## 基础篇

### 数据类型

Java程序的运行机制：

Java是两种语言的结合（编译命令：javac.exe 解释命令：java.exe）

当执行javac命令时，会启动java的编译器程序。对指定扩展名的.java文件进行编译。 生成了jvm可以识别的字节码文件，也就是class文件，也就是java的运行程序，而java命令，负责运行，执行java命令后，会启动JVM，加载运行时所需的类库，并执行编译好的class字节码文件

一个程序要执行，必须要有一个执行的起始点，这个起始点，就是java中的main函数，main函数是java中程序的入口

关键字：不同的关键字可以用来便是一种数据类型，或者表示程序的结构等

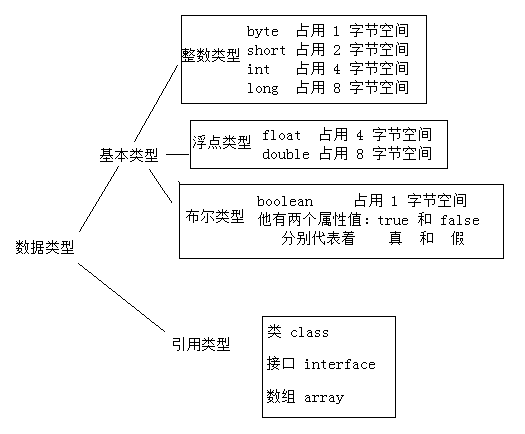
标识符：Java中的包，类，方法，参数和变量的名称总称为标识符

保留字：保留关键字是指Java中并没有使用goto，以后有可能使用

例如：goto就是Java的保留关键字

变量：变量是指在程序的运行过程中随时可以发生变化的量，是程序中数据的临时存放场所

常量：常量指程序里持续不变的值，在整个程序运行过程中它是不可改变的



### 运算符

基本的算数运算符： **+ - \* / % = ++ -- += -= \*= /= ；**



比较运算符： > >= < <= == !=

比较运算符只会产生两种结果：true 和 false （boolean类型）

逻辑运算符： & | ^ ! && ||

&： 两边都为true，结果是true。否则为false。两边只要有一个false。结果必然是false。

| ： 两边都为false，结果是false。否则为true。两边只要有一个true。结果必然是true。

^： 异或。两边相同为false。两边不同为true。

！： 非：非真为假，非假为真

&&： 两侧都为true，结果为true，否则为false，如果左侧为false，右侧将不参与运算

| |： 两侧有一侧true，结果为true，否则为false，如果左侧为true，右侧将不参与运算

位运算，他们的符号分别为：

按位与 & 按位或 | 按位异或 ^ 按位取反 ~ 左移 << 右移 >> 无符号右移 >>>

& 按位与运算 ，运算原则：相同二进制数位上数值为1，结果为1，否则结果为0；

3 & 5 先把 3和5以二进制表示，再进行按位于运算：

十进制 二进制 结果(十进制)

3 00000000\_00000000\_00000000\_00000011

5 00000000\_00000000\_00000000\_00000101

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

& : 00000000\_00000000\_00000000\_00000001 1

所以3 & 5等于 1.

| 按位或运算，运算原则：相同二进制数位上数值为0，结果为0，否则结果为1；

3 | 5 先把 3和5以二进制表示，再进行按位于运算：

十进制 二进制 结果(十进制)

3 00000000\_00000000\_00000000\_00000011

5 00000000\_00000000\_00000000\_00000101

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

| : 00000000\_00000000\_00000000\_00000111 7

所以3 | 5等于 7.

2的0次方 + 2 的1次方 + 2的2次方 = 1 + 2 + 4 = 7

^ 按位异或，运算原则：参与运算的两个数相应位相同则结果为0，异号则为1；

十进制 二进制 结果(十进制)

3 00000000\_00000000\_00000000\_00000011

5 00000000\_00000000\_00000000\_00000101

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

^ : 00000000\_00000000\_00000000\_00000110 6

所以3 ^ 5等于 6.

~ 按位取反，运算原则：二进制数位上0变1，1变0；

十进制 二进制 结果(十进制)

5 00000000\_00000000\_00000000\_00000101

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

~ : 11111111\_11111111\_11111111\_1111010 -6

所以~5等于 -6.

