|  |
| --- |
| [公司名称] |
| 标题 |
| 副标题 |

|  |
| --- |
| 姚斯元  2018-10-30 |

目录

[一、 123 1](#_Toc3539499)

[1.1 1 1](#_Toc3539500)

[1.1.1 3 1](#_Toc3539501)

# Java学习笔记

## 学习导论

### 经典语录：

* 想得越多，思路越多，工资越高
* 经验是从不断的错误中总结出来的。
* 学习是痛苦的，但未来的美好的。
* 技术的学习不是一蹴而就的，是苦练出来的。
* 基础不牢，地动山摇。
* 这个世界唯一不变的就是改变，拥抱变化，坚持好奇，才是处世的正确态度。

### 一定要做笔记做笔记

善于做笔记！！！多做学习总结。一定要写博客，比如我自己的博客全都是我平时开发遇到的各种问题，利己利人。

CSDN、博客园、简书、自检博客站。github

### 善于利用时间

第二件事：善于利用零碎时间！！！

### 写代码：

1. 明确需求。我要做什么？
2. 分析思路。我要怎么做？
3. 确定步骤。每一个思路部分用到哪些语句，方法，和对象。
4. 代码实现。先用伪代码，再用具体的java语言代码把思路体现出来。

### 学习方法：

多想，多看，多练。

今日事今日毕。自学比在培训机构的管理差很多，但是更自由，也就更需要自律。

不推荐看书，效率太低了，书中所介绍的知识点太基础太全面了，要完完全全看完一本书那需要的时间代价太多了。更推崇看视频。现在的培训机构太多了，什么尚硅谷、尚学堂、慕课网，他们都他自己的培训视频放在网上免费让人们下载学习，当然了，这些视频肯定不是全套的，但是对于新手来说，能够快速入门足够了。

最后就是动手了，写demo！写demo！写demo！重要的事情说三遍，好的程序员那一定是用代码堆出来的，没有谁一个代码不写就能成为高级程序员成为架构师。

### 学习计划

合理的学习计划！！！我每天下班后，大概八点钟吃完饭收拾好就开始继续学习，一般会学习到十一点关电脑睡觉。周六日的时候上午我一般会看球赛，下午一个午休后就继续开始学习，一般也是学习到晚上十一点睡觉。这个习惯一直坚持到了现在。有很多人看后可能会说我每天都在学习，一点生活时间都没有，这样的生活还有什么意思。我想说的是：一个年轻人，别在最该奋斗的年纪虚度了时光！

### 其他

1. 如果你现在大三或者大四还在学校，尽早出来实习或者工作，在学校学习和在公司学习是完全不一样的，在学校做的那些乱七八糟的项目和公司真正的商业项目也是天壤之别的。
2. 坚持一个方向。不管做什么都要有一个坚定的方向，一旦决定了就不要轻易改变，要现在一个行业尽力吃的开，让自己能在社会上生存下去，然后在考虑学习更多的技术充实自己。
3. 逼自己一把。我有很多同学大概也是大四出来实习，不过现在过了快一年了，刚开始3K现在还是3K，他们每天浪费了太多的时间，别在最该奋斗的年纪虚度了时光！
4. 敢于突破，做事要有魄力。有的公司确实不利于自己发展，那就要果断跳槽，不能安于现状，怕这怕那。

### 学习新技术的四点：

1. 该技术是什么？
2. 该技术有什么特点(使用注意)：
3. 该技术怎么使用。demo
4. 该技术什么时候用？test。

# java概述

## java历史

### java特性：

1.简单  2.面向对象 3.分布性 4.可移植性  5.解释性  6.安全性  7.健壮性 8.多线程  9.高性能  10.动态

*刚开始是叫Oak（橡树的意思）->又觉得不好听，所以又换为java*

1995年创立   sun --> oracle

java之父：来自加拿大的  詹姆斯-高斯林

java组件：   JDk    <    jre          <   jvm

           开发工具包       运行环境          虚拟机

java跨平台

计算机本身是不能自动运行java代码的，是Ocrale发布了不同操作系统版本的jvm，之所以才可以运行的

源代码是一样的，由于orackle创建了不同系统的jvm

不同操作系统的java虚拟机jvm是不一样的，当源代码（扩展名为.java）编译时会有相对应操作系统的jvm对其进行编译，生成 扩展名为.class的文件（二进制文件），因此java可以进行跨平台操作

所谓的平台就是跨越不同的操作系统（平台=操作系统）

java的跨平台特征就是：一次编写，处处运行！

JDK（java开发者工具包）>jre（java运行环境）>jvm（java虚拟机）

如果只是为了运行java代码，可以只下载JRE，jre里面就包括java虚拟机jvm

但是一般写代码推荐下载JDK

### 转义字符

\n 换行

\t 制表位 相当于一个tab键，打4个空格

\' 转义单引号，变成可编译输出的单引号

\" 转义双引号，变成可编译输出的双引号

\\ 转义反斜杠，使反斜杠失去反斜杠的含义，变成可输出的反斜杠

### 注释（有3种）

//单行注释

/\**多行注释\*/*

/\*\* 调用该方法或类时，会显示该文本注释，在方法/类上面的一行里

### java后缀名

源文件  .java      编译后  .class

在cmd中运行java代码

首先需要下载JDK

下载安装好之后，配置环境变量  在系统变量的Path变量中追加JDK目录中的bin目录

编译： javac 类名.java

运行： java 类名

java程序结构：

方法：main方法是一种特殊的方法，是程序运行的起点，jvm会找到类中的main方法，并执行其中的代码，直至main中所有的代码运行结束

JDK版本问题

javase   j2se

javaee   j2ee

1995到2005年命名为j2se  1.0到1.5

2002 c#1.0

2005 c#2.0

由于2005年时c#版本是2.0，java程序员气不过觉得版本低，因此改为javase 5.0。。。

命名        不能有空格

规定：

类名和 文件名相同

区分大小写

不能包含特殊字符，除了$和\_

不能使用关键字和保留字

代码以；结束

### 规范：

名字的意义

驼峰命名法

代码风格

输出语句：   print  不换行    println   换行

数据类型       java是一种强类型语言  在java的领域中，基本数据类型变量存的是数据本身，引用类型变量存的是保存数据的空间地址

### 8种基本数据类型

只有基本数据类型才会在定义的时候开辟固定的内存空间

**4种整型    进制：16进制：0x开头   8进制 ：0开头**

**byte         1个字节         -128~127      范围的计算方式，以byte为例：1字节是8位  每位有2种可能，总共8位，就是2^8种可能=256   正负对半，因为0也算在内，因此正数范围比负数少1**

**short        2个字节         -32768~32767**

**int            4个字节         -2147483648~2147483647     （约+-21亿）所有整数默认为int类型**

**long         8个字节         最大      声明加大写L**

**2种浮点型**

**float        4个字节      声明+F/f**

**double    8个字节      所有的小数默认为double类型     俩个double类型的相减会出现精度损失  例：2.0-1.1=0.9999....**

**出现该精度损失的情况是因为计算机进行加减时人进行乘除是类似的，而且计算机是要先转换为二进制再进行运算**

**1种字符型**

**char        2个字节     0~65535    遵循Unicode码，因此范围全是正数，所有的字符都有对应的数字        必须单引号引起**

**几个重要的Ascal码：    a  97   b 98    A   65   B  66      0(数字0)  48    9  57     Asall码和Unicode码一样，只是Unicode码范围更大而已**

**1种布尔类型**

Boolean  1个字节   只有true和FALSE俩个值，在java中不支持0,1

### 进制之间的转化方法：

#### 十进制转二进制

     方法为：十进制数除2取余法，即十进制数除2，余数为权位上的数，得到的商值继续除2，依此步骤继续向下运算直到商为0为止。其结果是从最后一个余数读到第一个

#### 二进制转八进制

 方法为：3位二进制数按权展开相加得到1位八进制数。（注意事项，3位二进制转成八进制是从右到左开始转换，不足时补0）

#### 八进制转成二进制

方法为：八进制数通过除2取余法，得到二进制数，对每个八进制为3个二进制，不足时在最左边补零。

每一个八进制数除以2得的余数，其每个除下来的余数也是倒着数，不够3位的补0，也就    是1个八进制数得3个余数

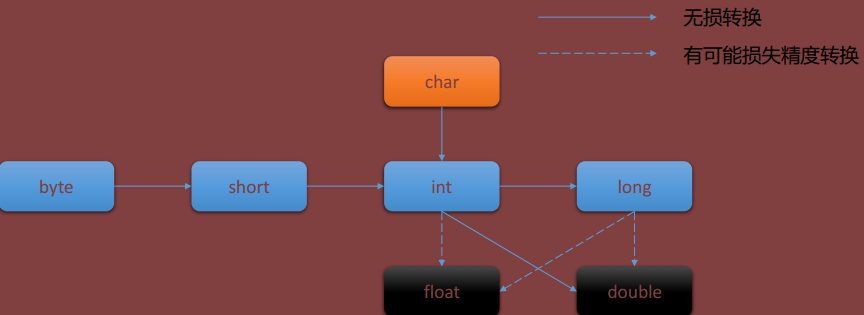
#### 二进制转十六进制

方法为：与二进制转八进制方法近似，八进制是取三合一，十六进制是取四合一。（注意事项，4位二进制转成十六进制是从右到左开始转换，不足时补0）。例：0001-》0\*2^3+0\*2^2+0\*2^1+1\*2^0-》1

#### 十六进制转二进制

 方法为：十六进制数通过除2取余法，得到二进制数，对每个十六进制为4个二进制，不足时在最左边补零。

### 基本数据类型之间的类型转化关系



强制类型转换：发生在当数据类型范围大的向数据类型范围小的变量进行赋值时，需要前面加上括号，写上要强制转换为什么类型（即：大杯子里的水往小杯子里倒，必定会有溢出，这就需要进行强制）

### java的自动装箱和拆箱：

在javac(java语言编程编译器)发现在一个需要Object ,Number,Integer的上下文里出现int类型的值，就会自动把这个值装箱为Integer，通过Integer.ValueOf(),反之，在一个需要int的上下文里出现了Integer类型的值的话，就会自动把资格值拆箱为int，通过Integer.intValue()

**N种引用类型 （没有固定的内存空间）  直接输出引用数据的变量时，默认输出内存地址**

**String  字符串   必须用双引号括起**

**定义电话或者身份证之类的号码，一般都是定义成String类型，只要不参与数学运算都是定义为String类型**

**数组**

**自定义的类**

# Java基础语法

### 变量

#### 使用：

数据类型  变量名 = 值；/  数据类型 变量名； 变量名 = 值；

变量名总是指向一个内存地址，并且和这个内存地址的对应关系不变，值则存储在内存地址对应的内存空间中

#### 变量命名：

变量名不能使用关键字和保留字

变量名首字母小写

变量名不能重复

使用之前必须先声明(变量声明后，必须使用赋值运算符=进行数值初始化，在java中不能使用没有初始化的变量)

#### 数值覆盖问题：

同一个变量只能声明一次，而且只有变量的值可以进行进行覆盖，而变量的数据类型是是不能进行覆盖

即：int a = 3; int a = 5 直接会报错        int a = 3; a = 5; 这样的结果为a=5

运算注意：

俩个操作数进行运算时，如果有一个操作数的数据类型范围大于另外一个，则运算结果的数据类型总是和数据类型范围大的操作数的数据类型相同

基本数据类型运算时，由于整数默认int，小数默认double，例：在  short  a = 1; a = a + 1;//这里必须进行强转      如果改成   a += 1; 则不需要进行强转

如果整型字面量的值在-128~127之间，就不会new新的Integer对象，而是直接引用常量池中的Integer对象

变量应用注意点：

当要求中国出现“连续”的字眼时，首先要想到先定义出错个数的变量，其次要想到不满足连续次数时要先清零然后去重新赋值

null和空字符串的本质区别

空字符串代表存在，指向一个空间，但值为“”  当对象值为null时，代表不存在，没指向一个空间，没有赋值

定义变量时 -------> 内存地址  ------------------> 内存空间

     变量名开辟空间，存放内容

都是整数参与的运算，结果还是整数，；俩数相除会将结果的小数抹除掉，而不是四舍五入

旗帜标志变量：flag(boolean)      根据旗帜来判断是否进行接下来的动作

判断字符串是否为null    num == null      判断字符串是否不为null    num != null

获取命令行中输入的值（即获取从键盘中输入的值）   java.util.Scanner input = new java.util.Scanner(System.in)

获取随机数(Random类  随机数生成器)     1.  Java.util.Random  rd = new java.util.Random();   rd.nextInt(100);    2.使用Math类  int res = (int)(Math.random()\*100)//这里的random是方法

将数字反转：要想到取余

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

System.*out*.println("请输入一个整数：");

int num=sc.nextInt();

int result=0;//存反转的数字

while(true) {

int n=num%10;//取出最低位上的数字

   //也可以直接打印出来System.out.println(n);不好，也可以用一个数组存起来，也不好，都因为翻转后的前后的那个0的问题

result=result\*10+n;//依次的反转存储得到反转的数字

num=num/10;//降位 将最低位舍弃

if(num==0) {

//当输入的数字除以10之后，即去除最低位之后的结果为0的时候，排除刚开始为0的结果（例：0000123去除前面的0）

   break;

 }

}

 System.*out*.println(result);

#### 交换 俩数

方法一：取一个空变量，然后哦进行调用赋值

方法二：

numA = numA + numB;

numB = numA - numB;

 numA = numA - numB;

方法三：

numA = numA ^ numB;

 numB = numB ^numA;

 numA = numA ^ numB;

  ("^"异或运算：即当俩个操作数的二进制表示相同，结果为0，不同为1)

#### 局部变量和实例变量：

在方法内部定义的叫变量叫做局部变量，在方法外部，类的内部定义的变量叫做实例变量或者属性，在没有赋值时，局部变量是没有默认值的，实例变量有默认值

当局部变量和实例(成员)变量命名相同时，方法内部需要使用使用实例变量时，应该使用this关键字。

实例变量在不赋值的情况下，总是有值的，在没有明确的赋值或没有调用setter方法进行赋值时，实例变量总是存在默认值

局部变量是在方法内部声明的变量，局部变量是没有默认值的，在没有赋值的情况下去使用局部变量，则会引发异常出现

### 常量：

* 定义：一旦获得初始值就不能改变的变量
* 声明：使用final关键字来表示，名称全部大写，Ctrl+shift+X 大写  Ctrl+shift+y 小写
* 使用：方法的参数可以是常量(被final修饰) 表示方法调用者传入的参数的值在方法内部不可更改
* final关键字作用：       //String和Math类(java预定义的类)都被final修饰
* final是一个修饰符，可以修饰变量，方法，类
* 被final修饰的类，该类不能被继承         被final修饰的方法，该方法不能被子类重写     被final修饰的变量称为常量，常量一旦获得初始值，就不能改变

### 方法

#### 格式：

访问修饰符 返回类型  方法名(参数列表) {

注：返回类型为void时代表不返回任何类型的数据

//方法体

访问修饰符：控制方法的访问级别

方法名：遵循驼峰命名法(规范)

}

返回类型：声明方法的返回类型

参数列表：方法的调用者可以通过参数列表可以将某些值或者变量传入到方法中

#### 方法调用：

对象.方法名();//类的外部

方法名(参数列表)  //类的内部调用其他方法，直接调用即可或者使用this关键字

#### 传参：

参数和局部变量相同，有自己的名字和数据类型，可以在方法内部使用。

如果一个方法需要参数，那么在调用的时候必须传东西给它，那个东西就是类型相同的变量或值，把变量传入方法时，只要类型相同即可

方法可以传入多个参数，多个参数之间使用逗号分割。传入参数的时候同样使用逗号分割。在调用时一定要按照正确的类型，顺序，数量来传入参数

#### 返回值：

如果方法声明有返回值，那么必须返回类型相同或者相互兼容(多态)的值，调用该方法时，可以使用类型相同或者相互兼容的变量接收到方法的返回值

Java中的返回值只能返回一个值，如果需要同时返回多个值，则可以使用数组或Java中的集合框架

Java中调用一个返回类型为非void的方法，可以不用理会返回值，那么就代表该次调用在乎是方法中的行为，而不是返回值

#### 构造方法：

构造方法是方法的一种特殊形式       唯一调用方式就是在创建类的对象时，通过new关键字

构造方法必须与类名相同，并且大小写都得一样         构造方法不能有返回值，连void也不能有               可以有参数列表

构造方法可调用普通方法，普通方法不可调用构造方法，构造方法可以通过this(参数列表)的形式来调用本类中的其他构造方法

构造方法在调用时必须在都必须是构造方法的第一行，无论是调用父类中的构造方法还是调用子类中的构造方法

当一个类没有显示声明一个构造方法时，jvm在运行时会自动创建一个参数列表为空，方法体为空的构造方法，当开发者声明了类的构造方法，jvm则不会在运行时自动创建

构造方法参数列表的应用：         如果参数是类必须的，那么可以在构造方法的参数列表中出现，可选的，那么可以考虑使用set方法

#### 方法签名：方法第一行的内容，除了方法体之外全是   访问修饰符+返回值类型+方法名+参数列表

#### 应用：

startup()   程序入口  统一在类里面书写，不在main方法中书写，直接调用该方法进行（实现低耦合） 实现多次运行，startup()方法最后调用自己即可，实现多次运行

system.exit();//系统自动退出

#### 方法重载和方法重写：

#### 方法重载(overload):

同一个类中，方法的访问修饰符。返回值类型，方法名相同，参数列表不同（顺序，类型，数量不同）         注：参数列表中不同类型的变换顺序才是重载，相同类型的顺序不同不是重载

一个方法有n种方法重载，要通过参数列表来调用

普通方法和构造方法都有

方法重写(override)：

子类可以根据自身需要，重写父类中的方法(子类中重新定义一个和父类中的方法签名一样的方法)

如果发生方法的重写，子类对象在调用该方法时，调用的是重写后的方法，而父类仍调用父类的方法

推荐在重写的方法开头加上@Override注解来防止打错单词，jdk5.0以上都支持

二者区别：

重写出现在继承关系中，子类可以根据自己的需要重写重写父类中的方法，当然父类中的private修饰和final修饰的方法不能被重写，重写后，子类对象调用该方法时就会执行子类中重写的方法，而不会再调用父类中的对应的方法。重载指在一个类中，方法名相同，但参数列表不同的多个方法，参数列表不同是指数量不同，类型不同和顺序不同

#### 赋值传递：

当俩个变量赋值相同时，会共同指向一片空间，不会新开辟空间，常量池（-127~127），

基本数据类型之间的赋值都是值传递，引用数据类型之间的赋值都是内存地址，对象的引用

                                    例：Cat c = new Cat();   Cat  x = c;//c将自己的内存地址，告诉x,这样x和c共同指向一片空间

对象数组：

引用数据类型的数组里存的值都是对象的内存地址

当给赋值时，相当于每个数组的值变成一个对象的引用，和对象共同指向一个内存地址

格式：Cat[]  cat = new Cat[];     cat[1] = new Cat()//或者写成  Cat c = new Cat();   cat[1] = c;

#### 不定项参数

格式：   （数据类型,,,数组名）        jdk5.0以上都支持

底层实现的原理就是相当于数组（对于方法内部来说，就是数组）

传参时可以传入相同类型的数据，可以传入任意多个（0~n个，包含0，n）

当不定项参数的方法和普通方法都满足时，优先使用普通方法

不定项参数只能在列表中存在一份，并且是最后是一个参数，    调用方法时可以忽略该参数

### 继承:

父类：基类或者超类（superclass）子类：subclass   继承的用途主要是代码的复用,当子类继承父类后,可以拥有父类中定义的属性和方法.在继承中,被继承的类成为父类，继承的类成为子类。

目的： 实现代码的复用  通过extends关键字实现

用途：在继承中，被继承的类成为父类，继承的类称为子类。继承的用途主要是代码的复用，当子类继承父类后，可以拥有父类中定义的属性和方法。

具有传递性（继承多层继承）和单根性（只能有一个直接父类）

子类也具有间接父类的属性和方法

子类可以调用父类的属性和方法，但父类不能调用子类的属性和方法  子类通过super.方法名的形式来调用父类的方法   也可直接通过父类方法名()的形式去使用父类中定义的方法,子类中没有的方法

创建子类对象的时必先创建父类对象，默认调用父类的无参的构造方法，如果父类中没有无参的构造方法，子类构造方法中必须指定调用父类的哪个构造方法来创建父类的对象，通过super(参数列表)的形式调用父类的构造方法

在Java中没有显式的使用extends关键字声明父类时，那么该类的父类为Java.lang.Object

总结：

在子类中调用其他构造方法使用 this(参数列表)的形式，调用父类的构造方法使用 super(参数列表)的形式，调用父类的普通方法使用 super.method()形式

### 多态：

多态在 java 中主要表现为父类指向子类对象，接口指向实现类对象。这样做的目的是降低类之间的耦合性。

作用：降低类型之间的耦合性，例如：方法采用父类作为参数，那么不仅可以传入父类对象，还可以传入所有子类对象；采用一个接口作为方法的参数，可以传入该接口所有实现类的对象。

俩种形式：

父类引用指向子类对象，反之则不行  Father father = new son();

该对象只能调用父类中定义的方法，如果该方法在子类中得到重写，那么就调用子类中特有的方法

当父类作为方法参数时，可以传入父类对象及其子类对象

当父类作为方法的返回值时，可以返回父类对象及其子类对象

当父类作为方法的返回值时，返回的是具体的子类对象，在调用的时候,可以使用父类接收（不能调用子类特有的方法）或者使用对应的子类对象接收（需要强转）,强转之后就可以调用子类特有的方法

父类引用指向子类对象，只能调用父类中特有的方法，若该方法在子类中得到重写，则调用重写后的方法，未重写就调用父类中原定义的方法。该对象不能调用子类中特有的方法

接口指向实现类对象  例：USB usb = new Mouse();

当接口指向实现类，只能调用接口中定义的方法，即使是该实现类中继承了多个接口，也只能单独调用某个接口（即先指向一个接口（接口类型的被实现类对象进行 ），然后通过变量名来调用接口中的方法（其实是调用实现类中重写的方法））

当接口作为方法的参数，可以传入接口所有实现类的对象，不能传接口，但可以在定义方法的时候在参数列表中出现

当接口作为方法的返回值，可以返回接口所有实现类对象  // 表面是接口类型，实质是实现类，在接收返回值的时候需要进行强制类型转换

高内聚低耦合

高内聚：处理特定功能越高越好

低耦合：和别的类依赖性越差越好

### 抽象：

子类去实现抽象类中的抽象方法

无论是抽象还是接口本质都是多态的实现手段

描述：

抽象类是使用 abstract 关键字修饰的类，这种类不能被实例化(想实例化就实例化子类对象)，并且可以有普通方法和抽象方法，抽象方法是指没有方法体的方法，需要在子类中重写得到实现。

抽象类存在的意义是作为父类存在，主要用于被继承，所以类不能用 final 修饰，当一个类继承自一个抽象类，必须要实现抽象类中定义的所有抽象方法，除非该类也是一个抽象类。

抽象类格式：public abstract class 类名{}

特点：不能被实例化(不能通过new来创建对象)，jvm会创建对象，但程序员不可以去new创建对象(即：创建子类对象时，首先jvm创建父类抽象类的对象)

#### 抽象方法

只有抽象类才有抽象方法，含有抽象方法的必须是抽象类

格式：public abstract class 方法名()      //抽象方法是没有方法体的 ，连大括号也不能有

注：访问修饰符只可以是public或者protected，如果定义为private方法对其他类不可见，就算子类也不行，而抽象类中的抽象方法是必须要子类重写后才能进行调用，因此，声明为私有的，编译是不会出错的，但是毫无意义

也不能使用static修饰，被static修饰的方法可以直接通过类名来访问该方法，抽象方法修饰完全没意义，因为没方法体      被statIC修饰之后，子类也不可以重写父类的方法

也不能被final修饰  被final修饰的方法不能被子类重写

存在原因：1.所有子类都存在该方法   2.所有子类对该方法的实现是不同的

### 包：

//bin文件下存放二进制文件.class

作用：防止命名冲突     分类管理实例变量对于类的实例（对象）来说，都拥有自己的的一份，每个对象可以有

不同的实例变量值。但是静态变量对于类的所有对象只有一份，所有对象的静态变

量的值是相同的

命名规范：

推荐使用公司或者组织的域名倒置（域名：[**baidu.com**](http://baidu.com/)）

使用package关键字来声明    该代码必须是类文件的第一行

包名全部小写，不能以.开头或者结尾，通过.进行分割路径

使用不在同一个包下的类方法：1.导入（Ctrl+shift+o）   2.完全限定名

所有的类会默认导入java.lang包里的所有接口和类

不推荐导入该包下的所有类，使用通配符\*

### 静态：

特点：

静态是随类一起的，而实例变量/普通方法是随对象一起的

定义情况：

静态使用static修饰，如果方法的运行不依赖属性（实例变量），那么我们就可以把方法定义成静态方法

静态方法/变量使用方法：

类名.方法名（参数列表）    类名.变量名

静态方法不依赖实例属性，也就不需要类的实例(对象)。使用这些方法只需要类本身就可以(静态方法子类构造直接使用类来调用)

可以使用类的对象调用静态方法/变量，但是不推荐

使用要点：

静态方法里都不能调用 属性和普通方法  静态方法可以调用静态方法/变量

非静态方法可以调用静态方法/变量

静态方法不能调用非静态的变量和方法

常量一般定义成静态    使用static修饰的实例变量称为静态变量   (实例变量是每个 对象都有一份，静态变量则是只有一份，所有对象共享一份）

静态变量：会在该类的任何对象创建之前完成初始化，会在该类的任何静态方法执行之前就初始化

静态变量和实例变量的区别：

实例变量对于类的实例（对象）来说，都拥有自己的的一份，每个对象可以有不同的实例变量值。但是静态变量对于类的所有对象只有一份，所有对象的静态变量的值是相同的

静态代码块：静态初始化程序(静态块)

作用：静态代码块是程序在第一次使用这个类(加载类时，会在其他类使用该类之前就执行)之前自动执行，并且只执行一次，适合作为类的初始化操作，可以调用静态方法和静态变量

格式：static {}   使用该格式定义的结构称之为静态块

静态方法、静态变量、静态代码快、属性、普通方法之间的相互调用

静态方法可以调用静态变量和静态方法，不可以调用属性和普通方法；静态代码快可以调用静态变量和静态方法，不能调用属性和普通方法；普通方法可以调用静态方法、静态变量、属性和普通方法。 静态代码快只有在第一次使用类的时候执行，并且只执行一次。

静态代码块和非静态代码块的区别：

都是在构造函数执行之前执行，在类中可以定义多个，一般是在代码块中对一些static变量进行赋值

静态代码块在非静态代码块之前运行（静态代码块->非静态代码块->构造方法）

静态代码块只在第一次new执行一次，之后不再运行，而非静态代码块在每new一次就执行一次，非静态代码块可以在普通方法中定义，而静态代码块就不行了

调用顺序  在执行创建子类对象时：new B()    A是B的父类       非静态代码块  格式为：{}

1.父类B的静态代码块   2.子类静态代码块  3.父类非静态代码块  4父类构造    5.子类非静态代码块   6.子类构造

调用顺序  在执行子类对象，调用有参的构造方法  new B("")

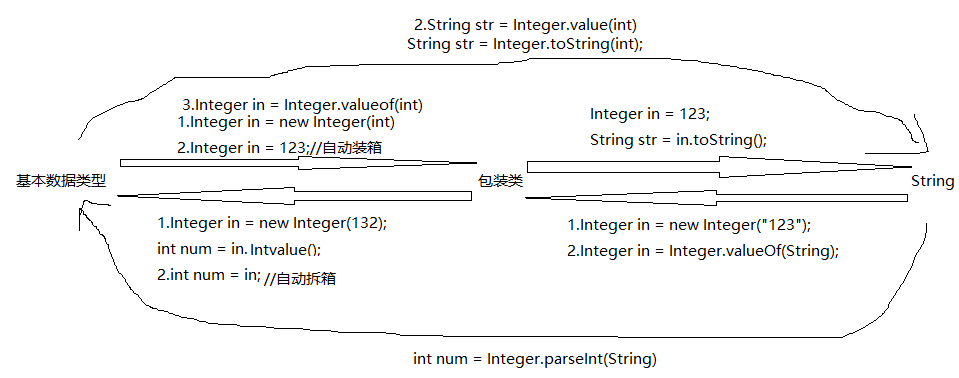
1.父类无参的构造   2.子类有参的构造  3.父类有参的构造

如果子类显式的调用父类的构造方法，即super(参数列表)，那么就不会执行父类无参的构造方法

如果没有显示的调用，那么在执行子类的有参的构造时，必须先执行父类无参的，然后执行子类有参的，然后执行父类有参的

### 包装类(封装类)

* + **作用：解决基本数据类型和String类型之间的相互转换问题**
  + **Byte,Double,Float,Long,Short,Integer的直接父类是java.lang.Number**
  + **int  -- > integer      char -->  character    其余的基本数据类型的包装类都是首字母小写变大写**
  + **包装类值的比较使用equals()**
  + **以integer为例：其余包装类转化类似  Double --> double  使用doubleValue()**

****

### Object:

所有java类的根类,基类，在这个类中定义的方法在所有对象中都有

用法：

在输出语句中输出一个对象时，默认会调用该对象的toString()，如果该类没有重写toString()，那么则默认调用Object的toString(),输出一个对象的字符串

自定义的类使用equals()进行比较的依然是内存地址，需要自定义比较哪些时，需要进行重写equals()

判断俩个对象是否相等： 使用instanceof关键字  //判断一个对象是否为一个特定类型的对象

计算当前时间距离时间戳的毫秒数：long starttime = System.carrentTimeMillis()   //时间戳是按1970年1月1日0时0分

### 运算符

* + **算数运算符：**
    - **+ - \* / %   -=  \*=  /=  %= 之类的运算符     运算的顺序遵循从左到右的顺序，然后根据运算的优先级去进行    小括号的优先级最高，+-的优先级最低**
    - **byte short 进行数学运算时，默认转为int           除法运算时，强转时，注意强转的是哪个值，小括号**
  + **关系运算符：**
    - **< > >= <= !=    其结果都是boolean类型的值**
  + **逻辑运算符：**
    - **&&   且  只有俩个条件都为true时，才为true             ||  或  只要其中一个为true，其结果就是为true    ！ 非  取反**
    - **短路情况：**
      * **&&短路 ：当条件一为FALSE时，另一个不再判断**
      * **||短路：当其中一个为true时，另一个不再判断**
    - **三元表达式：    条件？当条件为true时的值  ： 当条件为FALSE时的值**
  + **连字符： +**
    - **连接字符串相加的符号** **必须要有字符串的参与  “+”号左右，有字符串参与的结果总是字符串，没有时为算数运算符      运算都是从左到右运算的**
  + **a++与++a   a--与--a(道理与++是一致的)**
    - **无论哪个给其他变量赋值时，都是自增1      ++在前，先自加再赋值      ++在后，先赋值再自加**
  + **移位运算符：**
    - **有3种，分为为左移，右移，无符号右移 ，其操作的数据类型只有byte，short，char,int和long5种**
    - **语法格式： 左移：  需要移位的数字 << 移位的次数    按二进制形式把所有的数字向左移动对应的位数，高位移出(舍弃),低位的空位补零**
      * **数学意义：在数字没有溢出的前提下，对于正数和负数，左移一位都相当于乘以2的1次方，左移n位就相当于乘以2的n次方**
    - 右移：需要移位的数字 >>移位的次数   将二进制形式的所有的数字向右移动对应位移位数，低位移出(舍弃)，高位的空位补符号位，即正数补零，负数补1
      * 数学意义：右移一位相当于除以2，右移n位相当于除以2的n次方
    - 无符号右移：  >>>     无符号右移，忽略符号位，空位都以0补齐
      * value >>> num     --   num 指定要移位值value 移动的位数。
      * 无符号右移的规则只记住一点：忽略了符号位扩展，0补最高位  无符号右移运算符>>> 只是对32位和64位的值有意义

### 三大结构

* + - 顺序结构
    - 选择结构：

        if(条件){ 执行语句 }     //条件只能为Boolean类型的值或者变量，也可以是关系表达式   true执行，FALSE不执行     执行语句无大括号，紧跟其下一行的有效，其余无效

        多重选择：

 1.if(条件){}else if(条件){}else {}

 2.switch(变量){//变量可以是int，char,jdk7以上支持String   case子句中的常量不能重复

         case1:执行语句; break;    //break是为了防止case穿透，如果没有break，则会从该代码行直接执行到尾

         case2:执行语句;break;

         ...

         default:执行语句//default是默认，即总会执行的语句

    }

  俩种多重选择的优缺点：switch只能做等值判断，多重if可以做区间判断

* **循环结构**：//首先应该考虑：1.什么时候进入循环，2.什么时候推出循环   **一定要确保循环可以正常退出，否则将会形成死循环**

     while结构：//先判断再循环   初始条件不满足时，一次循环也不会执行

        格式： 1初始化变量

     while（条件：2.判断该变量是否满足这个条件）{

     3.循环体

     4.修改变量值

   }

    do...while结构：//先执行代码再判断（当条件为ture时才开始循环不是结束循环，当条件为FALSE时停止循环） 初始条件不满足时，会执行一次

        格式： 1.初始化变量

        do {

             2.循环体

            3.修改变量

             } while（条件）;    //不管是否满足，代码至少运行一次

    for循环 //括号里是以；隔开 语言简洁，是最常用的循环结构

    增强型for循环：//JDK5.0以上

         for(数据类型  变量：数组名){}    //其中变量为迭代的数组的值， 数据类型：存放数组的数据类型

         局限性： 1.只能取值，不能赋值    2.只能取得所有值    3.不用考虑数组下标越界      4.不能取代普通for循环

* + break与continue
    - break：结束循环，执行循环后的代码（退学）   在switch语句中可以作为case语句的结束
    - continue：结束本次循环继续循环（请假）
  + ==与equals
    - ==对于基本数据类型来说，用于比较俩个值是否相等，对于字符串来说比较的是内存地址，如果要比较俩个字符串的内容需要使用equals()方法
    - ==对于基本数据类型来说比较的是俩个值是否相等，对于引用数据类型来说比较的是俩个对象的内存地址是否相同，equals方法是Object类的方法，该方法默认使用==比较俩个对象内存地址，Object的子类可以通过重写equals方法来自定义比较规则，达到比较俩个对象的目的，例如：String重写equals方法之后，就可以通过该方法比较俩个字符串内容是否相同

### 数组

* + **数组中每个数据称为元素，每个元素在数组中有唯一的索引(下标)，索引从0开始，通过索引来操作数组中的元素，保存一组数据的数据结构，并且只能存放类型相同的数据，数组长度不可更改**
  + **定义：**
    - **int[] nums = new int[5];     int nums[] = new int[5]//c语言风格，java也支持    使用时：nums[0] = 12;进行赋值**
    - **int[] nums = { , , , , ,...}；**
    - **int[] nums = new int[]{ , , ,...};**
  + **数组中元素在赋值时，有默认值存在 ：** **在没有赋值时，对象的属性和数组才有默认值存在**
    - **byte   short  int  long 默认为0**
    - **float   double  默认为0.0**
    - **Boolean  默认为FALSE**
    - **String  默认为null（代表不存在）**
  + **通过使用 数组名.length来获取数组长度**
  + **应用：**
    - **循环赋值**
    - **循环取值（迭代数组）**
    - **求最大值    //即：遍历数组，然后进行比较，求出最大值**
    - **排序：  Leetcode.com**[**http://www.cnblogs.com/eniac12/p/5329396.html**](http://www.cnblogs.com/eniac12/p/5329396.html)
      * **冒泡排序**
        + **俩重for循环（一重控制轮数，一重控制每轮的比较次数）**

**int** **temp ; //定义临时变量**

**//进行冒泡排序**

**for** **(int** **i = 0; i < height.length - 1; i++) {** **//控制比较多少轮     可以从1开始**

**for** **(int** **j = 0; j < height.length-1-i; j++) {** **//控制每轮比较次数    按照数组下标从0开始**

**if(height[j] > height[j+1]){**

**//进行两数交换**

**temp = height[j];**

**height[j] = height[j+1];**

**height[j+1] = temp;**

**}**

**}**

**}**

* + - **插入排序**
    - **选择排序**
  + **随机数       java.util.Random  random = new java.util.Random();**

## 运算符号

## 语句

If switch do while while for

这些语句什么时候用？

1）、当判断固定个数的值的时候，可以使用if，也可以使用switch。但是建议使用switch，效率相对较高。

switch(变量){

case 值:要执行的语句;break;

…

default:要执行的语句;

}

工作原理：用小括号中的变量的值依次和case后面的值进行对比，和哪个case后面的值相同了

就执行哪个case后面的语句，如果没有相同的则执行default后面的语句；

细节：1）：break是可以省略的，如果省略了就一直执行到遇到break为止；

2）：switch 后面的小括号中的变量应该是byte,char,short,int四种类型中的一种；

3）：default可以写在switch结构中的任意位置；如果将default语句放在了第一行，则不管expression与case中的value是否匹配，程序会从default开始执行直到第一个break出现。

2）、当判断数据范围，获取判断运算结果boolean类型时，需要使用if。

3）、当某些语句需要执行很多次时，就用循环结构。

while和for可以进行互换。

区别在于：如果需要定义变量控制循环次数。建议使用for。因为for循环完毕，变量在内存中释放。

break:作用于switch ，和循环语句，用于跳出，或者称为结束。

break语句单独存在时，下面不要定义其他语句，因为执行不到，编译会失败。当循环嵌套时，break只跳出当前所在循环。要跳出嵌套中的外部循环，只要给循环起名字即可，这个名字称之为标号。

continue:只作用于循环结构，继续循环用的。

作用：结束本次循环，继续下次循环。该语句单独存在时，下面不可以定义语句，执行不到。

## 函 数：

为了提高代码的复用性，可以将其定义成一个单独的功能，该功能的体现就是java中的方法。方法就是体现之一。

java中的方法的定义格式：

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

执行语句；

return 返回值；

}

当方法没有具体的返回值时，返回的返回值类型用void关键字表示。

如果方法的返回值类型是void时，return语句可以省略不写的，系统会帮你自动加上。

return的作用：结束方法。结束功能。

特点：

定义函数可以将功能代码进行封装

便于对该功能进行复用

函数只有被调用才会被执行

函数的出现提高了代码的复用性

如何定义一个方法？

方法其实就是一个功能，定义方法就是实现功能，通过两个明确来完成：

1）、明确该功能的运算完的结果，其实是在明确这个方法的返回值类型。

2）、在实现该功能的过程中是否有未知内容参与了运算，其实就是在明确这个方法的参数列表(参数类型&参数个数)。

方法的作用：

1）、用于定义功能。

2）、用于封装代码提高代码的复用性。

注意：方法中只能调用方法，不能定义方法。

主方法：

1）、保证该类的独立运行。

2）、因为它是程序的入口。

3）、因为它在被jvm调用。

方法定义名称是为什么呢？

答：1）、为了对该功能进行标示，方便于调用。

2）、为了通过名称就可以明确方法的功能，为了增加代码的阅读性。

## 重载

概念：

在同一个类中，允许存在一个以上的同名方法。只要他们的参数个数或者参数类型，参数循序不同即可，

存在的原因：

屏蔽了一个对象的同一类方法由于参数不同所造成的差异。

特点：

与返回值类型无关，只看参数列表。

总结：

方法重载，方法名相同，但是参数列表不一致！（和方法覆写不一样，方法覆写的方法名和参数列表都必须一样）。注意方法重载和方法覆写的不同！

如何区分重载：

当方法同名时，只看参数列表。和返回值类型没关系。

## 接 口★★★★★

当一个类中的方法都是抽象的时候，java提供了另一种表示方式，叫接口。

1：是用关键字interface定义的。

2：接口中包含的成员，最常见的有全局常量、抽象方法。

注意：接口中的成员都有固定的修饰符。

成员变量：public static final

成员方法：public abstract

interface Inter{

public static final int x = 3;

public abstract void show();

}

3：接口中有抽象方法，说明接口不可以实例化。接口的子类必须实现了接口中所有的抽象方法后，该子类才可以实例化。否则，该子类还是一个抽象类。

4：类与类之间存在着继承关系，类与接口中间存在的是实现关系。

继承用extends ；实现用implements ；

5：接口和类不一样的地方，就是，接口可以被多实现，这就是多继承改良后的结果。java将多继承机制通过多现实来体现。

6：一个类在继承另一个类的同时，还可以实现多个接口。所以接口的出现避免了单继承的局限性。还可以将类进行功能的扩展。

7：其实java中是有多继承的。接口与接口之间存在着继承关系，接口可以多继承接口。

### 接口都用于设计上，设计上的特点：（可以理解主板上提供的接口）

1：接口是对外提供的规则。

2：接口是功能的扩展。

3：接口的出现降低了耦合性。

耦合(类与类之间的关系)

内聚(类完成功能的能力)

编程规范：低耦合，高内聚。

4：接口可以多实现。如：CPU和主板、笔记本的USB插口、插座

### 接口和抽象类的区别(问的比较多,概念性的问题)

关系

A:类与类的关系

是继承关系。类与类只能单继承，可以多重继承。

B:类和接口的关系

是实现关系。类可以多实现接口。

类在继承一个类的同时，可以实现多个接口。

C:接口和接口的关系

是继承关系。接口可以多继承接口。

### 抽象类和接口的区别：

1：抽象类只能被继承，而且只能单继承。

接口需要被实现，而且可以多实现。

2：抽象类中可以定义非抽象方法，子类可以直接继承使用。

接口中都有抽象方法，需要子类去实现。

3：抽象类使用的是  is a 关系。

接口使用的 like a 关系。

4：抽象类的成员修饰符可以自定义。

接口中的成员修饰符是固定的。全都是public的。

接口的定义格式

Interface 接口名{

成员变量

成员函数…

}

### 接口的作用：

1. 程序解耦。

2. 定义约束规范。

3. 扩展功能。

### 接口要注意的细节：

1. 接口中的成员变量都是常量，默认的修饰符是private static fianl

2. 接口中的方法全部都是抽象的方法。默认的修饰符：public abstract

3. 接口不能创建对象。

4. 接口是不没有构造函数的。

5. 非抽象类通过implements实现接口的时候，必须要把接口中所有方法全部实现。

### 接口与类之间的关系：

实现关系

注意：一个类可以实现多个接口。

接口与接口之间的关系是继承关系。

注意：一个接口是可以继承多个接口的。

关于抽象类中成员访问权限，其基本上继承了类的特性，但由于抽象类之所以为抽象类，是因为它是作为父类来使用的，是等待子类去实现的，而类中 private的权限只能是自个访问自个，所以在抽象类中方法为abstract时只有public,protected,default三种访问权限。

而接口中所有成员的属性都为public static final,也就是说接口中声明的变量都是常量，不能被继承。接口中所有方法的访问属性为public,所以实现接口中的方法必须标识为public,否则编译出错

### 抽象类与接口：

抽象类：一般用于描述一个体系单元，将一组共性内容进行抽取，特点：可以在类中定义抽象内容让子类实现，可以定义非抽象内容让子类直接使用。它里面定义的都是一些体系中的基本内容。

接口：一般用于定义对象的扩展功能，是在继承之外还需这个对象具备的一些功能。

#### 共性：

都是不断向上抽取的结果。

#### 区别：

A：抽象类只能被单继承

接口可以多实现,接口的出现避免了多继承的局限性。

B：抽象类中的数据特点：

成员变量：可以是变量，也可以是常量

成员方法：可以是抽象方法，也可以是非抽象方法

构造方法：有构造方法

接口中的数据特点：

成员变量：是常量。默认修饰 public static final

成员方法：都是抽象方法。都有默认修饰 public abstract

构造方法：没有构造方法

C：抽象类中定义的是继承体系中的共性功能。

接口中定义的是继承体系中的扩展功能。

D：抽象类被继承是"is a"关系:xx是yy的一种

接口被实现是"like a"关系:xx像yy的一种

接口中的成员修饰符是固定的。全都是public的。

在开发之前，先定义规则，A和B分别开发，A负责实现这个规则，B负责使用这个规则。至于A是如何对规则具体实现的，B是不需要知道的。这样这个接口的出现就降低了A和B直接耦合性。

使用implement关键字去实现抽象方法

## 描述：

接口中不能有普通方法和实例变量，接口中定义的都是抽象方法和常量，所以它比抽象类更“抽象” 。接口中定义的方法，主要用于被类实现，一个类可以实现多个接口，实现接口后，必须实现接口中定义的所有方法，除非是抽象类。接口之间可以继承，并可以多继承。

特殊的抽象类，只能有抽象方法，不允许普通的方法和属性，但可以有常量  同样不能实例化

接口中的变量默认是 final static的（即常量）方法默认是public的

接口使用Interface关键字来声明，例：public interface 类名{}          接口中的抽象方法的public abstract是固定写法，可以省略

一个类通过implement关键字来实现一个接口

一个类只能有一个直接父类，但可以同时实现多个接口

在java中类只能单继承，但接口可以实现多继承，接口可以继承接口

## this关键字：

* this关键字代表当前对象
* this.属性  操作当前对象的属性
* this.同类中的方法   调用当前对象的方法
* 封装对象的属性的时候，经常会使用this关键字：

### 作用：

1.访问被屏蔽的成员变量(当一个方法内，当局部变量(或方法的形参)和类的成员变量同名时，只有局部变量(或方法的形参)在方法内可见)

2.在构造方法中调用本类的构造方法

* 当方法中参数的变量名和实例变量的变量名相同时，则需要使用this.变量名来表示实例变量
* this关键字只能在方法内部使用，表示对“调用方法的那个对象”的引用。
* 构造器内可以用this调用一个构造器，并且必须将构造器调用置于最起始处。除构造器外，编译器禁止在其他任何方法中调用构造器。

### this和super

http://www.cnblogs.com/hasse/p/5023392.html

* 在Java中，this通常指当前对象，super则指父类的
* 遇到子类中的成员变量或者方法与父类中的成员变量或者方法同名时，因为子类中的成员变量或方法名优先级高，所以子类中的同名成员变量或方法就隐藏了超类的成员变量或者方法，但是我们想要使用超类中的这个成员变量或者方法时

### 优点：

表示当前对象

当前对象：当前正在调用实例成员的对象。谁调用了方法，谁就是当前对象

### 什么时候用：

在定义功能时，如果该功能内部使用到了调用该功能的对象，这时就用this来表示这个对象。

this 还可以用于构造方法间的调用。

调用格式：this(实际参数)；

this对象后面跟上 . 调用的是成员属性和成员方法(一般方法)；

this对象后面跟上 () 调用的是本类中的对应参数的构造方法。

### 注意：

用this调用构造方法，必须定义在构造方法的第一行。因为构造方法是用于初始化的，所以初始化动作一定要执行。否则编译失败。

this不能用在static修饰的方法里和static修饰的代码块里；

代码：

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

This关键字

PersonDeme3

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

this:看上去，是用于区分局部变量和成员变量同名情况。

this为什么可以解决这个问题？

this到底代表的是什么呢？

this：就代表本类的对象，到底代表哪一个呢?

this代表它所在函数所属对象的引用。

简单说：哪个对象在调用this所在的函数，this就代表哪个对象。

this的应用：当定义类中功能时，该函数内部要用到调用该函数的对象时，这是用this来表示这个对象。

但凡本类功能内部使用了本类对象，都用this表示。

\*/

class Person

{

private String name;

private int age;

Person(int age)

{

this.age = age;

}

Person(String name)

{

this.name = name;

}

Person(String name,int age)

{

this.name = name;

this.age = age;

}

public void speak()

{

System.out.println(

"name="+this.name+"..age="+this.age);

this.show();

}

public void show()

{

System.out.println(

this.name);

}

/\*

需求：给人定义一个用于比较年龄是否相同的功能，也就是是否是同龄人。

\*/

public boolean compare(Person p)

{

return this.age == p.age;

}

}

class PersonDemo3

{

public static void main(String [] args)

{

Person p1 = new Person(20);

Person p2 = new Person(25);

boolean b = p1.compare(p2);

System.out.println(b);

//Person p = new Person("lisi");

//Person p1 = new Person("zhangsan");

//p.speak();

//p1.speak();

//p.speak();

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

PersonDeme4

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

this语句：用于构造函数之间进行相互调用。

this语句只能定义在构造函数的第一样，因为初始化要先执行。

\*/

class Person

{

private String name;

private int aeg;

{

System.out.println("code run");

}

Person()

{

//this("haha");

System.out.println("Person run");

}

Person(String name)

{

//this();

this.name = name;

}

Person(String name,int age)

{

//this(name);

//this.name = name;

this.age = age;

}

}

class PersonDemo4

{

public static void main(String [] args)

{

new Person();

//Person p = new Person("lisi",30);

//Person p1 = new Person("lisi2",36);

}

}

## 访问修饰符：

可以用来修饰方法和属性

* public修饰的方法和属性            在任意位置都可以使用
* private修饰的方法和属性          仅在类的内部使用

 一般用于封装时修饰实体类的属性，并提供公有的getter, setter方法

* protected修饰的方法或属性      除了不能在不同包的非子类访问外，其他位置都可以访问
* 缺省修饰符修饰的方法或属性      不论是否为子类，同包可以访问，不同包不能访问

****

## Instanceof关键字

如果想用子类对象的特有方法，如何判断对象是哪个具体的子类类型呢？

可以可以通过一个关键字 instanceof ;//判断对象是否实现了指定的接口或继承了指定的类

格式：<对象 instanceof 类型> ，判断一个对象是否所属于指定的类型。

Student instanceof Person = true;//student继承了person类

\*\*\*\*\*多态在子父类中的成员上的体现的特点：

1、成员变量：在多态中，子父类成员变量同名。

在编译时期：参考的是引用型变量所属的类中是否有调用的成员。（编译时不产生对象，只检查语法错误）

运行时期：也是参考引用型变量所属的类中是否有调用的成员。

简单一句话：无论编译和运行，成员变量参考的都是引用变量所属的类中的成员变量。

再说的更容易记忆一些：成员变量 --- 编译运行都看 = 左边。

2、成员方法。

编译时期：参考引用型变量所属的类中是否有调用的方法。

运行事情：参考的是对象所属的类中是否有调用的方法。

为什么是这样的呢？因为在子父类中，对于一模一样的成员方法，有一个特性：覆盖。

简单一句：成员方法，编译看引用型变量所属的类，运行看对象所属的类。

更简单：成员方法 --- 编译看 = 左边，运行看 = 右边。

3、静态方法。

编译时期：参考的是引用型变量所属的类中是否有调用的成员。

运行时期：也是参考引用型变量所属的类中是否有调用的成员。

为什么是这样的呢？因为静态方法，其实不所属于对象，而是所属于该方法所在的类。

调用静态的方法引用是哪个类的引用调用的就是哪个类中的静态方法。

简单说：静态方法 --- 编译运行都看 = 左边。

多态中对象调用成员的特点

Fu f = new Zi();

