

管道（上篇）

ASP.NET Core 是一个 Web 开发平台，而不是一个单纯的开发框架。这是因为 ASP.NET Core 有一个极具扩展能力的请求处理管道，我们可以通过对这个管道的定制来满足各种场景下的 HTTP 处理需求。ASP.NET Core 应用的很多特性（如路由、会话、缓存、认证、授权等）都是通过对管道的定制来实现的，我们可以通过管道定制在 ASP.NET Core 平台上创建自己的 Web 框架。由于这部分内容是本书的核心，所以分为 3 章（第 11 章至第 13 章）对请求处理管道进行全方面讲解。

11.1 管道式的请求处理

HTTP 协议自身的特性决定了任何一个 Web 应用的工作模式都是监听、接收并处理 HTTP 请求，并且最终对请求予以响应。HTTP 请求处理是管道式设计典型的应用场景：可以根据具体的需求构建一个管道，接收的 HTTP 请求像水一样流入这个管道，组成这个管道的各个环节依次对其做相应的处理。虽然 ASP.NET Core 的请求处理管道从设计上来讲是非常简单的，但是具体的实现则涉及很多细节，为了使读者对此有深刻的理解，需要从编程的角度先了解 ASP.NET Core 管道式的请求处理方式。

11.1.1 两个承载体系

ASP.NET Core 框架目前存在两个承载（Hosting）系统。ASP.NET Core 最初提供了一个以 `IWebHostBuilder/IWebHost` 为核心的承载系统，其目的很单纯，就是通过图 11-1 所示的形式承载以服务器和中间件管道构建的 Web 应用。ASP.NET Core 3 依然支持这样的应用承载方式，但是本书不会涉及这种“过时”的承载方式。

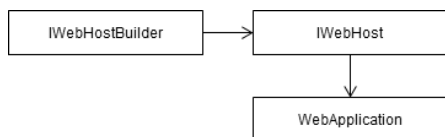


图 11-1 基于 `IWebHostBuilder/IWebHost` 应用的承载方式

除了承载 Web 应用本身，我们还有针对后台服务的承载需求，为此微软推出了以 IHostBuilder/IHost 为核心的承载系统，我们在第 10 章中已经对该系统做了详细的介绍。实际上，Web 应用本身就是一个长时间运行的后台服务，我们完全可以定义一个承载服务，从而将 Web 应用承载于这个系统中。如图 11-2 所示，这个用来承载 ASP.NET Core 应用的承载服务类型为 GenericWebHostService，这是一个实现了 IHostedService 接口的内部类型。

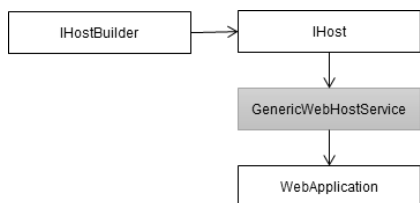


图 11-2 基于 IHostBuilder/IHost 应用的承载方式

IHostBuilder 接口上定义了很多方法（其中很多是扩展方法），这些方法的目的主要包括以下两点：第一，为创建的 IHost 对象及承载的服务在依赖注入框架中注册相应的服务；第二，为服务承载和应用提供相应的配置。其实 IWebHostBuilder 接口同样定义了一系列方法，除了这里涉及的两点，支撑 ASP.NET Core 应用的中间件也是由 IWebHostBuilder 注册的。

即使采用基于 IHostBuilder/IHost 的承载系统，我们依然会使用 IWebHostBuilder 接口。虽然我们不再使用 IWebHostBuilder 的宿主构建功能，但是定义在 IWebHostBuilder 上的其他 API 都是可以使用的。具体来说，可以调用定义在 IHostBuilder 接口和 IWebHostBuilder 接口的方法（大部分为扩展方法）来注册依赖服务与初始化配置系统，两者最终会合并在一起。利用 IWebHostBuilder 接口注册的中间件会提供给 GenericWebHostService，用于构建 ASP.NET Core 请求处理管道。

在基于 IHostBuilder/IHost 的承载系统中复用 IWebHostBuilder 的目的是通过如下所示的 ConfigureWebHost 扩展方法达成的，GenericWebHostService 服务也是在这个方法中被注册的。ConfigureWebHostDefaults 扩展方法则会在此基础上做一些默认设置（如 KestrelServer），后续章节的实例演示基本上会使用这个方法。

```

public static class GenericHostWebHostBuilderExtensions
{
    public static IHostBuilder ConfigureWebHost(this IHostBuilder builder,
        Action<IWebHostBuilder> configure);
}

public static class GenericHostBuilderExtensions
{
    public static IHostBuilder ConfigureWebHostDefaults(this IHostBuilder builder,
        Action<IWebHostBuilder> configure);
}
  
```

对 IWebHostBuilder 接口的复用导致很多功能都具有两种编程方式，虽然这样可以最大限度地复用和兼容定义在 IWebHostBuilder 接口上众多的应用编程接口，但笔者并不喜欢这样略显混乱的编程模式，这一点在下一个版本中也许会得到改变。

11.1.2 请求处理管道

下面创建一个最简单的 Hello World 程序。这是一个控制台应用，之前演示的大部分实例的 SDK 都采用 “Microsoft.NET.Sdk”，作为一个 ASP.NET Core Web 应用，对应的项目的 SDK 一般采用 “Microsoft.NET.Sdk.Web”。由于这种 SDK 会自动将常用的依赖或者引用添加进来，所以不需要在项目文件中显式添加针对 “Microsoft.AspNetCore.App” 的框架引用。

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp3.0</TargetFramework>
  </PropertyGroup>
</Project>
```

这个程序由如下所示的几行代码组成。运行这个程序之后，一个名为 KestrelServer 的服务器将会启动并绑定到本机上的 5000 端口进行请求监听。针对所有接收到的请求，我们都会采用 “Hello World” 字符串作为响应的主体内容。

```
class Program
{
    static void Main()
    {
        Host.CreateDefaultBuilder()
            .ConfigureWebHost(builder => builder
                .Configure(app => app.Run(context =>
                    context.Response.WriteAsync("Hello World"))))
            .Build()
            .Run();
    }
}
```

从如上所示的代码片段可以看出，我们利用第 10 章介绍的承载系统来承载一个 ASP.NET Core 应用。在调用 Host 类型的静态方法 CreateDefaultBuilder 创建了一个 IHostBuilder 对象之后，我们调用它的 ConfigureWebHost 方法对 ASP.NET Core 应用的请求处理管道进行定制。HTTP 请求处理流程始于对请求的监听与接收，终于对请求的响应，这两项工作均由同一个对象来完成，我们称之为服务器（Server）。ASP.NET Core 请求处理管道必须有一个服务器，它是整个管道的“龙头”。在演示程序中，我们调用 IWebHostBuilder 接口的 UseKestrel 扩展方法为后续构建的管道注册了一个名为 KestrelServer 的服务器。

当承载服务 GenericWebHostService 被启动之后，定制的请求处理管道会被构建出来，管道的服务器随后会绑定到一个预设的端口（如 KestrelServer 默认采用 5000 作为监听端口）开始监听请求。HTTP 请求一旦抵达，服务器会将其标准化，并分发给管道后续的节点，我们将位于服务器之后的节点称为中间件（Middleware）。

每个中间件都具有各自独立的功能，如专门实现路由功能的中间件、专门实施用户认证和授权的中间件。所谓的管道定制主要体现在根据具体需求选择对应的中间件来构建最终的管道。在演示程序中，我们调用 IWebHostBuilder 接口的 Configure 方法注册了一个中间件，用于响应 “Hello World” 字符串。具体来说，这个用来注册中间件的 Configure 方法具有一个类型为

Action<IApplicationBuilder>的参数，我们提供的中间件就注册到提供的 IApplicationBuilder 对象上。由服务器和中间件组成的请求处理管道如图 11-3 所示。

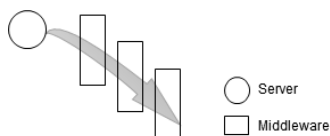


图 11-3 由服务器和中间件组成的请求处理管道

建立在 ASP.NET Core 之上的应用基本上是根据某个框架开发的。一般来说，开发框架本身就是通过某一个或者多个中间件构建起来的。以 ASP.NET Core MVC 开发框架为例，它借助“路由”中间件实现了请求与 Action 之间的映射，并在此基础上实现了激活（Controller）、执行（Action）及呈现（View）等一系列功能。应用程序可以视为某个中间件的一部分，如果一定要将它独立出来，由服务器、中间件和应用组成的管道如图 11-4 所示。

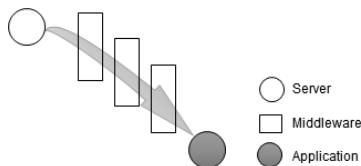


图 11-4 由服务器、中间件和应用组成的管道

11.1.3 中间件

ASP.NET Core 的请求处理管道由一个服务器和一组中间件组成，位于“龙头”的服务器负责请求的监听、接收、分发和最终的响应，而针对该请求的处理则由后续的中间件来完成。如果读者希望对请求处理管道具有深刻的认识，就需要对中间件有一定程度的了解。

RequestDelegate

从概念上可以将请求处理管道理解为“请求消息”和“响应消息”流通的管道，服务器将接收的请求消息从一端流入管道并由相应的中间件进行处理，生成的响应消息反向流入管道，经过相应中间件处理后由服务器分发给请求者。但从实现的角度来讲，管道中流通的并不是所谓的请求消息与响应消息，而是一个针对当前请求创建的上下文。这个上下文被抽象成如下这个 HttpContext 类型，我们利用 HttpContext 不仅可以获取针对当前请求的所有信息，还可以直接完成针对当前请求的所有响应工作。

```
public abstract class HttpContext
{
    public abstract HttpRequest Request { get; set; }
    public abstract HttpResponse Response { get; }
    ...
}
```

既然流入管道的只有一个共享的 HttpContext 上下文，那么一个 Func<HttpContext, Task>对象就可以表示处理 HttpContext 的操作，或者用于处理 HTTP 请求的处理器。由于这个委托对象非常重

要，所以 ASP.NET Core 专门定义了如下这个名为 `RequestDelegate` 的委托类型。既然有这样一个专门的委托对象来表示“针对请求的处理”，那么中间件是否能够通过该委托对象来表示？

```
public delegate Task RequestDelegate(HttpContext context);
```

Func<RequestDelegate, RequestDelegate>

实际上，组成请求处理管道的中间件可以表示为一个类型为 `Func<RequestDelegate, RequestDelegate>` 的委托对象，但初学者很难理解这一点，所以下面对此进行简单的解释。由于 `RequestDelegate` 可以表示一个 HTTP 请求处理器，所以由一个或者多个中间件组成的管道最终也体现为一个 `RequestDelegate` 对象。对于图 11-5 所示的中间件 `Foo` 来说，后续中间件 (`Bar` 和 `Baz`) 组成的管道体现为一个 `RequestDelegate` 对象，该对象会作为中间件 `Foo` 输入，中间件 `Foo` 借助这个委托对象将当前 `HttpContext` 分发给后续管道做进一步处理。

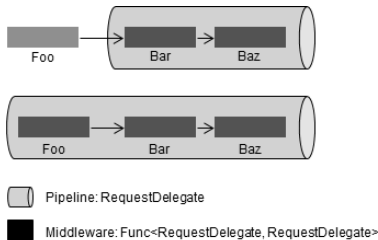


图 11-5 中间件

表示中间件的 `Func<RequestDelegate, RequestDelegate>` 对象的输出依然是一个 `RequestDelegate` 对象，该对象表示将当前中间件与后续管道进行“对接”之后构成的新管道。对于表示中间件 `Foo` 的委托对象来说，返回的 `RequestDelegate` 对象体现的就是由 `Foo`、`Bar` 和 `Baz` 组成的请求处理管道。

既然原始的中间件是通过一个 `Func<RequestDelegate, RequestDelegate>` 对象表示的，就可以直接注册这样一个对象作为中间件。中间件的注册可以通过调用 `IWebHostBuilder` 接口的 `Configure` 扩展方法来完成，该方法的参数是一个 `Action<IApplicationBuilder>` 类型的委托对象，可以通过调用 `IApplicationBuilder` 接口的 `Use` 方法将表示中间件的 `Func<RequestDelegate, RequestDelegate>` 对象添加到当前中间件链条上。

```
public static class WebHostBuilderExtensions
{
    public static IWebHostBuilder Configure(this IWebHostBuilder hostBuilder,
        Action<IApplicationBuilder> configureApp);
}

public interface IApplicationBuilder
{
    IApplicationBuilder Use(Func<RequestDelegate, RequestDelegate> middleware);
}
```

在如下所示的代码片段中，我们创建了两个 `Func<RequestDelegate, RequestDelegate>` 对象，它们会在响应中写入两个字符串（“Hello”和“World!”）。在针对 `IWebHostBuilder` 接口的

Configure 方法的调用中, 可以调用 IApplicationBuilder 接口的 Use 方法将这两个委托对象注册为中间件。

```
class Program
{
    static void Main()
    {
        static RequestDelegate Middleware1(RequestDelegate next)
            => async context =>
        {
            await context.Response.WriteAsync("Hello");
            await next(context);
        };
        static RequestDelegate Middleware2(RequestDelegate next)
            => async context =>
        {
            await context.Response.WriteAsync(" World!");
        };

        Host.CreateDefaultBuilder().ConfigureWebHostDefaults(builder => builder
            .Configure(app => app
                .Use(Middleware1)
                .Use(Middleware2)))
            .Build()
            .Run();
    }
}
```

由于我们注册了如上所示的两个中间件, 所以它们会按照注册的顺序对分发给它们的请求进行处理。运行该程序后, 如果利用浏览器对监听地址 (“http://localhost:5000”) 发送请求, 那么两个中间件写入的字符串会以图 11-6 所示的形式呈现出来。(S1101)

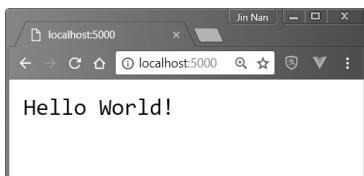


图 11-6 利用注册的中间件处理请求

虽然可以直接采用原始的 `Func<RequestDelegate, RequestDelegate>` 对象来定义中间件, 但是在大部分情况下, 我们依然倾向于将自定义的中间件定义成一个具体的类型。至于中间件类型的定义, ASP.NET Core 提供了如下两种不同的形式可供选择。

- 强类型定义: 自定义的中间件类型显式实现预定义的 `IMiddleware` 接口, 并在实现的方法中完成针对请求的处理。
- 基于约定的定义: 不需要实现任何接口或者继承某个基类, 只需要按照预定义的约定来定义中间件类型。

Run 方法的本质

在演示的 Hello World 应用中，我们调用 `IApplicationBuilder` 接口的 `Run` 扩展方法注册了一个 `RequestDelegate` 对象来处理请求，实际上，该方法仅仅是按照如下方式注册了一个中间件。由于注册的中间件并不会将请求分发给后续的中间件，如果调用 `IApplicationBuilder` 接口的 `Run` 方法后又注册了其他的中间件，后续中间件的注册将毫无意义。

```
public static class RunExtensions
{
    public static void Run(this IApplicationBuilder app, RequestDelegate handler)
        => app.Use( => handler);
}
```

11.1.4 定义强类型中间件

如果采用强类型的中间件类型定义方式，只需要实现如下这个 `IMiddleware` 接口，该接口定义了唯一的 `InvokeAsync` 方法，用于实现中间件针对请求的处理。这个 `InvokeAsync` 方法定义了两个参数：第一个参数是代表当前请求上下文的 `HttpContext` 对象，第二个参数是代表后续中间件组成的管道的 `RequestDelegate` 对象，如果当前中间件最终需要将请求分发给后续中间件进行处理，只需要调用这个委托对象即可，否则应用针对请求的处理就到此为止。

```
public interface IMiddleware
{
    Task InvokeAsync(HttpContext context, RequestDelegate next);
}
```

在如下所示的代码片段中，我们定义了一个实现了 `IMiddleware` 接口的 `StringContentMiddleware` 中间件类型，在实现的 `InvokeAsync` 方法中，可以将构造函数中指定的字符串作为响应的内容。由于中间件最终是采用依赖注入的方式来提供的，所以需要预先对它们进行服务注册，针对 `StringContentMiddleware` 的服务注册是通过调用 `IHostBuilder` 接口的 `ConfigureServices` 方法完成的。

```
class Program
{
    static void Main()
    {
        Host.CreateDefaultBuilder()
            .ConfigureServices(svcs => svcs.AddSingleton(
                new StringContentMiddleware("Hello World!")))
            .ConfigureWebHost(builder => builder
                .Configure(app => app.UseMiddleware<StringContentMiddleware>()))
            .Build()
            .Run();
    }

    private sealed class StringContentMiddleware : IMiddleware
    {
        private readonly string contents;
        public StringContentMiddleware(string contents)
    }
}
```

```

=> _contents = contents;
public Task InvokeAsync(HttpContext context, RequestDelegate next)
=> context.Response.WriteAsync(_contents);
}
}

```

针对中间件自身的注册则体现在针对 `IWebHostBuilder` 接口的 `Configure` 方法的调用上，最终通过调用 `IApplicationBuilder` 接口的 `UseMiddleware<TMiddleware>` 方法来注册中间件类型。如下面的代码片段所示，在注册中间件类型时，可以以泛型参数的形式来指定中间件类型，也可以调用另一个非泛型的方法重载，直接通过 `Type` 类型的参数来指定中间件类型。值得注意的是，这两个方法均提供了一个参数 `params`，它是为针对“基于约定的中间件”注册设计的，当我们注册一个实现了 `IMiddleware` 接口的强类型中间件的时候是不能指定该参数的。启动该程序后利用浏览器访问监听地址，依然可以得到图 11-6 所示的输出结果。(S1102)

```

public static class UseMiddlewareExtensions
{
    public static IApplicationBuilder UseMiddleware<TMiddleware>(
        this IApplicationBuilder app, params object[] args);
    public static IApplicationBuilder UseMiddleware(this IApplicationBuilder app,
        Type middleware, params object[] args);
}

```

11.1.5 按照约定定义中间件

可能我们已经习惯了通过实现某个接口或者继承某个抽象类的扩展方式，但是这种方式有时显得约束过重，不够灵活，所以可以采用另一种基于约定的中间件类型定义方式。这种定义方式比较自由，因为它并不需要实现某个预定义的接口或者继承某个基类，而只需要遵循一些约定即可。自定义中间件类型的约定主要体现在如下几个方面。

- 中间件类型需要有一个有效的公共实例构造函数，该构造函数要求必须包含一个 `RequestDelegate` 类型的参数，当前中间件利用这个委托对象实现针对后续中间件的请求分发。构造函数不仅可以包含任意其他参数，对于 `RequestDelegate` 参数出现的位置也不做任何约束。
- 针对请求的处理实现在返回类型为 `Task` 的 `InvokeAsync` 方法或者 `Invoke` 方法中，它们的第一个参数表示当前请求上下文的 `HttpContext` 对象。对于后续的参数，虽然约定并未对此做限制，但是由于这些参数最终由依赖注入框架提供，所以相应的服务注册必须存在。

采用这种方式定义的中间件类型同样是调用前面介绍的 `UseMiddleware` 方法和 `UseMiddleware<TMiddleware>` 方法进行注册的。由于这两个方法会利用依赖注入框架来提供指定类型的中间件对象，所以它会利用注册的服务来提供传入构造函数的参数。如果构造函数的参数没有对应的服务注册，就必须在调用这个方法的时候显式指定。

在如下所示的代码片段中，我们定义了一个名为 `StringContentMiddleware` 的中间件类型，在执行这个中间件时，它会将预先指定的字符串作为响应内容。`StringContentMiddleware` 的构造

函数具有两个额外的参数：`contents` 表示响应内容，`forwardToNext` 则表示是否需要将请求分发给后续中间件进行处理。在调用 `UseMiddleware<TMiddleware>` 扩展方法对这个中间件进行注册时，我们显式指定了响应的内容，至于参数 `forwardToNext`，我们之所以没有每次都显式指定，是因为这是一个具有默认值的参数。(S1103)

