CodeSmith学习手册

1.使用简介

下载：

官方网站：<http://www.codesmithtools.com/>

5.0破解文件下载：

<http://kewlshare.com/dl/0538fcf454d3/CodeSmith.5.0.Professional.Incl.Patch.DTCG.rar.html>

CodeSmith生成代码的基础为数据库，生成代码依赖于你编写的模板。

模板分成三部分：声明，输出，函数。

声明部分包括

CodeTemplate，Property，Assembly，Import，Register，Map，XmlProperty。

CodeTemplate声明：

CodeTemplate声明的属性		
Language	语言	编写输出和函数部分使用的语言
TargetLanguage	目标语言	生成代码的语言
Src	源文件	模板中使用的类文件名称
Inherits	继承的类	模板继承自的类名
Debug	调试	是否需要调试
Description	描述	简介
ResponseEncoding	输出编码格式	设置输出内容的编码格式

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。

CodeTemplate声明为必选声明。

Language：一般可以使用C#，VB，JScript等语言，此处要慎重，尽量选择熟悉的语言。

TargetLanguage：通过模板生成的代码的语言，一般来说，无关紧要，但还是尽量实事求是的写吧，因为CodeSmith需要这个属性来给代码着色。

Src和Inherits：这两个Src写的是源文件名称路径，Inherits写的是继承的父类名称。这两个属性可以不设置。

Debug：可选True或False，用来决定是否调试，简单模板一般设置为False。

ResponseEncoding：如果要让你的模板生成的内容中包括中文，那么设置这个属性为UTF-8。

例：

```
<%@ CodeTemplate Language="C#" TargetLanguage="C#" Src="" Inherits="" Debug="False" Description="产生实体类" ResponseEncoding="UTF-8" %>
```

Property声明：

属性被用做在模板运行时声明一个使用的参数		
Name	名称	模版使用的参数的名称,类似变量名称

Type	类型	类似变量类型，System.String，String都行
Default	默认值	默认值
Optional	是否可选	Bool类型，True：可选，False：必选
Category	分组名称	
Description	描述	在属性面板中对于这个属性的描述

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。

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。

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

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：The serializer parameter specifies the IPropertySerializer type to use when serializing the properties values. This is equivalent to using a ....

例：

```
<%@ Property Name="NameSpace" Default="BLL" Type="String" Category="Context"
Description="业务层" %>

<%@ Property Name="SourceTable" Type="SchemaExplorer.TableSchema" Category="Context"
Description="表名" %>
```

Assembly声明：

Assembly声明属性，Assembley声明类似.Net工程中的引用程序集。		
Name	名称	程序集名称，不包括.dll后缀

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。

例：<%@ Assembly Name="SchemaExplorer" %>//加载使用访问数据库的组件SchemaExplorer

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。

```
<%@ 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

Import声明：

Import声明属性，类似.Net工程中的使用命名空间，类似于using System		
Name	名称	命名空间

例：<%@ Import Namespace="SchemaExplorer" %>//声明其使用的命名空间

Register声明：

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

Name	名称	将模板作为一个类的别名使用
Template	模板名称	模板文件的全路径

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。

例：<%@ Register Name="MySubTemplate" Template="MySubTemplate.cst" MergeProperties="True" ExcludeProperties="SomeExcludedPropertyName,SomeProperties*" %>

XML属性声明（XmlProperty Directive）

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。

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

输出部分

这个部分的内容最终会到达最终生成的源代码文件，所以在编写这部分内容的时候，一定要对将要生成的内容的结构十分熟悉和了解。

输出部分可以划分为两部分：

1. 固定内容：指的是没有在<%%>中间包括的内容。必输出

2. 可变内容：在<%%>和<%=%>中间包括的内容。根据逻辑输出

在<%%>中间可以填写的内容为：在模板声明部分的CodeTemplate声明中Language属性所指的语言编写的任何内容，可以定义变量，常量，可以使用各种控制结构，几乎是任何语句都可以，因为CodeSmith毕竟是架构在.Net体系结构中的。

例子：

```
<%@ CodeTemplate Language="C#" TargetLanguage="Text" Src="" Inherits="" Debug="False" Description="Template description here." %>
<%@ Property Name="TestString" Type="System.String" Default="SomeValue" Optional="True" Category="Strings" Description="This is a sample string property." %>
<%for(int i=0;i<10;i++)
{
 Response.WriteLine("第"+i.ToString()+"次："+TestString);
}
%>
```

第0次：aaaaa
第1次：aaaaa

第2次：aaaaa
第3次：aaaaa
第4次：aaaaa
第5次：aaaaa
第6次：aaaaa
第7次：aaaaa
第8次：aaaaa
第9次：aaaaa

对象借用了Response的壳子，它的魂却是控制台应用程序中常用的输出输入对象Console。

```
<%for(int i=0;i<10;i++)%>
<%{%>
第<%Response.Write(i);%>次：<%Response.Write(TestString);%>
<%}%>
```

<%=变量名称%>或<%=常量名称%>（注意此处的<%=%>中没有“分号”）

函数部分

这部分可以放一些在输出部分使用的函数，函数当中也可以使用在声明部分定义的参数。
函数部分必须包括在<script runat="template"><script>

```
public int Sum(int x,int y)
{
   return x+y;
}
```

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。

其它部分

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：

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。

1.对象和控制台

CodeSmith对象

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。
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 说明信息
DirectoryName 全路径加文件加
FileName 模板文件名
FullPath 全路径
Language 语言
TargetLanguage 生成代码的语言
例：Response.WriteLine(this.CodeTemplateInfo.FileName) CodeTemplateInfo.cst

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。
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：

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.）
例：
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();

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：

```
<%for(int i = 0;i<List.Count;i++){%>
    Console.WriteLine(<%=List[i]%>);
 <%}%>
```

公共属性

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

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。

```
1<?xml version="1.0" encoding="us-ascii"?>
2<codeSmith>
3        <propertySet>
4                <property name="Accessibility">Public</property>
5                <property name="ClassName">PersonArray</property>
6                <property name="ItemType">Person</property>
7                <property name="ItemValueType">False</property>
8                <property name="ItemCustomSearch">False</property>
9                <property name="KeyName">PersonID</property>
10               <property name="KeyType">int</property>
11               <property name="IncludeInterfaces">True</property>
12               <property name="IncludeNamespaces">False</property>
13       </propertySet>
14</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
{
    GeneratedOrderEntity
     #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\BusinessObject.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。

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。

编写CodeSmith自定义属性的编辑器

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。

```
 1<%@ CodeTemplate Language="C#" TargetLanguage="C#" Description="Builds a class for each class in the assembly, and a test stub for every method." %>
 2
 3<%@ Import NameSpace="System.Reflection" %>
 4
 5<script runat="template">
 6
 7private Assembly assembly;
 8
 9public Assembly AssemblyToLoad
10{
11      get{return assembly;}
```

```
12      set{assembly = value;}
13}
14
15</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。

```
 1using System;
 2using NUnit.Framework;
 3
 4<%
 5     foreach(Type T in AssemblyToLoad.GetTypes())
 6     {
 7          if(T.IsClass)
 8          {
 9               %>
10
11               [TestFixture]
12               public class <%=T.Name%>Tests
13               {
14               <%
15                         MethodInfo[] methods = T.GetMethods ( BindingFlags.Public |
BindingFlags.Instance | BindingFlags.Static  );
16                         foreach(MethodInfo M in methods)
17                         {
18                              %>
19
20                              [Test]
21                              public void <%=M.Name%>Test
22                              {
23                                   //TODO Write this test
24                              }
25                              <%
26                         }
27
28               %>}<%
29          }
30     }
31%>
```

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，我们是用Visual Studio创建这个类。

/Files/Bear-Study-Hard/AssemblyHelper.zip

首先我们需要创建一个继承UITypeEditor的类。

```
1public class AssemblyFilePicker : UITypeEditor
2{
3     public AssemblyFilePicker(): base()
4     {
5     }
6}
```

关于UITypeEditor的说明请大家参看MSDN或Visual Studio.NET自带帮助中的说明，其中有详细的例子。

然后我们需要重载UITypeEditor类的两个不同的方法。第一个需要重载点的方法是GetEditStyle，这个方法是告诉属性面板对于当前类型是用什么类型的编辑器，在这个例子中我们设置编辑类型为Modal。这样大家可以在该属性格子的右边看到一个小按钮，它将引发一个对话框等模式的对话（trigger a modal dialog）。这是我们的GetEditStyle方法：

```
1public override UITypeEditorEditStyle GetEditStyle(ITypeDescriptorContext context)
2{
3     return UITypeEditorEditStyle.Modal;
4}
```

其中的Modal为显示一个省略号按钮。

需要重载的另一个方法是EditValue方法，当用户电击属性时会调用这个方法。按照我们需要加载的组件类型需要创建一个打开文件对话框（open file dialog）然后捕获这个对话框，在属性格子中返回对话框的结果。

```
1public override object EditValue(ITypeDescriptorContext context, IServiceProvider provider,
object value)
2{
3
4if (provider != null)
5{
```

首先我们要从当前的服务和控件中得到一个参考，有了控件的参考我们可以通过它转到ShowDialog方法。（原文：First we need to get a reference to the current service and control, we need the reference to the control so we can pass it to the ShowDialog method.）

```
1IWindowsFormsEditorService editorService = (IWindowsFormsEditorService)provider.GetService
(typeof(IWindowsFormsEditorService));
2Control editorControl = editorService as Control;
3
4if (editorControl != null)
5{
```

然后我们创建一个openFileDialog类并填入适合的属性。

```
1OpenFileDialog openFileDialog = new OpenFileDialog();
2
```

```
3openFileDialog.CheckFileExists = true;
4openFileDialog.DefaultExt = ".dll";
5openFileDialog.Multiselect = false;
6openFileDialog.Title = "Select an Assembly:";
7openFileDialog.Filter = "Assembly Files | *.dll";
```

然后我们通过控件的参考（reference）将对话框显示给用户。

```
1DialogResult result = openFileDialog.ShowDialog(editorControl);
```

下一步我们检查用户是否点击了OK按钮，如果点击了，通过文件选择对话框选择文件后使用LoadForm方法加载这个组件，最后返回这个值。

```
 1if (result == DialogResult.OK)
 2        {
 3Assembly assembly = Assembly.LoadFrom( openFileDialog.FileName ) ;
 4            value = assembly;
 5        }
 6        else
 7        {
 8            value = null;
 9        }
10    }
11}
12
13return value;
14}
```

这个值将被放在属性面板中并可以被模板读取，但是需要注意，在我们作这个之前要将组件import引入到模板中，并在模板中用一对属性声明。

加载这个模板我们仅需将这个组件assembly与模板放在同一目录下，然后再模板中加入下面两行代码。

```
1<%@ Assembly Name="AssemblyHelper" %>
2<%@ Import NameSpace="AssemblyHelper" %>
```

2.与数据库联系

CodeSmith与数据库的联系，在CodeSmith中自带一个程序集SchemaExplorer.dll，这个程序集中的类主要用于获取数据库中各种对象的结构。

```
<%@ Property Name="SourceTable" Type="SchemaExplorer.TableSchema" Default=""
Optional="False" Category="Context" Description="源表名" %>
<%@ Property Name="SourceDB" Type="SchemaExplorer.DatabaseSchema" Default=""
Optional="False" Category="Context" Description="" OnChanged="" Editor="" EditorBase=""
Serializer="" %>
<%@ Assembly Name="SchemaExplorer" %>
<%@ Import Namespace="SchemaExplorer" %>
```

SchemaExplorer中主要类的结构和功能：

DatabaseSchema

属性：
ConnectionString:一般填写类似于ADO.NET的连接字符串
Name:数据库名称
Provider:驱动程序提供者，一般实例化一个SqlSchemaProvider对象
集合：
Commands:所有存储过程集合
Tables:所有表的集合
Views:所有视图的集合

TableSchema

属性：
Name:表名
Database:所在数据库
DataCreated:创建日期
FullName:全名
HasPrimaryKey:是否有主键
Owner:所有者
PrimaryKey:主键信息

方法：GetTableData:获取表中所有数据，结果为DataTable

集合：
Columns:所有列集合
ForeignKeyColumns:所有外键列的集合
ForeignKeys:外键信息的集合
Indexes:所有索引的信合
Keys:所有主键和外键的列的集合
NonKeyColumns:所有非主键外键列的集合
NonForeignKeyColumns:所有非外键列的集合
NonPrimaryKeyColumns:所有非主键列的集合
PrimaryKeys:主键信息的集合

CommandSchema存储过程结构

属性：
Name:存储过程名
FullName:全名
Database:所在数据库
DataCreated:创建日期
Owner:所有者
ReturnValueParameter:返回值参数，SQLSERVER中似乎取不到
CommandText:存储过程的内容源代码

集合：
AllInputParameters:所有的输入参数集合，有可能包括即是输入又是输出的参数
AllOutputParameters:所有输出参数的集合，有可能包括即是输入又是输出的参数
CommandResults:存储过程的查询结果集合
InputOutputParameters:输入输出参数的集合
InputParameters:所有输入参数的集合
OutputParameters:所有输出参数的集合
Parameters:所有参数的集合
NonReturnValueParameters:除了返回值之外的参数的集合

ViewSchema视图结构
属性：
Name:视图名
FullName:视图全名
Database:所在数据库
DataCreated:创建日期
Owner:所有者
ViewText:视图源代码

方法：GetViewData:得到视图中的数据，返回类型为DataTable

集合：Columns:视图中所有列的集合

ColumnSchema列结构
属性：
Name:列的名称
NonDBNull:是否允许为空－－－AllowDBNull
Database:所在数据库
DataType:内部表示的数据类型
IsForeignKeyMember:是否为外键
IsPrimaryKeyMember:是否为主键，通用
IsUnique:是否唯一
NativeType:数据库中的数据类型
Precision:精度
Scale:小数位数
Size:列的长度
SystemType:当前列在所用语言中的类型
Table:所在的表

ParameterSchema参数结构
属性：
Name:参数名称
NonDBNull:是否为空
Command:所在存储过程名
Database:所在数据库
Direction:参数的类型：输入，输出，输入输出，返回值
NativeType:数据库中的数据类型
Size:长度
Precision:精度
Scale:小数位数
SystemType:当前列在所用语言中的类型

ViewColumnSchema视图列的结构
属性：
Name:视图的名称
NonDBNull:是否为空
View:所在的视图
Database:所在的数据库
NativeType:SqlServer中的类型
Size:长度
Precision:精度
Scale:小数位数
SystemType:当前列在所用语言中的类型

TableKeySchema表中键结构

属性：

Name:表的键的名称，即约束名称
Database:所在数据库
ForeignKeyTable:有外键的表，即子表
PrimaryKeyTable:主键表，即主表
PrimaryKey:主表的主键信息

集合：

PrimaryKeyMemberColumns:当前键信息中主键的成员列集合，即主表中的主键的列的集合
ForeignKeyMemberColumns:当前键信息中外键的成员列集合，即子表中某个外键的列集合

IndexSchema索引的结构

属性：

Name:索引名称
Table:所在表
DataBase:所在数据库
Is:是否聚集索引
IsPrimaryKey:是否为主键索引
IsUnique:是否为唯一索引

集合：MemberColumns:索引的列集合

ExtendedProperty扩展信息

Table:
CS_isIdentity:是否为标识符，不支持Access
CS_isComputed:是否为计算列
CS_isDeterministic:是否确定...
CS_IdentitySeed::标识列种子数
CS_IdentityIncrement:标识列递增量
CS_Default:列的默认值
CS_isRowGuidCol

View:
CS_isComputed:是否为计算列
CS_isDeterministic:

Command:
CS_Default:存储过程的默认参数

3.创建实体类

利用SchemaExplorer中的类：TableSchema，ColumnSchema来制造一个实体类。

首先我们得了解实体类的构成，一般的实体类由私有字段和公开方法组成，例如下面的类：

```
using System;
using System.Collections.Generic;
using System.Text;

namespace com.xaccp.Entity
```

```
{
    public class Person
    {
        private string name;
        private bool sex;
        private int age;
        public string Name
        {
            get { return name; }
            set { name = value; }
        }
, , , , ,
    }
}
```

实体类中的字段应该和数据库中的字段相同，只是类型要从Sql中的数据类型变为C#中的数据类型，一般实体类中的字段的名字可以采用与数据库中字段的名称相同，只不过字段名一般采用骆驼命名法，属性名称一般采用Pascal命名法。

私有字段部分有些内容是永远不变的，字段的访问修饰符永远为"private"，字段的结尾永远为";"；变化的部分为，每个字段根据数据库中的类型变化为C#中的类型，字段的名称也是变化的。

公开字段中也有内容为永远不变的

```
public 数据类型 属性名称
        {
            get { return 字段名称; }
            set { 字段名称 = value; }
        }
```

第一个模板

```
<%@ CodeTemplate Language="C#" TargetLanguage="C#" Src="" Inherits="" Debug="False"
Description="产生实体类" ResponseEncoding="UTF-8" %>
<%@ Property Name="SourceTable" Type="SchemaExplorer.TableSchema" Default=""
Optional="False" Category="Context" Description="源表名" %>
<%@ Assembly Name="SchemaExplorer" %>
<%@ Import Namespace="SchemaExplorer" %>
<%@ Assembly Name="System.Data" %>
<%@ Import Namespace="System.Data" %>

using System;
namespace com.xaccp.Entity
{
    public class Person
    {
        //这里的内容为私有字段和公开属性。
```

```
<%foreach(ColumnSchema column in this.SourceTable.Columns)%>
<%{%>
private <%=column.SystemType%> <%=column.Name%>;
<%}%>
<%foreach(ColumnSchema column in this.SourceTable.Columns)%>
<%{%>
public <%=column.SystemType%> <%=column.Name%>
{
 get{return <%=column.Name%>;}
 set{<%=column.Name%>=value;}
}
<%}%>
  }
}
```

所产生的实体类问题?

第一：所产生的这个类，根本编译不过去。

第二：这个类的类名永远是Person，即使产生了这个类，还得修改。

第三：命名空间也要改。

第四：.....

错误提示在title_id这个属性那里。原来是字段名称和属性名称冲突了，正确的应该是字段名称使用骆驼命名法，而属性用Pascal命名法。只需要将字段的第一个字母改为小写，属性的第一个字母改为大写。

写一个公共函数

```
//以骆驼命名法格式化字符串
public string ToCamel(string s)
{
    return s.Substring(0,1).ToLower()+s.Substring(1);
}

//以Pascal命名法格式化字符串
public string ToPascal(string s)
{
    return s.Substring(0,1).ToUpper()+s.Substring(1);
}
```

更改输出的代码段：

```
//字段部分
private <%=column.SystemType%> <%=ToCamel(column.Name)%>;
//属性部分
public <%=column.SystemType%> <%=ToPascal(column.Name)%>
{
    get{return <%=ToCamel(column.Name)%>;}
    get{<%=ToCamel(column.Name)%>=value;}
```

```
}
```

类名变化

最好让类名和表名有关系。表名能不能来做类名呢？因为表名通常为复数形式，能不能改成单数啊？只要去掉表名后的"s"

```
public string GetClassName(TableSchema table)
{
    string s=table.Name;
    if(s.Substring(s.Length-1)=="s")
    {
        return ToPascal(s.Substring(0,s.Length-1));
    }
    return ToPascal(s);
}

class <%=GetClassName(this.SourceTable)%>
```

命名空间

一般情况下，都将命名空间设置为Model或者Entity，可是到底选择哪一个什么？

在声明部分添加：

