CodeSmith 开发资料

整理制作：Terrylee

2005 年 12 月

目　　录

CodeSmith 基础（一）

创建好一个模板后第一步要指明这是一个 **C#**语言的模板。

```
<%@ CodeTemplate Language="C#" TargetLanguage="C#"
    Description="Generates a class including a special informational header" %
>
```

第二步，我们要指明模板在生成代码时的属性，即生成代码需要的输入值变量。

```
<%@ Property Name="NameSpace" Type="String"
    Category="Context"
    Description="The namespace to use for this class" %>
```

如上边所示，在进行代码生成时，在 **CodeSmith Explorer** 中选择模板后生成代码的窗口中，变量的名称为 **NameSpace**，类型是 **String**，类别是 **Context**，当用户选中这个属性时对于属性的描述 **Description**。

我们可以按照 **C#**语言的语法去使用定义的变量，例如：

```
////////////////////////////////////////////////////////////////////////////////
// File: <%=ClassName%>.cs
```

例如下面这个例子模板使用了上面介绍的知识。**Test.cst**

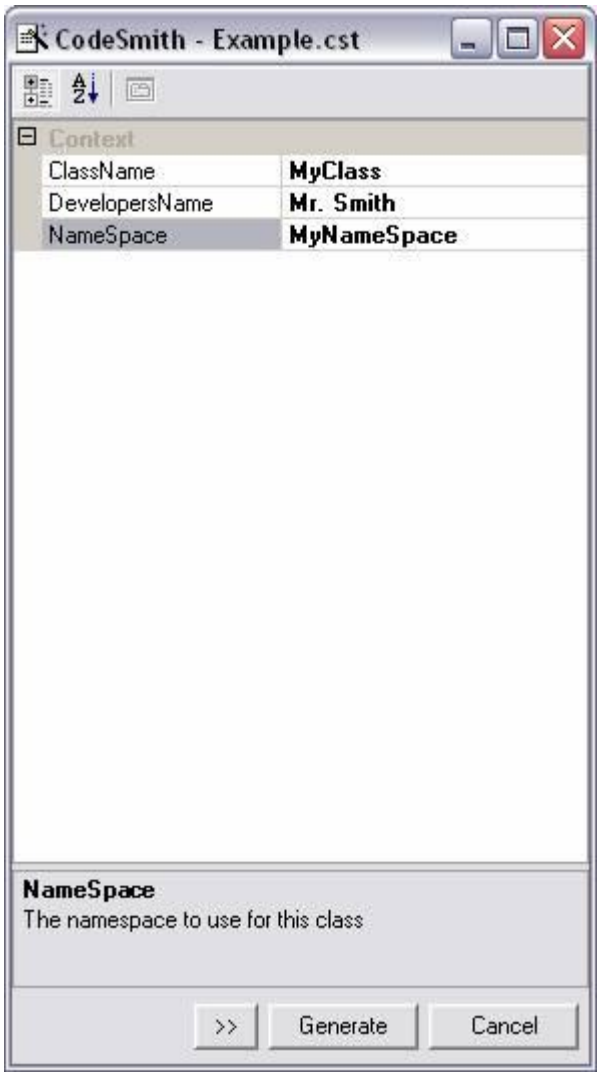
```
<%@ CodeTemplate Language="C#" TargetLanguage="C#"
    Description="Generates a class including a special informational header" %>
```

```
<%@ Property Name="NameSpace" Type="String"
    Category="Context"
    Description="The namespace to use for this class" %>

<%@ Property Name="ClassName" Type="String"
    Category="Context"
    Description="The name of the class to generate" %>

<%@ Property Name="DevelopersName" Type="String"
    Category="Context"
    Description="The name to include in the comment header" %>
//////////////////////////////////////////////////////////////////////////////////////////
// File: <%=ClassName%>.cs
// Description: Enter summary here after generation.
// ---------------------
// Copyright © <%= DateTime.Now.Year %> Our Client
// ---------------------
// History
//    <%= DateTime.Now.ToShortDateString() %>    <%= DevelopersName%>
  Original Version
//////////////////////////////////////////////////////////////////////////////////////////
```

```
using System;



namespace <%=NameSpace %>

{

    /// <summary>

    /// Summary description for <%=ClassName %>.

    /// </summary>

    public class <%=ClassName %>

    {

        public <%=ClassName %>()

        {

            //

            // TODO: Add constructor logic here

            //

        }

    }

}
```

然后我们在 **CodeSmith Explorer** 中双击这个模板就会看到相应的属性界面，这里的属性均是

我们在前边定义的属性。



按下 **Generate** 按钮生成，即可实现一个简单的类代码的生成。

```
1
//////////////////////////////////////////////////////////////////////////////////////////
2  // File: MyClass.cs
3  // Description: Enter summary here after generation.
4  // --------------------
5  // Copyright © 2003 Our Client
```

```
6  // --------------------
7  // History
8  //    12/2/2003    Mr. Smith    Original Version
9
//////////////////////////////////////////////////////////////////////////////////////////
10
11  using System;
12
13  namespace MyNameSpace
14  {
15      /// <summary>
16      /// Summary description for MyClass.
17      /// </summary>
18      public class MyClass
19      {
20          public MyClass()
21          {
22              //
23              // TODO: Add constructor logic here
24              //
25          }
```

```
26     }
27 }
```

生成后的代码即可放入 **Visual Studio .NET** 中使用，我们使用 **CodeSmith** 的目的就是为了快速高效的开发。

CodeSmith 基础（二）

本文将介绍 **CodeSmith** 与数据库进行交互生成相应的存储过程，本例使用的数据库为 **SQL Server 2000**。

在与数据库进行交互时，我们使用到了一个 **CodeSmith** 自带的组件 **SchemaExplorer**，利用这个组件我们可以访问数据库的数据表、存储过程、视图等，并可以得到相应的数据类型、标识列、列的（字段）名称等信息。

下面这个例子是教我们如何生成一个存储过程，虽然网上有很多一样的例子，但是我是从 **CodeSmith** 中的英文帮助中自己翻译出来的：）

使用的是 **SQL Server 2000** 自带的 **Northwind** 数据库，生成一个关于 **Orders** 订单表的更新存储过程。

第一步还是指明模板使用的语言和生成的目标语言。

```
<%@ CodeTemplate Language="C#" TargetLanguage="T-SQL" Description="Generates a update stored procedure." %>
```

第二步就是我们要加载使用访问数据库的组件 **SchemaExplorer**，并声明其使用的命名空间。

```
<%@ Assembly Name="SchemaExplorer" %>
<%@ Import Namespace="SchemaExplorer" %>
```

因为是针对表去生成存储过程，则首先要定义一个存储表名称使用的变量，然后指明这个变量类型为数据库中的表，这样我们可以通过这个数据表类型的变量得到相应的表的信息。

```
<%@ Property Name="SourceTable" Type="SchemaExplorer.TableSchema" Category="Context"  Description="Table that the stored procedures should be based on." %>
```

如果想访问视图的话，则将变量类型 **Type** 中的 **SchemaExplorer.TableSchema** 修改为 **SchemaExplorer.ViewSchema** 即可。

得到表名的方法

```
CREATE PROCEDURE dbo.Update<%= SourceTable.Name %>
```

下面利用循环语句遍历表的各个列，拼出存储过程需要传递的参数。

```
<% for (int i = 0; i < SourceTable.Columns.Count; i++) { %>
<%= GetSqlParameterStatement(SourceTable.Columns[i]) %><% if (i < SourceTable.Columns.Count - 1) { %>,<% } %>
<% } %>
```

调用的 **GetSqlParameterStatement** 方法是用来生成参数的字符串，例如生成"**@CustomerID nchar(5)**"，后边紧跟的 **if** 判断是用来生成参数之间相隔使用的逗号的。

生成参数字符串的方法，参数为 **SchemaExplorer.ColumnSchema** 列类型

```
<script runat="template">
public string GetSqlParameterStatement(ColumnSchema column)
{
    string param = "@" + column.Name + " " + column.NativeType;

    switch (column.DataType)
    {
        case DbType.Decimal:
        {
            param += "(" + column.Precision + ", " + column.Scale + ")";
            break;
        }
        default:
        {
            if (column.Size > 0)
            {
                param += "(" + column.Size + ")";
            }
```

```
19            break;
20        }
21    }
22
23    return param;
24 }
25 </script>
```

下面来生成需要更新的字段，更新时仅能更新非主键字段的值，在 **SchemaExplorer** 中支持这种区别，使用 SourceTable.NonPrimaryKeyColumns 即可得到非主键字段的集合。

```
1 UPDATE [<%= SourceTable.Name %>] SET
2       <% for (int i = 0; i < SourceTable.NonPrimaryKeyColumns.Coun
t; i++) { %>
3       [<%= SourceTable.NonPrimaryKeyColumns[i].Name %>] = @<%= Sou
rceTable.NonPrimaryKeyColumns[i].Name %><% if (i < SourceTable.NonPri
maryKeyColumns.Count - 1) { %>,<% } %>
4       <% } %>
```

然后再使用 SourceTable.PrimaryKey.MemberColumns 得到数据表中的主键集合，生成更新条件

```
1 WHERE
2       <% for (int i = 0; i < SourceTable.PrimaryKey.MemberColumns.C
ount; i++) { %>
```

```
3       <% if (i > 0) { %>AND <% } %>
4       [<%= SourceTable.PrimaryKey.MemberColumns[i].Name %>] = @<%=
SourceTable.PrimaryKey.MemberColumns[i].Name %>
5       <% } %>
```

以下为整体的代码结构

```
 1 <%@ CodeTemplate Language="C#" TargetLanguage="T-SQL"
 2       Description="Generates a update stored procedure." %>
 3
 4 <%@ Property Name="SourceTable" Type="SchemaExplorer.TableSchema"
 5       Category="Context"
 6       Description="Table that the stored procedures should be base
d on." %>
 7
 8 <%@ Assembly Name="SchemaExplorer" %>
 9
10 <%@ Import Namespace="SchemaExplorer" %>
11
12 <script runat="template">
13 public string GetSqlParameterStatement(ColumnSchema column)
14 {
15       string param = "@" + column.Name + " " + column.NativeType;
```

```
16
17       switch (column.DataType)
18       {
19             case DbType.Decimal:
20             {
21                   param += "(" + column.Precision + ", " + column.
Scale + ")";
22                   break;
23             }
24             default:
25             {
26                   if (column.Size > 0)
27                   {
28                         param += "(" + column.Size + ")";
29                   }
30                   break;
31             }
32       }
33
34       return param;
35 }
36 </script>
```

```
37
38 ------------------------------------------------------------------
39 -- Date Created: <%= DateTime.Now.ToLongDateString() %>
40 -- Created By:   Generated by CodeSmith
41 ------------------------------------------------------------------
42
43 CREATE PROCEDURE dbo.Update<%= SourceTable.Name %>
44       <% for (int i = 0; i < SourceTable.Columns.Count; i++) { %>
45       <%= GetSqlParameterStatement(SourceTable.Columns[i]) %><% if
 (i < SourceTable.Columns.Count - 1) { %>,<% } %>
46       <% } %>
47 AS
48
49 UPDATE [<%= SourceTable.Name %>] SET
50       <% for (int i = 0; i < SourceTable.NonPrimaryKeyColumns.Coun
t; i++) { %>
51       [<%= SourceTable.NonPrimaryKeyColumns[i].Name %>] = @<%= Sou
rceTable.NonPrimaryKeyColumns[i].Name %><% if (i < SourceTable.NonPri
maryKeyColumns.Count - 1) { %>,<% } %>
52       <% } %>
53 WHERE
54       <% for (int i = 0; i < SourceTable.PrimaryKey.MemberColumns.
```

```
Count; i++) { %>
55        <% if (i > 0) { %>AND <% } %>
56        [<%= SourceTable.PrimaryKey.MemberColumns[i].Name %>] = @<%=
 SourceTable.PrimaryKey.MemberColumns[i].Name %>
57        <% } %>
58
```

