

## 1 认识 NPOI

本章将介绍 NPOI 的一些基本信息，包括以下几个部分

- 什么是 NPOI
- 版权说明
- 相关资源
- 团队介绍
- 未来展望
- 各 Assembly 的作用

### 1.1 什么是 NPOI

NPOI，顾名思义，就是 POI 的 .NET 版本。那 POI 又是什么呢？POI 是一套用 Java 写成的库，能够帮助开发者在没有安装微软 Office 的情况下读写 Office 97-2003 的文件，支持的文件格式包括 xls, doc, ppt 等。在本文发布时，POI 的最新版本是 3.5 beta 6。

NPOI 1.x 是基于 POI 3.x 版本开发的，与 poi 3.2 对应的版本是 NPOI 1.2，目前最新发布的版本是 1.2.1，在该版本中仅支持读写 Excel 文件和 Drawing 格式，其他文件格式将在以后的版本中得到支持。

### 1.2 版权说明

NPOI 采用的是 Apache 2.0 许可证（poi 也是采用这个许可证），这意味着它可以被用于任何商业或非商业项目，你不用担心因为使用它而必须开放你自己的源代码，所以它对于很多从事业务系统开发的公司来说绝对是很不错的选择。

当然作为一个开源许可证，肯定也是有一些义务的，例如如果你在系统中使用 NPOI，你必须保留 NPOI 中的所有声明信息。对于源代码的任何修改，必须做出明确的标识。

完整的 apache 2.0 许可证请见 <http://www.phpx.com/man/Apache-2/license.html>

### 1.3 相关资源

官方网站: <http://npoi.codeplex.com/>

POIFS Browser 1.2 下载地址:

<http://npoi.codeplex.com/Release/ProjectReleases.aspx?ReleaseId=24305>

QQ 交流群: 78142590

## 1.4 团队介绍

Tony Qu 来自于中国上海,是这个项目的发起人和开发人员,时区是 GMT+8,2008 年 9 月开始了 NPOI 的开发,负责 NPOI 所有底层库的开发、测试和 bug 修复。个人 blog 地址为 <http://tonyqus.cnblogs.com/>

Hüseyin Tüfekçilerli 来自于土耳其的伊斯坦布尔,也是这个项目的开发人员,时区是 GMT+2,2008 年 11 月参与了 NPOI 的开发,主要负责 POIFS Browser 1.0 的开发工作。个人 blog 地址为 <http://huseyint.com/>

aTao.Xiang, 来自中国,2009 年 8 月开始参与该项目,主要参与了 NPOI 1.2 中文版的撰写工作和推广工作。个人 blog 地址为 <http://www.cnblogs.com/atao/>

## 1.5 回顾与展望

目前 POI 版本中的 HWPf (用于 Word 的读写库) 还不是很稳定,并非正式发布版本,且负责 HWPf 的关键开发人员已经离开,所以 NPOI 可能考虑自己重新开发 HWPf。另外,目前微软正在开发 Open XML Format SDK,NPOI 可能会放弃对 ooxml 的支持,当然这取决于用户的需求和 Open XML Format SDK 的稳定性和速度。从目前而言,NPOI 有几大优势:

第一,完全基于 .NET 2.0,而非 .NET 3.0/3.5。

第二,读写速度快(有个国外的兄弟回复说,他原来用 ExcelPackage 生成用了 4-5 个小时,现在只需要 4-5 分钟)

第三,稳定性好(相对于用 Office OIA 而言,毕竟那东西是基于 Automation 做的,在 Server 上跑个 Automation 的东西,想想都觉得可怕),跑过了将近 1000 个测试用例(来自于 POI 的 testcase 目录)

第四,API 简单易用,当然这得感谢 POI 的设计师们

第五,完美支持 Excel 2003 格式(据说 myxls 无法正确读取 xls 模板,但 NPOI 可以),以后也许是所有 Office 2003 格式

希望 NPOI 把这些优势继续发扬下去,这样 NPOI 才会更有竞争力。

## 1.6 Assembly 的作用

NPOI 目前有好几个 assembly,每个的作用各有不同,开发人员可以按需加载相应的 assembly。在这里大概罗列一下:

NPOI.Util: 基础辅助库

NPOI.POIFS: OLE2 格式读写库

NPOI.DDF: Microsoft Drawing 格式读写库

NPOI.SS: Excel 公式计算库

NPOI.HPSF: OLE2 的 Summary Information 和 Document Summary Information 属性读写库

NPOI.HSSF: Excel BIFF 格式读写库

## 2.1 创建基本内容

### 2.1.1 创建 Workbook 和 Sheet

创建 Workbook 说白了就是创建一个 Excel 文件,当然在 NPOI 中更准确的表示是在内存中创建一个 Workbook 对象流。

本节作为第 2 章的开篇文章,将做较为详细的讲解,以帮助 NPOI 的学习者更好的理解 NPOI 的组成和使用。

NPOI.HSSF 是专门负责 Excel BIFF 格式的命名空间,供开发者使用的对象主要位于 NPOI.HSSF.UserModel 和 NPOI.HSSF.Util 命名空间下,下面我们要讲到的 Workbook 的创建用的就是 NPOI.HSSF.UserModel.HSSFWorkbook 类,这个类负责创建 .xls 文档。

在开始创建 Workbook 之前,我们先要在项目中引用一些必要的 NPOI assembly,如下所示:

```
NPOI.dll
```

```
NPOI.POIFS.dll
```

```
NPOI.HSSF.dll
```

```
NPOI.Util.dll
```

要创建一个新的 xls 文件其实很简单,只要我们初始化一个新的 HSSFWorkbook 实例就行了,如下所示:

```
using NPOI.HSSF.UserModel;  
...  
HSSFWorkbook hssfworkbook = new HSSFWorkbook();
```

是不是很方便啊,没有任何参数或设置,但这么创建有一些限制,这样创建出来的 Workbook 在 Excel 中打开是会报错的,因为 Excel 规定一个 Workbook 必须至少带 1 个 Sheet,这也是为什么在 Excel 界面中,新建一个 Workbook 默认都会新建 3 个 Sheet。所以必须加入下面的创建 Sheet 的代码才能保证生成的文件正常:

```
HSSFSheet sheet = hssfworkbook.CreateSheet("new sheet");
```

如果要创建标准的 Excel 文件，即拥有 3 个 Sheet，可以用下面的代码：

```
hssfworkbook.CreateSheet("Sheet1");  
hssfworkbook.CreateSheet("Sheet2");  
hssfworkbook.CreateSheet("Sheet3");
```

最后就是把这个 HSSFWorkbook 实例写入文件了，代码也很简单，如下所示：

```
FileStream file = new FileStream(@"test.xls", FileMode.Create);  
hssfworkbook.Write(file);  
file.Close();
```

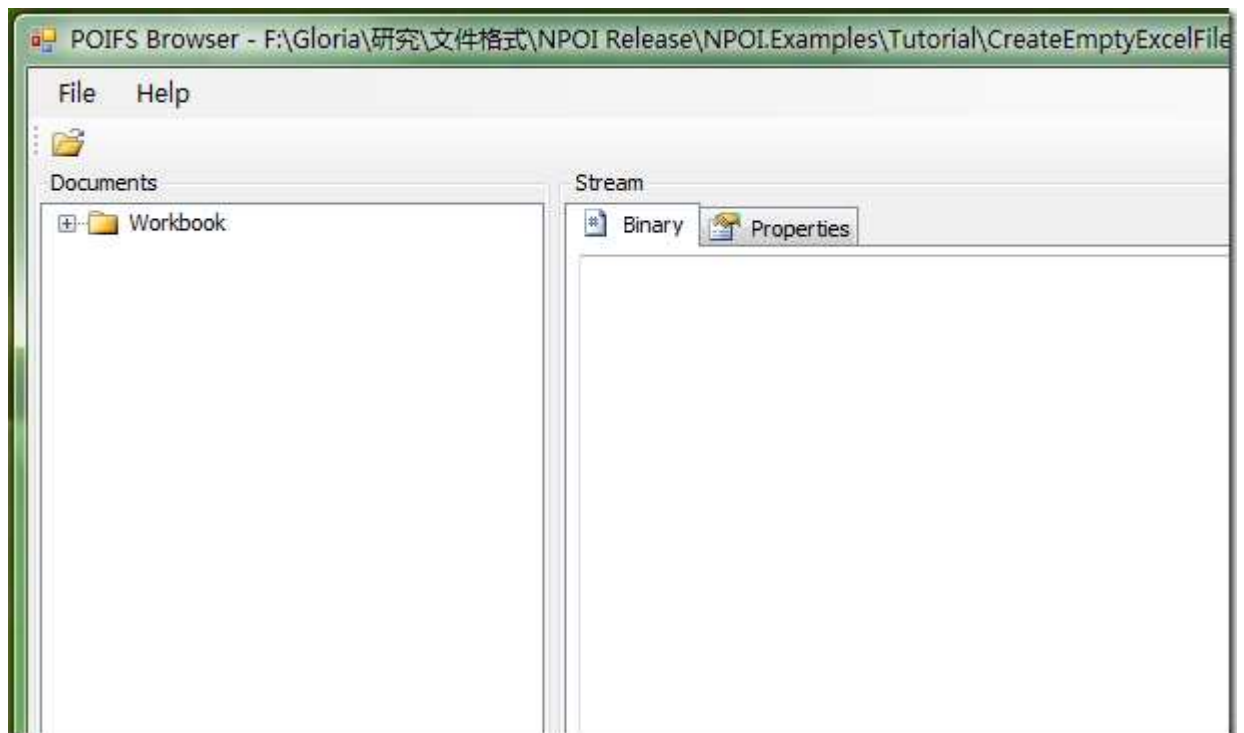
这里假设文件名是 test.xls，，在创建完 FileStream 之后，直接调用 HSSFWorkbook 类的 Write 方法就可以了。

最后你可以打开 test.xls 文件确认一下，是不是有 3 个空的 Sheet。

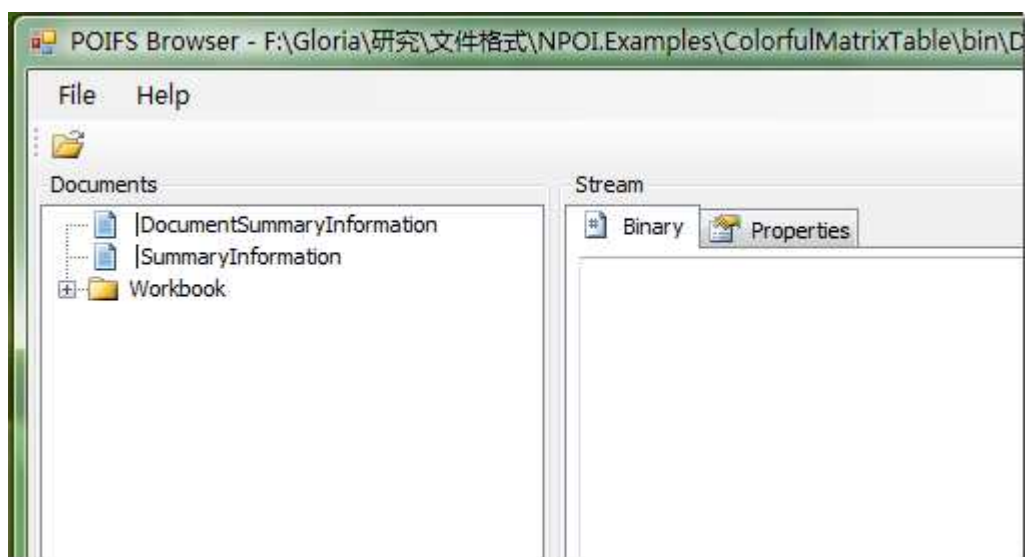
相关范例请见 [NPOI 1.2 正式版](#)中的 CreateEmptyExcelFile 项目。

## 2.1.2 创建 DocumentSummaryInformation 和 SummaryInformation

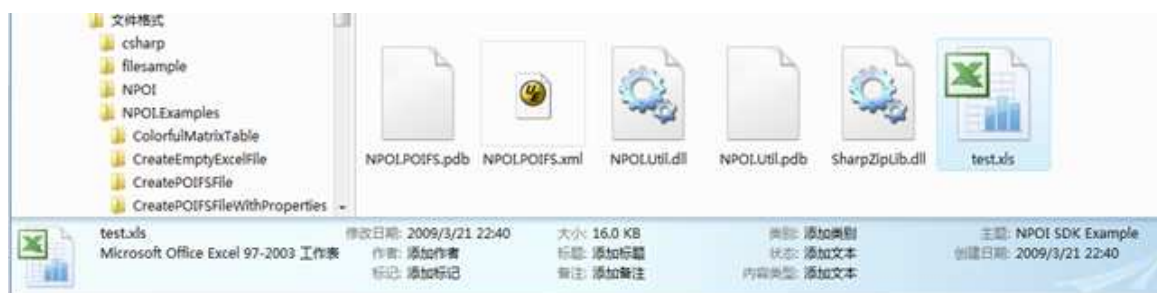
前一节中我们讲解了如何创建一个新的 Workbook，但在此过程中大家也许会发现一个细节，这些文件没有包括 DocumentSummaryInformation 和 SummaryInformation 头。如果你还不是很清楚我在说什么，可以看 POIFS Browser 打开 test.xls 文件后的截图：



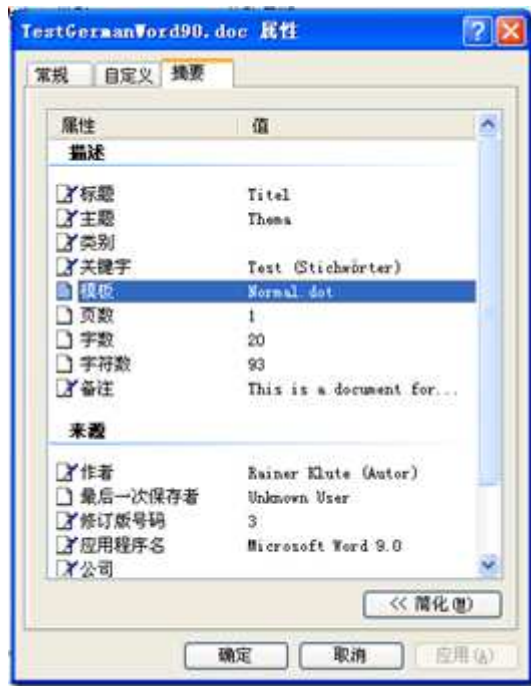
你会发现只有 Workbook 目录，其他什么都没有，但事实上一个正常的 xls 文件，比如说 Excel 生成的 xls 文件是类似下面的结构：



是不是多出来 DocumentSummaryInformation 和 SummaryInformation 两个头？很多人可能对 DocumentSummaryInformation 和 SummaryInformation 很陌生，可能第一次听说这玩意，没事，这很正常，因为普通用户很少会去使用这些东西，但它们其实比想象中有用。



请看上图中的信息，如作者、标题、标记、备注、主题等信息，其实这些信息都是存储在 DocumentSummaryInformation 和 SummaryInformation 里面的，这么一说我想大家应该明白了吧，这些信息是为了快速提取文件信息准备。在 Windows XP 中，也有对应的查看和修改界面，只是没有 Vista 这么方便，如下所示：



这恐怕也是很多人对于这些信息漠不关心的原因吧，因为没有人愿意通过 右击文件->属性 这样复杂的操作去查看一些摘要信息。

### 提示

DocumentSummaryInformation 和 SummaryInformation 并不是 Office 文件的专利，只要是 OLE2 格式，都可以拥有这两个头信息，主要目的就是为了在没有完整读取文件数据的情况下获得文件的摘要信息，同时也可用作桌面搜索的依据。要了解

DocumentSummaryInformation 的全部属性请见

[http://msdn.microsoft.com/en-us/library/aa380374\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/aa380374(VS.85).aspx); 要了解

SummaryInformation 的全部属性请见

[http://msdn.microsoft.com/en-us/library/aa369794\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/aa369794(VS.85).aspx)。

好了，说到这里，我想大家对于接下来我们要创建的内容有了初步的认识，下面我们就马上动手创建。

首先引用以下这些命名空间：

```
using NPOI.HSSF.UserModel;  
using NPOI.HPSF;  
using NPOI.POIFS.FileSystem;
```

其中与 DocumentSummaryInformation 和 SummaryInformation 密切相关的是 HPSF 命名空间。

首先创建 Workbook

```
HSSFWorkbook hssfworkbook = new HSSFWorkbook();
```

然后创建 DocumentSummaryInformation

```
DocumentSummaryInformation dsi =  
PropertySetFactory.CreateDocumentSummaryInformation();  
dsi.Company = "NPOI Team";
```

再创建 SummaryInformation

```
SummaryInformation si = PropertySetFactory.CreateSummaryInformation();  
si.Subject = "NPOI SDK Example";
```

因为是范例，这里仅各设置了一个属性，其他都没有设置。

现在我们把创建好的对象赋给 Workbook，这样才能保证这些信息被写入文件。

```
hssfworkbook.DocumentSummaryInformation = dsi;  
hssfworkbook.SummaryInformation = si;
```

最后和 2.1.1 节一样，我们把 Workbook 通过 FileStream 写入文件。

相关范例请见 [NPOI 1.2 正式版](#)中的 CreatePOIFSFileWithProperties

### 2.1.3 创建单元格

用过 Excel 的人都知道，单元格是 Excel 最有意义的东西，我们做任何操作恐怕都要和单元格打交道。在 Excel 中我们要添加一个单元格只需要点击任何一个单元格，然后输入内容就是了，但是 Excel 底层其实没有这么简单，不同的单元格是有不同的类型的，比如说数值单元格是用 NumberRecord 表示，文本单元格是用 LabelSSTRecord 表示，空单元格是用 BlankRecord 表示。这也就意味着，在设置单元格时，你必须告诉 NPOI 你需要创建哪种类型的单元格。

要创建单元格首先要创建单元格所在的行，比如，下面的代码创建了第 0 行：

```
HSSFSheet sheet1 = hssfworkbook.CreateSheet("Sheet1");  
HSSFRow row1=sheet1.CreateRow(0);
```

行建好了，就可以建单元格了，比如创建 A1 位置的单元格：

```
row1.CreateCell(0).SetCellValue(1);
```

这里要说明一下，SetCellValue 有好几种重载，你可以设置单元格为 bool、double、DateTime、string 和 HSSFRichTextString 类型。其中对于 string 类型的重载调用的就是 HSSFRichTextString 类型的重载，所以是一样的，HSSFRichTextString 可用于有字体或者 Unicode 的文本。

如果你觉得每一行要声明一个 HSSFRow 很麻烦，可以用下面的方式：

```
sheet1.CreateRow(0).CreateCell(0).SetCellValue("This is a Sample");
```

这么用有个前提，那就是第 0 行还没创建过，否则得这么用：

```
sheet1.GetRow(0).CreateCell(0).SetCellValue("This is a Sample");
```

注意：这里的行在 Excel 里是从 1 开始的，但是 NPOI 内部是从 0 开始的；列在 Excel 里面是用字母表示的，而 NPOI 中也是用从 0 开始的数字表示的，所以要注意转换。

如果你要获得某一个已经创建的单元格对象，可以用下面的代码：

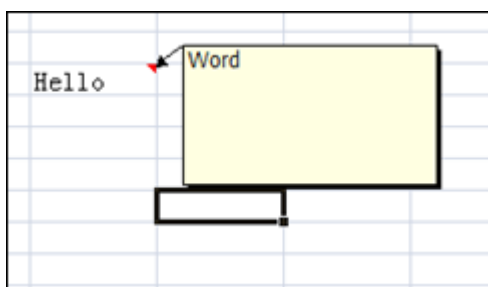
```
sheet1.GetRow(row_index).GetCell(column_index);
```

本节仅讲解最基本的单元格创建，有关单元格格式设置、样式等高级话题请见：2.2 节 单元格相关操作。

相关范例请见 [NPOI 1.2 正式版](#)中的 SetCellValuesInXls 项目。

#### 2.1.4 创建批注

很多人不怎么用 Excel 中的批注，所以我特地截了张图，让大家知道本节我们要创建的到底是什么东西。



在过去，我们恐怕没有办法实现这一功能，因为无论是 cvs 法、html 法、oledb 法都没有提供这样的接口，当然 Office PIA 法可以做到，但是性能实在太差，而且稳定性不好，经常莫名其妙 crash（这是某某兄弟给我的反馈，我引用了下，呵呵）。在以后的教程中，你将看到更多在过去无法通过传统方法实现的东西，好戏才刚刚开始。

批注主要有三个属性需要设置，一个是批注的位置和大小、一个是批注的文本、还有一个是批注的作者。

批注的位置和大小，在 Excel 中是与单元格密切相关的，NPOI 中通过 HSSFClientAnchor 的实例来表示，它的构造函数比较复杂，有 8 个参数，它们分别是

参数	说明
dx1	第 1 个单元格中 x 轴的偏移量
dy1	第 1 个单元格中 y 轴的偏移量
dx2	第 2 个单元格中 x 轴的偏移量
dy2	第 2 个单元格中 y 轴的偏移量



col1	第 1 个单元格的列号
row1	第 1 个单元格的行号
col2	第 2 个单元格的列号
row2	第 2 个单元格的行号

例如，如果我们打算让注释显示在 B3 和 E5 之间，就应该这么写：

```
HSSFPatriarch patr = sheet.CreateDrawingPatriarch();
HSSFComment comment1 = patr.CreateComment(new HSSFClientAnchor(0, 0, 0, 0, 1, 2,
4, 4));
```

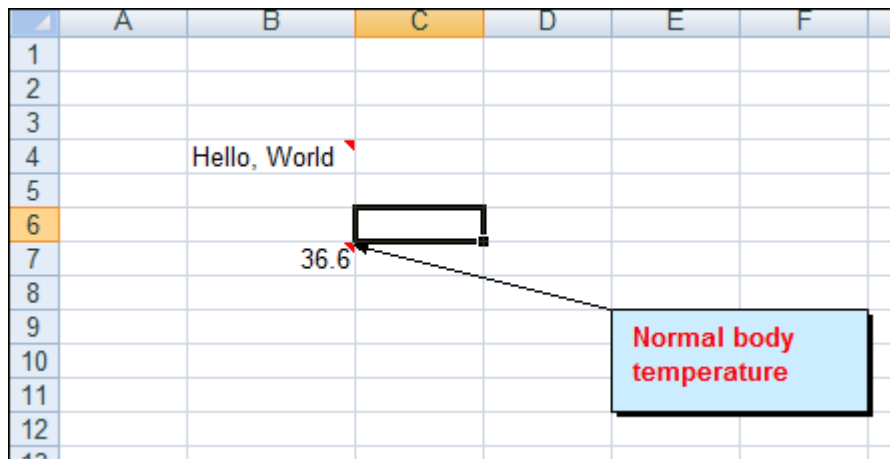
下面我们设置这个批注的内容和作者，这个比较简单：

```
comment1.String = new HSSFRichTextString("Hello World");
comment1.Author = "NPOI Team";
```

最后一步就是把批注赋给某个单元格：

```
HSSFCell cell = sheet.CreateRow(1).CreateCell(1);
cell.CellComment = comment1;
```

对于批注，你有两种选择，一种是隐藏（默认），一种是显示（即表单一打开就显示该批注），可以通过 `comment1.Visible` 属性来控制。

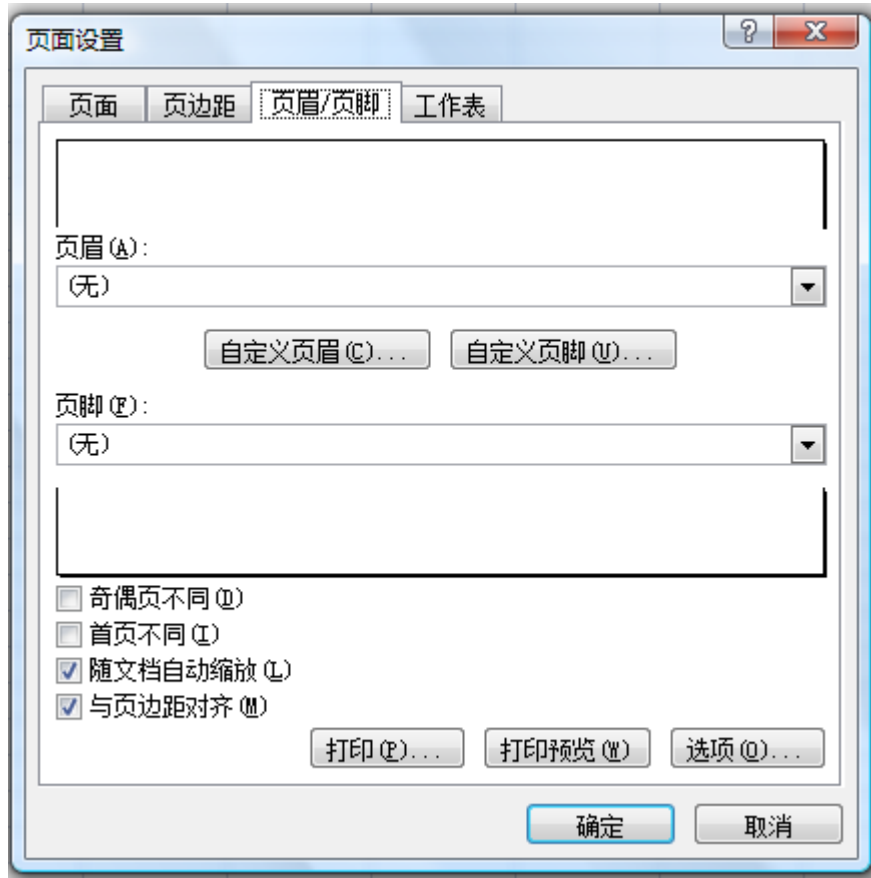


看了上面这张图大家就应该明白了，这里有 2 个批注，下面那个是显示的，上面那个是隐藏的。

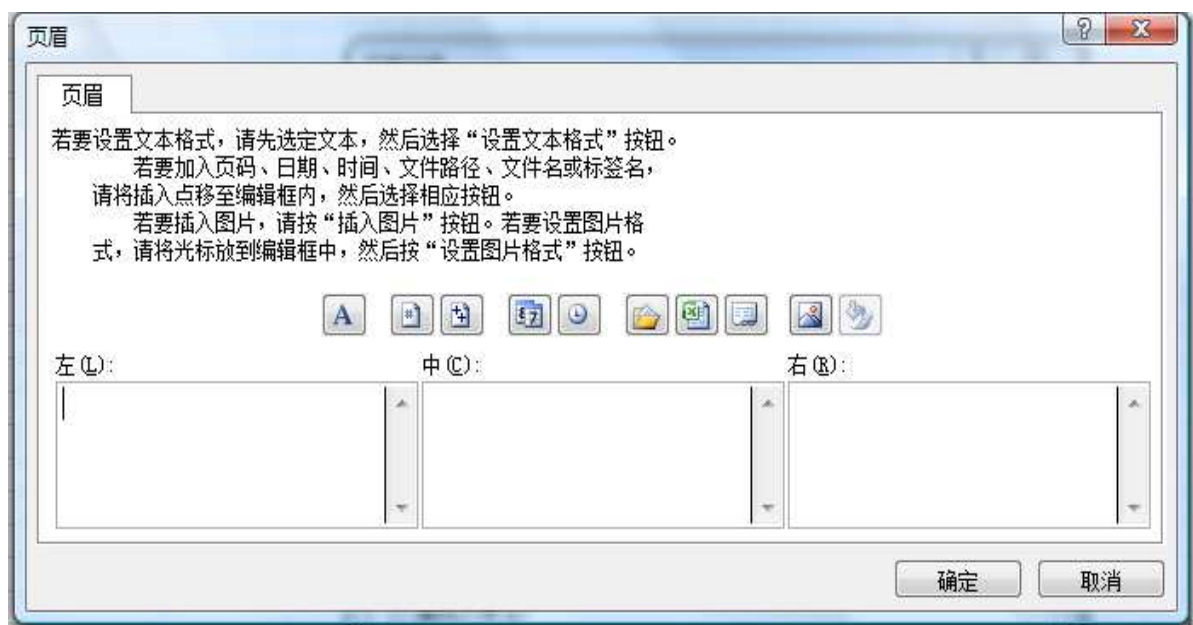
相关范例请见 [NPOI 1.2 正式版](#)中的 `SetCellCommentInXls`。

### 2.1.5 创建页眉和页脚

很多人并不知道 Excel 的页眉和页脚功能，因为在界面上是显示不了页眉和页脚的，必须在打印页面中才能看到，这也直接导致了其设置界面也显得更隐秘，你必须进入页面设置 -> 页眉和页脚才能设置。以下是 Office 2007 中的设置界面。



当你按“自定义页眉”或“自定义页脚”时，你会看到以下界面，Excel 把页眉、页脚分成了左中右三部分，这一点绝非单纯体现在界面上，在底层的存储中也是如此。如果你设置的是“左”的内容，底层的存储字符串就会在开头加上&L，如果是“右”的内容则会加上&R，所以 HeaderRecord 中的字符串看上去是这样的：“&C&LFooter A&R”，这个字符串的意思是仅设置了“左”的内容，内容是 Footer A。



看了这些我想你应该对页眉和页脚有所了解，回过头来说 NPOI，NPOI 中主要是靠 HSSFSheet.Header 和 HSSFSheet.Footer 来设置的，这两个属性分别是 HSSFHeader 和 HSSFFooter 类型的。

参考代码如下：

```
HSSFSheet s1 = hssfworkbook.CreateSheet("Sheet1");
s1.CreateRow(0).CreateCell(1).SetCellValue(123);

//set header text
s1.Header.Center = "This is a test sheet";
//set footer text
s1.Footer.Left = "Copyright NPOI Team";
s1.Footer.Right = "created by Tony Qu (鬲杰)";
```

以上代码中我添加了页眉的 Center 内容，Footer 的 Left 和 Right 内容，在打印预览中看到的 effect 大概是这样的：

页眉

This is a test sheet

页脚

Copyright@ NPOI Team

created by Tony Qu (鬲杰)

至于一些 Excel 特殊字符，比如说页码可以用 &P，当前日期可以用 &D，其他的东西你自己研究吧。

本范例完整代码请见 NPOI.Examples 中的 CreateHeaderFooterInXls 项目。

## 2.2 单元格操作

### 2.2.1 设置单元格格式

在 Excel 中我们经常要设置格式，比如说日期格式 (yyyyymmdd)、小数点格式 (1.20)、货币格式 (\$2000)、百分比格式 (99.99%) 等等，这些东西在过去我们恐怕只能在服务器端生成好，不但增加了服务器端的代码量，还造成了不必要的字符串替换操作，如今 NPOI 将让服务器从这种完全没有必要的操作中解放出来，一切都将由 Excel 在客户端处理。

使用 NPOI 时要注意，所有的格式都是通过 `CellStyle.DataFormat` 赋给单元格的，而不是直接赋给单元格。

#### 案例一 日期格式

假设我们现在需要显示的日期的格式为 2008 年 5 月 5 日，可以用下面的代码生成：

```
HSSFSheet sheet = hssfworkbook.CreateSheet("new sheet");
HSSFCell cell = sheet.CreateRow(0).CreateCell(0);
cell.SetCellValue(new DateTime(2008, 5, 5));
//set date format
HSSFCellStyle cellStyle = hssfworkbook.CreateCellStyle();
HSSFDataFormat format = hssfworkbook.CreateDataFormat();
cellStyle.DataFormat = format.GetFormat("yyyy 年 m 月 d 日");
cell.CellStyle=cellStyle;
```