```
<%@ Property Name="NameSpace" Type="System.String" Default="Model" Optional="False"
Category="" Description="" Editor="" EditorBase="" Serializer="" %>
namespace <%=NameSpace%>
```

类型问题

```
private <%=column.SystemType%> <%=ToCamel(column.Name)%>;
```

"System.String"和"string"都是一回事，我们写的程序毕竟不是用MSIL语言写的，是用C#，最好别用CTS中的数据类型？

解决的函数如下：

```
public string GetCSDataType(ColumnSchema column)
{
    switch (column.DataType)
    {
        case DbType.AnsiString: return "string";
        case DbType.AnsiStringFixedLength: return "string";
        case DbType.Binary: return "byte[]";
        case DbType.Boolean: return "bool";
        case DbType.Byte: return "byte";
        case DbType.Currency: return "decimal";
        case DbType.Date: return "DateTime";
        case DbType.DateTime: return "DateTime";
        case DbType.Decimal: return "decimal";
        case DbType.Double: return "double";
        case DbType.Guid: return "Guid";
        case DbType.Int16: return "short";
```

```
            case DbType.Int32: return "int";
            case DbType.Int64: return "long";
            case DbType.Object: return "object";
            case DbType.SByte: return "sbyte";
            case DbType.Single: return "float";
            case DbType.String: return "string";
            case DbType.StringFixedLength: return "string";
            case DbType.Time: return "TimeSpan";
            case DbType.UInt16: return "ushort";
            case DbType.UInt32: return "uint";
            case DbType.UInt64: return "ulong";
            case DbType.VarNumeric: return "decimal";
            default:
            {
                return "__UNKNOWN__" + column.NativeType;
            }
        }
}
```

无论是从列结构上获取的SystemType还是DataType属性，都是用来表达这个列的数据类型的，尽管他们分别是CTS和SQL中的数据类型，从CTS转换和从SQL转换，有什么本质差别不？可是DataType是个枚举类型的属性，如果在VS.Net2005中，一旦switch一个枚举变量，可以自动生成这个枚举类型中所有的枚举值的case分支，这样，咱们只需要填写case的结构就可以了。

4.生成其它文件

下面我们以一种非传统编程语言来演示CodeSmith生成内容的多样性。

一般情况下，我们是先有数据库文档，然后才有具体的数据库实现，可是现实中，并不总是这样的，很有可能我们是先设计的数据库，到了该提交工程的时候，才想起来，噢，麻烦了，数据库文档还没呢。

设计数据库文档
数据库文档，无外乎表结构，视图，存储过程。
一般的表结构中包括以下几项内容：列名，数据类型，长度，精度，小数位数，是否为标识列，是否为主键，是否为外键，是否为空，有无默认值等。

一般将这些内容放入一张表格中来展示，目前展示表格的文件有很多，比如Excel、Word、Html、Xml等，但是，Excel、Word文件内容一般为二进制格式，不是纯粹的文本文件，因而可以选择的文件格式为Html和Xml，相对来说Html对格式的要求不严格，因而我们准备将我们生成数据库文档设置

为Html格式。

在Html中也有表格，不过是以标记的方式展示的。样子应该象下图：

authors						
列名	**类型**	**长度**	**是否为空**	**主键**	**外键**	**是否标识列**
au_id	char	11		Y		
au_lname	varchar	40				
au_fname	varchar	20				
phone	char	12				
address	varchar	40	Y			
city	varchar	20	Y			
state	char	2	Y			
zip	char	5	Y			
contract	bit	1				

在这个需求中，我们要使用DatabaseSchema，TableSchema，ColumnSchema，在数据库结构中有表结构的集合，在表结构中有列结构的集合，这样，我们可以用两个循环嵌套的方式，生成所有的表的文档。

```
<%@ CodeTemplate Language="C#" TargetLanguage="Html" Src="" Inherits="" Debug="False"
Description="Template description here."  ResponseEncoding="UTF-8"%>
<%@ Property Name="Developer" Type="String" Default="" Optional="False"
Category="Context" Description="作者" OnChanged="" Editor="" EditorBase="" Serializer="" %>
<%@ Property Name="SourceDB" Type="SchemaExplorer.DatabaseSchema" Default=""
Optional="False" Category="Context" Description="数据库名称" OnChanged="" Editor=""
EditorBase="" Serializer="" %>

<%@ Assembly Name="SchemaExplorer" %>
<%@ Assembly Name="System.Data" %>
<%@ Import Namespace="System.Data" %>
<%@  Import Namespace="SchemaExplorer" %>

<%-- 类生成内容在此--%>
<html>
<head><title><%=this.SourceDB.Name%></title></head>
<style>
 .header{background-color:#336699;color:white;}
 </style>
<body>
<%foreach(TableSchema table in SourceDB.Tables){%>
 <table border="1" width="92%">
  <caption class="header"><%=table.Name%></caption>
  <tr>
   <th class="header">列名</th>
   <th class="header">类型</th>
   <th class="header">长度</th>
   <th class="header">是否为空</th>
```

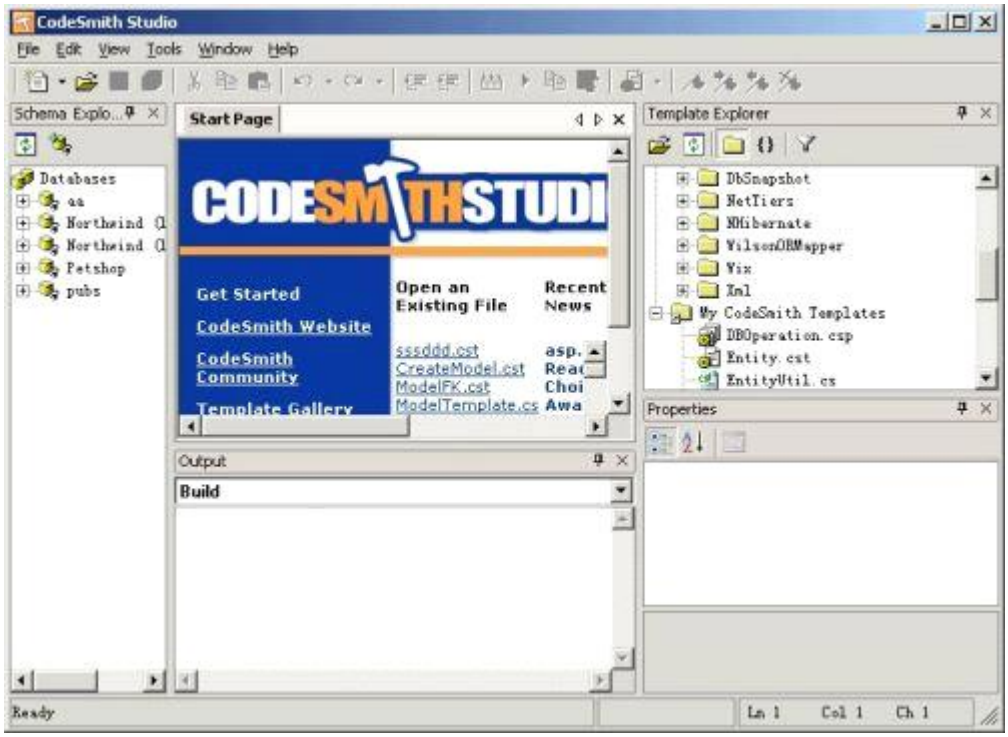
```
   <th class="header">主键</th>
   <th class="header">外键</th>
   <th class="header">是否标识列</th>
  </tr>
  <%foreach(ColumnSchema column in table.Columns){%>
   <tr>
    <td><%=column.Name%></td>
    <td><%=column.NativeType%></td>
    <td><%=column.Size%></td>
    <td><%=IsNull(column)%></td>
    <td><%=IsPK(column)%></td>
    <td><%=IsFK(column)%></td>
    <td><%=IsID(column)%></td>
   </tr>
  <%}%>
 </table>
<%}%>
</body>
</html>
<script runat="template">
public string IsID(ColumnSchema column)
{
 if((bool)column.ExtendedProperties["CS_IsIdentity"].Value)
 {
  return "Y";
 }
 return " ";
}
public string IsNull(ColumnSchema column)
{
 if(column.AllowDBNull)
 {
  return "Y";
 }
 return " ";
}
public string IsPK(ColumnSchema column)
{
 if(column.IsPrimaryKeyMember)
 {
  return "Y";
 }
 return " ";
}
public string IsFK(ColumnSchema column)
{
```

```
 if(column.IsForeignKeyMember)
 {
  return "Y";
 }
 return " ";
}

public string GetTableName(TableSchema table)
{
 return table.Name;
}

public string GetColumnName(ColumnSchema column)
{
 return column.Name;
}

public bool HasLength(ColumnSchema column)
{
 switch(column.NativeType)
 {
  case "char":
  case "varchar":
  case "nchar":
  case "nvarchar":
  case "varbinary":
  case "binary":
   return true;
  default:
   return false;
 }
}
</script>
```

5.图解制作模板

制作模板

这是典型的CodeSmith的界面：



<http://blog.photo.sina.com.cn/showpic.html>

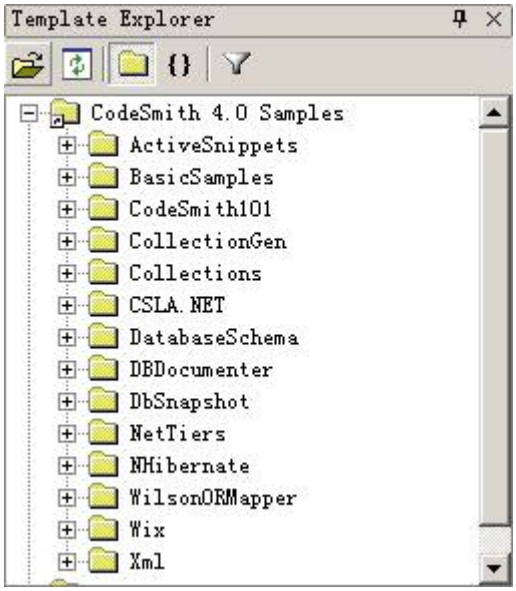
下面是数据库浏览窗口：

<http://blog.photo.sina.com.cn/showpic.html>

起始页窗口：

<http://blog.photo.sina.com.cn/showpic.html>

模板浏览器窗口：



<http://blog.photo.sina.com.cn/showpic.html>

属性窗口：

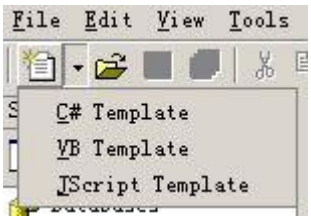
<http://blog.photo.sina.com.cn/showpic.html>

输出窗口：

<http://blog.photo.sina.com.cn/showpic.html>

制作模板：

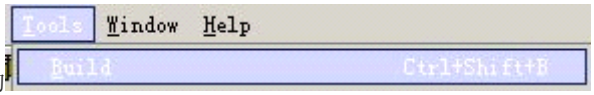
点工具栏上的新建C#Template：



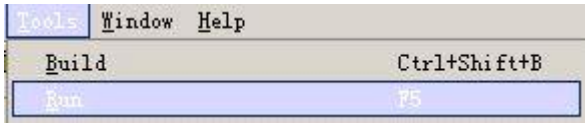
<http://blog.photo.sina.com.cn/showpic.html> ，产生一个普通C#模板。

四、生成代码

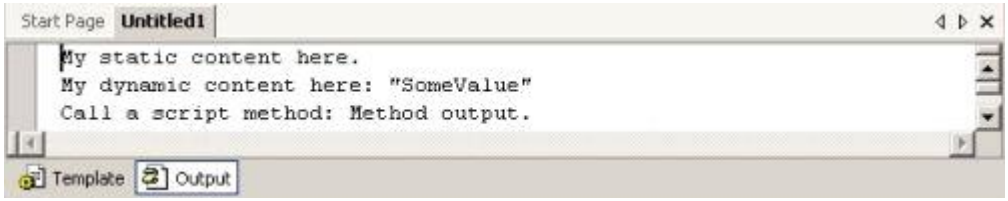
模板编辑完成后，点菜单上的



<http://blog.photo.sina.com.cn/showpic.html> ，编译模板，然后点菜单上的



<http://blog.photo.sina.com.cn/showpic.html> ，运行后会转到输出窗口



<http://blog.photo.sina.com.cn/showpic.html>。
将输出窗口的内容保存下来，就是CodeSmith生成的代码。

6.图解高级功能

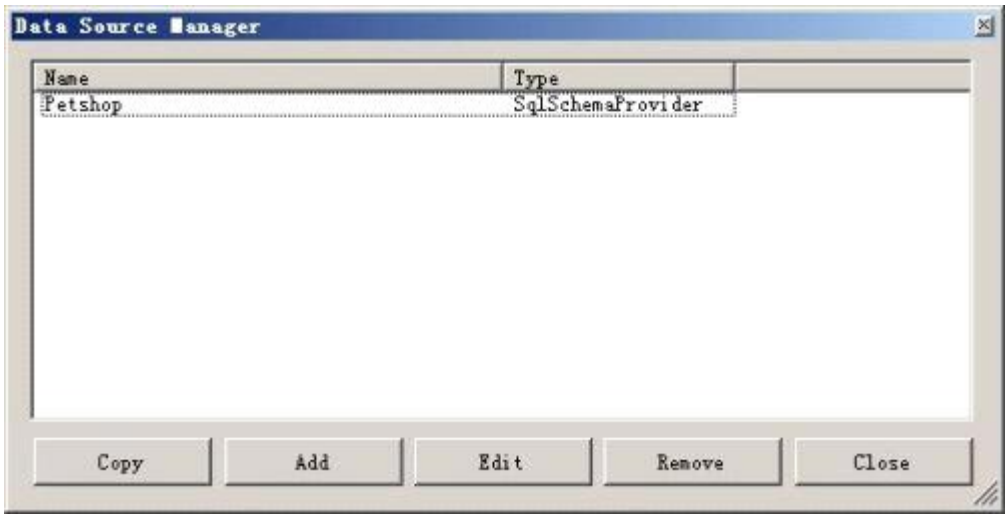
高级功能

模板声明中使用了DatabaseSchema和TableSchema，如何给这种属性赋值。

在CodeSmith的属性窗口中如下：

<http://blog.photo.sina.com.cn/showpic.html>
点“SourceDB”属性右边的按钮：

<http://blog.photo.sina.com.cn/showpic.html>
点“数据库选择”的浏览按钮



<http://blog.photo.sina.com.cn/showpic.html>

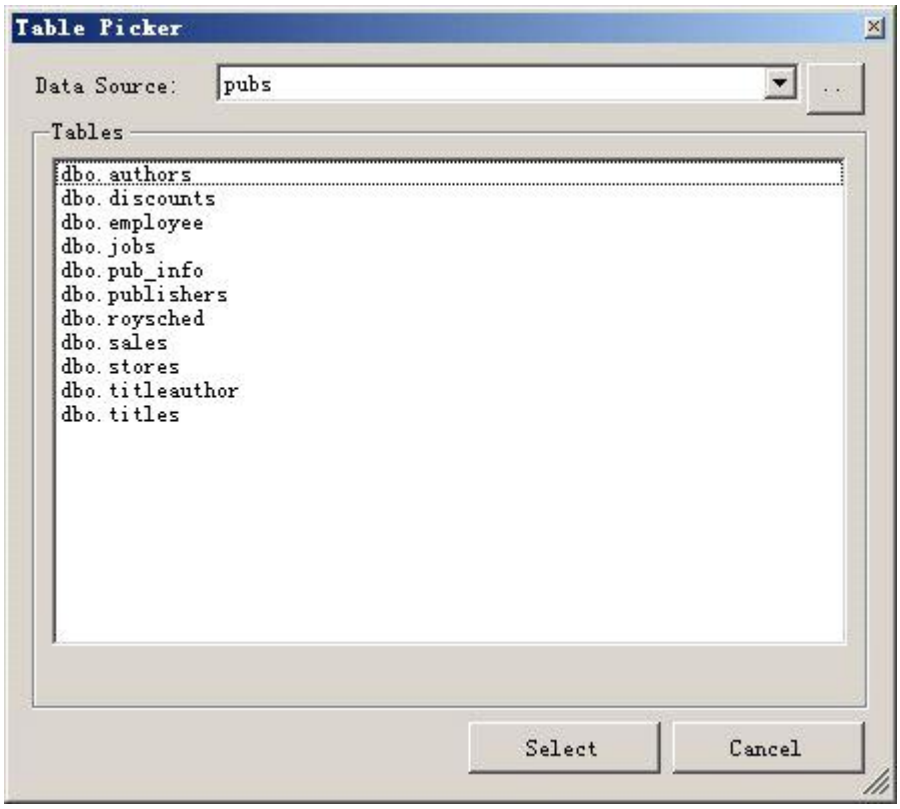
点“添加”，填写数据源信息：

<http://blog.photo.sina.com.cn/showpic.html>

点“Test”，测试数据源是否连接：

<http://blog.photo.sina.com.cn/showpic.html>

如果连接成功，那么显示数据库中所有的表：



<http://blog.photo.sina.com.cn/showpic.html>

选择其中某个表：

完成后如下图所示

<http://blog.photo.sina.com.cn/showpic.html>

7.数据库脚本模板

数据库脚本模板

```
<%--
Name:产生数据库的SQL脚本
Author: Jack.zhou
Description:
--%>
<%@ CodeTemplate Language="C#" TargetLanguage="T-SQL" Src="" Inherits=""
Debug="False" Description="Template description here." ResponseEncoding="UTF-8"%>
<%@ Property Name="SourceDB" Type="SchemaExplorer.DatabaseSchema" Default=""
Optional="False" Category="Context" Description="" OnChanged="" Editor="" EditorBase=""
Serializer="" %>
<%@ Property Name="NewDBName" Type="System.String" Default="NewDB" Optional="False"
Category="Context" Description="" OnChanged="" Editor="" EditorBase="" Serializer="" %>
<%@ Property Name="ProcessData" Type="System.Boolean" Default="False" Optional="False"
Category="Context" Description="" OnChanged="" Editor="" EditorBase="" Serializer="" %>
<%@ Assembly Name="SchemaExplorer" %>
<%@ Assembly Name="SchemaExplorer.Trigger" %>
<%@ Import Namespace="SchemaExplorer" %>
<%@ Assembly Name="System.Data" %>
<%@ Import Namespace="System.Data" %>
Use master
Go
--region 判断数据库是否存在，如果存在，那么删除这个数据库
IF Exists(Select '' From sysdatabases Where name='<%=this.NewDBName%>')
Begin
 Drop Database [<%=this.NewDBName%>]
End
Go
--endregion
--region 创建数据库<%=this.NewDBName%>
Create Database [<%=this.NewDBName%>]
Go
Use [<%=this.NewDBName%>]
Go
--endregion

--region 创建自定义类型
<%
UserDefineTypeCollection udtc=new UserDefineTypeCollection (this.SourceDB);
foreach(UserDefineType udt in udtc.UserDefineTypes)
{
 Response.WriteLine(udt.ToString());
}
%>
```

```
Go
--endregion

<%foreach(TableSchema table in this.SourceDB.Tables){%>
--region 判断数据表[<%=table.Name%>]是否存在，如果存在，那么删除这个数据表
If Exists(Select '' From sysobjects Where name='<%=table.Name%>')
Begin
 Drop Table [<%=table.Name%>]
End
Go
--endregion
<%}%>

