Java语言程序设计（第2版）

第5章 异常处理与输入/输出流

郑莉

第5章导学

目录

- 异常处理简介
- 输入/输出流的概念
- 文件读写

异常处理的概念

<5.1.1>~<5.1.2>

什么是异常处理?

- 在程序运行过程中有可能发生异常事件，这些事件的发生将阻止程序的正常运行
- 如何异常事件引起的错误？把错误交给谁去处理？程序又该如何从错误中恢复？
- Java语言具有运行错误处理机制。

异常的基本概念

- 又称为例外，是特殊的运行错误对象
- Java中声明了很多异常类，每个异常类都代表了一种运行错误，类中包含了
  - 该运行错误的信息
  - 处理错误的方法
- 每当Java程序运行过程中发生一个可识别的运行错误时，即该错误有一个异常类与之相对应时，系统都会产生一个相应的该异常类的对象，即产生一个异常。

java处理异常的方法

- 抛出(throw)异常
- 捕获(catch)异常

java处理异常的方法

- 抛出(throw)异常
  - 在方法的运行过程中，如果发生了异常，则该方法生成一个代表该异常的对象并把它交给运行时系统，运行时系统便寻找相应的代码来处理这一异常。
- 捕获(catch)异常
  - 运行时系统在方法的调用栈中查找，从生成异常的方法开始进行回溯，直到找到包含相应异常处理的方法为止。

Java异常处理机制的优点

Java异常处理机制的优点

- 将错误处理代码从常规代码中分离出来；
- 按错误类型和差别分组；
- 对无法预测的错误的捕获和处理；
- 克服了传统方法的错误信息有限的问题；
- 把错误传播给调用堆栈。

错误的分类

- 根据错误的严重程度不同，可分为两类
  - 错误
    - 致命性的，程序无法处理；
    - Error类是所有错误类的父类。
  - 异常
    - 非致命性的，可编制程序捕获和处理；
    - Exception类是所有异常类的父类。

异常的分类

- 非检查型异常
  - 不期望程序捕获的异常，在方法中不需要声明，编译器也不进行检查。
  - 继承自RuntimeException。
  - 不要求捕获和声明的原因：
    - 引发RuntimeException的操作在Java应用程序中会频繁出现。例如，若每次使用对象时，都必须编写异常处理代码来检查null引用，则整个应用程序很快将变成一个庞大的try-catch块。
    - 它们表示的问题不一定作为异常处理。如：可以在除法运算时检查0值，而不使用ArithmeticException。可以在使用引用前测试空值。
- 检查型异常
  - 其他类型的异常。
  - 如果被调用的方法抛出一个类型为E的检查型异常，那么调用者必须捕获E或者也声明抛出E（或者E的一个父类），对此编译器要进行检查。

预定义的一些常见异常

- Java预定义的一些常见异常
  - ArithmeticException
    - 整数除法中除数为0
  - NullPointerException
    - 访问的对象还没有实例化
  - NegativeArraySizeException
    - 创建数组时元素个数是负数
  - ArrayIndexOutOfBoundsException
    - 访问数组元素时，数组下标越界
  - ArrayStoreException
    - 程序试图向数组中存取错误类型的数据
  - FileNotFoundException
    - 试图存取一个并不存在的文件
  - IOException
    - 通常的I/O错误

非检查型异常

检查型异常

例：非检查型异常——数组越界异常

例：非检查型异常——数组越界异常

```
import java.io.*;
public class HelloWorld {
    public static void main (String args[ ]) {
        int i = 0;
        String greetings [ ] = {"Hello world!",  "No, I mean it!",
                                "HELLO WORLD!!"};
        while (i < 4) {
            System.out.println (greetings[i]);
            i++;
        }
    }
}
```

例：非检查型异常——数组越界异常

- 运行结果

```
Hello world!
No, I mean it!
HELLO WORLD!!
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException
        at HelloWorld.main(HelloWorld.java:7)
```

- 说明
  - 访问数组下标越界，导致ArrayIndexOutOfBoundsException异常。
  - 该异常是系统定义好的类，对应系统可识别的错误，所以Java虚拟机会自动中止程序的执行流程，并新建一个该异常类的对象，即抛出数组出界异常。

结束语

- 这一节介绍了异常的基本概念和一个简单的例子。

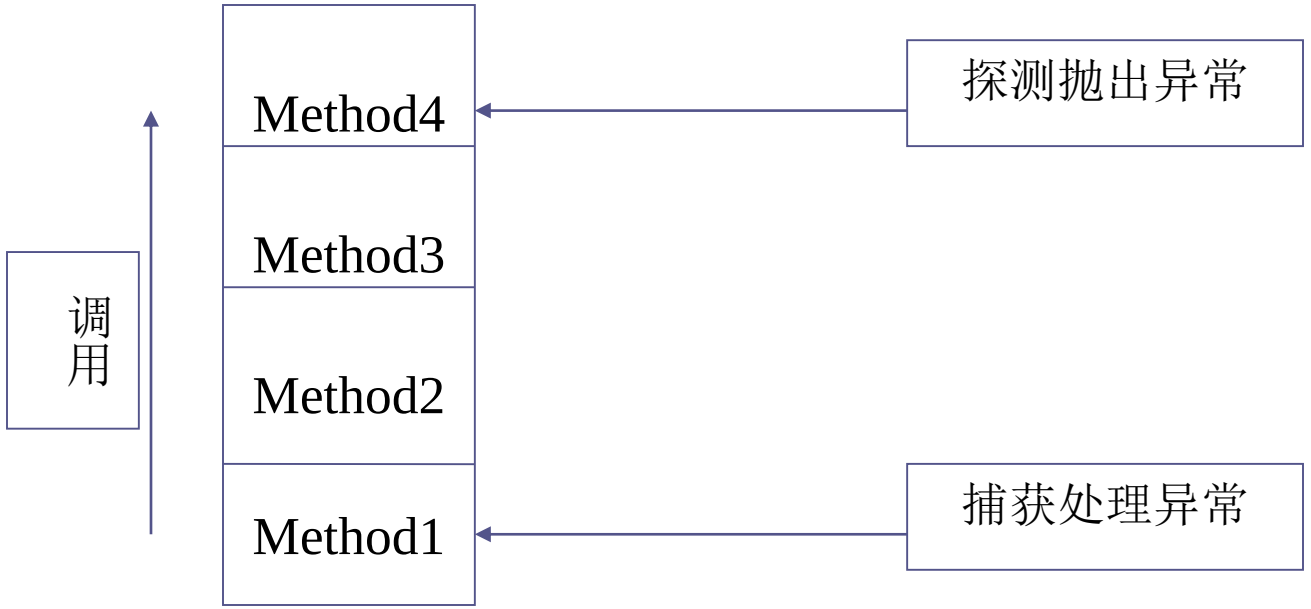
异常的处理

<5.1.3>~<5.1.5>

检查型异常的处理

- 声明抛出异常
  - 不在当前方法内处理异常，可以使用throws子句声明将异常抛出到调用方法中；
  - 如果所有的方法都选择了抛出此异常，最后 JVM将捕获它，输出相关的错误信息，并终止程序的运行。
- 捕获异常
  - 使用try{}catch(){}块，捕获到所发生的异常，并进行相应的处理

