JavaSE

# jre和jdk区别

1. jdk (Java development kit)是面向开发人员使用的SDK(software development kit软件开发包，包括函数库，编译程序等)，他提供了Java的开发环境和运行环境
2. jre(java runtime environment)是Java的运行环境，是面向Java程序的使用者，而不是开发者

# 基础知识

1. 记事本打开，notepad
2. javac -verbose 显示详细的编译过程
3. 在一个文件中允许多个class 存在，但是必须只能有一个public class ，每个class 里都可以有main 方法
4. 标识符 以字母、下划线、美元符号开始，后可跟数字
5. 基本数据类型：四类八种
6. 整型，浮点型，字符串型，布尔型
7. byte,short,int,long,float,double,char,boolean
8. byte 8位 short 16位 int 32位 long64 位 float 32位 double 64位
9. 直接量中如果是整数，默认是int型 如果是小数，默认为double 型
10. 如果将double型给float 型时，必须显示的写出：float = 0.24f
11. long a = 10000000000L; 10000000000首先会分配int型的内存空间，需显示的写上L
12. Unicode代码点采用十六进制表示
13. int i = 20013 System.out.println((char)i ); 向下强制转型
14. 也可以写成:System.out.println(‘\u4e2d’);\u表示所写的是unicode的代码点
15. 多种数据类型混合运算时，按照容量最大的数据类型作为结果存储的类型
16. byte char short 运算时提升为int
17. break 跳出循环 ，while签套时break跳出当前的while循环，continue 跳出本次循环，忽略continue后面的语句
18. do-while 语句中while 后面加分号 即：do {}while();
19. 扫描控制台输入

Java.util.Scanner sc = new java.util.Scanner(System.in)

System.out.println(sc.nextInt())

System.exit(0) 退出

1. 包的命名规则

首先为了确保其唯一性，减少重复，一般以公司的域名倒置作为包的前缀，以项目简称为一级org.leadfar.oa，按照功能划分org.leadfar.oa.system，按照程序的架构划分org.leadfar.oa.view ca .dao

1. 源码文件编译

javac–d . ATM.java 编译时在当前目录下创建对应的包(文件中有打包语句)

设置类路径classpath:set classpath=路径 ，仅在当前的黑窗口有效，在任何磁盘都能运行路径下的Java文件，注意，假如包在F：被复制到了E：，E：下的Java文件不能被执行，解决方法，在E：下　set classpath=e:\;.;

设置临时类路径Java –classpath 路径名 （有空格）

在工作目录执行其他盘的Java文件：

F：\>java –Duser.dir=d:\ftp\project\Test.java

Java的扩展目录：jar cvf test.jar ./org test.jar为包的名字，./org 为被打包的目录，讲jar文件拷贝到JDK\jre\lib\ext目录下

1. 类加载器

类加载器：启动类加载器，扩展类加载器，类路径类加载器

# 面向对象的三大特征

1. 封装

封装性就是尽可能的隐藏对象内部细节，对外形成一道边界，只保留有限的接口和方法与外界进行交互。封装的原则是使对象以外的部分不能随意的访问和操作对象的内部属性，从而避免了外界对对象内部属性的破坏。

可以通过对类的成员设置一定的访问权限，实现类中成员的信息隐藏。

private：类中限定为private的成员，只能被这个类本身访问。如果一个类的构造方法声明为private,则其它类不能生成该类的一个实例。

default：类中不加任何访问权限限定的成员属于缺省的（default）访问状态，可以被这个类本身和同一个包中的类所访问。

protected：类中限定为protected的成员，可以被这个类本身、它的子类（包括同一个包中以及不同包中的子类）和同一个包中的所有其他的类访问。

public：类中限定为public的成员，可以被所有的类访问。

封装最好理解了。封装是面向对象的特征之一，是对象和类概念的主要特性。

1. 继承

面向对象编程 (OOP) 语言的一个主要功能就是“继承”。继承是指这样一种能力：它可以使用现有类的所有功能，并在无需重新编写原来的类的情况下对这些功能进行扩展。

通过继承创建的新类称为“子类”或“派生类”。

被继承的类称为“基类”、“父类”或“超类”。

继承的过程，就是从一般到特殊的过程。

要实现继承，可以通过“继承”（Inheritance）和“组合”（Composition）来实现。

1. 多态

对象的多态性是指在父类中定义的属性或方法被子类继承之后，可以具有不同的数据类型或表现出不同的行为。这使得同一个属性或方法在父类及其各个子类中具有不同的语义。例如："几何图形"的"绘图"方法，"椭圆"和"多边形"都是"几何图"的子类，其"绘图"方法功能不同。

Java的多态性体现在两个方面：由方法重载实现的静态多态性（编译时多态）和方法重写实现的动态多态性（运行时多态）。

编译时多态：在编译阶段，具体调用哪个被重载的方法，编译器会根据参数的不同来静态确定调用相应的方法。

运行时多态：由于子类继承了父类所有的属性（私有的除外），所以子类对象可以作为父类对象使用。程序中凡是使用父类对象的地方，都可以用子类对象来代替。一个对象可以通过引用子类的实例来调用子类的方法。

# Super的用法

第一种：super.xxx(xxx为变量名或对象名)，获取父类中xxx变量或者方法引用，使用这种方法可以直接访问父类的变量或对象，进行操作

第二种：super.xxx()(xxx为方法名)，直接调用父类中的方法

第三种：super();调用父类的构造方法，出现在构造方法中的第一句

# field、 attribute、 property三者区别

field和attribute是同一概念 例如：

public class Person{

private int \_id;

} //此处的\_id 叫做field 或 attribute

public int getId(){} //此处的Id叫做property，就是由getters/setters方法来确定的

public void setId(int id){}//set get 里可以加入一些逻辑保护attribute被赋予非法值

Boolean 类型的比较特殊不是set get 命名，is开头

# 面向对象内存分析

栈（stack）：存放局部变量，一个方法开辟一个栈帧

堆（heap）：存放引用类型的对象（存放new出来的东西，包括对象的属性）

方法区（method area）：存放字符串常量和静态变量，字节码指令

PC寄存器：计数器，存放执行到哪一条指令了

如：public static void main (String args[]){

Student s = new Student();

s.setName(“张建国”);

s.setSex(“男”)；

}

此时，形参args 、s为局部变量，被放在栈中；new Student()被放在堆中，new出来的对象的属性也放在堆内存中，setName()方法在栈中开辟栈帧，用完后直接出栈，“张建国”这类字符串常量放在方法区。（当调用一个方法时，虚拟机会在当前线程对应的栈中压入一个栈帧，栈帧就是一块专门用来存放方法执行过程的数据的内存空间，栈帧中会存放方法的实参，方法内部定义的局部变量等内容，当方法调用返回之后，虚拟机弹出其栈帧，并抛弃）

最后由垃圾回收机制回收

# 实例变量、类变量的区别

实例变量：产生对象时产生，与对象相关，在堆中

类变量（静态变量）static修饰的成员变量，在方法区中，在内存中只有一份，与类同时初始化，对象修改的都是同一个静态变量

# 实例方法、静态方法的区别

实例方法需要将此类实例化之后才能够调用方法。

静态的方法就是不需要将类实例化就可以直接使用类名调用到方法或者属性。

# Java中的方法调用遵循值传递

⑴基本数据类型值传递

⑵引用数据类型，完全改写传入的对象，仍然是值传递（实际上new出来的是一个新的对象，方法执行完后，出栈，对原来的对象没有改变）

⑶如果传递的是对象，貌似是引用的传递，但是本质上并不是，因为地址是一个具体的值，传递的是具体的值，所以依然是值传递

Stack

“zhangjianguo”

s：0X001

setName等方法用完直接出栈

“0001”

MyDate对象

month=8

day=8

year=1949

Student对象

birthday：000X2

sex=0

name

stuno

md:000X2

args：???

main栈帧

执行完也出栈

Test

Student.class

Method Area

Heap

值传递：(形式参数类型是基本数据类型)：方法调用时，实际参数把它的值传递给对应的形式参数，形式参数只是用实际参数的值初始化自己的存储单元内容，是两个不同的存储单元，所以方法执行中形式参数值的改变不影响实际参数的值。

引用传递：(形式参数类型是引用数据类型参数)：也称为传地址。方法调用时，实际参数是对象(或数组)，这时实际参数与形式参数指向同一个地址，在方法执行中，对形式参数的操作实际上就是对实际参数的操作，这个结果在方法结束后被保留了下来，所以方法执行中形式参数的改变将会影响实际参数。

斐波那契数列：

public static int fab(int n){

if (n==2||n==1){

return 1;

}

return fab(n-1)+fab(n-2);

}

public static int fab(int n){

int f1=1;

int f2=1;

int fn=0;

for(int i=3;i<=n;i++){

fn=f1+f2;

f1=f2;

f2=fn;

}

return fn;

}

# ==和equals的区别

1. **==和equals的实质**

基本数据类型在栈中存的是其内容值，而对象类型在栈中存的是地址，这些地址指向堆中的对象，==比较的是栈中的内容，而equals比较的是堆中的内容。

1. **构造器形成的差别。**

对于String和Integer来说，由于他们特有的创建对象的方式。使用构造器和不使用构造器得到一个对象，==方法比较所产生的结果是不同的。 String a = “abc”; String b = "abc"; 此时a==b得到结果为true。String a = new String("abc"); String b = new String("abc");此时a==b得到的结果为false。Integer a = 1; Integer b = 1;此时a==b的结果是true。Integer a = new Integer(1); Integer b = new Integer(1);此时a==b得到的结果为false。

当运用引号内包含文本创建对象时,所创建的对象加入到字符串池里面.如果要创建下一个字符串对象,JVM首先会到字符串池中寻找,是否存在对应的字符串对象,如果存在,则返回一个己存在对象的引用.如果不存在,才会创建一个新的对象,并返回一个新对象引用. 所以'=='返回的是true

String str=new String("abc");创建了两个String对象，new String()是一个，参数“abc”又是一个。

1. **equals方法**。

用于比较两个独立对象的内容是否相同，字符串的比较基本都是使用equals方法。

# List、Set、Map

1. List有序

添加顺序与元素存放顺序一致有序的集合，元素可以重复

ArrayList(数组) LinkList（链表）

1. Set无序

元素不可重复HashSet数组+链表存储

1. Map

HashMap 无序，key不能重复 put(key,value)

遍历map的三种方式：

第一种：

Set keyset = map.keySet();

Iterator it = keyset.iterator();

While(it.hasNext()){

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

System.out.println(key+map.get(key));

}

第二种：

Collection values = map.values();

Interator iterator = values.iterator();

For (iterator;iterator.hasNext();){

Person person = (Person)iterator.next();

System.out.println(person);

}

第三种：

Set entries = map.entrySet();

for(Iterator iterator = entries.iterator();iterator.hasNext();){

Map.Entry entry = (Map.Entry)iterator.next();

System.out.println(entry. getKey()+entry.getValue());

}

Collections工具类 主要是针对list没有map

List l = new ArrayList();

Collections.addAll(l,”a”,”b”,”c”);

Collections.sort(l);

System.out.println(Collection.binarySearch(l,”b”));

# 包装类

**Int** [**intValue**](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\lang\Integer.html#intValue())()   
          返回该 Integer 类型的Int值

**static Int** [**parseInt**](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\lang\Integer.html#parseInt(java.lang.String))([String](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\lang\String.html) s)   
          将字符串参数作为有符号的十进制整数进行分析

**String** [**toString**](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\lang\Integer.html#toString())()   
          返回一个表示该 Integer 值的 String 对象

**static String** [**toString**](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\lang\Integer.html#toString())(i)   
          返回一个表示指定整数的 String 对象

**Integer** [**valueOf**](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\lang\Integer.html#valueOf(int))(int i)   
          返回一个表示指定的 int 值的 Integer 实例。

**Integer**[**valueOf**](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\lang\Integer.html#valueOf(java.lang.String))([String](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\lang\String.html) s)   
          返回保持指定的 String 的值的 Integer 对象。

inti = 10;

Integer ii = Integer.valueOf(i);整型转换成整型包装类

Integer k = new Integer(i); 整型转换成整型包装类

i= k.intValue(); 包装类转化成整型

String x=”100”;

Int y = Integer.valueOf(x).intValue();字符串转化成整型包装类再转化成整型

# String

new出来的对象在堆中，不new的在方法区中

Stack

abc

Method Area

Heap

m

new String(“abc”)

s

b

String.class

a

String a = “abc”;

String b = “abc”

String s = new String(“abc”)

String m = new String(“abc”)

特殊的：

String s6 = “a”;

String s6= s6+”bc”;此处有优化机制，引用s6指向new 出来的对象，不是执行方法区的字符串常量“abc“

**Int** [**charAt**](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\lang\String.html#charAt(int))(int index)   
          返回指定索引处的 char 值。