左移操作“<<”：将运算数的二进制码整体左移指定位数，左移之后的空使用“0”来补充，移出去的二进制数忽略不计。

9 << 2 :

十进制 二进制 结果(十进制)

9 00000000\_00000000\_00000000\_00001001

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

00000000\_00000000\_00000000\_0000100100 36

所以9<<2等于 36.

规律：<<：相当于给原数乘以2的倍数

右移操作“>>”：将运算数的二进制码整体右移指定位数，右移之后的空使用“符号位”来补充。移出去的二进制数忽略不计。若是正数使用“0”补充；若是负数使用“1”补充；

9 >>2 :

十进制 二进制 结果(十进制)

9 00000000\_00000000\_00000000\_00001001

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

0000000000\_00000000\_00000000\_00001001 2

所以9>>2等于 2.

规律：>>：相当于给原数除以2的倍数

-6 >> 2:

十进制 二进制 结果(十进制)

-6 11111111\_11111111\_11111111\_1111010

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

1111111111\_11111111\_11111111\_1111010

所以-6>>2等于 -2.

// 无符号右移">>>"：将运算数的二进制码整体右移指定位数，右移之后的空使用“0”来补充

三元运算符：

表达式1 ？ 表达式2 ： 表达式3

表达式1为逻辑表达式，返回一个boolean类型的值，如果结果为true返回第二个表达式的值，为false则返回第三个表达式的值

### 流程控制语句

**if else判断语句**：

if(条件表达式){

执行语句；

}else if (条件表达式){

执行语句；

}else{

执行语句；

}

**switch分支语句**：

switch(表达式){

case 取值1:

执行语句；

break；

case 取值2:

执行k；

break；

……

default:

执行语句；

}

**while循环**：

while（ 逻辑表达式 ）{

循环体

}

**do-while循环：**

do{

循环体

}while（ 逻辑表达式 ）

// do-while 循环有一个特点，那就是先运行程序，在判断表达式是否符合要求

// 这也就意味着无论逻辑表达式的结果是 true 还是false，循环体无论如何都会执行一次

**for循环：**

for（ 赋值表达式 ； 逻辑表达式 ； 算数表达式 ）{

循环体

}

流程控制关键字：

break： 终止该循环

continue： 跳过本次循环，执行下一次循环

return： 特殊的控制语句，结束当前方法的运行

小技巧：

在创建循环的时候可以给循环起名，然后通过流程控制关键字来操作对应名字的循环体

|  |  |
| --- | --- |
| outer:for (int x=0; x<3 ;x++ ){  inner:for (int y=0; y<4 ;y++ ){  System.out.println("x="+x);  break outer;  }  } | outer:for (int x=0; x<3 ;x++ ){  inner:for (int y=0; y<4 ;y++ ){  System.out.println("x="+x);  continue outer;  }  } |

### 进阶篇

数组：

数组是同一种类型数据的集合；即能够存放多个相同类型的数据的容器。

数组的定义格式为：

元素类型[] 数组名 = new 元素类型[元素个数或数组长度];

元素类型[] 数组名 = { 元素 ， 元素 … }；

数组涉及到的异常有：

NullPointerException：空指针异常

当数组没有没有初始化的时候就调用数组元素进行操作就会发生空指针异常

ArrayIndexOutOfBoundsException：数组下标越界异常

操作的元素下标超过或者等于数组长度时（或者小于0）就会发生数组下标越界异常

使用foreach循环遍历数组：

int []arr = { 1 ， 2 ， 3 ，4 ， 5 };

for( 数组元素类型 变量名 ：数组名 ){

System.out.println( 变量名 );

}

多维数组

多维数组中可以理解为在一维数组中再次存储了一个一维数组

元素类型[][] 数组名 = { { } ， { } ， { } }；

元素类型[][] 数组名 = new 元素类型[元素个数或数组长度] [元素个数或数组长度];

### 方法（函数）

方法：也被称为函数，它是定义在类中的具有特定功能的一段独立小程序，并能被多次使用

方法的作用：方法是被调用后才会执行的，他会返回结果给调用者

方法（函数）的定义格式：

修饰符 返回值类型 方法名 （ 参数列表 ）{

代码体

return；

}

方法的重载：

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不同即可，这就叫做方法的重载，**重载和返回值类型无关**