A:成员变量

编译看左边，运行看左边

B:成员方法

编译看左边，运行看右边

C:静态方法

编译看左边，运行看左边

多态的思想

指挥同一批对象做事情。举例：带兵打仗，下课等。

多态：父类的引用类型变量指向了子类的对象，或者是接口的引用类型变量指向了接口实现类的对象。

多态要注意的事项：

1、 多态情况下，如果子父类存在同名的成员变量时，访问的是父类的成员变量。

2、 多态情况下，如果子父类存同名的非静态函数时，访问的是子类的成员变量。

3、 多态情况下，如果子父类存在同名的静态函数时，访问的是父类的成员函数。

4、 多态情况下，不能访问子类特有的成员。

多态的应用：

1、 多态用于形参类型的时候，可以接受更多类型的数据。

2、 多态用于返回值类型的时候，可以返回更多类型的数据。

Object类

------java.lang.Object

Object：是所有类的根类，超类。java中提供的类以及我们自定义的类都直接或者间接的继承自Object类。

具体方法：

1、boolean equals(Object obj)：用于比较两个对象是否相等，其实内部比较的就是两个对象地址。如果根据 equals(Object) 方法，两个对象是相等的，那么对这两个对象中的每个对象调用 hashCode 方法都必须生成相同的整数结果;

而根据对象的属性不同，判断对象是否相同的具体内容也不一样。所以在定义类时，一般都会复写equals方法，建立本类特有的判断对象是否相同的依据。

public boolean equals(Object obj){

if(!(obj instanceof Person))

return false;

Person p = (Person)obj;

return this.age == p.age;

}

### ==和equals的用法：

A:==怎么用？

\*\*可以用于比较基本数据类型，比较的就是基本数据类型的值是否相等。

\*\*可以用于比较引用数据类型，比较的是对象的地址值是否相等。

B:equals怎么用？

equals只能用于比较引用数据类型的。

\*\*Object提供的equals是用于比较对象地址值是否相同。

\*\*自定义类中，如果重写了equals方法，那么就是按照你自己的需求来比较的。

2，String toString()：将对象变成字符串；默认返回的格式：类名@哈希值 = getClass().getName() + '@' + Integer.toHexString(hashCode())

为了对象对应的字符串内容有意义，可以通过复写，建立该类对象自己特有的字符串表现形式。

public String toString(){

return "person : "+age;

}

3，Class getClass()：获取任意对象运行时的所属字节码文件对象。

4，int hashCode()：返回该对象的哈希码值。支持此方法是为了提高哈希表的性能。将该对象的内部地址转换成一个整数来实现的。

通常equals，toString，hashCode，在应用中都会被复写，建立具体对象的特有的内容。

## 内部类☆☆

(1)把一个类定义在某个类中的，这个类就被称为内部类，内置类，嵌套类。

(2)访问特点：

A:内部类可以直接访问外部类中的成员，因为内部类持有外部类的引用，

格式为：外部类名.this

B:外部类要想访问内部类的成员，必须创建对象访问。

(3)内部类的访问格式：

A:当内部类定义在外部类的成员位置，而且非私有，则可以在其他外部类中直接建立内部类对象

格式：外部类名.内部类名 变量名 = new 外部类对象.内部类对象

如：Outer.Inner in = new Outer().new Inner()

B:当内部类在外部类成员位置，且被static修饰时

\*\*外部其他类可直接访问静态内部类的非静态成员

格式：new 外部类名.内部类名().内部类成员

如：new Outer.Inner().function();

\*\*外部其他类可直接访问静态内部类的静态成员

格式：new 外部类名.内部类名.内部类成员

如：new Outer.Inner.function();

(4)什么使用时候内部类呢？

假如有A类和B类，A类想直接访问B类的成员，B类访问A类成员的时候，

需要创建A类对象进行访问，这个时候，就可以把A类定义为B类的内部类。

(5)内部类的位置

A:成员位置

\*\*可以被private修饰(Body，Heart)

\*\*可以被static修饰。(它访问的外部类的成员必须是静态的)

B:局部位置

\*\*可以直接访问外部类中的成员，因为还持有外部类的持用

也可以直接访问局部成员，但是局部成员要用final修饰。

注意：局部内部类不能用private和static修饰

(6)通过class文件我们就可以区分是否带有内部类，以及内部类的位置

Outer$Inner:成员内部类

Outer$1Inner:局部内部类

如果A类需要直接访问B类中的成员，而B类又需要建立A类的对象。这时,为了方便设计和访问，直接将A类定义在B类中。就可以了。A类就称为内部类。

内部类可以直接访问外部类中的成员。而外部类想要访问内部类，必须要建立内部类的对象。

class Outer{

int num = 4;

class Inner {

void show(){

System.out.println("inner show run "+num);

}

}

public void method(){

Inner in = new Inner();//创建内部类的对象。

in.show();//调用内部类的方法。 //内部类直接访问外部类成员，用自己的实例对象；

}//外部类访问内部类要定义内部类的对象；

}

当内部类定义在外部类中的成员位置上，可以使用一些成员修饰符修饰 private、static。

1：默认修饰符。

直接访问内部类格式：外部类名.内部类名 变量名 = 外部类对象.内部类对象;

Outer.Inner in = new Outer.new Inner();//这种形式很少用。

但是这种应用不多见，因为内部类之所以定义在内部就是为了封装。想要获取内部类对象通常都通过外部类的方法来获取。这样可以对内部类对象进行控制。

2：私有修饰符。

通常内部类被封装，都会被私有化，因为封装性不让其他程序直接访问。

3：静态修饰符。

如果内部类被静态修饰，相当于外部类，会出现访问局限性，只能访问外部类中的静态成员。

注意；如果内部类中定义了静态成员，那么该内部类必须是静态的。

内部类编译后的文件名为：“外部类名$内部类名.java”；

为什么内部类可以直接访问外部类中的成员呢？

那是因为内部中都持有一个外部类的引用。这个是引用是 外部类名.this

内部类可以定义在外部类中的成员位置上，也可以定义在外部类中的局部位置上。

当内部类被定义在局部位置上，只能访问局部中被final修饰的局部变量。

### 匿名内部类（对象）：☆☆

没有名字的内部类。就是内部类的简化形式。一般只用一次就可以用这种形式。匿名内部类其实就是一个匿名子类对象。

前提：内部类必须继承一个类或者实现接口。(注意不要弄混匿名内部类的前提和多态的前提)

匿名内部类的格式：new 父类名&接口名(){ 定义子类成员或者覆盖父类方法 }.方法。

好处和弊端：

好处：简化代码书写

弊端：

不能直接调用自己的特有方法

不能执行强转换动作

如果该类里面方法较多，不允许使用匿名内部类

When：

当方法的参数是接口类型引用时，如果接口中的方法不超过3个。可以通过匿名内部类来完成参数的传递。

其实就是在创建匿名内部类时，该类中的封装的方法不要过多，最好两个或者两个以内。

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

//面试

//1

new Object(){

void show(){

System.out.println("show run");

}

}.show();//写法和编译都没问题

//2

Object obj = new Object(){

void show(){

System.out.println("show run");

}

};

obj.show();//写法正确，编译会报错

1和2的写法正确吗？有区别吗？说出原因。

写法是正确，1和2都是在通过匿名内部类建立一个Object类的子类对象。

区别：

第一个可是编译通过，并运行。

第二个编译失败，因为匿名内部类是一个子类对象，当用Object的obj引用指向时，就被提升为了Object类型，而编译时检查Object类中是否有show方法，所以编译失败。

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

class InnerClassDemo6 {

+（static）class Inner{

void show(){}

}

public void method(){

this.new Inner().show();//可以

}

public static void main(String[] args) {//static不允许this

This.new Inner().show();//错误，Inner类需要定义成static

}

}

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

interface Inter{

void show();

}

class Outer{//通过匿名内部类补足Outer类中的代码。

public static Inter method(){//返回Inter类型的变量；

return new Inter(){

public void show(){}

};

}

}

class InnerClassDemo7 {

public static void main(String[] args) {

Outer.method().show();

/\*

Outer.method():意思是：Outer中有一个名称为method的方法，而且这个方法是静态的。

Outer.method().show():当Outer类调用静态的method方法运算结束后的结果又调用了show方法，意味着：method()方法运算完一个是对象，而且这个对象是Inter类型的。

\*/

function (new Inter(){

public void show(){}

}); //匿名内部类作为方法的参数进行传递。

}

public static void function(Inter in){

in.show();

}

}

#### 匿名内部类：

没有名字的内部类。就是内部类的简化形式。一般只用一次就可以用

这种形式。匿名内部类其实就是一个匿名子类对象。想要定义匿名内部类：需要前提，内部类必须继承一个类或者实现接口。

匿名内部类的格式：new 父类名&接口名(){ 定义子类成员或者覆盖父类方法 }.方法。

 匿名内部类的使用场景：

当函数的参数是接口类型引用时，如果接口中的方法不超过3个。可以通过匿名内部类来完成参数的传递。

其实就是在创建匿名内部类时，该类中的封装的方法不要过多，最好两个或者两个以内。

### 内部类访问规则

将一个类定义在另一个类的里面,对里面那个类就称为内部类(内置类,嵌套类).

访问特点:内部类可以直接访问外部类中的成员,包括私有成员

之所以可以直接访问外部类中的成员,是因为内部类中持有了一个外部类的引用

格式: 外部类名.this

而外部类要访问内部类中的成员必须要建立内部类的对象

创建内部对象时写类名

Outer.Inner in = new Outre().new Inner();

method

private 内部类 私有内部类

外部类不能私有

## 包

定义包用package关键字。

1：对类文件进行分类管理。

2：给类文件提供多层名称空间。

如果生成的包不在当前目录下，需要最好执行classpath，将包所在父目录定义到classpath变量中即可。

一般在定义包名时，因为包的出现是为了区分重名的类。所以包名要尽量唯一。怎么保证唯一性呢？可以使用url域名来进行包名称的定义。

package pack;//定义了一个包，名称为pack。 注意：包名的写法规范：所有字母都小写。

//package cn.itcast.pack.demo;

类的全名称是 包名.类名

编译命令：javac –d 位置（.当前路径） java源文件 （就可以自动生成包）

包是一种封装形式，用于封装类，想要被包以外的程序访问，该类必须public；

类中的成员，如果被包以外访问，也必须public；

包与包之间访问可以使用的权限有两种：

1：public

2：protected：只能是不同包中的子类可以使用的权限。

总结java中的四种权限：

范围 publicprotecteddefaultprivate

同一个类中 ok ok okok

同一包中 ok ok ok

子类 ok

不同包中 ok

类 构造方法 成员变量 成员方法

privateOKOKOK

默认Ok OkOkOK

protected OKOKOk

public Ok OkOKOK

static OKOk

finalOk OKOK

abstract Ok OK

一般格式：

成员变量：

权限修饰符+static/final+数据类型+成员变量名

public static final int NUM = 10;

成员方法：

权限修饰符+static/final/abstract+返回类型+方法名

Import - 导入：

类名称变长，写起来很麻烦。为了简化，使用了一个关键字：import，可以使用这个关键字导入指定包中的类。记住：实际开发时,到的哪个类就导入哪个类，不建议使用\*.

import packa.\*;//这个仅仅是导入了packa当前目录下的所有的类。不包含子包。

import packa.abc.\*;//导入了packa包中的子包abc下的当前的所有类。

如果导入的两个包中存在着相同名称的类。这时如果用到该类，必须在代码中指定包名。

常见的软件包:

java.lang : language java的核心包，Object System String Throwable jdk1.2版本后，该包中的类自动被导入。

java.awt : 定义的都是用于java图形界面开发的对象。

javax.swing: 提供所有的windows桌面应用程序包括的控件,比如：Frame , Dialog, Table, List 等等,就是java的图形界面库。

java.net : 用于java网络编程方面的对象都在该包中。

java.io : input output 用于操作设备上数据的对象都在该包中。比如：读取硬盘数据，往硬盘写入数据。

java.util : java的工具包，时间对象，集合框架。

java.applet: application+let 客户端java小程序。server+let --> servlet 服务端java小程序。

jar ：java的压缩包，主要用于存储类文件，或者配置文件等。

命令格式：jar –cf 包名.jar 包目录

解压缩：jar –xvf 包名.jar

将jar包目录列表重定向到一个文件中：jar –tf 包名.jar >c:\1.txt

## MVC

**1.将一个应用程序的代码分成几个部分：视图层、控制层、模型层；**

**2.视图层(View)：用于从用户那里获取数据，以及向用户展示数据；**

**3.控制层(Controller)：用于接收视图层的数据，并进行验证。并将数据向其他层传递；**

**|--持久层：用于与数据库交互；**

**4.模型层(Model):层与层之间要进行数据传递(调用方法)，传递的就是JavaBean的对象；**

## 私有

private int age;//私有的访问权限最低，只有在本类中的访问有效。

注意：私有仅仅是封装的一种体现形式而已。

私有的成员：其他类不能直接创建对象访问，所以只有通过本类对外提供具体的访问方式来完成对私有的访问，可以通过对外提供方法的形式对其进行访问。

好处：

可以在方法中加入逻辑判断等操作，对数据进行判断等操作。

总结：

开发时，记住，属性是用于存储数据的，直接被访问，容易出现安全隐患，所以，类中的属性通常被私有化，并对外提供公共的访问方法。

这个方法一般有两个，规范写法：对于属性 xxx，可以使用setXXX(),getXXX()对其进行操作。

## 类中怎么没有定义主方法呢？

注意：主方法的存在，仅为该类是否需要独立运行，如果不需要，主方法是不用定义的。

主方法的解释：保证所在类的独立运行，是程序的入口，被jvm调用。

成员变量和局部变量的区别：

1：成员变量直接定义在类中。

局部变量定义在方法中，参数上，语句中。

2：成员变量在这个类中有效。

局部变量只在自己所属的大括号内有效，大括号结束，局部变量失去作用域。

3：成员变量存在于堆内存中，随着对象的产生而存在，消失而消失。

局部变量存在于栈内存中，随着所属区域的运行而存在，结束而释放。

## 匿名对象

(1)匿名对象就是没有名字的对象。是对象的一种简写形式。

(2)应用场景

A:只调用一次类中的方法。

B:可以作为实际参数在方法传递中使用

### 匿名对象使用场景：

1：当对方法只进行一次调用的时候，可以使用匿名对象。

2：当对象对成员进行多次调用时，不能使用匿名对象。必须给对象起名字。

在类中定义其实都称之为成员。成员有两种：

1：成员变量：其实对应的就是事物的属性。

2：成员方法：其实对应的就是事物的行为。

所以，其实定义类，就是在定义成员变量和成员方法。但是在定义前，必须先要对事物进行属性和行为的分析，才可以用代码来体现。

## 构造函数

用来构造类的实例（每一个类都默认有一个无参的构造方法，要使用new调用）

### 注意：

默认构造函数特点

多个构造函数是重载

### 小细节

一个类中没有定义构造函数，就默认给该类加入一空参数构造函数，当在类中自定义了构造函数，默认的构造函数就没有了。

### 构造和普通函数区别：

构造函数是在对象一建立就给对象初始化，而一般方法是对象调用才执行。

一个对象建立，构造函数只执行一次。一般方法可以被对象调用多次。

什么时候定义：

当分享食物时，事物需要具备一些特性和行为，那么将内容定义在构造函数中。

## 构造方法。

用于给对象进行初始化，是给与之对应的对象进行初始化，它具有针对性，方法中的一种。

### 发现子类构造方法运行时，先运行了父类的构造方法。为什么呢?

原因：

子类的所有构造方法中的第一行，其实都有一条隐身的语句super();

super(): 表示父类的构造方法，并会调用于参数相对应的父类中的构造方法。而super():是在调用父类中空参数的构造方法。

为什么子类对象初始化时，都需要调用父类中的方法？(为什么要在子类构造方法的第一行加入这个super()?)

因为子类继承父类，会继承到父类中的数据，所以必须要看父类是如何对自己的数据进行初始化的。所以子类在进行对象初始化时，先调用父类的构造方法，这就是子类的实例化过程。

### 注意：

子类中所有的构造方法都会默认访问父类中的空参数的构造方法，因为每一个子类构造内第一行都有默认的语句super();

如果父类中没有空参数的构造方法，那么子类的构造方法内，必须通过super语句指定要访问的父类中的构造方法。

如果子类构造方法中用this来指定调用子类自己的构造方法，那么被调用的构造方法也一样会访问父类中的构造方法。

### 特点：

1：该方法的名称和所在类的名称相同。

2：不需要定义返回值类型。

3：没有具体的返回值。

记住：所有对象创建时，都需要初始化才可以使用。

注意事项：一个类在定义时，如果没有定义过构造方法，那么该类中会自动生成一个空参数的构造方法，为了方便该类创建对象，完成初始化。如果在类中自定义了构造方法，那么默认的构造方法就没有了。

一个类中，可以有多个构造方法，因为它们的方法名称都相同，所以只能通过参数列表来区分。所以，一个类中如果出现多个构造方法。它们的存在是以重载体现的。

### 构造方法和一般方法有什么区别呢？

A:格式区别

构造方法和类名相同，并且没有返回类型，也没有返回值。

普通成员方法可以任意起名，必须有返回类型，可以没有返回值。

B:作用区别

构造方法用于创建对象，并进行初始化值。

普通成员方法是用于完成特定功能的。

C:调用区别

构造方法是在创建对象时被调用的，一个对象建立，只调用一次相应构造函数

普通成员方法是由创建好的对象调用，可以调用多次

### 什么时候使用构造方法呢？

分析事物时，发现具体事物一出现，就具备了一些特征，那就将这些特征定义到构造方法内。

## 构造代码块

1)作用：给对象进行初始化，对象一建立就执行，而且优先于构造函数执行

(2)构造代码块和构造函数的区别：

构造代码块是给所有不同对象的共性进行统一初始化

构造函数是给对应的对象进行初始化

构造代码块和构造方法有什么区别？

构造代码块：是给所有的对象进行初始化，也就是说，所有的对象都会调用一个代码块。只要对象一建立。就会调用这个代码块。

### Person p = new Person();创建一个对象都在内存中做了什么事情？

1：先将硬盘上指定位置的Person.class文件加载进内存。

2：执行main方法时，在栈内存中开辟了main方法的空间(压栈-进栈)，然后在main方法的栈区分配了一个变量p。

3：在堆内存中开辟一个实体空间，分配了一个内存首地址值。new

4：在该实体空间中进行属性的空间分配，并进行了默认初始化。

5：对空间中的属性进行显示初始化。

6：进行实体的构造代码块初始化。

7：调用该实体对应的构造方法，进行构造方法初始化。（）

8：将首地址赋值给p ，p变量就引用了该实体。(指向了该对象)

## 关键字private：

封装在代码中的体现

(1)私有的意思，权限修饰符

(2)用来修饰成员变量和成员函数

(3)用private修饰的成员只在本类中有效

(4)私有是封装的一种体现

## Static★★★

### 静态修饰符，用于修饰成员(成员变量和成员方法)。

Feature：

随着类的加载而加载

优先于对象存在

对所有对象共享

可以被类名直接调用

Abuse：

生命周期过长，随着类的消失而消失

Attention：

1，有些数据是对象特有的数据，是不可以被静态修饰的。因为那样的话，特有数据会变成对象的共享数据。这样对事物的描述就出了问题。所以，在定义静态时，必须要明确，这个数据是否是被对象所共享的。

2，访问权限（静态方法只能访问静态成员，不可以访问非静态成员。）

(这句话是针对同一个类环境下的，比如说，一个类有多个成员（属性，方法，字段），静态方法A，那么可以访问同类名下其他静态成员，你如果访问非静态成员就不行)

因为静态方法加载时，优先于对象存在，所以没有办法访问对象中的成员。

3，静态方法中不能使用this，super关键字。

因为this代表对象，而静态在时，有可能没有对象，所以this无法使用。

4.主方法是静态的。

public static void main(String[] args)

public:公共的意思，是最大权限修饰符。

static:由于jvm调用main方法的时候，没有创建对象。

只能通过类名调用。所以，main必须用static修饰。

void:由于main方法是被jvm调用，不需要返回值。用void修饰。

main:main是主要的意思，所以jvm采用了这个名字。是程序的入口。

String[]:字符串数组

args:数组名

在运行的时候，通过java命令给args数组赋值。

格式：java MainTest hello world itcast

5.静态的生命周期比较长，所以一般不推荐使用。

用static修饰成员表示它属于这个类共有，而不是属于该类的单个实例。

static 修饰的字段 = 类字段，

static 修饰的方法 = 类方法，

没使用static修饰字段和方法，成员属于类的单个实例。不属于类。

没有static 修饰的字段 = 实例字段，

没有static 修饰的方法 = 实例方法，

总结：static修饰的字段和方法，既可以通过类调用，也可以使用实例调用，

没static修饰的字段和方法，只能通过实例来调用。（建议使用：类名来调用；其实在底层，对象调用类成员，也会转换类名调用）

### 静态变量和成员变量的区别

A：调用方式

静态变量也称为类变量，可以直接通过类名调用。也可以通过对象名调用。

这个变量属于类。

成员变量也称为实例变量，只能通过对象名调用。这个变量属于对象。

所以，成员变量可以称为对象的特有数据，静态变量称为对象的共享数据。

B：存储位置

静态变量存储在方法区长中的静态区。

成员变量存储在堆内存。

C：生命周期

静态变量随着类的加载而存在，随着类的消失而消失。生命周期长。

成员变量随着对象的创建而存在，随着对象的消失而消失。

D：与对象的相关性

静态变量是所有对象共享的数据。

成员变量是每个对象所特有的数据。

### 什么时候定义静态成员呢？或者说：定义成员时，到底需不需要被静态修饰呢？

成员分两种：

1、成员变量。（数据共享时静态化）

该成员变量的数据是否是所有对象都一样：

如果是，那么该变量需要被静态修饰，因为是共享的数据。

如果不是，那么就说这是对象的特有数据，要存储到对象中。

2、成员方法。（方法中没有调用特有数据时就定义成静态）

如果判断成员方法是否需要被静态修饰呢？

只要参考，该方法内是否访问了对象中的特有数据：

如果有访问特有数据，那方法不能被静态修饰。

如果没有访问过特有数据，那么这个方法需要被静态修饰。

### 静态代码块

A:它只执行一次，它比main还先执行。

B:执行顺序

静态代码块--构造代码块--构造方法

静态代码块：就是一个有静态关键字标示的一个代码块区域。定义在类中。

作用：可以完成类的初始化。静态代码块随着类的加载而执行，而且只执行一次（new 多个对象就只执行一次）。如果和主方法在同一类中，优先于主方法执行。

Public：访问权限最大。

static：不需要对象，直接类名即可。

void：主方法没有返回值。

Main：主方法特定的名称。

(String[] args)：主方法的参数，是一个字符串数组类型的参数，jvm调用main方法时，传递的实际参数是 new String[0]。

jvm默认传递的是长度为0的字符串数组，我们在运行该类时，也可以指定具体的参数进行传递。可以在控制台，运行该类时，在后面加入参数。参数之间通过空格隔开。jvm会自动将这些字符串参数作为args数组中的元素，进行存储。

静态代码块、构造代码块、构造方法同时存在时的执行顺序：

静态代码块  构造代码块  构造方法；

## 子父类出现后，类中的成员都有了哪些特点：

1：成员变量。

当子父类中出现一样的属性时，子类类型的对象，调用该属性，值是子类的属性值。

如果想要调用父类中的属性值，需要使用一个关键字：super

This：代表是本类类型的对象引用。

Super：代表是子类所属的父类中的内存空间引用。

注意：子父类中通常是不会出现同名成员变量的，因为父类中只要定义了，子类就不用在定义了，直接继承过来用就可以了。

2：成员方法。

当子父类中出现了一模一样的方法时，建立子类对象会运行子类中的方法。好像父类中的方法被覆盖掉一样。所以这种情况，是方法的另一个特性：覆盖(复写，重写)

## 重写和重载的区别？

重载：在同一类中。方法名相同，参数列表不同。重载可以改变返回类型。

重写：在不同类中(子父类中)。

方法声明相同(返回类型，方法名，参数列表均相同)。

什么时候使用覆盖呢？当一个类的功能内容需要修改时，可以通过覆盖来实现。

## 问题：super()和this()是否可以同时出现的构造方法中。

两个语句只能有一个定义在第一行，所以只能出现其中一个。

super()或者this():为什么一定要定义在第一行？

因为super()或者this()都是调用构造方法，构造方法用于初始化，所以初始化的动作要先完成。

When：

当类与类之间存在着所属关系时，才具备了继承的前提。a是b中的一种。a继承b。狼是犬科中的一种。

英文书中，所属关系：" is a "

注意：不要仅仅为了获取其他类中的已有成员进行继承。

所以判断所属关系，可以简单看，如果继承后，被继承的类中的功能，都可以被该子类所具备，那么继承成立。如果不是，不可以继承。

Attention：

1：子类覆盖父类时，必须要保证，子类方法的权限必须大于等于父类方法权限可以实现继承。否则，编译失败。

2：覆盖时，要么都静态，要么都不静态。 (静态只能覆盖静态，或者被静态覆盖)

继承的一个弊端：打破了封装性。对于一些类，或者类中功能，是需要被继承，或者复写的。

这时如何解决问题呢？介绍一个关键字，final:最终。

Final

特点：

1：这个关键字是一个修饰符，可以修饰类，方法，变量。

2：被final修饰的类是一个最终类，不可以被继承。

3：被final修饰的方法是一个最终方法，不可以被覆盖。

4：被final修饰的变量是一个常量，只能赋值一次。

内部类只能访问被final修饰的局部变量。

其实这样的原因的就是给一些固定的数据起个阅读性较强的名称。

不加final修饰不是也可以使用吗？那么这个值是一个变量，是可以更改的。加了final，程序更为严谨。常量名称定义时，有规范，所有字母都大写，如果由多个单词组成，中间用 \_ 连接。

\*\*finally:永远被执行，除非退出jvm。System.exit(0);

\*\*\*：final,finally,finalize区别。

final是最终的意思。它可以用于修饰类，成员变量，成员方法。

它修饰的类不能被继承，它修饰的变量时常量，它修饰的方法不能被重写。

finally:是异常处理里面的关键字。

它其中的代码永远被执行。特殊情况：在执行它之前jvm退出。System.exit(0);

finalize:是Object类中的一个方法。

它是于垃圾回收器调用的方式。

## 抽象类: abstract ☆

多个类有相同的方法声明，但是方法体不一样。这个时候，我们考虑把方法声明进行抽取。

让子类继承后，自己去实现方法体。没有方法体的方法，我们需要用抽象标志下。

### 关键字

是：abstract。

抽象：不具体，看不明白。抽象类表象体现。

在不断抽取过程中，将共性内容中的方法声明抽取，但是方法不一样，没有抽取，这时抽取到的方法，并不具体，需要被指定关键字abstract所标示，声明为抽象方法。

抽象方法所在类一定要标示为抽象类，也就是说该类需要被abstract关键字所修饰。

### 特点：

1：抽象方法只能定义在抽象类中，抽象类和抽象方法必须由abstract关键字修饰（可以描述类和方法，不可以描述变量）。

2：抽象方法只定义方法声明，并不定义方法实现。

3：抽象类不可以被创建对象(实例化)。

4：只有通过子类继承抽象类并覆盖了抽象类中的所有抽象方法后，该子类才可以实例化。否则，该子类还是一个抽象类。

C:抽象类中不一定有抽象方法，但是，有抽象方法的类一定是抽象类。

### Detail：

#### 抽象类中是否有构造方法？

有，用于给子类对象进行初始化。供子类实例化调用。

#### 抽象类中是否可以定义非抽象方法？

可以。其实，抽象类和一般类没有太大的区别，都是在描述事物，只不过抽象类在描述事物时，有些功能不具体。所以抽象类和一般类在定义上，都是需要定义属性和行为的。只不过，比一般类多了一个抽象方法。而且比一般类少了一个创建对象的部分。

#### 抽象关键字abstract和哪些不可以共存？

final final修饰的方法不能被重写。而abstract修饰的方法，要求被重写。两者冲突

private私有内容子类继承不到，所以，不能重写。但是abstract修饰的方法，要求被重写。两者冲突。

static假如一个抽象方法能通过static修饰，那么这个方法，就可以直接通过类名调用。

而抽象方法是没有方法体的，这样的调用无意义。所以，不能用static修饰。

#### 抽象类中可不可以不定义抽象方法？

可以。抽象方法目的仅仅为了不让该类创建对象。

#### 抽象类中数据的特点

A:成员变量

抽象类中可以有变量，也可以有常量。

B:成员方法

抽象类中可以有抽象方法，也可以有非抽象方法。

C:构造方法

抽象类是一个类，所以，它有构造方法。

虽然本身不能实例化。但是可以给子类实例化使用

#### 什么时候使用抽象类呢？

在描述一类事物的时候，确实存在某种行为

但是现在这种行为是不具体的，这时候应该抽取这种行为的声明，而不去实现这种行为，这种行为把它称为抽象的行为，这时候应该使用抽象类。

#### 抽象类要注意的细节

1. 如果一个方法没有方法体，那么该方法必须使用abstract修饰。

2. 如果一个类含有抽象方法，那么这个类肯定是一个抽象类或者接口。

3. 抽象类不能创建对象。

4. 抽象类是含有构造方法的。

5. 抽象类可以存在非抽象方法与抽象方法。

6. 抽象方法可以不存在抽象方法。

7. 非抽象类继承抽象类的时候，必须要把抽象类中所有的抽象方法全部实现。

Abstract不能与以下关键字一起使用

1. abstract不能与static共同修饰一个方法。

2. abstrat不能与private共同修饰一个方法。

3. abstract不能以final关键字共同修饰一个方法。

# String字符串(重点)

(2)创建字符串对象

A:String():无参构造

\*\*举例：

String s = new String();

s = "hello";

sop(s);

B:String(byte[] bys):传一个字节数组作为参数 \*\*\*\*\*

\*\*举例

byte[] bys = {97,98,99,100,101};

String s = new String(bys);

sop(s);

C:String(byte[] bys,int index,int length):把字节数组的一部分转换成一个字符串 \*\*\*\*\*

\*\*举例

byte[] bys = {97,98,99,100,101};

String s = new String(bys,1,2);

sop(s);

D:String(char[] chs):传一个字符数组作为参数 \*\*\*\*\*

\*\*举例

char[] chs = {'a','b','c','d','e'};

String s = new String(chs);

sop(s);

E:String(char[] chs,int index,int length):把字符数组的一部分转换成一个字符串 \*\*\*\*\*

\*\*举例

char[] chs = {'a','b','c','d','e'};

String s = new String(chs,1,2);

sop(s);

F:String(String str):把一个字符串传递过来作为参数

char[] chs = {'a','b','c','d','e'};

String ss = new String(s);

sop(ss);

G:直接把字符串常量赋值给字符串引用对象(最常用) \*\*\*\*\*

\*\*举例

String s = "hello";

sop(s);

(3)面试题

A:请问String s = new String("hello");创建了几个对象。

两个。一个"hello"字符串对象，在方法区的常量池；一个s对象，在栈内存。

B:请写出下面的结果

String s1 = new String("abc");

Strign s2 = new String("abc");

String s3 = "abc";

String s4 = "abc";

sop(s1==s2); //false

sop(s1==s3); //false

sop(s3==s4); //true

C:字符串对象一旦被创建就不能被改变。

指的是字符串常量值不改变。

(4)字符串中各种功能的方法

A:判断

\*\*\*\* boolean equals(Object anObject):判断两个字符串的内容是否相同，复写了Object的方法

\*\*\*\* boolean equalsIgnoreCase(String anotherString):判断两个字符串的内容是否相同，

不区分大小写

\*\*\*\* boolean contains(String s):判断一个字符串中是否包含另一个字符串

注意：判断字符串是否包含特殊字符.直接表示为str.contains(".")

boolean endsWith(String suffix):测试此字符串是否以指定的后缀结束

boolean startsWith(String suffix):测试此字符串是否以指定的前缀开始

boolean isEmpty():测试字符串是否为空

B:获取

\*\*\*\*\* int length():返回此字符串的长度

\*\*\*\*\* char charAt(int index):返回指定索引处的 char值

\*\*\*\*\* int indexOf(int ch):返回指定字符在此字符串中第一次出现处的索引。

int indexOf(int ch, int fromIndex):返回在此字符串中第一次出现指定字符处的索引，

从指定的索引开始搜索。

int indexOf(String str):返回指定子字符串在此字符串中第一次出现处的索引。

int indexOf(String str, int fromIndex):返回指定子字符串在此字符串中第一次

出现处的索引，从指定的索引开始。

\*\*\* int lastIndexOf(int ch)：返回指定字符在此字符串中最后一次出现处的索引。

int lastIndexOf(int ch, int fromIndex)

返回指定字符在此字符串中最后一次出现处的索引,从指定的索引处开始进行反向搜索。

int lastIndexOf(String str)

返回指定子字符串在此字符串中最右边出现处的索引。

int lastIndexOf(String str, int fromIndex)

返回指定子字符串在此字符串中最后一次出现处的索引，从指定的索引开始反向搜索。

\*\*\*\*\* String substring(int beginIndex) (注意：该方法substring的String是小写！！！)

返回一个新的字符串，它是此字符串的一个子字符串。

String substring(int beginIndex, int endIndex) (注意该方法的String是小写！！！)

返回一个新字符串，它是此字符串的一个子字符串,包含头不包含尾。

C:转换

\*\*\*\*\* byte[] getBytes()：(很常用！)从字符串到字节数组的方法

void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)

将字符从此字符串复制到目标字符数组。

\*\*\*\*\* char[] toCharArray()：(很常用！)从字符串到字符数组的方法

\*\*\*\* static String copyValueOf(char[] data)

返回指定数组中表示该字符序列的 String。

static String copyValueOf(char[] data, int offset, int count)

返回指定数组中表示该字符序列的 String。

\*\*\*\*\* static String valueOf(数据类型):把该数据类型的数据转换成字符串。

\*\*\* String toLowerCase()：把字符串转换成小写

String toUpperCase()：把字符串转换成大写

\*\*\* 字符串的连接

String concat(String str):将指定字符串连接到此字符串的结尾。

D:替换

String replace(char oldChar, char newChar):用新字符替换旧字符(替换所有)

String replace(String target, String replacement):用新的子串换旧串

E:分割

String[] split(String regex)：根据指定的字符串把一个字符串分割成一个字符串数组

F:

String trim():去除字符串的前后空格

G:

int compareTo(String anotherString)

按字典顺序比较两个字符串。

int compareToIgnoreCase(String str)

按字典顺序比较两个字符串，不考虑大小写。

(5)练习

1:模拟登录,给三次机会,并提示还有几次.

默认的用户名和密码为admin。 区分大小写。

自己从键盘输入用户名和密码。

2:给定一个字符串统计,统计大写字母,小写字母,数字出现的个数.

\*\*\*注意:不包括特殊字符

从键盘输入一个不包含特殊字符的字符串(只有26个字母和0-9组成)。

3:给定一个字符串,把它变成首字母大写,其他字母小写的字符串.

从键盘输入一个字符串，全部26个字母组成的。

4:子串在整串中出现的次数。

也就是说:获取一个字符串中,指定的字串在该字符串中出现的次数.

例如:

"nbasdnbafllgnbahjnbakqqqqlnba" 在这个字符串中，多有个nba.

5:对字符串中字符进行自然顺序排序。

"basckd"-->"abcdks"

先留做思考内容：

6:两个字符串的最大相同子串。

两个字符串的最大相同子串。

比如:

"sadabcdfghjkl"

werabcdtyu"

JDK版本的升级原则：

A:提高效率

B:提高安全性

C:简化书写

4：基本数据类型的对象包装类

(1)为了更方便的操作每个基本数据类型，java对其提供了很多的属性和方法供我们使用。

(2)用途：

\*\*将基本数据类型封装成对象的好处在于可以在对象中定义更多的功能操作该数据。

\*\*常用的操作之一：用于基本数据类型与字符串之间的转换。

A:方便操作

B:用于和字符串进行相互转换

(3)基本数据类型和对象类型的对应

byte Byte

shortShort

int Integer

long Long

float Float

double Double

boolean Boolean

char Character

(4)构造方法

字段摘要：

static int MAX\_VALUE 值为 2^31-1 的常量，它表示 int 类型能够表示的最大值

static int MIN\_VALUE 值为 -2^31 的常量，它表示 int 类型能够表示的最小值

static Class<Integer> TYPE 表示基本类型int的Class 实例

Integer(int value) 构造一个新分配的Integer对象，它表示指定的int值。

Inreger(String s) 注意：s必须是纯数字的字符串。否则会有异常NumberFormatException

(5)几个常用的方法

Integer.toOctalString();

以八进制（基数 8）无符号整数形式返回一个整数参数的字符串表示形式。

Integer.toHexString();

以十六进制（基数 16）无符号整数形式返回一个整数参数的字符串表示形式。

static int Integer.parseInt(String s) 将字符串参数作为有符号的十进制整数进行解析,

字符串必须是int型范围内的数字字符串

static int Integer.parseInt(String s,int basic)

使用第二个参数指定的基数,将字符串参数解析为有符号的整数.

字符串必须是int型范围内的数字字符串

short shortValue() 以short类型返回该Integer的值。

int intValue() 以int类型返回该Integer的值。

static Integer valueOf(int num) 返回一个表示指定的 int 值的 Integer 实例。

static Integer valueOf(String s) 返回保存指定的String的值的Integer对象。

static Integer valueOf(String s, int radix)

返回一个Integer对象，该对象中保存了用第二个参数提供的基数进行

解析时从指定的String中提取的值。

(6)类型转换

int -- Integer

int num = 20;

A:Integer i = new Integer(num);

B:Integer i = Integer.valueOf(num);

Integer -- int

Integer i = new Integer(20);

A:int num = i.intValue();

int -- String

int num = 20;

A:String s = String.valueOf(num);

B:String s = ""+num;

C:String s = Integer.toString(num);

String -- int

String s = "20";

A:int num = Integer.parseInt(s);

B:Integer i = new Integer(s);或者Integer i = Integer.valueOf(s);

int num = i.intValue();

10、堆栈和队列

堆栈：先进后出，比如杯子里的水

队列：先进先出，比如水管的水

12、如果你想将一组对象按一定顺序存取，在不考虑并发访问的情况下会使用\_\_\_\_C\_\_\_\_\_ ,

反之则会使用\_\_\_\_A\_\_\_\_\_；如果你想存储一组无序但唯一的对象，你会使用\_\_\_B\_\_\_\_\_\_ ;

如果你想按关键字对对象进行存取，在不考虑并发访问的情况下会使用\_\_\_D\_\_\_\_\_\_ ,反之则会使用\_\_\_\_\_E\_\_\_\_。

A. Vector

B. HashSet

C. ArrayList

D. HashMap

E. Hashtable

## --< java.lang >-- String字符串：★★★☆

java中用String类进行描述。对字符串进行了对象的封装。这样的好处是可以对字符串这种常见数据进行方便的操作。对象封装后，可以定义N多属性和行为。

如何定义字符串对象呢？String s = "abc";只要是双引号引起的数据都是字符串对象。

特点：字符串一旦被初始化，就不可以被改变，存放在方法区中的常量池中。

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

String s1 = "abc"; // s1指向的内存中只有一个对象abc。

String s2 = new String("abc"); // s2指向的内容中有两个对象abc、new 。

System.out.println(s1==s2);//false

System.out.println(s1.equals(s2));//true ，字符串中equals比较的是字符串内容是否相同。

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

## 字符串的方法：

1：构造方法：将字节数组或者字符数组转成字符串。

String s1 = new String();//创建了一个空内容的字符串。

String s2 = null;//s2没有任何对象指向，是一个null常量值。

String s3 = "";//s3指向一个具体的字符串对象，只不过这个字符串中没有内容。

//一般在定义字符串时，不用new。

String s4 = new String("abc");

String s5 = "abc"; 一般用此写法

new String(char[]);//将字符数组转成字符串。

new String(char[],offset,count);//将字符数组中的一部分转成字符串。

2、一般方法：

按照面向对象的思想：

2.1 获取：

2.1.1：获取字符串的长度。length();

2.1.2：指定位置的字符。char charAt(int index);

2.1.3：获取指定字符的位置。如果不存在返回-1，所以可以通过返回值-1来判断某一个字符不存在的情况。

int indexOf(int ch);//返回第一次找到的字符角标

int indexOf(int ch,int fromIndex); //返回从指定位置开始第一次找到的角标

int indexOf(String str); //返回第一次找到的字符串角标

int indexOf(String str,int fromIndex);

int lastIndexOf(int ch);

int lastIndexOf(int ch,int fromIndex);

int lastIndexOf(String str);

int lastIndexOf(String str,int fromIndex);

2.1.4：获取子串。

String substring(int start);//从start位开始，到length()-1为止.

String substring(int start,int end);//从start开始到end为止。//包含start位，不包含end位。

substring(0,str.length());//获取整串

2.2 判断：

2.2.1：字符串中包含指定的字符串吗？

boolean contains(String substring);

2.2.2：字符串是否以指定字符串开头啊？

boolean startsWith(string);

2.2.3：字符串是否以指定字符串结尾啊？

boolean endsWith(string);

2.2.4：判断字符串是否相同

boolean equals(string);//覆盖了Object中的方法，判断字符串内容是否相同。

2.2.5：判断字符串内容是否相同，忽略大小写。

boolean equalsIgnoreCase(string) ;

2.3 转换：

2.3.1：通过构造方法可以将字符数组或者字节数组转成字符串。

2.3.2：可以通过字符串中的静态方法，将字符数组转成字符串。

static String copyValueOf(char[] );

static String copyValueOf(char[],int offset,int count);

static String valueOf(char[]);

static String valueOf(char[],int offset,int count);

2.3.3：将基本数据类型或者对象转成字符串。

static String valueOf(char);

static String valueOf(boolean);

static String valueOf(double);

static String valueOf(float);

static String valueOf(int);

static String valueOf(long);

static String valueOf(Object);

2.3.4：将字符串转成大小写。

String toLowerCase();

String toUpperCase();

2.3.5：将字符串转成数组。

char[] toCharArray();//转成字符数组。

byte[] getBytes();//可以加入编码表。转成字节数组。

2.3.6：将字符串转成字符串数组。切割方法。

String[] split(分割的规则-字符串);

2.3.7：将字符串进行内容替换。注意：修改后变成新字符串，并不是将原字符串直接修改。

String replace(oldChar,newChar);

String replace(oldstring,newstring);

2.3.8： String concat(string); //对字符串进行追加。

String trim();//去除字符串两端的空格

int compareTo();//如果参数字符串等于此字符串，则返回值 0；如果此字符串按字典顺序小于字符串参数，则返回一个小于 0 的值；如果此字符串按字典顺序大于字符串参数，则返回一个大于 0 的值。

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

## 2. StringBuilder

--< java.lang >-- StringBuilder字符串缓冲区：★★★☆

JDK1.5出现StringBuiler；构造一个其中不带字符的字符串生成器，初始容量为 16 个字符。该类被设计用作 StringBuffer 的一个简易替换，用在字符串缓冲区被单个线程使用的时候（这种情况很普遍）。

方法和StringBuffer一样；

StringBuffer 和 StringBuilder 的区别：

StringBuffer线程安全。

StringBuilder线程不安全。

单线程操作，使用StringBuilder 效率高。

多线程操作，使用StringBuffer 安全。

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

StringBuilder sb = new StringBuilder("abcdefg");

sb.append("ak"); //abcdefgak

sb.insert(1,"et");//aetbcdefg

sb.deleteCharAt(2);//abdefg

sb.delete(2,4);//abefg

sb.setLength(4);//abcd

sb.setCharAt(0,'k');//kbcdefg

sb.replace(0,2,"hhhh");//hhhhcdefg

//想要使用缓冲区，先要建立对象。

StringBuffer sb = new StringBuffer();

sb.append(12).append("haha");//方法调用链。

String s = "abc"+4+'q';

s = new StringBuffer().append("abc").append(4).append('q').toString();

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

class Test{

public static void main(String[] args) {

String s1 = "java";

String s2 = "hello";

method\_1(s1,s2);

System.out.println(s1+"...."+s2); //java....hello

StringBuilder s11 = new StringBuilder("java");

StringBuilder s22 = new StringBuilder("hello");

method\_2(s11,s22);

System.out.println(s11+"-----"+s22); //javahello-----hello

}

public static void method\_1(String s1,String s2){

s1.replace('a','k');

s1 = s2;

}

public static void method\_2(StringBuilder s1,StringBuilder s2){

s1.append(s2);

s1 = s2;

}

}

基本数据类型包装类

基本数据类型对象包装类：是按照面向对象思想将基本数据类型封装成了对象。

好处：

1：可以通过对象中的属性和行为操作基本数据。

2：可以实现基本数据类型和字符串之间的转换。

关键字 对应的类名

byte Byte

short Short paserShort(numstring);

int Integer 静态方法：parseInt(numstring)

long Long

float Float

double Double

char Character

Boolean Boolean

基本数据类型对象包装类：都有 XXX parseXXX 方法

只有一个类型没有parse方法：Character ；

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

## StringBuffer

--< java.lang >-- StringBuffer字符串缓冲区：★★★☆

构造一个其中不带字符的字符串缓冲区，初始容量为 16 个字符。

特点：

1：可以对字符串内容进行修改。

2：是一个容器。

3：是可变长度的。

4：缓冲区中可以存储任意类型的数据。

5：最终需要变成字符串。

API

构造方法

StringBuffer() 构造一个其中不带字符的字符串缓冲区，初始容量为 16 个字符。

StringBuffer(int num) 构造一个不带字符，但具有指定初始容量的字符串缓冲区。

StringBuffer(String str) 构造一个字符串缓冲区，并将其内容初始化为指定的字符串内容。

容器通常具备一些固定的方法：

1，添加。

StringBuffer append(data):在缓冲区中追加数据。追加到尾部。

StringBuffer insert(index,data):在指定位置插入数据。

2，删除。

StringBuffer delete(start,end);删除从start至end-1范围的元素

StringBuffer deleteCharAt(index);删除指定位置的元素

//sb.delete(0,sb.length());//清空缓冲区。

3，修改。

StringBuffer replace(start,end,string);将start至end-1替换成string

void setCharAt(index,char);替换指定位置的字符

void setLength(len);将原字符串置为指定长度的字符串

4，查找。（查不到返回-1）

int indexOf(string); 返回指定子字符串在此字符串中第一次出现处的索引。

int indexOf(string,int fromIndex);从指定位置开始查找字符串

int lastIndexOf(string); 返回指定子字符串在此字符串中最右边出现处的索引。

int lastIndexOf(string,int fromIndex); 从指定的索引开始反向搜索

5，获取子串。

string substring(start); 返回start到结尾的子串

string substring(start,end); 返回start至end-1的子串

6，反转。

StringBuffer reverse();字符串反转

### 字符串和StringBuffer的转换

String-->StringBuffer通过构造:

如:StringBuffer sb = new StringBuffer(String str)

StringBuffer--String通过toString方法

如:StringBuffer sb = new StringBuffer();

sb.toString();

# 数组集合

## 数 组

用于存储同一类型数据的一个容器。

好处：可以对该容器中的数据进行编号，从0开始。数组用于封装数据，就是一个具体的实体。

如何在java中表现一个数组呢？两种表现形式。

1）、元素类型[] 变量名 = new 元素类型[元素的个数]；

2）、元素类型[] 变量名 = {元素1，元素2...}；

元素类型[] 变量名 = new 元素类型[]{元素1，元素2...}；

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

//二分查找法。必须有前提：数组中的元素要有序。

public static int halfSeach\_2(int[] arr,int key){

int min,max,mid;

min = 0;

max = arr.length-1;

mid = (max+min)>>1; //(max+min)/2;

while(arr[mid]!=key){

if(key>arr[mid]){

min = mid + 1;

}

else if(key<arr[mid])

max = mid - 1;

if(max<min)

return -1;

mid = (max+min)>>1;

}

return mid;

}

//判断数组是否重复

public class Demo3{

public static void main(String [] args){

int [] age = {1,2,3,4,5};

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

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

if(age[i] ==age[j]){

System.out.println("有重复的！")；

break;

}

}

}

}

经典用法：冒泡法排序

class Bubblesort

{

public static void main(String args [])

{

int [] arr = {2,5,62,6,456,3};

bubble(arr);

printarray(arr);

}

public static void bubble(int [] arr)

{

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

{

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

{

if(arr[y] > arr[y+1])

{

int temp =0;

temp = arr[y];

arr[y]=arr[y+1];

arr[y+1] = temp;

}

}

}

}

}

public static void printarray(int [] arr)

{

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

{

if(i!arr.length -1)

System.out.println(arr[i]+",");

else

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

}

}

//选择排序：

public class Demo6{

public static void main(String [] args)

{

int [] age = {1,23,45,4,57,343,777};

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

for(int j = i+1;

j <= age.length -1;j++){

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

{

int temp = age[i];

age[i] = age[j];

age[j] = temp;

}

}

System.out.println(Array.toString(age));

}

}

用于存储数据的容器。

特点：

1：对象封装数据，对象多了也需要存储。集合用于存储对象。

2：对象的个数确定可以使用数组，但是不确定怎么办？可以用集合。因为集合是可变长度的。

3: 可存储不同类型的对象。

集合和数组的区别：

1：数组是固定长度的；集合可变长度的。

2：数组可以存储基本数据类型，也可以存储引用数据类型；集合只能存储引用数据类型。

3：数组存储的元素必须是同一个数据类型；集合存储的对象可以是不同数据类型。

数据结构：就是容器中存储数据的方式。

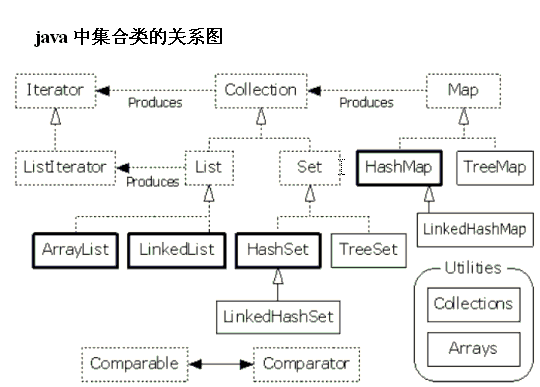
对于集合容器，有很多种。因为每一个容器的自身特点不同，其实原理在于每个容器的内部数据结构不同。