**Int** [**compareTo**](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\lang\String.html#compareTo(java.lang.String))([String](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\lang\String.html) anotherString)   
          按字典顺序比较两个字符串。

**Boolean**[**contains**](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\lang\String.html#contains(java.lang.CharSequence))([CharSequence](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\lang\CharSequence.html) s)   
          当且仅当此字符串包含 char 值的指定序列时，才返回 true。

**Boolean**[**equals**](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\lang\String.html#equals(java.lang.Object))([Object](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\lang\Object.html) anObject)   
          比较此字符串与指定的对象

**Boolean**[**getBytes**](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\lang\String.html#getBytes(java.lang.String))([String](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\lang\String.html) charsetName)   
          使用指定的字符集将此 String 解码为字节序列，并将结果存储到一个新的字节数组中。

**Int** [**indexOf**](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\lang\String.html#indexOf(int))(int ch)   
          返回指定字符在此字符串中第一次出现处的索引。

**Int** [**length**](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\lang\String.html#length())()   
          返回此字符串的长度。

**String** [**replace**](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\lang\String.html#replace(char, char))(char oldChar, char newChar)   
          返回一个新的字符串，它是通过用 newChar 替换此字符串中出现的所有 oldChar 而生成的。

**String** [**substring**](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\lang\String.html#substring(int, int))(int beginIndex, int endIndex)   
          返回一个新字符串，它是此字符串的一个子字符串。(包头不包尾)

**String** [**toString**](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\lang\String.html#toString())()   
          返回此对象本身（它已经是一个字符串！）。

**String[]** [**split**](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\lang\String.html#split(java.lang.String))([String](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\lang\String.html) regex)   
          根据给定的正则表达式的匹配来拆分此字符串。

String d = new String (s.getBytes("gbk"),"gbk");

# 正则表达式

\\代表“\”字符自身，\就是逃逸字符，对于其他的特殊字符都可以用\转义，“.”在正则表达式里有特殊含义，需要使用“\.”表示一个字符“.”

* \d 任意一个从0到9的数字
* \D 任意一个除0到9之外的字符
* \s 任意空白字符（空格、换行等等）
* \S 任意非空白字符
* . 任意字符
* \w 任意字母或数字或下划线，即a-z,A-Z,0-9或\_

定义和使用正则表达式的例子：

String s = “”;

Pattern pattern = Pattern.compile(“\\d”);//定义一个正则表达式模版

Matcher m = pattern.matcher(s);//用这个模版去匹配s

while(m.find()){

//每个被匹配的字符，该字符的起始位置，结束位置

sysout(m.group()+m.start()+m.end());

}

* 使用[]包含系列字符，能匹配其中之一；使用[^]包含系列字符，不匹配其中任意一个字符，如：
  + [abc] a或b或c
  + [^abc] 除a或b或c之外的任何字符
  + [d-g] d或e或f或g，即d到g
  + [^d-g] 除d到g之外的任意字符
  + [a-z&&[^d-g]] 从a到z之间的所有字母，但不包括d到g之间的字母
* 在之前的表达式中，尚未涉及到次数，所以，每次匹配只是一个字符，修饰次数的表达式是
  + ? 0次或1次
  + \* 0次或多次
  + + 1次或多次
  + {n} n次
  + {n,} 至少n次
  + {n,m} 至少n次，但不大于m次
* 匹配字符串开头和结尾的特殊字符
  + ^hello 字符串必需hello开头
  + hello$ 字符串必需hello结尾
* 表示“或”关系的符号“|”



* 分组符号“()”
  + 可以对()内的表达式修饰其次数
  + 想要对某个搜索结果分组获取其值的时候也用分组符号
    - 比如：((A)(B(C)))这个表达式，有4组：
      * 1、 ((A)(B(C)))
      * 2、(A)
      * 3、(B(C))
      * 4、(C)

分组下标从0开始

正则表达式的分组、断言详解：

<http://www.cnblogs.com/iyangyuan/archive/2013/05/30/3107390.html>

<http://blog.csdn.net/mutouyihao/article/details/7042160>

* 贪婪模式
  + 尽可能多地匹配
* 非贪婪模式
  + 尽可能少地匹配（只需在修饰次数的符号后面加上一个?即可）

# [正则表达式全部符号解释](http://www.cnblogs.com/yirlin/archive/2006/04/12/373222.html)

|  |  |
| --- | --- |
| **字符** | **描述** |
| \ | 将下一个字符标记为一个特殊字符、或一个原义字符、或一个向后引用、或一个八进制转义符。例如，'n' 匹配字符 "n"。'\n' 匹配一个换行符。序列 '\\' 匹配 "\" 而 "\(" 则匹配 "("。 |
| ^ | 匹配输入字符串的开始位置。如果设置了 RegExp 对象的 Multiline 属性，^ 也匹配 '\n' 或 '\r' 之后的位置。 |
| $ | 匹配输入字符串的结束位置。如果设置了RegExp 对象的 Multiline 属性，$ 也匹配 '\n' 或 '\r' 之前的位置。 |
| \* | 匹配前面的子表达式零次或多次。例如，zo\* 能匹配 "z" 以及 "zoo"。\* 等价于{0,}。 |
| + | 匹配前面的子表达式一次或多次。例如，'zo+' 能匹配 "zo" 以及 "zoo"，但不能匹配 "z"。+ 等价于 {1,}。 |
| ? | 匹配前面的子表达式零次或一次。例如，"do(es)?" 可以匹配 "do" 或 "does" 中的"do" 。? 等价于 {0,1}。 |
| {n} | n 是一个非负整数。匹配确定的 n 次。例如，'o{2}' 不能匹配 "Bob" 中的 'o'，但是能匹配 "food" 中的两个 o。 |
| {n,} | n 是一个非负整数。至少匹配n 次。例如，'o{2,}' 不能匹配 "Bob" 中的 'o'，但能匹配 "foooood" 中的所有 o。'o{1,}' 等价于 'o+'。'o{0,}' 则等价于 'o\*'。 |
| {n,m} | m 和 n 均为非负整数，其中n <= m。最少匹配 n 次且最多匹配 m 次。例如，"o{1,3}" 将匹配 "fooooood" 中的前三个 o。'o{0,1}' 等价于 'o?'。请注意在逗号和两个数之间不能有空格。 |
| ? | 当该字符紧跟在任何一个其他限制符 (\*, +, ?, {n}, {n,}, {n,m}) 后面时，匹配模式是非贪婪的。非贪婪模式尽可能少的匹配所搜索的字符串，而默认的贪婪模式则尽可能多的匹配所搜索的字符串。例如，对于字符串 "oooo"，'o+?' 将匹配单个 "o"，而 'o+' 将匹配所有 'o'。 |
| . | 匹配除 "\n" 之外的任何单个字符。要匹配包括 '\n' 在内的任何字符，请使用象 '[.\n]' 的模式。 |
| (pattern) | 匹配 pattern 并获取这一匹配。所获取的匹配可以从产生的 Matches 集合得到，在VBScript 中使用 SubMatches 集合，在JScript 中则使用 $0…$9 属性。要匹配圆括号字符，请使用 '\(' 或 '\)'。 |
| (?:pattern) | 匹配 pattern 但不获取匹配结果，也就是说这是一个非获取匹配，不进行存储供以后使用。这在使用 "或" 字符 (|) 来组合一个模式的各个部分是很有用。例如， 'industr(?:y|ies) 就是一个比 'industry|industries' 更简略的表达式。 |
| (?=pattern) | 正向预查，在任何匹配 pattern 的字符串开始处匹配查找字符串。这是一个非获取匹配，也就是说，该匹配不需要获取供以后使用。例如，'Windows (?=95|98|NT|2000)' 能匹配 "Windows 2000" 中的 "Windows" ，但不能匹配 "Windows 3.1" 中的 "Windows"。预查不消耗字符，也就是说，在一个匹配发生后，在最后一次匹配之后立即开始下一次匹配的搜索，而不是从包含预查的字符之后开始。 |
| (?!pattern) | 负向预查，在任何不匹配 pattern 的字符串开始处匹配查找字符串。这是一个非获取匹配，也就是说，该匹配不需要获取供以后使用。例如'Windows (?!95|98|NT|2000)' 能匹配 "Windows 3.1" 中的 "Windows"，但不能匹配 "Windows 2000" 中的 "Windows"。预查不消耗字符，也就是说，在一个匹配发生后，在最后一次匹配之后立即开始下一次匹配的搜索，而不是从包含预查的字符之后开始 |
| x|y | 匹配 x 或 y。例如，'z|food' 能匹配 "z" 或 "food"。'(z|f)ood' 则匹配 "zood" 或 "food"。 |
| [xyz] | 字符集合。匹配所包含的任意一个字符。例如， '[abc]' 可以匹配 "plain" 中的 'a'。 |
| [^xyz] | 负值字符集合。匹配未包含的任意字符。例如， '[^abc]' 可以匹配 "plain" 中的'p'。 |
| [a-z] | 字符范围。匹配指定范围内的任意字符。例如，'[a-z]' 可以匹配 'a' 到 'z' 范围内的任意小写字母字符。 |
| [^a-z] | 负值字符范围。匹配任何不在指定范围内的任意字符。例如，'[^a-z]' 可以匹配任何不在 'a' 到 'z' 范围内的任意字符。 |
| \b | 匹配一个单词边界，也就是指单词和空格间的位置。例如， 'er\b' 可以匹配"never" 中的 'er'，但不能匹配 "verb" 中的 'er'。 |
| \B | 匹配非单词边界。'er\B' 能匹配 "verb" 中的 'er'，但不能匹配 "never" 中的 'er'。 |
| \cx | 匹配由 x 指明的控制字符。例如， \cM 匹配一个 Control-M 或回车符。x 的值必须为 A-Z 或 a-z 之一。否则，将 c 视为一个原义的 'c' 字符。 |
| \d | 匹配一个数字字符。等价于 [0-9]。 |
| \D | 匹配一个非数字字符。等价于 [^0-9]。 |
| \f | 匹配一个换页符。等价于 \x0c 和 \cL。 |
| \n | 匹配一个换行符。等价于 \x0a 和 \cJ。 |
| \r | 匹配一个回车符。等价于 \x0d 和 \cM。 |
| \s | 匹配任何空白字符，包括空格、制表符、换页符等等。等价于 [ \f\n\r\t\v]。 |
| \S | 匹配任何非空白字符。等价于 [^ \f\n\r\t\v]。 |
| \t | 匹配一个制表符。等价于 \x09 和 \cI。 |
| \v | 匹配一个垂直制表符。等价于 \x0b 和 \cK。 |
| \w | 匹配包括下划线的任何单词字符。等价于'[A-Za-z0-9\_]'。 |
| \W | 匹配任何非单词字符。等价于 '[^A-Za-z0-9\_]'。 |
| \xn | 匹配 n，其中 n 为十六进制转义值。十六进制转义值必须为确定的两个数字长。例如，'\x41' 匹配 "A"。'\x041' 则等价于 '\x04' & "1"。正则表达式中可以使用 ASCII 编码。. |
| \num | 匹配 num，其中 num 是一个正整数。对所获取的匹配的引用。例如，'(.)\1' 匹配两个连续的相同字符。 |
| \n | 标识一个八进制转义值或一个向后引用。如果 \n 之前至少 n 个获取的子表达式，则 n 为向后引用。否则，如果 n 为八进制数字 (0-7)，则 n 为一个八进制转义值。 |
| \nm | 标识一个八进制转义值或一个向后引用。如果 \nm 之前至少有 nm 个获得子表达式，则 nm 为向后引用。如果 \nm 之前至少有 n 个获取，则 n 为一个后跟文字 m 的向后引用。如果前面的条件都不满足，若 n 和 m 均为八进制数字 (0-7)，则 \nm 将匹配八进制转义值 nm。 |
| \nml | 如果 n 为八进制数字 (0-3)，且 m 和 l 均为八进制数字 (0-7)，则匹配八进制转义值 nml。 |
| \un | 匹配 n，其中 n 是一个用四个十六进制数字表示的 Unicode 字符。例如， \u00A9 匹配版权符号 (?)。 |

# StringBuffer

可变长度的字符串

StringBuffer转化为String类型用toString

String类型转化为StringBuffer：

StringBuffer b = new StringBuffer(“hello”);

**StringBuffer** [**append**](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\lang\StringBuffer.html#append(java.lang.String))([String](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\lang\String.html) str)   
          将指定的字符串追加到此字符序列。

**StringBuffer**[**delete**](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\lang\StringBuffer.html#delete(int, int))(int start, int end)   
          移除此序列的子字符串中的字符。

**StringBuffer**[**insert**](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\lang\StringBuffer.html#insert(int, java.lang.String))(int offset, [String](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\lang\String.html) str)   
          将字符串插入此字符序列中。

