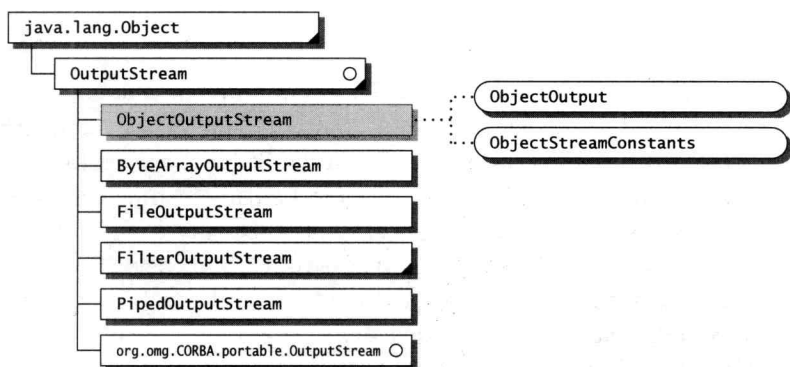


java.io

ObjectOutputStream



语法

```
public class ObjectOutputStream extends OutputStream implements ObjectOutput ,
ObjectStreamConstants
```

描述

`ObjectOutputStream`类用于序列化原始数据及对象(包括数组及字符串)到一个输出流。序列化数据可由一个相关的`ObjectOutputStream`读取。

参见《The Java Class Libraries , Second Edition , Volume 1》获得关于此类的更多信息。

版本1.2中所作的修改

对象代替

当将一个对象转化为一个序列化流时, `ObjectOutputStream`允许其自身的一个可靠子类用正在被序列化的该对象代替另一个对象, 这被称为对象代替。在版本 1.1 中, 这是由子类激活 `enableReplaceObject()` 并提供 `replaceObject()` 的实现(为了进行替换, `replaceObject()` 激活了 `writeObject()` 来完成的。在版本 1.2 中, 对象代替更灵活。子类可使用 `enableReplaceObject()` 及 `replaceObject()`, 或者根本不使用 `writeObject()` 的实现, 通过为新方法 `writeObjectOverride()` 提供一个实现来完成对象代替。

安全

`ObjectOutputStream`类被改变来使用版本 1.2 的安全模式(参见 `SecurityManager` 可了解其大概)。在版本 1.1 中, 对象代替仅能由一个可靠的子类来执行, 既由系统类加载器装载的一个子类。在版本 1.2 中, 当一个安全控制器被安装, 为了 `enableReplaceObject()` 能成功, 正在使用 `ObjectOutputStream` 子类的调用者及子类本身一定要已被授予 `SerializablePermission` (“enableSubstitution”) 权。对于这类涉及到 `writeObjectOverride()` 的对象代替, 当安全控制器

已被安装，调用者和子类一定要已被授予 `SerializablePermission` (“`enableSubclassImplementation`”) 权。

其他变化

`ObjectOutputStream` 增加了一个新的保护的构造函数。这个构造函数被 `ObjectOutputStream` 的子类所用，如果子类希望对 `ObjectOutputStream` 的实现有完全的控制 (通过为 `writeObjectOverride()` 方法提供一个实现)。

为了处理可序列化域，该类增加了两个方法：`putFields()` 与 `writeFields()`。`putFields()` 使用结构给一个对象的可序列化域赋值；`writeFields()` 将那些值写到输出流中。

`Externalizable` 数据的格式随着 JDK 1.1.6 版的发行而改变。1.1.6 版以前版本写的数据使用版本 1 的对象序列化协议，1.1.6 版以后写的数据使用版本 2 的对象序列化协议，新增了方法 `useProtocolVersion()`，当序列化对象时，允许调用者选择协议的版本。

方法 `defaultWriteObject()` 与 `enableReplaceObject()` 不再是最终的。`defaultWriteObject()` 在《The Java Class Libraries, Second Edition, Volume 1》中有描述。

成员概述

构造函数

Δ	<code>ObjectOutputStream()</code>	构造一个 <code>ObjectOutputStream</code> 实例。
	类的 <code>writeObject()</code> 方法	
Δ	<code>defaultWriteObject()</code>	向 <code>ObjectOutputStream</code> 中写正在被序列化的对象。
1.2	<code>putFields()</code>	检索赋值给对象可序列化域的结构。
1.2	<code>writeFields()</code>	向 <code>ObjectOutputStream</code> 中写用 <code>putFields()</code> 设置的域。
	流方法	
	<code>close()</code>	关闭 <code>ObjectOutputStream</code> 。
	<code>drain()</code>	写缓冲区中的数据到基础输出流。
	<code>flush()</code>	刷新 <code>ObjectOutputStream</code> 。
	<code>reset()</code>	重新设置 <code>ObjectOutputStream</code> 的状态。
1.2	<code>useProtocolVersion()</code>	确定当向 <code>ObjectOutputStream</code> 写数据时的协议版本。
	流定制方法	
	<code>annotateClass()</code>	向 <code>ObjectOutputStream</code> 中写和一个类相关的信息。
Δ	<code>enableReplaceObject()</code>	允许 / 禁止 <code>ObjectOutputStream</code> ，替代写入 <code>ObjectOutputStream</code> 中的对象。
	<code>replaceObject()</code>	当序列化时，用一个对象替代另一个对象。
1.2	<code>writeObjectOverride()</code>	向 <code>ObjectOutputStream</code> 中写一个对象。
	<code>writeStreamHeader()</code>	向基础输出流中写流头部。
	成员概述	
	<code>ObjectOutput</code> 方法	
	<code>write()</code>	向 <code>ObjectOutputStream</code> 中写一个或多个字节。
Δ	<code>writeObject()</code>	向 <code>ObjectOutputStream</code> 中写一个对象。

此方法在《The Java Class Libraries, Second Edition, Volume 1》中有描述。它唯一的变化是其不再是最终的。

(续)

成员概述	
DataOutput方法	
writeBoolean()	向ObjectOutputStream中写一逻辑值。
writeByte()	向ObjectOutputStream中写一8位字节。
writeBytes()	向ObjectOutputStream中写一字符串(一串字节)。
writeChar()	向ObjectOutputStream中写一16位字符。
writeChars()	向ObjectOutputStream中写一字符串(一串字符)。
writeDouble()	向ObjectOutputStream中写一64位双精度数。
writeFloat()	向ObjectOutputStream中写一32位浮点数。
writeInt()	向ObjectOutputStream中写一32位整形数。
WriteLong()	向ObjectOutputStream中写一64位长整形数。
WriteShort()	向ObjectOutputStream中写一16位短整形数。
WriteUTF()	向ObjectOutputStream中写一UTF字符串。

参见

ObjectOutputStream.PutField , SerializablePermission。
《The Java Class Libraries , Second Edition , Volume 1》中的ObjectOutputStream。

A enableReplaceObject()

目的	允许/禁止此ObjectOutputStream来替换写入其中的对象。
语法	protected boolean enableReplaceObject(boolean enable) throws SecurityException
描述	此方法允许或禁止此ObjectOutputStream来替换写入其中的对象。在序列化一个对象到一个基础流之前， writeObject()允许一个ObjectOutputStream的子类使用 replaceObject()返回的值来替代这个对象。一般来说， ObjectOutputStream不允许这种替代。为了允许这种替代， ObjectOutputStream的子类必须调用返回true的enableReplaceObject()。 当一个安全控制器已被安装，调用者必须已被授予 SerializablePermission (“ enableSubstitution ”)权。
版本1.2中的改动	在版本1.1中，这个方法是最终的。而且，在版本1.1中此方法仅能被系统类加载器装载的一个ObjectOutputStream子类的实例激活。在版本1.2中，使用的是新的安全模式(参见类描述)。
参数	
enable	若为true，允许写到ObjectOutputStream中的对象被替代；若为false，写到ObjectOutputStream中的对象不能被替代。
返回	在enableReplaceObject()被激活之前，返回原先的设置。如果替代可行则返回true，否则返回false。
异常	
SecurityException	

如果enable是true，并且安全控制器不允许调用者调用这个方法。

参见

java.lang.ClassLoader，ObjectInputStream.enableResolveObject()，replaceObject()，SerializablePermission(“enableSubstitution”)。

示例

参见《The Java Class Libraries，Second Edition，Volume 1》中的replaceObject()。

▲ ObjectOutputStream()

目的

构造一个ObjectOutputStream实例。

语法

public ObjectOutputStream(OutputStream out) throws IOException

从版本1.2

protected ObjectOutputStream() throws IOException，SecurityException

描述

此公有的构造函数创建ObjectOutputStream的一个实例，将序列化数据写到流out中。它写流头部(参见writeStreamHeader())到out中并准备接受对象及数据被序列化。

保护的构造函数被子类的构造函数所使用，如果那个子类将为writeObjectOverride()方法提供一个实现。当writeObject()在这样一个子类上被激活，它的writeObjectOverride()方法被调用并且由ObjectOutputStream提供的实现并没有被使用。仅当子类正在完全地重新实现ObjectOutputStream，构造函数应被使用(writeObjectOverride()被重载)。注意保护的构造函数不接受任何参数，所以，可以自由地按它选择的任意方式处理流及头部。这个构造函数不像公有的构造函数，它不执行任何初始化，因此子类必须重载close()，因为ObjectOutputStream.close()使用由公有构造函数初始化的私有域。

为了成功地使用保护的构造函数，构造函数的调用者必须已被授予SerializablePermission(“enableSubclassImplementation”)权，否则，SecurityException被抛出。

版本1.2中的改动

保护的构造函数在版本1.2中是新增加的。

参数

out

写出的非空输出流。

异常

IOException

当向out写入时如果发生IO错误。

SecurityException

如果调用者未被授予SerializablePermission(“enableSubclassImplementation”)权限。

参见

SerializablePermission、writeObject()、writeObjectOverride()。

示例

见writeObjectOverride()使用保护的构造函数的示例。

```
try {
    ObjectOutputStream out =
        new ObjectOutputStream(new FileOutputStream("test.ser"));
    out.writeObject(new java.awt.Label("hello there"));
    out.close();
} catch (IOException e) {
    System.out.println(e);
}
```

1.2 putFields()

目的	检索一个结构，用于给一个对象的可序列化域分配值。
语法	<code>public ObjectOutputStream.PutField putFields() throws IOException</code>
描述	此方法检索 <code>ObjectOutputStream.PutField</code> (为分配值给将被写入 <code>ObjectOutputStream</code> 中的可序列化域)，在 <code>Serializable</code> 对象的 <code>writeObject()</code> 方法内部被调用。 <code>writeObject()</code> 方法接着激活 <code>put()</code> 方法，以设置可序列化域的值。以这种方式设置的域必须使用 <code>writeFields()</code> 被刷新到输出流。
返回	为了分配值给一个对象的可序列化域，返回一个非空 <code>PutField</code> 。
参见	<code>writeFields()</code> 。
示例	见 <code>ObjectOutputStream.PutField</code> 的示例。

1.2 useProtocolVersion()

目的	当向这个 <code>ObjectOutputStream</code> 写数据时确定所使用的协议版本。
语法	<code>public void useProtocolVersion(int version) throws IOException</code>
描述	<code>Externalizable</code> 数据的格式随着 JDK 1.1.6 版的发布而改变。在 1.1.6 版之前写的数 据使用 <code>ObjectStreamConstants.PROTOCOL_VERSION_1</code> ，而用 1.1.6 版之后写的数 据使 用 <code>ObjectStreamConstants.PROTOCOL_VERSION_2</code> 。当序列化对象时，调用者使用 <code>useProtocolVersion()</code> 方法来 选择所使用的协议版本，缺省值是 <code>ObjectStreamConstants.PROTOCOL_VERSION_2</code> 。 这个方法必须在任何对象被写到 <code>ObjectOutputStream</code> 之前被调用。
参数	
version	<code>ObjectStreamConstants.PROTOCOL_VERSION_1</code> 或 <code>ObjectStreamConstants.PROTOCOL_VERSION_2</code> 。
异常	
<code>IllegalArgumentException</code>	如果 <code>version</code> 值不是两个合法版本值中的任何一个。
<code>IllegalStateException</code>	如果此方法在一个对象已被写到流中后调用。
<code>IOException</code>	当试图设置版本时，如果发生了 IO 错误。
示例	<pre>ObjectOutputStream objout = new ObjectOutputStream(out) ; objout. useProtocolVersion(ObjectStreamConstants.PROTOCOL_ VERSION_1) ;</pre>

1.2 writeFields()

目的	将用 <code>putFields()</code> 设置的域写入 <code>ObjectOutputStream</code> 中。
语法	<code>public void writeFields() throws IOException</code>
描述	此方法将由 <code>putFields()</code> 设置的域写入 <code>ObjectOutputStream</code> 中。一个 <code>Serializable</code> 对象的 <code>writeObject()</code> 方法通常调用 <code>ObjectOutputStream</code> 上的

putFields()来分配值给将被序列化的域，然后激活 ObjectOutputStream上的writeFields()来写入域值。

异常

IOException

当向该流写入时如果发生IO错误。

NotActiveException

如果没有对象正被序列化，或者对象的域没有通过 putFields()进行设置。

参见

putFields()。

示例

见ObjectOutputStream.PutField类示例。

▲ writeObject()

目的

向ObjectOutputStream中写入一个对象。

语法

```
public final void writeObject(Object obj) throws IOException
```

描述

此方法序列化 obj 并将它写入基础输出流。如果 obj 的域包括了其他对象的引用，那些对象及它们的引用也被序列化了，并被写入基础输出流，以至于基于 obj 对象的全部图都被序列化。通过使用下面的任意一个类方法将对象序列化：

- writeExternal()如果类实现 Externalizable。
- defaultWriteObject()如果类实施 Serializable 但并未定义 writeObject()方法。
- writeObject()如果类实现 Serializable 并且定义了 writeObject()方法。

如果 ObjectOutputStream 允许替代写入它的对象，则 replaceObject()被激活(obj作为其参数)，并且结果被序列化。

如果 writeObject()正被使用保护的 ObjectOutputStream 构造函数创建的 ObjectOutputStream 子类调用，writeObject()仅简单地调用 writeObjectOverride()并返回。

如果在 writeObject()中遇到一个异常，整个 ObjectOutputStream 数据的合法性受到怀疑，并且数据不再被进一步使用。

版本1.2中的改动

writeObjectOverride()在版本1.1中不存在，因此 writeObject()没有使用它。

参数

obj

要写的可能为 null 的对象。如果为 null，null 被写入流中。

异常

InvalidClassException

如果 obj 的类(或者基于 obj 的对象图中的某一对象的类)是一个未知原始类型的数组，或如果类域并不与正在被序列化的对象匹配。

IOException

当写入基础输出流时如果发生IO错误。

NotSerializableException

如果 obj 或它引用的某一对象并没有实现 Serializable(或 Externalizable) 接口。

参见

enableReplaceObject()、replaceObject()、writeObjectOverride()。

示例

见ObjectOutputStream()。

12 writeObjectOverride()

