Table样式

在表格中，影响对象显示的样式数量大大增加。除了正常的包含关系（表，和文件所有其他的元素，包含在另一个渲染对象，或在文档的正文最顶层），表中对象至少属于一个单元格，行或列，所有这一切都有自己的风格。此外，一个对象可以属于多个表格元素分组，这将变得更加复杂。关于表格中的样式如何工作，其细节将在"表格中的样式"主题中进行讨论。表格

表格由RenderTable类的实例表示。创建一个表格，只需要调用它的构造函数，比如像这样：
Visual Basic

```
Visual Basic
Dim rt1 As New C1.C1Preview.RenderTable()
```

C#

```
C#
RenderTable rt1 = new RenderTable();
```

C1PrintDocument 中的表格遵从Microsoft
Excel的模型。虽然新创建的表格从物理上是空的（也就是说，它不占用太多的内存空间），但是从逻辑上它是无限的：您可以访问一个表格中的任何元素（单元格，行或者列），而不需要事先创建并添加它们，向该对象写入将从逻辑上创建其之前的全部元素。例如，设置一个空白表格的位于索引值为9的行以及索引值为3的列所在位置的单元格的文本，将使得表格增长为10行4列。
向表格添加内容，您必须将单元格用数据进行填充。这可以通过以下方式之一完成：
通过设置单元格的RenderObject 这将插入指定的render对象至目标单元格。任何render对象可以添加到一个单元格，包括另外的一张表格，从而允许表格嵌套。
通过设置单元格的文本属性为一个字符串。这实际上是创建纯文本表格的一种方便快捷的方式，其内部创建了一个全新的RenderText 对象，设置单元格的RenderObject属性的值为该新建的RenderText，并设置该对象的文本属性为指定的字符串。
所以，例如，下面的代码片段将创建一个10行4列的一个表格：
Visual Basic

```
Visual Basic
Dim rt1 As New C1.C1Preview.RenderTable()
Dim row As Integer = 0
Do While (row < 10)
Dim col As Integer = 0 Do While (col < 4) rt1.Cells(row, col).Text = String.Format( _ "Text in cell({0}, {1})", row, col)
col += 1 Loop row += 1
```

Loop
C#

```
C#
RenderTable rt1 = new RenderTable(); for (int row = 0; row < 10; ++row) { for (int col = 0; col < 4; ++col) rt1.Cells[row, col].Text = string.Format( "Text in cell({0}, {1})", row, col);
}
```

在任何时候，您可以通过查询Cols.Count（返回当前列数）和Rows.Count（返回当前行数）属性的值，获取表格当前的尺寸大小。
访问单元格，列和行

和Tables主题中可以看到的示例代码一样，表格中全部的单元格由Cells集合表示，其类型为TableCellCollection。该集
合中的元素表示单个单元格，其类型为TableCell。为访问表格中的任意单元格，Cells集合可以按照单元格所在行和列的索引进行访问，就像这样：
Visual Basic

```
Visual Basic
Dim rt As New C1.C1Preview.RenderTable()

...
' 获取行索引10，列索引4的单元格:
Dim tc as TableCell = rt.Cells(10, 4)
```

C#

```
C#
RenderTable rt = new RenderTable();

...
// 获取行索引10，列索引4的单元格:
TableCell tc = rt.Cells[10, 4];
```

表格的列通过Cols集合进行访问，其类型为TableColCollection，它包含TableCol类型的元素。和单元格一样，只要触及某个列，将创建该列。例如，如果你设置一个列的Style属性，但如果该列不存在就将会创建该列。
表格的行通过Rows集合进行访问，其类型为TableRowCollection，它包含TableRow类型的元素。同单元格以及列一样，只要触及某个行，如果该行不存在将创建该行。例如，如果你设置一行的高度，则该行（以及它之前的所有行）将自动被创建。
请注意，所有的不包含具有实际内容的表格行将具有为零的高度，因此在呈现该表格时为不可见。

表格以及列宽，行高

C1PrintDocument表格中的行和列均可以为自动计算尺寸，但是行和列的默认行为有所不同。默认情况下，行的高度为自动计算（按照该行单元格的内容进行计算），而列的宽度为固定值。RenderTable的默认宽度下面的代码将创建一个页面等宽的表格，具有三个宽度相等的列，以及10行按照单元格的内容自动计算高度的行：
Visual Basic

```
Visual Basic
Dim rt As New C1.C1Preview.RenderTable() rt.Style.GridLines.All = LineDef.Default Dim row As Integer = 0
Do While (row < 10)
Dim col As Integer = 0 Do While (col < 3) rt.Cells(row, col).Text = String.Format( _ "Cell({0},{1})", row, col) col += 1
Loop row += 1 Loop doc.Body.Children.Add(rt)
```

C#

```
C#
RenderTable rt = new RenderTable(); rt.Style.GridLines.All = LineDef.Default; for (int row = 0; row < 10; ++row) for (int col = 0; col < 3; ++col) rt.Cells[row, col].Text = string.Format( "Cell({0}, {1})", row, col); doc.Body.Children.Add(rt);
```

使用一个全部自动计算尺寸的表格，相比默认设置，必须完成以下两件事情：
整个表格的宽度必须设置为自动（可以是字符串"auto"，或静态字段Unit.Auto）
表格的RenderTable.ColumnSizingMode必须设置为TableSizingModeEnum.Auto。
这是修改后的代码：
Visual Basic

```
Visual Basic
Dim rt As New C1.C1Preview.RenderTable() rt.Style.GridLines.All = LineDef.Default Dim row As Integer = 0

Do While (row < 10)
Dim col As Integer = 0 Do While (col < 3) rt.Cells(row, col).Text = String.Format( _
"Cell({0},{1})", row, col)

col += 1
Loop
row += 1
Loop
rt.Width = Unit.Auto
rt.ColumnSizingMode = TableSizingModeEnum.Auto doc.Body.Children.Add(rt)
```

C#

```
C#
RenderTable rt = new RenderTable(); rt.Style.GridLines.All = LineDef.Default; for (int row = 0; row < 10; ++row) for (int col = 0; col < 3; ++col) rt.Cells[row, col].Text = string.Format( "Cell({0}, {1})", row, col); rt.Width = Unit.Auto; rt.ColumnSizingMode = TableSizingModeEnum.Auto; doc.Body.Children.Add(rt);
```

修改后的代码使得表格中的每一列的宽度适显示应该列中单元格全部文本的宽度。

行和列的分组，页眉和页脚

元素组表功能是表格提供的强大功能。分组允许将表格的若干元素作为一个整体访问（例如，可以为分组设置样式，就好像它是一个独立的元素）。支持列的分组，行的分组以及单元格分组。
访问分组行，请使用RowGroups集合（其类型为TableVectorGroupCollection）。该集合的元素类型为TableVectorGroup。另一个有趣的属性是ColumnHeader该属性允许指定一组行为一个表格页眉，将在每一个新页面或者页面分栏的顶部进行重复。一个相关的属性是ColumnFooter，它允许指定一组行为一个表格页脚，同样也可以在每一页或者每一个页面分栏的末尾重复显示。
以下代码行显示如何指定一个表格的开始两行为表格页眉，在每一个分页符或者分栏符之后重复显示（这里的rt1是一个RenderTable对象）：
Visual Basic

```
Visual Basic
rt1.RowGroups(0, 2).Header = C1.C1Preview.TableHeaderEnum.Page
```

C#

```
C#
```

```
rt1.RowGroups[0, 2].Header = C1.C1Preview.TableHeaderEnum.Page;
```

如上所述，在TableVectorGroupCollection中的索引第一个值是包括在该组中的第一行的索引（在上面的代码示例中，为0）。第二个值表示分组中的行数（在上面的代码示例中，为2）。
访问分组列，请使用ColGroups集合该集合和行分组的集合具有相同的类型（TableVectorGroupCollection）特别感兴趣的是指定一组列为垂直的表格页眉或页脚的能力。C1PrintDocument
支持"水平的"或者（"extension"）的页面，允许宽对象水平横跨多个页面。为允许一个对象（例如，一个表）水平横跨几个页面，需要设置其SplitHorzBehavior
属性为一个不为SplitBehaviorEnum.Never的值如果对象的宽度超过页面宽度越宽，它将被分割为若干个水平页。特别是，一个过宽的表可以这样分割。为了让一组列沿着每一个页面的左侧重复显示，需要设置分组的ColumnHeader为了让一组列沿着每一页的右边缘重复显示，需要设置分组的ColumnFooter属性为True。

注意：注意：虽然表格的任何一组行（或列）可以被指定为页脚，通常你会想只包括表格的最后一行（或列）至页脚组。这将确保页脚的行为作为一个正常的页脚，只出现在页面的底部（或右边缘），也出现在表的末尾。（如果，例如，你把一个表的第一行指定为页脚，它仍然会出现在表格的开始位置，因此也不会打印在表格的结束位置。）

下面是一个示例代码，将创建一个具有100行10列的表格，设置表格的宽度为Auto，显式设置每列的宽度为1英寸，并指定水平和垂直表格页眉和页脚：
Visual Basic

Visual Basic

```
'创建并填充表格.
Dim rt1 As C1.C1Preview.RenderTable = New C1.C1Preview.RenderTable()
Dim row As Integer = 0
Dim col As Integer Do While (row < 100) col = 0 Do While (col < 6) rt1.Cells(row, col).Text = String.Format("Text in
cell({0}, {1})", row, col) col += 1 Loop row += 1 Loop

' 设置表格和列的宽度
rt1.Width = C1.C1Preview.Unit.Auto col = 0 Do While (col < 6) rt1.Cols(col).Width = "1in" col += 1 Loop

' 指定前两行作为页眉，同时设置其背景.
rt1.RowGroups(0, 2).PageHeader = True rt1.RowGroups(0, 2).Style.BackColor = Color.Red

'指定最后两行作为页脚，同时设置其背景.
rt1.RowGroups(98, 2).PageFooter = True rt1.RowGroups(98, 2).Style.BackColor = Color.Blue

'指定第一列作为页眉.
rt1.ColGroups(0, 1).PageHeader = True rt1.ColGroups(0, 1).Style.BackColor = Color.BlueViolet

' 指定最后一列作为页眉.
rt1.ColGroups(5, 1).PageFooter = True rt1.ColGroups(5, 1).Style.BackColor = Color.BurlyWood
```

C#

C#

```
//创建并填充表格.
RenderTable rt1 = new RenderTable(); for (int row = 0; row < 100; ++row) { for (int col = 0; col < 6; ++col) {
rt1.Cells[row, col].Text = string.Format("Text in cell({0}, {1})", row, col); }

}

// 设置表格和列的宽度
rt1.Width = Unit.Auto; for (int col = 0; col < 6; ++col) { rt1.Cols[col].Width = "1in"; }
// 指定前两行作为页眉，同时设置其背景.
rt1.RowGroups[0, 2].PageHeader = true; rt1.RowGroups[0, 2].Style.BackColor = Color.Red;
//指定最后两行作为页脚，同时设置其背景.
rt1.RowGroups[98, 2].PageFooter = true; rt1.RowGroups[98, 2]. Style.BackColor = Color.Blue;
//指定第一列作为页眉.
rt1.ColGroups[0, 1].PageHeader = true; rt1.ColGroups[0, 1].Style.BackColor = Color.BlueViolet;
// 指定最后一列作为页眉.
rt1.ColGroups[5, 1].PageFooter = true; rt1.ColGroups[5, 1].Style.BackColor = Color.BurlyWood;
```

在本示例中，背景色用来突出显示行和列的分组。
用户单元格分组
单元格，甚至表格中彼此互不相邻的单元格，可以被统一到一个分组。然后您可以通过一条命令，设置该分组中全部单元格的样式。定义一个用户单元格分组：

1. 创建UserCellGroup类型的一个对象该类型具有多个重载的构造器，允许您指定即将包含在分组中的单元格的坐标（全部的单元格应当在构造器中添加至分组）。
2. 将创建的UserCellGroup对象添加到表格的UserCellGroups集合中。
3. 现在您可以设置分组的样式。这将影响到该分组中的所有的单元格。