**String** [**toString**](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\lang\StringBuffer.html#toString())()   
          返回此序列中数据的字符串表示形式

**StringBuffer**[**replace**](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\lang\StringBuffer.html#replace(int, int, java.lang.String))(int start, int end, [String](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\lang\String.html) str)   
          使用给定 String 中的字符替换此序列的子字符串中的字符。

计算程序运行时间：

long start = System.currentTimeMillis();

long end = System.currentTimeMillis();

sysout(“共耗时：”+(end-start));

StringBuilder类是JDK1.5新增加的一个类，作用与StringBuffer一样，不过StringBuilder是线程不安全的，而StringBuffer是线程安全的，StringBuilder的性能要优于StringBuffer

# Math

**static int** [**abs**](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\lang\Math.html#abs(int))(int a)   
          返回 int 值的绝对值。

**staticdouble**[**cos**](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\lang\Math.html#cos(double))(double a)   
          返回角的三角余弦

**staticdouble**[**sin**](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\lang\Math.html#sin(double))(double a)   
          返回角的三角正弦。

**staticdouble**[**exp**](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\lang\Math.html#exp(double))(double a)   
          返回欧拉数 *e* 的 double 次幂的值。

**static int** [**max**](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\lang\Math.html#max(int, int))(int a, int b)   
          返回两个 int 值中较大的一个。

**static int**[**min**](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\lang\Math.html#min(int, int))(int a, int b)   
          返回两个 int 值中较小的一个。

**staticdouble**[**log10**](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\lang\Math.html#log10(double))(double a)   
          返回 double 值的底数为 10 的对数。

**staticdouble**[**pow**](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\lang\Math.html#pow(double, double))(double a, double b)   
          返回第一个参数的第二个参数次幂的值。

**static int**[**round**](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\lang\Math.html#round(float))(float a)   
          返回最接近参数的 int。

**staticdouble**[**sqrt**](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\lang\Math.html#sqrt(double))(double a)   
          返回正确舍入的 double 值的正平方根。

Math.random(); //取值范围[0,1)

(int)(Math.random()\*10) //10以内的随机数

Random r =new Random();

r.nextInt(10); //10以内的随机数

# File

取得环境变量 System.getenv(“Path”); //map类型

取得当前工作目录 System.getProperty（“user.dir”）;

//相对路径

File f=newFile("a\\b\\c.txt");

if (f.exists()) { //判断文件是否存在

if (!f.isDirectory()) { //如果不是目录

System.out.println("开始调用文件属性");

System.out.println("文件名称:" + f.getName());

System.out.println("是否文件:" + f.isFile());

System.out.println("文件大小:" + f.length());

System.out.println("文件对象代表的路径:" +

f.getPath());

System.out.println("父目录的完整路径:" +

f.getParentFile().getPath());

System.out.println("父目录的完整路径:" +

f.getParent());

System.out.println("绝对路径:" +

f.getAbsolutePath());

Date d=new Date(f.lastModified());

System.out.println("最后修改时间:"+d);

//f.delete();

f.renameTo(new File(f.getParent(),"ccc.txt"));

}else{

String[] files=f.list(); //当前目录下的子

目录或文件所组成名称数组

for(String filename:files){

System.out.println(filename);

}

}

}else{

f.getParentFile().mkdirs();//创建目录

try {

f.createNewFile();//创建文件

} catch (IOException e) {

e.printStackTrace();

}

**static String** [**separator**](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\io\File.html#separator)   
          与系统有关的默认名称分隔符，出于方便考虑，它被表示为一个字符串。

[**File**](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\io\File.html#File(java.io.File, java.lang.String))([File](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\io\File.html) parent, [String](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\lang\String.html) child)   
          根据 parent 抽象路径名和 child 路径名字符串创建一个新 File 实例

[**File**](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\io\File.html#File(java.lang.String))([String](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\lang\String.html) pathname)   
          通过将给定路径名字符串转换成抽象路径名来创建一个新 File 实例。

**Boolean** [**createNewFile**](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\io\File.html#createNewFile())()   
          当且仅当不存在具有此抽象路径名指定的名称的文件时，原子地创建由此抽象路径名指定的一个新的空文件。

**Boolean**[**delete**](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\io\File.html#delete())()   
          删除此抽象路径名表示的文件或目录。

**Boolean**[**exists**](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\io\File.html#exists())()   
          测试此抽象路径名表示的文件或目录是否存在。

**String** [**getAbsolutePath**](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\io\File.html#getAbsolutePath())()   
          返回抽象路径名的绝对路径名字符串。

**String** [**getParent**](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\io\File.html#getParent())()   
          返回此抽象路径名的父路径名的路径名字符串，如果此路径名没有指定父目录，则返回 null。

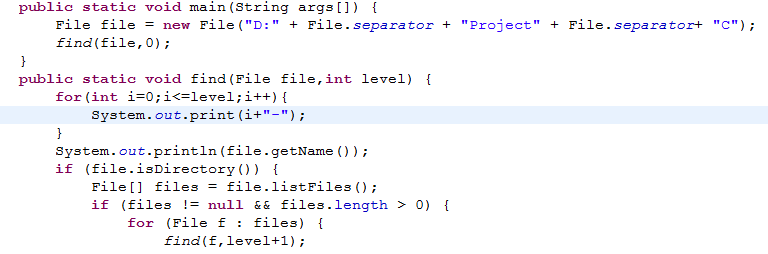
**File** [**getParentFile**](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\io\File.html#getParentFile())()   
          返回此抽象路径名的父路径名的抽象路径名，如果此路径名没有指定父目录，则返回 null。

**Boolean** [**isFile**](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\io\File.html#isFile())()   
          测试此抽象路径名表示的文件是否是一个标准文件

**String[]** [**list**](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\io\File.html#list())()   
          返回由此抽象路径名所表示的目录中的文件和目录的名称所组成字符串数组。

**Boolean** [**mkdirs**](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\io\File.html#mkdirs())()   
          创建此抽象路径名指定的目录，包括创建必需但不存在的父目录。

**Boolean** [**renameTo**](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\io\File.html#renameTo(java.io.File))([File](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\io\File.html) dest)   
          重新命名此抽象路径名表示的文件。



# 枚举类型

**publicenum**CardStatus {

*BLACK*,*INIT*,*BAD*

}

**publicstaticvoid** main(String[] args) {

**for** (CardStatus s : CardStatus.*values*()) {

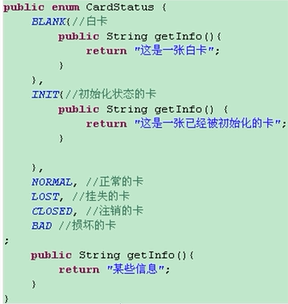
System.*out*.println(s.name());

}

}

可以把CardStatus看成一个类，而BLACK，INIT等是这个类的实例

可以通过名称访问枚举类型的实例



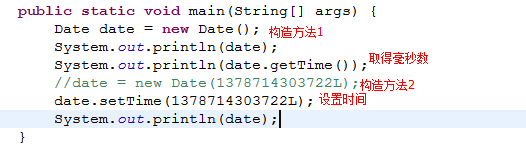
# 日期处理

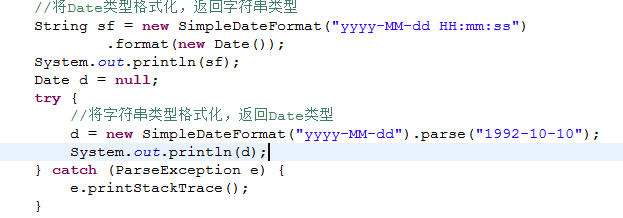
new Date()时间差八个小时的时候

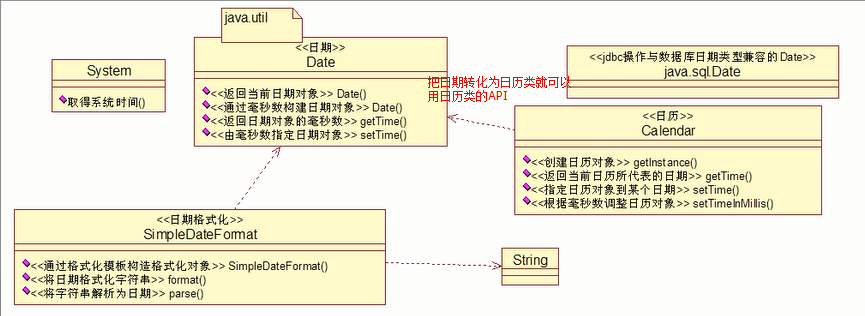
D:\Java\jdk1.7.0\_07\jre\lib\zi\Etc\GMT-8

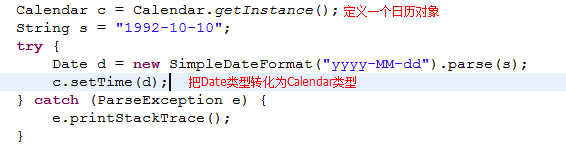
文件替换掉

D:\Java\jdk1.7.0\_07\jre\lib\zi\GMT











Timestamp是时间戳，年月日+时分秒

Date是时间类型，只有年月日

用日历类获取当前月

GregorianCalendar g=**new**GregorianCalendar();

**int** flag = (**int**)g.get(Calendar.*MONTH*)+1;

# 单例模式

把默认的构造方法设置成私有的，在类外边不能调用new出实例,new 是类创建对象的关键字，new 后面就是类的构造函数

单例设计模式：

1. 饿汉式(容易造成资源浪费)

public class UserService{

private static final UserService us = new UserService();

//如果new出来的是一个List集合，就称为多例

private UserService(){ }

public static UserService getInstance(){ //静态工厂方法

return us;

}

}

1. 懒汉式

public class UserService{

private static UserService us;

private UserService(){ }

public static synchronizedUserService getInstance(){

if (us==null){

us = new UserService();

}

return us;

}

}

# 反射机制

1. Class 对应字节码，是一个普通的类，继承了Object
2. Class代表了Java中的所有类型（包含基本数据类型）
3. 在java中，每种类型都有一个相应的Class对象，当编写好一个类，编译完成后，在生成的.class文件中，就产生一个Class对象，用来表示这个类的类型信息
4. 所有的引用类型包含基本数据类型都有一个 **成员.class** 能够生成此种类型对应的Class类型的实例
5. 所有引用类型都有一个继承自Object的getClass()，能够生成此种实例对应的类型对应的Class类型的实例
6. Class.forName()的用法

Java里的任何class都要装载在虚拟机上才能运行，Class.forName()就是装载类用的（和new不一样）

静态代码是和class绑定的，class装载成功就表示执行了你的静态代码，而且以后就不会在走这段静态代码了。

Class.forName()返回一个类，作用是要求JVM查找并加载指定的类，也就是说JVM会执行该类的静态代码段

如果是全路径类名字符串，可以由

Class.forName(“java.lang.String”).newInstance()来动态加载和创建Class对象

newInstance()和new一个是方法，一个是关键字，前者使用类加载机制，后者是创建一个新类

工厂模式常有以下用法

String className = readfromXMlConfig;//从xml 配置文件中获得字符串   
class c = Class.forName(className);   
factory = (ExampleInterface)c.newInstance();

使用newInstance()方法的时候，就必须保证：1、这个类已经加载；2、这个类已经连接了。而完成上面两个步骤的正是Class的静态方法forName()所完成的，这个静态方法调用了启动类加载器，即加载 java API的那个加载器。

newInstance: 弱类型。低效率。只能调用无参构造。

new: 强类型。相对高效。能调用任何public构造。

1. 拿到Class：

//基本数据类型对应的Class类型的实例

Class c = int.class; 打印c为int

后三种打印结果都是class java.lang.String

//由实例推导出其类型对应的Class类型的实例

Class c2 = “abc”.getClass();

//引用类型String对应的Class类型的实例

Class c3 = String.class;

//如果是全路径类名字符串，可以由Class.forName()来生成Class类型的实例

Class c4 = Class.forName(“java.lang.String”);

通过c4.newInstance()可以产生实例

堆

栈

Class

Person对象

方法区

p

Person.class

1. 取得类的属性方法等

Class c = Person.class;

//Class c2 = new Person().getClass();

//Class c3 = Class.forName(“com.Person”);

取得所有成员变量：(引入的类都是反射包里的)

getFields() 取得本类及父类的public成员变量

getDeclaredFields() 取得本类中所有权限的变量，取不到父类的

getMethods()本类及其父类（包括父类的父类）的public方法

getDeclaredMethods()仅取得本类中的所有方法，不考虑权限

getConstructors() 取得本类中的public构造器，没有父类的

getDeclaredConstructors() 取得本类的所有构造器

Java里边是单继承，只能继承一个类，但可以实现多个接口

getSuperclass()取得父类

getInterfaces()取得接口

总结：get一个单数的是按名字取得，get复数是取出一类的，并放在数组里；加declared的是本类所有权限的，不加declared是取得本类及其父类的（构造器除外）public类型

