# 访问控制权限：

JAVA中一共四种。public、protected、default、private

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| No: | 范围 | private | default | protected | public |
| 1 | 在同一个类中 | √ | √ | √ | √ |
| 2 | 在不同类中 |  | √ | √ | √ |
| 3 | 在不同包的子类 |  |  | √ | √ |
| 4 | 在不同包的非子类 |  |  |  | √ |

Protected也是封装，但是他的封装对本包没用，对不同包的子类没用，对不同步包非子类才有用。

JAVA的封装性是以private、protected、default三中全线三种权限的定义；

基本的原则：

声明属性用private;

声明方法用public;

# 关于命名要求：

·类名称每一个单词的首字母大写，其余的字母小写，例如：StudentInfo;

·属性名称的第一个单词字母小写，而后每个单词首字母大写，例如：studentName;

·方法名称的第一个单词字母小写，而后每个单词首字母大写，例如：studentName();

·常量名称用大写字母表示；例如:MSG；

包名称使用小写字母，例如：cn.test.demo;

如果是接口的命名，则在前面加上字母I，例如：IMessage

如果是抽象类的命名，则在前面加上Abstract,例如AbstractMessage

普通类就直接编写：Message

# 引用是JAVA的精髓所在

# 递归（定义）：

程序调用自身的编程技巧称为递归，一种计算过程，如果其中每一步都要用到前一步或前几步的结果，称为递归的。用递归过程定义的函数，称为递归函数，例如连加、连乘及阶乘等。凡是递归的函数，都是可计算的，即能行的。

# 输入：

int y=Integer.parseInt(args[0]); //获取的是输入的第一个

这个就是输入格式，

Int x=Integer.parseInt(args[1]}); //获取的是输入的第二个。

中间以空格隔开

# 方法（函数）

（重载就是一名多用）

## 定义方法其实就是c++里的类外定义函数，保证了主函数的简洁性

## 定义方法的方法

注意：每一个方法都是类，每个类都是一个java文件

定义：第一个单词的字母小写，后面的单词首字母大写

### Eg1:方法的定义

public class Hello{

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

public static void printInfo(){

}

}

### **格式：**

public class 返回值类型 方法名称（参数类型 参数变量，………可以很多个）{

方法体（即方法执行的若干操作）；

后面可以 return(返回值)； （通常都有返回值）

}

### 注意：

void 是没有返回值类型 所以不用ruturn

当一个功能需要多次使用，即重复出现的时候，可以定义为方法

## Eg2: 利用函数的直接或间接输出结果

public class Hello

{

public static void main (String args[]){

int result=add(3,4);

//直接将返回值输出

System.out.println(add(3,4));

//间接输出成绩

System.out.println("计算的级果："+result);

}

public static int add(int x,int y){

return x+y;

}

}

## Eg3:利用return来终止函数的调用

public class Hello

{

public static void main (String args[]){

set(100);

set(3);

set(10);

}

public static void set(int x){

if (x==3){

return ;

}

else{

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

}

}

}

# 面向对象的介绍

## 面向过程：

是指针对某一个问题单独的提出解决方案的以及代码开发。（分析出解决问题所需要的步骤，然后用函数把这些步骤一步一步实现，使用的时候一个一个依次调用就可以了。）

## 面向对象：

以一种组件化的形式进行代码的设计，这样开发出来的代码有一个最大的好处就是重用。（其实就是把构成问题事务分解成各个对象，建立对象的目的不是为了完成一个步骤，而是为了描叙某个事物在整个解决问题的步骤中的行为。）

面向对象更适合程序维护。

特性：

封装性：保护内部结构的安全性。

继承性：在已有的程序基础上，再继续扩充。

多态性：在某一个概念范围内的满足。

步骤：

OOA（面向对象的分析）、OOD（面向对象的设计）、OOP（面向对象的编程）

# 类与对象

# 格式：定义一个类

class 类名{

成员1；

成员2；

}

## 注意：

### 在主类中定义方法要加static，在另外定义的类中定义方法不用加上static。

### 每一个类都是一个java文件，类中定义的成员函数供对象调用

### 对象属于引用数据类型,基本数据类型是要进行内存分配的

## 对象的定义格式及步骤

### **声明并实例化对象：**类名称 对象名称=null;

### 实例化对象：对象名称=new 类名称（）；

或者 类名称 对象名称

## new的功能：

引用数据类型与基本数据类型的区别就是内存空间的开辟，而new的功能就是开辟内存空间，开辟了内存空间的才是实例化对象。new就是用来实例化对象。对象的创建一定经历过对象的实例化、默认值的设置、构造方法

#### 内存空间的概念：

**堆内存：**保存每一个对象的属性内容，堆内存需要关键字new才可以开辟。一定保存有一个地址。(开辟了堆内存的对象才是实例化对象)

**栈内存：**保存的是一块堆内存的地址，为了分析方便，可以简单的理解栈内存保存的是**对象名字**。

EG：

class student{}

student st1=new student(); st1就是栈内存,即对象名字

利用new 开辟的空间

栈内存保存地址

堆内存保存属性

st1.name

st1.age

堆内存

栈内存

st1

### Eg2:

//对象的引用传递

class student

{

String name;

int age;

double high;

public void get(){

System.out.println("学生姓名为："+name+" 年龄为："+age+" 身高为："+high);

}

}

public class studentTable

{

public static void main(String args[]){

student st1=new student();

student st2=null;

st1.name="xiaoming";

st1.age=18;

st1.high=170.1;

st2=st1; //这里就是引用传递

st2.name="bigpig";

st2.age=19;

st2.high=180.3;

st1.get();

st2.get();

}

}//注意上述代码的赋值顺序

## 引用：

在引用中一块堆内存可以同时被多个栈内存所使用，但是一块栈内存只能

一块没有栈内存指向的堆内存，将会等待电脑的清理，并释放掉占用空间，开发之中应尽量减少垃圾代码的产生，垃圾空间是指没有栈内存指向的堆内存空间。

# 类的封装性：就是仅内部可见。

class Book{

private String name;

private int age;

}

在开发之中，类中的所有属性都（要求）建议用private来声明，如果属性被外部所需要，那么按照要求定义相应的setter,getter方法

## setter方法：

setter方法主要是设置内容，eg: public void setname(String t); 有参

函数名跟着属性同名，首字母大写。

## getter方法：

getter方法主要是取得属性内容,eg: public String getTitle(); 无参

getter只是简单的返回数据。

标准的开发之中，setter是简单的设置数据，getter是简单的取得数据。

# 构造方法与隐匿函数

类名称

类名称（）

new

对象名称

对象名称

构造方法

若在类中定义,没有写构造方法,他还是会自动定义.

构造方法是对象在使用new实例化的时候调用，且只调用一次

而普通方法是在实例化对象之后调用多次。

## 格式：

public 类名称（） {} 注意：一定是在类中定义

若定义了构造方法，则默认的构造方法不会调用。一个类中至少保留有一个构造方法。

**总的来说构造方法是为属性初始化准备的！**

构造方法是没有返回值的，但也不用加void。构造方法重载时注意参数的类型及个数。

构造方法重载时，请按照参数的个数依次排列。

# 匿名对象：

匿名对象调用方法有意义，而调用类中的属性则没有意义：例如：new Student().name="wangwu";,这属于使用匿名对象调用属性，那么为什么说这样做没有意义呢？当代码执行到这一行代码时则修改new Student()对象的name属性，可是当该条语句被执行后呢？由于这时new Student()没有继续引用，所以此对象在堆中就成了垃圾。那么为什么说匿名对象调用方法有意义呢？方法存在于栈空间，当在内存中创建的对象调用方法时，JVM开始为局部变量开辟内存空间，方法开始执行，方法强调其功能，对象调用方法可以完成其功能，所以匿名对象调用方法有意义。

匿名对象只能够使用一次，且使用后就成了垃圾空间。

# JAVA开发的第一要求：

1. 类名称的存在必须有意义。
2. 类中的属性必须封装。pirvate，封装的属性必须有setter、getter。
3. 类中课提供多个构造方法，但必须保留一个无参构造。
4. 类中不予许出现任何输出语句，有也是只能用方法输出

# 数组：

## 定义：

声明并开辟数组 一步完成的 数组类型 数组名称[]=new 数据类型[长度]

分布完成的：

数组名称[]=null； 中括号在前在后都没问题，统一写后面吧。

数组名称=new 数据类型[长度]；

**跟C里面的数组差不多，注意：长度一定是确实长度，长度为3，下标是 0，1，2.数组的下标从0开始**

数组的定义是，先开辟内存空间，后赋予数组类型的初值。

数组的定义简化格式：数据类型 数组名称[]={值，值，值…}；

数组定义的完整格式：数据类型 数组名称[]=new 数据类型[]{值，值，值…}；

以上的格式是静态初始化。

### Eg:

int data[]=new int[3];

System.out.println(data.length);

## 数组的内存分配问题：

**数组也是引用数据类型，只要是引用数据类型，就会牵扯到内存分配问题，**对象的堆内存保存的是属性，数组中的堆内存保存的是一组信息（即索引值和内容）。栈内存就是数组名。

没开辟内存空间，会出现空指向异常。

数组一可以发生引用传递，**引用传递的实质**同一块堆内存可以被多块栈内存所指向。数组名是指向堆内存空间的地址。见**例子：数组的引用传递**

数组虽然支持按顺序的数据访问操作，但是有一个极大的缺点，就是数组的长度不会发生改变，开发中一般不会使用，但是会使用到数组的概念——保存多个变量。

## 二维数组：

保存多个数据，就像一张数据表，有行有列，就是定义一维数组的时候，后面再加个中括号。确认数据使用行和列一起定位。

推广到多维数组，后面继续加中括号。

### 定义格式：

**动态初始化：**数据类型 数组名称[][]=new 数据类型[行的个数][列的个数]

**静态初始化：**数据类型 数组名称[][]=new 数据类型[][]{{值，值，值，值}，{值，值，值，值}}；

相当于多个一维数组集合在一起。**二维地址的首地址：数组名称[]。**

二维数组的输出用两个for循环，外层控制**行**内容，内循环控制**列**的内容。

二维是平面，三维是立体。深入掌握用处感觉也不大。

## 数组与方法参数的传递

public class arrayDemo

{

public static void main(String args[]){

int data[]=new int[]{1,2,3}; //定义一个长度为3的整型数组并赋予1，2，3

int temp[]=data; //temp数组是在change方法里定义的，这也是引用传递问题

for (int x=0;x<temp.length ;x++ ) //数组值的翻倍方法

{

temp[x]\*=2;

}

for (int x=0;x<data.length ;x++ ) //利用for循环输出data数组的值

{

System.out.println(data[x]);

}

}

public static void change(int temp[]){ //定义一个方法，专供数组使用

}

}

多个栈内存，指向同一个堆内存。

在进行数组的引用传递过程中，方法对数组的修改一定会影响到原始数据。

# 冒泡排序法：

就是左边依次和右边的比较，小的就交换，大的就不动。

原始数据：2，1，3，5，4，8，7，6，9

//2，1比较，然后1移到左边，然后2，3比较，2不变。3在和后面的比较

第一次排序：1，2，3，4，5，7，6，8，9

第二次排序：1，2，3，4，5，6，7，8，9 //即完成排序

最大的次数不会超过长度\*长度，在这之前，一定会排序成功。

## 数组的转置：

数组的转置就是头尾交换,

思路一:定义一个数组将其尾部赋予头部.

思路二:利用算法在数组上完成转置操作.

原数组:1、2、3、4、5、6、7、8

第一次转换：8、2、3、4、5、6、7、1 转换次数为数组长度的一半；

第二次转换：8、7、3、4、5、6、2、1 当数组长度为奇数时，用int型，计算的结

第三次转换：8、7、6、4、5、3、2、1 果相同，

第四次转换：8、7、6、5、4、3、2、1

System.out.println(8/2);

System.out.println(7/2);结果时相同的，自动取整。

### 二维数组的转置：

即 行跟列的交换。中间那条线几乎不动，行数的索引和列数的索引相同。

## 数组的操作：

### 数组的拷贝：

System.arraycopy(源数组名称，源数组拷贝开始索引，目标数组名称，目标数组拷贝开始索引，长度)；//注意数组开始的索引是0

### 数组的排序：

格式： java.util.Arrays.sort(数组); 即可完成从小到大的排序。

# 对象数组（important）：

就是引用数据类型中包含了多个引用数据类型。

## 格式：

动态初始化：类名称 对象数组名称[]=new 类名称[长度]；

分步完成：类名称 对象数组名称[]=null; //开辟之后都是null值。

类名称 对象数组名称[]=new 类名称[长度]；

静态初始化：类名称 对象数组名称[]=new 类名称[]{实例化对象，实例化对象…}

**对象数组中的每个对象都需要分别实例化，对象数组一般定义为一维。数组有一个天生短板，不可改变数组长度。**

**对于笔试题，排序问题，不要写 java.util.Arrays.sort(数组名);**

# String型：

String型，也是String类

内容比较函数 public boolean equals(String str);

所有的比较可以直接用 == ，直接输出即可，若为真，即true，否则反之。

EG：public class StringDemo

{

public static void main(String args[]){

String str1="Hello"; //静态初始化

String str2=new String("Hello"); //动态初始化

String str3=str2; //引用传递

//字符串内容的比较

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

System.out.println(str1.equals(str3)); //true

System.out.println(str2.equals(str3)); //true

//字符串用"==的比较"

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

System.out.println(str1==str3); //false

System.out.println(str2==str3); //true

}

}

## 问：字符串比较中“A==B”和A.equals(B)两种比较的区别？

“==”是java提供的关系运算符，主要功能是进行数值的相等判断，如果用在了String对象上，表示的是内存地址的比较。

equals：是由String类提供的一个方法，主要供字符串内容的比较。

## String的匿名对象：

字符串常量就是String 的匿名对象。

Java里面提供了字符串，但是他不属于基本数据类型

public class StringDemo

{

public static void main(String args[]){

String str="Hello";

System.out.println("Hello".equals(str)); //true

}

}

//证明了字符串是匿名对象,原因是字符串能够被调用，匿名对象储存在堆内存里。

## Pay attention:

所谓的直接赋值就是给匿名对象设置了一个名字而已。

## 开发小技巧：

空指向异常的出现就是：使用了未实例化的对象，或者使用了未实例化的引用数据类型。为了避免因为用户的输入错误，导致输入为空，从而导致空指向异常。

### Eg1:

public class StringDemo

{

public static void main(String args[]){

String str="null ";

if (str.equals("Hello"))

{

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

}

}

}

假设是开发，如果用户的关系输入到的null，这样会出现空指向异常。

但是反过来，匿名对象在前面，调用方法，就不会出现空指向异常。

### Eg2:

public class StringDemo

{

public static void main(String args[]){

String str=null;

if ("Hello".equals(str))

{

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

}

}

}

所以开发中，判断输入是否相同，将字符串写在最前面，这也是开发准则。

## 两种实例化对象的区别：

共享对象池：在JVM的底层会有一个对象池（不一定只保存String 的对象），当代码中使用直接赋值的方式时，会自动的将匿名对象入池保存，而后的其他String类定义相同的内容时，那么将不会开辟新的内存空间，进行引用分配，从而继续使用，是节约内存的一种方式。

2：采用构造方法实例化对象

一旦使用了new就一定会开辟新的内存空间。

String str=new String (“Hello”)；

堆内存

栈内存

“Hello”

Str

“Hello”

最后会使用new开辟的空间，直接赋值的会成为垃圾空间

**但是还会开辟两块内存空间。虽说会产生垃圾空间，但是没有入池保存问题。直接赋值会直接入池。**

**但是有手工入池的方法eg: String str=new String(“Hello”).intern();**

**方法：.intern()**

## 字符串一旦定义则不可改变。

### 例子：

String str=”Hello”; str=str+”World”;

**这样的代码，他会先断开str指向“Hello”,然后指向“Hello World”，从而“Hello”和”World”就没有栈内存指向，从而成为垃圾空间。但是该字符串的内容是没有改变的，改变的是栈内存的指向！！！ 所以每次的改变都会产生垃圾空间。**

**注意：任何类型的数据和String相加减，都会转化为String，String内容不要频繁的修改，这样会产生大量的垃圾空间，这在开发中是禁止的。**

**开发要学会查文档，打开JDK1.8。打开文档找到java.lang**

**如果文档中有Drecpeted的字样，就是建议不在使用了**

# 字节与字符串：

见方法文档，字节与字符串

在后续的IO操作的时候，会牵扯到字节的操作

|  |  |  |  |
| --- | --- | --- | --- |
| NO： | 方法 | 类型 | 功能及说明 |
| 1 | public Boolean equals(String anObject) | 普通 | 进行相等判断，区分大小写 |
| 2 | public boolean equalsIgnoreCase(String anotherString) | 普通 | 进行相等的判断不区分大小写 |
| 3 | public int compare To(String anotherString) | 普通 | 判断两个字符串的大小（按编码比较），有三种结果：   1. =0，表示要比较的两个字符串内容相等 2. >0，表示大于的部分 3. <0，表示小于的部分 |

**格式：A·方法（B）;**

# this关键字：

## 概念：

this表示当前对象，可实现类属性的调用、类方法的调用、表示当前对象

## 调用属性：

在定义的时候，如果属性名称和参数名称重名了，就会取最近的括号里头找，如果需要准确的找到要访问的变量，属于类中的属性时，可以在变量前加上this。

EG：class Song

{

private String name;

private double lon;

public Song(String name,double lon){

this.name=name; //区别

this.lon=lon;

}

public String getinfo(){

return "名字: "+this.name+" 时长："+this.lon; //区别

}

//setter,getter省略

}

public class Hello

{

public static void main(String args[]){

Song son1=new Song("一首歌",83.5);

System.out.println(son1.getinfo());

}

}

以后再开发中，访问类中的属性都加上this ，这是最稳妥的开发形式。

## 调用方法：

在类中定义的方法就是，属于本类的，在开发中，方法前面是否加this没有明确的要求，即使不加this 也表示本类的方法，最好也加this，这并不麻烦，但是有构造方法，和普通方法，这就麻烦了，多个构造方法互相调用的时候this(参数，参数)。在开发之中，有重复代码就必须消除重复代码。

### Eg:

class Song

{

private String name;

private double lon;

public Song(){

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

}

public Song(String name){

this(); //调用本类中的无参构造

this.name=name;

}

public Song(String name,double lon){

this（name）; //调用本类中的单参构造 //区别

this.lon=lon;

}

public String getinfo(){

return "名字: "+this.name+" 时长："+this.lon; //区别

}

//setter,getter省略

}

public class Hello

{

public static void main(String args[]){

Song son1=new Song("一首歌",83.5);

System.out.println(son1.getinfo());

}

}

## Pay attention:

用this指针，调用无参构造函数时，必须放在首位，且只有构造函数才能调用构造函数。**普通函数是无法调用构造函数的！如Setter、getter函数。构造只能调用一次**

**且必须递归构造调用，尽量调用构造的时候保留一个没有调用其他构造的情况，比如例子：图书类。**

## 表示当前对象：

所谓的当前对象就是指当前正在调用类中方法的对象。

class Book

{

public void print(){

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

}

}

public class BookDemo

{

public static void main(String args[]){

Book book1=new Book();

System.out.println(book1);

book1.print();

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

Book book2=new Book();

System.out.println(book2);

book2.print();

}

}

即：this指向的是栈内存。this属性就是当前对象中的属性。this就是当前对象。

# 引用传递：（JAVA的精髓）

同一块堆内存空间，可以被多个栈内存所指向。不同的栈内存可以对同一块堆内存进行内容修改。

**String类对象的内容一旦声明便不可改变。**

## EG：

public class Hello

{

public static void main(String args[]){

String msg="Hello";

fun(msg); //引用传递

System.out.println(msg);

}

public static void fun(String temp){

temp="World";

}

}

**//最后的结果是“Hello”，对象内容的改变依靠的是引用地址的改变。**

**一开始引用传递，temp断开指向”World”指向堆内存里的”Hello”但是最后结束的时候，会断开temp的指向，temp和“World”成为垃圾空间，且msg一直是指向“Hello”没有改变，若不输出Hello则需要改变msg的指向。没有指向就会成为垃圾空间。很多时候直接把String当成基本数据类型来使就很好理解了。**

## 简单java类的编写原则：（数据表）

类名称=表名称；

属性名称（类型）=表字段（类型）；

一个是李华对象=一行记录；

多个实例化对象（对象数组）=多行记录；

引用关系=外界约束；

**代码链就看返回值。一块或多块的时候用对象数组来表示。合成设计模式。**

在细分

在细分

在细分

在细分

主体

主体部分

主体部分

**每一个方框都是一个类，类在互相关联。**

**所以说引用是实现两个不同类型之间相互关联的主要手段。**

# JAVA类的实践：

第一步，实现基本字段的转换。先罗列出类

第二部，实现关联字段的转换。

第三部，设置、取得数据。

# Java的对象比较：

## 注意：

如果属性用private封装了，那么在外部就不能直接通过对象直接调用属性。

如果说将一个对象传递回到类的方法里面，那么就相当于取消了封装形式，可以直接通过对象访问属性。

//在类的内部调用对象直接访问私有属性

class Info

{

private String msg="Hello";

public void print(){

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

}

//在本类接收对象

public void fun(Info temp){

//在类内部直接利用对象访问私有属性

temp.msg="修改内容";

}

}

public class Hello

{

public static void main(String args[]){

Info x=new Info();

x.print();

}

}

进行比较的时候利用b1.compare(b2)；compare函数自己定义。

可以利用if（this==自身）就直接返回，这样就可以直接节约时间。但是若为null则直接返回null的判断。以后的程序设计，必须提供一个对象比较的方法。

# Static 全局变量：

Static定义的全局变量，所有的堆内存都指向。避免了重复，全局变量就是一块公用的堆内存空间，只有保存的区域不同。Static属性可以由类名称，直接调用的。Static属性，在没有实例化对象产生的情况下，一样可以调用。虽然定义在类中，但是不受类的结构的控制。也可以用 . 来访问。

没有实例化，不用this 指针一样可以调用setter方法。

方法也变成了两组，static方法和非static方法。

Static方法不能调用非static方法。

非static方法可以调用，static方法，属性。

# 主方法：

1. public：主方法是程序的开始，所以这个方法对任何的操作都一定是可见的，public(公共的意思)
2. static：证明此方法是有类名称调用的；
3. void:主方法是一切执行的开始点，既然是所有的开头，那么就不能回头
4. main:是一个系统规定好的名称，不能够修改；
5. String args[]:指程序运行的时候传递的参数；

使用：

public class TestDemno

{

public static void main(String args[]){

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

System.out.println(args[x]);} }}

作用地点是执行dos命令的时候，即“java TestDemo 参数 参数”，所有输入的参数必须使用空格分割，如果是输入一连串的，那就用“”括起来，利用上述代码即可证明。