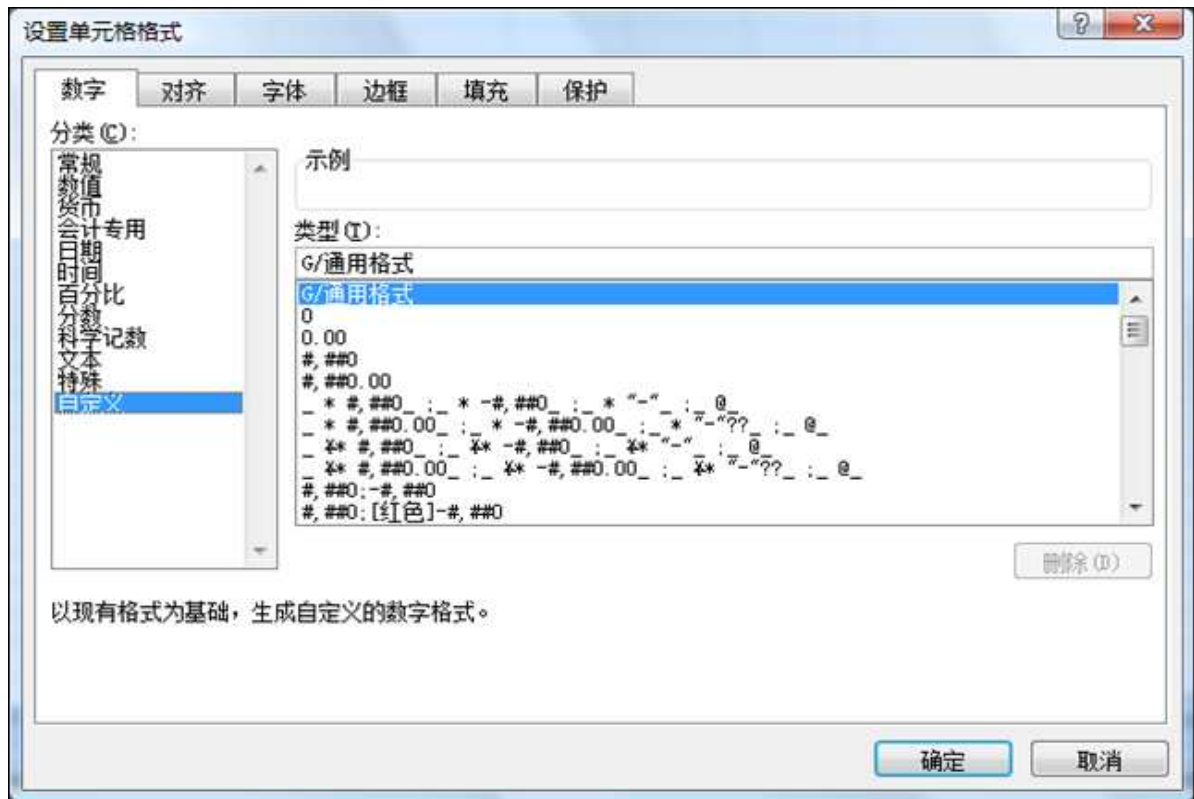
由于这里的“yyyy 年 m 月 d 日”属于自定义格式（区别于 Excel 内嵌的格式），所以必须用 `hssfworkbook.CreateDataFormat()` 创建一个 `HSSFDataFormat` 实例，然后使用 `format.GetFormat` 来获取相应的格式，只要是 Excel 支持的格式表示方式，这种方式都能够实现。

#### 案例二 保留 2 位小数

假设我们有个单元格的值为 1.2，怎么显示成 1.20 呢？在 Excel 中可以用“0.00”来表示，所以下面的代码就能完成：

```
// Create a row and put some cells in it. Rows are 0 based.
HSSFCell cell = sheet.CreateRow(0).CreateCell(0);
//set value for the cell
cell.SetCellValue(1.2);
//number format with 2 digits after the decimal point - "1.20"
HSSFCellStyle cellStyle = hssfworkbook.CreateCellStyle();
cellStyle.DataFormat = HSSFDataFormat.GetBuiltinFormat("0.00");
cell.CellStyle = cellStyle;
```

这里与上面有所不同，用的是 `HSSFDataFormat.GetBuiltinFormat()` 方法，之所以用这个，是因为 0.00 是 Excel 内嵌的格式，完整的 Excel 内嵌格式列表大家可以看这个窗口中的自定义列表：



这里就不一一列出了。

### 案例三 货币格式

货币格式在金融的项目中经常用到，比如说人民币符号¥，美元符号\$等，这里可以用下面的代码表示：

```
HSSFCell cell12 = sheet.CreateRow(1).CreateCell(0);
cell12.SetCellValue(20000);
HSSFCellStyle cellStyle2 = hssfworkbook.CreateCellStyle();
HSSFFormat format = hssfworkbook.CreateDataFormat();
cellStyle2.DataFormat = format.GetFormat("¥#, ##0");
cell12.CellStyle = cellStyle2;
```

注意，这里还加入了千分位分隔符，所以是#, ##，至于为什么这么写，你得去问微软，呵呵。

### 案例四 百分比

百分比在报表中也很常用，其实基本上和上面一样，只是格式表示是 0.00%，代码如下：

```
cellStyle4.DataFormat = HSSFFormat.GetBuiltinFormat("0.00%");
```

由于这里是内嵌格式，所以直接用 HSSFFormat.GetBuiltinFormat 即可。

### 案例五 中文大写

在表示金额时,我们时常会用到,我也见过不少兄弟实现了数字转中文大小写的工具类,以后你可以尝试让 Excel 去处理这一切,代码和刚才差不多,也是改格式的表达:

```
HSSFDataFormat format = hssfworkbook.CreateDataFormat();
cellStyle6.DataFormat = format.GetFormat("[DbNum2][¥-804]0");
```

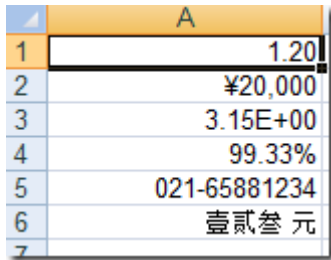
由于是自定义格式,所以用了 HSSFDataFormat.GetFormat,相信你对这两种获取格式的形式区别越来越熟悉了。

#### 案例六 科学计数法

这东西数学课上我们都学过,虽然用的不多,但是既然 Excel 支持,这里也提一下:

```
cellStyle3.DataFormat = HSSFDataFormat.GetBuiltinFormat("0.00E+00");
```

下面展示下以上这些例子的显示效果:



	A
1	1.20
2	¥20,000
3	3.15E+00
4	99.33%
5	021-65881234
6	壹贰叁元
7	

最后总结一下 HSSFDataFormat.GetFormat 和 HSSFDataFormat.GetBuiltinFormat 的区别:

当使用 Excel 内嵌的(或者说预定义)的格式时,直接用 HSSFDataFormat.GetBuiltinFormat 静态方法即可。

当使用自己定义的格式时,必须先调用 HSSFWorkbook.CreateDataFormat(),因为这时在底层会先找有没有匹配的内嵌 FormatRecord,如果没有就会新建一个 FormatRecord,所以必须先调用这个方法,然后你就可以用获得的 HSSFDataFormat 实例的 GetFormat 方法了,当然相对而言这种方式比较麻烦,所以内嵌格式还是用 HSSFDataFormat.GetBuiltinFormat 静态方法更加直接一些。不过自定义的格式也不是天马行空随便定义,还是要参照 Excel 的格式表示来定义,具体请看相关的 Excel 教程。

**注意:** 自定义的 FormatRecord 是嵌入 xls 文件内部的,所以不用担心对方 Excel 中有没有定义过这种格式,都是能够正常使用的。

相关范例请参考 [NPOI 1.2 正式版](#)中的 NumberFormatInXls 项目。

### 2.2.2 单元格合并

合并单元格在制作表格时很有用,比如说表格的标题就经常是把第一行的单元格合并居中。那么在 NPOI 中应该如何实现单元格的合并呢?

为了实现这一功能，NPOI 引入了新的概念，即 Region，因为合并单元格，其实就是设定一个区域。下面说一下 Region 类的参数，Region 总共有 4 个参数，如下所示

Region 的参数	说明
FirstRow	区域中第一个单元格的行号
FirstColumn	区域中第一个单元格的列号
LastRow	区域中最后一个单元格的行号
LastColumn	区域中最后一个单元格的列号

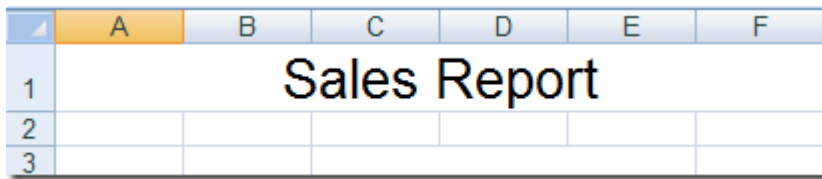
由于单元格的合并都是在表的基础上建立的，所以我们得先建 Sheet：

```
HSSFWorkbook hssfworkbook = new HSSFWorkbook();  
HSSFSheet sheet = hssfworkbook.CreateSheet("new sheet");
```

接下来我们根据实际场景来做一些演示。

#### 场景一 标题行的合并

这种场景是最常见的，比如说我们要建立一张销售情况表，英文叫 Sales Report



	A	B	C	D	E	F
1	Sales Report					
2						
3						

我们先设置居中和字体样式，这里我们采用 20 号字体，代码如下：

```
HSSFRow row = sheet.CreateRow(0);  
HSSFCell cell = row.CreateCell(0);  
cell.SetCellValue("Sales Report");  
HSSFCellStyle style = hssfworkbook.CreateCellStyle();  
style.Alignment = HSSFCellStyle.ALIGN_CENTER;  
HSSFFont font = hssfworkbook.CreateFont();  
font.FontHeight = 20*20;  
style.SetFont(font);  
cell.CellStyle = style;
```

要产生图中的效果，即把 A1:F1 这 6 个单元格合并，然后添加合并区域：

```
sheet.AddMergedRegion(new Region(0, 0, 0, 5));
```

#### 场景二 多行合并

看完场景一，你可不要认为多行合并就需要一行一行做，其实也只需要一行代码，比如说我们要把 C3:E5 合并为一个单元格，那么就可以用下面的代码：

2						
3						
4						
5						
6						

```
sheet.AddMergedRegion(new Region(2, 2, 4, 4));
```

提示：即使你没有用 `CreateRow` 和 `CreateCell` 创建过行或单元格，也完全可以直接创建区域然后把这一区域合并，Excel 的区域合并信息是单独存储的，和 `RowRecord`、`ColumnInfoRecord` 不存在直接关系。

相关范例请参考 [NPOI 1.2 正式版](#) 中的 `MergedCellInXls` 项目。

### 2.2.3 单元格对齐相关设置

本节将围绕“对齐”选项卡中的设置展开，虽然实际上你会发现该选项卡中的很多设置和对齐没有什么关系。合并单元格已经在 [2.2.2 节](#) 讲过了，这里就不提了。



首先我们用代码创建必要的单元格，代码如下：

```
HSSFWorkbook hssfworkbook = new HSSFWorkbook();
HSSFSheet sheet1 = hssfworkbook.CreateSheet("Sheet1");
HSSFRow row = sheet1.CreateRow(0);
row.CreateCell(0).SetCellValue("Test");
```

这里我们假设在 A0 单元格中加入了文本 Test。

请注意接下来我们要做的所有操作都是在 `CellStyle` 的基础上完成的，所以我们创建一个 `HSSFCellStyle`：



```
HSSFCellStyle style=hssfworkbook.CreateCellStyle();
```

### 水平对齐

这里用的是 `HSSFCellStyle.Alignment`，默认值自然是常规，即 `HSSFCellStyle.ALIGN_GENERAL`。

如果是左侧对齐就是 `style.Alignment = HSSFCellStyle.ALIGN_LEFT;`

如果是居中对齐就是 `style.Alignment = HSSFCellStyle.ALIGN_CENTER;`

如果是右侧对齐就是 `style.Alignment = HSSFCellStyle.ALIGN_RIGHT;`

如果是跨列举中就是 `style.Alignment = HSSFCellStyle.ALIGN_CENTER_SELECTION;`

如果是两端对齐就是 `style.Alignment = HSSFCellStyle.ALIGN_JUSTIFY;`

如果是填充就是 `style.Alignment = HSSFCellStyle.ALIGN_FILL;`

注意：以上选项仅当有足够的宽度时才能产生效果，不设置宽度恐怕看不出区别。

### 垂直对齐

这里用的是 `HSSFCellStyle.VerticalAlignment`，默认值为居中，即 `HSSFCellStyle.VERTICAL_CENTER`

如果是靠上就是 `style.VerticalAlignment=HSSFCellStyle.VERTICAL_TOP`

如果是居中就是 `style.VerticalAlignment=HSSFCellStyle.VERTICAL_CENTER`

如果是靠下就是 `style.VerticalAlignment=HSSFCellStyle.VERTICAL_BOTTOM`

如果是两端对齐就是 `style.VerticalAlignment=HSSFCellStyle.VERTICAL_JUSTIFY`

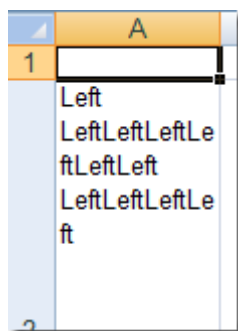
注意：以上选项仅当足够的高度时才能产生效果，不设置高度恐怕看不出区别。

### 自动换行

自动换行翻译成英文其实就是 `Wrap` 的意思，所以这里我们应该用 `WrapText` 属性，这是一个布尔属性

```
style.WrapText = true;
```

效果如下所示：



### 文本缩进



这是一个不太引人注意的选项，所以这里给张图出来，让大家知道是什么，缩进说白了就是文本前面的空白，我们同样可以用属性来设置，这个属性叫做 Indention。

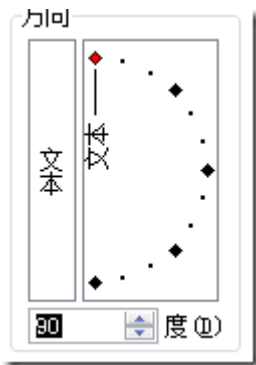
```
style.Indentation = 3;
```

### 文本旋转

文本方向大家一定在 Excel 中设置过，上图中就是调整界面，主要参数是度数，那么我们如何在 NPOI 中设置呢？

```
style.Rotation=(short)90;
```

以上代码是把单元格 A1 中的文本逆时针旋转 90 度，等同于下图中的设置：



请注意，这里的 Rotation 取值是从-90 到 90，而不是 0-180 度。

最后别忘了把样式变量 style 赋给 HSSFCellStyle.CellStyle，否则就前功尽弃了，呵呵！

以上的一些功能，比如文本旋转和自动换行，使用传统的 cvs 和 html 法恐怕是无法实现的。随着学习的不断深入，你将越来越意识到使用 NPOI 生成 Excel 其实如此简单。

相关范例请参考 [NPOI 1.2 正式版](#) 中的 SetAlignmentInXls 和 RotateTextInXls。














## 2.2.4 设置单元格边框

很多表格中都要使用边框，本节将为你重点讲解 NPOI 中边框的设置和使用。

边框和其他单元格设置一样也是在 HSSFCellStyle 上操作的，HSSFCellStyle 有 2 种和边框相关的属性，分别是：

边框相关属性	说明	范例
Border+方向	边框类型	BorderTop, BorderBottom, BorderLeft, BorderRight
方向+BorderColor	边框颜色	TopBorderColor, BottomBorderColor, LeftBorderColor,

其中边框类型分为以下几种：

边框范例图	对应的静态值
	HSSFCellStyle.BORDER_DOTTED
	HSSFCellStyle.BORDER_HAIR
	HSSFCellStyle.BORDER_DASH_DOT_DOT
	HSSFCellStyle.BORDER_DASH_DOT
	HSSFCellStyle.BORDER_DASHED
	HSSFCellStyle.BORDER_THIN
	HSSFCellStyle.BORDER_MEDIUM_DASH_DOT_DOT
	HSSFCellStyle.BORDER_SLANTED_DASH_DOT
	HSSFCellStyle.BORDER_MEDIUM_DASH_DOT
	HSSFCellStyle.BORDER_MEDIUM_DASHED
	HSSFCellStyle.BORDER_MEDIUM
	HSSFCellStyle.BORDER_THICK
	HSSFCellStyle.BORDER_DOUBLE

至于颜色那就很多了，全部在 HSSFColor 下面，如 HSSFColor.GREEN, HSSFColor.RED, 都是静态实例，可以直接引用。

下面我们假设我们要把一个单元格的四周边框都设置上，可以用下面的代码：

```
HSSFSheet sheet = hssfworkbook.CreateSheet("new sheet");

// Create a row and put some cells in it. Rows are 0 based.
HSSFRow row = sheet.CreateRow(1);
// Create a cell and put a value in it.
HSSFCell cell = row.CreateCell(1);
// Style the cell with borders all around.
HSSFCellStyle style = hssfworkbook.CreateCellStyle();
style.BorderBottom= HSSFCellStyle.BORDER_THIN;
style.BorderLeft= HSSFCellStyle.BORDER_THIN;
style.BorderRight= HSSFCellStyle.BORDER_THIN;
style.BorderTop = HSSFCellStyle.BORDER_THIN ;
cell.CellStyle= style;
```

这段代码使用了最普通的细边框，使得这个单元格看上去像块空心砖头。

	A	B
1		
2		
3		

注意：这里我们没有设置边框的颜色，但这不会影响最终的效果，因为 Excel 会用默认的颜色给边框上色。

如果要设置颜色的话，也很简单，如下：

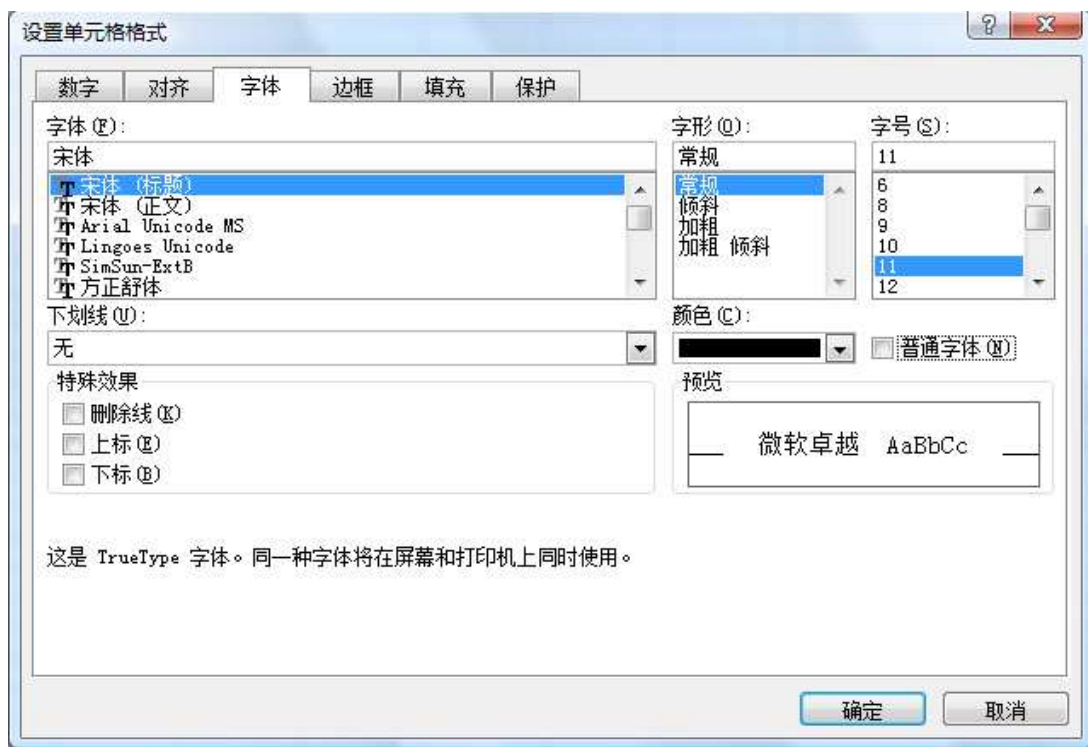
```
style.BottomBorderColor= HSSFColor.GREEN.index;
```

以上代码将底部边框设置为绿色，要注意，不是直接把 HSSFColor.GREEN 赋给 XXXXBorderColor 属性，而是把 index 的值赋给它。

相关范例请参考 [NPOI 1.2 正式版](#) 中的 SetBorderStyleInXls 项目。

## 2.2.5 设置单元格字体

本节我们将继续使用 NPOI 来设置单元格格式，这一节我们主要讲如何设置“字体”。



在设置字体之前，我们首先要做的就是创建字体对象，这和创建数字格式很相似。

```
HSSFFont font = hssfworkbook.CreateFont();
```

这句话会在 Excel 文件内部创建相应的 FontRecord，所以你不用客户因为自己机器上的 Excel 没有相应的字体设置而导致设置丢失。

字体在设置完成后，我们就可以把它赋给单元格样式，代码如下：

```
HSSFCellStyle style1 = hssfworkbook.CreateCellStyle();
style1.SetFont(font);
cell11.CellStyle=style1;
```

这里的 cell11 是 HSSFCell 的一个实例。

好了，下面我们就开始对字体进行设置。

### 字体名称

这里的字体名称是通过 HSSFFont.FontName 进行设置的，至于具体的名称，只要是常用字体都可以，比如说 Arial, Verdana 等，当然也可以是中文字体名，如宋体、黑体等。不过设置字体名称有个前提，那就是假设打开这个 xls 文件的客户机上有这种字体，如果没有，Excel 将使用默认字体。

下面就是设置字体名称为“宋体”的代码：font.FontName = “宋体”；

### 字号

与字号有关的属性有两个，一个是 FontHeight，一个是 FontHeightInPoints。区别在于，FontHeight 的值是 FontHeightInPoints 的 20 倍，通常我们在 Excel 界面中看到的字号，比如说 12，对应的是 FontHeightInPoints 的值，而 FontHeight 要产生 12 号字体的大小，值应该是 240。所以通常建议你用 FontHeightInPoint 属性。

如果要设置字号为 12，代码就是 font.FontHeightInPoints = 12；

### 字体颜色

这里可能会与 CellStyle 上的 ForegroundColor 和 BackgroundColor 产生混淆，其实所有的字体颜色都是在 HSSFFont 的实例上设置的，CellStyle 的 ForegroundColor 和 BackgroundColor 分别指背景填充色和填充图案的颜色，和文本颜色无关。

要设置字体颜色，我们可以用 HSSFFont.Color 属性，颜色可以通过 HSSFColor 获得，代码如下所示：

```
font.Color = HSSFColor.RED.index;
```

这行代码把文本设置为红色。

### 下划线

通常我们所说的下划线都是单线条的，其实 Excel 支持好几种下划线，如下所示：

类型	对应的值
单下划线	HSSFFont.U_SINGLE
双下划线	HSSFFont.U_DOUBLE
会计用单下划线	HSSFFont.U_SINGLE_ACCOUNTING

会计用双下划线	HSSFFont.U_DOUBLE_ACCOUNTING
无下划线	HSSFFont.U_NONE

当你要设置下划线时，可以用 HSSFFont.Underline 属性，这是一个 byte 类型的值，例如 font.Underline=HSSFFont.U\_SINGLE，这行代码就是设置单下划线的代码。

### 上标下标

设置这东西可以用 HSSFFont.TypeOffset 属性，值有以下几种：

TypeOffset 的值	说明
HSSFFont.SS_SUPER	上标
HSSFFont.SS_SUB	下标
HSSFFont.SS_NONE	普通，默认值

所以如果你要上标的话，可以用下面的代码：font.TypeOffset=HSSFFont.SS\_SUPER；

### 删除线

设置这东西可以用 HSSFFont.IsStrikeout 属性，当为 true 时，表示有删除线；为 false 则表示没有删除线。

相关范例请参考 [NPOI 1.2 正式版](#) 中的 ApplyFontInXls 的项目。

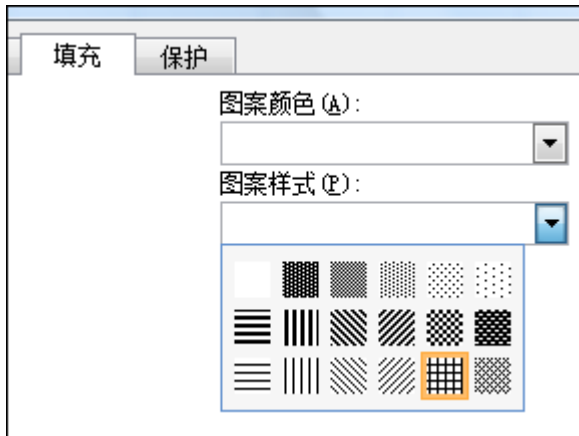
## 2.2.6 设置单元格的背景和图案

本节我们将用 NPOI 来为单元格添加背景和图案。

在之前的教程中，我们已经提到 HSSFCellStyle 有两个背景颜色属性，一个叫 FillBackgroundColor，另一个叫 FillForegroundColor，但其实这指的都是背景颜色，那为什么还有 ForegroundColor 呢？为了能够帮助大家理解，我们举一个实际的例子，下面这个图案是 Excel 的一个单元格：



线是白色的，背景是红色的。这里的线其实就是下面的 Excel 界面中的图案：



至于线的颜色则是图案颜色，即白色。








所以以上单元格如果要用 NPOI 来设置就可以用以下代码完成：


```
//fill background
```

```
HSSFCellStyle style8 = hssfworkbook.CreateCellStyle();
style8.FillForegroundColor = NPOI.HSSF.Util.HSSFColor.WHITE.index;
style8.FillPattern = HSSFCellStyle.SQUARES;
style8.FillBackgroundColor = NPOI.HSSF.Util.HSSFColor.RED.index;
sheet1.CreateRow(7).CreateCell(0).CellStyle = style8;
```

现在是不是清楚一些了，这里的 FillPattern 就图案样式，所有的枚举值都是 HSSFCellStyle 的常量；FillForegroundColor 就是图案的颜色，而 FillBackgroundColor 则是背景的颜色，即红色。

下面罗列一下图案样式及其对应的值：

图案样式	常量
	HSSFCellStyle.NO_FILL
	HSSFCellStyle.ALT_BARS
	HSSFCellStyle.FINE_DOTS
	HSSFCellStyle.SPARSE_DOTS
	HSSFCellStyle.LESS_DOTS
	HSSFCellStyle.LEAST_DOTS
	HSSFCellStyle.BRICKS

	HSSFCellStyle.BIG_SPOTS
	HSSFCellStyle.THICK_FORWARD_DIAG
	HSSFCellStyle.THICK_BACKWARD_DIAG
	HSSFCellStyle.THICK_VERT_BANDS
	HSSFCellStyle.THICK_HORZ_BANDS
	HSSFCellStyle.THIN_HORZ_BANDS
	HSSFCellStyle.THIN_VERT_BANDS
	HSSFCellStyle.THIN_BACKWARD_DIAG
	HSSFCellStyle.THIN_FORWARD_DIAG
	HSSFCellStyle.SQUARES
	HSSFCellStyle.DIAMONDS

通过这张表，你将很容易找到自己需要的样式，不用再去一个一个猜测了。

相关范例请参考 [NPOI 1.2 正式版](#) 中的 `ColorfullMatrixTable` 和 `FillBackgroundInXls`。

### 2.2.7 设置单元格的宽度和高度

在 Excel 中，单元格的宽度其实就是列的宽度，因为 Excel 假设这一列的单元格的宽度肯定一致。所以要设置单元格的宽度，我们就得从列的宽度下手，`HSSFSheet` 有个方法叫 `SetColumnWidth`，共有两个参数：一个是列的索引（从 0 开始），一个是宽度。

现在假设你要设置 B 列的宽度，就可以用下面的代码：

```
HSSFWorkbook hssfworkbook = new HSSFWorkbook();
HSSFSheet sheet1 = hssfworkbook.CreateSheet("Sheet1");
sheet1.SetColumnWidth(1, 100 * 256);
```

这里你会发现一个有趣的现象，`SetColumnWidth` 的第二个参数要乘以 256，这是怎么回事呢？其实，这个参数的单位是 1/256 个字符宽度，也就是说，这里是把 B 列的宽度设置为了 100 个字符。



刚才说的是如何设置，那如何去读取一个列的宽度呢？直接用 `GetColumnWidth` 方法，这个方法只有一个参数，那就是列的索引号。如下所示：

```
int collwidth = sheet1.GetColumnWidth(1);
```

说完宽度，我们来说高度，在 Excel 中，每一行的高度也是要求一致的，所以设置单元格的高度，其实就是设置行的高度，所以相关的属性也应该在 `HSSFRow` 上，它就是 `HSSFRow.Height` 和 `HeightInPoints`，这两个属性的区别在于 `HeightInPoints` 的单位是点，而 `Height` 的单位是 1/20 个点，所以 `Height` 的值永远是 `HeightInPoints` 的 20 倍。

要设置第一行的高度，可以用如下代码：

```
sheet1.CreateRow(0).Height = 200*20;
```

或者

```
sheet1.CreateRow(0).HeightInPoints = 200;
```

如果要获得某一行的行高，可以直接拿 `HSSFRow.Height` 属性的返回值。

你可能觉得一行一行设置行高或者一列一列设置列宽很麻烦，那你可以考虑使用 `HSSFSheet.DefaultColumnWidth`、`HSSFSheet.DefaultRowHeight` 和 `HSSFSheet.DefaultRowHeightInPoints` 属性。

一旦设置了这些属性，如果某一行或者某一列没有设置宽度，就会使用默认宽度或高度。代码如下：

```
sheet1.DefaultColumnWidth=100*256;
```

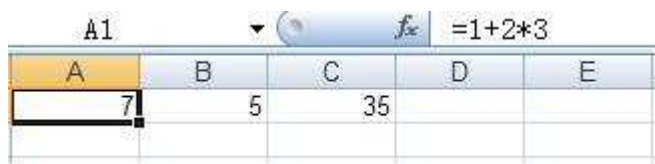
```
sheet1.DefaultRowHeight=30*20;
```

相关范例请见 [NPOI 1.2 正式版](#) 中的 `SetWidthAndHeightInXls` 项目

## 2.3 使用 Excel 公式

### 2.3.1 基本计算

从这节开始，我们将开始学习 Excel 高级一点的功能——公式。为某个单元格指定公式后，单元格中的内容将根据公式计算得出，如图：



A	B	C	D	E
7	5	35		

图中设置的是一个基本表达式“1+2\*3”，单元格 A1 中将显示此表达式计算的结果“7”，如图所示。对应的 C# 生成代码也很简单，如下：

```
HSSFSheet sheet1 = hssfworkbook.CreateSheet("Sheet1");
```

```
HSSFRow row1=sheet1.CreateRow(0);
```

```

HSSFCell cell1 = row1.CreateCell(0);
HSSFCell cell2 = row1.CreateCell(1);
HSSFCell cell3 = row1.CreateCell(2);
cell1.SetCellFormula("1+2*3");
cell2.SetCellValue(5);

```

同样，NPOI 也支持单元格引用类型的公式设置，如下图中的 C1=A1\*B1。

C1		=A1*B1		
A	B	C	D	E
7	5	35		

对应的公式设置代码为：

```

cell3.SetCellFormula("A1*B1");

```

是不是很简单呢？但要注意，在利用 NPOI 写程序时，行和列的计数都是从 0 开始计算的，但在设置公式时又是按照 Excel 的单元格命名规则来的。

### 2.3.2 SUM 函数

本节我们开始学习 Excel 中最常用的函数—Sum 求和函数。

首先，我们先看一上最简单的 Sum 函数：Sum(num1, num2, ...)。使用效果如图

E1		=SUM(A1, C1)			
A	B	C	D	E	F
1	2	3	6	4	6

图中的 E1=Sum(A1, C1) 表示将 A1 与 C1 的和填充在 E1 处，与公式” E1=A1+C1” 等效。对应的生成代码与上一节中的基本计算公式类似：

```

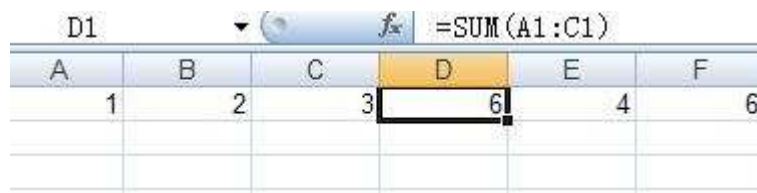
HSSFSheet sheet1 = hssfworkbook.CreateSheet("Sheet1");
HSSFRow row1 = sheet1.CreateRow(0);
HSSFCell cell1 = row1.CreateCell(0);
HSSFCell cell2 = row1.CreateCell(1);
HSSFCell cell3 = row1.CreateCell(2);
HSSFCell cellSum1 = row1.CreateCell(3);
HSSFCell cellSum2 = row1.CreateCell(4);
HSSFCell cellSum3 = row1.CreateCell(5);

cell1.SetCellValue(1);