CodeSmith 基础（三）

这里写的东东都是从 **CodeSmith** 自带的帮助文档中 **FAQ** 里学到的东东

1．如何在模板中添加注释

CodeSmith：

<%-- Comments --%>

VB.NET：

<%-- 'Comments --%>

C#：

<%-- // Comments --%>

<%-- /* Comments */ --%>

2．创建一个可以下拉选择的属性

首先定义一个枚举类型的变量，然后将属性的类型设置为枚举型

```
1 <%@ Property Name="CollectionType" Type="CollectionTypeEnum" Category
="Collection" Description="Type of collection" %>
2
3 <script runat="tempate">
4 public enum CollectionTypeEnum
5 {
6     Vector,
7     HashTable,
8     SortedList
9 }
10 </script>
```

3．解决 **ASP.NET** 中标签**<%**重复问题

先将 **ASP.NET** 中使用的这个重复标签写成**<%%**，避免在生成代码时由于是标签重复引起的编译错误或生成错误。

4．如何声明一个常量

```
<script runat="template">
private const string MY_CONST = "example";
</script>
```

5．如何对模板进行调试

如果要调试一个模板，首先要在代码模板里进行声明，然后在你想要进行调试的地方用**Debugger.Break()**语句设置断点即可。

```
<%@ CodeTemplate Language="C#" TargetLanguage="T-SQL" Description="Debugging your template" Debug="true" %>

<% Debugger.Break(); %>
```

6．如何将属性设置成选择一个文件夹的路径

```
[Editor(typeof(System.Windows.Forms.Design.FolderNameEditor), typeof(System.Drawing.Design.UITypeEditor))]

public string OutputDirectory
{
    get {return _outputDirectory;}
    set {_outputDirectory= value;}
}
```

7．怎样调用子模板

```
1 <%
2 foreach (TableSchema table in SourceDatabase.Tables)
```

```
3 {
4     OutputSubTemplate(table);
5 }
6  %>
7  <script runat="template">
8 private CodeTemplate _mySubTemplate;
9
10  [Browsable(false)]
11  public CodeTemplate MySubTemplate
12 {
13     get
14     {
15         if (_mySubTemplate == null)
16         {
17             CodeTemplateCompiler compiler = new CodeTemplateCompiler(this.Co
deTemplateInfo.DirectoryName + "MySubTemplate.cst");
18             compiler.Compile();
19             if (compiler.Errors.Count == 0)
20             {
21                 _mySubTemplate = compiler.CreateInstance();
22             }
23             else
```

```
24        {
25          for (int i = 0; i < compiler.Errors.Count; i++)
26          {
27            Response.WriteLine(compiler.Errors[ i].ToString());
28          }
29        }
30      }
31      return _mySubTemplate;
32    }
33  }
34
35  public void OutputSubTemplate(TableSchema table)
36  {
37    MySubTemplate.SetProperty("SourceTable", table);
38    MySubTemplate.SetProperty("IncludeDrop", false);
39    MySubTemplate.SetProperty("InsertPrefix", "Insert");
40    MySubTemplate.Render(Response);
41  }
42  </script>
```

FAQ 中给出的例子为生成一个数据库中所有表的更新 **Update** 存储过程

SubTemplatesExample.cst 文件源代码

```
1  <%@ CodeTemplate Language="C#" TargetLanguage="T-SQL"
2     Description="Generates a update stored procedure." %>
3
4
<%@ Property Name="SourceDatabase" Type="SchemaExplorer.DatabaseSchem
a"
5     Category="Context"
6     Description="Database" %>
7
8  <%@ Assembly Name="SchemaExplorer" %>
9
10  <%@ Import Namespace="SchemaExplorer" %>
11
12  <%
13  foreach (TableSchema table in SourceDatabase.Tables)
14  {
15     OutputSubTemplate(table);
16  }
17  %>
18
19  <script runat="template">
20  private CodeTemplate _mySubTemplate;
```

```
21
22
23
24
25  [Browsable(false)]
26  public CodeTemplate MySubTemplate
27  {
28      get
29      {
30          if (_mySubTemplate == null)
31          {
32              CodeTemplateCompiler compiler = new CodeTemplateCompiler(this.Co
deTemplateInfo.DirectoryName + "MySubTemplate.cst");
33              compiler.Compile();
34
35              if (compiler.Errors.Count == 0)
36              {
37                  _mySubTemplate = compiler.CreateInstance();
38              }
39              else
40              {
41                  for (int i = 0; i < compiler.Errors.Count; i++)
```

```
42          {
43              Response.WriteLine(compiler.Errors[ i].ToString());
44          }
45      }
46  }
47
48      return _mySubTemplate;
49  }
50 }
51
52  public void OutputSubTemplate(TableSchema table)
53 {
54      MySubTemplate.SetProperty("SourceTable", table);
55      MySubTemplate.SetProperty("IncludeDrop", false);
56      MySubTemplate.SetProperty("InsertPrefix", "Insert");
57
58      MySubTemplate.Render(Response);
59 }
60  </script>
```

MySubTemplate.cst 文件源代码

```
1  <%@ CodeTemplate Language="C#" TargetLanguage="T-SQL"
2      Description="Generates a update stored procedure." %>
```

```
3
4  <%@ Property Name="SourceTable" Type="SchemaExplorer.TableSchema"
5     Category="Context"
6     Description="Table that the stored procedures should be based on." %>
7
8  <%@ Assembly Name="SchemaExplorer" %>
9
10  <%@ Import Namespace="SchemaExplorer" %>
11
12
13  <script runat="template">
14  public string GetSqlParameterStatement(ColumnSchema column)
15 {
16     string param = "@" + column.Name + " " + column.NativeType;
17
18     switch (column.DataType)
19     {
20       case DbType.Decimal:
21       {
22         param += "(" + column.Precision + ", " + column.Scale + ")";
23         break;
24       }
```

```
25          default:
26            {
27              if (column.Size > 0)
28                {
29                  param += "(" + column.Size + ")";
30                }
31              break;
32            }
33        }
34
35      return param;
36  }
37  </script>
38
39  ------------------------------------------------------------
40  -- Date Created: <%= DateTime.Now.ToLongDateString() %>
41  -- Created By:   Generated by CodeSmith
42  ------------------------------------------------------------
43
44  CREATE PROCEDURE dbo.Update<%= SourceTable.Name %>
45      <% for (int i = 0; i < SourceTable.Columns.Count; i++) { %>
46
```

```
    <%= GetSqlParameterStatement(SourceTable.Columns[i]) %><% if (i < So
urceTable.Columns.Count - 1) { %>,<% } %>
47      <% } %>
48  AS
49
50  UPDATE [<%= SourceTable.Name %>] SET
51
    <% for (int i = 0; i < SourceTable.NonPrimaryKeyColumns.Count; i++) { %
>
52
    [<%= SourceTable.NonPrimaryKeyColumns[i].Name %>] = @<%= SourceT
able.NonPrimaryKeyColumns[i].Name %><% if (i < SourceTable.NonPrimaryKey
Columns.Count - 1) { %>,<% } %>
53      <% } %>
54  WHERE
55
    <% for (int i = 0; i < SourceTable.PrimaryKey.MemberColumns.Count; i++)
{ %>
56      <% if (i > 0) { %>AND <% } %>
57      [<%= SourceTable.PrimaryKey.MemberColumns[i].Name %>] = @<%= S
ourceTable.PrimaryKey.MemberColumns[i].Name %>
58      <% } %>
```

8．在加载模板时默认加载的命名空间 **Namespaces** 和组件 **Assemblies**

组件：**mscorlib, System, System.Xml, System.Data, System.Drawing, Microsoft.VisualBasic, System.Windows.Forms, CodeSmith.Engine**

命名空间：**System, System.Data, System.Diagnostics, System.ComponentModel, Microsoft.VisualBasic, CodeSmith.Engine**

9. 使用 **SchemaExplorer** 能否确定一个字段（**Field**）是标识字段（主键，**Identity Field**）

在字段的扩展属性集合中包含一个叫**"CS_IsIdentity"**的属性，如果这个属性的值为 **true**，则表名当前字段为一个标识字段

```
1 Identity Field = <% foreach(ColumnSchema cs in SourceTable.Columns) {
2      if( ((bool)cs.ExtendedProperties["CS_IsIdentity"].Value) == true)
3      {
4           Response.Write(cs.Name);
5      }
6 }
7 %>
```

CS_Identity_Example.cst 文件源代码

```
1 <%@ CodeTemplate Language="C#" TargetLanguage="T-SQL"
2    Description="Identifies the identity field of a table" %>
3
4 <%@ Property Name="SourceTable" Type="SchemaExplorer.TableSchema"
```

```
5    Category="Context"
6    Description="Table to target." %>
7
8 <%@ Assembly Name="SchemaExplorer" %>
9
10 <%@ Import Namespace="SchemaExplorer" %>
11
12
13
14 Identity Field = <% foreach(ColumnSchema cs in SourceTable.Columns) {
15                  if( ((bool)cs.ExtendedProperties["CS_IsIdentity"].Value) == t
rue)
16                  {
17                     Response.Write(cs.Name);
18                  }
19               }
20            %>
```

10．如何确定一个字段的默认值（各人认为翻译成如何知道一个字段有默认值并且默认值是什么）

在字段的扩展属性集合中包含一个叫**"CS_Default"**的属性

```
1 <%
2 foreach(ColumnSchema cs in SourceTable.Columns) {
```

```
3      if (cs.ExtendedProperties["CS_Default"] != null)
4      {
5          Response.WriteLine(cs.ExtendedProperties["CS_Default"].Value);
6      }
7 }
8 %>
```

11．如何使用 **SchemaExplorer** 得到存储过程的输入输出参数

使用 **CodeSmith** 提供的 **CommandSchema** 对象，它包含需要的输入输出参数集合

```
1 Input Parameters:
2 <%foreach(ParameterSchema ps in SourceProcedure.AllInputParameters)
3 {
4      Response.Write(ps.Name);
5      Response.Write("\n");
6 }
7 %>
8
9
10 Output Parameters:
11 <%foreach(ParameterSchema ps in SourceProcedure.AllOutputParameters)
12 {
13      Response.Write(ps.Name);
```

```
14      Response.Write("\n");
15 }
16 %>
```

InputOutputParameterExample.cst 文件源代码

```
1 <%@ CodeTemplate Language="C#" TargetLanguage="T-SQL"
2    Description="Generates a update stored procedure." %>
3
4 <%@ Property Name="SourceProcedure" Type="SchemaExplorer.CommandSc
hema"
5    Category="Context"
6    Description="The stored procedure to examine" %>
7
8 <%@ Assembly Name="SchemaExplorer" %>
9
10 <%@ Import Namespace="SchemaExplorer" %>
11
12 Input Parameters:
13 <%foreach(ParameterSchema ps in SourceProcedure.AllInputParameters)
14 {
15    Response.Write(ps.Name);
16    Response.Write("\n");
17 }
```

```
18 %>
19
20
21 Output Parameters:
22 <%foreach(ParameterSchema ps in SourceProcedure.AllOutputParameters)
23 {
24     Response.Write(ps.Name);
25     Response.Write("\n");
26 }
27 %>
```

