**ASP.NET Core 2.0 自定义 \_ViewStart 和 \_ViewImports 的目录位置**

2017年09月07日 13:01:54

阅读数：290

在 ASP.NET Core 里扩展 Razor 查找视图目录不是什么新鲜和困难的事情，但 \_ViewStart 和 \_ViewImports 这2个视图比较特殊，如果想让 Razor 在我们指定的目录中查找它们，则需要耗费一点额外的精力。本文将提供一种方法做到这一点。注意，文本仅适用于 ASP.NET Core 2.0+, 因为 Razor 在 2.0 版本里的内部实现有较大重构，因此这里提供的方法并不适用于 ASP.NET Core 1.x

为了全面描述 ASP.NET Core 2.0 中扩展 Razor 查找视图目录的能力，我们还是由浅入深，从最简单的扩展方式着手吧。

**准备工作**

首先，我们可以创建一个新的 ASP.NET Core 项目用于演示。

1. mkdir CustomizedViewLocation
2. cd CustomizedViewLocation
3. dotnet new web *# 创建一个空的 ASP.NET Core 应用*

接下来稍微调整下 Startup.cs 文件的内容，引入 MVC：

1. *// Startup.cs*
3. using Microsoft.AspNetCore.Builder;
4. using Microsoft.AspNetCore.Hosting;
5. using Microsoft.Extensions.DependencyInjection;
7. namespace CustomizedViewLocation
8. {
9. public class Startup
10. {
11. public void ConfigureServices(IServiceCollection services)
12. {
13. services.AddMvc();
14. }
16. public void Configure(IApplicationBuilder app, IHostingEnvironment env)
17. {
18. app.UseMvcWithDefaultRoute();
19. }
20. }
21. }

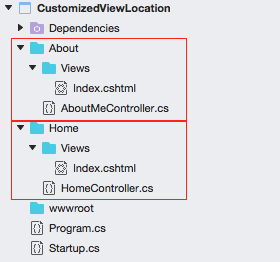
好了我们的演示项目已经搭好了架子。

**我们的目标**

在我们的示例项目中，我们希望我们的目录组织方式是按照功能模块组织的，即同一个功能模块的所有 Controller 和 View 都放在同一个目录下。对于多个功能模块共享、通用的内容，比如 \_Layout, \_Footer, \_ViewStart 和 \_ViewImports 则单独放在根目录下的一个叫 Shared 的子目录中。

**最简单的方式： ViewLocationFormats**

假设我们现在有2个功能模块 Home 和 About，分别需要 HomeController 和它的 Index view，以及 AboutMeController和它的 Index view. 因为一个 Controller 可能会包含多个 view，因此我选择为每一个功能模块目录下再增加一个 Views 目录，集中这个功能模块下的所有 View. 整个目录结构看起来是这样的：



从目录结构中我们可以发现我们的视图目录为 /{controller}/Views/{viewName}.cshtml, 比如 HomeController 的 Index视图所在的位置就是 /Home/Views/Index.cshtml，这跟 MVC 默认的视图位置 /Views/{Controller}/{viewName}.cshtml很相似（/Views/Home/Index.cshtml），共同的特点是路径中的 Controller 部分和 View 部分是动态的，其它的都是固定不变的。其实 MVC 默认的寻找视图位置的方式一点都不高端，类似于这样：

1. string controllerName = "Home"; *// “我”知道当前 Controller 是 Home*
2. string viewName = "Index"; *// "我“知道当前需要解析的 View 的名字*
4. *// 把 viewName 和 controllerName 带入一个代表视图路径的格式化字符串得到最终的视图路径。*
5. string viewPath = string.Format("/Views/{1}/{0}.cshtml", viewName, controllerName);
7. *// 根据 viewPath 找到视图文件做后续处理*

如果我们可以构建另一个格式字符串，其中 {0} 代表 View 名称， {1} 代表 Controller 名称，然后替换掉默认的 /Views/{1}/{0}.cshtml，那我们就可以让 Razor 到我们设定的路径去检索视图。而要做到这点非常容易，利用 ViewLocationFormats，代码如下：

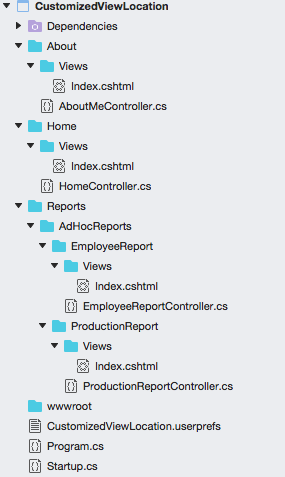
1. *// Startup.cs*
3. public void ConfigureServices(IServiceCollection services)
4. {
5. IMvcBuilder mvcBuilder = services.AddMvc();
6. mvcBuilder.AddRazorOptions(options => options.ViewLocationFormats.Add("/{1}/Views/{0}.cshtml"));
7. }

