# 一、Java基础知识

## 1 平台

**一、企业版(Java EE)**  
 Java EE是一种利用Java2平台来简化企业解决方案的开发、部署和管理相关的复杂问题的体系结构。J2EE技术的基础就是核心Java平台或Java2平台的标准版，Java EE不仅巩固了标准版中的许多优点，例如“编写一次、随处运行”的特性、方便存取数据库的JDBC API、CORBA技术以及能够在Internet应用中保护数据的安全模式等等，同时还提供了对 EJB（EnterpriseJavaBeans）、Java Servlets API、JSP（Java Server Pages）以及XML技术的全面支持。其最终目的就是成为一个能够使企业开发者大幅缩短投放市场时间的体系结构。  
 Java EE体系结构提供中间层集成框架用来满足无需太多费用而又需要高可用性、高可靠性以及可扩展性的应用的需求。通过提供统一的开发平台，J2EE降低了开发多层应用的费用和复杂性，同时提供对现有应用程序集成强有力支持，完全支持EJB，有良好的向导支持打包和部署应用，添加目录支持，增强了安全机制，提高了性能。  
  
**二、标准版(Java SE)**  
 Java SE 是Java平台标准版的简称（Java Platform, Standard Edition） (also known as Java 2 Platform) ，用于开发和部署桌面、服务器以及嵌入设备和实时环境中的Java应用程序。Java SE包括用于开发Java Web服务的类库，同时，Java SE为Java EE提供了基础。  
 Java SE（Java Platform, Standard Edition，Java标准版）就是基于JDK和JRE的。  
  
**三、Micro版(Java ME)**  
 Java ME是Java微版的简称（Java Platform,Micro Edition），是一个技术和规范的集合，它为移动设备（包括消费类产品、嵌入式设备、高级移动设备等）提供了基于Java环境的开发与应用平台。Java ME目前分为两类配置，一类是面向小型移动设备的CLDC（Connected Limited Device Profile），一类是面向功能更强大的移动设备如智能手机和机顶盒，称为CDC（Connected Device Profile CDC）。  
 Java ME有自己的类库，其中CLDC使用的是专用的Java虚拟机叫做KVM。

## 2 JDK

JDK是 [Java](http://baike.baidu.com/item/Java/85979) 语言的[软件开发工具包](http://baike.baidu.com/item/%E8%BD%AF%E4%BB%B6%E5%BC%80%E5%8F%91%E5%B7%A5%E5%85%B7%E5%8C%85)，主要用于移动设备、嵌入式设备上的java应用程序。JDK是整个java开发的核心，它包含了JAVA的运行环境（JVM+Java系统类库）和JAVA工具。

**JDK(Java Development Kit)** 是 Java 语言的软件开发工具包([SDK](http://baike.baidu.com/item/SDK))。

**SE(J2SE)，standard edition**，标准版，是我们通常用的一个版本，从JDK 5.0开始，改名为Java SE。

**EE(J2EE)，enterprise edition**，企业版，使用这种JDK开发J2EE应用程序，从JDK 5.0开始，改名为Java EE。

**ME(J2ME)，micro edition**，主要用于移动设备、嵌入式设备上的java应用程序，从JDK 5.0开始，改名为Java ME。

没有JDK的话，无法编译Java程序，如果想只运行Java程序，要确保已安装相应的JRE。

## 3 Jre

**JRE（Java Runtime Environment）**是可以在其上运行、测试和传输应用程序的Java平台。它包括**Java**[**虚拟机**](https://baike.baidu.com/item/%E8%99%9A%E6%8B%9F%E6%9C%BA)（**jvm**）、**Java核心类库**和**支持文件**。

**Jre的三项主要功能**：

1. 加载代码：由类加载器完成
2. 校验代码：由字节码校验器完成
3. 执行代码：由运行时解释器完成

使用JDK开发Java程序，交给jre去运行

JDK向下兼容

**Java升级三方面**：提高效率、简化书写、提高安全性

## 4 DOS常用命令

|  |  |
| --- | --- |
| 1.切换盘符 | D: |
| 2.列出当前目录及文件夹 | dir |
| 3.创建目录 | md 文件名 |
| 4.删除目录 | rd |
| 5.删除一个文件 | del 文件名（不进回收站） |
|  | deltree(删除文件夹所有东西) |
| 6.进入指定目录 | cd |
|  | cd..(退回到上一级目录) |
|  | cd/(退回到根目录) |
| 7.清屏 | cls |
| 8.调出上一条/下一条 | 键盘上下键 |

## 5 JDK目录结构

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | Bin目录 | 存放Java的编译器、解释器（可执行文件） | | Demo目录 | 存放演示程序 | | db:database | 纯Java开发的数据库，是一个开源的关系数据库 | | include目录 | 存放用于本地方法的文件 | | jre | Java运行环境文件 | | lib | Java类库 | | src.zip | JDK提供类的源代码 | |

## 6 hello world

**控制台运行java文件**

1. 切换到文件目录 cd c:/program filles/java/homework
2. 编译home8.java文件 Javac home8.java
3. 运行home8.class文件 java home8

## 7 编码风格

1. K&R风格

C语言最早的缩进风格,由c的发明者Ritchie和他的合作伙伴率先使用

Class hello{

Public static void main(String[] s){

}

}

2. BSD风格

又称allman style 源自于Unix BSD程序员Eric Allman

Class hello

{

Public static void main(String[] s)

{

}

}

## 8 笔记&Tips

1. **Class关键字 代表 类 一个源文件中至多只能有一个被public修饰的类**
2. **一个源文件中有N个类时，编译就会生成N个.class文件 执行java xxx的时候，执行的是类名**
3. Java语言中 单词拼写 严格区分大小写
4. Main方法是Java应用程序的入口，有固定的书写格式
5. 小数默认double类型，整数默认int类型。

# 二、Java基础晋级

## 1 java的注释

被注释的内容 不运行 只是用来给程序员阅读的

1. **单行注释**

//被注释的内容

1. **多行注释**

/\*被注释的内容\*/

1. **文档注释**

/\*\* \*/ 执行 javadoc 类名.java 可以生成程序使用说明书 index.html

## 2 标识符&命名规则

1. java对 包/类/方法/参数和变量等要命名时使用的字符序列称为**标识符**
2. java标识符**命名规则**:
   1. 由字母(英,日,俄等),数字,下划线和美元符号组成
   2. 不能以数字开头
   3. 区分大小写
   4. 长度无限制(一般不要超过15个字符)
   5. 不能是java的保留字和关键字
3. **命名习惯**:见名之意 **驼峰命名规则**
   1. **包名**(com.qf.xxx):最好是域名倒过来 要求 所有字母小写
   2. **类或者接口名**(UserHome)

如果是一个单词 首字母大写

如果是多个单词 每个单词首字母大写

* 1. **方法或者变量名**(userHome)

如果是一个单词 首字母小写

如果是多个单词 首单词小写，以后的单词首字母大写

* 1. **常量名**(USERHOME)

如果是一个单词 所有字母都大写

如果是多个单词 所有字母都大写 用下划线分割每个单词

* 1. **TIPS**

java/ user#name/ hello world

强烈建议 大家不要用java的类名来命名自己的类

Date

String Class1 原则上可以 但是不推荐

## 3 java关键字

java中有一些被赋予了特定含义的,用专门的用途 这些字符串称为 **关键字**(keyword) 全部都是小写

const goto 保留 不要用!

## 4 常量

常量：java中 固定不变的一些数据 固定不变的量

java中的常量分类:

|  |  |
| --- | --- |
| 1.整数常量 | 所有整数 |
| 2.小数常量 | 所有小数 |
| 3.布尔型常量 | true /false |
| 4.字符常量 | 将一个数字 字母 或 符号用单引号 括起来 'a' |
| 5.字符串常量 | 将一个或者多个字符 用双引号括起来 ""空内容的字符串 |
| 6.null常量 | 只有一个数值 就是null |

对于整数常量 java有四种表现形式:

|  |  |
| --- | --- |
| 二进制 | 0 /1 |
| 十进制 | 0-9 |
| 八进制 | 0-7 **用0开头** |
| 十六进制 | 0 - 9 A-F **用0x或0X开头** |

## 5 进制转换

1. 进制的由来

进制: 就是进位

1. 二进制

**位(bit)** 标识一个二进制数 0 或者 1 是计算机存储的基本单位

**字节(byte)** 一个字节由8位组成,它表示作为一个完成处理单位的8个二进制码

1. ASCII 美国标准信息交换码表

256个

前 128个为常用的字符 如 运算符 字母 数字等

后 128个位特殊字符 键盘上找不到的字符

**windows默认编码是 GBK** dos -->chcp -->活动代码页:936

**mac 默认文字编码是utf-8** 在windows下 解压mac打的包 就相当于用gbk来解析utf-8就出现乱码

1. 进制转换

十进制 转 二进制

原理: 对十进制数进行 除 2 的运算 取余数

6 --> 110

二进制 转 十进制

原理: 二进制 乘以 2 的n次幂 的过程

110 ->

0\*2(0) + 1\*2(1) + 1 \* 2(2)

0 + 2 + 4

=6

2(0) -->1

2(1) -->2

2(2) -->4

0 0 1 1 0 1 0 1

128 64 32 16 8 4 2 1 = 53

当 8个bit 都是 1 最大是 255 那么一个字节最大就是 255

ip地址 一个段最大就是 255

**负数的十进制转二进制**

其实就是 十进制数 转换成 **二进制数后 取反+1**

**负数的二进制转十进制**

其实就是 二进制数 转换成 **-1 取反** 后转化为十进制

1. 二进制转八进制、十六进制
   1. 二转八 三位二进制数 转化为以为 一位八进制数
   2. 二转十六 四位二进制数 转化为以为 一位十六进制数

## 6 变量

1. 变量:将不确定的数据进行存储 在内存空间中开辟一块空间 需要给一个类型 和名字

java 强类型语言 对类型是有严格的划分的 在内存中分配的空间也大小不同

js :var; alert

1. **变量的分类**:
   1. 按照所属的数据类型划分:
      1. **基本数据类型变量**

数值型

整数类型 byte short int long

浮点类型 float double

字符型 char

布尔型 boolean

整数 默认 是 int 类型 ; 小数 默认是 double

* + 1. **引用数据类型变量**

类(class)

接口(interface)

数组(Array [])

枚举

* 1. 按照声明的位置划分:
     1. **全局变量**:方法外部 类的内部定义的变量
     2. **局部变量**:方法内部 或者 语句块内部定义

1. 为什么要定义变量?

用来不断的存放同一类型的常量,并且可以重复使用

定义变量的格式:

**数据类型 变量名 = 初始化值;**

记住格式 以不变应万变

1. Tips

变量的作用域(范围) 是在一对{}之间有效

一定要给初始化值

## 7 数据类型

1. 数值型
   1. **整数类型 byte、short、 int、 long（L/l）**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 数据类型 | 字节数 | 位 | 取值范围 |  |  |
| byte | 1 | 8 | -128 | ~ | 127 |
| short | 2 | 16 | -2(15) | ~ | 2(15)-1 |
| int | 4 | 32 | -2(31) | ~ | 2(31)-1 |
| long | 8 | 64 | -2(63) | ~ | 2(63)-1 |

* 1. **浮点类型 float(F/f) double**

|  |  |  |  |
| --- | --- | --- | --- |
| 数据类型 | 字节数 | 位 |  |
| float | 4 | 32 |  |
| double | 8 | 64 |  |
| char | 1 | 8 | 汉字 |
| Boolean | 2 | 16 | True/false |

double型比float型存储范围更大,精度更高.所以通常的浮点型数据不声明是默认是double

浮点型数据在底层 使用近似值来表示 不适用于在银行 电信领域存储金额

**BigDecimal --> 小数点后精确到的位数可以任意定制**

字符型 char

有2个字节组成 16位 2的16次方 是 65535 表示范围 0-65535

一个汉字是两个字节

单个字符: 汉字,英文字母,特殊符号

不允许 什么都没有''

ASCII

a~z 'a' -->97

A~z 'A' -->65

0~9 '0' -->48

系统编码

|  |  |
| --- | --- |
| gb2312 | 简体中文 |
| gbk | 简体中文 繁体中文 |
| gb18030 | 简体 繁体 生僻字 |
| iso-8859-1 | 西文字符集 |

java的数据类型之间转换

|  |  |  |
| --- | --- | --- |
|  | 原来类型 | 转化类型 |
| 自动(隐式)类型转换 | Byte、short、char | Int |
|  | Long | long |
|  | float | float |
|  | Double（默认） | Double |
|  | Int（默认） | Int |
| 强制(显式)类型转换 | 字节小类型 | 字节大类型 |