1. 取得某一个属性的名字，类型等

Class c = Student.class;

Student s = (Student)c.newInstance();

Field f= c.getDeclaredField(“birthday”);

f.getName() 取得属性名

f.getType().getName()取得属性类型

Modifier.toString(f.getModifiers())取得属性的修饰符

访问属性值的时候就得基于类的实例层面了

f.get(s);取得实例对象s 的schoolName属性的值

f.set(s,”北大”);设置对象s的schoolName属性的值

由封装性决定了不能给private类型的属性在外面赋值，在反射里可以设置可见性

f.setAccessible(true);设置可见性

1. 取得某一个方法的名字，类型等

Class c = Student.class;

Method m = c.getDeclareMethod(“m5”);

m.getName();取得方法名

m.getReturnType()取得返回值类型

Modifier.toString(m.getModifiers())取得方法修饰符

取得有形参的方法：

Method m=

c.getDeclaredMethod(“m7”,String.class,int.class);

Arrays.toString(m.getParameterType()) 取得参数类型

动态调用方法也是基于实例对象的层面的：

Student s = (Student)c.newInstance();

Object r = m.invoke(s,”abc”,10);动态方法调用，第一参数是实例对象，后两个参数是给方法传的参数，可以取得返回值，如果为void ，返回null

在方法动态调用时如果出错，会包装成InvoketionException抛出，要想直接输出错误的根源可以在catch里这样：

system.out.println(e.getCause().getMessage());

练习：运用反射实现一个对象的属性值拷贝到另一个类的对象的同名属性上

public static void CopyProperties(Object src,Object dest){

Class srcClass = src.getClass();

Class destClass = dest.getClass();

Method[] methods = srcClass.getMethods();

for(Method m = methods){

if(m.getName().startWith(“get”)||

m.getName().startWith(“is”)){

Object value = m.invoke(src);//getter没有参数

//拼接出对应的setter的名字

String setterName=”set”+

(m.getName().startWith(“get”)?m.getName().substring(3): m.getName().substring(2));

//取得对应的setter方法,参数类型与getter的返回值类型保持一致

Method setterMethod =

destClass.getMehod(setterName,m.getReturnType());

//调用setter方法

setterMethod.invoke(dest,value);

}

}

}

# 内部类

内部类是指在一个外部类的内部再定义一个类。类名不需要和文件夹相同。

内部类可以是静态static的，也可用public，default，protected和private修饰，而外部顶级类（类名和文件名相同）只能使用public和default。

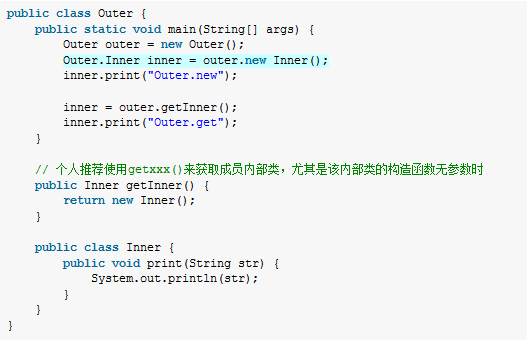
注意：内部类是一个编译时的概念，一旦编译成功，就会成为完全不同的两类。对于一个名为outer的外部类和其内部定义的名为inner的内部类。编译完成后出现outer.class和outer$inner.class两类。所以内部类的成员变量/方法名可以和外部类的相同。

1. 成员内部类：

成员内部类就是作为外部类的成员，可以直接使用外部类的所有成员和方法，即使是private的。同时外部类要访问内部类的所有成员变量/方法，则需要通过内部类的对象来获取。要注意的是，成员内部类不能含有static的变量和方法。**因为成员内部类需要先创建了外部类，才能创建它自己的**

在成员内部类要引用外部类对象时，使用outer.this来表示外部类对象；

而需要创建内部类对象Outer.Inner inner = outer.new Inner();

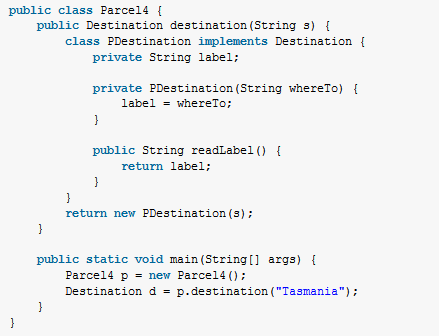


1. 局部内部类:

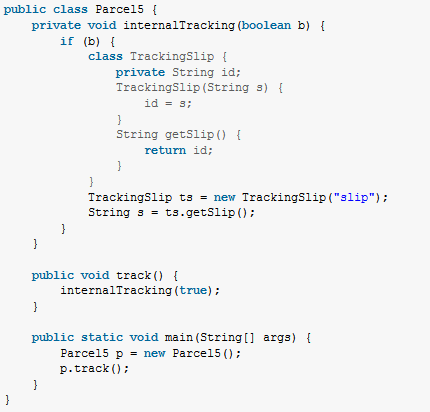
局部内部类，是指内部类定义在方法和作用域内

局部内部类也像别的类一样进行编译，但只是作用域不同而已，只在该方法或条件的作用域内才能使用，退出这些作用域后无法引用的。

定义在方法中：



定义在作用域里：



1. 嵌套内部类：

嵌套内部类，就是修饰为static的内部类。声明为static的内部类，不需要内部类对象和外部类对象之间的联系，就是说我们可以直接引用outer.inner，即不需要创建外部类，也不需要创建内部类。

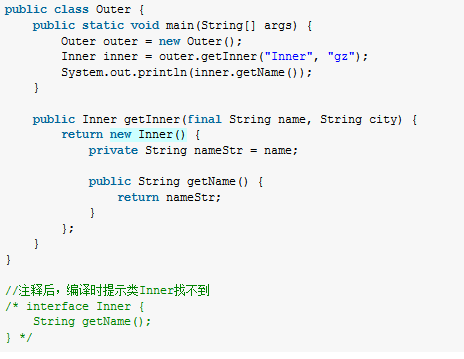
嵌套类和普通的内部类还有一个区别：普通内部类不能有static数据和static属性，也不能包含嵌套类，但嵌套类可以。而嵌套类不能声明为private，一般声明为public，方便调用。

1. 匿名内部类：

有时候我为了免去给内部类命名，便倾向于使用匿名内部类，因为它没有名字。

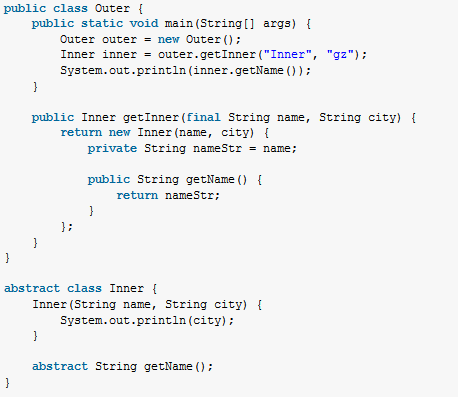
匿名内部类是不能加访问修饰符的。**要注意的是，new 匿名类，这个类是要先定义的**

**当所在的方法的形参需要被内部类里面使用时，该形参必须为final，final定义一个常量**



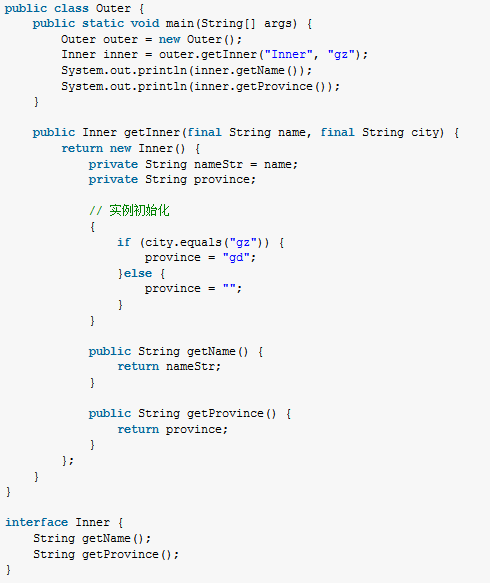
内部类并不是直接调用方法传进来的参数，而是内部类将传进来的参数通过自己的构造器备份到了自己的内部，自己内部的方法调用的实际是自己的属性而不是外部类方法的参数。

使用匿名类的带参数的构造方法



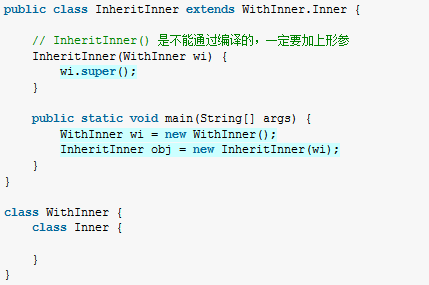
注意这里的形参city，由于它没有被匿名内部类直接使用，而是被抽象类Inner的构造函数所使用，所以不必定义为final。

而匿名内部类通过实例初始化，可以达到类似构造器的效果：



1. 内部类的继承

内部类的继承，是指内部类被继承，普通类 extents 内部类。而这时候代码上要有点特别处理



子类的构造函数里面要使用**父类的外部类对象.super()**;而这个对象需要从外面创建并传给形参。

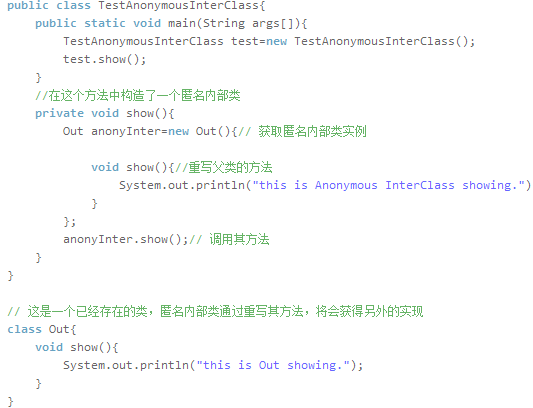
# 匿名类

匿名类是不能有名字的类，它们不能被引用，只能在创建时用New语句来声明它们匿名类的目的是在某个地方需要特殊的实现，因此在该处编写其实现，并获取它的实例，调用它的方法。不要在匿名内部类编写其他的方法，是不可见的。

匿名类的实现方式：

 继承一个类，重写其方法

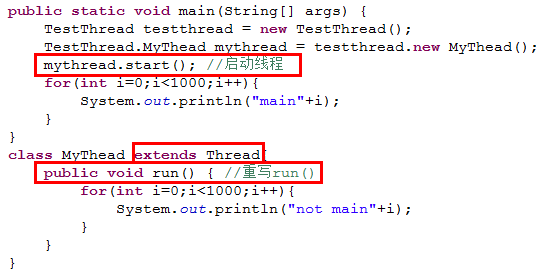
 实现一个接口，可以是多个



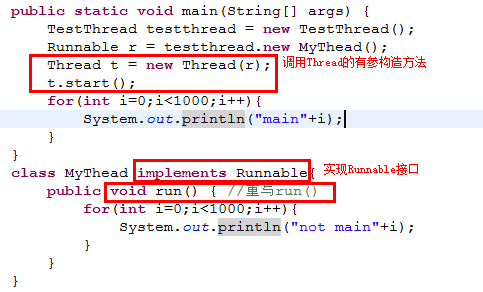
# 线程

1. 进程是一个应用程序的实例，一个进程由一个或多个线程组成
2. 创建线程，线程一旦启动就不能重新启动

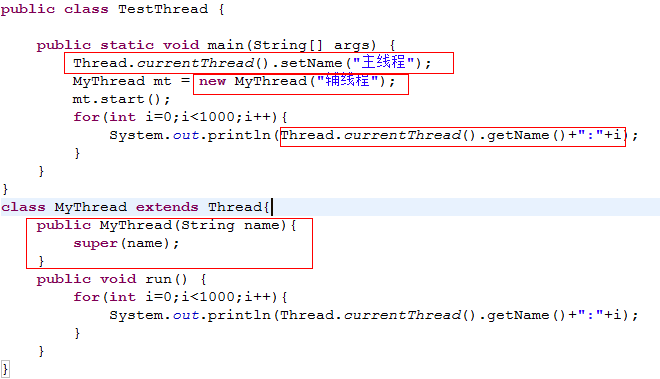
继承Thread类，重写run()



