网络牛人的 JAVA 精华笔记

1 JAVA SE

1.1 深入 JAVA API

1.1.1 Lang 包

1.1.1.1 String类和StringBuffer类

位于 java.lang包中，这个包中的类使用时不用导入

String 类一旦初始化就不可以改变，而 stringbuffer 则可以。它用于封装内容可变的字符串。它可以使用tostring（）转换成string 字符串。

String x=”a”+4+”c”编译时等效于String x=new StringBuffer().append(“a”).append(4).append(“c”).toString();

字符串常量是一种特殊的匿名对象，String s1=”hello”;String s2=”hello”;则 s1==s2;因为他们指向同一个匿名对象。

如果 String s1=new String(“hello”);String s2=new String(“hello”);则 s1!=s2;

```
/*逐行读取键盘输入，直到输入为“bye”时，结束程序
注：对于回车换行，在 windows 下面，有'\r'和'\n'两个，而 unix 下面只有'\n'，但是写程序的时候都要把他区分开*/
public class readline
{
public static void main(String args[])
{
String strInfo=null;
int pos=0;
byte[] buf=new byte[1024];//定义一个数组，存放换行前的各个字符
int ch=0; //存放读入的字符
system.out.println(“Please input a string:”);
while(true)
{
```

```
try
{
ch=System.in.read(); //该方法每次读入一个字节的内容到ch 变量中。
}
catch(Exception e)
{
}
switch(ch)
{
case '\r': //回车时，不进行处理
break;
case '\n': //换行时，将数组总的内容放进字符串中
strInfo=new String(buf,0,pos); //该方法将数组中从第0 个开始，到第 pos 个结束存入字符串。
if(strInfo.equals("bye"))     //如果该字符串内容为 bye，则退出程序。
{
return;
}
else     //如果不为bye，则输出，并且竟 pos 置为 0，准备下次存入。
{
System.out.println(strInfo);
pos=0;
break;
}
default:
buf[pos++]=(byte)ch;   //如果不是回车，换行，则将读取的数据存入数组中。
}
}
}
}
```

String 类的常用成员方法

1、 构造方法：

String(byte[] byte,int offset,int length);这个在上面已经用到。

2、 equalsIgnoreCase:忽略大小写的比较，上例中如果您输入的是BYE,则不会退出，因为大小写不同，但是如果使用这个方法，则会退出。

3、 indexOf(int ch);返回字符ch 在字符串中首次出现的位置

4、 substring(int benginIndex);

5、 substring(int beginIndex,int endIndex);

返回字符串的子字符串，4 返回从benginindex位置开始到结束的子字符串，5 返回 beginindex 和 endindex-1 之间的子字符串。

基本数据类型包装类的作用是：将基本的数据类型包装成对象。因为有些方法不可以直接处理基本数据类型，只能处理对象，例如 vector 的 add 方法，参数就只能是对象。这时就需要使用他们的包装类将他们包装成对象。

例：在屏幕上打印出一个*组成的矩形，矩形的宽度和高度通过启动程序时传递给 main（）方法的参数指定。

```
public class testInteger

{

public static  void main(String[] args)

//main()的参数是 string 类型的数组，用来做为长，宽时，要转换成整型。

{

int w=new Integer(args[0]).intValue();

int h=Integer.parseInt(args[1]);

//int h=Integer.valueOf(args[1]).intValue();

//以上为三种将字符串转换成整形的方法。

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

{

StringBuffer sb=new StringBuffer(); //使用 stringbuffer，是因为它是可追加的。

for(int j=0;j<w;j++)

{

sb.append('*');

}

System.out.println(sb.toString()); //在打印之前，要将 stringbuffer 转化为 string 类型。
```

```
}

}

}
```

比较下面两段代码的执行效率：

```
（1）String sb=new String();

For(int j=0;j<w;j++)

{

Sb=sb+’*’;

}

(2) StringBuffer sb=new StringBuffer();

For(int j=0;j<w;j++)

{

Sb.append(‘*’);

}
```

(1) 和（2）在运行结果上相同，但效率相差很多。

（1） 在每一次循环中，都要先将string类型转换为stringbuffer类型，然后将‘*’追加进去，然后再调用tostring（）方法，转换为string类型，效率很低。

（2） 在没次循环中，都只是调用原来的那个stringbuffer对象，没有创建新的对象，所以效率比较高。

1.1.1.2 System类与Runtime类

由于java不支持全局函数和全局变量，所以java设计者将一些与系统相关的重要函数和变量放在system类中。

我们不能直接创建runtime的实例，只能通过runtime.getruntime（）静态方法来获得。

编程实例：在java程序中启动一个windows记事本程序的运行实例，并在该运行实例中打开该运行程序的源文件，启动的记事本程序5秒后关闭。

```
public class Property

{

public static  void main(String[] args)

{

Process p=null; //java虚拟机启动的进程。

try

{

p=Runtime.getRuntime().exec("notepad.exe Property.java"); //启动记事本并且打开源文件。
```

```
Thread.sleep(5000); //持续5秒

p.destroy(); //关闭该进程

}

catch(Exception ex)

{

ex.printStackTrace();

}

}

}
```

1.1.1.3 Java语言中两种异常的差别

Java 提供了两类主要的异常:runtime exception 和 checked exception。所有的checked exception 是从 java.lang.Exception 类衍生出来的，而 runtime exception 则是从java.lang.RuntimeException 或 java.lang.Error 类衍生出来的。

它们的不同之处表现在两方面:机制上和逻辑上。

一、机制上

它们在机制上的不同表现在两点:1.如何定义方法;2. 如何处理抛出的异常。请看下面 CheckedException 的定义:

```
public class CheckedException extends Exception
{

public CheckedException() {}
public CheckedException( String message )
{
super( message );
}
}
```

以及一个使用 exception 的例子:

```
public class ExceptionalClass
{

public void method1()
throws CheckedException
{
   // ... throw new CheckedException( "...出错了" );
}
public void method2( String arg )
```

```
{
  if( arg == null)
  {
    throw new NullPointerException( "method2 的参数 arg 是 null!" );
  }
}
public void method3() throws CheckedException
{
  method1();
}
}
```

你可能已经注意到了，两个方法method1()和 method2()都会抛出 exception，可是只有method1()做了声明。另外，method3()本身并不会抛出 exception，可是它却声明会抛出 CheckedException。在向你解释之前，让我们先来看看这个类的 main()方法:

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

ExceptionalClass example = new ExceptionalClass();
try
{
example.method1();
example.method3();
}
catch( CheckedException ex ) { } example.method2( null);
}
```

在 main()方法中，如果要调用 method1()，你必须把这个调用放在try/catch程序块当中，因为它会抛出Checked exception。

相比之下，当你调用 method2()时，则不需要把它放在try/catch程序块当中，因为它会抛出的exception 不是checked exception，而是runtime exception。会抛出 runtime exception 的方法在定义时不必声明它会抛出exception。

现在，让我们再来看看method3()。它调用了 method1()却没有把这个调用放在 try/catch程序块当中。它是通过声明它会抛出 method1()会抛出的 exception 来避免这样做的。它没有捕获这个exception，而是把它传递下去。实际上main()方法也可以这样做，通过声明它会抛出 Checked exception 来避免使用try/catch 程序块(当然我们反对这种做法)。

小结一下:

* Runtime exceptions:

在定义方法时不需要声明会抛出 runtime exception;

在调用这个方法时不需要捕获这个runtime exception;

runtime exception 是从java.lang.RuntimeException 或 java.lang.Error 类衍生出来的。

* Checked exceptions:

定义方法时必须声明所有可能会抛出的checked exception;

在调用这个方法时，必须捕获它的checked exception，不然就得把它的exception 传递下去;

checked exception 是从 java.lang.Exception 类衍生出来的。

二、逻辑上

从逻辑的角度来说，checked exceptions和 runtime exception 是有不同的使用目的的。checked exception 用来指示一种调用方能够直接处理的异常情况。而runtime exception 则用来指示一种调用方本身无法处理或恢复的程序错误。

checked exception 迫使你捕获它并处理这种异常情况。以java.net.URL 类的构建器(constructor)为例，它的每一个构建器都会抛出 MalformedURLException。MalformedURLException 就是一种 checked exception。设想一下，你有一个简单的程序，用来提示用户输入一个URL，然后通过这个URL 去下载一个网页。如果用户输入的URL有错误，构建器就会抛出一个exception。既然这个 exception 是 checked exception，你的程序就可以捕获它并正确处理:比如说提示用户重新输入。

再看下面这个例子:

```
public void method()
{

int [] numbers = { 1, 2, 3 };
int sum = numbers[0]  numbers[3];
}
```

在运行方法 method()时会遇到 ArrayIndexOutOfBoundsException(因为数组 numbers 的成员是从 0 到 2)。对于这个异常，调用方无法处理/纠正。这个方法method()和上面的 method2()一样，都是runtime exception 的情形。上面我已经提到，runtime exception 用来指示一种调用方本身无法处理/恢复的程序错误。而程序错误通常是无法在运行过程中处理的，必须改正程序代码。

总而言之，在程序的运行过程中一个 checked exception 被抛出的时候，只有能够适当处理这个异常的调用方才应该用try/catch 来捕获它。而对于runtime exception，则不应当在程序中捕获它。如果你要捕获它的话，你就会冒这样一个风险:程序代码的错误(bug)被掩盖在运行当中无法被察觉。因为在程序测试过程中，系统打印出来的调用堆栈路径(StackTrace)往往使你更快找到并修改代码中的错误。有些程序员建议捕获runtime exception 并纪录在 log 中，我反对这样做。这样做的坏处是你必须通过浏览 log 来找出问题，而用来测试程序的测试系统(比如 Unit Test)却无法直接捕获问题并报告出来。

在程序中捕获runtime exception还会带来更多的问题:要捕获哪些runtime exception?什么时候捕获?runtime exception 是不需要声明的，你怎样知道有没有runtime exception 要捕获?你想看到在程序中每一次调用方法时，都使用try/catch 程序块吗?

1.1.1.4 类加载器（ClassLoader）

1.1.1.4.1 基础知识

静态库、动态连接库

程序编制一般需经编辑、编译、连接、加载和运行几个步骤。在我们的应用中，有一些公共代码是需要反复使用，就把这些代码编译为"库"文件；在连接步骤中，连接器将从库文件取得所需的代码，复制到生成的可执行文件中。这种库称为静态库，其特点是可执行文件中包含了库代码的一份完整拷贝；缺点就是被多次使用就会有多份冗余拷贝。

为了克服这个缺点可以采用动态连接库。这个时候连接器仅仅是在可执行文件中打上标志，说明需要使用哪些动态连接库；当运行程序时，加载器根据这些标志把所需的动态连接库加载到内存。

另外在当前的编程环境中，一般都提供方法让程序在运行的时候把某个特定的动态连接库加载并运行，也可以将其卸载（例如Win32 的 LoadLibrary()&FreeLibrary()和 Posix 的 dlopen()&dlclose()）。这个功能被广泛地用于在程序运行时刻更新某些功能模块或者是程序外观。

What is ClassLoader?

与普通程序不同的是，Java 程序（class 文件）并不是本地的可执行程序。当运行Java 程序时，首先运行 JVM（Java 虚拟机），然后再把 Java class 加载到 JVM 里头运行，负责加载 Java class 的这部分就叫做 Class Loader。

JVM 本身包含了一个 ClassLoader 称为 Bootstrap ClassLoader，和 JVM 一样，Bootstrap ClassLoader 是用本地代码实现的，它负责加载核心 Java Class（即所有 java.*开头的类）。另外 JVM 还会提供两个 ClassLoader，它们都是用 Java 语言编写的，由 Bootstrap ClassLoader 加载；其中 Extension ClassLoader 负责加载扩展的 Java class（例如所有 javax.*开头的类和存放在 JRE 的 ext 目录下的类），Application ClassLoader 负责加载应用程序自身的类。

When to load the class?

什么时候 JVM 会使用 ClassLoader 加载一个类呢？当你使用 java 去执行一个类，JVM 使用 Application ClassLoader 加载这个类；然后如果类 A 引用了类 B，不管是直接引用还是用 Class.forName()引用，JVM 就会找到加载类 A 的 ClassLoader，并用这个 ClassLoader 来加载类 B。

Why use your own ClassLoader?

似乎 JVM 自身的 ClassLoader 已经足够了，为什么我们还需要创建自己的 ClassLoader 呢？

因为 JVM 自带的 ClassLoader 只是懂得从本地文件系统加载标准的 java class 文件，如果编写你自己的 ClassLoader，你可以做到：

1）在执行非置信代码之前，自动验证数字签名

2）动态地创建符合用户特定需要的定制化构建类

3）从特定的场所取得 java class，例如数据库中

4）等等

事实上当使用 Applet 的时候，就用到了特定的 ClassLoader，因为这时需要从网络上加载 java class，并且要检查相关的安全信息。

目前的应用服务器大都使用了 ClassLoader 技术，即使你不需要创建自己的 ClassLoader，了解其原理也有助于更好地部署自己的应用。

ClassLoader Tree & Delegation Model

当你决定创建你自己的 ClassLoader 时，需要继承 java.lang.ClassLoader 或者它的子类。在实例化每个 ClassLoader 对象时，需要指定一个父对象；如果没有指定的话，系统自动指定 ClassLoader.getSystemClassLoader()为父对象。如下图：

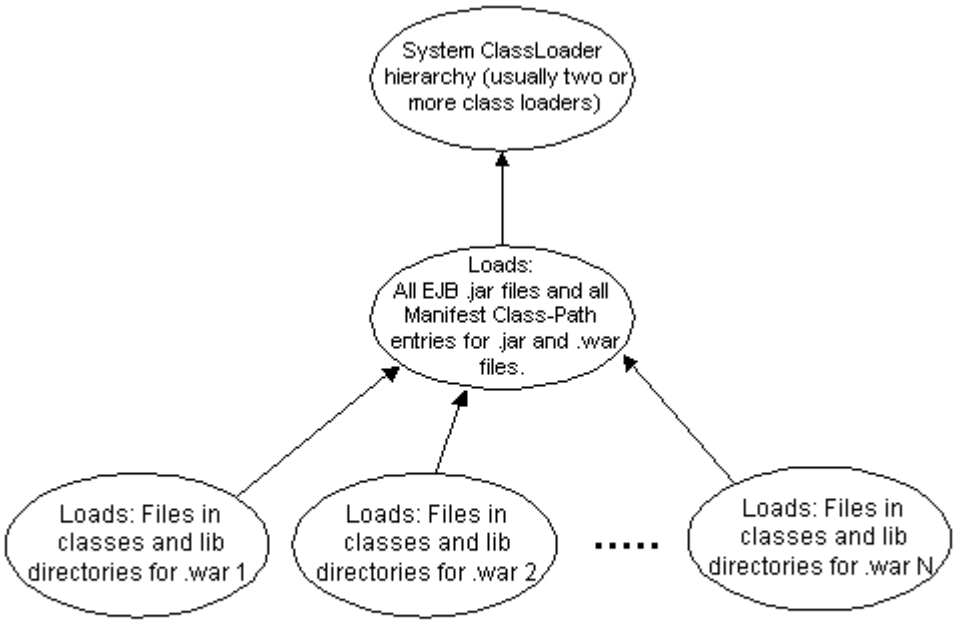


Figure 2. WebLogic 6.1 Service Pack 2 class loading architecture

在 Java 1.2 后，java class 的加载采用所谓的委托模式（Delegation Modle），当调用一个 ClassLoader.loadClass()加载一个类的时候，将遵循以下的步骤：

1）检查这个类是否已经被加载进来了？

2）如果还没有加载，调用父对象加载该类

3）如果父对象无法加载，调用本对象的 findClass()取得这个类。

所以当创建自己的 Class Loader 时，只需要重载 findClass()这个方法。

Unloading? Reloading?

当一个 java class 被加载到 JVM 之后，它有没有可能被卸载呢？我们知道 Win32 有 FreeLibrary()函数，Posix 有 dlclose()函数可以被调用来卸载指定的动态连接库，但是 Java 并没有提供一个 UnloadClass()的方法来卸载指定的类。

在 Java 中，java class 的卸载仅仅是一种对系统的优化，有助于减少应用对内存的占用。既然是一种优化方法，那么就完全是 JVM 自行决定如何实现，对 Java 开发人员来说是完全透明的。

在什么时候一个 java class/interface 会被卸载呢？Sun 公司的原话是这么说的："class or interface may be unloaded if and only if its class loader is unreachable. Classes loaded by the bootstrap loader may not be unloaded."

事实上我们关心的不是如何卸载类的，我们关心的是如何更新已经被加载了的类从而更新应用的功能。JSP 则是一个非常典型的例子，如果一个 JSP 文件被更改了，应用服务器则需要把更改后的 JSP 重新编译，然后加载新生成的类来响应后继的请求。

其实一个已经加载的类是无法被更新的，如果你试图用同一个 ClassLoader 再次加载同一个类，就会得到异常（java.lang.LinkageError: duplicate class definition），我们只能够重新创建一个新的 ClassLoader 实例来再次

加载新类。至于原来已经加载的类，开发人员不必去管它，因为它可能还有实例正在被使用，只要相关的实例都被内存回收了，那么 JVM 就会在适当的时候把不会再使用的类卸载。

1.1.1.4.2 类加载的表现形式

java 中的类是动态加载的，我们先看一下我们常用的类加载方式，先有一个感性的认识，才能进一步深入讨论,类加载无非就是下面三种方式。

```
class A{}
class B{}
class C{}
public class Loader{
   public static void main(String[] args) throws Exception{
      Class aa=A.class;
      Class bb=Class.forName("B");
      Class cc=ClassLoader.getSystemClassLoader().loadClass("C");
   }
}
```

我们先看.class 字面量方式，很多人可能不知道这种方式，因为这种用法不是一般 java 语法。

通过 javap 我们可以发现，这种方式的大致等价于定义了一个静态成员变量

```
   static Class class$0;(后面的编号是增长的)
```

你可以试图再定义一个 static Class class$0,应该会收到一个编译错误(重复定义)。

```
Class aa=A.class;
```

就相当于

```
   if(class$0==null){
 try{
        Class.forName("A");
 }
 cacth(ClassNotFoundException e){
   throw new NoClassDefFoundError(e);
 }
   }
   Class aa=class$0;
```

可以很清楚的看到，这种类的字面量定义其实不是加载类的方式，而是被编译器处理了，实质上是使用了 Class.forName 方法，但是使用这种方式有一个很大的好处就是不用处理异常，因为编译器处理的时候如果找不到类会抛出一个 NoClassDefFoundError。也许你觉得需要处理 ClassNotFoundException 这种异常，事实上 99%的情况下我们可以把这种异常认为是一个错误。所以大部分情况我们使用这种方式会更简洁。

最常用的方式就是 Class.forName 方式了，这也是一个通用的上层调用。这个方法有两个重载，可能很多人都忽略了第二个方法。

```
public static Class forName(String name) throws ClassNotFoundException
public static Class forName(String name, boolean initialize,ClassLoader loader) throws ClassNotFoundException
```

第二个方法后面多了两个参数，第二个参数表示是否初始化，第三个参数为指定的类加载器。

在上面的例子中：

```
Class bb=Class.forName("B");等价于
Class bb=Class.forName("B",true,Loader.class.getClassLoader());
```

这里要详细说一下这个类的初始化这个参数,如果这个参数为 false 的话，
类中的 static 成员不会被初始化，static 语句块也不会被执行。
也就是类虽然被加载了，但是没有被初始化，不过在第一次使用时仍然会初始化。
所以我们有时候会看到 Class.forName("XXX").newInstance()这样的语句，为什么这里要创建一个
不用的实例呢?不过是为了保证类被初始化(兼容以前的系统)。
其实第二个方法是比较难用的，需要指定类加载器，如果不指定而且又没有安装安全管理器的化，
是无法加载类的，只要看一下具体的实现就明白了。
最本质的方式当然是直接使用 ClassLoader 加载了，所有的类最终都是通过 ClassLoader 加载的，
Class cc=ClassLoader.getSystemClassLoader().loadClass("C");
这里通过使用系统类加载器来加载某个类，很直接的方式，但是很遗憾的是通过这种方式加载类，
类是没有被初始化的(也就是初始化被延迟到真正使用的时候).不过我们也可以借鉴上面的经验，加载
后实例化一个对象 Class cc=ClassLoader.getSystemClassLoader().loadClass("C").newInstance()。
这里使用了系统类加载器，也是最常用的类加载器，从 classpath 中寻找要加载的类。
java 中默认有三种类加载器：引导类加载器，扩展类加载器，系统类加载器。
java 中的类加载有着规范的层次结构，如果我们要了解类加载的过程，需要明确知道哪个类被谁
加载，某个类加载器加载了哪些类等等，就需要深入理解 ClassLoader 的本质。
以上只是类加载的表面的东西，我们还将讨论深层次的东西。

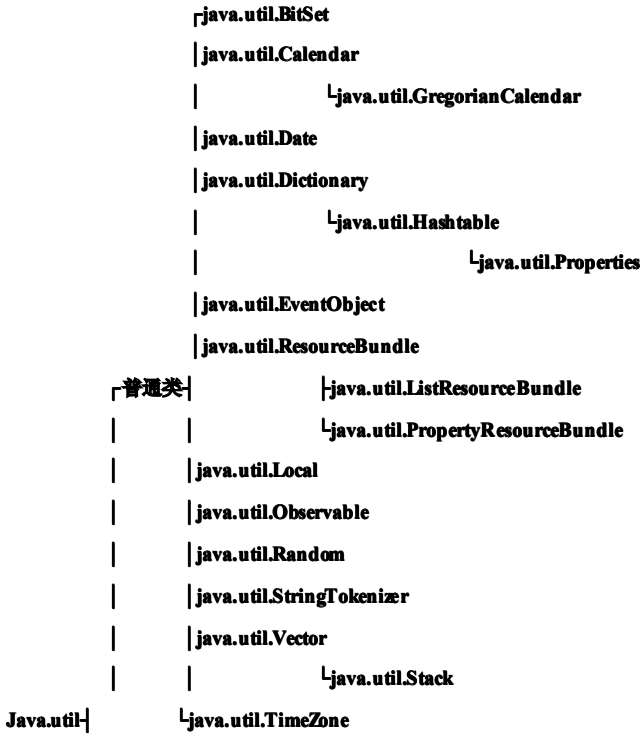
1.1.2 集合类

集合类用于存储一组对象。

1.1.2.1 Java Util 包使用详解

本章介绍 Java 的实用工具类库 java.util 包。在这个包中，Java 提供了一些实用的方法和数据结构。例如，Java 提供日期(Data)类、日历(Calendar)类来产生和获取日期及时间，提供随机数(Random)类产生各种类型的随机数，还提供了堆栈(Stack)、向量(Vector)、位集合(Bitset)以及哈希表(Hashtable)等类来表示相应的数据结构。

图 1.1 给出了 java.util 包的基本层次结构图。下面我们将具体介绍其中几个重要的类。



```
|                    └java.util.SimpleTimeZone
|          ┌java.util.Enumeration
├接  口┤java.util.EventListener
|          └java.util.Observer
|          ┌java.util.EmptyStackException
└异常类┤java.util.MissingResourceException
          |java.util.NoSuchElementException
          └java.util.TooManyListenersException
```

图 1.1 java.util 包的基本层次结构

1.1.2.1.1 日期类 Date

Java 在日期类中封装了有关日期和时间的信息，用户可以通过调用相应的方法来获取系统时间或设置日期和时间。Date 类中有很多方法在 JDK1.0 公布后已经过时了，在 8.3 中我们将介绍 JDK1.0 中新加的用于替代 Date 的功能的其它类。

在日期类中共定义了六种构造函数。

(1)public Date()

创建的日期类对象的日期时间被设置成创建时刻相对应的日期时间。

例 Date today=new Date()；//today 被设置成创建时刻相对应的日期时间。

(2)public Date (long date)

long 型的参数 date 可以通过调用 Date 类中的 static 方法 parse(String s)来获得。

例 long l=Date.parse("Mon 6 Jan 1997 13:3:00");

Date day=new Date(l);

//day 中时间为 1997 年 1 月 6 号星期一，13:3:00。

(3)public Date(String s)

按字符串 s 产生一日期对象。s 的格式与方法 parse 中字符串参数的模式相同。

例 Date day=new Date("Mon 6 Jan 1997 13:3:00");

//day 中时间为 1997 年 1 月 6 号星期一，13:3:00.

(4)public Date(int year,int month,int date)

(5)public Date(int year,int month,int date,int hrs,int min)

(6)public Date(int year,int month,int date,int hrs,int min,int sec)

按给定的参数创建一日期对象。

参数说明：

year 的值为：需设定的年份-1900。例如需设定的年份是 1997 则 year 的值应为 97，即 1997-1900 的结果。所以 Date 中可设定的年份最小为 1900；

month 的值域为 0～11，0 代表 1 月，11 表代表 12 月；

date 的值域在 1～31 之间；

hrs 的值域在 0～23 之间。从午夜到次日凌晨 1 点间 hrs=0，从中午到下午 1 点间 hrs=12；

min 和 sec 的值域在 0～59 之间。

例 Date day=new Date(11,3,4);

//day 中的时间为：04-Apr-11 12:00:00 AM

另外，还可以给出不正确的参数。

例 设定时间为 1910 年 2 月 30 日，它将被解释成 3 月 2 日。

Date day=new Date(10,1,30,10,12,34);

System.out.println("Day's date is:"+day);

//打印结果为：Day's date is:Web Mar 02 10:13:34 GMT+08:00 1910

下面我们给出一些 Date 类中常用方法。

(1)public static long UTC(int year,int month,int date,int hrs. int min,int sec)

该方法将利用给定参数计算 UTC 值。UTC 是一种计时体制，与 GMT(格林威治时间)的计时体系略有差别。UTC 计时体系是基于原子时钟的，而 GTMT 计时体系是基于天文学观测的。计算中使用的一般为 GMT 计时体系。

(2)public static long parse(String s)

该方法将字符串 s 转换成一个 long 型的日期。在介绍构造方法 Date(long date)时曾使用过这个方法。

字符串 s 有一定的格式，一般为：

(星期 日 年 时间 GMT+时区)

若不注明时区，则为本地时区。

(3)public void setMonth(int month)

(4)public int getMonth()

这两个方法分别为设定和获取月份值。

获取的月份的值域为 0～11，0 代表 1 月，11 代表 12 月。

(5)public String toString()

(6)public String toLocalString()

(7)public String toGMTString()

将给定日期对象转换成不同格式的字符串。它们对应的具体的格式可参看例子 8.1。

(8)public int getTimezoneOffset()

该方法用于获取日期对象的时区偏移量。

例 8.1 中对上面介绍的 Date 类中的基本方法进行了具体的应用，并打印了相应的结果。由于使用了一些过时的方法，所以编译时会有警告信息。另外，由于本例中的时间表示与平台有关，不同的 JDK 版本对此处理不完全相同，因此不同版本的 JDK 执行本例的结果可能有细微差异。

例 1.1 DateApp.java

```
import java.lang.System;
import java.util.Date;
public class DateApp{
  public static void main(String args[]){
    Date today=new Date();
    //today 中的日期被设成创建时刻的日期和时间，假设创建时刻为 1997 年 3 月
    //23 日 17 时 51 分 54 秒。
    System.out.println("Today's date is "+today);
    //返回一般的时间表示法，本例中结果为
    //Today's date is Fri May 23 17:51:54 1997
    System.out.println("Today's date(Internet GMT)is:"
      +today.toGMTString());
    //返回结果为 GMT 时间表示法，本例中结果为
    //Today's date(Internet GMT)is: 23 May 1997 09:51:54:GMT
    System.out.println("Today's date(Locale) is:"
      +today.toLocaleString());
    //返回结果为本地习惯的时间表示法，结果为
    //Today's date(Locale)is:05/23/97 17:51:54
    System.out.println("Today's year is: "+today.getYear());
    System.out.println("Today's month is: "+(today.getMonth()+1));
```

```
        System.out.println("Today's date is: "+today.getDate());
        //调用Date类中方法，获取年月日的值。
        //下面调用了不同的构造方法来创建Date类的对象。
        Date day1=new Date(100,1,23,10,12,34);
        System.out.println("Day1's date is: "+day1);
        Date day2=new Date("Sat 12 Aug 1996 13:3:00");
        System.out.println("Day2's date is: "+day2);
        long l= Date.parse("Sat 5 Aug 1996 13:3:00 GMT+0800");
        Date day3= new Date(l);
        System.out.println("Day3's date(GMT)is: "+day3.toGMTString());
        System.out.println("Day3's date(Locale)is: "
          +day3.toLocaleString());
        System.out.println("Day3's time zone offset is:"
          +day3.getTimezoneOffset());
    }
}
```

运行结果(JDK1.3 版，与原文不同，原文是 JDK1.0 版)：

```
E:\java\tutorial\java01>java DateApp
Today's date is Thu Dec 27 17:58:16 CST 2001
Today's date(Internet GMT)is:27 Dec 2001 09:58:16 GMT
Today's date(Locale) is:2001-12-27 17:58:16
Today's year is: 101
Today's month is: 12
Today's date is: 27
Day1's date is: Wed Feb 23 10:12:34 CST 2000
Day2's date is: Fri Aug 12 13:03:00 CST 1996
Day3's date(GMT)is: 5 Aug 1996 05:03:00 GMT
Day3's date(Locale)is: 1996-8-5 13:03:00
Day3's time zone offset is:-480

E:\java\tutorial\java01>
```

1.1.2.1.2 日历类 Calendar

在早期的 JDK 版本中，日期(Date)类附有两大功能：(1)允许用年、月、日、时、分、秒来解释日期；(2)允许对表示日期的字符串进行格式化和句法分析。在 JDK1.1 中提供了类 Calendar 来完成第一种功能，类 DateFormat 来完成第二项功能。dateFormat 是 java.text 包中的一个类。与 Date 类有所不同的是，DateFormat 类接受用各种语言和不同习惯表示的日期字符串。本节将介绍 java.util 包中的类 Calendar 及其它新增加的相关的类。

类 Calendar 是一个抽象类，它完成日期(Date)类和普通日期表示法(即用一组整型域如 YEAR，MONTH，DAY，HOUR 表示日期)之间的转换。

由于所使用的规则不同，不同的日历系统对同一个日期的解释有所不同。在 JDK1.1 中提供了 Calendar 类一个子类 GregorianCalendar??它实现了世界上普遍使用的公历系统。当然用户也可以通过继承 Calendar 类，并增加所需规则，以实现不同的日历系统。

第 GregorianCalendar 继承了 Calendar类。本节将在介绍类 GregorianCalendar 的同时顺带介绍 Calendar 类中的相关方法。

类 GregorianCalendar 提供了七种构造函数：

(1)public GregorianCalendar()

创建的对象中的相关值被设置成指定时区，缺省地点的当前时间，即程序运行时所处的时区、地点的当前时间。

(2)public GregorianCalendar(TimeZone zone)

创建的对象中的相关值被设置成指定时区zone，缺省地点的当前时间。

(3)public GregorianCalendar(Locale aLocale)

创建的对象中的相关值被设置成缺省时区，指定地点 aLocale 的当前时间。

(4)public GregorianCalendar(TimeZone zone,Local aLocale)

创建的对象中的相关值被设置成指定时区，指定地点的当前时间。

上面使用到的类 TimeZone 的性质如下：

TimeZone 是 java.util 包中的一个类，其中封装了有关时区的信息。每一个时区对应一组ID。类TimeZone 提供了一些方法完成时区与对应 ID 两者之间的转换。

(Ⅰ)已知某个特定的 ID，可以调用方法

public static synchronized TimeZone getTimeZone(String ID)

来获取对应的时区对象。

例 太平洋时区的ID 为 PST，用下面的方法可获取对应于太平洋时区的时区对象：

TimeZone tz=TimeZone.getTimeZone("PST");

调用方法 getDefault()可以获取主机所处时区的对象。

TimeZone tz=TimeZone.getDefault();

(Ⅱ)调用以下方法可以获取时区的 ID

■public static synchronized String[] getavailableIDs(int rawOffset)

根据给定时区偏移值获取 ID 数组。同一时区的不同地区的ID 可能不同，这是由于不同地区对是否实施夏时制意见不统一而造成的。

例 String s[]=TimeZone.getAvailableIDs(-7*60*60*1000);

打印 s，结果为s[0]=PNT，s[1]=MST

■public static synchronized String[] getAvailableIDs()

获取提供的所有支持的 ID。

■public String getID()

获取特定时区对象的 ID。

例 TimeZone tz=TimeZone.getDefault();

String s=tzgetID();

打印 s，结果为s=CTT。

上面使用类的对象代表了一个特定的地理、政治或文化区域。Locale 只是一种机制，它用来标识一类对象，Local 本身并不包含此类对象。

要获取一个 Locale 的对象有两种方法：

(Ⅰ)调用 Locale 类的构造方法

Locale(String language,String country)

Locale(String language,String country,String variant)

参数说明：language??在 ISO-639 中定义的代码，由两个小写字母组成。

country??在 ISO-3166 中定义的代码，由两个大写字母组成。

variant??售货商以及特定浏览器的代码，例如使用 WIN 代表 Windows。

(Ⅱ)调用 Locale 类中定义的常量

Local 类提供了大量的常量供用户创建 Locale 对象。

例 Locale.CHINA

为中国创建一个 Locale 的对象。

类 TimeZone 和类 Locale 中的其它方法，读者可查阅 API。

(5)public GregorianCalendar(int year,int month,int date)

(6)public GregorianCalendar(int year,int month,int date,int hour,int minute)

(7)public GregorianCalendar(int year,int month,int date,int hour,int minute,int second)

用给定的日期和时间创建一个 GregorianCalendar 的对象。

参数说明：

year-设定日历对象的变量 YEAR；month-设定日历对象的变量 MONTH；

date-设定日历对象的变量 DATE；hour-设定日历对象的变量 HOUR_OF_DAY；

minute-设定日历对象的变量 MINUTE；second-设定日历对象的变量 SECOND。

与 Date 类中不同的是 year 的值没有 1900 这个下限，而且 year 的值代表实际的年份。month 的含义与 Date 类相同，0 代表 1 月，11 代表 12 月。

例 GregorianCalendar cal=new GregorianCalendar(1991,2,4)

cal 的日期为 1991 年 3 月 4 号。

除了与 Date 中类似的方法外，Calendar 类还提供了有关方法对日历进行滚动计算和数学计算。计算规则由给定的日历系统决定。进行日期计算时，有时会遇到信息不足或信息不实等特殊情况。Calendar 采取了相应的方法解决这些问题。当信息不足时将采用缺省设置，在 GregorianCalendar 类中缺省设置一般为 YEAR=1970,MONTH=JANUARY,DATE=1。

当信息不实时，Calendar 将按下面的次序优先选择相应的 Calendar 的变量组合，并将其它有冲突的信息丢弃。

MONTH+DAY_OF_MONTH

MONTH+WEEK_OF_MONTH+DAY_OF_WEEK

MONTH+DAY_OF_WEEK_OF_MONTH+DAY_OF_WEEK

DAY_OF+YEAR

DAY_OF_WEEK_WEEK_OF_YEAR

HOUR_OF_DAY

1.1.2.1.3 随机数类 Random

Java 实用工具类库中的类 java.util.Random 提供了产生各种类型随机数的方法。它可以产生 int、long、float、double 以及 Goussian 等类型的随机数。这也是它与 java.lang.Math 中的方法 Random()最大的不同之处，后者只产生 double 型的随机数。

类 Random 中的方法十分简单，它只有两个构造方法和六个普通方法。

构造方法：

(1)public Random()

(2)public Random(long seed)

Java 产生随机数需要有一个基值 seed，在第一种方法中基值缺省，则将系统时间作为 seed。

普通方法：

(1)public synonronized void setSeed(long seed)

该方法是设定基值 seed。

(2)public int nextInt()

该方法是产生一个整型随机数。

(3)public long nextLong()

该方法是产生一个 long 型随机数。

(4)public float nextFloat()

该方法是产生一个 Float 型随机数。

(5)public double nextDouble()

该方法是产生一个 Double 型随机数。

(6)public synchronized double nextGoussian()

该方法是产生一个 double 型的 Goussian 随机数。

例 1.2 RandomApp.java。

```
//import java.lang.*;
import java.util.Random;

public class RandomApp{
   public static void main(String args[]){
      Random ran1=new Random();
      Random ran2=new Random(12345);
      //创建了两个类 Random 的对象。
      System.out.println("The 1st set of random numbers:");
      System.out.println("\t Integer:"+ran1.nextInt());
      System.out.println("\t Long:"+ran1.nextLong());
      System.out.println("\t Float:"+ran1.nextFloat());
      System.out.println("\t Double:"+ran1.nextDouble());
      System.out.println("\t Gaussian:"+ran1.nextGaussian());
      //产生各种类型的随机数
      System.out.print("The 2nd set of random numbers:");
      for(int i=0;i<5;i++){
         System.out.println(ran2.nextInt()+" ");
         if(i==2) System.out.println();
         //产生同种类型的不同的随机数。
         System.out.println();//原文如此
      }
   }
}
```

运行结果：

```
E:\java01>java RandomApp
The 1st set of random numbers:
      Integer:-173899656
      Long:8056223819738127077
      Float:0.6293638
      Double:0.7888394520265607
      Gaussian:0.5015701094568733
The 2nd set of random numbers:1553932502
-2090749135
-287790814
-355989640
-716867186
E:\java01>
```

1.1.2.1.4 向量类 Vector

Java.util.Vector 提供了向量(Vector)类以实现类似动态数组的功能。在Java 语言中。正如在一开始就提到过，是没有指针概念的，但如果能正确灵活地使用指针又确实可以大大提高程序的质量，比如在C、C++中所谓“动态数组”一般都由指针来实现。为了弥补这点缺陷，Java 提供了丰富的类库来方便编程者使用，Vector 类便是其中之一。事实上，灵活使用数组也可完成向量类的功能，但向量类中提供的大量方法大大方便了用户的使用。

创建了一个向量类的对象后，可以往其中随意地插入不同的类的对象，既不需顾及类型也不需预先选定向量的容量，并可方便地进行查找。对于预先不知或不愿预先定义数组大小，并需频繁进行查找、插入和删除工作的情况，可以考虑使用向量类.

向量类提供了三种构造方法：

public vector()

public vector(int initialcapacity,int capacityIncrement)

public vector(int initialcapacity)

使用第一种方法，系统会自动对向量对象进行管理。若使用后两种方法，则系统将根据参数initialcapacity 设定向量对象的容量(即向量对象可存储数据的大小)，当真正存放的数据个数超过容量时，系统会扩充向量对象的存储容量。参数capacityIncrement 给定了每次扩充的扩充值。当capacityIncrement 为 0 时，则每次扩充一倍。利用这个功能可以优化存储。

在 Vector 类中提供了各种方法方便用户使用：

■插入功能

(1)public final synchronized void addElement(Object obj)

将 obj 插入向量的尾部。obj 可以是任何类的对象。对同一个向量对象，可在其中插入不同类的对象。但插入的应是对象而不是数值，所以插入数值时要注意将数值转换成相应的对象。

例 要插入一个整数1 时，不要直接调用 v1.addElement(1)，正确的方法为：

Vector v1=new Vector();

Integer integer1=new Integer(1);

v1.addElement(integer1);

(2)public final synchronized void setElementAt(object obj,int index)

将 index 处的对象设成 obj，原来的对象将被覆盖。

(3)public final synchronized void insertElementAt(Object obj,int index)

在 index 指定的位置插入 obj，原来对象以及此后的对象依次往后顺延。

■删除功能

(1)public final synchronized void removeElement(Object obj)

从向量中删除 obj。若有多个存在，则从向量头开始试，删除找到的第一个与obj 相同的向量成员。

(2)public final synchronized void removeAllElement()

删除向量中所有的对象。

(3)public final synchronized void removeElementlAt(int index)

删除 index 所指的地方的对象。

■查询搜索功能

(1)public final int indexOf(Object obj)

从向量头开始搜索 obj ,返回所遇到的第一个 obj 对应的下标，若不存在此 obj，返回-1。

(2)public final synchronized int indexOf(Object obj,int index)

从 index 所表示的下标处开始搜索 obj。

(3)public final int lastIndexOf(Object obj)

从向量尾部开始逆向搜索 obj。

(4)public final synchronized int lastIndexOf(Object obj,int index)

从 index 所表示的下标处由尾至头逆向搜索 obj。

(5)public final synchronized Object firstElement()

获取向量对象中的首个 obj。

(6)public final synchronized Object lastelement()

获取向量对象中的最后一个 obj。

了解了向量的最基本的方法后，我们来看一下例 8.3VectorApp.java。

例 1.3 VectorApp.java。

```
import java.util.Vector;
import java.lang.*;//这一句不应该要，但原文如此
import java.util.Enumeration;
public class VectorApp{
   public static void main(String[] args){
      Vector v1=new Vector();
      Integer integer1=new Integer(1);
      v1.addElement("one");
      //加入的为字符串对象
      v1.addElement(integer1);
      v1.addElement(integer1);
      //加入的为 Integer 的对象
      v1.addElement("two");
      v1.addElement(new Integer(2));
      v1.addElement(integer1);
      v1.addElement(integer1);
      System.out.println("The vector v1 is:\n\t"+v1);
      //将 v1 转换成字符串并打印
      v1.insertElementAt("three",2);
      v1.insertElementAt(new Float(3.9),3);
      System.out.println("The vector v1(used method insertElementAt()) is:\n\t "+v1);
      //往指定位置插入新的对象，指定位置后的对象依次往后顺延
      v1.setElementAt("four",2);
      System.out.println("The vector v1(used method setElementAt()) is:\n\t "+v1);
      //将指定位置的对象设置为新的对象
      v1.removeElement(integer1);
      //从向量对象 v1 中删除对象 integer1 由于存在多个 integer1 所以从头开始
      //找，删除找到的第一个 integer1
      Enumeration enum=v1.elements();
      System.out.print("The vector v1(used method removeElement())is:");
      while(enum.hasMoreElements())
      System.out.print(enum.nextElement()+" ");
      System.out.println();
      //使用枚举类(Enumeration)的方法来获取向量对象的每个元素
      System.out.println("The position of object 1(top-to-bottom):"
        + v1.indexOf(integer1));
      System.out.println("The position of object 1(tottom-to-top):"
        +v1.lastIndexOf(integer1));
      //按不同的方向查找对象 integer1 所处的位置
```

```
        v1.setSize(4);
        System.out.println("The new vector(resized the vector)is:"+v1);
        //重新设置v1 的大小，多余的元素被行弃
      }
    }
```

运行结果：

```
E:\java01>java VectorApp
The vector v1 is:
        [one, 1, 1, two, 2, 1, 1]
The vector v1(used method insertElementAt()) is:
        [one, 1, three, 3.9, 1, two, 2, 1, 1]
The vector v1(used method setElementAt()) is:
        [one, 1, four, 3.9, 1, two, 2, 1, 1]
The vector v1(used method removeElement())is:one four 3.9 1 two 2 1 1
The position of object 1(top-to-bottom):3
The position of object 1(tottom-to-top):7
The new vector(resized the vector)is:[one, four, 3.9, 1]
E:\java01>
```

从例 1.3 运行的结果中可以清楚地了解上面各种方法的作用，另外还有几点需解释。

(1)类 Vector 定义了方法

public final int size()

此方法用于获取向量元素的个数。它的返回值是向是中实际存在的元素个数，而非向量容量。可以调用方法capactly()来获取容量值。

方法：

public final synchronized void setsize(int newsize)

此方法用来定义向量大小。若向量对象现有成员个数已超过了 newsize 的值，则超过部分的多余元素会丢失。

(2)程序中定义了 Enumeration 类的一个对象

Enumeration 是 java.util 中的一个接口类，在Enumeration 中封装了有关枚举数据集合的方法。

在 Enumeration 中提供了方法 hawMoreElement()来判断集合中是束还有其它元素和方法 nextElement()来获取下一个元素。利用这两个方法可以依次获得集合中元素。

Vector 中提供方法：

public final synchronized Enumeration elements()

此方法将向量对象对应到一个枚举类型。java.util 包中的其它类中也大都有这类方法，以便于用户获取对应的枚举类型。

1.1.2.1.5 栈类 Stack

Stack 类是 Vector 类的子类。它向用户提供了堆栈这种高级的数据结构。栈的基本特性就是先进后出。即先放入栈中的元素将后被推出。Stack 类中提供了相应方法完成栈的有关操作。

基本方法：

public Object push(Object Hem)

将 Hem 压入栈中，Hem 可以是任何类的对象。

public Object pop()

弹出一个对象。

public Object peek()

返回栈顶元素，但不弹出此元素。

public int search(Object obj)

搜索对象 obj,返回它所处的位置。

public boolean empty()

判别栈是否为空。

例 1.4 StackApp.java 使用了上面的各种方法。

例 1.4 StackApp.java。

```
import java.lang.*;
import java.util.*;
public class StackApp{
   public static void main(String args[]){
      Stack sta=new Stack();
      sta.push("Apple");
      sta.push("banana");
      sta.push("Cherry");
      //压入的为字符串对象
      sta.push(new Integer(2));
      //压入的为 Integer 的对象，值为 2
      sta.push(new Float(3.5));
      //压入的为 Float 的对象，值为 3.5
      System.out.println("The stack is,"+sta);
      //对应栈 sta
      System.out.println("The top of stack is:"+sta.peek());
      //对应栈顶元素，但不将此元素弹出
      System.out.println("The position of object Cherry is:"
      +sta.search("cherry"));
      //打印对象 Cherry 所处的位置
      System.out.print("Pop the element of the stack:");
      while(!sta.empty())
      System.out.print(sta.pop()+" ");
      System.out.println();
      //将栈中的元素依次弹出并打印。与第一次打印的 sta 的结果比较，可看出栈
      //先进后出的特点
   }
}
```

运行结果(略)

1.1.2.1.6 哈希表类 Hashtable

哈希表是一种重要的存储方式，也是一种常见的检索方法。其基本思想是将关系码的值作为自变量，通过一定的函数关系计算出对应的函数值，把这个数值解释为结点的存储地址，将结点存入计算得到存储地址所对应的存储单元。检索时采用检索关键码的方法。现在哈希表有一套完整的算法来进行插入、删除和解决冲突。在 Java 中哈希表用于存储对象，实现快速检索。

Java.util.Hashtable 提供了种方法让用户使用哈希表，而不需要考虑其哈希表真正如何工作。

哈希表类中提供了三种构造方法，分别是：

public Hashtable()

public Hashtable(int initialcapacity)

public Hashtable(int initialCapacity,float loadFactor)

参数 initialCapacity 是 Hashtable 的初始容量，它的值应大于 0。loadFactor 又称装载因子，是一个 0.0 到 0.1 之间的 float 型的浮点数。它是一个百分比，表明了哈希表何时需要扩充，例如，有一哈希表，容量为 100，而装载因子为 0.9，那么当哈希表 90%的容量已被使用时，此哈希表会自动扩充成一个更大的哈希表。如果用户不赋这些参数，系统会自动进行处理，而不需要用户操心。

Hashtable 提供了基本的插入、检索等方法。

■插入

public synchronized void put(Object key,Object value)

给对象 value 设定一关键字 key,并将其加到 Hashtable 中。若此关键字已经存在，则将此关键字对应的旧对象更新为新的对象 Value。这表明在哈希表中相同的关键字不可能对应不同的对象(从哈希表的基本思想来看，这也是显而易见的)。

■检索

public synchronized Object get(Object key)

根据给定关键字 key 获取相对应的对象。

public synchronized boolean containsKey(Object key)

判断哈希表中是否包含关键字 key。

public synchronized boolean contains(Object value)

判断 value 是否是哈希表中的一个元素。

■删除

public synchronized object remove(object key)

从哈希表中删除关键字 key 所对应的对象。

public synchronized void clear()

清除哈希表

另外，Hashtalbe 还提供方法获取相对应的枚举集合：

public synchronized Enumeration keys()

返回关键字对应的枚举对象。

public synchronized Enumeration elements()

返回元素对应的枚举对象。

例 1.5 Hashtable.java 给出了使用 Hashtable 的例子。

例 1.5 Hashtalbe.java。

```
//import java.lang.*;
import java.util.Hashtable;
import java.util.Enumeration;
public class HashApp{
   public static void main(String args[]){
      Hashtable hash=new Hashtable(2,(float)0.8);
      //创建了一个哈希表的对象 hash，初始容量为 2，装载因子为 0.8

      hash.put("Jiangsu","Nanjing");
      //将字符串对象“Jiangsu”给定一关键字“Nanjing”,并将它加入 hash
      hash.put("Beijing","Beijing");
      hash.put("Zhejiang","Hangzhou");

      System.out.println("The hashtable hash1 is: "+hash);
      System.out.println("The size of this hash table is "+hash.size());
```

```
        //打印hash的内容和大小

        Enumeration enum1=hash.elements();
        System.out.print("The element of hash is: ");
        while(enum1.hasMoreElements())
          System.out.print(enum1.nextElement()+" ");
        System.out.println();
        //依次打印hash中的内容
        if(hash.containsKey("Jiangsu"))
          System.out.println("The capatial of Jiangsu is "+hash.get("Jiangsu"));
        hash.remove("Beijing");
        //删除关键字Beijing对应对象
        System.out.println("The hashtable hash2 is: "+hash);
        System.out.println("The size of this hash table is "+hash.size());
      }
    }
```

运行结果：

```
The hashtable hash1 is: {Beijing=Beijing, Zhejiang=Hangzhou, Jiangsu=Nanjing}
The size of this hash table is 3
The element of hash is: Beijing Hangzhou Nanjing
The capatial of Jiangsu is Nanjing
The hashtable hash2 is: {Zhejiang=Hangzhou, Jiangsu=Nanjing}
The size of this hash table is 2
```

Hashtable是Dictionary(字典)类的子类。在字典类中就把关键字对应到数据值。字典类是一个抽象类。在java.util中还有一个类Properties，它是Hashtable的子类。用它可以进行与对象属性相关的操作。

1.1.2.1.7 位集合类BitSet

位集合类中封装了有关一组二进制数据的操作。

我们先来看一下例8.6 BitSetApp.java。

例8.6 BitSetApp.java

```
//import java.lang.*;
import java.util.BitSet;
public class BitSetApp{
  private static int n=5;
  public static void main(String[] args){
    BitSet set1=new BitSet(n);
    for(int i=0;i<N;I++) SET1.SET(I);
    //将set1的各位赋1，即各位均为true
    BitSet set2= new BitSet();
    set2=(BitSet)set1.clone();
    //set2为set1的拷贝
    set1.clear(0);
```

```
        set2.clear(2);
        //将 set1 的第 0 位 set2 的第 2 位清零
        System.out.println("The set1 is: "+set1);
        //直接将 set1 转换成字符串输出，输出的内容是 set1 中值 true 所处的位置
        //打印结果为 The set1 is:{1,2,3,4}
        System.out.println("The hash code of set2 is: "+set2.hashCode());
        //打印 set2 的 hashCode
        printbit("set1",set1);
        printbit("set2",set2);
        //调用打印程序 printbit(),打印对象中的每一个元素
        //打印 set1 的结果为 The bit set1 is: false true true true true
        set1.and(set2);
        printbit("set1 and set2",set1);
        //完成 set1 and set2,并打印结果
        set1.or(set2);
        printbit("set1 or set2",set1);
        //完成 set1 or set2,并打印结果
        set1.xor(set2);
        printbit("set1 xor set2",set1);
        //完成 set1 xor set2,并打印结果
    }
    //打印 BitSet 对象中的内容
    public static void printbit(String name,BitSet set){
        System.out.print("The bit "+name+" is: ");
        for(int i=0;i<N;I++)
            System.out.print(set.get(i)+" ");
        System.out.println();
    }
}
```

运行结果：

```
The set1 is: {1, 2, 3, 4}
The hash code of set2 is: 1225
The bit set1 is: false true true true true
The bit set2 is: true true false true true
The bit set1 and set2 is: false true false true true
The bit set1 or set2 is: true true false true true
The bit set1 xor set2 is: false false false false false
```

程序中使用了 BitSet 类提供的两种构造方法：

```
    public BitSet();
    public BitSet(int n);
```

参数 n 代表所创建的 BitSet 类的对象的大小。BitSet 类的对象的大小在必要时会由系统自动扩充。

其它方法：

public void set(int n)

将 BitSet对象的第 n 位设置成 1。

public void clear(int n)

将 BitSet对象的第 n 位清零。

public boolean get(int n)

读取位集合对象的第 n 位的值，它获取的是一个布尔值。当第n 位为 1 时，返回 true；第n 位为 0 时，返回 false。

另外，如在程序中所示，当把一 BitSet 类的对象转换成字符串输出时，输出的内容是此对象中 true 所处的位置。

在 BitSet 中提供了一组位操作，分别是：

public void and(BitSet set)

public void or(BitSet set)

public void xor(BitSet set)

利用它们可以完成两个位集合之间的与、或、异或操作。

BitSet 类中有一方法public int size()来取得位集合的大小，它的返回值与初始化时设定的位集合大小n 不一样，一般为 64。

1.1.2.2 Vector类与Enumeration接口

Vector类用于保存一组对象，由于 java 不支持动态数组，Vector 可以用于实现跟动态数组差不多的功能。如果要将一组对象存放在某种数据结构中，但是不能确定对象的个数时，Vector 是一个不错的选择。

例：将键盘上输入的一个数字序列的每位数字存储在vector 对象中，然后在屏幕上打印出各位数字相加的结果。

```
import java.util.*;  //Vector 类和 Enumeration 接口都在这个包中

public class TestVector

{

public static void main(String[] args)

{

Vector v=new Vector();

int b=0;

int num=0;

System.out.println("Please enter number:");

while(true)

{

try

{

b=System.in.read(); //从键盘读入一个字节内容

}

catch(Exception e)

{
```

```
e.printStackTrace();
}
if(b=='\r'||b=='\n') //如果是回车或换行的话，则退出 while 循环，即一串数据输入完成
{
break;
}
else
{
num=b-'0';
/*由于输入的是字符数字，它的数值是它的 ascii 码，例如‘0’=32;‘1’=33，
所以要想让输入的‘1’在计算机里为 1，必须减去 32，即‘0’*/
v.addElement(new Integer(num)); //将数字存入 vector
}
}
int sum=0;
Enumeration e=v.elements();
//取出 Vector 中的所有元素，必须使用 elements()方法，它返回一个 Enumeration 接口。
while(e.hasMoreElements())//如果当前指示器还指向一个对象，即还有数据
{
Integer intobj=(Integer)e.nextElement();
//取出当前指示器所指的对象，并将指示器指向下一个对象。
sum+=intobj.intValue(); //将 Integer 对象中所包装的整数取出来，并且加到 sum 中。
}
System.out.println(sum);//打印出这个和
}
}
```

Enumeration 的 nextelement（）方法返回的是指示器指示的对象，然后将指示器指向下一个对象。

由于 vector 可以存储各种类型的对象，所以编译器无法知道存储的是什么类型的对象，所以即使我们知道里面存储的是什么类型的，也要显示的说明它是什么类型的，如本例中的(Integer)e.nextElement();

Enumeration 接口实现了一种机制，通过这种机制，我们就可以只用 hasMoreElements()方法以及 nextElement()方法就可以实现所有对象的访问。

1.1.2.3 Collection接口与Iterator接口

Collection 接口的实现类跟 Vector 相似。要从实现了 Collection 接口的类的实例中取出保存在其中的元素对象，必须通过 Collection 接口的Iterator()方法，返回一个 Iterator 接口。

Iterator 接口与 Enumeration 接口非常相似。该接口的优点是其中的方法名比较简短。

Arraylist 类实现了 Collection 接口

例：将上例改写为用 ArrayList 类和 Iterator 接口来实现。

```
import java.util.*;

public class TestCollection

{

public static  void main(String[] args)

{

ArrayList v=new ArrayList();

int b=0;

int num=0;

System.out.println("Please enter  number:");

while(true)

{

try

{

b=System.in.read();

}

catch(Exception e)

{

e.printStackTrace();

}

if(b=='\r'||b=='\n')

{

break;

}

else
```

```
{

num=b-'0';


v.add(new Integer(num)); //方法为add（）而不是addelement（）

}

}

int sum=0;

Iterator e=v.iterator();


while(e.hasNext())//判断是否有数据时，使用的是hasNext()方法

{

Integer intobj=(Integer)e.next(); //取数据时使用next()方法，而不是nextElement()


sum+=intobj.intValue();

}

System.out.println(sum);

}

}
```

这两组实现的结果都是一样的，那什么时候使用哪种呢？

Vector类中的所有方法都是线程同步的，所有有两个以上的线程访问并发访问vector对象时，是安全的，但是只有一个线程访问时，仍然存在同步监视器检查的情况，需要额外的开销，影响了效率。而ArrayList中的所有方法是不同步的，所以程序中如果不存在多线程安全的问题，则ArrayList比Vector的效率高。如果存在多线程安全的问题，使用ArrayList要自己编写同步处理，而使用Vector则不要。

Collection,Set,List的区别如下：

Set,List是Collection的子类。

Collection各元素对象之间没有指定的顺序，允许有重复元素和多个Null元素对象。所以不可以排序，也不可以找出第几个第几个元素。

Set各元素对象之间没有指定的顺序，不允许有重复元素，最多允许有一个Null元素对象。

List各元素对象之间有指定的顺序，允许有重复元素和多个Null元素对象。

```
import java.util.*;
```

```
public class TestSort

{

public static  void main(String[] args)

{

ArrayList al=new ArrayList();     //新建一个arraylist ，由于它也实现了list接口，所以可以进行排序

al.add(new Integer(1));

al.add(new Integer(3));

al.add(new Integer(2));     //添加三个数字，随便添加，没有排序

System.out.println(al.toString());   //打印

Collections.sort(al);     //排序

/*Collections类本身并不是一个集合类，它只是提供了各种方法来操作集合类*/

System.out.println(al.toString());

}

}
```

1.1.2.4 Vector、ArrayList、List使用深入剖析

线性表，链表，哈希表是常用的数据结构，在进行Java开发时，JDK已经为我们提供了一系列相应的类来实现基本的数据结构。这些类均在java.util包中。本文试图通过简单的描述，向读者阐述各个类的作用以及如何正确使用这些类。

```
Collection
├List
│ ├LinkedList
│ ├ArrayList
│ └Vector
│    └Stack
└Set
Map
├Hashtable
├HashMap
└WeakHashMap
```

Collection接口

Collection是最基本的集合接口，一个Collection代表一组Object，即Collection的元素（Elements）。一些Collection允许相同的元素而另一些不行。一些能排序而另一些不行。Java SDK不提供直接继承自Collection的类，Java SDK提供的类都是继承自Collection的“子接口”如List和Set。

所有实现Collection接口的类都必须提供两个标准的构造函数：无参数的构造函数用于创建一个空的Collection，有一个Collection参数的构造函数用于创建一个新的Collection，这个新的Collection与传入的Collection有相同的元素。后一个构造函数允许用户复制一个Collection。

如何遍历 Collection 中的每一个元素？不论 Collection 的实际类型如何，它都支持一个 iterator()的方法，该方法返回一个迭代子，使用该迭代子即可逐一访问 Collection 中每一个元素。典型的用法如下：

```
Iterator it = collection.iterator(); // 获得一个迭代子
while(it.hasNext()) {
    Object obj = it.next(); // 得到下一个元素
}
```

由 Collection 接口派生的两个接口是 List 和 Set。

List 接口

List 是有序的 Collection，使用此接口能够精确的控制每个元素插入的位置。用户能够使用索引（元素在 List 中的位置，类似于数组下标）来访问 List 中的元素，这类似于 Java 的数组。

和下面要提到的 Set 不同，List 允许有相同的元素。

除了具有 Collection 接口必备的 iterator()方法外，List 还提供一个 listIterator()方法，返回一个 ListIterator 接口，和标准的 Iterator 接口相比，ListIterator 多了一些 add()之类的方法，允许添加，删除，设定元素，还能向前或向后遍历。

实现 List 接口的常用类有 LinkedList，ArrayList，Vector 和 Stack。

LinkedList 类

LinkedList 实现了 List 接口，允许 null 元素。此外 LinkedList 提供额外的 get，remove，insert 方法在 LinkedList 的首部或尾部。这些操作使 LinkedList 可被用作堆栈（stack），队列（queue）或双向队列（deque）。

注意 LinkedList 没有同步方法。如果多个线程同时访问一个 List，则必须自己实现访问同步。一种解决方法是在创建 List 时构造一个同步的 List:

```
List list = Collections.synchronizedList(new LinkedList(...));
```

ArrayList 类

ArrayList 实现了可变大小的数组。它允许所有元素，包括 null。ArrayList 没有同步。

size，isEmpty，get，set 方法运行时间为常数。但是 add 方法开销为分摊的常数，添加 n 个元素需要 O(n)的时间。其他的方法运行时间为线性。

每个 ArrayList 实例都有一个容量（Capacity），即用于存储元素的数组的大小。这个容量可随着不断添加新元素而自动增加，但是增长算法并没有定义。当需要插入大量元素时，在插入前可以调用 ensureCapacity 方法来增加 ArrayList 的容量以提高插入效率。

和 LinkedList 一样，ArrayList 也是非同步的（unsynchronized）。

Vector 类

Vector 非常类似 ArrayList，但是 Vector 是同步的。由 Vector 创建的 Iterator，虽然和 ArrayList 创建的 Iterator 是同一接口，但是，因为 Vector 是同步的，当一个 Iterator 被创建而且正在被使用，另一个线程改变了 Vector 的状态（例如，添加或删除了一些元素），这时调用 Iterator 的方法时将抛出 ConcurrentModificationException，因此必须捕获该异常。

Stack 类

Stack 继承自 Vector，实现一个后进先出的堆栈。Stack 提供 5 个额外的方法使得 Vector 得以被当作堆栈使用。基本的 push 和 pop 方法，还有 peek 方法得到栈顶的元素，empty 方法测试堆栈是否为空，search 方法检测一个元素在堆栈中的位置。Stack 刚创建后是空栈。

Set 接口

Set 是一种不包含重复的元素的 Collection，即任意的两个元素 e1 和 e2 都有 e1.equals(e2)=false，Set 最多有一个 null 元素。

很明显，Set 的构造函数有一个约束条件，传入的Collection 参数不能包含重复的元素。

请注意：必须小心操作可变对象（ Mutable Object）。如果一个 Set 中的可变元素改变了自身状态导致 Object.equals(Object)=true 将导致一些问题。

Map 接口

请注意，Map 没有继承 Collection 接口，Map 提供 key 到 value 的映射。一个 Map 中不能包含相同的 key，每个 key 只能映射一个 value。Map 接口提供 3 种集合的视图，Map 的内容可以被当作一组 key 集合，一组 value 集合，或者一组 key-value 映射。

Hashtable 类

Hashtable 继承 Map 接口，实现一个 key-value 映射的哈希表。任何非空（non-null）的对象都可作为 key 或者 value。

添加数据使用 put(key, value)，取出数据使用 get(key)，这两个基本操作的时间开销为常数。

Hashtable 通过 initial capacity 和 load factor 两个参数调整性能。通常缺省的 load factor 0.75 较好地实现了时间和空间的均衡。增大 load factor 可以节省空间但相应的查找时间将增大，这会影响像 get 和 put 这样的操作。

使用 Hashtable 的简单示例如下，将 1，2，3 放到 Hashtable 中，他们的 key 分别是”one”，”two”，”three”：

```
Hashtable numbers = new Hashtable();
numbers.put(“one”, new Integer(1));
numbers.put(“two”, new Integer(2));
numbers.put(“three”, new Integer(3));
```

要取出一个数，比如 2，用相应的 key：

