# Inside ASP.NET 2.0-即時編譯系統

文/黃忠成(原文刊登於 Run! PC)

## 從 ASP.NET 1.1 到 2.0，編譯系統的進化

在筆者撰寫『深入剖析 ASP.NET 元件設計』一書時，曾相當深入的探討 ASP.NET 1.1 的即時編譯模型，該章節以圖 1 爲開端，一步步的將隱身於後的設計理念攤開在讀者面前，時至今日，ASP.NET 即將邁入 2.0 了，這個即時編譯模型做了相當大幅度的變化，圖 2 是對照 1.1 與 2.0 的即時編譯模型概觀，讀者們可以發現，2.0 的即時編譯模型複雜了許多。

圖 1

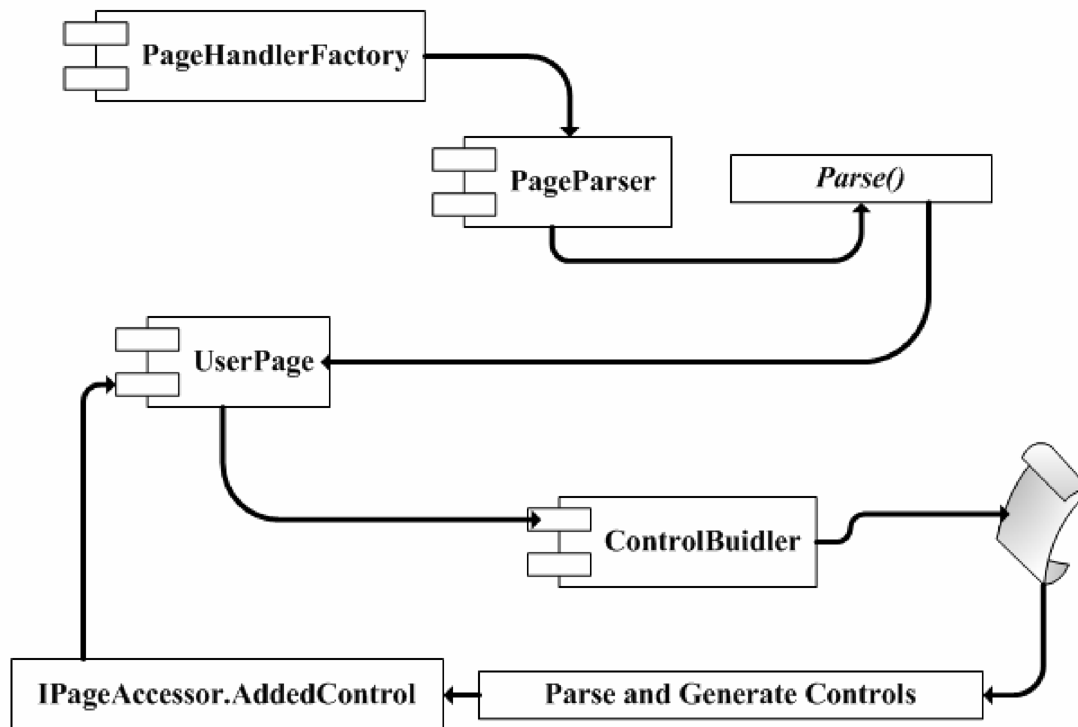
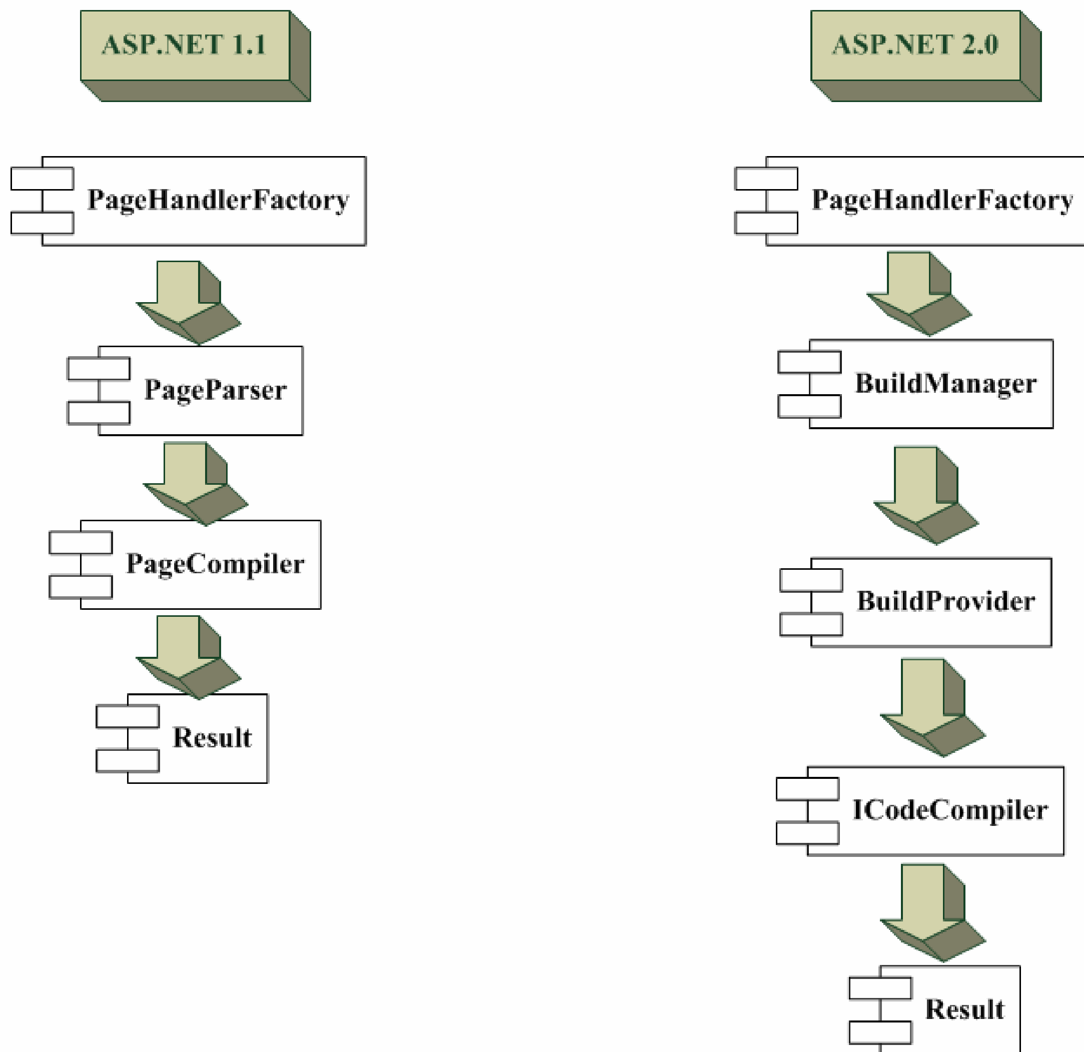
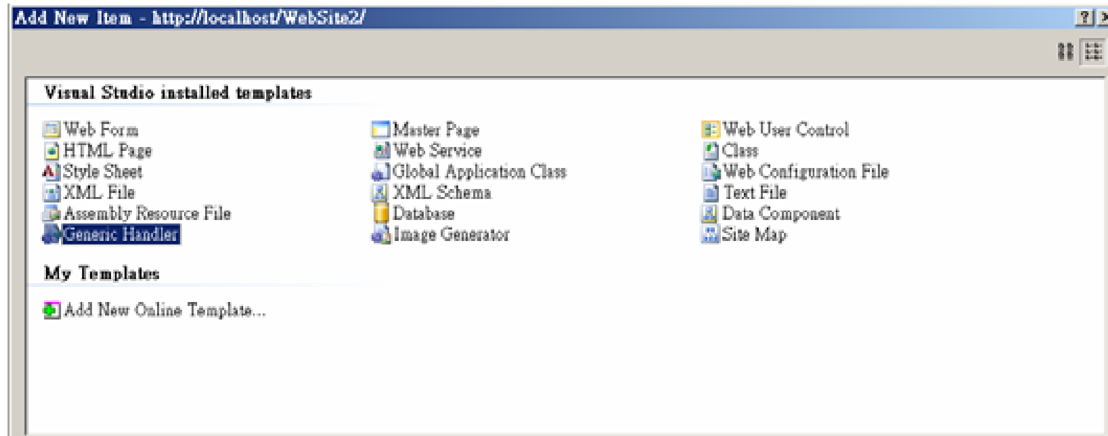