目的	写一个对象到ObjectOutputStream中。
语法	protected void writeObjectOverride(Object obj) throws IOException
描述	该方法的缺省实施并不做什么。该方法仅在使用了保护的ObjectOutputStream构造函数的ObjectOutputStream子类中被调用。这些子类必须为该方法提供实现来向输出流中写入obj数据。
参数	
obj	可能为空的序列化对象。
异常	
IOException	
示例	当写入序列化数据时如果发生IO错误。 这个示例说明了如何通过使用保护的ObjectOutputStream构造函数完整地重载writeObject()的功能，以及为writeObjectOverride()提供一个实现。注意子类重载close()方法来避免ObjectOutputStream.close()被调用。调用ObjectOutputStream.close()将引起NullPointerException，因为保护的构造函数并没有初始化被close()所使用的流。

```
import java.io.*;

class Main {
    public static void main(String[] args) {
        try {
            SillyObjectOutputStream out =
                new SillyObjectOutputStream(new FileOutputStream("test.ser"));
            out.writeObject(new java.awt.Label("hello there"));
            out.close();
        } catch (IOException e) {
            System.out.println(e);
        }
    }

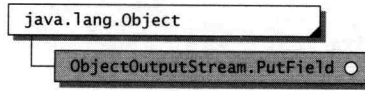
    static class SillyObjectOutputStream extends ObjectOutputStream {
        OutputStream out;
        byte[] sillyOutput = "silly".getBytes();

        SillyObjectOutputStream(OutputStream out) throws IOException {
            super(); // use protected constructor
            this.out = out;
        }

        protected void writeObjectOverride(Object obj) throws IOException {
            out.write(sillyOutput); // bogus implementation, just for demo
        }

        public void close() throws IOException {
            out.close();
        }
    }
}
```


ObjectOutputStream.PutField



语法

```
static public abstract class ObjectOutputStream.PutField
```

描述

抽象类 `ObjectOutputStream.PutField` 表示将被写到 `ObjectOutputStream` 中的一个对象的可序列化域。可序列化域是可序列化类中的域，该类已被指明为可序列化的类。可序列化类是实现 `Serializable` 接口的类。它有两种方法指明哪个域是可序列化的：缺省序列化与明确序列化。参见 `ObjectStreamField` 可得这两种方法的描述。该类用于表示将被序列化的域 (使用缺省或明确序列化方法)。

使用

一旦一个可序列化域为了一个类而存在，为了维持兼容性它就必须存在于将来所有的版本中。在序列化时，如果涉及到一个类，以至于它不再有一个域名与序列化的域相映射，它使用 `ObjectOutputStream.PutField` 给可序列化域赋一个值 (此值来自其当前的实现数据)。被涉及到的类应声明一个 `writeObject()` 方法，使用 `ObjectInputStream.putFields()` 获得 `ObjectOutputStream.PutField` 的一个实例。`writeObject()` 使用 `ObjectOutputStream.putFields` 的实例给可序列化域赋值，包括那些在被涉及类中没有相应域的。当所有的可序列化域值已经被分配，`writeObject()` 将使用 `ObjectOutputStream.writeFields()` 将它们写到流中。

如果一个类定义了 `writeObject()` 方法来写比可序列化域多的数据，那么它也需要定义一个相应的 `readObject()` 方法在反序列化时读回那些数据。参见 `Serializable` 类描述可获得更多信息。

在下面的示例中，`writeObject()` 使用一个被涉及类中相应的新域写出两个可序列化域，`strField` 与 `intField`。接着使用 `ObjectOutputStream.writeFields()` 将两个域刷新到输出流中。

```
private static final ObjectStreamField[] serialPersistentFields = {
    new ObjectStreamField("strField", String.class),
    new ObjectStreamField("intField", Integer.TYPE),
};

...
private void writeObject(ObjectOutputStream out) throws IOException {
    // Get PutField instance for assigning values to serializable fields
    ObjectOutputStream.PutField pfields = out.putFields();

    pfields.put("strField", newStrField);
    pfields.put("intField", newIntField);
    // Write fields out
    out.writeFields();
}
```


成员概述	
域赋值方法	
put()	赋值给一个可序列化域。
刷新输出方法	
write()	通过调用它写出可序列化域。

参见

ObjectInputStream.GetField , ObjectOutputStream.writeFields() ,
ObjectStreamClass , ObjectStreamField。
《The Java Class Libraries , Second Edition , Volume 1》中的Serializable。

示例

参见ObjectStreamField类的示例。在那个示例中，类classA最先声明明确可序列化域field1，发展成为用field10来代替field1。在扩展的类中，field10的值被记录为可序列化域field1(为保持序列化目的的兼容性)。

put()

目的	为可序列化域赋一个值。
语法	<pre>abstract public void put(String fname, boolean fvalue) abstract public void put(String fname, byte fvalue) abstract public void put(String fname, char fvalue) abstract public void put(String fname, short fvalue) abstract public void put(String fname, int fvalue) abstract public void put(String fname, long fvalue) abstract public void put(String fname, float fvalue) abstract public void put(String fname, double fvalue) abstract public void put(String fname, Object fvalue)</pre>
描述	该方法分配值fvalue给名为fname的可序列化域。
参数	
fname	一个标识可序列化域名字的非空字符串。
fvalue	分配给可序列化域的值。如果fvalue是非基本类型的，它可以为null。
异常	
IllegalArgumentException	如果fname没有命名一个序列化域，或者 defvalue的类型与实际的序列化域的类型不兼容。
参见	ObjectInputStream.GetField.get()。
示例	见“使用”以及ObjectStreamField类示例。

write()

目的	通过调用它写出可序列化域。
语法	abstract public void write(ObjectOutput out) throws IOException

描述	这个方法被 <code>ObjectOutputStream.writeFields()</code> 所使用来写出 <code>ObjectOutputStream.PutField</code> 实例的可序列化域到 <code>out</code> 。它不应被直接使用。
参数	
<code>out</code>	用来写可序列化域的非空 <code>ObjectOutput</code> 对象。
异常	
<code>IOException</code>	当写入域时如果发生 IO 错误。

java.io

ObjectStreamClass



语法

```
public class ObjectStreamClass implements java.io.Serializable
```

描述

ObjectStreamClass是一个用来描述可序列化类的描述器，它由类名及流统一标识符(Stream Unique Identifier, SUID)组成。而且，如果此类实现Serializable，则这个类描述器包含一系列名字以及类的可序列化域的类型名字。参见《The Java Class Libraries, Second Edition, Volume 1》中的Serializable和ObjectStreamField获得可序列化域的描述。如果类实施了Externalizable，类描述器就没有这个列表。这个描述器和一个序列化对象一起存储，用于标识序列化对象的类。

参见《The Java Class Libraries, Second Edition, Volume 1》可得关于该类的更多信息。

版本1.2中所作的修改

在该类中增加了两个方法和一个常量：getField()、getFields()及NO_FIELDS。这些变化允许程序从描述器中检索可序列化域。

成员概述

查找方法

lookup() 为一个类检索描述器。

Get方法和常量

forClass() 检索由此描述器描述的类。

1.2 getField() 为由此描述器表示的类检索一个可序列化域。

1.2 getFields() 为由此描述器表示的类检索可序列化域。

getName() 检索由此描述器描述的类名。

getSerialVersionUID() 检索由此描述器描述的类的SUID。

1.2 NO_FIELDS 表明一个类没有可序列化域。

Object方法

toString() 生成表示此描述器的字符串。

参见

ObjectStreamField。

《The Java Class Libraries, Second Edition, Volume 1》中的ObjectStreamClass。

1.2 getField()

目的	为由此描述器表示的类检索一个可序列化域。
语法	<code>public ObjectOutputStreamField getField(String name)</code>
描述	此方法为由此描述器表示的类检索名为 <code>name</code> 的可序列化域。它主要用于调试。
参数	
<code>name</code>	表示可序列化域名称的一个非空字符串。
返回	一个可能为 <code>null</code> 的 <code>ObjectStreamField</code> (表示名为 <code>name</code> 的可序列化域)；如果类没有这样的可序列化域则返回 <code>null</code> 。
参见	<code>getFields()</code> 。
示例	这个示例是 <code>ObjectStreamField.getName()</code> 示例的修改。它使用 <code>getField()</code> 明确地在每个可序列化域中检索 <code>ObjectStreamField</code> 。

Main.java

```
import java.io.*;

class Main {
    public static void main(String[] args) {
        try {
            // Write them out
            FileOutputStream f = new FileOutputStream("ClassA1.ser");
            ObjectOutputStream out = new ObjectOutputStream(f);

            out.writeObject(new ClassA("hello", 500));
            out.flush();
            out.close();

            // Read it back
            FileInputStream f2 = new FileInputStream("ClassA1.ser");
            ObjectInputStream in = new ObjectInputStream(f2);
            ClassA obj = (ClassA) in.readObject();
            in.close();

            System.out.println(obj);
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

ClassA.java

```
import java.io.*;
import java.awt.Button;

public class ClassA implements Serializable {
    /**
     * A string field.
     * @serial
     */
    String strField;

    /**
     * An int field.
     * @serial
     */
    int intField;

    /**
```

```

    * A byte array field.
    * @serial
    */
    byte[] byteArrayField = new byte[] {(byte)3, (byte)2, (byte)1};

    /**
     * An object field.
     * @serial
     */
    Button[] buttons = new Button[] {new Button("push me")};

    public ClassA(String one, int two) {
        strField = one;
        intField = two;
    }

    private void readObject(ObjectInputStream in)
        throws IOException, ClassNotFoundException {

        // Read in serial fields
        ObjectInputStream.GetField pfields = in.readFields();

        // Assign serialized values to fields
        strField = (String)pfields.get("strField", "out-of-luck");
        intField = (int)pfields.get("intField", 1000);
        byteArrayField = (byte[]) pfields.get("byteArrayField", null);
        buttons = (Button[]) pfields.get("buttons", null);

        // Get ObjectStreamClass to get ObjectStreamFields for each field
        ObjectStreamClass objclass = pfields.getObjectStreamClass();
        print("strField", objclass.getField("strField"));
        print("intField", objclass.getField("intField"));
        print("byteArrayField", objclass.getField("byteArrayField"));
        print("buttons", objclass.getField("buttons"));
        print("notThere", objclass.getField("notThere")); // null
    }

    private void print(String field, ObjectStreamField fd) {
        System.out.println("Field: " + field);
        if (fd != null) {
            System.out.println("name: " + fd.getName());
            System.out.println("type: " + fd.getType());
            System.out.println("type code: " + fd.getTypeCode());
            System.out.println("type string: " + fd.getTypeString());

            System.out.println("primitive: " + fd.isPrimitive());
            System.out.println("offset: " + fd.getOffset());
        }
    }

    public String toString() {
        return "strField: " + strField +
            " intField: " + intField +
            " byteArrayField: " + byteArrayField +
            " buttons: " + buttons;
    }
}

```

1.2 getFields()

目的	检索由此描述器表示的类的可序列化域。
语法	public ObjectStreamField[] getFields()
描述	该方法检索由此描述器表示的类的可序列化域，使用结果集中的每一个元素表示每一个可序列化域。如果此类不含可序列化域，则结果集是一个空数组。使用 getField()，通过可序列化域名字获取各自的可序列化域。这个方法主要用于调试目的，所以调用者可发现一个类的可序列化域。

返回	表示可序列化域的一个非空数组。
参见	getField()。
示例	参见ObjectStreamField.getName()的示例。

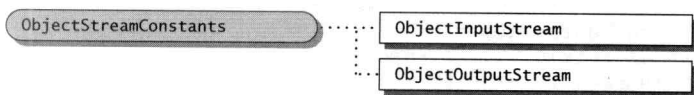
1.2 NO_FIELDS

目的	指明一个没有可序列化域的类。
语法	public static final ObjectStreamField[] NO_FIELDS
描述	该常数指明一个没有可序列化域的类。它的值是一个空数组。可以使用这个常数在getField()的结果集上进行相等性检查。
参见	getField()。
示例	

```
ObjectStreamField[] fields = objClass.getFields();
if (fields == ObjectOutputStream.NO_FIELDS) {
    // action for no fields case
} else {
    // action to process fields
}
```

java.io

ObjectStreamConstants



语法

```
public interface ObjectStreamConstants
```

描述

ObjectStreamConstants接口包含使用于对象序列化中的常量定义。除了 `PROTOCOL_VERSION_1`和`PROTOCOL_VERSION_2`(被用作`ObjectOutputStream.useProtocolVersion()`的参数)以外，所有常量通常仅在对象序列化内部被使用。仅当获取 `ObjectOutputStream/ObjectInputStream`子类重载它们的缺省实现时，需要使用常量。

参考`Serializable`、`ObjectStreamClass`、`ObjectInputStream`及`ObjectOutputStream`类可得一些术语的解释，如块数据。完整的有关序列化/反序列化算法的细节描述请在网址<http://java.sun.com/products/jdk/1.2/docs/guide/serialization>中查找。

成员概述

流头部

`STREAM_MAGIC` 0xaced

`STREAM_VERSION` 5

在串行流中用于标记项目的标志

`baseWireHandle` 0x7e0000 被赋予的第一个wire句柄。

`TC_ARRAY` 0x75 一个新数组。

`TC_BASE` 0x70 第一个标志值。

`TC_BLOCKDATA` 0x77 可选数据块的开始，下一个字节指明块中的字节数。

`TC_BLOCKDATALONG` 0x7a 可选长数据块的开始，下一个长字节指明块中的字节数。

`TC_CLASS` 0x76 类的一个引用

`TC_CLASSDESC` 0x72 一个新的类描述器 (序列化的 `ObjectStreamClass`)。

序列化流中用于标记项目的标志

`TC_ENDBLOCKDATA` 0x78 一个对象的可选块的结尾。

`TC_EXCEPTION` 0x7b 一个写被放弃标识。

`TC_MAX` 0x7b 最后一个标志值。

`TC_NULL` 0x70 一个空的对象引用。

(续)

成员概述		
TC_OBJECT	0x73	一个新对象。
TC_REFERENCE	0x71	一个对象的引用已写入流中。
TC_RESET	0x79	流被重新设置的标识。
TC_STRING	0x74	一个新字符串。
ObjectStreamClass标志的位屏蔽		
SC_BLOCK_DATA	0x08	写入块数据模式中的外部数据。
SC_EXTERNALIZABLE	0x04	外部化。
SC_SERIALIZABLE	0x02	可序列化类。
SC_WRITE_METHOD	0x01	可序列化类已定义了 writeObject()。
可序列化协议版本		
PROTOCOL_VERSION_1	1	使用per-JDK1.1.5外部数据格式。
PROTOCOL_VERSION_2	2	使用JDK1.1.5和post-JDK1.1.5外部数据格式。
安全许可 (参见SerializablePermission)		
SUBCLASS_IMPLEMENTATION_N		允许ObjectInputStream/ ObjectOutputStream
PERMISSIO		的子类实现重载readObject()/writeObject()。
SUBSTITUTION_PERMISSION		允许使用replaceObject()/resolveObject()。

参见

ObjectStreamClass、ObjectStreamField、ObjectInputStream、ObjectOutputStream、
SerializablePermission。
《The Java Class Libraries , Second Edition , Volume 1》 中的Externalizable、Serializable。

java.io

ObjectStreamField



语法

```
public class ObjectStreamField implements Comparable
```

描述

ObjectStreamField类表示一个可序列化域。一个可序列化域是在已被声明为可序列化的可序列化类中的域。一个可序列化类是一个实施 Serializable接口的类。它有两种方法指明它的哪个域是可序列化的：

- 缺省序列化
- 明确序列化

缺省串行

利用缺省序列化，一个类的可序列化域是它所有的非暂态和非静态的域。当该类的一个实例被序列化时，ObjectOutputStream.defaultWriteObject()被用于序列化这些域。因为文档编制目的，一个可串行域应在它的 javadoc 中有一个 @serial 标志，用于描述该域的内容。@serial 标志接收关于该域附加信息的可选说明。此可选说明及域的 javadoc 说明被 javadoc 用来产生域的序列化格式文档。

如果类定义了一个 writeObject() 方法，则它可以序列化这些域和/或向序列化流中添加其他信息。任何被 writeObject() 所写的数据和被称为可选数据的可序列化域没有联系。writeObject() 方法应被文档化(通过在其 javadoc 中使用一个 @serialData 标志)，用于描述格式、序列和可选数据类型。

当读/反序列化对象时(使用 ObjectInputStream 类)，一个相似但相反的过程发生。查看 The Java Class Libraries, Second Edition, Volume 1 中 Serializable 类可得细节。

下面是一个具有可序列化域 field1 的可序列化类。