```
Integer n = (Integer)numbers.get(“two”);
System.out.println(“two = ” + n);
```

由于作为 key 的对象将通过计算其散列函数来确定与之对应的 value 的位置，因此任何作为 key 的对象都必须实现 hashCode 和 equals 方法。hashCode 和 equals 方法继承自根类 Object，如果你用自定义的类当作 key 的话，要相当小心，按照散列函数的定义，如果两个对象相同，即 obj1.equals(obj2)=true，则它们的 hashCode 必须相同，但如果两个对象不同，则它们的 hashCode 不一定不同，如果两个不同对象的 hashCode 相同，这种现象称为冲突，冲突会导致操作哈希表的时间开销增大，所以尽量定义好的 hashCode()方法，能加快哈希表的操作。

如果相同的对象有不同的 hashCode，对哈希表的操作会出现意想不到的结果（期待的 get 方法返回 null），要避免这种问题，只需要牢记一条：要同时复写 equals 方法和 hashCode 方法，而不要只写其中一个。

Hashtable 是同步的。

HashMap 类

HashMap 和 Hashtable 类似，不同之处在于 HashMap 是非同步的，并且允许 null，即 null value 和 null key。，但是将 HashMap 视为 Collection 时（values()方法可返回 Collection），其迭代子操作时间开销和 HashMap 的容量成比例。因此，如果迭代操作的性能相当重要的话，不要将 HashMap 的初始化容量设得过高，或者 load factor 过低。

WeakHashMap 类

WeakHashMap 是一种改进的 HashMap，它对 key 实行“弱引用”，如果一个 key 不再被外部所引用，那么该 key 可以被 GC 回收。

总结

如果涉及到堆栈，队列等操作，应该考虑用 List，对于需要快速插入，删除元素，应该使用 LinkedList，如果需要快速随机访问元素，应该使用 ArrayList。

如果程序在单线程环境中，或者访问仅仅在一个线程中进行，考虑非同步的类，其效率较高，如果多个线程可能同时操作一个类，应该使用同步的类。

要特别注意对哈希表的操作，作为 key 的对象要正确复写 equals和 hashCode方法。

尽量返回接口而非实际的类型，如返回List而非ArrayList，这样如果以后需要将ArrayList 换成LinkedList 时，客户端代码不用改变。这就是针对抽象编程。

同步性

Vector 是同步的。这个类中的一些方法保证了Vector 中的对象是线程安全的。而 ArrayList 则是异步的，因此 ArrayList 中的对象并不是线程安全的。因为同步的要求会影响执行的效率，所以如果你不需要线程安全的集合那么使用 ArrayList 是一个很好的选择，这样可以避免由于同步带来的不必要的性能开销。

数据增长

从内部实现机制来讲 ArrayList 和 Vector 都是使用数组(Array)来控制集合中的对象。当你向这两种类型中增加元素的时候，如果元素的数目超出了内部数组目前的长度它们都需要扩展内部数组的长度，Vector 缺省情况下自动增长原来一倍的数组长度，ArrayList 是原来的 50%,所以最后你获得的这个集合所占的空间总是比你实际需要的要大。所以如果你要在集合中保存大量的数据那么使用 Vector 有一些优势，因为你可以通过设置集合的初始化大小来避免不必要的资源开销。

使用模式

在 ArrayList 和 Vector 中，从一个指定的位置（通过索引）查找数据或是在集合的末尾增加、移除一个元素所花费的时间是一样的，这个时间我们用 O(1)表示。但是，如果在集合的其他位置增加或移除元素那么花费的时间会呈线形增长：O(n-i)，其中 n 代表集合中元素的个数，i 代表元素增加或移除元素的索引位置。为什么会这样呢？以为在进行上述操作的时候集合中第 i 和第 i 个元素之后的所有元素都要执行位移的操作。这一切意味着什么呢？

这意味着，你只是查找特定位置的元素或只在集合的末端增加、移除元素，那么使用 Vector 或 ArrayList 都可以。如果是其他操作，你最好选择其他的集合操作类。比如，LinkList 集合类在增加或移除集合中任何位置的元素所花费的时间都是一样的?O(1)，但它在索引一个元素的使用缺比较慢—O(i),其中 i 是索引的位置.使用 ArrayList 也很容易，因为你可以简单的使用索引来代替创建 iterator 对象的操作。LinkList 也会为每个插入的元素创建对象，所有你要明白它也会带来额外的开销。

最后，在《Practical Java》一书中 Peter Haggar 建议使用一个简单的数组（Array）来代替 Vector 或 ArrayList。尤其是对于执行效率要求高的程序更应如此。因为使用数组(Array)避免了同步、额外的方法调用和不必要的重新分配空间的操作。

1.1.2.5 HashTable类

HashTable类不管可以像 Vector类一样动态的存储一系列的对象，而且对存储的每一个对象（称为值）都安排另一个对象（称为关键字）与它相关联。

用做关键字的类必须覆盖 Object.hashCode 方法和 Object.equals 方法，

因为要取出数据时，传递给 get（）方法的参数要跟里面的关键字比较，这时就要使用 equals（）方法。另外如果这两个关键字相等，他们的 hashcode（）返回值也要相等。

编程举例：使用自定义类作为 hashtable（）关键字的类。

```
public class MyKey

{

String name=null;

int age=0;

public boolean equals(Object obj)  //作为关键字的类要覆盖该方法

{
```

```
if(obj instanceof MyKey) //如果它是mykey 类型的，则先要转换为 mykey 类型，因为编译器不知道他是什么类型的
{
MyKey objtemp=(MyKey)obj; //然后下面再比较里面的内容是否相等。
if(name.equals(objtemp.name)&&age==objtemp.age)
{
return true;   //如果内容也相等，则相等
}
else
{
return false; //否则不等
}
}
else
{
return false; //如果它不是 mykey 类型的，则肯定不相等，因为类型都不同
}
}
public int hashCode()   //作为关键字的类要覆盖该方法
{
return name.hashCode()+age; //因为string 类已经覆盖了 hashcode（）方法，stringbuffer类没有覆盖该方法，所以不可以用做关键字类。
//如果name 和 age 有一个不等，返回值则不等
}
public MyKey(String name,int age)   //覆盖构造函数
{
this.name=name;
this.age=age;
}
public String toString() //如果不覆盖这个方法，则会产生乱码
{
```

```
return name+","+age;
}

}
```

```
import java.util.*;
public class HashTableTest
{
public static void main(String[] args)
{
Hashtable numbers=new Hashtable(); //新建一个 hashtable 类的实例
numbers.put(new MyKey("ZhangSan",18),new Integer(1)); 装入数据
numbers.put(new MyKey("LiSi",20),new Integer(2));
numbers.put(new MyKey("WangWu",16),new Integer(3));
Enumeration e=numbers.keys();
while(e.hasMoreElements())
{
MyKey Key=(MyKey)e.nextElement();
System.out.print(Key+"=");
System.out.println(numbers.get(Key));
}

}
}
```

1.1.2.6 Properties类

Properties 是 HashTable 的子类。增加了将hashtable 对象中的关键字和值保存到文件以及从文件中读取关键字和值到 hashtable 对象中的方法。

如果要用 properties.store（）方法存储 properties 中的内容，每个属性的关键字和值都必须是string 类型。

编程举例：使用 properties 把程序的启动运行次数记录在文件里，每次运行时打印出它的运行次数。

```
import java.util.*; //Properties 在这个包中
import java.io.*;   //fileinputstream 和 fileoutputstream 在这个包中
public class PropertiesFile
{
public static  void main(String[] args)
{
Properties settings=new Properties();
try
{
settings.load(new FileInputStream("count.txt")); //从文件"count.txt"装载
}
catch(Exception e)
{
settings.setProperty("count",String.valueOf(0));
//发生异常，说明第一次运行，第一次运行时还不存在文件“count.txt”，我们只能不从文件中取，而设置它的默认值为 0
}
int count=Integer.parseInt(settings.getProperty("count"))+1;
/*
本来可以使用从 hashtable 中继承的 get 方法，但是由于我们处理的是字符串，所以使用这个方法，
但由于返回的是字符串，所以转换成整数，由于存的是到上一次为止的运行次数，所以要加一。
*/
System.out.println("这是第"+count+"次运行！ ");
settings.setProperty("count",new Integer(count).toString());
/*
将次数存入 property 对象中，由于存入的是字符串，所以要转换成字符串。
*/
try
{
settings.store(new FileOutputStream("count.txt"),"program is used:");
```

```
//发生异常，说明第一次运行，这时会创建一个文件。
/*
将 property 对象中的结果存入文件，第一个参数为文件，第二个参数为标题。
*/
}
catch(Exception e)
{
e.printStackTrace();
}
}
}
```

1.1.2.7 读取 Properties 文件六种方法

1。使用 java.util.Properties 类的 load()方法

```
示例： InputStream in = lnew BufferedInputStream(new FileInputStream(name));
       Properties p = new Properties();
       p.load(in);
```

2。使用 java.util.ResourceBundle 类的 getBundle()方法

```
示例： ResourceBundle rb = ResourceBundle.getBundle(name, Locale.getDefault());
```

3。使用 java.util.PropertyResourceBundle 类的构造函数

```
示例： InputStream in = new BufferedInputStream(new FileInputStream(name));
       ResourceBundle rb = new PropertyResourceBundle(in);
```

4。使用 class 变量的 getResourceAsStream()方法

```
示例： InputStream in = JProperties.class.getResourceAsStream(name);
       Properties p = new Properties();
       p.load(in);
```

5。使用 class.getClassLoader()所得到的 java.lang.ClassLoader 的 getResourceAsStream()方法

示例： InputStream in = JProperties.class.getClassLoader().getResourceAsStream(name);

Properties p = new Properties();

p.load(in);

6。使用 java.lang.ClassLoader 类的 getSystemResourceAsStream()静态方法

示例： InputStream in = ClassLoader.getSystemResourceAsStream(name);

Properties p = new Properties();

p.load(in);

补充

Servlet 中可以使用 javax.servlet.ServletContext 的 getResourceAsStream()方法

示例：InputStream in = context.getResourceAsStream(path);

Properties p = new Properties();

p.load(in);

一般文件的 I/O

FileInputStream.read() //从本地文件读取二进制格式的数据

FileReader.read() //从本地文件读取字符（文本）数据

FileOutputStream.write() //保存二进制数据到本地文件

FileWriter.write() //保存字符数据到本地文件

XML 文件的 I/O

DocumentBuilderFactory.newDocumentBuilder().parse() //解析一个外部的 XML 文件，得到一个 Document 对象的 DOM 树

DocumentBuilderFactory.newInstance().newDocumentBuilder().newDocument() //初始化一棵 DOM 树

Document.getDocumentElement().appendChild() //为一个标签结点添加一个子结点

Document.createTextNode() //生成一个字符串结点

Node.getChildNodes() //取得某个结点的所有下一层子结点

Node.removeChild()　//删除某个结点的子结点

Document.getElementsByTagName() 查找所有指定名称的标签结点

Document.getElementById() //查找指定名称的一个标签结点，如果有多个符合，则返回某一个，通常是第一个

Element.getAttribute() //取得一个标签的某个属性的的值

Element.setAttribute() //设置一个标签的某个属性的的值

Element.removeAttribute() //删除一个标签的某个属性

TransformerFactory.newInstance().newTransformer().transform() //将一棵 DOM 树写入到外部 XML 文件

1.1.2.8　Timer类与TimerTask类

有个 schedule 方法，可以指定过多长时间定期的执行某个程序或某段代码，或者过多长时间启动一个线程等。

TimerTask 类实现了 Runnable 接口，要执行的类由它里面实现的 run 方法来完成。

编程实例：程序启动 30 秒后启动 windows 自带的计算器程序。

1.1.3　JAVA IO 包

1.1.3.1　存取程序状态几种方法—Java I-O 应用

1.1.3.1.1　文件 I/O：文件流→序列化

★文件流

文件操作是最简单最直接也是最容易想到的一种方式，我们说的文件操作不仅仅是通过 FileInputStream/FileOutputStream 这么“裸”的方式直接把数据写入到本地文件（像我以前写的一个扫雷的小游戏 JavaMine 就是这样保存一局的状态的），这样就比较“底层”了。

主要类与方法和描述

1. FileInputStream.read() *//从本地文件读取二进制格式的数据*
2. FileReader.read() *//从本地文件读取字符（文本）数据*
3. FileOutputStream.write() *//保存二进制数据到本地文件*
4. FileWriter.write() *//保存字符数据到本地文件*

★XML

和上面的单纯的 I/O 方式相比，XML 就显得“高档”得多，以至于成为一种数据交换的标准。以 DOM 方式为例，它关心的是首先在内存中构造文档树，数据保存在某个结点上（可以是叶子结点，也可以是标签结点的属性），构造好了以后一次性的写入到外部文件，但我们只需要知道文件的位置，并不知道 I/O 是怎么操作的，XML 操作方式可能多数人也实践过，所以这里也只列出相关的方法，供初学者预先了解一下。主要的包是 javax.xml.parsers，org.w3c.dom，javax.xml.transform。

主要类与方法和描述

1. DocumentBuilderFactory.newDocumentBuilder().parse() //解析一个外部的XML文件，得到一个Document对象的DOM树
2. DocumentBuilderFactory.newInstance().newDocumentBuilder().newDocument() //初始化一棵DOM树
3. Document.getDocumentElement().appendChild() //为一个标签结点添加一个子结点
4. Document.createTextNode() //生成一个字符串结点
5. Node.getChildNodes() //取得某个结点的所有下一层子结点
6. Node.removeChild() //删除某个结点的子结点
7. Document.getElementsByTagName() 查找所有指定名称的标签结点
8. Document.getElementById() //查找指定名称的一个标签结点，如果有多个符合，则返回某一个，通常是第一个
9. Element.getAttribute() //取得一个标签的某个属性的的值
10. Element.setAttribute() //设置一个标签的某个属性的的值
11. Element.removeAttribute() //删除一个标签的某个属性
12. TransformerFactory.newInstance().newTransformer().transform() //将一棵DOM树写入到外部XML文件

★序列化

使用基本的文件读写方式存取数据，如果我们仅仅保存相同类型的数据，则可以用同一种格式保存，譬如在我的JavaMine中保存一个盘局时，需要保存每一个方格的坐标、是否有地雷，是否被翻开等，这些信息组合成一个“复合类型”；相反，如果有多种不同类型的数据，那我们要么把它分解成若干部分，以相同类型（譬如String）保存，要么我们需要在程序中添加解析不同类型数据格式的逻辑，这就很不方便。于是我们期望用一种比较“高”的层次上处理数据，程序员应该花尽可能少的时间和代码对数据进行解析，事实上，序列化操作为我们提供了这样一条途径。

序列化（Serialization）大家可能都有所接触，它可以把对象以某种特定的编码格式写入或从外部字节流（即ObjectInputStream/ObjectOutputStream）中读取。序列化一个对象非常之简单，仅仅实现一下Serializable接口即可，甚至都不用为它专门添加任何方法：

```
1. public class MySerial implements java.io.Serializable
2. {
3.   //...
4. }
```

但有一个条件：即你要序列化的类当中，它的每个属性都必须是是“可序列化”的。这句话说起来有点拗口，其实所有基本类型（就是int,char,boolean之类的）都是“可序列化”的，而你可以看看JDK文档，会发现很多类其实已经实现了Serializable（即已经是“可序列化”的了），于是这些类的对象以及基本数据类型都可以直接作为你需要序列化的那个类的内部属性。如果碰到了不是“可序列化”的属性怎么办？对不起，那这个属性的类还需要事先实现Serializable接口，如此递归，直到所有属性都是“可序列化”的。

主要类与方法和描述

```
1.  ObjectOutputStream.writeObject() //将一个对象序列化到外部字节流
2.  ObjectInputStream.readObject() //从外部字节流读取并重新构造对象
```

从实际应用上看来，“Serializable”这个接口并没有定义任何方法，仿佛它只是一个标记（或者说像是Java的关键字）而已，一旦虚拟机看到这个“标记”，就会尝试调用自身预定义的序列化机制，除非你在实现Serializable接口的同时还定义了私有的readObject()或writeObject()方法。这一点很奇怪。不过你要是不愿意让系统使用缺省的方式进行序列化，那就必须定义上面提到的两个方法：

```
1.  public class MySerial implements java.io.Serializable
2.  {
3.    private void writeObject(java.io.ObjectOutputStream out) throws IOException
4.    {
5.      //...
6.    }
7.    private void readObject(java.io.ObjectInputStream in) throws IOException, ClassNotFoundException
8.    {
9.      //...
10.   }
11.   //...
12. }
```

譬如你可以在上面的writeObject()里调用默认的序列化方法ObjectOutputStream.defaultWriteObject();譬如你不愿意将某些敏感的属性和信息序列化，你也可以调用ObjectOutputStream.writeObject()方法明确指定需要序列化那些属性。关于用户可定制的序列化方法，我们将在后面提到。

★Bean

上面的序列化只是一种基本应用，你把一个对象序列化到外部文件以后，用notepad打开那个文件，只能从为数不多的一些可读字符中猜到这是有关这个类的信息文件，这需要你熟悉序列化文件的字节编码方式，那将是比较痛苦的（在《Core Java 2》第一卷里提到了相关编码方式，有兴趣的话可以查看参考资料），某些情况下我们可能需要被序列化的文件具有更好的可读性。另一方面，作为Java组件的核心概念“JavaBeans”，从JDK 1.4开始，其规范里也要求支持文本方式的“长期的持久化”（long-term persistence）。

打开JDK文档，java.beans包里的有一个名为“Encoder”的类，这就是一个可以序列化bean的实用类。和它相关的两个主要类有XMLEcoder和XMLDecoder，显然，这是以XML文件的格式保存和读取bean的工具。他们的用法也很简单，和上面ObjectOutputStream/ObjectInputStream比较类似。

主要类与方法和描述

```
1.  XMLEncoder.writeObject() //将一个对象序列化到外部字节流
```

```
2.  XMLDecoder.readObject() //从外部字节流读取并重新构造对象
```

如果一个bean是如下格式：

```
1.  public class MyBean
2.  {
3.    int i;
4.    char[] c;
5.    String s;
6.    //...(get和set操作省略)...
7.  }
```

那么通过XMLEcoder序列化出来的XML文件具有这样的形式：

```
1

    a
    b
    c

fox jump!
```

像 AWT和 Swing中很多可视化组件都是bean，当然也是可以用这种方式序列化的，下面就是从JDK文档中摘录的一个 JFrame 序列化以后的XML文件：

```
frame1
  0
  0
  200
  200
```

Hello

true

因此但你想要保存的数据是一些不是太复杂的类型的话，把它做成bean再序列化也不失为一种方便的选择。

★Properties

在以前我总结的一篇关于集合框架的小文章里提到过，Properties是历史集合类的一个典型的例子，这里主要不是介绍它的集合特性。大家可能都经常接触一些配置文件，如Windows的ini文件，Apache的conf文件，还有Java里的properties文件等，这些文件当中的数据以“关键字-值”对的方式保存。“环境变量”这个概念都知道吧，它也是一种“key-value”对，以前也常常看到版上问“如何取得系统某某信息”之类的问题，其实很多都保存在环境变量里，只要用一条

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

就能获得全部环境变量的列表：

```
-- listing properties --
java.runtime.name=Java(TM) 2 Runtime Environment, Stand...
sun.boot.library.path=C:\Program Files\Java\j2re1.4.2_05\bin
java.vm.version=1.4.2_05-b04
java.vm.vendor=Sun Microsystems Inc.
java.vendor.url=http://java.sun.com/
path.separator=;
java.vm.name=Java HotSpot(TM) Client VM
file.encoding.pkg=sun.io
user.country=CN
sun.os.patch.level=Service Pack 1
java.vm.specification.name=Java Virtual Machine Specification
user.dir=d:\my documents\项目\eclipse\SWTDemo
java.runtime.version=1.4.2_05-b04
java.awt.graphicsenv=sun.awt.Win32GraphicsEnvironment
java.endorsed.dirs=C:\Program Files\Java\j2re1.4.2_05\li...
os.arch=x86
java.io.tmpdir=C:\DOCUME~1\cn21x0q0\LOCALS~1\Temp\
line.separator=

java.vm.specification.vendor=Sun Microsystems Inc.
user.variant=
```

os.name=Windows XP

sun.java2d.fontpath=

java.library.path=C:\Program Files\Java\j2re1.4.2_05\bi...

java.specification.name=Java Platform API Specification

java.class.version=48.0

java.util.prefs.PreferencesFactory=java.util.prefs.WindowsPreferencesFac...

os.version=5.1

user.home=D:\Users\cn2lx0q0

user.timezone=

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

file.encoding=GBK

java.specification.version=1.4

user.name=cn2lx0q0

java.class.path=d:\my documents\项目\eclipse\SWTDemo\bi...

java.vm.specification.version=1.0

sun.arch.data.model=32

java.home=C:\Program Files\Java\j2re1.4.2_05

java.specification.vendor=Sun Microsystems Inc.

user.language=zh

awt.toolkit=sun.awt.windows.WToolkit

java.vm.info=mixed mode

java.version=1.4.2_05

java.ext.dirs=C:\Program Files\Java\j2re1.4.2_05\li...

sun.boot.class.path=C:\Program Files\Java\j2re1.4.2_05\li...

java.vendor=Sun Microsystems Inc.

file.separator=\

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

sun.cpu.endian=little

sun.io.unicode.encoding=UnicodeLittle

sun.cpu.isalist=pentium i486 i386

主要类与方法和描述

1. load() *//从一个外部流读取属性*
2. store() *//将属性保存到外部流（特别是文件）*
3. getProperty() *//取得一个指定的属性*
4. setProperty() *//设置一个指定的属性*
5. list() *//列出这个Properties对象包含的全部“key-value”对*
6. System.getProperties() *//取得系统当前的环境变量*

你可以这样保存一个properties文件：

```
1.  Properties prop = new Properties();
2.  prop.setProperty("key1", "value1");
3.  ...
4.  FileOutputStream out = new FileOutputStream("config.properties");
5.  prop.store(out, "—这里是文件头，可以加入注释—");
```

★Preferences

如果我说Java里面可以不使用JNI的手段操作Windows的注册表你信不信？很多软件的菜单里都有“Setting”或“Preferences”这样的选项用来设定或修改软件的配置，这些配置信息可以保存到一个像上面所述的配置文件当中，如果是Windows平台下，也可能会保存到系统注册表中。从JDK 1.4开始，Java在 java.util 下加入了一个专门处理用户和系统配置信息的 java.util.prefs包，其中一个类 Preferences 是一种比较“高级”的玩意。从本质上讲，Preferences本身是一个与平台无关的东西，但不同的OS对它的SPI（Service Provider Interface）的实现却是与平台相关的，因此，在不同的系统中你可能看到首选项保存为本地文件、LDAP目录项、数据库条目等，像在Windows平台下，它就保存到了系统注册表中。不仅如此，你还可以把首选项导出为XML文件或从XML文件导入。

主要类与方法和描述

```
1.  systemNodeForPackage() //根据指定的Class对象得到一个Preferences对象，这个对象的注册表路径是从“HKEY_LOCAL_MACHINE\”开始的
2.  systemRoot() //得到以注册表路径HKEY_LOCAL_MACHINE\SOFTWARE\Javasoft\Prefs 为根结点的Preferences对象
3.  userNodeForPackage() //根据指定的Class对象得到一个Preferences对象，这个对象的注册表路径是从“HKEY_CURRENT_USER\”开始的
4.  userRoot() //得到以注册表路径HKEY_CURRENT_USER\SOFTWARE\Javasoft\Prefs 为根结点的Preferences对象
5.  putXXX() //设置一个属性的值，这里XXX可以为基本数值型类型，如int、long等，但首字母大写，表示参数为相应的类型，也可以不写而直接用put，参数则为字符串
6.  getXXX() //得到一个属性的值
7.  exportNode() //将全部首选项导出为一个XML文件
8.  exportSubtree() //将部分首选项导出为一个XML文件
9.  importPreferences() //从XML文件导入首选项
```

你可以按如下步骤保存数据：

```
1.  Preferences myPrefs1 = Preferences.userNodeForPackage(this);// 这种方法是在“HKEY_CURRENT_USER\”下按当前类的路径建立一个注册表项
```

```
2.  Preferences myPrefs2 = Preferences.systemNodeForPackage(this);// 这种方法是在“HKEY_LOCAL_MACHINE\”下按当前类的路径建立一个注册表项
3.  Preferences myPrefs3 = Preferences.userRoot().node("com.jungleford.demo");// 这种方法是在“HKEY_CURRENT_USER\SOFTWARE\Javasoft\Prefs\”下按“com\jungleford\demo”的路径建立一个注册表项
4.  Preferences myPrefs4 = Preferences.systemRoot().node("com.jungleford.demo");// 这种方法是在“HKEY_LOCAL_MACHINE\SOFTWARE\Javasoft\Prefs\”下按“com\jungleford\demo”的路径建立一个注册表项
5.  myPrefs1.putInt("key1", 10);
6.  myPrefs1.putDouble("key2", -7.15);
7.  myPrefs1.put("key3", "value3");
8.  FileOutputStream out = new FileOutputStream("prefs.xml");
9.  myPrefs1.exportNode(out);
```

1.1.3.1.2 网络I/O：Socket→RMI

★Socket

Socket编程可能大家都很熟，所以就不多讨论了，只是说通过socket把数据保存到远端服务器或从网络socket读取数据也不失为一种值得考虑的方式。

★RMI

RMI机制其实就是RPC（远程过程调用）的Java版本，它使用socket作为基本传输手段，同时也是序列化最重要的一个应用。现在网络传输从编程的角度来看基本上都是以流的方式操作，socket就是一个例子，将对象转换成字节流的一个重要目标就是为了方便网络传输。

想象一下传统的单机环境下的程序设计，对于Java语言的函数（方法）调用（注意与C语言函数调用的区别）的参数传递，会有两种情况：如果是基本数据类型，这种情况下和C语言是一样的，采用值传递方式；如果是对象，则传递的是对象的引用，包括返回值也是引用，而不是一个完整的对象拷贝！试想一下在不同的虚拟机之间进行方法调用，即使是两个完全同名同类型的对象他们也很可能是不同的引用！此外对于方法调用过程，由于被调用过程的压栈，内存“现场”完全被被调用者占有，当被调用方法返回时，才将调用者的地址写回到程序计数器（PC），恢复调用者的状态，如果是两个虚拟机，根本不可能用简单压栈的方式来保存调用者的状态。因为种种原因，我们才需要建立RMI通信实体之间的“代理”对象，譬如“存根”就相当于远程服务器对象在客户机上的代理，stub就是这么来的，当然这是后话了。

本地对象与远程对象（未必是物理位置上的不同机器，只要不是在同一个虚拟机内皆为“远程”）之间传递参数和返回值，可能有这么几种情形：

- 值传递：这又包括两种子情形：如果是基本数据类型，那么都是“可序列化”的，统统序列化成可传输的字节流；如果是对象，而且不是“远程对象”（所谓“远程对象”是实现了java.rmi.Remote接口的对象），本来对象传递的应该是引用，但由于上述原因，引用是不足以证明对象身份的，所以传递的仍然是一个序列化的拷贝（当然这个对象也必须满足上述“可序列化”的条件）。

- 引用传递：可以引用传递的只能是“远程对象”。这里所谓的“引用”不要理解成了真的只是一个符号，它其实是一个留在（客户机）本地stub中的，和远端服务器上那个真实的对象张得一模一样的镜像而已！只是因为它有点“特权”（不需要经过序列化），在本地内存里已经有了一个实例，真正引用的其实是这个“孪生子”。

由此可见，序列化在RMI当中占有多么重要的地位。

1.1.3.1.3 数据库I/O：CMP、Hibernate

★什么是“Persistence”

用过VMWare的朋友大概都知道当一个guest OS正在运行的时候点击“Suspend”将虚拟OS挂起，它会把整个虚拟内存的内容保存到磁盘上，譬如你为虚拟OS分配了128M的运行内存，那挂起以后你会在虚拟OS所在的目录下找到一个同样是128M的文件，这就是虚拟OS内存的完整镜像！这种内存的镜像手段其实就是“Persistence”（持久化）概念的由来。

★CMP和Hibernate

因为我对J2EE的东西不是太熟悉，随便找了点材料看看，所以担心说的不到位，这次就不作具体总结了，人要学习……真是一件痛苦的事情 

1.1.3.1.4 序列化再探讨

从以上技术的讨论中我们不难体会到，序列化是Java之所以能够出色地实现其鼓吹的两大卖点??分布式（distributed）和跨平台（OS independent）的一个重要基础。TIJ（即“Thinking in Java”）谈到I/O系统时，把序列化称为“lightweight persistence”??“轻量级的持久化”，这确实很有意思。

★为什么叫做“序列”化？

开场白里我说更习惯于把“Serialization”称为“序列化”而不是“串行化”，这是有原因的。介绍这个原因之前先回顾一些计算机基本的知识，我们知道现代计算机的内存空间都是线性编址的（什么是“线性”知道吧，就是一个元素只有一个唯一的“前驱”和唯一的“后继”，当然头尾元素是个例外；对于地址来说，它的下一个地址当然不可能有两个，否则就乱套了），“地址”这个概念推广到数据结构，就相当于“指针”，这个在本科低年级大概就知道了。注意了，既然是线性的，那“地址”就可以看作是内存空间的“序号”，说明它的组织是有顺序的，“序号”或者说“序列号”正是“Serialization”机制的一种体现。为什么这么说呢？譬如我们有两个对象a和b,分别是类A和B的实例，它们都是可序列化的，而A和B都有一个类型为C的属性，根据前面我们说过的原则，C当然也必须是可序列化的。

```
1.  import java.io.*;
2.  ...
3.  class A implements Serializable
4.  {
5.    C c;
6.    ...
7.  }
8.
9.  class B implements Serializable
10. {
```

```
11.    C c;
12.    ...
13.  }
14.
15.  class C implements Serializable
16.  {
17.    ...
18.  }
19.
20.  A a;
21.  B b;
22.  C c1;
23.  ...
```

注意，这里我们在实例化a和b的时候，有意让他们的c属性使用同一个C类型对象的引用，譬如c1，那么请试想一下，但我们序列化a和b的时候，它们的c属性在外部字节流（当然可以不仅仅是文件）里保存的是一份拷贝还是两份拷贝呢？序列化在这里使用的是一种类似于“指针”的方案：它为每个被序列化的对象标上一个“序列号”（serial number），但序列化一个对象的时候，如果其某个属性对象是已经被序列化的，那么这里只向输出流写入该属性的序列号；从字节流恢复被序列化的对象时，也根据序列号找到对应的流来恢复。这就是“序列化”名称的由来！这里我们看到“序列化”和“指针”是极相似的，只不过“指针”是内存空间的地址链，而序列化用的是外部流中的“序列号链”。

使用“序列号”而不是内存地址来标识一个被序列化的对象，是因为从流中恢复对象到内存，其地址可能就未必是原来的地址了??我们需要的只是这些对象之间的引用关系，而不是死板的原始位置，这在RMI中就更是必要，在两台不同的机器之间传递对象（流），根本就不可能指望它们在两台机器上都具有相同的内存地址。

★更灵活的“序列化”：transient属性和Externalizable

Serializable确实很方便，方便到你几乎不需要做任何额外的工作就可以轻松将内存中的对象保存到外部。但有两个问题使得Serializable的威力收到束缚：

一个是效率问题，《Core Java 2》中指出，Serializable使用系统默认的序列化机制会影响软件的运行速度，因为需要为每个属性的引用编号和查号，再加上I/O操作的时间（I/O和内存读写差的可是一个数量级的大小），其代价当然是可观的.

另一个困扰是“裸”的Serializable不可定制，傻乎乎地什么都给你序列化了，不管你是不是想这么做。其实你可以有至少三种定制序列化的选择。其中一种前面已经提到了，就是在implements Serializable的类里面添加私有的writeObject()和readObject()方法（这种Serializable就不裸了，  ），在这两个方法里，该序列化什么，不该序列化什么，那就由你说了算了，你当然可以在这两个方法体里面分别调用ObjectOutputStream.defaultWriteObject()和ObjectInputStream.defaultReadObject()仍然执行默认的序列化动作（那你在代码上不就做无用功了？呵呵），也可以用ObjectOutputStream.writeObject()和ObjectInputStream.readObject()方法对你中意的属性进行序列化。但虚拟机一看到你定义了这两个方法，它就不再用默认的机制了。

如果仅仅为了跳过某些属性不让它序列化，上面的动作似乎显得麻烦，更简单的方法是对不想序列化的属性加上transient

关键字，说明它是个“暂态变量”，默认序列化的时候就不会把这些属性也塞到外部流里了。当然，你如果定义writeObject()和readObject()方法的化，仍然可以把暂态变量进行序列化。题外话，像transient、violate、finally这样的关键字初学者可能会不太重视，而现在有的公司招聘就偏偏喜欢问这样的问题 :(

再一个方案就是不实现Serializable而改成实现 Externalizable接口。我们研究一下这两个接口的源代码，发现它们很类似，甚至容易混淆。我们要记住的是：Externalizable默认并不保存任何对象相关信息！任何保存和恢复对象的动作都是你自己定义的。Externalizable包含两个public的方法：

```
1.  public void writeExternal(ObjectOutput out) throws IOException;
2.  public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException;
```

乍一看这和上面的writeObject()和readObject()几乎差不多，但Serializable和Externalizable走的是两个不同的流程：Serializable在对象不存在的情况下，就可以仅凭外部的字节序列把整个对象重建出来；但Externalizable在重建对象时，先是调用该类的默认构造函数（即不含参数的那个构造函数）使得内存中先有这么一个实例，然后再调用readExternal方法对实例中的属性进行恢复，因此，如果默认构造函数中和readExternal方法中都没有赋值的那些属性，特别他们是非基本类型的话，将会是空（null）。在这里需要注意的是，transient只能用在对Serializable而不是Externalizable的实现里面。

★序列化与克隆

从“可序列化”的递归定义来看，一个序列化的对象貌似对象内存映象的外部克隆，如果没有共享引用的属性的化，那么应该是一个深度克隆。关于克隆的话题有可以谈很多，这里就不细说了，有兴趣的话可以参考IBM developerWorks上的一篇文章：JAVA中的指针,引用及对象的clone

1.1.3.1.5 一点启示

作为一个实际的应用，我在写那个简易的邮件客户端JExp的时候曾经对比过好几种保存Message对象（主要是几个关键属性和邮件的内容）到本地的方法，譬如XML、Properties等，最后还是选择了用序列化的方式，因为这种方法最简单， 大约可算是“学以致用”罢。这里“存取程序状态”其实只是一个引子话题罢了，我想说的是??就如同前面我们讨论的关于logging的话题一样??在Java面前对同一个问题你可以有很多种solution：熟悉文件操作的，你可能会觉得Properties、XML或Bean比较方便，然后又发现了还有Preferences这么一个东东，大概又会感慨“天外有天”了，等到你接触了很多种新方法以后，结果又会“殊途同归”，重新反省Serialization机制本身。这不仅是Java，科学也是同样的道理。

1.1.3.2 调整 JavaTM I/O 性能

1.1.3.2.1 加速I/O的基本规则

作为这个讨论的开始，这里有几个如何加速I/O的基本规则:

1. 避免访问磁盘
2. 避免访问底层的操作系统
3. 避免方法调用
4. 避免个别的处理字节和字符

很明显这些规则不能在所有的问题上避免，因为如果能够的话就没有实际的I/O被执行。考虑下面的计算文件中的新行符('\n')的三部分范例。

方法1: read方法

第一个方法简单的使用FileInputStream的read方法:

```
import java.io.*;


public class intro1 {

     public static void main(String args[]) {

          if (args.length != 1) {

               System.err.println("missing filename");

               System.exit(1);

          }

          try {

               FileInputStream fis = new FileInputStream(args[0]);

               int cnt = 0;

               int b;

               while ((b = fis.read()) != -1) {

                    if (b == '\n')

                         cnt++;

               }

               fis.close();

               System.out.println(cnt);

          } catch (IOException e) {

               System.err.println(e);

          }

     }
```

}然而这个方法触发了大量的底层运行时系统调用——FileInputStream.read——返回文件的下一个字节的本机方法。

方法 2: 使用大缓冲区

第二种方法使用大缓冲区避免了上面的问题:

```
import java.io.*;


public class intro2 {
```

```
    public static void main(String args[]) {
        if (args.length != 1) {
            System.err.println("missing filename");
            System.exit(1);
        }
        try {
            FileInputStream fis = new FileInputStream(args[0]);
            BufferedInputStream bis = new BufferedInputStream(fis);
            int cnt = 0;
            int b;
            while ((b = bis.read()) != -1) {
                if (b == '\n')
                    cnt++;
            }
            bis.close();
            System.out.println(cnt);
        } catch (IOException e) {
            System.err.println(e);
        }
    }
}
```

BufferedInputStream.read 从输入缓冲区获取下一个字节，仅仅只访问了一次底层系统。

方法 3: 直接缓冲

第三种方法避免使用 BufferedInputStream 而直接缓冲，因此排除了 read 方法的调用:

```
import java.io.*;

public class intro3 {
    public static void main(String args[]) {
        if (args.length != 1) {
            System.err.println("missing filename");
```

```
                System.exit(1);
            }
            try {
                FileInputStream fis = new FileInputStream(args[0]);
                byte buf[] = new byte[2048];
                int cnt = 0;
                int n;
                while ((n = fis.read(buf)) != -1) {
                    for (int i = 0; i < n; i++) {
                        if (buf[i] == '\n')
                            cnt++;
                    }
                }
                fis.close();
                System.out.println(cnt);
            } catch (IOException e) {
                System.err.println(e);
            }
        }
}
```

对于一个 1 MB 的输入文件，以秒为单位的执行时间是:

intro1 6.9 intro2 0.9 intro3 0.4

或者说在最慢的方法和最快的方法间是17比1的不同。

这个巨大的加速并不能证明你应该总是使用第三种方法，即自己做缓冲。这可能是一个错误的倾向特别是在处理文件结束事件时没有仔细的实现。在可读性上它也没有其它方法好。但是记住时间花费在哪儿了以及在必要的时候如何矫正是很有用。

方法2 或许是对于大多应用的 "正确" 方法.

1.1.3.2.2 缓冲

方法 2 和 3 使用了*缓冲*技术，大块文件被从磁盘读取，然后每次访问一个字节或字符。缓冲是一个基本而重要的加速I/O 的技术,而且有几个类支持缓冲(BufferedInputStream 用于字节, BufferedReader 用于字符)。

一个明显得问题是: 缓冲区越大I/O越快吗？典型的Java缓冲区长1024 或者 2048 字节，一个更大的缓冲区有可能加速 I/O 但是只能占很小的比重，大约5 到 10%。

方法4: 整个文件

缓冲的极端情况是事先决定整个文件的长度，然后读取整个文件:

```
import java.io.*;


public class readfile {

     public static void main(String args[]) {

          if (args.length != 1) {

               System.err.println("missing filename");

               System.exit(1);

          }

          try {

               int len = (int) (new File(args[0]).length());

               FileInputStream fis = new FileInputStream(args[0]);

               byte buf[] = new byte[len];

               fis.read(buf);

               fis.close();

               int cnt = 0;

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

                    if (buf[i] == '\n')

                         cnt++;

               }

               System.out.println(cnt);

          } catch (IOException e) {

               System.err.println(e);

          }

     }
```

}这个方法很方便，在这里文件被当作一个字节数组。但是有一个明显得问题是有可能没有读取一个巨大的文件的足够的内存。

缓冲的另一个方面是向窗口终端的文本输出。缺省情况下， System.out （ 一个PrintStream）是行缓冲的，这意味着在遇到一个新行符后输出缓冲区被提交。对于交互来说这是很重要的，在那种情况下你可能喜欢在实际的输出前显示一个输入提示。

方法 5: 关闭行缓冲

行缓冲可以被禁止，像下面的例子那样:

```
import java.io.*;


public class bufout {

     public static void main(String args[]) {

          FileOutputStream fdout = new FileOutputStream(FileDescriptor.out);

          BufferedOutputStream bos = new BufferedOutputStream(fdout, 1024);

          PrintStream ps = new PrintStream(bos, false);

          System.setOut(ps);

          final int N = 100000;

          for (int i = 1; i <= N; i++)

               System.out.println(i);

          ps.close();

     }

}
```

这个程序输出整数 1 到 100000 缺省输出，比在缺省的行缓冲情况下快了三倍。

缓冲也是下面将要展示的例子的重要部分，在那里，缓冲区被用于加速文件随机访问。

1.1.3.2.3 读写文本文件

早些时候曾提到从文件里面读取字符的方法调用的消耗可能是重大的。这个问题在计算文本文件的行数的另一个例子中也可以找到。:

```
import java.io.*;


public class line1 {

     public static void main(String args[]) {

          if (args.length != 1) {

               System.err.println("missing filename");

               System.exit(1);

          }

          try {

               FileInputStream fis = new FileInputStream(args[0]);
```

```
                BufferedInputStream bis = new BufferedInputStream(fis);
                DataInputStream dis = new DataInputStream(bis);
                int cnt = 0;
                while (dis.readLine() != null)
                    cnt++;
                dis.close();
                System.out.println(cnt);
            } catch (IOException e) {
                System.err.println(e);
            }
        }
```

}这个程序使用老的DataInputStream.readLine 方法，该方法是使用用读取每个字符的 read 方法实现的。一个新方法是:

```
import java.io.*;

public class line2 {
    public static void main(String args[]) {
        if (args.length != 1) {
            System.err.println("missing filename");
            System.exit(1);
        }
        try {
            FileReader fr = new FileReader(args[0]);
            BufferedReader br = new BufferedReader(fr);
            int cnt = 0;
            while (br.readLine() != null)
                cnt++;
            br.close();
            System.out.println(cnt);
        } catch (IOException e) {
            System.err.println(e);
```

```
        }
    }
```

}这个方法更快。例如在一个有200,000行的 6 MB文本文件上，第二个程序比第一个快大约20%。

但是即使第二个程序不是更快的，第一个程序依然有一个重要的问题要注意。第一个程序在Java™ 2编译器下引起了不赞成警告，因为DataInputStream.readLine太陈旧了。它不能恰当的将字节转换为字符，因此在操作包含非ASCII字符的文本文件时可能是不合适的选择。(Java语言使用Unicode字符集而不是ASCII)

这就是早些时候提到的字节流和字符流之间的区别。像这样的一个程序:

```
import java.io.*;


public class conv1 {
    public static void main(String args[]) {
        try {
            FileOutputStream fos = new FileOutputStream("out1");
            PrintStream ps = new PrintStream(fos);
            ps.println("\uffff\u4321\u1234");
            ps.close();
        } catch (IOException e) {
            System.err.println(e);
        }
    }
```

}向一个文件里面写，但是没有保存实际的Unicode字符输出。Reader/Writer I/O 类是基于字符的，被设计用来解决这个问题。OutputStreamWriter 应用于字节编码的字符。

一个使用PrintWriter写入Unicode字符的程序是这样的:

```
import java.io.*;


public class conv2 {
    public static void main(String args[]) {
        try {
            FileOutputStream fos = new FileOutputStream("out2");
            OutputStreamWriter osw = new OutputStreamWriter(fos, "UTF8");
            PrintWriter pw = new PrintWriter(osw);
```

```
                pw.println("\uffff\u4321\u1234");
                pw.close();
            } catch (IOException e) {
                System.err.println(e);
            }
        }
}
```

这个程序使用UTF8编码，具有ASCII文本是本身而其他字符是两个或三个字节的特性。

1.1.3.2.4 格式化的代价

实际上向文件写数据只是输出代价的一部分。另一个可观的代价是数据格式化。考虑一个三部分程序，它像下面这样输出一行:

The square of 5 is 25

方法 1

第一种方法简单的输出一个固定的字符串，了解固有的I/O开销:

```
  public class format1 {
      public static void main(String args[]) {
          final int COUNT = 25000;
          for (int i = 1; i <= COUNT; i++) {
              String s = "The square of 5 is 25\n";
              System.out.print(s);
          }
      }
}
```

方法2

第二种方法使用简单格式"+":

```
public class format2 {
      public static void main(String args[]) {
          int n = 5;
          final int COUNT = 25000;
          for (int i = 1; i <= COUNT; i++) {
              String s = "The square of " + n + " is " + n * n + "\n";
```

```
                System.out.print(s);
            }
        }
}
```

方法 3

第三种方法使用 java.text 包中的 MessageFormat 类:

```
import java.text.*;

public class format3 {
    public static void main(String args[]) {
        MessageFormat fmt = new MessageFormat("The square of {0} is {1}\n");
        Object values[] = new Object[2];
        int n = 5;
        values[0] = new Integer(n);
        values[1] = new Integer(n * n);
        final int COUNT = 25000;
        for (int i = 1; i <= COUNT; i++) {
            String s = fmt.format(values);
            System.out.print(s);
        }
    }
}
```

这些程序产生同样的输出。运行时间是:

```
  format1   1.3 format2   1.8 format3   7.8
```

或者说最慢的和最快的大约是6 比 1。如果格式没有预编译第三种方法将更慢，使用静态的方法代替:

方法 4

```
MessageFormat.format(String, Object[])

import java.text.*;
```

```
public class format4 {

     public static void main(String args[]) {

          String fmt = "The square of {0} is {1}\n";

          Object values[] = new Object[2];

          int n = 5;

          values[0] = new Integer(n);

          values[1] = new Integer(n * n);

          final int COUNT = 25000;

          for (int i = 1; i <= COUNT; i++) {

               String s = MessageFormat.format(fmt, values);

               System.out.print(s);

          }

     }

}
```

这比前一个例子多花费1/3的时间。

第三个方法比前两种方法慢很多的事实并不意味着你不应该使用它，而是你要意识到时间上的开销。

在国际化的情况下信息格式化是很重要的，关心这个问题的应用程序通常从一个绑定的资源中读取格式然后使用它。

1.1.3.2.5 随机访问

RandomAccessFile 是一个进行随机文件I/O(在字节层次上)的类。这个类提供一个seek方法，和 C/C++中的相似,移动文件指针到任意的位置，然后从那个位置字节可以被读取或写入。

seek方法访问底层的运行时系统因此往往是消耗巨大的。一个更好的代替是在RandomAccessFile上建立你自己的缓冲，并实现一个直接的字节read方法。read方法的参数是字节偏移量（>= 0）。这样的一个例子是:

```
 import java.io.*;


public class ReadRandom {

     private static final int DEFAULT_BUFSIZE = 4096;


     private RandomAccessFile raf;


     private byte inbuf[];
```

```
private long startpos = -1;

private long endpos = -1;

private int bufsize;

public ReadRandom(String name) throws FileNotFoundException {
      this(name, DEFAULT_BUFSIZE);
}

public ReadRandom(String name, int b) throws FileNotFoundException {
      raf = new RandomAccessFile(name, "r");
      bufsize = b;
      inbuf = new byte[bufsize];
}

public int read(long pos) {
      if (pos < startpos || pos > endpos) {
            long blockstart = (pos / bufsize) * bufsize;
            int n;
            try {
                  raf.seek(blockstart);
                  n = raf.read(inbuf);
            } catch (IOException e) {
                  return -1;
            }
            startpos = blockstart;
            endpos = blockstart + n - 1;
            if (pos < startpos || pos > endpos)
                  return -1;
```

```
            }
            return inbuf[(int) (pos - startpos)] & 0xffff;
        }

        public void close() throws IOException {
            raf.close();
        }

        public static void main(String args[]) {
            if (args.length != 1) {
                System.err.println("missing filename");
                System.exit(1);
            }
            try {
                ReadRandom rr = new ReadRandom(args[0]);
                long pos = 0;
                int c;
                byte buf[] = new byte[1];
                while ((c = rr.read(pos)) != -1) {
                    pos++;
                    buf[0] = (byte) c;
                    System.out.write(buf, 0, 1);
                }
                rr.close();
            } catch (IOException e) {
                System.err.println(e);
            }
        }
}
```

这个程序简单的读取字节序列然后输出它们。

如果有访问位置，这个技术是很有用的，文件中的附近字节几乎在同时被读取。例如，如果你在一个排序的文件上实现二分法查找，这个方法可能很有用。如果你在一个巨大的文件上的任意点做随机访问的话就没有太大价值。

1.1.3.2.6 压缩

Java提供用于压缩和解压字节流的类，这些类包含在java.util.zip 包里面，这些类也作为 *Jar* 文件的服务基础（ *Jar* 文件是带有附加文件列表的 *Zip* 文件）。

下面的程序接收一个输入文件并将之写入一个只有一项的压缩的 *Zip* 文件:

```
 import java.io.*;
import java.util.zip.*;


public class compress {
     public static void doit(String filein, String fileout) {
          FileInputStream fis = null;
          FileOutputStream fos = null;
          try {
               fis = new FileInputStream(filein);
               fos = new FileOutputStream(fileout);
               ZipOutputStream zos = new ZipOutputStream(fos);
               ZipEntry ze = new ZipEntry(filein);
               zos.putNextEntry(ze);
               final int BUFSIZ = 4096;
               byte inbuf[] = new byte[BUFSIZ];
               int n;
               while ((n = fis.read(inbuf)) != -1)
                    zos.write(inbuf, 0, n);
               fis.close();
               fis = null;
               zos.close();
               fos = null;
          } catch (IOException e) {
               System.err.println(e);
          } finally {
```

```
                try {
                    if (fis != null)
                        fis.close();
                    if (fos != null)
                        fos.close();
                } catch (IOException e) {
                }
            }
        }

        public static void main(String args[]) {
            if (args.length != 2) {
                System.err.println("missing filenames");
                System.exit(1);
            }
            if (args[0].equals(args[1])) {
                System.err.println("filenames are identical");
                System.exit(1);
            }
            doit(args[0], args[1]);
        }
}
```

下一个程序执行相反的过程，将一个假设只有一项的Zip文件作为输入然后将之解压到输出文件:

```
 import java.io.*;
import java.util.zip.*;

public class uncompress {
    public static void doit(String filein, String fileout) {
        FileInputStream fis = null;
        FileOutputStream fos = null;
```

```
        try {
            fis = new FileInputStream(filein);
            fos = new FileOutputStream(fileout);
            ZipInputStream zis = new ZipInputStream(fis);
            ZipEntry ze = zis.getNextEntry();
            final int BUFSIZ = 4096;
            byte inbuf[] = new byte[BUFSIZ];
            int n;
            while ((n = zis.read(inbuf, 0, BUFSIZ)) != -1)
                fos.write(inbuf, 0, n);
            zis.close();
            fis = null;
            fos.close();
            fos = null;
        } catch (IOException e) {
            System.err.println(e);
        } finally {
            try {
                if (fis != null)
                    fis.close();
                if (fos != null)
                    fos.close();
            } catch (IOException e) {
            }
        }
    }

    public static void main(String args[]) {
        if (args.length != 2) {
            System.err.println("missing filenames");
```

```
            System.exit(1);
        }
        if (args[0].equals(args[1])) {
            System.err.println("filenames are identical");
            System.exit(1);
        }
        doit(args[0], args[1]);
    }
}
```

压缩是提高还是损害I/O性能很大程度依赖你的硬件配置，特别是和处理器和磁盘驱动器的速度相关。使用Zip技术的压缩通常意味着在数据大小上减少50%，但是代价是压缩和解压的时间。一个巨大(5到10 MB)的压缩文本文件，使用带有IDE硬盘驱动器的300-MHz Pentium PC从硬盘上读取可以比不压缩少用大约1/3的时间。

压缩的一个有用的范例是向非常慢的媒介例如软盘写数据。使用高速处理器(300 MHz Pentium)和低速软驱(PC上的普通软驱)的一个测试显示压缩一个巨大的文本文件然后在写入软盘比直接写入软盘快大约50% 。

1.1.3.2.7 高速缓存

关于硬件的高速缓存的详细讨论超出了本文的讨论范围。但是在有些情况下软件高速缓存能被用于加速I/O。考虑从一个文本文件里面以随机顺序读取一行的情况，这样做的一个方法是读取所有的行，然后把它们存入一个ArrayList (一个类似Vector的集合类):

```
import java.io.*;
import java.util.ArrayList;

public class LineCache {
    private ArrayList list = new ArrayList();

    public LineCache(String fn) throws IOException {
        FileReader fr = new FileReader(fn);
        BufferedReader br = new BufferedReader(fr);
        String ln;
        while ((ln = br.readLine()) != null)
            list.add(ln);
        br.close();
    }
```

```
    public String getLine(int n) {
        if (n < 0)
            throw new IllegalArgumentException();
        return (n < list.size() ? (String) list.get(n) : null);
    }

    public static void main(String args[]) {
        if (args.length != 1) {
            System.err.println("missing filename");
            System.exit(1);
        }
        try {
            LineCache lc = new LineCache(args[0]);
            int i = 0;
            String ln;
            while ((ln = lc.getLine(i++)) != null)
                System.out.println(ln);
        } catch (IOException e) {
            System.err.println(e);
        }
    }
}
```

getLine 方法被用来获取任意行。这个技术是很有用的，但是很明显对一个大文件使用了太多的内存，因此有局限性。一个代替的方法是简单的记住被请求的行最近的100行，其它的请求直接从磁盘读取。这个安排在局域性的访问时很有用，但是在真正的随机访问时没有太大作用。

1.1.3.2.8 分解

分解 是指将字节或字符序列分割为像单词这样的逻辑块的过程。Java 提供StreamTokenizer 类，像下面这样操作:

```
 import java.io.*;

public class token1 {
```

```
    public static void main(String args[]) {
        if (args.length != 1) {
            System.err.println("missing filename");
            System.exit(1);
        }
        try {
            FileReader fr = new FileReader(args[0]);
            BufferedReader br = new BufferedReader(fr);
            StreamTokenizer st = new StreamTokenizer(br);
            st.resetSyntax();
            st.wordChars('a', 'z');
            int tok;
            while ((tok = st.nextToken()) != StreamTokenizer.TT_EOF) {
                if (tok == StreamTokenizer.TT_WORD)
                    ;// st.sval has token
            }
            br.close();
        } catch (IOException e) {
            System.err.println(e);
        }
    }
}
```

这个例子分解小写单词（字母a-z）。如果你自己实现同等地功能，它可能像这样：

```
import java.io.*;

public class token2 {
    public static void main(String args[]) {
        if (args.length != 1) {
            System.err.println("missing filename");
            System.exit(1);
```

```
        }
        try {
            FileReader fr = new FileReader(args[0]);
            BufferedReader br = new BufferedReader(fr);
            int maxlen = 256;
            int currlen = 0;
            char wordbuf[] = new char[maxlen];
            int c;
            do {
                c = br.read();
                if (c >= 'a' && c <= 'z') {
                    if (currlen == maxlen) {
                        maxlen *= 1.5;
                        char xbuf[] = new char[maxlen];
                        System.arraycopy(wordbuf, 0, xbuf, 0, currlen);
                        wordbuf = xbuf;
                    }
                    wordbuf[currlen++] = (char) c;
                } else if (currlen > 0) {
                    String s = new String(wordbuf, 0, currlen); // do something
                                                                // with s
                    currlen = 0;
                }
            } while (c != -1);
            br.close();
        } catch (IOException e) {
            System.err.println(e);
        }
    }
}
```

第二个程序比前一个运行快大约 20%，代价是写一些微妙的底层代码。

StreamTokenizer 是一种混合类，它从字符流(例如 BufferedReader)读取，但是同时以字节的形式操作，将所有的字符当作双字节(大于 0xff) ，即使它们是字母字符。

1.1.3.2.9 串行化

串行化 以标准格式将任意的 Java 数据结构转换为字节流。例如，下面的程序输出随机整数数组:

```
import java.io.*;
import java.util.*;

public class serial1 {
     public static void main(String args[]) {
          ArrayList al = new ArrayList();
          Random rn = new Random();
          final int N = 100000;
          for (int i = 1; i <= N; i++)
               al.add(new Integer(rn.nextInt()));
          try {
               FileOutputStream fos = new FileOutputStream("test.ser");
               BufferedOutputStream bos = new BufferedOutputStream(fos);
               ObjectOutputStream oos = new ObjectOutputStream(bos);
               oos.writeObject(al);
               oos.close();
          } catch (Throwable e) {
               System.err.println(e);
          }
     }
}
```

而下面的程序读回数组:

```
  import java.io.*;
```

```
import java.util.*;


public class serial2 {

     public static void main(String args[]) {

          ArrayList al = null;

          try {

               FileInputStream fis = new FileInputStream("test.ser");

               BufferedInputStream bis = new BufferedInputStream(fis);

               ObjectInputStream ois = new ObjectInputStream(bis);

               al = (ArrayList) ois.readObject();

               ois.close();

          } catch (Throwable e) {

               System.err.println(e);

          }

     }

}
```

注意我们使用缓冲提高I/O操作的速度。

有比串行化更快的输出大量数据然后读回的方法吗？可能没有，除非在特殊的情况下。例如，假设你决定将文本输出为64位的整数而不是一组8字节。作为文本的长整数的最大长度是大约20个字符，或者说二进制表示的2.5倍长。这种格式看起来不会快。然而，在某些情况下，例如位图，一个特殊的格式可能是一个改进。然而使用你自己的方案而不是串行化的标准方案将使你卷入一些权衡。

除了串行化实际的I/O和格式化开销外(使用DataInputStream和 DataOutputStream)，还有其他的开销，例如在串行化恢复时的创建新对象的需要。

注意DataOutputStream 方法也可以用于开发半自定义数据格式，例如:

```
 import java.io.*;

import java.util.*;


public class binary1 {

     public static void main(String args[]) {

          try {
```

```
                FileOutputStream fos = new FileOutputStream("outdata");
                BufferedOutputStream bos = new BufferedOutputStream(fos);
                DataOutputStream dos = new DataOutputStream(bos);
                Random rn = new Random();
                final int N = 10;
                dos.writeInt(N);
                for (int i = 1; i <= N; i++) {
                    int r = rn.nextInt();
                    System.out.println(r);
                    dos.writeInt(r);
                }
                dos.close();
            } catch (IOException e) {
                System.err.println(e);
            }
        }
}
```

和:

```
 import java.io.*;

public class binary2 {
    public static void main(String args[]) {
        try {
            FileInputStream fis = new FileInputStream("outdata");
            BufferedInputStream bis = new BufferedInputStream(fis);
            DataInputStream dis = new DataInputStream(bis);
            int N = dis.readInt();
            for (int i = 1; i <= N; i++) {
                int r = dis.readInt();
                System.out.println(r);
```

```
                }
                dis.close();
            } catch (IOException e) {
                System.err.println(e);
            }
        }
}
```

这些程序将10个整数写入文件然后读回它们。

1.1.3.2.10 获取文件信息

迄今为止我们的讨论围绕单一的文件输入输出。但是加速I/O性能还有另一方面——和得到文件特性有关。例如，考虑一个打印文件长度的小程序:

```
import java.io.*;

public class length1 {
    public static void main(String args[]) {
        if (args.length != 1) {
            System.err.println("missing filename");
            System.exit(1);
        }
        File f = new File(args[0]);
        long len = f.length();
        System.out.println(len);
    }
}
```

Java运行时系统自身并不知道文件的长度，因此必须向底层的操作系统查询以获得这个信息，对于文件的其他信息这也成立，例如文件是否是一个目录，文件上次修改时间等等。 java.io包中的File 类提供一套查询这些信息的方法。这些方法总体来说在时间上开销很大因此应该尽可能少用。

下面是一个查询文件信息的更长的范例，它递归整个文件系统写出所有的文件路径:

```
  import java.io.*;

public class roots {
```

```
    public static void visit(File f) {
        System.out.println(f);
    }

    public static void walk(File f) {
        visit(f);
        if (f.isDirectory()) {
            String list[] = f.list();
            for (int i = 0; i < list.length; i++)
                walk(new File(f, list[i]));
        }
    }

    public static void main(String args[]) {
        File list[] = File.listRoots();
        for (int i = 0; i < list.length; i++) {
            if (list[i].exists())
                walk(list[i]);
            else
                System.err.println("not accessible: " + list[i]);
        }
    }
}
```

这个范例使用 File 方法，例如 isDirectory 和 exists,穿越目录结构。每个文件都被查询一次它的类型（普通文件或者目录）。

1.1.4 与时间有关的类Date,DateFormat,Calendar

Date类用于表示日期和时间。它没考虑国际化问题，所以又设计了另外两个类。

Calendar类：

主要是进行日期字段之间的相互操作。

编程实例：计算出距当前日期时间315天后的日期时间，并使用”xxxx年xx月xx日xx小时:xx分:xx秒”的格式输出。

```
import java.util.*;
import java.text.SimpleDateFormat; //由于simpledateformat和 dateformat在这个包中
public class TestCalendar
{
public static  void main(String[] args)
{
Calendar cl=Calendar.getInstance(); //创建一个实例
System.out.println(cl.get(Calendar.YEAR)+"年"+cl.get(cl.MONTH)+"月"+cl.get(cl.DAY_OF_MONTH)+"日
"+cl.get(cl.HOUR)+":"+cl.get(cl.MINUTE)+":"+cl.get(cl.SECOND));
/*
使用get方法来取得日期中的年月日等等，参数为类中的常数，可以直接使用类名调用常数，也可以使用对象名。
*/
cl.add(cl.DAY_OF_MONTH,315);
//加上315天，使用add方法，第一个参数为单位，也是常数。
System.out.println(cl.get(Calendar.YEAR)+"年"+cl.get(cl.MONTH)+"月"+cl.get(cl.DAY_OF_MONTH)+"日
"+cl.get(cl.HOUR)+":"+cl.get(cl.MINUTE)+":"+cl.get(cl.SECOND));

SimpleDateFormat sdf1=new SimpleDateFormat("yyyy-MM-dd"); //定义了格式
SimpleDateFormat sdf2=new SimpleDateFormat("yyyy年MM月dd日"); //定义了格式
try
{
        Date d=sdf1.parse("2003-03-15"); //将字符串强制转换成这种格式，使用parse（）
        System.out.println(sdf2.format(d));将格式1的日期转换成格式2，使用format（）
}
catch(Exception e)
{
        e.printStackTrace();
}
}
}
```

编程实例：将“2002-03-15“格式的日期转换成“2003 年 03 月 15 日”的格式。代码在上例中的黑体部分。

1.2 深入理解嵌套类和内部类

1.2.1 什么是嵌套类及内部类？

可以在一个类的内部定义另一个类，这种类称为嵌套类（nested classes）,它有两种类型:静态嵌套类和非静态嵌套类。静态嵌套类使用很少，最重要的是非静态嵌套类，也即是被称作为内部类(inner)。嵌套类从JDK1.1 开始引入。其中 inner 类又可分为三种：

其一、在一个类（外部类）中直接定义的内部类；

其二、在一个方法（外部类的方法）中定义的内部类;

其三、匿名内部类。

下面，我将说明这几种嵌套类的使用及注意事项。

1.2.2 静态嵌套类

如下所示代码为定义一个静态嵌套类，

```
public class StaticTest {
private static String name = "javaJohn";
private String id = "X001";

static class Person{
private String address = "swjtu,chenDu,China";
public String mail = "josserchai@yahoo.com";//内部类公有成员
public void display(){
//System.out.println(id);//不能直接访问外部类的非静态成员
System.out.println(name);//只能直接访问外部类的静态成员
System.out.println("Inner "+address);//访问本内部类成员。
}
}

public void printInfo(){
Person person = new Person();
person.display();

//System.out.println(mail);//不可访问
//System.out.println(address);//不可访问

System.out.println(person.address);//可以访问内部类的私有成员
System.out.println(person.mail);//可以访问内部类的公有成员

}
public static void main(String[] args) {
StaticTest staticTest = new StaticTest();
staticTest.printInfo();
```

```
}
}
```

在静态嵌套类内部，不能访问外部类的非静态成员，这是由 Java 语法中"静态方法不能直接访问非静态成员"所限定。若想访问外部类的变量，必须通过其它方法解决，由于这个原因，静态嵌套类使用很少。注意，外部类访问内部类的的成员有些特别，不能直接访问，但可以通过内部类来访问，这是因为静态嵌套内的所有成员和方法默认为静态的了。同时注意，内部静态类 Person 只在类 StaticTest 范围内可见，若在其它类中引用或初始化，均是错误的。

1.2.3 在外部类中定义内部类

如下所示代码为在外部类中定义两个内部类及它们的调用关系：