收工，就这么简单。顺便说一句，还有一个参数 {2}，代表 Area 名称。

这种做法是不是已经很完美了呢？No, No, No. 谁能看出来这种做法有什么缺点？

这种做法有2个缺点。

1. 所有的功能模块目录必须在根目录下创建，无法建立层级目录关系。且看下面的目录结构截图：



注意 Reports 目录，因为我们有种类繁多的报表，因此我们希望可以把各种报表分门别类放入各自的目录。但是这么做之后，我们之前设置的 ViewLocationFormats 就无效了。例如我们访问 URL /EmployeeReport/Index, Razor 会试图寻找 /EmployeeReport/Views/Index.cshtml，但其真正的位置是 /Reports/AdHocReports/EmployeeReport/Views/Index.cshtml。前面还有好几层目录呢～

1. 因为所有的 View 文件不再位于同一个父级目录之下，因此 \_ViewStart.cshtml 和 \_ViewImports.cshtml 的作用将受到极大限制。原因后面细表。

下面我们来分别解决这2个问题。

**最灵活的方式： IViewLocationExpander**

有时候，我们的视图目录除了 controller 名称 和 view 名称2个变量外，还涉及到别的动态部分，比如上面的 Reports 相关 Controller，视图路径有更深的目录结构，而 controller 名称仅代表末级的目录。此时，我们需要一种更灵活的方式来处理： IViewLocationExpander，通过实现 IViewLocationExpander，我们可以得到一个 ViewLocationExpanderContext，然后据此更灵活地创建 view location formats。

对于我们要解决的目录层次问题，我们首先需要观察，然后会发现目录层次结构和 Controller 类型的命名空间是有对应关系的。例如如下定义：

1. using Microsoft.AspNetCore.Mvc;
3. namespace CustomizedViewLocation.Reports.AdHocReports.EmployeeReport
4. {
5. public class EmployeeReportController : Controller
6. {
7. public IActionResult Index() => View();
8. }
9. }

观察 EmployeeReportController 的命名空间 CustomizedViewLocation.Reports.AdHocReports.EmployeeReport以及 Index 视图对应的目录 /Reports/AdHocReports/EmployeeReport/Views/Index.cshtml 可以发现如下对应关系：

| **命名空间** | **视图路径** | **ViewLocationFormat** |
| --- | --- | --- |
| CustomizedViewLocation | 项目根路径 | / |
| Reports.AdHocReports | Reports/AdHocReports | 把整个命名空间以“.”为分割点掐头去尾，然后把“.”替换为“／” |
| EmployeeReport | EmployeeReport | Controller 名称 |
|  | Views | 固定目录 |
|  | Index.cshtml | 视图名称.cshtml |

所以我们 IViewLocationExpander 的实现类型主要是获取和处理 Controller 的命名空间。且看下面的代码。

1. *// NamespaceViewLocationExpander.cs*
3. using System;
4. using System.Collections.Generic;
5. using System.Linq;
6. using System.IO;
7. using Microsoft.Extensions.DependencyInjection;
8. using Microsoft.AspNetCore.Mvc.Razor;
9. using Microsoft.AspNetCore.Hosting;
10. using Microsoft.AspNetCore.Mvc;
11. using Microsoft.AspNetCore.Mvc.Controllers;
13. namespace CustomizedViewLocation
14. {
15. public class NamespaceViewLocationExpander : IViewLocationExpander
16. {
17. private const string VIEWS\_FOLDER\_NAME = "Views";
19. public IEnumerable<string> ExpandViewLocations(ViewLocationExpanderContext context, IEnumerable<string> viewLocations)
20. {
21. ControllerActionDescriptor cad = context.ActionContext.ActionDescriptor as ControllerActionDescriptor;
22. string controllerNamespace = cad.ControllerTypeInfo.Namespace;
23. int firstDotIndex = controllerNamespace.IndexOf('.');
24. int lastDotIndex = controllerNamespace.LastIndexOf('.');
25. if (firstDotIndex < 0)
26. return viewLocations;
28. string viewLocation;
29. if (firstDotIndex == lastDotIndex)
30. {
31. *// controller folder is the first level sub folder of root folder*
32. viewLocation = "/{1}/Views/{0}.cshtml";
33. }
34. else
35. {
36. string viewPath = controllerNamespace.Substring(firstDotIndex + 1, lastDotIndex - firstDotIndex - 1).Replace(".", "/");
37. viewLocation = $"/{viewPath}/{{1}}/Views/{{0}}.cshtml";
38. }
40. if (viewLocations.Any(l => l.Equals(viewLocation, StringComparison.InvariantCultureIgnoreCase)))
41. return viewLocations;
43. if (viewLocations is List<string> locations)
44. {
45. locations.Add(viewLocation);
46. return locations;
47. }
49. *// it turns out the viewLocations from ASP.NET Core is List<string>, so the code path should not go here.*
50. List<string> newViewLocations = viewLocations.ToList();
51. newViewLocations.Add(viewLocation);
52. return newViewLocations;
53. }
55. public void PopulateValues(ViewLocationExpanderContext context)
56. {
58. }
59. }
60. }

