动软实战攻略

动软模板使用教程

文档编号：20110525

在线帮助:http://help.maticsoft.com

目录

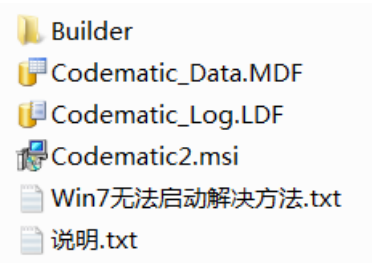
一. 功能介绍

动软代码生成器 是一款为程序员设计的全功能自动代码生成器，也是一个智能化软件开发平台，它可以生成基于面向对象的思想和三层架构设计的代码，结合了软件开发中经典的思想和设计模式，融入了工厂模式，反射机制等等一些思想。主要实现在对应数据库中表的基类代码的自动生成，包括生成属性、添加、修改、删除、查询、存在性、Model 类构造等基础代码片断,支持不同架构代码生成，使程序员可以节省大量机械录入的时间和重复劳动，而将精力集中于核心业务逻辑的开发。新版本中除了程序集组件模板，也同样支持用户自定义文本模板，像写 ASPX 代码一样写模板，一键代码生成，更方便，更灵活。

动软让软件开发变得轻松而快乐！让企业不断提升开发效率，同样的时间创造出更大的价值。

二. 下载安装

1. **系统要求**：
   Microsoft Windows2000/XP/2003/7 或者更高。机器必须安装.NET Framework v2.0。
2. 官方下载地址：http://www.maticsoft.com/download.aspx
3. 下载解压后安装包有如下文件：

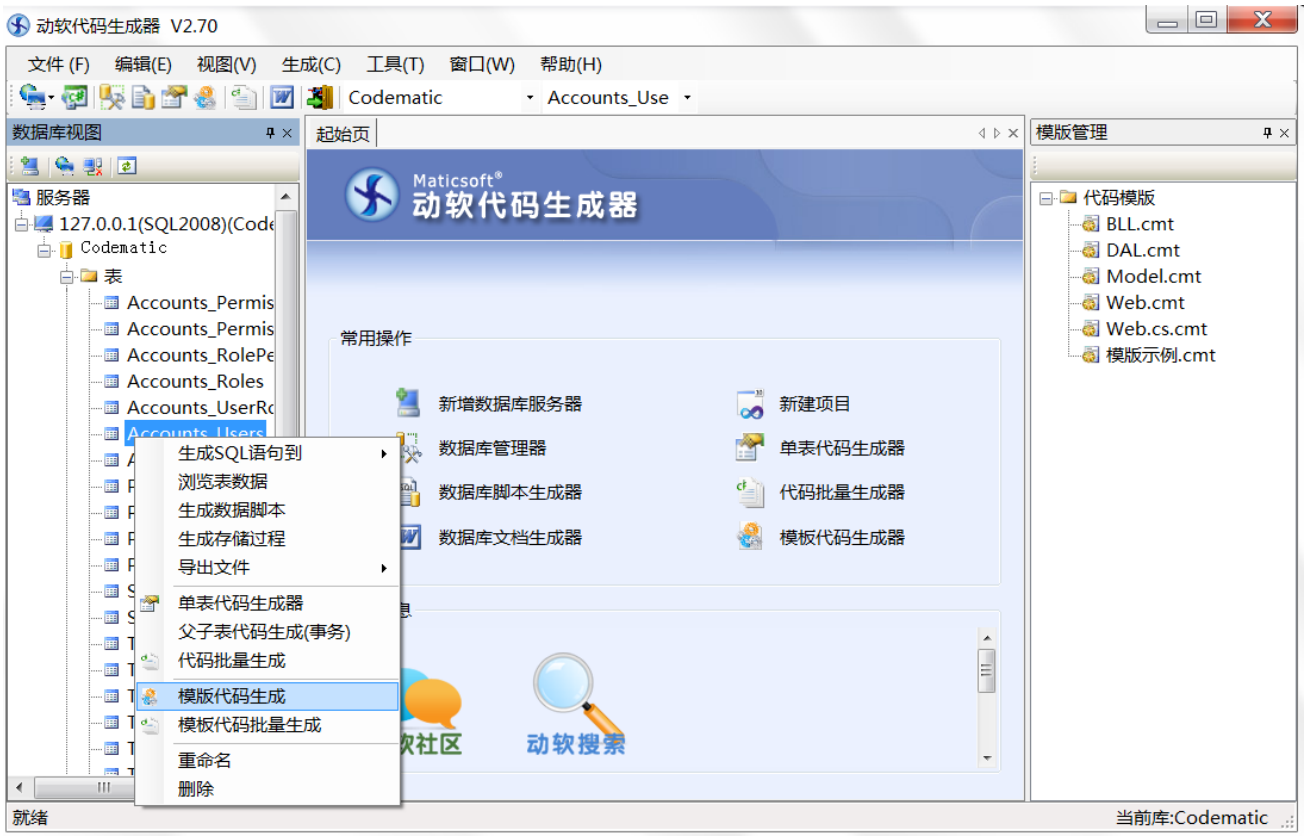


Codematic2.msi 是动软.NET 代码生成器的安装文件。
Builder 文件夹是代码生成插件的源码，动软.NET 代码生成器支持可扩展的代码生成插件，用户可以定制自己的代码生成的插件，根据接口开发自己的代码生成方式，按自己的需求进行代码生成。
Codematic_Data.MDF 和 Codematic_Log.LDF 是通过动软新建项目中所带管理模块所需要的数据库文件。后台管理员默认登录用户名:admin 密码：1