异常处理示意图



抛出异常的例子

现在看一个不捕获、直接抛出异常的例子

抛出异常的例子

```
public void openThisFile(String fileName)
     throws java.io.FileNotFoundException {
          //code  for method
 }
public void getCustomerInfo()
    throws java.io.FileNotFoundException {
          // do something
       this.openThisFile("customer.txt");
          // do something
     }
```

- 如果在openThisFile中抛出了FileNotfoundException异常，getCustomerInfo将停止执行，并将此异常传送给它的调用者

捕获异常的语法

```
try {
    statement(s)
} catch (exceptiontype name) {
    statement(s)
} finally {
    statement(s)
}
```

捕获异常的语法

- 说明
  - try 语句
    - 其后跟随可能产生异常的代码块。
  - catch语句
    - 其后跟随异常处理语句，通常都要用到两个方法：
      - getMessage() – 返回一个字符串，对发生的异常进行描述。
      - printStackTrace() – 给出方法的调用序列，一直到异常的产生位置 。
  - finally语句
    - 不论在try代码段是否产生异常，finally 后的程序代码段都会被执行。通常在这里释放内存以外的其他资源。
- 注意事项
  - 如果并列有多个catch语句捕获多个异常，则一般的异常类型放在后面，特殊的放在前面。

生成异常对象

- 异常对象由谁生成呢？

生成异常对象

- 三种方式
  - 由Java虚拟机生成；
  - 由Java类库中的某些类生成；
  - 在自己写的程序中生成和抛出异常对象。
- 抛出异常对象都是通过throw语句实现，异常对象必须是Throwable或其子类的实例：
  - throw new ThrowableObject();
  - ArithmeticException e = new ArithmeticException();
    throw e;

例：生成异常对象

- 下面看一个生成和抛出异常对象的例子。

例：生成异常对象

```
class ThrowTest
{
    public static void main(String args[])
    {
        try {  throw new ArithmeticException();
        } catch(ArithmeticException ae){
            System.out.println(ae);
        }
        try {  throw new ArrayIndexOutOfBoundsException();
        } catch(ArrayIndexOutOfBoundsException ai){
            System.out.println(ai);
        }
        try {  throw new StringIndexOutOfBoundsException();
        } catch(StringIndexOutOfBoundsException si){
            System.out.println(si);
        }
    }
}
```

例：生成异常对象

- 运行结果

  java.lang.ArithmeticException

  java.lang.ArrayIndexOutOfBoundsException

  java.lang.StringIndexOutOfBoundsException

声明自己的异常类

- 除使用系统预定义的异常类外，用户还可声明自己的异常类

声明自己的异常类

- 自定义的所有异常类都必须是Exception的子类
- 声明语法如下

```
public class MyExceptionName extends SuperclassOfMyException {
    public MyExceptionName() {
        super("Some string explaining the exception");
    }
}
```

结束语

- 这一节介绍了异常的处理机制: try catch finally语句，声明异常类、构造和抛出异常对象

输入/输出流的概念

<5.2>

I/O (Input/Output)流

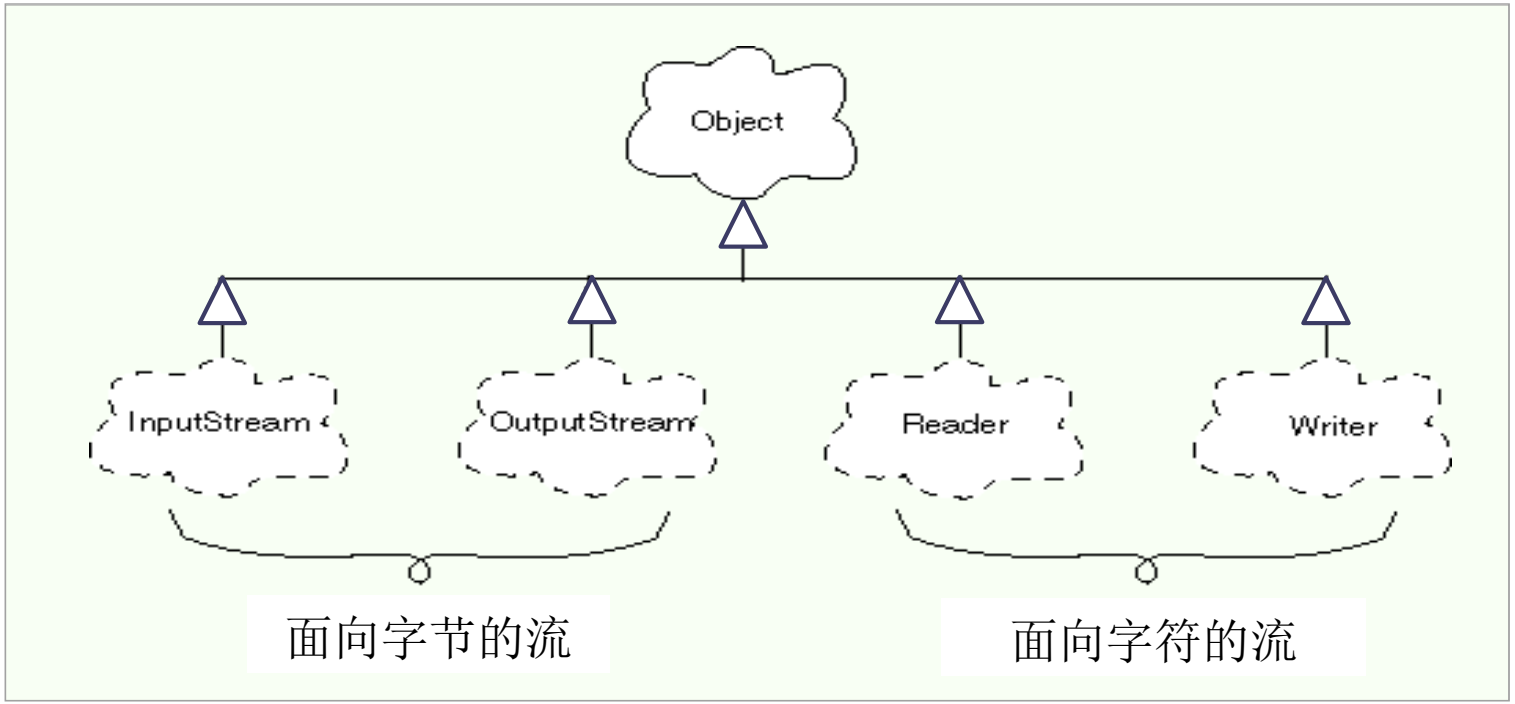
- 在Java中将信息的输入与输出过程抽象为I/O流
  - 输入是指数据流入程序
  - 输出是指数据从程序流出
- 一个流就是一个从源流向目的地的数据序列

