特别说明

.doc in alpha 豆丁

7.3 使用 POI 来处理 Excel 和 Word 文件格式

Microsoft 的 Office 系列产品拥有大量的用户，Word、Excel 也成为办公文件的首选。在 Java 中，已经有很多对于 Word、Excel 的开源的解决方案，其中比较出色的是 Apache 的 Jakata 项目的 POI 子项目。该项目的官方网站是 http://jakarta.apache.org/poi/。

POI 包括一系列的 API，它们可以操作基于 MicroSoft OLE 2 Compound Document Format 的各种格式文件，可以通过这些 API 在 Java 中读写 Excel、Word 等文件。POI 是完全的 Java Excel 和 Java Word 解决方案。POI 子项目包括：POIFS、HSSF、HDF、HPSF。表 7-2 对它们进行了简要介绍。

表 7-2 POI 子项目介绍

子项目名	说明
POIFS(POI File System)	POIFS 是 POI 项目中最早的最基础的一个模块，是 Java 到 OLE 2 Compound Document Format 的接口，支持读写功能，所有的其他项目都依赖与该项目。
HSSF(Horrible Spreadsheet Format)	HSSF 是 Java 到 Microsoft Excel 97(-2002)文件的接口，支持读写功能
HWPF(Horrible Word Processing Format)	HWPF 是 Java 到 Microsoft Word 97 文件的接口，支持读写功能，但目前该模块还处于刚开始开发阶段，只能实现一些简单文件的操作，在后续版本中，会提供更强大的支持
HPSF(Horrible Property Set Format)	HPSF 是 Java 到 OLE 2 Compound Document Format 文件的属性设置的接口，属性设置通常用来设置文档的属性（标题，作者，最后修改日期等），还可以设置用户定义的属性。HPSF 支持读写功能，当前发布版本中直支持读功能。

7.3.1 对 Excel 的处理类

下面通过 HSSF 提供的接口对 Excel 文件经行处理。首先需要下载 POI 的包，可以到 apache 的官方网站下载，地址为：http://apache.justdn.org/jakarta/poi/，本书采用的是 poi-2.5.1-final-20040804.jar，读者可以下载当前的稳定版本。把下载的包按照前面介绍的方式加入 Build Path，然后新建一个 ch7.poi 包，并创建一个 ExcelReader 类。

