第 2 章 运 行 环 境

本章要点

- 定义运行环境
- 适配多个运行环境

2.1 定义运行环境

ASP.NET Core 应用程序允许配置多个运行环境，并可以针对某个环境执行特定的代码。这使应用程序更易于被管理。例如，在项目开发过程中，为了能直观地发现、排查错误，可以选择启用带有详细异常信息的错误页面；一旦项目交付给客户并正式上线运行，出于安全考虑，应用程序在运行过程中将使用自定义错误页，并隐藏详细的异常信息（可以选择将详细信息写入服务器的日志中，仅把错误的摘要信息返回给客户端）。又如，在开发测试阶段，应用程序应当使用测试数据库；当项目正式投入使用后，才使用真实的数据库。这样做可有效避免因测试过程中出现的错误使数据库遭到破坏。

为了让应用程序能识别出不同的运行环境，需要为每个运行环境命名。原则上，环境名称只是一个字符串对象，开发人员可以定义任意有效的命名。当然，ASP.NET Core 应用程序预定义了 3 个运行环境，方便开发人员直接使用。这些预定义名称通过 Environments 类（位于 Microsoft.Extensions.Hosting 命名空间）的静态字段公开。

（1）Development：表示当前应用程序处于开发阶段。

（2）Staging：表示当前应用程序的开发基本完成，处于预发布阶段。

（3）Production：表示应用程序已完成开发和测试，即将投入生产环境中使用。

可以通过命令行参数和环境变量为应用程序指定运行环境。配置项的字段名为 environment，命令行参数可以使用--environment 或/environment，环境变量可以使用 ASPNETCORE_ENVIRONMENT。

下面的示例通过命令行参数将应用程序的运行环境配置为 Development。

```
dotnet demo.dll --environment=Development
```

也可以使用环境变量配置。

```
set ASPNETCORE_ENVIRONMENT=Production         // Windows
export ASPNETCORE_ENVIRONMENT=Production      // Linux

dotnet demo.dll
```

当然，也可以通过直接编写代码设置运行环境。

```
WebApplicationOptions appopt = new()
{
    // 命令行参数
    Args = args,
    // 指定运行环境
    EnvironmentName = "Shared"
};
var builder = WebApplication.CreateBuilder(appopt);
// 创建应用程序
var app = builder.Build();

…
// 启动应用程序
app.Run();
```

首先实例化 WebApplicationOptions 对象，然后通过 EnvironmentName 属性设置运行环境，最后把此 WebApplicationOptions 对象传递给 CreateBuilder()方法。该示例使用了一个自定义的环境名称——Shared。应用程序运行后，可以在输出的日志记录中查看正在使用的运行环境，如图 2-1 所示。

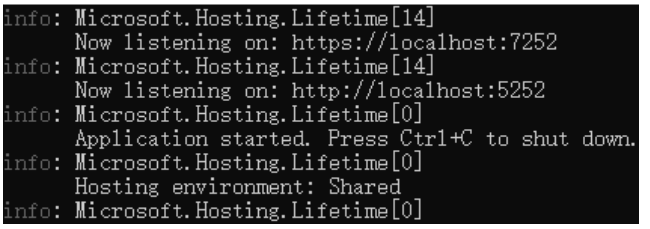

图 2-1　自定义的运行环境

本书推荐使用命令行参数或环境变量设置应用程序的运行环境，这样做比较灵活，切换环境不需要重新编译程序代码。

2.2　Is{EnvironmentName}扩展方法

ASP.NET Core 应用程序支持多个环境配置，在运行阶段需要根据不同的环境执行相应的代码。

WebApplication 类有一个名为 Environment 的公共属性，声明类型为 IWebHostEnvironment

接口，此接口派生自 IHostEnvironment 接口，因此它继承了 IHostEnvironment 接口的 EnvironmentName 属性。也就是说，应用程序代码通过 IWebHostEnvironment. EnvironmentName 属性可以获知当前正在使用的运行环境名称。虽然该属性定义了 set 访问器，但是不要在代码中修改它的值，那样做会导致应用程序对运行环境作出错误的判断。