圖 2




在 1.1 時，當訪問者要求一個文件時，ISAPIRuntime(IIS 的要求處理物件)會依照文件類型來喚起適當的 Http Handler，以.aspx 來說就是 PageHandlerFactory，她也是即時編譯系統的入口，這段流程在 2.0 仍然沒有改變，但後面的動作就完全變樣了，在 1.1 時，PageHandlerFactory 會使用 PageParser 來解譯.aspx 文件，再交由 PageCompiler 來產生出編譯檔案。在 2.0 時，同樣的動作是交由 BuildManager 來完成，其會呼叫適當的 BuildProvider 來處理要求的文件，最後交由適當的 Compiler 來產生編譯檔案。不知讀者們是否看出上面這段話所隱含的意義，是的!BuildManager 具備依照不同附檔名使用不同 BuildProvider 的能力，這代表著設計者可能擁有撰寫自訂的 BuildProvider 來參與即時編譯流程。讀者們可以在 Visual Web Developer 的 New Items 選項中看到圖 3 的畫面。

圖 3




其中最引人注意的是 ASP.NET 2.0 允許使用者撰寫 Generic Handler，也就是 1.1 中的自定 Http Handler 程式檔，該 Wizard 會產生出程式 1 的碼。

程式 1

```
<%@ WebHandler Language="C#" Class="Handler" %>
using System.Web;
public class Handler : IHttpHandler {
    public void ProcessRequest (HttpContext context) {
        context.Response.ContentType = "text/plain";
        context.Response.Write("Hello World");
    }
    public bool IsReusable {
        get {
            return false;
        }
    }
}
```