表格中的样式
虽然表格的单元格，列，以及行不是render对象（它们没有继承自RenderObject）但是，它们都具有Style属性。

操纵样式将影响到相关的元素以及其全部的内容。设置一行的样式将影响到该行中全部的单元格。设置一列的样式将影响到该列中全部的单元格。位于行和列交叉位置的单元格的样式将会是指定到该行以及该列的样式的一个组合结果。
如果同一个样式属性同时在行上和列上进行设置，则列上的属性值将起作用。
此外，分组（行分组，列分组以及用户单元格分组）都有自己的样式，这也影响到单元格中的数据显示，同时也影响表格的行和列。
以下规则支配表格中样式的应用：
环境样式属性穿透表格元素进行传播（整个表格，行以及列分组，单元格分组，单独的行和列，以及单个单元格），基于几何上的包含关系，这一点和表格外部的render对象的包含关系继承环境样式属性的机制是类似的。
环境属性影响单元格的内容，而不影响这些容器元素。例如，设置整个表格的样式上的字体，将影响此表格中全部的文本，除非在某个低层次显式地设置了其字体。类似地，设置一行的样式的字体将影响该行内所有单元格的字体。
当一个特定的环境属性两个以上的表格元素改变时，以下优先级顺序将用来计算最终用来绘制单元格的属性的有效值：单元格自身的样式（具有最高的优先级）
UserCellGroup样式，如果单元格包含在其中 列样式
列分组样式，如果存在的话 行样式 行分组样式，如果存在的话
表格样式（具有最低的优先级）
设置在表格元素样式上的非环境属性如上面所列（整个表格，列和行的分组，行，列以及单元格），将应用到这些元素自身，而不会影响单元格的内容，即使这些元素不是render对象（整个表格对象除外）。例如，为了在表格中的一行绘制一个边框，可以设置该行的Style.Border的值为期望值。
为了设置表格中全部单元格的非环境样式属性，请使用RenderTable.CellStyle。如果指定了该属性，该样式将实际上做为单元格内render对象样式的父样式。
CellStyle属性同样定义在行，列以及表格元素分组上，如果指定了这些样式，将影响单元格内部对象的非环境属性。例如，设置在一个表格中的所有单元格的背景图像，可以设置表格的CellStyle.BackgroundImage属性这将在表格中的全部
单元格中重复该图像，而设置表格的Style.BackgroundImage属性则会将该图像做为整个表格的背景（如果图像为拉伸显示，则两种模式下区别相当明显）。

锚点和超链接

Reports for WinForms 超链接。超链接可以附加到render对象上（RenderObject以及其派生类），以及段落对象上
（ParagraphObject以及其派生类），并可以链接到：
位于当前文档的锚点。
位于另一个C1PrintDocument文档内的锚点。
位于当前文档内的一个位置。
一个外部文件。
当前文档中的某一页。
一个用户事件。
C1.Win.C1Preview 程序集中的各个预览控件支持超链接（C1PreviewPane，C1PrintPreviewControl 以及
C1PrintPreviewDialog）。当一个具有超链接的文档进行预览时，鼠标悬停到一个超链接上方会导致光标变成一个手的形状。按照链接目标的不同，单击这个超链接会有以下可能的效果，跳转至文档中的其他位置，打开另一个文档并跳转至其中的某个位置，打开一个外部文件，或者调用一个用户事件。

注意：注意：在以下主题中的代码片段均已经假设在文件中添加了"using C1.C1Preview"指令（C#语法；或者其他语法

的等效语句），因此我们可以只书写类名部分（比如说RenderText）而不是使用完全限定类型名称
（C1.C1Preview.RenderText）。

添加一个到同一个文档内部一个锚点的超链接

您需要做两件事情以便将一个文档中的一个部分链接到另一个部分：
创建一个链接指向的位置（称作一个锚点）。
为文档的另一个部分添加一个链接（一个超链接）指向此位置（当然，可以让几个不同的超链接指向同一个锚点）。
为了在一个render对象上创建一个锚点，您可以向该render对象的Anchors集合添加一个元素（C1Anchor类型）。例如，如果rt是一个RenderTable对象，则您可以添加以下代码：
Visual Basic

Visual Basic
rt.Anchors.Add(New C1.C1Preview.C1Anchor("anchor1"))

C#

C#
rt.Anchors.Add(new C1Anchor("anchor1"));

这将在这个render
table对象上定义一个叫做anchor1的锚点（该名称用作引用该锚点）。为了在另一个render对象上创建一个链接，比如说一个RenderText对象，指向此对象，您可以书写以下代码：
Visual Basic

Visual Basic
Dim rtxt As New C1.C1Preview.RenderText() rtxt.Text = "Link to anchor1" rtxt.Hyperlink = New C1.C1Preview.C1Hyperlink("anchor1")

C#

C#
RenderText rtxt = new RenderText(); rtxt.Text = "Link to anchor1"; rtxt.Hyperlink = new C1Hyperlink("anchor1");

当然，您必须添加两个相关的render对象（一个包含锚点，另一个包含该超链接）至该文档。
Hyperlink是RenderObject类的一个属性，此类是全部render对象的基类，因此，和上面显示的方式相同，任何一个
render对象可以通过设置该属性转换为一个超链接。

添加一个到另一个C1PrintDocument中的某个锚点的超链接

将一个文档中的一个位置链接到另一个文档的某个位置，您需要按照以下步骤：•
如上面描述到的那样，向目标文档添加一个锚点，生成该文档并将其保存为C1D格式的文档至磁盘。您可以通过预览控件上的Save按钮保存该文档，或者使用文档本身的Save方法通过代码进行保存。
添加一个指向另外一个文档中锚点的链接，这和如何添加一个内部链接的方式很类似。唯一不同的是，除了目标锚点的名称之外，您还必须提供包含该文档的文件名。
这是段完整的程序代码，它将创建一个具有锚点的文档，并保存为磁盘文件（myDocument1.c1d），之后创建另外一个文档，添加一个到第一个文档中的锚点的链接，最后在预览对话框显示第二个文档：
Visual Basic

```
Visual Basic
```

```
' 在目标文档创建一个锚点
Dim targetDoc As New C1.C1Preview.C1PrintDocument Dim rt1 As New C1.C1Preview.RenderText("This is anchor1 in
myDocument1.") rt1.Anchors.Add(New C1.C1Preview.C1Anchor("anchor1")) targetDoc.Body.Children.Add(rt1) targetDoc.Generate()
targetDoc.Save("c:\myDocument1.c1d")

' 向文档添加一个指向该锚点的超链接.
Dim sourceDoc As New C1.C1Preview.C1PrintDocument
Dim rt2 As New C1.C1Preview.RenderText("This is hyperlink to myDocument1.")
Dim linkTarget As C1.C1Preview.C1LinkTarget = New
C1.C1Preview.C1LinkTargetExternalAnchor("c:\myDocument1.c1d", "anchor1") rt2.Hyperlink = New
C1.C1Preview.C1Hyperlink(linkTarget) sourceDoc.Body.Children.Add(rt2) sourceDoc.Generate()

' 在预览中显示具有超链接的文档.
Dim preview As New C1.Win.C1Preview.C1PrintPreviewDialog() preview.Document = sourceDoc preview.ShowDialog()
```

C#

```
C#
```

```
// 在目标文档创建一个锚点
C1PrintDocument targetDoc = new C1PrintDocument(); RenderText rt1 = new RenderText("This is anchor1 in myDocument1.");
rt1.Anchors.Add(new C1Anchor("anchor1")); targetDoc.Body.Children.Add(rt1); targetDoc.Generate();
targetDoc.Save(@"c:\myDocument1.c1d");

// 向文档添加一个指向该锚点的超链接.
C1PrintDocument sourceDoc = new C1PrintDocument();
RenderText rt2 = new RenderText("This is hyperlink to myDocument1.");
C1LinkTarget linkTarget = new C1LinkTargetExternalAnchor(@"c:\myDocument1.c1d", "anchor1");
```

```
rt2.Hyperlink = new C1Hyperlink(linkTarget); sourceDoc.Body.Children.Add(rt2); sourceDoc.Generate();

// 在预览中显示具有超链接的文档.
C1PrintPreviewDialog preview = new C1PrintPreviewDialog(); preview.Document = sourceDoc; preview.ShowDialog();
```

注意以下几点：•
该锚点的创建方式和在同一个文档内部创建链接的方式使用类似的方式。实际上，压根没有任何区别；同一个锚点可以同时作为来自于同一个文档或者不同文档的链接的目标。
为保存此文档，我们使用了Save方法。该方法将文档保存为原生的C1PrintDocument格式，默认的文件扩展名为C1D。按照该格式保存的文件之后可以加载到C1PrintDocument对象以便对其进一步进行处理，或者使用
ComponentOne打印预览控件进行预览。
在创建超链接之前，必须创建一个目标对象，并传递给超链接对象的构造器。从C1LinkTarget基类型派生了若干
不同的链接目标类型。对于外部的锚点，应当使用C1LinkTargetExternalAnchor类型。链接目标包含跳转到该链接所需要的信息，在本示例中，将包含文档所在的文件名以及其中的锚点名称。

添加一个到当前文档某个位置的超链接

您可以不创建任何锚点而添加到相同文档的一个链接。此时可以使用通过一个render对象直接创建一个
C1LinkTargetExternalAnchor链接目标，就像下面这样，ro1表示当前文档中任意一个render对象：
Visual Basic

```
Visual Basic
```

```
Dim linkTarget = New C1.C1Preview.C1LinkTargetDocumentLocation(ro1)
```

C#

```
C#
```

```
C1LinkTarget linkTarget = new C1LinkTargetDocumentLocation(ro1);
```

设置超链接的链接目标将导致拥有该超链接的render对象在发生鼠标单击时，跳转到指定的render对象。比方说，假定
ro2是一个您希望转换为超链接的render对象，以下代码将链接到ro1所在的位置。这里的linkTarget就是上面的代码段所创建的对象：
Visual Basic

```
Visual Basic
rt2.Hyperlink = New C1.C1Preview.C1Hyperlink() rt2.Hyperlink.LinkTarget = linkTarget
```

C#
C#
rt2.Hyperlink = new C1Hyperlink(); rt2.Hyperlink.LinkTarget = linkTarget;
注意在本示例中，超链接的LinkTarget属性必须在此超链接创建之后进行设置。添加一个到外部文件的超链接
!MISSING PHRASE 'Show All'! !MISSING PHRASE 'Hide All'!
一个到外部文件的超链接和一个链接到外部文件的锚点的超链接唯一不同的是链接目标的类型不同。指向一个外部文件
链接的链接目标类型叫做C1LinkTargetFile。单击这样一个链接将使用Windows
Shell打开该文件。例如，如果在前面一个章节的示例代码中，您替换创建外部锚点链接的代码为如下代码：
C#

```
C#
C1LinkTarget linkTarget = new C1LinkTargetFile(@"c:\");
```

则单击之前的超链接会在Windows Explorer中间打开C:盘的根目录。以下是完整的程序：
Visual Basic

```
Visual Basic
' 创建一个具有指向外部文件超链接的文档.
Dim doc As New C1.C1Preview.C1PrintDocument
Dim rt As New C1.C1Preview.RenderText("Explore drive C:...")
Dim linkTarget As C1.C1Preview.C1LinkTarget = New C1.C1Preview.C1LinkTargetFile("c:\") rt.Hyperlink = New
C1.C1Preview.C1Hyperlink(linkTarget) doc.Body.Children.Add(rt) doc.Generate()

' 在预览中显示具有超链接的文档
Dim preview As New C1.Win.C1Preview.C1PrintPreviewDialog() preview.Document = doc preview.ShowDialog()
```

To write code in C#

```
C#
// 创建一个具有指向外部文件超链接的文档.
C1PrintDocument doc = new C1PrintDocument();
RenderText rt = new RenderText("Explore drive C:..."); C1LinkTarget linkTarget = new C1LinkTargetFile(@"c:\");
rt.Hyperlink = new C1Hyperlink(linkTarget); doc.Body.Children.Add(rt); doc.Generate();

// 在预览中显示具有超链接的文档
C1PrintPreviewDialog preview = new C1PrintPreviewDialog();
```

preview.Document = doc; preview.ShowDialog();
添加一个到同一个文档中某个页面的超链接

可以使用C1LinkTargetPage链接对象添加一个到同一个文档其他页面的超链接，而不需要定义任何锚点。支持以下页面跳转逻辑：
跳转到文档首页。 跳转到文档尾页。
跳转到上一页。 跳转到下一页。
跳转到指定的页码。
从当前页面跳转指定的页数
例如，可以使用以下代码创建一个跳转到文档首页的链接目标：
Visual Basic

```
Visual Basic
Dim linkTarget = New
C1.C1Preview.C1LinkTargetPage(C1.C1Preview.PageJumpTypeEnum.First)
```