程序代码可以根据 EnvironmentName 属性的值判断应用程序正在使用的运行环境。在实际使用中，调用扩展方法会更方便。这些扩展方法以 Is{EnvironmentName}的格式命名，由 HostEnvironmentEnvExtensions 类公开。具体说明如下。

（1）IsDevelopment()方法：如果当前正在使用的环境名称为 Development，该方法就返回 true，否则返回 false。

（2）IsProduction()方法：如果正在使用的环境名称为 Production 就返回 true，否则返回 false。

（3）IsStaging()方法：同上，判断当前的运行环境是否为 Staging。

（4）IsEnvironment()方法：如果所使用的运行环境名称在 Development、Production 和 Staging 之外，就需要调用该方法进行判断。

下面的示例将根据多个运行环境构建 HTTP 管线。

```
var builder = WebApplication.CreateBuilder(args);

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.MapGet("/", () =>"当前项目正处在开发阶段");
}
else if (app.Environment.IsProduction())
{
    app.MapGet("/", () =>"当前项目已投放使用");
}
else if(app.Environment.IsEnvironment("NoAPI"))
{
    app.MapGet("/", () =>"此版本不公开 Web API");
}
…

app.Run();
```

上述代码中，针对不同的环境调用 MapGet()方法。由于方法调用被写在 if…else 语句中，这使得 MapGet()方法在整个 HTTP 管线的构建中仅调用一次。传递给 MapGet()方法的第 1 个参数值“/”表示匹配 URL 的根地址（如 https://testhost/）。在此 if...else 语句之后如果还需要调用 MapGet()方法，那就不能再匹配根地址“/”，否则会导致该地址存在多个匹配项，应用程序无法选择执行哪个地址，进而发生错误。

在最后一个 else 分支中，由于 NoAPI 环境是自定义的名称，需要调用 IsEnvironment()方法验证当前环境。

假设通过命令行参数设置运行环境为 NoAPI，那么示例程序运行后将执行以下代码，结果如图 2-2 所示。

```
app.MapGet("/", () =>"此版本不公开 Web API");
```

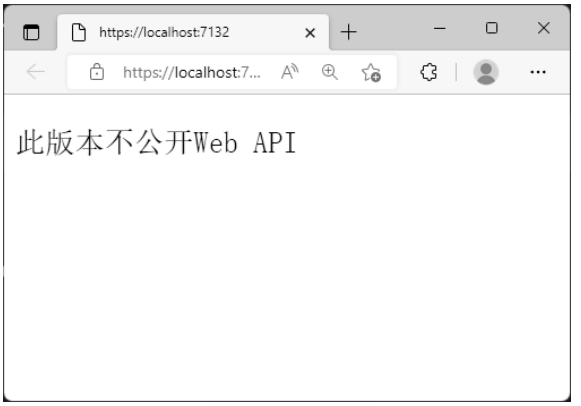

图 2-2　应用程序运行在 NoAPI 环境中

2.3　多运行环境下的配置文件

ASP.NET Core 项目模板会生成一个名为 appsettings.json 的配置文件，默认的 Web 应用程序配置除了加载 appsettings.json 文件外，还会查找以 appsettings.{EnvironmentName}.json 格式命名的配置文件。这些配置文件是专用于特定运行环境的。例如，若当前运行环境为 Production，应用程序在启动时会加载 appsettings.json 和 appsettings.Production.json 两个配置文件。

下面给出一个示例。应用程序项目中包含以下 3 个配置文件。

（1）appsettings.json：适用于所有运行环境的配置文件。

（2）appsettings.Demo.json：适用于名为 Demo 的运行环境的配置文件。

（3）appsettings.Development.json：适用于名为 Development 的运行环境的配置文件。

3 个配置文件都包含一个相同的字段——my_key。

```
// appsettings.json
{
    "my_key": "通用于各环境的配置"
}

// appsettings.Development.json
{
    "my_key": "适用于开发阶段的配置"
}

// appsettings.Demo.json
{
```

```
    "my_key": "适用于 Demo 运行环境的配置"
}
```

