目录

[2015年8月11日09:17:55 4](#_Toc458981515)

[1. 第一个java程序 4](#_Toc458981516)

[2015年9月1日10:44:29 4](#_Toc458981517)

[java的字符串的操作 4](#_Toc458981518)

[2015年9月1日21:58:29 6](#_Toc458981519)

[1. 引用传递 6](#_Toc458981520)

[2015年9月2日10:47:53 6](#_Toc458981521)

[1. Main方法可以输入多个参数 6](#_Toc458981522)

[2015年9月9日09:47:18 8](#_Toc458981523)

[2015-9-22 20:37:49 8](#_Toc458981524)

[2015年10月14日20:51:25 8](#_Toc458981525)

[2015年12月4日22:05:02 8](#_Toc458981526)

[2015年12月7日20:08:55 8](#_Toc458981527)

[2016年6月22日10:26:17 10](#_Toc458981528)

[2016年6月22日23:11:51 10](#_Toc458981529)

[2016年6月28日16:54:56 10](#_Toc458981530)

[1、 StringBuffer和StringBuilder 10](#_Toc458981531)

[2. 线程的停止等旧方法已经过时 10](#_Toc458981532)

[2016年6月28日23:40:59 11](#_Toc458981533)

[1.定义泛型对象的时候注意前面也要指定类名称 11](#_Toc458981534)

[2016年6月29日22:21:13 11](#_Toc458981535)

[1. 泛型的通配符是？ 11](#_Toc458981536)

[2. 泛型的上下限 11](#_Toc458981537)

[3. 子类的泛型无法使用父类的泛型接收 12](#_Toc458981538)

[2016年7月12日12:06:48 12](#_Toc458981539)

[1. StringBuffer类 12](#_Toc458981540)

[2. Runtime类 12](#_Toc458981541)

[3. java国际化 13](#_Toc458981542)

[3.1 Locale类 13](#_Toc458981543)

[3.2 ResourceBundle类 13](#_Toc458981544)

[4. System类 13](#_Toc458981545)

[4.1获取当前时间 14](#_Toc458981546)

[4.2 获取系统属性 14](#_Toc458981547)

[4.3 垃圾回收 15](#_Toc458981548)

[5. 日期操作类 15](#_Toc458981549)

[5.1 Date类 15](#_Toc458981550)

[5.2 Calendar类 15](#_Toc458981551)

[6. 日期格式化 15](#_Toc458981552)

[6.1 DateFormat类 16](#_Toc458981553)

[6.2 SimpleDateFormat类 16](#_Toc458981554)

[7. Math类 17](#_Toc458981555)

[8. Random类 17](#_Toc458981556)

[9. NumberFormat类 17](#_Toc458981557)

[10 DecimalFormat 17](#_Toc458981558)

[11. 大数字 18](#_Toc458981559)

[11.1 BigInteger 18](#_Toc458981560)

[11.2 BigDecimal 18](#_Toc458981561)

[12 对象克隆 18](#_Toc458981562)

[13 数组Arrays类 19](#_Toc458981563)

[13.1 sort排序方法 19](#_Toc458981564)

[13.2 toString转换为字符串 19](#_Toc458981565)

[13.3 二分查询binarySearch 19](#_Toc458981566)

[13.4 数组填充 fill 19](#_Toc458981567)

[14 Comparable接口 20](#_Toc458981568)

[15 Comparator接口 21](#_Toc458981569)

[16 Observer观察者设计模式 22](#_Toc458981570)

[2016年8月4日15:07:34 23](#_Toc458981571)

[1.正则表达式 23](#_Toc458981572)

[1.1.1 判断日期格式 24](#_Toc458981573)

[1.1.2 替换 24](#_Toc458981574)

[2016年8月4日23:02:09 25](#_Toc458981575)

[1. JAVA IO操作 25](#_Toc458981576)

[1.1.创建一个新文件 25](#_Toc458981577)

[1.2 删除一个指定的文件 26](#_Toc458981578)

[1.3创建文件夹 26](#_Toc458981579)

[1.4 判断一个给定的路径是否是路径 26](#_Toc458981580)

[2016年8月5日17:08:03 26](#_Toc458981581)

[1、RandAccessFile类 26](#_Toc458981582)

[2. 字节流与字符流 27](#_Toc458981583)

[2.1字节流 28](#_Toc458981584)

[2.2字符流 29](#_Toc458981585)

[2016年8月7日16:25:38 29](#_Toc458981586)

[1.字节字符转换流 29](#_Toc458981587)

[2. 内存操作流 30](#_Toc458981588)

[3. 管道流 30](#_Toc458981589)

[4. 打印流 32](#_Toc458981590)

[5.System类对IO的支持 32](#_Toc458981591)

[6. 输入、输出重定向 34](#_Toc458981592)

[7. BufferedReader类 34](#_Toc458981593)

[2016年8月9日17:13:15 35](#_Toc458981594)

[1. Scanner类 35](#_Toc458981595)

[2016年8月9日22:55:37 36](#_Toc458981596)

[1. 数据操作流 36](#_Toc458981597)

[2. 合并流 36](#_Toc458981598)

[2016年8月10日10:56:29 37](#_Toc458981599)

[1.压缩流 37](#_Toc458981600)

[2016年8月11日18:46:41 37](#_Toc458981601)

[1. List接口 37](#_Toc458981602)

[2. LinkedList类 38](#_Toc458981603)

[2. Set接口 38](#_Toc458981604)

[2016年8月13日18:04:24 43](#_Toc458981605)

[1. Iterator接口 43](#_Toc458981606)

[2016年8月13日18:30:47 44](#_Toc458981607)

[1.ListIterator接口 44](#_Toc458981608)

[2. foreach 45](#_Toc458981609)

[2016年8月13日22:09:53 46](#_Toc458981610)

[1. Map接口 46](#_Toc458981611)

[2. Map.Entry接口 48](#_Toc458981612)

[2. TreeMap子类 48](#_Toc458981613)

[Map接口输出 49](#_Toc458981614)

[2.使用foreach输出 50](#_Toc458981615)

[2016年8月14日01:30:00 51](#_Toc458981616)

[1. IdentityHashMap类 51](#_Toc458981617)

[1. SortedMap类 52](#_Toc458981618)

[2. 集合工具类Collections 52](#_Toc458981619)

[2. Stack类 54](#_Toc458981620)

[3. Properties类 55](#_Toc458981621)

[4. 一对多关系 56](#_Toc458981622)

[5. 多对多关系 57](#_Toc458981623)

[2016年8月14日19:10:04 59](#_Toc458981624)

[1. 枚举 enum 59](#_Toc458981625)

[2. enum关键字和Enum类的关系 59](#_Toc458981626)

[3.类集对Enum的支持 61](#_Toc458981627)

[3.1 EnumMap 61](#_Toc458981628)

[3.2 EnumSet 62](#_Toc458981629)

[将全部的集合设置到EnumSet集合中 62](#_Toc458981630)

[只设置一个内容到集合 62](#_Toc458981631)

[创建只能放入指定枚举类型的集合 63](#_Toc458981632)

[创建不包含指定元素的集合 63](#_Toc458981633)

[拷贝一个集合的内容 63](#_Toc458981634)

[4. 枚举的其他应用 64](#_Toc458981635)

[2016年8月9日10:46:52 64](#_Toc458981636)

# 2015年8月11日09:17:55

JDK是java开发包,JRE是java运行环境，JVM是java虚拟机。安装JDK后会自带JRE，JVM也包含在JRE里面。JVM保证了java程序运行于平台无关。所有的java程序都运行在java虚拟机上。

安装完JDK后，还不能用java来编程，需要配置环境变量。主要是配置JAVA\_HOME为安装路径，还有path和classpath，path为java的exe可执行文件，后者为库文件，就是一些java包。

垃圾回收

在 Java 里面，垃圾回收是一个自动的系统行为，程序员不能控制垃圾回收的功  
能和行为。比如垃圾回收什么时候开始，什么时候结束，还有到底哪些资源需要回收等，都  
是程序员不能控制的。  
（2） ：有一些跟垃圾回收相关的方法，比如：System.gc() , 记住一点，调用这些方法，  
仅仅是在通知垃圾回收程序，至于垃圾回收程序运不运行，什么时候运行，都是无法控制的。  
（3） ：程序员可以通过设置对象为 null（后面会讲到）来标示某个对象不再被需要了，

## 1. 第一个java程序

public class test

{

public static void main(String[] args)

{

System.out.println("hello world");

}

}

Java的mian方法和C#的Main方法有区别。

Java的是小写m，C#是大写M,参数和C#控制台类似。C#的string可以小写。而java不行，必须大写String.

# 2015年9月1日10:44:29

## java的字符串的操作

字符串转换为char数组

用toCharArray方法

String str1= "hello";

**char** c[] = str1.toCharArray();

b. 将字符数组转换为string直接将数组放在构造函数的参数里，也可以指定转换部分数组为字符串

String str2= **new** String(c);

String str3 = **new** String(c,0,3); //部分转换

将字符串转换为byte数组

**byte** b[] = str1.getBytes() ; // 将字符串变为byte数组

System.***out***.println(**new** String(b)) ; // 将全部的byte数组变为字符串

System.***out***.println(**new** String(b,1,3)) ; // 将部分的byte数组变为字符串

d. 取出某一个字符charAt

String str1 = "hello" ; // 定义String对象

System.out.println(str1.charAt(3)) ; // 取出字符串中第四个字符

e.求字符串的长度，用length()方法(估计是对属性用了getter和setter封装)

str1.length()

f. 查找

System.out.println(str1.indexOf("c")) ; // 查到返回位置

System.out.println(str1.indexOf("c",3)) ; // 查到返回位置，从第4个位置开始查

//找

System.out.println(str1.indexOf("x")) ; // 没有查到返回-1

g． 去掉空格

String str4 = " hello " ; // 定义字符串

System.***out***.println(str4.trim()) ; // 去掉左右空格后输出

h. 截取字符串

String str1 = "hello world" ; // 定义字符串

System.out.println(str1.substring(6)) ; // 从第7个位置开始截取

System.out.println(str1.substring(0,5)) ; // 截取0~5个位置的内容

分割字符串

String str6 = "hello world" ; // 定义字符串

String s[] = str6.split(" ") ; // 按空格进行字符串的拆分

**for**(**int** i=0;i<s.length;i++){ // 循环输出

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

}

j. 转换大小写

System.out.println("将\"hello world\"转成大写：" + "hello world".toUpperCase()) ;

System.out.println("将\"HELLO WORLD\"转成小写：" + "HELLO WORLD".toLowerCase()) ;

k. 判断开头和结尾

String str1 = "\*\*HELLO" ; // 定义字符串

String str2 = "HELLO\*\*" ; // 定义字符串

if(str1.startsWith("\*\*")){ // 判断是否以“\*\*”开头

System.out.println("（\*\*HELLO）以\*\*开头") ;

}

if(str2.endsWith("\*\*")){ // 判断是否以“\*\*”结尾

System.out.println("（HELLO\*\*）以\*\*结尾") ;

}

l.判断内容是否相等

String str1 = "HELLO" ; // 定义字符串

String str2 = "hello" ; // 定义字符串

System.out.println("\"HELLO\" equals \"hello\" " + str1.equals(str2)) ;

System.out.println("\"HELLO\" equalsIgnoreCase \"hello\" "

+ str1.equalsIgnoreCase(str2)) ; // 不区分大小写的比较

j. 替换

String str = "hello" ; // 定义字符串

String newStr = str.replaceAll("l","x") ; // 现在将所有的l替换成x

System.out.println("替换之后的结果：" + newStr) ;

# 2015年9月1日21:58:29

## 1. 引用传递

public static void main(String [] args)

{

String str1="hello";

System.out.println(str1);

fun(str1);

System.out.println(str1);

}

public static void fun(String str2)

{

Str2="world";

}

输出：

hello

hello

虽然string是引用传递，但在java中string具有内容不可变行，一旦赋值该内存空间的内容就不变，如果给变量重新赋值就会新开辟空间。在调用fun时str1和str2都指向“hello”，然偶执行str2=“world”时，str指向了新的空间，该空间内容为world。而str1指向的内存空间不变还是“hello”

如果传递的不是string，是自定义的类，就是普通的引用传递了，内容就会变。如果是int类型，则不变。这些特性和C#是一样的。

1. 类中调用本类的对象

class Demo{ // 定义Demo类

private int temp = 30 ; // 声明temp属性并封装

public void fun(Demo d2){ // 接收本类的引用

d2.temp = 50 ; // 直接通过对象调用本类的私有属性

}

public int getTemp(){ // getter

return temp ;

}

public void setTemp(int t){ // setter

temp = t ;

}

};

public class RefDemo04{

public static void main(String args[]){

Demo d1 = new Demo() ; // 实例化Demo对象

d1.setTemp(50) ; // 只能通过setter方法修改内容

d1.fun(d1) ; // 此处把Demo的对象传回到自己的类中

System.out.println("temp = " + d1.getTemp()) ;

}

};

# 2015年9月2日10:47:53

## 1. Main方法可以输入多个参数

public class StaticDemo08{

public static void main(String args[]){

if(args.length!=3){ // 输入的参数如果不足3个，则出错

System.out.println("输入的参数不足三个，程序退出~") ;

System.exit(1) ; // 直接退出此程序

}

for(int i=0;i<args.length;i++){ // 循环输出输入的参数

System.out.println("第"+(i+1)+"个参数：" + args[i]) ;

}

}

};

代码块（普通块，构造块，静态块，）

构造块，构造块优先于构造方法之前执行，直接在类用用花括号括起就行。

class Demo{

{ // 直接在类中编写代码块，称为构造块

System.out.println("1、构造块。") ;

}

public Demo(){ // 定义构造方法

System.out.println("2、构造方法。") ;

}

};

public class CodeDemo02{

public static void main(String args[]){

new Demo() ; // 实例化对象

new Demo() ; // 实例化对象

new Demo() ; // 实例化对象

}

};

静态块，优先于主方法执行，在类中，优先于构造方法执行。用于为静态属性初始化。如果有多个对象产生，也只执行一次。而构造块每产生一次对象，执行一次构造块。

class Demo{

{ // 直接在类中编写代码块，称为构造块

System.out.println("1、构造块。") ;

}

static{ // 使用static，称为静态代码块

System.out.println("0、静态代码块") ;

}

public Demo(){ // 定义构造方法

System.out.println("2、构造方法。") ;

}

};

public class CodeDemo03{

static{ // 在主方法所在的类中定义静态块

System.out.println("在主方法所在类中定义的代码块") ;

}

public static void main(String args[]){

new Demo() ; // 实例化对象

new Demo() ; // 实例化对象

new Demo() ; // 实例化对象

}

};

不用main方法实现输出hello worl  
public class Test3

{ static

{

System.out.println("hello world");

}

public static void main(String [] args)

{

}

}

教程里面直接没有main方法则段代码，也能输出hello world，但是有报错，提示没有main方法，

实际测试，如果没有main方法这段直接不能输出结果。

构造方法私有化

用private修饰构造方法，则不能在外部实例化对象。但是可以在类的内部实例化对象。

常常用于单例模式。

class Test

{

private static Test t = new Test();

private Test()

{}

public static Test getInstance()

{

if(t==null)

{

t= new Test();

return t;

}

return t;

}

public void print()

{

System.out.println("hello");

}

}

# 2015年9月9日09:47:18

1. 子类重写父类的方法时，子类方法的访问权限不能比父类低。

权限大小 public > default > private

比如 父类的方法为public 子类如果不写(default)或者写为private则报错。

2. 通过super关键字可以从子类访问父类发方法。用于子类重写了父类的方法的时候。如果不加super,则成为了自己调用自己，成为了递归的死循环。

使用super也不一定要复写之后才使用。加上它可以明确表示是由父类来的方法。

3． 如果父类中的方法是private，子类为default，则不存在复写，调用super会出错。父类和子类中的这两个方法分别属于各自的。

# 2015-9-22 20:37:49

1. final 关键字

final关键字修饰的类不能被继承，修饰的方法不能被复写，修饰的变量不能被修改。

1. 用static加final 声明的常量为全局常量。

# 2015年10月14日20:51:25

1. Instanceof关键字

Instanceof运算符是用来在运行时指出对象是否是特定类的一个实例。通过返回一个布尔值来指出。

用于增加验证，以保证转型时，不会发生ClassCastException

 用法：  
result = object instanceof class  
参数：  
Result：布尔类型。  
Object：必选项。任意对象表达式。  
Class：必选项。任意已定义的对象类。

如果 B类继承A类

A a= new B();

那么a即是A的实例，也是B的实例。

# 2015年12月4日22:05:02

1. 子类和父类间的转换

B类继承A类

A a = new B();

那么a这个对象只能访问A类中的方法，但是如果B类有同名方法，则会调用同名B类中的方法，B类中独有的，a这个对象是访问不了的。

# 2015年12月7日20:08:55

1. 关于引用传递和值传递

String和int都是值传递，将其作为参数传递时都会开辟空间创建副本，也就是在一个方法中改变它的值是不能改变的。

类是引用传递，在一个方法的参数中传递一个类的对象，会改变原来对象的变量的值。不管这个类中的变量是string还是int都会变。

class Ref2

{

String temp="hello";

}

**public** **static** **void** tell2(Ref2 r2)

{

r2.temp="world";

}

主方法：

Ref2 r2 = **new** Ref2();

r2.temp="java";

System.***out***.println(r2.temp);

*tell2*(r2);

System.***out***.println(r2.temp);

1. 和C#一样 private修饰的变量，在类外是不能访问的，包括对象
2. 关于static关键字

在C#中static关键字修饰的变量，只能通过类名来访问，而不能通过对象来访问

但是在java中就不一样了，类名和对象都可以访问static修饰的变量，不管创建了几个对象，一旦有对象访问了这个static变量，并且修改了值，或者类名直接放了这个变量，修改了了值，那么以最后一次修改为准，所有对象的这个static变量都是相同的值。因此在java中static修饰的变量具有全局性，任何一个对象修改了它，其余对象再访问都是修改后的，也就是说创建对象的时候，不会为它开辟新的空间，它不属于某个固定对象。

专业解释： 1. 使用static声明属性

Static声明全局属性

2 使用static声明方法

直接通过类名调用

3 注意点

使用static方法的时候，只能访问static声明的方法和属性，而非static声明的属性和方法不能访问。

Static修饰的属性和方法最好用类名来调用（在java中也可以用对象来调用），否则会报警告。

1. 关于this关键字

This关键字用于表示本类对象，在本类中调用变量，在java和C#中是没有区别的

但是this用于调用重载的构造方法时原理一样，写法不一样。

在C#中this写在构造方法的后面并加冒号：

在java中this写在构造方法内部首行

C#： public string name;

public int age;

public People()

{

}

public People(string name)

{

this.name = name;

this.age = 22;

}

public People(string name, int age) :this(name)

{

this.age = age;

}

Java：

**public** People(String name)

{

**this**.name=name;

}

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

{

**this**(name);

**this**.age=age;

}

# 2016年6月22日10:26:17

1. Java多线程中获取当前线程用的是Thread.currentThread()

注意是currentThread()是个方法要带括号。

1. Java多线程可以用哦getName和setName的方法读取和设置线程名称，线程运行前，如果没有设置名称，那么系统会设置一个默认的名称。
2. java中要想实现多线程可以采用以下两种方式：

继承Thread类

实现Runable接口

3.1 java.lang包会在程序运行时自动导入，所以无需手动编写import语句

一个类继承了Thread类之后，那么此类就具备了多线程的功能

class MyThread extends Thread{

private String name;

public MyThread(String name)

{

this.name=name;

}

public void run()

{

for(int i=0;i<10;i++)

{

System.out.println(name+"运行，i="+i);

}

}

}

public class Test1

{

public static void main(String[] args)

{

MyThread mt1= new MyThread("线程1");

MyThread mt2= new MyThread("线程2");

mt1.run();

mt2.run();

}

}

上面的程序并没有达到并发执行的效果，因为上面的代码是通过对象的方式调用的，如果要启动一个现成，必须使用Thread.Start()方法。

修改为：

/\* mt1.run();

mt2.run();\*/

mt1.start();

mt2.start();

native关键字，表示的是一个由java调用本机操作系统函数的一个关键字，在java中运行程序调用本机操作系统的函数以完成特定的功能。

如果要想实现多线程的话，则肯定需要操作系统支持，因为多线程操作中牵扯到一个抢占CPU的情况，要等待cpu进行调度，那么这一点肯定需要操作系统支持，所以使用了native调用本机的系统函数。而且在各个操作系统中多线程的实现底层代码肯定是不同的，所以使用native关键字也可以让JVM自动去调整不同的JVM实现。

threadStatus也表示一种状态，如果线程已经启动了再调用start()方法就有可能产生异常。

3.2 实现runable接口

如果想要启动线程则肯定要依靠Thread类，如果继承了Thread类，则可以将start方法直接继承下来并使用，但是在Runnable中并没有start方法。

Thread的构造方法里面有接收runnable实例的构造方法。

class MyThread implements Runnable{

private String name;

public MyThread(String name)

{

this.name=name;

}

public void run()

{

for(int i=0;i<10;i++)

{

System.out.println(name+"运行，i="+i);

}

}

}

public class Tes2

{

public static void main(String[] args)

{

MyThread mt1= new MyThread("线程1");

MyThread mt2= new MyThread("线程2");

Thread t1= new Thread(mt1);

Thread t2= new Thread(mt2);

t1.start();

t2.start();

}

}

3.3 Thread类和Runnable接口

thread类也是runnable接口的子类

使用thread类再操作多线程的时候无法达到资源共享的目的，而使用Rnnable接口实现的多线程操作可以实现多线程操作。

4． 使用同步问题

要想解决资源共享的同步操作问题，可以使用同步代码块及同步方法两种方式完成。

* 1. 同步代码块

使用synchronized关键字声明的代码块，称为同步。

同步的时候，必须指明同步的对象，一般情况下会将当前对象作为同步的对象，使用this表示。

* 1. 同步方法

public synchronized 返回值 方法名（）

5 程序中过多的同步会产生死锁

资源共享的时候需要同步操作

# 2016年6月22日23:11:51

* + - 1. 一个java程序运行的时候至少启动了两个线程，一个是主线程，另一个是GC线程，及垃圾回收线程。
      2. 通过isAlive()方法来判断线程是否执行完毕。在调用start方法之前返回false，在执行过程中返回true，线程执行完返回false。
      3. 线程的强制运行

调用线程对象的join方法，如果多个线程同时在运行，那么如果某个线程调用了join方法，那么它将被强制执行，即优先执行它，其余的线程等待，直到它运行完了才继续运行。

注意要捕获InterrupetedException异常。

4. 线程的休眠

调用sleep()方法

5. 程的中断

调用interrupt()方法,进入中断异常

6.线程的礼让

调用yield()方法

# 2016年6月28日16:54:56

## 1、 StringBuffer和StringBuilder

主要区别是StringBuffer类的方法是线程安全的，而StringBuilder不是线程安全的。

String的对象一旦创建，其内容不能再被修改，而StringBuffer和StringBuilder可以修改。

## 2. 线程的停止等旧方法已经过时

比如suspend，stop，resume方法已经过时，不建议使用，因为使用它们会造成死锁。

一个线程的停止是run方法运行完毕就直接停止了，不需要别的干预。一般run方法里面如果是个死循环，控制循环标志来退出循环，即结束线程。

# 2016年6月28日23:40:59

## 1.定义泛型对象的时候注意前面也要指定类名称

Point<Integer> p = new Point<Integer>() ;

# 2016年6月29日22:21:13

## 1. 泛型的通配符是？

当给一个方法传递一个泛型对象的时候，在写形参的时候，形参可以直接用类名加对象名来表示，也可以用<>指定实际的泛型类型。

比如有泛型类

class Info<T>

{

T var;

public void setVar(T var)

{

this.var=var;

}

public T getVar()

{

return this.var;

}

public String toString()

{

return this.var.toString();

}

}

方法：

public static void fun(Info temp)

{

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

System.out.println(temp) ;

}

public static void fun(Info<String> temp)

{

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

System.out.println(temp) ;

}

这两种方法都可以，但都没有达到泛型的特点，可以用？表示任意类型

public static void fun(Info<?> temp)

{

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

System.out.println(temp) ;

}

这样传递任意类型都可以

## 2. 泛型的上下限

可以设置泛型类型的上下限，即它的类型继承的最高等级和最低等级的类型，比如上限类型为Number，那么该类型只能是Number类和其子类。下限表示必须是该类是父类。声明上限用extends，下限用super。

设置上限：

声明对象：类名称<? extends 类> 对象名称

定义类：[访问权限] 类名称<泛型标示 extends 类>{}

设置下限：

声明对象：类名称<? super类> 对象名称

定义类：[访问权限] 类名称<泛型标示 super 类>{}

## 3. 子类的泛型无法使用父类的泛型接收

比如Info<String>不能用Info<Object>来接收

# 2016年7月12日12:06:48

## 1. StringBuffer类

类型可变的字符串

(1).连接字符串用append方法，在String中是用+号。append方法可以连续使用。

public static void main(String [] args)

{

StringBuffer buf = new StringBuffer();

buf.append("hello");

buf.append(" world").append("!!!");

buf.append('\n');

buf.append('C');

buf.append(true);

fun(buf);

System.out.println(buf);

}

public static void fun(StringBuffer s)

{

s.append("guo").append("zhang");

}

(2)插入字符串用insert

StringBuffer buf = new StringBuffer();

buf.append("hello");

buf.insert(0,"world");

(3)字符串反转功能reverse

StringBuffer buf1= buf.reverse();

buf和buf1都是反转后的结果。

(4)替换指定范围的内容replace

StringBuffer buf = new StringBuffer();

buf.append("hello").append("whorld");

buf.replace(6,11,"guozhang");

(5)字符串截取substring

该方法返回值直接是String

String res= buf.substring(2,4);

截取索引2,3，不截取4。

(6)删除指定范围的内容delete，该方法返回stringBuffer

buf.delete(2,4)

buf也被修改了。

(7)查找指定的内容是否存在

使用indexof来查找，和string是一样的，如果找到，返回指定的位置，没有的话返回-1.

int index = buf.indexOf("llo");

(8)stringbuffer的应用，用在需要频繁修改字符串内容的地方。比如很多次的for循环。

(9)StringBuffer中有而string中却没有的方法，比如：

insert，delete

## 2. Runtime类

运行时类，是一个封装了JVM进程的类，每一个java程序，实际上都是启动了一个JVM进程，那么每一个JVM进程都是对应的一个Runtime实例，此实例是又JVM为其实例化的。

(1)获得JVM信息

Runtime run = Runtime.getRuntime(); // 通过Runtime类的静态方法进行实例化操作

System.out.println("JVM最大内存量：" + run.maxMemory()) ; // 观察最大的内存，根据机器的不同，环境也会有所不同

System.out.println("JVM空闲内存量：" + run.freeMemory()) ; // 取得程序运行的空闲内存

run.gc() ; // 进行垃圾收集，释放空间

(2)打开系统程序

Runtime run = Runtime.getRuntime();

try{

run.exec("notepad.exe");

}catch(Exception e)

{

e.printStackTrace();

