**Java 常用 API 的运用，效率及技巧**



## 1. Java 面向对象基本概念

Java 基本上是面向对象的程序设计语言， 除了一些简单类型(primitive)的变量以外，一切都是对象， 程序是对象的组合， 每个对象都有自己的空间， 并且每个对象都有一种类型， 同一类所有对象都能接受相同的消息。 下面只对 Java 中对象的结构作简单的说明：

类（class）： class 是定义类的关键字， 类中包含类变量， 方法， 内部类， 内部接口等。由 class 可以生成类的实例，即一个个对象。 如果一个类的成员被定义成 static 的，则这个成员不专属于任何对象， 而是属于这个类， 所有的对象共享这个成员。

抽象类(abstract class): 抽象类不能直接生成一个实例， 抽象类中必需有方法是 abstract 的，抽象类的意思就是它实现了一部分的方法， 而定义为 abstract 的方法则需要在它的字类中去实现。

接口(interface): 接口可以理解为纯抽象的类，它的每个方法都是未实现的，它可以有成员变量，但必须是 static 的。一个类如果从这个接口继承（implements）则它必须实现这个接口的所有方法。

继承类用关键字：extends，继承接口用关键字：implements。 一个类只能从一个类继承下来， 但可以从多个接口继承（类似于 C++ 的多重继承）。 字类可以覆盖父类的方法(method)， 但不能覆盖父类的成员变量(field)。 如果父类的方法为 final 或 static 的则不能被覆盖。类的初始化顺序是， 如果有父类， 则先初始化父类的 field，然后执行父类的构造函数， 如果子类没有显式的去调父类的构造函数则缺省的会去调父类的无参数构造函数。 然后是子类的 field 与构造函数的初始化。

```
public interface SuperInterface {
        public staitc String SOME_FLAG = “1”;
        public void someMethod();
}
public Class SuperClass {
        { System.out.println( “init SuperClass field” );}
        public SuperClass() {System.out.println( “init SuperClass Constructor” ); }
        public void runMethod() { System.out.println( “run SuperClass runMethod()” ); }
}


public Class SubClass extends SuperClass implements SuperInterface {
```

```
        { System.out.println( “init SubClass field” ); }
        public SubClass() {System.out.println( “init SubClass Constructor” ); }
        public void someMethod() {System.out.println( “run SubClass someMethod()” ); }
        public void runMethod() {System.out.println( “run SubClass runMethod()” ); }
    }
```

有以下 test 代码：

```
public class Test {
        public void main(String[] args) {
            SubClass sub = new SubClass();
            sub. runMethod();
        }
}
```

则会输出：

```
init SuperClass field
init SuperClass Constructor
init SubClass field
init SubClass Constructor
run SubClass runMethod()
```

以下章节所讲述到的常用的 Java API 就是一些 Java 自带的一些 Class 或 Interface 的用法。

**2．System**

System 类位于 package java.lang 下面，凡是此 package 下面的类我们可以直接引用无需先 import 进来，因为 JVM 缺省就 load 了这下面的所有 class。

System 包含了一些我们常用的方法与成员变量。System 不能被实例化，所有的方法都可以直接引用。主要作用大致有：

输入输出流：

(PrintStream) System.out（标准终端输出流），

(PrintStream) System.err（标准错误输出流），

(InputStream) System.in（标准输入流）。

我们还可以重定向这些流，比如将所有的 System.out 的输出全部重定向至一文件中去。

System.setOut(PrintStream) 标准输出重定向

System.setErr(PrintStream) 标准错误输出重定向

System.setIn(InputStream) 标准输入重定向

取当前时间：

System.currentTimeMillis() 所取到的时间是从 1970/01/01 以来 1/1000 秒计算的 long 型值。这个值可以转换至 Date 或 Timestamp 值。它一般还可以用来计算程序执行的时间。例：

```
long beginTime = System. currentTimeMillis();
…
…
System.out.println( “run time = ”  + (System. currentTimeMillis() – beginTime));
```

数组拷贝：

System.arraycopy(Object src, int src_position, Object dst, int dst_position, int length)

src： 源数组。

src_position： 源数组拷贝的起始位置。

dst： 目标数组

dst_position： 拷贝至目标数组的起始位置

length： 拷贝元素的长度

利用 System.arraycopy 进行数组的拷贝效率是最高的， 一般情况下我们自己很少直接用到这个方法，但在集合类的内部中都大量使用了这个方法。

例：

```
int[] array1 = {1, 2, 3, 4, 5};
int[] array2 = {4, 5, 6, 7, 8};
int array3 = new int[8];
System.arraycopy(array1, 0, array3, 0, 5);
System.arraycopy(array2, 2, array3, 5, 3);
```

此时 array3 = {1, 2, 3, 4, 5, 6, 7, 8}

这比用 for 循环来进行赋值效率要高。

存取系统的 Properties：

System.getProperties()：取得当前所有的 Properties， Properties 将在后面的集合一节进行详细的论述。

System.setProperties(Properties props)：设置系统的 Properties。

System.getProperty(String key)： 根据一个键值来取得一个 Property。

System.setProperty(String key, String value)： 设置系统的一个 Property。

JVM 起动的时候将会有一些缺省的 Properties 值， 例如：

java.version Java 运行环境版本

java.home Java 主目录 installation directory

java.class.path Java 的 class path

java.ext.dirs Java 的扩展目录路径

file.separator 文件分隔符("/" on UNIX)

path.separator 路径分隔符(":" on UNIX)

line.separator 行分隔符 ("\n" on UNIX)

user.name 用户名

user.home 用户主目录

user.dir 用户当前工作目录

更详细的信息请参照 Java API。 另外在起动一个 java 程序的时候可以通过-D 来设置系统的 Property， 比如 java –Dejb.file=ejb_Test PrintTest 在 PrintTest 里面就可以通过 System.getProperty(“ejb.file”)来取得值 ejb_Test。

其它

System. loadLibrary(String libname)： 加载 native 的动态库。 可以用 C 写 JNI 的库， 然后在 java 中通过 native 方法来调用。

System.setSecurityManager(SecurityManager s)

System.getSecurityManager()： 设置与取得系统的 security class。

**3．String, StringBuffer**

**3.1 基本用法**

String 可以说是我们最常用的一个类， 熟练掌握它的一些基本用法是很有用的。

String 是由一组字符组成的字符串，下标由 0 开始。一旦有必要改变原来的内容，每个 String 方法都有返回了一个新的 String 对象。

char charAt(int index) 返回指定位置的字符。

int compareTo(Object o)

int compareTo(String anotherString)

与另外一个对象进行比较。

int compareToIgnoreCase(String str) 与另一个 String 进行比较， 不区分大小写

String concat(String str) 连接两字符串， 可以直接用+， 因为 Java 给 String 覆盖了+

static String copyValueOf(char[] data)

static String copyValueOf(char[] data, int offset, int count)

将 data 数组转换至 String

boolean endsWith(String suffix) 测试此 String 是否以 suffix 结尾。

boolean startsWith(String prefix) 测试此 String 是否以 prefix 开头。

boolean equals(Object anObject)

boolean equalsIgnoreCase(String anotherString)