```
public class Outer{
int outer_x = 100;

class Inner{
public int y = 10;
private int z = 9;
int m = 5;
public void display(){
System.out.println("display outer_x:"+ outer_x);
}
private void display2(){
System.out.println("display outer_x:"+ outer_x);
}

}

void test(){
Inner inner = new Inner();
inner.display();
inner.display2();
//System.out.println("Inner y:" + y);//不能访问内部内变量
System.out.println("Inner y:" + inner.y);//可以访问
System.out.println("Inner z" + inner.z);//可以访问
System.out.println("Inner m:" + inner.m);//可以访问

InnerTwo innerTwo= new InnerTwo();
innerTwo.show();
}

class InnerTwo{
Inner innerx = new Inner();
public void show(){
//System.out.println(y);//不可访问 Innter 的 y 成员
//System.out.println(Inner.y);//不可直接访问 Inner 的任何成员和方法
```

```
innerx.display();//可以访问
innerx.display2();//可以访问
System.out.println(innerx.y);//可以访问
System.out.println(innerx.z);//可以访问
System.out.println(innerx.m);//可以访问
}
}

public static void main(String args[]){
Outer outer = new Outer();
outer.test();
}
}
```

以上代码需要说明有，对于内部类，通常在定义类的class关键字前不加public或private等限制符，若加了没有任何影响，同时好像这些限定符对内部类的变量和方法也没有影响(?)。另外，就是要注意，内部类Inner及InnterTwo只在类Outer的作用域内是可知的，如果类Outer外的任何代码尝试初始化类Inner或使用它，编译就不会通过。同时，内部类的变量成员只在内部内内部可见，若外部类或同层次的内部类需要访问，需采用示例程序中的方法，不可直接访问内部类的变量。

1.2.4 在方法中定义内部类

如下所示代码为在方法内部定义一个内部类:

```
public class FunOuter {
int out_x = 100;

public void test(){
class Inner{
String x = "x";
void display(){
System.out.println(out_x);
}
}
Inner inner = new Inner();
inner.display();
}

public void showStr(String str){
//public String str1 = "test Inner";//不可定义，只允许final修饰
//static String str4 = "static Str";//不可定义，只允许final修饰
String str2 = "test Inner";
final String str3 = "final Str";
class InnerTwo{
public void testPrint(){
```

```
System.out.println(out_x);//可直接访问外部类的变量
//System.out.println(str);//不可访问本方法内部的非 final变量
//System.out.println(str2);//不可访问本方法内部的非 final变量
System.out.println(str3);//只可访问本方法的 final型变量成员
}
}
InnerTwo innerTwo= new InnerTwo();
innerTwo.testPrint();
}

public void use(){
//Inner innerObj = new Inner();//此时 Inner 已不可见了。
//System.out.println(Inner.x);//此时 Inner 已不可见了。
}


public static  void main(String[] args) {
FunOuter outer = new FunOuter();
outer.test();
}
}
```

从上面的例程我们可以看出定义在方法内部的内部类的可见性更小，它只在方法内部可见，在外部类(及外部类的其它方法中)中都不可见了。同时，它有一个特点，就是方法内的内部类连本方法的成员变量都不可访问，它只能访问本方法的 final型成员。同时另一个需引起注意的是方法内部定义成员，只允许final修饰或不加修饰符，其它像 static 等均不可用。

1.2.5 匿名内部类

如下所示代码为定义一个匿名内部类:匿名内部类通常用在 Java 的事件处理上

```
import java.applet.*;
import java.awt.event.*;

  public class AnonymousInnerClassDemo extends Applet{
  public void init(){
    addMouseListener(new  MouseAdapter(){
      public void mousePressed(MouseEvent me){
       showStatus("Mouse Pressed!");
    }
    })
  }
  public void showStatus(String str){
    System.out.println(str);
```

}
}

在 上面的例子中，方法 addMouseListener 接受一个对象型的参数表达式，于是，在参数里，我们定义了一个匿名内部类这个类是一个 MouseAdapter 类型的类，同时在这个类中定义了一个继承的方法 mousePressed，整个类做为一个参数。这个类没有名称，但是当执行这个 表达式时它被自动实例化。同时因为，这个匿名内部类是定义在 AnonymousInnerClassDemo 类内部的，所以它可以访问它的方法 showStatus。这同前面的内部类是一致的。

1.2.6 内部类使用的其它的问题

通过以上，我们可以清楚地看出内部类的一些使用方 法，同时，在许多时候，内部类是在如 Java 的事件处理、或做为值对象来使用的。同时，我们需注意最后一个问题，那就是，内部类同其它类一样被定义，同样 它也可以继承外部其它包 的类和实现外部其它地方的接口。同样它也可以继承同一层次的其它的内部类,甚至可以继承外部类本身。下面我们给出最后一个例子做为结束：

```
public class Layer {
//Layer 类的成员变量
private String testStr = "testStr";

//Person 类，基类
class Person{
String name;
Email email;
public void setName(String nameStr){
this.name = nameStr;
}
public String getName(){
return this.name;
}
public void setEmail(Email emailObj){

this.email = emailObj;
}
public String getEmail(){
return this.email.getMailStr();
}
//内部类的内部类，多层内部类
class Email{
String mailID;
String mailNetAddress;
Email(String mailId,String mailNetAddress){
this.mailID = mailId;
```

```
this.mailNetAddress = mailNetAddress;
}
String getMailStr(){
return this.mailID +"@"+this.mailNetAddress;
}
}
}
//另一个内部类继承外部类本身
class ChildLayer extends Layer{
void print(){
System.out.println(super.testStr);//访问父类的成员变量
}
}
//另个内部类继承内部类Person
class OfficePerson extends Person{
void show(){
System.out.println(name);
System.out.println(getEmail());
}
}
//外部类的测试方法
public void testFunction(){
//测试第一个内部类
ChildLayer childLayer = new ChildLayer();
childLayer.print();

//测试第二个内部类
OfficePerson officePerson = new OfficePerson();
officePerson.setName("abner chai");
//注意此处，必须用 对象.new 出来对象的子类对象
//而不是Person.new Email(...)
//也不是new Person.Email(...)
officePerson.setEmail(officePerson.new Email("josserchai","yahoo.com"));

officePerson.show();
}
public static void main(String[] args) {
Layer layer = new Layer();
layer.testFunction();
}
}
```

1.3 文件和流

Java I/O系统的类实在是太多了，这里我们只学习一些基本的和常用的，相信能够掌握这些就可以解决我们以后的普通应用

了

1.3.1 什么是数据流

数据流是指所有的数据通信通道

有两类流，InputStream and OutputStream，Java 中每一种流的基本功能依赖于它们

InputStream 用于 read,OutputStream 用于 write, 读和写都是相对与内存说的，读就是从其他地方把数据拿进内存，写就是把数据从内存推出去

这两个都是抽象类，不能直接使用

1.3.2 InputStream 的方法有：

read() 从流中读入数据 有 3 中方式：

int read() 一次读一个字节

int read(byte[]) 读多个字节到数组中

int read(byte[],int off,int len) 指定从数组的哪里开始，读多长

skip() 跳过流中若干字节

available() 返回流中可用字节数，但基于网络时无效，返回 0

markSupported() 判断是否支持标记与复位操作

mark() 在流中标记一个位置，要与 markSupported()连用

reset() 返回标记过的位置

close() 关闭流

1.3.3 OutputStream 的方法：

write(int) 写一个字节到流中

write(byte[]) 将数组中的内容写到流中

write(byte[],int off,int len) 将数组中从 off 指定的位置开始 len 长度的数据写到流中

close() 关闭流

flush() 将缓冲区中的数据强制输出

1.3.4 File 类

File 可以表示文件也可以表示目录，File 类控制所有硬盘操作

构造器：

File(File parent,String child) 用父类和文件名构造

File(String pathname) 用绝对路径构造

File(String parent,String child) 用父目录和文件名构造

File(URI uri) 用远程文件构造

常用方法：

boolean createNewFile();

boolean exists();

例子：

//建立 test.txt 文件对象，判断是否存在，不存在就创建

```
import java.io.*;
```

```
public class CreateNewFile{
public static void main(String args[]){
File f=new File("test.txt");
try{
if(!f.exists())
f.createNewFile();
else
System.out.println("exists");
}catch(Exception e){
e.printStackTrace();
}
}
}
```

boolean mkdir()/mkdirs()

boolean renameTo(File destination)

例子：//看一下这 mkdir()/mkdirs() 的区别和 renameTo 的用法

```
import java.io.*;
public class CreateDir{
public static void main(String args[]){
File f=new File("test.txt");
File f1=new File("Dir");
File f2=new File("Top/Bottom");
File f3=new File("newTest.txt");
try{
f.renameTo(f3);
f1.mkdir();
f2.mkdirs();
}catch(Exception e){
e.printStackTrace();
}
}
}
```

String getPath()/getAbsolutePath()

String getParent()/getName()

例子：//硬盘上并没有 parent 目录和 test.txt 文件，但我们仍然可以操作，因为我们创建了他们的对象，是对对象进行操作

```
import java.io.*;
public class Test{
public static void main(String args[]){
File f=new File("parent/test.txt");
File f1=new File("newTest.txt");
try{
System.out.println(f.getParent());
```

```
System.out.println(f.getName());
System.out.println(f1.getPath());
System.out.println(f1.getAbsolutePath());
}catch(Exception e){
e.printStackTrace();
}
}
}
```

String list[] //显示目录下所有文件

long lastModified() //返回 1970.1.1 到最后修改时间的秒数

boolean isDirectory()

例子：//列出目录下的所有文件和目录，最后修改时间，是目录的后面标出<DIR>,是文件的后面标出文件长度

```
import java.io.*;
import java.util.*;
public class Dir{
public static void main(String args[]){
File f=new File("Dir");
String[] listAll=null;
File temp=null;
try{
listAll=f.list();
for(int i=0;i<listAll.length;i++){
temp=new File(listAll<i>);
System.out.print(listAll<i>+"\t");
if(temp.isDirectory())
System.out.print("\t<DIR>\t");
else
System.out.print(temp.length()+"\t");
System.out.println(new Date(temp.lastModified()));
}
}catch(Exception e){
e.printStackTrace();
}
}
}
```

1.3.5 文件流的建立

```
File f=new File("temp.txt");
FileInputStream in=new FileInputStream(f);
FileOutputStream out=new FileOutputStream(f);
```

例子：文件拷贝

```
import java.io.*;
public class Copy{
public static void main(String args[]){
FileInputStream fis=null;
FileOutputStream fos=null;
try{
fis=new FileInputStream("c2.gif");
fos=new FileOutputStream("c2_copy.gif");
int c;
while((c=fis.read()) != -1)
fos.write(c);
}catch(Exception e){
e.printStackTrace();
}finally{
if(fis != null)try{ fis.close(); }catch(Exception e){ e.printStackTrace(); }
if(fos!= null) try{ fos.close(); }catch(Exception e){ e.printStackTrace(); }
}
}
}
```

1.3.6 缓冲区流

BufferedInputStream

BufferedOutputStream

他们是在普通文件流上加了缓冲的功能，所以构造他们时要先构造普通流

例子：文件拷贝的缓冲改进

```
import java.io.*;
public class Copy{
public static void main(String args[]){
BufferedInputStream bis=null;
BufferedOutputStream bos=null;
byte buf[]=new byte[100];
try{
bis=new BufferedInputStream(new FileInputStream("persia.mp3"));
bos=new BufferedOutputStream(new FileOutputStream("persia_copy.mp3"));
int len=0;
while( true ){
len=bis.read(buf);
if(len<=0) break;
bos.write(buf,0,len);
}
bos.flush();//缓冲区只有满时才会将数据输出到输出流，用flush()将未满的缓冲区中数据强制输出
}catch(Exception e){
e.printStackTrace();
```

```
}finally{
if(bis != null)try{ bis.close(); }catch(Exception e){ e.printStackTrace(); }
if(bos!= null) try{ bos.close(); }catch(Exception e){ e.printStackTrace(); }
}
}
}
```

1.3.7 原始型数据流

DataInputStream

DataOutputStream

他们是在普通流上加了读写原始型数据的功能，所以构造他们时要先构造普通流

方法：

readBoolean()/writeBoolean()

readByte()/writeByte()

readChar()/writeByte()

.....

例子：//这个流比较简单，要注意的就是读时的顺序要和写时的一样

```
import java.io.*;
public class DataOut{
public static void main(String args[]){
DataOutputStream dos=null;
try{
dos=new DataOutputStream(new FileOutputStream("dataout.txt"));
dos.writeInt(1);
dos.writeBoolean(true);
dos.writeLong(100L);
dos.writeChar('a');
}catch(Exception e){
e.printStackTrace();
}finally{
if(dos!=null)
try{
dos.close();
}catch(Exception e){
}
}
}
}
```

```
import java.io.*;
public class DataIn{
public static void main(String args[]){
DataInputStream dis=null;
```

```
try{
dis=new DataInputStream(new FileInputStream("dataout.txt"));
System.out.println(dis.readInt());
System.out.println(dis.readBoolean());
System.out.println(dis.readLong());
System.out.println(dis.readChar());
}catch(Exception e){
e.printStackTrace();
}finally{
if(dis!=null)
try{
dis.close();
}catch(Exception e){
}
}
}
}
```

1.3.8 对象流

串行化：对象通过写出描述自己状态的数值来记述自己的过程叫串行话

对象流：能够输入输出对象的流

将串行化的对象通过对象流写入文件或传送到其他地方

对象流是在普通流上加了传输对象的功能，所以构造对象流时要先构造普通文件流

注意：只有实现了 Serializable 接口的类才能被串行化

例子：

```
import java.io.*;
class Studentimplements Serializable{
private String name;
private int age;

public Student(String name,int age){
this.name=name;
this.age=age;
}

public void greeting(){
System.out.println("hello ,my name is "+name);
}

public String toString(){
return "Student["+name+","+age+"]";
}
}
public class ObjectOutTest{
```

```
public static void main(String args[]){
ObjectOutputStream oos=null;
try{
oos=new ObjectOutputStream(
new FileOutputStream("student.txt"));
Student s1=new Student("Jerry",24);
Student s2=new Student("Andy",33);

oos.writeObject(s1);
oos.writeObject(s2);
}catch(Exception e){
e.printStackTrace();
}finally{
if(oos!=null)
try{
oos.close();
}catch(Exception e){
e.printStackTrace();
}
}
}
}

import java.io.*;
public class ObjectInTest{
public static void main(String args[]){
ObjectInputStream ois=null;
Student s=null;
try{
ois=new ObjectInputStream(
new FileInputStream("student.txt"));
System.out.println("----------");
s=(Student)ois.readObject();
System.out.println(s);
s.greeting();
System.out.println("----------");
s=(Student)ois.readObject();
System.out.println(s);
s.greeting();
}catch(Exception e){
e.printStackTrace();
}finally{
if(ois!=null)
try{
```

```
ois.close();
}catch(Exception e){
e.printStackTrace();
}
}
}
}
```

1.3.9 字符流 InputStreamReader/OutputStreamWriter

上面的几种流的单位是 byte，所以叫做字节流，写入文件的都是二进制字节，我们无法直接看，下面要学习的是字节流

Java 采用 Unicode 字符集，每个字符和汉字都采用 2 个字节进行编码，ASCII 码是 Unicode 编码的自集

InputStreamReader 是 字节流 到 字符桥的桥梁 （ byte->char 读取字节然后用特定字符集编码成字符)

OutputStreamWriter 是 字符流 到 字节流的桥梁 （ char->byte)

他们是在字节流的基础上加了桥梁作用，所以构造他们时要先构造普通文件流

我们常用的是:

BufferedReader 方法：readLine()

PrintWriter 方法： println()

例子：

```
import java.io.*;
public class PrintWriterTest{
public static void main(String args[]){
PrintWriter pw=null;
try{
pw=new PrintWriter(
new OutputStreamWriter(
new FileOutputStream("bufferedwriter.txt")));
pw.println("hello world");
}catch(Exception e){
e.printStackTrace();
}finally{
if(pw!=null)
try{
pw.close();
}catch(Exception e){
e.printStackTrace();
}
}
}
}
```

```
import java.io.*;
public class BufferedReaderTest{
public static void main(String args[]){
```

```
BufferedReader br=null;
try{
br=new BufferedReader(
new InputStreamReader(
new FileInputStream("bufferedwriter.txt")));
System.out.println(br.readLine());
}catch(Exception e){
e.printStackTrace();
}finally{
if(br!=null)
try{
br.close();
}catch(Exception e){
e.printStackTrace();
}
}
}
}
```

1.3.10 随机存取文件 RandomAccessFile

可同时完成读写操作

支持随机文件操作的方法:

readXXX()/writeXXX()

seek() 将指针调到所需位置

getFilePointer() 返回指针当前位置

length() 返回文件长度

例子：把若干个 32 位的整数写到一个名为 “temp.txt”的文件中,然后利用 seek 方法,以相反的顺序再读取这些数据

```
import java.io.*;
public class RandomFile{
public static void main(String args[]){
RandomAccessFile raf=null;
int data[]={12,31,56,23,27,1,43,65,4,99};
try{
raf=new RandomAccessFile("temp.txt","rw");
for(int i=0;i<data.length;i++)
raf.writeInt(data<i>);
for(int i=data.length-1;i>=0;i--){
raf.seek(i*4);
System.out.println(raf.readInt());
}
}catch(Exception e){
e.getMessage();
}finally{
```

```
if(raf!=null)
try{
raf.close();
}catch(Exception e){
e.getMessage();
}
}
}
}
```

1.3.11 小结

这部分的难点就是类比较复杂，尤其是每个类的构造方式，我认为记住下面这个图比记类的继承关系更好些

字节流：

InputStream
|-- FileInputStream (基本文件流）
|-- BufferedInputStream
|-- DataInputStream
|-- ObjectInputStream
OutputStream 同上图
BufferedInputStream DataInputStream ObjectInputStream 只是在 FileInputStream 上增添了相应的功能，构造时先构造 FileInputStream

字符流：
Reader
|-- InputStreamReader (byte->char 桥梁）
|-- BufferedReader (常用）

Writer
|-- OutputStreamWriter (char->byte 桥梁）
|-- BufferedWriter
|-- PrintWriter （常用）

随机存取文件 RandomAccessFile

1.4 java 中的一些常用词汇

Abstract class 抽象类:抽象类是不允许实例化的类，因此一般它需要被进行扩展继承。

Abstract method 抽象方法:抽象方法即不包含任何功能代码的方法。

Access modifier 访问控制修饰符:访问控制修饰符用来修饰 Java 中类、以及类的方法和变量的访问控制属性。

Anonymous class 匿名类:当你需要创建和使用一个类，而又不需要给出它的名字或者再次使用的使用，就可以利用匿名类。

Anonymous inner classes 匿名内部类:匿名内部类是没有类名的局部内部类。

API 应用程序接口:提供特定功能的一组相关的类和方法的集合。

Array 数组:存储一个或者多个相同数据类型的数据结构，使用下标来访问。在Java中作为对象处理。

Automatic variables 自动变量:也称为方法局部变量method local variables，即声明在方法体中的变量。

Base class 基类:即被扩展继承的类。

Blocked state 阻塞状态:当一个线程等待资源的时候即处于阻塞状态。阻塞状态不使用处理器资源

Call stack 调用堆栈:调用堆栈是一个方法列表，按调用顺序保存所有在运行期被调用的方法。

Casting 类型转换 :即一个类型到另一个类型的转换，可以是基本数据类型的转换，也可以是对象类型的转换。

char 字符:容纳单字符的一种基本数据类型。

Child class 子类:见继承类Derived class

Class 类:面向对象中的最基本、最重要的定义类型。

Class members 类成员:定义在类一级的变量，包括实例变量和静态变量。

Class methods 类方法:类方法通常是指的静态方法，即不需要实例化类就可以直接访问使用的方法。

Class variable 类变量:见静态变量Static variable

Collection 容器类:容器类可以看作是一种可以储存其他对象的对象，常见的容器类有Hashtables和Vectors。

Collection interface 容器类接口:容器类接口定义了一个对所有容器类的公共接口。

Collections framework 容器类构架:接口、实现和算法三个元素构成了容器类的架构。

Constructor 构造函数:在对象创建或者实例化时候被调用的方法。通常使用该方法来初始化数据成员和所需资源。

Containers容器:容器是一种特殊的组件，它可以容纳其他组件。

Declaration 声明:声明即是在源文件中描述类、接口、方法、包或者变量的语法。

Derived class 继承类:继承类是扩展继承某个类的类。

Encapsulation 封装性:封装性体现了面向对象程序设计的一个特性，将方法和数据组织在一起，隐藏其具体实现而对外体现出公共的接口。

Event classes 事件类:所有的事件类都定义在java.awt.event包中。

Event sources 事件源:产生事件的组件或对象称为事件源。事件源产生事件并把它传递给事件监听器event listener*。

Exception 异常:异常在Java中有两方面的意思。首先，异常是一种对象类型。其次，异常还指的是应用中发生的一种非标准流程情况，即异常状态。

Extensibility扩展性:扩展性指的是面向对象程序中，不需要重写代码和重新设计，能容易的增强源设计的功能。

Finalizer 收尾:每个类都有一个特殊的方法finalizer，它不能被直接调用，而被JVM在适当的时候调用，通常用来处理一些清理资源的工作，因此称为收尾机制。

Garbage collection 垃圾回收机制:当需要分配的内存空间不再使用的时候，JVM将调用垃圾回收机制来回收内存空间。

Guarded region 监控区域:一段用来监控错误产生的代码。

Heap堆:Java中管理内存的结构称作堆。

Identifiers 标识符:即指定类、方法、变量的名字。注意Java是大小写敏感的语言。

Import statement 引入语法:引入语法允许你可以不使用某个类的全名就可以参考这个类。

Inheritance 继承:继承是面向对象程序设计的重要特点，它是一种处理方法，通过这一方法，一个对象可以获得另一个对象的特征。

Inner classes 内部类:内部类与一般的类相似，只是它被声明在类的内部，或者甚至某个类方法体中。

Instance 实例:类实例化以后成为一个对象。

Instance variable 实例变量:实例变量定义在对象一级，它可以被类中的任何方法或者其他类的中方法访问，但是不能被静态方法访问。

Interface 接口:接口定义的是一组方法或者一个公共接口，它必须通过类来实现。

Java source file Java源文件:Java源程序包含的是Java程序语言计算机指令。

Java Virtual Machine (JVM) Java虚拟机:解释和执行Java字节码的程序，其中Java字节码由Java编译器生成。

javac Java编译器:Javac是Java编译程序的名称。

JVM Java虚拟机:见Java虚拟机

Keywords 关键字:即Java中的保留字，不能用作其他的标识符。

Layout managers 布局管理器:布局管理器是一些用来负责处理容器中的组件布局排列的类。

Local inner classes 局部内部类:在方法体中，或者甚至更小的语句块中定义的内部类。

Local variable 局部变量:在方法体中声明的变量

Member inner classes 成员内部类:定义在封装类中的没有指定static修饰符的内部类。

Members 成员:类中的元素，包括方法和变量。

Method 方法:完成特定功能的一段源代码，可以传递参数和返回结果，定义在类中。

Method local variables 方法局部变量:见自动变量Automatic variables

Modifier 修饰符:用来修饰类、方法或者变量行为的关键字。

Native methods 本地方法:本地方法是指使用依赖平台的语言编写的方法，它用来完成Java无法处理的某些依赖于平台的功能。

Object 对象:一旦类实例化之后就成为对象。

Overloaded methods 名称重载方法:方法的名称重载是指同一个类中具有多个方法，使用相同的名称而只是其参数列表不同。

Overridden methods 覆盖重载方法:方法的覆盖重载是指父类和子类使用的方法采用同样的名称、参数列表和返回类型。

Package 包:包即是将一些类聚集在一起的一个实体。

Parent class 父类:被其他类继承的类。也见基类。

Private members 私有成员:私有成员只能在当前类被访问，其他任何类都不可以访问之。

Public members 公共成员:公共成员可以被任何类访问，而不管该类属于那个包。

Runtime exceptions 运行时间异常:运行时间异常是一种不能被你自己的程序处理的异常。通常用来指示程序BUG。

Source file 源文件:源文件是包含你的Java代码的一个纯文本文件。

Stack trace 堆栈轨迹:如果你需要打印出某个时间的调用堆栈状态，你将产生一个堆栈轨迹。

Static inner classes 静态内部类:静态内部类是内部类最简单的形式，它于一般的类很相似，除了被定义在了某个类的内部。

Static methods 静态方法:静态方法声明一个方法属于整个类，即它可以不需要实例化一个类就可以通过类直接访问之。

Static variable 静态变量:也可以称作类变量。它类似于静态方法，也是可以不需要实例化类就可以通过类直接访问。

Superclass 超类:被一个或多个类继承的类。

Synchronized methods 同步方法:同步方法是指明某个方法在某个时刻只能由一个线程访问。

Thread 线程:线程是一个程序内部的顺序控制流。

Time-slicing 时间片:调度安排线程执行的一种方案。

Variable access 变量访问控制:变量访问控制是指某个类读或者改变一个其他类中的变量的能力。

Visibility 可见性: 可见性体现了方法和实例变量对其他类和包的访问控制。

1.5 J2SE 学习中的 30 个基本概念

前言: 在我们学习 Java 的过程中,掌握其中的基本概念对我们的学习无论是J2SE,J2EE, J2ME 都是很重要的,J2SE 是 Java 的基础,所以有必要对其中的基本概念做以归纳,以便大家在以后的学习过程中更好的理解java 的精髓,在此我总 结了 30 条基本的概念.

Java 概述:

目前 Java 主要应用于中间件的开发(middleware)—处理客户机于服务器之间的 通信技术,早期的实践证明,Java 不适合 pc 应用程序的开发,其发展逐渐变成在开发手持设备,互联网信息站,及车载计算机的开发.Java 于其他语言所 不同的是程序运行时提供了平台的独立性,称许可以在windows,solaris,linux 其他操作系统上使用完全相同的代码.Java 的语法与 C++语法类似,C++/C 程序员很容易掌握,而且Java 是完全的彻底的面向对象的,其中提出了很好的GC(Garbage Collector)垃圾处理 机制,防止内存溢出.

Java 的白皮书为我们提出了 Java 语言的 11 个关键特性.

(1)Easy:Java 的语法比 C++的相对简单,另一个方面就是 Java 能使软件在很小的机器上运行,基础解释其和类库的支持的大小约为 40kb,增加基本的标准库和线程支持的内存需要增加 125kb.

(2)分布式:Java 带有很强大的 TCP/IP 协议族的例程库,Java 应用程序能够通过 URL 来穿过网络来访问远程对象,由于 servlet 机制的出现,使 Java 编程非常的高效,现在许多的大的 web server 都支持 servlet.

(3)OO:面向对象设计是把重点放在对象及对象的接口上的一个编程技术.其面向对象和 C++有很多不同,在与多重继承的处理及 Java 的原类模型.

(4)健壮特性:Java 采取了一个安全指针模型,能减小重写内存和数据崩溃的可能型

(5)安全:Java 用来设计网路和分布系统,这带来了新的安全问题,Java 可以用来构建防病毒和防攻击的System.事实证明 Java 在防

毒这一方面做的比较好.

(6)中立体系结构:Java 编译其生成体系结构中立的目标文件格式可以在很多处理器上执行,编译器产生的指令字节码(Javabytecode)实现此特性,此字节码可以在任何机器上解释执行.

(7)可移植性:Java 中对基本数据结构类型的大小和算法都有严格的规定所以可移植性很好.

(8)多线程:Java 处理多线程的过程很简单,Java 把多线程实现交给底下操作系统或线程程序完成.所以多线程是 Java 作为服务器端开发语言的流行原因之一

(9)Applet 和 servlet: 能够在网页上执行的程序叫 Applet,需要支持 Java 的浏览器很多,而 applet 支持动态的网页,这是很多其他语言所不能做到的.

基本概念:

1.OOP 中唯一关系的是对象的接口是什么,就像计算机的销售商她不管电源内部结构是怎样的,他只关系能否给你提供电就行了,也就是只要知道can or not而 不是how and why.所有的程序是由一定的属性和行为对象组成的,不同的对象的访问通过函数调用来完成,对象间所有的交流都是通过方法调用,通过 对封装对象数据,很大限度上提高复用率.

2.OOP 中最重要的思想是类,类是模板是蓝图,从类中构造一个对象,即创建了这个类的一个实例(instance)

3.封装:就是把数据和行为结合起在一个包中)并对对象使用者隐藏数据的实现过程,一个对象中的数据叫他的实例字段(instance field)

4.通过扩展一个类来获得一个新类叫继承(inheritance),而所有的类都是由Object根超类扩展而得,根超类下文会做介绍.

5.对象的3个主要特性

behavior—说明这个对象能做什么.

state—当对象施加方法时对象的反映.

identity—与其他相似行为对象的区分标志.

每个对象有唯一的 indentity 而这3者之间相互影响.

6.类之间的关系:

use-a :依赖关系

has-a :聚合关系

is-a :继承关系--例:A 类继承了B 类,此时A类不仅有了B类的方法,还有其自己的方法.(个性存在于共性中)

7.构造对象使用构造器:构造器的提出,构造器是一种特殊的方法,构造对象并对其初始化.

例 ata 类的构造器叫 Data

new Data()—构造一个新对象,且初始化当前时间.

Data happyday=new Data()—把一个对象赋值给一个变量happyday,从而使该对象能够多次使用,此处要声明的使变量与对象变量二者是不同的.new 返回的值是一个引用.

构造器特点:构造器可以有0个,一个或多个参数

构造器和类有相同的名字

一个类可以有多个构造器

构造器没有返回值

构造器总是和new运算符一起使用.

8.重载:当多个方法具有相同的名字而含有不同的参数时,便发生重载.编译器必须挑选出调用哪个方法.

9.包(package)Java允许把一个或多个类收集在一起成为一组,称作包,以便于组织任务,标准Java库分为许多包.java.lang java.util java,net等,包是分层次的所有的java包都在java和javax包层次内.

10.继承思想:允许在已经存在的类的基础上构建新的类,当你继承一个已经存在的类时,那么你就复用了这个类的方法和字段,同时你可以在新类中添加新的方法和字段.

11.扩展类:扩展类充分体现了is-a的继承关系. 形式为:class (子类) extends (基类).

12.多态:在java中,对象变量是多态的.而java中不支持多重继承.

13.动态绑定:调用对象方法的机制.

(1)编译器检查对象声明的类型和方法名.

(2)编译器检查方法调用的参数类型.

(3)静态绑定:若方法类型为 priavte static final编译器会准确知道该调用哪个方法.

(4)当程序运行并且使用动态绑定来调用一个方法时,那么虚拟机必须调用 x 所指向的对象的实际类型相匹配的方法版本.

(5)动态绑定:是很重要的特性,它能使程序变得可扩展而不需要重编译已存代码.

14.final类:为防止他人从你的类上派生新类,此类是不可扩展的.

15.动态调用比静态调用花费的时间要长.

16.抽象类:规定一个或多个抽象方法的类本身必须定义为 abstract

例: public abstract string getDescripition

17.Java 中的每一个类都是从 Object类扩展而来的.

18.object 类中的 equal 和 toString 方法.

equal 用于测试一个对象是否同另一个对象相等.

toString 返回一个代表该对象的字符串,几乎每一个类都会重载该方法,以便返回当前状态的正确表示.

(toString 方法是一个很重要的方法)

19.通用编程:任何类类型的所有值都可以同 object 类性的变量来代替.

20.数组列表:ArrayList动态数组列表,是一个类库,定义在java.uitl 包中,可自动调节数组的大小.

21.class类 object 类中的getclass 方法返回ckass 类型的一个实例,程序启动时包含在 main 方法的类会被加载,虚拟机要加载他需要的所有类,每一个加载的类都要加载它需要的类.

22.class类为编写可动态操纵java 代码的程序提供了强大的功能反射,这项功能为JavaBeans特别有用,使用反射 Java 能支持VB程序员习惯使用的工具.

能够分析类能力的程序叫反射器,Java 中提供此功能的包叫Java.lang.reflect 反射机制十分强大.

1.在运行时分析类的能力.

2.在运行时探察类的对象.

3.实现通用数组操纵代码.

4.提供方法对象.

而此机制主要针对是工具者而不是应用及程序.

反射机制中的最重要的部分是允许你检查类的结构,用到的 API 有:

java.lang.reflect.Field 返回字段.

java.reflect.Method 返回方法.

java.lang.reflect.Constructor 返回参数.

方法指针:java 没有方法指针,把一个方法的地址传给另一个方法,可以在后面调用它,而接口是更好的解决方案.

23.接口(Interface)说明类该做什么而不指定如何去做,一个类可以实现一个或多个 interface.

24.接口不是一个类,而是对符合接口要求的类的一套规范.

若实现一个接口需要 2 个步骤:

1.声明类需要实现的指定接口.

2.提供接口中的所有方法的定义.

声明一个类实现一个接口需要使用implements 关键字

class actionB implements Comparable 其 actionb 需要提供 CompareTo 方法,接口不是类,不能用 new 实例化一个接口.

25.一个类只有一个超类,但一个类能实现多个接口.Java 中的一个重要接口

Cloneable

26.接口和回调.编程一个常用的模式是回调模式,在这种模式中你可以指定当一个特定时间发生时回调对象上的方法.

例:ActionListener 接口监听.

类似的 API 有:java.swing.JOptionPane

java.swing.Timer

java.awt.Tookit

27.对象clone:clone 方法是 object 一个保护方法,这意味着你的代码不能简单的调用它.

28.内部类:一个内部类的定义是定义在另一个内部的类

原因是:1.一个内部类的对象能够访问创建它的对象的实现,包括私有数据

2.对于同一个包中的其他类来说,内部类能够隐藏起来.

3.匿名内部类可以很方便的定义回调

4.使用内部类可以非常方便的编写事件驱动程序.

29.代理类(proxy):1.指定接口要求所有代码

2.object 类定义的所有的方法(toString equals)

30.数据类型:Java 是强调类型的语言,每个变量都必须先申明它都类型,java 中总共有8个基本类型.4 种是整型,2种是浮点型,一种是字符型,被用于

Unicode编码中的字符,布尔型.

1.6 Java 线程

在论坛上面常常看到初学者对线程的无可奈何，所以总结出了下面一篇文章，希望对一些正在学习使用java线程的初学者有所帮助。

首先要理解线程首先需要了解一些基本的东西，我们现在所使用的大多数操作系统都属于多任务，分时操作系统。正是由于这种操作系统的出现才有了多线程这个概念。我们使用的windows,linux就属于此列。什么是分时操作系统呢，通俗一点与就是可以同一时间执行多个程序的操作系统，在自己的电脑上面，你是不是一边听歌，一边聊天还一边看网页呢？但实际上，并不上 cpu在同时执行这些程序，cpu只是将时间切割为时间片，然后将时间片分配给这些程序，获得时间片的程序开始执行，不等执行完毕，下个程序又获得时间片开始执行，这样多个程序轮流执行一段时间，由于现在cpu的高速计算能力，给人的感觉就像是多个程序在同时执行一样。

一般可以在同一时间内执行多个程序的操作系统都有进程的概念.一个进程就是一个执行中的程序,而每一个进程都有自己独立的一块内存空间,一组系统资源.在进程概念中,每一个进程的内部数据和状态都是完全独立的.因此可以想像创建并执行一个进程的系统开像是比较大的，所以线程出现了。在java中，程序通过流控制来执行程序流,程序中单个顺序的流控制称为线程,多线程则指的是在单个程序中可以同时运行多个不同的线程,执行不同的任务.多线程意味着一个程序的多行语句可以看上去几乎在同一时间内同时运行.（你可以将前面一句话的程序换成进程，进程是程序的一次执行过程,是系统运行程序的基本单位）

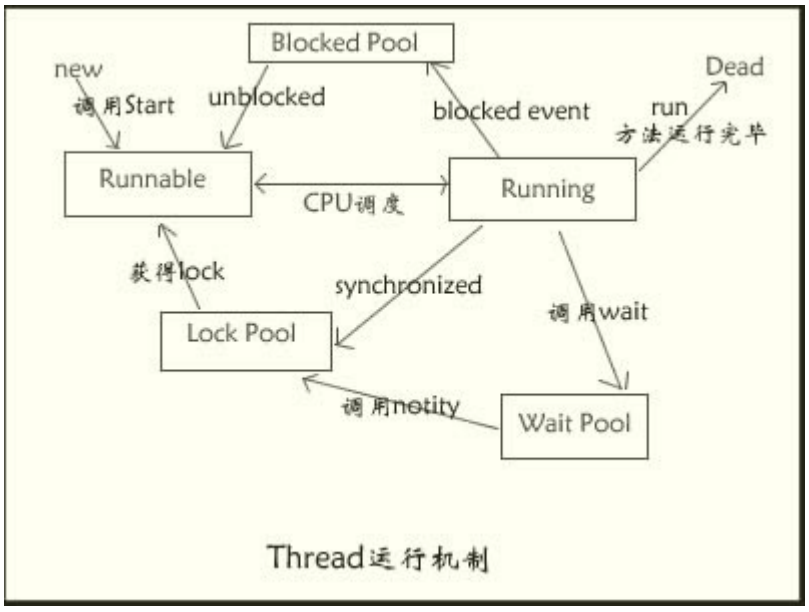
线程与进程相似,是一段完成某个特定功能的代码,是程序中单个顺序的流控制;但与进程不同的是,同类的多个线程是共享一块内存空间和一组系统资源,而线程本身的数据通常只有微处理器的寄存器数据,以及一个供程序执行时使用的堆栈.所以系统在产生一个线程,或者在各个线程之间切换时,负担要比进程小的多,正因如此,线程也被称为轻负荷进程(light-weight process).一个进程中可以包含多个线程.

多任务是指在一个系统中可以同时运行多个程序,即有多个独立运行的任务,每个任务对应一个进程，同进程一样,一个线程也有从创建,运行到消亡的过程,称为线程的生命周期.用线程的状态(state)表明线程处在生命周期的哪个阶段.线程有创建,可运行,运行中,阻塞,死亡五中状态.通过线程的控制与调度可使线程在这几种状态间转化每个程序至少自动拥有一个线程,称为主线程.当程序加载到内存时,启动主线程.

[线程的运行机制以及调度模型]

java中多线程就是一个类或一个程序执行或管理多个线程执行任务的能力，每个线程可以独立于其他线程而独立运行，当然也可以和其他线程协同运行，一个类控制着它的所有线程，可以决定哪个线程得到优先级，哪个线程可以访问其他类的资源，哪个线程开始执行，哪个保持休眠状态。

下面是线程的机制图：



线程的状态表示线程正在进行的活动以及在此时间段内所能完成的任务.线程有创建,可运行,运行中,阻塞,死亡五中状态.一个具有生命的线程,总是处于这五种状态之一：

1.创建状态

使用new运算符创建一个线程后,该线程仅仅是一个空对象,系统没有分配资源,称该线程处于创建状态(new thread)

2.可运行状态

使用start()方法启动一个线程后,系统为该线程分配了除CPU外的所需资源,使该线程处于可运行状态(Runnable)

3.运行中状态

Java运行系统通过调度选中一个Runnable的线程,使其占有CPU并转为运行中状态(Running).此时,系统真正执行线程的run()方法.

4.阻塞状态

一个正在运行的线程因某种原因不能继续运行时,进入阻塞状态(Blocked)

5.死亡状态

线程结束后是死亡状态(Dead)

同一时刻如果有多个线程处于可运行状态,则他们需要排队等待CPU资源.此时每个线程自动获得一个线程的优先级(priority),优先级的高低反映线程的重要或紧急程度.可运行状态的线程按优先级排队,线程调度依据优先级基础上的"先到先服务"原则.

线程调度管理器负责线程排队和CPU在线程间的分配,并由线程调度算法进行调度.当线程调度管理器选种某个线程时,该线程获得CPU资源而进入运行状态.

线程调度是先占式调度,即如果在当前线程执行过程中一个更高优先级的线程进入可运行状态,则这个线程立即被调度执行.先占式调度分为:独占式和分时方式.

独占方式下,当前执行线程将一直执行下去,直 到执行完毕或由于某种原因主动放弃CPU,或CPU被一个更高优先级的线程抢占

分时方式下,当前运行线程获得一个时间片,时间到时,即使没有执行完也要让出CPU,进入可运行状态,等待下一个时间片的调度.系统选中其他可运行状态的线程执行

分时方式的系统使每个线程工作若干步,实现多线程同时运行

另外请注意下面的线程调度规则（如果有不理解，不急，往下看）：

①如果两个或是两个以上的线程都修改一个对象，那么把执行修改的方法定义为被同步的（Synchronized）,如果对象更新影响

到只读方法，那么只度方法也应该定义为同步的

②如果一个线程必须等待一个对象状态发生变化，那么它应该在对象内部等待，而不是在外部等待，它可以调用一个被同步的方法，并让这个方法调用wait()

③每当一个方法改变某个对象的状态的时候，它应该调用notifyAll()方法，这给等待队列的线程提供机会来看一看执行环境是否已发生改变

④记住wait(),notify(),notifyAll()方法属于Object类，而不是Thread类，仔细检查看是否每次执行wait()方法都有相应的notify()或notifyAll()方法，且它们作用与相同的对象 在java中每个类都有一个主线程，要执行一个程序，那么这个类当中一定要有main方法，这个man方法也就是java class中的主线程。你可以自己创建线程，有两种方法，一是继承Thread类，或是实现Runnable接口。一般情况下，最好避免继承，因为java中是单根继承，如果你选用继承，那么你的类就失去了弹性，当然也不能全然否定继承Thread,该方法编写简单,可以直接操作线程,适用于单重继承情况。至于选用那一种，具体情况具体分析。

eg.继承Thread

```
public class MyThread_1 extends Thread
{
public void run()
{
//some code
}
}
```

eg.实现Runnable接口

```
public class MyThread_2 implements Runnable
{
public void run()
{
//some code
}
}
```

当使用继承创建线程，这样启动线程：

```
new MyThread_1().start()
```

当使用实现接口创建线程，这样启动线程：

```
new Thread(new MyThread_2()).start()
```

注意，其实是创建一个线程实例，并以实现了Runnable接口的类为参数传入这个实例，当执行这个线程的时候，MyThread_2中run里面的代码将被执行。

下面是完成的例子：

```
public class MyThread implements Runnable
{

public void run()
{
System.out.println("My Name is "+Thread.currentThread().getName());
}
public static void main(String[] args)
{
new Thread(new MyThread()).start();
}
}
```

执行后将打印出：

```
My Name is Thread-0
```

你也可以创建多个线程，像下面这样

```
new Thread(new MyThread()).start();
new Thread(new MyThread()).start();
new Thread(new MyThread()).start();
```

那么会打印出：

```
My Name is Thread-0
My Name is Thread-1
My Name is Thread-2
```

看了上面的结果，你可能会认为线程的执行顺序是依次执行的，但是那只是一般情况，千万不要用以为是线程的执行机制；影响线程执行顺序的因素有几点：首先看看前面提到的优先级别

```
public class MyThread implements Runnable
{

public void run()
{
System.out.println("My Name is "+Thread.currentThread().getName());
}
public static void main(String[] args)
{
Thread t1=new Thread(new MyThread());
Thread t2=new Thread(new MyThread());
Thread t3=new Thread(new MyThread());
t2.setPriority(Thread.MAX_PRIORITY);//赋予最高优先级
t1.start();
t2.start();
```

```
t3.start();
}
}
```

再看看结果：

```
My Name is Thread-1
My Name is Thread-0
My Name is Thread-2
```

线程的优先级分为10级，分别用1到10的整数代表，默认情况是5。上面的t2.setPriority(Thread.MAX_PRIORITY)等价与t2.setPriority(10)

然后是线程程序本身的设计，比如使用sleep,yield,join，wait等方法（详情请看JDKDocument）

```
public class MyThread implements Runnable
{
public void run()
{
try
{
int sleepTime=(int)(Math.random()*100);//产生随机数字，
Thread.currentThread().sleep(sleepTime);//让其休眠一定时间，时间又上面sleepTime决定
//public static void sleep(long millis)throw InterruptedException （API）
System.out.println(Thread.currentThread().getName()+" 睡了 "+sleepTime);
}catch(InterruptedException ie)//由于线程在休眠可能被中断，所以调用sleep方法的时候需要捕捉异常
{
ie.printStackTrace();
}
}
public static void main(String[] args)
{
Thread t1=new Thread(new MyThread());
Thread t2=new Thread(new MyThread());
Thread t3=new Thread(new MyThread());
t1.start();
t2.start();
t3.start();
}
}
```

执行后观察其输出：

```
Thread-0 睡了 11
Thread-2 睡了 48
Thread-1 睡了 69
```

上面的执行结果是随机的，再执行很可能出现不同的结果。由于上面我在run中添加了休眠语句，当线程休眠的时候就会让出

cpu，cpu将会选择执行处于runnable状态中的其他线程，当然也可能出现这种情况，休眠的Thread立即进入了runnable状态，cpu再次执行它。

[线程组概念]

线程是可以被组织的，java中存在线程组的概念，每个线程都是一个线程组的成员,线程组把多个线程集成为一个对象,通过线程组可以同时对其中的多个线程进行操作,如启动一个线程组的所有线程等.Java的线程组由 java.lang包中的Thread——Group类实现.

ThreadGroup类用来管理一组线程,包括:线程的数目,线程间的关系,线程正在执行的操作,以及线程将要启动或终止时间等.线程组还可以包含线程组.在Java的应用程序中,最高层的线程组是名位main的线程组,在main中还可以加入线程或线程组,在mian的子线程组中也可以加入线程和线程组,形成线程组和线程之间的树状继承关系。像上面创建的线程都是属于main这个线程组的。

借用上面的例子，main里面可以这样写：

```
public static void main(String[] args)
{
/**************************************
ThreadGroup(String name)
ThreadGroup(ThreadGroup parent, String name)
**********************************/
ThreadGroup group1=new ThreadGroup("group1");
ThreadGroup group2=new ThreadGroup(group1,"group2");
Thread t1=new Thread(group2,new MyThread());
Thread t2=new Thread(group2,new MyThread());
Thread t3=new Thread(group2,new MyThread());
t1.start();
t2.start();
t3.start();
}
```

线程组的嵌套，t1,t2,t3被加入group2,group2加入group1。

另外一个比较多就是关于线程同步方面的，试想这样一种情况，你有一笔存款在银行，你在一家银行为你的账户存款，而你的妻子在另一家银行从这个账户提款，现在你有1000块在你的账户里面。你存入了1000，但是由于另一方也在对这笔存款进行操作，人家开始执行的时候只看到账户里面原来的1000元，当你的妻子提款1000元后，你妻子所在的银行就认为你的账户里面没有钱了，而你所在的银行却认为你还有2000元。

看看下面的例子：

```
class BlankSaving //储蓄账户
{
private static int money=10000;
public void add(int i)
{
money=money+i;
System.out.println("Husband 向银行存入了 [¥"+i+"]");
}
public void get(int i)
{
```

```
money=money-i;
System.out.println("Wife 向银行取走了 [¥"+i+"]");
if(money<0)
System.out.println("余额不足！");
}
public int showMoney()
{
return money;
}
}

class Operater implements Runnable
{
String name;
BlankSaving bs;
public Operater(BlankSaving b,String s)
{
name=s;
bs=b;

}
public static void oper(String name,BlankSaving bs)
{

if(name.equals("husband"))
{
try
{
for(int i=0;i<10;i++)
{
Thread.currentThread().sleep((int)(Math.random()*300));
bs.add(1000);
}
}catch(InterruptedException e){}
}else
{
try
{

for(int i=0;i<10;i++)
{
Thread.currentThread().sleep((int)(Math.random()*300));
bs.get(1000);
}
```

```
}catch(InterruptedException e){}
}
}
public void run()
{
oper(name,bs);
}
}
public class BankTest
{
public static void main(String[] args)throws InterruptedException
{
BlankSaving bs=new BlankSaving();
Operater o1=new Operater(bs,"husband");
Operater o2=new Operater(bs,"wife");
Thread t1=new Thread(o1);
Thread t2=new Thread(o2);
t1.start();
t2.start();
Thread.currentThread().sleep(500);
}

}
```

下面是其中一次的执行结果：

```
————first————————
Husband 向银行存入了 [￥1000]
Wife 向银行取走了 [￥1000]
Wife 向银行取走了 [￥1000]
Husband 向银行存入了 [￥1000]
Wife 向银行取走了 [￥1000]
Husband 向银行存入了 [￥1000]
Wife 向银行取走了 [￥1000]
Husband 向银行存入了 [￥1000]
Wife 向银行取走了 [￥1000]
Husband 向银行存入了 [￥1000]
Husband 向银行存入了 [￥1000]
Wife 向银行取走了 [￥1000]
Husband 向银行存入了 [￥1000]
Husband 向银行存入了 [￥1000]
Wife 向银行取走了 [￥1000]
Wife 向银行取走了 [￥1000]
Husband 向银行存入了 [￥1000]
```

Wife 向银行取走了 [￥1000]
Wife 向银行取走了 [￥1000]
Husband 向银行存入了 [￥1000]

看到了吗，这可不是正确的需求，在husband还没有结束操作的时候，wife就插了进来，这样很可能导致意外的结果。解决办法很简单，就是将对数据进行操作方法声明为synchronized,当方法被该关键字声明后，也就意味着，如果这个数据被加锁，只有一个对象得到这个数据的锁的时候该对象才能对这个数据进行操作。也就是当你存款的时候，这笔账户在其他地方是不能进行操作的，只有你存款完毕，银行管理人员将账户解锁，其他人才能对这个账户进行操作。
修改public static void oper(String name,BlankSaving bs)为public static void oper(String name,BlankSaving bs)，再看看结果:

Husband 向银行存入了 [￥1000]
Husband 向银行存入了 [￥1000]
Husband 向银行存入了 [￥1000]
Husband 向银行存入了 [￥1000]
Husband 向银行存入了 [￥1000]
Husband 向银行存入了 [￥1000]
Husband 向银行存入了 [￥1000]
Husband 向银行存入了 [￥1000]
Husband 向银行存入了 [￥1000]
Husband 向银行存入了 [￥1000]
Wife 向银行取走了 [￥1000]
Wife 向银行取走了 [￥1000]
Wife 向银行取走了 [￥1000]
Wife 向银行取走了 [￥1000]
Wife 向银行取走了 [￥1000]
Wife 向银行取走了 [￥1000]
Wife 向银行取走了 [￥1000]
Wife 向银行取走了 [￥1000]
Wife 向银行取走了 [￥1000]
Wife 向银行取走了 [￥1000]

当丈夫完成操作后，妻子才开始执行操作，这样的话，对共享对象的操作就不会有问题了。
[wait and notify]
你可以利用这两个方法很好的控制线程的执行流程，当线程调用wait方法后，线程将被挂起，直到被另一线程唤醒（notify）或则是如果wait方法指定有时间得话，在没有被唤醒的情况下，指定时间时间过后也将自动被唤醒。但是要注意一定，被唤醒并不是指马上执行，而是从组塞状态变为可运行状态，其是否运行还要看cpu的调度。
事例代码：

```
class MyThread_1 extends Thread
{
Object lock;
public MyThread_1(Object o)
{
lock=o;
```

```
}
public void run()
{
try
{
synchronized(lock)
{
System.out.println("Enter Thread_1 and wait");
lock.wait();
System.out.println("be notified");
}
}catch(InterruptedException e){}
}
}
class MyThread_2 extends Thread
{
Object lock;
public MyThread_2(Object o)
{
lock=o;
}
public void run()
{
synchronized(lock)
{
System.out.println("Enter Thread_2 and notify");
lock.notify();
}
}
}
public class MyThread
{
public static void main(String[] args)
{
int[] in=new int[0];//notice
MyThread_1 t1=new MyThread_1(in);
MyThread_2 t2=new MyThread_2(in);
t1.start();
t2.start();
}
}
```

执行结果如下：

```
Enter Thread_1 and wait
```

Enter Thread_2 and notify

Thread_1 be notified

可能你注意到了在使用wait and notify方法得时候我使用了synchronized块来包装这两个方法，这是由于调用这两个方法的时候线程必须获得锁，也就是上面代码中的lock[]，如果你不用synchronized包装这两个方法的得话，又或则锁不一是同一把，比如在MyThread_2中synchronized(lock)改为 synchronized(this),那么执行这个程序的时候将会抛出java.lang.IllegalMonitorStateException执行期异常。另外wait and notify方法是Object中的，并不在Thread这个类中。最后你可能注意到了这点：int[] in=new int[0];为什么不是创建new Object而是一个0长度的数组，那是因为在java中创建一个0长度的数组来充当锁更加高效。

Thread作为java中一重要组成部分，当然还有很多地方需要更深刻的认识，上面只是对Thread的一些常识和易错问题做了一个简要的总结，若要真正的掌握java的线程，还需要自己多做总结

1.7 Java 5.0 多线程编程

Java自1995年面世以来得到了广泛得一个运用，但是对多线程编程的支持Java很长时间一直停留在初级阶段。在Java 5.0之前Java里的多线程编程主要是通过Thread类，Runnable接口，Object对象中的wait()、 notify()、 notifyAll()等方法和synchronized关键词来实现的。这些工具虽然能在大多数情况下解决对共享资源的管理和线程间的调度，但存在以下几个问题

1. 过于原始，拿来就能用的功能有限，即使是要实现简单的多线程功能也需要编写大量的代码。这些工具就像汇编语言一样难以学习和使用，比这更糟糕的是稍有不慎它们还可能被错误地使用，而且这样的错误很难被发现。

2. 如果使用不当，会使程序的运行效率大大降低。

3. 为了提高开发效率，简化编程，开发人员在做项目的时候往往需要写一些共享的工具来实现一些普遍适用的功能。但因为没有规范，相同的工具会被重复地开发，造成资源浪费。

4. 因为锁定的功能是通过Synchronized来实现的，这是一种块结构，只能对代码中的一段代码进行锁定，而且锁定是单一的。如以下代码所示：

```
synchronized（lock）{
    //执行对共享资源的操作
    ……
}
```

一些复杂的功能就很难被实现。比如说如果程序需要取得lock A和lock B来进行操作1，然后需要取得lock C并且释放lock A来进行操作2，Java 5.0之前的多线程框架就显得无能为力了。

因为这些问题，程序员对旧的框架一直颇有微词。这种情况一直到Java 5.0才有较大的改观，一系列的多线程工具包被纳入了标准库文件。这些工具包括了一个新的多线程程序的执行框架，使编程人员可方便地协调和调度线程的运行，并且新加入了一些高性能的常用的工具，使程序更容易编写，运行效率更高。本文将分类并结合例子来介绍这些新加的多线程工具。

在我们开始介绍Java 5.0里的新Concurrent工具前让我们先来看一下一个用旧的多线程工具编写的程序，这个程序里有一个Server线程，它需要启动两个Component，Server线程需等到Component线程完毕后再继续。相同的功能在Synchronizer一章里用新加的工具CountDownLatch有相同的实现。两个程序，孰优孰劣，哪个程序更容易编写，哪个程序更容易理解，相信大家看过之后不难得出结论。

```
public class ServerThread {
        Object concLock = new Object();
        int count = 2;
public void runTwoThreads() {
        //启动两个线程去初始化组件
                new Thread(new ComponentThread1(this)).start();
                new Thread(new ComponentThread1(this)).start();
                // Wait for other thread
while(count != 0) {
                        synchronized(concLock) {
                                try {
                                        concLock.wait();
                                        System.out.println("Wake up.");
                                } catch (InterruptedException ie) { //处理异常}
                        }
                }
                System.out.println("Server is up.");
        }
        public void callBack() {
synchronized(concLock) {
                        count--;
                        concLock.notifyAll();
                }
        }
        public static void main(String[] args){
                ServerThread server = new ServerThread();
                server.runTwoThreads();
        }
}
```

```
public class ComponentThread1 implements Runnable {
        private ServerThread server;
        public ComponentThread1(ServerThread server) {
                this.server = server;
        }
public void run() {
        //做组件初始化的工作
                System.out.println("Do component initialization.");
                server.callBack();
        }
}
```

1：三个新加的多线程包

Java 5.0 里新加入了三个多线程包：java.util.concurrent, java.util.concurrent.atomic, java.util.concurrent.locks.

java.util.concurrent 包含了常用的多线程工具，是新的多线程工具的主体。

java.util.concurrent.atomic 包含了不用加锁情况下就能改变值的原子变量，比如说AtomicInteger 提供了 addAndGet()方法。Add 和 Get 是两个不同的操作，为了保证别的线程不干扰，以往的做法是先锁定共享的变量，然后在锁定的范围内进行两步操作。但用 AtomicInteger.addAndGet()就不用担心锁定的事了，其内部实现保证了这两步操作是在原子量级发生的，不会被别的线程干扰。

java.util.concurrent.locks 包包含锁定的工具。

2：Callable 和 Future 接口

Callable 是类似于 Runnable 的接口，实现 Callable 接口的类和实现 Runnable 的类都是可被其它线程执行的任务。Callable 和 Runnable 有几点不同：

Callable 规定的方法是 call()，而 Runnable 规定的方法是 run().

Callable 的任务执行后可返回值，而 Runnable 的任务是不能返回值的。

call（）方法可抛出异常，而 run（）方法是不能抛出异常的。

运行 Callable 任务可拿到一个 Future 对象，通过 Future 对象可了解任务执行情况，可取消任务的执行，还可获取任务执行的结果。

以下是 Callable 的一个例子：

```
public class DoCallStuffimplements Callable{ // *1
```

```
        private int aInt;
        public DoCallStuff(int aInt) {
                this.aInt = aInt;
        }
        public String call() throws Exception { //*2
                boolean resultOk = false;
                if(aInt == 0){
                        resultOk = true;
                }   else if(aInt == 1){
                        while(true){ //infiniteloop
                                System.out.println("looping....");
                                Thread.sleep(3000);
                        }
                } else {
                        throw new Exception("Callable terminated with Exception!"); //*3
                }
                if(resultOk){
                        return "Task done.";
                } else {
                        return "Task failed";
                }
        }
}
```

*1: 名为 DoCallStuff类实现了 Callable，String 将是 call 方法的返回值类型。例子中用了 String，但可以是任何 Java 类。

*2: call 方法的返回值类型为 String，这是和类的定义相对应的。并且可以抛出异常。

*3: call 方法可以抛出异常，如加重的斜体字所示。

以下是调用 DoCallStuff的主程序。

```
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
```

```
import java.util.concurrent.Executors;

import java.util.concurrent.Future;

public class Executor {

	public static void main(String[] args){

		//*1

		DoCallStuff call1 = new DoCallStuff(0);

		DoCallStuff call2 = new DoCallStuff(1);

		DoCallStuff call3 = new DoCallStuff(2);

		//*2

		ExecutorService es = Executors.newFixedThreadPool(3);

		//*3

		Future future1 = es.submit(call1);

		Future future2 = es.submit(call2);

		Future future3 = es.submit(call3);

		try {

			//*4

			System.out.println(future1.get());

			 //*5

			Thread.sleep(3000);

			System.out.println("Thread 2 terminated? :" + future2.cancel(true));

			//*6

			System.out.println(future3.get());

		} catch (ExecutionException ex) {

			ex.printStackTrace();

		} catch (InterruptedException ex) {

			ex.printStackTrace();

		}

	}

}
```

*1: 定义了几个任务

*2: 初始了任务执行工具。任务的执行框架将会在后面解释。

*3: 执行任务，任务启动时返回了一个 Future 对象，如果想得到任务执行的结果或者是异常可对这个 Future 对象进行操作。Future 所含的值必须跟 Callable 所含的值对映，比如说例子中 Future 对印 Callable

*4: 任务 1 正常执行完毕，future1.get()会返回线程的值

*5: 任务 2 在进行一个死循环，调用 future2.cancel(true)来中止此线程。传入的参数标明是否可打断线程，true 表明可以打断。

*6: 任务 3 抛出异常，调用 future3.get() 时会引起异常的抛出。

运行 Executor 会有以下运行结果：

```
looping...

Task done. //*1

looping...

looping...//*2

looping...

looping...

looping...

looping...