```

public class ClassA implements Serializable {
    /**
     * A string for testing.
     * @serial
     */
    String field1;
    transient String field2;
    public ClassA(String f1, String f2) {
        field1 = f1;
        field2 = f2;
    }
}
  
```

明确序列化

明确序列化是重载缺省序列化的一个方法,并且明确指明了一个可序列化类的可序列化域。它还允许声明并不作为类定义成员而存在的可序列化域,使类的实现域与它的序列化形式的可序列化域分离开来。

通过声明一个名为 `serialPersistentFields` 的私有静态最终域可以为一个类使用明确序列化。这个域是 `ObjectStreamField` 实例的一个数组,此实例包括域的字符串名及它们相应的 `Class` 对象。因为文档编制目的,一个这样的域应有一个相应的描述该域内容的 `@serialField` 标志(在它的 javadoc 中)。这些域被 `ObjectOutputStream.defaultWriteObject()` 和 `ObjectInputStream.defaultReadObject()` 方法所使用,除非对象定义了 `writeObject()/readObject()` 方法。既然如此, `writeObject()` 和 `readObject()` 将维护正被序列化的域。就像前面所提到的,此 `writeObject()` 方法应被文档化(通过在其 javadoc 中使用一个 `@serialData` 标志)。

在下面的示例中,类 `ClassA` 具有一个可串行域 `field1`。

```
Class ClassA implements Serializable {
    String field1;
    String field2;
    /**
     * @serialField field1 String
     *      Can be null, empty string or any String value.
     */
    private static final ObjectStreamField[] serialPersistentFields = {
        new ObjectStreamField("field1", String.class);
    }
    public ClassA(String f1, String f2) {
        field1 = f1;
        field2 = f2;
    }
}
```

当一个类使用明确序列化时,非静态和非暂态的域不被序列化,仅有那些在 `serialPersistentFields` 中指明的域被序列化。在前面的示例中,虽然 `field2` 即不是暂态的也不是静态的,但它也没有被序列化。

使用

像前面所示的那样,可以使用这个类在 `serialPersistentFields` 中声明明确可序列化域。一般不使用在此类中声明的任何一种方法;仅使用它的构造函数。这些方法由更高级的用户来使用,用于检查一个可序列化类的域(使用 `ObjectStreamClass.getFields()` 或 `ObjectStreamClass.getField()`)。

此类也同样被 `ObjectStreamClass`、`ObjectInputStream` 和 `ObjectOutputStream` 所使用,用于处理缺省及明确可序列化域(参见后面关于 `getName()` 示例的讨论)。

记录的有关可序列化域的信息

下面是记录的有关可序列化域的信息:

- 在源代码中声明的域名。
- 针对序列化目的,此类型域的完全合格的名称(例如,“`[[Ljava/lang/Object;`”, “`Ljava/lang/String;`”)。这被称为类型字符串。注意这个字符串是派生来的,但又不同

于由java.lang.Class.getName()返回的类型描述器(参见《The Java Class Libraries , Second Edition , Volume 1》中有关类描述中的类型描述器)。原始类型没有类型字符串。一个类的类型字符串是跟在 Class.getName()之后的一个 “ L ”,所有的点字符 (“ . ”)被斜杠字符 (“ / ”)所替代,而且后面跟着分号字符(“ ; ”)。一个数组的类型字符串是d个方括号字符 (“ [”)及后面跟着的数组类型字符串,其中d是数组的维数。表13包括类型字符串的示例。

表13 类型字符串

类型	类型字符串	类型	类型字符串
Object[][]	[[Ljava/lang/Object	double[]	[D
String	Ljava/lang/String	int[][]	[[I

域的类型代码。如果此域是非原始类型的,是类型字符串的第一个字符。如果域是原始类型的,则类型代码是原始类型的类型描述器 (例如,“ F ”对于float)。参见《The Java Class Libraries , Second Edition , Volume 1》中第366页的表13。表14列出了类型代码。

表14 类型代码

类型	类型代码	类型	类型代码
byte	B	long	J
char	C	class或interface	L
double	D	short	S
float	F	boolean	Z
int	I	array	[

- 表示域类型的Class对象。
- 表示正被序列化域的反射对象(java.lang.reflect.Field)。
- 实例数据中域的偏移量被序列化。

成员概述	
构造函数	
ObjectStreamField()	构造一个ObjectStreamField实例。
访问方法	
getName()	当ObjectStreamField已在原代码中声明,检索由其表示的可序列化域名。
getOffset()	在正被序列化的实例数据中检索由 ObjectStreamField表示的可序列化域偏移量。
getType()	检索由ObjectStreamField表示的可序列化域Class对象。
getTypeCode()	检索由ObjectStreamField表示的可序列化域类型代码。
getTypeString()	检索由ObjectStreamField表示的可序列化域的类型字符串。
isPrimitive()	确定由ObjectStreamField表示的可序列化域是否有一个原始类型。
setOffset()	在将被序列化的实例数据中设置由 ObjectStreamField表示的可序列化域偏移量。
比较方法	
compareTo()	比较此ObjectStreamField和另一ObjectStreamField的顺序。
对象方法	
toString()	返回描述此ObjectStreamField的字符串。

参见

ObjectStreamClass.getFields(), ObjectStreamClass.getField(),
ObjectInputStream.GetField, ObjectOutputStream.PutField.

参见《The Java Class Libraries, Second Edition, Volume I》中的Serializable。

示例

这个示例声明一个可序列化类 ClassA。ClassA声明了一唯一的明确可序列化域 field。MainWrite序列化ClassA的一个实例。

ClassA接着扩展成一个新的实现，在其中 field1被field10替代。为了适应此变化，被扩展的实现声明了一个 readObject()方法，首先在明确域中明确读，然后将其值赋给重命名的域 field10。

为了运行该示例，首先编译 ClassA.java-old和MainWrite.java，并运行MainWrite在文件 ClassA.ser中存储可序列化数据。接着编译 ClassA.java-new和MainRead.java。当运行MainRead时，它从文件ClassA.ser中读旧的可序列化数据，并将field1的序列化值赋给field10。

ClassA.java-old

```
import java.io.Serializable;
import java.io.ObjectStreamField;

public class ClassA implements Serializable {
    String field1;
    String field2;

    public ClassA(String one, String two) {
        field1 = one;
        field2 = two;
    }

    private static final long serialVersionUID = -563791230232631613L;

    /**
     * @serialField field1 String null, empty string, or any String value
     */
    private static final ObjectStreamField[] serialPersistentFields = {
        new ObjectStreamField("field1", String.class)};

    public String toString() {
        return ("field1 " + field1 + "; field2 " + field2);
    }
}
```

MainWrite.java

```
import java.io.*;

class MainWrite {
    public static void main(String[] args) {
        try {
            // Write them out
            FileOutputStream f = new FileOutputStream("ClassA1.ser");
            ObjectOutputStream out = new ObjectOutputStream(f);

            out.writeObject(new ClassA("hello", "there"));
            out.flush();
            out.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
    }
    }
}
```

ClassA.java-new

```
import java.io.*;

public class ClassA implements Serializable {
    String field10;
    String field2;

    public ClassA(String one, String two) {
        field10 = one;
        field2 = two;
    }
    private static final long serialVersionUID = -563791230232631613L;

    /**
     * @serialField field1 String null, empty string, or any String value.
     */
    private static final ObjectStreamField[] serialPersistentFields = {
        new ObjectStreamField("field1", String.class)};

    /**
     * @serialData write out field10's value for field1
     */
    private void writeObject(ObjectOutputStream out) throws IOException {
        // Get PutField instance for assigning values to serializable fields
        ObjectOutputStream.PutField pfields = out.putFields();

        pfields.put("field1", field10);

        // Write fields out
        out.writeFields();
    }

    private void readObject(ObjectInputStream in)
        throws IOException, ClassNotFoundException {
        // Read in serialized fields
        ObjectInputStream.GetField pfields = in.readFields();

        // Get the one we want
        field10 = (String)pfields.get("field1", "out-of-luck");
    }

    public String toString() {
        return ("field10 " + field10 + "; field2 " + field2);
    }
}
```

MainRead.java

```
import java.io.*;

class MainRead {
    public static void main(String[] args) {
        try {
            // Read it back
            FileInputStream f2 = new FileInputStream("ClassA1.ser");
            ObjectInputStream in = new ObjectInputStream(f2);
            ClassA obj = (ClassA) in.readObject();
            in.close();

            System.out.println(obj);
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {

```

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

```
    }  
}
```

compareTo()

目的	决定此ObjectStreamField与另一个ObjectStreamField的顺序。
语法	<code>public int compareTo(Object obj)</code>
描述	该方法决定此ObjectStreamField与另一个ObjectStreamField的顺序。原始类型的域在非原始类型的域之前。在每组中（原始的和非原始的），域根据它们的名字使用String.compareTo()来排序。
参数	
obj	为了排序而用来比较的非空对象。
返回	如果这个ObjectStreamField 在obj之前则返回-1；如果ObjectStreamField与obj有同样的名字则返回0；如果ObjectStreamField在obj之后则返回1。
异常	
ClassCastException	如果obj不是ObjectStreamField的一个实例。
NullPointerException	如果obj是null。
参见	Comparable.compareTo()。
示例	见java.lang.Comparable。

getName()

目的	当ObjectStreamField已在源代码中声明，检索由其表示的可序列化域名。
语法	<code>public String getName()</code>
返回	在源代码中用于命名表示ObjectStreamField可序列化域的非空字符串。
参见	java.lang.reflect.Field.getName()。
示例	这个示例用一串(缺省的)可序列化域声明一个ClassA类。为了示范目的，它重载readObject()来在可序列化域中手工读，然后使用ObjectStreamField中的方法来显示有关此域的信息。注意：非原始类型的类型描述器是java.lang.Object。这是因为，当为了在非原始类型的域中读入而创建ObjectStreamField实例时，序列化代码总是使用那个类。注意尽管ClassA使用缺省可序列化域，但它的readObject()方法可以使用readField()在域中取值。这不是必须的也不是推荐的，我们在这里用它仅仅是为了示范目的。

```
import java.io.*;  
import java.awt.Button;  
  
public class ClassA implements Serializable {  
    String strField;  
    int intField;  
    byte[] byteArrayField = new byte[] {(byte)3, (byte)2, (byte)1};  
    Button[] buttons = new Button[] {new Button("push me")};  
  
    public ClassA(String one, int two) {  
        strField = one;  
        intField = two;  
    }  
}
```



```

private void readObject(ObjectInputStream in)
    throws IOException, ClassNotFoundException {

    // Read in serial fields
    ObjectInputStream.GetField pfields = in.readFields();

    // Assign serialized values to fields
    strField = (String)pfields.get("strField", "out-of-luck");
    intField = (int)pfields.get("intField", 1000);
    byteArrayField = (byte[]) pfields.get("byteArrayField", null);
    buttons = (Button[]) pfields.get("buttons", null);

    // Get ObjectOutputStreamClass to get ObjectOutputStreamFields for each field
    ObjectOutputStreamClass objclass = pfields.getObjectStreamClass();
    ObjectOutputStreamField[] fields = objclass.getFields();

    // See what we got
    for (int i = 0; i < fields.length; i++) {
        print(fields[i]);
    }
}

private void print(ObjectStreamField fd) {
    System.out.println("name: " + fd.getName());
    System.out.println("type: " + fd.getType());
    System.out.println("type code: " + fd.getTypeCode());
    System.out.println("type string: " + fd.getTypeString());
    System.out.println("primitive: " + fd.isPrimitive());
    System.out.println("offset: " + fd.getOffset());
}

public String toString() {
    return "strField: " + strField +
        " intField: " + intField +
        " byteArrayField: " + byteArrayField +
        " buttons: " + buttons;
}
}

```

输出

```

java Main
name: intField
type: int
type code: I
type string: null
primitive: true
offset: 0
name: buttons
type: class java.lang.Object
type code: [
type string: [Ljava/awt/Button;
primitive: false
offset: 0
name: byteArrayField
type: class java.lang.Object
type code: [
type string: [B
primitive: false
offset: 1
name: strField
type: class java.lang.Object
type code: L
type string: Ljava/lang/String;
primitive: false
offset: 2
strField: hellointField: 500byteArrayField: [B@85d2bc05buttons:
[Ljava.awt.Button;@8616bc05

```

getOffset()

目的	在正被序列化的实例数据中检索由 <code>ObjectStreamField</code> 表示的可序列化域偏移量。
语法	<code>public int getOffset()</code>
描述	该方法在正被序列化的实例数据中检索由 <code>ObjectStreamField</code> 表示的可序列化域偏移量。所有需要被序列化的实例数据，可以被认为填充了一个字节数组。对于一个具有原始类型的域，相对于字节数组的开始而言，偏移量是这个域的原始数据的第一个字节位置。对于一个具有非原始类型的域，偏移量是 <code>n</code> ，此域的值是正在被序列化的第 <code>n</code> 个对象引用。如果此域是第 5 个非原始类型的域， <code>n</code> 将为 5。
返回	一个表示此域偏移量的 <code>int</code> 。
参见	<code>setOffset()</code> 。
示例	见 <code>getName()</code> 。

getType()

目的	检索由 <code>ObjectStreamField</code> 表示的可序列化域 <code>Class</code> 对象。
语法	<code>public Class getType()</code>
返回	由 <code>ObjectStreamField</code> 表示的可序列化域的非空 <code>Class</code> 对象。
参见	<code>getTypeString()</code> 、 <code>java.lang.Object.getClass()</code> 。
示例	见 <code>getName()</code> 。

getTypeCode()

目的	检索由 <code>ObjectStreamField</code> 表示的可序列化域类型代码。
语法	<code>public char getTypeCode()</code>
描述	参见类描述及表 14。
返回	由 <code>ObjectStreamField</code> 表示的可序列化域的类型代码。
示例	见 <code>getName()</code> 。

getTypeString()

目的	检索由 <code>ObjectStreamField</code> 表示的可序列化域的类型字符串。
语法	<code>public String getTypeString()</code>
描述	该方法检索由 <code>ObjectStreamField</code> 表示的可序列化域的类型字符串。类型字符串从域类名中派生而来。如果域是原始类型的，则返回空。参见类描述及表 13。
返回	如果此域有一个原始类型，则返回由 <code>ObjectStreamField</code> 表示的可序列化域的类型字符串或 <code>null</code> 。
参见	<code>getType()</code> 。
示例	见 <code>getName()</code> 。

isPrimitive()

目的	确定由 <code>ObjectStreamField</code> 表示的可序列化域是否有一个原始类型。
语法	<code>public boolean isPrimitive()</code>

返回	如果由 ObjectStreamField 表示的可序列化域有一个原始类型，则返回 true；否则返回 false。
示例	见 getName()。

ObjectStreamField()

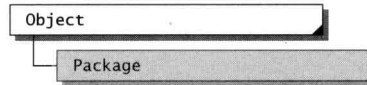
目的	构造一个 ObjectStreamField 实例。
语法	public ObjectStreamField(String fname, Class clazz)
描述	此构造函数构造一个 ObjectStreamField 实例。它被典型地用于 serialPersistentFields 的声明中。它为名为 fname、类型为 clazz 的类的可序列化域，创建一个新的 ObjectStreamField 实例。
参数	
clazz	由新的 ObjectStreamField 表示的可序列化域的非空 Class 对象。
fname	由新的 ObjectStreamField 表示的可序列化域的非空名。
示例	见类示例 getName()。

setOffset()

目的	在将被序列化的实例数据中设置由 ObjectStreamField 表示的可序列化域偏移量。
语法	protected void setOffset(int offset)
描述	此方法在将被序列化的实例数据中设置由 ObjectStreamField 表示的可序列化域偏移量。参看 getOffset() 可得到 offset 的描述。此方法被子类用来设置偏移量，并且应该仅由正在执行序列化的代码所用。
参数	
offset	由该 ObjectStreamField 确定的一个指定可序列化域偏移量的非负数。
参见	getOffset()。

toString()

目的	返回一个描述 ObjectStreamField 的字符串。
语法	public String toString()
描述	这个方法返回一个描述 ObjectStreamField 的字符串。该字符串由它的类型串或类型码及名字组成。
返回	一个描述 ObjectStreamField 的非空字符串。
重载	java.lang.Object.toString()。
示例	见 java.lang.Object.toString()。



语法