比较两字符串的值。 不相等则返回 false

byte[] getBytes() 根据缺省的字符编码将 String 转换成字节数组。

byte[] getBytes(String enc) 根据指定的编码将 String 转换万字节数组。

void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin) 拷贝字符至一数组中

int indexOf(int ch) 从字串的起始位置查找字符 ch 第一次出现的位置

int indexOf(int ch, int fromIndex) 从指定的 fromIndex 位置向后查找第一次出现 ch 的位置，

int indexOf(String str)

int indexOf(String str, int fromIndex)

如果不存在 ch 或 str 都返回-1

int lastIndexOf(int ch) 从字串的最终位置往前查找第一次出现 ch 的位置

int lastIndexOf(int ch, int fromIndex) 从指定的位置往前查找第一次出现 ch 的位置，

int lastIndexOf(String str)

int lastIndexOf(String str, int fromIndex)

如果不存在则返回-1

int length() 该字符串的字符长度（一个全角的汉字长度为 1）

String replace(char oldChar, char newChar) 将字符 oldChar 全部替换为 newChar， 返回一个新的字符串。

String substring(int beginIndex) 返回从 beginIndex 开始的字符串子集

String substring(int beginIndex, int endIndex) 返回从 beginIndex 至 endIndex 结束的字符串的子集。 其中 endIndex – beginIndex 等于子集的字符串长度

char[] toCharArray() 返回该字符串的内部字符数组

String toLowerCase() 转换至小写字母的字符串

String toLowerCase(Locale locale)

String toUpperCase() 转换至大写字母的字符串

String toUpperCase(Locale locale)

String toString() 覆盖了 Object 的 toString 方法， 返回本身。

String trim() 将字符串两边的半角空白字符去掉， 如果需要去掉全角的空白字符得要自己写。

static String valueOf(primitive p) 将其它的简单类型的值转换为一个 String

StingBuffer 是一个可变的字符串，它可以被更改。同时 StringBuffer 是 Thread safe 的， 你可以放心的使用， 常用的方法如下：

StringBuffer append(param) 在 StringBuffer 对象之后追加 param(可以为所有的简单类型和 Object) 返回追加后的 StringBuffer， 与原来的对象是同一份。

char charAt(int index) 返回指定位置 index 的字符。

StringBuffer delete(int start, int end) 删除指定区域 start~end 的字符。

StringBuffer deleteCharAt(int index) 删除指定位置 index 的字符。

void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin) 同 String 的 getChars 方法

StringBuffer insert(int offset, boolean b) 在指定位置 offset 插入 param(为所有的简单类型与 Object)

int length() 同 String 的 length()

StringBuffer replace(int start, int end, String str) 将指定区域 start~end 的字符串替换为 str

StringBuffer reverse() 反转字符的顺序

void setCharAt(int index, char ch) 设置字符 ch 至 index 位置。

String substring(int start)

String substring(int start, int end) 同 String 的 subString

String toString() 返回一个 String

大家可能已经注意到很多方法都返回了一个 StringBuffer 对象， 但返回的这个对象与 String 的方法返回的 String 不一样， 返回的 StringBuffer 对象与被操作的 StringBuffer 对象是同一份， 而 String 的方法返回的 String 则肯定是重新生成的一个 String。

**3.2 性能对比**

因为 String 被设计成一种安全的字符串， 避免了 C/C++中的尴尬。因此在内部操作的时候会频繁的进行对象的交换， 因此它的效率不如 StringBuffer。如果需要频繁的进行字符串的增删操作的话最好用 StringBuffer。比如拼 SQL 文，写共函。另：编绎器对 String 的+操作进行了一定的优化。

```
x = "a" + 4 + "c"
```

会被编绎成

```
x = new StringBuffer().append("a").append(4).append("c").toString()
```

但：

```
x = “a”;
x = x + 4;
```

x = x + “c”；

则不会被优化。 可以看出如果在一个表达式里面进行 String 的多次+操作会被优化， 而多个表达式的+操作不会被优化。

**3.3 技巧**

1. 在 Servlet2.3 与 JSP1.1 以前画面 post 到后台的数据是通过 ISO88591 格式进行编码的， 则当遇到全角日文字的时候, 在后台 request 得到的数据可能就是乱码, 这个时候就得自己进行编码转换, 通过 String.getBytes(String enc)方法得到一个字节流, 然后通过 String(byte[] bytes, String enc)这个构造函数得到一个用新的编码生成的字符串. 例如将 ISO88591 的字符串转换成 Shift_JIS 的字符串, 方法如下:

```
public static String convertString(String str) {
    if (str == null) { return null; }
    try {
        byte[] buf = str.getBytes("ISO8859_1");
        return new String(buf, "Shift_JIS");
    } catch (Exception ex) {ex.printStackTrace();return null; }
}
```

不过在最新的 Servlet2.3 与 Jsp1.2 中可以通过 request.setCharacterEncoding 来进行设置取值的码制, 不需要自己再做转换。

2．因为 Java 在计算 String 的长度是以字符为单位的， 因此一个全角与半角的字符长度是一样的， 但是 DB 中往往是根据字节来计算长度的， 因此我们在做 Check 的时候得要判断 String 的字节长， 可以用以下的方法：

```
public static String length(String str) {
    if (str == null) { return 0; }
    return str.getBytes().length;
}
```

**4．数值，字符，布尔对象与简单类型的操作**

简单的对照表如下：

| Object | | Primitive | 范围 |
|---|---|---|---|
| Number | Long | long | -9223372036854775808 to 9223372036854775807 |
| | Integer | int | -2147483648 to 2147483647 |
| | Short | short | -32768 to 32767 |
| | Byte | byte | -128 to 127 |
| | Double | double | |
| | Float | float | |
| Character | | char | '\u0000' to '\uffff' |
| Boolean | | boolean | false and true |

与 C 等其它语言不同的是数值的范围不随平台的改变而改变， 这就保证了平台之间的统一性，提高了可移植性。

**Number:** Number 本身是个抽象类， 不能直接使用， 所有直接从 Number 继承下来的子类都有以下几种方法：

byte byteValue() 返回字节值

double doubleValue() 返回 double 值

float floatValue() 返回 float 值

int intValue() 返回 float 值

long longValue() 返回 long 值

short shortValue() 返回 short 值

在需要通过 Object 来取得简单数据类型的值的时候就得用到以上的方法，不过我不推荐不同类型之间的取值，比如 Long 型的 Object 不要直接去调用 intValue()，精度可能会丢失。

如果想通过 String 来得到一个数值类型的简单类型值，一般在每个 Number 的类里面都有一个 parseXXX(String)的静态方法，如下：

byte Byte.parseByte(String s)

double Double.parseDouble(String s)

float Float.parseFloat(String s)

int Integer.parseInt(String s)

long Long.parseLong(String s)

short Short.parseShort(String s)

如果想直接从 String 得到一个 Number 型的 Object，则每个 Number 类里面都有 valueOf(String s) 这个静态方法。如：

Byte Byte.valueOf(String s)

Double Double.valueOf(String s)

Float Float.valueOf(String s)

Integer Integer.valueOf(String s)

Long Long.valueOf(String s)

Short Short.valueOf(String s)

一般的在构造一个 Number 的时候都可以通过一个 String 来完成， 比如：

```
Long longObject = new Long(“1234567890”);
```

等价于

```
Long longObject = Long.valueOf(“1234567890”);
```

因为每个 Number 的子类都实现了 Object 的 toString()方法， 所以， 如果想得到一个 String 型的数值， 直接调用 XXX.toString() 就可以了。 如果想得到一个简单类型的 String， 方法很多总结如下：