集合容器在不断向上抽取过程中。出现了集合体系。

在使用一个体系时，原则：参阅顶层内容。建立底层对象。

**自己能画出来,并且附带方法和步骤以及特性以及各自的遍历方式.**

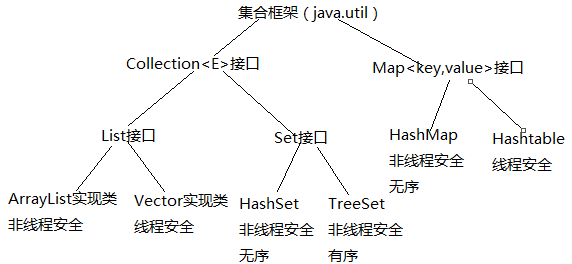
**常用的ArrayList、HashSet**

****

## 集合

有俩大分支，分别为Collection和map,全部来自java.util包

集合在底层是与数组是

****

### 集合(Collection)(掌握)

#### 集合的由来?

我们学习的是Java -- 面向对象 -- 操作很多对象 -- 存储 -- 容器(数组和StringBuffer) -- 数组

而数组的长度固定，所以不适合做变化的需求，Java就提供了集合供我们使用。

#### 集合和数组的区别?

A:长度区别

数组固定

集合可变

B:内容区别

数组可以是基本类型，也可以是引用类型

集合只能是引用类型

C:元素内容

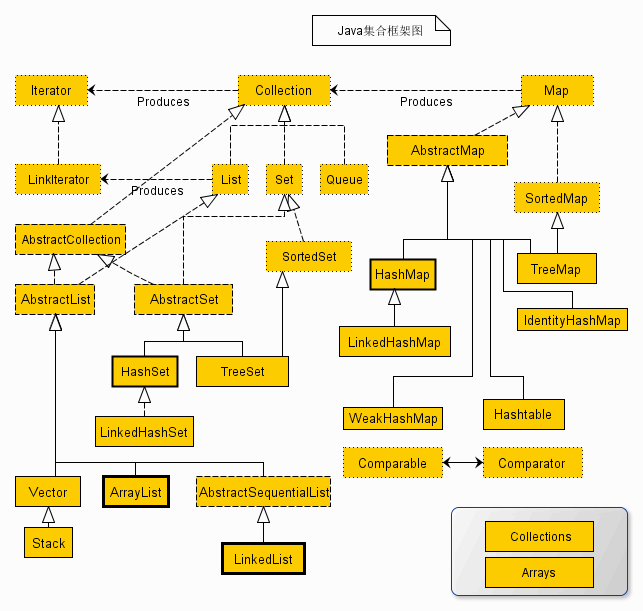
数组只能存储同一种类型

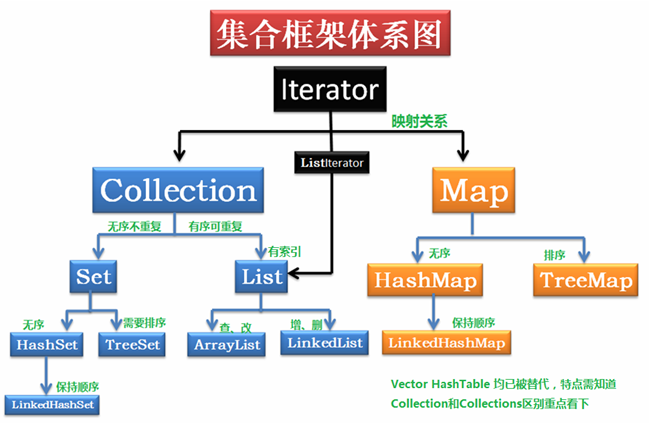
集合可以存储不同类型(其实集合一般存储的也是同一种类型)

#### 集合的继承体系结构?

由于需求不同，Java就提供了不同的集合类。这多个集合类的数据结构不同，但是它们都是要提供存储和遍历功能的，

我们把它们的共性不断的向上提取，最终就形成了集合的继承体系结构图。

类似的



Collection

|--List

|--ArrayList

|--Vector

|--LinkedList

|--Set

|--HashSet

|--TreeSet

#### Collection的功能概述(自己补齐)

A:添加功能

B:删除功能

C:判断功能

D:获取功能

E:长度功能

F:交集(了解)

G:把集合转数组(了解)

#### Collection集合的遍历

A:把集合转数组(了解)

B:迭代器(集合专用方式)

#### 迭代器

A:是集合的获取元素的方式。

B:是依赖于集合而存在的。

C:迭代器的原理和源码。

a:为什么定义为了一个接口而不是实现类?

b:看了看迭代器的内部类实现。

#### Collection集合的案例(遍历方式 迭代器)

集合的操作步骤：

A:创建集合对象

B:创建元素对象

C:把元素添加到集合

D:遍历集合

A:存储字符串并遍历

import java.util.Collection;

import java.util.ArrayList;

import java.util.Iterator;

public class CollectionDemo {

public static void main(String[] args) {

//创建集合对象

Collection c = new ArrayList();

//创建并添加元素

c.add("hello");

c.add("world");

c.add("java");

//遍历集合

Iterator it = c.iterator();

while(it.hasNext()) {

String s =(String) it.next();

System.out.println(s);

}

}

}

B:存储自定义对象并遍历

public class Student {

private String name;

private int age;

public Student(){}

public Student(String name,int age) {

this.name = name;

this.age = age;

}

//getXxx()/setXxx()

}

import java.util.Collection;

import java.util.ArrayList;

import java.util.Iterator;

public class StudentDemo {

public static void main(String[] args) {

//创建集合对象

Collection c = new ArrayList();

//创建学生对象

Student s1 = new Student("林青霞",27);

Student s2 = new Student("风清扬",30);

Student s3 = new Student("刘意",30);

Student s4 = new Student("武鑫",25);

Student s5 = new Student("刘晓曲",16);

//添加元素

c.add(s1);

c.add(s2);

c.add(s3);

c.add(s4);

c.add(s5);

//遍历集合

Iterator it = c.iterator();

while(it.hasNext()) {

Student s = (Student)it.next();

System.out.println(s.getName()+"---"+s.getAge());

}

}

}

|--List：有序(元素存入集合的顺序和取出的顺序一致)，元素都有索引。元素可以重复。

|--Set：无序(存入和取出顺序有可能不一致)，不可以存储重复元素。必须保证元素唯一性。

1，添加

add(object)：添加一个元素

addAll(Collection) ：添加一个集合中的所有元素。

 2，删除：

clear()：将集合中的元素全删除，清空集合。

remove(obj) ：删除集合中指定的对象。注意：删除成功，集合的长度会改变。

removeAll(collection) ：删除部分元素。部分元素和传入Collection一致。

 3，判断：

boolean contains(obj) ：集合中是否包含指定元素 。

boolean containsAll(Collection) ：集合中是否包含指定的多个元素。

 boolean isEmpty()：集合中是否有元素。

  4，获取：

int size()：集合中有几个元素。

 5，取交集：

boolean  retainAll(Collection) ：对当前集合中保留和指定集合中的相同的元素。如果两个集合元素相同，返回flase；如果retainAll修改了当前集合，返回true。

6，获取集合中所有元素：

 Iterator  iterator()：迭代器

7，将集合变成数组：

 toArray();

#### List

List本身是Collection接口的子接口，具备了Collection的所有方法。现在学习List体系特有的共性方法，查阅方法发现List的特有方法都有索引，这是该集合最大的特点。

**List：有序(元素存入集合的顺序和取出的顺序一致)，元素都有索引。元素可以重复。**

**|--ArrayList：底层的数据结构是数组,线程不同步，ArrayList替代了Vector，查询元素的速度非常快。**

**|--LinkedList：底层的数据结构是链表，线程不同步，增删元素的速度非常快。**

**|--Vector：底层的数据结构就是数组，线程同步的，Vector无论查询和增删都巨慢。**

**1，添加：**

add(index,element) ：在指定的索引位插入元素。

addAll(index,collection) ：在指定的索引位插入一堆元素。

**2，删除：**

remove(index) ：删除指定索引位的元素。 返回被删的元素。

**3，获取：**

Object get(index) ：通过索引获取指定元素。

int **indexOf**(obj) ：获取指定元素第一次出现的索引位，如果该元素不存在返回-1；

 所以，通过-1，可以判断一个元素是否存在。

int lastIndexOf(Object o) ：反向索引指定元素的位置。

 List**subList**(start,end) ：获取子列表。

**4，修改：**

Object set(index,element) ：对指定索引位进行元素的修改。

**5，获取所有元素：**

ListIterator listIterator()：list 集合特有的迭代器。

**List集合支持对元素的增、删、改、查。**

**List集合因为角标有了自己的获取元素的方式：**

**遍历:**

for(int x=0; x<list.size(); x++) {

sop("get:"+list.get(x));

}

在进行list列表元素迭代的时候，如果想要在迭代过程中，想要对元素进行操作的时候，比如满足条件添加新元素。**会发生.ConcurrentModificationException并发修改异常。**

**导致的原因是：**

集合引用和迭代器引用在同时操作元素，通过集合获取到对应的迭代器后，在迭代中，进行集合引用的元素添加，迭代器并不知道，所以会出现异常情况。

**如何解决呢？**

既然是在迭代中对元素进行操作,找迭代器的方法最为合适.可是Iterator中只有hasNext,next,remove方法.通过查阅的它的子接口,ListIterator,发现该列表迭代器接口具备了对元素的增、删、改、查的动作。

ListIterator是List集合特有的迭代器。

ListIterator it = list.listIterator;//取代Iterator it = list.iterator;

##### 遍历List集合的三种方法

List<String> list = new ArrayList<String>();

list.add("aaa");

list.add("bbb");

list.add("ccc");

方法一：

超级for循环遍历

for(String attribute : list) {

System.out.println(attribute);

}

方法二：

对于ArrayList来说速度比较快, 用for循环, 以size为条件遍历:

for(int i = 0 ; i < list.size() ; i++) {

system.out.println(list.get(i));

}

方法三：

集合类的通用遍历方式, 从很早的版本就有, 用迭代器迭代

Iterator it = list.iterator();

while(it.hasNext()) {

System.ou.println(it.next);

}

##### ArrayList

|--ArrayList：底层的数据结构是数组,线程不同步，ArrayList替代了Vector，查询快，增删慢，默认长度10，50%延长长度。

|--->ArrayList:

(1)当往ArrayList里面存入元素没什么要求时，即只要求有序就行时；

(2)当往ArrayList里面存入元素要求不重复时，比如存入学生对象，当同名同姓时

视为同一个人，则不往里面存储。则定义学生对象时，需复写equals方法

public boolean equals(Object obj)

{

if(!(obj instanceof Student))

return false;

Student stu = (Student)obj;

return this.name.equals(stu.name)&&this.age==stu.age;

}

则往ArrayList集合通过add存入学生对象时，集合底层自己会调用学生类的equals方法，

判断重复学生则不存入。

注：对于List集合，无论是add、contains、还是remove方法，判断元素是否相同，

都是通过复写equals方法来判断！

API

ArrayList：

|--->构造方法摘要：(少用，不是重点)

ArrayList()：构造一个初始容量为 10 的空列表。

ArrayList(Collection<? extends E> c)： 构造一个包含指定 collection 的元素的列表，

ArrayList(int initialCapacity)： 构造一个具有指定初始容量的空列表。

|--->方法摘要：

|--->添加：

boolean add(E e)： 将指定的元素添加到此列表的尾部。

void add(int index, E element)： 将指定的元素插入此列表中的指定位置。

boolean addAll(Collection<? extends E> c):按照指定 collection 的迭代器所返回的元素顺序，将该 collection 中的所有元素添加到此列表的尾部

boolean addAll(int index, Collection<? extends E> c)： 从指定的位置开始，将指定 collection中的所有元素插入到此列表中。

|--->删除：

void clear (): 移除此列表中的所有元素。

E remove(int index): 移除此列表中指定位置上的元素。

boolean remove(Object o): 移除此列表中首次出现的指定元素（如果存在）。

protected void removeRange(int fromIndex, int toIndex): 移除列表中索引在 fromIndex（包括）和 toIndex（不包括）之间的所有元素。

boolean removeAll(Collection<?> c): 从列表中移除指定 collection 中包含的其所有元素

|--->获取：

E get(int index): 返回此列表中指定位置上的元素。

int indexOf(Object o): 返回此列表中首次出现的指定元素的索引，或如果此列表不包含元素，则返回 -1。

int lastIndexOf(Object o) 返回此列表中最后一次出现的指定元素的索引，或如果此列表不包含索引，则返回 -1。

public List<E> subList(int fromIndex,int toIndex): 返回列表中指定的 fromIndex（包括 ） 和 toIndex（不包括）之间的部分视图。

Iterator<E> iterator(): 返回按适当顺序在列表的元素上进行迭代的迭代器。

ListIterator<E> listIterator(int index):返回列表中元素的列表迭代器(按适当顺序),从列表的指定位置开始。

|--->修改：(特有方法！！)

E set(int index, E element): 用指定的元素替代此列表中指定位置上的元素。

##### LinkedList

|--LinkedList：底层的数据结构是链表，线程不同步，增删非常快，查询慢。

|--Vector：底层的数据结构就是数组，线程同步的，Vector无论查询和增删都巨慢。默认10，100%增删长度。被ArrayList取代。

|--->LinkedList

(1)LinkLedist的特有方法：

boolean offerFirst(E e) 在此列表的开头插入指定的元素。

boolean offerLast(E e) 在此列表末尾插入指定的元素。

E peekFirst() 获取但不移除此列表的第一个元素；如果此列表为空，则返回 null。

E peekLast() 获取但不移除此列表的最后一个元素；如果此列表为空，则返回 null。

E pollFirst() 获取并移除此列表的第一个元素；如果此列表为空，则返回 null。

E pollLast() 获取并移除此列表的最后一个元素；如果此列表为空，则返回 null。

(2)通过LinkLedist的特有方法，可以实现某些数据特殊方式的存取，比如堆栈和队列。

一般情况下，使用哪种List接口下的实现类呢？

如果要求增删快，考虑使用LinkedList

如果要求查询快，考虑使用ArrayList

如果要求线程安全，考虑使用Vector。

1，添加：

add(index,element) ：在指定的索引位插入元素。

addAll(index,collection) ：在指定的索引位插入一堆元素。

2，删除：

remove(index) ：删除指定索引位的元素。 返回被删的元素。

3，获取：

Object get(index) ：通过索引获取指定元素。

int indexOf(obj) ：获取指定元素第一次出现的索引位，如果该元素不存在返回-1；

所以，通过-1，可以判断一个元素是否存在。

int lastIndexOf(Object o) ：反向索引指定元素的位置。

List subList(start,end) ：获取子列表。

4，修改：

Object set(index,element) ：对指定索引位进行元素的修改。

5，获取所有元素：

ListIterator listIterator()：list集合特有的迭代器。

API

|--->构造方法摘要：

LinkedList(): 构造一个空列表。

LinkedList(Collection<? extends E> c): 构造一个包含指定 collection 中的元素的列表，这些元素按其 collection 的迭代器返回的顺序排列。

|--->方法摘要:(特有的)

|--->添加

void addFirst(E e): 将指定元素插入此列表的开头。

void addLast(E e): 将指定元素添加到此列表的结尾。

|--->获取元素，但不删除元素

E get(int index): 返回此列表中指定位置处的元素。

E getFirst(): 返回此列表的第一个元素。

E getLast(): 返回此列表的最后一个元素。

|--->获取元素且删除元素

E remove()： 获取并移除此列表的头（第一个元素）。

E remove(int index)： 移除此列表中指定位置处的元素。

boolean remove(Object o)： 从此列表中移除首次出现的指定元素（如果存在）。

E removeFirst()： 移除并返回此列表的第一个元素。

E removeLast(): 移除并返回此列表的最后一个元素。

|--->修改

E set(int index, E element) 将此列表中指定位置的元素替换为指定的元素。

##### Vector

|--->构造方法摘要：

Vector(): 构造一个空向量，使其内部数据数组的大小为 10，其标准容量增量为零。

Vector(Collection<? extends E> c): 构造一个包含指定 collection 中的元素的向量，这些元素按其 collection 的迭代器返回元素的顺序排列。

|--->方法摘要：

|--->添加：

boolean add(E e): 将指定元素添加到此向量的末尾。

void add(int index, E element): 在此向量的指定位置插入指定的元素。

boolean addAll(Collection<? extends E> c):

将指定 Collection 中的所有元素添加到此向量的末尾，

按照指定 collection 的迭代器所返回的顺序添加这些元素。

boolean addAll(int index, Collection<? extends E> c): 在指定位置将指定 Collection 中的所有元素插入到此向量中。

|--->获取:

Enumeration<E> elements(): 返回此向量的组件的枚举。

Vector特有的取出方式:

枚举和迭代器很像，其实枚举和迭代器是一样的，只是因为枚举的名称和方法的名称

名字都过长，所以枚举被迭代器取代了。

|--->枚举Enumeration的方法摘要：

boolean hasMoreElements(): 测试此枚举是否包含更多的元素。

E nextElement(): 如果此枚举对象至少还有一个可提供的元素，

则返回此枚举的下一个元素。

List集合支持对元素的增、删、改、查。

List集合因为角标有了自己的获取元素的方式： 遍历。

for(int x=0; x<list.size(); x++){

sop("get:"+list.get(x));

}

在进行list列表元素迭代的时候，如果想要在迭代过程中，想要对元素进行操作的时候，比如满足条件添加新元素。会发生.ConcurrentModificationException并发修改异常。

导致的原因是：

集合引用和迭代器引用在同时操作元素，通过集合获取到对应的迭代器后，在迭代中，进行集合引用的元素添加，迭代器并不知道，所以会出现异常情况。

如何解决呢？

既然是在迭代中对元素进行操作,找迭代器的方法最为合适.可是Iterator中只有hasNext,next,remove方法.通过查阅的它的子接口,ListIterator,发现该列表迭代器接口具备了对元素的增、删、改、查的动作。

ListIterator是List集合特有的迭代器。

ListIterator it = list.listIterator;//取代Iterator it = list.iterator;

方法摘要

void add(E e) 将指定的元素插入列表（可选操作）。 boolean hasNext() 以正向遍历列表时，如果列表迭代器有多个元素，则返回 true（换句话说，如果 next 返回一个元素而不是抛出异常，则返回 true）。

boolean hasPrevious() 如果以逆向遍历列表，列表迭代器有多个元素，则返回 true。 E

next() 返回列表中的下一个元素。

int nextIndex() 返回对 next 的后续调用所返回元素的索引。

E

previous() 返回列表中的前一个元素。

int previousIndex() 返回对 previous 的后续调用所返回元素的索引。

void remove() 从列表中移除由 next 或 previous 返回的最后一个元素（可选操作）。

void set(E e) 用指定元素替换 next 或 previous 返回的最后一个元素（可选操作）。

#### Set接口：

Set接口中的方法和Collection中方法一致的。Set接口取出方式只有一种，迭代器。

Set:集合，底层是数组，无序的(无索引)，不可重复。可null。

**|--HashSet：**底层数据结构是哈希表，线程是不同步的。无序，高效；

**HashSet集合保证元素唯一性：**通过元素的hashCode方法，和equals方法完成的。 当元素的hashCode值相同时，才继续判断元素的equals是否为true。 如果为true，那么视为相同元素，不存。如果为false，那么存储。 如果hashCode值不同，那么不判断equals，从而提高对象比较的速度。

**|--LinkedHashSet：有序，hashset的子类。**

**|--TreeSet：**对Set集合中的元素的进行指定顺序的排序。不同步。TreeSet底层的数据结构就是二叉树。

##### HashSet

|--HashSet：底层哈希表，线程不同步的。无序，唯一，高效。

保证元素唯一性：

\*\*如果两元素的hashCode值不同，则不会调用equals方法

\*\*如果两元素的hashCode值相同，则继续判断equals是否返回true；

\*\*hashCode和equals方法虽然定义在自定义对象类里面，但不是我们手动调用

而是往HashSet集合里面存储元素的时候，集合底层自己调用hashCode和equals

它自己拿对象去判断，自己判断两元素是否是同一个元素。

通过new的方式往HashSet里面存的元素的hashCode都不同，但通常我们定义对象，

比如学生对象时，虽然是new的两个学生对象，但是当他们name和age一样时，我们认为是

同一个对象，所以为了保证元素的唯一性，我们通常在往HashSet集合里面存储元素时，

在定义对象的类中通常复写hashCode和equals方法。

public int hashCode()

{

return name.hashCode()+age\*39;

}

public boolean equals(Object obj)

{

if(!(obj instanceof Student))

return false;

Student stu = (Student)obj;

return this.name.equals(stu.name)&&this.age==stu.age;

}

\*\*当我们往HashSet集合存放自定义的元素时(比如学生对象)，通常都要复写hashCode和equals方法，

而且hashCode和equals方法不通过我们调用，HashSet集合底层内部自己调用，自己拿元素去比较

API

|--->构造方法：

HashSet() 构造一个新的空 set，其底层 HashMap 实例的默认初始容量是 16，加载因子是 0.75。

HashSet(Collection<? extends E> c) 构造一个包含指定 collection 中的元素的新 set。

|--->方法摘要：

boolean add(E e) 如果此 set 中尚未包含指定元素，则添加指定元素。

void clear() 从此 set 中移除所有元素。

Object clone() 返回此 HashSet 实例的浅表副本：并没有复制这些元素本身。

boolean contains(Object o) 如果此 set 包含指定元素，则返回 true。

boolean isEmpty() 如果此 set 不包含任何元素，则返回 true。

Iterator<E> iterator() 返回对此 set 中元素进行迭代的迭代器。

boolean remove(Object o) 如果指定元素存在于此 set 中，则将其移除。

int size() 返回此 set 中的元素的数量（set 的容量）。

LinkedHashSet

|--LinkedHashSet：有序，hashset的子类。

##### TreeSet

|--TreeSet：对存放的元素进行排序，弥补了Set集合元素无序的缺点，且元素是唯一的有序，不同步。底层是二叉树，线程不同步。

用于对Set集合进行元素的指定顺序排序，排序需要依据元素自身具备的比较性。

使用元素的自然顺序对元素进行排序，或者根据创建 set 时提供的 Comparator 进行排序.

如果元素不具备比较性，在运行时会发生ClassCastException异常。

所以需要元素实现Comparable接口，强制让元素具备比较性，复写compareTo方法。

依据compareTo方法的返回值，确定元素在TreeSet数据结构中的位置。

TreeSet方法保证元素唯一性的方式：就是参考比较方法的结果是否为0，如果return 0，视为两个对象重复，不存。

注意：在进行比较时，如果判断元素不唯一，比如，同姓名，同年龄，才视为同一个人。

在判断时，需要分主要条件和次要条件，当主要条件相同时，再判断次要条件，按照次要条件排序。

排序方式

TreeSet集合排序有两种方式，Comparable和Comparator区别：

1：让元素自身具备比较性，需要元素对象实现Comparable接口，覆盖compareTo方法。

定义对象类，实现Compareble接口,复写compareTo方法，此方式是元素的自然顺序

class Student implements Comparable

{

private String name;

private int age;

public Student(String name,int age)

{

this.name=name;

this.age=age;

}

public String getName()

{

return name;

}

public int getAge()

{

return age;

}

public int compareTo(Object obj)

{

if(!(obj instanceof Student))

throw new RuntimeException("不是学生对象！");

Student stu = (Student)obj;

int num = this.age-stu.age;

if(num==0)

return this.name.compareTo(stu.name);

return num;

}

}

2：当元素自身不具备比较性(比如存储学生对象时)或者具备的比较性不是我们所需要的比较性时(比如想字符串的长度排序),此时就需要让集合自身具备自定义的比较性。让集合自身具备比较性，需要定义一个实现了Comparator接口的比较器，并覆盖compare方法，并将该类对象作为实际参数传递给TreeSet集合的构造方法。

第二种方式较为灵活。

class StringLengthComparator implements Comparator

{

public int compare(Object obj1,Object obj2)

{

String s1 = (String)obj1;

String s2 = (String)obj2;

int num = new Integer(s1.length()).compareTo(new Integer(s2.length()));

if(num==0)

return s1.compareTo(s2);

return num;

}

}

class TreeSetTest

{

public static void main(String[] args)

{

TreeSet ts = new TreeSet(new StringLengthComparator());

ts.add("addfg");

ts.add("dfg");

ts.add("agtuug");

ts.add("vgjkg");

sop(ts);

}

}

基本数据类型或字符串对象均实现了Comparable接口，故同种类型基本数据间具备比较性，即自然顺序。

注：

\*\*判断元素唯一时，当主要条件一样时，判断次要条件

\*\*两种排序方式都在时，以比较器为主！！！

API

|--->构造方法：

TreeSet() 构造一个新的空 set，该set根据其元素的自然顺序进行排序。

TreeSet(Collection<? extends E> c)

构造一个包含指定 collection 元素的新 TreeSet，它按照其元素的自然顺序进行排序。

TreeSet(Comparator<? super E> comparator) 构造一个新的空 TreeSet，它根据指定比较器进行排序。

|--->方法摘要：

添加：

boolean add(E e) 将指定的元素添加到此 set（如果该元素尚未存在于 set 中）。

boolean addAll(Collection<? extends E> c) 将指定 collection 中的所有元素添加到此 set 中。

删除：

void clear() 移除此 set 中的所有元素。

boolean remove(Object o) 将指定的元素从 set 中移除（如果该元素存在于此 set 中）。

E pollFirst() 获取并移除第一个（最低）元素；如果此 set 为空，则返回 null。

E pollLast() 获取并移除最后一个（最高）元素；如果此 set 为空，则返回 null。

获取：

Iterator<E> iterator() 返回在此 set 中的元素上按升序进行迭代的迭代器。

E first() 返回此 set 中当前第一个（最低）元素。

E last() 返回此 set 中当前最后一个（最高）元素。

int size() 返回 set 中的元素数（set 的容量）。

判断：

boolean isEmpty() 如果此 set 不包含任何元素，则返回 true。

boolean contains(Object o) 如果此 set 包含指定的元素，则返回 true。

##### 哈希表的原理：

对对象元素中的关键字(对象中的特有数据)，进行哈希算法的运算，并得出一个具体的算法值，这个值 称为哈希值。

哈希值就是这个元素的位置。

3，如果哈希值出现冲突，再次判断这个关键字对应的对象是否相同。如果对象相同，就不存储，因为元素重复。如果对象不同，就存储，在原来对象的哈希值基础 +1顺延。 4，存储哈希值的结构，我们称为哈希表。

5，既然哈希表是根据哈希值存储的，为了提高效率，最好保证对象的关键字是唯一的。 这样可以尽量少的判断关键字对应的对象是否相同，提高了哈希表的操作效率。

 对于ArrayList集合，判断元素是否存在，或者删元素底层依据都是equals方法。

对于HashSet集合，判断元素是否存在，或者删除元素，底层依据的是hashCode方法和equals方法。

### Iterator接口

--< java.util >-- Iterator接口：

迭代器：是一个接口。作用：用于取集合中的元素。

boolean hasNext() 如果仍有元素可以迭代，则返回 true。 E

next() 返回迭代的下一个元素。 void remove() 从迭代器指向的 collection 中移除迭代器返回的最后一个元素（可选操作）。

1) Why：

每一个集合都有自己的数据结构(就是容器中存储数据的方式)，都有特定的取出自己内部元素的方式。为了便于操作所有的容器，取出元素。将容器内部的取出方式按照一个统一的规则向外提供，这个规则就是Iterator接口。

也就说，只要通过该接口就可以取出Collection集合中的元素，至于每一个具体的容器依据自己的数据结构，如何实现的具体取出细节，这个不用关心，这样就降低了取出元素和具体集合的耦合性。

Iterator it = coll.iterator();//获取容器中的迭代器对象，至于这个对象是是什么不重要。这对象肯定符合一个规则Iterator接口。

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

public static void main(String[] args) {

Collection coll = new ArrayList();

coll.add("abc0");

coll.add("abc1");

coll.add("abc2");

//--------------方式1----------------------

Iterator it = coll.iterator();

while(it.hasNext()){

System.out.println(it.next());

}

//---------------方式2用此种----------------------

for(Iterator it = coll.iterator();it.hasNext(); ){

System.out.println(it.next());

}

}

Iterator：对collection进行迭代的迭代器.迭代器取代了Enumeration。

迭代器和枚举的区别：

迭代器允许调用者利用定义良好的语义在迭代期间从迭代器所指向的collection移除元素

方法名称得到了改进，简化书写

\*\*LisIterator：系列表迭代器，允许程序员按任一方向遍历列表、迭代期间修改列表

\*\*Comparable：此接口强行对实现它的每个类的对象进行整体自然排序。使元素具备比较性

\*\*Comparator：强行对某个对象collection进行整体排序的比较函数，使集合具备比较性

\*\*Collections：此类完全由在 collection 上进行操作或返回 collection 的静态方法组成。

\*\*Arrays：此类包含用来操作数组(比如排序和搜索)的各种静态方法

2) 获取迭代器的方法：

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

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

3) 迭代器方法：

boolean hasNext() 如果仍有元素可以迭代,则返回 true。

E next() 返回迭代的下一个元素。

void remove() 从迭代器指向的collection中移除迭代器返回的最后一个元素（可选操作）。

Attention:

\*\*迭代器的next方法是自动向下取元素，要避免出现NoSuchElementException。

也就是在迭代循环中调用一次next方法一次就要hasNext判断一次，比如语句

sop(it.next()+"..."+it.next())会发生上述异常。

\*\*迭代器的next方法返回值类型是Object，所以要记得类型转换,应用泛型后就不用强转

4) 列表迭代器：ListIterator

(1)List集合特有的迭代器ListIterator是Iterator的子接口，在迭代时，不可以通过集合对象的

方法操作集合中的元素，因为会发生ConcurrentModificationException(当方法检测到对象的并发修改，

但不允许这种修改时，抛出此异常)

(2)Iterator方法有限，只能对元素进行判断、取出和删除的操作

ListIterator可以对元素进行添加和修改动作等。

(3)获取列表迭代器方法：

ListIterator<E> listIterator() 返回此列表元素的列表迭代器（按适当顺序）。

ListIterator<E> listIterator(int index)

返回此列表中的元素的列表迭代器（按适当顺序），从列表中指定位置开始。

(4)列表迭代器方法：

void add(E e) 将指定的元素插入列表（可选操作）。

boolean hasPrevious() 如果以逆向遍历列表，列表迭代器有多个元素，则返回 true。

int nextIndex() 返回对 next 的后续调用所返回元素的索引。

E previous() 返回列表中的前一个元素。

int previousIndex() 返回对 previous 的后续调用所返回元素的索引。

void set(E e) 用指定元素替换 next 或 previous 返回的最后一个元素（可选操作）。

#### 使用集合的技巧：

看到Array就是数组结构，有角标，查询速度很快。

看到link就是链表结构：增删速度快，而且有特有方法。

addFirst； addLast； removeFirst()； removeLast()； getFirst()；getLast()；

看到hash就是哈希表，就要想要哈希值，就要想到唯一性，就要想到存入到该结构的中的元素必须覆盖

hashCode，equals方法。

看到tree就是二叉树，就要想到排序，就想要用到比较。(两种比较方法)

**比较的两种方式：**

**一个是Comparable：覆盖compareTo方法；**

**一个是Comparator：覆盖compare方法。**

 LinkedHashSet，LinkedHashMap:这两个集合可以保证哈希表有存入顺序和取出顺序一致，保证哈希表有序。

**集合什么时候用？**

当存储的是一个元素时，就用Collection。(*Collection 层次结构* 中的根接口)

当存储对象之间存在着映射关系时，就使用Map集合。

**保证唯一，就用Set。**

**不保证唯一，就用List。**

### Map集合：

|--Hashtable：底层是哈希表数据结构，是线程同步的。不可以存储null键，null值。

 |--HashMap：底层是哈希表数据结构，是线程不同步的。可以存储null键，null值。替代了Hashtable.

|--TreeMap：底层是二叉树结构，可以对map集合中的键进行指定顺序的排序。

**Map集合存储和Collection有着很大不同：**

Collection一次存一个元素；Map一次存一对元素。 Collection是单列集合；Map是双列集合。

Map中的存储的一对元素：一个是键，一个是值，键与值之间有对应(映射)关系。

特点：要保证map集合中键的唯一性。

**1，添加。**

 put(key,value)：当存储的键相同时，新的值会替换老的值，并将老值返回。如果键没有重复，返回null。 void putAll(Map);

**2，删除。**

void clear()：清空

value remove(key) ：删除指定键。

**3，判断。**

boolean isEmpty()：

boolean containsKey(key)：是否包含key

boolean containsValue(value) ：是否包含value

**4，取出。**

int size()：返回长度 value get(key) ：通过指定键获取对应的值。如果返回null，可以判断该键不存在。当然有特殊情况，就是在hashmap集合中，是可以存储null键null值的。

 Collection values()：**获取map集合中的所有的值。**

**5，想要获取map中的所有元素：**

原理：map中是没有迭代器的，collection具备迭代器，只要将map集合转成Set集合，可以使用迭代器了。之所以转成set，是因为map集合具备着键的唯一性，其实set集合就来自于map，set集合底层其实用的就是map的方法。

#### ★ 把map集合转成set的方法：

Set keySet();

Set entrySet();//取的是键和值的映射关系。

Entry就是Map接口中的内部接口；

为什么要定义在map内部呢？entry是访问键值关系的入口，是map的入口，访问的是map中的键值对。

取出map集合中所有元素的方式一：keySet()方法。

可以将map集合中的键都取出存放到set集合中。对set集合进行迭代。迭代完成，再通过get方法对获取到的键进行值的获取。

 Set keySet = map.keySet(); //把键放到set集合中

Iterator it = keySet.iterator(); //迭代器

while(it.hasNext()) {

Object key = it.next(); //取键

Object value = map.get(key);// 利用键取键值

System.out.println(key+":"+value);

}

**取出map集合中所有元素的方式二：entrySet()方法。**

Set entrySet = map.entrySet(); //把键值对放到set集合中

Iterator it = entrySet.iterator(); //迭代

while(it.hasNext()) {

Map.Entry  me = (Map.Entry)it.next(); //取键值对

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

}

\*\*Map:顶层接口,该集合存储的是键值对,键唯一,Map和Set很像,Set集合底层就是使用了Map集合。

Map集合没有迭代器，要取出元素必须先将Map集合转换成Set集合才能遍历元素

特点：要保证map集合中键的唯一性。

#### 取出方式☆☆

第一种：Set<K> keySet()

可以将map集合中的键都取出存放到set集合中。对set集合进行迭代。迭代完成，再通过get方法对获取到的键进行值的获取。

Set keySet = map.keySet();

Iterator it = keySet.iterator();

while(it.hasNext()) {

Object key = it.next();

Object value = map.get(key);

System.out.println(key+":"+value);

}

第二种：Set<Map.Entry<K,V>> entrySet()

返回此映射中包含的映射关系的Set视图，将Map集合中的映射关系存入到Set集合中，

这个映射关系的数据类型是Map.entry,再通过Map.Entry类的方法再要取出关系里面的键和值

Set entrySet = map.entrySet();

Iterator it = entrySet.iterator();

while(it.hasNext()) {

Map.Entry me = (Map.Entry)it.next();

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

}

#### Map.Entry的方法摘要：

boolean equals(Object o) 比较指定对象与此项的相等性。

K getKey() 返回与此项对应的键。

V getValue() 返回与此项对应的值。

int hashCode() 返回此映射项的哈希码值。

V setValue(V value) 用指定的值替换与此项对应的值(特有！！！)。

|--Hashtable：底层哈希表，线程同步的。不可空键值，效率低。 用作键的对象必须实现hashCode和equals方法来保证键的唯一性

#### HashMap

|--HashMap：底层哈希表，线程不同步。可null键值。替代Hashtable.效率高。

保证元素唯一性的：

原理：先判断元素的hashCode值是否相同，再判断两元素的equals方法是否为true

(往HashSet里面存的自定义元素要复写hashCode和equals方法，

以保证元素的唯一性！)

class Student {

private String name;

private int age;

public Student(String name, int age) {

super();

this.name = name;

this.age = age;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

@Override

public int hashCode(){

return name.hashCode()+age\*34;

}

@Override

public boolean equals(Object obj){

if(!(obj instanceof Student))

return false;

Student stu = (Student)obj;

return this.name.equals(stu.name)&&this.age==stu.age;

}

public class HashMapDemo1 {

public static void main(String[] args) {

Map<Student , String> hmap = new HashMap<Student , String>();

hmap.put(new Student("001",20), "beijing");

hmap.put(new Student("002",25), "hebei");

hmap.put(new Student("003",50), "hainan");

hmap.put(new Student("001",20), "beijing");

System.out.println(hmap.size());

Set<Student> keySet = hmap.keySet();

Iterator<Student> it = keySet.iterator();

while(it.hasNext()){

Student stu = it.next();

String addr = hmap.get(stu);

System.out.println(stu.getName()+".."+stu.getAge()+"::"+addr);

}

}

}

#### TreeMap

|--TreeMap：底层二叉树，可空键值，线程不同步，可以对map集合中的键进行指定顺序的排序。

TreeMap排序的第一种方式:让元素自身具备比较性，比如八种基本数据类型或则字符串，

实现Compareble接口,覆盖compareTo方法，

此方式是元素的自然顺序

TreeMap排序的第一种方式:当元素自身不具备比较性(比如存储学生对象时)或者具备的

比较性不是我们所需要的比较性时(比如想字符串的长度排序),

此时就需要让集合自身具备自定义的比较性。

那如何让集合自身具备比较性呢？可在集合初始化时，

就让集合具备比较方式。即定义一个类，

实现Comparator接口，覆盖compare方法。

class Student implements Comparable<Student>{

private String name;

private int age;

public Student(String name, int age) {

super();

this.name = name;

this.age = age;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

@Override

public int compareTo(Student stu) {

int num = new Integer(this.age).compareTo(new Integer(stu.age));

if(num==0)

return this.name.compareTo(stu.name);

return num;

}

}

public class HashMapDemo1 {

public static void main(String[] args) {

Map<Student , String> tmap = new TreeMap<Student , String>();

tmap.put(new Student("001",20), "beijing");

tmap.put(new Student("002",25), "hebei");

tmap.put(new Student("003",50), "hainan");

tmap.put(new Student("001",20), "beijing");

System.out.println(tmap.size());

Set<Student> keySet1 = tmap.keySet();

Iterator<Student> it1 = keySet1.iterator();

while(it1.hasNext()){

Student stu = it1.next();

String addr = tmap.get(stu);

System.out.println(stu.getName()+".."+stu.getAge()+"::"+addr);

}

}

}

#### Map集合和Collection集合的区别？

1、Map中一次存储是键值对（双列）。Collection中一次存储是单个元素（单列）。

2，Map的存储使用的put方法。Collection存储使用的是add方法。

3，Map集合没有迭代器，Map的取出，是将Map转成Set，在使用迭代器取出。Collection取出，使用就是迭代器。

4，如果对象很多，必须使用集合存储。如果元素存在着映射关系，可以优先考虑使用Map存储或者用数组，如果没有映射关系，可以使用Collection存储。

1，添加。

put(key,value)：当存储的键相同时，新的值会替换老的值，并将老值返回。如果键没有重复，返回null。

void putAll(Map);

2，删除。

void clear()：清空

value remove(key) ：删除指定键。

3，判断。

boolean isEmpty()：

boolean containsKey(key)：是否包含key

boolean containsValue(value) ：是否包含value

4，取出。

int size()：返回长度

value get(key) ：通过指定键获取对应的值。如果返回null，可以判断该键不存在。当然有特殊情况，就是在hashmap集合中，是可以存储null键null值的。

Collection values()：获取map集合中的所有的值。

5，想要获取map中的所有元素：

原理：map中是没有迭代器的，collection具备迭代器，只要将map集合转成Set集合，可以使用迭代器了。之所以转成set，是因为map集合具备着键的唯一性，其实set集合就来自于map，set集合底层其实用的就是map的方法。

##### Api

|--->方法摘要：

|--->添加：

V put(K key, V value) 将指定的值与此映射中的指定键关联（可选操作）。

void putAll(Map<? extends K,? extends V> m) 从指定映射中将所有映射关系复制到此映射中

|--->删除：

void clear() 从此映射中移除所有映射关系（可选操作）。

V remove(Object key) 如果存在一个键的映射关系，则将其从此映射中移除（可选操作）。

|--->判断

boolean containsKey(Object key) 如果此映射包含指定键的映射关系，则返回 true。

boolean containsValue(Object value) 如果此映射将一个或多个键映射到指定值，则返回 true。

boolean isEmpty() 如果此映射未包含键-值映射关系，则返回 true。

|--->获取

int size() 返回此映射中的键-值映射关系数。

Collection<V> values() 返回此映射中包含的值的 Collection 视图。

#### 使用集合的技巧：

看到Array就是数组结构，有角标，查询速度很快。

看到link就是链表结构：增删速度快，而且有特有方法。addFirst； addLast； removeFirst()； removeLast()； getFirst()；getLast()；

看到hash就是哈希表，就要想要哈希值，就要想到唯一性，就要想到存入到该结构的中的元素必须覆盖hashCode，equals方法。

看到tree就是二叉树，就要想到排序，就想要用到比较。

比较的两种方式：

一个是Comparable：覆盖compareTo方法；

一个是Comparator：覆盖compare方法。

LinkedHashSet，LinkedHashMap:这两个集合可以保证哈希表有存入顺序和取出顺序一致，保证哈希表有序。

#### 集合什么时候用？

当存储的是一个元素时，就用Collection。当存储对象之间存在着映射关系时，就使用Map集合。

保证唯一，就用Set。不保证唯一，就用List。

Collections：它的出现给集合操作提供了更多的功能。这个类不需要创建对象，内部提供的都是静态方法。

静态方法：

Collections.sort(list);//list集合进行元素的自然顺序排序。

Collections.sort(list,new ComparatorByLen());//按指定的比较器方法排序。

class ComparatorByLen implements Comparator<String>{

public int compare(String s1,String s2){

int temp = s1.length()-s2.length();

return temp==0?s1.compareTo(s2):temp;

}

}

Collections.max(list); //返回list中字典顺序最大的元素。

int index = Collections.binarySearch(list,"zz");//二分查找，返回角标。

Collections.reverseOrder();//逆向反转排序。

Collections.shuffle(list);//随机对list中的元素进行位置的置换。

将非同步集合转成同步集合的方法：Collections中的 XXX synchronizedXXX(XXX);

List synchronizedList(list);

Map synchronizedMap(map);

原理：定义一个类，将集合所有的方法加同一把锁后返回。

Collection 和 Collections的区别：

Collections是个java.util下的类，是针对集合类的一个工具类,提供一系列静态方法,实现对集合的查找、排序、替换、线程安全化（将非同步的集合转换成同步的）等操作。

Collection是个java.util下的接口，它是各种集合结构的父接口，继承于它的接口主要有Set和List,提供了关于集合的一些操作,如插入、删除、判断一个元素是否其成员、遍历等。

Arrays：

用于操作数组对象的工具类，里面都是静态方法。

asList方法：将数组转换成list集合。

String[] arr = {"abc","kk","qq"};

List<String> list = Arrays.asList(arr);//将arr数组转成list集合。

将数组转换成集合，有什么好处呢？用aslist方法，将数组变成集合；

可以通过list集合中的方法来操作数组中的元素：isEmpty()、contains、indexOf、set；

注意（局限性）：数组是固定长度，不可以使用集合对象增加或者删除等，会改变数组长度的功能方法。比如add、remove、clear。（会报不支持操作异常UnsupportedOperationException）；

如果数组中存储的引用数据类型，直接作为集合的元素可以直接用集合方法操作。

如果数组中存储的是基本数据类型，asList会将数组实体作为集合元素存在。

集合变数组：用的是Collection接口中的方法：toArray();

如果给toArray传递的指定类型的数据长度小于了集合的size，那么toArray方法，会自定再创建一个该类型的数据，长度为集合的size。

如果传递的指定的类型的数组的长度大于了集合的size，那么toArray方法，就不会创建新数组，直接使用该数组即可，并将集合中的元素存储到数组中，其他为存储元素的位置默认值null。

所以，在传递指定类型数组时，最好的方式就是指定的长度和size相等的数组。

将集合变成数组后有什么好处？限定了对集合中的元素进行增删操作，只要获取这些元素即可。

#### Jdk5.0新特性：

Collection在jdk1.5以后，有了一个父接口Iterable，这个接口的出现的将iterator方法进行抽取，提高了扩展性。

### **Collections：**

**它的出现给集合操作提供了更多的功能。这个类不需要创建对象，内部提供的都是静态方法。**

**静态方法：**

Collections.**sort**(list);//list集合进行元素的自然顺序排序。

Collections.sort(list,new ComparatorByLen());///按指定的比较器方法排序。

 class ComparatorByLen implements Comparator<String>{

 public int compare(String s1,String s2){

 int temp = s1.length()-s2.length();

 return temp==0?s1.compareTo(s2):temp;

}

}

Collections.max(list); //返回list中字典顺序最大的元素。

int index = Collections**.binarySearch**(list,"zz");//二分查找，返回角标。

Collections.**reverseOrder**();//逆向反转排序。

Collections.shuffle(list);//随机对list中的元素进行位置的置换。

**将非同步集合转成同步集合的方法：**

**Collections中的** XXX synchronizedXXX(XXX);

 List synchronizedList(list);

Map synchronizedMap(map);

**原理：定义一个类，将集合所有的方法加同一把锁后返回。**

**Collection 和 Collections的区别：**

Collections是个java.util下的类，是针对集合类的一个工具类,提供一系列静态方法,实现对集合的查找、排序、替换、线程安全化（将非同步的集合转换成同步的）等操作。

 Collection是个java.util下的接口，它是各种集合结构的父接口，继承于它的接口主要有Set和List,提供了关于集合的一些操作,如插入、删除、判断一个元素是否其成员、遍历等。

### Arrays：

用于操作数组对象的工具类，里面都是静态方法。

**asList方法：将数组转换成list集合。**

String[] arr = {"abc","kk","qq"}

List<String> list = Arrays.asList(arr);//将arr数组转成list集合。

**将数组转换成集合，有什么好处呢？用aslist方法，将数组变成集合；**

 可以通过list集合中的方法来操作数组中的元素：isEmpty()、contains、indexOf、set；

**注意（局限性）：**数组是固定长度，不可以使用集合对象增加或者删除等，会改变数组长度的功能方法。 比如add、remove、clear。（会报不支持操作异常UnsupportedOperationException）；

如果数组中存储的引用数据类型，直接作为集合的元素可以直接用集合方法操作。

**如果数组中存储的是 存在。**

**集合变数组：**用的是Collection接口中的方法：toArray();

如果给toArray传递的指定类型的数据长度小于了集合的size，那么toArray方法，会自定再创建一个该类型的数据，长度为集合的size。

如果传递的指定的类型的数组的长度大于了集合的size，那么toArray方法，就不会创建新数组，直接使用该数组即可，并将集合中的元素存储到数组中，其他为存储元素的位置默认值null。

所以，在传递指定类型数组时，最好的方式就是指定的长度和size相等的数组。

**将集合变成数组后有什么好处？**限定了对集合中的元素进行增删操作，只要获取这些元素即可。

#### 可变长度数组的原理：

当元素超出数组长度，会产生一个新数组，将原数组的数据复制到新数组中，再将新的元素添加到新数组中。

ArrayList：是按照原数组的50%延长。构造一个初始容量为 10 的空列表。

Vector：是按照原数组的100%延长。

注意：对于list集合，底层判断元素是否相同，其实用的是元素自身的equals方法完成的。所以建议元素都要复写equals方法，建立元素对象自己的比较相同的条件依据。

LinkedList：的特有方法。

addFirst();

addLast();

在jdk1.6以后。

offerFirst();

offerLast();

getFirst():获取链表中的第一个元素。如果链表为空，抛出NoSuchElementException;

getLast();获取链表中的最后一个元素。如果链表为空，抛出NoSuchElementException;

在jdk1.6以后。

peekFirst();获取链表中的第一个元素。如果链表为空，返回null。

peekLast();

removeFirst()：获取链表中的第一个元素，但是会删除链表中的第一个元素。如果链表为空，抛出NoSuchElementException

removeLast();

在jdk1.6以后。

pollFirst();获取链表中的第一个元素，但是会删除链表中的第一个元素。如果链表为空，返回null。

pollLast();

attention：

\*\*List集合有迭代器Iterator，还有一个特有迭代器列表ListIterator

\*\*List集合中判断元素是否相同都是用equals方法，无论contains、remove都依赖equals方法

比如往ArrayList集合里面存放学生，同名同年龄视为同一个人，此时就需要在学生类复写Object类

里面的equals方法(非常重要！！！要注意！！)

### 哈希表的原理：

1，对对象元素中的关键字(对象中的特有数据)，进行哈希算法的运算，并得出一个具体的算法值，这个值 称为哈希值。

2，哈希值就是这个元素的位置。

3，如果哈希值出现冲突，再次判断这个关键字对应的对象是否相同。如果对象相同，就不存储，因为元素重复。如果对象不同，就存储，在原来对象的哈希值基础 +1顺延。

4，存储哈希值的结构，我们称为哈希表。

5，既然哈希表是根据哈希值存储的，为了提高效率，最好保证对象的关键字是唯一的。

这样可以尽量少的判断关键字对应的对象是否相同，提高了哈希表的操作效率。

对于ArrayList集合，判断元素是否存在，或者删元素底层依据都是equals方法。

对于HashSet集合，判断元素是否存在，或者删除元素，底层依据的是hashCode方法和equals方法。

### Arrays类：

此类包含用来操作数组（比如排序和搜索）的各种方法。里面都是静态方法。

如果指定数组引用为 null，则此类中的方法都会抛出 NullPointerException。

(1)静态方法摘要：

static <T> List<T> asList(T... a)

返回一个受指定数组支持的固定大小的列表。

注意：

A:该方法将一个数组变成集合后，不可以使用集合的增删方法，因为数组的长度是固定的！

如果增删，则发生UnsupportedOprationException(不支持操作异常)

B:如果数组中的元素都是基本数据类型，则该数组变成集合时，会将该数组作为集合的一个

元素出入集合

C:如果数组中的元素都是对象，如String，那么数组变成集合后，数组中的元素就直接转成

集合中的元素

### 19、数组变集合以及集合变数组的对比：

(1)数组变集合：

方法：static <T> List<T> asList(T... a) 返回一个受指定数组支持的固定大小的列表。

好处：可以使用集合的思想和方法操作数组中的元素，数组是一个对象，但是数组中的功能很少

(2)集合变数组：

方法：Collction中的toArray方法

好处：可以限定对集合元素的操作，防止对集合的元素进行增删，因为数组长度是固定的。

### 20、Collections类和Arrays类的使用。(重点)

A:Collections

排序

二分查找

发转

B:Arrays

把数组变成字符串输出

排序

二分查找

### 对象数组(掌握)

(1)数组既可以存储基本数据类型，也可以存储引用类型。它存储引用类型的时候的数组就叫对象数组。

(2)案例：

用数组存储5个学生对象，并遍历数组。

### 3:集合(List)(掌握)

(1)List是Collection的子接口

特点：有序(存储顺序和取出顺序一致)，可重复。

(2)List的特有功能：(自己补齐)

A:添加功能

B:删除功能

C:获取功能

D:迭代器功能

E:修改功能

(3)List集合的特有遍历功能

A:由size()和get()结合。

B:代码演示

//创建集合对象

List list = new ArrayList();

//创建并添加元素

list.add("hello");

list.add("world");

list.add("java");

//遍历集合

Iterator it = list.iterator();

while(it.hasNext()) {

String s =(String) it.next();

System.out.println(s);

}

System.out.println("----------");

for(int x=0; x<list.size(); x++) {

String s =(String) list.get(x);

System.out.println(s);

}

### 列表迭代器的特有功能；(了解)

可以逆向遍历，但是要先正向遍历，所以无意义，基本不使用。

(5)并发修改异常

A:出现的现象

迭代器遍历集合，集合修改集合元素

B:原因

迭代器是依赖于集合的，而集合的改变迭代器并不知道。

C:解决方案

a:迭代器遍历，迭代器修改(ListIterator)

元素添加在刚才迭代的位置

b:集合遍历，集合修改(size()和get())

元素添加在集合的末尾

### 常见数据结构

A:栈 先进后出

B:队列 先进先出

C:数组 查询快，增删慢

D:链表 查询慢，增删快

(7)List的子类特点(面试题)

ArrayList

底层数据结构是数组，查询快，增删慢。

线程不安全，效率高。

Vector

底层数据结构是数组，查询快，增删慢。

线程安全，效率低。

LinkedList

底层数据结构是链表，查询慢，增删快。

线程不安全，效率高。

到底使用谁呢?看需求?