预定义的I/O流类

- 从流的方向划分
  - 输入流
  - 输出流
- 从流的分工划分
  - 节点流
  - 处理流
- 从流的内容划分
  - 面向字符的流
  - 面向字节的流

java.io包的顶级层次结构

- 面向字符的流：专门用于字符数据
- 面向字节的流：用于一般目的



面向字符的流

- 针对字符数据的特点进行过优化，提供一些面向字符的有用特性。

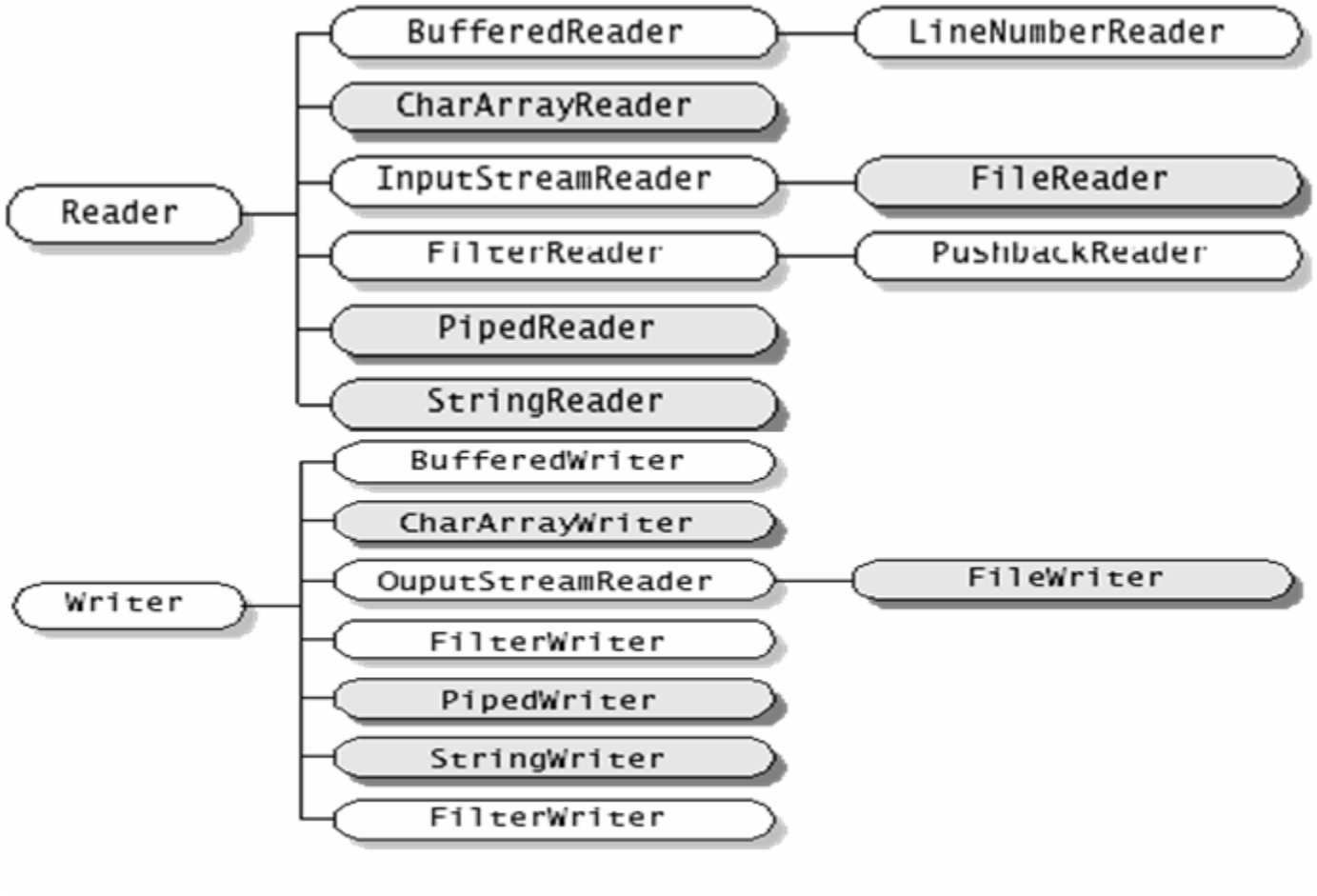
面向字符的流

- 源或目标通常是文本文件；
- 实现内部格式和文本文件中的外部格式之间转换
  - 内部格式：16-bit char 数据类型
  - 外部格式：
    - UTF(Universal character set Transformation Format)：很多人称之为"Universal Text Format"
    - 包括ASCII 码及非ASCII 码字符，比如： 斯拉夫(Cyrillic)字符, 希腊字符, 亚洲字符等

面向字符的抽象流类——Reader和Writer

- java.io包中所有字符流的抽象超类。
- Reader提供了输入字符的API。
- Writer提供了输出字符的API。
- 它们的子类又可分为两大类
  - 节点流：从数据源读入数据或往目的地写出数据；
  - 处理流：对数据执行某种处理。
- 多数程序使用这两个抽象类的一系列子类来读入/写出文本信息
  - 例如FileReader/FileWriter用来读/写文本文件。

面向字符的流



阴影部分为节点流，其他为处理流

面向字节的流

- 数据源或目标中含有非字符数据，必须用字节流来输入/输出
- 通常被用来读写诸如图片、声音之类的二进制数据
- 绝大多数数据是被存储为二进制文件的，通常二进制文件要比含有相同数据量的文本文件小得多

面向字节的抽象流类——InputStream和OutputStream

- 是用来处理字节流的抽象基类，程序使用这两个类的子类来读写字节信息
- 分为两部分
  - 节点流
  - 处理流

标准输入输出流对象

- System类的静态成员变量
- 包括
  - System.in： InputStream类型的，代表标准输入流，默认状态对应于键盘输入。
  - System.out：PrintStream类型的，代表标准输出流，默认状态对应于显示器输出。
  - System.err：PrintStream类型的，代表标准错误信息输出流，默认状态对应于显示器输出。

按类型输入/输出数据

- 如果需要按类型控制输出信息的格式，和从二进制文件中读取有类型的数据，Java有很方便的方式。

按类型输入/输出数据

- printf方法

  System.out.printf("%-12s is %2d long", name, l);

  System.out.printf("value = %2.2F", value);

  - %n 是平台无关的换行标志。
- Scanner

  Scanner s = new Scanner(System.in);

  int n = s.nextInt();

  - 还有下列方法：nextByte(),nextDouble(),nextFloat,nextInt(), nextLine(),nextLong(),nextShort()。

标准输入/输出重新定向

- setIn(InputStream)： 设置标准输入流
- setOut(PrintStream)： 设置标准输出流
- setErr(PrintStream)： 设置标准错误输出流

例：标准输入/输出重定向（复制文件）

```
import java.io.*;
public class Redirecting {
public static void main(String[] args)  throws IOException {
    BufferedInputStream in = new BufferedInputStream(
                    new FileInputStream( "Redirecting.java"));
    PrintStream out = new PrintStream( new
          BufferedOutputStream( new FileOutputStream("test.out")));
    System.setIn(in);    System.setOut(out);    System.setErr(out);
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    String s;
    while((s = br.readLine()) != null)   System.out.println(s);
    in.close();
    out.close();
  }
}
```

结束语

- 这一节介绍了输入/输出流的概念，面向字符的流、面向字节的流，和预定义的流对象。

写文本文件

<5.3.1>

本节知识点：FileWriter类 、创建一个磁盘文件 、关闭一个磁盘文件 、write() 方法、捕获I/O异常 、BufferedWriter 类

I/O异常

- 多数IO方法在遇到错误时会抛出异常，因此调用这些方法时必须
  - 在方法头声明抛出IOException异常；
  - 或者在try块中执行IO，然后在catch块中捕获IOException异常。