首先生成对应的 Number Object 类型， 然后调用 toString()

调用 Number 子类 .toString(type t) 其中 t 就是简单类型的数据。

调用 String.valueOf(type t)（推荐使用这种方法）

大家可以看出，往往一种结果可以用多种方法实现，总的原则就是深度最少优先。比如由一个 String 得到一个简单类型的值可以有以下两种方法：

```
Integer.parseInt(“12345”);
```

或

(new Integer(s)).intValue(“12345”);

当然应该使用第一种方法。

**Character:** Character 对应着 char 类型， Character 类里面有很多静态的方法来对 char 进行判断操作， 详细的操作请参照 JDK API。Java 对字符的判断操作基本都是以 Unicode 进行的，比如 Character.isDigit(char ch)这个方法，不光半角的 0-9 符合要求，全角的日文 0-9 也是符合要求的。

**Boolean:** Boolean 对应着 boolean 类型，boolean 只有 true 和 false 两个值，不能与其它数值类型互换，可以通过字符串”true”以及”false”来得到 Object 的 Boolean， 也可以通过简单类型的 boolean 得到 Boolean， 常用方法如下：

Boolean(boolean value) 通过简单类型的 boolean 构造 Boolean

Boolean(String s) 通过 String(“true”， “false”)构造 Boolean

boolean booleanValue() 由 Object 的 Boolean 得到简单类型的 boolean 值

boolean equals(Object obj) 覆盖了 Object 的.equals 方法， Object 值比较

static Boolean valueOf(String s) 功能与构造函数 Boolean(String s)一样

**5．Class, ClassLoader**

Java 是一种介于解释与编绎之间的语言， Java 代码首先编绎成字节码， 在运行的时候再翻译成机器码。 这样在运行的时候我们就可以通过 Java 提供的反射方法(reflect)来得到一个 Object 的 Class 的额外信息， 灵活性很大，可以简化很多操作。

**Class:** 任何一个 Object 都能通过 getClass()这个方法得到它在运行期间的 Class。 得到这个 Class 之后可做的事情就多了，比如动态得到它的构造函数， 成员变量， 方法等等。 还可以再生成一份新的实例， 下面只给出几个我们常用的方法， 更详细的用法参照 Java API

Class Class.forName(String className) throws ClassNotFoundException： 这是个静态方法， 通过一个 Class 的全称来得到这个 Class。

String getName() 取得这个 Class 的全称， 包括 package 名。

Object newInstance() 得到一个实例， 调用缺省的构造函数。

例如我们有一个类： com.some.util.MyClass 如果得到它的一个实例呢？ 可能有以下两种方法：

MyClass myClass = new MyClass()， 直接通过操作符 new 生成；

或者：

MyClass myClass = (MyClass) Class.forName(“com.some.util.MyClass”).newInstance();

也许有人就会怀疑第二种方法实际意义， 能够直接 new 出来干嘛绕弯。 但实际上它的用处却很大， 举个例子： 用过 struts 的人都知道， 在 action-config.xml 当中定义了一系列的 formBean 与 actionBean， 当然每个 form 与 action 都具有同类型， 这样在一个 request 过来的时候我可以动态的生成一个 form 与 action 的实例进行具体的操作， 但在编码的时候我并不知道具体是何种的 form 与 action， 我只调用它们父类的方法。 你如果要用第一种方法的话， 你得在编码的时候通过一个标志来判断每一次 request 需要具体生成的 form 与 action，代码的灵活性大大降低。总的来说在面向接口的编程当中经常使用这种方法，比如不同数据库厂家的 JDBC Driver 都是从标准的 JDBC 接口继承下去的，我们在写程序的时候用不着管最终是何种的 Driver，只有在运行的时候确定。还有 XML 的 Parser 也是， 我们使用的只是标准的接口， 最后到底是谁来实现它的， 我们用不着去管。

**ClassLoader:** ClassLoader 是一个抽象类，一般的系统有一个缺省的 ClassLoader 用来装载 Class， 用 ClassLoader.getSystemClassLoader()可以得到。不过有时候为了安全或有其它的特殊需要我们可以自定义自己的 ClassLoader 来进行 loader 一些我们需要的 Class， 比如有的产品它用了自己的 ClassLoader 可以指定 Class 只从它指定的特定的 JAR 文件里面来 loader，如果你想通过覆盖 ClassPath 方法来想让它用你的 Class 是行不通的。 有兴趣的可以参照 Java API 的更详细的用法说

**6. Java IO 系统**

**JDK1.0 输入流**

InputStream

ByteArrayInputStream

FileInputStream

FilterInputStream

ObjectInputStream

PipedInputStream

SequenceInputStream

BufferedInputStream

DataInputStream

PushbackInputStream

~~LineNumberInputStream~~

**JDK1.1 输入流**

Reader

BufferedReader

FilterReader

PipedReader

StringReader

InputStreamReader

CharArrayReader

FileReader

PushbackReader

LineNumberReader

**JDK1.0 输出流**

OutputStream

BufferedOutputStream

DataOutputStream
PrintStream
ByteArrayOutputStream
FileOutputStream
FilterOutputStream
ObjectOutputStream
PipedOutputStream
**JDK1.1 输出流**

Writer

BufferedWriter
FilterWriter
PipedWriter
StringWriter
FileWriter
PrintWriter
OutputStreamWriter
CharArrayWriter

如果你刚刚接触 Java 的 IO 部分， 你可能会感觉无从入手， 确实 Java 提供了过多的类，反而让人感到很乱。
可将 Java 库的 IO 类分为输入与输出两个部分， 在 1.0 版本中提供了两个抽象基类， 所有输入的类都从 InputStream 继承， 所有输出的类都从 OutputStream 继承， 1.1 提供了两个新的基类，负责输入的 Reader 与输出的 Writer，但它们并不是用来替换原来老的 InputStream 与 OutputStream， 它们主要是让 Java 能更好的支持国际化的需求。 原来老的 IO 流层只支持 8 位字节流， 不能很好地控制 16 位 Unicode 字符。 Java 内含的 char 是 16 位的 Unicode， 所以添加了 Reader 和 Writer 层次以提供对所有 IO 操作中的 Unicode 的支持。 除此之外新库也对速度进行了优化， 可比旧库更快地运行。

InputStream 的类型：

1. 字节数组
2. String 对象
3. 文件
4. 管道， 可以从另外一个输出流得到一个输入流
5. 一系列的其他流， 可以将这些流统一收集到单独的一个流内。
6. 其他起源（如 socket 流等）

还有一个是 File 类， Java 中一个目录也是一个文件，可以用 file.isFile()和 file.isDirectory()来进行判断是文件还是目录。 File 对象可能作为参数转换为文件流进行操作。 具体操作参照 Java IO API。