分析：

要安全吗?

要：Vector(即使要，也不使用这个，后面再说)

不要：ArrayList或者LinkedList

查询多；ArrayList

增删多：LinkedList

什么都不知道，就用ArrayList。

(8)List集合的案例(遍历方式 迭代器和普通for)

A:存储字符串并遍历

B:存储自定义对象并遍历

# 网络编程：

端口：

物理端口：

逻辑端口：用于标识进程的逻辑地址，不同进程的标识；有效端口：0~65535，其中0~1024系统使用或保留端口。

java 中ip对象：InetAddress.

import java.net.\*;

class IPDemo{

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

//通过名称(ip字符串or主机名)来获取一个ip对象。

InetAddress ip = InetAddress.getByName("www.baidu.com");//java.net.UnknownHostException

System.out.println("addr:"+ip.getHostAddress());

System.out.println("name:"+ip.getHostName());

}

}

## Socket：★★★★，套接字，通信的端点。

就是为网络服务提供的一种机制，通信的两端都有Socket，网络通信其实就是Socket间的通信，数据在两个Socket间通过IO传输。

## UDP传输：

1，只要是网络传输，必须有socket 。

2，数据一定要封装到数据包中，数据1包中包括目的地址、端口、数据等信息。

直接操作udp不可能，对于java语言应该将udp封装成对象，易于我们的使用，这个对象就是DatagramSocket. 封装了udp传输协议的socket对象。

因为数据包中包含的信息较多，为了操作这些信息方便，也一样会将其封装成对象。这个数据包对象就是：DatagramPacket.通过这个对象中的方法，就可以获取到数据包中的各种信息。

DatagramSocket具备发送和接受功能，在进行udp传输时，需要明确一个是发送端，一个是接收端。

udp的发送端：

1，建立udp的socket服务，创建对象时如果没有明确端口，系统会自动分配一个未被使用的端口。

2，明确要发送的具体数据。

3，将数据封装成了数据包。

4，用socket服务的send方法将数据包发送出去。

5，关闭资源。

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

import java.net.\*;

class UdpSend{

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

//1，建立udp的socket服务。

DatagramSocket ds = new DatagramSocket(8888);//指定发送端口，不指定系统会随机分配。

//2，明确要发送的具体数据。

String text = "udp传输演示 哥们来了";

byte[] buf = text.getBytes();

//3，将数据封装成了数据包。

DatagramPacket dp = new DatagramPacket(buf,

buf.length,InetAddress.getByName("10.1.31.127"),10000);

//4，用socket服务的send方法将数据包发送出去。

ds.send(dp);

//5，关闭资源。

ds.close();

}

}

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

udp的接收端：

1，创建udp的socket服务，必须要明确一个端口，作用在于，只有发送到这个端口的数据才是这个接收端可以处理的数据。

2，定义数据包，用于存储接收到数据。

3，通过socket服务的接收方法将收到的数据存储到数据包中。

4，通过数据包的方法获取数据包中的具体数据内容，比如ip、端口、数据等等。

5，关闭资源。

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

class UdpRece {

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

//1，创建udp的socket服务。

DatagramSocket ds = new DatagramSocket(10000);

//2，定义数据包，用于存储接收到数据。先定义字节数组，数据包会把数据存储到字节数组中。

byte[] buf = new byte[1024];

DatagramPacket dp = new DatagramPacket(buf,buf.length);

//3，通过socket服务的接收方法将收到的数据存储到数据包中。

ds.receive(dp);//该方法是阻塞式方法。

//4，通过数据包的方法获取数据包中的具体数据内容，比如ip，端口，数据等等。

String ip = dp.getAddress().getHostAddress();

int port = dp.getPort();

String text = new String(dp.getData(),0,dp.getLength());//将字节数组中的有效部分转成字符串。

System.out.println(ip+":"+port+"--"+text);

//5，关闭资源。

ds.close();

}

}

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

TCP传输：两个端点的建立连接后会有一个传输数据的通道，这通道称为流，而且是建立在网络基础上的流，称之为socket流。该流中既有读取，也有写入。

tcp的两个端点：一个是客户端，一个是服务端。

客户端：对应的对象，Socket

服务端：对应的对象，ServerSocket

TCP客户端：

1，建立tcp的socket服务，最好明确具体的地址和端口。这个对象在创建时，就已经可以对指定ip和端口进行连接(三次握手)。

2，如果连接成功，就意味着通道建立了，socket流就已经产生了。只要获取到socket流中的读取流和写入流即可，只要通过getInputStream和getOutputStream就可以获取两个流对象。

3，关闭资源。

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

import java.net.\*;

import java.io.\*;

//需求：客户端给服务器端发送一个数据。

class TcpClient{

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

Socket s = new Socket("10.1.31.69",10002);

OutputStream out = s.getOutputStream();//获取了socket流中的输出流对象。

out.write("tcp演示，哥们又来了!".getBytes());

s.close();

}

}

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

TCP服务端：

1，创建服务端socket服务，并监听一个端口。

2，服务端为了给客户端提供服务，获取客户端的内容，可以通过accept方法获取连接过来的客户端对象。

3，可以通过获取到的socket对象中的socket流和具体的客户端进行通讯。

4，如果通讯结束，关闭资源。注意：要先关客户端，再关服务端。

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

class TcpServer{

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

ServerSocket ss = new ServerSocket(10002);//建立服务端的socket服务

Socket s = ss.accept();//获取客户端对象

String ip = s.getInetAddress().getHostAddress();

System.out.println(ip+".....connected");

//可以通过获取到的socket对象中的socket流和具体的客户端进行通讯。

InputStream in = s.getInputStream();//读取客户端的数据，使用客户端对象的socket读取流

byte[] buf = new byte[1024];

int len = in.read(buf);

String text = new String(buf,0,len);

System.out.println(text);

//如果通讯结束，关闭资源。注意：要先关客户端，在关服务端。

s.close();

ss.close();

}

}

# 泛型：

**泛型：**jdk1.5版本以后出现的一个安全机制。表现格式：< >

## 好处：

1：将运行时期的问题ClassCastException问题转换成了编译失败，体现在编译时期，程序员就可以解决问题。

2：避免了强制转换的麻烦。

只要带有<>的类或者接口，都属于带有类型参数的类或者接口，在使用这些类或者接口时，必须给<>中传递一个具体的引用数据类型。

## 泛型技术：

其实应用在编译时期，是给编译器使用的技术，到了运行时期，泛型就不存在了。 为什么? 因为

## 泛型的擦除：

也就是说，编辑器检查了泛型的类型正确后，在生成的类文件中是没有泛型的。

**在运行时，如何知道获取的元素类型而不用强转呢？**

## 泛型的补偿：

因为存储的时候，类型已经确定了是同一个类型的元素，所以在运行时，只要获取到该元素的类型，在内部进行一次转换即可，所以使用者不用再做转换动作了。 什么时候用泛型类呢？

当类中的操作的引用数据类型不确定的时候，以前用的Object来进行扩展的，现在可以用泛型来表示。这样可以避免强转的麻烦，而且将运行问题转移到的编译时期。

## 泛型在程序定义上的体现：

**//泛型类：将泛型定义在类上。**

**class Tool<Q> {**

**private Q obj;**

**public  void setObject(Q obj) {**

**this.obj = obj; }**

**public Q getObject() { r**

**eturn obj;**

**}**

**}**

**//当方法操作的引用数据类型不确定的时候，可以将泛型定义在方法上。**

**public <W> void method(W w) {**

**System.out.println("method:"+w);**

**}**

**//静态方法上的泛型：静态方法无法访问类上定义的泛型。如果静态方法操作的引用数据类型不确定的时候，必须要将泛型定义在方法上。**

**public static <Q> void function(Q t) {**

**System.out.println("function:"+t);**

**}**

**//泛型接口.**

**interface Inter<T> { void show(T t); }**

**class InterImpl<R> implements Inter<R> {**

**public void show(R r) {**

**System.out.println("show:"+r);**

**}**

**}**

**------------------------------------------------------------**

## 泛型中的通配符：

可以解决当具体类型不确定的时候，这个通配符就是 ?  ；当操作类型时，不需要使用类型的具体功能时，只使用Object类中的功能。那么可以用 ? 通配符来表未知类型。

**泛型限定：**

上限：？extends E：可以接收E类型或者E的子类型对象。

下限：？super E：可以接收E类型或者E的父类型对象。

**上限什么时候用：**往集合中添加元素时，既可以添加E类型对象，又可以添加E的子类型对象。为什么？因为取的时候，E类型既可以接收E类对象，又可以接收E的子类型对象。

**下限什么时候用：**当从集合中获取元素进行操作的时候，可以用当前元素的类型接收，也可以用当前元素的父类型接收。

## 泛型的细节：

 1）、泛型到底代表什么类型取决于调用者传入的类型，如果没传，默认是Object类型；

1. 、使用带泛型的类创建对象时，等式两边指定的泛型必须一致； 原因：编译器检查对象调用方法时只看变量，然而程序运行期间调用方法时就要考虑对象具体类型了；
2. 3）、等式两边可以在任意一边使用泛型，在另一边不使用(考虑向后兼容)；

 ArrayList<String> al = new ArrayList<Object>();  //错

**//要保证左右两边的泛型具体类型一致就可以了，这样不容易出错。**

ArrayList<? extends Object> al = new ArrayList<String>(); al.add("aa");  //错

//因为集合具体对象中既可存储String，也可以存储Object的其他子类，所以添加具体的类型对象不合适，类型检查会出现安全问题。 ？extends Object 代表Object的子类型不确定，怎么能添加具体类型的对象呢？

public static void method(ArrayList<? extends Object> al) {

al.add("abc");  //错

**//只能对al集合中的元素调用Object类中的方法，具体子类型的方法都不能用，因为子类型不确定。 }**

## (泛型的形式

\*\*泛型类：即自定义泛型类

A：当类中要操作的引用数据类型不确定时，早起定义Object来完成扩展，现在定义泛型来完成

B：局限性：泛型类定义的泛型，在整个类中有效，如果该泛型类的方法被调用，

当泛型类的对象明确要操作的类型后，所有要操作的类型就被固定。

\*\*泛型方法：泛型放在返回值前面，修饰符的后面

A:为了避免泛型类的局限性，让不同方法可以操作不同的类型，而且类型还不确定，

则可以将泛型定义在方法上

B:特殊之处：静态方法不可以反问类上定义的泛型

如果静态方法操作的应用数据类型不确定，可以讲泛型定义在静态方法上

\*\*泛型接口：

当泛型定义在接口上时，则子类中要指定实现接口类型，同时还可以子类也可以定义为泛型类

## 泛型的高级应用：？通配符

\*\*

改为？，如Iterator<?> it=al.iterator();

\*\*两种泛型限定

向上限定： ? extends E ;E可以接收E类型或者E的子类

向下限定： ? super E ;E可以接收E类型或者E的父类

## Collections类：

(1)此类完全由在 collection 上进行操作或返回 collection 的静态方法组成。

(2)静态方法摘要：

static <T> boolean addAll(Collection<? super T> c, T... elements)

将所有指定元素添加到指定 collection 中。

static <T> void fill(List<? super T> list, T obj)

使用指定元素替换指定列表中的所有元素。

static <T> boolean replaceAll(List<T> list, T oldVal, T newVal)

使用另一个值替换列表中出现的所有某一指定值。

static void reverse(List<?> list)

反转指定列表中元素的顺序。

static <T> Comparator<T> reverseOrder()

返回一个比较器，它强行逆转实现了 Comparable 接口的对象 collection 的自然顺序

static <T> Comparator<T> reverseOrder(Comparator<T> cmp)

返回一个比较器，它强行逆转指定比较器的顺序。

(3)Collections类特牛的方法：

集合有一个共同的缺点，那就是线程不安全，被多线程操作时，容易出现问题，虽然可以自己加锁

但是麻烦。Collections提供特牛的方法，就是给它一个不同步的集合，它返回一个同步的安全的集合

static <T> Collection<T> synchronizedCollection(Collection<T> c)

返回指定 collection 支持的同步（线程安全的）collection。

static <T> List<T> synchronizedList(List<T> list)

返回指定列表支持的同步（线程安全的）列表。

static <K,V> Map<K,V> synchronizedMap(Map<K,V> m)

返回由指定映射支持的同步（线程安全的）映射。

static <T> Set<T> synchronizedSet(Set<T> s)

返回指定 set 支持的同步（线程安全的）set。

static <K,V> SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V> m)

返回指定有序映射支持的同步（线程安全的）有序映射。

static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> s)

返回指定有序 set 支持的同步（线程安全的）有序 set。

## Integer对象： ★★★☆

数字格式的字符串转成基本数据类型的方法：

1：将该字符串封装成了Integer对象，并调用对象的方法intValue();

2：使用Integer.parseInt(numstring)—>类.方法名:不用建立对象，直接类名调用；

将基本类型转成字符串：

1：Integer中的静态方法 String toString(int);

2：int+"";

将一个十进制整数转成其他进制：

转成二进制：toBinaryString

转成八进制：toOctalString

转成十六进制：toHexString

toString(int num,int radix);

将其他进制转换十进制：

parseInt(string,radix); //将给定的数转成指定的基数进制；

在jdk1.5版本后，对基本数据类型对象包装类进行升级。在升级中，使用基本数据类型对象包装类可以像使用基本数据类型一样，进行运算。

Integer i = new Integer(4); //1.5版本之前的写法；

Integer i = 4; //自动装箱，1.5版本后的写法；

i = i + 5;

//i对象是不能直接和5相加的，其实底层先将i转成int类型，在和5相加。而转成int类型的操作是隐式的。自动拆箱：拆箱的原理就是i.intValue();i+5运算完是一个int整数。如何赋值给引用类型i呢？其实有对结果进行装箱。

Integer c = 127;

Integer d = 127;

System.out.println(c = = d); //true

//在装箱时，如果数值在byte范围之内，那么数值相同，不会产生新的对象，也就是说多个数值相同的引用指向的是同一个对象。

# 常用工具类

## System

API--- java.lang.System: 属性和行为都是静态的。

long currentTimeMillis(); // 返回当前时间毫秒值

exit(); // 退出虚拟机

Properties getProperties() ; // 获取当前系统的属性信息

Properties prop = System.getProperties(); //获取系统的属性信息，并将这些信息存储到Properties集合中。

System.setProperty("myname","毕老师"); //给系统属性信息集添加具体的属性信息

//临时设置方式：运行jvm时，可以通过jvm的参数进行系统属性的临时设置，可以在java命令的后面加入 –D<name>=<value> 用法：java –Dmyname=小明 类名。

String name = System.getProperty("os.name");//获取指定属性的信息

//想要知道该系统是否是该软件所支持的系统中的一个。

Set<String> hs = new HashSet<String>();

hs.add("Windows XP");

hs.add("Windows 7");

if(hs.contains(name))

System.out.println("可以支持");

else

System.out.println("不支持");

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

## Runtime

API--- java.lang.Runtime: 类中没有构造方法，不能创建对象。

但是有非静态方法。说明该类中应该定义好了对象，并可以通过一个static方法获取这个对象。用这个对象来调用非静态方法。这个方法就是 static Runtime getRuntime();

这个Runtime其实使用单例设计模式进行设计。

class RuntimeDemo {

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

Runtime r = Runtime.getRuntime();

Process p = r.exec("notepad.exe SystemDemo.java"); //运行指定的程序

Thread.sleep(4000);

p.destroy(); //杀掉进程

}

}

## Math

API--- java.util.Math: 用于数学运算的工具类，属性和行为都是静态的。该类是final不允许继承。

static double ceil(double a) ; //返回大于指定数值的最小整数

static double floor(double a) ; //返回小于指定数值的最大整数

static long round(double a) ; //四舍五入成整数

static double pow(double a, double b) ; //a的b次幂

static double random(); //返回0~1的伪随机数

public static void main(String[] args) {

Random r = new Random();

for(int x=0; x<10; x++) {

//double d = Math.floor(Math.random()\*10+1);

//int d = (int)(Math.random()\*10+1);

int d = r.nextInt(10)+1;

System.out.println(d);

}

}

## Date

API--- java.util.Date：日期类，月份从0-11；

/\*

日期对象和毫秒值之间的转换。

1，日期对象转成毫秒值。Date类中的getTime方法。

2，如何将获取到的毫秒值转成具体的日期呢？

Date类中的setTime方法。也可以通过构造方法。

\*/

//日期对象转成毫秒值

Date d = new Date();

long time1 = d.getTime();

long time2 = System.currentTimeMillis(); / /毫秒值。

//毫秒值转成具体的日期

long time = 1322709921312l;

Date d = new Date();

d.setTime(time);

/\*

将日期字符串转换成日期对象：使用的就是DateFormat方法中的 Date parse(String source) ；

\*/

public static void method() throws Exception {

String str\_time = "2011/10/25";

DateFormat df = new SimpleDateFormat("yyyy/MM/dd"); //SimpleDateFormat作为可以指定用户自定义的格式来完成格式化。

Date d = df.parse(str\_time);

}

/\*

如果不需要使用特定的格式化风格，完全可以使用DateFormat类中的静态工厂方法获取具体的已经封装好风格的对象。getDateInstance();getDateTimeInstance();

\*/

Date d = new Date();

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

df = DateFormat.getDateTimeInstance(DateFormat.LONG,DateFormat.LONG);

String str\_time = df.format(d);

//将日期对象转换成字符串的方式：DateFormat类中的format方法。

//创建日期格式对象。

DateFormat df = new SimpleDateFormat(); //该对象的建立内部会封装一个默认的日期格式。11-12-1 下午1:48

//如果想要自定义日期格式的话。可使用SimpleDateFormat的构造方法。将具体的格式作为参数传入到构造方法中。如何表示日期中年的部分呢？可以必须要参与格式对象文档。

df = new SimpleDateFormat("yyyy年MM月dd日 HH:mm:ss");

//调用DateFormat中的format方法。对已有的日期对象进行格式化。

String str\_time = df.format(d);

## Calendar类

API--- java.util. Calendar：日历类

public static void method(){

Calendar c = Calendar.getInstance();

System.out.println(c.get(Calendar.YEAR)+"年"+(c.get(Calendar.MONTH)+1)+"月"

+getNum(c.get(Calendar.DAY\_OF\_MONTH))+"日"

+"星期"+getWeek(c.get(Calendar.DAY\_OF\_WEEK)));

}

public static String getNum(int num){

return num>9 ? num+"" : "0"+num;

}

public static String getWeek(int index){

/\*

查表法：建立数据的对应关系.

最好：数据个数是确定的，而且有对应关系。如果对应关系的一方，是数字，而且可以作为角标，那么可以通过数组来作为表。

\*/

String[] weeks = {"","日","一","二","三","四","五","六"};

return weeks[index];

}

## Math类的使用(重点)

(1)数学操作类:该类没有构造函数，方法均为静态的

(2)掌握内容

A:成员变量

\*\*E：比任何其他值都更接近e（即自然对数的底数）的double值。

\*\*PI：比任何其他值都更接近pi（即圆的周长与直径之比）的double值。

B:成员方法

\*\*static double abs(double a)

返回 double 值的绝对值。返回绝对值

\*\*static double ceil(double a)

返回最小的（最接近负无穷大）double 值，该值大于等于参数，并等于某个整数。

\*\*static double floor(double a)

返回最大的（最接近正无穷大）double 值，该值小于等于参数，并等于某个整数。

\*\*max：返回两个值中较大的那个

\*\*min：返回两个值中较小的那个

\*\*static long round(double a) 返回最接近参数的 long。

static int round(float a) 返回最接近参数的 int。

\*\*static double random()

返回带正号的 double 值，该值大于等于 0.0 且小于 1.0。

\*\*static double pow(double a, double b)

返回第一个参数的第二个参数次幂的值。

\*\*static double sqrt(double a)

返回正确舍入的 double 值的正平方根。

## Random类的使用(重点)

(1)产生随机数的类

(2)掌握内容

A:构造方法

\*\*Random() 创建一个新的随机数生成器。

\*\*Random(long seed) 使用单个 long 种子创建一个新的随机数生成器。

B:成员方法

\*\*int nextInt() 返回下一个伪随机数，它是此随机数生成器的序列中均匀分布的 int 值。

\*\*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。

Scanner类的使用

(1)可以获取从键盘的输入数据

(2)掌握内容

构造方法：

Scanner(InputStream source) 构造一个新的 Scanner，它生成的值是从指定的输入流扫描的。

如：Scanner sc = new Scanner(System.in);

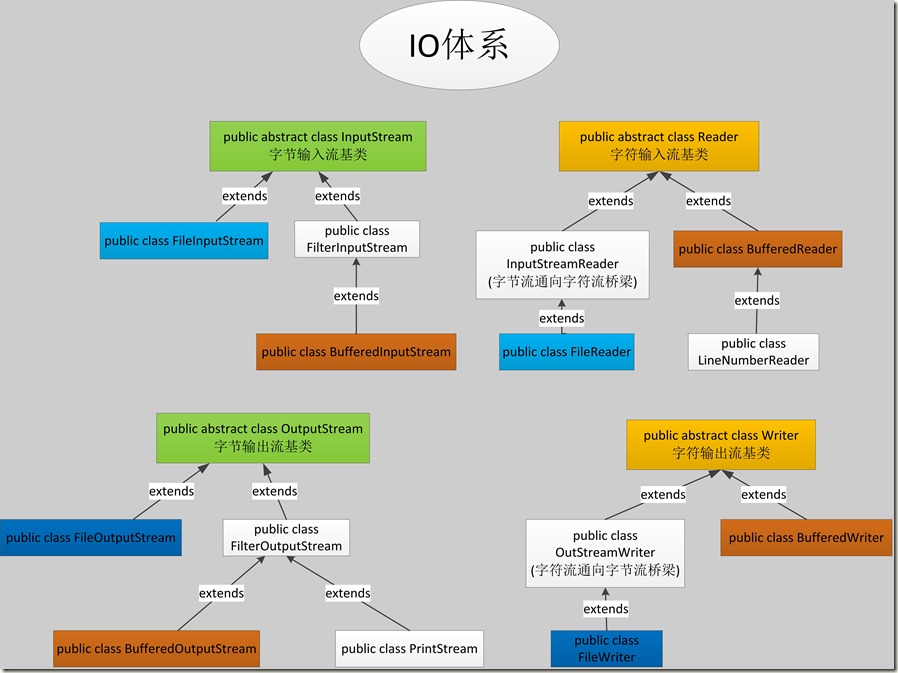
方法摘要

sc.nextInt();获取整型数据

sc.nextLine();获取字符串数据

# IO流

 以下这个图要求自己能够独立画出来并理解（老师有给我们总结，放大图片看）



**字符流：**

Reader：用于读取字符流的抽象类。子类必须实现的方法只有 read(char[], int, int) 和

close()。

 |---BufferedReader：从字符输入流中读取文本，缓冲各个字符，从而实现字符、数组和行的高效读取。 可以指定缓冲区的大小，或者可使用默认的大小。大多数情况下，默认值就足够大了。

  |---LineNumberReader：跟踪行号的缓冲字符输入流。此类定义了方法 **setLineNumber(int) 和 getLineNumber()，**它们可分别用于设置和获取当前行号。

     |---InputStreamReader：是字节流通向字符流的桥梁：它使用指定的 charset 读取字节并将其解码为字符。它使用的字符集可以由名称指定或显式给定，或者可以接受平台默认的字符集。

        |---FileReader：用来读取字符文件的便捷类。此类的构造方法假定默认字符编码和默认字节缓冲区大小都是适当的。要自己指定这些值，可以先在 FileInputStream 上构造一个 InputStreamReader。

  |---CharArrayReader：

**|---StringReader：**

**Writer：写入字符流的抽象类。子类必须实现的方法仅有 write(char[], int, int)、flush() 和**

close()。

     |---BufferedWriter：将文本写入字符输出流，缓冲各个字符，从而提供单个字符、数组和字符串的高效写入。

     |---OutputStreamWriter：是字符流通向字节流的桥梁：可使用指定的 charset 将要写入流中的字符编码成字节。它使用的字符集可以由名称指定或显式给定，否则将接受平台默认的字符集。

  |---FileWriter：用来写入字符文件的便捷类。此类的构造方法假定默认字符编码和默认字节缓冲区大小都是可接受的。要自己指定这些值，可以先在 FileOutputStream 上构造一个 OutputStreamWriter。

  |---PrintWriter：

     |---CharArrayWriter：

   |---StringWriter：

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

**字节流：**

**InputStream：**是表示字节输入流的所有类的超类。

     |--- FileInputStream：从文件系统中的某个文件中获得输入字节。哪些文件可用取决于主机环境。FileInputStream 用于读取诸如图像数据之类的原始字节流。要读取字符流，请考虑使用 FileReader。

     |--- FilterInputStream：包含其他一些输入流，它将这些流用作其基本数据源，它可以直接传输数据或提供一些额外的功能。

        |--- BufferedInputStream：该类实现缓冲的输入流。

 |--- Stream：

     |--- ObjectInputStream：

  |--- PipedInputStream：

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

**OutputStream：**此抽象类是表示输出字节流的所有类的超类。

     |--- FileOutputStream：文件输出流是用于将数据写入 File 或 FileDescriptor 的输出流。

     |--- FilterOutputStream：此类是过滤输出流的所有类的超类。

   |--- BufferedOutputStream：该类实现缓冲的输出流。

   |--- PrintStream：

        |--- DataOutputStream：

 |--- ObjectOutputStream：

  |--- PipedOutputStream：

 --------------------------------

缓冲区是提高效率用的，给谁提高呢？

**BufferedWriter：**是给字符输出流提高效率用的，那就意味着，缓冲区对象建立时，必须要先有流对象。明确要提高具体的流对象的效率。

 FileWriter fw = new FileWriter("bufdemo.txt");

BufferedWriter bufw = new BufferedWriter(fw);//**让缓冲区和指定流相关联。**

**for(int x=0; x<4; x++){**

**bufw.write(x+"abc");**

**bufw.newLine(); //写入一个换行符，这个换行符可以依据平台的不同写入不同的换行符。**

**bufw.flush();//对缓冲区进行刷新，可以让数据到目的地中。**

**}**

**bufw.close();//关闭缓冲区，其实就是在关闭具体的流。**

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

**BufferedReader：**

**FileReader fr = new FileReader("bufdemo.txt");**

**BufferedReader bufr  = new BufferedReader(fr);**

**String line = null;**

**while((line=bufr.readLine())!=null){  //readLine方法返回的时候是不带换行符的。**

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

**}**

**bufr.close();**

//记住，只要一读取键盘录入，就用这句话。

//接收字节在转换成字符

**BufferedReader bufr = new BufferedReader(new InputStreamReader(System.in));**

**BufferedWriter bufw = new BufferedWriter(new OutputStreamWriter(System.out));//**

**输出到控制台**

**String line = null;**

**while((line=bufr.readLine())!=null){**

**if("over".equals(line))**

**break;**

**bufw.write(line.toUpperCase());//将输入的字符转成大写字符输出**

**bufw.newLine();**

**bufw.flush();**

**}**

**bufw.close();**

**bufr.close();**

流对象：其实很简单，就是读取和写入。但是因为功能的不同，流的体系中提供N多的对象。那么开始时，到底该用哪个对象更为合适呢？这就需要明确流的操作规律。

流的操作规律：

 1，明确源和目的。

数据源：就是需要读取，可以使用两个体系：InputStream、Reader； 数据汇：就是需要写入，可以使用两个体系：OutputStream、Writer；

 2，操作的数据是否是纯文本数据？

 如果是：数据源：Reader

数据汇：Writer

如果不是：数据源：InputStream

 数据汇：OutputStream

3，虽然确定了一个体系，但是该体系中有太多的对象，到底用哪个呢？

明确操作的数据设备。

数据源对应的设备：硬盘(File)，内存(数组)，键盘(System.in)

数据汇对应的设备：硬盘(File)，内存(数组)，控制台(System.out)。

 4，需要在基本操作上附加其他功能吗？比如缓冲。

如果需要就进行装饰。

转换流特有功能：转换流可以将字节转成字符，原因在于，将获取到的字节通过查编码表获取到指定对应字符。

**转换流的最强功能就是基于 字节流 + 编码表 。没有转换，没有字符流。**

 发现转换流有一个子类就是操作文件的字符流对象：

**InputStreamReader**

**|--FileReader**

**OutputStreamWriter**

**|--FileWrier**

想要操作文本文件，必须要进行编码转换，而编码转换动作转换流都完成了。所以操作文件的流对象只要继承自转换流就可以读取一个字符了。

**但是子类有一个局限性，就是子类中使用的编码是固定的，是本机默认的编码表，对于简体中文版的系统默认码表是**GBK。 FileReader fr = new FileReader("a.txt");

InputStreamReader isr = new InputStreamReader(new FileInputStream("a.txt"),"gbk"); 以上两句代码功能一致，

如果仅仅使用平台默认码表，就使用FileReader fr = new FileReader("a.txt"); //因为简化。 如果需要制定码表，必须用转换流。

 转换流 = 字节流+编码表。

转换流的子类File = 字节流 + 默认编码表。

**凡是操作设备上的文本数据，涉及编码转换，必须使用转换流。**

## IO流概述

(1)用来处理设备(硬盘，控制台，内存)间的数据。

(2)java中对数据的操作都是通过流的方式。

(3)java用于操作流的类都在io包中。

(4)按照流操作的数据的类型不同：分为字节流和字符流。字符流是为了方便中文的操作而来的。

(5)按照流的流向不同分为：输入流，输出流

## IO流常用基类：

### 字节流

输出字节流：OutputStream：字节写入流抽象类

|--->FileOutputStream：

字节写入流

|--->BufferedOutputStream：

字节写入流缓冲区

|--->PrintStream：

打印流

输入字节流：InputStream：字节读取流抽象类

|--->FileInputStream：

字节读取流

|--->BufferedInputStream：

字节读取流缓冲区

### 字符流

输出字符流：Writer：字符写入流的抽象

|--->FileWriter：

字符写入流

|--->BufferedWriter：

字符写入流缓冲区

|--->OutputStreamWriter：

字符通向字节的转换流(涉及键盘录入时用)

|--->OutputStreamWriter：

打印流，可处理各种类型的数据

输入字符流：Reader: 字符读取流的抽象类

|--->FileReader：

字符读取流

|--->LineNumberReader：

跟踪行号的缓冲字符读取流

|--->BufferedReader：

字符读取流缓冲区

|--->InputStreamReader：

字节通向字符的转换流(涉及键盘录入时用)

### IO流常用基类方法摘要：

\*\*字节写入流：OutputStream：

void close() 关闭此输出流并释放与此流有关的所有系统资源。

void flush()刷新此输出流并强制写出所有缓冲的输出字节。

abstract void write(int b) 将指定的字节写入此输出流。

void write(byte[] b) 将 b.length 个字节从指定的 byte 数组写入此输出流。

void write(byte[] b, int off, int len)

将指定 byte 数组中从偏移量 off 开始的 len 个字节写入此输出流。

\*\*字节读取流：InputStream：

void close() 关闭此输入流并释放与该流关联的所有系统资源。

int available() (特有方法！！)

返回此输入流下一个方法调用可以不受阻塞地从此输入流读取（或跳过）的估计字节数。

abstract int read() 从输入流中读取数据的下一个字节。

int read(byte[] b) 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。

int read(byte[] b, int off, int len) 将输入流中最多 len 个数据字节读入 byte 数组。

long skip(long n) 跳过和丢弃此输入流中数据的 n 个字节。

### \*\*字符写入流：Writer：

abstract void close() 关闭此流，但要先刷新它。

abstract void flush() 刷新该流的缓冲。

void write(int c) 写入单个字符。

void write(char[] cbuf) 写入字符数组。

abstract void write(char[] cbuf, int off, int len) 写入字符数组的某一部分。

void write(String str) 写入字符串。

void write(String str, int off, int len) 写入字符串的某一部分。

### \*\*字符读取流：Reader：

abstract void close() 关闭该流并释放与之关联的所有资源。

int read() 读取单个字符。

int read(char[] cbuf) 将字符读入数组

abstract int read(char[] cbuf, int off, int len) 将字符读入数组的某一部分。

long skip(long n) 跳过字符。

## IO流常用字节流基类的子类：

### 写入流：

#### FileOutputStream：

\*\*构造方法：

FileOutputStream(String name)

创建一个向具有指定名称的文件中写入数据的输出文件流。

FileOutputStream(String name, boolean append)

创建一个向具有指定 name 的文件中写入数据的输出文件流。

FileOutputStream(File file)

创建一个向指定 File 对象表示的文件中写入数据的文件输出流。

FileOutputStream(File file, boolean append)

创建一个向指定 File 对象表示的文件中写入数据的文件输出流。

\*\*方法摘要：

public void flush()

void close() 关闭此文件输出流并释放与此流有关的所有系统资源。

void write(int b) 将指定字节写入此文件输出流。

void write(byte[] b, int off, int len)

将指定 byte 数组中从偏移量 off 开始的 len 个字节写入此文件输出流。

void write(int b) 将指定字节写入此文件输出流。

#### (2)BufferedOutputStream：

\*\*构造方法：

BufferedOutputStream(OutputStream out)

创建一个新的缓冲输出流，以将数据写入指定的底层输出流。

BufferedOutputStream(OutputStream out, int size)

创建一个新的缓冲输出流，以将具有指定缓冲区大小的数据写入指定的底层输出流。

\*\*方法摘要：

void flush() 刷新此缓冲的输出流。

void write(byte[] b, int off, int len)

将指定 byte 数组中从偏移量 off 开始的 len 个字节写入此缓冲的输出流。

void write(int b) 将指定的字节写入此缓冲的输出流。

#### (3)PrintStream：打印流，可将各种类型的数据原样打印，有自动刷新功能

\*\*构造方法：

PrintStream(String fileName)

创建具有指定文件名称且不带自动行刷新的新打印流。

PrintStream(File file)

创建具有指定文件且不带自动行刷新的新打印流。

PrintStream(OutputStream out)

创建新的打印流。

PrintStream(OutputStream out, boolean autoFlush) (当autoFlush为true时具有自动刷新功能)

创建新的打印流。

\*\*方法摘要：

PrintStream append(char c)

将指定字符添加到此输出流。

void close()

关闭流。

void flush()

刷新该流的缓冲。

void print(各种类型的数据：)

打印各种类型的数据

void println(各种类型的数据：)：自动换行

打印各种类型的数据

void write(byte[] buf, int off, int len)

将 len 字节从指定的初始偏移量为 off 的 byte 数组写入此流。

void write(int b)

将指定的字节写入此流。

### \*\*读取流：

#### (1)FileInputStream：

\*\*构造方法：

FileInputStream(String name)

通过打开一个到实际文件的连接来创建一个 FileInputStream，

该文件通过文件系统中的路径名 name 指定。

FileInputStream(File file)

通过打开一个到实际文件的连接来创建一个 FileInputStream，

该文件通过文件系统中的 File 对象 file 指定。

\*\*方法摘要：

int available() (字节读取流特有方法！！！)

返回下一次对此输入流调用的方法可以不受阻塞地从此输入流读取（或跳过）的估计剩余字节数。

int read()

从此输入流中读取一个数据字节。

int read(byte[] b)

从此输入流中将最多 b.length 个字节的数据读入一个 byte 数组中。

int read(byte[] b, int off, int len)

从此输入流中将最多 len 个字节的数据读入一个 byte 数组中。

long skip(long n)

从输入流中跳过并丢弃 n 个字节的数据。

#### (2)BufferedInputStream:

\*\*构造方法：

BufferedInputStream(InputStream in)

创建一个 BufferedInputStream 并保存其参数，即输入流 in，以便将来使用。

BufferedInputStream(InputStream in, int size)

创建具有指定缓冲区大小的 BufferedInputStream 并保存其参数，即输入流 in，以便将来使用。

\*\*方法摘要：

int available() (字节读取流特有方法！！！)

返回可以从此输入流读取（或跳过）、且不受此输入流接下来的方法调用阻塞的估计字节数。

int read()

参见 InputStream 的 read 方法的常规协定。

int read(byte[] b, int off, int len)

从此字节输入流中给定偏移量处开始将各字节读取到指定的 byte 数组中。

long skip(long n)

参见 InputStream 的 skip 方法的常规协定。

## 字符流常用基类的子类

\*\*写入流：

(1)FileWriter:

\*\*构造方法：

FileWriter(String fileName)

根据给定的文件名构造一个 FileWriter 对象。

FileWriter(String fileName, boolean append)

根据给定的文件名以及指示是否附加写入数据的 boolean 值来构造 FileWriter 对象。

FileWriter(File file)

根据给定的 File 对象构造一个 FileWriter 对象。

FileWriter(File file, boolean append)

根据给定的 File 对象构造一个 FileWriter 对象。

FileWriter(FileDescriptor fd)

构造与某个文件描述符相关联的 FileWriter 对象。

\*\*方法摘要：跟Writer一样

abstract void close() 关闭此流，但要先刷新它。

abstract void flush() 刷新该流的缓冲。

void write(int c) 写入单个字符。

void write(char[] cbuf) 写入字符数组。

abstract void write(char[] cbuf, int off, int len) 写入字符数组的某一部分。

void write(String str) 写入字符串。

void write(String str, int off, int len) 写入字符串的某一部分。

(2)BufferedWriter:

\*\*构造方法：

BufferedWriter(Writer out)

创建一个使用默认大小输出缓冲区的缓冲字符输出流。

BufferedWriter(Writer out, int sz)

创建一个使用给定大小输出缓冲区的新缓冲字符输出流。

\*\*方法摘要：

void close()

关闭此流，但要先刷新它。

void flush()

刷新该流的缓冲。

void newLine()

写入一个行分隔符。

void write(char[] cbuf, int off, int len)

写入字符数组的某一部分。

void write(int c)

写入单个字符。

void write(String s, int off, int len)

写入字符串的某一部分。

(3)OutputStreamWriter：字节通向字符的转换流

\*\*构造方法：

OutputStreamWriter(OutputStream out)

创建使用默认字符编码的 OutputStreamWriter。

\*\*方法摘要：

void write(char[] cbuf, int off, int len)

写入字符数组的某一部分。

void write(int c)

写入单个字符。

void write(String str, int off, int len)

写入字符串的某一部分。

(4)PrintWriter:

\*\*构造方法：

PrintWriter(String fileName)

创建具有指定文件名称且不带自动行刷新的新 PrintWriter。

PrintWriter(File file)

使用指定文件创建不具有自动行刷新的新 PrintWriter。

PrintWriter(Writer out)

创建不带自动行刷新的新 PrintWriter。

PrintWriter(Writer out, boolean autoFlush)

创建新 PrintWriter。

PrintWriter(OutputStream out)

根据现有的 OutputStream 创建不带自动行刷新的新 PrintWriter。

PrintWriter(OutputStream out, boolean autoFlush)

通过现有的 OutputStream 创建新的 PrintWriter。

\*\*方法摘要：

PrintWriter append(char c)

将指定字符添加到此 writer。

void print(各种类型的数据：)

打印各种类型的数据

void println(各种类型的数据：)：自动换行

打印各种类型的数据

void write(char[] buf)

写入字符数组。

void write(char[] buf, int off, int len)

写入字符数组的某一部分。

void write(int c)

写入单个字符。

void write(String s)

写入字符串。

void write(String s, int off, int len)

写入字符串的某一部分。

\*\*读取流：

(1)FileReader:

\*\*构造方法：

FileReader(String fileName)

在给定从中读取数据的文件名的情况下创建一个新 FileReader。

FileReader(File file)

在给定从中读取数据的 File 的情况下创建一个新 FileReader。

FileReader(FileDescriptor fd)

在给定从中读取数据的 FileDescriptor 的情况下创建一个新 FileReader。

\*\*方法摘要：和Reader基类方法一致：

abstract void close() 关闭该流并释放与之关联的所有资源。

int read() 读取单个字符。

int read(char[] cbuf) 将字符读入数组

abstract int read(char[] cbuf, int off, int len) 将字符读入数组的某一部分。

long skip(long n) 跳过字符。

(2)BufferedReader:

\*\*构造方法：

BufferedReader(Reader in)

创建一个使用默认大小输入缓冲区的缓冲字符输入流。

\*\*方法摘要：

int read()

读取单个字符。

int read(char[] cbuf, int off, int len)

将字符读入数组的某一部分。

String readLine()

读取一个文本行。

(3)InputStreamReader：字符通向字节的桥梁：

\*\*构造方法：

InputStreamReader(InputStream in)

创建一个使用默认字符集的 InputStreamReader。

\*\*方法摘要:

int read() 读取单个字符。

int read(char[] cbuf) 将字符读入数组

abstract int read(char[] cbuf, int off, int len) 将字符读入数组的某一部分。

long skip(long n) 跳过字符。

(4)LineNumberReader：

\*\*构造方法：

LineNumberReader(Reader in)

使用默认输入缓冲区的大小创建新的行编号 reader。

\*\*方法摘要：

int read()

读取单个字符。

int read(char[] cbuf, int off, int len)

将字符读入数组中的某一部分。

String readLine()

读取文本行。

long skip(long n)

跳过字符。

int getLineNumber()

获得当前行号。

void setLineNumber(int lineNumber)

设置当前行号。

6、IO流常见需求：

\*\*\*\*字符流：

(1)需求1：在硬盘上创建一个文件并写入信息

用字符写入流：FileWriter

FileWriter fw = new FileWriter("g:\\filewriter.txt");

fw.write("输入信息");

fw.write("也可以写入字符数组".toCharArray());

fw.flush();

fw.close();

(2)需求2：在原有文件上续写数据

FileWriter fw = new FileWriter("g:\\filewriter.txt",true);

fw.write("还可以续写信息");

fw.write("也可以写入字符数组".toCharArray());

fw.flush();

fw.close();

(3)需求3：读取硬盘上的文本文件，并将数据打印在控制台

FileReader fr = new FileReader("g:\\filewriter.txt");

\*\*第一种读取方法：一个一个字节的读

int ch = 0;

ch = fr.read();

sop((char)ch);

fr.close();

\*\*第二种读取方法：利用数组来提高效率

char[] buf = new char[1024];

int len = 0;

while((len = fr.read(buf))!=-1)

{

sop(new String(buf,0,len));

}

fr.close();

(4)需求4:拷贝文本文件

利用缓冲区提高数据读写效率

(无缓冲区就相当于一滴一滴的喝水，有缓冲区就相当于一杯一杯的喝水)

BufferedReader bufr = new BufferedReader(new FileReader("g:\\filewriter.txt"));

BufferedWriter bufw = new BufferedWriter(new FileWriter("d:\\copyfilewriter.txt"));

String line = null;

while((line = bufr.readLine())!=null)

{

burw.write(line);

bufw.newLine();

bufw.flush();

}

bufr.close();

bufw.close();

\*\*\*\*字节流：字节流写入时没有刷新

(1)需求1：在硬盘上创建一个文件并写入信息(字节流写入时没有刷新)

FileOutputStream fos = new FileOutputStream("g:\\filestream.txt");

fos.write(97);//写入一个字节,int：97代表写入char：a

fos.write("也可以写入字节数组".getBytes());//通常使用此种方式写入，直观！

fos.close();

(2)需求2：在硬盘已有文件上续写数据(字节流写入时没有刷新)

FileOutputStream fos = new FileOutputStream("g:\\filestream.txt",true);

fos.write("创建字节写入流时，传进去一个true参数就可以继续写入信息".getBytes());

fos.close();

(3)需求3：读取硬盘上的文件

FileInputStream fis = new FileInputStream("g:\\filestream.txt");

\*\*第一种读法：一个字节一个字节的读(此种读法慢)

int ch = 0;

while((ch = fis.read())!=-1)

{

sop((char)ch);

}

\*\*第一种读法：利用字节数组读(此种读法效率有一定提高)

byte[] buf = new byte[1024];

int len = 0;

while((len = fis.read())!=-1)

{

sop(new String(buf,0,len));

}

(4)需求4:拷贝字节文件，如图片或者MP3或者电影

\*\*第一种拷贝：不带缓冲区(慢，还是效率问题)

FileInputStream fis = new FileInputStream("g:\\1.mp3");

FileOutputStream fos = new FileOutputStream("g:\\copy1.mp3");

byte[] buf = new byte[1024];

int len = 0;

while((len = fis.read(buf))!=-1)

{

fos.(buf,0,len);//字节流写入无需刷新

}

fis.close();

fos.close();

\*\*第二种拷贝：带缓冲区，高效

BufferedInputStream bufi = new BufferedInputStream(new FileInputStream("g:\\1.mp3"));

BufferedOutputStream bufo = new BufferedOutputStream(new FileOutputStream("g:\\copy1.mp3"));

int ch = 0;

while((ch = bufi.read())!=-1)

{

bufo.write(ch);

}

bufi.close();

bufo.close();

\*\*\*\*转换流：

(1)需求1：读取一个键盘录入

InputStream in = System.in;//创建一个键盘录入流，流不关则可以一直录入

int by1 = in.read();//一次读一个字节

int by2 = in.read();//一次读一个字节

sop(by1);//假设键盘录入的是abcd,则打印a

sop(by2);//假设键盘录入的是abcd,则打印b

in.close();

(2)需求2：键盘录入一行数据打印一行数据，如果录入的是over则结束录入

InputStream in = System.in;

StringBuilder sb = new StringBuilder();

while(true)

{

int ch = in.read();

if(ch=='\r')

continue;

if(ch=='\n')

{

String line = sb.toString();

if("over".equals(line))

break;

sop(line.toUpperCase());//输出大写

sb.delete(0.sb.length());//清除上一行录入的数据

}

else

sb.append((char)ch);

}

in.close();

(3)需求3：发现需求2中其实就是读一行的原理，故引入字节通向字符的桥梁：InputStreamReader

为提高效率加入缓冲区：

BufferedReader bufr = new BufferedReader(new InputStreamReader(System.in));

String line = null;

while((line = bufr.readLine())!=null)

{

if("over".equals(line))

break;

sop(line.toUpperCase());//输出大写

}

bufr.close();

(4)需求4：键盘录入数据并打印到控制台

BufferedReader bufr = new BufferedReader(new InputStreamReader(System.in));

BufferedWriter bufw = new BufferedWriter(new OntputStreamWriter(System.out));

String line = null;

while((line = bufr.readLine())!=null)

{

if("over".equals(line))

break;

bufw.write(line.toUpperCase());

bufw.newLine();

bufw.flush();

}

bufr.close();

bufw.close();

(5)需求5:将键盘录入的数据存储到硬盘文件

则只需将(4)中的

BufferedWriter bufw = new BufferedWriter(new OntputStreamWriter(System.out));

改为：

BufferedWriter bufw = new BufferedWriter(new OntputStreamWriter(new FileWriter("g:\\demo.txt")));

即：

BufferedReader bufr = new BufferedReader(new InputStreamReader(System.in));

BufferedWriter bufw = new BufferedWriter(new OntputStreamWriter(new FileWriter("g:\\demo.txt")));

String line = null;

while((line = bufr.readLine())!=null)

{

if("over".equals(line))

break;

bufw.write(line.toUpperCase());

bufw.newLine();

bufw.flush();

}

bufr.close();

bufw.close();

(6)需求6：将硬盘文件的数据打印到控制台

则只需将(4)中的

BufferedReader bufr = new BufferedReader(new InputStreamReader(System.in));

改为：

BufferedReader bufr = new BufferedReader(new InputStreamReader(new FileReader("g:\\demo.txt")));

即：

BufferedReader bufr = new BufferedReader(new InputStreamReader(new FileReader("g:\\demo.txt")));

BufferedWriter bufw = new BufferedWriter(new OntputStreamWriter(System.out));

String line = null;

while((line = bufr.readLine())!=null)

{

if("over".equals(line))

break;

bufw.write(line.toUpperCase());

bufw.newLine();

bufw.flush();

}

bufr.close();

bufw.close();

7、流操作的规律：

\*\*\*\*流操作的难点：流对象很多，不知道具体用哪个

\*\*\*\*规律：

(1)第一步：先明确源和目的

源：

文本：用Reader

字节：用InputStream

目的：

文本：用Writer

字节：用OutputStream

(2)第二步：明确是不是纯文本

是：用字符流；

不是：用字节流

(3)第三步：明确流体系后，通过设备来明确具体使用哪个流对象

源设备：

键盘：System.in

硬盘：文件流File

内存：数组流ArrayStream

目的设备：

键盘：System.out

硬盘：文件流File

内存：数组流ArrayStream

8、File类

构造方法：

File(String pathname)

通过将给定路径名字符串转换为抽象路径名来创建一个新 File 实例。

File(String parent, String child)

根据 parent 路径名字符串和 child 路径名字符串创建一个新 File 实例。

File(File parent, String child)

根据 parent 抽象路径名和 child 路径名字符串创建一个新 File 实例。

方法摘要：

(1)创建：

boolean createNewFile()

当且仅当不存在具有此抽象路径名指定名称的文件时，不可分地创建一个新的空文件。

boolean mkdir()

创建一级文件夹

boolean mkdirs()

创建多级文件夹

(判断)：

boolean canExecute()

测试应用程序是否可以执行此抽象路径名表示的文件。

boolean canRead()

测试应用程序是否可以读取此抽象路径名表示的文件。

boolean canWrite()

测试应用程序是否可以修改此抽象路径名表示的文件。

int compareTo(File pathname)

按字母顺序比较两个抽象路径名。

boolean isAbsolute()

测试此抽象路径名是否为绝对路径名。

boolean isDirectory()

测试此抽象路径名表示的文件是否是一个目录。

boolean isFile()

测试此抽象路径名表示的文件是否是一个标准文件。

boolean isHidden()

测试此抽象路径名指定的文件是否是一个隐藏文件。

boolean exists()

测试此抽象路径名表示的文件或目录是否存在。

(3)获取：

String getParent()

返回此抽象路径名父目录的路径名字符串；如果此路径名没有指定父目录，则返回 null。

File getParentFile()

返回此抽象路径名父目录的抽象路径名；如果此路径名没有指定父目录，则返回 null。

String getName()

返回由此抽象路径名表示的文件或目录的名称。

String getPath()

将此抽象路径名转换为一个路径名字符串。

String getAbsolutePath()

返回此抽象路径名的绝对路径名字符串。

File getAbsoluteFile()

返回此抽象路径名的绝对路径名形式。

(4)删除：

boolean delete()

删除此抽象路径名表示的文件或目录。

oid deleteOnExit()

在虚拟机终止时，请求删除此抽象路径名表示的文件或目录。

(5)获取全部：(非常重要！！！)

String[] list()

返回一个字符串数组，这些字符串指定此抽象路径名表示的目录中的文件和目录。

String[] list(FilenameFilter filter)

返回一个字符串数组，这些字符串指定此抽象路径名表示的目录中满足指定过滤器的文件和目录。

File[] listFiles()

返回一个抽象路径名数组，这些路径名表示此抽象路径名表示的目录中的文件。

File[] listFiles(FileFilter filter)

返回抽象路径名数组，这些路径名表示此抽象路径名表示的目录中满足指定过滤器的文件和目录。

\*\*\*\*FilenameFilter接口只有一个方法：

boolean accept(File dir, String name)

测试指定文件是否应该包含在某一文件列表中。

\*\*\*\*FileFilter接口只有一个方法：

boolean accept(File dir, String name)

测试指定文件是否应该包含在某一文件列表中。

8、File类常见需求：

(1)文件名过滤:列出给定目录的所有.java文件

public void showFileName(File file)