```
class Program
{
    static void Main()
    {
        Host.CreateDefaultBuilder()
            .ConfigureWebHostDefaults(builder => builder
                .Configure(app => app
                    .UseMiddleware<StringContentMiddleware>("Hello")
                    .UseMiddleware<StringContentMiddleware>(" World!", false)))
            .Build()
            .Run();
    }

    private sealed class StringContentMiddleware
    {
        private readonly RequestDelegate next;
        private readonly string contents;
        private readonly bool _forwardToNext;

        public StringContentMiddleware(RequestDelegate next, string contents,
            bool forwardToNext = true)
        {
            next = next;
            _forwardToNext = forwardToNext;
            contents = contents;
        }

        public async Task Invoke(HttpContext context)
        {
            await context.Response.WriteAsync(_contents);
            if (forwardToNext)
            {
                await _next(context);
            }
        }
    }
}
```

启动该程序后，利用浏览器访问监听地址依然可以得到图 11-6 所示的输出结果。对于前面介绍的两个中间件，它们的不同之处除了体现在定义和注册方式上，还体现在自身生命周期的差异上。具体来说，强类型方式定义的中间件可以注册为任意生命周期模式的服务，但是按照约定定义的中间件则总是一个 `Singleton` 服务。

11.2 依赖注入

基于 `IHostBuilder/IHost` 的服务承载系统建立在依赖注入框架之上，它在服务承载过程中依赖的服务（包括作为宿主的 `IHost` 对象）都由代表依赖注入容器的 `IServiceProvider` 对象提供。在定义承载服务时，也可以采用依赖注入方式来消费它所依赖的服务。作为依赖注入容器的 `IServiceProvider` 对象能否提供我们需要的服务实例，取决于相应的服务注册是否预先添加到依赖注入框架中。服务注册可以通过调用 `IHostBuilder` 接口或者 `IWebHostBuilder` 接口相应的方法来完成，前者在第 10 章已经有详细介绍，下面介绍基于 `IWebHostBuilder` 接口的服务注册。

11.2.1 服务注册

ASP.NET Core 应用提供了两种服务注册方式，一种是调用 `IWebHostBuilder` 接口的 `ConfigureServices` 方法。如下面的代码片段所示，`IWebHostBuilder` 定义了两个 `ConfigureServices` 方法重载，它们的参数类型分别是 `Action<IServiceCollection>` 和 `Action<WebHostBuilderContext, IServiceCollection>`，我们注册的服务最终会被添加到作为这两个委托对象输入的 `IServiceCollection` 集合中。`WebHostBuilderContext` 代表当前 `IWebHostBuilder` 在构建 `WebHost` 过程中采用的上下文，我们可以利用它得到当前应用的配置和与承载环境相关的信息。

```
public interface IWebHostBuilder
{
    IWebHostBuilder ConfigureServices(Action<IServiceCollection> configureServices);
    IWebHostBuilder ConfigureServices(Action<WebHostBuilderContext, IServiceCollection>
        configureServices);
    ...
}

public class WebHostBuilderContext
{
    public IConfiguration Configuration { get; set; }
    public IWebHostEnvironment HostingEnvironment { get; set; }
}
```

除了直接调用 `IWebHostBuilder` 接口的 `ConfigureServices` 方法注册服务，还可以利用注册的 `Startup` 类型来完成服务的注册。所谓的 `Startup` 类型就是通过调用如下两个扩展方法注册到 `IWebHostBuilder` 接口上用来对应用程序进行初始化的。由于 ASP.NET Core 应用针对请求的处理能力与方式完全取决于注册的中间件，所以这里所谓的针对应用程序的初始化主要体现在针对中间件的注册上。

```
public static class WebHostBuilderExtensions
{
    public static IWebHostBuilder UseStartup<TStartup>(this IWebHostBuilder hostBuilder)
        where TStartup: class;
    public static IWebHostBuilder UseStartup(this IWebHostBuilder hostBuilder,
        Type startupType);
}
```

对于注册的中间件来说，它往往具有针对其他服务的依赖。当 ASP.NET Core 框架在创建具体的中间件对象时，会利用依赖注入框架来提供注入的依赖服务。中间件依赖的这些服务自然需要被预先注册，所以中间件和服务注册成为 `Startup` 对象的两个核心功能。与中间件类型类似，我们在大部分情况下会采用约定的形式来定义 `Startup` 类型。如下所示的代码片段就是一个典型的 `Startup` 的定义，中间件和服务的注册分别实现在 `Configure` 方法和 `ConfigureServices` 方法中。由于并不是在任何情况下都有服务注册的需求，所以 `ConfigureServices` 方法并不是必需的。`Startup` 对象的 `ConfigureServices` 方法的调用发生在整个服务注册的最后阶段，在此之后，ASP.NET Core 应用就会利用所有的服务注册来创建作为依赖注入容器的 `IServiceProvider` 对象。

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services);
    public void Configure(IApplicationBuilder app);
}
```

除了可以采用上述两种方式为应用程序注册所需的服务，ASP.NET Core 框架本身在构建请求处理管道之前也会注册一些服务，这些公共服务除了供框架自身消费，也可以供应用程序使用。那么 ASP.NET Core 框架究竟预先注册了哪些服务？为了得到这个问题的答案，我们编写了如下这段简单的程序。

```
class Program
{
    static void Main()
    {
        Host.CreateDefaultBuilder().ConfigureWebHostDefaults(builder => builder
            .UseStartup<Startup>())
            .Build()
            .Run();
    }
}

public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        var provider = services.BuildServiceProvider();
        foreach (var service in services)
        {
            var serviceName = GetName(service.ServiceType);
            var implementationType = service.ImplementationType
                ?? service.ImplementationInstance?.GetType()
                ?? service.ImplementationFactory?.Invoke(provider)?.GetType();
            if (implementationType != null)
            {
                Console.WriteLine($"{service.Lifetime,-15}
                    {GetName(service.ServiceType),-50}{GetName(implementationType)}");
            }
        }
    }
}
```

```

public void Configure(IApplicationBuilder app) { }

private string GetName(Type type)
{
    if (!type.IsGenericType)
    {
        return type.Name;
    }
    var name = type.Name.Split('.') [0];
    var args = type.GetGenericArguments().Select(it => it.Name);
    return $"{name}<{string.Join(", ", args)}>";
}
}

```

在如上所示的 `Startup` 类型的 `ConfigureServices` 方法中，我们从作为参数的 `ServiceCollection` 对象中获取当前注册的所有服务，并打印每个服务对应的声明类型、实现类型和生命周期。这段程序执行之后，系统注册的所有公共服务会以图 11-7 所示的方式输出到控制台上，我们可以从这个列表中发现很多熟悉的类型。(S1104)

```

C:\Windows\System32\cmd.exe - dotnet run
C:\App>dotnet run
Singleton      IHostingEnvironment
Singleton      IHostingEnvironment
Singleton      HostBuilderContext
Singleton      IConfiguration
Singleton      IApplicationLifetime
Singleton      IApplicationLifetime
Singleton      IHostLifetime
Singleton      IHost
Singleton      IOptions<TOptions>
Scoped         IOptionsSnapshot<TOptions>
Singleton      IOptionsMonitor<TOptions>
Transient      IOptionsFactory<TOptions>
Singleton      IOptionsCache<TOptions>
Singleton      ILoggerFactory
Singleton      ILogger<TCategoryName>
Singleton      IConfigurationOptions<LoggerFilterOptions>
Singleton      IConfigurationOptions<LoggerFilterOptions>
Singleton      ILoggerProviderConfigurationFactory
Singleton      ILoggerProviderConfiguration<T>
Singleton      IConfigurationOptions<LoggerFilterOptions>
Singleton      IOptionsChangeTokenSource<LoggerFilterOptions>
Singleton      LoggingConfiguration
Singleton      ILoggerProvider
Singleton      IConfigurationOptions<ConsoleLoggerOptions>
Singleton      IOptionsChangeTokenSource<ConsoleLoggerOptions>
Singleton      ILoggerProvider
Singleton      LoggingEventSource
Singleton      ILoggerProvider
Singleton      IConfigurationOptions<LoggerFilterOptions>
Singleton      IOptionsChangeTokenSource<LoggerFilterOptions>
Singleton      ILoggerProvider
Singleton      IWebHostEnvironment
Singleton      IHostingEnvironment
Singleton      IApplicationLifetime
Singleton      IConfigurationOptions<GenericWebHostServiceOptions>
Singleton      DiagnosticListener
Singleton      DiagnosticSource
Singleton      IHttpContextFactory
Scoped         IMiddlewareFactory
Singleton      IApplicationBuilderFactory
Singleton      IConnectionListenerFactory
Transient      IConfigurationOptions<KestrelServerOptions>
Singleton      IServer
Singleton      IConfigurationOptions<KestrelServerOptions>
Singleton      IPostConfigureOptions<HostFilteringOptions>
Singleton      IOptionsChangeTokenSource<HostFilteringOptions>
Transient      IStartupFilter
Transient      IInlineConstraintResolver
Transient      ObjectPoolProvider
Singleton      ObjectPool<UriBuildingContext>
Transient      TreeRouteBuilder
Singleton      RoutingMarkerService
Transient      IConfigurationOptions<RouteOptions>
Singleton      EndpointDataSource
Singleton      ParameterPolicyFactory
Singleton      MatcherFactory
HostingEnvironment
HostingEnvironment
HostBuilderContext
ConfigurationRoot
ApplicationLifetime
ApplicationLifetime
ConsoleLifetime
Host
OptionsManager<TOptions>
OptionsManager<TOptions>
OptionsMonitor<TOptions>
OptionsFactory<TOptions>
OptionsCache<TOptions>
LoggerFactory
Logger<T>
DefaultLoggerLevelConfigureOptions
ConfigureNamedOptions<LoggerFilterOptions>
LoggerProviderConfigurationFactory
LoggerProviderConfiguration<T>
LoggerFilterConfigureOptions
ConfigurationChangeTokenSource<LoggerFilterOptions>
LoggingConfiguration
ConsoleLoggerProvider
LoggerProviderConfigureOptions<ConsoleLoggerOptions, ConsoleLoggerProvider>
LoggerProviderOptionsChangeTokenSource<ConsoleLoggerOptions, ConsoleLoggerProvider>
DebugLoggerProvider
LoggingEventSource
EventSourceLoggerProvider
EventLogFiltersConfigureOptions
EventLogFiltersConfigureOptionsChangeSource
EventLogLoggerProvider
HostingEnvironment
HostingEnvironment
GenericWebHostApplicationLifetime
ConfigureNamedOptions<GenericWebHostServiceOptions>
DiagnosticListener
DiagnosticListener
DefaultHttpContextFactory
MiddlewareFactory
ApplicationBuilderFactory
SocketTransportFactory
KestrelServerOptionsSetup
KestrelServer
ConfigureNamedOptions<KestrelServerOptions>
PostConfigureOptions<HostFilteringOptions>
ConfigurationChangeTokenSource<HostFilteringOptions>
HostFilteringStartupFilter
DefaultInlineConstraintResolver
DefaultObjectPoolProvider
DefaultObjectPool<UriBuildingContext>
TreeRouteBuilder
RoutingMarkerService
ConfigureRouteOptions
CompositeEndpointDataSource
DefaultParameterPolicyFactory
DefaultMatcherFactory

```

图 11-7 ASP.NET Core 框架注册的公共服务

11.2.2 服务的消费

ASP.NET Core 框架中的很多核心对象都是通过依赖注入方式提供的, 如用来对应用进行初始化的 `Startup` 对象、中间件对象, 以及 ASP.NET Core MVC 应用中的 `Controller` 对象和 `View` 对象等, 所以我们可以定义它们的时候采用注入的形式来消费已经注册的服务。下面简单介绍几种服务注入的应用场景。

在 `Startup` 中注入服务

构成 `HostBuilderContext` 上下文的两个核心对象 (表示配置的 `IConfiguration` 对象和表示承载环境的 `IHostEnvironment` 对象) 可以直接注入 `Startup` 构造函数中进行消费。由于 ASP.NET Core 应用中的承载环境通过 `IWebHostEnvironment` 接口表示, `IWebHostEnvironment` 接口派生于 `IHostEnvironment` 接口, 所以也可以通过注入 `IWebHostEnvironment` 对象的方式得到当前承载环境相关的信息。

我们可以通过一个简单的实例来验证针对 `Startup` 的构造函数注入。如下面的代码片段所示, 我们在调用 `IWebHostBuilder` 接口的 `Startup<TStartup>` 方法时注册了自定义的 `Startup` 类型。在定义 `Startup` 类型时, 我们在其构造函数中注入上述 3 个对象, 提供的调试断言不仅证明了 3 个对象不为 `Null`, 还表明采用 `IHostEnvironment` 接口和 `IWebHostEnvironment` 接口得到的其实是同一个实例。(S1105)

```
class Program
{
    static void Main()
    {
        Host.CreateDefaultBuilder().ConfigureWebHostDefaults(builder => builder
            .UseStartup<Startup>())
            .Build()
            .Run();
    }
}

public class Startup
{
    public Startup(IConfiguration configuration, IHostEnvironment hostingEnvironment,
        IWebHostEnvironment webHostEnvironment)
    {
        Debug.Assert(configuration != null);
        Debug.Assert(hostingEnvironment != null);
        Debug.Assert(webHostEnvironment != null);
        Debug.Assert(ReferenceEquals(hostingEnvironment, webHostEnvironment));
    }
    public void Configure(IApplicationBuilder app) { }
}
```

依赖服务还可以直接注入用于注册中间件的 `Configure` 方法中。如果构造函数注入还可以对注入的服务有所选择, 那么对于 `Configure` 方法来说, 通过任意方式注册的服务都可以注入其中,

包括通过调用 `IHostBuilder`、`IWebHostBuilder` 和 `Startup` 自身的 `ConfigureServices` 方法注册的服务，还包括框架自行注册的所有服务。

如下面的代码片段所示，我们分别调用 `IWebHostBuilder` 和 `Startup` 的 `ConfigureServices` 方法注册了针对 `IFoo` 接口与 `IBar` 接口的服务，这两个服务直接注入 `Startup` 的 `Configure` 方法中。另外，`Configure` 方法要求提供一个用来注册中间件的 `IApplicationBuilder` 对象作为参数，但是对该参数出现的位置并未做任何限制。(S1106)

```
class Program
{
    static void Main()
    {
        Host.CreateDefaultBuilder().ConfigureWebHostDefaults(builder => builder
            .UseStartup<Startup>()
            .ConfigureServices(svcs => svcs.AddSingleton<IFoo, Foo>()))
            .Build()
            .Run();
    }
}

public class Startup
{
    public void ConfigureServices(IServiceCollection services)
        => services.AddSingleton<IBar, Bar>();
    public void Configure(IApplicationBuilder app, IFoo foo, IBar bar)
    {
        Debug.Assert(foo != null);
        Debug.Assert(bar != null);
    }
}
```

在中间件中注入服务

ASP.NET Core 请求处理管道最重要的对象是真正用来处理请求的中间件。由于 ASP.NET Core 在创建中间件对象并利用它们构建整个请求处理管道时，所有的服务都已经注册完毕，所以注册的任何一个服务都可以注入中间件类型的构造函数中。如下所示的代码片段体现了针对中间件类型的构造函数注入。(S1107)

```
class Program
{
    static void Main()
    {
        Host.CreateDefaultBuilder().ConfigureWebHostDefaults(builder => builder
            .ConfigureServices(svcs => svcs
                .AddSingleton<FoobarMiddleware>()
                .AddSingleton<IFoo, Foo>()
                .AddSingleton<IBar, Bar>())
            .Configure(app => app.UseMiddleware<FoobarMiddleware>()))
            .Build()
            .Run();
    }
}
```

```

    }
}

public class FoobarMiddleware: IMiddleware
{
    public FoobarMiddleware(IFoo foo, IBar bar)
    {
        Debug.Assert(foo != null);
        Debug.Assert(bar != null);
    }

    public Task InvokeAsync(HttpContext context, RequestDelegate next)
    {
        Debug.Assert(next != null);
        return Task.CompletedTask;
    }
}

```

如果采用基于约定的中间件类型定义方式，注册的服务还可以直接注入真正用于处理请求的 `InvokeAsync` 方法或者 `Invoke` 方法中。另外，将方法命名为 `InvokeAsync` 更符合 TAP（Task-based Asynchronous Pattern）编程模式，之所以保留 `Invoke` 方法命名，主要是出于版本兼容的目的。如下所示的代码片段展示了针对 `InvokeAsync` 方法的服务注入。（S1108）

```

class Program
{
    static void Main()
    {
        Host.CreateDefaultBuilder().ConfigureWebHostDefaults(builder => builder
            .ConfigureServices(svcs => svcs
                .AddSingleton<IFoo, Foo>()
                .AddSingleton<IBar, Bar>())
            .Configure(app => app.UseMiddleware<FoobarMiddleware>()))
        .Build()
        .Run();
    }
}

public class FoobarMiddleware
{
    private readonly RequestDelegate next;

    public FoobarMiddleware(RequestDelegate next) => _next = next;
    public Task InvokeAsync(HttpContext context, IFoo foo, IBar bar)
    {
        Debug.Assert(context != null);
        Debug.Assert(foo != null);
        Debug.Assert(bar != null);
        return _next(context);
    }
}

```

虽然约定定义的中间件类型和 `Startup` 类型采用了类似的服务注入方式，它们都支持构造函数注入和方法注入，但是它们之间有一些差别。中间件类型的构造函数、`Startup` 类型的 `Configure` 方法和中间件类型的 `Invoke` 方法或者 `InvokeAsync` 方法都具有一个必需的参数，其类型分别为 `RequestDelegate`、`IApplicationBuilder` 和 `HttpContext`，对于该参数在整个参数列表的位置，前两者都未做任何限制，只有后者要求表示当前请求上下文的参数 `HttpContext` 必须作为方法的第一个参数。按照上述约定，如下这个中间件类型 `FoobarMiddleware` 的定义是不合法的，但是 `Startup` 类型的定义则是合法的。对于这一点，笔者认为可以将这个限制放开，这样不仅可以使中间件类型的定义更加灵活，还能保证注入方式的一致性。

```
public class FoobarMiddleware
{
    public FoobarMiddleware(RequestDelegate next);
    public Task InvokeAsync(IFoo foo, IBar bar, HttpContext context);
}

public class Startup
{
    public void Configure(IFoo foo, IBar bar, IApplicationBuilder app);
}
```