实现runnable接口，重写run()，需要new线程类

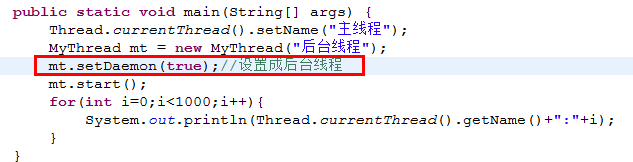


取得当前线程的名字：Thread.currentThread().getName() 等同this.getName()只有是线程实例的时候可以用this，主线程就不能用了

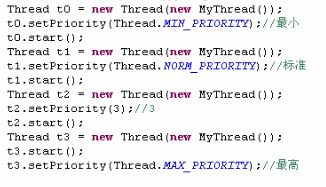


1. 后台线程

当非后台线程结束时，后台线程也结束



1. isAlive()，判断线程是否终止，只要run方法没有结束就一直返回true
2. Thread .sleep(1000)静态方法,睡眠
3. Thread .yield()让步，让给其他线程执行
4. 设置线程优先级：（1到10，数越大优先级越高）



SetPriority()方法可以在任何时候调用（不一定在start()之前调用）

1. 线程合并，join()调用了哪个线程实例的join()方法，哪个线程会先执行完，类似于方法调用
2. 线程中断，stop()方法，不建议使用

mt.interrupt()通知线程中断了

if（mt.interrupted）{……}判断是否有中断通知

这种方法在与sleep()同时使用时有时候会不起作用

解决办法就是重写interrupt()方法

Private Boolean status = true;

@overide

interrupt(){ Boolean = false ;}

if(Boolean = false){……}

1. 共享资源

（1）对象级别的锁（非静态方法）

Synchronized 锁是加在对象上的，一个对象只有一把锁，不会有一个对象的多把锁同时起作用的情况（Synchronized无论加在哪里，只要能保证是同一个对象就可以达到排他的目的），只要一个线程访问了其中一个Synchronized方法，其他线程不能同时访问这个对象中任何一个Synchronized方法（对象级别的，因为锁是加在对象上的）

如果把Synchronized加到一个实例方法前，那么不能达到预期目的

（2）类级别的锁（静态方法）

Synchronized加在一个静态方法上，相当于给类类型加了一把锁，只要一个静态方法被访问，同一个类的其他静态方法就不能访问了（类级别的）

（3）Synchronized加到一个同步代码块上，把Synchronized的控制粒度变小

1. 死锁
2. 线程同步协作

Notify()唤醒一个线程

唤醒在此对象监视器上等待的单个线程。如果所有线程都在此对象上等待，则会选择唤醒其中一个线程。被唤醒的线程将以常规方式与在该对象上主动同步的其他所有线程进行竞争，一次只能有一个线程拥有对象的监视器

NotifyAll()唤醒所有的线程

唤醒在此对象监视器上等待的所有线程

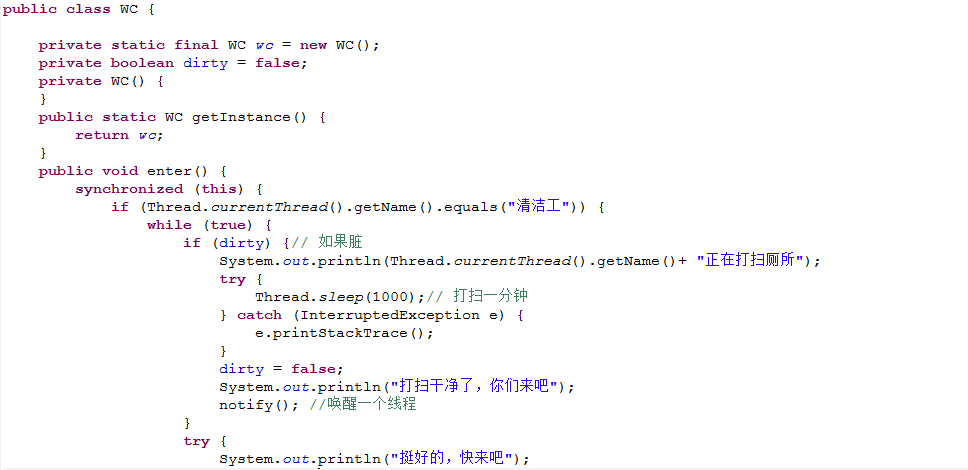
Wait()释放锁，进入等待

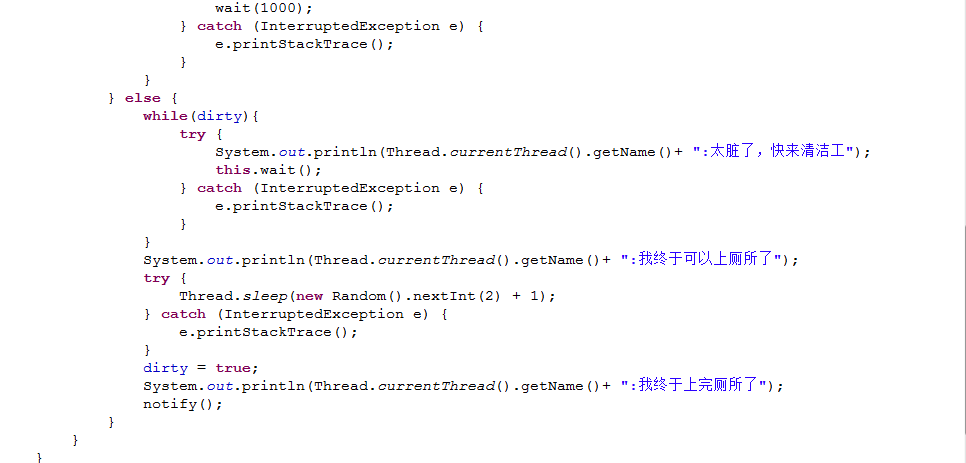
导致当前的线程等待，直到其他线程调用此对象的 [notify()](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\lang\Object.html#notify%28%29) 方法或 [notifyAll()](file:///D:\%E5%AD%A6%E4%B9%A0%E8%A7%86%E9%A2%91\0100_JavaSE\document\jdk-1_5_0-doc-api-zh_CN\api\java\lang\Object.html#notifyAll%28%29) 方法

Wait(1000)释放锁，进入等待，1000毫秒后自动唤醒



生产者消费者





# Thread的start和run

1） start：

用 start方法来启动线程，真正实现了多线程运行，这时无需等待run方法体代码执行完毕而直接继续执行下面的代码（因为run在新的线程里执行）。通过调用Thread类的 start()方法来启动一个线程，这时此线程处于就绪（可运行）状态，并没有运行，一旦得到cpu时间片，就开始执行run()方法，这里方法 run()称为线程体，它包含了要执行的这个线程的内容，Run方法运行结束，此线程随即终止。

2） run：

run()方法只是类的一个普通方法而已，如果直接调用Run方法，程序中依然只有主线程这一个线程，其程序执行路径还是只有一条，还是要顺序执行，还是要等待run方法体执行完毕后才可继续执行下面的代码，这样就没有达到写线程的目的。

总结：调用start方法方可启动线程，而run方法只是thread的一个普通方法调用，还是在主线程里执行。

# 线程同步

什么是线程同步？

当使用多个线程来访问同一个数据时，非常容易出现线程安全问题(比如多个线程都在操作同一数据导致数据不一致),所以我们用同步机制来解决这些问题。

参考网址：<http://www.cnblogs.com/sunzn/archive/2013/02/13/2910895.html>

实现同步机制有两个方法：

1。同步代码块：

synchronized(同一个数据){} 同一个数据：就是N条线程同时访问一个数据。

public class SynchronizeCode {

public static void main(String[] args) {

/\*\*\*\*\*\*\*\*\*\*\*\* 创建锁对象 \*\*\*\*\*\*\*\*\*\*\*\*/

final Object lock = new Object();

/\*\*\*\*\*\*\*\*\*\*\*\* 开启线程一 \*\*\*\*\*\*\*\*\*\*\*\*/

new Thread() {

public void run() {

while (true) {

synchronized (lock) {

System.out.print("同步");

System.out.println("代码");

}

}

};

}.start();

/\*\*\*\*\*\*\*\*\*\*\*\* 开启线程二 \*\*\*\*\*\*\*\*\*\*\*\*/

new Thread() {

public void run() {

while (true) {

synchronized (lock) {

System.out.print("Synchronize");

System.out.println("Code");

}

}

};

}.start();

}

}

2。

同步方法：

public synchronized 数据返回类型方法名(){}

就是使用 synchronized 来修饰某个方法，则该方法称为同步方法。对于同步方法而言，无需显示指定同步监视器，同步方法的同步监视器是 this 也就是该对象的本身（这里指的对象本身有点含糊，其实就是调用该同步方法的对象）通过使用同步方法，可非常方便的将某类变成线程安全的类，具有如下特征：

1，该类的对象可以被多个线程安全的访问。

2，每个线程调用该对象的任意方法之后，都将得到正确的结果。

3，每个线程调用该对象的任意方法之后，该对象状态依然保持合理状态。

注：synchronized关键字可以修饰方法，也可以修饰代码块，但不能修饰构造器，属性等。

实现同步机制注意以下几点：安全性高，性能低，在多线程用。性能高，安全性低，在单线程用。

1，不要对线程安全类的所有方法都进行同步，只对那些会改变共享资源方法的进行同步。

2，如果可变类有两种运行环境，当线程环境和多线程环境则应该为该可变类提供两种版本：线程安全版本和线程不安全版本(没有同步方法和同步块)。在单线程中环境中，使用线程不安全版本以保证性能，在多线程中使用线程安全版本.

线程通讯：

为什么要使用线程通讯？

当使用synchronized 来修饰某个共享资源时(分同步代码块和同步方法两种情况）,当某个线程获得共享资源的锁后就可以执行相应的代码段，直到该线程运行完该代码段后才释放对该共享资源的锁，让其他线程有机会执行对该共享资源的修改。当某个线程占有某个共享资源的锁时，如果另外一个线程也想获得这把锁运行就需要使用wait() 和notify()/notifyAll()方法来进行线程通讯了。

Java.lang.object 里的三个方法wait() notify() notifyAll()

wait方法导致当前线程等待，直到其他线程调用同步监视器的notify方法或notifyAll方法来唤醒该线程。

wait(mills)方法

都是等待指定时间后自动苏醒，调用wait方法的当前线程会释放该同步监视器的锁定，可以不用notify或notifyAll方法把它唤醒。

notify()

唤醒在同步监视器上等待的单个线程，如果所有线程都在同步监视器上等待，则会选择唤醒其中一个线程，选择是任意性的，只有当前线程放弃对该同步监视器的锁定后，也就是使用wait方法后，才可以执行被唤醒的线程。

notifyAll()方法

唤醒在同步监视器上等待的所有的线程。只用当前线程放弃对该同步监视器的锁定后，才可以执行被唤醒的线程

# 静态代码块和静态方法

1. 两者的区别就是:

静态代码块是在项目启动的时候自动执行的（只执行一次，静态代码块常用来执行类属性的初始化）;  
静态方法在项目启动的时候就初始化，在被调用的时候才执行的.