Thread 2 terminated? :true //*3

//*4

java.util.concurrent.ExecutionException: java.lang.Exception: Callable terminated with Exception!

        at java.util.concurrent.FutureTask$Sync.innerGet(FutureTask.java:205)

        at java.util.concurrent.FutureTask.get(FutureTask.java:80)

        at concurrent.Executor.main(Executor.java:43)

        .....
```

*1: 任务 1 正常结束

*2: 任务 2 是个死循环，这是它的打印结果

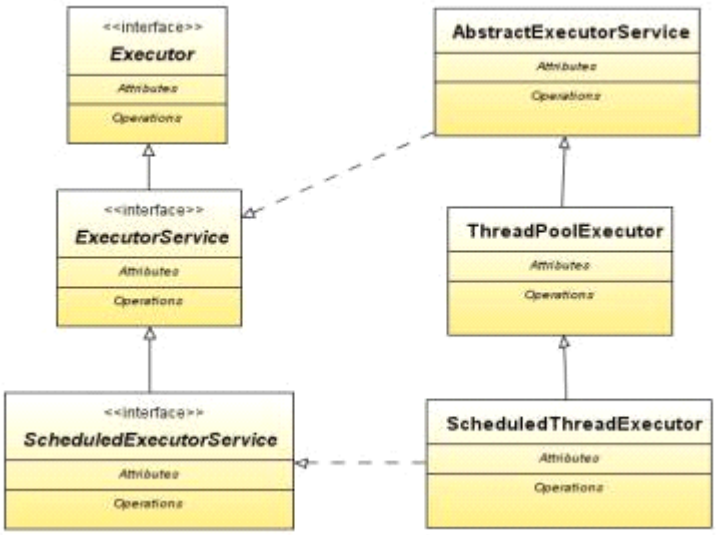
*3: 指示任务 2 被取消

*4: 在执行 future3.get()时得到任务 3 抛出的异常

3：新的任务执行架构

在 Java 5.0 之前启动一个任务是通过调用 Thread 类的 start()方法来实现的，任务的提于交和执行是同时进行的，如果你想对任务的执行进行调度或是控制同时执行的线程数量就需要额外编写代码来完成。5.0 里提供了一个新的任务执行架构使你可以轻

松地调度和控制任务的执行，并且可以建立一个类似数据库连接池的线程池来执行任务。这个架构主要有三个接口和其相应的具体类组成。这三个接口是 Executor, ExecutorService 和 ScheduledExecutorService，让我们先用一个图来显示它们的关系：



图的左侧是接口，图的右侧是这些接口的具体类。注意 Executor 是没有直接具体实现的。

Executor 接口：

是用来执行 Runnable 任务的，它只定义一个方法：

execute(Runnable command)：执行 Ruannable 类型的任务

ExecutorService 接口：

ExecutorService 继承了 Executor 的方法，并提供了执行 Callable 任务和中止任务执行的服务，其定义的方法主要有：

submit(task)：可用来提交 Callable 或 Runnable 任务，并返回代表此任务的 Future 对象

invokeAll(collection of tasks)：批处理任务集合，并返回一个代表这些任务的 Future 对象集合

shutdown()：在完成已提交的任务后关闭服务，不再接受新任务

shutdownNow()：停止所有正在执行的任务并关闭服务。

isTerminated()：测试是否所有任务都执行完毕了。

isShutdown()：测试是否该 ExecutorService 已被关闭

ScheduledExecutorService 接口

在 ExecutorService 的基础上，ScheduledExecutorService 提供了按时间安排执行任务的功能，它提供的方法主要有：

schedule(task, initDelay): 安排所提交的 Callable 或 Runnable 任务在 initDelay 指定的时间后执行。

scheduleAtFixedRate()：安排所提交的 Runnable 任务按指定的间隔重复执行

scheduleWithFixedDelay()：安排所提交的 Runnable 任务在每次执行完后，等待 delay 所指定的时间后重复执行。

代码：ScheduleExecutorService 的例子

```
public class ScheduledExecutorServiceTest {

        public static void main(String[] args)

                throws InterruptedException, ExecutionException{

                //*1

                 ScheduledExecutorService service = Executors.newScheduledThreadPool(2);

                 //*2

                 Runnable task1 = new Runnable() {

                        public void run() {

                            System.out.println("Task repeating.");

                        }

                 };

                 //*3

                 final ScheduledFuture future1 =

                            service.scheduleAtFixedRate(task1, 0, 1, TimeUnit.SECONDS);

                 //*4

                 ScheduledFuture future2 = service.schedule(new Callable(){

                        public String call(){

                                    future1.cancel(true);

                                    return "task cancelled!";

                        }

                 }, 5, TimeUnit.SECONDS);

                 System.out.println(future2.get());

//*5

service.shutdown();

        }

}
```

这个例子有两个任务，第一个任务每隔一秒打印一句“Task repeating”,第二个任务在 5 秒钟后取消第一个任务。

*1: 初始化一个 ScheduledExecutorService 对象，这个对象的线程池大小为 2。

*2: 用内函数的方式定义了一个 Runnable 任务。

*3: 调用所定义的 ScheduledExecutorService 对象来执行任务，任务每秒执行一次。能重复执行的任务一定是Runnable 类型。注意我们可以用 TimeUnit 来制定时间单位，这也是 Java 5.0 里新的特征，5.0 以前的记时单位是微秒，现在可精确到奈秒。

*4: 调用 ScheduledExecutorService 对象来执行第二个任务，第二个任务所作的就是在 5 秒钟后取消第一个任务。

*5: 关闭服务。

Executors 类

虽然以上提到的接口有其实现的具体类，但为了方便 Java 5.0 建议使用 Executors 的工具类来得到 Executor 接口的具体对象，需要注意的是 Executors 是一个类，不是 Executor 的复数形式。Executors 提供了以下一些static 的方法：

callable(Runnable task): 将 Runnable 的任务转化成 Callable 的任务

newSingleThreadExecutor: 产生一个 ExecutorService 对象，这个对象只有一个线程可用来执行任务，若任务多于一个，任务将按先后顺序执行。

newCachedThreadPool(): 产生一个 ExecutorService 对象，这个对象带有一个线程池，线程池的大小会根据需要调整，线程执行完任务后返回线程池，供执行下一次任务使用。

newFixedThreadPool(int poolSize)：产生一个 ExecutorService 对象，这个对象带有一个大小为 poolSize 的线程池，若任务数量大于 poolSize，任务会被放在一个 queue 里顺序执行。

newSingleThreadScheduledExecutor；产生一个ScheduledExecutorService 对象，这个对象的线程池大小为1，若任务多于一个，任务将按先后顺序执行。

newScheduledThreadPool(int poolSize): 产生一个 ScheduledExecutorService 对象，这个对象的线程池大小为 poolSize，若任务数量大于 poolSize，任务会在一个 queue 里等待执行

以下是得到和使用 ExecutorService 的例子：

代码：如何调用 Executors 来获得各种服务对象

```
//Single Threaded ExecutorService
        ExecutorService singleThreadeService = Executors.newSingleThreadExecutor();
//Cached ExecutorService
        ExecutorService cachedService = Executors.newCachedThreadPool();
//Fixed number of ExecutorService
        ExecutorService fixedService = Executors.newFixedThreadPool(3);
//Single ScheduledExecutorService
        ScheduledExecutorService singleScheduledService =
                Executors.newSingleThreadScheduledExecutor();
//Fixed number of ScheduledExecutorService
ScheduledExecutorService fixedScheduledService =
        Executors.newScheduledThreadPool(3);
```

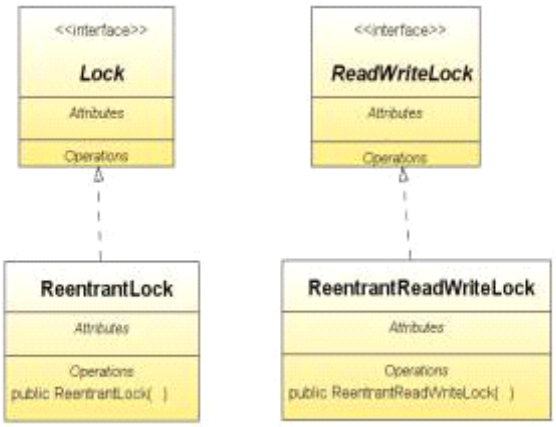
4：Lockers 和 Condition 接口

在多线程编程里面一个重要的概念是锁定，如果一个资源是多个线程共享的，为了保证数据的完整性，在进行事务性操作时需要将共享资源锁定，这样可以保证在做事务性操作时只有一个线程能对资源进行操作，从而保证数据的完整性。在 5.0 以前，锁定的功能是由 Synchronized 关键字来实现的，这样做存在几个问题：

每次只能对一个对象进行锁定。若需要锁定多个对象，编程就比较麻烦，一不小心就会出现死锁现象。

如果线程因拿不到锁定而进入等待状况，是没有办法将其打断的

在 Java 5.0 里出现两种锁的工具可供使用，下图是这两个工具的接口及其实现：



Lock 接口

ReentrantLock 是 Lock 的具体类，Lock 提供了以下一些方法：

lock(): 请求锁定，如果锁已被别的线程锁定，调用此方法的线程被阻断进入等待状态。

tryLock()：如果锁没被别的线程锁定，进入锁定状态，并返回 true。若锁已被锁定，返回 false，不进入等待状态。此方法还可带时间参数，如果锁在方法执行时已被锁定，线程将继续等待规定的时间，若还不行才返回 false。

unlock()：取消锁定，需要注意的是 Lock 不会自动取消，编程时必须手动解锁。

代码：

```
//生成一个锁
Lock lock = new ReentrantLock();
public void accessProtectedResource() {
  lock.lock(); //取得锁定
  try {
    //对共享资源进行操作
  } finally {
    //一定记着把锁取消掉，锁本身是不会自动解锁的
```

```
    lock.unlock();
  }
}
```

ReadWriteLock 接口

为了提高效率有些共享资源允许同时进行多个读的操作，但只允许一个写的操作，比如一个文件，只要其内容不变可以让多个线程同时读，不必做排他的锁定，排他的锁定只有在写的时候需要，以保证别的线程不会看到数据不完整的文件。ReadWriteLock 可满足这种需要。ReadWriteLock 内置两个 Lock，一个是读的 Lock，一个是写的 Lock。多个线程可同时得到读的 Lock，但只有一个线程能得到写的 Lock，而且写的 Lock 被锁定后，任何线程都不能得到 Lock。ReadWriteLock 提供的方法有：

readLock(): 返回一个读的 lock

writeLock(): 返回一个写的 lock，此 lock 是排他的。

ReadWriteLock 的例子：

```
public class FileOperator{
        //初始化一个 ReadWriteLock
        ReadWriteLock lock = new ReentrantReadWriteLock();
public String read() {
        //得到 readLock 并锁定
                Lock readLock = lock.readLock();
                readLock.lock();
                try {
                        //做读的工作
                        return "Read something";
                } finally{
                        readLock.unlock();
                }
        }

public void write(String content) {
        //得到 writeLock 并锁定
                Lock writeLock = lock.writeLock();
```

```
            writeLock.lock();
            try {
                //做读的工作
            } finally {
                writeLock.unlock();
            }
        }
}
```

需要注意的是 ReadWriteLock 提供了一个高效的锁定机理，但最终程序的运行效率是和程序的设计息息相关的，比如说如果读的线程和写的线程同时在等待，要考虑是先发放读的 lock 还是先发放写的 lock。如果写发生的频率不高，而且快，可以考虑先给写的 lock。还要考虑的问题是如果一个写正在等待读完成，此时一个新的读进来，是否要给这个新的读发锁，如果发了，可能导致写的线程等很久。等等此类问题在编程时都要给予充分的考虑。

Condition 接口：

有时候线程取得 lock 后需要在一定条件下才能做某些工作，比如说经典的 Producer 和 Consumer 问题，Consumer 必须在篮子里有苹果的时候才能吃苹果，否则它必须暂时放弃对篮子的锁定，等到 Producer 往篮子里放了苹果后再去拿来吃。而 Producer 必须等到篮子空了才能往里放苹果，否则它也需要暂时解锁等 Consumer 把苹果吃了才能往篮子里放苹果。在 Java 5.0 以前，这种功能是由 Object 类的 wait(), notify()和 notifyAll()等方法实现的，在 5.0 里面，这些功能集中到了 Condition 这个接口来实现，Condition 提供以下方法：

await()：使调用此方法的线程放弃锁定，进入睡眠直到被打断或被唤醒。

signal(): 唤醒一个等待的线程

signalAll()：唤醒所有等待的线程

Condition 的例子：

```
public class Basket {
Lock lock = new ReentrantLock();
//产生 Condition 对象
    Condition produced = lock.newCondition();
    Condition consumed = lock.newCondition();
    boolean available = false;

    public void produce() throws InterruptedException {
```

```
            lock.lock();
            try {
                if(available){
                    consumed.await(); //放弃lock 进入睡眠
                }
                /*生产苹果*/
                System.out.println("Apple produced.");
                available = true;
                produced.signal(); //发信号唤醒等待这个 Condition 的线程
            } finally{
                lock.unlock();
            }
        }

        public void consume() throws InterruptedException {
            lock.lock();
            try {
                if(!available){
                    produced.await();//放弃lock 进入睡眠
                }
                /*吃苹果*/
                System.out.println("Apple consumed.");
                available = false;
                consumed.signal();//发信号唤醒等待这个 Condition 的线程
            } finally{
                lock.unlock();
            }
        }
}
```

ConditionTester:

```
public class ConditionTester {



        public static void main(String[] args) throws InterruptedException{
final Basket basket = new Basket();
//定义一个producer
                Runnable producer = new Runnable() {
                        public void run() {
                                try {
                                        basket.produce();
                                } catch (InterruptedException ex) {
                                        ex.printStackTrace();
                                }
                        }
};
//定义一个consumer
                Runnable consumer = new Runnable() {
                        public void run() {
                                try {
                                        basket.consume();
                                } catch (InterruptedException ex) {
                                        ex.printStackTrace();
                                }
                        }
};
//各产生10 个 consumer 和 producer
                ExecutorService service = Executors.newCachedThreadPool();
                for(int i=0; i < 10; i++)
                        service.submit(consumer);
```

```
            Thread.sleep(2000);
            for(int i=0; i<10; i++)
                service.submit(producer);
            service.shutdown();
        }
}
```

5: Synchronizer：同步装置

Java 5.0 里新加了 4 个协调线程间进程的同步装置，它们分别是 Semaphore, CountDownLatch, CyclicBarrier 和 Exchanger.

Semaphore:

用来管理一个资源池的工具，Semaphore 可以看成是个通行证，线程要想从资源池拿到资源必须先拿到通行证，Semaphore 提供的通行证数量和资源池的大小一致。如果线程暂时拿不到通行证，线程就会被阻断进入等待状态。以下是一个例子：

```
public class Pool {
        ArrayList pool= null;
        Semaphore pass = null;
        public Pool(int size){
            //初始化资源池
            pool= new ArrayList();
            for(int i=0; i
                pool.add("Resource "+i);
            }
            //Semaphore 的大小和资源池的大小一致
            pass = new Semaphore(size);
        }
        public String get() throws InterruptedException{
            //获取通行证,只有得到通行证后才能得到资源
            pass.acquire();
            return getResource();
        }
        public void put(String resource){
```

```
            //归还通行证，并归还资源
            pass.release();
            releaseResource(resource);
        }
        private synchronized String getResource() {
            String result = pool.get(0);
            pool.remove(0);
            System.out.println("Give out "+result);
            return result;
        }
        private synchronized void releaseResource(String resource) {
            System.out.println("return "+resource);
            pool.add(resource);
        }
}
```

SemaphoreTest:

```
public class SemaphoreTest {
    public static void main(String[] args){
        final Pool aPool = new Pool(2);
        Runnable worker = new Runnable() {
            public void run() {
                String resource = null;
                try {
                    //取得resource
                    resource = aPool.get();
                } catch (InterruptedException ex) {
                    ex.printStackTrace();
                }
                //用 resource 做工作
```

```
                        System.out.println("I worked on "+resource);
                        //归还resource
                        aPool.put(resource);
                    }
                };
                ExecutorService service = Executors.newCachedThreadPool();
                for(int i=0; i<20; i++){
                    service.submit(worker);
                }
                service.shutdown();
        }
}
```

CountDownLatch:

CountDownLatch 是个计数器，它有一个初始数，等待这个计数器的线程必须等到计数器倒数到零时才可继续。比如说一个 Server 启动时需要初始化 4 个部件，Server 可以同时启动 4 个线程去初始化这 4 个部件，然后调用 CountDownLatch(4).await() 阻断进入等待，每个线程完成任务后会调用一次 CountDownLatch.countDown() 来倒计数，当 4 个线程都结束时 CountDownLatch 的计数就会降低为 0，此时 Server 就会被唤醒继续下一步操作。CountDownLatch 的方法主要有：

await()：使调用此方法的线程阻断进入等待

countDown(): 倒计数，将计数值减 1

getCount(): 得到当前的计数值

CountDownLatch 的例子：一个 server 调了三个 ComponentThread 分别去启动三个组件，然后 server 等到组件都启动了再继续。

```
public class Server {
    public static void main(String[] args) throws InterruptedException{
        System.out.println("Server is starting.");
        //初始化一个初始值为 3 的 CountDownLatch
        CountDownLatch latch = new CountDownLatch(3);
        //起 3 个线程分别去启动 3 个组件
        ExecutorService service = Executors.newCachedThreadPool();
        service.submit(new ComponentThread(latch, 1));
```

```
            service.submit(new ComponentThread(latch, 2));
            service.submit(new ComponentThread(latch, 3));
            service.shutdown();
            //进入等待状态
            latch.await();
            //当所需的三个组件都完成时，Server 就可继续了
            System.out.println("Server is up!");
        }
}

public class ComponentThread implements Runnable{
        CountDownLatch latch;
        int ID;
        /** Creates a new instance of ComponentThread */
        public ComponentThread(CountDownLatch latch, int ID) {
            this.latch = latch;
            this.ID = ID;
        }
        public void run() {
            System.out.println("Component "+ID + " initialized!");
            //将计数减一
            latch.countDown();
        }
}
```

运行结果：

```
Server is starting.
Component 1 initialized!
Component 3 initialized!
Component 2 initialized!
```

```
Server is up!
```

CyclicBarrier:

CyclicBarrier 类似于CountDownLatch 也是个计数器，不同的是CyclicBarrier 数的是调用了 CyclicBarrier.await()进入等待的线程数，当线程数达到了CyclicBarrier 初始时规定的数目时，所有进入等待状态的线程被唤醒并继续。CyclicBarrier 就象它名字的意思一样，可看成是个障碍，所有的线程必须到齐后才能一起通过这个障碍。CyclicBarrier 初始时还可带一个 Runnable 的参数，此 Runnable 任务在 CyclicBarrier 的数目达到后，所有其它线程被唤醒前被执行。

CyclicBarrier 提供以下几个方法：

await()：进入等待

getParties()：返回此 barrier 需要的线程数

reset()：将此 barrier 重置

以下是使用 CyclicBarrier 的一个例子：两个线程分别在一个数组里放一个数，当这两个线程都结束后，主线程算出数组里的数的和（这个例子比较无聊，我没有想到更合适的例子）

```
public class MainThread {

public static  void main(String[] args)

        throws InterruptedException, BrokenBarrierException, TimeoutException{

                final int[] array = new int[2];

                CyclicBarrier barrier = new CyclicBarrier(2,

                        new Runnable() {//在所有线程都到达 Barrier 时执行

                        public void run() {

                                System.out.println("Total is:"+(array[0]+array[1]));

                        }

                });

                //启动线程

                new Thread(new ComponentThread(barrier, array, 0)).start();

                new Thread(new ComponentThread(barrier, array, 1)).start();

        }

}



public class ComponentThread implements Runnable{

        CyclicBarrier barrier;
```

```
        int ID;
        int[] array;
        public ComponentThread(CyclicBarrier barrier, int[] array, int ID) {
                this.barrier = barrier;
                this.ID = ID;
                this.array = array;
        }
        public void run() {
                try {
                        array[ID] = new Random().nextInt();
                        System.out.println(ID+ " generates:"+array[ID]);
                        //该线程完成了任务等在Barrier 处
                        barrier.await();
                } catch (BrokenBarrierException ex) {
                        ex.printStackTrace();
                } catch (InterruptedException ex) {
                        ex.printStackTrace();
                }
        }
}
```

Exchanger:

顾名思义 Exchanger 让两个线程可以互换信息。用一个例子来解释比较容易。例子中服务生线程往空的杯子里倒水，顾客线程从装满水的杯子里喝水，然后通过 Exchanger 双方互换杯子，服务生接着往空杯子里倒水，顾客接着喝水，然后交换，如此周而复始。

```
class FillAndEmpty{
        //初始化一个Exchanger，并规定可交换的信息类型是 DataCup
        Exchanger exchanger = new Exchanger();
        Cup initialEmptyCup = ...; //初始化一个空的杯子
        Cup initialFullCup = ...; //初始化一个装满水的杯子
```

```
//服务生线程
class Waiter implements Runnable {
        public void run() {
                Cup currentCup = initialEmptyCup;
                try {
                        //往空的杯子里加水
                        currentCup.addWater();
                        //杯子满后和顾客的空杯子交换
                        currentCup = exchanger.exchange(currentCup);
                } catch (InterruptedException ex) { ... handle ... }
        }
}
//顾客线程
class Customer implements Runnable {
        public void run() {
                DataCup currentCup = initialFullCup;
                try {
                        //把杯子里的水喝掉
                        currentCup.drinkFromCup();
                        //将空杯子和服务生的满杯子交换
                        currentCup = exchanger.exchange(currentCup);
                } catch (InterruptedException ex) { ... handle ...}
        }
}

void start() {
        new Thread(new Waiter()).start();
        new Thread(new Customer()).start();
}
```

```
}
```

6: BlockingQueue接口

BlockingQueue是一种特殊的Queue，若BlockingQueue是空的，从BlockingQueue取东西的操作将会被阻断进入等待状态直到BlocingkQueue进了新货才会被唤醒。同样，如果BlockingQueue是满的任何试图往里存东西的操作也会被阻断进入等待状态，直到BlockingQueue里有新的空间才会被唤醒继续操作。BlockingQueue提供的方法主要有：

add(anObject)：把anObject加到BlockingQueue里，如果BlockingQueue可以容纳返回true，否则抛出IllegalStateException异常。

offer(anObject)：把anObject加到BlockingQueue里，如果BlockingQueue可以容纳返回true，否则返回false。

put(anObject)：把anObject加到BlockingQueue里，如果BlockingQueue没有空间，调用此方法的线程被阻断直到BlockingQueue里有新的空间再继续。

poll(time)：取出BlockingQueue里排在首位的对象，若不能立即取出可等time参数规定的时间。取不到时返回null。

take()：取出BlockingQueue里排在首位的对象，若BlockingQueue为空，阻断进入等待状态直到BlockingQueue有新的对象被加入为止。

根据不同的需要BlockingQueue有4种具体实现：

ArrayBlockingQueue：规定大小的BlockingQueue，其构造函数必须带一个int参数来指明其大小。其所含的对象是以FIFO（先入先出）顺序排序的。

LinkedBlockingQueue：大小不定的BlockingQueue，若其构造函数带一个规定大小的参数，生成的BlockingQueue有大小限制，若不带大小参数，所生成的BlockingQueue的大小由Integer.MAX_VALUE来决定。其所含的对象是以FIFO（先入先出）顺序排序的。LinkedBlockingQueue和ArrayBlockingQueue比较起来，它们背后所用的数据结构不一样，导致LinkedBlockingQueue的数据吞吐量要大于ArrayBlockingQueue，但在线程数量很大时其性能的可预见性低于ArrayBlockingQueue。

PriorityBlockingQueue：类似于LinkedBlockingQueue，但其所含对象的排序不是FIFO，而是依据对象的自然排序顺序或者是构造函数所带的Comparator决定的顺序。

SynchronousQueue：特殊的BlockingQueue，对其的操作必须是放和取交替完成的。

下面是用BlockingQueue来实现Producer和Consumer的例子：

```
public class BlockingQueueTest {

        static BlockingQueue basket;

        public BlockingQueueTest() {

                //定义了一个大小为2的BlockingQueue，也可根据需要用其他的具体类

                basket = new ArrayBlockingQueue(2);

        }

        class Producor implements Runnable {

                public void run() {
```

```
                while(true){
                        try {
                                //放入一个对象，若basket 满了，等到basket 有位置
                                basket.put("An apple");
                        } catch (InterruptedException ex) {
                                ex.printStackTrace();
                        }
                }
        }
}
class Consumer implements Runnable {
        public void run() {
                while(true){
                        try {
                                //取出一个对象，若basket 为空，等到basket 有东西为止
                                String result = basket.take();
                        } catch (InterruptedException ex) {
                                ex.printStackTrace();
                        }
                }
        }
}
public void execute(){
        for(int i=0; i<10; i++){
                new Thread(new Producor()).start();
                new Thread(new Consumer()).start();
        }
}
public static void main(String[] args){
```

```
        BlockingQueueTest test = new BlockingQueueTest();
        test.execute();
    }
}
```

7：Atomics 原子级变量

原子量级的变量，主要的类有 AtomicBoolean, AtomicInteger, AotmicIntegerArray, AtomicLong, AtomicLongArray, AtomicReference .....。这些原子量级的变量主要提供两个方法：

compareAndSet(expectedValue, newValue): 比较当前的值是否等于expectedValue,若等于把当前值改成 newValue，并返回true。若不等，返回 false。

getAndSet(newValue): 把当前值改为 newValue，并返回改变前的值。

这些原子级变量利用了现代处理器（CPU）的硬件支持可把两步操作合为一步的功能，避免了不必要的锁定，提高了程序的运行效率。

8：Concurrent Collections 共点聚集

在 Java 的聚集框架里可以调用 Collections.synchronizeCollection(aCollection)将普通聚集改变成同步聚集，使之可用于多线程的环境下。但同步聚集在一个时刻只允许一个线程访问它，其它想同时访问它的线程会被阻断，导致程序运行效率不高。Java 5.0 里提供了几个共点聚集类，它们把以前需要几步才能完成的操作合成一个原子量级的操作，这样就可让多个线程同时对聚集进行操作，避免了锁定，从而提高了程序的运行效率。Java 5.0 目前提供的共点聚集类有： ConcurrentHashMap, ConcurrentLinkedQueue, CopyOnWriteArrayList 和 CopyOnWriteArraySet.

1.8 Java Socket 编程

第一步 充分理解 Socket

1. 什么是 socket

所谓 socket 通常也称作"套接字"，用于描述 IP 地址和端口，是一个通信链的句柄。应用程序通常通过"套接字"向网络发出请求或者应答网络请求。

以 J2SDK-1.3 为例，Socket 和 ServerSocket 类库位于 java.net 包中。ServerSocket 用于服务器端，Socket 是建立网络连接时使用的。在连接成功时，应用程序两端都会产生一个 Socket 实例，操作这个实例，完成所需的会话。对于一个网络连接来说，套接字是平等的，并没有差别，不因为在服务器端或在客户端而产生不同级别。不管是 Socket 还是 ServerSocket 它们的工作都是通过 SocketImpl 类及其子类完成的。

重要的 Socket API：

java.net.Socket 继承于 java.lang.Object，有八个构造器，其方法并不多，下面介绍使用最频繁的三个方法，其它方法大家可以见 JDK-1.3 文档。

. Accept 方法用于产生"阻塞"，直到接受到一个连接，并且返回一个客户端的 Socket 对象实例。"阻塞"是一个术语，它

使程序运行暂时"停留"在这个地方，直到一个会话产生，然后程序继续；通常"阻塞"是由循环产生的。

. getInputStream方法获得网络连接输入，同时返回一个IutputStream对象实例，。

. getOutputStream方法连接的另一端将得到输入，同时返回一个OutputStream对象实例。

注意：其中getInputStream和getOutputStream方法均会产生一个IOException，它必须被捕获，因为它们返回的流对象，通常都会被另一个流对象使用。

2.如何开发一个Server-Client模型的程序

开发原理：

服务器，使用ServerSocket监听指定的端口，端口可以随意指定（由于1024以下的端口通常属于保留端口，在一些操作系统中不可以随意使用，所以建议使用大于1024的端口），等待客户连接请求，客户连接后，会话产生；在完成会话后，关闭连接。

客户端，使用Socket对网络上某一个服务器的某一个端口发出连接请求，一旦连接成功，打开会话；会话完成后，关闭Socket。客户端不需要指定打开的端口，通常临时的、动态的分配一个1024以上的端口。

{建立服务器}

```
import java.net.*;
import java.io.*;

public class Server
{
private ServerSocket ss;
private Socket socket;
private BufferedReader in;
private PrintWriter out;

public Server()
{
try
{
ss = new ServerSocket(10000);

while (true)
{
socket = ss.accept();
in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
out = new PrintWriter(socket.getOutputStream(),true);

String line = in.readLine();
```

```
out.println("you input is :" + line);
out.close();
in.close();
socket.close();
}
ss.close();
}
catch (IOException e)
{}
}

public static void main(String[] args)
{
new Server();
}
}
```

这个程序建立了一个服务器，它一直监听10000端口，等待用户连接。在建立连接后给客户端返回一段信息，然后结束会话。这个程序一次只能接受一个客户连接。

{建立客户端}

```
import java.io.*;
import java.net.*;

public class Client
{
Socket socket;
BufferedReader in;
PrintWriter out;

public Client()
{
try
{
socket = new Socket("xxx.xxx.xxx.xxx", 10000);
in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
out = new PrintWriter(socket.getOutputStream(),true);
BufferedReader line = new BufferedReader(new InputStreamReader(System.in));

out.println(line.readLine());
line.close();
out.close();
in.close();
```

```
socket.close();
}
catch (IOException e)
{}
}

public static void main(String[] args)
{
new Client();
}
}
```

这个客户端连接到地址为xxx.xxx.xxx.xxx的服务器，端口为10000，并从键盘输入一行信息，发送到服务器，然后接受服务器的返回信息，最后结束会话。

第二步　多个客户同时连接

在实际的网络环境里，同一时间只对一个用户服务是不可行的。一个优秀的网络服务程序除了能处理用户的输入信息，还必须能够同时响应多个客户端的连接请求。在java中，实现以上功能特点是非常容易的。

设计原理：

主程序监听一端口，等待客户接入；同时构造一个线程类，准备接管会话。当一个Socket会话产生后，将这个会话交给线程处理，然后主程序继续监听。运用Thread类或Runnable接口来实现是不错的办法。

{实现消息共享}

```
import java.io.*;
import java.net.*;

public class Server extends ServerSocket
{
private static final int SERVER_PORT = 10000;

public Server() throws IOException
{
super(SERVER_PORT);

try
{
while (true)
{
Socket socket = accept();
new CreateServerThread(socket);
}
```

```
}
catch (IOException e)
{}
finally
{
close();
}
}
//—— CreateServerThread
class CreateServerThread extends Thread
{
private Socket client;
private BufferedReader in;
private PrintWriter out;

public CreateServerThread(Socket s) throws IOException
{
client = s;

in = new BufferedReader(new InputStreamReader(client.getInputStream(), "GB2312"));
out = new PrintWriter(client.getOutputStream(), true);
out.println("—— Welcome ——");
start();
}

public void run()
{
try
{
String line = in.readLine();

while (!line.equals("bye"))
{
String msg = createMessage(line);
out.println(msg);
line = in.readLine();
}
out.println("—— See you, bye! ——");
client.close();
}
catch (IOException e)
{}
}
```

```
private String createMessage(String line)
{
xxxxxxxxx;
}
}

public static void main(String[] args) throws IOException
{
new Server();
}
}
```

这个程序监听10000端口，并将接入交给CreateServerThread线程运行。CreateServerThread线程接受输入，并将输入回应客户，直到客户输入"bye"，线程结束。我们可以在createMessage方法中，对输入进行处理，并产生结果，然后把结果返回给客户。

第三步 实现信息共享:在Socket上的实时交流

网络的伟大之一也是信息共享，Server可以主动向所有Client广播消息，同时Client也可以向其它Client发布消息。下面看看如何开发一个可以实时传递消息的程序。

设计原理：

服务器端接受客户端的连接请求，同时启动一个线程处理这个连接，线程不停的读取客户端输入，然后把输入加入队列中，等候处理。在线程启动的同时将线程加入队列中，以便在需要的时候定位和取出。

{源码}

```
import java.io.*;
import java.net.*;
import java.util.*;
import java.lang.*;

public class Server extends ServerSocket
{
private static ArrayList User_List = new ArrayList();
private static ArrayList Threader = new ArrayList();
private static LinkedList Message_Array = new LinkedList();
private static int Thread_Counter = 0;
private static boolean isClear = true;
protected static final int SERVER_PORT = 10000;
protected FileOutputStream LOG_FILE = new FileOutputStream("d:/connect.log", true);

public Server() throws FileNotFoundException, IOException
{
```

```
super(SERVER_PORT);
new Broadcast();

//append connection log
Calendar now = Calendar.getInstance();
String str = "[" + now.getTime().toString() + "] Accepted a connection\015\012";
byte[] tmp = str.getBytes();
LOG_FILE.write(tmp);

try
{
while (true)
{
Socket socket = accept();
new CreateServerThread(socket);
}
}
finally
{
close();
}
}

public static void main(String[] args) throws IOException
{
new Server();
}

//── Broadcast
class Broadcast extends Thread
{
public Broadcast()
{
start();
}

public void run()
{
while (true)
{
if (!isClear)
{
String tmp = (String)Message_Array.getFirst();
```

```
for (int i = 0; i < Threader.size(); i++)
{
CreateServerThread client = (CreateServerThread)Threader.get(i);
client.sendMessage(tmp);
}

Message_Array.removeFirst();
isClear = Message_Array.size() > 0 ? false : true;
}
}
}
}

//—— CreateServerThread
class CreateServerThread extends Thread
{
private Socket client;
private BufferedReader in;
private PrintWriter out;
private String Username;

public CreateServerThread(Socket s) throws IOException
{
client = s;
in = new BufferedReader(new InputStreamReader(client.getInputStream()));
out = new PrintWriter(client.getOutputStream(), true);
out.println("—— Welcome to this chatroom ——");
out.println("Input your nickname:");
start();
}

public void sendMessage(String msg)
{
out.println(msg);
}

public void run()
{
try
{
int flag = 0;
Thread_Counter++;
String line = in.readLine();
```

```
while (!line.equals("bye"))
{
if (line.equals("1"))
{
out.println(listOnlineUsers());
line = in.readLine();
continue;
}

if (flag++ == 0)
{
Username = line;
User_List.add(Username);
out.println(listOnlineUsers());
Threader.add(this);
pushMessage("[< " + Username + " come o n in >]");
}
else
{
pushMessage("<" + Username + ">" + line);
}

line = in.readLine();
}

out.println("—— See you, bye! ——");
client.close();
}
catch (IOException e)
{}
finally
{
try
{
client.close();
}
catch (IOException e)
{}

Thread_Counter—;
Threader.remove(this);
User_List.remove(Username);
pushMessage("[< " + Username + " left>]");
}
```

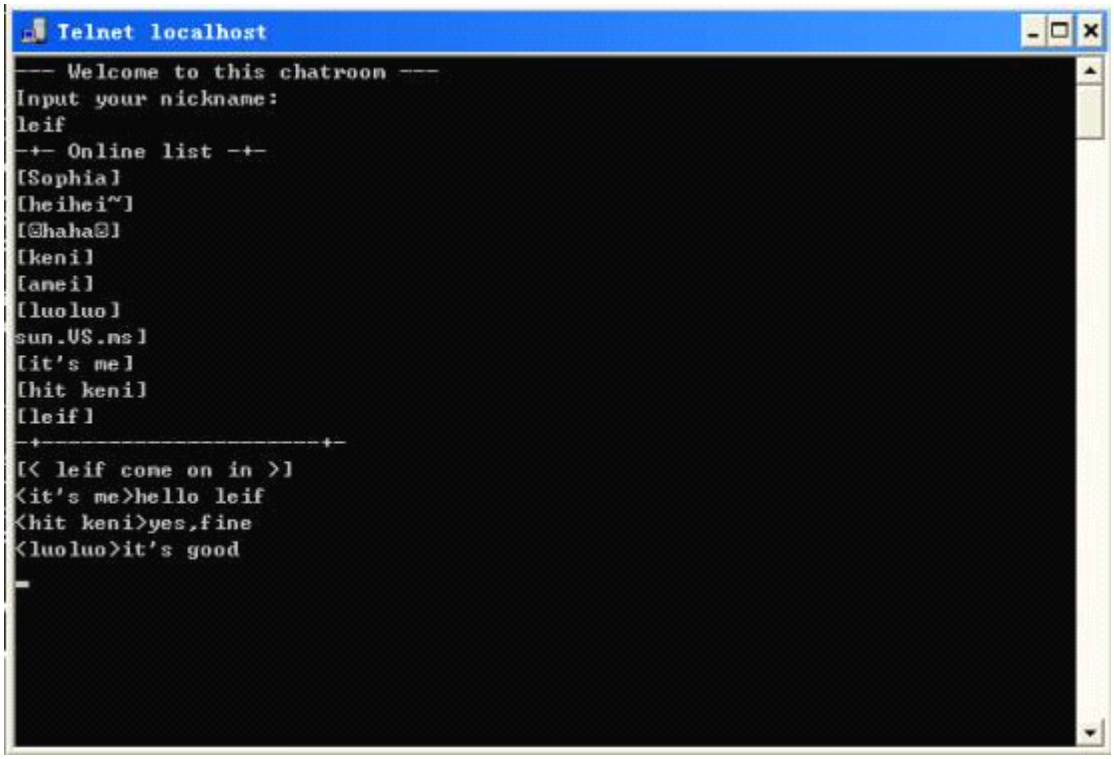
```
}

private String listOnlineUsers()
{
String s ="-+- o nline list -+-\015\012";

for (int i = 0; i < User_List.size(); i++)
{
s += "[" + User_List.get(i) + "]\015\012";
}

s += "-+--------------------+-";
return s;
}

private void pushMessage(String msg)
{
Message_Array.addLast(msg);
isClear = false;
}
}
}
```




```
screen.width-333)this.width=screen.width-333;" border=0>
```

这就是程序运行后，多用户登陆并且输入信息后的屏幕。实现了信息的实时广播。用户输入"1"就可以列出在线人员表。

1.9 Java 的内存泄漏

一 问题的提出

Java 的一个重要优点就是通过垃圾收集器(Garbage Collection，GC)自动管理内存的回收，程序员不需要通过调用函数来释放内存。因此，很多程序员认为 Java 不存在内存泄漏问题，或者认为即使有内存泄漏也不是程序的责任，而是 GC 或 JVM 的问题。其实，这种想法是不正确的，因为 Java 也存在内存泄露，但它的表现与 C++不同。

随着越来越多的服务器程序采用 Java 技术，例如 JSP，Servlet， EJB 等，服务器程序往往长期运行。另外，在很多嵌入式系统中，内存的总量非常有限。内存泄露问题也就变得十分关键，即使每次运行少量泄漏，长期运行之后，系统也是面临崩溃的危险。

二 Java 是如何管理内存

为了判断 Java 中是否有内存泄露，我们首先必须了解 Java 是如何管理内存的。Java 的内存管理就是对象的分配和释放问题。在 Java 中，程序员需要通过关键字 new 为每个对象申请内存空间 (基本类型除外)，所有的对象都在堆 (Heap)中分配空间。另外，对象的释放是由 GC 决定和执行的。在 Java 中，内存的分配是由程序完成的，而内存的释放是有 GC 完成的，这种收支两条线的方法确实简化了程序员的工作。但同时，它也加重了 JVM 的工作。这也是 Java 程序运行速度较慢的原因之一。因为，GC 为了能够正确释放对象，GC 必须监控每一个对象的运行状态，包括对象的申请、引用、被引用、赋值等，GC 都需要进行监控。

监视对象状态是为了更加准确地、及时地释放对象，而释放对象的根本原则就是该对象不再被引用。

为了更好理解 GC 的工作原理，我们可以将对象考虑为有向图的顶点，将引用关系考虑为图的有向边，有向边从引用者指向被引对象。另外，每个线程对象可以作为一个图的起始顶点，例如大多程序从 main 进程开始执行，那么该图就是以 main 进程顶点开始的一棵根树。在这个有向图中，根顶点可达的对象都是有效对象，GC 将不回收这些对象。如果某个对象 (连通子图)与这个根顶点不可达(注意，该图为有向图)，那么我们认为这个(这些)对象不再被引用，可以被 GC 回收。

以下，我们举一个例子说明如何用有向图表示内存管理。对于程序的每一个时刻，我们都有一个有向图表示 JVM 的内存分配情况。以下右图，就是左边程序运行到第 6 行的示意图。

Java 使用有向图的方式进行内存管理，可以消除引用循环的问题，例如有三个对象，相互引用，只要它们和根进程不可达的，那么 GC 也是可以回收它们的。这种方式的优点是管理内存的精度很高，但是效率较低。另外一种常用的内存管理技术是使用计数器，例如 COM 模型采用计数器方式管理构件，它与有向图相比，精度行低(很难处理循环引用的问题)，但执行效率很高。

三 什么是 Java 中的内存泄露

下面，我们就可以描述什么是内存泄漏。在 Java 中，内存泄漏就是存在一些被分配的对象，这些对象有下面两个特点，首先，这些对象是可达的，即在有向图中，存在通路可以与其相连；其次，这些对象是无用的，即程序以后不会再使用这些对象。如果对象满足这两个条件，这些对象就可以判定为 Java 中的内存泄漏，这些对象不会被 GC 所回收，然而它却占用内存。

在 C++中，内存泄漏的范围更大一些。有些对象被分配了内存空间，然后却不可达，由于 C++中没有 GC，这些内存将永远收不回来。在 Java 中，这些不可达的对象都由 GC 负责回收，因此程序员不需要考虑这部分的内存泄露。

通过分析，我们得知，对于 C++，程序员需要自己管理边和顶点，而对于 Java 程序员只需要管理边就可以了(不需要管理顶点的释放)。通过这种方式，Java 提高了编程的效率。

因此，通过以上分析，我们知道在 Java 中也有内存泄漏，但范围比 C++要小一些。因为 Java 从语言上保证，任何对象都是可达的，所有的不可达对象都由 GC 管理。

对于程序员来说，GC 基本是透明的，不可见的。虽然，我们只有几个函数可以访问 GC，例如运行 GC 的函数 System.gc()，但是根据 Java 语言规范定义， 该函数不保证 JVM 的垃圾收集器一定会执行。因为，不同的 JVM 实现者可能使用不同的算法管理 GC。通常，GC 的线程的优先级别较低。JVM 调用 GC 的策略也有很多种，有的是内存使用到达一定程度时，GC 才开始工作，也有定时执行的，有的是平缓执行 GC，有的是中断式执行 GC。但通常来说，我们不需要关心这些。除非在一些特定的场合，GC 的执行影响应用程序的性能，例如对于基于 Web 的实时系统，如网络游戏等，用户不希望 GC 突然中断应用程序执

行而进行垃圾回收，那么我们需要调整GC 的参数，让GC 能够通过平缓的方式释放内存，例如将垃圾回收分解为一系列的小步骤执行，Sun 提供的 HotSpot JVM 就支持这一特性。

下面给出了一个简单的内存泄露的例子。在这个例子中，我们循环申请Object对象，并将所申请的对象放入一个Vector中，如果我们仅仅释放引用本身，那么 Vector 仍然引用该对象，所以这个对象对 GC 来说是不可回收的。因此，如果对象加入到 Vector 后，还必须从 Vector 中删除，最简单的方法就是将 Vector 对象设置为 null。

```
Vector v=new Vector(10);
for (int i=1;i<100; i++)
{
Object o=new Object();
v.add(o);
o=null;
}
//此时，所有的Object 对象都没有被释放，因为变量 v 引用这些对象。
```

四 如何检测内存泄漏

最后一个重要的问题，就是如何检测Java 的内存泄漏。目前，我们通常使用一些工具来检查Java 程序的内存泄漏问题。市场上已有几种专业检查 Java 内存泄漏的工具，它们的基本工作原理大同小异，都是通过监测 Java 程序运行时，所有对象的申请、释放等动作，将内存管理的所有信息进行统计、分析、可视化。开发人员将根据这些信息判断程序是否有内存泄漏问题。这些工具包括 Optimizeit Profiler，JProbe Profiler，JinSight，Rational 公司的 Purify 等。

下面，我们将简单介绍 Optimizeit 的基本功能和工作原理。

Optimizeit Profiler 版本4.11 支持Application，Applet，Servlet 和 Romote Application 四类应用，并且可以支持大多数类型的 JVM，包括 SUN JDK 系列，IBM 的 JDK 系列，和 Jbuilder 的 JVM 等。并且，该软件是由 Java 编写，因此它支持多种操作系统。Optimizeit 系列还包括 Thread Debugger 和 Code Coverage 两个工具，分别用于监测运行时的线程状态和代码覆盖面。

当设置好所有的参数了，我们就可以在OptimizeIt 环境下运行被测程序，在程序运行过程中，Optimizeit 可以监视内存的使用曲线(如下图)，包括 JVM 申请的堆(heap)的大小，和实际使用的内存大小。另外，在运行过程中，我们可以随时暂停程序的运行，甚至强行调用 GC，让GC 进行内存回收。通过内存使用曲线，我们可以整体了解程序使用内存的情况。这种监测对于长期运行的应用程序非常有必要，也很容易发现内存泄露。

在运行过程中，我们还可以从不同视角观查内存的使用情况，Optimizeit 提供了四种方式：

堆视角。 这是一个全面的视角，我们可以了解堆中的所有的对象信息(数量和种类)，并进行统计、排序，过滤。了解相关对象的变化情况。

方法视角。通过方法视角，我们可以得知每一种类的对象，都分配在哪些方法中，以及它们的数量。

对象视角。给定一个对象，通过对象视角，我们可以显示它的所有出引用和入引用对象，我们可以了解这个对象的所有引用关系。

引用图。 给定一个根，通过引用图，我们可以显示从该顶点出发的所有出引用。

在运行过程中，我们可以随时观察内存的使用情况，通过这种方式，我们可以很快找到那些长期不被释放，并且不再使用的对象。我们通过检查这些对象的生存周期，确认其是否为内存泄露。在实践当中，寻找内存泄露是一件非常麻烦的事情，它需要程序员对整个程序的代码比较清楚，并且需要丰富的调试经验，但是这个过程对于很多关键的Java 程序都是十分重要的。

综上所述，Java 也存在内存泄露问题，其原因主要是一些对象虽然不再被使用，但它们仍然被引用。为了解决这些问题，我们可以通过软件工具来检查内存泄露，检查的主要原理就是暴露出所有堆中的对象，让程序员寻找那些无用但仍被引用的对象。

1.10 抽象类与接口的区别

有以下几个方面：

1 自身的定义不同

抽象类可以有属性，接口即使有属性也必须是为常数

抽象类是用来继承的，接口是用来实现的

2 与使用他们的类的关系不同

抽象类的方法必须实现，而接口则可以不实现

抽象类与子类是父子关系，而接口跟类是没有任何关系的，接口可以让任何类去实现

他们的本质区别体现在他们对于一个系统的模型的理解不同

1.11 Java 变量类型间的相互转换

我们知道，Java 的数据类型分为三大类，即布尔型、字符型和数值型，而其中数值型又分为整型和浮点型；相对于数据类型，Java 的变量类型为布尔型 boolean；字符型char；整型byte、short、int、long；浮点型float、double。其中四种整型变量和两种浮点型变量分别 对应于不同的精度和范围。此外，我们还经常用到两种类变量，即 String 和 Date。对于这些变量类型之间的相互转换在我们编程中经常要用到，在我们今 天的这篇文章中，我们将来看看如何实现这些转换。

一、 整型、实型、字符型变量中的相互转换

在 Java 中整型、实型、字符型被视为同一类数据，这些类型由低级到高级分别为(byte，short，char)??int??long??float??double，低级变量可以直接转换为高级变量，例如，下面的语句可以在 Java 中直接通过：

```
byte b;
int i=b;
```

而将高级变量转换为低级变量时，情况会复杂一些，你可以使用强制类型转换。即你必须采用下面这种语句格式：

```
int i;
byte b=(byte)i;
```

可以想象，这种转换肯定可能会导致溢出或精度的下降，因此我们并不推荐使用这种转换。

二、Java 的包装类

在 我们讨论其它变量类型之间的相互转换时，我们需要了解一下 Java 的包装类，所谓包装类，就是可以直接将简单类型的变量表示为一个类，在执行变量类型的相 互转换时，我们会大量使用这些包装类。Java 共有六个包装类，分别是 Boolean、Character、Integer、Long、Float 和 Double，从字面上我们就可以看出它们分别对应于 boolean、char、int、long、float 和 double。而 String 和 Date 本身就是类。所以也就不存在什么包装类的概念了。

三、简单类型变量和包装类之间的相互转换

简单类型的变量转换为相应的包装类，可以利用包装类的构造函数。即：

Boolean(boolean value)、Character(char value)、Integer(int value)、Long(long value)、Float(float value)、Double(double value)

而 在各个包装类中，总有形为××Value()的方法，来得到其对应的简单类型数据。利用这种方法，也可以实现不同数值型变量间的转换，例如，对于一个双精 度实型类，intValue()可以得到其对应的整型变量，而 doubleValue()可以得到其对应的双精度实型变量。

四、String 类和其它数据类型的相互转换

对于上面的这些包装类，除了 Character 以外，都有可以直接使用字符串参数的构造函数，这也就使得我们将 String 类转换为这些数据类型变得相当之简单，即：

Boolean(String s)、Integer(String s)、Long(String s)、Float(String s)、Double(String s)

而将 String 类转换为 Date 类也可以使用这样的构造函数：Date(String s)

现 在我们还剩下一个字符型变量，事实上String 类可以理解为一个 char 型数组，因此我们可以在String 类中找到这样的方法来实现这种转换： charAt(int index)可以得到String 类中某一位置上的字符，toCharArray()更可以将整个String 类转换成一个 char 的数组。

对于所有的包装类都存在一个名为 toString()的方法可以将其转换成对应的String 类，而对于整型类和长整型类，还可 以使用 toBinaryString(int i)、toHexString(int i)、toOctalString(int i)分别以二进制、十 六进制和八进制的形式进行到 String 类的转换。

五、将字符型直接做为数值转换为其它数据类型

将字符型变量转换为数值型变量实 际上有两种对应关系，在我们在第一部分所说的那种转换中，实际上是将其转换成对应的 ASCII 码，但是我们有时还需要另一种转换关系，例如，‘1’就是指 的数值 1，而不是其 ASCII 码，对于这种转换，我们可以使用 Character 的 getNumericValue(char ch)方法。

六、Date 类与其它数据类型的相互转换

整型和 Date 类之间并不存在直接的对应关系，只是你可以使用 int 型为分别表示年、月、日、时、分、秒，这样就在两者之间建立了一个对应关系，在作这种转换时，你可以使用 Date 类构造函数的三种形式：

Date(int year, int month, int date)：以 int 型表示年、月、日

Date(int year, int month, int date, int hrs, int min)：以 int 型表示年、月、日、时、分

Date(int year, int month, int date, int hrs, int min, int sec)：以 int 型表示年、月、日、时、分、秒

在长整型和 Date 类之间有一个很有趣的对应关系，就是将一个时间表示为距离格林尼治标准时间 1970 年 1 月 1 日 0 时 0 分 0 秒的毫秒数。对于这种对应关系，Date 类也有其相应的构造函数：Date(long date)

获 取 Date 类中的年、月、日、时、分、秒以及星期你可以使用 Date 类的 getYear()、getMonth()、getDate()、 getHours()、getMinutes()、getSeconds()、getDay()方法，你也可以将其理解为将 Date 类转换成 int。

而 Date 类的 getTime()方法可以得到我们前面所说的一个时间对应的长整型数，与包装类一样，Date 类也有一个 toString()方法可以将其转换为 String 类。

在 Java 的数据类型转换中，你还有一些其它方法可用，但是，上面所介绍的这些方法对于你的实际编程已经足够了，不是吗？

2 JAVA 与 WEB

2.1 JMX 规范

2.1.1 JMX 概述

JMX--Java Management Extensions，即 Java 管理扩展,是一个为应用程序、设备、系统等植入管理功能的框架。JMX 可以跨越一系列异构操作系统平台、系统体系结构和网络传输协议，灵活的开发无缝集成的系统、网络和服务管理应用。

JMX 体系结构分为以下四个层次：

1)设备层（Instrumentation Level）：主要定义了信息模型。在JMX 中，各种管理对象以管理构件的形式存在，需要管理时，向 MBean 服务器进行注册。该层还定义了通知机制以及一些辅助元数据类。

2)代理层（Agent Level）：主要定义了各种服务以及通信模型。该层的核心是一个 MBean 服务器，所有的管理构件都需要向它注册，才能被管理。注册在 MBean 服务器上管理构件并不直接和远程应用程序进行通信，它们通过协议适配器和连接器进行通信。而协议适配器和连接器也以管理构件的形式向 MBean 服务器注册才能提供相应的服务。

3)分布服务层（Distributed Service Level）：主要定义了能对代理层进行操作的管理接口和构件，这样管理者就可以操作代理。然而，当前的 JMX 规范并没有给出这一层的具体规范。

4)附加管理协议 API：定义的 API 主要用来支持当前已经存在的网络管理协议，如 SNMP、TMN、CIM/WBEM 等。

2.1.2 设备层（Instrumentation Level）

该层定义了如何实现 JMX 管理资源的规范。一个 JMX 管理资源可以是一个 Java 应用、一个服务或一个设备，它们可以用 Java 开发，或者至少能用 Java 进行包装，并且能被置入 JMX 框架中，从而成为 JMX 的一个管理构件(Managed Bean)，简称 MBean。管理构件可以是标准的，也可以是动态的，标准的管理构件遵从 JavaBeans 构件的设计模式；动态的管理构件遵从特定的接口，提供了更大的灵活性。

该层还定义了通知机制以及实现管理构件的辅助元数据类。

2.1.2.1 管理构件（MBean）

在 JMX 规范中，管理构件定义如下：它是一个能代表管理资源的 Java 对象，遵从一定的设计模式，还需实现该规范定义的特定的接口。该定义了保证了所有的管理构件以一种标准的方式来表示被管理资源。

管理接口就是被管理资源暴露出的一些信息，通过对这些信息的修改就能控制被管理资源。一个管理构件的管理接口包括：
1)能被接触的属性值；
2)能够执行的操作；
3)能发出的通知事件；
4)管理构件的构建器。

管理构件通过公共的方法以及遵从特定的设计模式封装了属性和操作，以便暴露给管理应用程序。例如，一个只读属性在管理构件中只有 Get 方法，既有 Get 又有 Set 方法表示是一个可读写的属性。

其余的 JMX 的构件，例如 JMX 代理提供的各种服务，也是作为一个管理构件注册到代理中才能提供相应的服务。

JMX 对管理构件的存储位置没有任何限制，管理构件可以存储在运行 JMX 代理的 Java 虚拟机的类路径的任何位置，也可以从网络上的任何位置导入。

JMX 定义了四种管理构件：标准、动态、开放和模型管理构件。每一种管理构件可以根据不同的环境需要进行制定。

1.标准管理构件

标准管理构件的设计和实现是最简单的，它们的管理接口通过方法名来描述。标准管理构件的实现依靠一组命名规则，称之为设计模式。这些命名规则定义了属性和操作。检查标准管理构件接口和应用设计模式的过程被称为内省（Introspection）[22]。JMX 代理通过内省来查看每一个注册在 MBean 服务器上的管理构件的方法和超类，看它是否遵从一定设计模式，决定它是否代表了一个管理构件，并辨认出它的属性和操作。

2.动态管理构件

动态管理构件提供了更大的灵活性，它可以在运行期暴露自己的管理接口。它的实现是通过实现一个特定的接口 DynamicMBean（如下图）。

JMX 代理通过 getMBeanInfo 方法来获取该动态管理构件暴露的管理接口，该方法返回的对象是 MbeanInfo 类的实例，包含了属性和操作的签名。由于该方法的调用是发生在动态管理构件向 MBean 服务器注册以后，因此管理接口是在运行期获取的。不同于标准管理构件，JMX 代理不需要通过内省机制来确定动态管理构件的管理接口。由于 DynamicMBean 的接口是不变的，因此可以屏蔽实现细节。由于这种在运行期获取管理接口的特性，动态管理构件提供了更大的灵活性。

3.开放管理构件

开放管理构件是一种专门化的动态管理构件，其中所有的与该管理构件相关的参数、返回类型和属性都围绕一组预定义的数据类型（String、Integer、Float 等）来建立，并且通过一组特定的接口来进行自我描述。JMX 代理通过获得一个 OpenMBeanInfo 对象来获取开放管理构件的管理接口，OpenMBeanInfo 是 MbeanInfo 的子类。

4.模型管理构件

模型管理构件也是一种专门化的动态管理构件。它是预制的、通用的和动态的 MBean 类，已经包含了所有必要缺省行为的实现，并允许在运行时添加或覆盖需要定制的那些实现。JMX 规范规定该类必须实现为 javax.management.modelmbean.RequiredModelMBean，管理者要做的就是实例化该类，并配置该构件的默认行为并注册到 JMX 代理中，即可实现对资源的管理。JMX 代理通过获得一个 ModelMBeanInfo 对象来获取管理接口。

模型管理构件具有以下新的特点[23]：
1)持久性
定义了持久机制，可以利用 Java 的序列化或 JDBC 来存储模型 MBean 的状态。
2)通知和日志功能
能记录每一个发出的通知，并能自动发出属性变化通知。
3)属性值缓存
具有缓存属性值的能力。

2.1.2.2 通知模型

一个管理构件提供的管理接口允许代理对其管理资源进行控制和配置。然而，对管理复杂的分布式系统来说，这些接口只是提供了一部分功能。通常，管理应用程序需要对状态变化或者当特别情况发生变化时作出反映。

为此，JMX 定义了通知模型。通知模型仅仅涉及了在同一个 JMX 代理中的管理构件之间的事件传播。JMX 通知模型依靠以下几个部分：

1)Notification，一个通用的事件类型，该类标识事件的类型，可以被直接使用，也可以根据传递的事件的需要而被扩展。
2)NotificationListener 接口，接受通知的对象需实现此接口。
3)NotificationFilter 接口，作为通知过滤器的对象需实现此接口，为通知监听者提供了一个过滤通知的过滤器。
4)NotificationBroadcaster 接口，通知发送者需实现此接口，该接口允许希望得到通知的监听者注册。

发送一个通用类型的通知，任何一个监听者都会得到该通知。因此，监听者需提供过滤器来选择所需要接受的通知。

任何类型的管理构件，标准的或动态的，都可以作为一个通知发送者，也可以作为一个通知监听者，或两者都是。

2.1.2.3 辅助元数据类

辅助元数据类用来描述管理构件。辅助元数据类不仅被用来内省标准管理构件，也被动态管理构件用来进行自我描述。这些类根据属性、操作、构建器和通告描述了管理接口。JMX 代理通过这些元数据类管理所有管理构件，而不管这些管理构件的类型。

部分辅助元类如下：
1)MBeanInfo--包含了属性、操作、构建器和通知的信息。
2)MBeanFeatureInfo--为下面类的超类。
3)MBeanAttributeInfo--用来描述管理构件中的属性。
4)MBeanConstructorInfo--用来描述管理构件中的构建器。
5)MBeanOperationInfo--用来描述管理构件中的操作。
6)MBeanParameterInfo--用来描述管理构件操作或构建器的参数。
7)MBeanNotificationInfo--用来描述管理构件发出的通知。

2.1.3 代理层

代理层是一个运行在 Java 虚拟机上的管理实体，它活跃在管理资源和管理者之间，用来直接管理资源，并使这些资源可以被远程的管理程序所控制。代理层由一个 MBean 服务器和一系列处理被管理资源的服务所组成。下图表示了代理层的组成：

2.1.3.1 MBean 服务器

Mbean 服务器为代理层的核心，设备层的所有管理构件都在其注册，管理者只用通过它才能访问管理构件。管理构件可以通过以下三种方法实例化和注册：
1)通过另一个管理构件
2)管理代理本身

3)远程应用程序

注册一个管理构件时，必须提供一个唯一的对象名。管理应用程序用这个对象名进行标识管理构件并对其操作。这些操作包括：

1)发现管理构件的管理接口

2)读写属性值

3)执行管理构件中定义的操作

4)获得管理构件发出的通告

5)基于对象名和属性值来查询管理构件

2.1.3.2 协议适配器和连接器

MBean 服务器依赖于协议适配器和连接器来和运行该代理的 Java 虚拟机之外的管理应用程序进行通信。协议适配器通过特定的协议提供了一张注册在 MBean 服务器的管理构件的视图。例如，一个 HTML 适配器可以将所有注册过的管理构件显示在 Web 页面上。不同的协议，提供不同的视图。

连接器还必须提供管理应用一方的接口以使代理和管理应用程序进行通信，即针对不同的协议，连接器必须提供同样的远程接口来封装通信过程。当远程应用程序使用这个接口时，就可以通过网络透明的和代理进行交互，而忽略协议本身。

适配器和连接器使 MBean 服务器与管理应用程序能进行通信。因此，一个代理要被管理，它必须提供至少一个协议适配器或者连接器。面临多种管理应用时，代理可以包含各种不同的协议适配器和连接器。

当前已经实现和将要实现的协议适配器和连接器包括：

1)RMI 连接器

2)SNMP 协议适配器

3)IIOP 协议适配器

4)HTML 协议适配器

5)HTTP 连接器

2.1.3.3 代理服务

代理服务可以对注册的管理构件执行管理功能。通过引入智能管理，JMX 可以帮助我们建立强有力的管理解决方案。代理服务本身也是作为管理构件而存在，也可以被 MBean 服务器控制。

JMX 规范定义了代理服务有：

1)动态类装载--通过管理小程序服务可以获得并实例化新的类，还可以使位于网络上的类库本地化。

2)监视服务--监视管理构件的属性值变化，并将这些变化通知给所有的监听者。

3)时间服务--定时发送一个消息或作为一个调度器使用。

4)关系服务--定义并维持管理构件之间的相互关系。

1.动态类装载

动态类装载是通过 m-let（management applet）服务来实现的，它可以从网络上的任何 URL 处下载并实例化管理构件，然后向 MBean 服务器注册。在一个 M-let 服务过程中，首先是下载一个 m-let 文本文件，

该文件是XML 格式的文件，文件的内容标识了管理构件的所有信息，比如构件名称、在MBean 服务器中唯一标识该构件的对象名等。然后根据这个文件的内容，m-let 服务完成剩余的任务。下图例示这一过程：

2.监视服务

通过使用监视服务，管理构件的属性值就会被定期监视，从而保证始终处于一个特定的范围。当监视的属性值的变化超出了预期定义的范围，一个特定的通告就会发出。JMX 规范当前规定了三种监视器：

1)计数器监视器，监视计数器类型的属性值，通常为整型，且只能按一定规律递增。
2)度量监视器，监视度量类型的属性值，通常为实数，值能增能减。
3)字符串监视器，监视字符串类型的属性值。

每一个监视器都是作为一个标准管理构件存在的，需要提供服务时，可以由相应的管理构件或远程管理应用程序动态创建并配置注册使用。

下图例示了计数器监视器的使用情况：

3.时间服务

时间服务可以在制定的时间和日期发出通告，也可以定期的周期性的发出通告，依赖于管理应用程序的配置。时间服务也是一个管理构件，它能帮助管理应用程序建立一个可配置的备忘录，从而实现智能管理服务。

4.关系服务

JMX 规范定义了管理构件之间的关系模型。一个关系是用户定义的管理构件之间的N 维联系。

关系模型定义如下一些术语：

1)角色：就是是一个关系中的一类成员身份，它含有一个角色值。
2)角色信息：描述一个关系中的一个角色。
3)关系类型：由角色信息组成，作为创建和维持关系的模板。
4)关系：管理构件之间的当前联系，且必须满足一个关系类型的要求。
5)角色值：在一个关系中当前能满足给定角色的管理构件的列表。
6)关系服务：是一个管理构件，能接触和维持所有关系类型和关系实例之间的一致性。

在关系服务中，管理构件之间的关系由通过关系类型确定的关系实例来维护。仅仅只有注册到MBean服务器上并且能被对象名标识的管理构件才能成为一个关系的成员。关系服务从来就不直接操作它的成员--管理构件，为了方便查找它仅仅提供了对象名。

关系服务能锁定不合理关系类型的创建，同样，不合理的关系的创建也会被锁定。角色值的修正也要遵守一致性检查。

由于关系是定义在注册的管理构件之间的联系，所以当其中的管理构件卸载时，就会更改关系。关系服务会自动更改角色值。所有对关系实例的操作比如创建、更新、删除等都会使关系服务发出通告，通告会提供有关这次操作的信息。

JMX关系模型只能保证所有的管理构件满足它的设计角色，也就是说，不允许一个管理构件同时出现在许多关系中。

2.1.4 分布服务层

当前，SUN并没有给出这一层的具体规范，下面给出的只是一个简要描述。

该层规定了实现JMX应用管理平台的接口。这一层定义了能对代理层进行操作的管理接口和组件。这些组件能：
1)为管理应用程序提供一个接口，以便它通过一个连接器能透明和代理层或者JMX管理资源进行交互。
2)通过各种协议的映射（如SNMP、HTML等），提供了一个JMX代理和所有可管理组件的视图。
3)分布管理信息，以便构造一个分布式系统，也就是将高层管理平台的管理信息向其下众多的JMX代理发布。
4)收集多个JMX 代理端的管理信息并根据管理终端用户的需要筛选用户感兴趣的信息并形成逻辑视图送给相应的终端用户。
5)提供了安全保证。

通过管理应用层和另一管理代理和以及他的设备层的联合，就可以为我们提供一个完整的网络管理的解决方案。这个解决方案为我们带来了独一无二的一些优点：轻便、根据需要部署、动态服务、还有安全性。

2.1.5 附加管理协议API

该层提供了一些API来支持当前已经存在的一些管理协议。

这些附加的协议API并没有定义管理应用的功能，或者管理平台的体系结构，他们仅仅定义了标准的Java API和现存的网络管理技术通信，例如SNMP。

网络管理平台和应用的开发者可以用这些API来和他们的管理环境进行交互，并将这个交互过程封装在一个JMX管理资源中。例如，通过SNMP可以对一个运行有SNMP代理的交换机进行管理，并将这些管理接口封装成为一个管理构件。在动态网络管理中，可以随时更换这些管理构件以适应需求。

这些API可以帮组开发者根据最通常的工业标准来部署他们的管理平台和应用。新的网路管理的解决方案可以和现存的基础结构合为一体，这样，现存的网络管理也能很好的利用基于Java技术的网络管理应用，

这些API目前在JCP（Java Community Process）内作为独立的JSR（Java Specification Request）开发。

他们包括：
1)SNMP Manager API
2)CIM/WBEM manager and protocol API

2.1.6 JMX的当前实现及应用

自从SUN发布了JMX规范，许多大公司纷纷行动起来，实现规范或者实现相应的基于JMX的网络管理系统，下面列出了当前的主要实现及应用情况：

1)SUN为JMX规范了作出了相应的参考实现，并在此基础上开发了一个全新的用于网络管理的产品JDMK（Java动态管理工具集），其中定义了资源的开发过程和方法、动态JMX代理的实现、远程管理应用的实现。同时，JDMK也提供了一个完整的体系结构用来构造分布式的网络管理系统，并提供了多种协议适配器和连接器，如SNMP协议适配器、HTML协议适配器、HTTP连接器、RMI连接器。
2)IBM Tivoli实现了JMX规范的产品为TivoliJMX，它为JAVA管理应用程序和网络提供了架构、设计模式、一些API集和一些服务。
3)Adventnet开发的关于JMX的产品为AdventNet Agent Toolkit，它使得定义新的SNMP MIB、开发JMX和Java SNMP Agent的过程自动化。
4)JBoss实现的J2EE应用服务器以JMX为微内核，各个模块以管理构件的形式提供相应的服务。
5)BEA的Weblogic应用服务器也将JMX技术作为自己的管理基础。
6)金蝶的Apusic也是一个以JMX为内核开发出的J2EE应用服务器。

2.1.7 小结

本文详细介绍了JMX规范。JMX体系结构分为四层，即设备层、代理层、分布服务层和附加协议API.但SUN当前只实现了前两层的具体规范，其余的规范还在制定当中。JMX代理要和远程应用程序通信，需要提供至少一个连接器和协议适配器。

2.2 应用 JMX 最佳实践

构建 Java 应用程序需要许多复杂的分布式组件。现今，几乎所有的应用程序都要连接到遗留系统或其他 IT 资源。这些应用程序的分布式本质，为 IT 提出了一个难以置信的挑战，即一旦开发出一个应用程序，就要担负起维护该应用程序及其所有相关程序的重担。

由于企业开始采用面向服务的体系结构（Service-Oriented Architectures ， SOA ），问题变得进一步复杂化了。SOA 引入了一种设计风格，即把应用程序公开为，以松散耦合方式连接的服务。在 SOA 方法中，通常使用不同的编程语言和平台，来开发应用程序组件。在某些情况下，客户端和服务提供者之间的连接，直到运行时都无法确定。

寄希望于利用 SOA 的企业，现在需要一种更好的方式，来管理它们的分布式应用程序和服务。对处理现今应用程序异构和动态的本质来说，使用既定的底层管理技术（如 SNMP ）已经不能满足需求。管理需要面向服务的风格—— META Group 将此称为面向服务的管理体系结构（ Service-Oriented Management Architecture ）或（SOMA）.（ 参见 参考资料 ）

SOMA 允许异构的托管系统和管理应用程序和平共处。面向服务的管理风格，可以消除现存的人工屏障，这些屏障，是由于依赖特定平台上的特定管理 API 而造成的。让我们进一步考察，如何通过 Java 管理扩展（ Java Management Extension ， JMX ） API ，而在 Java 中实现 SOMA 。

除了支持在管理产品之间进行更好的集成之外， SOMA 还使得开发自定义管理应用程序变得更加轻松。尽管企业通常依赖于开箱即用的管理解决方案，但也需要构建用于监控管理数据的，特定子集的自定义工具板。 SOA 风格的管理将使一个团队，能够使用反馈自 Web 服务的数据，来快速构建管理应用程序。 Web 服务是 SOA 实现中使用的常见技术。

SOMA 表述

区分管理接口和管理实现是相当重要的。 SOMA 中提出的设计原则，主要与托管系统和被托管应用程序之间的接口有关。关于在公开一项托管资源的过程中，所使用的底层实现， SOMA 没有任何表述。一个应用程序可能需要使用特定的 API ，并借助不同的管理接口进行公开。例如， Java 开发人员可以使用 JMX ，在他们的应用程序中增加易管理性。 JMX 使开发人员可以在他们的应用程序中使用 JMX MBean ，这样 JMX Mbean 服务器就可以发现并访问这些托管资源。

JMX 不仅仅是一个编程 API ，它还定义了一个包括监控和管理服务的体系结构，以及一个包括连接器和适配器的分布层。开发人员可以使用标准的 RMI 连接器，来外部公开管理接口。然而，如果您希望公开更多面向服务的管理接口，那么使用 RMI 连接器并非最佳方法。

现在已经提出了几个用于解决 SOMA 问题的标准。 Hewlett-Packard (HP) 所进行的早期工作，导致了第一批基于 SOA 的管理标准的出现，其中之一是： Web 服务管理框架（ Web Services Management Framework ， WSMF ）。 HP 把 WSMF 提供给了 Web Services Distributed Management (WSDM) ， WSDM 是一个 OASIS 技术委员会，创建它的目的是，为用于可管理资源的 Web 服务接口定义规范。

我们相信，业界会为 JMX 和其他基于 SOA 的管理标准开发 WSDM 协议适配器的。一般性的概念是， WSDM 协议适配器将支持 WSDM 客户端或管理应用程序，来使用 Web 服务协议与 JMX Mbean 连接（参见 图 1 ）。

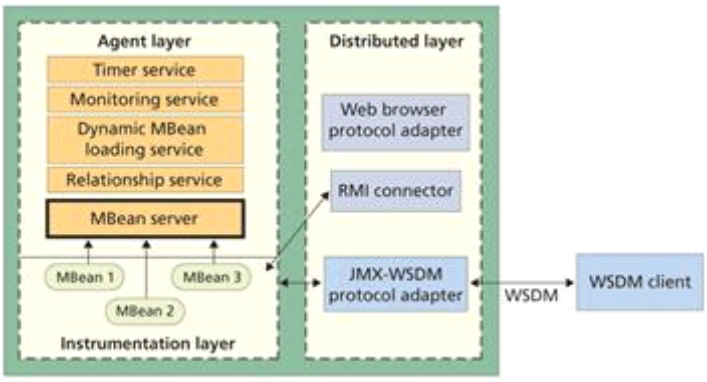


图 1. 从 JMX 到 WSDM 的连通性

通常， JMX-WSDM 协议适配器会使用 Web 服务协议，来支持 WSDM 客户端或管理应用程序，与 JMX Mbean 连接。

如果开发人员希望通过 JMX 来实现易管理性，那么他们是如何实现 SOMA 的呢？幸运的是，有许多良好的证据点，演示了从 JMX 到 SOA 管理风格的映射。例如， HP 发布了 HP OpenView Smart Plug-In (SPI) ，它可以管理和监控 WebLogic Integration 中的业务流程（参见参考资料） SPI 不得不包括由 BEA Weblogic Integration 公开的 JMX Mbean 和基于 WSMF 的接口之间的一座桥梁。

从我们自己的经验出发，我们已经找出了，在把 JMX 映射为 Web 服务过程中的几处障碍。要想克服这些难题，我们需要考虑服务和松散耦合体系结构方面的问题。我们必须应用一些设计原则，比如简单性、模块性和互操作性。

例如，假定一个典型的 JMX Mbean 与客户端有着十分紧密的耦合，公开了有关可管理资源的许多底层细节。JMX Mbean 可能映射为 Java 类或 Enterprise JavaBean (EJB) 。我们不想把这个 Mbean 映射为单个的 Web 服务端点。相反， SOA 需要一个用于公开易管理性的、更加粗粒度的方法。

Java 和 Web 服务使用的数据类型之间的互操作性，也是必须解决的一个主要难题。我们发现， JMX 接口使用的数据类型，不能自动转换为 Web 服务的数据类型。另外， JMX Mbean 可以向 Java 对象返回一个远程引用，这在 Web 服务世界中没有相对应的部分。

管理挑战

我们研究 JMX 定义的编程模型和体系结构时，发现了编程接口和 管理模型 之间的明显区别。 JMX 提供了一个非常灵活且功能强大的 API ，用于实现应用程序的易管理性，但是单独使用 JMX 无法定义或利用任何特定的管理模型。

建立良好的管理模型，对于功能丰富的管理应用程序来说是必不可少的。如果没有这种管理模型，跨应用程序一致地提取和处理管理数据，就将成为一大挑战。

最后，在 JMX 中尚未完全支持的新兴 Web 服务管理规范中，定义了几种重要的管理抽象。例如， WSDM Management Using Web Services (MUWS) 包括一个线级规范，用于基于 Web 服务技术的交换管理信息。这个规范对 Metrics, ResourceState 和 Relationships 的管理功能进行了建模（参见 图 2 ）。

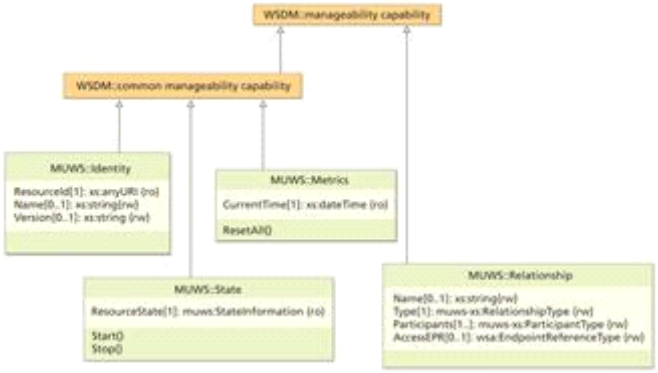


图 2. 管理功能

这个规范对 Metrics, ResourceState 和 Relationships 的管理功能进行了建模。

JMX 无法充分地对许多此类功能建模。例如，尽管 JMX 为 Mbean 之中的关系定义了一个 Relation 服务，但它使用起来还是相当的复杂，而且很少在实践中实现。 JMX 也不直接支持像 Metrics 和 State 这样的功能。例如， JMX 中并没有预定义的类，来代表不同类别的量度。

WSDM 的这些缺点，使从 JMX 到 SOA 的易管理性接口的自动转换，成为了一个大大的难题。为了帮助您克服一部分此类难题，我们给出了一系列的 5 个最佳实践，用于为 WSDM 作准备的 JMX 开发。我们相信，结合这些实践，将会开发出更加易于管理的应用程序，而与您是否计划使用 WSDM 无关。

最佳实践 1 ：从管理模型开始。 管理模型 定义了要交换的管理信息，以及这些信息的底层语义。使用管理模型，对于确保易于发现、标识和监控托管资源的健康和可用性是必不可少的。管理模型对于确保，可以在运行时统一地配置和控制应用程序来说，同样很重要。

应用程序是不会孤立存在的，它们和其他应用程序、系统以及网络组件都有着相关性。即使主要考虑一个应用程序时，应用程序的易管理性也必须把这些相关性考虑在内。管理模型应该捕捉这些相关性和关系。

创建一个管理模型，要求您在软件生命周期的早期，考虑到易管理性的需求。通常，开发人员总是在事后才想起易管理性

的问题。为了在构建 SOMA 方面获得真正的成功，您必须从一开始就考虑易管理性和管理模型。事实上，无论您使用何种技术来公开易管理性，一个优秀的管理模型总是会为您带来好处的。在开发管理模型的过程中，您应该询问一些问题。什么是托管 资源 ？什么是托管资源的 状态 ，如何去控制它？托管资源之间存在 关系 吗？您需要跟踪什么特定的业务或性能 量度 ？应该公开什么样的额外管理 属性 和 操作 ？托管资源关心的是什么事件和消息 通知 ？

定义模型

理解这些问题的答案，能够帮助您定义一个可以跨 IT 基础架构利用的管理模型。您还应该从操作人员的角度考虑这个管理模型，因为操作人员必须在部署应用程序之后对其进行管理。

记住，应用程序公开的管理模型和它的内部结构不是一回事。前者的目标是允许外部管理系统有效地监控、配置和控制应用程序，而后者的目标则是实现业务功能。

例如，我们设想一台 Web 应用服务器，它允许部署多个 Web 应用程序或 Web 模块，每个 Web 模块由一个或多个 servlet 组成（参见 图 3 中高度简化的系统管理模型）。

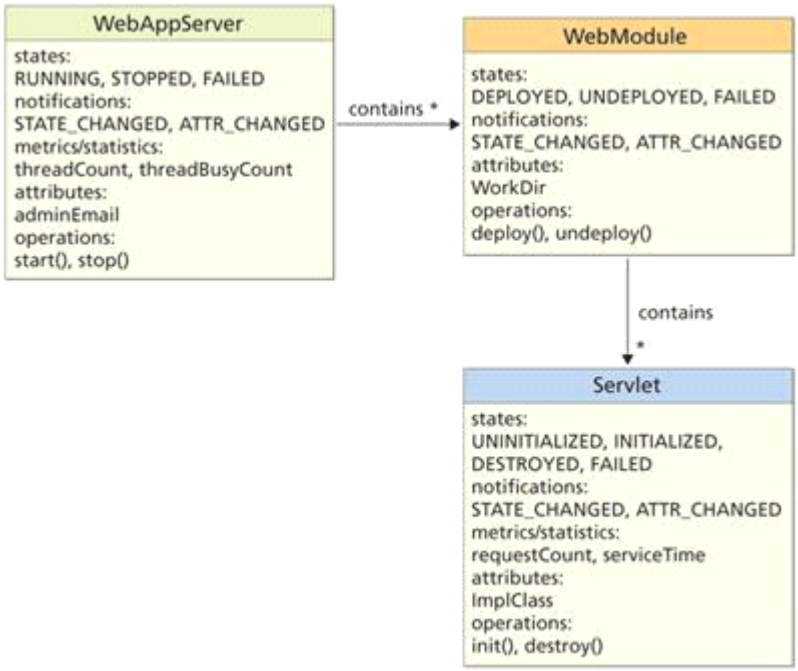


图 3. 管理模型

在高度简化的系统管理模型中，一台 Web 应用服务器允许部署多个 Web 应用程序或 Web 模块，而每个 Web 模块由一个或多个 servlet 组成。

这个模型为每项可管理资源，都定义了状态、通知、量度、属性和操作。我们将利用这个过分简化的模型，围绕在 SOMA 中使用 JMX 实现易管理性，来应用另外的最佳实践。另外，我们还开发了 实现这个模型的完整源代码 。

最佳实践 2 ：设计互操作性。 拥有一个定义良好的管理模型，是走向 SOMA 的第一步。 Java 社区已经意识到了这种需要，并且已经通过 JSR 77 和 174 为 J2SE 和 J2EE 平台定义了管理模型。

JSR 174 现在是 J2SE 5.0 规范的一部分，它提供了一种为 Java 虚拟机（ Java Virtual Machine ， JVM ）公开管理模型的方式。它引入了平台 MXBean 的概念，这个概念是用于建模，代表特定的 JVM 监控指示器的，专门 JMX MBean .这些 MXBean 可以用于监控内存使用、线程争用问题、类装载行为和垃圾收集频率。

所有 MXBean 都被实现为 OpenMBean ，这是一个提供增强级别互操作性的 JMX Mbean 类型。OpenMBean 限制了 Mbean 接

口中某些数据类型的使用。这种限制最小化了客户端对于访问 Mbean 的需求，进一步确保了可以使用 XML 轻松访问和操作这些 MBean ，同时不用求助于特定的封送和解除封送逻辑。

所有 MXBean 操作和属性必须遵循特定的数据类型集合，叫做开放类型（ open type ），它包括原始类型 （ int, long 和 boolean ）、枚举、 CompositeData 类型和 Map and List 类型。 清单 1 显示了一个遵从这些数据类型要求的 ServletMXBean 接口的例子。

注意， J2SE 5 天生就不支持从用户定义的 MXBean 到 Open Mbean 的映射，这意味着您无法引入您自己的 MXBean ，并指望它的行为像平台 MXBean 一样。可以开发一个一般类集合，来支持把用户定义的 MXBean 作为 Open Mbean 注册到 MBeanServer ，而且我们希望这些类将通过 JCP 过程变为可用。

即使您没有直接使用平台 MXBean ，您仍然可以把您的 JMX MXBean 设计为符合 Open Mbean 的模型。最终结果是一个互操作性更强，并满足 SOMA 需要的 JMX 接口。

补充 JMX

最佳实践 3 ：利用 J2EE 管理机制。JSR 77 提出了一个为 J2EE 平台公开管理信息的管理模型。模型支持管理大量 J2EE 资源的能力，包括 EJB 、 Web 容器、 JMS 和 JDBC 连接。该模型可以用于收集、监控和控制有关应用服务器的运行时信息。这个通用模型支持使用管理工具，轻松地管理多个 J2EE 的厂商实现。

JSR 77 定义了许多抽象，在处理性能统计信息、应用程序状态和关系方面，对 JMX 进行了补充。它定义了 Statistic 接口，用于对 J2EE 组件的性能数据建模。例如，模型定义了一个 EJBStats 接口，为所有的 EJB 组件指定了统计信息。这个接口公开了基本的 CountStatistics ，用于跟踪，创建和删除的对象的数目：