```

```
cel2.SetCellValue(2);
cel3.SetCellValue(3);
celSum2.SetCellFormula("sum(A1,C1)");
```

当然，把每一个单元格作为 Sum 函数的参数很容易理解，但如果要求 and 的单元格很多，那么公式就会很长，既不方便阅读也不方便书写。所以 Excel 提供了另外一种多个单元格求和的写法：



A	B	C	D	E	F
1	2	3	6	4	6

如上图中的“Sum(A1:C1)”表示求从 A1 到 C1 所有单元格的和，相当于 A1+B1+C1。  
对应的代码为：

```
celSum1.SetCellFormula("sum(A1:C1)");
```

最后，还有一种求和的方法。就是先定义一个区域，如“range1”，然后再设置 Sum(range1)，此时将计算区域中所有单元格的和。

定义区域的代码为：

```
HSSFName range = hssfworkbook.CreateName();
range.Reference = "Sheet1!$A1:$C1";
range.NameName = "range1";
```

执行此代码后的 Excel 文件将在的公式菜单下的名称管理器 (Excel2007 的菜单路径, 2003 稍有不同) 中看到如下区域定义：



给单元格 F1 加上公式:

```
celSum3.SetCellFormula("sum(range1)");
```

生成的 Excel 如下图所示:

F1						fx	=SUM(range1)
A	B	C	D	E	F		
1	2	3	6	4			6

### 2.3.3 日期函数

Excel 中有非常丰富的日期处理函数, 在 NPOI 中同样得到了很好的支持。如下图:

D2												fx	=CONCATENATE(DATEDIF(B2,TODAY(),"y"),"年",DATEDIF(B2,TODAY(),"ym"),"个月")	
A	B	C	D	E	F	G	H	I	J	K	L	M		
姓名	参加工作时间	当前日期	工作年限											
aTao.Xiang	2004-7-1	2009-9-12	5年2个月											

对应的与前面的基本公式设置类似:

```
HSSFSheet sheet1 = hssfworkbook.CreateSheet("Sheet1");
HSSFRow row1 = sheet1.CreateRow(0);
HSSFRow row2 = sheet1.CreateRow(1);
row1.CreateCell(0).SetCellValue("姓名");
row1.CreateCell(1).SetCellValue("参加工作时间");
row1.CreateCell(2).SetCellValue("当前日期");
row1.CreateCell(3).SetCellValue("工作年限");
```

```

HSSFCell cel1 = row2.CreateCell(0);
HSSFCell cel2 = row2.CreateCell(1);
HSSFCell cel3 = row2.CreateCell(2);
HSSFCell cel4 = row2.CreateCell(3);

cel1.SetCellValue("aTao.Xiang");
cel2.SetCellValue(new DateTime(2004, 7, 1));
cel3.SetCellFormula("TODAY()");
cel4.SetCellFormula("CONCATENATE(DATEDIF(B2, TODAY(), \"y\"), \"年\", DATEDIF(B2, TODAY(), \"ym\"), \"个月\")");

```

```

//在 poi 中日期是以 double 类型表示的，所以要格式化
HSSFCellStyle cellStyle = hssfworkbook.CreateCellStyle();
HSSFDataFormat format = hssfworkbook.CreateDataFormat();
cellStyle.DataFormat = format.GetFormat("yyyy-m-d");

```

```

cel2.CellStyle = cellStyle;
cel3.CellStyle = cellStyle;

```

下面对上例中用到的几个主要函数作一些说明：  
TODAY()：取得当前日期；  
DATEDIF(B2, TODAY(), "y")：取得 B2 单元格的日期与前日期以年为单位的时间间隔。  
("Y" :表示以年为单位, "m" 表示以月为单位;" d" 表示以天为单位)；  
CONCATENATE(str1, str2,...)：连接字符串。

另外附上 Excel 中常用的日期函数列表，只需要将此句代码作适当修改即可：

```

cel4.SetCellFormula("CONCATENATE(DATEDIF(B2, TODAY(), \"y\"), \"年\", DATEDIF(B2, TODAY(), \"ym\"), \"个月\")");

```

函数名	函数说明	语法
DATE	返回代表特定日期的系列数。	DATE(year, month, day)
DATEDIF	计算两个日期之间的天数、月数或年数。	DATEDIF(start_date, end_date, unit)
DATEVALUE	函数 DATEVALUE 的主要功能是将以文字表示的日期转换成一个系列数。	DATEVALUE(date_text)
DAY	返回以系列数表示的某日期的天数，用整数 1 到 31 表示。	DAY(serial_number)

DAYS360	按照一年 360 天的算法（每个月以 30 天计，一年共计 12 个月），返回两日期间相差的天数。	DAYS360(start_date, end_date, method)
EDATE	返回指定日期 (start_date) 之前或之后指定月份数的日期系列数。使用函数 EDATE 可以计算与发行日处于一月中同一天的到期日的日期。	EDATE(start_date, months)
EOMONTH	返回 start-date 之前或之后指定月份中最后一天的系列数。用函数 EOMONTH 可计算特定月份中最后一天的时间系列数，用于证券的到期日等计算。	EOMONTH(start_date, months)
HOUR	返回时间值的小时数。即一个介于 0 (12:00 A.M.) 到 23 (11:00 P.M.) 之间的整数。	HOUR(serial_number)
MINUTE	返回时间值中的分钟。即一个介于 0 到 59 之间的整数。	MINUTE(serial_number)
MONTH	返回以系列数表示的日期中的月份。月份是介于 1 (一月) 和 12 (十二月) 之间的整数。	MONTH(serial_number)
NETWORKDAYS	返回参数 start-data 和 end-data 之间完整的工作日数值。工作日不包括周末和专门指定的假期	NETWORKDAYS(start_date, end_date, holidays)
NOW	返回当前日期和时间所对应的系列数。	NOW( )
SECOND	返回时间值的秒数。返回的秒数为 0 至 59 之间的整数。	SECOND(serial_number)
TIME	返回某一特定时间的小数值，函数 TIME 返回的小数值为从 0 到 0.99999999 之间的数值，代表从 0:00:00 (12:00:00 A.M) 到 23:59:59 (11:59:59 P.M) 之间的时间。	TIME(hour, minute, second)
TIMEVALUE	返回由文本串所代表的时间的小数值。该小数值为从 0 到 0.999999999 的数值，代表从 0:00:00 (12:00:00 AM) 到 23:59:59 (11:59:59 PM) 之间的时间。	TIMEVALUE(time_text)
TODAY	返回当前日期的系列数，系列数是 Microsoft Excel 用于日期和时间计算的日期-时间代码。	TODAY( )
WEEKDAY	返回某日期为星期几。默认情况下，其值为 1	WEEKDAY(serial_number, retur

	(星期天) 到 7 (星期六) 之间的整数。	n_type)
WEEKNUM	返回一个数字, 该数字代表一年中的第几周。	WEEKNUM(serial_num, return_type)
WORKDAY	返回某日期(起始日期)之前或之后相隔指定工作日的某一日期日期值。工作日不包括周末和专门指定的假日。	WORKDAY(start_date, days, holidays)
YEAR	返回某日期的年份。返回值为 1900 到 9999 之间的整数。	YEAR(serial_number)
YEARFRAC	返回 start_date 和 end_date 之间的天数占全年天数的百分比。	YEARFRAC(start_date, end_date, basis)

### 2.3.4 字符串函数

这一节我们开始学习 Excel 另一类非常常见的函数—字符串函数。在 Excel 中提供了非常丰富的字符串函数, 在 NPOI 中同样得到了很好的支持。

#### 一、大小写转换类函数

LOWER(String): 将一个字符串中的所有大写字母转换为小写字母。

UPPER(String): 将文本转换成大写形式。

PROPER(String): 将字符串的首字母及任何非字母字符之后的首字母转换成大写。将其余的字母转换成小写。

对应的 C# 代码与前几节讲的设置公式的代码类似:

```
HSSFRow row1 = sheet1.CreateRow(0);
row1.CreateCell(0).SetCellValue("待操作字符串");
row1.CreateCell(1).SetCellValue("操作函数");
row1.CreateCell(2).SetCellValue("操作结果");

HSSFRow row2 = sheet1.CreateRow(1);
row2.CreateCell(0).SetCellValue("This is a NPOI example!");
row2.CreateCell(1).SetCellValue("LOWER(A2)");
//将此句中的“LOWER(A2)”换成 UPPER(A2)、PROPER(A2) 可以看到不同效果。
row2.CreateCell(2).SetCellFormula("LOWER(A2)");
```

#### 二、取出字符串中的部分字符

LEFT(text, num\_chars): LEFT(text, num\_chars) 其中 Text 是包含要提取字符的文本串。Num\_chars 指定要由 LEFT 所提取的字符数。

MID(text, start\_num, num\_chars): MID(text, start\_num, num\_chars)其中 Text 是包含要提取字符的文本串。Start\_num 是文本中要提取的第一个字符的位置, num\_chars 表示要提取的字符的数。

RIGHT(text, num\_chars): RIGHT(text, num\_chars)其中 Text 是包含要提取字符的文本串。Num\_chars 指定希望 RIGHT 提取的字符数。

代码与上面类似, 就不写了。

### 三、 去除字符串的空白

TRIM(text): 其中 Text 为需要清除其中空格的文本。需要注意的是, 与 C#中的 Trim 不同, Excel 中的 Trim 函数不仅会删除字符串头尾的字符, 字符串中的多余字符也会删除, 单词之间只会保留一个空格。

### 四、 字符串的比较

EXACT(text1, text2): 比较两个字符串是否相等, 区分大小写。

执行效果如下:

A	B	C
待操作字符串	操作函数	操作结果
This is a NPOI example!	LOWER(A2)	this is a npoi example!
This is a NPOI example!	UPPER(A3)	THIS IS A NPOI EXAMPLE!
This is a NPOI example!	PROPER(A4)	This Is A Npoi Example!
This is a NPOI example!	LEFT(A5,5)	This
This is a NPOI example!	RIGHT(A6,5)	mple!
This is a NPOI example!	MID(A7,5,10)	is a NPOI
This is a NPOI example!	TRIM(A8)	This is a NPOI example!
This is a NPOI example!	EXACT(A9,"THIS IS A NPOI EXAMPLE!")	FALSE

在此只简单的讲了一下常用的函数, Excel 中还有很多的字符串函数, 在此附上, 读者可以一个一个去测试。

函数名	函数说明	语法
ASC	将字符串中的全角(双字节)英文字母更改为半角(单字节)字符。	ASC(text)
CHAR	返回对应于数字代码的字符, 函数 CHAR 可将其他类型计算机文件中的代码转换为字符。	CHAR(number)
CLEAN	删除文本中不能打印的字符。对从其他应用程序中输入的字符串使用 CLEAN 函数, 将删除其中含有的当前操作系统无法打印的字符。例如, 可	CLEAN(text)



	以删除通常出现在数据文件头部或尾部、无法打印的低级计算机代码。	
CODE	返回字符串中第一个字符的数字代码。返回的代码对应于计算机当前使用的字符集。	CODE(text)
CONCATENATE	将若干字符串合并到一个字符串中。	CONCATENATE (text1, text2, ...)
DOLLAR	依照货币格式将小数四舍五入到指定的位数并转换成文字。	DOLLAR 或 RMB(number, decimals)
EXACT	该函数测试两个字符串是否完全相同。如果它们完全相同，则返回 TRUE；否则，返回 FALSE。函数 EXACT 能区分大小写，但忽略格式上的差异。利用函数 EXACT 可以测试输入文档内的文字。	EXACT(text1, text2)
FIND	FIND 用于查找其他文本串 (within_text) 内的文本串 (find_text)，并从 within_text 的首字符开始返回 find_text 的起始位置编号。	FIND(find_text, within_text, start_num)
FIXED	按指定的小数位数进行四舍五入，利用句点和逗号，以小数格式对该数设置格式，并以字符串形式返回结果。	FIXED(number, decimals, no_commas)
JIS	将字符串中的半角（单字节）英文字母或片假名更改为全角（双字节）字符。	JIS(text)
LEFT	LEFT 基于所指定的字符数返回文本串中的第一个或前几个字符。 LEFTB 基于所指定的字节数返回文本串中的第一个或前几个字符。此函数用于双字节字符。	LEFT(text, num_chars) LEFTB(text, num_bytes)
LEN	LEN 返回文本串中的字符数。 LENB 返回文本串中用于代表字符的字节数。此函数用于双字节字符。	LEN(text) LENB(text)
LOWER	将一个字符串中的所有大写字母转换为小写字母。	LOWER(text)

MID	<p>MID 返回文本串中从指定位置开始的特定数目的字符，该数目由用户指定。</p> <p>MIDB 返回文本串中从指定位置开始的特定数目的字符，该数目由用户指定。此函数用于双字节字符。</p>	<p>MID(text, start_num, num_chars)</p> <p>MIDB(text, start_num, num_bytes)</p>
PHONETIC	<p>提取文本串中的拼音 (furigana) 字符。</p>	<p>PHONETIC(reference)</p>
PROPER	<p>将文字串的首字母及任何非字母字符之后的首字母转换成大写。将其余的字母转换成小写。</p>	<p>PROPER(text)</p>
REPLACE	<p>REPLACE 使用其他文本串并根据所指定的字符数替换某文本串中的部分文本。</p> <p>REPLACEB 使用其他文本串并根据所指定的字符数替换某文本串中的部分文本。此函数专为双字节字符使用。</p>	<p>REPLACE(old_text, start_num, num_chars, new_text)</p> <p>REPLACEB(old_text, start_num, num_bytes, new_text)</p>
REPT	<p>按照给定的次数重复显示文本。可以通过函数 REPT 来不断地重复显示某一文字串，对单元格进行填充。</p>	<p>REPT(text, number_times)</p>
RIGHT	<p>RIGHT 根据所指定的字符数返回文本串中最后一个或多个字符。</p> <p>RIGHTB 根据所指定的字符数返回文本串中最后一个或多个字符。此函数用于双字节字符。</p>	<p>RIGHT(text, num_chars)</p> <p>RIGHTB(text, num_bytes)</p>
SEARCH	<p>SEARCH 返回从 start_num 开始首次找到特定字符或文本串的位置上特定字符的编号。使用 SEARCH 可确定字符或文本串在其他文本串中的位置，这样就可使用 MID 或 REPLACE 函数更改文本。</p> <p>SEARCHB 也可在其他文本串 (within_text) 中查找文本串 (find_text)，并返回 find_text 的</p>	<p>SEARCH(find_text, within_text, start_num)</p> <p>SEARCHB(find_text, within_text, start_num)</p>

	起始位置编号。此结果是基于每个字符所使用的字节数，并从 start_num 开始的。此函数用于双字节字符。此外，也可使用 FINDB 在其他文本串中查找文本串。	
SUBSTITUTE	在文字串中用 new_text 替代 old_text。如果需要在某一文字串中替换指定的文本，请使用函数 SUBSTITUTE；如果需要在某一文字串中替换指定位置处的任意文本，请使用函数 REPLACE。	SUBSTITUTE(text, old_text, new_text, instance_num)
T	将数值转换成文本。	T(value)
TEXT	将一数值转换为按指定数字格式表示的文本。	TEXT(value, format_text)
TRIM	除了单词之间的单个空格外，清除文本中所有的空格。在从其他应用程序中获取带有不规则空格的文本时，可以使用函数 TRIM。	TRIM(text)
UPPER	将文本转换成大写形式。	UPPER(text)
VALUE	将代表数字的字符串转换成数字。	VALUE(text)
WIDECHAR	将单字节字符转换为双字节字符。	WIDECHAR(text)
YEN	使用 ¥（日圆）货币格式将数字转换成文本，并对指定位置后的数字四舍五入。	YEN(number, decimals)

### 2.3.5 If 函数

在 Excel 中, IF(logical\_test, value\_if\_true, value\_if\_false) 用来用作逻辑判断。其中 Logical\_test 表示计算结果为 TRUE 或 FALSE 的任意值或表达式；value\_if\_true 表示当表达式 Logical\_test 的值为 TRUE 时的返回值；value\_if\_false 表示当表达式 Logical\_test 的值为 FALSE 时的返回值。同样在 NPOI 中也可以利用这个表达式进行各种逻辑运算。如下代码分别设置了 B2 和 D2 单元格的用于逻辑判断的公式。

```
HSSFSheet sheet1 = hssfworkbook.CreateSheet("Sheet1");
```

```
HSSFRow row1 = sheet1.CreateRow(0);
```

```
row1.CreateCell(0).SetCellValue("姓名");
```

```

row1.CreateCell(1).SetCellValue("身份证号");
row1.CreateCell(2).SetCellValue("性别");
row1.CreateCell(3).SetCellValue("语文");
row1.CreateCell(4).SetCellValue("是否合格");

HSSFRow row2 = sheet1.CreateRow(1);
row2.CreateCell(0).SetCellValue("令狐冲");
row2.CreateCell(1).SetCellValue("420821198808101014");
row2.CreateCell(2).SetCellFormula("IF(MOD(MID(B2,18,1),2)=0,\"男\",\"女\")");
row2.CreateCell(3).SetCellValue(85);
row2.CreateCell(4).SetCellFormula("IF(D2>60,IF(D2>90,\"优秀\",\"合格\"),\"不合格\")");

```

其中最关键的两句执行结果如下：

```
row2.CreateCell(2).SetCellFormula("IF(MOD(MID(B2,18,1),2)=0,\"男\",\"女\")");
```

C2		fx =IF(MOD(MID(B2,18,1),2)=0,"男","女")					
A	B	C	D	E	F	G	H
姓名	身份证号	性别	语文	是否合格			
令狐冲	420821198808101014	男	85	合格			

```
row2.CreateCell(4).SetCellFormula("IF(D2>60,IF(D2>90,\"优秀\",\"合格\"),\"不合格\")");
```

E2		fx =IF(D2>60,IF(D2>90,"优秀","合格"),"不合格")						
A	B	C	D	E	F	G	H	I
姓名	身份证号	性别	语文	是否合格				
令狐冲	420821198808101014	男	85	合格				

下面分别对这几个函数作一些说明：

MOD(MID(B2,18,1),2)：我们知道18位身份证号的第18位表示性别，偶数为男性，奇数为女性，所以用了MID(B2,18,1)取第18位数字（与C#中一般从0计数不同，第二个参数是从1算起，有关MID函数的更多信息，请参见[字符串函数](#)），用MOD取余函数判断奇偶。在Excel中对数据类型的控制没有C#中那么严格，如此例中我截取出来的是字符串，但当我做取余运算时Excel会自动转换。

IF(D2>60,IF(D2>90,"优秀","合格"),"不合格")：这是IF的嵌套使用，表示90分以上为优秀，60分以上为合格，否则为不合格。

## 2.3.6 COUNTIF 和 SUMIF 函数

### 一、COUNTIF

这一节，我们一起来学习 Excel 中另一个常用的函数—COUNTIF 函数，看函数名就知道这是一个用来在做满足某条件的计数的函数。先来看一看它的语法：  
COUNTIF(range, criteria)，参数说明如下：

Range	需要进行读数的计数
Criteria	条件表达式，只有当满足此条件时才进行计数

接下来看一个例子，代码如下：

```
HSSFSheet sheet1 = hssfworkbook.CreateSheet("Sheet1");

HSSFRow row1 = sheet1.CreateRow(0);
row1.CreateCell(0).SetCellValue("姓名");
row1.CreateCell(1).SetCellValue("成绩");

HSSFRow row2 = sheet1.CreateRow(1);
row2.CreateCell(0).SetCellValue("令狐冲");
row2.CreateCell(1).SetCellValue(85);

HSSFRow row3 = sheet1.CreateRow(2);
row3.CreateCell(0).SetCellValue("任盈盈");
row3.CreateCell(1).SetCellValue(90);

HSSFRow row4 = sheet1.CreateRow(3);
row4.CreateCell(0).SetCellValue("任我行");
row4.CreateCell(1).SetCellValue(70);

HSSFRow row5 = sheet1.CreateRow(4);
row5.CreateCell(0).SetCellValue("左冷禅");
row5.CreateCell(1).SetCellValue(45);

HSSFRow row6 = sheet1.CreateRow(5);
row6.CreateCell(0).SetCellValue("岳不群");
row6.CreateCell(1).SetCellValue(50);

HSSFRow row7 = sheet1.CreateRow(6);
row7.CreateCell(0).SetCellValue("合格人数：");
row7.CreateCell(1).SetCellFormula("COUNTIF(B2:B6, \">60\")");
```

执行结果如下：

B7		fx =COUNTIF(B2:B6, ">60")			
A	B	C	D	E	F
姓名	成绩				
令狐冲	85				
任盈盈	90				
任我行	70				
左冷禅	45				
岳不群	50				
合格人数：	3				

我们可以看到，CountIf 函数成功的统计出了区域“B2: B6”中成绩合格的人数（这里定义成绩大于 60 为合格）

## 二、SUMIF

接下来，顺便谈谈另一个与 CountIF 类似的函数—SumIf 函数。此函数用于统计某区域内满足某条件的值的求和(CountIf 是计数)。与 CountIF 不同，SumIF 有三个参数，语法为 SumIF(criteria\_range, criteria, sum\_range)，各参数的说明如下：

criteria_range	条件测试区域，第二个参数 Criteria 中的条件将与此区域中的值进行比较
criteria	条件测试值，满足条件的对应的 sum_range 项将进行求和计算
sum_range	汇总数据所在区域，求和时会排除掉不满足 Criteria 条件的对应的项

我们还是以例子来加以说明：

```
HSSFSheet sheet1 = hssfworkbook.CreateSheet("Sheet1");
```

```
HSSFRow row1 = sheet1.CreateRow(0);  
row1.CreateCell(0).SetCellValue("姓名");  
row1.CreateCell(1).SetCellValue("月份");  
row1.CreateCell(2).SetCellValue("销售额");
```

```
HSSFRow row2 = sheet1.CreateRow(1);  
row2.CreateCell(0).SetCellValue("令狐冲");  
row2.CreateCell(1).SetCellValue("一月");  
row2.CreateCell(2).SetCellValue(1000);
```

```
HSSFRow row3 = sheet1.CreateRow(2);  
row3.CreateCell(0).SetCellValue("任盈盈");  
row3.CreateCell(1).SetCellValue("一月");  
row3.CreateCell(2).SetCellValue(900);
```

```
HSSFRow row4 = sheet1.CreateRow(3);
row4.CreateCell(0).SetCellValue("令狐冲");
row4.CreateCell(1).SetCellValue("二月");
row4.CreateCell(2).SetCellValue(2000);

HSSFRow row5 = sheet1.CreateRow(4);
row5.CreateCell(0).SetCellValue("任盈盈");
row5.CreateCell(1).SetCellValue("二月");
row5.CreateCell(2).SetCellValue(1000);

HSSFRow row6 = sheet1.CreateRow(5);
row6.CreateCell(0).SetCellValue("令狐冲");
row6.CreateCell(1).SetCellValue("三月");
row6.CreateCell(2).SetCellValue(3000);

HSSFRow row7 = sheet1.CreateRow(6);
row7.CreateCell(0).SetCellValue("任盈盈");
row7.CreateCell(1).SetCellValue("三月");
row7.CreateCell(2).SetCellValue(1200);

HSSFRow row8 = sheet1.CreateRow(7);
row8.CreateCell(0).SetCellValue("令狐冲一季度销售额: ");
row8.CreateCell(2).SetCellFormula("SUMIF(A2:A7, \"=令狐冲\", C2:C7)");

HSSFRow row9 = sheet1.CreateRow(8);
row9.CreateCell(0).SetCellValue("任盈盈一季度销售额: ");
row9.CreateCell(2).SetCellFormula("SUMIF(A2:A7, \"=任盈盈\", C2:C7)");
```

执行结果如下:

A	B	C	D	E	F	G
姓名	月份	销售额				
令狐冲	一月	1000				
任盈盈	一月	900				
令狐冲	二月	2000				
任盈盈	二月	1000				
令狐冲	三月	3000				
任盈盈	三月	1200				
令狐冲一季度销售额:		6000				
任盈盈一季度销售额:		3100				

如上图，SUMIF 统计出了不同人一季度的销售额。

### 2.3.7 LOOKUP 函数

今天，我们一起学习 Excel 中的查询函数——LOOKUP。其基本语法形式为 LOOKUP(lookup\_value, lookup\_vector, result\_vector)。还是以例子加以说明更容易理解：

```

HSSFSheet sheet1 = hssfworkbook.CreateSheet("Sheet1");
HSSFRow row1 = sheet1.CreateRow(0);
row1.CreateCell(0).SetCellValue("收入最低");
row1.CreateCell(1).SetCellValue("收入最高");
row1.CreateCell(2).SetCellValue("税率");

HSSFRow row2 = sheet1.CreateRow(1);
row2.CreateCell(0).SetCellValue(0);
row2.CreateCell(1).SetCellValue(3000);
row2.CreateCell(2).SetCellValue(0.1);

HSSFRow row3 = sheet1.CreateRow(2);
row3.CreateCell(0).SetCellValue(3001);
row3.CreateCell(1).SetCellValue(10000);
row3.CreateCell(2).SetCellValue(0.2);

HSSFRow row4 = sheet1.CreateRow(3);
row4.CreateCell(0).SetCellValue(10001);
row4.CreateCell(1).SetCellValue(20000);
row4.CreateCell(2).SetCellValue(0.3);

HSSFRow row5 = sheet1.CreateRow(4);

```



```

row5.CreateCell(0).SetCellValue(20001);
row5.CreateCell(1).SetCellValue(50000);
row5.CreateCell(2).SetCellValue(0.4);

HSSFRow row6 = sheet1.CreateRow(5);
row6.CreateCell(0).SetCellValue(50001);
row6.CreateCell(2).SetCellValue(0.5);

HSSFRow row8 = sheet1.CreateRow(7);
row8.CreateCell(0).SetCellValue("收入");
row8.CreateCell(1).SetCellValue("税率");

HSSFRow row9 = sheet1.CreateRow(8);
row9.CreateCell(0).SetCellValue(7800);
row9.CreateCell(1).SetCellFormula("LOOKUP(A9,$A$2:$A$6,$C$2:$C$6)");

```

这是一个根据工资查询相应税率的例子。我们首先创建了不同工资区间对应税率的字典，然后根据具体的工资在字典中找出对应的税率。执行后生成的 Excel 如下：

	A	B	C	D	E	F	G	H
1	收入最低	收入最高	税率					
2	0	3000	0.1					
3	3001	10000	0.2					
4	10001	20000	0.3					
5	20001	50000	0.4					
6	50001		0.5					
7								
8	收入	税率						
9	7800	0.2						

下面对各参数加以说明：

第一个参数：需要查找的内容，本例中指向 A9 单元格，也就是 7800；

第二个参数：比较对象区域，本例中的工资需要与 \$A\$2:\$A\$6 中的各单元格中的值进行比较；第三个参数：查找结果区域，如果匹配到会将此区域中对应的数据返回。如本例中返回 \$C\$2:\$C\$6 中对应的值。

可能有人会问，字典中没有 7800 对应的税率啊，那么 Excel 中怎么匹配的呢？答案是模糊匹配，并且 LOOKUP 函数只支持模糊匹配。Excel 会在 \$A\$2:\$A\$6 中找小于 7800 的最大值，也就是 A3 对应的 3001，然后将对应的 \$C\$2:\$C\$6 区域中的 C3 中的值返回，这就是最终结果 0.2 的由来。这下明白了吧：)

## VLOOKUP

另外，LOOKUP 函数还有一位大哥——VLOOKUP。两兄弟有很多相似之处，但大哥本领更大。Vlookup 用对比数与一个“表”进行对比，而不是 Lookup 函数的某 1 列或 1 行，并且 Vlookup 可以选择采用精确查询或是模糊查询方式，而 Lookup 只有模糊查询。将上例中设置公式的代码换成：

```
row9.CreateCell(1).SetCellFormula("VLOOKUP(A9,$A$2:$C$6,3,TRUE)");
```

执行后生成的 Excel 样式如下：

	A	B	C	D	E	F	G
1	收入最低	收入最高	税率				
2	0	3000	0.1				
3	3001	10000	0.2				
4	10001	20000	0.3				
5	20001	50000	0.4				
6	50001		0.5				
7							
8	收入	税率					
9	7800	0.2					

第一个参数：需要查找的内容，这里是 A9 单元格；

第二个参数：需要比较的表，这里是 \$A\$2:\$C\$6，注意 VLOOKUP 匹配时只与表中的第一列进行匹配。

第三个参数：匹配结果对应的列序号。这里要对应的是税率列，所以为 3。

第四个参数：指明是否模糊匹配。例子中的 TRUE 表示模糊匹配，与上例中一样。匹配到的是第三行。如果将此参数改为 FALSE，因为在表中的第 1 列中找不到 7800，所以会报“#N/A”的计算错误。

另外，还有与 VLOOKUP 类似的 HLOOKUP。不同的是 VLOOKUP 用于在表格或数值数组的首列查找指定的数值，并由此返回表格或数组当前行中指定列处的数值。而 HLOOKUP 用于在表格或数值数组的首行查找指定的数值，并由此返回表格或数组当前列中指定行处的数值。读者可以自己尝试。

### 2.3.8 随机数函数

我们知道，在大多数编程语言中都有随机数函数。在 Excel 中，同样存在着这样一个函数——RAND() 函数，用于生成随机数。先来看一个最简单的例子：

```
HSSFSheet sheet1 = hssfworkbook.CreateSheet("Sheet1");  
sheet1.CreateRow(0).CreateCell(0).SetCellFormula("RAND()");
```

RAND() 函数将返回一个 0-1 之间的随机数，执行后生成的 Excel 文件如下：

	A1			
	A	B	C	D
1	0.872880152			

这只是最简单直接的 RAND() 函数的应用，只要我们稍加修改，就可以作出很多种变换。

如取 0-100 之前的随机整数，可设置公式为：

```
sheet1.CreateRow(0).CreateCell(0).SetCellFormula("int(RAND()*100)");
```

取 10-20 之间的随机实数，可设置公式为：

```
sheet1.CreateRow(0).CreateCell(0).SetCellFormula("rand()*(20-10)+10");
```

随机小写字母：

```
sheet1.CreateRow(0).CreateCell(0).SetCellFormula("CHAR(INT(RAND()*26)+97)");
```

随机大写字母：

```
sheet1.CreateRow(0).CreateCell(0).SetCellFormula("CHAR(INT(RAND()*26)+65)");
```

随机大小写字母：

```
sheet1.CreateRow(0).CreateCell(0).SetCellFormula("CHAR(INT(RAND()*26)+if(INT(RAND()*2)=0,65,97))");
```

上面几例中除了用到 RAND 函数以外，还用到了 CHAR 函数用来将 ASCII 码换为字母，INT 函数用来取整。值得注意的是 INT 函数不会四舍五入，无论小数点后是多少都会被舍去。

这里只是 RAND 函数的几个简单应用，还有很多随机数的例子都可以根据这些，再结合不同的其它函数引申出来。

### 2.3.9 通过 NPOI 获得公式的返回值

前面我们学习了通过 NPOI 向 Excel 中设置公式，那么有些读者可能会问：“NPOI 能不能获取公式的返回值呢？”，答案是可以！

#### 一、 获取模板文件中公式的返回值

如在 D 盘中有一个名为 test.xls 的 Excel 文件，其内容如下：

	C1				
	A	B	C	D	E
1	3	4	12		

注意 C1 单元格中设置的是公式 “\$A1\*\$B1”，而不是值 “12”。利用 NPOI，只需要写简单的几句代码就可以取得此公式的返回值：

```
HSSFWorkbook wb = new HSSFWorkbook(new FileStream("d:/test.xls", FileMode.Open));
HSSFCell cell = wb.GetSheet("Sheet1").GetRow(0).GetCell(2);
System.Console.WriteLine(cell.NumericCellValue);
```

输出结果为：



可见 NPOI 成功的“解析”了此.xls 文件中的公式。注意 NumericCellValue 属性会自动根据单元格的类型处理，如果为空将返回 0，如果为数值将返回数值，如果为公式将返回公式计算后的结果。单元格的类型可以通过 CellType 属性获取。