# 代码块

在程序的编写之中，用“{}”定义的一段语句，其实就是用“{}括起来的一段语句”，代码块分为四种：普通代码块、构造快、静态块、同步代码块（多线程的时候用的到）。写代码的时候建议不要使用代码块

## 普通代码块：

### 观察，比较下面两段代码：

### ：

public class TestDemno

{

public static void main(String args[]){

int num=10;

System.ou.println("num= "+num);

int num=100;

System.ou.println("num= "+num);

}

}

}

这是一段错误代码块，因为同一个变量在一个代码块类不能重复定义。

### ：

public class TestDemo

{

public static void main(String args[]){

if(true){ //条件一定满足

int num=10; //局部变量

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

}

int num=100; //全局变量

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

}

}

### ：

public class TestDemo

{//普通代码块

public static void main(String args[]){

{ //条件一定满足

int num=10; //局部变量

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

}

int num=100; //全局变量

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

}

}

把if语句去掉了，剩下的就是 普通代码块了

## 构造代码块：

如果代码块写在了一个类里面，那么就称其为构造块

### 观察下列代码的结果：

class Book

{

public Book(){

//构造方法

System.out.println("A 普通版块"); //构造方法中的普通版块

}

//构造板块

{

System.out.println("B 构造板块");

}

}

public class TestDemo

{

public static void main(String args[]){

new Book();

}

}

### 结果是：

D:\testdemo>java TestDemo

B 构造板块

A 普通版块

实例化对象的时候，虽然执行的是Book的构造方法，但是最终构造块的调用优先于构造方法的执行。如果实例化了多个对象，构造块会被重复执行。

## 静态块：

如果一个代码在定义中使用了static进行定义的话，那么就称其为静态块，静态块的使用又分为两种情况；

### ：在非主类中使用

实例：

class Book

{

public Book(){

//构造方法

System.out.println("A 普通版块"); //构造方法中的普通版块

}

//构造板块

{

System.out.println("B 构造板块");

}

static {

System.out.println("C 静态块");

}

}

public class TestDemo

{

public static void main(String args[]){

new Book();

new Book();

new Book();

}}

结论：静态块的执行在构造块之前，且不管实例化多少次对象，只执行一次。

目的大多数是执行程序的初始化操作。如随机赋初值。

### 在主方法中定义静态块：

public class TestDemo

{

static {

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

}

public static void main(String args[]){

System.out.println("Hello word");

}

}

结果是：

D:\testdemo>java TestDemo

\*\*\*\*\*\*\*\*\*

Hello word

说明静态块比主函数还要优先执行，JDK1.5还是1.7以前，静态块是可以代替主方法的，但是现在不行了。必须要有主方法。

# 注：所有的代码块，能不用就别用，唯一好用的就是静态块。

# 内部类：

1. 内部类的基本定义结构；
2. 使用static定义内部类；
3. 在方法中定义内部类；

## 注：从开发的角度，内部类能少用就少用，优先考虑普通类

## 定义：

就是在一个类的内部定义了其他内部结构类的情况，可以再一个内部类中定义多个内部类。

### 实例：

class Outer //外部类

{

private String msg="Hello world!";

class Inner //定义了一个内部类

{

public void print(){

System.out.println(msg);

}

}

public void fun(){

//实例化内部类对象

new Inner().print();

}

}

public class TestDemo

{

public static void main(String args[]){

Outer out=new Outer(); //实例化外部类对象

out.fun(); //调用外部类方法

}

}

一个类中定义了另一个类，很别扭，相当于牺牲了结构，但是达到了一个很重要的目的，内部类可以方便的访问外部类的私有属性，反之，外部类也可以通过内部类对象访问内部类的私有属性。

若在文件中查class文件的时候，查处了$符号，这个换回到程序里的时候就是“.”符号。

内部类不可能离开外部类的实例化对象，所以在内部类实例化对象之前，一定要先实例化外部类对象， 内部类是不可能在外部实例化对象的。

## 在方法中定义内部类：（最常用的形式）

实例：

class Outer //外部类

{

private String msg="Hello world!";

public void fun(){

class Inner //在方法中定义内部类

{

public void print(){

System.out.println(Outer.this.msg);

}

}

new Inner().print();

}

}

public class TestDemo

{

public static void main(String args[]){

new Outer().fun();

}

}

若不加修饰，方法中的内部类可以访问方法参数以及定义的变量。但是这种操作只适合于JDK1.8以后的版本。因为大多数企业以前开发用的是1.7版本，所以在1.7版本以前有一个严格的要求，方法中定义的内部类如果要想访问方法的参数或者定义的变量，那么参数前必须加上“final”标记。，就是说在上述代码中定义形参的时候，即final int num=100;

## 匿名内部类：

**package** cn.test.deom;

**interface** Message{

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

}

**public** **class** TestDemo {

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

*fun*(**new** Message(){

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

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

}

});//注意，括号

}

**private** **static** **void** fun(Message msg) {

msg.print();

}

}

这样子在省略了实例化对象，使用匿名内部类有一个前提：必须基于接口或者抽象类的应用。节约了定义的子类。

注意：如果匿名内部类定义在了方法里面，方法的参数或者是变量要被匿名内部类所访问的话，那么必须加上final关键字（JDK1.8以后此要求被改变）。

# 链表：

保存数据

指向下一个节点指针

保存数据

Node

一个基点的示意图。

实例：

class Node //定义节点类

{

private String Data; //要保存的数据

private Node next; //连接下一个节点的指针

public Node(String Data){

this.Data=Data;

}

public void setNext(Node next){ //每一个相应的Node必须保存有相应的数据

this.next=next;

}

public String getData(){

return this.Data;

}

public Node getNext(){

return this.next;

}

}

public class LinkDemo

{

public static void main(String args[]){

Node head=new Node("头结点");

Node n1=new Node("第一个节点");

Node n2=new Node("第二个节点");

Node n3=new Node("第三个节点");

head.setNext(n1);

n1.setNext(n2);

n2.setNext(n3);

//从头结点开始读取

Node currentNode=head; //该句话的意思是将头结点的地址赋给currentNode

System.out.println(currentNode);

while (currentNode !=null)

{ //从当前节点读取数据

System.out.println(currentNode.getData());

//将下一个节点设置为当前节点

currentNode=currentNode.getNext();

}

}

}

## 建议：

另外建一个类，专门实现数据的增加，和输出数据，但是数据要保存在主类中

基本操作特点：

客户端代码不用去关注具体的Node以及引用关系的细节，只关注于提供的Link类中的方法

Link类的主要功能是控制Node类对象的产生和根节点；

Node类主要负责数据的保存以及引用关系的分配。

**我们可以使用一个内部类来完成操作**

### 优点：

内部类可以用private来定义，这样就可以只被一个外部类来使用，另外，内部类可以方便的与外部类进行私有数据的直接访问。

## 给定一个链表的开发结构：

class Link //链表类，外部类

{

//内部类，定义在内部是为了链表的服务,加上private是为了他只被一个外部类所服务

private class Node //定义的节点类

{

private String data;

private Node next;

private Node(String data){

this.data=data;

}

}

private Node root; //根节点

}

## 十大功能：

|  |  |  |  |
| --- | --- | --- | --- |
| NO： | 方法名称 | 类型 | 描述 |
| 1 | public add(数据类型 数据变量) | 普通 | 向链表中增加新的数据 |
| 2 | public int size() | 普通 | 取得链表中保存元素的个数 |
| 3 | Public boolean isEmpty() | 普通 | 判断是否为空链表size()==0 |
| 4 | Public Boolean countain(数据类型 数据变量) | 普通 | 判断某一个数据是否存在 |
| 5 | Public 数据类型 get(int index) | 普通 | 根据索引取得数据 |
| 6 | Public void set（int index，数据类型 变量） | 普通 | 修改链表数据 |
| 7 | Pubic void remove(数据类型 变量) | 普通 | 删除指定数据 |
| 8 | Public 数据类型 []toArray() | 普通 | 将链表以对象数组的形式返回 |
| 9 |  | 普通 |  |

### 增加数据

class Link //链表类，外部类

{

//内部类，定义在内部是为了链表的服务,加上private是为了他只被一个外部类所服务

private class Node //定义的节点类

{

private String data;

private Node next;

private Node(String data){

this.data=data;

}

public void addNode(Node newNode){

if (this.next==null) //当前的下一个节点为空

{

this.next=newNode;

}else{ //向后继续保存

this.next.addNode(newNode);

}

}

}

private Node root; //根节点

public void add(String data){//假设不予许有空

if(data==null){

return;

}

Node newNode=new Node(data); //要保存的数据

if (this.root=null) //如果没有根节点

{

this.root=newNode; //保存根节点

} else{ //根节点存在，其它节点交给Node类处理

this.root.addNode(newNode);

}

}

}

public class LinkDemo

{

public static void main(String args[]){

Link all=new Link()'

all.add("Hello");

all.add("World");

all.add(null);

}

}

使用了一个不许为null的判断，但并不是所有的链表都不许为空

### 取得保存元素的个数

每一个元素都只有一个根节点，没完成一次数据的添加，就可以用一个count属性来自加。

public int size(){

return this.count; //返回的就是链表的长度

}

本程序中，空不会被保存，故只有两个。

### 判断是否为空链表

第一：判断root有对象（是否为null）；

第二：判断保存的数据量（count）.

以上两种方法皆可以判断书否为空链表。

public boolean isEmpty(){

return this.count==0;

}

本次是一个链表内部的剖析。

### 内容查询

以String为例，利用A.equals()函数,从头开始查询

因为基础数据时String类，所以用equals方法，如果传递的是一个对象，那么就需要定义一个对象的比较方法（暂时把方法名称定义为compare）

### 根据索引取得数据

链表中保存了多个String类对象，在程序里面只有数组可以保存多个对象，所以说我们的链表可以说成是一个动态对象数组，所以必须要有取得元素的功能。

由于是动态数组，所以每一个元素的索引内容都一定是动态生成的。

所以我们可以在Link中增加一个foot的属性，表示每一个Node元素的编号，当然，此编号是自动生成的。

private int foot=0;

在查询的时候，（一个链表可能查询多次），那么foot应该在每一次查询时都从头开始

前提：如果有内容，或着是要查询的索引小雨个数。在Node类里面实现getNode()方法，内部类和外部类之间可以方便的进行私有属性的转换

### 修改链表数据

修改数据，只需查询满足索引值的数据，然后返回，然后重新赋值即可，在Link类里增加set()方法

### 删除指定节点的数据

外部类：

public void remove(String data) {

if (this.contains(data))

{

//要删除数据是否是根节点的数据

//root是Node类的对象，此处直接访问了内部类私有操作

if (data.equals(this.root.data))

{

this.root=this.root.next; //空出当前根节点

}else{ //不是根元素

this.root.next.removeNode(this.root,data);

}

this.count--;

}

}

内部类：

//第一次调用（Link）,previous=Link.root、this=Link.root.next

//第二次调用（Node），previous=Link.root.next、this=Link.root.next.next

//要传递上一个节点以及要删除的数据

public void removeNode(Node previous,String data){

if (data.equals(this.data)) //当前空节点为要删除节点

{

previous.next=this.next;//空出当前节点

}else{ //应该向后继续查询

this.next.removeNode(this,data);

}

}

### 将链表以对象数组的形式返回

第一步、开辟一个数组，数组长度以count为标准

利用foot来确定，依次存放数据

Link类方法最终一定要返回一个对象数组，且要被Link类操作

建议定义在Link类里面，这样内部类可以访问，外部类也可以访问。

在Node里面处理数组，数据的保存

# 继承：

Class 子类 extends 父类{}

这样就完成了继承，

class Person

{

private String name;

private int age;

public void setName(String name)

{

this.name=name;

}

public void setAge(int age){

this.age=age;

}

public String getName(){

return this.name;

}

public int getAge(){

return this.age;

}

}

class Student extends Person

{

}

public class Hello

{

public static void main(String args[]){

Student std1=new Student();

std1.setName("我");

std1.setAge(18);

System.out.println(std1.getAge());

}

}

## JAVA中没有多继承，

例如：

class A{}

class B{}

classC extendes A,B{}

只能逐渐继承

多重继承这样写：

class A{}

class B extends A{}

class C extends B{}

### 多重继承没有层数得限制

但是不要超过3层，建议限制在3层以内

子类在继承父类得到的是全部内容

公有、私有也被继承

在子类对象构造之前，一定会默认调用父类得构造（默认使用无参构造）

class A{}

class B extends A{}

class C extends B{}

注：class c中一定会调用前两个类得构造方法

super(); 父类中有无参构造时，加不加没什么区别

诺无无参构造，就必须用super();来调用父类中的有参构造

this();就是调用无参构造

super()；与this();不能同时存在。

若父类中有构造函数，则子类中的构造要加上super();在开头

## 解释：

**总结1：构造函数不能继承，只是调用而已。**

**如果父类没有无参构造函数**

**创建子类时，不能编译，除非在构造函数代码体中第一行，必须是第一行显示调用父类有参构造函数**

**如下：**

**SonClass (){**

**super(777);//显示调用父类有参构造函数**

**System.out.println(66);**

**}**

**如果不显示调用父类有参构造函数，系统会默认调用父类无参构造函数super();**

**但是父类中没有无参构造函数，那它不是不能调用了。所以编译就无法通过了。**

总结2：创建有参构造函数后，系统就不再有默认无参构造函数。

如果没有任何构造函数，系统会默认有一个无参构造函数。

# 覆写：

## 方法得覆写（重点）：

以被动为原则

如果发现父类中的方法名称功能不足（不适合于本子类），但是又必须使用这方法名称的时候，就可以利用到方法的覆写。

但是如果要想更好的实现覆写的操作，还必须考虑到权限问题，被子类覆写的方法不能够拥有比父类更严格的访问控制权限

### 访问控制权限：public>default>private ，默认是default权限

就是说如果父类的方法使用的是public声明，那么子类覆写此方法的时候只能是public声明，如果父类的方法是default的话，那么子类覆写此方法的时候，只能使用default或者public。99%的情况下只要是方法都用public 就行

class A

{

public void fun(){

System.out.println("A类中的fun()方法");

}

}

class B extends A

{

public void fun(){ //方法覆写

System.out.println("B类中的fun()方法");

}

}

public class Hello

{

public static void main(String args[]){

B b=new B();

b.fun();

}

}

发生覆写的时候，会调用实例化子类中的方法

分析要素：

观察实例化的是哪个类；

观察这个实例化的类里面调用的方法时候已经

重载，覆写，返回回值必须是一模一样的

一旦有了覆写之后会发现，默认情况下子类能够调用的一定是被覆写过的方法。为了能够明确的由子类调用父类中已经被覆写的方法，那么可以使用“super.方法（）”来访问

### 关于super.方法（）；与this.方法（）；的区别

使用“this.方法()”会首先查找本类中是否存在要调用方法名称，如果存在则直接调用，如果不存在，则查找父类中是否具备有此方法，如果有就调用，如果没有，就会发生编译时的错误

使用“super.方法（）”,明确的表示调用的不是子类方法（不查找子类中是否存在），而是直接调用父类中的指定方法。

|  |  |  |  |
| --- | --- | --- | --- |
| No: | 区别 | this | super |
| 1 | 功能 | 调用本类构造、本类方法、本类属性 | 子类调用父类构造、父类方法、父类属性 |
| 2 | 形式 | 先查找本类中是否存在有指定的调用结构，如果有则直接调用，如果没有则直接调用父类定义 | 不查找子类，直接调用父类操作。 |
| 3 | 特殊 | 表示本类的当前对象 | - |

在以后的开发之中，对于本类或者父类的操作中，强烈建议加上“this.”或者是“super.”，这样好区分。

### 面试题：

请解释重载与覆写的区别？（请解释Overloading与Overrid的区别？）

|  |  |  |  |
| --- | --- | --- | --- |
| No: | 区别 | 重载 | 覆写 |
| 1 | 英文单词 | Overlodaing | Overrid |
| 2 | 发生范围 | 发生在一个类里面 | 发生在继承关系中 |
| 3 | 定义 | 方法名称相同、参数的类型及个数相同 | 方法名称相同、参数的类型，个数相同、方法返回值相同 |
| 4 | 权限 | 没有权限的限制 | 被覆写的方法不能拥有比父类更为严格的访问控制权限 |

在发生重载关系的时候，返回值可以不同，但是考虑到程序设计的统一性，尽量保证方法的返回值相同。

## 属性的覆写：

一直强调，在任何的开发之中，类中的属性必须封装（99%），那么一旦封装之后，这样的覆盖是没有任何意义的，因为父类定义的私有属性子类根本就看不见，更不会互相影响了。

# Final关键字：

在Java之中，final称为终结器，在JAVA里面可以使用final定义类、方法以及属性。

使用final定义的类不能再有子类

范例：错误的范例

final class A{ //太监类

}

class B extends A{ //错误的

}

String中就包含有final，我们一开始说String的时候，就说String 其实是一个类。

一般在进行一些系统类的时候会使用到final，如果你要进行一些架构代码的开发，也会用到final，但是使用者的话，很少看到final定义的类。

使用final定义的方法不能被子类所覆写

在一些时候，由于父类中的某些方法具备某些隐藏的特性，那么并且子类必须使用此方法操作的时候，就加上final。

意思是：子类不要去破坏这个方法的重要作用。

### 第一种final的使用：

见上。

### 第二种final的使用：

class A

{

public final void fun(){} //注意，返回值必须跟在函数前面

}

class B extends A

{

public void fun(){} //错误的

}

public class Hello

{

public static void main(String args[]){

}

}

使用final定义的变量，常量必须在定义的时候设置好内容。并且不能修改

不管怎么调用，都不能修改。

### 注：为了让常量与变量进行有效的区分，所有的常量必须用大写。可以用下划线来分割单词

全局变量：public static final 数据类型 名字=xxx；

static的数据保存在公共数据区，所以此处的常量就是一个公共常量。

全局常量的字母必须全部大写！！！认为规定。

# 对象的多态性：

方法的多态性：

|-方法的重载：同一个方法名称，会根据传入参数的类型及个数不同执行不同的方法体。

|-方法的覆写：同一个方法，会根据子类的不同，实现不同的功能。

对象的多态性：指的是发生在继承关系类之中，子类和父类之间的关系转换问题

|-向上转型（自动完成）：父类 父类对象=子类实例；

|-向下转型（强制完成）：子类 子类对象=（子类）父类实例；

父类描述的一定是更大的范围，大范围向小范围转换一定是强制转换

例子：

class A

{

public void print(){

System.out.println("A");

}

}

class B extends A

{

public void print(){

System.out.println("B");

}

}

class C extends A

{

public void print(){

System.out.println("C");

}

}

public class Hello

{

public static void main(String args[]){

//只要是A类的子类都可以使用向上转型，都可以由A类接受

A a1=new B(); //向上转型

A a2=new C(); //向上转型

a1.print();

a2.print();

}

}

## 向上转型：就是瘦身——子类向父类转型

可以完成参数统一，还可以调用子类覆写后的方法体，即：同一个方法，针对于不同的子类可以有不同的实现。所有的父类发生了向上转型之后，只能够看见父类定义的全部方法信息，但是看不见子类特殊的方法。

## 向下转型：就是增肥——父类向子类转型，功能增多等

将父类对象转换为子类对象，这样就可以使用子类的特殊功能了

class A

{

public void print(){

System.out.println("A");

}

}

class B extends A

{

public void print(){

System.out.println("B");

}

public void funB(){

System.out.println("B扩充");

}

}

public class Hello

{

public static void main(String args[]){

A a=new B();

B b=(B) a;

b.funB();

}

}

以前开发说：

第一步：设置数据（保存数据），最需要的就是参数的统一功能。

第二部：取出数据；

## 对于对象的转型，给出以下的经验总结：

80%的情况下都只会使用向上转型，因为可以得到参数的统一，方便便于我们的程序设计；

|-子类定义的方法大部分情况如下，以父类的方法名称为标准进行覆写，不要过多的扩充方法

5%的情况下会使用向下转型，目的是调用子类的特殊方法；

15%的情况下是不转型，例如：String。

个性化的操作在一个标准的开发之中应该尽量少出现，因为对象的转型操作存在较大的安全隐患。

### 错误范例：

class A

{

public void print(){

System.out.println("A");

}

}

class B extends A

{

public void print(){

System.out.println("B");

}

}

public class Hello

{

public static void main(String args[]){

A a=new A();

B b=(B) a;

b.print();

}

}

Exception in thread "main" java.lang.ClassCastException: A cannot be cast to B

此时出现的异常表示的是类转换异常，指的是两个没有关系的类对象强制发生向下转型时所带来的异常。所以说向下转型是有风险的.