ExcelReader 类可以读取一个 XLS 文件，然后将其内容逐行提取出来，写入文本文件。其代码如下。

代码 7.6

```
public class ExcelReader {
// 创建文件输入流
private BufferedReader reader = null;
// 文件类型
private String filetype;
// 文件二进制输入流
```

```
private InputStream is = null;
// 当前的 Sheet
private int currSheet;
// 当前位置
private int currPosition;
// Sheet 数量
private int numOfSheets;
// HSSFWorkbook
HSSFWorkbook workbook = null;
// 设置 Cell 之间以空格分割
private static String EXCEL_LINE_DELIMITER = " ";
// 设置最大列数
private static int MAX_EXCEL_COLUMNS = 64;
// 构造函数创建一个 ExcelReader
public ExcelReader(String inputfile) throws IOException, Exception {
// 判断参数是否为空或没有意义
if (inputfile == null || inputfile.trim().equals("")) {
    throw new IOException("no input file specified");
}
// 取得文件名的后缀名赋值给 filetype
this.filetype = inputfile.substring(inputfile.lastIndexOf(".") + 1);
// 设置开始行为 0
currPosition = 0;
// 设置当前位置为 0
currSheet = 0;
// 创建文件输入流
is = new FileInputStream(inputfile);
// 判断文件格式
if (filetype.equalsIgnoreCase("txt")) {
```

```
        // 如果是 txt 则直接创建 BufferedReader 读取
        reader = new BufferedReader(new InputStreamReader(is));
    }
     else if (filetype.equalsIgnoreCase("xls")) {
        // 如果是 Excel 文件则创建 HSSFWorkbook 读取
        workbook = new HSSFWorkbook(is);
            // 设置 Sheet 数
            numOfSheets = workbook.getNumberOfSheets();
        }
            else {
            throw new Exception("File Type Not Supported");
        }
    }
    // 函数 readLine 读取文件的一行
    public String readLine() throws IOException {
        // 如果是 txt 文件则通过 reader 读取
        if (filetype.equalsIgnoreCase("txt")) {
            String str = reader.readLine();
            // 空行则略去，直接读取下一行
            while (str.trim().equals("")) {
                str = reader.readLine();
            }
            return str;
        }
        // 如果是 XLS 文件则通过 POI 提供的 API 读取文件
        else if (filetype.equalsIgnoreCase("xls")) {
            // 根据 currSheet 值获得当前的 sheet
            HSSFSheet sheet = workbook.getSheetAt(currSheet);
            // 判断当前行是否到但前 Sheet 的结尾
```

```
            if (currPosition > sheet.getLastRowNum()) {
                // 当前行位置清零
                currPosition = 0;
                // 判断是否还有 Sheet
                while (currSheet != numOfSheets - 1) {
                    // 得到下一张 Sheet
                    sheet = workbook.getSheetAt(currSheet + 1);
                    // 当前行数是否已经到达文件末尾
                    if (currPosition == sheet.getLastRowNum()) {
                        // 当前 Sheet 指向下一张 Sheet
                        currSheet++;
                        continue;
                    } else {
                        // 获取当前行数
                        int row = currPosition;
                        currPosition++;
                        // 读取当前行数据
                        return getLine(sheet, row);
                    }
                }
                return null;
            }
            // 获取当前行数
            int row = currPosition;
            currPosition++;
            // 读取当前行数据
            return getLine(sheet, row);
        }
        return null;
```

```
}
// 函数 getLine 返回 Sheet 的一行数据
private String getLine(HSSFSheet sheet, int row) {
    // 根据行数取得 Sheet 的一行
    HSSFRow rowline = sheet.getRow(row);
    // 创建字符创缓冲区
    StringBuffer buffer = new StringBuffer();
    // 获取当前行的列数
    int filledColumns = rowline.getLastCellNum();
    HSSFCell cell = null;
    // 循环遍历所有列
    for (int i = 0; i < filledColumns; i++) {
        // 取得当前 Cell
        cell = rowline.getCell((short) i);
        String cellvalue = null;
        if (cell != null) {
            // 判断当前 Cell 的 Type
            switch (cell.getCellType()) {
            // 如果当前 Cell 的 Type 为 NUMERIC
            case HSSFCell.CELL_TYPE_NUMERIC: {
                // 判断当前的 cell 是否为 Date
                if (HSSFDateUtil.isCellDateFormatted(cell)) {
                    // 如果是 Date 类型则，取得该 Cell 的 Date 值
                    Date date = cell.getDateCellValue();
                    // 把 Date 转换成本地格式的字符串
                    cellvalue = cell.getDateCellValue().toLocaleString();
                }
                // 如果是纯数字
                else {
```

```
                    // 取得当前 Cell 的数值
                    Integer num = new Integer((int) cell
                            .getNumericCellValue());
                    cellvalue = String.valueOf(num);
                }
                break;
            }
            // 如果当前 Cell 的 Type 为 STRIN
            case HSSFCell.CELL_TYPE_STRING:
                // 取得当前的 Cell 字符串
                cellvalue = cell.getStringCellValue().replaceAll("", "");
                break;
            // 默认的 Cell 值
            default:
                cellvalue = " ";
            }
        } else {
            cellvalue = "";
        }
        // 在每个字段之间插入分割符
        buffer.append(cellvalue).append(EXCEL_LINE_DELIMITER);
    }
    // 以字符串返回该行的数据
    return buffer.toString();
}
// close 函数执行流的关闭操作
public void close() {
    // 如果 is 不为空，则关闭 InputSteam 文件输入流
    if (is != null) {
```

```
        try {
            is.close();
        } catch (IOException e) {
            is = null;
        }
    }
    // 如果 reader 不为空则关闭 BufferedReader 文件输入流
    if (reader != null) {
        try {
            reader.close();
        } catch (IOException e) {
            reader = null;
        }
    }
}
```