对于基于约定的中间件，构造函数注入与方法注入存在一个本质区别。由于中间件被注册为一个 `Singleton` 对象，所以我们不应该在它的构造函数中注入 `Scoped` 服务。`Scoped` 服务只能注入中间件类型的 `InvokeAsync` 方法中，因为依赖服务是在针对当前请求的服务范围中提供的，所以能够确保 `Scoped` 服务在当前请求处理结束之后被释放。

MVC 应用的依赖注入

在一个 ASP.NET Core MVC 应用中，我们主要在两个地方注入注册的服务：一是在定义的 `Controller` 中以构造函数注入的方式注入所需的服务；二是在呈现的 `View` 中使用 `@inject` 指令实现服务注入。下面通过一个实例演示这两种服务注入方式。下面是 ASP.NET Core MVC 程序涉及的相关代码，我们调用 `IWebHostBuilder` 的 `ConfigureServices` 方法注册了针对 `IFoo` 接口和 `IBar` 接口的两个服务，前者注入 `HomeController` 的构造函数中。

```
class Program
{
    static void Main()
    {
        Host.CreateDefaultBuilder().ConfigureWebHostDefaults(builder => builder
            .ConfigureServices(svcs => svcs
                .AddSingleton<IFoo, Foo>()
                .AddSingleton<IBar, Bar>()
                .AddControllersWithViews())
            .Configure(app => app
                .UseRouting()
                .UseEndpoints(endpoints => endpoints.MapControllers()))
        .Build()
        .Run();
    }
}
```



```

    }
}

public class HomeController : Controller
{
    private readonly IFoo foo;

    public HomeController(IFoo foo) => _foo = foo;

    [HttpGet("/")]
    public IActionResult Index()
    {
        ViewBag.Foo = _foo;
        return View();
    }
}

```

我们为 HomeController 定义了一个路由指向根路径 (“/”) 的 Action 方法 Index，该方法在调用 View 方法呈现默认的 View 之前，将注入的 IFoo 服务以 ViewBag 的形式传递到 View 中。如下所示的代码片段是这个 Action 方法对应 View (“/Views/Home/Index.cshtml”) 的定义，我们通过 @inject 指令注入了 IBar 服务，并将属性名设置为 Bar，这意味着当前 View 对象将添加一个 Bar 属性来引用注入的服务。

```

@inject App.IBar Bar
Foo: @ViewBag.Foo.GetType().AssemblyQualifiedName <br/>
Bar: @Bar.GetType().AssemblyQualifiedName

```

在上面这个 View 中，我们将通过 ViewBag 传递的 IFoo 服务和 IBar 服务的类型名称呈现出来。当程序启动后，如果利用浏览器访问这个 ASP.NET Core MVC 应用的基地址，这个 View 的内容最终将会以图 11-8 所示的形式呈现在浏览器中。(S1109)

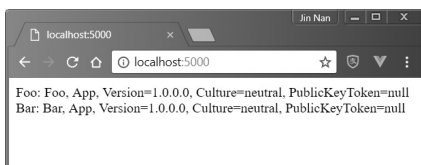


图 11-8 在 Controller 和 View 中注入服务

11.2.3 生命周期

当我们调用 IServiceCollection 相关方法注册服务的时候，总是会指定一种生命周期。由第 3 章和第 4 章的介绍可知，作为依赖注入容器的多个 IServiceProvider 对象通过 ServiceScope 构成一种层次化结构。Singleton 服务实例保存在作为根容器的 IServiceProvider 对象上，而 Scoped 服务实例以及需要回收释放的 Transient 服务实例则保存在当前 IServiceProvider 对象中，只有不需要回收的 Transient 服务才会用完就被丢弃。

至于服务实例是否需要回收释放，取决于服务实现类型是否实现 IDisposable 接口，服务实

例的回收释放由保存它的 `IServiceProvider` 对象负责。具体来说，当 `IServiceProvider` 对象因自身的 `Dispose` 方法被调用而被回收释放时，它会调用自身维护的所有服务实例的 `Dispose` 方法。对于一个非根容器的 `IServiceProvider` 对象来说，其生命周期决定于对应的 `ServiceScope`，调用 `ServiceScope` 的 `Dispose` 方法会导致对封装 `IServiceProvider` 对象的回收释放。

两个 `IServiceProvider` 对象

如果在一个具体的 ASP.NET Core 应用中讨论服务生命周期会更加易于理解：`Singleton` 是针对应用程序的生命周期，而 `Scoped` 是针对请求的生命周期。换句话说，`Singleton` 服务的生命周期会一直延续到应用程序关闭，而 `Scoped` 服务的生命周期仅仅与当前请求上下文绑定在一起，那么这样的生命周期模式是如何实现的？

ASP.NET Core 应用针对服务生命周期管理的实现原理其实也很简单。在应用程序正常启动后，它会利用注册的服务创建一个作为根容器的 `IServiceProvider` 对象，我们可以将它称为 `ApplicationServices`。如果应用在处理某个请求的过程中需要采用依赖注入的方式激活某个服务实例，那么它会利用这个 `IServiceProvider` 对象创建一个代表服务范围的 `ServiceScope` 对象，后者会指定一个 `IServiceProvider` 对象作为子容器，请求处理过程中所需的服务实例均由它来提供，我们可以将它称为 `RequestServices`。

在处理完当前请求后，这个 `ServiceScope` 对象的 `Dispose` 方法会被调用，与它绑定的这个 `IServiceProvider` 对象也随之被回收释放，由它提供的实现了 `IDisposable` 接口的 `Transient` 服务实例也会随之被回收释放，最终由它提供的 `Scoped` 服务实例变成可以被 GC 回收的垃圾对象。表示当前请求上下文的 `HttpContext` 类型具有如下所示的 `RequestServices` 属性，它返回的就是这个针对当前请求的 `IServiceProvider` 对象。

```
public abstract class HttpContext
{
    public abstract IServiceProvider RequestServices { get; set; }
    ...
}
```

为了使读者对注入服务的生命周期有深刻的认识，下面演示一个简单的实例。这是一个 ASP.NET Core MVC 应用，我们在该应用中定义了 3 个服务接口 (`IFoo`、`IBar` 和 `IBaz`) 和对应的实现类 (`Foo`、`Bar` 和 `Baz`)，后者派生于实现了 `IDisposable` 接口的基类 `Base`。我们分别在 `Base` 的构造函数和实现的 `Dispose` 方法中输出相应的文字，以确定服务实例被创建和释放的时间。

```
class Program
{
    static void Main()
    {
        Host.CreateDefaultBuilder().ConfigureWebHostDefaults(builder => builder
            .ConfigureServices(svcs => svcs
                .AddSingleton<IFoo, Foo>()
                .AddScoped<IBar, Bar>()
                .AddTransient<IBaz, Baz>()
                .AddControllersWithViews())
    }
}
```

```

        .Configure(app => app
            .Use(next => httpContext => {
                Console.WriteLine($"Receive request to {httpContext.Request.Path}");
                return next(httpContext);
            })
            .UseRouting()
            .UseEndpoints(endpoints => endpoints.MapControllers()))
        .ConfigureLogging(builder=>builder.ClearProviders())
        .Build()
        .Run();
    }
}

public class HomeController: Controller
{
    private readonly IHostApplicationLifetime _lifetime;

    public HomeController(IHostApplicationLifetime lifetime, IFoo foo,
        IBar bar1, IBar bar2, IBaz baz1, IBaz baz2)
        => lifetime = lifetime;

    [HttpGet("/index")]
    public void Index() {}

    [HttpGet("/stop")]
    public void Stop() => lifetime.StopApplication();
}

public interface IFoo {}
public interface IBar {}
public interface IBaz {}
public class Base : IDisposable
{
    public Base()=> Console.WriteLine($"{this.GetType().Name} is created.");
    public void Dispose() => Console.WriteLine($"{this.GetType().Name} is disposed.");
}
public class Foo : Base, IFoo {}
public class Bar : Base, IBar {}
public class Baz : Base, IBaz {}

```

在注册 ASP.NET Core MVC 框架相关的服务之前，我们采用不同的生命周期对这 3 个服务进行了注册。为了确定应用程序何时开始处理接收的请求，可以利用注册的中间件打印出当前请求的路径。我们在 HomeController 的构造函数中注入了上述 3 个服务和 1 个用来远程关闭应用的 IHostApplicationLifetime 服务，其中 IBar 和 IBaz 被注入了两次。HomeController 包含 Index 和 Stop 两个 Action 方法，它们的路由指向的路径分别为 “/index” 和 “/stop”，Stop 方法利用注入的 IHostApplicationLifetime 服务关闭当前应用。

我们先采用命令行的形式来启动该应用程序，然后利用浏览器依次向该应用发送 3 个请求，

前两个请求指向 Action 方法 Index (“/index”), 后一个指向 Action 方法 Stop (“/stop”), 此时控制台上出现的输出结果如图 11-9 所示。由输出结果可知: 由于 IFoo 服务采用的生命周期模式为 Singleton, 所以在整个应用的生命周期中只会创建一次。对于每个接收的请求, 虽然 IBar 和 IBaz 都被注入了两次, 但是采用 Scoped 模式的 Bar 对象只会被创建一次, 而采用 Transient 模式的 Baz 对象则被创建了两次。再来看释放服务相关的输出, 采用 Singleton 模式的 IFoo 服务会在应用被关闭的时候被释放, 而生命周期模式分别为 Scoped 和 Transient 的 IBar 服务与 IBaz 服务都会在应用处理完当前请求之后被释放。(S1110)



```

Microsoft V... - □ ×
Receive request to /index
Foo is created.
Bar is created.
Baz is created.
Baz is created.
Baz is disposed.
Baz is disposed.
Bar is disposed.
Receive request to /index
Bar is created.
Baz is created.
Baz is created.
Baz is disposed.
Baz is disposed.
Bar is disposed.
Receive request to /stop
Bar is created.
Baz is created.
Baz is created.
Baz is disposed.
Baz is disposed.
Bar is disposed.
Foo is disposed.

```

图 11-9 服务的生命周期

基于服务范围的验证

由第 4 章的介绍可知, Scoped 服务既不应该由作为根容器的 ApplicationServices 来提供, 也不能注入一个 Singleton 服务中, 否则它将无法在请求结束之后释放。如果忽视了这个问题, 就容易造成内存泄漏, 下面是一个典型的例子。

如下所示的实例程序使用了一个名为 FoobarMiddleware 的中间件。在该中间件初始化过程中, 它需要从数据库中加载由 Foobar 类型表示的数据。在这里我们采用 Entity Framework Core 提供的基于 SQL Server 的数据访问, 所以可以为实体类型 Foobar 定义对应的 FoobarDbContext, 它以服务的形式通过调用 IServiceCollection 的 AddDbContext<TDbContext> 扩展方法进行注册, 注册的服务默认采用 Scoped 生命周期。

```

class Program
{
    static void Main()
    {
        Host.CreateDefaultBuilder().ConfigureWebHostDefaults(builder => builder
            .UseDefaultServiceProvider(options=>options.ValidateScopes = false)
            .ConfigureServices(svcs => svcs.AddDbContext<FoobarDbContext>(
                options=>options.UseSqlServer("connection string")))
            .Configure(app =>app.UseMiddleware<FoobarMiddleware>())
            .Build()
    }
}

```

```

        .Run();
    }
}
public class FoobarMiddleware
{
    private readonly RequestDelegate next;
    private readonly Foobar _foobar;
    public FoobarMiddleware(RequestDelegate next, FoobarDbContext dbContext)
    {
        _next = next;
        _foobar = dbContext.Foobar.SingleOrDefault();
    }

    public Task InvokeAsync(HttpContext context)
    {
        ...
        return _next(context);
    }
}

public class Foobar
{
    [Key]
    public string Foo { get; set; }
    public string Bar { get; set; }
}

public class FoobarDbContext : DbContext
{
    public DbSet<Foobar> Foobar { get; set; }
    public FoobarDbContext(DbContextOptions options) : base(options){}
}

```

采用约定方式定义的中间件实际上是一个 `Singleton` 对象，而且它是在应用初始化过程中由根容器的 `IServiceProvider` 对象创建的。由于 `FoobarMiddleware` 的构造函数中注入了 `FoobarDbContext` 对象，所以该对象自然也由同一个 `IServiceProvider` 对象来提供。这就意味着 `FoobarDbContext` 对象的生命周期会延续到当前应用程序被关闭的那一刻，造成的后果就是数据库连接不能及时地被释放。

在一个 ASP.NET Core 应用中，如果将服务的生命周期注册为 `Scoped` 模式，那么我们希望服务实例真正采用基于请求的生命周期模式。由第 4 章的介绍可知，我们可以通过启用针对服务范围的验证来避免采用作为根容器的 `IServiceProvider` 对象来提供 `Scoped` 服务实例。我们只需要调用 `IWebHostBuilder` 接口的两个 `UseDefaultServiceProvider` 方法重载将 `ServiceProviderOptions` 的 `ValidateScopes` 属性设置为 `True` 即可。

```

public static class WebHostBuilderExtensions
{
    public static IWebHostBuilder UseDefaultServiceProvider(
        this IWebHostBuilder hostBuilder, Action<ServiceProviderOptions> configure);
}

```

```

public static IWebHostBuilder UseDefaultServiceProvider(
    this IWebHostBuilder hostBuilder, Action<WebHostBuilderContext,
    ServiceProviderOptions> configure);
}

public class ServiceProviderOptions
{
    public bool ValidateScopes { get; set; }
    public bool ValidateOnBuild { get; set; }
}

```

出于性能方面的考虑，如果在 **Development** 环境下调用 **Host** 的静态方法 **CreateDefaultBuilder** 来创建 **IHostBuilder** 对象，那么该方法会将 **ValidateScopes** 属性设置为 **True**。在上面演示的实例中，我们刻意关闭了针对服务范围的验证，如果将这行代码删除，在开发环境下启动该程序之后会出现图 11-10 所示的异常。

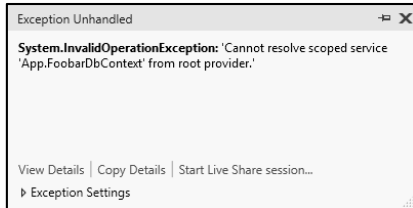


图 11-10 针对 Scoped 服务的验证

如果确实需要在中间件中注入 **Scoped** 服务，可以采用强类型（实现 **IMiddleware** 接口）的中间件定义方式，并将中间件以 **Scoped** 服务进行注册即可。如果采用基于约定的中间件定义方式，我们有两种方案来解决这个问题：第一种解决方案就是按照如下所示的方式在 **InvokeAsync** 方法中利用 **HttpContext** 的 **RequestServices** 属性得到基于当前请求的 **IServiceProvider** 对象，并利用它来提供依赖的服务。

```

public class FoobarMiddleware
{
    private readonly RequestDelegate _next;
    public FoobarMiddleware(RequestDelegate next)=> next = next;
    public Task InvokeAsync(HttpContext context)
    {
        var dbContext = context.RequestServices.GetRequiredService<FoobarDbContext>();
        Debug.Assert(dbContext != null);
        return _next(context);
    }
}

```

第二种解决方案则是按照如下所示的方式直接在 **InvokeAsync** 方法中注入依赖的服务。我们在上面介绍两种中间件定义方式时已经提及：**InvokeAsync** 方法注入的服务就是由基于当前请求的 **IServiceProvider** 对象提供的，所以这两种解决方案其实是等效的。

```

public class FoobarMiddleware
{

```

```

private readonly RequestDelegate next;
public FoobarMiddleware(RequestDelegate next) => _next = next;
public Task InvokeAsync(HttpContext context, FoobarDbContext dbContext)
{
    Debug.Assert(dbContext != null);
    return next(context);
}
}

```

11.2.4 集成第三方依赖注入框架

由第 10 章的介绍可知, 通过调用 `IHostBuilder` 接口的 `UseServiceProviderFactory<TContainerBuilder>` 方法注册 `IServiceProviderFactory<TContainerBuilder>` 工厂的方式可以实现与第三方依赖注入框架的整合。该接口定义的 `ConfigureContainer<TContainerBuilder>` 方法可以对提供的依赖注入容器做进一步设置, 这样的设置同样可以定义在注册的 `Startup` 类型中。

第 3 章创建了一个名为 `Cat` 的简易版依赖注入框架, 并在第 4 章为其创建了一个 `IServiceProviderFactory<TContainerBuilder>` 实现, 具体类型为 `CatServiceProvider`, 下面演示如何通过注册这个 `CatServiceProvider` 实现与第三方依赖注入框架 `Cat` 的整合。如果使用 `Cat` 框架, 我们可以通过在服务类型上标注 `MapToAttribute` 特性的方式来定义服务注册信息。在创建的演示程序中, 我们采用如下方式定义了 3 个服务 (`Foo`、`Bar` 和 `Baz`) 和对应的接口 (`IFoo`、`IBar` 和 `IBaz`)。

```

public interface IFoo { }
public interface IBar { }
public interface IBaz { }

[MapTo(typeof(IFoo), Lifetime.Root)]
public class Foo : IFoo { }

[MapTo(typeof(IBar), Lifetime.Root)]
public class Bar : IBar { }

[MapTo(typeof(IBaz), Lifetime.Root)]
public class Baz : IBaz { }

```

在如下所示的代码片段中, 我们调用 `IHostBuilder` 接口的 `UseServiceProviderFactory` 方法注册了 `CatServiceProviderFactory` 工厂。我们将针对 `Cat` 框架的服务注册实现在注册 `Startup` 类型的 `ConfigureContainer` 方法中, 这是除 `Configure` 方法和 `ConfigureServices` 方法外的第三个约定的方法。我们将 `CatBuilder` 对象作为该方法的参数, 并调用它的 `Register` 方法实现了针对当前程序集的批量服务注册。

```

class Program
{
    static void Main()
    {
        Host.CreateDefaultBuilder()
            .ConfigureWebHostDefaults(builder => builder
                .UseStartup<Startup>())
    }
}

```

```

        .UseServiceProviderFactory(new CatServiceProviderFactory())
        .Build()
        .Run();
    }
}

public class Startup
{
    public void Configure(IApplicationBuilder app, IFoo foo, IBar bar, IBaz baz)
    {
        app.Run(async context =>
        {
            var response = context.Response;
            response.ContentType = "text/html";
            await response.WriteAsync($"foo: {foo}<br/>");
            await response.WriteAsync($"bar: {bar}<br/>");
            await response.WriteAsync($"baz: {baz}<br/>");
        });
    }

    public void ConfigureContainer(CatBuilder container)
        => container.Register(Assembly.GetEntryAssembly());
}

```

为了检验 ASP.NET Core 能否利用 Cat 框架来提供所需的服务，我们将注册的 3 个服务直接注入 Startup 类型的 Configure 方法中。我们在该方法中利用注册的中间件将这 3 个注入的服务实例的类型写入相应的 HTML 文档中。如果利用浏览器访问该应用，得到的输出结果如图 11-11 所示。(S1111)