一旦 byte short char类型的变量参与运算,则运算结果表达式类型为 int型，首先会自动提升 然后再参与运算。

**表达式**就是 由变量,常量,运算符组成的式子

## 8 java的运算符

|  |  |  |
| --- | --- | --- |
|  | 符号 | Tips |
| 算术运算符 | +, - ,\*, /, %, ++, -- | 除数不能为零；  i++(先赋值，后计算)，++i（先计算，后赋值） |
| 赋值运算符 | = , += ,-=,\*=,/=,%= | S=S+1;S+=1;与前者相比，由于后者是**位操作**，效率也较前者高。 |
| 关系运算符 | > ,< ,>=,<=,==,!= | True/false |
| 逻辑运算符 | ! , & , |,^,&&,|| | ^异或（不同时为true）; &&、|| 短路运算符 |
| 位运算符 | &,|,^,~,>>,<<,>>> | &,|,^按位与、或、异或  << 乘以2的n次方，右侧补零 |
| 字符串连接符号 | + |  |
| 三元运算符/三目运算符 |  | A=B?C:D;(C、D必须有结果) |

**两个变量的值的三种交换形式**

1. 使用第三方变量

Temp=x;

X=y;

Y=temp;

1. 两个数相加

Int a=10,b=20;

a=a+b;

b=a-b;

a=a-b;

1. 一个数异或（^）同一个数两次还是他自己

Int a=10,b=20;

a=a^b;

b=a^b;

a=a^b;

## 9 运算符的优先级和转义字符

赋值运算符 从 右向左看

转义字符： 用在字符串中，用来定制字符串输出格式的特殊字符

\t 一个退格键

\n 换行（Linux、Unix）

\r 回车 **光标回行首**（window下面 换行 /r/t）

# 三、程序流程控制

1. **结构化程序设计**的的三种基本结构：顺序结构、选择结构、循环结构。
2. **顺序结构**是按照程序书写的顺序，依次执行各语句，直到程序结束=的结构。
3. **选择结构**是根据判断条件是否成立从而选择执行的结构中的执行体。
4. **循环结构**是重复执行一个或几个模块，直到满足某一条件或者break提前结束循环体为止。

## 1分支语句

为什么要使用分支？

当程序面临选择的时候，每一种选择对应一种操作（结果）

1. if语句

**结构**：

if（表达式） 》》 **关系表达式 和 逻辑表达式**

{

执行语句；

}

else if(表达式)

{

执行语句；

}

else 》》 **可以省略，可以对范围内的错误进行提示**

{

执行语句；

}

1. switch语句

switch(表达式)

{

case 常量 : 语句体;break;

case 常量 : 语句体;break;

default:

D;

Break;

}

Java1.6（之前） 类型byte\short\int\char 枚举

Java1.7（之后） 增加一个 String 类型

Tips：

1. case后面的只能是常量，不能是变量，多个case后面不能出现相同的值；
2. break可以省略吗？
   1. 最后一个可以省略；
   2. 小心case穿透；
3. Switch default：可以写在case前面，但是执行时先执行case；

## Switch和if的区别

|  |  |  |
| --- | --- | --- |
|  | Switch | If |
| 适用场景 | 区间范围 | 判断单个定值 |
| 表达式类型 | Byte、short、int、char、枚举、String | Boolean |

## 2 循环结构

**阶乘**：零的阶乘是1，所有小于和等于该数的正整数的乘积

**重复的行为 反复出现**

**2.1 While**

初始化语句

While（判断条件语句）

{

循环体语句；

控制条件语句，防止死循环

}

**2.2 do while**

初始化语句；

Do

{

循环体语句；

控制条件语句；

}while（判断条件语句）

Do while 和while的区别

当第一次执行的时候，若判断语句伟false，while不会执行，do while 循环至少会出现一次

* 1. **For**

For（初始化语句；判断条件语句；控制条件语句）

{

执行体；

}

## 流程控制语句

Break

适用场景：只能在switch和循环中

在多重循环体中，最内层的循环break结束，跳出最外层的循环可以借助***标记***来完成

Continue

适用场景：循环体内

碰到continue结束本次循环

Return

用来结束方法，碰到return，方法就结束

## 3 方法

什么是方法？

方法就是一段完成特定功能的代码块

为什么要有方法？

提高程序的复用性和可读性

方法和函数是一个意思

**方法的格式**

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

{

方法体语句；

Return返回值；

}

**方法的格式说明**

|  |  |
| --- | --- |
| 修饰符 | 目前就用public static后面我们再详细分解 |
| 返回值类型 | 就是功能结果的数据类型 |
| 方法名 | 符合命名规则即可，驼峰命名 |
| 参数列表 | 就是方法定义上的 用于接受实际参数的 |
| 参数类型 | 就是参数的数据类型 |
| 参数名 | 就是变量名 |
| 方法体语句 | 就是你这个方法要完成的功能 |
| Return | 结束方法的返回值 |
| 返回值 | 就是这个方法的功能的结果，由return带给调用者 |

## 方法的重载

1. **概念**：在同一个了类中，方法名相同，参数列表不同，与返回值无关。
2. **重载的好处**：设计重载的目的是减少方法名的个数，便与程序员掌握，程序在运行的过程中，通过传递不同的参数，动态去调用与之最匹配方法。
3. **最佳匹配原则 和小匹配大原则**：

当初按时餐的时候，如果多个重载的方法都能够接受该实参，那么他回去找本身的数据类型；如果没有，向上转型的最近的那个。

## 递归

1. 阶乘
2. 递归：递归的出口；在符合条件的情况下，简化同类问题，调用自己；

## 数组

1. 数组：就是用来存储同一种数据类型，定长的容器
2. 类型：数组是引用数据类型
3. 数组的好处：可以存储多个同类型数据；可以给这些数据从零开始编号；
4. 语法格式
   1. 动态初始化
      1. **数据元素类型[ ] 数组名=new 数组元素类型[数组中元素的个数] ；**
   2. 静态初始化

**数组元素类型[] 数组名=new 数组元素类型[]{元素1，元素2};**

* 1. 内存分析

java程序在运行时,需要在内存中分配空间,为了提高运算效率,又对空间进行了不同区域的划分,

因为每一篇区域都有特定的处理数据方式和内存管理方式

**栈内存**

用于存储局部变量 (方法里的,形式参数列表,定义在for循环) 当数据使用完 自动释放

**堆内存**

用于存储数组和对象 通过new建立的实例都放在堆内存中

每一个实体都有一个内存地址值

实体中的变量都有默认初始化值

byte,short,int --->0

long---> 0L

float ---> 0.0F

double --->0.00

boolean --->false

char ---> \u0000 --->表示空

引用数据类型 --->null

垃圾回收机制 当实体不在被使用,会在不确定的时间呗垃圾回收器回收

小知识: c++ 需要手动回收 java是自动回收

方法区 本地方法区 寄存器

### 遍历数组的两种方式

1.普通for循环

for(int i=0;i<x.length;i++){ //i不只是循环条件 还是数组的角标 从0开始

System.out.print(x[i]+" ");

}

2.增强for循环[ForEach循环]

语法:

for(数组中元素的类型 变量 : 数组名){

数组中的元素类型 临时变量 = 变量;

}

区别:

普通for 循环变量 代表角标 在循环体中 可以 获取角标 /元素 可以进行复杂的操作

增强for 变量 代表的是 数组中的元素 不存在角标的意思

### 数组角标的查找:

1.顺序查找

2.二分查找/折半查找

前提:必须是有序数列 升序/降序都可以

通过将待查找的元素与中间索引值对应的元素进行比较,

若小于中间元素,去左边 end = middle-1

若大于中间元素,去右边 start = middle+1

继续折半 直到当arr[middle] == 元素 就找到了 否则返回一个负数

## 4 排序

### 选择排序

依次和最左边比较

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

for(int j=i+1;j<s.length;j++)

if(s[i]>s[j])

swap(s[i],s[j]);

### 冒泡排序

小气泡上浮，大气泡下沉

for(int i=0;i<s.length-1;i++)

{

//设置标志，当某一次排序完成以后，没有需要交换的值，则代表排序完成。

Flag=0;

for(int j=0;j<s.length-1-i;j++)

if(s[j]>s[j+1])

swap(s[j,s[j+1]]);

flag=1

if(flag==0)

break;

}

### 快速排序

* + 1. 1）设置两个变量i、j，[排序](http://baike.baidu.com/item/%E6%8E%92%E5%BA%8F)开始的时候：i=0，j=N-1；
    2. 2）以第一个数组元素作为关键数据，赋值给**key**，即**key**=A[0]；
    3. 3）从j开始向前搜索，即由后开始向前搜索(j--)，找到第一个小于**key**的值A[j]，将A[j]和A[i]互换；
    4. 4）从i开始向后搜索，即由前开始向后搜索(i++)，找到第一个大于**key**的A[i]，将A[i]和A[j]互换；
    5. 5）重复第3、4步，直到i=j； (3,4步中，没找到符合条件的值，即3中A[j]不小于**key**,4中A[i]不大于**key**的时候改变j、i的值，使得j=j-1，i=i+1，直至找到为止。找到符合条件的值，进行交换的时候i， j指针位置不变。另外，i==j这一过程一定正好是i+或j-完成的时候，此时令循环结束）。

## 5 可变长度参数

* 1. 可以理解成一个 可以变长的数组；
  2. 什么是可变参数？
     1. 在设计方法时，方法的最后一个形参的个数是变动的
     2. Jdk1.5增加的新特性
  3. 格式：

访问权限修饰符 返回值类型 方法名（类型1 参数1，类型2…参数2）{

}

* 1. 特点：
     1. 可变参数 可以理解成一个一维数组
     2. 可变参数 只能写在方法形参列表的后面
     3. 这…只能有三个 介于参数类型和参数之间

public static void add2(double d,int...args){ //编译时期错误 是语法错误

System.out.println(d);

int sum = 0;

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

sum +=args[i];

}

System.out.println(sum);

}

# 四、面向对象

## Arrays类

* 1. Arrays.toString(arr);
  2. Arrays.sort(arr);
  3. Arrays.binarySearch(arr,n);
  4. 复制指定的数组
     1. Arrays.copyOfRange (arr,from,to);
     2. Arrays.copyOf (arr,newlength);
  5. Arrays.equals(arr1,arr2);

## 二维数组

* 1. 格式
     1. 数据类型 数组名[][]=new 数据类型[m][n];
     2. 数据类型[][] 数组名=new 数据类型[m][n];
     3. 数据类型[] 数组名 []=new 数据类型[m][n];
     4. 数据类型 数组名[][]={{}，{}，{}}
     5. Int[] x,y[];//x一维数组,y二维数组
  2. Java中“传值”和“传引用”的问题
     1. （传值）形参不会改变原来的值；
     2. （传引用）引用数据类型作为参数传入方法中时，改变时会改变堆内存中的值；

## 面向对象

* 1. 三大特征：封装、继承、多态
     1. 封装：类和方法定义本身，就是封装性的体现。
     2. 继承：
     3. 多态：
  2. 什么是对象？

万物皆对象

* 1. 类和对象之间的关系？对象》》类
  2. 描述现实事物》》行为（名词）、属性（动词）
  3. **Java中最基本的单位是类**，在类中描述事物
     1. 成员变量（行为）
     2. 成员方法(属性)

1. 成员变量和局部变量的区别

|  |  |  |
| --- | --- | --- |
|  | 成员变量 | 局部变量 |
| 类中位置 | 类中、方法外 | 方法内 |
| 内存位置 | 堆内存 | 栈内存 |
| 生命周期 | 对象调用创建，调用完成销毁 | 方法调用创建，创建完成销毁 |
| 初始化值 | 默认初始值 | 定义、赋值 |

## 继承

继承可以使得子类具有父类的属性和方法或者重新定义、追加属性和方法等。

继承中的构造方法的关系

而且子类中所有的构造函数默认第一条都是super();

为什么?

因为子类会继承父类中的数据,可能还会使用父类的数据

所以,子类初始化之前,一定要先完成对父类数据的初始化

当父类中没有无参构造 怎么办?

1.在子类的构造函数中 调用父类的有参构造 super(x,x)解决

2.在子类的构造函数中 调用子类其他的构造函数 从其他的构造函数在调父类 this() 解决

**this() super()都要在构造函数第一行 那么只能用一个 只能二选一**

this和super区别

|  |  |  |
| --- | --- | --- |
| Differences | This (当前对象引用) | Super (父类对象引用) |
| 调用成员变量 | 调用本类或父类 | 调用父类 |
| 调用成员方法 | 调用本类或父类 | 调用父类 |
| 调用构造方法 | **调用本类** | **调用父类** |

**二.final**

最终的

final 可以修饰 类 方法 变量

final 修饰的类不可以被继承

final 修饰的方法不可以被覆盖

final 修饰的变量 是一个常量 只能被赋值一次 命名规则 单词全部大写 多个单词之间用下划线分割

三.权限修饰符

包 package

为什么有包?