}

这段代码打开了系统的记事本。

exec方法返回进程Process类的对象

调用destroy方法可以结束进程。

## 3. java国际化

### 3.1 Locale类

### 3.2 ResourceBundle类

此类是专门完成属性文件的读取操作的，读取的时候指定文件名称即可(此文件名称一般不需要指定后缀，后缀统一为：\*.properties),可以根据Local所指定的区域码来自动选择所需要的资源文件。

public static final ResourceBundle getBundle(String baseName)此方法就是指定所有啊操作的资源文件，此方法找到的是默认的操作系统的语言的Local对象。

public static final ResourceBundle getBundle(String baseName，Locale local)此方法也是擦操作指定的资源文件，并传入Locale对象。

public static final String getString(Strig key) 根据key获取相应的value

例子：

新建一个文本文件，命名为Message.Properties，在里面写入inof=HELLO

在同一目录下新建java程序

ResourceBundle rb = ResourceBundle.getBundle("Message");

System.out.println(rb.getString("info"));

将输出:HELLO

3.3 利用ResourceBundle和Local实现国际化

不同的资源文件要建不同的properties文件，

中文：Message\_zh\_CN.properties

英文：Message\_en\_US.properties

法文：Message\_fr\_FR.properties

记得导入包；

import java.util.ResourceBundle;

import java.util.Locale;

Locale znloc = new Locale("zh","CN");

Locale enloc = new Locale("en","US");

ResourceBundle rb1 = ResourceBundle.getBundle("Message",znloc);

ResourceBundle rb2 = ResourceBundle.getBundle("Message",enloc);

System.out.println(rb1.getString("info"));

System.out.println(rb2.getString("info"));

在中文操作系统中，输出中文是乱码，因为zh\_CN的资源文件中直接 使用的是“你好”字符串，正确的应该是 用unicode编码，转换为java认识的16进制。

使用jdk自带的转换工具C:\Program Files\Java\jdk1.8.0\_65\bin\ native2ascii.exe,不同的语言环境转换结果不同，中文抽中系统下转换结果是：\u4f60\u597d

记住：只要是中文，就必须进行转码。

## 4. System类

### 4.1获取当前时间

long time1 = System.currentTimeMillis();

### 4.2 获取系统属性

获取全部属性：

System.getProperties().list(System.out);

-- listing properties --

java.runtime.name=Java(TM) SE Runtime Environment

sun.boot.library.path=C:\Program Files\Java\jre1.8.0\_92\bin

java.vm.version=25.92-b14

java.vm.vendor=Oracle Corporation

java.vendor.url=http://java.oracle.com/

path.separator=;

java.vm.name=Java HotSpot(TM) 64-Bit Server VM

file.encoding.pkg=sun.io

user.script=

user.country=CN

sun.java.launcher=SUN\_STANDARD

sun.os.patch.level=

java.vm.specification.name=Java Virtual Machine Specification

user.dir=D:\javatest

java.runtime.version=1.8.0\_92-b14

java.awt.graphicsenv=sun.awt.Win32GraphicsEnvironment

java.endorsed.dirs=C:\Program Files\Java\jre1.8.0\_92\lib...

os.arch=amd64

java.io.tmpdir=C:\Users\guozhang\AppData\Local\Temp\

line.separator=

java.vm.specification.vendor=Oracle Corporation

user.variant=

os.name=Windows 10

sun.jnu.encoding=GBK

java.library.path=C:\ProgramData\Oracle\Java\javapath;C...

java.specification.name=Java Platform API Specification

java.class.version=52.0

sun.management.compiler=HotSpot 64-Bit Tiered Compilers

os.version=10.0

user.home=C:\Users\guozhang

user.timezone=

java.awt.printerjob=sun.awt.windows.WPrinterJob

file.encoding=GBK

java.specification.version=1.8

user.name=guozhang

java.class.path=.;C:\Program Files\Java\jdk1.8.0\_65\l...

java.vm.specification.version=1.8

sun.arch.data.model=64

java.home=C:\Program Files\Java\jre1.8.0\_92

sun.java.command=SystemDemo02

java.specification.vendor=Oracle Corporation

user.language=zh

awt.toolkit=sun.awt.windows.WToolkit

java.vm.info=mixed mode

java.version=1.8.0\_92

java.ext.dirs=C:\Program Files\Java\jre1.8.0\_92\lib...

sun.boot.class.path=C:\Program Files\Java\jre1.8.0\_92\lib...

sun.stderr.encoding=ms936

java.vendor=Oracle Corporation

file.separator=\

java.vendor.url.bug=http://bugreport.sun.com/bugreport/

sun.cpu.endian=little

sun.io.unicode.encoding=UnicodeLittle

sun.stdout.encoding=ms936

sun.desktop=windows

sun.cpu.isalist=amd64

获取部分属性：类似键值对来查找

获取系统版本：

System.out.println("系统版本："+System.getProperty("os.name"));

os.version

os.arch

### 4.3 垃圾回收

一个对象如果不使用，则肯定要等待进行垃圾收集，垃圾收集可以自动调用也可以手工调用，手工调用的时候就是调用System.gc()或者Runtime.getRuntime().gc()

class Person{

private String name;

private int age;

public Person(String name,int age)

{

this.name=name;

this.age=age;

}

public String toString()

{

return "姓名："+this.name+"年龄："+this.age;

}

public void finalize() throws Throwable{

System.out.println("对象被释放-->"+this);

}

}

public class SystemDemo02

{

public static void main(String[] args)

{

Person pr = new Person("张三",20);

pr=null;

System.gc();

}

}

对象被释放-->姓名：张三年龄：20

## 5. 日期操作类

### 5.1 Date类

需要导入包 import java.util.Date;

Date date = new Date();

System.out.println(date);

输出：Mon Jul 18 17:21:10 CST 2016

### 5.2 Calendar类

使用此类可以直接将日期精确到毫秒

Calendar是一个抽象类

Calendar calendar = new GregorianCalendar(); // 实例化Calendar类对象

System.out.println("YEAR: " + calendar.get(Calendar.YEAR));

System.out.println("MONTH: " + (calendar.get(Calendar.MONTH) + 1));

System.out.println("DAY\_OF\_MONTH: " + calendar.get(Calendar.DAY\_OF\_MONTH));

System.out.println("HOUR\_OF\_DAY: " + calendar.get(Calendar.HOUR\_OF\_DAY));

// System.out.println("MINUTE: " + calendar.get(Calendar.MINUTE));

System.out.println("SECOND: " + calendar.get(Calendar.SECOND));

System.out.println("MILLISECOND: " + calendar.get(Calendar.MILLISECOND)); Calendar calendar = new GregorianCanlendar();

注意，取得月份的时候要加1.

## 6. 日期格式化

### 6.1 DateFormat类

此类定义在java.text包中，是一个抽象类。

java.util.Date本身包含完整是日期信息，所以只需将这个日期格式化就可以。

DateFormat df1= null;

DateFormat df2= null;

df1 = DateFormat.getDateInstance();

df2= DateFormat.getDateTimeInstance();

System.out.println("Date "+df1.format(new Date()));

System.out.println("DateTime "+df2.format(new Date()));

输出：

Date 2016-7-18

DateTime 2016-7-18 23:58:33

此时采用的是默认的格式化操作，也可以使用Locale对象指定要显示的区域。

df1 = DateFormat.getDateInstance(DateFormat.YEAR\_FIELD,new Locale("zh","CN"));

df2=DateFormat.getDateTimeInstance(DateFormat.YEAR\_FIELD,DateFormat.ERA\_FIELD,new Locale("zh","CN"));

结果：

Date 2016年7月19日

DateTime 2016年7月19日 上午12时12分37秒 CST

注意导入包 import java.util.Locale;

### 6.2 SimpleDateFormat类

如果要实现转换，需要准备好一个模板，通过此模板进行日期数字的提取工作。

|  |  |  |
| --- | --- | --- |
| No | 标记 | 描述 |
| 1 | y | 表示年，年份是四位数字，所以要使用“yyyy”表示年。 |
| 2 | M | 表示月，月份是两位数字，所以要使用“MM”表示月。 |
| 3 | d | 表示日，日是两位数字，所以要使用“dd“表示日。 |
| 4 | H | 表示时，时是两位数字，所以要使用“HH“表示时。 |
| 5 | m | 表示分，两位数字“mm” |
| 6 | s | 表示秒，两位数字“ss” |
| 7 | S | 表示毫秒，三位数字“SSS” |

在SimpleDateFormat类使用的时候，必须注意的是在构造对象的时候要传入匹配的模板。

构造方法：public SimpleDateFormat（String pattern）

转换：public Date parse（String source）throws ParseException 此时取得的是全部的时间数

格式化；public final String format（Date date） 将时间格式化成字符串

String strDate= "2016-07-19 00:53:30.345";

String pat1 = "yyyy-MM-dd HH:mm:ss.SSS";

String pat2 = "yyyy年MM月dd日 HH时mm分ss秒SSS毫秒";

SimpleDateFormat sdf1 = new SimpleDateFormat(pat1);

SimpleDateFormat sdf2 = new SimpleDateFormat(pat2);

Date d = null;

try{

d = sdf1.parse(strDate);

}

catch(Exception e)

{

e.printStackTrace();

}

System.out.println(sdf2.format(d));

输出：

2016年07月19日 00时53分30秒345毫秒

其他：String timeStamp = **new** SimpleDateFormat(**"yyyyMMdd\_HHmmss"**).format(**new** Date());

## 7. Math类

静态类，调用静态方法

sqrt:平方根

max：求两数的最大值

min：求两数的最小值

pow：求幂

round：求四舍五入

Math.round(33.6) 输出34

## 8. Random类

导入包import java.util.Random;

先定义一Random对象

Random r = new Random();

public int nextInt(int n)

该方法的作用是生成一个随机的int值，该值介于[0,n)的区间，也就是0到n之间的随机int值，包含0而不包含n。

public double nextDouble()

该方法的作用是生成一个随机的double值，数值介于[0,1.0)之间

## 9. NumberFormat类

完成数字的格式化操作

NumberFormat nf = null;

nf = NumberFormat.getInstance();

System.out.println(nf.format(1234567));

输出：1,234,567

NumberFormat.getInstance();参数可以指定Locale区域，没有参数这调用默认本地locale

## 10 DecimalFormat

与simpleDateFormat类似，要指定模板

class FormatDemo

{

public void format1(String pattern,double value)

{

DecimalFormat df = null;

df = new DecimalFormat(pattern) ;

String str = df.format(value);

System.out.println(str);

}

}

FormatDemo demo = new FormatDemo();

demo.format1("###,###.###",111222.34567);

demo.format1("000,000.000",11222.34567);

demo.format1("000,000.000☆",11222.34567);

demo.format1("##.###%",0.34567);

demo.format1("00.###%",0.034567);

demo.format1("##.###\u2030",0.34567);

输出：

111,222.346

011,222.346

011,222.346☆

34.567%

03.457%

345.67‰

## 11. 大数字

正常情况下，一个数字最多只能放到long中。

操作整数：BigInteger

操作小数：BigDecimal

### 11.1 BigInteger

导入包：import java.math.BigInteger;

BigInteger bi1 = new BigInteger("123456789");

BigInteger bi2 = new BigInteger("987654321");

System.out.println("加法："+bi1.add(bi2));

System.out.println("减法："+bi1.subtract(bi2));

System.out.println("乘法："+bi1.multiply(bi2));

System.out.println("除法："+bi1.divide(bi2));

System.out.println("最大数："+bi1.max(bi2));

System.out.println("最小数："+bi1.min(bi2));

输出：

加法：1111111110

减法：-864197532

乘法：121932631112635269

除法：0

最大数：987654321

最小数：123456789

divide触发默认没有余数，如果要余数，要用divideAndRemainder方法

BigInteger result[]= bi2.divideAndRemainder(bi1); //求出余数

System.out.println(result[0]+" "+result[1]);

输出： 8 9

### 11.2 BigDecimal

## 12 对象克隆

在object类中存在clone方法。

如果一个对象能克隆，必须要继承自Cloneable接口。

为什么需要克隆：

在实际编程过程中，我们常常要遇到这种情况：有一个对象A，在某一时刻A中已经包含了一些有效值，此时可能会需要一个和A完全相同新对象B，并且此后对B任何改动都不会影响到A中的值，也就是说，A与B是两个独立的对象，但B的初始值是由A对象确定的。在Java语言中，用简单的赋值语句是不能满足这种需求的，要满足这种需求有很多途径。

观察一下Object类的clone()方法是一个native方法，native方法的效率一般来说都是远高于java中的非native方法。这也解释了为什么要用Object中clone()方法而不是先new一个类，然后把原始对象中的信息赋到新对象中，虽然这也实现了clone功能。Object类中的clone()还是一个protected属性的方法，重载之后要把clone()方法的属性设置为public。

Object类中clone()方法产生的效果是：先在内存中开辟一块和原始对象一样的空间，然后原样拷贝原始对象中的内容。对基本数据类型，这样的操作是没有问题的，但对非基本类型变量，我们知道它们保存的仅仅是对象的引用，这也导致clone后的非基本类型变量和原始对象中相应的变量指向的是同一个对象。

对象的序列化还有另一个容易被大家忽略的功能就是对象复制（Clone），Java中通过Clone机制可以复制大部分的对象，但是众所周知，Clone有深度Clone和浅度Clone，如果你的对象非常非常复杂，并且想实现深层 Clone，如果使用序列化，不会超过10行代码就可以解决。

class Person implements Cloneable{ // 实现Cloneable接口表示可以被克隆

private String name ;

public Person(String name){

this.name = name ;

}

public void setName(String name){

this.name = name ;

}

public String getName(){

return this.name ;

}

public String toString(){

return "姓名：" + this.name ;

}

public Object clone()

throws CloneNotSupportedException

{

return super.clone() ; // 具体的克隆操作由父类完成

}

};

public class CloneDemo01{

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

Person p1 = new Person("张三") ;

Person p2 = (Person)p1.clone() ;

p2.setName("李四") ;

System.out.println("原始对象：" + p1) ;

System.out.println("克隆之后的对象：" + p2) ;

}

};

## 13 数组Arrays类

### 13.1 sort排序方法

int temp[] ={4,6,2,6,8,22,53,24};

Arrays.sort(temp);

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

{

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

}

sort有很多重载的方法，sort(Ojbect[] temp),能排序对象。

### 13.2 toString转换为字符串

System.out.println(Arrays.toString(temp));

输出: [2, 4, 6, 6, 8, 22, 24, 53]

### 13.3 二分查询binarySearch

int point = Arrays.binarySearch(temp,22); //查找22元素的位置

### 13.4 数组填充 fill

Arrays.fill(temp,9);

System.out.println(Arrays.toString(temp));

输出 [9, 9, 9, 9, 9, 9, 9, 9]

注意数组是引用传递

## 14 Comparable接口

public interface Comparable<T>{

public int compareTo(T o);

}

class Student implements Comparable<Student> { // 指定类型为Student

private String name ;

private int age ;

private float score ;

public Student(String name,int age,float score){

this.name = name ;

this.age = age ;

this.score = score ;

}

public String toString(){

return name + "\t\t" + this.age + "\t\t" + this.score ;

}

public int compareTo(Student stu){ // 覆写compareTo()方法，实现排序规则的应用

if(this.score>stu.score){

return -1 ;

}else if(this.score<stu.score){

return 1 ;

}else{

if(this.age>stu.age){

return 1 ;

}else if(this.age<stu.age){

return -1 ;

}else{

return 0 ;

}

}

}

};

public class ComparableDemo{

public static void main(String args[]){

Student stu[] = {new Student("张三",20,90.0f),

new Student("李四",22,90.0f),new Student("王五",20,99.0f),

new Student("赵六",20,70.0f),new Student("孙七",22,100.0f)} ;

java.util.Arrays.sort(stu) ; // 进行排序操作

for(int i=0;i<stu.length;i++){ // 循环输出数组中的内容

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

}

}

};

二叉树排序原理：

被排序的对象要实现Comparable接口，下面的例子Integer就实现了Comprable接口

Array.Sort就是使用这种排序方法

class BinaryTree

{

class Node{

private Comparable data;

private Node left;

private Node right;

public Node(Comparable data)

{

this.data=data;

}

public void add(Node newNode)

{

if(newNode.data.compareTo(this.data)<0)

{

if(this.left==null)

{

this.left= newNode;

}

else

{

this.left.add(newNode) ;

}

}

else

{

if(this.right==null)

{

this.right=newNode;

}

else

{

this.right.add(newNode);

}

}

}

public void printNode()

{

if(this.left!=null)

{

this.left.printNode();

}

System.out.print(this.data+"\t");

if(this.right!=null)

{

this.right.printNode();

}

}

} ;

private Node root;

public void add(Comparable data)

{

Node newNode = new Node(data);

if(root ==null)

{

root = newNode;

}

else

{

root.add(newNode);

}

}

public void print()

{

this.root.printNode();

}

};

public class ComparableDemo2{

public static void main(String[] args)

{

BinaryTree bt = new BinaryTree();

bt.add(8);

bt.add(3);

bt.add(8);

bt.add(3);

bt.add(83);

bt.add(32);

bt.add(48);

bt.add(13);

bt.add(58);

bt.add(39);

bt.print();

}

}

## 15 Comparator接口

**如果一个已经定义好的类没有实现Comparable接口，可以定义一个Comparator类来排序**

import java.util.\* ;

class Student{ // 指定类型为Student

private String name ;

private int age ;

public Student(String name,int age){

this.name = name ;

this.age = age ;

}

public boolean equals(Object obj){ // 覆写equals方法

if(this==obj){

return true ;

}

if(!(obj instanceof Student)){

return false ;

}

Student stu = (Student) obj ;

if(stu.name.equals(this.name)&&stu.age==this.age){

return true ;

}else{

return false ;

}

}

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 ;

}

public String toString(){

return name + "\t\t" + this.age ;

}

};

class StudentComparator implements Comparator<Student>{ // 实现比较器

// 因为Object类中本身已经有了equals()方法

public int compare(Student s1,Student s2){

if(s1.equals(s2)){

return 0 ;

}else if(s1.getAge()<s2.getAge()){ // 按年龄比较

return 1 ;

}else{

return -1 ;

}

}

};

public class ComparatorDemo{

public static void main(String args[]){

Student stu[] = {new Student("张三",20),

new Student("李四",22),new Student("王五",20),

new Student("赵六",20),new Student("孙七",22)} ;

java.util.Arrays.sort(stu,new StudentComparator()) ; // 进行排序操作

for(int i=0;i<stu.length;i++){ // 循环输出数组中的内容

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

}

}

总结：在使用中尽可能使用Comparable在需要排序的类上实现此接口，而Comparator需要单独建立一个排序的类，这样如果有很多的话，则排序规则类就会非常的多，操作起来非常麻烦。

## 16 Observer观察者设计模式

import java.util.\*;

class House extends Observable{

private float price;

public House(float price)

{

this.price=price;

}

public float getPrice()

{

return this.price;

}

public void setPrice(float price)

{

super.setChanged();

super.notifyObservers(price) ;

this.price=price;

}

public String toString()

{

return "房子价格为："+this.price;

}

}

class HousePriceObserver implements Observer

{

private String name;

public HousePriceObserver(String name)

{

this.name=name;

}

public void update(Observable o,Object arg){

if(arg instanceof Float)

{

System.out.print(this.name+"观察到价格更改为：");

System.out.println(((Float)arg).floatValue());

}

}

}

public class ObserDemo{

public static void main(String[] args)

{

House h= new House(10000);

HousePriceObserver hop1= new HousePriceObserver("购房者1");

HousePriceObserver hop2= new HousePriceObserver("购房者2");

HousePriceObserver hop3= new HousePriceObserver("购房者3");

h.addObserver(hop1);

h.addObserver(hop2);

h.addObserver(hop3);

System.out.println(h);

h.setPrice(6666);

System.out.println(h);

}

}

首先要定义观察的对象类，继承自Observerable类，然后定义观察者类，继承自Observer接口。

在Observer接口中的update方法里面的两个参数：

o：代表Observerable类的对象

arg：需要被观察的内容

**此模式是一种哦本身的机制，不使用Obser和Obserable也是可以实现的，只是比较复杂。**

# 2016年8月4日15:07:34

## 1.正则表达式

判断是否由数字组成：

import java.util.regex.Pattern;

public class RegexDemo

{

public static void main(String[] args)

{

String str="1234567890";

if(Pattern.compile("[0-9]+").matcher(str).matches())

{

System.out.println("数字组成");

}

}

}

1.1 Pattern和Matcher类

这两个类都定义在java.util.regex包中

### 1.1.1 判断日期格式

String str="1990-09-19";

String pat ="\\d{4}-\\d{2}-\\d{2}";

Pattern p = Pattern.compile(pat); //实例化Pattern类

Matcher m = p.matcher(str); //实例化Matcher类

if(m.matches())

{

System.out.println("日期格式合法");

}

else

{

System.out.println("日期格式不合法");

}

1.1.2 使用split拆分

String str= "A12B33Cfd3234E43R";

String pat="\\d+"; //按照数字拆分

Pattern p= Pattern.compile(pat);

String s[] = p.split(str);

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

{

System.out.print(s[i]+"\t") ;

}

输出：A B Cfd E R

### 1.1.2 替换

String str= "A12B33Cfd3234E43R";

String pat="\\d+";

Pattern p= Pattern.compile(pat);

Matcher m = p.matcher(str);

String newString=m.replaceAll("--");

System.out.println(newString);

输出：A--B--Cfd--E—R

1.2 在JDK 1.4之后，java对正则进行了一些扩充，在String中开始直接支持正则操作。

替换

String str= "A12B33Cfd3234E43R".replaceAll("\\d+","--");

System.out.println(str)；

输出：A--B--Cfd--E—R

匹配：

boolean temp ="1991-09-19".matches("\\d{4}-\\d{2}-\\d{2}");

System.out.println(temp);

输出：true

拆分：

String s[] ="A12B33Cfd3234E43R".split("\\d+");

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

{

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

}

输出：A B Cfd E R

2. 定时调度Timer和TimerTask

Timer是一种线程设施，可以用来实现某一个时间或某一段时间后，安排某一个任务执行一次，或定期重复执行。

每一个Timer对象对应的上一个线程，因此计时器所执行任务应该迅速完成，否则可能会延迟后续任务的执行，而这些后续任务就有可能堆在一起，等到该任务完成后才能快速连续执行。

// 完成具体的任务操作

import java.util.TimerTask ;

import java.util.Date ;

import java.text.SimpleDateFormat ;

class MyTask extends TimerTask{ // 任务调度类都要继承TimerTask

public void run(){

SimpleDateFormat sdf = null ;

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

System.out.println("当前系统时间为：" + sdf.format(new Date())) ;

}

};

import java.util.Timer ;

public class TestTask{

public static void main(String args[]){

Timer t = new Timer() ; // 建立Timer类对象

MyTask mytask = new MyTask() ; // 定义任务

t.schedule(mytask,1000,2000) ; // 设置任务的执行，1秒后开始，每2秒重复

}

};

# 2016年8月4日23:02:09

## 1. JAVA IO操作

### 1.1.创建一个新文件

创建文件：createNewFile

File f = new File("d:\\test.txt")；

try{

f.createNewFile();

}

catch(IOException e)

{

e.printStackTrace();

}

在各个操作系统中，实际上路径的分隔符是不一样的，在windows中使用反斜杠“\”,在linux中使用正斜杠“/”

如果想让java程序可移植性继续保持，则最好根据所在的操作系统自动使用分隔符。

pathSeparator和separator

System.out.println("pathSeparator:"+File.pathSeparator);

System.out.println("separator:"+File.separator);

输出:

pathSeparator:;

separator:\

File f = new File("d:"+File.separator+"test.txt");

try{

f.createNewFile();

}

catch(IOException e)

{

e.printStackTrace();

}

### 1.2 删除一个指定的文件

delete方法

File f = new File("d:"+File.separator+"test.txt");

f.delete();

1.3 判断一个文件是否存在

exists方法

if(f.exists())

{

System.out.println("存在");

}

### 1.3创建文件夹

makedir方法

File f = new File("d:"+File.separator+"hello");

f.mkdir();

**1.4 列出指定目录的全部文件**

以字符串数组的方式返回 public String[] list()

以File数组的形式返回 public File[] listFiles()

File f = new File("d:"+File.separator);

String str[] = f.list();

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

{

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

}

列出文件名

File f = new File("d:"+File.separator);

File files[] = f.listFiles();

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

{

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

}

列出完整路径

### 1.4 判断一个给定的路径是否是路径

public boolean isDirectory()

File f = new File("d:");

boolean temp =f.isDirectory();

System.out.println(f.getPath()+" "+temp);

输出：d: true

其中 f.getPath是获取路径字符串

# 2016年8月5日17:08:03

## 1、RandomAccessFile类

完成随机读取功能，可以读取指定位置的内容。

import java.io.File ;

import java.io.RandomAccessFile ;

public class RandomAccessFileDemo01{

// 所有的异常直接抛出，程序中不再进行处理

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

File f = new File("d:" + File.separator + "test.txt") ; // 指定要操作的文件

RandomAccessFile rdf = null ; // 声明RandomAccessFile类的对象

rdf = new RandomAccessFile(f,"rw") ;// 读写模式，如果文件不存在，会自动创建

String name = null ;

int age = 0 ;

name = "zhangsan" ; // 字符串长度为8

age = 30 ; // 数字的长度为4

rdf.writeBytes(name) ; // 将姓名写入文件之中

rdf.writeInt(age) ; // 将年龄写入文件之中

name = "lisi " ; // 字符串长度为8

age = 31 ; // 数字的长度为4

rdf.writeBytes(name) ; // 将姓名写入文件之中

rdf.writeInt(age) ; // 将年龄写入文件之中

name = "wangwu " ; // 字符串长度为8

age = 32 ; // 数字的长度为4

rdf.writeBytes(name) ; // 将姓名写入文件之中

rdf.writeInt(age) ; // 将年龄写入文件之中

rdf.close() ; // 关闭

}

};

以上代码完成了写的操作

import java.io.File ;

import java.io.RandomAccessFile ;

public class RandomAccessFileDemo02{

// 所有的异常直接抛出，程序中不再进行处理

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

File f = new File("d:" + File.separator + "test.txt") ; // 指定要操作的文件

RandomAccessFile rdf = null ; // 声明RandomAccessFile类的对象

rdf = new RandomAccessFile(f,"r") ;// 以只读的方式打开文件

String name = null ;

int age = 0 ;

byte b[] = new byte[8] ; // 开辟byte数组

// 读取第二个人的信息，意味着要空出第一个人的信息

rdf.skipBytes(12) ; // 跳过第一个人的信息

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

b[i] = rdf.readByte() ; // 读取一个字节

}

name = new String(b) ; // 将读取出来的byte数组变为字符串

age = rdf.readInt() ; // 读取数字

System.out.println("第二个人的信息 --> 姓名：" + name + "；年龄：" + age) ;

// 读取第一个人的信息

rdf.seek(0) ; // 指针回到文件的开头

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

b[i] = rdf.readByte() ; // 读取一个字节

}

name = new String(b) ; // 将读取出来的byte数组变为字符串

age = rdf.readInt() ; // 读取数字

System.out.println("第一个人的信息 --> 姓名：" + name + "；年龄：" + age) ;

rdf.skipBytes(12) ; // 空出第二个人的信息

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

b[i] = rdf.readByte() ; // 读取一个字节

}

name = new String(b) ; // 将读取出来的byte数组变为字符串

age = rdf.readInt() ; // 读取数字

System.out.println("第三个人的信息 --> 姓名：" + name + "；年龄：" + age) ;

rdf.close() ; // 关闭

}

};