**常用方法：**

```
/**
 * 拼文件名, 在 windows(\)跟 unix(/)上的文件分割符是不一样的
 * 可以通过 File 类的静态成员变量 separator 取得
 */
public static String concatFileName(String dir, String fileName) {
String fullFileName = "";
if (dir.endsWith(File.separator)) {fullFileName = dir + fileName;
} else {fullFileName = dir + File.separator + fileName;}
return fullFileName; }

/**
 * 从控制台读取输入的数据, System.in (InputStream) 先转换至 InputStreamReader 再用
 * BufferedReader 进行读取.
 *
 */
BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
String str = "";
while (str != null) {str = in.readLine();// process(str); }

/**
 * 从文件中按行进行读取数据处理
 */
BufferedReader in = new BufferedReader(new FileReader("infilename"));
String str;
while ((str = in.readLine()) != null) {// process(str); }
in.close();

/**
 * 写数据至一个新的文件中去.
 */
BufferedWriter out = new BufferedWriter(new FileWriter("outfilename"));
out.write("a String");
out.close();

/**
 * 追加新的数据到一个文件中去, 如果原文件不存在
 * 则新建这个文件.
 */
BufferedWriter out = new BufferedWriter(new FileWriter("filename", true));
```

```
out.write("aString");
out.close();

/**
 * 将一个可序列化的 Java 的 object 以流方式写到一个文件中去中.
 */
ObjectOutput out = new ObjectOutputStream(new FileOutputStream("filename.ser"));
out.writeObject(object);
out.close();
/**
 * 从文件中恢复序列化过的 Java Object
 */
ObjectInputStream in = new ObjectInputStream(new FileInputStream("filename.ser"));
Object object = (Object) in.readObject();
in.close();

/**
 * 以指定的编码方式从文件中读取数据
 */
BufferedReader in = new BufferedReader(new InputStreamReader(
                        new FileInputStream("infilename"), "UTF8"));
String str = in.readLine();

/**
 * 以指定的编码方式写数据到文件中
 */
Writer out = new BufferedWriter(new OutputStreamWriter(
                                new FileOutputStream("outfilename"), "UTF8"));
out.write("a String");
out.close();
```

**由上面的例子可以看出， 流之间可以互相转换， 如果想对流进行字符操作最好将之转换成 BufferedReader 与 BufferedWriter 这样可以提高读写的效率。**

需要注意的是一般来说流的大小是得不到的， 虽然一般的 InputStream 都有一个 available()的方法可以返回这个流可以读到的字节数， 不过这个方法有时候不会很准确， 比如在读取网络传输的流的时候， 取到的长度并不一定是真实有效的长度。

大家在写程序的时候可能已经注意到一些类都继承了 java.io.Serializable 这个接口， 其实继承了这个接口之后这个类的本身并不做任何事情， 只是这个类可以被序列化并通过流来进行传输， 并可被还原成原 Object。 这个流可以通过网络传输（EJB 中进行传递参数和返回值的 Data Class ）， 也可以保存到文件中去（ ObjectOutputStream ）。

**7. Java 集合类**

在写程序的时候并不是每次只使用一个对象， 更多的是对一组对象进行操作， 就需要知道如何组合这些对象， 还有在编码的时候我们有时并不知道到底有多少对象，它们需要进行动态的分配存放。

Java 的集合类只能容纳对象句柄， 对于简单类型的数据存放， 只能通过数据来存放， 数组可以存放简单类型的数据也能存放对象。

Java 提供了四种类型的集合类： Vector(矢量)， BitSet(位集)， Stack(堆栈)， Hashtable(散列表)。

1. 矢量： 一组有序的元素， 可以通过 index 进行访问。
2. 位集： 其实就是由二进制位构成的 Vector， 用来保存大量”开-关”信息， 它所占的空间比较小， 但是效率不是很高， 如果想高效率访问， 还不如用固定长度的数组。
3. 堆栈： 先入后出（ LIFO ）集合， java.util.Stack 类其实就是从 Vector 继承下来的， 实现了 pop, push 方法。
4. 散列表： 由一组组“键--值”组成， 这里的键必须是 Object 类型。 通过 Object 的 hashCode 进行高效率的访问。

对于这些集合之间的关联关系见下图， 其中标色的部分为我们常用的类。

由上图可以看出， 基本接口有两个：

**Collection：**所有的矢量集合类都从它继承下去的， 但并不直接从它继承下去的。 List 与 Set 这两个接口直接继承了 Collection，他们的区别是 List 里面可以保存相同的对象句柄， 而 Set 里面的值是不重复的。 我们经常用的 Vector 与 ArrayList 就是从 List 继承下去的， 而 HashSet 是从 Set 继承的。

**Map：**散列表的接口， Hashtable 与 HashMap 继承了这个接口。

下面给出常用集合类的常用方法。

```
/**
 * Vector 与 ArrayList 的操作几乎是一样的
 * 常用的追加元素用 add(), 删除元素用 remove()
 * 取元素用 get(), 遍历它可以循环用 get()取. 或者
 * 先得到一个 Iterator, 然后通过遍历 Iterator 的方法
 * 遍历 Vector 或 ArrayList
 */
// 生成一个空的 Vector
Vector vector = new Vector();
// 在最后追加一个元素。
vector.add("one");
vector.add("two");
// 在指定的地方设置一个值
vector.set(0, "new one");
// 移走一个元素或移走指定位置的元素
vector.remove(0);
// 用 for 循环遍历这个 Vector
for (int i = 0; i < vector.size(); i++) {
String element = (String) vector.get(i); }
// 用枚举器(Enumeration)遍历它(只有 Vector 有，ArrayList 没有)
Enumeration enu = vector.elements();
while (enu.hasMoreElements()) {
```

```
enu.nextElement();}
// 用反复器(Iterator)遍历它
Iterator it = vector.iterator();
while (it.hasNext()) {it.next();}
/**
 * Hashtable 与 HashMap 的操作, 追加元素用 put(不是 add)
 * 删除元素用 remove, 遍历可以用 Iterator 既可以遍历
 * 它的 key, 也可以是 value
 */
// 生成一个空的 Hashtable 或 HashMap
Hashtable hashtable = new Hashtable();
// 追加一个元素
hashtable.put("one", "one object value");
// 删除一个元素
hashtable.remove("one");
// 用 Iterator 遍历
Iterator keyIt = hashtable.keySet().iterator();
while (keyIt.hasNext()) {
Object keyName = keyIt.next();
String value = (String) hashtable.get(keyName); }
Iterator valueIt = hashtable.values().iterator();
while (valueIt.hasNext()) {
valueIt.next();}

// 用 Enumeration 遍历, 只有 Hashtable 有, HashMap 没有.
Enumeration enu = hashtable.elements();
while (enu.hasMoreElements()) {
enu.nextElement();}
```

**说明：**Enumeration 是老集合库中的接口，而 Iterator 是新集合（1.2）中出现的，而 Vector 与 Hashtable 也都是老集合中的类，所以只有 Vector 与 Hashtable 可以用 Enumeration。

**Vector 与 ArrayList 对比：**

虽然在使用的时候好象这两个类没什么区别，它们都是从 List 继承下来的，拥有相同的方法，但它们的内部还是有些不同的，

首先 Vector 在内部的一些方法作了线程同步(synchronized)。同步的代价就是降低了执行效率，但提高了安全性。而 ArrayList 则是线程不同步的，可以多线程并发读写它。

内部数据增长率。所有的这些矢量集合在内部都是用 Object 的数组进行存储和操作的。所以也就明白了为什么它可以接受任何类型的 Object，但取出来的时候需要进行类型再造。Vector 与 ArrayList 具有自动伸缩的功能，我们不用管它 size 多大，我们都可以在它的后面追加元素。Vector 与 ArrayList 内部的数组增长率是不一样的，当内部的数组不能容纳更多元素的时候，Vector 会自动增长到原两倍大小，ArrayList 会变为原一倍半大小，而不是我们所想象的一个元素一个元素的增长。