**将字节码(.class)文件 进行分类存放**

包其实就是文件夹

包

1.按照功能分

删除

修改

查询

增加

2.按照模块分

定义包的格式

package 包名;

多级包用.分开即可

将字节码文件生成到 类中package指定的目录中

**javac -d . Demo\_1.java**

注意:

1.package必须写在程序的第一条可执行代码

2.package语句在一个java文件中只能有一个

3.如果没有package 默认表示无包名

不同包下的类之间如何访问?

import 包名

import java.util.Scanner; -->这种方式导入是到类的名称 具体到哪个类

import java.util.\*;

\* :通配符 util包下的所有类 都被导入

访问权限修饰符

四个

private 私有的

default(什么也不写) 默认的

protected 保护的

public 公有的

**访问权限修饰符 本类 同包(子类和无关类) 不同包下(子类) 不同包(无关类)**

private √

default √ √

protected √ √ √

public √ √ √ √

修饰符:

权限修饰符: private 默认 protected public

状态修饰符: static final

抽象修饰符: abstract

类:

权限修饰符: 默认 public

状态修饰符: final

抽象修饰符: abstract

成员:

权限修饰符: private 默认 protected public

状态修饰符: static final

用的最多的是 private

构造方法:

权限修饰符: private 默认 protected public

用的最多的是 public

## 抽象

一 概念

看不清

被abstract修饰的类 叫抽象类

被abstract修饰的方法 叫抽象方法

**抽象类的特点:**

1.抽象方法一定在抽象类中

2.抽象方法和抽象类都必须用 abstract关键字修饰

3.抽象类不可以用new创建对象 **抽象类无法实例化** 因为调用抽象方法没意义

4.抽象类中的方法想要被使用 必须由子类重写其所有抽象方法 建立子类对象调用

如果只覆盖部分 会继承父类的抽象方法 那么子类也是个抽象类

5.一个抽象类中可以包含抽象方法和非抽象方法 但一个方法被抽象修饰,它所在的类必须是抽象的

抽象类可以不定义抽象方法

**抽象的好处?**

抽象父类 让 子类 必须完成父类的全部抽象方法

功能不确定 子类去做

**修饰的类无法创建对象**

abstract只是一个修饰符 如果定义的抽象类中没有抽象方法 那么这么定义只是为了不让该类创建对象

二 语法格式

抽象类

访问权限修饰符 abstract class 类名{

//普通属性

//普通方法

//抽象方法

}

抽象方法

访问权限修饰符 abstract 返回值类型 方法名(形参列表);

**普通子类继承抽象类时，抽象类方法或者接口方法必须重写。**

模板方法设计模式

当代码完成优化后,就可以解决这类问题

什么是模板方法呢?

在定义功能时,功能的一部分是确定的,但是有一部分是不确定的.而确定的部分在使用不确定的部分

那么这时就将不确定的部分暴露出去,由该类的子类去完成

不要记代码 记思想

面向对象三大特征之一

## 多态

定义:某一类事物的多种存在形态

例如:动物中 猫 狗

猫 c = new 猫();

动物 d = new 猫();

**多态的前提**

1.父类引用 指向 子类对象

2.必须是类与类之间有关系 要么继承 要么实现

3.**子类存在重写**

**多态的好处**

提高了程序的扩展性 作用体现在可以当做形参 可以接收任意子类的对象

**多态的弊端**

但是只能使用父类的引用访问父类中的成员

不能使用子类的特有属性和方法 如果想要使用 必须向下转型

对象的向上转型和向下转型

// byte b = 2; int x = b;

//基本数据类型自动提升 体现在了引用数据类型中 向上转型

//强转 向下转型 父类型转成子类型

/\*

问:可以这么做么?

Animal a = new Animal();

Cat c = (Cat)a;

错误 不能强行将一个动物转成猫

不能将父类对象 转成子类对象

多态转的是父类引用 指向了自己的子类对象时 该应用可以被提升 也可以被强转

多态至始至终 都是子类对象在做着变化

\*/

instanceof

java中instanceof运算符是用来运行时指出对象是否是特定类的一个实例

通过放回一个boolean类型的值来指出

对象 instanceof 类型

结果:

true: 对象是属于这个类型的实例

false: 对象不是属于这个类型的实例

//instanceof 不常用

// 1.当子类有限时

// 2.当传输的类型需要进行其他操作的时候 比如比较

**多态的注意事项**

1.多态下 对成员方法的调用 编译看左边(父类) 运行看右边(子类)

2.多态下 对成员变量的调用 编译看左边(父类) 运行看左边(父类)

3.多态下 静态成员 编译看左边(父类) 运行看左边(父类)

## 接口

什么是接口:前期理解 可以理解为是一个特殊的抽象类

当抽象类中的方法都是抽象的,那么该类可以通过接口的形式来定义

class:用于定义类

interface:用于定义接口

1,语法:

访问权限修饰符 interface 接口名{

//全局常量

//抽象方法

}

接口中的成员修饰符是固定的

**成员变量: public static final**

**成员函数: public abstract**

如果不写 默认会给加上 只要是定义为接口 interface 那么成员的修饰符就是固定的 建议写上

implements: 实现

类与接口的关系 实现

继承:extends

**2.特点:**

1.接口是一个特殊的抽象类 特殊在"接口中所有的属性都是常量,方法都是抽象的"

2.接口允许多实现 弥补了java在类之间单继承的不足 因为接口中的抽象类没有方法体 实现谁都行 不冲突

3.java中允许单继承 的同时 实现多个接口

4.接口与接口之间 也有关系 接口继承接口 一个接口可以继承多个接口 因为都是抽象的

剪短版:

1.接口可以用来多实现

2.类与接口之间是实现关系 而且类继承一个类的同时 可以实现多个接口

3.接口与接口之间可以有多继承关系

3.**接口的好处**

它是对外暴露的规则 提高了程序的扩展性

封装 保证了程序的安全性

4.**接口的本质**

从接口设计的角度而言 接口中定义了一套规则和标准

## Object类

java.lang.Object

类 Object 是类层次结构的根类。每个类都使用 Object 作为超类。所有对象（包括数组）都实现这个类的方法。

boolean equals(Object obj)  **//重写equals和hashcode方法用于set集合判重**

java认为所有的对象都具备可比性 都能比较是否相同 只要是实体 就能比

**== 和 equals的区别**

== 是一个比较运算符 可以比较基本数据类型 也可以比较引用数据类型 但是引用数据类型比较的是地址值

equals是一个方法,只能比较引用数据类型,所有的对象都具备equals方法 因为他们都继承自Object类

**如果没有重写Object类中的equals 那么比较的是地址值**

重写之后 比较对象中的属性

**equals和hashCode间的关系:**

1.如果两个对象相同(即 用equals比较返回true),那么他们的hashCode 值一定相同

2.如果两个对象的hashCode相同,他们并不一定相同(即 用equals比较返回true) -->集合要讲

先判断 hashCode是否相同 如果不同 那就比没有必要再用equals去比较 减少了equals的使用次数

String toString()

java认为所有对象不仅具备比较性 还能以字符串的形式打印出来

返回该对象的字符串表示

## 内部类

将一个类定义在另一个类里面,那么里面的那个类就叫**内部类(内置类 嵌套类)**

1.**成员内部类**

1.内部类可以直接访问外部类中的成员,包括私有成员

是因为内部类中持有了一个外部类的引用

外部类名.this.

2.外部类中可以直接创建私有的内部类对象 调用成员

内部类非私有 外部其他类要访问内部了中的成员 必须要建立内部类对象

外部类名.内部类名 对象名 = 外部类对象.内部类对象

3.当类在成员位置时(是内部类的时候) 就可以被成员修饰符修饰

比如: private

static

2.**静态内部类**

被static 修饰的内部类

3.**局部内部类**

定义在成员方法中的类

1.不可以被成员修饰符修饰

2.可以直接访问外部类中的成员,因为还持有外部类中的引用

3.不可以访问它所在的局部中的变量,只能访问被final修饰的局部常量

jdk 1.8 新特性 可以不用写final 关键字 但是 局部变量本身还是常量 不能被改变

为什么?

方法 进栈内存 用完了 就弹栈了 这个局部变量也消失了

但是 局部内部类对象 还没消失 还想用这个变量 就没有了

被final 修饰的变量 会存储在常量池中 就算方法弹栈 常量也依然存在 可以继续使用

栈内存 先进后出 压栈 和 弹栈的过程

4.**匿名内部类**

匿名内部类 其实 就是内部类的简写格式

//内部类 写成 匿名内部类 必须有前提条件

/\* 内部类 必须是继承一个类 或者实现接口

\* 格式:

\* new 类名或者接口名(){

\* 重写方法

\* }.方法名();

\* 匿名对象 对方法的调用 只能调用一次 想要调用其他的必须重新建立匿名对象来调用

\* 匿名内部类中定义的方法 最好不要超过3个

\* \*/

## 基本数据类型包装类

将基本数据类型 包装成引用数据类型

因为 引用数据类型 可以调很多方法

**JVM中当integer的值落在-128~127（byte）,上时，如果有该值，将不再分配内存空间。**

**Collection集合中基本数据类型包装类会自动排序。都继承于 Number**

|  |  |  |  |
| --- | --- | --- | --- |
| 基本数据类型 | 默认值 | 包装类型 | 包装类型的默认值 |
| byte | 0 | Byte | Null |
| short | 0 | Short | Null |
| int | 0 | Integer | Null |
| long | 0 | Long | Null |
| float | 0.0 | Float | Null |
| double | 0.0 | Double | Null |
| char | ‘ ’/’\u0000 ’ | Character | Null |
| boolean | false | Boolean | Null |

**装箱 和 拆箱**

1装箱

语法: jdk>=1.5 自动装箱

包装类型 对象 = 基本数据类型;

jdk<1.5 手动装箱

包装类型 对象 = new 包装类型(基本数据类型);

2.拆箱

语法: jdk>=1.5 自动拆箱

基本数据类型 变量 = 包装类型的对象

jdk<1.5 手动拆箱

基本数据类型 变量 = 包装类型的对象.xxxValue();

所有基本数据类型的包装类 equals 比的都是值

**基本数据类型 转成 字符串**

基本数据类型 + ""

基本数据类型包装类.toString(基本数据类型)

Integer.toString(10); //"10"

**字符串转成 基本数据类型**

static int parseInt(String s);

**int a = Integer.parseInt(str);**

**int b = Integer.valueOf(str).intValue();**

# 五、String

所有类中使用最频繁的类

字符串最大的特点:一旦被创建 就不能被改变 字符串常量

String s = "abc"; s是一个类类型变量 "abc" 是一个对象

常见的操作都有哪些?

"abcd"

**1.获取**

1.1 获取字符串的长度

int length()

1.2 根据位置 获取位置上某个字符

char charAt(int index);

1.3 根据字符获取该字符在字符串中的位置

int indexOf(int ch);

int indexOf(int ch, int fromIndex) 从fromIndex指定位置开始 获取ch在字符串中的位置

int indexOf(String str)

**2.判断**

2.1 字符串中是否包含某一个子串

boolean contains(str)

2.2 字符串中是否有内容

boolean isEmpty()

2.3 字符串是否以指定内容开头

boolean startsWith(str);

2.4 字符串是否以指定内容结尾

boolean endWith(str);

2.5 比较两个字符串是否相同

boolean equals(str);

2.6 比较两个字符串是否相同 (忽略大小写)

boolean equalsIgnoreCase(String anotherString)

**3.转换**

3.1 将字符数组转换成字符串

构造函数 String(char[])

String(char[],int offset,int count)

静态方法

static String copyValueOf(char[])

3.2 将字符串转成字符数组

char[] toCharArray();

3.3 将字节数组转成字符串 (io 流再讲)

3.4 将字符串转成字节数组 (io 流再讲)

3.5 基本数据类型 转成 字符串

String valueOf();

**4.替换**

4.1 替换字符中的旧字符

String replace(char oldChar, char newChar)

**5.切割**

String[] split(String regex)

**6.截取**

String substring(int beginIndex)

**7.转换/去空格/对两个字符串 进行自然顺序的比较**

7.1 小写转大写

toUpperCase()

7.2 大写转小写

toLowerCase()

7.3 去除两端空格

trim()

7.4 对两个字符串 进行自然顺序的比较

int compareTo(String anotherString)