7.3.2 ExcelReader 的运行效果

下面创建一个 main 函数，用来测试上面的 ExcelReader 类，代码如下。

代码 7.7

```
public static void main(String[] args) {
    try{
        ExcelReader er=new ExcelReader("c:\\xp.xls");
        String line=er.readLine();
        while(line != null){
            System.out.println(line);
            line=er.readLine();
        }
        er.close();
```

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

main 函数先创建一个 ExcelReader 类，然后调用它提供的接口 readLine，对 XLS 文件进行读取，打印到控制台，处理前的 XLS 文件如图 7-12 所示。

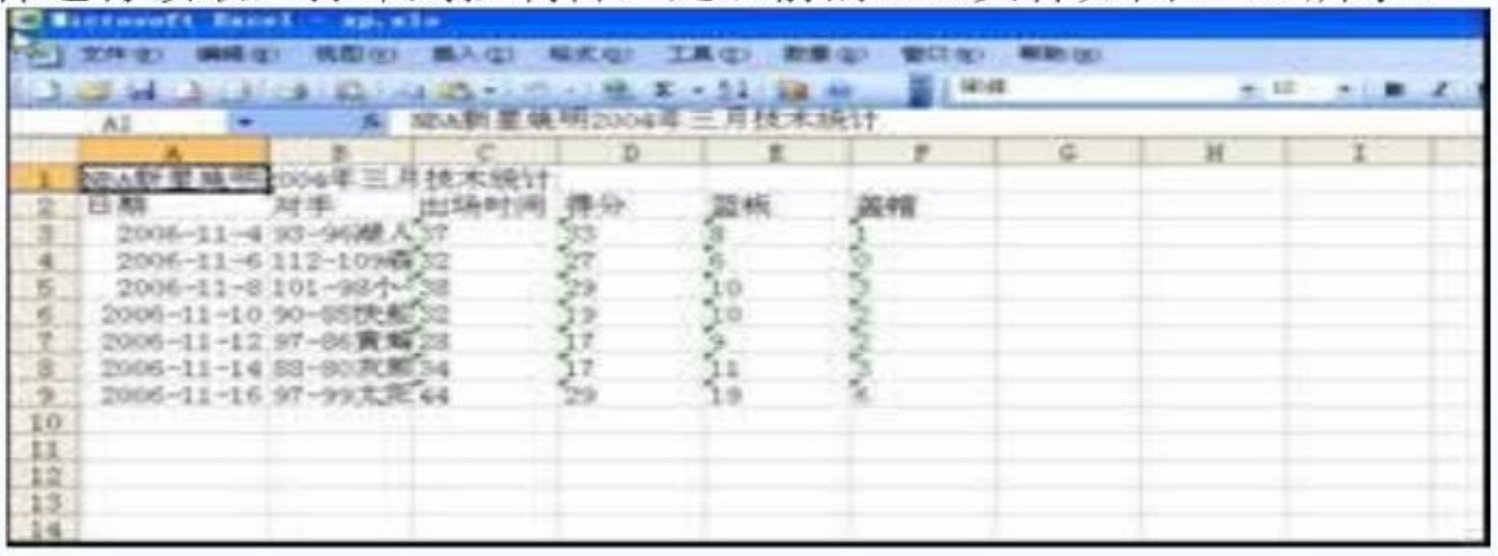

图 7-12 处理前的 XLS 文件内容

运行 main 函数进行内容提取后，Eclipse 的控制台输出如图 7-13 所示。

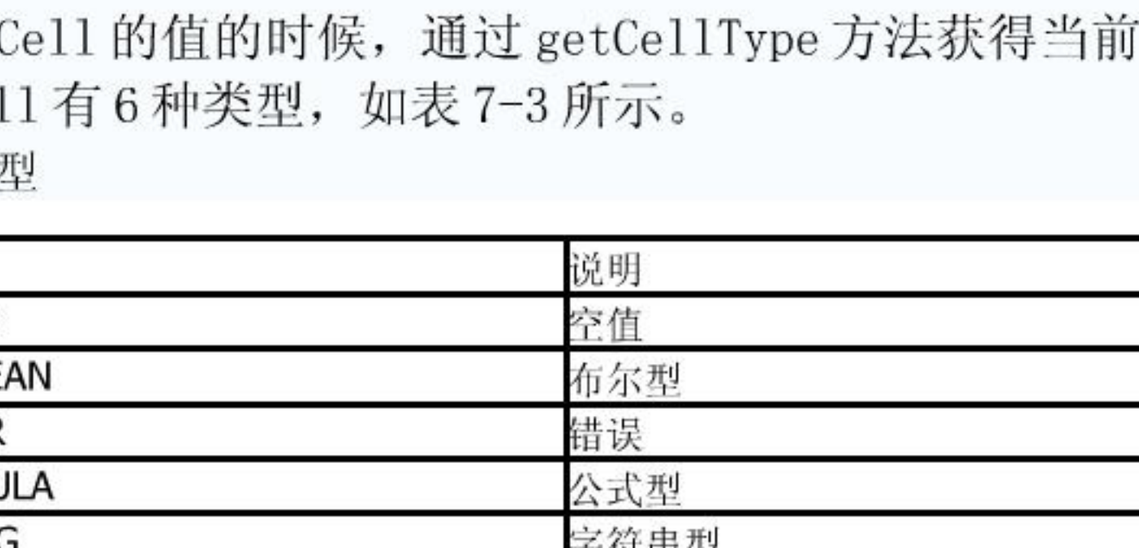

图 7-13 输出结果

可以看到，Excel 文件中的内容已经被成功的输出了出来。

7.3.3 POI 中 Excel 文件 Cell 的类型

在读取每一个 Cell 的值的时候，通过 getCellType 方法获得当前 Cell 的类型，在 Excel 中 Cell 有 6 种类型，如表 7-3 所示。

表 7-3 Cell 的类型

CellType	说明
CELL_TYPE_BLANK	空值
CELL_TYPE_BOOLEAN	布尔型
CELL_TYPE_ERROR	错误
CELL_TYPE_FORMULA	公式型
CELL_TYPE_STRING	字符串型
CELL_TYPE_NUMERIC	数值型

