解

元注解的作用就是负责注解其他注解。Java5.0定义了4个标准的meta-annotation类型，它们被用来提供对其它 annotation类型作说明。Java5.0定义的元注解：

　　　　1.@Target,

　　　　2.@Retention,

　　　　3.@Documented,

　　　　4.@Inherited

　　这些类型和它们所支持的类在java.lang.annotation包中可以找到。下面我们看一下每个元注解的作用和相应分参数的使用说明。

### @Target

@Target说明了Annotation所修饰的对象范围：Annotation可被用于 packages、types（类、接口、枚举、Annotation类型）、类型成员（方法、构造方法、成员变量、枚举值）、方法参数和本地变量（如循环变量、catch参数）。在Annotation类型的声明中使用了target可更加明晰其修饰的目标。

**作用：用于描述注解的使用范围（即：被描述的注解可以用在什么地方）**

**取值(ElementType)有：**

　　　　1.CONSTRUCTOR:用于描述构造器  
　　　　2.FIELD:用于描述域  
　　　　3.LOCAL\_VARIABLE:用于描述局部变量  
　　　　4.METHOD:用于描述方法  
　　　　5.PACKAGE:用于描述包  
　　　　6.PARAMETER:用于描述参数  
　　　　7.TYPE:用于描述类、接口(包括注解类型) 或enum声明

　　使用实例：

@Target(ElementType.TYPE)

**public** **@interface** Table {

/\*\*

\* 数据表名称注解，默认值为类名称

\* **@return**

\*/

**public** String tableName() **default** "className";

}

@Target(ElementType.FIELD)

**public** **@interface** NoDBColumn {

}

注解Table 可以用于注解类、接口(包括注解类型) 或enum声明,而注解NoDBColumn仅可用于注解类的成员变量。

### @Retention

**@Retention**定义了该Annotation被保留的时间长短：某些Annotation仅出现在源代码中，而被编译器丢弃；而另一些却被编译在class文件中；编译在class文件中的Annotation可能会被虚拟机忽略，而另一些在class被装载时将被读取（请注意并不影响class的执行，因为Annotation与class在使用上是被分离的）。使用这个meta-Annotation可以对 Annotation的“生命周期”限制。

**作用：表示需要在什么级别保存该注释信息，用于描述注解的生命周期（即：被描述的注解在什么范围内有效）**

**取值（RetentionPoicy）有：**

　　　　1.SOURCE:在源文件中有效（即源文件保留）

　　　　2.CLASS:在class文件中有效（即class保留）

　　　　3.RUNTIME:在运行时有效（即运行时保留）

　　Retention meta-annotation类型有唯一的value作为成员，它的取值来自java.lang.annotation.RetentionPolicy的枚举类型值。具体实例如下：

@Target(ElementType.FIELD)

@Retention(RetentionPolicy.RUNTIME)

**public** **@interface** Column {

**public** String name() **default** "fieldName";

**public** String setFuncName() **default** "setField";

**public** String getFuncName() **default** "getField";

**public** **boolean** defaultDBValue() **default** **false**;

}

Column注解的的RetentionPolicy的属性值是RUTIME,这样注解处理器可以通过反射，获取到该注解的属性值，从而去做一些运行时的逻辑处理

### @Documented

**@**Documented用于描述其它类型的annotation应该被作为被标注的程序成员的公共API，因此可以被例如javadoc此类的工具文档化。Documented是一个标记注解，没有成员。

@Target(ElementType.FIELD)

@Retention(RetentionPolicy.RUNTIME)

@Documented

**public** **@interface** Column {

**public** String name() **default** "fieldName";

**public** String setFuncName() **default** "setField";

**public** String getFuncName() **default** "getField";

**public** **boolean** defaultDBValue() **default** **false**;

}

### @Inherited

@Inherited 元注解是一个标记注解，@Inherited阐述了某个被标注的类型是被继承的。如果一个使用了@Inherited修饰的annotation类型被用于一个class，则这个annotation将被用于该class的子类。