## 正则表达式

|  |  |
| --- | --- |
| **字符类** | |
| [abc] | a、b 或 c（简单类） |
| [^abc] | 任何字符，除了 a、b 或 c（否定） |
| [a-zA-Z] | a 到 z 或 A 到 Z，两头的字母包括在内（范围） |
| [a-d[m-p]] | a 到 d 或 m 到 p：[a-dm-p]（并集） |
| [a-z&&[def]] | d、e 或 f（交集） |
| [a-z&&[^bc]] | a 到 z，除了 b 和 c：[ad-z]（减去） |
| [a-z&&[^m-p]] | a 到 z，而非 m 到 p：[a-lq-z]（减去） |
|  |  |
| **预定义字符类** | |
| . | 任何字符（与[行结束符](mk:@MSITStore:E:\TED\JDK_API_1_6_zh_CN.CHM::/java/util/regex/Pattern.html#lt)可能匹配也可能不匹配） |
| \d | 数字：[0-9] |
| \D | 非数字： [^0-9] |
| \s | 空白字符：[ \t\n\x0B\f\r] |
| \S | 非空白字符：[^\s] |
| \w | 单词字符：[a-zA-Z\_0-9] |
| \W | 非单词字符：[^\w] |
| **边界匹配器** | |
| ^ | 行的开头 |
| $ | 行的结尾 |
| \b | 单词边界 |
| \B | 非单词边界 |
| \A | 输入的开头 |
| \G | 上一个匹配的结尾 |
| \Z | 输入的结尾，仅用于最后的[结束符](mk:@MSITStore:E:\TED\JDK_API_1_6_zh_CN.CHM::/java/util/regex/Pattern.html#lt)（如果有的话） |
| \z | 输入的结尾 |
|  |  |
| **Greedy 数量词** | |
| *X*? | *X*，一次或一次也没有 |
| *X*\* | *X*，零次或多次 |
| *X*+ | *X*，一次或多次 |
| *X*{*n*} | *X*，恰好 *n* 次 |
| *X*{*n*,} | *X*，至少 *n* 次 |
| *X*{*n*,*m*} | *X*，至少 *n* 次，但是不超过 *m* 次 |

符合一定规则的表达式

作用:用于专门操作字符串 是一个特殊的字符串 用于验证其他字符串是否合法

1.字符类

[abc] a,b或者c 其中的任意一个 但不能是全部

2.预定义字符

\d 数字[0-9]

\D 非数字 [^0-9]

\s 空白字符

\S 非空白字符

\w 单词字符 [a-zA-Z 0-9]

\W 非单词字符

3.数量词

+:一次或多次

?:一次或0次

\*:0次或多次

{n}: 出现的次数 正好是n次

{n,}: 出现的次数 至少是n次

{n,m}: 出现的次数至少是n次 最多是m次 >n <=m

1.匹配 **boolean matches(String regex)**

告知此字符串是否匹配给定的正则表达式。

2.切割 **String[] split(String regex)**

根据给定正则表达式的匹配拆分此字符串。

3.替换 **String replaceAll(String regex, String replacement)**

使用给定的 replacement 替换此字符串所有匹配给定的正则表达式的子字符串。

4.**分组** (.)\\1(.)\\2

/\*替换 切割 匹配 获取

\* 1.如果只想知道该字符是对是错 使用匹配

\* 2.想要将已有的字符串变成另一个字符串 替换

\* 3.想要按照自定的方式将字符串变成多个字符串 切割 获取规则以外的子串

\* 4.想要拿到符合要求的字符串子串 获取 获取符合规则的子串

\* \*/

**好处:简化对字符串的复杂操作**

**弊端:符号定义越多,正则越长,阅读性越差**

二

## StringBuffer

JDK1.0

字符串组成的原理就是通过该类实现的

字符串缓冲区 是一个容器 可以增删改查

很多方法和String是一样的

StringBuffer 是可变长度的

C U R D

create

update

read

delete

**1.存储**

在结尾插入

StringBuffer append(int i) 将指定的数据作为参数添加到已有的数据的结尾处

没有byte 和 short

在指定位置插入

StringBuffer insert(int offset, char c)

**2.删除**

StringBuffer delete(int start, int end) 包含头 不包含尾 删除缓冲区中的数据

StringBuffer deleteCharAt(int index) 删除指定位置的字符

**3.获取**

char charAt(int index)

int indexOf(String str)

int lastIndexOf(String str)

int length()

String substring(int start, int end)

**4.修改**

StringBuffer replace(int start, int end, String str)

**5.反转**

StringBuffer reverse()

特点:

1.长度是可变的

2.可以直接操作多个数据类型

3.最终会通过toString() 方法变成字符串

**String , StringBuffer 区别**

同:

* + - 1. 都是操作字符串的

异:String效率低 String每拼接一次都会新创建一个字符串 大批量占用内存

StringBuffer的效率高 在原有字符串的基础上进行拼接,占用内存小

基本数据类型是值传递 不改变其值

引用数据类型是址传递,改变其值

String 类虽然是引用数据类型 但是他当做参数传递时和基本数据类型一样

基本数据类型包装类当做参数传递时和基本数据类型一样

StringBuilder JDK1.5

和StringBuffer一样 StringBuffer怎么调用 StringBuilder就怎么调

**StringBuffer 和 StringBuilder 的区别**

同:都是操作字符串

异:StringBuffer是线程同步的 线程安全的 效率低

(在多线程并发访问的前提下,访问当前方法 只有一个线程能进行访问 只有操作完毕,其他线程才能执行)

StringBuilder是线程不同步的 线程不安全的 效率高

(在多线程并发访问的前提下,访问当前方法,所有线程都能执行当前方法)

操作字符串的效率 StringBuilder > StringBuffer > String

**升级的三个因素:**

1.提高效率

2.简化书写

3.提高安全性

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

# 六、java中的常用类

## System类

System类包含一些有用的类字段和方法,它不能被实例化

常用的属性

System.in -->InputStream 标准输入流

System.out -->PrintStream 标准输出流

常用的方法

static void gc() //运行垃圾回收器 相当于呼叫保洁阿姨

//当垃圾回收器将要释放无用的对象内存时,先调用Object类中该对象的finalize()方法

static void exit(int status) //无条件终止jvm的执行

**static long currentTimeMillis() //以毫秒值返回的系统当前时间**

static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)

第一个:要复制的数组

第二个:从要复制的数组的第几个开始

第三个:复制到哪

第四个:复制到的数组第几个开始

第五个:复制长度

Arrays工具类中的 复制数组

static int[] copyOf(int[] original, int newLength)

## Runtime类

每个java应用程序都有一个Runtime类实例 使应用程序能够与其运行环境相连接

可以通过 **getRuntime()方法获取到当前运行时**

应用程序不能创建自己的Runtime类实体

//获取java程序关联的运行时对象

Runtime rt = Runtime.getRuntime();

System.out.println("处理器数量:"+rt.availableProcessors());

System.out.println("空闲内存数:"+rt.freeMemory());

System.out.println("总内存数:"+rt.totalMemory());

System.out.println("可用最大内存数:"+rt.maxMemory());

## Math类

Math类包含用于执行基于数学运算的方法

平方根 三角函数

常用方法:

**pow 次方**

**random() 随机数 生成double 0.0>= x <1.0**

**伪随机数 底层就是算法 只要是算法 就有可能有规律**

**max()**

**min()**

**abs() 绝对值**

**sqrt() 开平方**

## Random类

util包下的

此类的实例是用于生成伪随机数流

int nextInt(int n) 返回一个伪随机数，它是取自此随机数生成器序列的、在 0（包括）和指定值（不包括）之间均匀分布的 int 值。

## Date类

util包下的 还有一个Date是sql包下的 所以注意别导错包了

Date类是一个时间的抽取与封装 精确到 毫秒

构造函数

Date()

Date(long date)

成员方法

long getTime()

void setTime(long time)

SimpleDateFormat 类 实现日期和字符串的相互转换 text包

构造函数:

SimpleDateFormat()

SimpleDateFormat(String pattern)

成员方法:

public final String format(Date date)

public Date parse(String source)

## Calender类

util 包 抽象类

Calendar rightNow = Calendar.getInstance();

getInstance();使用默认时区和语言环境获得一个日历

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

# 七、异常

1.异常的概念

异常就是java程序在运行的过程中出现的错误 不正常的情况

分类:

throwable

|--Error 错误 不可治愈的疾病

内存溢出 java.lang.OutOfMemoryError JVM

服务器宕机 数据库崩溃

|--Exception 异常 可治愈的疾病

|--受检异常 checked (编译时异常)

只有java语言提供了checked异常 java认为checked异常都是可以被处理的异常

所以java程序必须显示处理checked异常 如果程序没有处理checked异常 该程序在 编译 时

就会发生错误

ClassNotFoundException

NoSuchMethodException

io.IOException

|--运行时异常 RuntimeExcepiton

NullPointerException

ArrayIndexOutOfBoundsException

NumberFormatException new Integer("abc")

ClassCastException 强转

ArithmeticException

java为了阅读性好 把父类名称 作为子类的后缀名 能看出是谁的小弟

**2.处理异常**

JVM 默认是如何处理异常的?

main函数收到这个问题 有两种处理方式

a:自己将该问题处理 然后继续执行

b:自己没有针对的处理方式 只有交给调用main 的JVM处理

JVM 有一个默认的异常处理机制 就将该异常进行处理

并将该异常的名称,异常信息,异常出现的位置 打印在控制台上了

第一种:

try...catch...finally

第二种:

throws

第一种 格式

try{

需要被检测的代码

}catch(异常类 变量){

处理异常的代码(处理方式)

}finally{

一定会被执行的语句

}

**Throwable常见的方法**

a: String getMessage() 获取异常信息 返回字符串

b: String toString() 获取异常类名和异常信息 返回字符串

c: void printStackTrace() jvm默认的异常处理机制 就是调用这个方法

打印异常在堆栈的跟踪信息

try...catch处理异常的特殊情况

try{

}catch{

}catch{

}

**3.抛出异常**

throws

语法:

访问权限修饰符 返回值类型 方法名(参数列表) throws 异常1,异常2...{

方法体

}

建议在方法上预先设置可能出现的异常 那么谁调用我 谁处理我

**4.自定义异常**

throw

语法:

访问权限修饰符 返回值类型 方法名(形参列表){

方法体

throw 异常对象

}

因为在项目中会有出现特有的异常情况 这些问题java没有描述 那么就需要自定义

1.继承于异常

Exception 编译时异常 (必须处理)

RuntimeException 运行时异常(如果不发生意外 可以暂时不处理)

Exception中有一个特殊的子类异常 RuntimeException 运行时异常

如果在函数内部抛出该异常,函数上可以不用声明,编译一样通过

如果在函数上声明了该异常,调用者可以不用进行处理 编译一样通过

功能定义没有错 需要在运行的时候 解决某个可能发生的问题

**throws 和 throw 的区别**

throws

用在方法声明上 跟的是异常类名

可以跟多个异常类名,用逗号隔开

表示抛出异常,由该方法的调用者来处理

throw

用在方法内,跟的是异常对象名

只能抛出一个异常对象名

表示抛出异常 由方法内部的语句处理

finally

被finally控制的语句体一定会执行

特殊情况 在执行finally之前 jvm退出了 (System.exit(1));

作用: 用于释放资源 io 数据库

try{

}catch(){

}catch(){

}finally{

一定会被执行

}

异常格式 记住一点 catch 用于处理异常 如果没有catch就代表异常没有被处理过

try{

}finally{

}

面试题:

**1.final finally 和 finalize的区别**

final:可以修饰类 不能被继承

修饰方法,不能被重写

修饰变量,只能赋值一次

finally: 是 try语句中的一个语句体 不能单独使用 用来释放资源 一定会执行

finalize: 是Object类中的一个方法,当垃圾回收器确定不存在该对象的等多引用时,垃圾回收器调用此方法

**5.异常在继承中的特点**

1.子类重写父类方法时,子类的方法必须抛出相同的异常或者父类异常的子类异常(父亲坏了 儿子不能比父亲更坏)

2.如果父类抛出了多个异常,子类重写父类时,只能抛出相同的异常或者父类异常的子类异常

子类不能抛出父类没有的异常

3.如果被重写的方法没有异常抛出,那么子类的方法绝对不可以抛出异常 如果有异常 在方法内处理 try...catch

什么时候用 try..catch / throws

区别

后续程序需要继续运行 就 try

后续程序不需要继续运行 就 throws

finally中写返回值try和catch将无效

int x=10;

try{

x=20;

syso(1/0);

}catch(Exception e)

{

x=30;

return 30;

}finally

{

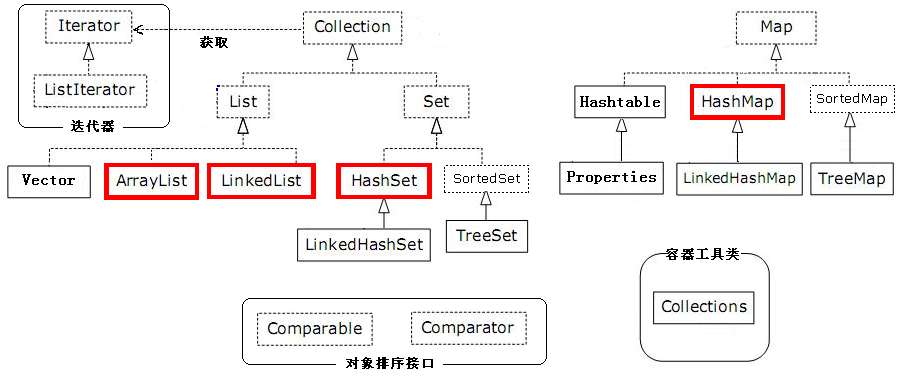
X=40;

Return x;

}

结果为40，不过finally中一般不能写return（犯罪行为）

# 八、集合



集合的由来?