本例采用了 CELL_TYPE_STRING 和 CELL_TYPE_NUMERIC 类型，因为在 Excel 文件中只有字符串和数字。如果 Cell 的 Type 为 CELL_TYPE_NUMERIC 时，还需要进一步判断该 Cell 的数据格式，因为它有可能是 Date 类型，在 Excel 中的 Date 类型也是以 Double 类型的数字存储的。Excel 中的 Date 表示当前时间与 1900 年 1 月 1 日相隔的天数，所以需要调用 HSSFDateUtil 的 isCellDateFormatted 方法，判断该 Cell 的数据格式是否是 Excel Date 类型。如果是，则调用 getDateCellValue 方法，返回一个 Java 类型的 Date。

实际上 Excel 的数据格式有很多，还支持用户自定义的类型，在 Excel 中，选择一个单元格然后右键选择“设置单元格格式”，在弹出的单元格格式中选中“数字”，如图 7-14 所示。

图 7-14 Excel 的单元格格式

图中的数据有数值、货币、时间、日期、文本等格式。这些数据格式在 POI 中的 HSSFDataFormat 类里都有相应的定义。

HSSFDataFormat 是 HSSF 子项目里面定义的一个类。类 HSSFDataFormat 允许用户新建数据格式类型。HSSFDataFormat 类包含静态方法 static java.lang.String getBuiltinFormat(short index)，它可以根据编号返回内置数据类型。另外 static short getBuiltinFormat(java.lang.String format)方法则可以根据数据类型返回其编号，static java.util.List getBuiltinFormats()可以返回整个内置的数据格式列表。

在 HSSFDataFormat 里一共定义了 49 种内置的数据格式，如表 7-4 所示。

表 7-4 HSSFDataFormat 的数据格式

内置数据类型	编号
"General"	0
"0"	1
"0.00"	2
"#,##0"	3
"#,##0.00"	4
"($#,##0_);($#,##0)"	5
"($#,##0_);[Red]($#,##0)"	6
"($#,##0.00);($#,##0.00)"	7
"($#,##0.00_);[Red]($#,##0.00)"	8
"0%"	9
"0.00%"	0xa
"0.00E+00"	0xb
"# ?/?"	0xc