　　注意：@Inherited annotation类型是被标注过的class的子类所继承。类并不从它所实现的接口继承annotation，方法并不从它所重载的方法继承annotation。

　　当@Inherited annotation类型标注的annotation的Retention是RetentionPolicy.RUNTIME，则反射API增强了这种继承性。如果我们使用java.lang.reflect去查询一个@Inherited annotation类型的annotation时，反射代码检查将展开工作：检查class和其父类，直到发现指定的annotation类型被发现，或者到达类继承结构的顶层。

　　实例代码：

/\*\*

\*

\* **@author** xzk

\*

\*/

@Inherited

**public** **@interface** Greeting {

**public** **enum** FontColor{ *BULE*,*RED*,*GREEN*};

String name();

FontColor fontColor() **default** FontColor.*GREEN*;

}

## 17.4 自定义注解

使用@interface自定义注解时，自动继承了java.lang.annotation.Annotation接口，由编译程序自动完成其他细节。在定义注解时，不能继承其他的注解或接口。@interface用来声明一个注解，其中的每一个方法实际上是声明了一个配置参数。方法的名称就是参数的名称，返回值类型就是参数的类型（返回值类型只能是基本类型、Class、String、enum）。可以通过default来声明参数的默认值。

**定义注解格式：**  
　　public @interface 注解名 {定义体}

**注解参数的可支持数据类型：**

　　　　1.所有基本数据类型（int,float,boolean,byte,double,char,long,short)  
　　　　2.String类型  
　　　　3.Class类型  
　　　　4.enum类型  
　　　　5.Annotation类型  
　　　　6.以上所有类型的数组

　　Annotation类型里面的参数该怎么设定:   
　　第一,只能用public或默认(default)这两个访问权修饰.例如,String value();这里把方法设为defaul默认类型；　 　  
　　第二,参数成员只能用基本类型byte,short,char,int,long,float,double,boolean八种基本数据类型和 String,Enum,Class,annotations等数据类型,以及这一些类型的数组.例如,String value();这里的参数成员就为String;　　  
　　第三,如果只有一个参数成员,最好把参数名称设为"value",后加小括号.例:下面的例子FruitName注解就只有一个参数成员。

　　简单的自定义注解和使用注解实例：

**package** annotation;

**import** java.lang.annotation.Documented;

**import** java.lang.annotation.ElementType;

**import** java.lang.annotation.Retention;

**import** java.lang.annotation.RetentionPolicy;

**import** java.lang.annotation.Target;

/\*\*

\* 水果名称注解

\* **@author** xzk

\*

\*/

@Target(ElementType.FIELD)

@Retention(RetentionPolicy.RUNTIME)

@Documented

**public** **@interface** FruitName {

String value() **default** "";

}

**package** annotation;

**import** java.lang.annotation.Documented;

**import** java.lang.annotation.ElementType;

**import** java.lang.annotation.Retention;

**import** java.lang.annotation.RetentionPolicy;

**import** java.lang.annotation.Target;

/\*\*

\* 水果颜色注解

\* **@author** xzk

\*

\*/

@Target(ElementType.FIELD)

@Retention(RetentionPolicy.RUNTIME)

@Documented

**public** **@interface** FruitColor {

/\*\*

\* 颜色枚举

\* **@author** xzk

\*

\*/

**public** **enum** Color{ BULE,RED,GREEN};

/\*\*

\* 颜色属性

\* **@return**

\*/

Color fruitColor() **default** Color.GREEN;

}

**package** annotation;

**import** annotation.FruitColor.Color;

**public** **class** Apple {

@FruitName("Apple")

**private** String appleName;

@FruitColor(fruitColor=Color.RED)

**private** String appleColor;

**public** **void** setAppleColor(String appleColor) {

**this**.appleColor = appleColor;

}

**public** String getAppleColor() {

**return** appleColor;

}

**public** **void** setAppleName(String appleName) {

**this**.appleName = appleName;

}

**public** String getAppleName() {

**return** appleName;

}

**public** **void** displayName(){

System.out.println("水果的名字是：苹果");