CodeSmith 基础（四）

本文是翻译的第四篇，内容为在 **CodeSmith** 中使用的语法和标签的参考。

CodeSmith 模板语法参考

本文的目的是在编写一个 **CodeSmith** 模板时遇到的各种类型的变量和对象提供参考。本文的目的不是要介绍**CodeSmith**，如果您想快速了解**CodeSmith**请查看我翻译的**CodeSmith**基础（一）和 **CodeSmith** 基础（二）。

标签

标签一般出现在模板的头部，被用做设置许多不同的属性。

代码模板的声明（CodeTemplate Directive）

这个是模板中唯一必须的声明，包含一些模板特殊的属性，包含模板使用的语言、生成的语言和一些对于模板的描述。

例：

```
<%@ CodeTemplate Language="C#" TargetLanguage="C#" Description="Generates a class." %>
```

参数的介绍：

Language：在开发编写模板时使用的语言，例如 **C#**，**VB.NET**，**Jscript** 等。

TargetLanguage：只是对模板代码的一个分类，不会影响生成的代码语言。是模板的一个属性，说明模板要基于那种语言生成相应的代码。例如你可以用 **CodeSmith** 从任何一种语言生成 **C#**代码。

Description：对于模板的一些说明信息，在 **CodeSmith Explorer** 中选中该模板时会显示这里的信息。

Inherits：所有 **CodeSmith** 模板默认继承自 **CodeSmith.Engine.CodeTemplate**，这个类提供模板使用的一些基本功能，像 **ASP.NET** 页面的 **Page** 类，这些被继承的类的属性可以被修改，但是这些新的类也必须继承 **CodeSmith.Engine.CodeTemplate**。**CodeSmith** 也同样可以找到这个类，当然你要引入一个组件包含这个类。

Src：在某些方面 **Src** 和继承 **Inherits** 比较相似，它们都允许你从其他的类包含一些功能进模板。这两个属性的区别是，**Src** 可以让类与你的模板被动态编译，而 **Inherits** 仅允许你提供一个已经编译好的类或组件。

Debug：可以确定是否在模板中可以包含调试符号。如果将这个属性设置为 **True**，则可以使用 **System.Diagnostics.Debugger.Break()**方法来设置断点。

LinePragmas：设置为 **True**，模板的错误将被指向到模板的源代码。设置为 **False**，模板的错误将被指向到编译的源代码。

属性的声明（Property Directive）

属性被用做在模板运行时声明一个使用的参数，例：

```
<%@ Property Name="ClassName" Type="String" Default="Class1" Category="C
ontext" Description="The name of the class to generate" Optional="true" %>
```

属性参数的介绍：

Name：模版使用的参数的名称。

Type：参数类型可以是任何**.NET** 有效的数据类型，例如简单的 **String** 类型或者是 **CodeSmith** 的 **SchemaExplorer.DatabaseSchema** 类型。注意，类型必须是基类库的类型，例如用 **String** 或者 **Int32** 代替 **string** 和 **int**。

Default：设置默认值。

Category：用来说明这个属性在 **CodeSmith Explorer** 的属性面板中显示成什么类型，例如下拉选择、直接输入等。

Description：在属性面板中对于这个属性的描述。

Optional：设置这个属性是否是必须的，设置为 **True** 则这个参数必须有值，设置为 **False** 则表明这个参数值可有可无。

Editor：表明在属性面板中输入这个属性的值时使用何种 **GUI**（图形界面编辑器）编辑器。

EditorBase：编辑器使用的基本类型，如果没有被说明，**UITypeEditor** 为默认编辑器。

Serializer：这块我的水平不太会犯疑：）**The serializer parameter specifies the IPropertySerializer type to use when serializing the properties values. This is**

equivalent to using a [PropertySerializerAttribute].

XML 属性声明（XmlProperty Directive）

例：

```
<%@ XmlProperty Name="EntityMap" Schema="EntityMap.xsd" Optional="False"
Category="Context" Description="EntityMap XML file to base the output on." %>
```

XML 属性的参数：

Name：名称。

Schema：这个参数用来指定一个 **XSD** 文件，创建一个强类型对象模型。如果这个计划被指定，编译器会尝试分析这个 **XSD** 文件并为这个计划生成一个强类型对象模型，这样可以在模版中使用强类型和智能与 **XML** 协同工作。如果这个计划没有被设定，这个参数将为 **XmlDocument** 类型并且将使用 **XML DOM** 去导航到一个 **XML** 内容并生成代码。

Category：在 **CodeSmith** 属性面板中的类别。

Description：描述。

Optional：这个参数是否是必须的，如果设置为 **True**，则参数不是必须的，反之 **False** 则为必须的。在设置为 **False** 时，如果用户没有提供参数则 **CodeSmith** 不能继续运行。

注册的声明（Register Directive）

这个属性通常被用作引入另一个模版文件并与当前的模版文件同时被编译。这是一种使用子模版的交互方法。

例：

```
<%@ Register Name="MySubTemplate" Template="MySubTemplate.cst" MergePr
operties="True" ExcludeProperties="SomeExcludedPropertyName,SomeProperties
*" %>
```

模版一旦被注册，就可以建立一个模版的实例，然后象这样设置它的属性：

```
1 <script runat="template">
2 public void OutputSubTemplate()
3 {
4    MySubTemplate mySubTemplate = new MySubTemplate();
5
6    // set an individual properties value.
7    mySubTemplate.SomeExcludedPropertyName = "SomeValue";
8
9    // copy all properties with matching name and type to the sub template insta
nce.
10    this.CopyPropertiesTo(mySubTemplate);
11
12    // render the template to the current templates Response object.
13    mySubTemplate.Render(this.Response);
14
15    // render the template to a file.
16    mySubTemplate.RenderToFile("C:\SomeFile.txt");
```

```
17 }
18 </script>
```

注册的参数：

Name：代表被引入的模版的名称。它可以被用作创建一个模版的实例。

Template：被引入模版文件的相对路径，它可以与当前的模版一起被动态的编译。

MergeProperties：设置成 **True** 时，所有被引用的面板的属性将被动态的添加到当前模版中。

ExcludePorperties：当使用 **MergeProperties** 时，你可能不需要某些属性被添加到当前模版中。将不需要的属性以逗号分隔放在这里，*****号可以被用作通配符使用。

组件的声明（Assembly Directive）

用作在模版中引用一个外部部组件，或者包含一个编译好的源文件。

例：

```
<%@ Assembly Name="SchemaExplorer" %>
```

或

```
<%@ Assembly Src="MySourceFile.cs" %>
```

CodeSmith 自动加载一些不同的组件：**System, System.Diagnostics, System.ComponentModel, Microsoft.VisualBasic, CodeSmith.Engine**

组件的参数：

Name：需要引用组件的名称，组建必须存在于 **Global Assembly Cache**，与 **CodeSmith**

在同一路径下或与模版文件在同一路径下。

Src：要包含文件的相对路径。

引入的声明（Import Directive）

在模版中引入一个命名空间，这个与 **VB.NET** 中的 **Imports** 和 **C#**中的 **using** 相同。

例：

```
<%@ Import Namespace="SchemaExplorer" %>
```

引入的参数：

NameSpace：被引入的命名空间的名字。记住同时必须要加载包含这个命名空间的相应组件，除非这个组件是被默认加载的。

CodeSmith 基础（五）

本篇将介绍 CodeSmith 的模版中的语法。

代码标签

<% %>标签

可以放置任意数量的代码在其中，但并不能直接输出到模版中。

```
<% foreach (ColumnSchema column in SourceTable.Columns) { %>
<%= column.Name %>
<% } %>
```

<%= %>标签

在模版中输出一个字符串。上例中的<%=column.Name%>

脚本标签

在这个标签中可以包含一段代码，但是他不直接影响输出的模版。可以放置一些比较有帮助的方法在其中，然后在模版的各个地方可以调用它。在脚本标签中必须包含这个参数 runat=”template”，否则他会被处理成普通文本。

例：

```
<script runat="template">



private string GetColumnName(ColumnSchema cs)

{

      return cs.Name;

}

</script>



<% foreach (ColumnSchema cs in SourceTable.Columns) { %>

<%= GetColumnName(cs) %>

<% } %>
```

使用标签可以大量减少代码，并使模版更加的易读和一管理。

Include 标签

和 ASP.NET 一样，可以在模版中包含一些文本文件，但同 ASP.NET 一样它也不是总能达到你的目标。

例：

```
<!-- #include file="myfile.inc" -->
```

有时在多个模版中引用一个组件中的功能，调用其中的方法，这时我们引用组件。但有些情况下，适用 Include 标签可以得到更好的效果。

Comment 标签

注释标签，在前边已经做过介绍。

例：

```
<%-- This is a comment --%>
```

CodeSmith 基础（六）

本文主要介绍 CodeSmith 对象。

CodeSmith Object

CodeSimth 中有许多对象可以在编写模板的时候使用，这里将介绍这些对象的一些公用方法和属性以及怎么使用它们。

代码模板对象（*CodeTemplate Object*）

在模板中，“this”（或者“Me”在 VB.NET 中）在当前模板中代码代码模板对象。

代码模板的方法（*CodeTemplate Methods*）

1．**public virtual void GetFileName()**

可以重载这个方法设置模板输出到文件的名称。否则 CodeSmith 将基于模板名称和 TargetLanguage 设置它的文件名。

2．**public void CopyPropertiesTo(CodeTemplate target)**

这个方法可以实现从一个模板中将其所有属性的值拷贝到另一个模板所有对应属性中，并按照相应的属性值类型进行匹配。

3．**public object GetProperty(string propertyName)**

这个方法将返回一个给定名称的属性的值。

4．**public void SetProperty(string propertyName, object value)**

此方法可以根据给定名称的属性设置其值。

5．**public string SavePropertiesToXml ()**

这个方法将现有的属性值保存成一个 XML 的属性字符串。

6．**public void SavePropertiesToXmlFile (string fileName)**

这个方法将当前属性值保存成一个 XML 的属性文件。

7. **public void RestorePropertiesFromXml(string propertySetXml, string baseDirectory)**

从保存在 XML 文件中的属性字符串，将模板的属性值恢复。

8．**public void RestorePropertiesFromXmlFile(string fileName)**

从保存在 XML 文件中的属性文件，将模板的属性值恢复。

代码模板的属性（*CodeTemplate Properties*）

Response：此属性可以访问当前的 TextWriter 对象，这个对象是用来输出模板用的。

CodeTemplateInfo：这个属性用来访问当前的 CodeTemplateInfo 对象，这个对象包含当前模板的一些信息。

Progress：这个属性用来报告当前模板的执行过程。

Response Object

这个对象提供直接写输出模板的方法。与 ASP.NET 的 response 对象很相似。下面是一个利用 Response 的 Write 方法在模板上输出一段文字的例子。

```
<% Response.Write("This will appear in the template") %>
```

IndentLevel (Int32)

当使用 Response 对象时输出文本的缩进级别。

Indent() Method

将输出缩进一个级别。

Unindent() Method

将输出少缩进一个级别。

AddTextWriter(TextWriter writer) Method

为 Response 对象增加一个 TextWriter。这样可以使在同一时间用多个 TextWriter 输出模板。

CodeTemplateInfo Object

此对象包含一些当前模板的信息。下面是一些 CodeTemplateInfo 可用的属性。

DateCreated (DateTime)

返回一个 date 类型值，是模板创建的时间。

DateModified (DateTime)

返回模板最后一次被修改的时间。

Description (string)

返回模板声明时对模版的描述信息。

DirectoryName (string)

返回当前模板文件所在的路径。

FileName (string)

返回当前模版文件的文件名称。

FullPath (string)

返回当前模板的完整路径，路径名+文件名。

Language (string)

返回代码模版声明时使用的语言。

TargetLanguage (string)