```
public class Package
```

描述

一个包对象代表一个被类加载器载入的 java 包。类加载器包含一个所有被加载过的类所属的包的列表。例如，如果一个类加载器载入 `java.beans.Beans`, `java.beans` 会出现在包列表中。

包对象不仅包含包的名字，而且包含其他与包相关的信息。此信息通常包含于含包代码的 java 文档文件(JAR)的名单中(见 Manifest)。

包对象最典型的应用是检验指定包的版本。例如，如果程序需要版本为 4.2 的包，则可以通过检验确保包的版本为 4.2 或更高。

使用

Package 对象只能由类加载器创建。如果需要一个 Package 对象，应将它从一个类或类加载器中检索出来。如果需要一个特定包的 Package 对象，该包的类必须在类加载器载入 Package 对象之前载入。

这种类支持两种检索 Package 对象的方法。第一种返回所有包对象的列表（参见 `getPackages()`）。第二种返回特定包的包对象(参见 `getPackage()`)。

规范和实现信息

每个包包含关于它所实现规范的信息和它本身实现时的信息。这些信息由包含该包的 JAR 文件的说明提供出来(参见 Manifest 以获得更多信息)。规范和执行信息分别被分成三部分：标题、版本和厂商。这些信息足以用来区分包的规范或实现信息以及和厂商联系。

在全部六部分信息中，只有规范版本有语法要求。

规范版本

规范版本有特定的语法规则，这样各规范版本之间可以相互比较。这种特性允许在使用一个包之前检查包的兼容性。例如，如果程序需要版本 4.2 的包，但使用的版本是 4.1 则程序会出错。详情参见 `isCompatibleWith()`。

规范版本是一个由小数点分隔的非负十进制数序列。如，“1.2.3.0”是一个有效的规范版本。一个规范版本如果比 v 版本低，则与 v 版本兼容。这种检查通过从左至右比较序各位数字来实现。如果规范版本中的数字比 v 版本中的相应数字小，那么该版本与 v 版本不兼容。

如果一个版本字符串比另一个短，可以看作短的那个版本以 “.0” 结尾。就是说版本 “1.2.0” 与 “1.2” 兼容。

封装包

默认情况下，如果有一个含类的包，就能将该包中的一些文件放入一个 JAR 文件,其余的放入另一个 JAR 文件。如果将两个不兼容的同名包都在同一个类的路径中，将可能导致不可预期的结果。

可以通过封装一个包来防止这个问题。当封装了一个包后，该包中的所有类必须定位在一个 JAR 文件中。当类加载器从一个 JAR 文件的封装包中载入一个类时,它会拒绝从其他 JAR 文件中载入相同的包。相反，当从未封装的包中载入一个类。类加载器将拒绝从同名的封装包中载入类，当这类错误发生时，类加载器会抛出 SecurityException。

要注意封装包对类加载器寻找一个类没有影响。例如，可能会以为一旦类加载器被从 JAR 文件的封装包中加载，当它需要包中的某个类时，会在该 JAR 文件中查找。这种假设是错误的。类加载器查找某个类时并不关心该类是否被加载，只有当类加载器找到该类时，才会检查是否被允许加载它。当检查失败，SecurityException 会被抛出。

缺省包(没有名字的包)不能被封装。
参见 Attribute.Name.SEALED 关于怎样封装一个包。

成员概述	
包检索方法	
getPackage()	检索特定包的 Package 对象。
getPackages()	检索当前已知包的列表。
对象方法	
hashCode()	计算 Package 对象的哈希码。
toString()	产生代表该包的字符串。
查询方法	
getImplementationTitle()	检索该包的实现名称。
getImplementationVendor()	检索该包的实现的厂商。
getImplementationVersion()	检索包的实现版本号。
getName()	检索包的名字。
getSpecificationTitle()	检索包实现的规范名。
getSpecificationVendor()	检索包实现的规范的厂商。
getSpecificationVersion()	检索包实现的规范版本号。
isCompatibleWith()	比较该包与指定包的版本号。
isSealed()	确定包是否被封装。

参见

Class.getPackage(), ClassLoader.getPackage(), java.util.jar.JarFile, java.util.jar.Manifest。
示例 这个示例演示了怎样设定和检索一个包的规范和执行信息。程序声明了包含一个简单类 C 的包 p。

```
p/C.java
package p;

public class C {
}
```

这个类使用下面的名单加入一个JAR文件，包含了包p的所有规范和实现信息。

p/manifest.mf

Manifest-Version: 1.0

Name: p/
Specification-Title: Demo API Specification
Specification-Version: 1.2.3
Specification-Vendor: Chan, Lee, Kramer
Implementation-Title: package.example
Implementation-Version: 5.3.b1
Implementation-Vendor: Xeo Enterprises

Main.java

```
class Main {
    public static void main(String[] args) {
        // Load p.C.
        new p.C();

        Package[] pkgs = Package.getPackages();

        for (int i=0; i<pkgs.length; i++) {
            System.out.println(pkgs[i].getName());
            System.out.println(" Specification Title: "
                + pkgs[i].getSpecificationTitle());
            System.out.println(" Specification Version: "
                + pkgs[i].getSpecificationVersion());
            System.out.println(" Specification Vendor: "
                + pkgs[i].getSpecificationVendor());
            System.out.println(" Implementation Title: "
                + pkgs[i].getImplementationTitle());
            System.out.println(" Implementation Version: "
                + pkgs[i].getImplementationVersion());
            System.out.println(" Implementation Vendor: "
                + pkgs[i].getImplementationVendor());

            if (pkgs[i].isSealed()) {
                System.out.println(" Sealed: true");
            }
        }
    }
}
```

输出

第一个命令创建包含P.C的JAR文件。第二个命令运行Main并产生下列输出。

```
> jar cfm c.jar p/manifest.mf p/C.class
> java -cp c.jar;. Main
p
Specification Title:      Demo API Specification
Specification Version:    1.2.3
Specification Vendor:     Chan, Lee, Kramer
Implementation Title:     package.example
Implementation Version:   5.3.b1
Implementation Vendor:    Xeo Enterprises
java.net
Specification Title:      Java Platform API Specification
Specification Version:    1.2
Specification Vendor:     Sun Microsystems, Inc.
Implementation Title:     Java Runtime Environment
Implementation Version:   1.2beta4
Implementation Vendor:    Sun Microsystems, Inc.
java.util.jar
Specification Title:      Java Platform API Specification
```

```
Specification Version: 1.2
Specification Vendor:  Sun Microsystems, Inc.
Implementation Title:  Java Runtime Environment
Implementation Version: 1.2beta4
Implementation Vendor:  Sun Microsystems, Inc.
...    <removed for brevity>
```

getImplementationTitle()

目的	检索包实现名。
语法	<code>public String getImplementationTitle()</code>
描述	实现标题识别包的详细实现。例如：如果厂商提供几种平台上的实现，平台的名称应该包含在标题中。
返回	包执行的可能为空的标题。
示例	见类的示例。

getImplementationVendor()

目的	检索包实现的厂商。
语法	<code>public String getImplementationVendor()</code>
描述	这个方法返回负责包实现的组织、公司，或个人的标识。这个字符串对于使用该包的用户与厂商联系是足够的。
返回	包实现的厂商(可能为null)。
示例	见类的示例。

getImplementationVersion()

目的	检索包实现的版本号。
语法	<code>public String getImplementationVersion()</code>
描述	这个方法返回标识包实现的版本号的字符串。返回字符串的格式严格按照实现所有者的规定，例如：“1.2beta4.1 7/9/1998”和“build 1003”。
返回	包实现的版本号(可能为null)。
示例	见类的示例。

getName()

目的	检索包的名字。
语法	<code>public String getName()</code>
返回	包的非空完整名称(例如“java.net”)。
示例	见类的示例。

getPackage()

目的	检索特定包的Package对象。
语法	<code>public static Package getPackage(string pname)</code>
描述	这个方法在调用程序的类加载器的包列表中查找名为 pname的包，返回代表该包的包对象。如果在类加载器的列表中没有找到 pname，将会在类加载器的父类中查找，若还没有找到，上溯到祖先类加载器层次。如果调用程序的类加载器为空(即Class.getClassLoader()返回空值)，将查找

系统类加载器的列表。

参数

pname

包的非空的完整文件名(例如：“java.net”)

返回

代表包pname的Package对象。如果pname未找到，返回null。

参见

Class.getPackage()、ClassLoader.getPackage()。

示例

见getPackages()。

getPackages()

目的

检索当前已知包的列表。

语法

```
public static Package[] getPackages()
```

描述

这个方法检索调用程序的类加载器的包列表。如果调用程序的类加载器为空(即Class.getClassLoader()返回空值)，将返回系统类加载器的包列表。返回的列表中不能有两个同名的包对象。包列表是被类加载器及其祖先载入的。(某类加载器的父类是加载它的类加载器)。

返回

一个新的代表每个已知包的非空包对象指针。

参见

Class.getPackage()、ClassLoader.getPackage()。

示例

这个示例演示两个不同的类加载器向包列表中载入包的影响，一个类加载器是缺省的系统类加载器，(参见ClassLoader),另一个是载入Main类的类加载器。在我们的示例中，Main从一个JAR文件中载入p.C并且，p.C载入java.beans.Beans。
从系统加载器得到包列表的方法是专用的。进行处理时为了遵守这项限制，程序使用了反射来调用这一方法(involveMethod())。(不要担心，这不是一处安全漏洞，应用程序有允许或禁止这种访问的能力。)

```
import java.lang.reflect.*;
import java.util.*;

class Main {
    public static void main(String[] args) {
        // Load in the p.C class.
        new p.C();

        // Dump the package list for Main's class loader.
        System.out.println("Packages loaded by Main's class loader");
        Package[] pkgs = Package.getPackages();
        printPackages(pkgs);

        System.out.println(Package.getPackage("p")!=null);    // true

        // Now dump the package list of the system class loader.
        System.out.println("Packages loaded by the system class loader.");
        pkgs = (Package[])invokeMethod(
            Package.class, "getSystemPackages", new Object[0]);
        printPackages(pkgs);

        System.out.println(invokeMethod(ClassLoader.getSystemClassLoader(),
            "getPackage", new Object[]{"p"})!=null);    // false
    }

    public static void printPackages(Package[] pkgs) {
        // First sort the list.
        Arrays.sort(pkgs,
            new Comparator() {
```

```

        public int compare(Object a, Object b) {
            return ((Package)a).getName().compareTo(
                ((Package)b).getName());
        }
    }
};
for (int i=0; i<pkgs.length; i++) {
    System.out.print(" " + pkgs[i].getName());
    if (pkgs[i].isSealed()) {
        System.out.print(" *sealed*");
    }
    System.out.println();
}
}

public static Object invokeMethod(
    Object o, String name, Object[] params) {
    // Create parameter type array.
    Class[] paramTypes = new Class[params.length];
    for (int i=0; i<params.length; i++) {
        paramTypes[i] = params[i].getClass();
    }

    // Retrieve the object o's class, unless the o
    // is itself a class.
    Class c = o instanceof Class ? (Class)o : o.getClass();

    // Find the declared method by checking all superclasses as well.
    while (c != null) {
        try {
            Method m = c.getDeclaredMethod(name, paramTypes);
            m.setAccessible(true);

            // Invoke the method and return the results.
            return m.invoke(o, params);
        } catch (NoSuchMethodException e) {
            c = c.getSuperclass();
        } catch (Exception e) {
            break;
        }
    }
    return null;
}
}
}

```

输出 JAR文件中p.C类被替换为c.jar。注意p在Main的类加载器中，而不在系统类加载器中。

```

> java -cp .;c.jar Main
Packages loaded by Main's class loader
java.beans
java.io
java.lang
java.lang.ref
java.lang.reflect
java.net
java.security
java.util
java.util.jar
java.util.zip
p
sun.io
...
true

```

<list truncated for brevity>

```

Packages loaded by the system class loader.
java.beans
java.io
java.lang

```

```
java.lang.ref
java.lang.reflect
java.net
java.security
java.security.cert
java.util
java.util.jar
java.util.zip
sun.io
...
false
```

<list truncated for brevity>

getSpecificationTitle()

目的	检索包实现的规范的名字。
语法	<code>public String getSpecificationTitle()</code>
描述	这个方法返回标识包实现的特定规范的规范名。例如，一个包可能有两个规范——基本规范和高级规范——其高级版本包含更多的类。这些信息都将包含在规范名之中。也可以假设提供商为不同的平台的规范提供不同的实现版本。如果其所有行为相同，且规范的实现也相同，那么规范的标题也应相同。
返回	这个包实现的规范名(可能为null)。
示例	见类示例。

getSpecificationVendor()

目的	检索包实现的规范的厂商名。
语法	<code>public String getSpecificationVendor()</code>
描述	这个方法返回负责包实现规范的组织、公司或个人的识别信息。这些信息足够给用户来与厂商联系。
返回	包的规范的厂商名(可能为null)。
示例	见类示例。

getSpecificationVersion

目的	检索包实现的规范版本号。
语法	<code>public String getSpecificationVersion()</code>
描述	规范版本号是一个由小数点分隔的非负十进制数序列。如“ 1.2.3.0 ”是一个有效的规范版本号。关于规范版本的更多信息参见类描述。
返回	包实现的规范版本号(可能为null)。
参见	<code>isCompatibleWith()</code> 。
示例	见类的示例。

hashCode()

目的	计算Package对象的哈希码。
语法	<code>public int hashCode()</code>
描述	Package对象的哈希码严格依赖于包的名字。两个同名的包对象将具有相同的哈希码。虽然哈希码算法减少了这种可能,但不同的包对象还是可能具有相同的哈希码。哈希码典型地作为哈希表中的关键字使用。
返回	包对象的哈希码。

重载	java.lang.Object.hashCode()
参见	equals()、java.lang.Object.equals()。
示例	见java.lang.Object.hashCode()。

isCompatibleWith()

目的	将包的规范版本与指定版本比较。
语法	public boolean isCompatibleWith (String v) throws NumberFormatException
描述	<p>这个方法将包的规范版本号与 v 比较。如果包的规范版本号比 v 小，该方法返回 false，否则返回 true。</p> <p>规范版本号是由小数点分隔的非负十进制数序列。如 “ 1.2.3.0 ” 是一个有效的规范版本。规范版本号从左至右与 v 比较序列中相应位置的数字。当规范版本号中的数字比 v 中的相应数字小，那么规范版本比 v 低，立即返回 false。</p> <p>如果一个版本字符串比另一个短，可以看作短的那个版本以 “ 0 ” 结尾，就是说版本 “ 1.2.0 ” 与 “ 1.2 ” 兼容。</p> <p>在类加载器载入一个包对象之前，它必须首先从该包中加载类。如果确切地知道包是怎样打包的(例如，在一个 JAR 文件中)，可以在任何包中的类被加载之前，决定包的规范版本号。但是强迫类加载器从包中加载类，然后检验它的规范版本号是更方便的。</p>
返回	如果包的规范版本号高于或等于 v，返回 true。
异常	
NumberFormatException	<p>如果一个组件在规范版本号中，或 v 不是个非负十进制数。</p>
参见	getSpecificationVersion()。
示例	

```
// Load p.C.
new p.C();