在应用程序中读入 my_key 的值并返回给客户端。

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/", () =>
{
    // 读取 my_key 配置项的值
    string cfgValue = app.Configuration["my_key"];
    return $"【my_key】的值：{cfgValue}";
});

app.Run();
```

app.Configuration 属性中存储了应用程序从各个配置源加载的配置信息（包括 appsettings.json、appsettings.Development.json 等配置文件），其数据类型实现了 IConfiguration 接口。程序代码可以通过以下索引器访问特定配置项的值。

```
[DefaultMember("Item")]
public interface IConfiguration
{
    // 支持读写某个配置项的值
    string this[string key] { get; set; }

    …
}
```

其中，参数 key 用于指定配置项的字段名，在本示例中是 my_key。

假设以 Demo 环境运行示例程序，并从 Web 浏览器访问其根 URL，将得到如图 2-3 所示的响应信息。

如果给应用程序指定的运行环境没有匹配的配置文件，默认会读取 appsettings.json 文件中的配置信息。例如，以 Test 环境运行应用程序，会得到如图 2-4 所示的响应信息。

图 2-3　以 Demo 环境运行示例程序

图 2-4　默认读取 appsettings.json 文件中的配置

2.4　用于环境筛选的 Razor 标记

在 ASP.NET Core 项目中，运行于服务器的 HTML 标记都会使用到 Razor 标记语法。它可以在 HTML 标记中插入 C#或 VB（Visual Basic）代码。当客户端（通常是 Web 浏览器）发出请求后，服务器会执行 Razor 标记并生成最终呈现给客户端的 HTML 代码。

HTML 标记本身无法检查 ASP.NET Core 应用程序正在使用的运行环境，因此需要借助 Razor 标记对运行环境进行分析，不同运行环境下显示相应的内容。例如，如果应用程序运行在 Development 环境下，为了便于测试，开发人员会考虑在 HTML 页面中显示一些调试信息；而当应用程序运行在 Production 环境下时，由于应用程序已投入生产环境中使用，就不再需要在 HTML 页面上显示详细的调试信息了。又如，某个应用程序即将发布到两台服务器上，运行于 A 服务器的应用程序是面向内网用户的，设置运行环境为 ENV1；而运行在 B 服务器上的应用程序是面向外网用户的，设置运行环境为 ENV2。在组织 HTML 标记时，如果运行环境为 ENV1，就显示登录用户的完整信息，并且显示公司内部通知；如果运行环境为 ENV2，就显示登录用户的简要信息，并且不显示公司内部通知。

Razor 语法使用 EnvironmentTagHelper 类（位于 Microsoft.AspNetCore.Mvc.TagHelpers 命名空间）扩展 HTML 功能，对运行环境进行筛选。该类定义了 3 个公共属性。

（1）Include：指定一个环境列表，只要当前应用程序所使用的环境在此列表中，就会呈现 HTML 内容。

（2）Exclude：其含义为“排除”，即指定一个环境列表，如果应用程序正在使用的环境在此列表中，就不会呈现 HTML 内容。其逻辑与 Include 属性相反。

（3）Names：指定一个环境列表，如果应用程序正在使用的环境在此列表中，就会呈现 HTML 内容。其处理方式与 Include 属性相同。

这 3 个属性的类型都是字符串，指定运行环境列表时，环境名称之间使用逗号（英文）分隔。例如：

```
<environment include="Development,Test">
<div>你好，世界</div>
</environment>
```

上述示例中 Include 属性指定了两个环境名称——Development 和 Test，只要当前应用程序的运行环境为 Development 或 Test，那么服务器就会将“你好，世界”输出到响应流中（呈现在 Web 浏览器上）；如果程序正在使用的环境不是 Development，也不是 Test，那么“你好，世界”将不会输出到响应流中。

下面的示例将使用 Razor Pages 呈现 HTML，并在 HTML 页面上通过 environment 标记筛选运行环境。大致步骤如下。

（1）在构建 app 前，需要向服务容器注册 Razor Pages 功能。

```
var builder = WebApplication.CreateBuilder(args);
// 添加 Razor Pages 功能
```

```
builder.Services.AddRazorPages();

var app = builder.Build();
```

（2）当 app 构建后，向 HTTP 管线中添加 Razor Pages 终结点。

```
app.MapRazorPages();
```

（3）调用 MapFallback()方法，向 HTTP 管线中添加一个“回退”终结点，当客户端所请求的 URL 无效时执行。