为保证转型的顺利进行，在JAVA里提供有一个关键字：instanceof,

此对象的使用：对象 instanceof 类 返回布尔型。、

如果某个对象是某各类的实例，那么久返回true，否则返回false。可以利用if语句来加以避免操作的失误。

# 抽象类：

抽象类必须有子类，即：每一个抽象类一定要被子类所继承；

抽象类的子类（子类不是抽象类）必须要覆写 抽象类之中的全部抽象方法（强制子类覆写）

抽象类的对象实例化需要依靠子类来完成，必须采用向上转型的方式处理

abstract class A{} //定义一个抽象类

抽象类是无法实例化的。

抽象类包含有，普通方法，抽象方法等。

一个子类只能够继承一个抽象类，属于单继承局限

调用方法的使用的是被覆写子类中的方法，

虽然一个子类可以去继承任意一个普通类，但是从开发的实际要求来讲普通类不要去继承另外一个普通类，只能继承抽象类。

## 注意：

抽象类里面会出现一些属性，那么抽象类之中一定会存在构造方法，目的：为属性初始化，且子类对象实例化的时候，依旧会先执行父类中的构造，在调用子类的构造。

抽象类不能用final来声明，用final声明了以后就没有子类了。

外部抽象类不允许使用static声明，而内部的抽象类允许使用static声明，用static声明的内部抽象类，就相当于 一个外部抽象类，继承的时候使用“外部类.内部类”的格式。

任何情况下，如果要执行类中的static方法的时候，都可以在没有对象的情况下直接调用。

## 巧用：

abstract class A

//static abstract定义的内部类属于外部类

{

public abstract void print();

}

class X extends A.B

{

public void print(){

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

}

}

public class Hello

{

public static void main(String args[]){

A.B ab=new X(); //向上转型

ab.print();

}

}

执行static方法的时候可以用 类名称.static方法（）；即可调用

我们可以在抽象类中定义子类，为用户隐藏子类，但依旧可以使用

abstract class A

{

public abstract void print();

private static class B extends A{ //内部抽象类子类

public abstract void print(){

System.out.println("\*\*\*\*\*\*");//覆写方法

}

}

//此方法不受实例化对象的控制

public static A getInstance(){

return new B();

}

}

public class Hello

{

public static void main(String args[]){

A a=A.getInstance();

a.print();

}

}

构造方法是整个构造过程的最后一步，所有属性设置的内容都是在构造完成之后，才会被设置上去的。

abstract class A

{

public A(){ //2、父类构造方法

this.print(); //3、调用print()方法

}

public abstract void print();

}

class B extends A

{

private int num=100;

public B(int num){

this.num=num;

}

public void print(){ //4、调用覆写方法

//num还没初始化，内容是其构造的初值。

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

}

}

public class Hello

{

public static void main(String args[]){

new B(30); //1、执行构造

}

}

在任何一个类的构造执行完之前，所有属性的内容都是其对应数据类型的初值，而子类构造执行之前，先执行父类构造，所以上述的num为最初始的值。

# 抽象方法：

[java](https://baike.baidu.com/item/java/85979" \t "_blank)中的抽象方法就是以abstract修饰的方法，这种方法只声明返回的数据类型、方法名称和所需的参数，没有方法体，也就是说抽象方法只需要声明而不需要实现。即：方法后面没有{}。定义的抽象方法意味着该方法必定会被覆写。

## 总结：

如果你真的要使用类继承的话，那么就使用抽象类（20%的情况下可能这样使用）；

抽象类强制规定了子类必须要做的事情，而且可以与抽象类的普通方法相配合；

不管抽象类如何努力都有一个天生的最大问题：单继承局限。

# 接口：

可以不使用抽象类，但是没有不使用接口的。

可以说不会接口，就不会JAVA。 接口也是抽象的，无法实例化。

## 接口的基本定义：

如果一个类之中只是由抽象方法和全局常量所组成的，那么这种情况下我们不会将其定义为抽象类，而是将其定义为接口，所谓的额接口严格来讲就属于一个特殊的类，而且这个类里面只有抽象方法与全局常量，连构造都没有。

接口中的方法都是public，写不写都是public

## 接口定义的关键字：interface、并使用implements来实现多个子类

实例：

interface A // 定义接口

{

public static final String MSG="Hello";

public abstract void print(); //抽象方法

}

由于接口里面存在有抽象方法，所以接口对象不能直接使用关键字new，进行实例化的操作，所以接口的使用原则如下：

·接口必须要有子类，但是此时一个子类可以使用implements关键字实现多个接口；

·接口的子类（如果不是抽象类），必须要覆写接口中的全部抽象方法；

·接口对象可以利用子类对象的向上转型进行实例化操作。

上面也是接口为什么使用向上转型的原因

## 注：子类可以实现多个接口，但是只能继承一个抽象类。

多个接口实例：

interface A // 定义接口

{

public static final String MSG="Hello";

public abstract void print(); //抽象方法

}

interface B

{

public void abstract void get();

}

class X implements A,B

{

public void print(){

System.out.println("A接口的抽象方法");

}

public void get(){

System.out.println("B接口的抽象方法");

}

}

public class Hello

{

public static void main(String args[]){

X x=new X(); //实例化子类对象

A a=x; //向上转型

B b=x; //向上转型

a.print();

b.get();

}

}

以上可以改为

A a=new X();

B b=(B) a;

b.get();

在定义结构上来讲A和B接口没有任何的直接联系，但是这两个接口却是同时拥有一个子类：X子类，千万不要被他打得类型和名称所迷惑，最终实例化的是X子类，而这个子类属于B类的对象，所以以上代码就可以行的通，只不过从代码的编写上来讲，并不是很好.

但是要知道一点，对于子类而言，除了接口之外，还可能回去继承抽象类，所以说一个子类又要继承抽象类又要去继承接口的话，先用extends继承，然后使用implements实现。

例如：

interface A // 定义接口

{

public static final String MSG="Hello";

public abstract void print(); //抽象方法

}

interface B

{

public abstract void get();

}

abstract class C

{

public abstract void change();

}

class X extends C implements A,B //先继承C在，定义多个接口

{

public void print(){

System.out.println("A接口的抽象方法");

}

public void get(){

System.out.println("B接口的抽象方法");

}

public void change(){

System.out.println("C类的抽象方法！");

}

}

对接口而言，发现里面的组成就是抽象方法和全局变量，所以很多的时候也有一些人为了省略编写，可以不用写上abstract或public static final,并且在方法上是否编写public结果都是一样的，因为在接口里面只能使用一种访问权限——public。一下两个结构的定义最终效果是完全相同的：

在街口里面没有写上public，其最终的访问权限也是public，绝对不是default。但是在下面定义接口的时候一定要加上public 。

所以强烈建议在接口定义方法的时候一定要写上public。

interface A // 定义接口

{

public static final String MSG="Hello";

public abstract void print(); //抽象方法

}

上述如同：

interface A // 定义接口

{

String MSG="Hello";

void print(); //抽象方法

}

对于接口的额组成99%的情况下都是以抽象方法组成为主，很少有借口只是单纯的去定义全局常量。

注意：

一个抽象类可以继承多个抽象类，但是反过来，一个借口可以使用extends关键字同时继承多个接口（接口不能继承抽象类）。

interface A // 定义接口

{

public void funA();

}

interface B

{

public void funB();

}

interface C extends A,B

{

public void funC();

}

class X implements C //C接口同时继承了A和B两个父接口

{

public void funA(){}

public void funB(){}

public void funC(){}

}

在继承关系上讲抽象类的限制要比接口多了太多：

·一个抽象类只能够继承一个抽象的父类，而接口没有这个这个限制；

·一个子类里面只能继承一个抽象类，而却可以实现多个接口。

在整个JAVA里面，接口的主要功能是解决单继承局限问题。

虽然概念上来讲接口只能由抽象方法和全局常量所组成，但所有的内部结构是不受这些要求限制的，也就是说在接口里面可以定义普通类、抽象内部类、内部接口。

## 实例：在接口里面定义抽象类

interface A // 定义接口

{

public void funA(); //定义接口时候可以不用加上abstract

abstract class B

{

public abstract void funB(); //但是在抽象类里必须加上

}

}

class X implements A //X实现了A接口

{

//这样是没有错误的，接口A中的内部类搁哪里不动，他是不影响的，他是一个独立的class文件

public void funA(){

System.out.println("Hello Word");

}

//若要想使用抽象类B可以定义一个内部抽象类

class Y extends B //内部抽象类的子类

{

public void funB(){}

}

}

## 实例：在一个接口里面利用了static去定义一个内部接口表示是一个外部接口。（用static声明的类也就是外部类）

interface A // 定义接口

{

public void funA();

static interface B

{

public void funB();

}

}

class X implement A.B

{

public void funB(){}

}

public class Hello

{

public static void main(String args[]){

}

}

在今后的开发中，若看到“A.B”形式的表示B为内部类

## 接口的三大核心作用：

* 定义不同层之间的操作标准
* 表示一种操作能力
* 表示将服务器端的远程方法师徒暴露给客户端。

# 接口的标准：

所有的代码如果要进行开发，一定要首先开发出“USB接口标准”，因为有了标准后，电脑才可以去使用这一标准。

实例：

//标准可以连接不通层的操作类

interface USB //定义标准的一定就是接口

{

public void start();

public void stop();

}

class Computer

{

public void plugin(USB usb){ //插入USB

usb.start(); //固定操作

usb.stop();

}

}

class Flash implements USB

{

public void start(){

System.out.println("U盘开始工作");

}

public void stop(){

System.out.println("U盘停止工作");

}

}

class Print implements USB

{

public void start(){

System.out.println("打印机开始工作");

}

public void stop(){

System.out.println("打印机停止工作");

}

}

public class Hello

{

public static void main(String args[]){

Computer com=new Computer();

com.plugin(new Flash());

com.plugin(new Print());

}

}

这样可以很好的描述出现实的关系

### 在程序里面，标准都是由接口定义的。

# 工厂设计模式：

如果要想确认一个接口真的好，有几个标准

·客户端调用简单，不需要关注具体的细节；

·客户端之外的代码修改，不影响用户的使用，即：用户可以不用去关心代码是否变更。

interface Fruit //接口水果，标准

{

public void eat(); //抽象函数

}

class Apple implements Fruit //继承接口

{

public void eat(){

System.out.println("\*\*\*\* 吃水果")

}

}

public class TestDemo

{

public static void main(String args[]){

Fruit f=new Apple(); //new Apple()提供了子类实例。向上转型，即瘦身

f.eat()；

}

}

如上述程序，一个水果肯定不可能一个子类，有可能有橙子类、西瓜类等。若要进行其他类的调用，就要在new一个对象出来，这样显得略有些复杂，每次在客户端上产生了实例化对象，每一次想要更换对象，都需要修改客户端的执行代码，这样的做法明显是不好打额。我们只需要知道我们要拿到的水果，在进行吃的动作，不关心水果怎么来的，耦合度太高（相当于A一直要与B绑定在一起）。这时候就需要**工厂设计模式（相当于增加了一个过渡）**。

工厂中方法可以用static来定义，因为我们在Factory中不打算写属性。

## 注：

1. 、但凡类中有static方法，如果都是由单独的static方法组成，那么可以说这个类里面不需要定义属性。②、就是之前抽象类的情况，可能需要取得实例化对象。

## 工厂类实例：

class Factory

{

public static Fruit getInstance(String className){

if("apple".equals(className)){

return new Apple();

}else if ("orange".equals(className))

{

return new Orange();

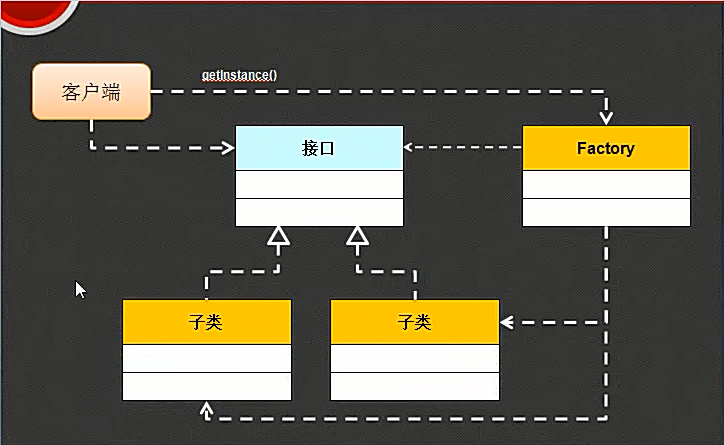
}else{

return null;

}

}

}

现在的客户端不会看见具体的子类，因为所有的接口对象都是通过Factory类取得，如果之后要扩充Fruit对象，则只需修改Fruit类对象即可

# 代理设计模式：（Proxy程序）

代理设计模式的核心精髓就在于有一个主题操作接口（可能有多种方法），核心业务主题只完成核心功能，例如：吃饭，而代理主题完成所有与核心主题有关的辅助性操作。真真实主题类只完成核心业务，代理主题类完成所有与核心业务有关的操作

interface Subject //整个操作的核心主题

{

public void make(); //核心主题的核心功能

}

class RealSubject implements Subject //定义了一个主题类

{

public void make(){

System.out.println("吃东西");

}

}

class ProxySubject implements Subject

//虽说是代理主题类，但是他依旧是子类，一样需要覆写方法

{

private Subject subject;

//要接收一个真实主题的操作对象

public ProxySubject(Subject subject){

this.subject=subject;

}

public void prepare(){

System.out.println("为主题类的核心操作做准备");

}

public void make(){

this.prepare();

//通过这里调用了主题类的功能，虽说是覆写了主题类的函数功能，但依旧完成了调用

//向上转型，覆写之后的函数功能为子类中的功能。

this.subject.make();

this.destory();

}

public void destory(){

System.out.println("做最后的结束工作");

}

}

public class TestDemo

{

public static void main(String args[]){

Subject sub=new ProxySubject(new RealSubject());

sub.make();

}

}

# 抽象类和接口的区别：

|  |  |  |  |
| --- | --- | --- | --- |
| No: | 区别 | 抽象类 | 接口 |
| 1 | 关键字 | abstract class | Interface |
| 2 | 组成 | 构造方法、普通方法、抽象方法、static方法、常量、变量 | 抽象方法、全局常量 |
| 3 | 子类使用 | class子类 extends 抽象类 | Class子类implements 接口、接口 |
| 4 | 关系 | 抽象类可以实现多个接口 | 接口不能继承抽象类、却可以继承多个父接口 |
| 5 | 权限 | 可以使用各种权限 | 只能够使用public权限 |
| 6 | 限制 | 单继承局限 | 没有单继承局限 |
| 7 | 子类 | 抽象类和接口都必须有子类，子类必须要覆写父类的全部抽象方 | |
| 8 | 实例化对象 | 依靠子类对象向上转型进行对象的实例化 | |

经过比较可以发现，抽象类中支持的功能绝对要比接口更多，但是只有一点不好，抽象类有单继承局限，但是就这一点，就足以掩盖所有的优点，即：当抽象类 喝酒口都可以使用的时候。优先考虑接口。

可以说：

·在进行某些公共操作的时候一定要定义出接口；

·有了接口就需要利用自雷完善方法；

·如果是你们自己写的接口，那么绝对不要去使用关键字new直接实例化对象，利用Fctory工厂类来完成。

# 接口的增强：

在JDK1.8以后就可以在接口中定义普通方法，必须明确的使用default来定义普通方法。Static也可以定义。利用default和static的原因是避免子类重复实现同样的代码。

接口的使用还是要以抽象概念为主

# Object类：

Object类是所有类的父类；

class Book{} 和 class Book{} extends Object

是一模一样的，在整个JAVA里面累的继承关系一直都存在（除了Object类的

## Object类的作用：

利用Object类可以接接收全部类的对象，因为可以向上自动转型。

class Book extends Object //写不写都一样，因Object类是所有对象的父类

{

}

public class TestDemo

{

public static void main(String args[]){

Object obja=new Book(); //向上转型

Object objb="Helli"; //向上转型

Book b=(Book) obja; //向下转型

String s=(String)objb; //向下转型

}

}

在不去定参数类型的时候，使用Object是最好的。

注：在Object类里面要定义一个无参构造方法。

既然Object类是所有类的父类，那么所有类对象实例化的时候，子类方法一定会默认调用父类的构造方法。

·既然Object是所有类的父类，那么所有累对象实例化的时候，都应该覆写Object类的方法，但是从严格意义上来讲，一般不遵守。

但是一个简单JAVA类都应该覆写如下方法：

·取得对象信息：public String toString();

·对象比较：public bolean equals(Object obj); //注意，传的是Object类对象

如果接收的是Object那么一定要向下转型后再输出。

·取得对象HASH码：puubic int hashCode()。

## Object类的常用方法：

### ·取得对象信息：toString();

其实就是输出。

直接输出兑现个调用toString输出对象的方法是完全一样的。在对象输出的时候，输出操作会自动的调用对象中的toString方法，将对象变为字符串方法进行输出，Object类的toString()为了适应于所有的对象的输出，所以只输出了对象的编码。 如果有需要可以进行toString方法的覆写。

以后的时候直接输出对象就调用toString();这样等于数处对象的时候代码又节约了。

### 对象比较：equals();

·对象比较在很多的开发中是一定要用到的核心概念，之前使用的一个compare的比较方法的名称，但是这个不标准，标准的做法是使用equals();方法来完成。最常用

## Object类的接受引用数据类型：

Object类是所有类的父类，所以Object类的对象可以接受所有类的对象，可是除了所有类的对象之外，Object类连对象数组和接口都可以接收。

public class TestDemo

{

public static void main(String args[]){

Object obj=new int[]{1,2,3}; //向上转型，静态初始化

System.out.println(obj); //[I@15db9742 这是直接输出数组的结果，以后只要看到有[开头的地址就是数组的

if (obj instanceof int[]) //是否是整型数组

{

int data[]=(int[]) obj; //向下转型

for (int x=0;x<data.length ;x++ )

{

System.out.println(data[x]);

}

}

}}

Object什么参数都能接收

### Object类接收接口：

interface A

{

public void fun();

}

class B extends Object implements A

{

public void fun(){

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

}

}

public class TestDemo

{

public static void main(String args[]){

A a=new B(); //接口对象

Object obj=a; //接收接口对象

A t=(A) obj; //向下转型

t.fun();

}

}

## 利用Object类修改链表：

见实验，

# 构造防范方法私有化： 构造方法前加prviate

·单例设计模式（Singleton）

**package** cn.test.deom;

**class** Singleton{

**private** **static** Singleton *instance*=**new** Singleton();

**public** **static** Singleton getInstance() {

**return** *instance*;

}

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

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

}

}

**public** **class** TestDemo {

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

Singleton s=**null**; //声明对象

s=Singleton.*getInstance*(); //直接访问static函数

s.print();

}

}

代码的意义：

如果说现在想要控制一个类中实例化对象产生的个数，那么首先要锁定的是就是类中的构造方法，因为在实例化任何新对象都要使用构造方法，如果构造方法被锁了，那么就无法产生新的实例化对象。

可是既然需要一个实例化对象，那么就可以在类的内部使用static方式来定义一个公共的对象，并且没一次通过static方法返回唯一的一个对象，这样外部不管有多少次调用，那么最终的一个类只能产生唯一的一个对象，这样的设计就属于单类设计模式（Singleto）。

最完美的形式：

**package** cn.test.deom;

**class** Singleton{

**private** **static** **final** Singleton ***INSTANCE***=**new** Singleton();

**public** **static** Singleton getInstance() {

**return** ***INSTANCE***;

}

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

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

}

}

**public** **class** TestDemo {

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

Singleton s=**null**; //声明对象

s=Singleton.*getInstance*(); //直接访问static函数

s.print();

}

}

程序特点：构造方法私有化，在类的内部定义static属性与方法，利用static方法取得本类的实例化对象。那么不管外部产生多少个Singleton对象，但是本质上永远只有唯一的一个实例化对象。

可是对于单例设计模式有两种形式：饿汉式、懒汉式。

在之前所编写的单例实际上就属于饿汉式的应用，在Singleton类定义的时候就已经准备好了一个Singleton类的实例化对象INSTANCE，而现在并不关心这个对象是否使用。

而懒汉式的最大特点就是在第一次使用的时候才进行实例化操作。

**package** cn.test.deom;

**class** Singleton{

**private** **static** **final** Singleton ***instance***;

**private** Singleton() {} //构造方法私有化

**public** **static** Singleton getInstance() {

**if**(***instance***==**null**) {

***instance*** = **new** Singleton();

}

**return** ***instance***;

}

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

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

}

}

# 多例设计模式：

就是一个类中产生一个对象。

例如：现在要定义一个表示一周时间的类，这个类只能够取七个对象。

例如：现在要求定义有个别表示性别的类，这个类只能够取两个对象。

**package** cn.test.deom;

**class** Sex{

**private** String title;

**private** **static** **final** Sex ***MALE***=**new** Sex("男");

**private** **static** **final** Sex ***FEMALE***=**new** Sex("女");

**private** Sex(String title) { //构造函数私有化

**this**.title=title;

}

**public** String toString() {

**return** **this**.title;

}

**public** **static** Sex getInstance(**int** ch) {

**switch**(ch) {

**case** 1:

**return** ***MALE***;

**case** 2:

**return** ***FEMALE***;

**default**:

**return** **null**;

}

}

}

**public** **class** TestDemo {

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

Sex sex=Sex.*getInstance*(2);

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

}

}

# 类图：

利用PowerDesigner可以描述出类的结构图与时序图。

如果是一个项目开发设计书上，一定提供有UML图。

·第一层：类名称，如果是抽象类使用斜体字；

·第二层：描述类中的属性，对于属性肯定要封装使用“-”表示。

|-public(+)、protected(#)‘private(-);

第三层：类中定义的方法。

# 时序图：UML Sequence Diagram

用于描述每个步骤进行性

# 可变参数：

**package** cn.test.deom;

**public** **class** TestDemo {

//可变参数支持接收数组

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

System.***out***.println(*add*(**new** **int**[] {1,2,3})); //传递3个整型数据

System.***out***.println(*add*(**new** **int**[] {10,20}));//传递两个参数

//或者使用“，”区分不同的参数，接收的时候还是数组

System.***out***.println(*add*(1,2,3));

System.***out***.println(*add*(10,20));

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

}

/\*\*

\* 实现任意整型数据的相加操作

\* **@param** data 由于要接收多个整型数据，所以使用数组完成接收

\* **@return** 多个整型数据累加的结果

\*/

**public** **static** **int** add(**int** ... data) {

**int** sum=0;

**for**(**int** x=0;x<data.length;x++) {

sum+=data[x];

}

**return** sum;

}

}

主要是上面data哪里，要着重看一下

# 异常处理：

try…catch、try…catch…finally、try…finally

·出现异常的目的是为了解决异常，发现也可以正常的执行完毕。

public class TestDemo

{

public static void main(String args[]){

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

try

{

System.out.println("10/0");

System.out.println("1########");

}

catch (ArithmeticException e) //计算异常

{

System.out.println("异常错误");

}

}

}

注意：catch异常后，后面要加上一个e 。规定

## 异常处理的方法：

·printStackTrace（）的方行异常的输出；

程序继续执行，但是返回的是完整的的“异常信息”

## Finally的特性：

public class TestDemo

{

public static void main(String args[]){

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

try

{

System.out.println("10/0");

System.out.println("1########");

}

catch (ArithmeticException e) //计算异常

{

System.out.println("异常错误");

}finally{

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

}

}

}

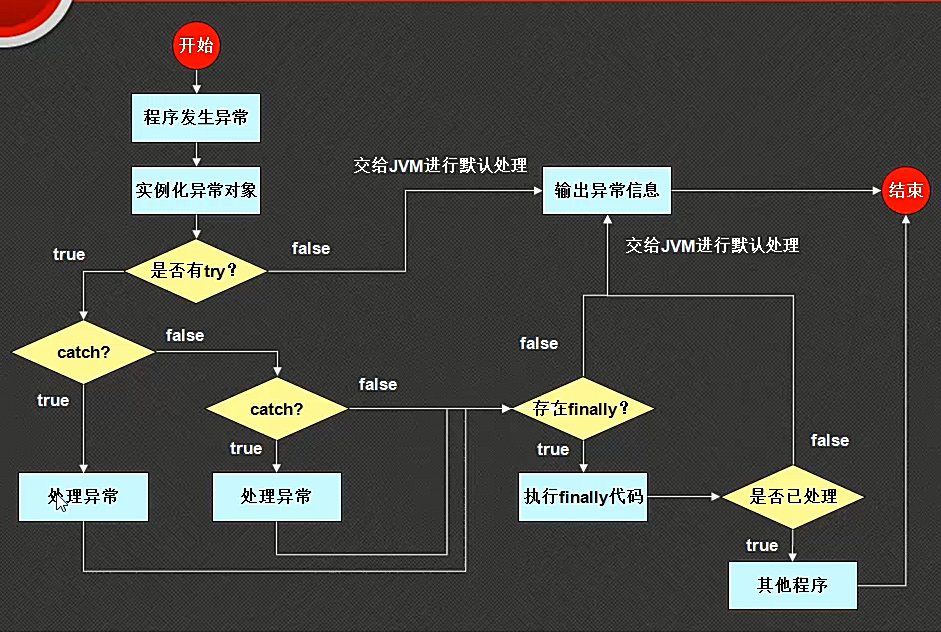
一个catch只能处理一个异常，所以可以设置多个catch。

## 异常的处理流程：（核心）

## Error和Exception的区别：

Error是JVM错误，即：此时的程序还没有执行，如果没有执行用户无法处理；

Exception:指的得是程序运行中产生的异常，用户可以处理。

也就是所谓的异常处理指的就是所有Exception以及它的子类异常。

·当程序在运行的过程中出现了异常后，那么会由JVM自动根据异常的类型实例化一个与之类型匹配的异常类对象。（系统会自动new一个 ）。

·产生了异常对象过之后会判断当前的语句上是否存在有异常处理，如果没有，那么就交给JVM进行默认的异常处理，处理的方式：输出异常信息，而后结束程序的调用。

·如果此时存在捕获操作，那么会由try语句来捕获产生的异常类实例化对象，而后与try语句后的每一个catch进行比较，如果现在有符合的捕获类型，则使用catch的语句来进行异常的处理，如果不匹配，则向下继续匹配其他的catch；