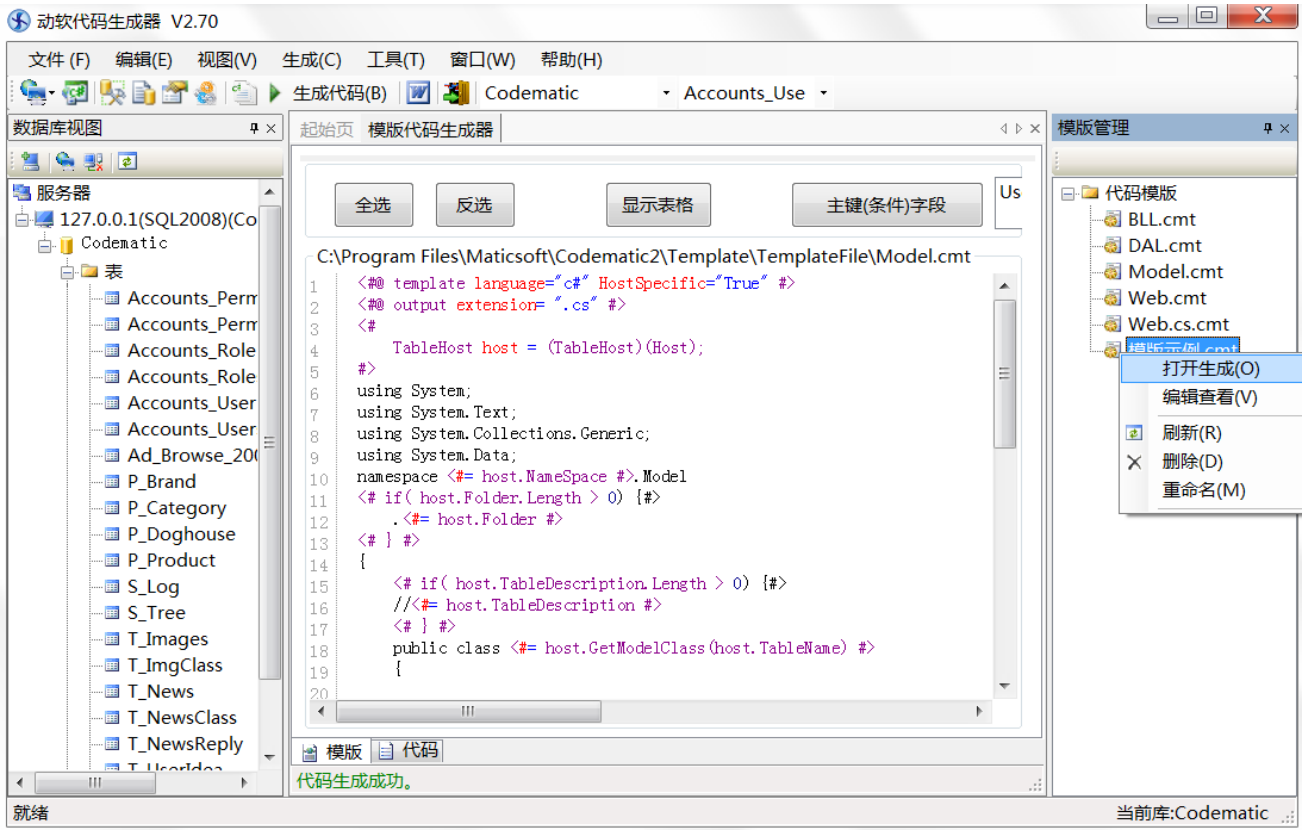
4. 双击 Codematic2.msi 进行直接安装即可。
   安装动软时，如果用户机器 360 弹出警告，那仅仅是个签名认证提示，并非木马，选择"继续安装"，然后点击"确定"即可。
   动软郑重声明： 动软.NET 代码生成器，绝无插件木马，纯绿色软件。请放心安装。
5. 安装成功后，在开始-菜单和桌面上会有动软.NET 代码生成器的图标。