## 二、 获取 NPOI 生成的 Excel 文件中公式的返回值

上例中是从一个已经存在的 Excel 文件中获取公式的返回值，那么如果 Excel 文件是通过 NPOI 创建的，直接用上面的方法获取，可能得不到想要的结果。如：

```
HSSFWorkbook hssfworkbook = new HSSFWorkbook();
HSSFSheet sheet1 = hssfworkbook.CreateSheet("Sheet1");
HSSFRow row = sheet1.CreateRow(0);
row.CreateCell(0).SetCellValue(3);
row.CreateCell(1).SetCellValue(4);
HSSFCell cell = row.CreateCell(2);

cell.SetCellFormula("$A1+$B1");
System.Console.WriteLine(cell.NumericCellValue);
```

执行上面代码，将输出结果“0”，而不是我们想要的结果“7”。那么将如何解决呢？这时要用到 HSSFFormulaEvaluator 类。在第 8 行后加上这两句就可以了：

```
HSSFFormulaEvaluator e = new HSSFFormulaEvaluator(hssfworkbook);
cell = e.EvaluateInCell(cell);
```


运行结果如下：



## 2.4 创建图形

### 2.4.1 画线

之所以说 NPOI 强大，是因为常用的 Excel 操作她都可以通过编程的方式完成。这节开始，我们开始学习 NPOI 的画图功能。先从最简单的开始，画一条直线：

	A	B	C	D	E
1					
2					
3					
4					
5					
6					
7					

对应的代码为：

```
HSSFSheet sheet1 = hssfworkbook.CreateSheet("Sheet1");
HSSFPatriarch patriarch = sheet1.CreateDrawingPatriarch();
HSSFClientAnchor a1 = new HSSFClientAnchor(255, 125, 1023, 150, 0, 0, 2, 2);
HSSFShape line1 = patriarch.CreateSimpleShape(a1);

line1.ShapeType = HSSFShape.OBJECT_TYPE_LINE;
line1.LineStyle = HSSFShape.LINESTYLE_SOLID;
//在 NPOI 中线的宽度 12700 表示 1pt, 所以这里是 0.5pt 粗的线条。
line1.LineWidth = 6350;
```

通常，利用 NPOI 画图主要有以下几个步骤：

1. 创建一个 Patriarch;
2. 创建一个 Anchor，以确定图形的位置;
3. 调用 Patriarch 创建图形;
4. 设置图形类型(直线，矩形，圆形等)及样式(颜色，粗细等)。

关于 HSSFClientAnchor(dx1, dy1, dx2, dy2, col1, row1, col2, row2)的参数，有必要在这里说明一下：

dx1: 起始单元格的 x 偏移量，如例子中的 255 表示直线起始位置距 A1 单元格左侧的距离;

dy1: 起始单元格的 y 偏移量，如例子中的 125 表示直线起始位置距 A1 单元格上侧的距离;

dx2: 终止单元格的 x 偏移量，如例子中的 1023 表示直线起始位置距 C3 单元格左侧的距离;

dy2: 终止单元格的 y 偏移量，如例子中的 150 表示直线起始位置距 C3 单元格上侧的距离;

col1: 起始单元格列序号，从 0 开始计算;

row1: 起始单元格行序号，从 0 开始计算，如例子中 col1=0, row1=0 就表示起始单元格为 A1;

col2: 终止单元格列序号, 从 0 开始计算;

row2: 终止单元格行序号, 从 0 开始计算, 如例子中 col2=2, row2=2 就表示起始单元格为 C3;

最后, 关于 LineStyle 属性, 有如下一些可选值, 对应的效果分别如图所示:

A	B	C	D	E	F	G	H
			HSSFShape.LINESTYLE_DASHDOTDOTSYS				
			HSSFShape.LINESTYLE_DASHDOTGEL				
			HSSFShape.LINESTYLE_DASHDOTSYS				
			HSSFShape.LINESTYLE_DASHGEL				
			HSSFShape.LINESTYLE_DASHSYS				
			HSSFShape.LINESTYLE_DOTGEL				
			HSSFShape.LINESTYLE_DOTSYS				
			HSSFShape.LINESTYLE_LONGDASHDOTDOTGEL				
			HSSFShape.LINESTYLE_LONGDASHDOTGEL				
			HSSFShape.LINESTYLE_LONGDASHGEL				
			HSSFShape.LINESTYLE_NONE				
			HSSFShape.LINESTYLE_SOLID				

## 2.4.2 画矩形

上一节我们讲了 NPOI 中画图的基本步骤:

1. 创建一个 Patriarch;
2. 创建一个 Anchor, 以确定图形的位置;
3. 调用 Patriarch 创建图形;
4. 设置图形类型(直线, 矩形, 圆形等)及样式(颜色, 粗细等)。

这一节我们将按照这个步骤创建一个矩形。废话少说, 上代码:

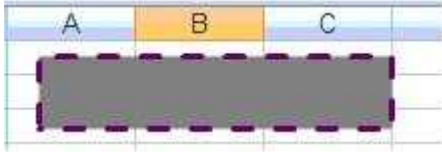
```
HSSFSheet sheet1 = hssfworkbook.CreateSheet("Sheet1");
HSSFPatriarch patriarch = sheet1.CreateDrawingPatriarch();
HSSFClientAnchor a1 = new HSSFClientAnchor(255, 125, 1023, 150, 0, 0, 2, 2);
HSSFShape rec1 = patriarch.CreateSimpleShape(a1);
//此处设置图形类型为矩形
rec1.ShapeType = HSSFShape.OBJECT_TYPE_RECTANGLE;
//设置填充色
rec1.SetFillColor(125, 125, 125);
//设置边框样式
rec1.LineStyle = HSSFShape.LINESTYLE_DASHGEL;
//设置边框宽度
rec1.LineWidth = 25400;
```



```
//设置边框颜色
```

```
rec1.SetLineStyleColor(100, 0, 100);
```

代码执行效果:



其中 SetFillColor 和 SetLineStyleColor 函数的三个参数分别是 RGB 三色值，具体表示什么颜色，找个 Photoshop 试试:)

关于 HSSFClientAnchor 参数说明、边框样式，边框宽度的说明可以参见前一博文:

<http://www.cnblogs.com/atao/archive/2009/09/13/1565645.html>

### 2.4.3 画圆形

前面我们学习了 NPOI 中的画简单直线和矩形的功能，今天我们一起学习一下它支持的另一种简单图形——圆形。同样，按照前面所讲的绘图“四步曲”:

1. 创建一个 Patriarch;
2. 创建一个 Anchor，以确定图形的位置;
3. 调用 Patriarch 创建图形;
4. 设置图形类型(直线，矩形，圆形等)及样式(颜色，粗细等)。

还是以例子加以说明:

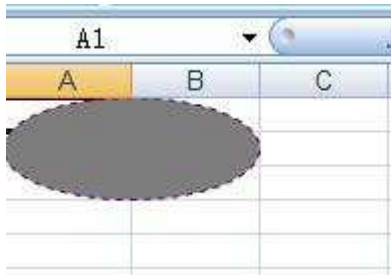
```
HSSFSheet sheet1 = hssfworkbook.CreateSheet("Sheet1");
HSSFPatriarch patriarch = sheet1.CreateDrawingPatriarch();
HSSFClientAnchor a1 = new HSSFClientAnchor(0, 0, 1023, 0, 0, 0, 1, 3);
HSSFShape rec1 = patriarch.CreateSimpleShape(a1);
rec1.ShapeType = HSSFShape.OBJECT_TYPE_OVAL;

rec1.SetFillColor(125, 125, 125);
rec1.LineStyle = HSSFShape.LINestyle_DASHGEL;
rec1.LineWidth = 12700;
rec1.SetLineStyleColor(100, 0, 100);
WriteToFile();
```

这里 rec1.ShapeType = HSSFShape.OBJECT\_TYPE\_OVAL;表示图形为椭圆。适当调整 HSSFClientAnchor 的各参数可以得到圆形。

关于 HSSFClientAnchor 构造函数和边框、填充色等前两节都有介绍，这里不再重述。详情情见：[画矩形](#) 和 [画线](#)。

上面代码执行生成的 Excel 如下：



#### 2.4.4 画 Grid

在 NPOI 中，本身没有画 Grid 的方法。但我们知道 Grid 其实就是由横线和竖线构成的，所以可以通过画线的方式来模拟画 Grid。

```
HSSFSheet sheet1 = hssfworkbook.CreateSheet("Sheet1");

HSSFRow row = sheet1.CreateRow(2);
row.CreateCell(1);
row.HeightInPoints = 240;
sheet1.SetColumnWidth(2, 9000);
int linesCount = 20;

HSSFPatriarch patriarch = sheet1.CreateDrawingPatriarch();
//因为 HSSFClientAnchor 中 dx 只能在 0-1023 之间, dy 只能在 0-255 之间, 所以这里采用
//比例的方式
double xRatio = 1023.0 / (linesCount*10);
double yRatio = 255.0 / (linesCount*10);

//画竖线
int x1 = 0;
int y1 = 0;
int x2 = 0;
int y2 = 200;
for (int i = 0; i < linesCount; i++)
{
    HSSFClientAnchor a2 = new HSSFClientAnchor();
    a2.SetAnchor((short)2, 2, (int)(x1 * xRatio), (int)(y1 * yRatio),
                (short)2, 2, (int)(x2 * xRatio), (int)(y2 * yRatio));
}
```



```

HSSFSSimpleShape shape2 = patriarch.CreateSimpleShape(a2);
shape2.ShapeType = (HSSFSSimpleShape.OBJECT_TYPE_LINE);

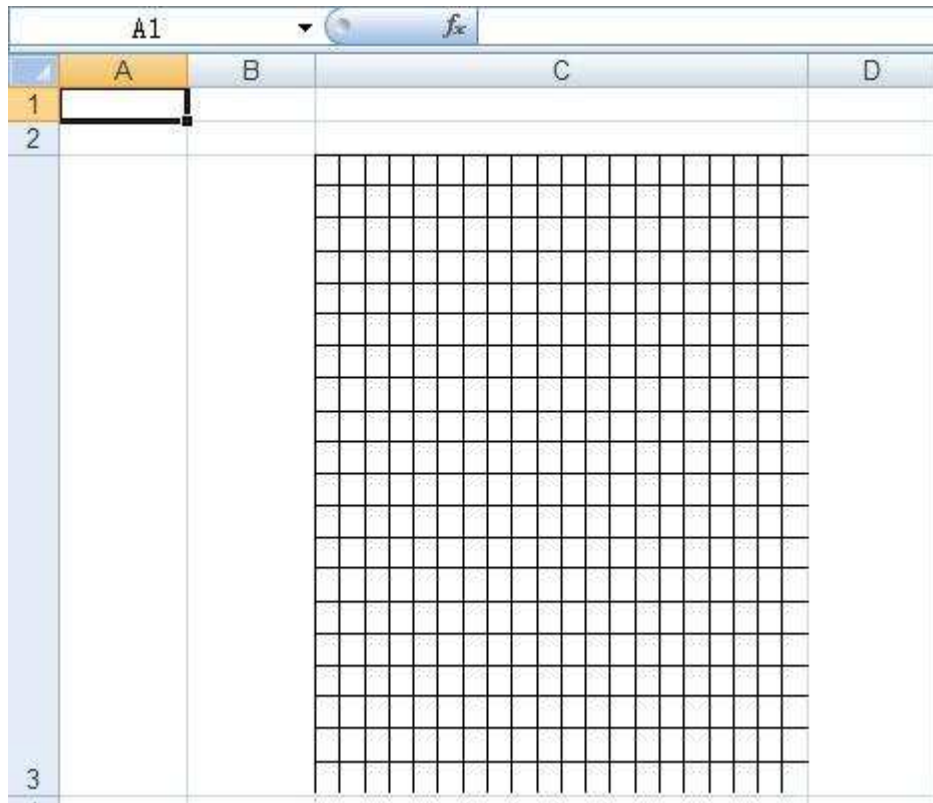
x1 += 10;
x2 += 10;
}

//画横线
x1 = 0;
y1 = 0;
x2 = 200;
y2 = 0;
for (int i = 0; i < linesCount; i++)
{
    HSSFClientAnchor a2 = new HSSFClientAnchor();
    a2.SetAnchor((short)2, 2, (int)(x1 * xRatio), (int)(y1 * yRatio),
                (short)2, 2, (int)(x2 * xRatio), (int)(y2 * yRatio));
    HSSFSSimpleShape shape2 = patriarch.CreateSimpleShape(a2);
    shape2.ShapeType = (HSSFSSimpleShape.OBJECT_TYPE_LINE);

    y1 += 10;
    y2 += 10;
}

```

请注意 HSSFClientAnchor 对象中的 dx 只能取 0-1023 之间的数，dy 只能取 0-255 之间的数。我们可以理解为是将单元格的宽和高平分成了 1023 和 255 份，设置 dx 和 dy 时相当于按比例取对应的座标。最终生成的 Excel 如下：



#### 2.4.5 插入图片

我们知道，在 Excel 中是可以插入图片的。操作菜单是“插入->图片”，然后选择要插入图片，可以很容易地在 Excel 插入图片。同样，在 NPOI 中，利用代码也可以实现同样的效果。在 NPOI 中插入图片的方法与画图的方法有点类似：

```
//add picture data to this workbook.  
byte[] bytes = System.IO.File.ReadAllBytes(@"D:\MyProject\NPOIDemo\ShapeImage\  
image1.jpg");  
int pictureIdx = hssfworkbook.AddPicture(bytes, HSSFWorkbook.PICTURE_TYPE_JPEG  
);  
  
//create sheet  
HSSFSheet sheet = hssfworkbook.CreateSheet("Sheet1");  
  
// Create the drawing patriarch. This is the top level container for all shap  
es.  
HSSFPatriarch patriarch = sheet.CreateDrawingPatriarch();  
  
//add a picture
```

```
HSSFClientAnchor anchor = new HSSFClientAnchor(0, 0, 1023, 0, 0, 0, 1, 3);  
HSSFPicture pict = patriarch.CreatePicture(anchor, pictureIdx);
```

与画简单图形不同的是，首先要将图片读入到 byte 数组，然后添加到 workbook 中；最后调用的是 `patriarch.CreatePicture(anchor, pictureIdx)` 方法显示图片，而不是 `patriarch.CreateSimpleShape(anchor)` 方法。上面这段代码执行后生成的 Excel 文件样式如下：



我们发现，插入的图片被拉伸填充在 `HSSFClientAnchor` 指定的区域。有时可能我们并不需要拉伸的效果，怎么办呢？很简单，在最后加上这样一句用来自动调节图片大小：

```
pict.Resize();
```

添加代码后再执行上述代码，生成的 Excel 样式如下：



图片已经自动伸缩到原始大小了。

## 2.5 打印相关设置

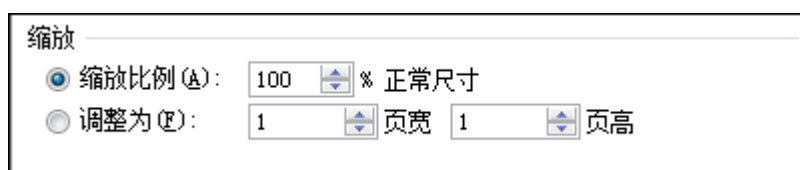
打印设置主要包括方向设置、缩放、纸张设置、页边距等。NPOI 1.2 支持大部分打印属性，能够让你轻松满足客户的打印需要。

首先是方向设置，Excel 支持两种页面方向，即纵向和横向。



在 NPOI 中如何设置呢？你可以通过 `HSSFWorkbook.PrintSetup.Landscape` 来设置，`Landscape` 是布尔类型的，在英语中是横向的意思。如果 `Landscape` 等于 `true`，则表示页面方向为横向；否则为纵向。

接着是缩放设置，



这里的缩放比例对应于 `HSSFWorkbook.PrintSetup.Scale`，而页宽和页高分别对应于 `HSSFWorkbook.PrintSetup.FitWidth` 和 `HSSFWorkbook.PrintSetup.FitHeight`。要注意的是，这里的 `PrintSetup.Scale` 应该被设置为 0-100 之间的值，而不是小数。



接下来就是纸张设置了，对应于 `HSSFWorkbook.PrintSetup.PaperSize`，但这里的 `PaperSize` 并不是随便设置的，而是由一些固定的值决定的，具体的值与对应的纸张如下表所示：

值	纸张
1	US Letter 8 1/2 x 11 in
2	US Letter Small 8 1/2 x 11 in
3	US Tabloid 11 x 17 in
4	US Ledger 17 x 11 in
5	US Legal 8 1/2 x 14 in
6	US Statement 5 1/2 x 8 1/2 in
7	US Executive 7 1/4 x 10 1/2 in
8	A3 297 x 420 mm
9	A4 210 x 297 mm
10	A4 Small 210 x 297 mm
11	A5 148 x 210 mm
12	B4 (JIS) 250 x 354
13	B5 (JIS) 182 x 257 mm

14 Folio 8 1/2 x 13 in  
15 Quarto 215 x 275 mm  
16 10 x 14 in  
17 11 x 17 in  
18 US Note 8 1/2 x 11 in  
19 US Envelope #9 3 7/8 x 8 7/8  
20 US Envelope #10 4 1/8 x 9 1/2  
21 US Envelope #11 4 1/2 x 10 3/8  
22 US Envelope #12 4 1/2 x 11  
23 US Envelope #14 5 x 11 1/2  
24 C size sheet  
25 D size sheet  
26 E size sheet  
27 Envelope DL 110 x 220mm  
28 Envelope C5 162 x 229 mm  
29 Envelope C3 324 x 458 mm  
30 Envelope C4 229 x 324 mm  
31 Envelope C6 114 x 162 mm  
32 Envelope C65 114 x 229 mm  
33 Envelope B4 250 x 353 mm  
34 Envelope B5 176 x 250 mm  
35 Envelope B6 176 x 125 mm  
36 Envelope 110 x 230 mm  
37 US Envelope Monarch 3.875 x 7.5 in  
38 6 3/4 US Envelope 3 5/8 x 6 1/2 in  
39 US Std Fanfold 14 7/8 x 11 in  
40 German Std Fanfold 8 1/2 x 12 in  
41 German Legal Fanfold 8 1/2 x 13 in  
42 B4 (ISO) 250 x 353 mm  
43 Japanese Postcard 100 x 148 mm  
44 9 x 11 in  
45 10 x 11 in  
46 15 x 11 in  
47 Envelope Invite 220 x 220 mm  
48 RESERVED--DO NOT USE

49 RESERVED--DO NOT USE  
50 US Letter Extra 9 \275 x 12 in  
51 US Legal Extra 9 \275 x 15 in  
52 US Tabloid Extra 11.69 x 18 in  
53 A4 Extra 9.27 x 12.69 in  
54 Letter Transverse 8 \275 x 11 in  
55 A4 Transverse 210 x 297 mm  
56 Letter Extra Transverse 9\275 x 12 in  
57 SuperA/SuperA/A4 227 x 356 mm  
58 SuperB/SuperB/A3 305 x 487 mm  
59 US Letter Plus 8.5 x 12.69 in  
60 A4 Plus 210 x 330 mm  
61 A5 Transverse 148 x 210 mm  
62 B5 (JIS) Transverse 182 x 257 mm  
63 A3 Extra 322 x 445 mm  
64 A5 Extra 174 x 235 mm  
65 B5 (ISO) Extra 201 x 276 mm  
66 A2 420 x 594 mm  
67 A3 Transverse 297 x 420 mm  
68 A3 Extra Transverse 322 x 445 mm  
69 Japanese Double Postcard 200 x 148 mm  
70 A6 105 x 148 mm  
71 Japanese Envelope Kaku #2  
72 Japanese Envelope Kaku #3  
73 Japanese Envelope Chou #3  
74 Japanese Envelope Chou #4  
75 Letter Rotated 11 x 8 1/2 11 in  
76 A3 Rotated 420 x 297 mm  
77 A4 Rotated 297 x 210 mm  
78 A5 Rotated 210 x 148 mm  
79 B4 (JIS) Rotated 364 x 257 mm  
80 B5 (JIS) Rotated 257 x 182 mm  
81 Japanese Postcard Rotated 148 x 100 mm  
82 Double Japanese Postcard Rotated 148 x 200 mm  
83 A6 Rotated 148 x 105 mm

84 Japanese Envelope Kaku #2 Rotated  
85 Japanese Envelope Kaku #3 Rotated  
86 Japanese Envelope Chou #3 Rotated  
87 Japanese Envelope Chou #4 Rotated  
88 B6 (JIS) 128 x 182 mm  
89 B6 (JIS) Rotated 182 x 128 mm  
90 12 x 11 in  
91 Japanese Envelope You #4  
92 Japanese Envelope You #4 Rotated  
93 PRC 16K 146 x 215 mm  
94 PRC 32K 97 x 151 mm  
95 PRC 32K(Big) 97 x 151 mm  
96 PRC Envelope #1 102 x 165 mm  
97 PRC Envelope #2 102 x 176 mm  
98 PRC Envelope #3 125 x 176 mm  
99 PRC Envelope #4 110 x 208 mm  
100 PRC Envelope #5 110 x 220 mm  
101 PRC Envelope #6 120 x 230 mm  
102 PRC Envelope #7 160 x 230 mm  
103 PRC Envelope #8 120 x 309 mm  
104 PRC Envelope #9 229 x 324 mm  
105 PRC Envelope #10 324 x 458 mm  
106 PRC 16K Rotated  
107 PRC 32K Rotated  
108 PRC 32K(Big) Rotated  
109 PRC Envelope #1 Rotated 165 x 102 mm  
110 PRC Envelope #2 Rotated 176 x 102 mm  
111 PRC Envelope #3 Rotated 176 x 125 mm  
112 PRC Envelope #4 Rotated 208 x 110 mm  
113 PRC Envelope #5 Rotated 220 x 110 mm  
114 PRC Envelope #6 Rotated 230 x 120 mm  
115 PRC Envelope #7 Rotated 230 x 160 mm  
116 PRC Envelope #8 Rotated 309 x 120 mm  
117 PRC Envelope #9 Rotated 324 x 229 mm  
118 PRC Envelope #10 Rotated 458 x 324 mm

(此表摘自《Excel Binary File Format (.xls) Structure Specification.pdf》)

HSSFSheet 下面定义了一些 xxxx\_PAPERSIZE 的常量，但都是非常常用的纸张大小，如果满足不了你的需要，可以根据上表自己给 PaperSize 属性赋值。所以，如果你要设置纸张大小可以用这样的代码：

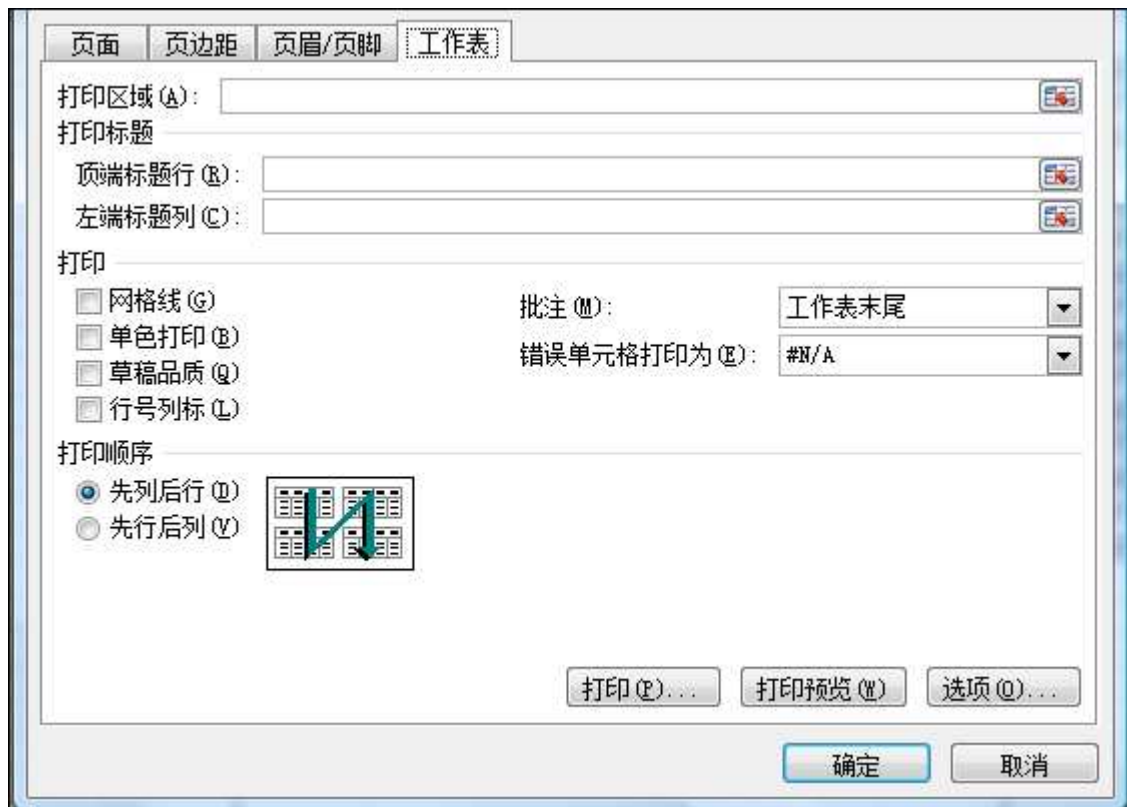
```
HSSFSheet.PrintSetup.PaperSize=HSSFSheet.A4_PAPERSIZE;
```

或

```
HSSFSheet.PrintSetup.PaperSize=9; (A4 210*297mm)
```

起始页码 (R): 自动

再下来就是打印的起始页码，它对应于 HSSFSheet.PrintSetup.PageStart 和 HSSFSheet.PrintSetup.UsePage，如果 UsePage=false，那么就相当于“自动”，这时 PageStart 不起作用；如果 UsePage=true，PageStart 才会起作用。所以在设置 PageStart 之前，必须先把 UsePage 设置为 true。



“打印”栏中的“网格线”设置对应于 HSSFSheet.IsPrintGridlines，请注意，这里不是 HSSFSheet.PrintSetup 下面，所以别搞混了。这里之所以不隶属于 PrintSetup 是由底层存储该信息的 record 决定的，底层是把 IsGridsPrinted 放在 GridsetRecord 里面的，而不是 PrintSetupRecord 里面的，尽管界面上是放在一起的。另外还有一个



HSSFSheet.IsGridsPrinted 属性，这个属性对应于底层的 gridset Record，但这个 record 是保留的，从微软的文档显示没有任何意义，所以这个属性请不要去设置。

“单色打印”则对应于 HSSFSheet.PrintSetup.NoColors，这是布尔类型的，值为 true 时，表示单色打印。

“草稿品质”对应于 HSSFSheet.PrintSetup.IsDraft，也是布尔类型的，值为 true 时，表示用草稿品质打印。

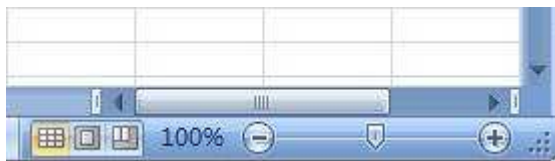
这里的打印顺序是由 HSSFSheet.PrintSetup.LeftToRight 决定的，它是布尔类型的，当为 true 时，则表示“先行后列”；如果是 false，则表示“先列后行”。

在 NPOI 1.2 中，“行号列标”、“批注”和“错误单元格打印为”、“页边距”暂不支持，将在以后的版本中支持。

## 2.6 高级功能

### 2.6.1 调整表单显示比例

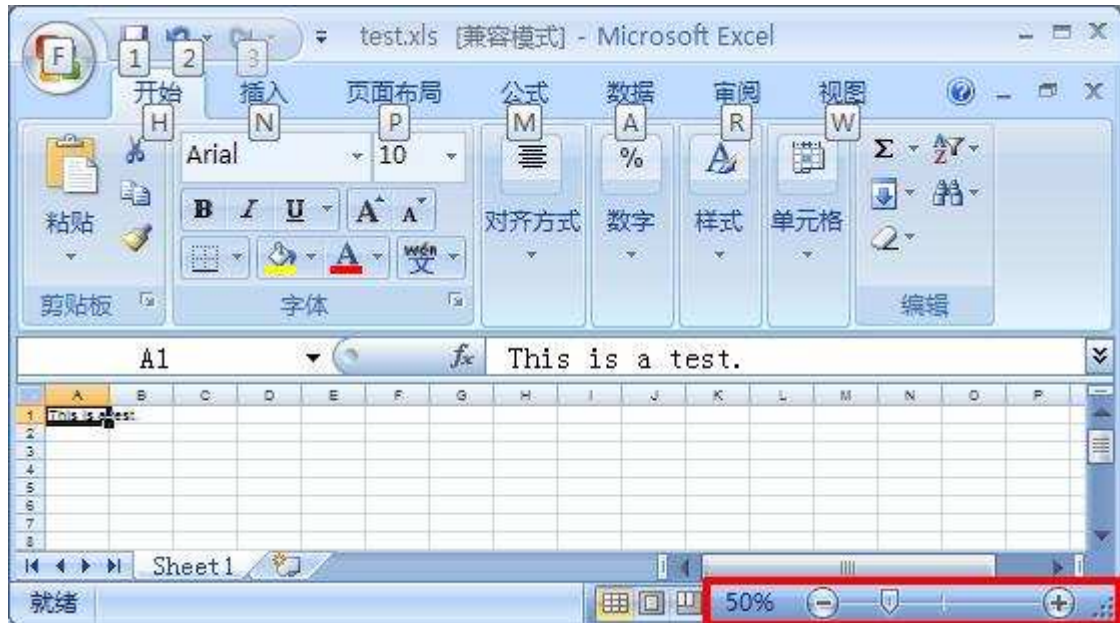
在 Excel 中，可以通过调整右下角的滚动条来调整 Sheet 的显示比例。如图：



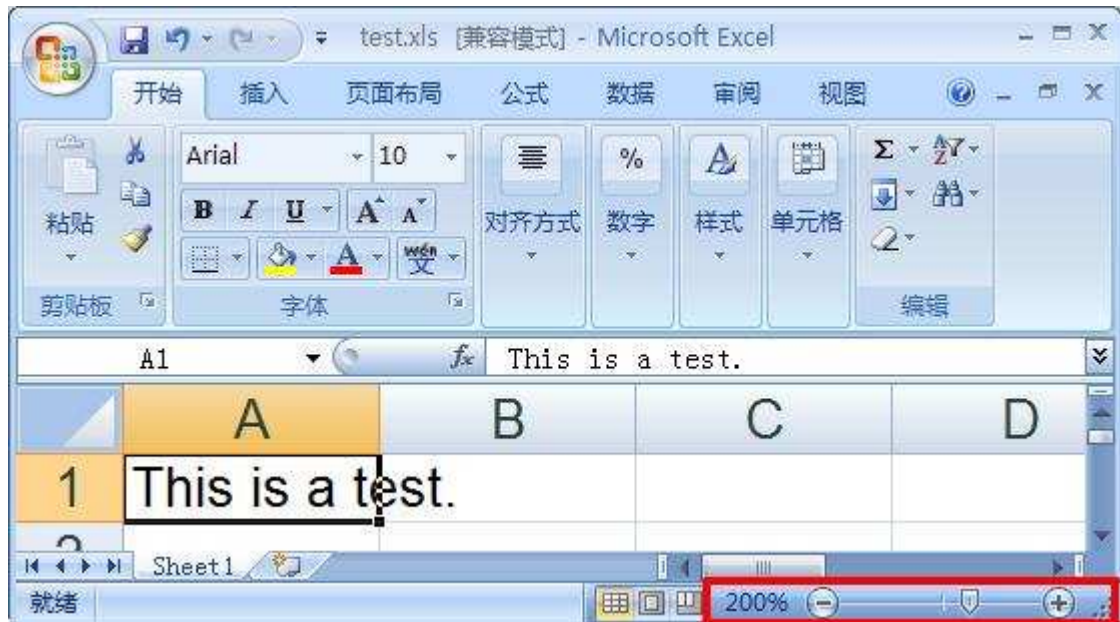
在 NPOI 中，也能通过代码实现这样的功能，并且代码非常简单：

```
HSSFSheet sheet1 = hssfworkbook.CreateSheet("Sheet1");
sheet1.CreateRow(0).CreateCell(0).SetCellValue("This is a test.");
//50% zoom
sheet1.SetZoom(1,2);
```