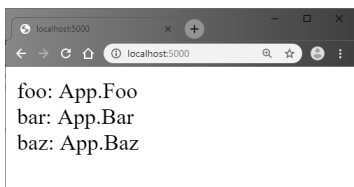


图 11-11 与第三方依赖注入框架的整合

11.3 配置

通过第 10 章的介绍可知，IHostBuilder 接口中定义了 ConfigureHostConfiguration 方法和 ConfigureAppConfiguration 方法，它们可以帮助我们设置面向宿主（IHost 对象）和应用（承载服务）的配置。针对配置的初始化也可以借助 IWebHostBuilder 接口来完成。

11.3.1 初始化配置

当 IWebHostBuilder 对象被创建的时候，它会将当前的环境变量作为配置源来创建承载最初配置数据的 IConfiguration 对象，但它只会选择名称以“ASPNETCORE_”为前缀的环境变量

(通过静态类型 `Host` 的 `CreateDefaultBuilder` 方法创建的 `HostBuilder` 默认选择的是前缀为 “DOTNET_” 的环境变量)。在演示针对环境变量的初始化配置之前，需要先解决配置的消费问题，即如何获取配置数据。

11.2 节演示了针对 `Startup` 类型的构造函数注入，表示配置的 `IConfiguration` 对象是能够注入 `Startup` 类型构造函数中的两个服务对象之一。接下来我们采用 `Options` 模式来消费以环境变量形式提供的配置，如下所示的 `FoobarOptions` 是我们定义的 `Options` 类型。在注册的 `Startup` 类型中，可以直接在构造函数中注入 `IConfiguration` 服务，并在 `ConfigureServices` 方法中将其映射为 `FoobarOptions` 类型。在 `Configure` 方法中，可以通过注入的 `IOptions<FoobarOptions>` 服务得到通过配置绑定的 `FoobarOptions` 对象，并将其序列化成为 JSON 字符串。在通过调用 `IApplicationBuilder` 的 `Run` 方法注册的中间件中，这个 JSON 字符串直接作为请求的响应内容。

```
class Program
{
    static void Main()
    {
        Environment.SetEnvironmentVariable("ASPNETCORE_FOOBAR:FOO", "Foo");
        Environment.SetEnvironmentVariable("ASPNETCORE_FOOBAR:BAR", "Bar");
        Environment.SetEnvironmentVariable("ASPNETCORE_Baz", "Baz");

        Host.CreateDefaultBuilder().ConfigureWebHostDefaults(builder => builder
            .UseStartup<Startup>())
            .Build()
            .Run();
    }
}

public class Startup
{
    private readonly IConfiguration configuration;

    public Startup(IConfiguration configuration)
        => configuration = configuration;

    public void ConfigureServices(IServiceCollection services)
        => services.Configure<FoobarOptions>(configuration);

    public void Configure(IApplicationBuilder app,
        IOptions<FoobarOptions> optionsAccessor)
    {
        var options = optionsAccessor.Value;
        var json = JsonConvert.SerializeObject(
            options, Formatting.Indented);
        app.Run(async context =>
        {
            context.Response.ContentType = "text/html";
            await context.Response.WriteAsync($"<pre>{json}</pre>");
        });
    }
}
```

```

    }

    public class FoobarOptions
    {
        public Foobar Foobar { get; set; }
        public string Baz { get; set; }
    }

    public class Foobar
    {
        public string Foo { get; set; }
        public string Bar { get; set; }
    }
}

```

为了能够提供绑定为 `FoobarOptions` 对象的原始配置，我们在 `Main` 方法中设置了 3 个对应的环境变量，这些环境变量具有相同的前缀“`ASPNETCORE_`”。应用程序启动之后，如果利用浏览器访问该应用，得到的输出结果如图 11-12 所示。(S1112)

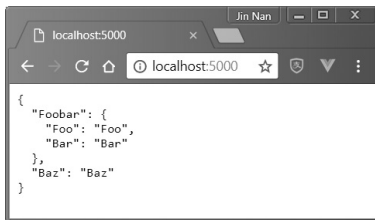


图 11-12 由环境变量提供的原始配置

11.3.2 以键值对形式读取和修改配置

第 6 章对配置模型进行了深入分析，由此可知，`IConfiguration` 对象是以字典的结构来存储配置数据的，该接口定义的索引可供我们以键值对的形式来读取和修改配置数据。在 ASP.NET Core 应用中，我们可以通过调用定义在 `IWebHostBuilder` 接口的 `GetSetting` 方法和 `UseSetting` 方法达到相同的目的。

```

public interface IWebHostBuilder
{
    string GetSetting(string key);
    IWebHostBuilder UseSetting(string key, string value);
    ...
}

```

上面演示的实例采用环境变量来提供最终绑定为 `FoobarOptions` 对象的原始配置，这样的配置数据也可以通过如下所示的调用 `IWebHostBuilder` 接口的 `UseSetting` 方法来提供。修改后的应用程序启动之后，如果利用浏览器访问该应用，同样可以得到图 11-12 所示的输出结果。(S1113)

```

class Program
{
    static void Main()
    {

```

```

Host.CreateDefaultBuilder().ConfigureWebHostDefaults(builder => builder
    .UseSetting("Foobar:Foo", "Foo")
    .UseSetting("Foobar:Bar", "Bar")
    .UseSetting("Baz", "Baz")
    .UseStartup<Startup>())
    .Build()
    .Run();
}
}

```

配置不仅仅供应用程序来使用，ASP.NET Core 框架自身的很多特性也都可以通过配置进行定制。如果希望通过修改配置来控制 ASP.NET Core 框架的某些行为，就需要先知道对应的配置项的名称是什么。例如，ASP.NET Core 应用的服务器默认使用 `launchSettings.json` 文件定义的监听地址，但是我们可以通过修改配置采用其他的监听地址。包括端口在内的监听地址是通过名称为 `urls` 的配置项来控制的，如果记不住这个配置项的名称，也可以直接使用定义在 `WebHostDefaults` 中对应的只读属性 `ServerUrlsKey`，该静态类型中还提供了其他一些预定义的配置项名称，所以这也是一个比较重要的类型。

```

public static class WebHostDefaults
{
    public static readonly string ServerUrlsKey = "urls";
    ...
}

```

针对上面演示的这个实例，如果希望为服务器设置不同的监听地址，直接调用 `IWebHostBuilder` 接口的 `UseSetting` 方法将新的地址作为 `urls` 配置项的内容即可。既然配置项被命名为 `urls`，就意味着服务器的监听地址不仅限于一个，如果希望设置多个监听地址，我们可以采用分号作为分隔符。

```

class Program
{
    static void Main()
    {
        Host.CreateDefaultBuilder().ConfigureWebHostDefaults(builder => builder
            .UseSetting("Foobar:Foo", "Foo")
            .UseSetting("Foobar:Bar", "Bar")
            .UseSetting("Baz", "Baz")
            .UseSetting("urls", "http://0.0.0.0:8888;http://0.0.0.0:9999")
            .UseStartup<Startup>())
            .Build()
            .Run();
    }
}

```

为了使实例程序采用不同的监听地址，可以采用如上所示的方式调用 `IWebHostBuilder` 接口的 `UseSetting` 方法设置两个针对 8888 和 9999 端口号的监听地址。由图 11-13 所示的程序启动后的输出结果可以看出，服务器确实采用我们指定的两个地址监听请求，通过浏览器针对这两个地址发送的请求能够得到相同的结果。(S1114)

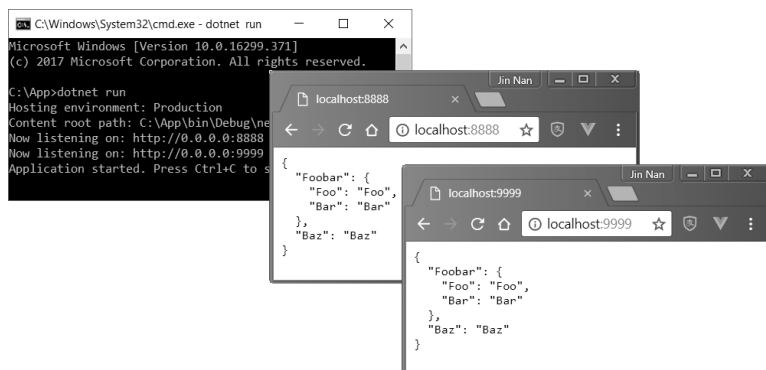


图 11-13 通过配置控制服务器的监听地址

除了调用 `UseSetting` 方法设置 `urls` 配置项来修改服务器的监听地址，直接调用 `IWebHostBuilder` 接口的 `UseUrls` 扩展方法也可以达到相同的目的。另外，我们提供的监听地址只能包含主机名称/IP 地址 (Host/IP) 和端口号，不能包含基础路径 (PathBase)。如果我们提供 “`http://0.0.0.0/3721/foobar`” 这样一个 URL，系统会抛出一个 `InvalidOperationException` 类型的异常。基础路径可以通过注册中间件的方式进行设置，相关内容会在第 24 章进行介绍。

```
public static class HostingAbstractionsWebHostBuilderExtensions
{
    public static IWebHostBuilder UseUrls(this IWebHostBuilder hostBuilder,
        params string[] urls);
}
```

11.3.3 合并配置

在启动一个 ASP.NET Core 应用时，我们可以自行创建一个承载配置的 `IConfiguration` 对象，并通过调用 `IWebHostBuilder` 接口的 `UseConfiguration` 扩展方法将它与应用自身的配置进行合并。如果应用自身存在重复的配置项，那么该配置项的值会被指定的 `IConfiguration` 对象覆盖。

```
public static class HostingAbstractionsWebHostBuilderExtensions
{
    public static IWebHostBuilder UseConfiguration(this IWebHostBuilder hostBuilder,
        IConfiguration configuration);
}
```

如果前面演示的实例需要采用这种方式来提供配置，我们可以对程序代码做如下修改。如下面的代码片段所示，我们创建了一个 `ConfigurationBuilder` 对象，并通过调用 `AddInMemoryCollection` 扩展方法注册了一个 `MemoryConfigurationSource` 对象，它提供了绑定 `FoobarOptions` 对象所需的所有配置数据。我们最终利用 `ConfigurationBuilder` 创建出一个 `IConfiguration` 对象，并通过调用上述 `UseConfiguration` 方法将提供的配置数据合并到当前应用中。修改后的应用程序启动之后，如果利用浏览器访问该应用，同样会得到图 11-12 所示的输出结果。(S1115)

```
class Program
{
    static void Main()
```

```

{
    var configuration = new ConfigurationBuilder()
        .AddInMemoryCollection(new Dictionary<string, string>
            {
                ["Foobar:Foo"]      = "Foo",
                ["Foobar:Bar"]     = "Bar",
                ["Baz"]            = "Baz"
            })
        .Build();

    Host.CreateDefaultBuilder().ConfigureWebHostDefaults(builder => builder
        .UseConfiguration(configuration)
        .UseStartup<Startup>())
        .Build()
        .Run();
}
}

```

11.3.4 注册 IConfigurationSource

配置系统最大的特点是可以注册不同的配置源。借助 `IWebHostBuilder` 接口的 `UseConfiguration` 扩展方法，虽然可以将利用配置系统提供的 `IConfiguration` 对象应用到 ASP.NET Core 程序中，但是这样的整合方式总显得不够彻底，更加理想的方式应该是可以直接在 ASP.NET Core 应用中注册 `IConfigurationSource`。

针对 `IConfigurationSource` 的注册可以调用 `IWebHostBuilder` 接口的 `ConfigureAppConfiguration` 方法来完成，该方法与在 `IHostBuilder` 接口上定义的同名方法基本上是等效的。如下面的代码片段所示，这个方法的参数是一个类型为 `Action<WebHostBuilderContext, IConfigurationBuilder>` 的委托对象，这意味着我们可以就承载上下文对配置做针对性设置。如果设置与当前承载上下文无关，我们还可以调用 `ConfigureAppConfiguration` 方法重载，该方法的参数类型为 `Action<IConfigurationBuilder>`。

```

public interface IWebHostBuilder
{
    IWebHostBuilder ConfigureAppConfiguration(Action<WebHostBuilderContext,
        IConfigurationBuilder> configureDelegate);
}

public static class WebHostBuilderExtensions
{
    public static IWebHostBuilder ConfigureAppConfiguration(
        this IWebHostBuilder hostBuilder, Action<IConfigurationBuilder> configureDelegate);
}

```

对于上面演示的这个程序来说，如果将针对 `IWebHostBuilder` 接口的 `UseConfiguration` 方法的调用替换成如下所示的针对 `ConfigureAppConfiguration` 方法的调用，依然可以达到相同的目的。修改后的应用程序启动之后，如果利用浏览器访问该应用，同样会得到图 11-12 所示的输

出结果。(S1116)

```
class Program
{
    static void Main()
    {
        Host.CreateDefaultBuilder().ConfigureWebHostDefaults(builder => builder
            .ConfigureAppConfiguration(config => config
                .AddInMemoryCollection(new Dictionary<string, string>
                {
                    ["Foobar:Foo"] = "Foo",
                    ["Foobar:Bar"] = "Bar",
                    ["Baz"] = "Baz"
                }
            )))
            .UseStartup<Startup>()
            .Build()
            .Run();
    }
}
```

11.4 承载环境

基于 `IHostBuilder/IHost` 的承载系统通过 `IHostEnvironment` 接口表示承载环境，我们利用它不仅可以得到当前部署环境的名称，还可以获知当前应用的名称和存放内容文件的根目录路径。对于一个 Web 应用来说，我们需要更多的承载环境信息，额外的信息定义在 `IWebHostEnvironment` 接口中。

11.4.1 IWebHostEnvironment

如下面的代码片段所示，派生于 `IHostEnvironment` 接口的 `IWebHostEnvironment` 接口定义了两个属性：`WebRootPath` 和 `WebRootFileProvider`。`WebRootPath` 属性表示用于存放 Web 资源文件根目录的路径，`WebRootFileProvider` 属性则返回该路径对应的 `IFileProvider` 对象。如果我们希望外部可以采用 HTTP 请求的方式直接访问某个静态文件（如 JavaScript、CSS 和图片文件等），只需要将它存放于 `WebRootPath` 属性表示的目录之下即可。

```
public interface IWebHostEnvironment : IHostEnvironment
{
    string WebRootPath { get; set; }
    IFileProvider WebRootFileProvider { get; set; }
}
```

下面简单介绍与承载环境相关的 6 个属性（包含定义在 `IHostEnvironment` 接口中的 4 个属性）是如何设置的。`IHostEnvironment` 接口的 `ApplicationName` 代表当前应用的名称，它的默认值取决于注册的 `IStartup` 服务。`IStartup` 服务旨在完成中间件的注册，不论是调用 `IWebHostBuilder` 接口的 `Configure` 方法，还是调用它的 `UseStartup/UseStartup<TStartup>` 方法，最终都是为了注册 `IStartup` 服务，所以这两个方法是不能被重复调用的。如果多次调用这两个方

法，最后一次调用针对 `IStartup` 的服务注册会覆盖前面的注册。

如果 `IStartup` 服务是通过调用 `IWebHostBuilder` 接口的 `Configure` 方法注册的，那么应用名称由调用该方法提供的 `Action<IApplicationBuilder>` 对象来决定。具体来说，每个委托对象都会绑定到一个方法上，而方法是定义在某个类型中的，该类型所在程序集的名称会默认作为应用名称。如果通过调用 `IWebHostBuilder` 接口的 `UseStartup/UseStartup<TStartup>` 方法来注册 `IStartup` 服务，那么注册的 `Startup` 类型所在的程序集名称就是应用名称。在默认情况下，针对应用名称的设置体现在如下所示的代码片段中。

```
public static IWebHostBuilder Configure(this IWebHostBuilder hostBuilder,
    Action<IApplicationBuilder> configure)
{
    var applicationName = configureApp.GetMethodInfo().DeclaringType
        .GetTypeInfo().Assembly.GetName().Name;
    ...
}

public static IWebHostBuilder UseStartup(this IWebHostBuilder hostBuilder,
    Type startupType)
{
    var applicationName = startupType.GetTypeInfo().Assembly.GetName().Name;
    ...
}
```

`EnvironmentName` 表示当前应用所处部署环境的名称，其中开发（`Development`）、预发（`Staging`）和产品（`Production`）是 3 种典型的部署环境。根据不同的目的可以将同一个应用部署到不同的环境中，在不同环境中部署的应用往往具有不同的设置。在默认情况下，环境的名称为 `Production`。

当我们编译发布一个 `ASP.NET Core` 项目时，项目的源代码文件会被编译成二进制并打包到相应的程序集中，而另外一些文件（如 `JavaScript`、`CSS` 和表示 `View` 的 `.cshtml` 文件等）会复制到目标目录中，我们将这些文件称为内容文件（`Content File`）。`ASP.NET Core` 应用会将所有的内容文件存储在同一个目录下，这个目录的绝对路径通过 `IWebHostEnvironment` 接口的 `ContentRootPath` 属性来表示，而 `ContentRootFileProvider` 属性则返回针对这个目录的 `PhysicalFileProvider` 对象。部分内容文件可以直接作为 `Web` 资源（如 `JavaScript`、`CSS` 和图片等）供客户端以 `HTTP` 请求的方式获取，存放此种类型内容文件的绝对目录通过 `IWebHostEnvironment` 接口的 `WebRootPath` 属性来表示，而针对该目录的 `PhysicalFileProvider` 自然可以通过对应的 `WebRootFileProvider` 属性来获取。

在默认情况下，由 `ContentRootPath` 属性表示的内容文件的根目录就是当前应用程序域的基础目录，也就是表示当前应用程序域的 `AppDomain` 对象的 `BaseDirectory` 属性返回的目录，静态类 `AppContext` 的 `BaseDirectory` 属性返回的也是这个目录。对于一个通过 `Visual Studio` 创建的 `.NET Core` 项目来说，该目录就是编译后保存生成的程序集的目录（如“`\bin\Debug\netcoreapp3.0`”或者“`\bin\Release\netcoreapp3.0`”）。如果该目录下存在一个名为“`wwwroot`”的子目录，那么它将用来

存放 Web 资源，WebRootPath 属性将返回这个目录；如果这样的子目录不存在，那么 WebRootPath 属性会返回 Null。针对这两个目录的默认设置体现在如下所示的代码片段中。

```
class Program
{
    static void Main()
    {
        Host.CreateDefaultBuilder().ConfigureWebHostDefaults(builder => builder
            .UseStartup<Startup>())
            .Build()
            .Run();
    }
}
public class Startup
{
    public Startup(IWebHostEnvironment environment)
    {
        Debug.Assert(environment.ContentRootPath == AppDomain.CurrentDomain.BaseDirectory);
        Debug.Assert(environment.ContentRootPath == AppContext.BaseDirectory);

        var wwwRoot = Path.Combine(AppContext.BaseDirectory, "wwwroot");
        if (Directory.Exists(wwwRoot))
        {
            Debug.Assert(environment.WebRootPath == wwwRoot);
        }
        else
        {
            Debug.Assert(environment.WebRootPath == null);
        }
    }
    public void Configure(IApplicationBuilder app) {}
}
```