·不管最后异常处理是否能够匹配，那么都要向后执行，如果此程序中在有finally语句，那么就先执行finally类的代码，但是执行完毕后需要根据之前的catch匹配结果来决定如何执行，如果之前已经成功的捕获为了异常，那么就继续执行finally之后的代码，如果之前没有成功捕获异常，那么就将此异常交给JVM处理（输出异常信息，而后结束程序执行）

### 注：

整个过程就好像方法重载一样。根据catch后面的参数类型进行匹配，但是所有的JAVA对象都存在有自动向上转型的操作支持，也就是说，如果真的要匹配类型，简单的做法是匹配Exception就够了。

public class TestDemo

{

public static void main(String args[]){

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

try

{

System.out.println("10/0");

System.out.println("1########");

}

catch (Exception e) //计算异常

{

System.out.println("异常错误");

}finally{

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

}

}

}

见上述；

### 注意两点：

·在catch捕获异常的时候，捕获范围大的异常一定要放在捕获范围小的异常之后，否则程序编译错误；如果可以，只用Exception就好。

·虽然直接用Exception比较方便，但是这样不好，因为所有的异常都会按照同样的一种方式进行处理；一些要求的严格项目里面，异常一定要分开处理会更好。

## throws关键字:

throws关键字主要用于方法的声明上，值得是当方法之中出现异常的之后交由被调用出来处理;调用了throws方法之后，那么不管操作是否出现个异常，都必须使用try…catch来进行异常的处理。（强制要求）

在主方法上如果继续抛出了遗产个，那么这个异常将会交给JVM进行处理，也就是采用默认的处理方式，输出异常信息，而后结束程序调用。

所以不应该在主方法上加上throws因为，当出现异常后，也希望结束调用。

## throw关键字：

public class TestDemo

{

public static void main(String args[]){

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

try

{

throw new Exception("自己定义的异常");

}

catch (Exception e)

{

e.printStackTrace();

}

}

}

## throw和throws的区别：

throw：

·指的是在方法之中人为的抛出一个异常类对象（这个异常类对象可能是自己实例化或者是抛出已存在的）

throw：

·在方法的声明上使用。表示此方法在调用是必须处理异常，如果出现就处理，不出现就正常执行。

## 异常处理重要的代码模型：异常的使用格式

class MyMath

{

//此时表示div()方法上如果出现了异常交给被调用处处理。

public static int div(int x,int y) throws Exception{

int result=0;

System.out.println("\*\*\* 1、除法开始计算 \*\*\*"); //把门打开

try

{

result=x/y;

}

catch (Exception e)

{

throw e; //继续抛出异常

} finally{

System.out.println("\*\*\* 2、出发计算结束\*\*\*"); //把门关上

}

return result;

}

}

public class TestDemo

{

public static void main(String args[]){

try

{

System.out.println(MyMath.div(10,0));

}

catch (Exception e)

{

e.printStackTrace();

}

}

}

如果现在你直接使用了try...finally，那么表示你连处理异常的机会都没有了。

try {}catch{}finally，为标准形式。

## RuntimeException类：

### 观察：

public class TestDemo

{

public static void main(String args[]){

//将字符串转换为整型数据

int temp=Integer.parseInt("100");

}

}

现在来观察一下parseInt()方法的定义：

·public static int parseInt(String s)throws NumberFormatException;

刚才说，用throws抛出的异常，必须进行强制性的异常捕获，可是现在并没有强制性的异常捕获。

RuntimeException表示运行时的异常，在JAVA里面为了方便用户的代码编写，专门提供了一种RuntimeException类，这种异常类最大的特征在于：程序在编译的时候不会强制性的要求用户处理异常，用户可以根据自己的需要选择性的进行处理，但是如果没有处理，又发生异常了，将交给JVM默认处理。RuntimeException的子类异常，可以由用户选择性处理。

### 请解释Exception和RuntimeException的区别？请列举出你常见的RuntimeException类。

·Exception是RuntimeException的父类；

·使用Exception定义的异常必须要处理，而RuntimeException的异常可以选择性处理。

·常见的RuntimeException：ArithmeException、NullPointerException、ClassCastException。

## assert关键字（了解）:

又名断言，在JAVA中的作用是当程序执行到某行代码处时一定是预期的结果。默认情况下断言是不应该影响程序的运行的，也就是说java解释程序的时候，断言是默认不起作用的。

int num=10;

assert num==20:”num的内容不是20”;

启用断言是使用： -ea java文件名 来调用

## 自定义异常：

Java本身已经提供了大量的异常，但是有时候不够用，那么就要自定义异常用户开通一个自己的一个异常类，（这属于系统架构的问题）。

### 范例：

class AddException extends Exception

{

public AddException(String msg){

super(msg); //调用父类的无参构造

}

}

public class TestDemo

{

public static void main(String args[]){

//将字符串转换为整型数据

int num=20;

try

{

if (num>10) //出现了错误，应该产生异常

{

throw new AddException("数值传递的过程");

}

}

catch (Exception e)

{

e.printStackTrace();

}

}

}

这种代码只能说是介绍自定义异常的形式，但是不能说明自定义异常的实际使用。

## SUM：

·Exception的父类是throwable，所以在定义子类的时候建议不要使用throwable，因为throwable中还有一个Error。

# 包：

## 定义：

包就是一个目录； 格式如下：package cn.mldn.demo

关键字package cn主目录、然后用点隔开子包mldn再用点隔开子包demo，即作用

不要去手工去创建这些目录，应该使用命令自动生成；

·打包编译：javac-d.Hello.java;

|-“-d”:生成目录，根据package的自定义生成；

|-“.”:设置保存的路径，如果为“.”表示当前所在路径下生成；

·有了包的定义以后解释程序的时候不要进入都包里面解释程序,应该在包外输入完整的名称（java cn.mldn.demo.Hello）包.类名称。

以后所有的类一定要定义在包之中，那么完整的包名称就是：“包.类”。

## 包的导入：

使用包可以将一个完整的程序拆分为不同的文件进行分别保存，这样就会造成一个问题，不同的包之间有可能要进行互相访问，那么此时就需要导包操作，使用import语句即可完成。

import: 加上完整的类名称。即可完成导包操作。可用通配符“\*”来解决问题。

Javac -d.\*.java 然后再编译

系统常见的包：

·java.lang:包含了String、Object、Integer等类。

·java.lang.reflect：反射开发包。

·java.util:Java的工具包，提供了大量的工具类，像链表；

·java.until.regex:正则工具包；

·java.text:国际化处理程序包

·java.io:进行输入、输出处理以及文件操作；

·java.net:网络编程开发包

·java.sql：数据库程序开发包；

## Jar命令：

Jar cvf my.jar cn //其实就是打包文件

生成的my.jar文件不能够直接的使用，必须配置CLASSPATH才可以加载、

可以直接在我的电脑的环境变量设置完成。

# 认识包装类：

JAVA在设计之初有一个原则：一切皆对象，一切的操作要求用对象的形式进行描述。但是这里面就会出现一个矛盾，基本数据类型不是对象。在JAV里面专门的提供了一组包装类，来实现包装所有的数据类型。

Bye(Byte)、short(Short)、int(Integer)、long(Long)、boolean(Boolean)、char(Character)。

·对象型包装类（Object的直接子类）： Boolean、Character。

·数值型包装类（Number的直接子类）：Byte、Short、Integer、Long

Number是一个抽象类，里面一共定义了六个操作方法：

intValue()、doubleValue()、float Value()、byte Value()、short Value()、long Value()、

## 装箱与拆箱操作：

### 装箱操作：

将基本数据类型变为包装类的形式；

|-每个包装类的构造方法都可以接收各自的数据类型的变量；

### 拆箱操作：

从包装类中取出被包装的数据；

|-利用Number类中提供的一系列的：xxxValue()方法

范例：

**package** cn.test.deom;

**public** **class** TestDemo {

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

Integer obj=**new** Integer(10); //将基本的数据类型装箱

**int** temp=obj.intValue(); //将基本数据拆箱

System.***out***.println(temp\*2);

}

}

范例：对象型

**package** cn.test.deom;

**public** **class** TestDemo {

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

Boolean obj=**new** Boolean(**true**); //将基本的数据类型装箱

Boolean temp=obj.booleanValue(); //将基本数据拆箱

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

}

}

## JAVA类后期提供了自动装箱自动拆箱类；

就是说直接调用就可以了。判断内容相同的时候利用，一定是JDK1.5以后才有自动装箱等操作。

数据类型的强制转换：

一、**package** cn.test.deom;

**public** **class** TestDemo {

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

**int** num=100;

String str=num+"";//变为String

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

}

}

这样的操作虽然可以完成，但是会存在有垃圾问题。空字符串是没有用的。

二、

public static String valueOf(数据类型 变量)

利用这个方法可以完成转换，且不会产生垃圾。

**package** cn.test.deom;

**public** **class** TestDemo {

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

**int** num=100;

String str=String.*valueOf*(num);

System.***out***.println(str.replaceAll("0","9"));

}

}

# foreach循环：

JDK1.5以后才有。

## 定义：

for(类型 变量：数组|集合){

//每一次循环会自动的将数组内容设置给变量

}

## 范例：

**package** cn.test.deom;

**public** **class** TestDemo {

//可变参数支持接收数组

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

**int** data[]=**new** **int**[] {1,2,3,4,5};

**for** (**int** x : data) {

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

}

}

}

这样就避免了索引的出现。

# 静态导入：

了解就好。

# 泛型：

从JDK1.5以后开始增加了泛型技术，而泛型技术的核心意义在与：类在定义的时候，可以用过一个标记，此标记就表示类中属性或方法参数的类型，在使用的时候才动态的设置类型。

## 问题的引出：

范例:

**package** cn.test.deom;

**class** Point<T>{ //定义坐标，Type=T,是一个类型

**private** T x; //x轴坐标，具体类型动态生成

**private** T y; //y轴坐标，具体类型动态生成

**public** T getX() {

**return** x;

}

**public** **void** setX(T x) {

**this**.x = x;

}

**public** T getY() {

**return** y;

}

**public** **void** setY(T y) {

**this**.y = y;

}

}

**public** **class** TestDemo {

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

//设置数据

Point<String> p=**new** Point<String>();

p.setX("东经100度");

p.setY("南纬20度");

//设置数据

String X=(String) p.getX();

String Y=(String) p.getY();

System.***out***.println("x坐标："+X+",y坐标:"+Y);

}

}

这样就可以避免了转换为Object类，避免了向下转型，避免了安全问你，即避免了ClasscastException.

但是需要特别说明的是，如果想要使用泛型，那么他能够采用的类型不能是基本类型，只能是引用类型 ，像int就不行，但是可以用包装类Integer来表示。利用的是自动装箱设置。

对泛型有两点说明：

·如果在使用泛型或者是接口的时候，没有设置泛型具体类，那么会出现编译时的警告，同时为了保证泛型不出错，所有的泛型都将使用Object表示 。也就是说T将换会Object，下面依旧要使用向下转型。

·从JDK1.7以后实例化的泛型可以省略。

有人觉得声明两次有些啰嗦，所以后面可以省略。

## 通配符的使用：

需要有一种方式，可以接收任意的泛型类型，但是不可以修改只能够取出，那么就可以使用“?”来描述。

范例：

**package** cn.test.deom;

**class** Message<T> {

**private** T msg;

**public** **void** setMsg(T msg) {

**this**.msg=msg;

}

**public** T getMsg() {

**return** msg;

}

}

**public** **class** TestDemo {

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

Message<Integer> m1=**new** Message<Integer>();

m1.setMsg(100);

*fun*(m1); //引用传递

}

**private** **static** **void** fun(Message<?> temp) { //不能够设置，但是可以取出

System.***out***.println(temp.getMsg());

}

}

### 注意：

在“？”的通配符基础上还会有两个子的通配符：

·？extends类：设置泛型上限，可以再声明上和方法参数上使用；

|-？extends Number:意味着可以设置Number或者是Number的子类（Integer,double）

#### 操作实例：

**class** Message<T **extends** Number> {

**private** T msg;

**public** **void** setMsg(T msg) {

**this**.msg=msg;

}

**public** T getMsg() {

**return** msg;

}

}

**public** **class** TestDemo {

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

Message<Integer> m1=**new** Message<Integer>();

m1.setMsg(100);

*fun*(m1); //引用传递

}

**private** **static** **void** fun(Message<? e**xtends Number**> temp) { //不能够设置，但是可以取出

System.***out***.println(temp.getMsg());

}

}

如果设置了不是Number的子类

·？super类：设置泛型下限，方法参数上使用；

|-？super String：意味着只能够设置String或者是它的父类Object。

例子方式同上

要求看懂！

## 泛型接口：

形式一、

**package** cn.test.deom;

**interface** IMessage<T>{ //设置泛型接口

**public** **void** print(T t);

}

**class** MessageImpl<T> **implements** IMessage<T>{

**public** **void** print(T t) {

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

}

}

**public** **class** TestDemo {

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

IMessage<String> msg=**new** MessageImpl<String>();

msg.print("Hello World!");

}

}

形式二、

在子类中设置泛型，这样也决定了方法中参数的泛型。

## 泛型方法：（要求看的懂）

泛型方法不一定非要定义在支持泛型的类里面。在之前所编写的所有的泛型方法都是在泛型支持类里面定义的。

范例：尽量不要使用

**package** cn.test.deom;

**public** **class** TestDemo {

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

String str=*fun*("Hello");

System.***out***.println(str.length());

}

**public** **static** <T> T fun(T t) {

**return** t;

}

}

## 总结：

·泛型解决的是向下转型所带来的额安全隐患，其核心的组成就是在声明或接口的时候不设置参数或属性的内容

·“？”可以接受任意的泛型类型，只能够取出，但是不能够修改。

# 枚举：

简化了多例设计模式。

枚举可以实现接口

枚举里面最变态的一种做法是可以再每一个对象后面使用匿名内部类的形式实现抽象方法。

在枚举里面还可以直接定义抽象方法。此时每一个枚举对象都必须要扩充抽象方法。

一个类里面实现了普通方法、抽象方法。有点夸张

枚举可以直接在switch上使用。

Enum和enum的区别。

·enum是一个关键字，而Enum是一个抽象类；

·使用enum定义的枚举就相当于一个类继承了Enum这个抽象类。

枚举的使用有两个要求：

·枚举之中定义的构造方法不能够使用public声明；如果没有无参构造，请手工调用。因为继承了抽象类。

·枚举对象必须要放在首行，随后才可以定义属性、构造、普通方法。

枚举可以说是一个高级的多例设计模式。

# Annotation:

Annotation是JDK1.5最大的特色，利用注解的形式来实现程序的不同功能实现。

三个最为常用的基础@Override、@Deprecated、@SuppressWarnings

## @Override：准确覆写

**class** Book{

**public** String tostring() {

**return** "这是一本书";

}

}

注意上述的s, toString的S是要大写的。有可能小细节没有发现，但是编译的时候是不会报错的，但是在运行的时候会报错。

**class** Book{

@Override //这个可以自动生成

**public** String toString() {

**return** "这是一本书";

}

}

不然他会提示你覆写错误。

# 声明过期操作：@Deprecated

就是说现在有一个专门负责完成某些功能的工具包，里面有一个Hello类，Hello类里有个函数，fun（），在开发初期fun的功能十分的完善，但是后来随着开发技术的不断增强发现fun过时了，这时候对于开发者有两种选择：

·选择一、直接在新版的工具包里面取消掉fun（）的方法，同时给出新的fun2方法，

选择二、在新版本的开发包里面保存fun（）方法，但是通过某种途径告诉新的开发者，此方法有问题，并且提供fun2这个新的方法供开发者使用。

很明显第二种比较合适，因为可以兼顾到如今的使用。

范例：

**class** Book{

@Deprecated

**public** **void** ~~fun~~() {}

}

就像这样，原来的fun依旧可以使用，但是加了一个中线相当于提示。

# @SuppressWarnings //压制警告

打完代码，下面黄线的就表示警告

如果是开发者故意留下来的警告信息，但是又不希望其总是提示，就可以选择压制警告

# @Test //测试方法，可以不用在主类中测试

EG：

Public class Person{

@Test

Public void eat{

System.out.print(“跑。。。。”);

}

}

直接run可以直接测试Test方法

# Lanmda语法的三种形式：

·（参数）->单行语句；

·（参数）->{单行语句}

·（参数）->表达式。

## 单行语句：

实例：

**package** cn.test.deom;

**interface** IMessage{

**public** **void** print(String str);

}

**public** **class** TestDemo {

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

//首先要定义此表达式里面需要接受变量，单行语句直接进行输出

*fun*((s)->System.***out***.println(s));

}

**private** **static** **void** fun(IMessage msg) {

msg.print("Hello world!"); //设置参数内容

}

}

## 多行语句：

实例：

**package** cn.test.deom;

**interface** IMessage{

**public** **void** print(String str);

}

**public** **class** TestDemo {

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

//首先要定义此表达式里面需要接受变量，单行语句直接进行输出

*fun*((s)->

{

s=s.toUpperCase(); //转大写

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

});

}

**private** **static** **void** fun(IMessage msg) {

msg.print("Hello world!"); //设置参数内容

}

}

## 表达式：

如果代码里只存在一个简单的计算表达式，那么操作也可以很容易

范例：

**package** cn.test.deom;

**interface** IMessage{

**public** **int** add(**int** x,**int** y);

}

**public** **class** TestDemo {

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

//首先要定义此表达式里面需要接受变量，单行语句直接进行输出

*fun*((s1,s2)->s1+s2);

}

**private** **static** **void** fun(IMessage msg) {

System.***out***.println(msg.add(10,20));

}

}

如果要写return,可以fun((s1,s2)->{return s1+s2;});

还是在多行的时候比较好。

## 好处：

利用Lamda表达式最终解决的问题：避免了匿名内部类定义过多无用的操作。

# 方法引用：(有点类似C++中的类外定义)

对象的引用就是不同的对象可以操作同一块内容。指为一个方法设置别名，相当于一个方法定义了不同的名字。

方法的引用就是：

·相当于一个方法定义了不同的名字。

方法引用在Java8之中一共定义了四种形式：

·引用静态方法： 类名称::static 方法名称;

·引用某个对象的方法： 实例化对象::普通方法;

·引用特定类型的方法： 特定类::普通方法;

·引用构造方法： 类名称::new.

## 范例：静态方法

**package** cn.test.deom;

/\*\*

\* **@param**<P>引用方法的参数类型

\* **@param**<R>引用方法的返回类型

\*/

**interface** IMessage<P,R>{

**public** R zhuanhuan(P R);

}

**public** **class** TestDemo {

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

//即：将Stirng.valueOf()方法变味了IMessage接口里的zhuanhuan()方法

IMessage<Integer,String> msg = String::*valueOf*;

String str=msg.zhuanhuan(1000);

System.***out***.println(str.replaceAll("0","9"));

}

}

## 范例：普通方法引用：

**package** cn.test.deom;

/\*\*

\* **@param**<P>引用方法的参数类型

\* **@param**<R>引用方法的返回类型

\*/

**interface** IMessage<R>{

**public** R upper();

}

**public** **class** TestDemo {

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

//Stirng类的toUpperCase()定义：public String toUpperCase()

//这个方法没有参数，但是有返回值，并且这个方法一定要在有实例化对象的情况下才可以调用

//“hello”字符串是String类的实例化对象，所以可以直接调用toUpperCase()方法

//将toUpperCase()函数的应用交给了IMessage接口

IMessage<String> msg="hello" :: toUpperCase;

String str=msg.upper(); //相当于“"hello".toUpperCase()”

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

}

}

通过两个代码的演示应该应景发现了，如果要实现函数的引用，那么必须要有接口，而且最为关心的是，接口里面需要只存在一个方法。

## 函数式接口注解声明：

@FunctionalInterface //表示此为函数式接口，不能够定义多个方法，只能定义一个方法。

## 特定类型的引用：

**package** cn.test.deom;

@FunctionalInterface

**interface** IMessage<P>{

**public** **int** compare(P p1,P p2);

}

**public** **class** TestDemo {

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

IMessage<String> msg=String ::compareTo;

System.***out***.println(msg.compare("A","B"));

}

}

与之前相比，方法引用不需要定义对象，而是可以理解为将对象定义在了方法上。

## 引用构造方法：

**package** cn.test.deom;

@FunctionalInterface

**interface** IMessage<C>{

**public** C create(String t,**double** p);

}

**class** Book{

**public** Book(String title, **double** price) {

**super**();

**this**.title = title;

**this**.price = price;

}

@Override

**public** String toString() {

**return** "书名："+**this**.title+"，价格："+**this**.price;

}

**private** String title;

**private** **double** price;

}

**public** **class** TestDemo {

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

IMessage<Book> msg=Book :: **new**;

Book book=msg.create("JAVA",20.2);

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

}

}

虽然是create（），但是这个方法引用的是Book类的构造

对象的引用是使用不同的名字，而方法引用是需要一个函数式的接口，并且要设置好我们的参数。

# 內建的函数式接口：

看好格式

对于方法方法引用都是要有一个接口，可是不管如何操作，实际上有可能操作的接口只有四种：（在java.util.function）里面

1·功能型接口（Function）:public interface Function<T,R>{public R apply(T,t);}

|-此接口需要接收一个参数，并且返回一个处理结果；

2·消费型接口（Consumer）:public interface Consumer<T>{public void accept(T t);}

|-此接口只负责接收数据（引用数据是不需要返回），并且不返回处理结果。

3·供给型接口（Supplier）:public interface Supplier<T>{public T get();}

|-此接口不接受参数，但是可以返回结果；

4·断言型接口（Predicate）：public interface Predicate<T>{public Boolean test(T t);}

|-进行判断操作使用；

以上的4个功能型接口，存在于JDK1.8中。

## 功能型接口：

**package** cn.test.deom;

**import** java.util.function.Function;

**public** **class** TestDemo{

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

Function<String,Boolean> fun="##hello" ::startsWith;