我们发现，SetZoom 有两个参数。其中第一个参数表示缩放比例的分子，第二个参数表示缩放比例的分母，所以 SetZoom(1,2) 就表示缩小到 1/2，也就是 50%。代码执行后生成的 Excel 样式如下：

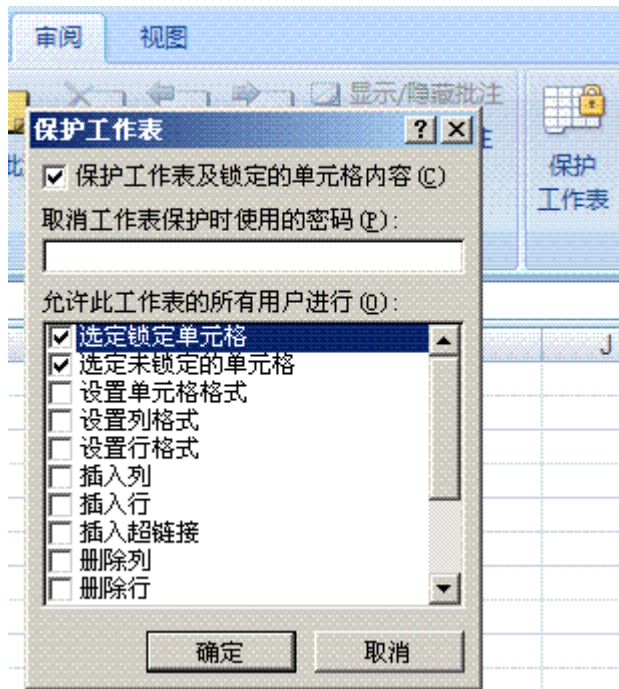


如果将 SetZoom 的参数改成 (2, 1)，代码执行后生成的 Excel 样式如下，表示扩大两倍：



## 2.6.2 设置密码

有时，我们可能需要某些单元格只读，如在做模板时，模板中的数据是不能随意让别人改的。在 Excel 中，可以通过“审阅->保护工作表”来完成，如下图：



那么，在 NPOI 中有没有办法通过编码的方式达到这一效果呢？答案是肯定的。

```
HSSFSheet sheet1 = hssfworkbook.CreateSheet("Sheet1");
```

```
HSSFRow row1 = sheet1.CreateRow(0);
```

```
HSSFCell cell1 = row1.CreateCell(0);
```

```
HSSFCell cel2 = row1.CreateCell(1);
```

```
HSSFCellStyle unlocked = hssfworkbook.CreateCellStyle();
```

```
unlocked.IsLocked = false;
```

```
HSSFCellStyle locked = hssfworkbook.CreateCellStyle();
```

```
locked.IsLocked = true;
```

```
cell1.SetCellValue("没被锁定");
```

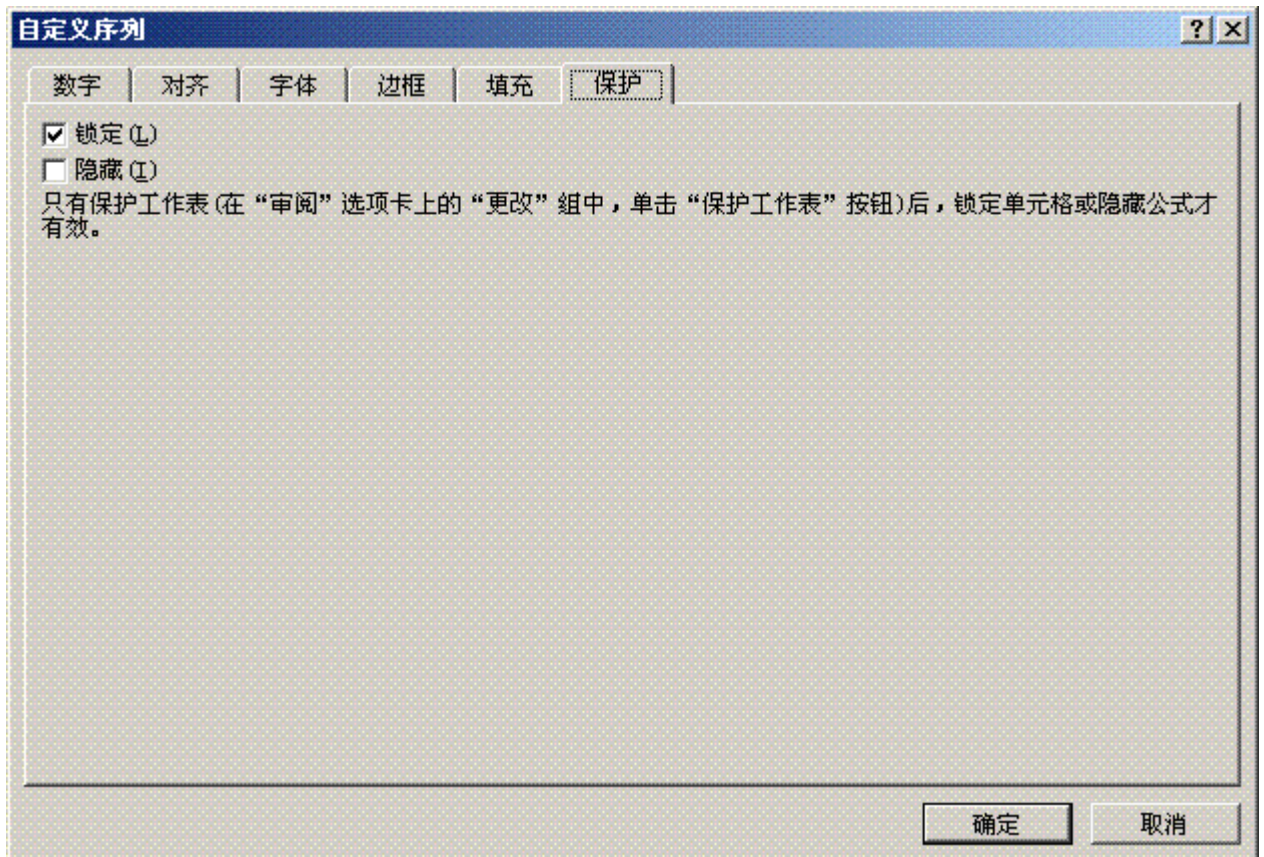
```
cell1.CellStyle = unlocked;
```

```
cel2.SetCellValue("被锁定");
```

```
cel2.CellStyle = locked;
```

```
sheet1.ProtectSheet("password");
```

正如代码中所看到的，我们通过设置 CellStyle 的 IsLocked 为 True，表示此单元格将被锁定。相当于在 Excel 中执行了如下操作：

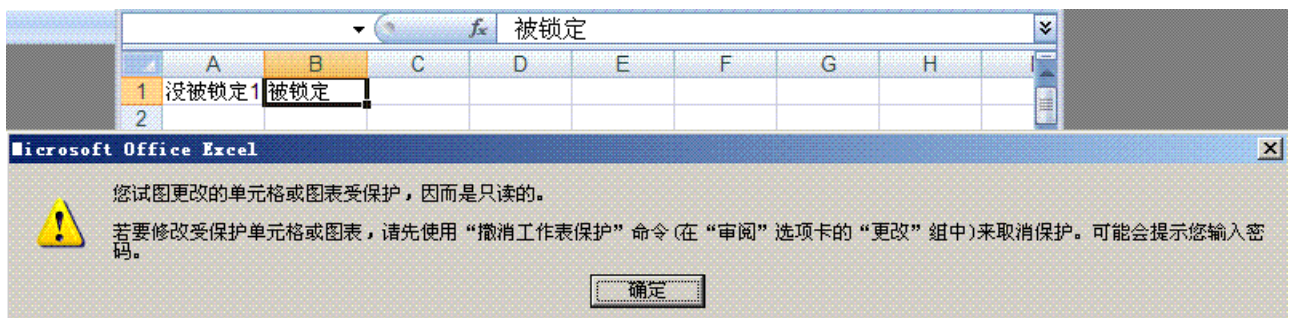


然后通过 ProtectSheet 设置密码。

执行结果如下:

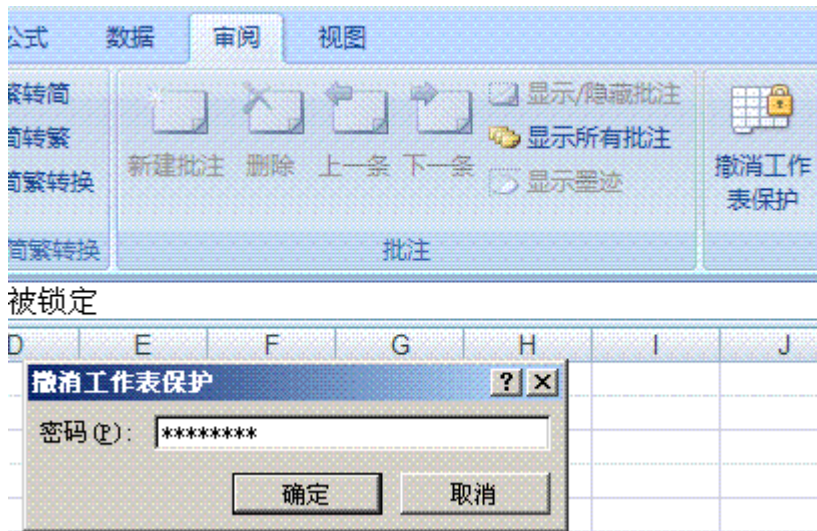


没被锁定的列可以任意修改。



被锁定的列不能修改。





输入密码可以解除锁定。

### 2.6.3 显示/隐藏 Excel 网格线

有些时候，我们需要网格线，而有些时候我们不需要，这取决于实际的业务需求。前两天 inmejin 兄就问我，怎么把网格给去掉，因为他们要把 Excel 文档当 Word 使，也许是因为 Excel 排版方便吧。

Excel 中的网格线设置是以表 (Sheet) 为单位进行管理的，这也就意味着你可以让一个表显示网格线，而另一个表不显示，这是不冲突的。

在 Excel 2007 中，我们通常用“工作表选项”面板来设置这个属性：



在面板中，你会发现有 2 个多选框，一个是查看，一个是打印，也就是说 Excel 是把查看和打印网格线作为两个设置来处理的，存储的 Record 也是不同的。

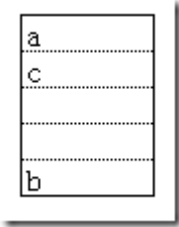
在 NPOI 中，如果要让网格线在查看时显示/隐藏，你可以 `HSSFWorksheet.DisplayGridlines` 属性，默认值为 `true`（这也是为什么默认情况下我们能够看到网格线）。下面的代码就是让网格线在查看时不可见的：

```
HSSFWorkbook hssfworkbook = new HSSFWorkbook();  
HSSFWorksheet s1 = hssfworkbook.CreateSheet("Sheet1");  
s1.DisplayGridlines = false;
```

如果要在打印时显示/隐藏网格线，你可以用 `HSSFSheet.IsGridlinesPrinted` 属性，默认值为 `false`（这就是默认情况下打印看不到网格线的原因）。代码和上面差不多：

```
s1.IsGridsPrinted = true;
```

上面的代码将在打印时显示网格线，打印的效果如下所示。

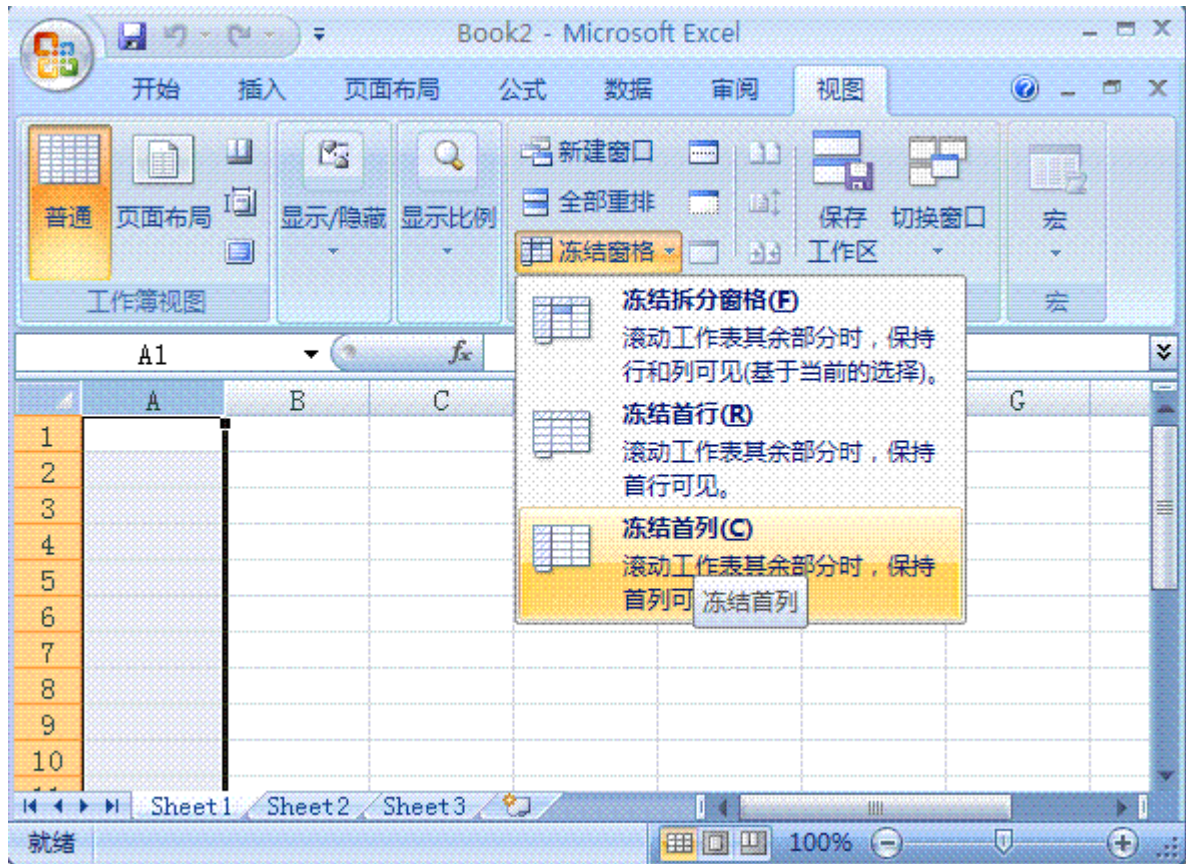


在此也提醒大家，如果这个 Excel 最终客户有打印意向，可别忘了把 `IsGridPrinted` 属性也设置上。

相关范例可以参考 NPOI 1.2 正式版中的 `DisplayGridlinesInXls` 项目。

#### 2.6.4 锁定列

在 Excel 中，有时可能会出现列数太多或是行数太多的情况，这时可以通过锁定列来冻结部分列，不随滚动条滑动，方便查看。在 Excel 中设置冻结列的方法如下：

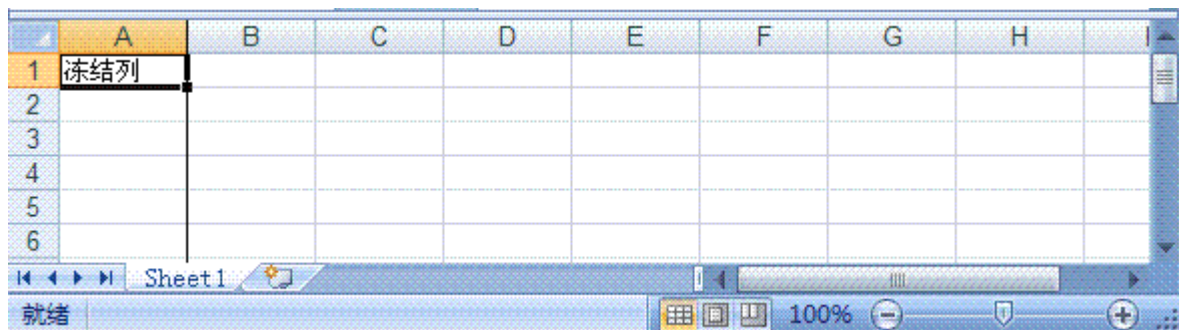


同样，利用 NPOI，通过代码也能实现上面的效果：

```
HSSFSheet sheet1 = hssfworkbook.CreateSheet("Sheet1");
```

```
HSSFRow row1 = sheet1.CreateRow(0);  
row1.CreateCell(0).SetCellValue("冻结列");  
sheet1.CreateFreezePane(1, 0, 1, 0);
```

代码执行结果如下：



下面对 CreateFreezePane 的参数作一下说明：

第一个参数表示要冻结的列数；

第二个参数表示要冻结的行数，这里只冻结列所以为 0；

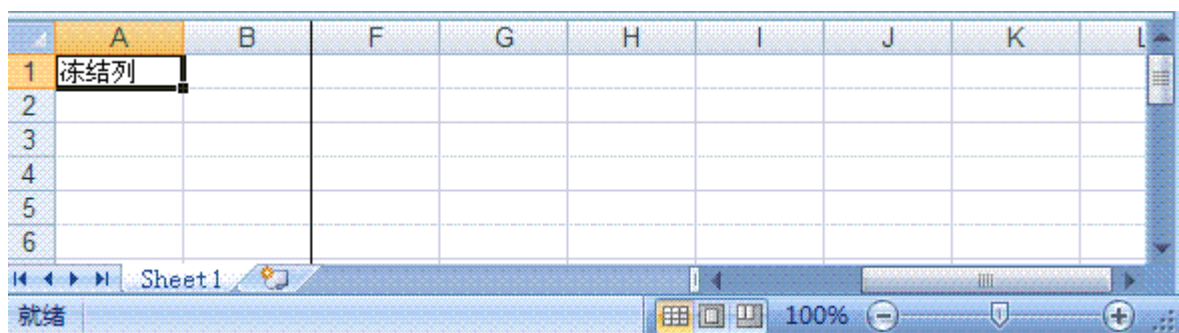
第三个参数表示右边区域可见的首列序号，从 1 开始计算；

第四个参数表示下边区域可见的首行序号，也是从 1 开始计算，这里是冻结列，所以为 0；

举例说明也许更好理解，将各参数设置为如下：

```
sheet1.CreateFreezePane(2, 0, 5, 0);
```

得到的效果如下图：

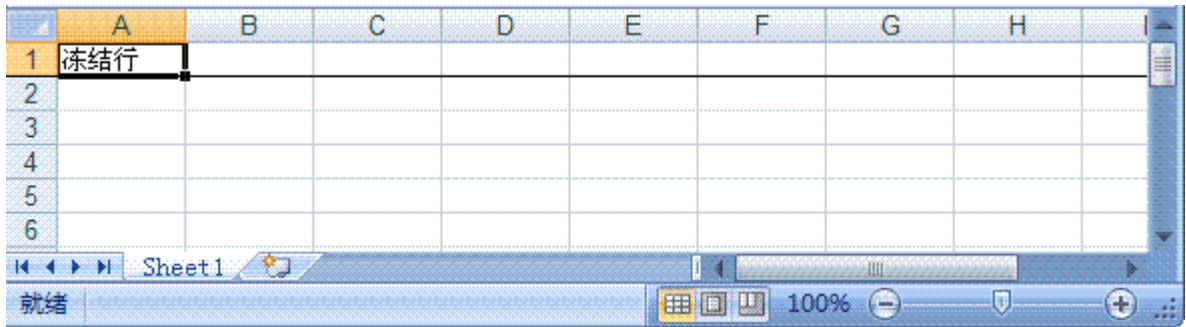


注意图中 C、D 和 E 列默认是看不到的，滚动才看得到，这就是第三个参数 5 起了作用，是不是很好理解了呢：)

接下来，看一下冻结行的效果。将上面的代码稍作修改：

```
HSSFSheet sheet1 = hssfworkbook.CreateSheet("Sheet1");  
HSSFRow row1 = sheet1.CreateRow(0);  
row1.CreateCell(0).SetCellValue("冻结行");  
sheet1.CreateFreezePane(0, 1, 0, 1);
```

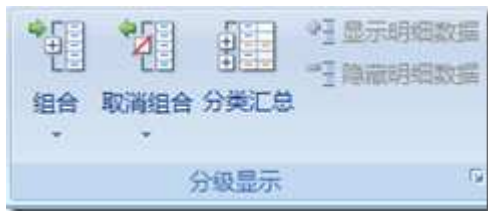
执行后生成的 Excel 文件效果见下图：



那么，如果要行和列同时冻结该怎么做呢？聪明的你一定能想得到，呵呵~~

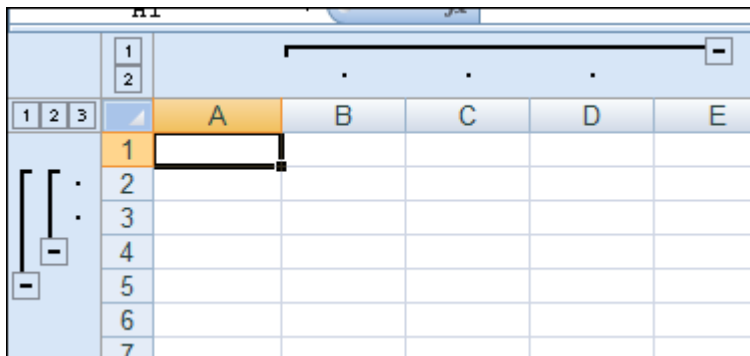
### 2.6.5 组合行、列

Excel 2007 中有一个面板是专门用于设置组合功能的，叫做“分级显示”面板，如下所示：



可能我们在过去生成 Excel 文件的时候根本不会用这个功能，也没办法用，因为 cvs 法和 html 法没办法控制这些东西。这里简单的介绍一下什么叫做组合：

组合分为行组合和列组合，所谓行组合，就是让 n 行组合成一个集合，能够进行展开和合拢操作，在 Excel 中显示如下：



图中左侧就是用于控制行组合折叠的图标，图中上部就是用于控制列组合的，是不是有点像 TreeView 中的折叠节点？很多时候由于数据太多，为了让用户对于大量数据一目了然，我们可以使用行列组合来解决显示大纲，这和 Visual Studio 里面的 region 的概念是类似的。

细心的朋友可能已经注意到了，我们其实可以对一行做多次组合操作，这就是分级显示的概念，图中就把行 2-3 分为 2 个组合，第 2 行到第 4 行为一个组合，第 2 行到第 5 行一个组合，所以是分两级。



在 NPOI 中，要实现分组其实并不难，你只需要调用 `HSSFSSheet.GroupRow` 和 `HSSFSSheet.GroupColumn` 这两个方法就可以了。

首先我们来看 `HSSFSSheet.GroupRow`，`GroupRow` 有 2 个参数，分别是 `fromRow` 和 `toRow`，表示起始行号和结束行号，这些行号都是从 0 开始算起的。

```
HSSFWorkbook hssfworkbook = new HSSFWorkbook();
HSSFSSheet s = hssfworkbook.CreateSheet("Sheet1");
s.GroupRow(1, 3);
```

上面的代码把第 2 行到第 4 行做了组合。

要组合列，其实代码很相似，如下所示：

```
s.GroupColumn(1, 3)
```

上面的代码把 B 至 D 列做了组合。

正如上图中 Excel 的“分级显示”面板所示，有“组合”，也一定有“取消组合”，NPOI 中你可以用 `HSSFSSheet.UngroupRow` 和 `HSSFSSheet.UngroupColumn`，参数和 `GroupXXX` 是一样的，如果要取消第 2 到第 4 行的组合，就可以用下面的代码：

```
s.UngroupColumn(1, 3)
```

相关范例请见 [NPOI 1.2 正式版](#) 中的 `GroupRowAndColumnInXls` 项目。

## 2.6.6 设置初始视图的行、列

有些时候，我们可能希望生成的 Excel 文件在被打开的时候自动将焦点定位在某个单元格或是选中某个区域中。在 NPOI 中可以通过 `SetAsActiveCell` 和 `SetActiveCellRange` 等几个方法实现。

首先我们看一下设置初始视图中选中某个单元格的方法：

```
//use HSSFCell.SetAsActiveCell() to select B6 as the active column
HSSFSSheet sheet1 = hssfworkbook.CreateSheet("Sheet A");
CreateCellArray(sheet1);
sheet1.GetRow(5).GetCell(1).SetAsActiveCell();
//set TopRow and LeftCol to make B6 the first cell in the visible area
sheet1.TopRow = 5;
sheet1.LeftCol = 1;
```

其中 `CreateCellArray(sheet1)` 方法用来写示范数据，其代码为（下同）：

```
static void CreateCellArray(HSSFSSheet sheet)
{
    for (int i = 0; i < 300; i++)
```

```

{
    HSSFRow row=sheet.CreateRow(i);
    for (int j = 0; j < 150; j++)
    {
        HSSFCell cell = row.CreateCell(j);
        cell.SetCellValue(i*j);
    }
}
}

```

生成的 Excel 打开时效果如下，注意 B6 为默认选中状态，TopRow 和 LeftCol 设置 B6 为当前可见区域的第一个单元格：

	B	C	D	E	F
6	5	10	15	20	25
7	6	12	18	24	30
8	7	14	21	28	35
9	8	16	24	32	40
10	9	18	27	36	45
11	10	20	30	40	50
12	11	22	33	44	55
13	12	24	36	48	60

如果不设置 TopRow 和 LeftCol 属性，默认的可见域的第一个单元格为 A1，如下是另一种设置活动单元格的方法，但没有设置此 Sheet 的 TopRow 和 LeftCol：

```

HSSFSheet sheet2 = hssfworkbook.CreateSheet("Sheet B");
sheet2.Sheet.SetActiveCell(1, 5);

```

对应生成的 Excel 显示为：

	A	B	C	D	E	F	G
1							
2							
3							
4							
5							
6							
7							
8							

除了设置某个单元格为选中状态外，还 NPOI 可以设置某个区域为选中状态：

```

//use Sheet.SetActiveCellRange to select a cell range

```

```
HSSFSheet sheet3 = hssfworkbook.CreateSheet("Sheet C");
CreateCellArray(sheet3);
sheet3.Sheet.SetActiveCellRange(2, 5, 1, 5);
```

以上代码设置了 Sheet C 的选中区域为 B3:F6:

	A	B	C	D	E	F	G	H
1	0	0	0	0	0	0	0	0
2	0	1	2	3	4	5	6	7
3	0	2	4	6	8	10	12	14
4	0	3	6	9	12	15	18	21
5	0	4	8	12	16	20	24	28
6	0	5	10	15	20	25	30	35
7	0	6	12	18	24	30	36	42
8	0	7	14	21	28	35	42	49
9	0	8	16	24	32	40	48	56
10	0	9	18	27	36	45	54	63
11	0	10	20	30	40	50	60	70
12	0	11	22	33	44	55	66	77
13	0	12	24	36	48	60	72	84

还有更强大的，设置多个选中区域:

```
//use Sheet.SetActiveCellRange to select multiple cell ranges
HSSFSheet sheet4 = hssfworkbook.CreateSheet("Sheet D");
CreateCellArray(sheet4);
List<CellRangeAddress8Bit> cellranges = new List<CellRangeAddress8Bit>();
cellranges.Add(new CellRangeAddress8Bit(1, 3, 2, 5));
cellranges.Add(new CellRangeAddress8Bit(6, 7, 8, 9));
sheet4.Sheet.SetActiveCellRange(cellranges, 1, 6, 9);
```

	A	B	C	D	E	F	G	H	I	J
1	0	0	0	0	0	0	0	0	0	0
2	0	1	2	3	4	5	6	7	8	9
3	0	2	4	6	8	10	12	14	16	18
4	0	3	6	9	12	15	18	21	24	27
5	0	4	8	12	16	20	24	28	32	36
6	0	5	10	15	20	25	30	35	40	45
7	0	6	12	18	24	30	36	42	48	54
8	0	7	14	21	28	35	42	49	56	63
9	0	8	16	24	32	40	48	56	64	72
10	0	9	18	27	36	45	54	63	72	81
11	0	10	20	30	40	50	60	70	80	90
12	0	11	22	33	44	55	66	77	88	99
13	0	12	24	36	48	60	72	84	96	108

如果一个 Excel 文件中有多个 Sheet，还可以通过如下语句设置打开时的初始 Sheet：

```
hssfworkbook.ActiveSheetIndex = 2;
```

## 2.6.7 数据有效性

在有些情况下（比如 Excel 引入），我们可能不允许用户在 Excel 随意输入一些无效数据，这时就要在模板中加一些数据有效性的验证。在 Excel 中，设置数据有效性的方步骤如下：

(1) 先选定一个区域；

“数据有效性”中设置数据有效性验证（如图）。(2) 在菜单“数据



同样，利用 NPOI，用代码也可以实现：

```
HSSFSheet sheet1 = hssfworkbook.CreateSheet("Sheet1");  
  
sheet1.CreateRow(0).CreateCell(0).SetCellValue("日期列");  
CellRangeAddressList regions1 = new CellRangeAddressList(1, 65535, 0, 0);  
DVConstraint constraint1 = DVConstraint.CreateDateConstraint(DVConstraint.OperatorType.BETWEEN, "1900-01-01", "2999-12-31", "yyyy-MM-dd");  
HSSFDataValidation dataValidate1 = new HSSFDataValidation(regions1, constraint1);  
dataValidate1.CreateErrorBox("error", "You must input a date.");  
sheet1.AddValidationData(dataValidate1);
```

上面是一个在第一列要求输入 1900-1-1 至 2999-12-31 之间日期的有效性验证的例子，生成的 Excel 效果如下，当输入非法时将给出警告：



下面对刚才用到的几个方法加以说明：

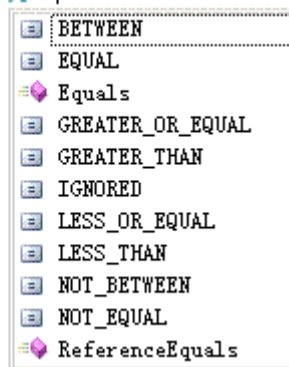
CellRangeAddressList 类表示一个区域，构造函数中的四个参数分别表示起始行序号，终止行序号，起始列序号，终止列序号。所以第一列所在区域就表示为：

//所有序号都从零算起，第一行标题行除外，所以第一个参数是 1，65535 是一个 Sheet 的最大行数

```
new CellRangeAddressList(1, 65535, 0, 0);
```

另外，CreateDateConstraint 的第一个参数除了设置成 DVConstraint.OperatorType.BETWEEN 外，还可以设置成如下一些值，大家可以自己一个个去试看看效果：

DVConstraint.OperatorType.