你是否看到了一個介於 ASP.NET Script 與一般程式檔的怪異程式碼呢？在存檔後執行時會看到圖 4 的結果。

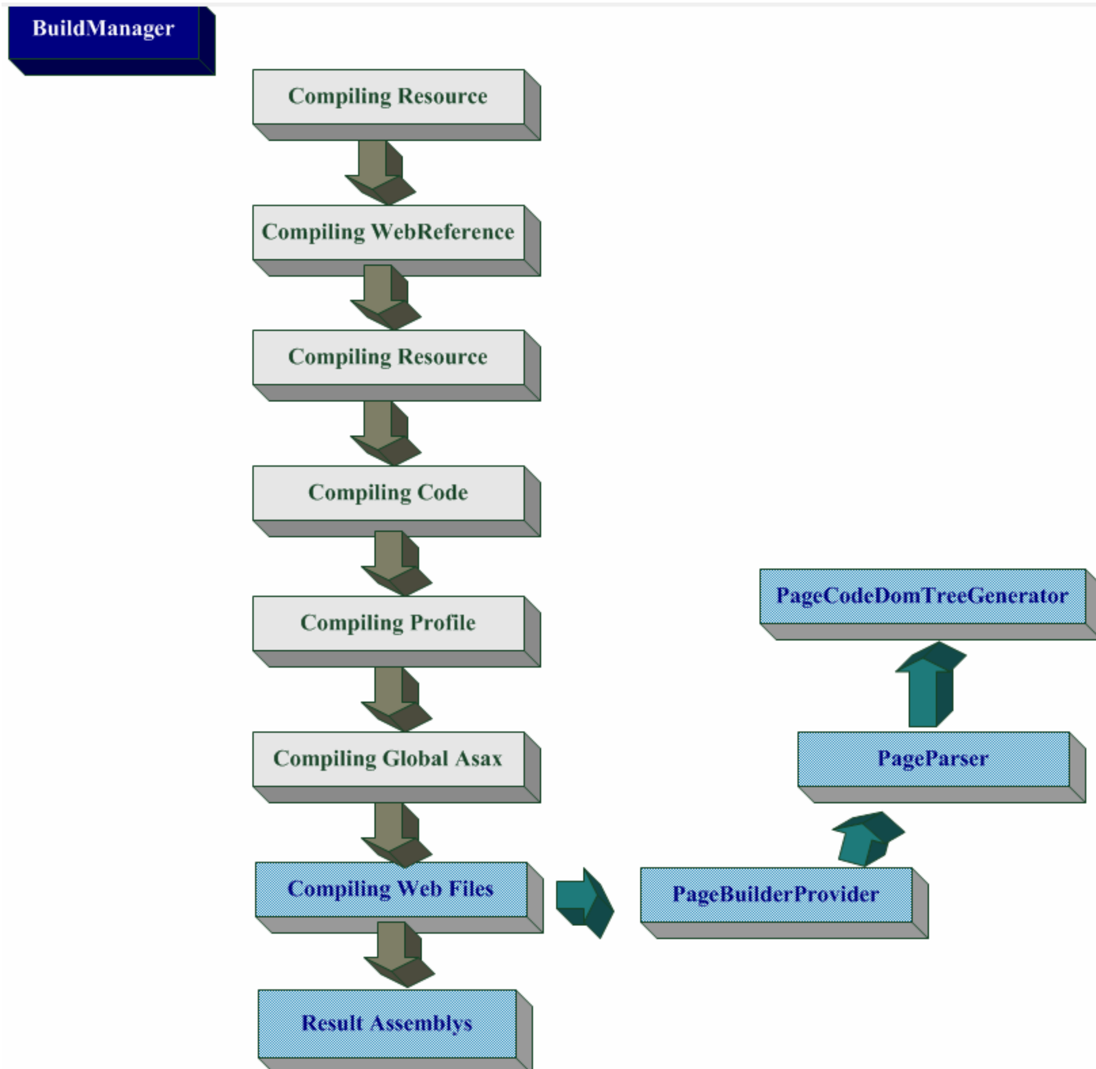
圖 4

問題來了，以往撰寫這種自定義的 Http Handler 時，設計師必須預先將程式碼編譯好，放置於網站目錄下，這個 Handler 才能正常運作，但現在並未執行這個編譯動作啊？那是誰為我們編譯這個檔案，又是如何做的呢？答案與 1.1 時相同，就是 SimpleHandlerFactory，但後面的動作就不同了，以往的 SimpleHandlerFactory 只是載入對應的 Assembly 來啓動 Http Handler，在 2.0 時此動作換成了 BuildManager，她會尋找.ashx 對應的 BuildProvider，也就是 WebHandlerBuildProvider 來運行即時編譯模型。以上的討論說明一件事，Handler Factory 的大部份工作已經下放給 BuildManager 了，而目的就是提供一個更強大的即時編譯模型，不只是可以編譯.aspx、.ascx，還可以編譯各式各樣的文件，例如.masterpage 也是這個模型中的一員，這帶來了一個難以想像的極大優點，設計師以後將具備自定 Script 文件的能力，只要有需求，設計師可以自行定義一種 Script 語言，再提供對應的 BuildProvider 物件，BuildManager 將很樂意的為你完成即時編譯動作，而且優點還不只於此，BuildManager 支援預編譯模型，也就是說設計師只要提供 Script 文件與 BuildProvider 後，就能享受即時編譯與預編譯兩種模型。舉一個較實務的例子，一個設計師希望提供某種較簡單的 Script 語言供使用者應用，那麼該設計師只需提供一個 MyScriptBuildProvider，將其與特定的附檔名對應之後，再利用 CodeDom 來產生真正的程式碼就可以了，接下來的動作 BuildManager 將很樂意的幫你完成。