Package p = Package.getPackage("p");
if (p == null) {
    System.out.println("Package p is not found.");
    System.exit(1);
}
System.out.println("Specification Version of p: "
    + p.getSpecificationVersion());           // 1.2.3

System.out.println( p.isCompatibleWith("1") );           // true
System.out.println( p.isCompatibleWith("1.0") );        // true
System.out.println( p.isCompatibleWith("1.2") );        // true
System.out.println( p.isCompatibleWith("1.2.3") );      // true
System.out.println( p.isCompatibleWith("1.2.3.0") );    // true
System.out.println( p.isCompatibleWith("1.2.3.0.0.0") ); // true
System.out.println( p.isCompatibleWith("1.2.3.1") );    // false

System.out.println( p.isCompatibleWith("0") );          // true
System.out.println( p.isCompatibleWith("2") );          // false
System.out.println( p.isCompatibleWith("1.beta") );     // NumberFormatException
```

isSealed()

目的	确定包是否封装。
----	----------

语法	<pre>public boolean isSealed() public boolean isSealed(URL url)</pre>
描述	<p>如果包用url关系封装，返回 true。这表示类加载器只能从 url所指JAR文件中加载类。关于封装包的更多信息，参见类描述。</p> <p>如果url没有定义，且包被封装，方法返回 true。</p>
参数	
url	非空的URL对象。
返回	如果包用url关系封装或未指定，返回 true。
示例	<p>这个例子演示了怎样通过构造试图中断一个工作的脚本来封装该工作。</p> <p>首先，我们声明两个类 Far和Near的包p，类Far放入一个JAR文件，该文件说明包p被封装。封装JAR文件安装在一个远程点。类Near被包括在同样的JAR文件中，作为Main类。Main程序首先载入Far。这导致类加载器为p建立一个Package对象，并用JAR文件的位置关系来封装它。</p> <p>接着Main程序加载Near。类加载器找到p.Near.class并且确定它的包为p。然后类加载器声明p被封装，确定p.Near.class不是从封装的URL中得到，并抛出一个SecurityException。</p> <p>如果Near在Far之前加载，会发生同样的错误。</p>

toString()

Main.java

```
import java.net.*;

class Main {
    public static void main(String[] args) {
        new p.Far();

        try {
            Package pkg = Package.getPackage("p");

            System.out.println( pkg.isSealed() );
            // true
            System.out.println( pkg.isSealed(new URL("http://www.xeo.com")) );
            // false
            System.out.println(
                pkg.isSealed(new URL("http://www.xeo.com/bookegs/far.jar")) );
            // true
        } catch (MalformedURLException e) {
            e.printStackTrace();
        }

        // Now cause the failure.
        new p.Near();
    }
}
```

main.mf

以下是main.jar文件的名单

```
Manifest-Version: 1.0
Main-Class: Main
Class-Path: http://www.xeo.com/bookegs/far.jar
```

p/Near.java

```
package p;

public class Near {
    public void print() {
        System.out.println(getClass());
    }
}
```

p/Far.java

```
package p;

public class Far {
    public void print() {
        System.out.println(getClass());
    }
}
```

p/far.mf This is the manifest for the far.jar file.

Manifest-Version: 1.0

Name: p/

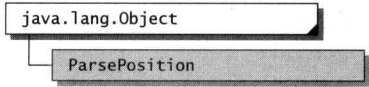
Sealed: true

输出

```
> java -jar main.jar
true
false
true
Exception in thread "main" java.lang.SecurityException: sealing violation
    at java.net.URLClassLoader.defineClass(URLClassLoader.java:176)
    at java.net.URLClassLoader.access$1(URLClassLoader.java:156)
    at java.net.URLClassLoader$1.run(URLClassLoader.java:137)
    at java.security.AccessController.doPrivileged(Native Method)
    at java.net.URLClassLoader.findClass(URLClassLoader.java:131)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:245)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:195)
    at Main.main(Main.java:16)
```

目的	产生一个代表包的字符串。
语法	public String toString()
描述	该方法产生一个包含包的名字和所有规范信息的字符串。
返回	代表包的非空字符串。
重载	Object.toString()。
示例	

```
Package p = Package.getPackage("java.lang");
if (p == null) {
    System.out.println("Package p is not found.");
    System.exit(1);
}
System.out.println(p);
// package java.lang, Java Platform API Specification, version 1.2
```



语法

```
public class ParsePosition extends Object
```

描述

解析是识别一个数字串模式的操作，并返回数字、日期或消息对象。ParsePosition是由Format使用的类，它的子类在解析时用一个字符串来跟踪记录当前位置。下标属性记录当前位置。见图 12。设计时，可用不同的Format对象解析字符串，所有的Format对象可以使用相同的ParsePosition。ParsePosition对象跟踪统一码字符，因而这是非场所敏感的。

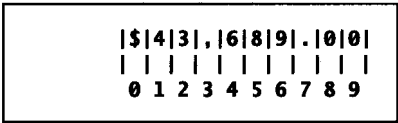


图12 ParsePosition

使用一个ParsePosition对象的类包括Format和它的子类。例如：当对 NumberFormat及它的子类：DecimalFormat、DateFormat、SimpleDateFormat和MessageFormat执行parse()方法时，parseobject()方法产生一个ParsePosition对象。

关于更多该类的信息，参见《The Java Class Libraries，Second Edition，Volume 1》。

版本1.2中所作的修改

这里有两个新的获取/设置方法：getErrorIndex()和setErrorIndex()。这个类也重载了Object方法：equals()、hashCode()和toString()

成员概述	
构造函数	
ParsePosition()	构造一个ParsePosition实例，具有一个初始下标。
解析方法	
getIndex()	检索当前解析位置。
setIndex()	设置当前解析位置。
1.2 getIndex()	检索解析错误发生的下标。
1.2 setErrorIndex()	设置解析错误发生的下标。
对象方法	
Δ toString()	产生ParsePosition实例的字符串表示。
Δ equals()	比较ParsePosition实例与另一对象是否相等。
Δ hashCode()	计算ParsePosition实例的哈希码。

参见

Format。

《The Java Class Libraries, Second Edition, Volume 1》中的ParsePosition。

▲ equals()

目的	比较ParsePosition实例与另一对象是否相等。
语法	public boolean equals(object obj)
描述	本方法比较 ParsePosition实例与 obj是否相等。如果 obj是一个 ParsePosition对象并且具有与前者相同的当前下标值及错误下标，则两对象相等，本方法返回 true。如果ParsePosition的值不相等，或obj为null或不是ParsePosition对象，本方法返回 false。
版本1.2中的改动	在版本1.1中，本方法不在ParsePosition中实现，而是在Object中调用。
参数	
obj	进行比较的对象(可能为null)。
返回	如果obj非空，为ParsePosition类型，并且与本ParsePosition实例相等，其他情况则返回 false。
重载	java.lang.Object.equals()。
参见	hashCode()。
示例	这个示例建立两个 ParsePosition实例，判断它们是否相等，然后比较它们的哈希码。

```
import java.text.ParsePosition;

class Main {
    public static void main(String args[]) {
        // Create two parse positions.
        ParsePosition parsePos1 = new ParsePosition(0);
        System.out.println(parsePos1.toString());
        // java.text.ParsePosition[index=0,errorIndex=-1]

        ParsePosition parsePos2 = new ParsePosition(0);
        System.out.println(parsePos2.toString());
        // java.text.ParsePosition[index=0,errorIndex=-1]

        // Tests for equality.
        System.out.println( parsePos1.equals(parsePos2) ); // true

        // Compute hashcode.
        System.out.println( parsePos1.hashCode() );          // -65536
    }
}
```

getErrorIndex()

目的	检索解析错误发生的下标。
语法	public int getErrorIndex()
返回	返回一个整型的错误发生处的下标。或错误下标未设置时返回 - 1。
参见	setErrorIndex()。

hashCode()

目的	计算ParsePosition实例的哈希码。
语法	public int hashCode()
描述	本方法计算一个整型的哈希码，是 ParsePosition实例基于当前下标和错误下标得到的，两个相等的 ParsePosition实例将有相同的哈希码。但是，尽管哈希码算法已经减小了这种可能，不具有相同属性的两个 ParsePosition实例还是可能具有相同的哈希码。哈希码典型地被作为关键字用于哈希表中。
版本1.2中的改动	版本1.1中，本方法不在 ParsePosition中实现，而是在 Object中调用。
返回	ParsePosition实例的哈希码。
重载	java.lang.Object.hashCode()。
参见	equals()。
示例	见equals()。

SetErrorIndex()

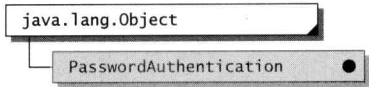
目的	设置解析错误发生的下标。
语法	public void setErrorIndex(int index)
描述	本方法设置解析错误发生的下标。格式化对象在从 parseObject()方法返回错误码之前，应设置本方法。如果未设置，缺省值则为 - 1。
参数	
index	解析错误发生处的字符下标，一个整数。
参见	getErrorIndex()。

toString()

目的	生成代表ParsePosition实例的字符串。
语法	public String toString()
描述	本方法返回代表 ParsePosition实例的字符串，包括类名称，当前下标，和错误下标。这是此类字符串的一个示例： java.text.ParsePosition [index=0,errorIndex=-1]
版本1.2中的改动：	在版本1.1中，本方法不在 ParsePosition中执行，而是在 Object中调用。
返回值	表示ParsePosition实例的非空字符串。
重载	java.lang.Object.toString()。
示例	见equals()。

java.net

PasswordAuthentication



语法

Public final class PasswordAuthentication

描述

passwordAuthentication类表示一个名字/口令对，它被 Authenticator类用来返回认证信息。

使用

PasswordAuthentication类被 Authenticator类用来返回信息，如：静态方法 requestPasswordAuthentication()的调用返回一个 PasswordAuthentication实例，从中能提取用户名和口令。

```

PasswordAuthentication pw = Authenticator.requestPasswordAuthentication(
    server, 80, "http", null, "basic");
// use password
...
// clear it out after we're done
char pwbuf = pw.getPassword();
for (int i = 0; i < pwbuf.length; i++) {
    pwbuf[i] = (char)0;
}
  
```

成员概述	
构造函数	
PasswordAuthentication()	使用用户名和口令构造一个 PasswordAuthentication实例。
访问方法	
getPassword()	检索 PasswordAuthentication实例的口令。
getUserName()	检索 PasswordAuthentication实例的用户名。

参见

Authenticator。

示例

参见 Authenticator类的示例。

getPassword()

目的 检索 PasswordAuthentication实例的口令。

语法	<code>public char[] getPassword()</code>
描述	本方法返回一个被用来存放口令的直接指向字符数组的指针。如果调用程序改变了数组，同样会在 <code>PasswordAuthentication</code> 实例中发生变化。为了将口令从内存中清除（保密起见），调用程序在不需要该数组时，将数组内容清零。
返回	返回包含口令的非空数组。
参见	<code>getUserName()</code> 。
示例	见 <code>Authenticator</code> 类的示例。

getUserName()

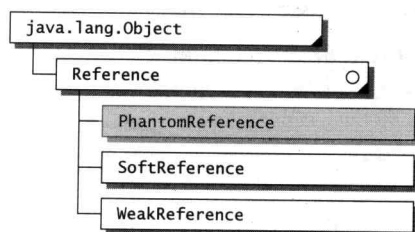
目的	检索 <code>PasswordAuthentication</code> 实例的用户名。
语法	<code>public String getUserName()</code>
返回	包含用户名的非空字符串
示例	见 <code>Authenticator</code> 类的示例。

PasswordAuthentication()

目的	用用户名和口令构造一个新的 <code>PasswordAuthentication</code> 实例。
语法	<code>public PasswordAuthentication(String userName, char[] password)</code>
描述	本方法用用户名和口令构造一个新的 <code>PasswordAuthentication</code> 实例。创建一个口令数组的拷贝。调用程序应尽可能在不再需要口令时，使用清零数组的方法来保证口令的安全性。这不会影响新建的 <code>PasswordAuthentication</code> 实例的内容。
参数	
password	含口令的非空数组。
userName	识别和口令对应的实体的非空字符串。
示例	见 <code>Authenticator</code> 类的示例。

java.Lang.ref

PhantomReference



语法

```
public class PhantomReference extends Reference
```

描述

PhantomReference类表示一个幻像引用。幻像引用是一种引用对象(参见Reference),可以用来在对象结束后,回收前接收一个对象的声明信息。死亡在《The Java Language Specification, First Edition》中的第12.6节有描述)。见图13。

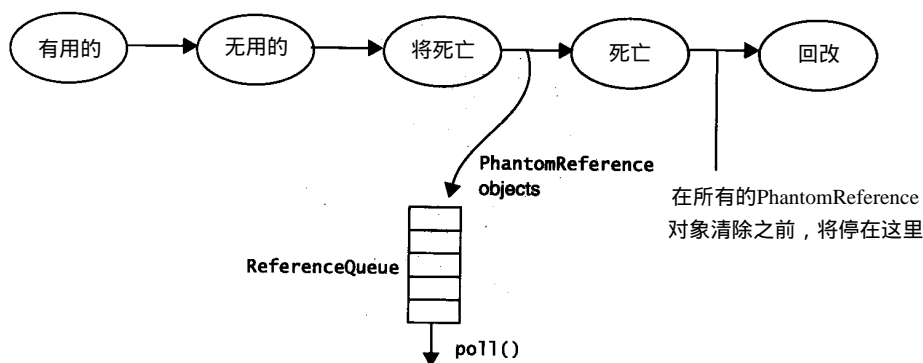


图13 幻像引用声明

幻像引用声明与弱引用声明非常相似。可以使用任一种引用类型来确定何时对象被收集。仅当一个对象能再生时,才与所选择的那种引用有关。(再生发生在对象在结束时又调用自身。参见《The Java Language Specification, First Edition》中第12.6节)特别是,如果当一个对象是可再生的并且想知道对象何时被再声明,必须使用幻像引用。如果对无用信息收集何时收集一个再生对象感兴趣,必须使用一个弱引用。然而,再生很少被使用(也不提倡)所以这种情况很少见。

幻像引用与其他引用类型不同,为了对象能够回收,必须明确地清除它(即,它的存储器是返回的堆栈)。参见图13。然而,当幻像引用自己被收集时,它会自动清除。

使用

为了接收对象obj何时被回收的信息,首先要建立一个引用队列(参见ReferenceQueue)。然

后构造一个幻像引用对象，提供 obj和引用队列。最后，查询引用队列。当幻像引用出现在队列中，obj将被回收。下面是演示这些步骤的示例：

```
ReferenceQueue q = new ReferenceQueue();
PhantomReference pr = new PhantomReference(obj, q);
try {
    Reference r = rq.remove();
    r.clear();           // Free the object for final reclamation
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

成员概述

构造函数

PhantomReference() 构造一个PhantomReference实例。

引用检索方法

get() 返回null。

参见

Reference、ReferenceQueue、SoftReference、Weak Reference

示例

这个示例建立了一个监视对象变为未用的进程。每个对象有一个标签，对象标签保存在一个集合中，当一个对象变为未使用，进程从集合中删除它的标签。

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

class Main {
    public static void main(String[] args) throws IOException {
        Set isUsed = new HashSet();
        Object o = new Object();
        WatcherThread thread = new WatcherThread(isUsed);
        thread.start();

        thread.watch("A", o);
        System.out.println( isUsed.contains("A") );    // true

        // Wait a few seconds before making o unused.
        try {
            Thread.sleep(5000);
        } catch (Exception e) {
            e.printStackTrace();
        }
        o = null;

        // Use all available memory to force a garbage-collection.
        eatMemory(new LinkedList());

        System.out.println( isUsed.contains("A") );    // false
    }
    static void eatMemory(LinkedList l) {
        try {
            l.add(new double[100000]);
            eatMemory(l);
        }
    }
}
```