数组长度是固定的,添加的元素类型也是固定的,操作起来很不方便

java给我们提供了一个集合类

能够存储任意对象,

长度是可变的 随着元素的增加而增加 随着元素的减少而减少

**数组和集合的区别**

区别1:

数组可以存储基本数据类型/引用数据类型

基本数据类型存的是值 引用数据类型存的是地址

数组在创建的时候 就会定义存储的数据类型 也就是只能存储一种数据类型

集合只能存储引用数据类型(对象)

集合中也可以存储基本数据类型(装箱)最终存储的还是 Object

如果没有泛型限定 默认存储的都是 Object类型的数据 也就是任意类型

区别2

数组长度是固定的,不能自动增长

集合是长度可变的,根据元素的多少来决定长度

## Collection

//Collection 是个接口 接口不能创建对象 需要找接口的实现类

Collection 对象名 = new 接口的实现类();

基本功能演示

boolean add(E e)

boolean remove(Object obj) 删除指定元素 删除第一个出现的元素

void clear() 清空集合

boolean contains(Object obj) 判断集合是否包含指定元素 底层调用equals 判断值是否相同

boolean isEmpty() 判断集合是否为空

int size() 获取元素的个数

带all方法

boolean addAll(Collection c);

boolean removeAll(Collection c); 移除此 collection 中那些也包含在指定 collection 中的所有元素（可选操作）。

boolean containsAll(Collection c); 如果此 collection 包含指定 collection 中的所有元素，则返回 true

boolean retainAll(Collection c); 保留两个列表的 交集

**迭代器 Iterator 接口**

Iterator<E> iterator() 返回在此 collection 的元素上进行迭代的迭代器

迭代器是对集合的遍历,就是把取出方式定义在集合的内部 这样取出方式就可以直接访问集合内部的元素

iterator()

Collection

|--List 元素是有序的(存取顺序一致) 元素可以重复 因为该集合体系有索引

|--Set 元素是无序的(存取顺序不一致) 元素不可以重复

## List

特有方法 凡是可以操作角标的方法都是该体系特有的方法

List集合 是以线性方式存储的

增加

void add(index,element)

boolean addAll(int index, Collection<? extends E> c)

删除

remove(index)

修改

set(index,element)

查询

get(index)

subList(from,to)

listIterator() list集合特有的迭代器

**List集合中的三个常用子类**

|--Vector

底层数据结构是数组 查询快 增删慢 线程安全 效率低

默认长度是10 超过就会100%延长 变成20 浪费空间

|--ArrayList

底层数据结构是数组 查询快 增删慢 线程不安全 效率高

默认长度是10 超过就会new一个新的数组 50%延长 节省空间

|--LinkedList

底层数据结构是链表(双向链表) 查询慢 增删快 线程不安全 效率高

到底用谁呢?

查询多 ArrayList

增删多 LinkedList

如果都多 ArrayList

### ArrayList

集合

特点:

1.底层是数组实现的

2.有序(存取顺序一致)

3.可重复

4.查询快 增删慢

### linkedList

集合

特点:

1.底层是双向链表

2.有序(存取顺序一致)

3.可重复

4.查询慢 增删快

特有方法

void addFirst(E e)

void addLast(E e)

E getFirst()

E getLast()

E removeFirst()

E removeLast()

### Vector

Vector 的方法 和 ArrayList 一样

java.util

接口 Enumeration<E>

此接口的功能与 Iterator 接口的功能是重复的。

**List 集合的特点**

其元素是以线性方式存储 集合中可存储 重复 对象 有序(存取一致)

ArrayList

长度可变的数组 增删慢 查找快

LinkedList

双向链表 增删快 查找慢

Collection集合的子接口

## Set

集合

Set集合的特点:

最简单的一种集合

无序(存取不一致) 无序不代表随机

不可重复

无索引

Set中存入的是对象的引用,没有重复对象

Set集合的方法和Collection集合的方法一致

### HashSet

按照哈希算法来存取集合中的对象 存取速度比较快

当程序向HashSet中 add()的一个对象的时候,

先用hashCode方法计算出该对象的哈希码

哈希码不一致 添加 哈希码一致 不添加 相同对象不添加

然后用equals判断对象的属性是否一致

比较结果为false就添加 true就不添加 不同对象添加

基本数据类型包装类/String 已经重写了hashCode 和 equals 会自动比较

### LinkedHashSet

特点:

1.有序(存取一致)

是Set集合中唯一一个能保证怎么存就怎么取的集合对象

2.排重

3.底层是双向链表

HashSet的方法怎么用 LinkedHashSet就怎么用

### TreeSet

二叉树 实现了SortedSet接口 能够对集合中对象进行排序

特点:

1.排序的(升序) 自然排序 根据ASCII表大小排序

2.排重

3.无序(存取不一致)

4.底层是一个二叉树(左序中序右序)

5.只能存储同一种类型 才能排序 不然就会出现转换异常

自定义实现类 如何比较

方式一 实现 **Comparable 接**口 重写 comparaTo方法

这种方式也称为元素的自然排序 或者是默认顺序排序

int compareTo(T o)

比较此对象与指定对象的顺序。

如果该对象小于、等于或大于指定对象，则分别返回负整数、零或正整数。

负整数 此对象 < 指定对象

0 此对象 == 指定对象

正整数 此对象 > 指定对象

方式二 **Comparator**比较器

TreeSet(Comparator<? super E> comparator)

接口 Comparator<T>

int compare(T o1, T o2)

比较用来排序的两个参数。

根据第一个参数小于、等于或大于第二个参数分别返回负整数、零或正整数。

在实际工作中 尽量不要修改源码 使用Comparator比较器 比较个性的东西

区别:

TreeSet构造函数未传入比较器,默认按照类中的Comparable比较 如果不写 就报错

TreeSet传入Comparator比较器 就优先按照Comparator比较

总结:

**遍历集合的时候**

List:

普通for循环 使用get()获取元素

调用 iterator() hasNext() next()获取元素

增强for循环 只要可以使用Iterator 的类都可以使用

Vector 集合用的是 Enumeration

Set:

调用 iterator()

增强 for

普通for循环 迭代器 增强for循环 是否可以在遍历的过程中删除元素?

普通 for 可以

迭代器 自己的remove() 可以

增强 for 不可以

## Map

将键映射到值的对象。一个映射不能包含重复的键；每个键最多只能映射到一个值。

**Map接口和Collection接口的不同**

1.Map是双列的(夫妻 有结婚证这个联系)

Collection是单列的(单身狗)

2.Map的键是唯一的

Collection的子体系 Set是唯一的

3.Map集合的数据结构是针对键有效的 跟值无关

Collection集合的数据结构是针对元素有效

**特点:**

1.根据键来排重 也就是相同键的元素存储在同一个Map集合中 后添加的键值元素会将之前存储的相同的键值元素替代

2.键和值 都需要是引用数据类型

实现类:

1.HashTable

底层是哈希表 不可以存入null键null值 线程同步的 jdk1.0 效率低

2.HashMap

底层是哈希表 可以存入null键null值 线程不同步的 jdk1.2 效率高

3.TreeMap

底层是二叉树 线程不同步的 可以用于给Map集合中的键进行排序

和Set很像 其实 Set底层就是用了Map集合

Map

**集合中的方法**

增

V put(K key, V value) 添加元素

如果键是第一次存储 就直接存储元素 返回 null

如果键不是第一次存储 就用值把以前的值替换掉 返回被覆盖的值

void putAll(Map<? extends K,? extends V> m)

删

void clear()

V remove(Object key)

根据键删除键值对元素 并把值返回

如果删除的是不存在的key 那么 返回值是一个null

判断

boolean containsKey(Object key) 判断集合是否包含指定的键

boolean containsValue(Object value) 判断集合是否包含指定的值

boolean isEmpty() 判断集合是否为空

查

Set<K> keySet()

Set<Map.Entry<K,V>> entrySet()

V get(Object key)

Collection<V> values()

长度

int size()

**Map集合的两种取出方式**

1.Set<K> keySet()

2.Set<Map.Entry<K,V>> entrySet()

interface Map{

public static interface Entry{

public abstract Object getKey();

public abstract Object getValue();

}

}

### HashMap

特点:

1.无序(存取顺序不一致) 无序不等于随机

2.底层是哈希表

3.允许存入null键null值

LinkedHashMap<K,V>

1.底层存储是有序的(存取一致)

2.底层实现采用双层链表

### TreeMap

特点:

1.排序的(根据键来排序) 前提 键必须具备可比性

2.底层是二叉树

3.无序(存取顺序不一致)

**TreeMap 和 TreeSet一样 需要实现两个接口**

**Comparable 或 Comparator 如果两个都存在以Comparator比较器为准**

## Collections

此类完全在Collection上进行操作或返回 由静态方法组成 专门用来操作集合使用

排序

sort(List<T> list) //1.8新特性

static <T> int binarySearch(List<? extends Comparable<? super T>> list, T key)

static T max(Collection<? extends T> coll)

static void reverse(List<?> list)

static <T> Comparator<T> reverseOrder()

static void shuffle(List<?> list)

static void swap(List<?> list, int i, int j)

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

## 泛型

规定容器存储的数据类型是什么

数组相对安全 因为再创建的时候 就规定了存储的数据是什么类型的

为了提高安全性 我们给集合设置一个指定的存储类型

泛型的基本使用

<> 中放的必须是引用数据类型

泛型 将运行时异常 转移到了编译时 方便解决

泛型分类

**1.泛型类**

把泛型定义在类上

格式:

权限修饰符 class 类名<泛型类型1,泛型类型2....>{

}

为什么要定义泛型类

当类中操作的引用数据类型 不确定的时候 定义泛型来完成确定

泛型的参数名<参数名>

必须是合法的标识符 数字 字符 下划线 $ 组成 但是 不能以数字开头

至于泛型到底是何种类型 由调用者来决定

**2.泛型方法**

格式:

权限修饰符<类型1,类型2...> 返回值类型 方法名(参数列表){

方法体

}

泛型的声明 必须在方法修饰符 public static final abstract等之后 返回值声明之前

可以声明多个 和泛型类一样 用逗号分隔

**3.泛型接口**

格式:

访问权限修饰符 interface 接口名<类型1,....>{

}

**4.泛型高级通配符**

? 通配符 代表的某个

泛型限定

<? extends 类> ? 代表的类型是 [类] 的子类 或 [类]本身

向下限定 类及其子类

<? super 类> ? 代表的类型是 [类]类型 或 当前[类]的父类

向上限定 类及其父类

## Collection&map

### Collection (单列)

List(存取有序,有索引,可重复)

Vector

底层是数组 线程安全 增删慢 查询快 (jdk1.0) 无论增删改查都慢

ArrayList

底层是数组 线程不安全 增删慢 查询快

LinkedList

底层是双向链表 线程不安全 增删快 查询慢

如果查询和修改多 用 ArrayList

如果增和删多 用LinkedList

如果都多 用ArrayList

ListIterator迭代器

Set(存取无序,无索引,不可重复)

HashSet

底层是哈希算法实现

LinkedHashSet

底层是双向链表 但是也可以保证元素唯一 和HashSet原理一致

TreeSet

底层是二叉树

一般在开发中不需要对存储的元素排序 所以用HashSet比较多

TreeSet 面试时候比较多 两种排序

遍历集合

有Iterator 迭代器

### Map (双列)

HashMap

底层是哈希算法实现 针对键

LinkedHashMap 底层是链表 针对键

TreeMap

底层是二叉树 针对键

开发中 用 HashMap较多

遍历集合 要转成Set 在使用Iterator

keySet()

entrySet()

# 九、IO流

用来处理设备之间的数据传输

java对数据的操作是通过流的方式

java用于操作流的对象都在IO包中

## IO的种类

流向分:

输入流:从磁盘等物理介质到当前内存中

InputStream(字节流) Reader(字符流)

输出流:将程序处理好的数据 从 内存中输出到磁盘等存储介质中

OutputStream(字节流) Writer(字符流)

数据类型不同分:

字节流:流中的数据以字节为单位流动

InputStream OutputStream

字符流:流中的数据以字符为单位流动

Reader Writer

## 字符流

字符流就是可以直接读写字符的IO流

底层 也是字节流

ASCII 美国信息标准交换码

GB2312 -->扩容 GBK(2万多)-->18030

unicode --->UTF-8 国际编码表

为了处理乱码问题 字符流可以在内部融合不同的编码表(由开发人员指定)

方便处理字符类文件

Writer

FileWriter

Reader

FileReader

## 字节流

InputStream :字节输入流 从磁盘到内存 -->Reader

OutputStream :字节输出流 从内存到磁盘 -->Writer

FileInputStream

FileOutputStream

//根据文件大小做一个字节数组

byte[] arr = new byte[fis.available()];//弊端 有可能会造成内存溢出

字节流缓冲区

BufferedInputStream(InputStream in)

BufferedOutputStream(OutputStream in)

字节流一次读写一个数组的速度明显比一次读一个字节速度快

java本身在设计的时候 就加入了这样的功能 就是缓冲区

BufferedInputStream会一次性从文件中读取8192个 存在缓冲区中,返回给程序一个

程序再次读取的时候 就不用找文件了 直接从缓冲区中获取

BufferedOutputStream程序向流中写出字节时,不会直接写到文件中,先写到缓冲区中

直到缓冲区装满 才会把缓冲区中的数据异型性写到文件里

/\*数组的读写和带Buffered的读写哪个更快?