<%foreach(TableSchema table in this.SourceDB.Tables){%>
--region 创建数据表<%=table.Name%>
Create Table [<%=table.Name%>]
(
 <%for(int i=0;i<table.Columns.Count;i++){%>
 <%if(i!=table.Columns.Count-1){%>
 <%=GetColumn(table.Columns[i])%>,
 <%}%>
 <%else{%>
 <%=GetColumn(table.Columns[i])%>
 <%}%>
 <%}%>
)
Go
--endregion
<%}%>

--region 添加数据
<%if(this.ProcessData){%>
use <%=this.SourceDB.Name%>New
Go
<%foreach(TableSchema table in this.SourceDB.Tables){%>
<%if(HasIdentity(table)){%>
DBCC CheckIdent ('[<%= table.Owner %>].[<%= table.Name %>]', reseed, 1)
Set Identity_Insert [<%=table.Owner%>].[<%=table.Name%>] On
<%}%>
<%DataTable dt=table.GetTableData();
foreach(DataRow dr in dt.Rows){%>
Insert Into [<%=table.Owner%>].[<%=table.Name%>](<%=GetTableColumnList(table)%>)
Values(<%=GetData(dr,table)%>)
 <%}%>
<%if(HasIdentity(table)){%>
Set Identity_Insert [<%=table.Owner%>].[<%=table.Name%>] Off
```

```
<%}%>
Go
<%}%>
<%}%>
--endregion

<%foreach(TableSchema table in this.SourceDB.Tables){%>
--region 为[<%=table.Name%>]添加主键约束
<%if(table.HasPrimaryKey){%>
Alter Table  [<%=table.Name%>] Add Constraint <%=table.PrimaryKey.Name%> Primary Key
(<%=GetPkString(table)%>)
<%}%>
Go
--endregion
<%}%>

<%foreach(TableSchema table in this.SourceDB.Tables){%>
--region 为[<%=table.Name%>]添加外键约束
<%foreach(TableKeySchema key in table.ForeignKeys){%>
Alter Table  [<%=table.Name%>] Add Constraint <%=key.Name%> Foreign Key <%
=GetFKString(key)%>
<%}%>
Go
--endregion
<%}%>

<%foreach(TableSchema table in this.SourceDB.Tables){%>
--region 为[<%=table.Name%>]添加其他约束
<%foreach(ColumnSchema column in table.Columns){%>
<%=GetOtherConstrain(column)%>
<%}%>
Go
--endregion
<%}%>

<%foreach(TableSchema table in this.SourceDB.Tables){%>
--region 为[<%=table.Name%>]添加索引
<%
//foreach(TableKeySchema prop in table.Keys)
//{
// Response.WriteLine(prop.Name+":"+prop.Name);
//}
%>
<%foreach(IndexSchema index in table.Indexes){%>
<%if(GetIndexType(index)!=""){%>
Create <%=GetIndexType(index)%> Index [<%=index.Name%>] On [<%
```

```
=index.Table.Name%>] (<%=GetIndexColumn(index)%>) <%=GetIndexOption(index)%>
<%}%>
<%}%>
Go
--endregion
<%}%>

<%foreach(ViewSchema view in this.SourceDB.Views){%>
--region 创建视图<%=view.Name%>
<%=view.ViewText%>
Go
--endregion
<%}%>

<%foreach(CommandSchema command in this.SourceDB.Commands){%>
--region 创建存储过程<%=command.Name%>
<%=command.CommandText%>
Go
--endregion
<%}%>

--region 创建触发器
<%foreach(TableSchema table in this.SourceDB.Tables){%>
<%TriggerCollection triggers=new TriggerCollection(table);%>
<%foreach(Trigger trigger in triggers.Triggers){%>
--region 创建<%=trigger.Name%>触发器
<%=trigger.Text%>
--endregion
<%}%>
Go
<%}%>--endregion

<script runat="template">
#region 生成数据库结构
public string GetIndexOption(IndexSchema index)
{
 string s="With ";
 if(int.Parse(index.ExtendedProperties["CS_OrigFillFactor"].Value.ToString())!=0)
 {
  s+="FillFactor="+int.Parse(index.ExtendedProperties["CS_OrigFillFactor"].Value.ToString())+",";
 }
 if(bool.Parse(index.ExtendedProperties["CS_PadIndex"].Value.ToString()))
 {
  s+="PAD_INDEX,";
 }
 if(bool.Parse(index.ExtendedProperties["CS_IgnoreDupKey"].Value.ToString()))
```

```
 {
  s+="IGNORE_DUP_KEY,";
 }
 if(bool.Parse(index.ExtendedProperties["CS_DropExist"].Value.ToString()))
 {
  s+="DROP_EXISTING,";
 }
 if(s.Length==5)
 {
  return "";
 }
 return s.Substring(0,s.Length-1);

}
public string GetIndexColumn(IndexSchema index)
{
 string s="";
 foreach(MemberColumnSchema column in index.MemberColumns)
 {
  if((int)column.ExtendedProperties["CS_IsDescending"].Value==0)
  {
   s+=column.Name+",";
  }
  else
  {
   s+=column.Name+" Desc,";
  }
 }
 if(s=="")
 {
  return s;
 }
 return s.Substring(0,s.Length-1);
}
public string GetIndexType(IndexSchema index)
{
 if(index.IsClustered==true && index.IsPrimaryKey==true && index.IsUnique==true)
 {
  return "";
 }
 else if(index.IsClustered==false && index.IsPrimaryKey==false && index.IsUnique==false)
 {
  return "NonClustered";
 }
 else if(index.IsClustered==false && index.IsPrimaryKey==false && index.IsUnique==true)
 {
```

```
  return "Unique";
 }
 return "";
}
public string GetColumn(ColumnSchema column)
{
 string s="";
 s+="["+column.Name+"] "+column.NativeType+" ";
 if(IsId(column))
 {
  s+="identity("+column.ExtendedProperties
["CS_IdentitySeed"].Value+","+column.ExtendedProperties["CS_IdentityIncrement"].Value+") ";
 }
 if(HasLength(column))
 {
  s+="("+column.Size+") ";
 }
 if(column.AllowDBNull)
 {
  s+="null ";
 }
 else
 {
  s+="not null";
 }
 return s;
}
public bool IsId(ColumnSchema column)
{
 return (bool)column.ExtendedProperties["CS_IsIdentity"].Value;
}
public bool HasLength(ColumnSchema column)
{
 switch(column.NativeType)
 {
  case "char":
  case "varchar":
  case "nchar":
  case "nvarchar":
   return true;
  default:
   return false;
 }
}
public string GetPkString(TableSchema table)
{
```

```
 string s="";
 foreach(MemberColumnSchema column in table.PrimaryKey.MemberColumns)
 {
  s+="["+column.Name+"]"+",";
 }
 return s.Substring(0,s.Length-1);
}
public string GetFKString(TableKeySchema key)
{
 string s="(";
 foreach(MemberColumnSchema column in key.ForeignKeyMemberColumns)
 {
  s+="["+column.Name+"],";
 }
 s=s.Substring(0,s.Length-1);
 s+=")";
 s+=" References ["+key.PrimaryKeyTable.Name+"](";
 foreach(MemberColumnSchema column in key.PrimaryKeyMemberColumns)
 {
  s+="["+column.Name+"],";
 }
 s=s.Substring(0,s.Length-1);
 s+=")";
 return s;
}
public string GetOtherConstrain(ColumnSchema column)
{
 string s="";
 int i=1;
 while(true)
 {
  if(column.ExtendedProperties["CS_Constraint"+i+"Name"]==null)
  {
   break;
  }

  if(column.ExtendedProperties["CS_Constraint"+i+"Type"].Value.ToString().ToUpper().Trim()
=="DEFAULT")
  {
   s+="Alter Table ["+column.Table.Name+"] Add Constraint "+column.ExtendedProperties
["CS_Constraint"+i+"Name"].Value+" Default "+column.ExtendedProperties
["CS_Constraint"+i+"Definition"].Value.ToString()+" For ["+column.Name+"]\r\n";
  }
  else
  {
   s+="Alter Table ["+column.Table.Name+"] Add Constraint "+column.ExtendedProperties
```

```
["CS_Constraint"+i+"Name"].Value+" Check "+column.ExtendedProperties
["CS_Constraint"+i+"Definition"].Value.ToString()+"\r\n";
  }
  i++;
 }
 if(s=="")
 {
  return null;
 }
 return s;
}
#endregion

#region 产生数据
public bool HasIdentity(TableSchema table)
{
 foreach(ColumnSchema column in table.Columns)
 {
  if((bool)column.ExtendedProperties["CS_IsIdentity"].Value)
  {
   return true;
  }
 }
 return false;
}
public string GetData(DataRow dr,TableSchema table)
{
 string s="";
 foreach(ColumnSchema column in table.Columns)
 {
  if(dr[column.Name] is System.DBNull)
  {
   s+="null,";
  }
  else
  {
   if(IsNumeric(column))
   {
    if(column.SystemType.ToString()!="System.Boolean")
    {
     s+=FormatString(dr[column.Name].ToString())+",";
    }
    else
    {
     s+= (((bool)dr[column.Name])==true?1:0) +",";
    }
```

```
    }
    else
    {
     if(column.SystemType.ToString()!="System.Byte[]")
     {
      s+="'"+FormatString(dr[column.Name].ToString())+"',";
     }
     else
     {
      byte[] bs=(byte[])dr[column.Name];
      string x="0x";
      foreach(byte b in bs)
      {
       x+=b.ToString("X");
      }
      s+="'"+x+"',";
     }
    }
   }

  }
  if(s.Length==0)
  {
   return s;
  }
  return s.Substring(0,s.Length-1);
 }
 public string FormatString(string s)
 {
  return s.Replace("'","''");
 }
 public string GetTableColumnList(TableSchema table)
 {
  string s="";
  foreach(ColumnSchema column in table.Columns)
  {
   s+="["+column.Name+"],";
  }
  if(s.Length==0)
  {
   return "";
  }
  return s.Substring(0,s.Length-1);
 }
 public bool IsNumeric(ColumnSchema column)
 {
```

```
switch(column.NativeType.ToLower())
{
 case "int":
 case "smallint":
 case "bigint":
 case "float":
 case "decimal":
 case "money":
 case "numeric":
 case "real":
 case "smallmoney":
 case "tinyint":
 case "bit":
  return true;
 default:
  return false;
 }
}
#endregion

</script>
```

先从模板声明开始：

属性声明中的ProcessData表示的是是否要生成插入数据的脚本，也就是说不光可以生成数据库脚本，也可以附带数据库中的内容，这样导入导出数据也比较方便，比SQLServer自带的工具要放心很多。

属性声明中的NewDBName，只是给了一个新的数据库的名称。

SourceDB源数据库

这次的模板主要解决了以下问题：

1.解决了一般数据库脚本中缺失的触发器和自定义数据类型的定义。

2.解决了插入数据时候binary，image类型使用普通SQL语句插入的问题。

3.捎带这解决了字符串类型中的单引号问题和数据为null的问题。

4.Copy着解决了标识列插入数据的问题。

5.发现并解决了约束脚本中缺失Check约束的问题。

6.发现并解决了插入数据过程中表之间存在引用关系的问题。

7.索引部分解决问题不完善，但可以应付98%的状况了。

8.触发器和自定义数据类型部分解决方案有缺陷，只能应付SQLServer数据库，而且连接字符串必须为以下形式"server=?;databaes=?;uid=?;pwd=?;"。

以下逐个讲解这些问题的发现并解决的过程

1.其实在CodeSmith中并没有一个类可以获取触发器或自定义数据类型，自定义数据类型的发现比较困难，奇怪每次生成的SQL脚本都在创建数据表的时候挂了，拿提示消息来分析：

服务器: 消息 2715，级别 16，状态 7，行 4

第 1 个列或参数: 无法找到数据类型 id。

确实SQL的系统数据类型中没有这个id类型，一般的系统中很少使用到自定义数据类型

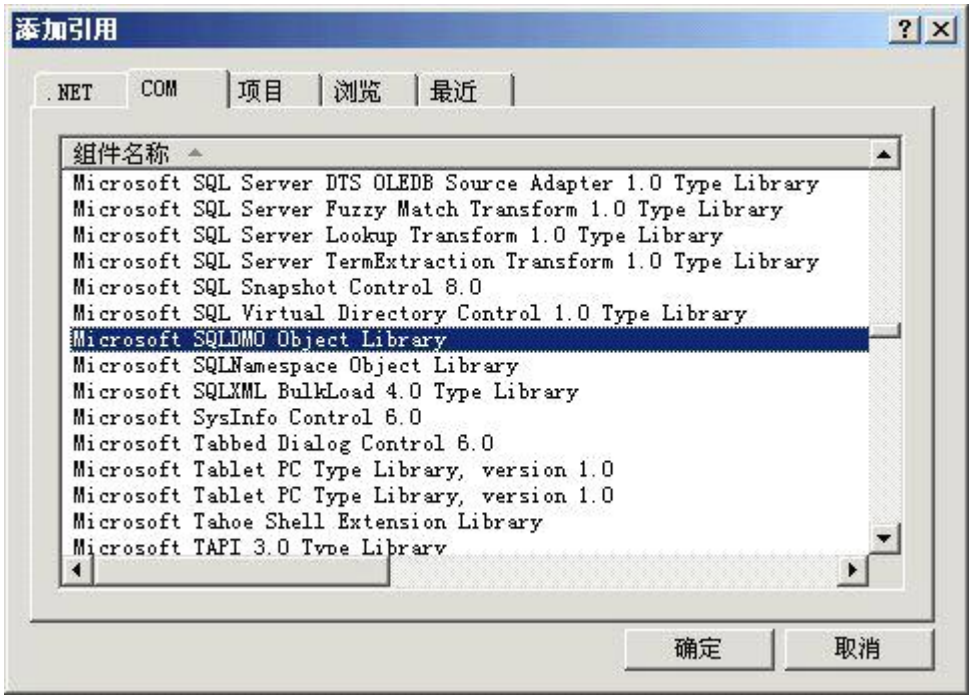
企业管理器或查询分析器，怎么将自定义数据类型的脚本给拿到呢？
问题：微软是怎么生成SQL脚本的？当然是用它自己的程序"企业管理器"了，"查询分析器"也可以做到，上世纪90年代整个就是个COM的年代，COM组件统治了大部分共享组件的空间，微软在写SQLServer的时候没理由不使用COM组件。COM组件其实就是个类库，不过很遗憾，当时的很多类库是C，或C++的天下，现在的.Net中可以使用的语言和以前的C，C++差别较大，不能兼容，好在.Net也是微软自己家的，因而，.Net体系中还可以使用COM组件。

CodeSmith中不能直接支持COM组件，可是CodeSmith支持使用.Net组件。

翻到"SQL数据管理对象"的章节找到SQLDMO，其实SQLDMO并不是SQL数据管理对象，含义是：分布式管理对象。

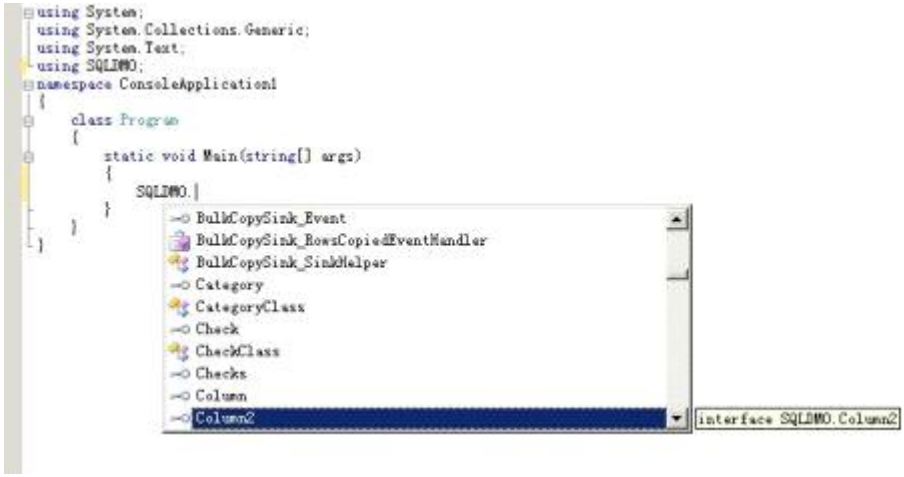
要从SQLDMO中获取用户自定义数据类型，就要先想办法通过C#使用SQLDMO组件。

先添加SQLDMO组件：在.Net工程中首先添加SQLDMO组件的引用。



<http://blog.photo.sina.com.cn/showpic.html>
然后在类中使用“SQLDMO”命名空间。

```
using SQLDMO;
```



<http://blog.photo.sina.com.cn/showpic.html>

连接到服务器，列出服务器上的数据库。

<http://blog.photo.sina.com.cn/showpic.html>

这三个玩意正好是“查询分析器”要连接到某个服务器，所需要的3个参数

如果连接成功，那么就要完成列出服务器上所有的数据库的功能了。

还好，SQLServerClass中有个集合Databases中放的就是所有的数据库对象了。

那么只需要一个foreach就可以了。

```
using System;
using System.Collections.Generic;
using System.Text;
using SQLDMO;
namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            SQLServerClass sqlServer = new SQLServerClass();
            sqlServer.Connect("SERVER", "sa", "");
            foreach (Database db in sqlServer.Databases)
            {
                Console.WriteLine(db.Name);
            }
        }
    }
}
```

<http://blog.photo.sina.com

.cn/showpic.html>
接下来要从数据库中获得自定义数据类型。
Database中还有个集合UserDefinedDatatypes，就是当前数据库中所有的自定义数据类型。

当然我们需要的不光是自定义数据类型的名称，还需要其他的内容，到底需要多少，那得看SQLServer产生一个自定义数据类型需要哪些信息了。
查SQL的联机丛书，可以知道，要创建自定义数据类型，要使用系统存储过程“sp_addtype”。那么这个存储过程需要哪些参数呢。

sp_addtype

创建用户定义的数据类型。

语法

```
sp_addtype [ @typename = ] type,
   [ @phystype = ] system_data_type
   [ ,[ @nulltype = ] 'null_type' ]
   [ ,[ @owner = ] 'owner_name' ]
```

<http://blog.photo.sina.com.cn/showpic.html>
前三个参数比较重要。@typename为自定义的名称，@phystype为自定义类型的基础类型，@nulltype为是否为空。
示例：

```
USE master
EXEC sp_addtype telephone, 'varchar(24)', 'NOT NULL'
EXEC sp_addtype fax, 'varchar(24)', 'NULL'
```

<http://blog.photo.sina.com.cn/showpic.html>
从示例中可以看出要从SQLDMO的UserDefinedDatatype中获取的内容：
@typename对应Name属性。
@nulltype对应AllowNulls属性。
@phystype比较复杂，它中间可能包括：具体的数据类型，长度，精度，小数位数等。
那么我们要获取的内容有以下几项：
Name：自定义数据类型的名称。
BaseType：自定义类型的基础类型。
Length：长度。
AllowNulls：是否允许为空。
NumericPrecision：精度。
NumericScale：小数位数。

那么我们应该做一个实体类，用来存储这些内容。
示例如下：