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

}

}

消费型接口：

**package** cn.test.deom;

**import** java.util.function.Consumer;

**class** MyDemo{

**public** **void** print(String str) {

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

}

}

**public** **class** TestDemo{

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

Consumer<String> cons=**new** MyDemo() :: print ;

cons.accept("Hello World!");

}

}

# 线程与进程：

## 进程：

### 定义：

在操作系统的定义中，进程指的是一次程序的完整执行，这个运行过程之中内存、处理器、IO等资源操作都要为这个进程服务。

在同一个时间段上，会有多进程轮流去抢占资源，但是在漏一个时间点上，只会有一个进程运行。

## 线程：

是在进程的基础上进行新一步划分的结果，即：一个进程可以同时创建多个线程。

线程是比进程更快的处理单元，而且所占资源也小。那么多线程的应用，就是性能最高的应用。

### 实现：

JAVA中实现多线程有两种途径：

·继承Thread类；

·实现Runnable接口（Callable接口）；

## Thread类：

Class MyThread extends Thread{} //这就是一个多现程的操作类。

所有的方法起点是main()方法，但是所有的线程也要有一个自己的起点，就是public void run(){}方法，也是就是在多线程的主题类中都必须要覆写Thread类中所提供的run()方法。public void run(){}方法上没有返回值，也就意味着线程一旦开始，就只能一直执行，不能够返回内容。

### 范例：线程类

**package** cn.test.deom;

**class** MyThread **extends** Thread{ //这是一个多线程的类

**private** String name; //定义类中的属性

**public** MyThread(String name) {

**this**.name=name;

}

@Override

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

**for** (**int** i = 0; i < 200; i++) {

System.***out***.println(**this**.name+"-->"+i);

}

}

上述代码执行的结果是A执行完，然后B执行，然后C执行，不符合多进程。他要进行资源的抢占，应该是多个线程的彼此交替执行。

多线程的唯一启动方法是public void start(){}（调用此方法执行的方法体是run()方法定义的）。

范例：

**package** cn.test.deom;

**class** MyThread **extends** Thread{ //这是一个多线程的类

**private** String name; //定义类中的属性

**public** MyThread(String name) {

**this**.name=name;

}

@Override

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

**for** (**int** i = 0; i < 200; i++) {

System.***out***.println(**this**.name+"-->"+i);

}

}

}

**public** **class** TestDemo{

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

MyThread mt1=**new** MyThread("线程A");

MyThread mt2=**new** MyThread("线程B");

MyThread mt3=**new** MyThread("线程C");

mt1.start();

mt2.start();

mt3.start();

}

}

此时，每个线程的方法交替执行。

为什么用start()不用run()方法来调用多线程，是因为：

·使用Thread类的start()方法不仅仅要启动多线程的执行代码，还要根据不同的操作系统进行资源的分配。

## 实现Runnable接口：

虽然Thread类可以实现多继承的主体类定义，但是它有一个问题，Java局有单继承局限，正因为如此在任何情况下针对于类的继承都应该是回避的问题，那么多线程也一样，为了解决单继承局限，在JAVA里面提供了Runnable接口。多个线程要想访问同一资源，必须要使用runnable接口。

### 定义：

@FunctionalInterface //表明是函数式接口

public interface Runnable{

public void run();

}

在街口里面任何方法都是public定义的权限，不存在默认的权限。

那么只需要让一个类实现Runnable接口即可，并且也需要覆写run()方法。

所以用Thread定义的改用run方法，可以不用去动。

与继承Thread类相比，此时的MyThread类在结构上是没有区别的，但是但是有一点有严重区别的，如果此时继承了Thread类，那么可以直接继承start（）方法，但是如果实现的是Runnable接口，但是如果实现的是Runnable接口，并没有start方法可以继承。

不管何种情况下，如果要想启动多线程一定依靠Thread类完成，在Thread类里面定义有一下的构造方法：

Thread(Runnable target){}

Allocates a new Thread object

注意：**接收的是Runnable接口对象；也就是说可以接收Runnable的任意子类。**

范例：

**package** cn.test.deom;

**class** MyThread **implements** Runnable{ //这是一个多线程的类

**private** String name; //定义类中的属性

**public** MyThread(String name) {

**this**.name=name;

}

@Override

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

**for** (**int** i = 0; i < 200; i++) {

System.***out***.println(**this**.name+"-->"+i);

}

}

}

**public** **class** TestDemo{

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

MyThread mt1=**new** MyThread("线程A");

MyThread mt2=**new** MyThread("线程B");

MyThread mt3=**new** MyThread("线程C");

**new** Thread(mt1).start();

**new** Thread(mt2).start();

**new** Thread(mt3).start();

}

}

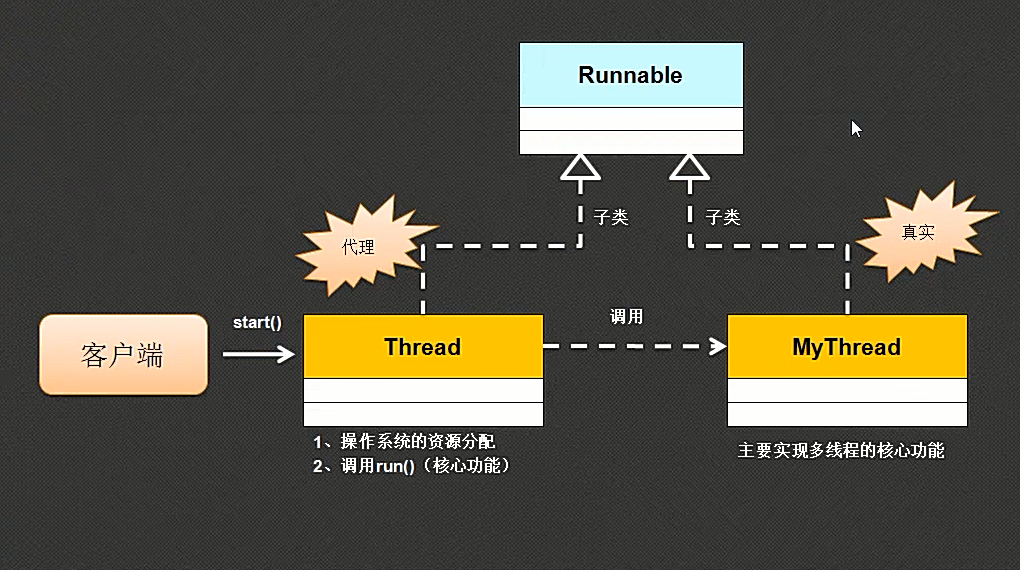
此时避免了单继承局限，那么也就是说在实际工作中，使用Runnable接口是最合适的。

## 多线程两种实现方式的区别：

·明确： Runnable接口与Thread类相比，解决了单继承的定义局限，所以不管后面的区别于联系是什么，如果要使用，就是用Runnable接口。

观察：Thread类

public class Thread extends Object implements Runnable

不仅继承了Object类，还继承了Runnable接口

有点代理设计模式的味道。

如果是带李设计模式，客户端调用的代理类的方法也应该是接口里提供的方法，那么也应该是run（）才对。

看一下像代理设计模式，但又不是。

使用Runnable接口可以比Thread类的接口更能体现出数据共享这一特点。

数据共享指的是多个线程访问统一资源的操作 。

## Callable接口：（接口）

## 定义：Interface Callable<V>

内含方法：

@FunctionalInterface

public interface Callable<V>

这个方法可以返回值，返回的是与泛型相同的值

但是Thread类里面不能接受Callable接口的多线程应用。

第三种多线程的实现操作，目前了解就好。

## 线程的命名与取得：

所有的线程程序的执行，每一次都是不同的运行结果，因为它会根据自己的情况进行资源的抢占，所以如果想要区分每一个线程，那么就必须依靠线程的名字。对于线程的名字一般而言会在其启动之前进行定义，不建议对已经启动的线程进行更改名称，或者是为不同的线程设置重名的情况。

如果想要进行线程名称的操作，可以使用如下的方法：

构造方法：

·Thread(ThreadGroup group, Runnable target){}

Setter/getter方法：

·public final String getName(){} //带有final，说明不能够被覆写

·public final void setName(String name){}

对于线程名字的操作会出现一个问题，这些方法是属于Thread类里面的，如果换回到线程类（Runnable）子类，这个子类里面并没有继承Thread类，如果你想去的名称名字，那么就能取得的就是当前执行本方法线程的名字。所以在Thread类里面提供有一个方法：

·取得当前线程对象：

public static Thread currentThread()。 //定义为static方法表明可以有Thread类直接调用。

**package** cn.test.deom;

**class** MyThread **implements** Runnable{ //这是一个多线程的类

@Override

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

System.***out***.println(Thread.*currentThread*().getName());

}

}

**public** **class** TestDemo{

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

MyThread mt=**new** MyThread();

**new** Thread(mt).start();

**new** Thread(mt).start();

**new** Thread(mt).start();

**new** Thread(mt,"线程1").start();

**new** Thread(mt,"线程2").start();

}

}

没有设置名字的时候，线程对象会自动的命名，保证编号对象的名称不重复。

main

自己的线程对象

**public** **class** TestDemo{

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

MyThread mt=**new** MyThread();

**new** Thread(mt,"自己的线程对象").start();

mt.run(); //直接调用run（）方法，结果：main

}

}

若这样定义：即可说明，原来主方法就是一个线程（main线程），那么在所有在主方法上创建的线程实际上都可以将其表示为子线程。

通过以上的代码可以发现线程一直都存在，但是我们的线程去哪里了？

每当使用java命令去解释一个程序类的时候，对于操作系统而言，都相当于启动了一个新的进程，而main只是这个新进程上的子进程。

## 每个JVM进程启动的时候至少启动几个线程？

至少两个

·main线程：程序的主要执行，以及启动子线程

·gc线程：负责垃圾收集。

## 线程的休眠操作：

线程的休眠操作就是让线程的执行变得慢一点，

### 休眠的方法：

public static void sleep(long millis) throws InterruptedException

休眠后会抛出一个中断异常。

### 注意：

默认情况下，在休眠的时候如果设置了多个线程对象，那么所有的额线程对象将一起进入到run()方法（所谓的一起进入实际上是因为先后顺序实在是太短了，但实际上有区别）细微的，但是这细微的差别就可能造成细微的错误。

因为每一次执行run方法的时候都要进行休眠，所以执行方法变慢了。

默认情况下，在休眠的时候如果设置了多个线程对象，那么所有的线程对象将一起进入休眠，虽然是一起出现的，实际上是因为先后顺序实在是太短为了，但实际上是有区别。）

## 线程优先级：

所谓的优先级指的是越高的优先级，越有可能执行。在Thread类里面提供有以下的两个方法进行优先级操作。

·设置优先级：public final void setPriority(int newPriority);

·取得优先级：public final int getPriority()。

发现设置和取得优先级都是使用了int数据类型，对于此内容有三种取值：

·最高优先级：public static final int MAX\_PRIORITY； 值为10；

·中等优先级：public static final int NORM\_PRIORITY; 值为5；

·最低优先级：public static final int MIN\_PRIORITY; 值为1；

主线程属于中等优先级

Thread.currentThread()可以取得当前线程类对象；

优先级越高的线程对象，有可能先执行。

# 线程的同步与死锁（了解）：

实际上所谓的同步指的就是多个线程访问统一资源时所需要考虑到的问题。

不同步指的是每一个访问的用户，进来（经过了休眠）看的比较清楚，开始是进来，然后判断条件，然后满足，下一步，休眠，然后执行下一步，有可能不同的用户进来，导致一块执行，从而导致安全隐患。俗话说：异步会造成安全隐患。

## 线程同步处理操作：

所谓的同步就是指多个操作在同一时间内只能有一个线程进行，其他线程要等待此线程完成之后才可以继续执行。

·使用**synchronized**关键字。可以通过两种方式使用。

·一种是同步代码块；

·另外一种是同步方法。

### 同步块：

要一个锁定对象，当前对象最为合适，

Synchronized(锁定对象){}来表示，这样表示当前操作只允许一个对象进入。

锁定对象通常为当前对象，即this。

### 同步方法：

Public synchronized void sale(){ }

Public synchronized 返回值 方法名称（）{}

## 优缺点：

同步操作与异步操作相比，异步操作的执行速度要比同步操作的速度快但是同步操作时数据的安全性较高，属于安全的线程操作。90%时写的是异步操作，所以 要同步的时候，需添加关键字。

## 线程死锁的情况：

所谓的同步就是指的是一个线程等待另一个线程对象执行完毕后的操作形式。同步过多就会造成死锁。总是在互相等待，导致程序执行不了，就造成了死锁问题。

死锁是一种不定的状态，只能在代码的检测中出现。

### 面试题：

请解释多个线程访问同一资源时需要考虑到那些情况？有可能带来那些问题？

·多个线程访问同一资源时一定要处理好同步，可以使用同步代码块或者同步方法。

·但是过多的使用同步，可能造成死锁问题。

# StringBuffer类：

String类的特点：

|-直接赋值：只开辟一块堆内存空间，不可以自动入池

|-构造方法：开辟两块堆内存空间，不会自动入池，使用intern（）手工入池

·任何一个字符串都是String类的匿名对象；

·字符串一旦声明则不可改变，可以改变的只是String类的对象引用。且String类的内容不可改变。为此提供了一个StringBuffer类（里面的内容可修改）

String类必须要用“+”来进行字符串的连接操作，StringBuffer类里面必须使用append()方法来进行字符串的连接。

String和StringBuffer类都继承了一个CharSequence接口的子类。而在以后的开发之中，如果看见某些方法的操作上出现的是CharSequence接口即可。

CharSequence接口接收的就是字符串，虽说这两个类都有着同样的接口，但是这两个类对象之间如果要转换，不能够直接转换。

如需进行转换有以下两种方式；

|-方式一：利用StringBuffer类的构造方法（public StringBuffer（String str））;

|-方式二：利用append（）方法接收字符串

String 和 StringBuffer里面也可以直接进行比较：

利用 String类.contentEquals(StringBuffer类) 来比较，返回的是boolean值。

StringBuffer和String中的方法基本都是互补的

1. 字符串的反转：public StringBuffer reverse()就是正序转倒序
2. 在指定索引位置增加数据：public StringBuffer insert(int offset,数据类型 变量)

左到右，连续两个插入的时候，先执行左边的。

1. 删除指定位置的数据：public StringBuffer delete(int start,int end)

## 面试题：请解释String、StringBuffer、StringBuilder的区别？

·String的内容一旦声明则不可改变，而StringBuffer与StringBuilder声明的内容可以改变；

·StringBuffer类中提供的方法都是同步方法，属于安全的线程操作，而StrinBuilder类中的方法都属于异步方法，属于非线程的安全操作。

日后再开发之中，如果看到了字符串的应用，不需要思考95%使用的都是String类，只有在需要频繁修改的时候会考虑到使用StringBuffer或者是StringBuilder类操作。

# Runtime类：（了解）继承的是Object类。

每个JVM进程里面都会存在一个Runtime类的对象，这个类的主要功能是取得一些与运行时有关的环境的属性，或者创建新的进程等操作。

在Runtime类定义的时候它的构造方法已经被私有化了，这就是属于我们的单例设计模式应用，因为要保证在整个进程里面只有一个。所以一定会设置有一个static方法用于取得Runtime对象：

Public static Runtime getRuntime();

方法：

·返回所有可用内存空间：public long totalMemory();

·返回最大可用额你存空间：public long maxMemory();

·返回空余内存空间：public long freeMemory().

·释放垃圾空间：public void gc()

请解释什么叫GC？如何处理？

·GC（GarbageCollector）垃圾收集器，值得是释放无用的内存空间；

·GC会由系统不定期进行自动的回收，或者调用Runtime类中的gc（）方法手工回收。

Runtime类还有一个更加有意思的功能，就是说他可以嗲用本机的可执行程序，并且创建进程。

## 创建进程：

public Process exec(String[] cmdarray)throws IOException

例子：调用本机程序，创建画板：

**package** cn.test.deom;

**public** **class** Hello

{

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

Runtime run=Runtime.*getRuntime*(); //取得Runtime类的实例化对象

Process pro=run.exec("mspaint.exe");//调用本机程序

Thread.*sleep*(2000);

pro.destroy(); //销毁进程

}

}

# System类：

之前一直使用的System.out.println()就属于System类的操作 功能。

重要的方法：

·取得当前系统的时间：public static long currentTimeMillis().

## 例子：计算执行所花的时间

**package** cn.test.deom;

**public** **class** Hello

{

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

**long** start=System.*currentTimeMillis*(); //取得开始时间

String str="";

**for** (**int** x = 0; x < 3000; x++) {

str+=x;

}

**long** end=System.*currentTimeMillis*(); //取得结束时间

System.***out***.println("本次操作所花的时间："+(end-start));

}

}

统计的是毫秒时间

System类里面也定义了一个操作方法：public static void gc()，这个GC就是调用前面的gc就是同一个方法。

对象产生一定会调用构造方法，可以进行一些处理操作，但是某一个对象如果要被回收了，连一个收尾的机会都没有

如果要给对象一个收尾的机会，那么就可以考虑覆写Object类中的finalize（）方法，

·finalize()方法：protected void finalize()throws Throwable

|-在对象回收时就算抛出了任何的异常，也不会影响到整个程序的正常执行 。

其实就是析构函数。

## 解释：

请解释final、finally、finalize的区别？

·final:关键字，定义不能被继承的类、不能被覆写的方法、常量；

·finally：关键字，异常的统一出口；

·finalize:方法。Object类提供的方法（protected void finalize()throws Throwable），指的是对象回收前的收尾工作，即使出现了异常也不会导致程序的中断执行。

Gc的调用是随机的，所以可以手动调用，

System类中的gc（）方法就直接调用了“Runtime.getRuntime().gc”方法。

## 对象克隆：

对象克隆就是对象的复制操作,是Object类里面的。

Protected Object clone() throws CloneNotSupportedException

抛出的是克隆不支持。

如果要使用对象克隆，就要使用Cloneable接口, Cloneable看不见方法，这里的表示的是标识接口，没有任何方法的定义，只有空接口，调用父类的克隆方法 。

对象克隆的理论价值大于实际价值，因为在实际工作里面很少会用到对象克隆的操作，重点在标识接口一样。

# 比较器：

## Arrays类：

在之前一直使用的“java.util.Arrays.sort()”可以实现数组的排序，而Arrays类就是java。Util包中提供的一个工具类，这个工具类主要是完成与数组有关的操作功能

有个折半查找法，前提是排过序的

java.util.Arrays.sorts(数组);表示排序

arrays.binarySearch(data,9);表示二分查找法

## comparable(核心):

public static void sort(Object[] a,int fromIndex,int toIndex)

·对象数组排序

比较的接口类：comparable

String类中用的是比较是用compareTo()方法ccc

## 二叉排序树：binaryTree

## 挽救的比较器：comparator（主要用comparable）

# 日期处理类：

## 取得当前系统时间：

package cn.test.deom;

import java.util.Date;

public class Hello

{

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

Date data=new Date();

System.out.println(data);

}

}

## 重要的方法：

·无参构造：public Date();

·有参构造：public Date(long date);

·转换为long型：public long getTime().

例子：

package cn.test.deom;

import java.util.Date;

public class Hello

{

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

//取得当前的日期时间以long型返回

long cur=System.currentTimeMillis();

Date data=new Date(cur);

System.out.println(data); //输出对象信息

System.out.println(date.getTime());

}

}

## 日期格式化：SimpleDateFomat

SimpleDateFormat是一个专门将Date型的对象转化为String型的形式显示。而主要的是以下列方法实现的：

**·构造方法：public SimpleDateFormat(String pattern),需要传递转换格式;**

**·将Date转换为String：public final String format(Date date);**

**·将String转换为Date:public Date parse（String source）throws ParseException**

### 时间格式：

常见的转换单位：年（yyyy）、月（MM）、日（dd）、时（HH）、分（mm）、秒(ss)、毫秒(SSS)

## 日期格式化的转换：

package cn.test.deom;

import java.text.SimpleDateFormat;

import java.util.Date;

public class Hello

{

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

Date date=new Date();

SimpleDateFormat sdf=new SimpleDateFormat("yyyy-MM-dd HH:mm:ss:SSS");

String str=sdf.format(date); //将Date型变为了String

System.out.println(date);

System.out.println(str);

}

}

## 将字符串转换为日期：

package cn.test.deom;

import java.text.SimpleDateFormat;

import java.util.Date;

public class Hello

{

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

String str="2000-11-11 11:11:11.111";

SimpleDateFormat sdf=new SimpleDateFormat("yyyy-MM-dd HH:mm:ss:SSS");

Date date=sdf.parse(str);

System.out.println(date);

}

}

在数据表里的操作中有几个常用的类型：VARCHAR2(String)、CLOB(String)、Number（Double、int）、Date(java.util.Date)包来表示。

Date与String类之间的转换依靠的是SimpleDateFormat;

String与基本类型之间的转换依靠的是包装类与String.valueOf()方法；

Long与Date转换依靠的是Date类提供的构造以及getTime()方法。

## Calendar类：（主要进行一些简单的日期计算）

是一个抽象类，依靠子类进行对象的实例化

### 重要的一个方法：

Public static Calendar getInstance()

返回的是此类对象。

## 代码例子：（取得当前的日期时间）

package cn.test.deom;

import java.text.SimpleDateFormat;

import java.util.Calendar;

import java.util.Date;

import java.util.spi.CalendarDataProvider;

public class Hello

{

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

Calendar cal=Calendar.getInstance();//取得本类对象

StringBuffer buf=new StringBuffer();

buf.append(cal.get(Calendar.YEAR)).append("-");

buf.append(cal.get(Calendar.MONTH)+1).append("-");

buf.append(cal.get(Calendar.DAY\_OF\_MONTH)).append(" ");

buf.append(cal.get(Calendar.HOUR\_OF\_DAY)).append(":");

buf.append(cal.get(Calendar.MINUTE)).append(":");

buf.append(cal.get(Calendar.SECOND));

System.out.println(buf);

}

}

使用Calendar方便日期的计算。

# 正则表达式：

所有正则的类都在JAVA.UTIL.REGEX包里 由离散数学可深入探究

Matcher类：通过pattern类取得

Pattern类：此类对象如果要想取得必须使用compile（）方法，方法的功能是编译正则。

## 正则标记：（背）

1. 单个字符（数量：1）

字符：表示由一位字符所组成；

\\：表示转义字符“\”（一个杠要用两个杠来表示）

\t：表示一个“\t”符号；

\n：匹配换行（\n）符号；

1. 字符集（）

[abc]：表示可能是字符a或者b或者c中的一位

[^abc]：表示不可能是字符a或者b或者c中的一位

[a-z]：表示所有的小写字母

[a-zA-Z]：表示任意的一位字母，不区分大小写

[0-9]：表示任意的一位数字

1. 简化的字符集表达式

.（点）：表示任意的一位字符；

\d：表示[0-9]的缩写

[\\d](file:///\\\\d)+表示多位数字

\D：表示[^0-9]