以上代码完成了读的操作

## 2. 字节流与字符流

在java.io包中操作文件内容的主要有两大类：字节流、字符流，两类都分为输入和输出操作。在字节流中输出数据主要使用OutputStream完成，输入使用InputStream，在字符流中输出主要使用Writer类完成，输入主要使用Reader类完成。

操作流程：

使用File类打开一个文件

通过字节流或字符流的子类，指定输出位置

进行读写操作

关闭输入输出

### 2.1字节流

**写文件**

File f = new File("d:"+File.separator+"test.txt");

OutputStream out = new FileOutputStream(f);

String str ="Hello World";

byte b[] = str.getBytes();

out.write(b);

out.close();

在操作的时候，如果文件不存在，系统会自动创建文件。

以上方法，如果重新执行程序，则肯定会覆盖文件中的内容，那么此时可以通过FileOutputStream向文件中追加内容。

public FileOutputStream(File file,boolean append)throws FileNotFoundException

如果将append设置为true，则表示在文件末尾追加内容。

也可以按照一个一个字节来写

File f= new File("d:" + File.separator + "test.txt") ; // 声明File对象

// 第2步、通过子类实例化父类对象

OutputStream out = null ; // 准备好一个输出的对象

out = new FileOutputStream(f) ; // 通过对象多态性，进行实例化，第二个参数为true，表示追加

// 第3步、进行写操作

String str = "Hello World!!!" ; // 准备一个字符串

byte b[] = str.getBytes() ; // 只能输出byte数组，所以将字符串变为byte数组

for(int i=0;i<b.length;i++){ // 采用循环方式写入

out.write(b[i]) ; // 每次只写入一个内容

}

// 第4步、关闭输出流

out.close() ;

**读文件**

import java.io.File;

import java.io.InputStream;

import java.io.FileInputStream;

public class InputStreamDemo

{

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

{

File f = new File("d:"+File.separator+"test.txt");

InputStream input = new FileInputStream(f);

byte[] b= new byte[1024];

input.read(b); //返回值是实际读取到的长度

input.close();

System.out.println(new String(b));

}

}

上面会输出很多的空格

修改为：

File f = new File("d:"+File.separator+"test.txt");

InputStream input = new FileInputStream(f);

byte[] b= new byte[1024];

int len = input.read(b);

input.close();

System.out.println(len);

System.out.println(new String(b,0,len));

上面的代码会开辟多余的数组空间，应该根据文件大小来设置数组长度

byte[] b= new byte[(int)f.length()];

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

b[i] = (byte)input.read() ; // 读取内容

}

// 第4步、关闭输出流

input.close() ;

如果事先不知道文件大小，可以使用读字节的方式，上面的read是读到一个byte数组里面，返回长度，如果read没有参数，则是读取一个字节，返回该字节内容，如果返回值为-1，则表示读到文件末尾了。

File f = new File("d:"+File.separator+"test.txt") ;

InputStream input = new FileInputStream(f);

int len=0;

int temp=0;

char c[]= new char[1024];

while((temp=input.read())!=-1)

{

c[len]=(char)temp;

len++;

}

System.out.println(new String(c,0,len));

### 2.2字符流

字符流读取

File f= new File("d:"+File.separator+"test.txt");

Reader rd = new FileReader(f);

char c[] = new char[1024];

int len = rd.read(c);

rd.close();

System.out.println(new String(c,0,len));

File f= new File("d:" + File.separator + "test.txt") ;

Reader input = null ;

input = new FileReader(f) ; // 通过对象多态性，进行实例化

char c[] = new char[1024] ; // 所有的内容都读到此数组之中

int temp = 0 ; // 接收每一个内容

int len = 0 ; // 读取内容

while((temp=input.read())!=-1){

// 如果不是-1就表示还有内容，可以继续读取

c[len] = (char)temp ;

len++ ;

}

// 第4步、关闭输出流

input.close() ; // 关闭输出流

System.out.println("内容为：" + new String(c,0,len)) ; // 把字符数组变为字符串输出

**字符流写入**

File f = new File("d:"+File.separator+"test2.txt");

Writer wr = new FileWriter(f);

String str="Hello world";

wr.write(str);

wr.close();

注意一定要调用close方法，否则不会讲内容写入文件，如果是用outputstream方法，则即使不close也能写入文件。

# 2016年8月7日16:25:38

## 1.字节字符转换流

读取：

File f = new File("d:"+File.separator+"test.txt");

Reader reader = new InputStreamReader(new FileInputStream(f)) ;

char c[] = new char[1024];

int len= reader.read(c);

reader.close();

System.out.println(new String(c,0,len));

写入：

File f = new File("d:" + File.separator + "test.txt") ;

Writer out = null ; // 字符输出流

out = new OutputStreamWriter(new FileOutputStream(f)) ; // 字节流变为字符流

out.write("hello world!!") ; // 使用字符流输出

out.close() ;

## 2. 内存操作流

ByteArrayInputStream和ByteArrayOutputStream

String str = "HELLOWORLD" ; // 定义一个字符串，全部由大写字母组成

ByteArrayInputStream bis = null ; // 内存输入流

ByteArrayOutputStream bos = null ; // 内存输出流

bis = new ByteArrayInputStream(str.getBytes()) ; // 向内存中输出内容

bos = new ByteArrayOutputStream() ; // 准备从内存ByteArrayInputStream中读取内容

int temp = 0 ;

while((temp=bis.read())!=-1){

char c = (char) temp ; // 读取的数字变为字符

bos.write(Character.toLowerCase(c)) ; // 将字符变为小写

}

// 所有的数据就全部都在ByteArrayOutputStream中

String newStr = bos.toString() ; // 取出内容

try{

bis.close() ;

bos.close() ;

}catch(IOException e){

e.printStackTrace() ;

}

System.out.println(newStr) ;

## 3. 管道流

管道流主要作用是可以进行两个线程间通讯，分为管道输出流PipedOutputStream,管道输入流PipedInputStream，如果想进行管道输出，则必须把输出流连在输入流之上

import java.io.\* ;

class Send implements Runnable{ // 线程类

private PipedOutputStream pos = null ; // 管道输出流

public Send(){

this.pos = new PipedOutputStream() ; // 实例化输出流

}

public void run(){

String str = "Hello World!!!" ; // 要输出的内容

try{

this.pos.write(str.getBytes()) ;

}catch(IOException e){

e.printStackTrace() ;

}

try{

this.pos.close() ;

}catch(IOException e){

e.printStackTrace() ;

}

}

public PipedOutputStream getPos(){ // 得到此线程的管道输出流

return this.pos ;

}

};

class Receive implements Runnable{

private PipedInputStream pis = null ; // 管道输入流

public Receive(){

this.pis = new PipedInputStream() ; // 实例化输入流

}

public void run(){

byte b[] = new byte[1024] ; // 接收内容

int len = 0 ;

try{

len = this.pis.read(b) ; // 读取内容

}catch(IOException e){

e.printStackTrace() ;

}

try{

this.pis.close() ; // 关闭

}catch(IOException e){

e.printStackTrace() ;

}

System.out.println("接收的内容为：" + new String(b,0,len)) ;

}

public PipedInputStream getPis(){

return this.pis ;

}

};

public class PipedDemo{

public static void main(String args[]){

Send s = new Send() ;

Receive r = new Receive() ;

try{

s.getPos().connect(r.getPis()) ; // 连接管道

}catch(IOException e){

e.printStackTrace() ;

}

new Thread(s).start() ; // 启动线程

new Thread(r).start() ; // 启动线程

}

};

## 4. 打印流

字节打印流PrintStream和字符打印流PrintWriter

可以打印任何的数据类型，例如：小数、整数、字符串等

PrintStream ps = null ; // 声明打印流对象

// 如果现在是使用FileOuputStream实例化，意味着所有的输出是向文件之中

ps = new PrintStream(new FileOutputStream(new File("d:" + File.separator + "test.txt"))) ;

ps.print("hello ") ;

ps.println("world!!!") ;

ps.print("1 + 1 = " + 2) ;

ps.close() ;

构造方法：public PrintStream(OutputStream out) 接收OutputStream的子类

PrintStream可以更加方便的输出数据，这就好比将OutputStream类重新包装了一下，使之输出更加方便。

格式化输出，在JDK1.5后对printstream进行了扩充，增加了格式化输出，直接使用printf()方法可以完成操作，类似c语言的printf。

%s 表示内容为字符串

%d 表示内容为整数

%f 表示内容为小数

%c 表示内容为字符

PrintStream ps = null ; // 声明打印流对象

// 如果现在是使用FileOuputStream实例化，意味着所有的输出是向文件之中

ps = new PrintStream(new FileOutputStream(new File("d:" + File.separator + "test.txt"))) ;

String name = "李兴华" ; // 定义字符串

int age = 30 ; // 定义整数

float score = 990.356f ; // 定义小数

char sex = 'M' ; // 定义字符

ps.printf("姓名：%s；年龄：%d；成绩：%f；性别：%c",name,age,score,sex) ;

ps.close() ;

## 5.System类对IO的支持

System对io的三种支持

System.out.

sytem.err

system.in

System.out输出的时候是将输出的位置定义在了显示器之中

Fileoutputstream是定位在文件里。

import java.io.OutputStream ;

import java.io.IOException ;

public class SystemDemo01{

public static void main(String args[]){

OutputStream out = System.out ; // 此时的输出流是向屏幕上输出

try{

out.write("hello world!!!".getBytes()) ; // 向屏幕上输出

}catch(IOException e){

e.printStackTrace() ; // 打印异常

}

try{

out.close() ; // 关闭输出流

}catch(IOException e){

e.printStackTrace() ;

}

}

};

System.err是标准的错误输出

public class SystemDemo02{

public static void main(String args[]){

String str = "hello" ; // 声明一个非数字的字符串

try{

System.out.println(Integer.parseInt(str)) ; // 转型

}catch(Exception e){

System.err.println(e) ;

}

}

};

**System.in**

完成键盘数据读取的功能

InputStream input = System.in;

byte b[] = new byte[1024];

System.out.print("请输入内容：");

int len = input.read(b);

System.out.println("输入内容为："+new String(b,0,len));

input.close();

上面的代码有问题：  
1. 如果字节数组比较小，超出的内容就不会写入

2. 如果输入的内容有中文，数组长度为奇数，则会乱

码，因为汉字两个字节

修改1：

InputStream input = System.in;

StringBuffer buf = new StringBuffer();

int temp=0;

while((temp=input.read()) !=-1)

{

char c = (char) temp;

if(c=='\n') //如果回车，则退出

{

break;

}

buf.append(c);

}

input.close();

System.out.println(buf);

修改后输入英文内容，没有问题，但是输入汉字就全是乱码了

要想实现汉字的读入，只能用**BufferedReader**类来完成

## 6. 输入、输出重定向

**为System.out输出重定向**

使用setOut方法

System.setOut(new PrintStream(new FileOutputStream("d:"+File.separator+"read.txt")));

System.out.println("hello world");

输出到文件了

通过此操作可以完成错误重定向

System.out是希望用户看得到的信息，一旦有错误，最好保存

String str = "hello";

try{

System.out.println(Integer.parseInt(str));

}

catch(Exception e)

{

try{

System.setOut(new PrintStream(new FileOutputStream("d:"+File.separator+"err.txt")));

}

catch(Exception e1)

{

}

System.out.println(e);

}

**为System.err重定向**

1.

System.setErr(new PrintStream(new FileOutputStream("d:"+File.separator+"err.txt")));

System.err.println("hello world");

2.

ByteArrayOutputStream bos = null ; // 声明内存输出流

bos = new ByteArrayOutputStream() ; // 实例化

System.setErr(new PrintStream(bos)) ; // 输出重定向

System.err.print("www.mldnjava.cn") ; // 错误输出，不再向屏幕上输出

System.err.println("李兴华") ; // 向内存中输出

System.out.println(bos) ; // 输出内存中的数据

**为System.in重定向**

默认情况下System.in是指屏幕键盘输入，也可以通过setIn()方法，将其输入流的位置改变。

System.setIn(new FileInputStream("d:"

+ File.separator + "demo.txt")) ; // 设置输入重定向

InputStream input = System.in ; // 从文件中接收数据

byte b[] = new byte[1024] ;// 开辟空间，接收数据

int len = input.read(b) ; //接收

System.out.println("输入的内容为：" + new String(b,0,len)) ;

input.close() ; // 关闭输入流

上面的代码让从文件中接收数据，而不是接收键盘的数据

## 7. BufferedReader类

为了可以从键盘读取任意长度，且可以是中文的字符，要用到这个类，还用用到字符字节转换流InputStreamReader

BufferedReader = null;

buf = new BufferedReader(new InputStreamReader(System.in));

将字节流转换为字符流，然后用readLine方法读数据

BufferedReader buf =null;

buf = new BufferedReader(new InputStreamReader(System.in)) ;

String str =null;

try{

str= buf.readLine();

}

catch(IOException e)

{

e.printStackTrace();

}

System.out.println(str);

}

上面的代码完成了读取任意长度，即使是中文也可以读取的功能。

# 2016年8月9日17:13:15

## 1. Scanner类

Scanner scan= new Scanner(System.in) ;

scan.useDelimiter("\n");

String str= scan.next();

System.out.println(str);

完成键盘输入功能

可以用hasNextInt判断输入的是否是整数，用hasNextFloat判断输入的是否是小数

这个类可以接收各种类型，但是对于日期型的数据却无法接收，

如果要接收Date类型的数据，则只能通过字符串转型，但是在接收的时候依然可以使用Scanner类中提供的方法进行验证，因为可以接收正则表达式

验证：public String hasNext(Pattern pattern)

接收：public String next(String pattern)

scanner还可以从文件中读取数据

File f = new File("d:"+File.separator+"test.txt");

Scanner scan= new Scanner(f) ;

String str=null;

if(scan.hasNext())

{

str= scan.next();

}

System.out.println(str);

以上的代码只能读取第一行且第一个空格前的数据

File f = new File("d:"+File.separator+"test.txt");

Scanner scan= new Scanner(f) ;

StringBuffer str= new StringBuffer();

while(scan.hasNext())

{

str.append(scan.next()).append("\n");

}

System.out.println(str);

# 2016年8月9日22:55:37

## 1. 数据操作流

DataInputStream和DataOutputStream

import java.io.DataInputStream ;

import java.io.File ;

import java.io.FileInputStream ;

public class DataInputStreamDemo{

public static void main(String args[]) throws Exception{ // 所有异常抛出

DataInputStream dis = null ; // 声明数据输入流对象

File f = new File("d:" + File.separator + "order.txt") ; // 文件的保存路径

dis = new DataInputStream(new FileInputStream(f)) ; // 实例化数据输入流对象

String name = null ; // 接收名称

float price = 0.0f ; // 接收价格

int num = 0 ; // 接收数量

char temp[] = null ; // 接收商品名称

int len = 0 ; // 保存读取数据的个数

char c = 0 ; // '\u0000'

try{

while(true){

temp = new char[200] ; // 开辟空间

len = 0 ;

while((c=dis.readChar())!='\t'){ // 接收内容

temp[len] = c ;

len ++ ; // 读取长度加1

}

name = new String(temp,0,len) ; // 将字符数组变为String

price = dis.readFloat() ; // 读取价格

dis.readChar() ; // 读取\t

num = dis.readInt() ; // 读取int

dis.readChar() ; // 读取\n

System.out.printf("名称：%s；价格：%5.2f；数量：%d\n",name,price,num) ;

}

}catch(Exception e){}

dis.close() ;

}

};

## 2. 合并流

SequenceInputStream

将两个文件合并到一个文件

import java.io.File ;

import java.io.SequenceInputStream ;

import java.io.FileInputStream ;

import java.io.InputStream ;

import java.io.FileOutputStream ;

import java.io.OutputStream ;

public class SequenceDemo{

public static void main(String args[]) throws Exception { // 所有异常抛出

InputStream is1 = null ; // 输入流1

InputStream is2 = null ; // 输入流1

OutputStream os = null ; // 输出流

SequenceInputStream sis = null ; // 合并流

is1 = new FileInputStream("d:" + File.separator + "a.txt") ;

is2 = new FileInputStream("d:" + File.separator + "b.txt") ;

os = new FileOutputStream("d:" + File.separator + "ab.txt") ;

sis = new SequenceInputStream(is1,is2) ; // 实例化合并流

int temp = 0 ; // 接收内容

while((temp=sis.read())!=-1){ // 循环输出

os.write(temp) ; // 保存内容

}

sis.close() ; // 关闭合并流

is1.close() ; // 关闭输入流1`

is2.close() ; // 关闭输入流2

os.close() ; // 关闭输出流

}

};

# 2016年8月10日10:56:29

## 1.压缩流

java支持三种压缩格式zip、jar、gzip

zip需要导入java.uitl.zip包，使用ZipFile、ZipOutputStream、ZipInputStream、ZipEntry几个类完成操作。

jar需要导入java.util.jar包，使用JarOutputStream、JarInputStream、JARFile、JAREntry

# 2016年8月10日22:44:36

对象序列化

import java.io.Serializable ;

一个对象要想序列化，必须实现Serializable接口

要想完成序列化，还必须依靠对象输出流ObjectOutputStream和对象输入流ObjectIntputStream

使用对象输出流叫序列化，使用对象输入流叫反序列化

File f = new File("D:" + File.separator + "test.txt") ; // 定义保存路径

ObjectOutputStream oos = null ; // 声明对象输出流

OutputStream out = new FileOutputStream(f) ; // 文件输出流

oos = new ObjectOutputStream(out) ;

oos.writeObject(new Person("张三",30)) ; // 保存对象

oos.close() ; // 关闭

File f = new File("D:" + File.separator + "test.txt") ; // 定义保存路径

ObjectInputStream ois = null ; // 声明对象输入流

InputStream input = new FileInputStream(f) ; // 文件输入流

ois = new ObjectInputStream(input) ; // 实例化对象输入流

Object obj = ois.readObject() ; // 读取对象

ois.close() ; // 关闭

System.out.println(obj) ;

如果某个属性不希望被序列化，可以使用transient关键字

private transient Stirng name;

# 2016年8月11日18:46:41

## 1. List接口

Collection下分为很多子接口，其中一个List借口哦，List接口中可以存放任意数据。而且在List接口中内容是允许重复的。

List接口的功能要比Collection接口强大很多，因为大量扩充了Collection接口的操作。

List接口的方法：

public void add(int index,E element)

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

E get(int index)

public int indexOf(Object o)

public int lastIndexOf(Object o)

public ListIterator<E>listIterator()

public E remove(int index) 按指定位置删除元素

public List<E> subList(int fromIndex,int toIndex) 取出集合中的子集

public E set(int index,E element) 替换指定位置的元素

**List接口的常用子类：ArrayList**

常用操作：

判断集合是否为空： boolean isEmpty()

截取部分集合：List<E> subList(int fromIndex,int toIndex)

查找指定对象是否存在： int indexOf(Object o) 如果找到返回位置，否则返回-1

查找是否存在：boolean contains(Object o)

**如果要对ArrayList进行排序，需要使用Collections类的sort方法**

挽救的子类：Vector

在list接口中还有一个子类：Vectro，Vector属于一个挽救的子类，和之前的操作区别不大。

ArrayList与Vector的区别，ArrayList采用异步处理方式，性能高，Vector采用同步处理方式，性能低。

ArrayList属于非线程安全的操作类，Vector属性线程安全的操作类。

## 2. LinkedList类

LinkedList表示一个链表操作类，继承List接口和Queue接口

**Queue接口定义的方法：**

public E element() 找到链表的表头

public boolean offer(E o) 将指定的元素增加到链表的结尾

public E peek() 找到但并不删除链表的头

public E poll() 找到并删除此链表的头

public E remove() 检索并删除表头

**LinkedList中操作链表的部分方法：**

public void addFirst(E o) 在链表开头增加元素

public void addLast(E o) 在链表结尾增加元素

public boolean offer(E o) 将指定元素增加到链表结尾

public E removeFirst() 删除链表的第一个元素

public E removeLast() 删除链表的最后一个元素

JDK中**LinkedList的方法**



注意：[add](file:///D:\java%20API\api\java\util\LinkedList.html#add(int, E))(int index, [E](file:///D:\java%20API\api\java\util\LinkedList.html) element)   在此列表中指定的位置插入指定的元素

该方法来自于List接口，可以在指定的位置插入元素。ArrayList也有这个方法。因为它们都来自于List接口。

**LinkedList**使用过一次，是在帮李赛写内存管理的例子的时候。用**LinkedList**来保存空间块，就用到在指定位置插入元素。

**LinkedList**不仅能像链表一样操作，还能像队列一样操作，因为它继承于queue。可以在队首或队尾进行操作。

## 2. Set接口

和List接口不同的是Set接口中的元素不能重复，而List接口可以重复

Set接口也是Collection接口的子类

set接口的主要方法与Conllection接口是一致的

Set接口的实例无法向List接口那样进行双向输出

Set接口常用子类：

HashSet ：散列存放

TreeSet ： 有序存放

import java.util.HashSet ;

import java.util.Set ;

public class HashSetDemo01{

public static void main(String args[]){

Set<String> allSet = new HashSet<String>() ;

allSet.add("A") ; // 增加内容

allSet.add("B") ; // 增加内容

allSet.add("C") ; // 增加内容

allSet.add("C") ; // 重复内容

allSet.add("C") ; // 重复内容

allSet.add("D") ; // 增加内容

allSet.add("E") ; // 增加内容

System.out.println(allSet) ;

}

};

输出: [A, B, C, D, E]

import java.util.TreeSet ;

import java.util.Set ;

public class TreeSetDemo01{

public static void main(String args[]){

Set<String> allSet = new TreeSet<String>() ;

allSet.add("C") ; // 增加内容

allSet.add("C") ; // 重复内容

allSet.add("C") ; // 重复内容

allSet.add("D") ; // 增加内容

allSet.add("B") ; // 增加内容

allSet.add("A") ; // 增加内容

allSet.add("E") ; // 增加内容

System.out.println(allSet) ;

}

};

输出：[A, B, C, D, E]

TreeSet是可以排序的

**i**mport java.util.Set ;

import java.util.HashSet ;

class Person{

private String name ;

private int age ;

public Person(String name,int age){

this.name = name ;

this.age = age ;

}

public String toString(){

return "姓名：" + this.name + "；年龄：" + this.age ;

}

};

public class RepeatDemo01{

public static void main(String args[]){

Set<Person> allSet = new HashSet<Person>() ;

allSet.add(new Person("张三",30)) ;

allSet.add(new Person("李四",31)) ;

allSet.add(new Person("王五",32)) ;

allSet.add(new Person("王五",32)) ;

allSet.add(new Person("王五",32)) ;

allSet.add(new Person("赵六",33)) ;

allSet.add(new Person("孙七",33)) ;

System.out.println(allSet) ;

}

};

上面的代码不能排序，因为Person没有实现Comparable接口

import java.util.Set ;

import java.util.TreeSet ;

class Person implements Comparable<Person>{

private String name ;

private int age ;

public Person(String name,int age){

this.name = name ;

this.age = age ;

}

public String toString(){

return "姓名：" + this.name + "；年龄：" + this.age ;

}

public int compareTo(Person per){

if(this.age>per.age){

return 1 ;

}else if(this.age<per.age){

return -1 ;

}else{

return 0 ;

}

}

};

public class TreeSetDemo03{

public static void main(String args[]){

Set<Person> allSet = new TreeSet<Person>() ;

allSet.add(new Person("张三",30)) ;

allSet.add(new Person("李四",31)) ;

allSet.add(new Person("王五",32)) ;

allSet.add(new Person("王五",32)) ;

allSet.add(new Person("王五",32)) ;

allSet.add(new Person("赵六",33)) ;

allSet.add(new Person("孙七",33)) ;

System.out.println(allSet) ;

}

};

结果：[姓名：张三；年龄：30, 姓名：李四；年龄：31, 姓名：王五；年龄：32, 姓名：赵六；年龄：33]

上面的代码 孙七没有加进来

import java.util.Set ;

import java.util.TreeSet ;

class Person implements Comparable<Person>{

private String name ;

private int age ;

public Person(String name,int age){

this.name = name ;

this.age = age ;

}

public String toString(){

return "姓名：" + this.name + "；年龄：" + this.age ;

}

public int compareTo(Person per){

if(this.age>per.age){

return 1 ;

}else if(this.age<per.age){

return -1 ;

}else{

return this.name.compareTo(per.name) ; // 调用String中的compareTo()方法

}

}

};

public class TreeSetDemo04{

public static void main(String args[]){

Set<Person> allSet = new TreeSet<Person>() ;

allSet.add(new Person("张三",30)) ;

allSet.add(new Person("李四",31)) ;

allSet.add(new Person("王五",32)) ;

allSet.add(new Person("王五",32)) ;

allSet.add(new Person("王五",32)) ;

allSet.add(new Person("赵六",33)) ;

allSet.add(new Person("孙七",33)) ;

System.out.println(allSet) ;

}

};

import java.util.Set ;

import java.util.HashSet ;

class Person{

private String name ;

private int age ;

public Person(String name,int age){

this.name = name ;

this.age = age ;

}

public String toString(){

return "姓名：" + this.name + "；年龄：" + this.age ;

}

};

public class RepeatDemo01{

public static void main(String args[]){

Set<Person> allSet = new HashSet<Person>() ;

allSet.add(new Person("张三",30)) ;

allSet.add(new Person("李四",31)) ;

allSet.add(new Person("王五",32)) ;

allSet.add(new Person("王五",32)) ;

allSet.add(new Person("王五",32)) ;

allSet.add(new Person("赵六",33)) ;

allSet.add(new Person("孙七",33)) ;

System.out.println(allSet) ;

}

};

输出： [姓名：王五；年龄：32, 姓名：赵六；年龄：33, 姓名：孙七；年龄：33, 姓名：王五；年龄：32, 姓名：张三；年龄：30, 姓名：李 四；年龄：31, 姓名：王五；年龄：32]

上面的代码并没有去掉重复的元素，如果要想去掉重复元素，则需要Object类中的两个方法帮助：

hashCode() 表示一个唯一的编码，一般通过计算表示

equals(): 进行对象的比较操作

import java.util.Set ;

import java.util.HashSet ;

class Person{

private String name ;

private int age ;

public Person(String name,int age){

this.name = name ;

this.age = age ;

}

public boolean equals(Object obj){ // 覆写equals，完成对象比较

if(this==obj){

return true ;

}

if(!(obj instanceof Person)){

return false ;

}

Person p = (Person)obj ; // 向下转型

if(this.name.equals(p.name)&&this.age==p.age){

return true ;

}else{

return false ;

}

}

public int hashCode(){

return this.name.hashCode() \* this.age ; // 定义一个公式

}

public String toString(){

return "姓名：" + this.name + "；年龄：" + this.age ;

}

};

public class RepeatDemo02{

public static void main(String args[]){

Set<Person> allSet = new HashSet<Person>() ;

allSet.add(new Person("张三",30)) ;

allSet.add(new Person("李四",31)) ;

allSet.add(new Person("王五",32)) ;

allSet.add(new Person("王五",32)) ;

allSet.add(new Person("王五",32)) ;

allSet.add(new Person("赵六",33)) ;

allSet.add(new Person("孙七",33)) ;

System.out.println(allSet) ;

}

};

输出：[姓名：赵六；年龄：33, 姓名：王五；年龄：32, 姓名：张三；年龄：30, 姓名：李四；年龄：31, 姓名：孙七；年龄：33]

总结：

1. 一个好的类应该覆写Object类中的equals()、hashCode()、toString()方法，实际上再String中已经全部覆写完成了。

2. Set接口依靠hashCode()和euals()完成重复元素的判断，关于这一点在日后的Map接口中也有体现。

3. TreeSet依靠Comparable接口完成排序操作。

2018年3月5日21:34:25

1. Queue

Queue用于模拟队列这种数据结构，队列是指具有先进先出FIFO的容器。

Kd-tree-master这个kd树的工程里面有用到一个自定义的双向队列Deque。而且还用到了PriorityQueue

PriorityQueue保存队列元素的顺序并不是按加入队列的顺序，而是按队列元素的大小进行重新排序。因此当调用peek()方法或者poll()方法取出队列中的元素的时，并不是取出先进入队列中的元素。但是它并不能很好地按照元素大小进行排序。

PriorityQueue的元素有两种排序方式：

1. 自然排序：采用自然顺序的PriorityQueue集合中的元素必须实现了Comparable接口，而且应该是同一个类的多个实例。
2. 定制排序：创建PriorityQueue队列时传入一个comparator对象，该对象负责对队列中的所有元素进行排序。这样不需要实现comparable接口。

例子：

PriorityQueue<NeighborEntry<T>> m\_Queue;

// constructor

**public** NearestNeighborList(**int** capacity) {

m\_Capacity = capacity;

m\_Queue = **new** PriorityQueue<NeighborEntry<T>>(m\_Capacity);

}

**static** **class** NeighborEntry<T> **implements** Comparable<NeighborEntry<T>> {

**final** T data;

**final** **double** value;

**public** NeighborEntry(**final** T data, **final** **double** value) {

**this**.data = data;

**this**.value = value;

}

**public** **int** compareTo(NeighborEntry<T> t) {

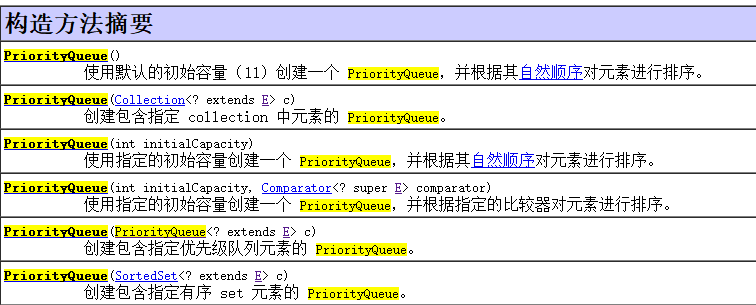
// note that the positions are reversed!