C#

```
C#
C1LinkTarget linkTarget = new C1LinkTargetPage(PageJumpTypeEnum.First);
```

这里，PageJumpTypeEnum可以指定页面跳转的类型（当然，对于需要指定一个绝对或者相对的页码的跳转命令，您还必须使用接受这些参数的构造器）。
该功能提供在文档自身各个页面之间进行导航的简单方式。例如，您可以向文档页脚添加类似DVD播放器的控件（跳转至首页，跳转至上一页，跳转至下一页，跳转至尾页）。
添加到用户事件的超链接

最后您可以在C1PreviewPane上添加一个将触发一个事件的超链接，该事件可以被您的代码逻辑捕获并处理。您应当通过C1LinkTargetUser做到这一点。以下是演示此概念的完整示例代码：
Visual Basic

```
Visual Basic
```

```
Private Sub UserLinkSetup()

' 创建一个具有用户事件超链接的文档.
Dim doc As New C1.C1Preview.C1PrintDocument
```

```
Dim rt As New C1.C1Preview.RenderText("Click this to show message box...") Dim linkTarget As C1.C1Preview.C1LinkTarget =
New C1.C1Preview.C1LinkTargetUser rt.Hyperlink = New C1.C1Preview.C1Hyperlink(linkTarget) rt.Hyperlink.UserData = "My
hyperlnk user data" doc.Body.Children.Add(rt) doc.Generate()

' 创建预览.
Dim preview As New C1.Win.C1Preview.C1PrintPreviewDialog()

' 向UserHyperlinkJump事件关联事件处理程序
AddHandler preview.PreviewPane.UserHyperlinkJump, New
C1.Win.C1Preview.HyperlinkEventHandler(AddressOf Me.C1PreviewPane1_UserHyperlinkJump)

' 预览此文档.
preview.Document = doc preview.ShowDialog()End Sub
Private Sub C1PreviewPane1_UserHyperlinkJump(ByVal sender As Object, ByVal e As
C1.Win.C1Preview.HyperlinkEventArgs) Handles C1PreviewPane1.UserHyperlinkJump
MessageBox.Show(e.Hyperlink.UserData.ToString())
End Sub
```

C#

```
C#
```

```
private void UserLinkSetup() {

// 创建一个具有用户事件超链接的文档.
C1PrintDocument doc = new C1PrintDocument();
RenderText rt = new RenderText("Click this to show message box...");

C1LinkTarget linkTarget = new C1LinkTargetUser(); rt.Hyperlink = new C1Hyperlink(linkTarget); rt.Hyperlink.UserData = "My
hyperlnk user data"; doc.Body.Children.Add(rt); doc.Generate();

// 创建预览.
C1PrintPreviewDialog preview = new C1PrintPreviewDialog();

// 向UserHyperlinkJump事件关联事件处理程序 preview.PreviewPane.UserHyperlinkJump += new
HyperlinkEventHandler(PreviewPane_UserHyperlinkJump);

// 预览此文档.
preview.Document = doc; preview.ShowDialog(); } private void PreviewPane_UserHyperlinkJump(object sender,
HyperlinkEventArgs e)
```

```
{
MessageBox.Show(e.Hyperlink.UserData.ToString());
}
```

该示例将在单击超链接时，弹出一个消息对话框，显示设置给超链接的UserData属性的字符串。（在本示例中，将会显示"超链接自定义用户数据示例"）。
链接目标类型继承关系为了总结超链接章节，这里是链接目标类型的继承关系列表：

Class			Description
C1LinkTarget			整个继承关系树的基类。
	C1LinkTargetAnchor		描述链接目标为当前文档中间的一个锚点。
		C1LinkTargetExternalAnchor	描述链接目标为另一个文档中间的一个锚点。
	C1LinkTargetDocumentLocation		描述链接目标为一个render对象。
	C1LinkTargetFile		描述链接目标为可以被OS Shell打开的外部文件。
	C1LinkTargetPage		描述链接目标为当前文档的一个页面。
	C1LinkTargetUser		描述链接目标为一个在C1PreviewPane上调用的用户事件处理函数。

表达式，脚本，标签

表达式（或脚本 - 这两个术语会在这里交替使用）可以在整个文档的不同的地方使用。通过一对方括号标记一个表达式，如下面的代码所示：
Visual Basic

```
Visual Basic
Dim doc As New C1PrintDocument()
doc.Body.Children.Add(New RenderText("2 + 2 = [2+2]"))
```

C#

```
C#
C1PrintDocument doc = new C1PrintDocument();
doc.Body.Children.Add(new RenderText("2 + 2 = [2+2]"));
```

此代码将在生成的文档中产生以下文字：
2 + 2 = 4
这种情况下， "[2+2]" 被解释为一个表达式，并计算出最终的文本。

注意：注意：如果表达式解析器无法解析包含在方括号内的文本，比方说文本无法被理解为一个表达式，则将文本原样

插入文档。
标签

标签与表达密切相关。事实上，标签是变量，可在表达式中使用，或者简单地作为表达式。标签允许您在希望向某个目标位置插入某个特定的字符串，但尚未确定该字符串的值的情况下，使用一个占位符。一个标签的典型例子是页码，您
需要打印页码，但是目前还无法确认页码到底显示什么，或者在文档重新生成时，页码还可能发生变化。
标签有两个主要的属性：Name 以及 Value。Name属性来识别标签，而Value属性则是表示标签将被替换为的值。
C1PrintDocument 提供两种标签：预定义标签和自定义标签。预定义的标签包括：
[PageNo] -将使用当前的页码替代。
[PageCount] -将使用总页数替代。
[PageX] -使用当前水平方向上的页码替代。
[PageXCount] -使用当水平方向上的总页数替代。
[PageY] -使用当前垂直方向的页码替代（如果不存在水平方向上的分页，则这和 [PageNo]结果相等）。
[PageYCount] -使用当前垂直方向上的总页数替代（如果不存在水平方向上的分页，则这和 [PageCount]结果相等）。
自定义标签存储在文档的Tags集合中。为了向该集合添加一个标签，您可以使用下面的代码：
Visual Basic

```
Visual Basic
doc.Tags.Add(New C1.C1Preview.Tag("tag1", "tag value"))
```

C#

```
C#
doc.Tags.Add(new C1.C1Preview.Tag("tag1", "tag value"));
```

当创建标签时标签的值可以保留为未指定状态，并可以在稍后的某一个时机指定它（即使是在标签应用到文本之后的某个时机）。
在希望使用标签的位置，将标签的名字包含在一对方括号中以使用该标签，例如，就像这样：
Visual Basic

```
Visual Basic
Dim rt As New C1.C1Preview.RenderText()
t.Text = "The value of tag1 will appear here: [tag1]."
```

C#

```
C#
RenderText rt = new RenderText();
rt.Text = "The value of tag1 will appear here: [tag1].";
```

标签/表达式语法

如果需要，您可以将用来包括标签或者用文本表示的脚本的方括号改成任意字符串，通过文档的 TagOpenParen 以及
TagCloseParen
属性。在您的文档可能包含大量正常方括号字符串时，这种修改能力是一种很好的主意，因为默认情况下，这些方括号将触发表达式解析，即使不影响显示结果，也会消耗大量的资源。所以，您可以这样做：
Visual Basic

```
Visual Basic
doc.TagOpenParen = "@@@[" doc.TagCloseParen = "@@@]"
```

C#

```
C#
doc.TagOpenParen = "@@@["; doc.TagCloseParen = "@@@]";
```

将确保只有被"@@@[" 和"@@@]"包含的字符串被会被解释为表达式。
表达式的括号也可以通过在之前添加转义字符进行转义，例如以下代码：
Visual Basic

```
Visual Basic
Dim doc As New C1PrintDocument()
doc.Body.Children.Add(new RenderText("2 + 2 = \\[2+2
]"))
```

C#

```
C#
C1PrintDocument doc = new C1PrintDocument();
doc.Body.Children.Add(new RenderText("2 + 2 = \\[2+2
]"));
```

将在生成的文档中产生以下文本：
2 + 2 = [2+2]
因为括号已被转义。
文档的TagEscapeString可以被用作将转义字符指定为任意字符串。
在运行时编辑标签值
您可以给最终用户显示一个窗体，允许查看并编辑位于C1PrintDocument组件的Tags集合中的标签的标签值。有几个新成员支持该选项，允许您创建一个自定义的窗体，用来显示特定的标签。
对于您创建的标签窗体，您可以设置几个选项。你可以：
每次C1PrintDocument生成之后，允许用户编辑每一个标签。更多信息，请参见显示所有标签。
让用户能够编辑一部分标签，而不是全部。可以在不希望用户编辑的标签上，设置其Flags属性的值为None。
最后您可以选择最终用户何时能够看到这个标签对话框。设置文档上的ShowTagsInputDialog属性为False，并在您希望用户看到这个标签值输入对话框的时机调用EditTags()方法。
显示所有标签

默认情况下，ShowTagsInputDialog属性设置为false，标签对话框不显示。为了让用户在每一次C1PrintDocument文档生成时输入全部的标签，可以设置文档上的ShowTagsInputDialog属性为True。您所添加到文档的Tags集合的任何标签都将被自动展示在一个对话框里，每次即将生成文本时将显示给用户进行编辑。这使得最终用户有机会在Tags对话框编辑每一个标签值。
例如，下面的代码在 Form_Load事件向文档增加了三个标签，并向这些标签指定了文本值：
Visual Basic

```
Visual Basic
Dim doc As New C1PrintDocument()
Me.C1PrintPreviewControl1.Document = doc
' 在文档生成时显示Tags对话框.
doc.ShowTagsInputDialog = True
' 创建将在Tags对话框中显示的标签
doc.Tags.Add(New C1.C1Preview.Tag("Statement", "Hello World!")) doc.Tags.Add(New C1.C1Preview.Tag("Name", "ComponentOne"))

doc.Tags.Add(New C1.C1Preview.Tag("Location", "Pittsburgh, PA"))
' 向文档添加标签并生成文档.
Dim rt As New C1.C1Preview.RenderText() rt.Text = "[Statement] My name is [Name] and my current location is [Location]."
doc.Body.Children.Add(rt) doc.Generate()
```

C#

```
C#
```

```
C1PrintDocument doc = new C1PrintDocument(); this.c1PrintPreviewControl1.Document = doc;
// 在文档生成时显示Tags对话框.
doc.ShowTagsInputDialog = true;
// 创建将在Tags对话框中显示的标签
doc.Tags.Add(new C1.C1Preview.Tag("Statement", "Hello World!")); doc.Tags.Add(new C1.C1Preview.Tag("Name",
"ComponentOne"));
doc.Tags.Add(new C1.C1Preview.Tag("Location", "Pittsburgh, PA"));
// 向文档添加标签并生成文档.
C1.C1Preview.RenderText rt = new C1.C1Preview.RenderText(); rt.Text = "[Statement] My name is [Name] and my current
location is [Location]."; doc.Body.Children.Add(rt); doc.Generate();
```

当应用程序运行时，以下对话框将在文档生成之前显示：

在任何文本框中改变文本将改变最终生成的文档中实际出现的文本。如果保留默认的文本，将在生成的文档产生以下文字：
Hello World! My name is ComponentOne and I'm currently located in Pittsburgh, PA.