## Reloaded! Page Compiler-Time

既然 2.0 已經改變即時編譯模型了，那麼就讓我們從新了解這個編譯系統究竟是如何動作的，在 1.1 中，即時編譯系統最令人注意的是 PageParser 物件，此物件會讀入.aspx 文件，將其解譯成一群 Control Builder 物件交由 PageCompiler 物件來產生原始程式碼後編譯，這段過程在 2.0 中依然沒變，不同的是 PageParser 在 2.0 中已經不是由 PageHandlerFactory 來呼叫了，圖 5 是 2.0 中 Web Page 的編譯時期概觀。

圖 5

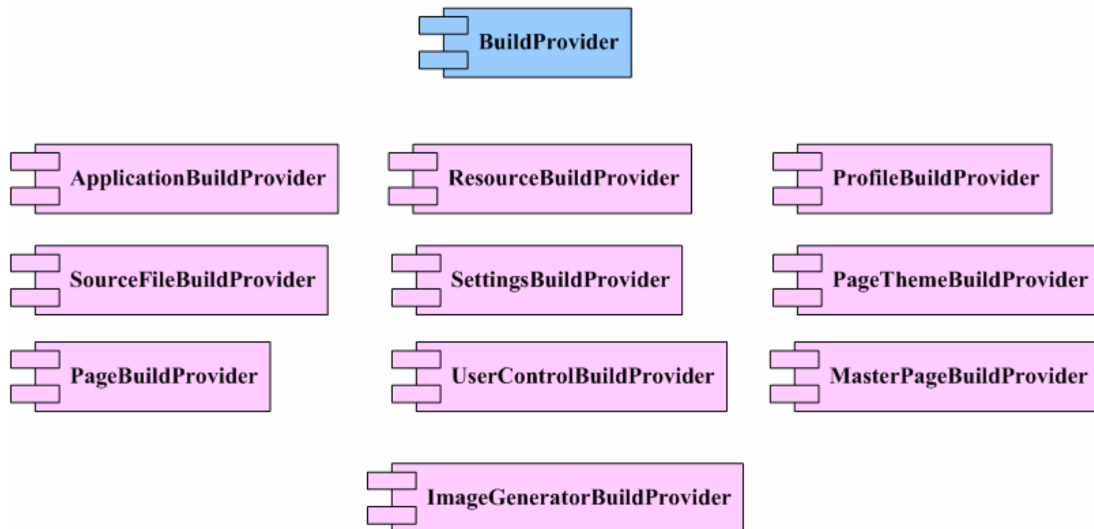



呼，一段不算短的旅行景點，在 BuildManager 接到編譯命令時，會先將目錄中的幾個外部檔案編譯好，這些檔案就是 Resource、Web Reference、Code、Profile、Global.asax，Resource 指的是資源檔，Web Reference 通常是引用 Web Services 時用的檔案，Code 是位於 Code 目錄下的程式檔，Profile 則是位於 web.config 中的 Profile 區段定義，Global Asax 則是大家所熟悉的 Global.asax 檔案。接下來是本節的重頭戲了，Compiling Web Files！這個動作將會編譯網站中的.aspx 或是其它擁有相對應 BuildProvider 物件的檔案，例如.ashx 與.masterpage 等等。回到 Page 的編譯週期上，.aspx 所對應的 BuildProvider 是 PageBuilderProvider 物件，此物件會使用 PageParser 來解譯.aspx，再利用 PageCodeDomTreeGenerator 來產生出原始碼，最後交由適當的 Compiler 編譯。