}

}

## 17.5 注解元素的默认值

注解元素必须有确定的值，要么在定义注解的默认值中指定，要么在使用注解时指定，非基本类型的注解元素的值不可为null。因此, 使用空字符串或0作为默认值是一种常用的做法。这个约束使得处理器很难表现一个元素的存在或缺失的状态，因为每个注解的声明中，所有元素都存在，并且都具有相应的值，为了绕开这个约束，我们只能定义一些特殊的值，例如空字符串或者负数，一次表示某个元素不存在，在定义注解时，这已经成为一个习惯用法。例如：

**package** annotation;

**import** java.lang.annotation.Documented;

**import** java.lang.annotation.ElementType;

**import** java.lang.annotation.Retention;

**import** java.lang.annotation.RetentionPolicy;

**import** java.lang.annotation.Target;

/\*\*

\* 水果供应者注解

\* **@author** xzk

\*

\*/

@Target(ElementType.FIELD)

@Retention(RetentionPolicy.RUNTIME)

@Documented

**public** **@interface** FruitProvider {

/\*\*

\* 供应商编号

\* **@return**

\*/

**public** **int** id() **default** -1;

/\*\*

\* 供应商名称

\* **@return**

\*/

**public** String name() **default** "";

/\*\*

\* 供应商地址

\* **@return**

\*/

**public** String address() **default** "";

}

定义了注解，并在需要的时候给相关类，类属性加上注解信息，如果没有响应的注解信息处理流程，注解可以说是没有实用价值。如何让注解真真的发挥作用，主要就在于注解处理方法，下一步我们将学习注解信息的获取和处理！

## 17.6注解处理器

如果没有用来读取注解的方法和工作，那么注解也就不会比注释更有用处了。使用注解的过程中，很重要的一部分就是创建于使用注解处理器。Java SE5扩展了反射机制的API，以帮助程序员快速的构造自定义注解处理器。

### 注解处理器类库(java.lang.reflect.AnnotatedElement)

　　Java使用Annotation接口来代表程序元素前面的注解，该接口是所有Annotation类型的父接口。除此之外，Java在java.lang.reflect 包下新增了AnnotatedElement接口，该接口代表程序中可以接受注解的程序元素，该接口主要有如下几个实现类：

　　Class：类定义

　　Constructor：构造器定义

　　Field：累的成员变量定义

　　Method：类的方法定义

　　Package：类的包定义

　　java.lang.reflect 包下主要包含一些实现反射功能的工具类，实际上，java.lang.reflect 包所有提供的反射API扩充了读取运行时Annotation信息的能力。当一个Annotation类型被定义为运行时的Annotation后，该注解才能是运行时可见，当class文件被装载时被保存在class文件中的Annotation才会被虚拟机读取。

　　AnnotatedElement 接口是所有程序元素（Class、Method和Constructor）的父接口，所以程序通过反射获取了某个类的AnnotatedElement对象之后，程序就可以调用该对象的如下四个个方法来访问Annotation信息：

　　方法1：<T extends Annotation> T getAnnotation(Class<T> annotationClass): 返回改程序元素上存在的、指定类型的注解，如果该类型注解不存在，则返回null。

　　方法2：Annotation[] getAnnotations():返回该程序元素上存在的所有注解。

　　方法3：boolean is AnnotationPresent(Class<?extends Annotation> annotationClass):判断该程序元素上是否包含指定类型的注解，存在则返回true，否则返回false.

