

Java基础教程

作者: JAVE_LOVER <http://jave-lover.iteye.com>

本教程为想学习Java语言的朋友提供帮助，让Java小白轻松掌握

目 录

1. Java学习系列

1.1 Java学习系列(一)Java的运行机制、JDK的安装配置及常用命令详解 3

1.2 Java学习系列(二)Java注释、标识符、基本数据类型及其转换易错点详解 5

1.3 Java学习系列(三)Java运算符、控制语句、数组及其在内存中的运行分析 9

1.4 Java学习系列(四)Java面向对象之修饰符、封装、继承、多态详解 15

1.5 Java学习系列(五)Java面向对象之抽象类、接口、内部类、枚举类详解 22

1.6 Java学习系列(六)Java面向对象之Jar命令、正则表达式、国际化详解 33

1.7 Java学习系列(七)Java面向对象之集合框架详解(上) 38

1.8 Java学习系列(八)Java面向对象之集合框架详解(下) 48

1.9 Java学习系列(九)Java面向对象之异常机制详解 51

1.10 Java学习系列(十)Java面向对象之I/O流(上) 55

1.11 Java学习系列(十一)Java面向对象之I/O流(下) 64

1.12 Java学习系列(十二)Java面向对象之序列化机制及版本 70

1.13 Java学习系列(十三)Java面向对象之界面编程 76

1.14 Java学习系列(十四)Java面向对象之细谈线程、线程通信(上) 84

1.15 Java学习系列(十五)Java面向对象之细谈线程、线程通信(下) 89

1.16 Java学习系列(十六)Java面向对象之基于TCP协议的网络通信 94

1.17 Java学习系列(十七)Java面向对象之开发聊天工具 100

1.18 Java学习系列(十八)Java面向对象之基于UDP协议的网络通信 107

1.19 Java学习系列(十九)Java面向对象之数据库编程 110

1.1 Java学习系列(一)Java的运行机制、JDK的安装配置及常用命令详解

发表时间: 2013-09-21 关键字: java, jdk, 虚拟机, 编程, jvm

俗话说：“十五的月亮十六圆”。那学习是不是也是如此呢？如果把月亮看成是我们的愿望，那十五便是我们所处的“高原期”，坚持迈过这个坎，我相信你的愿望终究会现实的。记得马云曾说：今天很残酷，明天更残酷，后天很好，但绝大部分人是死在明天晚上，所以每个人不要放弃今天。是的，我们不应该放弃今天，因为每个脚印都值得期待，每一次机会都值得尝试。不扯了，还是步入正题吧！

Java的运行机制：Java源程序经过编译器编译成平台无关的字节码，字节码由虚拟机解释执行，虚拟机将每一条要执行的字节码发送给特定平台的解释器，解释器将其翻译成相应平台上的机器码，然后运行在该平台上，又由于运行时依然保留了解释这样就保证了Java程序能跨平台了。Java是一种介于编译型语言与解释型语言之间的一门强类型的语言，所谓编译型语言，像c/c++等运行时只需编译，无需解释，而且只能在特定运行在特定平台上；而解释型语言就是每次运行时，经过解释器负责翻译成特定平台的机器指令，每次运行时，都需要经过一次解释。所以说Java的跨平台型主要得益于它的虚拟机。

JDK(Java Development Kit)开发工具包)的安装配置：

安装JDK时，无需安装公共JRE (Java Runtime Environment，由JVM+核心库组成)，因为JDK默认包含了JRE；其它步骤直接点击下一步即可。

JDK的配置：1、把JDK的bin目录添加到path环境变量中，这样就可以保证用户无需输入绝对路径，即可使用javac、java两个工具。配置PATH的作用：让操作系统能找到我们所使用的命令，与java本身无关！2、添加系统变量JAVA_HOME，并在JAVA_HOME添加JDK安装路径，比如D:\Program Files\Java\jdk1.5.0，具体要根据jdk1.5的安装路径来定。配置JAVA_HOME变量的作用：JAVA_HOME路径下包括lib，bin，jre等文件夹，以后配置tomcat，eclipse等java开发软件可能需要依赖此变量。3、添加系统变量CLASSPATH，并在CLASSPATH变量下添加如下路径：%JAVA_HOME%\lib;%JAVA_HOME%\lib\tools.jar (注意前面的“.”)配置CLASSPATH变量的作用：为系统指明java加载类(class or lib)的路径，只有类在classpath中，java命令才能识别。

配置成功的效果如下：

```
C:\Documents and Settings\Administrator>java -version
java version "1.6.0_21"
Java(TM) SE Runtime Environment (build 1.6.0_21-b06)
Java HotSpot(TM) Client VM (build 17.0-b16, mixed mode, sharing)
```

下面以一个著名程序：HelloWorld结束本章内容，然后谈下编译和运行java的两个常用命令javac、java

```
class _你好世界
{
```

```
public static void main(String[] args)
{
    System.out.println("Hello World!");
}
}
```

效果如图（其中的HelloWorld.java为java源文件名，这里的点号指的是当前目录，当然你也可以换成其它目录，注意命令参数与文件名之间要有空格，java区分大小写，而源文件不区分大小写，下面两图效果可以看到是一样的）：

```
F:\>javac -d . HelloWorld.java
```

```
F:\>java -cp . _你好世界
Hello World!
```

```
F:\>javac -d . helloWorld.java
```

```
F:\>java -cp . _你好世界
Hello World!
```

命令解释：

编译源文件：javac -d [目录] java源文件 用于将生成的二进制文件放在指定目录下。

运行class文件：java -cp [目录] 类名 指定JVM到哪个目录下去搜索Java类。每个class对应一个类。

如果Java源文件中有public类，该源文件的主文件名必须和public类名相同。

如果Java源文件中没有public类，该源文件的主文件名可以是任意的。

关于java中类的修饰符，后面讲到面向对象的时候我会详细讲解。

好啦，今天就写到这，先把HelloWorld跑起来吧！

结束语：java的常用命令还有很多，具体会在后面陆续讲到。对于编程，个人觉得只要你肯多练，不断去调试和思考其脉络，再去慢慢的研究其细节，我想Java对你来说将不会很难。好啦，今天就写到这，明天开始讲ava的基本类型，内容可能会比较多，但是我会尽量讲的详细一些。

1.2 Java学习系列(二)Java注释、标识符、基本数据类型及其转换易错点详解

发表时间: 2013-09-21 关键字: java, 编程, JVM

今天看到一则小笑话，分享给大家。一对情侣甜蜜的在公园中依偎着，男的看到女的头发如此柔顺，便忍不住偷摸了一下，女的娇滴滴的说：“唉呀！讨厌啦！”

男的听了心更痒，于是又偷摸了一下，女的又说：“嗯，不要啦！”男的一听，心都要飞起来了，又再摸了一下，突然那女的站起来，粗暴的说道：“不要摸了！我的假发都快掉了！！！”🤔

Java注释

前面我们讲了下Java的运行机制及JDK的安装配置等相关操作，今天我们先从Java的注释开始。Java注释对程序本身没有影响，主要是开发者提供一些辅助信息来更好的理解。首先，Java的注释分三种：单行、多行以及文档注释。单行：`//`后面的内容就是单行注释；多行：`/*` 中间部分 就是多行注释 `*/`；至于文档注释，我们以一个简单的Java小程序为例。

```
/**
 * 此处为文档注释，我们可以用
 * javadoc命令直接提取文档注释，
 * 并根据文档注释来生成API文档
 */
public class $月饼
{
    //此处为单行注释。主方法(程序入口)
    public static void main(String[] args)
    {
        System.out.println("五仁月饼味道真心不错！");
    }

    /*
     *
     */
    public void info(){
        System.out.println("此方法被多行注释了。。");
    }
}
```

```
F:\>javac -d . $月饼.java
F:\>java -cp . $月饼
五仁月饼味道真心不错!
```

此时如果我们想用javadoc来帮我们生成API文档，命令及运行效果如下图：

```
F:\>javadoc -d myFirstAPI $月饼.java
正在装入源文件 $月饼.java...
正在构造 Javadoc 信息...
标准 Doclet 版本 1.6.0_21
正在构建所有软件包和类的树...
正在生成 myFirstAPI\月饼.html...
正在生成 myFirstAPI\package-frame.html...
正在生成 myFirstAPI\package-summary.html...
正在生成 myFirstAPI\package-tree.html...
正在生成 myFirstAPI\constant-values.html...
正在构建所有软件包和类的索引...
正在生成 myFirstAPI\overview-tree.html...
正在生成 myFirstAPI\index-all.html...
正在生成 myFirstAPI\deprecated-list.html...
正在构建所有类的索引...
正在生成 myFirstAPI\allclasses-frame.html...
正在生成 myFirstAPI\allclasses-noframe.html...
正在生成 myFirstAPI\index.html...
正在生成 myFirstAPI\help-doc.html...
正在生成 myFirstAPI\stylesheet.css...
```

打开我们的myFirstAPI目录下的index.html你会发现javadoc这个命令的强大所在。

Java标识符

标识符(合法的名字)：类名、方法名等。可以以中文、美元\$开头，但不能以数字开头，且不能是Java的关键字和保留字(都必须小写)。直接量：true,false,null。共53个关键字，其中有2个保留字：const、goto，它们在Java中目前没有被使用，因此不具有意义。上面的类名写成\$月饼 是可以编译通过的，但是一般不这么写，我们的类名应尽量有意义，当别人看的时候能一目了然。

基本数据类型

Java语言是一门强类型语言，所有变量(计算机内存里的数据就需要通过变量来访问它，变量就是一块内存的访问方式)必须先声明类型，再使用。Java有8种基本类型：

整数取值范围：-128~127 -32768~32767 -2G~2G-1 -2的63次方~2的63次方-1

4个整型（不支持小数）：byte(1字节)、short(2字节)、int(4字节)、long(8字节)

2个浮点型（支持小数）：float(4字节)、double(8字节)

1个字符型：char(2字节)

1个布尔型：boolean(1字节) 只能有两种取值：true、false

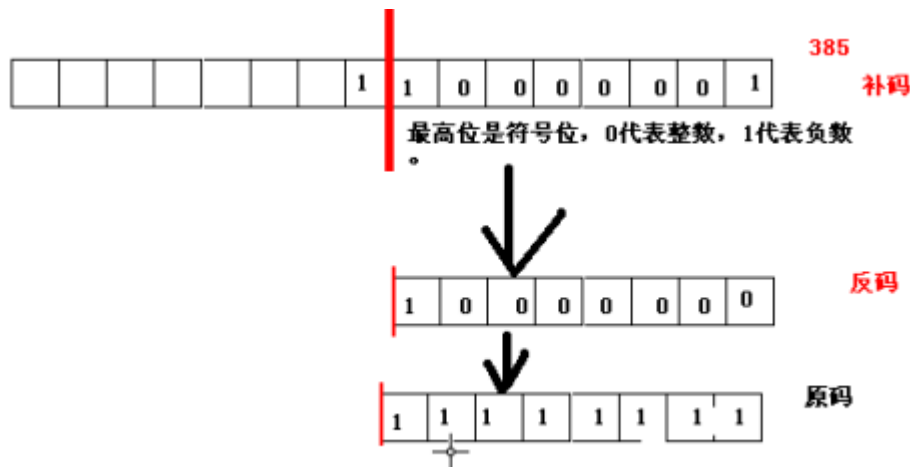
当我们直接使用一个整数时，整数默认是int类型。如果想使用一个long型，应该在正数后加L或l。Java的整型常数有三种形式：1.十进制整数,如123,-456,0。2.八进制整数,以0开头,如0123表示十进制数83,-011表示十进制数-9。3.十六进制整数,以0x或0X开头,如0x123表示十进制数291,-0X12表示十进制数-18。当我们直接使用浮点数时，浮点数默认是double类型。如果想使用一个float型，应该在正数后加F或f，如：浮点数表示：.512f(整数0部分可以省略，10十进制表示)；0.12e4f(0.12乘以10的4次方，科学计数法)。实际编程时，尽量使用double(范围大，精度高)。字符型：每个字符型的变量只能装一个字符。可以是英文或中文。字符型的表示方式如：1. 'q'、'中'；转义字符：'\n'、'\t'、'\r'、'\b'等。所以我们在给变量赋值时，一定要注意变量的取值范围。需要补充的是：所有的正无穷大和负无穷大都分别相等，非数(NaN)就是用0.0/0，非数连自己都不相等。

实例说明：

1、byte b = 385;//由于变量b超过了byte类型的取值范围，所以此时编译器此时会报错(可能损失精度)。----由于byte只占一个字节，所以要注意值溢出的问题。

2、如果我们在385这个数值进行强制类型(下面会讲到)，结果会是多少呢？byte b = (byte)385;思考一下。

计算过程如下：



我们知道数据的存储要遵循两条原则：1.最高位是符号位。最高位是0代表整数，最高位是1代表负数。2.所有的数值在计算机中都是以补码(在计算机内，有符号数有3种表示法：原码、反码和补码。原码：直接换算出的二进制码。反码：负数时除符号位不变之外，其它位都取反；正数反码=原码。补码：负数补码=反码+1；正数补码=原码)的形式保存的。在控制台输出给我们的是在转换成原码后再计算返回给我们的十进制数值。我们可以看到转换成的原码后7位1，符号位(最高位)也为1，所以我们可以算出结果为-127。

类型转换：byte-->short-->int-->long-->float-->double (---> 表示类型之间可以转换，由于char可以转化为int类型(也就是说字符型变量，可直接作为“正整数”使用)，所以这7种基本类型之间是相互可以转化的)，还有一个布尔型，由于它只支持两种取值(true/false)，所以数值类型只有7种。在数据类型进行转换时要注意：范围小的可以自动转

化为范围大的。范围大的可以强制转化(可能丢失精度)为范围小的。如果直接写一个整数，默认是int型，但如果它的范围在byte、short表示范围内，而且程序【直接将该值】(直接在源代码中指定的值)赋给byte、short类型变量，系统会自动强转。

表达式类型的自动提升

规则：整个表达式的数据类型，与表达式中最高等级的运算数的类型相同。int it=10/4;//这里最高为int,所以整个表达式的数据类型为int，it结果为2。但如果写成：int i = 10/0.2;就会报错，因为等式右边的最高等级的运算数的类型为double，而左边类型为int，所以编译器此时会报“可能损失精度”的错误。--以上部分内容来源于网路。

结束语

今天的内容就到这了，明天开始讲运算符、控制语句、数组等。

1.3 Java学习系列(三)Java运算符、控制语句、数组及其在内存中的运行分析

发表时间: 2013-09-22 关键字: java, 读书, 编程

梭罗说：“从圆到圆心有多少条半径，人们的生活方式也应该有这么多。”同样学习也是如此，学习的渠道有很多种，但能找适合比较适合自己的却不是一件简单的事。比如说有的人喜欢看书，从书中学到一些自己感兴趣的东西。当然有时也迫于无赖，必须得学一些比较枯燥的理论，这就造成了有些人喜欢读书，有些人对读书比较厌倦，大学里的这种现象比较常见。记得中国达人秀的舞台有一个叫卓君的街舞达人，他是通过上网看视频自学的街舞。街舞跳的那么好，悟性是一方面，但坚持练习肯定必不可少。虽然本人没有学过街舞，但是我觉得编程和街舞还是有相似之处的，每一个动作好比是一个方法或者称函数，那么连贯起来的一整套动作便是我们的程序。尽管动作有好有坏，但只要不断去练习改进(调试、调优)，相信你离成功不远了。

运算符

算术运算符：+,-,*,/,%,++,--。对于++ -- 只能操作一个变量。++既可以放在变量前(先把变量+1，然后再拿来用)，也可以放在变量后(先拿来用，再把变量+1)，--也是也是如此。这是比较容易出错的地方。

赋值运算符：【= += -= *= /= %= >>= <<= &= |= ^=】赋值运算符优先级别低于其他的运算符，所以对该运算符往往最后读取。注意等号左边只能是变量，赋值运算符可以与所有的双目运算符结合，就形成了扩展后的赋值运算符。需要注意的：a+=b相当于a=(a的类型)(a+b);//隐式类型转换。注意+=之间不能有空格。赋值表达式的值就是等号右边被赋的值。

位运算符(基于2进制码运算)：&、|、~、^(按位异或)、<<、>>、>>>。注意运算时要用补码去算，最终结果要转换为原码。>>：当右移除不尽的情况下，实际得到的整数总是比实际运算的结果值略小。>>>：无符号右移，在左边补0。^：同0异1。

关系运算符：等于运算符【=】、不等于运算符【!=】、关系运算符【< > <= >=】

按运算所需变量的个数来分：运算所需变量为一个的运算符叫单目运算符,例如【!,~,()】；运算所需变量为两个的运算符叫做双目运算符；算所需变量为三个的运算符叫三目运算符：

【?:】

逻辑运算符(操作数要求只能是boolean值)：&&、&(不会短路)、||、|(不会短路)、!、^(只有当两个操作数不同时，才返回true)。【自加的陷阱】：int c=1;c=c++;//a).先拿出c来值为1，b).c自加成2，c).把刚取出的1赋给c，c又变回1。

流程控制语句

流程控制语句一共有3种结构：1.顺序结构 如果没有流程控制，计算机总是从上到下，一次执行每一行。2.分支 满足某个条件时，才去执行代码。3.循环 重复执行一段代码。

分支控制：if/switch 如果省略了花括号，if条件只控制到第一个分号前面。else本身就是条件(在if条件上取反)。建议：即使条件执行体只有一行代码，也保留花括号。当有多个else if语句块时，建议先处理范围小的条件。switch：当等于某个值时，才执行一段代码。每次写完case,在写其它代码之前立即写break; 循环控制：while/do..while/forwhile

```
do{  
  
//do something...  
  
}while(返回boolean值的表达式)
```

这个循环与while相比，把循环放前面，先执行一次循环体，如果条件为true，将执行下一次循环。while使用规则：如果省略了花括号，while条件只控制到第一个分号前面。注意只要是循环体，就永远跳到循环条件。

举例说明：

```
public class LoopTest  
{  
    public static void main(String[] args)  
    {  
        int i = 1;  
        while(i++<5);  
        {  
            System.out.println("i="+i);  
        }  
    }  
}
```

```
}
```

分析一下，看看结果是不是i=6。

for(初始化语句;返回boolean值得表达式;每次循环体执行完后执行的代码)

```
{
```

```
//do something...
```

```
}
```

初始化语句：可以省略；如果有“初始化语句”，则只有在循环开始之前，会执行一次“初始化语句”。返回boolean值得表达式：可以省略；如果省略，意味着它的值永远是true。