例：创建文件并写入若干行文本

- 下面我们通过例题看一下，怎么样通过FileWriter流创建一个文本文件，并写入若干行文本。

- 例题中直接抛出了I/O异常（本行文字不显示）。

例：创建文件并写入若干行文本

- 在C盘根目录创建文本文件Hello.txt，并往里写入若干行文本

```
import java.io.*;
class FileWriterTester {
  public static void main ( String[] args ) throws IOException {
    //main方法中声明抛出IO异常
    String fileName = "C:\\Hello.txt";
    FileWriter writer = new FileWriter( fileName );
    writer.write( "Hello!\n");
    writer.write( "This is my first text file,\n"  );
    writer.write( "You can see how this is done.\n" );
    writer.write("输入一行中文也可以\n");
    writer.close();
  }
}
```

例：创建文件并写入若干行文本

- 打开C盘根目录下的Hello.txt文件：

换行有些问题，例6-4中将解决这个问题。

每次运行都会删除旧文件生成新文件。

- 下面的例题演示了捕获和处理I/O异常

例：写入文本文件，处理IO异常

```
import java.io.*;
class FileWriterTester {
  public static void main ( String[] args ) {
    String fileName = "c:\\Hello.txt" ;
    try {  //将所有IO操作放入try块中
              FileWriter writer = new FileWriter( fileName ,true );
              writer.write( "Hello!\n");
              writer.write( "This is my first text file,\n"   );
              writer.write( "You can see how this is done. \n" );
              writer.write("输入一行中文也可以\n");
              writer.close();
    }
    catch ( IOException iox) {  System.out.println("Problem writing" + fileName ); }
    }
}
```

例：写入文本文件，处理IO异常

- 说明：
  - 运行此程序，会发现在原文件内容后面又追加了重复的内容，这就是将构造方法的第二个参数设为true的效果。
  - 如果将文件属性改为只读属性，再运行本程序，就会出现IO错误，程序将转入catch块中，给出出错信息。

BufferedWriter类

- 如果需要写入的内容很多，就应该使用更为高效的缓冲器流类BufferedWriter。

BufferedWriter类

- FileWriter和BufferedWriter类都用于输出字符流，包含的方法几乎完全一样，但BufferedWriter多提供了一个newLine()方法用于换行。
  - 不同的系统对文字的换行方法不同。newLine()方法可以输出在当前计算机上正确的换行符。

例：写入文本文件，使用BufferedWriter

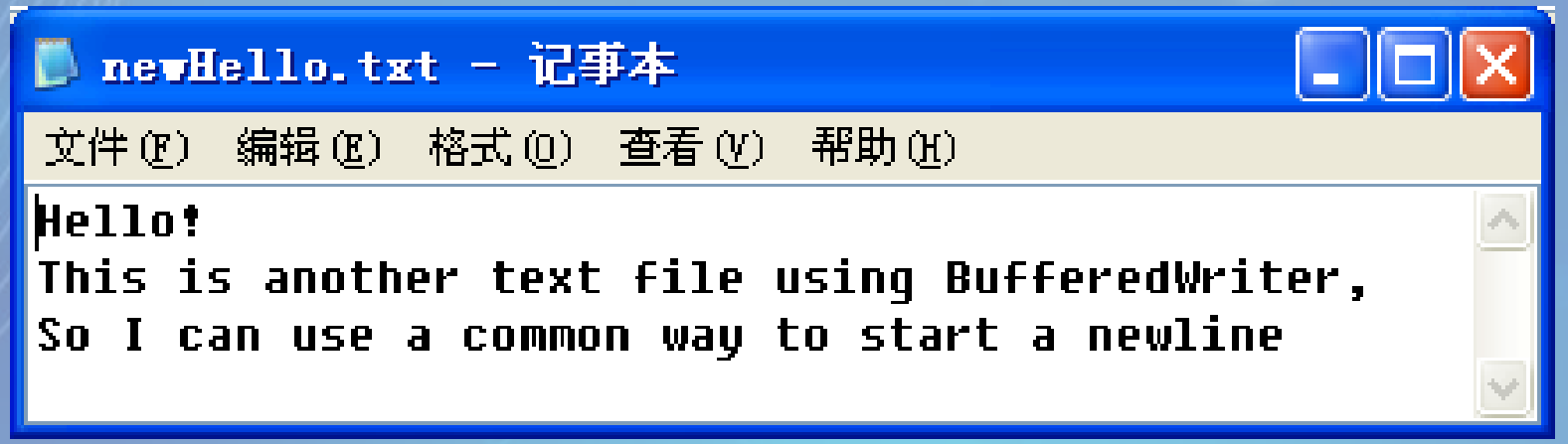
- 下面的例子演示了BufferedWriter的使用

例：写入文本文件，使用BufferedWriter

```
import java.io.*;
class BufferedWriterTester {
    public static void main ( String[] args ) throws IOException {
        String fileName = "C:/newHello.txt" ;
        BufferedWriter out = new BufferedWriter(new  FileWriter( fileName ) );
        out.write( "Hello!"  );
        out.newLine() ;
        out.write( "This is another text file using BufferedWriter,"  );
        out.newLine(); ;
        out.write( "So I can use a common way to start a newline" );
        out.close();
    }
}
```

例：写入文本文件，使用BufferedWriter

- 用任何文本编辑器打开newHello.txt都会出现正确的换行效果



读文本文件

<5.3.2>

本节知识点：Reader 、FileReader 、BufferedReader和readLine() 、文本文件复制

本节知识点

- Reader 类
- FileReader 类
- BufferedReader类和readLine()方法
- 举例：文本文件复制

读文本文件相关的类

- FileReader类
  - 从文本文件中读取字符。
  - 继承自Reader抽象类的子类InputStreamReader。
- BufferedReader
  - 读文本文件的缓冲器类。
  - 具有readLine()方法，可以对换行符进行鉴别，一行一行地读取输入流中的内容。
  - 继承自Reader。

例：读文本文件并显示在屏幕上

- 下面的例子从文本文件中读取数据并在显示器上显示

例：读文本文件并显示

```
import java.io.*;
class BufferedReaderTester {
    public static void main ( String[] args ) {
        String fileName = "C:/Hello.txt" , line;
        try {
            BufferedReader in = new BufferedReader(new FileReader( fileName  ) );
            line = in.readLine();   //读取一行内容
            while ( line != null ) {
            System.out.println( line );  line = in.readLine();
            }
            in.close();
         }
         catch ( IOException iox ) { System.out.println("Problem reading " + fileName ); }
    }
}
```

例：读文本文件并显示

- 运行该程序，屏幕上将逐行显示出Hello.txt文件中的内容。
- FileReader对象：创建后将打开文件，如果文件不存在，会抛出一个IOException
- BufferedReader类的readLine()方法：从一个面向字符的输入流中读取一行文本。如果其中不再有数据，返回null
- Reader类的read()方法：也可用来判别文件结束。该方法返回的一个表示某个字符的int型整数，如果读到文件末尾，返回 -1。据此，可修改本例中的读文件部分：

```
int c;
while((c=in.read())!= -1)  System.out.print((char)c);
```

- close()方法：为了操作系统可以更为有效地利用有限的资源，应该在读取完毕后，调用该方法

例：文件复制

- 接下来再看一个文件复制的例子

例：文件复制

- 指定源文件和目标文件名，将源文件的内容复制到目标文件。调用方式为：
  java copy sourceFile destinationFile
  共包括两个类
- CopyMaker
  - private boolean openFiles()
  - private boolean copyFiles()
  - private boolean closeFiles()
  - public boolean copy(String src, String dst)
- FileCopy
  - main()