```
        } catch (OutOfMemoryError e) {
            System.out.println("**** OUT OF MEMORY ****");
            l.clear();
            System.gc();
        }
    }
}

class WatcherThread extends Thread {
    ReferenceQueue refq = new ReferenceQueue();
    Map map = new HashMap();
    Set set;

    public WatcherThread(Set s) {
        set = s;
    }

    public void watch(String label, Object o) {
        Reference r = new PhantomReference(o, refq);
        set.add(label);
        map.put(r, label);
    }

    public void run() {
        try {
            while (true) {
                // Wait for phantom reference notification.
                Reference r = refq.remove();

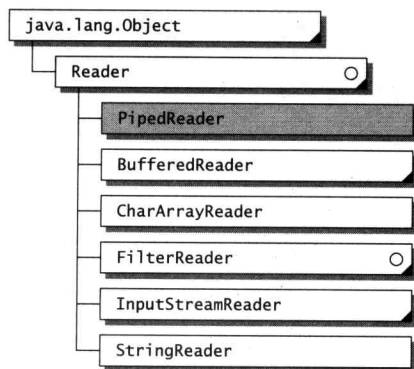
                r.clear();
                set.remove(map.get(r));
                map.remove(r);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

get()

目的	返回null。
语法	public Object get()
返回	本方法总是返回null。
重载	Reference.get()。
示例	见Reference。

PhantomReference()

目的	构造一个PhantomReference实例。
语法	public PhantomReference(Object referent, ReferenceQueue queue)
描述	本构造函数构造一个幻像引用实例。无用信息收集将在 referent结束后将新的PhantomReference实例放入队列queue中。
参数	
queue	非空引用队列。
referent	非空对象。
示例	见类示例。



语法

```
public class PipedReader extends Reader
```

描述

PipedReader类用于向一个管道(pipe)提供字符输入流。管道是在两个进程通信之间通信的非常有用的编程范型。要得到更多关于管道用法的内容请参考 PipedInputStream。

PipedReader类与向管道中写入字符的PipedWriter类是一对。

要得到更多的关于该类的信息请参考《The Java Class Libraries, Second Edition, Volume 1》。

版本1.2中所作的修改

ready()方法重载了Reader.ready(), 由数据是否可从管道中读出来决定返回一个 true或false。在版本1.1中ready()从Reader中继承而且总是返回 false。


这个类的实现已经作了修改, 它做更多的错误检查并记录下向管道中写数据的线程的状态。

成员概述

构造函数

PipedReader() 构造一个PipedReader实例。

输入方法

 read() 从该PipedReader中读取字符。

 ready() 确定这个PipedReader是否可读。

管道方法

close() 关闭这个PipedReader。

connect() 将该PipedReader同一个管道写入者相连。

参见

PipedWriter。

《The Java Class Libraries, Second Edition, Volume 1》中的PipedReader。

▲ read()

目的	从PipedReader中读取字符。
语法	<pre>public synchronized int read() throws IOException public synchronized int read(char buf[],int offset,int count)throws IOException</pre>
描述	该方法的第一种形式从该 PipedReader中读取一个字符。第二种形式从PipedReader中在偏移量 offset处开始读出 count个字符到 buf中。在输入数据可以获得前,或是数据流结束前,或是抛出了一个异常,这个方法会被阻塞。
版本1.2中的改动	在版本 1.1 中从Reader中继承了不带参数形式的方法。在版本 1.2中,该形式被重载。同时,该方法现在的实现对于管道的状态做更多的检查,如管道是否连接和关闭,处于活动状态的线程是否向管道中写入了数据。
参数	
buf	存储读取字符的非空字符数组。
count	要读取的字符的个数。0 <= count <= buf.length-offset。
offset	buf中开始存储字符的下标。0 <= offset <= buf.length。
返回	该方法的第一种形式返回读到的字符;第二种形式返回读到的字符的个数。如果到达了数据流的末尾还没有读到一个数据,则这两种形式的方法都返回 - 1。
异常	
ArrayIndexOutOfBoundsException	如果count或offset超出了指定的范围。
InterruptedException	如果在从管道中读取数据的时候当前的线程被中断。
IOException	如果没有连接上管道,或是管道已经关闭,或是向管道写数据的线程没有激活。
重载	Reader.read()。
示例	见《The Java Class Libraries,Second Edition,Volume 1》中的PipedReader类的示例。

▲ ready()

目的	确定该PipeReader是否作好了接收读数据的准备。
语法	<pre>public synchronized boolean ready() throws IOException</pre>
描述	该方法确定PipedReader是否作好接收读数据的准备。如果管道中有数据要读则PipedReader作好了读数据的准备。
版本1.2中的改动	在版本1.1中PipedReader 从Reader中继承了ready()方法,且Reader.ready()总是返回false。
返回	如果管道中有要读的数据就返回 true,否则返回 false。

异常

IOException

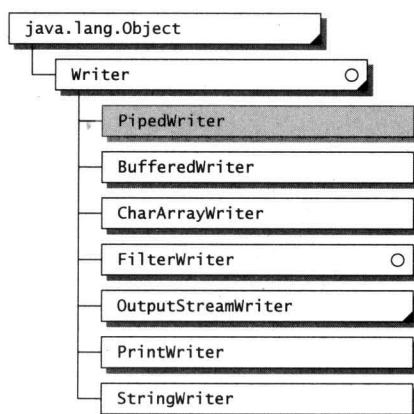
如果管道没有连接，或者已被关闭了，或者是向管道写数据的线程没有激活。

重载

Reader.ready()。

Java.io

PipedWriter



语法

```
public class PipedWriter extends Writer
```

描述

PipedWriter用于向管道提供一个字符输出流。管道是在两个进程通信之间通信的非常有用的编程范型。要得到更多的关于使用管道的知识请参考 PipedInputStream。PipedWriter与从管道中接收数据的PipedReader是一对。

要得到更多的关于该类的信息请参考《The Java Class Libraries , Second Edition,Volume 1》。

版本1.2中所作的修改

这个类的实现已经作了修改，它执行更多的错误检查并记录下向管道中写数据的线程的状态。

成员概述

构造函数

PipedWriter() 构造一个PipedWriter()实例。

管道方法

close() 关闭这个PipedWriter。

connect() 将该PipedWriter同一个管道读取者相连。

flush() 刷新该PipedWriter。

输出方法

 write() 向该PipedWriter中写入一个或多个字符。

参见

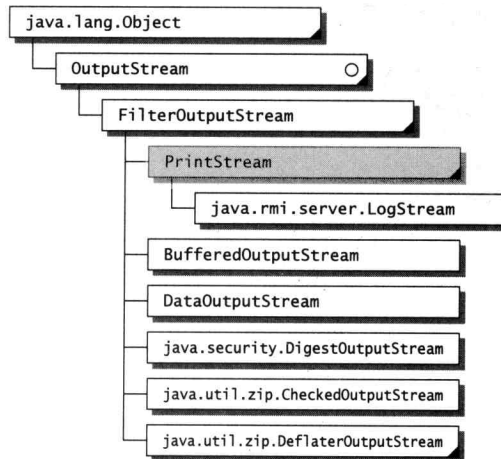
PipedReader。

《The Java Class Libraries, Second Edition, Volume 1》中的PipedWriter。

▲ write()

目的	向该PipedWriter中写入一个或多个字符。
语法	<pre>public void write(int ch) throws IOException public void write(char[] but,int offset, int count) throws IOE xception</pre>
描述	<p>该方法的第一种形式写入 ch的低16位。第二种形式向该 PipedWriter写入 count个从字符数组buffer的下标offset处开始的字符。</p> <p>写入该PipedWriter中的字符可以从连接到该 PipedWriter的管道读取者中读出。没有从管道读取者中读出的数据存储在管道的缓冲区中。如果管道充满，write()方法将阻塞。</p>
版本1.2中的改动	在版本1.2中，这个方法实现了更多有关管道状态的检查，如它是否被连接，是否关闭以及从管道中读取数据的线程是否处于激活状态。在版本1.1中该方法的第一种形式是从Writer类继承来的。
参数	
buffer	包含要写入字符的非空字符数组。
ch	要写入的字符。
count	要写入字符的个数。0 <= count <= buffer.length-offset。
offset	buffer中的下标，指示取得写入字符的起始位置。0 <= offset <= buffer.length
异常	
ArrayIndexOutOfBoundsException	如果count或offset超出了指定的边界。
InterruptedIOException	如果管道满且等待管道腾出空间时当前的线程被中断时。
IOException	如果该PipedWriter已经关闭或者读数据的线程已经处于非活跃状态时（在没有关闭的情况下结束）。
NullPointerException	如果在没有连接前向该PipedWriter中写入数据。
重载	Writer.write()。
示例	见《The Java Class Libraries, Second Edition, Volume 1》中的PipedReader类。

java.io PrintStream



语法

```
public class PrintStream extends FilterOutputStream
```

描述

`PrintStream`代表了一个打印流，它是一个接受 Java 数据值并输出代表字节流的字符串的过滤流。字符串表示中的字符转化为字节，缺省的字节编码与平台相关。

要得到更多的关于该类的信息请参见《The Java Class Libraries, Second Edition, Volume 1》。

版本1.2中所作的修改

在版本1.1中废除了`PrintStream`构造函数。要版本1.2中该构造函数可以使用。

成员概述

构造函数

Δ `PrintStream()` 构造一个`PrintStream`实例。

输出方法

`print()` 向该`PrintStream`中打印数据值。

`println()` 向该`PrintStream`中打印带有直线分隔符的数据值。

`write()` 向该`PrintStream`中写入字节。

错误检查方法

`checkError()` 确定该`PrintStream`是否已经发生了异常。

`setError()` 记录该`PrintStream`发生过IO异常。

流方法

`close()` 关闭该`PrintStream`。

`flush()` 从该`PrintStream`中清除任何缓冲的输出。

参见

PrintWriter。

《The Java Class Libraries, Second Edition, Volume 1》中的PrintStream。

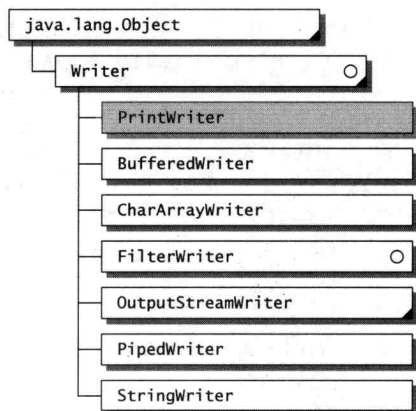
▲ PrintStream()

目的	构造一个PrintStream实例。
语法	<code>public PrintStream(OutputStream out)</code> <code>public PrintStream(OutputStream out,boolean autoFlush)</code>
描述	该构造函数创建了一个PrintStream实例，它写入基础输出流out。 如果autoFlush设置为true,则当使用println()时送往该流的输出会自动填充。如果autoFlush设置为false或没有指定就不会自动填充。
版本1.2中的改动	该方法在版本1.1中已经废除。
参数	
autoFlush	如果设为true，则写完线分隔符后该PrintStream会自动填充输出。如果设为false不执行自动填充。
out	为该打印流过滤器创建的非空输出流。
示例	

```
Socket sock = ...;  
PrintStream out = new PrintStream(sock.getOutputStream());  
out.println("hello");
```

Java.io

PrintWriter



语法

```
public class PrintWriter extends Writer
```

描述

PrintWriter类表示一个打印写入者，它是一个字符输出流，或是一个接受 Java 数据值并写出字符串表示中的字符写入者。

要得到更多的关于该类的信息请参考《The Java Class Libraries, Second Edition, Volume 1》。

版本1.2中所作的修改

PrintWriter的子类使用的out域改为受保护的(而不是私有的)。接受单参数的println()方法已被重写为不带参数的println()函数，这样子类就可以重载不带参数的方法来改变各种 println()方法的行为。

成员概述

构造函数

PrintWriter() 构造一个PrintWriter实例。

输出方法

print() 向该PrintWriter输入一个数据值。

A println() 向该PrintWriter输入一个后面跟着换行符的数据值。

write() 向该PrintWriter中写入字符。

错误检查函数

checkError 确定该PrintWriter是否已经产生了异常。

setError 记录下该PrintWriter已经发生过了IO异常。

(续)

成员概述

关闭方法

close() 关闭该PrintWriter。
flush() 清除该PrintWriter中所有输出缓冲区的内容。

受保护的域

1.2 out 容纳该PrintWriter写入的基础Writer。

参见

PrintStream。

《The Java Class Libraries, Second Edition, Volume 1》中的PrintWriter。

1.2 out

目的 容纳该PrintWriter写入的基础Writer。
语法 protected Writer out
描述 该域由PrintWriter构造函数初始化，当向基础流写入数据时所的方法都有要使用它。

A println()

目的 将后跟行终止符的数据值打印到该PrintWriter中。

语法 `public void println()`
 `public void println(boolean bool)`
 `public void println(int inum)`
 `public void println(long lnum)`
 `public void println(float fnum)`
 `public void println(double dnum)`
 `public void println(Object obj)`
 `public void println(String str)`
 `public void println(char ch)`
 `public void println(char[] charArray)`

描述 该方法的第一种形式向该PrintWriter中打印一个行终止符。下面的九种形式向该PrintWriter中打印后面带有一个行终止符的数据值。产生的输出是要打印的表示数据值的字符串。对数据值使用String.valueOf()方法得到它的字符串表示。

 使用write()方法将数据写入基础的PrintWriter。如果该PrintWriter是一个自动填充写入者，则当它被写入后就会自动填充输出。如果想让PrintWriter的子类对打印直线终止符产生影响(如产生双空格)，只需重载不带参数的println()方法。它会影响所有其他形式的println()方法。

版本1.2中的改动 所有接受参数的println()方法已被重写成使用不带参数的形式以使子类更加简单。

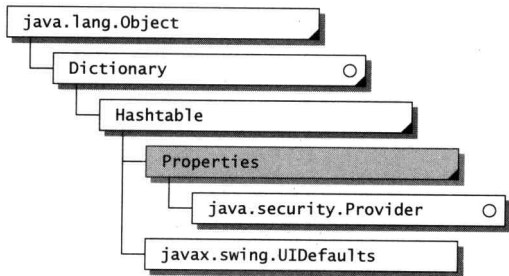
参数

bool	要打印的boolean值(true或false)。
ch	要打印的char值。
charArray	要打印的非空字符数组值。
dnum	要打印的double值。
fnum	要打印的float值。
inum	要打印的int值。
lnum	要打印的long值。
obj	要打印的对象(可接受null)。
str	要打印的字符串(可接受null)。
参见	flush()、java. long. String.valueOf()、write()。
示例	

```
import java.io.*;

class Main {
    public static void main(String[] args) {
        PrintWriter out = new DoublePrintWriter(System.out);
        out.println("hello");
        out.println("there");
        out.println("readers");
        out.flush();
    }

    static public class DoublePrintWriter extends PrintWriter {
        public DoublePrintWriter(PrintStream s) {
            super(s);
        }
        public void println() {
            super.println();
            super.println();
        }
    }
}
```



语法

```
public class Properties extends Hashtable
```

描述

Properties类用来表示一个属性列表。列表中的每一项称为一个属性，包括一个属性名和一个属性值。每一个属性名和一个属性值是一个双字节字符集的字符串。

要得到更多关于该类的更多信息请参见《The Java Class Libraries, Second Edition, Volume 1》。

版本1.2中所作的修改

有两个新的方法。第一个是setProperty(),它增加或修改Perproperties对象的属性。在版本1.1中，使用put()来增加或修改属性。但是put()可接受任意对象，它有可能提供错误的非字符串值。应使用setProperty方法来取代put()，它只接受字符串。

第二种新的方法是store()。它取代了save()并且起的作用同save()。使用save()的麻烦在于当发生一个IO异常时它不抛出异常。结果save()被废除了。