显示特定的标记

当ShowTagsInputDialog属性设置为True时，默认情况下全部的标签将在Tags对话框中显示。您可以通过设置Tag.ShowInDialog属性防止用户编辑某些特定的标签。为了让用户仅编辑部分标签，在不希望用户编辑的标签上设置标签的Flags属性的值为None。
例如，下面的代码在Form_Load事件中向文档添加三个标签，其中一个无法编辑：
Visual Basic

```
Visual Basic
```

```
Dim doc As New C1PrintDocument()
Me.C1PrintPreviewControl1.Document = doc
' 在文档生成时，显示Tags对话框
doc.ShowTagsInputDialog = True
' 创建一个标签，但不要在Tags对话框显示它
doc.Tags.Add(New C1.C1Preview.Tag("Statement", "Hello World!")) doc.Tags("Statement").ShowInDialog = False
'将显示创建的标签
doc.Tags.Add(New C1.C1Preview.Tag("Name", "ComponentOne"))
doc.Tags.Add(New C1.C1Preview.Tag("Location", "Pittsburgh, PA"))
' 向文档添加标签并生成文档.
Dim rt As New C1.C1Preview.RenderText() rt.Text = "[Statement] My name is [Name] and my current location is [Location]."
doc.Body.Children.Add(rt) doc.Generate()
```

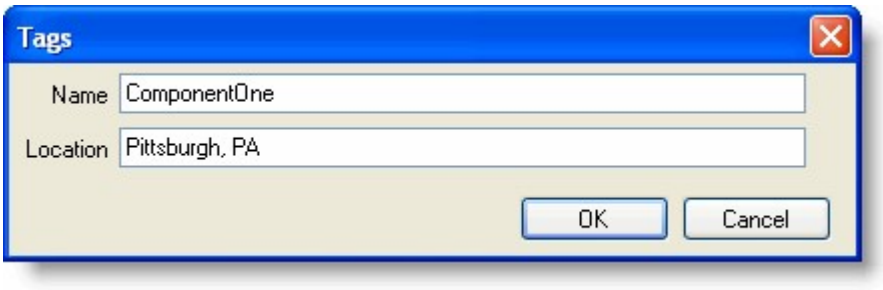
C#

```
C#
```

```
C1PrintDocument doc = new C1PrintDocument();

this.c1PrintPreviewControl1.Document = doc;
// 在文档生成时，显示Tags对话框
doc.ShowTagsInputDialog = true;
```

```
<span style="color: #008000">// 创建一个标签，但不要在Tags对话框显示它</span>
doc.Tags.Add(new C1.C1Preview.Tag("Statement", "Hello World!")); doc.Tags["Statement"].ShowInDialog = false;
<span style="color: #008000">//将显示创建的标签</span>
doc.Tags.Add(new C1.C1Preview.Tag("Name", "ComponentOne"));
doc.Tags.Add(new C1.C1Preview.Tag("Location", "Pittsburgh, PA"));
<span style="color: #008000">// 向文档添加标签并生成文档.</span>
C1.C1Preview.RenderText rt = new C1.C1Preview.RenderText(); rt.Text = "[Statement] My name is [Name] and my current location
is [Location]."; doc.Body.Children.Add(rt); doc.Generate();
```

当应用程序运行时，下面的对话框将在文本生成之前显示：



改变Tags对话框中任何文本框中的文本，将改变出现在生成文档中的文本。请注意，Statement标签将不显示，也不能从对话框中修改。如果保留默认的文本，则将在生成的文档中产生以下文字：

Hello World! My name is ComponentOne and I'm currently located in Pittsburgh, PA.

指定Tags对话框何时显示

Unable to render embedded object: File (MISSING PHRASE 'Show All') not found. Unable to render embedded object: File (MISSING PHRASE 'Hide All') not found.

当ShowTagsInputDialog属性设置为True，则会在文档生成之前显示Tags对话框。您可以通过编程方式调用EditTags方法显示该对话框，（和ShowTagsInputDialog属性的设置是完全独立的）。例如，下面的代码将会使得单击一个按钮时，显示的标签输入对话框：

Visual Basic

```
Visual Basic
Public Class Form1
Dim doc As New C1PrintDocument()
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
Me.C1PrintPreviewControl1.Document = doc
' 创建将用来显示的标签
doc.Tags.Add(New C1.C1Preview.Tag("Statement", "Hello World!")) doc.Tags("Statement").ShowInDialog = True
doc.Tags.Add(New C1.C1Preview.Tag("Name", "ComponentOne")) doc.Tags.Add(New C1.C1Preview.Tag("Location", "Pittsburgh,
PA")) ' 向文档添加标签
Dim rt As New C1.C1Preview.RenderText() rt.Text = "[Statement] My name is [Name] and my current location is
[Location]."

doc.Body.Children.Add(rt) End Sub
Private Sub EditTagsNow_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles EditTagsNow.Click ' 在单击按钮时显示Tags对话框 doc.ShowTagsInputDialog = True doc.EditTags()
End Sub
Private Sub GenerateDocNow_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles
GenerateDocNow.Click doc.ShowTagsInputDialog = False
' 单击按钮时生成文档
doc.Generate() End Sub
End Class
```

C#

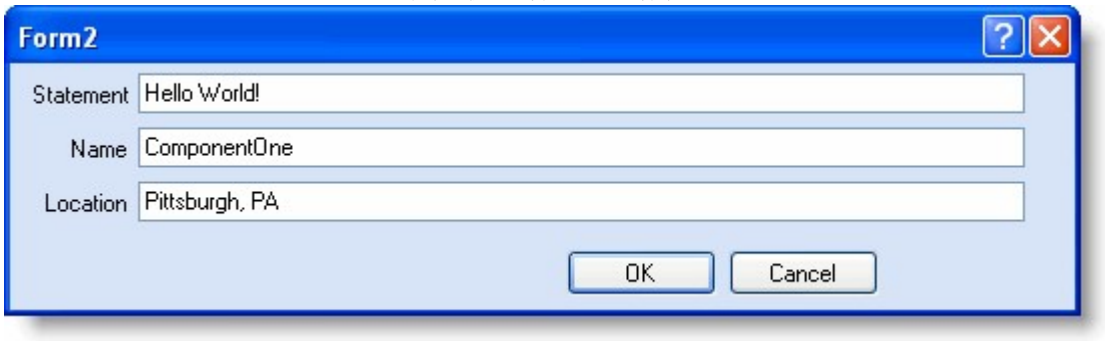
```
C#
public partial class Form1 : Form { public Form1() {
InitializeComponent();
} C1PrintDocument doc = new C1PrintDocument(); private void Form1_Load(object sender, EventArgs e) {
this.c1PrintPreviewControl1.Document = doc;
// 创建将用来显示的标签
doc.Tags.Add(new C1.C1Preview.Tag("Statement", "Hello World!")); doc.Tags["Statement"].ShowInDialog = true;
doc.Tags.Add(new C1.C1Preview.Tag("Name", "ComponentOne")); doc.Tags.Add(new C1.C1Preview.Tag("Location", "Pittsburgh,
PA")); // 向文档添加标签

C1.C1Preview.RenderText rt = new C1.C1Preview.RenderText(); rt.Text = "[Statement] My name is [Name] and my current
location is [Location].";

doc.Body.Children.Add(rt);

}

private void EditTagsNow_Click(object sender, EventArgs e)

{
// 在单击按钮时显示Tags对话框
doc.ShowTagsInputDialog = true; doc.EditTags(); } private void GenerateDoc_Click(object sender, EventArgs e) {
doc.ShowTagsInputDialog = false;
// 单击按钮时生成文档
```

```
doc.Generate();
```

}
}
在上面的例子中，单击EditTagsNow按钮时，会显示Tags对话框。
定义默认Tags对话框
通过向工程添加继承自TagsInputForm或者TagsInputFormBase的窗体，您可以很容易地自定义Tags对话框。具体使用哪一种方式取决于您是打算对窗体做一些小改动还是希望彻底地重新定义该窗体。

做一些小小的变动

如果您仅仅是希望在默认的窗体上做一些小改动（比如，添加一个帮助按钮），您可以添加一个继承自TagsInputForm的窗体到工程，按照需要对其进行调整，并指定该窗体的类型名至文档的TagsInputDialogClassName属性。
例如，在下面的窗体中，标题栏上添加了一个帮助按钮，并且窗体的背景色发生了变化：



彻底改变窗体

如果您希望，您可以彻底改变默认的窗体。例如，您可以提供您自己的用来输入标签值的控件，等等。为了达到这一点，您需要将Form继承自TagsInputFormBase，按照需要修改，并重写EditTags方法。
脚本/表达式语言
!MISSING PHRASE 'Show All'!
!MISSING PHRASE 'Hide All'!
表达式使用的语言由C1PrintDocument.ScriptingOptions.Language属性的值决定。这个属性可以是以下几个值之一：
VB. 这是默认值，表示使用标准VB.NET作为脚本语言。
C1Report. 该值表示将使用C1Report脚本语言。该语言类似VB，但是有一些细微的差别。该选项主要为了提供向后兼容性。
CSharp. 该值表示将使用标准的C#作为脚本语言。
如果用VB语言作为表达式语言，则当文档生成时，将在内部为每一个表达式构件一个独立的程序集，每一个程序集包含一个从ScriptExpressionBase派生的类型。该类型包含protected的属性，可以被表达式使用。表达式本身是作为一个类的函数实现。
例如：
Visual Basic

```
Visual Basic
```

```
Dim doc As New C1PrintDocument()
doc.ScriptingOptions.Language = ScriptLanguageEnum.CSharp Dim rt As New RenderText("[PageNo == 1 ? ""First"" : ""Not first""]") doc.Body.Children.Add(rt)
```

C#

```
C#
```

```
C1PrintDocument doc = new C1PrintDocument();
doc.ScriptingOptions.Language = ScriptLanguageEnum.CSharp; RenderText rt = new RenderText("[PageNo == 1 ? \"First\" : \"Not first\"]"); doc.Body.Children.Add(rt);
```

程序集和命名空间
!MISSING PHRASE 'Show All'! !MISSING PHRASE 'Hide All'!
默认情况下，下列程序集对脚本可用（引用）：
System
System.Drawing
为了添加另一个程序集（系统或自定义）到脚本引用的程序集列表，请将其添加至文档的C1PrintDocument.ScriptingOptions.ExternalAssemblies集合。例如，以下代码将添加一个到System.Data程序集的引用至文档的脚本：
Visual Basic

```
Visual Basic
```

```
Dim doc As New C1PrintDocument()
doc.ScriptingOptions.ExternalAssemblies.Add("System.Data.dll")
```

C#

```
C#
```

```
C1PrintDocument doc = new C1PrintDocument();
doc.ScriptingOptions.ExternalAssemblies.Add("System.Data.dll");
```

下列命名空间在脚本中是默认可用的（导入的）：•
System
System.Collections
System.Collections.Generic
System.Text
Microsoft.VisualBasic System.Drawing
要添加另一个命名空间，将其添加到文档的C1PrintDocument.ScriptingOptions.Namespaces集合。例如，这将允在文档的脚本中使用ystem.Data命名空间声明的类型，不需要书写完全类型的全名：
Visual Basic

```
Visual Basic
```

```
Dim doc As New C1PrintDocument()
doc.ScriptingOptions.  Namespaces.Add("System.Data")
```

C#

```
C#
```

```
C1PrintDocument doc = new C1PrintDocument(); doc.ScriptingOptions.  Namespaces.Add("System.Data");
```

文本表达式对ID的访问
!MISSING PHRASE 'Show All'! !MISSING PHRASE 'Hide All'!
如上面提到的，括号内的表达式可以在RenderText以及ParagraphText对象的Text属性中使用。
Document （C1PrintDocument类型）
这个变量的引用生成的文档。这可以用在许多方面，例如，下面的代码将打印当前文档的作者：
Visual Basic

```
Visual Basic
```

```
Dim doc As New C1PrintDocument()
Dim rt As New RenderText("Landscape is " + _ "[Iif(Page.PageSettings.Landscape,\"TRUE\",\"FALSE\")].")
doc.Body.Children.Add(rt)
```

C#

```
C#
```

```
C1PrintDocument doc = new C1PrintDocument();
RenderText rt = new RenderText("Landscape is " + "[Iif(Page.PageSettings.Landscape,\"TRUE\",\"FALSE\")].");
doc.Body.Children.Add(rt);
```

RenderObject （RenderObject类型）
这个变量引用当前的render对象。例如，下面的代码将打印当前render对象的名称：
Visual Basic

```
Visual Basic
```

```
Dim doc As New C1PrintDocument()
Dim rt As New RenderText( _ "The object's name is [RenderObject.Name]") rt.Name = "MyRenderText" doc.Body.Children.Add(rt)
```

C#

```
C#
```

```
C1PrintDocument doc = new C1PrintDocument();
RenderText rt = new RenderText( "The object's name is [RenderObject.Name]"); rt.Name = "MyRenderText";
doc.Body.Children.Add(rt);
```

Page（C1Page类型）
这个变量引用当前页面（C1Page类型的对象）。而脚本中最常使用的关于页面对象的成员则可以直接访问（参见下面的PageNo，PageCount等属性），其他数据则可以通过Page变量进行访问，比如当前页面的设置。例如，下面的代码将在当前页面布局是水平方向打印时打印"Landscape is TRUE"，否则将打印"Landscape is FALSE"：
Visual Basic

```
Visual Basic
```

```
Dim doc As New C1PrintDocument()
Dim rt As New RenderText("Landscape is " + _ "[Iif(Page.PageSettings.Landscape,\"TRUE\",\"FALSE\")].")
doc.Body.Children.Add(rt)
```