**Hashtable 与 HashMap 对比：**

Hashtable 与 HashMap 都是从 Map 继承下来的， 方法几乎都一样， 它们内部有两个不同点：

与 Vector 和 ArrayList 一样， 它们在线程同步是不同的， Hashtable 在内部做了线程同步， 而 HashMap 是线程不同步的。

HashMap 的键与值都可以为 null， 而 Hashtable 不可以， 如果你试图将一个 null 值放到 Hashtable 里面去， 会抛一个 NullPointException 的。

**性能对比：**

抛开不常用的集合不讲， 每种集合都应该有一个我们常用的集合类， 而在不同的场合下应该使用效率最高的一个。 一般来说我推荐尽量使用新的集合类， 除非不得已， 比如说需要用用了老集合类写的产品的程序。 也就是说尽量使用 ArrayList 与 HashMap， 而少使用 Vector 与 Hashtable。

在单线程中使用 ArrayList 与 HashMap， 而在多线程中如果需要进行线程同步可以使用 Vector 与 Hashtable， 但也可以用 synchronized 对 ArrayList 与 HashMap 进行同步，不过同步后的 ArrayList 与 HashMap 是比 Vector 与 Hashtable 慢的。 不过我认为需要进行线程同步的地方并不多。如果一个变量定义在方法内部同时只可能有一个线程对之进行操作， 就不必要进行同步， 如果定义在类的内部并且不是静态的， 属于实例变量， 而这个类并没有被多线程使用也就不必要同步。

一般自己写的程序很少会自己去另开线程的， 但在 Web 开发的时候， 如果用了 Servlet， 则每个 request 都是一个线程， 也就是说每个 Servlet 都是在多线程环境下运行的， 如果 Servlet 中使用了全局静态的成员变量就得小心点儿， 如果需要同步就得在方法上加上 synchronized 修饰符， 如果允许多个线程操作它， 并且你知道不会有什么冲突问题就可以大胆的使用 ArrayList 与 HashMap。 另外如果在多线程中有线程在对 ArrayList 或 HashMap 进行修改（结构上的修改）， 而有一个线程在用 Iterator 进行读取操作，这个时候就有可能会抛 ConcurrentModificationException, 因为用 Iterator 的时候，不允许原 List 的结构改变。但可以用 get 方法来取。

**常用技巧：**

1. 采用面向接口的编程技巧， 比如现在需要写一个共通函数，对矢量集合类诸如 Vector，ArrayList，HashSet 等等进行操作， 但我并不知道最终用户会具体传给我什么类型的类， 这个时候我们可以使用 Collection 接口， 从而使代码具有很大的灵活性。 代码示例如下：

```
/**
 * 将 list 里面的所有元素用 sep 连接起来，
 * list 可以为 Vector, ArrayList, HashSet 等。
 */
public static String join(String sep, Collection list) {
  StringBuffer sb = new StringBuffer();
  Iterator iterator = list.iterator();
  while (iterator.hasNext()) {
        sb.append(iterator.next());
        if (iterator.hasNext()) { sb.append(sep); }
  }
  return sb.toString();
}
```

2. 利用 Set 进行 Unique，比如有一组对象（其中有对象是重复的），但我们只对不同的对象感兴趣，这个时候可以使用 HashSet 这个集合类，然后可以通过覆盖 Object 的 equals 方法来选择自定义判断相等的 rule。缺省的是地址判断。例：

```
class DataClass {
  private String code = null;
  private String name = null;

  public void setCode(String code) {this.code = code; }
  public String getCode() {return this.code; }
  public void setName(String name) {this.name = name; }
  public String getName() {return this.name; }
  public boolean equals(DataClass otherData) {
      if (otherData != null) {
          if (this.getCode() != null&& this.getCode().equals(otherData.getCode()) {
              return true;
          }
      }
      return false;
  }}
DataClass data1 = new DataClass();
DataClass data2 = new DataClass();
data1.setCode("1");
data2.setCode("1");
HashSet singleSet = new HashSet();
singleSet.add(data1);
singleSet.add(data2);
```

结果 singleSet 里面只有 data1，因为 data2.equals(data1)，所以 data2 并没有加进去。

3. 灵活的设计集合的存储方式，以获得较高效的处理。集合里面可以再嵌套集合，例：在 ArrayList 里面存放 HashMap，HashMap 里面再嵌套 HashMap。

**8．ResourceBundle, Properties**

**ResourceBundle：**开发一个项目，配置文件是少不了的，一些需要根据环境进行修改的参数，都有得放到配置文件中去，在 Java 中一般是通过一个 properties 文件来实现的，这个文件以 properties 结尾。内部结构是二维的，以 key=value 的形式存在。如下：

```
options.column.name.case=1
options.column.bean.serializable=1
options.column.bean.defaultconstructor=1
options.column.method.setter=1
options.general.user.version=1.0
database.connection[0]=csc/csc@localhost_oci8
```

database.connection[1]=cscweb/cscweb@localhost_thin

ResourceBundle 用来解析这样的文件，它的功能是可以根据你的 Locale 来进行解析配置文件，如果一个产品需要进行多语言支持，比如在不同语种的系统上，会显示根据它的语言显示相应的界面语言，就可以定义多份的 properties 文件，每个文件的 key 是一样的，只是 value 不一样， 然后在 application 起动的时候， 可以判别本机的 Locale 来解析相应的 properties 文件。 Properties 文件里面的数据得要是 Unicode。 在 jdk 下面可以用 native2ascii 这个命令进行转换。 例： native2ascii Message.txt Message.properties 会生成一个 Unicode 的文件。

**Properties:** Properties 这个类其实就是从 Hashtable 继承下来的，也就是说它是一个散列表，区别在于它的 key 与 value 都是 String 型的， 另外也加了几个常用的方法：

String getProperty(String key) 取得一个 property

String getProperty(String key, String defaultValue) 取 property， 如果不存在则返回 defaultValue。

void list(PrintStream out) 向 out 输出所有的 properties

void list(PrintWriter out)

Enumeration propertyNames() 将所有的 property key 名以 Enumeration 形式返回。

Object setProperty(String key, String value) 设置一个 property。

ResourceBundle 与 Properties 一般结合起来使用。 它们的用法很简单， 由 ResourceBundle 解析出来的 key 与 value 然后放至到一个静态的 Properties 成员变量里面去， 然后就可以通过访问 Properties 的方法进行读取 Property。 下面给个简单的例子：

```
public class PropertyManager implements Serializable {
/** 定义一个静态的 Properties 变量 */
private static Properties properties = new Properties();
/**
 * 通过一个类似于类名的参数进行 Property 文件的初期化
 * 比如现在有一个文件叫 Message.properties， 它存放在
 * ejb/util 下面并且， 这个目录在运行的 classpath 下面
 * 则 in 就为 ejb.util.Message
 *
 */
public static void init(String in) throws MissingResourceException {
   ResourceBundle bundle = ResourceBundle.getBundle(in);
   Enumeration enum = bundle.getKeys();
   Object key = null;
   Object value = null;
   while (enum.hasMoreElements()) {
      key = enum.nextElement();
      value = bundle.getString(key.toString());
      properties.put(key, value);
   }}
/**
 * 取得一个 Property 值
```

```
 */
public static String getProperty(String key) {return properties.get(key); }
/**
 * 设置一个 Property 值
 */
public static void setProperty(String key, String value) {properties.put(key, value); }
}
```