1. 类装载步骤

在Java中，类装载器把一个类装入Java虚拟机中，要经过三个步骤来完成：装载、链接和初始化，其中链接又可以分成校验、准备和解析三步，除了解析外，其它步骤是严格按照顺序完成的，各个步骤的主要工作如下：

装载：查找和导入类或接口的二进制数据；  
链接：执行下面的校验、准备和解析步骤，其中解析步骤是可以选择的；  
校验：检查导入类或接口的二进制数据的正确性；  
准备：给类的静态变量分配并初始化存储空间；  
解析：将符号引用转成直接引用；  
初始化：激活类的静态变量的初始化Java代码和静态Java代码块。  
初始化类中属性是静态代码块的常用用途，但只能使用一次。

1. 对象的初始化顺序

首先执行父类静态的内容，父类静态的内容执行完毕后，接着去执行子类的静态的内容，当子类的静态内容执行完毕之后，再去看父类有没有非静态代码块，如果有就 执行父类的非静态代码块，父类的非静态代码块执行完毕，接着执行父类的构造方法；父类的构造方法执行完毕之后，它接着去看子类有没有非静态代码块，如果有 就执行子类的非静态代码块。子类的非静态代码块执行完毕再去执行子类的构造方法。总之一句话，静态代码块内容先执行，接着执行父类非静态代码块和构造方 法，然后执行子类非静态代码块和构造方法。

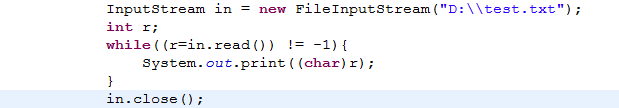
# I/O

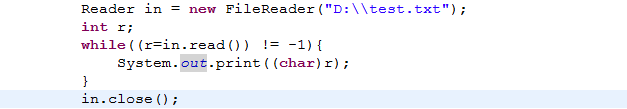
1. 输入、输出是站在程序的角度
2. 输入流：InputStream Reader 输出流： OutputStream Writer

字节流：InputStream OutputStream 字符流：Reader Writer

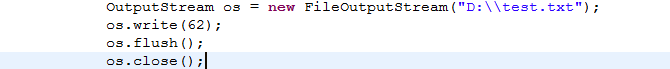
（一个字符占两个字节，一个汉字占两个字节，一个英文字母占一个字节）

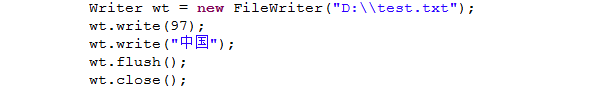
1. 输入流





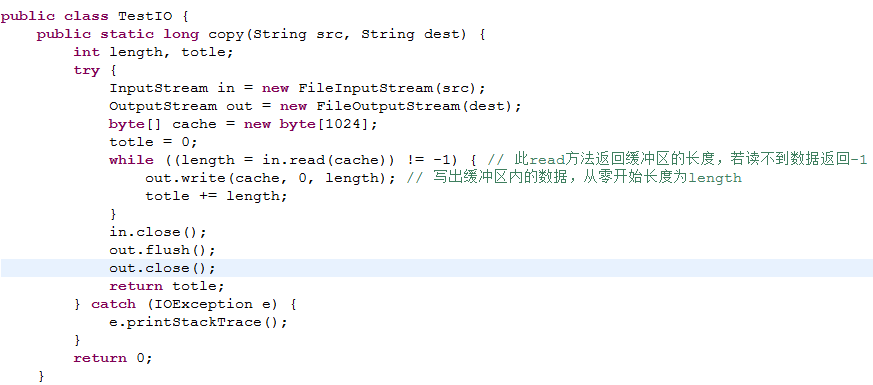
1. 输出流不需要手动catch FileNotFoundException，可以自动创建文件，但不会自动创建不存在的目录结构

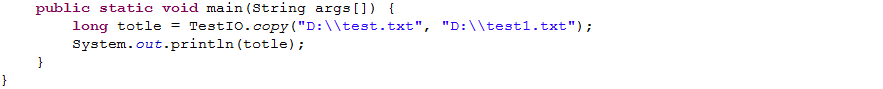




FileOutputStream(String name,Boolean b),第一个参数是路径，第二个参数表示在文件后面追加内容，不覆盖原有的内容

1. 拷贝

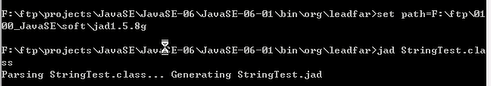




1. 原始流是直接作用于数据源的流，利用适配器模式，它把File，String，ByteArray等与四大基流整合在一起（File，String，ByteArray作为构造器的参数传递给FileInputStream，StringReader等）



反编译的方法





在cmd到字节码的根目录

set path=反编译工具路径

jad -sjava 字节码文件名

type .jad文件

jad -o -r -dF:\src -sjava F:\classes\\*\*\\*.class   
-o  - overwrite output files without confirmation (default: no) 无需确定覆写文件   
-r  - restore package directory structrure 恢复包目录结构   
-s - output file extension (by default '.jad') 如果不设置为-sjava，则默认扩展名为.jad   
其他的，F:\classes\\*\*\\*.class 中的两颗接连的星，表示任意层次的子目录。

**Java关键字final、static使用总结**

一、final

根据程序上下文环境，Java关键字final有“这是无法改变的”或者“终态的”含义，它可以修饰非抽象类、非抽象类成员方法和变量。你可能出于两种理解而需要阻止改变：设计或效率。

final类不能被继承，没有子类，final类中的方法默认是final的。

final方法不能被子类的方法覆盖，但可以被继承。

final成员变量表示常量，只能被赋值一次，赋值后值不再改变。

final不能用于修饰构造方法。

注意：父类的private成员方法是不能被子类方法覆盖的，因此private类型的方法默认是final类型的。

1、final类

final类不能被继承，因此final类的成员方法没有机会被覆盖，默认都是final的。在设计类时候，如果这个类不需要有子类，类的实现细节不允许改变，并且确信这个类不会载被扩展，那么就设计为final类。

2、final方法

如果一个类不允许其子类覆盖某个方法，则可以把这个方法声明为final方法。

使用final方法的原因有二：

第一、把方法锁定，防止任何继承类修改它的意义和实现。

第二、高效。编译器在遇到调用final方法时候会转入内嵌机制，大大提高执行效率。

例如：

public class Test1 {

public static void main(String[] args) {

// TODO 自动生成方法存根

}

public void f1() {

System.out.println("f1");

}

//无法被子类覆盖的方法

public final void f2() {

System.out.println("f2");

}

public void f3() {

System.out.println("f3");

}

private void f4() {

System.out.println("f4");

}

}

public class Test2 extends Test1 {

public void f1(){

System.out.println("Test1父类方法f1被覆盖!");

}

public static void main(String[] args) {

Test2 t=new Test2();

t.f1();

t.f2(); //调用从父类继承过来的final方法

t.f3(); //调用从父类继承过来的方法

//t.f4(); //调用失败，无法从父类继承获得

}

}

3、final变量（常量）

用final修饰的成员变量表示常量，值一旦给定就无法改变！

final修饰的变量有三种：静态变量、实例变量和局部变量，分别表示三种类型的常量。

从下面的例子中可以看出，一旦给final变量初值后，值就不能再改变了。

另外，final变量定义的时候，可以先声明，而不给初值，这中变量也称为final空白，无论什么情况，编译器都确保空白final在使用之前必须被初始化。但是，final空白在final关键字final的使用上提供了更大的灵活性，为此，一个类中的final数据成员就可以实现依对象而有所不同，却有保持其恒定不变的特征。

package org.leizhimin;

public class Test3 {

private final String S = "final实例变量S";

private final int A = 100;

public final int B = 90;

public static final int C = 80;

private static final int D = 70;

public final int E; //final空白,必须在初始化对象的时候赋初值

public Test3(int x) {

E = x;

}

/\*\*

\* @param args

\*/

public static void main(String[] args) {

Test3 t = new Test3(2);

//t.A=101; //出错,final变量的值一旦给定就无法改变

//t.B=91; //出错,final变量的值一旦给定就无法改变

//t.C=81; //出错,final变量的值一旦给定就无法改变

//t.D=71; //出错,final变量的值一旦给定就无法改变

System.out.println(t.A);

System.out.println(t.B);

System.out.println(t.C); //不推荐用对象方式访问静态字段

System.out.println(t.D); //不推荐用对象方式访问静态字段

System.out.println(Test3.C);

System.out.println(Test3.D);

//System.out.println(Test3.E); //出错,因为E为final空白,依据不同对象值有所不同.

System.out.println(t.E);

Test3 t1 = new Test3(3);

System.out.println(t1.E); //final空白变量E依据对象的不同而不同

}

private void test() {

System.out.println(new Test3(1).A);

System.out.println(Test3.C);

System.out.println(Test3.D);

}

public void test2() {

final int a; //final空白,在需要的时候才赋值

final int b = 4; //局部常量--final用于局部变量的情形

final int c; //final空白,一直没有给赋值.

a = 3;

//a=4; 出错,已经给赋过值了.

//b=2; 出错,已经给赋过值了.

}

}

4、final参数

当函数参数为final类型时，你可以读取使用该参数，但是无法改变该参数的值。

public class Test4 {

public static void main(String[] args) {

new Test4().f1(2);

}

public void f1(final int i) {

//i++; //i是final类型的,值不允许改变的.

System.out.print(i);

}

}

二、static

static表示“全局”或者“静态”的意思，用来修饰成员变量和成员方法，也可以形成静态static代码块，但是Java语言中没有全局变量的概念。

被static修饰的成员变量和成员方法独立于该类的任何对象。也就是说，它不依赖类特定的实例，被类的所有实例共享。只要这个类被加载，Java虚拟机就能根据类名在运行时数据区的方法区内定找到他们。因此，static对象可以在它的任何对象创建之前访问，无需引用任何对象。

用public修饰的static成员变量和成员方法本质是全局变量和全局方法，当声明它类的对象市，不生成static变量的副本，而是类的所有实例共享同一个static变量。

static变量前可以有private修饰，表示这个变量可以在类的静态代码块中，或者类的其他静态成员方法中使用（当然也可以在非静态成员方法中使用--废话），但是不能在其他类中通过类名来直接引用，这一点很重要。实际上你需要搞明白，private是访问权限限定，static表示不要实例化就可以使用，这样就容易理解多了。static前面加上其它访问权限关键字的效果也以此类推。

static修饰的成员变量和成员方法习惯上称为静态变量和静态方法，可以直接通过类名来访问，访问语法为：

类名.静态方法名(参数列表...)

类名.静态变量名

用static修饰的代码块表示静态代码块，当Java虚拟机（JVM）加载类时，就会执行该代码块（用处非常大，呵呵）。

1、static变量

按照是否静态的对类成员变量进行分类可分两种：一种是被static修饰的变量，叫静态变量或类变量；另一种是没有被static修饰的变量，叫实例变量。两者的区别是：

对于静态变量在内存中只有一个拷贝（节省内存），JVM只为静态分配一次内存，在加载类的过程中完成静态变量的内存分配，可用类名直接访问（方便），当然也可以通过对象来访问（但是这是不推荐的）。

对于实例变量，没创建一个实例，就会为实例变量分配一次内存，实例变量可以在内存中有多个拷贝，互不影响（灵活）。

2、静态方法

静态方法可以直接通过类名调用，任何的实例也都可以调用，因此静态方法中不能用this和super关键字，不能直接访问所属类的实例变量和实例方法(就是不带static的成员变量和成员成员方法)，只能访问所属类的静态成员变量和成员方法。因为实例成员与特定的对象关联！这个需要去理解，想明白其中的道理，不是记忆！！！

因为static方法独立于任何实例，因此static方法必须被实现，而不能是抽象的abstract。

3、static代码块

static代码块也叫静态代码块，是在类中独立于类成员的static语句块，可以有多个，位置可以随便放，它不在任何的方法体内，JVM加载类时会执行这些静态的代码块，如果static代码块有多个，JVM将按照它们在类中出现的先后顺序依次执行它们，每个代码块只会被执行一次。例如：

public class Test5 {

private static int a;

private int b;

static {

Test5.a = 3;

System.out.println(a);

Test5 t = new Test5();

t.f();

t.b = 1000;

System.out.println(t.b);

}

static {

Test5.a = 4;

System.out.println(a);

}

public static void main(String[] args) {

// TODO 自动生成方法存根

}

static {

Test5.a = 5;

System.out.println(a);

}

public void f() {

System.out.println("hhahhahah");

}

}

运行结果：

3

hhahhahah

1000

4

5

利用静态代码块可以对一些static变量进行赋值，最后再看一眼这些例子，都一个static的main方法，这样JVM在运行main方法的时候可以直接调用而不用创建实例。

4、static和final一块用表示什么

static final用来修饰成员变量和成员方法，可简单理解为“全局常量”！

对于变量，表示一旦给值就不可修改，并且通过类名可以访问。

对于方法，表示不可覆盖，并且可以通过类名直接访问。

特别要注意一个问题：

对于被static和final修饰过的实例常量，实例本身不能再改变了，但对于一些容器类型（比如，ArrayList、HashMap）的实例变量，不可以改变容器变量本身，但可以修改容器中存放的对象，这一点在编程中用到很多。

# 直接引用和符号引用

JVM在装载class文件的时候，会有一步是将符号引用解析为直接引用的过程。

那么这里的直接引用到底是什么呢？

对于指向“类型”【Class对象】、类变量、类方法的直接引用可能是指向方法区的本地指针。

指向实例变量、实例方法的直接引用都是偏移量。实例变量的直接引用可能是从对象的映像开始算起到这个实例变量位置的偏移量。实例方法的直接引用可能是方法表的偏移量。

在《深入java虚拟机》书的第197页我们可以看到，子类中方法表的偏移量和父类中的方法表的偏移量是一致的。比如说父类中有一个say()方法的偏移量是7，那么子类中say方法的偏移量也是7。

书中第199页说，通过“接口引用”来调用一个方法，jvm必须搜索对象的类的方法表才能找到一个合适的方法。这是因为实现同一个接口的这些类中，不一定所有的接口中的方法在类方法区中的偏移量都是一样的。他们有可能会不一样。这样的话可能就要搜索方法表才能确认要调用的方法在哪里。