返回代码模版声明时生成的目标语言。

Progress Object

这个属性用来报告当前模板的执行过程。下面是一些 Progress 可用的成员。

MaximumValue (Int32)

模版 progress 允许的最大值。

MinimumValue (Int32)

模版 progress 允许的最小值。

Step (Int32)

模版每执行一不 progress 的增长值。

Value (Int32)

Progress 的当前值。

PerformStep() Method

按照指定好的 progress 的增加值执行一步。（原文：Perform a progress step incrementing

the progress value by the amount specified in the Step property.）

Increment(Int32 amount) Method

指定 progress 的增加值。（原文：Increment the progress value by the specified amount.）

OnProgress (ProgressEventHandler) Event

这个事件用来报告模版的执行过程。（原文：This event can be used to be notified of template execution progress.）

```
this.Progress.OnProgress += new

ProgressEventHandler(this.OnProgress);



public void OnProgress(object sender, ProgressEventArgs e)

{

  Trace.WriteLine(e.Value);

}
```

CodeSmith 基础（七）

本文翻译的内容为 CodeSmith 控制台指南。

很多人仅仅知道 CodeSmith 像一个图形应用程序，或者可能是一个 Visual Studio 的附件，但是通过 CodeSmith 的控制台应用程序还有好多其他的使用方法。控制台应用程序是很有价值的，因为可以通过它去生成脚本，或者其他一些自动工具。这篇文档的目的就是要告诉你怎样使用它的控制台应用程序并且如何去定义变量和参数。

Basic Usage

大多数情况下是用控制台应用程序来创建一个模板，一个属性文件，然后保存输出的文件。这有一个很好的例子介绍将合并模版的处理过程放到一个过程中，就像使用 NAnt 工具。

首先我们要确定完成一个什么样的模版，为这个模板创建一个什么样的 XML 属性文件。XML 属性文件提供在执行模版是需要的各个属性。生成一个属性文件最简单的方法是在 CodeSmith Explorer 中打开一个模版，填写属性，点击生成按钮 generate，然后再点击 Save Property Set XML 按钮。这个按钮会在点击完生成按钮后找到，在 Save Output 和 Copy Output 按钮旁边。然后系统提示输入保存 XML 属性文件的文件名，下面看一个 ArrayList.cst 模版创建的 XML 属性文件。

```
<?xml version="1.0" encoding="us-ascii"?>
<codeSmith>
    <propertySet>
        <property name="Accessibility">Public</property>
        <property name="ClassName">PersonArray</property>
        <property name="ItemType">Person</property>
        <property name="ItemValueType">False</property>
        <property name="ItemCustomSearch">False</property>
        <property name="KeyName">PersonID</property>
        <property name="KeyType">int</property>
        <property name="IncludeInterfaces">True</property>
        <property name="IncludeNamespaces">False</property>
    </propertySet>
</codeSmith>
```

就像看到的一样，也可以手动创建这个文件，但是使用 CodeSmith Explorer 会更简便。

现在我们有了这个 XML 文件，我们继续看一下如何去执行这个模版并是用控制台工具保存结果。首先我们需要是用/template 参数去声明我们要是用的模版，像这样：

```
C:\Program                                          Files\CodeSmith\v3.0>cs
/template:Samples\Collections\ArrayList.cst
```

在这个例子中我们使用了 ArrayList.cst 模版，它存储在本地的 Samples\Collections 文件夹下。下一步我们要去声明我们在最后一步需要创建的 XML 文件，我们是用/propertyset 参数去实现。

```
C:\Program                                          Files\CodeSmith\v3.0>cs
/template:Samples\Collections\ArrayList.cst  /propertyset:PersonArray
.xml
```

这个/property 参数用来指定我们的 XML 属性文件。最后一个我们需要用的参数是/output 参数，用来指定输出怎样被保存。

```
C:\Program Files\CodeSmith\v3.0>cs
/template:Samples\Collections\ArrayList.cst
/propertyset:PersonArray.xml /out:test.cs
```

使用/out 参数指定将结果输出到一个叫 test.cs 文件中保存。执行这个命令后，模板将开始运行，使用属性文件将结果输出到 test.cs 文件保存。

这是大多数情况下有效使用控制台。

Merging Output

在各种代码生成中最大的挑战就是将生成的代码和开发人员编写或修改的代码区分开。

控制台对这个问题提供了一个有效的独特的解决方案，使用一个指定的参数在当前已存在的代码文件中需要将模板生成的代码添加的地方指定一块区域。

下面是一个简单的代码文件，包含了我们要添加生成代码的区域。

```
using System;


namespace Entities
{
      #region GeneratedOrderEntity




      #endregion
}
```

我们的目标是将 DatabaseSchema\BusinessObject.cst 模版生成的代码添加到类文件的 GeneratedOrderEntity 区域中。和上一个例子一样，使用 CodeSmith console 控制台应用程序执行这个模版，但是这次要使用另一个参数 merge。

```
C:\Program Files\CodeSmith\v3.0>cs
/template:Samples\DatabaseSchema\BusinessObjec
  t.cst /propertyset:OrderEntity.xml /out:OrderEntity.cs
  /merge:InsertRegion= "RegionName=Sample Generated
  Region;Language=C#;"
```

使用 merge 参数我们可以指定区域的名称，在这个例子中是 GeneratedOrderEntity，然后控制台应用程序将执行模版，并将结果添加到这个区域中。我们来看一下执行完这个指令

后生成的代码。

```
using System;


namespace Infozerk.AuthServices.UnitTestSuite
{
	#region GeneratedOrderEntity


	#region Order
	/// <summary>
	/// This object represents the properties and methods of a Order.
	/// </summary>
	public class Order
	{
		protected int _id;
		protected string _customerID = String.Empty;
		protected int _employeeID;
		protected DateTime _orderDate;
		protected DateTime _requiredDate;
		protected DateTime _shippedDate;
		protected int _shipVia;
```

--为了简短省略了类的其他部分

就像看到的一样，Order 类被添加到了我们指定的区域中。在代码文件中使用 merge 参数生成的内容在其他部分被修改或手写后很容易重新再次生成而不会产生影响。

参数介绍 Parameter Reference

Specifying Output

/out:<file>

指定从模版创建的输出文件的名称。

/out:default

指定这个文件被默认保存成模版是用的名称。

/merge:<mergetype>=<init>

指定模版输出的区域。可以简写为/m

Specifying Input

/template:<file>

选择要执行的模版，简写为/t

/propertyset:<file>

生成代码时需要使用的 XML 属性文件。简写为/p

Compiler Options

/debug[+|-]

指定模版需要包含的调试信息。（允许在运行模版时进行调试）

/tempfiles[+|-]

指定保留临时文件。（如果在临时文件上调试也可以）

Miscellaneous

/help

显示帮助信息。

/nologo

禁止生成器版权信息。

CodeSmith 基础（八）

编写 CodeSmith 自定义属性的编辑器（Writing Custom Property Editors）

当你开始编写自定义的CodeSmith模板时，很可能对于使用它的strings或integers属性很满意，但有时你会发现需要创建一个不同类型的属性，可能是一个自定义的类型或者是.NET framework中但是在属性面板中没有提供的类型。在模板中去作这些很简单，但是怎样指定一个类型在运行模板时显示在属性面板中呢？例如创建了一个Person类并且具有很多不同的属性，但是却没有办法让用户去组装这个类……除非创建一个自定义属性编辑器。

属性面板提供了方法去编写自定义的属性编辑器，当用户在面板上选择一个属性时可以激发相应的方法，同时也可以通过编写代码实现提示用户输入一个必要的值。下面我们举个例子，创建一个接受组建的属性并且是用影射循环贯串所有的类，是所有的类都可以使用它和它的方法，去创建一个 NUnit 测试基础。（这句翻译的不好，原文：As an example we are going to build a template which accepts an assembly as a property and then using reflection loops through all of the classes, and the methods of those classes, to build NUnit test stubs.）

首先，我们来关注一下模板的组件属性，暂且不看自定义编写的代码。模板的第一部分是一些声明定义和属性。将属性放在脚本标签中代替使用属性声明，在下一部分将看到这样做的必要。

```
<%@ CodeTemplate Language="C#" TargetLanguage="C#" Description="Builds
a class for each class in the assembly, and a test stub for every method."
%>
<%@ Import NameSpace="System.Reflection" %>

<script runat="template">

private Assembly assembly;

public Assembly AssemblyToLoad
{
      get{return assembly;}
      set{assembly = value;}
}

</script>
```

然后我们为组建 assembly 中的每一个类创建一个类，为每一个类创建他的方法。然后直接将模板的输出内容放入 Visual Studio.NET，然后在编写组件的单元测试时使用向导。

```
using System;
```

```
using NUnit.Framework;



<%

      foreach(Type T in AssemblyToLoad.GetTypes())

      {

            if(T.IsClass)

            {

                  %>



                  [TestFixture]

                  public class <%=T.Name%>Tests

                  {

                  <%

                              MethodInfo[] methods = T.GetMethods

( BindingFlags.Public | BindingFlags.Instance | BindingFlags.Static  );

                              foreach(MethodInfo M in methods)

                              {

                                    %>



                                    [Test]

                                    public void <%=M.Name%>Test

                                    {
```

```
                                        //TODO Write this test

                                    }

                                    <%

                                }

                    %>}<%

            }

        }
%>
```

这个模板仅仅可以编译通过，但是由于我们编写显示了一个类型属性面板并不知道如何去操作它，所以我们没有办法自定义指定组件在加载时想要加载的组件。

我们需要创建一个 UITypeEditor，这是一个建立属性面板是用的特殊属性的类。UITypeEditor 需要创建在一个和模板分离的组件中，

CodeSmith 实用技巧（一）：使用 StringCollection

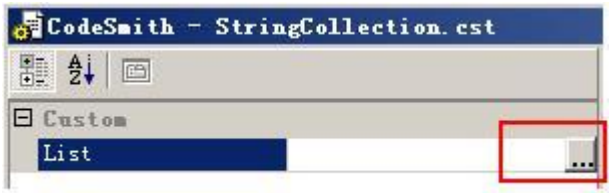
StringCollection 提供了一种集合的输入方式，在代码中，可以用 **Array** 的方式来引用。在使用这个类之前，在模版中我们必须添加对 `CodeSmith.CustomProperties` 程序集的引用：

```
<%@ Assembly Name="CodeSmith.CustomProperties"%>
```