C#

```
C#
C1PrintDocument doc = new C1PrintDocument();
RenderText rt = new RenderText("Landscape is " + "[Iif(Page.PageSettings.Landscape,\"TRUE\",\"FALSE\")].");
doc.Body.Children.Add(rt);
```

PageNo（整数类型）
此名称解析为从1开始计数的页码。相当于 Page.PageNo。
PageCount（整数类型）
此名称解析为文档的总页数。相当于Page.PageCount。例如，下面的代码可以用于产生通用的"第X页，共Y 页"的页眉：
Visual Basic

```
Visual Basic
Dim doc As New C1PrintDocument() doc.PageLayout.PageHeader = New RenderText( _

"Page [PageNo] of [PageCount]")
```

C#

```
C#
C1PrintDocument doc = new C1PrintDocument(); doc.PageLayout.PageHeader = new RenderText(
```

"Page [PageNo] of [PageCount]");
PageX （整数类型）
此名称解析为当前水平方向上的从1开始计算的页码。（对于没有水平分页符的文档，将返回1。）
PageY （整数类型）
此名称解析为当前垂直方向上的页码。（对于没有水平分页符的文档，则和PageNo相同。）
PageXCount （整数类型）
此名称解析为文档水平方向的分页总数。（对于没有水平分页符的文档，将始终返回1。）
PageYCount （整数类型）
此名称解析为文档的总页数。（对于没有水平分页符的文档，则和PageCount相等。）
需要着重注意的是，这里提到的任何和页面编码相关的值可以用在文档的任何地方，不一定必须放在页眉或页脚上。
Fields（FieldCollection类型）
该变量引用数据库所有可用字段的集合，类型为C1.C1Preview.DataBinding.FieldCollection。它只能用于数据绑定的文档。例如，下面的代码将打印NWIND数据库Products数据表的产品名称列表。
Visual Basic

```
Visual Basic
Dim doc As New C1PrintDocument() Dim dSrc As New DataSource() dSrc.ConnectionProperties.DataProvider =
DataProviderEnum.OLEDB dSrc.ConnectionProperties.ConnectString = _ "Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=NWIND.MDB" Dim dSet1 As New C1.C1Preview.DataBinding.DataSet( _ dSrc, "select * from Products")
doc.DataSchema.DataSources.Add(dSrc) doc.DataSchema.DataSets.Add(dSet1) Dim rt As New RenderText()
rt.DataBinding.DataSource = dSet1 rt.Text = "[Fields!ProductName.Value]" doc.Body.Children.Add(rt)
```

C#

```
C#
C1PrintDocument doc = new C1PrintDocument(); DataSource dSrc = new DataSource(); dSrc.ConnectionProperties.DataProvider =
DataProviderEnum.OLEDB; dSrc.ConnectionProperties.ConnectString = @"Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=NWIND.MDB"; C1.C1Preview.DataBinding.DataSet dSet1 = new C1.C1Preview.DataBinding.DataSet(dSrc, "select * from
Products"); doc.DataSchema.DataSources.Add(dSrc);
```

doc.DataSchema.DataSets.Add(dSet1); RenderText rt = new RenderText(); doc.Body.Children.Add(rt); rt.DataBinding.DataSource =
dSet1; rt.Text = "[Fields!ProductName.Value]";
请注意在最后一行是如何通过"! "访问字段数组的元素。或者，你可以写：
Visual Basic

```
Visual Basic
rt.Text = "[Fields(\"ProductName\").Value]"
```

C#

```
C#
rt.Text = "[Fields(\"ProductName\").Value]";
```

但使用"!"符号更短更容易阅读。
Aggregates （AggregateCollection类型）
这个变量来访问的文档中定义的汇总的集合。集合的类型为C1.C1Preview.DataBinding.AggregateCollection，其中的元素类型为C1.C1Preview.DataBinding.Aggregate。例如，下面的代码将在产品的列表后打印平均单价：
Visual Basic

```
Visual Basic
Dim doc As New C1PrintDocument() Dim dSrc As New DataSource() dSrc.ConnectionProperties.DataProvider = DataProviderEnum.OLEDB dSrc.ConnectionProperties.ConnectString = _ "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=NWIND.MDB" C1.C1Preview.DataBinding.DataSet dSet1 = _ new C1.C1Preview.DataBinding.DataSet(dSrc, _ "select * from Products") doc.DataSchema.DataSources.Add(dSrc) doc.DataSchema.DataSets.Add(dSet1) Dim rt As New RenderText() doc.Body.Children.Add(rt) rt.DataBinding.DataSource = dSet1 rt.Text = "[Fields!ProductName.Value]" doc.DataSchema.Aggregates.Add(new Aggregate( _ "AveragePrice", "Fields!UnitPrice.Value", _ rt.DataBinding, RunningEnum.Document, _ AggregateFuncEnum.Average)) doc.Body.Children.Add(new RenderText( _ "Average price: [Aggregates!AveragePrice.Value]"))
```

C#

```
C#
C1PrintDocument doc = new C1PrintDocument();
```

DataSource dSrc = new DataSource(); dSrc.ConnectionProperties.DataProvider = DataProviderEnum.OLEDB;
dSrc.ConnectionProperties.ConnectString = @"Provider=Microsoft.Jet.OLEDB.4.0;Data Source=NWIND.MDB";
C1.C1Preview.DataBinding.DataSet dSet1 = new C1.C1Preview.DataBinding.DataSet(dSrc, "select * from Products");
doc.DataSchema.DataSources.Add(dSrc); doc.DataSchema.DataSets.Add(dSet1); RenderText rt = new RenderText();
doc.Body.Children.Add(rt); rt.DataBinding.DataSource = dSet1; rt.Text = "[Fields!ProductName.Value]";
doc.DataSchema.Aggregates.Add(new Aggregate("AveragePrice", "Fields!UnitPrice.Value", rt.DataBinding, RunningEnum.Document, AggregateFuncEnum.Average)); doc.Body.Children.Add(new RenderText("Average price: [Aggregates!AveragePrice.Value]"));
DataBinding（C1DataBinding类型）
该变量允许访问当前render对象的DataBinding属性，类型为C1.C1Preview.DataBinding.C1DataBinding。例如，
下面的代码（从展示Fileds变量用法的示例修改而来）将通过使用render对象的DataBinding属性的RowNumber
成员在文本表达式中生成一个带有编码的列表：
Visual Basic

```
Visual Basic
Dim doc As New C1PrintDocument() Dim dSrc As New DataSource() dSrc.ConnectionProperties.DataProvider = DataProviderEnum.OLEDB dSrc.ConnectionProperties.ConnectString = _ "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=NWIND.MDB" C1.C1Preview.DataBinding.DataSet dSet1 = _ new C1.C1Preview.DataBinding.DataSet(dSrc, _ "select * from Products") doc.DataSchema.DataSources.Add(dSrc) doc.DataSchema.DataSets.Add(dSet1) Dim rt As New RenderText() rt.DataBinding.DataSource = dSet1 rt.Text = "[DataBinding.RowNumber]: [Fields!ProductName.Value]" doc.Body.Children.Add(rt)
```

C#

```
C#
C1PrintDocument doc = new C1PrintDocument(); DataSource dSrc = new DataSource(); dSrc.ConnectionProperties.DataProvider = DataProviderEnum.OLEDB; dSrc.ConnectionProperties.ConnectString = @"Provider=Microsoft.Jet.OLEDB.4.0;Data Source=NWIND.MDB"; C1.C1Preview.DataBinding.DataSet dSet1 = new C1.C1Preview.DataBinding.DataSet(dSrc,
```

"select * from Products");
doc.DataSchema.DataSources.Add(dSrc); doc.DataSchema.DataSets.Add(dSet1);
RenderText rt = new RenderText(); rt.DataBinding.DataSource = dSet1;
rt.Text = "[DataBinding.RowNumber]: [Fields!ProductName.Value]"; doc.Body.Children.Add(rt);
筛选，分组和排序中的ID访问
在筛选，分组和排序表达式，下面的ID的子集的可被访问：

Document （ C1PrintDocument类型）
DataBinding（C1DataBinding类型）
Fields（FieldCollection类型）
对于这些对象类型的详细信息，请参见文本表达式对ID的访问。
表达式访问ID用作指定Dataset中指定的计算字段
在用作指定数据集指定的计算字段的表达式中，可以访问以下的ID的子集：
Document （C1PrintDocument类型）
Fields （FieldCollection类型）
对于这些对象类型的详细信息，请参见文本表达式中可访问的ID。
数据绑定
除了可以通过代码完整的创建一个C1PrintDocument之外，也可以通过数据绑定方式创建C1PrintDocument文档。在这种情况下，实际的文件将在生成时，使用来自于数据库的数据填充之后产生。
关于数据绑定最主要的属性是位于RenderObject上的DataBinding属性，类型为C1DataBinding，允许指定由此render对象显示的数据的数据源。除此之外，数据绑定可以表示该render对象必须为数据源中全部的记录重复显示，在这种情况下，render对象成为了类似于一种条带状报表生成器的"条带"。这一点和Microsoft的RDL定义类似。
因此，数据绑定文件，文档生成包括两个阶段：•
所有的数据绑定的render对象被选中（做为模版）并基于数据创建"真正的"render对象。
产生的文档做为一个非数据绑定的文档进行分页。
文档可以包含数据库Schema（由C1DataSchema类型表示，包括数据库连接信息，SQL查询，等等）在内。文档中的C1DataBinding对象可以引用该Schema的属性。如果文档中全部的数据绑定对象仅引用文档本身的C1DataSchema的属性，则该文档将变成"数据可重排的"，指的是，该文档可以独立于用来生成它的程序，使用来自于数据源的数据完全重新更新并生成。
同时，C1DataBinding可以引用由Form或者任何其他创建该C1PrintDocument的程序创建的现有数据源（DataTable等）。当然，在这种情况下，文档只能在该程序的上下文中，通过更新后的数据源重新生成，同时，保存并在稍后再次打开该文档（C1D或C1DX文件）将打断和该数据的连接关系。
Render对象上的数据绑定
Unable to render embedded object: File (MISSING PHRASE 'Show All') not found. Unable to render embedded object: File (MISSING PHRASE 'Hide All') not found.
当创建一个render对象时，其数据绑定是没有初始创建的。它将在DataBinding属性被用户代码引用时创建。例如：
Visual Basic

```
Visual Basic
Dim rt As RenderText = New RenderText ' ...

If Not (rt.DataBinding Is Nothing) Then
MessageBox.Show("Data binding defined.")
End If
```

C#

```
C#
RenderText rt = new RenderText(); // ... if (rt.DataBinding != null) {

MessageBox.Show("Data binding defined.");
}
```

以上代码的条件将始终计算为True。因此，如果您只想检查是否在某个特定的render对象上存在数据绑定，您应当使用DataBindingDefined属性替代：
Visual Basic

```
Visual Basic
Dim rt As RenderText = New RenderText ' ...

If rt.DataBindingDefined Then
MessageBox.Show("Data binding defined.")
End If
```

C#

```
C#
RenderText rt = new RenderText(); // ... if (rt.DataBindingDefined) {
MessageBox.Show("Data binding defined.");
}
```

注意：注意：这一点和WinForms平台下Control类型的Handle以及IsHandleCreated属性类似。

在文档的生成过程中，将形成RenderObjectsList集合。结果可能是以下三种情况：
render对象的Copies属性为null。这意味着该对象没有进行数据绑定，应当按照常规方式进行处理。对象的Fragments属性可以用作访问生成的页面中，实际呈现的对象的分段。
render对象的Copies属性不为null，但是Copies.Count为零。这意味着对象已经进行了数据绑定，但数据集是空的（不包含任何记录）。在这种情况下，对象将完全不会在文档中显示，因为对象是不是在所有的文件，显示，无碎片产生的对象。请参见"数据绑定时绑定到空白列表"中

的示例。
render对象的Copies属性不为null，并且Copies.Count大于零。这意味着对象绑定至数据源，且数据源不为空（包含记录）。在文档生成过程中，将创建render对象的若干份复制并存在该集合中。这些复制将被处理并且每一份复制将照常生成一个分段。