"# ??/??"	0xd
"m/d/yy"	0xe
"d-mmm-yy"	0xf
"d-mmm"	0x10
"mmm-yy"	0x11
"h:mm AM/PM"	0x12
"h:mm:ss AM/PM"	0x13
"h:mm"	0x14
"h:mm:ss"	0x15
"m/d/yy h:mm"	0x16
保留为过国际化用	0x17 - 0x24
"(#,##0_);(#,##0)"	0x25
"(#,##0_);[Red](#,##0)"	0x26
"(#,##0.00_);(#,##0.00)"	0x27
"(#,##0.00_);[Red](#,##0.00)"	0x28
"_($*#,##0_);_($*(#,##0);_($* \"-\"_);_(@_)"	0x29
"_(*#,##0.00_);_(*(#,##0.00);_(*\"-\"??_);_(@_)"	0x2a
"_($*#,##0.00_);_($*(#,##0.00);_($*\"-\"??_);_(@_)"	0x2b
"_($*#,##0.00_);_($*(#,##0.00);_($*\"-\"??_);_(@_)"	0x2c
"mm:ss"	0x2d
"[h]:mm:ss"	0x2e
"mm:ss.0"	0x2f
"##0.0E+0"	0x30
"@" - This is text format	0x31

在上面表中，字符串类型所对应的是数据格式为"@"（最后一行），也就是HSSFDataFormat中定义的值为0x31（49）的那行。Date类型的值的范围是0xe-0x11，本例子中的Date格式为""m/d/yy""，在HSSFDataFormat定义的值为0xe（14）。

需要注意的一点是，所创建的Excel必须是在Microsoft Excel 97到Excel XP的版本上的，如果在Excel 2003中创建文件后，在使用POI进行解析时，可能会出现问题。它会把Date类型当作自定义类型。POI目前只提供对Microsoft Excel XP以下的版本的支持，在以后的版本中，希望会提供对Microsoft Excel 2003更好的支持。

7.3.4 对Word的处理类

除了支持对Excel文件的读取外，POI还提供对Word的DOC格式文件的读取。但在它的发行版本中没有发布对Word支持的模块，需要另外下载一个POI的扩展的Jar包。用户可以到http://www.ibiblio.org/maven2/org/textmining/tm-extractors/0.4/下载，本书采用的是tm-extractors-0.4_zip。

下载后，把该包加入工程的Build Path中，然后在ch7.poi包下新建一个类WordReader，该类提供一个静态方法readDoc，读取一个DOC文件并返回文本。函数内容很简单，就是调用WordExtractor的API来提取DOC的内容到字符串，该函数的代码如下。

代码7.8

```
public static String readDoc(String doc) throws Exception {

    // 创建输入流读取DOC文件

    FileInputStream in = new FileInputStream(new File(doc));
```

```
    WordExtractor extractor = null;
    String text = null;
    // 创建 WordExtractor
    extractor = new WordExtractor();
    // 对 DOC 文件进行提取
    text = extractor.extractText(in);
    return text;
}
```

在同一个类里创建一个 main 函数，测试 WordReader，该 main 函数代码如下。

代码 7.9

```
public static void main(String[] args) {
        try{
            String text = WordReader.readDoc("c:/test.doc");
            System.out.println(text);
        }catch(Exception e){
            e.printStackTrace();
        }
    }
```

处理前的 Doc 文件如图 7-15 所示。

图 7-15　处理前的 Word 文档

使用代码处理后的文本如图 7-16 所示。

图 7-16 处理后的结果

可以看到 Word 文档内的文本已经全部被提取了出来。

7.4 使用 Jacob 来处理 Word 文档

Word 或 Excel 程序是以一种 COM 组件形式存在的。如果能够在 Java 中调用 Word 的 COM 组件，就能使用它的方法来获取 Word 文档中的文本信息。目前网上有许多提供这样的工具。

7.4.1 Jacob 的下载

Jacob 是 Java-COM Bridge 的缩写，它在 Java 与微软的 COM 组件之间构建一座桥梁。使用 Jacob 自带的 DLL 动态链接库，并通过 JNI 的方式实现了在 Java 平台上对 COM 程序的调用。Jacob 下载的地址为：
http://sourceforge.net/project/showfiles.php?group_id=109543&package_id=118368。本书采用的是 jacob_1.11_zip。解压下载的 Jacob_1.11_zip 文件后，如图 7-17 所示。