上面对命名空间的处理略显繁琐。其实你可以不用管，重点是我们可以得到 ViewLocationExpanderContext，并据此构建新的 view location format 然后与现有的 viewLocations 合并并返回给 ASP.NET Core。

细心的同学可能还注意到一个空的方法 PopulateValues，这玩意儿有什么用？具体作用可以参照这个 StackOverflow 的问题，基本上来说，一旦某个 Controller 及其某个 View 找到视图位置之后，这个对应关系就会缓存下来，以后就不会再调用 ExpandViewLocations方法了。但是，如果你有这种情况，就是同一个 Controller， 同一个视图名称但是还应该依据某些特别条件去找不同的视图位置，那么就可以利用 PopulateValues 方法填充一些特定的 Value， 这些 Value 会参与到缓存键的创建， 从而控制到视图位置缓存的创建。

下一步，把我们的 NamespaceViewLocationExpander 注册一下：

1. *// Startup.cs*
3. public void ConfigureServices(IServiceCollection services)
4. {
5. IMvcBuilder mvcBuilder = services.AddMvc();
6. mvcBuilder.AddRazorOptions(options =>
7. {
8. *// options.ViewLocationFormats.Add("/{1}/Views/{0}.cshtml"); we don't need this any more if we make use of NamespaceViewLocationExpander*
9. options.ViewLocationExpanders.Add(new NamespaceViewLocationExpander());
10. });
11. }

另外，有了 NamespaceViewLocationExpander， 我们就不需要前面对 ViewLocationFormats 的追加了，因为那种情况作为一种特例已经在 NamespaceViewLocationExpander 中处理了。  
至此，目录分层的问题解决了。

**\_ViewStart.cshtml 和 \_ViewImports 的起效机制与调整**

对这2个特别的视图，我们并不陌生，通常在 \_ViewStart.cshtml 里面设置 Layout 视图，然后每个视图就自动地启用了那个 Layout 视图，在 \_ViewImports.cshtml 里引入的命名空间和 TagHelper 也会自动包含在所有视图里。它们为什么会起作用呢？

\_ViewImports 的秘密藏在 RazorTemplateEngine 类 和 MvcRazorTemplateEngine 类中。

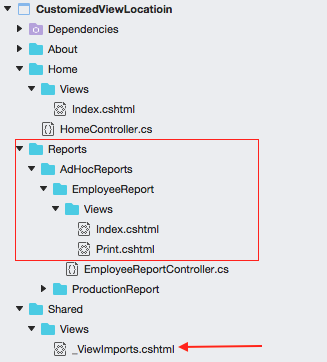
MvcRazorTemplateEngine 类指明了 "\_ViewImports.cshtml" 作为默认的名字。

1. *// MvcRazorTemplateEngine.cs 部分代码*
2. *// 完整代码: https://github.com/aspnet/Razor/blob/rel/2.0.0/src/Microsoft.AspNetCore.Mvc.Razor.Extensions/MvcRazorTemplateEngine.cs*
4. public class MvcRazorTemplateEngine : RazorTemplateEngine
5. {
6. public MvcRazorTemplateEngine(RazorEngine engine, RazorProject project)
7. : base(engine, project)
8. {
9. Options.ImportsFileName = "\_ViewImports.cshtml";
10. Options.DefaultImports = GetDefaultImports();
11. }
12. }