11.4.2 通过配置定制承载环境

IWebHostEnvironment 对象承载的 4 个与承载环境相关的属性 (ApplicationName、EnvironmentName、ContentRootPath 和 WebRootPath) 可以通过配置的方式进行定制，对应配置项的名称分别为 applicationName、environment、contentRoot 和 webroot。如果记不住这些配置项的名称也没有关系，因为我们可以利用定义在静态类 WebHostDefaults 中如下所示的 4 个只读属性来得到它们的值。通过第 11 章的介绍可知，前三个配置项的名称同样以静态只读字段的形式定义在 HostDefaults 类型中。

```
public static class WebHostDefaults
{
    public static readonly string EnvironmentKey = "environment";
    public static readonly string ContentRootKey = "contentRoot";
    public static readonly string ApplicationKey = "applicationName";
    public static readonly string WebRootKey = "webroot";
}
```



```
public static class HostDefaults
{
    public static readonly string EnvironmentKey = "environment";
    public static readonly string ContentRootKey = "contentRoot";
    public static readonly string ApplicationKey = "applicationName";
}
```

下面演示如何通过配置的方式来设置当前的承载环境。在如下这段实例程序中，我们调用 `IWebHostBuilder` 接口的 `UseSetting` 方法针对上述 4 个配置项做了相应的设置。由于针对 `UseStartup<TStartup>` 方法的调用会设置应用的名称，所以通过调用 `UseSetting` 方法针对应用名称的设置需要放在后面才有意义。相对于当前目录（项目根目录）的两个子目录“contents”和“contents/web”是我们为 `ContentRootPath` 属性与 `WebRootPath` 属性设置的，由于系统会验证设置的目录是否存在，所以必须预先创建这两个目录。

```
class Program
{
    static void Main()
    {
        Host.CreateDefaultBuilder().ConfigureWebHostDefaults(builder => builder
            .ConfigureLogging(options => options.ClearProviders())
            .UseStartup<Startup>()
            .UseSetting("environment", "Staging")
            .UseSetting("contentRoot",
                Path.Combine(Directory.GetCurrentDirectory(), "contents"))
            .UseSetting("webroot",
                Path.Combine(Directory.GetCurrentDirectory(), "contents/web"))
            .UseSetting("ApplicationName", "MyApp"))
        .Build()
        .Run();
    }

    public class Startup
    {
        public Startup(IWebHostEnvironment environment)
        {
            Console.WriteLine($"ApplicationName: {environment.ApplicationName}");
            Console.WriteLine($"EnvironmentName: {environment.EnvironmentName}");
            Console.WriteLine($"ContentRootPath: {environment.ContentRootPath}");
            Console.WriteLine($"WebRootPath: {environment.WebRootPath}");
        }

        public void Configure(IApplicationBuilder app) { }
    }
}
```

我们在注册的 `Startup` 类型的构造函数中注入了 `IWebHostEnvironment` 服务，并直接将这 4 个属性输出到控制台上。我们在目录“C:\App”下运行这个程序后，设置的 4 个与承载相关的属性会以图 11-14 所示的形式呈现在控制台上。(S1117)

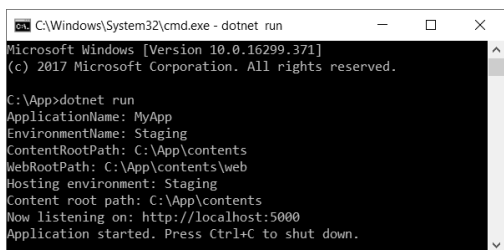


图 11-14 利用配置定义承载环境

由于 `IWebHostEnvironment` 服务提供的应用名称会被视为一个程序集名称，针对它的设置会影响类型的加载，所以我们基本上不会设置应用的名称。至于其他 3 个属性，除了采用最原始的方式设置相应的配置项，我们还可以直接调用 `IWebHostBuilder` 接口中如下 3 个对应的扩展方法来设置。通过第 10 章的介绍可知，`IHostBuilder` 接口也有类似的扩展方法。

```

public static class HostingAbstractionsWebHostBuilderExtensions
{
    public static IWebHostBuilder UseEnvironment(this IWebHostBuilder hostBuilder,
        string environment);
    public static IWebHostBuilder UseContentRoot(this IWebHostBuilder hostBuilder,
        string contentRoot);
    public static IWebHostBuilder UseWebRoot(this IWebHostBuilder hostBuilder,
        string webRoot);
}

public static class HostingHostBuilderExtensions
{
    public static IHostBuilder UseContentRoot(this IHostBuilder hostBuilder,
        string contentRoot);
    public static IHostBuilder UseEnvironment(this IHostBuilder hostBuilder,
        string environment);
}

```

11.4.3 针对环境的编程

对于同一个 ASP.NET Core 应用来说，我们添加的服务注册、提供的配置和注册的中间件可能会因部署环境的不同而有所差异。有了这个可以随意注入的 `IWebHostEnvironment` 服务，我们可以很方便地知道当前的部署环境并进行有针对性的差异化编程。

`IHostEnvironment` 接口提供了如下这个名为 `IsEnvironment` 的扩展方法，用于确定当前是否为指定的部署环境。除此之外，`IHostEnvironment` 接口还提供额外 3 个扩展方法来进行针对 3 种典型部署环境（开发、预发和产品）的判断，这 3 种环境采用的名称分别为 `Development`、`Staging` 和 `Production`，对应静态类型 `EnvironmentName` 的 3 个只读字段。

```

public static class HostEnvironmentEnvExtensions
{
    public static bool IsDevelopment(this IHostEnvironment hostEnvironment);
    public static bool IsProduction(this IHostEnvironment hostEnvironment);
}

```

```

    public static bool IsStaging(this IHostEnvironment hostEnvironment);
    public static bool IsEnvironment(this IHostEnvironment hostEnvironment,
        string environmentName);
}

public static class EnvironmentName
{
    public static readonly string Development      = "Development";
    public static readonly string Staging         = "Staging";
    public static readonly string Production      = "Production";
}

```

注册服务

下面先介绍针对环境的服务注册。ASP.NET Core 应用提供了两种服务注册方式：第一种是调用 `IWebHostBuilder` 接口的 `ConfigureServices` 方法；第二种是调用 `UseStartup` 方法或者 `UseStartup<TStartup>` 方法注册一个 `Startup` 类型，并在其 `ConfigureServices` 方法中完成服务注册。对于第一种服务注册方式，用于注册服务的 `ConfigureServices` 方法具有一个参数类型为 `Action<WebHostBuilderContext, IServiceCollection>` 的重载，所以我们可以利用提供的 `WebHostBuilderContext` 对象以如下所示的方式针对具体的环境注册相应的服务。

```

class Program
{
    public static void Main()
    {
        Host.CreateDefaultBuilder().ConfigureWebHostDefaults(builder => builder
            .ConfigureServices((context, svcs) => {
                if (context.HostingEnvironment.IsDevelopment())
                {
                    svcs.AddSingleton<IFoobar, Foo>();
                }
                else
                {
                    svcs.AddSingleton<IFoobar, Bar>();
                }
            })
            .Build()
            .Run();
    }
}

```

如果利用 `Startup` 类型来添加服务注册，我们就可以按照如下所示的方式通过构造函数注入的方式得到所需的 `IWebHostEnvironment` 服务，并在 `ConfigureServices` 方法中根据它提供的环境信息来注册对应的服务。另外，`Startup` 类型的 `ConfigureServices` 方法要么是无参的，要么具有一个类型为 `IServiceCollection` 的参数，所以我们无法直接在这个方法中注入 `IWebHostEnvironment` 服务。

```

public class Startup
{

```

```

private readonly IWebHostEnvironment environment;

public Startup(IWebHostEnvironment environment) => environment = environment;
public void ConfigureServices(IServiceCollection svcs)
{
    if (environment.IsDevelopment())
    {
        svcs.AddSingleton<IFoobar, Foo>();
    }
    else
    {
        svcs.AddSingleton<IFoobar, Bar>();
    }
}
public void Configure(IApplicationBuilder app) { }
}

```

除了在注册 `Startup` 类型中的 `ConfigureServices` 方法完成针对承载环境的服务注册，我们还可以将针对某种环境的服务注册实现在对应的 `Configure{EnvironmentName}Services` 方法中。上面定义的 `Startup` 类型完全可以改写成如下形式。

```

public class Startup
{
    public void ConfigureDevelopmentServices(IServiceCollection svcs)
        => svcs.AddSingleton<IFoobar, Foo>();
    public void ConfigureServices(IServiceCollection svcs)
        => svcs.AddSingleton<IFoobar, Bar>();
    public void Configure(IApplicationBuilder app) {}
}

```

注册中间件

与服务注册类似，中间件的注册同样具有两种方式：一种是直接调用 `IWebHostBuilder` 接口的 `Configure` 方法；另一种则是调用注册的 `Startup` 类型的同名方法。不管采用何种方式，中间件都是借助 `IApplicationBuilder` 对象来注册的。由于针对应用程序的 `IServiceProvider` 对象可以通过其 `ApplicationServices` 属性获得，所以我们可以利用它提供承载环境信息的 `IWebHostEnvironment` 服务，进而按照如下所示的方式实现针对环境的中间件注册。

```

class Program
{
    public static void Main()
    {
        Host.CreateDefaultBuilder().ConfigureWebHostDefaults(builder => builder
            .Configure(app=> {
                var environment = app.ApplicationServices
                    .GetRequiredService<IWebHostEnvironment>();
                if (environment.IsDevelopment())
                {
                    app.UseMiddleware<FooMiddleware>();
                }
            })
        app
    }
}

```

```

        .UseMiddleware<BarMiddleware>()
        .UseMiddleware<BazMiddleware>());
    })
    .Build()
    .Run();
}
}

```

其实，用于注册中间件的 `IApplicationBuilder` 接口还有 `UseWhen` 的扩展方法。顾名思义，这个方法可以帮助我们根据指定的条件来注册对应的中间件。注册中间件的前提条件可以通过一个 `Func<HttpContext, bool>` 对象来表示，对于某个具体的请求来说，只有对应的 `HttpContext` 对象满足该对象设置的断言，指定的中间件注册操作才会生效。

```

public static class UseWhenExtensions
{
    public static IApplicationBuilder UseWhen(this IApplicationBuilder app,
        Func<HttpContext, bool> predicate, Action<IApplicationBuilder> configuration);
}

```

如果调用 `UseWhen` 方法来实现针对具体环境注册对应的中间件，我们就可以按照如下所示的方式利用 `HttpContext` 来提供针对当前请求的 `IServiceProvider` 对象，进而得到承载环境信息的 `IWebHostEnvironment` 服务，最终根据提供的环境信息进行有针对性的服务注册。

```

class Program
{
    public static void Main()
    {
        Host.CreateDefaultBuilder().ConfigureWebHostDefaults(builder => builder
            .Configure(app=> app
                .UseWhen(context=>context.RequestServices
                    .GetRequiredService<IWebHostEnvironment>().IsDevelopment(),
                    builder => builder.UseMiddleware<FooMiddleware>())
                .UseMiddleware<BarMiddleware>()
                .UseMiddleware<BazMiddleware>()))
            .Build()
            .Run();
    }
}

```

如果应用注册了 `Startup` 类型，那么针对环境的中间件注册就更加简单，因为用来注册中间件的 `Configure` 方法自身是可以注入任意依赖服务的，所以我们可以该方法中按照如下所示的方式直接注入 `IWebHostEnvironment` 服务来提供环境信息。

```

public class Startup
{
    public void Configure(IApplicationBuilder app, IWebHostEnvironment environment)
    {
        if (environment.IsDevelopment())
        {
            app.UseMiddleware<FooMiddleware>();
        }
        app
    }
}

```

```

        .UseMiddleware<BarMiddleware>()
        .UseMiddleware<BazMiddleware>();
    }
}

```

与服务注册类似，针对环境的中间件注册同样可以定义在对应的 `Configure{EnvironmentName}` 方法中，上面这个 `Startup` 类型完全可以改写成如下形式。

```

public class Startup
{
    public void ConfigureDevelopment (IApplicationBuilder app)
    {
        app.UseMiddleware<FooMiddleware>();
    }

    public void Configure(IApplicationBuilder app)
    {
        app
            .UseMiddleware<BarMiddleware>()
            .UseMiddleware<BazMiddleware>();
    }
}

```

配置

上面介绍了针对环境的服务和中间件注册，下面介绍如何根据当前的环境来提供有针对性的配置。通过前面的介绍可知，`IWebHostBuilder` 接口提供了一个名为 `ConfigureAppConfiguration` 的方法，我们可以调用这个方法注册相应的 `IConfigureSource` 对象。这个方法具有一个类型为 `Action<WebHostBuilderContext, IConfigurationBuilder>` 的参数，所以可以通过提供的这个 `WebHostBuilderContext` 上下文得到提供环境信息的 `IWebHostEnvironment` 对象。

如果采用配置文件，我们可以将配置内容分配到多个文件中。例如，我们可以将与环境无关的配置定义在 `Appsettings.json` 文件中，然后针对具体环境提供对应的配置文件 `Appsettings.{EnvironmentName}.json`（如 `Appsettings.Development.json`、`Appsettings.Staging.json` 和 `Appsettings.Production.json`）。最终我们可以按照如下所示的方式将针对这两类配置文件的 `IConfigureSource` 注册到提供的 `IConfigurationBuilder` 对象上。

```

class Program
{
    public static void Main()
    {
        Host.CreateDefaultBuilder().ConfigureWebHostDefaults(builder => builder
            .ConfigureAppConfiguration((context, builder) => builder
                .AddJsonFile(path: "AppSettings.json", optional: false)
                .AddJsonFile(
                    path: $"AppSettings.{context.HostingEnvironment.EnvironmentName}.json",
                    optional: true)))
            .UseStartup<Startup>()
            .Build()
            .Run();
    }
}

```

```

    }
}

```

11.5 初始化

一个 ASP.NET Core 应用的核心就是由一个服务器和一组有序中间件组成的请求处理管道，服务器只负责监听、接收和分发请求，以及最终完成对请求的响应，所以一个 ASP.NET Core 应用针对请求的处理能力和处理方式由注册的中间件来决定。一个 ASP.NET Core 在启动过程中的核心工作就是注册中间件，本节主要介绍应用启动过程中以中间件注册为核心的初始化工作。

11.5.1 Startup

由于 ASP.NET Core 应用承载于以 IHost/IHostBuilder 为核心的承载系统中，所以在启动过程中需要的所有操作都可以直接调用 IHostBuilder 接口相应的方法来完成，但是我们倾向于将这些代码单独定义在按照约定定义的 Startup 类型中。由于注册 Startup 的核心目的是注册中间件，所以 Configure 方法是必需的，用于注册服务的 ConfigureServices 方法和用来设置第三方依赖注入容器的 ConfigureContainer 方法是可选的。如下所示的代码片段体现了典型的 Startup 类型定义方式。

```

public class Startup
{
    public void Configure(IApplicationBuilder app);
    public void ConfigureServices(IServiceCollection services);
    public void ConfigureContainer(FoobarContainerBuilder container);
}

```

除了显式调用 IWebHostBuilder 接口的 UseStartup 方法或者 UseStartup<TStartup>方法注册一个 Startup 类型，如果另外一个程序集中定义了合法的 Startup 类型，我们可以通过配置将它作为启动程序集。作为启动程序集的配置项目的名称为 startupAssembly，对应静态类型 WebHostDefaults 的只读字段 StartupAssemblyKey。

```

public static class WebHostDefaults
{
    public static readonly string StartupAssemblyKey;
    ...
}

```

一旦启动程序集通过配置的形式确定下来，系统就会试着从该程序集中找到一个具有最优匹配度的 Startup 类型。下面列举了一系列 Startup 类型的有效名称，Startup 类型加载器正是按照这个顺序从启动程序集类型列表中进行筛选的，如果最终没有任何一个类型满足条件，那么系统会抛出一个 InvalidOperationException 异常。

- Startup{EnvironmentName}（全名匹配）。
- Startup（全名匹配）。
- {StartupAssembly}.Startup{EnvironmentName}（全名匹配）。
- {StartupAssembly}.Startup（全名匹配）。

- Startup{EnvironmentName} (任意命名空间)。
- Startup (任意命名空间)。

由此可以看出，当 ASP.NET Core 框架从启动程序集中定位 Startup 类型时会优先选择类型名称与当前环境名称相匹配的。为了使读者对这个选择策略有更加深刻的认识，下面做一个实例演示。我们利用 Visual Studio 创建一个名为 App 的控制台应用，并编写了如下这段简单的程序。在如下所示的代码片段中，我们将当前命令行参数作为配置源。我们既没有调用 IWebHostBuilder 接口的 Configure 方法注册任何中间件，也没有调用 UseStartup 方法或者 UseStartup<TStartup>方法注册 Startup 类型。

```
class Program
{
    static void Main(string[] args)
    {
        Host.CreateDefaultBuilder(args).ConfigureWebHostDefaults(builder => builder
            .ConfigureLogging(options => options.ClearProviders()))
            .Build()
            .Run();
    }
}
```

我们创建了另一个名为 AppStartup 的类库项目，并在其中定义了如下 3 个继承自抽象类 StartupBase 的类型。根据命名约定，StartupDevelopment 类型和 StartupStaging 类型分别针对 Development 环境与 Staging 环境，而 Startup 类型则不针对某个具体的环境（环境中性）。

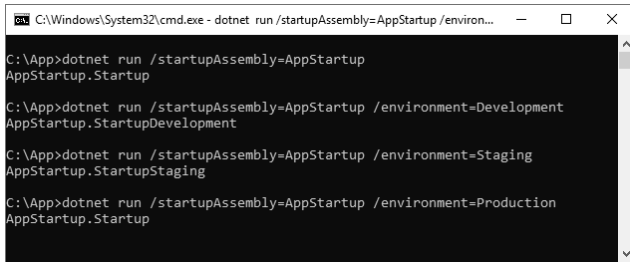
```
namespace AppStartup
{
    public abstract class StartupBase
    {
        public StartupBase() => Console.WriteLine(this.GetType().FullName);
        public void Configure(IApplicationBuilder app) { }
    }

    public class StartupDevelopment : StartupBase { }
    public class StartupStaging : StartupBase { }
    public class Startup : StartupBase { }
}
```

由于基类 StartupBase 的构造函数会将自身类型的全名输出到控制台上，所以可以根据这个输出确定哪个 Startup 类型会被选用。我们采用命令行的形式多次启动 App 应用，并以命令行参数的形式指定启动程序集名称和当前环境名称，控制台上呈现的输出结果如图 11-15 所示。

(S1118)

如图 11-15 所示，如果没有显式指定环境名称，当前应用就会采用默认的 Production 环境名称，所以不针对具体环境的 AppStartup.Startup 被选择作为 Startup 类型。当我们将环境名称分别显式设置为 Development 和 Staging 之后，被选择作为 Startup 类型的分别为 StartupDevelopment 和 StartupStaging。



```

C:\Windows\System32\cmd.exe - dotnet run /startupAssembly=AppStartup /environ...
C:\App>dotnet run /startupAssembly=AppStartup
AppStartup.Startup
C:\App>dotnet run /startupAssembly=AppStartup /environment=Development
AppStartup.StartupDevelopment
C:\App>dotnet run /startupAssembly=AppStartup /environment=Staging
AppStartup.StartupStaging
C:\App>dotnet run /startupAssembly=AppStartup /environment=Production
AppStartup.Startup