每次循环体执行完后执行的代码：可以省略。如果有，将在每次循环体执行之后，开始下一次循环之前执行。【注意】只要两个分号中间的表达式返回true，程序将执行循环体。循环嵌套：把内存循环当成外层循环的一条语句即可。控制循环的关键词：break(完全结束一个循环本身)、continue(停止当前循环体的执行【忽略continue后面的语句】，开始下一次循环体)、return(结束整个方法)。

【注意】break、continue后面可以跟一个标号。带标号的break用于结束标号所标识的循环。带标号的continue忽略标号所标识的循环后面的剩下语句。另外注意Java里面没有goto(当做保留字)。

for循环的执行机制如图

```
for(int i=1;i < 10;i+=2){  
    System.out.println("i的值为: "+ i);  
}
```

再举一例：

```
public class LoopTest  
{  
    public static void main(String[] args)  
    {  
        for(int i=1;i < 10;i+=2){
```

```
        System.out.println("i的值为：" + (i*=1.2));  
    }  
}  
}
```

思考一下：运行结果会是多少呢？这里需要注意的是：这里有个隐式类型转换，原型为 $a*=b$ 相当于 $a=(a\text{的类型})(a*b)$ ；这样我想不难得出结果为：1、3、6、9。

还有一种foreach循环：如使用foreach循环对数组进行遍历

for(元素类型 变量名:数组|集合)

```
{  
    //此处即可通过“变量名”依次访问每个数组|集合中的元素。  
}
```

/*注意：在foreach中对循环变量赋值并不会改变原数组(下面会讲到)元素值*/

数组

数组类型(引用类型)（说明：如`int[] a`则变量`a`的数据类型为`int[]`，注意是`int[]`，而不是`int`）。数组里的每个元素，就相当于一个普通的变量。这里补充一下，引用类型的本质其实就是指针，只是Java对它做了一些封装和异常等检查。定义数组时，不能指定数组的长度。

【数组的初始化】：数组变量只是一个引用，必须让它指向有效的内存之后才能使用。数组的初始化分为两种：1.静态初始化：`int[] a = new int[]{1,2,3}` //只指定数组的元素，让系统来决定数组的长度,推荐使用这种。还有一种写法：`int[] a={1,2,3}`; //直接指定数组元素。2,动态初始化：`int[] a=new int[3]`;//只指定数组的长度，让系统来决定数组的元素值，注意如果数组元素是基本类型，那么所有数组元素的默认值都是0/0.0/false/0000；如果数组元素是引用类型，那么所有数组元素的默认值都是null。

方法栈(所有在方法中声明的变量，都放在相应的方法栈中)

每个方法运行时，系统都会为之建立一个方法栈。栈内存是临时性的内存，方法结束时，方法栈会被立即释放，所以栈不适合存放长有效的数据。java中的所有的对象都放在"堆"内存中，每个JVM只有一个"堆"内存。java不允许直接操作堆内存，只能通过引用来访问，只要JVM不退出，堆内存一直存在。这里补充一下：JVM有一条后台进程：垃圾回收器，它会用

一种机制记录 堆内存中“每个对象”是否有引用变量(指针)引用它。如果有，垃圾回收器就不会管它；如果没有，垃圾回收器就会在合适的时候去回收该对象所占的内存。注意：当数组对象的引用变量被销毁之后，数组并不一定会回收(它在堆内存)，它不会随着数组变量被回收，数组变量只是一个引用，它只是暂时指向了某块堆内存的地址而已。再说一下二维数组：Java允许初始化数组时只初始化左边的维数(如：int[][] a = new int[5][];)。二维数组的元素(引用变量,必须要有指向)是一维数组。

举例说明：

```
public class BinaryArrayTest
{
    public static void main(String[] args)
    {
        //定义二维数组a
        int[][] a;
        //动态初始化二维数组a
        a= new int[5][];
        //静态初始化一维数组a[1]
        a[1]=new int[]{1,2,3,4,5,6,7,8,9,10};
        for(int i=0;i <10;i++){
            System.out.print(a[1][i]+"\\t");
        }
    }
}
```

运行结果如下：

结束语

今天内容比较多，也有很多我们平常做题时容易出错的地方，大家以后一定要注意一下。以上部分内容来源于网络。

明天开始正式开始学习Java面向对象中的三大特征：封装、继承、多态。

1.4 Java学习系列(四)Java面向对象之修饰符、封装、继承、多态详解

发表时间: 2013-09-23 关键字: java, jdk, 编程

今天内容比较多，直接步入正题吧。

类和对象的定义

类是现实世界或思维世界中的实体在计算机中的反映，它将数据以及这些数据上的操作封装在一起。而对象是具有类类型的变量，存在于堆内存中。类是对象的抽象，而对象是类的具体实例。类是抽象的，不占用内存，而对象是具体的，占用存储空间。

定义一个类语法格式

[修饰符] class 类名

```
{  
//可以有属性、方法、构造方法、初始化块、枚举类、内部类等。。  
}
```

注意：类的修饰符可以省略(必须在同一个包里，下面会讲到)，还可以是：public，final(不能被继承)，abstract(不能创建实例，但增加了一个可以包含抽象方法的功能)。类名必须是多个有意义的单词(首字母大写)连缀。注意修饰符之间，没有先后顺序。

属性定义的语法格式： [修饰符] 类型 field名字 [=默认值];

[修饰符] --可以省略。 还可以是public|protected|private,static,final。类型 --不能省略。可以是基本类型，数组，任意类(JDK的类和自定义的类)，接口，枚举。field名字 --不能省略。必须是多个有意义的单词(第一个首字母小写)。[=默认值] --可以省略。

方法定义的语法格式：

```
[修饰符] 返回值类型 方法名(多个形参声明){  
//0~N条可执行语句  
//如果方法签名中有返回值类型声明，则该方法里必须包含【有效的】return语句  
}
```

[修饰符] --可以省略。还可以是public|protected|private,static,final|abstract。返回值类型 --不能省略。---编译时，系统认为if里面的return是有可能不被执行的(不算有效的)。方法名 --不能省略。--一般是动词，第一个首字母小写，后面首字母大写。形参声明 --每个形参必须满足“类型” “变量” 的格式。注意：Java方法的参数传递机制只有一种：值传

递。补充一下：Java中有种形参个数可变的方法，如public void info(int... nums)：这里的nums是一个int类型数组。(调用时可不传参)每个方法最多只能有一个形参个数可变的参数(本质是数组)。形参个数可变的参数必须放在形参位置的最后。

构造器(系统默认提供，可重载)定义的语法格式：

[修饰符] 构造器名(0~N个形参声明)

```
{  
//0~N条可执行语句  
}
```

[修饰符] --可以省略。还可以是public|protected|private。构造器名 --不能省略，必须与类名相同。构造器可认为是一种特殊的方法，作用是产生对象(通过构造器初始化类的实例)。注意前一章讲的赋值属于可执行语句，要放在方法或构造器中。

初始化块(没有名字)的语法：

```
[修饰符]{  
//可执行语句  
}
```

修饰符 只能出现static。有static说明它是类初始化块。实例初始化块(没有修饰符)的代码，会在每次调用构造器之前被“隐式”执行。规则：位于构造器之前，而且无需传参的代码可被提取到初始化块中，所以它的优先级比构造器更高，编译时实例初始化块的代码都会被提取到构造器的“最前面”。类(static)初始化：当类被加载后，对类初始化后时被隐式执行。一个JVM对一个类只初始化一次。当程序第一次【主动】使用该类就会初始化该类，下面几种情况会初始化该类。

- A.访问了该类静态field或静态方法。
- B.初始化了该类的子类--因为Java初始化一个类，永远先从最顶层父类开始初始化。
- C.使用反射来Class.forName(类名字符串)
- D.如果该类作为主类使用(有main方法，而且程序确实从main方法开始执行)
- E.使用该类来创建对象。

装箱和拆箱：

自动装箱：基本类型的值，可以自动当成它的包装类实例使用。它可以自动当成对象使用，也可以作为对象传入方法。自动拆箱：当有需要时，程序会自动把包装类的实例拆成基本类型的值。

局部变量

局部变量可以是方法里的局部变量、形参、代码块里面的局部变量(只在花括号中有效)。方法中的局部变量属于方法，修饰符只能是final,且必须初始化。局部变量本身既不属于实例，也不属于类，所以局部变量不能用static修饰。而成员变量系统会对其进行初始化。

平常我们写A a=new A();其实上我们做的是创建一个A对象(存放在堆中)，并让a这个引用变量指向A对象。当我们打印一个对象时，实际上就是打印输出该对象的toString()方法的返回值。

修饰符：static：有static(类的标识)修饰符修饰的成员(属性或方法)属于类本身(我们称为类成员)，反之属于类的实例(我们称为实例成员)。记住永远不要使用对象去调用static修饰的方法、属性，如果使用了的话，一定要把对象翻译成类(当程序通过实例来访问类变量时，由于类变量本身不属于实例，因此底层是委托给通过类来访问，且有static修饰的成员不能访问非static成员)。this(在任何非static修饰的方法、构造器中可用)关键字的使用有两种：1.this引用：当this在方法中，this代表调用该方法的对象。当this在构造器中，this代表该构造器正在初始化的对象。2.this调用。---只能在构造器的第一行出现。

final修饰符：

final修饰的成员变量(filed),必须由程序员执行初始化(只能被赋值一次，不可改变)。---只能在定义时、初始化块、构造器中指定初始值。如果final修饰类变量(被static修饰的成员)，则只能在定义时或类(static)初始化块中指定初始值。final修饰的局部变量：不能被重新赋值。JVM会把所有用过的对象进行缓存。final修饰的变量,它会被执行宏替换。如果final修饰的变量，可以在编译时就能确定它的值，那么这个变量就不存在。final修饰的方法不能被重写。好处：禁止父类的方法被重写，避免了父类被破坏。final类不能有子类,比较安全。

方法重载(Overload)：同类中相同方法名的方法参数不同，与返回值类型和修饰符static无关。

方法重写(Override)：重写也叫覆盖：当子类从父类那里继承得到的方法不能真正满足子类的需求时,子类可以重写父类的方法。方法名相同、参数列表相同；子类重写的方法访问权限必须比父类方法的访问权限更大或相等；重写方法的返回值类型必须比父类方法的返回值类型更小或相等，重写方法的声明抛出的异常必须比父类声明抛出的异常更小或相等。注意：我们常在方法上加@Override注解，它的作用是让编译器执行检查，要求该类必须重写父类的该方法。

包、静态导入：包的作用其实即相当于命名空间的作用。Java要求一个类的完整类名 = 包名 + 类名。包名统一小写，当我们要把一个类放入包中,必须在该类的源代码中使用package来

声明包,生成的class文件必须放在相应文件结构下面。普通的import是省略包,而import static 导入指定类下的静态成员(调用时可省略类名)。

面向对象之封装：封装就是隐藏不想被外界操作的属性、方法、构造器。提供给别人调用的方法。封装目的：简化编程，能更好地保证对象的完整性。提到封装不得不谈：访问权限修饰符：private --当前类访问权限。没有访问修饰符 --包访问权限。protected --子类访问权限。public --公共访问权限。访问权限由小到大：private-->没有(默认)访问修饰符-->protected-->public。如果直接对属性赋值，很容易导致类的数据完整性被破坏，一般不推荐。

面向对象之继承(extends)：类与类之间从一般到特殊(“is a”)的关系，如苹果是一个种水果，那么苹果就可以从水果那里继承，子类继承父类(隐式地获得父类的对象)，就可以获得的属性和方法。但这样会增加耦合度(一旦父类属性或方法变了，子类也要跟着变)，所以一般推荐使用组合，继承能不用尽量不用。补充一下：组合是一种“has a”的关系可以显式地获得被包含类的对象。组合关系在运行期决定，而继承关系在编译期就已经决定了，组合是在组合类和被包含类之间的一种松耦合关系，而继承则是父类和子类之间的一种紧耦合关系。

super关键字：super有两种使用方式：super限定：强制去访问父类的方法和field。--如果子类中定义了与父类同名的field，并不会被覆盖。super调用：用于显式调用父类的构造器。规则：子类的构造器【总会】先调用父类的构造器【一次】，注意：super调用、this调用都必须出现在构造器的第一行，且不能重复出现。如果有this调用，子类构造器会先找到this调用所对应子类中被重载的构造器。

面向对象之多态(Ploymorphism)：同一种类型的变量，在执行同一个方法时，表现出多种行为特征。

举例说明1：

```
/*
 * @Author: lhy
 * @Description: 水果类，用作父类；苹果类从Fruit继承
 */
class Fruit {
    public void info() {
        System.out.println("父类的info方法被调用！");
    }
}
```

```
}

class Apple extends Fruit {
    public void info() {
        System.out.println("子类的info方法被调用前!");
        super.info();
        System.out.println("子类的info方法被调用后!");
    }
}

public class Test {
    public static void main(String[] args) {
        Fruit f1 = new Apple();
        Fruit f2 = new Fruit();
        f1.info();
        f2.info();
    }
}
```

运行一下，可以看到同一种类型(Fruit)的变量f1，f2执行用一个方法，表现出不同的行为特征。而且子类的实例完全可以当成父类的对象使用。多态增加了Java语言的灵活性，也是和设计模式紧密相连的(之后会讲到)。

再看一个：

```
class Fruit {
    protected int i=1000;
    public void test() {
        System.out.println("父类的test方法被调用!");
    }
}
```

```
}

class Apple extends Fruit {
    protected int i=10;
    public void test() {
        System.out.println("子类的test方法被调用！");
    }
}

public class Test {
    public static void main(String[] args) {
        Fruit f1 = new Apple();
        //当调用引用变量时，它总是呈现出它的运行时特征。
        f1.test();
        System.out.println(f1.i);
        /*引用类型之间，只能在具有继承关系的两个类型之间转换，否则编译时就会报错！
        强制类型转换的运算符是(类型)*
        System.out.println(((Apple)f1).i);
    }
}
```

需要特别注意的：Java的引用变量有两种类型，编译时类型(由声明它的类型决定)，运行时类型(由实际赋给该变量的类型决定)。在编译阶段：编译器并不知道引用变量实际所引用的对象的类型。

instanceof关键字：用来判断前面的变量所引用的对象是否为后面类型的实例。注意：instanceof前面操作数的类型要么与后面的类相同，要么与后面的类有父子关系，否则会引起编译时错误。

结束语：

内容比较多，所以打算分2--3章讲。明天开始讲接口、抽象类、内部类、枚举类，这个比较重要，在以后的设计模式中会经常看到，而且在以后得Java web开发中经常用到，web 框架本身并不难，只是有些xml文件里面的配置比较麻烦，但这些配置一般不需要我们刻意去记，注重理解会用就行。所以无论是以后的Java web还是Android，学好Java基础是非常有必要的。

1.5 Java学习系列(五)Java面向对象之抽象类、接口、内部类、枚举类详解

发表时间: 2013-09-24 关键字: java, 编程

抽象类、接口常常与设计模式紧密相连。掌握抽象类、接口等其实很简单。下面以说明+实例的方式来讲，这样更容易理解。

抽象类

先看一个关键字“abstract”，我们知道它是抽象的意思。所谓抽象，说的直白一点就是同一件事情，有不同的实现。比如呼吸这个动作，人需要肺呼吸，而鱼需要鳃呼吸。代码实现如下：

```
abstract class CommonMode{
    abstract void breathe(String name);
}

class Fish extends CommonMode{
    @Override
    void breathe(String name) {
        System.out.println("鱼呼吸用"+name);
    }
}

class Person extends CommonMode{
    @Override
    void breathe(String name) {
        System.out.println("人呼吸用"+name);
    }
}

public class Test {
    public static void main(String[] args) {
        CommonMode fish = new Fish();
        CommonMode person = new Person();
        fish.breathe("鳃");
        person.breathe("肺");
    }
}
```

这样我们就不难理解所谓“抽象化”，就是指从具体问题中，提取出具有共性的模式，再使用通用的解决方法加以处理。需要注意的是abstract修饰符只能修饰类或方法。修饰类时该类就成了抽象类，**抽象类**用四个字就可以概括：有得有失。所谓“得”就是增加了一个可以包含抽象方法(由子类实现)的功能，所谓“失”就是不能创建实例，而其它的功能普通类有的抽象类都有(如：定义普通方法、初始化块、内部类等)，需要补充的是抽象类的构造器主要是提供其子类的构造器调用。**抽象方法**：使用abstract修饰，它没有方法体，而且它必须由子类重写，由于子类必须重写抽象父类的方法，所以abstract不能与final(有final修饰的方法，意味着不能被重写)同时出现，而且子类继承了(抽象)父类，意味着abstract不能与final同时出现；除此之外，abstract还不能与static、private(private意味着不能被子类访问)同时存在。注意子类要实现抽象类的所有抽象方法，否则子类也是抽象类。抽象类的作用主要是与“模板模式”联系在一起，以后会在设计模式中讲到，这里就不细说了。为加深理解，再举一例：

```
abstract class ParentClass {
    //定义抽象方法
    abstract void learn(String name);
    //抽象父类的构造器
    ParentClass(){
        System.out.println("父类无参构造器被调用！");
    }
    {
        System.out.println("实例初始化块被调用！");
    }
    static{
        System.out.println("类初始化块被调用！");
    }
    //定义内部类(下面会讲到)
    class A{
    }
    //定义普通方法
    public void info(){
        System.out.println("抽象父类中的info方法被调用！");
    }
}

class SubClass extends ParentClass{
    //当执行这个方法时，会调用抽象父类的构造器
    public SubClass(){}
    @Override
    void learn(String name) {
```

```
        System.out.println("我在学习"+name);
    }
    public void test(){
        System.out.println("子类test方法被调用！");
    }
}

public class AbstractClassTest {
    public static void main(String[] args) {
        //说明：new SubClass()会创建一个SubClass类的对象存放在堆内存中，
        //并让pc这个引用变量指向这个刚创建的对象，这个就是我们常听到的父类引用指向子类对象。
        //当我们只写new SubClass(); 来创建子类对象，我们从打印语句中我们可以看到：父类无参构造器被调用！

        //这说明父类构造器主要是提供其子类的构造器调用
        ParentClass pc=new SubClass();
        //因为子类的实例完全可以当成父类的对象使用，并且子类SubClass重写了抽象父类的learn方法，所以当然可以调用

        pc.learn("Java");
        pc.info();
        //当我们写pc.test();我们会看到编译器会报错，因为抽象父类ParentClass中根本没有test方法被定义
    }
}
```

接口(interface)