图 7-17 Jacob 包解压后的内容

7.4.2 在 Eclipse 中配置

（1）将 jacob.jar 导入工程的 Build Path，然后确认自己机器的 CPU 类型（X86 或 AMD64），并选择不同目录下的 jacob.dll 文件。
（2）将 jacob.dll 放到%JAVA_HOME%\jre\bin 目录下，其中，%JAVA_HOME%就是 JDK 的安装目录。注意这个的 jre 目录必须是 Eclipse 当前正在使用的目录，在 Eclipse 中选择“window->Preferences”菜单，在弹出的对话框中选择“Java->Installed JREs”项，如图 7-18 所示。

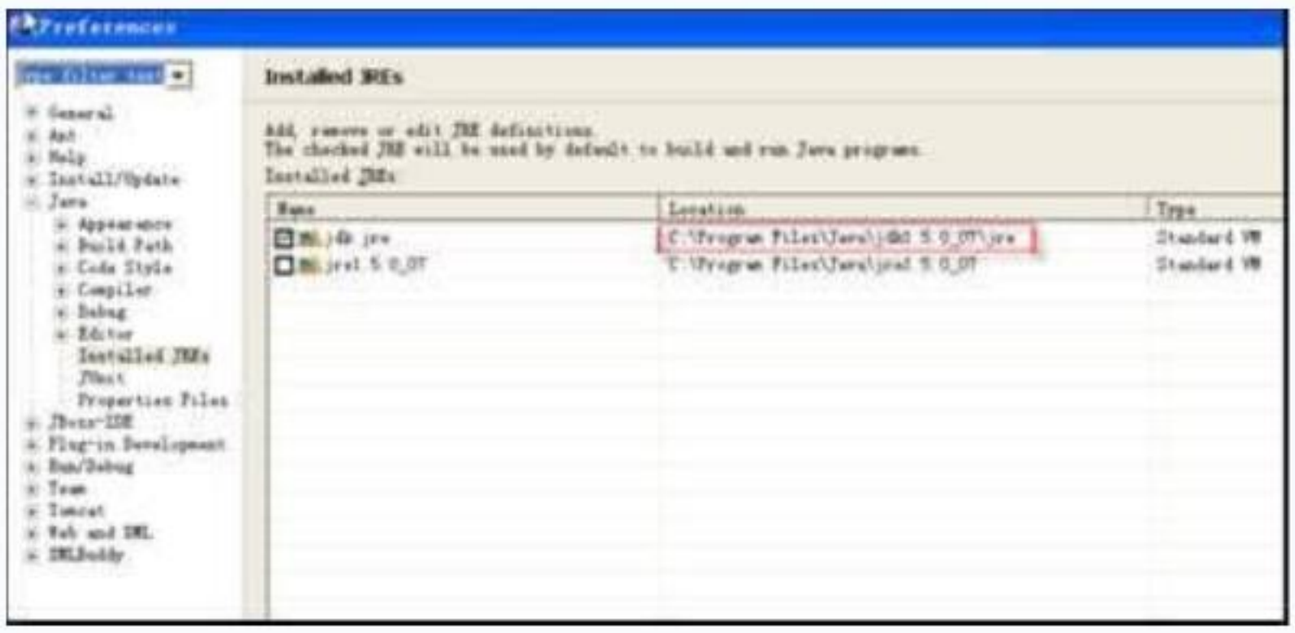

图 7-18 Eclipse 中 JRE 设置的对话框

（3）当前选择的 JRE 是“C:\Program Files\Java\jdk1.5.0_07\jre”目录下的，所以需要把 jacob.dll 复制到“C:\Program Files\Java\jdk1.5.0_07\jre\bin”目录下面。

（4）在工程中新建一个 ch7.jacob 包，并在包中创建 WordReader 类。该类将提供一个静态的 extractDoc()方法。它接收两个参数，一个是要处理的 DOC 文件名，另一个则是输出的文件名，然后通过 JNI 调用 Word 的 API 转换内容，该函数的代码如下。