\* 如果数组定义8192个字节大小 和 Buffered比较的话

\* 数组会略胜一筹 因为 读和谐 操作的都是同一个数组

\* 而Buffered操作的是两个数组

\* \*/

## 转换流

Reader 类 字节转字符 输入流 InputStreamReader 是字节流通向字符流的桥梁：

InputStreamReader(InputStream in) 创建一个使用默认字符集的InputStreamReader

InputStreamReader(InputStream in, Charset cs) 创建使用给定字符集的InputStreamReader

InputStreamReader(InputStream in, String charsetName) 创建使用指定字符集的InputStreamReader

Writer 类 字节转字符 输出流 OutputStreamWriter 是字节流通向字符流的桥梁

OutputStreamWriter(OutputStream out) 创建使用默认字符编码的OutputStreamWriter

OutputStreamWriter(OutputStream in, Charset cs) 创建使用给定字符集的OutputStreamWriter

OutputStreamWriter(OutputStream in, String charsetName) 创建使用指定字符集的OutputStreamWriter

流操作的基本规律

流对象有很多 不知道用哪一个

**通过三个明确来完成**

1.明确源和目的地

源:输入流 InputStream Reader

目的: 输出流 OutputStream Writer

2.明确操作的数据是否是纯文本

是:字符流

否:字节流

3.当体系明确后,再明确要使用哪个具体的对象

通过设备来进行划分

源设备:内存,硬盘,键盘

目的设备:内存,硬盘,控制台

**一.转换流的设计目的**

InputStreamReader(InputStream in, Charset cs)

InputStreamReader(InputStream in, String charsetName)

OutputStreamWriter(OutputStream out, Charset cs)

OutputStreamWriter(OutputStream out, String charsetName)

字符编码

字符流的出现为了方便操作字符

更重要的是加入了编码转换 就是通过子类转换流来完成

在两个对象构造的时候 可以加入字符集

编码表的由来

计算机只能识别二进制数据 早期由来是电信号

为了方便计算机的应用 让它可以识别多国的文字 就将各个国家的文字用数字来表示 并一一对应 形成一张表 就叫编码表

常见的编码表

ASCII 美国标准信息交换码

用一个字节的7位可以表示

ISO8859-1 拉丁码表 欧洲码表

用一个字节的8位表示

GB2312 中国的中文编码表

一个中文两个字节 16位 中国的码表兼容ASCII码表 汉字高位都是1

GBK 中国的中文编码表升级 融合了更多的中文文字符号

Unicode 国际标准码 融合了多种文字

所有文字都用两个字节来表示 java语言使用的就是unicode

UTF-8 最多用三个字节来表示一个字符

编码的转换

编码:字符串变成字节数组

String --> byte[] 即 str.getBytes(charsetName);

解码:字节数组变成字符串

byte[]--> String 即 new String(byte[],charsetName);

转换流的好处

1.操作文本文件时,将字节流转换成字符流 便于操作

2.可以解决字符集乱码问题

在读取的过程中 读和写都是设置同一个字符集 就不会出现乱码 但是要保证

读的时候 就是对的 原文件是什么编码集 你就需要设置相同的编码集去读 去写

## 字符流缓冲区

BufferedReader

BufferedWriter

缓冲区的出现提高了对数据的读写效率

缓冲区要结合流在可以使用

在流的基础上 对流的功能进行了增强

BufferedReader

|-----LineNumberReader

**什么情况下使用字符流**

文本文件 读取一段文本 或者是写出一段文本

不可以拷贝非纯文本文件

因为在读的时候 会将字节转换成字符 在转换的过程中 可能找不到对应的字符

就会用?替代 写出的时候 将字符转换成字节写出 如果是? 直接写出 这样写出之后的文件就乱了

图片,音频,视频 这些不要使用当前的字符流

## 装饰设计模式

IO流部分

read()一次读一个

readLine()一次读一行 其实就是在增强了 read方法的功能

BufferedReader 就是 对 FileReader的增强

这种模式 我们叫做 装饰设计模式

概念:

装饰类 通常会通过构造方法 去接收被装饰对象

并给予被装饰类的功能 提供更强的功能

装饰和继承的区别

例如:

MyReader //专门用于读取数据的类

|---MyTextReader

|--- MyBufferedTextReader

|---MyMediaReader

|--- MyBufferedMediaReader

|--MyDataReader

|--- MyBufferedDataReader

弊端:如果再出现其他子类 又要给一个Buffered 体系臃肿且复杂 需要记忆的类太多了

装饰设计模式

MyReader

|--MyBufferedReader(MyReader reader)

你是读 那么我增强完之后 还是读 所以 继承于MyReader

好处:比继承要更灵活,避免了继承体系的臃肿,而且降低了类与类之间的关系

## File

概念

用来将文件或者文件夹封装成对象

好处

方便对文件或文件夹的属性信息进行操作

File对象可以作为参数传递给流的构造器

路径

绝对路径和相对路径

绝对路径:固定的一个路径 从盘符开始

相对路径:相对于某个位置 在eclipse下是指当前项目下

流只能操作数据 想要操作数据封装成的文件信息 必须用File

构造方法

File(String pathname)

File(File parent, String child)

File(String parent, String child)

创建功能

public boolean createNewFile() 创建文件 如果存在这样的文件 就不创建了

public boolean mkdir() 创建文件夹 如果存在 就不创建了

public boolean mkdirs() 创建文件夹 如果父文件夹不存在 会帮你创建出来

重命名和删除功能

public boolean renameTo(File dest) 把文件重命名为制定的文件路径

重命名注意事项:

如果路径名相同 就是改名

如果路径名不同 就是改名并剪切

public boolean delete() 删除文件或文件夹

删除注意事项:

如果删除一个文件夹 那么文件夹必须是空的

java中的删除不走回收站

判断功能

public boolean isDirectory() 判断是否是目录

public boolean isFile() 判断是否是文件

public boolean exists() 判断文件或者文件夹是否存在

public boolean canRead() 判断是否可读

public boolean canWrite() 判断是否可写

public boolean isHidden() 判断是否隐藏

Linux

u: User 即文件或者目录的拥有者

g: Group 即文件或目录的所属群组

o: other 除了文件或目录拥有者或群组之外的 其他用户都属于这个范围

r:读取权限 数字代表 4

w:写入权限 数字代表 2

x:执行或切换权限 数字代表 1

chmod 777

获取功能

public String getAbsolutePath() 获取绝对路径

public String getPath() 获取路径

public String getName() 获取名字

public long length() 获取长度 字节数

public long lastModified() 最后一次修改时间 毫秒值

public String[] list() 获取指定目录下的所有文件或文件夹的名称数组

public File[] listFile() 获取指定目录下的所有文件或文件夹的File数组

static String separator 跨平台分隔符 File.separator

## Properties

Properties 是HashTable的子类

也就是说它具备Map集合的特点 而且它里面存储的键值对都是字符串 不需要泛型

是集合中和IO技术相结合的集合容器

主要适用于键值对形式的配置文件

配置文件:将配置信息进行持久化存储

配置文件中以#开头 都是注释

特殊功能:

public Object setProperties(String key,String value);

public String getProperties(String key);

void load(InputStream inStream) 将流中的数据加载进集合

void store(OutputStream out, String comments) 将内存中的结果 存入到流中 并存到文件上

commets : 是注释信息 可不写 不要写中文 解析不出来 爱加不加 #注释信息 不会被加载

## 其他流对象

### 打印流

PrintWriter 与 PrintStream 可以直接操作输入流和文件

该流提供了打印方法 可以对基本数据类型进行直接操作 保持数据的原样性 打印出去

PrintStream 字节打印流

构造函数可以接收的参数类型

1.File对象

2.字符串路径

3.字节输出流 OutputStream

PrintWriter 字符打印流

在web阶段常用 web阶段都是用它将数据打到客户端 让客户端对他解析执行

B/S

C/S

构造函数可以接收的参数类型

1.File对象

2.字符串路径

3.字节输出流 OutputStream

4.字符输出流 Writer

PrintWriter(OutputStream out, boolean autoFlush)

autoFlush - boolean 变量；如果为 true，则 println、printf 或 format 方法将刷新输出缓冲区

PrintWriter(Writer out, boolean autoFlush)

此类中的方法不会抛出 I/O 异常

### 标准输入输出流

System.in是 inputStream 标准输入流 默认可以从键盘输入读取字节数据

System.out是 PrintStream 标准输出流 默认可以向控制台中输出字符和字节数据

修改标准输入输出流

修改输入流: System.setIn(InputStream)

修改输出流: System.setOut(PrintStream)

修改标准输入输出流的源和目的地 这样可以实现拷贝的功能

### 序列流

什么是序列流?

序列流可以把多个字节输入流整合成一个

从序列流中读取数据时,将从被整合的第一个流开始读,读完一个读下一个 以此类推

把最后一个小流的 -1 作为 大流的结束

(就是将多个流对象 拼接成一个流对象 逻辑串联)

N vs 1 --> 1 vs 1

SequenceInputStream 将多个 源 合并 将多个源头 变成一个源头 没有 outPut的!!!!

SequenceInputStream(InputStream s1, InputStream s2)

SequenceInputStream(Enumeration<? extends InputStream> e)

### 对象操作流

什么是对象操作流?

该流可以将一个对象写出 或者 读取一个对象到程序中 也就是执行了序列化和反序列化的操作

ObjectInputStream

ObjectOutputStream

Serializable 这种接口 没有方法 称为 标记接口 给你的类打了个戳

其实就是给类加了一个 serialVersionUID 号 就证明你的类具备了序列化

持久化/序列化: 找一个介质 长期保存数据

对象存储在堆内存中 想要将对象存储在物理介质中 ---> 对象的序列化/持久化

数据随着对象 存入到硬盘上 如果下次想要使用 只要读取该文件 就可以再次使用

序列化 是将 堆内存中的数据 序列化 被static修饰的成员 无法序列化

如果想要非静态的成员还是存在堆内存中 又不想被序列化

就添加一个关键字 transient

transient使用小结:

当类实现了Serializable接口 那么类中属性就是默认能被序列化的

为了让某些数据 即能使用 又不被序列化 用一次就消亡 就可以使用当前关键字

1.一旦被transient修饰 变量将不再是对象序列化的一部分 该变量内容在序列化后 无法获得访问

2.transient关键字只能修饰变量 不能修饰方法和类

3.被transient关键字修饰的变量不再被序列化 和 static修饰的一样

但是 在内存中的存储区域不一样 不可混淆

一个被static的修饰的变量 是不能被序列化的

例如:

网吧登录 密码 银行卡号 这些 敏感信息 不希望被序列化 用一次就可以了

对象操作流注意事项:

1.ObjectInputStream和ObjectOutputStream所读写的对象 必须实现Serializable接口

2.对象中被static 或者 transient 修饰的成员变量 不能被序列化

3.自己写serialVersionUID 号

### 数据输入输出流

DataInputStream

DataOutputStream

可以按照基本数据类型大小读写数据

例如: 按Long大小写出一个数字 写出时该数据占8个字节 读取的时候也可以按照Long来读取 一次读8个

仅为了操作基本数据类型 是 流 和 数据相结合的功能 分别存储对应的字节数

构造函数

DataOutputStream(OutputStream out)

void write(int b)

1 int = 4 byte = 32 bit 将指定字节（参数 b 的八个低位）写入基础输出流。

void writeUTF(String str)

以与机器无关方式使用 UTF-8 修改版编码将一个字符串写入基础输出流。

gbk 一个中文俩字节

utf-8 一个中文仨字节

writeUTF 一个中文四个字节

### 内存流ByteArrayStream

可以直接操作字节数组中数据的流对象

字节流内部本来就可以操作字节数组 这个流有和意义呢?