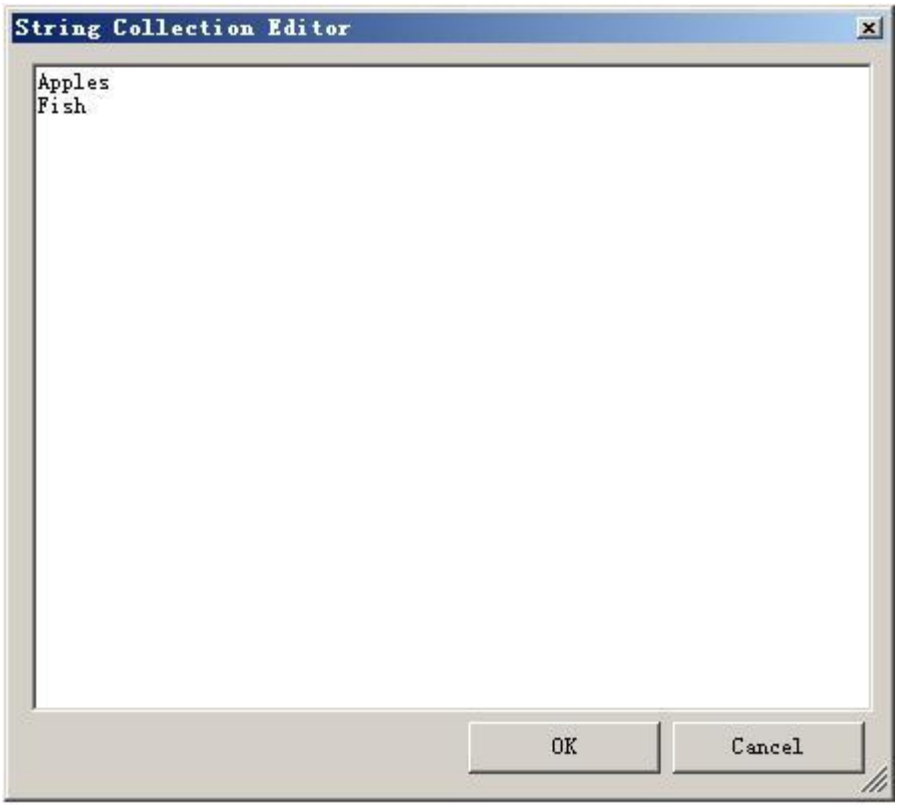
添加完程序集之后，我们就可以使用 `StringCollection` 在脚本块中定义一个属性：

```
<%@ Property Name="List" Type="CodeSmith.CustomProperties.StringCollection
" Category="Custom" Description="This is a sample StringCollection"%>
```

执行该模版时，这个属性将在属性窗体中显示为一个按钮：



单击按钮，将会弹出一个 String Collection Editor 对话框：



当然也可以直接在属性窗口中编辑 StringCollection。

模版代码如下：

```
1  <%@ CodeTemplate Language="C#" TargetLanguage="C#" %>
2
3  <%@ Assembly Name="CodeSmith.CustomProperties" %>
4
5
<%@ Property Name="List" Type="CodeSmith.CustomProperties.StringCollection" Category="Custo
m" Description="This is a sample StringCollection" %>
6
7  using System;
8  namespace Test
9
10 {
11      /// <summary>
12
13      ///     测试 StringCollection
14
15      /// </summary>
16
17      public class Test
18
19      {
```

```
20
21              public static void Main(string[] args)
22
23              {
24
25                  <%for(int i = 0;i<List.Count;i++){%>
26
27                      Console.WriteLine(<%=List[i]%>);
28
29                  <%}%>
30
31              }
32
33      }
34
35  }
36
37
```

生成后的代码：

```
1  using System;
2
3  namespace Test
4
5  {
6          ///<summary>
7
8          ///     测试 StringCollection
9
10         /// </summary>
11
12         public class Test
13
14         {
15
16                 public static void Main(string[] args)
17
18                 {
19
20                         Console.WriteLine(Apples);
21
22                         Console.WriteLine(Fish);
```

```
23
24              }
25
26          }
27
28  }
29
```

StringCollection 的重要属性和方法：

公共属性

名称	描述
Count	获取 StringCollection 中包含的字符串的数目
IsReadOnly	获取用于指示 StringCollection 是否为只读的值
IsSynchronized	获取一个值，该值指示对 StringCollection 的访问是否为同步的（线程安全的）
Item	获取或设置指定索引处的元素。在 C# 中，该属性为 StringCollection 类的索引器
SyncRoot	获取可用于同步对 StringCollection 的访问的对象

公共方法

名称	描述
Add	将字符串添加到 StringCollection 的末尾

AddRange	将字符串数组的元素复制到 StringCollection 的末尾
Clear	移除 StringCollection 中的所有字符串
Contains	确定指定的字符串是否在 StringCollection 中
CopyTo	从目标数组的指定索引处开始，将全部 StringCollection 值复制到一维字符串数组中
IndexOf	搜索指定的字符串并返回 StringCollection 内的第一个匹配项的从零开始的索引
Insert	将字符串插入 StringCollection 中的指定索引处
Remove	从 StringCollection 中移除特定字符串的第一个匹配项
RemoveAt	移除 StringCollection 的指定索引处的字符串

CodeSmith 实用技巧（二）：使用 FileNameEditor

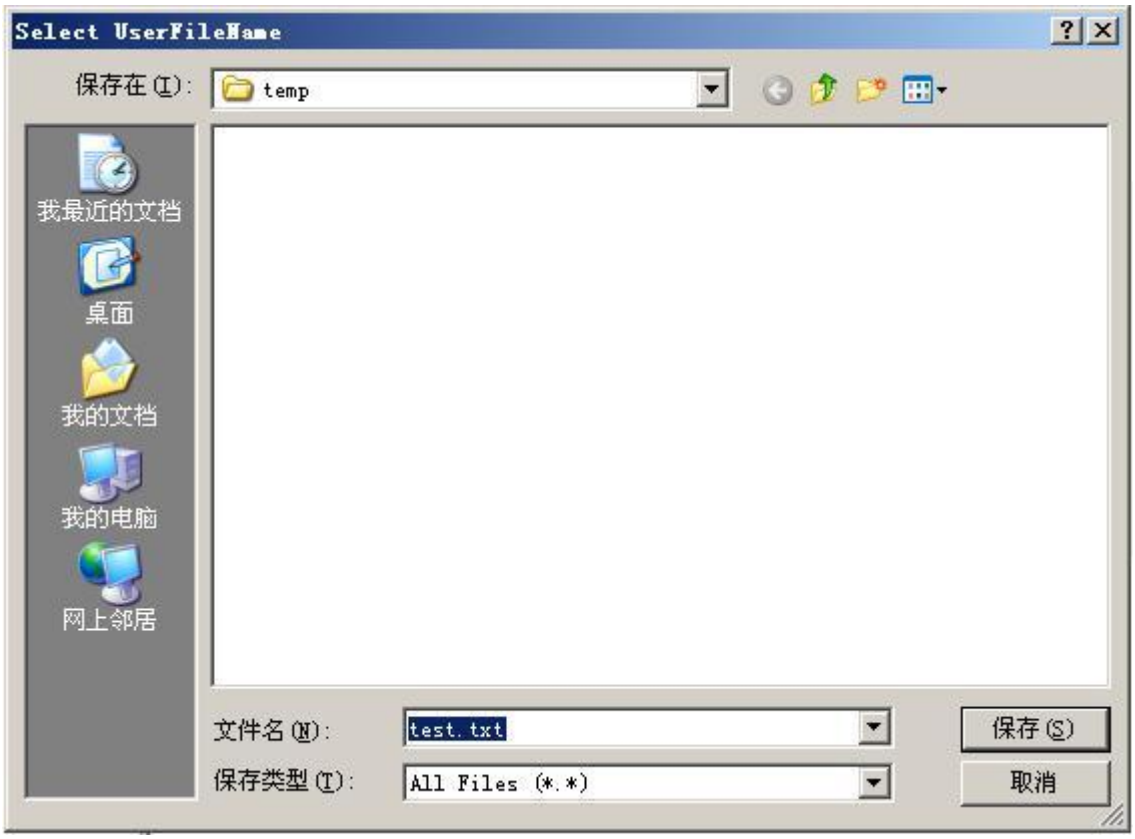
FileNameEditor 类给我们提供了在 CodeSmith 属性面板中弹出打开或保存文件对话框的方式，在使用时，首先在模版中得添加对程序集 CodeSmith.CustomProperties 的引用。然后就可以在模版中定义一个属性来使用 FileNameEditor：

```
1  <script runat="template">
2
3  private string _userFileName = @"c:\temp\test.txt";
4
5
6
7  [Editor(typeof(FileNameEditor), typeof(System.Drawing.Design.UITypeEditor)),
```

```
8
9  Category("Custom"), Description("User selected file.")]
10
11
12
13   public string UserFileName
14
15 {
16
17       get {return _userFileName;}
18
19       set {_userFileName= value;}
20
21 }
22
23   </script>
24
25
```

当我们执行该模版时，在属性面板中同样显示为一个按钮：

单击该按钮，弹出一个保存文件的对话框：

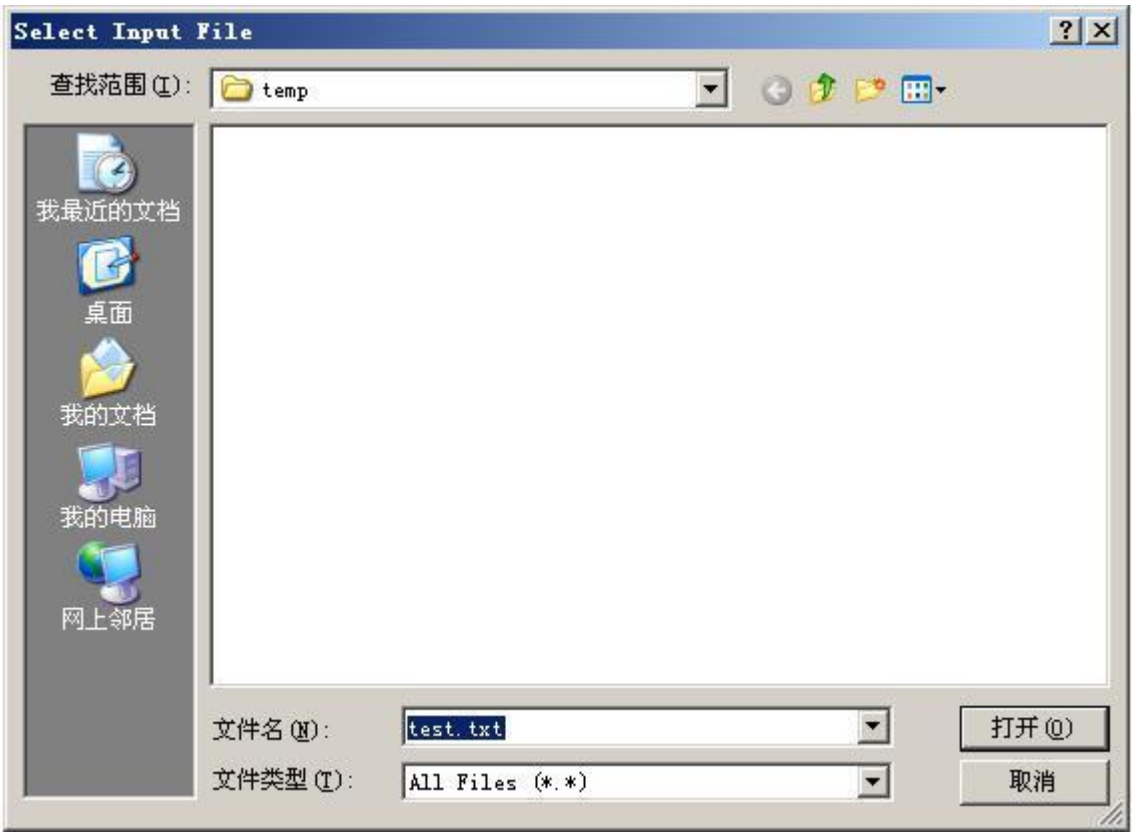


我们也可以通过 `FileDialogAttribute` 来自定义弹出的对话框，修改模版为：

```
1  private string _openFileName = @"c:\temp\test.txt";
2
3
4  [Editor(typeof(FileNameEditor), typeof(System.Drawing.Design.UITypeEditor)),
5
```

```
6   FileDialogAttribute(FileDialogType.Open, Title="Select Input File"),
7
8   Category("Custom"), Description("User selected file.")]
9
10
11
12   public string OpenFileName
13
14  {
15
16      get {return _openFileName;}
17
18      set {_openFileName= value;}
19
20  }
21
22
```