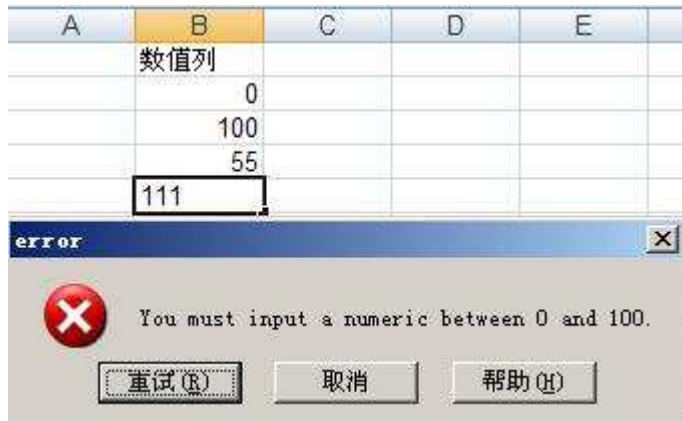
最后，dataValidate1.CreateErrorBox(title, text)，用来创建出错时的提示信息。第一个参数表示提示框的标题，第二个参数表示提示框的内容。

理解了上面这些，创建一个整数类型的有效性验证也不难实现：

```
sheet1.CreateRow(0).CreateCell(1).SetCellValue("数值列");
CellRangeAddressList regions2 = new CellRangeAddressList(1, 65535, 1, 1);
DVConstraint constraint2 = DVConstraint.CreateNumericConstraint(DVConstraint.ValidationType.INTEGER, DVConstraint.OperatorType.BETWEEN, "0", "100");
HSSFDataValidation dataValidate2 = new HSSFDataValidation(regions2, constraint2);
dataValidate2.CreateErrorBox("error", "You must input a numeric between 0 and 100.");
```

sheet1.AddValidationData(dataValidate2);

生成的 Excel 效果为：



下一节我们将学习利用数据有效性创建下拉列表的例子。

## 2.6.8 生成下拉列表

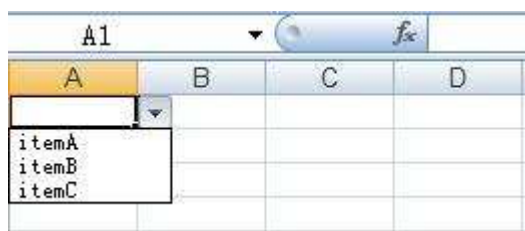
上一节我们讲了简单的数据有效性验证,这一节我们学习一下数据有效性的另一个应用——下拉列表。在 Excel 中,并没有类似 Web 中的下拉控件,其下拉效果是通过数据有效性来实现的。设置步骤为:

- (1) 选定一个要生成下拉列表的区域;
- (2) 设置数据有效性为序列,并在来源中填充可选下拉的值,用“,”隔开(如图)。



对应的效果为：





同样，利用 NPOI 代码也可以实现上面的效果：

```
HSSFSheet sheet1 = hssfworkbook.CreateSheet("Sheet1");

CellRangeAddressList regions = new CellRangeAddressList(0, 65535, 0, 0);
DVConstraint constraint = DVConstraint.CreateExplicitListConstraint(new string
[] { "itemA", "itemB", "itemC" });
HSSFDataValidation dataValidate = new HSSFDataValidation(regions, constraint);
sheet1.AddValidationData(dataValidate);
```

下面对代码作一下简要说明：

先设置一个需要提供下拉的区域，关于 CellRangeAddressList 构造函数参数的说明请参见[上一节](#)：

```
CellRangeAddressList regions = new CellRangeAddressList(0, 65535, 0, 0);
```

然后将下拉项作为一个数组传给 CreateExplicitListConstraint 作为参数创建一个约束，根据要控制的区域和约束创建数据有效性就可以了。

但是这样会有一个问题：Excel 中允许输入的序列来源长度最大为 255 个字符，也就是说当下拉项的总字符串长度超过 255 是将会出错。那么如果下拉项很多的情况下应该怎么处理呢？答案是通过引用的方式。步骤如下：

先创建一个 Sheet 专门用于存储下拉项的值，并将各下拉项的值写入其中：

```
HSSFSheet sheet2 = hssfworkbook.CreateSheet("ShtDictionary");
sheet2.CreateRow(0).CreateCell(0).SetCellValue("itemA");
sheet2.CreateRow(1).CreateCell(0).SetCellValue("itemB");
sheet2.CreateRow(2).CreateCell(0).SetCellValue("itemC");
```

然后定义一个名称，指向刚才创建的下拉项的区域：

```
HSSFName range = hssfworkbook.CreateName();
range.Reference = "ShtDictionary!$A1:$A3";
range.NameName = "dicRange";
```

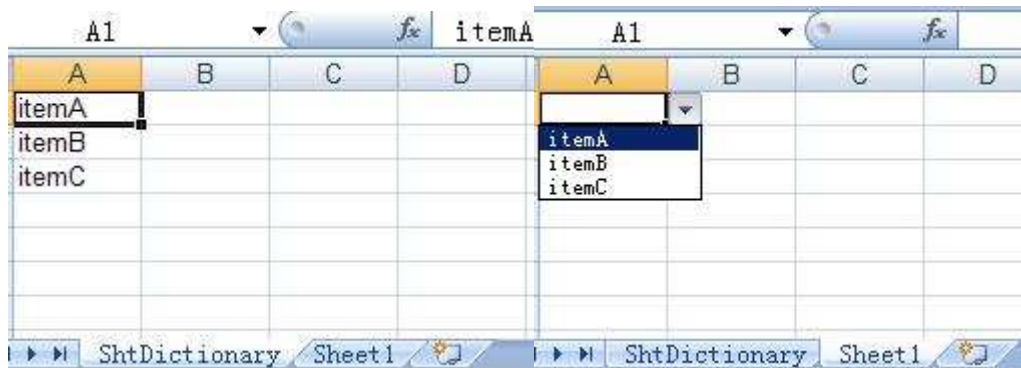
最后，设置数据约束时指向这个名称而不是字符数组：

```
HSSFSheet sheet1 = hssfworkbook.CreateSheet("Sheet1");
CellRangeAddressList regions = new CellRangeAddressList(0, 65535, 0, 0);

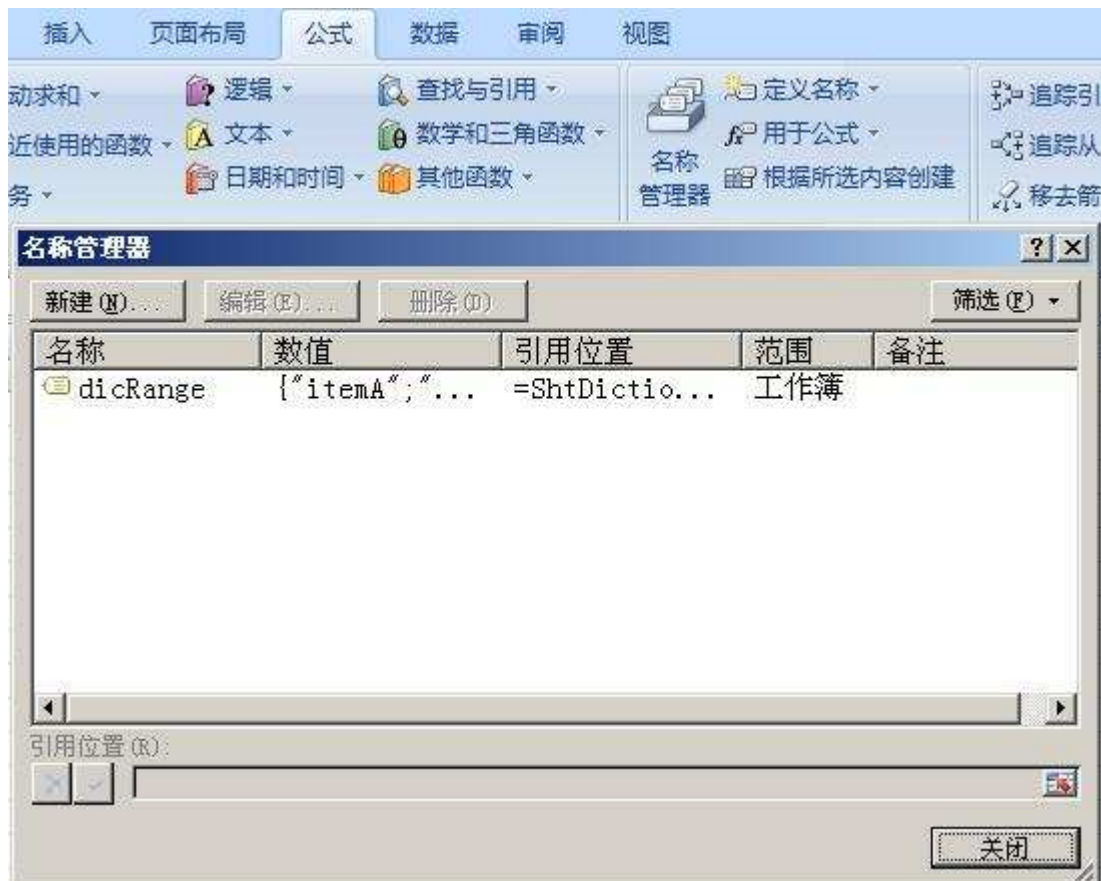
DVConstraint constraint = DVConstraint.CreateFormulaListConstraint("dicRange");
```

```
HSSFDataValidation dataValidate = new HSSFDataValidation(regions, constraint);  
sheet1.AddValidationData(dataValidate);
```

执行这段代码，生成的 Excel 效果如下：



在名称管理器中会发现有一个名为“dicRange”的名称，指向“ShtDictionary!\$A1:\$A3”的下拉项区域：



在数据有效性中会发现来源变成了“=dicRange”，指向上面定义的名称。而不是以前的“itemA, itemB, itemC”：





### 3. 项目实践

#### 3.1 基于.xls 模板生成 Excel 文件

尽管 NPOI 能够从头开始生成 Excel 文件，但在实际生产环境中有很多现成的表格，我们不可能没事就去从头开始生成一个 Excel，更多时候我们更愿意选择比较偷懒的方法——那就是用模板文件。NPOI 一大特色之一就是能够轻松读取 Office Excel 97-2003 的格式，即使里面有 NPOI 不支持的 VBA 宏、图表以及 Pivot 表之类的高级记录，NPOI 也能够保证不丢失数据（说实话，要完全能够识别所有的 Excel 内部记录几乎是不可能的，更何况如今又多出了 Office Excel 2007 binary file，即.xlsb）。

现在我们转入正题，出于演示目的，我做了一个简单的销售量表，里面应用了文字颜色、背景色、文本居中、公式、千分位分隔符、边框等效果，当然实际的生产环境里可能还有更加复杂的 Excel 模板。如下图

A1		fx
	A	B
1		<b>Sales Amount</b>
2	Jan	
3	Feb	
4	Mar	
5	Apr	
6	May	
7	Jun	
8	Jul	
9	Aug	
10	Sep	
11	Oct	
12	Nov	
13	Dec	
14		
15	Total	0
16		

我们的程序就是要填充 12 个月的销售量，Total 能够自动根据填充的值计算出总量。

（这里要提一下，以往如果我们用 HTML 方式输出 xls，我们必须在服务器端做 Total 计算，并且这个值在下载后永远都是静态的，没有公式，即使用户要修改里面的数据，总值也不会改变。这也是为什么 NPOI 一直提倡生成真正的 Excel 文件。）

代码其实很简单：

```
//read the template via FileStream, it is suggested to use FileAccess.Read to
prevent file lock.

//book1.xls is an Excel-2007-generated file, so some new unknown BIFF records are
added.

FileStream file = new FileStream(@"template/book1.xls",
FileMode.Open, FileAccess.Read);

HSSFWorkbook hssfworkbook = new HSSFWorkbook(file);
HSSFSheet sheet1 = hssfworkbook.GetSheet("Sheet1");
sheet1.GetRow(1).GetCell(1).SetCellValue(200200);
sheet1.GetRow(2).GetCell(1).SetCellValue(300);
sheet1.GetRow(3).GetCell(1).SetCellValue(500050);
sheet1.GetRow(4).GetCell(1).SetCellValue(8000);
sheet1.GetRow(5).GetCell(1).SetCellValue(110);
sheet1.GetRow(6).GetCell(1).SetCellValue(100);
```

```
sheet1.GetRow(7).GetCell(1).SetCellValue(200);
sheet1.GetRow(8).GetCell(1).SetCellValue(210);
sheet1.GetRow(9).GetCell(1).SetCellValue(2300);
sheet1.GetRow(10).GetCell(1).SetCellValue(240);
sheet1.GetRow(11).GetCell(1).SetCellValue(180123);
sheet1.GetRow(12).GetCell(1).SetCellValue(150);

//Force excel to recalculate all the formula while open

sheet1.ForceFormulaRecalculation = true;

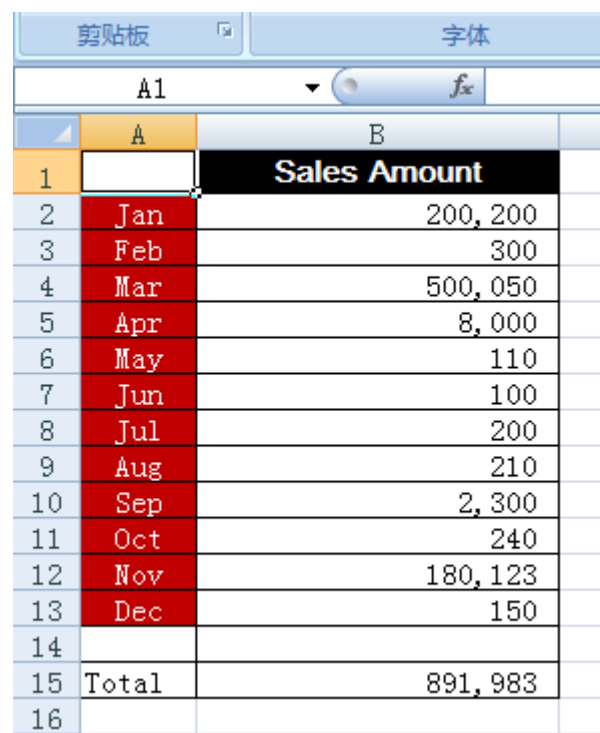
FileStream file = new FileStream(@"test.xls", FileMode.Create);
hssfworkbook.Write(file);
file.Close();
```

首先打开模板文件时要使用 `FileAccess.Read`，这样可以保证文件不被占用。

这里的 `ForceFormulaRecalculation` 是强制要求 Excel 在打开时重新计算的属性，在拥有公式的 xls 文件中十分有用，大家使用时可别忘了设。

是不是比你想象的简单？你甚至不用去了解它是在何时读取文件内容的，对于 NPOI 的使用者来说基本上和读取普通文件没有什么两样。

最终生成的效果如下所示：



	A	B
1		<b>Sales Amount</b>
2	Jan	200,200
3	Feb	300
4	Mar	500,050
5	Apr	8,000
6	May	110
7	Jun	100
8	Jul	200
9	Aug	210
10	Sep	2,300
11	Oct	240
12	Nov	180,123
13	Dec	150
14		
15	Total	891,983
16		

发觉没，就连千分位分隔符也都保留着，一切就像人工填写的一样。

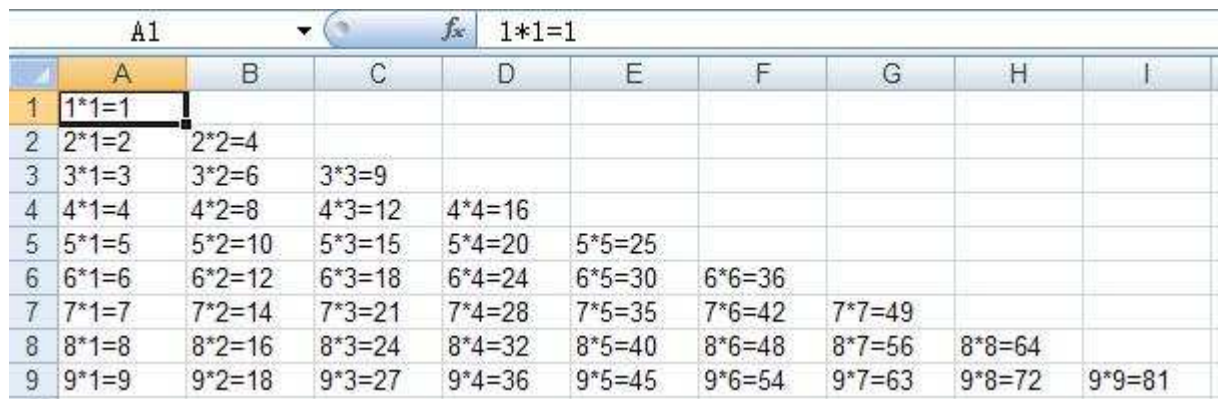
本范例完整代码请见 NPOI.Examples 中的 GenerateXlsFromXlsTemplate 项目。

### 3.2 生成九九乘法表

还记得小学时候学的九九乘法表吗？这节课我们一起学习利用 NPOI 通过 C# 代码生成一张 Excel 的九九乘法表。要生成九九乘法表，循环肯定是少不了的，如下：

```
HSSFSheet sheet1 = hssfworkbook.CreateSheet("Sheet1");
HSSFRow row;
HSSFCell cell;
for (int rowIndex = 0; rowIndex < 9; rowIndex++)
{
    row = sheet1.CreateRow(rowIndex);
    for (int colIndex = 0; colIndex <= rowIndex; colIndex++)
    {
        cell = row.CreateCell(colIndex);
        cell.SetCellValue(String.Format("{0}*{1}={2}", rowIndex + 1, colIndex + 1, (rowIndex + 1) * (colIndex + 1)));
    }
}
```

代码其实很简单，就是循环调用 cell.SetCellValue(str) 写入 9 行数据，每一行写的单元格数量随行数递增。执行完后生成的 Excel 样式如下：



	A	B	C	D	E	F	G	H	I
1	1*1=1								
2	2*1=2	2*2=4							
3	3*1=3	3*2=6	3*3=9						
4	4*1=4	4*2=8	4*3=12	4*4=16					
5	5*1=5	5*2=10	5*3=15	5*4=20	5*5=25				
6	6*1=6	6*2=12	6*3=18	6*4=24	6*5=30	6*6=36			
7	7*1=7	7*2=14	7*3=21	7*4=28	7*5=35	7*6=42	7*7=49		
8	8*1=8	8*2=16	8*3=24	8*4=32	8*5=40	8*6=48	8*7=56	8*8=64	
9	9*1=9	9*2=18	9*3=27	9*4=36	9*5=45	9*6=54	9*7=63	9*8=72	9*9=81

完整的代码如下：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using NPOI.HSSF.UserModel;
using System.IO;
```

```

using NPOI.HPSF;

namespace TimesTables
{
    public class Program
    {
        static HSSFWorkbook hssfworkbook;

        static void Main(string[] args)
        {
            InitializeWorkbook();

            HSSFSheet sheet1 = hssfworkbook.CreateSheet("Sheet1");
            HSSFRow row;
            HSSFCell cell;
            for (int rowIndex = 0; rowIndex < 9; rowIndex++)
            {
                row = sheet1.CreateRow(rowIndex);
                for (int colIndex = 0; colIndex <= rowIndex; colIndex++)
                {
                    cell = row.CreateCell(colIndex);
                    cell.SetCellValue(String.Format("{0}*{1}={2}", rowIndex +
1, colIndex + 1, (rowIndex + 1) * (colIndex + 1)));
                }
            }

            WriteToFile();
        }

        static void WriteToFile()
        {
            //Write the stream data of workbook to the root directory
            FileStream file = new FileStream(@"test.xls", FileMode.Create);
            hssfworkbook.Write(file);
            file.Close();
        }
    }
}

```

```

    }

    static void InitializeWorkbook()
    {
        hssfworkbook = new HSSFWorkbook();

        //create a entry of DocumentSummaryInformation
        DocumentSummaryInformation dsi = PropertySetFactory.CreateDocument
        SummaryInformation();
        dsi.Company = "NPOI Team";
        hssfworkbook.DocumentSummaryInformation = dsi;

        //create a entry of SummaryInformation
        SummaryInformation si = PropertySetFactory.CreateSummaryInformatio
        n();
        si.Subject = "NPOI SDK Example";
        hssfworkbook.SummaryInformation = si;
    }
}
}
}

```

### 3.3 生成一张工资单

这一节，我们将综合 NPOI 的常用功能（包括创建和填充单元格、合并单元格、设置单元格样式和利用公式），做一个工资单的实例。先看创建标题行的代码：

```

//写标题文本
HSSFSheet sheet1 = hssfworkbook.CreateSheet("Sheet1");
HSSFCell cellTitle = sheet1.CreateRow(0).CreateCell(0);
cellTitle.SetCellValue("XXX 公司 2009 年 10 月工资单");

//设置标题行样式
HSSFCellStyle style = hssfworkbook.CreateCellStyle();
style.Alignment = HSSFCellStyle.ALIGN_CENTER;
HSSFFont font = hssfworkbook.CreateFont();
font.FontHeight = 20 * 20;

```

```
style.SetFont(font);
```

```
cellTitle.CellStyle = style;
```

```
//合并标题行
```

```
sheet1.AddMergedRegion(new Region(0, 0, 1, 6));
```

其中用到了我们前面讲的设置单元格样式和合并单元格等内容。接下来我们循环创建公司每个员工的工资单:

```
DataTable dt=GetData();
```

```
HSSFRow row;
```

```
HSSFCell cell;
```

```
HSSFCellStyle celStyle=getCellStyle();
```

```
HSSFPatriarch patriarch = sheet1.CreateDrawingPatriarch();
```

```
HSSFClientAnchor anchor;
```

```
HSSFSimpleShape line;
```

```
int rowIndex;
```

```
for (int i = 0; i < dt.Rows.Count; i++)
```

```
{
```

```
    //表头数据
```

```
    rowIndex = 3 * (i + 1);
```

```
    row = sheet1.CreateRow(rowIndex);
```

```
    cell = row.CreateCell(0);
```

```
    cell.SetCellValue("姓名");
```

```
    cell.CellStyle = celStyle;
```

```
    cell = row.CreateCell(1);
```

```
    cell.SetCellValue("基本工资");
```

```
    cell.CellStyle = celStyle;
```

```
    cell = row.CreateCell(2);
```

```
    cell.SetCellValue("住房公积金");
```

```
    cell.CellStyle = celStyle;
```

```
cell = row.CreateCell(3);
cell.SetCellValue("绩效奖金");
cell.CellStyle = celStyle;

cell = row.CreateCell(4);
cell.SetCellValue("社保扣款");
cell.CellStyle = celStyle;

cell = row.CreateCell(5);
cell.SetCellValue("代扣个税");
cell.CellStyle = celStyle;

cell = row.CreateCell(6);
cell.SetCellValue("实发工资");
cell.CellStyle = celStyle;

DataRow dr = dt.Rows[i];
//设置值和计算公式
row = sheet1.CreateRow(rowIndex + 1);
cell = row.CreateCell(0);
cell.SetCellValue(dr["FName"].ToString());
cell.CellStyle = celStyle;

cell = row.CreateCell(1);
cell.SetCellValue((double)dr["FBasicSalary"]);
cell.CellStyle = celStyle;

cell = row.CreateCell(2);
cell.SetCellValue((double)dr["FAccumulationFund"]);
cell.CellStyle = celStyle;

cell = row.CreateCell(3);
cell.SetCellValue((double)dr["FBonus"]);
cell.CellStyle = celStyle;
```



```

cell = row.CreateCell(4);
cell.SetCellFormula(String.Format("$B{0}*0.08", rowIndex+2));
cell.CellStyle = celStyle;

cell = row.CreateCell(5);
cell.SetCellFormula(String.Format("SUM($B{0}:$D{0})*0.1", rowIndex+2));
cell.CellStyle = celStyle;

cell = row.CreateCell(6);
cell.SetCellFormula(String.Format("SUM($B{0}:$D{0})-SUM($E{0}:$F{0})", rowI
ndex+2));
cell.CellStyle = celStyle;

//绘制分隔线
sheet1.AddMergedRegion(new Region(rowIndex+2, 0, rowIndex+2, 6));
anchor = new HSSFClientAnchor(0, 125, 1023, 125, 0, rowIndex + 2, 6, rowIn
dex + 2);
line = patriarch.CreateSimpleShape(anchor);
line.ShapeType = HSSFShape.OBJECT_TYPE_LINE;
line.LineStyle = HSSFShape.LINESTYLE_DASHGEL;
}

```

其中为了文件打印为单元格增加了黑色边框的样式（如果不设置边框样式，打印出来后是没有边框的）。另外，注意循环过程中 excel 中的行号随数据源中的行号变化处理。完整代码如下：

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using NPOI.HSSF.UserModel;
using System.IO;
using NPOI.HPSF;
using NPOI.HSSF.Util;

```

```

using System. Data;

namespace Payroll
{
    public class Program
    {
        static HSSFWorkbook hssfworkbook;

        static void Main(string[] args)
        {
            InitializeWorkbook();

            //写标题文本
            HSSFSheet sheet1 = hssfworkbook. CreateSheet("Sheet1");
            HSSFCell cellTitle = sheet1. CreateRow(0). CreateCell(0);
            cellTitle. SetCellValue("XXX 公司 2009 年 10 月工资单");

            //设置标题行样式
            HSSFCellStyle style = hssfworkbook. CreateCellStyle();
            style. Alignment = HSSFCellStyle. ALIGN_CENTER;
            HSSFFont font = hssfworkbook. CreateFont();
            font. FontHeight = 20 * 20;
            style. SetFont(font);

            cellTitle. CellStyle = style;

            //合并标题行
            sheet1. AddMergedRegion(new Region(0, 0, 1, 6));

            DataTable dt=GetData();
            HSSFRow row;
            HSSFCell cell;
            HSSFCellStyle celStyle=getCellStyle();

            HSSFPatriarch patriarch = sheet1. CreateDrawingPatriarch();

```

```
HSSFClientAnchor anchor;
HSSFSimpleShape line;
int rowIndex;
for (int i = 0; i < dt.Rows.Count; i++)
{
    //表头数据
    rowIndex = 3 * (i + 1);
    row = sheet1.CreateRow(rowIndex);

    cell = row.CreateCell(0);
    cell.SetCellValue("姓名");
    cell.CellStyle = celStyle;

    cell = row.CreateCell(1);
    cell.SetCellValue("基本工资");
    cell.CellStyle = celStyle;

    cell = row.CreateCell(2);
    cell.SetCellValue("住房公积金");
    cell.CellStyle = celStyle;

    cell = row.CreateCell(3);
    cell.SetCellValue("绩效奖金");
    cell.CellStyle = celStyle;

    cell = row.CreateCell(4);
    cell.SetCellValue("社保扣款");
    cell.CellStyle = celStyle;

    cell = row.CreateCell(5);
    cell.SetCellValue("代扣个税");
    cell.CellStyle = celStyle;

    cell = row.CreateCell(6);
    cell.SetCellValue("实发工资");
```

```

cell.CellStyle = celStyle;

DataRow dr = dt.Rows[i];
//设置值和计算公式
row = sheet1.CreateRow(rowIndex + 1);
cell = row.CreateCell(0);
cell.SetCellValue(dr["FName"].ToString());
cell.CellStyle = celStyle;

cell = row.CreateCell(1);
cell.SetCellValue((double)dr["FBasicSalary"]);
cell.CellStyle = celStyle;

cell = row.CreateCell(2);
cell.SetCellValue((double)dr["FAccumulationFund"]);
cell.CellStyle = celStyle;

cell = row.CreateCell(3);
cell.SetCellValue((double)dr["FBonus"]);
cell.CellStyle = celStyle;

cell = row.CreateCell(4);
cell.SetCellFormula(String.Format("$B{0}*0.08", rowIndex+2));
cell.CellStyle = celStyle;

cell = row.CreateCell(5);
cell.SetCellFormula(String.Format("SUM($B{0} : $D{0})*0.1", rowIn
dex+2));
cell.CellStyle = celStyle;

cell = row.CreateCell(6);
cell.SetCellFormula(String.Format("SUM($B{0} : $D{0})-SUM($E{0} :
$F{0})", rowIndex+2));
cell.CellStyle = celStyle;

```

```

        //绘制分隔线
        sheet1.AddMergedRegion(new Region(rowIndex+2, 0, rowIndex+2, 6
));
        anchor = new HSSFClientAnchor(0, 125, 1023, 125, 0, rowIndex +
2, 6, rowIndex + 2);
        line = patriarch.CreateSimpleShape(anchor);
        line.ShapeType = HSSFShape.OBJECT_TYPE_LINE;
        line.LineStyle = HSSFShape.LINESTYLE_DASHGEL;

    }

    WriteToFile();
}

static DataTable GetData()
{
    DataTable dt = new DataTable();
    dt.Columns.Add("FName", typeof(System.String));
    dt.Columns.Add("FBasicSalary", typeof(System.Double));
    dt.Columns.Add("FAccumulationFund", typeof(System.Double));
    dt.Columns.Add("FBonus", typeof(System.Double));

    dt.Rows.Add("令狐冲", 6000, 1000, 2000);
    dt.Rows.Add("任盈盈", 7000, 1000, 2500);
    dt.Rows.Add("林平之", 5000, 1000, 1500);
    dt.Rows.Add("岳灵珊", 4000, 1000, 900);
    dt.Rows.Add("任我行", 4000, 1000, 800);
    dt.Rows.Add("风清扬", 9000, 5000, 3000);

    return dt;
}

```

```

static HSSFCellStyle getCellStyle()
{
    HSSFCellStyle cellStyle = hssfworkbook.CreateCellStyle();
    cellStyle.BorderBottom = HSSFCellStyle.BORDER_THIN;
    cellStyle.BorderLeft = HSSFCellStyle.BORDER_THIN;
    cellStyle.BorderRight = HSSFCellStyle.BORDER_THIN;
    cellStyle.BorderTop = HSSFCellStyle.BORDER_THIN;
    return cellStyle;
}

static void WriteToFile()
{
    //Write the stream data of workbook to the root directory
    FileStream file = new FileStream(@"test.xls", FileMode.Create);
    hssfworkbook.Write(file);
    file.Close();
}

static void InitializeWorkbook()
{
    hssfworkbook = new HSSFWorkbook();

    //create a entry of DocumentSummaryInformation
    DocumentSummaryInformation dsi = PropertySetFactory.CreateDocument
SummaryInformation();
    dsi.Company = "NPOI Team";
    hssfworkbook.DocumentSummaryInformation = dsi;

    //create a entry of SummaryInformation
    SummaryInformation si = PropertySetFactory.CreateSummaryInformatio
n();
    si.Subject = "NPOI SDK Example";
    hssfworkbook.SummaryInformation = si;
}
}

```

}

生成的 Excel 文件样式如下：

G5							=SUM(\$B5:\$D5)-SUM(\$E5:\$F5)
	A	B	C	D	E	F	G
1	XXX公司2009年10月工资单						
2							
3							
4	姓名	基本工资	住房公积金	绩效奖金	社保扣款	代扣个税	实发工资
5	令狐冲	6000	1000	2000	480	900	7620
6							
7	姓名	基本工资	住房公积金	绩效奖金	社保扣款	代扣个税	实发工资
8	任盈盈	7000	1000	2500	560	1050	8890
9							
10	姓名	基本工资	住房公积金	绩效奖金	社保扣款	代扣个税	实发工资
11	林平之	5000	1000	1500	400	750	6350
12							
13	姓名	基本工资	住房公积金	绩效奖金	社保扣款	代扣个税	实发工资
14	岳灵珊	4000	1000	900	320	590	4990
15							
16	姓名	基本工资	住房公积金	绩效奖金	社保扣款	代扣个税	实发工资
17	任我行	4000	1000	800	320	580	4900
18							
19	姓名	基本工资	住房公积金	绩效奖金	社保扣款	代扣个税	实发工资
20	风清扬	9000	5000	3000	720	1700	14580
21							

### 3.4 从 Excel 中抽取文本

我们知道，搜索引擎最擅长处理的就是文本，而 Excel 中的内容并不是以文本方式存储的。那么如果想要搜索引擎爬虫能够抓取到 Excel 中的内容是比较困难的，除非搜索引擎爬虫对 Excel 格式进行专门的处理。那么有没有办法解决此问题呢？有，通过 NPOI 将 Excel 内容文本化！

如下，有这样一张 Excel，如果想让它被搜索引擎收录，常用的方式是以 HTML 形式展现，但将一个个这样的 Excel 手工做成 HTML 页面显然比较麻烦。接下来，我们将提供一种方案，自动将 Excel 中的内容以 HTML 形式展现。

CPU参考报价				
品牌	型号	规格(主频/缓存)	报价	涨跌
Intel	赛扬D 347 (散)	3.06G/512K/65nm	206	
	奔腾4 2.8G (散)	2.8G/256K/65nm	455	
	奔腾4 631 (散)	3.0G/2M/65nm	360	
	赛扬 CIV430 (散)	1.8G/512K/65nm	219	3
	奔腾双核 E1200 (散)	1.6G/512K/65nm	260	
	酷睿2 E8400 (盒)	3.0G/6M/45nm	1260	
	酷睿2 Q8200 (散)	2330MHz/4M/45nm	915	
	酷睿2 Q8200 (盒)	2330MHz/4M/45nm	1020	
	AMD	闪龙 2600 (散)	1.8G/256KB/130nm/62W	80
速龙64 X2 5600+ (盒)		2.9GHz/2MB/65nm/65W	510	
速龙64 X2 7850+ (黑盒)		2.7GHz/1MB/65nm/95W	442	-18
羿龙X3 8650 (盒)		2.3GHz/1.5MB/65nm	520	-5
Phenom 9750 (盒)		2400MHz/2MB/65nm	940	
Phenom 9650 (盒)		2000MHz/2MB/65nm	785	-20