该输出流可以向内存中写数据 把内存当做一个缓冲区 写出之后可以一次性获取出所有数据

ByteArrayInputStream

在构造的时候 需要接收数据源 而且数据源是一个字节数组

ByteArrayOutputStream

在构造的时候 不用定义数据的目的地 因为该对象中已经内部封装了可变长度的字节数组 这就是目的地

因为这两个流都操作的数组 并没有使用系统资源 所以不用进行close关闭

ByteArrayInputStream(byte[] buf)

ByteArrayOutputStream()

//写出到文件中

bos.writeTo(new FileOutputStream("x.txt"));

//将内存缓冲区中所有的字节存储到数组中

byte[] newArr = bos.toByteArray();

### RandomAccessFile

java.io

类 RandomAccessFile 不属于流 是 Object类的子类,

但是它融合了InputStream和 OutputStream的功能

支持对随机访问文件的读取和写入

这个对象内部封装了一个byte数组 可以通过指针对数组的元素进行操作

可以通过

getFilePointer方法获取指针位置

seek方法 改变指针的位置

构造函数

public RandomAccessFile(File file,String mode)

mode 参数指定用以打开文件的访问模式。

值 含意

"r" 以只读方式打开。调用结果对象的任何 write 方法都将导致抛出 IOException。

"rw" 打开以便读取和写入。如果该文件尚不存在，则尝试创建该文件。

"rws" 打开以便读取和写入，对于 "rw"，还要求对文件的内容或元数据的每个更新都同步写入到底层存储设备。

"rwd" 打开以便读取和写入，对于 "rw"，还要求对文件内容的每个更新都同步写入到底层存储设备。

void seek(long pos)

例如:有一个文件 这个流可以分段写入数据

我们用一个线程负责文件的其中一段 多线程可以同时向文件中写入内容

互不冲突 这就是多线程下载

如果用其他的流 都是从头向后写 那么第一个线程写个数据 第二个线程接着写另一个数据

以此类推 虽然数据存完了 但是 这样会导致读出来的都是错误的 不是完整的文件 解码错误

io中 只有这个可以实现多线程下载 seek方法

# 十、线程

一.线程的概念

**程序**:一段代码,一组指令,它本身没有任何运行的含义 它只是一个实体 是应用软件执行的蓝本

**进程**:是程序一次动态执行 它对应着从代码加载 执行 到执行完毕的一个完整的过程

它是动态的 它有自己的生命周期

它因创建而产生 因调度而运行 因等待或事件而处于等待状态 因完成任务而结束

**线程**: 线程就是进程里的一个控制单元 是线程在控制进程

每一个进程里面都包含有若干个线程 一个进程至少要有一个线程

多线程并发执行可以提高程序的效率,可以同时完成多项工作

**进程和线程的区别:**

线程:进程的某个执行单元

进程:一个进程可以包含多个线程,进程一般是由线程组成的 一个进程至少要有一个线程

**什么是多线程?**

多线程是实现了并发机制的一种有效手段一个进程包含多个线程共享内存

同时运行 可以提高程序的执行效率

**多线程的并发和并行**

并行:就是两个任务同时运行 多核cpu (用两台电脑 分别和两个人聊天)

并发:是指两个任务同时请求运行 而处理器一次只能接受一个任务 就会把两个任务安排轮流执行

由cpu来决定 (用一台电脑 分别和两个聊天)

**java的程序运行原理**

java命令会启动jvm 等于启动了一个应用程序

也就是启动了一个进程 该进程为自动启动一个"主线程"

然后主线程去掉用某个类的main函数

**java的启动是多线程的吗?**

jvm启动至少启动了 主线程 和 垃圾回收线程 所以是多线程的

二.线程的创建方式

线程运行状态

虽然wait 但是 线程没结束 例如 程序死了 但进程还在

任务管理器-->结束进程(强制干掉线程)

线程被start()了之后却不一定运行 例如 开了四个线程 抢cpu但是 有一个一直没抢到

冻结:放弃了执行资格 解冻之后 会回到 临时状态

NEW(出生)

至今尚未启动的线程处于这种状态。

RUNNABLE(准备就绪和执行)

正在 Java 虚拟机中执行的线程处于这种状态。

BLOCKED(睡眠)

受阻塞并等待某个监视器锁的线程处于这种状态。

WAITING(等待)

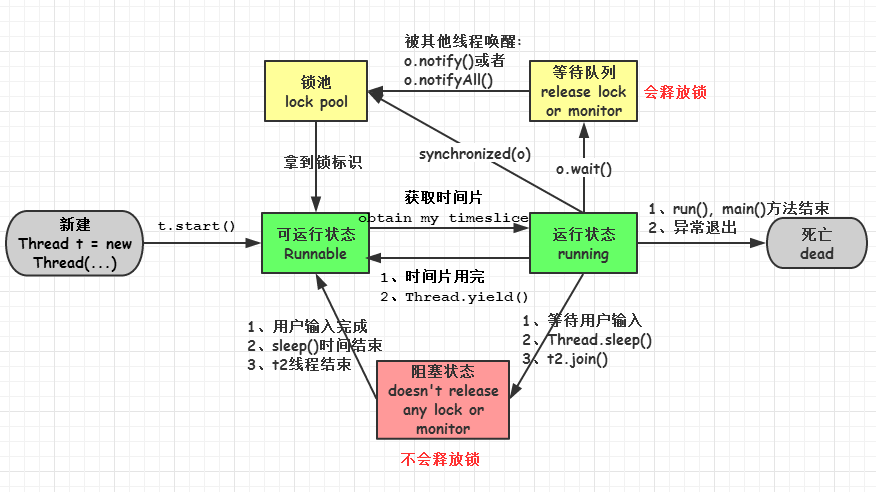
无限期地等待另一个线程来执行某一特定操作的线程处于这种状态。

TIMED\_WAITING(时间等待)

等待另一个线程来执行取决于指定等待时间的操作的线程处于这种状态。

TERMINATED(消亡)

已退出的线程处于这种状态。



## 线程间通信

操作同一个资源 多个线程 操作的动作不同

等待唤醒机制

wait() 等待 进入冻结状态 如果无人唤醒 就一直等待

notify() 唤醒

notifyAll() 唤醒所有

sleep(long) 睡眠 自然醒来

输入线程进来 先判断标记 如果标记是false 就输入 输入后 将标记改为true

输入线程第二次进来 发现标记是true 等着 等待输出线程执行完毕 将它唤醒 继续执行

输出线程进来 先判断标记 如果标记是true 如果是 true 就输出 输出后 将标记改为false

并且 唤醒输入线程

输出线程第二次进来 发现标记是false 就等着 等待输入线程执行完毕 将它唤醒 继续执行

input线程 执行后 唤醒output线程 自己等待

output线程 执行后 唤醒input线程 自己等待

始终保持着 一个等待执行 一个唤醒执行的状态

例如:两个人吃一碗饭 交替着吃

我吃的时候你别动 你吃的时候我不动

wait() notify() notifyAll()

都用在同步中 因为要对持有监听器(锁)的线程进行操作

所以要使用在同步中 因为了只有同步 才具有锁

因为 等待 和 唤醒 必须是同一个锁

为什么 wait() notify() notifyAll() 这三个方法是Object类的方法

因为锁 可以是任意对象 所以可以被任意对象调用的方法 定义在 Object类中

## 生产消费模式

先有产品 才能消费

//线程池等待 先进去的先被唤醒

/\*

改成while之后 是让所有的线程 醒了之后都判断标记 就不会出现 生产两个 消费一个的情况

用了 while 之后的结果 造成 全部等待的局面

需要用 notifyAll() 唤醒所有的等待线程

notify() 是唤醒线程池中第一个 所以可能会唤醒 本方线程

notifyAll(); 全部唤醒 循环判断标记 就是到什么时候该懂 什么时候不动了

\*/

分析:

假设 生产者获得cpu执行权

t1 t2

t1 判断标志位 false 赋值 打印一次 改成true notify()

t1 有可能还持有cpu执行权 判断标志位 true t1 wait() 放弃执行权

t2/t3/t4 都有可能抢到执行权

如果 t2 抢到执行权 判断标志位 true t2 wait() 放弃执行权

如果 t3 抢到执行权 判断标志位 !true 打印 改成false notify() t1 t1 有资格 但不一定有执行权

(注意:线程池等待 先进去的先被唤醒)

t3 又抢到了执行权 !false t3 wait() t3放弃资格

t4 抢到了 !false t4 wait() t4放弃资格

t1 活了 t1 count++ 打印一次 改成 true notify() 唤醒 t2 t2有资格

t1 再进来 判断标志是 true wait()

这时候活着的只有 t2

t2 执行了 不会判断标志位 因为之前是 wait() 直接向下执行代码

count ++ 前一个t1生成的数 被覆盖了

修改标记 唤醒 t3

t3 活了 打印一次 打印的是t2的值

如何才能在每次醒来的时候都判断标记呢?

if 改成 while 判断多次 --> t2醒来 判断标记 true wait()

改成while让每次醒来 都判断 不管是谁醒来 都要判断 会造成 全部等待的局面

没有线程可以启动notify()

notifyAll() 全部唤醒 唤醒了全部之后 循环判断标记 就知道是哪个线程该启动了

**public** **class** Produce\_Consume {

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

// **TODO** Auto-generated method stub

Goods g=**new** Goods();

**new** Thread(**new** Produce("生产者1", g)).start();

**new** Thread(**new** Produce("生产者2", g)).start();

**new** Thread(**new** Consume("消费者1", g)).start();

**new** Thread(**new** Consume("消费者2", g)).start();

}

}

**class** Goods

{

**private** String name;

**private** **int** count=0;

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

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

{

**while**(flag)

{

**try** {

**this**.wait();

} **catch** (Exception e) {

// **TODO**: handle exception

}

}

System.***out***.println(Thread.*currentThread*().getName()+": "+name+" is make "+(++count));

flag=**true**;

**this**.notifyAll();

}

**public** **synchronized** **void** out(String name)

{

**while**(!flag)

{

**try** {

**this**.wait();

} **catch** (Exception e) {

// **TODO**: handle exception

}

}

System.***out***.println(Thread.*currentThread*().getName()+": "+name+" is consume "+count);

flag=**false**;

**this**.notifyAll();

}

}

**class** Produce **implements** Runnable

{

**private** Goods g;

String name;

Produce(String name,Goods g) {

**this**.g=g;

**this**.name=name;

}

@Override

**public** **void** run() {

// **TODO** Auto-generated method stub

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

{

**try** {

g.set(name);

} **catch** (Exception e) {

// **TODO**: handle exception

}

}

}

}

**class** Consume **implements** Runnable

{

**private** Goods g;

**private** String name;

Consume( String name,Goods g) {

**this**.g=g;

**this**.name=name;

}

@Override

**public** **void** run() {

// **TODO** Auto-generated method stub

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

{

**try** {

g.out(name);

} **catch** (Exception e) {

// **TODO**: handle exception

}

}

}

}

## Thread类中的其他方法

### 1.停止线程

public void interrupt()

如果线程在调用 Object 类的 wait()、wait(long) 或 wait(long, int) 方法，

或者该类的 join()、join(long)、join(long, int)、sleep(long) 或 sleep(long, int) 方法过程中受阻，

则其中断状态将被清除，它还将收到一个 InterruptedException。

定义循环结束标记

因为线程运行代码一般都是循环 只要控制了循环 线程就能被停止

interrupt() 中断方法

该方法是结束线程的冻结状态 使线程回到运行状态中来

注意: stop() 方法已经过时不再使用 是强行停止 不管什么情况 线程都会被干掉

如何停止线程?

run方法结束

interrupt()方法 是 将线程从 冻结状态 强制 转成 运行状态

### 2.守护线程

void setDaemon(boolean on)

设置一个线程为守护线程 那么该线程就不会单独执行 当其他非守护线程都执行结束后 守护线程自动结束

后台线程 当线程被标记为守护线程 开启后 和前台线程一样 都抢夺cpu的执行权

但是 结束不一样 当所有前台线程都结束后 后台线程会自动结束

/\*

注意:

1.在启动线程前设置

2.前台线程结束了 所有后台线程就结束了

例如:

圣斗士星矢 守护雅典娜

\*/

### 3.加入线程

join() 插队 当前线程暂停 等待指定的线程执行结束后 当前线程再继续

join(long) 可以等待指定的毫秒之后继续

// 插队 想要执行权 那么被插队的线程放出执行权 让 插队的先执行

// join 可以用来临时加入线程执行

### 4.设置线程的优先级

setPriority()

public final void setPriority(int newPriority)

线程的优先级 1-10

默认线程的优先级 5

toString() 方法 返回该线程的字符串表示形式，包括线程名称、优先级和线程组。

当修改了优先级 抢夺cpu 的频率会发生改变

static int MAX\_PRIORITY

线程可以具有的最高优先级。

static int MIN\_PRIORITY

线程可以具有的最低优先级。

static int NORM\_PRIORITY

分配给线程的默认优先级。

### 5.礼让线程

public static void yield() 让位 稍微减缓线程执行频率

让线程的使用率 变得更平均

计时器

Timer()