三. 模板代码生成

1. 在左侧【数据库视图】，选中表，右键菜单【模板代码生成】



2. 然后，出现单表的代码生成器界面，我们设置自己需要更改的信息

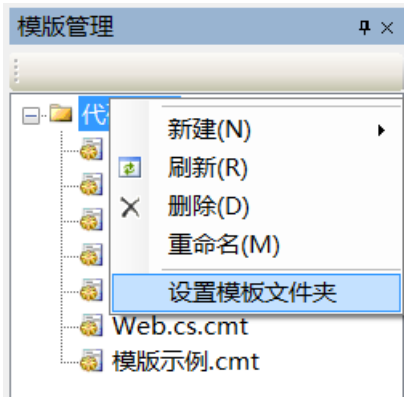


备注：代码还有一些生成规则，是在菜单【工具】-【选项】-【代码生成设置】中进行设置。

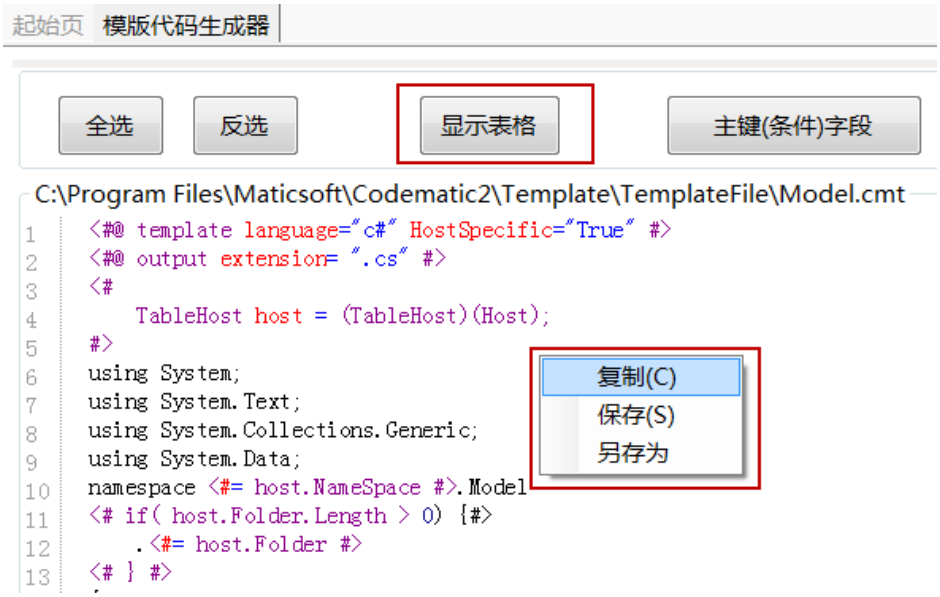
3. 然后，点击工具栏【生成代码】或者【生成代码】按钮，即可生成该类的代码：

生成的代码，可以直接复制到项目文件中，也可以右键保存成 CS 文件。通过窗体下面的 Tab 按钮可以来回切换设计视图和代码。

4. 设置模板目录



5. 模板保存



6.

四. 模板代码批量生成

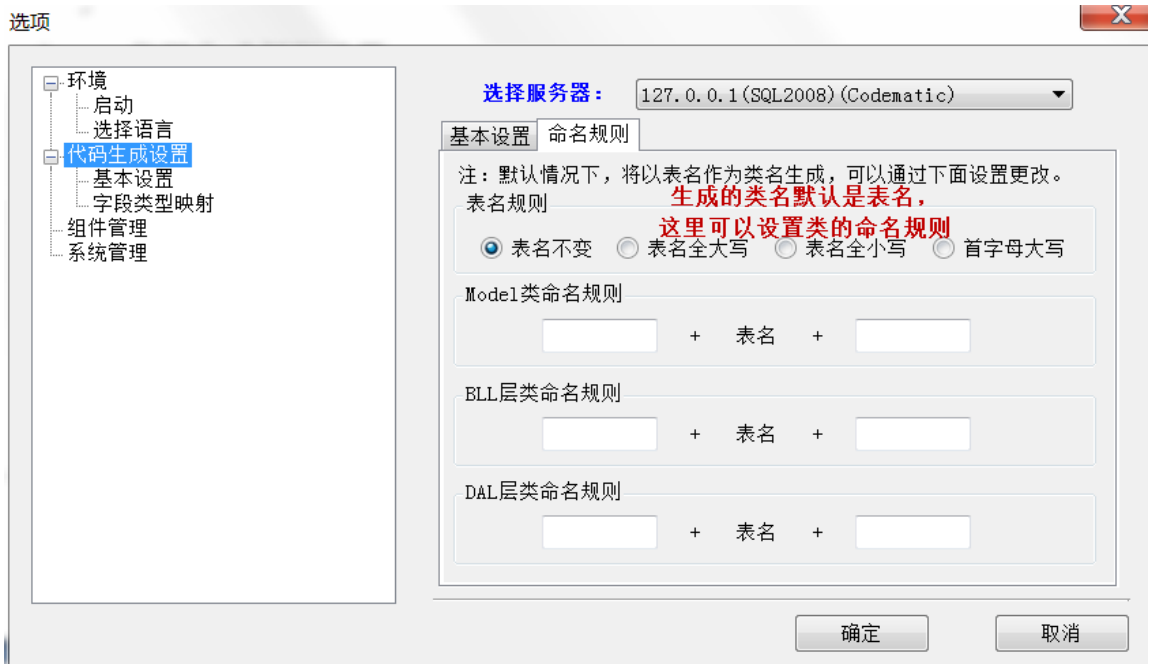
1. 选中数据库或者表，然后单击右键菜单【模板代码批量生成】
2. 出现的窗口和新建项目基本相似，只是多了一个选中架构的选项：