接口的作用非常丰富，接口往往是和设计模式结合在一起的。接口是从多个相似的类中提取出来的一种规范，接口可认为是一种彻底的抽象类。接口定义的语法格式如下：

```
[修饰符] interface 接口名{
//可以有属性定义、抽象方法(有方法的话只能是抽象的)、内部类/内部接口/内部枚举定义
}
```

修饰符：public/省略；接口名：每个单词首字母大写，推荐接口用“形容词”。field默认有3个修饰符：public static final；抽象方法默认有2个修饰符：public abstract；内部类/内部接口/内部枚举定义默认也有2个修饰符：public static；注意属性声明时必须指定初始值，原因是final修饰的类变量只能在声明时、静态初始化块中指定初始值，但接口不包含初始化块，所以属性声明时必须指定初始值，并且推荐其属性变量所有字母全部大写。下面通过实例说明：

```
interface Identifiable{
    void insert(String name);
}

interface A{
    //接口里的属性声明必须指定初始值
    int VALUE = 100;
    //接口里面可以有内部接口
    interface B{
        int INNERVALUE = 20;
    }
}

//一个接口可以有N个直接父接口
interface IRechargeable extends Identifiable,A{
    void charge(String name);
}

class Usb implements Identifiable{
    @Override
    public void insert(String name) {
        System.out.println("插入"+name);
    }
}

//子类可以实现多个接口
class DataLine implements Identifiable,IRechargeable{
    @Override
    public void insert(String name) {
        System.out.println("插入"+name);
    }
    @Override
    public void charge(String name) {
        System.out.println("数据线为"+name+"充电");
    }
}

public class InterfaceTest {
    public static void main(String[] args) {
        Identifiable usb = new Usb();
    }
}
```

```
        IRechargeable dl = new DataLine();
        usb.insert("U盘");
        dl.insert("数据线");
        dl.charge("手机");
        System.out.println(IRechargeable.VALUE);
        System.out.println(A.B.INNERVALUE);
    }
}
```

一般我们在功要实现的功能少时用抽象类；功能多时用接口。

内部类

内部类(在界面编程点击处理事件中经常使用)也称寄生类，实质就是把一个类作为类的成员放在类里面定义。听起来有点绕，下面通过实例来说明：

```
//在外部类里面访问非静态内部类
public class Outer {
    int value = 1;
    class Inner{
        int value = 2;
        public void info(){
            int value = 3;
            //由于info方法里面的局部变量,所以此时输出的value为3
            System.out.println("value="+value);
            //由于this在方法中代表调用该方法的对象,所以value为该Inner类对象的属性值2
            System.out.println("value="+this.value);
            //由于该方法所在的类时没有static修饰的内部类,
            //所以该类必须寄生在“外部类”的实例里,这里的Outer.this就代表了外部类的实例

            //关于this,你只要记住一点就行:方法是谁的,就用谁来调用
            System.out.println("value="+Outer.this.value);
            //比如下面的调用:Outer.this 内部类寄生在外部类的实例里面,
            //既然Outer.this代表外部类(宿主)的实例,那么当然可以访问外部类的成员(包括
            field、方法)。
```

```
        Outer.this.test2();
    }
}
public void test(){
    new Inner().info();
}
public void test2(){
    System.out.println("外部类的成员方法test2被调用!");
}
public static void main(String[] args) {
    Outer outer = new Outer();
    outer.test();
}
}
```

//在外部类里面访问静态内部类

```
public class Outer {
    int value = 1;
    static class Inner{
        int value = 2;
        public void info(){
            int value = 3;
            System.out.println("value="+this.value);
        }
    }
}
public static void main(String[] args) {
    new Inner().info();
}
}
```

//在外部类外面访问非静态内部类

```
class Out{
    int value = 1;
    class In{
        int value = 2;
```

```
        public void info(){
            int value = 3;
            System.out.println("value="+value);
            System.out.println("value="+this.value);
            System.out.println("value="+Out.this.value);
        }
    }
}

public class Test {
    public static void main(String[] args) {
        //注意：没有static修饰的内部类，必须寄生在“外部类”的实例里
        Out.In in = new Out().new In();
        in.info();
    }
}
```

//在外部类外面访问静态内部类

```
class Out{
    int value = 1;
    static class In{
        int value = 2;
        public void info(){
            int value = 3;
            System.out.println("value="+value);
            System.out.println("value="+this.value);
            //静态内部类也属于静态成员，因此它不能访问外部类的非静态成员(包括属性、方法)。所以下面写法是错的
            //System.out.println("value="+Out.this.value);
        }
    }
}

public class Test {
    public static void main(String[] args) {
        //注意：有static修饰的内部类属于外部类本身，这是只需将外部类看成是内部类的包名即可
        Out.In in = new Out.In();
        in.info();
    }
}
```

```
    }  
}
```

还有一种非静态内部类派生子类：由于子类的构造器必须调用父类构造器一次，因此必须在子类构造器中使用宿主对象来调用它的构造器。这个用的比较少，这里就不细谈了。

下面讲一下匿名内部类：当程序创建匿名内部类时，会立即创建匿名内部类(实现类)的实例。

定义语法格式如下：

```
new 接口() | 父类构造器(参数){  
//类体部分。  
};
```

使用规则：1.匿名内部类必须显式的继承一个父类，或实现一个接口。2.匿名内部类必须实现接口或抽象类中所有的抽象方法。3.匿名内部类不能有构造器而且程序以后无法再访问它，因为它没有类名。实例说明如下：

```
interface Identifiable {  
    void insert(String name);  
}  
  
public class Test {  
    public static void main(String[] args) {  
        //此处相当于创建了Identifiable匿名的实现类,并创建了匿名内部类的实例  
        //将实现类的实例赋值给接口变量，这是典型的“向上转型”  
        Identifiable idf = new Identifiable(){  
            @Override  
            public void insert(String name) {  
                System.out.println("插入"+name);  
            }  
        };  
        idf.insert("U盘");  
    }  
}
```

局部类：在当前方法里面定义，它只在该方法里有效，你可以把它当做一个局部变量，这个很少用，了解即可。

枚举类：枚举类是一种实例数固定的类，既然是实例固定，那当然不能创建实例。定义语法格式如下：

修饰符 enum 枚举名{

//立即在第一行列出该枚举的所有实例（但实际上是创建枚举实例,会默认调用无参构造器）。

}

--修饰符 public|省略| abstract|final（这两个必须出现一个），默认final；构造器无论是否使用private修饰，默认总是private。下面举例说明：

```
public enum Test {  
    // 列出所有枚举值,也就是该枚举类的所有可能的实例;  
    // 下面相当于:Test MALE = new Test("男");  
    MALE("男"), FEMALE("女");  
    private String name;  
    Test(String name) {  
        this.name = name;  
    }  
    {  
        System.out.println("实例初始化快被调用~~");  
    }  
    class A {}  
    public void info() {  
        System.out.println("普通的info方法~" + this.name);  
    }  
    public static void main(String[] args) {  
        //Test.FEMAIL会调用构造方法,枚举类中有几个实例就调用几次构造方法  
        //通过上一章,我们知道初始化快会在构造方法被调用之前调用(其实就是把初始化快放在了构造  
        方法的第一行)  
        //运行一下,我们会看到“实例初始化快被调用”被执行了两次  
        Test.MALE.info();  
    }  
}
```

下面再看一个实现接口的枚举：

```
interface Directionable {  
    void pointDirection();  
}  
  
//实现接口的枚举，如果直接实现所有的抽象方法，则此时枚举类就不再是抽象枚举。  
public enum Test implements Directionable {  
    //下面这4个实例,相当于是public static final修饰的  
    EAST, WEST, SOUTH, NORTH;  
    @Override  
    public void pointDirection() {  
        System.out.println("指向" + this);  
    }  
    public static void main(String[] args) {  
        Test.SOUTH.pointDirection();  
    }  
}
```

看了上了两个实例，枚举类是不是很简单呢，呵呵。下面再看一个难一点的。

```
//抽象枚举需要创建匿名内部类  
public enum Test {  
    // 当枚举类是抽象类时,还需要立即创建匿名内部类的实例  
    //这里的ADD和ADD()其实是一样的,都是创建实例并实现抽象方法。  
    ADD {  
        @Override  
        public double eval(double m, double n) {  
            return m + n;  
        }  
    }, SUB {  
        @Override  
        public double eval(double m, double n) {  
            return m - n;  
        }  
    }  
}
```

```
    }, MULTI {  
        @Override  
        public double eval(double m, double n) {  
            return m * n ;  
        }  
    }, DIV {  
        @Override  
        public double eval(double m, double n) {  
            return m / n;  
        }  
    };  
    // 如果枚举里面已经有了抽象方法,该枚举类默认就有了abstract修饰,此时该枚举类就没有了final修饰  
    public abstract double eval(double m,double n);  
    public static void main(String[] args) {  
        System.out.println(Test.MULTI.eval(9, 9));  
    }  
}
```

结束语：

今天就到这，明天开始学习Jar命令打包、正则表达式、国际化。

1.6 Java学习系列(六)Java面向对象之Jar命令、正则表达式、国际化详解

发表时间: 2013-09-28 关键字: java, 正则表达式, 虚拟机, jvm

首先向大家道个歉，前面3天由于在忙着写项目说明文档，所以耽误了一下。今天借着中午这段时间把前面的补回来。话不多说，下面步入正题吧！

Jar命令：jar可以把多个文件打包成一个压缩包，得到的压缩包通常有3种：1) *.jar -它里面包含N个class文件。2) *.war(Web) -它是一个Web应用打包生成的包。

3) *.ear(Enterprise) -它是一个企业应用打包生成的包。这里我们学习时常用的*.jar这种方法。在运行命令窗口输入jar我们可以看到jar命令的如下参数选项：

选项包括：

```
-c 创建新的归档文件
-t 列出归档目录
-x 解压缩已归档的指定（或所有）文件
-u 更新现有的归档文件
-v 在标准输出中生成详细输出
-f 指定归档文件名
-m 包含指定清单文件中的清单信息
-e 为捆绑到可执行 jar 文件的独立应用程序
指定应用程序入口点
-0 仅存储；不使用任何 ZIP 压缩
-M 不创建条目的清单文件
-i 为指定的 jar 文件生成索引信息
-C 更改为指定的目录并包含其中的文件
如果有任何目录文件，则对其进行递归处理。
清单文件名、归档文件名和入口点名的指定顺序
与 "m"、"f" 和 "e" 标志的指定顺序相同。
```

举例说明几个常用的：

命令：jar -cvf test.jar *.class //为新创建的jar包取名为test

jar -cMf test.jar *.class //创建压缩包，不生成清单文件(META-INF下面的manifest.mf文件)

jar -tvf test.jar //查看压缩包

jar -xvf test.jar //解压 (加v可以看到详细的过程，可不加)

jar -uvf test.jar A.class //添加A.class到test.jar中。

【打包成可执行的JAR包】jar -cvfe my.jar UserTest *.class //通过-e告诉JVM UserTest为主类,并打包。

这样如果机器上安装了独立的虚拟机，只要双击“可执行型”的JAR包即可运行。

如果没装则：java -jar my.jar //运行

javaw -jar my.jar //主要用于有界面的程序(javaw和java差不多)

若不需要清单文件，可以使用windows里的WinRAR或WinZi压缩，然后把后缀改成.jar即可。

Jar命令的好处：方便管理。JVM可以直接加载Jar包，这样就可以一次加载N个类，效率更高。

Java程序入口(main)方法说明：`public` -系统来调用该方法时，不可能在同一个包中，也不会是它的子类

`static` -无需创建实例，直接用类名调用入口方法。--执行：`java 主类类名 数组元素1 数组元素2...`

`String[] args`：`args`默认是长度为0的数组，可在程序运行时动态传入`args`参数值。

正则表达式：就是一个可以匹配N个字符串的字符串模板。

正则表达式所支持的通配符：

`.` -可以匹配任意字符。

`\s` -代表一个任意空白(空格、Tab)

`\S` -代表一个任意的非空白

`\d` -代表一个任意的数字(digital)

`\D` -代表一个任意的非数字

`\w` -代表一个单词字符

`-W` -代表一个任意的非单词字符

`str.matches(regex);`//判断`str`字符串是否匹配给定的正则表达式。

`"123.123".replaceAll("\\.", "#");` //结果：`123#123`

特殊字符（实际用时记住要转义`\`）：`()[]{} \ ? * + ^`（一行的开头）`$`（一行的结尾）`|`

表示出现次数的“副词”：

`?` --代表它前面的东西可以出现0~1次

`*` --代表它前面的东西可以出现0~N次

`+` --代表它前面的东西可以出现1~N次

`{n,m}` --代表它前面的东西可以出现n~m次

`{n,}` --代表它前面的东西至少出现n次

`{,m}` --代表它前面的东西最多出现m次

`{n}` --代表它前面的东西必须出现n次

方括号表达式：

枚举：`[ab中]` --代表a或x或者“中”。

范围：`[a-c]` --代表a,b,c中的任意一个字符。

枚举与范围：`[a-c1-3]`--代表a,b,c,1,2,3中的任意一个字符。

表示求否：`[^a-c]` --代表不含a,b,c其中任意一个字符。

表示求交：`[a-g&&[^b-d]]`：--代表a,e,f,g中的任意一个字符。--总结：一个字符用`\`，多个

字符用[]，字符次数用{}

(com|org|cn)：表示必须含有其中之一。

--除此自外，Java专门提供了两个工具类：

Pattern - 它就是代表正则表达式模板。

Matcher -代表一个匹配工具类。

find()：查找下一个匹配正则表达式的字符串。group()取出上一次与正则表达式匹配的字符串。

举例说明(找出邮箱)：

```
public class Regextest {
    public static void main(String[] args) {
        String str = "这本书价格是52.9元,作者的两个邮箱：一个是123@qq.com，另外一个
是234@163.com";
        Pattern pattern = Pattern.compile("[a-zA-Z0-9]{2,}@[a-zA-Z0-9]{2,}\\.(com|org|r
        Matcher matcher = pattern.matcher(str);
        while(matcher.find()){
            System.out.println(matcher.group());
        }
    }
}
```

国际化(简称：i18N)：希望一个程序，可以“自适应”所有用户环境。--本质是“查找”、“替换”。

国际化的步骤：

(1)提供资源文件，负责为程序提供国际化消息，同时资源文件的文件名必须满足：

<baseName>_语言代码_国家代码.properties。如果资源文件包含了非西欧字符，需要使用native2ascii工具类处理这个文件。

处理格式：native2ascii 要处理的文件 生成的新文件名(资源文件)。

(2)使用ResourceBundle来加载国际化资源文件

(3)从所加载的文件中查找key对应的字符串进行替换。

--调用ResourceBundle的getMessage()方法（负责为消息中的占位符填充参数值）输出国际化消息。

举例说明1：

```
/*
 * Description:获取Java语言的所支持的所有Locale
 * */
public class LocaleTest {
    public static void main(String[] args) {
        Locale[] locals = Locale.getAvailableLocales();
        for (Locale ls : locals) {
            System.out.println(ls.getDisplayCountry() + "    "
                               + ls.getLanguage() + "    " + ls.getCountry());
        }
    }
}
```

举例说明2：

1)我们写一个test.properties文件，内容为：hi=你好！welcome={0},欢迎你！2)执行命令行：native2ascii test.properties test_zh_CN.properties

我们将看到生成的内容如下：hi=\u4f60\u597d\uff01
welcome={0},\u6b22\u8fce\u4f60\uff01

3)编码如下：

```
/*
 * Description:获取并替换国际化资源文件信息
 * */
public class LocaleTest {
    public static void main(String[] args) {
        // Java 虚拟机实例的当前默认语言环境值
        Locale currentLocale = Locale.getDefault();
        // 加载国际化资源文件,test为资源文件名前缀
```

```
ResourceBundle bundle = ResourceBundle.getBundle("test",currentLocale);  
    // 从bundle所加载的文件中查找hi对应的字符串进行替换。  
    System.out.println(bundle.getString("hi"));  
    //将占位符的内容替换为 "张三"  
    System.out.println(MessageFormat.format(bundle.getString("welcome"), "张三"));  
}  
}
```

结束语：

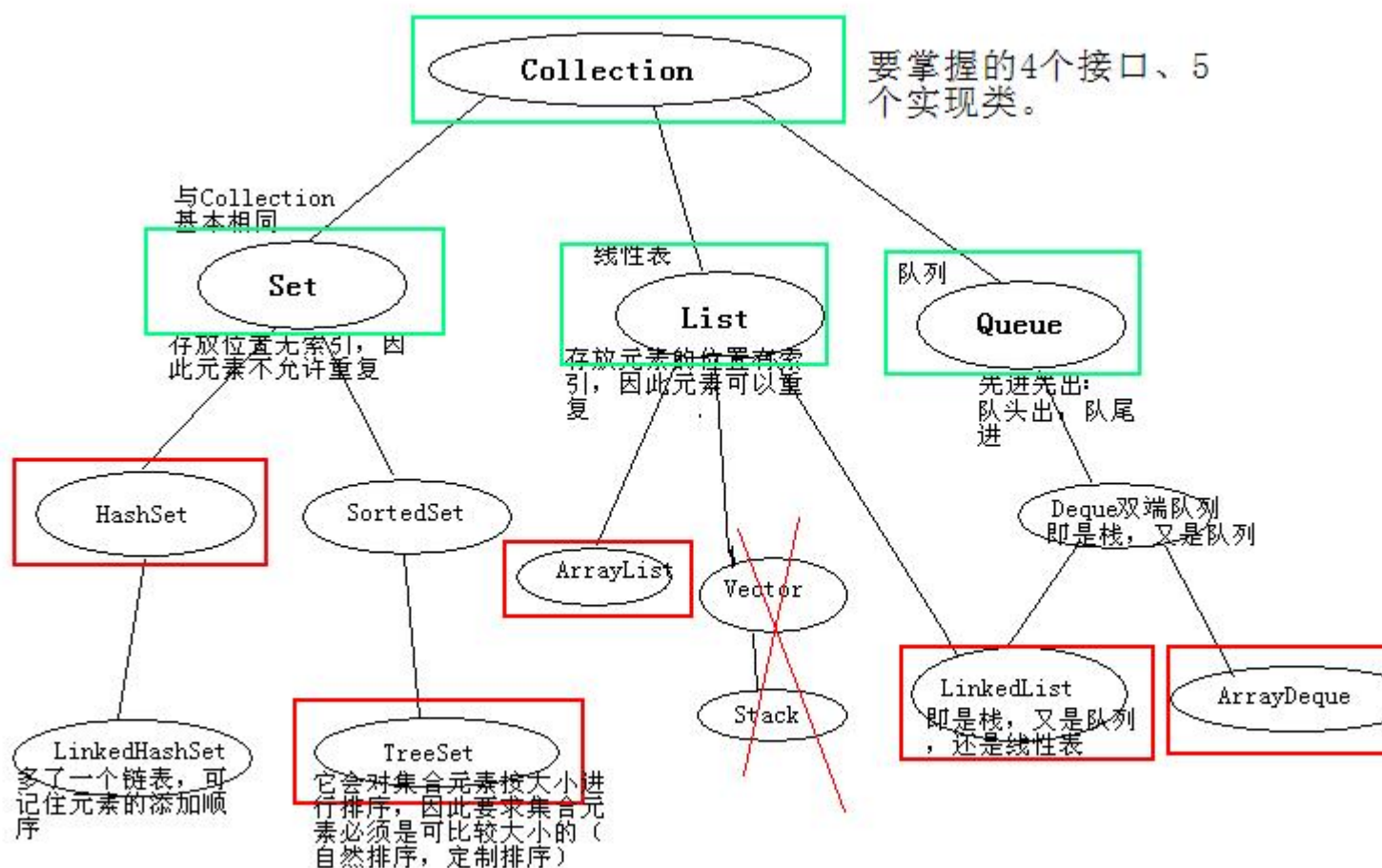
今天内容就讲这么多，明天开始学习Java面向对象之集合框架。内容比较多，所以打算分两次讲。