弹出的对话框如下所示：



当我们想用一个文件夹的名称来代替文件时，可以使用 **FolderNameEditor** 类。

```
1  <%@ Assembly Name="System.Design" %>
2  <script runat="template">
3  private string _outputDirectory = @"c:\temp";
4
[Editor(typeof(System.Windows.Forms.Design.FolderNameEditor), typeof(System
.Drawing.Design.UITypeEditor)),
5  Category("Custom"), Description("Output directory.")]
6  public string OutputDirectory
7  {
8        get {return _outputDirectory;}
```

```
9        set {_outputDirectory= value;}
10   }
11   </script>
12
13
```

FileNameEditor 重要方法和属性介绍：

公共方法：

名称	描述
EditValue	使用由 GetEditStyle 方法提供的编辑器样式编辑指定的对象
GetEditStyle	获取 EditValue 方法所使用的编辑样式

CodeSmith 实用技巧（三）：使用 FileDialogAttribute

使用 **FileDialogAttribute** 可以设置 **FileNameEditor** 中的属性，基本成员如下：

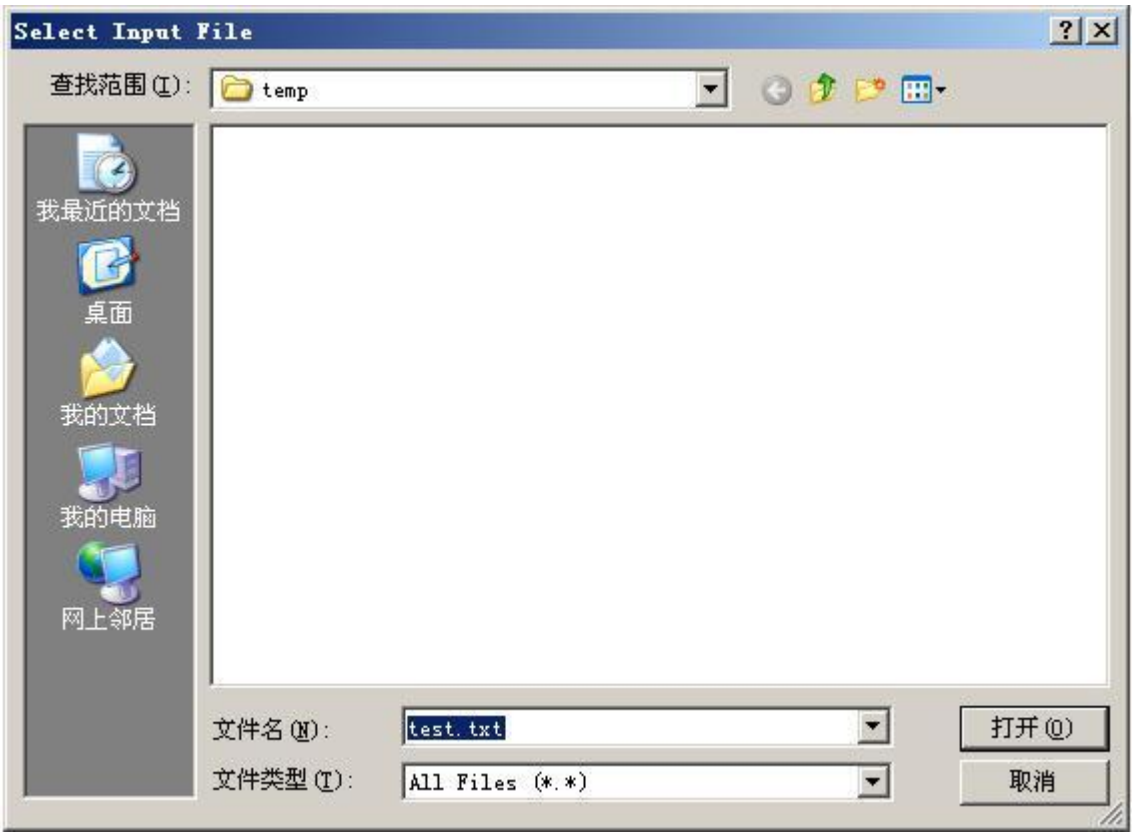
属性	描述	默认值
FileDialogType	**Save or Open**	**FileDialogType.Save**
Filter	**Filter string for file extensions**	**All Files (*.*)|*.***
Title	**Dialog box title**	**Select *propertyname***

DefaultExtension	**Default file extensions**	**None**
CheckFileExists	**True to only allow selecting existing files**	**False**
CheckPathExists	**True to only allow using existing paths**	**False**

在下面这段模版代码中，我们设置了弹出的对话框的类型为打开文件对话框，标题为“Select Input File”。

```
1  private string _openFileName = @"c:\temp\test.txt";
2  [Editor(typeof(FileNameEditor), typeof(System.Drawing.Design.UITypeEditor)),
3  FileDialogAttribute(FileDialogType.Open, Title="Select Input File"),
4  Category("Custom"), Description("User selected file.")]
5  public string OpenFileName
6  {
7      get {return _openFileName;}
8      set {_openFileName= value;}
9  }
10
11  执行
12
```

后弹出的对话框界面如下：



CodeSmith 实用技巧（四）：使用扩展属性

SchemaExplorer 允许我们获取数据库中一些对象的信息。如果你使用的是 **SQL Server2000** 数据库，你可以通过扩展属性获取很多对象的有用的信息。例如：**SQL Server** 定义了一个扩展属性来标识表中的唯一标识字段，在模版中可以这样写：

```
Identity Field =<% foreach(ColumnSchema cs in SourceTable.Columns) {
    if( ((bool)cs.ExtendedProperties["CS_IsIdentity"].Value) == true)
    {
        Response.Write(cs.Name);
    }
}
 %>
```

CodeSmith 定义的扩展属性包括 **table columns, view columns,** 和 **command parameters**。

Table columns

CS_IsRowGuidCol

CS_IsIdentity

CS_IsComputed

CS_IsDeterministic

CS_IdentitySeed

CS_IdentityIncrement

CS_Default

view columns

CS_IsComputed

CS_IsDeterministic

command parameters

CS_Default

另外，每个对象都有一个 CS_Description 的扩展属性。你也可以通过 SQL Server 中的系统存储过程 sp_addextendedproperty 来创建自定义的扩展属性。例如：我们执行如下命令为 Customer 这张表的 ID 字段添加一个 Caption 的扩展属性：

```
sp_addextendedproperty 'caption', 'Customer ID', 'user', dbo, 'table', Customers, 'column', id
```

在数据库中执行完这条语句后，**CodeSmith** 中将会在这个字段的扩展属性集合中加上 **Caption** 属性。有关 **SQL Server** 中的扩展属性的内容请参考联机丛书。

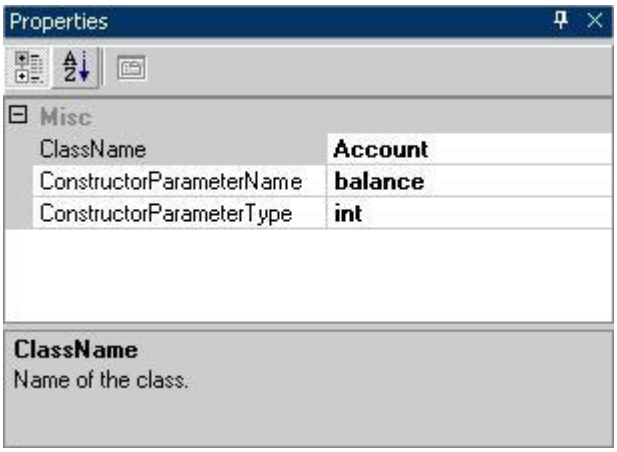
CodeSmith 实用技巧（五）：利用继承生成可变化的代码

用 **CodeSmith** 生成可变化的代码，其实是先利用 **CodeSmith** 生成一个基类，然后自定义其它类继承于该类。当我们重新生成基类时 **CodeSmith** 不要接触继承的子类中的代码。看下面的这段模版脚本：

```
<%@ CodeTemplate Language="C#" TargetLanguage="C#" Description="Base class generator."%>

<%@ Property Name="ClassName" Type="System.String" Description="Name
```

```
of the class." %>


<%@ Property Name="ConstructorParameterName" Type="System.String" Des
cription="Constructor parameter name." %>


<%@ Property Name="ConstructorParameterType" Type="System.String" Des
cription="Data type of the constructor parameter." %>
 class <%= ClassName %>
{
    <%= ConstructorParameterType %> m_<%= ConstructorParameterName
%>;

    public <%= ClassName %>(<%= ConstructorParameterType %> <%= Con
structorParameterName %>)
    {
        m_<%= ConstructorParameterName %> = <%= ConstructorParamete
rName %>
    }
}
```

执行该模版并输入如下数据：



该模版生成的代码可能如下：

```
1  class Account
2  {
3      int m_balance;
4
5      public Account(int balance)
6      {
7          m_balance = balance
8      }
9
10 }
11
12
```

把生成的文件保存为 Account.cs 文件。这时我们可以编写第二个类生成 Check.cs 文件代码：

```
1 class Checking : Account
2 {
3     public Checking : base(0)
4     {
5     }
6 }
```

现在如果需要改变 Account Balance 的类型为浮点型，我们只需要改变 ConstructorParameterType 属性为 float，并重新生成 Account.cs 文件即可而不需要直接在 Account.cs 中进行手工修改，并且不需要修改 Check.cs 文件的任何代码。

CodeSmith 实用技巧（六）：使用 XML 属性

CodeSmith 允许我们存储元数据在 **XML** 文件中，然后在执行模版时直接打开 **XML** 文件填写到属性面板中。

1. XML Property With a Schema

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <xs:schema targetNamespace=http://www.codesmithtools.com/PO
3     xmlns:xs=http://www.w3.org/2001/XMLSchema
4     xmlns=http://www.codesmithtools.com/PO
5     elementFormDefault="qualified" attributeFormDefault="unqualified">
6   <xs:element name="PurchaseOrder">
7     <xs:complexType>
```