**return** Double.*compare*(t.value, **this**.value);

}

};

上面定义的PriorityQueue的泛型里面的类型是继承于comparable接口的，因此可以实现排序。构造方法的参数是队列的容量。





**package** edu.wlu.cs.levy.cg;

**import** java.util.PriorityQueue;

// Bjoern Heckel's solution to the KD-Tree n-nearest-neighbor problem

**class** NearestNeighborList<T> {

**static** **class** NeighborEntry<T> **implements** Comparable<NeighborEntry<T>> {

**final** T data;

**final** **double** value;

**public** NeighborEntry(**final** T data, **final** **double** value) {

**this**.data = data;

**this**.value = value;

}

**public** **int** compareTo(NeighborEntry<T> t) {

// note that the positions are reversed!

**return** Double.*compare*(t.value, **this**.value);

}

};

PriorityQueue<NeighborEntry<T>> m\_Queue;

**int** m\_Capacity = 0;

// constructor

**public** NearestNeighborList(**int** capacity) {

m\_Capacity = capacity;

m\_Queue = **new** PriorityQueue<NeighborEntry<T>>(m\_Capacity);

}

**public** **double** getMaxPriority() {

NeighborEntry<T> p = m\_Queue.peek();

**return** (p == **null**) ? Double.***POSITIVE\_INFINITY*** : p.value;

}

**public** **boolean** insert(T object, **double** priority) {

**if** (isCapacityReached()) {

**if** (priority > getMaxPriority()) {

// do not insert - all elements in queue have lower priority

**return** **false**;

}

m\_Queue.add(**new** NeighborEntry<T>(object, priority));

// remove object with highest priority

m\_Queue.poll();

} **else** {

m\_Queue.add(**new** NeighborEntry<T>(object, priority));

}

**return** **true**;

}

**public** **boolean** isCapacityReached() {

**return** m\_Queue.size() >= m\_Capacity;

}

**public** T getHighest() {

NeighborEntry<T> p = m\_Queue.peek();

**return** (p == **null**) ? **null** : p.data;

}

**public** **boolean** isEmpty() {

**return** m\_Queue.size() == 0;

}

**public** **int** getSize() {

**return** m\_Queue.size();

}

**public** T removeHighest() {

// remove object with highest priority

NeighborEntry<T> p = m\_Queue.poll();

**return** (p == **null**) ? **null** : p.data;

}

}

# 2016年8月13日18:04:24

## 1. Iterator接口

在集合的操作中支持以下几种方式：

Iterator

ListIterator

foreach输出

Enumeration输出

集合输出的标准操作：

在使用集合输出的时候必须形成以下的一个思路：只要是喷到了集合的输出操作，就一定使用Iterator接口。

原理：Iteartor是专门的迭代输出接口，所谓迭代输出就是将元素一个个进行判断，判断其是否有内容，如果有内容则把内容取出。

**Iterator本身是一个接口，所以要想实例化必须依靠Collection接口完成。**

**Iterator<E> iterator()**

import java.util.List;

import java.util.ArrayList;

import java.util.Iterator;

public class IteratorDemo

{

public static void main(String[] args)

{

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

all.add("a");

all.add("b");

all.add("c");

all.add("d");

all.add("e");

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

while(iter.hasNext()) //判断是否有内容

{

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

}

}

}

Iterator接口提供了remove方法，此方法删除当前的对象。

在实际中，很少调用删除操作，本身的功能就是输出，集合本身就具有删除功能。

import java.util.List ;

import java.util.ArrayList ;

import java.util.Iterator ;

public class IteratorDemo02{

public static void main(String args[]){

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

all.add("hello") ;

all.add("\_") ;

all.add("world") ;

Iterator<String> iter = all.iterator() ; // 为Iterator接口实例化

while(iter.hasNext()){ // 判断是否有内容

String str = iter.next() ;

if("\_".equals(str)){

iter.remove() ; // 删除元素

}else{

System.out.println(str) ; // 输出内容

}

}

System.out.println("删除之后的集合：" + all) ;

}

};

List接口中本身存在删除方法：remove

如果在使用迭代输出的过程中使用了List中的remove()方法执行删除操作，则代码将出现问题。

import java.util.List ;

import java.util.ArrayList ;

import java.util.Iterator ;

public class IteratorDemo03{

public static void main(String args[]){

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

all.add("hello") ;

all.add("\_") ;

all.add("world") ;

Iterator<String> iter = all.iterator() ; // 为Iterator接口实例化

while(iter.hasNext()){ // 判断是否有内容

String str = iter.next() ;

if("\_".equals(str)){

all.remove(str) ; // 删除元素

}else{

System.out.println(str) ; // 输出内容

}

}

System.out.println("删除之后的集合：" + all) ;

}

};

输出：

hello

删除之后的集合：[hello, world]

使用list的删除后循环就不执行了

总结：

1. Iterator接口的功能是从前向后输出，属于单向输出

2.Iteratro的主要功能就是完成迭代输出操作

3.在使用Iterator的时候最好不要删除数据

# 2016年8月13日18:30:47

## 1.ListIterator接口

Iterator接口的主要功能是由前向后输出，而此时如果想要实现由后向前或者由前向后输出，则就必须使用Itertor的子接口ListIterator

虽然此接口可以进行双向输出，但是Collection接口中并没有为此类可以实例化的操作，只有List接口中才存在ListIterator接口的实例化操作。

import java.util.ArrayList ;

import java.util.List ;

import java.util.ListIterator ;

public class ListIteratorDemo01{

public static void main(String argsp[]){

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

all.add("hello") ;

all.add("\_") ;

all.add("world") ;

ListIterator<String> iter = all.listIterator() ;

System.out.print("由前向后输出：") ;

while(iter.hasNext()){

String str = iter.next() ;

System.out.print(str + "、") ;

}

System.out.print("\n由后向前输出：") ;

while(iter.hasPrevious()){

String str = iter.previous() ;

System.out.print(str + "、") ;

}

}

};

必须先由前向后输出，才能由后向前输出，否则不能由后向前输出。

ListIterator还可以进行增加和修改操作add 和set

import java.util.ArrayList ;

import java.util.List ;

import java.util.ListIterator ;

public class ListIteratorDemo02{

public static void main(String argsp[]){

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

all.add("hello") ;

all.add("\_") ;

all.add("world") ;

ListIterator<String> iter = all.listIterator() ;

System.out.print("由前向后输出：") ;

while(iter.hasNext()){

String str = iter.next() ;

System.out.print(str + "、") ;

iter.set("LI - " + str) ; // 修改内容

}

System.out.print("\n由后向前输出：") ;

iter.add("LXH");

while(iter.hasPrevious()){

String str = iter.previous() ;

System.out.print(str + "、") ;

}

}

};

总结：

要想使用ListIterator接口则只能依靠List接口完成

如果要进行由后向前的输出，则只能先进行由前向后输出

对于此接口中的增加及修改操作了解即可

## 2. foreach

foreach可以输出数组，同样也支持集合的输出

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

all.add("hello") ;

all.add("\_") ;

all.add("world") ;

for(String str:all){

System.out.print(str + "、") ;

}

3. Enumeration接口

Iterator接口是一比较新的接口，最早使用Enumeration接口，

但是在java中因为存在发展的历史问题，所以有些地方还会使用到Enumeration输出

二期必须注意的是在使用Enumeration输出的时候一般都是直接操作Vector类完成的

import java.util.Vector;

import java.util.Enumeration;

public class EnumerationDemo01{

public static void main(String args[]){

Vector<String> all = new Vector<String>() ;

all.add("hello") ;

all.add("\_") ;

all.add("world") ;

Enumeration<String> enu = all.elements() ;

while(enu.hasMoreElements()){ //判断是否有内容，hasNext()

System.out.print(enu.nextElement() + "、") ; // 输出元素：next()

}

}

};

# 2016年8月13日22:09:53

## 1. Map接口

Map接口与Collection接口的不同：

Collection接口每次保存的是一个对象，但是在Map接口中保存的是一对对象，以键值对的形式保存。

public interface Map<K,V>

Map接口常用子类：

HashMap: 无序存放的，是新的操作类，key不允许重复

HashTable：无序存放的，是旧的操作类，key不允许重复

TreeMap：可以排序的Map集合，按集合中的key排序，key不允许重复

WeakHashMap： 弱引用的Map集合，当集合中的某些内容不再使用时，可以清除掉无用的数据，可以使用gc进行回收。

IdentityHashMap： key可以重复的Map集合

增加内容用put方法，取出内容用get方法。

import java.util.HashMap ;

import java.util.Map ;

public class HashMapDemo01{

public static void main(String args[]){

Map<String,String> map = null; // 声明Map对象，其中key和value的类型为String

map = new HashMap<String,String>() ;

map.put("mldn","www.mldn.cn") ; // 增加内容

map.put("zhinangtuan","www.zhinangtuan.net.cn") ; // 增加内容

map.put("mldnjava","www.mldnjava.cn") ; // 增加内容

String val = map.get("mldn") ; // 根据key取出值

System.out.println("取出的内容是：" + val) ;

}

};

判断键是否存在用containsKey()方法

判断值是否存在用containsValue方法

import java.util.HashMap ;

import java.util.Map ;

public class HashMapDemo02{

public static void main(String args[]){

Map<String,String> map = null; // 声明Map对象，其中key和value的类型为String

map = new HashMap<String,String>() ;

map.put("mldn","www.mldn.cn") ; // 增加内容

map.put("zhinangtuan","www.zhinangtuan.net.cn") ; // 增加内容

map.put("mldnjava","www.mldnjava.cn") ; // 增加内容

if(map.containsKey("mldn")){ // 判断key是否存在

System.out.println("搜索的key存在！") ;

}else{

System.out.println("搜索的key不存在！") ;

}

if(map.containsValue("www.mldn.cn")){ // 判断value是否存在

System.out.println("搜索的value存在！") ;

}else{

System.out.println("搜索的value不存在！") ;

}

}

};

**如果要输出全部的key，可以使用keySet方法**

Set<K> keySet()

Map<String,String> map = null;

map= new HashMap<String,String>();

map.put("a","1") ;

map.put("b","2");

Set<String> keys = map.keySet();

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

while(iter.hasNext())

{

String str = iter.next();

System.out.println(str) ;

};

如果要输出全部的value，可以用vaules方法

Collection<V> valuse()

Collection<String> values = map.values() ; // 得到全部的value

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

while(iter.hasNext()){

String str = iter.next() ;

System.out.print(str + "、") ;

}

HashTable类，属于旧类

HashTble与HashMap的区别：

(1) 继承不同

public class Hashtable extends Dictionary implements Map  
public class HashMap extends AbstractMap implements Map

(2)

Hashtable 中的方法是同步的，而HashMap中的方法在缺省情况下是非同步的。在多线程并发的环境下，可以直接使用Hashtable，但是要使用HashMap的话就要自己增加同步处理了。

(3)

Hashtable中，key和value都不允许出现null值。

在HashMap中，null可以作为键，这样的键只有一个；可以有一个或多个键所对应的值为null。当get()方法返回null值时，即可以表示 HashMap中没有该键，也可以表示该键所对应的值为null。因此，在HashMap中不能由get()方法来判断HashMap中是否存在某个键， 而应该用containsKey()方法来判断。

(4)

两个遍历方式的内部实现上不同。

Hashtable、HashMap都使用了 Iterator。而由于历史原因，Hashtable还使用了Enumeration的方式

(5)

哈希值的使用不同，HashTable直接使用对象的hashCode。而HashMap重新计算hash值。

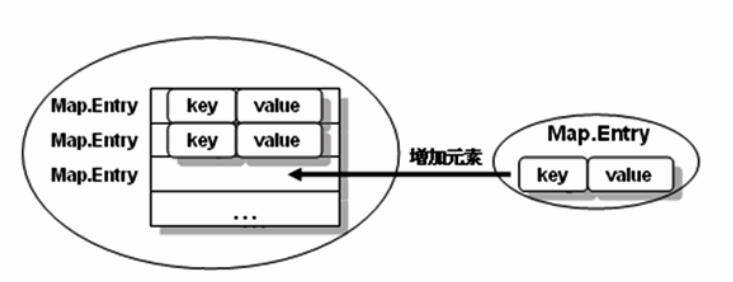
(6)

Hashtable和HashMap它们两个内部实现方式的数组的初始大小和扩容的方式。HashTable中hash数组默认大小是11，增加的方式是 old\*2+1。HashMap中hash数组的默认大小是16，而且一定是2的指数。

## 2. Map.Entry接口

public static interface Map.Entry<K,V>

Map与Map.Entry



一个例子：

Map<Marker, Collectpoint\_Attribute> markerlist = CollectpointActivity.*markerlist*;  
Iterator iter = markerlist.entrySet().iterator();  
**while** (iter.hasNext()) {  
 Map.Entry entry = (Map.Entry) iter.next();  
 Object key = entry.getKey();  
 Object val = entry.getValue();  
  
 temp\_attr = (Collectpoint\_Attribute) val;  
 **if** (temp\_attr.getUuid\_attr().equals(temp\_uuid)) { *//字符串的比较要equals，不能用==* temp\_marker = (Marker) key;  
 **break**;  
 }  
}

## 2. TreeMap子类

此类属于排序类

import java.util.TreeMap ;

import java.util.Map ;

import java.util.Set ;

import java.util.Iterator ;

import java.util.Collection ;

public class TreeMapDemo01{

public static void main(String args[]){

Map<String,String> map = null; // 声明Map对象，其中key和value的类型为String

map = new TreeMap<String,String>() ;

map.put("A、mldn","www.mldn.cn") ; // 增加内容

map.put("C、zhinangtuan","www.zhinangtuan.net.cn") ; // 增加内容

map.put("B、mldnjava","www.mldnjava.cn") ; // 增加内容

Set<String> keys = map.keySet() ; // 得到全部的key

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

while(iter.hasNext()){

String str = iter.next() ;

System.out.println(str + " --> " + map.get(str)) ; // 取出内容

}

}

};

输出：

A、mldn --> www.mldn.cn

B、mldnjava --> www.mldnjava.cn

C、zhinangtuan --> www.zhinangtuan.net.cn

3. 弱引用类WeakHashMap

之前所讲解的Map子类中的数据都是强引用保存的，即里面的内容哦哦不管是否使用都始终在集合中保留，如果希望集合可以自动清理暂时不用的数据就可以使用WeakHashMap类

import java.util.WeakHashMap ;

import java.util.Map ;

import java.util.Set ;

import java.util.Iterator ;

import java.util.Collection ;

public class WeakHashMapDemo01{

public static void main(String args[]){

Map<String,String> map = null; // 声明Map对象，其中key和value的类型为String

map = new WeakHashMap<String,String>() ;

map.put(new String("mldn"),new String("www.mldn.cn")) ;

map.put(new String("zhinangtuan"),new String("www.zhinangtuan.net.cn")) ;

map.put(new String("mldnjava"),new String("www.mldnjava.cn")) ;

System.gc() ; // 强制性进行垃圾的收集操作

map.put(new String("lxh"),new String("lixinghua")) ;

System.out.println(map) ;

}

};

## Map接口输出

1. 对于Map接口来说，其本身是不能直接使用迭代(如：Iterator、foreach)进行输出，因为Map中每个位置存放的是一对值，而Iterator中每次只能找到一个值。所以如果非要使用迭代输出的话，得按照以下方法(iterator为例)

(1) 将Map的实例通过entrySet()方法变为Set接口对象

(2) 通过Set接口实例为Iterator实例化

(3) 通过Iterator迭代输出，每个内容都是Map.Entry的对象

(4) 通过Map.Entry进行key->valuse的分离

Map map = new HashMap();  
map.put("Rajib Sarma","100");  
map.put("Rajib Sarma","200");//The value "100" is replaced by "200".  
map.put("Sazid Ahmed","200");  
  
Iterator iter = map.entrySet().iterator();  
while (iter.hasNext()) {  
 Map.Entry entry = (Map.Entry) iter.next();  
 Object key = entry.getKey();  
 Object val = entry.getValue();  
}

因为Set接口是Collection的子接口，因此可以使用Iterator接口来输出

Map<String,String> map = null; // 声明Map对象，其中key和value的类型为String

map = new HashMap<String,String>() ;

map.put("mldn","www.mldn.cn") ; // 增加内容

map.put("zhinangtuan","www.zhinangtuan.net.cn") ; // 增加内容

map.put("mldnjava","www.mldnjava.cn") ; // 增加内容

Set<Map.Entry<String,String>> allSet = null ;

allSet = map.entrySet() ;

Iterator<Map.Entry<String,String>> iter = null ;

iter = allSet.iterator() ;

while(iter.hasNext()){

Map.Entry<String,String> me = iter.next() ;

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

}

## 2.使用foreach输出

Map<String,String> map = null; // 声明Map对象，其中key和value的类型为String

map = new HashMap<String,String>() ;

map.put("mldn","www.mldn.cn") ; // 增加内容

map.put("zhinangtuan","www.zhinangtuan.net.cn") ; // 增加内容

map.put("mldnjava","www.mldnjava.cn") ; // 增加内容

for(Map.Entry<String,String> me:map.entrySet()){

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

}

如果使用非系统类作为Map的key，则此类必须覆写Object类的hashCode和equals方法

import java.util.Map ;

import java.util.HashMap ;

class Person{

private String name ;

private int age ;

public Person(String name,int age){

this.name = name ;

this.age = age ;

}

public String toString(){

return "姓名：" + this.name + "；年龄：" + this.age ;

}

public boolean equals(Object obj){

if(this==obj){

return true ;

}

if(!(obj instanceof Person)){

return false ;

}

Person p = (Person)obj ;

if(this.name.equals(p.name)&&this.age==p.age){

return true ;

}else{

return false ;

}

}

public int hashCode(){

return this.name.hashCode() \* this.age ;

}

};

public class HashMapDemo08{

public static void main(String args[]){

Map<Person,String> map = null ;

map = new HashMap<Person,String>() ;

map.put(new Person("张三",30),"zhangsan"); // 增加内容

System.out.println(map.get(new Person("张三",30))) ;

}

};

如果没有覆写equals和hashcode方法，直接使用匿名对象，则new出来的对象即使和之前的对象内容完全一样，也不是同一个对象，那么map中get到的只能是null。

hashcoe代表一个编码，如果编码相等并且属性相等，则认为是同一个对象。

作为对象的时候，实际上是依靠hashCode()和equls()来判断两个匿名对象是否相等的，这一点由系统内部自动完成。

复习：

==和equals都表示地址是否相等，但是在String中重写了equals，那么在String中，equals表示内容是否相等，如果两个字符串内容相等，那么equals为true，==为false。

总结：

Map可以使用迭代输出：

map🡪 entrySet🡪Set🡪Iterator🡪Map.Entry🡪key和value

如果使用非系统类作为key，则一定保证覆写equals和hashCode方法，否则无效。

# 2016年8月14日01:30:00

## 1. IdentityHashMap类

在正常的map中，key是不允许重复的。如果想要重复，可以使用该类，这个类在实际中用的比较少。

import java.util.IdentityHashMap ;

import java.util.Set ;

import java.util.Iterator ;

import java.util.Map ;

class Person{

private String name ;

private int age ;

public Person(String name,int age){

this.name = name ;

this.age = age ;

}

public boolean equals(Object obj){

if(this==obj){

return true ;

}

if(!(obj instanceof Person)){

return false ;

}

Person p = (Person)obj ;

if(this.name.equals(p.name)&&this.age==p.age){

return true ;

}else{

return false ;

}

}

public int hashCode(){

return this.name.hashCode() \* this.age ;

}

public String toString(){

return "姓名：" + this.name + "，年龄：" + this.age ;

}

};

public class IdentityHashMapDemo02{

public static void main(String args[]){

Map<Person,String> map = null ; // 声明Map对象

map = new IdentityHashMap<Person,String>() ;

map.put(new Person("张三",30),"zhangsan\_1") ; // 加入内容

map.put(new Person("张三",30),"zhangsan\_2") ; // 加入内容

map.put(new Person("李四",31),"lisi") ; // 加入内容

Set<Map.Entry<Person,String>> allSet = null ; // 准备使用Set接收全部内容

allSet = map.entrySet() ;

Iterator<Map.Entry<Person,String>> iter = null ;

iter = allSet.iterator() ;

while(iter.hasNext()){

Map.Entry<Person,String> me = iter.next() ;

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

}

}

};

2016年8月14日12:35:43

## 1. SortedMap类

回顾：SortedSet是TreeSet的实现接口，那么此类可以进行排序操作

sortedMap也是排序操作，之前学过treemap，那么此类也可以排序。

import java.util.Map ;

import java.util.SortedMap ;

import java.util.TreeMap ;

public class SortedMapDemo{

public static void main(String args[]){

SortedMap<String,String> map = null ;

map = new TreeMap<String,String>() ; // 通过子类实例化接口对象

map.put("D、jiangker","http://www.jiangker.com/") ;

map.put("A、mldn","www.mldn.cn") ;

map.put("C、zhinangtuan","www.zhinangtuan.net.cn") ;

map.put("B、mldnjava","www.mldnjava.cn") ;

System.out.print("第一个元素的内容的key：" + map.firstKey()) ;

System.out.println("：对应的值：" + map.get(map.firstKey())) ;

System.out.print("最后一个元素的内容的key：" + map.lastKey()) ;

System.out.println("：对应的值：" + map.get(map.lastKey())) ;

System.out.println("返回小于指定范围的集合：") ;

for(Map.Entry<String,String> me:map.headMap("B、mldnjava").entrySet()){

System.out.println("\t|- " + me.getKey() + " --> " + me.getValue()) ;

}

System.out.println("返回大于指定范围的集合：") ;

for(Map.Entry<String,String> me:map.tailMap("B、mldnjava").entrySet()){

System.out.println("\t|- " + me.getKey() + " --> " + me.getValue()) ;

}

System.out.println("部分集合：") ;

for(Map.Entry<String,String> me:map.subMap("A、mldn","C、zhinangtuan").entrySet()){

System.out.println("\t|- " + me.getKey() + " --> " + me.getValue()) ;

}

}

};

输出：

第一个元素的内容的key：A、mldn：对应的值：www.mldn.cn

最后一个元素的内容的key：D、jiangker：对应的值：http://www.jiangker.com/

返回小于指定范围的集合：

|- A、mldn --> www.mldn.cn

返回大于指定范围的集合：

|- B、mldnjava --> www.mldnjava.cn

|- C、zhinangtuan --> www.zhinangtuan.net.cn

|- D、jiangker --> http://www.jiangker.com/

部分集合：

|- A、mldn --> www.mldn.cn

|- B、mldnjava --> www.mldnjava.cn

## 2. 集合工具类Collections

Collections与Collection的区别

没有直接关系，但是与集合中的各个接口都有操作的方法支持。

### 空集合操作

import java.util.Collections ;

import java.util.List ;

import java.util.Set ;

public class CollectionsDemo01{

public static void main(String args[]){

List<String> allList = Collections.emptyList() ; // 返回空的 List集合

Set<String> allSet = Collections.emptySet() ; // 返回空的 List集合

allList.add("Hello") ; // 加入数据

}

};

上面的代码不能添加数据

import java.util.Collections ;

import java.util.List ;

import java.util.ArrayList ;

import java.util.Iterator ;

public class CollectionsDemo02{

public static void main(String args[]){

List<String> all = new ArrayList<String>() ; // 返回空的 List集合

Collections.addAll(all,"MLDN","LXH","mldnjava") ;

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

while(iter.hasNext()){

System.out.print(iter.next() + "、") ;

}

}

};

一般直接使用list的添加方法，很少直接使用Collections的addAll方法

### 反转操作

import java.util.Collections ;

import java.util.List ;

import java.util.ArrayList ;

import java.util.Iterator ;

public class CollectionsDemo03{

public static void main(String args[]){

List<String> all = new ArrayList<String>() ; // 返回空的 List集合

Collections.addAll(all,"MLDN","LXH","mldnjava") ;

Collections.reverse(all) ; // 内容反转

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

while(iter.hasNext()){

System.out.print(iter.next() + "、") ;

}

}

};

输出：mldnjava、LXH、MLDN、

### 二分检索方法 binarySearch

import java.util.Collections ;

import java.util.List ;

import java.util.ArrayList ;

import java.util.Iterator ;

public class CollectionsDemo04{

public static void main(String args[]){

List<String> all = new ArrayList<String>() ; // 返回空的 List集合

Collections.addAll(all,"MLDN","LXH","mldnjava") ;

int point = Collections.binarySearch(all,"LXH") ; // 检索数据

System.out.println("检索结果：" + point) ;

}

};

返回位置：1

### 替换指定内容 replaceAll

import java.util.Collections ;

import java.util.List ;

import java.util.ArrayList ;

import java.util.Iterator ;

public class CollectionsDemo05{

public static void main(String args[]){

List<String> all = new ArrayList<String>() ; // 返回空的 List集合

Collections.addAll(all,"MLDN","LXH","mldnjava") ;

if(Collections.replaceAll(all,"LXH","李兴华")){// 替换内容

System.out.println("内容替换成功！") ;