```
app.MapFallback(() =>"找不到指定的资源");
```

（4）开始执行 app。

```
app.Run();
```

（5）在项目目录下新建子目录，命名为 Pages。

（6）在 Pages 目录下新建文件，命名为 index.cshtml。

（7）在 index.cshtml 中输入以下内容。

```
@page
@addTagHelper Microsoft.AspNetCore.Mvc.TagHelpers.EnvironmentTagHelper,
Microsoft.AspNetCore.Mvc.TagHelpers

<html lang="zh-cn">
<head>
<meta charset="utf-8"/>
<title>Demo</title>
</head>
<body>
<environment include="Development">
<span>项目开发中……</span>
</environment>
<environment include="Production">
<span>项目已上线</span>
</environment>
</body>
</html>
```

在 HTML 文档的第 1 行必须写上@page 指令，表示此页面用于 Razor Pages。第 2 行使用@addTagHelper 指令将 EnvironmentTagHelper 类导入，其格式为“<完整类名>, <程序集名>”。EnvironmentTagHelper 类所在的程序集名称与命名空间名称相同，都是 Microsoft.AspNetCore.Mvc.TagHelpers。

当运行环境为 Development 时，将呈现一个元素，包含文本“项目开发中.....”；当运行环境为 Production 时，元素中的文本呈现为“项目已上线”。

假设应用名称为 DemoApp，在启动应用程序时通过命令行参数指定运行环境为 Development。

```
dotnet DemoApp.dll --environment=Development
```

Web 浏览器中呈现的内容如图 2-5 所示。

按 Ctrl+C 快捷键退出应用程序，接着以 Production 环境运行。

```
dotnet DemoApp.dll --environment=Production
```

在 Web 浏览器中导航到应用程序 URL，页面呈现效果如图 2-6 所示。

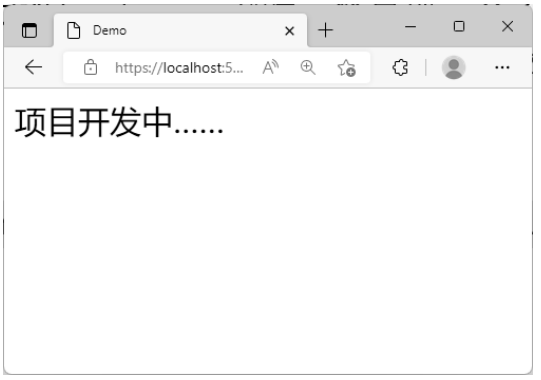

图 2-5　以 Development 环境运行

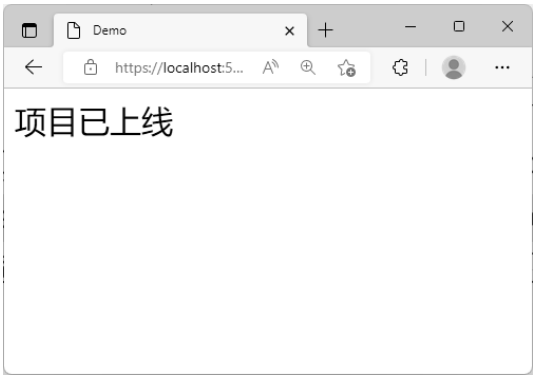

图 2-6　以 Production 环境运行

2.5　运行环境与依赖注入

被添加到服务容器中的对象都可以使用依赖注入。由.NET 运行时自动创建类型实例，并传递该实例所需要访问的对象引用。最常见的做法是通过构造函数进行注入。假设 A 类的实例中需要访问 B 类的实例，那么在 A 类的构造函数中声明一个 B 类的参数，并在构造函数内部接收 B 类实例的引用，存放到 A 类的私有字段中。最后将 A、B 两个类都注册到服务容器中。当调用者获取 A 类的实例时，服务容器会自动调用 A 类的构造函数，且自动将 B 类的实例传递给 A 类的构造函数。依赖注入是 ASP.NET Core 的一个核心功能，在后续章节中会详细阐述。

另外，像 MVC 中的控制器类，或者 Razor Pages 中的页面模型类，尽管开发人员不需要将它们放进服务容器中，但它们的实例是由.NET 运行时自动创建的，因此也支持依赖注入。

而包含应用程序运行环境信息的相关类型在程序初始化过程也会被注册到服务容器中。对于使用通用主机启动的.NET 应用程序，可以通过 IHostEnvironment 接口得知当前运行环境；而 ASP.NET Core 应用程序是在通用主机的基础上启用 Web 主机的，因此除了 IHostEnvironment 接口，还可以通过 IWebHostEnvironment 接口获取运行环境信息。这是因为 IWebHost-Environment 接口派生自 IHostEnvironment 接口，所以继承了 EnvironmentName 属性。

在下面的示例中，将定义一个名为 TestService 的类。该类公开一个 GetHubName()方法，此方法会根据应用程序当前所处的运行环境返回不同的字符串。