代码 7.10

```
public static void extractDoc(String inputFIle, String outputFile) {
boolean flag = false;


// 打开 Word 应用程序
ActiveXComponent app = new ActiveXComponent("Word.Application");
try {
    // 设置 word 不可见
    app.setProperty("Visible", new Variant(false));
    // 打开 word 文件
    Dispatch doc1 = app.getProperty("Documents").toDispatch();
Dispatch doc2 = Dispatch.invoke(
        doc1,
        "Open",
        Dispatch.Method,
        new Object[] { inputFIle, new Variant(false),
```

```
                new Variant(true) }, new int[1]).toDispatch();
// 作为 txt 格式保存到临时文件
Dispatch.invoke(doc2, "SaveAs", Dispatch.Method, new Object[] {
        outputFile, new Variant(7) }, new int[1]);
// 关闭 word
Variant f = new Variant(false);
    Dispatch.call(doc2, "Close", f);
    flag = true;
} catch (Exception e) {
    e.printStackTrace();
} finally {
    app.invoke("Quit", new Variant[] {});
}
if (flag == true) {
    System.out.println("Transformed Successfully");
    } else {
        System.out.println("Transform Failed");
    }
}
```

（5）创建一个 main 函数来测试 WordReader 类，该 main 函数代码如下。

```
public static void main(String[] args) {
        WordReader.extractDoc("c:/test.doc","c:/jacob.txt");
}
```

（6）新生成的 txt 文件被保存到 c:\jacob.txt 下，如图 7-19 所示。

图 7-19　使用 Jacob 处理的效果

在使用 Jacob 时，很重要的一点是，**用户本地系统中必须安装有 Word 的应用程序**。否则也就无法建立 Java-COM 桥，进而无法解析了。

7.5 小结

本章向读者详细介绍了 Word、Excel 和 PDF 文件的文本提取工具。有关这些文本提取工具的使用问题是任何一个 Lucene 论坛上都会被提出的问题。不过，一直没有任何一篇资料把这些工具集合在一起进行详细的讲解。希望本篇的内容能够对读者有所帮助，以解决大家在遇到这些格式的文件时，能提取出想要的文本。

POI 使用初步

POI 提供给用户使用的对象在 org.apache.poi.hssf.usermodel 包中，主要部分包括 Excel 对象、样式和格式，还有辅助操作等。

最主要的几个对象如表 3.1 所示：

表 3.1 POI 主要对象

POI 对象名称	所对应的 Excel 对象
HSSFWorkbook	工作簿
HSSFSheet	工作表
HSSFRow	行
HSSFCell	单元格

下面我们来看如下的例子，使用表 3.1 中的对象在程序的当前目录下创建一个 Excel 文件 test.xls，在第一个单元格中写入内容，然后读出第一个单元格的内容。

完整的程序如下：

```
import org.apache.poi.hssf.usermodel.HSSFWorkbook;
import org.apache.poi.hssf.usermodel.HSSFSheet;
```

```
import org.apache.poi.hssf.usermodel.HSSFRow;
import org.apache.poi.hssf.usermodel.HSSFCell;
import java.io.FileOutputStream;
import java.io.FileInputStream;

public class CreateXL
{
  public static String xlsFile="test.xls"; //产生的Excel 文件的名称
  public static void main(String args[])
  {
    try
    {
      HSSFWorkbook workbook = new HSSFWorkbook(); //产生工作簿对象
      HSSFSheet sheet = workbook.createSheet(); //产生工作表对象
      //设置第一个工作表的名称为 firstSheet
      //为了工作表能支持中文，设置字符编码为 UTF_16
      workbook.setSheetName(0,"firstSheet",HSSFWorkbook.ENCODING_UTF_16);
      //产生一行
      HSSFRow row = sheet.createRow((short)0);
      //产生第一个单元格
      HSSFCell cell = row.createCell((short) 0);
      //设置单元格内容为字符串型
      cell.setCellType(HSSFCell.CELL_TYPE_STRING);
      //为了能在单元格中写入中文，设置字符编码为 UTF_16。
      cell.setEncoding(HSSFCell.ENCODING_UTF_16);
      //往第一个单元格中写入信息
      cell.setCellValue("测试成功");
      FileOutputStream fOut = new FileOutputStream(xlsFile);
      workbook.write(fOut);
      fOut.flush();
      fOut.close();
      System.out.println("文件生成...");
      //以下语句读取生成的 Excel 文件内容
      FileInputStream fIn=new FileInputStream(xlsFile);
      HSSFWorkbook readWorkBook= new HSSFWorkbook(fIn);
      HSSFSheet readSheet= readWorkBook.getSheet("firstSheet");
      HSSFRow readRow =readSheet.getRow(0);
      HSSFCell readCell = readRow.getCell((short)0);
      System.out.println("第一个单元是：" + readCell.getStringCellValue());
    }
    catch(Exception e)
    {
      System.out.println(e);
    }
  }
}
```