现在哈希表实现了映射表。向哈希表中增加了一些新的方法如 putAll()和keySet()(要得到更详细的信息请参见 Map。)由于属性类是从 Hashtable中继承的，所以这些新的方法可以从该类中得到。

成员概述	
构造函数	
Properties()	构造一个属性列表。
属性方法	
getProperty()	获取属性列表中的属性值。
list()	向输出流或写入者中写入该属性列表。
load()	从一个输入流中读取属性值并把它们加入到属性列表中。
propertyNames()	从该属性列表中获取该列表属性的名字。
setProperty	设置该属性列表中属性的值。

(续)

成员概述		
1.2	store()	将该属性列表写入到一个输出流。
	受保护的域	
	defaults	缺省的属性。
	废弃的方法	
△	save()	由store()取代。

参见

《The Java Class Libraries, Second Edition, Volume 1》中的Properties。

△ save()		禁止
目的	由store()取代。	
语法	public synchronized void save(OutputStream out,String comment)	
版本 1.2 中的改动	该方法在版本 1.2 中被禁止。	
描述	该方法已被 store()取代。除了当发生一个 IO 错误时 store()会抛出一个 IOException异常而save()什么也不做以外， save()和store()进行的操作是一样的。	
参数		
comment	写入该流的注释(可能为null)。	
out	写入的非空输出流。	
参见	store()。	

1.2 setProperty()	
目的	设置该属性列表中属性的值。
语法	public Object setProperty(String key, String val)
描述	该方法将属性列表中属性 key 的值设为 val。如果 key 没有出现在属性列表中，属性 key/val 就被加入到该属性列表中并且返回 null。如果 key 出现在属性列表中，则该属性的值就被设置为 val 并且返以前的值。
参数	
key	包含属性 key 的非空字符串。
val	包含属性 key 的非空字符串。
返回	返回 key 以前的值，或者，如果属性列表中没有属性 key 则返回 null。
参见	getProperty()。
示例	<pre>Properties props = new Properties(); props.setProperty("title","The Java Class Libraries"); System.out.println(props); // {title=The Java Class Libraries}</pre>

1.2 store()	
目的	将该属性列表写入输出流中。

语法	<code>public synchronized void store(OutputStream out,String comment)</code> <code>throws java.io.IOException</code>
描述	该方法将属性列表写入到输出流 <code>out</code> 中。只写入了主属性列表。如果 <code>comment</code> 字符串为非空，就会将它作为一个以前导字符（“#”）开始的注释向该流中输出第一行。在写出属性之前会向该流中写入一个带有当前时间的注释。然后按 <code>Properties</code> 类中描述的格式输出各属性。
参数	
<code>comment</code>	写入流的注释(可能为 <code>null</code>)。
<code>out</code>	要写入的非空输出流。
异常	
<code>IOException</code>	在向 <code>out</code> 写入属性列表时发生一个 IO 错误。
参见	<code>list()</code> 、 <code>load()</code> 。
示例	

```
Properties props = new Properties();
props.setProperty("title", "The Java Class Libraries");

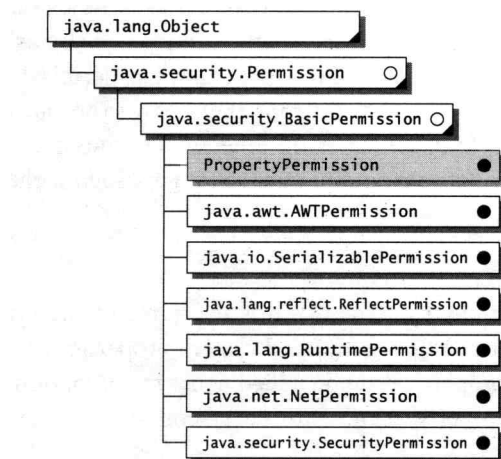
try {
    props.store(new FileOutputStream("out.properties"), "comment");
} catch (IOException e) {
    e.printStackTrace();
}
```

输出

```
#comment
#Sun Jan 10 17:47:19 PST 1999
title=The\ Java\ Class\ Libraries
```

java.util

PropertyPermission



语法

```
public final class PropertyPermission extends BasicPermission
```

描述

`PropertyPermission`类描述了如何访问一个系统属性(参见`System`)。无论什么时候要访问一个系统属性时就会由Java运行时系统在内部使用它。一个`PropertyPermission`对象由一个权限名,和一个读写操作的集合组成。前者是一个系统属性的集合(如`user.dir`),后者列出了请求的访问类型。例如一个带有权限名`"user.dir"`和一个称作`"read"`的操作的`PropertyPermission`表示了一个读系统属性`"user.dir"`的请求。安全管理员用它来检查是否允许调用者执行请求的操作。如果不允许就会抛出一个`SecurityException`异常。

在版本 1.2 中 `SecurityManager` 类中和系统属性有关的两个方法 `checkPropertiesAccess()` 和 `checkPropertyAccess()` 已经用 `PropertyPermission` 类重写过了。这个重写对于以前写和安装自己的安全管理程序的系统非常有用,它对于 `SecurityManager` 类的使用是透明的。同时该系统类有一个新的方法 `setProperty()` 更新单个属性。(以前不得不用在一个调用中更新所有的系统属性。)这个新的方法使用 `PropertyPermission` 检查调用者对试图更新的系统属性具有“write”的权力。

在 `SecurityManager` 类的描述中对版本 1.2 安全模型进行了简单地讨论。有关 Java 1.2 的安全模型和框架的更详细信息可从 <http://java.sun.com/products/jdk/1.2/docs/guide/security/index.html> 得到。

使用

applet 和应用程序一般不直接使用 `PropertyPermission` 类。它是由Java运行时系统用来增强系统安全性的。例如,当一个程序执行 `System.getProperty("user.dir")` 时,运行时系统使用 `PropertyPermission("user.dir","read")` 对象来检查调用者是否允许这样做。

应用程序可显式地创建一个 `PropertyPermission` 对象，并使用 `SecurityManager.checkPermission()` 来检查它是否具有这样的权力。但是试图访问系统属性的操作会自动触发这样的一个检查，所以很少有理由重复该操作。

PropertyPermission 的名字

`PropertyPermission` 的权限名是系统属性的名字。关于一系列预定义的系统属性名请参见 `System` 类的描述。正如在 `System` 类描述的那样，附加的系统属性可以使用 `-D` 选项加入到 Java 命令中。注意，因为系统属性名是大小写敏感的，所以 `PropertyPermission` 名也是大小写敏感的。

有一个特殊的约定关于使用单个权限名确定多个系统属性，如表 15 所示。

表15 PropertyPermission 名字语法

语 法	描 述
*	所有的系统属性
sname	sname 命名的系统属性
component.*	所有以 “ component ” 为前缀的系统属性

PropertyPermission 动作

`PropertyPermission` 对象的动作确定了对系统属要进行的操作。参见表 16。动作栏包含了当指定 `PropertyPermission` 对象的动作时要使用的关键字。空白字符 (blank, \n, \f, \t 和 \r) 可在关键字的前面或后面出现。字符的大小写是不重要的。例如，字符串 “ read, write ” 指出了代码可对权限中定义的系统属性进行读和修改动作。

表16 PropertyPermission 动作

动 作	请 求
Read	读指定的系统属性
write	修改指定的系统属性

蕴含

一个 `PropertyPermission` 对象的权限名和操作用于确定一个 `PropertyPermission` 是否蕴含另一个 `PropertyPermission`。权限 A 蕴含权限 B 指的是如果权限 A 被授权则权限 B 也得到了授权。如果 B 的动作是 A 的动作的子集且由 B 权限名确定的系统属性是 A 的子集，则 `PropertyPermission A` 蕴含 `PropertyPermission B`。

读

带有读动作的 `PropertyPermission` 表示请求读特定的系统属性。下述方法使用带这种动作的 `PropertyPermission: System.getProperty()` 和 `System.getProperties()`。

写

带有写动作的 `PropertyPermission` 表示请求修改特定的系统属性。下述方法使用带这种动作的 `PropertyPermission: System.setProperty()` 和 `System.setProperties()`。

成员概述	
构造函数	
PropertyPermission()	构造一个新的PropertyPermission实例。
动作获取方法	
getActions()	产生表示该PropertyPermission动作的正规字符串。
检查权限的方法	
implies()	检查该PropertyPermission是否蕴含另一个权限。
newPermissionCollection()	产生一个存储PropertyPermission对象的新的权限集合。
对象方法	
equals()	确定一个对象是否等于该PropertyPermission对象。
hashCode()	计算出该PropertyPermission对象的哈希码。

参见

java.lang.SecurityManager、java.lang.System.getProperties(),
java.lang.System.getProperty()、java.lang.System.setProperties(),
java.lang.System.setProperty()、java.security.Permission, Properties。

示例

该例子向我们展示了如何使用PropertyPermission实例来检查程序是否有权读取具有前缀“user”的系统属性。正如在类描述中提到的那样，一般情况下程序没有必要进行显式的检查，因为System.getProperty()会自动执行检查。这个例子只是为了达到描述的目的。

用.policy文件运行这个程序。如果省略了策略文件，会得到一个AccessControlException异常(一个SecurityException异常的子类)。

```
>java -Djava.security.policy=.policy Main
Main.java
PropertyPermission propperm = new PropertyPermission("user.*", "read");
try {
    AccessController.checkPermission(propperm);
    System.out.println(System.getProperty("user.home")); // also checks perm
} catch (SecurityException e) {
    e.printStackTrace();
}

.policy
keystore ".keystore";
grant {
    permission java.util.PropertyPermission "user.*", "read";
};
```

equals()

目的	确定一个对象是否同该PropertyPermission相等。
语法	public boolean equals(Object obj)
描述	该方法确定一个对象是否同该PropertyPermission相等。如果两个PropertyPermission对象具有相同的属性名(大小写敏感的)和动作则它们相等。使用equals()来检查权限的名字。

参数

obj 要检查的可能为null的对象。

返回 如果obj同该PropertyPermission对象具有相同的许可名和动作则返回true。

重载 java.lang.Object.equals()。

参见 hashCode()。

示例

```
PropertyPermission p0 = new PropertyPermission("*", "write,read");
PropertyPermission p1 = new PropertyPermission("*", "read, write");
PropertyPermission p2 = new PropertyPermission("os.*", "read");
PropertyPermission p3 = new PropertyPermission("user.dir", "write");
PropertyPermission p4 = new PropertyPermission("user.dir", "read");

System.out.println(p1.equals(p0)); // true (action order irrelevant)
System.out.println(p1.equals(p2)); // false
System.out.println(p2.equals(p3)); // false
System.out.println(p3.equals(p4)); // false (actions different)
System.out.println(p4.equals(new FilePermission("user.dir", "read")));
// classes different

System.out.println(p1.hashCode()); // 42
System.out.println(p2.hashCode()); // 3418784
System.out.println(p3.hashCode()); // -267617302
```

getActions()

目的 产生表示该PropertyPermission的动作的正规字符串。

语法 public String getActions()

描述 该方法产生表示该PropertyPermission的动作的正规字符串。其结果可能不同于传递到构造函数的字符串，但是它表示的操作和传递到构造函数中的操作是一样的。PropertyPermission的动作主关键字按以下述顺序列出(小写)：read,write。没有列出不是PropertyPermission部分的动作。每一个主关键字用一个逗号隔开，字符串中没有空格。

返回 表示该PropertyPermission动作的非空正规字符串。

重载 java.security.Permission.getActions()。

示例

```
System.out.println(
    new PropertyPermission("file.separator", " write , read").getActions());
// read,write
System.out.println(new PropertyPermission("user.dir", " write").getActions());
// write
```

hashCode()

目的 计算出该PropertyPermission的哈希码。

语法 public int hashCode()

描述 该方法计算出该PropertyPermission的哈希码。PropertyPermission对象的哈希码是使用它的权限名计算出来的。相等的两个 PropertyPermission (由equals()判断得来)将具有相同的哈希码。然而两个不同的PropertyPermission也可能具有相同的哈希码。哈希码一般用作哈希表中的关键字。

返回	表示该对象哈希码的整数。
重载	java.lang.Object.hashCode()。
参见	equals()、java.util.HashMap()、java.util.Hashtable()。
示例	见 equals()。

implies()

目的	检查该PropertyPermission是否蕴含另一个权限。
语法	public boolean implies(Permission perm)
描述	该方法检查该PropertyPermission是否蕴含另一个权限perm。当且仅当对由perm确定的系统属性动作，PropertyPermission同时也允许进行该动作时，该PropertyPermission蕴含了perm。也就是说，perm的操作必需是该PropertyPermission动作的子集并且由perm确定的系统属性必需是由PropertyPermission确定的属性的子集。如果perm为null或不是PropertyPermission的实例，该方法返回false。

参数

perm	要检查的权限(可能为null)。
返回	如果该对象隐含perm，该方法返回true，否则返回false。
重载	java.lang.Permission.implies()。

示例

```
PropertyPermission p1 = new PropertyPermission("*", "read, write");
PropertyPermission p2 = new PropertyPermission("*", "read");
PropertyPermission p3 = new PropertyPermission("java.*", "read");
PropertyPermission p4 = new PropertyPermission("java.home", "read");
PropertyPermission p5 = new PropertyPermission("foo.bar", "write");

System.out.println(p1.implies(p2));           // true
System.out.println(p1.implies(p3));           // true
System.out.println(p4.implies(p3));           // false; diff name
System.out.println(p3.implies(p4));           // true
System.out.println(p1.implies(p5));           // true
System.out.println(p2.implies(p5));           // false; diff action
System.out.println(p4.implies(p5));           // false; diff name, action

System.out.println(
    p1.implies(new SocketPermission("java.sun.com:80", "connect")));
System.out.println(p1.implies(null));          // false; diff Permission
// false; null Permission
```

PropertyPermission()

目的	构造一个新的PropertyPermission实例。
语法	public PropertyPermission(String pname,String actions)
描述	该构造函数构造一个新的PropertyPermission。该权限的名字是pname,它使用类描述中规定的语法。动作包含一系列由一个或多个逗号分隔的关键字：read或是write。它指定可对由pname命名的对象能实现的动作，正如类描述中规定的那样。

参数

actions	一个指定新的PropertyPermission动作的非空字符串。
pname	确定该PropertyPermission系统属性或系统属性组的非空字符串。

示例 见equals()。

newPermissionCollection()

目的	为存储PropertyPermission对象创建一个新的PermissionCollection。
语法	public PermissionCollection newPermissionCollection()
描述	该方法为存储PropertyPermission对象创建一个新的PermissionCollection。对于单个PropertyPermission可以查看(通过implies()方法)它是否隐含另一个权限。对于PermissionCollection可以查看该权限集合是否隐含另外的权限。例如 java运行时系统在试图确定调用者是否有权执行在安全策略中授予调用者以PropertyPermission为基础的动作时进行这样的一个测试。
返回	一个适合存储PropertyPermission的新的非空PropertyPermission对象。
重载	java.security.Permission.newPermissionCollection()。
参见	implies(), java.security.PermissionCollection。
示例	

```
PropertyPermission[] p = new PropertyPermission[3];

// Allow entire java. hierarchy to be read
p[0] = new PropertyPermission("java.*", "read");
// Allow update to java.vendor.* hierarchy
p[1] = new PropertyPermission("java.vendor.*", "write");

// Allow read/update to os.version
p[2] = new PropertyPermission("os.version", "read, write");

PermissionCollection col = p[0].newPermissionCollection();
for (int i = 0; i < p.length; i++) {
    col.add(p[i]);
}

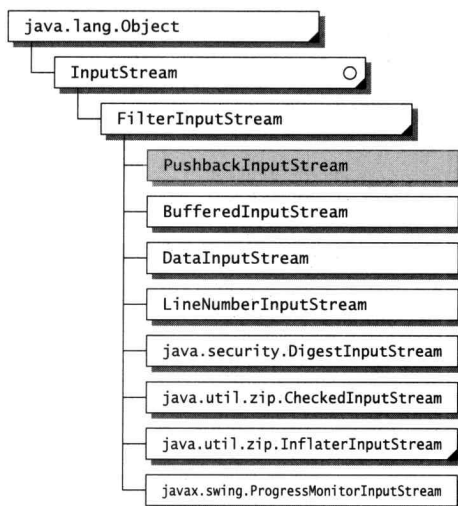
// OK to read/write java.vendor.url?
System.out.println(col.implies(
    new PropertyPermission("java.vendor.url", "read, write")));
// true

// OK to read user.dir?
System.out.println(col.implies(
    new PropertyPermission("user.dir", "read"))); // false

// OK to write java.vendor ?
System.out.println(col.implies(
    new PropertyPermission("java.vendor", "write")));
// false; java.vendor.* doesn't include java.vendor
```

java.io

PushbackInputStream



语法

```
public class PushbackInputStream extends FilterInputStream
```

描述

该PushbackInputStream类表代了一个允许字节“推回”(或是从流中反读取)到流的过滤流。例如，可以读取一些字节，然后将它们推回到流中，这样下一个读操作将取得被推回的字节。当创建一个解析程序需要从流中提前读取数据以便确定如何处理该流时，这种能力相当有用。