```

图 11-15 利用额外程序集来提供 Startup 类型

与具体承载环境进行关联除了可以体现在 Startup 类型的命名 (Startup{EnvironmentName}) 上, 还可以体现在对方法的命名 (Configure{EnvironmentName}、Configure{EnvironmentName}Services 和 Configure{EnvironmentName}Container) 上。如下所示的这个 Startup 类型针对开发环境、预发环境和产品环境定义了对应的方法, 如果还有其他的环境, 不具有环境名称的 3 个方法将会被使用, 在上面介绍服务注册和中间件注册时已经有明确的说明。

```

public class Startup
{
    public void Configure(IApplicationBuilder app);
    public void ConfigureServices(IServiceCollection services);
    public void ConfigureContainer(FoobarContainerBuilder container);

    public void ConfigureDevelopment(IApplicationBuilder app);
    public void ConfigureDevelopmentServices(IServiceCollection services);
    public void ConfigureDevelopmentContainer(FoobarContainerBuilder container);

    public void ConfigureStaging(IApplicationBuilder app);
    public void ConfigureStagingServices(IServiceCollection services);
    public void ConfigureStagingContainer(FoobarContainerBuilder container);

    public void ConfigureProduction(IApplicationBuilder app);
    public void ConfigureProductionServices(IServiceCollection services);
    public void ConfigureProductionContainer(FoobarContainerBuilder container);
}

```

11.5.2 IHostingStartup

除了通过注册 Startup 类型来初始化应用程序, 我们还可以通过注册一个或者多个 IHostingStartup 服务达到类似的目的。由于 IHostingStartup 服务可以通过第三程序集来提供, 如果第三方框架、类库或者工具需要在应用启动时做相应的初始化工作, 就可以将这些工作实现在注册的 IHostingStart 服务中。如下所示的代码片段是服务接口 IHostingStartup 的定义, 它只定义了一个唯一的 Configure 方法, 该方法可以利用输入参数得到当前使用的 IWebHostBuilder 对象。

```

public interface IHostingStartup
{
    void Configure(IWebHostBuilder builder);
}

```

IHostingStartup 服务是通过如下所示的 HostingStartupAttribute 特性来注册的。从给出的定义可以看出这是一个针对程序集的特性，在构造函数中指定的就是注册的 IHostingStartup 类型。由于在同一个程序集中可以多次使用该特性 (AllowMultiple=true)，所以同一个程序集可以提供多个 IHostingStartup 服务类型。

```
[AttributeUsage((AttributeTargets) AttributeTargets.Assembly, Inherited=false,
    AllowMultiple=true)]
public sealed class HostingStartupAttribute : Attribute
{
    public Type HostingStartupType { get; }
    public HostingStartupAttribute(Type hostingStartupType);
}
```

如果希望某个程序集提供的 IHostingStartup 服务类型能够真正应用到当前程序中，我们需要采用配置的形式对程序集进行注册。注册 IHostingStartup 程序集的配置项名称为 hostingStartupAssemblies，对应静态类型 WebHostDefaults 的只读字段 HostingStartupAssembliesKey。通过配置形式注册的程序集名称以分号进行分隔。当前应用名称会作为默认的 IHostingStartup 程序集进行注册，如果针对 IHostingStartup 类型的注册定义在该程序集中，就不需要对该程序集进行显式配置。

```
public static class WebHostDefaults
{
    public static readonly string HostingStartupAssembliesKey;
    public static readonly string PreventHostingStartupKey;
    public static readonly string HostingStartupExcludeAssembliesKey;
}
```

这一特性还有一个全局开关。如果不希望第三方案程序集对当前应用程序进行干预，我们可以通过配置项 preventHostingStartup 关闭这一特性，该配置项的名称对应 WebHostDefaults 的 PreventHostingStartupKey 属性。另外，对于布尔值类型的配置项，“true”（不区分大小写）和“1”都表示 True，其他值则表示 False。WebHostDefaults 还通过 HostingStartupExcludeAssembliesKey 属性定义了另一个配置项，其名称为 hostingStartupExcludeAssemblies，用于设置需要被排除的程序集列表。

下面通过对前面的程序略加修改来演示针对 IHostingStartup 服务的初始化。首先在 App 项目中定义了如下这个实现了 IHostingStartup 接口的类型 Foo，它实现的 Configure 方法会在控制台上打印出相应的文字以确定该方法是否被调用。这个自定义的 IHostingStartup 服务类型通过 HostingStartupAttribute 特性进行注册。IHostingStartup 相关的配置只有通过环境变量和调用 IWebHostBuilder 接口的 UseSetting 方法进行设置才有效，所以虽然我们采用命令行参数提供原始配置，但是必须调用 UseSetting 方法将它们应用到 IWebHostBuilder 对象上。

```
[assembly: HostingStartup(typeof(Foo))]

class Program
{
    static void Main(string[] args)
    {
        var config = new ConfigurationBuilder()
```

```

        .AddCommandLine(args)
        .Build();

    Host.CreateDefaultBuilder().ConfigureWebHostDefaults(builder => builder
        .ConfigureLogging(options => options.ClearProviders())
        .UseSetting("hostingStartupAssemblies", config["hostingStartupAssemblies"])
        .UseSetting("preventHostingStartup", config["preventHostingStartup"])
        .Configure(app => app.Run(context => Task.CompletedTask)))
        .Build()
        .Run();
    }
}

public class Foo : IHostingStartup
{
    public void Configure(IWebHostBuilder builder) => Console.WriteLine("Foo.Configure()");
}

```

另一个 `AppStartup` 项目包含如下两个自定义的 `IHostingStartup` 服务类型 `Bar` 和 `Baz`，我们采用如下方式利用 `HostingStartupAttribute` 特性对它们进行了注册。

```

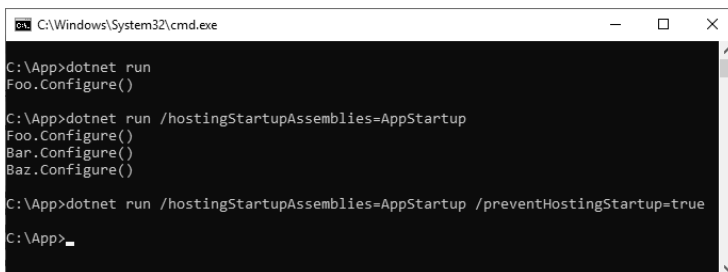
[assembly: HostingStartup(typeof(Bar))]
[assembly: HostingStartup(typeof(Baz))]

public abstract class HostingStartupBarBase : IHostingStartup
{
    public void Configure(IWebHostBuilder builder)
        => Console.WriteLine($"{GetType().Name}.Configure()");
}

public class Bar : HostingStartupBarBase {}
public class Baz : HostingStartupBarBase {}

```

我们采用命令行以图 11-16 所示的形式两次启动 `App` 应用。对于第一次应用启动，由于对启动程序集 `AppStartup` 进行了显式设置，由它提供的两个 `IHostingStartup` 服务（`Bar` 和 `Baz`）都得以正常执行。而注册的 `IHostingStartup` 服务 `Foo`，由于被注册到当前应用程序对应的程序集，虽然我们没有将它显式地添加到启动程序集列表中，但它依然会执行，而且是在其他程序集注册的 `IHostingStartup` 服务之前执行。至于第二次应用启动，由于我们通过命令行参数关闭了针对 `IHostingStartup` 服务的初始化功能，所以 `Foo`、`Bar` 和 `Baz` 这 3 个自定义 `IHostingStartup` 服务都不会执行。（S1119）



```

C:\Windows\System32\cmd.exe

C:\App>dotnet run
Foo.Configure()

C:\App>dotnet run /hostingStartupAssemblies=AppStartup
Foo.Configure()
Bar.Configure()
Baz.Configure()

C:\App>dotnet run /hostingStartupAssemblies=AppStartup /preventHostingStartup=true
C:\App>_

```

图 11-16 利用注册的 `IHostingStartup` 服务对应用程序进行初始化

11.5.3 IStartupFilter

中间件的注册离不开 `IApplicationBuilder` 对象，注册的 `IStartup` 服务的 `Configure` 方法会利用该对象帮助我们完成中间件的构建与注册。调用 `IWebHostBuilder` 接口的 `Configure` 方法时，系统会注册一个类型为 `DelegateStartup` 的 `IStartup` 服务，`DelegateStartup` 会利用提供的 `Action<IApplicationBuilder>` 对象完成中间件的构建与注册。

如果调用 `IWebHostBuilder` 接口的 `UseStartup` 方法或者 `UseStartup<Startup>` 方法注册了一个 `Startup` 类型并且该类型没有实现 `IStartup` 接口，系统就会按照约定规则创建一个类型为 `ConventionBasedStartup` 的 `IStartup` 服务。如果注册的 `Startup` 类型实现了 `IStartup` 接口，意味着注册的就是 `IStartup` 服务。

除了采用上述两种方式利用系统提供的 `IStartup` 服务来注册中间件，我们还可以通过注册 `IStartupFilter` 服务来达到相同的目的。一个应用程序可以注册多个 `IStartupFilter` 服务，它们会按照注册的顺序组成一个链表。`IStartupFilter` 接口具有如下所示的唯一方法 `Configure`，中间件的注册体现在它返回的 `Action<IApplicationBuilder>` 对象上，而作为唯一参数的 `Action<IApplicationBuilder>` 对象则代表了针对后续中间件的注册。

```
public interface IStartupFilter
{
    Action<IApplicationBuilder> Configure(Action<IApplicationBuilder> next);
}
```

虽然注册中间件是 `IStartup` 对象和 `IStartupFilter` 对象的核心功能，但是两者之间还是不尽相同的，它们之间的差异在于：`IStartupFilter` 对象的 `Configure` 方法会在 `IStartup` 对象的 `Configure` 方法之前执行。正因为如此，如果需要将注册的中间件前置或者后置，就需要利用 `IStartupFilter` 对象来注册它们。

接下来我们同样会演示一个针对 `IStartupFilter` 的中间件注册的实例。首先定义如下两个中间件类型 `FooMiddleware` 和 `BarMiddleware`，它们派生于同一个基类 `StringContentMiddleware`。当 `InvokeAsync` 方法被执行时，中间件在将请求分发给后续中间件之前和之后会分别将一段预先指定的文字写入响应消息的主体内容中，它们代表了中间件针对请求的前置和后置处理。

```
public abstract class StringContentMiddleware
{
    private readonly RequestDelegate _next;
    private readonly string preContents;
    private readonly string postContents;

    public StringContentMiddleware(RequestDelegate next, string preContents,
        string postContents)
    {
        next = next;
        _preContents = preContents;
        _postContents = postContents;
    }
}
```

```

public async Task InvokeAsync(HttpContext context)
{
    await context.Response.WriteAsync( preContents);
    await next(context);
    await context.Response.WriteAsync( _postContents);
}
}

public class FooMiddleware : StringContentMiddleware
{
    public FooMiddleware (RequestDelegate next) : base(next, "Foo=>", "Foo") { }
}

public class BarMiddleware : StringContentMiddleware
{
    public BarMiddleware (RequestDelegate next) : base(next, "Bar=>", "Bar=>") { }
}

```

可以采用如下方式对 `FooMiddleware` 和 `BarMiddleware` 这两个中间件进行注册。具体来说,我们为中间件类型 `FooMiddleware` 创建了一个自定义的 `IStartupFilter` 类型 `FooStartupFilter`, `FooStartupFilter` 实现的 `Configure` 方法中注册了这个中间件。`FooStartupFilter` 最终通过 `IWebHostBuilder` 接口的 `ConfigureServices` 方法进行注册。至于中间件类型 `BarMiddleware`, 我们调用 `IWebHostBuilder` 接口的 `Configure` 方法对它进行注册。

```

class Program
{
    static void Main()
    {
        Host.CreateDefaultBuilder().ConfigureWebHostDefaults(builder => builder
            .ConfigureServices(svcs => svcs
                .AddSingleton<IStartupFilter, FooStartupFilter>())
            .Configure(app => app
                .UseMiddleware<BarMiddleware>()
                .Run(context => context.Response.WriteAsync("...=>")))
        .Build()
        .Run();
    }
}

public class FooStartupFilter : IStartupFilter
{
    public Action<IApplicationBuilder> Configure(Action<IApplicationBuilder> next)
    {
        return app => {
            app.UseMiddleware<FooMiddleware>();
            next(app);
        };
    }
}

```

由于 `IStartupFilter` 的 `Configure` 方法会在 `IStartup` 的 `Configure` 方法之前执行，所以对于最终构建的请求处理管道来说，`FooMiddleware` 中间件置于 `BarMiddleware` 中间件前面。换句话说，当管道在处理某个请求的过程中，`FooMiddleware` 中间件的前置请求处理操作会在 `BarMiddleware` 中间件之前执行，而它的后置请求处理操作则在 `BarMiddleware` 中间件之后执行。在启动这个程序之后，如果利用浏览器对该应用发起请求，得到的输出结果如图 11-17 所示。

(S1120)



图 11-17 `IStartupFilter` 和 `Startup` 注册的中间件

管道（中篇）

第 11 章利用一系列实例演示了 ASP.NET Core 应用的编程模式，并借此来体验 ASP.NET Core 管道对请求的处理流程。这个管道由一个服务器和多个有序排列的中间件构成。这看似简单，但 ASP.NET Core 真实管道的构建其实是一个很复杂的过程。由于这个管道对 ASP.NET Core 框架非常重要，为了使读者对此有深刻的认识，本章将介绍真实的管道，而且会按照类似的设计重建一个 Mini 版的 ASP.NET Core 框架。

12.1 中间件委托链

第 13 章会详细介绍 ASP.NET Core 请求处理管道的构建以及它对请求的处理流程，作为对这一部分内容的铺垫，笔者将管道最核心的部分提取出来构建一个 Mini 版的 ASP.NET Core 框架。较之真正的 ASP.NET Core 框架，虽然重建的模拟框架要简单很多，但是它们采用完全一致的设计。为了能够在真实框架中找到对应物，在定义接口或者类型时会采用真实的名称，但是在 API 的定义上会做最大限度的简化。

12.1.1 HttpContext

一个 HttpContext 对象表示针对当前请求的上下文。要理解 HttpContext 上下文的本质，需要从请求处理管道的层面来讲。对于由一个服务器和多个中间件构成的管道来说，面向传输层的服务器负责请求的监听、接收和最终的响应，当它接收到客户端发送的请求后，需要将请求分发给后续中间件进行处理。对于某个中间件来说，完成自身的请求处理任务之后，在大部分情况下需要将请求分发给后续的中间件。请求在服务器与中间件之间，以及在中间件之间的分发是通过共享上下文的方式实现的。

如图 12-1 所示，当服务器接收到请求之后，会创建一个通过 HttpContext 表示的上下文对象，所有中间件都在这个上下文中完成针对请求的处理工作。那么一个 HttpContext 对象究竟会携带什么样的上下文信息？一个 HTTP 事务（Transaction）具有非常清晰的界定，如果从服务器的角度来说就是始于请求的接收，而终于响应的回复，所以请求和响应是两个基本的要素，也是 HttpContext 承载的最核心的上下文信息。



图 12-1 中间件共享上下文

我们可以将请求和响应理解为一个 Web 应用的输入与输出，既然 `HttpContext` 上下文是针对请求和响应的封装，那么应用程序就可以利用这个上下文对象得到当前请求所有的输入信息，也可以利用它完成我们所需的所有输出工作。所以，我们为 ASP.NET Core 模拟框架定义了如下这个极简版本的 `HttpContext` 类型。

```
public class HttpContext
{
    public abstract HttpRequest Request { get; }
    public abstract HttpResponse Response { get; }
}

public class HttpRequest
{
    public abstract Uri Url { get; }
    public abstract NameValueCollection Headers { get; }
    public abstract Stream Body { get; }
}

public class HttpResponse
{
    public abstract int StatusCode { get; set; }
    public abstract NameValueCollection Headers { get; }
    public abstract Stream Body { get; }
}
```

如上面的代码片段所示，我们可以利用 `HttpRequest` 对象得到当前请求的地址、请求消息的报头集合和主体内容。利用 `HttpResponse` 对象，我们不仅可以设置响应的状态码，还可以添加任意的响应报头和写入任意的主体内容。

12.1.2 中间件

`HttpContext` 对象承载了所有与当前请求相关的上下文信息，应用程序针对请求的响应也利用它来完成，所以可以利用一个 `Action<HttpContext>` 类型的委托对象来表示针对请求的处理，我们姑且将它称为请求处理器（Handler）。但 `Action<HttpContext>` 仅仅是请求处理器针对“同步”编程模式的表现形式，对于面向 `Task` 的异步编程模式，这个处理器应该表示成类型为 `Func<HttpContext, Task>` 的委托对象。

由于这个表示请求处理器的委托对象具有非常广泛的应用，所以我们为它专门定义了如下这个 `RequestDelegate` 委托类型，可以看出它就是对 `Func<HttpContext, Task>` 委托的表达。一个 `RequestDelegate` 对象表示的是请求处理器，那么中间件在模型中应如何表达？


```
public delegate Task RequestDelegate(HttpContext context);
```

作为请求处理管道核心组成部分的中间件可以表示成类型为 `Func<RequestDelegate, RequestDelegate>` 的委托对象。换句话说，中间件的输入与输出都是一个 `RequestDelegate` 对象。我们可以这样来理解：对于管道中的某个中间件（图 12-2 所示的第一个中间件）来说，后续中间件组成的管道体现为一个 `RequestDelegate` 对象，由于当前中间件在完成了自身的请求处理任务之后，往往需要将请求分发给后续中间件进行处理，所以它需要将后续中间件构成的 `RequestDelegate` 对象作为输入。

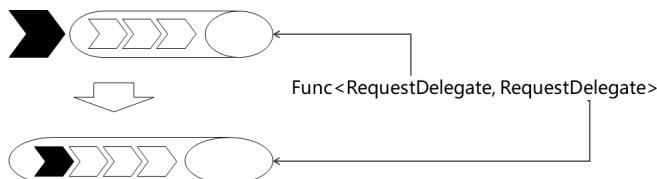


图 12-2 中间件

当代表当前中间件的委托对象执行之后，如果将它自己“纳入”这个管道，那么代表新管道的 `RequestDelegate` 对象就成为该委托对象执行后的输出结果，所以中间件自然就表示成输入和输出类型均为 `RequestDelegate` 的 `Func<RequestDelegate, RequestDelegate>` 对象。

12.1.3 中间件管道的构建

从事软件行业 10 多年来，笔者对架构设计越来越具有这样的认识：好的设计一定是“简单”的。所以在设计某个开发框架时笔者的目标是再简单点。上面介绍的请求处理管道的设计就具有“简单”的特质：`Pipeline = Server + Middlewares`。但是“再简单点”其实是可行的，我们可以将多个中间件组成一个单一的请求处理器。请求处理器可以通过 `RequestDelegate` 对象来表示，所以整个请求处理管道将具有更加简单的表达：`Pipeline = Server + RequestDelegate`（见图 12-3）。