与数据库结合使用

使用 POI，结合 JDBC 编程技术，我们就可以方便地将数据库中的数据导出生成 Excel 报表。其关键代码如下：

```
/*把数据集rs中的数据导出至Excel工作表中。
*传入参数：数据集rs，Excel文件名称xlsName，工作表名称sheetName。
*/

public static void resultSetToExcel(ResultSet rs,String xlsName,String sheetName)
throws Exception
{
  HSSFWorkbook workbook = new HSSFWorkbook();
  HSSFSheet sheet = workbook.createSheet();
  workbook.setSheetName(0,sheetName,HSSFWorkbook.ENCODING_UTF_16);
  HSSFRow row= sheet.createRow((short)0);;
  HSSFCell cell;
  ResultSetMetaData md=rs.getMetaData();
  int nColumn=md.getColumnCount();
  //写入各个字段的名称
  for(int i=1;i<=nColumn;i++)
  {
    cell = row.createCell((short)(i-1));
    cell.setCellType(HSSFCell.CELL_TYPE_STRING);
    cell.setEncoding(HSSFCell.ENCODING_UTF_16);
    cell.setCellValue(md.getColumnLabel(i));
  }

  int iRow=1;
  //写入各条记录，每条记录对应Excel中的一行
  while(rs.next())
  {row= sheet.createRow((short)iRow);;
    for(int j=1;j<=nColumn;j++)
    {
      cell = row.createCell((short)(j-1));
      cell.setCellType(HSSFCell.CELL_TYPE_STRING);
      cell.setEncoding(HSSFCell.ENCODING_UTF_16);
      cell.setCellValue(rs.getObject(j).toString());
    }
    iRow++;
  }
  FileOutputStream fOut = new FileOutputStream(xlsName);
  workbook.write(fOut);
  fOut.flush();
  fOut.close();
  JOptionPane.showMessageDialog(null,"导出数据成功！");
}
```

结　　束　　语

POI功能强大，还可以设置单元格格式、设置页眉页脚等。限于篇幅的关系就不一一举例了，感兴趣的读者可以参考其帮助文档（在图2.1的doc文件夹中）。总之，使用POI，我们可以较好地解决Java编程中的Excel报表问题，进一步满足用户的需求

POI------HWPF

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileWriter;
import org.apache.poi.hwpf.HWPFDocument;
import org.apache.poi.hwpf.usermodel.Range;

/**
 * @author JUSTIN
 * TODO 要更改此生成的类型注释的模板，请转至
 * 窗口 — 首选项 — Java — 代码样式 — 代码模板
 */
public class WordToText {
    String origFileName;
    String tempFile;
    HWPFDocument wd;
public WordToText(String origFileName,String tempFile)
    {
        this.tempFile = tempFile;
        this.origFileName = origFileName;
    }
    public void getText() {
        try {
wd = new HWPFDocument(new FileInputStream(origFileName));
            Range r = wd.getRange();
            String str = r.text();
            saveFile(str);
        } catch (Exception eN) {
System.out.println("Error reading document:" + origFileName + "\n"
                + eN.toString());
            eN.printStackTrace();
        }
    }

public void saveFile(String saveStr) {
```

```
                boolean error = false;
                    try {
                    File saveFile;
        FileWriter writer = new FileWriter(tempFile);
                int saveStrLen = saveStr.length();
    for (int i = 0; i < saveStrLen; i++)
                    writer.write((int) saveStr.charAt(i));
                                writer.close();
                } catch (Exception eF) {
                                eF.printStackTrace();
                } // end for catch
                        }
}
```