```
public interface EJBStats extends

Stats {

CountStatistic getCreateCount();

CountStatistic getRemoveCount();
```

可以通过 StateManageable 接口来管理一个对象的状态。您可以查询资源的状态，并启动和停止特定的组件。另外，JSR 77 为模型中的代表关系定义了一个基本的惯例。可以定义包容关系，在这种关系中，特定的容器可以维护一个托管对象的数组。

我们可以将多个此类设计原则，应用到我们的管理模型例子（参见 图 4）。我们说明了如何增强 MXBean ，来支持状态管理、事件、规格和包容的能力。参考前面 清单 1 中的源代码，您会发现 Servlet MXBean 接口是按照前面描述的模型，来定义 Servlet Managed 对象的易管理性接口的。

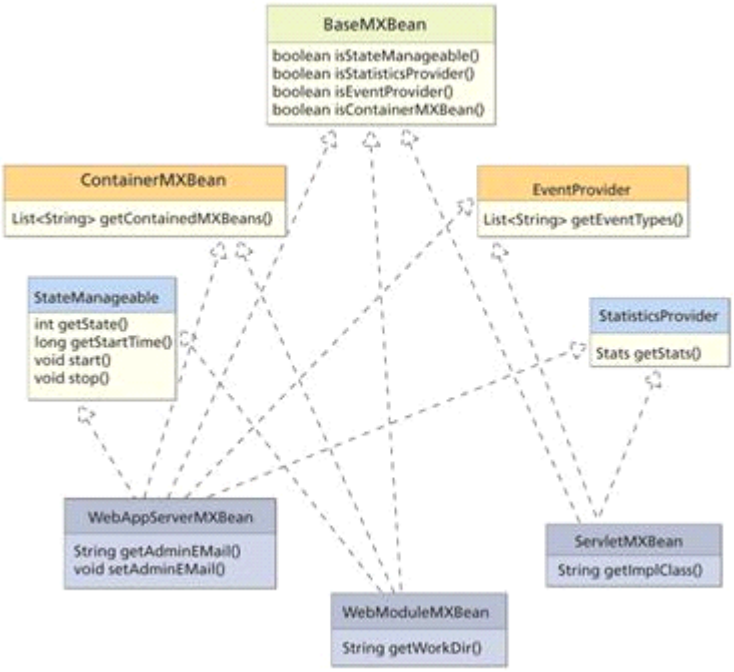


图 4. MXBean 接口

在我们的管理模型例子中，我们可以看到如何增强 MXBean ，来支持状态管理、事件、量度和包容功能。

ServletMXBean 及相关接口

清单 1. 这个 ServletMXBean 接口符合开放类型，开放类型包括原始类型（ int, long 和 boolean ）以及它们的包装类、枚举、 CompositeData 类型和 Map and List 类型。

```
public interface ServletMXBean {

public boolean isStateManageable();

public boolean isStatisticsProvider();

public boolean isEventProvider();

public boolean isContainerMXBean();

public List getEventTypes();

public Stats getStats();

public String getImplClass();

}

public interface Stats {

public Statistic getStatistic(String name);

public Map<String, Statistic> getStatistics();

}

public interface Statistic {
```

```
public String getName();

public String getUnit();

public String getDescription();

public long getStartTime();

public long getLastSampleTime();

}
```

使用量度、统计信息、状态管理和关系，是 SOMA 的重要方面。没有这些功能，提供一个用于监控和管理 Java 应用程序的一致管理接口，就成为了一件困难的事情。在您自己的开发项目中，要注意使用一些这样的 J2EE 管理机制。

最佳实践 4 ：使用 JMX 通知进行通信。 JMX 为托管对象定义了一个事件模型，您可以使用这个模型来接收或生成事件。事件机制对于处理，由托管资源生成的关键事件，以及监控状态变化，是必不可少的。事件还可以用于为托管资源去监控服务级别的违规。 JMX 通知机制允许 Mbean 发送通知给其他 Mbean 或其他管理应用程序。以通知为中心的设计原则，可以应用于我们的示例管理模型。例如，我们可能需要，在某个属性出现变化或者加入新的模块时，生成通知。

清单 2 显示了 WebAppServerMXBean 类的实现，其中包括给应用服务添加新的 WebModule 时，要调用的 addWebModule() 方法。注意 Notification 对象的创建和对 sendNotification() 方法的调用。这个方法是从基类继承而来的，使用一个实用程序类（ javax.management.NotificationBroadcasterSupport ）来发送通知。

生成 JMX 通知

清单 2. WebAppServerMXBean 类的这个实现，包括 给应用服务添加新的 WebModule 时，要调用的 addWebModule() 方法。

```
public class WebAppServerMXBeanImpl

extends NEStandardMBean

implements WebAppServerMXBean {

// Implementation details removed — see

// full source code

private ArrayList<String> webModules =

new ArrayList<String>();

private String adminEMail;

public void setAdminEMail(String adminEMail){

String oldEMail = this.adminEMail;

this.adminEMail = adminEMail;

// create and send a notification.

Notification notif = new Notification(

"jmxbp.attribute_changed", "hello", 1,

System.currentTimeMillis(),
```

```
"attribute: AdminEMail, old value: " +

oldEMail + ", new value: " + adminEMail);

sendNotification(notif);

}

// non-MBean methods.

public void addWebModule(String objectName){

webModules.add(objectName);

// create and send a notification.

Notification notif = new Notification(

"jmxbp.child_added", "hello", 1,

System.currentTimeMillis(), "test message");

sendNotification(notif);

}

}
```

使用 JMX 通知是 SOMA 的关键设计原则之一。在面向服务的设计中，您需要获得托管资源的能力，以便与管理系统进行简易的通信。 JMX 通知允许您，更容易地生成要处理的应用程序事件。另外，正确使用通知机制，可以减轻潜在的性能问题，具体方法是，让管理系统频繁地轮询应用程序，看看有无变化。

聚会时间

最佳实践 5 ：分离管理和业务关系。有一个良好的设计实践是， 把需要托管的业务对象，从管理它们的管理接口中分离出去。许多当前的 JMX 最佳实践，并没有使这种分离表现为显式的，这就导致了把业务层和 JMX 接口相结合的应用程序的开发。

把这两个方面相混合，可能会带来一些严重的后果，比如需要为一个类的每个实例注册一个 JMX MBean 。另一方面，分离管理和业务，允许管理接口独立于所定义的业务对象而变化。

您可以应用一系列非常确实的设计模式，来辅助对您管理层的建模工作（ 参见参考资料 ）。例如， SeparateMBean 模式（ 参见 图 5 ）描述了，与必须托管的，从业务对象分离的单独对象的， MBean 的创建。创建业务对象时，它们要维持一个引用，引用了充当它们管理方的，单独的 MBean 对象。

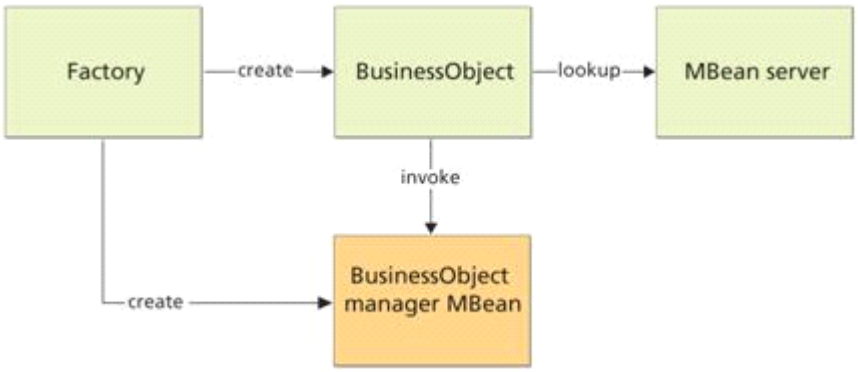


图 5. SeparateMBean 设计模式

SeparateMBean 模式描述的是，与必须托管的业务对象分离的，单独对象的 Mbean 的创建。每个业务对象维护一个充当它们管理方的，单独 MBean 对象的引用。

尽管我们的例子没有包括 ServletMXBean 的真正实现，但实际上是可以使用 SeparateMBean 模式的，在该模式中，MXBean 是由 Servlet 对象更新的。

在 Web 服务管理接口中获得正确的粒度级别时，其他设计模式可能会对您有所帮助。因为每个 Mbean 都有与注册和查找相关的开销，设计过多的 Mbean 会影响性能。通过使用级联模式，您可以使这种性能影响变为最小，并向管理应用程序提供更加粗粒度的接口。

还有一件事情也很重要，即，要把 JMX 管理接口看作是管理数据的库，而不是管理逻辑的库。当您需要监控应用程序的阈值违规时，应把这种逻辑放入管理系统，而不是放入托管应用程序中。这样做可以提供更多的灵活性，以便在部署应用程序之后，更新服务级别的监控。

随着所部署体系结构的复杂程度和分布式程度的增加， IT 将需要一个更加灵活而开放的体系结构，来管理 SOA 组件。诸如 JMX 这样的管理技术，为实现 J2EE 应用程序的易管理性提供了具体的解决方案，但是它们并不能满足需要。

为了实现 SOMA ，您必须将 SOA 原则应用于管理，这需要考虑每个应用程序，对易管理性的要求。然后，管理功能就组成了资源的管理模型。拥有一组通用管理模型，是构建跨厂商实现的，标准管理接口的关键。

在 Web 服务技术领域的经验中，强调设计中互操作性的重要性，而且在您使用 JMX 技术时，应该注意选择正确的数据类型，来缓解这些问题。另外，您可以期待 JSR 77 中提供的功能，能够帮助您，在您的应用程序中构建管理功能。然而，使用 JMX 开发您的应用程序，需要花费更多的工夫。它要求围绕您的易管理性需求，进行仔细地规划，还要求仔细地分离管理和业务问题，从而使您的设计具有最大的灵活性。

最近引入的 JSR 255 把目标定为更新 JMX 和 JMX Remote API ，从而从可用性的角度改善现有的接口（ 参见资源 ）。我们希望，经过修订的规范可以结合这里给出的最佳实践，或者能够让使用这些最佳实践变得更加容易。我们还要清楚如何努力，去引入单独的 JSR ，以探讨，用于搭建从 JMX 到 Web 服务标准（比如 SOAP ）的桥梁的可用方法。

通过结合一些方法，以及这里讨论的 JMX 技术，您可以更加高效地公开使用 JMX 的、带有 WSDM 的应用程序或者其他管理接口。最后，您将能够设计更多管理感知的、能够满足业务需求的应用程序。

想要了解关于 JMX 技术的更多信息，您可以访问 HP 的 Dev Resource Central ，在那里，您可以找到有关这个主题的大量技术资源、文章和指南（参见 参考资料 ）。

2.3 Java/J2EE 中文问题终极解决之道

Java 中文问题一直困扰着很多初学者，如果了解了Java 系统的中文问题原理，我们就可以对中文问题能够采取根本的解决之

道。

最古老的解决方案是使用 String 的字节码转换，这种方案问题是不方便，我们需要破坏对象封装性，进行字节码转换。

还有一种方式是对J2EE容器进行编码设置，如果J2EE应用系统脱离该容器，则会发生乱码，而且指定容器配置不符合J2EE应用和容器分离的原则。

在 Java 内部运算中，涉及到的所有字符串都会被转化为 UTF-8 编码来进行运算。那么，在被Java 转化之前，字符串是什么样的字符集？ Java 总是根据操作系统的默认编码字符集来决定字符串的初始编码，而且 Java 系统的输入和输出的都是采取操作系统的默认编码。

因此，如果能统一 Java 系统的输入、输出和操作系统 3 者的编码字符集合，将能够使 Java 系统正确处理和显示汉字。这是处理 Java 系统汉字的一个原则，但是在实际项目中，能够正确抓住和控制住 Java 系统的输入和输出部分是比较难的。J2EE 中，由于涉及到外部浏览器和数据库等，所以中文问题乱码显得非常突出。

J2EE 应用程序是运行在 J2EE 容器中。在这个系统中，输入途径有很多种：一种是通过页面表单打包成请求（request）发往服务器的；第二种是通过数据库读入；还有第3 种输入比较复杂，JSP 在第一次运行时总是被编译成 Servlet，JSP 中常常包含中文字符，那么编译使用 javac 时，Java 将根据默认的操作系统编码作为初始编码。除非特别指定，如在Jbuilder/eclipse 中可以指定默认的字符集。

输出途径也有几种：第一种是JSP 页面的输出。由于JSP 页面已经被编译成 Servlet，那么在输出时，也将根据操作系统的默认编码来选择输出编码，除非指定输出编码方式；还有输出途径是数据库，将字符串输出到数据库。

由此看来，一个J2EE 系统的输入输出是非常复杂，而且是动态变化的，而 Java 是跨平台运行的，在实际编译和运行中，都可能涉及到不同的操作系统，如果任由Java 自由根据操作系统来决定输入输出的编码字符集，这将不可控制地出现乱码。

正是由于 Java 的跨平台特性，使得字符集问题必须由具体系统来统一解决，所以在一个 Java 应用系统中，解决中文乱码的根本办法是明确指定整个应用系统统一字符集。

指定统一字符集时，到底是指定 ISO8859_1 、GBK 还是UTF-8 呢？

（1）如统一指定为ISO8859_1，因为目前大多数软件都是西方人编制的，他们默认的字符集就是 ISO8859_1，包括操作系统 Linux和数据库MySQL等。这样，如果指定 Jive 统一编码为 ISO8859_1，那么就有下面 3 个环节必须把握：

开发和编译代码时指定字符集为 ISO8859_1。

运行操作系统的默认编码必须是 ISO8859_1，如 Linux。

在 JSP 头部声明：。

（2）如果统一指定为 GBK 中文字符集，上述3 个环节同样需要做到，不同的是只能运行在默认编码为 GBK 的操作系统，如中文 Windows。

统一编码为 ISO8859_1 和 GBK 虽然带来编制代码的方便，但是各自只能在相应的操作系统上运行。但是也破坏了 Java 跨

平台运行的优越性，只在一定范围内行得通。例如，为了使得GBK编码在linux上运行，设置Linux编码为GBK。

那么有没有一种除了应用系统以外不需要进行任何附加设置的中文编码根本解决方案呢？

将 Java/J2EE 系统的统一编码定义为 UTF-8。UTF-8 编码是一种兼容所有语言的编码方式，惟一比较麻烦的就是要找到应用系统的所有出入口，然后使用UTF-8 去“结扎”它。

一个 J2EE 应用系统需要做下列几步工作：

开发和编译代码时指定字符集为 UTF-8。JBuilder 和 Eclipse 都可以在项目属性中设置。使用过滤器，如果所有请求都经过一个 Servlet控制分配器，那么使用 Servlet 的 filter 执行语句，将所有来自浏览器的请求（request）转换为 UTF-8，因为浏览器发过来的请求包根据浏览器所在的操作系统编码，可能是各种形式编码。关键一句：

request.setCharacterEncoding("UTF-8")。

网上有此 filter 的源码，Jdon框架源码中 com.jdon.util.SetCharacterEncodingFilter

需要配置web.xml 激活该Filter。

在JSP头部声明：。

在Jsp 的 html 代码中，声明 UTF-8:

设定数据库连接方式是 UTF-8。例如连接 MYSQL 时配置 URL 如下：

jdbc:mysql://localhost:3306/test?useUnicode=true&characterEncoding=UTF-8

注意，上述写法是 JBoss 的 mysql-ds.xml 写法，多亏网友提示，在tomcat 中&要写成&即可。一般其他数据库都可以通过管理设置设定 UTF-8

其他和外界交互时能够设定编码时就设定UTF-8，例如读取文件，操作XML 等。

笔者以前在 Jsp/Servlet 时就采取这个原则，后来使用 Struts、Tapestry、EJB、Hibernate、Jdon等框架时，从未被乱码困扰过，可以说适合各种架构。希望本方案供更多初学者分享，减少 Java/J2EE 的第一个拦路虎，也避免因为采取一些临时解决方案，导致中文问题一直出现在新的技术架构中。

2.4 Java Web 应用中的任务调度

为何需要任务调度？

在 web应用中，大多数任务是以一种"防止用户长时间等待"的方式完成的。在Google搜索这样的例子中，减少等待时间对用户体验来说至关重要。异步任务的一种解决方案是在用户提交后生成一个线程（来处理异步任务），但这也不能解决那些需要以一定时间间隔重复运行任务、或在每天的指定时间运行任务的情况。

让我们从一个数据库报表的例子来看看任务调度能如何帮助改善系统设计。报表可能是错综复杂的，这取决于用户所需数据的种类，以及是否需要从一个或多个数据库收集大量数据。用户可能需要很长时间来运行这样的"按需"报表。因此，我们向这个报表示例中添加任务调度机制，以便用户可以安排在任何他们需要的时间生成报表，并以PDF或其他格式在email中发送。用户可以让报表在每天的凌晨2:22，系统正处于低负荷时运行；也可以选择只在特定时间运行一次。通过在报表应用中加入任务调度，我们可以为产品添加一项有用的功能，并改善用户体验。

幸运的是，有一个强大的开源解决方案可以让我们以标准的方式在web应用（或任何Java应用）中实施任务调度。以下示例展示了在web应用中，如何使用Quartz来创建一个任务调度框架。这个示例还使用了Struts Action framework 插件，以便在web应用启动时初始化任务调度机制。Struts是最常见的MVC框架，为大多数开发人员所熟悉。当然除此之外还有许多框架可以协助在web应用中实现MVC模式。

启动时初始化任务调度器

我们首先要做的是建立一个Struts插件，让它在容器启动时创建我们的任务调度器。在以下例子中，我们选择Tomcat作为web应用容器，不过这些示例在其他容器中也应当可以运行。我们要创建一个Struts插件类，并在struts-config.xml中加入几行代码以使之可以工作。

这个插件有两个可配置的初始化参数：startOnLoad指定是否要在容器启动时立即启动任务调度器，而 startupDelay指定启动任务调度器之前的等待时间。启动延时很有用，因为我们可能需要首先执行一些更重要的初始化步骤。此外还可以使用listener机制，以更复杂的方式来通知SchedulerPlugIn何时启动Quartz Scheduler。

```
<plug-in className="SchedulerPlugIn">
   <set-property property="startOnLoad" value="false" />
   <set-property property="startupDelay" value="0" />
</plug-in>
```

我们要创建的是一个实现Struts插件接口org.apache.struts.action.PlugIn的单子类SchedulerPlugIn。Struts会按照配置文件中出现的顺序初始化各个插件。要特别注意的是init()方法中的代码，在此我们初始化了所需的Quartz对象，并得到 Scheduler。我们的任务信息就要提交到此org.quartz.Scheduler对象，后者将在随后讨论。Scheduler对象由Quartz servlet根据其配置初始化，就像Struts初始化它的ActionServlet类一样。让我们来看init()方法：

```
public void init(ActionServlet actionServlet,
                 ModuleConfig moduleConfig) {

  System.out.println("Initializing Scheduler PlugIn for Jobs!");
  // Retrieve the ServletContext
// 获取ServletContext
  ServletContext ctx = actionServlet.getServletContext();
  // The Quartz Scheduler
  // Quartz Scheduler对象
  Scheduler scheduler = null;

  // Retrieve the factory from the ServletContext.
  // It will be put there by the Quartz Servlet
  // 从ServletContext取得由Quartz Servlet放置在此的factory对象。
  StdSchedulerFactory factory =  (StdSchedulerFactory)
      ctx.getAttribute(QuartzInitializerServlet.QUARTZ_FACTORY_KEY);

  try{
        // Retrieve the scheduler from the factory
        // 从factory取得scheduler
        scheduler = factory.getScheduler();

        // Start the scheduler in case, it isn't started yet
        // 如果scheduler尚未启动，则启动它
```

```
        if (m_startOnLoad != null &&
            m_startOnLoad.equals(Boolean.TRUE.toString())){
            System.out.println("Scheduler Will start in " +
            m_startupDelayString + " milliseconds!");
            //wait the specified amount of time before
            // starting the process.
            // 在启动之前等待指定长度的时间
            Thread delayedScheduler =
                new Thread(new DelayedSchedulerStarted (
                                scheduler, m_startupDelay));
            //give the scheduler a name. All good code needs a name
            //给任务调度器命名。好的代码总该有名字!
            delayedScheduler.setName("Delayed_Scheduler");
            //Start out scheduler
            //启动任务调度器
            delayedScheduler.start();
        }
    } catch (Exception e){
       e.printStackTrace();
    }
      sm_scheduler = scheduler;
}
```

配置过程的第二步是在web.xml中加入用来初始化Quartz servlet(org.quartz.ee.servlet.QuartzInitializerServlet)的内容，因为需要它将SchedulerFactory添加到ServletContext中，以便在我们的Struts插件中可以访问。SchedulerFactory就是我们在Struts插件中获得Scheduler对象的来源。除了struts-config.xml 和 web.xml之外，还要在web应用的classes目录下放置一个quartz.properties文件。此文件的位置也可以在web.xml中作为QuartzInitializerServlet的启动参数来指定。

```
  <servlet>

     <servlet-name>QuartzInitializer</servlet-name>
     <display-name>Quartz Initializer Servlet</display-name>

  <servlet-class>
    org.quartz.ee.servlet.QuartzInitializerServlet
  </servlet-class>
  <load-on-startup>1</load-on-startup>
  <init-param>
    <param-name>shutdown-on-unload</param-name>
    <param-value>true</param-value>
  </init-param>

  <init-param>
```

```
    <param-name>start-scheduler-on-load</param-name>
    <param-value>false</param-value>
    </init-param>

 </servlet>
```

这里其实完全可以不使用Struts和SchedulerPlugIn，但如果将来决定要以其它的任务调度框架替换Quartz的话，额外的抽象层就很有用了。长远看来，让一切保持松散耦合总会使工作变得容易些。如果你使用其它MVC框架，也可以用SchedulerPlugIn.init()方法中的代码达到同样的效果。此外，还可以用Servlet 2.3规范中的ServletContextListener来实现同样的初始化过程。

到此为止web应用已配置完毕，我们可以创建一个.war文件并部署到服务器上，从控制台观察SchedulerPlugIn的输出信息。然而在此之前，让我们先看看如何向任务调度器提交一项任务。

我们可以从web应用中的任何类访问SchedulerPlugIn的唯一实例，并调度一些要执行的工作。首先需要一个Trigger（触发器）对象来告诉任务何时运行、每隔多久运行一次。Quartz支持多种触发器，在这个例子中我们使用CronTrigger。

```
Trigger trigger = new CronTrigger("trigger1", "group1");
trigger.setCronExpression("0 0 15 ? * WED");
```

以上的触发器会在每周三的下午3点执行指定任务。现在我们只要创建一个JobDetail对象，并把它和上面的触发器一起传递给SchedulerPlugIn的scheduleWork()方法。

```
    JobDetail jobDetail =
        new JobDetail("Hello World Job",
                      "Hello World Group",
                      HelloWorld.class,
                      true, true, true);
//Schedule The work
//调度这项任务
    SchedulerPlugIn.scheduleWork(scheduledJobDetail, trigger);
```

实际工作在何处？

至此我们已决定Trigger，可以开始调度工作了。看上去一切都已完成，但实际上我们只是调度了一项任务，还有最重要的一步有待完成。注意HelloWorld.class作为参数传递给了JobDetail的构造函数。这个类就是实际完成工作的地方。HelloWorld继承了Quartz的Job类，并覆盖了execute()方法。当任务管理器决定运行这个任务时，execute()方法将被调用。来看代码：

```
import org.quartz.JobDataMap;
import org.quartz.JobDetail;
import org.quartz.JobExecutionContext;

//extend the proper Quartz class
```

```
//继承适当的Quartz类
public class HelloWorld extends Job {

//override the execute method
//覆盖execute方法
    public void execute(JobExecutionContext context) {

// Every job has it's own job detail
//每个Job都有独立的JobDetail
    JobDetail jobDetail = context.getJobDetail();
// The name is defined in the job definition
//name在Job定义中指定
    String jobName = jobDetail.getName();
//Every job has a Job Data map for storing extra information
//每个Job都有一个Job Data map来存放额外的信息
    JobDataMap dataMap = jobDetail.getJobDataMap();

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

出于测试的目的，你可能希望将触发器的频率调的高一点，以便观察到HelloWorld的动作。毕竟，你不想一直等到凌晨2点才能确定调度的任务确实运行了。相反，你可能需要一个每隔10秒运行的触发器：

```
Trigger trigger = new SimpleTrigger("trigger1", "group1");
trigger.setRepeatCount(SimpleTrigger.REPEAT_INDEFINITELY);
trigger.setRepeatInterval(10000L); // milliseconds毫秒
```

注意，这个触发器没有使用类cron的语法。Quartz有大量各类的选项和配置方法，可适用于任何任务调度的需要。

其它计时方式的配置

Quartz提供了多种调度任务的方式。CronTrigger可能是最复杂的一种，不过还有其它的选择。大多数触发器可以由Quartz提供的TriggerUtils类创建。以下是一些常见的触发器的例子。如谚语所言，条条大路通罗马！

每天凌晨2:22触发的触发器

```
// 方法一：使用makeDailyTrigger

Trigger trigger = TriggerUtils.makeDailyTrigger(2, 22);
trigger.setName("trigger1");
trigger.setGroup("group1");

// 方法二：使用CronTrigger
```

```
Trigger trigger = new CronTrigger("trigger1", "group1");
trigger.setCronExpression("0 22 2 * * ?");
```

每5秒执行一次的触发器

```
/*  *
 * 方法一：makeSecondlyTrigger
 * 注意以下代码将创建一个立即启动的触发器。要控制启动时间，使用
 * trigger.setStartTime(Date)方法。
 */
Trigger trigger = TriggerUtils.makeSecondlyTrigger(5);
trigger.setName("MyFiveSecondTrigger");
trigger.setGroup("MyTriggerGroup");


 /*
 *
 * 方法二：设置SimpleTrigger的重复次数和间隔时间。
 * 注意以下代码将创建一个立即启动的触发器。要控制启动时间，使用
 * trigger.setStartTime(Date)方法。
 */
Trigger trigger = new SimpleTrigger("trigger1", "group1");
trigger.setRepeatCount(SimpleTrigger.REPEAT_INDEFINITELY);
trigger.setRepeatInterval(5000L); // milliseconds
```

按间隔时间运行任务

```
Trigger trigger = new SimpleTrigger("trigger1", "group1");
// 24 hours * 60(minutes per hour) *
// 60(seconds per minute) * 1000(milliseconds per second)
// 24小时 * 60（分钟每小时） * 60（秒每分钟）* 1000（毫秒每秒钟）
trigger.setRepeatInterval(24L * 60L * 60L * 1000L);
```

结论

在这个演示中，我们只接触了Quartz框架的一些初级功能。记住，Java 5 和J2EE 5也有自己的任务调度机制，但是它们不像Quartz那样灵活易用。Quartz是目前唯一的开源Java任务调度框架，它的确为开发者的锦囊中增加了很有用的内容。你可从Open Symphony下载Quartz，并得到一份很好的教程和使用说明。

2.5 用连接池提高Servlet访问数据库的效率

Java Servlet作为首选的服务器端数据处理技术，正在迅速取代CGI脚本。Servlet超越CGI的优势之一在于，不仅多个请求可以共享公用资源，而且还可以在不同用户请求之间保留持续数据。本文介绍一种充分发挥该特色的实用技术，即数据库连接池。

一、实现连接池的意义

动态Web站点往往用数据库存储的信息生成Web页面，每一个页面请求导致一次数据库访问。连接数据库不仅要开销一定的通讯和内存资源，还必须完成用户验证、安全上下文配置这类任务，因而往往成为最为耗时的操作。当然，实际的连接时间开销千变万化，但1到2秒延迟并非不常见。如果某个基于数据库的Web应用只需建立一次初始连接，不同页面请求能够共享同一连接，就能获得显著的性能改善。

Servlet是一个Java类。Servlet引擎（它可能是Web服务软件的一部分，也可能是一个独立的附加模块）在系统启动或Servlet第一次被请求时将该类装入Java虚拟机并创建它的一个实例。不同用户请求由同一Servlet实例的多个独立线程处理。那些要求在不同请求之间持续有效的数据既可以用Servlet的实例变量来保存，也可以保存在独立的辅助对象中。

用JDBC访问数据库首先要创建与数据库之间的连接，获得一个连接对象（Connection），由连接对象提供执行SQL语句的方法。本文介绍的数据库连接池包括一个管理类DBConnectionManager，负责提供与多个连接池对象（DBConnectionPool类）之间的接口。每一个连接池对象管理一组JDBC连接对象，每一个连接对象可以被任意数量的Servlet共享。

类DBConnectionPool提供以下功能：

1）从连接池获取（或创建）可用连接。

2）把连接返回给连接池。

3）在系统关闭时释放所有资源，关闭所有连接。

此外， DBConnectionPool类还能够处理无效连接（原来登记为可用的连接，由于某种原因不再可用，如超时，通讯问题），并能够限制连接池中的连接总数不超过某个预定值。

管理类DBConnectionManager用于管理多个连接池对象，它提供以下功能：

1）装载和注册JDBC驱动程序。

2）根据在属性文件中定义的属性创建连接池对象。

3）实现连接池名字与其实例之间的映射。

4）跟踪客户程序对连接池的引用，保证在最后一个客户程序结束时安全地关闭所有连接池。

本文余下部分将详细说明这两个类，最后给出一个示例演示Servlet使用连接池的一般过程。

二、具体实现

DBConnectionManager.java程序清单如下：

```
001 import java.io.*;
002 import java.sql.*;
003 import java.util.*;
004 import java.util.Date;
005
006 /**
007 * 管理类DBConnectionManager支持对一个或多个由属性文件定义的数据库连接
008 * 池的访问.客户程序可以调用 getInstance()方法访问本类的唯一实例.
009 */
010 public class DBConnectionManager {
011 static private DBConnectionManager instance; // 唯一实例
012 static private int clients;
013
014 private Vector drivers = new Vector();
015 private PrintWriter log;
016 private Hashtable pools = new Hashtable();
017
018 /**
019 * 返回唯一实例.如果是第一次调用此方法,则创建实例
020 *
021 * @return DBConnectionManager 唯一实例
022 */
023 static synchronized public DBConnectionManager getInstance() {
024 if (instance == null) {
025 instance = new DBConnectionManager();
026 }
027 clients++;
028 return instance;
```

```
029 }
030
031 /**
032 * 建构函数私有以防止其它对象创建本类实例
033 */
034 private DBConnectionManager() {
035 init();
036 }
037
038 /**
039 * 将连接对象返回给由名字指定的连接池
040 *
041 * @param name 在属性文件中定义的连接池名字
042 * @param con 连接对象
043 */
044 public void freeConnection(String name, Connection con) {
045 DBConnectionPool pool = (DBConnectionPool) pools.get(name);
046 if (pool != null) {
047 pool.freeConnection(con);
048 }
049 }
050
051 /**
052 * 获得一个可用的(空闲的)连接.如果没有可用连接,且已有连接数小于最大连接数
053 * 限制,则创建并返回新连接
054 *
055 * @param name 在属性文件中定义的连接池名字
056 * @return Connection 可用连接或null
057 */
058 public Connection getConnection(String name) {
```

```
059 DBConnectionPool pool = (DBConnectionPool) pools.get(name);
060 if (pool != null) {
061 return pool.getConnection();
062 }
063 return null;
064 }
065
066 /**
067 * 获得一个可用连接.若没有可用连接,且已有连接数小于最大连接数限制,
068 * 则创建并返回新连接.否则,在指定的时间内等待其它线程释放连接.
069 *
070 * @param name 连接池名字
071 * @param time 以毫秒计的等待时间
072 * @return Connection 可用连接或null
073 */
074 public Connection getConnection(String name, long time) {
075 DBConnectionPool pool = (DBConnectionPool) pools.get(name);
076 if (pool != null) {
077 return pool.getConnection(time);
078 }
079 return null;
080 }
081
082 /**
083 * 关闭所有连接,撤销驱动程序的注册
084 */
085 public synchronized void release() {
086 // 等待直到最后一个客户程序调用
087 if (--clients != 0) {
088 return;
```

```
089 }
090
091 Enumeration allPools = pools.elements();
092 while (allPools.hasMoreElements()) {
093 DBConnectionPool pool = (DBConnectionPool) allPools.nextElement();
094 pool.release();
095 }
096 Enumeration allDrivers = drivers.elements();
097 while (allDrivers.hasMoreElements()) {
098 Driver driver = (Driver) allDrivers.nextElement();
099 try {
100 DriverManager.deregisterDriver(driver);
101 log("撤销JDBC驱动程序 " + driver.getClass().getName()+"的注册");
102 }
103 catch (SQLException e) {
104 log(e, "无法撤销下列JDBC驱动程序的注册: " + driver.getClass().getName());
105 }
106 }
107 }
108
109 /**
110 * 根据指定属性创建连接池实例.
111 *
112 * @param props 连接池属性
113 */
114 private void createPools(Properties props) {
115 Enumeration propNames = props.propertyNames();
116 while (propNames.hasMoreElements()) {
117 String name = (String) propNames.nextElement();
118 if (name.endsWith(".url")) {
```

```
119 String poolName = name.substring(0, name.lastIndexOf("."));
120 String url = props.getProperty(poolName + ".url");
121 if (url == null) {
122 log("没有为连接池" + poolName + "指定URL");
123 continue;
124 }
125 String user = props.getProperty(poolName + ".user");
126 String password = props.getProperty(poolName + ".password");
127 String maxconn = props.getProperty(poolName + ".maxconn", "0");
128 int max;
129 try {
130 max = Integer.valueOf(maxconn).intValue();
131 }
132 catch (NumberFormatException e) {
133 log("错误的最大连接数限制: " + maxconn + " .连接池: " + poolName);
134 max = 0;
135 }
136 DBConnectionPool pool =
137 new DBConnectionPool(poolName, url, user, password, max);
138 pools.put(poolName, pool);
139 log("成功创建连接池" + poolName);
140 }
141 }
142 }
143
144 /**
145 * 读取属性完成初始化
146 */
147 private void init() {
148 InputStream is = getClass().getResourceAsStream("/db.properties");
```

```
149 Properties dbProps = new Properties();
150 try {
151 dbProps.load(is);
152 }
153 catch (Exception e) {
154 System.err.println("不能读取属性文件. " +
155 "请确保db.properties在CLASSPATH指定的路径中");
156 return;
157 }
158 String logFile = dbProps.getProperty("logfile", "DBConnectionManager.log");
159 try {
160 log = new PrintWriter(new FileWriter(logFile, true), true);
161 }
162 catch (IOException e) {
163 System.err.println("无法打开日志文件: " + logFile);
164 log = new PrintWriter(System.err);
165 }
166 loadDrivers(dbProps);
167 createPools(dbProps);
168 }
169
170 /**
171 * 装载和注册所有JDBC驱动程序
172 *
173 * @param props 属性
174 */
175 private void loadDrivers(Properties props) {
176 String driverClasses = props.getProperty("drivers");
177 StringTokenizer st = new StringTokenizer(driverClasses);
178 while (st.hasMoreElements()) {
```

```
179 String driverClassName = st.nextToken().trim();
180 try {
181 Driver driver = (Driver)
182 Class.forName(driverClassName).newInstance();
183 DriverManager.registerDriver(driver);
184 drivers.addElement(driver);
185 log("成功注册JDBC驱动程序" + driverClassName);
186 }
187 catch (Exception e) {
188 log("无法注册JDBC驱动程序: " +
189 driverClassName + ", 错误: " + e);
190 }
191 }
192 }
193
194 /**
195 * 将文本信息写入日志文件
196 */
197 private void log(String msg) {
198 log.println(new Date() + ": " + msg);
199 }
200
201 /**
202 * 将文本信息与异常写入日志文件
203 */
204 private void log(Throwable e, String msg) {
205 log.println(new Date() + ": " + msg);
206 e.printStackTrace(log);
207 }
208
```

```
209 /**
210 * 此内部类定义了一个连接池.它能够根据要求创建新连接,直到预定的最
211 * 大连接数为止.在返回连接给客户程序之前,它能够验证连接的有效性.
212 */
213 class DBConnectionPool {
214 private int checkedOut;
215 private Vector freeConnections = new Vector();
216 private int maxConn;
217 private String name;
218 private String password;
219 private String URL;
220 private String user;
221
222 /**
223 * 创建新的连接池
224 *
225 * @param name 连接池名字
226 * @param URL 数据库的JDBC URL
227 * @param user 数据库帐号,或 null
228 * @param password 密码,或 null
229 * @param maxConn 此连接池允许建立的最大连接数
230 */
231 public DBConnectionPool(String name, String URL, String user, String password,
232 int maxConn) {
233 this.name = name;
234 this.URL = URL;
235 this.user = user;
236 this.password = password;
237 this.maxConn = maxConn;
238 }
```

```
239
240 /**
241 * 将不再使用的连接返回给连接池
242 *
243 * @param con 客户程序释放的连接
244 */
245 public synchronized void freeConnection(Connection con) {
246 // 将指定连接加入到向量末尾
247 freeConnections.addElement(con);
248 checkedOut--;
249 notifyAll();
250 }
251
252 /**
253 * 从连接池获得一个可用连接.如没有空闲的连接且当前连接数小于最大连接
254 * 数限制,则创建新连接.如原来登记为可用的连接不再有效,则从向量删除之,
255 * 然后递归调用自己以尝试新的可用连接.
256 */
257 public synchronized Connection getConnection() {
258 Connection con = null;
259 if (freeConnections.size() > 0) {
260 // 获取向量中第一个可用连接
261 con = (Connection) freeConnections.firstElement();
262 freeConnections.removeElementAt(0);
263 try {
264 if (con.isClosed()) {
265 log("从连接池" + name+"删除一个无效连接");
266 // 递归调用自己,尝试再次获取可用连接
267 con = getConnection();
268 }
```

```
269 }
270 catch (SQLException e) {
271 log("从连接池" + name+"删除一个无效连接");
272 // 递归调用自己,尝试再次获取可用连接
273 con = getConnection();
274 }
275 }
276 else if (maxConn == 0 || checkedOut < maxConn) {
277 con = newConnection();
278 }
279 if (con != null) {
280 checkedOut++;
281 }
282 return con;
283 }
284
285 /**
286 * 从连接池获取可用连接.可以指定客户程序能够等待的最长时间
287 * 参见前一个getConnection()方法.
288 *
289 * @param timeout 以毫秒计的等待时间限制
290 */
291 public synchronized Connection getConnection(long timeout) {
292 long startTime = new Date().getTime();
293 Connection con;
294 while ((con = getConnection()) == null) {
295 try {
296 wait(timeout);
297 }
298 catch (InterruptedException e) {}
```

```
299 if ((new Date().getTime() - startTime) >= timeout) {
300 // wait()返回的原因是超时
301 return null;
302 }
303 }
304 return con;
305 }
306
307 /**
308 * 关闭所有连接
309 */
310 public synchronized void release() {
311 Enumeration allConnections = freeConnections.elements();
312 while (allConnections.hasMoreElements()) {
313 Connection con = (Connection) allConnections.nextElement();
314 try {
315 con.close();
316 log("关闭连接池" + name+"中的一个连接");
317 }
318 catch (SQLException e) {
319 log(e, "无法关闭连接池" + name+"中的连接");
320 }
321 }
322 freeConnections.removeAllElements();
323 }
324
325 /**
326 * 创建新的连接
327 */
328 private Connection newConnection() {
```

```
329 Connection con = null;
330 try {
331 if (user == null) {
332 con = DriverManager.getConnection(URL);
333 }
334 else {
335 con = DriverManager.getConnection(URL, user, password);
336 }
337 log("连接池" + name+"创建一个新的连接");
338 }
339 catch (SQLException e) {
340 log(e, "无法创建下列URL的连接: " + URL);
341 return null;
342 }
343 return con;
344 }
345 }
346 }
```

三、类DBConnectionPool说明

该类在209至345行实现，它表示指向某个数据库的连接池。数据库由JDBC URL标识。一个JDBC URL由三部分组成：协议标识（总是jdbc），驱动程序标识（如 odbc、idb、oracle等），数据库标识（其格式依赖于驱动程序）。例如，jdbc:odbc:demo，即是一个指向demo数据库的JDBC URL，而且访问该数据库要使用JDBC-ODBC驱动程序。每个连接池都有一个供客户程序使用的名字以及可选的用户帐号、密码、最大连接数限制。如果Web应用程序所支持的某些数据库操作可以被所有用户执行，而其它一些操作应由特别许可的用户执行，则可以为两类操作分别定义连接池，两个连接池使用相同的JDBC URL，但使用不同的帐号和密码。

类DBConnectionPool的建构函数需要上述所有数据作为其参数。如222至238行所示，这些数据被保存为它的实例变量：

如252至283行、285至305行所示， 客户程序可以使用DBConnectionPool类提供的两个方法获取可用连接。两者的共同之处在于：如连接池中存在可用连接，则直接返回，否则创建新的连接并返回。如果没有可用连接且已有连接总数等于最大限制数，第一个方法将直接返回null，而第二个方法将等待直到有可用连接为止。

所有的可用连接对象均登记在名为freeConnections的向量（Vector）中。如果向量中有多于一个的连接，getConnection()总是选取第一个。同时，由于新的可用连接总是从尾部加入向量，从而使得数据库连接由于长时间闲置而被关闭的风险减低到最小程度。

第一个getConnection()在返回可用连接给客户程序之前，调用了isClosed()方法验证连接仍旧有效。如果该连接被关闭或触发异常，getConnection()递归地调用自己以尝试获取另外的可用连接。如果在向量freeConnections中不存在任何可用连接，getConnection()方法检查是否已经指定最大连接数限制。如已经指定，则检查当前连接数是否已经到达极限。此处maxConn为0表示没有限制。如果没有指定最大连接数限制或当前连接数小于该值，该方法尝试创建新的连接。如创建成功，则增加已使用连接的计数并返回，否则返回空值。

如325至345行所示，创建新连接由newConnection()方法实现。创建过程与是否已经指定数据库帐号、密码有关。

JDBC的DriverManager类提供多个getConnection()方法，这些方法要用到JDBC URL与其它一些参数，如用户帐号和密码等。DriverManager将使用指定的JDBC URL确定适合于目标数据库的驱动程序及建立连接。

在285至305行实现的第二个getConnection()方法需要一个以毫秒为单位的时间参数，该参数表示客户程序能够等待的最长时间。建立连接的具体操作仍旧由第一个getConnection()方法实现。

该方法执行时先将startTime初始化为当前时间。在while循环中尝试获得一个连接。如果失败，则以给定的时间值为参数调用wait()。wait()的返回可能是由于其它线程调用notify()或notifyAll()，也可能是由于预定时间已到。为找出wait()返回的真正原因，程序用当前时间减开始时间（startTime），如差值大于预定时间则返回空值，否则再次调用getConnection()。

把空闲的连接登记到连接池由240至250行的freeConnection()方法实现，它的参数为返回给连接池的连接对象。该对象被加入到freeConnections向量的末尾，然后减少已使用连接计数。调用notifyAll()是为了通知其它正在等待可用连接的线程。

许多Servlet引擎为实现安全关闭提供多种方法。数据库连接池需要知道该事件以保证所有连接能够正常关闭。DBConnectionManager类负协调整个关闭过程，但关闭连接池中所有连接的任务则由DBConnectionPool类负责。在307至323行实现的release()方法供DBConnectionManager调用。该方法遍历freeConnections向量并关闭所有连接，然后从向量中删除这些连接。

四、类DBConnectionManager 说明

该类只能创建一个实例，其它对象能够调用其静态方法（也称为类方法）获得该唯一实例的引用。如031至036行所示，DBConnectionManager类的建构函数是私有的，这是为了避免其它对象创建该类的实例。

DBConnectionManager类的客户程序可以调用getInstance()方法获得对该类唯一实例的引用。如018至029行所示，类的唯一实例在getInstance()方法第一次被调用期间创建，此后其引用就一直保存在静态变量instance中。每次调用getInstance()都增加一个DBConnectionManager的客户程序计数。即，该计数代表引用DBConnectionManager唯一实例的客户程序总数，它将被用于控制连接池的关闭操作。

该类实例的初始化工作由146至168行之间的私有方法init()完成。其中getResourceAsStream()方法用于定位并打开外部文件。外部文件的定位方法依赖于类装载器的实现。标准的本地类装载器查找操作总是开始于类文件所在路径，也能够搜索CLASSPATH中声明的路径。db.properties是一个属性文件，它包含定义连接池的键-值对。可供定义的公用属性如下：

drivers 以空格分隔的JDBC驱动程序类列表

logfile 日志文件的绝对路径

其它的属性和特定连接池相关，其属性名字前应加上连接池名字：

<poolname>.url 数据库的 JDBC URL

<poolname>.maxconn 允许建立的最大连接数，0表示没有限制

<poolname>.user 用于该连接池的数据库帐号

<poolname>.password 相应的密码

其中url属性是必需的，而其它属性则是可选的。数据库帐号和密码必须合法。用于Windows平台的db.properties文件示例如下：

```
drivers=sun.jdbc.odbc.JdbcOdbcDriver jdbc.idbDriver

logfile=D:\\user\\src\\java\\DBConnectionManager\\log.txt

idb.url=jdbc:idb:c:\\local\\javawebserver1.1\\db\\db.prp

idb.maxconn=2

access.url=jdbc:odbc:demo

access.user=demo

access.password=demopw
```

注意在Windows路径中的反斜杠必须输入2个，这是由于属性文件中的反斜杠同时也是一个转义字符。

init()方法在创建属性对象并读取db.properties文件之后，就开始检查logfile属性。如果属性文件中没有指定日志文件，则默认为当前目录下的DBConnectionManager.log文件。如日志文件无法使用，则向System.err输出日志记录。

装载和注册所有在drivers属性中指定的JDBC驱动程序由170至192行之间的loadDrivers()方法实现。该方法先用StringTokenizer将drivers属性值分割为对应于驱动程序名称的字符串，然后依次装载这些类并创建其实例，最后在DriverManager中注册该实例并把它加入到一个私有的向量drivers。向量drivers将用于关闭服务时从DriverManager取消所有JDBC驱动程序的注册。

init()方法的最后一个任务是调用私有方法createPools()创建连接池对象。如109至142行所示，createPools()方法先创建所有属性名字的枚举对象（即Enumeration对象，该对象可以想象为一个元素系列，逐次调用其nextElement()方法将顺序返

回各元素），然后在其中搜索名字以“.url”结尾的属性。对于每一个符合条件的属性，先提取其连接池名字部分，进而读取所有属于该连接池的属性，最后创建连接池对象并把它保存在实例变量pools中。散列表（Hashtable类 ）pools实现连接池名字到连接池对象之间的映射，此处以连接池名字为键，连接池对象为值。

为便于客户程序从指定连接池获得可用连接或将连接返回给连接池，类DBConnectionManager提供了方法getConnection()和freeConnection()。所有这些方法都要求在参数中指定连接池名字，具体的连接获取或返回操作则调用对应的连接池对象完成。它们的实现分别在051至064行、066至080行、038至049行。

如082至107行所示，为实现连接池的安全关闭，DBConnectionManager提供了方法release()。在上面我们已经提到，所有DBConnectionManager的客户程序都应该调用静态方法getInstance()以获得该管理器的引用，此调用将增加客户程序计数。客户程序在关闭时调用release()可以递减该计数。当最后一个客户程序调用release()，递减后的引用计数为0，就可以调用各个连接池的release()方法关闭所有连接了。管理类release()方法最后的任务是撤销所有JDBC驱动程序的注册。

五、Servlet使用连接池示例

Servlet API所定义的Servlet生命周期类如：

1) 创建并初始化Servlet（init()方法）。

2) 响应客户程序的服务请求（service()方法）。

3) Servlet终止运行，释放所有资源（destroy()方法）。

本例演示连接池应用，上述关键步骤中的相关操作为：

1) 在init()，用实例变量connMgr 保存调用DBConnectionManager.getInstance()所返回的引用。

2) 在service()，调用getConnection()，执行数据库操作，用freeConnection()将连接返回给连接池。

3) 在destroy()，调用release()关闭所有连接，释放所有资源。

示例程序清单如下：

```
import java.io.*;

import java.sql.*;

import javax.servlet.*;

import javax.servlet.http.*;
```

```
public class TestServlet extends HttpServlet {

private DBConnectionManager connMgr;


public void init(ServletConfig conf) throws ServletException {

super.init(conf);

connMgr = DBConnectionManager.getInstance();

}


public void service(HttpServletRequest req, HttpServletResponse res)

throws IOException {


res.setContentType("text/html");

PrintWriter out = res.getWriter();

Connection con = connMgr.getConnection("idb");

if (con == null) {

out.println("不能获取数据库连接.");

return;

}

ResultSet rs = null;

ResultSetMetaData md = null;

Statement stmt = null;

try {

stmt = con.createStatement();

rs = stmt.executeQuery("SELECT * FROM EMPLOYEE");

md = rs.getMetaData();

out.println("<H1>职工数据</H1>");

while (rs.next()) {

out.println("<BR>");

for (int i = 1; i < md.getColumnCount(); i++) {

out.print(rs.getString(i) + ", ");
```

```
}

}

stmt.close();

rs.close();

}

catch (SQLException e) {

e.printStackTrace(out);

}

connMgr.freeConnection("idb", con);

}


public void destroy() {

connMgr.release();

super.destroy();

}

}
```

2.6 应用服务器的集群策略及Java EE 5.0

开源代表的经常是理想主义者，而商业公司代表的经常是现实主义者，两者之间有相互竞争的地方，但从长远来看，更多的是一种是相互补充、相互促进的过程……

编者按：在中国Java技术界，袁红岗是一个不能忽视的名字。他的观点，及对中间件趋势的看法，是很多人感兴趣的。日前，在金蝶Apusic于广州花园酒店举办的“Java俱乐部”上，记者和这位极少露面的金蝶中间件首席科学家就集群、Java EE5.0等热门话题展开了直率的深入对话。果然，袁红岗出语惊人，带来了很多独特的视角和精彩的观点。

记者：不管是一般的技术观点，还是在平时打单过程中，我们似乎可以感觉到，集群功能一直是国外中间件厂商攻击国内中间件的弱点。而据我们所知，你们金蝶中间件在去年下半年推出了自己的集群功能，并且在宣传中提及，在国家质检总局全行业这个大单中和几个主要国外产品同等测试，测试结果甚至排在前面，这是否表示Apusic的集群功能已经能满足客户的需求？你对集群功能又怎么看，你认为中J2EE集群的本质是什么？

袁红岗：首先我可以向你证实，在国家质检总局的核心电子业务系统“大通关”项目中，金蝶Apusic中间件与三家世界主要中间件厂商的产品，在同一平台和环境下用国际测试工具进行了全方位的性能测试，经过三轮严苛的点对点、兼容性和性能测试，结果我们成功夺标。在测试结果中，Apusic在集群性能上并不逊色国外同类产品。

集群是中间件厂商经常热捧的一个概念，说只有采取集群策略你的应用系统的性能才能提高。不明就里的用户在付出了数倍的价钱去购买集群设备和软件以后，却往往得不到所应该得到的效果。Apusic作为一家负责任的公司，应当向大家来澄清所谓的“集群悖论”。所谓集群，只有在细粒度计算中其效果才会明显，也就是将计算过程以一定的并行算法进行细分，将计算分布到多个处理机运行，最后再将计算结果合并。有一个很有名计划叫做SETI@home，是一项利用全球联网的闲置计算机共同搜索

地外文明的科学实验计划，只需要下载一个小程序就可以对从射点望远镜得到数据进行分析。这就是一个典型的细粒度计算，所有的参与计划的计算机并行地计算浩如烟海的庞大数据库中的一小段数据，再将计算的结果汇总，从而发现可能的智能信号。而反过来我们看到在J2EE应用中大多数计算都是粗粒度的，再加上事务处理需要在分布式计算中进行协调，更降低了集群的整体处理能力。因此集群并不是解决性能问题的最佳途径，在单机低并发的情况下如果你认为性能不理想，那么请不要指望集群能给你带来性能的提升，相反你会发现性能反而还会有所下降。

?