1.7 Java学习系列(七)Java面向对象之集合框架详解(上)

发表时间: 2013-09-29 关键字: java, jdk, 编程

Java集合

有时也将集合称为容器类，它的作用就是用来“装对象”的。这里要注意的是集合也可以是对象。下面先看一张图：



HashSet：底层用一个数组存元素 --而且这个数组的长度永远是2的N次方。

HashSet底层就是HashMap实现的。

HashSet的构造器：HashSet(int initialCapacity, float loadFactor)

--initialCapacity：控制底层数组的长度。

如果传入数组长度不是2的N次方，HashSet会自动扩展到2的N次方。

--loadFactory：当HashSet感觉到底层数组快满时，它会再次创建一个长度是原来数组长度

2倍的数组。原有的数组就变成垃圾，并且要把原有数组中的元素复制到新数组中，这个过程也叫“重hash”。loadFactory越小,越耗内存；loadFactory越大，性能越低。

举例说明1：

```
public class Test {  
    public static void main(String[] args) {  
        //没有泛型限制的集合  
        Collection c1 = new HashSet();  
        c1.add(1); //本来1不是对象，无法装入集合，但由于jdk提供的自动装箱功能，它会将1包装成  
对象  
        c1.add(new Date());  
        //加入泛型限制的集合，意味着该集合只能装“指定类型”的对象。  
        //jdk1.7可使用“菱形语法”将new HashSet<String>换成new HashSet<>  
        Collection<String> c2 = new HashSet<String>();  
        //c2.add(1);错误：只能装String类型的对象  
        c2.add("张三");  
        c2.add("李四");  
        c2.add("王五");  
        //判断集合是否包含指定元素  
        System.out.println(c2.contains("张三"));  
        System.out.println(c2.contains("赵六"));  
        //遍历Set集合，有两种方式：1)foreach循环,2)迭代器  
        System.out.print("使用foreach循环遍历:");  
        for(String e : c2){  
            System.out.print(e+" ");  
        }  
        Iterator<String> it = c2.iterator();  
        System.out.print("\n使用迭代器遍历:");  
        while(it.hasNext()){  
            System.out.print(it.next()+" ");  
        }  
        System.out.println();  
        Collection<String> c3 = new HashSet<String>();  
        c3.add("张三");  
        c3.add("王五");
```

```
        c3.add("赵六");
        //求差集c2-c3
//      c2.removeAll(c3);
//      System.out.println("c2与c3差集："+c2);
        //求并集
//      c2.addAll(c3);
//      System.out.println("c2与c3并集"+c2);
        //求交集
        c2.retainAll(c3);
        System.out.println("c2与c3交集："+c2);
    }
}
```

举例说明2：

```
public class TestHashSet {
    public static void main(String[] args) {
        HashSet<String> hs = new HashSet<String>(3); //底层数组容量会自动会发展到4(注意：2
的N次方)
        hs.add("1");
        hs.add("2");
        hs.add("3");
        hs.add("4");
        hs.add("5");
        //当HashSet感觉到底层数组快满时，它会再次创建一个长度是原来数组长度2倍的数组，此时新
的数组长度为8
        System.out.println(hs);
    }
}
```

HashSet存入机制:

1)当有元素加进来时，HashSet会调用该对象的hashCode()方法，得到一个int值；

2)根据hashCode()返回的int值,计算出它在HashSet的存储位置(数组中的索引);

3)如果要加入的位置是空的,直接方法即可。如果要加入的位置已经有元素,此处就会形成“链表”,数组越满,就越有可能出现链表。

HashSet取元素机制:

1)当有元素加进来时,HashSet会调用该对象的hashCode()方法,得到一个int值;

2)根据hashCode()返回的int值,计算出它在HashSet的存储位置(数组中的索引);

3)如果该位置恰好是要找的元素,直接取出即可。如果该位置有“链表”,HashSet要“挨个”地搜索链表里的元素。

HashSet怎么才会认为两个对象是相等的? (要求自定义类的hashCode()和equals()方法是一致的,即方法中所用到的关键属性要一致)

1、两个对象的hashCode()返回值相等;

2、两个对象通过equals比较返回true。

LinkedHashSet(HashSet的一个子类):它与HashSet的存储机制相似,但LinkedHashSet额外地有一个链表,这个链表可以保证**LinkedHashSet**能记住元素的添加顺序。

TreeSet(sortedset接口的实现类):它保证Set里添加的元素后是“大小排序”的。底层用一个“红黑树”存放元素。

举例说明(使用TreeSet要求集合元素必须是可以比较大小的):

```
public class TestTreeSet1 {  
    public static void main(String[] args) {  
        TreeSet<Integer> ts = new TreeSet<Integer>();  
        ts.add(3);  
        ts.add(1);  
        ts.add(2);  
        ts.add(5);  
        ts.add(4);  
        System.out.println(ts);  
    }  
}
```

```
}  
}
```

```
public class TestTreeSet2 {  
    public static void main(String[] args) {  
        TreeSet<String> ts = new TreeSet<String>();  
        ts.add("t");  
        ts.add("r");  
        ts.add("e");  
        ts.add("e");  
        System.out.println(ts); //字符串的大小  
    }  
}
```

TreeSet元素存入、检索的性能也比较好。

Java的比较大小有两种方式：

A. --自然排序。所有集合元素要实现Comparable接口。

B. --定制排序。要求创建TreeSet时，提供一个Comparator对象(负责比较元素大小)。

举例说明：

```
class Apple implements Comparable<Apple> {  
    private double weight; // 规定:苹果重量大的苹果越大  
  
    public Apple(double weight) {  
        this.weight = weight;  
    }  
  
    @Override //自然排序  
    public int compareTo(Apple obj) {  
        return this.weight > obj.weight ? 1 : this.weight < obj.weight ? -1 : 0;  
    }  
  
    @Override  
    public String toString() {
```

```
        return "Apple[weight=" + this.weight + "];"
    }
}

public class TreeSetTest {

    public static void main(String[] args) {
        TreeSet<Apple> set = new TreeSet<Apple>();
        set.add(new Apple(2.3));
        set.add(new Apple(1.2));
        set.add(new Apple(3.5));
        for (Apple ele : set) {
            System.out.println(ele);
        }
    }
}
```

```
class Bird {
    private String name;

    public String getName() {
        return name;
    }

    public Bird(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "Bird[name=" + name + "];"
    }
}

public class TreeSetTest2 {

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

```
// 如果集合元素本身没有实现Comparable接口
// 那就要求创建TreeSet时传入一个Comparator对象
TreeSet<Bird> set = new TreeSet<Bird>(new Comparator<Bird>() {
    @Override
    public int compare(Bird o1, Bird o2) {
        if (o1.getName().compareTo(o2.getName()) > 0) {
            return -1;
        } else if (o1.getName().compareTo(o2.getName()) < 0) {
            return 1;
        } else {
            return 0;
        }
    }
});
set.add(new Bird("aabc"));
set.add(new Bird("abc"));
set.add(new Bird("Z"));
set.add(new Bird("dx"));
System.out.println(set);
}
```

ArrayList(JDK1.2)与Vector(JDK1.0 , 已经过时)都是基于数组实现的，它们的**区别**如下：
ArrayList(可根据索引来存取元素)是线程不安全的，Vector是线程安全的。
--ArrayList的性能比Vector要好，即使在多线程环境下，可以使用Collections集合工具类的synchronizedXxx方法把ArrayList包装成线程安全的。

而LinkedList底层是基于链表实现的，通常认为它的性能比不上ArrayList，它们的**区别**如下：

ArrayList：由于可以根据底层数组的索引存取元素，所以性能非常快，但当插入元素、删除元素时性能低。

LinkedList：由于底层采用了链表来存储元素，因此根据索引存取元素性能较慢，但当插入、删除元素时性能非常快。

举例说明：

```
public class Test {  
    public static void main(String[] args) {  
        List<String> list = new ArrayList<String>();  
        list.add("2");  
        list.add("1");  
        list.add("5");  
        list.add("3");  
        System.out.println(list);  
        list.add(2, "在索引2处插入元素");  
        System.out.println(list);  
        list.set(2, "在索引2处替换的元素");  
        System.out.println(list);  
        list.remove(2);  
        System.out.println(list);  
        //遍历list(类似数组)  
        for(int i =0;i<list.size();i++){  
            System.out.print(list.get(i)+"  ");  
        }  
    }  
}
```

Deque集合：功能被限制的线性表。

```
//把Deque当成栈用  
public class Test {  
    public static void main(String[] args) {  
        //把Deque当成栈用  
        Deque<String> dq = new ArrayDeque<String>();  
        //进栈(压栈):后进先出  
        dq.push("a");  
        dq.push("b");  
        dq.push("c");  
    }  
}
```

```
        dq.push("d");
        //打印首先出栈的元素
        System.out.println(dq.pop());
        //打印访问栈顶的元素
        System.out.println(dq.peek());
    }
}
```

//把Deque当成队列用：先进先出

```
public class Test2 {
    public static void main(String[] args) {
        //把Deque当成队列用：先进先出
        Deque<String> dq = new ArrayDeque<String>();
        //从队列尾部入队
        dq.offer("a");
        dq.offer("b");
        dq.offer("c");
        dq.offer("d");
        //从队列头部出队
        System.out.println(dq.poll());
        //打印访问队头元素
        System.out.println(dq.peek());
    }
}
```

Collections集合工具类：

```
public class TestCollections {
    public static void main(String[] args) {
        List<String> list = new ArrayList<String>();
        list.add("a");
        list.add("b");
        list.add("c");
        list.add("d");
        System.out.println(list);
        //对集合进行反转
        Collections.reverse(list);
    }
}
```

```
System.out.println(list);  
//把b, c位置进行交换  
Collections.swap(list, 1, 2);  
System.out.println(list);  
//将list集合元素进行随机排列  
Collections.shuffle(list);  
System.out.println(list);
```

```
}
```

```
}
```

结束语

今天内容比较多，而且有些和数据结构相关，所以理解起来可能会有些困难。编程我觉得还是要多练，只要你肯多练，就不会那么难了。今天就讲到这，明天开始讲Map。

1.8 Java学习系列(八)Java面向对象之集合框架详解(下)

发表时间: 2013-11-05 关键字: java, apple, jdk, JVM

今天接着上次的来讲，主要谈谈Map。下面先看一张图：

Map里面存的东西是：每个数据项都是key-value对组成。假如我们把value当成是key的“附属物”，Map存储key-value对时，只要考虑key的存储即可，key存储之后，value跟着key即可。再进一步：如果只管Map里面的key，并把所有的key收集起来 ----- 就变成了Set。所以Map与Set是一一对应的。通过查看源码我们可以发现，HashSet底层是由HashMap实现的。HashMap会根据key的hashCode()方法的返回值来计算key的存、取位置。

HashMap怎样才算两个key重复？

- a) 通过equals方法比较返回true；
- b) 两个key的hashCode()返回值相等。

要求自定义类的hashCode()和equals()方法是一致的(即方法中所用到的关键属性要一致)。

TreeMap要求key必须是可比较大小的。

- a) 自然排序：要求所有的key实现Comparable接口；
- b) 定制排序：要求创建TreeMap时提供一个Comparator接口。

TreeMap怎样才算两个key重复？

- a) 通过compareTo()比较大小时返回0，就表明两个元素相等。

Hashtable与HashMap的区别：

- a.Hashtable从JDK1.0就有的，尽量少用。
- b.Hashtable不允许使用null作为key、value，但HashMap允许。
- c.Hashtable是线程安全的。--线程不安全的性能好。

举例说明1(HashMap的使用)：

```
class Apple {  
    private String color;  
    private double weight;  
  
    public Apple(String color, double weight) {  
        this.color = color;  
        this.weight = weight;  
    }  
}
```

```
    }

    @Override
    public String toString() {
        return "Apple[" + color + "," + weight + "]";
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (obj != null && obj.getClass() == Apple.class) {
            Apple apple = (Apple) obj;
            return this.weight == apple.weight
                    && this.color.equals(apple.color);
        }
        return false;
    }

    @Override
    public int hashCode() {
        return color.hashCode() + 13 * (int) weight;
    }
}

public class Test {

    public static void main(String[] args) {
        HashMap<Apple, Double> apples = new HashMap<Apple, Double>();
        apples.put(new Apple("红色", 3.5), 3.4);
        apples.put(new Apple("红色", 4.5), 3.4);
        apples.put(new Apple("黄色", 3.5), 3.4);
        apples.put(new Apple("红色", 3.5), 6.8); //覆盖掉了第一个key
        System.out.println(apples);
    }
}
```

举例说明2(TreeMap的使用)：

```
public class Test {

    public static void main(String[] args) {
        TreeMap<String, Double> score = new TreeMap<String, Double>(
            new Comparator<String>() {
                @Override
                public int compare(String o1, String o2) {
                    return o1.length() > o2.length() ? 1 : o1.length() < o2.length() ? -1 : 0;
                }
            });
        score.put("abd", 89.0);
        score.put("aaaa", 78.0);
        score.put("aa", 90.0);
        score.put("ds", 78.0);
        System.out.println(score.size());
        System.out.println(score);
    }
}
```

结束语

通过上一篇的学习，我们可以发现其实Set和Map用法几乎是一样的，把Map看成是Collection子接口Set的一个分支即可。

今天就讲到这里，明天开始学习Java的异常处理进制。

1.9 Java学习系列(九)Java面向对象之异常机制详解

发表时间: 2013-11-06 关键字: java, jvm, 编程

异常处理机制用来保障我们的程序更加健壮，无论用户怎么操作，都能保证我们的程序都能正常应对的一种处理机制。

一般格式如下：

try{ // 尝试让它执行业务处理，如果可以执行完成，就代表一切正常。

// 业务处理

}catch(异常1 e1){

// 进行异常1处理

}catch(异常2 e2){

// 进行异常2处理

}

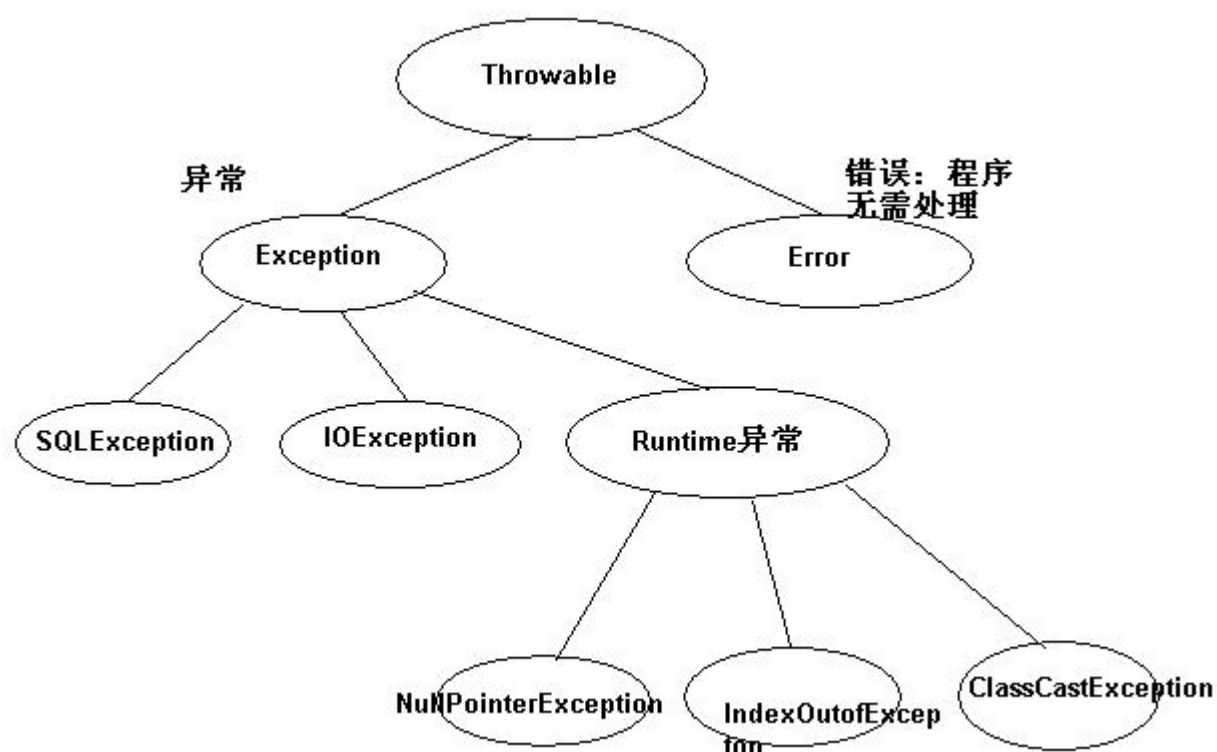
举例说明1：