{

String[] filenames = file.list(new FilenameFilter()//匿名内部类

{

public boolean accept(File dir,String name)//复写唯一方法

{

return name.endsWith(".java");//列出所有.java文件

}

});

}

(2)列出指定目录下的所有文件和文件夹(递归)

\*\*示例1：不带层次递归：

public static void showDir(File dir)

{

File[] files = dir.listFile();

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

{

if(files[i].isDirectory&&!files[i].isHidden())

showDir(files[i]);

else

sop(files[i]);

}

}

\*\*示例2：带层次递归：

public static void showDir(File dir,int level)

{

sop(getLevel(level)+C);//进来先打印层次和目录

level++;

File[] files = dir.listFile();

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

{

if(files[i].isDirectory&&!files[i].isHidden())

showDir(files[i]);

else

sop(getLevel(level)+files[i]);//是文件就打印层次和目录

}

}

public static String getLevel(int level)

{

sop("|--");

StringBuilder sb = new StringBuilder();

for(int i=0;i<level;i++)

{

sb.inset(0."| ")

}

return sb.toString();

}

(3)需求：删除带内容的目录：

public static void removeDir(File dir)

{

File[] files = file.listFile();

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

{

if(files[i].isDirectory&&!files[i].isHidden())

removeDir(files[i]);//如果是文件夹则继续调用函数

else//如果是文件则删除。注意删除的时候打印删除的结果，防止误删或者重删的情况

sop(files[i].toString()+"::"+files[i].delete());

}

sop(dir+"::"+dir.delete());

}

(4)需求：将制定目录下的java文件的绝对路径存储到文本文件中。

思路：

\*\*对指定目录进行递归

\*\*获取递归过程中所有java文件的路径

\*\*将这些路径存储到集合中

\*\*将集合中的数据写入文件中

//对指定目录进行递归并将所以Java文件存储到集合中

public static void getFileName(File file,ArrayList<File> arraylist){

File[] files = file.listFiles();

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

if(files[i].isDirectory()&&!files[i].isHidden()){

getFileName(files[i],arraylist);

}else{

if(files[i].getName().endsWith(".java")){

arraylist.add(files[i]);

}

}

}

}

//将集合中所有数据存储到新文件中

public static void saveFileToNewDir(ArrayList<File> arraylist,File newDir){

BufferedWriter bufw = null;

try {

bufw = new BufferedWriter(new FileWriter(newDir));

for (File file : arraylist) {

String fileAbsolutePath = file.getAbsolutePath();

bufw.write(fileAbsolutePath);

bufw.newLine();

bufw.flush();

}

} catch (Exception e) {

System.out.println("文件写入失败");

}finally{

try {

if(bufw!=null)

bufw.close();

} catch (Exception e2) {

System.out.println("文件写入流关闭失败");

}

}

}

9、Properties

(1)Properties是HashTable的子类，具备Map集合的特点，里面存储的是键值对

(2)Properties是IO流合集合相结合的集合容器

(3)Properties的特点是可以用于存储键值对形式的配置文件

(4)构造方法：

Properties()

创建一个无默认值的空属性列表。

Properties(Properties defaults)

创建一个带有指定默认值的空属性列表。

(5)方法摘要：

Object setProperty(String key, String value)

调用 Hashtable 的方法 put。

String getProperty(String key)

用指定的键在此属性列表中搜索属性。

void load(InputStream inStream)

从输入流中读取属性列表（键和元素对）。

void load(Reader reader)

按简单的面向行的格式从输入字符流中读取属性列表（键和元素对）。

void list(PrintStream out)

将属性列表输出到指定的输出流。

void list(PrintWriter out)

将属性列表输出到指定的输出流。

void store(OutputStream out, String comments)

以适合使用 load(InputStream) 方法加载到 Properties 表中的格式，

将此 Properties 表中的属性列表（键和元素对）写入输出流。

void store(Writer writer, String comments)

以适合使用 load(Reader) 方法的格式，将此 Properties 表中的

属性列表（键和元素对）写入输出字符。

Set<String> stringPropertyNames()

返回此属性列表中的键集，其中该键及其对应值是字符串，如果在主属性列表中

未找到同名的键，则还包括默认属性列表中不同的键

(6)Properties代码示例：

public static void show()

{

Properties prop = new Properties();

prop.setProperty("张三","26");

prop.setProperty("李四","30");

prop.setProperty("王五","35");

sop(prop);

String value = prop.getProperty("张三");

Set<String> keys = prop.stringPropertyName();

for(String key : values)

{

sop(key+":"+prop.getPropety(key));

}

}

(7)需求：记录应用程序的使用次数，如果使用次数已到，则提示用户注册。

思路：

\*\*第一次使用时建立一个配置文件用于记录使用次数

\*\*每次使用都加载该配置文件，并先判断已使用次数

\*\*每次使用完使用次数加1，写入配置文件

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

Properties prop = new Properties();//定义Properties，用来和IO流结合

File file = new File("library\\time.ini");//配置文件

if(!file.exists())

file.createNewFile();//如果文件不存在则创建文件(用于第一次使用时创建文件)

FileInputStream fis = new FileInputStream(file);//定义字节读取流，读取配置文件中记录的使用次数

prop.load(fis);//载入流，以获取文件中配置的键值对

int count = 0;//定义使用次数

String countValue = prop.getProperty("time");//通过键获取值

if(countValue!=null){//第一次时countValue为null

count = Integer.parseInt(countValue);//将字符串次数变成数字次数

if(count>3){

System.out.println("您使用次数已到，继续使用请注册！");

return;

}

}

count++;//如果使用次数未到则次数加1

prop.setProperty("time", count+"");//配置新的键值对

FileWriter fos = new FileWriter(file);

prop.store(fos, "这是应用程序使用次数的配置文件");//将新的键值对写入文件

fis.close();

fos.close();

}

10、IO中的其他流：

(1)打印流：

\*\*PrintWriter:字符打印流

\*\*\*\*构造方法：

PrintWriter(String fileName)

创建具有指定文件名称且不带自动行刷新的新 PrintWriter。

PrintWriter(File file)

使用指定文件创建不具有自动行刷新的新 PrintWriter。

PrintWriter(Writer out)

创建不带自动行刷新的新 PrintWriter。

PrintWriter(Writer out, boolean autoFlush)

自动刷新

PrintWriter(OutputStream out)

根据现有的 OutputStream 创建不带自动行刷新的新 PrintWriter。

PrintWriter(OutputStream out, boolean autoFlush)

自动刷新

\*\*\*\*方法摘要：

PrintWriter append(char c)

将指定字符添加到此 writer。

void close()

关闭该流并释放与之关联的所有系统资源。

void flush()

刷新该流的缓冲。

void print(Object obj)

打印对象。

void print(String s)

打印字符串。

void println()

通过写入行分隔符字符串终止当前行。

\*\*PrintStream:字节打印流

\*\*\*\*构造方法：

PrintStream(String fileName)

创建具有指定文件名称且不带自动行刷新的新打印流。

PrintStream(File file)

创建具有指定文件且不带自动行刷新的新打印流。

PrintStream(OutputStream out)

创建新的打印流。

PrintStream(OutputStream out, boolean autoFlush)

创建新的打印流。

\*\*\*\*方法摘要：

PrintWriter append(char c)

将指定字符添加到此 writer。

void close()

关闭该流并释放与之关联的所有系统资源。

void flush()

刷新该流的缓冲。

void print(Object obj)

打印对象。

void print(String s)

打印字符串。

void println()

通过写入行分隔符字符串终止当前行。

(2)对象系列化：

\*\*对象实体化：找一个介质，能长期的存储对象。

\*\*对象的属性在Java程序中，都是存在于对内存中，随着对象的消失而消失，

而ObjectOutputStream可以将对象实体化

\*\*Serializable接口没有一个方法，也就是说其是一个标记接口。比如盖章的猪肉才是安全的。

\*\*只有实现Serializable接口的子类才能被ObjectOutputStream系列化写入流，当某个

类实现该接口后，会被Java自动分配UID号，以便编译器识别，区分不同对象。

\*\*用ObjectOutputStream系列化的对象存储到文件后，该文件是乱码，也就是不可读的

的用ObjectInputStream读取该类对象的属性。

\*\*由于对象是有Java给对象分配相应的UID号，而UID号是根据对象的属性不同而分配的。

当一个类对象被系列化到文件后，如果该类改动了对象的属性，比如将某个成员变量变成私有

则该对象再用ObjectInputStream读取时会报异常，也就是说该系列化到文件的对象不能再被使用了

那么，要想继续使用属性被改动后的对象，我们可以自定义给对象分配UID号，让UID号不随对象的属性

变化而变化。

自定义对象分配UID方法如下：

public static final long serialVersion UID = 43L;

\*\*注意：

静态不能被系列化，因为静态成员变量实在内存的方法区，而ObjectOutputStream只能

对对内存里面的数据进行系列化

被transient修饰的非静态成员变量也不能被系列化

被系列化的对象存储到文件中，该文件是不可读的，所以该文件的扩展名一般

不写成.txt，通常后缀名写.object

\*\*ObjectOutputStream

\*\*ObjectInputStream

(3)管道流：

PipedInputStream

PipedOutputStream

(4)随机访问文件：RandomAccess(重要！！！)

\*\*自身具备读写方法(很牛逼！又可以读又可以写)

\*\*通过skipByte(int x)和seek(int x)来达到随机访问文件

\*\*该类不是IO体系子类，而是直接继承Object，但它是IO包中的成员，因为它具备读写方法

\*\*该类内部封装了数组，而且通过指针对数组的元素进行操作，可以通过getFilePoint获取指针位置

同时可以通过seek改变指针位置

\*\*该类完成读写的原理是内部封装了字节输入输出流

\*\*通过该类的构造看出，该类只能操作文件，而且操作的文件只能有固定模式：

"r":只读

"rw":读写

"rws":

"red":

\*\*构造方法：

RandomAccessFile(File file, String mode)

创建从中读取和向其中写入（可选）的随机访问文件流，该文件由 File 参数指定。

RandomAccessFile(String name, String mode)

创建从中读取和向其中写入（可选）的随机访问文件流，该文件具有指定名称。

\*\*方法摘要：

void write(byte[] b)

将 b.length 个字节从指定 byte 数组写入到此文件，并从当前文件指针开始。

void write(byte[] b, int off, int len)

将 len 个字节从指定 byte 数组写入到此文件，并从偏移量 off 处开始。

void write(int b)

向此文件写入指定的字节。

int read()

从此文件中读取一个数据字节。

int read(byte[] b)

将最多 b.length 个数据字节从此文件读入 byte 数组。

int read(byte[] b, int off, int len)

将最多 len 个数据字节从此文件读入 byte 数组。

String readLine()

从此文件读取文本的下一行。

long getFilePointer()

返回此文件中的当前偏移量。

long length()

返回此文件的长度。

void seek(long pos)

设置到此文件开头测量到的文件指针偏移量，在该位置发生下一个读取或写入操作。

(4)操作基本数据类型的流对象：DateStream

(5)操作字节数组流：

ByteArrayInputStream

ByteArrayOutputStream

11、IO流转换流的字符编码

(1)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

(2)通过子类转换流来完成

InputStreamReander

OutputStreamWriter

(3)在两个子类对象进行构造的时候可以加入编码表

(4)编码表：

将各个国家的文字用二进制数字表示并一一对应，形成一张表，这就是编码表

(5)常见的编码表：

\*\*ASCII：美国标准信息交换码，用一个字节的七位表示

\*\*ISO8859-1：拉丁码表，欧洲码表，用一个字节的八位表示

\*\*GB2312：中文编码表，用两个字节表示

\*\*GBK：中文编码表升级，融合录入更多的中文字符，用两个字节表示，为避免和老美重复

两字节的最高位都是1，即汉字都是用负数表示

\*\*Unicode：国际标准码，融合了多种文字，所有文字都用两个字节表示

\*\*UTF-8：用一个字节到三个字节表示。

注：Unicode能识别中文，UTF-8也能识别中文，但两种编码表示一个汉字所用的字节数不同

Unicode用两个字节，UTF-8用三个字节，故涉及到编码转换。

(6)在流中涉及编码表的转换只有转换流：

InputStreamReander

OutputStreamWriter

(7)代码示例：

public static void write() throws IOException

{

OutputStreamWriter osw1 = new OutputStreamWriter(new FileOutputStream("gbk.txt"),"GBK");

osw1.write("你好");

osw1.close();

OutputStreamWriter osw2 = new OutputStreamWriter(new FileOutputStream("utf-8.txt"),"UTF-8");

osw2.write("你好");

osw2.close();

}

public static void read() throws IOException

{

InputStreamReader isr = new InputStreamReader(new FileInputStream("gbk.txt"),"GBK");

byte[] buf = new byte[1024];

int len = isr.read(buf);

sop(new String(buf,0,len));

}

(8)编码解码

编码：字符串变成字节数组：String-->getBytes()-->byte[]()

解码：字节数组变成字符串：byte[]-->new String(byte[],0,len)-->String

(9)代码示例：

public static void main(String[] args)

{

//编码解码1：默认编码

String str1 = "你好";

byte[] buf1 = str1.getBytes();//默认解码：Unicode，四个字节

//编码解码2：指定编码

String str2 = "你好";

byte[] buf2 = str2.getBytes("UTF-8");//指定解码：UTF-8,六个字节

//编码解码3：编码正确解码错误

String str3 = "你好";

byte[] buf3 = str3.getBytes("GBK");//指定编码：GBK,四个字节

String str3 = new String(buf3,"ISO8859-1");//错误解码

//编码解码4：错误编码正确解码

String str4 = "你好";

byte[] buf4 = str4.getBytes("ISO8859-1");//错误编码

String str4 = new String(buf4,"GBK");//正确解码，读不出来

//编码解码5：编码对了，但是解码错误了，怎么办呢？

//此时可以将错误的解码再错编回去，载用正确编码解码

String str5 = "你好";

byte[] buf5 = str5.getBytes("GBK");//正确编码

String str6 = new String(buf5,"ISO8859-1");//错误解码，读不出来

byte[] buf6 = str6.getBytes("ISO8859-1");//再错误编码

String str7 = new String(buf6,"GBK");//再正确解码，这样就可以读出来了

}

六、网络编程：

1、网络编程概述

(1)网络模型

OSI参考模型

TCP/IP参考模型

(2)网络通讯要素

IP地址

端口号

传输协议

(3)网络通讯前提：

\*\*找到对方IP

\*\*数据要发送到指定端口。为了标示不同的应用程序，所以给这些网络应用程序都用数字进行标示

。这个表示就叫端口。

\*\*定义通信规则。这个规则称为通信协议，国际组织定义了通用协议TCP/IP

(4)计算机网络：

是指将地理位置不同的具有独立功能的多台计算机及其外部设备，

通过通信线路连接起来，在网络操作系统，网络管理软件及网络通信协议的管理和协调下，

实现资源共享和信息传递的计算机系统。

(5)IP地址：

IP地址 = 网络号码+主机地址

A类IP地址:第一段号码为网络号码，剩下的三段号码为本地计算机的号码

B类IP地址:前二段号码为网络号码，剩下的二段号码为本地计算机的号码

C类IP地址:前三段号码为网络号码，剩下的一段号码为本地计算机的号码

特殊地址:

127.0.0.1 回环地址,可用于测试本机的网络是否有问题. ping 127.0.0.1

ipconfig:查看本机IP地址

xxx.xxx.xxx.0 网络地址

xxx.xxx.xxx.255 广播地址

A类 1.0.0.1---127.255.255.254 10.X.X.X是私有地址(私有地址就是在互联网上不使用，而被用在局域网络中的地址) (2)127.X.X.X是保留地址，用做循环测试用的。

B类 128.0.0.1---191.255.255.254 172.16.0.0---172.31.255.255是私有地址。169.254.X.X是保留地址。

C类 192.0.0.1---223.255.255.254 192.168.X.X是私有地址

D类 224.0.0.1---239.255.255.254

E类 240.0.0.1---247.255.255.254

(6)各种网络分类方式

A:按网络覆盖范围划分

　　局域网(几米至10公里以内) 　　城域网(10~100公里) 　　广域网(几百公里到几千公里) 　　国际互联网

B:按网络拓扑结构划分

　　总线型网络 　　星形网络 　　环型网络 　　树状网络 　　混合型网络

C:按传输介质划分

　　有线网 　　无线网

D:按网络使用性质划分

　　公用网 　　专用网

(7)虚拟专用网络（Virtual Private Network ，简称VPN)指的是在公用网络上建立专用网络的技术。

其之所以称为虚拟网，主要是因为整个VPN网络的任意两个节点之间的连接并没有传统专网

所需的端到端的物理链路，而是架构在公用网络服务商所提供的网络平台，如Internet、

ATM(异步传输模式〉、Frame Relay （帧中继）等之上的逻辑网络，

用户数据在逻辑链路中传输。它涵盖了跨共享网络或公共网络的封装、

加密和身份验证链接的专用网络的扩展。VPN主要采用了隧道技术、加解密技术、

密钥管理技术和使用者与设备身份认证技术。

(8)网络模型：

\*\*\*\*OSI模型

应用层

表示层

会话层

传输层

网络层

数据连接层

物理层

\*\*\*\*TCP/IP模型

应用层

传输层

网际层

主机至网络层

2、TCP和UDP

(1)UDP和TCP的区别：

UDP

将数据及源和目的封装成数据包中，不需要建立连接

每个数据报的大小在限制在64k内

因无连接，是不可靠协议

不需要建立连接，速度快

TCP

建立连接，形成传输数据的通道。

在连接中进行大数据量传输

通过三次握手完成连接，是可靠协议

必须建立连接，效率会稍低

注：三次握手：

第一次：我问你在么？

第二次：你回答在。

第三次：我反馈哦我知道你在。

3、Socket(UDP传输)

\*\*Socket就是为网络服务提供的一种机制。

\*\*通信的两端都有Socket。

\*\*网络通信其实就是Socket间的通信。

\*\*数据在两个Socket间通过IO传输。

\*\*玩Socket主要就是记住流程，代码查文档就行

(1)UDP传输：DatagramSocket与DatagramPacket

\*\*发送端：

建立DatagramSocket服务；

提供数据，并将数据封装到字节数组中；

创建DatagramPacket数据包，并把数据封装到包中，同时指定IP和接收端口

通过Socket服务，利用send方法将数据包发送出去；

关闭DatagramSocket和DatagramPacket服务。

\*\*接收端：

建立DatagramSocket服务，并监听一个端口；

定义一个字节数组和一个数据包，同时将数组封装进数据包；

通过DatagramPacket的receive方法，将接收的数据存入定义好的数据包；

通过DatagramPacke关闭t的方法，获取发送数据包中的信息；

关闭DatagramSocket和DatagramPacket服务。

DatagramSocket与DatagramPacket方法摘要：

\*\*\*\*\*DatagramSocket

构造方法：

DatagramSocket()

构造数据报套接字并将其绑定到本地主机上任何可用的端口。

DatagramSocket(int port)

创建数据报套接字并将其绑定到本地主机上的指定端口。

DatagramSocket(int port, InetAddress laddr)

创建数据报套接字，将其绑定到指定的本地地址。

方法摘要:

void close()

关闭此数据报套接字。

InetAddress getInetAddress()

返回此套接字连接的地址。

InetAddress getLocalAddress()

获取套接字绑定的本地地址。

int getPort()

返回此套接字的端口。

void receive(DatagramPacket p)

从此套接字接收数据报包。

void send(DatagramPacket p)

从此套接字发送数据报包。

\*\*\*\*DatagramPacket

构造方法：

DatagramPacket(byte[] buf, int length)

构造 DatagramPacket，用来接收长度为 length 的数据包。

DatagramPacket(byte[] buf, int length, InetAddress address, int port)

构造数据报包，用来将长度为 length 的包发送到指定主机上的指定端口号。

InetAddress getAddress()

返回某台机器的 IP 地址，此数据报将要发往该机器或者是从该机器接收到的。

byte[] getData()

返回数据缓冲区。

int getLength()

返回将要发送或接收到的数据的长度。

int getPort()

返回某台远程主机的端口号，此数据报将要发往该主机或者是从该主机接收到的。

代码示例：

\*\*\*\*发送端：

class UDPSend

{

public static void main(String[] args) throws Exception

{

DatagramSocket ds = new DatagramSocket();

byte[] buf = "这是UDP发送端".getBytes();

DatagramPacket dp = new DatagramPacket(

buf,buf.length,InetAddress.getByName("192.168.1.253"),10000);

ds.send(dp);

ds.close();

}

}

\*\*\*\*接收端

class UDPRece

{

public static void main(String[] args) throws Exception

{

DatagramSocket ds = new DatagramSocket(10000);

byte[] buf = new byte[1024];

DatagramPacket dp = new DatagramPacket(buf,buf.length);

ds.receive(dp);//将发送端发送的数据包接收到接收端的数据包中

String ip = dp.getAddress().getHosyAddress();//获取发送端的ip

String data = new String(dp.getData(),0,dp.getLength());//获取数据

int port = dp.getPort();//获取发送端的端口号

sop(ip+":"+data+":"+port);

ds.close();

}

}

需求1：UDP键盘录入数据，并发送给接收端

发送端：

class UDPSend

{

public static void main(String[] args) throws Exception

{

DatagramSocket ds = new DatagramSocket();

BufferedReader bufr = new BufferedReader(new InputStreamReader(System.in));

String line = null;

while((line = bufr.readLine())!=null)

{

if("886".equals(line))

break;

byte[] buf = line.getBytes();

DatagramPacket dp = new DatagramPacket(

buf,buf.length,InetAddress.getByName("192.168.1.253"),10000);

ds.send(dp);

}

ds.close();

}

}

接收端：

class UDPRece

{

public static void main(String[] args) throws Exception

{

DatagramSocket ds = new DatagramSocket(10000);

while(true)

{

byte[] buf = new byte[1024];

DatagramPacket dp = new DatagramPacket(buf,buf.length);

ds.receive(dp);//将发送端发送的数据包接收到接收端的数据包中

String ip = dp.getAddress().getHosyAddress();//获取发送端的ip

String data = new String(dp.getData(),0,dp.getLength());//获取数据

int port = dp.getPort();//获取发送端的端口号

sop(ip+":"+data+":"+port);

ds.close();

}

}

}

需求2：编写简单的聊天工具

思路：

使用多线程技术

发送端：

class UDPSend implements Runnable

{

private DatagramSocket ds;

public UDPSend(){}

public UDPSend(DatagramSocket ds)

{

this.ds=ds;

}

public void run()

{

try

{

BufferedReader bufr = new BufferedReader(

new InputStreamReader(System.in));

String line = null;

while((line = bufr.readLine())!=null)

{

if("886".equals(line))

break;

byte[] buff = line.getBytes();

DatagramPacket dp = new DatagramPacket(

buf,buf.length,InetAddress.getByName("192.168.1.253"),10000);

ds.send(dp);

}

}

catch(Exception e)

{

throw new RuntimeException("发送失败");

}

}

}

接收端：

class UDPRece implements Runnable

{

private DatagramSocket ds;

public UDPSend(){}

public UDPSend(DatagramSocket ds)

{

this.ds=ds;

}

public void run()

{

try

{

while(true)

{

byte[] buf = new byte[1024];

DatagramPacket dp = new DatagramPacket(buf,buf.length);

ds.receive(dp);//将发送端发送的数据包接收到接收端的数据包中

String ip = dp.getAddress().getHosyAddress();//获取发送端的ip

String data = new String(dp.getData(),0,dp.getLength());//获取数据

int port = dp.getPort();//获取发送端的端口号

sop(ip+":"+data+":"+port);

}

}

catch(Exception e)

{

throw new RuntimeException("接收失败");

}

}

}

测试类：

class UDPTest

{

public static void main(String[] args)

{

DatagramSocket sendSocket = new DatagramSocket();

DatagramSocket receSocket = new DatagramSocket(10000);

new Thread(new UDPSend(sendSocket)).start();

new Thread(new UDPRece(receSocket)).start();

}

}

(2)TCP传输

Socket和ServerSocket

建立客户端和服务器端

建立连接后，通过Socket中的IO流进行数据的传输

关闭socket

同样，客户端与服务器端是两个独立的应用程序。

\*\*\*\*Socket

\*\*构造方法：

Socket()

通过系统默认类型的 SocketImpl 创建未连接套接字

Socket(InetAddress address, int port)

创建一个流套接字并将其连接到指定 IP 地址的指定端口号。

Socket(String host, int port)

创建一个流套接字并将其连接到指定主机上的指定端口号。

\*\*方法摘要：

void close()

关闭此套接字。

InetAddress getInetAddress()

返回套接字连接的地址。

InputStream getInputStream()

返回此套接字的输入流。

OutputStream getOutputStream()

返回此套接字的输出流。

int getPort()

返回此套接字连接到的远程端口。

void shutdownInput()

此套接字的输入流置于“流的末尾”。

void shutdownOutput()

禁用此套接字的输出流。

String toString()

将此套接字转换为 String。

\*\*\*\*ServerSocket

\*\*构造方法：

ServerSocket()

创建非绑定服务器套接字。

ServerSocket(int port)

创建绑定到特定端口的服务器套接字。

方法摘要：

Socket accept()

侦听并接受到此套接字的连接。

void close()

关闭此套接字。

InetAddress getInetAddress()

返回此服务器套接字的本地地址。

\*\*\*\*TCP传输流程：

\*\*客户端：

建立Socket服务，并制定要连接的主机和端口；

获取Socket流中的输出流OutputStream，将数据写入流中，通过网络发送给服务端；

获取Socket流中的输出流InputStream，获取服务端的反馈信息；

关闭资源。

\*\*服务端：

建立ServerSocket服务，并监听一个端口；

通过ServerSocket服务的accept方法，获取Socket服务对象；

使用客户端对象的读取流获取客户端发送过来的数据；

通过客户端对象的写入流反馈信息给客户端；

关闭资源；

\*\*\*\*代码示例：

客户端：

class TCPClient

{

public static void main(String[] args)

{

Socket s = new Socket("192.168.1.253",10000);

OutputStream os = s.getOutputStream();

out.write("这是TCP发送的数据".getBytes());

s.close();

}

}

服务端：

class TCPServer

{

public static void main(String[] args)

{

ServerSocket ss = new ServerSocket(10000);

Socket s = ss.accept();

String ip = s.getInetAddress().getHostAddress();

sop(ip);

InputStream is = s.getInputStream();

byte[] buf = new byte[1024];

int len = is.read(buf);

sop(new String(buf,0,len));

s.close();

ss.close();

}

}

TCP需求1：客户端给服务端发送数据，服务端接收到后反馈信息给客户端

客户端：

class TCPClient

{

public static void main(String[] args)

{

Socket s = new Socket("192.168.1.253",10000);

OutputStream os = s.getOutputStream();

out.write("这是TCP发送的数据".getBytes());

InputStream is = s.getInputStream();

byte[] buf = new byte[1024];

int len = is.read(buf);

sop(new String(buf,0,len));

s.close();

}

}

服务端：

class TCPServer

{

public static void main(String[] args)

{

ServerSocket ss = new ServerSocket(10000);

Socket s = ss.accept();

String ip = s.getInetAddress().getHostAddress();

sop(ip);

InputStream is = s.getInputStream();

byte[] buf = new byte[1024];

int len = is.read(buf);

sop(new String(buf,0,len));

OutputStream os = s.getOutputStream();

out.write("这是TCP发送的数据".getBytes());

s.close();

ss.close();

}

}

TCP需求2：建立一个文本转换服务端，客户给服务端发送文本，服务端将数据转换成大写后返回给客户端

当客户端输入over时，转换结束

客户端：

class TCPClient

{

public static void main(String[] args)

{

Socket s = new Socket("192.168.1.253",10000);

BufferedReader bufr = new BufferedReader(new InputStreamReader(System.in));

BufferedWriter bufOut = new BufferedWriter(new OutputStreamWriter(

s.getOutputStream()));

BufferedReader bufIn = new BufferedReader(new InputStreamReader(

s.getInputStream()));

String line = null;

while((line = bufr.readLine())!=null)

{

if("over".equals(line))

break;

bufOut.write(line);

bufOut.newLine();

bufOut.flush();

String retVal = bufIn.readLine();

sop("server:"+retVal);

}

bufr.close();

s.close();

}

}

服务端：

class TCPServer

{

public static void main(String[] args)

{

ServerSocket ss = new ServerSocket(10000);

Socket s = ss.accept();

String ip = s.getInetAddress().getHostAddress();

sop(ip);

BufferedReader bufIn = new BufferedReader(new InputStreamReader(

s.getInputStream()));

BufferedWriter bufOut = new BufferedWriter(new OutputStreamWriter(

s.getOutputStream()));

while((line = bufIn.readLine())!=null)

{

bufOut.write(line.toUpperCase());

bufOut.newLine();

bufOut.flush();

}

s.close();

ss.close();

}

}

\*\*需求3：拷贝文件

客户端：

class TCPClient

{

public static void main(String[] args)

{

Socket s = new Socket("192.168.1.253",10000);

BufferedReader bufr = new BufferedReader(new FileReader("g:\\demo.txt"));

PrintWriter pw = new PrintWriter(s.getOutputStream(),true);

String line = null;

while((line = bufr.readLine())!=null)

{

pw.println();

}

s.shutDownOutput();

BufferedReader bufIn = new BufferedReader(new InputStreamReader(

s.getInputStream()));

String retVal = bufIn.readLine();

sop(retVal);

bufr.close();

s.close();

}

}

服务端：

class TCPServer

{

public static void main(String[] args)

{

ServerSocket ss = new ServerSocket(10000);

Socket s = ss.accept();

String ip = s.getInetAddress().getHostAddress();

sop(ip);

BufferedReader bufIn = new BufferedReader(new InputStreamReader(

s.getInputStream()));

PrintWriter out = new PrintWriter(new FileWriter"copy.txt",true);

String line =null;

while((line = bufIn.readLine())!=null)

{

out.write(line);

}

PrintWriter pw = new PrintWriter(s.getOutputStream(),true);

pw.println("上传成功");

out.close();

s.close();

ss.close();

}

}

需求4：上传图片

客户端：

class TCPClient

{

public static void main(String[] args)

{

Socket s = new Socket("192.168.1.253",10000);

FileInputStream fis = new FileInputStream("g:\\1.bmp");

OutputStream out = s.getOutputStream();

byte[] buf = new byte[1024];

int len = 0;

while((len = bufr.read())!=-1)

{

out.write(buf,0,len);

}

s.shutDownOutput();

InputStream in = s.getInputStream();

byte[] bufIn = new byte[1024];

int lenIn = in.read(bufIn);

sop(new String(bufIn,0,lenIn);

fis.close();

s.close();

}

}

服务端：

class TCPServer

{

public static void main(String[] args)

{

ServerSocket ss = new ServerSocket(10000);

Socket s = ss.accept();

String ip = s.getInetAddress().getHostAddress();

sop(ip);

FileOutputStream fos = new FileOutputStream("g:\\copy.bmp");

InputStream in = s.getInputStream();

byte[] bufIn = new byte[1024];

int lenIn = 0;

while((lenIn=bufIn.read())!=-1)

{

fos.write(bufIn,0,lenIn)

}

OutputStream outIn = s.getOutputStream();

outIn.write("上传成功".getBytes());

fos.close();

s.close();

ss.close();

}

}

需求5：客户端并发登陆

客户端通过键盘录入用户名，服务端对这个用户名进行校验

如果用户存在，在服务端现实xxx已登录，并在客户端现实欢迎xxx

如果用户不存在，在服务端现实xxx正在尝试登陆，并在客户端现实xxx用户不存在

最多登陆三次。

校验端：

class User implements Runnable

(

private Socket s;

public User(){}

public User(Socket s)

{

this.s=s;

}

public void run()

{

try

{

BufferedReader bufrIn = new BufferedReader(

new InputStream(s.getInputStream()))

String name = bufrIn.readLine();

if(name==null)

{

sop("用户名为空");

break;

}

BufferedReader bufr = new BufferedReader(

new FileReader("user.txt"));

PrintWriter pw = new PrintWriter(s.getOutputStream(),true);

String line = null;

boolean flag = false;

while((line = bufr.reanLine())!=null)

{

if(line.equals(name))

{

flag = true;

break;

}

if(flag)

{

sop(name+"已登陆");

pw.println("欢迎"+name);

break;

}

else

{

sop(name+"正尝试登陆");

pw.println(name+"用户不存在");

}

}

s.close();

}

catch(Exception e)

{

throw new RuntimeException("用户校验失败");

}

}

)

客户端：

class LoginClient

{

public static void main(String[] args)

{

Socket s = new Socket("192.168.1.253",10000);

BufferedReader bufr = new BufferedReader(

new InputStreamReader(System.in)));

PrintWriter out = new PrintWriter(s.getOutputStream(),true);

BufferedReader bufIn = new BufferedReader(

new InputStreamReader(s.getInputStream()));

for(int i=0;i<3;i++)

{

String line = bufr.readLine();

if(line == null)

{

sop("用户名不能为空！");

break;

}

out.write(line);

String retVal = bufIn.readLine();

sop(retVal);

}

bufr.close();

s.close();

}

}

服务端：

class LoginServer

{

public static void main(String[] args)

{

ServerSocket ss = new ServerSocket(10000);

while(true)

{

Socket s = ss.accept();

new Thread(new User()).start();

}

}

}

## File类：

将文件系统中的文件和文件夹封装成了对象。提供了更多的属性和行为可以对

这些文件和文件夹进行操作。这些是流对象办不到的，因为流只操作数据。

### File类常见方法：

#### 1：创建。

boolean createNewFile()：在指定目录下创建文件，如果该文件已存在，则不创建。而对操作文件的输出流而言，输出流对象已建立，就会创建文件，如果文件已存在，会覆盖。除非续写。

**boolean mkdir()：创建此抽象路径名指定的目录。 boolean mkdirs()：**创建多级目录。

#### 2：删除。

boolean delete()：删除此抽象路径名表示的文件或目录。

void deleteOnExit()：在虚拟机退出时删除。

**注意：在删除文件夹时，必须保证这个文件夹中没有任何内容，才可以将该文件夹用delete删除。**

window的删除动作，是从里往外删。**注意**：java删除文件不走回收站。要慎用。

#### 3：获取.

long length()：获取文件大小。

String getName()：返回由此抽象路径名表示的文件或目录的名称。

 String getPath()：将此抽象路径名转换为一个路径名字符串。

String getAbsolutePath()：返回此抽象路径名的绝对路径名字符串。

String getParent()：返回此抽象路径名父目录的抽象路径名，如果此路径名没有指定父目录，则返回 null。

long lastModified()：返回此抽象路径名表示的文件最后一次被修改的时间。

 File.pathSeparator：返回当前系统默认的路径分隔符，windows默认为 “；”。

 File.Separator：返回当前系统默认的目录分隔符，windows默认为 “\”。

#### 4：判断：

boolean exists()：判断文件或者文件夹是否存在。

boolean isDirectory()：测试此抽象路径名表示的文件是否是一个目录。

 boolean isFile()：测试此抽象路径名表示的文件是否是一个标准文件。

 boolean isHidden()：测试此抽象路径名指定的文件是否是一个隐藏文件。

 boolean isAbsolute()：测试此抽象路径名是否为绝对路径名。

#### 5：重命名。

**boolean renameTo(File dest)：可以实现移动的效果。剪切+重命名。**  返回值是boolean

String[] list()：列出指定目录下的当前的文件和文件夹的名称。包含隐藏文件。

 如果调用list方法的File 对象中封装的是一个文件，那么list方法返回数组为null。如果封装的对象不存在也会返回null。只有封装的对象存在并且是文件夹时，这个方法才有效。

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

#### 递归：

就是函数自身调用自身。

什么时候用递归呢？

当一个功能被重复使用，而每一次使用该功能时的参数不确定，都由上次的功能元素结果来确定。

简单说：**功能内部又用到该功能，但是传递的参数值不确定。**(每次功能参与运算的未知内容不确定)。 递归的注意事项：

1：一定要定义递归的条件。

2：递归的次数不要过多。容易出现 StackOverflowError 栈内存溢出错误。

**其实递归就是在栈内存中不断的加载同一个函数。**

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

Java.util.Properties：一个可以将键值进行持久化存储的对象。Map--Hashtable

的子类。

 Map

|--Hashtable

|--Properties：用于属性配置文件，键和值都是字符串类型。

特点：1：可以持久化存储数据。2：键值都是字符串。3：一般用于配置文件。

 |-- load()：将流中的数据加载进集合。

原理：其实就是将读取流和指定文件相关联，并读取一行数据，因为数据是规则的key=value，所以获取一行后，通过 = 对该行数据进行切割，左边就是键，右边就是值，将键、值存储到properties集合中。

|-- store()：写入各个项后，刷新输出流。

|-- list()：将集合的键值数据列出到指定的目的地。

用于处理设备上数据。在流中一般以字节的形式存放着数据！

流：可以理解数据的流动，就是一个数据流。IO流最终要以对象来体现，对象都存在IO包中。

流也进行分类：

1：输入流（读）和输出流（写）。

2：因为处理的数据不同，分为字节流和字符流。

字节流：

处理字节数据的流对象。设备上的数据无论是图片或者dvd，文字，它们都以二进制存储的。二进制的最终都是以一个8位为数据单元进行体现，所以计算机中的最小数据单元就是字节。意味着，字节流可以处理设备上的所有数据，所以字节流一样可以处理字符数据。

那么为什么要有字符流呢？因为字符每个国家都不一样，所以涉及到了字符编码问题，那么GBK编码的中文用unicode编码解析是有问题的，所以需要获取中文字节数据的同时+指定的编码表才可以解析正确数据。为了方便于文字的解析，所以将字节流和编码表封装成对象，这个对象就是字符流。只要操作字符数据，优先考虑使用字符流体系。

注意：流的操作只有两种：读和写。

流的体系因为功能不同，但是有共性内容，不断抽取，形成继承体系。该体系一共有四个基类，而且都是抽象类。

字节流：InputStream OutputStream

字符流：Reader Writer

在这四个系统中，它们的子类，都有一个共性特点：子类名后缀都是父类名，前缀名都是这个子类的功能名称。

public static void main(String[] args) throws IOException { //读、写都会发生IO异常

/\*

1：创建一个字符输出流对象，用于操作文件。该对象一建立，就必须明确数据存储位置，是一个文件。

2：对象产生后，会在堆内存中有一个实体，同时也调用了系统底层资源，在指定的位置创建了一个存储数据的文件。

3：如果指定位置，出现了同名文件，文件会被覆盖。

\*/

FileWriter fw = new FileWriter("demo.txt"); // FileNotFoundException

/\*

调用Writer类中的write方法写入字符串。字符串并未直接写入到目的地中，而是写入到了流中，(其实是写入到内存缓冲区中)。怎么把数据弄到文件中？

\*/

fw.write("abcde");

fw.flush(); // 刷新缓冲区，将缓冲区中的数据刷到目的地文件中。

fw.close(); // 关闭流，其实关闭的就是java调用的系统底层资源。在关闭前，会先刷新该流。

}

## close()和flush()的区别：

flush()：将缓冲区的数据刷到目的地中后，流可以使用。

close()：将缓冲区的数据刷到目的地中后，流就关闭了，该方法主要用于结束调用的底层资源。这个动作一定做。

io异常的处理方式：io一定要写finally；

FileWriter写入数据的细节：

1：window中的换行符：\r\n两个符号组成。 linux：\n。

2：续写数据，只要在构造方法中传入新的参数true。

3：目录分割符：window \\ /

public static void main(String[] args) {

FileWriter fw = null;

try {

fw = new FileWriter("demo.txt",true);

fw.write("abcde");

}

catch (IOException e ){

System.out.println(e.toString()+"....");

}

finally{

if(fw!=null)

try{

fw.close();

}

catch (IOException e){

System.out.println("close:"+e.toString());

}

}

}

FileReader：使用Reader体系，读取一个文本文件中的数据。返回 -1 ，标志读到结尾。

import java.io.\*;

class FileReaderDemo {

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

/\*

创建可以读取文本文件的流对象，FileReader让创建好的流对象和指定的文件相关联。

\*/

FileReader fr = new FileReader("demo.txt");

int ch = 0;

while((ch = fr.read())!= -1) { //条件是没有读到结尾

System.out.println((char)ch); //调用读取流的read方法，读取一个字符。

read()方法一次读一个字节的二进制数据—是int型的！

}

fr.close();

}

}

读取数据的第二种方式：第二种方式较为高效，自定义缓冲区。

import java.io.\*;

class FileReaderDemo2 {

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

FileReader fr = new FileReader("demo.txt"); //创建读取流对象和指定文件关联。

//因为要使用read(char[])方法，将读取到字符存入数组。所以要创建一个字符数组，一般数组的长度都是1024的整数倍。

char[] buf = new char[1024];//读取的字符数组长度是1024

int len = 0;

while(( len=fr.read(buf)) != -1) {

System.out.println(new String(buf,0,len));//将char类型的数据从0到len转换成String

}

fr.close();

}

}

IO中的使用到了一个设计模式：装饰设计模式。

装饰设计模式解决：对一组类进行功能的增强。

包装：写一个类(包装类)对被包装对象进行包装；

\* 1、包装类和被包装对象要实现同样的接口；

\* 2、包装类要持有一个被包装对象；

\* 3、包装类在实现接口时，大部分方法是靠调用被包装对象来实现的，对于需要修改的方法我们自己实现；

## 字符流：

Reader：用于读取字符流的抽象类。子类必须实现的方法只有 read(char[], int, int) 和 close()。

|---BufferedReader：从字符输入流中读取文本，缓冲各个字符，从而实现字符、数组和行的高效读取。 可以指定缓冲区的大小，或者可使用默认的大小。大多数情况下，默认值就足够大了。

|---LineNumberReader：跟踪行号的缓冲字符输入流。此类定义了方法 setLineNumber(int) 和 getLineNumber()，它们可分别用于设置和获取当前行号。

|---InputStreamReader：是字节流通向字符流的桥梁：它使用指定的 charset 读取字节并将其解码为字符。它使用的字符集可以由名称指定或显式给定，或者可以接受平台默认的字符集。

|---FileReader：用来读取字符文件的便捷类。此类的构造方法假定默认字符编码和默认字节缓冲区大小都是适当的。要自己指定这些值，可以先在 FileInputStream 上构造一个 InputStreamReader。

|---CharArrayReader：

|---StringReader：

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

Writer：写入字符流的抽象类。子类必须实现的方法仅有 write(char[], int, int)、flush() 和 close()。

|---BufferedWriter：将文本写入字符输出流，缓冲各个字符，从而提供单个字符、数组和字符串的高效写入。

|---OutputStreamWriter：是字符流通向字节流的桥梁：可使用指定的 charset 将要写入流中的字符编码成字节。它使用的字符集可以由名称指定或显式给定，否则将接受平台默认的字符集。

|---FileWriter：用来写入字符文件的便捷类。此类的构造方法假定默认字符编码和默认字节缓冲区大小都是可接受的。要自己指定这些值，可以先在 FileOutputStream 上构造一个 OutputStreamWriter。

|---PrintWriter：

|---CharArrayWriter：

|---StringWriter：

## 字节流：

InputStream：是表示字节输入流的所有类的超类。

|--- FileInputStream：从文件系统中的某个文件中获得输入字节。哪些文件可用取决于主机环境。FileInputStream 用于读取诸如图像数据之类的原始字节流。要读取字符流，请考虑使用 FileReader。

|--- FilterInputStream：包含其他一些输入流，它将这些流用作其基本数据源，它可以直接传输数据或提供一些额外的功能。

|--- BufferedInputStream：该类实现缓冲的输入流。

|--- Stream：

|--- ObjectInputStream：

|--- PipedInputStream：

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

OutputStream：此抽象类是表示输出字节流的所有类的超类。

|--- FileOutputStream：文件输出流是用于将数据写入 File 或 FileDescriptor 的输出流。

|--- FilterOutputStream：此类是过滤输出流的所有类的超类。

|--- BufferedOutputStream：该类实现缓冲的输出流。

|--- PrintStream：

|--- DataOutputStream：

|--- ObjectOutputStream：

|--- PipedOutputStream：

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

## 缓冲区是提高效率用的，给谁提高呢？

BufferedWriter(将流和缓冲区结合)：是给字符输出流提高效率用的，那就意味着，缓冲区对象建立时，必须要先有流对象。明确要提高具体的流对象的效率。

FileWriter fw = new FileWriter("bufdemo.txt");

BufferedWriter bufw = new BufferedWriter(fw);//让缓冲区和指定流相关联。

for(int x=0; x<4; x++){

bufw.write(x+"abc");

bufw.newLine(); //写入一个换行符，这个换行符可以依据平台的不同写入不同的换行符。

bufw.flush();//对缓冲区进行刷新，可以让数据到目的地中。

}

bufw.close();//关闭缓冲区，其实就是在关闭具体的流。

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

BufferedReader：

FileReader fr = new FileReader("bufdemo.txt");

BufferedReader bufr = new BufferedReader(fr);

String line = null;

while((line=bufr.readLine())!=null){ //readLine方法返回的时候是不带换行符的。

System.out.println(line);

}

bufr.close();

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

//记住，只要一读取键盘录入，就用这句话。

BufferedReader bufr = new BufferedReader(new InputStreamReader(System.in));

//将读取到的从键盘输入的字节转化成字符存在流中，并将其放入缓冲区

BufferedWriter bufw = new BufferedWriter(new OutputStreamWriter(System.out));//输出到控制台

String line = null;

while((line=bufr.readLine())!=null){

if("over".equals(line))

break;

bufw.write(line.toUpperCase());//将输入的字符转成大写字符输出

bufw.newLine();//换行

bufw.flush();

}

bufw.close();

bufr.close();

## 流对象：

其实很简单，就是读取和写入。但是因为功能的不同，流的体系中提供N多的对象。那么开始时，到底该用哪个对象更为合适呢？这就需要明确流的操作规律。

## 流的操作规律：

1，明确源和目的。

数据源：就是需要读取，可以使用两个体系：InputStream、Reader；

数据汇：就是需要写入，可以使用两个体系：OutputStream、Writer；

2，操作的数据是否是纯文本数据？

如果是：数据源：Reader

数据汇：Writer

如果不是：数据源：InputStream

数据汇：OutputStream

3，虽然确定了一个体系，但是该体系中有太多的对象，到底用哪个呢？

明确操作的数据设备。

数据源对应的设备：硬盘(File)，内存(数组)，键盘(System.in)

数据汇对应的设备：硬盘(File)，内存(数组)，控制台(System.out)。

4，需要在基本操作上附加其他功能吗？比如缓冲。

如果需要就进行装饰。以提高效率！

转换流特有功能：转换流可以将字节转成字符，原因在于，将获取到的字节通过查编码表获取到指定对应字符。

转换流的最强功能就是基于 字节流 + 编码表 。没有转换，没有字符流。

发现转换流有一个子类就是操作文件的字符流对象：

InputStreamReader

|--FileReader

OutputStreamWriter

|--FileWrier

想要操作文本文件，必须要进行编码转换，而编码转换动作转换流都完成了。所以操作文件的流对象只要继承自转换流就可以读取一个字符了。

但是子类有一个局限性，就是子类中使用的编码是固定的，是本机默认的编码表，对于简体中文版的系统默认码表是GBK。

FileReader fr = new FileReader("a.txt");

InputStreamReader isr = new InputStreamReader(new FileInputStream("a.txt"),"gbk");

以上两句代码功能一致，

如果仅仅使用平台默认码表，就使用FileReader fr = new FileReader("a.txt"); //因为简化。

如果需要制定码表，必须用转换流。

转换流 = 字节流+编码表。

转换流的子类File = 字节流 + 默认编码表。

凡是操作设备上的文本数据，涉及编码转换，必须使用转换流。

File类：

将文件系统中的文件和文件夹封装成了对象。提供了更多的属性和行为可以对这些文件和文件夹进行操作。这些是流对象办不到的，因为流只操作数据。

## 递归：就是方法自身调用自身。

什么时候用递归呢？

当一个功能被重复使用，而每一次使用该功能时的参数不确定，都由上次的功能元素结果来确定。

简单说：功能内部又用到该功能，但是传递的参数值不确定。(每次功能参与运算的未知内容不确定)。

### 递归的注意事项：

1：一定要定义递归的条件。

2：递归的次数不要过多。容易出现 StackOverflowError 栈内存溢出错误。

其实递归就是在栈内存中不断的加载同一个方法。

==========================

Java.util.Properties：一个可以将键值进行持久化存储的对象。Map--Hashtable的子类。

Map

|--Hashtable

|--Properties：用于属性配置文件，键和值都是字符串类型。

特点：1：可以持久化存储数据。2：键值都是字符串。3：一般用于配置文件。

|-- load()：将流中的数据加载进集合。

原理：其实就是将读取流和指定文件相关联，并读取一行数据，因为数据是规则的key=value，所以获取一行后，通过 = 对该行数据进行切割，左边就是键，右边就是值，将键、值存储到properties集合中。

|-- store()：写入各个项后，刷新输出流。

|-- list()：将集合的键值数据列出到指定的目的地。

以下介绍IO包中扩展功能的流对象：基本都是装饰设计模式。

Java.io.outputstream.PrintStream：打印流

1：提供了更多的功能，比如打印方法。可以直接打印任意类型的数据。

2：它有一个自动刷新机制，创建该对象，指定参数，对于指定方法可以自动刷新。

3：它使用的本机默认的字符编码.

4：该流的print方法不抛出IOException。

该对象的构造方法。

PrintStream(File file) ：创建具有指定文件且不带自动行刷新的新打印流。

PrintStream(File file, String csn) ：创建具有指定文件名称和字符集且不带自动行刷新的新打印流。

PrintStream(OutputStream out) ：创建新的打印流。

PrintStream(OutputStream out, boolean autoFlush) ：创建新的打印流。

PrintStream(OutputStream out, boolean autoFlush, String encoding) ：创建新的打印流。

PrintStream(String fileName) ：创建具有指定文件名称且不带自动行刷新的新打印流。

PrintStream(String fileName, String csn)

PrintStream可以操作目的：1：File对象。2：字符串路径。3：字节输出流。

前两个都JDK1.5版本才出现。而且在操作文本文件时，可指定字符编码了。

当目的是一个字节输出流时，如果使用的println方法，可以在printStream对象上加入一个true参数。这样对于println方法可以进行自动的刷新，而不是等待缓冲区满了再刷新。最终print方法都将具体的数据转成字符串，而且都对IO异常进行了内部处理。

既然操作的数据都转成了字符串，那么使用PrintWriter更好一些。因为PrintWrite是字符流的子类，可以直接操作字符数据，同时也可以指定具体的编码。

====

PrintWriter：具备了PrintStream的特点同时，还有自身特点：

该对象的目的地有四个：1：File对象。2：字符串路径。3：字节输出流。4：字符输出流。

开发时尽量使用PrintWriter。

方法中直接操作文件的第二参数是编码表。

直接操作输出流的，第二参数是自动刷新。

//读取键盘录入将数据转成大写显示在控制台.