3. 选则要生成的表，然后点击【导出】。
4. 在生成的文件夹中，我们可以看到：

批量生成代码只生成业务表的代码，不再有解决方案文件和项目文件，以及其它类库等。我们可以将生成的这些文件直接拖到现有的解决方案中即可

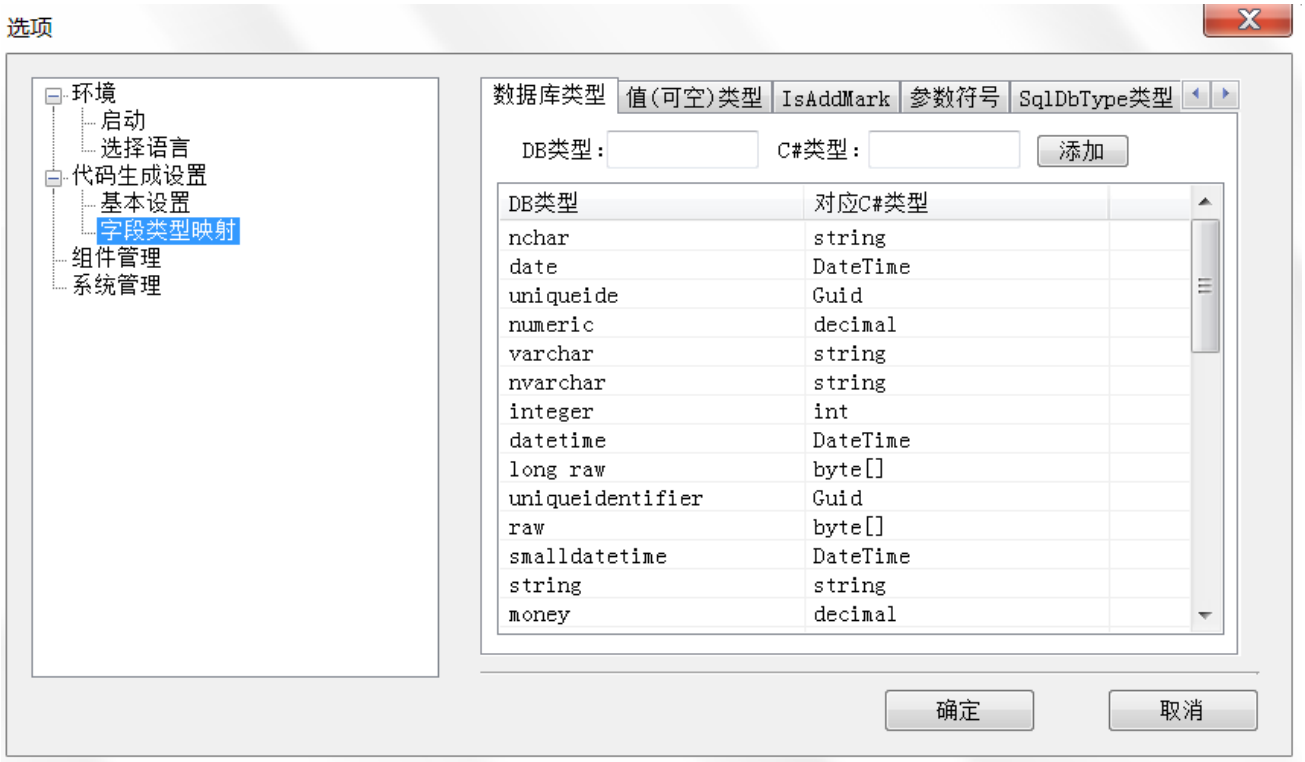
五. 代码生成规则设置

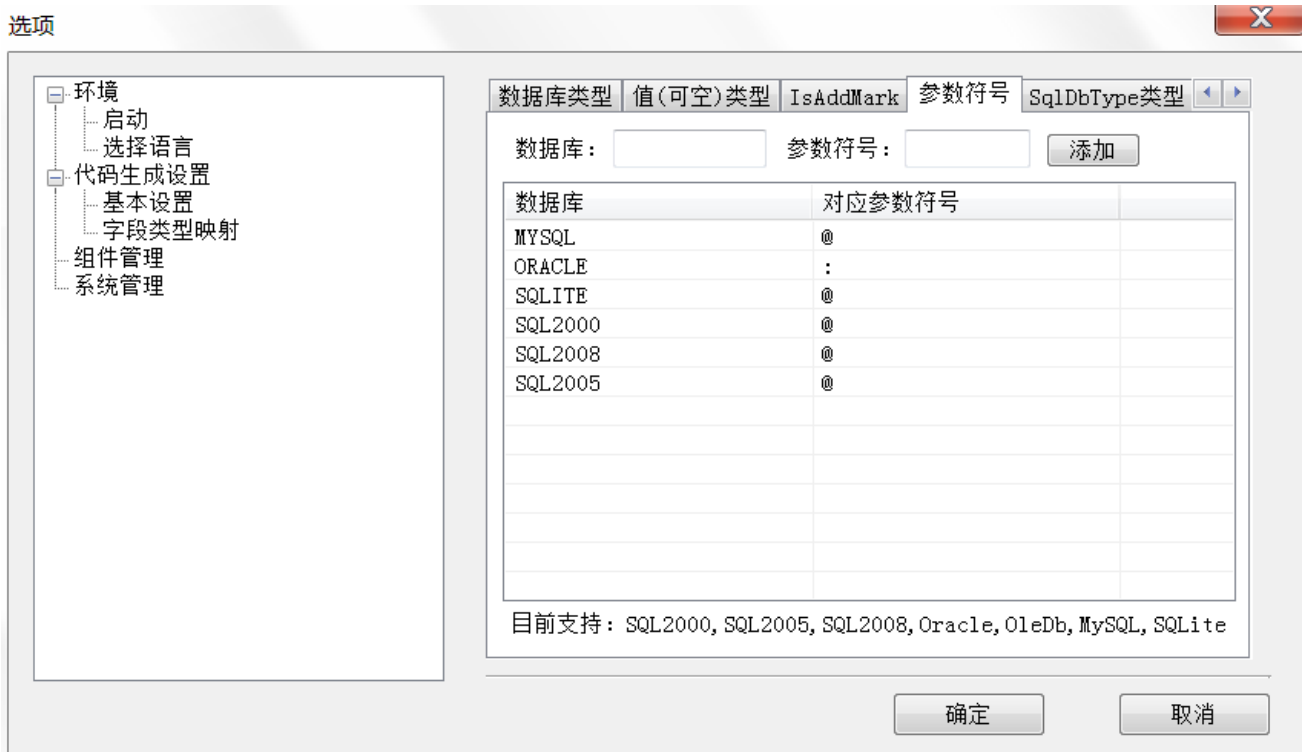
打开菜单【工具】-【选项】-【代码生成设置】



这些配置保存后，在生成代码的时候将按照这个规则进行生成。
代码生成的规则设置范围还在不断增加中。

另外，不同数据库类型的数据类型各有不同，这里提供了字段类型和 C#中的类型建立映射关系，生成代码时将按映射关系来生成代码字段属性的类型。


选项
环境
启动
选择语言
代码生成设置
基本设置
字段类型映射
组件管理
系统管理
数据库类型
值(可空)类型
IsAddMark
参数符号
SqlDbType类型
DB类型：
C#类型：
添加
DB类型
对应C#类型
nchar string
date DateTime
uniqueide Guid
numeric decimal
varchar string
nvarchar string
integer int
datetime DateTime
long raw byte[]
uniqueidentifier Guid
raw byte[]
smalldatetime DateTime
string string
money decimal
确定
取消


选项
环境
启动
选择语言
代码生成设置
基本设置
字段类型映射
组件管理
系统管理
数据库类型
值(可空)类型
IsAddMark
参数符号
SqlDbType类型
数据库：
参数符号：
添加
数据库
对应参数符号
MYSQL @
ORACLE :
SQLITE @
SQL2000 @
SQL2008 @
SQL2005 @
目前支持：SQL2000,SQL2005,SQL2008,Oracle,OleDb,MySQL,SQLite
确定
取消

六. 模板编写教程

模板的大体分为 5 块内容：模板指令声明，代码语句块，表达式块，类功能控制块，文本块输出。下面我们分别来介绍一下这 5 块内容的语法和使用说明。

1. 模板指令声明块 <#@ #>

和 ASP.NET 页面的指令一样，它们出现在文件头，通过<#@…#>表示。其中<#@ template …#>指令是必须的，用于定义模板的基本属性。

(1) 模板指令

<#@ template [language="VB"] [hostspecific="true"] [debug="true"] [inherits="templateBaseClass"] [culture="code"] [compilerOptions="options"] #>

例如

```
<#@ template language="C#v3.5" hostSpecific="true" debug="true" #>
```

Language 这里可以指定模板使用的语言。

hostSpecific="true" 表示是否使用特定的 host，host 里面包含了模板使用的各种对象。

注意：

所有属性值必须用双引号都括起来。如果值本身包含引号，则必须使用 \ 字符对这些引号进行转义。指令通常是模板文件或包含的文件中的第一个元素。