那么，集群仅是厂商宣传的噱头吗？在以下两种情况下集群是有用的：1. 高并发超负荷运行的主机，例如google这样的网站，它的访问量是相当大的，因此google会采取集群策略来分散客户的请求，以提高整体响应能力。我们接触的很多J2EE应用负荷量都不大，其实每秒访问量在500以下的应用都没有必要采取集群策略。2. 失效转移，其实我认为这才是集群真正有用的地方，使用一台低成本计算设备作为主设备的备份，在主设备发生故障时及时接替，以保证7x24小时不间断服务。综上所述，在准备采用集群之前，一定要仔细分析具体的应用环境，以避免不必要的浪费。

作为一种选择，Apusic同样实现了集群技术，但我们并没有沿用大多数应用服务器厂商所采取的内存复制技术（in-memory replication），我们知道在集群中需要在各结点之间同步一些状态信息，如果采用内存复制技术，将耗费大量的网络带宽，对性能也有很大影响。这是因为每当一个结点的状态发生变化时，都需要通过多播等方式向其他结点传递状态信息，随着集群内部结点的增多，内存复制将会非常频繁，从而造成广播风暴，严重阻塞带宽。Apusic所采取的技术是客户端缓存，即直接将状态信息保存在客户端，当服务器失效时将状态转移到可用服务器。

?

记者：其实直到现在，还有人对中国人能做出中间件不相信、对产品不信任。你在去年曾说“大家在同一个标准下开发，Apusic和IBM、BEA的产品没什么本质区别”、对于这句话，你今天能否再解释一下？

?

袁红岗：这个问题其实不需要证明，没有人认为神舟飞船和阿波罗飞船在本质上有什么区别，都是为载人航天而制造出来的工具，并不会因为一个是中国制造、另一个是美国制造，在用途上就存在什么区别。诚然，我们和国外产品还存在一些差距，但在J2EE标准框架之下，我们提供了完全可供用户使用的产品，用户的选择是对我们产品最大的肯定。中国软件起步较晚，基础较薄弱，但在中间件领域我们是及时跟进的，当时站在同一条起跑线上，现在仍然没有被淘汰出局，相反差距还在逐步缩小。我相信凭我们的技术实力，我们完全有资格和国外产品同台竞技。

?

记者：在我参加各种技术大会，包括去年北京Java10周年大会时，跟许多技术人员交流、聊天的时候，他们都反映Apusic的启动速度非常快，很快就启动了，和同类产品相比非常突出。看来使用者们对它快速启动的特点非常喜爱。据我了解，Apusic的代码只是其它产品的几分之一，是因为这个原因吗？你设计时是怎么想的？

?

袁红岗：很多人不理解，为什么Apusic和其他产品比起来代码规模上要小很多，但使用起来并没有感觉到有什么功能缺失呢？这里要涉及到软件使用上的一个“二八原则”，即80%的使用者通常只会用到一个软件20%的功能。象微软的产品个个都是巨无霸，但对某个产品真正做到完全精通的可以说寥寥无几。以Word为例，平时我们只是用它来写写文档，很多高级功能其实根本用不上。在Apusic应用服务器的开发上我们也是遵循同样的原则，我们将尽可能地将整个软件产品最重要的20%的功能做好、做完善，以保证大多数用户的需求，剩下的80%功能将根据需要逐步增加。譬如国外产品很早就有的集群功能我们最近才推出来，并不是我们没有能力实现集群功能，而是在我们看来，集群并不是解决性能问题的最好方案，只有在真正大并发请求下集群才会展现它的优势。因此，我们把集群功能归结为低优先级需求，只有在其他方面的性能和稳定性有了很大提高后再来考虑集群。

?

另一个使Apusic运行轻便的重要原因在于软件架构的设计。架构是一个软件的灵魂，好的架构将延长软件的生命力，轻松应付各种变化。Apusic的架构在2001年时就已定型，以微内核和多路复用为其核心，历经产品多次重大升级而未影响核心体系，展现了顽强的生命力。相反，如果架构设计不合理，每次升级都要对架构进行调整，势必引入大量冗余代码，使整个产品臃肿不堪。

第三个原因在于代码编写的简洁性上。莎士比亚有一句名言：“简洁是智慧的灵魂”，在科学界同样也推崇简洁性，麦克斯韦方程组简洁深刻，被誉为是上帝谱写的诗歌，爱因斯坦的著名公式 E=mc^2 更是将简洁性发挥到了极致。程序设计语言不仅仅是为计算机运行而设计的，它也是一种思想表达工具，甚至比自然语言更简洁、深刻、无歧义。我平时很少写文档，因为我认为代码本身就已经表达了作者的思想。当我看到简洁优美的代码时，我认为是在读一篇美丽的诗篇，并为作者深邃奔放的思想所折服。相反，当看到混乱、繁复而无章法的代码时，我相信作者的思想同样是混乱的。

?

记者：去年你曾预言 J2EE 正在迎来一次划时代的变革，关键词是EJB，对此你能再做一次阐述吗？

?

袁红岗：J2EE 可以说是这几年发展非常快速的一个应用开发平台，这和 Java 这种灵活、方便、开放、跨平台的语言具有不可分隔的联系。Java 是一个讲求实用的语言，所有对应用开发有用的特性都被吸收进来，成为 Java 平台的一部分，而那些花稍但无实际作用的特性被摒弃。Java 的类库丰富、开发社区完善等特色标志着它还将在相当长的时期内保持旺盛的生命力。

?

J2EE 可以说是在 Java 平台中应用最为广泛的技术，而且正在逐步走向成熟。JCP 组织在总结了过去 J2EE 实践中的经验和教训，在将来的 Java EE 5.0 规范中将对 J2EE 技术做一个相当大的调整，其中最受影响的就是 EJB 规范。过去 EJB 给人的感觉过重，而且开发过程复杂，不易维护，因此在实际项目中使用 EJB 的很少。在 EJB3 中，EJB 的开发方法将彻底改变，不再使用 Home, Remote 接口等概念，而采取轻量级的开发模式，以 Java 5.0 中引进的 annotation 作为 EJB 描述工具，特别是实体 Bean 的角色将被重新定位，使其单纯担负起 O/R Mapping 的作用。所有这些举措都是为了使 EJB 的开发过程更加简单、效率更高、维护更方便。

?

在 Web 开发方面，Java EE 5 增加了新的 JSF 规范，这是一个类似于 struts 之类的 web 开发框架，但比它们更方便。JSF 基于事件及 UI 组件模型的开发方式颠覆了传统的 web 开发习惯，结合开发工具的支持，将使其更接近于一个真正的 MVC 编程环境。使用 JSF 开发表现层现在看起来更象 swing 编程，基于 UI 组件的模型能够把 web 界面定制的很多工作独立出来，厂商可以针对具体的目标定制更多更好用的组件。而我们作为国产中间件厂商，比国外厂商理应更了解国内客户的需求，因此我们的应用开发平台 Apusic Studio 将成为以 JSF 为基础的一个强大的 web 应用开发平台。

记者：现在很多应用已经转移到所谓的轻量级 J2EE 方案上去了，比如 Spring，已经红了快两年了。现在才推出 Java EE 5.0 是不是来得太晚些了？怎样预期 Java EE 对于整个产业的影响？

袁红岗：在我们主办的“Java 俱乐部”地面活动上，也有技术爱好者问及我这个问题，当时我回答到“非官方和官方互相促进，Java EE 5.0 来得并不晚”。

开源社区（往往是非官方标准）代表的经常是理想主义者，而商业公司（往往是官方标准）代表的经常是现实主义者。两者之间有相互竞争的地方，但从长远来看，更多的是一种是相互补充、相互促进的过程。

Spring 及 Hibernate 等开源框架，已经对 Java EE 的技术发展趋势产生了非常重大的影响。或者说 Java EE 的发展，也借鉴并吸取了这些开源框架的一些优秀的思想。从技术层面来看，这反而会使 Java EE 具有一定的后发优势；同时，这些新技术在业界的广泛使用，总是需要一定的周期，并且，采用 Java EE 构建业务系统的最终用户，也总是希望能够获得诸多厂商的官方支持与商业标准。因此，从这些方面来看，Java EE 5.0 来得其实并不晚。至于 Java EE 对于整个产业的影响，可能是 J2EE 这个概念自诞生以来比较重量级的一次震撼。它所带来的影响是深远的（更多的表现在对 J2EE 开发过程的影响），但用户自 1.4 平台向 5.0 平台之上的迁移，却是平缓的。

记者：让我们把话题再回到你的心血杰作 Apusic Platform 产品家族上，有人曾说，国产中间件的出路是专注于某个产品、用单一化的差异优势竞争。以前同样有人讲 APUSIC 产品线单一，但现在我听说你们马上要推出消息中间件和开发平台，在这里你能否对这两种产品作一点介绍？

?

袁红岗：过去我们的产品线是比较单一，仅有一个应用服务器产品，这次我们推出了 Apusic MQ 和 Apusic Studio 这两个产品，使产品线有了很大的扩充，形成一个相对比较完整的 Apusic Platform 产品家族。在这个产品家族中 Apusic Studio 将扮演一个

非常重要的角色。我们知道，各厂商的应用服务器产品都是大同小异的，用户主要根据性能、易用性、售后服务等指标来进行选择。但是仅有应用服务器产品是不够的，还需要一个好的开发工具来支持。过去我们就深受缺少开发工具的困扰，很多用户其实很喜欢 Apusic 应用服务器，觉得用起来很不错，但开发起来太麻烦，甚至有人先在其他平台上开发，等开发完了再移植过来，这种困扰多少打击了一部分开发人员的积极性。

?

在 Apusic Studio立项会议上，我们一开始是想为JBuilder、Eclipse 等写一些插件，利用第三方开发工具来支持Apusic 的开发，但我们最终抛弃了这一想法，决定搞一个和Apusic 应用服务器紧密结合的，更接近于应用开发的平台。金蝶集团具有十几年的应用开发历史，积累了大量的经验和教训，在这样的基础上，我们更应该秉承“帮助客户成功”的集团宗旨，不仅向用户提供一个方便、高效的开发工具，更重要的是要在产品中和用户分享金蝶的应用开发经验，以开发思想、开发模式、开发工具、应用组件等全方位的开发平台提供给客户。

?

Apusic Studio 以 Eclipse 为基础，不仅具有大多数J2EE 开发工具所应当具有的功能，还提供了以XP 方法为代表的轻量级开发方面的工具，另外，我们还提供了一些现成的组件，以帮助用户快速构建应用系统。我们知道，在J2EE 规范中，将一个应用的开发分解成若干个角色，包括开发、配置、部署、管理等，但实际上，部署、配置、管理等阶段，都是开发过程中必不可少的一个阶段。那么，怎么把这些阶段有机的集成在一起呢？Apusic Studio就是这样一个统一的开发平台，它基于 Eclipse 技术，通过与 Apusic 应用服务器的紧密集成，给开发人员提供了一个轻量级的J2EE 开发平台。

记者：最后一个问题说点轻松的，国内很多程序员都曾看过你那篇创下点击几十万的牛帖《程序员的几个基本原则》，那么现实生活中你真是这样生活的吗？比如写累了休息时疯狂打暴力游戏？

2.7 Servlet 中的 Listener 的应用

由于工作需要,最近在找一些解决方案,发现Listener是一个很好的东西,

能够监听到session,application的 create,destroy,可以监听到session,application

属性绑定的变化,考虑了一下,可以应用在"在线人数统计","数据缓存"等各个方面,

下面是整理的一些资料.

Listener是Servlet的监听器，它可以监听客户端的请求、服务端的操作等。通过监听器，可以自动激发一些操作，比如监听在线的用户的数量。当增加一个HttpSession时，就激发sessionCreated(HttpSessionEvent se)方法，这样就可以给在线人数加1。常用的监听接口有以下几个：

ServletContextAttributeListener监听对ServletContext属性的操作，比如增加、删除、修改属性。

ServletContextListener监听ServletContext。当创建ServletContext时，激发contextInitialized(ServletContextEvent sce)方法；当销毁ServletContext时，激发contextDestroyed(ServletContextEvent sce)方法。

HttpSessionListener监听HttpSession的操作。当创建一个Session时，激发session Created(HttpSessionEvent se)方法；当销毁一个Session时，激发sessionDestroyed (HttpSessionEvent se)方法。

HttpSessionAttributeListener监听HttpSession中的属性的操作。当在Session增加一个属性时，激发attributeAdded(HttpSessionBindingEvent se) 方法；当在Session删除一个属性时，激发attributeRemoved(HttpSessionBindingEvent se)方法；当在Session属性被重新设置时，激发attributeReplaced(HttpSessionBindingEvent se) 方法。

下面我们开发一个具体的例子，这个监听器能够统计在线的人数。在ServletContext初始化和销毁时，在服务器控制台打印对应的信息。当ServletContext里的属性增加、改变、删除时，在服务器控制台打印对应的信息。

要获得以上的功能，监听器必须实现以下3个接口：

HttpSessionListener

ServletContextListener

ServletContextAttributeListener

我们看具体的代码，见示例14-9。

【程序源代码】

```
1 // ═══════════════ Program Discription ═══════════════
2 // 程序名称：示例14-9 : EncodingFilter .java
3 // 程序目的：学习使用监听器
4 // ═══════════════════════════════════════════════
5 import javax.servlet.http.*;
6 import javax.servlet.*;
7
8 public class OnLineCountListener implements HttpSessionListener,
ServletContextListener,ServletContextAttributeListener
9 {
10  private int count;
11  private ServletContext context = null;
12
13  public OnLineCountListener()
14  {
15   count=0;
16   //setContext();
17  }
18  //创建一个session时激发
19  public void sessionCreated(HttpSessionEvent se)
20  {
21   count++;
22   setContext(se);
23
24  }
25  //当一个session失效时激发
26  public void sessionDestroyed(HttpSessionEvent se)
27  {
28   count--;
29   setContext(se);
30  }
31  //设置context的属性，它将激发attributeReplaced或attributeAdded方法
32  public void setContext(HttpSessionEvent se)
33  {
34   se.getSession().getServletContext().
```

```
setAttribute("onLine",new Integer(count));
35  }
36   //增加一个新的属性时激发
37  public void attributeAdded(ServletContextAttributeEvent event) {
38
39  log("attributeAdded('" + event.getName() + "', '" +
40       event.getValue() + "')");
41
42      }
43
44    //删除一个新的属性时激发
45      public void attributeRemoved(ServletContextAttributeEvent event) {
46
47  log("attributeRemoved('" + event.getName() + "', '" +
48       event.getValue() + "')");
49
50      }
51
52  //属性被替代时激发
53      public void attributeReplaced(ServletContextAttributeEvent event) {
54
55   log("attributeReplaced('" + event.getName() + "', '" +
56         event.getValue() + "')");
57      }
58      //context删除时激发
59       public void contextDestroyed(ServletContextEvent event) {
60
61   log("contextDestroyed()");
62   this.context = null;
63
64      }
65
66      //context初始化时激发
67      public void contextInitialized(ServletContextEvent event) {
68
69   this.context = event.getServletContext();
70   log("contextInitialized()");
71
72      }
73      private void log(String message) {
74
75       System.out.println("ContextListener: " + message);
76      }
77 }
```

【程序注解】

在OnLineCountListener里，用count代表当前在线的人数，OnLineCountListener将在Web服务器启动时自动执行。当OnLineCountListener构造好后，把count设置为0。每增加一个Session，OnLineCountListener会自动调用sessionCreated(HttpSessionEvent se)方法；每销毁一个Session，OnLineCountListener会自动调用sessionDestroyed(HttpSessionEvent se)方法。当调用sessionCreated(HttpSessionEvent se)方法时，说明又有一个客户在请求，此时使在线的人数（count）加1，并且把count写到ServletContext中。ServletContext的信息是所有客户端共享的，这样，每个客户端都可以读取到当前在线的人数。

从作用域范围来说,Servlet的作用域有ServletContext,HttpSession,ServletRequest.

Context范围:

ServletContextListener:

对一个应用进行全局监听.随应用启动而启动,随应用消失而消失主要有两个方法:

contextDestroyed(ServletContextEvent event)

在应用关闭的时候调用

contextInitialized(ServletContextEvent event)

在应用启动的时候调用

这个监听器主要用于一些随着应用启动而要完成的工作,也就是很多人说的我想在容器启动的时候干.....

一般来说对"全局变量"初始化,如

```
public void contextInitialized(ServletContextEvent event){
ServletContex sc = event.getServletContext();
sc.setAttribute(name,value);
}
```

以后你就可以在任何servlet中getServletContext().getAttribute(name);

我最喜欢用它来做守护性工作,就是在contextInitialized(ServletContextEvent event)方法中实现一个Timer,然后就让应用在每次启动的时候让这个Timer工作:

程序代码：

```
public void contextInitialized(ServletContextEvent event){
timer = new Timer();
timer.schedule(new TimerTask(){
public void run(){
//do any things
}
},0,时间间隔);
}
```

有人说Timer只能规定从现在开始的多长时间后,每隔多久做一次事或在什么时间做一次事,那我想在每月1号或每天12点做一项工作如何做呢?你只要设一个间隔,然后每次判断一下当时是不是那个时间段就行了啊,比如每月一号做,那你时间间隔设为天,即24小时一个循环,然后在run方法中判断当时日期new Date().getDate()==1就行了啊.如果是每天的12点,那你时间间隔设为小时,然后在run中判断new Date().getHour()==12,再做某事就行了.

ServletContextAttributeListener:

这个监听器主要监听ServletContex对象在setAttribute()和removeAttribute()的事件,注意,也就是一个"全局变量"在被Add(第一次set),replace(对已有的变量重新赋值)和remove的时候,分别调用下面三个方法:
public void attributeAdded(ServletContextAttributeEvent scab)这个方法不仅可以知道哪些全局变量被加进来,而且可获取容器在启动时自动设置了哪些context变量:
程序代码:

```
public void attributeAdded(ServletContextAttributeEvent scab){
System.out.println(scab.getName());
}
public void attributeRemoved(ServletContextAttributeEvent scab)

public void attributeReplaced(ServletContextAttributeEvent scab)
```

Session范围:
HttpSessionListener:
这个监听器主要监听一个Session对象被生成和销毁时发生的事件.对应有两个方法:
程序代码:

```
public void sessionCreated(HttpSessionEvent se)

public void sessionDestroyed(HttpSessionEvent se)
```

一般来说,一个session对象被create时,可以说明有一个新客端进入.可以用来粗略统计在线人数,注意这不是精确的,因为这个客户端可能立即就关闭了,但sessionDestroyed方法却会按一定的策略很久以后才会发生.

HttpSessionAttributeListener:和ServletContextAttributeListener一样,它监听一个session对象的Attribut被Add(一个特定名称的Attribute每一次被设置),replace(已有名称的Attribute的值被重设)和remove时的事件.对就的有三个方法.
程序代码:

```
public void attributeAdded(HttpSessionBindingEvent se)

public void attributeRemoved(HttpSessionBindingEvent se)

public void attributeReplaced(HttpSessionBindingEvent se)
```

上面的几个监听器的方法,都是在监听应用逻辑中servlet逻辑中发生了什么事,一般的来说.我们只要完成逻辑功能,比如session.setAttribute("aaa","111");我只要把一个名为aaa的变量放在session中以便以后我能获取它,我并不关心当session.setAttribute("aaa","111");发生时我还要干什么.(当然有些时候要利用的),但对于下面这个监听器,你应该好好发解一下:

HttpSessionBindingListener:
上面的监听器都是作为一个独立的Listener在容器中控制事件的.而HttpSessionBindingListener对在一对象中监听该对象的状态,实现了该接口的对象如果被作为value被add到一个session中或从session中remove,它就会知道自己已经作为一个

session对象或已经从session删除,这对于一些非纯JAVA对象,生命周期长于session的对象,以及其它需要释放资源或改变状态的对象非常重要.

比如:

```
session.setAttribute("abcd","1111");
```

以后session.removeAttribute("abcd");因为abcd是一个字符中,你从session中remove后,它就会自动被垃圾回收器回收,而如果是一个connection:(只是举例,你千万不要加connection往session中加入)

程序代码:

```
session.setAttribute("abcd",conn);
```

以后session.removeAttribute("abcd");这时这个conn被从session中remove了,你已经无法获取它的句柄,所以你根本没法关闭它.而在没有remove之前你根本不知道什么时候要被remove,你又无法close(),那么这个connection对象就死了.另外还有一些对象可以在被加入一个session时要锁定还要被remove时要解锁,应因你在程序中无法判断什么时候被remove(),add还好操作,我可以先加锁再add,但remove就后你就找不到它的句柄了,根本没法解锁,所以这些操作只能在对象自身中实现.也就是在对象被add时或remove时通知对象自己回调相应的方法:

程序代码:

```
MyConn extends Connection implements HttpSessionBindingListener{
public void valueBound(HttpSessionBindingEvent se){
this.initXXX();
}
public void valueUnbound(HttpSessionBindingEvent se){
this.close();
}
}

session.setAttribute("aaa",new MyConn());
```

这时如果调用session.removeAttribute("aaa"),则触发valueUnbound方法,就会自动关闭自己.而其它的需要改变状态的对象了是一样.

2.8 JSTL（JSP标准标签库）介绍

前言

从JSP 1.1规范开始，JSP就支持在JSP中使用自定义标签了，自定义标签的广泛使用造成了程序员重复定义，这样就促成了JSTL（JavaServer Pages Standard Tag Library）的诞生。

因为工作中需要用到JSTL，但网上却苦于找不到有关JSTL的中文资料，所以就有了这篇文章。

JSTL简介

JSTL是一个不断完善的开放源代码的JSP标签库，是由apache的jakarta小组来维护的。JSTL只能运行在支持JSP1.2和Servlet2.3规范的容器上，如tomcat 4.x。但是在即将推出的JSP 2.0中是作为标准支持的。

JSTL目前的最新版本为1.02，最终发布版为1.0。JSTL包含两个部分：标签库和EL（Expression Language表达式语言）语言。标签库目前支持四种标签： 标签 URI 前缀 示例

Core http://java.sun.com/jstl/core c <c:tagname ..>

XML processing http://java.sun.com/jstl/xml x <x:tagname ...>

I18N capable formatting http://java.sun.com/jstl/fmt fmt <fmt:tagname ...>

Database access (SQL) http://java.sun.com/jstl/sql sql <sql:tagname ..>

Core 支持 JSP 中的一些基本的操作；

XML processing 支持 XML 文档的处理；

I18N capable formatting 支持对 JSP 页面的国际化；

Database access (SQL)支持 JSP 对数据库的操作。

由于本人水平有限，本文仅介绍 Core 标签，如有兴趣，可一起探讨其它三种标签的使用与扩充。

EL 语言介绍

EL 语言是 JSTL 输出（输入）一个 JAVA表达式的表示形式。

在 JSTL 中，EL 语言只能在属性值中使用。EL 语言只能通过建立表达式 ${exp1}来进行调用。在属性值中使用表达式有三种方式。

1、 value 属性包含一个表达式

<some:tag value="${expr}"/>

在这种情况下，表达式值被计算出来并根据类型转换规则赋值给 value 属性。比如：<c:out value="${username}" />中的 ${username} 就是一个 EL，它相当于 JSP 语句 <%=request.getAttribute(“username”)%> 或 <%=session.getAttribute(“username”)%>

2、 value 属性包含一个或多个属性，这些属性被文本分割或围绕

<some:tag value="some${expr}${expr}text${expr}"/>

在这种情况下，表达式从左到右进行计算，并将结果转换为字符串型（根据类型转换规则），并将结果赋值给value 属性

3、 value 属性仅仅包含文本

<some:tag value="sometext"/>

在这种情况下，字符串型属性value 将根据类型转换规则转换为标签所希望的类型。

EL 语言的操作符

取得某个对象或集合中的属性值

为了获得集合中的属性，EL 支持以下两种操作

1. 使用.操作符来获得有名字的属性。例如表达式${user.username}表明对象 user 的 username 属性

2. 使用[]操作符来获得有名字或按数字排列的属性。

表达式${user["username"]}和表达式${user. username }含义相同

表达式${row[0]} 表明 row 集合的第一个条目。

在这里 user 是一个类的对象，它的属性 username 必须符合标准 JavaBean 的规范，即必须为 username 属性定义相应的 getter、setter 方法。

Empty 操作符(空值检查)

使用 empty 操作符来决定对象、集合或字符串变量是否为空或 null。例如：

${empty param.username}

如果 request 的参数列表中的 username 值为 null，则表达式的值为 true。 EL 也可以直接使用比较操作符与 null 进行比较。如 ${param.firstname == null}。

比较操作符 操作符 描述

==或 eq 相等检查

!=或 ne 不等检查

<或 lt 小于检查

>或 gt 大于检查

<=或 le 小于等于检查

>=或 ge 大于等于检查

数字运算符与逻辑运算符均与 JAVA 语言相同，不再列表。

Core 标签库

1、 通用标签

<c:out>

<c:out>标签用于在 JSP 中显示数据，它有如下属性 属 性 描 述 是否必须 缺省值

value 输出的信息，可以是 EL 表达式或常量 是 无

default value 为空时显示信息 否 无

escapeXml 为 true 则避开特殊的 xml 字符集 否 true

例子: 您的用户名是: <c:out value=”${user.username}” default=”guest”/>

显示用户的用户名，如为空则显示 guest

<c:out value="${sessionScope.username}"/>

指定从 session 中获取 username 的值显示；

<c:out value="${username}" />

显示 username 的值，默认是从 request(page)中取，如果 request 中没有名为 username 的对象则从 session 中取，session 中没有则从 application(servletContext)中取，如果没有取到任何值则不显示。

<c:set>

<c:set>标签用于保存数据，它有如下属性 属 性 描 述 是否必须 缺省值

value 要保存的信息，可以是 EL 表达式或常量 否

target 需要修改属性的变量名，一般为 javabean 的实例 否 无

property 需要修改的 javabean 属性 否 无

var 需要保存信息的变量 否 无

scope 保存信息的变量的范围 否 page

如果指定了 target 属性, 那么 property 属性也必须指定。

例子： <c:set value="${test.testinfo}" var="test2" scope=”session”/>

将 test.testinfo 的值保存到 session 的 test2 中，其中 test 是一个 javabean 的实例，testinfo 是 test 对象的属性。

<c:set target="${cust.address}" property="city" value="${city}"/>

将对象 cust.address 的 city 属性值保存到变量 city 中

<c:remove>

<c:remove>标签用于删除数据，它有如下属性 属 性 描 述 是否必须 缺省值

var 要删除的变量 是 无

scope 被删除变量的范围 否 所有范围，包括 page、request、session、application 等

例子: <c:remove var="test2" scope="session"/>

从 session 中删除 test2 变量。

2、 流控制标签

<c:if>

<c:if>标签有如下属性 属 性 描 述 是否必须 缺省值

test 需要评价的条件，相当于 if (...){}语句中的条件 是 无

var 要求保存条件结果的变量名 否 无

scope 保存条件结果的变量范围 否 page

<c:choose>

这个标签不接受任何属性

<c:when>

<c:when>标签有以下属性 属 性 描 述 是否必须 缺省值

test 需要评价的条件 是 无

<c:otherwise>

这个标签同样不接受任何属性

例子： <c:if test="${user.wealthy}">

user.wealthy is true.

</c:if>

如果 user.wealthy 值 true，则显示 user.wealthy is true.

<c:choose>

<c:when test="${user.generous}">

user.generous is true.

</c:when>

<c:when test="${user.stingy}">

user.stingy is true.

</c:when>

<c:otherwise>

user.generous and user.stingy are false.

</c:otherwise>

</c:choose>

只有当条件 user.generous 返回值是 true 时，才显示 user.generous is true.

只有当条件 user.stingy 返回值是 true 时，才显示 user.stingy is true.

其它所有的情况（即 user.generous 和 user.stingy 的值都不为 true）全部显示 user.generous and user.stingy are false.

由于 JSTL 没有形如 if (){…} else {…}的条件语句，所以这种形式的语句只能用<c:choose>、<c:when>和<c:otherwise>标签共同来完成了。

3、 循环控制标签

<c:forEach>

<c:forEach>标签用于通用数据，它有以下属性 属 性 描 述 是否必须 缺省值

items 进行循环的项目 否 无

begin 开始条件 否 0

end 结束条件 否 集合中的最后一个项目

step 步长 否 1

var 代表当前项目的变量名 否 无

varStatus 显示循环状态的变量 否 无

例子： <c:forEach items="${vectors}" var="vector">

<c:out value="${vector}"/>

</c:forEach>

相当于 java 语句 for (int i=0;i<vectors.size();i++) {

out.println(vectors.get(i));

}

在这里 vectors 是一个 java.util.Vector 对象，里面存放的是 String 数据，vector 是当前循环条件下 String 对象。实际上这里的 vectors 可以是任何实现了 java.util. Collection 接口的对象。

<c:forEach begin="0" end="100" var="i" step="1">

count=<c:out value="${i}"/>

</c:forEach>

输出:

count=0

...

count=100

<c:forTokens>

<c:forTokens>标签有以下属性 属 性 描 述 是否必须 缺省值

items 进行循环的项目 是 无

delims 分割符 是 无

begin 开始条件 否 0

end 结束条件 否 集合中的最后一个项目

step 步长 否 1

var 代表当前项目的变量名 否 无

varStatus 显示循环状态的变量 否 无

例子

<c:forTokens items="a:b:c:d" delims=":" var="token">

<c:out value="${token}"/>

</c:forTokens>

这个标签的使用相当于 java.util.StringTokenizer 类。在这里将字符串 a:b:c:d 以：分开循环四次，token 是循环到当前分割到的字符串。

4.导入文件和URL

JSTL核心标签库支持使用<c:import>来包含文件，使用<c:url>来打印和格式化URL,使用<c:redirect>来重定向URL。

<c:import>

<c:import>标签包含另外一个页面代码到当前页，它有以下属性 属 性 描 述 是否必须 缺省值

url 需要导入页面的 url 是 无

context /后跟本地 web 应用程序的名字 否 当前应用程序

charEncoding 用于导入数据的字符集 否 ISO-8859-1

var 接受导入文本的变量名 否 page

scope 接受导入文本的变量的变量范围 否 1

varReader 用于接受导入文本的 java.io.Reader 变量名 否 无

varStatus 显示循环状态的变量 否 无

<c:url>

<c:url>标签输出一个 url 地址，它有以下属性 属 性 描 述 是否必须 缺省值

url url 地址 是 无

context /后跟本地 web 应用程序的名字 否 当前应用程序

charEncoding 用于导入数据的字符集 否 ISO-8859-1

var 接受处理过的 url 变量名，该变量存储 url 否 输出到页

scope 存储url 的变量名的变量范围 否 page

例子：

<c:import url="http://www.url.com/edit.js" var="newsfeed"/>

将 url http://www.url.com/edit.js 包含到当前页的当前位置，并将 url 保存到 newsfeed 变量中

<a href="<c:url url="/index.jsp"/>"/>

在当前页的当前位置输出<a href="http://www.yourname.com/index.jsp"/>，http://www.yourname.com是当前页的所在的位置。

<c:redirect>

<c:redirect>标签将请求重新定向到另外一个页面，它有以下属性 属 性 描 述 是否必须 缺省值

url url 地址 是 无

context /后跟本地 web 应用程序的名字 否 当前应用程序

例子：

<c:redirect url="http://www.yourname.com/login.jsp"/>

将请求重新定向到 http://www.yourname.com/login.jsp 页，相当于 response.setRedirect("http://www.yourname.com/login.jsp");

<c:param>

<c:param>标签用来传递参数给一个重定向或包含页面，它有以下属性 属 性 描 述 是否必须 缺省值

name 在 request 参数中设置的变量名 是 无

value 在 request 参数中设置的变量值 否 无

例子：

<c:redirect url="login.jsp">

<c:param name="id" value="888"/>

</c:redirect>

将参数 888 以 id 为名字传递到 login.jsp 页面，相当于 login.jsp?id=888

JSTL 的优点

1、 在应用程序服务器之间提供了一致的接口，最大程序地提高了 WEB 应用在各应用服务器之间的移植。

2、 简化了 JSP 和 WEB 应用程序的开发。

3、 以一种统一的方式减少了 JSP 中的 scriptlet 代码数量，可以达到没有任何 scriptlet 代码的程序。在我们公司的项目中是不允许有任何的 scriptlet 代码出现在 JSP 中。

4、 允许 JSP 设计工具与 WEB 应用程序开发的进一步集成。相信不久就会有支持 JSTL 的 IDE 开发工具出现。

总结

上面介绍的仅仅是 JSTL 的一部分，如果有时间我会继续把其它部分写出来分享给大家。如果要使用 JSTL，则必须将 jstl.jar 和 standard.jar 文件放到 classpath 中，如果你还需要使用 XML processing 及 Database access (SQL)标签，还要将相关 JAR 文件放到 classpath 中，这些 JAR 文件全部存在于下载回来的 zip 文件中。这个 zip 文件可以从 http://jakarta.apache.org/builds/jakarta-taglibs/releases/standard/jakarta-taglibs-standard-1.0.zip 下载。

参考资料

1、 http://java.sun.com/products/jsp/jstl/

sun 公司的 JSTL 站点

2、 http://jakarta.apache.org/taglibs/doc/standard-doc/intro.html

jakarta 小组的 JSTL 站点

3、 http://www.manning.com/bayern/appendixA.pdf

JSTL的参考文档，本文很多内容都是从这个PDF文件里翻译的。

4、 <<J2EE编程指南（1.3版）>>

介绍了JSTL的雏形，wrox的书都是精品。

3 JAVA扩展

3.1 Log4j基本使用方法

Log4j由三个重要的组件构成：日志信息的优先级，日志信息的输出目的地，日志信息的输出格式。日志信息的优先级从高到低有ERROR、WARN、INFO、DEBUG，分别用来指定这条日志信息的重要程度；日志信息的输出目的地指定了日志将打印到控制台还是文件中；而输出格式则控制了日志信息的显示内容。

一、定义配置文件

其实您也可以完全不使用配置文件，而是在代码中配置Log4j环境。但是，使用配置文件将使您的应用程序更加灵活。Log4j支持两种配置文件格式，一种是XML格式的文件，一种是Java特性文件（键=值）。下面我们介绍使用Java特性文件做为配置文件的方法：

1.配置根Logger，其语法为：

log4j.rootLogger = [level] , appenderName, appenderName, …

其中，level 是日志记录的优先级，分为OFF、FATAL、ERROR、WARN、INFO、DEBUG、ALL或者您定义的级别。Log4j建议只使用四个级别，优先级从高到低分别是ERROR、WARN、INFO、DEBUG。通过在这里定义的级别，您可以控制到应用程序中相应级别的日志信息的开关。比如在这里定义了INFO级别，则应用程序中所有DEBUG级别的日志信息将不被打印出来。 appenderName就是指定日志信息输出到哪个地方。您可以同时指定多个输出目的地。

2.配置日志信息输出目的地Appender，其语法为：

log4j.appender.appenderName = fully.qualified.name.of.appender.class
log4j.appender.appenderName.option1 = value1
…
log4j.appender.appenderName.option = valueN

其中，Log4j提供的appender有以下几种：
org.apache.log4j.ConsoleAppender（控制台），
org.apache.log4j.FileAppender（文件），
org.apache.log4j.DailyRollingFileAppender（每天产生一个日志文件），
org.apache.log4j.RollingFileAppender（文件大小到达指定尺寸的时候产生一个新的文件），
org.apache.log4j.WriterAppender（将日志信息以流格式发送到任意指定的地方）

3.配置日志信息的格式（布局），其语法为：

log4j.appender.appenderName.layout = fully.qualified.name.of.layout.class
log4j.appender.appenderName.layout.option1 = value1
…
log4j.appender.appenderName.layout.option = valueN

其中，Log4j 提供的layout有以下几种：

org.apache.log4j.HTMLLayout（以 HTML 表格形式布局），

org.apache.log4j.PatternLayout（可以灵活地指定布局模式），

org.apache.log4j.SimpleLayout（包含日志信息的级别和信息字符串），

org.apache.log4j.TTCCLayout（包含日志产生的时间、线程、类别等等信息）

Log4J 采用类似 C 语言中的 printf 函数的打印格式格式化日志信息，打印参数如下： %m 输出代码中指定的消息

%p 输出优先级，即DEBUG，INFO，WARN，ERROR，FATAL

%r 输出自应用启动到输出该 log 信息耗费的毫秒数

%c 输出所属的类目，通常就是所在类的全名

%t 输出产生该日志事件的线程名

%n 输出一个回车换行符，Windows 平台为“\r\n”，Unix 平台为“\n”

%d 输出日志时间点的日期或时间，默认格式为 ISO8601，也可以在其后指定格式，比如： %d{yyy MMM dd HH:mm:ss,SSS}，输出类似：2002 年 10 月 18 日 22：10：28，921

%l 输出日志事件的发生位置，包括类目名、发生的线程，以及在代码中的行数。举例：Testlog4.main(TestLog4.java:10)

二、在代码中使用 Log4j

1.得到记录器

使用 Log4j，第一步就是获取日志记录器，这个记录器将负责控制日志信息。其语法为：

public static Logger getLogger(String name)

通过指定的名字获得记录器，如果必要的话，则为这个名字创建一个新的记录器。Name一般取本类的名字，比如：

static Logger logger = Logger.getLogger (ServerWithLog4j.class.getName ())

2.读取配置文件

当获得了日志记录器之后，第二步将配置Log4j 环境，其语法为：

BasicConfigurator.configure()： 自动快速地使用缺省 Log4j 环境。

PropertyConfigurator.configure (String configFilename) ：读取使用 Java 的特性文件编写的配置文件。

DOMConfigurator.configure (String filename) ：读取XML 形式的配置文件。

3.插入记录信息（格式化日志信息）

当上两个必要步骤执行完毕，您就可以轻松地使用不同优先级别的日志记录语句插入到您想记录日志的任何地方，其语法如下：

Logger.debug (Object message) ;

Logger.info (Object message) ;

Logger.warn (Object message) ;

Logger.error (Object message) ;

3.2 Dom4j 使用简介

DOM4J 是 dom4j.org 出品的一个开源 XML 解析包，它的网站中这样定义：

Dom4j is an easy to use, open source library for working with XML, XPath and XSLT on the Java platform using the Java Collections Framework and with full support for DOM, SAX and JAXP.

Dom4j是一个易用的、开源的库，用于XML，XPath和XSLT。它应用于Java平台，采用了Java集合框架并完全支持DOM，SAX和JAXP。

DOM4J使用起来非常简单。只要你了解基本的XML-DOM模型，就能使用。然而他自己带的指南只有短短一页（html），不过说的到挺全。国内的中文资料很少。因而俺写这个短小的教程方便大家使用，这篇文章仅谈及基本的用法，如需深入的使用，请……自己摸索或查找别的资料。

之前看过IBM developer社区的文章（参见附录），提到一些XML解析包的性能比较，其中DOM4J的性能非常出色，在多项测试中名列前茅。（事实上DOM4J的官方文档中也引用了这个比较）所以这次的项目中我采用了DOM4J作为XML解析工具。

在国内比较流行的是使用JDOM作为解析器，两者各擅其长，但DOM4J最大的特色是使用大量的接口，这也是它被认为比JDOM灵活的主要原因。大师不是说过么，“面向接口编程”。目前使用DOM4J的已经越来越多。如果你善于使用JDOM，不妨继续用下去，只看看本篇文章作为了解与比较，如果你正要采用一种解析器，不如就用DOM4J吧。

它的主要接口都在org.dom4j这个包里定义：

Attribute	Attribute定义了XML的属性
Branch	Branch为能够包含子节点的节点如XML元素(Element)和文档(Documents)定义了一个公共的行为，
CDATA	CDATA 定义了XML CDATA 区域
CharacterData	CharacterData是一个标识借口，标识基于字符的节点。如CDATA，Comment，Text.
Comment	Comment 定义了XML注释的行为
Document	定义了XML文档
DocumentType	DocumentType 定义XML DOCTYPE声明
Element	Element定义XML 元素
ElementHandler	ElementHandler定义了 Element 对象的处理器
ElementPath	被 ElementHandler 使用，用于取得当前正在处理的路径层次信息
Entity	Entity定义 XML entity
Node	Node为所有的dom4j中XML节点定义了多态行为
NodeFilter	NodeFilter 定义了在dom4j节点中产生的一个滤镜或谓词的行为 (predicate)
ProcessingInstruction	ProcessingInstruction 定义 XML 处理指令.

Text	Text 定义XML 文本节点.
Visitor	Visitor 用于实现Visitor模式.
XPath	XPath 在分析一个字符串后会提供一个XPath 表达式

看名字大致就知道它们的涵义如何了。

要想弄懂这套接口，关键的是要明白接口的继承关系：

- interface java.lang.Cloneable
  - interface org.dom4j.Node
    - interface org.dom4j.Attribute
    - interface org.dom4j.Branch
      - interface org.dom4j.Document
      - interface org.dom4j.Element
    - interface org.dom4j.CharacterData
      - interface org.dom4j.CDATA
      - interface org.dom4j.Comment
      - interface org.dom4j.Text
    - interface org.dom4j.DocumentType
    - interface org.dom4j.Entity
    - interface org.dom4j.ProcessingInstruction

一目了然，很多事情都清楚了。大部分都是由Node继承来的。知道这些关系，将来写程序就不会出现ClassCastException了。

下面给出一些例子（部分摘自DOM4J自带的文档），简单说一下如何使用。

1．　　读取并解析XML文档：

读写XML文档主要依赖于org.dom4j.io包，其中提供 DOMReader和SAXReader两类不同方式，而调用方式是一样的。这就是依靠接口的好处。

```
// 从文件读取XML，输入文件名，返回XML文档
public Document read(String fileName) throws MalformedURLException, DocumentException {
   SAXReader reader = new SAXReader();
   Document document = reader.read(new File(fileName));
```

```
    return document;
}
```

其中，reader的read方法是重载的，可以从InputStream, File, Url等多种不同的源来读取。得到的Document对象就带表了整个XML。

根据本人自己的经验，读取的字符编码是按照XML文件头定义的编码来转换。如果遇到乱码问题，注意要把各处的编码名称保持一致即可。

2． 取得Root节点

读取后的第二步，就是得到Root节点。熟悉XML的人都知道，一切XML分析都是从Root元素开始的。

```
public Element getRootElement(Document doc){
    return doc.getRootElement();
}
```

3． 遍历XML树

DOM4J提供至少3种遍历节点的方法：

1) 枚举(Iterator)

```
// 枚举所有子节点
for ( Iterator i = root.elementIterator(); i.hasNext(); ) {
    Element element = (Element) i.next();
    // do something
}
// 枚举名称为foo的节点
for ( Iterator i = root.elementIterator(foo); i.hasNext();) {
    Element foo = (Element) i.next();
    // do something
}
```

```
   // 枚举属性
   for ( Iterator i = root.attributeIterator(); i.hasNext(); ) {
      Attribute attribute = (Attribute) i.next();
      // do something
   }
```

2)递归

递归也可以采用Iterator作为枚举手段，但文档中提供了另外的做法

```
    public void treeWalk() {
      treeWalk(getRootElement());
   }
   public void treeWalk(Element element) {
      for (int i = 0, size = element.nodeCount(); i < size; i++)     {
          Node node = element.node(i);
          if (node instanceof Element) {
             treeWalk((Element) node);
          } else { // do something....
          }
      }
}
```

3) Visitor模式

最令人兴奋的是DOM4J对Visitor的支持，这样可以大大缩减代码量，并且清楚易懂。了解设计模式的人都知道，Visitor是GOF设计模式之一。其主要原理就是两种类互相保有对方的引用，并且一种作为Visitor去访问许多Visitable。我们来看DOM4J中的Visitor模式(快速文档中没有提供)

只需要自定一个类实现Visitor接口即可。

```
        public class MyVisitor extends VisitorSupport {
         public void visit(Element element){
             System.out.println(element.getName());
         }
```

```
        public void visit(Attribute attr){
            System.out.println(attr.getName());
        }
    }
    调用：  root.accept(new MyVisitor())
```

Visitor接口提供多种Visit()的重载，根据XML不同的对象，将采用不同的方式来访问。上面是给出的Element和Attribute的简单实现，一般比较常用的就是这两个。VisitorSupport是DOM4J提供的默认适配器，Visitor接口的Default Adapter模式，这个模式给出了各种visit(*)的空实现，以便简化代码。

注意，这个Visitor是自动遍历所有子节点的。如果是root.accept(MyVisitor)，将遍历子节点。我第一次用的时候，认为是需要自己遍历，便在递归中调用Visitor，结果可想而知。

4. XPath支持

DOM4J对XPath有良好的支持，如访问一个节点，可直接用XPath选择。

```
public void bar(Document document) {
    List list = document.selectNodes( //foo/bar );
    Node node = document.selectSingleNode(//foo/bar/author);
    String name = node.valueOf( @name );
}
```

例如，如果你想查找XHTML文档中所有的超链接，下面的代码可以实现：

```
public void findLinks(Document document) throws DocumentException {
    List list = document.selectNodes( //a/@href );
    for (Iterator iter = list.iterator(); iter.hasNext(); ) {
        Attribute attribute = (Attribute) iter.next();
        String url = attribute.getValue();
    }
}
```

5. 字符串与XML的转换

有时候经常要用到字符串转换为XML或反之，

```
// XML转字符串
```

```
  Document document = ...;
   String text = document.asXML();
// 字符串转XML
   String text = <person> <name>James</name> </person>;
   Document document = DocumentHelper.parseText(text);
```

6 用XSLT转换XML

```
   public Document styleDocument(
      Document document,
      String stylesheet
   ) throws Exception {
   // load the transformer using JAXP
   TransformerFactory factory = TransformerFactory.newInstance();
   Transformer transformer = factory.newTransformer(
      new StreamSource( stylesheet )
   );
   // now lets style the given document
   DocumentSource source = new DocumentSource( document );
   DocumentResult result = new DocumentResult();
   transformer.transform( source, result );
   // return the transformed document
   Document transformedDoc = result.getDocument();
   return transformedDoc;
}
```

7. 创建XML

一般创建XML是写文件前的工作，这就像StringBuffer一样容易。

```
    public Document createDocument() {
      Document document = DocumentHelper.createDocument();
      Element root = document.addElement(root);
```

```
    Element author1 =
        root
            .addElement(author)
            .addAttribute(name, James)
            .addAttribute(location, UK)
            .addText(James Strachan);
    Element author2 =
        root
            .addElement(author)
            .addAttribute(name, Bob)
            .addAttribute(location, US)
            .addText(Bob McWhirter);
    return document;
}
```

8. 文件输出

一个简单的输出方法是将一个Document或任何的Node通过write方法输出

```
 FileWriter out = new FileWriter( foo.xml );
document.write(out);
```

如果你想改变输出的格式，比如美化输出或缩减格式，可以用XMLWriter类

```
 public void write(Document document) throws IOException {
   // 指定文件
   XMLWriter writer = new XMLWriter(
       new FileWriter( output.xml )
   );
   writer.write( document );
   writer.close();
   // 美化格式
   OutputFormat format = OutputFormat.createPrettyPrint();
```

```
    writer = new XMLWriter( System.out, format );
    writer.write( document );
    // 缩减格式
    format = OutputFormat.createCompactFormat();
    writer = new XMLWriter( System.out, format );
    writer.write( document );
}
```

如何，DOM4J够简单吧，当然，还有一些复杂的应用没有提到，如ElementHandler等。如果你动心了，那就一起来用DOM4J.

DOM4J官方网站：(我老连不上)

http://www.dom4j.org/

DOM4J下载(SourceForge)，最新版本为1.4

http://sourceforge.net/projects/dom4j

用Dom4j解析XML及中文问题

发表于 2004年9月27日 20:21

本文主要讨论了用dom4j解析XML的基础问题，包括建立XML文档，添加、修改、删除节点，以及格式化（美化）输出和中文问题。可作为dom4j的入门资料。

转载自:http://jalorsoft.com/holen/

作者：陈光（holen@263.net）

时间：2004-09-11

本文主要讨论了用dom4j解析XML的基础问题，包括建立XML文档，添加、修改、删除节点，以及格式化（美化）输出和中文问题。可作为dom4j的入门资料。

1. 下载与安装

dom4j是sourceforge.net上的一个开源项目，主要用于对XML的解析。从2001年7月发布第一版以来，已陆续推出多个版本，目前最高版本为1.5。

dom4j专门针对Java开发，使用起来非常简单、直观，在Java界，dom4j正迅速普及。

可以到 http://sourceforge.net/projects/dom4j下载其最新版。

dom4j1.5的完整版大约13M，是一个名为dom4j-1.5.zip的压缩包，解压后有一个dom4j-1.5.jar文件，这就是应用时需要引入的类包，另外还有一个jaxen-1.1-beta-4.jar文件，一般也需要引入，否则执行时可能抛java.lang.NoClassDefFoundError: org/jaxen/JaxenException异常，其他的包可以选择用之。

2. 示例XML文档（holen.xml）

为了述说方便，先看一个XML文档，之后的操作均以此文档为基础。

holen.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<books>
    <!—This is a test for dom4j, holen, 2004.9.11—>
    <book show="yes">
       <title>Dom4j Tutorials</title>
    </book>
    <book show="yes">
       <title>Lucene Studing</title>
    </book>
    <book show="no">
       <title>Lucene in Action</title>
    </book>
    <owner>O'Reilly</owner>
</books>
```

这是一个很简单的XML文档，场景是一个网上书店，有很多书，每本书有两个属性，一个是书名[title]，一个为是否展示[show]，最后还有一项是这些书的拥有者[owner]信息。

3. 建立一个XML文档

```
/**
 * 建立一个XML文档,文档名由输入属性决定
 * @param filename 需建立的文件名
 * @return 返回操作结果, 0表失败, 1表成功
 */
public int createXMLFile(String filename){
   /** 返回操作结果, 0表失败, 1表成功 */
   int returnValue = 0;
   /** 建立document对象 */
   Document document = DocumentHelper.createDocument();
   /** 建立XML文档的根books */
   Element booksElement = document.addElement("books");
   /** 加入一行注释 */
   booksElement.addComment("This is a test for dom4j, holen, 2004.9.11");
   /** 加入第一个book节点 */
   Element bookElement = booksElement.addElement("book");
   /** 加入show属性内容 */
   bookElement.addAttribute("show","yes");
   /** 加入title节点 */
   Element titleElement = bookElement.addElement("title");
   /** 为title设置内容 */
   titleElement.setText("Dom4j Tutorials");

   /** 类似的完成后两个book */
   bookElement = booksElement.addElement("book");
   bookElement.addAttribute("show","yes");
   titleElement = bookElement.addElement("title");
   titleElement.setText("Lucene Studing");
   bookElement = booksElement.addElement("book");
```

```
        bookElement.addAttribute("show","no");
        titleElement = bookElement.addElement("title");
        titleElement.setText("Lucene in Action");

        /** 加入owner节点 */
        Element ownerElement = booksElement.addElement("owner");
        ownerElement.setText("O'Reilly");

        try{
            /** 将document中的内容写入文件中 */
            XMLWriter writer = new XMLWriter(new FileWriter(new File(filename)));
            writer.write(document);
            writer.close();
            /** 执行成功,需返回1 */
            returnValue = 1;
        }catch(Exception ex){
            ex.printStackTrace();
        }

        return returnValue;
    }
```

说明：

Document document = DocumentHelper.createDocument();

通过这句定义一个XML文档对象。

Element booksElement = document.addElement("books");

通过这句定义一个XML元素，这里添加的是根节点。

Element有几个重要的方法：

l addComment：添加注释

l addAttribute：添加属性

l addElement：添加子元素

最后通过XMLWriter生成物理文件，默认生成的XML文件排版格式比较乱，可以通过OutputFormat类的createCompactFormat()方法或createPrettyPrint()方法格式化输出，默认采用createCompactFormat()方法，显示比较紧凑，这点将在后面详细谈到。

生成后的holen.xml文件内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<books><!—This is a test for dom4j, holen, 2004.9.11—><book show="yes"><title>Dom4j Tutorials</title></book><book show="yes"><title>Lucene Studing</title></book><book show="no"><title>Lucene in Action</title></book><owner>O'Reilly</owner></books>
```

4. 修改XML文档

有三项修改任务，依次为：

l 如果book节点中show属性的内容为yes,则修改成no

l 把owner项内容改为Tshinghua，并添加date节点

l 若title内容为Dom4j Tutorials,则删除该节点

```
/**
 * 修改XML文件中内容,并另存为一个新文件
 * 重点掌握dom4j中如何添加节点,修改节点,删除节点
 * @param filename 修改对象文件
 * @param newfilename 修改后另存为该文件
 * @return 返回操作结果, 0表失败, 1表成功
 */
public int ModiXMLFile(String filename,String newfilename){
```

```
int returnValue = 0;
try{
    SAXReader saxReader = new SAXReader();
    Document document = saxReader.read(new File(filename));
    /** 修改内容之一: 如果book节点中show属性的内容为yes,则修改成no */
    /** 先用xpath查找对象 */
    List list = document.selectNodes("/books/book/@show" );
    Iterator iter = list.iterator();
    while(iter.hasNext()){
       Attribute attribute = (Attribute)iter.next();
       if(attribute.getValue().equals("yes")){
           attribute.setValue("no");
       }
    }

    /**
     * 修改内容之二: 把 owner项内容改为Tshinghua
     * 并在owner节点中加入date节点,date节点的内容为2004-09-11,还为date节点添加一个属性type
     */
    list = document.selectNodes("/books/owner" );
    iter = list.iterator();
    if(iter.hasNext()){
       Element ownerElement = (Element)iter.next();
       ownerElement.setText("Tshinghua");
       Element dateElement = ownerElement.addElement("date");
       dateElement.setText("2004-09-11");
       dateElement.addAttribute("type","Gregorian calendar");
    }
```

```
        /** 修改内容之三: 若 title内容为Dom4j Tutorials,则删除该节点 */
        list = document.selectNodes("/books/book");
        iter = list.iterator();
        while(iter.hasNext()){
           Element bookElement = (Element)iter.next();
           Iterator iterator = bookElement.elementIterator("title");
           while(iterator.hasNext()){
               Element titleElement=(Element)iterator.next();
               if(titleElement.getText().equals("Dom4j Tutorials")){
                  bookElement.remove(titleElement);
               }
           }
        }
        try{
           /** 将 document中的内容写入文件中 */
           XMLWriter writer = new XMLWriter(new FileWriter(new File(newfilename)));
           writer.write(document);
           writer.close();
           /** 执行成功,需返回 1 */
           returnValue = 1;
        }catch(Exception ex){
           ex.printStackTrace();
        }

    }catch(Exception ex){
        ex.printStackTrace();
    }
    return returnValue;
}
```

说明：

List list = document.selectNodes("/books/book/@show");

list = document.selectNodes("/books/book");

上述代码通过xpath查找到相应内容。

通过setValue()、setText()修改节点内容。

通过remove()删除节点或属性。

5. 格式化输出和指定编码

默认的输出方式为紧凑方式，默认编码为UTF-8，但对于我们的应用而言，一般都要用到中文，并且希望显示时按自动缩进的方式的显示，这就需用到OutputFormat类。

```
/**
 * 格式化XML文档,并解决中文问题
 * @param filename
 * @return
 */
public int formatXMLFile(String filename){
   int returnValue = 0;
   try{
       SAXReader saxReader = new SAXReader();
       Document document = saxReader.read(new File(filename));
       XMLWriter writer = null;
       /** 格式化输出,类型IE浏览一样 */
       OutputFormat format = OutputFormat.createPrettyPrint();
       /** 指定XML编码 */
       format.setEncoding("GBK");
```

```
            writer= new XMLWriter(new FileWriter(new File(filename)),format);
            writer.write(document);
            writer.close();
            /** 执行成功,需返回1 */
            returnValue = 1;
        }catch(Exception ex){
            ex.printStackTrace();
        }
        return returnValue;
    }
```

说明：

OutputFormat format = OutputFormat.createPrettyPrint();

这句指定了格式化的方式为缩进式，则非紧凑式。

format.setEncoding("GBK");

指定编码为GBK。

XMLWriter writer = new XMLWriter(new FileWriter(new File(filename)),format);

这与前面两个方法相比，多加了一个OutputFormat对象，用于指定显示和编码方式。

6. 完整的类代码

前面提出的方法都是零散的，下面给出完整类代码。

Dom4jDemo.java

```
package com.holen.dom4j;
```

```
import java.io.File;
import java.io.FileWriter;
import java.util.Iterator;
import java.util.List;

import org.dom4j.Attribute;
import org.dom4j.Document;
import org.dom4j.DocumentHelper;
import org.dom4j.Element;
import org.dom4j.io.OutputFormat;
import org.dom4j.io.SAXReader;
import org.dom4j.io.XMLWriter;

/**
 * @author Holen Chen
 */
public class Dom4jDemo {

    public Dom4jDemo() {
    }

    public int createXMLFile(String filename){…}
    public int ModiXMLFile(String filename,String newfilename){…}
    public int formatXMLFile(String filename){…}

    public static void main(String[] args) {
        Dom4jDemo temp = new Dom4jDemo();
        System.out.println(temp.createXMLFile("d://holen.xml"));        System.out.println(temp.ModiXMLFile("d:
//holen.xml","d://holen2.xml"));
```

```
        System.out.println(temp.formatXMLFile("d://holen2.xml"));
    }
}
```

说明：

main()方法中依次调用三个方法，第一个方法用于生成holen.xml，第二个方法用于修改holen.xml，并且修改后的内容另存为holen2.xml，第三个方法将holen2.xml格式化缩进式输出，并指定编码方式为GBK。

一个应用Dom4j的例子

[2005-4-21]

Created with Colorer-take5 Library. Type 'net.sf.colorer.FileType@777255'

```
 0: /*
 1:  * Created on 2005-4-19
 2:  *
 3:  * Copyright  (c) 2005 Julysea
 4:  * Window - Preferences - Java - Code Style - Code Templates
 5:  */
 6:
 7: /*应用此log4j的log4j.properties配置文件
 8:  *
 9:  ###############################################################################
10:  ## Categories and levels
11:  ###############################################################################
12:  *
13:  *log4j.rootCategory=DEBUG, FileApp, ConApp
14:  *log4j.category.de.jayefem=DEBUG, FileApp, ConApp
15:  *
```

```
16: *#########################################################################
17: *# Appenders
18: *#########################################################################
19: *
20: *# ConApp is set to be a ConsoleAppender.
21: *log4j.appender.ConApp=org.apache.log4j.ConsoleAppender
22: *log4j.appender.ConApp.Target=System.out
23: *log4j.appender.ConApp.layout=org.apache.log4j.PatternLayout
24: *log4j.appender.ConApp.layout.ConversionPattern=%d{ABSOLUTE} %5p %c{1}:%L - %m%n
25: *
26: *# FileApp
27: *log4j.appender.FileApp=org.apache.log4j.RollingFileAppender
28: *log4j.appender.FileApp.File=./log4e.log
29: *log4j.appender.FileApp.MaxFileSize=500KB
30: *# Keep one backup file
31: *log4j.appender.FileApp.MaxBackupIndex=1
32: *log4j.appender.FileApp.layout=org.apache.log4j.PatternLayout
33: *log4j.appender.FileApp.layout.ConversionPattern=%d [%t] %-5p %c - %m%n
34: */
35:
36:
37: /*应用此XML文件做测试
38: *
39: *<EW cmd="login" mod="Login" version="6.0">
40: *<Source uns="" type="user"/>
41: *<Username>zhangzhiyun@hp</Username>
42: *<Password>111111</Password>
43: *<Version>6.01.06.00</Version>
44: *</EW>
```

```
45:  */
46: package xml;
47:
48: import java.io.BufferedReader;
49: import java.io.BufferedWriter;
50: import java.io.File;
51: import java.io.FileReader;
52: import java.io.FileWriter;
53: import java.io.IOException;
54:
55: import org.apache.log4j.Logger;
56: import org.dom4j.Attribute;
57: import org.dom4j.DocumentException;
58: import org.dom4j.DocumentHelper;
59: import org.dom4j.Element;
60:
61: /**
62:  * @author julysea
63:  *
64:  * 一个用Dom4j解析xml的例子
65:  *
66:  */
67: public class Dom4jTest {
68:
69:     private static final Logger logger = Logger.getLogger(Dom4jTest.class);
70:
71:     public static void main(String[] args) throws IOException,
72:             DocumentException {
73:         BufferedReader reader=new BufferedReader(new FileReader("ew.xml"));
```

```
74:         String tempStr;
75:         String ewXml="";
76:         while((tempStr=reader.readLine())!=null) {
77:             ewXml=ewXml+tempStr;
78:             logger.debug(tempStr);
79:         }
80:         Element root = null;
81:
82:         root = DocumentHelper.parseText(ewXml).getRootElement();
83:         Attribute rootCmd=root.attribute("cmd");
84:         Attribute rootVersion=root.attribute("version");
85:         logger.debug("rootNmae = "+root.getName());
86:         logger.debug("EW'cmd = "+rootCmd.getValue());
87:         logger.debug("EW'version = "+rootVersion.getValue());
88:
89:         Element usrName=root.element("Username");
90:         logger.debug("EW.Username value = "+usrName.getTextTrim());
91:
92:         Element source=root.element("Source");
93:         Attribute sourceUns=source.attribute("uns");
94:         logger.debug("EW.Source'uns"+sourceUns.getValue());
95:         Attribute sourceType=source.attribute("type");
96:         logger.debug("EW.Source'type = "+sourceType.getValue());
97:
98:
99:         //创建一个Xml文件
100:        Element user=DocumentHelper.createElement("User");
101:        user.addAttribute("type", "user");
102:        user.addElement("name").addAttribute("type", "PinYin").setText("Julysea");
```

```
103:          user.addElement("age").setText("29");
104:          String oneXml=user.asXML();
105:
106:          BufferedWriter out=new BufferedWriter(new FileWriter("oneXml.xml"));
107:          out.write(oneXml);
108:          out.close();
109:     }
110: }
```