不过现在的 Java 产品中，越来越倾向于用 XML 替换 Properties 文件来进行配置。 XML 配置具有层次结构清楚的优点。

**9. Exceptions**

Throwable
Exception
Error
RuntimeException
ClassNotFoundException
IOException
SQLException
IndexOutOfBoundsException
ClassCastException
NullPointerException
OutOfMemoryError
StackOverflowError
NoClassDefFoundError

Java 采用违例(Exception)处理机制来进行错误处理。 违例机制的一个好处就是能够简化错误控制代码， 我们再也不用检查一个特定的错误，然后在程序的多处地方对其进行控制。此外，也不需要在方法调用的时候检查错误( 因为保证有人能够捕获这里的错误 )。我们只需要在一个地方处理问题：” 违例控制模块” 或者” 违例控制器” 。 这样可有效减少代码量， 并将那些用于描述具体操作的代码与专门纠正错误的代码分隔开。
一个完整的违例例子：

```
public void throwTest() throws MyException {
try {
        ...
} catch (SQLException se) {
        cat.error("", se);
        throw new MyException(se.getMessage());
} catch (Exception e) {cat.error("", e);
} finally {
        ...
```

```
    }
    }
```

如果一段代码有可能会抛出违例可以用 try {} catch {}来处理。 被 catch 到的违例可以再抛出， 也可以转换为其它类型的 Exception 抛出。 finally 块里面的代码总会被执行到的， 不管前面是否已经 throw 或 return 了。

Throwable 是所有违例的基类， 它有两种常规类型。 其中， Error 代表编绎期和系统错误， 我们一般不必特意捕获它们。 Exception 是可以从任何标准 Java 库的类方法中掷出的基本类型。

看上面的图， 如果是 Error 的子类或是 RuntimeException 的子类这种违例有一定的特殊性， 可以说我们可以当它们不存在， 当这种违例抛出的时候， 我们可以不 catch 它， 也可以不在方法上 throws 它。 RuntimeException 一般代表的是一个编程错误， 是完全可以避免的。

**性能注意点：** 因为使用了 Exception 之后是要影响一些效率的， 所以 Exception 不能滥用。一般的不要用 Exception 来控制业务流程， 其次不要循环体内使用。

**技巧：**我们可以从 Exception 或直接从 Throwable 继承写我们自己的 Exception， 然后根据业务需要抛不同种类的 Exception。

**10. JDBC 类库**

有了 JDBC，向各种关系数据库发送 SQL 语句就是一件很容易的事。换言之，有了 JDBC API，就不必为访问 Sybase 数据库专门写一个程序，为访问 Oracle 数据库又专门写一个程序，为访问 Informix 数据库又写另一个程序，等等。您只需用 JDBC API 写一个程序就够了，它可向相应数据库发送 SQL 语句。而且，使用 Java 编程语言编写的应用程序，就无须去忧虑要为不同的平台编写不同的应用程序。将 Java 和 JDBC 结合起来将使程序员只须写一遍程序就可让它在任何平台上运行。

下面为常用的处理流程：

DriverManager

Connection

Statement

PreparedStatement

CallableStatement

ResultSet

简单地说，JDBC 可做三件事：

1. 与数据库建立连接
2. 发送 SQL 语句
3. 处理结果

下列代码段给出了以上三步的基本示例：

```
Class.forName("oracle.jdbc.driver.OracleDriver").newInstance();
Connection conn = DriverManager.getConnection (
                                    "jdbc:oracle:thin:@eai-sol:1521:eai_db", "csc2", "csc2");
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT CONTACTID FROM CONTACTINFO");
while (rs.next()) {long contactID = rs.getLong("CONTACTID");}
```

下面对常用的几个类和接口做些简单的说明。

**DriverManager:**

DriverManager 类是 JDBC 的管理层，作用于用户和驱动程序之间。它跟踪可用的驱动程序，并在数据库和相应驱动程序之间建立连接。另外，DriverManager 类也处理诸如驱动程序登录时间限制及登录和跟踪消息的显示等事务。对于简单的应用程序，一般程序员需要在此类中直接使用的唯一方法是 DriverManager.getConnection。正如名称所示，该方法将建立与数据库的连接。

```
Class.forName("oracle.jdbc.driver.OracleDriver").newInstance();
Connection conn = DriverManager.getConnection (
                                    "jdbc:oracle:thin:@eai-sol:1521:eai_db", "csc2", "csc2");
```

其中第一句话的作用是在当前的环境中 load 一个 DB Driver，有人可能觉得奇怪，这句话执行完之后，后面怎么知道去用这个 Driver 呢？ 其实 DriverManager 可以从 load 的 classes 里面找到注册过的 driver ,然后使用它所找到的第一个可以成功连接到给定 URL 的驱动程序。 第二句话的三个参数分别是 URL, User, Password。 Driver 不一样， URL 可能也不一样。

**Statement:**

Statement 对象用于将 SQL 语句发送到数据库中。实际上有三种 Statement 对象，它们都为在给定连接上执行 SQL 语句的包容器：Statement、PreparedStatement（它从 Statement 承而来）和 CallableStatement（它从 PreparedStatement 继承而来）。它们都专用于发送定类型的 SQL 语句： Statement 对象用于执行不带参数的简单 SQL 语句；PreparedStatement 对象用于执行带或不带 IN 参数的预编译 SQL 语句；CallableStatement 对象用于执行对数据库已存储过程的调用。

Statement 接口提供了执行语句和获取结果的基本方法。PreparedStatement 接口添加了处理 IN 参数的方法；

而 CallableStatement 添加了处理 OUT 参数的方法。

创建 Statement 对象

建立了到特定数据库的连接之后，就可用该连接发送 SQL 语句。Statement 对象用 Connection 的方法 createStatement 创建，如下列代码段中所示：

```
Statement stmt = conn.createStatement();
```

为了执行 Statement 对象，被发送到数据库的 SQL 语句将被作为参数提供给 Statement 的方法：

```
ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM table1");
```

使用 Statement 对象执行语句

Statement 接口提供了三种执行 SQL 语句的方法：executeQuery、executeUpdate 和 execute。使用哪一个方法由 SQL 语句所产生的内容决定。

方法 executeQuery 用于产生单个结果集的语句，例如 SELECT 语句。

方法 executeUpdate 用于执行 INSERT、UPDATE 或 DELETE 语句以及 SQL DDL（数据定义语言）语句，例如 CREATE TABLE 和 DROP TABLE。INSERT、UPDATE 或 DELETE 语句的效果是修改表中零行或多行中的一列或多列。executeUpdate 的返回值是一个整数，指示受影响的行数（即更新计数）。对于 CREATE TABLE 或 DROP TABLE 等不操作行的语句，executeUpdate 的返回值总为零。

方法 execute 用于执行返回多个结果集、多个更新计数或二者组合的语句。

语句完成

当连接处于自动提交模式时，其中所执行的语句在完成时将自动提交或还原。语句在已执行且所有结果返回时，即认为已完成。对于返回一个结果集的 executeQuery 方法，在检索完 ResultSet 对象的所有行时该语句完成。对于方法 executeUpdate，当它执行时语句即完成。但在少数调用方法 execute 的情况中，在检索所有结果集或它生成的更新计数之后语句才完成。

关闭 Statement 对象

Statement 对象将由 Java 垃圾收集程序自动关闭。而作为一种好的编程风格，应在不需要 Statement 对象时显式地关闭它们。这将立即释放 DBMS 资源，有助于避免潜在的内存问题。 关闭 Statement 用 stmt.close()　方法。