```
 8       <xs:sequence>
 9        <xs:element name="PONumber" type="xs:string"/>
10        <xs:element name="CustomerName" type="xs:string"/>
11        <xs:element name="CustomerCity" type="xs:string"/>
12        <xs:element name="CustomerState" type="xs:string"/>
13        <xs:element name="Items">
14         <xs:complexType>
15          <xs:sequence>
16           <xs:element name="Item" maxOccurs="unbounded">
17            <xs:complexType>
18             <xs:attribute name="ItemNumber" type="xs:string" use="required"/>
19             <xs:attribute name="Quantity" type="xs:integer" use="required"/>
20            </xs:complexType>
21           </xs:element>
22          </xs:sequence>
23         </xs:complexType>
24        </xs:element>
25       </xs:sequence>
26      </xs:complexType>
27     </xs:element>
28  </xs:schema>
```

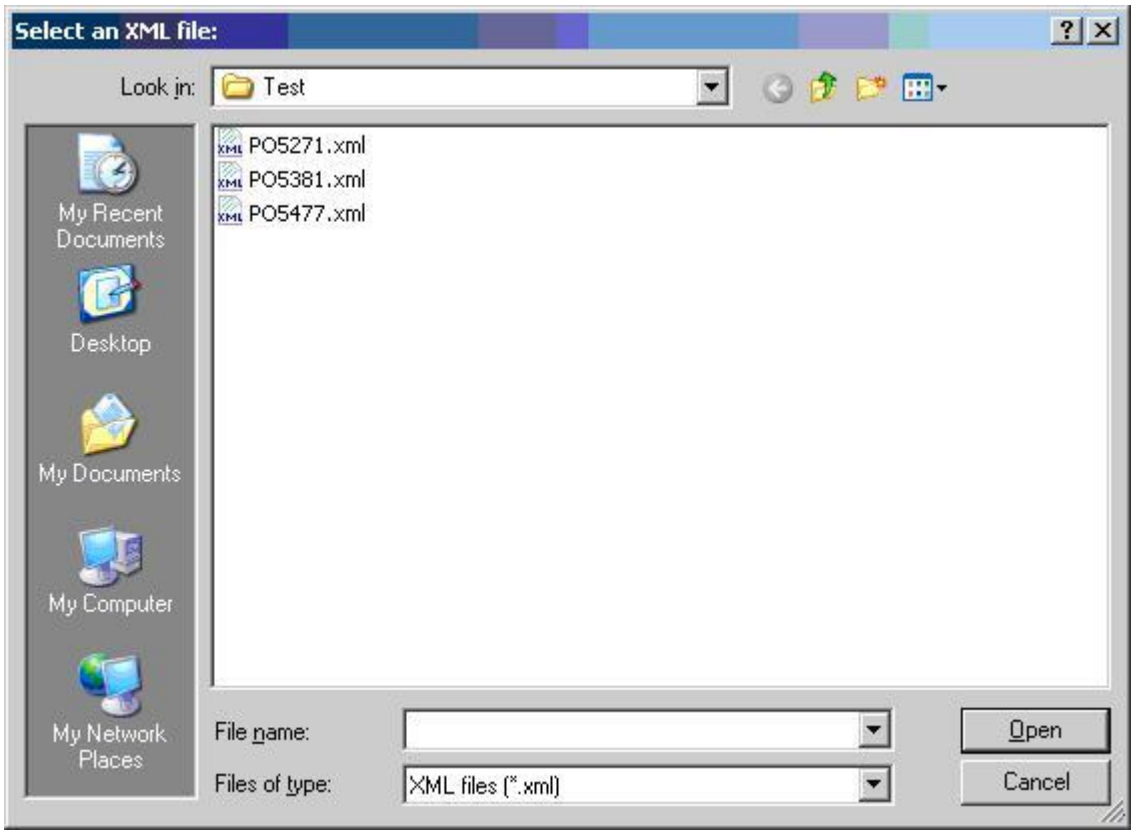
```
29

30
```

这是一个简单的带有 Schema 的 XML Property 的例子：

利用这个 Schema 文件，我们可以定义一个 XML Property 来在运行时读去元数据。

```
<%@ CodeTemplate Language="C#" TargetLanguage="Text" Description="Create packing list from
XML PO." %>

<%@ XmlProperty Name="PurchaseOrder" Schema="PO.xsd" Optional="False" Category="Data" De
scription="Purchase Order to generate packing list for." %>
  Packing List
  ref: PO#<%= PurchaseOrder.PONumber %>
  Ship To:
  <%= PurchaseOrder.CustomerName %>
  <%= PurchaseOrder.CustomerCity  %>, <%= PurchaseOrder.CustomerState %>
  Contents:
<% for (int i = 0; i < PurchaseOrder.Items.Count; i++) { %>
 <%= PurchaseOrder.Items[i].ItemNumber %>, Quantity <%= PurchaseOrder.Items[i].Quantity %>
<% } %>
```

在运行时，**PurchaseOrder** 属性在属性面板中显示为按钮，单击后弹出一个对话框供用户选择 **XML** 文件。



选择一个 XML 文件。在该例子 XML 文件内容如下：

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <PurchaseOrder xmlns=http://www.codesmithtools.com/PO
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
4    <PONumber>5271</PONumber>
5    <CustomerName>John Nelson</CustomerName>
6    <CustomerCity>Gamonetta</CustomerCity>
7    <CustomerState>MS</CustomerState>
8    <Items>
```

```
9       <Item ItemNumber="HM85" Quantity="12"/>
10      <Item ItemNumber="JR82" Quantity="4"/>
11      <Item ItemNumber="PR43" Quantity="6"/>
12    </Items>
13  </PurchaseOrder>
14
15
```

生成后的代码如下：

```
Packing List
ref: PO#5271
Ship To:
John Nelson
Gamonetta, MS
Contents:
HM85, Quantity 12
JR82, Quantity 4
PR43, Quantity 6
```

2. XML Property Without a Schema

这是一个不带 Schema 的 XML Property 的例子。这个模版在运行时可以访问任何 XML 文件。

```
<%@ CodeTemplate Language="VB" TargetLanguage="Text" Description="List top-level nodes in an
 XML file." %>

<%@ XmlProperty Name="TargetFile" Optional="False" Category="Data" Description="XML file to
iterate." %>
  <%@ Assembly Name="System.Xml" %>
  <%@ Import Namespace="System.Xml" %>
  Top-level nodes:
  <% Dim currNode as XmlNode
  currNode = TargetFile.DocumentElement.FirstChild
  Do Until currNode Is Nothing%>
    <%= currNode.InnerXml %>
  <% currNode = currNode.NextSibling()
  Loop %>
```

概莫版对目标文件的属性并没有定义一个 Schema，所以属性在模版中是作为一个 XMLDocument。如果我们选择的 XML 文件如下所示：

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <Books>
3   <Book>UML 2.0 In a Nutshell</Book>
4   <Book>The Best Software Writing</Book>
5   <Book>Coder to Developer</Book>
6   <Book>Code Complete</Book>
7 </Books>
```

生成后的代码：

```
Top-level nodes:

UML 2.0 In a Nutshell

The Best Software Writing

Coder to Developer

Code Complete
```

CodeSmith 实用技巧（七）：从父模版拷贝属性

在使用 **CodeSmith** 进行代码生成的时候，你可能需要在子模版和父模版之间共享属性。比如，写一个基于数据库生成代码的模版，在每个模版里面都定义了一个名为 **Server** 的属性。当你在父模版中使用此属性时，它的值只对父模版起作用。想要设置此值到子模版，可以在父模版中使

用 CopyPropertiesTo 方法，当在父模版中使用此属性时，它的值会发送到子模版中去。下面这段代码展示了如何使用该方法：

```
   // instantiate the sub-template
    Header header = new Header();
    // copy all properties with matching name and type to the sub-te
mplate instance
    this.CopyPropertiesTo(header);
```

CodeSmith 实用技巧（八）：生成的代码输出到文件中

在 CodeSmith 中，要把生成的代码文件输出到文件中，你需要在自己的模版中继承 OutputFileCodeTemplate 类。

```
<%@ CodeTemplate Language="C#" TargetLanguage="C#" Inherits="OutputFi
leCodeTemplate" Description="Build custom access code."%>
 <%@ Assembly Name="CodeSmith.BaseTemplates" %>
```

OutputFileCodeTemplate 主要做两件事情：

1. 它添加一个名为 OutputFile 的属性到你的模版中，该属性要求你必须选择一个文件；

2. 模版重载了方法 OnPostRender()，在 CodeSmith 生成代码完成后把相应的内容写入到指定的文件中去。

如果想要自定义 OutputFile 属性弹出的保存文件对话框，你需要在你的模版中重载 OutputFile 属性。例如：你希望用户选择一个.cs 文件来保存生成的代码，需要在你的模版中添加如下代码：

```
 <script runat="template">


// Override the OutputFile property and assign our specific settings
to it.


[FileDialog(FileDialogType.Save, Title="Select Output File", Filter="
C# Files (*.cs)|*.cs", DefaultExtension=".cs")]
 publicoverride string OutputFile
{
    get {return base.OutputFile;}
    set {base.OutputFile = value;}
}
 </script>
```

CodeSmith 实用技巧（九）：重载 Render 方法来控制输出

在 **CodeSmith** 中，CodeTemplate.Render 方法是在模版执行完成进行模版输出时执行，你可以通过重载 CodeTemplate.Render 方法来修改 CodeSmith 输出时的事件处理。例如：你

可以修改模版输出时的方式来代替现在默认的方式，下面这段代码展示了在保持 CodeSmith 默认的窗口显示的同时，把结果输出到两个不同的文件。

```
 1
<%@ CodeTemplate Language="C#" TargetLanguage="Text" Description="Add
TextWriter Demonstration."%>
 2 <%@ Import Namespace="System.IO" %>
 3 This template demonstrates using the AddTextWriter method
 4  to output the template results to multiple locations concurrently.
 5 <script runat="template">
 6 public override void Render(TextWriter writer)
 7{
 8    StreamWriter fileWriter1 = new StreamWriter(@"C:\test1.txt",
true);
 9     this.Response.AddTextWriter(fileWriter1);
10
11    StreamWriter fileWriter2 = new StreamWriter(@"C:\test2.txt",
true);
12     this.Response.AddTextWriter(fileWriter2);
13
14    base.Render(writer);
15
16    fileWriter1.Close();
```

```
17      fileWriter2.Close();
18  }
19  </script>
```

注意不能忘了 base.Render(writer);这句话，否则你将不能获得默认的输出。当重载 CodeTemplate.Render 方法时，你也可

以访问 **TextWriter**，也就是说你也可以直接添加其它的附属信息到模版输出的内容中。

CodeSmith 实用技巧（十）：通过编程执行模版

CodeSmith 在执行模版时通过调用一些 API 来完成的，主要经过了以下这几步的操作：

- 编译一个模版
- 显示编译错误信息
- 创建一个新的模版实例
- 用元数据填充模版
- 输出结果

下面这段代码显示了这些操作：

```
CodeTemplateCompiler compiler = new CodeTemplateCompiler("..\\..\\StoredProcedures.cst");
compiler.Compile();
```

```
 if (compiler.Errors.Count == 0)
{
    CodeTemplate template = compiler.CreateInstance();

    DatabaseSchema database = new DatabaseSchema(new SqlSchemaProvider(), @"Server=(local)\
NetSDK;Database=Northwind;Integrated Security=true;");
    TableSchema table = database.Tables["Customers"];

    template.SetProperty("SourceTable", table);
    template.SetProperty("IncludeDrop", false);
    template.SetProperty("InsertPrefix", "Insert");

    template.Render(Console.Out);
}
 else
{
    for (int i = 0; i < compiler.Errors.Count; i++)
    {
        Console.Error.WriteLine(compiler.Errors[i].ToString());
    }
}
```

在这里我们用了 Render 方法，其实 CodeTemplate.RenderToFile 和 CodeTemplate.RenderToString 方法可能更有用，它可以直接让结果输出到文件中或赋给字符型的变量。

注意：该功能只能在 CodeSmith 专业版中使用

CodeSmith 实用技巧（十一）：添加设计器的支持

如果你需要提供一个复杂的组合用户界面来输入元数据,这时就要添加设计器的支持。换句话说，除此之外没有别的办法来输入你自定义的元数据类型。添加设计器的支持，首先你要创建一个 Editor 作为自定义的类型，一个 Editor 其实就一个继承于.NET 中的 System.Drawing.Design.UITypeEditor 类的子类。

安装 CodeSmith 后在，在 C:\Program File\CodeSmith\ SampleProjects 文件夹下有很多 SampleCustomProperties 的工程。例如：DropDownEditorProperty 是一个把字符串和布尔类型的值结合在一起的元数据，它提供了一个类 DropDownEditorPropertyEditor 继承于 System.Drawing.Design.UITypeEditor。

```
[Editor(typeof(CodeSmith.Samples.DropDownEditorPropertyEditor), typeo
```

```
f(System.Drawing.Design.UITypeEditor))]
 publicclass DropDownEditorProperty
```

在使用的时候跟其它的元数据类型是一样，不过别忘记添加对程序集的引用，引用 CodeSmith 默认的是不认识该类型的。