数据绑定表

!MISSING PHRASE 'Show All'!
!MISSING PHRASE 'Hide All'!
通过使用TableVectorGroup的DataBinding属性，可以将一个RenderTable对象进行数据绑定。TableVectorGroup是表格行和列分组的基类。
关于数据绑定到RenderTable的示例，请参见安装在ComponentOne Samples文件夹中的DataBinding示例。
请注意，不仅行分组可以进行数据绑定，列分组也可以。也就是说，一个表格不仅仅可以向下增长，同时也可以横向增长。
分组将生效，但是请注意分组的层次关系基于TableVectorGroup对象的层次关系，如以下代码所示：
Visual Basic

Visual Basic

```
Dim rt As C1.C1Preview.RenderTable = New C1.C1Preview.RenderTable rt.Style.GridLines.All = C1.C1Preview.LineDef.Default


' 表头:
Dim c As C1.C1Preview.TableCell = rt.Cells(0, 0)
c.SpanCols = 3
c.Text = "Header"

' 分组页眉:
c = rt.Cells(1, 0)
c.Text = "GroupId = [Fields!GroupId.Value]"

c.SpanCols = 3
c.Style.TextAlignHorz = C1.C1Preview.AlignHorzEnum.Center
c.Style.TextAlignVert = C1.C1Preview.AlignVertEnum.Center

' 子分组页眉:
c = rt.Cells(2, 0)
c.Text = "GroupId = [Fields!GroupId.Value] SubGroupId = [Fields!SubGroupId.Value]" c.SpanCols = 3

c.Style.TextAlignHorz = C1.C1Preview.AlignHorzEnum.Center
c.Style.TextAlignVert = C1.C1Preview.AlignVertEnum.Center

' 子分组数据:
rt.Cells(3, 0).Text = "GroupId=[Fields!GroupId.Value]"
```

```
rt.Cells(3, 1).Text = "SubGroupId=[Fields!SubGroupId.Value]" rt.Cells(3, 2).Text = "IntValue=[Fields!IntValue.Value]"
<span style="color: #008000">' 创建一组数据绑定行，按照GroupId字段进行分组:</span>
Dim g As C1.C1Preview.TableVectorGroup = rt.RowGroups(1, 3)
g.CanSplit = True
g.DataBinding.DataSource = MyData.Generate(20, 0)
g.DataBinding.Grouping.Expressions.Add("Fields!GroupId.Value")
g.Style.BackColor = Color.LightCyan
<span style="color: #008000">' 创建一个嵌套分组，按照SubGroupId字段进行分组:</span>
Dim ng As C1.C1Preview.TableVectorGroup = rt.RowGroups(2, 2) ng.CanSplit = True ng.DataBinding.DataSource =
g.DataBinding.DataSource ng.DataBinding.Grouping.Expressions.Add("Fields!SubGroupId.Value") ng.Style.BackColor =
Color.LightPink
<span style="color: #008000">' 创建更深一层的嵌套数据绑定分组:</span>
Dim ng2 As C1.C1Preview.TableVectorGroup = rt.RowGroups(3, 1) ng2.DataBinding.DataSource = g.DataBinding.DataSource
ng2.Style.BackColor = Color.LightSteelBlue
```

C#

C#

```
RenderTable rt = new RenderTable(); rt.Style.GridLines.All = LineDef.Default;

// 表头:
TableCell c = rt.Cells[0, 0];
c.SpanCols = 3;
c.Text = "Header";

// 分组页眉:
c = rt.Cells[1, 0];
c.Text = "GroupId = [Fields!GroupId.Value]";
c.SpanCols = 3;
c.Style.TextAlignHorz = AlignHorzEnum.Center;
c.Style.TextAlignVert = AlignVertEnum.Center;

// 子分组页眉:
c = rt.Cells[2, 0];
c.Text = "GroupId = [Fields!GroupId.Value] SubGroupId = [Fields!SubGroupId.Value]"; c.SpanCols = 3;
c.Style.TextAlignHorz = AlignHorzEnum.Center;
c.Style.TextAlignVert = AlignVertEnum.Center;

// 子分组数据:
rt.Cells[3, 0].Text = "GroupId=[Fields!GroupId.Value]"; rt.Cells[3, 1].Text = "SubGroupId=[Fields!SubGroupId.Value]";
rt.Cells[3, 2].Text = "IntValue=[Fields!IntValue.Value]";
```

```
<span style="color: #008000">// 创建一组数据绑定行，按照GroupId字段进行分组:</span>
TableVectorGroup g = rt.RowGroups[1, 3];
g.CanSplit = true;
g.DataBinding.DataSource = MyData.Generate(20, 0);
g.DataBinding.Grouping.Expressions.Add("Fields!GroupId.Value");
g.Style.BackColor = Color.LightCyan;
<span style="color: #008000">// 创建一个嵌套分组，按照SubGroupId字段进行分组:</span>
TableVectorGroup ng = rt.RowGroups[2, 2]; ng.CanSplit = true; ng.DataBinding.DataSource = g.DataBinding.DataSource;
ng.DataBinding.Grouping.Expressions.Add("Fields!SubGroupId.Value"); ng.Style.BackColor = Color.LightPink;
<span style="color: #008000">// 创建更深一层的嵌套数据绑定分组:</span>
TableVectorGroup ng2 = rt.RowGroups[3, 1]; ng2.DataBinding.DataSource = g.DataBinding.DataSource; ng2.Style.BackColor =
Color.LightSteelBlue;
```

以上代码可以由以下表格演示：

			Header		
Group 1, 3			GroupId = [Fields!GroupId.Value]		
	Group 2, 2		GroupId = [Fields!GroupId.Value] SubGroupId = [Fields!SubGroupId.Value]		
		Group 3, 1	GroupId= [Fields!GroupId.Value]	SubGroupId= [Fields!SubGroupId.Value]	IntValue= [Fields!IntValue.Value]

数据绑定示例

数据绑定示例，位于HelpCentral，包含几个数据绑定文档的示例。以下主题将讨论这些示例其中的一些问题。

使用分组

一个典型的应用分组的示例由以下代码进行演示：

Visual Basic

```
Visual Basic

' 创建一个按照分组重复的RenderArea.
Dim ra As C1.C1Preview.RenderArea = New C1.C1Preview.RenderArea ra.Style.Borders.All = New C1.C1Preview.LineDef("2mm",
Color.Blue)

' MyData 对象数组用作数据源:
ra.DataBinding.DataSource = MyData.Generate(100, 0)

' 数据按照GroupId字段进行分组:
```

```
ra.DataBinding.Grouping.Expressions.Add("Fields!GroupId.Value")

' 创建一个做为分组页眉的RenderText；通常情况下，页眉可以非常复杂，其本身可以是数据绑定的:
Dim rt As C1.C1Preview.RenderText = New C1.C1Preview.RenderText

' 分组页眉看起来像是"GroupId=XXX"的格式，这里XXX表示分组中GroupId字段的值:
rt.Text = "GroupId: [Fields!GroupId.Value]" rt.Style.BackColor = Color.Yellow

' 为分组区域添加标题:
ra.Children.Add(rt)

'该 RenderText 将会在每一个分组打印一条记录:
rt = New C1.C1Preview.RenderText

' 为每一条记录打印的文本:
rt.Text = "GroupId: [Fields!GroupId.Value]" & Microsoft.VisualBasic.Chr(13) &

"IntValue: [Fields!IntValue.Value]" rt.Style.Borders.Bottom = C1.C1Preview.LineDef.Default rt.Style.BackColor =
Color.FromArgb(200, 210, 220)
'设置该文本的数据源为其所包含的RenderArea的数据源，这表示该render对象绑定到当前分组的特定对象上: rt.DataBinding.DataSource
= ra.DataBinding.DataSource

' 向区域添加文本:
ra.Children.Add(rt)
```

C#

```
C#
```

```
// 创建一个按照分组重复的RenderArea.
RenderArea ra = new RenderArea(); ra.Style.Borders.All = new LineDef("2mm", Color.Blue);

// MyData 对象数组用作数据源:
ra.DataBinding.DataSource = MyData.Generate(100, 0);

// 数据按照GroupId字段进行分组:
ra.DataBinding.Grouping.Expressions.Add("Fields!GroupId.Value");

// 创建一个做为分组页眉的RenderText；通常情况下，页眉可以非常复杂，其本身可以是数据绑定的:
RenderText rt = new RenderText();

// 分组页眉看起来像是"GroupId=XXX"的格式，这里XXX表示分组中GroupId字段的值:
rt.Text = "GroupId: [Fields!GroupId.Value]"; rt.Style.BackColor = Color.Yellow;
// 为分组区域添加标题:
ra.Children.Add(rt);

//该 RenderText 将会在每一个分组打印一条记录:
```

```
rt = new RenderText();

// 为每一条记录打印的文本:
rt.Text = "GroupId: [Fields!GroupId.Value]\rIntValue: [Fields!IntValue.Value]"; rt.Style.Borders.Bottom = LineDef.Default;
rt.Style.BackColor = Color.FromArgb(200, 210, 220);

// 设置该文本的数据源为其所包含的RenderArea的数据源，这表示该render对象绑定到当前分组的特定对象上: rt.DataBinding.DataSour
ce = ra.DataBinding.DataSource;

// 向区域添加文本:
ra.Children.Add(rt);
```

使用汇总功能

以下代码扩展了之前的示例，介绍了为文档和分组使用汇总功能，该示例将和之前的代码做为一个整体：
Visual Basic

```
Visual Basic
```

```
' 创建一个对每一个分组重复的Render区域:
Dim ra As C1.C1Preview.RenderArea = New C1.C1Preview.RenderArea ra.Style.Borders.All = New C1.C1Preview.LineDef("2mm",
Color.Blue) ra.DataBinding.DataSource = MyData.Generate(20, 0, True)
ra.DataBinding.Grouping.Expressions.Add("Fields!GroupId.Value")

' 创建一个用于计算每一个分组中IntValue字段之和的汇总calc the sum of IntValue fields within each group:
Dim agg As C1.C1Preview.DataBinding.Aggregate = New
C1.C1Preview.DataBinding.Aggregate("Group_IntValue")

' 定义计算求和的表达式:
agg.ExpressionText = "Fields!IntValue.Value"

' 指定该汇总操作应当针对当前分组范围:
agg.Running = C1.C1Preview.DataBinding.RunningEnum.Group

' 指定汇总操作的数据源:
agg.DataBinding = ra.DataBinding

''添加汇总至文档:
doc.DataSchema.Aggregates.Add(agg)

' 创建一个用于计算每一个分组中IntValue字段之和的汇总calc the sum of IntValue fields over the whole document:
agg = New C1.C1Preview.DataBinding.Aggregate("Total_IntValue")

' 定义用作计算求和的表达式:
```

```
agg.ExpressionText = "Fields!IntValue.Value" ' Specify that aggregate should have document scope: agg.Running =
C1.C1Preview.DataBinding.RunningEnum.All

' 指定汇总操作的数据源:
agg.DataBinding = ra.DataBinding

''添加汇总至文档:
doc.DataSchema.Aggregates.Add(agg)

' 添加分组页眉:
Dim rt As C1.C1Preview.RenderText = New C1.C1Preview.RenderText rt.Text = "GroupId: [Fields!GroupId.Value]"
rt.Style.BackColor = Color.Yellow ra.Children.Add(rt)

' 该呈现文本将打印分组记录；和可以看到的那样，分组汇总的值不仅仅可以在分组页脚中引用，也可以在分组页眉和分组内容区域引用
rt = New C1.C1Preview.RenderText rt.Text = "GroupId:
[Fields!GroupId.Value]\rIntValue:[Fields!IntValue.Value]\rGroup_IntValue:
[Aggregates!Group_IntValue.Value]\rTotal_IntValue:

[Aggregates!Total_IntValue.Value]\rTatalNested_IntValue: [Aggregates!TatalNested_IntValue.Value]"

rt.Style.Borders.Bottom = C1.C1Preview.LineDef.Default rt.Style.BackColor = Color.FromArgb(200, 210, 220)
rt.DataBinding.DataSource = ra.DataBinding.DataSource ra.Children.Add(rt)

' 该汇总也在分组内进行计算，但是关联到内嵌对象的数据绑定:
agg = New C1.C1Preview.DataBinding.Aggregate("TotalNested_IntValue") agg.ExpressionText = "Fields!IntValue.Value"
agg.Running = RunningEnum.All agg.DataBinding = rt.DataBinding doc.DataSchema.Aggregates.Add(agg)

' 添加该区域至文档:
doc.Body.Children.Add(ra)
```