例：文件复制

```
import java.io.*;
class CopyMaker {
    String sourceName, destName;
    BufferedReader source;
    BufferedWriter dest;
    String line;
```

例：文件复制

```
private boolean openFiles() {
      try {
          source = new BufferedReader(new FileReader( sourceName ));
      }
      catch ( IOException iox ) {
         System.out.println("Problem opening " + sourceName );
         return false;
      }
      try {
          dest = new BufferedWriter(new FileWriter( destName ));
      }
      catch ( IOException iox )
      {
         System.out.println("Problem opening " + destName );
         return false;
      }
      return true;
}
```

例：文件复制

```
private boolean copyFiles() {
    try {
        line = source.readLine();
        while ( line != null ) {
            dest.write(line);
            dest.newLine();
            line = source.readLine();
        }
    }
    catch ( IOException iox ) {
        System.out.println("Problem reading or writing" );
        return false;
    }
    return true;
}
```

例：文件复制

```
private boolean closeFiles() {
      boolean retVal=true;
     try {  source.close(); }
     catch ( IOException iox ) {
          System.out.println("Problem closing " + sourceName );
       retVal = false;
      }
      try {  dest.close();   }
      catch ( IOException iox ) {
           System.out.println("Problem closing " + destName );
           retVal = false;
       }
       return retVal;
}
```

例：文件复制

```
    public boolean copy(String src, String dst ) {
         sourceName = src ;
        destName    = dst ;
        return openFiles() && copyFiles() && closeFiles();
    }
}
public class FileCopy //一个文件中只能有一个公有类
{
     public static void main ( String[] args ) {
         if ( args.length == 2 )  new CopyMaker().copy(args[0], args[1]);
        else   System.out.println("Please Enter File names");
     }
}
```

例：文件复制

- 此文件FileCopy.java编译后生成FileCopy.class和CopyMaker.class两个字节码文件。
- 运行结果
  - 在命令行方式下执行如下命令

    java FileCopy c:/Hello.txt c:/CopyHello.txt
  - 则在C盘根目录下会出现CopyHello.txt文件，内容与Hello.txt完全相同。

结束语

- 这一节我们通过实例学习了如何从文本文件中读取数据。

写二进制文件

<5.3.3>

本节知识点：

- OutputStream
- FileOutputStream
- BufferedOutputStream
- DataOutputStream
  - writeInt()
  - writeDouble()
  - writeBytes()
  - ……

为什么需要二进制文件

- 输入输出更快
- 比文本文件小很多
- 有些数据不容易被表示为字符

抽象类OutputStream

- 派生类FileOutputStream
  - 用于一般目的输出（非字符输出）；
  - 用于成组字节输出。
- 派生类DataOutputStream
  - 具有写各种基本数据类型的方法；
  - 将数据写到另一个输出流；
  - 它在所有的计算机平台上使用同样的数据格式；
  - 其中size方法，可作为计数器，统计写入的字节数。

DataOutputStream类的成员

名称	说明
public DataOutputStream(OutputStream out)	构造函数，参数为一个OutputStream对象作为其底层的输出对象
protected int written	私有属性，代表当前已写出的字节数
public void flush() public final int size()	冲刷此数据流，使流内的数据都被写出 返回私有变量 written 的值，即已经写出的字节数
public void write(int b)	向底层输出流输出 int 变量的低八位。执行后，记录写入字节数的计数器加 1
public final void writeBoolean(boolean b)	写出一个布尔数，true 为 1，false 为 0,执行后计数器增加 1
public final void writeByte(int b)	将 int 参数的低八位写入，舍弃高 24 位，计数器增加 1

DataOutputStream类的成员

public void writeBytes(String s)	字符串中的每个字符被丢掉高八位写入流中，计数器增加写入的字节数，即字符个数
public final void writeChar(int c)	将16-bit字符写入流中，高位在前，计数器增加2
public void writeDouble(double v)	写双精度数，计数器增加8
public void writeFloat(float f)	写单精度数，计数器增加4
public void writeInt(int I)	写整数，计数器增加4
public void writeLong(long l)	写长整数，计数器增加8
public final void writeShort(int s)	写短整数，计数器增加2

例：将int写入文件

- 下面的例题中用DataOutputStream流对象将int数据写入二进制文件。

例：将int写入文件

- 将三个int型数字255/0/ - 1写入数据文件data1.dat

```
import java.io.*;
class FileOutputstreamTester {
  public static void main ( String[] args ) {
     String fileName = "c:/data1.dat" ;
     int value0  = 255, value1  = 0, value2 = -1;
     try {
        DataOutputStream out = new DataOutputStream(
                                  new FileOutputStream( fileName  ) );
       out.writeInt( value0 );
       out.writeInt( value1 );
       out.writeInt( value2 );
       out.close();
     }
     catch ( IOException iox ){
     System.out.println("Problem writing " + fileName );   }
     }
}
```

例：将int写入文件

- 运行结果
  - 运行程序后，在C盘生成数据文件data1.dat
  - 用写字板打开没有任何显示
  - 用ultraEdit打开查看其二进制信息，内容为00 00 00 FF 00 00 00 00 FF FF FF FF，每个int数字都是32个bit的
- 说明
  - FileOutputStream类的构造方法负责打开文件“data1.dat”用于写数据
  - FileOutputStream类的对象与DataOutputStream对象连接，写基本类型的数据

BufferedOutputStream类

- 写二进制文件的缓冲流类
- 对于大量数据的写入，可提高效率

BufferedOutputStream类

- 用法示例：

```
DataOutputStream out = new DataOutputStream(
                    new BufferedOutputStream( new FileOutputStream( fileName ) ) );
```

例：向文件写入多种数据，统计字节数

- 下面的例题通过DataOutputStream流向文件中写多种类型的数据，并统计字节数

例：向文件写入多种数据，统计字节数

```
import java.io.*;
class BufferedOutputStreamTester {
  public static void main ( String[] args ) throws IOException {
        String fileName = "mixedTypes.dat" ;
        DataOutputStream dataOut = new DataOutputStream(
                        new BufferedOutputStream(
                          new FileOutputStream( fileName  ) ) );
        dataOut.writeInt( 0 );
        System.out.println( dataOut.size()  + " bytes have been written.");
        dataOut.writeDouble( 31.2 );
        System.out.println( dataOut.size()  + " bytes have been written.");
        dataOut.writeBytes("JAVA");
        System.out.println( dataOut.size()  + " bytes have been written.");
        dataOut.close();
  }
}
```

例：向文件写入多种数据，统计字节数

- 运行结果
  - 4 bytes have been written
  - 12 bytes have been written
  - 16 bytes have been written

例：向文件写一个字节并读取

- 向文件中写入内容为-1的一个字节，并读取出来。

```
import java.io.*;
public class FileOutputStreamTester {
   public static void main(String[] args) throws Exception {
      DataOutputStream out=new DataOutputStream(
                        new FileOutputStream("c:/trytry.dat"));
       out.writeByte(-1);  out.close();
       DataInputStream in=new DataInputStream(
                        new FileInputStream("c:/trytry.dat"));
       int a=in.readByte();
       System.out.println(Integer.toHexString (a));
       System.out.println(a);
       in.skip (-1);  //往后一个位置，以便下面重新读出
       a=in.readUnsignedByte();
       System.out.println(Integer.toHexString (a));
       System.out.println(a);  in.close();
   }
}
```