}

System.out.print("替换之后的结果：") ;

System.out.print(all) ;

}

};

输出：

内容替换成功！

替换之后的结果：[MLDN, 李兴华, mldnjava]

### 排序操作:

**对ArrayList进行排序**

import java.util.Collections ;

import java.util.List ;

import java.util.ArrayList ;

import java.util.Iterator ;

public class CollectionsDemo06{

public static void main(String args[]){

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

Collections.addAll(all,"1、MLDN","2、LXH","3、mldnjava") ;

Collections.addAll(all,"B、www.mldn.cn") ;

Collections.addAll(all,"A、www.mldnjava.cn") ;

System.out.println("排序之前的集合：" + all) ;

Collections.sort(all) ;

System.out.println("排序之后的集合：" + all) ;

}

};

输出：

排序之前的集合：[1、MLDN, 2、LXH, 3、mldnjava, B、www.mldn.cn, A、www.mldnjava.cn]

排序之后的集合：[1、MLDN, 2、LXH, 3、mldnjava, A、www.mldnjava.cn, B、www.mldn.cn]

排序操作也需要依靠Comparable接口

**交换指定位置的内容：**

import java.util.Collections ;

import java.util.List ;

import java.util.ArrayList ;

import java.util.Iterator ;

public class CollectionsDemo07{

public static void main(String args[]){

List<String> all = new ArrayList<String>() ; // 返回空的 List集合

Collections.addAll(all,"1、MLDN","2、LXH","3、mldnjava") ;

System.out.println("交换之前的集合：" + all) ;

Collections.swap(all,0,2) ;

System.out.println("交换之后的集合：" + all) ;

}

};

输出：

交换之前的集合：[1、MLDN, 2、LXH, 3、mldnjava]

交换之后的集合：[3、mldnjava, 2、LXH, 1、MLDN]

## 2. Stack类

栈是采用现金后出的操作方式

Stack类是Vector的子类

public class Stack<E> extends Vector<E>

常用方法：

public boolean empty() 测试栈是否为空

public E peek() 查看栈顶，但不删除

public E pop() 出栈，同时删除

public E push(E item) 入栈

public int search(Object o)在栈中查找

import java.util.Stack ;

public class StackDemo{

public static void main(String args[]){

Stack<String> s = new Stack<String>() ;

s.push("A") ; // 入栈

s.push("B") ; // 入栈

s.push("C") ; // 入栈

System.out.print(s.pop() + "、") ;

System.out.print(s.pop() + "、") ;

System.out.println(s.pop() + "、") ;

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

}

};>

输出：

C、B、A、

Exception in thread "main" java.util.EmptyStackException

at java.util.Stack.peek(Unknown Source)

at java.util.Stack.pop(Unknown Source)

at StackDemo.main(StackDemo.java:11)

## 3. Properties类

public class Properties extends HashTable<Object,Object>

Properties 是HashTable的子类，则也是Map的子类，可以使用Map的全部操作，但是一般情况下属性类是单独使用的。

设置属性：

public Object setProperty(String key,String value)

得到属性：

public String getProperty(String key,String defaultValue)

import java.util.Properties;

public class PropertiesDemo01{

public static void main(String args[]){

Properties pro = new Properties() ; // 创建Properties对象

pro.setProperty("BJ","BeiJing") ; // 设置属性

pro.setProperty("TJ","TianJin") ;

pro.setProperty("NJ","NanJing") ;

System.out.println("1、BJ属性存在：" + pro.getProperty("BJ")) ;

System.out.println("2、SC属性不存在：" + pro.getProperty("SC")) ;

System.out.println("3、SC属性不存在，同时设置显示的默认值：" + pro.getProperty("SC","没有发现")) ;

}

};

将属性保存到文件中

import java.util.Properties;

import java.io.File;

import java.io.FileOutputStream;

import java.io.FileNotFoundException;

import java.io.IOException;

public class PropertiesDemo02{

public static void main(String args[]){

Properties pro = new Properties() ; // 创建Properties对象

pro.setProperty("BJ","BeiJing") ; // 设置属性

pro.setProperty("TJ","TianJin") ;

pro.setProperty("NJ","NanJing") ;

File file = new File("D:" + File.separator + "area.properteis") ; // 指定要操作的文件

try{

pro.store(new FileOutputStream(file),"Area Info") ; // 保存属性到普通文件

}catch(FileNotFoundException e){

e.printStackTrace() ;

}catch(IOException e){

e.printStackTrace() ;

}

}

};

从文件中读取属性

import java.util.Properties;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileNotFoundException;

import java.io.IOException;

public class PropertiesDemo03{

public static void main(String args[]){

Properties pro = new Properties() ; // 创建Properties对象

File file = new File("D:" + File.separator + "area.properteis") ; // 指定要操作的文件

try{

pro.load(new FileInputStream(file)) ; // 读取属性文件

}catch(FileNotFoundException e){

e.printStackTrace() ;

}catch(IOException e){

e.printStackTrace() ;

}

System.out.println("1、BJ属性存在：" + pro.getProperty("BJ")) ;

System.out.println("2、SH属性存在：" + pro.getProperty("SH")) ;

}

};

将属性保存到xml中

import java.util.Properties;

import java.io.File;

import java.io.FileOutputStream;

import java.io.FileNotFoundException;

import java.io.IOException;

public class PropertiesDemo04{

public static void main(String args[]){

Properties pro = new Properties() ; // 创建Properties对象

pro.setProperty("BJ","BeiJing") ; // 设置属性

pro.setProperty("TJ","TianJin") ;

pro.setProperty("NJ","NanJing") ;

File file = new File("D:" + File.separator + "area.xml") ; // 指定要操作的文件

try{

pro.storeToXML(new FileOutputStream(file),"Area Info") ; // 保存属性到普通文件

}catch(FileNotFoundException e){

e.printStackTrace() ;

}catch(IOException e){

e.printStackTrace() ;

}

}

};

从xml中读取属性

import java.util.Properties;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileNotFoundException;

import java.io.IOException;

public class PropertiesDemo05{

public static void main(String args[]){

Properties pro = new Properties() ; // 创建Properties对象

File file = new File("D:" + File.separator + "area.xml") ; // 指定要操作的文件

try{

pro.loadFromXML(new FileInputStream(file)) ; // 读取属性文件

}catch(FileNotFoundException e){

e.printStackTrace() ;

}catch(IOException e){

e.printStackTrace() ;

}

System.out.println("1、BJ属性存在：" + pro.getProperty("BJ")) ;

}

};

## 4. 一对多关系

使用类集可以表示出一下的关系：一个学校可以包含多个学生，一个学生属于一学校，那么这就是一个典型的一对多关系，此时就可以通过类集进行关系的表示。

import java.util.List ;

import java.util.ArrayList ;

public class School{

private String name ;

private List<Student> allStudents ;

public School(){

this.allStudents = new ArrayList<Student>() ;

}

public School(String name){

this() ;

this.setName(name) ;

}

public void setName(String name){

this.name = name ;

}

public String getName(){

return this.name;

}

public List<Student> getAllStudents(){

return this.allStudents ;

}

public String toString(){

return "学校名称：" + this.name ;

}

};

public class Student{

private String name ;

private int age ;

private School school; // 一个学生属于一个学校

public Student(String name,int age){

this.setName(name) ;

this.setAge(age) ;

}

public void setSchool(School school){

this.school = school ;

}

public School getSchool(){

return this.school ;

}

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 ;

}

public String toString(){

return "学生姓名：" + this.name + "；年龄：" + this.age ;

}

};

import java.util.Iterator ;

public class TestDemo{

public static void main(String args[]){

School sch = new School("清华大学") ; // 定义学校

Student s1 = new Student("张三",21) ;

Student s2 = new Student("李四",22) ;

Student s3 = new Student("王五",23) ;

sch.getAllStudents().add(s1) ;

sch.getAllStudents().add(s2) ;

sch.getAllStudents().add(s3) ;

s1.setSchool(sch) ;

s2.setSchool(sch) ;

s3.setSchool(sch) ;

System.out.println(sch) ;

Iterator<Student> iter = sch.getAllStudents().iterator() ;

while(iter.hasNext()){

System.out.println("\t|- " + iter.next()) ;

}

}

};

## 5. 多对多关系

import java.util.List ;

import java.util.ArrayList ;

public class Course{

private String name ;

private int credit ;

private List<Student> allStudents ;

public Course(){

this.allStudents = new ArrayList<Student>() ;

}

public Course(String name,int credit){

this() ;

this.name = name ;

this.credit = credit ;

}

public List<Student> getAllStudents(){

return this.allStudents ;

}

public void setName(String name){

this.name = name ;

}

public void setCredit(int credit){

this.credit = credit ;

}

public String getName(){

return this.name ;

}

public int getCredit(){

return this.credit ;

}

public String toString(){

return "课程名称：" + this.name + "；课程学分：" + this.credit ;

}

};

import java.util.List ;

import java.util.ArrayList ;

public class Student{

private String name ;

private int age ;

private List<Course> allCourses ;

public Student(){

this.allCourses = new ArrayList<Course>() ;

}

public Student(String name,int age){

this() ;

this.name = name ;

this.age = age ;

}

public List<Course> getAllCourses(){

return this.allCourses ;

}

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 ;

}

public String toString(){

return "学生姓名：" + this.name + "；年龄：" + this.age ;

}

};

import java.util.Iterator ;

public class TestMore{

public static void main(String args[]){

Course c1 = new Course("英语",3 ) ; // 第一门课程

Course c2 = new Course("计算机",5) ; // 第二门课程

Student s1 = new Student("张三",20) ;

Student s2 = new Student("李四",21) ;

Student s3 = new Student("王五",22) ;

Student s4 = new Student("赵六",23) ;

Student s5 = new Student("孙七",24) ;

Student s6 = new Student("钱八",24) ;

// 第一门课程有三个学生参加

c1.getAllStudents().add(s1) ;

c1.getAllStudents().add(s2) ;

c1.getAllStudents().add(s6) ;

s1.getAllCourses().add(c1) ;

s2.getAllCourses().add(c1) ;

s6.getAllCourses().add(c1) ;

// 第二门课程有六个学生参加

c2.getAllStudents().add(s1) ;

c2.getAllStudents().add(s2) ;

c2.getAllStudents().add(s3) ;

c2.getAllStudents().add(s4) ;

c2.getAllStudents().add(s5) ;

c2.getAllStudents().add(s6) ;

s1.getAllCourses().add(c2) ;

s2.getAllCourses().add(c2) ;

s3.getAllCourses().add(c2) ;

s4.getAllCourses().add(c2) ;

s5.getAllCourses().add(c2) ;

s6.getAllCourses().add(c2) ;

// 输出一门课程的信息，观察一门课程有多少个学生参加\

System.out.println(c1) ;

Iterator<Student> iter1 = c1.getAllStudents().iterator() ;

while(iter1.hasNext()){

Student s = iter1.next() ;

System.out.println("\t|- " + s) ;

}

// 通过学生找到学生参加的课程

System.out.println(s6) ;

Iterator<Course> iter2 = s6.getAllCourses().iterator() ;

while(iter2.hasNext()){

Course c = iter2.next() ;

System.out.println("\t|- " + c) ;

}

}

};

# 2016年8月14日19:10:04

## 1. 枚举 enum

所谓的枚举就是规定好了指定的取值范围，所有的内容只能从指定的范围中取得。

public enum Color{

RED,GREEN,BLUE ; // 定义三个枚举的类型

};

public class GetEnumContent{

public static void main(String args[]){

Color c = Color.BLUE ; // 取出蓝色

System.out.println(c) ;

}

};

public class PrintEnum{

public static void main(String args[]){

for(Color c:Color.values()){ // 输出枚举中的全部内容

System.out.println(c) ;

}

}

};

## 2. enum关键字和Enum类的关系

使用enum关键字可以定义一个枚举，实际上此关键字表示的是java.lang.Enum类型，即：使用enmu声明的枚举类型，就相当于定义一个类，而此类则默认继承java.lang.Enum类。

public abstract class **Enum<E extends Enum<E>>**

extends [Object](http://tool.oschina.net/uploads/apidocs/jdk_7u4/java/lang/Object.html)

implements [Comparable](http://tool.oschina.net/uploads/apidocs/jdk_7u4/java/lang/Comparable.html)<E>, [Serializable](http://tool.oschina.net/uploads/apidocs/jdk_7u4/java/io/Serializable.html)

此类定义的时候使用了泛型机制，而且实现了Comparable接口以及Serialzable接口，证明此种类型是可以比较，可以被序列化的。

构造方法：

protected Enum([String](http://tool.oschina.net/uploads/apidocs/jdk_7u4/java/lang/String.html) name, int ordinal)

不能调用这个构造方法

public class GetEnumInfo{

public static void main(String args[]){

for(Color c:Color.values()){

System.out.println(c.ordinal() + " --> " + c.name()) ;

}

}

};

输出：

0 --> RED

1 --> GREEN

2 --> BLUE

enum Color{

RED("红色"),GREEN("绿色"),BLUE("兰色") ;

private Color(String name){

this.setName(name) ;

}

private String name ; // 定义name属性

public void setName(String name){

this.name = name ;

}

public String getName(){

return this.name ;

}

}

public class ConstructorEnum{

public static void main(String args[]){

for(Color c:Color.values()){

System.out.println(c.ordinal() + " --> " + c.name()

+"(" + c.getName() + ")") ;

}

}

};

输出：

0 --> RED(红色)

1 --> GREEN(绿色)

2 --> BLUE(兰色)

enum Color{

RED,GREEN,BLUE ;

private String name ; // 定义name属性

public void setName(String name){

switch(this){ // 判断操作的是那个枚举对象

case RED:{

if("红色".equals(name)){

this.name = name ; // 允许设置名字

}else{

System.out.println("设置内容错误。") ;

}

break ;

}

case GREEN:{

if("绿色".equals(name)){

this.name = name ; // 允许设置名字

}else{

System.out.println("设置内容错误。") ;

}

break ;

}

case BLUE:{

if("蓝色".equals(name)){

this.name = name ; // 允许设置名字

}else{

System.out.println("设置内容错误。") ;

}

break ;

}

}

this.name = name ;

}

public String getName(){

return this.name ;

}

}

public class SetEnum{

public static void main(String args[]){

Color c = Color.BLUE ; // 得到兰色

c.setName("兰色") ; // 名字错误

c.setName("蓝色") ; // 名字正确

System.out.println(c.getName()) ;

}

};

排序：

import java.util.Iterator;

import java.util.Set ;

import java.util.TreeSet ;

enum Color{

RED,GREEN,BLUE ;

}

public class ComparableEnum{

public static void main(String args[]){

Set<Color> t = new TreeSet<Color>() ; // 设置类型

t.add(Color.GREEN) ; // 加入绿色

t.add(Color.RED) ; // 加入红色

t.add(Color.BLUE) ; // 加入蓝色

Iterator<Color> iter = t.iterator() ;

while(iter.hasNext()){

System.out.print(iter.next() + "、") ;

}

}

};

输出：

RED、GREEN、BLUE、

## 3.类集对Enum的支持

EnumMap和EnumSet

### 3.1 EnumMap

在java.util.EnumMap包中

public class **EnumMap<K extends** [**Enum**](http://docs.oracle.com/javase/7/docs/api/java/lang/Enum.html)**<K>,V>**

extends [AbstractMap](http://docs.oracle.com/javase/7/docs/api/java/util/AbstractMap.html)<K,V>

implements [Serializable](http://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html), [Cloneable](http://docs.oracle.com/javase/7/docs/api/java/lang/Cloneable.html)

键必须是枚举类型

import java.util.EnumMap ;

import java.util.Map ;

enum Color{

RED , GREEN , BLUE ;

}

public class EnumMapDemo{

public static void main(String args[]){

Map<Color,String> desc = null ; // 定义Map对象，同时指定类型

desc = new EnumMap<Color,String>(Color.class) ; // 实例化EnumMap对象

desc.put(Color.RED,"红色") ;

desc.put(Color.GREEN,"绿色") ;

desc.put(Color.BLUE,"蓝色") ;

System.out.println("====== 输出全部的内容 ======") ;

for(Color c:Color.values()){

System.out.println(c.name() + " --> " + desc.get(c)) ;

}

System.out.println("====== 输出全部的键值 ======") ;

for(Color c:desc.keySet()){

System.out.print(c.name() + "、") ;

}

System.out.println() ;

System.out.println("====== 输出全部的内容 ======") ;

for(String s:desc.values()){

System.out.print(s + "、") ;

}

}

};

====== 输出全部的内容 ======

RED --> 红色

GREEN --> 绿色

BLUE --> 蓝色

====== 输出全部的键值 ======

RED、GREEN、BLUE、

====== 输出全部的内容 ======

红色、绿色、蓝色、

### 3.2 EnumSet

* public abstract class EnumSet<E extends [**Enum**](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\Enum.html)<E>>
* extends [AbstractSet](file:///D:\jdk-8u101-docs-all\docs\api\java\util\AbstractSet.html)<E>

implements [Cloneable](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\Cloneable.html), [Serializable](file:///D:\jdk-8u101-docs-all\docs\api\java\io\Serializable.html)

EnumSet是Set的子类，所以里面的内容是无法重复的，在使用EnumSet的时候是不能直接使用关键字new为其进行实例化的，所以在此类中提供了很多的静态方法。

### 将全部的集合设置到EnumSet集合中

import java.util.EnumSet ;

enum Color{

RED , GREEN , BLUE ;

}

public class EnumSetDemo01{

public static void main(String args[]){

EnumSet<Color> es = null ; // 声明一个EnumSet对象

System.out.println("======== EnumSet.allOf(Color.class) =====") ;

es = EnumSet.allOf(Color.class) ; // 将枚举的全部类型设置到EnumSet对象之中

print(es) ;

}

public static void print(EnumSet<Color> temp){ // 专门的输出操作

for(Color c:temp){ // 循环输出EnumSet中的内容

System.out.print(c + "、") ;

}

System.out.println() ;

}

};

### 只设置一个内容到集合

import java.util.EnumSet ;

enum Color{

RED , GREEN , BLUE ;

}

public class EnumSetDemo02{

public static void main(String args[]){

EnumSet<Color> es = null ; // 声明一个EnumSet对象

System.out.println("======== EnumSet.of(Color.BLUE) =====") ;

es = EnumSet.of(Color.BLUE) ; // 将枚举的全部类型设置到EnumSet对象之中

print(es) ;

}

public static void print(EnumSet<Color> temp){ // 专门的输出操作

for(Color c:temp){ // 循环输出EnumSet中的内容

System.out.print(c + "、") ;

}

System.out.println() ;

}

};

### 创建只能放入指定枚举类型的集合

import java.util.EnumSet ;

enum Color{

RED , GREEN , BLUE ;

}

public class EnumSetDemo03{

public static void main(String args[]){

EnumSet<Color> es = null ; // 声明一个EnumSet对象

System.out.println("======== EnumSet.noneOf(Color.class) =====") ;

es = EnumSet.noneOf(Color.class) ; // 将枚举的全部类型设置到EnumSet对象之中

es.add(Color.RED) ; // 增加内容

es.add(Color.GREEN) ; // 增加内容

print(es) ;

}

public static void print(EnumSet<Color> temp){ // 专门的输出操作

for(Color c:temp){ // 循环输出EnumSet中的内容

System.out.print(c + "、") ;

}

System.out.println() ;

}

};

### 创建不包含指定元素的集合

import java.util.EnumSet ;

enum Color{

RED , GREEN , BLUE ;

}

public class EnumSetDemo04{

public static void main(String args[]){

EnumSet<Color> esOld = null ; // 声明一个EnumSet对象

EnumSet<Color> esNew = null ;

System.out.println("======== EnumSet.complementOf(Color.class) =====") ;

esOld = EnumSet.noneOf(Color.class) ; // 将枚举的全部类型设置到EnumSet对象之中

esOld.add(Color.RED) ; // 增加内容

esOld.add(Color.GREEN) ; // 增加内容

esNew = EnumSet.complementOf(esOld) ; // 不包含指定内容

print(esNew) ;

}

public static void print(EnumSet<Color> temp){ // 专门的输出操作

for(Color c:temp){ // 循环输出EnumSet中的内容

System.out.print(c + "、") ;

}

System.out.println() ;

}

};

输出新的集合只包含蓝色，不包含esOld中的红和绿。

### 拷贝一个集合的内容

import java.util.EnumSet ;

enum Color{

RED , GREEN , BLUE ;

}

public class EnumSetDemo05{

public static void main(String args[]){

EnumSet<Color> esOld = null ; // 声明一个EnumSet对象

EnumSet<Color> esNew = null ;

System.out.println("======== EnumSet.copyOf(Color.class) =====") ;

esOld = EnumSet.noneOf(Color.class) ; // 将枚举的全部类型设置到EnumSet对象之中

esOld.add(Color.RED) ; // 增加内容

esOld.add(Color.GREEN) ; // 增加内容

esNew = EnumSet.copyOf(esOld) ; // 从已有的集合拷贝过来

print(esNew) ;

}

public static void print(EnumSet<Color> temp){ // 专门的输出操作

for(Color c:temp){ // 循环输出EnumSet中的内容

System.out.print(c + "、") ;

}

System.out.println() ;

}

};

总结：

EnumMap是符合Map的操作形式的，而EnumSet基本上都是使用静态方法完成的。

在操作中大量使用了枚举.class,这属于java反射机制。

## 4. 枚举的其他应用

可以使用枚举实现一个接口

可以在枚举中定义抽象方法

4.1

枚举类型可以跟普通类型一样实现一个接口，但是实现接口的时候要求枚举中的每个对象都必须单独覆写好接口中的抽象方法。

interface Print{

public String getColor() ;

}

enum Color implements Print{

RED{

public String getColor(){

return "红色" ;

}

},GREEN{

public String getColor(){

return "绿色" ;

}

},BLUE{

public String getColor(){

return "蓝色" ;

}

} ;

}

public class InterfaceEnumDemo{

public static void main(String args[]){

for(Color c:Color.values()){

System.out.print(c.getColor() + "、") ;

}

}

};

4.2

还可以直接在枚举中定义抽象方法，但是要求枚举中的每个对象都分别实现此抽象方法。

enum Color{

RED{

public String getColor(){

return "红色" ;

}

},GREEN{

public String getColor(){

return "绿色" ;

}

},BLUE{

public String getColor(){

return "蓝色" ;

}

} ;

public abstract String getColor() ;

}

public class AbstractMethodEnum{

public static void main(String args[]){

for(Color c:Color.values()){

System.out.print(c.getColor() + "、") ;

}

}

};

# 2016年8月15日12:18:19

## 1. 包及访问控制权限

在java中，可以将一个大型项目中的类分别独立出来，分门别类地存到文件里，再将这些文件一起编译执行，如此的程序代码将更易于维护。

如果有多人共同开发一个项目的时候，则肯定会出现类名称相同的情况，那么这样一来就会比较麻烦。

package是在使用多个类或接口时，为了避免名称重复而采用的一种措施，直接在程序中加入package关键字即可。

定义格式：

package 包名称.子包名称;

package org ; // 定义一个包

class Demo{

public String getInfo(){

return "Hello World!!!" ;

}

};

public class PackageDemo{

public static void main(String args[]){

System.out.println(new Demo().getInfo());

}

};

实际上包名就是文件夹名，因此编译后的.class文件应该放到org文件夹下。如果只用javac编译的话，回放到当前目录下，需要手动已到指定文件夹，这样非常麻烦，可以用javac –d 命令

**javac -d . PackageDemo.java**

注意d后面和.后面都是空格

使用上面的命令会自动生成相应文件夹目录。

运行程序要带上包名java org.PackageDemo

注意只能在当前文件夹下调用java命令执行程序，不能进入org文件夹再用java命令，因为进入，系统再次在org文件夹下找org文件夹，自然找不到。

## 2. 包的导入

当一个包的class文件需要使用另外一个包的class文件的时候就需要使用导入指令。

improt语句

之前的程序，两个类是放到同一个包中的，因此代码与之前没有什么不同，如果几个类放在不同的包中，则在使用类的时候就必须通过imiport语句导入。

import 包名称.子包名称;

**Demo.java**

package guozhang.com;

public class Demo

{

public String getInfo()

{

return "hello world" ;

}

}

**ImportDemo.java**

import guozhang.com.Demo;

public class ImportDemo

{

public static void main(String[] agrs)

{

System.out.println(new Demo().getInfo());

}

}

注意Demo类必须声明为public

如果一个类声明为public class，则文件名必须与类名称一致，而在一个类中只能有一个pbulic class，如果再使用class声明一个类，则文件名可以与类名称不一致，但是执行的时候必须执行生成的class文件名称。

如果一个类需要被外包所访问，则此类一定要定义成public class。

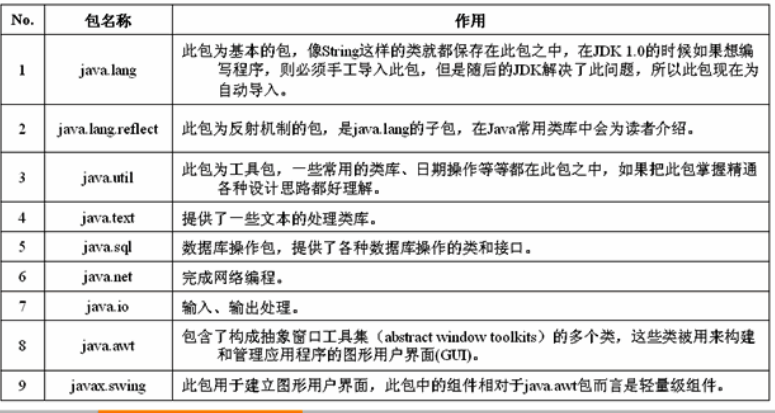
如果有很多类要导入，可以不用每个类名称都写一遍，直接用 \*代替。

import guozhang.com.\*;

在导入不同包的同名类的时候就可能会出现不明确的错误信息。

所以在定义对象的时候，最好写完整的类名称。

常见系统包



3 静态导入

如果一个类中的方法全部是使用static声明的静态方法，就可以直接使用import static的方法导入，如果按照之前的方法，则调用静态方法肯定是类.方法()。 如果用静态导入，则可以直接调用方法，而不需要写类名。

使用**静态导入**可以使被导入类的**静态变量**和**静态方法**在当前类**直接可见**，使用这些静态成员无需再给出他们的类名。

## Operate.java

package org.lxh.demo08.e ;

public class Operate{ // 里面的方法全部都是static类型

public static int add(int i,int j) { // 加法操作

return i + j ;

}

public static int sub(int i,int j) { // 减法操作

return i - j ;

}

public static int mul(int i,int j) { // 乘法操作

return i \* j ;

}

public static int div(int i,int j) { // 除法操作

return i / j ;

}

};

## StaticImportDemo.java

package org.lxh.demo08.f ;

import static org.lxh.demo08.e.Operate.\* ; // 静态导入

public class StaticImportDemo{

public static void main(String args[]){

System.out.println("3 + 3 = " + add(3,3)) ; // 直接调用静态方法

System.out.println("3 - 2 = " + sub(3,2)) ; // 直接调用静态方法

System.out.println("3 \* 3 = " + mul(3,3)) ; // 直接调用静态方法

System.out.println("3 / 3 = " + div(3,3)) ; // 直接调用静态方法

}

};