(2) 参数指令

```
<#@ parameter type="Full.TypeName" name="ParameterName" #>
```

(3) 输出指令

```
<#@ output extension=".fileNameExtension"  encoding="encoding"  #>
```

output extension=".cs" 指定生成文件的扩展名。

encoding="encoding" 指定生成文件的编码。

(4) 程序集指令

```
<#@ assembly name="System.Data" #>
```

用于添加程序集引用，如果要使用第三方程序集，那么最好在项目中添加引用。

注：您应使用绝对路径的名称，或者路径名称中使用标准的宏的名称。

例如： <#@ assembly name="$(SolutionDir)library\MyAssembly.dll" #>

(5) 导入指令

```
<#@ import namespace="System.Data" #>
```

导入要使用的命名空间，注意：这里的命名空间必须要在前面指定的程序集里面找得到的，比如我指定命名空间"System.Data"，"System.Data.Common"，这些在程序集 System.Data 中都有的

(6) 包含指令

<#@ include file="test.tt" #> 导入模板，类似 Html 的 include 用法

include 指令插入其他模板文件的文本。

例如，下面的指令插入 test.txt 的内容。 <#@ include file="c:\test.txt" #>

在处理时，被包含内容就像是包含文本模板的组成部分一样。 不过，即使 include 指令后跟普通文本块和标准控制块，也可以包含编写有类功能块 <#+...#> 的文件

2. 代码语句块：<# #>

在模板文件中，可以混合使用任意数量的文本块和标准控制块。中间是一段通过相应编程语言编写的程序调用，我们可以通过代码语句快控制文本转化的流程。

注意：不能在控制块中嵌套控制块。

```
<#@ output extension=".txt" #>
<#
    for(int i = 0; i < 4; i++)
    {
#>
     Hello!
<#
    }
#>
```

3. 表达式块：<#= #>

表达式控制块计算表达式并将其转换为字符串。 该字符串将插入到输出文件中。

例如： <#= 2 + 3 #>

表达式可以包含作用域中的任何变量。 例如，下面的块输出数字行：

```
<#@ output extension=".txt" #>
<#
    for(int i = 0; i < 4; i++)
    {
#>
This is hello number <#= i+1 #>: Hello!
<#
    }
#>
```

4. 类功能控制块：<#+ #>

如果文本转化需要一些比较复杂的逻辑，我们需要写在一个单独的辅助方法中，甚至是定义一些单独的类，我们就是将它们定义在类特性块中。类功能控制块定义属性、方法或不应包含在主转换中的所有其他代码。 类功能块常用于编写帮助器函数。 通常，类功能块位于单独的文件中，这样它们可以包含在多个文本模板中。 类功能控制块以 <#+ ... #> 符号分隔。