\s：表示任意的非空白字符串，例如:“\t”,“\n”；

\S：表示任意的非空白字符串

\w：等价于“[a-zA-Z\_0-9]”，表示任意的字母、数字、下划线组成

\W：等价于“[^a-zA-Z\_0-9]”,表示不是由任意的字母、数字、\_所组成

1. 边界匹配（不要再JAVA中使用，在Javascript里使用）：

**^**：正则的开始；

$：正则的结束；

1. 数量表达

正则（跟上问号）？：表示此正则可以出现0次或者1次；

正则（跟上加号）+：表示此正则可以出现0次或者1次以上；

正则（跟上星号）\*：表示此正则可以出现0次、1次、或多次；

正则{n}：表示此正则正好出现n次；

正则{n,}：表示此正则正好出现n次以上

1. 逻辑运算：

正则1正则2：正则1判断完判断正则2；

正则1|正则2：正则1或者是正则2有一组满足即可；

（正则）：将多个正则作为一组，可以为这一组单独设置出现的次数

## String类对正则的支持（重点）

|  |  |  |  |
| --- | --- | --- | --- |
| No. | 方法名称 | 类型 | 功能 |
| 1 | Public Boolean matches() | 普通 | 正则验证，使指定的字符串判断是否符合正则表达式的结构 |
| 2 | Public String replaceAll(String regex,String replacement) | 普通 | 全部替换 |
| 3 | Public String replaceFirst(String regex,String replacement) | 普通 | 替换收个 |
| 4 | Public String[] split(String regex) | 普通 | 全部拆分 |
| 5 | Public String[] split(String regex,int limit) | 普通 | 部分拆分 |

代码例子：保留字符串中的字母

package cn.test.deom;

public class Hello

{

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

String str="as;ldfk;'ask'c;zl,v;zzxc/,z/.c";

String regex="[^a-z]"; //此处编写正则

System.out.println(str.replaceAll(regex, ""));

}

}

## 正则表达式判断是否为ip地址：

package cn.test.deom;

public class Hello

{

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

String str="192.168.1.1";

String regex="(\\d{1,3}\\.) {3}\\d{1,3}"; //此处编写正则

System.out.println(str.replaceAll(regex, ""));

}

}

## 判断日期时候是时间格式：

package cn.test.deom;

import java.text.SimpleDateFormat;

import java.util.Date;

public class Hello

{

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

String str="2009-03-23";

//判断是否是日期格式

String regex="\\d{4}-\\d{2}-\\d{2}"; //此处编写正则

System.out.println(str.matches(regex));

if (str.matches(regex)) {

Date date=new SimpleDateFormat("yyyy-MM-dd").parse(str);

System.out.println(date);

}

}

}

## Java.util.regex包的支持

## 总结：

利用正则表达式可以使代码最小化

# 反射机制：

反就是指利用对象找到对象的出处，在Object类里面提供有一个方法：

取得class对象： public final Class<?>getClass()

所有的类都会有getclass方法

## Class类对象的实例化方法：

Java.lang.Class是一个类，这个类是所有反射操作的源头，即：所有的反射都要从此类开始进行，而这个类有三种实例化方式：

例

**·第一种：**调用Object类中的getClass()方法;

package cn.test.deom;

import java.util.Date;

public class Hello

{

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

Date date=new Date(); //产生对象

Class<?> cls=date.getClass();

System.out.println(cls);

}

}

**·第二种：**使用“类.class”取得;

package cn.test.deom;

import java.util.Date;

public class Hello

{

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

Date date=new Date(); //产生对象

Class<?> cls=Date.class;

System.out.println(cls);

}

}

之前在产生实例化对象之后取得的Class类对象，但是此时并没有实例化对象的产生。

**·第三种：**调用Class类提供的一个方法；

package cn.test.deom;

public class Hello

{

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

Class<?> cls=Class.forName("java.util.Date");

System.out.println(cls);

}

}

此时可以不用import语句倒入一个明确的类，而类名称是采用字符串的形式进行描述的

## 反射实例化对象：

当拿到一个类的时候，肯定要直接使用关键字new进行对象的实例化操作，这属于一个习惯性的做法，但是有了Class类对象，那么就可以做到，利用反射来进行对象实例化操作：

**·实例化对象方法：public T newInstance()throws InstantiationException,IllegalAccessException**

代码例子：

package cn.test.deom;

class Book{

public Book() {

System.out.println("\*\*\*\*\*\*Book类的无参构造方法\*\*\*\*\*\*\*\*");

}

@Override

public String toString() {

return null;}

}

public class Hello

{

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

Class<?> cls=Class.forName("cn.test.deom.Book");

Object obj=cls.newInstance();

Book book=(Book)obj;

System.out.println(book);

}

}

什么时候用new什么时候用newInstance要注意，new是大多情况下造成耦合的真正元凶.

## 改良版工厂设计模式：

package cn.test.deom;

interface Fruit{

public void eat();

}

class Apple implements Fruit{

@Override

public void eat() {

System.out.println("\*\*\*吃苹果");

}

}

class Orange implements Fruit{

@Override

public void eat() {

System.out.println("\*\*\*吃橘子");

}

}

class Factory{

public static Fruit getInstance(String className) {

Fruit f=null;

try {

f=(Fruit)Class.forName(className).newInstance();

} catch (Exception e) {

// TODO: handle exception

return f;

}

return f;

}

}

public class Hello

{

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

Fruit f=Factory.getInstance("cn.test.deom.Orange");

f.eat();

}

}

## 使用反射调用构造：

Class类里面提供有一个构造方法可以取得构造：

**·取得全部构造：**public Constructor<?>[] getConstructors()throws SecurityException

**·取得一个指定参数顺序的构造函数：**

**public Constructor<T> getConstructor(类<?>... parameterTypes)throws NoSuchMethodException,SecurityException**

## 反射调用普通方法：

## 反射调用成员：

# 国际化：

所谓的国际化应用指的就是根据当前的语言换进度去指定的语言资源文件。

所谓的资源文件指的是后缀名称为“\*.properties”里面保存的内容按照“key=value”的形式保存，而且资源文件的命名标准与Java类完全一样。

Info=……

·如果保存的是中文信息，那么必须将其变为UNICODE编码，这里的info就是就是这个信息的key，以后要根据这个key取得对应的value。

读取资源文件的信息使用java.util.ResourseBundle类，这是一个抽象类，但是内部也定义有一个static方法用于取得本类对象：

·根据当前语言环境取出：public static final ResourceBundle getBundle(String baseName)

·设置指定语言环境：public static final ResourceBundle getBundle(String baseName,Locale locale)

注意：baseName就是Messages

访问的时候一定不要加上后缀，因为默认找到的后缀就是“\*.properties”

//此时的Message.properties文件一定要放在CLASSPATH路径下

通过ResourceBunde类可以读取在指定的CLASSPATH下的资源文件，读取时不需要输入文件后缀

|-动态占位

Locale类用于指定读取的资源文件的语言环境

# 文件操作类：（基本操作）

抽象类中定义的抽象方法会根据实例化子类的不同，也会完成不同的功能。

主要在java.io包中。

·五个核心类：File、InputStream、Reader、Writer；

·一个和新街口：Serializable.

File类是唯一一个与文件本身操作有关的类，但是不涉及文件的具体内容。

构造方法：

·设置完整路径：public File(String pathname),注：大部分情况下使用此类操作

·设置父路径与子文件路径：public File(File parent,String child)注：主要在安卓的情况下使用此类操作。

File类中定义有一个常量

Public static final separator

File.separator表示文档的分隔符即：“\”

单杠表示功能

记住转义字符要双杠

**范例：操作文件**

## ·创建文件：

public boolean createNewFile()throws IOException

为什么会抛异常：

|-如果目录不能访问；

|-如果现在文件重名，或者是文件名称错误

package cn.test.deom;

import java.io.File;

public class Hello

{

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

File file=new File("e:\\test.txt"); //设置文件的路径

System.out.println(file.createNewFile());

}

}

## 注意：

**上面的创建文件\t，\t是制表符，不能直接使用，遇到之后要用转义字符,上面没有考虑文件夹不存在时候的情况。**

## 删除文件：public Boolean delete();

## 判断文件是否存在：public Boolean exists();

package cn.test.deom;

import java.io.File;

public class Hello

{

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

File file=new File("e:\\test.txt"); //设置文件的路径

if (file.exists()) { //判断文件是否存在

file.delete(); //存在则删除

}else { //文件不存在，进行文件的创建

System.out.println(file.createNewFile());

}

}

}

在Window里面支持的是“\”路径分隔符，Linux下使用的是“/”路径分隔符；在实现的时候通常是Window来实现，然后在linux下进行部署

在进行java.io操作的过程之中，会出现有延迟的情况，因为现在的问题是Java程序是通过JVM间接的调用操作系统的文件处理函数进行的文件操作，所以中间会出现延迟情况。

**·取得文件大小：public long length();按字节返回**

**·判断是否是文件：public boolean isFile();**

**·判断是否是目录：public boolean isDirectory();**

**·最近一次修改日期：public long lastModified();返回的是一串long型数字，只需**

**用new SimpleDateFormat(“yyyy-MM-dd HH:mm:ss”).format(new Date(file.lastModified()))即可，在进行输出，即可获得转化后的时间。**

要使用，只需要将上述的类名改一下即可

还有很多方法，去看文档

## 操作目录：

**package** cn.test.deom;

**import** java.io.File;

**public** **class** Hello

{

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

File file=**new** File("e:"+File.***separator***+"文件路径"); //设置文件的路径

*print*(file);

}

**public** **static** **void** print(File file) {

**if** (file.isDirectory()) { //如果现在给定的是一个路径

File result[]=file.listFiles();

**if** (result !=**null**) {

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

*print*(result[i]);

}

}

}

System.***out***.println(file); //功能可以替换为删除操作

}

}

## 总结：

设置完整路径: public File(String pathname)

删除文件：public Boolean dete();

判断文件是否存在：public Boolean exists()

找到父路径：public File getParentFile();

创建目录：public Boolean mkdirs();

文件的分隔符：File.separator

# 字符流与字节流：

File类知识进行文件的操作，如果要进行内容的操作就要使用字符流或者字节流。

## 基本步骤：

步骤分别以如下步骤进行：

·通过File类定义一个要操作文件的路径；

·通过字节流或字符流的子类对象为父类对象实例化；

·进行数据的读（输入）、写（输出）操作；

·数据流属于资源操作，资源操作必须关闭。

字节流：InputStream、OutputStream

字符流：Reader、Writer

## 字节输出流：OutputStream

·输出单个字节：public abstract void write(int b)throws IOException;虽说输入的是int型，但导入的一定是byte型，

·输出全部个字节：public void write(byte[] b)throws IOException;

**·输出部分字节：public void write(byte[] b,int off,int len)throws IOException; OFF表示开始点，len表示长度**

**最重要，记住抽象类的方法不能直接使用，还需要利用抽象类的子类的进行实例化，看文档，**

**例子：package** cn.test.deom;

**import** java.io.File;

**import** java.io.FileOutputStream;

**import** java.io.OutputStream;

**public** **class** Hello

{

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

//定义要输出文件的路径

File file=**new** File("D:"+File.***separator***+"testdemo"+File.***separator***+"file.txt");

//1、此时由于目录不存在，所以文件不能够输出，那么应该首先创建目录

**if** (!file.getParentFile().exists()) {

file.getParentFile().mkdirs();

}

//2、应该使用OutputStream和其他子类进行对象的实例化，此时目录不存在，文件还不存在

OutputStream output=**new** FileOutputStream(file);

//3、要进行文件内容的输出

String str="好好学习，天天向上";

**byte** data[]=str.getBytes(); //将字符串转化为字节数组

output.write(data); //将内容输出

//4、资源操作的最后一定要进行关闭

output.close();

}

}

## 注意：

每次使用都会将新的内容替换旧的内容，记住一个汉字是两个字节。

OutputStream output=**new** FileOutputStream(file,true);即可实现数据的追加

若要实现换行，在此处加上\r\n

String str="好好学习，天天向上\r\n";

## 字节输入流：

InputStream也是一个抽象类，所以如果想要进行文件读取使用FileputStream类，构造方法接收file类对象

就是数据的读取操作，也实现了Closeable接口

## 方法：

·将读取的数据保存在字节数组里：public int read(byte[] b)throws IOException;

|-返回值：返回读取的数据长度，但是如果已经读取到结尾了，返回-1。

·将读取的数据保存在部分字节数组里：public int read(byte[] b,int) throws IOException;

|-返回值：读取部分数据的长度，如果已经读取到结尾了，返回-1.InputStream in = JedisPoolUtils.class.getClassLoader().getResourceAsStream(“路径/文件”)

Properties pro =new Properties();

Pro.load(in);

InputStream是一个抽象类。所以如果想要进行文件读取使用FileInputStream子类，而这个子类的构造：

·构造方法：public FileInputStream(File file)throws FileNotFoundException；

## 加载配置文件：

以redis为例子

## 字符流：

Writer类

Writer类是一个抽象类，继承了Closeable,Appemdable,Flushable接口，利用FileWriter类进行对象的实例化

在Appendable接口里面定义了追加操作，而且追加的数据都是字符或者是字符串。

在Writer类里定义了以下的输出方法：

·输出全部字符数组：public void write(char[] cbuf)throws IOException;

·输出字符串：public void write(String str)throws IOException;

## 字符输入流：Reader

Reader类里面也提供有一系列的read()方法：

·读取内容到字符数组：public int read(char[] cbuf)throws IOException

|-返回值：表示读取的数据长度，如果已经读取到结尾了，返回-1；

实例化用FileReader子类完成，方式差不多。

## 字符流与字节流的区别？

最大的区别就是：字节流直接与终端进行数据交互，字符流需要将数据经过缓冲区处理后才可以输出。

·在使用OutputStream输出数据的时候即使最后没有关闭输出流，那么内容也可以正常输出。如果是字符输出流，如果不关闭，那么久表示在缓冲区之中处理的内容不会被强制性的清空，所以不会输出数据。如果现在有特殊情况不能关闭输出流，**可以使用flush()方法，**强制清空缓冲区。

但是内容还是会存在的。

字符流是因为可以进行中文的有效处理，如果用字节流可能会出现问题。

## 字节流和字符流可以转换

字符流处理中文比较方便，所以转换

## 结论：

方法一：可以一次性读完，然后一次性输出;但是如果文件太大，电脑将会崩溃

方法二：可以边读边输出，这样就不会占用内存过大

正常工作中异常都不能直接抛出。

# 字符编码：

·GBK、GB2312:中文的国标编码，其中GBK包含有简体中文与繁体中文两种，而GB2312只包含简体；

·ISO8859-1：是国际编码，可以描述任何的文字信息

·UNICODE：是十六进制编码，造成传输的无用数据过多；

·UTF编码（UTF-8）：融合了ISO8859-1和UNICODE编码的特点；在以后的开发之中，使用的都是UTF-8编码。

乱码最根本的就是编码与解码不统一

# 内存流：

文件操作：

|-输出（OutputStream）:程序→OutputStream→文件；

|-输入（InputStream）:程序←InputStream←文件；

内存操作：

|-输出（OutputStream）:程序→OutputStream→内存；

|-输入（InputStream）:程序←InputStream←内存；

## 打印流：PrintStream见C

解决的是打印流

PrintStream（打印字节流）、PrintWriter（打印字符流）。

只要由程序输出内容，都会采用打印流的模式进行输出

其实就是C里面的控制格式的输出，

String.format是控制字符串

## System类对IO的支持（输出）：

·错误输出：public static final PrintStream err;

·输出到标准设备（显示器）:public static final PrintStream out;

·从标准输入设备读取（键盘）：public static final InputStream in.

严格上来讲，System.err和System.out的功能是完全一样的，之所以这样设计，主要的目的是为了让用户看见不能直接看见的错误，而out输出的是可以让用户看见的信息，但是现在基本上没人再去分了。

System.out是专门用来支持屏幕输出的一个类，由系统自动实例化，

# IO高级应用

## 缓冲输入流：

目的是为了解决数据的乱码问题

缓冲区的操作有两种流：

·字符缓冲区流：BufferedReader、BufferedWriter;

·字节缓冲区流：BufferedInputStream、BufferedOutputStream;

## BufferedReader: 是Reader类的子类，且接收Reader类的对象

这个最为重要，里面有两个重要的读取方法：public String readLine() throws IOException 读取一行数据，且返回的是String类型。

数据输入的标准格式：

## Scanner数据输入流：

Scanner解决了输入的问题，util包

构造方法：

public Scanner(InputStream source)

·判断是否有指定数据：public Boolean hasNextXxx();

·取出数据：public 数据类型 next数据类型().

如果说现在输入的是字符串，是否有hasNext（）没有关系，但是如果说是要输入其他字符，这个hasNext就有意义了。但是为了保持统一，建议有。

也可以实现文件读取，但是文件读取的时候需要考虑分隔符。

## 总结：

先用Scanner ，当它不好用了，就用BufferedReader

# 对象序列化：

对象序列化就是将保存在内存中的对象数据转换为二进制数据流进行传输的操作，但是不是所有的类都可以进行序列化，如果要进行序列化对象，那么其所在的类一定要实现java.io.Seriakizable接口。

压制警告就行了，只要继承了接口，就可以进行二进制传输了。

序列化之后，只能进行反序列化。

## transint关键字：

pribate transient String title;

表示对象序列化是title将无法保存

# 网络编程：（了解）

网络编程是进行服务器端和客户端进行的开发

但是实际情况下有两种形式：

·形式一：C/S结构（Client/Server）,此类模式的开发一般要编写两套代码，一套是客户端代码，另一套属于服务器端代码，这样的程序开发非常的麻烦，因为要维护两套程序的使用，但是这样安全性高，使用的是自己的连接端口，并且使用的是自己的通讯协议。

·形式二：B/S结构（Broweser/Server）,不在单独开发客户端代码，只开发一套服务器端程序。客户端将利用浏览器进行访问，这种模式只需要开发一套程序，但是安全性不高，因为使用的是公共的HTTP协议以及公共的80端口。

C/S结构的程序分为两类：

·TCP程序：是采用可靠的连接方式进行的传输；

·UDP程序：不可靠的链接，属于数据报协议。

# 开发网络程序：java.net

如果要进行网络程序的开发，那么最为核心的两个类：

·服务器类：ServerSocket,主要工作在服务器端，用于接收用户的请求

·客户端类：Socket，每一个连接到服务器上的用户都通过Socket表示

一般用8000以上的端口，

客户端连接一次服务器端，并获得一次服务器端的输出数据，然后输出

## 范例定义服务器——主要使用SeverSocket:

·构造方法:public ServerSocket(int port) throws IOException，设置监听器端口;

·接收客户端连接：public Socket accept() throws IOException

·取得客户端的输出功能，Socket类定义方法：

public OutputStream getOutputStream() throws IOException;

如果客户端在等待连接的过程中，没有与其连接，将会出现堵塞的情况。将会极大的占用资源。

## Echo模型：

客户端随意输入信息，并且将信息发送给服务器端，服务器端接收后，前面加上一个“ECHO:”的标记返回。

# 类集框架：

类集就是JAVA实现数据结构，指的对象数组的使用

所谓的类集就是对像数组。

通常说Hash…，就是无序，Tree…，就是有序。

## Collection接口：

Collection是整个类集中单值保存的最大父接口。

|  |  |  |  |
| --- | --- | --- | --- |
| No: | 方法名称 | 类型 | 功能 |
| 1 | public boolean add(E e) | 普通 | 向集合里面保存数据 |
| 2 | public boolean addAll(Collection<? extends E>c) | 普通 | 追加一个集合 |
| 3 | public void clear() | 普通 | 清空集合，根元素为NULL |
| 4 | public boolean contains(Object o) | 普通 | 判断是否是包含有指定的内容，需要equals()支持 |
| 5 | public boolean isEmpty() | 普通 | 判断是否是空集合（不是NULL） |
| 6 | public boolean remove(Object o) | 普通 | 删除对象，需要equals()支持 |
| 7 | public int size() | 普通 | 取得集合中保存的元素个数 |
| 8 | public Object[] toArray() | 普通 | 将集合变为对象数组保存 |
| 9 | public Iterator<E> iterator() | 普通 | 为Iterator接口实例化 |

在开发之中，add()和iterator()两个方法的使用机率是最高的，其他方法几乎可以忽略，但是你必须知道

### 注意：

**contains()与remove()两个方法一定要靠equals（）支持。**

**基本上不用collection()接口了。**

## List子接口：

List子接口是Collection中最为常用的一个子接口，（80%）的情况是用他。

|  |  |  |  |
| --- | --- | --- | --- |
| No. | 方法名称 | 类型 | 描述 |
| 1 | public E get(int index) | 普通 | 取得索引编号的内容 |
| 2 | public E set(int index,E element) | 普通 | 修改指定索引编号的内容 |
| 3 | public ListIterator<E> listIterator() | 普通 | 为ListIterator接口实例化 |

而List本身是属于接口，所以如果想要使用此接口进行操作，那么就必须存在有子类，两个ArrayList(90%)和Vector(10%)的情况下

**注意**：删除操作和内容的查询操作都必须提供有equals()方法的支持

### ArrayList与Vector的区别：

|  |  |  |  |
| --- | --- | --- | --- |
| NO： | 区别点 | ArrayList(90%) | Vector(10%) |
| 1 | 推出时间 | JDK1.2退出，属于新的类 | JDK1.0时退出，属于旧的类 |
| 2 | 性能 | 采用异步处理 | 采用同步处理 |
| 3 | 数据安全 | 非线程安全 | 线程安全 |
| 4 | 输出 | Iterator、ListIterator、foreach | Iterator、ListIterator、foreach、Enumeration |

基本上用了List，就用ArrayList。

Eg：

package study.wjn.list\_01;

import java.util.ArrayList;

import java.util.List;

public class Test {

public Test() {

// TODO Auto-generated constructor stub

}

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

//设置了泛型，从而保证集合中所有的数据类型都一样

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

System.out.println("长度"+all.size()+"，是否为空："+all.isEmpty());

all.add("Hello");

all.add("Hello"); //重复数据

all.add("Word");

System.out.println("长度"+all.size()+"，是否为空："+all.isEmpty());

**//Collection接口定义了size()方法 ，可以取得集合的长度**

**//List子接口扩充了一个get()方法，可以根据索引取得数据**

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

String str=all.get(x); //取得索引数据

System.out.println(str);

}}}

## Set子接口：

在Collection接口下的另一个子接口，Set接口（20%）的情况下。

Set接口只是简单的继承了Collection接口，并没有像List接口那样进行大量的扩充。

也就没有了之前集合所提供的get()方法了。

Set接口下有两个常用的子类：就是HashSet、TreeSet。

|-Set集合下没有重复元素，同时发现里面保存的数据是没有任何顺序的，

HashSet的子类没有重复数据，且是无序的

TreeSet的子类没有重复数据，其所保存的内容自动排序。

## 关于数据排序的说明：

集合就是一个动态的对象数组，那么如果要为一组对象进行排序，在Java里面就必须要使用比较器，应该使用Comparable接口中的compareTo()方法完成比较。

### 关于重复元素的说明：

很明显，Comparable接口只能够负责TreeSet子类进行重复元素的判断，他并不是真正的用于能够进行重复元素验证的操作。

如果要想判断重复元素那么饿只能够依靠Object类中提供的方法：

·取得哈希码： public int hashCode()；

|-先判断对象的哈希码是否相同，依靠哈希码取得一个对象的内容；

·对象比较：public boolean equals(Object obj)。

|-再将对象的属性进行依次的比较。

写太麻烦，直接用工具生成。

## 总结：

1、在开发志宏，Set接口绝对不是首选，如果真要使用也建议使用HashSet子类；

2、Comparable这种比较器大部分情况下只会存在于Java理论范畴内，例如：要进行TreeSet.