其实基本思想也很简单，就是通过 NPOI 读取每个 Cell 中的内容，然后以 HTML 的形式输出。但要保证输出的 HTML 页面布局与 Excel 中的一致，还有点小技巧。下面是构造 Table 的代码：

```
private HSSFWorkbook sht;
protected String excelContent;

protected void Page_Load(object sender, EventArgs e)
{
    HSSFWorkbook wb = new HSSFWorkbook(new FileStream(Server.MapPath("~/App_Data/quotation.xls"), FileMode.Open));
    sht = wb.GetSheet("Sheet1");

    //取行 Excel 的最大行数
    int rowCount = sht.PhysicalNumberOfRows;
    //为保证 Table 布局与 Excel 一样，这里应该取所有行中的最大列数（需要遍历整个 Sheet）。
    //为少一交全 Excel 遍历，提高性能，我们可以人为把第 0 行的列数调整至所有行中的最大列数。
    int colsCount = sht.GetRow(0).PhysicalNumberOfCells;

    int colSpan;
    int rowSpan;
```



```

bool isByRowMerged;

StringBuilder table = new StringBuilder(rowsCount * 32);

table.Append("<table border=' 1px' >");
for (int rowIndex = 0; rowIndex < rowsCount; rowIndex++)
{
    table.Append("<tr>");
    for (int colIndex = 0; colIndex < colsCount; colIndex++)
    {
        GetTdMergedInfo(rowIndex, colIndex, out colSpan, out rowSpan, out
isByRowMerged);
        //如果已经被行合并包含进去了就不输出 TD 了。
        //注意被合并的行或列不输出的处理方式不一样，见下面一处的注释说明了
列合并后不输出 TD 的处理方式。
        if (isByRowMerged)
        {
            continue;
        }

        table.Append("<td");
        if (colSpan > 1)
            table.Append(string.Format(" colspan={0}", colSpan));
        if (rowSpan > 1)
            table.Append(string.Format(" rowspan={0}", rowSpan));
        table.Append(">");

        table.Append(sht.GetRow(rowIndex).GetCell(colIndex));

        //列被合并之后此行将少输出 colSpan-1 个 TD。
        if (colSpan > 1)
            colIndex += colSpan - 1;

        table.Append("</td>");
    }
}

```

```

    }
    table.Append("</tr>");
}
table.Append("</table>");

this.excelContent = table.ToString();
}

```

其中用到的 GetTdMergedInfo 方法代码如下：

```

/// <summary>
/// 获取 Table 某个 TD 合并的列数和行数等信息。与 Excel 中对应 Cell 的合并行数和列
/// 数一致。
/// </summary>
/// <param name="rowIndex">行号</param>
/// <param name="colIndex">列号</param>
/// <param name="colspan">TD 中需要合并的行数</param>
/// <param name="rowspan">TD 中需要合并的列数</param>
/// <param name="rowspan">此单元格是否被某个行合并包含在内。如果被包含在内，将
/// 不输出 TD。</param>
/// <returns></returns>
private void GetTdMergedInfo(int rowIndex, int colIndex, out int colspan, out
int rowspan, out bool isByRowMerged)
{
    colspan = 1;
    rowspan = 1;
    isByRowMerged = false;
    int regionsCuont = sht.NumMergedRegions;
    Region region;
    for (int i = 0; i < regionsCuont; i++)
    {
        region = sht.GetMergedRegionAt(i);
        if (region.RowFrom == rowIndex && region.ColumnFrom == colIndex)
        {
            colspan = region.ColumnTo - region.ColumnFrom + 1;
            rowspan = region.RowTo - region.RowFrom + 1;

```

```

        return;
    }
    else if (rowIndex > region.RowFrom && rowIndex <= region.RowTo && colIndex >= region.ColumnFrom && colIndex <= region.ColumnTo)
    {
        isByRowMerged = true;
    }
}
}
}

```

最后在.aspx 页面中输出构建好的 Table:

```
<%=excelContent %>
```

执行效果如下:

CPU参考报价				
品牌	型号	规格(主频/缓存)	报价	涨跌
Intel	赛扬D 347 (散)	3.06G/512K/65nm	206	
	奔腾4 2.8G (散)	2.8G/256K/65nm	455	
	奔腾4 631 (散)	3.0G/2M/65nm	360	
	赛扬 CIV430 (散)	1.8G/512K/65nm	219	3
	奔腾双核 E1200 (散)	1.6G/512K/65nm	260	
	酷睿2 E8400 (盒)	3.0G/6M/45nm	1260	
	酷睿2 Q8200 (散)	2330MHz/4M/45nm	915	
	酷睿2 Q8200 (盒)	2330MHz/4M/45nm	1020	
AMD	闪龙 2600 (散)	1.8G/256KB/130nm/62W	80	
	速龙64 X2 5600+ (盒)	2.9GHz/2MB/65nm/65W	510	
	速龙64 X2 7850+ (黑盒)	2.7GHz/1MB/65nm/95W	442	-18
	羿龙X3 8650 (盒)	2.3GHz/1.5MB/65nm	520	-5
	Phenom 9750(盒)	2400MHz/2MB/65nm	940	
	Phenom 9650(盒)	2000MHz/2MB/65nm	785	-20

我们发现，与 Excel 中的布局完全一样（这里没有处理单元格的样式，只处理了内容，有兴趣的读者也可以将 Excel 中单元格的样式也应用在 HTML 中）。这里为保证布局一致，主要是将 Excel 中的 Region 信息解析成 Table 的 colSpan 和 rowSpan 属性，如果对这两个属性不太了解，可以结合以下代码和示例加以了解：

```
<table width="300px" border="1px">
<tr>
  <td colspan="2" rowspan="2">0,0</td>
  <td>0,3</td>
</tr>
<tr>
  <td>1,3</td>
</tr>
<tr>
  <td rowspan="2">2,0</td>
  <td colspan="2">2,1</td>
</tr>
<tr>
  <td>3,1</td>
  <td>3,2</td>
</tr>
</table>
```

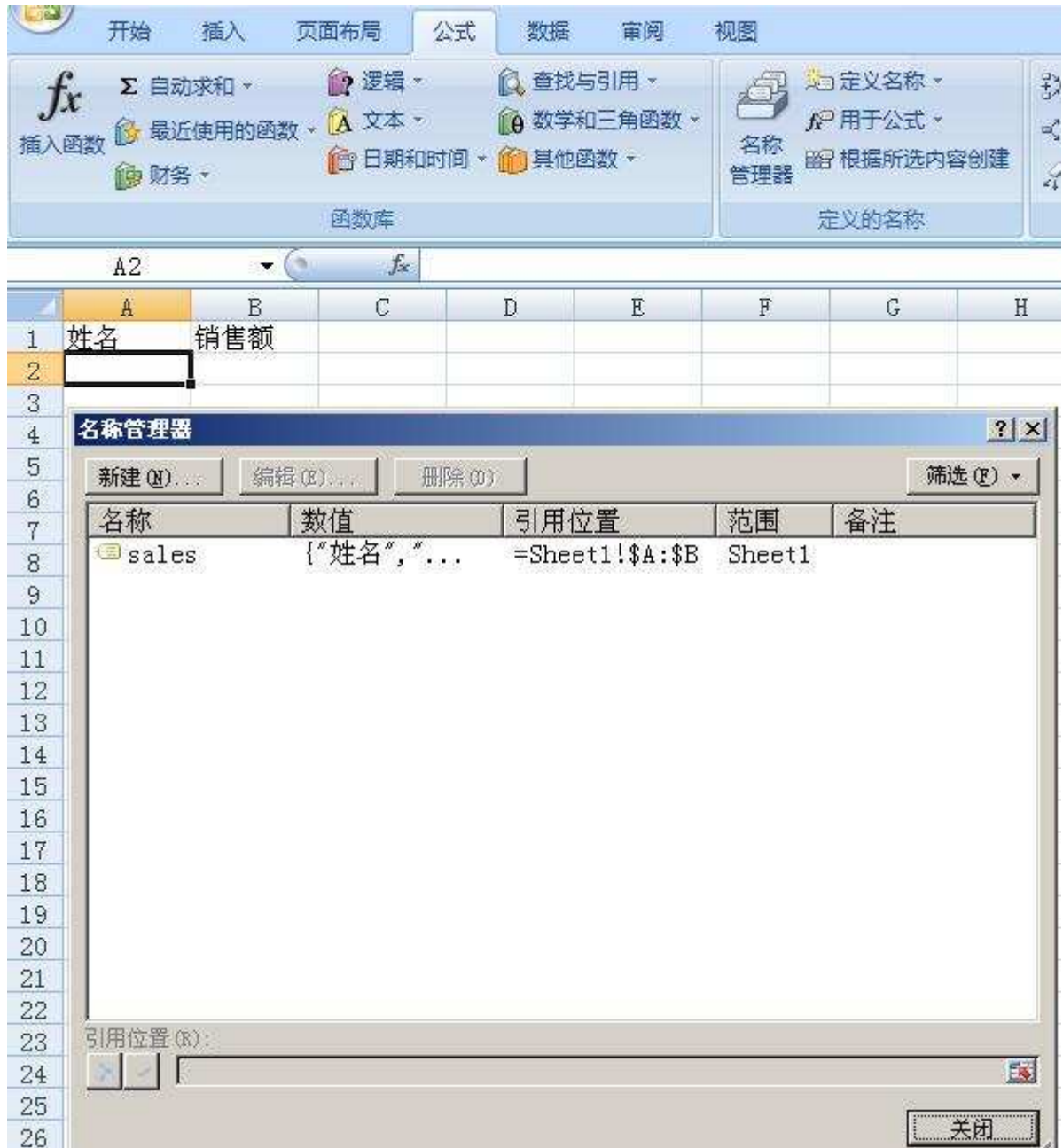
以上 HTML 代码对应的 Table 展现为：

0,0		0,3
		1,3
2,0	2,1	
	3,1	3,2

### 3.5 巧妙使用 Excel Chart

在 NPOI 中，本身并不支持 Chart 等高级对象的创建，但通过 1 模板的方式可以巧妙地利用 Excel 强大的透视和图表功能，请看以下例子。

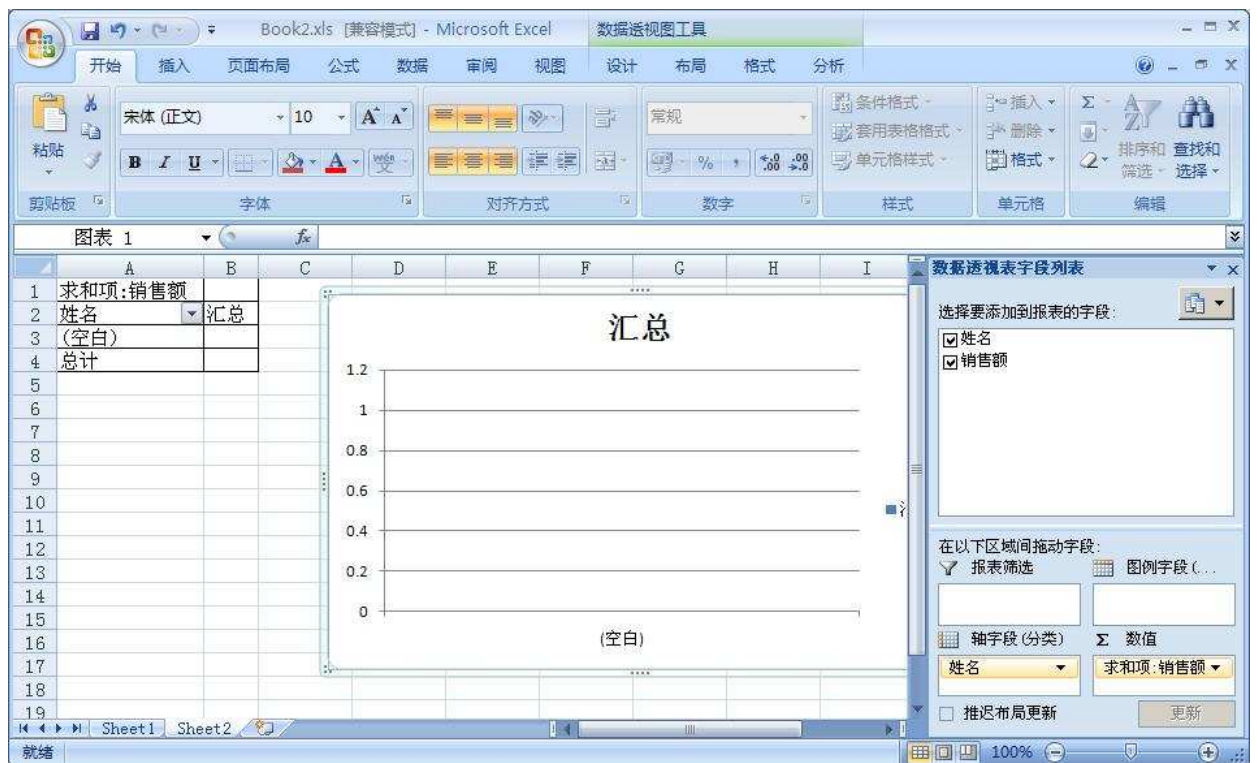
首先建立模板文件，定义两列以及指向此区域的名称“sales”：



创建数据表，数据来源填入刚才定义的区域：



最后生成的数据透视表所在 Sheet 的样式如下：



至此，模板已经建好，另存为“D:\MyProject\NPOIDemo\Chart\Book2.xls”。我们发现，模板就相当于一个“空架子”，仅仅有操作方式并没有任何数据。下一步，我们往这个“空架子”中填入数据。我们通过如下代码往这个“空架子”中写入数据：

```
static void Main(string[] args)
{
    HSSFWorkbook wb = new HSSFWorkbook(new FileStream(@"D:\MyProject\NPOIDemo\
Chart\Book2.xls", FileMode.Open));
```

```

HSSFSheet sheet1 = wb.GetSheet("Sheet1");

HSSFRow row = sheet1.CreateRow(1);
row.CreateCell(0).SetCellValue("令狐冲");
row.CreateCell(1).SetCellValue(50000);

row = sheet1.CreateRow(2);
row.CreateCell(0).SetCellValue("任盈盈");
row.CreateCell(1).SetCellValue(30000);

row = sheet1.CreateRow(3);
row.CreateCell(0).SetCellValue("风清扬");
row.CreateCell(1).SetCellValue(80000);

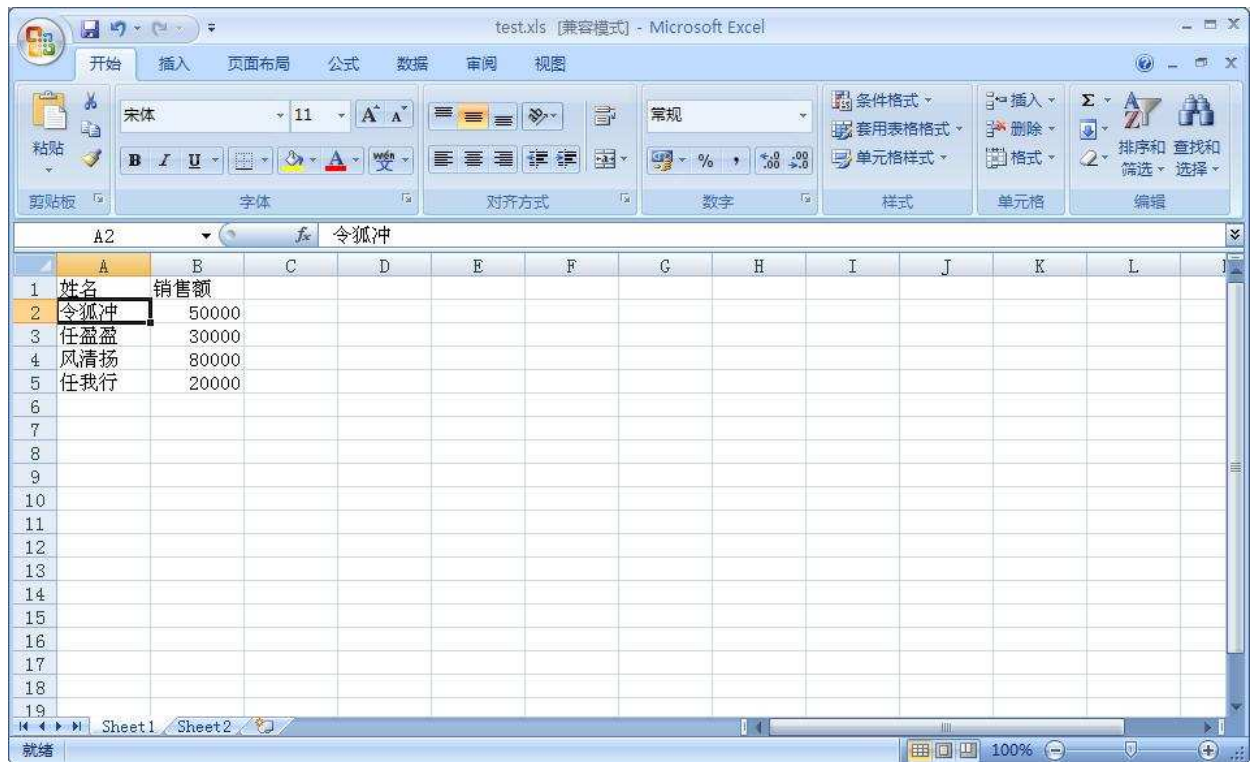
row = sheet1.CreateRow(4);
row.CreateCell(0).SetCellValue("任我行");
row.CreateCell(1).SetCellValue(20000);

//Write the stream data of workbook to the root directory
FileStream file = new FileStream(@"test.xls", FileMode.Create);
wb.Write(file);
file.Close();
}

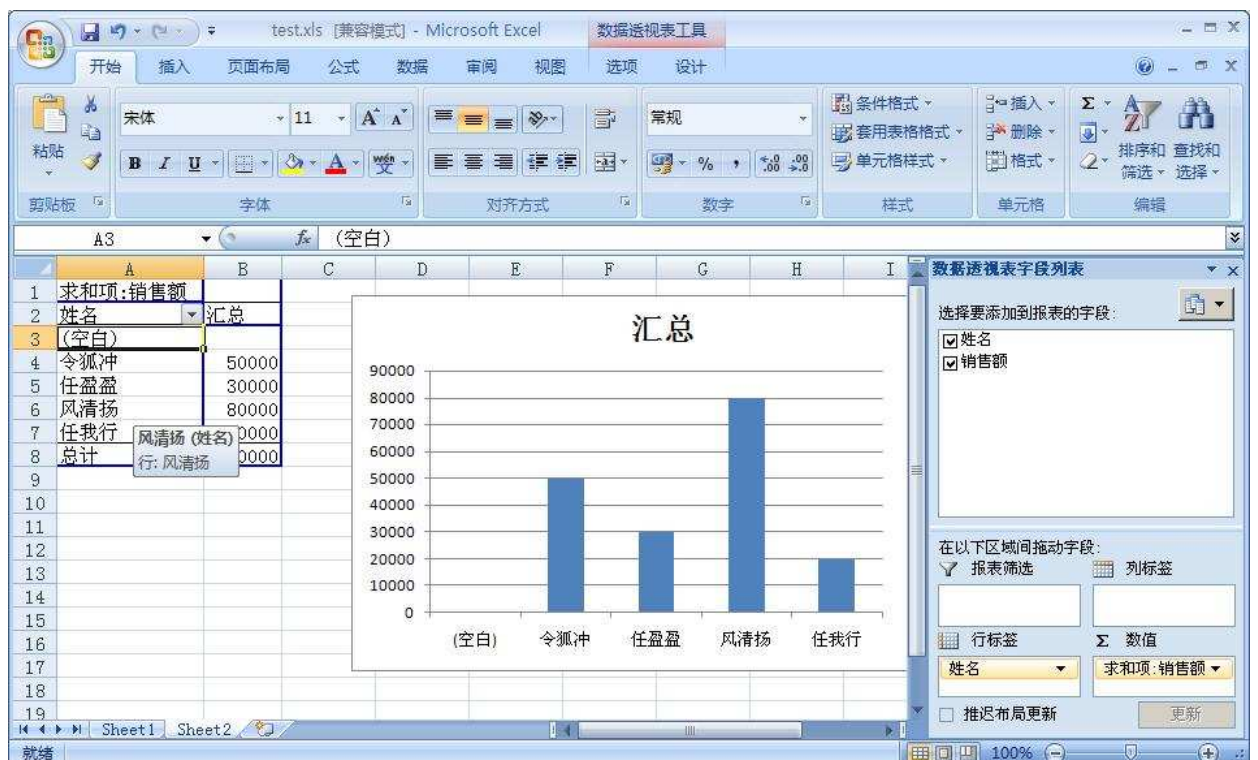
```

打开生成的 test.xls 文件，发现数据已经被填进去了：





再看数据透视表，也有数据了：



总结：

Excel 有着强大的报表透视和图表功能，而且简单易用，利用 NP01，可以对其进行充分利用。在做图形报表、透视报表时将非常有用！

### 3.6 .NET 导入 Excel 文件的另一种选择

NPOI 之所以强大，并不是因为它支持导出 Excel，而是因为它支持导入 Excel，并能“理解”OLE2 文档结构，这也是其他一些 Excel 读写库比较弱的方面。通常，读入并理解结构远比导出来得复杂，因为导入你必须假设一切情况都是可能的，而生成你只要保证满足你自己需求就可以了，如果把导入需求和生成需求比做两个集合，那么生成需求通常都是导入需求的子集，这一规律不仅体现在 Excel 读写库中，也体现在 pdf 读写库中，目前市面上大部分的 pdf 库仅支持生成，不支持导入。

如果你不相信 NPOI 能够很好的理解 OLE2 文档格式，那就去下载 [POIFS Brower](#)。具体可以参考这篇文章的介绍：[Office 文件格式解惑](#)。当然单单理解 OLE2 是不够的，因为 Excel 文件格式是 BIFF，但 BIFF 是以 OLE2 为基础的，做个很形象的比喻就是：OLE2 相当于磁盘的 FAT 格式，BIFF 相当于文件和文件夹。NPOI 负责理解 BIFF 格式的代码基本都在 HSSF 命名空间里面。

好了，刚才废话了一会儿，主要是给大家打打基础，现在进入正题。

本文将以 DataTable 为容器读入某 xls 的第一个工作表的数据（最近群里面很多人问这个问题）。

在开始之前，我们先来补些基础知识。每一个 xls 都对应一个唯一的 HSSFWorkbook，每一个 HSSFWorkbook 会有若干个 HSSFSheet，而每一个 HSSFSheet 包含若干 HSSFRow（Excel 2003 中不得超过 65535 行），每一个 HSSFRow 又包含若干个 HSSFCell（Excel 2003 中不得超过 256 列）。

为了遍历所有的单元格，我们就得获得某一个 HSSFSheet 的所有 HSSFRow，通常可以用 HSSFSheet.GetRowEnumerator()。如果要获得某一特定行，可以直接用 HSSFSheet.GetRow(rowIndex)。另外要遍历我们就必须知道边界，有一些属性我们是可以用到的，比如 HSSFSheet.FirstRowNum（工作表中第一个有数据行的行号）、HSSFSheet.LastRowNum（工作表中最后一个有数据行的行号）、HSSFRow.FirstCellNum（一行中第一个有数据列的列号）、HSSFRow.LastCellNum（一行中最后一个有数据列的列号）。

基础知识基本上补得差不多了，现在开工！

首先我们要准备一个用于打开文件流的函数 InitializeWorkbook，由于文件读完后就没用了，所以这里直接用 using（养成好习惯，呵呵）。

```
HSSFWorkbook hssfworkbook;  
  
void InitializeWorkbook(string path)  
{  
    //read the template via FileStream, it is suggested to use FileAccess.Read to  
    prevent file lock.
```

```
//book1.xls is an Excel-2007-generated file, so some new unknown BIFF records are added.
```

```
using (FileStream file = new FileStream(path, FileMode.Open, FileAccess.Read))
{
    hssfworkbook = new HSSFWorkbook(file);
}
}
```

接下来我们要开始写最重要的函数 ConvertToDataTable，即把 HSSF 的数据放到一个 DataTable 中。

```
HSSFSheet sheet = hssfworkbook.GetSheetAt(0);
System.Collections.IEnumerator rows = sheet.GetRowEnumerator();

while (rows.MoveNext())
{
    HSSFRow row = (HSSFRow)rows.Current;
    //TODO::Create DataTable row

    for (int i = 0; i < row.LastCellNum; i++)
    {
        HSSFCell cell = row.GetCell(i);
        //TODO::set cell value to the cell of DataTables
    }
}
```

上面的结构大家都应该能看懂吧，无非就是先遍历行，再遍历行中的每一列。这里引出了一个难点，由于 Excel 的单元格有好几种类型，类型不同显示的东西就不同，具体的类型有 布尔型、数值型、文本型、公式型、空白、错误。

```
public enum HSSFCellType
{
    Unknown = -1,
    NUMERIC = 0,
    STRING = 1,
    FORMULA = 2,
    BLANK = 3,
    BOOLEAN = 4,
```

```
    ERROR = 5,  
}
```

这里的 `HSSFCellType` 描述了所有的类型，但细心的朋友可能已经发现了，这里没有日期型，这是为什么呢？这是因为 Excel 底层并没有一定日期型，而是通过数值型来替代，至于如何区分日期和数字，都是由文本显示的样式决定的，在 NPOI 中则是由 `HSSFDataFormat` 来处理。为了能够方便的获得所需要的类型所对应的文本，我们可以使用 `HSSFCell.ToString()` 来处理。

于是刚才的代码则变成了这样：

```
HSSFSheet sheet = hssfworkbook.GetSheetAt(0);  
System.Collections.IEnumerator rows = sheet.GetRowEnumerator();  
  
DataTable dt = new DataTable();  
for (int j = 0; j < 5; j++)  
{  
    dt.Columns.Add(Convert.ToChar(((int)'A')+j).ToString());  
}  
  
while (rows.MoveNext())  
{  
    HSSFRow row = (HSSFRow)rows.Current;  
    DataRow dr = dt.NewRow();  
  
    for (int i = 0; i < row.LastCellNum; i++)  
    {  
        HSSFCell cell = row.GetCell(i);  
  
        if (cell == null)  
        {  
            dr[i] = null;  
        }  
        else  
        {  
            dr[i] = cell.ToString();  
        }  
    }  
}
```

```
dt.Rows.Add(dr);  
}
```

是不是很简单，呵呵！

当然，如果你要对某个特定的单元格类型做特殊处理，可以通过判 `HSSFCell.CellType` 来解决，比如下面的代码：

```
switch(cell.CellType)  
{  
    case HSSFCellType.BLANK:  
        dr[i] = "[null]";  
        break;  
    case HSSFCellType.BOOLEAN:  
        dr[i] = cell.BooleanCellValue;  
        break;  
    case HSSFCellType.NUMERIC:  
        dr[i] = cell.ToString(); //This is a trick to get the correct  
value of the cell. NumericCellValue will return a numeric value no matter the cell  
value is a date or a number.  
        break;  
    case HSSFCellType.STRING:  
        dr[i] = cell.StringCellValue;  
        break;  
    case HSSFCellType.ERROR:  
        dr[i] = cell.ErrorCellValue;  
        break;  
    case HSSFCellType.FORMULA:  
    default:  
        dr[i] = "=" + cell.CellFormula;  
        break;  
}
```

这里只是举个简单的例子。

完整代码下载：<http://files.cnblogs.com/tonyqus/ImportXlsToDataTable.zip>

注意，此代码中不包括 NPOI 的 assembly，否则文件会很大，所以建议去 [npoi.codeplex.com](http://npoi.codeplex.com) 下载。

在 Server 端存取 Excel 檔案的利器：NPOI Library

## 在 Server 端控制 Excel 的難處

在今日 Microsoft Excel 被廣為業界接受之際，Excel 已幾乎是每個人必會的工具，不論是在校園或是職場，接觸到 Excel 的機率很高，而且 Excel 靠著簡單易用以及高度容錯的能力，讓使用者可以近乎無痛的操控 Excel，它內建的強大試算以及資料整理的功能，也讓很多使用者樂於使用它，這一點由 Excel 的高市佔率得以印證，用 Excel 來整理與包裝資料已經是司空見慣的事，因此很多的使用者會利用它來處理日常的業務資料或是基本檔等等，不過這可就苦了 IT 人員了。

在論壇上經常會看到一種需求，因為使用者不論如何都要用 Excel 檔來放資料，所以總是要求 IT 人員在系統中直接產出 Excel 檔案直接下載給使用者，目前由官方公布，由伺服器端存取 Excel 表格兩種方法：

### 1. 使用 Excel 物件模型來存取

這是官方公布最正統的方法，直接將 Excel 物件模型加入專案參考，並使用 COM 的方式來呼叫內含在 Excel 物件庫中的物件，像是 Workbook、Worksheet、Range、Formula、Row、Cell 等等，它的好處是可以精確的控制 Excel 檔案中的各種屬性（儲存格格式、樣式、資料、公式以及條件等等），輸出的檔案也絕對是最正確的 Excel 資料檔，不過它卻有下列缺點：

- 物件模型複雜不易學習。  
在 Excel 物件庫中，擁有數百種物件以及數以百計的列舉常數值，每個物件之間又和數個不同的物件有交互關係，在程式的操控上相對不容易。
- 無法使用資料流方式控制 Excel 檔案。  
Excel 物件模型是基於 EXCEL.exe（Excel 的執行檔）來存取檔案的內容，物件顯露出來的讀取方法只有實體檔案的方式，而無法使用資料流（如 MemoryStream），這會迫使開發人員必須要另外處理產生的實體檔案，對開發人員來說並不方便。同時實體檔案的作業系統 I/O 功能，也會涉及到權限控制的問題。
- 以單機為基礎的執行引擎。  
Excel.exe 本身並不是針對網路環境來設計的，因此它基本上並不支援多人操作與共享的應用程式（簡言之，就是 Web 應用程式），因此經常會在論壇上看到諸如『為什麼 Excel 釋放不掉』的問題，其根本原因就是 Excel 本身的行程特性，在多人產生執行個體以及多行程鎖定的情況下，Excel 行程會被佔住無法釋放，無法放掉的行程愈多，對伺服器的記憶體就愈傷，甚至導致伺服器不穩定。

Excel 物件模型可以在 MSDN Excel Developer Center 中找到：

<http://msdn.microsoft.com/en-us/office/aa905411.aspx>

#### NOTE

微軟官方並不建議在伺服器端使用直接存取 Excel 物件模型的方式來控制 Excel 檔案，除了上述的資源無法釋放的問題外，還有像是權限的問題，以及安全性問題等等，詳細的資料請參考：

<http://support.microsoft.com/default.aspx/kb/257757>

## 2. 使用 OLE DB Provider for Jet 來存取

這是控制 Excel 檔案的另一種作法，經由 Microsoft Jet OLE DB Provider 資料庫引擎來存取，Jet 引擎可以支援多種以檔案為主的資料庫（file-based database），像是 Access、dBase 等等，以 SQL 指令為主的存取能力，在 Excel 上也可以實現，開發人員可以不用特別熟悉 Excel 物件模型，就可以控制 Excel 檔案的內容，不過它也不是一個好的伺服器端 Excel 解決方案，因為：

- 不支援資料流存取。

Jet 引擎和 Excel.exe 一樣，也只能使用實體檔案，無法使用資料流存取，因此也是要由開發人員自行管理使用過的檔案，若疏於管理的話，會讓伺服器上充斥許多的無用檔案。

- 控制資料的程度有限。

Jet 引擎雖然可以存取 Excel 檔案內容，但它畢竟不是 Excel 物件模型本身，對於 Excel 的控制無法做到跟物件模型完全相同的能力，而且也受限於 Jet 所支援的 SQL 指令，對於 Excel 檔案只能使用 SQL 指令來操控，因此像是樣式、條件以及高度依賴物件模型的功能，都沒有辦法被 Jet 控制。