```
public class Test {  
    public static void main(String[] args) {  
        try {  
            Double d1 = Double.parseDouble(args[0]);  
            Double d2 = Double.parseDouble(args[1]);  
            System.out.println("d1+d2=" + (d1 + d2));  
        } catch (ArrayIndexOutOfBoundsException e1) {  
            System.out.println("请输入两个数字！");  
        } catch (NumberFormatException e2) {  
            System.out.println("输入的必须是数字！");  
        }  
    }  
}
```

Java异常机制的处理流程：

出现异常时，系统会自动生成一个异常对象ex，然后由catch(异常类 e1) ex instanceof 异常类 判断是否为true，为true时传给catch块后的形参，并执行catch块中的代码。对于多个catch块捕捉异常，应该是先捕捉小(子类)的异常，再捕捉大(父类)的异常。通常发生异常时应从第一个自己写的程序位置开始查找并排错。【注意】对于一个异常，最多只有一个catch块捕捉到异常。

异常体系图：



异常处理的完整语法格式：

```
try{  
    // 业务处理  
}catch(异常 ex){ //0~N个  
    // 进行异常1处理  
}finally{ // 0~1个  
  
}
```

注意：try不能独立存在，也就是说catch块和finally块必须有一个。

finally --通常用于回收(关闭)资源，JVM会保证【finally块总会得到执行 --不管有无return语句】，但如果遇到System.exit(0); (用于退出JVM) 就会阻止finally语句的执行。

异常的分类。

checked异常：没有完善错误处理的代码，不会得到执行的机会。--编译器会强制检查。

runtime异常：运行时异常。

checked异常可以更好地保证我们一定会对其进行异常处理，但checked也给编程带来繁琐：要么用throws声明抛出异常，要么显式用catch来捕捉。

异常转译

```
/**
 * @author lhy
 * @description 自定义异常
 */
class MyException extends RuntimeException {
    public MyException() {
    }

    public MyException(String msg) {
        super(msg);
    }

    public MyException(String detailMessage, Throwable throwable) {
        super(detailMessage, throwable);
    }
}

public class Test {
    public static void main(String[] args) {
        test(args);
    }

    public static void test(String[] args) {
        try {
            Integer num1 = Integer.parseInt(args[0]);
            Integer num2 = Integer.parseInt(args[1]);
            System.out.println(num1 / num2);
        }

        // 先用catch来捕捉原有的异常,再用throw抛出一个自定义异常对象;
        // 这种做法也称为“异常转译”
        catch (ArrayIndexOutOfBoundsException ae) { // 数组下标越界异常
            throw new MyException("请输入两个数字");
        } catch (NumberFormatException ne) { // 数字格式异常
```

```
        throw new MyException("必须输入数字");
    } catch (ArithmeticException et) { // 算术异常
        throw new MyException("被除数不能为0");
    } finally {
        System.out.println("finally块总会得到执行");
    }
}
}
```

【注意】(catch)捕捉的异常的范围一定要比抛出的异常范围大。

throws和throw的区别：

A. throws只能在方法签名中用，方法已经注意到了这几个异常，但目前不想处理该异常。

B. throw就是一条普通的语句，throw后紧跟【一个异常实例】。

结束语：今天内容比较简单，所以写的不是很多。明天开始学习IO。

1.10 Java学习系列(十)Java面向对象之I/O流(上)

发表时间: 2013-11-07 关键字: java, 编程, JVM

IO流

我们知道应用程序运行时数据是保存在内存中的，但由于内存中的数据不可持久保存(如断电或程序退出时数据会丢失)，因此需要一种手段将数据写入硬盘或读入内存。面向IO流编程就是一种很好的选择。IO：Input/Output 完成输入输出，所谓输入：是指将外部存储器把数据读入内存，而输出：是指将内存中的数据写入外部存储器(如硬盘、磁盘、U盘)中。

File：代表磁盘上的文件或目录。它的特征是只能盘问磁盘上的文件和目录，但无法访问文件内容，必须使用IO流。

举例说明1(遍历根目录)：

```
public class FileTest {
    public static void main(String[] args) {
        // 列出系统中所有的根目录
        File[] roots = File.listRoots();
        for (File root : roots) {
            System.out.println(root);
        }
        // 下面是相对路径,生成的mydir文件和src同一目录;
        // 注意只有从根目录开始的路径才是绝对路径
        File myFile = new File("mydir");
        if (!myFile.exists()) {
            // 创建相应目录
            myFile.mkdir();
        }
    }
}
```

举例说明2(遍历某个磁盘上的所有文件)：

```
/**
 * @author lhy
 * @description 递归遍历某个磁盘下的所有目录及其子目录
 */
public class FileTest {
```

```
public static void main(String[] args) {
    File f = new File("f:/");
    myList(f);
}

public static void myList(File dir) {
    System.out.println(dir + "目录下包含的目录和子文件有 : ");
    // 返回当前目录所包含的目录和子文件
    File[] files = dir.listFiles();
    for (File file : files) {
        System.out.println("    " + file);
        // 如果file是目录,继续列出该目录下的所有文件
        if (file.isDirectory()) {
            myList(file);
        }
    }
}
```

举例说明3(过滤文件):

```
public class FileTest {
    public static void main(String[] args) {
        File f = new File("f:/");
        File[] files = f.listFiles(new FileFilter() {
            @Override
            public boolean accept(File pathname) {
                try {
                    // getCanonicalPath()规范路径名是绝对路径名,并且是唯一的。

                    if (pathname.getCanonicalPath().endsWith(".txt")) {
                        return true;
                    }
                } catch (IOException e) {
```

```
        e.printStackTrace();
    }
    return false;
}

});
for (File file : files) {
    System.out.println(file);
}
}
```

IO流的分类(如果要访问文件内容，必须使用IO流)：

- a) 按流的方向来分(从程序所在内存的角度来看)：分为输入流、输出流。输入流是指将外部输入读入当前程序所在内存；而输出流是指将当前程序所在的内存的数据输出到外部。
- b) 按流处理的数据来分：分为字节流、字符流。字节流处理的数据单元是字节，适应性广、能共强大；而字符流处理的数据单元是字符，通常来说它主要用于处理文本文件，而且它在处理文本文件时比字节流更方便。
- c) 按流的角色来分：分为节点流、包装流(处理流/过滤流)。节点流直接和一个IO的物理节点(磁盘上的文件、键盘、网络等)关联；而包装流以节点为基础，经包装之后得到的流。

【流的概念模型】

输入流：只要你得到一个输入流对象，你就相当于得到一根“水管”，每个水滴就代表数据单元(字节/字符)。

输出流：只要你得到一个输出流对象，你也相当于得到一根“水管”，水管中无水滴。

下面讲下IO里面常用的几个类。

FileInputStream类的使用：

```
public class Test {
    public static void main(String[] args) {
        // 准备水桶,可以装 "1024" 个水滴(数据单元)
        byte[] bt = new byte[1024];
        FileInputStream fis = null;
        try {
```

```
        fis = new FileInputStream("f:/Test.java");
        int hasRead = -1;
        while ((hasRead = fis.read(bt)) > 0) {
            System.out.println(new String(bt, 0, hasRead));
        }
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        try {
            fis.close();
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
}
```

FileOutputStream类的使用：

```
public class Test {
    public static void main(String[] args) {
        FileOutputStream fos = null;
        try {
            fos = new FileOutputStream(new File("f:/1.txt"));
            fos.write("Java学习系列(十)Java面向对象之I/O流".getBytes());
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                fos.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
    }  
}
```

文件的复制：

```
public class Test {  
    public static void main(String[] args) {  
        FileOutputStream fos = null;  
        FileInputStream fis = null;  
        try {  
            fis = new FileInputStream(new File("f:/Test.java"));  
            fos = new FileOutputStream(new File("f:/Copy.java"));  
            int hasRead = -1;  
            // 作為水桶  
            byte[] buff = new byte[1024];  
            while ((hasRead = fis.read(buff)) > 0) {  
                fos.write(buff, 0, hasRead);  
            }  
        } catch (IOException e) {  
            e.printStackTrace();  
        } finally {  
            try {  
                fis.close();  
                fos.close();  
            } catch (IOException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

ByteArrayOutputStream的使用：

```
public class Test {  
    public static void main(String[] args) {  
        ByteArrayOutputStream bos = new ByteArrayOutputStream();
```

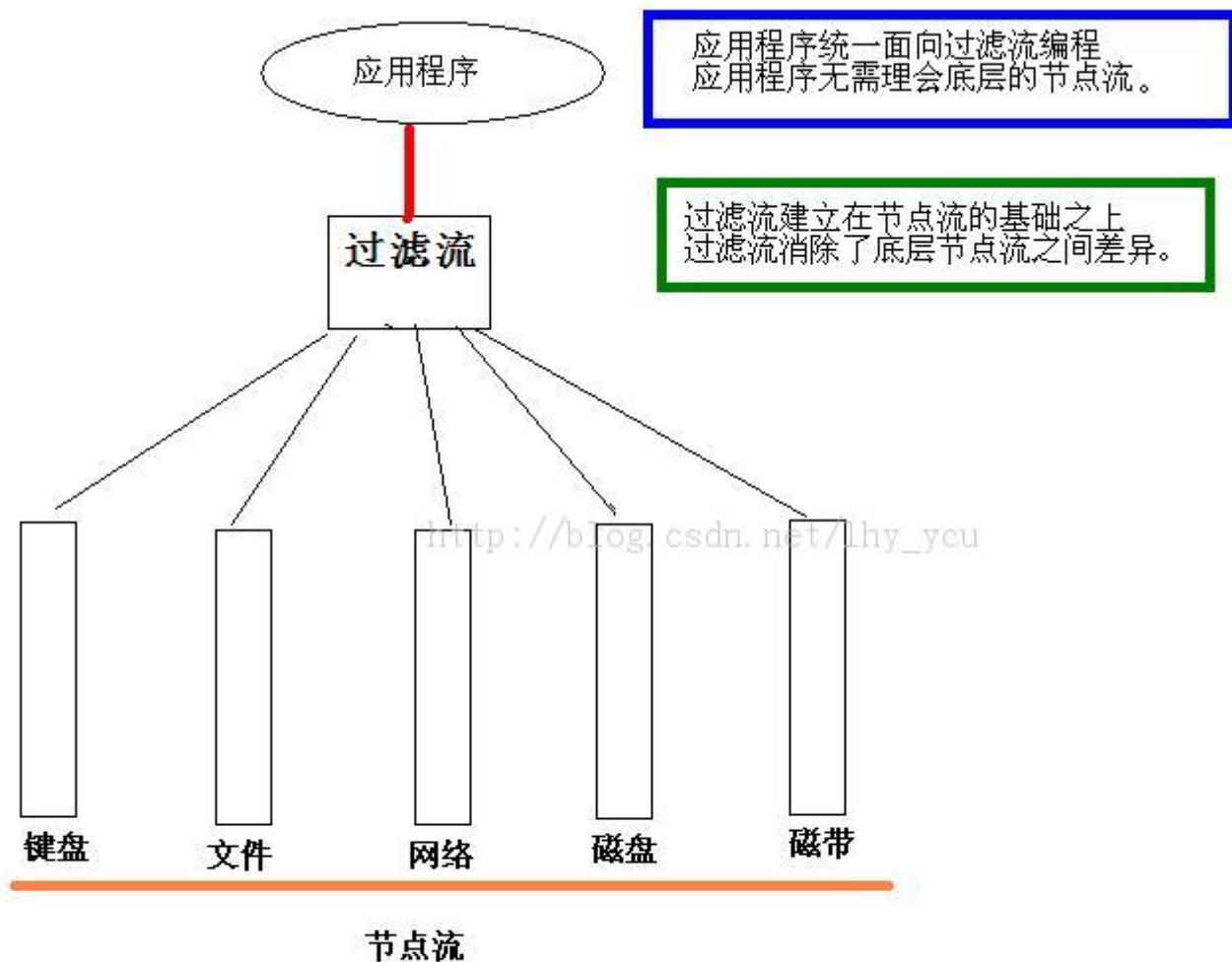
```
String str = "你好,世界!";
try {
    bos.write(str.getBytes());
    System.out.println(new String(bos.toByteArray()));
} catch (IOException e) {
    e.printStackTrace();
} finally {
    try {
        bos.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

缓冲流：我们知道外部存储器的速度要比内存速度慢，所以外部存储器的读写与内存的读写是不同步的。注意：流用完了，别忘了调用flush(把缓冲中的内容刷入实际的节点)或者调用close()也可-系统会在关闭之前，自动刷缓冲。

节点流与缓冲流的联系：

节点流：直接与IO节点关联，--IO节点很多：键盘、网络、文件、磁带....

过滤流：建立在节点流的基础之上。



好处：消除底层节点之间的差异。使用过滤流的方法执行IO更加便捷。

PrintStream的使用：

```
public class Test {  
    public static void main(String[] args) {  
        PrintStream ps = null;  
        try {  
            // 过滤流  
            ps = new PrintStream(new FileOutputStream("f:/1.txt"));  
            ps.println("春");  
            ps.println("夏");  
            ps.println("冬");  
            ps.println("冬");  
        } catch (FileNotFoundException e) {  

```

```
        e.printStackTrace();
    }
}
}
```

重定向标准输入输出。

`System.setOut(new PrintStream("out.txt"));` //将标准输出重定向到指定的输出流 --> out.txt

将字节流转换为字符流(使用BufferedReader可以每次读一行)：

```
public class Test {
    public static void main(String[] args) {
        BufferedReader br = null;
        try {
            br = new BufferedReader(new InputStreamReader(new FileInputStream(
                "f:/Test.txt")));
            String hasLine = null;
            while ((hasLine = br.readLine()) != null) {
                System.out.println(hasLine);
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                br.close();
            } catch (Exception e2) {
            }
        }
    }
}
```

【**规律**】除DataInputStream和DataOutputStream(这两个类后面会讲)这两个特殊的过滤流外，其它所有以Stream结尾的都是字节流，所有以Reader/Writer结尾的结尾的都是字符流。还有两个类：InputStreamReader、OutputStreamWriter是转换类，负责将字节流转换成字符流。

结束语

由于IO流的的内容比较多，所以打算分两次讲。

1.11 Java学习系列(十一)Java面向对象之I/O流(下)

发表时间: 2013-11-08 关键字: java, 虚拟机, socket, 编程

今天接着昨天的IO流讲，内容可能会比较多。

DataInputStream与DataOutputStream

它们是建立在已有的IO的基础上的两个特殊的过滤流。规律：它们只是增加了一些特定的方法读取特定的数据。

举例说明1：

```
public class Test {
    public static void main(String[] args) {
        DataOutputStream dos = null;
        try {
            dos = new DataOutputStream(new FileOutputStream("F:/price.txt"));
            dos.writeDouble(243.21);
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                dos.close();
            } catch (IOException e2) {
                e2.printStackTrace();
            }
        }
    }
}
```

举例说明2：

```
public class Test {
    public static void main(String[] args) {
        DataInputStream dis = null;
        try {
            dis = new DataInputStream(new FileInputStream("F:/price.txt"));
            System.out.println(dis.readDouble());
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
        } finally {
            try {
                dis.close();
            } catch (IOException e2) {
                e2.printStackTrace();
            }
        }
    }
}
```

节点流(System.in是读取键盘输入，可以换成new FileInputStream("f:/1.txt")读文件，也可以换成读网络(Socket)--以后会讲)包装成过滤流编程：

```
public class Test {
    public static void main(String[] args) {
        BufferedReader br = null;
        try {
            br = new BufferedReader(new InputStreamReader(System.in));
            String line = null;
            while ((line = br.readLine()) != null) {
                System.out.println(line);
            }

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Java虚拟机读取其他进程的数据：

Java如何启动其他进程：Runtime实例.exec(String command)

举例说明：

```
public class Test {  
    public static void main(String[] args) {  
        Process process = null;  
        BufferedReader br = null;  
        try {  
            process = Runtime.getRuntime().exec("javac.exe");  
            br = new BufferedReader(new InputStreamReader(  
                process.getErrorStream()));  
            String line = null;  
            System.out.println("编译出错,错误信息如下~~~~~");  
            while ((line = br.readLine()) != null) {  
                System.out.println(line);  
            }  
        } catch (IOException e) {  
            e.printStackTrace();  
        } finally {  
            try {  
                br.close();  
            } catch (IOException e2) {  
                e2.printStackTrace();  
            }  
        }  
    }  
}
```

RandomAccessFile ---随机(任意)访问文件。 --创建时需要指定r/w模式。

Random --想访问文件的哪个点，就访问文件的哪个点(任意)。

特征：既可读、又可写、还可追加，不会覆盖原有文件内容。但它只能访问文件。

举例说明1：

```
public class Test {  
    public static void main(String[] args) throws IOException {  
        RandomAccessFile raf = new RandomAccessFile("f:/1.txt", "rw");  
        byte[] buff = new byte[1024];  
        int hasRead = -1;  
        while ((hasRead = raf.read(buff)) > 0) {  
            System.out.println(new String(buff, 0, hasRead));  
        }  
        raf.close();  
    }  
}
```

举例说明2：

```
/**  
 * @author lhy  
 * @description 向文件中的指定位置插入内容  
 */  
public class Test {  
    public static void main(String[] args) {  
        RandomAccessFile raf = null;  
        try {  
            raf = new RandomAccessFile("f:/1.txt", "rw");  
            insertContent(raf, 100, "Java面向对象之I/O流之RandomAccessFile的使用");  
        } catch (IOException e) {  
            e.printStackTrace();  
        } finally {  

```

```
        try {
            raf.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

public static void insertContent(RandomAccessFile raf, int pos,
    String content) {
    ByteArrayOutputStream bos = null;
    try {
        bos = new ByteArrayOutputStream();
        raf.seek(pos); // 把记录指针移到要插入的地方
        byte[] buff = new byte[1024];
        int hasRead = -1;
        // 将raf从pos位置开始、直到结尾所有的内容
        while ((hasRead = raf.read(buff)) > 0) {
            bos.write(buff, 0, hasRead); // 将读取的数据(从pos位置开始)放入
bos中

        }
        raf.seek(pos); // 再次将记录指针移到要插入的地方
        raf.write(content.getBytes()); // 写入要插入的内容
        raf.write(bos.toByteArray()); // 写入之前保存的内容

    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        try {
            bos.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
}
```