### DeBug调试

F5：跳转到该函数内

F6：下一行代码

F7：从F5内跳出

F8：跳转到下一个断点的位置，没有则结束调试

## 进阶篇

### 面向对象

面向对象的三个特点： 封装 继承 多态

OOA - Object-Oriented Analysis (面向对象分析)

OOP - Object-Oriented Programming (面向对象编程)

OOD - Object-Oriented Design (面向对象设计)

OOT - Object-Oriented Testing (面向对象测试)

OOM - Object-Oriented Method (面向对象方法)

OOSM - Object-Oriented Software Maintenance （面向对象软件维护)

类：他是对某一种共性特诊的封装和描述，他就是一个模板

对象：类的实例化就是对象，基于类的模板创建出来的就叫对象，**对象通过new关键字创建出来存放到堆内存中**，可以在类中直接使用

创建对象的两种表现形式:

动态声明: String s = new String ( “ 字符串 ” )；

静态声明： String s = “ 字符串 ”;

通过new关键字创建出来的对象都属于动态声明

值的传递：

1.基本类型传递：将某个空间中具体的数据复制给另外一个空间，修改另外一个空间的数据并不会对原始空间的数据造成影响

2.引用类型传递：将一个内存地址传递给其他的引用变量，那么就会导致多个引用变量指向同一个内存区域，这样其中任何的引用都可以去操作这块共享的内存区域，那么将导致一旦修改，其他的引用再去操作一定是被改变的后数据

构造方法：

构造方法是一种特殊的方法，在创建对象的时候他会自动调用，也就是说我们创建对象实际上是在调用该类中的构造方法，当一个类中没有构造方法的时候，系统会自动生成一个隐藏的空参构造

构造方法定义格式：

public 类名（ 参数列表 ）{

方法体

}

构造方法的方法名必须与类名一致，且构造方法内不能添加return

创建对象时的构造方法是内存操作符(new)来调用的

同一个构造方法不能同时调用多个this和super方法

this关键字：

this关键字一共有两个作用：

1. 调用构造函数
2. 区分成员变量和局部变量

**创建对象时的执行过程**

1. 父类静态初始化代码块
2. 子类静态初始化代码块
3. 父类初始化代码块
4. 父类构造方法
5. 子类初始化代码块
6. 子类构造方法

### 封装

封装：

尽可能将不需要外界知道的东西全部隐藏在其中，仅仅对外界暴露需要操作的部分

方法就是一个封装体，不需要让外界知道里面是什么，只需要知道调用方法有什么用就够了

修饰符：

public：公开的，被public修饰的类或者成员，它们的访问权限最大，在任何地方都可以访问被public修饰的内容。

private：私有的，被private关键字修饰的成员只能在这个类中被访问，类以外的其他地方永远无法正常访问。

protected：受保护的，同一个包下的类 和 其他包的子类可以调用

default：默认的，仅允许同一个包下的类可以调用，其他包不可以继承

### 继承

什么是继承：

继承，也称泛化，继承性是子类自动共享父类属性和方法的机制，继承的关键字为extends

一个父类可以拥有多个子类，但一个子类只能拥有一个父类

在java中类与类之间，仅支持单继承，也就是说一个类只有一个父亲，Java中不支持多继承，如果一个类有多个父类，会导致子类调用父类的时候出现不稳定因素（ 不稳定因素：如果可以多继承，两个父类中有同名方法，那么子类应该如何调用 ）

继承的体现：

class A{

代码

}

class B extends A{

代码

}

// B继承了A，这里A就是父类（基类），B就是子类（派生类）

继承父类或实现某个接口时进行函数的复写：

如果父类有子类相同的方法，但是父类中的方法并不能满足子类，这个时候就需要复写父类中的方法，在自己的类中进行重新编辑，被称为复写

方法复写的要求：

1. 相同的方法名
2. 相同的参数列表
3. 相同的返回值类型
4. 子类复写的方法的访问权限必须大于等于父类访问权限

同一个类中实现类似的功能可以使用相同的方法名，这一机制被称为重载

方法重载的要求:

1. 方法名相同
2. 参数列表的类型、数量、顺序不同

super关键字：用来调用父类构造方法

final关键字：被final关键字修饰的：

1. 类不可以被继承
2. 方法不可以复写
3. 变量不可以修改

### 抽象

被abstract修饰符修饰的类就是抽象类

抽象类也是一个类，我们发现在描述一个方法的时候没有那么清晰，这是我们可以将这个方法修改为抽象方法（使用abstract修饰）

抽象类注意事项

1. 抽象类中的变量可以正常定义且继承

2. 抽象类不能实例化

3. 抽象类中可以书写一般方法并且带有方法体，但是抽象类中的抽象方法不可以有方法体

4. 抽象方法不能被 private static 和 final 修饰（因为抽象方法要求子类必须复写，有了这些修饰符会影响复写规则）

5. 一般类中不能书写抽象方法

6. 当一个类中有一个或多个抽象方法的时候，那么这个类就要改为抽象类

7. 被继承的抽象类中的抽象方法必须被子类复写

### 接口

接口是一种特殊的抽象类（关键字interface）

接口中只包含常量和抽象方法，而没有变量和方法的实现

接口的定义格式：

interface Demo{

}

一个类想要实现一个接口的话需要用到implements关键字

接口和接口之间可以多继承（extends），一个类也可以实现多个接口（implements）

接口的注意事项：

1. 接口没有构造方法，所以不能实例化
2. 接口的修饰符只有interface
3. 接口中所有方法都是抽象方法
4. 接口中的方法默认修饰符为：public abstract且不能更改
5. 接口中常量默认的修饰符为：public static final且不能更改