**ResultSet:**

ResultSet 包含符合 SQL 语句中条件的所有行，并且它通过一套 get 方法（这些 get 方法可以访问当前行中的不同列）提供了对这些行中数据的访问。ResultSet.next 方法用于移动到 ResultSet 中的下一行，使下一行成为当前行。

结果集一般是一个表，其中有查询所返回的列标题及相应的值。例如，如果查询为 SELECT a, b, c FROM Table1，则结果集将具有如下形式：

```
a       b         c
-------- --------- --------
12345   Cupertino CA
83472   Redmond   WA
83492   Boston    MA
```

下面的代码段是执行 SQL 语句的示例。该 SQL 语句将返回行集合，其中列 1 为 int，列 2 为 String，而列 3 则为日期型：

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM Table1");
while (rs.next()) {
int i = rs.getInt("a");
String s = rs.getString("b");
Timestamp t = rs.getTimestamp("c");
}
```

行和光标

ResultSet 维护指向其当前数据行的光标。每调用一次 next 方法 ,光标向下移动一行。最初它位于第一行之前 ,因此第一次调用 next 将把光标置于第一行上，使它成为当前行。随着每次调用 next 导致光标向下移动一行，按照从上至下的次序获取 ResultSet 行。

在 ResultSet 对象或其父辈 Statement 对象关闭之前，光标一直保持有效。

列

方法 getXXX 提供了获取当前行中某列值的途径。在每一行内，可按任何次序获取列值。但为了保证可移植性，应该从左至右获取列值，并且一次性地读取列值。列名或列号可用于标识要从中获取数据的列。例如，如果 ResultSet 对象 rs 的第二列名为“title”，并将值存储为字符串，则下列任一代码将获取存储在该列中的值：

```
String s = rs.getString("title");
String s = rs.getString(2);
```

注意列是从左至右编号的，并且从列 1 开始。同时，用作 getXXX 方法的输入的列名不区分大小写。 **为了代码的可维护性与可读性，应该禁止用 index 的方法来取值，　要用读列名的方法，　如上面的第一行取值方法。**

数据类型和转换

对于 getXXX 方法，JDBC 驱动程序试图将基本数据转换成指定 Java 类型，然后返回适合的 Java 值。例如，如果 getXXX 方法为 getString，而基本数据库中数据类型为 VARCHAR，则 JDBC 驱动程序将把 VARCHAR 转换成 Java String。getString 的返回值将为 Java String 对象。

NULL 结果值

要确定给定结果值是否是 JDBC NULL，必须先读取该列，然后使用 ResultSet.wasNull 方法检查该次读取是否返回 JDBC NULL。

**PreparedStatement:**

该 PreparedStatement 接口继承 Statement，并与之在两方面有所不同：

PreparedStatement 实例包含已编译的 SQL 语句。这就是使语句“准备好”。 包含于 PreparedStatement 对象中的 SQL 语句可具有一个或多个 IN 参数。IN 参数的值在 SQL 语句创建时未被指定。相反的，该语句为每个 IN 参数保留一个问号（ “？” ）作为占位符。每个问号的值必须在该语句执行之前，通过适当的 setXXX 方法来提供。

由于 PreparedStatement 对象已预编译过，所以其执行速度要快于 Statement 对象。因此，多次执行的 SQL 语句经常创建为 PreparedStatement 对象，以提高效率。

作为 Statement 的子类，PreparedStatement 继承了 Statement 的所有功能。另外它还添加了一整套方法，用于设置发送给数据库以取代 IN 参数占位符的值。同时，三种方法 execute、 executeQuery 和 executeUpdate 已被更改以使之不再需要参数。

创建 PreparedStatement 对象

以下的代码段（其中 conn 是 Connection 对象）创建包含带两个 IN 参数占位符的 SQL 语句的 PreparedStatement 对象：

```
PreparedStatement pstmt=conn.prepareStatement("UPDATE table1 SET a = ? WHERE b = ?");
```

pstmt 对象包含语句 " UPDATE table1 SET a = ? WHERE b = ?"，它已发送给 DBMS，并为执行作好了准备。

传递 IN 参数

在执行 PreparedStatement 对象之前，必须设置每个？参数的值。这可通过调用 setXXX 方法来完成，其中 XXX 是与该参数相应的类型。例如，如果参数具有 Java 类型 long，则使用的方法就是 setLong。setXXX 方法的第一个参数是要设置的参数的序数位置，第二个参数是设置给该参数的值。例如，以下代码将第一个参数设为 123456789，第二个参数设为 100000000：

```
pstmt.setLong(1, 123456789);
pstmt.setLong(2, 100000000);
```

**CallableStatement:**

CallableStatement 对象为所有的 DBMS 提供了一种以标准形式调用已储存过程(也就是 S P)的方法。已储存过程储存在数据库中。对已储存过程的调用是 CallableStatement 对象所含的内容。有两种形式：一种形式带结果参数，另一种形式不带结果参数。结果参数是一种输出 (OUT) 参数，是已储存过程的返回值。两种形式都可带有数量可变的输入（ IN 参数）、输出（ OUT 参数）或输入和输出（ INOUT 参数）的参数。问号将用作参数的占位符。

在 JDBC 中调用已储存过程的语法如下所示。注意，方括号表示其间的内容是可选项；方括号本身并不是语法的组成部份。

```
{call 过程名[(?, ?, ...)]}
```

返回结果参数的过程的语法为：

```
{? = call 过程名[(?, ?, ...)]}
```

不带参数的已储存过程的语法类似：

```
{call 过程名}
```

通常，创建 CallableStatement 对象的人应当知道所用的 DBMS 是支持已储存过程的，并且知道这些过程都是些什么。然而，如果需要检查，多种 DatabaseMetaData 方法都可以提供这样的信息。例如，如果 DBMS 支持已储存过程的调用，则 supportsStoredProcedures 方法将返回 true，而 getProcedures 方法将返回对已储存过程的描述。

CallableStatement 继承 Statement 的方法（它们用于处理一般的 SQL 语句），还继承了 PreparedStatement 的方法（它们用于处理 IN 参数）。CallableStatement 中定义的所有方法都用于处理 OUT 参数或 INOUT 参数的输出部分：注册 OUT 参数的 JDBC 类型（一般 SQL 类型）、从这些参数中检索结果，或者检查所返回的值是否为 JDBC NULL。

创建 CallableStatement 对象

CallableStatement 对象是用 Connection 方法 prepareCall 创建的。下例创建 CallableStatement 的实例，其中含有对已储存过程 Csc_ GetCustomId 调用。该过程有两个变量，但不含结果参数：

```
CallableStatement cstmt = con.prepareCall("{call CSC_GetCustomId (?, ?, ?)}");
```

其中？占位符为 IN、 OUT 还是 INOUT 参数，取决于已储存过程 Csc_ GetCustomId。

IN 和 OUT 参数

将 IN 参数传给 CallableStatement 对象是通过 setXXX 方法完成的。该方法继承自 PreparedStatement。所传入参数的类型决定了所用的 setXXX 方法（例如，用 setFloat 来传入 float 值等）。

如果已储存过程返回 OUT 参数，则在执行 CallableStatement 对象以前必须先注册每个 OUT 参数的 JDBC 类型（这是必需的，因为某些 DBMS 要求 JDBC 类型）。注册 JDBC 类型是用 registerOutParameter 方法来完成的。语句执行完后，

CallableStatement 的 getXXX 方法将取回参数值。正确的 getXXX 方法是为各参数所注册的 JDBC 类型所对应的 Java 类型也就是说， registerOutParameter 使用的是 JDBC 类型（因此它与数据库返回的 JDBC 类型匹配），而 getXXX 将之转换为 Java 类型。下面给出 CSC 中的一个例子：