运行结果：

ffffffff

-1

ff

255

结束语

- 这一节我们学习了如何向二进制文件中写数据

读二进制文件

< 5.3.4 >

本节知识点

- FileInputStream
- DataInputStream
- BufferedInputSteam
- 例题：
  - 读写整数
  - 读写单字节

例：读取二进制文件中的3个int型数字并相加

- 前面有例题曾经向二进制文件中写了三个int数据，接下来的程序，从文件中读取这三个数据。

例：读取二进制文件中的3个int型数字并相加

```
import java.io.*;
class DataInputStreamTester {
        public static void main ( String[] args ) {
            String fileName = "c:\\data1.dat";
            int sum = 0;
      try {
          DataInputStream instr = new DataInputStream(
          new BufferedInputStream(new FileInputStream( fileName )));
          sum += instr.readInt();
          sum += instr.readInt();
          sum += instr.readInt();
          System.out.println( "The sum is: " + sum );
          instr.close();
      }
            catch ( IOException iox ) {
                System.out.println("Problem reading " + fileName ); }
            }
    }
}
```

运行结果：
254

例：通过捕获异常控制读取结束

- 接下来的例题中，就通过捕获异常来判断读到文件尾。

例：通过捕获异常控制读取结束

```
import java.io.*;
class DataInputStreamTester {
  public static void main ( String[] args ) {
     String fileName = "c:/data1.dat" ;   long sum = 0;
     try {
         DataInputStream instr = new DataInputStream(new BufferedInputStream(
                                                          new ileInputStream(fileName)));
         try {
            while ( true )
            sum += instr.readInt();
         }
         catch ( EOFException  eof ) {
           System.out.println( "The sum is: " + sum );
           instr.close();
         }
     }
     catch ( IOException iox ) {
                System.out.println("IO Problems with " + fileName ); }
     }
  }
}
```

例：用字节流读取文本文件

- 由于文本文件的存储方式其实也是二进制代码，因此也可使用InputStream类的方法读取
- 下面的例题用InputStream类读取文本文件并显示：

例：用字节流读取文本文件

```
import java.io.*;
public class InputStreamTester {
        public static void main(String[] args) throws IOException {
            FileInputStream s=new FileInputStream("c:/Hello.txt");
        int c;
        while ((c = s.read()) != -1) //读取1字节，结束返回-1
            System.out.write(c);
        s.close();
    }
}
```

- 说明
  - read()方法读取一个字节，转化为[0 ,255]的之间的一个整数，返回一个int。如果读到了文件末尾，则返回-1。
  - write(int)方法写一个字节的低8位，忽略高24位。

读写字节

- DataOutputStream类有专门写一个字节的方法
- DataInputStream类有专门读一个字节的方法

读写字节

- DataOutputStream的writeByte方法
  - public final void writeByte(int b) throws IOException
  - 将int的最不重要字节写入输出流
- DataInputStream的readUnsignedByte方法
  - public final int readUnsignedByte() throws IOException
  - 从输入流中读取1字节存入int的最不重要字节

例：文件复制

- 下面的例题，通过读写字节的方法进行文件的复制

例：文件复制

- 从命令行输入源文件名和目标文件名，将源文件复制为目标文件。

```
import java.io.*;
class CopyBytes {
  public static void main ( String[] args ) {
    DataInputStream  instr;
    DataOutputStream outstr;
    if ( args.length != 2 ) {
        System.out.println("Please enter file names");
        return;
    }
    try {
        instr = new DataInputStream(new
          BufferedInputStream(new FileInputStream( args[0] )));
        outstr = new DataOutputStream(new
          BufferedOutputStream(new FileOutputStream( args[1] )));
```

例：文件复制

```
        try {
            int data;
            while ( true ) {
                data = instr.readUnsignedByte() ;
                outstr.writeByte( data ) ;
            }
        }
        catch ( EOFException  eof ) {
            outstr.close();
            instr.close();
            return;
        }
    }
    catch ( FileNotFoundException nfx )
    {   System.out.println("Problem opening files" );    }
    catch ( IOException iox )
    {   System.out.println("IO Problems" );    }
  }
}
```

File类

<5.3.5>

File类

- 表示磁盘文件信息
- 定义了一些与平台无关的方法来操纵文件

File类的作用

- 创建、删除文件；
- 重命名文件；
- 判断文件的读写权限及是否存在；
- 设置和查询文件的最近修改时间等；
- 构造文件流可以使用File类的对象作为参数。

例：File类举例

- 下面的例题演示了File类的使用

例：File类举例

- 在C盘创建文件Hello.txt，如果存在则删除旧文件，不存在则直接创建新的

```
import java.io.*;
public class FileTester {
    public static void main(String[] args) {
    File f=new File("c:"+File.separator+"Hello.txt");
    if (f.exists())  f.delete();
    else
        try{
            f.createNewFile();
        }
        catch(Exception e){
            System.out.println(e.getMessage());
        }
    }
}
```

例：File类举例

- 运行结果
  - 因为在前面的例子中已经创建了c:\Hello.txt，所以第一次运行将删除这个文件
  - 第二次运行则又创建了一个此名的空文件
- 分析
  - 在试图打开文件之前，可以使用File类的isFile方法来确定File对象是否代表一个文件而非目录）
  - 还可通过exists方法判断同名文件或路径是否存在，进而采取正确的方法，以免造成误操作

例：改进的文件复制程序

- 接下来我们对前面讲过的复制文件的程序进行一点改进，在复制之前先判断文件是否存在

例：改进的文件复制程序

```
import java.io.*;
class NewCopyBytes{
   public static void main ( String[] args ) {
       DataInputStream  instr;
       DataOutputStream outstr;
       if ( args.length != 2 ) {
          System.out.println("Please Enter file names!");
          return;
       }
      File inFile  = new File( args[0] );
      File outFile = new File( args[1] );
      if ( outFile.exists() ) {
         System.out.println( args[1] + " already exists");
         return;
      }
      if ( !inFile.exists() )  {
         System.out.println( args[0] + " does not exist");
         return;
      }
```

例：改进的文件复制程序

```
try{
    instr = new DataInputStream(new BufferedInputStream(
                                    new FileInputStream( inFile )));
    outstr = new DataOutputStream(new BufferedOutputStream(
                                    new FileOutputStream( outFile  )));
    try {
        int data;
        while ( true ) {
        data = instr.readUnsignedByte() ; outstr.writeByte( data ) ;  }
    }
    catch ( EOFException  eof )
    {    outstr.close();   instr.close();    return;     }
  }
  catch ( FileNotFoundException nfx )
  {  System.out.println("Problem opening files" );    }
  catch ( IOException iox )
  {  System.out.println("IO Problems" );    }
 }
}
```

结束语

- 这一节介绍了用来表示文件信息和管理文件的File类

处理压缩文件

<5.3.6>

压缩流类

- java.util.zip包中提供了一些类，使我们可以以压缩格式对流进行读写
- 都继承自字节流类OutputStream和InputStream

压缩流类

- GZIPOutputStream和ZipOutputStream
  - 可分别把数据压缩成GZIP格式和Zip格式
- GZIPInputStream和ZipInputStream
  - 可以分别把压缩成GZIP格式或Zip的数据解压缩恢复原状

简单的GZIP压缩格式

- GZIPOutputStream
  - 父类是DeflaterOutputStream
  - 可以把数据压缩成GZIP格式
- GZIPInputStream
  - 父类是 InflaterInputStream
  - 可以把压缩成GZIP格式的数据解压缩