## 3. jar命令

当开发者为客户开发出了一套java类后，肯定要把这些类交给用户使用，但是如果所有的类是直接通过\*.class的格式给用户，则会比较麻烦，所以一般情况下会将这些\*.class文件压缩成一个文件交付给客户使用，那么这样的文件就称为jar文件(Java Archive File),如果要想生成jar文件，直接使用JDK中bin目录里的jar.exe就可以将所有的类文件进行压缩。

jar命令主要参数：

“C” : 创建新的文档

“V”: 生成详细的输出信息

“F”: 指定存档的文件名

package org.lxh.demo08.demo ;

public class Hello{

public String getInfo(){

return "Hello World!!!" ;

}

};

将其打包编译： javac –d . Hello.java

将其打成jar包： jar –cvf my.jar org

一个jar包如果要想使用的话，则必须配置classpath路径。

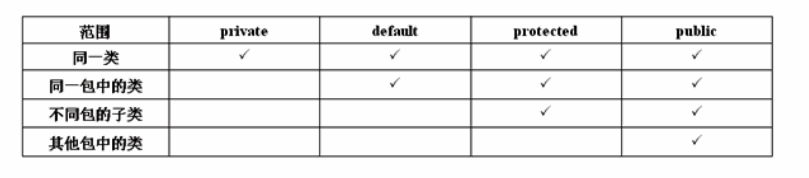
set classpath= 路径

# 2016年8月15日21:44:48

## 1. 访问控制权限

java中存在四种访问权限：

private 、 default、 protected 、public



注意protected访问权限，在不同的包中，只能是不同包的子类才能访问不同包父类的protected成员，如果不是子类，则不能访问。

**HelloDemo.java**

package org.lxh.demo08.g ;

public class HelloDemo{

protected String name = "LXH" ; // 只能在本包及不同包的子类中访问

};

**ProtectedDemo01.java**

package org.lxh.demo08.h ;

import org.lxh.demo08.g.HelloDemo ;

class SubHelloDemo extends HelloDemo{

public void print(){

System.out.println("访问受保护属性：" + super.name) ;

}

};

public class ProtectedDemo01{

public static void main(String args[]){

SubHelloDemo sub = new SubHelloDemo() ;

sub.print() ;

}

};

上面的代码能访问父类的name成员。

**ProtectedDemo02.java**

package org.lxh.demo08.h ;

import org.lxh.demo08.g.HelloDemo ;

public class ProtectedDemo02{

public static void main(String args[]){

HelloDemo sub = new HelloDemo() ;

System.out.println(sub.name) ; // 错误的，不同包的类无法访问

}

};

这段代码这不能访问name成员，因为不是子类。

继承中private属性也是不能被子类访问到的，所以为了方便，不想让外部类(不同包)访问，但是能让子类访问，就可以使用protected进行封装。

## 2. 命名规范

类： 所有单词的首字母大写，如:TestJava

方法：第一个单词的首字母小写，之后每个单词的首字母大写，如：getInfo()

属性：第一个单词的首字母小写，之后每个单词的首字母大写，如：studentName

包：所有单词的字母小写：如：org.lxh.demo

常量：所有单词的字母大写：如：FLAG

# 2016年8月15日12:04:59

## 1. java反射机制

目标：

掌握Object类中getClass()方法的作用

认识java反射机制

object类的方法：

|  |  |
| --- | --- |
| protected [**Object**](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\Object.html) | [**clone**](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\Object.html#clone--)()  Creates and returns a copy of this object. |
| boolean | [**equals**](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\Object.html#equals-java.lang.Object-)([**Object**](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\Object.html) obj)  Indicates whether some other object is "equal to" this one. |
| protected void | [**finalize**](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\Object.html#finalize--)()  Called by the garbage collector on an object when garbage collection determines that there are no more references to the object. |
| [**Class**](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\Class.html)<?> | [**getClass**](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\Object.html#getClass--)()  Returns the runtime class of this Object. |
| int | [**hashCode**](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\Object.html#hashCode--)()  Returns a hash code value for the object. |
| void | [**notify**](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\Object.html#notify--)()  Wakes up a single thread that is waiting on this object's monitor. |
| void | [**notifyAll**](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\Object.html#notifyAll--)()  Wakes up all threads that are waiting on this object's monitor. |
| [**String**](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\String.html) | [**toString**](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\Object.html#toString--)()  Returns a string representation of the object. |
| void | [**wait**](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\Object.html#wait--)()  Causes the current thread to wait until another thread invokes the [**notify()**](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\Object.html#notify--) method or the [**notifyAll()**](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\Object.html#notifyAll--) method for this object. |
| void | [**wait**](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\Object.html#wait-long-)(long timeout)  Causes the current thread to wait until either another thread invokes the [**notify()**](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\Object.html#notify--) method or the [**notifyAll()**](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\Object.html#notifyAll--) method for this object, or a specified amount of time has elapsed. |
| void | [**wait**](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\Object.html#wait-long-int-)(long timeout, int nanos)  Causes the current thread to wait until another thread invokes the [**notify()**](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\Object.html#notify--) method or the [**notifyAll()**](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\Object.html#notifyAll--) method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed. |

在正常情况下，必须知道一个类的完整路劲之后才可以实例化对象，但是在java中允许通过一个对象来找到其所在的类的信息，那么实际上就是class类的功能

在object中定义了以下的方法，此方法将被所有子类继承：

public final Class getClass()

以上的方法返回值是一Class类，实际上此类是java反射的源头，实际上所谓反射从程序运行结果来看也很好理解，即：可以通过对象反射求出类名称。

通过Class可以得到一个类中的完整结构，包括此类中的方法定义，属性定义等。

此类没有构造方法，构造方法被私有化了。

**实例化Class类对象的方法有三种：**

一、通过forName()方法

二、类.class

三、对象.getClass();

package org.lxh.demo15.getclassdemo ;

class X{

};

public class GetClassDemo02{

public static void main(String args[]){

Class<?> c1 = null ; // 指定泛型

Class<?> c2 = null ; // 指定泛型

Class<?> c3 = null ; // 指定泛型

try{

// 以下的操作形式是在开发中最常用的一种形式

c1 = Class.forName("org.lxh.demo15.getclassdemo.X") ;

}catch(ClassNotFoundException e){

e.printStackTrace() ;

}

c2 = new X().getClass() ; // 通过Object类中的方法实例化

c3 = X.class ; // 通过类.class实例化

System.out.println("类名称：" + c1.getName()) ; // 得到类的名称

System.out.println("类名称：" + c2.getName()) ; // 得到类的名称

System.out.println("类名称：" + c3.getName()) ; // 得到类的名称

}

};

## 2. Class类的作用

可以使用Class类完成对象实例化操作

可以通过Constructor类调用有参构造方法完成对象的实例化操作

Class主要是反射的源头，不光可以取得对象所在的类的信息，也可以直接通过Class类的方法进行对象实例化操作，正常情况下，使用关键字new为对象实例化，如果现在已经实例化好了Class对象，则就可以通过class类提供的newInstance()方法实例化对象。

package org.lxh.demo15.instancedemo ;

class Person{

private String name ; // name属性

private int age ; // age属性

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 ;

}

public String toString(){ // 覆写toString()方法

return "姓名：" + this.name + "，年龄：" + this.age ;

}

};

public class InstanceDemo01{

public static void main(String args[]){

Class<?> c = null ; // 声明Class对象

try{

c = Class.forName("org.lxh.demo15.instancedemo.Person") ;

}catch(ClassNotFoundException e){

e.printStackTrace() ;

}

Person per = null ; // 声明Person对象

try{

per = (Person)c.newInstance() ; // 实例化对象

}catch(Exception e){

e.printStackTrace() ;

}

per.setName("李兴华") ; // 设置姓名

per.setAge(30) ; // 设置年龄

System.out.println(per) ; // 内容输出，调用toString()

}

};

通过上面的代码，可以发现，即使不使用关键字new对象也可以进行实例化操作，反射的作用。但是在使用以上操作的时候有一点必须注意，在操作中类中必须存在无参的构造方法，否则无法实现实例化。

上面的代码没有看到构造方法，是因为有个默认的无参构造方法，如果写一个有参的构造方法，则会隐藏无参的构造方法，那么将无法实例化。如果必须要有个有参的构造方法，那么需要再添加一个无参的构造方法。也可以在调用的时候，明确指定要调用的构造方法，并传递参数。

package org.lxh.demo15.instancedemo ;

import java.lang.reflect.Constructor ; // 导入反射机制包

class Person{

private String name ; // name属性

private int age ; // age属性

public Person(String name,int age){

this.setName(name) ;

this.setAge(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 ;

}

public String toString(){ // 覆写toString()方法

return "姓名：" + this.name + "，年龄：" + this.age ;

}

};

public class InstanceDemo03{

public static void main(String args[]){

Class<?> c = null ; // 声明Class对象

try{

c = Class.forName("org.lxh.demo15.instancedemo.Person") ;

}catch(ClassNotFoundException e){

e.printStackTrace() ;

}

Person per = null ; // 声明Person对象

Constructor<?> cons[] = null ;

cons = c.getConstructors() ;

try{

per = (Person)cons[0].newInstance("李兴华",30) ; // 实例化对象

}catch(Exception e){

e.printStackTrace() ;

}

System.out.println(per) ; // 内容输出，调用toString()

}

};

## 3.取得类的结构

可以使用反射取得实现的全部接口

可以使用反射取得一个类所继承的父类

可以使用反射取得一个类的全部构造方法

java.lang.reflect包中的几个子类：

Constructor： 表示类中的构造方法

Field：表示类中的属性

Method：表示类中的方法

这三个类都是AccessibleObject类的子类

package org.lxh.demo15 ;

interface China{ // 定义China接口

public static final String NATIONAL = "China" ; // 定义全局常量

public static final String AUTHOR = "李兴华" ; // 定义全局常量

public void sayChina() ; // 无参的，没有返回值的方法

public String sayHello(String name,int age) ; // 定义有两个参数的方法，并返回内容

}

public class Person implements China{

private String name ;

private int age ;

public Person(){ // 无参构造

}

public Person(String name){

this.name = name ; // 设置name属性

}

public Person(String name,int age){

this(name) ;

this.age = age ;

}

public void sayChina(){ // 覆写方法

System.out.println("作者：" + AUTHOR + "，国籍：" + NATIONAL) ;

}

public String sayHello(String name,int age){

return name + "，你好！我今年：" + age + "岁了！" ;

}

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 ;

}

};

## 3.1 取得类所实现的全部接口

使用Class类中的getInterfaces()方法，然后可以使用getName()方法取得名称。

public Class[] getInterfaces()

package org.lxh.demo15.classinfodemo ;

public class GetInterfaceDemo{

public static void main(String args[]){

Class<?> c1 = null ; // 声明Class对象

try{

c1 = Class.forName("org.lxh.demo15.Person") ; // 实例化对象

}catch(ClassNotFoundException e){

e.printStackTrace() ;

}

Class<?> c[] = c1.getInterfaces() ; // 以数组形式返回实现的全部接口

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

System.out.println("实现的接口名称：" + c[i].getName()) ; // 输出接口名称

}

}

};

## 3.2取得所继承的父类

使用Class类的getSuperClass()方法

public Class<?super T> getSuperclass()

package org.lxh.demo15.classinfodemo ;

public class GetSuperClassDemo{

public static void main(String args[]){

Class<?> c1 = null ; // 声明Class对象

try{

c1 = Class.forName("org.lxh.demo15.Person") ; // 实例化对象

}catch(ClassNotFoundException e){

e.printStackTrace() ;

}

Class<?> c2 = c1.getSuperclass() ; // 取得父类

System.out.println("父类名称：" + c2.getName()) ;

}

};

## 3.3 取得类中的全部构造方法

使用getConstructors()方法

package org.lxh.demo15.classinfodemo ;

import java.lang.reflect.Constructor ; // 导入构造方法的包

public class GetConstructorDemo01{

public static void main(String args[]){

Class<?> c1 = null ; // 声明Class对象

try{

c1 = Class.forName("org.lxh.demo15.Person") ; // 实例化对象

}catch(ClassNotFoundException e){

e.printStackTrace() ;

}

Constructor<?> con[] = c1.getConstructors() ; // 取得一个类中的全部构造

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

System.out.println("构造方法：" + con[i]) ; // 输出构造，直接打印

}

}

};

Constructor类中还存在以下的几个方法:

(1) 取得修饰符：public int getModifiers()

(2) 取得方法名称：public String getName()

(3) 取得参数类型: public Class<?>[] getParameterTypes()

package org.lxh.demo15.classinfodemo ;

import java.lang.reflect.Constructor ; // 导入构造方法的包

public class GetConstructorDemo02{

public static void main(String args[]){

Class<?> c1 = null ; // 声明Class对象

try{

c1 = Class.forName("org.lxh.demo15.Person") ; // 实例化对象

}catch(ClassNotFoundException e){

e.printStackTrace() ;

}

Constructor<?> con[] = c1.getConstructors() ; // 取得一个类中的全部构造

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

Class<?> p[] = con[i].getParameterTypes() ; // 得到构造方法中的全部参数

System.out.print("构造方法：" ) ; // 输出构造，直接打印

System.out.print(con[i].getModifiers() + " ") ; // 得到修饰符

System.out.print(con[i].getName()) ; // 取得构造方法的名字

System.out.print("(") ;

for(int j=0;j<p.length;j++){

System.out.print(p[j].getName() + " arg" + i) ;

if(j<p.length-1){

// 判断此是否是最后一个参数

System.out.print(","); // 输出“,”

}

}

System.out.println("){}") ;

}

}

};

package org.lxh.demo15.classinfodemo ;

import java.lang.reflect.Constructor ; // 导入构造方法的包

import java.lang.reflect.Modifier ; // 导入构造方法的包

public class GetConstructorDemo03{

public static void main(String args[]){

Class<?> c1 = null ; // 声明Class对象

try{

c1 = Class.forName("org.lxh.demo15.Person") ; // 实例化对象

}catch(ClassNotFoundException e){

e.printStackTrace() ;

}

Constructor<?> con[] = c1.getConstructors() ; // 取得一个类中的全部构造

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

Class<?> p[] = con[i].getParameterTypes() ; // 得到构造方法中的全部参数

System.out.print("构造方法：" ) ; // 输出构造，直接打印

int mo = con[i].getModifiers() ; // 得到所要的访问权限

System.out.print(Modifier.toString(mo) + " ") ; // 得到修饰符

System.out.print(con[i].getName()) ; // 取得构造方法的名字

System.out.print("(") ;

for(int j=0;j<p.length;j++){

System.out.print(p[j].getName() + " arg" + i) ;

if(j<p.length-1){

// 判断此是否是最后一个参数

System.out.print(","); // 输出“,”

}

}

System.out.println("){}") ;

}

}

};

## 3.4 取得类中的方法

要想取得一个类中的全部方法，可以使用Class类中的getDeclaredMethods()方法，此方法返回一个Method类的对象数组，而如果想要进一步取得方法的具体信息，例如：方法的参数，抛出的异常声明等等，则必须依靠Method类。

public [Method](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\reflect\Method.html)[] getDeclaredMethods()

throws [SecurityException](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\SecurityException.html) 输出本类的方法

public [Method](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\reflect\Method.html)[] getMethods()

throws [SecurityException](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\SecurityException.html) 输出所有方法，包括父类（到object类）

package org.lxh.demo15.classinfodemo ;

import java.lang.reflect.Method ; // 导入构造方法的包

import java.lang.reflect.Modifier ; // 导入构造方法的包

public class GetMethodDemo{

public static void main(String args[]){

Class<?> c1 = null ; // 声明Class对象

try{

c1 = Class.forName("org.lxh.demo15.Person") ; // 实例化对象

}catch(ClassNotFoundException e){

e.printStackTrace() ;

}

Method m[] = c1.getMethods() ; // 取得全部方法

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

Class<?> r = m[i].getReturnType() ; // 得到返回值类型

Class<?> p[] = m[i].getParameterTypes() ; // 取得全部参数的类型

int xx = m[i].getModifiers() ; // 得到修饰符

System.out.print(Modifier.toString(xx) + " ") ; // 输出修饰符

System.out.print(r + " ") ;

System.out.print(m[i].getName()) ;

System.out.print("(") ;

for(int j=0;j<p.length;j++){

System.out.print(p[j].getName() + " " + "arg" + j) ;

if(j<p.length-1){

System.out.print(",") ;

}

}

Class<?> ex[] = m[i].getExceptionTypes() ; // 取出异常

if(ex.length>0){

System.out.print(") throws ") ;

}else{

System.out.print(")") ;

}

for(int j=0;j<ex.length;j++){

System.out.print(ex[j].getName()) ;

if(j<p.length-1){

System.out.print(",") ;

}

}

System.out.println() ;

}

}

};

## 3.5 取得类中的属性

public [Method](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\reflect\Method.html)[] getMethods()

throws [SecurityException](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\SecurityException.html)

public [Method](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\reflect\Method.html) getMethod([String](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\String.html) name,

[Class](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\Class.html)<?>... parameterTypes)

throws [NoSuchMethodException](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\NoSuchMethodException.html),

[SecurityException](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\SecurityException.html)

public [Method](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\reflect\Method.html)[] getDeclaredMethods()

throws [SecurityException](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\SecurityException.html)

package org.lxh.demo15.classinfodemo ;

import java.lang.reflect.Field ; // 导入构造方法的包

import java.lang.reflect.Modifier ; // 导入构造方法的包

public class GetFieldDemo{

public static void main(String args[]){

Class<?> c1 = null ; // 声明Class对象

try{

c1 = Class.forName("org.lxh.demo15.Person") ; // 实例化对象

}catch(ClassNotFoundException e){

e.printStackTrace() ;

}

{ // 本类属性

Field f[] = c1.getDeclaredFields() ; // 取得本类中的属性

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

Class<?> r = f[i].getType() ; // 得到属性类型

int mo = f[i].getModifiers() ; // 得到修饰符的数字

String priv = Modifier.toString(mo) ; // 还原修饰符

System.out.print("本类属性：") ;

System.out.print(priv + " ") ;

System.out.print(r.getName() + " ") ; // 得到属性类型

System.out.print(f[i].getName()) ; // 输出属性名称

System.out.println(" ;") ;

}

}

{ // 公共属性

Field f[] = c1.getFields() ; // 取得本类中的公共属性

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

Class<?> r = f[i].getType() ; // 得到属性类型

int mo = f[i].getModifiers() ; // 得到修饰符的数字

String priv = Modifier.toString(mo) ; // 还原修饰符

System.out.print("公共属性：") ;

System.out.print(priv + " ") ;

System.out.print(r.getName() + " ") ; // 得到属性类型

System.out.print(f[i].getName()) ; // 输出属性名称

System.out.println(" ;") ;

}

}

}

};

# 2016年8月16日23:21:25

## 1. 反射机制的深入研究

目标：

使用反射调用类中的指定方法

使用反射完成setter及getter方法

使用反射直接操作类中的属性

使用反射操作数组

## 1.1 通过反射调用类中的方法

正常情况下，一个类的对象产生之后，就可以直接调用类中的分离，如果想要调用的话，则必须知道要调用的方法名称是什么，之后通过Class类中的getMethod方法得到Method对象，之后通过此方对象来执行方法，但是在方法调用的时候，因为会牵扯到方法中的参数问题，所以通过getMethod取得的时候，必须设置好需要的参数类型。

public [Method](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\reflect\Method.html) getMethod([String](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\String.html) name,

[Class](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\Class.html)<?>... parameterTypes)

throws [NoSuchMethodException](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\NoSuchMethodException.html),

[SecurityException](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\SecurityException.html)

调用有参数的方法时，getMethod方法要设置参数类型，比如：String.class,int.class

调用没有参数的方法时，不用设置类型

package org.lxh.demo15.invokedemo;

import java.lang.reflect.Method ;

public class InvokeSayChinaDemo

{

public static void main(String[] args)

{

Class<?> cl= null;

try

{

cl = Class.forName("org.lxh.demo15.Person");

}

catch(Exception e)

{

}

try{

Method met =cl.getMethod("sayChina");

met.invoke(cl.newInstance());

}

catch(Exception e)

{

e.printStackTrace();

}

}

}

以上程序，通过class类的getMethod()方法根据一个类中的方法名称取得Method对象，并通过invoke调用指定的方法，但是在使用invoke方法的时候必须传入一个类的实例化对象，因为在sayChina()方法上没有任何的参数，所以此处没有设置参数类型和参数内容。

如果在调用的方法中存在了参数，则必须设置参数的类型及内容。

package org.lxh.demo15.invokedemo ;

import java.lang.reflect.Method ;

public class InvokeSayHelloDemo{

public static void main(String args[]){

Class<?> c1 = null ;

try{

c1 = Class.forName("org.lxh.demo15.Person") ; // 实例化Class对象

}catch(Exception e){}

try{

Method met = c1.getMethod("sayHello",String.class,int.class) ; // 找到sayChina()方法

String rv = null ;

rv = (String)met.invoke(c1.newInstance(),"李兴华",30) ; // 调用方法

System.out.println(rv) ;

}catch(Exception e){

e.printStackTrace() ;

}

}

};

## 1.2 通过反射调用setter和getter

setter和getter是一个标准的属性访问方法，如果一个雷达属性被封装，则必须通过setter和getter方法设置和取得，实际上此方法的操作之所以要这样规定，主要是由于反射机制可以给予支持的。

package org.lxh.demo15.invokedemo ;

import java.lang.reflect.Method ;

public class InvokeSetGetDemo{

public static void main(String args[]){

Class<?> c1 = null ;

Object obj = null ;

try{

c1 = Class.forName("org.lxh.demo15.Person") ; // 实例化Class对象

}catch(Exception e){}

try{

obj = c1.newInstance() ;

}catch(Exception e){}

setter(obj,"name","李兴华",String.class) ; // 调用setter方法

setter(obj,"age",30,int.class) ; // 调用setter方法

System.out.print("姓名：") ;

getter(obj,"name") ;

System.out.print("年龄：") ;

getter(obj,"age");

}

/\*\*

Object obj：要操作的对象

String att：要操作的属性

Object value：要设置的属性内容

Class<?> type：要设置的属性类型

\*/

public static void setter(Object obj,String att,Object value,Class<?> type){

try{

Method met = obj.getClass().getMethod("set"+initStr(att),type) ; // 得到setter方法

met.invoke(obj,value) ; // 设置setter的内容

}catch(Exception e){

e.printStackTrace() ;

}

}

public static void getter(Object obj,String att){

try{

Method met = obj.getClass().getMethod("get"+initStr(att)) ; // 得到setter方法

System.out.println(met.invoke(obj)) ; // 调用getter取得内容

}catch(Exception e){

e.printStackTrace() ;

}

}

public static String initStr(String old){ // 将单词的首字母大写

String str = old.substring(0,1).toUpperCase() + old.substring(1) ;

return str ;

}

};

## 1.3 通过反射调用属性

如果现在需要操作一个类中的属性，则也可以通过Field完成，而不必麻烦的通过setter和getter得到公共属性。

public [Field](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\reflect\Field.html) getField([String](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\String.html) name)

throws [NoSuchFieldException](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\NoSuchFieldException.html),

[SecurityException](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\SecurityException.html)

public [Field](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\reflect\Field.html)[] getFields()

throws [SecurityException](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\SecurityException.html)

public [Field](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\reflect\Field.html) getDeclaredField([String](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\String.html) name)

throws [NoSuchFieldException](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\NoSuchFieldException.html),

[SecurityException](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\SecurityException.html)

public [Field](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\reflect\Field.html)[] getDeclaredFields()

throws [SecurityException](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\SecurityException.html)

在包java.lang.reflect.Field 中

取得属性内容：

public [Object](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\Object.html) get([Object](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\Object.html) obj)

throws [IllegalArgumentException](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\IllegalArgumentException.html),

[IllegalAccessException](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\IllegalAccessException.html)

设置属性内容：

public void set([Object](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\Object.html) obj,

[Object](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\Object.html) value)

throws [IllegalArgumentException](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\IllegalArgumentException.html),

[IllegalAccessException](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\IllegalAccessException.html)

在访问私有属性的时候，必须让这个属性可见

public void setAccessible(boolean flag)

throws [SecurityException](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\SecurityException.html)

package org.lxh.demo15.invokedemo ;

import java.lang.reflect.Field ;

public class InvokeFieldDemo{

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

Class<?> c1 = null ;

Object obj = null ;

c1 = Class.forName("org.lxh.demo15.Person") ; // 实例化Class对象

obj = c1.newInstance() ;

Field nameField = null ;

Field ageField = null ;

nameField = c1.getDeclaredField("name") ; // 取得name属性

ageField = c1.getDeclaredField("age") ; // 取得name属性

nameField.setAccessible(true) ; // 此属性对外部可见

ageField.setAccessible(true) ; // 此属性对外部可见

nameField.set(obj,"李兴华") ; // 设置name属性内容

ageField.set(obj,30) ; // 设置age属性内容

System.out.println("姓名：" + nameField.get(obj)) ;

System.out.println("年龄：" + ageField.get(obj)) ;

}

};

## 1.4 通过反射操作数组

反射机制不光能用在类中，也可以应用在任意类型的引用数据类型上，当然，则就包含了数组，数组使用Array类完成。

class类中存在以下的一个方法：

public [Class](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\Class.html)<?> getComponentType()

java.lang.reflect.Array 中有如下方法：

得到数组指定下标的内容：

public static [Object](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\Object.html) get([Object](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\Object.html) array,

int index)

throws [IllegalArgumentException](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\IllegalArgumentException.html),

[ArrayIndexOutOfBoundsException](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\ArrayIndexOutOfBoundsException.html)

修改内容：

public static void set([Object](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\Object.html) array,

int index,

[Object](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\Object.html) value)

throws [IllegalArgumentException](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\IllegalArgumentException.html),

[ArrayIndexOutOfBoundsException](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\ArrayIndexOutOfBoundsException.html)

开辟新数组

public static [Object](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\Object.html) newInstance([Class](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\Class.html)<?> componentType,

int... dimensions)

throws [IllegalArgumentException](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\IllegalArgumentException.html),

[NegativeArraySizeException](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\NegativeArraySizeException.html)

package org.lxh.demo15.invokedemo ;

import java.lang.reflect.Array ;

public class ClassArrayDemo{

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

int temp[] = {1,2,3} ;// 声明一整型数组

Class<?> c = temp.getClass().getComponentType() ; // 取得数组的Class对象

System.out.println("类型：" + c.getName()) ; // 取得数组类型名称

System.out.println("长度：" + Array.getLength(temp)) ;

System.out.println("第一个内容：" + Array.get(temp,0)) ;

Array.set(temp,0,6) ;

System.out.println("第一个内容：" + Array.get(temp,0)) ;

}

};

使用Array类还可以修改数组的大小

创建一个新的数组，把旧的数组的内容拷贝到新的数组里面

package org.lxh.demo15.invokedemo ;

import java.lang.reflect.Array ;

public class ChangeArrayDemo{

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

int temp[] = {1,2,3} ;// 声明一整型数组

int newTemp[] = (int []) arrayInc(temp,5) ; // 重新开辟空间5

print(newTemp) ;

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

String t[] = {"lxh","mldn","mldnjava"} ;

String nt[] = (String [])arrayInc(t,8) ;

print(nt) ;

}

public static Object arrayInc(Object obj,int len){

Class<?> c = obj.getClass() ;

Class<?> arr = c.getComponentType() ; // 得到数组的

Object newO = Array.newInstance(arr,len) ; // 开辟新的大小

int co = Array.getLength(obj) ;