```
String sqlSp = "{call CSC_GetCustomId(?, ?, ?)}";
cstmt = conn.prepareCall(sqlSp.toString());
cstmt.registerOutParameter(1, Types.NUMERIC);
cstmt.registerOutParameter(2, Types.NUMERIC);
cstmt.registerOutParameter(3, Types.VARCHAR);
cstmt.execute();
long customerID = cstmt.getLong(1);
long lRet = cstmt.getLong(2);
String sErr = cstmt.getString(3);
```

INOUT 参数

既支持输入又接受输出的参数（INOUT 参数）除了调用 registerOutParameter 方法外，还要求调用适当的 setXXX 方法（该方法是从 PreparedStatement 继承来的）。setXXX 方法将参数值设置为输入参数，而 registerOutParameter 方法将它的 JDBC 类型注册为输出参数。setXXX 方法提供一个 Java 值，而驱动程序先把这个值转换为 JDBC 值，然后将它送到数据库中。这种 IN 值的 JDBC 类型和提供给 registerOutParameter 方法的 JDBC 类型应该相同。然后，要检索输出值，就要用对应的 getXXX 方法。例如，Java 类型为 byte 的参数应该使用方法 setByte 来赋输入值。应该给 registerOutParameter 提供类型为 TINYINT 的 JDBC 类型，同时应使用 getByte 来检索输出值。下例假设有一个已储存过程 reviseTotal，其唯一参数是 INOUT 参数。方法 setByte 把此参数设为 25，驱动程序将把它作为 JDBC TINYINT 类型送到数据库中。接着，registerOutParameter 将该参数注册为 JDBC TINYINT。执行完该已储存过程后，将返回一个新的 JDBC TINYINT 值。方法 getByte 将把这个新值作为 Java byte 类型检索。

```
CallableStatement cstmt = con.prepareCall("{call reviseTotal(?)}");
cstmt.setByte(1, 25);
cstmt.registerOutParameter(1, java.sql.Types.TINYINT);
cstmt.executeUpdate();
byte x = cstmt.getByte(1);
```

**11. 常用设计模式**

**1．Singleton 模式**

Singleton 模式主要作用是保证在 Java 应用程序中，一个 Class 只有一个实例存在。一般有两种方法：

定义一个类，它的构造函数为 private 的，所有方法为 static 的。其他类对它的引用全部是通过类名直接引用。例如：

```
private SingleClass() {}
public static String getMethod1() {}
public static ArrayList getMethod2() {}
```

定义一个类，它的构造函数为 private 的，它有一个 static 的 private 的该类变量，通过一个 public 的 getInstance 方法获取对它的引用,继而调用其中的方法。例如：

```
private staitc SingleClass _instance = null;
private SingleClass() {}
public static SingleClass getInstance() {
   if (_instance == null) {_instance = new SingleClass();}
   return _instance; }
public String getMethod1() {}
public ArrayList getMethod2() {}
```

**2．Prototype 模式**

Prototype 模式用于创建对象，尤其是当创建对象需要许多时间和资源时。在 Java 中，Prototype 模式的实现是通过方法 clone(),该方法定义在 Java 的根对象 Object 中, 因此 ,Java 中的其他对象只要覆盖它就行了。通过 clone(),我们可以从一个对象获得更多的对象，并请可以按照我们的需要修改他们的属性。

```
public class Prototype implements Cloneable {
 private String Name;
 public rototype(String Name) {this.Name = Name; }
 public void setName(String Name) {this.Name = Name; }
 public String getName() {return Name; }
 public Object clone() {
    try{return super.clone();
    }catch(CloneNotSupportedException cnse){
       cnse.printStackTrace();
       return null;
    }
 }}
Prototype p = new Prototype("My First Name");
Prototype p1 = p.clone();
p.setName("My Second Name");
Prototype p2 = p.clone();
```

**3. Factory 模式和 Abstract Factory 模式**

Factory 模式

利用给 Factory 对象传递不同的参数，以返回具有相同基类或实现了同一接口的对象。

Abstract Factory 模式

先利用 Factory 模式返回 Factory 对象，在通过 Factory 对象返回不同的对象！

下面给出 Sun XML Parser 中的例子：

```
// 1. Abstract Factory 模式
SAXParserFactory spf = SAXParserFactory.newInstance();
String validation = System.getProperty ("javax.xml.parsers.validation", "false");
if (validation.equalsIgnoreCase("true")) {spf.setValidating (true); }
// 2. Factory 模式
SAXParser sp = spf.newSAXParser();
parser = sp.getParser();
parser.setDocumentHandler(this);
parser.parse (uri);
```

1. SAXParserFactory 中的静态方法 newInstance()根据系统属性 javax.xml.parsers.SAXParserFactory 不同的值生成不同的 SAXParserFactory 对象 spf。然后 SAXParserFactory 对象又利用方法 newSAXParser()生成 SAXParser 对象。

注意：

SAXParserFactory 的定义为：

```
public abstract class SAXParserFactory extends java.lang.Object
```

SAXParserFactoryImpl 的定义为：

```
public class SAXParserFactoryImpl extends javax.xml.parsers.SAXParserFactory
public static SAXParserFactory newInstance() {
String factoryImplName = null;
try {
    factoryImplName =System.getProperty("javax.xml.parsers.SAXParserFactory",
                                "com.sun.xml.parser.SAXParserFactoryImpl");
} catch (SecurityException se) {
        factoryImplName = "com.sun.xml.parser.SAXParserFactoryImpl";
}
SAXParserFactory factoryImpl;
try {
    Class clazz = Class.forName(factoryImplName);
    factoryImpl = (SAXParserFactory) clazz.newInstance();
} catch (ClassNotFoundException cnfe) {
    throw new FactoryConfigurationError(cnfe);
} catch (IllegalAccessException iae) {
    throw new FactoryConfigurationError(iae);
} catch (InstantiationException ie) {
    throw new FactoryConfigurationError(ie);
}
return factoryImpl;
}
```

2. newSAXParser() 方法在 SAXParserFactory 定义为抽象方法，

SAXParserFactoryImpl 继承了 SAXParserFactory，它实现了方法 newSAXParser()：

```
public SAXParser newSAXParser()
                                throws SAXException, ParserConfigurationException {
SAXParserImpl saxParserImpl = new SAXParserImpl(this);
return saxParserImpl;
}
```

注意：

SAXParserImpl 的定义为：

```
public class SAXParserImpl extends javax.xml.parsers.SAXParser
```

SAXParserImpl 的构造函数定义为：

```
public SAXParserImpl(SAXParserFactory spf)
                                throws SAXException, ParserConfigurationException {
super();
this.spf = spf;
if (spf.isValidating ()) {
    parser = new ValidatingParser();
    validating = true;
} else { parser = new Parser();}
if (spf.isNamespaceAware ()) {
    namespaceAware = true;
    throw new ParserConfigurationException(
                                                "Namespace not supported by SAXParser");
}
}
```