## **Manager**、**Provider**、Generator

承上節，BuildManager 與 BuildProvider 及其 CodeGenerator 擁有不可分的關係，圖 6 是目前 2.0 所提供的一部份 BuildProvider 物件，讀者們可以在其中發現許多熟悉的物件名稱，她們就是對應到目前你能在 ASP.NET 中撰寫的文件。

圖 6




有趣的是 Page Theme 居然也擁有一個 PageThemeBuilderProvider，這代表著什麼呢？筆者原來以爲 Theme 只是一個簡單的文字檔，當 Page 套用某一個 Theme 時，只是由該文字檔中讀取定義來套用至控件上，但結果不然，由 PageThemeBuilderProvider 的出現來看，Theme 是一個編譯後的文件，PageThemeBuilderProvider 會編譯所有的 Theme 檔案，也就是.skin，事實上，所有於內的控件定義都會被編譯成控件實體，當 Page 需要套用某個 Theme 至控件時，只是將控件的屬性複製過來罷了，沒有解譯動作，速度自然快上不少。基本上，所有可編譯型的 BuildProvider 物件都會提供兩個物件，一個是 Parser，用來解譯文件用，另一個是 CodeDomTreeGenerator，用來將 Control Builder 物件群轉換爲可編譯的原始碼，以 PageTheme 來說，就如圖 7 所示。

圖 7

當然，這並不是說每一個 BuildProvider 都得提供這些東西，契約層僅到達 BuildProvider 就停止了，只要該 BuildProvider 能傳回一個實體，BuildManager 並不管其內部是如何達到的。

## 預編譯系統

截至目前爲止，我們一直在即時編譯系統上打轉，並未談到另一個系統，那就是預編

譯系統，事實上這個系統只是即時編譯系統的一種呈現型式，當 BuildManager 啓動時，會先判別要求的目錄中是否擁有.compiled 的檔案，存在的話就將其視爲預編譯模式，載入.compiled 文件中所定義的 Assembly，等會！預編譯後的檔案連真正的.aspx 都不存在了，BuildManager 如何做接下來的動作了，又是如何與 BuildProviders 互動呢？哦，沒有！在預編譯情況下，BuildManager 根本就不會用到 BuildProvider，這跟即時編譯系統的二次動作一樣，當即時編譯系統完成後，會將結果存放到暫存目錄中，順帶著也會放一份到 Cache 中，待下次收到要求時，就直接取用了，預編譯系統只是跳過了第一次那一段動作而已，這代表著，自定的 BuildProvider 不用做特別的動作，就可以享受到預編譯系統的優點。

## Custom Build Provider

由於目前 ASP.NET 2.0 仍處於 Beta 版本，有關於 Build Provider 的資訊少之又少，不過我還是在文件中找到程式 2 的說明。

**程式 2**

```
<configuration>
   <system.web>
     <compilation>
         …….
       <buildProviders>
         <buildProvider extension=".mafx" type="BuildProviderType,
           BuildProviderAssembly" />
         </buildProviders>
     </compilation>
   </system.web>
</configuration>
```

粗體字的部份就是定義自定 Build Provider 的地方，這可以證明在 ASP.NET 2.0 中，設計師是被允許撰寫 Build Provider 的，不過除了這個文件外，我再也找不到更深入的資訊了，而很不幸的，這個文件有一個錯誤，其中的 buildProvider 定義是不被接受的，實際上的語法應該如程式 3。