BufferedReader bufr = new BufferedReader(new InputStreamReader(System.in));//源：键盘输入

//目的：把数据写到文件中，还想自动刷新。

PrintWriter out = new PrintWriter(new FileWriter("out.txt"),true);//设置true后自动刷新

String line = null;

while((line=bufr.readLine())!=null){

if("over".equals(line))

break;

out.println(line.toUpperCase());//转大写输出

}

//注意：System.in，System.out这两个标准的输入输出流，在jvm启动时已经存在了。随时可以使用。当jvm结束了，这两个流就结束了。但是，当使用了显示的close方法关闭时，这两个流在提前结束了。

out.close();

bufr.close();

=======

SequenceInputStream：序列流，作用就是将多个读取流合并成一个读取流。实现数据合并。

表示其他输入流的逻辑串联。它从输入流的有序集合开始，并从第一个输入流开始读取，直到到达文件末尾，接着从第二个输入流读取，依次类推，直到到达包含的最后一个输入流的文件末尾为止。

这样做，可以更方便的操作多个读取流，其实这个序列流内部会有一个有序的集合容器，用于存储多个读取流对象。

该对象的构造方法参数是枚举，想要获取枚举，需要有Vector集合，但不高效。需用ArrayList，但ArrayList中没有枚举，只有自己去创建枚举对象。

但是方法怎么实现呢？因为枚举操作的是具体集合中的元素，所以无法具体实现，但是枚举和迭代器是功能一样的，所以，可以用迭代替代枚举。

合并原理：多个读取流对应一个输出流。三一

切割原理：一个读取流对应多个输出流。一三

import java.io.\*;

import java.util.\*;

class SplitFileDemo{

private static final String CFG = ".properties";

private static final String SP = ".part";

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

File file = new File("c:\\0.bmp");

File dir = new File("c:\\partfiles");

meger(dir);

}

//数据的合并。

public static void meger(File dir)throws IOException{

if(!(dir.exists() && dir.isDirectory()))

throw new RuntimeException("指定的目录不存在，或者不是正确的目录");

File[] files = dir.listFiles(new SuffixFilter(CFG));

if(files.length==0)

throw new RuntimeException("扩展名.proerpties的文件不存在");

//获取到配置文件

File config = files[0];

//获取配置文件的信息。

Properties prop = new Properties();

FileInputStream fis = new FileInputStream(config);

prop.load(fis);

String fileName = prop.getProperty("filename");

int partcount = Integer.parseInt(prop.getProperty("partcount"));

//--------------------------

File[] partFiles = dir.listFiles(new SuffixFilter(SP));

if(partFiles.length!=partcount)

throw new RuntimeException("缺少碎片文件");

//---------------------

ArrayList<FileInputStream> al = new ArrayList<FileInputStream>();

for(int x=0; x<partcount; x++){

al.add(new FileInputStream(new File(dir,x+SP)));

}

Enumeration<FileInputStream> en = Collections.enumeration(al);

SequenceInputStream sis = new SequenceInputStream(en);

File file = new File(dir,fileName);

FileOutputStream fos = new FileOutputStream(file);

byte[] buf = new byte[1024];

int len = 0;

while((len=sis.read(buf))!=-1){

fos.write(buf,0,len);

}

fos.close();

sis.close();

}

//带有配置信息的数据切割。

public static void splitFile(File file)throws IOException{

//用一个读取流和文件关联。

FileInputStream fis = new FileInputStream(file);

//创建目的地。因为有多个。所以先创建引用。

FileOutputStream fos = null;

//指定碎片的位置。

File dir = new File("c:\\partfiles");

if(!dir.exists())

dir.mkdir();

//碎片文件大小引用。

File f = null;

byte[] buf = new byte[1024\*1024];

//因为切割完的文件通常都有规律的。为了简单标记规律使用计数器。

int count = 0;

int len = 0;

while((len=fis.read(buf))!=-1){

f = new File(dir,(count++)+".part");

fos = new FileOutputStream(f);

fos.write(buf,0,len);

fos.close();

}

//碎片文件生成后，还需要定义配置文件记录生成的碎片文件个数。以及被切割文件的名称。

//定义简单的键值信息，可是用Properties。

String filename = file.getName();

Properties prop = new Properties();

prop.setProperty("filename",filename);

prop.setProperty("partcount",count+"");

File config = new File(dir,count+".properties");

fos = new FileOutputStream(config);

prop.store(fos,"");

fos.close();

fis.close();

}

}

class SuffixFilter implements FileFilter{

private String suffix;

SuffixFilter(String suffix){

this.suffix = suffix;

}

public boolean accept(File file){

return file.getName().endsWith(suffix);

}

}

## RandomAccessFile:

特点：

1：该对象即可读取，又可写入。

2：该对象中的定义了一个大型的byte数组，通过定义指针来操作这个数组。

3：可以通过该对象的getFilePointer()获取指针的位置，通过seek()方法设置指针的位置。

4：该对象操作的源和目的必须是文件。

5：其实该对象内部封装了字节读取流和字节写入流。

注意：实现随机访问，最好是数据有规律。

class RandomAccessFileDemo{

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

write();

read();

randomWrite();

}

//随机写入数据，可以实现已有数据的修改。

public static void randomWrite()throws IOException{

RandomAccessFile raf = new RandomAccessFile("random.txt","rw");

raf.seek(8\*4);

System.out.println("pos :"+raf.getFilePointer());

raf.write("王武".getBytes());

raf.writeInt(102);

raf.close();

}

public static void read()throws IOException{

RandomAccessFile raf = new RandomAccessFile("random.txt","r");//只读模式。

//指定指针的位置。

raf.seek(8\*1);//实现随机读取文件中的数据。注意：数据最好有规律。

System.out.println("pos1 :"+raf.getFilePointer());

byte[] buf = new byte[4];

raf.read(buf);

String name = new String(buf);

int age = raf.readInt();

System.out.println(name+"::"+age);

System.out.println("pos2 :"+raf.getFilePointer());

raf.close();

}

public static void write()throws IOException{

//rw：当这个文件不存在，会创建该文件。当文件已存在，不会创建。所以不会像输出流一样覆盖。

RandomAccessFile raf = new RandomAccessFile("random.txt","rw");//rw读写模式

//往文件中写入人的基本信息，姓名，年龄。

raf.write("张三".getBytes());

raf.writeInt(97);

raf.close();

}

}

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

### 管道流：

管道读取流和管道写入流可以像管道一样对接上，管道读取流就可以读取管道写入流写入的数据。

注意：需要加入多线程技术，因为单线程，先执行read，会发生死锁，因为read方法是阻塞式的，没有数据的read方法会让线程等待。

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

PipedInputStream pipin = new PipedInputStream();

PipedOutputStream pipout = new PipedOutputStream();

pipin.connect(pipout);

new Thread(new Input(pipin)).start();

new Thread(new Output(pipout)).start();

}

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

### 对象的序列化：

目的：将一个具体的对象进行持久化，写入到硬盘上。

注意：静态数据不能被序列化，因为静态数据不在堆内存中，是存储在静态方法区中。

如何将非静态的数据不进行序列化？用transient 关键字修饰此变量即可。

Serializable：用于启动对象的序列化功能，可以强制让指定类具备序列化功能，该接口中没有成员，这是一个标记接口。这个标记接口用于给序列化类提供UID。这个uid是依据类中的成员的数字签名进行运行获取的。如果不需要自动获取一个uid，可以在类中，手动指定一个名称为serialVersionUID id号。依据编译器的不同，或者对信息的高度敏感性。最好每一个序列化的类都进行手动显示的UID的指定。

import java.io.\*;

class ObjectStreamDemo {

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

writeObj();

readObj();

}

public static void readObj()throws Exception{

ObjectInputStream ois = new ObjectInputStream(new FileInputStream("obj.txt"));

Object obj = ois.readObject();//读取一个对象。

System.out.println(obj.toString());

}

public static void writeObj()throws IOException{

ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("obj.txt"));

oos.writeObject(new Person("lisi",25)); //写入一个对象。

oos.close();

}

}

class Person implements Serializable{

private static final long serialVersionUID = 42L;

private transient String name;//用transient修饰后name将不会进行序列化

public int age;

Person(String name,int age){

this.name = name;

this.age = age;

}

public String toString(){

return name+"::"+age;

}

}

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

DataOutputStream、DataInputStream：专门用于操作基本数据类型数据的对象。

DataOutputStream dos = new DataOutputStream(new FileOutputStream("data.txt"));

dos.writeInt(256);

dos.close();

DataInputStream dis = new DataInputStream(new FileInputStream("data.txt"));

int num = dis.readInt();

System.out.println(num);

dis.close();

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

ByteArrayInputStream：源：内存

ByteArrayOutputStream：目的：内存。

这两个流对象不涉及底层资源调用，操作的都是内存中数组，所以不需要关闭。

直接操作字节数组就可以了，为什么还要把数组封装到流对象中呢？因为数组本身没有方法，只有一个length属性。为了便于数组的操作，将数组进行封装，对外提供方法操作数组中的元素。

对于数组元素操作无非两种操作：设置（写）和获取（读），而这两操作正好对应流的读写操作。这两个对象就是使用了流的读写思想来操作数组。

//创建源：

ByteArrayInputStream bis = new ByteArrayInputStream("abcdef".getBytes());

//创建目的：

ByteArrayOutputStream bos = new ByteArrayOutputStream();

int ch = 0;

while((ch=bis.read())!=-1){

bos.write(ch);

}

System.out.println(bos.toString());

# 正则

**正则表达式：★★★☆，**其实是用来操作字符串的一些规则。

**好处：**正则的出现，对字符串的复杂操作变得更为简单。

**特点：**将对字符串操作的代码用一些符号来表示。只要使用了指定符号，就可以调用底层的代码对字符串进行操作。符号的出现，简化了代码的书写。

**弊端：**符号的出现虽然简化了书写，但是却降低了阅读性。 其实更多是用正则解决字符串操作的问题。

**组：**用小括号标示，每定义一个小括号，就是一个组，而且有自动编号，从1开始。 只要使用组，对应的数字就是使用该组的内容。别忘了，数组要加\\。 (aaa(wwww(ccc))(eee))技巧，从左括号开始数即可。有几个左括号就是几组。

**常见操作：**

**1，匹配：**其实用的就是String类中的matches方法。

 String reg = "[1-9][0-9]{4,14}";

boolean b = qq.matches(reg);//将正则和字符串关联对字符串进行匹配。

**2，切割：**

其实用的就是String类中的split方法。

**3，替换：**

其实用的就是String类中的replaceAll();

**4，获取：**

1）先要将正则表达式编译成正则对象。使用的是Pattern中静态方法 compile(regex);

2）通过Pattern对象获取Matcher对象。 Pattern用于描述正则表达式，可以对正则表达式进行解析。 而将规则操作字符串，需要从新封装到匹配器对象Matcher中。 然后使用Matcher对象的方法来操作字符串。 如何获取匹配器对象呢？

通过Pattern对象中的matcher方法。该方法可以正则规则和字符串想关联。并返回匹配器对象。

 3）使用Matcher对象中的方法即可对字符串进行各种正则操作。

一步一步满足条件，从左往右

格式：字符串.matches(表达式)

具体的参考API

正则表达式中\和.都要进行转义，才能进行表达

.代表所有字符

\\.将正则表达式中.的特性去掉

转义规则是前面加一个\即可

符号：

\* 零次或者多次

. 任何字符

\W 非单词字符

+ 一次或者多次

\d 数字0~9

^ 取反

？ 一次或者一次也没有

\D 非数字

x{n} 恰好n次

\s 空白字符

x{n,} 至少n次

\S 非空白字符

x{n,m} 至少n次，但不超过m次

\w 单词字符（a-zA-Z\_0-9）包括下划线

是用来操作字符串的一些规则。

好处：

正则的出现，对字符串的复杂操作变得更为简单。

特点：

将对字符串操作的代码用一些符号来表示。只要使用了指定符号，就可以调用底层的代码对字符串进行操作。符号的出现，简化了代码的书写。

弊端：符号的出现虽然简化了书写，但是却降低了阅读性。

其实更多是用正则解决字符串操作的问题。

组：用小括号标示，每定义一个小括号，就是一个组，而且有自动编号，从1开始。

只要使用组，对应的数字就是使用该组的内容。别忘了，数组要加\\。

(aaa(wwww(ccc))(eee))技巧，从左括号开始数即可。有几个左括号就是几组。

常见操作：

1，匹配：其实用的就是String类中的matches方法。

String reg = "[1-9][0-9]{4,14}";

boolean b = qq.matches(reg);//将正则和字符串关联对字符串进行匹配。

2，切割：其实用的就是String类中的split方法。

3，替换：其实用的就是String类中的replaceAll();

4，获取：

1），先要将正则表达式编译成正则对象。使用的是Pattern中静态方法 compile(regex);

2），通过Pattern对象获取Matcher对象。

Pattern用于描述正则表达式，可以对正则表达式进行解析。

而将规则操作字符串，需要从新封装到匹配器对象Matcher中。

然后使用Matcher对象的方法来操作字符串。

如何获取匹配器对象呢？

通过Pattern对象中的matcher方法。该方法可以正则规则和字符串想关联。并返回匹配器对象。

1. 使用Matcher对象中的方法即可对字符串进行各种正则操作。

## 案例

### 从字符串 str 中提取数字部分的内容(匹配一次)：

var str = "abc123def";

var patt1 = /[0-9]+/;

document.write(str.match(patt1));

以下标记的文本是获得的匹配的表达式：

123

# 多线程

## 线程与进程

线程是在进程的基础上进行的，main是主线程

随机休眠线程,连续加锁 Thread.sleep(new Random().nextInt());

### 单任务与多任务

单任务  例如：命令提示符（只能等待前一行代码运行完才可以继续运行）

多任务：同一时刻做多个任务，依靠CPU快速切换呈现一种假象

计算机解决办法是：频率，在一秒中切换的速度

CPU在同一时间只能做一件事

## 为什么有线程

程序员控制不了进程（CPU），但是可以控制线程

进程和进程之间是不能数据共享的，但线程之间可以

线程

## 线程常用方法：

run()     start()     getPriority()返回线程优先级

start()和run()之间的联系

start()用来启动一个线程，当调用start()方法后，系统才会开启一个新的线程，进而调用run()方法执行，而当单独调用run()方法就和调普通方法一样，失去线程特性

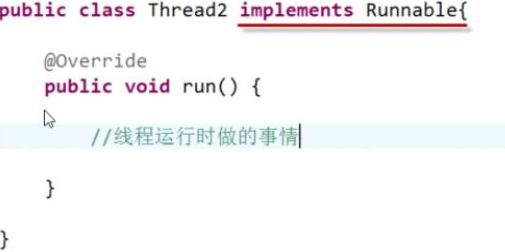
## java中创建线程的方式

主线程和子线程之间互不影响，与在main方法中代码的顺序无关   继承Thread类后，无法再继承其他类，但是实现Runnable接口，还可以继承其他类。

继承java.lang.thread类，并重写其中的run方法（线程运行要做的事情），调用start()方法启动线程



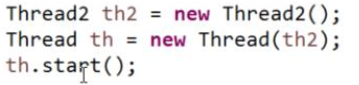
实现java.lang.Runnable接口。并实现接口中定义的run方法（线程运行时做的事），这种方式定义的子线程需要借助Thread类的start()方法来启动



当线程继承自java.lang.Thread类的时候，那么可以调用父类的start()方法启动线程，启动成功后，jvm会调用线程类中的run方法

D:\1 - 副本\Evernote_files\Image [9].png

当线程实现Runnable接口的时候，那么必须通过Thread类开启动线程



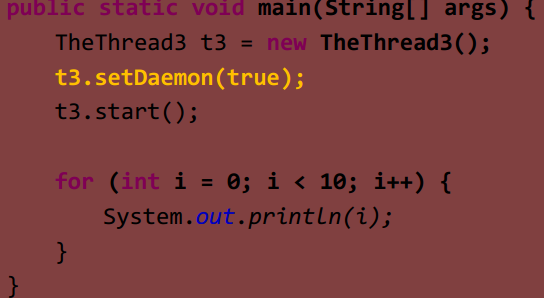
线程一旦死亡，就不能再次使用（再次使用指的是调用Start方法）

调用thread类的静态方法sleep(long)可以让线程进入休眠，休眠后的线程将会进入阻塞状态

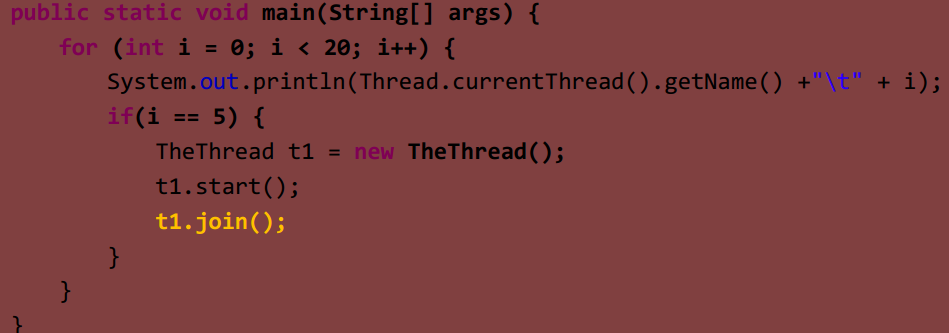
## 线程分类：

守护线程（主仆关系）

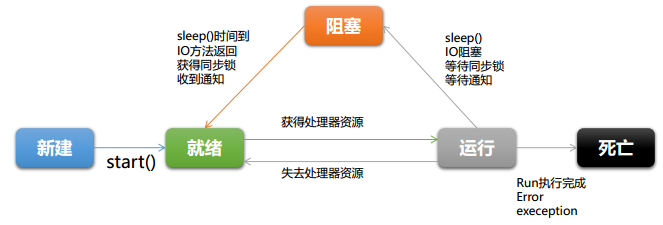
解释：随着主线程死亡而死亡的子线程。在子线程启动之前，调用setDeamon(true)方法



join线程：当某个程序在执行过程中调用了其他线程的join()方法，则当前线程被阻塞，知道join()方法假如的线程完成为止

解释：join 线程可以让其他线程进入阻塞状态，直到自己执行完。在线程启动后调用join()方法即可                            

## 线程生命周期



新建就是new  Thread      使用就是调用start()方法

新建—>就绪—>运行—>死亡

新建—>就绪—>运行—>（阻塞—>就绪—>运行）—>死亡

## 线程同步:

当多个线程同时对同一个对象的实例变量进行操作时，会引起线程的同步问题

产生线程不安全的条件：（必须3个条件都满足）

多个线程

同一个对象

实例变量（属性）

叙述：多线程的情况下，多个线程访问同一个对象的实例变量会引起线程安全问题。第一种在方法的声明处添加 synchronized 关键字；第二种使用 synchronized 同步代码块；第三种使用 ReentrantLock 同步锁

一些小窍门：

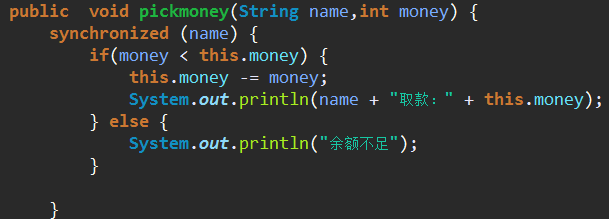
线程中调用实体类中的方法要善于利用构造方法传参（传实体类型的变量）

## 解决线程不安全：

### 线程加锁（使用synchronized关键字对方法进行加锁）

D:\1 - 副本\Evernote_files\Image [5] [1].png

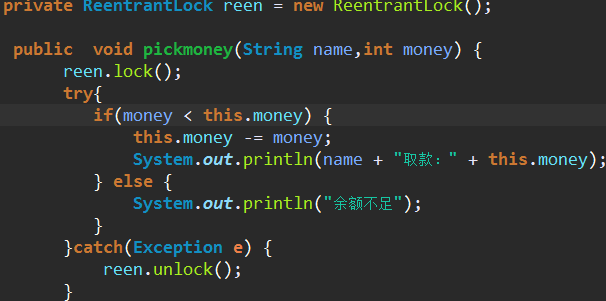
### 同步代码块（使用synchronized关键字对某段代码块进行加锁）



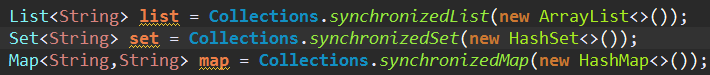
### 使用同步锁

私有化创建ReentrantLock类对象

调用Lock()方法和unLock()方法，并将代码块使用try catch包住



### 使用collections工具类的synchronizedXxx()方法获得线程安全的集合



## Collection和Collections的区别

Collection 是 List、 Set 的父接口。 Collections 是 java 提供的工具类，包含了操作集合的一系列工具方法，例如提供了实现线程安全的集合的方法和二分法搜索集合的方法等

## 同步与异步

同步：做完一件事再做一件事（一心一意）

异步：一边做一件事，一边做另一件事（三心二意）    多线程使之可能性出现

## 线程的２种创建方式（必问）

### 创建线程的第一种方式：

继承Thread ，由子类复写run方法。

步骤：

1，定义类继承Thread类；

2，目的是复写run方法，将要让线程运行的代码都存储到run方法中；

3，通过创建Thread类的子类对象，创建线程对象；

4，调用线程的start方法，开启线程，并执行run方法。

线程状态：

**新建：**start()

**运行：**具备执行资格，同时具备执行权；

**冻结：**sleep(time),wait()—notify()唤醒；线程释放了执行权，同时释放执行资格；

**临时阻塞状态：**线程具备cpu的执行资格，没有cpu的执行权；

**消亡：**stop()

### 创建线程的第二种方式：

**实现一个接口Runnable。**

 步骤：

1，定义类实现Runnable接口。

2，覆盖接口中的run方法（用于封装线程要运行的代码）。

3，通过Thread类创建线程对象；

4，将实现了Runnable接口的子类对象作为实际参数传递给Thread类中的构造函数。 为什么要传递呢？因为要让线程对象明确要运行的run方法所属的对象。

5，调用Thread对象的start方法。开启线程，并运行Runnable接口子类中的run方法。

**Ticket t = new Ticket();**

/\*

直接创建Ticket对象，并不是创建线程对象。

因为创建对象只能通过new Thread类，或者new Thread类的子类才可以。

所以最终想要创建线程。既然没有了Thread类的子类，就只能用Thread类。 \*/

Thread t1 = new Thread(t); //创建线程。

 /\* 只要将t作为Thread类的构造函数的实际参数传入即可完成线程对象和t之间的关联

为什么要将t传给Thread类的构造函数呢？其实就是为了明确线程要运行的代码run方法。

 \*/

## 多线程安全问题的原因：

通过图解：发现一个线程在执行多条语句时，并运算同一个数据时，在执行过程中，其他线程参与进来，并操作了这个数据。导致到了错误数据的产生。

 涉及到两个因素：

1/多个线程在操作共享数据。

2/有多条语句对共享数据进行运算。

原因：这多条语句，在某一个时刻被一个线程执行时，还没有执行完，就被其他线程执行了。

解决安全问题的原理：

只要将操作共享数据的语句在某一时段让一个线程执行完，在执行过程中，其他线程不能进来执行就可以解决这个问题。

如何进行多句操作共享数据代码的封装呢？

**java中提供了一个解决方式：**就是同步代码块。

格式：

**synchronized**(对象) {  **// 任意对象都可以。这个对象就是锁。**

需要被同步的代码；

 }

**Synchronized**

wait和sleep区别：

分析这两个方法：从执行权和锁上来分析：

wait：可以指定时间也可以不指定时间。不指定时间，只能由对应的notify或者notifyAll来唤醒。

sleep：必须指定时间，时间到自动从冻结状态转成运行状态(临时阻塞状态)。

**wait：线程会释放执行权，而且线程会释放锁。**

**Sleep：线程会释放执行权，但不是不释放锁。**

## 什么时候产生死锁,该怎么办?

## 线程的概述：

**进程**：

正在进行中的程序。其实进程就是一个应用程序运行时的内存分配空间。

**线程**：

其实就是进程中一个程序执行控制单元，一条执行路径。进程负责的是应用程序的空间的标示。线程负责的是应用程序的执行顺序。

1、进程和线程：

进程：正在进行的程序。每一个进程执行都有一个执行顺序，该顺序是一个执行路径，或者叫一个控制单元。

线程：进程内部的一条执行路径或者一个控制单元。

两者的区别：

一个进程至少有一个线程

进程在执行过程中拥有独立的内存单元，而多个线程共享内存；

一个进程至少有一个线程在运行，当一个进程中出现多个线程时，就称这个应用程序是多线程应用程序，每个线程在栈区中都有自己的执行空间，自己的方法区、自己的变量。

jvm在启动的时，首先有一个**主线程**，负责程序的执行，调用的是main方法。主线程执行的代码都在main方法中。

当产生垃圾时，收垃圾的动作，是不需要主线程来完成，因为这样，会出现主线程中的代码执行会停止，会去运行垃圾回收器代码，效率较低，所以由单独一个线程来负责垃圾回收。

jvm多线程的启动是多线程吗？

java的虚拟机jvm启动的是单线程，就有发生内存泄露的可能，而我们使用java程序没出现这样的问题，

也就是jvm启动至少有两个线程，一个执行java程序，一个执行垃圾回收。所以是多线程。

### 多线程的好处：

解决了一个进程里面可以同时运行多个任务。（执行路径）

提供资源的利用率，而不是提供效率。

### 线程的弊端：

1. 降低了一个进程里面的线程的执行频率。
2. 对线程进行管理要求额外的CPU开销，线程的使用会给系统带来上下文切换的额外负担。
3. 公有变量的同时读或写，当多个线程需要对共有变量进行写操作时，后一个线程往往会修改掉前一个线程存放的数据，发生线程安全问题。
4. 线程的死锁。即较长时间的等待或资源竞争以及死锁等多线程症状。

### 随机性的原理：

因为cpu的快速切换造成，哪个线程获取到了cpu的执行权，哪个线程就执行。

返回当前线程的名称：**Thread.currentThread().getName()**

线程的名称是由：Thread-编号定义的。编号从0开始。

线程要运行的代码都统一存放在了**run方法**中。

线程要运行必须要通过类中指定的方法开启。**start方法**。（启动后，就多了一条执行路径）

**start方法：**1）、启动了线程；2）、让jvm调用了run方法。

## 创建线程的方式

### 继承Thread创建线程

**创建线程的第一种方式：继承Thread ，由子类复写run方法。**

步骤：

1，定义类继承Thread类；

2，目的是复写run方法，将要让线程运行的代码都存储到run方法中；

3，通过创建Thread类的子类对象，创建线程对象；

4，调用线程的start方法，开启线程，并执行run方法。

1. 代码

Class Demo1 extends Thread{

Public Demo1(String name){

Super(name);

}

Public void

}

**线程状态：**

**新建：new一个Thread对象或者其子类对象就是创建一个线程，当一个线程对象被创建，但是没有开启，这个时候，**

**只是对象线程对象开辟了内存空间和初始化数据。**

**就绪：新建的对象调用start方法，就开启了线程，线程就到了就绪状态。**

**在这个状态的线程对象，具有执行资格，没有执行权。**

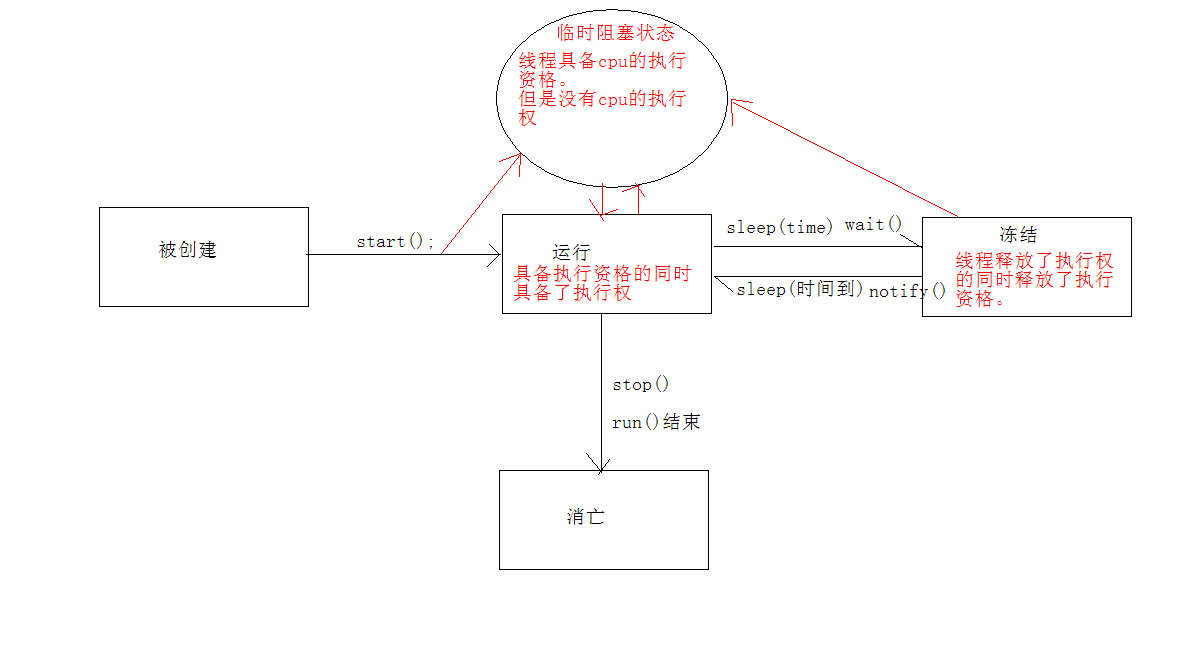
**运行：当线程对象获取到了CPU的资源。**

**在这个状态的线程对象，既有执行资格，也有执行权。**

**冻结：运行过程中的线程由于某些原因(比如wait,sleep)，释放了执行资格和执行权。**

**当然，他们可以回到运行状态。只不过，不是直接回到。**

**而是先回到就绪状态。**

**死亡：当线程对象调用的run方法结束，或者直接调用stop方法，就让线程对象死亡，在内存中变成了垃圾。**

### 实现Runnable创建线程

**创建线程的第二种方式：实现一个接口Runnable。**

步骤：

1，定义类实现Runnable接口。

2，覆盖接口中的run方法（用于封装线程要运行的代码）。

3，通过Thread类创建线程对象；

4，**将实现了Runnable接口的子类对象作为实际参数传递给Thread类中的构造方法。**

为什么要传递呢？因为要让线程对象明确要运行的run方法所属的对象。

5，调用Thread对象的start方法。开启线程，并运行Runnable接口子类中的run方法。

**Ticket t = new Ticket();**

/\*

直接创建Ticket对象，并不是创建线程对象。

因为创建对象只能通过new Thread类，或者new Thread类的子类才可以。

所以最终想要创建线程。既然没有了Thread类的子类，就只能用Thread类。

\*/

**Thread t1 = new Thread(t);** //创建线程。

/\*

只要将t作为Thread类的构造方法的实际参数传入即可完成线程对象和t之间的关联

为什么要将t传给Thread类的构造方法呢？其实就是为了明确线程要运行的代码run方法。

\*/

**t1.start();**

## 为什么要有Runnable接口的出现？

**1：**通过继承Thread类的方式，可以完成多线程的建立。但是这种方式有一个局限性，如果一个类已经有了自己的父类，就不可以继承Thread类，因为**java单继承**的局限性。

可是该类中的还有部分代码需要被多个线程同时执行。这时怎么办呢？

只有对该类进行额外的功能扩展，java就提供了一个接口Runnable。这个接口中定义了run方法，其实run方法的定义就是为了存储多线程要运行的代码。

所以，通常创建线程都用第二种方式。

**因为实现Runnable接口可以避免单继承的局限性。**

**2：**其实是将不同类中需要被多线程执行的代码进行抽取。将多线程要运行的代码的位置单独定义到接口中。为其他类进行功能扩展提供了前提。

所以Thread类在描述线程时，内部定义的run方法，也来自于Runnable接口。

**实现Runnable接口可以避免单继承的局限性。**而且，继承Thread，是可以对Thread类中的方法，进行子类复写的。但是不需要做这个复写动作的话，只为定义线程代码存放位置，实现Runnable接口更方便一些。所以**Runnable接口将线程要执行的任务封装成了对象**。

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

**//面试**

new Thread(new Runnable(){ //匿名

public void run(){

System.out.println("runnable run");

}

})

{

public void run(){

System.out.println("subthread run");

}

}.start(); //**结果：subthread run**

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

**Try {**

**Thread.sleep(10);**

**}catch(InterruptedException e){}**// 当刻意让线程稍微停一下，模拟cpu 切换情况。

**多线程安全问题的原因**：

通过图解：发现一个线程在执行多条语句时，并运算同一个数据时，在执行过程中，其他线程参与进来，并操作了这个数据。导致到了错误数据的产生。

**涉及到两个因素：**

**1、多个线程在操作共享数据。**

**2、有多条语句对共享数据进行运算。**

原因：这多条语句，在某一个时刻被一个线程执行时，还没有执行完，就被其他线程执行了。

**解决安全问题的原理**：

只要将操作共享数据的语句在某一时段让一个线程执行完，在执行过程中，其他线程不能进来执行就可以解决这个问题。

如何进行多句操作共享数据代码的封装呢？

**java中提供了一个解决方式：就是同步代码块。**

格式：

**synchronized(对象) { // 任意对象都可以。这个对象就是锁。**

**需要被同步的代码；**

**}**

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

实现方式和继承方式有什么区别？★（面试经常考）

实现方式好处：避免了单继承的局限性。

在定义线程时，建议使用实现方式。

两种方式的区别：

继承Thread：线程代码存放在Thread子类run方法中

实现Runnable，线程代码存放在接口的子类fun方法中。

## 创建线程是为什么要复写run方法？

Thread类用于描述线程。Thread类定义了一个功能，用于存储线程要运行的代码，该存储功能就是run方法。

## start()和run方法有什么区别？

调用start方法方可启动线程，而run方法只是thread的一个普通方法，调用run方法不能实现多线程；

Start()方法:

start方法用来启动线程,实现了多线程运行,这时无需等待run方法体代码执行完毕而直接继续执行下面的

代码。通过调用Thread类的start()方法来启动一个线程,这时此线程处于就绪(可运行)状态，并没有运行，

一旦得到cpu时间片(执行权),就开始执行run()方法,这里方法run()称为线程体，

它包含了要执行的这个线程的内容，Run方法运行结束,此线程随即终止。

Run()方法:

run()方法只是Thread类的一个普通方法,如果直接调用Run方法,程序中依然只有主线程这一个线程,

其程序执行路径还是只有一条，还是要等待run方法体执行完毕后才可继续执行下面的代码，

这样就没有达到多线程的目的。

## 同步：★★★★★

//就是在操作共享数据代码时，访问时只能让一个线程进去访问，此线程执行完退出后，别的线程才能再对此共享数据代码进行访问。

**好处：**解决了线程安全问题。Synchronized

**弊端**：相对降低性能，因为判断锁需要消耗资源，产生了死锁。

**定义同步是有前提的**：

1，必须要有两个或者两个以上的线程，才需要同步。

2，多个线程必须保证使用的是同一个锁。

3,必须保证同步中只有一个线程在运行

同步代码块:

可以指定需要获取哪个对象的同步锁,使用synchronized的代码块同样需要锁,但他的锁可以是任意对象

考虑到安全问题，一般还是使用同一个对象，相对来说效率较高。

注意：

\*\*虽然同步代码快的锁可以使任何对象，但是在进行多线程通信使用同步代码快时，

必须保证同步代码快的锁的对象和，否则会报错。

\*\*同步函数的锁是this，也要保证同步函数的锁的对象和调用wait、notify和notifyAll的对象是

同一个对象，也就是都是this锁代表的对象。

格式：

synchronized(对象)

{

需同步的代码;

}

同步的第二种表现形式：**//对共享资源的方法定义同步**

**同步方法**：其实就是将同步关键字定义在方法上，让方法具备了同步性。

同步函数

同步方法是指进入该方法时需要获取this对象的同步锁，在方法上使用synchronized关键字，

使用this对象作为锁，也就是使用了当前对象，因为锁住了方法，所以相对于代码块来说效率相对较低。

注:静态同步函数的锁是该方法所在的类的字节码文件对象，即类名.class文件

格式：

修饰词 synchronized 返回值类型 函数名(参数列表)

{

需同步的代码;

}

在jdk1.5后，用lock锁取代了synchronized，个人理解也就是对同步代码块做了修改，

并没有提供对同步方法的修改，主要还是效率问题吧。

**同步方法是用的哪个锁呢？//synchronized(this)用以定义需要进行同步的某一部分代码块**

通过验证，方法都有自己所属的对象this，所以同步方法所使用的锁就是this锁。**This.方法名**

**当同步方法被static修饰时，这时的同步用的是哪个锁呢？**

静态方法在加载时所属于类，这时有可能还没有该类产生的对象，但是该类的字节码文件加载进内存就已经被封装成了对象，这个对象就是**该类的字节码文件对象**。

所以静态加载时，只有一个对象存在，那么静态同步方法就使用的这个对象。

这个对象就是 **类名.class**

## 同步代码块和同步方法的区别？

同步代码块使用的锁可以是任意对象。

**同步方法使用的锁是this，静态同步方法的锁是该类的字节码文件对象**。

**在一个类中只有一个同步的话，可以使用同步方法。如果有多同步，必须使用同步代码块，来确定不同的锁。所以同步代码块相对灵活一些。**

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

**★考点问题：请写一个延迟加载的单例模式？写懒汉式；当出现多线程访问时怎么解决？加同步，解决安全问题；效率高吗？不高；怎样解决？通过双重判断的形式解决。**

//懒汉式：延迟加载方式。

当多线程访问懒汉式时，因为懒汉式的方法内对共性数据进行多条语句的操作。所以容易出现线程安全问题。为了解决，加入同步机制，解决安全问题。但是却带来了效率降低。

**为了效率问题，通过双重判断的形式解决。**

class Single{

private static Single s = null;

private Single(){}

public static Single getInstance(){ //**锁是谁？字节码文件对象；**

**if(s == null){**

**synchronized(Single.class){**

**if(s == null)**

**s = new Single();**

**}**

**}**

return s;

}

}

## 什么是锁?锁的作用是什么?

锁就是对象

锁的作用是保证线程同步，解决线程安全问题。

持有锁的线程可以在同步中执行，没有锁的线程即使获得cpu执行权，也进不去。

**同步死锁**：通常只要将同步进行嵌套，就可以看到现象。同步方法中有同步代码块，同步代码块中还有同步方法。

## 多线程间通讯：

**多线程间通讯就是多个线程在操作同一资源,但是操作的动作不同.**

**线程间通信**：思路：多个线程在操作同一个资源，但是操作的动作却不一样。

1：将资源封装成对象。

2：将线程执行的任务(任务其实就是run方法。)也封装成对象。

(1)为什么要通信

多线程并发执行的时候, 如果需要指定线程等待或者唤醒指定线程, 那么就需要通信.比如生产者消费者的问题，

生产一个消费一个,生产的时候需要负责消费的进程等待,生产一个后完成后需要唤醒负责消费的线程,

同时让自己处于等待，消费的时候负责消费的线程被唤醒，消费完生产的产品后又将等待的生产线程唤醒，

然后使自己线程处于等待。这样来回通信，以达到生产一个消费一个的目的。

(2)怎么通信

在同步代码块中, 使用锁对象的wait()方法可以让当前线程等待, 直到有其他线程唤醒为止.

使用锁对象的notify()方法可以唤醒一个等待的线程，或者notifyAll唤醒所有等待的线程.

多线程间通信用sleep很难实现，睡眠时间很难把握。

**等待唤醒机制：**涉及的方法：

**wait:**将同步中的线程处于冻结状态。释放了执行权，释放了资格。同时将线程对象存储到线程池中。

**notify：**唤醒线程池中某一个等待线程。是一个静态方法，调用此方法要捕捉InterruptedException异常。

**notifyAll:**唤醒的是线程池中的所有线程。注意的是在调用此方法的时候，并不能确切的唤醒某一个等待状态的线程，而是由JVM确定唤醒哪个线程(一般是最先开始等待的线程)，而且不是按优先级。

注意：

1：**这些方法都需要定义在同步中**。

2：因为这些方法必须要标示所属的锁。

你要知道 A锁上的线程被wait了,那这个线程就相当于处于A锁的线程池中，只能A锁的notify唤醒。

3：这三个方法都定义在Object类中。为什么操作线程的方法定义在Object类中？

因为这三个方法都需要定义同步内，并标示所属的同步锁，既然被锁调用，而锁又可以是任意对象，那么能被任意对象调用的方法一定定义在Object类中。

**wait和sleep区别：**

(1)这两个方法来自不同的类，sleep()来自Thread类，和wait()来自Object类。

(2)sleep是Thread的静态类方法，谁调用的谁去睡觉，即使在a线程里调用了b的sleep方法，实际上还是a去睡觉，要让b线程睡觉要在b的代码中调用sleep。而wait()是Object类的非静态方法

(3)sleep()释放资源不释放锁，而wait()释放资源释放锁；

(4)使用范围：wait,notify和notifyAll只能在同步控制方法或者同步控制块里面使用,而sleep可以在任何地方使用

wait：可以指定时间也可以不指定时间。不指定时间，只能由对应的notify或者notifyAll来唤醒。

sleep：必须指定时间，时间到自动从冻结状态转成运行状态(临时阻塞状态)。

**线程的停止**：通过stop方法就可以停止线程。但是这个方式过时了。

停止线程：原理就是：让线程运行的代码结束，也就是结束run方法。

怎么结束run方法？一般run方法里肯定定义循环。所以只要结束循环即可。

第一种方式：**定义循环的结束标记。**

第二种方式：如果线程处于了冻结状态，是不可能读到标记的，这时就需要通过Thread类中的interrupt方法，将其冻结状态强制清除。让线程恢复具备执行资格的状态，让线程可以读到标记，并结束。

**---------< java.lang.Thread >----------**

**interrupt()：**中断线程。

**setPriority(int newPriority)：**更改线程的优先级。

**getPriority()：**返回线程的优先级。

**toString()：**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。

**Thread.yield()：**暂停当前正在执行的线程对象，并执行其他线程。

**setDaemon(true)：**将该线程标记为守护线程或用户线程。将该线程标记为守护线程或用户线程。当正在运行的线程都是守护线程时，Java 虚拟机退出。该方法必须在启动线程前调用。

**join**：临时加入一个线程的时候可以使用join方法。

当A线程执行到了B线程的join方式。A线程处于冻结状态，释放了执行权，B开始执行。A什么时候执行呢？只有当B线程运行结束后，A才从冻结状态恢复运行状态执行。

## Lock和Condition

实现提供比synchronized方法和语句可获得的更广泛的锁的操作，可支持多个相关的Condition对象

Lock是个接口

锁是控制多个线程对共享数据进行访问的工具。

**LOCK的出现替代了同步：lock.lock();………lock.unlock();**

解决线程安全问题使用同步的形式，(同步代码块，要么同步方法)其实最终使用的都是锁机制。

到了后期版本，直接将锁封装成了对象。线程进入同步就是具备了锁，执行完，离开同步，就是释放了锁。

在后期对锁的分析过程中，发现，获取锁，或者释放锁的动作应该是锁这个事物更清楚。所以将这些动作定义在了锁当中，并把锁定义成对象。

所以**同步是隐示的锁操作，而Lock对象是显示的锁操作**，它的出现就替代了同步。

在之前的版本中使用Object类中wait、notify、notifyAll的方式来完成的。那是因为同步中的锁是任意对象，所以操作锁的等待唤醒的方法都定义在Object类中。

而现在锁是指定对象Lock。所以查找等待唤醒机制方式需要通过Lock接口来完成。而Lock接口中并没有直接操作等待唤醒的方法，而是将这些方式又单独封装到了一个对象中。这个对象就是**Condition**，将Object中的三个方法进行单独的封装。并提供了功能一致的方法 **await()、signal()、signalAll()**体现新版本对象的好处。

Lock的方法摘要：

void lock() 获取锁。

Condition newCondition() 返回绑定到此 Lock 实例的新 Condition 实例。

void unlock() 释放锁。

Condition方法摘要：

void await() 造成当前线程在接到信号或被中断之前一直处于等待状态。

void signal() 唤醒一个等待线程。

void signalAll() 唤醒所有等待线程。

< java.util.concurrent.locks > Condition接口：await()、signal()、signalAll()；

====================================================================

class BoundedBuffer {

**final Lock lock = new ReentrantLock();**

final Condition notFull = **lock.newCondition();**

final Condition notEmpty = **lock.newCondition();**

final Object[] items = new Object[100];

int putptr, takeptr, count;

public void put(Object x) throws InterruptedException {

**lock.lock();**

**try {**

while (count == items.length)

**notFull.await();**

items[putptr] = x;

if (++putptr == items.length) putptr = 0;

++count;

**notEmpty.signal();**

**}**

**finally {**

**lock.unlock();**

**}**

}

public Object take() throws InterruptedException {

**lock.lock();**

**try {**

while (count == 0)

**notEmpty.await();**

Object x = items[takeptr];

if (++takeptr == items.length) takeptr = 0;

--count;

**notFull.signal();**

return x;

**}**

**finally {**

**lock.unlock();**

**}**

}

}

停止线程：

stop方法已经过时，如何停止线程？

停止线程的方法只有一种，就是run方法结束。如何让run方法结束呢？

开启多线程运行，运行代码通常是循环体，只要控制住循环，就可以让run方法结束，也就是结束线程。

特殊情况：当线程属于冻结状态，就不会读取循环控制标记，则线程就不会结束。

为解决该特殊情况，可引入Thread类中的Interrupt方法结束线程的冻结状态；

当没有指定的方式让冻结线程恢复到运行状态时，需要对冻结进行清除，强制让线程恢复到运行状态

22、interrupt:

void interrupt() 中断线程:

中断状态将被清除，它还将收到一个 InterruptedException

22、守护线程(后台线程)

setDaemon(boolean on):将该线程标记为守护线程或者用户线程。

当主线程结束，守护线程自动结束，比如圣斗士星矢里面的守护雅典娜，

在多线程里面主线程就是雅典娜，守护线程就是圣斗士，主线程结束了，

守护线程则自动结束。

当正在运行的线程都是守护线程时，java虚拟机jvm退出；所以该方法必须在启动线程前调用；

守护线程的特点：

守护线程开启后和前台线程共同抢夺cpu的执行权，开启、运行两者都没区别，

但结束时有区别，当所有前台线程都结束后，守护线程会自动结束。

23、多线程join方法：

void join() 等待该线程终止。

void join(long millis) 等待该线程终止的时间最长为 millis 毫秒。

throws InterruptedException

特点：当A线程执行到B线程的join方法时，A就会等待B线程都执行完，A才会执行

作用: join可以用来临时加入线程执行；

24、多线程优先级：yield()方法

yield():暂停当前正在执行的线程对象，并执行其他线程

setPriority(int newPriority):更改线程优先级

int getPriority() 返回线程的优先级。

String toString() 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

(1)MAX\_PRIORITY:最高优先级(10级)

(1)Min\_PRIORITY:最低优先级(1级)

(1)Morm\_PRIORITY:默认优先级(5级)

## 什么是ThreadLocal类,怎么使用它？

ThreadLocal类提供了线程局部 (thread-local) 变量。是一个线程级别的局部变量，并非“本地线程”。

ThreadLocal 为每个使用该变量的线程,提供了一个独立的变量副本，每个线程修改副本时不影响其它线程对象的副本

下面是线程局部变量(ThreadLocal variables)的关键点：

一个线程局部变量(ThreadLocal variables)为每个线程方便地提供了一个单独的变量。

ThreadLocal 实例通常作为静态的私有的(private static)字段出现在一个类中，这个类用来关联一个线程。

当多个线程访问 ThreadLocal 实例时，每个线程维护 ThreadLocal 提供的独立的变量副本。

常用的使用可在 DAO 模式中见到，当 DAO 类作为一个单例类时，

数据库链接(connection)被每一个线程独立的维护，互不影响。(基于线程的单例)

## 什么时候抛出InvalidMonitorStateException异常?为什么？

调用 wait ()/notify ()/notifyAll ()中的任何一个方法时，如果当前线程没有获得该对象的锁，

那么就会抛出 IllegalMonitorStateException 的异常

也就是说程序在没有执行对象的任何同步块或者同步方法时，

仍然尝试调用 wait ()/notify ()/notifyAll ()时。由于该异常是 RuntimeExcpetion 的子类，

所以该异常不一定要捕获(尽管你可以捕获只要你愿意

作为 RuntimeException，此类异常不会在 wait (),notify (),notifyAll ()的方法签名提及。

# 新特性

## jdk1.5的新特性

### 增强for循环：

(1)JDK1.5新特性，代替迭代器使用时的不爽，简化书写，底层原理是迭代器凡是支持迭代器的都支持高级for循环

高级for循环，只用于集合和数组的遍历，集合只能用Collection不能用Map集合

只能把Map集合转化成Set集合，才能用for循环。

(2)格式

for(数据类型 变量名:被遍历的集合(Collection)或者数组)

{

}

\*\*语法：

for ( type 变量名：集合变量名 ) { … }

\*\*注意事项：

迭代变量必须在( )中定义！

集合变量可以是数组或实现了Iterable接口的集合类

\*\*举例：

public static int add(int x,int ...args) {

int sum = x;

for(int arg:args) {

sum += arg;

}

return sum;

}

\*\*增强for循环代替了迭代器使用的不爽，简化书写

\*\*增强for循环局限性：

对集合或者数组进行遍历时，只能取元素，不能对集合进行操作

foreach语句，foreach简化了迭代器。

格式：// 增强for循环括号里写两个参数，第一个是声明一个变量，第二个就是需要迭代的容器

**for( 元素类型 变量名 : Collection集合 & 数组 ) {**

**„**

**}**

**高级for循环和传统for循环的区别：**

高级for循环在使用时，必须要明确被遍历的目标。这个目标，可以是Collection集合或者数组，如果遍历Collection集合，在遍历过程中还需要对元素进行操作，比如删除，

**需要使用迭代器。**

如果遍历数组，还需要对数组元素进行操作，建议用传统for循环因为可以定义角标通过角标操作元素。如果只为遍历获取，可以简化成高级for循环，它的出现为了简化书写。

#### 高级for循环可以遍历map集合吗？

不可以。但是可以将map转成set后再使用foreach语句。

**1、作用：**对存储对象的容器进行迭代： 数组  collection   map

**2)、增强for循环迭代数组：**

String [] arr = {"a", "b", "c"};//数组的静态定义方式，只试用于数组首次定义的时候

for(String s : arr) {

System.out.println(s);

}

**3)、单列集合 Collection：**

List list = new ArrayList();

list.add("aaa");

// 增强for循环, 没有使用泛型的集合能不能使用增强for循环迭代？能

**for(Object obj : list) {**

**String s = (String) obj; //强转**

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

**}**

**4)、双列集合 Map：**

Map map = new HashMap();

 map.put("a", "aaa");

**// 传统方式：必须掌握这种方式**

**Set entrys = map.entrySet(); // 1.获得所有的键值对Entry对象**

**iter = entrys.iterator(); // 2.迭代出所有的**

**entry while(iter.hasNext()) {**

**Map.Entry entry = (Entry) iter.next();**

**String key = (String) entry.getKey(); // 分别获得key和value**

**String value = (String) entry.getValue();**

**System.out.println(key + "=" + value);**

**}**

**//这个方式不行??????**

**//Set entrySet = map.entrySet(); //把键值对放到set集合中**

**//Iterator it = entrySet.iterator(); //迭代**

**//while(it.hasNext()) {**

**//Map.Entry  me = (Map.Entry)it.next(); //取键值对**

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

**//}**

**// 增强for循环迭代：**原则上map集合是无法使用增强for循环来迭代的，因为增强for循环只能针对实现了Iterable接口的集合进行迭代；Iterable是jdk5中新定义的接口，就一个方法iterator方法，只有实现了Iterable接口的类，才能保证一定有iterator方法，java有这样的限定是因为增强for循环内部还是用迭代器实现的，而实际上，我们可以通过某种方式来使用增强for循环。

**for(Object obj : map.entrySet()) {**

**Map.Entry entry = (Entry) obj;  // obj 依次表示Entry**

**System.out.println(entry.getKey() + "=" + entry.getValue());**

**}**

#### 集合迭代注意问题：

**在迭代集合的过程中，不能对集合进行增删操作（会报并发访问异常）；可以用迭代器的方法进行操作（子类listIterator：有增删的方法）。**

##### 6)、增强for循环注意问题：

**在使用增强for循环时，不能对元素进行赋值；**

**int[] arr = {1,2,3};**

**for(int num : arr) {**

**num = 0; //不能改变数组的值 }**

**System.out.println(arr[1]); //2**

#### (3)局限性：

必须要有遍历的目标

对集合或者数组进行遍历时，只能获取集合元素，不能对集合元素进行操作

迭代器除了遍历，还可以进行remove操作集合中的元素

列表迭代器还可以在遍历过程中进行增删改查的操作

(4)传统for循环和高级for循环的区别

高级for循环有一个局限性，就是必须要有遍历的目标(集合或者数组)

遍历数组时建议使用传统for循环，因为可以定义角标，比如打印100次helloworld时用传统for循环方便

### (4)基本数据类型的自动装箱和拆箱

\*\*基本数据类型

byte ---> Byte

short ---> Short

int ---> Integer

long ---> Long

float ---> Float

double ---> Double

char ---> Character

boolean ---> Boolean

\*\*例子：

\*\*装箱：自动把一个基本数据类型的数据装箱成一个该类型数据的对象引用

Integer i = 3;(jdk1.5之前这样写是不行的，编译报错)

\*\*拆箱：自动把一个基本数据类型的对象引用拆箱成一个基本数据类型的数据，再参与运算

Integer i = 12;

sop(i+4);

\*\*享元模式：

Integer num1 = 12;

Integer num2 = 12;

System.out.println(num1 == num2);//打印true

Integer num5 = Integer.valueOf(12);

Integer num6 = Integer.valueOf(12);

System.out.println(num5 == num6);//打印true

Integer num3 = 129;

Integer num4 = 129;

System.out.println(num3 == num4);//打印false

为什么前面的返回true而后面的运算返回false呢？

对于基本数据类型的整数，装箱成Integer对象时，如果该数值在一个字节内,(-128~127)，

一旦装箱成Integer对象后，就把它缓存到磁里面，当下次，又把该数值封装成Integer对象时

会先看磁里面有没有该对象，有就直接拿出来用，这样就节省了内存空间。因为比较小的整数，

用的频率比较高，就没必要每个对象都分配一个内存空间。

这就是享元模式！比如26个英文字母，10个阿拉伯数字

### (5)枚举

\*\*为什么要有枚举？

问题：要定义星期几或性别的变量，该怎么定义？假设用1-7分别表示星期一到星期日，

但有人可能会写成int weekday = 0;或即使使用常量方式也无法阻止意外。

枚举就是要让某个类型的变量的取值只能为若干个固定值中的一个，否则，编译器就会报错。

枚举可以让编译器在编译时就可以控制源程序中填写的非法值，

普通变量的方式在开发阶段无法实现这一目标。

\*\*用普通类如何实现枚举的功能？定义一个Weekday类来模拟实现：

步骤：

\*私有化构造方法

\*每个元素分别用一个公有的静态成员变量表示(public static final)

\*可以有若干公有方法或抽象方法。采用抽象方法定义nextDay就将大量的if.else语句

转移成了一个个独立的类。

\*\*枚举的应用：

举例：定义一个Weekday的枚举。

扩展：枚举类的values,valueOf,name,toString,ordinal等方法

（记住，讲课时要先于自定义方法前介绍，讲课更流畅）

总结：枚举是一种特殊的类，其中的每个元素都是该类的一个实例对象。

例如可以调用WeekDay.SUN.getClass().getName和WeekDay.class.getName()。

\*\*枚举的高级应用：

\*\*枚举就相当于一个类，其中也可以定义构造方法、成员变量、普通方法和抽象方法。

\*\*枚举元素必须位于枚举体中的最开始部分，枚举元素列表的后要有分号与其他成员分隔。

把枚举中的成员方法或变量等放在枚举元素的前面，编译器报告错误。

\*\*带构造方法的枚举

构造方法必须定义成私有的

如果有多个构造方法，该如何选择哪个构造方法？

枚举元素MON和MON()的效果一样，都是调用默认的构造方法。

\*\*带方法的枚举

定义枚举TrafficLamp

实现普通的next方法

实现抽象的next方法：每个元素分别是由枚举类的子类来生成的实例对象，

这些子类采用类似内部类的方式进行定义。增加上表示时间的构造方法

\*\*枚举只有一个成员时，就可以作为一种单例的实现方式。

### (6)泛型：

\*\*泛型是提供给javac编译器使用的，可以限定集合中的输入类型，让编译器挡住源程序中的非法输入，

编译器编译带类型说明的集合时会去除掉“类型”信息，使程序运行效率不受影响，

对于参数化的泛型类型，getClass()方法的返回值和原始类型完全一样。

由于编译生成的字节码会去掉泛型的类型信息，只要能跳过编译器，

就可以往某个泛型集合中加入其它类型的数据，例如，用反射得到集合，再调用其add方法即可。

\*\*ArrayList<E>类定义和ArrayList<Integer>类引用中涉及如下术语：

整个称为ArrayList<E>泛型类型

ArrayList<E>中的E称为类型变量或类型参数

整个ArrayList<Integer>称为参数化的类型

ArrayList<Integer>中的Integer称为类型参数的实例或实际类型参数

ArrayList<Integer>中的<>念着typeof

ArrayList称为原始类型

\*\*参数化类型与原始类型的兼容性：

参数化类型可以引用一个原始类型的对象，编译报告警告，

例如，Collection<String> c = new Vector();//可不可以，不就是编译器一句话的事吗？

原始类型可以引用一个参数化类型的对象，编译报告警告，

例如，Collection c = new Vector<String>();//原来的方法接受一个集合参数，新的类型也要能传进去

\*\*参数化类型不考虑类型参数的继承关系：

Vector<String> v = new Vector<Object>(); //错误!///不写<Object>没错，写了就是明知故犯

Vector<Object> v = new Vector<String>(); //也错误!