3、Set不管如何操作，必须始终保持一个前提：数据不能够重复。

# 集合输出

Collection、List、Set三个接口是最方便进行输出操作的，所以本次要讲解集合的四种输出操作形式。

## 四种输出：

集合JDK1.8之前支持

四种输出：

Iterator(95%)、ListIterator(0.05%)、Enumeration(4.9%)、foreach(0.05%)。

Enumeration通常是作为程序架构的关键接口，Itreator是最常用的

### Iterator迭代输出：

注意Scanner就是Iterator的子类

如果遇见集合操作，那么一般而言都会使用Iterator接口进行集合的输出操作，

需要掌握的定义：设置了泛型

public interface Iterator<E>{

public boolean hasNext(); //判断是否有下一个

public E next(); //获取下一个

}

Iterator本身是一个接口，如果要取得本接口的实例化，只能够依靠Collection接口，在Collection接口里面定义有如下的一个操作方法：public Iterator<E>iterator()；

**注意：**

**☆**该方法返回的是Iterator，所以可以使用该方法为Iterator进行实例化。

**观察下面的使用：**

**package** study.Iterator;

**import** java.util.HashSet;

**import** java.util.Iterator;

**import** java.util.Set;

**public** **class** Test {

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

Set<String> all=**new** HashSet<String>();

all.add("Hello");

all.add("word");

Iterator<String> iter=all.iterator();

**while** (iter.hasNext()) {

String string = (String) iter.next();

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

}}}

### ListIterator（）双向迭代输出

明确ListIterator是Iterator的子接口，所以ListIterator也可以用Iterator的方法；

**ListIterator扩充的两个方法**：

**·判断是否有前一个元素：public boolean hasPrevious();**

**·取得前一个元素：public E previous()。**

ListIterator是专门为List子接口定义的输出接口，**获得实例化对象的方法**：public ListIterator<E>IistIterator()。

记住由后向前输出的时候必须要由前向后输出，要不然就会出现问题。即没有内容输出。（基本上不用的，了解）

### foreach输出：

可以直接输出集合

package study.Iterator;

import java.util.ArrayList;

import java.util.List;

public class Test {

public static void main(String[] args) {

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

all.add("Hello");

all.add("word");

for (String string : all) {

System.out.println(string);

}}}

初学不建议使用。

### Enumeration输出：

枚举

## 接口的定义：

public interface Enumeration<E>{}

**两个方法：**

public boolean hasMoreElements() //判断是否有下一个元素，等同于hasNext()

public E nextElement(); //取出当前元素，等同于next()

但是要想取得Enumeration接口的实例化对象只能依靠Vector子类。

·取得Enumeration接口对象：public Enumeration<E>elements();

## 总结：

集合的输出一定一Iterator与Enumeration接口为主

# Map接口：

如果说现在要保存一堆关联数据（key=value）的时候，那么如果直接食用Colletion就不能直接满足要求，可以使用Map接口实现此类数据的保存，并且Map接口还提供有根据key查找value的功能。

|  |  |  |  |
| --- | --- | --- | --- |
| No. | 方法名称 | 类型 | 说明 |
| 1 | public V put(K key,V value) | 普通 | 向集合中保存数据 |
| 2 | public V get(Object key) | 普通 | 根据key查找对应的数据 |
| 3 | public Set<Map.Entry<K,V>>entrySet() | 普通 | 将Map集合转化为Set集合 |
| 4 | public Set<K>keyset() | 普通 | 取出全部的key |

注意key或者value的值为null的两种情况。

在使用Map接口的时候，几乎可以使用任意的类型来作为key或value来说明。这个**作为key的自定义类必须要覆写Object类中的hashCode()与equals()两个方法。**

Map也是用Iterator进行输出，但是Map的是Iterator输出很特别。

但是通常key的值只用String类型。

# Stack子类：

就是栈，先进后出。Stack是Vector的子类，但是Stack使用的是自己的方法。

**栈就是先进后出**

·入栈：public E push(E item);

·出栈：public E pop();

没有内容后出栈会报EmptyStackException的错误。

在安卓的使用中会比较常用到

# Properties子类：（能不用就不用）

资源文件都要使用Properties后缀来保存

Properties是Hashtable的子类，主要是进行属性的操作（属性的最大特点是利用字符串设置key和value）

定义：

public class Properties extends Hashtable<Object,Object>

在使用Properties类的时候不需要设置泛型类型，因为他从一开始就只能够保存String。在Properties用如下操作方法：

**·设置属性：public Object setProperty(String key,String value);**

**·取得属性：public String getProperties(String key),如果key不存在返回null；**

**·取得属性：public String getProperties(String key,String defaultValue),如果key不存在，返回默认值。**

# JDBC编程：（ORACLE版本）

### |-ODBC技术：基本不用

### |-JDBC技术：

### |-JDBC网络连接：

**用的最多**

### |-模拟指定数据库的通讯协议自己编写数据库操作。

主要以Oracle数据库为主

用完数据库一定要记得关闭。

## 主要内容：JDBC网络连接

·一个类：DriverManager类；

·四个接口：Connection、Statement、ResultSet、PreparedStatement。

# ORACLE

## 一共四个步骤：

1、加载数据库的驱动程序（向容器加载）;

2、进行数据连接（通过DriverManager类完成，Connection表示连接）；

3、进行数据的CRUD(Statement、PreparedStatement、ResultSet)；

4、关闭数据操作以及连接（直接关闭连接）。

### //连接数据库的前提

数据库的服务要打开；

3个重要的东西，

·数据库的连接地址：jdbc:oracle:连接方式：主机名称：端口名称 ：数据库的SID；

·数据库的用户名

### |-加载数据库驱动类

Driver是每个驱动类必须实现的接口

（mysql版本）

Class.forName("com.mysql.jdbc.Driver");

数据库驱动类的名字（oracle版本）

**“oracle.jdbc.driver.OracleDriver”**

//前提是你得加载数据库驱动，先导包，然后在利用Class.forName（）方法 进行数据库驱动的类的实例化。

### |-连接数据库：

在JDBC里面，每一个数据库连接都要求使用一个Connection对象进行封装，所以只要有一个新得Connection对像就表示要连接一次数据库。

Connection里面有重要的方法，Close方法,

数据库的连接属于资源操作。所以每次 操作完成 都需要 关闭连接。

### |-关闭数据库：

#### Close方法。

void close()

throws SQLException

#### 判断是否关闭

boolean isClosed()

throws SQLException

### |-操作数据库对象：

Statement接口实现：

取得Statement接口对象；

public Statement createStatement() throws SQLException

两个方法：

#### |-数据更新：

public **int** executeUpdate(String sql)throws SQLException

返回的是更新的行数

#### |-数据查询：

public ResultSet executeQuery(String sql) throws SQLException

返回的是数据库查询的结果

#### 执行数据库操作语句

public boolean execute(String sql) throws SQLException

注意返回的是是否执行成功。

## Connection接口：

## Statement接口 ：

Statement如果想要变成为灵活的应用就要采用拼凑字符串的方式完成，但是只要SQL语句中出现’单引号，整个SQL就会出现错误，也就是说Statement的执行模式不适合执行一些敏感字符串的 模式。

### 事物的批处理：

|-增加批处理语句：public void addBath(String sql)throws SQLException;

|-执行批处理：public int[] excuteBath() throws SQLException;

|-返回的结果中数组是包含了所有所有批处理语句的执行结果；

EG：

## Resultset接口：

## PreparedStatement接口：

不用Statement用PreparedStatement

PreparedStatement与Statement不同的地方在于，他执行的是一个完整的具备特殊占位标记的SQL语句，并且可以动态的设置数据。

PreparedStatement的接口实例化还是要用的Connection

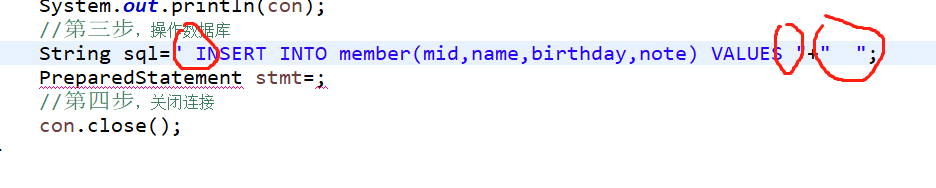
public **PreparedStatement** **prepareStatement**(String sql)throws SQLException{}

注意观察上述 ，一个是有d的，一个是没有d的。

需要用一系列的setXxx()方法用于为所使用的标记设置具体内容，

这里是导入了SQL语句的，注意看

### ·更新操作：



换行的时候注意前后面都要加上空格，避免不必要的麻烦

public int executeUpdate() throws SQLException

不用输入SQL语句了

PreparedStatement接口里面的setDate()方法，的DATE数据是SQL.DATE不是util.DATE

在更新数据更新数据这一块

DATE：

new java.sql.Date(birthday.getTime())

这样既可

**建表的时候表的字段和名字都建议大写。**

测试的时候还出了蛮多岔子的。

### ·查询操作：

public ResultSet executeQuery() throws SQLException

不用输入SQL语句了

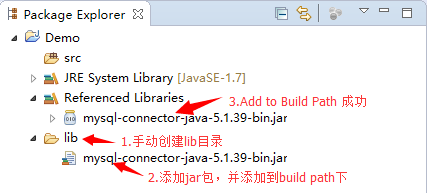
批处理的方法的：

# MySQL

### 导入驱动jar包

创建lib目录，用于存放当前项目需要的所有jar包

选择jar包，右键执行build path / Add to Build Path



### API详解：注册驱动

代码：Class.forName("com.mysql.jdbc.Driver");

JDBC规范定义驱动接口：java.sql.Driver，MySql驱动包提供了实现类：com.mysql.jdbc.Driver

DriverManager工具类，提供注册驱动的方法 registerDriver()，方法的参数是java.sql.Driver，所以我们可以通过如下语句进行注册:

DriverManager.registerDriver(new com.mysql.jdbc.Driver());

以上代码不推荐使用，存在两方面不足

1. 硬编码，后期不易于程序扩展和维护
2. 驱动被注册两次。

通常开发我们使用Class.forName() 加载一个使用字符串描述的驱动类。

如果使用Class.forName()将类加载到内存，该类的静态代码将自动执行。

通过查询com.mysql.jdbc.Driver源码，我们发现Driver类“主动”将自己进行注册

public class Driver extends NonRegisteringDriver implements java.sql.Driver {

static {

try {

java.sql.DriverManager.registerDriver(new Driver());

} catch (SQLException E) {

throw new RuntimeException("Can't register driver!");

}

}

……

}

### API详解：获得链接

代码：Connection con = DriverManager.getConnection  
(“jdbc:mysql://localhost:3306/mydb”,”root”,”root”);

获取连接需要方法 DriverManager.getConnection(url,username,password)，三个参数分别表示，url 需要连接数据库的位置（网址） user用户名 password 密码

url比较复杂，下面是mysql的url：

jdbc:mysql://localhost:3306/mydb

JDBC规定url的格式由三部分组成，每个部分中间使用冒号分隔。

 第一部分是jdbc，这是固定的；

 第二部分是数据库名称，那么连接mysql数据库，第二部分当然是mysql了；

 第三部分是由数据库厂商规定的，我们需要了解每个数据库厂商的要求，mysql的第三部分分别由数据库服务器的IP地址（localhost）、端口号（3306），以及DATABASE名称(mydb)组成。

### API详解：获得语句执行平台

String sql = "某SQL语句";

获取Statement语句执行平台：Statement stmt = con.createStatement();

常用方法：

* + int executeUpdate(String sql); --执行insert update delete语句.
  + ResultSet executeQuery(String sql); --执行select语句.
  + boolean execute(String sql); --执行select返回true 执行其他的语句返回false.

### API详解：处理结果集（执行insert、update、delete无需处理）

ResultSet实际上就是一张二维的表格，我们可以调用其boolean next()方法指向某行记录，当第一次调用next()方法时，便指向第一行记录的位置，这时就可以使用ResultSet提供的getXXX(int col)方法(与索引从0开始不同个，列从1开始)来获取指定列的数据：

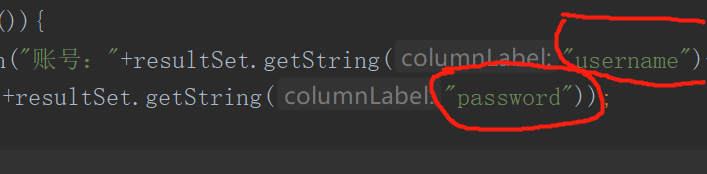
rs.next();//指向第一行

rs.getInt(1);//获取第一行第一列的数据

常用方法：

* Object getObject(int index) / Object getObject(String name) 获得任意对象
* String getString(int index) / Object getObject(String name) 获得字符串
* int getInt(int index) / Object getObject(String name) 获得整形
* double getDouble(int index) / Object getObject(String name) 获得双精度浮点型

### 注意：



如果是第几行数据，直接加数字，就不用双引号了

### API详解：释放资源

与IO流一样，使用后的东西都需要关闭！关闭的顺序是先得到的后关闭，后得到的先关闭。

rs.close();

stmt.close();

con.close();

### 代码例子：

package NoprepareStatement;  
  
import java.sql.Connection;  
import java.sql.DriverManager;  
import java.sql.Statement;  
  
public class TestConnection {  
 public static void main(String args[]) throws Exception{  
// 1、实例化数据库驱动类  
 Class.*forName*("com.mysql.cj.jdbc.Driver");  
// 2、编写url之类，注意是8.0的数据库我  
 String url="jdbc:mysql://localhost:3306/mybase?" +  
 "useUnicode=true&characterEncoding=utf-8&useSSL=false&serverTimezone = GMT";  
 String username="root";  
 String password="rose";  
// 3、获取数据库连接  
// 注意是在DriverManager下连接到数据库的连接  
 Connection connection= DriverManager.*getConnection*(url,username,password);  
  
// 4、编写要执行的SQL语句  
// EG:  
 String sql="SELECT \* FROM 表名";  
  
// 注意是在Statement下才能执行  
 Statement statement = connection.createStatement();  
// 然后，statement下在执行sql语句  
  
// 里面接的是sql语句  
 statement.executeUpdate(sql);  
  
// 6、最后关闭资源的操作  
// 如果有resultset的话，这个也要关闭  
  
// 只要属于资源操作的范畴内，都要进行资源的关闭  
 statement.close();  
// resultset.close();  
 connection.close();  
  
  
  
 }  
}

## SQL注入问题

假设有登录案例SQL语句如下**:**

SELECT \* FROM 用户表 WHERE NAME = 用户输入的用户名 AND PASSWORD = 用户输的密码;

此时，当用户输入正确的账号与密码后，查询到了信息则让用户登录。但是当用户输入的账号为XXX 密码为：XXX’ OR ‘a’=’a时，则真正执行的代码变为：

SELECT \* FROM 用户表 WHERE NAME = ‘XXX’ AND PASSWORD =’ XXX’ OR ’a’=’a’;

此时，上述查询语句时永远可以查询出结果的。那么用户就直接登录成功了，显然我们不希望看到这样的结果，这便是SQL注入问题。

为此，我们使用PreparedStatement来解决对应的问题。

# 预处理对象：PreparedStatement

使用PreparedStatement预处理对象时，建议每条sql语句所有的实际参数，都使用逗号分隔。

String sql = "insert into sort(sid,sname) values(?,?)";;

PreparedStatement预处理对象代码：

PreparedStatement psmt = conn.prepareStatement(sql)

常用方法：

1. 执行SQL语句:

* int executeUpdate(); --执行insert update delete语句.
* ResultSet executeQuery(); --执行select语句.
* boolean execute(); --执行select返回true 执行其他的语句返回false.

1. 设置实际参数

* void **setXxx**(int index, Xxx xx) 将指定参数设置为给定Java的xx值。在将此值发送到数据库时，驱动程序将它转换成一个 SQL Xxx类型值。

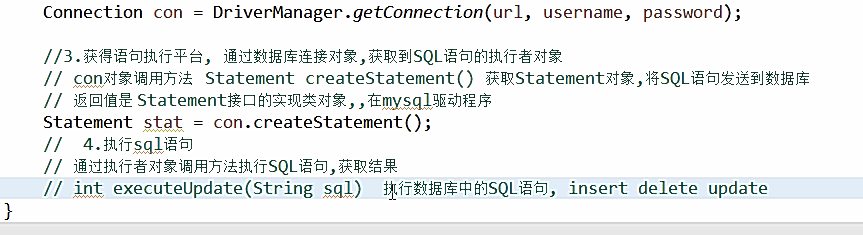
例如：

setString(2, "家用电器") 把SQL语句中第2个位置的占位符？ 替换成实际参数 "家用电器"

## 预处理对象executeUpdate方法

Int executeUpdate();

返回的是执行了多少行的数目



注意：excuteUpadate方法只能执行insert、delete、update

通过预处理对象的executeUpdate方法，完成记录的insert\update\delete语句的执行。操作格式统一如下：

1. 注册驱动

2. 获取连接

3. 获取预处理对象

4. SQL语句占位符设置实际参数

5. 执行SQL语句

6. 释放资源

### 插入记录：insert

* 实现向分类表中插入指定的新分类

**public** **void** demo01() **throws** Exception {

// 1注册驱动

Class.*forName*("com.mysql.jdbc.Driver");

// 2获取连接

Connection conn = DriverManager.*getConnection*("jdbc:mysql://localhost:3306/mydb", "root", "root");

// 3获得预处理对象

String sql = "insert into sort(sname) values(?)";

PreparedStatement stat = conn.prepareStatement(sql);

// 4 SQL语句占位符设置实际参数

stat.setString(1, "奢侈品");

// 5执行SQL语句

**int** line = stat.executeUpdate();

System.***out***.println("新添加记录数：" + line);

// 6释放资源

stat.close();

conn.close();

}

### 更新记录：update

* 实现更新分类表中指定分类ID所对应记录的分类名称

**public** **void** demo02() **throws** Exception {

// 1注册驱动

Class.*forName*("com.mysql.jdbc.Driver");

// 2获取连接

Connection conn = DriverManager.*getConnection*("jdbc:mysql://localhost:3306/mydb", "root", "root");

// 3获得预处理对象中

String sql = "update sort set sname=? where sid=?";

PreparedStatement stat = conn.prepareStatement(sql);

// 4 SQL语句占位符设置实际参数

stat.setString(1, "数码产品");

stat.setInt(2, 1);

// 5执行SQL语句

**int** line = stat.executeUpdate();

System.***out***.println("更新记录数：" + line);

// 6释放资源

stat.close();

conn.close();

}

### 删除记录：delete

* 实现删除分类表中指定分类ID的记录

**public** **void** demo03() **throws** Exception {

// 1注册驱动

Class.*forName*("com.mysql.jdbc.Driver");

// 2获取连接

Connection conn = DriverManager.*getConnection*("jdbc:mysql://localhost:3306/mydb", "root", "root");

// 3获得预处理对象

String sql = "delete from sort where sid=?";

PreparedStatement stat = conn.prepareStatement(sql);

// 4 SQL语句占位符设置实际参数

stat.setInt(1, 1);

// 5执行SQL语句

**int** line = stat.executeUpdate();

System.***out***.println("删除记录数：" + line);

// 6释放资源

stat.close();

conn.close();

}

## 预处理对象executeQuery方法

通过预处理对象的executeQuery方法，完成记录的select语句的执行。操作格式统一如下：

1. 注册驱动

2. 获取连接

3. 获取预处理对象

4. SQL语句占位符设置实际参数

5. 执行SQL语句

6. 处理结果集(遍历结果集合)

7. 释放资源

### 查询记录：select

* 实现查询分类表所有记录

**public** **void** demo04() **throws** Exception {

// 1注册驱动

Class.*forName*("com.mysql.jdbc.Driver");

// 2获取连接

Connection conn = DriverManager.*getConnection*("jdbc:mysql://localhost:3306/mydb", "root", "root");

// 3获得预处理对象

String sql = "select \* from sort";

PreparedStatement stat = conn.prepareStatement(sql);

// 4 SQL语句占位符设置实际参数

// 5执行SQL语句

ResultSet rs = stat.executeQuery();

// 6处理结果集(遍历结果集合)

**while**( rs.next() ){

//获取当前行的分类ID

String sid = rs.getString("sid");//方法参数为数据库表中的列名

//获取当前行的分类名称

String sname = rs.getString("sname");

//显示数据

System.***out***.println(sid+"-----"+sname);

}

// 7释放资源

rs.close();

stat.close();

conn.close();

}

* 实现查询分类表中指定分类名称的记录

**public** **void** demo05() **throws** Exception {

// 1注册驱动

Class.*forName*("com.mysql.jdbc.Driver");

// 2获取连接

Connection conn = DriverManager.*getConnection*("jdbc:mysql://localhost:3306/mydb", "root", "root");

// 3获得预处理对象

String sql = "select \* from sort where sname=?";

PreparedStatement stat = conn.prepareStatement(sql);

// 4 SQL语句占位符设置实际参数

stat.setString(1, "奢侈品");

// 5执行SQL语句

ResultSet rs = stat.executeQuery();

// 6处理结果集(遍历结果集合)

**while**( rs.next() ){

//获取当前行的分类ID

String sid = rs.getString("sid");//方法参数为数据库表中的列名

//获取当前行的分类名称

String sname = rs.getString("sname");

//显示数据

System.***out***.println(sid+"-----"+sname);

}

// 7释放资源

rs.close();

stat.close();

conn.close();

}

# 数据层：

·数据层数据层既然是进行数据操作的，那么就将其保存在dao包中；

·不同的数据表可能使用不同的数据层进行开发，那么就有不同的命名方式。

## 命名方式：

|-emp表，那么数据层的接口就应该定义为IEmpDAO

解释：I表示接口Emp表示表，dao表示数据层的操作。

### |-数据更新：

建议它的操作方法以doXxx()形式命名，例如：修改：doUpdate(),创建：doCreate()，删除：doRemove();

### 数据查询：

|-查询表中的数据：findById()、findByName()、findAll();

|-统计表中的数据：getAllCount()