```
using System;
using System.Collections.Generic;
using System.Text;
using SchemaExplorer;
using SQLDMO;

namespace SchemaExplorer
{
public class UserDefineType
{
string name;

public string Name
{
get { return name; }
```

```
set { name = value; }
}
bool isNull;

public bool IsNull
{
get { return isNull; }
set { isNull = value; }
}

int size;

public int Size
{
get { return size; }
set { size = value; }
}

string baseType;

public string BaseType
{
get { return baseType; }
set { baseType = value; }
}
int precision;

public int Precision
{
get { return precision; }
set { precision = value; }
}
int scale;

public int Scale
{
get { return scale; }
set { scale = value; }
}

public override string ToString()
{
string s =string.Format( "Execute sp_addtype {0}, '{1}', '{2}'",name,GetTypeString
(),isNull?"null":"not null");
return s;
}

private string GetTypeString()
{
if (HasLength(baseType))
{
return baseType + "(" + size + ")";
}
return baseType;
}
```

```
private bool HasLength(string s)
{
switch (s)
{
case "varchar":
case "char":
case "nvarchar":
case "nchar":
return true;
default:
return false;
}
}
}
}
```

还需要一个用来产生这个对象的类：

```
using System;
using System.Collections.Generic;
using System.Text;
using SchemaExplorer;
using SQLDMO;
namespace SchemaExplorer
{
public class UserDefineTypeCollection
{
DatabaseSchema database;
List<UserDefineType> userDefineTypes = new List<UserDefineType>();

public List<UserDefineType> UserDefineTypes
{
get { return userDefineTypes; }
set { userDefineTypes = value; }
}
public UserDefineTypeCollection(DatabaseSchema database)
{
this.database = database;
string cnnString =database.ConnectionString;
string[] ary = cnnString.Split(';');
string dbName = GetValue("database", ary);
string serverName = GetValue("server", ary);
string login = GetValue("uid", ary);
string passWord = GetValue("pwd", ary);
SQLDMO.SQLServer2Class server = new SQLDMO.SQLServer2Class();
server.Connect(serverName, login, passWord);
SQLDMO._Database db = server.Databases.Item(dbName, "dbo");

foreach (_UserDefinedDatatype udt in db.UserDefinedDatatypes)
{
UserDefineType userDefineType = new UserDefineType();
userDefineType.Name = udt.Name;
userDefineType.IsNull = udt.AllowNulls;
userDefineType.Size = udt.Length;
userDefineType.BaseType = udt.BaseType;
userDefineType.Precision = udt.NumericPrecision;
userDefineType.Scale = udt.NumericScale;
```

```
userDefineTypes.Add(userDefineType);

}
}
private string GetValue(string find, string[] ary)
{
foreach (string s in ary)
{
string[] temp = s.Split('=');
if (temp[0] == find)
{
return temp[1];
}
}
return "";
}
}
}
```

将工程编译完成后，把程序集放入CodeSmith文件夹的Addins文件夹中。然后就可以在CodeSmith模板中使用了

Image、单引号、null的问题，微软给的示例数据库中这些问题都存在
单引号，内容中单引号，用替换的办法把单个单引号替换为两个单引号就可以解决了

null问题是由于在.Net中如果字符串为null，那么输出的将是个空字符串，所以这个问题只有看模板文件的输出结果才看得到，对于普通的Insert语句而言，在values列表中可以出现null关键字，因而只要判断读取的内容中有null即可，但判断数据为null不能用C#中的null来判断，只能用obj is DBNull来判断。

对于图像和二进制数据在数据库中的存储来说，我们一般都认为在数据库中存储的是二进制的byte数组。平常我们向数据库中存储图像时，采用的方式也是二进制数组，获取的时候也是。

但在CodeSmith中如果直接用对象的ToString方法，可以获取大部分数据的字符串表达形式，对于byte数组，只能得到"System.Byte[]"，并不是我们想要的内容，考虑将byte数组中的内容，挨个变成String，采用格式化的方式，将每个byte变为字符串，然后拼接起来，但作为values中的内容的时候，这个字符串的前面必须加"0x"，而且不能在前后加单引号。

标识列在插入数据的时候，需要先允许在标识列插入数据，等插入完成后，再设置为不允许即可。

检查约束可以由ColumnSchema的扩展属性中获取，比较麻烦的是，不知道到底有几个检查约束。

表和表之间存在引用关系，但是我们用foreach遍历数据库中表的时候，并不能保证主表的数据先添加上，而后再添加子表的数据，采用了一个比较极端的办法：一旦表结构做好了，约束脚本，主键，外键，触发器等都没有的时候，那么就开始插入数据，这个时候，表和表之间没有任何关系，因而先添加谁，后添加谁，并没有必然的顺序。

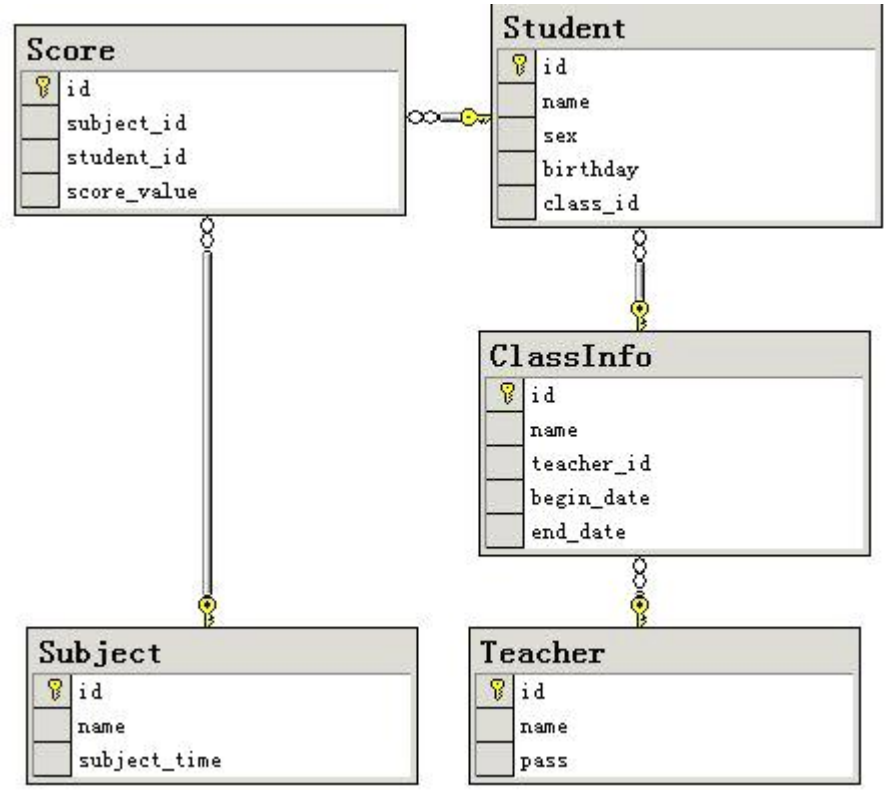
在获取触发器，自定义类型时，需要服务器名称，登陆名，密码，这三项是从CodeSmith模板的属性声明中的SourceDB中获取的，只处理了"server=?;databaes=?;uid=?;pwd=?;"，没有处理其他的连接字符串，因而适应性不广

8.再论实体类

实体类

实体类在MIS系统中用来做业务层和表现层之间传递数据的桥梁。界面上展示数据从实体对象中获取，从界面上获取的数据将会包装成为实体对象，以便传递给业务层。

由于实体类多数情况下是从数据表中得到的，因而和数据表的设计关系密切。一般的表和表之间可能存在一种主外键关系，在数据库中设计这种关系，一般比较容易实现，通过在子表中添加主表主键的办法做到。这样在子表中就有了主表的字段了。如果原封不动的使用表中的字段产生一个实体类，那么在“子对象”中，和“主对象”有联系的内容，也就是那个“主对象”的主键了，这样在“子对象”中要获取“主对象”的其他信息，就比较麻烦。必须先从“子对象”中得到“主对象”的主键信息，然后调用获取“主对象”的业务方法，获取“主对象”，然后才能得到其他的信息。比如在下图中：



<http://blog.photo.sina.com.cn/showpic.html>

要获取某个成绩的学生的班主任名称，那就要先获取当前成绩是属于哪个学生的，然后获取这个学生所在的班级，然后获取这个班级的班主任，这样中间经过的层次太多，会让程序开发陷入复杂的控制中。

如果我们换一种思路，既然成绩是属于某个学生的，那么为什么不在成绩实体中，设置一个学生属性，而非要设置一个学生编号属性？为什么不在学生实体中，设置一个班级属性，而非要设计一个班级编号属性？为什么不在班级实体中，设置一个班主任属性，而非要设置一个班主任编号属性？如果以上部分都实现了，那么获取某个成绩的班主任姓名就比较容易取得。

示例：score.Student.ClassInfo.Teacher.Name，这种方式比较容易理解。

前提：由于复合主键不太容易处理，下面的例子以单主键的形式展开。

下面显示传统方式的实体类设计示例：

```
public class Teacher
{
    int id;
    string name;
    string pass;
    public int Id
    {
        get { return id; }
        set { id = value; }
    }
    public string Name
    {
        get { return name; }
        set { name = value; }
    }
    public string Pass
    {
        get { return pass; }
        set { pass = value; }
    }
}
```

<http://blog.photo.sina.com.cn/showpic.html>

```
public class ClassInfo
{
    int id;

    public int Id
    {
        get { return id; }
        set { id = value; }
    }
    string name;

    public string Name
    {
        get { return name; }
        set { name = value; }
    }
    int teacher_id;

    public int Teacher_id
    {
        get { return teacher_id; }
        set { teacher_id = value; }
    }
    DateTime begin_date;

    public DateTime Begin_date
    {
        get { return begin_date; }
        set { begin_date = value; }
    }
    DateTime end_date;

    public DateTime End_date
    {
        get { return end_date; }
        set { end_date = value; }
    }
}
```

<http://blog.photo.sina.com.cn/showpic.html>

下面是采用新设计思路设计的ClassInfo类：

```
public class ClassInfo
{
    int id;

    public int Id
    {
        get { return id; }
        set { id = value; }
    }
    string name;

    public string Name
    {
        get { return name; }
        set { name = value; }
    }
    Teacher teacher;

    public Teacher Teacher
    {
        get { return teacher; }
        set { teacher = value; }
    }

    DateTime begin_date;

    public DateTime Begin_date
    {
        get { return begin_date; }
        set { begin_date = value; }
    }
    DateTime end_date;

    public DateTime End_date
    {
        get { return end_date; }
        set { end_date = value; }
    }
}
```

<http://blog.photo.sina.com.cn/showpic.html>

这样如果有了班级对象，那么很容易获取教师的其他信息。

变化的内容为：从原来的只包含单个父表字段，到包含完整的父表对象。

```
int teacher_id;

public int Teacher_id
{
    get { return teacher_id; }
    set { teacher_id = value; }
}
Teacher teacher;

public Teacher Teacher
{
    get { return teacher; }
    set { teacher = value; }
}
```

<http://blog.photo.sina.com.cn/showpic.html>

具体分析为：

1. 原来外键字段的类型有变化，不再使用数据库中字段的类型。
2. 原来外键字段的名称有变化，不再使用数据库中字段的名称，改为父表的名称。

这样以来，我们就要解决这个问题：如果某个字段为外键，那么就要获取这个字段所属的表的名称，也就是主键表的名称，用来做在实体类中的数据类型和属性名称。

类型名称和属性名称相同，难道不冲突吗？

不冲突，反而有利于代码的可读性。

实体类生成的关键点为以下四点：
1.获得字段和属性的类型。
2.获得字段和属性的名称。
3.获得类名。
4.字段，属性，类名的命名规则。

字段和属性的名称有以下规则：
1.字段名称采用下划线+骆驼形式的数据表字段名，例如：_id,_name。
2.属性名称采用Pascal形式的数据表字段名，例如：Id，Name。

类名有以下规则：
如果表名为复数，那么去掉结尾的“s”，将其用Pascal形式表达，例如：Student，Teacher，Book。

字段和属性的类型一般采取数据库中的类型，在C#语言中的类型对应的形式。

变化的内容如下：
1.如果数据表的某个列为非外键，那么一切照旧。
2.如果数据表的某个列为外键，那么字段名称为“主表”的类名的骆驼表达形式，属性的名称为“主表”的类名。

实现了哪些：
1.命名规则，已经有了，骆驼或Pascal。
2.表名的获取，可以以TableSchema的Name属性为基础。
3.非外键列的字段属性的名称和类型可以从ColumnSchema的Name属性和SystemType或NativeType中获得。

对于外键列，需要获得的内容为，当前外键列的主表为哪个

对于TableSchema，可以从ForeignKeyColumns获得所有的外键列，可是每个外键列的属性中并没有这个列所属的“主表”是哪个的属性，只从ForeignKeyColumns中获取，并不完全可行。
对于ColumnSchema而言，只能从IsForeignKeyMember中获得当前列是否为外键，也不完全可行。

TableSchema中有另外一个与外键相关的集合：ForeignKeys，这个集合中是所有外键信息的集合，每个外键信息中都有这个外键信息的“主表”属性：PrimaryKeyTable，这个属性的类型为TableSchema，那我取这个PrimaryKeyTable的Name属性，那不就是我们要找的“主表”类的基础吗！

在TableSchema除了有ForeignKeys之外，我们还有一个集合NonForeignKeyColumns，“非外键列集合”，两个循环，一个做非外键列，一个做外键列，合起来不就是我们想要的内容吗。
参考模板如下：

```
<%--
Name:生成实体类的模板
Author: Jack.Zhou
Description:
--%>
<%@ CodeTemplate Language="C#" TargetLanguage="C#" Src="" Inherits="" Debug="False"
Description="Template description here." ResponseEncoding="UTF-8" %>
<%@ Property Name="SourceTable" Type="SchemaExplorer.TableSchema" Default=""
Optional="False" Category="" Description="" OnChanged="" Editor="" EditorBase=""
Serializer="" %>
<%@ Property Name="NameSpace" Type="System.String" Default="Model" Optional="False"
Category="" Description="" OnChanged="" Editor="" EditorBase="" Serializer="" %>
<%@ Assembly Name="SchemaExplorer" %>
```

```
<%@ Assembly Name="System.Data" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="SchemaExplorer" %>
using System;
namespace <%=this.NameSpace%>
{
 public class <%=GetClassName(this.SourceTable.Name)%>
 {
  #region 非外键列
  <%foreach(ColumnSchema column in this.SourceTable.NonForeignKeyColumns){%>
  private <%=GetCSDataType(column)%> _<%=ToCamel(column.Name)%>;
  public <%=GetCSDataType(column)%> <%=ToPascal(column.Name)%>
  {
   get{return _<%=ToCamel(column.Name)%>;}
   set{_<%=ToCamel(column.Name)%>=value;}
  }

  <%}%>
  #endregion
  #region 外键列
  <%foreach(TableKeySchema key in this.SourceTable.ForeignKeys){%>
  private <%=GetClassName(key.PrimaryKeyTable.Name)%> _<%=GetObjectName
(key.PrimaryKeyTable.Name)%>;
  public <%=GetClassName(key.PrimaryKeyTable.Name)%> <%=GetClassName
(key.PrimaryKeyTable.Name)%>
  {
   get{return _<%=GetObjectName(key.PrimaryKeyTable.Name)%>;}
   set{_<%=GetObjectName(key.PrimaryKeyTable.Name)%>=value;}
  }

  <%}%>
  #endregion
 }
}
<script runat="template">
public string ToPascal(string s)
{
 return s.Substring(0,1).ToUpper()+s.Substring(1);
}
public string ToCamel(string s)
{
 return s.Substring(0,1).ToLower()+s.Substring(1);
}
public string GetClassName(string s)
{
 if(s.EndsWith("s"))
 {
  return ToPascal(s.Substring(0,s.Length-1));
 }
 return ToPascal(s);
}
public string GetObjectName(string s)
{
 if(s.EndsWith("s"))
 {
```

```
  return ToCamel(s.Substring(0,s.Length-1));
 }
 return ToCamel(s);
}
public static string GetCSDataType(ColumnSchema column)
        {
            if (column.Name.EndsWith("TypeCode")) return column.Name;
            switch (column.DataType)
            {
                case DbType.AnsiString: return "string";
                case DbType.AnsiStringFixedLength: return "string";
                case DbType.Binary: return "byte[]";
                case DbType.Boolean: return "bool";
                case DbType.Byte: return "byte";
                case DbType.Currency: return "decimal";
                case DbType.Date: return "DateTime";
                case DbType.DateTime: return "DateTime";
                case DbType.Decimal: return "decimal";
                case DbType.Double: return "double";
                case DbType.Guid: return "Guid";
                case DbType.Int16: return "short";
                case DbType.Int32: return "int";
                case DbType.Int64: return "long";
                case DbType.Object: return "object";
                case DbType.SByte: return "sbyte";
                case DbType.Single: return "float";
                case DbType.String: return "string";
                case DbType.StringFixedLength: return "string";
                case DbType.Time: return "TimeSpan";
                case DbType.UInt16: return "ushort";
                case DbType.UInt32: return "uint";
                case DbType.UInt64: return "ulong";
                case DbType.VarNumeric: return "decimal";
                default:
                    {
                        return "__UNKNOWN__" + column.NativeType;
                    }
            }
        }
</script>
```

9.三层模板生成

按照三层结构划分一个MIS系统，可以分解为：表现层、业务逻辑层、数据访问层

表现层：主要负责收集数据和展示数据。
业务逻辑层：主要负责处理从表现层传递过来的数据，或给表现层提供数据用于展示。
数据访问层：主要负责和具体的数据库系统交互。

在数据访问层一般会写一些和数据库有直接关系的方法，例如：添加、修改、根据主键删除、根据外键删除、根据主键查询单个实体、根据外键查询实体集合、查询所有实体等，而这些也可以使用模板

来生成。

首先要做的是确定方法声明：返回类型、方法名称、参数列表。

返回类型：

1. 增删改方法的返回类型一般为int，用来表达影响的行数。
2. 查询单个实体方法的返回类型，一般为实体对象。
3. 查询实体集合的返回类型，一般为实体集合，可以使用泛型集合。

方法名称：

1. 添加方法的名称可以用动词（Add或Insert）+名词（实体类名）或单个动词（Add或Insert）。
2. 修改方法的名称可以用动词（Edit或Modify）+名词（实体类名）或单个动词（Edit或Modify）。
3. 根据主键删除方法的名称可以采用动词（Delete）+名词（实体类名）或单个动词（Delete）+"By"+主键属性名称。
4. 根据外键删除方法的名称可以采用动词（Delete）+名词（实体类名）或单个动词（Delete）+"By"+外键属性名称。
5. 根据主键查询单个实体方法的名称可以采用动词（Get）+名词（实体类名）+"By"+主键属性名称。
6. 根据外键查询单个实体方法的名称可以采用动词（Get）+名词（实体类名）+"By"+外键属性名称。
7. 查询所有实体的方法名称可以采用动词（Get）+名词（实体类目）+"s"。

参数列表：

1. 添加修改方法的参数列表一般为实体对象。
2. 根据主键删除方法的参数列表一般为主键属性。
3. 根据外键删除方法的参数列表一般为外键属性。
4. 根据主键查询单个实体方法的参数列表一般为主键属性。
5. 根据外键查询实体集合方法的参数列表一般为外键属性。
6. 查询所有实体方法的参数列表一般没有。

10.生成SQL脚本和数据

```
<%--
Name:产生数据库的SQL脚本
Author: Jack.zhou
Description:
--%>
<%@ CodeTemplate Language="C#" TargetLanguage="T-SQL" Src="" Inherits=""
Debug="False" Description="Template description here." ResponseEncoding="UTF-8"%>
<%@ Property Name="SourceDB" Type="SchemaExplorer.DatabaseSchema" Default=""
Optional="False" Category="Context" Description="" OnChanged="" Editor="" EditorBase=""
Serializer="" %>
<%@ Property Name="NewDBName" Type="System.String" Default="NewDB" Optional="False"
Category="Context" Description="" OnChanged="" Editor="" EditorBase="" Serializer="" %>
<%@ Property Name="ProcessData" Type="System.Boolean" Default="False" Optional="False"
```