Dom4j编码问题彻底解决

这几天开始学习dom4j，在网上找了篇文章就开干了，上手非常的快，但是发现了个问题就是无法以UTF-8保存xml文件，保存后再次读出的时候会报“Invalid byte 2 of 2-byte UTF-8 sequence.”这样一个错误，检查发现由dom4j生成的这个文件，在使用可正确处理XML编码的任何的编辑器中中文成乱码，从记事本查看并不会出现乱码会正确显示中文。让我很是头痛。试着使用GBK、gb2312编码来生成的xml文件却可以正常的被解析。因此怀疑的dom4j没有对utf-8编码进行处理。便开始查看dom4j的原代码。终于发现的问题所在，是自己程序的问题。

在dom4j的范例和网上流行的《DOM4J 使用简介》这篇教程中新建一个xml文档的代码都类似如下

```
public void createXML(String fileName) {
    document. nbspdoc = org.dom4j.document. elper.createdocument. );
    Element root = doc.addElement("book");
    root.addAttribute("name", "我的图书");

    Element childTmp;
    childTmp = root.addElement("price");
    childTmp.setText("21.22");

    Element writer = root.addElement("author");
    writer.setText("李四");
    writer.addAttribute("ID", "001");

    try {
        org.dom4j.io.XMLWriter xmlWriter = new org.dom4j.io.XMLWriter(
                new FileWriter(fileName));
        xmlWriter.write(doc);
        xmlWriter.close();
    }
    catch (Exception e) {
        System.out.println(e);
```

```
        }
    }
```

在上面的代码中输出使用的是FileWriter对象进行文件的输出。这就是不能正确进行文件编码的原因所在，java中由Writer类继承下来的子类没有提供编码格式处理，所以dom4j也就无法对输出的文件进行正确的格式处理。这时候所保存的文件会以系统的默认编码对文件进行保存，在中文版的window下java的默认的编码为GBK，也就是所虽然我们标识了要将xml保存为utf-8格式但实际上文件是以GBK格式来保存的，所以这也就是为什么能够我们使用GBK、GB2312编码来生成xml文件能正确的被解析，而以UTF-8格式生成的文件不能被xml解析器所解析的原因。

好了现在我们找到了原因所在了，我们来找解决办法吧。首先我们看看dom4j是如何实现编码处理的

```
    public XMLWriter(OutputStream out) throws UnsupportedEncodingException {
        //System.out.println("In OutputStream");
        this.format = DEFAULT_FORMAT;
        this.writer = createWriter(out, format.getEncoding());
        this.autoFlush = true;
       namespaceStack.push(Namespace.NO_NAMESPACE);
    }


    public XMLWriter(OutputStream out, OutputFormat format) throws UnsupportedEncodingException {
        //System.out.println("In OutputStream,OutputFormat");
        this.format = format;
        this.writer = createWriter(out, format.getEncoding());
        this.autoFlush = true;
       namespaceStack.push(Namespace.NO_NAMESPACE);
    }
    /**
     * Get an OutputStreamWriter, use preferred encoding.
     */
    protected Writer createWriter(OutputStream outStream, String encoding) throws UnsupportedEncodingExceptio
n {
        return new BufferedWriter(
            new OutputStreamWriter( outStream, encoding )
        );
    }
```

由上面的代码我们可以看出dom4j对编码并没有进行什么很复杂的处理，完全通过java本身的功能来完成。所以我们在使用dom4j的来生成我们的XML文件时不应该直接为在构建XMLWriter时，不应该直接为其赋一个Writer对象，而应该通过一个OutputStream的子类对象来构建。也就是说在我们上面的代码中，不应该用FileWriter对象来构建xml文档，而应该使用FileOutputStream对象来构建所以将代码修改入下：

```
    public void createXML(String fileName) {
        document. nbspdoc = org.dom4j.document. elper.createdocument. );
        Element root = doc.addElement("book");
        root.addAttribute("name", "我的图书");

        Element childTmp;
        childTmp = root.addElement("price");
        childTmp.setText("21.22");
```

```
        Element writer = root.addElement("author");
        writer.setText("李四");
        writer.addAttribute("ID", "001");

        try {
            //注意这里的修改
            org.dom4j.io.XMLWriter xmlWriter = new org.dom4j.io.XMLWriter(
                    new FileOutputStream(fileName));
            xmlWriter.write(doc);
            xmlWriter.close();
        }
        catch (Exception e) {
            System.out.println(e);
        }
    }
```

至此DOM4J的问题编码问题算是告一段落，希望对此文章对其他朋友有用。

Dom4j的基本使用

下载dom4j后，在其文档中就用详细的使用说明，我又将其封装了一下：

```
package org.tju.msnrl.butil;

import java.io.*;
import java.util.*;
import org.dom4j.*;
import org.dom4j.io.XMLWriter;
import org.dom4j.io.SAXReader;

/**
 * Dom4j封装类
 * <p>Title: 天津大学博士后流动站</p>
 * <p>Description: 天津大学人事处制作维护</p>
 * <p>Copyright: Copyright (c) 2005</p>
 * <p>Company: 天津大学软件学院.NET实验室（MSNRL）</p>
 * @author Jonathan Q. Bo
 * @version 1.0
 */

public class BDom4j {
  /**XML文件路径*/
  private String XMLPath = null;
  /**XML文档*/
  private Document document = null;
```

```
public BDom4j() {
}

/**
 * 初始化xml文件
 * @param XMLPath 文件路径
 */
public BDom4j(String XMLPath){
  this.XMLPath = XMLPath;
}

/**
 * 打开文档
 */
public void openXML(){
  try{
    SAXReader reader = new SAXReader();
    this.document = reader.read(this.XMLPath);
    System.out.println("openXML() successful ...");
  }catch(Exception e){
    System.out.println("openXML() Exception:" + e.getMessage());
  }
}

/**
 * 创建文档
 * @param rootName 根节点名称
 */
public void createXML(String rootName){
  try{
    this.document = DocumentHelper.createDocument();
    Element root = document.addElement(rootName);
    System.out.println("createXML() successful...");
  }catch(Exception e){
    System.out.println("createXML() Exception:" + e.getMessage());
  }
}

/**
 * 添加根节点的child
 * @param nodeName 节点名
 * @param nodeValue 节点值
 */
public void addNodeFromRoot(String nodeName, String nodeValue){
  Element root = this.document.getRootElement();
  Element level1 = root.addElement(nodeName);
```

```
  level1.addText(nodeValue);
}

/**
 * 打开文档
 * @param filePath 文档路径
 */
public void openXML(String filePath){
  try{
    SAXReader saxReader = new SAXReader();
    this.document = saxReader.read(filePath);
    System.out.println("openXML(String filePath) successful ...");
  }catch(Exception e){
    System.out.println("openXML() Exception:" + e.getMessage());
  }
}

/**
 * 保存文档
 */
public void saveXML(){
  try{
    XMLWriter output = new XMLWriter(new FileWriter(new File(this.XMLPath)));
    output.write(document);
    output.close();
    System.out.println("saveXML() successful ...");
  }catch(Exception e1){
    System.out.println("saveXML() Exception:" + e1.getMessage());
  }
}

/**
 * 保存文档
 * @param toFilePath 保存路径
 */
public void saveXML(String toFilePath) {
  try {
    XMLWriter output = new XMLWriter(new FileWriter(new File(toFilePath)));
    output.write(document);
    output.close();
  }
  catch (Exception e1) {
    System.out.println("saveXML() Exception:" + e1.getMessage());
  }
}
```

```
/**
 * 获得某个节点的值
 * @param nodeName 节点名称
 */
public String getElementValue(String nodeName){
  try {
    Node node = document.selectSingleNode("//" + nodeName);
    return node.getText();
  }
  catch (Exception e1) {
    System.out.println("getElementValue() Exception：" + e1.getMessage());
    return null;
  }
}

/**
 * 获得某个节点的子节点的值
 * @param nodeName
 * @param childNodeName
 * @return
 */
public String getElementValue(String nodeName, String childNodeName){
  try {
    Node node = this.document.selectSingleNode("//" + nodeName + "/" + childNodeName);
    return node.getText();
  }
  catch (Exception e1) {
    System.out.println("getElementValue() Exception：" + e1.getMessage());
    return null;
  }
}

/**
 * 设置一个节点的text
 * @param nodeName 节点名
 * @param nodeValue 节点值
 */
public void setElementValue(String nodeName, String nodeValue){
  try{
    Node node = this.document.selectSingleNode("//" + nodeName);
    node.setText(nodeValue);
  }catch(Exception e1){
    System.out.println("setElementValue() Exception:" + e1.getMessage());
```

```
    }
  }

  /**
   * 设置一个节点值
   * @param nodeName 父节点名
   * @param childNodeName 节点名
   * @param nodeValue 节点值
   */
  public void setElementValue(String nodeName, String childNodeName,
                              String nodeValue) {
    try {
      Node node = this.document.selectSingleNode("//" + nodeName + "/" + childNodeName);
      node.setText(nodeValue);
    }
    catch (Exception e1) {
      System.out.println("setElementValue() Exception:" + e1.getMessage());
    }
  }

}
```

简单封装后，可以用来读写XML文档，对网站进行配置：如指定页面整体风格的css文件，在context init 时读取存入context中，在页面中通过读取context中的相应属性来确定css文件名，完成一项配置，其它的动态配置都类似；

```
public class BListener extends HttpServlet implements ServletContextListener, ServletContextAttributeListener,
 HttpSessionListener, HttpSessionAttributeListener {
  private static String XML_FILE_PATH = "c:/test.xml";

  //Notification that the web application is ready to process requests
  public void contextInitialized(ServletContextEvent sce) {
    BDom4j xmlmng = new BDom4j(XML_FILE_PATH);
    xmlmng.openXML();
    sce.getServletContext().setAttribute("css",xmlmng.getElementValue("style-sheet"));
    System.out.println("### context initialized...");
  }
```

3.3 Java 语言的 XML 验证 API

检查文档是否遵循了模式中规定的规则。不同的解析器和工具支持不同的模式语言如 DTD、W3C XML Schema 语言、RELAX NG 和 Schematron。Java 5™ 增加了统一的验证应用程序编程接口（API），可以把文档和用这种或那种语言编写的模式作比较。了解这种 XML 验证 API。

验证是一种强大的工具。它可以快速检查输入是否大体上符合预期的形式，立刻拒绝与处理目标相距甚远的文档。如果数据中存在问题，早发现要比晚发现好。

对于可扩展标记语言（XML）来说，验证一般意味着用各种模式语言为文档内容编写详细的规范，这些语言包括万维网联盟（W3C）的XML Schema Language(XSD)、RELAX NG、文档类型定义（DTD）和 Schematron 等。有时候验证在解析的同

时进行，有时候在解析完成后立刻进行。但一般在对输入的其他处理之前完成。（这一段描述只是粗略来说，因为存在例外。）

直到最近，程序请求验证的具体应用程序编程接口（API）还随着模式语言和解析器的不同而不同。DTD 和 XSD 是 Simple API for XML (SAX)、文档对象模型（DOM）和 Java™ API for XML Processing (JAXP) 常见的配置选项。RELAX NG 需要自定义的库和 API。Schematron 可以使用 Transformations API for XML(TrAX)，还有其他模式也要求程序员学习更多的 API，尽管执行的操作基本相同。

Java 5 引入了 javax.xml.validation 包，提供了独立于模式语言的验证服务接口。这个包也可用于 Java 1.3 及更高版本，不过要单独安装 JAXP 1.3。其他产品中，Xerces 2.8 包含了这个库的实现。

验证

javax.xml.validation API 使用三个类来验证文档：SchemaFactory、Schema 和 Validator。还大量使用了 TrAX 的 javax.xml.transform.Source 接口来表示 XML 文档。简言之，SchemaFactory 读取模式文档（通常是 XML 文件）并创建 Schema 对象。Schema 创建一个 Validator 对象。最后，Validator 对象验证表示为 Source 的 XML 文档。

清单 1 显示了一个简单的程序，用 DocBook XSD 模式验证在命令行中输入的 URL。

清单 1. 验证可扩展超文本标记语言（XHTML）文档

```
import java.io.*;
import javax.xml.transform.Source;
import javax.xml.transform.stream.StreamSource;
import javax.xml.validation.*;
import org.xml.sax.SAXException;

public class DocbookXSDCheck {

  public static void main(String[] args) throws SAXException, IOException {

    // 1. Lookup a factory for the W3C XML Schema language
    SchemaFactory factory =
      SchemaFactory.newInstance("http://www.w3.org/2001/XMLSchema");

    // 2. Compile the schema.
    // Here the schema is loaded from a java.io.File, but you could use
    // a java.net.URL or a javax.xml.transform.Source instead.
    File schemaLocation = new File("/opt/xml/docbook/xsd/docbook.xsd");
    Schema schema = factory.newSchema(schemaLocation);

    // 3. Get a validator from the schema.
    Validator validator = schema.newValidator();

    // 4. Parse the document you want to check.
    Source source = new StreamSource(args[0]);
```

```
    // 5. Check the document
    try {
      validator.validate(source);
      System.out.println(args[0] + " is valid.");
    }
    catch (SAXException ex) {
      System.out.println(args[0] + " is not valid because ");
      System.out.println(ex.getMessage());
    }

  }

}
```

下面是用捆绑到 Java 2 Software Development Kit (JDK) 5.0 的 Xerces 版本检查一个无效文档时的典型输出。

```
file:///Users/elharo/CS905/Course_Notes.xml is not valid because cvc-complex-type.2.3: Element 'legalnotice' cannot have character [children], because the type's content type is element-only.
```

改变验证所依据的模式、要验证的文档甚至使用的模式语言都很简单。但无论什么情况，验证都需要经过下列五个步骤：

为编写模式所用的语言加载一个模式工厂。
编译源文件中的模式。
用编译后的模式创建一个验证程序。
为需要验证的文档创建 Source 对象。StreamSource 通常最简单。
验证输入的源文档。如果文档无效，validate() 方法将抛出 SAXException。否则什么也不显示。
可以反复使用同一个验证程序和同一个模式多次。但是所有类都不是线程安全的或者可重入的。如果用多个线程同时验证，一定要保证每个线程有自己的 Validator 和 Schema 对象。

用文档指定的模式验证

有些文档指定了希望作为验证基础的模式，一般使用 xsi:noNamespaceSchemaLocation 和/或 xsi:schemaLocation 属性来指定，比如：

```
<DOCUMENT XMLNS:XSI="HTTP: XMLSchema-instance?
 xsi:noNamespaceSchemaLocation="http://www.example.com/document.xsd">
 ...
```

如果创建的模式没有指定 URL、文件或者源，则 Java 语言就会创建一个这样的东西，用于在要验证的文档中查找应该使用的模式。比如：

```
SchemaFactory factory = SchemaFactory.newInstance("http://www.w3.org/2001/XMLSchema");
Schema schema = factory.newSchema();
```

不过通常不希望这样做。一般应该由文档的消费者而不是生产者选择模式。另外，这种方法仅适用于 XSD。其他模式语言都需

要明确指定模式的位置。

抽象工厂

SchemaFactory 是一个抽象工厂。抽象工厂设计模式使得这种 API 能够支持多种不同的模式语言和对象模型。一种实现通常只能支持多种语言和模型的一部分。但是，一旦掌握了用 RELAX NG 模式（比方说）验证 DOM 文档的 API，就能用相同的 API 对 W3C 模式验证 JDOM 文档。

比如，清单 2 中的程序使用 DocBook 的 RELAX NG 模式验证 DocBook 文档。基本上与清单 1 相同。惟一的变化是模式位置和标识模式语言的 URL。

清单 2. 使用 RELAX NG 验证 DocBook 文档

```
import java.io.*;
import javax.xml.transform.Source;
import javax.xml.transform.stream.StreamSource;
import javax.xml.validation.*;
import org.xml.sax.SAXException;

public class DocbookRELAXNGCheck {

  public static void main(String[] args) throws SAXException, IOException {

    // 1. Specify you want a factory for RELAX NG
    SchemaFactory factory
     = SchemaFactory.newInstance("http://relaxng.org/ns/structure/1.0");

    // 2. Load the specific schema you want.
    // Here I load it from a java.io.File, but we could also use a
    // java.net.URL or a javax.xml.transform.Source
    File schemaLocation = new File("/opt/xml/docbook/rng/docbook.rng");
    // 3. Compile the schema.
    Schema schema = factory.newSchema(schemaLocation);

    // 4. Get a validator from the schema.
    Validator validator = schema.newValidator();

    // 5. Parse the document you want to check.
    String input
     = "file:///Users/elharo/Projects/workspace/CS905/build/Java_Course_Notes.xml";

    // 6. Check the document
    try {
      validator.validate(source);
      System.out.println(input + " is valid.");
```

```
      }
      catch (SAXException ex) {
        System.out.println(input + " is not valid because ");
        System.out.println(ex.getMessage());
      }
    }
}
```

如果用普通的 Sun JDK 不增加其他库，运行该程序时可能会看到如下所示的结果：

```
Exception in thread "main" java.lang.IllegalArgumentException:
http://relaxng.org/ns/structure/1.0
    at javax.xml.validation.SchemaFactory.newInstance(SchemaFactory.java:186)
    at DocbookRELAXNGCheck.main(DocbookRELAXNGCheck.java:14)
```

这是因为，JDK 本身没有带 RELAX NG 验证程序。如果不能识别模式语言，SchemaFactory.newInstance() 就会抛出 IllegalArgumentException。但是如果安装了 RELAX NG 库，比如 Jing 和 JAXP 1.3 适配程序，就会与 W3C 模式显示同样的结果。

确定模式语言

javax.xml.constants 类定义了几个常量来标识模式语言：

XMLConstants.W3C_XML_SCHEMA_NS_URI：http://www.w3.org/2001/XMLSchema
XMLConstants.RELAXNG_NS_URI：http://relaxng.org/ns/structure/1.0
XMLConstants.XML_DTD_NS_URI：http://www.w3.org/TR/REC-xml

这是一个不完全的列表。实现可以随时向该表增加其他 URL 来标识其他的模式语言。URL 通常是模式语言的名称空间统一资源标识符（URI）。比如，URL http://www.ascc.net/xml/schematron 标识了 Schematron 模式。

Sun 的 JDK 5 仅支持 XSD 模式。虽然也支持 DTD 验证，但是它不能通过 javax.xml.validation API 使用。对于 DTD，必须使用常规的 SAX XMLReader 类。不过可以另外安装支持不同模式语言的其他库。

如何定位模式工厂

Java 编程语言没有限制模式工厂只能有一种。可以把标识某种模式语言的 URI 传递给 SchemaFactory.newInstance()，它按照下列顺序搜索匹配的工厂：

用 "javax.xml.validation.SchemaFactory:schemaURL" 系统属性命名的类
用 $java.home/lib/jaxp.properties 文件中的 "javax.xml.validation.SchemaFactory:schemaURL" 属性命名的类
在任何 Java Archive (JAR) 文件的 META-INF/services 目录中发现的 javax.xml.validation.SchemaFactory 服务提供程序
平台默认的 SchemaFactory，JDK 5 中为 com.sun.org.apache.xerces.internal.jaxp.validation.xs.SchemaFactoryImpl

要支持自定义的模式语言和对应的验证程序，只需要编写 SchemaFactory、Schema 和 Validator（它们知道如何处理模式语言）的子类。然后将您的 JAR 文件安装到上述四个位置中的一个。对于添加与 W3C XML Schema 语言这类声明性语言相比更适合用 Java 之类的图灵完整语言检查的约束，这一点很重要。可以定义一种微模式语言，编写简单的实现，然后将其插入到验证层 。

错误处理

模式的默认响应方式是，如果遇到问题则抛出 SAXException，否则什么也不做。但是，可以提供 SAX ErrorHandler 来接收关于文档问题的更详尽的信息。比方说，假设要记录所有验证错误，但又不希望遇到错误时停止处理。可以安装一个像 清单 3 那样的错误处理程序。

清单 3. 使用 RELAX NG 验证 DocBook 文档

```
import org.xml.sax.ErrorHandler;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;

public class ForgivingErrorHandler implements ErrorHandler {

  public void warning(SAXParseException ex) {
    System.err.println(ex.getMessage());
  }

  public void error(SAXParseException ex) {
    System.err.println(ex.getMessage());
  }

  public void fatalError(SAXParseException ex) throws SAXException {
    throw ex;
  }
}
```

要安装该错误处理程序，需要创建它的一个实例并传递给 Validator 的 setErrorHandler() 方法：

```
ErrorHandler lenient = new ForgivingErrorHandler();
validator.setErrorHandler(lenient);
```

模式扩充

有些模式不仅仅执行验证。除了用是否回答文档有效与否的问题外，还为文档补充其他信息。比方说，可以提供默认的属性值。还可以给元素或属性赋予 int 或 gYear 这样的类型。验证程序可以创建这种补充了类型信息的文档，并写入 javax.xml.transform.Result 对象。只需要传递 Result 作为验证的第二个参数。比如，清单 4 在验证输入文档的同时，还创建结合有模式输入的扩展后的 DOM 文档。

清单 4. 用模式扩充文档

```
import java.io.*;
import javax.xml.transform.dom.*;
import javax.xml.validation.*;
import javax.xml.parsers.*;
```

```
import org.w3c.dom.*;
import org.xml.sax.SAXException;

public class DocbookXSDAugmenter {

  public static void main(String[] args)
   throws SAXException, IOException, ParserConfigurationException {

    SchemaFactory factory
     = SchemaFactory.newInstance("http://www.w3.org/2001/XMLSchema");
    File schemaLocation = new File("/opt/xml/docbook/xsd/docbook.xsd");
    Schema schema = factory.newSchema(schemaLocation);
    Validator validator = schema.newValidator();

    DocumentBuilderFactory domFactory = DocumentBuilderFactory.newInstance();
    domFactory.setNamespaceAware(true); // never forget this
    DocumentBuilder builder = domFactory.newDocumentBuilder();
    Document doc = builder.parse(new File(args[0]));

    DOMSource source = new DOMSource(doc);
    DOMResult result = new DOMResult();

    try {
      validator.validate(source, result);
      Document augmented = (Document) result.getNode();
      // do whatever you need to do with the augmented document...
    }
    catch (SAXException ex) {
      System.out.println(args[0] + " is not valid because ");
      System.out.println(ex.getMessage());
    }
  }
}
```

这个过程的输入和输出都有一定的限制。不能用于所有的流输入和输出。SAX 源可以扩展成 SAX 结果，DOM 源扩展成 DOM 结果，但是 SAX 源不能扩展成 DOM 结果，反之亦然。如果需要这么做，首先扩展成匹配的结果：SAX 对 SAX、DOM 对 DOM，然后使用 TrAX 的恒等转换改变模型。

但不建议使用这种技术。将文档需要的全部信息放在一个实例中，要比分解成实例和模式更可靠。您可以验证，但并非所有的人都能验证。

类型信息

W3C XML Schema Language在很大程度上依赖于类型 这一概念。元素和属性被声明为 int、double、date、duration、person、

PhoneNumber 或其他您能够想到的类型。Java Validation API 提供了一种手段来报告这些类型，虽然令人吃惊的是该特性独立于包的其他部分。

类型用 org.w3c.dom.TypeInfo对象表示。这个简单的接口通过 清单 5 来说明，它给出了类型的本地名和名称空间 URI。还可以告诉您它是否派生自其他类型。除此以外，理解这种类型就是您的程序的任务了。Java 语言没有说明它的含义，或者将其转化成 double和 java.util.Date 这样的 Java 类型。

清单 5. DOM TypeInfo 接口

```
package org.w3c.dom;

public interface TypeInfo {

  public static final int DERIVATION_RESTRICTION;
  public static final int DERIVATION_EXTENSION;
  public static final int DERIVATION_UNION;

  public String  getTypeName();
  public String  getTypeNamespace()
  public boolean isDerivedFrom(String namespace, String name, int derivationMethod);

}
```

要获得 TypeInfo 对象，需要向 Schema 对象请求 ValidatorHandler 而不是 Validator。ValidatorHandler 实现了 SAX 的 ContentHandler 接口。然后将该处理程序安装到 SAX 解析器中。

还要在 ValidatorHandler（不是解析器）中安装您自己的 ContentHandler，ValidatorHandler 将把扩展的事件转发到您的 ContentHandler。

ValidatorHandler 提供了 TypeInfoProvider，ContentHandler 可以随时调用查看当前元素或其属性的类型。它还可以告诉您该属性是否是 ID，属性是在文档中明确指定的还是模式中的默认值。清单 6 对这个类作了概括。

清单 6. TypeInfoProvider 类

```
package javax.xml.validation;

public abstract class TypeInfoProvider {

  public abstract TypeInfo getElementTypeInfo();
  public abstract TypeInfo getAttributeTypeInfo(int index);
  public abstract boolean isIdAttribute(int index);
  public abstract boolean isSpecified(int index);

}
```

最后，用 SAX XMLReader 解析文档。清单 7 是一个简单的程序，它利用了所有这些类和接口打印出文档所有元素的类型名。

清单 7. 列举元素类型

```
import java.io.*;
import javax.xml.validation.*;

import org.xml.sax.*;
import org.xml.sax.helpers.*;

public class TypeLister extends DefaultHandler {

   private TypeInfoProvider provider;

   public TypeLister(TypeInfoProvider provider) {
      this.provider = provider;
   }

   public static void main(String[] args) throws SAXException, IOException {

      SchemaFactory factory
       = SchemaFactory.newInstance("http://www.w3.org/2001/XMLSchema");
      File schemaLocation = new File("/opt/xml/docbook/xsd/docbook.xsd");
      Schema schema = factory.newSchema(schemaLocation);

      ValidatorHandler vHandler = schema.newValidatorHandler();
      TypeInfoProvider provider = vHandler.getTypeInfoProvider();
      ContentHandler   cHandler = new TypeLister(provider);
      vHandler.setContentHandler(cHandler);

      XMLReader parser = XMLReaderFactory.createXMLReader();
      parser.setContentHandler(vHandler);
      parser.parse(args[0]);

   }
   public void startElement(String namespace, String localName,
     String qualifiedName, Attributes atts) throws SAXException {
      String type = provider.getElementTypeInfo().getTypeName();
      System.out.println(qualifiedName + ": " + type);
   }
}
```

下面列出了对典型的 DocBook 文档运行上述代码的结果的开始部分：

```
book: #AnonType_book
```

title: #AnonType_title

subtitle: #AnonType_subtitle

info: #AnonType_info

copyright: #AnonType_copyright

year: #AnonType_year

holder: #AnonType_holder

author: #AnonType_author

personname: #AnonType_personname

firstname: #AnonType_firstname

othername: #AnonType_othername

surname: #AnonType_surname

personblurb: #AnonType_personblurb

para: #AnonType_para

link: #AnonType_link

可以看到，DocBook 模式赋予大多数元素以匿名的复杂类型。显然，结果会随着模式的不同而变化。

结束语

如果所有人都说同一种语言，世界会变得更加单调。如果只有一种编程语言可以选择，程序员也会感到不高兴。不同的语言更适合不同的任务，有些任务需要不只一种语言。XML 模式也不例外。您可以从各种各样的模式语言中选择。拥有了 Java 5 及其 javax.xml.validation，就能用一种 API 处理所有这些模式语言。

3.4 hibernate 的 hello word

终于有点对 hibernate 入门的感觉。方便门外的学习者，给一个简单的入门例子。 如果你有用过其他持久架构 转到 hibernate 其实很简单。一些原理方面就不讲了， robbin 讲的肯定比我好的多，自己去精华版看看。 我所给的只是我当初刚开始接触 hibernate 时候很想要的一个简单例子和设置方法。 一直没有找到，所以现在放到这里给大家看看，（只给想要入门的一个直观的感应，呵呵）

首先当然要新建一个项目

然 后在Project Properties->Paths->Required Libraries->add->new 这里定义hibernate 的类库 把 hibernate 的 lib 下面的所有 jar 包进去 当然还有 hibernate2.jar 也要 ，然后一路 ok 下去就可以了。

再来就是 hibernate.properties

从 hibernate 的 src 下面找到 把它拷到你项目的 src 目录下 （什么，你的项目没有 src 目录，新建一个随便的类就有 src 目录了）

这样一个 JB 下面的 hibernate 的开发环境就好了

然后在 hibernate.properties 里面设置你的数据库连接

默认是 HypersonicSQL

嗯 接下来的是你最想要做的事情了 其实很简单

新建一个类 Message.java

代码如下

代码:

```
package hello;

import java.io.Serializable;

/**
* @author getdown
* @version 1.0
*/

public class Message implements Serializable {
private Long id;
private String text;
//定义一个简单链表 指向另外的一个 Message
private Message nextMessage;
public Message() {}

public Message(Long id) {
this.id = id;
}

public Message(String text) {
this.text = text;
}

public Message(Long id, String text) {
this.id = id;
this.text = text;
}

public LonggetId() {
return id;
}

private void setId(Long id) {
this.id = id;
}

public StringgetText() {
return text;
}
```

```
public void setText(String text) {
this.text = text;
}

public Message getNextMessage() {
return nextMessage;
}

public void setNextMessage(Message nextMessage) {
this.nextMessage = nextMessage;
}

}
```

接下来是这个类对应的 hibernate 的配置文件 Message.hbm.xml

代码:

```
"-//Hibernate/Hibernate Mapping DTD 2.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">

name="hello.Message"
table="Messages"
>

column="MESSAGE_ID"
>

name="text"
type="string">

name="TEXT"
length="100"
not-null="true"
/>

name="nextMessage"
cascade="all"
column="NEXT_MESSAGE_ID"
/>
```

然后就是测试类

代码:

```
package hello;

import net.sf.hibernate.cfg.Configuration;
import net.sf.hibernate.SessionFactory;
import net.sf.hibernate.tool.hbm2ddl.SchemaExport;
import net.sf.hibernate.Session;
import net.sf.hibernate.Query;
import net.sf.hibernate.Hibernate;
import net.sf.hibernate.type.LongType;
import net.sf.hibernate.Transaction;

/**
* @author getdown
* @version 1.0
*/

public class Hello{
public Hello(){
}

public static void main(String[] args) throws Exception {
Configuration cfg = new Configuration().addClass(Message.class);

/** 顾名思义 构建表。。。第一次运行的时候运行下面语句可以在数据库生成表
* 之后可以把下面这句去掉
* */
// new SchemaExport(cfg).create(true, true);

//先生成sessionFactory
SessionFactory sessions = cfg.buildSessionFactory();
//再从sessionFactory 得到一个 session
Session session = sessions.openSession();
//开始对数据库的操作

/*—对数据库的创建操作——-*/
Message message = new Message("helloWorld");
//创建一条记录
session.save(message);
//存入记录
session.flush();

/*—对数据库的查询操作————*/
// Message message = new Message();
```

```
// Query q = session.createQuery("from Message as message where message.id=1");
// message = (Message) q.list().get(0);
// message.getNextMessage().setText("helloNext");
// session.flush();
// session.close();
// Long id = new Long(1);
// Message message = (Message) session.find("from Message as message where message.id=?", id, Hibernate.LONG).get(0);
// System.out.println(message.getText());

// /*——事务的处理————*/
// Transaction tx = session.beginTransaction();
// try {
// Message message = new Message("hello");
// session.save(message);
// session.flush();
// message = new Message("hello");
// session.save(message);
// session.flush();
// tx.commit();
// }
// catch (HibernateException ex) {
// tx.rollback();
// }

/*——添加 1000 条记录时间————-*/
// Message message;
// long start = System.currentTimeMillis();
// for(int i = 0; i < 1000; i ++) {
// message = new Message("hello");
// session.save(message);
// session.flush();
// }
// long end = System.currentTimeMillis();
// System.out.println("添加 1000 条记录时间－－－" + (end-start)/1000 + "s");

session.close();
}
}
```

ok 了 运行一下 Hello 看看出来什么吧

怎么样 比起 CMP 的持久 hibernate 的持久是不是显得很轻量级。还可以试试看hibernate 的性能 把 Hello.java 的最后一段注释去掉运行看看

当然 hibernate最重要的还是它的原理，还有很多很好的，很有趣的功能和 O/RM 设计思想等着你自己发掘。 多看看它自己的文档，可以学到很多东西，它的文档真的非常好。

3.5 JavaMail（JAVA邮件服务）API详解

一、JavaMail API简介

JavaMail API是读取、撰写、发送电子信息的可选包。我们可用它来建立如Eudora、Foxmail、MS Outlook Express一般的邮件用户代理程序(Mail User Agent,简称MUA)。而不是像sendmail或者其它的邮件传输代理(Mail Transfer Agent，简称MTA)程序那样可以传送、递送、转发邮件。从另外一个角度来看，我们这些电子邮件用户日常用MUA程序来读写邮件，而MUA依赖着MTA处理邮件的递送。

在清楚了到MUA与MTA之间的关系后，让我们看看JavaMail API是如何提供信息访问功能的吧！JavaMail API被设计用于以不依赖协议的方式去发送和接收电子信息，这个API被分为两大部分：

基本功能：如何以不依赖于协议的方式发送接收电子信息，这也是本文所要描述的，不过在下文中，大家将看到这只是一厢情愿而已。

第二个部分则是依赖特定协议的，比如SMTP、POP、IMAP、NNTP协议。在这部分的JavaMail API是为了和服务器通讯，并不在本文的内容中。

二、相关协议一览

在我们步入JavaMail API之前，先看一下API所涉及的协议。以下便是大家日常所知、所乐于使用的4大信息传输协议：

SMTP

POP

IMAP

MIME

当然，上面的4个协议，并不是全部，还有NNTP和其它一些协议可用于传输信息，但是由于不常用到，所以本文便不提及了。理解这4个基本的协议有助于我们更好的使用 JavaMail API。然而JavaMail API是被设计为与协议无关的，目前我们并不能克服这些协议的束缚。确切的说，如果我们使用的功能并不被我们选择的协议支持，那么JavaMail API并不可能如魔术师一样神奇的赋予我们这种能力。

1. SMTP

简单邮件传输协议定义了递送邮件的机制。在下文中，我们将使用基于Java-Mail的程序与公司或者ISP的SMTP服务器进行通讯。这个SMTP服务器将邮件转发到接收者的SMTP服务器，直至最后被接收者通过POP或者IMAP协议获取。这并不需要SMTP服务器使用支持授权的邮件转发，但是却的确要注意SMTP服务器的正确设置（SMTP服务器的设置与JavaMail API无关）。

2. POP

POP是一种邮局协议，目前为第3个版本，即众所周知的POP3。POP定义了一种用户如何获得邮件的机制。它规定了每个用户使用一个单独的邮箱。大多数人在使用POP时所熟悉的功能并非都被支持，例如查看邮箱中的新邮件数量。而这个功能是微软的Outlook内建的，那么就说明微软Outlook之类的邮件客户端软件是通过查询最近收到的邮件来计算新邮件的数量来实现前面所说的功能。因此在我们使用JavaMail API时需要注意，当需要获得如前面所讲的新邮件数量之类的信息时，我们不得不自己进行计算。

3. IMAP

IMAP使用在接收信息的高级协议，目前版本为第4版，所以也被称为IMAP4。需要注意的是在使用IMAP时，邮件服务器必须支持该协议。从这个方面讲,我们并不能完全使用IMAP来替代POP,不能期待IMAP在任何地方都被支持。假如邮件服务器支持IMAP,那么我们的邮件程序将能够具有以下被IMAP所支持的特性：每个用户在服务器上可具有多个目录，这些目录能在多个用户之间共享。

其与POP相比高级之处显而易见，但是在尝试采取IMAP时，我们认识到它并不是十分完美的：由于IMAP需要从其它服务器上接收新信息，将这些信息递送给用户，维护每个用户的多个目录，这都为邮件服务器带来了高负载。并且IMAP与POP的一个不

同之处是POP用户在接收邮件时将从邮件服务器上下载邮件，而IMAP允许用户直接访问邮件目录，所以在邮件服务器进行备份作业时，由于每个长期使用此邮件系统的用户所用的邮件目录会占有很大的空间，这将直接导致邮件服务器上磁盘空间暴涨。

4. MIME

MIME并不是用于传送邮件的协议，它作为多用途邮件的扩展定义了邮件内容的格式：信息格式、附件格式等等。一些RFC标准都涉及了MIME：RFC 822, RFC 2045, RFC 2046, RFC 2047，有兴趣的Matrixer可以阅读一下。而作为JavaMail API的开发者，我们并不需关心这些格式定义，但是这些格式被用在了程序中。

5. NNTP和其它的第三方协议

正因为JavaMail API在设计时考虑到与第三方协议实现提供商之间的分离，故我们可以很容易的添加一些第三方协议。SUN维护着一个第三方协议实现提供商的列表：http://java.sun.com/products/javamail/Third_Party.html，通过此列表我们可以找到所需要的而又不被SUN提供支持的第三方协议：比如NNTP这个新闻组协议和S/MIME这个安全的MIME协议。

三、安装

1. 安装JavaMail

为了使用JavaMail API，需要从 http://java.sun.com/products/javamail/downloads/index.html 下载文件名格式为javamail-[version].zip的文件（这个文件中包括了JavaMail实现），并将其中的mail.jar文件添加到CLASSPATH中。这个实现提供了对SMTP、IMAP4、POP3的支持。

注意：在安装JavaMail实现之后，我们将在demo目录中发现许多有趣的简单实例程序。

在安装了JavaMail之后,我们还需要安装JavaBeans Activation Framework，因为这个框架是JavaMail API所需要的。如果我们使用J2EE的话，那么我们并无需单独下载JavaMail，因为它存在于J2EE.jar中，只需将J2EE.jar加入到CLASSPATH即可。

2. 安装JavaBeans Activation Framework

从 http://java.sun.com/products/javabeans/glasgow/jaf.html 下载 JavaBeans Activation Framework，并将其添加到CLASSPATH中。此框架增加了对任何数据块的分类、以及对它们的处理的特性。这些特性是JavaMail API需要的。虽然听起来这些特性非常模糊，但是它对于我们的JavaMail API来说只是提供了基本的MIME类型支持。

到此为止，我们应当把mail.jar和activation.jar都添加到了CLASSPATH中。

当然如果从方便的角度讲，直接把这两个Jar文件复制到JRE目录的lib/ext目录中也可以。

四、初次认识JavaMail API

1. 了解我们的JavaMail环境

A. 纵览JavaMail核心类结构

打开JavaMail.jar文件，我们将发现在javax.mail的包下面存在着一些核心类：Session、Message、Address、Authenticator、Transport、Store、Folder。而且在javax.mail.internet包中还有一些常用的子类。

B. Session

Session类定义了基本的邮件会话。就像Http会话那样，我们进行收发邮件的工作都是基于这个会话的。Session对象利用了java.util.Properties对象获得了邮件服务器、用户名、密码信息和整个应用程序都要使用到的共享信息。

Session类的构造方法是私有的，所以我们可以使用Session类提供的getDefaultInstance()这个静态工厂方法获得一个默认的Session对象：

```
Properties props = new Properties();// fill props with any informationSession session = Session.getDefaultInstance(props, null);
```

或者使用getInstance()这个静态工厂方法获得自定义的Session:

```
Properties props = new Properties();// fill props with any informationSession session = Session.getInstance(p
```

rops, null);

从上面的两个例子中不难发现，getDefaultInstance()和 getInstance()方法的第二个参数都是null，这是因为在上面的例子中并没有使用到邮件授权，下文中将对授权进行详细介绍。

从很多的实例看，在对mail server进行访问的过程中使用共享的Session是足够的，即使是工作在多个用户邮箱的模式下也不例外。

C. Message

当我们建立了Session对象后，便可以被发送的构造信息体了。在这里SUN提供了Message类型来帮助开发者完成这项工作。由于Message是一个抽象类，大多数情况下，我们使用javax.mail.internet.MimeMessage这个子类，该类是使用MIME类型、MIME信息头的邮箱信息。信息头只能使用US-ASCII字符，而非ASCII字符将通过编码转换为ASCII的方式使用。

为了建立一个MimeMessage对象，我们必须将Session对象作为MimeMessage构造方法的参数传入：

```
MimeMessage message = new MimeMessage(session);
```

注意：对于MimeMessage类来讲存在着多种构造方法，比如使用输入流作为参数的构造方法。

在建立了MimeMessage对象后，我们需要设置它的各个part，对于MimeMessage类来说，这些part就是MimePart接口。最基本的设置信息内容的方法就是通过表示信息内容和米么类型的参数调用setContent()方法：

```
message.setContent("Hello", "text/plain");
```

然而，如果我们所使用的MimeMessage中信息内容是文本的话，我们便可以直接使用setText()方法来方便的设置文本内容。

```
message.setText("Hello");
```

前面所讲的两种方法，对于文本信息，后者更为合适。而对于其它的一些信息类型，比如HTML信息，则要使用前者。

别忘记了，使用setSubject()方法对邮件设置邮件主题：

```
message.setSubject("First");
```

D. Address

到这里，我们已经建立了Session和Message，下面将介绍如何使用邮件地址类：Address。像Message一样，Address类也是一个抽象类，所以我们将使用javax.mail.internet.InternetAddress这个子类。

通过传入代表邮件地址的字符串，我们可以建立一个邮件地址类：

```
Address address = new InternetAddress("president@whitehouse.gov");
```

如果要在邮件地址后面增加名字的话，可以通过传递两个参数：代表邮件地址和名字的字符串来建立一个具有邮件地址和名字的邮件地址类：

```
Address address = new InternetAddress("president@whitehouse.gov", "George Bush");
```

本文在这里所讲的邮件地址类是为了设置邮件信息的发信人和收信人而准备的，在建立了邮件地址类后，我们通过message的setFrom()和 setReplyTo()两种方法设置邮件的发信人：

```
message.setFrom(address);message.setReplyTo(address);
```

若在邮件中存在多个发信人地址，我们可用addForm()方法增加发信人：

```
Address address[] = ...;message.addFrom(address);
```

为了设置收信人，我们使用addRecipient()方法增加收信人，此方法需要使用Message.RecipientType的常量来区分收信人的类型：

```
message.addRecipient(type, address)
```

下面是Message.RecipientType的三个常量:

```
Message.RecipientType.TO
Message.RecipientType.CC
Message.RecipientType.BCC
```

因此，如果我们要发送邮件给总统，并发用一个副本给第一夫人的话，下面的方法将被用到：

Address toAddress = new InternetAddress("vice.president@whitehouse.gov");Address ccAddress = new InternetAddress("first.lady@whitehouse.gov");message.addRecipient(Message.RecipientType.TO, toAddress);message.addRecipient(Message.RecipientType.CC, ccAddress);

JavaMail API并没有提供检查邮件地址有效性的机制。当然我们可以自己完成这个功能：验证邮件地址的字符是否按照RFC822规定的格式书写或者通过DNS服务器上的MX记录验证等。

E. Authenticator

像java.net类那样，JavaMail API通过使用授权者类（Authenticator）以用户名、密码的方式访问那些受到保护的资源，在这里“资源”就是指邮件服务器。在javax.mail包中可以找到这个JavaMail的授权者类（Authenticator）。

在使用Authenticator这个抽象类时，我们必须采用继承该抽象类的方式，并且该继承类必须具有返回PasswordAuthentication对象（用于存储认证时要用到的用户名、密码）getPasswordAuthentication()方法。并且要在Session中进行注册，使Session能够了解在认证时该使用哪个类。

下面代码片断中的MyAuthenticator就是一个Authenticator的子类。

Properties props = new Properties();// fill props with any informationAuthenticator auth = new MyAuthenticator();Session session = Session.getDefaultInstance(props, auth);

F. Transport

在发送信息时，Transport类将被用到。这个类实现了发送信息的协议（通称为SMTP），此类是一个抽象类，我们可以使用这个类的静态方法send()来发送消息：

Transport.send(message);

当然，方法是多样的。我们也可由Session获得相应协议对应的Transport实例。并通过传递用户名、密码、邮件服务器主机名等参数建立与邮件服务器的连接，并使用sendMessage()方法将信息发送，最后关闭连接：

message.saveChanges(); // implicit with send()Transport transport = session.getTransport("smtp");transport.connect(host, username, password);transport.sendMessage(message, message.getAllRecipients());transport.close();

评论：上面的方法是一个很好的方法，尤其是在我们在同一个邮件服务器上发送多个邮件时。因为这时我们将在连接邮件服务器后连续发送邮件，然后再关闭掉连接。send()这个基本的方法是在每次调用时进行与邮件服务器的连接的，对于在同一个邮件服务器上发送多个邮件来讲可谓低效的方式。

注意：如果需要在发送邮件过程中监控mail命令的话，可以在发送前设置debug标志：

session.setDebug(true)。

G. Store和Folder

接收邮件和发送邮件很类似都要用到Session。但是在获得Session后，我们需要从Session中获取特定类型的Store，然后连接到Store，这里的Store代表了存储邮件的邮件服务器。在连接Store的过程中，极有可能需要用到用户名、密码或者Authenticator。

// Store store = session.getStore("imap");Store store = session.getStore("pop3");store.connect(host, username, password);

在连接到Store后，一个Folder对象即目录对象将通过Store的getFolder()方法被返回，我们可从这个Folder中读取邮件信息：

Folder folder = store.getFolder("INBOX");folder.open(Folder.READ_ONLY);Message message[] = folder.getMessages();

上面的例子首先从Store中获得INBOX这个Folder（对于POP3协议只有一个名为INBOX的Folder有效），然后以只读（Folder.READ_ONLY）的方式打开Folder，最后调用Folder的getMessages()方法得到目录中所有Message的数组。

注意：对于POP3协议只有一个名为INBOX的Folder有效，而对于IMAP协议，我们可以访问多个Folder（想想前面讲的IMAP协议）。而且SUN在设计Folder的getMessages()方法时采取了很智能的方式：首先接收新邮件列表，然后再需要的时候（比

如读取邮件内容）才从邮件服务器读取邮件内容。

在读取邮件时，我们可以用Message类的getContent()方法接收邮件或是writeTo()方法将邮件保存，getContent()方法只接收邮件内容（不包含邮件头），而writeTo()方法将包括邮件头。

System.out.println((MimeMessage)message).getContent());

在读取邮件内容后，别忘记了关闭Folder和Store。

folder.close(aBoolean);store.close();

传递给Folder.close()方法的boolean 类型参数表示是否在删除操作邮件后更新Folder。

H. 继续向前进！

在讲解了以上的七个Java Mail核心类定义和理解了简单的代码片断后，下文将详细讲解怎样使用这些类实现JavaMail API所要完成的高级功能。

五、使用JavaMail API

在明确了JavaMail API的核心部分如何工作后，本人将带领大家学习一些使用Java Mail API任务案例。

1. 发送邮件

在获得了Session后，建立并填入邮件信息，然后发送它到邮件服务器。这便是使用Java Mail API发送邮件的过程，在发送邮件之前，我们需要设置SMTP服务器：通过设置Properties的mail.smtp.host属性。

String host = ...;String from = ...;String to = ...;// Get system propertiesProperties props = System.getProperties();// Setup mail serverprops.put("mail.smtp.host", host);// Get sessionSession session = Session.getDefaultInstance(props, null);// Define messageMimeMessage message = new MimeMessage(session);message.setFrom(new InternetAddress(from));message.addRecipient(Message.RecipientType.TO, new InternetAddress(to));message.setSubject("Hello JavaMail");message.setText("Welcome to JavaMail");// Send messageTransport.send(message);

由于建立邮件信息和发送邮件的过程中可能会抛出异常，所以我们需要将上面的代码放入到try-catch结构块中。

2. 接收邮件

为了在读取邮件，我们获得了session，并且连接到了邮箱的相应store，打开相应的Folder，然后得到我们想要的邮件，当然别忘记了在结束时关闭连接。

String host = ...;String username = ...;String password = ...;// Create empty propertiesProperties props = new Properties();// Get sessionSession session = Session.getDefaultInstance(props, null);// Get the storeStore store = session.getStore("pop3");store.connect(host, username, password);// Get folderFolder folder = store.getFolder("INBOX");folder.open(Folder.READ_ONLY);// Get directoryMessage message[] = folder.getMessages();for (int i=0, n=message.length; i<N; I++) {   SYSTEM.OUT.PRINTLN(I + ": " + MESSAGE[I].GETFROM()[0]   + "\T" + MESSAGE[I].GETSUBJECT());} Close connection folder.close(false);store.close();

上面的代码所作的是从邮箱中读取每个邮件，并且显示邮件的发信人地址和主题。从技术角度讲，这里存在着一个异常的可能：当发信人地址为空时，getFrom()[0]将抛出异常。

下面的代码片断有效的说明了如何读取邮件内容，在显示每个邮件发信人和主题后，将出现用户提示从而得到用户是否读取该邮件的确认，如果输入YES的话，我们可用Message.writeTo(java.io.OutputStream os)方法将邮件内容输出到控制台上，关于Message.writeTo()的具体用法请看JavaMail API。

BufferedReader reader = new BufferedReader (new InputStreamReader(System.in));// Get directoryMessage message[] = folder.getMessages();for (int i=0, n=message.length; i<N; I++) { SYSTEM.OUT.PRINTLN(I + ": " + MESSAGE[I].GETFROM()[0]   + "\T" + MESSAGE[I].GETSUBJECT()); SYSTEM.OUT.PRINTLN("DO YOU WANT TO READ

MESSAGE? " +   "[YES TO READ QUIT to end]?); String line =' reader.readLine(); if ("YES".equals(line)) { message[i].writeTo(System.out); } else if ("QUIT".equals(line)) { break; }}

3. 删除邮件和标志

设置与message相关的Flags是删除邮件的常用方法。这些Flags表示了一些系统定义和用户定义的不同状态。在Flags类的内部类Flag中预定义了一些标志：

Flags.Flag.ANSWERED

Flags.Flag.DELETED

Flags.Flag.DRAFT

Flags.Flag.FLAGGED

Flags.Flag.RECENT

Flags.Flag.SEEN

Flags.Flag.USER

但需要在使用时注意的：标志存在并非意味着这个标志被所有的邮件服务器所支持。例如，对于删除邮件的操作，POP协议不支持上面的任何一个。所以要确定哪些标志是被支持的??通过访问一个已经打开的Folder对象的getPermanetFlags()方法，它将返回当前被支持的Flags类对象。

删除邮件时，我们可以设置邮件的DELETED标志：

message.setFlag(Flags.Flag.DELETED, true);

但是首先要采用READ_WRITE的方式打开Folder：

folder.open(Folder.READ_WRITE);

在对邮件进行删除操作后关闭Folder时，需要传递一个true作为对删除邮件的擦除确认。

folder.close(true);

Folder类中另一种用于删除邮件的方法expunge()也同样可删除邮件，但是它并不为sun提供的POP3实现支持，而其它第三方提供的POP3实现支持或者并不支持这种方法。

另外，介绍一种检查某个标志是否被设置的方法：Message.isSet(Flags.Flag flag)方法，其中参数为被检查的标志。

4. 邮件认证

我们在前面已经学会了如何使用 Authenticator 类来代替直接使用用户名和密码这两字符串作为Session.getDefaultInstance()或者Session.getInstance()方法的参数。在前面的小试牛刀后，现在我们将了解到全面认识一下邮件认证。

我们在此取代了直接使用邮件服务器主机名、用户名、密码这三个字符串作为连接到POP3 Store的方式，使用存储了邮件服务器主机名信息的属性文件，并在获得Session时传入自定义的Authenticator实例：

// Setup propertiesProperties props = System.getProperties();props.put("mail.pop3.host", host);// Setup authentication, get sessionAuthenticator auth = new PopupAuthenticator();Session session = Session.getDefaultInstance(props, auth);// Get the storeStore store = session.getStore("pop3");store.connect();

PopupAuthenticator类继承了抽象类Authenticator，并且通过重载Authenticator类的getPasswordAuthentication()方法返回PasswordAuthentication类对象。而getPasswordAuthentication()方法的参数param是以逗号分割的用户名、密码组成的字符串。

import javax.mail.*;import java.util.*;public class PopupAuthenticator extends Authenticator { public PasswordAuthentication getPasswordAuthentication(String param) { String username, password; StringTokenizer st = new StringTokenizer(param, ","); username = st.nextToken(); password = st.nextToken(); return new PasswordAuthentication(username, password); }}

5. 回复邮件

回复邮件的方法很简单：使用Message类的reply()方法，通过配置回复邮件的收件人地址和主题（如果没有提供主题的话，系统将默认将“Re：”作为邮件的主体），这里不需要设置任何的邮件内容，只要复制发信人或者reply-to到新的收件人。而reply()方法中的boolean参数表示是否将邮件回复给发送者（参数值为false），或是恢复给所有人（参数值为true）。
补充一下，reply-to地址需要在发信时使用setReplyTo()方法设置。

```
MimeMessage reply = (MimeMessage)message.reply(false);reply.setFrom(new InternetAddress("president@whitehouse.gov"));reply.setText("Thanks");Transport.send(reply);
```

6. 转发邮件

转发邮件的过程不如前面的回复邮件那样简单，它将建立一个转发邮件，这并非一个方法就能做到。
每个邮件是由多个部分组成，每个部分称为一个邮件体部分，是一个BodyPart类对象，对于MIME类型邮件来讲就是MimeBodyPart类对象。这些邮件体包含在成为Multipart的容器中对于MIME类型邮件来讲就是MimeMultiPart类对象。在转发邮件时，我们建立一个文字邮件体部分和一个被转发的文字邮件体部分，然后将这两个邮件体放到一个Multipart中。说明一下，复制一个邮件内容到另一个邮件的方法是仅复制它的DataHandler（数据处理者）即可。这是由JavaBeans Activation Framework定义的一个类，它提供了对邮件内容的操作命令的访问、管理了邮件内容操作，是不同的数据源和数据格式之间的一致性接口。

```
// Create the message to forwardMessage forward = new MimeMessage(session);// Fill in headerforward.setSubject("Fwd: " + message.getSubject());forward.setFrom(new InternetAddress(from));forward.addRecipient(Message.RecipientType.TO,    new InternetAddress(to));// Create your new message partBodyPart messageBodyPart = new MimeBodyPart();messageBodyPart.setText(  "Here you go with the original message:\n\n");// Create a multi-part to combine the partsMultipart multipart = new MimeMultipart();multipart.addBodyPart(messageBodyPart);// Create and fill part for the forwarded contentmessageBodyPart = new MimeBodyPart();messageBodyPart.setDataHandler(message.getDataHandler());// Add part to multi partmultipart.addBodyPart(messageBodyPart);// Associate multi-part with messageforward.setContent(multipart);// Send messageTransport.send(forward);
```

7. 使用附件

附件作为与邮件相关的资源经常以文本、表格、图片等格式出现，如流行的邮件客户端一样，我们可以用JavaMail API从邮件中获取附件或是发送带有附件的邮件。

A. 发送带有附件的邮件

发送带有附件的邮件的过程有些类似转发邮件，我们需要建立一个完整邮件的各个邮件体部分，在第一个部分（即我们的邮件内容文字）后，增加一个具有DataHandler的附件而不是在转发邮件时那样复制第一个部分的DataHandler。

如果我们将文件作为附件发送，那么要建立FileDataSource类型的对象作为附件数据源；如果从URL读取数据作为附件发送，那么将要建立URLDataSource类型的对象作为附件数据源。

然后将这个数据源（FileDataSource或是URLDataSource）对象作为DataHandler类构造方法的参数传入，从而建立一个DataHandler对象作为数据源的DataHandler。

接着将这个DataHandler设置为邮件体部分的DataHandler。这样就完成了邮件体与附件之间的关联工作，下面的工作就是BodyPart的setFileName()方法设置附件名为原文件名。

最后将两个邮件体放入到Multipart中，设置邮件内容为这个容器Multipart，发送邮件。

```
// Define messageMessage message = new MimeMessage(session);message.setFrom(new InternetAddress(from));message.addRecipient(Message.RecipientType.TO,    new InternetAddress(to));message.setSubject("Hello JavaMail Attach
```

ment");// Create the message part BodyPart messageBodyPart = new MimeBodyPart();// Fill the messagemessageBodyPart.setText("Pardon Ideas");Multipart multipart = new MimeMultipart();multipart.addBodyPart(messageBodyPart);// Part two is attachmentmessageBodyPart = new MimeBodyPart();DataSource source = new FileDataSource(filename);messageBodyPart.setDataHandler(new DataHandler(source));messageBodyPart.setFileName(filename);multipart.addBodyPart(messageBodyPart);// Put parts in messagemessage.setContent(multipart);// Send the messageTransport.send(message);

如果我们使用servlet实现发送带有附件的邮件，则必须上传附件给servlet，这时需要注意提交页面form中对编码类型的设置应为multipart/form-data。

<FORM ENCTYPE="MULTIPART form-data? method='post action="/myservlet"'> <INPUT TYPE="FILE" NAME="THEFILE"> <INPUT TYPE="SUBMIT" VALUE="UPLOAD">

窗体底端

B. 读取邮件中的附件

读取邮件中的附件的过程要比发送它的过程复杂一点。因为带有附件的邮件是多部分组成的，我们必须处理每一个部分获得邮件的内容和附件。

但是如何辨别邮件信息内容和附件呢？Sun在Part类（BodyPart类实现的接口类）中提供了getDisposition()方法让开发者获得邮件体部分的部署类型，当该部分是附件时，其返回之将是Part.ATTACHMENT。但附件也可以没有部署类型的方式存在或者部署类型为Part.INLINE，无论部署类型为Part.ATTACHMENT还是Part.INLINE，我们都能把该邮件体部分导出保存。

Multipart mp = (Multipart)message.getContent();for (int i=0, n=multipart.getCount(); i<N; I++) { PART PART = MULTIPART.GETBODYPART(I)); STRING DISPOSITION = PART.GETDISPOSITION(); IF ((DISPOSITION != NULL) &&   ((DISPOSITION.EQUALS(PART.ATTACHMENT) || (DISPOSITION.EQUALS(PART.INLINE))) {   SAVEFILE(PART.GETFILENAME(), PART.GETINPUTSTREAM()); }}

下列代码中使用了saveFile方法是自定义的方法，它根据附件的文件名建立一个文件，如果本地磁盘上存在名为附件的文件，那么将在文件名后增加数字表示区别。然后从邮件体中读取数据写入到本地文件中（代码省略）。

// from saveFile()File file = new File(filename);for (int i=0; file.exists(); i++) { file = new File(filename+i);}

以上是邮件体部分被正确设置的简单例子，如果邮件体部分的部署类型为null，那么我们通过获得邮件体部分的MIME类型来判断其类型作相应的处理，代码结构框架如下：

if (disposition == null) { // Check if plain MimeBodyPart mbp = (MimeBodyPart)part; if (mbp.isMimeType("text/plain")) { // Handle plain } else { // Special non-attachment cases here of // image/gif, text/html, ... }...}

8. 处理HTML邮件

前面的例子中发送的邮件都是以文本为内容的（除了附件），下面将介绍如何接收和发送基于HTML的邮件。

A. 发送HTML邮件

假如我们需要发送一个HTML文件作为邮件内容，并使邮件客户端在读取邮件时获取相关的图片或者文字的话，只要设置邮件内容为html代码，并设置内容类型为text/html即可：

String htmlText = "

Hello

" + "";message.setContent(htmlText, "text/html"));

请注意：这里的图片并不是在邮件中内嵌的，而是在URL中定义的。邮件接收者只有在线时才能看到。

在接收邮件时，如果我们使用JavaMail API接收邮件的话是无法实现以HTML方式显示邮件内容的。因为JavaMail API邮件内容视为二进制流。所以要显示HTML内容的邮件，我们必须使用JEditorPane或者第三方HTML展现组件。

以下代码显示了如何使用JEditorPane显示邮件内容：

```
if (message.getContentType().equals("text/html")) { String content = (String)message.getContent(); JFrame f
rame = new JFrame(); JEditorPane text = new JEditorPane("text/html", content); text.setEditable(false); JS
crollPane pane = new JScrollPane(text); frame.getContentPane().add(pane); frame.setSize(300, 300); frame.s
etDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE); frame.show();}
```

B. 在邮件中包含图片

如果我们在邮件中使用HTML作为内容，那么最好将HTML中使用的图片作为邮件的一部分，这样无论是否在线都会正确的显示HTML中的图片。处理方法就是将HTML中用到的图片作为邮件附件并使用特殊的cid URL作为图片的引用，这个cid就是对图片附件的Content-ID头的引用。

处理内嵌图片就像向邮件中添加附件一样，不同之处在于我们必须通过设置图片附件所在的邮件体部分的header中Content-ID为一个随机字符串，并在HTML中img的src标记中设置为该字符串。这样就完成了图片附件与HTML的关联。

```
String file = ...;// Create the messageMessage message = new MimeMessage(session);// Fill its headersmessage.
setSubject("Embedded Image");message.setFrom(new InternetAddress(from));message.addRecipient(Message.Recipien
tType.TO, new InternetAddress(to));// Create your new message partBodyPart messageBodyPart = new MimeBodyPa
rt();String htmlText = "

Hello

" + "<CCID_FILE VALUES="\CID:MEMEMEME\" />";messageBodyPart.setContent(htmlText, "text/html");// Creat
e a related multi-part to combine the partsMimeMultipart multipart = new MimeMultipart("related");multipart.a
ddBodyPart(messageBodyPart);// Create part for the imagemessageBodyPart = new MimeBodyPart();// Fetch the ima
ge and associate to partDataSource fds = new FileDataSource(file);messageBodyPart.setDataHandler(new DataHand
ler(fds));messageBodyPart.setHeader("Content-ID","");// Add part to multi-
partmultipart.addBodyPart(messageBodyPart);// Associate multi-part with messagemessage.setContent(multipart);
```

9. 在邮件中搜索短语

JavaMail API提供了过滤器机制，它被用来建立搜索短语。这个短语由javax.mail.search包中的SearchTerm抽象类来定义，在定义后我们便可以使用Folder的Search()方法在Folder中查找邮件：

```
SearchTerm st = ...;Message[] msgs = folder.search(st);
```

下面有22个不同的类（继承了SearchTerm类）供我们使用：

AND terms (class AndTerm)

OR terms (class OrTerm)

NOT terms (class NotTerm)

SENT DATE terms (class SentDateTerm)

CONTENT terms (class BodyTerm)

HEADER terms (FromTerm / FromStringTerm, RecipientTerm / RecipientStringTerm, SubjectTerm, etc.)