JDK7之后，接口中可以声明一个非抽象方法,他是由public default 修饰。这个方法必须有方法体。将这个方法称为默认方法

### 多态

**什么是多态**：

一个事物呈现出来的不同的形态；将某个子类的个体，使用了父类或接口类型进行展示。就叫做多态

**Java中的多态**：

相同类型的不同对象,调用相同的方法,产生的结果不同,这种现象就叫做多态

**多态实现的三个必要条件：**

1. 要有继承或实现
2. 要有复写（重写）
3. 父类引用指向子类对象

**多态中的类型转换：**

转型：转型是以子类为基准

**向上转型**：

父类 变量名 = new 子类（）；

这里将子类的对象存放到了父类类型中，类型扩大，属于向上转型

使用向上转型后得到的实例化对象，只能使用父类中被子类继承复写的方法

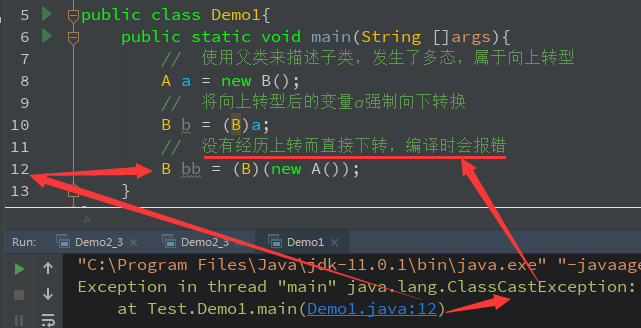
**向下转型**：

子类 变量名 = new 父类（）；

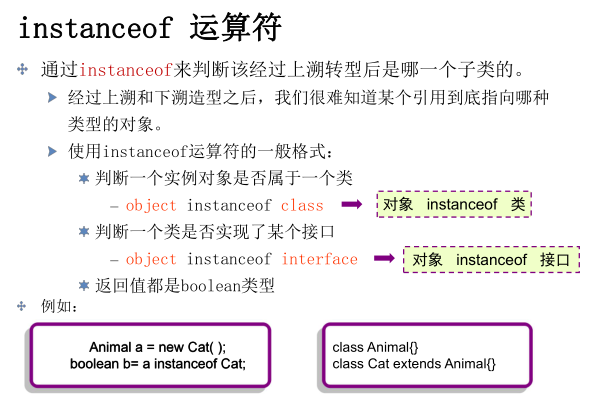
这里将父类的对象存放到了子类类型中，类型缩小，属于向下转型

向下转型的时候需要进行强制类型转换，才可以将父类对象转换为子类类型

只有曾将向上转换过的才可以向下转换，对象不允许不经过向上转型而直接向下转型。



instanceof运算符：



### 正则表达式

需要补习

## 最终篇

### 集合

#### 集合概述

集合是什么：

集合和数组有些相似，是一个容器，用来存放且只能存放对象，其中每个对象作为元素出现（集合类and接口位于java.utiu包下）

集合的意义：

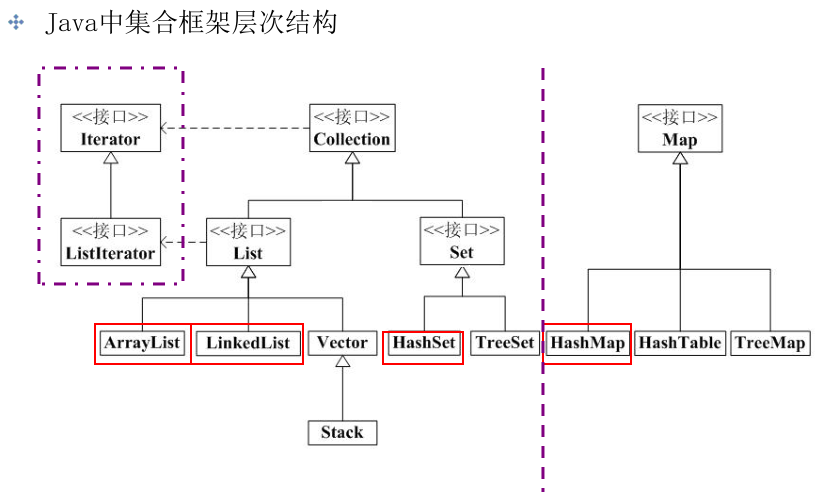
如果想要存储不定长的数据操作，可以使用集合，因为集合支持自动扩容

集合和数组的区别：

数组也是容器，他是定长的，访问较快，但是数组不会自动扩容（我规定数组长度为10，那么在10之后再次存储就会抛出数组下标越界异常），集合支持自动扩容

数组可以包含基本数据类型和引用类型的对象，而集合中能包含且只能包含引用类型对象

集合的 继承/实现 关系：



#### Collection接口

一组成为元素的对象，一个Collection中可以存放不同类型的数据

是List接口和Set接口的父类

#### List接口

继承了Collection接口，用来存储有序有重复的对象，List中的元素都对应一个整数型的序号，记载其在容器中的位置，可以根据序号春去容器中的元素

List接口下的实现类有：

**ArrayList：** 线性顺序存储，是一种线性表，他和数组类似，只不过ArrayList的大小是可以动态改变的

**LinkedList：**数据结构中链表的java表现，他是通过头和尾对的方式对集合中的元素进行增删改查，如果仅仅是对下标进行操作的话，那么使用ArrayList就可以了，使用它的时候，基本都是**使用它特有的方法实现的功能**：

队列结构：

例如：火车过山洞，存储数据的时候，最先到容器的数据是最先被取出，先进先出，后进后出

堆栈结构：

例如：手枪弹夹，最先存储的数据，最后才被取出，先进后出。后进先出

**Vector：**他是JDK1.0时期的集合，在JDK1.2的时候被称为List接口的一个实现类，在JDK1.2版本后被ArrayList代替，只要在程序中看到Vector集合，都可以把它看错成ArrayList使用

#### Set接口

继承了Collection接口，用来存储无序无重复的对象，

Set接口下的实现类有：

**HashSet：** 其内部对象散列存取，即采用哈希技术

**TreeSet：** 存入的顺序和存储的顺序不懂，但是存储是按照排序存储的

#### Map接口

Map内存储的是 key/value，键/值对 这样以成对出现的对象组（可以把一组对象当成一个元素），通过key来查询value对象

Map是不同于Collection的另外一种集合接口

Map中key值是唯一的（不能重复），而key对象是与value对象关联在一起的

Map接口下的实现类有：

**HashMap：**

HashMap中的key-value是按照哈希算法存储的，基于哈希表实现，可以通过调优初始容量和负载因子，优化HashMap空间的使用，性能方面高于TreeMap

**TreeMap：**

TreeMap中的key-value是按照key的排序存储的，基于树实现，没有调优选项，因此该树总处于平衡状态，性能方面低于HashMap。

TableMap：

Map集合的两个特殊的遍历：

// 1 使用entrySet方法，将每一个key-value封装为对象放在set集合中，通过遍历集合取出每个Entry对象，获取key和value

HashMap<String,String> hm = **new** HashMap<String,String>();

hm.put("01","曾"); hm.put("02","崔"); hm.put("03","房");

Set<Entry<String,String>> entrySet = hm.entrySet();