例如，下面的模板文件声明并使用一个方法：

```
<#@ output extension=".txt" #>
Squares:
<#
    for(int i = 0; i < 4; i++)
    {
#>
    The square of <#= i #> is <#= Square(i+1) #>.
<#
```

```
        }
    #>
    That is the end of the list.
    <#+    // Start of class feature block
    private int Square(int i)
    {
        return i*i;
    }
    #>
```

类功能必须编写在文件末尾。不过，即使 include 指令后跟标准块和文本，也可以 <#@include #> 包含类功能的文件。例如下面代码则会报错：

```
List of Squares:
<#
    for(int i = 0; i < 4; i++)
    {    WriteSquareLine(i); }
#>
End of list.
<#+    // Class feature block
private void WriteSquareLine(int i)
{
#>
    The square of <#= i #> is <#= i*i #>.
<#
}
#>
```

5. 文本块输出

可以使用 Write() 和 WriteLine() 方法在标准代码块内追加文本，而不必使用表达式代码块。 它们可帮助缩进输出和报告错误。

下面两个代码块在功能上是等效的。

包含表达式块的代码块

```
 <#
      int i = 10;
      while (i-- > 0)
      { #>
            <#= i #>
      <# }
#>
```

使用 WriteLine() 的代码块

```
<#
    int i = 10;
    while (i-- > 0)
```

```
        {
            WriteLine((i.ToString()));
        }
        #>
```

Write() 和 WriteLine() 方法有两个重载，其中一个重载接受单个字符串参数，另一个重载接受一个复合格式字符串以及将包含在字符串中的对象数组（与 Console.WriteLine() 方法类似）。
下面两种 WriteLine() 用法在功能上是等效的：

```
<#
    string msg = "Say: {0}, {1}, {2}";
    string s1 = "hello";
    string s2 = "goodbye";
    string s3 = "farewell";

    WriteLine(msg, s1, s2, s3);
    WriteLine("Say: hello, goodbye, farewell");
#>
```

设置文本模板输出缩进的格式。
CurrentIndent 字符串属性显示文本模板中的当前缩进，该类还具有一个 indentLengths 字段，该字段是已添加的缩进的列表。
PushIndent() 方法增加缩进，
PopIndent() 方法减少缩进。
ClearIndent() 方法，删除所有缩进。

下面的代码块演示这些方法的用法：

```
<#
    WriteLine(CurrentIndent + "Hello");
    PushIndent("    ");
    WriteLine(CurrentIndent + "Hello");
    PushIndent("    ");
    WriteLine(CurrentIndent + "Hello");
    ClearIndent();
    WriteLine(CurrentIndent + "Hello");
    PushIndent("    ");
    WriteLine(CurrentIndent + "Hello");
#>
```

此代码块产生以下输出：

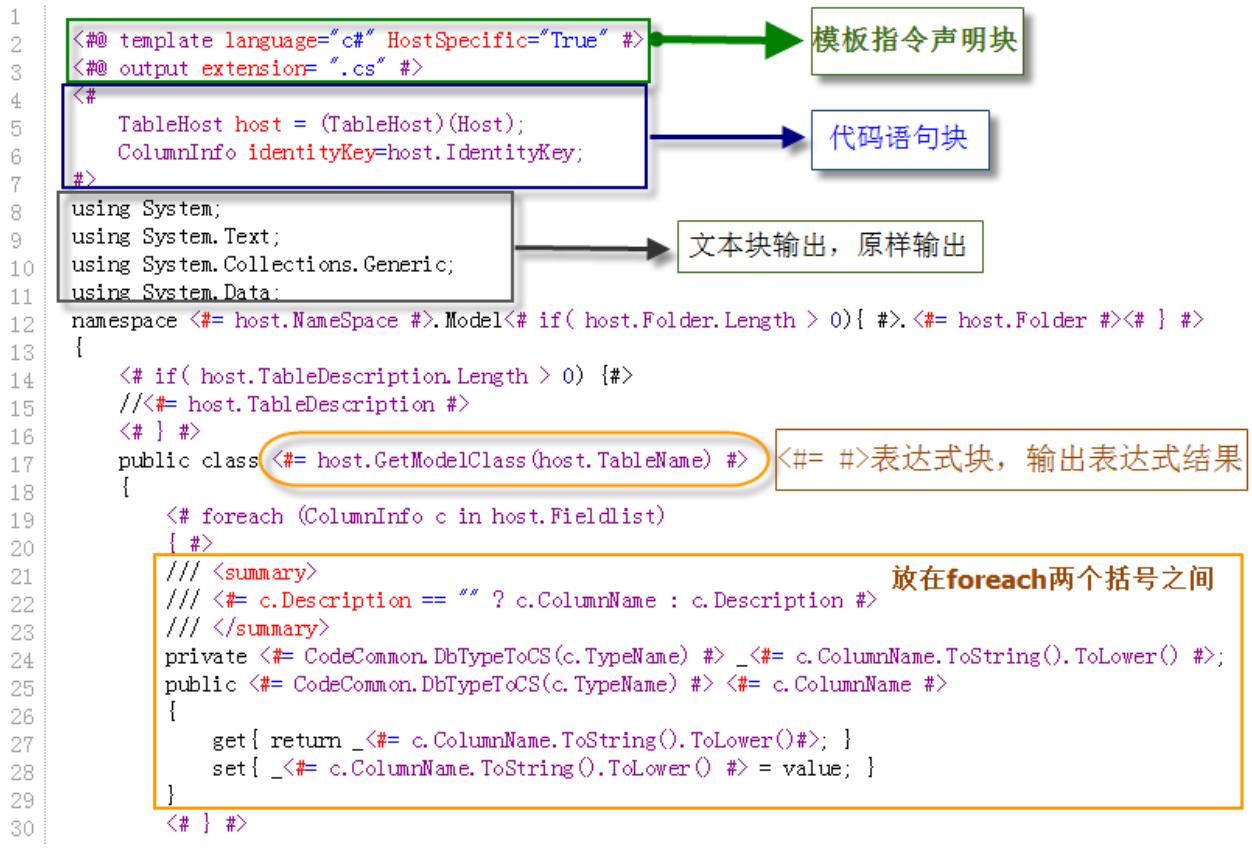
```
Hello
        Hello
                Hello
Hello
        Hello
```