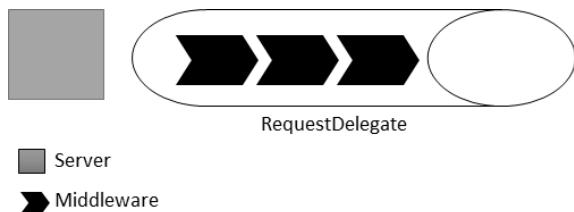


图 12-3 `Pipeline = Server + RequestDelegate`

表示中间件的 `Func<RequestDelegate, RequestDelegate>` 对象向表示请求处理器的 `RequestDelegate` 对象的转换是通过 `IApplicationBuilder` 对象来完成的。从接口命名可以看出，`IApplicationBuilder` 对象是用来构建“应用程序”（`Application`）的，实际上，由所有注册中间件构建的 `RequestDelegate` 对象就是对应用程序的表达，因为应用程序的意图完全是由注册的中间件达成的。

```
public interface IApplicationBuilder
```

```

{
    RequestDelegate Build();
    IApplicationBuilder Use(Func<RequestDelegate, RequestDelegate> middleware);
}

```

如上所示的代码片段是模拟框架对 `IApplicationBuilder` 接口的简化定义。它的 `Use` 方法用来注册中间件，而 `Build` 方法则将所有中间件按照注册的顺序组装成一个 `RequestDelegate` 对象。在如下所示的代码片段中，`ApplicationBuilder` 类型是对 `IApplicationBuilder` 接口的默认实现。我们给出的代码片段还体现了这样一个细节：当我们将注册的中间件转换成一个表示请求处理器的 `RequestDelegate` 对象时，会在管道的尾端添加一个处理器，用来响应一个状态码为 404 的响应。这个细节意味着如果没有注册任何的中间件或者注册的所有中间件都将请求分发给后续管道，那么应用程序会回复一个状态码为 404 的响应。

```

public class ApplicationBuilder : IApplicationBuilder
{
    private readonly IList<Func<RequestDelegate, RequestDelegate>> _middlewares
        = new List<Func<RequestDelegate, RequestDelegate>>();

    public RequestDelegate Build()
    {
        RequestDelegate next = context =>
        {
            context.Response.StatusCode = 404;
            return Task.CompletedTask;
        };
        foreach (var middleware in _middlewares.Reverse())
        {
            next = middleware.Invoke(next);
        }
        return next;
    }

    public IApplicationBuilder Use(Func<RequestDelegate, RequestDelegate> middleware)
    {
        _middlewares.Add(middleware);
        return this;
    }
}

```

12.2 服务器

服务器在管道中的职责非常明确：负责 HTTP 请求的监听、接收和最终的响应。具体来说，启动后的服务器会绑定到指定的端口进行请求监听。一旦有请求抵达，服务器会根据该请求创建代表请求上下文的 `HttpContext` 对象，并将该上下文分发给注册的中间件进行处理。当中间件管道完成了针对请求的处理之后，服务器会将最终生成的响应回复给客户端。

12.2.1 IServer

在模拟的 ASP.NET Core 框架中，我们将服务器定义成一个极度简化的 `IServer` 接口。在如下所示的代码片段中，`IServer` 接口具有唯一的 `StartAsync` 方法，用来启动自身代表的服务器。服务器最终需要将接收的请求分发给注册的中间件，而注册的中间件最终会被 `IApplicationBuilder` 对象构建成一个代表请求处理器的 `RequestDelegate` 对象，`StartAsync` 方法的参数 `handler` 代表的就是这样一个对象。

```
public interface IServer
{
    Task StartAsync(RequestDelegate handler);
}
```

12.2.2 针对服务器的适配

面向应用层的 `HttpContext` 对象是对请求和响应的抽象与封装，但是请求最初是由面向传输层的服务器接收的，最终的响应也会由服务器回复给客户端。所有 ASP.NET Core 应用使用的都是同一个 `HttpContext` 类型，但是它们可以注册不同类型的服务器，应如何解决两者之间的适配问题？计算机领域有这样一句话：“任何问题都可以通过添加一个抽象层的方式来解决，如果解决不了，那就再加一层。”同一个 `HttpContext` 类型与不同服务器类型之间的适配问题自然也可以通过添加一个抽象层来解决。我们将定义在该抽象层的对象称为特性（Feature），特性可以视为对 `HttpContext` 某个方面的抽象化描述。

如图 12-4 所示，我们可以定义一系列特性接口来为 `HttpContext` 提供某个方面的上下文信息，具体的服务器只需要实现这些 `Feature` 接口即可。对于所有用来定义特性的接口，最重要的是提供请求信息的 `IRequestFeature` 接口和完成响应的 `IResponseFeature` 接口。

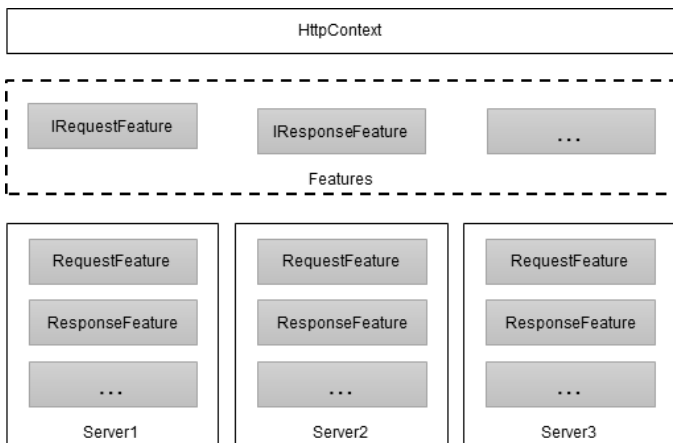


图 12-4 利用特性实现对不同服务器类型的适配

下面阐述用来适配不同服务器类型的特性在代码层面的定义。如下面的代码片段所示，我们定义了一个 `IFeatureCollection` 接口，用来表示存放特性的集合。可以看出，这是一个将 `Type`

和 `Object` 作为 `Key` 与 `Value` 的字典, `Key` 代表注册 `Feature` 所采用的类型, 而 `Value` 代表 `Feature` 对象本身, 也就是说, 我们提供的特性最终是以对应类型 (一般为接口类型) 进行注册的。为了便于编程, 我们定义了 `Set<T>` 方法和 `Get<T>` 方法, 用来设置与获取特性对象。

```
public interface IFeatureCollection : IDictionary<Type, object> { }

public class FeatureCollection : Dictionary<Type, object>, IFeatureCollection { }

public static partial class Extensions
{
    public static T Get<T>(this IFeatureCollection features)
        => features.TryGetValue(typeof(T), out var value) ? (T)value : default(T);

    public static IFeatureCollection Set<T>(this IFeatureCollection features, T feature)
    {
        features[typeof(T)] = feature;
        return features;
    }
}
```

最核心的两种特性类型就是分别用来表示请求和响应的特性, 我们可以采用如下两个接口来表示。可以看出, `IHttpRequestFeature` 接口和 `IHttpResponseFeature` 接口具有与抽象类型 `HttpRequest` 和 `HttpResponse` 完全一致的成员定义。

```
public interface IHttpRequestFeature
{
    Uri                Url { get; }
    NameValueCollection Headers { get; }
    Stream             Body { get; }
}

public interface IHttpResponseFeature
{
    int                StatusCode { get; set; }
    NameValueCollection Headers { get; }
    Stream             Body { get; }
}
```

我们在前面给出了用于描述请求上下文的 `HttpContext` 类型的成员定义, 下面介绍其具体实现。如下面的代码片段所示, 表示请求和响应的 `HttpRequest` 与 `HttpResponse` 分别是由对应的特性 (`IHttpRequestFeature` 对象和 `IHttpResponseFeature` 对象) 创建的。`HttpContext` 对象本身则是通过一个表示特性集合的 `IFeatureCollection` 对象来创建的, 它会在初始化过程中从这个集合中提取出对应的特性来创建 `HttpRequest` 对象和 `HttpResponse` 对象。

```
public class HttpContext
{
    public HttpRequest    Request { get; }
    public HttpResponse   Response { get; }

    public HttpContext(IFeatureCollection features)
    {
```

```

        Request = new HttpRequest(features);
        Response = new HttpResponse(features);
    }
}

public class HttpRequest
{
    private readonly IHttpRequestFeature _feature;

    public Uri Url
        => _feature.Url;
    public NameValueCollection Headers
        => _feature.Headers;
    public Stream Body
        => feature.Body;

    public HttpRequest(IFeatureCollection features)
        => feature = features.Get<IHttpRequestFeature>();
}

public class HttpResponse
{
    private readonly IHttpResponseFeature feature;

    public NameValueCollection Headers
        => feature.Headers;
    public Stream Body
        => _feature.Body;
    public int StatusCode
    {
        get => _feature.StatusCode;
        set => feature.StatusCode = value;
    }

    public HttpResponse(IFeatureCollection features)
        => feature = features.Get<IHttpResponseFeature>();
}

```

换句话说，我们利用 `HttpContext` 对象的 `Request` 属性提取的请求信息最初来源于 `IHttpRequestFeature` 对象，利用它的 `Response` 属性针对响应所做的任意操作最终都会作用到 `IHttpResponseFeature` 对象上。这两个对象最初是由注册的服务器提供的，这正是同一个 ASP.NET Core 应用可以自由地选择不同服务器类型的根源所在。

12.2.3 HttpListenerServer

在对服务器的职责和它与 `HttpContext` 的适配原理有了清晰的认识之后，我们可以尝试定义一个服务器。我们将接下来定义的服务器类型命名为 `HttpListenerServer`，因为它对请求的监听、

接收和响应是由一个 `HttpListener` 对象来实现的。由于服务器接收到请求之后需要借助“特性”的适配来构建统一的请求上下文（即 `HttpContext` 对象），这也是中间件的执行上下文，所以提供针对性的特性实现是自定义服务类型的关键所在。

对 `HttpListener` 有所了解的读者都知道，当它在接收到请求之后同样会创建一个 `HttpContext` 对象表示请求上下文。如果使用 `HttpListener` 对象作为 ASP.NET Core 应用的监听器，就意味着不仅所有的请求信息会来源于这个 `HttpContext` 对象，我们针对请求的响应最终也需要利用这个上下文对象来完成。`HttpListenerServer` 对应特性所起的作用实际上就是在 `HttpContext` 和 `HttpContext` 这两种上下文之间搭建起一座图 12-5 所示的桥梁。

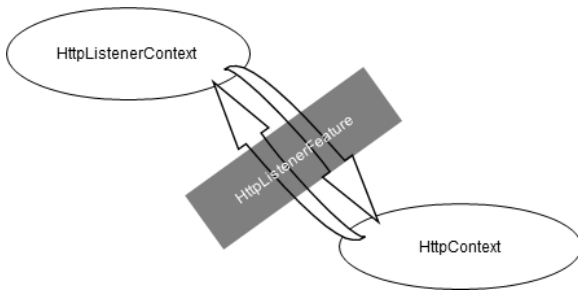


图 12-5 利用 `HttpListenerFeature` 适配 `HttpListenerContext` 和 `HttpContext`

图 12-5 中用来在 `HttpListenerContext` 和 `HttpContext` 这两个上下文类型之间完成适配的特性类型被命名为 `HttpListenerFeature`。如下面的代码片段所示，`HttpListenerFeature` 类型同时实现了针对请求和响应的特性接口 `IHttpRequestFeature` 与 `IHttpResponseFeature`。

```
public class HttpListenerFeature : IHttpRequestFeature, IHttpResponseFeature
{
    private readonly HttpContext context;
    public HttpListenerFeature(HttpContext context)
        => _context = context;

    Uri IHttpRequestFeature.Url
        => _context.Request.Url;
    NameValueCollection IHttpRequestFeature.Headers
        => _context.Request.Headers;
    NameValueCollection IHttpResponseFeature.Headers
        => context.Response.Headers;
    Stream IHttpRequestFeature.Body
        => context.Request.InputStream;
    Stream IHttpResponseFeature.Body
        => _context.Response.OutputStream;
    int IHttpResponseFeature.StatusCode
    {
        get => _context.Response.StatusCode;
        set => context.Response.StatusCode = value;
    }
}
```

创建 `HttpListenerFeature` 对象时需要提供一个 `HttpListenerContext` 对象, `IHttpRequestFeature` 接口的实现成员所提供的请求信息全部来源于这个 `HttpListenerContext` 上下文, `IHttpResponseFeature` 接口的实现成员针对响应的操作最终也转移到这个 `HttpListenerContext` 上下文上。如下所示的代码片段是针对 `HttpListener` 的服务器类型 `HttpListenerServer` 的完整定义。我们在创建 `HttpListenerServer` 对象的时候可以显式提供一组监听地址, 如果没有提供, 监听地址会默认设置“localhost:5000”。在实现的 `StartAsync` 方法中, 我们启动了在构造函数中创建的 `HttpListenerServer` 对象, 并且在一个无限循环中通过调用其 `GetContextAsync` 方法实现了针对请求的监听和接收。

```
public class HttpListenerServer : IServer
{
    private readonly HttpListener _httpListener;
    private readonly string[] urls;

    public HttpListenerServer(params string[] urls)
    {
        _httpListener = new HttpListener();
        urls = urls.Any() ? urls : new string[] { "http://localhost:5000/" };
    }

    public async Task StartAsync(RequestDelegate handler)
    {
        Array.ForEach(_urls, url => _httpListener.Prefixes.Add(url));
        httpListener.Start();
        while (true)
        {
            var listenerContext = await httpListener.GetContextAsync();
            var feature = new HttpListenerFeature(listenerContext);
            var features = new FeatureCollection()
                .Set<IHttpRequestFeature>(feature)
                .Set<IHttpResponseFeature>(feature);
            var httpContext = new HttpContext(features);
            await handler(httpContext);
            listenerContext.Response.Close();
        }
    }
}
```

当 `HttpListener` 监听到抵达的请求后, 我们会得到一个 `HttpListenerContext` 对象, 此时只需要利用它创建一个 `HttpListenerFeature` 对象, 并且分别以 `IHttpRequestFeature` 接口和 `IHttpResponseFeature` 接口的形式注册到创建的 `FeatureCollection` 集合上即可。我们最终利用这个 `FeatureCollection` 集合创建出代表请求上下文的 `HttpContext` 对象, 当将它作为参数调用由所有注册中间件共同构建的 `RequestDelegate` 对象时, 中间件管道将接管并处理该请求。

12.3 承载服务

到目前为止，我们已经了解构成 ASP.NET Core 请求处理管道的两个核心要素（服务器和中间件），现在我们的目标是利用 .NET Core 承载服务来承载这一管道。毫无疑问，还需要通过实现 `IHostedService` 接口来定义对应的承载服务，为此我们定义了一个名为 `WebHostedService` 的承载服务。

12.3.1 WebHostedService

由于服务器是整个请求处理管道的“龙头”，所以从某种意义上来说，启动一个 ASP.NET Core 应用就是为了启动服务器，所以可以将服务的启动在 `WebHostedService` 承载服务中实现。如下面的代码片段所示，创建一个 `WebHostedService` 对象时，需要提供服务器对象和由所有注册中间件构建的 `RequestDelegate` 对象。在实现的 `StartAsync` 方法中，我们只需要调用服务器对象的 `StartAsync` 方法启动它即可。

```
public class WebHostedService : IHostedService
{
    private readonly IServer      _server;
    private readonly RequestDelegate _handler;

    public WebHostedService(IServer server, RequestDelegate handler)
    {
        _server      = server;
        _handler     = handler;
    }

    public Task StartAsync(CancellationToken cancellationToken)
        => _server.StartAsync(_handler);
    public Task StopAsync(CancellationToken cancellationToken)
        => Task.CompletedTask;
}
```

到目前为止，我们基本上已经完成了所有的核心工作，如果能够将一个 `WebHostedService` 实例注册到 .NET Core 的承载系统中，它就能够帮助我们启动一个 ASP.NET Core 应用。为了使这个过程在编程上变得更加便利和“优雅”，我们定义了一个辅助的 `WebHostBuilder` 类型。

12.3.2 WebHostBuilder

要创建一个 `WebHostedService` 对象，必须显式地提供一个表示服务器的 `IServer` 对象，以及由所有注册中间件构建而成的 `RequestDelegate` 对象，`WebHostBuilder` 提供了更加便利和“优雅”的服务器与中间件注册方式。如下面的代码片段所示，`WebHostBuilder` 是对额外两个 `Builder` 对象的封装：一个是用来构建服务宿主的 `IHostBuilder` 对象，另一个是用来注册中间件并最终帮助我们创建 `RequestDelegate` 对象的 `IApplicationBuilder` 对象。

```
public class WebHostBuilder
```



```

{
    public WebHostBuilder(IHostBuilder hostBuilder, IApplicationBuilder applicationBuilder)
    {
        HostBuilder          = hostBuilder;
        ApplicationBuilder    = applicationBuilder;
    }

    public IHostBuilder          HostBuilder { get; }
    public IApplicationBuilder ApplicationBuilder { get; }
}

```

我们为 `WebHostBuilder` 定义了如下两个扩展方法：`UseHttpListenerServer` 方法完成了针对自定义的服务器类型 `HttpListenerServer` 的注册；`Configure` 方法提供了一个 `Action<IApplicationBuilder>` 类型的参数，利用该参数来注册任意中间件。

```

public static partial class Extensions
{
    public static WebHostBuilder UseHttpListenerServer(
        this WebHostBuilder builder, params string[] urls)
    {
        builder.HostBuilder.ConfigureServices(svcs => svcs
            .AddSingleton<IServer>(new HttpListenerServer(urls)));
        return builder;
    }

    public static WebHostBuilder Configure(this WebHostBuilder builder,
        Action<IApplicationBuilder> configure)
    {
        configure?.Invoke(builder.ApplicationBuilder);
        return builder;
    }
}

```

代表 ASP.NET Core 应用的请求处理管道最终是利用承载服务 `WebHostedService` 注册到 .NET Core 的承载系统中的，针对 `WebHostedService` 服务的创建和注册体现在为 `IHostBuilder` 接口定义的 `ConfigureWebHost` 扩展方法上。如下面的代码片段所示，`ConfigureWebHost` 方法定义了一个 `Action<WebHostBuilder>` 类型的参数，利用该参数可以注册服务器、中间件及其他相关服务。

```

public static partial class Extensions
{
    public static IHostBuilder ConfigureWebHost(this IHostBuilder builder,
        Action<WebHostBuilder> configure)
    {
        var webHostBuilder = new WebHostBuilder(builder, new ApplicationBuilder());
        configure?.Invoke(webHostBuilder);
        builder.ConfigureServices(svcs => svcs.AddSingleton<IHostedService>(provider => {
            var server = provider.GetRequiredService<IServer>();
            var handler = webHostBuilder.ApplicationBuilder.Build();
            return new WebHostedService(server, handler);
        }));
    }
}

```

```

    }));
    return builder;
}
}

```

在 `ConfigureWebHost` 方法中，我们创建了一个 `ApplicationBuilder` 对象，并利用它和当前的 `IHostBuilder` 对象创建了一个 `WebHostBuilder` 对象，然后将这个 `WebHostBuilder` 对象作为参数调用了指定的 `Action<WebHostBuilder>` 委托对象。在此之后，我们调用 `IHostBuilder` 接口的 `ConfigureServices` 方法在依赖注入框架中注册了一个用于创建 `WebHostedService` 服务的工厂。对于由该工厂创建的 `WebHostedService` 对象来说，服务器来源于注册的服务，而作为请求处理器的 `RequestDelegate` 对象则由 `ApplicationBuilder` 对象根据注册的中间件构建而成。