**for**( Entry en : entrySet ) {

System.***out***.print( en.getKey() + "=" + en.getValue() + " " );

}

// 2 使用values方法获取到所有的value值，封装到Collection集合中，遍历Collection集合即可

HashMap<String,String> hm = **new** HashMap<String,String>();

hm.put("01","曾"); hm.put("02","崔"); hm.put("03","房");

Collection<String> values = hm.values();

**for**( String s : values )

System.***out***.print( s + " " );

#### Iterator接口

Iterator一般被称作为迭代器，用来方便的实现对容器内的元素进行遍历操作

所有实现了Collection接口的集合都有一个Iterator（）方法，调用该方法后会返回一个Iterator迭代器

迭代器的遍历规则为：

1、判断集合中有没有元素。

2、如果有元素就取出当前这个元素。

// 演示使用迭代器遍历集合( List与Set同理 )  
ArrayList al = **new** ArrayList();

al.add("元素1"); al.add("元素2");al.add("元素3");

**for**( Iterator it = al.iterator(); it.hasNext() ; ){

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

}--------------------------------------------------------------------------------------------------------------------

// 演示使用Iterator遍历Map集合  
HashMap hs = **new** HashMap();

hs.put("key值1","value值"); hs.put("key值2","value值"); hs.put("key值3","value值");

Set set = hs.keySet();

**for**( Iterator it = set.iterator() ; it.hasNext() ; ){

Object key = it.next();

Object value = hs.get( key );

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

}

#### Enumeration接口

Enumeration旧版本迭代器，功能和Iterator重复，已经被Iterator迭代器取代，但是Iterator迭代器是从JDK1.2版本才开始诞生的，对于JDK1.2版本之前的集合应该如何遍历？，这个时候就需要使用这个古老的迭代器（Iterator迭代器也可以遍历）

Vector vt = **new** Vector();

vt.add("我"); vt.add("学习"); vt.add("Java"); vt.add("！！！");

**for**( Enumeration en = vt.elements(); en.hasMoreElements() ; ){

System.out.print(en.nextElement());

}

集合类总结：

1. Set集合不允许元素重复存放，而List就可以
2. Set集合内存放元素是无序的，而List存放元素有一定的顺序
3. Map的应用主要在利用 key-value 对元素进行快速查询
4. ArrayList和LinkedList的区别在于随机查询性能上ArrayList比较好但是LinkedList的中间元素的插入与删除性能比较好。

### 文件与流

#### File类

File类：主要是针对文件或是目录路径名的管理，File类在java.io包中，他主要对文件进行：

* 创建
* 删除
* 判断
* 获取
* 改名
* 列举

使用 【判断 + 获取 + 列举 + 递归】 实现的多级文件遍历

public class Date\_ {  
 static StringBuilder s = new StringBuilder("|");  
 public static void main(String []args){  
 // 使用递归方法遍历指定目录下的所有文件及文件夹并将相关信息打印在控制台上  
 String fileName = "D:/";  
 File file = new File(fileName);

// 通过file.listFiles方法获取到File类型的数组  
 File[] files = file.listFiles();  
 System.out.println("【" + file.getName() + "】" + "属于文件夹，即将开始遍历该文件夹^\_^");  
 long l1 = System.currentTimeMillis();

// 将数组传入递归方法内  
 System.out.println(printFile( files ));  
 long l2 = System.currentTimeMillis();  
  
 String m = ( (l2-l1) / 1000 / 60 ) + "";  
 String s = ( (l2-l1) / 1000 % 60 ) + "";  
 System.out.println( fileName + "下的文件已经全部遍历完成！共耗时" + m + "：" + s + "分钟");  
 }  
 public static String printFile( File[] arr ){

// 如果是文件类型就直接将信息打印出来  
 for( File f : arr ) {  
 if (!f.isDirectory()) {  
 System.out.println(s + "【" + f.getName() + "】" + "是文件");  
 } else {

// 如果是文件夹类型就将文件夹打印出来并再次返回该文件夹下的所有文件传至递归方法中，无限循环  
 System.out.println(s + "【" + f.getName() + "】" + "是文件夹");  
 File []files = f.listFiles();  
 if( files == null )  
 continue;  
 s.append("--");  
 printFile( files );  
 s = new StringBuilder(s.substring(0, s.length() - 2));  
 }  
 }