```
Category="Context" Description="" OnChanged="" Editor="" EditorBase="" Serializer="" %>
<%@ Assembly Name="SchemaExplorer" %>
<%@ Assembly Name="SchemaExplorer.Trigger" %>
<%@ Import Namespace="SchemaExplorer" %>
<%@ Assembly Name="System.Data" %>
<%@ Import Namespace="System.Data" %>
Use master
Go
--region 判断数据库是否存在，如果存在，那么删除这个数据库
IF Exists(Select '' From sysdatabases Where name='<%=this.NewDBName%>')
Begin
 Drop Database [<%=this.NewDBName%>]
End
Go
--endregion
--region 创建数据库<%=this.NewDBName%>
Create Database [<%=this.NewDBName%>]
Go
Use [<%=this.NewDBName%>]
Go
--endregion

--region 创建自定义类型
<%
UserDefineTypeCollection udtc=new UserDefineTypeCollection (this.SourceDB);//自定义导入的
DLL，上例操作SQLDMO的一个类
foreach(UserDefineType udt in udtc.UserDefineTypes)
{
 //Response.WriteLine(udt.ToString());
}
%>
Go
--endregion

<%foreach(TableSchema table in this.SourceDB.Tables){%>
--region 判断数据表[<%=table.Name%>]是否存在，如果存在，那么删除这个数据表
If Exists(Select '' From sysobjects Where name='<%=table.Name%>')
Begin
 Drop Table [<%=table.Name%>]
End
Go
--endregion
<%}%>

<%foreach(TableSchema table in this.SourceDB.Tables){%>
--region 创建数据表<%=table.Name%>
```

```
Create Table [<%=table.Name%>]
(
 <%for(int i=0;i<table.Columns.Count;i++){%>
 <%if(i!=table.Columns.Count-1){%>
 <%=GetColumn(table.Columns[i])%>,
 <%}%>
 <%else{%>
 <%=GetColumn(table.Columns[i])%>
 <%}%>
 <%}%>
)
Go
--endregion
<%}%>

--region 添加数据
<%if(this.ProcessData){%>
use <%=this.SourceDB.Name%>New
Go
<%foreach(TableSchema table in this.SourceDB.Tables){%>
<%if(HasIdentity(table)){%>
DBCC CheckIdent ('[<%= table.Owner %>].[<%= table.Name %>]', reseed, 1)
Set Identity_Insert [<%=table.Owner%>].[<%=table.Name%>] On
<%}%>
<%DataTable dt=table.GetTableData();
foreach(DataRow dr in dt.Rows){%>
Insert Into [<%=table.Owner%>].[<%=table.Name%>](<%=GetTableColumnList(table)%>)
Values(<%=GetData(dr,table)%>)
 <%}%>
<%if(HasIdentity(table)){%>
Set Identity_Insert [<%=table.Owner%>].[<%=table.Name%>] Off
<%}%>
Go
<%}%>
<%}%>
--endregion

<%foreach(TableSchema table in this.SourceDB.Tables){%>
--region 为[<%=table.Name%>]添加主键约束
<%if(table.HasPrimaryKey){%>
Alter Table  [<%=table.Name%>] Add Constraint <%=table.PrimaryKey.Name%> Primary Key
(<%=GetPkString(table)%>)
<%}%>
Go
--endregion
<%}%>
```

```
<%foreach(TableSchema table in this.SourceDB.Tables){%>
--region 为[<%=table.Name%>]添加外键约束
<%foreach(TableKeySchema key in table.ForeignKeys){%>
Alter Table  [<%=table.Name%>] Add Constraint <%=key.Name%> Foreign Key <%
=GetFKString(key)%>
<%}%>
Go
--endregion
<%}%>

<%foreach(TableSchema table in this.SourceDB.Tables){%>
--region 为[<%=table.Name%>]添加其他约束
<%foreach(ColumnSchema column in table.Columns){%>
<%=GetOtherConstrain(column)%>
<%}%>
Go
--endregion
<%}%>

<%foreach(TableSchema table in this.SourceDB.Tables){%>
--region 为[<%=table.Name%>]添加索引
<%foreach(IndexSchema index in table.Indexes){%>
<%if(GetIndexType(index)!=""){%>
Create <%=GetIndexType(index)%> Index [<%=index.Name%>] On [<%
=index.Table.Name%>] (<%=GetIndexColumn(index)%>) <%=GetIndexOption(index)%>
<%}%>
<%}%>
Go
--endregion
<%}%>

<%foreach(ViewSchema view in this.SourceDB.Views){%>
--region 创建视图<%=view.Name%>
<%=view.ViewText%>
Go
--endregion
<%}%>

<%foreach(CommandSchema command in this.SourceDB.Commands){%>
--region 创建存储过程<%=command.Name%>
<%=command.CommandText%>
Go
--endregion
<%}%>
```

```
--region 创建触发器
<%foreach(TableSchema table in this.SourceDB.Tables){%>
<%TriggerCollection triggers=new TriggerCollection(table);%>
<%foreach(Trigger trigger in triggers.Triggers){%>
--region 创建<%=trigger.Name%>触发器
<%=trigger.Text%>
--endregion
<%}%>
Go
<%}%>--endregion

<script runat="template">
#region 生成数据库结构
public string GetIndexOption(IndexSchema index)
{
 string s="With ";
 if(int.Parse(index.ExtendedProperties["CS_OrigFillFactor"].Value.ToString())!=0)
 {
  s+="FillFactor="+int.Parse(index.ExtendedProperties["CS_OrigFillFactor"].Value.ToString())+",";
 }
 if(bool.Parse(index.ExtendedProperties["CS_PadIndex"].Value.ToString()))
 {
  s+="PAD_INDEX,";
 }
 if(bool.Parse(index.ExtendedProperties["CS_IgnoreDupKey"].Value.ToString()))
 {
  s+="IGNORE_DUP_KEY,";
 }
 if(bool.Parse(index.ExtendedProperties["CS_DropExist"].Value.ToString()))
 {
  s+="DROP_EXISTING,";
 }
 if(s.Length==5)
 {
  return "";
 }
 return s.Substring(0,s.Length-1);

}
public string GetIndexColumn(IndexSchema index)
{
 string s="";
 foreach(MemberColumnSchema column in index.MemberColumns)
 {
  if((int)column.ExtendedProperties["CS_IsDescending"].Value==0)
  {
```

```
   s+=column.Name+",";
  }
  else
  {
   s+=column.Name+" Desc,";
  }
 }
 if(s=="")
 {
  return s;
 }
 return s.Substring(0,s.Length-1);
}
public string GetIndexType(IndexSchema index)
{
 if(index.IsClustered==true && index.IsPrimaryKey==true && index.IsUnique==true)
 {
  return "";
 }
 else if(index.IsClustered==false && index.IsPrimaryKey==false && index.IsUnique==false)
 {
  return "NonClustered";
 }
 else if(index.IsClustered==false && index.IsPrimaryKey==false && index.IsUnique==true)
 {
  return "Unique";
 }
 return "";
}
public string GetColumn(ColumnSchema column)
{
 string s="";
 s+="["+column.Name+"] "+column.NativeType+" ";
 if(IsId(column))
 {
  s+="identity("+column.ExtendedProperties
["CS_IdentitySeed"].Value+","+column.ExtendedProperties["CS_IdentityIncrement"].Value+") ";
 }
 if(HasLength(column))
 {
  s+="("+column.Size+") ";
 }
 if(column.AllowDBNull)
 {
  s+="null ";
 }
```

```
 else
 {
  s+="not null";
 }
 return s;
}
public bool IsId(ColumnSchema column)
{
 return (bool)column.ExtendedProperties["CS_IsIdentity"].Value;
}
public bool HasLength(ColumnSchema column)
{
 switch(column.NativeType)
 {
  case "char":
  case "varchar":
  case "nchar":
  case "nvarchar":
   return true;
  default:
   return false;
 }
}
public string GetPkString(TableSchema table)
{
 string s="";
 foreach(MemberColumnSchema column in table.PrimaryKey.MemberColumns)
 {
  s+="["+column.Name+"]"+",";
 }
 return s.Substring(0,s.Length-1);
}
public string GetFKString(TableKeySchema key)
{
 string s="(";
 foreach(MemberColumnSchema column in key.ForeignKeyMemberColumns)
 {
  s+="["+column.Name+"],";
 }
 s=s.Substring(0,s.Length-1);
 s+=")";
 s+=" References ["+key.PrimaryKeyTable.Name+"](";
 foreach(MemberColumnSchema column in key.PrimaryKeyMemberColumns)
 {
  s+="["+column.Name+"],";
 }
```

```
 s=s.Substring(0,s.Length-1);
 s+=")";
 return s;
}
public string GetOtherConstrain(ColumnSchema column)
{
 string s="";
 int i=1;
 while(true)
 {
  if(column.ExtendedProperties["CS_Constraint"+i+"Name"]==null)
  {
   break;
  }

  if(column.ExtendedProperties["CS_Constraint"+i+"Type"].Value.ToString().ToUpper().Trim()
=="DEFAULT")
  {
   s+="Alter Table ["+column.Table.Name+"] Add Constraint "+column.ExtendedProperties
["CS_Constraint"+i+"Name"].Value+" Default "+column.ExtendedProperties
["CS_Constraint"+i+"Definition"].Value.ToString()+" For ["+column.Name+"]\r\n";
  }
  else
  {
   s+="Alter Table ["+column.Table.Name+"] Add Constraint "+column.ExtendedProperties
["CS_Constraint"+i+"Name"].Value+" Check "+column.ExtendedProperties
["CS_Constraint"+i+"Definition"].Value.ToString()+"\r\n";
  }
  i++;
 }
 if(s=="")
 {
  return null;
 }
 return s;
}
#endregion

#region 产生数据
public bool HasIdentity(TableSchema table)
{
 foreach(ColumnSchema column in table.Columns)
 {
  if((bool)column.ExtendedProperties["CS_IsIdentity"].Value)
  {
   return true;
```

```
  }
 }
 return false;
}
public string GetData(DataRow dr,TableSchema table)
{
 string s="";
 foreach(ColumnSchema column in table.Columns)
 {
  if(dr[column.Name] is System.DBNull)
  {
   s+="null,";
  }
  else
  {
   if(IsNumeric(column))
   {
    if(column.SystemType.ToString()!="System.Boolean")
    {
     s+=FormatString(dr[column.Name].ToString())+",";
    }
    else
    {
     s+= (((bool)dr[column.Name])==true?1:0) +",";
    }
   }
   else
   {
    if(column.SystemType.ToString()!="System.Byte[]")
    {
     s+="'"+FormatString(dr[column.Name].ToString())+"',";
    }
    else
    {
     byte[] bs=(byte[])dr[column.Name];
     string x="0x";
     foreach(byte b in bs)
     {
      x+=b.ToString("X");
     }
     s+=""+x+",";
    }
   }
  }

 }
```

```
 if(s.Length==0)
 {
  return s;
 }
 return s.Substring(0,s.Length-1);
}
public string FormatString(string s)
{
 return s.Replace("'","''");
}
public string GetTableColumnList(TableSchema table)
{
 string s="";
 foreach(ColumnSchema column in table.Columns)
 {
  s+="["+column.Name+"],";
 }
 if(s.Length==0)
 {
  return "";
 }
 return s.Substring(0,s.Length-1);
}
public bool IsNumeric(ColumnSchema column)
{
 switch(column.NativeType.ToLower())
 {
  case "int":
  case "smallint":
  case "bigint":
  case "float":
  case "decimal":
  case "money":
  case "numeric":
  case "real":
  case "smallmoney":
  case "tinyint":
  case "bit":
   return true;
  default:
   return false;
 }
}
#endregion

</script>
```

11.生成实体类

```
<%--
Name:实体类
Author: Jack.Zhou
Description:
--%>
<%@ CodeTemplate Language="C#" TargetLanguage="C#" Src="" Inherits="" Debug="False"
Description="Template description here." ResponseEncoding="UTF-8" %>
<%@ Property Name="SourceTable" Type="SchemaExplorer.TableSchema" Default=""
Optional="False" Category="Table" Description="源表名" OnChanged="" Editor="" EditorBase=""
Serializer="" %>
<%@ Property Name="NameSpace" Type="System.String" Default="Model" Optional="False"
Category="NameSpace" Description="命名空间" OnChanged="" Editor="" EditorBase=""
Serializer="" %>
<%@ Property Name="IsFK" Type="System.Boolean" Default="False" Optional="False"
Category="Other" Description="是否处理外键" OnChanged="" Editor="" EditorBase=""
Serializer="" %>
<%@ Property Name="Author" Type="System.String" Default="Jack.Zhou" Optional="False"
Category="Other" Description="" OnChanged="" Editor="" EditorBase="" Serializer="" %>
<%@ Assembly Name="SchemaExplorer" %>
<%@ Assembly Name="System.Data" %>
<%@ Assembly Name="mscorlib" %>
<%@ Import Namespace="SchemaExplorer" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Collections.Generic" %>
using System;
using System.Collections.Generic;
using System.Text;

namespace <%=this.NameSpace%>
{
 /// <summary>
 /// 实体类<%=this.GetClassName()%>
 /// </summary>
 public class <%=this.GetClassName()%>
 {
  #region 私有字段,外键
  <%foreach(ColumnSchema column in this.SourceTable.ForeignKeyColumns){%>
  <%if(!IsFK){%>
  private <%=this.GetCSharpVariableType(column)%> _<%=this.ToCamel(column.Name)%>;
  <%}else{%>
  private <%=this.GetFKClassName(column)%> _<%=this.ToCamel(column.Name)%>;
  <%}%>
```

```
<%}%>
<%foreach(ColumnSchema column in this.SourceTable.NonForeignKeyColumns){%>
private <%=this.GetCSharpVariableType(column)%> _<%=this.ToCamel(column.Name)%>;
<%}%>
#endregion

#region 公开属性
<%foreach(ColumnSchema column in this.SourceTable.ForeignKeyColumns){%>
<%if(!IsFK){%>
public <%=this.GetCSharpVariableType(column)%> <%=this.ToPascal(column.Name)%>
{
 get{return _<%=this.ToCamel(column.Name)%>;}
 set{_<%=this.ToCamel(column.Name)%>=value;}
}
<%}else{%>
public <%=this.GetFKClassName(column)%> <%=this.ToPascal(column.Name)%>
{
 get{return _<%=this.ToCamel(column.Name)%>;}
 set{_<%=this.ToCamel(column.Name)%>=value;}
}
<%}%>
<%}%>
<%foreach(ColumnSchema column in this.SourceTable.NonForeignKeyColumns){%>
public <%=this.GetCSharpVariableType(column)%> <%=this.ToPascal(column.Name)%>
{
 get{return _<%=this.ToCamel(column.Name)%>;}
 set{_<%=this.ToCamel(column.Name)%>=value;}
}
<%}%>
#endregion
 }
}
<script runat="template">
#region Pascal命名法
public string ToPascal(string s)
{
 return s.Substring(0,1).ToUpper()+s.Substring(1);
}
#endregion
#region 骆驼命名法
public string ToCamel(string s)
{
 return s.Substring(0,1).ToLower()+s.Substring(1);
}
#endregion
```

```
#region 获取实体类类名,去除表的复数形式S
public string GetClassName()
{
 string s=this.SourceTable.Name;
 if(s.EndsWith("s"))
 {
  s=s.Substring(0,s.Length-1);
 }
 return this.ToPascal(s);
}
public string GetClassName(TableSchema table)
{
 string s=table.Name;
 if(s.EndsWith("s"))
 {
  s=s.Substring(0,s.Length-1);
 }
 return this.ToPascal(s);
}
#endregion
#region 获取实体对象名
public string GetObjectName()
{
 return this.ToCamel(this.GetClassName());
}
#endregion
#region 获取文件名
public override string GetFileName()
{
 return this.GetClassName()+".cs";
}
#endregion
#region 获取列的数据类型
public string GetCSharpVariableType(ColumnSchema column)
{
 if (column.Name.EndsWith("TypeCode")) return column.Name;

 switch (column.DataType)
 {
  case DbType.AnsiString: return "string";
  case DbType.AnsiStringFixedLength: return "string";
  case DbType.Binary: return "byte[]";
  case DbType.Boolean: return "bool";
  case DbType.Byte: return "byte";
  case DbType.Currency: return "decimal";
  case DbType.Date: return "DateTime";
```

```
  case DbType.DateTime: return "DateTime";
  case DbType.Decimal: return "decimal";
  case DbType.Double: return "double";
  case DbType.Guid: return "Guid";
  case DbType.Int16: return "short";
  case DbType.Int32: return "int";
  case DbType.Int64: return "long";
  case DbType.Object: return "object";
  case DbType.SByte: return "sbyte";
  case DbType.Single: return "float";
  case DbType.String: return "string";
  case DbType.StringFixedLength: return "string";
  case DbType.Time: return "TimeSpan";
  case DbType.UInt16: return "ushort";
  case DbType.UInt32: return "uint";
  case DbType.UInt64: return "ulong";
  case DbType.VarNumeric: return "decimal";
  default:
  {
   return "__UNKNOWN__" + column.NativeType;
  }
 }
}
#endregion
#region 获取外键类名
public string GetFKClassName(ColumnSchema column)
{
 foreach(TableKeySchema key in this.SourceTable.ForeignKeys)
 {
  foreach(MemberColumnSchema fk in key.ForeignKeyMemberColumns)
  {
   if(fk.Name==column.Name)
   {
    return this.GetClassName(key.PrimaryKeyTable);
   }
  }
 }
 return "";
}
#endregion
</script>
```

12.数据访问类

返回类型：

1. 增删改方法的返回类型一般为int，用来表达影响的行数。
2. 查询单个实体方法的返回类型，一般为实体对象。
3. 查询实体集合的返回类型，一般为实体集合，可以使用泛型集合。

方法名称：

1. 添加方法的名称可以用动词（Add或Insert）+名词（实体类名）或单个动词（Add或Insert）。
2. 修改方法的名称可以用动词（Edit或Modify）+名词（实体类名）或单个动词（Edit或Modify）。
3. 根据主键删除方法的名称可以采用动词（Delete）+名词（实体类名）或单个动词（Delete）+"By"+主键属性名称。
4. 根据外键删除方法的名称可以采用动词（Delete）+名词（实体类名）或单个动词（Delete）+"By"+外键属性名称。
5. 根据主键查询单个实体方法的名称可以采用动词（Get）+名词（实体类名）+"By"+主键属性名称。
6. 根据外键查询单个实体方法的名称可以采用动词（Get）+名词（实体类名）+"By"+外键属性名称。
7. 查询所有实体的方法名称可以采用动词（Get）+名词（实体类目）+"s"。

参数列表：

1. 添加修改方法的参数列表一般为实体对象。
2. 根据主键删除方法的参数列表一般为主键属性。
3. 根据外键删除方法的参数列表一般为外键属性。
4. 根据主键查询单个实体方法的参数列表一般为主键属性。
5. 根据外键查询实体集合方法的参数列表一般为外键属性。
6. 查询所有实体方法的参数列表一般没有。