而通过“类引用”来调用一个方法的时候，直接通过偏移量就可以找到要调用的方法的位置了。【因为子类中的方法的偏移量跟父类中的偏移量是一致的】

所以，通过接口引用调用方法会比类引用慢一些。

下面介绍下什么是接口引用。

interface A{void say();}

class B implements A{}

class C{public static void main(String []s){A a=new B();a.say()}}

在上面的第三行代码中，就是用“接口引用”来调用方法。

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

符号引用：

符号引用是一个字符串，它给出了被引用的内容的名字并且可能会包含一些其他关于这个被引用项的信息——这些信息必须足以唯一的识别一个类、字段、方法。这样，对于其他类的符号引用必须给出类的全名。对于其他类的字段，必须给出类名、字段名以及字段描述符。对于其他类的方法的引用必须给出类名、方法名以及方法的描述符。

# 多继承的缺点

1. 如果在一个子类继承的多个父类中拥有相同名字的实例变量，子类在引用该变量时将产生歧义，无法判断应该使用哪个父类的变量。
2. 如果在一个子类继承的多个父类中拥有相同方法，子类中有没有覆盖该方法，那么调用该方法时将产生歧义，无法判断应该调用哪个父类的方法。

# 实现接口和继承父类

在接口中不能有实例变量，只能有静态的常量，不能有具体的方法（包含方法体），只能有抽象方法，因此也就摒弃了多继承的缺点。

对于一个类实现多个接口的情况，因为接口只有抽象方法，具体方法只能由实现接口的类实现，在调用的时候始终只会调用实现类的方法（不存在歧义），因此不存在 多继承的第二个缺点；而又因为接口只有静态的常量，但是由于静态变量是在编译期决定调用关系的，即使存在一定的冲突也会在编译时提示出错；而引用静态变量一般直接使用类名或接口名，从而避免产生歧义，因此也不存在多继承的第一个缺点。

对于一个接口继承多个父接口的情况也一样不存在这些缺点。

一个类可以实现多个接口，一个类只能继承一个类，一个接口可以继承多个接口

# 接口和抽象类

接口和抽象类显著的共同点是接口和抽象类都可以有抽象方法。

接口和抽象类的不同点有：

(1)抽象类可以有实例变量，而接口不能拥有实例变量，接口中的变量都是静态（static）的常量（final）。

(2)抽象类可以有非抽象方法，而接口只能有抽象方法。

# 常量

Java 常量，有2种意思，我分别说明：

第1种意思，就是一个值，这个值本身，我们可以叫它常量，举几个例子：

整型常量: 123

实型常量：3.14

字符常量: 'a'

逻辑常量：true、false

字符串常量："helloworld"

第2种意思，表示不可变的变量，这种也叫常量，从语法上来讲也就是，加上final，使用final关键字来修饰某个变量，然后只要赋值之后，就不能改变了，就不能再次被赋值了，举个例子：

final int i = 0;

那么这个i的值是绝对不能再被更改了，只能是0，所以说是 不可变的变量，这句话看似矛盾，其实不矛盾，这句话这样理解：

i就是一个int类型的变量，变量本身是可变的（可被更改值），但是现在加了final，所以不可变了，所以是不可变的变量。

# Java中main方法

在Java中，main()方法是Java应用程序的入口方法，也就是说，程序在运行的时候，第一个执行的方法就是main()方法，这个方法和其他的方法有很大的不同，比如方法的名字必须是main，方法必须是public static void 类型的，方法必须接收一个字符串数组的参数等等。

在看Java中的main()方法之前，先看一个最简单的Java应用程序HelloWorld，我将通过这个例子说明Java类中main()方法的奥秘，程序的代码如下：

/\*\*

\* Java中的main()方法详解

\*/

public class HelloWorld {

public static void main(String args[]) {

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

}

}

一、先说类：

HelloWorld 类中有main()方法，说明这是个java应用程序，通过JVM直接启动运行的程序。

既然是类，java允许类不加public关键字约束，当然类的定义只能限制为public或者无限制关键字（默认的）。

二、再说main()方法

这个main()方法的声明为：public static void main(String args[])。必须这么定义，这是Java的规范。

为什么要这么定义，和JVM的运行有关系。

当一个类中有main()方法，执行命令“java 类名”则会启动虚拟机执行该类中的main方法。

由于JVM在运行这个Java应用程序的时候，首先会调用main方法，调用时不实例化这个类的对象，而是通过类名直接调用因此需要是限制为public static。

对于java中的main方法，jvm有限制，不能有返回值，因此返回值类型为void。

main方法中还有一个输入参数，类型为String[]，这个也是java的规范，main()方法中必须有一个入参，类细必须String[]，至于字符串数组的名字，这个是可以自己设定的，根据习惯，这个字符串数组的名字一般和sun java规范范例中mian参数名保持一致，取名为args。

因此，main()方法定义必须是：“public static void main(String 字符串数组参数名[])”。

三、main()方法中可以throw Exception

因此main()方法中可以抛出异常，main()方法上也可以声明抛出异常。

比如，下面这个写法是正确的：

public class TestMain {

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

System.out.println("哈哈哈哈哈");

throw new Exception("");

}

}

运行结果：

哈哈哈哈哈

Exception in thread "main" java.lang.Exception:

at maintest.TestMain.main(TestMain.java:11)

at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)

at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)

at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)

at java.lang.reflect.Method.invoke(Method.java:585)

at com.intellij.rt.execution.application.AppMain.main(AppMain.java:90)

Process finished with exit code 1

四、main()方法中字符串参数数组作用

main()方法中字符串参数数组作用是接收命令行输入参数的，命令行的参数之间用空格隔开。

下面给出一个例子，看看如何初始化和使用这个数组的。

/\*\*

\* 打印main方法中的输入参数

\*/

public class TestMain {

public static void main(String args[]){

System.out.println("打印main方法中的输入参数！");

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

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

}

}

}

执行方法和运行结果

D:\Study\basetest\src>javac TestMain.java

D:\Study\basetest\src>java TestMain 1 2 3

打印main方法中的输入参数！

1

2

3

五、给出HelloWorld的另外一个版本

/\*\*

\* 变态版的HelloWorld.呵呵

\*/

public class HelloWorld2 {

static {

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

}

public static void main(String args[]){

System.exit(0);

}

}

这个main()方法执行的内容就一句"System.exit(0);" ，目的是让程序正常结束。那“HelloWorld！”是从哪里打印的，秘密就是在static打印的，因为static代码块的内容会在main调用前调用。

总结:

main方法作为一个特殊的规范，与普通的方法有很大区别，限制很多，理解其原理需要学习JVM相关知识。是Java中学习中大障碍。这是我对main原理和使用的总结，欢迎各位在此讨论。

# 静态方法不能调用非静态变量和方法

程序最终都将在内存中执行，变量只有在内存中占有一席之地时才能被访问。

类的静态成员(变量和方法)属于类本身，在类加载的时候就会分配内存，可以通过类名直接去访问；非静态成员（变量和方法）属于类的对象，所以只有在类的对象产生（创建类的实例）时才会分配内存，然后通过类的对象（实例）去访问。

在一个类的静态成员中去访问其非静态成员之所以会出错是因为在类的非静态成员不存在的时候类的静态成员就已经存在了，访问一个内存中不存在的东西当然会出错：

而类又是在什么时候加载的呢?

由引导类加载器负责加载的核心类比如 String 类在 JVM 启动时（main 方法开始执行前）就会被加载，其它类在使用前（new 它对象或调用其静态方法访问静态域等等前）会被动态加载，要注意的是子类被加载前它的所有超类要根据由父到子的顺序被逐一加载。

class A1

{

public static int a=5;

}

class B1 extends A1

{

public static int a=8;

void print(){

System.out.println(super.a);

System.out.println(a);

}

}

public class TestStatic

{

public static void main(String args[])

{

System.out.println("b.a="+B1.a);

System.out.println("b.a="+A1.a);

new B1().print();

}

}

就这段代码说一下.

当你在doc环境下输入：

java TestStatic时，虚拟机会先加载TestStatic类，此时虚拟机会先看看TestStatic类有没有静态的字段，没有，直接执行main方法。main方法中第一句代码是打印B1.a，虚拟机便会去找类B1，找到类B1时，虚拟机发现B1的父亲是A1，于是父亲优先，先加载A1，同样，在加载A1时，会看看A1中有什么静态的东西，有，static int a = 5；a是静态的，先加载，当把静态的字段加载完后，一个类就算加载完了，所以A1已经加载完毕，以后不用在加载了。父亲加载完了，轮到B1了，同样先看看里面有什么静态的字段，有，static int a = 8；此时B1也加载完毕了。第一条打印语句到此时也执行完毕了，轮到第二条打印语句了。当执行new B1().print();这句时，会发生动态绑定，此时会有一个代表B1对象的this对象传递给print（）方法，所以print（）方法中的System.out.println(a);其实是，System.out.println(this.a);会打印出一个8出来。至于super.a就简单了，打印父类中的a，结果是5.到此，main（）方法执行完，整个程序退出。

# String StringBuffer与StringBuilder的区别

String 字符串常量

StringBuffer 字符串变量（线程安全）

StringBuilder 字符串变量（非线程安全）

String 类型和 StringBuffer 类型的主要性能区别其实在于 String 是不可变的对象, 因此在每次对 String 类型进行改变的时候其实都等同于生成了一个新的 String 对象，然后将指针指向新的 String 对象，所以经常改变内容的字符串最好不要用 String ，因为每次生成对象都会对系统性能产生影响，特别当内存中无引用对象多了以后， JVM 的 GC 就会开始工作，那速度是一定会相当慢的。

而如果是使用 StringBuffer 类则结果就不一样了，每次结果都会对 StringBuffer 对象本身进行操作，而不是生成新的对象，再改变对象引用。所以在一般情况下我们推荐使用 StringBuffer ，特别是字符串对象经常改变的情况下。而在某些特别情况下， String 对象的字符串拼接其实是被 JVM 解释成了 StringBuffer 对象的拼接，所以这些时候 String 对象的速度并不会比 StringBuffer 对象慢，而特别是以下的字符串对象生成中， String 效率是远要比 StringBuffer 快的：

String S1 = “This is only a” + “ simple” + “ test”;

StringBuffer Sb = new StringBuilder(“This is only a”).append(“ simple”).append(“ test”);

你会很惊讶的发现，生成 String S1 对象的速度简直太快了，而这个时候 StringBuffer 居然速度上根本一点都不占优势。其实这是 JVM 的一个把戏，在 JVM 眼里，这个

String S1 = “This is only a” + “ simple” + “test”; 其实就是：

String S1 = “This is only a simple test”; 所以当然不需要太多的时间了。但大家这里要注意的是，如果你的字符串是来自另外的 String 对象的话，速度就没那么快了，譬如：

String S2 = “This is only a”;

String S3 = “ simple”;

String S4 = “ test”;

String S1 = S2 +S3 + S4;

这时候 JVM 会规规矩矩的按照原来的方式去做

在大部分情况下 StringBuffer > String

StringBuffer

Java.lang.StringBuffer线程安全的可变字符序列。一个类似于 String 的字符串缓冲区，但不能修改。虽然在任意时间点上它都包含某种特定的字符序列，但通过某些方法调用可以改变该序列的长度和内容。

可将字符串缓冲区安全地用于多个线程。可以在必要时对这些方法进行同步，因此任意特定实例上的所有操作就好像是以串行顺序发生的，该顺序与所涉及的每个线程进行的方法调用顺序一致。

StringBuffer 上的主要操作是 append 和 insert 方法，可重载这些方法，以接受任意类型的数据。每个方法都能有效地将给定的数据转换成字符串，然后将该字符串的字符追加或插入到字符串缓冲区中。append 方法始终将这些字符添加到缓冲区的末端；而 insert 方法则在指定的点添加字符。

例如，如果 z 引用一个当前内容是“start”的字符串缓冲区对象，则此方法调用 z.append("le") 会使字符串缓冲区包含“startle”，而 z.insert(4, "le") 将更改字符串缓冲区，使之包含“starlet”。

在大部分情况下 StringBuilder > StringBuffer

java.lang.StringBuilde

java.lang.StringBuilder一个可变的字符序列是5.0新增的。此类提供一个与 StringBuffer 兼容的 API，但不保证同步。该类被设计用作 StringBuffer 的一个简易替换，用在字符串缓冲区被单个线程使用的时候（这种情况很普遍）。如果可能，建议优先采用该类，因为在大多数实现中，它比 StringBuffer 要快。两者的方法基本相同。

package com.hct.test;

import java.util.ArrayList;

import java.util.Iterator;

import java.util.List;

/\*\*

\* @author: chengtai.he

\* @created:2009-12-9 上午09:59:57

\*/

public class StringBuilderTester {

private static final String base = " base string. ";

private static final int count = 2000000;

public static void stringTest() {

long begin, end;

begin = System.currentTimeMillis();

String test = new String(base);

for (int i = 0; i < count/100; i++) {

test = test + " add ";

}

end = System.currentTimeMillis();

System.out.println((end - begin)

+ " millis has elapsed when used String. ");