例：压缩和解压缩Gzip文件

- 下面的例题演示了如何将文件压缩为Gzip格式，以及解压缩。

例：压缩和解压缩Gzip文件

- 将文本文件“Hello.txt” 压缩为文件“test.gz” ，再解压该文件，显示其中内容，并另存为“newHello.txt”

```
import java.io.*;
import java.util.zip.*;
public class GZIPTester {
   public static void main(String[] args) throws IOException {
     FileInputStream in = new FileInputStream("c:/Hello.txt");
     GZIPOutputStream out = new GZIPOutputStream(
                              new FileOutputStream("c:/test.gz"));
     System.out.println("Writing compressing file from
                                 c:/Hello.txt to c:/test.gz");
     int c;
     while((c = in.read()) != -1) out.write(c); //写压缩文件
in.close();
out.close();
```

例：压缩和解压缩Gzip文件

```
        System.out.println("Reading file form c:/test.gz to monitor");
        BufferedReader in2 = new BufferedReader( new InputStreamReader(new GZIPInputStream(
                                                                new FileInputStream("c:/test.gz"))));
        String s;
        while((s = in2.readLine()) != null) System.out.println(s);
        in2.close();
        System.out.println("Writing decompression to c:/newHello.txt");
        GZIPInputStream in3=new GZIPInputStream(
                                    new FileInputStream("c:/test.gz"));
        FileOutputStream out2=new FileOutputStream("c:/newHello.txt");
        while((c=in3.read())!=-1) out2.write(c);
        in3.close();
        out2.close();
    }
}
```

例：压缩和解压缩Gzip文件

- 运行结果
  - 首先生成了压缩文件“test.gz”
  - 再读取显示其中的内容，和“Hello.txt”中的内容完全一样
  - 解压缩文件“newHello.txt”，和“Hello.txt”中的内容也完全相同
- 说明
  - read()方法读取一个字节，转化为[0 ,255]的之间的一个整数，返回一个int。如果读到了文件末尾，则返回-1。
  - write(int)方法写一个字节的低8位，忽略了高24位。

Zip文件压缩与解压缩

- Zip文件
  - 可能含有多个文件，所以有多个入口（Entry）

Zip文件压缩与解压缩

- Zip文件
  - 可能含有多个文件，所以有多个入口（Entry）
  - 每个入口用一个ZipEntity对象表示，该对象的getName()方法返回文件的最初名称
- ZipOutputStream
  - 父类是DeflaterOutputStream
  - 可以把数据压缩成ZIP格式
- ZipInputStream
  - 父类是InflaterInputStream
  - 可以把压缩成ZIP格式的数据解压缩

例：Zip文件压缩与解压缩

例：Zip文件压缩与解压缩

- 从命令行输入若干个文件名，将所有文件压缩为“c:/test.zip”，再从此压缩文件中解压并显示

```
import java.io.*;
import java.util.*;
import java.util.zip.*;
public class ZipOutputStreamTester {
   public static void main(String[] args) throws IOException {
      ZipOutputStream out=new ZipOutputStream(
                                 new BufferedOutputStream(
                              new FileOutputStream("c:/test.zip")));
```

例：Zip文件压缩与解压缩

```
for(int i = 0; i < args.length; i++) {
   System.out.println("Writing file " + args[i]);
   BufferedInputStream in =new BufferedInputStream(
                                 new FileInputStream(args[i]));
   out.putNextEntry(new ZipEntry(args[i]));
   int c;
   while((c = in.read()) != -1)  out.write(c);
   in.close();
}
out.close();
```

例：Zip文件压缩与解压缩

```
    System.out.println("Reading file");
    ZipInputStream in2 =new ZipInputStream(
                              new BufferedInputStream(
                               new FileInputStream("c:/test.zip")));
    ZipEntry ze;
    while((ze = in2.getNextEntry()) != null) {
       System.out.println("Reading file " + ze.getName());
       int x;
       while((x = in2.read()) != -1)  System.out.write(x);
       System.out.println();
    }
    in2.close();
  }
}
```

例：Zip文件压缩与解压缩

- 运行结果
  - 在命令行输入两个文本文件名后，将生成c:/test.zip文件
  - 在屏幕上显示出解压后每个文件的内容
  - 在资源管理器窗口中，使用winzip软件可解压缩该文件，恢复出和原来文件相同的两个文本文件

例：解压缩Zip文件，并恢复其原来路径

例：解压缩Zip文件，并恢复其原来路径

```
import java.util.*;
import java.util.zip.*;
import java.lang.*;
import java.io.*;
class Unzip {
   byte doc[]=null;   //存储解压缩数据的缓冲字节数组
   String Filename=null;  //压缩文件名字符串
   String UnZipPath=null; //解压缩路径字符串
   public Unzip(String filename,String unZipPath) {
     this.Filename=filename;
     this.UnZipPath=unZipPath;
     this.setUnZipPath (this.UnZipPath);
   }
   public Unzip(String filename) {
     this.Filename=new String(filename);
     this.UnZipPath=null;
     this.setUnZipPath (this.UnZipPath);
   }
```

例：解压缩Zip文件，并恢复其原来路径

```
private void setUnZipPath(String unZipPath) {
  if(unZipPath.endsWith("\\"))
    this.UnZipPath=new String(unZipPath);
  else
    this.UnZipPath=new String(unZipPath+"\\");
}
public void doUnZip() {
   try {
   ZipInputStream zipis=new ZipInputStream(
                              new FileInputStream(Filename));
   ZipEntry fEntry=null;
   while((fEntry=zipis.getNextEntry())!=null) {
     if (fEntry.isDirectory())  //是路径则创建路径
        checkFilePath(UnZipPath+fEntry.getName());
```

例：解压缩Zip文件，并恢复其原来路径

```
else {   //是文件则解压缩文件
        String fname=new String(UnZipPath+fEntry.getName());
        try{
            FileOutputStream out = new FileOutputStream(fname);
            doc=new byte[512];
            int n;
            while ((n = zipis.read(doc,0,512)) != -1)
               out.write(doc, 0, n);
            out.close();
            out=null;
            doc=null;
          }
          catch (Exception ex) {   }
      }
    }
    zipis.close(); //关闭输入流
  }
     catch(IOException ioe) { System.out.println(ioe);    }
}
```

例：解压缩Zip文件，并恢复其原来路径

```
 private void checkFilePath(String dirName) throws IOException {
        File dir = new File(dirName);
        if(!dir.exists())
           dir.mkdirs();
  }
}
//主类，用于输入参数，生成Unzip类的实例
public class UnZipTester {
   public static void main(String [] args) {
     String zipFile=args[0]; //第一个参数为zip文件名
     String unZipPath=args[1]+"\\";  //第二个参数为指定解压缩路径
     Unzip myZip=new Unzip(zipFile,unZipPath);
     myZip.doUnZip();
   }
}
```

对象序列化

<5.3.7>

对象序列化

- 保存对象的信息，在需要的时候，再读取这个对象

ObjectInputStream/ObjectOutputStream类

- 实现对象的读写
  - 通过ObjectOutputStream把对象写入磁盘文件
  - 通过ObjectInputStream把对象读入程序
- 不保存对象的transient和static类型的变量
- 对象要想实现序列化，其所属的类必须实现Serializable接口