System.arraycopy(obj,0,newO,0,co) ; // 拷贝内容

return newO ;

}

public static void print(Object obj){ // 数组输出

Class<?> c = obj.getClass() ;

if(!c.isArray()){ // 判断是否是数组

return;

}

Class<?> arr = c.getComponentType() ;

System.out.println(arr.getName()+"数组的长度是：" + Array.getLength(obj)) ; // 输出数组信息

for(int i=0;i<Array.getLength(obj);i++){

System.out.print(Array.get(obj,i) + "、") ; // 通过Array输出

}

}

};

# 2016年8月17日10:39:49

## 1. 动态代理

代理设计：一个操作的接口有两个子类，其中一个是真实主题的实现类，另外一个是代理类，代理实现类要完成比真实实现类更多的内容，而且本身还需要处理一些与具体业务相关的程序代码。

要实现动态代理机制，需要java.lang.reflect. InvocationHandler接口和java.lang.reflect. Proxy类

[Object](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\Object.html) invoke([Object](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\Object.html) proxy,

[Method](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\reflect\Method.html) method,

[Object](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\Object.html)[] args)

throws [Throwable](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\Throwable.html)

Proxy类是专门完成代理操作的类，可以通过此类为一个或多个接口动态生成实现类，

public static [Object](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\Object.html) newProxyInstance([ClassLoader](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\ClassLoader.html) loader,

[Class](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\Class.html)<?>[] interfaces,

[InvocationHandler](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\reflect\InvocationHandler.html) h)

throws [IllegalArgumentException](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\IllegalArgumentException.html)

import java.lang.reflect.InvocationHandler ;

import java.lang.reflect.Proxy ;

import java.lang.reflect.Method ;

interface Subject{

public String say(String name,int age) ; // 定义抽象方法say

}

class RealSubject implements Subject{ // 实现接口

public String say(String name,int age){

return "姓名：" + name + "，年龄：" + age ;

}

};

class MyInvocationHandler implements InvocationHandler{

private Object obj ;

public Object bind(Object obj){

this.obj = obj ; // 真实主题类

return Proxy.newProxyInstance(obj.getClass().getClassLoader(),obj.getClass().getInterfaces(),this) ;

}

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

Object temp = method.invoke(this.obj,args) ; // 调用方法

return temp ;

}

};

public class DynaProxyDemo{

public static void main(String args[]){

Subject sub = (Subject)new MyInvocationHandler().bind(new RealSubject()) ;

String info = sub.say("李兴华",30) ;

System.out.println(info) ;

}

};

## 2.工厂设计模式

工厂设计模式最大的好处是可以在应用进行接耦合操作。

package org.lxh.demo15.factorydemo01 ;

interface Fruit{

public void eat() ; // 吃水果

}

class Apple implements Fruit{

public void eat(){ // 覆写eat()方法

System.out.println("\*\* 吃苹果");

}

};

class Orange implements Fruit{

public void eat(){

System.out.println("\*\* 吃橘子") ;

}

};

class Factory{

public static Fruit getInstance(String className){

Fruit fruit = null ;

try{

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

}catch(Exception e){

e.printStackTrace() ;

}

return fruit ;

}

};

public class FactoryDemo01{

public static void main(String args[]){

Fruit f = Factory.getInstance("org.lxh.demo15.factorydemo01.Apple") ;

if(f!=null){

f.eat() ;

}

}

};

以上的操作虽然可以通过反射取得接口的实例，但是在操作的时候需要传入完整的包.类名称，而且用户也无法知道一个接口有多少个可以使用的子类，所以此时可以通过属性文件的形式配置所要的子类信息。

程序运行的时候，就可以将属性文件的内容读取出来，之后直接操作属性文件中的key，就可以避免输入过长的类路径。

package org.lxh.demo15.factorydemo02 ;

import java.util.Properties ;

import java.io.File ;

import java.io.FileOutputStream ;

import java.io.FileInputStream ;

interface Fruit{

public void eat() ; // 吃水果

}

class Apple implements Fruit{

public void eat(){ // 覆写eat()方法

System.out.println("\*\* 吃苹果");

}

};

class Orange implements Fruit{

public void eat(){

System.out.println("\*\* 吃橘子") ;

}

};

class Init{

public static Properties getPro(){

Properties pro = new Properties() ;

File f = new File("d:\\fruit.properties") ; // 找到属性文件

try{

if(f.exists()){ // 文件存在

pro.load(new FileInputStream(f)) ; // 读取属性

}else{

pro.setProperty("apple","org.lxh.demo15.factorydemo02.Apple") ;

pro.setProperty("orange","org.lxh.demo15.factorydemo02.Orange") ;

pro.store(new FileOutputStream(f),"FRUIT CLASS") ;

}

}catch(Exception e){}

return pro ;

}

};

class Factory{

public static Fruit getInstance(String className){

Fruit fruit = null ;

try{

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

}catch(Exception e){

e.printStackTrace() ;

}

return fruit ;

}

};

public class FactoryDemo02{

public static void main(String args[]){

Properties pro = Init.getPro() ;

Fruit f = Factory.getInstance(pro.getProperty("apple")) ;

if(f!=null){

f.eat() ;

}

}

};

# 2016年8月9日10:46:52

**1. Annotation注释**

系统自带的annotation

@Override : 覆写

@Deprecated :不赞成使用

@ SuppressWarnings:压制安全警告

1.1 override的作用是保证覆写的正确性

覆写父类的方法的时候，有可能方法写错了，加上@Override后，如果覆写错误，编译会提示错误。

1.2 添加Deprecated注解后，在调用这个方法编译会出现安全警告

该注解不仅可以用在方法前，还可以用在类前面，表示类也不能使用了

1.3 SuppressWarnings

@SuppressWarnings("unchecked")

public static void main(String[] args)

{

Demo d= new Demo(); //没有指明泛型类型，会出现警告

d.setVar("hello") ;

System.out.println(d.getVar());

}

关键字:

(1). deprecation 使用了不赞成的类或方法的警告

(2). unchecked 执行了未检查的转换时警告，例如：泛型操作中没有指定泛型类型

(3).fallthrough 当switch程序块直接通往下种情况而没有break时的警告

(4).path 在类的路径、源文件路径中有不存在的路径时的警告

(5). serial当在可序列化的类上缺少serialVerisonUID定义是的警告

(6). finally 任何finally子句不能正常完成时的警告

(7). all 关于以上所有情况的警告

使用数组传递更多的关键字

@ SuppressWarnings(value={“unchecked”,”deprecation”})

# 2016年8月17日20:52:04

## 1. 自定义Annotation

Annotation定义格式：

public @interface Annotation名称

{

数据类型 变量名称():

}

定义一个简单的Annotation

**MyDefaultAnnotationNoneParam.java**

public @interface MyDefaultAnnotationNoneParam{

}

之后就可以直接在程序中使用“@Annotation名称”的格式

**UseDefaultAnnotationNoneParam.java**

@MyDefaultAnnotationNoneParam

class Demo{

};

此时就表示在Demo类上使用Annotation

还可以向Annotation中设置内容，使用变量接收参数。

**MyDefaultAnnotationSingleParam.java**

public @interface MyDefaultAnnotationSingleParam{

public String value() ; // 接收设置的内容

}

在使用的时候必须清楚的指明变量的内容

**UseDefaultAnnotationSingleParam01.java**

@MyDefaultAnnotationSingleParam("李兴华")

class Demo{

};

以上的 参数是要赋给value属性的

也可以明确的标记，表示内容赋给哪个参数

@MyDefaultAnnotationSingleParam(value="李兴华")

class Demo{

};

既然可以设置一个参数，则也就可以同时设置多个参数

**MyDefaultAnnotationMoreParam.java**

public @interface MyDefaultAnnotationMoreParam{

public String key() ;

public String value() ; // 接收设置的内容

}

**UseDefaultAnnotationMoreParam.java**

@MyDefaultAnnotationMoreParam(key="MLDN",value="李兴华")

class Demo{

};

也可以设置一个数组进去

**MyDefaultAnnotationArrayParam.java**

public @interface MyDefaultAnnotationArrayParam{

public String[] value() ; // 接收设置的内容

}

**UseDefaultAnnotationArrayParam.java**

@MyDefaultAnnotationArrayParam(value={"MLDN","李兴华"})

class Demo{

};

以上所定义的全部Annotation中有一个特点，所有的参数内容需要在使用注释的时候设置上去，那么也可以为一个参数设置默认的内容，在声明的时候使用default即可。

**MyDefaultAnnotationValue.java**

public @interface MyDefaultAnnotationValue{

public String key() default "MLDN" ; // 指定好了默认值

public String value() default "李兴华" ; // 指定好了默认值

}

在使用此Annotation的时候就可以不用设置内容

@MyDefaultAnnotationValue

class Demo{

};

在操作中，对于一个Annotation而言有时候会固定其取值范围，只能取固定的几个值，那么这个时候实际上就需要依靠枚举

**MyName.java**

public enum MyName{ // 定义枚举类型

MLDN,LXH,LI ;

}

**MyDefaultAnnotationEnum.java**

public @interface MyDefaultAnnotationEnum{

public MyName name() default MyName.LXH ; // 指定默认值

}

**UseDefaultAnnotationEnum.java**

@MyDefaultAnnotationEnum(name=MyName.LI) // 只能从固定的枚举中取内容

class Demo{

};

## 2. Retention和RetentionPolicy

**在Annotation，可以使用Retention定义一个Annotation的保存范围**

是在Annotation的定义中设置该属性的

[@Documented](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\annotation\Documented.html)

[@Retention](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\annotation\Retention.html)([value](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\annotation\Retention.html#value--)=[RUNTIME](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\annotation\RetentionPolicy.html#RUNTIME))

[@Target](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\annotation\Target.html)([value](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\annotation\Target.html#value--)=[ANNOTATION\_TYPE](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\annotation\ElementType.html#ANNOTATION_TYPE))

public @interface Retention

{

RetetionPolicy value();

}

在以上的Retetion定义中存在了一个RetetionPolicy的变量，此变量用于指定Annotation的保存范围，RetentionPolicy包含三种范围。

[java.lang.Object](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\Object.html)

[java.lang.Enum](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\Enum.html)<[RetentionPolicy](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\annotation\RetentionPolicy.html)>

java.lang.annotation.RetentionPolicy

**All Implemented Interfaces:**

[Serializable](file:///D:\jdk-8u101-docs-all\docs\api\java\io\Serializable.html), [Comparable](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\Comparable.html)<[RetentionPolicy](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\annotation\RetentionPolicy.html)>

**RetentionPolicy**是个枚举类型，其取值范围：

1. SOURCE :

此Annotation类型的信息只会 保留在程序源文件之中(\*.java),编译之后不会保存在编译好的类文件(\*.class)之中。

2. CLASS:

此Annotation类型将保留在程序源文件(\*.java)和编译之后的类文件(\*.class)之中，在使用此类的时候，这些Annotation信息将不会被加载到虚拟机JVM之中，如果一个Annotation声明时没有指定范围，则默认是此范围

3. RUNTIME

此Annotation类型的信息保留在源文件(\*.java)、类文件(\*.java)在执行时也会加载到JVM之中。

在三个范围中，最需要关心的就是RUNTIME范围，因为在执行的时候起作用。

使用Rentention指定一个Annotation的范围，范围为RUNTIME范围

import java.lang.annotation.Retention ;

import java.lang.annotation.RetentionPolicy ;

@Retention(value=RetentionPolicy.RUNTIME) // 表示此Annotation在运行时有效

public @interface MyDefaultRententionAnnotation{

public String name() default "李兴华" ;

}

此Annotation可以在运行时起作用，而且在以后要讲解反射与Annotation的操作中，也必须是此种范围。

**内建Annotation的RetentinoPolicy**

Override采用的是@Retention(value=SOURCE),只能在源文件中出现；不会影响程序运行

Deprecated采用的是@Retention(value=RUNTIME),可以在执行时出现

SupressWarning采用的是@Retention(value=SOURCE)，只能在源文件中出现

**2018年4月6日20:54:15**

**补充：**

**1、元注解（meta-annotation）：**

元注解的作用就是负责注解其他注解。Java5.0定义了4个标准的meta-annotation类型，它们被用来提供对其它 annotation类型作说明。Java5.0定义的元注解：  
　　　　1.@Target,  
　　　　2.@Retention,  
　　　　3.@Documented,  
　　　　4.@Inherited

**@Target：**

　　　@Target说明了Annotation所修饰的对象范围：Annotation可被用于 packages、types（类、接口、枚举、Annotation类型）、类型成员（方法、构造方法、成员变量、枚举值）、方法参数和本地变量（如循环变量、catch参数）。在Annotation类型的声明中使用了target可更加明晰其修饰的目标。

　**作用：用于描述注解的使用范围（即：被描述的注解可以用在什么地方）**

**取值(ElementType)有：**

　　　　1.CONSTRUCTOR:用于描述构造器  
　　　　2.FIELD: 只能修饰成员变量  
　　　　3.LOCAL\_VARIABLE:用于描述局部变量  
　　　　4.METHOD:用于描述方法  
　　　　5.PACKAGE:用于描述包  
　　　　6.PARAMETER:用于描述参数  
　　　　7.TYPE:用于描述类、接口(包括注解类型) 或enum声明

8.ANNOTATION\_TYPE: 只能修饰Annotation，比如@Target

JDK API上的说明：

## java.lang.annotation  注释类型 Target

[@Documented](file:///D:\java%20API\api\java\lang\annotation\Documented.html)

[@Retention](file:///D:\java%20API\api\java\lang\annotation\Retention.html)([value](file:///D:\java%20API\api\java\lang\annotation\Retention.html#value())=[RUNTIME](file:///D:\java%20API\api\java\lang\annotation\RetentionPolicy.html#RUNTIME))

[@Target](file:///D:\java%20API\api\java\lang\annotation\Target.html)([value](file:///D:\java%20API\api\java\lang\annotation\Target.html#value())=[ANNOTATION\_TYPE](file:///D:\java%20API\api\java\lang\annotation\ElementType.html#ANNOTATION_TYPE))

public @interface **Target**

指示注释类型所适用的程序元素的种类。如果注释类型声明中不存在 Target 元注释，则声明的类型可以用在任一程序元素上。如果存在这样的元注释，则编译器强制实施指定的使用限制。 例如，此元注释指示该声明类型是其自身，即元注释类型。它只能用在注释类型声明上：

@Target(ElementType.ANNOTATION\_TYPE)

public @interface MetaAnnotationType {

...

}

此元注释指示该声明类型只可作为复杂注释类型声明中的成员类型使用。它不能直接用于注释：

@Target({})

public @interface MemberType {

...

}

这是一个编译时错误，它表明一个 ElementType 常量在 Target 注释中出现了不只一次。例如，以下元注释是非法的：

@Target({ElementType.FIELD, ElementType.METHOD, ElementType.FIELD})

public @interface Bogus {

...

}



**@Documented:**

**@**Documented用于描述其它类型的annotation应该被作为被标注的程序成员的公共API，因此可以被例如javadoc此类的工具文档化。Documented是一个标记注解，没有成员。

[复制代码](javascript:void(0);)

@Target(ElementType.FIELD)

@Retention(RetentionPolicy.RUNTIME)

@Documented

public @interface Column {

public String name() default "fieldName";

public String setFuncName() default "setField";

public String getFuncName() default "getField";

public boolean defaultDBValue() default false;

}

[复制代码](javascript:void(0);)

# 2016年8月17日23:04:57

## 反射与Annotation

目标：

可以通过java反射机制取得全部Annotation

可以通过java反射机制取得指定的Annotation

**一个Annotation如果要想让其变得有意义，则必须结合反射机制取得Annotation中设置的全部内容。**

class类或者method类提供了getAnnotation方法，用于获取该类类或者该方法上面的annotation。，然后调用该annotation的value()方法，就可以获取到给该类或者方法设置注解的内容了。

然后通过反射，找到该类，比如Class<?> handlerType = activity.getClass();

然后通过class的getMethod方法获取到想要的方法，比如：Method setContentViewMethod = handlerType.getMethod(**"setContentView"**, **int**.**class**);

获取到想哟的方法后就调用这个方法既可以了,比如：setContentViewMethod.invoke(activity, viewId);

有个很好的开源实例是xUtils3这个Android上的开源工具，这是一个很强大的http工具，但其中还包含了view的注解工具，能直接在activity上通过注解的方式设置contentView。

注解的定义如下：

@Target(ElementType.***TYPE***)  
@Retention(RetentionPolicy.***RUNTIME***)  
**public** @**interface** ContentView {  
 **int** value();  
}

反射的实现过程：

@Override  
public void inject(Activity activity) {  
 //获取Activity的ContentView的注解  
 Class<?> handlerType = activity.getClass();  
 try {  
 ContentView contentView = findContentView(handlerType);  
 if (contentView != null) {  
 int viewId = contentView.value();  
 if (viewId > 0) {  
 Method setContentViewMethod = handlerType.getMethod("setContentView", int.class);  
 setContentViewMethod.invoke(activity, viewId);  
 }  
 }  
 } catch (Throwable ex) {  
 LogUtil.e(ex.getMessage(), ex);  
 }  
 injectObject(activity, handlerType, new ViewFinder(activity));  
}

**private static ContentView findContentView(Class<?> thisCls) {  
 if (thisCls == null || *IGNORED*.contains(thisCls)) {  
 return null;  
 }  
 ContentView contentView = thisCls.getAnnotation(ContentView.class);  
 if (contentView == null) {  
 return *findContentView*(thisCls.getSuperclass());  
 }  
 return contentView;  
}**

现在的问题是：在给一个类或者方法设置自定义注解后，注解是怎么运行的呢，是什么时候运行的呢，

看了下xUtils3的源码，明白了注解是什么时候运行的了，简单说或就是在需要的时候运行，网上的例子是在java的 main方法里面通过反射来找到类，然后找到annotation，然后得到注解的参数内容，并通过反射来得到方法，通过反射的方法设置注解参数。

在xUtils3里面也是类似的，它所有个activity都继承自BaseActivity，在这个BaseActivity的oncreate方法里面调用反射的方法来设置注解的内容，

**public class** BaseActivity **extends** AppCompatActivity {  
  
 @Override  
 **protected void** onCreate(Bundle savedInstanceState) {  
 **super**.onCreate(savedInstanceState);  
 **x.*view*().inject(this);** }  
}

在这个inject方法里面，传递该activity，然后会设置activity的contentview ，同过setContentView方法来设置，对于acitvity里面的每个子元素的注解，使用这个方法Field[] fields = handlerType.getDeclaredFields();来得到所有使用过注解的子元素。

那么play中的安全认证注解又是怎么运行的呢？

@Security.Authenticated(Secured.**class**)

**public** **class** Main **extends** Controller{

**public** Result index() {

**return** ok(views.html.main.render());

}

}

在Class类中与Annotation相关的操作方法：

public <A extends [Annotation](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\annotation\Annotation.html)> A getAnnotation([Class](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\Class.html)<A> annotationClass) 如果在一个元素中存在注释，则取得全部注释

public [Annotation](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\annotation\Annotation.html)[] getAnnotations() 返回此元素的所有注释

public <A extends [Annotation](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\annotation\Annotation.html)> A getDeclaredAnnotation([Class](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\Class.html)<A> annotationClass) 返回直接存放在此元素上的所有注释

public boolean isAnnotation() 判断元素是否表示一个注释

public boolean isAnnotationPresent([Class](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\Class.html)<? extends [Annotation](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\annotation\Annotation.html)> annotationClass) 判断一个元素上是否存在注释

**取得所有的Annotation**

**SimpleBeanOne.java**

package org.lxh.demo16.reflectannotation ;

public class SimpleBeanOne{

@SuppressWarnings("unchecked")

@Deprecated

@Override

public String toString(){

return "Hello LiXingHua!!!" ;

}

};

**ReflectDemo01.java**

import java.lang.annotation.Annotation ;

import java.lang.reflect.Method ;

public class ReflectDemo01{

public static void main(String args[]) throws Exception{ // 所有异常抛出

Class <?> c = null ;

c = Class.forName("org.lxh.demo16.reflectannotation.SimpleBeanOne") ;

Method toM = c.getMethod("toString") ; // 找到toString()方法

Annotation an[] = toM.getAnnotations() ; // 取得全部的Annotation

for(Annotation a:an){ // 使用 foreach输出

System.out.println(a) ;

}

}

};

输出：@java.lang.Deprecated()

因为只有Deprecated的Retention是RUNTIME

**MyDefaultAnnotationReflect.java**

package org.lxh.demo16.reflectannotation ;

import java.lang.annotation.Retention ;

import java.lang.annotation.RetentionPolicy ;

@Retention(value=RetentionPolicy.RUNTIME) // 此Annotation在类执行时依然有效

public @interface MyDefaultAnnotationReflect{

public String key() default "LXH" ;

public String value() default "李兴华" ;

}

**SimpleBeanTwo.java**

package org.lxh.demo16.reflectannotation ;

public class SimpleBeanTwo{

@SuppressWarnings("unchecked")

@Deprecated

@Override

@MyDefaultAnnotationReflect(key="MLDN",value="www.mldnjava.cn")

public String toString(){

return "Hello LiXingHua!!!" ;

}

};

**ReflectDemo02.java**

import org.lxh.demo16.reflectannotation.MyDefaultAnnotationReflect ;

import java.lang.annotation.Annotation ;

import java.lang.reflect.Method ;

public class ReflectDemo02{

public static void main(String args[]) throws Exception{ // 所有异常抛出

Class <?> c = null ;

c = Class.forName("org.lxh.demo16.reflectannotation.SimpleBeanTwo") ;

Method toM = c.getMethod("toString") ; // 找到toString()方法

if(toM.isAnnotationPresent(MyDefaultAnnotationReflect.class)){

// 判断是否是指定的Annotation

MyDefaultAnnotationReflect mda = null ;

mda = toM.getAnnotation(MyDefaultAnnotationReflect.class) ; // 得到指定的Annotation

String key = mda.key() ; // 取出设置的key

String value = mda.value() ; // 取出设置的value

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

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

}

}

};

总结：

Annotation在实际开发中，不管如何使用，其最终肯定是结合反射机制，也就是说可以通过Annotation设置一些内容到一个方法上去，以完成特定的功能。

## 2. 深入Annotation

目标：

@Target注释的作用

@Documented注释的作用

@Inherited注释的作用

### **@Target注释**

之前定义的Annotation，如果没有哦明确的说明可以在任意的位置上使用。如果现在需要制定其使用的范围的话，则就必须使用@Target注释

[@Documented](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\annotation\Documented.html)

[@Retention](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\annotation\Retention.html)([value](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\annotation\Retention.html#value--)=[RUNTIME](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\annotation\RetentionPolicy.html#RUNTIME))

[@Target](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\annotation\Target.html)([value](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\annotation\Target.html#value--)=[ANNOTATION\_TYPE](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\annotation\ElementType.html#ANNOTATION_TYPE))

public @interface Target

public static final [ElementType](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\annotation\ElementType.html) ANNOTATION\_TYPE

[ElementType](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\annotation\ElementType.html)是枚举类型，其取值范围：

public static final [ElementType](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\annotation\ElementType.html) ANNOTATION\_TYPE 只能用在注释上声明

public static final [ElementType](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\annotation\ElementType.html) CONSTRUCTOR 只能用再构造方法上声明

public static final [ElementType](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\annotation\ElementType.html) FIELD 只能用在字段声明(包括枚举常量)上

public static final [ElementType](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\annotation\ElementType.html) LOCAL\_VARIABLE 只能用在局部变量上声明

public static final [ElementType](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\annotation\ElementType.html) METHOD 只能用在方法上声明

public static final [ElementType](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\annotation\ElementType.html) PACKAGE 只能用在包上声明

public static final [ElementType](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\annotation\ElementType.html) PARAMETER 只能用再参数上声明

public static final [ElementType](file:///D:\jdk-8u101-docs-all\docs\api\java\lang\annotation\ElementType.html) TYPE 只能用在类、接口、枚举类型上

现在定义一个Annotation，只能在类上使用，类型是TYPE

**MyTargetAnntation.java**

import java.lang.annotation.Target ;

import java.lang.annotation.ElementType ;

import java.lang.annotation.Retention ;

import java.lang.annotation.RetentionPolicy ;

@Target({ElementType.TYPE,ElementType.METHOD}) // 此注释只能用在类和方法上使用

@Retention(value=RetentionPolicy.RUNTIME)

public @interface MyTargetAnntation{

public String key() default "LXH" ;

public String value() default "李兴华" ;

}

**SimpleBean.java**

@MyTargetAnntation(key="MLDN",value="www.mldnjava.cn")

public class SimpleBean{

@MyTargetAnntation(key="MLDN",value="www.mldnjava.cn")

public String toString(){

return "Hello LiXingHua!!!" ;

}

};

### @Documented注释

@Documented，可以用在任何的Annotation上使用，所有的Annotation默认情况下都是使用@Documented进行注释的，而且生成javadoc的时候客户通过@Documented设置一些说明信息。

**MyDocumentedAnntation.java**

import java.lang.annotation.Documented ;

@Documented

public @interface MyDocumentedAnntation{

public String key() default "LXH" ;

public String value() default "李兴华" ;

}

**SimpleBeanDocumented.java**

@MyDocumentedAnntation(key="MLDN",value="www.mldnjava.cn")

public class SimpleBeanDocumented{

/\*\*

\* 此方法在对象输出时调用，返回对象信息

\*/

@MyDocumentedAnntation(key="MLDN",value="www.mldnjava.cn")

public String toString(){

return "Hello LiXingHua!!!" ;

}

};

之后，通过javadoc命令，生成java doc文档

javadoc -d doc SimpleBeanDocumented.java

在doc目录下生成文档，进去后发现很多html文档，可以打开index.html查看。

### @Inherited注释

此注释表示一个Annotation是否可以被继承下来。

**MyInheritedAnnotation.java**

package org.lxh.demo16.inheriteddemo ;

import java.lang.annotation.Retention ;

import java.lang.annotation.RetentionPolicy ;

import java.lang.annotation.Documented ;

import java.lang.annotation.Inherited ;

@Documented

@Inherited

@Retention(value=RetentionPolicy.RUNTIME)

public @interface MyInheritedAnnotation{

public String name() ;

}

**Person.java**

package org.lxh.demo16.inheriteddemo ;

@MyInheritedAnnotation(name="李兴华")

public class Person{

};

**Student.java**

package org.lxh.demo16.inheriteddemo ;

public class Student extends Person{

};

**ReflectInheritedDemo.java**

import java.lang.annotation.Annotation ;

import org.lxh.demo16.inheriteddemo.MyInheritedAnnotation ;

public class ReflectInheritedDemo{

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

Class<?> c = null ;

c = Class.forName("org.lxh.demo16.inheriteddemo.Student") ;

Annotation ann[] = c.getAnnotations() ; // 取得全部的Annotation

for(Annotation a:ann){ // 输出

System.out.println(a) ;

}

// 继续取得此Annotation设置的内容

if(c.isAnnotationPresent(MyInheritedAnnotation.class)){

MyInheritedAnnotation mda = null ;

mda = c.getAnnotation(MyInheritedAnnotation.class) ;