## 线程池

假设一个服务器完成一项任务

需要是时间为 T1 创建线程时间 T2在线程中执行任务的时间 T3销毁线程时间

如果 T1+T3 远大于 T2 则可以采用线程池 以提高服务器的性能

如果并发的线程数量很多 并且每个线程都是执行一个时间很短的任务就结束了

这样频繁创建线程 就会大大降低系统的效率 因为频繁创建/销毁线程都需要时间

为了提高线程的复用性 就是执行完一个任务 并不销毁 而是继续执行其他任务

### 线程池概述

线程池是一种多线程处理形式,处理过程中将任务添加到队列,然后再创建线程后

自动启动这些任务,线程池线程都是后台线程.每个线程都有默认的堆栈大小,

以默认的优先级运行 并处于多线程单元中

线程池中的线程并不会死亡,而是再次回到线程池中成为空闲状态,等待下一个对象来使用

ThreadPoolExecutor(

int corePoolSize, 核心线程池大小

int maximumPoolSize, 线程池最大容量大小

long keepAliveTime, 线程池空闲时,线程存活的时间

TimeUnit unit, 时间单位 keepAliveTime存活时间的时间单位

BlockingQueue<Runnable> workQueue, 任务队列

ThreadFactory threadFactory, 线程工厂

RejectedExecutionHandler handler 线程拒绝策略

)

void execute(Runnable command) 将线程提交到线程池中

第一步:

调用execute方法提交线程 首先检查CorePool

如果corePool内的线程小于corePoolSize 就新创建线程执行任务

第二步:

如果当前corePool内的线程大于等于corePoolSize 那么将线程加入到

BlockingQueue 等待队列中

第三步:

如果不能加入BlockingQueue 在小于maximumPoolSize的情况下创建线程执行任务

第四步:

如果线程数大于maximumPoolSize 那么执行拒绝策略

一个 ExecutorService，它使用可能的几个池线程之一执行每个提交的任务，

通常使用 Executors 工厂方法配置。

强烈建议程序员使用较为方便的 Executors 工厂方法

### Executors 四种线程池

1. **static ExecutorService newCachedThreadPool()**

创建一个可缓存线程池 如果线程池长度超过处理需要,可灵活回收空闲线程

若无可回收 则新建线程

1. **static ExecutorService newFixedThreadPool(int nThreads)**

创建一个定长线程池,可以控制线程的最大并发数,超出的线程会在队列中等待

1. **static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)**

创建一个定长线程池 支持定时及周期性 延迟

1. **static ExecutorService newSingleThreadExecutor()**

创建一个单线程化的线程池 它只会用唯一的工作线程来执行任务

此线程保证所有的任务执行顺序按照任务的提交顺序执行

这个线程池可以在线程死后(或发生异常时)重新启动一个线程来替代原来的线程继续执行下去

FIFO:first input first output 先进先出队列

LIFO:last in first out 后进先出队列

优先级

**static ExecutorService newCachedThreadPool()**

execute 向线程池中提交任务

submit(Runnable task) 提交一个 Runnable 任务用于执行，并返回一个表示该任务的 Future。

shutdown();

/\*

\* 线程池为无限大 当执行第二个任务时 第一个任务如果已经完成 会复用执行第一个任务的线程

\* 而不会每次创建新线程

\* 如果现有线程没有可用的,则创建一个新的线程并添加到池中

\* 终止并从缓存中移除那些已有 60 秒钟未被使用的线程。

\* \*/

**static ExecutorService newFixedThreadPool(int nThreads)**

/\*

\* 因为线程池大小设置为3 每个任务输出index后 sleep2秒 所以每次两秒打印3个数字

\* 注意:

\* 定长的线程池大小最好根据系统资源进行设置

\* 如: Runtime.getRuntime().availableProcessors()

\* 此方法放回到虚拟机的最大可用处理器数量 cpu核数

\* \*/

**static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)**

ScheduledExecutorService

一个 ExecutorService，可安排在给定的延迟后运行或定期执行的命令。

ScheduledFuture<?> schedule(Runnable command, long delay, TimeUnit unit) 延迟 执行

command - 要执行的任务

delay - 从现在开始延迟执行的时间

unit - 延迟参数的时间单位

**ScheduledFuture<?> scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit)**

command - 要执行的任务

initialDelay - 首次执行的延迟时间

period - 连续执行之间的周期

unit - initialDelay 和 period 参数的时间单位

**static ExecutorService newSingleThreadExecutor()**

# 十一、Lambda表达式\*\*\*\*\*

语法糖 (Syntactic Sugar)也叫糖衣语法

**java语法糖:**

1.泛型

2.自动装箱拆箱

3.forEach循环(增强for循环)

4.内部类

5.枚举

6.switch支持 枚举 和 字符串

在计算机语言中添加某种语法,这种语法使程序员使用更方便,同时增强程序代码的可读性

避免出错的机会,这种语法对语言本身的功能并没有影响

**什么是Lambda表达式**

允许你通过表达式来代替功能接口,其就和方法一样,提供了一个正常的参数列表和一个使用这些参数的主体

Lambda表达式还增强了集合库

**功能接口**:只包含一个方法的接口 叫做功能接口

例如 Runnable Comparator

Lambda表达式 匿名函数 即没有函数名的函数

**使用场景**

1.Lambda表达式用来实现单一的行为 并且该行为要被传递给其他的代码

2.当只需要一个功能接口的简单实例,而不需要类型,构造方法和一些相关的类内容

3.只在需要新的字段和功能的时候使用匿名内部类

**语法格式**

(parameters)-> expression 或者(parameters)-> {expression;}

1.->:表示"转到"或"成为" 左边指定输入参数 右边是Lambda的主体

一个参数 param->expr

两个参数 (param1,param2)->expr

参数列表 用括号包裹 以逗号分隔

例如:

public int add(int x,int y){

return x+y;

}

Lambda写法

(int x,int y)->x+y;

(x,y)->x+y 编译器会根据上下文自动推理出来

(x,y)->{return x+y;} 显式指明返回值

语法由三部分组成:参数列表,箭头,以及一个表达式或者语句块

语法有如下特征

[可选] 类型声明

[可选] 小括号 单个参数可以不写

[可选] 花括号 如果只有一条语句 可以不写

[可选] return关键字 如果表达式主体是单一 return可以不写 编译器可以自动返回

当然 如果写了return 就必须加上花括号

基本的Lambda表达式应用案例

## 线程中的使用

用 Lambda来实现Runnable接口

//基本版

Thread td = new Thread(new Runnable(){

public void run() {

System.out.println("基本");

}

});

td.start();

//Lambda版本

Thread td2 = new Thread(()->System.out.println("Lambda版本"));

td2.start();

## 自定义(仿照线程)

interface MyRunnable{

public void myRun();

}

class MyThread{

private MyRunnable myrunn;

public MyThread(MyRunnable myrunn) {

this.myrunn = myrunn;

}

public void start() {

myrunn.myRun();

System.out.println("start!!");

}

}

public class Demo\_MyThread {

public static void main(String[] args) {

MyThread my = new MyThread(new MyRunnable() {

public void myRun() {

System.out.println("我跑起来了");

}

});

my.start();

System.out.print("lambda===========");

MyThread my2 = new MyThread(()->System.out.println("我跑起来了"));

my2.start();

}

}

## forEach

**public** **static** **void** lam\_map\_foreach() {

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

map.put("B", 20);

map.put("A", 30);

map.put("C", 40);

**for** (Entry<String, Integer> en:map.entrySet()) {

System.***out***.println(en.getKey()+":"+en.getValue());

}

map.forEach((a,s)->System.***out***.println(a+":"+s));

}

**public** **static** **void** lam\_list\_foreach() {

String[] arr= {"df","sf","ffd","tom"};

List<String> list=Arrays.*asList*(arr);

list.forEach((ss)->System.***out***.println(ss));

}

## 集合的排序

**public** **static** **void** lam\_list\_sort() {

List<String> list=Arrays.*asList*(**new** String[] {"a","b","c"});

//Comparator<String> com2 = (String s1,String //s2)->s1.length()-s2.length();

// Arrays.sort(list,com2);

Collections.*sort*(list,(a,b)->a.compareTo(b));

System.***out***.println(list);

}

## 数组的排序

**public** **static** **void** lam\_array\_sort() {

String[] arr= {"df","sf","ffd","tom"};

Arrays.*sort*(arr,(a,b)->a.compareTo(b));

System.***out***.println(Arrays.*toString*(arr));

}

## 类的加载和加载时机

当程序要使用某个类的时候

如果该类还未加载进入内存中 则系统会通过 加载,连接,初始化三步骤 将类加载入内存

### 加载

就是指将class文件读入到内存中 并为之创建一个Class对象

任何类被使用 系统都会创建一个Class对象

就是将硬盘上.class字节码文件加载到内存中 变成Class对象

### 连接

验证 是否有正确的内部结构

准备 负责为类的静态成员分配内存 并设置默认初始化值

static 随着类的加载而加载

解析 将类的二进制数据中的符号替换为直接引用

### 初始化

就是我们以前讲过的初始化的步骤

## 反射 (reflect)

**简介**

java反射机制 在运行状态中 对于任意一个类 都能够知道这个类的所有属性和方法

对于任意一个对象 都能够调用它的任意一个方法和属性

这种动态获取信息以及动态调用对象的方法 称为 java语言的 **反射**

想要解剖一个类 必须先要得到这个类的字节码对象

"正" 先知道类 然后去创建对象 调用类中的方法或者属性

"反" 通过对象找到类

这个功能 就是编译工具 自动去审查这个类

### 字节码文件的获取

获取每一个字节码文件对应的Class类型对象 有三种方式

\* A Object类的getClass() 方法 获取Class类型对象 多用于判断字节码文件是否是同一个

\* B 静态属性 class 多用于静态方法 同步 的锁

\* C Class类中的forName()方法 多用于读取配置文件

### 应用

获取配置文件中的内容

创建对象的方式2 通过Class类中的newInstance()方法 创建对象

public T newInstance()创建此 Class 对象所表示的类的一个新实例。

### 反射的四个类

### java.lang

Class 代表的类 对象

### java.lang.reflect

Constructor 构造函数

Field 属性

Method 方法

#### Constructor 构造函数

获取空参构造

Public T newInstance() throws InstantiationException,IllegalAccessException

创建此 Class 对象所表示的类的一个新实例。

如同用一个带有一个空参数列表的 new 表达式实例化该类。

如果该类尚未初始化，则初始化这个类。

(就是用类的空参构造方法 来创建对象 只是把原来的创建对象方式 封装到一个方法里了)

获取有参构造

Constructor<T> getConstructor(Class<?>... parameterTypes)

返回一个 Constructor 对象，它反映此 Class 对象所表示的类的指定公共构造方法。

T newInstance(Object... initargs)

使用此 Constructor 对象表示的构造方法来创建该构造方法的声明类的新实例，

并用指定的初始化参数初始化该实例。

#### Field 属性

##### 非私有

Class 类中 Field getField(String name)

Field 类中

Object get(Object obj)

void set(Object obj, Object value)

##### 私有

Class 类中 public Field getDeclaredField(String name)暴力反射

java.lang.reflect.AccessibleObject

public void setAccessible(boolean flag)去除私有权限

#### Method 方法

##### 无参方法

Method getMethod(String name, Class<?>... parameterTypes)

返回一个 Method 对象，它反映此 Class 对象所表示的类或接口的指定公共成员方法。

Class<?>... parameterTypes 可写可不写 按照方法来选择是否要写

##### 有参方法

Method getMethod(String name, Class<?>... parameterTypes)

调用方法 使用 Method类中的invoke方法

Object invoke(Object obj, Object... args)

java.lang.reflect.Proxy

Proxy 提供用于创建动态代理类和实例的静态方法，它还是由这些方法创建的所有动态代理类的超类。

## 动态代理

### 创建某一接口的代理：

static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h)

返回一个指定接口的代理类实例，该接口可以将方法调用指派到指定的调用处理程序。

/\*

\* ClassLoader loader, 类加载器 固定写法 xx类.getClass().getClassLoader()

\* Class<?>[] interfaces, 这个类实现的所有接口 固定写法 xx类.getClass().getInterfaces()

\* InvocationHandler h 动态代理接口

\* \*/

想要创建一个代理类 必须实现InvocationHandler接口

### 创建代理类 实现InvocationHandler

public interface InvocationHandler

InvocationHandler 是代理实例的调用处理程序 实现的接口。

Object invoke(Object proxy, Method method, Object[] args)

### 在代理实例上处理方法调用并返回结果

通过使用这个类和接口就可以生成动态代理对象

JDK 提供的代理只能针对接口做代理 我们以后会用到更强大的代理cglib

**public** **class** agent {

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

// **TODO** Auto-generated method stub

Student stu=**new** Student();

Stu s=(Stu) Proxy.*newProxyInstance*(stu.getClass().getClassLoader(), stu.getClass().getInterfaces(), **new** Invocation(stu));

s.logIn();

s.logOut();

}

}

**interface** Stu

{

**public** **void** logIn();

**public** **void** logOut();

}

**class** Invocation **implements** InvocationHandler

{

**private** Student stu;

Invocation(Student stu)

{

**this**.stu=stu;

}

@Override

**public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {

// **TODO** Auto-generated method stub

System.***out***.println("authority verify");

method.invoke(stu, args);

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

**return** **null**;

}

}

**class** Student **implements** Stu

{

**public** **void** logIn()

{

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

}

**public** **void** logOut()

{

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

}

}