// 当代码执行到这里的时候，证明循环已经结束，所有文件及文件夹都已遍历完成，返回结果即可  
 return "【============所有文件已经遍历结束============】";  
 }  
}

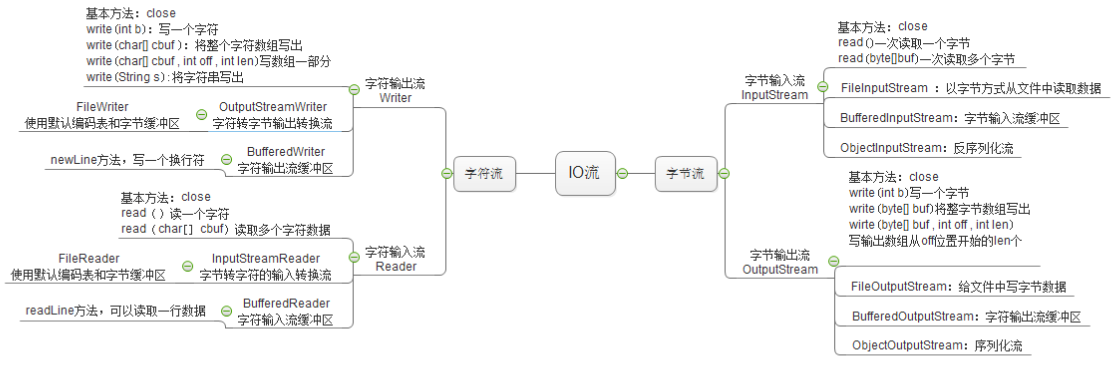
FileFilter接口：

FileFilter过滤器的主要目的是将符合条件的内容留下，将不符合条件的剔除掉

定义一个类，实现FileFilter方法，并复写其中的accept方法，

#### IO流

IO流结构大纲：



输入流：

将网络；文件；磁盘等数据写入到应用程序中

输出流：

将应用程序中的数据写出到网络；文件；磁盘中

### 多线程

什么是进程：

加载到内存中被运行的应用

什么是线程：

负责运行应用作用或应用中的某个功能的一块内存

一个应用中必须至少有一个线程

什么是多线程：

在一个时间段中有多个线程工作，被称为多线程

多线程运行原理：

多线程实际上运行的就是一个线程，不过他在极短的时间内在多个

### 反射技术

当一个普通的java类或者抽象类,接口,枚举或内部类被编译之后,都会生成一个后缀名为class的字节码文件,在Java中用Class类来描述这类事务.

一段Java代码在计算机中会经历三个阶段：

1. 源码阶段，就是程序员书写出来后缀名为**.**java的文件，被存放在硬盘中，而后又翻译为后缀名为**.**class的文件

2. 加载阶段，硬盘中书写好的Java程序不会自己主动运行，需要将程序加载到内存中，加载的过程由Java中的类加载器实现

3. 运行阶段，当Java代码被加载到内存的时候就时运行时状态，可以调用其允许被调用的所有方法及属性等

#### 获取Class对象

获取Class对象的三种主要方法:

1. 通过Class类的静态方法forName获取

Class cs = Class.forName(“全限定包名”)

1. 通过类的class属性获取

Class cs = Person.class;

1. 通过Object类的getClass方法获取

Class cs = Person.getClass();

只要操作的是同一个class文件，无论用什么方法，获取多少个，所获取到的都是同一个class对象

*// 获取class对象的三种方式：*Class c1, c2, c3, c4;  
*// 1. 通过全限定名获取*c1 = Class.*forName*(**"club.hanzhe.bean.Cat"**);  
*// 2. 通过对象的class属性获取*c2 = Cat.**class**;  
c3 = Cat.**class**;  
*// 3. 通过对象中的方法获取*c4 = **new** Cat().getClass();  
System.***out***.println(c1);  
System.***out***.println(c1 == c2); *// true*  
System.***out***.println(c1 == c3); *// true*  
System.***out***.println(c1 == c4); *// true*

#### 反射类中属性

反射类中的属性，有四种方法：

1. getField(String name); 获取指定name的属性（public）

2. getFields(); 获取全部属性（public）

3. getDeclaredField(String name); 获取指定name的属性 (无视权限修饰符)