C#

```
C#
```

```
// 创建一个对每一个分组重复的Render区域:
RenderArea ra = new RenderArea(); ra.Style.Borders.All = new LineDef("2mm", Color.Blue); ra.DataBinding.DataSource =
MyData.Generate(20, 0, true); ra.DataBinding.Grouping.Expressions.Add("Fields!GroupId.Value");

// 创建一个用于计算每一个分组中IntValue字段之和的汇总calc the sum of IntValue fields within
```


```
<span style="color: #008000">each group:</span> Aggregate agg = new Aggregate("Group_IntValue");
<span style="color: #008000">// 定义计算求和的表达式:</span>
agg.ExpressionText = "Fields!IntValue.Value";
<span style="color: #008000">// 指定该汇总操作应当针对当前分组范围:</span>
agg.Running = RunningEnum.Group;
<span style="color: #008000">// 指定汇总操作的数据源:</span>
agg.DataBinding = ra.DataBinding;
<span style="color: #008000">//添加汇总至文档:</span>
```

doc.DataSchema.Aggregates.Add(agg);
// 创建一个用于计算每一个分组中IntValue字段之和的汇总:
agg = new Aggregate("Total_IntValue");
// 定义用作计算求和的表达式:
agg.ExpressionText = "Fields!IntValue.Value"; // Specify that aggregate should have document scope: agg.Running = RunningEnum.All;
// 指定汇总操作的数据源:
agg.DataBinding = ra.DataBinding;
//添加汇总至文档:
doc.DataSchema.Aggregates.Add(agg);
// 添加分组页眉:
RenderText rt = new RenderText(); rt.Text = "GroupId: [Fields!GroupId.Value]"; rt.Style.BackColor = Color.Yellow; ra.Children.Add(rt);
//
该呈现文本将打印分组记录；和可以看到的那样，分组汇总的值不仅仅可以在分组页脚中引用，也可以在分组页眉和分组内容区域引用:
rt = new RenderText(); rt.Text = "GroupId: [Fields!GroupId.Value]\rIntValue:[Fields!IntValue.Value]\rGroup_IntValue: [Aggregates!Group_IntValue.Value]\rTotal_IntValue: [Aggregates!Total_IntValue.Value]\rTatalNested_IntValue: [Aggregates!TatalNested_IntValue.Value]";
rt.Style.Borders.Bottom = LineDef.Default; rt.Style.BackColor = Color.FromArgb(200, 210, 220); rt.DataBinding.DataSource = ra.DataBinding.DataSource; ra.Children.Add(rt);
// 该汇总也在分组内进行计算，但是关联到内嵌对象的数据绑定:
agg = new Aggregate("TotalNested_IntValue"); agg.ExpressionText = "Fields!IntValue.Value"; agg.Running = RunningEnum.All; agg.DataBinding = rt.DataBinding; doc.DataSchema.Aggregates.Add(agg);
// 添加该区域至文档:
doc.Body.Children.Add(ra);
注意：也有可以在数据绑定的C1PrintDocuments使用的汇总类型，他们不需要声明在文档的汇总集合（Aggregates）。更多的细节和例子，请参见数据汇总主题。.
数据汇总
Unable to render embedded object: File (MISSING PHRASE 'Show All') not found. Unable to render embedded object: File (MISSING PHRASE 'Hide All') not found.
在发布的2010 V1版本中，向Reports for WinForms添加了新的汇总。这些汇总类型可以应用在数据绑定的C1PrintDocument中，而不需要在文档的汇总集合（Aggregates）中声明它们。例如，如果"Balance"是一个数据绑定文档的数据字段，以下RenderText可以用于打印数据集的余额总量：
Visual Basic

Visual Basic
Dim rt As New RenderText("[Sum(""Fields!Balance.Value"")]")

C#

C#
RenderText rt = new RenderText("[Sum(\"Fields!Balance.Value\")]");

以下新的属性和方法被添加到DataSet和C1DataBinding类型上以支持此功能：

类	成员	描述
C1DataBinding	Name 属性	获取或设置当前C1DataBinding名称。该名称可以在汇总函数中使用，表示该汇总表示的目标数据绑定
DataSet	Name 属性	获取或设置当前数据集的名称。该名称可以在汇总函数中使用，表示该汇总表示的目标数据源。

所有汇总函数具有以下格式：
AggFunc(expression, scope)
参数解释：
expression是一个定义用做计算每一个行分组或者数据集行的表达式。 scope是一个用来标识汇总计算所包括的数据集范围的字符串。如果省略，则汇总会基于当前的数据集进行计算（比如当前的分组，数据集等等）。如果指定的话，应当是目标分组或者数据集的名称。
例如，如果一个数据集有以下字段，ID，GroupID，SubGroupID，NAME，Q，同时记录按照GroupID和SubGroupID进行分组，则可以创建以下文档：
Visual Basic

Visual Basic

```
Dim doc As New C1PrintDocument()
Dim raGroupId As New RenderArea() ' 按照需求设置raGroupId的属性...
raGroupID.DataBinding.DataSource = dataSet raGroupID.DataBinding.Name = "GroupID"
raGroupID.DataBinding.Grouping.Expressions.Add("Fields!GroupID.Value")
Dim raSubGroupID As New RenderArea() ' 按照需求设置raSubGroupID的属性...
raSubGroupID.DataBinding.DataSource = dataSet raSubGroupID.DataBinding.Grouping.Expressions.Add("Fields!SubGroupID.Value")
raGroupID.Children.Add(raSubGroupID)
Dim raDetail As New RenderArea() ' 按照需求设置raDetail属性...
raDetail.DataBinding.DataSource = dataSet raSubGroupID.Children.Add(raDetail)

' 显示 Q 字段的值:
Dim rtQ As New RenderText() rtQ.Text = "[Fields!Q.Value]" raDetail.Children.Add(rtQ)

' 显示内嵌分组 Q 字段的值之和（SubGroupID）
Dim rtSumQ1 As New RenderText() rtSumQ1.Text = "[Sum(""Fields!Q.Value"")]" raDetail.Children.Add(rtSumQ1)

' 显示GroupID的所有Q字段的值之和:
Dim rtSumQ2 As New RenderText() rtSumQ2.Text = "[Sum(\"Fields!Q.Value\", \"GroupID\"")]" raDetail.Children.Add(rtSumQ2)

' 显示整个数据集Q字段的值之和:
Dim rtSumQ3 As New RenderText() rtSumQ3.Text = "[Sum(\"Fields!Q.Value\", \"DataSet\"")]" raDetail.Children.Add(rtSumQ3)
doc.Body.Children.Add(raGroupId)
```

C#

C#

```
C1PrintDocument doc = new C1PrintDocument();
RenderArea raGroupId = new RenderArea(); // 按照需求设置raGroupId的属性...
raGroupID.DataBinding.DataSource = dataSet; raGroupID.DataBinding.Name = "GroupID";
raGroupID.DataBinding.Grouping.Expressions.Add("Fields!GroupID.Value");
RenderArea raSubGroupID = new RenderArea();
```

```
// 按照需求设置raSubGroupID的属性...
raSubGroupID.DataBinding.DataSource = dataSet;
raSubGroupID.DataBinding.Grouping.Expressions.Add("Fields!SubGroupID.Value"); raGroupID.Children.Add(raSubGroupID);
RenderArea raDetail = new RenderArea(); // 按照需求设置raDetail属性...
raDetail.DataBinding.DataSource = dataSet; raSubGroupID.Children.Add(raDetail);

// 显示 Q 字段的值:
RenderText rtQ = new RenderText(); rtQ.Text = "[Fields!Q.Value]"; raDetail.Children.Add(rtQ);

// 显示内嵌分组 Q 字段的值之和（SubGroupID）
RenderText rtSumQ1 = new RenderText(); rtSumQ1.Text = "[Sum(\"Fields!Q.Value\")]"; raDetail.Children.Add(rtSumQ1);

// 显示GroupID的所有Q字段的值之和:
RenderText rtSumQ2 = new RenderText(); rtSumQ2.Text = "[Sum(\"Fields!Q.Value\", \"GroupID\"")]";
raDetail.Children.Add(rtSumQ2);

// 显示整个数据集Q字段的值之和:
RenderText rtSumQ3 = new RenderText(); rtSumQ3.Text = "[Sum(\"Fields!Q.Value\", \"DataSet\"")]";
raDetail.Children.Add(rtSumQ3); doc.Body.Children.Add(raGroupId);
```

当以上文档生成时，每一个raDetail分组的实例将显示以下四个值：
"Q"字段的当前值 位于当前SubGroupID中间的"Q"字段的值之和 位于当前GroupID中的"Q"字段的值之和 整个文档的"Q"值之和目录
C1PrintDocument支持自动生成目录（TOC）。目录本身由专门的render对象呈现，类型为RenderToc，该类型派生自RenderArea 类型并增加了TOC相关的功能。TOC中间每一个单独的项目由 RenderTocItem表示（继承
自RenderParagraph）。每一个TOC项目有一个超链接（ RenderTocItem.Hyperlink 属性），指向文档中的一个位置
（由锚点表示）。因此，用于连接TOC项目和文档内容的机制和超链接一致。提供了方便的创建TOC的方法，详见下文。
为了向文档添加一个目录，请执行以下步骤：

1. 创建一个RenderToc类型的实例，并将其添加到文档中您希望出现TOC的位置。
2. 添加单个项目（RenderTocItem类型）至RenderToc实例。下列方法（或它们的组合）可以用于此操作：

您可以使用RenderToc.AddItem方法的任意一个重载向TOC添加TOC项目，向项目提供其显示的文本，所指向的文档中的位置，以及可选地，TOC的缩进等级。
您可以在代码中创建RenderTocItem类型的实例，设置其属性并将其添加至TOC render对象的Children集合。
RenderTocItem提供的几个不同的重载的其中一个，接受一个RenderToc的实例做为参数。当使用该构造函数时，新创建的TOC项目会添加至该指定的TOC实例，因此您将不必手动将其添加至TOC对象。
关于如何使用特定的RenderToc render对象为文档创建一个目录的完整实例，请参见安装在ComponentOne Samples文件夹下的RenderTOC示例。
单词索引
您现在可以通过C1PrintDocument自动生成索引。每一个索引（一个文档可以具有多个索引）由一个条目首字母按字母序排序的列表，跟随一个该条目出现的页码组成。
索引由一个RenderIndex类的实例表示，该类是一个派生自RenderArea的render对象。可以将该对象和其他render对象一样添加到文档中，但是有一个重要的限制：索引必须出现在其包含的所有条目（项）之后；因此，和传统的索引一样，这里的索引最好也出现在文档末尾。
词汇是单词以及单词的组合，在文档中应当作为索引中条目出现。当文档创建之后，这些词汇将被添加到RenderIndex
对象上，同时带有关于它们在文档中出现位置的相关信息（通常RenderText或者RenderParagraph对象包含这些项）。

之后，在文档生成时，RenderIndex对象将产生实际的索引。
支持Index功能的类
以下专门的类型支持索引：
RenderIndex:该类型派生自RenderArea，并在插入到C1PrintDocument并生成文档时产生索引。
RenderIndex必须出现在所有的索引条目出现的位置之后。存在该限制的原因是，该索引的实际内容（从而，索引实际所要占据的空间大小）会根据项目出现的多少有很大的不同。
IndexEntry:该类型用作描述索引中的一个索引条目（词汇）。
每一个条目可以出现多次（该条目在文档中描述或者出现的位置）。一个条目出现的不同位置的集合由IndexEntry上的Occurrences属性描述。
在文档生成时，索引中每一个条目的出现位置将生成一个带有超链接的页码。除此之外，每一个条目也可以包含一个子条目的列表（由Children属性暴露）。内嵌层次是没有限制的，不过通常使用最多三层嵌套。
最后，为了允许将一个条目连接到索引中的其他条目，条目上的SeeAlso属性可以用来包含一个索引项的列表，用来在生成的索引中将当前索引链接到其他索引上。
IndexEntryOccurrence:该类型描述了文档中一个条目单次出现的位置。
该类型的元素将包含在IndexEntry的Occurrences集合中。
当创建一个索引项时，可以指定一个或者多个occurrence（做为传递给构造器的参数），并且可以在稍后向该索引项添加更多的occurrence对象。
该类型最主要的功能性的属性是Target，该属性的类型是C1LinkTarget，指向出现位置。
用代码生成一个索引
典型地，以下步骤将通过代码向一个文档添加一个简单的一级嵌套的索引：