```
<%--
Name:生成数据访问类
Author: Jack.zhou
Description:
--%>
<%@ CodeTemplate Language="C#" TargetLanguage="C#" Src="" Inherits="" Debug="False"
Description="Template description here." ResponseEncoding="UTF-8" %>
<%@ Property Name="SourceTable" Type="SchemaExplorer.TableSchema" Default=""
Optional="False" Category="Table" Description="源表名" %>
<%@ Property Name="NameSpace" Type="System.String" Default="DAL" Optional="False"
Category="NameSpace" Description="命名空间" %>
<%@ Property Name="DALClassPostFix" Type="System.String" Default="Service"
Optional="False" Category="Other" Description="数据访问类名后缀"%>
<%@ Property Name="ModelNameSpace" Type="System.String" Default="Model"
Optional="False" Category="NameSpace" Description="实体类命名空间" %>
<%@ Property Name="DBHelpNameSpace" Type="System.String" Default="DAL"
Optional="False" Category="NameSpace" Description="数据执行类命名空间"%>
<%@ Property Name="IsFK" Type="System.Boolean" Default="False" Optional="False"
Category="Other" Description="是否处理外键"%>
<%@ Property Name="Author" Type="System.String" Default="Jack.Zhou" Optional="False"
Category="Other" Description="" %>
```

```
<%@ Assembly Name="SchemaExplorer" %>
<%@ Assembly Name="System.Data" %>
<%@ Assembly Name="mscorlib" %>
<%@ Import Namespace="SchemaExplorer" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Collections.Generic" %>

using System;
using System.Collections.Generic;
using System.Data;
using System.Data.SqlClient;
using <%=this.ModelNameSpace%>;//实体类
using <%=this.DBHelpNameSpace%>;
namespace <%=this.NameSpace%>
{
 public class <%=this.GetClassName()+this.DALClassPostFix%>
 {
  <%=this.GetAddMethodDefine()%>
  {
   <%=this.GetAddMethodBody()%>
  }

  <%if(this.SourceTable.PrimaryKeys.Count>0){%>
  <%=this.GetUpdateMethodDefine()%>
  {
   <%=this.GetUpdateMethodBody()%>
  }
  <%=this.GetDeleteMethodByPKDefine()%>
  {
   <%=this.GetDeleteMethodByPKBody()%>
  }
  <%=this.GetDeleteMethodByPKDefine2()%>
  {
   <%=this.GetDeleteMethodByPKBody2()%>
  }
  <%}%>
  <%foreach(ColumnSchema column in this.SourceTable.ForeignKeyColumns){%>
  <%=this.GetDeleteMethodByFKDefine(column)%>
  {
   <%=this.GetDeleteMethodByFKBody(column)%>
  }
  <%}%>

  <%=this.GetPrivateQueryMethodDefine()%>
  {
   <%=this.GetPrivateQueryMethodBody()%>
```

```
  }

  <%=this.GetQueryAllMethodDefine()%>
  {
   <%=this.GetQueryAllMethodBody()%>
  }

  <%if(this.SourceTable.PrimaryKeys.Count>0){%>
  <%=this.GetQueryMethodByPKDefine()%>
  {
   <%=this.GetQueryMethodByPKBody()%>
  }
  <%}%>
  <%foreach(ColumnSchema column in this.SourceTable.ForeignKeyColumns){%>
  <%=this.GetQueryMethodByFKDefine(column)%>
  {
   <%=this.GetQueryMethodByFKBody(column)%>
  }
  <%}%>
 }

}
<script runat="template">
const string Enter="\r\n";
const string Tab="\t";
#region Pascal命名法
public string ToPascal(string s)
{
 return s.Substring(0,1).ToUpper()+s.Substring(1);
}
#endregion
#region 骆驼命名法
public string ToCamel(string s)
{
 return s.Substring(0,1).ToLower()+s.Substring(1);
}
#endregion

#region 获取实体类类名
public string GetClassName()
{
 string s=this.SourceTable.Name;
 if(s.EndsWith("s"))
 {
  s=s.Substring(0,s.Length-1);
 }
```

```
 return this.ToPascal(s);
}
public string GetClassName(TableSchema table)
{
 string s=table.Name;
 if(s.EndsWith("s"))
 {
  s=s.Substring(0,s.Length-1);
 }
 return this.ToPascal(s);
}
#endregion
#region 获取实体对象名
public string GetObjectName()
{
 return this.ToCamel(this.GetClassName());
}
#endregion
#region 获取文件名
public override string GetFileName()
{
 return this.GetClassName()+this.DALClassPostFix+".cs";
}
#endregion
#region 获取列的数据类型
public string GetCSharpVariableType(ColumnSchema column)
{
 if (column.Name.EndsWith("TypeCode")) return column.Name;

 switch (column.DataType)
 {
  case DbType.AnsiString: return "string";
  case DbType.AnsiStringFixedLength: return "string";
  case DbType.Binary: return "byte[]";
  case DbType.Boolean: return "bool";
  case DbType.Byte: return "byte";
  case DbType.Currency: return "decimal";
  case DbType.Date: return "DateTime";
  case DbType.DateTime: return "DateTime";
  case DbType.Decimal: return "decimal";
  case DbType.Double: return "double";
  case DbType.Guid: return "Guid";
  case DbType.Int16: return "short";
  case DbType.Int32: return "int";
  case DbType.Int64: return "long";
  case DbType.Object: return "object";
```

```
  case DbType.SByte: return "sbyte";
  case DbType.Single: return "float";
  case DbType.String: return "string";
  case DbType.StringFixedLength: return "string";
  case DbType.Time: return "TimeSpan";
  case DbType.UInt16: return "ushort";
  case DbType.UInt32: return "uint";
  case DbType.UInt64: return "ulong";
  case DbType.VarNumeric: return "decimal";
  default:
  {
   return "__UNKNOWN__" + column.NativeType;
  }
 }
}
#endregion
#region 获取外键类名
public string GetFKClassName(ColumnSchema column)
{
 foreach(TableKeySchema key in this.SourceTable.ForeignKeys)
 {
  foreach(MemberColumnSchema fk in key.ForeignKeyMemberColumns)
  {
   if(fk.Name==column.Name)
   {
    return this.GetClassName(key.PrimaryKeyTable);
   }
  }
 }
 return "";
}
#endregion

#region 添加方法定义 Add(User user)
public string GetAddMethodDefine()
{
 string para=string.Format("({0} {1})",this.GetClassName(),this.GetObjectName());
 string s="public static int Add"+this.GetClassName()+para;
 return s;
}
#endregion
#region 添加方法体
public string GetAddMethodBody()
{
 string fieldList="";
 foreach(ColumnSchema column in this.GetInsertColumn())
```

```
 {
  fieldList+=column.Name+",";
 }
 fieldList=fieldList.Substring(0,fieldList.Length-1);

 string valueList="";
 foreach(ColumnSchema column in this.GetInsertColumn())
 {
  valueList+="@"+column.Name+",";
 }
 valueList=valueList.Substring(0,valueList.Length-1);

 string sql=string.Format("string sql=\"Insert Into {0}({1}) Values({2})
\";"+Enter,this.SourceTable.Name,fieldList,valueList);

 string sqlPara=Tab+Tab+Tab+"SqlParameter[] ps={"+Enter;
 foreach(ColumnSchema column in this.GetInsertColumn())
 {
  sqlPara+=Tab+Tab+Tab+Tab+"new SqlParameter
(\"@"+column.Name+"\","+this.GetObjectName()+"."+this.GetPropertyName(column)
+"),"+Enter;
 }
 sqlPara=sqlPara.Substring(0,sqlPara.Length-3)+"};"+Enter;
 string exeString=Tab+Tab+Tab+"return DBHelper.ExecuteCommand(sql,ps);"+Enter;
 return sql+sqlPara+exeString;
}
#endregion

#region 获取应该插入值的列,CS_IsIdentity字段是否是标识列即字段
public List<ColumnSchema> GetInsertColumn()
{
 List<ColumnSchema> list=new List<ColumnSchema>();
 foreach(ColumnSchema column in this.SourceTable.Columns)
 {
  if(!(bool)column.ExtendedProperties["CS_IsIdentity"].Value)
  {
   list.Add(column);
  }
 }
 return list;
}
#endregion

#region 获取属性名称
public string GetPropertyName(ColumnSchema column)
{
```

```
 if(!IsFK)
 {
  return this.ToPascal(column.Name);
 }
 if(!column.IsForeignKeyMember)
 {
  return this.ToPascal(column.Name);
 }
 else
 {
  return this.ToPascal(column.Name)+"."+GetFKPropertyName(column);
 }
}
#endregion
#region 获取外键属性名称
public string GetFKPropertyName(ColumnSchema column)
{
 foreach(TableKeySchema key in this.SourceTable.ForeignKeys)
 {
  foreach(MemberColumnSchema fk in key.ForeignKeyMemberColumns)
  {
   if(fk.Name==column.Name)
   {
    return this.ToPascal(key.PrimaryKeyMemberColumns[0].Name);
   }
  }
 }
 return "";
}
#endregion

#region 更新方法定义，UpdateUser(User user)
public string GetUpdateMethodDefine()
{
 string para=string.Format("({0} {1})",this.GetClassName(),this.GetObjectName());
 string s="public static int Update"+this.GetClassName()+para;
 return s;
}
#endregion
#region 更新方法体
public string GetUpdateMethodBody()
{
 string updateList="";
 foreach(ColumnSchema column in this.SourceTable.NonPrimaryKeyColumns)
 {
  updateList+=column.Name+"=@"+column.Name+",";
```

```
 }
 updateList=updateList.Substring(0,updateList.Length-1);

 string whereList="";
 foreach(ColumnSchema column in this.SourceTable.Columns)
 {
  if(column.IsPrimaryKeyMember)
  {
   whereList+=column.Name+"=@"+column.Name+" And";
  }
 }
 whereList=whereList.Substring(0,whereList.Length-3);

 string sql=string.Format("string sql=\"Update {0} Set {1} Where {2}
\";"+Enter,this.SourceTable.Name,updateList,whereList);

 string sqlPara=Tab+Tab+Tab+"SqlParameter[] ps={"+Enter;
 foreach(ColumnSchema column in this.SourceTable.Columns)
 {
  sqlPara+=Tab+Tab+Tab+Tab+"new SqlParameter
(\"@"+column.Name+"\","+this.GetObjectName()+"."+this.GetPropertyName(column)
+"),"+Enter;
 }
 sqlPara=sqlPara.Substring(0,sqlPara.Length-3)+"};"+Enter;
 string exeString=Tab+Tab+Tab+"return DBHelper.ExecuteCommand(sql,ps);"+Enter;
 return sql+sqlPara+exeString;
}
#endregion

#region 根据主键删除方法定义
public string GetDeleteMethodByPKDefine()
{
 string para=string.Format("({0} {1})",this.GetClassName(),this.GetObjectName());
 string s="public static int Delete"+this.GetClassName()+para;
 return s;
}
#endregion
#region 根据主键删除方法体
public string GetDeleteMethodByPKBody()
{

 string whereList="";
 foreach(ColumnSchema column in this.SourceTable.Columns)
 {
  if(column.IsPrimaryKeyMember)
  {
```

```
   whereList+=column.Name+"=@"+column.Name+" And";
  }
 }
 whereList=whereList.Substring(0,whereList.Length-3);

 string sql=string.Format("string sql=\"Delete From {0} Where {1}
\";"+Enter,this.SourceTable.Name,whereList);

 string sqlPara=Tab+Tab+Tab+"SqlParameter[] ps={"+Enter;
 foreach(ColumnSchema column in this.SourceTable.Columns)
 {
  if(column.IsPrimaryKeyMember)
  {
   sqlPara+=Tab+Tab+Tab+Tab+"new SqlParameter
(\"@"+column.Name+"\","+this.GetObjectName()+"."+this.GetPropertyName(column)
+"),"+Enter;
  }
 }
 sqlPara=sqlPara.Substring(0,sqlPara.Length-3)+"};"+Enter;
 string exeString=Tab+Tab+Tab+"return DBHelper.ExecuteCommand(sql,ps);"+Enter;
 return sql+sqlPara+exeString;
}
#endregion

#region 重载根据主键删除方法定义
public string GetDeleteMethodByPKDefine2()
{
 string para="";
 foreach(ColumnSchema column in this.SourceTable.Columns)
 {
  if(column.IsPrimaryKeyMember)
  {
   para+=this.GetCSharpVariableType(column)+" "+column.Name+",";
  }
 }
 para=para.Substring(0,para.Length-1);
 para=string.Format("({0})",para);
 string s="public static int Delete"+this.GetClassName()+para;
 return s;
}
#endregion
#region 重载根据主键删除方法体
public string GetDeleteMethodByPKBody2()
{
 string whereList="";
 foreach(ColumnSchema column in this.SourceTable.Columns)
```

```
 {
  if(column.IsPrimaryKeyMember)
  {
   whereList+=column.Name+"=@"+column.Name+" And";
  }
 }
 whereList=whereList.Substring(0,whereList.Length-3);

 string sql=string.Format("string sql=\"Delete From {0} Where {1}
\";"+Enter,this.SourceTable.Name,whereList);

 string sqlPara=Tab+Tab+Tab+"SqlParameter[] ps={"+Enter;
 foreach(ColumnSchema column in this.SourceTable.Columns)
 {
  if(column.IsPrimaryKeyMember)
  {
   sqlPara+=Tab+Tab+Tab+Tab+"new SqlParameter
(\"@"+column.Name+"\","+column.Name+"),"+Enter;
  }
 }
 sqlPara=sqlPara.Substring(0,sqlPara.Length-3)+"};"+Enter;
 string exeString=Tab+Tab+Tab+"return DBHelper.ExecuteCommand(sql,ps);"+Enter;
 return sql+sqlPara+exeString;
}
#endregion

#region 根据外键进行删除方法定义
public string GetDeleteMethodByFKDefine(ColumnSchema column)
{
 string para="";
 para=this.GetCSharpVariableType(column)+" "+column.Name+",";
 para=para.Substring(0,para.Length-1);
 para=string.Format("({0})",para);
 string s="public static int Delete"+this.GetClassName()+"By"+this.ToPascal(column.Name)+para;
 return s;
}
#endregion
#region 根据外键进行删除方法体
public string GetDeleteMethodByFKBody(ColumnSchema column)
{
 string whereList="";

 whereList=column.Name+"=@"+column.Name+" And";
 whereList=whereList.Substring(0,whereList.Length-3);

 string sql=string.Format("string sql=\"Delete From {0} Where {1}
```

```
\";"+Enter,this.SourceTable.Name,whereList);

 string sqlPara=Tab+Tab+Tab+"SqlParameter[] ps={"+Enter;
 sqlPara+=Tab+Tab+Tab+Tab+"new SqlParameter
(\"@"+column.Name+"\","+column.Name+"),"+Enter;
 sqlPara=sqlPara.Substring(0,sqlPara.Length-3)+"};"+Enter;
 string exeString=Tab+Tab+Tab+"return DBHelper.ExecuteCommand(sql,ps);"+Enter;
 return sql+sqlPara+exeString;
}
#endregion

#region 私有查询方法定义
public string GetPrivateQueryMethodDefine()
{
 string s=string.Format("private static List<{0}> Get{1}s(string sql,SqlParameter[]
ps)",this.GetClassName(),this.GetClassName());
 return s;
}
#endregion
#region 私有查询方法定义
public string GetPrivateQueryMethodBody()
{
 string s="DataTable dt=DBHelper.GetTable(sql,ps);"+Enter;
 s+=Tab+Tab+Tab+string.Format("List<{0}> list=new List<{1}>();",this.GetClassName
(),this.GetClassName())+Enter;
 s+=Tab+Tab+Tab+"foreach(DataRow dr in dt.Rows)"+Enter;
 s+=Tab+Tab+Tab+"{"+Enter;
 s+=Tab+Tab+Tab+Tab+string.Format("{0} {1}=new {2}();",this.GetClassName
(),this.GetObjectName(),this.GetClassName())+Enter;
 foreach(ColumnSchema column in this.SourceTable.Columns)
 {
  if(!IsFK || !column.IsForeignKeyMember)
  {
   s+=Tab+Tab+Tab+Tab+string.Format("{0}.{1}=({2})dr[\"{3}\"];",this.GetObjectName
(),this.ToPascal(column.Name),this.GetCSharpVariableType(column),column.Name)+Enter;
  }
  else
  {
   s+=Tab+Tab+Tab+Tab+string.Format("{0}.{1}={2}{3}.Get{4}(({5})dr[\"{6}
\"]);",this.GetObjectName(),this.ToPascal(column.Name),this.GetFKClassName
(column),this.DALClassPostFix,this.GetFKClassName(column),this.GetCSharpVariableType
(column),column.Name)+Enter;
  }
 }
 s+=Tab+Tab+Tab+Tab+string.Format("list.Add({0});",this.GetObjectName())+Enter;
 s+=Tab+Tab+Tab+"}"+Enter;
```

```
 s+=Tab+Tab+Tab+"return list;"+Enter;
 return s;
}
#endregion

#region 查询所有对象方法定义
public string GetQueryAllMethodDefine()
{
 string s=string.Format("public static List<{0}> GetAll{1}()",this.GetClassName
(),this.GetClassName());
 return s;
}
#endregion
#region 查询所有对象方法体
public string GetQueryAllMethodBody()
{
 string sql=string.Format("string sql=\"Select * From {0}\";",this.SourceTable.Name)+Enter;
 string para=Tab+Tab+Tab+"SqlParameter[] ps=new SqlParameter[0];"+Enter;
 string executeString=Tab+Tab+Tab+string.Format("return Get{0}",this.GetClassName())+"s
(sql,ps);"+Enter;
 return sql+para+executeString;
}
#endregion

#region 根据主键查询单个对象方法定义
public string GetQueryMethodByPKDefine()
{
 string para="";
 string name="";
 foreach(ColumnSchema column in this.SourceTable.Columns)
 {
  if(column.IsPrimaryKeyMember)
  {
   para+=this.GetCSharpVariableType(column)+" "+column.Name+",";
   name+=this.ToPascal(column.Name)+"And";
  }
 }
 para=para.Substring(0,para.Length-1);
 name=name.Substring(0,name.Length-3);

 string s=string.Format("public static {0} Get{1}By{2}({3})",this.GetClassName
(),this.GetClassName(),name,para);
 return s;
}
#endregion
#region 根据主键查询单个对象方法体
```

```
public string GetQueryMethodByPKBody()
{
 string whereList="";
 foreach(ColumnSchema column in this.SourceTable.Columns)
 {
  if(column.IsPrimaryKeyMember)
  {
   whereList+=column.Name+"=@"+column.Name+" And";
  }
 }
 whereList=whereList.Substring(0,whereList.Length-3);

 string sql=string.Format("string sql=\"Select * From {0} Where {1}
\";"+Enter,this.SourceTable.Name,whereList);

 string sqlPara=Tab+Tab+Tab+"SqlParameter[] ps={"+Enter;
 foreach(ColumnSchema column in this.SourceTable.Columns)
 {
  if(column.IsPrimaryKeyMember)
  {
   sqlPara+=Tab+Tab+Tab+Tab+"new SqlParameter
(\"@"+column.Name+"\","+column.Name+"),"+Enter;
  }
 }
 sqlPara=sqlPara.Substring(0,sqlPara.Length-3)+"};"+Enter;
 string exeString=Tab+Tab+Tab+string.Format("List<{0}> list=Get{0}s
(sql,ps);",this.GetClassName())+Enter;
 exeString+=Tab+Tab+Tab+"if(list.Count==0)"+Enter;
 exeString+=Tab+Tab+Tab+"{"+Enter;
 exeString+=Tab+Tab+Tab+Tab+"return null;"+Enter;
 exeString+=Tab+Tab+Tab+"}"+Enter;
 exeString+=Tab+Tab+Tab+"return list[0];"+Enter;
 return sql+sqlPara+exeString;