RazorTemplateEngine 类则表明了 Razor 是如何去寻找 \_ViewImports.cshtml 文件的。

1. *// RazorTemplateEngine.cs 部分代码*
2. *// 完整代码：https://github.com/aspnet/Razor/blob/rel/2.0.0/src/Microsoft.AspNetCore.Razor.Language/RazorTemplateEngine.cs*
4. public class RazorTemplateEngine
5. {
6. public virtual IEnumerable<RazorProjectItem> GetImportItems(RazorProjectItem projectItem)
7. {
8. var importsFileName = Options.ImportsFileName;
9. if (!string.IsNullOrEmpty(importsFileName))
10. {
11. return Project.FindHierarchicalItems(projectItem.FilePath, importsFileName);
12. }
14. return Enumerable.Empty<RazorProjectItem>();
15. }
16. }

FindHierarchicalItems 方法会返回一个路径集合，其中包括从视图当前目录一路到根目录的每一级目录下的 \_ViewImports.cshtml 路径。换句话说，如果从根目录开始，到视图所在目录的每一层目录都有 \_ViewImports.cshtml 文件的话，那么它们都会起作用。这也是为什么通常我们在 根目录下的 Views 目录里放一个 \_ViewImports.cshtml 文件就会被所有视图文件所引用，因为 Views 目录是是所有视图文件的父／祖父目录。那么如果我们的 \_ViewImports.cshtml 文件不在视图的目录层次结构中呢？



在这个 DI 为王的 ASP.NET Core 世界里，RazorTemplateEngine 也被注册为 DI 里的服务，因此我目前的做法继承 MvcRazorTemplateEngine 类，微调 GetImportItems 方法的逻辑，加入我们的特定路径，然后注册到 DI 取代原来的实现类型。代码如下：

1. *// ModuleRazorTemplateEngine.cs*
3. using System.Collections.Generic;
4. using System.Linq;
5. using Microsoft.AspNetCore.Mvc.Razor.Extensions;
6. using Microsoft.AspNetCore.Razor.Language;
8. namespace CustomizedViewLocation
9. {
10. public class ModuleRazorTemplateEngine : MvcRazorTemplateEngine
11. {
12. public ModuleRazorTemplateEngine(RazorEngine engine, RazorProject project) : base(engine, project)
13. {
14. }
16. public override IEnumerable<RazorProjectItem> GetImportItems(RazorProjectItem projectItem)
17. {
18. IEnumerable<RazorProjectItem> importItems = base.GetImportItems(projectItem);
19. return importItems.Append(Project.GetItem($"/Shared/Views/{Options.ImportsFileName}"));
20. }
21. }
22. }

然后在 Startup 类里把它注册到 DI 取代默认的实现类型。

1. *// Startup.cs*
3. *// using Microsoft.AspNetCore.Razor.Language;*
5. public void ConfigureServices(IServiceCollection services)
6. {
7. services.AddSingleton<RazorTemplateEngine, ModuleRazorTemplateEngine>();
9. IMvcBuilder mvcBuilder = services.AddMvc();
11. *// 其它代码省略*
12. }

下面是 \_ViewStart.cshtml 的问题了。不幸的是，Razor 对 \_ViewStart.cshtml 的处理并没有那么“灵活”，看代码就知道了。

1. *// RazorViewEngine.cs 部分代码*
2. *// 完整代码：https://github.com/aspnet/Mvc/blob/rel/2.0.0/src/Microsoft.AspNetCore.Mvc.Razor/RazorViewEngine.cs*
4. public class RazorViewEngine : IRazorViewEngine
5. {
6. private const string ViewStartFileName = "\_ViewStart.cshtml";
8. internal ViewLocationCacheResult CreateCacheResult(
9. HashSet<IChangeToken> expirationTokens,
10. string relativePath,
11. bool isMainPage)
12. {
13. var factoryResult = \_pageFactory.CreateFactory(relativePath);
14. var viewDescriptor = factoryResult.ViewDescriptor;
15. if (viewDescriptor?.ExpirationTokens != null)
16. {
17. for (var i = 0; i < viewDescriptor.ExpirationTokens.Count; i++)
18. {
19. expirationTokens.Add(viewDescriptor.ExpirationTokens[i]);
20. }
21. }
23. if (factoryResult.Success)
24. {
25. *// Only need to lookup \_ViewStarts for the main page.*
26. var viewStartPages = isMainPage ?
27. GetViewStartPages(viewDescriptor.RelativePath, expirationTokens) :
28. Array.Empty<ViewLocationCacheItem>();
29. if (viewDescriptor.IsPrecompiled)
30. {
31. \_logger.PrecompiledViewFound(relativePath);
32. }
34. return new ViewLocationCacheResult(
35. new ViewLocationCacheItem(factoryResult.RazorPageFactory, relativePath),
36. viewStartPages);
37. }
39. return null;
40. }
42. private IReadOnlyList<ViewLocationCacheItem> GetViewStartPages(
43. string path,
44. HashSet<IChangeToken> expirationTokens)
45. {
46. var viewStartPages = new List<ViewLocationCacheItem>();
48. foreach (var viewStartProjectItem in \_razorProject.FindHierarchicalItems(path, ViewStartFileName))
49. {
50. var result = \_pageFactory.CreateFactory(viewStartProjectItem.FilePath);
51. var viewDescriptor = result.ViewDescriptor;
52. if (viewDescriptor?.ExpirationTokens != null)
53. {
54. for (var i = 0; i < viewDescriptor.ExpirationTokens.Count; i++)
55. {
56. expirationTokens.Add(viewDescriptor.ExpirationTokens[i]);
57. }
58. }
60. if (result.Success)
61. {
62. *// Populate the viewStartPages list so that \_ViewStarts appear in the order the need to be*
63. *// executed (closest last, furthest first). This is the reverse order in which*
64. *// ViewHierarchyUtility.GetViewStartLocations returns \_ViewStarts.*
65. viewStartPages.Insert(0, new ViewLocationCacheItem(result.RazorPageFactory, viewStartProjectItem.FilePath));
66. }
67. }
69. return viewStartPages;
70. }
71. }