由於官方本身所提供的 Excel 資料檔控制的方法基本上不支援伺服器端的處理，因此開始有很多第三方軟體廠商開始發展不需要 Excel 就可以處理的方法，像是 Aspose 的 Excel Library (Aspose.Cells)、或是在 ComponentSource 中可以找到的許多元件，多數都可以在不需要安裝 Excel 在伺服器的情況下就可以存取 Excel 檔案，不過這些都是要 \$\$ 的，所以有部份開發人員（包含開放原始碼陣營）亦發展出一些免費且開放原始碼的套件，像是 ExcelLibrary (on Google code) 以及 Java 陣營開發的 Apache POI 專案 (Apache POI Project)。

#### NOTE



Apache POI 專案是為 Java 所設計可支援 Office 系列檔案的存取類別庫，目前已經可以支援到 Excel 97-2003 XLS 與最新的 Excel 2007 XLSX 格式等，可至 Apache POI 專案網站參考它的功能：<http://poi.apache.org/>

### 在 .NET 上的 POI: NPOI

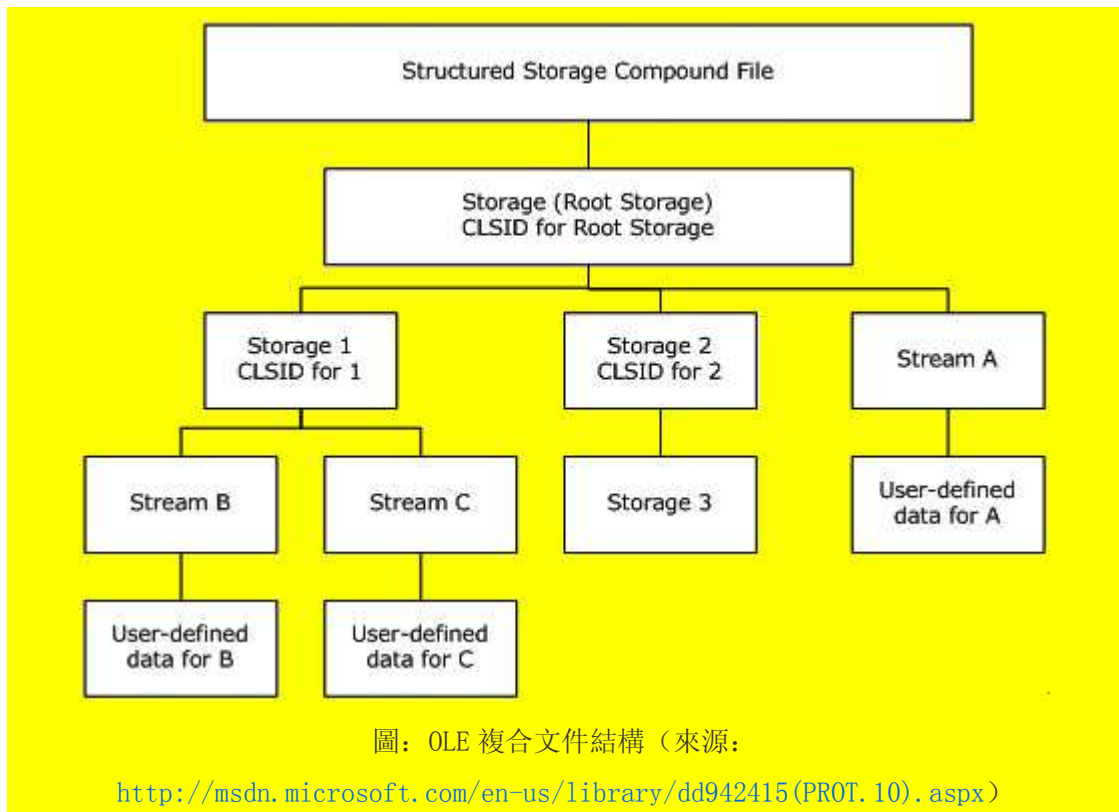
拜 Apache 的 POI 專案之賜，Java 的開發人員可以輕鬆的存取 Excel 檔案，而反觀 .NET 陣營幾乎只能在 Excel 物件模型以及 Jet 資料庫引擎中打轉，對於 .NET 陣營本身的開發人員似乎也不太公平，所以有幾位佛心來的開發者另外開發可直接存取 Excel 的函式庫，或是將 Java 中好用的函式庫移植到 .NET 環境來，POI 專案就是一例，在 .NET 上被稱為 NPOI。

POI 專案本身是處理 Office 檔案的函式庫，包含 Word、Excel、PowerPoint、Outlook、Visio、Publisher 等檔案，這些檔案都有一個共通的特性，就是它們都是微軟發展的 OLE Compound Document（複合文件），以 OLE Structured Storage（結構化儲存）格式儲存在檔案中，OLE 規範（以及處理 OLE API 呼叫等）對一般的開發人員來說是有相當的難度，因此利用 Excel 本身的物件模型是最容易的一件事。但 POI 專案並沒有使用到 Excel 的任何東西，它直接深入 OLE Compound Document 格式內去存取資料，也可以直接控制到各種儲存格的資訊（顏色，儲存格格式與樣式等），並將它物件導向化，外部開發人員只需要利用 POI 提供的屬性就可以控制 Office 格式的檔案資料。

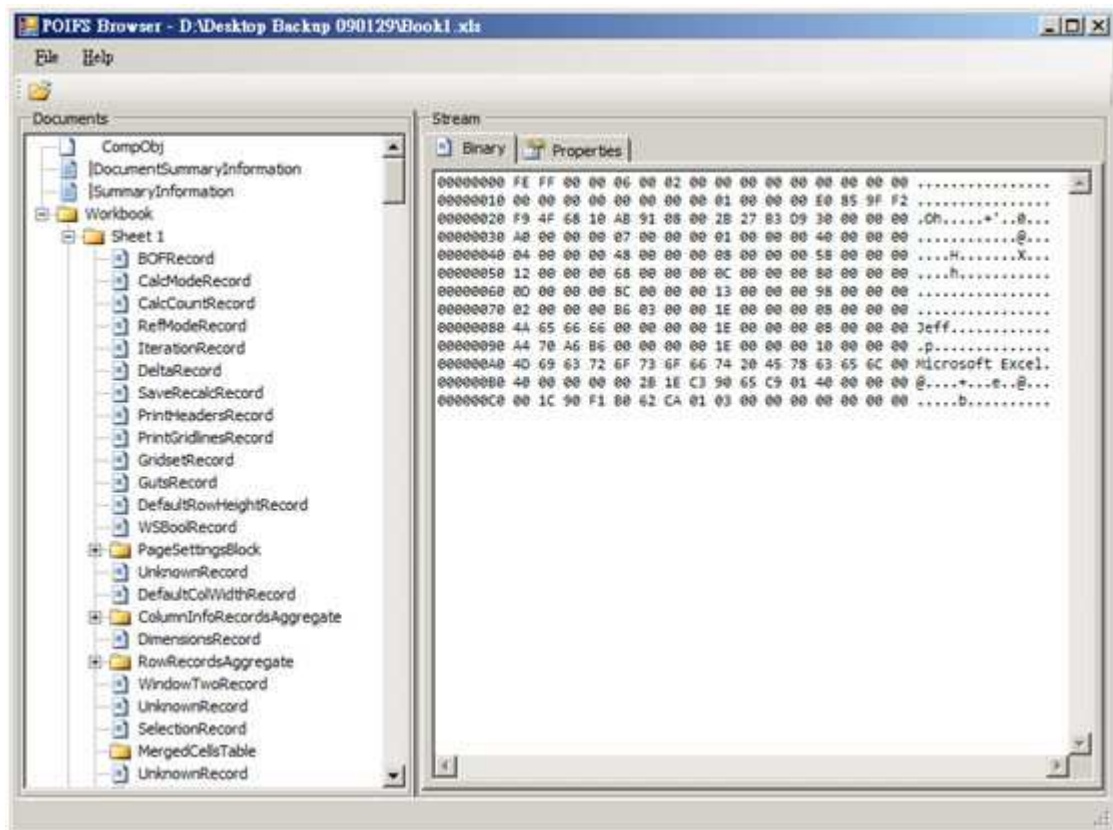
### NOTE

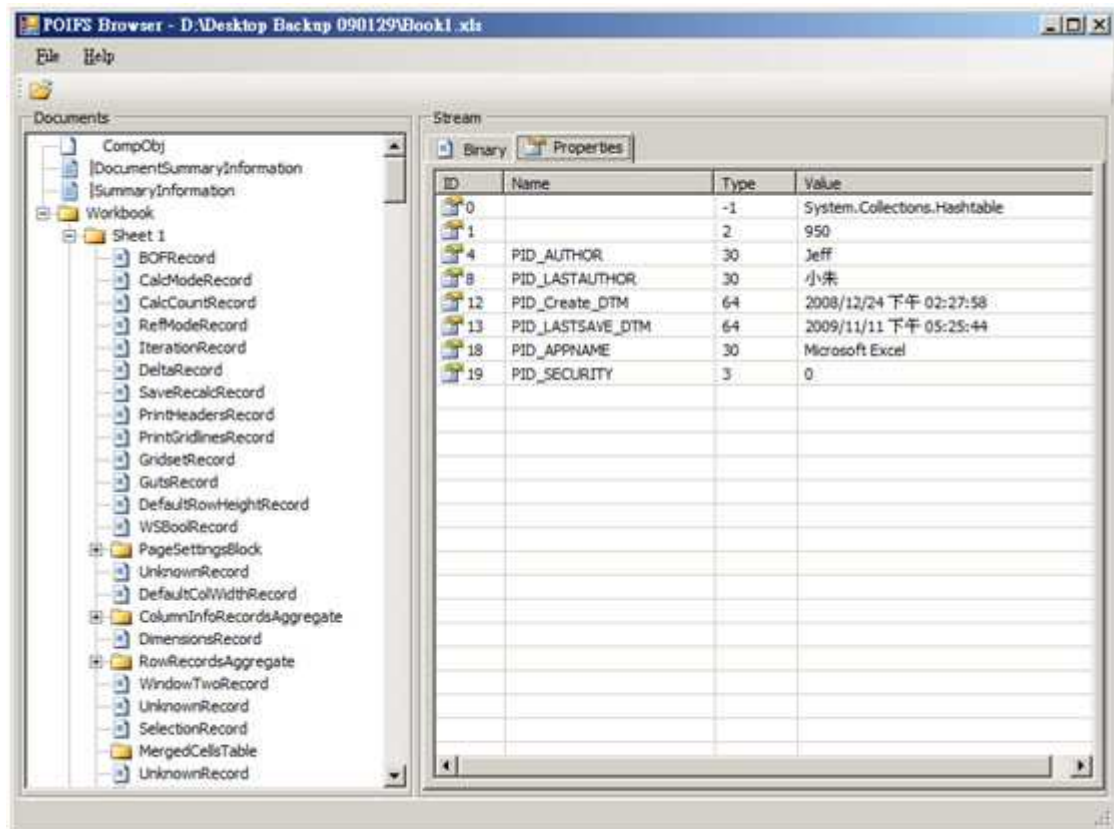
OLE Compound Document 是一種檔案儲存的格式，它是植基在 OLE 結構化儲存（Structured Storage）的基礎上，可以在同一個檔案資料流中儲存多種資料格式，以 Excel 為例，它可以同時儲存試算表（Spreadsheet）、圖表（Chart）、樣式（Style）、圖片（Pictures）以及方程式（Equation）等不同型式的資料，這些不同型式的資料都是由一組獨立格式的 CLSID 識別，再由 CLSID 在檔案區段中找出不同的 CLSID 儲存區，再深入儲存區讀出資料流，即可取回指定的資料。





如果讀者對 Excel 檔案的實際內容有興趣，可以在 NPOI 網站中下載 POIFS Explorer，並用它開啟 Excel 檔案，就可以看到 Excel 檔案的實際組成：





圖：POIFS Explorer

## NPOI 函式庫

NPOI 函式庫可以在 <http://npoi.codeplex.com/> 中下載，目前的版本為 1.2.1，有分為 .NET 1.1 與 .NET 2.0 以上版本兩種，支援主要的 POI 專案提供的功能，但專案中的範例程式碼都是以 Excel 為標的，原因應該是 Excel 在伺服器端的處理遠比 Word 和 PowerPoint 等文件要多太多了，故筆者在本篇文章也是以 Excel 檔案為主要說明的標的。NPOI 函式庫檔案有七個，分別是：

- NPOI.DLL：NPOI 核心函式庫。
- NPOI.DDF.DLL：NPOI 繪圖區讀寫函式庫。
- NPOI.HPSF.DLL：NPOI 文件摘要資訊讀寫函式庫。
- NPOI.HSSF.DLL：NPOI Excel BIFF 檔案讀寫函式庫。
- NPOI.Util.DLL：NPOI 工具函式庫。
- NPOI.POIFS.DLL：NPOI OLE 格式存取函式庫。
- ICSharpCode.SharpZipLib.DLL：檔案壓縮函式庫。

一般需要存取 Excel 97-2003 格式 (.xls) 的檔案時，需要使用 NPOI、NPOI.HSSF、NPOI.POIFS 與 NPOI.Util 函式庫，因此專案中要引用這四個 DLL，若要一併存取文件摘要資訊時，則也要引用 NPOI.HPSF.DLL 檔案，以取得必要的類別宣告。開發人員通常只要集

中精神在 `NPOI.HSSF.UserModel` 命名空間即可，它包含了控制 Excel 資料的各式類別物件供開發人員取用。

例如下列的 ASP.NET 程式碼可以生成一個空白的 Excel 檔案，並且添加三個指定名稱的試算表：

[C#]

```
HSSFWorkbook workbook = new HSSFWorkbook();

MemoryStream ms = new MemoryStream();

// 新增試算表。

workbook.CreateSheet("試算表 A");

workbook.CreateSheet("試算表 B");

workbook.CreateSheet("試算表 C");

workbook.Write(ms);

Response.AddHeader("Content-Disposition", string.Format("attachment;
filename=EmptyWorkbook.xls"));

Response.BinaryWrite(ms.ToArray());

workbook = null;

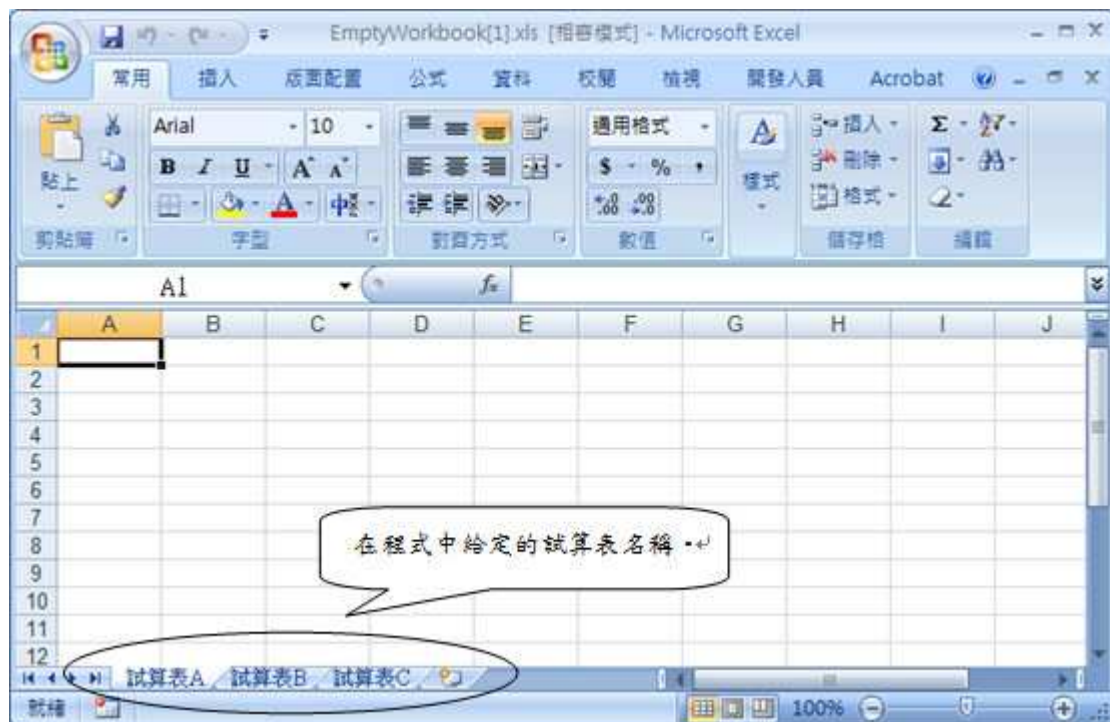
ms.Close();

ms.Dispose();
```

其執行結果就有如一般的檔案下載般，不過它的資料卻是一個完整的 Excel 資料檔：



將它用 Excel 打開來看, 可以看到它的內容確實是以指定的試算表名稱所建立:



再試一些程式, 我們可以在裡面添加資料, 例如下列的程式碼:

[C#]

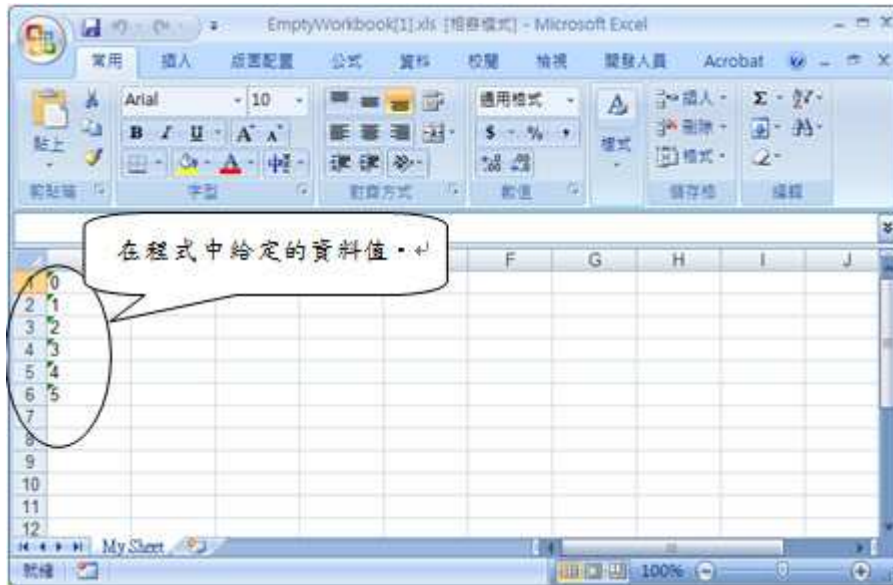
```
HSSFWorkbook workbook = new HSSFWorkbook();
```

```
MemoryStream ms = new MemoryStream();
```

```
// 新增試算表。
```

```
HSSFSSheet sheet = workbook.CreateSheet("My Sheet");  
  
// 插入資料值。  
  
sheet.CreateRow(0).CreateCell(0).SetCellValue("0");  
sheet.CreateRow(1).CreateCell(0).SetCellValue("1");  
sheet.CreateRow(2).CreateCell(0).SetCellValue("2");  
sheet.CreateRow(3).CreateCell(0).SetCellValue("3");  
sheet.CreateRow(4).CreateCell(0).SetCellValue("4");  
sheet.CreateRow(5).CreateCell(0).SetCellValue("5");  
  
workbook.Write(ms);  
  
Response.AddHeader("Content-Disposition", string.Format("attachment;  
filename=EmptyWorkbook.xls"));  
  
Response.BinaryWrite(ms.ToArray());  
  
workbook = null;  
  
ms.Close();  
  
ms.Dispose();
```

將它下載下來，用 Excel 開啟，即可看到插入的資料值：



這樣還不夠，我們再設定一些東西，例如設定儲存格的背景色：

[C#]

```
HSSFWorkbook workbook = new HSSFWorkbook();

MemoryStream ms = new MemoryStream();

// 新增試算表。
HSSFSheet sheet = workbook.CreateSheet("My Sheet");

// 建立儲存格樣式。
HSSFCellStyle style1 = workbook.CreateCellStyle();

style1.FillForegroundColor = NPOI.HSSF.Util.HSSFColor.BLUE.index2;
style1.FillPattern = HSSFCellStyle.SOLID_FOREGROUND;

HSSFCellStyle style2 = workbook.CreateCellStyle();

style2.FillForegroundColor = NPOI.HSSF.Util.HSSFColor.YELLOW.index2;
style2.FillPattern = HSSFCellStyle.SOLID_FOREGROUND;

// 設定儲存格樣式與資料。
HSSFCell cell = sheet.CreateRow(0).CreateCell(0);

cell.CellStyle = style1;
```

```
cell.SetCellValue(0);

cell = sheet.CreateRow(1).CreateCell(0);
cell.CellStyle = style2;
cell.SetCellValue(1);

cell = sheet.CreateRow(2).CreateCell(0);
cell.CellStyle = style1;
cell.SetCellValue(2);

cell = sheet.CreateRow(3).CreateCell(0);
cell.CellStyle = style2;
cell.SetCellValue(3);

cell = sheet.CreateRow(4).CreateCell(0);
cell.CellStyle = style1;
cell.SetCellValue(4);

workbook.Write(ms);

Response.AddHeader("Content-Disposition", string.Format("attachment;
filename=EmptyWorkbook.xls"));

Response.BinaryWrite(ms.ToArray());

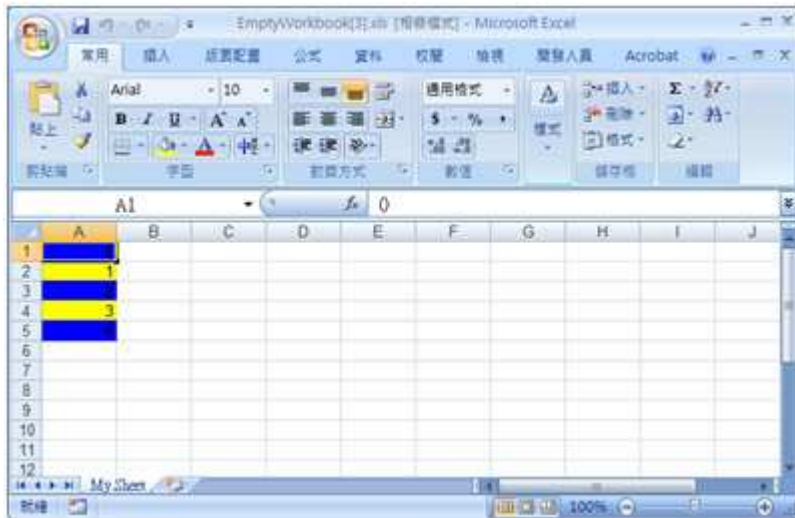
workbook = null;

ms.Close();

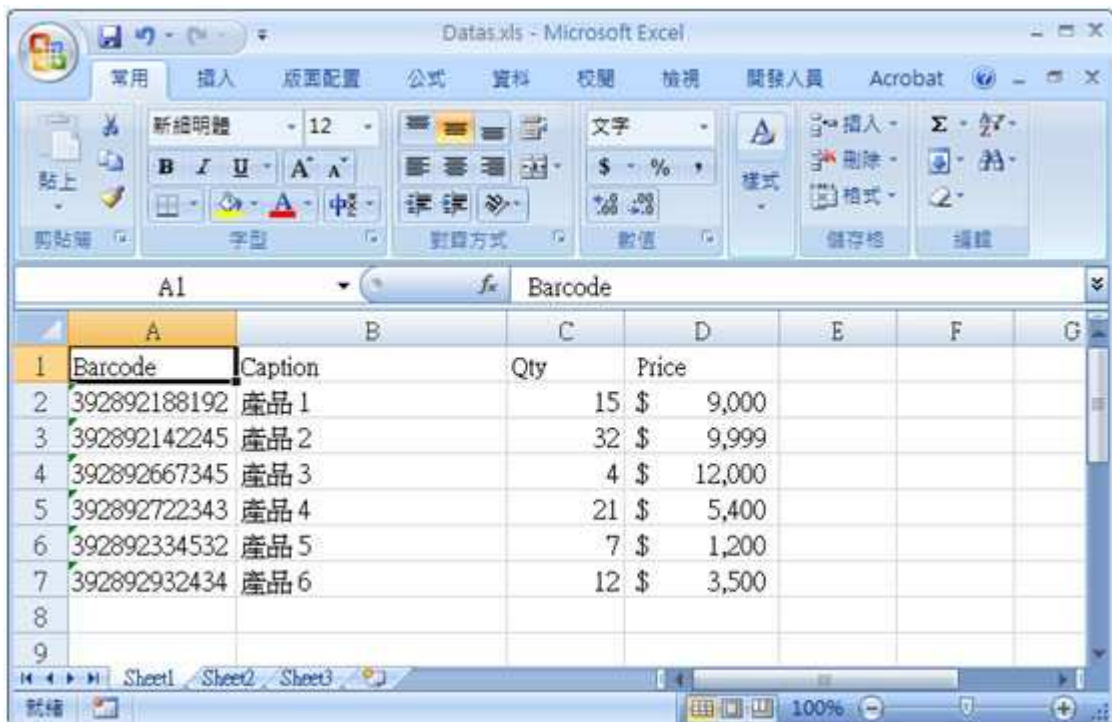
ms.Dispose();
```



將它下載下來，用 Excel 開啟，即可看到設定樣式的試算表：



輸出沒有問題，那麼輸入呢？當然也沒有問題啦。例如目前手上有一個 Datas.xls 資料檔，它的內容是：



然後利用下列的程式碼：

[C#]

```
if (this.fuUpload.HasFile)
{
```



```

HSSFWorkbook workbook = new HSSFWorkbook(this.fuUpload.FileContent);

HSSFSheet sheet = workbook.GetSheetAt(0);

DataTable table = new DataTable();

HSSFRow headerRow = sheet.GetRow(0);

int cellCount = headerRow.LastCellNum;

for (int i = headerRow.FirstCellNum; i < cellCount; i++)
{
    DataColumn column = new
DataColumn(headerRow.GetCell(i).StringCellValue);

    table.Columns.Add(column);
}

int rowCount = sheet.LastRowNum;

for (int i = (sheet.FirstRowNum + 1); i < sheet.LastRowNum; i++)
{
    HSSFRow row = sheet.GetRow(i);

    DataRow dataRow = table.NewRow();

    for (int j = row.FirstCellNum; j < cellCount; j++)
    {
        if (row.GetCell(j) != null)

            dataRow[j] = row.GetCell(j).ToString();
    }
}

```

```

    }

    table.Rows.Add(dataRow);
}

workbook = null;

sheet = null;

this.gvExcel.DataSource = table;

this.gvExcel.DataBind();
}

```

執行結果如下：



### 實例應用：將 DataTable 和 Excel 檔案間互轉

有了 NPOI 的支持，在伺服器端將資料轉換成 Excel 檔案的功能將不再是大問題，也無須再使用匯出 HTML 表格的方式來模擬 Excel 檔案的暫行方案來解決，只要使用 NPOI 就可以得到正規的 Excel 資料檔，筆者也特別撰寫了一個簡單的由 DataTable 物件自動轉成 Excel 資料檔的小程式供讀者自行取用。

[C#]

```
using System;

using System.Collections.Generic;

using System.Data;

using System.IO;

using System.Linq;

using System.Web;

using NPOI;

using NPOI.HPSF;

using NPOI.HSSF;

using NPOI.HSSF.UserModel;

using NPOI.POIFS;

using NPOI.Util;

public class DataTableRenderToExcel
{
    public static Stream RenderDataTableToExcel(DataTable SourceTable)
    {
        HSSFWorkbook workbook = new HSSFWorkbook();

        MemoryStream ms = new MemoryStream();

        HSSFSheet sheet = workbook.CreateSheet();

        HSSFRow headerRow = sheet.CreateRow(0);

        // handling header.

        foreach (DataColumn column in SourceTable.Columns)
```

```
headerRow.CreateCell(column.Ordinal).SetCellValue(column.ColumnName);

// handling value.
int rowIndex = 1;

foreach (DataRow row in SourceTable.Rows)
{
    HSSFRow dataRow = sheet.CreateRow(rowIndex);

    foreach (DataColumn column in SourceTable.Columns)
    {
dataRow.CreateCell(column.Ordinal).SetCellValue(row[column].ToString());

    }

    rowIndex++;
}

workbook.Write(ms);
ms.Flush();
ms.Position = 0;

sheet = null;
headerRow = null;
workbook = null;
```

```

        return ms;
    }

    public static void RenderDataTableToExcel(DataTable SourceTable, string
FileName)
    {
        MemoryStream ms = RenderDataTableToExcel(SourceTable) as MemoryStream;

        FileStream fs = new FileStream(FileName, FileMode.Create,
FileAccess.Write);

        byte[] data = ms.ToArray();

        fs.Write(data, 0, data.Length);

        fs.Flush();

        fs.Close();

        data = null;

        ms = null;

        fs = null;
    }

    public static DataTable RenderDataTableFromExcel(Stream ExcelFileStream,
string SheetName, int HeaderRowIndex)
    {
        HSSFWorkbook workbook = new HSSFWorkbook(ExcelFileStream);

        HSSFSheet sheet = workbook.GetSheet(SheetName);

```

```

DataTable table = new DataTable();

HSSFRow headerRow = sheet.GetRow(HeaderRowIndex);

int cellCount = headerRow.LastCellNum;

for (int i = headerRow.FirstCellNum; i < cellCount; i++)
{
    DataColumn column = new
DataColumn(headerRow.GetCell(i).StringCellValue);
    table.Columns.Add(column);
}

int rowCount = sheet.LastRowNum;

for (int i = (sheet.FirstRowNum + 1); i < sheet.LastRowNum; i++)
{
    HSSFRow row = sheet.GetRow(i);
    DataRow dataRow = table.NewRow();

    for (int j = row.FirstCellNum; j < cellCount; j++)
        dataRow[j] = row.GetCell(j).ToString();
}

ExcelFileStream.Close();

workbook = null;

sheet = null;

```

```

        return table;
    }

    public static DataTable RenderDataTableFromExcel(Stream ExcelFileStream, int
SheetIndex, int HeaderRowIndex)
    {
        HSSFWorkbook workbook = new HSSFWorkbook(ExcelFileStream);

        HSSFSheet sheet = workbook.GetSheetAt(SheetIndex);

        DataTable table = new DataTable();

        HSSFRow headerRow = sheet.GetRow(HeaderRowIndex);

        int cellCount = headerRow.LastCellNum;

        for (int i = headerRow.FirstCellNum; i < cellCount; i++)
        {
            DataColumn column = new
DataColumn(headerRow.GetCell(i).StringCellValue);

            table.Columns.Add(column);
        }

        int rowCount = sheet.LastRowNum;

        for (int i = (sheet.FirstRowNum + 1); i < sheet.LastRowNum; i++)
        {
            HSSFRow row = sheet.GetRow(i);

```

```

        DataRow dataRow = table.NewRow();

        for (int j = row.FirstCellNum; j < cellCount; j++)
        {
            if (row.GetCell(j) != null)
                dataRow[j] = row.GetCell(j).ToString();
        }

        table.Rows.Add(dataRow);
    }

    ExcelFileStream.Close();

    workbook = null;

    sheet = null;

    return table;
}
}

```

它的呼叫方法很簡單，若是要將 DataTable 輸出到 Excel 檔案，只要將 DataTable 丟給 RenderDataTableToExcel() 方法即可。

[C#]

```

DataTable table = new DataTable();

// 填充資料（由讀者自行撰寫）

// 產生 Excel 資料流。

MemoryStream ms = DataTableRenderToExcel.RenderDataTableToExcel(table) as

```



```

MemoryStream;

// 設定強制下載標頭。

Response.AddHeader("Content-Disposition", string.Format("attachment;
filename=Download.xls"));

// 輸出檔案。

Response.BinaryWrite(ms.ToArray());

ms.Close();

ms.Dispose();

```

若是要讀取 Excel 檔案並存到 DataTable，只要設定上傳的 Excel 檔案資料流、試算表索引（或名稱）以及起始列索引值即可：

[C#]

```

if (this.fuUpload.HasFile)
{
    // 讀取 Excel 資料流並轉換成 DataTable。

    DataTable table =
DataTableRenderToExcel.RenderDataTableFromExcel(this.fuUpload.FileContent, 1,
0);

    this.gvExcel.DataSource = table;

    this.gvExcel.DataBind();
}

```

## 結語

NPOI 是一個好用又簡單的函式庫，可以幫助開發人員解決長久以來在伺服端的 Excel 檔案產生與存取的問題，它還有很多特別的功能可以利用（像是讀寫摘要資料），正等著讀者發掘它呢。