## 数据层接口：



示意图

# Response（响应）：

### HttpServletResponse概述

我们在创建Servlet时会覆盖service()方法，或doGet()/doPost(),这些方法都有两个参数，一个为代表请求的request和代表响应response。

service方法中的response的类型是ServletResponse，而doGet/doPost方法的response的类型是HttpServletResponse，HttpServletResponse是ServletResponse的子接口，功能和方法更加强大，今天我们学习HttpServletResponse。

### Response的运行流程：请求--响应请求

### HTTP响应例子抓包工具抓的：

3个部分

## 方法：1563802735(1)

### 通过response设置响应行

设置响应行的状态码

**[setStatus](mk:@MSITStore:D:\\api文档\\java_ee_api_中英文对照版.chm::/javax/servlet/http/../../../javax/servlet/http/HttpServletResponse.html" \l "setStatus(int))**(int sc)

### 通过response设置响应头

**[addHeader](mk:@MSITStore:D:\\api文档\\java_ee_api_中英文对照版.chm::/javax/servlet/http/../../../javax/servlet/http/HttpServletResponse.html" \l "addHeader(java.lang.String, java.lang.String))**([String](http://java.sun.com/j2se/1.5/docs/api/java/lang/String.html" \o "class or interface in java.lang) name, [String](http://java.sun.com/j2se/1.5/docs/api/java/lang/String.html" \o "class or interface in java.lang) value)

**[addIntHeader](mk:@MSITStore:D:\\api文档\\java_ee_api_中英文对照版.chm::/javax/servlet/http/../../../javax/servlet/http/HttpServletResponse.html" \l "addIntHeader(java.lang.String, int))**([String](http://java.sun.com/j2se/1.5/docs/api/java/lang/String.html" \o "class or interface in java.lang) name, int value)

**[addDateHeader](mk:@MSITStore:D:\\api文档\\java_ee_api_中英文对照版.chm::/javax/servlet/http/../../../javax/servlet/http/HttpServletResponse.html" \l "addDateHeader(java.lang.String, long))**([String](http://java.sun.com/j2se/1.5/docs/api/java/lang/String.html" \o "class or interface in java.lang) name, long date)

**[setHeader](mk:@MSITStore:D:\\api文档\\java_ee_api_中英文对照版.chm::/javax/servlet/http/../../../javax/servlet/http/HttpServletResponse.html" \l "setHeader(java.lang.String, java.lang.String))**([String](http://java.sun.com/j2se/1.5/docs/api/java/lang/String.html" \o "class or interface in java.lang) name, [String](http://java.sun.com/j2se/1.5/docs/api/java/lang/String.html" \o "class or interface in java.lang) value)

**[setDateHeader](mk:@MSITStore:D:\\api文档\\java_ee_api_中英文对照版.chm::/javax/servlet/http/../../../javax/servlet/http/HttpServletResponse.html" \l "setDateHeader(java.lang.String, long))**([String](http://java.sun.com/j2se/1.5/docs/api/java/lang/String.html" \o "class or interface in java.lang) name, long date)

**[setIntHeader](mk:@MSITStore:D:\\api文档\\java_ee_api_中英文对照版.chm::/javax/servlet/http/../../../javax/servlet/http/HttpServletResponse.html" \l "setIntHeader(java.lang.String, int))**([String](http://java.sun.com/j2se/1.5/docs/api/java/lang/String.html" \o "class or interface in java.lang) name, int value)

其中，add表示添加，而set表示设置

### 通过response设置响应体

#### 响应体设置文本

[PrintWriter](http://java.sun.com/j2se/1.5/docs/api/java/io/PrintWriter.html" \o "class or interface in java.io) **[getWriter](mk:@MSITStore:D:\\api文档\\java_ee_api_中英文对照版.chm::/javax/servlet/../../javax/servlet/ServletResponse.html" \l "getWriter())**()

获得字符流，通过字符流的**[write](http://docs.oracle.com/javase/1.5.0/docs/api/java/io/../../java/io/PrintWriter.html" \l "write(java.lang.String))**([String](http://docs.oracle.com/javase/1.5.0/docs/api/java/io/../../java/lang/String.html" \o "class in java.lang) s)方法可以将字符串设置到response 缓冲区中，随后Tomcat会将response缓冲区中的内容组装成Http响应返回给浏览 器端。

关于设置中文的乱码问题

原因：response缓冲区的默认编码是iso8859-1，此码表中没有中文，可以通过 response的**[setCharacterEncoding](mk:@MSITStore:D:\\api文档\\java_ee_api_中英文对照版.chm::/javax/servlet/../../javax/servlet/ServletResponse.html" \l "setCharacterEncoding(java.lang.String))**([String](http://java.sun.com/j2se/1.5/docs/api/java/lang/String.html" \o "class or interface in java.lang) charset) 设置response的编码

但我们发现客户端还是不能正常显示文字

原因：我们将response缓冲区的编码设置成UTF-8，但浏览器的默认编码是本地系 统的编码，因为我们都是中文系统，所以客户端浏览器的默认编码是GBK，我们可以 手动修改浏览器的编码是UTF-8。

我们还可以在代码中指定浏览器解析页面的编码方式，

通过response的**[setContentType](mk:@MSITStore:D:\\api文档\\java_ee_api_中英文对照版.chm::/javax/servlet/../../javax/servlet/ServletResponse.html" \l "setContentType(java.lang.String))**([String](http://java.sun.com/j2se/1.5/docs/api/java/lang/String.html" \o "class or interface in java.lang) type)方法指定页面解析时的编码是UTF-8

response.setContentType("text/html;charset=UTF-8");

上面的代码不仅可以指定浏览器解析页面时的编码，同时也内含 setCharacterEncoding的功能，所以在实际开发中只要编写 **response.setContentType("text/html;charset=UTF-8");**就可以解决页面输出中文乱码问题。

#### 响应头设置字节

[ServletOutputStream](mk:@MSITStore:D:\\api文档\\java_ee_api_中英文对照版.chm::/javax/servlet/../../javax/servlet/ServletOutputStream.html" \o "class in javax.servlet) **[getOutputStream](mk:@MSITStore:D:\\api文档\\java_ee_api_中英文对照版.chm::/javax/servlet/../../javax/servlet/ServletResponse.html" \l "getOutputStream())**()

获得字节流，通过该字节流的write(byte[] bytes)可以向response缓冲区中写入字 节，在由Tomcat服务器将字节内容组成Http响应返回给浏览器。

### EG：文件下载（即文件的拷贝）

文件下载的实质就是文件拷贝，将文件从服务器端拷贝到浏览器端。所以文件下载需 要IO技术将服务器端的文件使用InputStream读取到，在使用 ServletOutputStream写到response缓冲区中

上述代码可以将图片从服务器端传输到浏览器，但浏览器直接解析图片显示在页面上， 而不是提供下载，我们需要设置两个响应头，告知浏览器文件的类型和文件的打开方 式。

1. 告知浏览器文件的类型：response.setContentType(文件的MIME类型);
2. 告示浏览器文件的打开方式是下载：

response.setHeader("Content-Disposition","attachment;filename=文件名称");

代码如下：



但是，如果下载中文文件，页面在下载时会出现中文乱码或不能显示文件名的情况， 原因是不同的浏览器默认对下载文件的编码方式不同，ie是UTF-8编码方式，而火狐 浏览器是Base64编码方式。所里这里需要解决浏览器兼容性问题，解决浏览器兼容 性问题的首要任务是要辨别访问者是ie还是火狐（其他），通过Http请求体中的一 个属性可以辨别

解决乱码方法如下（不要记忆--了解）：

if (agent.contains("MSIE")) {

// IE浏览器

filename = URLEncoder.encode(filename, "utf-8");

filename = filename.replace("+", " ");

} else if (agent.contains("Firefox")) {

// 火狐浏览器

BASE64Encoder base64Encoder = new BASE64Encoder();

filename = "=?utf-8?B?"

+ base64Encoder.encode(filename.getBytes("utf-8")) + "?=";

} else {

// 其它浏览器

filename = URLEncoder.encode(filename, "utf-8");

}

其中agent就是请求头User-Agent的值





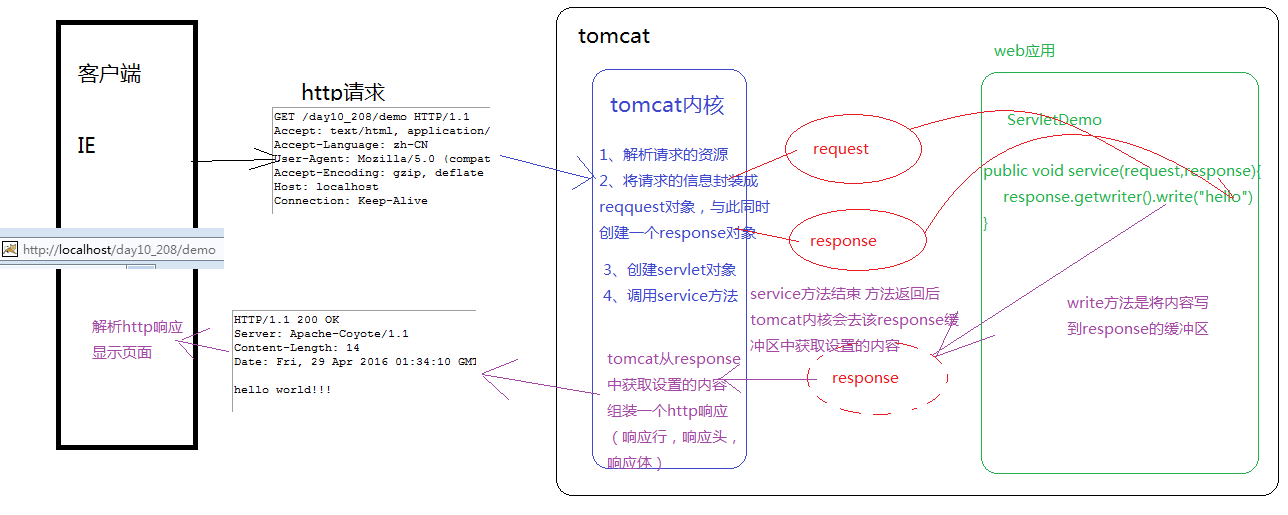
# Request(请求)

### HttpServletRequest概述

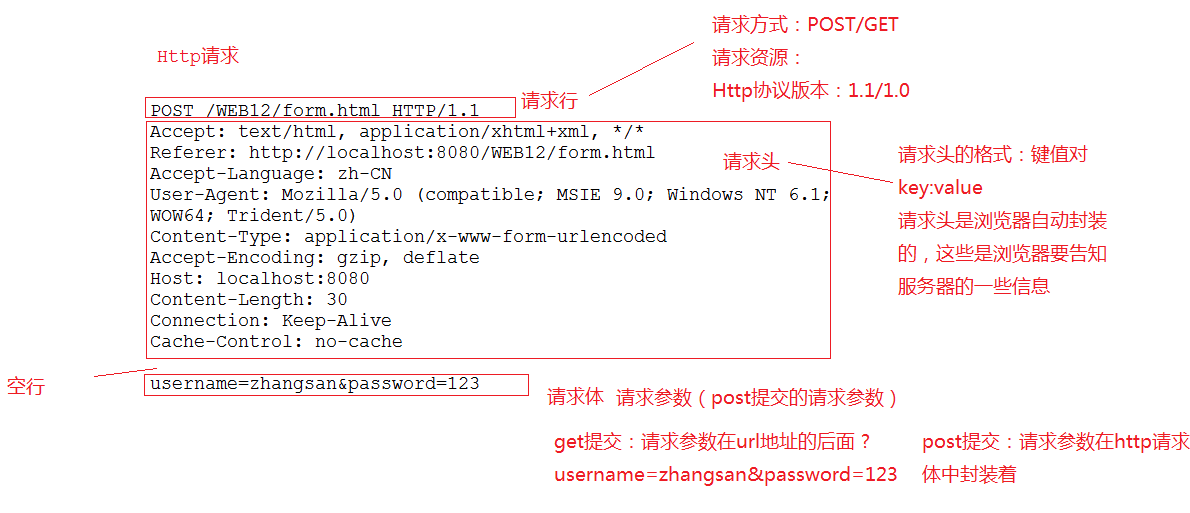
我们在创建Servlet时会覆盖service()方法，或doGet()/doPost(),这些方法都有两个参数，一个为代表请求的request和代表响应response。

service方法中的request的类型是ServletRequest，而doGet/doPost方法的request的类型是HttpServletRequest，HttpServletRequest是ServletRequest的子接口，功能和方法更加强大，今天我们学习HttpServletRequest。

### request的运行流程



### 通过抓包工具抓取Http请求



因为request代表请求，所以我们可以通过该对象分别获得Http请求的请求行，请 求头和请求体

### 通过request获得请求行

获得客户端的请求方式：[String](http://java.sun.com/j2se/1.5/docs/api/java/lang/String.html" \o "class or interface in java.lang) **[getMethod](mk:@MSITStore:D:\\api文档\\java_ee_api_中英文对照版.chm::/javax/servlet/http/../../../javax/servlet/http/HttpServletRequest.html" \l "getMethod())**()

获得请求的资源：

[String](http://java.sun.com/j2se/1.5/docs/api/java/lang/String.html" \o "class or interface in java.lang) **[getRequestURI](mk:@MSITStore:D:\\api文档\\java_ee_api_中英文对照版.chm::/javax/servlet/http/../../../javax/servlet/http/HttpServletRequest.html" \l "getRequestURI())**()

[StringBuffer](http://java.sun.com/j2se/1.5/docs/api/java/lang/StringBuffer.html" \o "class or interface in java.lang) **[getRequestURL](mk:@MSITStore:D:\\api文档\\java_ee_api_中英文对照版.chm::/javax/servlet/http/../../../javax/servlet/http/HttpServletRequest.html" \l "getRequestURL())**()

[String](http://java.sun.com/j2se/1.5/docs/api/java/lang/String.html" \o "class or interface in java.lang) **[getContextPath](mk:@MSITStore:D:\\api文档\\java_ee_api_中英文对照版.chm::/javax/servlet/http/../../../javax/servlet/http/HttpServletRequest.html" \l "getContextPath())**() ---web应用的名称

[String](http://java.sun.com/j2se/1.5/docs/api/java/lang/String.html" \o "class or interface in java.lang) **[getQueryString](mk:@MSITStore:D:\\api文档\\java_ee_api_中英文对照版.chm::/javax/servlet/http/../../../javax/servlet/http/HttpServletRequest.html" \l "getQueryString())**() ---- get提交url地址后的参数字符串

username=zhangsan&password=123

注意：request获得客户机（客户端）的一些信息

request.getRemoteAddr() --- 获得访问的客户端IP地址

### 通过request获得请求头

long **[getDateHeader](mk:@MSITStore:D:\\api文档\\java_ee_api_中英文对照版.chm::/javax/servlet/http/../../../javax/servlet/http/HttpServletRequest.html" \l "getDateHeader(java.lang.String))**([String](http://java.sun.com/j2se/1.5/docs/api/java/lang/String.html" \o "class or interface in java.lang) name)

[String](http://java.sun.com/j2se/1.5/docs/api/java/lang/String.html" \o "class or interface in java.lang) **[getHeader](mk:@MSITStore:D:\\api文档\\java_ee_api_中英文对照版.chm::/javax/servlet/http/../../../javax/servlet/http/HttpServletRequest.html" \l "getHeader(java.lang.String))**([String](http://java.sun.com/j2se/1.5/docs/api/java/lang/String.html" \o "class or interface in java.lang) name)

[Enumeration](http://java.sun.com/j2se/1.5/docs/api/java/util/Enumeration.html" \o "class or interface in java.util) **[getHeaderNames](mk:@MSITStore:D:\\api文档\\java_ee_api_中英文对照版.chm::/javax/servlet/http/../../../javax/servlet/http/HttpServletRequest.html" \l "getHeaderNames())**()

[Enumeration](http://java.sun.com/j2se/1.5/docs/api/java/util/Enumeration.html" \o "class or interface in java.util) **[getHeaders](mk:@MSITStore:D:\\api文档\\java_ee_api_中英文对照版.chm::/javax/servlet/http/../../../javax/servlet/http/HttpServletRequest.html" \l "getHeaders(java.lang.String))**([String](http://java.sun.com/j2se/1.5/docs/api/java/lang/String.html" \o "class or interface in java.lang) name)

int **[getIntHeader](mk:@MSITStore:D:\\api文档\\java_ee_api_中英文对照版.chm::/javax/servlet/http/../../../javax/servlet/http/HttpServletRequest.html" \l "getIntHeader(java.lang.String))**([String](http://java.sun.com/j2se/1.5/docs/api/java/lang/String.html" \o "class or interface in java.lang) name)

referer头的作用：执行该此访问的的来源

做防盗链

### 通过request获得请求体

请求体中的内容是通过post提交的请求参数，格式是：

username=zhangsan&password=123&hobby=football&hobby=basketball

key ---------------------- value

username [zhangsan]

password [123]

hobby [football，basketball]

以上面参数为例，通过一下方法获得请求参数：

[String](http://java.sun.com/j2se/1.5/docs/api/java/lang/String.html" \o "class or interface in java.lang) **[getParameter](mk:@MSITStore:D:\\api文档\\java_ee_api_中英文对照版.chm::/javax/servlet/../../javax/servlet/ServletRequest.html" \l "getParameter(java.lang.String))**([String](http://java.sun.com/j2se/1.5/docs/api/java/lang/String.html" \o "class or interface in java.lang) name)

[String](http://java.sun.com/j2se/1.5/docs/api/java/lang/String.html" \o "class or interface in java.lang)[] **[getParameterValues](mk:@MSITStore:D:\\api文档\\java_ee_api_中英文对照版.chm::/javax/servlet/../../javax/servlet/ServletRequest.html" \l "getParameterValues(java.lang.String))**([String](http://java.sun.com/j2se/1.5/docs/api/java/lang/String.html" \o "class or interface in java.lang) name)

[Enumeration](http://java.sun.com/j2se/1.5/docs/api/java/util/Enumeration.html" \o "class or interface in java.util) **[getParameterNames](mk:@MSITStore:D:\\api文档\\java_ee_api_中英文对照版.chm::/javax/servlet/../../javax/servlet/ServletRequest.html" \l "getParameterNames())**()

[Map](http://java.sun.com/j2se/1.5/docs/api/java/util/Map.html" \o "class or interface in java.util)<String,String[]> **[getParameterMap](mk:@MSITStore:D:\\api文档\\java_ee_api_中英文对照版.chm::/javax/servlet/../../javax/servlet/ServletRequest.html" \l "getParameterMap())**()

注意：get请求方式的请求参数 上述的方法一样可以获得

解决post提交方式的乱码：request.setCharacterEncoding("UTF-8");

解决get提交的方式的乱码：

parameter = new String(parameter.getbytes("iso8859-1"),"utf-8");

### request的其他功能

#### request是一个域对象

request对象也是一个存储数据的区域对象，所以也具有如下方法：

**[setAttribute](mk:@MSITStore:D:\\api文档\\java_ee_api_中英文对照版.chm::/javax/servlet/../../javax/servlet/ServletRequest.html" \l "setAttribute(java.lang.String, java.lang.Object))**([String](http://java.sun.com/j2se/1.5/docs/api/java/lang/String.html" \o "class or interface in java.lang) name, [Object](http://java.sun.com/j2se/1.5/docs/api/java/lang/Object.html" \o "class or interface in java.lang) o)

**[getAttribute](mk:@MSITStore:D:\\api文档\\java_ee_api_中英文对照版.chm::/javax/servlet/../../javax/servlet/ServletRequest.html" \l "getAttribute(java.lang.String))**([String](http://java.sun.com/j2se/1.5/docs/api/java/lang/String.html" \o "class or interface in java.lang) name)

**[removeAttribute](mk:@MSITStore:D:\\api文档\\java_ee_api_中英文对照版.chm::/javax/servlet/../../javax/servlet/ServletRequest.html" \l "removeAttribute(java.lang.String))**([String](http://java.sun.com/j2se/1.5/docs/api/java/lang/String.html" \o "class or interface in java.lang) name)

注意：request域的作用范围：一次请求中

#### request完成请求转发

获得请求转发器----path是转发的地址

[RequestDispatcher](mk:@MSITStore:D:\\api文档\\java_ee_api_中英文对照版.chm::/javax/servlet/../../javax/servlet/RequestDispatcher.html" \o "interface in javax.servlet) **[getRequestDispatcher](mk:@MSITStore:D:\\api文档\\java_ee_api_中英文对照版.chm::/javax/servlet/../../javax/servlet/ServletRequest.html" \l "getRequestDispatcher(java.lang.String))**([String](http://java.sun.com/j2se/1.5/docs/api/java/lang/String.html" \o "class or interface in java.lang) path)

通过转发器对象转发

**requestDispathcer.[forward](mk:@MSITStore:D:\\api文档\\java_ee_api_中英文对照版.chm::/javax/servlet/../../javax/servlet/RequestDispatcher.html" \l "forward(javax.servlet.ServletRequest, javax.servlet.ServletResponse))**([ServletRequest](mk:@MSITStore:D:\\api文档\\java_ee_api_中英文对照版.chm::/javax/servlet/../../javax/servlet/ServletRequest.html" \o "interface in javax.servlet) request, [ServletResponse](mk:@MSITStore:D:\\api文档\\java_ee_api_中英文对照版.chm::/javax/servlet/../../javax/servlet/ServletResponse.html" \o "interface in javax.servlet) response)

注意：ServletContext域与Request域的生命周期比较？

ServletContext：

创建：服务器启动

销毁：服务器关闭

域的作用范围：整个web应用

request：

创建：访问时创建request

销毁：响应结束request销毁

域的作用范围：一次请求中

注意：转发与重定向的区别？

1）重定向两次请求，转发一次请求

2）重定向地址栏的地址变化，转发地址不变

3）重新定向可以访问外部网站 转发只能访问内部资源

4）转发的性能要优于重定向

注意：客户端地址与服务器端地址的写法？

客户端地址：

是客户端去访问服务器的地址，服务器外部的地址，特点：写上web应用名 称

直接输入地址：

重定向

服务器端地址：

服务器内部资源的跳转的地址，特点：不需要写web应用的名称

转发

总结：

request获得行的内容

request.getMethod()

request.getRequestURI()

request.getRequestURL()

request.getContextPath()

request.getRemoteAddr()

request获得头的内容

request.getHeader(name)

request获得体（请求参数）

String request.getParameter(name)

Map<String,String[]> request.getParameterMap();

String[] request.getParameterValues(name);

注意：客户端发送的参数 到服务器端都是字符串

获得中文乱码的解决：

post:request.setCharacterEncoding(“UTF-8”);

get:

parameter = new String(parameter.getBytes(“iso8859-1”),”UTF-8”);

request转发和域

request.getRequestDispatcher(转发的地址).forward(req,resp);

request.setAttribute(name,value)

request.getAttribute(name)

# 动态代理和cglib代理：

动态代理必须要接口（多用）

cglib可以对任何生成代理对象