**程式 3**

```
<compilation debug="true">
     <buildProviders>
           <add extension=".ppp"
                type="TestBuildProvider.MyCSharpBuilder, TestBuildProvider"
                    appliesTo="Code"/>
     </buildProviders>
 </compilation>
```

extension 是定義此 BuildProvider 對應至何種副檔名，type 指的是 BuildProvider 的 Assembly 與 Type，最後的 appliesTo 是代表著使用於何種模式，有四個選擇，一是 Code，代表程式檔，二是 Resource，就是資源檔，三是 Web，代表著網頁檔案(.aspx、.ascx…)，四是 All，代表任何類型檔案。要撰寫 BuildProvider，讀者必須先準備 Visual Studio 2005 Beta 或是 Visual C# Express，**這些工具才有提供 Class Library 的 Wizard，否則就得使用 Command Line 方式編譯範例了**，程式 4 是我們的第一個 BuildProvider 範例。

程式 **4**

```
#region Using directives
using System;
using System.Collections.Generic;
using System.Text;
using System.IO;
using System.CodeDom;
using System.CodeDom.Compiler;
using System.Web.Compilation;
#endregion

namespace TestBuildProvider
{
    public class MyCSharpBuilder:BuildProvider
    {
        public override void GenerateCode(AssemblyBuilder assemblyBuilder)
        {
            TextReader reader = base.OpenReader();
            string scriptString = reader.ReadLine();
            CodeCompileUnit unit = new CodeCompileUnit();
            unit.Namespaces.Add(new CodeNamespace("TEST"));
            CodeTypeDeclaration class1 = new CodeTypeDeclaration("HelloClass");
            class1.IsClass = true;
            CodeMemberMethod method1 = new CodeMemberMethod();
            method1.Name = "SayHello";
            method1.ReturnType = new CodeTypeReference("System.String");
            method1.Statements.Add(new CodeMethodReturnStatement(
                                    new CodePrimitiveExpression(scriptString)));
            method1.Attributes = MemberAttributes.Public;
            class1.Members.Add(method1);
            unit.Namespaces[0].Types.Add(class1);
            assemblyBuilder.AddCodeCompileUnit(this, unit);
```

```
        }
    }
}
```

讓我稍微解釋一下這個範例，程式中以 CodeDom 產生出一個 HelloClass 類別，在其中加入一個方法：SayHello，爲其定義一個字串型別的傳回值，特別注意的是此值是由 OpenReader 所傳回的 TextReader 中讀回來的，OpenReader 會以 TextReader 來開啓目前處理的檔案，此例中就是 class1.ppp(後詳)。編譯後將其複製到網站目錄中的 bin 目錄下，假設你的網站目錄下並沒有 bin 目錄，那就自行建一個吧。接著修改 web.config 加入程式 5 的定義。

**程式 5**

```
<compilation debug="true">
   <buildProviders>
   <add extension=".ppp"  type="TestBuildProvider.MyCSharpBuilder, TestBuildProvider"
          appliesTo="Code"/>
    </buildProviders>
</compilation>
```

完成之後建立一個檔案 class1.ppp，內容如程式 6。

**程式 6**

```
hello i am buildprovider,this message is define in class1.ppp.
```

將這個檔案放在網站目錄下的 Code 目錄中，沒有 Code 目錄的話就建一個吧。現在讓我們來試試這個 BuildProvdier 能否正常運作吧，在 Default.aspx 中放入一個 Button，撰寫其事件函式，如程式 7。

**程式 7**

```
Type FindType()
{
    Assembly[] assems = AppDomain.CurrentDomain.GetAssemblies();
    foreach(Assembly assem in assems)
    {
        Type t = assem.GetType("TEST.HelloClass");
        if (t != null) return t;
    }
    return null;
}

void Button2_Click(object sender, EventArgs e)
{
    Type t = FindType();
    if (t != null)
```

```
    {
        object obj = Activator.CreateInstance(t);
        string s = (string)obj.GetType().InvokeMember("SayHello",
        BindingFlags.InvokeMethod | BindingFlags.Instance | BindingFlags.Public, null, obj, new object[]{});
        Button2.Text = s;
    }
}
```