上面的代码里 GetViewStartPages 方法是个 private，没有什么机会让我们加入自己的逻辑。看了又看，好像只能从 \_razorProject.FindHierarchicalItems(path, ViewStartFileName) 这里着手。这个方法同样在处理 \_ViewImports.cshtml时用到过，因此和 \_ViewImports.cshtml 一样，从根目录到视图当前目录之间的每一层目录的 \_ViewStarts.cshtml 都会被引入。如果我们可以调整一下 FindHierarchicalItems 方法，除了完成它原本的逻辑之外，再加入我们对我们 /Shared/Views 目录的引用就好了。而 FindHierarchicalItems 这个方法是在 Microsoft.AspNetCore.Razor.Language.RazorProject 类型里定义的，而且是个 virtual 方法，而且它是注册在 DI 里的，不过在 DI 中的实现类型是 Microsoft.AspNetCore.Mvc.Razor.Internal.FileProviderRazorProject。我们所要做的就是创建一个继承自 FileProviderRazorProject 的类型，然后调整 FindHierarchicalItems 方法。

1. using System.Linq;
2. using System.Collections.Generic;
3. using Microsoft.AspNetCore.Mvc.Razor.Internal;
4. using Microsoft.AspNetCore.Razor.Language;
6. namespace CustomizedViewLocation
7. {
8. public class ModuleBasedRazorProject : FileProviderRazorProject
9. {
10. public ModuleBasedRazorProject(IRazorViewEngineFileProviderAccessor accessor)
11. : base(accessor)
12. {
14. }
16. public override IEnumerable<RazorProjectItem> FindHierarchicalItems(string basePath, string path, string fileName)
17. {
18. IEnumerable<RazorProjectItem> items = base.FindHierarchicalItems(basePath, path, fileName);
20. *// the items are in the order of closest first, furthest last, therefore we append our item to be the last item.*
21. return items.Append(GetItem("/Shared/Views/" + fileName));
22. }
23. }
24. }

完成之后再注册到 DI。

1. *// Startup.cs*
3. *// using Microsoft.AspNetCore.Razor.Language;*
5. public void ConfigureServices(IServiceCollection services)
6. {
7. *// services.AddSingleton<RazorTemplateEngine, ModuleRazorTemplateEngine>(); // we don't need this any more if we make use of ModuleBasedRazorProject*
8. services.AddSingleton<RazorProject, ModuleBasedRazorProject>();
10. IMvcBuilder mvcBuilder = services.AddMvc();
12. *// 其它代码省略*
13. }

有了 ModuleBasedRazorProject 我们甚至可以去掉之前我们写的 ModuleRazorTemplateEngine 类型了，因为 Razor 采用相同的逻辑 —— 使用 RazorProject 的 FindHierarchicalItems 方法 —— 来构建应用 \_ViewImports.cshtml 和 \_ViewStart.cshtml 的目录层次结构。所以最终，我们只需要一个类型来解决问题 —— ModuleBasedRazorProject。

回顾这整个思考和尝试的过程，很有意思，最终解决方案是自定义一个 RazorProject。是啊，毕竟我们的需求只是一个不同目录结构的 Razor Project，所以去实现一个我们自己的 RazorProject 类型真是再自然不过的了。

文本中的示例代码在这里