```
<%@ Property Name="DropDownEditorProperty" Type="CodeSmith.Samples.Dr
opDownEditorProperty" Category="Options" Description="This property u
ses a custom dropdown editor."%>
 <%@ Assembly Name="SampleCustomProperties" %>
```

当用户想要编辑 DropDownEditProperty 时，单击 CodeSmith 属性面板将会显示如下的自定义对话框：

CodeSmith 实用技巧（十二）：自动执行 SQL 脚本

在 CodeSmith 中，如果生成的代码是 SQL 脚本，则可以在生成代码完成时自动执行生成的代码，也就是在生成的 SQL 脚本的同时在数据库中创建新的对象。

用 `BaseTemplates.ScriptUtility` 对象提供 `ExecuteScript` 方法可以实现，如果想在生成代码完成后立即执行生成的脚本，可以很方便的通过重载 `OnPostRender` 来实现。

在使用之前，先添加对下列程序集的引用：

```
<%@ Assembly Name="CodeSmith.BaseTemplates"%>

<%@ Import Namespace="CodeSmith.BaseTemplates" %>
```

看下面的这个例子：

```
protected override void OnPostRender(string result)
{
    // execute the output on the same database as the source table.
    CodeSmith.BaseTemplates.ScriptResult scriptResult =
     CodeSmith.BaseTemplates.ScriptUtility.ExecuteScript(this.SourceTable.Database.ConnectionStri
ng,
     result, new System.Data.SqlClient.SqlInfoMessageEventHandler(cn_InfoMessage));
    Trace.Write(scriptResult.ToString());
    base.OnPostRender(result);
}
```

在这个例子中 `SourceTable` 是一个类型为 `SchemaExplorer.TableSchema.`的属性，使用的时候需要调整部分代码来获取数据库的连接以便在生成代码完成后执行脚本。

CodeSmith 实用技巧（十三）：使用 CodeTemplateInfo 对象

在 `CodeSmith` 中使用 `CodeTemplateInfo` 可以获取当前模版的一些信息：

属性	返回值
`CodeBehind`	**Gets the full path to the code-behind file for the template (or an empty string if there is no code-behind file).**

ContentHashCode	**Gets the hash code based on the template content and all template dependencies.**
DateCreated	**Gets the date the template was created.**
DateModified	**Gets the date the template was modified.**
Description	**Gets the description.**
DirectoryName	**Gets the name of the directory the template is located in.**
FileName	**Gets the name of the template file.**
FullPath	**Gets the full path to the template.**
Language	**Gets the template language.**
TargetLanguage	**Gets the target language.**

看一下一个具体的使用例子：

```
<%@ CodeTemplate Language="VB" TargetLanguage="Text" Description="Demonstrates CodeTemplateInfo." %>
  <% DumpInfo() %>
  <script runat="template">
  Public Sub DumpInfo()
```

```
    Response.WriteLine("Template:       {0}", Me.CodeTemplateInfo.FileName)
    Response.WriteLine("Created:        {0}", Me.CodeTemplateInfo.DateCreated)
    Response.WriteLine("Description:    {0}", Me.CodeTemplateInfo.Description)
    Response.WriteLine("Location:       {0}", Me.CodeTemplateInfo.FullPath)
    Response.WriteLine("Language:       {0}", Me.CodeTemplateInfo.Language)
    Response.WriteLine("Target Language: {0}", Me.CodeTemplateInfo.TargetLanguage)
  End Sub
  </script>
```

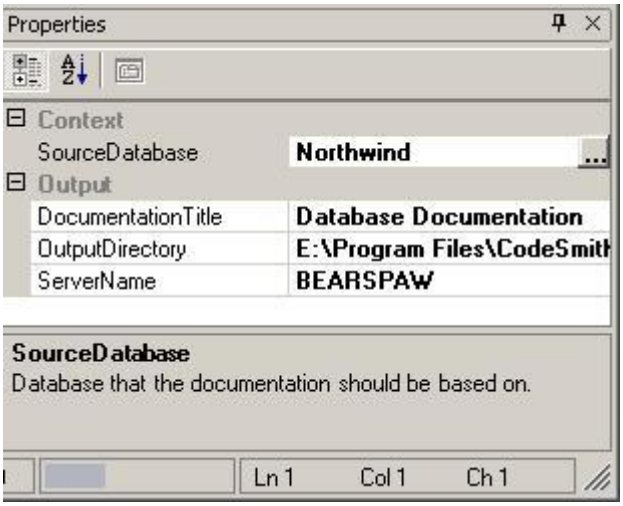
执行该模版输出如下（环境不同，输出也不同）：

```
Template:       CodeTemplateInfo.cst
Created:        6/29/20058:54:19 PM
Description:    Demonstrates CodeTemplateInfo.

Location:       C:\Program Files\CodeSmith\v3.0\SampleTemplates\Test\CodeTe
mplateInfo.cst
Language:       VB
Target Language: Text
```

CodeSmith 实用技巧（十四）：使用 Progress 对象

Progress 对象可以在 CodeSmith 生成代码时给用户显示一个进度条，当生成代码的时间很长时非常有用。如果你使用的是 **CodeSmith Explorer**，进度条将显示在 **Generate** 按钮的左边：

如果使用的是 **CodeSmith Studio**，进度条将显示在状态栏上：



使用 Progress 和在 WinForm 中使用进度条差不多，需要设置它的最大值和步长：

```
this.Progress.MaximumValue =25;

this.Progress.Step = 1;
```

如果想显示出进度，需要调用 PerformStep 方法：

```
this.Progress.PerformStep();
```

CodeSmith 实用技巧（十五）：使用快捷键

在 CodeSmith 中，以下几个快捷键有助于我们快速输入。

1. Ctrl + Shift + C

在空行上，按下 Ctrl + Shift + C 后将会录入一个代码块。

```
<%  %>
```

2. Ctrl + Shift + Q

按下 Ctrl + Shift + Q 后录入一个脚本块。

```
<script runat="template">


</script>
```

3. Ctrl + Shift + V

对代码块反转，如有下面这样一行代码：

```
<%for(int i=0;i<10;i++){}%>
```

在两个大括号之间按下 Ctrl + Shift + V 后，将变成如下代码：

```
<%for(int i=0;i<10;i++){%>  <%}%>
```

4. Ctrl + Shift + W

按下 Ctrl + Shift + W 后会录入一个输出的代码块：

```
<%=  %>
```

注意：在使用快捷键的时候，如果想要把一段代码之间放在录入的标记中间，首先选中这些代码，再按下快捷键组合。比如我们有一段这样的代码，想把它放在<script>里面。

```
public enum CollectionTypeEnum

{

    Vector,

    HashTable,

    SortedList
```

```
}

public override void Render(TextWriter writer)

{

    StreamWriter fileWriter1 = new StreamWriter(@"C:\test1.cs", true);

    this.Response.AddTextWriter(fileWriter1);



    StreamWriter fileWriter2 = new StreamWriter(@"C:\test2.cs", true);

     this.Response.AddTextWriter(fileWriter2);



    base.Render(writer);


```

```
    fileWriter1.Close();

    fileWriter2.Close();

}
```

选中它，再按下 Ctrl + Shift + Q 后就会变成：

```
<script runat="template">

publicenum CollectionTypeEnum

{

  Vector,



  HashTable,


```

```
SortedList

}

public override void Render(TextWriter writer)

{

    StreamWriter fileWriter1 = new StreamWriter(@"C:\test1.cs", true);

    this.Response.AddTextWriter(fileWriter1);



    StreamWriter fileWriter2 = new StreamWriter(@"C:\test2.cs", true);

     this.Response.AddTextWriter(fileWriter2);



    base.Render(writer);
```

```
    fileWriter1.Close();

    fileWriter2.Close();

}


 </script>
```

CodeSmith 的基础模版类

基础模版类

类型描述：

Batch

OutputFileCodeTemplate 模版通过继承此类能够在生成过程中把他们的输出保存到文件中

ScriptError 在脚本执行中出现一个错误

ScriptErrorCollection

ScriptResult 一个脚本的运行结果包含一些已经发生的错误

ScriptUtility 这个类能用来在数据库上执行 **Sql** 脚本。

SqlCodeTemplate 继承此类的模版当从一个 **Sql** 数据源生成代码时能够获得很多有用的帮

助方法

StringUtility 多种处理 **string** 型的方法

各类型下的成员属性及方法

Batch Class

属性

Content

LineCount

StartLineNumber

方法

Finalize 在一个对象再次创建之前获得空闲资源并且执行其他的清空操作

MemberwiseClone 建立现有对象的副本

OutputFileCodeTemplate Class

属性

CodeTemplateInfo 得到当前模版的信息

OutputFile 此属性用来指定一个保存模版输出的输出文件名

Progress 提供一种方式汇报模版的执行进程

Response 模版输出返回流。此属性可以在程序中写出流

State 模版实例的状态

ValidationErrors 得到模版的错误

方法

CopyPropertiesTo 把匹配的属性拷贝到另一个代码模版实例中

GetCodeTemplateInstance 重载，得到指定模版的实例

GetFileName 为模版的输出得到一个默认的名字

GetProperties 得到模版的所有属性

GetProperty 得到模版的指定属性

GetRequiredProperties 得到模版上所有必要的属性

GetType 得到当前实例类型

ParseDefaultValue 解析属性的默认值

SavePropertiesToXml 以 **XML** 保存属性

SavePropertiesToXmlFile 保存属性到一个 **XML** 文档

SetProperty 重载，保存指定的属性值

ToString

ScriptError Class

属性

方法

Finalize 在一个对象再次创建之前获得空闲资源并且执行其他的清空操作

MemberwiseClone 建立现有对象的副本

ScriptUtility Class

属性

ConnectionString 执行脚本时使用此连接字符串

Script 执行的脚本

方法

ExecuteScript 重载，执行脚本

SqlCodeTemplate Class

属性

CodeTemplateInfo 得到当前模版的信息

OutputFile 此属性用来指定一个保存模版输出的输出文件名

Progress 提供一种方式汇报模版的执行进程

Response 模版输出返回流。此属性可以在程序中写出流

State 模版实例的状态

ValidationErrors 得到模版的错误

方法

CopyPropertiesTo 把匹配的属性拷贝到另一个代码模版实例中

GetCamelCaseName Returns a camel cased name from the given identifier.

GetCodeTemplateInstance 重载，得到指定模版的实例

GetCSharpVariableType 基于给定列返回 C#的变量类型

GetFileName 为模版的输出得到一个默认的名字

GetMemberVariableDeclarationStatement

重载，返回 C#成员变量声明语句

GetMemberVariableDefaultValue

基于一个列的数据类型返回一个默认值

GetMemberVariableName 为一个给定标示返回一个 C#成员变量名

GetProperties 得到模版的所有属性

GetProperty　得到模版的指定属性

GetPropertyName　返回指定列的公有属性的名字

GetReaderMethod　Returns the name of the typed reader method for a given column.

GetRequiredProperties　得到模版上所有必要的属性

GetSpacedName　Returns a spaced out version of the identifier.

GetSqlDbType　返回一个给定列的 SqlDbType

GetSqlParameterExtraParams

为 ADO 的参数声明生成额外的参数

GetSqlParameterStatement

重载，返回给定列的 T-Sql 的参数声明

GetSqlParameterStatements

重载，给指定列加一个参数到 ADO 对象生成一个指定声明（Generates an assignment statement that adds a parameter to a ADO object for the given column. ）

GetValidateStatements 基于某列生成一组确认声明

IncludeEmptyCheck 确定一个给定列是否可以为空

IncludeMaxLengthCheck 确定一个给定列的类型是否需要最大长度的定义

IsUserDefinedType 确定是否一个给定列用了一个 UDT（用户定义类型）

ParseDefaultValue　解析属性的默认值

SavePropertiesToXml　以 XML 保存属性

SavePropertiesToXmlFile 保存属性到一个 XML 文档

SetProperty　重载，保存指定的属性值