FindType 函式是爲了找尋 HelloClass 這個 Type 而寫的，在網站執行時雖然會載入 BuildProvider 所產生的 Assemblys，但是在這裡並無法知道 class1.ppp 所產生出來的 Assembly 實際名稱，自然也就無法取到 HelloClass 了，所以用 FindType 來搜尋所有的 Assemblys 來取得 HelloClass。連 Assembly 都無法確定了，當然也無法用一般的方式來建立物件及呼叫方法了，所以就只剩下 Reflection 可以用了，此處利用 Reflection 來建立 HelloClass 物件實體，接著呼叫其 SayHello 來取回 class1.ppp 中的文字。

## 另一個編譯子系統：Expression Builder

Build Provider 是一個蠻不錯的設計，設計師可以撰寫自訂的 Provider 來延伸 ASP.NET 的編譯系統，但有時候設計師只是需要一個簡單的動態決議系統，而不是一個以檔案爲基礎的編譯動作，例如程式 8 中所示。

### 程式 8

```
ConnectionString="<%$ ConnectionStrings:AppConnectionString1 %>"
```

這是 ASP.NET 內建的一項簡易設計，<%$ 後的字串在編譯時期時會被解譯成程式 9 的碼。

### 程式 9

```
source1.ConnectionString =
        ConnectionStringsExpressionBuilder.GetConnectionString("AppConnectionString1");
```

藉由此設計，設計師可以將組態檔中的值指給某個屬性，達到以外部檔案改變應用程式行爲的目的，也可以減少程式重編譯的次數，那這是如何達到的呢？如果我們要自定這種功能，又該如何做呢？答案已經在程式 9 中出現了，那就是 ExpressionBuilder 物件。ASP.NET 2.0 允許設計師如撰寫 Build Provider 一般撰寫自訂的 ExpressionBuilder 物件，程式 10 是一個簡單到不行的範例。

### 程式 10

```
#region Using directives
using System;
using System.Collections.Generic;
using System.Text;
using System.Web.Compilation;
```

```
using System.Reflection;
using System.ComponentModel;
using System.CodeDom;
#endregion

namespace TestExpressionBuilder
{
    public class MyExpressionBuilder:ExpressionBuilder
    {
        public override System.CodeDom.CodeExpression
           GetCodeExpression(System.Web.UI.BoundPropertyEntry entry,
                              object parsedData,
                              ExpressionBuilderContext context)
        {
            return new CodePrimitiveExpression(entry.Expression);
        }

        public MyExpressionBuilder()
        {
        }
    }
}
```

程式中覆載了 GetCodeExpression 函式，此函式會在解譯時期被呼叫，此時必須傳回一個 CodeExpression 物件，解譯器藉此產生出類似程式 9 的程式碼。要套用這個 ExpressionBuilder，web.config 中必須包含程式 11 的設定。

程式 11

```
<compilation debug="true">
 <expressionBuilders>
          <add expressionPrefix="MyExpression"
            type="TestExpressionBuilder.MyExpressionBuilder, TestExpressionBuilder"/>
 </expressionBuilders>
</compilation>
```

expressionPrefix 屬性代表著 ExpressionBuilder 所能解析的 Expression 的開頭驗證字碼，只有符合這個字串的 Expression 才會交給 MyExpressionBuilder 來處理，程式 12 是測試碼。

程式 12

```
<asp:Button ID="Button2" Runat="server"
    Text="<%$ MyExpression:i am expxression builder %>" />
```

不過可惜的是，目前的 VWD 似乎完全不認識自定義的 ExpressionBuilder，因此無法在設

計時期顯示出正確的結果。

## 後記

.提醒讀者，NET Framework 2.0 仍處於 Beta 階段，這也代表著目前所談的技術都是未定數，雖然 Build Provider 與 Expression Builder 技術帶給設計師無限的想像空間，但除了 Microsoft 之外沒人能確定，最終版本會不會仍然開放這些功能給設計師使用。