```
public class TestService
{
```

```
    readonly IHostEnvironment _hostEnv;

    public TestService(IHostEnvironment hostenv)
    {
        _hostEnv = hostenv;
    }

    public string GetHubName()
    {
        return _hostEnv.EnvironmentName switch
        {
            "DATA_CNT"      =>"数据中心",
            "REPORT_SVR"    =>"报表中心",
            "NEWS_GRP"      =>"新闻组",
            _               =>"未知环境"
        };
    }
}
```

在TestService类中要先声明一个IHostEnvironment接口类型的私有字段，命名为_hostEnv，在构造函数中进行赋值。对象的引用来自构造函数参数，此参数会通过依赖注入自动获得所需对象的引用。

在GetHubName()方法中，使用switch子句对EnvironmentName属性的值进行分析，如果当前运行环境的名称为DATA_CNT，则返回“数据中心”；如果当前的运行环境是REPORT_SVR，则返回“报表中心”；如果当前的运行环境是NEWS_GRP，则返回“新闻组”；如果运行环境的名称与筛选的表达式均不匹配，则返回“未知环境”。

该示例也可以使用IWebHostEnvironment接口。

```
readonly IWebHostEnvironment _hostEnv;

public TestService(IWebHostEnvironment hostenv)
{
    _hostEnv = hostenv;
}
```

在项目的Program.cs文件（即ASP.NET Core应用程序的初始化代码）中，创建WebApplicationBuilder实例后，需要将上文定义的TestService类注册到服务容器中。

```
var builder = WebApplication.CreateBuilder(args);
// 向服务容器注册自定义类型
builder.Services.AddSingleton<TestService>();
…
```

AddSingleton()方法表示TestService类型将使用单实例模式，即在整个应用程序生命周期内，它只创建一个对象实例（仅调用一次构造函数）。

在调用Build()方法后，需要为app对象添加一个绑定到根URL的终结点，并在处理代码中调用GetRequiredService()方法主动获取TestService实例，然后调用GetHubName()方法，将

结果返回给客户端。代码如下。

```
app.MapGet("/", () =>
{
    // 获取自定义服务类的实例
    var svr = app.Services.GetRequiredService<TestService>();
    // 调用服务的实例方法
    return svr.GetHubName();
});
```

以 NEWS_GRP 环境启动应用程序，然后使用 Web 浏览器访问应用程序的根地址，服务器返回“新闻组”，如图 2-7 所示。

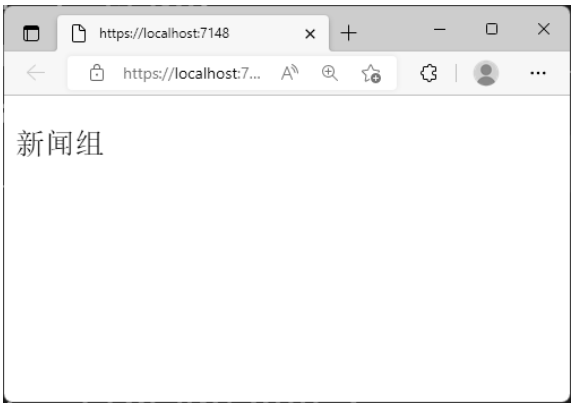

图 2-7　返回特定于运行环境的字符串