1. 创建一个RenderIndex类型的实例并保存在一个本地变量（和之前提到的一样，索引不能放置在其包含的条目之前）。
2. 当内容（render对象）添加至文档时，应当有一些代码逻辑标识哪些字符串将变成索引的条目（项）。每一个这样的字符串应当被测试，检查是否已经被加入到步骤一中间创建的索引对象的Entries集合。如果这是一个新的条目，则应当创建一个新的IndexEntry对象，并添加至索引。
3. 为了指定一个条目在文档中出现的位置，应当为现有的或者新创建的条目添加一个条目出现位置的描述

（IndexEntryOccurrence对象）。通常该位置由包含该条目，并且已经添加到文档的RenderObject唯一标识。

1. 当该条目全部出现的位置已经添加到文档之后，在步骤一中间创建的RenderIndex对象可以被添加到文档。
2. 在文档生成之后，RenderIndex对象将生成已添加条目的超链接索引。这些条目将自动排序，按照每一个条目的首字母进行分组，并在每一个分组之前添加该首字母做为分组标签。

当然这仅仅是一个简单的可能的应用场景，用来演示在创建索引时包含的主要对象之间的关系。其他一些可能的用法，包括在创建文档之前（基于一个外部的项目字典）创建索引项（索引条目），添加内嵌的条目（子条目）等等。
自定义索引的外观

自定义索引的外观

以下属性用来自定义生成的索引的外观：
Styles (参见 Styles):
Style:指定整个索引的样式（包括标题，条目，等等）。
<span style="color:
#1364c4">HeadingStyle:指定用作字母标题的样式（标题指的是一组条目，首字母以该字母开头）。在生成的索引中，每一个标题（通常是位于某个以该字母开头的分组之前）由一个独立的render对象（RenderText）表示，该对象将应用此样式。
<span style="color:
#1364c4">EntryStyles:一个具有索引下标表示的属性，指定不同级别条目的样式。例如，EntryStyles[0]（VB中为EntryStyles(0)）指定最顶级条目的样式，EntryStyles[1]（VB中为EntryStyles(1)）指定子条目的样式，以此类推。（如果索引中的内嵌级别高于EntryStyles集合中项目的个数，则对于多出的内嵌级别，将使用集合中的最后一个样式。）
在生成的索引中，每一个条目（一个紧随着其出现位置页码的项）由一个独立的 <span style="color:
#1364c4">RenderParagraph对象表示，在该对象中应用由该属性指定的对应内嵌级别的样式。例如，该样式允许指定一个条目文本在允许插入一个换页之前，所必须出现的最小行数（通过MinOrphanLines）。
EntryStyle:这是指向EntryStyles集合中的第一个元素（索引值为0）的快捷方式。
SeeAlsoStyle:允许指定用作在条目间互相引用的"see also"的文本样式（参见SeeAlso）。
Style:允许为一个特定的条目单独覆盖其样式。
SeeAlsoStyle:允许为某一个特定的条目覆盖其"see also"文本的样式。
其他属性：其他属性：
<span style="color:
#1364c4">RunIn:一个布尔型值（默认值为False），如果设置为True，表示子条目应当和主标题显示在同一行，而不是缩进显示为单独的行。
EntryIndent:一个带单位的属性，指定子条目相对于主条目的缩进值。默认值为0.25英寸。
<span style="color:
#1364c4">EntryHangingIndent:一个带单位的属性，指定一个条目的首行相对于其他行的缩进值（相对于左侧），通常应用于显示为多行的情况。默认值为-0.125英寸。
<span style="color:
#1364c4">LetterSplitBehavior:一个SplitBehaviorEnum类型的属性，确定一个字母分组（以相同首字母开始的条目）如何在垂直方向上分开。默认值为SplitBehaviorEnum.SplitIfNeeded。注意标题（默认显示该分组的首字母）始终和第一个条目一起显示。
Italic:和Bold类似，不过使用斜体显示而不是粗体。
LetterFormat:用作格式化字母标题的字符串。默认值为"{0}"。
<span style="color:
#1364c4">TermDelimiter:用来分隔条目项和该项出现位置列表（页码）的字符串。默认值为一个逗号跟上一个空格。
<span style="color:
#1364c4">RunInDelimiter:用作当一个run-in类型（参见RunIn）的索引生成时，用作分隔不同条目的字符串。默认值为分号。
<span style="color:

#1364c4">OccurrenceDelimiter:一个用作分隔每一个条目出现位置列表（页码）的字符串，默认值为一个逗号带着一个空格。
PageRangeFormat:一个用作格式化条目出现位置页码范围的字符串，默认值为"{0}-{1}"。
SeeAlsoFormat:用作格式化"see
also"引用的字符串。默认值为"(see{0})"（一个空格，接下来是左括号，在紧接着是输出格式化项目，最后是右括号）。
<span style="color:
#1364c4">FillChar:当页码向右对齐时，用作填充的字符（PageNumbersAtRight属性设置为True）。该属性的默认值为一个点字符。
PageNumbersAtRight:一个布尔类型的属性，决定是否向右对齐页码。默认值为False。
<span style="color:
#1364c4">EntrySplitBehavior:一个SplitBehaviorEnum类型的属性，确定如何在垂直方向分隔一个单独的条目。默认值为
SplitBehaviorEnum.SplitIfLarge。该属性应用到全部级别的条目。
<span style="color:
#1364c4">Bold:一个布尔型的值，允许通过粗体高亮显示某个条目的某个特定的出现位置。（例如，这可以用做高亮显示某个条目出现的主要定义的位置。）索引样式层次
The hierarchy of index-specific styles is as follows:
Style, 用作RenderIndex对象，做为其他索引特定样式的AmbientParent
HeadingStyle
EntryStyles
Style
SeeAlsoStyle
SeeAlsoStyle
除了RenderIndex的Style之外，以上所列出的全部样式做为相关对象内联样式的Parent以及AmbientParent。因此比如说，设置RenderIndex的Style上的字体将影响该索引中全部元素（除了被低层次样式覆盖的以外），指定该样式上的边框将在整个索引绘制一个边框，但是每一个单独的元素则不受影响。与此同时，指定SeeAlsoStyle的边框将在每一个索引中的"see also"元素绘制一个边框。
生成索引的结构
下图显示在生成文档时由一个RenderIndex对象创建的render对象的树形结构和层次关系（这里，仅仅最顶层的
RenderIndex对象由用户代码创建；其他对象为自动生成）：
RenderIndex
RenderArea (表示一个具有相同首字母的项目分组)
RenderText (prints letter group header)
RenderParagraph (prints top-level IndexEntry)
RenderParagraph (prints sub-entry, offset via Left)
...
RenderArea (represents a group of entries starting with the same letter)
RenderText (prints letter group header)
RenderParagraph (prints top-level IndexEntry)
RenderParagraph (prints sub-entry, offset via Left)
... 大纲视图
C1PrintDocument支持大纲视图。大纲视图是一个具有指向文档中位置的节点（OutlineNode类型）的树形结构（由
Outlines属性指定）。大纲视图在预览的导航面板的一个标签页中显示，允许通过单击其中的某个项目导航到关联到该项目的相关位置。同时，大纲视图可以导出到支持该标记的文件格式（比如说PDF）。
使用Outline的任意一个构造器重载创建一个大纲节点。可以指定该大纲节点的文本，在文本中指向的位置（一个render
对象或者一个锚点），以及在预览时大纲树形视图面板上显示的图标。顶级的节点应当添加到文档的Outlines集合。每一个大纲节点可以按照顺序在Children集合中按照顺序包含子节点集合，层层嵌套。

提示：每一个大纲视图节点项目可以被单击。单击一个节点将会显示关联到该节点的项目。关于如何添加大纲视图节点的示例，请参见Adding Outline Entries to the Outline Tab.

内嵌字体
当 C1PrintDocument文档保存为C1DX或者C1D格式的文件时，文档中使用到的字体将会内嵌到该文件中。这样的话，
在一个不同的系统中打开这个文档时，即便是当前系统不具备所有的安装字体，也将使用内嵌的字体进行确保文本能够正确呈现。字体内嵌技术在使用了罕见字体或者专用字体时尤其有用（比如用作绘制条码的字体）。注意，当一个字体内嵌到C1PrintDocument文档时，并不意味着该字体中间的全部字形均被嵌入。只是在该字体中实际使用到的字形被嵌入到文档中。
以下C1PrintDocument属性和字体嵌入嵌入功能相关：
EmbeddedFonts-这是包含文档中全部内嵌字体的集合。注意除了可以自动产生该集合，也可以为自定义控件手动改变内嵌字体（更多信息，请参见以下章节）。
DocumentFonts-该集合为自动生成，取决于C1PrintDocument.FontHandling属性的值。可以被用作查找文档中使用了哪些字体。
FontHandling -该属性确定是否以及如何自动生成两个相关的集合（EmbeddedFonts 以及DocumentFonts）。
字体替代
当使用某种字体呈现一段文本，而表示该文本的字形在指定的字体中间不存在，则会选择使用替代字体呈现该字形。例如，如果使用Arial字体呈现日文的象形文字，则会实际使用ArialUnicodeMS字体呈现文本。C1PrintDocument可以分析这种情况并添加实际使用的字体（而不是指定的那些字体）至DocumentFonts和/或EmbeddedFonts集合。为了达到这一效果，FontHandling属性必须设置为FontHandling.BuildActualDocumentFonts或者
FontHandling.EmbedActualFonts。这些设置的缺点在于这一过程需要消耗额外的时间，使得文档生成速度变慢。因此，只有在已知文档包含字体不包含的字形时才指定这一设置（比如说使用通用拉丁文字体显示东亚语言文字）。
当分析字体替代时，将搜索以下预定义的字体集合以查找包含缺失字形的最佳匹配字体：
MS UI Gothic
MS Mincho
Arial Unicode MS
Batang
Gulim
Microsoft YaHei
Microsoft JhengHei
MingLiU
SimHei
SimSun

有选择性的字体嵌入

如果您的文档同时使用通用字体，比如说Arial，和某种专用字体，您可能希望仅仅将该专用字体（或者几种专用字体）嵌入到文档中，而不是全部用到的字体。按照以下步骤做到：

1. 设置FontHandling属性的值为除了FontHandling.EmbedFonts或者FontHandling.EmbedActualFonts以外的值。这将使得在生成文档时，EmbeddedFonts集合保持为空；
2. 通过代码手动添加该专用字体至文档的EmbeddedFonts集合（这些字体必须确保已经安装在当前系统上）。将.NET Font对象传递给EmbeddedFont的构造器以生成一个EmbeddedFont对象。通过EmbeddedFont.AddGlyphs方法（该方法提供了若干重载）添加所需要的字形至该内嵌字体。

当一个通过以上方式创建的具有EmbeddedFonts集合的C1PrintDocument保存时（作为一个C1DX或者C1D格式的文件），只有集合中指定的字体被内嵌到文档中。

字典

!MISSING PHRASE 'Show All'!
!MISSING PHRASE 'Hide All'!
如果一个项目（比方说一个图片或者图标）在文档的多个地方被使用时，您可以将其保存在一个全文档可用的位置，并在需要用到的地方引用这个实例，而不是在所需要用到它们的地方插入重复的图片或图标。为了做到这一点，C1PrintDocument提供了一个字典。
通过Dictinary属性访问该字典。该字典包含DictionaryItem基类型的项目，这是一个抽象类型。提供了两个派生类型，DictionaryImage以及DictionaryIcon，相应地用来存储图片和图标。字典中的项目按照名称进行引用。为了能够使用一个项目，您必须为其指定一个名字。在字典内部，该名称必须唯一。
可以在文档的以下位置使用字典项目：
通过BackgroundImageName属性，做为样式的背景图。
通过ImageName属性，做为RenderImage对象包含的图像。
因此，如果如果您的文档在多处使用同一个文档，请按照以下步骤处理：
向字典添加一个图片，比如说像下面这样：
Visual Basic

```
Visual Basic
Me.C1PrintDocument1.Dictionary.Add(New C1.C1Preview.DictionaryImage("image1",
Image.FromFile("myImage.jpg")))
```

C#

```
C#
this.c1PrintDocument1.Dictionary.Add(new DictionaryImage("image1",
Image.FromFile("myImage.jpg")));
```

• 通过设置样式上的BackgroundImageName属性或者RenderImage对象的ImageName属性，使用该图片，比如说像下面这样：
Visual Basic

```
Visual Basic
Dim ri As New C1.C1Preview.RenderImage() ri.ImageName = "image1"
```

C#

```
C#
RenderImage ri = new RenderImage(); ri.ImageName = "image1";
```