　　方法4：Annotation[] getDeclaredAnnotations()：返回直接存在于此元素上的所有注释。与此接口中的其他方法不同，该方法将忽略继承的注释。（如果没有注释直接存在于此元素上，则返回长度为零的一个数组。）该方法的调用者可以随意修改返回的数组；这不会对其他调用者返回的数组产生任何影响。

　　一个简单的注解处理器：

/\*\*\*\*\*\*\*\*\*\*\*注解声明\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\* 水果名称注解

\* **@author** xzk

\*

\*/

@Target(ElementType.FIELD)

@Retention(RetentionPolicy.RUNTIME)

@Documented

**public** **@interface** FruitName {

String value() **default** "";

}

/\*\*

\* 水果颜色注解

\* **@author** xzk

\*

\*/

@Target(ElementType.FIELD)

@Retention(RetentionPolicy.RUNTIME)

@Documented

**public** **@interface** FruitColor {

/\*\*

\* 颜色枚举

\* **@author** xzk

\*

\*/

**public** **enum** Color{ *BULE*,*RED*,*GREEN*};

/\*\*

\* 颜色属性

\* **@return**

\*/

Color fruitColor() **default** Color.*GREEN*;

}

/\*\*

\* 水果供应者注解

\* **@author** xzk

\*

\*/

@Target(ElementType.FIELD)

@Retention(RetentionPolicy.RUNTIME)

@Documented

**public** **@interface** FruitProvider {

/\*\*

\* 供应商编号

\* **@return**

\*/

**public** **int** id() **default** -1;

/\*\*

\* 供应商名称

\* **@return**

\*/

**public** String name() **default** "";

/\*\*

\* 供应商地址

\* **@return**

\*/

**public** String address() **default** "";

}

/\*\*\*\*\*\*\*\*\*\*\*注解使用\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**public** **class** Apple {

@FruitName("Apple")

**private** String appleName;

@FruitColor(fruitColor=Color.RED)

**private** String appleColor;

@FruitProvider(id=1,name="陕西红富士集团",address="陕西省西安市延安路89号红富士大厦")

**private** String appleProvider;

**public** **void** setAppleColor(String appleColor) {

**this**.appleColor = appleColor;

}

**public** String getAppleColor() {

**return** appleColor;

}

**public** **void** setAppleName(String appleName) {

**this**.appleName = appleName;

}

**public** String getAppleName() {

**return** appleName;

}

**public** **void** setAppleProvider(String appleProvider) {

**this**.appleProvider = appleProvider;

}

**public** String getAppleProvider() {

**return** appleProvider;

}

**public** **void** displayName(){

System.*out*.println("水果的名字是：苹果");

}

}

/\*\*\*\*\*\*\*\*\*\*\*注解处理器\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**public** **class** FruitInfoUtil {

**public** **static** **void** getFruitInfo(Class<?> clazz){

String strFruitName=" 水果名称：";

String strFruitColor=" 水果颜色：";

String strFruitProvicer="供应商信息：";

Field[] fields = clazz.getDeclaredFields();

**for**(Field field :fields){

**if**(field.isAnnotationPresent(FruitName.**class**)){

FruitName fruitName = (FruitName) field.getAnnotation(FruitName.**class**);

strFruitName=strFruitName+fruitName.value();

System.*out*.println(strFruitName);

}

**else** **if**(field.isAnnotationPresent(FruitColor.**class**)){

FruitColor fruitColor= (FruitColor) field.getAnnotation(FruitColor.**class**);

strFruitColor=strFruitColor+fruitColor.fruitColor().toString();

System.*out*.println(strFruitColor);

}

**else** **if**(field.isAnnotationPresent(FruitProvider.**class**)){

FruitProvider fruitProvider= (FruitProvider) field.getAnnotation(FruitProvider.**class**);

strFruitProvicer=" 供应商编号："+fruitProvider.id()+" 供应商名称："+fruitProvider.name()+" 供应商地址："+fruitProvider.address();

System.*out*.println(strFruitProvicer);

}

}

}

}

/\*\*\*\*\*\*\*\*\*\*\*输出结果\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**public** **class** FruitRun {

/\*\*

\* **@param** args

\*/

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

FruitInfoUtil.getFruitInfo(Apple.**class**);

}

}

====================================

水果名称：Apple

水果颜色：RED

供应商编号：1 供应商名称：陕西红富士集团 供应商地址：陕西省西安市延安路89号红富士大厦

Java注解的基础知识点（见下面导图）基本都过了一遍