结束语

有关Java中的IO流类比较多，而且方法大同小异，大家有空可以多查查API文档。今天就讲到这，明天开始讲Java面向对象之序列化机制及版本。

1.12 Java学习系列(十二)Java面向对象之序列化机制及版本

发表时间: 2013-11-11 关键字: java, apple, 编程, jdk, JVM

序列化：内存中的Java对象<——>二进制流

目的：a)有时候需要把对象存储到外部存储器中持久化保存，b)还有时候，需要把对象通过网络传输。

可序列化的对象，Java要求可序列化的类实现下面两个接口之一。

——Serializable：接口只是一个标记性的接口，实现该接口无需实现任何方法；——Externalizable实现该接口需要实现方法。

序列化的IO流：

ObjectInputStream ——负责从二进制流“恢复”对象-->从文件中提取对象；ObjectOutputStream ——负责将内存中的对象写入磁盘

举例说明1(注意：一定要实现Serializable接口)：

```
public class Test {  
    public static void main(String[] args) {  
        Apple apple = new Apple("Xx苹果", "红色", 2.3);  
        // System.out.println(apple);  
        // 这里利用了JDK7里面的try()自动关闭资源,好处是不用手动关闭oos  
        try (ObjectOutputStream oos = new ObjectOutputStream(  
            new FileOutputStream("f:/1.txt"));) {  
            oos.writeObject(apple);  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}  
  
class Apple implements Serializable {
```

```
private String name;
private String color;
private double weight;

public Apple() {
}

public Apple(String name, String color, double weight) {
    this.name = name;
    this.color = color;
    this.weight = weight;
}

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public String getColor() {
    return color;
}

public void setColor(String color) {
    this.color = color;
}

public double getWeight() {
    return weight;
}

public void setWeight(double weight) {
    this.weight = weight;
}
```

```
@Override
public String toString() {
    return "Apple [color=" + color + ", name=" + name + ", weight="
        + weight + "]";
}
}
```

而读取文件中的对象就更简单了(下面省略了上面的Apple类)：

```
public class Test {
    public static void main(String[] args) {
        try (
            ObjectInputStream ois = new ObjectInputStream(new FileInputStream(
                "f:/1.txt"));) {
            System.out.println(ois.readObject().toString());
        } catch (ClassNotFoundException | IOException e) {
            e.printStackTrace();
        }
    }
}
```

引用变量的序列化机制：

- A. 引用变量所引用的对象的所有属性都应该是可序列化的。
- B. 如果要序列化的对象是之前已经序列化的，此时系统序列化一个编号。

这种序列化机制，就是为了保存磁盘里的二进制流与内存中的对象是对应的。transient：用于修饰实例成员变量(不能与static修饰符同时使用)。--用于指定被修饰的field不会被序列化。好处：比如银行卡账号、密码就不应该被序列化出来。【注意】由于static修饰的类变量存储在类信息中，并不存储在对象里，所以有static修饰的类变量不能被序列化。

自定义序列化类：

```
/**
 * @author lhy
 * @description 自定义序列化类
 */
class User implements Serializable {
    private static final long serialVersionUID = 546525067577254190L;
```

```
private String account;
private String password;

public User() {

}

public User(String account, String password) {
    this.account = account;
    this.password = password;
}

public String getAccount() {
    return account;
}

public void setAccount(String account) {
    this.account = account;
}

public String getPassword() {
    return password;
}

public void setPassword(String password) {
    this.password = password;
}

@Override
public String toString() {
    return "User [account=" + account + ", password=" + password + "]";
}

// 下面两个方法,提供给系统调用,系统会调用者两个方法完成实际的序列化
private void writeObject(ObjectOutputStream out) throws IOException {
    // 序列化User的两个属性
    out.writeUTF(account);
}
```

```
        out.writeUTF(new StringBuilder(password).reverse().toString());
    }

    private void readObject(ObjectInputStream in) throws IOException,
        ClassNotFoundException {
        account = in.readUTF();
        password = new StringBuilder(in.readUTF()).reverse().toString();
    }
}

public class Test {
    public static void main(String[] args) {
        User user = new User("张三", "123");
        ObjectOutputStream oos = null;
        try {
            oos = new ObjectOutputStream(new FileOutputStream("f:/1.txt"));
            oos.writeObject(user);
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                oos.close();
            } catch (Exception e2) {
                e2.printStackTrace();
            }
        }
    }
}
```

读取文件中的对象：

```
public class Test {
    public static void main(String[] args) {
        ObjectInputStream ois = null;
        try {
            ois = new ObjectInputStream(new FileInputStream("f:/1.txt"));
```

```
        User user = (User)ois.readObject();
        System.out.println(user.toString());
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        try {
            ois.close();
        } catch (Exception e2) {
            e2.printStackTrace();
        }
    }
}
```

运行一下，我们可以看到输出：User [account=张三, password=123]

自定义(稳定)序列化：可以借助于“定制序列化”对属性进行一些“加密”。

【版本号】当我们的类经常使用时，有时候系统无法确定“发序列化”是的class文件是否还正确。--建议显式为“可序列化”指定一个版本号。--因为系统默认的版本号不稳定(经常改变)。serialver.exe -专门用来查看类的版本号。用法：serialver 序列化的类。--当我们修改了类时，记得要修改版本号。

结束语：

有关Java中的序列化今天就讲到这里，明天开始学习Java面向对象之界面编程。

1.13 Java学习系列(十三)Java面向对象之界面编程

发表时间: 2013-11-11 关键字: java, 编程, jdk

Java的界面编程

Java在客户端上表现并不突出，客户端往往都是局限在windows平台。AWT(JDK1.0发布，Sun希望在所有平台上都能运行)，它并未为界面提供实现，直接调用的是操作系统上相应的界面组件，AWT只能使用各操作系统上界面组件的交集。Swing为绝大部分的界面组件提供了实现，这些组件都是直接回执在空白区域上，Swing自己实现了这些界面组件，因此Swing无需使用各操作系统上界面组件的交集，Swing的UI界面更加统一。

先看几个GUI工具类：

Component：一切【界面组件】的祖先。

MenuComponent：一切【菜单组件】的祖先。

Container(Window-顶级窗口/Panel-不能独立存在)：它既可以是普通组件，也可以是存放组件的容器。

【注意】：Swing组件中除了JFrame、JDialog之外，都是轻量级组件。

在AWT里画图：创建Canvas或panel的子类，重写它的onPaint(Graphic g)

Graphic --相当于一个画笔，它们都是空白“矩形区域”。

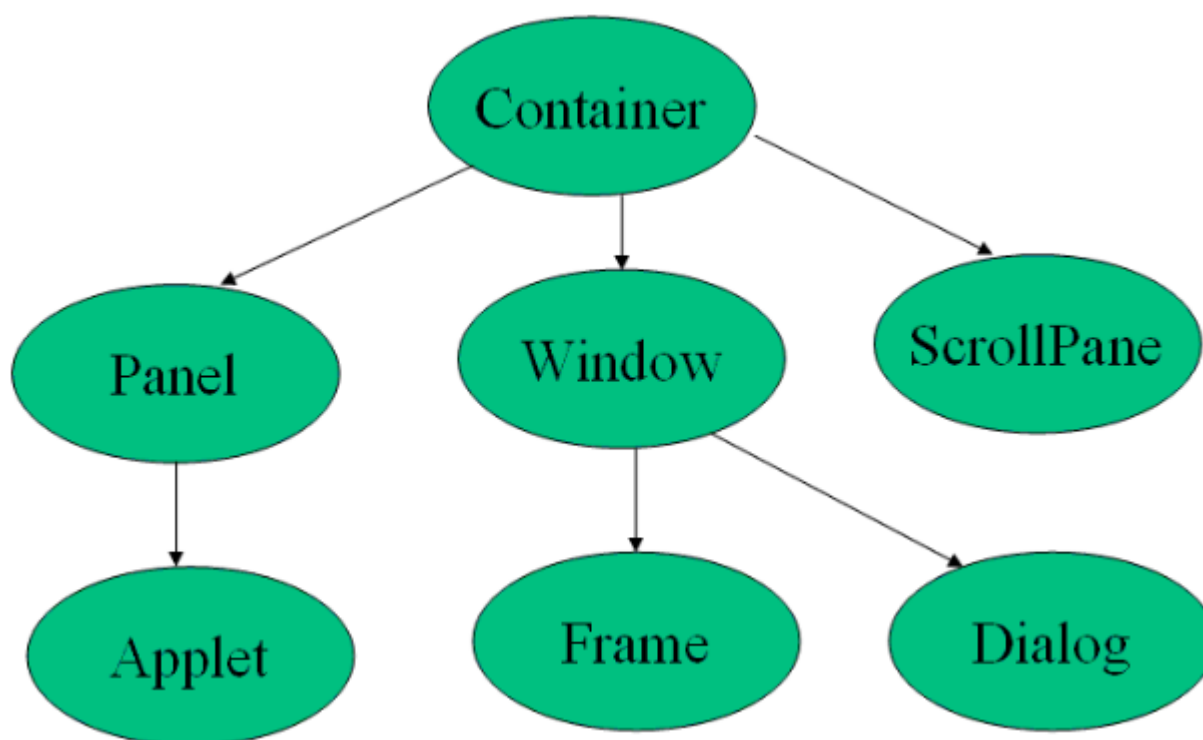
JPanel，也是一个空白的“矩形区域”，它多了一个“双缓冲”机制。

双缓冲：当我们想往某个“组件”上绘制内容时，程序先在“内存”中绘制一张图片。

----这样就避免了在组件上一个一个地绘制，从而可提高性能。

ImageIO：专门用来做图片输出。UUID：生成随机字符串。

window可以独立存在，而Panel不能。



布局管理器：如果不使用布局管理器，而是直接通过x, y, width, height来控制组件的大小，存在弊端：a)当窗口大小改变时，界面变得非常难看；b)当改变运行平台时，界面无法自适应。而使用布局管理器后，我们无需显式控制组件的大小和外观，只要选择合适的布局管理器即可。

FlowLayout是Panel和Applet的默认管理器

```
public class FlowLayoutDemo {  
    public static void main(String[] args) {  
        Frame frame = new Frame("FlowLayoutDemo");  
        FlowLayout layout = new FlowLayout(FlowLayout.LEFT, 20, 30);  
        frame.setBounds(80, 80, 100, 200);  
        frame.setLayout(layout);  
        frame.add(new Button("确定"));  
        frame.add(new Button("取消"));  
        frame.setVisible(true);  
    }  
}
```

BorderLayout将Container分为EAST、SOUTH、WEST、NORTH、CENTER五个区域，Component可以放置在这五

个区域的一个位置，BorderLayout是Frame、Dialog的默认管理器。【注意】每个区域只能放一个组件。由于awt基本已经被淘汰，所以有关界面方面的知识点不会讲那么多。

JTable使用举例：

```
public class JTableDemo {  
    public static void main(String[] args) {  
        JFrame jf = new JFrame("MyTable");  
        String[] titles = new String[] { "姓名", "年龄", "性别" };  
        String[][] data = new String[][] {  
            new String[] { "zhangsan", "20", "男" },  
            new String[] { "lisi", "21", "男" },  
            new String[] { "wangwu", "23", "女" },  
            new String[] { "zhaoliu", "22", "女" }, };  
        JTable table = new JTable(data, titles);  
        jf.add(new JScrollPane(table));  
        jf.pack();  
        jf.setVisible(true);  
    }  
}
```

JDialog使用实例：

```
public class JDialogDemo {  
    JFrame jf = new JFrame("My对话框");  
    JButton bt1 = new JButton("打开mode对话框");  
    JButton bt2 = new JButton("打开非mode对话框");  
    JDialog jDialog1 = new JDialog(jf, "模式对话框", true);  
    JDialog jDialog2 = new JDialog(jf, "非模式对话框", false);  
  
    public void init() {  
        jf.add(bt1);  
    }  
}
```

```
jf.add(bt2);
jf.setLayout(new FlowLayout(FlowLayout.LEFT));
bt1.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent arg0) {
        jDialog1.add(new Button("xxx"));
        jDialog1.show();
    }
});
bt2.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent arg0) {
        jDialog2.show();
    }
});
jf.pack();
jf.setVisible(true);
}

public static void main(String[] args) {
    new JDialogDemo().init();
}
}
```

JDialog读取文件内容：

```
public class JDialogDemo {
    JFrame jf = new JFrame("对话框测试");
    JButton bn = new JButton(" 打开文件");
    JFileChooser jfc = new JFileChooser("f:/");
    JTextArea jta = new JTextArea(10, 80);
```

```
public void init() {
    jf.add(new JScrollPane(jta)); // 放在中间
    JPanel jp = new JPanel();
    jp.add(bn);
    jp.setLayout(new FlowLayout());
    jf.add(jp, BorderLayout.SOUTH);
    bn.addActionListener(new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent e) {
            jfc.showOpenDialog(bn);
            File file = jfc.getSelectedFile();
            BufferedReader br = null;
            try {
                jta.setText("");
                br = new BufferedReader(new InputStreamReader(
                    new FileInputStream(file)));
                String line = null;
                while ((line = br.readLine()) != null) {
                    jta.append(line + "\n\r");
                }
            } catch (IOException ex) {
                ex.printStackTrace();
            }
        }
    });
    jf.pack();
    jf.setVisible(true);
}

public static void main(String[] args) {
    new JDialogDemo().init();
}
}
```

事件编程：一个有UI的程序，无非就是new——>add....，而Java的事件处理模式，采用委托类的事件处理。

委托类的事件处理：a) 当事件源发生事件时，事件源不会对事件进行处理，他只是发出一个事件，事件会传给监听器，监听就【只能通过】事件对象来获取关于事件的详细信息，事件监听器都要实现一个【特定】的方法。【事件源】一切组件都可能是事件源。【事件】无需我们理会，类似的发出事件的过程也无需我们操心。【监听器】需要我们实现。监听器要实现相应的接口(规范)。不同事件，就会有相应的监听器，需要为事件源上不同类型的事件，注册(声明)相应的监听器。

实例说明：

```
public class ActionListenerDemo {
    public void init() {
        JFrame jf = new JFrame("事件监听实例");
        JButton jbButton = new JButton("点击");
        jf.setLayout(new FlowLayout(FlowLayout.LEFT));
        jf.add(jbButton);
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.setSize(300, 200);
        jf.pack();
        jf.setVisible(true);
        jbButton.addActionListener(new MyActionListener());
    }

    class MyActionListener implements ActionListener {
        @Override
        public void actionPerformed(ActionEvent e) {
            System.out.println("事件在 " + e.getWhen() + "发生");
        }
    }

    public static void main(String[] args) {
        new ActionListenerDemo().init();
    }
}
```

菜单：AWT的菜单中，MenuBar包含多个Menu，Menu包含多个MenuItem。而Swing的菜单要比AWT的菜单的功能更加强大，比如Swing菜单支持图标菜单。

ImageIO:

使用 ImageIO 类的静态方法可以执行许多常见的图像 I/O 操作。

此包包含一些基本类和接口，有的用来描述图像文件内容（包括元数据和缩略图）(IIOImage)；有的用来控制图像读取过程（ImageReader、ImageReadParam 和 ImageTypeSpecifier）和控制图像写入过程（ImageWriter 和 ImageWriteParam）；还有的用来执行格式之间的代码转换 (ImageTranscoder) 和报告错误 (IOException)。

举例说明(生成随机验证码)：

```
public class Test {  
    public static void main(String[] args) throws IOException {  
        Random rand = new Random();  
        // 在内存中创建一张真彩色图片  
        BufferedImage image = new BufferedImage(150, 25,  
            BufferedImage.TYPE_4BYTE_ABGR);  
        Graphics2D g = (Graphics2D) image.getGraphics();// 获取一支画笔  
        g.setColor(Color.RED);  
        g.setFont(new Font("Arial Black", Font.ITALIC, 16));  
        StringBuffer sb = new StringBuffer();  
        for (int i = 0; i < 6; i++) {  
            char c = (char) (rand.nextInt(26) + 65);  
            sb.append(c);  
            double angle = (rand.nextDouble() * (-Math.PI / 6)) + Math.PI / 12;  
            g.rotate(angle, i * 24 + 18, 25);// 控制旋转,angle为旋转角度,后面两个参  
            数用来指定旋转中心  
            g.drawString(c + "", i * 22 + 10, 15);  
            g.rotate(-angle, i * 24 + 18, 25);// 再转回去  
        }  
        System.out.println(sb.toString());  
        // 生成的图片与src在同一目录  
        ImageIO.write(image, "png", new File(UUID.randomUUID() + ".png"));  
    }  
}
```

结束语：事件编程的重点就是实现事件处理器，重点就是实现事件的处理方法。有关Java中的界面编程就讲到这，内容很多，不可能面面俱到，而且JavaSE中的界面等学了Java web后基本不用，而且安卓的界面更有优势，所以JavaSE中的界面编程就将这些，了解一些常用的组件即可，不必死记。明天开始讲Java中的感觉非常有用的部分——线程。

1.14 Java学习系列(十四)Java面向对象之细谈线程、线程通信(上)

发表时间: 2013-11-24 关键字: java, thread, 多线程, jvm

线程与进程的关系：

进程 --运行中的程序。进程有如下特征：

1).独立性。拥有自己的资源，拥有自己独立的内存区。

通常来说，一个进程的内存空间，是不允许其他进程访问的。

但像Windows，如A进程可以通过某种方式修改其他进程的内存值。

2).动态性。程序是静止的，运行起来才叫进程。

3).并发性。一个操作系统可以同时“并发(concurrent)”运行多个进程。

线程 --进程中的“并发(concurrent)”执行流，轻量级进程。

线程与进程的典型区别：Process(进程)是有独立内存的，因此创建Process的成本比创建线程的成本高。

什么是“并发”？什么是“并行”？

1) 并发：即使只有一个CPU，多个进程、或多个线程在CPU上【快速轮换】的执行。在同一个时刻，只有与CPU个数相同的进程真正在执行，其他进程都处于等待状态。--对用户来说，[感觉]是多个进程在同时执行。

2) 并行(Parallel)：必须有一个以上的cpu，在同一时刻，至少有与CPU个数相同的进程[真正]在执行。

多线程的好处：

1) 功能上类似多进程；

2) 创建成本低，效率高；

3) 所有线程共享进程的内存，因此线程之间的通信非常方便。

Java创建多线程的方法(3种)：(注意 Java默认有个main方法主线程的执行体)