4. getDeclaredFields(); 获取全部属性 (无视权限修饰符)

针对获取到的对象也可以进行get set方法

对于获取到的私有属性，在进行操作之前需要取消访问权限的检查

*// 获取反射对象*Cat c = **new** Cat();  
Class cat = c.getClass();  
  
Field[] fields1 = cat.getFields();*// 获取全部字段（public）  
// 因为只能获取到public修饰的字段，所以只获取到了sex 一个属性***for**( Field f : fields1 )  
 System.***out***.println(f);  
  
Field sex = cat.getField(**"sex"**); *// 获取指定名称的字段（public），返回一个field对象*System.***out***.println( sex.get(c) ); *// 打印获取到的字段，因未赋值，结果为null*sex.set(c,**"雄性"**);  
System.***out***.println( sex.get(c) ); *// 打印获取到的字段，因未赋值，结果为null*System.***out***.println(**"【=========================】"**);  
  
Field[] fields2 = cat.getDeclaredFields(); *// 获取全部字段（无视修饰符）***for**( Field f : fields2 )  
 System.***out***.println(f);  
Field name = cat.getDeclaredField(**"name"**);  
***// 无视修饰符的情况下，所有字段都已经获取到，但是想要操作非public的字段，还需要设置忽略权限访问修饰符*name.setAccessible(true);**name.set(c,**"袁华"**);  
System.***out***.println( name.get(c) );

#### 反射类中构造器

*// 首先获取到class对象*Class person = Class.*forName*(**"club.hanzhe.bean.Cat"**);  
  
*// 反射有参构造，其参数应该对应构造方法内的参数的class对象*Constructor con1 = person.getConstructor(Class.*forName*(**"java.lang.String"**), String.**class**, **new** String().getClass());  
Object p1 = con1.newInstance(**"放假撒了点"**,**"橘黄色"**,**"雌性"**);  
System.***out***.println(p1);  
  
*// 反射无参构造*Constructor con2 = person.getConstructor();  
Object p2 = con2.newInstance();  
System.***out***.println(p2);  
  
*// 针对无参构造，可以进行省略缩写*Object p3 = person.newInstance();  
System.***out***.println(p3);

## 设计模式

### 单例设计模式

单例设计模式，只允许外部获取同一个实例，不会创建第二个实例

单例模式实现代码：

**class** Demo{  
 **private static** Demo *d* = **null**;  
 **private** Demo(){}  
 **public static** Demo getInstance(){  
 **if**( *d* == **null** )  
 *d* = **new** Demo();  
 **return** *d*;  
 }  
}

上方单例模式仅仅在单线程中可以使用，在多线程的情况下是不安全的，如果想要实现多线程下安全需要对代码进行一些改动：

**class** Demo{  
 **private static** Demo *d* = **null**;  
 **private** Demo(){}  
 **public static synchronized** Demo getInstance(){  
 **if**( *d* == **null** )  
 *d* = **new** Demo();  
 **return** *d*;  
 }  
}

### 代理设计模式

代理设计模式是针对方法进行增强的一种手段，代理模式分为两种：

1. 静态代理 → 依靠继承或实现接口

2. 动态代理 ↓

2.1 基于类的动态代理cglib

2.2 基于接口的动态代理JDK

#### 静态代理

应用场景-例如：当一个项目已经完成，但临时需要增加一个日志功能，本着不修改源代码的原则，就可以使用代理模式

源代码：

**public class** UserDaoImpl **implements** UserDao {  
 @Override  
 **public void** add() {  
 System.***out***.println(**"添加一名用户"**);  
 }  
 @Override  
 **public void** drop() {  
 System.***out***.println(**"删除一名用户"**);  
 }  
}

代理类：

**public class** Proxy2 **implements** UserDao {  
 **private** UserDaoImpl **userDao**;  
 **public** Proxy2( UserDaoImpl userDao ){  
 **this**.**userDao** = userDao;  
 }  
 @Override  
 **public void** add() {  
 log(**"add"**);  
 **userDao**.add();  
 }  
 @Override  
 **public void** drop() {  
 log(**"drop"**);  
 **userDao**.drop();  
 }  
 **public void** log(String msg){  
 System.***out***.println(**"[Debug] 执行了"** + msg + **"方法"**);  
 System.***out***.print(**"[Debug] "**);  
 }  
}

#### 动态代理