编译器不允许创建泛型变量的数组。即在创建数组实例时，

数组的元素不能使用参数化的类型，

例如，下面语句有错误：

Vector<Integer> vectorList[] = new Vector<Integer>[10];

\*\*泛型限定：

\*\*限定通配符的上边界：

正确：Vector<? extends Number> x = new Vector<Integer>();

错误：Vector<? extends Number> x = new Vector<String>();

\*\*限定通配符的下边界：

正确：Vector<? super Integer> x = new Vector<Number>();

错误：Vector<? super Integer> x = new Vector<Byte>();

\*\*提示：

限定通配符总是包括自己。

?只能用作引用，不能用它去给其他变量赋值

Vector<? extends Number> y = new Vector<Integer>();

Vector<Number> x = y;

上面的代码错误，原理与Vector<Object > x11 = new Vector<String>();相似，

只能通过强制类型转换方式来赋值。

**Jdk5.0新特性：**

Collection在jdk1.5以后，有了一个父接口Iterable，这个接口的出现的将iterator方法进行抽取，提高了扩展性。

### 可变参数（...）：

用到函数的参数上，当要操作的同一个类型元素个数不确定的时候，可是用这个方式，这个参数可以接受任意个数的同一类型的数据。

**和以前接收数组不一样的是：**

以前定义数组类型，需要先创建一个数组对象，再将这个数组对象作为参数传递给函数。现在，直接将数组中的元素作为参数传递即可。底层其实是将这些元素进行数组的封装，而这个封装动作，是在底层完成的，被隐藏了。所以简化了用户的书写，少了调用者定义数组的动作。

如果在参数列表中使用了可变参数，**可变参数必须定义在参数列表结尾(也就是必须是最后一个参数，否则编译会失败。)。**

如果要获取多个int数的和呢？可以使用将多个int数封装到数组中，直接对数组求和即可。

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

**静态导入：**导入了类中的所有静态成员，简化静态成员的书写。

import static java.util.Collections.\*;  //导入了Collections类中的所有静态成员

### 枚举：关键字 enum

**问题：对象的某个属性的值不能是任意的，必须为固定的一组取值其中的某一个；**

 解决办法：

1.在setGrade方法中做判断，不符合格式要求就抛出异常；

2. 直接限定用户的选择，通过自定义类模拟枚举的方式来限定用户的输入，写一个Grade类，私有构造函数，对外提供5个静态的常量表示类的实例；

3,jdk5中新定义了枚举类型，专门用于解决此类问题；

4,枚举就是一个特殊的java类，可以定义属性、方法、构造函数、实现接口、继承类；

### 自动拆装箱：

java中数据类型分为两种 ： 基本数据类型   引用数据类型(对象)

在 java程序中所有的数据都需要当做对象来处理，针对8种基本数据类型提供了包装类，如下：

int --> Integer

byte --> Byte

 short --> Short

long --> Long

char --> Character

double --> Double

float --> Float

boolean --> Boolean

jdk5以前基本数据类型和包装类之间需要互转： 基本---引用   Integer x = new Integer(x); 引用---基本   int num = x.intValue();

1)、Integer x = 1; x = x + 1;  经历了什么过程？装箱  拆箱  装箱；

2)、为了优化，虚拟机为包装类提供了缓冲池，**Integer池的大小 -128~127 一个字节的大小；**

3)、**String池：**Java为了优化字符串操作 提供了一个缓冲池；

## Jdk1.5的

#### 1. 增强for循环

foreach语句，foreach简化了迭代器。

格式：// 增强for循环括号里写两个参数，第一个是声明一个变量，第二个就是需要迭代的容器

for( 元素类型 变量名 : Collection集合 & 数组 ) {

…

}

高级for循环和传统for循环的区别：

高级for循环在使用时，必须要明确被遍历的目标。这个目标，可以是Collection集合或者数组，如果遍历Collection集合，在遍历过程中还需要对元素进行操作，比如删除，需要使用迭代器。

如果遍历数组，还需要对数组元素进行操作，建议用传统for循环因为可以定义角标通过角标操作元素。如果只为遍历获取，可以简化成高级for循环，它的出现为了简化书写。

高级for循环可以遍历map集合吗？不可以。但是可以将map转成set后再使用foreach语句。

1)、作用：对存储对象的容器进行迭代： 数组 collection map

2)、增强for循环迭代数组：

String [] arr = {"a", "b", "c"};//数组的静态定义方式，只试用于数组首次定义的时候

for(String s : arr) {

System.out.println(s);

}

3)、单列集合 Collection：

List list = new ArrayList();

list.add("aaa");

// 增强for循环, 没有使用泛型的集合能不能使用增强for循环迭代？能

for(Object obj : list) {

String s = (String) obj;

System.out.println(s);

}

4)、双列集合 Map：

Map map = new HashMap();

map.put("a", "aaa");

// 传统方式：必须掌握这种方式

Set entrys = map.entrySet(); // 1.获得所有的键值对Entry对象

iter = entrys.iterator(); // 2.迭代出所有的entry

while(iter.hasNext()) {

Map.Entry entry = (Entry) iter.next();

String key = (String) entry.getKey(); // 分别获得key和value

String value = (String) entry.getValue();

System.out.println(key + "=" + value);

}

// 增强for循环迭代：原则上map集合是无法使用增强for循环来迭代的，因为增强for循环只能针对实现了Iterable接口的集合进行迭代；Iterable是jdk5中新定义的接口，就一个方法iterator方法，只有实现了Iterable接口的类，才能保证一定有iterator方法，java有这样的限定是因为增强for循环内部还是用迭代器实现的，而实际上，我们可以通过某种方式来使用增强for循环。

for(Object obj : map.entrySet()) {

Map.Entry entry = (Entry) obj; // obj 依次表示Entry

System.out.println(entry.getKey() + "=" + entry.getValue());

}

5）、集合迭代注意问题：在迭代集合的过程中，不能对集合进行增删操作（会报并发访问异常）；可以用迭代器的方法进行操作（子类listIterator：有增删的方法）。

6)、增强for循环注意问题：在使用增强for循环时，不能对元素进行赋值；

int[] arr = {1,2,3};

for(int num : arr) {

num = 0; //不能改变数组的值

}

System.out.println(arr[1]); //2

#### 2. 可变参数

\*\*可变参数的特点：

\*可变参数只能出现在参数列表的最后；

\*...位于变量类型和变量名之间，前后有无空格都可以;

\*调用可变参数的方法时，编译器为该可变参数隐含创建一个数组，

在方法体中以数组的形式访问可变参数。

用到方法的参数上，当要操作的同一个类型元素个数不确定的时候，可是用这个方式，这个参数可以接受任意个数的同一类型的数据。

和以前接收数组不一样的是：

以前定义数组类型，需要先创建一个数组对象，再将这个数组对象作为参数传递给方法。现在，直接将数组中的元素作为参数传递即可。底层其实是将这些元素进行数组的封装，而这个封装动作，是在底层完成的，被隐藏了。所以简化了用户的书写，少了调用者定义数组的动作。

如果在参数列表中使用了可变参数，可变参数必须定义在参数列表结尾(也就是必须是最后一个参数，否则编译会失败。)。

如果要获取多个int数的和呢？可以使用将多个int数封装到数组中，直接对数组求和即可。

\*\*可变参数举例：

\*变量类型... 变量名 如 int... arr 表示可变参数数组

\*public static void show(String str , int... arr){}

#### 3. 静态导入

\*\*import语句可以导入一个类或某个包中的所有类

\*\*import static语句导入一个类中的某个静态方法或所有静态方法

静态导入后，静态方法前面就不用写类名.方法的方式类调用

\*\*语法举例：

import static java.lang.Math.sin;//导入一个静态方法

import static java.lang.Math.\*; //导入一个类中的所有静态方法

静态导入：导入了类中的所有静态成员，简化静态成员的书写。

import static java.util.Collections.\*; //导入了Collections类中的所有静态成员

\*\*当类名重名时，需要制定具体的包名

\*\*当方法重名时，需要制定具体所属的对象或者类

#### 4. 枚举

枚举：关键字 enum

问题：对象的某个属性的值不能是任意的，必须为固定的一组取值其中的某一个；

解决办法：

1）、在setGrade方法中做判断，不符合格式要求就抛出异常；

2）、直接限定用户的选择，通过自定义类模拟枚举的方式来限定用户的输入，写一个Grade类，私有构造方法，对外提供5个静态的常量表示类的实例；

3)、jdk5中新定义了枚举类型，专门用于解决此类问题；

4)、枚举就是一个特殊的java类，可以定义属性、方法、构造方法、实现接口、继承类；

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

#### 5. 自动拆装箱

java中数据类型分为两种 ： 基本数据类型 引用数据类型(对象)

在 java程序中所有的数据都需要当做对象来处理，针对8种基本数据类型提供了包装类，如下：

int --> Integer byte --> Byte short --> Short long --> Long char --> Character double --> Double float --> Float boolean --> Boolean

jdk5以前基本数据类型和包装类之间需要互转：

基本---引用 Integer x = new Integer(x);

引用---基本 int num = x.intValue();

1)、Integer x = 1; x = x + 1; 经历了什么过程？装箱  拆箱  装箱；

2)、为了优化，虚拟机为包装类提供了缓冲池，Integer池的大小 -128~127 一个字节的大小；

3)、String池：Java为了优化字符串操作 提供了一个缓冲池；

九、 泛型

jdk1.5版本以后出现的一个安全机制。表现格式：< >

好处：

1：将运行时期的问题ClassCastException问题转换成了编译失败，体现在编译时期，程序员就可以解决问题。

2：避免了强制转换的麻烦。

只要带有<>的类或者接口，都属于带有类型参数的类或者接口，在使用这些类或者接口时，必须给<>中传递一个具体的引用数据类型。

ArrayList<String> al = new ArrayList<String>;

#### 泛型技术：

其实应用在编译时期，是给编译器使用的技术，到了运行时期，泛型就不存在了。

为什么? 因为泛型的擦除：也就是说，编辑器检查了泛型的类型正确后，在生成的类文件中是没有泛型的。

在运行时，如何知道获取的元素类型而不用强转呢？

泛型的补偿：

因为存储的时候，类型已经确定了是同一个类型的元素，所以在运行时，只要获取到该元素的类型，在内部进行一次转换即可，所以使用者不用再做转换动作了。

什么时候用泛型类呢？

当类中的操作的引用数据类型不确定的时候，以前用的Object来进行扩展的，现在可以用泛型来表示。这样可以避免强转的麻烦，而且将运行问题转移到的编译时期。

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

泛型在程序定义上的体现：

//泛型类：将泛型定义在类上。

class Tool<Q> {

private Q obj;

public void setObject(Q obj) {

this.obj = obj;

}

public Q getObject() {

return obj;

}

}

//当方法操作的引用数据类型不确定的时候，可以将泛型定义在方法上。

public <W> void method(W w) {

System.out.println("method:"+w);

}

//静态方法上的泛型：静态方法无法访问类上定义的泛型。如果静态方法操作的引用数据类型不确定的时候，必须要将泛型定义在方法上。

public static <Q> void function(Q t) {

System.out.println("function:"+t);

}

//泛型接口.

interface Inter<T> {

void show(T t);

}

class InterImpl<R> implements Inter<R> {

public void show(R r) {

System.out.println("show:"+r);

}

}

#### 通配符

当指定两种泛型的集合，则迭代时也要定义两种泛型的迭代器，麻烦，此时可通过将迭代器的泛型改为？例如：Iterator<?> it = al.iterator();

可以解决当具体类型不确定的时候，这个通配符就是 ? ；当操作类型时，不需要使用类型的具体功能时，只使用Object类中的功能。那么可以用 ? 通配符来表未知类型。

#### 泛型限定：

上限：？extends E：表示这个对象的实例，可以接收E类型或者E的子类型对象。

下限：？super E：可以接收E类型或者E的父类型对象。

上限什么时候用：往集合中添加元素时，既可以添加E类型对象，又可以添加E的子类型对象。为什么？因为取的时候，E类型既可以接收E类对象，又可以接收E的子类型对象。

下限什么时候用：当从集合中获取元素进行操作的时候，可以用当前元素的类型接收，也可以用当前元素的父类型接收。

泛型的细节：

1）、泛型到底代表什么类型取决于调用者传入的类型，如果没传，默认是Object类型；

2）、使用带泛型的类创建对象时，等式两边指定的泛型必须一致；

原因：编译器检查对象调用方法时只看变量，然而程序运行期间调用方法时就要考虑对象具体类型了；

3）、等式两边可以在任意一边使用泛型，在另一边不使用(考虑向后兼容)；

ArrayList<String> al = new ArrayList<Object>(); //错

//要保证左右两边的泛型具体类型一致就可以了，这样不容易出错。

ArrayList<? extends Object> al = new ArrayList<String>();

al.add("aa"); //错，不能加String类型的对象

//因为集合具体对象中既可存储String，也可以存储Object的其他子类，所以添加具体的类型对象不合适，类型检查会出现安全问题。 ？extends Object 代表Object的子类型不确定，怎么能添加具体类型的对象呢？

public static void method(ArrayList<? extends Object> al) {

al.add("abc"); //错

//只能对al集合中的元素调用Object类中的方法，具体子类型的方法都不能用，因为子类型不确定。

}

自定义注解与设计模式

# 快捷键

Alt + Insert 生成代码

Ctrl + I 实现代码

Ctrl + O 覆盖，实现代码

Ctrl+Alt+M 提取方法

# 自定义注解

## 什么是注解？

Jdk1.5新增新技术，注解。很多框架为了简化代码，都会提供有些注解。可以理解为插件，是代码级别的插件，在类的方法上写：@XXX，就是在代码上插入了一个插件。

注解不会也不能影响代码的实际逻辑，仅仅起到辅助性的作用。

**注解分类：内置注解(也成为元注解 jdk 自带注解)、自定义注解（Spring框架）**

## 什么是内置注解

 比如

（1） @SuppressWarnings   再程序前面加上可以在javac编译中去除警告--阶段是SOURCE  
（2） @Deprecated   带有标记的包，方法，字段说明其过时----阶段是SOURCE  
（3）@Overricle   打上这个标记说明该方法是将父类的方法重写--阶段是SOURCE

### 1.1 @Overricle 案例演示

|  |
| --- |
| **@Override**  **public String toString() {**  **return null;**  **}** |

### 1.2 @ Deprecated案例演示

|  |
| --- |
| **new Date().parse("");** |

### 1.3 @ SuppressWarnings  案例演示

|  |
| --- |
| @SuppressWarnings({ "all" })  **public** **void** save() {  java.util.List list = **new** ArrayList();  } |

## 实现自定义注解

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**](mailto:1.@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。

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

[**2.@Retention**](mailto:2.@Retention)

**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）**  
3.@Documented  
4.@Inherited

代码:

使用@interface 定义注解。

|  |
| --- |
| @Target(value = { ElementType.***METHOD***, ElementType.***TYPE*** })  @Retention(RetentionPolicy.***RUNTIME***)  **public** **@interface** OneAnnotation {  **int** beanId() **default** 0;  String className() **default** "";  String[]arrays();  } |

**使用:**

|  |
| --- |
| @OneAnnotation(beanId = 123, className = "className", arrays = { "111", "222" })  **public** **void** add() {  } |

## 实现ORM框架映射

完成案例，ORM框架实体类与表字段不一致,底层生成sql语句原理。

### 自定义表映射注解

|  |
| --- |
| /\*\*  \*  \* **@classDesc**: 功能描述:(**自定义表映射注解** )  \* **@author**: YaoSiyuan  \* **@createTime**: 2017年8月27日 上午12:09:53  \* **@version**: v1.0  \* **@copyright**:  \*/  @Target(value = { ElementType.***TYPE*** })  @Retention(RetentionPolicy.***RUNTIME***)  **public** **@interface** SetTable {  /\*\*  \*  \* **@methodDesc**: 功能描述:(对应数据库表名称)  \* **@author**: YaoSiyuan  \* **@param**: **@return**  \* **@createTime**:2017年8月27日 上午12:10:49  \* **@returnType**:@return String  \* **@copyright**:  \*/  String value();  } |

### 自定义字段属性

|  |
| --- |
| /\*\*  \*  \* @classDesc: 功能描述:(定义字段属性)  \* @author: YaoSiyuan  \* @createTime: 2017年8月27日 上午12:13:32  \* @version: v1.0  \* @copyright:  \*/  @Retention(RetentionPolicy.*RUNTIME*)  public @interface SetProperty {  /\*\*  \*  \* @methodDesc: 功能描述:(字段名称)  \* @author: YaoSiyuan  \* @param: @return  \* @createTime:2017年8月27日 上午12:14:02  \* @returnType:@return String  \* @copyright:  \*/  String name();  /\*\*  \*  \* @methodDesc: 功能描述:(长度)  \* @author: YaoSiyuan  \* @param: @return  \* @createTime:2017年8月27日 上午12:14:25  \* @returnType:@return int  \* @copyright:  \*/  int leng();  } |

### 自定义注解代码实现

|  |
| --- |
| /\*\*  \*  \* @classDesc: 功能描述:(使用java自定义注解 模拟ORM框架注解版本 )  \* @author: YaoSiyuan  \* @createTime: 2017年8月27日 上午12:41:52  \* @version: v1.0  \* @copyright:  \*/  public class Main {  public static void main(String[] args) throws ClassNotFoundException {  // 1.反射class  Class<?> classForName = Class.*forName*("com.entity.Sudent");  // 2.获取表名称注解F  SetTable setTable = classForName.getAnnotation(SetTable.class);  // 3.获取所有的成员属性  Field[] declaredFields = classForName.getDeclaredFields();  StringBuffer sf = new StringBuffer();  sf.append(" select ");  String fromName = setTable.value();  for (int i = 0; i < declaredFields.length; i++) {  Field field = declaredFields[i];  // 4.属性字段  SetProperty sb = field.getAnnotation(SetProperty.class);  sf.append(" " + sb.name() + " ");  if (i == declaredFields.length - 1) {  sf.append(" from ");  } else {  sf.append(" , ");  }  }  sf.append(" " + fromName);  System.*out*.println(sf.toString());  }  } |

# 设计模式

解决问题最行之有效的思想。是规律的总结。

是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性。

## 什么是设计模式？（提高代码复用）

**设计模式（Design pattern）是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性。 毫无疑问，设计模式于己于他人于系统都是多赢的，设计模式使代码编制真正工程化，设计模式是软件工程的基石，如同大厦的一块块砖石一样。项目中合理的运用设计模式可以完美的解决很多问题，每种模式在现在中都有相应的原理来与之对应，每一个模式描述了一个在我们周围不断重复发生的问题，以及该问题的核心解决方案，这也是它能被广泛应用的原因。**

总体来说设计模式分为三大类：

创建型模式，共五种：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。

结构型模式，共七种：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。

行为型模式，共十一种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

其实还有两类：并发型模式和线程池模式。用一个图片来整体描述一下：



## 设计模式的六大原则

1、开闭原则（Open Close Principle）

开闭原则就是说对扩展开放，对修改关闭。在程序需要进行拓展的时候，不能去修改原有的代码，实现一个热插拔的效果。所以一句话概括就是：为了使程序的扩展性好，易于维护和升级。想要达到这样的效果，我们需要使用接口和抽象类，后面的具体设计中我们会提到这点。

2、里氏代换原则（Liskov Substitution Principle）

里氏代换原则(Liskov Substitution Principle LSP)面向对象设计的基本原则之一。 里氏代换原则中说，任何基类可以出现的地方，子类一定可以出现。 LSP是继承复用的基石，只有当衍生类可以替换掉基类，软件单位的功能不受到影响时，基类才能真正被复用，而衍生类也能够在基类的基础上增加新的行为。里氏代换原则是对“开-闭”原则的补充。实现“开-闭”原则的关键步骤就是抽象化。而基类与子类的继承关系就是抽象化的具体实现，所以里氏代换原则是对实现抽象化的具体步骤的规范。—— From Baidu 百科

3、依赖倒转原则（Dependence Inversion Principle）

这个是开闭原则的基础，具体内容：真对接口编程，依赖于抽象而不依赖于具体。

4、接口隔离原则（Interface Segregation Principle）

这个原则的意思是：使用多个隔离的接口，比使用单个接口要好。还是一个降低类之间的耦合度的意思，从这儿我们看出，其实设计模式就是一个软件的设计思想，从大型软件架构出发，为了升级和维护方便。所以上文中多次出现：降低依赖，降低耦合。

5、迪米特法则（最少知道原则）（Demeter Principle）

为什么叫最少知道原则，就是说：一个实体应当尽量少的与其他实体之间发生相互作用，使得系统功能模块相对独立。

6、合成复用原则（Composite Reuse Principle）

原则是尽量使用合成/聚合的方式，而不是使用继承。

## 创建型

### 单例模式

#### 什么是单例模式？

单例保证一个对象JVM中只能有一个实例,常见单例 懒汉式、饿汉式

#### 实现步骤

* 私有化构造方法
* 提供一个公有的静态的获取当前类的实例的方法
* 在静态方法内部保证该实例只有一个

什么是懒汉式,就是什么是饿汉式,就是单例写法

**懒汉式代码**

需要的才会去实例化,线程不安全。

|  |
| --- |
| class SingletonTest {  public static void main(String[] args) {  Singleton sl1 = Singleton.*getSingleton*();  Singleton sl2 = Singleton.*getSingleton*();  System.*out*.println(sl1 == sl2);  }  }  public class Singleton {  // 当需要的才会被实例化  private static Singleton *singleton*;  private Singleton() {  }  synchronized public static Singleton getSingleton() {  if (*singleton* == null) {  *singleton* = new Singleton();  }  return *singleton*;  }  } |

**饿汉式代码**

当class文件被加载的时候，初始化，天生线程安全。

|  |
| --- |
| class SingletonTest1 {  public static void main(String[] args) {  Singleton1 sl1 = Singleton1.*getSingleton*();  Singleton1 sl2 = Singleton1.*getSingleton*();  System.*out*.println((sl1 == sl2)+"-");  }  }  public class Singleton1 {  //当class 文件被加载初始化  private static Singleton1 *singleton* = new Singleton1();  private Singleton1() {  }  public static Singleton1 getSingleton() {  return *singleton*;  }  } |

### 工厂模式

#### 什么是工厂模式？

**实现创建者和调用者分离**

#### 简单工厂代码

|  |
| --- |
| public interface Car {  public void run();  }  public class AoDi implements Car {  @Override  public void run() {  System.*out*.println("奥迪....");  }  }  public interface Car {  public void run();  } |

|  |
| --- |
| **public** **class** CarFactory {  **static** **public** Car createCar(String carName) {  Car car = **null**;  **if** (carName.equals("奥迪")) {  car = **new** AoDi();  } **else** **if** (carName.equals("奔驰")) {  car = **new** BenChi();  }  **return** car;  }  **public** **static** **void** main(String[] args) {  Car car1 = CarFactory.*createCar*("奥迪");  Car car2 = CarFactory.*createCar*("奔驰");  car1.run();  car2.run();  }  } |

#### 工厂方法

|  |
| --- |
| public interface Car {  public void run();  }  public class AoDi implements Car {  @Override  public void run() {  System.*out*.println("奥迪....");  }  }  public class BenChi implements Car {  @Override  public void run() {  System.*out*.println("奔驰....");  }  } |

|  |
| --- |
| public class AoDiChiFactory {  static public Car createCar() {  return new AoDi();  }  }  public interface BenChiFactory {  static public Car createCar() {  return new BenChi();  }  }  public class Main {  public static void main(String[] args) {  Car c1 = AoDiChiFactory.*createCar*();  Car c2 = BenChiFactory.*createCar*();  c1.run();  c2.run();  }  } |

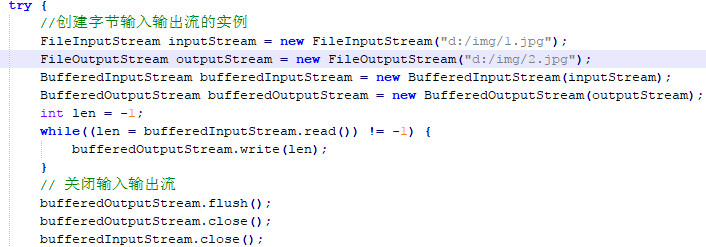
## 结构型

### 装饰者模式：

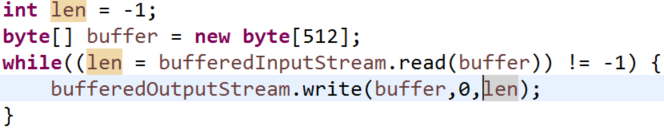
* **BufferInputStream 和BufferOutputStream**
  + **BufferInputSream为inputStream添加超能力，即自带缓冲区**
  + **BufferOutputStream为OutputStream添加超能力，即自带缓冲区**
  + **BufferedinputStream和BufferedOutputStream分别是inputStream和OutputStream的子类，使用装饰者模式对输入流FileinputStream和输出流FileOutStream进行装饰，使得输入流和输出流自带缓冲功能，从而提高效率**
  + **DataInputStream和DataOutputStream是inputStream和OutputStream的子类，使用装饰者模式对输入流输出流进行装饰，是的输入输出流具有可以将java中基本数据类型和字符串输出到输出流或者从输入流中读取的功能**
    - **缓冲区默认设置大小为8192字节**
      * **缓冲区大小可以根据自己需要自己设置**

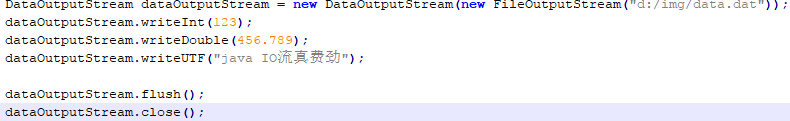
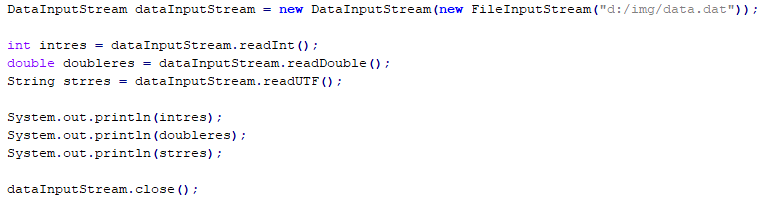
D:\1 - 副本\Evernote_files\Image [286].png

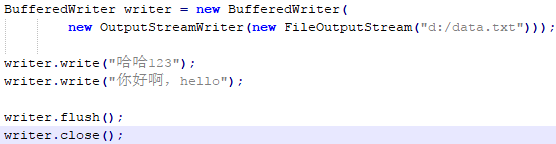
* 方法一：

****

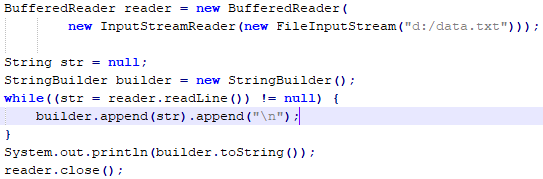
* 方法二：使用小推车



* **DataInputStream和DataOutputStream**
  + **这俩个类是inputStream和OutputStream的子类，使用装饰者模式对输入输出流进行装饰，使得输入输出流具有可以将java中的基本数据类型和字符串输出到输出流或者从输入流中读取的功能**
  + **DatainputStream在读取是，必须按照写入的顺序读取**
  + **DataOutPutStream**
    - **数据输出流允许应用程序以适当方式将基本java数据类型写入输出流中。然后，应用程序可以使用数据输入流将数据读入**
    - **构造方法：**
      * **DataOutputStream(OutputStream out) 创建一个新的数据输出流，将数据写入指定基础输出流**
  + **写入：**
  + **读取：              **
* **BufferedWriter和BufferdeRead**
  + **输出：**

****

* **读取：**

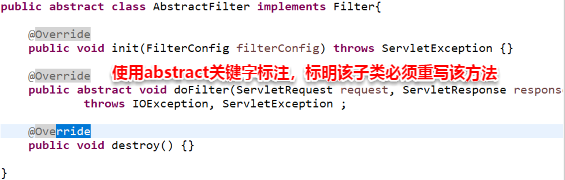
****

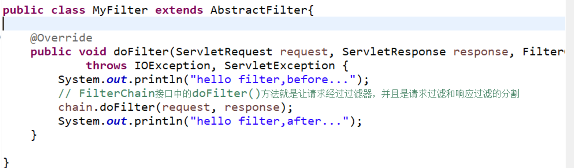
### 适配器模式

为了解决自定义类实现接口，要重写所有抽象方法时，给自定义类定义一个父类(父类实现该接口)

子类就可以根据需要重写方法，为了必须重写父类中的某一方法，将该方法定义为抽象方法即可

例：实现过滤器时：



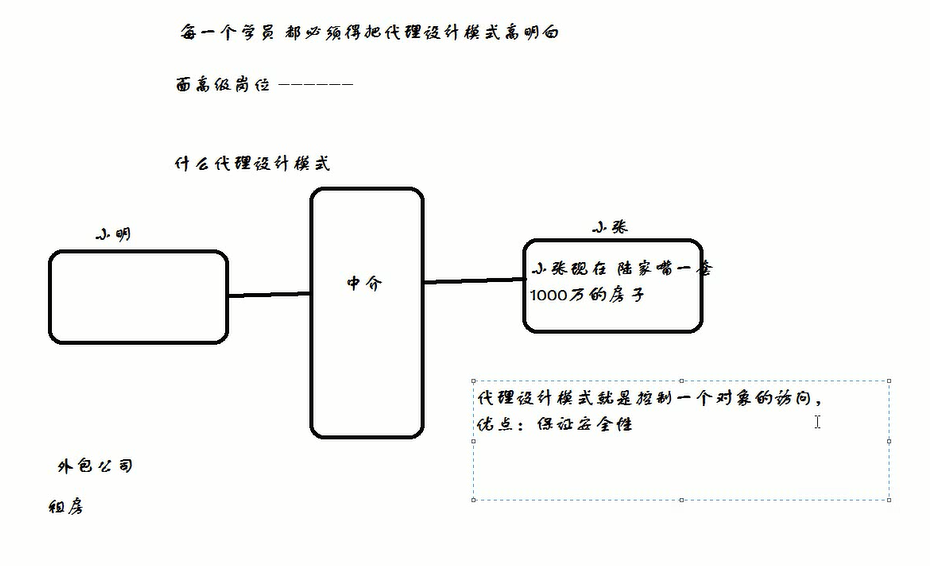


## 代理模式（★★★★★）

### 什么是代理？

通过代理控制对象的访问,可以详细访问某个对象的方法，在这个方法调用处理，或调用后处理。既(AOP微实现)，AOP核心技术面向切面编程。





代理设计就是控制一个对象的访问，

优点：保证安全性。

### 代理应用场景

安全代理 可以屏蔽真实角色

远程代理 远程调用代理类RMI

延迟加载 先加载轻量级代理类,真正需要在加载真实

### 代理的分类

静态代理(静态定义代理类)：

需要自己去生成代理类

动态代理(动态生成代理类)：

不需要自己写，实现类，jdk动态代理，cglib代理

Jdk自带动态代理

Cglib 、javaassist（字节码操作库）

### 静态代理

静态代理需要自己生成代理类

|  |
| --- |
| public class XiaoMing implements Hose {  @Override  public void mai() {  System.*out*.println("我是小明,我要买房啦!!!!haha ");  }  }  class Proxy implements Hose {  private XiaoMing xiaoMing;  public Proxy(XiaoMing xiaoMing) {  this.xiaoMing = xiaoMing;  }  public void mai() {  System.*out*.println("我是中介 看你买房开始啦!");  xiaoMing.mai();  System.*out*.println("我是中介 看你买房结束啦!");  }  public static void main(String[] args) {  Hose proxy = new Proxy(new XiaoMing());  proxy.mai();  }  } |

### JDK动态代理(不需要生成代理类)

实现InvocationHandler 就可以了。

|  |
| --- |
| public interface Hose {  /\*\*  \*  \* @methodDesc: 功能描述:(买房代理)  \* @author: YaoSiyuan  \* @param:  \* @createTime:2017年8月27日 上午2:54:34  \* @returnType: void  \* @copyright:  \*/  public void mai();  }  public class XiaoMing implements Hose {  @Override  public void mai() {  System.*out*.println("我是小明,我要买房啦!!!!haha ");  }  } |

|  |
| --- |
| **public** **class** JDKProxy **implements** InvocationHandler {  **private** Object tarjet;  **public** JDKProxy(Object tarjet) {  **this**.tarjet = tarjet;  }  @Override  **public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {  System.***out***.println("我是房产中介.....开始监听你买房啦!");  Object oj = method.invoke(tarjet, args);  System.***out***.println("我是房产中介.....结束监听你买房啦!");  **return** oj;  }  }  **class** Test222 {  **public** **static** **void** main(String[] args) {  XiaoMing xiaoMing = **new** XiaoMing();  JDKProxy jdkProxy = **new** JDKProxy(xiaoMing);  Hose hose=(Hose) Proxy.*newProxyInstance*(xiaoMing.getClass().getClassLoader(), xiaoMing.getClass().getInterfaces(), jdkProxy);  hose.mai();  }  } |

### CGLIB动态代理

实现

|  |
| --- |
| import java.lang.reflect.Method;  import net.sf.cglib.proxy.Enhancer;  import net.sf.cglib.proxy.MethodInterceptor;  import net.sf.cglib.proxy.MethodProxy;  public class Cglib implements MethodInterceptor {  @Override  public Object intercept(Object o, Method method, Object[] args, MethodProxy methodProxy) throws Throwable {  System.*out*.println("我是买房中介 ， 开始监听你买房了....");  Object invokeSuper = methodProxy.invokeSuper(o, args);  System.*out*.println("我是买房中介 ， 开结束你买房了....");  return invokeSuper;  }  }  class Test22222 {  public static void main(String[] args) {  Cglib cglib = new Cglib();  Enhancer enhancer = new Enhancer();  enhancer.setSuperclass(XiaoMing.class);  enhancer.setCallback(cglib);  Hose hose = (Hose) enhancer.create();  hose.mai();  }  } |

### CGLIB与JDK动态代理区别

jdk动态代理是由Java内部的反射机制来实现的，cglib动态代理底层则是借助asm来实现的。总的来说，反射机制在生成类的过程中比较高效，而asm在生成类之后的相关执行过程中比较高效（可以通过将asm生成的类进行缓存，这样解决asm生成类过程低效问题）。还有一点必须注意：jdk动态代理的应用前提，必须是目标类基于统一的接口。如果没有上述前提，jdk动态代理不能应用。

注:asm其实就是java字节码控制.

# 注解

public class ProImpl implements Pro{

public String className(){

return "cn.itcast.annotation.Demo1";

}

public String methodName(){

return "show";

}

}

3. 调用注解中的抽象方法获取配置的属性值

\* 案例：简单的测试框架

\* 小结：

1. 以后大多数时候，我们会使用注解，而不是自定义注解

2. 注解给谁用？

1. 编译器

2. 给解析程序用

3. 注解不是程序的一部分，可以理解为注解就是一个标签

# Junit单元测试：

## \* 测试分类：

1. 黑盒测试：不需要写代码，给输入值，看程序是否能够输出期望的值。

2. 白盒测试：需要写代码的。关注程序具体的执行流程。

### \* Junit使用：白盒测试

#### \* 步骤：

1. 定义一个测试类(测试用例)

\* 建议：

\* 测试类名：被测试的类名Test CalculatorTest

\* 包名：xxx.xxx.xx.test cn.itcast.test

2. 定义测试方法：可以独立运行

\* 建议：

\* 方法名：test测试的方法名 testAdd()

\* 返回值：void

\* 参数列表：空参

3. 给方法加@Test

4. 导入junit依赖环境

#### \* 判定结果：

\* 红色：失败

\* 绿色：成功

\* 一般我们会使用断言操作来处理结果

\* Assert.assertEquals(期望的结果,运算的结果);

#### \* 补充：

\* @Before:

\* 修饰的方法会在测试方法之前被自动执行

\* @After:

\* 修饰的方法会在测试方法执行之后自动被执行

## 单元测试框架-junit

一些专业名词：

测试用例：单元测试代码

作用：

测试，提高代码的质量

当成开发文档使用

测试模块：只能测试service,dao里的类和方法

servlet可以使用postman插件去测试 表单值放在from-data中

测试覆盖率：几个用例几个方法

进行测试   junit.org/junit4

使用步骤：

引入maven插件

            <groupId>junit</groupId>

            <artifactId>junit</artifactId>

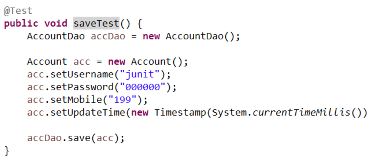
            <version>4.12</version>

            <scope>test</scope>

新建一个类用于测试

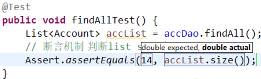
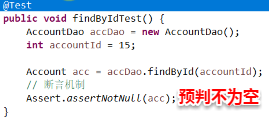
书写方法，在方法上面加注解@Test即可  运行方法(方法上点右键)，

绿色为成功，红色为抛异常/与预想的不一样

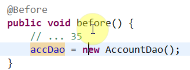


其他用法：

断言机制：



@before  在执行所有方法之前执行



@After  在执行所有方法之后执行

D:\1 - 副本\Evernote_files\Image [154].png

TDD开发，测试优先

静态导入，所有方法必须是静态的 例：

//建议不使用，多个不好分辨

import static java.lang.Math.\*;

# 反射

定义：程序运行的时候，能够创建出对应的类的对象的机制叫做反射机制

其实就是动态加载一个指定的类，并获取该类中的所有的内容。而且将字节码文件封装成对象，并将字节码文件中的内容都封装成对象，这样便于操作这些成员。

简单说：反射技术可以对一个类进行解剖。

## 反射：框架设计的灵魂

\* 框架：半成品软件。可以在框架的基础上进行软件开发，简化编码

\* 反射：将类的各个组成部分封装为其他对象，这就是反射机制

\* 好处：

1. 可以在程序运行过程中，操作这些对象。

2. 可以解耦，提高程序的可扩展性。

\* 获取Class对象的方式：

1. Class.forName("全类名")：将字节码文件加载进内存，返回Class对象

\* 多用于配置文件，将类名定义在配置文件中。读取文件，加载类

2. 类名.class：通过类名的属性class获取

\* 多用于参数的传递

3. 对象.getClass()：getClass()方法在Object类中定义着。

\* 多用于对象的获取字节码的方式

\* 结论：

同一个字节码文件(\*.class)在一次程序运行过程中，只会被加载一次，不论通过哪一种方式获取的Class对象都是同一个。

## Class对象功能：

\* 获取功能：

1. 获取成员变量们

\* Field[] getFields() ：获取所有public修饰的成员变量

\* Field getField(String name) 获取指定名称的 public修饰的成员变量

\* Field[] getDeclaredFields() 获取所有的成员变量，不考虑修饰符

\* Field getDeclaredField(String name)

2. 获取构造方法们

\* Constructor<?>[] getConstructors()

\* Constructor<T> getConstructor(类<?>... parameterTypes)

\* Constructor<T> getDeclaredConstructor(类<?>... parameterTypes)

\* Constructor<?>[] getDeclaredConstructors()

3. 获取成员方法们：

\* Method[] getMethods()

\* Method getMethod(String name, 类<?>... parameterTypes)

\* Method[] getDeclaredMethods()

\* Method getDeclaredMethod(String name, 类<?>... parameterTypes)

4. 获取全类名

\* String getName()

\* Field：成员变量

\* 操作：

1. 设置值

\* void set(Object obj, Object value)

2. 获取值

\* get(Object obj)

3. 忽略访问权限修饰符的安全检查

\* setAccessible(true):暴力反射

\* Constructor:构造方法

\* 创建对象：

\* T newInstance(Object... initargs)

\* 如果使用空参数构造方法创建对象，操作可以简化：Class对象的newInstance方法

\* Method：方法对象

\* 执行方法：

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

\* 获取方法名称：

\* String getName:获取方法名

\* 案例：

\* 需求：写一个"框架"，不能改变该类的任何代码的前提下，可以帮我们创建任意类的对象，并且执行其中任意方法

\* 实现：

1. 配置文件

2. 反射

\* 步骤：

1. 将需要创建的对象的全类名和需要执行的方法定义在配置文件中

2. 在程序中加载读取配置文件

3. 使用反射技术来加载类文件进内存

4. 创建对象

5. 执行方法

## 反射的好处

大大的增强了程序的扩展性。

## 反射的基本步骤：

1、获得Class对象，就是获取到指定的名称的字节码文件对象。

2、实例化对象，获得类的属性、方法或构造方法。

3、访问属性、调用方法、调用构造方法创建对象。

## 获取这个Class对象，有三种方式：

1：通过每个对象都具备的方法getClass来获取。弊端：必须要创建该类对象，才可以调用getClass方法。

2：每一个数据类型(基本数据类型和引用数据类型)都有一个静态的属性class。弊端：必须要先明确该类。

前两种方式不利于程序的扩展，因为都需要在程序使用具体的类来完成。

3：使用的Class类中的方法，静态的forName方法。

指定什么类名，就获取什么类字节码文件对象，这种方式的扩展性最强，只要将类名的字符串传入即可。

// 1. 根据给定的类名来获得 用于类加载

String classname = "cn.itcast.reflect.Person";// 来自配置文件

Class clazz = Class.forName(classname);// 此对象代表Person.class

// 2. 如果拿到了对象，不知道是什么类型 用于获得对象的类型

Object obj = new Person();

Class clazz1 = obj.getClass();// 获得对象具体的类型

// 3. 如果是明确地获得某个类的Class对象 主要用于传参

Class clazz2 = Person.class;

## 反射的用法：

1）、需要获得java类的各个组成部分，首先需要获得类的Class对象，获得Class对象的三种方式：

Class.forName(classname) 用于做类加载

obj.getClass() 用于获得对象的类型

类名.class 用于获得指定的类型，传参用

2)、反射类的成员方法：

Class clazz = Person.class;

Method method = clazz.getMethod(methodName, new Class[]{paramClazz1, paramClazz2});

method.invoke();

3)、反射类的构造方法：

Constructor con = clazz.getConstructor(new Class[]{paramClazz1, paramClazz2,...})

con.newInstance(params...)