启动线程：调用Thread对象的start()方法，千万不要调用run()方法。就是普通方法的调用，就不会启动多线程了。

a)继承Thread，重写一个run()方法，---这个run方法就是线程执行体(就是该线程将要做的事情)。

举例说明1：

```
public class Test extends Thread {  
    @Override  
    public void run() {  
        for (int i = 0; i < 100; i++) {  
            // Thread.currentThread()用于获取当前正在运行的线程  
            System.out.println(Thread.currentThread().getName() + ",i=" + i);  
        }  
    }  
}
```

```
        }  
    }  
  
    public static void main(String[] args) {  
        new Test().start();// 创建匿名实例并启动线程  
    }  
}
```

b)实现Runnable接口，重写run方法。--推荐

举例说明2：

```
public class Test implements Runnable {  
    @Override  
    public void run() {  
        for (int i = 0; i < 100; i++) {  
            // Thread.currentThread()用于获取当前正在运行的线程  
            System.out.println(Thread.currentThread().getName() + ",i=" + i);  
        }  
    }  
  
    public static void main(String[] args) {  
        new Thread(new Test()).start();// 这里把Runnable对象包装成Thread对象。  
    }  
}
```

c)实现Callable（就是Runnable增强版），重写call方法(有返回值，可以声明抛出异常)。

举例说明3：

```
public class Test implements Callable<Integer> {  
    @Override  
    public Integer call() throws Exception {  
        for (int i = 0; i < 100; i++) {  
            // Thread.currentThread()用于获取当前正在运行的线程  
            System.out.println(Thread.currentThread().getName() + ",i=" + i);  
        }  
    }  
}
```

```
        return 100;
    }

    public static void main(String[] args) {
        // 将Callable包装成FutureTask,再包装成Thread,最后启动线程
        new Thread(new FutureTask<Integer>(new Test())).start();
    }
}
```

创建线程方式的对比，创建线程的方式可分为2类：

1. 继承Thread类；2. 实现Runnable或Callable接口。

总的来说，实现接口的方式更好，原因如下：

- 1.实现接口之后，依然可以继承其他类；但如果继承了Thread类，就无法继承其他类了。
- 2.实现接口时，可以让多个线程共享同一个Runnable对象。可以更好的实现代码与数据的分离，形成更清晰的逻辑。

线程的状态(当调用start()方法之后，只是启动了线程，线程并不会立即执行)：

新建：刚刚创建出来的Thread对象。

就绪：调用start()之后，处于就绪状态。

从就绪到运行：是不可控的，靠线程调度器来分配。

从就绪到运行：靠线程调度器来分配(yield())方法可以主动的让出cpu，进入就绪状态)。

阻塞：调用sleep()、IO阻塞、等待同步锁、等待通知等将进入阻塞Blocked状态；sleep()时间到、IO阻塞解除、获取同步锁、收到通知后等将进入就绪状态。

正常死亡：线程执行体执行完成；遇到了未捕获的异常。

控制线程的方法：

Join线程：启动一条线程，多条线程并发执行，被joined线程必须先执行完成。

举例说明：

```
class JoinThread extends Thread {
    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            // Thread.currentThread()用于获取当前正在运行的线程
        }
    }
}
```

```
        System.out.println(Thread.currentThread().getName() + ",i=" + i);
    }
}

public class Test {
    public static void main(String[] args) throws InterruptedException {
        JoinThread jt1 = new JoinThread();
        JoinThread jt2 = new JoinThread();
        for (int i = 0; i < 100; i++) {
            System.out.println("主线程正在执行：i=" + i);
            if (i == 20) {
                // 主线程执行到此处时,必须等到jt1、jt2执行完后,主线程才能继续向下
                // 执行。

                jt1.start();
                // 將jt1这条进程join进来, 等待jt1线程终止。
                jt1.join();
                jt2.start();
                jt2.join();
            }
        }
    }
}
```

后台线程(Daemon Thread)：又称守护线程、精灵线程。如果所有的前台线程结束，它会自动死亡。JVM的垃圾回收器就是一个典型的后台进程。调用Thread对象的setDaemon(true)方法可将指定线程设置为后台线程。

线程暂停：Thread.sleep(100)：让线程暂停100ms，并且进入阻塞状态。--推荐(更稳定)

线程让步：Thread.yield()：让线程让出cpu，并进入就绪状态。

改变线程的优先级：优先级越高，线程会获得更多的执行机会。

举例说明：(优先级高的先执行)

```
class PriorityThread extends Thread {
    @Override
```

```
public void run() {
    for (int i = 0; i < 100; i++) {
        // Thread.currentThread()用于获取当前正在运行的线程
        System.out.println(Thread.currentThread().getName() + ",i=" + i);
    }
}

}

public class Test {
    public static void main(String[] args) throws InterruptedException {
        PriorityThread jt1 = new PriorityThread();
        jt1.setPriority(Thread.MIN_PRIORITY);
        PriorityThread jt2 = new PriorityThread();
        jt2.setPriority(Thread.MAX_PRIORITY);
        jt1.start();
        jt2.start();
        System.out.println("~~~~~主线程结束~~~~~");
    }
}
```

1.15 Java学习系列(十五)Java面向对象之细谈线程、线程通信(下)

发表时间: 2013-11-24 关键字: java, thread, jdk, 编程, JVM

竞争资源(共享资源)：如果有多条线程需要并发访问、并修改某个对象，该对象就是“竞争资源”。为了避免多个线程“自由竞争”修改共享资源所导致的不安全问题。

线程同步(像Vector、Hashtable等都是线程安全的)：

解决线程异步有两种方式：

- 1)同步代码块(需要显式的指定同步监视锁)；
- 2).同步方法(相当于使用方法的调用者，如果方法是实例方法，相当于this为同步监视锁；如果方法是类方法，相当于类作为同步监视锁)。

它们的实现机制是完全相同的，当线程要进入被“同步监视锁”监视的代码之前，线程必须先获得“同步监视锁”，这样就可以保证在任意一个时刻，只有一条线程能进入被“同步监视锁”监视的代码。从程序逻辑来看，选择“竞争资源”作为同步监视锁。

举例说明1--同步代码块(由于代码都是些属性,所以这里只抽取了片段，掌握思想即可)：

```
// 同步代码块,synchronized后的括号中的对象被称为同步锁
synchronized (account) {
    if (account.getBalance() > drawAmount) {
        System.out.println("取钱成功,吐出钱数：" + drawAmount);
        account.setBalance(account.getBalance() - drawAmount);
        System.out.println("还剩余额为：" + account.getBalance());
    } else {
        System.out.println("取钱失败,余额不足！");
    }
}
```

举例说明2：--同步方法，以this作为同步监视锁。

```
public synchronized void draw(double drawAmount){
    if (getBalance() > drawAmount) {
        System.out.println("取钱成功,吐出钱数：" + drawAmount);
        setBalance(getBalance() - drawAmount);
        System.out.println("还剩余额为：" + getBalance());
    }
}
```

```
        } else {  
            System.out.println("取钱失败,余额不足!");  
        }  
    }  
}
```

线程同步的关键：任何同步监视器监视的代码之前，必须先对同步监视器加锁。

何时释放对【同步监视器】的加锁？

- 1.同步代码块或同步方法执行完成。
- 2.在代码中遇到了break、return语句跳出该代码块。
- 3.执行同步代码块或同步方法时遇到未捕获的异常时。
- 4.调用了同步监视器的wait()方法。

【注意】使用sleep()、yield()都不会释放。

线程通信：

- 1.如果不加控制，多个线程“自由”地并发执行。
- 2.可以通过同步，来解决多个线程并发访问竞争资源的问题。线程安全，必然降低性能。
- 3.如果希望线程之间能更有序地执行。

线程组ThreadGroup与未处理的异常：

怎样把线程放入指定的线程组中呢？--在创建一个Thread实例时，通过传入的ThreadGroup对象，即可将该线程放入指定的线程组中，进而通过线程组对这批线程进行整体的管理。

ThreadGroup提供了如下两个方法：setDaemon(boolean daemon) 控制将线程组本身都设置为后台线程组，并不是将它所包含的线程设为后台线程。setMaxPriority(优先级)：它是设置该线程组已有的线程的优先级不会受影响，对以后新添加的线程的优先级才会有影响。

对线程异常进行处理：在JDK1.5之前，系统会自动回调它所在线程组的uncaughtException(Thread t,Throwable e)方法来修复该异常；在JDK1.5之后，线程允许自行设置“异常处理器”，无需线程组。public static void setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh)：为所有线程设置默认的异常处理器。public void setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh)：为当前线程实例设置异常处理器。

举例说明1(jdk1.5之前)：

```
public class ThreadExceptionTest implements Runnable {  
    @Override  
    public void run() {  
        for (int i = 0; i < 100; i++) {  
            System.out.println(Thread.currentThread().getName() + "--->" + i  
                                / (i - 20));  
        }  
    }  
  
    public static void main(String[] args) {  
        // 创建一个线程组  
        ThreadGroup tg = new ThreadGroup("mytg") {  
            @Override  
            public void uncaughtException(Thread t, Throwable e) {  
                System.out.println(t.getName() + "出现了异常,信息  
是：" + e.getMessage());  
            }  
        };  
        ThreadExceptionTest test = new ThreadExceptionTest();  
        new Thread(tg, test).start();  
    }  
  
}
```

举例说明2(jdk1.5之后)：

```
public class ThreadExceptionTest implements Runnable {  
    @Override  
    public void run() {  
        for (int i = 0; i < 100; i++) {  
            System.out.println(Thread.currentThread().getName() + "--->" + i  
                                / (i - 20));  
        }  
    }  
  
}
```

```
public static void main(String[] args) {  
    // 为所有线程设置默认的异常处理器  
    Thread.setDefaultUncaughtExceptionHandler(new UncaughtExceptionHandler() {  
        @Override  
        public void uncaughtException(Thread t, Throwable e) {  
            System.out.println(t.getName() + "出现了异常,信息  
是：" + e.getMessage());  
        }  
    });  
    ThreadExceptionTest test = new ThreadExceptionTest();  
    new Thread(test).start();  
}  
}
```

线程池(Pool)：池的本质，就是一种“缓存”技术。是否要缓存一个对象，要看该对象的创建成本。

Executors --创建线程池，线程工厂的工具类。ExecutorService：它就是线程池。

缓存的本质：牺牲空间(内存)来换取时间。线程对象的创建成本比较大(尽管比创建进程的成本小的多，但相对普通java对象，Thread的创建成本依然较大)，为解决这个问题，我们用线程池。

编程步骤：

- 1.通过Executors的静态工厂方法创建ExecutorService或ScheduledExecutorService
- 2.调用ExecutorService的方法提交线程即可。
- 3.调用线程池的.shutdown方法关闭线程池。

举例说明1:

```
public class ThreadPoolTest implements Runnable {  
    @Override  
    public void run() {  
        for (int i = 0; i < 100; i++) {  
            System.out.println(Thread.currentThread().getName() + ",i=" + i);  
        }  
    }  
}  
  
public static void main(String[] args) {  
    ExecutorService es = Executors.newFixedThreadPool(10);  
    es.submit(new ThreadPoolTest());  
    es.shutdownNow();// 关闭线程池
```

```
    }  
}
```

举例说明2:

```
public class ThreadPoolTest implements Runnable {  
    @Override  
    public void run() {  
        for (int i = 0; i < 100; i++) {  
            System.out.println(Thread.currentThread().getName() + ",i=" + i);  
        }  
    }  
  
    public static void main(String[] args) {  
        ScheduledExecutorService es = Executors.newScheduledThreadPool(10);  
        // 延迟5s,以后每隔2s执行一次run方法  
        es.scheduleAtFixedRate(new ThreadPoolTest(), 5, 2, TimeUnit.SECONDS);  
    }  
}
```

1.16 Java学习系列(十六)Java面向对象之基于TCP协议的网络通信

发表时间: 2013-11-25 关键字: java, 互联网, 浏览器, mysql, xml

TCP/IP的网络分层模型：应用层(HTTP/FTP/SMTP/POPS...)，传输层(TCP协议)，网络层(IP协议,负责为网络上节点分配唯一标识)，物理层+数据链路层)。

IP地址用于标识网络中的一个通信实体，通常这个实体可以是一台主机，也可以是一台打印机，或者是路由器的某一个端口。而基于IP协议网络中传输的数据包，都必须使用IP地址来进行标识。

IP地址与端口：IP地址就是为网络上的每个物理节点(广义的)分配一个“门牌号”。通过IP地址，可以保证网络上的数据包能正确地找到每个物理节点，但每个物理节点上可能有多个应用程序在同时对外提供服务。端口：每个应用程序在网络上通信时，占用一个端口，相当于“房间号”，端口保证了物理节点的数据包能正确找到对应的应用程序。

端口的约定：(0~65535个端口)

0~1023：公用端口。80(HTTP)、21(FTP)、110(POP)...

1023~49152：应用程序端口。MySQL：3306；Oracle：1521

49152~65535：动态分配端口。

先了解几个常用的类：

InetAddress：此类表示互联网协议 (IP) 地址。它有两个子类：Inet4Address, Inet6Address。

InetSocketAddress：它代表了IP地址+端口号

```
public class Test {  
    public static void main(String[] args) {  
        try {  
            // InetAddress代表了IP地址  
            InetAddress address = InetAddress.getByAddress(new byte[] {  
                (byte) 192, (byte) 168, 0, 8 });  
            // 打印主机名  
            System.out.println(address.getHostName());  
            // 打印主机地址  
            System.out.println(address.getHostAddress());  
            // 测试是否可以到达该地址,有点类似于Ping  
            System.out.println(address.isReachable(3000));  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

```
    }  
}
```

两个工具类：URLEncoder、URLDecoder。在有些场景，无法传输和存储“非西欧文字”，此时就需要用到URLEncoder。典型的像Cookie，Cookie的值就不能是中文。

举例说明1：

```
public class URLEncoderDeCoderUtil {  
    public static void main(String[] args) {  
        String str = "Java学习系列(十六)Java面向对象之基于TCP协议的网络通信";  
        try {  
            // 对字符进行编码  
            str = URLEncoder.encode(str, "GBK");  
            System.out.println(str);  
            // 对字符进行解码  
            System.out.println(URLDecoder.decode(str, "GBK"));  
        } catch (UnsupportedEncodingException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

URL：代表一个网络地址。

URLConnection：代表与网络地址的连接。

HttpURLConnection：基于HTTP协议的网络连接。

举例说明2：

```
public class Test {  
    public static void main(String[] args) {  
        try {  
            URL url = new URL("http://localhost:8080/test/index.jsp");  
        }  
    }  
}
```

```
System.out.println("协议：" + url.getProtocol());
System.out.println("主机：" + url.getHost());
System.out.println("端口：" + url.getPort());
System.out.println("资源文件：" + url.getFile());
// 建立于远程URL地址之间的连接，
// 当我们的协议用的是http时,打开的连接实际上就是URLConnection
URLConnection conn = (URLConnection) url.openConnection();
conn.connect();// 建立于远程服务器的连接
BufferedReader br = new BufferedReader(new InputStreamReader(
    conn.getInputStream()));

String line = null;
// 读取页面资源
while ((line = br.readLine()) != null) {
    System.out.println(line);
}
} catch (IOException e) {
    e.printStackTrace();
}
}
}
```

破解密码简单演示：

- 1)准备密码字典:password.txt (文件每行随便写上几个密码就行)
- 2)登录页面片段：

```
<form action="loginPro.jsp" method="post">
    用户名：
    <input name="username" type="text" />
    密码：
    <input name="passwd" type="password" />
    <br />
    <input type="submit" value="登录" />
    <input type="reset" value="取消" />
```

```
<br />
</form>
```

3)登录处理页面片段：

```
<%
    String username = request.getParameter("username");
    String passwd = request.getParameter("passwd");
    if (username.equals("liu") && passwd.equals("123")) {
        out.print("登录成功！");
    } else {
        out.print("登录失败！");
    }
%>
```

4).程序实现代码：

【注意】连接要设置相应属性。可以打开Google浏览器进入处理页面后，按CTRL+SHIFT+I，将看到：

Elements	Resources	Network	Sources	Timeline	Profiles	Audits	Console
		x Headers Preview Response Cookies Timing					
loginPro.jsp /test		Request URL: http://localhost:8888/test/loginPro.jsp Request Method: POST Status Code: 200 OK					
style.css cpngackimfmofbokmjmlja		▼ Request Headers view source Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*; Accept-Encoding: gzip,deflate,sdch Accept-Language: zh-CN,zh;q=0.8 Connection: keep-alive					
page_context.js cpngackimfmofbokmjmlja							

```
public class Test {
    public static void main(String[] args) {
        try {
```

```
URL url = new URL("http://localhost:8888/test/loginPro.jsp");
BufferedReader br = new BufferedReader(new InputStreamReader(
    new FileInputStream("f:/password.txt")));
String passwd = null;
while ((passwd = br.readLine()) != null) {
    // 每次读取一行(字典文件),创建一次连接
    HttpURLConnection conn = (HttpURLConnection) url
        .openConnection();

    conn
        .setRequestProperty("Accept",
            "application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8");
    conn.setRequestProperty("Accept-Encoding", "gzip,deflate,sdch");
    conn.setRequestProperty("Connection", "keep-alive");
    conn.setDoInput(true);
    conn.setDoOutput(true);
    // 打开远程输出流,准备向服务器发送请求参数
    PrintStream ps = new PrintStream(conn.getOutputStream());
    ps.print("username=liu&passwd=" + passwd);
    ps.flush();
    // 从远程服务器读取响应
    BufferedReader br2 = new BufferedReader(new InputStreamReader(
        conn.getInputStream()));

    String line = null;
    while ((line = br2.readLine()) != null) {
        if (line.contains("登录成功")) {
            System.out.println("正确的密码为：" + passwd);
        }
    }
}
} catch (IOException e) {
    e.printStackTrace();
}
}
```

TCP协议：它是一种可靠的端对端的协议。这是因为它为两台计算机之间的连接起了重要的作用：当一台计算机需要与另一台计算机连接时，TCP协议会让它们建立一个连接 用于发送和接收数据的虚拟链路。TCP协议保证了数据包在传送中准备无误。

TCP协议使用重发机制：当一个通信实体发送一个消息给另一个通信实体后，需要收到另一个通信实体确认信息，如果没有收到另一个通信实体的确认信息，则会再次重发刚才发送的信息。通过这个重发机制，TCP协议向应用程序提供可靠的通信连接，使它能够自动适应网上的各种变化，即使在Internet暂时出现阻塞的情况下，TCP也能够保证通信的可靠。

1.17 Java学习系列(十七)Java面向对象之开发聊天工具

发表时间: 2013-11-25 关键字: java, socket, 编程, thread, JVM

TCP通信：

Socket --相当于“虚拟链路两端的插座”。Socket负责完成通信。

ServerSocket --它只负责“接收”连接。它用于产生Socket。

服务器端编程：

- 1) 创建ServerSocket 对象，该对象负责“接收”连接。
- 2) 如果客户端有连接，ServerSocket 对象调用accept()方法返回一个Socket。
- 3) 通过IO流读取对方的信息，也可向对方发送数据。