}
#endregion

#region 根据外键查询对象集合方法定义
public string GetQueryMethodByFKDefine(ColumnSchema column)
{
 string para="";
 string name="";

 para+=this.GetCSharpVariableType(column)+" "+column.Name+",";
 name+=this.ToPascal(column.Name)+"And";
 para=para.Substring(0,para.Length-1);
```

```
 name=name.Substring(0,name.Length-3);

 string s=string.Format("public static List<{0}> Get{1}By{2}({3})",this.GetClassName
(),this.GetClassName(),name,para);
 return s;
}
#endregion
#region 根据外键查询对象集合方法体
public string GetQueryMethodByFKBody(ColumnSchema column)
{
 string whereList="";

 whereList+=column.Name+"=@"+column.Name+" And";
 whereList=whereList.Substring(0,whereList.Length-3);

 string sql=string.Format("string sql=\"Select * From {0} Where {1}
\";"+Enter,this.SourceTable.Name,whereList);

 string sqlPara=Tab+Tab+Tab+"SqlParameter[] ps={"+Enter;

 sqlPara+=Tab+Tab+Tab+Tab+"new SqlParameter
(\"@"+column.Name+"\","+column.Name+"),"+Enter;
 sqlPara=sqlPara.Substring(0,sqlPara.Length-3)+"};"+Enter;
 string exeString=Tab+Tab+Tab+string.Format("List<{0}> list=Get{0}s
(sql,ps);",this.GetClassName())+Enter;
 exeString+=Tab+Tab+Tab+"return list;"+Enter;
 return sql+sqlPara+exeString;
}
#endregion


</script>
```

13.业务类

```
<%--
Name:生成业务类
Author: Jack.zhou
Description:
--%>
<%@ CodeTemplate Language="C#" TargetLanguage="C#" Src="" Inherits="" Debug="False"
Description="Template description here." ResponseEncoding="UTF-8" %>
<%@ Property Name="SourceTable" Type="SchemaExplorer.TableSchema" Default=""
Optional="False" Category="Table" Description="源表名" OnChanged="" Editor="" EditorBase=""
```

```
Serializer="" %>
<%@ Property Name="NameSpace" Type="System.String" Default="BLL" Optional="False"
Category="NameSpace" Description="命名空间" OnChanged="" Editor="" EditorBase=""
Serializer="" %>
<%@ Property Name="DALClassPostFix" Type="System.String" Default="Service"
Optional="False" Category="Other" Description="数据访问类名后缀" OnChanged="" Editor=""
EditorBase="" Serializer="" %>
<%@ Property Name="BLLClassPostFix" Type="System.String" Default="Manager"
Optional="False" Category="Other" Description="业务类名后缀" OnChanged="" Editor=""
EditorBase="" Serializer="" %>
<%@ Property Name="ModelNameSpace" Type="System.String" Default="Model"
Optional="False" Category="NameSpace" Description="实体类命名空间" OnChanged="" Editor=""
EditorBase="" Serializer="" %>
<%@ Property Name="DALNameSpace" Type="System.String" Default="DAL" Optional="False"
Category="NameSpace" Description="数据访问类命名空间" OnChanged="" Editor=""
EditorBase="" Serializer="" %>
<%@ Property Name="IsFK" Type="System.Boolean" Default="False" Optional="False"
Category="Other" Description="是否处理外键" OnChanged="" Editor="" EditorBase=""
Serializer="" %>
<%@ Property Name="Author" Type="System.String" Default="Jack.Zhou" Optional="False"
Category="Other" Description="" OnChanged="" Editor="" EditorBase="" Serializer="" %>
<%@ Assembly Name="SchemaExplorer" %>
<%@ Assembly Name="System.Data" %>
<%@ Assembly Name="mscorlib" %>
<%@ Import Namespace="SchemaExplorer" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Collections.Generic" %>

using System;
using System.Collections.Generic;
using System.Data;
using System.Data.SqlClient;
using <%=this.ModelNameSpace%>;
using <%=this.DALNameSpace%>;

namespace <%=this.NameSpace%>
{
 public class <%=this.GetClassName()+this.BLLClassPostFix%>
 {

  <%=this.GetAddMethodDefine()%>
  {
   <%=this.GetAddMethodBody()%>
  }
  <%if(this.SourceTable.PrimaryKeys.Count>0){%>
  <%=this.GetUpdateMethodDefine()%>
```

```
 {
  <%=this.GetUpdateMethodBody()%>
 }
 <%=this.GetDeleteMethodByPKDefine()%>
 {
  <%=this.GetDeleteMethodByPKBody()%>
 }
 <%=this.GetDeleteMethodByPKDefine2()%>
 {
  <%=this.GetDeleteMethodByPKBody2()%>
 }
 <%}%>
 <%foreach(ColumnSchema column in this.SourceTable.ForeignKeyColumns){%>
 <%=this.GetDeleteMethodByFKDefine(column)%>
 {
  <%=this.GetDeleteMethodByFKBody(column)%>
 }
 <%}%>
 <%=this.GetQueryAllMethodDefine()%>
 {
  <%=this.GetQueryAllMethodBody()%>
 }
 <%if(this.SourceTable.PrimaryKeys.Count>0){%>
 <%=this.GetQueryMethodByPKDefine()%>
 {
  <%=this.GetQueryMethodByPKBody()%>
 }
 <%}%>
 <%foreach(ColumnSchema column in this.SourceTable.ForeignKeyColumns){%>
 <%=this.GetQueryMethodByFKDefine(column)%>
 {
  <%=this.GetQueryMethodByFKBody(column)%>
 }
 <%}%>
}

}
<script runat="template">
const string Enter="\r\n";
const string Tab="\t";
#region Pascal命名法
public string ToPascal(string s)
{
 return s.Substring(0,1).ToUpper()+s.Substring(1);
}
#endregion
```

```
#region 骆驼命名法
public string ToCamel(string s)
{
 return s.Substring(0,1).ToLower()+s.Substring(1);
}
#endregion

#region 获取实体类类名
public string GetClassName()
{
 string s=this.SourceTable.Name;
 if(s.EndsWith("s"))
 {
  s=s.Substring(0,s.Length-1);
 }
 return this.ToPascal(s);
}
public string GetClassName(TableSchema table)
{
 string s=table.Name;
 if(s.EndsWith("s"))
 {
  s=s.Substring(0,s.Length-1);
 }
 return this.ToPascal(s);
}
#endregion
#region 获取实体对象名
public string GetObjectName()
{
 return this.ToCamel(this.GetClassName());
}
#endregion
#region 获取文件名
public override string GetFileName()
{
 return this.GetClassName()+this.BLLClassPostFix+".cs";
}
#endregion
#region 获取列的数据类型
public string GetCSharpVariableType(ColumnSchema column)
{
 if (column.Name.EndsWith("TypeCode")) return column.Name;

 switch (column.DataType)
 {
```

```
  case DbType.AnsiString: return "string";
  case DbType.AnsiStringFixedLength: return "string";
  case DbType.Binary: return "byte[]";
  case DbType.Boolean: return "bool";
  case DbType.Byte: return "byte";
  case DbType.Currency: return "decimal";
  case DbType.Date: return "DateTime";
  case DbType.DateTime: return "DateTime";
  case DbType.Decimal: return "decimal";
  case DbType.Double: return "double";
  case DbType.Guid: return "Guid";
  case DbType.Int16: return "short";
  case DbType.Int32: return "int";
  case DbType.Int64: return "long";
  case DbType.Object: return "object";
  case DbType.SByte: return "sbyte";
  case DbType.Single: return "float";
  case DbType.String: return "string";
  case DbType.StringFixedLength: return "string";
  case DbType.Time: return "TimeSpan";
  case DbType.UInt16: return "ushort";
  case DbType.UInt32: return "uint";
  case DbType.UInt64: return "ulong";
  case DbType.VarNumeric: return "decimal";
  default:
  {
   return "__UNKNOWN__" + column.NativeType;
  }
 }
}
#endregion
#region 获取外键类名
public string GetFKClassName(ColumnSchema column)
{
 foreach(TableKeySchema key in this.SourceTable.ForeignKeys)
 {
  foreach(MemberColumnSchema fk in key.ForeignKeyMemberColumns)
  {
   if(fk.Name==column.Name)
   {
    return this.GetClassName(key.PrimaryKeyTable);
   }
  }
 }
 return "";
}
```

```
#endregion

#region 添加方法定义,AddUser(User user)
public string GetAddMethodDefine()
{
 string para=string.Format("({0} {1})",this.GetClassName(),this.GetObjectName());
 string s="public static int Add"+this.GetClassName()+para;
 return s;
}
#endregion
#region 添加方法体,return UserDAL.User.AddUser(user)
public string GetAddMethodBody()
{
 string s=string.Format("return {0}{1}.Add{2}({3});",this.GetClassName
(),this.DALClassPostFix,this.GetClassName(),this.GetObjectName());
 return s;
}
#endregion

#region 获取应该插入值的列
public List<ColumnSchema> GetInsertColumn()
{
 List<ColumnSchema> list=new List<ColumnSchema>();
 foreach(ColumnSchema column in this.SourceTable.Columns)
 {
  if(!(bool)column.ExtendedProperties["CS_IsIdentity"].Value)
  {
   list.Add(column);
  }
 }
 return list;
}
#endregion

#region 获取属性名称
public string GetPropertyName(ColumnSchema column)
{
 if(!IsFK)
 {
  return this.ToPascal(column.Name);
 }
 if(!column.IsForeignKeyMember)
 {
  return this.ToPascal(column.Name);
 }
 else
```

```
 {
  return this.ToPascal(column.Name)+"."+GetFKPropertyName(column);
 }

}
#endregion
#region 获取外键属性名称
public string GetFKPropertyName(ColumnSchema column)
{
 foreach(TableKeySchema key in this.SourceTable.ForeignKeys)
 {
  foreach(MemberColumnSchema fk in key.ForeignKeyMemberColumns)
  {
   if(fk.Name==column.Name)
   {
    return this.ToPascal(key.PrimaryKeyMemberColumns[0].Name);
   }
  }
 }
 return "";
}
#endregion

#region 更新方法定义
public string GetUpdateMethodDefine()
{
 string para=string.Format("({0} {1})",this.GetClassName(),this.GetObjectName());
 string s="public static int Update"+this.GetClassName()+para;
 return s;
}
#endregion
#region 更新方法体
public string GetUpdateMethodBody()
{
 string s=string.Format("return {0}{1}.Update{2}({3});",this.GetClassName
(),this.DALClassPostFix,this.GetClassName(),this.GetObjectName());
 return s;
}
#endregion

#region 根据主键删除方法定义
public string GetDeleteMethodByPKDefine()
{
 string para=string.Format("({0} {1})",this.GetClassName(),this.GetObjectName());
 string s="public static int Delete"+this.GetClassName()+para;
 return s;
```

```
}
#endregion
#region 根据主键删除方法体
public string GetDeleteMethodByPKBody()
{

 string s=string.Format("return {0}{1}.Delete{2}({3});",this.GetClassName
(),this.DALClassPostFix,this.GetClassName(),this.GetObjectName());
 return s;
}
#endregion

#region 重载根据主键删除方法定义
public string GetDeleteMethodByPKDefine2()
{
 string para="";
 foreach(ColumnSchema column in this.SourceTable.Columns)
 {
  if(column.IsPrimaryKeyMember)
  {
   para+=this.GetCSharpVariableType(column)+" "+column.Name+",";
  }
 }
 para=para.Substring(0,para.Length-1);
 para=string.Format("({0})",para);
 string s="public static int Delete"+this.GetClassName()+para;
 return s;
}
#endregion
#region 重载根据主键删除方法体
public string GetDeleteMethodByPKBody2()
{
 string para="";
 foreach(ColumnSchema column in this.SourceTable.Columns)
 {
  if(column.IsPrimaryKeyMember)
  {
   para+=column.Name+",";
  }
 }
 para=para.Substring(0,para.Length-1);
 string s=string.Format("return {0}{1}.Delete{2}({3});",this.GetClassName
(),this.DALClassPostFix,this.GetClassName(),para);
 return s;
}
#endregion
```

```
#region 根据外键进行删除方法定义
public string GetDeleteMethodByFKDefine(ColumnSchema column)
{
 string para="";
 para=this.GetCSharpVariableType(column)+" "+column.Name+",";
 para=para.Substring(0,para.Length-1);
 para=string.Format("({0})",para);
 string s="public static int Delete"+this.GetClassName()+"By"+this.ToPascal(column.Name)+para;
 return s;
}
#endregion
#region 根据外键进行删除方法体
public string GetDeleteMethodByFKBody(ColumnSchema column)
{
 string para="";
 para+=column.Name+",";
 para=para.Substring(0,para.Length-1);
 string s=string.Format("return {0}{1}.Delete{2}By{3}({4});",this.GetClassName
(),this.DALClassPostFix,this.GetClassName(),this.ToPascal(column.Name),para);
 return s;
}
#endregion

#region 查询所有对象方法定义
public string GetQueryAllMethodDefine()
{
 string s=string.Format("public static List<{0}> GetAll{1}()",this.GetClassName
(),this.GetClassName());
 return s;
}
#endregion
#region 查询所有对象方法体
public string GetQueryAllMethodBody()
{
 string s=string.Format("return {0}{1}.GetAll{2}();",this.GetClassName
(),this.DALClassPostFix,this.GetClassName());
 return s;
}
#endregion

#region 根据主键查询单个对象方法定义
public string GetQueryMethodByPKDefine()
{
 string para="";
 string name="";
```

```
 foreach(ColumnSchema column in this.SourceTable.Columns)
 {
  if(column.IsPrimaryKeyMember)
  {
   para+=this.GetCSharpVariableType(column)+" "+column.Name+",";
   name+=this.ToPascal(column.Name)+"And";
  }
 }
 para=para.Substring(0,para.Length-1);
 name=name.Substring(0,name.Length-3);

 string s=string.Format("public static {0} Get{1}By{2}({3})",this.GetClassName
(),this.GetClassName(),name,para);
 return s;
}
#endregion
#region 根据主键查询单个对象方法体
public string GetQueryMethodByPKBody()
{
 string para="";
 string name="";
 foreach(ColumnSchema column in this.SourceTable.Columns)
 {
  if(column.IsPrimaryKeyMember)
  {
   para+=column.Name+",";
   name+=this.ToPascal(column.Name)+"And";
  }
 }
 para=para.Substring(0,para.Length-1);
 name=name.Substring(0,name.Length-3);
 string s=string.Format("return {0}{1}.Get{2}By{3}({4});",this.GetClassName
(),this.DALClassPostFix,this.GetClassName(),name,para);
 return s;
}
#endregion

#region 根据外键查询对象集合方法定义
public string GetQueryMethodByFKDefine(ColumnSchema column)
{
 string para="";
 string name="";

 para+=this.GetCSharpVariableType(column)+" "+column.Name+",";
 name+=this.ToPascal(column.Name)+"And";
 para=para.Substring(0,para.Length-1);
```

```
 name=name.Substring(0,name.Length-3);

 string s=string.Format("public static List<{0}> Get{1}By{2}({3})",this.GetClassName
();,this.GetClassName(),name,para);
 return s;
}
#endregion
#region 根据外键查询对象集合方法体
public string GetQueryMethodByFKBody(ColumnSchema column)
{

 string para="";
 para+=column.Name+",";
 para=para.Substring(0,para.Length-1);
 string s=string.Format("return {0}{1}.Get{2}By{3}({4});",this.GetClassName
();,this.DALClassPostFix,this.GetClassName(),this.ToPascal(column.Name),para);
 return s;
}
#endregion
</script>
```

14.自动生成

```
<%--
Name:自动生成代码
Author: Jack.Zhou
Description:
--%>
<%@ CodeTemplate Language="C#" TargetLanguage="C#" Src="" Inherits="" Debug="False"
Description="Template description here." ResponseEncoding="UTF-8" %>
<%@ Property Name="SourceDB" Type="SchemaExplorer.DatabaseSchema" Default=""
Optional="False" Category="Table" Description="源数据库" OnChanged="" Editor=""
EditorBase="" Serializer="" %>
<%@ Property Name="DALClassPostFix" Type="System.String" Default="Service"
Optional="False" Category="Other" Description="数据访问类名后缀" OnChanged="" Editor=""
EditorBase="" Serializer="" %>
<%@ Property Name="BLLClassPostFix" Type="System.String" Default="Manager"
Optional="False" Category="Other" Description="业务类名后缀" OnChanged="" Editor=""
EditorBase="" Serializer="" %>
<%@ Property Name="ModelNameSpace" Type="System.String" Default="Model"
Optional="False" Category="NameSpace" Description="实体类命名空间" OnChanged="" Editor=""
EditorBase="" Serializer="" %>
```

```
<%@ Property Name="DALNameSpace" Type="System.String" Default="DAL" Optional="False"
Category="NameSpace" Description="数据访问类命名空间" OnChanged="" Editor=""
EditorBase="" Serializer="" %>
<%@ Property Name="DBHelpNameSpace" Type="System.String" Default="DAL"
Optional="False" Category="NameSpace" Description="数据执行类命名空间" OnChanged=""
Editor="" EditorBase="" Serializer="" %>
<%@ Property Name="BLLNameSpace" Type="System.String" Default="BLL" Optional="False"
Category="NameSpace" Description="业务类命名空间" OnChanged="" Editor="" EditorBase=""
Serializer="" %>


<%@ Property Name="IsFK" Type="System.Boolean" Default="False" Optional="False"
Category="Other" Description="是否处理外键" OnChanged="" Editor="" EditorBase=""
Serializer="" %>
<%@ Property Name="Author" Type="System.String" Default="Jack.Zhou" Optional="False"
Category="Other" Description="" OnChanged="" Editor="" EditorBase="" Serializer="" %>
<%@ Property Name="OutDir" Type="System.String" Default="C:\aaa" Optional="False"
Category="Other" Description="输出目录" OnChanged="" Editor="" EditorBase=""
Serializer="" %>

<%@ Assembly Name="SchemaExplorer" %>
<%@ Assembly Name="System.Data" %>
<%@ Assembly Name="mscorlib" %>
<%@ Import Namespace="SchemaExplorer" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Collections.Generic" %>
<%@ Import Namespace="System.IO" %>
<%@ Register Name="Model" Template="实体类.cst" MergeProperties="False"
ExcludeProperties="" %>
<%@ Register Name="Service" Template="数据访问类.cst" MergeProperties="False"
ExcludeProperties="" %>
<%@ Register Name="Manager" Template="业务类.cst" MergeProperties="False"
ExcludeProperties="" %>

<%
 if(!Directory.Exists(this.OutDir))
 {
  Directory.CreateDirectory(this.OutDir);
 }
 if(!Directory.Exists(this.OutDir+"\\"+this.ModelNameSpace))
 {
  Directory.CreateDirectory(this.OutDir+"\\"+this.ModelNameSpace);
 }
 if(!Directory.Exists(this.OutDir+"\\"+this.DALNameSpace))
 {
  Directory.CreateDirectory(this.OutDir+"\\"+this.DALNameSpace);
```

```
}
if(!Directory.Exists(this.OutDir+"\\"+this.BLLNameSpace))
{
 Directory.CreateDirectory(this.OutDir+"\\"+this.BLLNameSpace);
}
if(!Directory.Exists(this.OutDir+"\\"+this.DBHelpNameSpace))
{
 Directory.CreateDirectory(this.OutDir+"\\"+this.DBHelpNameSpace);
}