String name = mda.name() ; // 取出name的内容

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

}

}

}

# 2016年8月18日21:43:17

## 1. java数据库编程

JDBC

JDBC 提供了一种与平台无关的用于执行SQL语句的标准javaAPI,可以方便的实现多种关系型数据库的统一操作，它由一组用java语言编写的类和接口组成。

在实际开发中可以直接使用JDBC进行各个数据库的连接与操作，而且可以方便的向数据库中发送各种SQL命令。在JDBC中提供的是一套标准的接口，这样，各个支持java的数据库生产商只要按照此接口提供的实现，则就都可以使用JDBC进行操作。极大的体现了java的可移植性的思想。

在数据库编程中一切都是以SQL语句为操作的标准的，所以只要更好的掌握了SQL语法，则可以更加方便的开发各种用户所需的程序。

在JDBC中所有的类和接口都保存在java.sql包中，在此包中规定了大量的接口。

实际上JDBC本身是一操作的标准，那么所有的数据生产商，只要是想支持JAVA,则肯定要符合JDBC规范。

JDBC驱动分类

JDBC本身提供的是一套数据库操作标准，而这些标准又需要各个数据库厂商实现，所以针对每一个数据厂商都会提供一个JDBC的驱动程序，目前比较常见的JDBC驱动程序可以分为以下四类：

JDBC-ODBC桥驱动

JDBC本地驱动

JDBC网络驱动

本地协议纯JDBC驱动

但是一般在开发中会将以上的四类划分成以下的三类：

JDBC-ODBC

纯JDBC连接

网络的JDBC连接

### 1.1 JDBC-ODBC

JDBC-ODBC是SUN公司提供的一个标准的JDBC操作，直接利用微软的ODBC进行数据库连接操作，但是，这样的操作性能较低，所以通常情况下是不推荐使用这种方式进行操作的。

### 1.2 JDBC本地驱动

直接使用各个数据库生产商提供的JDBC驱动程序，但是因为其只能应用在特定的数据库上，会丧失掉程序的可移植性，但是这样的操作性能较高。

JDBC本身提供的是一组类和接口，那么各个数据库生产商要实现这些接口，所以这些驱动程序都是需要单独配置的，往往是以一组jar包的形式出现的，如果要使用的话，则肯定要配置classpath。

在开发中大部分情况都基于一种数据库的开发，所以一般使用此种模式是最多的。

### 1.3 JDBC网络驱动

这种驱动程序将JDBC转换为与DBMS无关的网络协议，之后这种协议又被某个服务器转换为一种DBMS协议。这种网络服务器中间件能够将它的纯java客户机连接到多种不同的数据库上。所用的具体协议取决于提供者。通常，这是最为灵活的JDBC驱动程序。

### 1.4 本地协议纯JDBC驱动

这种类型的驱动程序将JDBC调用直接转换为DBMS所使用的网络协议。这将允许从客户机器上直接调用DBMS服务器，是Internet访问的一个很实用的解决方法。

数据库的操作过程：

打开数据库的服务

打开数据库，连接的时候一般都要输入用户名或密码

操作数据库：创建表、查询表、更新记录

关闭

DriverManager： 是一个最常用的类，使用此类可以取得一数据库的连接。

Connection：每一个Connection的实例化对象都表示一个数据库连接

数据库的操作：Statement、PreparedStatement

数据库的查询：ResultSet

调用的存储过程：CallableStatement

2016年8月19日16:04:49

1.变态题

Integer i1 = 100;

Integer i2 = 100;

boolean b1 = i1 == i2;//结果为true

Integer i3 = 200;

Integer i4 = 200;

boolean b2 = i3 == i4;//结果为false

Integer i1 = Integer.valueOf(100);

Integer i2 = Integer.valueOf(100);

boolean b1 = i1 == i2;//结果为true

Integer i3 = Integer.valueOf(200);

Integer i4 = Integer.valueOf(200);

boolean b2 = i3 == i4;//结果为false

这时你应该可以看到，疑团在Integer.valueOf()方法身上。传递给这个方法100时，它返回的Integer对象是同一个对象，而传递给这个方法200时，返回的却是不同的对象。这是我们需要打开Integer的源码（这里就不粘贴Integer的源代码了），查看它的valueOf()方法内容。

Integer类的内部缓存了-128~127之间的256个Integer对象，如果valueOf()方法需要把这个范围之内的整数转换成Integer对象时，valueOf()方法不会去new对象，而是从缓存中直接获取，这就会导致valueOf(100)两次，都是从缓存中获取的同一个Integer对象！如果valueOf()方法收到的参数不在缓存范围之内，那么valueOf()方法会new一个新对象！这就是为什么Integer.valueOf(200)两次返回的对象不同的原因了。

# 2016年8月19日18:42:53

1. MySQL数据库

MySQL是一个小型的关系型数据库管理系统。

mysql安装好后要

mysqld --initialize-insecure --user=mysql

登录命令：mysql -u root –p

**其中登录用户名是：root ， 密码为空**

mysql的默认安装后是没有可视化管理界面的，不像sql server有个管理界面，mysql可以直接在命令行用命令来执行sql语句。

也可以使用mysql官方提供的 mysql workbench软件来可视化管理。

## 2. JDBC

JDBC即java数据库连接，提供了一种与平台无关的用于执行sql语句的标准javaAPI,可以方便的实现多种关系型数据库的统一操作，它由一组用java语言编写的类和接口组成。JDBC不仅可以连接mysql，也可以练级sql server。

JDBC中所有的类和接口都保存在java.sql包中，在此包中定义了大量的接口。

JDBC只是一个操作标准，那么所有的数据库生产商，要想支持java，则肯定要符合JDBC规范。所有每一个数据库厂商都会提供一个JDBC的驱动程序。

JDBC 操作步骤：

1.加载数据库驱动程序：各个数据库都会提供JDBC的驱动程序开发包，直接把JDBC操作所需的开发包配置到classpath路径即可。

2. 连接数据库，数据库不同，连接地址也不同，一般都要输入用户名和密码

3. 使用语句执行数据库操作

4.关闭数据库连接，关闭以释放资源。

创建数据库

create database 数据库名称

删除数据库

drop database 数据库名称

mysql 的基本操作

DROP TABLE user ; -- 删除表

CREATE TABLE user(

id INT AUTO\_INCREMENT PRIMARY KEY ,

name VARCHAR(30) NOT NULL ,

password VARCHAR(32) NOT NULL ,

age INT NOT NULL ,

sex VARCHAR(2) DEFAULT '男' ,

birthday DATE

) ;

INSERT INTO user(name,password,age,sex,birthday) VALUES ('李兴华','www.mldn.cn',30,'男','2008-08-27') ;

INSERT INTO user(name,password,age,sex,birthday) VALUES ('李康','www.mldnjava.cn',23,'女','2005-08-27') ;

DELETE FROM user WHERE id=3 ;

UPDATE user SET name='MLDN',age=5,birthday='2000-08-27' WHERE id=4 ;

SELECT name,birthday FROM user ;

SELECT \* FROM user WHERE name LIKE '%m%' OR password LIKE '%m%' ;

SELECT \* FROM user WHERE name LIKE '%m%' OR password LIKE '%m%' LIMIT 10,5 ;

LIMIT 10,5 是部分查询，从第10条开始查询共5条数据。

Mysql 的connector的下载地址：https://dev.mysql.com/downloads/connector/

### 要想用java连接mysql，需要使用mysql提供的连接工具。选择[Connector/J](https://dev.mysql.com/downloads/connector/j/)

# 2016年9月3日20:25:14

## 1.连接数据库

使用JDBC连接mysql需要mysql提供的驱动程序，也就是jar包: **mysql-connector-java-5.1.30-bin.jar**

这个包可以从mysql官网下载。

然后是配置classpath环境变量，在系统环境变量的classpath里面添加这个包的全路径即可。

注意：如果在命令行里面输入javac编译java代码没问题，但是输入java 类名 却提示无法加载主类，原因是系统环境变量的classpath没有设置对。按照网上的JDK环境变量配置方法来修改。

**PATH=.;%JAVA\_HOME%\bin**  
**CLASSPATH=.;%JAVA\_HOME%\lib\dt.jar;%JAVA\_HOME%\lib\toos.jar;**  
**JAVA\_HOME=C:\Program Files\Java\jdk1.8.0\_73**

mysql-connector-java-5.1.44-bin.jar的路径也要加载到classpath的后面。

加载驱动，使用反射：

public static final String DBDRIVER ="org.gjt.mm.mysql.Driver";

try{

Class.forName(DBDRIVER);

}catch(ClassNotFoundException e)

{

e.printStackTrace();

}

"org.gjt.mm.mysql.Driver是根据jar包里面的class文件的路径来决定的。

连接数据库使用DriverManager类进行操作，使用其中的getConnection方法连接。

getConnection可以直接数据连接地址，也可以在输入地址的同时输入用户名和密码。

连接地址的格式：

jdbc:mysql://IP地址:端口号/数据库名称

如：jdbc:mysql://localhost:3306/firstdatabase

其中ip地址前的格式是固定的。

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;

public class ConnectionDemo

{

public static final String DBDRIVER ="org.gjt.mm.mysql.Driver";

public static final String DBURL="jdbc:mysql://localhost:3306/firstdatabase";

public static final String DBUSER="root";

public static final String DBPASS ="";

public static void main(String[] agrs)

{

Connection conn=null;

try{

Class.forName(DBDRIVER);

}catch(ClassNotFoundException e)

{

e.printStackTrace();

}

try{

conn= DriverManager.getConnection(DBURL,DBUSER,DBPASS);

}

catch(SQLException e)

{

e.printStackTrace();

}

try{

conn.close();

}catch(SQLException e)

{

e.printStackTrace();

}

}

}

## 2.数据库更新操作

使用Connection对象取得Statement实例

使用Statement进行数据增加、修改、删除操作

### 2.1 satement接口

如果要对数据库进行操作，则肯定要使用Statement接口完成，此接口可以使用Conection接口中提供的createStatement()方法实例化。

需要导入包：**import java.sql.Statement;**

**executeUpdate(Stirng sql)：** 执行数据库更新操作，例如：INSERT 、UPDATE、DELETE等

executeQuery(String sql): 执行数据库查询操作，返回一个结果对象，ResultSet

addBatch(String sql) 增加一个执行的sql语句

int[] executeBatch() 批量执行sql语句

close() 关闭statement操作

execute(String sql) 执行sql语句

**插入数据**：

使用Statement接口下的executeUpdate(String sql)方法来执行插入的sql语句，

String sql ="insert into user(name,password,age,birthday) values('guozhang3','123456','25','1991-09-19')";

Statement stmt;

Class.forName(DBDRIVER);

conn= DriverManager.getConnection(DBURL,DBUSER,DBPASS);

stmt = conn.createStatement();

stmt.executeUpdate(sql);

stmt.close();

conn.close();

要先关闭statement，然后再关闭connection

其中name字段的类型是VARCHAR(30)，不能输入汉字。 sql 字符串是单引号。

**更新数据：**

String sql2 ="update user set name='zhangsan',password='789',age=88,birthday='2001-01-14' where id =3";

删除数据：

String sql3 ="DELETE FROM USER WHERE id=5";

删除后其他行的id不会变化。

## 3. ResultSet接口

主要用于查询操作

import java.sql.ResultSet;

Statement stmt;

ResultSet rs=null;

Class.forName(DBDRIVER);

conn= DriverManager.getConnection(DBURL,DBUSER,DBPASS);

stmt = conn.createStatement();

rs = stmt.executeQuery(sql4);

while(rs.next())

{

int id = rs.getInt("id");

String name=rs.getString("name");

String password = rs.getString("password");

int age = rs.getInt("age");

System.out.println(id+" "+name+" "+password+" "+age);

}

上面红色的操作者是取得某个字段，如果列名称太长，肯定不方便，所以可以采用顺序的编号获取。

int id = rs.getInt(1);

String name = rs.getString(2);

String password = rs.getString(3);

int age = rs.getInt(4);

## 4. PreparedStatement接口

PreparedStatement是Statement的子接口，属于预处理操作，PreparedStatement在操作时，是先在数据表之中准备好一条SQL语句，但是此sql语句的具体内容暂时不设置，而是之后再设置，以插入数据为例，执行sql语句后，插入的数据是空的，需要之后设置。

注意日期的输入，PreparedStatement里面的日期是java.sql.date类型，正常情况下是java.util.date类型，需要转换。

String birthday = "2007-08-27" ; // 生日

java.util.Date temp = null ;

temp = new SimpleDateFormat("yyyy-MM-dd").parse(birthday) ;

java.sql.Date bir = new java.sql.Date(temp.getTime()) ;

import java.sql.Connection ;

import java.sql.DriverManager ;

import java.sql.SQLException ;

import java.sql.PreparedStatement ;

import java.text.SimpleDateFormat ;

public class PreparedStatementDemo01 {

// 定义MySQL的数据库驱动程序

public static final String DBDRIVER = "org.gjt.mm.mysql.Driver" ;

// 定义MySQL数据库的连接地址

public static final String DBURL = "jdbc:mysql://localhost:3306/mldn" ;

// MySQL数据库的连接用户名

public static final String DBUSER = "root" ;

// MySQL数据库的连接密码

public static final String DBPASS = "mysqladmin" ;

public static void main(String args[]) throws Exception{ // 所有异常抛出

Connection conn = null ; // 数据库连接

PreparedStatement pstmt = null ; // 数据库操作

String name = "李兴华" ; // 姓名

String password = "www.mldnjava.cn" ; // 密码

int age = 30 ; // 年龄

String sex = "男" ; // 性别

String birthday = "2007-08-27" ; // 生日

java.util.Date temp = null ;

temp = new SimpleDateFormat("yyyy-MM-dd").parse(birthday) ;

java.sql.Date bir = new java.sql.Date(temp.getTime()) ;

String sql = "INSERT INTO user(name,password,age,sex,birthday) VALUES (?,?,?,?,?) " ;

Class.forName(DBDRIVER) ; // 加载驱动程序

conn = DriverManager.getConnection(DBURL,DBUSER,DBPASS) ;

pstmt = conn.prepareStatement(sql) ; // 实例化PreapredStatement对象

pstmt.setString(1,name) ;

pstmt.setString(2,password) ;

pstmt.setInt(3,age) ;

pstmt.setString(4,sex) ;

pstmt.setDate(5,bir) ;

int t = pstmt.executeUpdate() ; // 执行更新

System.out.println(t);

pstmt.close() ;

conn.close() ; // 数据库关闭

}

};

2017年10月10日09:22:28

1. static 修饰父方法，不能调用非static修饰的属性，因为static修饰的方法或属性先于对象生成，而非static修饰的方法或属性只属于对象。

非static修饰的方法可以调用static修饰的属性。

1. java的main方法都是static方法，不能直接调用非static方法。但是可以使用对象的调用非static方法。

public class Test2

{

public static void main(String args[])

{

new Test2().info();

}

public void info()

{

System.out.println("hello");

}

}

2017年10月12日09:11:06

1. 使用instanceof 判断是不是整型数组

public class Test6

{

public static void main(String[] args)

{

int temp[]={1,3,5,7,9};

Object obj =temp;

print(obj);

}

public static void print(Object o)

{

**if(o instanceof int[])**

{

int x[]=(int [])o;

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

{

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

}

}

}

}

1. **包装类**

一切皆对象，八种数据类型不是对象。

把int包装成一个类，这样就可以以对象的形式操作int类型了。

byte -- Byte

int -- Integer

long -- Long

float -- Float

double -- Double

char -- Character

boolean -- Boolean

装箱与拆箱

将基本数据类型变为包装类型称为装箱

将包装类的类型变为基本数据类型称为拆箱

public class Test7

{

public static void main(String[] args)

{

int x =30; //基本数据类型

Integer i = new Integer(x); //装箱,将基本数据类型变为包装类

int temp =i.intValue(); //拆箱：将一个包装类型变为基本数据类型

System.out.println(temp);

}

}

float f = 40.4f;

Float x = new Float(f);

float temp = x.floatValue();

System.out.println(temp);

在jdk1.5之前包装类是不能直接进行加减乘除操作的，因为是一个类，所以不可以这样。但是在之后的版本，增加了自动装箱和自动拆箱的功能，而且也可以使用包装类直接进行数字运算。

Integer i =30; //自动装箱成Integer

Float f = 30.3f; //自动装箱成Float

int x= i; //自动拆箱成int

float y= f; //自动拆箱成float

包装类的应用： 包装类在实际应用中用得最多的还在于字符串变为基本数据类型的操作，如：将一个全由数字组成的字符串变为一个int或float类型。

在Integer中 使用 parseInt（String s）

在Floath中使用 parseFloat（String s）

String str1= "30";

String str2= "30.3f";

int x = Integer.parseInt(str1);

float y = Float.parseFloat(str2);

以上操作字符串必须由数字组成

2017年10月12日14:54:32

1. **匿名内部类**

匿名内部类是在抽象类和接口的基础上发展起来的

interface A

{

public void printInfo();

}

class X

{

public void fun1()

{

this.fun2**(new A(){**

public void printInfo()

{

System.out.println("hello world");

}

});

}

public void fun2(A a)

{

a.printInfo();

}

}

public class Test10

{

public static void main(String[] args)

{

new X().fun1();

}

}

1. 异常

e.printStackTrace();

2.1 throw和throws关键字

在定义一个方法的时候开业使用throws关键字声明，使用throws声明的方法表示此方法不处理异常，而交给方法的调用者进行处理。

使用格式： public 返回值类型 方法名称（参数） throws 异常类{}

class Math

{

public int div(int i,int j)throws Exception

{

int temp=i/j;

return temp;

}

}

public class Test11

{

public static void main(String[] args)

{

Math m = new Math();

try{

System.out.println("除法操作："+m.div(10,0));

}

catch(Exception e)

{

e.printStackTrace();

}

}

}

如果在主方法的声明上也使用了throws关键字呢，意味着主方法可以不处理异常，表示一切的异常交给JVM进行处理。

class Math

{

public int div(int i,int j)throws Exception

{

int temp=i/j;

return temp;

}

}

public class Test11

{

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

{

Math m = new Math();

/\* try{\*/

System.out.println("除法操作："+m.div(10,0));

/\*

}

catch(Exception e)

{

e.printStackTrace();

}\*/

}

}

throw关键字的作用是在程序中抛出一个异常，抛出的时候抛出的是一个异常类的实例化对象。

public class Test12

{

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

{

try{

throw new Exception("自己抛着玩的");

}

catch(Exception e)

{

System.out.println(e);

}

}

}

2018年1月20日11:38:39

1. 关于java的接口能否实例化的问题

在Android中经常看到new一个接口的情况。实际上接口是不能实例化的。但是这里重写了接口中的方法，这样也是可以的。

person.java

public interface Person

{

public void say();

}

Chinese.java

public class Chinese{

public void sayhello(Person p)

{

p.say();

}

Test.java

public class Test{

public static void main(String[] args)

{

Person p = new Person(){

@Override

public void say()

{

System.out.println("hello world");

}

};

Chinese ch = new Chinese();

ch.sayhello(p);

}

}

2018年3月5日19:42:35

1. Object类提供了如下几个常用方法：
2. [clone](file:///C:\Users\Administrator\Desktop\api\java\lang\Object.html#clone())()   
             创建并返回此对象的一个副本。
3. [equals](file:///C:\Users\Administrator\Desktop\api\java\lang\Object.html#equals(java.lang.Object))([Object](file:///C:\Users\Administrator\Desktop\api\java\lang\Object.html) obj)   
             指示其他某个对象是否与此对象“相等”。
4. [finalize](file:///C:\Users\Administrator\Desktop\api\java\lang\Object.html#finalize())()   
             当垃圾回收器确定不存在对该对象的更多引用时，由对象的垃圾回收器调用此方法。
5. [getClass](file:///C:\Users\Administrator\Desktop\api\java\lang\Object.html#getClass())()   
             返回此 Object 的运行时类。
6. [hashCode](file:///C:\Users\Administrator\Desktop\api\java\lang\Object.html#hashCode())()   
             返回该对象的哈希码值。
7. [notify](file:///C:\Users\Administrator\Desktop\api\java\lang\Object.html#notify())()   
             唤醒在此对象监视器上等待的单个线程。
8. [notifyAll](file:///C:\Users\Administrator\Desktop\api\java\lang\Object.html#notifyAll())()   
             唤醒在此对象监视器上等待的所有线程。
9. [toString](file:///C:\Users\Administrator\Desktop\api\java\lang\Object.html#toString())()   
             返回该对象的字符串表示。
10. [wait](file:///C:\Users\Administrator\Desktop\api\java\lang\Object.html#wait())()   
              在其他线程调用此对象的 [notify()](file:///C:\Users\Administrator\Desktop\api\java\lang\Object.html#notify()) 方法或 [notifyAll()](file:///C:\Users\Administrator\Desktop\api\java\lang\Object.html#notifyAll()) 方法前，导致当前线程等待。
11. [wait](file:///C:\Users\Administrator\Desktop\api\java\lang\Object.html#wait(long))(long timeout)   
              在其他线程调用此对象的 [notify()](file:///C:\Users\Administrator\Desktop\api\java\lang\Object.html#notify()) 方法或 [notifyAll()](file:///C:\Users\Administrator\Desktop\api\java\lang\Object.html#notifyAll()) 方法，或者超过指定的时间量前，导致当前线程等待。
12. [wait](file:///C:\Users\Administrator\Desktop\api\java\lang\Object.html#wait(long, int))(long timeout, int nanos)   
              在其他线程调用此对象的 [notify()](file:///C:\Users\Administrator\Desktop\api\java\lang\Object.html#notify()) 方法或 [notifyAll()](file:///C:\Users\Administrator\Desktop\api\java\lang\Object.html#notifyAll()) 方法，或者其他某个线程中断当前线程，或者已超过某个实际时间量前，导致当前线程等待。
13. HashCode

然后讲下什么是HashCode，总结几个关键点：

1、HashCode的存在主要是为了查找的快捷性，HashCode是用来在散列存储结构中确定对象的存储地址的

2、如果两个对象equals相等，那么这两个对象的HashCode一定也相同

3、如果对象的equals方法被重写，那么对象的HashCode方法也尽量重写

4、如果两个对象的HashCode相同，不代表两个对象就相同，只能说明这两个对象在散列存储结构中，存放于同一个位置

**如果两个类的HashCode都不相同，那么这两个类必定是不同的**。

关于hashCode方法，一致的约定是：

重写了euqls方法的对象必须同时重写hashCode()方法。

如果2个对象通过equals调用后返回是true，那么这个2个对象的hashCode方法也必须返回同样的int型散列码  
  
如果2个对象通过equals返回false，他们的hashCode返回的值**允许相同**。

2018年4月4日20:20:25

1. 正则表达式
2. 判断一字符串是否包含子字符串

import java.util.regex.\*;

public class test

{

public static void main(String[] args)

{

String content = "I am noob " +

"from runoob.com.";

String pattern = ".\*runoob.\*";

boolean isMatch = Pattern.matches(pattern, content);

System.out.println("字符串中是否包含了 'runoob' 子字符串? " + isMatch);

}

}

**验证email是否正确：**

public static void main(String[] args) {

    // 要验证的字符串

    String str = "service@xsoftlab.net";

    // 邮箱验证规则

    String regEx = "[a-zA-Z\_]{1,}[0-9]{0,}@(([a-zA-z0-9]-\*){1,}\\.){1,3}[a-zA-z\\-]{1,}";

    // 编译正则表达式

    Pattern pattern = Pattern.compile(regEx);

    // 忽略大小写的写法

    // Pattern pat = Pattern.compile(regEx, Pattern.CASE\_INSENSITIVE);

    Matcher matcher = pattern.matcher(str);

    // 字符串是否与正则表达式相匹配

    boolean rs = matcher.matches();

    System.out.println(rs);

}

正则表达式的组成：

**元字符**：具有特殊含义的符号，如：\d代表一个数字

**普通字符**：普通预言中的单词

**匹配**：字符串与正则表达式是否等价

现在是北京时间：10:20 == 现在是北京时间：\d\d:\d\d

　在regex包中，包括了两个类，Pattern(模式类)和Matcher(匹配器类)。Pattern类是用来表达和陈述所要搜索模式的对象，Matcher类是真正影响搜索的对象。另加一个新的例外类，PatternSyntaxException，当遇到不合法的搜索模式时，会抛出例外。

首先让我们构成一个正则表达式。为简单起见，先构成一个正则表达式来识别下面格式的电话号码数字:(nnn)nnn-nnnn。

　　第一步，创建一个pattern对象来匹配上面的子字符串。一旦程序运行后，如果需要的话，可以让这个对象一般化。匹配上面格式的正则表达可以这样构成:(/d{3})/s/d{3}-/d{4}，其中/d单字符类型用来匹配从0到9的任何数字，另外{3}重复符号，是个简便的记号，用来表示有3个连续的数字位，也等效于(/d/d/d)。/s也另外一个比较有用的单字符类型，用来匹配空格，比如Space键，tab键和换行符。

是不是很简单?但是，如果把这个正则表达式的模式用在java程序中，还要做两件事。对java的解释器来说，在反斜线字符(/)前的字符有特殊的含义。在java中，与regex有关的包，并不都能理解和识别反斜线字符(/)，尽管可以试试看。但为避免这一点，即为了让反斜线字符(/)在模式对象中被完全地传递，应该用双反斜线字符(/)。此外圆括号在正则表达中两层含义，如果想让它解释为字面上意思(即圆括号)，也需要在它前面用双反斜线字符(/)。

1.1：正则表达式中反斜杠的作用：

在其他语言中，\\表示“我想要在正则表达式中插入一个普通的（字面上的）反斜线，请不要给它任何特殊的意义。  
”而在Java中，\\ 的意思是“我要插入一个正则表达武的反斜线，所以其后的字符具有特殊的意义。”

(1) 例如，如果你想表示一位数字，那么正则表达式应该是\\d。如果你想插入一个普通的反斜线，则应该这样\\\\。

(2) 要表示 “一个或多个之前的表达式“，应该使用 + ,  所以，如果要表示“可能有一个负号，后面跟着一位或多位数字”，可以这样：-? \\ d+

(3)正则表达式中，括号有着将表达式分组的效果，而竖直线 | 则表示或操作。

(-|\\+)?

这个正则表达式表示字符串的起始字符可能是一个  -或 +，或二者皆没有（因为后面跟着？修饰符）。

因为字符 +  在正则表达式中有特殊的意义，所以必须使用\\ 其转义，使之成为表达式中的一个普通字符。

2018年4月8日22:21:25

1. java的静态类，用static修饰类

这种情况一般用在修饰内部内。

如果一个内部类使用static关键字声明，则此内部类就称为外部类，可以直接通过外部类.内部类进行访问。

2018年10月9日21:11:07

# ORM

[编辑](javascript:;)

1. 对象关系映射（英语：(Object Relational Mapping，简称ORM，或O/RM，或O/R mapping），是一种程序技术，用于实现面向对象编程语言里不同类型系统的数据之间的转换 [1]  。从效果上说，它其实是创建了一个可在编程语言里使用的--“虚拟对象数据库”。

ORM框架的主要功能就是根据映射配置文件，完成数据在对象模型与关系模型之间的映射，同时也屏蔽了重复的代码，只暴露简单的API供开发人员使用。

数据库资源比较珍贵，不能直接简单粗暴地访问数据库，关闭数据库。应用程序一般需要通过集成缓存、数据源、数据库连接池等组件进行优化。

很多ORM框架都提供了集成第三方缓存、第三方数据源等组件的接口。

1. DAO 数据访问对象Data Access Object
2. 常见持久化框架(ORM):

Hibernate、JPA、Spring JDBC、MyBatis