要得到更多的关于该类的信息请参见《The Java Class Libraries, Second Edition, Volume 1》。

版本1.2中所作的修改

close()和skip()方法重载了FilterInputStream中对应的部分。

成员概述

构造函数

PushbackInputStream() 为输入流创建一个PushbackInputStream实例。

输入方法

read() 从PushbackInputStream中读取字节。

 skip() 从PushbackInputStream中跳过字节。

unread() 推回一个或多个字节。

流方法

available() 确定可以无阻塞地读取的字符个数。

 close() 关闭该PushbackInputStream。

(续)

成员概述

markSupported()	确定该PushbackInputStream是否支持标记/重置。
受保护的域	
buf	包含反读字节的缓冲区。
pos	从buf中读取的下一个回推字节的位置。

参见

《The Java Class Libraries, Second Edition, Volume 1》中的PushbackInputStream。

△ close()

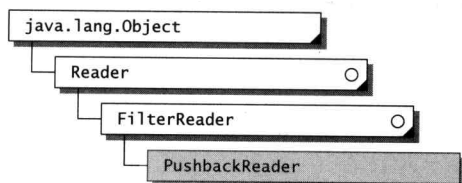
目的	关闭该PushbackInputStream。
语法	public synchronized void close() throws IOException
描述	该方法关闭基础输入流并释放由该输入流所使用的缓冲区。
版本1.2中的改动	在版本1.1中该方法从FilterInputStream中继承，它只简单地关闭基础输入流。
异常	
IOException	如果基础输入流正被关闭而同时发生一个IO错误。
重载	FilterInputStream.close()。
示例	见《The Java Class Libraries, Second Edition, Volume 1》中的PushbackInputStream.available()。

skip()

目的	从PushBackInputStream中跳过字节。
语法	public long skip(long count) throws IOException
描述	此方法从PushBackInputStream中跳过count个字节。后续read()调用不返回跳过的字节(除非使用了标记/重置)。此方法首先跳过被推回的字节，然后调用基础输入流的skip()跳过余下的字节。 如果 count<=0，没有跳过任何字节。
版本1.2中的改动	在版本1.1中，此方法从FilterInputStream继承而来，它仅简单调用基础输入流的skip()跳过字节。尽管它的效果与版本1.2没有什么不同，但版本1.2更有效。
参数	
count	将要跳过的字节数。
返回	跳过的实际字节数。
异常	
IOException	当跳过字节时出现IO错误。
重载	FilterInputStream.skip()。
示例	见《The Java Class Libraries, Second Edition, Volume 1》中的InputStream.skip()。

java.io

PushbackReader



语法

```
public class PushbackReader extends FilterReader
```

描述

PushbackReader类代表一个过滤阅读器(过滤字符流), 在过滤阅读器中允许字符被“推回”进流中。例如, 可以从流中读取一些字符然后按顺序将它们推回到流中去, 使下一个读取操作可以获取这些被推回的字符。当建立语法分析器时, 为了决定如何处理输入, 需要首先进行读取, 这时这种推回能力是有用的。

有关这个类的详细内容请参见 《The Java Class Libraries, Second Edition, Volume 1》。

版本1.2中所作的修改

为了正确反映 PushbackReader类不支持标记/重置, 在版本 1.2中对FilterReader类中的 mark()和reset()方法进行了重载。

成员概述

构造函数

PushbackReader() 创建一个PushbackReader类的实例。

输入方法

read() 从这个PushbackReader对象中读取一个或多个字符。

unread() 推回一个或多个字符。

流方法

close() 关闭这个PushbackReader对象。

ready() 判断这个PushbackReader对象是否为无阻塞读取做好了准备。

标记/复位方法

⚠ mark() 该方法不应该被调用。

markSupported() 返回false。

⚠ reset() 该方法不应该被调用。

参见

《The Java Class Libraries, Second Edition, Volume 1》中的PushbackReader类。

▲ mark()

目的	该方法不应该被调用。
语法	<code>public void mark(int readAheadLimit) throws IOException</code>
描述	PushbackReader不支持标记/重复。
版本1.2中的改动	在版本1.1中，这个方法在流上调用 <code>mark()</code> 方法。在版本1.2中这样做是不正确的，因为PushbackReader不支持标记/重置。
参数	
<code>readAheadLimit</code>	忽略。
异常	
IOException	因为PushbackReader不支持标记/重置，所以这个异常经常被抛出。
重载	<code>FilterReader.mark()</code> 。
参见	<code>markSupported()</code> 、 <code>reset()</code> 。

▲ reset()

目的	该方法不应该被调用。
语法	<code>public void reset() throws IOException</code>
描述	PushbackReader不支持标记/重置。
版本1.2中的改动	在版本1.1中，这个方法在基础流上调用 <code>reset()</code> 方法。在版本1.2中这样做是不正确的，因为PushbackReader不支持标记/重置。
异常	
IOException	因为PushbackReader不支持标记/重置，所以这个异常经常被抛出。
重载	<code>FilterReader.reset()</code> 。
参见	<code>markSupported()</code> 、 <code>mark()</code> 。

java.util

Random

语法

```
public class Random implements Serializable
```

描述

Random类代表一个产生一系列伪随机数的伪随机数产生器。可以为这个类提供一个“种子”(seed)来创建一个伪随机数产生器。伪随机数产生器使用这粒“种子”根据一种算法生成伪随机数。

有关这个类的详细内容请参见《The Java Class Libraries , Second Edition , Volume 1》。

版本1.2中所作的修改

在版本1.2中新增加了nextBoolean()方法和重载版本的nextInt()。

成员概述		
构造函数		
	Random()	构造一个伪随机数产生器。
种子方法		
	setSeed()	为这个伪随机数产生器设置一粒种子。
生成方法		
	nextBytes()	生成一个按均匀分布的字节序列中的下一个伪随机数。
1.2	nextBoolean()	生成下一个按均匀分布的 boolean值伪随机数。
	nextDouble()	生成下一个按均匀分布的 double值伪随机数。
	nextFloat()	生成下一个按均匀分布的 float值伪随机数。
	nextGaussian()	生成下一个高斯分布的 double值伪随机数。
Δ	nextInt()	生成下一个按均匀分布的 int值伪随机数。
	nextLong()	生成下一个按均匀分布的 long值伪随机数。
受保护的方法		
	next()	生成一个序列中的下一个伪随机数。
	参见	《The Java Class Libraries , Second Edition , Volume 1》中的 Random类。

1.2 nextBoolean()

目的	生成下一个按均匀分布的boolean值伪随机数。
语法	public int nextBoolean()
返回	true或false。
示例	

```
Random r = new Random() ;
System.out.println(r.nextBoolean()) ; //true
System.out.println(r.nextBoolean()) ; //true
System.out.println(r.nextBoolean()) ; //false
```

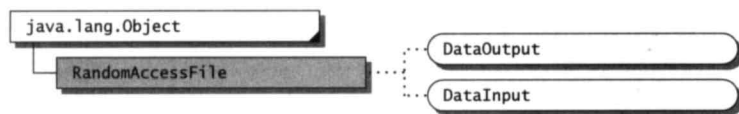
▲ nextInt()

目的	生成下一个按均匀分布的int值伪随机数。
语法	public int nextInt()
从版本1.2	public int nextInt(int n)
描述	该方法返回一个按均匀分布的 int值(或正或负)。如果说明了参数n，则返回的按均匀分布的int值的范围将是[0..n - 1]。
版本1.2中的改动	在版本1.2中增加了一个接受int值作为参数的新形式。
参数	
n	一个限制返回值的范围的正数，返回值被限定在 [0..n - 1]。
返回	一个int型随机值。
参见	java.lang.Integer.MAX_VALUE，java.lang.Integer.MIN_VALUE，next()，nextLong()。
示例	

```
Random r = new Random();
System.out.println( r.nextInt() );           // -1434938125
System.out.println( r.nextInt() );           // 1792700005
System.out.println( r.nextInt(10) );         // 3
System.out.println( r.nextInt(10) );         // 8
// r.nextInt(-10);                           // IllegalArgumentException
```


java.io

RandomAccessFile



语法

```
public class RandomAccessFile implements DataOutput , DataInput
```

描述

RandomAccessFile类代表一个随机访问文件。在随机访问文件中可以指向文件的任何一个位置，并从那个位置开始对文件进行IO操作。该类提供了设置当前文件指针 (IO操作将从此位置开始进行)的方法。还定义了DataOutput和DataInput接口，通过这两个接口可以将数值从文件中读出或写入文件。

有关这个类的详细内容请参见《The Java Class Libraries , Second Edition , Volume 1》。

版本1.2中所作的修改

该版本中增加了setLength()方法，该方法令文件可以被截断或扩展。

接受字符串名的RandomAccessFile构造函数，现在改为抛出FileNotFoundException，而不是更普通的IOException。

成员概述

构造函数

A RandomAccessFile() 创建一个RandomAccessFile类的实例。

随机访问方法

getFilePointer() 获取这个RandomAccessFile对象的当前文件指针。

seek() 设置这个RandomAccessFile对象的文件指针。

1.2 setLength() 设置这个RandomAccessFile对象的长度。

输入方法

read() 从这个RandomAccessFile对象中读取字符。

readBoolean() 从这个RandomAccessFile对象中读取一个boolean值。

readByte() 从这个RandomAccessFile对象中读取一个8位的byte值。

readChar() 从这个RandomAccessFile对象中读取一个16位的char值。

readDouble() 从这个RandomAccessFile对象中读取一个64位的double值。

readFloat() 从这个RandomAccessFile对象中读取一个32位的float值。

readFully() 按所请求的字节数从这个RandomAccessFile对象中读取相应数量的字节，当所有的字节都被读完时停止。

(续)

成员概述	
readInt()	从这个RandomAccessFile对象中读取一个32位的int值。
readLine()	从这个RandomAccessFile对象中读取一行。
readLong()	从这个RandomAccessFile对象中读取一个64位的long值。
readShort()	从这个RandomAccessFile对象中读取一个16位的short值。
readUnsignedByte()	从这个RandomAccessFile对象中读取一个8位的无符号byte值。
readUnsignedShort()	从这个RandomAccessFile对象中读取一个16位的无符号short值。
readUTF()	从这个RandomAccessFile对象中读取一个统一码字符串。
skipBytes()	跳过这个RandomAccessFile对象中一定数量的字节。
输出方法	
write()	向这个RandomAccessFile对象中写入字符。
writeBoolean()	向这个RandomAccessFile对象中写入一个boolean值。
writeByte()	向这个RandomAccessFile对象中写入一个8位的byte值。
writeBytes()	以一个字节序列的形式向这个RandomAccessFile对象中写入一个字符串。
writeChar()	向这个RandomAccessFile对象中写入一个16位的char。
writeChars()	以一个16位char值序列的形式向这个RandomAccessFile对象中写入一个字符串。
writeDouble()	向这个RandomAccessFile对象中写入一个64位的double值。
writeFloat()	向这个RandomAccessFile对象中写入一个32位的float值。
writeInt()	向这个RandomAccessFile对象中写入一个32位的int值。
writeLong()	向这个RandomAccessFile对象中写入一个64位的long值。
writeShort()	向这个RandomAccessFile对象中写入一个16位的short值。
writeUTF()	向这个RandomAccessFile对象中写入一个统一码字符串。
获取对象信息的方法	
length()	判断这个RandomAccessFile对象中字节的数量。
getFD()	获取这个RandomAccessFile对象的文件描述符。
关闭方法	
close()	关闭这个RandomAccessFile对象。

参见

《The Java Class Libraries , Second Edition , Volume 1》中的RandomAccessFile类。

▲ RandomAccessFile()

目的	创建一个RandomAccessFile类的实例。
语法	<pre>public RandomAccessFile(File file , String mode)throws IOException public RandomAccessFile(String fileName , String mode)throws</pre>

描述	FileNotFoundException
	RandomAccessFile类有两种构造函数的形式。构造函数的第一种形式使用File类的一个实例file的文件创建一个RandomAccessFile类的实例。构造函数的第二种形式为文件路径名为filename文件创建一个RandomAccessFile类的实例。当RandomAccessFile实例被创建，文件便以字符串mode中所说明的模式打开文件。mode可以是“r”，这就意味着文件以只读方式打开，或“rw”，这就意味着文件以读写方式被打开。如果mode为其他值将会导致一个IllegalArgumentException异常。文件只有在获得安全管理员许可的情况下才能被打开。
版本1.2的改动	在版本1.1中如果构造函数接受了一个作为文件名的字符串，将会抛出一个IOException异常。
参数	
file	一个要打开文件的非空File对象。
fileName	一个作为要打开文件的名字的非空字符串。
mode	一个非空的访问模式(“r”或“rw”)。
异常	
FileNotFoundException	如果fileName是一个目录而不是一个文件或文件没有找到。
IllegalArgumentException	如果mode不是“r”或“rw”。
IOException	在试图打开所说明的文件时发生IO错误。
SecurityException	由于安全原因文件不能按所说明的模式打开。
参见	java.lang.SecurityManager, FilePermission。
示例	见《The Java Class Libraries, Second Edition, Volume 1》中RandomAccessFile类示例。

1.2 setLength()

目的	设置这个RandomAccessFile对象的长度。
语法	public native void setLength(long newLength) throws IOException
描述	该方法把这个RandomAccessFile对象的长度设置为newLength。如果this.length()>newLength(), 该方法将对这个文件进行截取以使它包含newLength个字节。如果getFilePointer()>newLength, 这个文件指针将被设置为newLength。(文件指针是一个位置, 从这个位置开始进行下一个文件读/写操作。) 如果this.length()<newLength(), 该方法将对文件进行扩展以使文件长度为newLength个字节。文件的扩展部分的内容不确定。
参数	
newLength	希望文件所要达到的长度(字节)。
异常	
IOException	在设置长度时发生IO错误。

参见

length()。

示例

在这个示例中按指定的长度创建了一个文件。

```
import java.io.*;

class Main {
    public static void main(String[] args) {
        if (args.length != 2) {
            System.err.println(
                "usage: java Main <new filename> <size in bytes>");
            System.exit(-1);
        }
        long size = 0;
        try {
            size = Long.parseLong(args[1]);
        } catch (NumberFormatException e) {
            System.err.println(
                "usage: java Main <new filename> <size in bytes>");
            System.exit(-1);
        }

        try {
            RandomAccessFile raf = new RandomAccessFile(args[0], "rw");
            raf.setLength(size);    // extend file to be specified size
            raf.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```