ObjectOutputStream

- 必须通过另一个流构造ObjectOutputStream:

```
FileOutputStream out = new FileOutputStream("theTime");
ObjectOutputStream s =  new ObjectOutputStream(out);
s.writeObject("Today");
s.writeObject(new Date());
s.flush();
```

ObjectInputStream

- 必须通过另一个流构造ObjectInputStream：

```
FileInputStream in = new FileInputStream("theTime");
ObjectInputStream s = new ObjectInputStream(in);
String today = (String)s.readObject();
Date date = (Date)s.readObject();
```

Seriealizable

- 空接口，使类的对象可实现序列化

Seriealizable

- Serializable 接口的定义

```
package java.io;
public interface Serializable {
    // there's nothing in here!
};
```

- 实现Serializable接口的语句

```
public class MyClass implements Serializable {
    ...
}
```

- 使用关键字transient可以阻止对象的某些成员被自动写入文件

例：创建一个书籍对象输出并读出

例：创建一个书籍对象输出并读出

- 创建一个书籍对象，并把它输出到一个文件book.dat中，然后再把该对象读出来，在屏幕上显示对象信息

```
class Book implements Serializable {
    int id;
    String name;
    String author;
    float price;
    public Book(int id,String name,String author,float price) {
            this.id=id;
            this.name=name;
            this.author=author;
            this.price=price;
    }
}
```

例：创建一个书籍对象输出并读出

```
import java.io.*;
public class SerializableTester {
    public static void main(String args[]) throws
    IOException,ClassNotFoundException {
        Book book=new Book(100032,"Java Programming
                                    Skills","Wang Sir",30);
        ObjectOutputStream oos=new ObjectOutputStream(
                        new FileOutputStream("c:/book.dat"));
       oos.writeObject(book);
        oos.close();
```

例：创建一个书籍对象输出并读出

```
        book=null;
         ObjectInputStream ois=new ObjectInputStream(
                              new FileInputStream("c:/book.dat"));
         book=(Book)ois.readObject();
          ois.close();
          System.out.println("ID is:"+book.id);
          System.out.println("name is:"+book.name);
          System.out.println("author is:"+book.author);
          System.out.println("price is:"+book.price);
    }
}
```

运行结果：

将生成book.dat文件，并在屏幕显示：

ID is:100032
name is:Java Programming Skills
author is:Wang Sir
price is:30.0

Externalizable接口

- 实现该接口可以控制对象的读写

Externalizable接口

- API中的说明为

  public interface Externalizable extends Serializable
- 其中有两个方法writeExternal()和readExternal()，因此实现该接口的类必须实现这两个方法
- ObjectOutputStream的writeObject()方法只写入对象的标识，然后调用对象所属类的writeExternal()
- ObjectInputStream的readObject()方法调用对象所属类的readExternal()

随机文件读写

<5.3.8>

RandomAccessFile类

- 可跳转到文件的任意位置读/写数据
- 可在随机文件中插入数据，而不破坏该文件的其他数据
- 实现了DataInput 和 DataOutput 接口，可使用普通的读写方法
- 有个位置指示器，指向当前读写处的位置。刚打开文件时，文件指示器指向文件的开头处。对文件指针显式操作的方法有：
  - int skipBytes(int n)：把文件指针向前移动指定的n个字节
  - void seek(long)：移动文件指针到指定的位置。
  - long getFilePointer()：得到当前的文件指针。
- 在等长记录格式文件的随机读取时有很大的优势，但仅限于操作文件，不能访问其它IO设备，如网络、内存映像等

RandomAccessFile类

- 构造方法

```
public RandomAccessFile(File file,String mode)
        throws FileNotFoundException
public RandomAccessFile(String name, String mode)
        throws FileNotFoundException
```

- 构造RandomAccessFile对象时，要指出操作：仅读，还是读写

```
new RandomAccessFile("farrago.txt", "r");
new RandomAccessFile("farrago.txt", "rw");
```

RandomAccessFile类常用API

名称	说明
public RandomAccessFile(File f, String mode)	构造函数，指定关联的文件，以及处理方式：’r’为只读,’rw’为读写
public void setLength(long newLength)	设置文件长度，即字节数
public long length()	返回文件的长度，即字节数
public void seek(long pos)	移动文件位置指示器，pos 指定从文件开头的偏离字节数。可以超过文件总字节数，但只有写操作后，才能扩展文件大小
public int skipBytes(int n)	跳过 n 个字节，返回数为实际跳过的字节数
public int read()	从文件中读取一字节，字节的高 24 位为 0。如遇到结尾，则返回-1
public final double readDouble()	读取 8 个字节
public final void writeChar(int v)	写入一个字符，两个字节，高位先写入
public final void writeInt(int v)	写入四个字节的 int 型数字

例：随机文件读写

例：随机文件读写

- 创建一个雇员类，包括姓名、年龄。姓名不超过8个字符，年龄是int类型。每条记录固定为20个字节。使用RandomAccessFile向文件添加、修改、读取雇员信息

```
import java.io.*;
class Employee {
   char name[]={'\u0000','\u0000','\u0000','\u0000',
                '\u0000','\u0000','\u0000','\u0000'};
   int age;
   public Employee(String name,int age) throws Exception {
      if(name.toCharArray ().length>8)
        System.arraycopy(name.toCharArray(),0,this.name,0,8);
      else
        System.arraycopy(name.toCharArray(),0,this.name,
                                  0,name.toCharArray().length);
        this.age=age;
   }
}
```

例：随机文件读写

```
public class RandomAccessFileTester {
        String Filename;
        public RandomAccessFileTester (String Filename)     {
            this.Filename=Filename;
        }
        public void writeEmployee(Employee e,int n) throws Exception {
          RandomAccessFile ra=new RandomAccessFile(Filename,"rw");
          ra.seek(n*20); //将位置指示器移到指定位置上
          for(int I=0;I<8;I++)  ra.writeChar (e.name[I]);
          ra.writeInt(e.age);
          ra.close();
        }
      public void readEmployee(int n) throws Exception {
          char buf[]=new char[8];
          RandomAccessFile ra=new RandomAccessFile(Filename,"r");
          ra.seek(n*20);
          for(int I=0;I<8;I++) buf[I]=ra.readChar();
          System.out.print("name:");
          System.out.println(buf);
        System.out.println("age:"+ra.readInt());
           ra.close();
        }
```

例：随机文件读写

```
public static void main(String[] args) throws Exception {
 RandomAccessFileTester t=new RandomAccessFileTester ("d:/temp/1.txt");
   Employee e1=new Employee("ZhangSantt",23);
    Employee e2=new Employee( “李晓珊” ,33);
   Employee e3=new Employee( “王华” ,19);
   t.writeEmployee(e1,0);
    t.writeEmployee(e3,2);
   System.out.println("第一个雇员信息");
     t.readEmployee(0);
    System.out.println("第三个雇员信息");
    t.readEmployee(2);
    t.writeEmployee (e2,1);
    System.out.println("第二个雇员信息");
    t.readEmployee (1);
  }
}
```

运行结果：

```
第一个雇员信息
name:ZhangSan
age:23
第三个雇员信息
name:王华
age:19
第二个雇员信息
name:李晓珊
```

第5章小结

本章内容

- I/O流的概念以及分类
- 读写文本文件、二进制文件的方法
- 处理流的概念及用法
- File类
- 压缩流类
- 对象序列化的常用流类及接口
- 随机读写文件的流类

结束语

- 这一章的内容就是这些