客户端编程：

- 1) new Socket()来建立与远程服务器的连接。
- 2) 通过IO流读取对方的信息，也可向对方发送数据。

举例说明1(简单通信)：

```
/**
 * @author lhy
 * @description 服务器端
 */
public class ServerTest {
    public static void main(String[] args) {
        PrintStream ps = null;
        try {
            // ServerSocket只负责“接收”连接,20000为端口号(标识该应用程序)
            ServerSocket ss = new ServerSocket(20000);
            System.out.println("服务器端等待连接...");
            // 接收连接,它会阻塞线程
            Socket socket = ss.accept();

            // -----下面统一面向IO编程-----//
            ps = new PrintStream(socket.getOutputStream());
            ps.println("ServerTest你好");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            ps.close();
        }
    }
}
```

```
/**
 * @author lhy
 * @description 客户端
 */
public class ClientTest {
    public static void main(String[] args) {
        BufferedReader br = null;
        try {
            Socket socket = new Socket("192.168.0.8", 20000);

            // -----下面统一面向IO编程-----//
            br = new BufferedReader(new InputStreamReader(socket
                .getInputStream()));

            String line = null;
            while ((line = br.readLine()) != null) {
                System.out.println(line);
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                br.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
    }  
}
```

举例说明2(控制台多人聊天):

服务器端：

```
public class ServerTest {  
    static Set<Socket> clients = Collections  
        .synchronizedSet(new HashSet<Socket>());  
  
    public static void main(String[] args) {  
        ServerSocket ss = null;  
        try {  
            ss = new ServerSocket(20000);  
            System.out.println("服务器端等待连接...");  
            while (true) {  
                // 接收连接,它会阻塞线程  
                Socket socket = ss.accept();  
                clients.add(socket);  
                // 启动线程  
                new ServerThread(socket).start();  
            }  
        } catch (IOException e) {  
            e.printStackTrace();  
        } finally {  
            try {  
                ss.close();  
            } catch (IOException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

```
// 单独封装一个线程类
class ServerThread extends Thread {
    private Socket socket;

    public ServerThread(Socket socket) {
        this.socket = socket;
    }

    @Override
    public void run() {
        BufferedReader br = null;
        try {
            while (true) {
                // -----下面统一面向IO编程
                -----//

                br = new BufferedReader(new InputStreamReader(socket
                    .getInputStream()));

                String line = null;
                while ((line = br.readLine()) != null) {
                    for (Socket s : ServerTest.clients) {
                        PrintStream ps = new PrintStream(s.getOutputStream());
                        ps.println(line); // 输出各个客户端对应的Socket
                    }
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                br.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

客户端：

```
public class ChatServer {
    // 定义一个线程安全的Set集合
    public static Set<Socket> clients = Collections
        .synchronizedSet(new HashSet<Socket>());

    public static void main(String[] args) throws IOException {
        // ServerSocket只负责“接受”连接,不能通信
        // 该服务器程序就在30002端口监听
        ServerSocket ss = new ServerSocket(20000);
        System.out.println("服务器等待连接...");
        while (true) { // 这样保证每个客户端有一条线程
            // 接受连接。它会阻塞线程
            Socket socket = ss.accept();// 只要连接成功,它会返回socket
            clients.add(socket);// 每次客户端连接进来,就将该客户端添加到clients集合中
            System.out.println("当前用户数量：" + clients.size());
            new ServerThread(socket).start();
        }
    }
}

class ServerThread extends Thread {
    private Socket socket;

    public ServerThread(Socket socket) {
        this.socket = socket;
    }

    @Override
    public void run() {
        // -----原来读文件,现在改为读网络,只要改节点
        try {
            BufferedReader socketBr = new BufferedReader(new InputStreamReader(
```

```
        socket.getInputStream()));  
String line = null;// line代表从网络中读取数据  
while ((line = socketBr.readLine()) != null) {  
    for (Iterator<Socket> it = ChatServer.clients.iterator(); it  
        .hasNext();) {  
        Socket s = it.next();  
        try {  
  
            PrintStream ps = new PrintStream(s.getOutputStream());  
            ps.println(line);// 输出到各客户端对应的socket  
        } catch (SocketException ex) {  
            it.remove();// 捕获到该socket的异常,即表明  
Socket已经断开  
  
            System.out.println("当前用户数  
量：" + ChatServer.clients.size());  
  
            // ex.printStackTrace();  
        }  
    }  
}  
}  
} catch (SocketException se) {  
    ChatServer.clients.remove(socket);  
    System.out.println("当前用户数量：" + ChatServer.clients.size());  
} catch (Exception e) {  
    e.printStackTrace();  
}  
}  
}
```

小结：

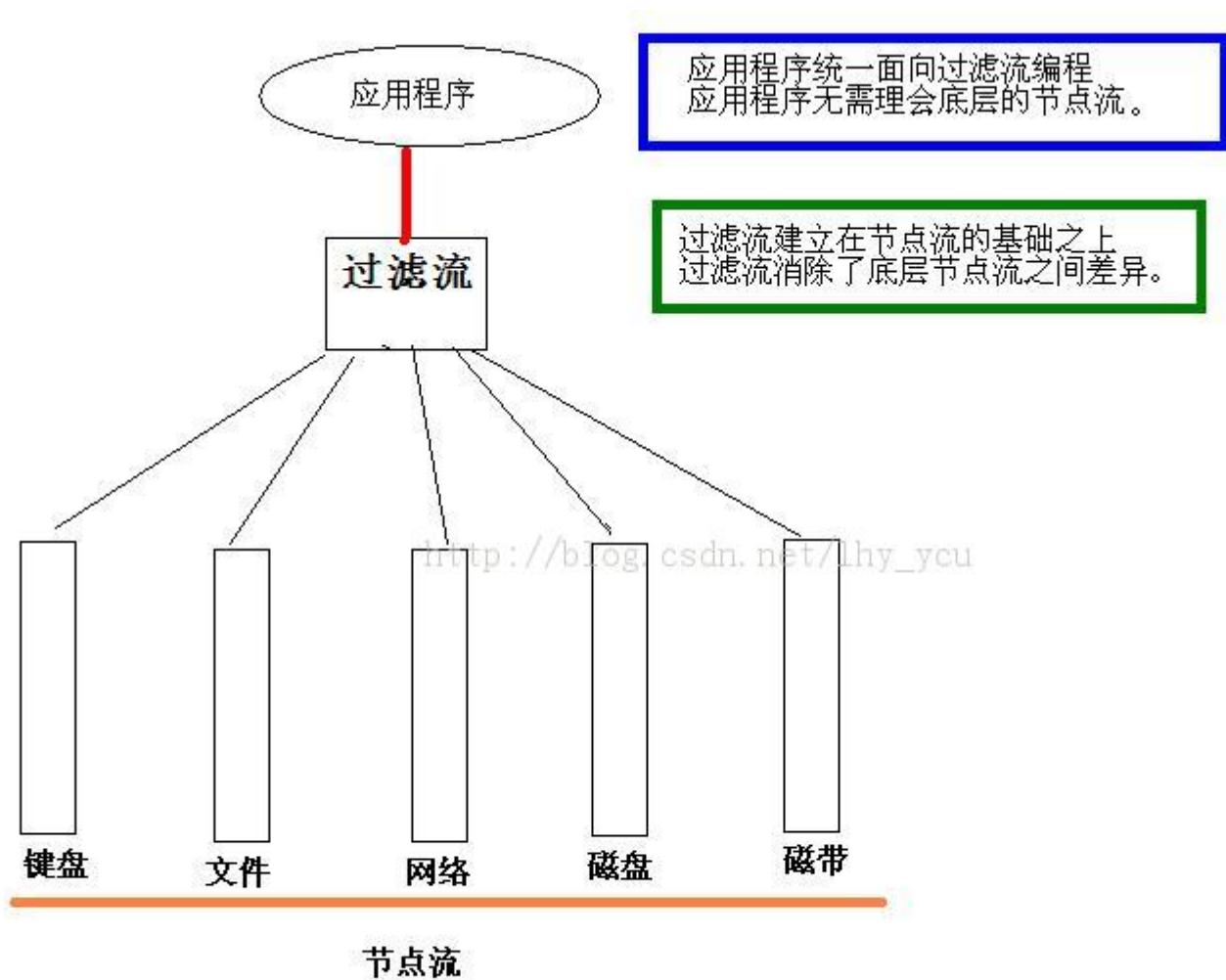
System.in ：读取键盘输入。包装方法：BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));

socket.getInputStream() ：读取网络。包装方法：BufferedReader socketBr = new BufferedReader(new
InputStreamReader(socket.getInputStream()));

socket.getOutputStream()：写入(输出到)网络。包装方法：PrintStream socketOut = new PrintStream(socket.getOutputStream());

System.out：输出到屏幕。

下面以之前IO讲的一张图来结束：



1.18 Java学习系列(十八)Java面向对象之基于UDP协议的网络通信

发表时间: 2013-11-26 关键字: java, JVM

UDP协议：无需建立虚拟链路，协议是不可靠的。

A节点以DatagramSocket发送数据包，数据报携带数据，数据报上还有目的地地址，大部分情况下，数据报可以抵达；但有些情况下，数据报可能会丢失 --丢失了也不管。

先了解2个类：

DatagramSocket：相当于“码头”，此类表示用来发送和接收数据报的套接字。

DatagramPacket：代表数据报。

举例说明1：

服务器端

```
public class SimpleUDPServer {  
    final static int SERVER_PORT = 30000;  
    final static int PACKET_SIZE = 4092;  
  
    public static void main(String[] args) {  
        DatagramSocket datagramSocket = null;  
        DatagramPacket datagramPacket = null;  
        try {  
            while (true) {  
                // 创建datagramSocket,准备用于发送和接收数据报  
                datagramSocket = new DatagramSocket(SERVER_PORT);  
                datagramPacket = new DatagramPacket(new byte[PACKET_SIZE],  
                    PACKET_SIZE);  
                // 使用一个空的packet去装datagramSocket接收到的数据  
                datagramSocket.receive(datagramPacket);  
                // 将datagramPacket中接收到的字节数组转换为字符串,然后输出  
                System.out.println(new String(datagramPacket.getData(), 0,  
                    datagramPacket.getLength()));  
            }  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

```
    }  
}
```

客户端：

```
public class SimpleUDPClient {  
    final static int SERVER_PORT = 30000;  
  
    public static void main(String[] args) {  
        DatagramSocket datagramSocket = null;  
        DatagramPacket datagramPacket = null;  
        try {  
            // 创建datagramSocket,准备用于发送和接收数据报  
            // datagramSocket使用动态端口(以便选择一个空闲的端口)  
            datagramSocket = new DatagramSocket();  
            String content = "Java学习系列(十八)Java面向对象之基于UDP协议的网络通信";  
  
            // 创建一个有数据、有目的地址的datagramPacket  
            datagramPacket = new DatagramPacket(content.getBytes(), content  
                .getBytes().length, InetAddress.getByAddress(new byte[]  
                    (byte) 192, (byte) 168, 0, 8 }), SERVER_PORT);  
  
            // 发送数据报  
            datagramSocket.send(datagramPacket);  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

MulticastSocket将数据报发送到“广播地址”，数据报会被自动“广播”到所有加入该IP地址的客户端。

TTL是控制数据可以跨过多少个网段。

- 当ttl为0时，说明该数据报只能停留在本机；
- 当ttl为1时，说明数据报只能停留在当前局域网。
- 当ttl为32时，说明数据报只能停留在本站点的网络。
- 当ttl为64时，说明数据报只能停留在本地区。
- 当ttl为128时，说明数据报只能停留在本大洲(如亚洲)。
- 当ttl为255时，说明数据报达到全球。

1.19 Java学习系列(十九)Java面向对象之数据库编程

发表时间: 2013-11-26 关键字: java, 编程, jdbc, mysql, jsp

JDBC (Java Data Base Connectivity : java数据库连接) : 它定义了一组标准的操作数据库的接口, 既然是接口, 那它就是一种规范, 是Java操作数据库的技术规范。

Java数据库编程有两步常用操作:

1.加载(或注册)JDBC驱动程序

Class.forName("com.mysql.jdbc.Driver"); 推荐这种方式, 不会对具体的驱动类产生依赖。

DriverManager.registerDriver(com.mysql.jdbc.Driver); 会造成DriverManager中产生两个一样的驱动, 并会对具体的驱动类产生依赖。

2.建立数据库Connection

Connection conn = DriverManager.getConnection(url, user, password); DriverManager是一个驱动管理器, 内部有一个驱动注册表 (Map结构), 可以向其注册多个JDBC驱动。

举例说明1:

```
/**
 * @author lhy
 * @description 数据库工具类
 */
public class DBUtil {
    // 创建连接
    public static Connection createConn() {
        Connection conn = null;
        try {
            // 加载驱动程序
            Class.forName("com.mysql.jdbc.Driver");
            // 获取连接(这里用户名为root, 密码为空)
            conn = DriverManager.getConnection(
                "jdbc:mysql://localhost:3306/test", "root", "");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

```
        }
        return conn;
    }

    // 创建回话,获取预处理语句 ( 可以防止sql语句注入 )
    public static PreparedStatement prepare(Connection conn, String sql) {
        PreparedStatement ps = null;
        try {
            ps = conn.prepareStatement(sql);
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return ps;
    }

    // 关闭连接
    public static void close(Connection conn) {
        if (conn != null) {
            try {
                conn.close();
                conn = null;
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }

    // 关闭回话
    public static void close(Statement st) {
        if (st != null) {
            try {
                st.close();
                st = null;
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
    }

    // 关闭查询结果集
    public static void close(ResultSet rs) {
        if (rs != null) {
            try {
                rs.close();
                rs = null;
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
}
```

下面用户表为例，调用Statement对象的executeQuery()方法或executeUpdate()方法，让DBMS 执行具体的SQL语句，以便对数据执行查询、增、删、改等操作；

我们先建好一张用户表(user，数据库为test)

主索引			
Id	int(11)	no	<auto_increment>
username	varchar(20)	no	
password	varchar(20)	no	

创建实体类(User):

```
public class User {
    private int id;
    private String username;
    private String password;

    public User() {
    }
}
```

```
public User(int id, String username, String password) {
    this.id = id;
    this.username = username;
    this.password = password;
}

public int getId() {
    return id;
}

public void setId(int id) {
    this.id = id;
}

public String getUsername() {
    return username;
}

public void setUsername(String username) {
    this.username = username;
}

public String getPassword() {
    return password;
}

public void setPassword(String password) {
    this.password = password;
}

@Override
public String toString() {
    return "User [id=" + id + ", password=" + password + ", username="
        + username + "]";
}
```

```
}
```

```
/**
 * @author lhy
 * @description 对用户的CRUD相关操作
 */
public class UserDao {
    // 添加用户
    public void add(User u) {
        Connection conn = DBUtil.createConn();
        String sql = "insert into user values(null,?,?)";
        PreparedStatement ps = DBUtil.prepare(conn, sql);
        try {
            ps.setString(1, u.getUsername());
            ps.setString(2, u.getPassword());
            ps.executeUpdate();
        } catch (SQLException e) {
            e.printStackTrace();
        }
        DBUtil.close(ps);
        DBUtil.close(conn);
    }

    // 根据Id删除用户
    public void deleteById(int id) {
        Connection conn = DBUtil.createConn();
        String sql = "delete from user where id=?";
        PreparedStatement ps = DBUtil.prepare(conn, sql);
        try {
            ps.setInt(1, id);
            ps.executeUpdate();
        } catch (SQLException e) {
            e.printStackTrace();
        }
        DBUtil.close(ps);
        DBUtil.close(conn);
    }
}
```

```
}

// 删除用户
public void delete(User u) {
    deleteById(u.getId());
}

// 更新用户
public void update(User u) {
    Connection conn = DBUtil.createConn();
    String sql = "update user set username=?,password=? where id=?";
    PreparedStatement ps = DBUtil.prepare(conn, sql);
    try {
        ps.setString(1, u.getUsername());
        ps.setString(2, u.getPassword());
        ps.setInt(3, u.getId());
        ps.executeUpdate();
    } catch (SQLException e) {
        e.printStackTrace();
    }
    DBUtil.close(ps);
    DBUtil.close(conn);
}

// 根据查询用户
public User loadById(int id) {
    Connection conn = DBUtil.createConn();
    String sql = "select * from user where id=?";
    PreparedStatement ps = DBUtil.prepare(conn, sql);
    User user = null;
    ResultSet rs = null;
    try {
        ps.setInt(1, id);
        rs = ps.executeQuery();
        if (rs.next()) {
            user = new User();
            user.setId(rs.getInt("Id"));
        }
    }
}
```

```
        user.setUsername(rs.getString("username"));
        user.setPassword(rs.getString("password"));
    }
} catch (SQLException e) {
    e.printStackTrace();
}
DBUtil.close(rs);
DBUtil.close(ps);
DBUtil.close(conn);
return user;
}
```

// 查询所有用户信息

```
public List<User> listUser() {
    Connection conn = DBUtil.createConn();
    String sql = "select * from user";
    PreparedStatement ps = DBUtil.prepare(conn, sql);
    List<User> list = new ArrayList<User>();
    ResultSet rs = null;
    try {
        rs = ps.executeQuery();
        while (rs.next()) {
            User user = new User();
            user.setId(rs.getInt("Id"));
            user.setUsername(rs.getString("username"));
            user.setPassword(rs.getString("password"));
            list.add(user);
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
    DBUtil.close(rs);
    DBUtil.close(ps);
    DBUtil.close(conn);
    return list;
}
```

客户端(测试类)：

```
public class TestUser {  
    public static void main(String[] args) {  
        User user = new User();  
        user.setUsername("张三");  
        user.setPassword("123");  
        UserDao userDao = new UserDao();  
        // userDao.add(user); // 添加用户  
        // userDao.deleteById(1); // 删除Id为1的用户  
        // userDao.update(new User(2, "李四", "1234546")); // 更新Id为2的用户信息  
        // System.out.println(userDao.loadById(2)); // 查询Id为2的用户信息  
        // 列出所有的用户  
        List<User> list = userDao.listUser();  
        for (User u : list) {  
            System.out.println(u);  
        }  
    }  
}
```

Java的数据库编程比较简单，这里就不再赘述。

结束语：

Javase基础部分就到这了，之后会更新 Java的反射、注解、代理、设计模式、jsp、Struts、Hibernate、Spring等内容。😊 相信学完了Javase基础之后，对后面的学习会更加轻松。

本教程修订于2014年3月8日。

编者：Leo

Java基础教程

作者: JAVE_LOVER

<http://jave-lover.iteye.com>

本书由ITeye提供电子书DIY功能制作并发行。

更多精彩博客电子书，请访问：<http://www.iteye.com/blogs/pdf>