}

public static void stringBufferTest() {

long begin, end;

begin = System.currentTimeMillis();

StringBuffer test = new StringBuffer(base);

for (int i = 0; i < count; i++) {

test = test.append(" add ");

}

end = System.currentTimeMillis();

System.out.println((end - begin)

+ " millis has elapsed when used StringBuffer. ");

}

public static void stringBuilderTest() {

long begin, end;

begin = System.currentTimeMillis();

StringBuilder test = new StringBuilder(base);

for (int i = 0; i < count; i++) {

test = test.append(" add ");

}

end = System.currentTimeMillis();

System.out.println((end - begin)

+ " millis has elapsed when used StringBuilder. ");

}

public static String appendItemsToStringBuiler(List list) {

StringBuilder b = new StringBuilder();

for (Iterator i = list.iterator(); i.hasNext();) {

b.append(i.next()).append(" ");

}

return b.toString();

}

public static void addToStringBuilder() {

List list = new ArrayList();

list.add(" I ");

list.add(" play ");

list.add(" Bourgeois ");

list.add(" guitars ");

list.add(" and ");

list.add(" Huber ");

list.add(" banjos ");

System.out.println(StringBuilderTester.appendItemsToStirngBuffer(list));

}

public static String appendItemsToStirngBuffer(List list) {

StringBuffer b = new StringBuffer();

for (Iterator i = list.iterator(); i.hasNext();) {

b.append(i.next()).append(" ");

}

return b.toString();

}

public static void addToStringBuffer() {

List list = new ArrayList();

list.add(" I ");

list.add(" play ");

list.add(" Bourgeois ");

list.add(" guitars ");

list.add(" and ");

list.add(" Huber ");

list.add(" banjos ");

System.out.println(StringBuilderTester.appendItemsToStirngBuffer(list));

}

public static void main(String[] args) {

stringTest();

stringBufferTest();

stringBuilderTest();

addToStringBuffer();

addToStringBuilder();

}

}

上面的程序结果如下：

5266 millis has elapsed when used String.

375 millis has elapsed when used StringBuffer.

281 millis has elapsed when used StringBuilder.

I play Bourgeois guitars and Huber banjos

I play Bourgeois guitars and Huber banjos

从上面的结果来看，这三个类在单线程程序中的性能差别一目了然，采用String对象时，即使运行次数仅是采用其他对象的1/100，其执行时间仍然比其他对象高出25倍以上；而采用StringBuffer对象和采用StringBuilder对象的差别也比较明显，前者是后者的1.5倍左右。由此可见，如果我们的程序是在单线程下运行，或者是不必考虑到线程同步问题，我们应该优先使用StringBuilder类；当然，如果要保证线程安全，自然非StringBuffer莫属了。

除了对多线程的支持不一样外，这两个类的使用几乎没有任何差别，上面的例子就是个很好的说明。appendItemsToStringBuiler和appendItemsToStirngBuffer两个方法除了采用的对象分别为StringBuilder和StringBuffer外，其他完全相同，而效果也完全相同。

# break和continue

break语句可以强迫程序中断循环，当程序执行到break语句时，即会离开循环，继续执行循环外的下一个语句，如果Break语句出现在嵌套循环中的内层循环，则break语句只会跳出当前层的循环 switch语句中的case中的break是跳出整个switch语句

continue语句可以强迫程序跳到循环的起始处，当程序运行到continue语句时，即会停止运行剩余的循环主体，而是回到循环的开始处继续运行，记住，不是跳出整个循环执行下一条语句，这是Break和continue的主要区别所在，实际上使用continue就是中断一次循环的执行

# 读取配置文件

JAVA里面对于类进行调用配置资源的文件数据，以this.getClass().getResourceAsStream()来读取比较合适。

路径采用相对路径直接可以从工程的path路径去找。

主要问题是如果类中采用的是静态块的话，则该this.getClass()报错，因为读静态块时，可能该对象并为构造，所以用this来指向当前对象不行。

ClassLoader提供了两个方法用于从装载的类路径中取得资源：

public URL getResource(String name);

public InputStream getResourceAsStream(String name);

参考资料：<http://blog.csdn.net/mhmyqn/article/details/7683909>

这里name是资源的类路径，它是相对与“/”根路径下的位置。getResource得到的是一个URL对象来定位资源，而getResourceAsStream取得该资源输入流的引用保证程序可以从正确的位置抽取数据。

然而，程序中调用的通常并不是ClassLoader的这两个方法，而是Class的getResource和getResourceAsStream方法，因为Class对象可以从你的类得到（如YourClass.class或YourClass.getClass()），而ClassLoader则需要再调用一次YourClass.getClassLoader()方法，但根据JDK文档的说法，Class对象的这两个方法其实是“委托”（delegate）给装载它的ClassLoader来做的，所以只需要使用Class对象的这两个方法就可以了。

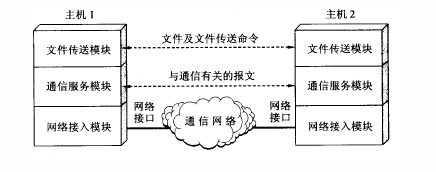
举例说明：

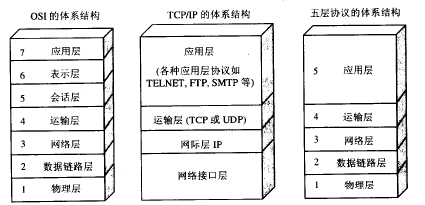
mypackage.Hello.class.getResourceAsStream("/config/config.ini");

从classpath下的config相对路径中读取config.ini

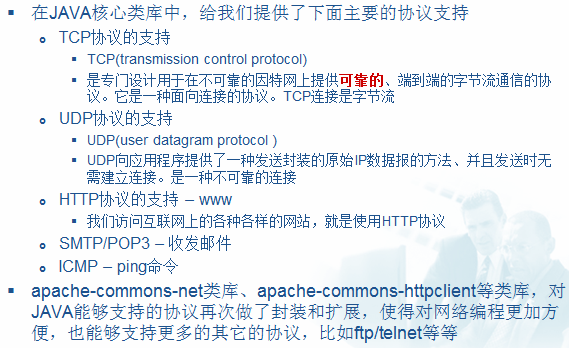
# 网络编程

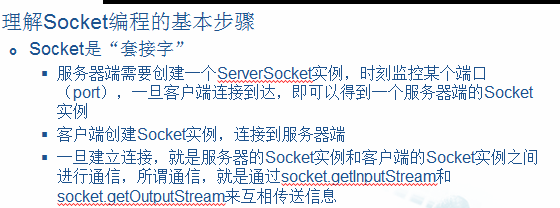
**分层**

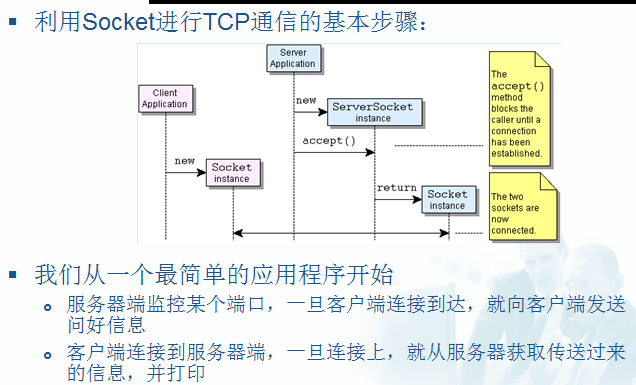




**TCP　UDP**







# Session内部机制

session机制是一种服务器端的机制，服务器使用一种类似于散列表的结构（也可能就是使用散列表）来保存信息。

当程序需要为某个客户端的请求创建一个session的时候，服务器首先检查这个客户端的请求里是否已包含了一个session标识 - 称为 session id，如果已包含一个session id则说明以前已经为此客户端创建过session，服务器就按照session id把这个 session检索出来使用（如果检索不到，可能会新建一个），如果客户端请求不包含session id，则为此客户端创建一个session并且生成一个与此session相关联的session id，session id的值应该是一个既不会重复，又不容易被找到规律以仿造的字符串，这个 session id将被在本次响应中返回给客户端保存。

保存这个session id的方式可以采用cookie，这样在交互过程中浏览器可以自动的按照规则把这个标识发挥给服务器。一般这个cookie的名字都是类似于SEEESIONID，而。比如weblogic对于web应用程序生成的cookie，JSESSIONID= ByOK3vjFD75aPnrF7C2HmdnV6QZcEbzWoWiBYEnLerjQ99zWpBng!-145788764，它的名字就是 JSESSIONID。

由于cookie可以被人为的禁止，必须有其他机制以便在cookie被禁止时仍然能够把session id传递回服务器。经常被使用的一种技术叫做URL重写，就是把session id直接附加在URL路径的后面，附加方式也有两种，一种是作为URL路径的附加信息，表现形式为http://...../xxx;jsessionid= ByOK3vjFD75aPnrF7C2HmdnV6QZcEbzWoWiBYEnLerjQ99zWpBng!-145788764

另一种是作为查询字符串附加在URL后面，表现形式为http://...../xxx?jsessionid=ByOK3vjFD75aPnrF7C2HmdnV6QZcEbzWoWiBYEnLerjQ99zWpBng!-145788764

这两种方式对于用户来说是没有区别的，只是服务器在解析的时候处理的方式不同，采用第一种方式也有利于把session id的信息和正常程序参数区分开来。

为了在整个交互过程中始终保持状态，就必须在每个客户端可能请求的路径后面都包含这个session id。

<http://blog.csdn.net/yinyuan1987/article/details/3321213>

浏览器处理session的机制是在URL的参数中隐藏的传递一个sessionid的参数来表示客户端究竟对应服务器的哪个session,从一个窗口打开的新的窗口都会继承这个参数.而新建的窗口则没有这个参数,所以新建的窗口使用的是新的session.

你把SESSION理解成一个MAP对象就行了，其实很多框架，比如STRUTS2就把容器的SESSION封装成了一个MAP对象  
一个用户对应一个KEY，他是唯一的  
你只需要根据KEY验证具体的用户是否存在就好啦

# ArrayList和LinkedList的区别

ArrayList 采用的是数组形式来保存对象的，这种方式将对象放在连续的位置中，所以最大的缺点就是插入删除时非常麻烦

LinkedList 采用的将对象存放在独立的空间中，而且在每个空间中还保存下一个链接的索引但是缺点就是查找非常麻烦要丛第一个索引开始

Hashtable和HashMap类有三个重要的不同之处。第一个不同主要是历史原因。Hashtable是基于陈旧的Dictionary类的，HashMap是Java 1.2引进的Map接口的一个实现。

也许最重要的不同是Hashtable的方法是同步的，而HashMap的方法不是。这就意味着，虽然你可以不用采取任何特殊的行为就可以在一个多线程的应用程序中用一个Hashtable，但你必须同样地为一个HashMap提供外同步。一个方便的方法就是利用Collections类的静态的synchronizedMap()方法，它创建一个线程安全的Map对象，并把它作为一个封装的对象来返回。这个对象的方法可以让你同步访问潜在的HashMap。这么做的结果就是当你不需要同步时，你不能切断Hashtable中的同步（比如在一个单线程的应用程序中），而且同步增加了很多处理费用。

第三点不同是，只有HashMap可以让你将空值作为一个表的条目的key或value。HashMap中只有一条记录可以是一个空的key，但任意数量的条目可以是空的value。这就是说，如果在表中没有发现搜索键，或者如果发现了搜索键，但它是一个空的值，那么get()将返回null。如果有必要，用containKey()方法来区别这两种情况。

一些资料建议，当需要同步时，用Hashtable，反之用HashMap。但是，因为在需要时，HashMap可以被同步，HashMap的功能比Hashtable的功能更多，而且它不是基于一个陈旧的类的，所以有人认为，在各种情况下，HashMap都优先于Hashtable。

关于Properties

有时侯，你可能想用一个hashtable来映射key的字符串到value的字符串。DOS、Windows和Unix中的环境字符串就有一些例子，如key的字符串PATH被映射到value的字符串C:\WINDOWS;C:\WINDOWS\SYSTEM。Hashtables是表示这些的一个简单的方法，但Java提供了另外一种方法。

Java.util.Properties类是Hashtable的一个子类，设计用于String keys和values。Properties对象的用法同Hashtable的用法相象，但是类增加了两个节省时间的方法，你应该知道。

Store()方法把一个Properties对象的内容以一种可读的形式保存到一个文件中。Load()方法正好相反，用来读取文件，并设定Properties对象来包含keys和values。

注意，因为Properties扩展了Hashtable，你可以用超类的put()方法来添加不是String对象的keys和values。这是不可取的。另外，如果你将store()用于一个不包含String对象的Properties对象，store()将失败。作为put()和get()的替代，你应该用setProperty()和getProperty()，它们用String参数。