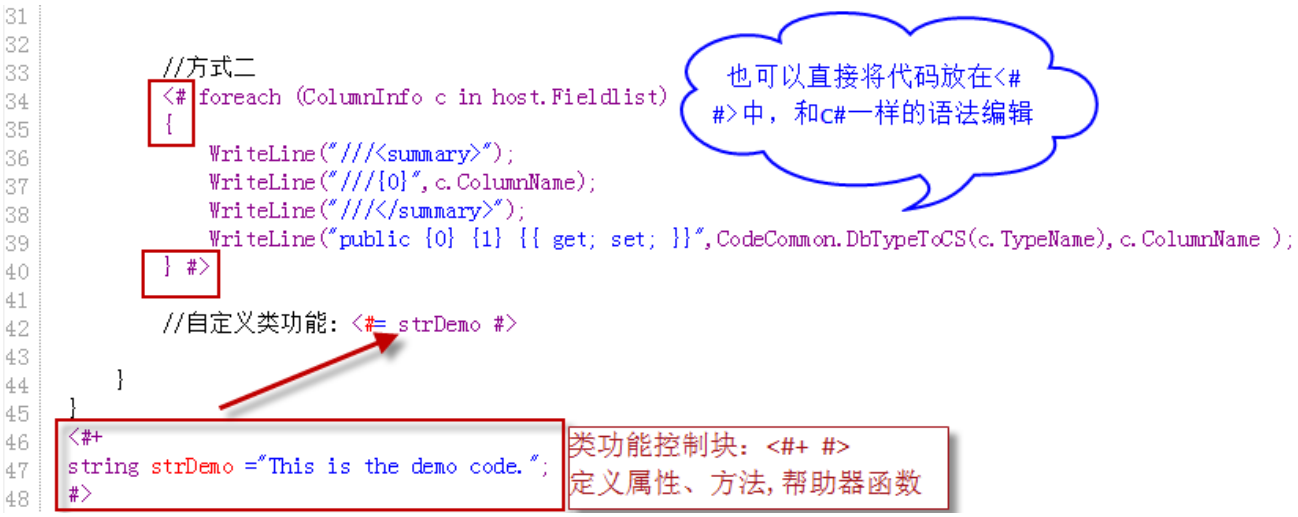
6. 显示错误和警告

可以使用错误和警告实用工具方法向 Visual Studio 错误列表添加消息。 例如，下面的代码向错误列表添加一条错误消息。

```
<#
  try
  {
     string str = null;
     Write(str.Length.ToString());
  }
  catch (Exception e)
  {
     Error(e.Message);
  }
#>
```

7. 模板示例讲解





8. Host 对象属性列表

Host 提供了一下属性和方法，方便编写模板时调用。

TableHost 属性	类型	说明
DbName	string	当前数据库名。
DbType	string	当前数据库类型： SQL2000,SQL2005,SQL2008,Oracle,OleDb,MySQL,SQLite
TableList	List<TableInfo>	当前数据库所有的表集合，返回List<TableInfo>
ViewList	List<TableInfo>	当前数据库所有的视图集合，返回List<TableInfo>
ProcedureList	List<TableInfo>	当前数据库所有的存储过程集合，返回List<TableInfo>
DbHelperName	string	数据库访问类名，例如：DbHelperSQL
ProjectName	string	项目名称
ProcPrefix	string	存储过程前缀，例如 sp_
ModelPrefix	string	Model 类名前缀
ModelSuffix	string	Model 类名后缀
BLLPrefix	string	BLL 类名前缀
BLLSuffix	string	BLL 类名后缀
DALPrefix	string	DAL 类名前缀
DALSuffix	string	DAL 类名后缀
TabNameRule	string	类命名规则：same(保持原样)　lower（全部小写）　upper（全部大写） 工具-选项-代码生成设置中进行设置。
DbParaDbType	string	得到数据库字段 DbType 的类名。例如：SqlDbType
preParameter	string	当前数据库类型对应的存储过程参数符号，例如：@
Folder	string	所属文件夹
TableName	string	表名
TableDescription	string	表的描述信息
Fieldlist	List<ColumnInfo>	字段集合
Keys	List<ColumnInfo>	主键字段集合
IdentityKey	ColumnInfo	自动增长标识列；如果没有则返回 null。

9. Host 对象方法列表

TableHost 方法	说明
string GetModelClass(string TabName)	组合得到 Model 类名：前缀+类名+后缀
string GetBLLClass(string TabName)	组合得到 BLL 类名：前缀+类名+后缀
string GetDALClass(string TabName)	组合得到 DAL 类名：前缀+类名+后缀