4)、反射类的属性：

Field field = clazz.getField(fieldName);

field.setAccessible(true);

field.setObject(value);

获取了字节码文件对象后，最终都需要创建指定类的对象：

创建对象的两种方式(其实就是对象在进行实例化时的初始化方式)：

1，调用空参数的构造方法：使用了Class类中的newInstance()方法。

2，调用带参数的构造方法：先要获取指定参数列表的构造方法对象，然后通过该构造方法的对象的newInstance(实际参数) 进行对象的初始化。

综上所述，第二种方式，必须要先明确具体的构造方法的参数类型，不便于扩展。所以一般情况下，被反射的类，内部通常都会提供一个公有的空参数的构造方法。

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

// 如何生成获取到字节码文件对象的实例对象。

Class clazz = Class.forName("cn.itcast.bean.Person");//类加载

// 直接获得指定的类型

clazz = Person.class;

// 根据对象获得类型

Object obj = new Person("zhangsan", 19);

clazz = obj.getClass();

Object obj = clazz.newInstance();//该实例化对象的方法调用就是指定类中的空参数构造方法，给创建对象进行初始化。

当指定类中没有空参数构造方法时，该如何创建该类对象呢？请看method\_2();

public static void method\_2() throws Exception {

Class clazz = Class.forName("cn.itcast.bean.Person");

//既然类中没有空参数的构造方法,那么只有获取指定参数的构造方法,用该方法来进行实例化。

//获取一个带参数的构造器。

Constructor constructor = clazz.getConstructor(String.class,int.class);

//想要对对象进行初始化，使用构造器的方法newInstance();

Object obj = constructor.newInstance("zhagnsan",30);

//获取所有构造器。

Constructor[] constructors = clazz.getConstructors();//只包含公共的

constructors = clazz.getDeclaredConstructors();//包含私有的

for(Constructor con : constructors) {

System.out.println(con);

}

}

## 反射指定类中的方法：

//获取类中所有的方法。

public static void method\_1() throws Exception {

Class clazz = Class.forName("cn.itcast.bean.Person");

Method[] methods = clazz.getMethods();//获取的是该类中的公有方法和父类中的公有方法。

methods = clazz.getDeclaredMethods();//获取本类中的方法，包含私有方法。

for(Method method : methods) {

System.out.println(method);

}

}

//获取指定方法；

public static void method\_2() throws Exception {

Class clazz = Class.forName("cn.itcast.bean.Person");

//获取指定名称的方法。

Method method = clazz.getMethod("show", int.class,String.class);

//想要运行指定方法，当然是方法对象最清楚，为了让方法运行，调用方法对象的invoke方法即可，但是方法运行必须要明确所属的对象和具体的实际参数。

Object obj = clazz.newInstance();

method.invoke(obj, 39,"hehehe");//执行一个方法

}

//想要运行私有方法。

public static void method\_3() throws Exception {

Class clazz = Class.forName("cn.itcast.bean.Person");

//想要获取私有方法。必须用getDeclearMethod();

Method method = clazz.getDeclaredMethod("method", null);

// 私有方法不能直接访问，因为权限不够。非要访问，可以通过暴力的方式。

method.setAccessible(true);//一般很少用，因为私有就是隐藏起来，所以尽量不要访问。

}

//反射静态方法。

public static void method\_4() throws Exception {

Class clazz = Class.forName("cn.itcast.bean.Person");

Method method = clazz.getMethod("function",null);

method.invoke(null,null);

}

• .class与getClass()的区别：

o .class是在java运行时就加载的，getClass()是运行程序是动态加载的(即运行那个类就显示哪个，与继承无关)

• 使用：没办法创建类的对象时

o 获取类的详细信息而不是对象：

第一种： Class clazz = Class.forName("类的完全限定名")

第二种： Class clazz = 类名.class//Class类的实例表示正在运行的java应用程序中的类和接口

第三种： 例：Account acc = new Account(); Class clazz = acc.getClass();

• getMethod()//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

例：Method method =getMethod("方法名",参数列表中参数类型的详细信息...)

method.getName(）//获取方法名

method.invoke(实例对象,参数赋值)//对带有指定参数的指定对象调用,由此Method对象表示的底层方法

获取基本数据类型的详细信息：Integer.TYPE //表示基本数据类型的Class实例

• getMethods()//获取一个方法数组

• 创建此Class对象所表示的类的实例：newinstance()

例：Object obj = clazz.newInstance();

判断俩个对象是不是一样，使用instanceof关键字 对象A instanceof 对象B

# jvm

## JVM

* + 垃圾回收
    - 堆是垃圾回收管理的主要区域 N/O操作直接内存
    - 分代处理(新生代，老年代) 新生代经过几次GC之后，进入老年代
    - 大小相同，可以互换，互换正在用的对象，清空一个
    - 新声明的对象超出新生代内存，该对象会进入老年代
    - 老年代默认15 新生代 8:1:1
  + 服务器优化：
    - 1.分代
    - 2.标记清理，复制，标记整理
    - 3.CMS。Paraller 虚拟机配置
    - 4.类加载机制流程
      * 主动调用
      * 被动调用
      * 类加载器
      * 双亲委派模型
  + CMS 浮动垃圾
  + GL 最终标记
  + classLoader 类加载
  + 生命周期：
    - 1.加载：查找并加载类的二进制数据
    - 2.连接：
      * 1.验证：确保正确
      * 2.准备：为类的静态变量分配内存，并将起初始化为默认值
      * 3.解析：把类中的符号yiny9ong转化为直接引用
    - 3.初始化：赋予正确的初始值
  + 首次主动使用才会初始化
    - New 自己 反射 静态类
    - New 子类 main
  + 类本身+加载器 ==保证类的唯一性，使用双亲委派模型，自己不支持- 🡪父
  + 方法区默认64M
    - 查看方法区内容：Runtime runtime = Runtime.getRuntime(){}
  + 深入理解java虚拟机 1.2.3.6.7.12.13章

## 自动内存管理机制

### Java虚拟机原理

所谓虚拟机，就是一台虚拟的机器。他是一款软件，用来执行一系列虚拟计算指令，大体上虚拟机可以分为系统虚拟机和程序虚拟机，大名鼎鼎的VisualBox、Vmare就属于系统虚拟机，他们完全是对物理计算的仿真，提供了一个可以运行完整操作系统的软件平台。

程序虚拟机典型代码就是Java虚拟机，它专门为执行单个计算程序而计算，在Java虚拟机中执行的指令我们成为Java自己码指令。无论是系统虚拟机还是程序虚拟机，在上面运行的软件都被限制于虚拟机提供的资源中。

Java发展至今，出现过很多虚拟机，做初Sun使用的一款叫ClassIc的Java虚拟机，到现在引用最广泛的是HotSpot虚拟机，除了Sum意外，还有BEA的Jrockit，目前Jrockit和HostSopt都被oralce收入旗下，大有整合的趋势。

### Java内存结构



static、class类的加载信息，存到方法区（永久区）不会销毁静态标示的字段，也就是方法区中的。不要太多定义static，不会被垃圾回收。占内存。所有线程都会被共享。

只要是new出来的对象都会存在堆里。所有线程都会被共享的。线程安全问题。

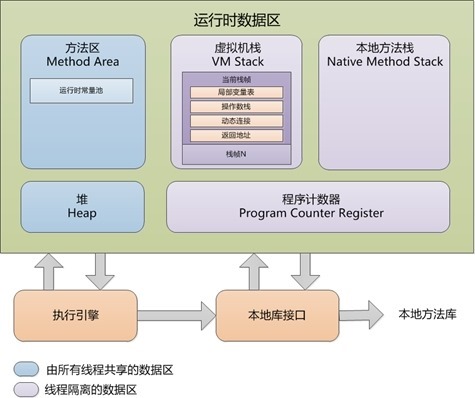
栈：主要存放最基本的临时变量。为什么在多线程情况下，共享一个内存变量不会出现线程安全问题，因为栈每个线程都有自己的本地线程栈。

本地方法栈：

安卓开发，c语言，都会使用java语言，jni，就是java语言去调用c语言。

执行引擎：

怎么去操作堆栈，



类加载子系统:负责从文件系统或者网络加载Class信息，加载的信息存放在一块称之方法区的内存空间。

方法区:就是存放类的信息、常量信息、常量池信息、包括字符串字面量和数字常量等。

Java堆：在Java虚拟机启动的时候建立Java堆，它是Java程序最主要的内存工作区域，几乎所有的对象实例都存放到

Java堆中，堆空间是所有线程共享。

直接内存：JavaNio库允许Java程序直接内存，从而提高性能，通常直接内存速度会优于Java堆。读写频繁的场合可能会考虑使用。

每个虚拟机线程都有一个私有栈，一个线程的Java栈在线程创建的时候被创建，Java栈保存着局部变量、方法参数、同事Java的方法调用、返回值等。

本地方法栈，最大不同为本地方法栈用于本地方法调用。Java虚拟机允许Java直接调用本地方法（通过使用C语言写）

垃圾收集系统是Java的核心，也是不可少的，Java有一套自己进行垃圾清理的机制，开发人员无需手工清理。

PC（ProgramCouneter）寄存器也是每个线程私有的空间，Java虚拟机会为每个线程创建PC寄存器，在任意时刻，一个Java线程总是在执行一个方法，这个方法称为当前方法，如果当前方法不是本地方法，PC寄存器总会执行当前正在被执行的指令，

如果是本地方法，则PC寄存器值为Underfined，寄存器存放如果当前执行环境指针、程序技术器、操作栈指针、计算的变量指针等信息。

虚拟机核心的组件就是执行引擎，它负责执行虚拟机的字节码，一般户先进行编译成机器码后执行。

### 堆、栈、方法区概念区别

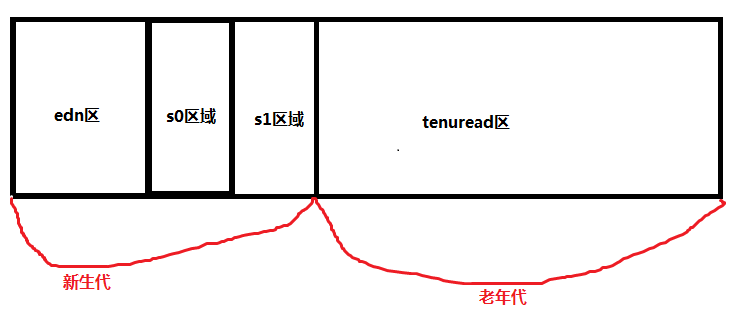
#### Java堆

堆内存用于存放由new创建的对象和数组。在堆中分配的内存，由java虚拟机自动垃圾回收器来管理。在堆中产生了一个数组或者对象后，还可以在栈中定义一个特殊的变量，这个变量的取值等于数组或者对象在堆内存中的首地址，在栈中的这个特殊的变量就变成了数组或者对象的引用变量，以后就可以在程序中使用栈内存中的引用变量来访问堆中的数组或者对象，引用变量相当于为数组或者对象起的一个别名，或者代号。

根据垃圾回收机制的不同，Java堆有可能拥有不同的结构，最为常见的就是将整个Java堆分为新生代和老年代。其中新声带存放新生的对象或者年龄不大的对象，老年代则存放老年对象。

新生代分为den区、s0区、s1区，s0和s1也被称为from和to区域，他们是两块大小相等并且可以互相角色闺女访问日5繁体版国土局，的空间。

绝大多数情况下，对象首先分配在eden区，在新生代回收后，如果对象还存活，则进入s0或s1区，之后每经过一次新生代回收，如果对象存活则它的年龄就加1，对象达到一定的年龄后，则进入老年代。





jvm挑优

新生代：刚new出不久。不经常使用。

老年代。经常被使用的对象。

主要优化堆内存，栈是局部临时，会出栈。

eden：当我们new一个对象出来，首先存放在eden区。

老年代作用：

gc去回收该对象，发现一只在使用没有被释放，达到一定次数，就把eden中的user升级为老年代。

GC算法会先从新生代去回收。再去回收老年代中。新生代回收次数比老年代多。

#### Java栈

Java栈是一块线程私有的空间，一个栈，一般由三部分组成:局部变量表、操作数据栈和帧数据区

局部变量表：用于报错函数的参数及局部变量

操作数栈：主要保存计算过程的中间结果，同时作为计算过程中的变量临时的存储空间。

帧数据区:除了局部变量表和操作数据栈以外，栈还需要一些数据来支持常量池的解析，这里帧数据区保存着

访问常量池的指针，方便计程序访问常量池，另外当函数返回或出现异常时卖虚拟机子必须有一个异常处理表，方便发送异常

的时候找到异常的代码，因此异常处理表也是帧数据区的一部分。

****

#### Java方法区

Java方法区和堆一样，方法区是一块所有线程共享的内存区域，他保存系统的类信息。

比如类的字段、方法、常量池等。方法区的大小决定系统可以保存多少个类。如果系统定义太多的类，导致方法区溢出。虚拟机同样会抛出内存溢出的错误。方法区可以理解为永久区。

### 虚拟机参数配置

#### 什么是虚拟机参数配置

在虚拟机运行的过程中，如果可以跟踪系统的运行状态，那么对于问题的故障排查会有一定的帮助，为此，在虚拟机提供了一些跟踪系统状态的参数，使用给定的参数执行Java虚拟机，就可以在系统运行时打印相关日志，用于分析实际问题。我们进行虚拟机参数配置，其实就是围绕着堆、栈、方法区、进行配置。

你说下你熟悉那些jvm参数调优

#### 堆的参数配置

-XX:+PrintGC每次触发GC的时候打印相关日志

-XX:+UseSerialGC串行回收

-XX:+PrintGCDetails更详细的GC日志

-Xms堆初始值

-Xmx堆最大可用值

-Xmn新生代堆最大可用值

-XX:SurvivorRatio用来设置新生代中eden空间和from/to空间的比例.

含以-XX:SurvivorRatio=eden/from=den/to

总结:在实际工作中，我们可以直接将初始的堆大小与最大堆大小相等，

这样的好处是可以减少程序运行时垃圾回收次数，从而提高效率。

设置比较小的时候，垃圾回收机制会频繁的去回收。每次回收垃圾都会影响程序效率。

-XX:SurvivorRatio用来设置新生代中eden空间和from/to空间的比例.

##### 设置最大堆内存

参数:-Xms5m-Xmx20m-XX:+PrintGCDetails-XX:+UseSerialGC-XX:+PrintCommandLineFlags

-Xms5m-Xmx20m-XX:+PrintGCDetails-XX:+UseSerialGC-XX:+PrintCommandLineFlags

|  |
| --- |
| importjava.text.DecimalFormat;  /\*\*  \*@AuthorYaoSiyuan  \*@Description//查看jvm堆内存大小  \*设置jvm参数-Xms5m-Xmx20m-XX:+PrintGCDetails-XX:+UseSerialGC-XX:+PrintCommandLineFlags  \*@Date19:462019/3/14  \*@Param  \*@return  \*\*/  publicclassMain{  publicstaticvoidmain(String[]args)throwsInterruptedException{  byte[]b1=newbyte[1\*1024\*1024];  System.out.println("分配了1m");  jvmInfo();  Thread.sleep(3000);  byte[]b2=newbyte[4\*1024\*1024];  System.out.println("分配了4m");  Thread.sleep(3000);  jvmInfo();  //System.out.println("HelloWorld!");  //  ////获取到当前运行环境最大的堆内存  //System.out.println("maxMemory:"+Runtime.getRuntime().maxMemory()/(1024\*1024));  //  ////获取到当前运行环境的空闲内存  //System.out.println("freememory:"+Runtime.getRuntime().freeMemory()/(1024\*1024));  //  ////获取到当前使用了多少内存  //System.out.println("totalmemory:"+Runtime.getRuntime().totalMemory()/(1024\*1024));  }  staticprivatevoidjvmInfo(){  //最大内存  longmaxMemory=Runtime.getRuntime().maxMemory();  System.out.println("maxMemory:"+maxMemory+",转换为M:"+toM(maxMemory));  //当前空闲内存  longfreeMemory=Runtime.getRuntime().freeMemory();  System.out.println("freeMemory:"+freeMemory+",转换为M:"+toM(freeMemory));  //已经使用内存  longtotalMemory=Runtime.getRuntime().totalMemory();  System.out.println("totalMemory:"+totalMemory+",转换为M"+toM(totalMemory));  }  /\*\*  \*转换为m  \*  \*@return  \*@parammaxMemory  \*/  staticprivateStringtoM(longmaxMemory){  floatnum=(float)maxMemory/(1024\*1024);  //格式化小数  DecimalFormatdf=newDecimalFormat("0.00");  //返回的是String类型  Strings=df.format(num);  returns;  }  } |

##### 设置新生代与老年代优化参数

**-Xmn新生代大小，一般设为整个堆的1/3到1/4左右**

**-XX:SurvivorRatio设置新生代中eden区和from/to空间的比例关系n/1**

###### 设置新生代比例参数

**参数:-Xms20m-Xmx20m-Xmn1m-XX:SurvivorRatio=2-XX:+PrintGCDetails-XX:+UseSerialGC**

|  |
| --- |
| /\*\*  \*@ClassNamedemo  \*@Description设置新生代参数配置  \*@Authoryaosiyuan  \*@Date2019/3/1420:35  \*@Version1.0  \*\*/  publicclassdemo{  publicstaticvoidmain(String[]args){  //-Xms20m-Xmx20m-Xmn1m-XX:SurvivorRatio=2-XX:+PrintGCDetails-XX:+UseSerialGC  byte[]b=null;  for(inti=0;i<10;i++){  b=newbyte[1\*1024\*1024];  }  }  } |

###### 设置新生与老年代代参数

-Xms20m-Xmx20m-XX:SurvivorRatio=2-XX:+PrintGCDetails-XX:+UseSerialGC

-Xms20m-Xmx20m-Xmn1m-XX:SurvivorRatio=2-XX:+PrintGCDetails-XX:+UseSerialGC**-XX:NewRatio=2**

总结:不同的堆分布情况，对系统执行会产生一定的影响，在实际工作中，

应该根据系统的特点做出合理的配置，基本策略：尽可能将对象预留在新生代，

减少老年代的GC次数。

除了可以设置新生代的绝对大小(-Xmn),可以使用(-XX:NewRatio)设置新生代和老年

代的比例:-XX:NewRatio=老年代/新生代

需求：可能会有很多经常使用的对象。

优化：老年代越大越好。尽量减少老年代的ＧＣ

##### 内存溢出解决办法

###### 设置堆内存大小

错误原因:java.lang.OutOfMemoryError:Javaheapspace

解决办法:设置堆内存大小-Xms1m-Xmx70m-XX:+HeapDumpOnOutOfMemoryError

|  |
| --- |
| publicstaticvoidmain(String[]args)throwsInterruptedException{  List<Object>list=newArrayList<>();  Thread.*sleep*(3000);  *jvmInfo*();  for(inti=0;i<10;i++){  System.*out*.println("i:"+i);  Byte[]bytes= newByte[1\*1024\*1024];  list.add(bytes);  *jvmInfo*();  }  System.*out*.println("添加成功...");  } |

###### 设置栈内存大小

错误原因:java.lang.StackOverflowError

栈溢出产生于递归调用，循环遍历是不会的，但是循环方法里面产生递归调用，也会发生栈溢出。

解决办法:设置线程最大调用深度

-Xss5m设置最大调用深度

|  |
| --- |
| publicclassJvmDemo04{  privatestaticint*count*;  publicstaticvoidcount(){  try{  *count*++;  *count*();  }catch(Throwablee){  System.*out*.println("最大深度:"+*count*);  e.printStackTrace();  }  }  publicstaticvoidmain(String[]args){  *count*();  }  } |

##### Tomcat内存溢出在catalina.sh修改JVM堆内存大小

JAVA\_OPTS="-server-Xms800m-Xmx800m-XX:PermSize=256m-XX:MaxPermSize=512m-XX:MaxNewSize=512m"

#### JVM参数调优总结

主要设置堆内存，不要让gc频繁回收(最大=初始)，减少对老年代的回收。

在JVM启动参数中，可以设置跟内存、垃圾回收相关的一些参数设置，默认情况不做任何设置JVM会工作的很好，但对一些配置很好的Server和具体的应用必须仔细调优才能获得最佳性能。通过设置我们希望达到一些目标：

GC的时间足够的小

GC的次数足够的少

发生FullGC的周期足够的长

前两个目前是相悖的，要想GC时间小必须要一个更小的堆，要保证GC次数足够少，必须保证一个更大的堆，我们只能取其平衡。

（1）针对JVM堆的设置，一般可以通过-Xms-Xmx限定其最小、最大值，为了防止垃圾收集器在最小、最大之间收缩堆而产生额外的时间，我们通常把最大、最小设置为相同的值  
（2）年轻代和年老代将根据默认的比例（1：2）分配堆内存，可以通过调整二者之间的比率NewRadio来调整二者之间的大小，也可以针对回收代，比如年轻代，通过-XX:newSize-XX:MaxNewSize来设置其绝对大小。同样，为了防止年轻代的堆收缩，我们通常会把-XX:newSize-XX:MaxNewSize设置为同样大小

（3）年轻代和年老代设置多大才算合理？这个我问题毫无疑问是没有答案的，否则也就不会有调优。我们观察一下二者大小变化有哪些影响

更大的年轻代必然导致更小的年老代，大的年轻代会延长普通GC的周期，但会增加每次GC的时间；小的年老代会导致更频繁的FullGC

更小的年轻代必然导致更大年老代，小的年轻代会导致普通GC很频繁，但每次的GC时间会更短；大的年老代会减少FullGC的频率

如何选择应该依赖应用程序对象生命周期的分布情况：如果应用存在大量的临时对象，应该选择更大的年轻代；如果存在相对较多的持久对象，年老代应该适当增大。但很多应用都没有这样明显的特性，在抉择时应该根据以下两点：（A）本着FullGC尽量少的原则，让年老代尽量缓存常用对象，JVM的默认比例1：2也是这个道理（B）通过观察应用一段时间，看其他在峰值时年老代会占多少内存，在不影响FullGC的前提下，根据实际情况加大年轻代，比如可以把比例控制在1：1。但应该给年老代至少预留1/3的增长空间

## 垃圾回收机制算法

### 垃圾回收

http://www.cnblogs.com/dolphin0520/p/3783345.html

https://segmentfault.com/a/1190000010463373

一旦引用连接断开，在计算机中就再也找不到了，执行java垃圾回收机制

例：Dog dog = new Dog();//创建对象，dog对象指向一个堆内存中的空间

dog = new Dog();//创建对象，dog变量指向新创建的内存空间，原先创建的废弃

### 垃圾回收机制概述

为什么要有垃圾回收：

c语言需要程序员手动释放资源，java有一个叫gc的线程不定时释放内存。不会立即释放，可以手动调用gc方法释放。此线程关闭，进程会死掉。

程序代码执行完毕，gc也会消失。

Java语言中一个显著的特点就是引入了垃圾回收机制，使c++程序员最头疼的内存管理的问题迎刃而解，它使得Java程序员在编写程序的时候不再需要考虑内存管理。由于有个垃圾回收机制，Java中的对象不再有“作用域”的概念，只有对象的引用才有“作用域”。垃圾回收可以有效的防止内存泄露，有效的使用空闲的内存。

内存溢出值得是本质内存不足，例如程序启动需要10m但是只指定了5m。

ps:内存泄露是指该内存空间使用完毕之后未回收，在不涉及复杂数据结构的一般情况下，Java的内存泄露表现为一个内存对象的生命周期超出了程序需要它的时间长度，我们有时也将其称为“对象游离”。重启就可以了。

#### 垃圾回收简要过程

这里必须点出一个很重要的误区：不可达的对象并不会马上就会被直接回收，而是至少要经过两次标记的过程。

第一次被标记过的对象，会检查该对象是否重写了finalize()方法。如果重写了该方法，则将其放入一个F-Query队列中，否则，直接将对象加入“即将回收”集合。在第二次标记之前，F-Query队列中的所有对象会逐个执行finalize()方法，但是不保证该队列中所有对象的finalize()方法都能被执行，这是因为JVM创建一个低优先级的线程去运行此队列中的方法，很可能在没有遍历完之前，就已经被剥夺了运行的权利。那么运行finalize()方法的意义何在呢？这是对象避免自己被清理的最后手段：如果在执行finalize()方法的过程中，使得此对象重新与GCRoots引用链相连，则会在第二次标记过程中将此对象从F-Query队列中清除，避免在这次回收中被清除，恢复成了一个“正常”的对象。但显然这种好事不能无限的发生，对于曾经执行过一次finalize()的对象来说，之后如果再被标记，则不会再执行finalize()方法，只能等待被清除的命运。

之后，GC将对F-Queue中的对象进行第二次小规模的标记，将队列中重新与GCRoots引用链恢复连接的对象清除出“即将回收”集合。所有此集合中的内容将被回收。

#### 手动GC回收

|  |
| --- |
| publicclassJVMDemo05{  publicstaticvoidmain(String[]args){  JVMDemo05jvmDemo05=newJVMDemo05();  //jvmDemo05=null;  System.*gc*();  }  protectedvoidfinalize()throwsThrowable{  System.*out*.println("gc在回收对象...");  }  } |

#### finalize作用

Java技术使用finalize()方法在垃圾收集器将对象从内存中清除出去前，做必要的清理工作。这个方法是由垃圾收集器在确定这个对象没有被引用时对这个对象调用的。它是在Object类中定义的，因此所有的类都继承了它。子类覆盖finalize()方法以整理系统资源或者执行其他清理工作。finalize()方法是在垃圾收集器删除对象之前对这个对象调用的。

### 垃圾回收机制算法

#### 引用计数法

这种已经不怎么使用

##### 概述

给对象中添加一个引用计数器，每当有一个地方引用它时，计数器值就加1；当引用失效时，计数器值就减1；任何时刻计数器都为0的对象就是不再被使用的，垃圾收集器将回收该对象使用的内存。

##### 优缺点

优点：

引用计数收集器可以很快的执行，交织在程序运行中。对程序需要不被长时间打断的实时环境比较有利。

缺点：

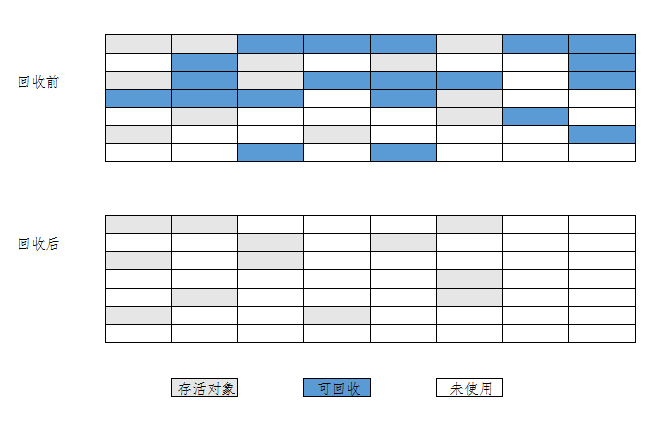
无法检测出循环引用。如父对象有一个对子对象的引用，子对象反过来引用父对象。这样，他们的引用计数永远不可能为0.而且每次加减非常浪费内存。

#### 标记清除算法

标记-清除（Mark-Sweep）算法顾名思义，主要就是两个动作，一个是标记，另一个就是清除。

标记就是根据特定的算法（如：引用计数算法，可达性分析算法等）标出内存中哪些对象可以回收，哪些对象还要继续用。

标记指示回收，那就直接收掉；标记指示对象还能用，那就原地不动留下。



缺点

标记与清除效率低;

清除之后内存会产生大量碎片；

所以碎片这个问题还得处理，怎么处理，看标记-整理算法。

#### 复制算法

非常核心，涉及到新生代，

S0和s1将可用内存按容量分成大小相等的两块，每次只使用其中一块，当这块内存使用完了，就将还存活的对象复制到另一块内存上去，然后把使用过的内存空间一次清理掉。这样使得每次都是对其中一块内存进行回收，内存分配时不用考虑内存碎片等复杂情况，只需要移动堆顶指针，按顺序分配内存即可，实现简单，运行高效。

复制算法的缺点显而易见，可使用的内存降为原来一半。

复制算法用于在新生代垃圾回收

#### 标记-压缩算法

标记清除的升级版本

标记压缩法在标记清除基础之上做了优化，把存活的对象压缩到内存一端,而后进行垃圾清理，不会产生碎片化(java中老年代使用的就是标记压缩法)

#### 分代收集算法

用的特别多。

根据内存中对象的存活周期不同，将内存划分为几块，java的虚拟机中一般把内存划分为新生代和年老代，当新创建对象时一般在新生代中分配内存空间，当新生代垃圾收集器回收几次之后仍然存活的对象会被移动到年老代内存中，当大对象在新生代中无法找到足够的连续内存时也直接在年老代中创建。

对于新生代和老年代来说,新生代回收频率很高,但是每次回收耗时很短,而老年代回收频率较低,但是耗时会相对较长,所以应该尽量减少老年代的GC.

#### 分区算法

#### GC经常回收好不好

不好，GC在回收线程的时候，其他线程全部等待。时间很短，速度快，看不到效果。

#### 为什么老年代使用标记压缩、新生代使用复制算法。

#### 垃圾回收时的停顿现象

垃圾回收的任务是识别和回收垃圾对象进行内存清理，为了让垃圾回收器可以更高效的执行，大部分情况下，会要求系统进如一个停顿的状态。停顿的目的是为了终止所有的应用线程，只有这样的系统才不会有新垃圾的产生。同时停顿保证了系统状态在某一个瞬间的一致性，也有利于更好的标记垃圾对象。因此在垃圾回收时，都会产生应用程序的停顿。

### 垃圾收集器

#### 什么是Java垃圾回收器

Java垃圾回收器是Java虚拟机(JVM)的三个重要模块(另外两个是解释器和多线程机制)之一，为应用程序提供内存的**自动分配(MemoryAllocation)、自动回收(GarbageCollect)**功能，这两个操作都发生在Java堆上(一段内存快)。某一个时点，一个对象如果有一个以上的引用(Rreference)指向它，那么该对象就为活着的(Live)，否则死亡(Dead)，视为垃圾，可被垃圾回收器回收再利用。**垃圾回收操作需要消耗CPU、线程、时间等资源**，所以容易理解的是**垃圾回收操作不是实时的发生**(对象死亡马上释放)，当内存消耗完或者是达到某一个指标(Threshold,使用内存占总内存的比列，比如0.75)时，触发垃圾回收操作。有一个对象死亡的例外，java.lang.Thread类型的对象即使没有引用，只要线程还在运行，就不会被回收。

#### 串行回收器(SerialCollector)

单线程执行回收操作，回收期间暂停所有应用线程的执行，client模式下的默认回收器，通过-XX:+UseSerialGC命令行可选项强制指定。**参数可以设置使用新生代串行和老年代串行回收器**

年轻代的回收算法(MinorCollection)  
把Eden区的存活对象移到To区，To区装不下直接移到年老代，把From区的移到To区，To区装不下直接移到年老代，From区里面年龄很大的升级到年老代。回收结束之后，Eden和From区都为空，此时把From和To的功能互换，From变To，To变From，每一轮回收之前To都是空的。设计的选型为复制。

年老代的回收算法(FullCollection)  
年老代的回收分为三个步骤，标记(Mark)、清除(Sweep)、合并(Compact)。标记阶段把所有存活的对象标记出来，清除阶段释放所有死亡的对象，合并阶段把所有活着的对象合并到年老代的前部分，把空闲的片段都留到后面。设计的选型为合并，减少内存的碎片。

#### 并行回收

##### 并行回收器(ParNew回收器)

并行回收器在串行回收器基础上做了改进，他可以使用多个线程同时进行垃  
圾回收，对于计算能力强的计算机而言，可以有效的缩短垃圾回收所需的尖  
际时间。  
**ParNew回收器是一个工作在新生代的垃圾收集器**，他只是简单的将串行回收  
器多线程快他的回收策略和算法和串行回收器一样。

使用**XX:+UseParNewGC新生代ParNew回收器**，老年代则使用市行回收器  
ParNew回收器工作时的线程数量可以使用**XX:ParaleiGCThreads**参数指  
定，一般最好和计算机的CPU相当，避免过多的栽程影响性能。

##### 并行回收集器(ParallelGC)

老年代ParallelOldGC回收器也是一种多线程的回收器，和新生代的  
ParallelGC回收器一样，也是一种关往吞吐量的回收器，他使用了标记压缩  
算法进行实现。  
-XX:+UseParallelOldGC进行设置  
-XX:+ParallelCThread也可以设置垃圾收集时的线程教量。

#### 并CMS(并发GC)收集器

CMS(ConcurrentMarkSweep)收集器是一种以获取最短回收停顿时间为目标的收集器。CMS收集器是基于“标记-清除”算法实现的，整个收集过程大致分为4个步骤：

①.初始标记(CMSinitialmark)

②.并发标记(CMSconcurrenrmark)

③.重新标记(CMSremark)

④.并发清除(CMSconcurrentsweep)

其中初始标记、重新标记这两个步骤任然需要停顿其他用户线程。初始标记仅仅只是标记出GCROOTS能直接关联到的对象，速度很快，并发标记阶段是进行GCROOTS根搜索算法阶段，会判定对象是否存活。而重新标记阶段则是为了修正并发标记期间，因用户程序继续运行而导致标记产生变动的那一部分对象的标记记录，这个阶段的停顿时间会被初始标记阶段稍长，但比并发标记阶段要短。

由于整个过程中耗时最长的并发标记和并发清除过程中，收集器线程都可以与用户线程一起工作，所以整体来说，CMS收集器的内存回收过程是与用户线程一起并发执行的。

CMS收集器的优点：并发收集、低停顿，但是CMS还远远达不到完美，器主要有三个显著缺点：

CMS收集器对CPU资源非常敏感。在并发阶段，虽然不会导致用户线程停顿，但是会占用CPU资源而导致引用程序变慢，总吞吐量下降。CMS默认启动的回收线程数是：(CPU数量+3)/4。

CMS收集器无法处理浮动垃圾，可能出现“ConcurrentModeFailure“，失败后而导致另一次FullGC的产生。由于CMS并发清理阶段用户线程还在运行，伴随程序的运行自热会有新的垃圾不断产生，这一部分垃圾出现在标记过程之后，CMS无法在本次收集中处理它们，只好留待下一次GC时将其清理掉。这一部分垃圾称为“浮动垃圾”。也是由于在垃圾收集阶段用户线程还需要运行，  
即需要预留足够的内存空间给用户线程使用，因此CMS收集器不能像其他收集器那样等到老年代几乎完全被填满了再进行收集，需要预留一部分内存空间提供并发收集时的程序运作使用。在默认设置下，CMS收集器在老年代使用了68%的空间时就会被激活，也可以通过参数-XX:CMSInitiatingOccupancyFraction的值来提供触发百分比，以降低内存回收次数提高性能。要是CMS运行期间预留的内存无法满足程序其他线程需要，就会出现“ConcurrentModeFailure”失败，这时候虚拟机将启动后备预案：临时启用SerialOld收集器来重新进行老年代的垃圾收集，这样停顿时间就很长了。所以说参数-XX:CMSInitiatingOccupancyFraction设置的过高将会很容易导致“ConcurrentModeFailure”失败，性能反而降低。

最后一个缺点，CMS是基于“标记-清除”算法实现的收集器，使用“标记-清除”算法收集后，会产生大量碎片。空间碎片太多时，将会给对象分配带来很多麻烦，比如说大对象，内存空间找不到连续的空间来分配不得不提前触发一次FullGC。为了解决这个问题，CMS收集器提供了一个-XX:UseCMSCompactAtFullCollection开关参数，用于在FullGC之后增加一个碎片整理过程，还可通过-XX:CMSFullGCBeforeCompaction参数设置执行多少次不压缩的FullGC之后，跟着来一次碎片整理过程。

#### G1回收器

G1回收器(Garbage-First)实在]dk1.7中提出的垃圾回收器，从长期目标来看是为了取  
代CMS回收器，G1回收器拥有独特的垃圾回收策略，G1属于分代垃圾回收器，区分  
新生代和老年代，依然有eden和from/to区,它并不要求整个eden区或者新生代、老  
年代的空间都连续，它使用了分区算法。  
并行性:G1回收期间可多线程同时工作。  
井发性G1拥有与应用程序交替执行能力，部分工作可与应用程序同时执行，在整个  
GC期间不会完全阻塞应用程序。  
分代GC:G1依然是一个分代的收集器，但是它是非两新生代和老年代一杯政的杂尊。  
空间基理，G1在国收过程中，不会微CMS那样在若千tacAy要进行碎片整理。  
G1  
来用了有效复制对象的方式，减少空间碎片。  
利得程，用于分区的原因，G可以贝造取都分区城进行回收，帽小了国收的格想，  
提升了性能。  
使用.XXX:+UseG1GC应用G1收集器，  
Mills指定最大停顿时间  
使用-XX:MaxGCPausel  
设置并行回收的线程数量  
使用-XX:ParallelGCThreads

### Tomcat配置调优测试

#### Jmeter压力测试工具

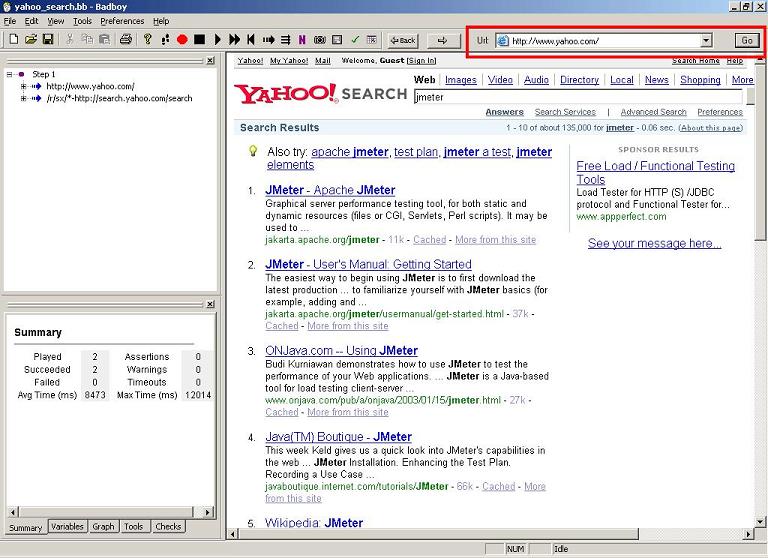
JMeter是一款在国外非常流行和受欢迎的开源性能测试工具，像LoadRunner一样，它也提供了一个利用本地ProxyServer（代理服务器）来录制生成测试脚本的功能，但是这个功能并不好用。所以在本文中介绍一个更为常用的方法——使用[Badboy](http://www.badboy.com.au/)录制生成[JMeter](http://jakarta.apache.org/jmeter/)脚本。

简单的介绍一下[Badboy](http://www.badboy.com.au/)。[Badboy](http://www.badboy.com.au/)是一款不错的Web自动化测试工具，如果你将它用于非商业用途，或者用于商业用途但是安装[Badboy](http://www.badboy.com.au/)的机器数量不超过5台，你是不需要为它支付任何费用的。也许是一种推广策略，[Badboy](http://www.badboy.com.au/)提供了将Web测试脚本直接导出生成[JMeter](http://jakarta.apache.org/jmeter/)脚本的功能，并且这个功能非常好用，也非常简单。你可以跟着下面的试验步骤来迈出你在开源世界的第一步。

1.通过[Badboy](http://www.badboy.com.au/)的官方网站下载[Badboy](http://www.badboy.com.au/)的最新版本；

2.安装[Badboy](http://www.badboy.com.au/)。安装过程同一般的Windows应用程序没有什么区别，安装完成后你可以在桌面和Windows开始菜单中看到相应的快捷方式——如果找不到，可以找一下[Badboy](http://www.badboy.com.au/)安装目录下的[Badboy](http://www.badboy.com.au/).exe文件，直接双击启动[Badboy](http://www.badboy.com.au/)；

3.启动[Badboy](http://www.badboy.com.au/)，你可以看到下面的界面。



**在地址栏（图中红色方框标注的部分）中输入你需要录制的Web应用的URL——这里我们以**[**http://www.yahoo.com**](http://www.yahoo.com/)**为例，并点击GO按钮开始录制。如果你用过LoadRunner之类的商业工具，对于这个操作一定不会陌生吧^\_^**

4.开始录制后，你可以直接在[**Badboy**](http://www.badboy.com.au/)内嵌的浏览器（主界面的右侧）中对被测应用进行操作，所有的操作都会被记录在主界面左侧的编辑窗口中——在这个试验中，我们在Yahoo的搜索引擎中输入[**JMeter**](http://jakarta.apache.org/jmeter/)进行搜索。不过你将看到，录制下来的脚本并不是一行行的代码，而是一个个Web对象——这就有点像LoadRunner的VuGen中的TreeView视图；

5.录制完成后，点击工具栏中的“停止录制”按钮，完成脚本的录制；

6.选择“File->Exportto[**JMeter**](http://jakarta.apache.org/jmeter/)”菜单，填写文件名“login\_mantis.jmx”，将录制好脚本导出为[**JMeter**](http://jakarta.apache.org/jmeter/)脚本格式。也可以选择“File->Save”菜单保存为[**Badboy**](http://www.badboy.com.au/)脚本；

7.启动[JMeter](http://jakarta.apache.org/jmeter/)并打开刚刚生成的测试脚本。

也许你已经急不可待的准备开始尝试着用[JMeter](http://jakarta.apache.org/jmeter/)处理你手头的工作了^\_^在下面的几节，我将继续为大家介绍如何在[JMeter](http://jakarta.apache.org/jmeter/)中完成一个测试场景的设置和[JMeter](http://jakarta.apache.org/jmeter/)测试结果分析入门，以及如何参数化[JMeter](http://jakarta.apache.org/jmeter/)脚本。

当然，如果你的动手能力很强，几分钟你就可以熟悉这些内容。不过还是请允许我一点点由浅入深的来帮大家完成“[JMeter](http://jakarta.apache.org/jmeter/)从入门到精通”的过程。我相信在这个过程中你将会了解到更多有关性能测试的知识和经验，甚至包括一些LoadRunner等商业测试工具所无法提供给你的经验。

#### 测试串行吞吐量

访问和返回，在一秒钟固定执行多少次。

|  |
| --- |
| -XX:+PrintGCDetails-Xmx32M-Xms32M  -XX:+HeapDumpOnOutOfMemoryError  -XX:+UseSerialGC  -XX:PermSize=32M |
| GC回收6次吞吐量301 |

#### 扩大堆的内存

|  |
| --- |
| -XX:+PrintGCDetails-Xmx512M–Xms32M  -XX:+HeapDumpOnOutOfMemoryError  -XX:+UseSerialGC  -XX:PermSize=32M |
| GC回收2次吞吐量349 |
| 结论最大内存越大，吞吐量越高。 |

#### 调整初始堆

|  |
| --- |
| -XX:+PrintGCDetails-Xmx512M–Xms256M  -XX:+HeapDumpOnOutOfMemoryError  -XX:+UseSerialGC  -XX:PermSize=32M |
| GC回收0次吞吐量419 |

#### 并行回收（UseParNewGC）

|  |
| --- |
| -XX:+PrintGCDetails-Xmx512M–Xms256M  -XX:+HeapDumpOnOutOfMemoryError  -XX:+UseParNewGC  -XX:PermSize=32M |
| GC回收0次吞吐量532 |

#### 并行合并回收（UseParallelGC）

|  |
| --- |
| -XX:+PrintGCDetails-Xmx512M-Xms256M  -XX:+HeapDumpOnOutOfMemoryError  -XX:+UseParallelGC  -XX:+UseParallelOldGC  -XX:ParallelGCThreads=8  -XX:PermSize=32M |
|  |

测试结果2100209

总结：

初始内存和最大内存越大，吞吐量越大。

并行回收比串行回收吞吐量要多，多线程吞吐更多。

垃圾回收机制次数越少，性能越高。

### 什么是栈溢出

如果想要栈溢出，可以递归调用方法，这样随着栈深度的增加，JVM维持着一条长长的方法调用轨迹，

## 双亲委派模型

### 类加载器

#### 加载器的开放性

类加载器（ClassLoader）是Java语言的一个创新，也是java流行的一个重要原因

在类加载的第一阶段“加载”过程中，需要通过一个类的全限定名来获取定义此类的二进制字节流，完成这个动作的代码块就是类加载器，这一动作是放在Java虚拟机外部去实现的，以便让应用程序自己决定如何获取所需要的类。

虚拟机规范并没有指明二进制字节流要从一个Class文件获取，或者说根本没有指明从哪里获取，怎样获取。这种开放使得java在很多领域得到充分运用。例如：

* 从zip包中读取，这很常见，称为jar，war，ear格式的基础。
* 从网络中获取，最典型的应用就是Applet。
* 运行时计算生成，最典型的是动态代理技术，在java.lang.reflect.proxy中，就是用了ProxyGenerator.generateProxyClass来为特定接口生成形式为“$Proxy”的代理类的二进制字节流。
* 有其他文件生成，最典型的jsp应用，由jsp文件生成对应的class类。

#### 类加载器与类的唯一性

类加载器虽然用于实现类的加载动作，但是对于任意一个类，都需要由加载它的类加载器和找个类本身共同确立在Java虚拟机中的唯一性，通俗的说，JVM中两个类是否相等，首先就必须是同一个类加载器加载的，否则，即使这两个类来源于同一个Class文件，被同一个虚拟机加载，只要类加载器不同，那么这两个类必定是不相等的，这里的相等，包括代表类的Class对象的equals方法，isAssignableFrom() 方法、isInstance() 方法的返回结果，也包括使用 instanceof 关键字做对象所属关系判定等情况。

下代码说明了不同的类加载器对 instanceof 关键字运算的结果的影响。

package com.jvm.classloading;

import java.io.IOException;

import java.io.InputStream;

/\*\*

\* 类加载器在类相等判断中的影响

\*

\* instanceof关键字

\*

\*/

public class ClassLoaderTest {

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

// 自定义类加载器

ClassLoader myLoader = new ClassLoader() {

@Override

public Class<?> loadClass(String name) throws ClassNotFoundException {

try {

String fileName = name.substring(name.lastIndexOf(".") + 1) + ".class";

InputStream is = getClass().getResourceAsStream(fileName);

if (is == null) {

return super.loadClass(fileName);

}

byte[] b = new byte[is.available()];

is.read(b);

return defineClass(name, b, 0, b.length);

} catch (IOException e) {

throw new ClassNotFoundException();

}

}

};

// 使用 ClassLoaderTest 的类加载器加载本类

Object obj1 = ClassLoaderTest.class.getClassLoader().loadClass("com.jvm.classloading.ClassLoaderTest").newInstance();

System.out.println(obj1.getClass());

System.out.println(obj1 instanceof com.jvm.classloading.ClassLoaderTest);

// 使用自定义类加载器加载本类

Object obj2 = myLoader.loadClass("com.jvm.classloading.ClassLoaderTest").newInstance();

System.out.println(obj2.getClass());

System.out.println(obj2 instanceof com.jvm.classloading.ClassLoaderTest);

}

}

输出结果：

class com.jvm.classloading.ClassLoaderTest true

class com.jvm.classloading.ClassLoaderTest false

myLoader 是自定义的类加载器，可以用来加载与自己在同一路径下的 Class 文件。main 函数的第一部分使用系统加载主类 ClassLoaderTest 的类加载器加载 ClassLoaderTest，输出显示，obj1 的所属类型检查正确，这是虚拟机中有 2 个 ClassLoaderTest 类，一个是主类，另一个是 main() 方法中加载的类，由于这两个类使用同一个类加载器加载并且来源于同一个 Class 文件，因此这两个类是完全相同的。

第二部分使用自定义的类加载器加载 ClassLoaderTest，class com.jvm.classloading.ClassLoaderTest 显示，obj2 确实是类 com.jvm.classloading.ClassLoaderTest 实例化出来的对象，但是第二句输出 false。此时虚拟机中有 3 个 ClassLoaderTest 类，由于第 3 个类的类加载器与前面 2 个类加载器不同，虽然来源于同一个 Class 文件，但它是一个独立的类，所属类型检查是返回结果自然是 false。

## 双亲委派模型

### 类加载器种类

从 Java 虚拟机的角度来说，只存在两种不同的类加载器：一种是启动类加载器（Bootstrap ClassLoader），这个类加载器使用 C++ 语言实现（HotSpot 虚拟机中），是虚拟机自身的一部分；另一种就是所有其他的类加载器，这些类加载器都有 Java 语言实现，独立于虚拟机外部，并且全部继承自 java.lang.ClassLoader。

从开发者的角度，类加载器可以细分为：

#### 启动（Bootstrap）类加载器：

负责将 Java\_Home/lib 下面的类库加载到内存中（比如 rt.jar）。由于引导类加载器涉及到虚拟机本地实现细节，开发者无法直接获取到启动类加载器的引用，所以不允许直接通过引用进行操作。

* 标准扩展（Extension）类加载器：是由 Sun 的 **ExtClassLoader（sun.misc.Launcher$ExtClassLoader）** 实现的。它负责将 Java\_Home /lib/ext 或者由系统变量 java.ext.dir 指定位置中的类库加载到内存中。开发者可以直接使用标准扩展类加载器。
* 应用程序（Application）类加载器：是由 Sun 的 **AppClassLoader（sun.misc.Launcher$AppClassLoader）** 实现的。它负责将系统类路径（CLASSPATH）中指定的类库加载到内存中。开发者可以直接使用系统类加载器。由于这个类加载器是 ClassLoader 中的 getSystemClassLoader() 方法的返回值，因此一般称为系统（System）加载器。

除此之外，还有自定义的类加载器，它们之间的层次关系被称为类加载器的 **双亲委派模型**。该模型要求除了顶层的启动类加载器外，其余的类加载器都应该有自己的父类加载器，而这种父子关系一般通过组合（Composition）关系来实现，而不是通过继承（Inheritance）。

**双亲委派模型**

**双亲委派模型过程**

某个特定的类加载器在接到加载类的请求时，首先将加载任务委托给父类加载器，依次递归，如果父类加载器可以完成类加载任务，就成功返回；只有父类加载器无法完成此加载任务时，才自己去加载。

使用双亲委派模型的好处在于 **Java 类随着它的类加载器一起具备了一种带有优先级的层次关系**。例如类 java.lang.Object ，它存在在 **rt.jar** 中，无论哪一个类加载器要加载这个类，最终都是委派给处于模型最顶端的 **Bootstrap ClassLoader** 进行加载，因此 Object 类在程序的各种类加载器环境中都是同一个类。相反，如果没有双亲委派模型而是由各个类加载器自行加载的话，如果用户编写了一个 java.lang.Object 的同名类并放在 ClassPath 中，那系统中将会出现多个不同的 Object 类，程序将混乱。因此，如果开发者尝试编写一个与 **rt.jar** 类库中重名的 Java 类，可以正常编译，但是永远无法被加载运行。

**双亲委派模型的系统实现**

在 java.lang.ClassLoader 的 loadClass() 方法中，先检查是否已经被加载过，若没有加载则调用父类加载器的 loadClass() 方法，若父加载器为空则默认使用启动类加载器作为父加载器。如果父加载失败，则抛出 ClassNotFoundException 异常后，再调用自己的 findClass() 方法进行加载。

protected synchronized Class < ?>loadClass(String name, boolean resolve) throws ClassNotFoundException {

//check the class has been loaded or not

Class c = findLoadedClass(name);

if (c == null) {

try {

if (parent != null) {

c = parent.loadClass(name, false);

} else {

c = findBootstrapClassOrNull(name);

}

} catch(ClassNotFoundException e) {

//if throws the exception ,the father can not complete the load

}

if (c == null) {

c = findClass(name);

}

}

if (resolve) {

resolveClass(c);

}

return c;

}

注意，双亲委派模型是 Java 设计者推荐给开发者的类加载器的实现方式，并不是强制规定的。大多数的类加载器都遵循这个模型，但是 JDK 中也有较大规模破坏双亲模型的情况，例如线程上下文类加载器（Thread Context ClassLoader）的出现

# 问题整理

从最早的ASP，VB开始，ASP的代码很垃圾，很混乱。

PHP：好东西，PHP大部分的项目都是中小型项目，而且以小型项目居多，PHP必须要MySql支持。

没有大公司支持。

.NET：平台问题。中型项目开发。

Java：注重的是理论，注重的是分层。多个公司支持。JAVA做中大型项目的非常多。

不是完全由技术决定的，更多的是综合方面的影响决定的。

每天坚持学习，可以让自己更加有信心。

每天看书有助于睡眠。

成功的人，都是在一直不断努力工作着，努力学习着。

如果采用的是自学的方式，最好的学习手段就是进行熟练，先从代码开始做，之后再研究理论，实际上如果EJB已经掌握了，则基本的卢纶根本就不用学了。‘

软性能力:

做人，素质，模仿名人。

不断准备。

MVC设计模式很重要。

问题处理

Java Web开发实战1200例 第1卷

HtmlConverter打不开 J 7页

XML

Xpath

@选属性值

name（）函数返回元素的名称，start-with()函数在该函数的第一个参数字符串是以第二个参数字符开始的情况返回true,contains函数当其第一个字符串参数包含有第二个字符串参数时返回true。

多个路径可以用分隔符|合并在一起。

## 使用方法

idea开发JSP&servlet的问题？

<init-param>

<url-pattern>

之类的东西 ，在相应的 Servlet 文件里面的

@WebServlet

里面添加就可以了，类似这样：

@WebServlet(value = "/Servlet",name = "Servlet")

也可以这样

urlPatterns

@WebServlet(name = "ServletDemo4",urlPatterns={"/demo4"} )

@WebServlet(value = "/LifeCycleServlet",

name = "LifeCycleServlet",

initParams = {@WebInitParam(name = "startPoint", value = "1600"),

@WebInitParam(name = "Point", value = "1600")})

也可以参考一下：Servlet3.0中Servlet的使用http://elim.iteye.com/blog/2013691

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

其实刚刚也测试了一下，即使没加 @WebServlet 在 web.xml配置文件中填写好信息后，虽然Idea无法直接点右上角的浏览器直接访问，但是通过输入 http://localhost:8080/XXX.jsp 或者 http://localhost:8080/XXX 其中XXX是 servlet的文件名 也是可以访问的