使用这些类定义的断语集合，我们可以构造一个逻辑表达式，并在Folder中进行搜索。下面是一个实例：在Folder中搜索邮件主题含有“ADV”字符串或者发信人地址为friend@public.com的邮件。

```
SearchTerm st = new OrTerm( new SubjectTerm("ADV:"), new FromStringTerm("friend@public.com"));Messag
e[] msgs = folder.search(st);
```

六、参考资源

JavaMail API Home

Sun’s JavaMail API基础

JavaBeans Activation Framework Home

javamail-interest mailing list

Sun's JavaMail FAQ

jGuru's JavaMail FAQ

Third Party Products List

七、代码下载

http://java.sun.com/developer/onlineTraining/JavaMail/exercises.html

3.6 jxl.jar 包简介

下载地址

http://www.andykhan.com/jexcelapi/ 当前的最高版本是2.6。

作者的网站上对它的特征有如下描述：

● 支持Excel 95-2000的所有版本

● 生成Excel 2000标准格式

● 支持字体、数字、日期操作

● 能够修饰单元格属性

● 支持图像和图表

应该说以上功能已经能够大致满足我们的需要。最关键的是这套API是纯Java的，并不依赖Windows系统，即使运行在Linux下，它同样能够正确的处理Excel文件。另外需要说明的是，这套API对图形和图表的支持很有限，而且仅仅识别PNG格式。

搭建环境

将下载后的文件解包，得到jxl.jar，放入classpath，安装就完成了。

基本操作

一、创建文件

拟生成一个名为“测试数据.xls”的Excel文件，其中第一个工作表被命名为“第一页”，大致效果如下：

代码（CreateXLS.java）：

```
//生成Excel的类
import java.io.*;

import jxl.*;

import jxl.write.*;



public class CreateXLS {

     public static void main(String args[]) {

          try {

               // 打开文件

               WritableWorkbook book = Workbook.createWorkbook(new File("测试.xls"));

               // 生成名为“第一页”的工作表，参数0表示这是第一页
```

```
                WritableSheet sheet = book.createSheet("第一页", 0);
                // 在Label对象的构造子中指名单元格位置是第一列第一行(0,0)
                // 以及单元格内容为test
                Label label = new Label(0, 0, "test");
                // 将定义好的单元格添加到工作表中
                sheet.addCell(label);
                /*
                 * 生成一个保存数字的单元格 必须使用Number的完整包路径，否则有语法歧义 单元格位置是第二列，第一行，值为789.123
                 */
                jxl.write.Number number = new jxl.write.Number(1, 0, 789.123);
                sheet.addCell(number);
                // 写入数据并关闭文件
                book.write();
                book.close();
            } catch (Exception e) {
                System.out.println(e);
            }
        }
}
```

编译执行后，会在当前位置产生一个Excel文件。

三、读取文件

以刚才我们创建的Excel文件为例，做一个简单的读取操作，程序代码如下：

```
//读取Excel的类
import java.io.*;
import jxl.*;

public class ReadXLS {
    public static void main(String args[]) {
        try {
            Workbook book = Workbook.getWorkbook(new File("测试.xls"));
            // 获得第一个工作表对象
```

```
            Sheet sheet = book.getSheet(0);
            // 得到第一列第一行的单元格
            Cell cell1 = sheet.getCell(0, 0);
            String result = cell1.getContents();
            System.out.println(result);
            book.close();
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

程序执行结果：test

四、修改文件

利用 jExcelAPI 可以修改已有的 Excel 文件，修改 Excel 文件的时候，除了打开文件的方式不同之外，其他操作和创建 Excel 是一样的。下面的例子是在我们已经生成的 Excel 文件中添加一个工作表：

```
//修改Excel的类，添加一个工作表
import java.io.*;
import jxl.*;
import jxl.write.*;

public class UpdateXLS {
    public static void main(String args[]) {
        try {
            // Excel获得文件
            Workbook wb = Workbook.getWorkbook(new File("测试.xls"));
            // 打开一个文件的副本，并且指定数据写回到原文件
            WritableWorkbook book = Workbook.createWorkbook(new File("测试.xls"),
                    wb);
            // 添加一个工作表
            WritableSheet sheet = book.createSheet("第二页", 1);
            sheet.addCell(new Label(0, 0, "第二页的测试数据"));
            book.write();
```

```
                book.close();
            } catch (Exception e) {
                System.out.println(e);
            }
        }
}
```

高级操作

一、 数据格式化

在Excel中不涉及复杂的数据类型，能够比较好的处理字串、数字和日期已经能够满足一般的应用。

1、 字串格式化

字符串的格式化涉及到的是字体、粗细、字号等元素，这些功能主要由WritableFont和WritableCellFormat类来负责。假设我们在生成一个含有字串的单元格时，使用如下语句，为方便叙述，我们为每一行命令加了编号：

```
WritableFont font1=
new WritableFont(WritableFont.TIMES,16,WritableFont.BOLD); 或//设置字体格式为 excel 支持的格式 WritableFont font3=new WritableFont(WritableFont.createFont("楷体_GB2312"),12,WritableFont.NO_BOLD );① WritableCellFormat format1=new WritableCellFormat(font1); ② Label label=new Label(0,0,” data 4 test”,format1) ⑧ 其中①指定了字串格式：字体为TIMES，字号16，加粗显示。WritableFont有非常丰富的构造子，供不同情况下使用，jExcelAPI的java-doc中有详细列表，这里不再列出。 ②处代码使用了WritableCellFormat类，这个类非常重要，通过它可以指定单元格的各种属性，后面的单元格格式化中会有更多描述。⑧处使用了Label类的构造子，指定了字串被赋予那种格式。在WritableCellFormat类中，还有一个很重要的方法是指定数据的对齐方式，比如针对我们上面的实例，可以指定：

 //把水平对齐方式指定为居中

format1.setAlignment(jxl.format.Alignment.CENTRE);

//把垂直对齐方式指定为居中

format1.setVerticalAlignment(jxl.format.VerticalAlignment.CENTRE);

//设置自动换行
 format1.setWrap(true);
```

二、单元格操作

Excel中很重要的一部分是对单元格的操作，比如行高、列宽、单元格合并等，所幸jExcelAPI提供了这些支持。这些操作相对比较简单，下面只介绍一下相关的API。

1、 合并单元格

WritableSheet.mergeCells(int m,int n,int p,int q);

作用是从(m,n)到(p,q)的单元格全部合并，比如：

```
WritableSheet sheet=book.createSheet(“第一页”,0);
//合并第一列第一行到第六列第一行的所有单元格
sheet.mergeCells(0,0,5,0);
```

合并既可以是横向的，也可以是纵向的。合并后的单元格不能再次进行合并，否则会触发异常。

2、 行高和列宽

WritableSheet.setRowView(int i,int height);

作用是指定第i+1行的高度，比如：

//将第一行的高度设为200

sheet.setRowView(0,200);

WritableSheet.setColumnView(int i,int width);

作用是指定第i+1列的宽度，比如：

//将第一列的宽度设为30

sheet.setColumnView(0,30);

五、操作图片

```
public static void write()throws Exception{
        WritableWorkbook wwb=Workbook.createWorkbook(new File("c:/1.xls"));
        WritableSheet ws=wwb.createSheet("Test Sheet 1",0);
        File file=new File("C:\\jbproject\\PVS\\WebRoot\\weekhit\\1109496996281.png");
        WritableImage image=new WritableImage(1, 4, 6, 18,file);
        ws.addImage(image);
        wwb.write();
        wwb.close();
    }
```

很简单和插入单元格的方式一样，不过就是参数多了些，WritableImage这个类继承了Draw，上面只是他构造方法的一种，最后一个参数不用了说了，前面四个参数的类型都是double，依次是 x, y, width, height,注意，这里的宽和高可不是图片的宽和高，而是图片所要占的单位格的个数，因为继承的Draw所以他的类型必须是double，具体里面怎么实现的我还没细看：）因为着急赶活，先完成功能，其他的以后有时间慢慢研究。以后会继续写出在使用中的心得给大家。

读:

读的时候是这样的一个思路,先用一个输入流(InputStream)得到Excel文件,然后用jxl中的Workbook得到工作薄,用Sheet从工作薄中得到工作表,用Cell得到工作表中得某个单元格.

InputStream->Workbook->Sheet->Cell,就得到了excel文件中的单元格

代码:

```
String path="c:\\excel.xls";//Excel文件URL
InputStream is = new FileInputStream(path);//写入到FileInputStream
jxl.Workbook wb = Workbook.getWorkbook(is); //得到工作薄
jxl.Sheet st = wb.getSheet(0);//得到工作薄中的第一个工作表
Cell cell=st.getCell(0,0);//得到工作表的第一个单元格,即A1
String content=cell.getContents();//getContents()将Cell中的字符转为字符串
wb.close();//关闭工作薄
is.close();//关闭输入流
```

我们可以通过Sheet的getCell(x,y)方法得到任意一个单元格,x,y和excel中的坐标对应.

例如A1对应(0,0),A2对应(0,1),D3对应(3,2).Excel中坐标从A,1开始,jxl中全部是从0开始.

还可以通过Sheet的getRows(),getColumns()方法得到行数列数,并用于循环控制,输出一个sheet中的所有内容.

写:

往Excel中写入内容主要是用jxl.write包中的类.

思路是这样的:

OutputStream←WritableWorkbook←WritableSheet←Label

这里面Label代表的是写入Sheet的Cell位置及内容.

代码:

```
OutputStream os=new FileOutputStream("c:\\test.xls");//输出的Excel文件URL
WritableWorkbook wwb = Workbook.createWorkbook(os);//创建可写工作薄
WritableSheet ws = wwb.createSheet("sheet1", 0);//创建可写工作表
Label labelCF=new Label(0, 0, "hello");//创建写入位置和内容
ws.addCell(labelCF);//将Label写入sheet中
```

Label的构造函数Label(int x, int y,String aString)xy意同读的时候的xy,aString是写入的内容.

```
WritableFont wf = new WritableFont(WritableFont.TIMES, 12, WritableFont.BOLD, false);//设置写入字体
WritableCellFormat wcfF = new WritableCellFormat(wf);//设置CellFormat
Label labelCF=new Label(0, 0, "hello");//创建写入位置,内容和格式
```

Label的另一构造函数Label(int c, int r, String cont, CellFormat st)可以对写入内容进行格式化,设置字体及其它的属性.

现在可以写了

```
wwb.write();
```

写完后关闭

```
wwb.close();
```

输出流也关闭吧

```
os.close;
```

OK,只要把读和写结合起来,就可以在N个Excel中读取数据写入你希望的Excel新表中,还是比较方便的.

3.7 Java与XML联合编程之SAX篇

SAX概念

SAX是Simple API for XML的缩写，它并不是由W3C官方所提出的标准，可以说是“民间”的事实标准。实际上，它是一种社区性质的讨论产物。虽然如此，在XML中对SAX的应用丝毫不比DOM少，几乎所有的XML解析器都会支持它。

与DOM比较而言，SAX是一种轻量型的方法。我们知道，在处理DOM的时候，我们需要读入整个的XML文档，然后在内存中创建DOM树，生成DOM树上的每个Node对象。当文档比较小的时候，这不会造成什么问题，但是一旦文档大起来，处理DOM就会变得相当费时费力。特别是其对于内存的需求，也将是成倍的增长，以至于在某些应用中使用DOM是一件很不划算的事（比如在applet中）。这时候，一个较好的替代解决方法就是SAX。

SAX在概念上与DOM完全不同。首先，不同于DOM的文档驱动，它是事件驱动的，也就是说，它并不需要读入整个文档，而文档的读入过程也就是SAX的解析过程。所谓事件驱动，是指一种基于回调（callback）机制的程序运行方法。（如果你对Java新的代理事件模型比较清楚的话，就会很容易理解这种机制了）

在XMLReader接受XML文档，在读入XML文档的过程中就进行解析，也就是说读入文档的过程和解析的过程是同时进行的，这和DOM区别很大。解析开始之前,需要向XMLReader注册一个ContentHandler,也就是相当于一个事件监听器,在ContentHandler中定义了很多方法，比如startDocument()，它定制了当在解析过程中，遇到文档开始时应该处理的事情。当XMLReader读到合适的内容，就会抛出相应的事件，并把这个事件的处理权代理给ContentHandler，调用其相应的方法进行响应。

这样泛泛的说来或许有些不容易理解，别急，后面的例子会让你明白SAX的解析过程。看看这个简单XML文件：

Ogden Nash

Adam

当 XMLReader读到标签时，就会调用 ContentHandler.startElement()方法，并把标签名POEM作为参数传递过去。在你实现的startElement()方法中需要做相应的动作，以处理当出现时应该做的事情。各个事件随着解析的过程（也就是文档读入的过程）一个个顺序的被抛出，相应的方法也会被顺序的调用，最后，当解析完成，方法都被调用后，对文档的处理也就完成了。下面的这个表，列出了在解析上面的那个XML文件的时候，顺序被调用的方法：

遇到的项目　　方法回调

{文档开始}　　startDocument()

　　startElement(null,"POEM",null,{Attributes})

"\n"　　characters("\n...", 6, 1)

　　startElement(null,"AUTHOR",null,{Attributes})

"Ogden Nash"　　characters("\n...", 15, 10)

　　endElement(null,"AUTHOR",null)

"\n"　　characters("\n...", 34, 1)

　　endElement(null,"TITLE",null)

"\n"　　characters("\n...", 55, 1)

　　startElement(null,"LINE",null,{Attributes})

"Adam"　　characters("\n...", 62, 4)

　　endElement(null,"LINE",null)

"\n"　　characters("\n...", 67, 1)

　　endElement(null,"POEM",null)

{文档结束}　　endDocument()

ContentHandler实际上是一个接口，当处理特定的XML文件的时候，就需要为其创建一个实现了ContentHandler的类来处理特定的事件，可以说，这个实际上就是SAX处理XML文件的核心。下面我们来看看定义在其中的一些方法：

void characters(char[] ch, int start, int length)：

这个方法用来处理在XML文件中读到字符串，它的参数是一个字符数组，以及读到的这个字符串在这个数组中的起始位置和长度，我们可以很容易的用String类的一个构造方法来获得这个字符串的String类：String charEncontered=new String(ch,start,length)。

void startDocument()：

当遇到文档的开头的时候，调用这个方法，可以在其中做一些预处理的工作。

void endDocument()：

和上面的方法相对应，当文档结束的时候，调用这个方法，可以在其中做一些善后的工作。

void startElement(java.lang.String namespaceURI, java.lang.String localName, java.lang.String qName, Attributes atts)

当读到一个开始标签的时候，会触发这个方法。在SAX1.0版本中并不支持名域，而在新的2.0版本中提供了对名域的支持，这儿参数中的namespaceURI就是名域，localName是标签名，qName是标签的修饰前缀，当没有使用名域的时候，这两个参数都未null。而atts是这个标签所包含的属性列表。通过atts，可以得到所有的属性名和相应的值。要注意的是SAX中一个重要的特点就是它的流式处理，在遇到一个标签的时候，它并不会纪录下以前所碰到的标签，也就是说，在startElement()方法中，所有你所知道的信息，就是标签的名字和属性，至于标签的嵌套结构，上层标签的名字，是否有子元属等等其它与结构相关的信息，都是不得而知的，都需要你的程序来完成。这使得SAX在编程处理上没有DOM来得那么方便。

void endElement(java.lang.String namespaceURI, java.lang.String localName, java.lang.String qName)

这个方法和上面的方法相对应，在遇到结束标签的时候，调用这个方法。

因为ContentHandler是一个接口，在使用的时候可能会有些不方便，因而SAX中还为其制定了一个Helper类：DefaultHandler，它实现了这个接口，但是其所有的方法体都为空，在实现的时候，你只需要继承这个类，然后重载相应的方法即可。

到这儿SAX的基本知识已经差不多讲完了，下面我们来看看两个具体的例子，以更好的理解SAX地用法。

SAX编程实例

我们还是沿用讲DOM的时候使用的那个文档例子，但首先，我们先看一个简单一些的应用，我们希望能够统计一下XML文件中各个标签出现的次数。这个例子很简单，但是足以阐述SAX编程的基本思路了。

一开始当然还是import语句了：

```
import org.xml.sax.helpers.DefaultHandler;

import javax.xml.parsers.*;

import org.xml.sax.*;

import org.xml.sax.helpers.*;

import java.util.*;

import java.io.*;
```

然后，我们创建一个继承于DefaultHandler的类，具体的程序逻辑在这儿可以暂且放在一边，要注意的是程序的结构：

```
public class SAXCounter extends DefaultHandler {

private Hashtable tags; //这个Hashtable用来记录tag出现的次数

// 处理文档前的工作

public void startDocument() throws SAXException {

tags = new Hashtable();//初始化Hashtable

}

//对每一个开始元属进行处理

public void startElement(String namespaceURI, String localName,

String rawName, Attributes atts)

throws SAXException

{

String key = localName;

Object value = tags.get(key);

if (value == null) {

// 如果是新碰到的标签，这在Hastable中添加一条记录
```

```
tags.put(key, new Integer(1));

} else {

// 如果以前碰到过，得到其计数值，并加1

int count = ((Integer)value).intValue();

count++;

tags.put(key, new Integer(count));

}

}

//解析完成后的统计工作

public void endDocument() throws SAXException {

Enumeration e = tags.keys();

while (e.hasMoreElements()) {

String tag = (String)e.nextElement();

int count = ((Integer)tags.get(tag)).intValue();

System.out.println("Tag <" + tag + "> occurs " + count

+ " times");

}

}

//程序入口，用来完成解析工作

static public void main(String[] args) {

String filename = null;

boolean validation = false;
```

```
filename="links.xml";

SAXParserFactory spf = SAXParserFactory.newInstance();

XMLReader xmlReader = null;

SAXParser saxParser=null;

try {

// 创建一个解析器SAXParser对象

saxParser = spf.newSAXParser();

// 得到SAXParser中封装的SAX XMLReader

xmlReader = saxParser.getXMLReader();

} catch (Exception ex) {

System.err.println(ex);

System.exit(1);

}

try {

//使用指定的ContentHandler，解析给XML文件，这儿要注意的是，为了

//程序的简单起见，这儿将主程序和ContentHandler放在了一起。实际上

//main方法中所作的所有事情，都与ContentHandler无关。

xmlReader.parse(new File(filename),new SAXCounter());

} catch (SAXException se) {

System.err.println(se.getMessage());

System.exit(1);

} catch (IOException ioe) {
```

```
System.err.println(ioe);

System.exit(1);

}

}

}
```

我们来看看这段程序作了些什么，在main()方法中，主要做的就是创建解析器，然后解析文档。实际上，在这儿创建SAXParser对象的时候，为了使程序代码于具体的解析器无关，使用了同DOM中一样的设计技巧：通过一个SAXParserFactory类来创建具体的SAXParser对象，这样，当需要使用不同的解析器的时候，要改变的，只是一个环境变量的值，而程序的代码可以保持不变。这就是FactoryMethod模式的思想。在这儿不再具体讲了，如果还有不明白的，可以参看上面DOM中的解释，原理是一样的。

不过在这儿还有一点点要注意的地方，就是SAXParser类和XMLReader类之间的关系。你可能有些迷糊了吧，实际上SAXParser是JAXP中对XMLReader的一个封装类，而XMLReader是定义在SAX2.0种的一个用来解析文档的接口。你可以同样的调用SAXParser或者XMLReader中的parser()方法来解析文档，效果是完全一样的。不过在SAXParser中的parser()方法接受更多的参数，可以对不同的XML文档数据源进行解析，因而使用起来要比XMLReader要方便一些。

这个例子仅仅涉及了SAX的一点皮毛，而下面的这个，可就要高级一些了。下面我们要实现的功能，在DOM的例子中已经有实现了，就是从XML文档中读出内容并格式化输出，虽然程序逻辑看起来还是很简单，但是SAX可不比DOM哦，看着吧。

前面说过，当遇到一个开始标签的时候，在startElement()方法中，我们并不能够得到这个标签在XML文档中所处的位置。这在处理XML文档的时候是个大麻烦，因为在XML中标签的语义，有一部分是由其所处的位置所决定的。而且在一些需要验证文档结构的程序中，这更是一个问题。当然，没有解决不了的问题了，我们可以使用一个栈来实现对文档结构的纪录。

栈的特点是先进先出，我们现在的想法是，在startElemnt()方法中用push将这个标签的名字添加到栈中，在endElement()方法中在把它pop出来。我们知道对一个结构良好的XML而言，其嵌套结构是完备的，每一个开始标签总会对应一个结束标签，而且不会出现标签嵌套之间的错位。因而，每一次startElement()方法的调用，必然会对应一个endElement()方法的调用，这样push和pop也是成对出现的，我们只需要分析栈的结构，就可以很容易的知道当前标签所处在文档结构中的位置了。

```
public class SAXReader extends DefaultHandler {

java.util.Stack tags=new java.util.Stack();

//————————XML Content————————

String text=null;

String url=null;

String author=null;
```

```
String description=null;

String day=null;

String year=null;

String month=null;

//————————————————————————————

public void endDocument() throws SAXException {

System.out.println("————Parse End————");

}

public void startDocument() throws SAXException {

System.out.println("————Parse Begin————");

}

public void startElement(String p0, String p1, String p2, Attributes p3) throws SAXException {

tags.push(p1);

}

public void endElement(String p0, String p1, String p2) throws SAXException {

tags.pop();

//一个link节点的信息收集齐了，将其格式化输出

if (p1.equals("link")) printout();

}

public void characters(char[] p0, int p1, int p2) throws SAXException {

//从栈中得到当前节点的信息

String tag=(String) tags.peek();
```

```
if (tag.equals("text")) text=new String(p0,p1,p2);

else if (tag.equals("url")) url=new String(p0,p1,p2);

else if (tag.equals("author")) author=new String(p0,p1,p2);

else if (tag.equals("day")) day=new String(p0,p1,p2);

else if (tag.equals("month")) month=new String(p0,p1,p2);

else if (tag.equals("year")) year=new String(p0,p1,p2);

else if (tag.equals("description")) year=new String(p0,p1,p2);

}

private void printout(){

System.out.print("Content: ");

System.out.println(text);

System.out.print("URL: ");

System.out.println(url);

System.out.print("Author: ");

System.out.println(author);

System.out.print("Date: ");

System.out.println(day+"-"+month+"-"+year);

System.out.print("Description: ");

System.out.println(description);

System.out.println();

}

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

```
String filename = null;

boolean validation = false;

filename="links.xml";

SAXParserFactory spf = SAXParserFactory.newInstance();

SAXParser saxParser=null;

try {

saxParser = spf.newSAXParser();

} catch (Exception ex) {

System.err.println(ex);

System.exit(1);

}

try {

saxParser.parse(new File(filename),new SAXReader());

} catch (SAXException se) {

System.err.println(se.getMessage());

System.exit(1);

} catch (IOException ioe) {

System.err.println(ioe);

System.exit(1);

}

}

}
```

在这儿虽然没有使用到栈的分析，但实际上栈的分析是一件很容易的事情，应为 java.util.Stack继承了 java.util.Vector 类，而且Stack 中的元素是按栈的结构由底至上排列的，因个，我们可以使用Vector 类的 size()方法来得到Stack 的元素个数，还可以使用Vector 的 get(int)方法来得到具体的每一个元属。实际上，如果把Stack 的元素从底向上逐一排列出来，我们就得到了从XML根节点到当前节点的一条唯一的路径，有了这条路径的信息，文档的结构就在清楚不过了。

小节

好了，到这儿为止，我们已经掌握了对于XML编程的两大利器：DOM和 SAX，也知道了该如何在一个Java 程序中使用它们。DOM 编程相对简单，但是速度比较慢，占用内存多，而SAX 编程复杂一些，但是速度快，占用内存少。所以，我们应该根据不同的环境选择使用不同的方法。大部分的 XML应用基本都可以用它们来解决。需要特别说明的是，DOM和 SAX其实都是语言无关的，并非 Java所独有，也就是说，只要有相应的语言实现，DOM和 SAX 可以应用在任何面向对象的语言中。

3.8 Java 与 XML 联合编程之 DOM 篇

DOM 是 Document Object Model 的缩写，即文档对象模型。前面说过，XML 将数据组织为一颗树，所以 DOM 就是对这颗树的一个对象描叙。通俗的说，就是通过解析XML 文档，为XML 文档在逻辑上建立一个树模型，树的节点是一个个对象。我们通过存取这些对象就能够存取 XML 文档的内容。

下面我们来看一个简单的例子，看看在DOM 中，我们是如何来操作一个 XML 文档的。

这是一个 XML 文档，也是我们要操作的对象：

Good-bye serialization, hello Java!

下面，我们需要把这个文档的内容解析到一个个的 Java 对象中去供程序使用，利用 JAXP，我们只需几行代码就能做到这一点。首先，我们需要建立一个解析器工厂，以利用这个工厂来获得一个具体的解析器对象：

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
```

我们在这里使用 DocumentBuilderFacotry 的目的是为了创建与具体解析器无关的程序，当 DocumentBuilderFactory 类的静态方法 newInstance()被调用时，它根据一个系统变量来决定具体使用哪一个解析器。又因为所有的解析器都服从于 JAXP 所定义的接口，所以无论具体使用哪一个解析器，代码都是一样的。所以当在不同的解析器之间进行切换时，只需要更改系统变量的值，而不用更改任何代码。这就是工厂所带来的好处。这个工厂模式的具体实现，可以参看下面的类图。

```
DocumentBuilder db = dbf.newDocumentBuilder();
```

当获得一个工厂对象后，使用它的静态方法 newDocumentBuilder()方法可以获得一个 DocumentBuilder 对象，这个对象代表了具体的 DOM 解析器。但具体是哪一种解析器，微软的或者 IBM 的，对于程序而言并不重要。

然后，我们就可以利用这个解析器来对 XML 文档进行解析了：

```
Document doc = db.parse("c:/xml/message.xml");
```

DocumentBuilder 的 parse()方法接受一个 XML 文档名作为输入参数，返回一个Document 对象，这个Document对象就代表了一个 XML文档的树模型。以后所有的对 XML文档的操作，都与解析器无关，直接在这个Document对象上进行操作就可以了。而具体对 Document操作的方法，就是由 DOM 所定义的了。

Jaxp支持 W3C所推荐的 DOM 2。如果你对DOM 很熟悉，那么下面的内容就很简单了：只需要按照DOM 的规范来进行方法调用就可以。当然，如果你对DOM 不清楚，也不用着急，后面我们会有详细的介绍。在这儿，你所要知道并牢记的是：DOM是用来描叙 XML 文档中的数据的模型，引入DOM 的全部原因就是为了用这个模型来操作XML 文档的中的数据。DOM 规范中定义有节点（即对象）、属性和方法，我们通过这些节点的存取来存取 XML 的数据。

从上面得到的 Document对象开始，我们就可以开始我们的DOM 之旅了。使用 Document对象的 getElementsByTagName()方法，我们可以得到一个 NodeList对象，一个Node 对象代表了一个 XML 文档中的一个标签元素，而NodeList 对象，观其名而知其意，所代表的是一个 Node 对象的列表：

```
NodeList nl = doc.getElementsByTagName("message");
```

我们通过这样一条语句所得到的是XML 文档中所有标签对应的 Node 对象的一个列表。然后，我们可以使用NodeList对象的 item()方法来得到列表中的每一个 Node对象：

```
Node my_node = nl.item(0);
```

当一个Node对象被建立之后，保存在XML 文档中的数据就被提取出来并封装在这个Node中了。在这个例子中，要提取Message标签内的内容，我们通常会使用Node 对象的getNodeValue()方法：

```
String message = my_node.getFirstChild().getNodeValue();
```

请注意，这里还使用了一个getFirstChild()方法来获得 message 下面的第一个子 Node 对象。虽然在message标签下面除了文本外并没有其它子标签或者属性，但是我们坚持在这里使用 getFirseChild()方法，这主要和W3C 对 DOM 的定义有关。W3C把标签内的文本部分也定义成一个 Node，所以先要得到代表文本的那个 Node，我们才能够使用getNodeValue()来获取文本的内容。

现在，既然我们已经能够从 XML 文件中提取出数据了，我们就可以把这些数据用在合适的地方，来构筑应用程序。

下面的内容，我们将更多的关注DOM，为DOM 作一个较为详细的解析，使我们使用起来更为得心应手。

DOM 详解

1. 基本的 DOM 对象

DOM 的基本对象有 5 个：Document，Node，NodeList，Element 和 Attr。下面就这些对象的功能和实现的方法作一个大致的介绍。

Document对象代表了整个 XML 的文档，所有其它的Node，都以一定的顺序包含在 Document对象之内，排列成一个树形的结构，程序员可以通过遍历这颗树来得到XML 文档的所有的内容，这也是对XML 文档操作的起点。我们总是先通过解析XML

源文件而得到一个 Document 对象，然后再来执行后续的操作。此外，Document 还包含了创建其它节点的方法，比如 createAttribut()用来创建一个 Attr 对象。它所包含的主要的方法有：

createAttribute(String)：用给定的属性名创建一个 Attr 对象，并可在其后使用 setAttributeNode 方法来放置在某一个 Element 对象上面。

createElement(String)：用给定的标签名创建一个 Element 对象，代表 XML 文档中的一个标签，然后就可以在这个 Element 对象上添加属性或进行其它的操作。

createTextNode(String)：用给定的字符串创建一个 Text 对象，Text 对象代表了标签或者属性中所包含的纯文本字符串。如果在一个标签内没有其它的标签，那么标签内的文本所代表的 Text 对象是这个 Element 对象的唯一子对象。

getElementsByTagName(String)：返回一个 NodeList 对象，它包含了所有给定标签名字的标签。

getDocumentElement()：返回一个代表这个 DOM 树的根节点的 Element 对象，也就是代表 XML 文档根元素的那个对象。

Node 对象是 DOM 结构中最为基本的对象，代表了文档树中的一个抽象的节点。在实际使用的时候，很少会真正的用到 Node 这个对象，而是用到诸如 Element、Attr、Text 等 Node 对象的子对象来操作文档。Node 对象为这些对象提供了一个抽象的、公共的根。虽然在 Node 对象中定义了对其子节点进行存取的方法，但是有一些 Node 子对象，比如 Text 对象，它并不存在子节点，这一点是要注意的。Node 对象所包含的主要的方法有：

appendChild(org.w3c.dom.Node)：为这个节点添加一个子节点，并放在所有子节点的最后，如果这个子节点已经存在，则先把它删掉再添加进去。

getFirstChild()：如果节点存在子节点，则返回第一个子节点，对等的，还有 getLastChild()方法返回最后一个子节点。

getNextSibling()：返回在 DOM 树中这个节点的下一个兄弟节点，对等的，还有 getPreviousSibling()方法返回其前一个兄弟节点。

getNodeName()：根据节点的类型返回节点的名称。

getNodeType()：返回节点的类型。

getNodeValue()：返回节点的值。

hasChildNodes()：判断是不是存在有子节点。

hasAttributes()：判断这个节点是否存在有属性。

getOwnerDocument()：返回节点所处的 Document 对象。

insertBefore(org.w3c.dom.Node new，org.w3c.dom.Node ref)：在给定的一个子对象前再插入一个子对象。

removeChild(org.w3c.dom.Node)：删除给定的子节点对象。

replaceChild(org.w3c.dom.Node new，org.w3c.dom.Node old)：用一个新的 Node 对象代替给定的子节点对象。

NodeList 对象，顾名思义，就是代表了一个包含了一个或者多个 Node 的列表。可以简单的把它看成一个 Node 的数组，我们可以通过方法来获得列表中的元素：

GetLength()：返回列表的长度。

Item(int)：返回指定位置的 Node 对象。

Element 对象代表的是 XML 文档中的标签元素，继承于 Node，亦是 Node 的最主要的子对象。在标签中可以包含有属性，因而 Element 对象中有存取其属性的方法，而任何 Node 中定义的方法，也可以用在 Element 对象上面。

getElementsByTagName(String)：返回一个 NodeList 对象，它包含了在这个标签中其下的子孙节点中具有给定标签名字的标签。

getTagName()：返回一个代表这个标签名字的字符串。

getAttribute(String)：返回标签中给定属性名称的属性的值。在这儿需要主要的是，应为 XML 文档中允许有实体属性出现，而这个方法对这些实体属性并不适用。这时候需要用到 getAttributeNodes()方法来得到一个 Attr 对象来进行进一步的操作。

getAttributeNode(String)：返回一个代表给定属性名称的 Attr 对象。

Attr 对象代表了某个标签中的属性。Attr 继承于 Node，但是因为 Attr 实际上是包含在 Element 中的，它并不能被看作是 Element 的子对象，因而在 DOM 中 Attr 并不是 DOM 树的一部分，所以 Node 中的 getparentNode()，getpreviousSibling()和 getnextSibling()返回的都将是 null。也就是说，Attr 其实是被看作包含它的 Element 对象的一部分，它并不作为 DOM 树中单独的一个节点出现。这一点在使用的时候要同其它的 Node 子对象相区别。

需要说明的是，上面所说的 DOM 对象在 DOM 中都是用接口定义的，在定义的时候使用的是与具体语言无关的 IDL 语言来定义的。因而，DOM 其实可以在任何面向对象的语言中实现，只要它实现了 DOM 所定义的接口和功能就可以了。同时，有些方法在 DOM 中并没有定义，是用 IDL 的属性来表达的，当被映射到具体的语言时，这些属性被映射为相应的方法。

2．DOM 实例

有了上面的介绍，相信你对 DOM 理解的更多了吧。下面的例子将让你对 DOM 更加熟悉起来。

先说说这个例子到底要做的是什么吧，我们希望在一个名为 link.xml 文件中保存了一些 URL 地址，通过一个简单的程序，我们可以通过 DOM 把这些 URL 读出并显示出来，也可以反过来向这个 XML 文件中写入加入的 URL 地址。很简单，却很实用，也足够来例示 DOM 的绝大部分用法了。

XML 文件本身不复杂，就不给出它的 DTD 了。link.xml:

JSP Insider

http://www.jspinsider.com

JSP Insider

2

1

2001

A JSP information site.

The makers of Java

http://java.sun.com

Sun Microsystems

3

1

2001

Sun Microsystem's website.

The standard JSP container

http://jakarta.apache.org

Apache Group

4

1

2001

Some great software.

第一个程序我们称为xmldisplay.java，具体的程序清单可以在附件中找到。主要的功能就是读取这个XML文件中各个节点的内容，然后在格式化输出在System.out上，我们来看看这个程序：

```
import javax.xml.parsers.*;

import org.w3c.dom.*;
```

这是引入必要的类，因为在这里使用的是Sun所提供的XML解析器，因而需要引入java.xml.parsers包，其中包含了有DOM解析器和SAX解析器的具体实现。org.w3c.dom包中定义了w3c所制定的DOM接口。

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();

DocumentBuilder builder=factory.newDocumentBuilder();

Document doc=builder.parse("links.xml");

doc.normalize();
```

除了上面讲到的，还有一个小技巧，对Document对象调用normalize()，可以去掉XML文档中作为格式化内容的空白而映射在DOM树中的不必要的Text Node对象。否则你得到的DOM树可能并不如你所想象的那样。特别是在输出的时候，这个normalize()更为有用。

```
NodeList links =doc.getElementsByTagName("link");
```

刚才说过，XML文档中的空白符也会被作为对象映射在DOM树中。因而，直接调用Node方法的getChildNodes方法有时候会有些问题，有时不能够返回所期望的NodeList对象。解决的办法是使用Element的getElementByTagName(String)，返回的

NodeLise就是所期待的对象了。然后，可以用item()方法提取想要的元素。

```
for (int i=0;i<LINKS.GETLENGTH();I++){

Element link=(Element) links.item(i);

System.out.print("Content: ");

System.out.println(link.getElementsByTagName("text").item(0).getFirstChild().getNodeValue());

System.out.print("URL: ");

System.out.println(link.getElementsByTagName("url").item(0).getFirstChild().getNodeValue());

System.out.print("Author: ");

System.out.println(link.getElementsByTagName("author").item(0).getFirstChild().getNodeValue());

System.out.print("Date: ");

Element linkdate=(Element) link.getElementsByTagName("date").item(0);

String day=linkdate.getElementsByTagName("day").item(0).getFirstChild().getNodeValue();

String month=linkdate.getElementsByTagName("month").item(0).getFirstChild().getNodeValue();

String year=linkdate.getElementsByTagName("year").item(0).getFirstChild().getNodeValue();

System.out.println(day+"-"+month+"-"+year);

System.out.print("Description: ");

System.out.println(link.getElementsByTagName("description").item(0).getFirstChild().getNodeValue());

System.out.println();

}
```

上面的代码片断就完成了对 XML 文档内容的格式化输出。只要注意到一些细节的问题，比如 getFirstChile()方法和getElementsByTagName()方法的使用，这些还是比较容易的。

下面的内容，就是在修改了DOM树后重新写入到XML文档中去的问题了。这个程序名为xmlwrite.java。在JAXP1.0版本中，并没有直接的类和方法能够处理XML文档的写入问题，需要借助其它包中的一些辅助类。而在JAXP1.1版本中，引入了对XSLT的支持，所谓XSLT，就是对XML文档进行变换（Translation）后，得到一个新的文档结构。利用这个新加入的功能，我们就

能够很方便的把新生成或者修改后的DOM树从新写回到XML文件中去了，下面我们来看看代码的实现，这段代码的主要功能是向links.xml文件中加入一个新的link节点：

```
import javax.xml.parsers.*;

import javax.xml.transform.*;

import javax.xml.transform.dom.DOMSource;

import javax.xml.transform.stream.StreamResult;

import org.w3c.dom.*;
```

新引入的java.xml.transform包中的几个类，就是用来处理XSLT变换的。

我们希望在上面的XML文件中加入一个新的link节点，因而首先还是要读入links.xml文件，构建一个DOM树，然后再对这个DOM树进行修改（添加节点），最后把修改后的DOM写回到links.xml文件中：

```
DocumentBuilderFactory  factory = DocumentBuilderFactory.newInstance();

DocumentBuilder builder=factory.newDocumentBuilder();

Document doc=builder.parse("links.xml");

doc.normalize();

//—取得变量—

String text="Hanzhong's Homepage";

String url="www.hzliu.com";

String author="Hzliu Liu";

String discription="A site from Hanzhong Liu, give u lots of suprise!!!";
```

为了看清重点，简化程序，我们把要加入的内容硬编码到记忆String对象中，而实际操作中，往往利用一个界面来提取用户输入，或者通过JDBC从数据库中提取想要的内容。

```
Text textseg;

Element link=doc.createElement("link");
```

首先应该明了的是，无论什么类型的Node，Text型的也好，Attr型的也好，Element型的也好，它们的创建都是通过Document

对象中的createXXX()方法来创建的（XXX代表具体要创建的类型），因此，我们要向XML文档中添加一个link项目，首先要创建一个link对象：

```
Element linktext=doc.createElement("text");

textseg=doc.createTextNode(text);

linktext.appendChild(textseg);

link.appendChild(linktext);

Element linkurl=doc.createElement("url");

textseg=doc.createTextNode(url);

linkurl.appendChild(textseg);

link.appendChild(linkurl);

Element linkauthor=doc.createElement("author");

textseg=doc.createTextNode(author);

linkauthor.appendChild(textseg);

link.appendChild(linkauthor);

java.util.Calendar rightNow = java.util.Calendar.getInstance();

String day=Integer.toString(rightNow.get(java.util.Calendar.DAY_OF_MONTH));

String month=Integer.toString(rightNow.get(java.util.Calendar.MONTH));

String year=Integer.toString(rightNow.get(java.util.Calendar.YEAR));

Element linkdate=doc.createElement("date");

Element linkdateday=doc.createElement("day");

textseg=doc.createTextNode(day);

linkdateday.appendChild(textseg);

Element linkdatemonth=doc.createElement("month");
```

```
textseg=doc.createTextNode(month);

linkdatemonth.appendChild(textseg);

Element linkdateyear=doc.createElement("year");

textseg=doc.createTextNode(year);

linkdateyear.appendChild(textseg);

linkdate.appendChild(linkdateday);

linkdate.appendChild(linkdatemonth);

linkdate.appendChild(linkdateyear);

link.appendChild(linkdate);

Element linkdiscription=doc.createElement("description");

textseg=doc.createTextNode(discription);

linkdiscription.appendChild(textseg);

link.appendChild(linkdiscription);
```

创建节点的过程可能有些千篇一律，但需要注意的地方是，对 Element 中所包含的 text（在 DOM 中，这些 text 也是代表了一个 Node 的，因此也必须为它们创建相应的 node），不能直接用 Element 对象的 setNodeValue()方法来设置这些 text 的内容，而需要用创建的 Text 对象的 setNodeValue()方法来设置文本，这样才能够把创建的 Element 和其文本内容添加到 DOM 树中。看看前面的代码，你会更好的理解这一点：

```
doc.getDocumentElement().appendChild(link);
```

最后，不要忘记把创建好的节点添加到 DOM 树中。Document 类的 getDocumentElement()方法，返回代表文档根节点的 Element 对象。在 XML 文档中，根节点一定是唯一的。

```
TransformerFactory tFactory =TransformerFactory.newInstance();

Transformer transformer = tFactory.newTransformer();

DOMSource source = new DOMSource(doc);

StreamResult result = new StreamResult(new java.io.File("links.xml"));
```

transformer.transform(source, result);

然后就是用 XSLT把 DOM 树输出了。这里的TransformerFactory 也同样应用了工厂模式，使得具体的代码同具体的变换器无关。实现的方法和DocumentBuilderFactory 相同，这儿就不赘述了。Transformer 类的 transfrom方法接受两个参数、一个数据源 Source和一个输出目标 Result。这里分别使用的是 DOMSource 和 StreamResult，这样就能够把DOM 的内容输出到一个输出流中，当这个输出流是一个文件的时候，DOM 的内容就被写入到文件中去了。

4 其他

4.1 代码复用的规则

代码复用是绝大多数程序员所期望的，也是OO的目标之一。总结我多年的编码经验，为了使代码能够最大程度上复用，应该特别注意以下几个方面。

对接口编程

"对接口编程"是面向对象设计（OOD）的第一个基本原则。它的含义是：使用接口和同类型的组件通讯，即，对于所有完成相同功能的组件，应该抽象出一个接口，它们都实现该接口。具体到JAVA中，可以是接口(interface)，或者是抽象类(abstract class），所有完成相同功能的组件都实现该接口，或者从该抽象类继承。我们的客户代码只应该和该接口通讯，这样，当我们需要用其它组件完成任务时，只需要替换该接口的实现，而我们代码的其它部分不需要改变！

当现有的组件不能满足要求时，我们可以创建新的组件，实现该接口，或者，直接对现有的组件进行扩展，由子类去完成扩展的功能。

优先使用对象组合，而不是类继承

"优先使用对象组合，而不是类继承"是面向对象设计的第二个原则。并不是说继承不重要，而是因为每个学习OOP的人都知道OO的基本特性之一就是继承，以至于继承已经被滥用了，而对象组合技术往往被忽视了。下面分析继承和组合的优缺点：

类继承允许你根据其他类的实现来定义一个类的实现。这种通过生成子类的复用通常被称为白箱复用(white-box reuse)。术语"白箱"是相对可视性而言：在继承方式中，父类的内部细节对子类可见。

对象组合是类继承之外的另一种复用选择。新的更复杂的功能可以通过组合对象来获得。对象组合要求对象具有良好定义的接口。这种复用风格被称为黑箱复用(black-box reuse)，因为被组合的对象的内部细节是不可见的。对象只以"黑箱"的形式出现。

继承和组合各有优缺点。类继承是在编译时刻静态定义的，且可直接使用，类继承可以较方便地改变父类的实现。但是类继承也有一些不足之处。首先，因为继承在编译时刻就定义了，所以无法在运行时刻改变从父类继承的实现。更糟的是，父类通常至少定义了子类的部分行为，父类的任何改变都可能影响子类的行为。如果继承下来的实现不适合解决新的问题，则父类必须重写或被其他更适合的类替换。这种依赖关系限制了灵活性并最终限制了复用性。

对象组合是通过获得对其他对象的引用而在运行时刻动态定义的。由于组合要求对象具有良好定义的接口，而且，对象只能通过接口访问，所以我们并不破坏封装性；只要类型一致，运行时刻还可以用一个对象来替代另一个对象；更进一步，因为对象的实现是基于接口写的，所以实现上存在较少的依赖关系。

优先使用对象组合有助于你保持每个类被封装，并且只集中完成单个任务。这样类和类继承层次会保持较小规模，并且不太可能增长为不可控制的庞然大物（这正是滥用继承的后果）。另一方面，基于对象组合的设计会有更多的对象(但只有较少的类)，且系统的行为将依赖于对象间的关系而不是被定义在某个类中。

注意：理想情况下，我们不用为获得复用而去创建新的组件，只需要使用对象组合技术，通过组装已有的组件就能获得需要的功能。但是事实很少如此，因为可用的组件集合并不丰富。使用继承的复用使得创建新的组件要比组装已有的组件来得容易。这样，继承和对象组合常一起使用。然而，正如前面所说，千万不要滥用继承而忽视了对象组合技术。

相关的设计模式有： Bridge、Composite、Decorator、Observer、Strategy等。

下面的例子演示了这个规则，它的前提是：我们对同一个数据结构，需要以任意的格式输出。

第一个例子，我们使用基于继承的框架，可以看到，它很难维护和扩展。

```
abstract class AbstractExampleDocument
{
    // skip some code ...
    public void output(Example structure)
        {
            if( null != structure )
                {
                this.format( structure );
                }
        }
    protected void format(Example structure);
}
```

第二个例子，我们使用基于对象组合技术的框架，每个对象的任务都清楚的分离开来，我们可以替换、扩展格式类，而不用考虑其它的任何事情。

```
class DefaultExampleDocument
{
  // skip some code ...
  public void output(Example structure)
  {
     ExampleFormatter formatter =
       (ExampleFormatter) manager.lookup(Roles.FORMATTER);
     if( null != structure )
     {
       formatter.format(structure);
     }
  }
}
```

这里，用到了类似于"抽象工厂"的组件创建模式，它将组件的创建过程交给manager来完成；ExampleFormatter是所有格式的抽象父类；

将可变的部分和不可变的部分分离

"将可变的部分和不可变的部分分离"是面向对象设计的第三个原则。如果使用继承的复用技术，我们可以在抽象基类中定义好不可变的部分，而由其子类去具体实现可变的部分，不可变的部分不需要重复定义，而且便于维护。如果使用对象组合的复用技术，我们可以定义好不可变的部分，而可变的部分可以由不同的组件实现，根据需要，在运行时动态配置。这样，我们就有更多的时间关注可变的部分。

对于对象组合技术而言，每个组件只完成相对较小的功能，相互之间耦合比较松散，复用率较高，通过组合，就能获得新的功能。

减少方法的长度

通常，我们的方法应该只有尽量少的几行，太长的方法会难以理解，而且，如果方法太长，则应该重新设计。对此，可以总结为以下原则：

☆ 三十秒原则：如果另一个程序员无法在三十秒之内了解你的函数做了什么（What），如何做(How)以及为什么要这样做(Why)，那就说明你的代码是难以维护的，必须得到提高；

☆ 一屏原则：如果一个函数的代码长度超过一个屏幕，那么或许这个函数太长了，应该拆分成更小的子函数；

☆ 一行代码尽量简短，并且保证一行代码只做一件事：那种看似技巧性的冗长代码只会增加代码维护的难度。

消除case / if语句

要尽量避免在代码中出现判断语句，来测试一个对象是否某个特定类的实例。通常，如果你需要这么做，那么，重新设计可能会有所帮助。我在工作中遇到这样的一个问题：我们在使用JAVA做XML解析时，对每个标签映射了一个JAVA类，采用SAX（简单的XML接口API：Simple API for XML）模型。结果，代码中反复出现了大量的判断语句，来测试当前的标签类型。为此，我们重新设计了DTD（文档类型定义：Document Type Definition），为每个标签增加了一个固定的属性：classname，而且重新设计了每个标签映射的JAVA类的接口，统一了每个对象的操作：

addElement(Element aElement); //增加子元素

addAttribute(String attName, String attValue); //增加属性；

则彻底消除了所有的测试当前的标签类型的判断语句。每个对象通过
Class.forName(aElement.attributes.getAttribute("classname")).newInstence(); 动态创建，

减少参数个数

有大量参数需要传递的方法，通常很难阅读。我们可以将所有参数封装到一个对象中来完成对象的传递，这也有利于错误跟踪。

许多程序员因为，太多层的对象包装对系统效率有影响。是的，但是，和它带来的好处相比，我们宁愿做包装。毕竟，"封装"也是OO的基本特性之一，而且，"每个对象完成尽量少（而且简单）的功能"，也是OO的一个基本原则。

类层次的最高层应该是抽象类

在许多情况下，提供一个抽象基类有利做特性化扩展。由于在抽象基类中，大部分的功能和行为已经定义好，使我们更容易理解接口设计者的意图是什么。

由于JAVA不允许"多继承"，从一个抽象基类继承，就无法再从其它基类继承了。所以，提供一个抽象接口(interface)是个好主意，一个类可以实现多个接口，从而模拟实现了"多继承"，为类的设计提供了更大的灵活性。

尽量减少对变量的直接访问

对数据的封装原则应该规范化，不要把一个类的属性暴露给其它类，而是应该通过访问方法去保护他们，这有利于避免产生波纹效应。如果某个属性的名字改变，你只需要修改它的访问方法，而不是修改所有相关的代码。

子类应该特性化，完成特殊功能

如果一个子类只是使一个组件变成组件管理器，而不是实现接口功能，或者，重载某个功能，那么，就应该使用一个外部的容器类，而不是创建一个子类。

建议：类层次结构图，不要太深；

例如：下面的接口定义了组件的功能：发送消息；类Transceiver实现了该接口；而其子类Pool只是管理多个Transceiver对象，而没有提供自己的接口实现。建议使用组合方式，而不是继承！

```
public interface ITransceiver{
  public abstract send(String msg);
}

public class Transceiver implements ITransceiver {
  public send(String msg){
    System.out.println(msg);
  }
}

//使用继承方式的实现
public class Pool extends Transceiver{
  private List  pool = new Vector();
  public void add(Transceiver aTransceiver){
    pool.add(aTransceiver);
}
public Transceiver get(int index){
    pool.get(index);
}
}
```

```
//使用组合方式的实现
public class Pool {
  private List  pool = new Vector();
  public void add(Transceiver aTransceiver){
    pool.add(aTransceiver);
}
public Transceiver get(int index){
    pool.get(index);
}
}
```

拆分过大的类

如果一个类有太多的方法（超过50个），那么它可能要做的工作太多，我们应该试着将它的功能拆分到不同的类中，类似于规则四。

作用截然不同的对象应该拆分

在构建的过程中，你有时会遇到这样的问题：对同样的数据，有不同的视图。某些属性描述的是数据结构怎样生成，而某些属性描述的是数据结构本身。最好将这两个视图拆分到不同的类中，从类名上就可以区分出不同视图的作用。

类的域、方法也应该有同样的考虑！

尽量减少对参数的隐含传递

两个方法处理类内部同一个数据（域），并不意味着它们就是对该数据（域）做处理。许多时候，该数据（域）应该作为方法的参输入数，而不是直接存取，在工具类的设计中尤其应该注意。例如：

```
public class Test{
  private List  pool = new Vector();
  public void testAdd(String str){
    pool.add(str);
}
public Object testGet(int index){
    pool.get(index);
}
}
```

两个方法都对List对象pool做了操作，但是，实际上，我们可能只是想对List接口的不同实现Vector、ArrayList等做存取测试。所以，代码应该这样写：

```
public class Test{
  private List  pool = new Vector();
  public void testAdd(List  pool, String str){
    pool.add(str);
}
public Object testGet(List  pool, int index){
    pool.get(index);
```

4.2 Java IO 包中的 Decorator 模式

JDK 为程序员提供了大量的类库，而为了保持类库的可重用性，可扩展性和灵活性，其中使用到了大量的设计模式，本文将介绍 JDK 的 I/O 包中使用到的Decorator 模式，并运用此模式，实现一个新的输出流类。

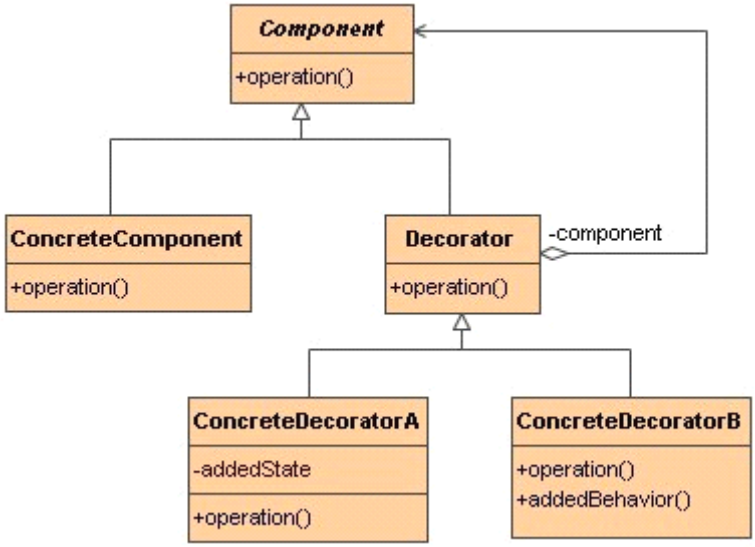
Decorator 模式简介

Decorator 模式又名包装器(Wrapper)，它的主要用途在于给一个对象动态的添加一些额外的职责。与生成子类相比，它更具有灵活性。

有时候，我们需要为一个对象而不是整个类添加一些新的功能，比如，给一个文本区添加一个滚动条的功能。我们可以使用继承机制来实现这一功能，但是这种方法不够灵活，我们无法控制文本区加滚动条的方式和时机。而且当文本区需要添加更多的功能时，比如边框等，需要创建新的类，而当需要组合使用这些功能时无疑将会引起类的爆炸。

我们可以使用一种更为灵活的方法，就是把文本区嵌入到滚动条中。而这个滚动条的类就相当于对文本区的一个装饰。这个装饰(滚动条)必须与被装饰的组件(文本区)继承自同一个接口，这样，用户就不必关心装饰的实现，因为这对他们来说是透明的。装饰会将用户的请求转发给相应的组件(即调用相关的方法)，并可能在转发的前后做一些额外的动作(如添加滚动条)。通过这种方法，我们可以根据组合对文本区嵌套不同的装饰，从而添加任意多的功能。这种动态的对对象添加功能的方法不会引起类的爆炸，也具有了更多的灵活性。

以上的方法就是 Decorator 模式，它通过给对象添加装饰来动态的添加新的功能。如下是Decorator 模式的 UML 图：



Component 为组件和装饰的公共父类，它定义了子类必须实现的方法。

ConcreteComponent 是一个具体的组件类，可以通过给它添加装饰来增加新的功能。

Decorator 是所有装饰的公共父类，它定义了所有装饰必须实现的方法，同时，它还保存了一个对于Component 的引用，以便将用户的请求转发给 Component，并可能在转发请求前后执行一些附加的动作。

ConcreteDecoratorA 和 ConcreteDecoratorB 是具体的装饰，可以使用它们来装饰具体的Component。

Java IO 包中的Decorator 模式

JDK 提供的java.io 包中使用了 Decorator 模式来实现对各种输入输出流的封装。以下将以java.io.OutputStream 及其子类为例，讨论一下 Decorator 模式在IO 中的使用。

首先来看一段用来创建 IO 流的代码：

以下是代码片段：

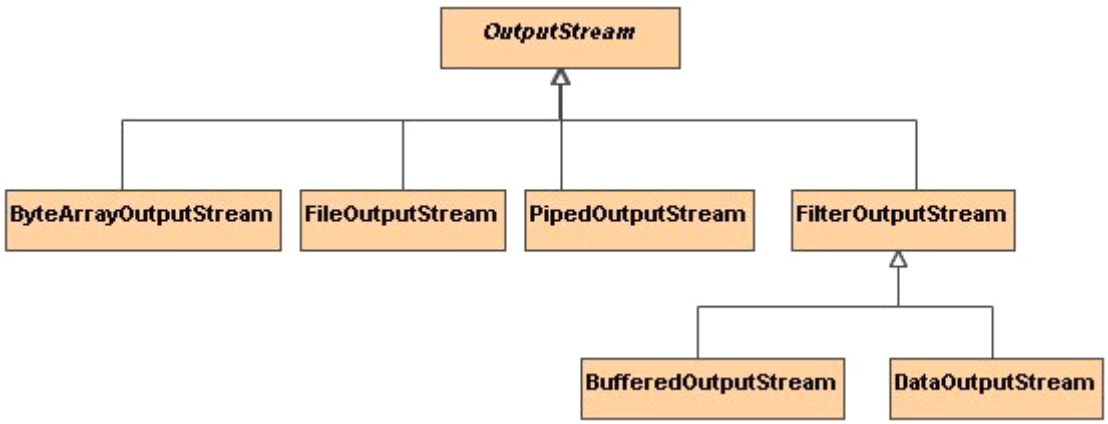
```
try {
   OutputStream out = new DataOutputStream(new FileOutputStream("test.txt"));
} catch (FileNotFoundException e) {
   e.printStackTrace();
}
```

这段代码对于使用过JAVA 输入输出流的人来说再熟悉不过了，我们使用DataOutputStream 封装了一个FileOutputStream，这是一个典型的 Decorator 模式的使用，FileOutputStream 相当于 Component，DataOutputStream 就是一个 Decorator。将代码改成如下，将会更容易理解：

以下是代码片段：

```
try {
   OutputStream out = new FileOutputStream("test.txt");
   out = new DataOutputStream(out);
} catch(FileNotFoundException e) {
   e.printStatckTrace();
}
```

由于FileOutputStream 和 DataOutputStream 有公共的父类OutputStream，因此对对象的装饰对于用户来说几乎是透明的。下面就来看看 OutputStream 及其子类是如何构成 Decorator 模式的：



OutputStream 是一个抽象类，它是所有输出流的公共父类，其源代码如下：

以下是代码片段：

```
public abstract class OutputStream implements Closeable, Flushable {
public abstract void write(int b) throws IOException;
...
}
```

它定义了 write(int b)的抽象方法。这相当于 Decorator 模式中的 Component 类。

ByteArrayOutputStream ， FileOutputStream 和 PipedOutputStream 三个类都直接从 OutputStream 继承，以 ByteArrayOutputStream 为例：

以下是代码片段：

```
public class ByteArrayOutputStream extends OutputStream {
protected byte buf[];
protected int count;
public ByteArrayOutputStream() {
this(32);
}
public ByteArrayOutputStream(int size) {
if (size 〈 0) {
throw new IllegalArgumentException("Negative initial size: " + size);
}
buf = new byte[size];
}
public synchronized void write(int b) {
int newcount = count + 1;
if (newcount 〉 buf.length) {
byte newbuf[] = new byte[Math.max(buf.length 〈〈 1, newcount)];
System.arraycopy(buf, 0, newbuf, 0, count);
buf = newbuf;
}
buf[count] = (byte)b;
count = newcount;
}
...
}
```

它实现了 OutputStream 中的 write(int b)方法，因此我们可以用来创建输出流的对象，并完成特定格式的输出。它相当于 Decorator 模式中的 ConcreteComponent 类。

接着来看一下 FilterOutputStream，代码如下：

以下是代码片段：

```
public class FilterOutputStream extends OutputStream {
protected OutputStream out;
public FilterOutputStream(OutputStream out) {
this.out = out;
}
public void write(int b) throws IOException {
out.write(b);
}
...
}
```

同样，它也是从OutputStream 继承。但是，它的构造函数很特别，需要传递一个OutputStream 的引用给它，并且它将保存对此对象的引用。而如果没有具体的 OutputStream 对象存在，我们将无法创建 FilterOutputStream。由于 out 既可以是指向 FilterOutputStream 类型的引用，也可以是指向 ByteArrayOutputStream 等具体输出流类的引用，因此使用多层嵌套的方式，我们可以为 ByteArrayOutputStream 添加多种装饰。这个 FilterOutputStream 类相当于 Decorator 模式中的 Decorator 类，它的 write(int b)方法只是简单的调用了传入的流的 write(int b)方法，而没有做更多的处理，因此它本质上没有对流进行装饰，所以继承它的子类必须覆盖此方法，以达到装饰的目的。

BufferedOutputStream 和 DataOutputStream 是 FilterOutputStream 的两个子类，它们相当于 Decorator 模式中的 ConcreteDecorator，并对传入的输出流做了不同的装饰。以 BufferedOutputStream 类为例：

以下是代码片段：

```
public class BufferedOutputStream extends FilterOutputStream {
...
private void flushBuffer() throws IOException {
if (count 〉 0) {
out.write(buf, 0, count);
count = 0;
}
}
public synchronized void write(int b) throws IOException {
if (count 〉= buf.length) {
flushBuffer();
}
buf[count++] = (byte)b;
}
...
}
```

这个类提供了一个缓存机制，等到缓存的容量达到一定的字节数时才写入输出流。首先它继承了 FilterOutputStream，并且覆盖了父类的 write(int b)方法，在调用输出流写出数据前都会检查缓存是否已满，如果未满，则不写。这样就实现了对输出流对象动态的添加新功能的目的。

下面，将使用 Decorator 模式，为IO 写一个新的输出流。

自己写一个新的输出流

了解了 OutputStream 及其子类的结构原理后，我们可以写一个新的输出流，来添加新的功能。这部分中将给出一个新的输出流的例子，它将过滤待输出语句中的空格符号。比如需要输出 "java io OutputStream"，则过滤后的输出为 "javaioOutputStream"。以下为 SkipSpaceOutputStream 类的代码：

以下是代码片段：

```
import java.io.FilterOutputStream;
import java.io.IOException;
import java.io.OutputStream;
/**
* A new output stream, which will check the space character
* and won't write it to the output stream.
* @author Magic
*
*/
public class SkipSpaceOutputStream extends FilterOutputStream {
  public SkipSpaceOutputStream(OutputStream out) {
    super(out);
  }
  /**
  * Rewrite the method in the parent class, and
  * skip the space character.
  */
  public void write(int b) throws IOException{
    if(b!=' '){
      super.write(b);
    }
  }
}
```

它从 FilterOutputStream 继承，并且重写了它的 write(int b)方法。在 write(int b)方法中首先对输入字符进行了检查，如果不是空格，则输出。

以下是一个测试程序：

以下是代码片段：

```
import java.io.BufferedInputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
/**
* Test the SkipSpaceOutputStream.
* @author Magic
```

```
*
*/
public class Test {
   public static void main(String[] args){
      byte[] buffer = new byte[1024];

      /**
      * Create input stream from the standard input.
      */
      InputStream in = new BufferedInputStream(new DataInputStream(System.in));

      /**
      * write to the standard output.
      */
      OutputStream out = new SkipSpaceOutputStream(new DataOutputStream(System.out));

      try {
         System.out.println("Please input your words: ");
         int n = in.read(buffer,0,buffer.length);
         for(int i=0;i 〈n;i++){
            out.write(buffer[i]);
         }
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

执行以上测试程序，将要求用户在 console 窗口中输入信息，程序将过滤掉信息中的空格，并将最后的结果输出到 console 窗口。比如：

以下是引用片段：

```
Please input your words:
a b c d e f
abcdef
```

总 结

在 java.io 包中，不仅 OutputStream 用到了 Decorator 设计模式，InputStream，Reader，Writer 等都用到了此模式。而作为一个灵活的，可扩展的类库，JDK 中使用了大量的设计模式，比如在 Swing 包中的 MVC 模式，RMI 中的 Proxy 模式等等。对于 JDK 中模式的研究不仅能加深对于模式的理解，而且还有利于更透彻的了解类库的结构和组成。