10. TableInfo 表对象属性

属性	类型	说明
TabName	string	表名称
TabUser	string	用户对象
TabType	string	表类型
TabDate	string	创建日期

11. ProcedureHost 表对象属性

属性	类型	说明
ProcedureName	string	表名称
Parameterlist	List<ColumnInfo>	参数列表
OutParameter	ColumnInfo	输出参数字段信息

12. ColumnInfo 字段信息对象

属性	类型	说明
ColumnOrder	string	序号
ColumnName	string	字段名
TypeName	string	字段类型
Length	string	长度
Precision	string	精度
Scale	string	小数位数
IsIdentity	bool	是否是标识列
IsPrimaryKey	bool	是否是主键
Nullable	bool	是否允许空
DefaultVal	string	默认值
Description	string	备注

13. CodeCommon 工具类常用方法

CodeCommon 方法	说明

string DbTypeToCS(string dbtype)	得到“数据库字段类型”对应的“c#类型”
bool isValueType(string cstype)	是否 C#中的值（可空）类型
string DbTypeLength(string dbtype,string datatype, string Length)	得到数据库字段 DbType 的类型和长度
ColumnInfo GetIdentityKey(List<ColumnInfo> keys)	得到自动增长标识列字段
string DbParaDbType(string DbType)	得到不同数据库 DbType 类名，例如 SqlDbType
string preParameter(string DbType)	得到不同数据库的存储过程参数符号，例如：@
string GetWhereParameterExpression(List<ColumnInfo> keys, bool IdentityisPrior, string DbType)	得到 Where 条件语句 - Parameter 方式（例如：用于 Exists Delete GetModel 的 where） 例如：where NewsId=@NewsId
string GetPreParameter(List<ColumnInfo> keys, bool IdentityisPrior, string DbType)	生成 sql 语句中的参数列表(例如：用于 Exists Delete GetModel 的 where 参数赋值)
string GetInParameter(List<ColumnInfo> keys, bool IdentityisPrior)	得到方法输入参数定义的列表（例如：用于 Exists Delete GetModel 的参数传入）
string GetFieldstrlist(List<ColumnInfo> keys, bool IdentityisPrior)	字段的 select 列表，和方法传递的参数值
string GetWhereExpression(List<ColumnInfo> keys, bool IdentityisPrior)	得到 Where 条件语句 - SQL 方式（例如：用于 Exists Delete GetModel 的 where）
string GetModelWhereExpression(List<ColumnInfo> keys, bool IdentityisPrior)	得到 Where 条件语句 - SQL 方式（例如：用于 Exists Delete GetModel 的 where）
string CutDescText(string descText,int cutLen,string ReplaceText)	字符串描述截取：要截取的字符串，长度，替代字符串。

输出效果模板代码：（复制到模板代码生成器中执行即可看到效果）

```
<#@ template language="c#" HostSpecific="True" #>
<#@ output extension= ".cs" #>
<#
 TableHost host = (TableHost)(Host);
 ColumnInfo identityKey=host.IdentityKey;
#>
数据库名:<#= host.DbName #>
数据库类型:<#= host.DbType #>
表名:<#= host.TableName #>
表描述:<#= host.TableDescription #>
数据库访问类名:<#= host.DbHelperName #>
项目名称:<#= host.ProjectName #>
存储过程前缀:<#= host.ProcPrefix #>
类命名规则:<#= host.TabNameRule #>
数据库 DbType 类名:<#= host.DbParaDbType #>
存储过程参数符号:<#= host.preParameter #>
表集合:
<# foreach (TableInfo tab in host.TableList)
```

```
    {
    WriteLine(tab.TabName);
    } #>
    字段集合:
    <# foreach (ColumnInfo c in host.Fieldlist)
    {
    WriteLine("public {0} {1} {{ get;
set; }}",CodeCommon.DbTypeToCS(c.TypeName),c.ColumnName );
    } #>
    字段集合增加连接符号：
    <# for(int i=0;i< host.Keys.Count;i++)
    {   ColumnInfo key = host.Keys[i]; #>
        <# if (key.IsPrimaryKey || !key.IsIdentity)
        {#>
             strSql.Append(" <#= key.ColumnName#> = <#=preParameter#><#=key.ColumnName#> <#
if (i< host.Keys.Count-1 ) {#>and <#}#> ");
        <#}#>
    <# }#>
    主键字段集合集合:
    <# foreach (ColumnInfo c in host.Keys)
    {
    WriteLine("public {0} {1} {{ get;
set; }}",CodeCommon.DbTypeToCS(c.TypeName),c.ColumnName );
    } #>
    Model 类名:<#= host.GetModelClass(host.TableName) #>
    BLL 类名:<#= host.GetBLLClass(host.TableName) #>
    DAL 类名:<#= host.GetDALClass(host.TableName) #>
    <#= CodeCommon.DbParaDbType(host.DbType) #>
    <#= CodeCommon.preParameter(host.DbType) #>
    <#= CodeCommon.GetWhereParameterExpression(host.Keys,true,host.DbType) #>
    <#= CodeCommon.GetPreParameter(host.Keys,true,host.DbType) #>
    <#= CodeCommon.GetInParameter(host.Keys,true) #>
    <#= CodeCommon.GetFieldstrlist(host.Keys,true) #>
<#= CodeCommon.GetWhereExpression(host.Keys,true) #>
```