File.Copy("DBHelper.cs",this.OutDir+"\\"+this.DBHelpNameSpace+"\\"+"DBHelper.cs",true);

foreach(TableSchema table in this.SourceDB.Tables)
{
 Model model=new Model();
 model.Author=this.Author;
 model.NameSpace=this.ModelNameSpace;
 model.IsFK=this.IsFK;
 model.SourceTable=table;
 model.RenderToFile(this.OutDir+"\\"+this.ModelNameSpace+"\\"+model.GetFileName(),true);

 Service service=new Service();
 service.Author=this.Author;
 service.NameSpace=this.DALNameSpace;
 service.IsFK=this.IsFK;
 service.SourceTable=table;
 service.DALClassPostFix=this.DALClassPostFix;
 service.ModelNameSpace=this.ModelNameSpace;
 service.DBHelpNameSpace=this.DBHelpNameSpace;
 service.RenderToFile(this.OutDir+"\\"+this.DALNameSpace+"\\"+service.GetFileName(),true);

 Manager manager=new Manager();
 manager.Author=this.Author;
 manager.NameSpace=this.BLLNameSpace;
 manager.IsFK=this.IsFK;
 manager.SourceTable=table;
 manager.DALClassPostFix=this.DALClassPostFix;
 manager.DALNameSpace=this.DALNameSpace;
 manager.ModelNameSpace=this.ModelNameSpace;
 manager.BLLClassPostFix=this.BLLClassPostFix;
 manager.RenderToFile(this.OutDir+"\\"+this.BLLNameSpace+"\\"+manager.GetFileName(),true);
 }
%>
<script runat="template">
</script>
```

功能技巧等

下载：

官方网站：<http://www.codesmithtools.com/>

5.0破解文件下载：

<http://kewlshare.com/dl/0538fcf454d3/CodeSmith.5.0.Professional.Incl.Patch.DTCG.rar.html>

资料：

<http://blog.sina.com.cn/s/articlelist_1229294631_3_1.html>
<http://www.cnblogs.com/Terrylee/category/44974.html>

快捷键：

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

<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
}
```

选中它，再按下Ctrl + Shift + Q后就会变成：

技巧集：

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

```
<%@ Property Name="CollectionType" Type="CollectionTypeEnum" Category="Collection"
Description="Type of collection" %>

<script runat="template">
public enum CollectionTypeEnum
{
      Vector,
      HashTable,
      SortedList
```

```
}
</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

```
<%
foreach (TableSchema table in SourceDatabase.Tables)
{
   OutputSubTemplate(table);
}
%>
<script runat="template">
private CodeTemplate _mySubTemplate;

Browsable(false)]
public CodeTemplate MySubTemplate
{
   get
   {
      if (_mySubTemplate == null)
      {
         CodeTemplateCompiler compiler = new CodeTemplateCompiler
(this.CodeTemplateInfo.DirectoryName + "MySubTemplate.cst");
         compiler.Compile();
         if (compiler.Errors.Count == 0)
         {
            _mySubTemplate = compiler.CreateInstance();
         }
         else
         {
            for (int i = 0; i < compiler.Errors.Count; i++)
```

```
            {
                Response.WriteLine(compiler.Errors[ i].ToString());
            }
        }
    }
    return _mySubTemplate;
  }
}

public void OutputSubTemplate(TableSchema table)
{
  MySubTemplate.SetProperty("SourceTable", table);
  MySubTemplate.SetProperty("IncludeDrop", false);
  MySubTemplate.SetProperty("InsertPrefix", "Insert");
  MySubTemplate.Render(Response);
}
</script>
```

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

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

10. 如何确定一个字段的默认值

在字段的扩展属性集合中包含一个叫“CS_Default”的属性

```
 <%
 foreach(ColumnSchema cs in SourceTable.Columns) {
       if (cs.ExtendedProperties["CS_Default"] != null)
       {
            Response.WriteLine(cs.ExtendedProperties["CS_Default"].Value);
       }
 }
 %>
```

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

```
  Input Parameters:
 <%foreach(ParameterSchema ps in SourceProcedure.AllInputParameters)
 {
       Response.Write(ps.Name);
       Response.Write("\n");
 }
 %>
```

```
Output Parameters:
<%foreach(ParameterSchema ps in SourceProcedure.AllOutputParameters)
{
      Response.Write(ps.Name);
      Response.Write("\n");
}
%>
```

12.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。

13.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。

```
<%@ CodeTemplate Language="C#" TargetLanguage="Text" Description="AddTextWriter
Demonstration." %>
<%@ Import Namespace="System.IO" %>
This template demonstrates using the AddTextWriter method
to output the template results to multiple locations concurrently.
<script runat="template">
public override void Render(TextWriter writer)
{
    StreamWriter fileWriter1 = new StreamWriter(@"C:\test1.txt", true);
```

```
        this.Response.AddTextWriter(fileWriter1);

        StreamWriter fileWriter2 = new StreamWriter(@"C:\test2.txt", true);
        this.Response.AddTextWriter(fileWriter2);

        base.Render(writer);

        fileWriter1.Close();
        fileWriter2.Close();
}
```

</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。

14.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="OutputFileCodeTemplate"
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")]
public override string OutputFile
{
    get {return base.OutputFile;}
    set {base.OutputFile = value;}
}
</script>
```

15.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-template instance
   this.CopyPropertiesTo(header);
```

16.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"
```

```
Description="Constructor parameter name." %>
<%@ Property Name="ConstructorParameterType" Type="System.String" Description="Data type
of the constructor parameter." %>
class <%= ClassName %>
{
    <%= ConstructorParameterType %> m_<%= ConstructorParameterName %>;
    public <%= ClassName %>(<%= ConstructorParameterType %> <%= ConstructorParameterName
%>)
    {
        m_<%= ConstructorParameterName %> = <%= ConstructorParameterName %>
    }
}
```

该模版生成的代码可能如下：

```
 class Account
 {
     int m_balance;
     public Account(int balance)
     {
         m_balance = balance
     }

}
```

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：

```
class Checking : Account
{
    public Checking : base(0)
    {
    }
}
```

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。

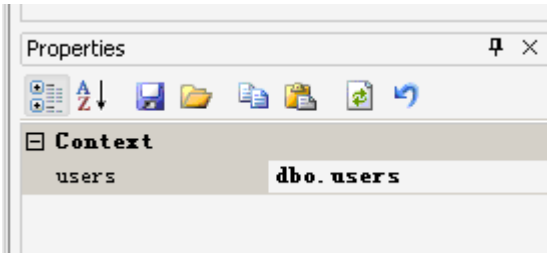
错误总结

清除CodeSmith二次安装前的烦恼

第一次使用CodeSmith的时候，因为不小心，选择了标准版的注册号，使得CodeSmith Studio不能正常使用，后来想卸载再次安装，发现不论怎么做，都自动识别了先前的注册码，后无意发现，原来在系统目录下，有个CodeSmith的LICENSE文件，一直没有删除掉，赶紧delete，啊。。。。终于世界太平了。。。。立马选择专业版注册，可以感受强大的代码生成乐趣了。

C:\Program Files\Common Files\XHEO\SharedLicenses下的.lic文件

如果出现 " 表名 is required "



表明Context的连接数据库属性没选

表中是否有主键

<%=this.SourceTable.PrimaryKeys.Count%>获取不到主键信息，为0.<%if (this.SourceTable.HasPrimaryKey){%>即可

或者this.SourceTable.PrimaryKeysColumnsthis.Count

表名不能是C＃或所使用语言中的**关键字**，例如，news为表名，当做为参数的时候就为new了.这样就会出错.

System.NullReferenceException: 未将对象引用设置到对象的实例。

是否为主键的判断？

```
//  if(!(bool)column.ExtendedProperties["CS_IsIdentity"].Value)，这样好像在Access中不管用
  if(!column.IsPrimaryKeyMember)
```

汉字不支持的解决办法：

打开CodeSmith Studio设置CodeSmith tools->option->enable unicode support

然后在模板中：加上ResponseEncoding="UTF-8" 如下：

```
<%@ CodeTemplate Language="C#" ResponseEncoding="UTF-8" %>
```

实例

```
<%@ CodeTemplate Language="C#" TargetLanguage="T-SQL" Description="Create a procedure which have insert function base on a table." %>
<%@ Assembly Name="SchemaExplorer" %>
<%@ Import Namespace="SchemaExplorer" %>
<%@ Property Name="SourceTable" Type="SchemaExplorer.TableSchema" Category="DataTable" Description="Table that the stored procedures should be based on." %>
<%@ Property Name="Author" Type="String" Category="Context" Description="The author for this procedure."%>
<%@ Property Name="Description" Type="String" Category="Context" Description="The description for this procedure."%>
<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 + ")";
            }
            break;
        }
    }
    return param;
}
</script>
CREATE PROCEDURE dbo.<%=SourceTable.Name %>Insert
/*
==============================================
Author:<%= Author %>
CreatedTime:<%= System.DateTime.Now.ToShortDateString() %>
Description:<%= Description %>
==============================================
*/
<% for (int i = 0; i < SourceTable.Columns.Count; i++) { %>
<%= GetSqlParameterStatement(SourceTable.Columns[i]) %><% if (i < SourceTable.Columns.Count - 1) { %>,<% } %>    <% if (SourceTable.Columns[i].Description != "") { %>--<%= SourceTable.Columns[i].Description %><% } %>
<% } %>
AS
Insert Into [<%= SourceTable.Name %>]
(
<% for (int i = 0; i < SourceTable.Columns.Count; i++) { %>
[<%= SourceTable.Columns[i].Name %>]<% if (i < SourceTable.Columns.Count - 1) { %>,<% } %>    <% if (SourceTable.Columns[i].Description != "") { %>--<%= SourceTable.Columns[i].Description %><% } %>
<% } %>
)
Values
(
<% for (int i = 0; i < SourceTable.Columns.Count; i++) { %>
@<%= SourceTable.Columns[i].Name %><% if (i < SourceTable.Columns.Count - 1) { %>,<% } %>
<% } %>
)
```

二、具有删除功能的模板

今天又根据CodeSmith的几个基本组件写出了基于表生成删除功能的存储过程代码生

成模板。

昨天觉得添加的存储过程模板写的比较简单，今天准备详细介绍一下这个删除的模板。

首先介绍我们使用到的一个教本函数GetSqlParameterStatement(ColumnSchema column)，其函数代码如下：

```
1  public string GetSqlParameterStatement(ColumnSchema column)
2  {
3      string param = "@" + column.Name + " " + column.NativeType;
4      switch (column.DataType)
5      {
6          case DbType.Decimal:
7          {
8              param += "(" + column.Precision + ", " + column.Scale + ")";
9              break;
10         }
11         default:
12         {
13             if (column.Size > 0)
14             {
15                 param += "(" + column.Size + ")";
16             }
17             break;
18         }
19     }
20     return param;
21 }
```

大家可以看到，这个函数需要传入一个ColumnSchema类型的参数，它代表一个数据表中的列，并且是一个列，然后根据ColumnSchema这个类具有的属性，对传入的列进行一些操作然后返回我们生成存储过程时需要的代码。

首先介绍一下ColumnSchema的一些常用属性，如下表：

属性Property	描述Description
AllowDBNull	是否允许空值NULL
Database	通过DatabaseSchema对象得到当前列所属的数据库
DataType	此数据对象的数据类型
Description	当前对象的描述
ExtendedProperties	用来存储SchemaObject的其他附加信息
IsForeignKeyMember	当前列是否为外键
IsPrimaryKeyMember	当前列是否为主键
IsUnique	当前列是否唯一
Name	列的名称
NativeType	列定义的数据类型

Precision	数据对象的精度
Scale	数据对象的范围（个人理解为需要保留小数的范围）
Size	数据对象的大小（例如：字符串长度为10）
SystemType	数据对象的系统类型
Table	当前列所属的数据表

下面为我们首先要生成存储过程，要自动生成的代码分成了红、绿、蓝三部分。

```
CREATE PROCEDURE dbo.CustomersDelete
/*
================================================
Author:Bear-Study-Hard
CreatedTime:2005-12-28
Description:Delete a record from table Customers
================================================
*/
@CustomerID nchar(5) --客户ID
AS
Delete From [Customers]
Where
[CustomerID] = @CustomerID
```

我们的这个脚本函数就是要实现拼出红色的部分，GetSqlParameterStatement函数接收到ColumnSchema类型的参数后，从其Name属性和NativeType属性拼出@CustomerID nchar部分，然后由于不同的数据类型尤其是数值类型和字符串类型的区别，会导致数据类型的大小会有所不同，这里仅对Decimal的数据类型进行了判断（Numeric和float等均需要这种处理），然后根据Precision属性得到精度并通过Scale属性得到了需要保留小数的范围。如果传出的为非Decimal类型字段则直接通过Size属性取出其大小即可。得到了(5)部分。最后的注释是为了生成的存储过程解读性好加上的，使用的是Description属性。

剩下的绿色部分和蓝色部分生成时比较简单，请各位自行学习。模板代码为：

```
<%@ CodeTemplate Language="C#" TargetLanguage="T-SQL" Description="Create a procedure which have delete function base on a table.Must use PrimaryKey to delete a record." %>
<%@ Assembly Name="SchemaExplorer" %>
<%@ Import Namespace="SchemaExplorer" %>
<%@ Property Name="SourceTable" Type="SchemaExplorer.TableSchema" Category="DataTable" Description="Table that the stored procedures should be based on." %>
<%@ Property Name="Author" Type="String" Category="Context" Description="The author for this procedure." Optional="true"%>
<%@ Property Name="Description" Type="String" Category="Context" Description="The description for this procedure." Optional="true"%>
<script runat="template">
public string GetSqlParameterStatement(ColumnSchema column)
{
    string param = "@" + column.Name + " " + column.NativeType;
    switch (column.DataType)
    {
      case DbType.Decimal:
```

```
14      {
15        param += "(" + column.Precision + ", " + column.Scale + ")";
16        break;
17      }
18      default:
19      {
20        if (column.Size > 0)
21        {
22          param += "(" + column.Size + ")";
23        }
24        break;
25      }
26    }
27    return param;
28  }
29  </script>
30  CREATE PROCEDURE dbo.<%=SourceTable.Name %>Delete
31  /*
32  ====================================================
33  Author:<%= Author %>
34  CreatedTime:<%= System.DateTime.Now.ToShortDateString() %>
35  Description:<%= Description %>
36  ====================================================
37  */
38  <% for (int i = 0; i < SourceTable.PrimaryKey.MemberColumns.Count; i++) { %>
39  <%= GetSqlParameterStatement(SourceTable.PrimaryKey.MemberColumns[i]) %><%
if (i < SourceTable.PrimaryKey.MemberColumns.Count - 1) { %>,<% } %>  <% if
(SourceTable.Columns[i].Description != "") { %>--<%= SourceTable.Columns
[i].Description %><% } %>
40  <% } %>
41  AS
42  Delete From [<%= SourceTable.Name %>]
43  Where
44  <% for (int i = 0; i < SourceTable.PrimaryKey.MemberColumns.Count; i++) { %>
45  <% if (i > 0) { %>AND <% } %>[<%= SourceTable.PrimaryKey.MemberColumns
[i].Name %>] = @<%= SourceTable.PrimaryKey.MemberColumns[i].Name %>
46  <% } %>
```

如果有问题我会尽力帮助大家解决的，共同提高^_^

三、微软的一个例子

今天在微软的网站看到的一篇使用CodeSmith的例子，现在写出来大家一起研究研究。

首先，我还是要简要介绍一下其中用到的基础知识。

1. 在模板中的代码区中（<%= %>或<% %>）可以使用.NET中的一些类和方

法。但是就像和.NET项目中一样需要添加应用，就像C#中的using

```
<%@ Assembly Name="System.Data" %>
<%@ Import Namespace="System. Data" %>
```

2．在脚本区域中可以编写生成模板时使用到的函数，其中的语言根据在声明模板时定义的使用的语言相同。

```
<script runat="template">

</script>
```

3．个人感觉在编写模板时就像以前编写ASP页面一样，很灵活。其中的循环和判断的使用结构上和ASP基本一样，只不过在代码区域中使用的语言是声明模板时定义的语言，像C#、VB.NET等。

这次我们要实现的一个功能是将某一文件夹下所有匹配后缀的文件全部找出并输出它们的绝对路径。其实这个模板本身的功能并不强大，主要是大家可以利用这个模板在生成代码时批量向代码文件中生成代码，例如向某一文件夹下的所有.cs文件中插入相应的代码。这个思想是基于一定条件的：①首先要学习我前边在CodeSmith基础中提到的，可以向代码文件中定义的region中添加相应的代码，②文件夹下的.cs文件需要有一定的命名规范，③.cs代码文件中的region区域的定义的名称也需要有一定的命名规范，②③这两个要求是结合起来的，因为我们得到同一文件夹下的所有.cs文件后，根据文件名去确定这个代码文件是哪一层的那个类文件，通过这个名称我们就知道需要读出那个数据表的结构，然后按照这个数据表去像相应命名好的region中添加相应的代码。主要思想就是这样的。下面将模板文件的代码贴出来大家研究。

```
1  <%@ CodeTemplate Language="C#"  TargetLanguage="Text" Description="Simple template to show main syntax" %>
2  <%@ Property Name="Filter" Default="*.cst" Type="System.string" Category="Masks" Description="Mask for files in the directory" %>
3  <%@ Assembly Name="SchemaExplorer" %>
4  <%@ Assembly Name="System.Design" %>
5  <%@ Import Namespace="SchemaExplorer" %>
6  <%@ Import Namespace="System.IO" %>
7  Simple Template Example used to show syntax and structure of template.
8
9  <%= DateTime.Now.ToLongDateString() %>
10
11  <%
12  // Comments within code delimiters or script blocks
13  // http://www.livebaby.cn ( C#)
14  Response.WriteLine("List of files in template directory (using mask "+ Filter + ")");
15  DisplayDirectoryContents(Filter);
16  Response.WriteLine(">> Code Generation Complete.");
17  %>
18
```

```
<script runat="template">
// Iterates through the current directory and displays
// a list of the files that conform to the supplied
// mask.
public void DisplayDirectoryContents(string sFilter)
{
    string[] dirFiles = Directory.GetFiles
      (this.CodeTemplateInfo.DirectoryName, sFilter);
    for (int i = 0; i < dirFiles.Length; i++)
    {
      Response.WriteLine(dirFiles[i]);
    }
}
</script>
```

四、实现选择路径对话框

首先我们要添加<%@ Assembly Name="System.Design" %>命名空间。然后我们在模板中自定义一个属性，用来表示要存储的路径。其中我们使用了this.CodeTemplateInfo.DirectoryName得到当前模版所在路径作为默认路径。

```
private string _outputDirectory = String.Empty;

[Editor(typeof(System.Windows.Forms.Design.FolderNameEditor), typeof
(System.Drawing.Design.UITypeEditor))]
[Optional]
[Category("Output")]
[Description("The directory to output the results to.")]
public string OutputDirectory
{
	get
	{
		// default to the directory that the template is located in
		if (_outputDirectory.Length == 0) return this.CodeTemplateInfo.DirectoryName +
"output\\";

		return _outputDirectory;
	}
	set
	{
		if (!value.EndsWith("\\")) value += "\\";
		_outputDirectory = value;
	}
}
```

这样编译运行后我们就可以看到如下效果：

单击选择路径按钮后我们就可以看到这样的窗口

选择后相应的路径值就会填入属性框。