12.3.3 应用构建

到目前为止，这个用来模拟 ASP.NET Core 请求处理管道的 Mini 版框架已经构建完成，下面尝试在它上面开发一个简单的应用。如下面的代码片段所示，我们调用静态类型 `Host` 的 `CreateDefaultBuilder` 方法创建了一个 `IHostBuilder` 对象，然后调用 `ConfigureWebHost` 方法并利用提供的 `Action<WebHostBuilder>` 对象注册了 `HttpListenerServer` 服务器和 3 个中间件。在调用 `Build` 方法构建出作为服务宿主的 `IHost` 对象之后，我们调用其 `Run` 方法启动所有承载的 `IHostedService` 服务。

```

class Program
{
    static void Main()
    {
        Host.CreateDefaultBuilder()
            .ConfigureWebHost(builder => builder
                .UseHttpListenerServer()
                .Configure(app => app
                    .Use(FooMiddleware)
                    .Use(BarMiddleware)
                    .Use(BazMiddleware)))
            .Build()
            .Run();
    }

    public static RequestDelegate FooMiddleware(RequestDelegate next)
        => async context =>
    {
        await context.Response.WriteAsync("Foo=>");
        await next(context);
    };

    public static RequestDelegate BarMiddleware(RequestDelegate next)
        => async context =>
    {
        await context.Response.WriteAsync("Bar=>");
    };
}

```

```
        await next(context);
    };

    public static RequestDelegate BazMiddleware(RequestDelegate next)
        => context => context.Response.WriteAsync("Baz");
}
```

由于中间件最终体现为一个类型为 `Func<RequestDelegate, RequestDelegate>` 的委托对象，所以可以利用与之匹配的方法来定义中间件。演示实例中定义的 3 个中间件 (`FooMiddleware`、`BarMiddleware` 和 `BazMiddleware`) 对应的正是 3 个静态方法，它们调用 `WriteAsync` 扩展方法在响应中写了一段文字。

```
public static partial class Extensions
{
    public static Task WriteAsync(this HttpResponse response, string contents)
    {
        var buffer = Encoding.UTF8.GetBytes(contents);
        return response.Body.WriteAsync(buffer, 0, buffer.Length);
    }
}
```

应用启动之后，如果利用浏览器向应用程序采用的默认监听地址（“`http://localhost:5000`”）发送一个请求，得到的输出结果如图 12-6 所示。浏览器上呈现的文字正是注册的 3 个中间件写入的。

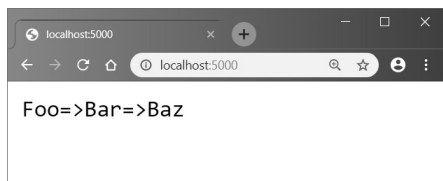


图 12-6 在模拟框架上构建的 ASP.NET Core 应用

管道（下篇）

有了第 11 章和第 12 章的铺垫，读者对 ASP.NET Core 框架的请求处理管道已经有了相对充分的了解。第 12 章使用少量的代码模拟了 ASP.NET Core 框架的实现，虽然两者在设计思想上完全一致，但是省略了太多的细节。本章会弥补这些细节，还原一个真实的 ASP.NET Core 框架。

13.1 请求上下文

ASP.NET Core 请求处理管道由一个服务器和一组有序排列的中间件构成，所有中间件针对请求的处理都在通过 `HttpContext` 对象表示的上下文中进行。由于应用程序总是利用服务器来完成对请求的接收和响应工作，所以原始请求上下文的描述由注册的服务器类型来决定。但是 ASP.NET Core 需要在上层提供具有一致性的编程模型，所以我们需要一个抽象的、不依赖具体服务器类型的请求上下文描述，这就是本节着重介绍的 `HttpContext`。

13.1.1 HttpContext

在第 12 章创建的模拟管道中，我们定义了一个简易版的 `HttpContext` 类，它只包含表示请求和响应的两个属性，实际上，真正的 `HttpContext` 具有更加丰富的成员定义。对于一个 `HttpContext` 对象来说，除了描述请求和响应的 `Request` 属性与 `Response` 属性，我们还可以通过它获取与当前请求相关的其他上下文信息，如用来表示当前请求用户的 `ClaimsPrincipal` 对象、描述当前 HTTP 连接的 `ConnectionInfo` 对象和用于控制 Web Socket 的 `WebSocketManager` 对象等。除此之外，我们还可以通过 `Session` 属性获取并控制当前会话，也可以通过 `TraceIdentifier` 属性获取或者设置调试追踪的 ID。

```
public abstract class HttpContext
{
    public abstract HttpRequest Request { get; }
    public abstract HttpResponse Response { get; }

    public abstract ClaimsPrincipal User { get; set; }
    public abstract ConnectionInfo Connection { get; }
```

```

public abstract WebSocketManager      WebSockets { get; }
public abstract ISession              Session { get; set; }
public abstract string                TraceIdentifier { get; set; }

public abstract IDictionary<object, object> Items { get; set; }
public abstract CancellationToken    RequestAborted { get; set; }
public abstract IServiceProvider     RequestServices { get; set; }
...
}

```

当客户端中止请求（如请求超时）时，我们可以通过 `RequestAborted` 属性返回的 `CancellationToken` 对象接收到通知，进而及时中止正在进行的请求处理操作。如果需要针对整个管道共享一些与当前上下文相关的数据，我们可以将它保存在通过 `Items` 属性表示的字典中。`HttpContext` 的 `RequestServices` 返回的是针对当前请求的 `IServiceProvider` 对象，换句话说，该对象的生命周期与表示当前请求上下文的 `HttpContext` 对象绑定。对于一个 `HttpContext` 对象来说，表示请求和响应的 `Request` 属性与 `Response` 属性是它最重要的两个成员，请求通过如下这个抽象类 `HttpRequest` 表示。

```

public abstract class HttpRequest
{
    public abstract HttpContext      HttpContext { get; }
    public abstract string           Method { get; set; }
    public abstract string           Scheme { get; set; }
    public abstract bool            IsHttps { get; set; }
    public abstract HostString       Host { get; set; }
    public abstract PathString       PathBase { get; set; }
    public abstract PathString       Path { get; set; }
    public abstract QueryString      QueryString { get; set; }
    public abstract IQueryCollection Query { get; set; }
    public abstract string           Protocol { get; set; }
    public abstract IHeaderDictionary Headers { get; }
    public abstract IRequestCookieCollection Cookies { get; set; }
    public abstract string           ContentType { get; set; }
    public abstract Stream           Body { get; set; }
    public abstract bool            HasFormContentType { get; }
    public abstract IFormCollection Form { get; set; }

    public abstract Task<IFormCollection> ReadFormAsync(
        CancellationToken cancellationToken);
}

```

如上所示的抽象类 `HttpRequest` 是对 HTTP 请求的描述，它是 `HttpContext` 对象的只读属性 `Request` 的返回类型。我们可以利用 `HttpRequest` 对象获取与当前请求相关的各种信息，如请求的协议（HTTP 或者 HTTPS）、HTTP 方法、地址，也可以获取代表请求的 HTTP 消息的首部和主题。`HttpRequest` 中的属性/方法列表如表 13-1 所示。

表 13-1 HttpRequest 中的属性/方法列表

属性/方法	含 义
Body	读取请求主体内容的输入流对象
ContentLength	请求主体内容的字节数
ContentType	请求主体内容的媒体类型（如 text/xml、text/json 等）
Cookies	请求携带的 Cookie 列表，对应 HTTP 请求消息的 Cookie 首部。该属性的返回类型为 IRequestCookieCollection 接口，它具有与字典类似的数据结构，其 Key 和 Value 分别代表 Cookie 的名称与值
Form	请求提交的表单。该属性的返回类型为 IFormCollection，它具有一个与字典类似的数据结构，其 Key 和 Value 分别代表表单元素的名称与携带值。由于同一个表单中可以包含多个同名元素，所以 Value 是一个字符串列表
HasFormContentType	请求主体是否具有一个针对表单的媒体类型，一般来说，表单内容采用的媒体类型为 application/x-www-form-urlencoded 或者 multipart/form-data
Headers	请求首部列表。该属性的返回类型为 IHeaderDictionary，它具有一个与字典类似的数据结构，其 Key 和 Value 分别代表首部的名称与携带值。由于同一个请求中可以包含多个同名首部，所以 Value 是一个字符串列表
Host	请求目标地址的主机名（含端口号）。该属性返回的是一个 HostString 对象，它是对主机名称和端口号的封装
IsHttps	是否是一个采用 TLS/SSL 的 HTTPS 请求
Method	请求采用的 HTTP 方法
PathBase	请求的基础路径，一般体现为应用站点所在路径
Path	请求相对于 PathBase 的路径。如果当前请求的 URL 为 “http://www.artech.com/webapp/home/index”（PathBase 为 “/webapp”），那么 Path 属性返回 “/home/index”
Protocol	请求采用的协议及其版本，如 HTTP/1.1 代表针对 1.1 版本的 HTTP 协议。
Query	请求携带的查询字符串。该属性的返回类型为 IQueryCollection，它具有一个与字典类似的数据结构，其 Key 和 Value 分别代表以查询字符串形式定义的变量名称与值。由于查询字符串中可以定义多个同名变量（如 “?foobar=123&foobar=456”），所以 Value 是一个字符串列表
QueryString	请求携带的查询字符串。该属性返回一个 QueryString 对象，它的 Value 属性值代表整个查询字符串的原始表现形式，如 “{?foo=123&bar=456}”
Scheme	请求采用的协议前缀（“http” 或者 “https”）
ReadFormAsync	从请求的主体部分读取表单内容。该属性的返回类型为 IFormCollection，它具有一个与字典类似的数据结构，其 Key 和 Value 分别代表表单元素的名称与携带值。由于同一个表单可以包含多个同名元素，所以 Value 是一个字符串列表

在了解了表示请求的抽象类 HttpRequest 之后，下面介绍另一个与之相对的用于描述响应的 HttpResponseMessage 类型。如下面的代码片段所示，HttpResponse 依然是一个抽象类，我们可以通过它定义的属性和方法来控制对请求的响应。从原则上讲，我们对请求所做的任意形式的响应都可以利用它来实现。

```
public abstract class HttpResponseMessage
{
    public abstract HttpContext HttpContext { get; }
    public abstract int StatusCode { get; set; }
    public abstract IHeaderDictionary Headers { get; }
```

```

public abstract Stream                Body { get; set; }
public abstract long?                 ContentLength { get; set; }
public abstract IResponseCookies     Cookies { get; }
public abstract bool                  HasStarted { get; }

public abstract void OnStarting(Func<object, Task> callback, object state);
public virtual void OnStarting(Func<Task> callback);
public abstract void OnCompleted(Func<object, Task> callback, object state);
public virtual void RegisterForDispose(IDisposable disposable);
public virtual void OnCompleted(Func<Task> callback);
public virtual void Redirect(string location);
public abstract void Redirect(string location, bool permanent);
}

```

当通过表示当前上下文的 `HttpContext` 对象得到表示响应的 `HttpResponse` 对象之后，我们不仅可以将内容写入响应消息的主体部分，还可以设置响应状态码，并添加相应的报头。`HttpResponse` 中的属性/方法列表如表 13-2 所示。

表 13-2 `HttpResponse` 中的属性/方法列表

属性/方法	含 义
<code>Body</code>	将内容写入响应消息主体部分的输出流对象中
<code>ContentLength</code>	响应消息主体内容的长度 (字节数)
<code>ContentType</code>	响应内容采用的媒体类型/MIME 类型
<code>Cookies</code>	返回一个用于设置 (添加或者删除) 响应 Cookie (对应响应消息的 <code>Set-Cookie</code> 首部) 的 <code>ResponseCookies</code> 对象
<code>HasStarted</code>	表示响应是否已经开始发送。由于 HTTP 响应消息总是从首部开始发送，所以这个属性表示响应首部是否开始发送
<code>Headers</code>	响应消息的首部集合。该属性的返回类型为 <code>IHeaderDictionary</code> ，它具有一个与字典类似的数据结构，其 <code>Key</code> 和 <code>Value</code> 分别代表首部的名称与携带值。由于同一个响应消息中可以包含多个同名首部，所以 <code>Value</code> 是一个字符串列表
<code>StatusCode</code>	响应状态码
<code>OnCompleted</code>	注册一个回调操作，以便在响应消息发送结束时自动执行
<code>OnStarting</code>	注册一个回调操作，以便在响应消息开始发送时自动执行
<code>Redirect</code>	发送一个针对指定目标的地址的重定向响应消息。参数 <code>permanent</code> 表示重定向类型，即状态码为“302”的暂时重定向或者状态码为“301”的永久重定向
<code>RegisterForDispose</code>	注册一个需要回收释放的对象，该对象对应的类型必须实现 <code>IDisposable</code> 接口，所谓的释放体现在对其 <code>Dispose</code> 方法的调用

13.1.2 服务器适配

由于应用程序总是利用这个抽象的 `HttpContext` 上下文来获取与当前请求有关的信息，需要完成的所有响应操作也总是作用在这个 `HttpContext` 对象上，所以不同的服务器与这个抽象的 `HttpContext` 需要进行“适配”。通过第 12 章针对模拟框架的介绍可知，ASP.NET Core 框架会采用一种针对特性 (Feature) 的适配方式。

如图 13-1 所示，ASP.NET Core 框架为抽象的 `HttpContext` 定义了一系列标准的特性接口来

对请求上下文的各个方面进行描述。在一系列标准的接口中，最核心的是用来描述请求的 `IHttpRequestFeature` 接口和描述响应的 `IHttpResponseFeature` 接口。我们在应用层使用的 `HttpContext` 上下文就是根据这样一组特性集合来创建的，对于某个具体的服务器来说，它需要提供这些特性接口的实现，并在接收到请求之后利用自行实现的特性来创建 `HttpContext` 上下文。

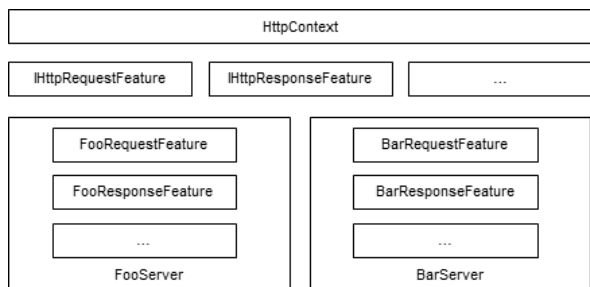


图 13-1 服务器与 HttpContext 之间针对 Feature 的适配

由于 `HttpContext` 上下文是利用服务器提供的特性集合创建的，所以可以统一使用抽象的 `HttpContext` 获取真实的请求信息，也能驱动服务器完成最终的响应工作。在 ASP.NET Core 框架中，由服务器提供的特性集合通过 `IFeatureCollection` 接口表示。第 12 章创建的模拟框架为 `IFeatureCollection` 接口提供了一个极简版的定义，实际上该接口具有更加丰富的成员定义。

```
public interface IFeatureCollection : IEnumerable<KeyValuePair<Type, object>>
{
    TFeature Get<TFeature>();
    void Set<TFeature>(TFeature instance);

    bool    IsReadOnly { get; }
    object  this[Type key] { get; set; }
    int     Revision { get; }
}
```

如上面的代码片段所示，一个 `IFeatureCollection` 对象本质上就是一个 `Key` 和 `Value` 类型分别为 `Type` 与 `Object` 的字典。通过调用 `Set` 方法可以将一个特性对象作为 `Value`，以指定的类型（一般为特性接口）作为 `Key` 添加到这个字典中，并通过 `Get` 方法根据该类型获取它。除此之外，特性的注册和获取也可以利用定义的索引来完成。如果 `IsReadOnly` 属性返回 `True`，就意味着不能注册新的特性或者修改已经注册的特性。整数类型的只读属性 `Revision` 可以视为 `IFeatureCollection` 对象的版本，不论是采用何种方式注册新的特性还是修改现有的特性，都将改变该属性的值。

具有如下定义的 `FeatureCollection` 类型是对 `IFeatureCollection` 接口的默认实现。它具有两个构造函数重载：默认无参构造函数帮助我们创建一个空的特性集合，另一个构造函数则需要指定一个 `IFeatureCollection` 对象来提供默认或者后备特性对象。对于采用第二个构造函数创建的 `FeatureCollection` 对象来说，当我们通过指定的类型试图获取对应的特性对象时，如果没有注册到当前 `FeatureCollection` 对象上，它会从这个后备的 `IFeatureCollection` 对象中查找目标特性。

```
public class FeatureCollection : IFeatureCollection
```



```

{
    //其他成员
    public FeatureCollection();
    public FeatureCollection(IFeatureCollection defaults);
}

```

对于一个 `FeatureCollection` 对象来说，它的 `IsReadOnly` 属性总是返回 `False`，所以它永远是可读可写的。对于调用默认无参构造函数创建的 `FeatureCollection` 对象来说，它的 `Revision` 属性默认返回零。如果我们通过指定另一个 `IFeatureCollection` 对象为参数调用第二个构造函数来创建一个 `FeatureCollection` 对象，前者的 `Revision` 属性值将成为后者同名属性的默认值。无论采用何种形式（调用 `Set` 方法或者索引）添加一个新的特性或者改变一个已经注册的特性，`FeatureCollection` 对象的 `Revision` 属性都将自动递增。上述这些特性都体现在如下所示的调试断言中。

```

var defaults = new FeatureCollection();
Debug.Assert(defaults.Revision == 0);

defaults.Set<IFoo>(new Foo());
Debug.Assert(defaults.Revision == 1);

defaults[typeof(Bar)] = new Bar();
Debug.Assert(defaults.Revision == 2);

FeatureCollection features = new FeatureCollection(defaults);
Debug.Assert(features.Revision == 2);
Debug.Assert(features.Get<IFoo>().GetType() == typeof(Foo));

features.Set<IBaz>(new Baz());
Debug.Assert(features.Revision == 3);

```

最初由服务器提供的 `IFeatureCollection` 对象体现在 `HttpContext` 类型的 `Features` 属性上。虽然特性最初是为了解决不同的服务器类型与统一的 `HttpContext` 上下文之间的适配设计的，但是它的作用不限于此。由于注册的特性是附加在代表当前请求的 `HttpContext` 上下文上，所以可以将任何基于当前请求的对象以特性的方式进行保存，它其实与 `Items` 属性的作用类似。

```

public abstract class HttpContext
{
    public abstract IFeatureCollection Features { get; }
    ...
}

```

上述这种基于特性来实现不同类型的服务器与统一请求上下文之间的适配体现在 `DefaultHttpContext` 类型上，它是对 `HttpContext` 这个抽象类型的默认实现。`DefaultHttpContext` 有一个如下所示的构造函数，作为参数的 `IFeatureCollection` 对象就是由服务器提供的特性集合。

```

public class DefaultHttpContext : HttpContext
{
    public DefaultHttpContext(IFeatureCollection features);
}

```

不论是组成管道的中间件还是建立在管道上的应用，在默认情况下都利用 `DefaultHttpContext` 对象来获取当前请求的相关信息，并利用这个对象完成针对请求的响应。但是 `DefaultHttpContext` 对象在这个过程中只是一个“代理”，针对它的调用（属性或者方法）最终都需要转发给由具体服务器创建的那个原始上下文，在构造函数中指定的 `IFeatureCollection` 对象所代表的特性集合成为这两个上下文对象进行沟通的唯一渠道。对于定义在 `DefaultHttpContext` 中的所有属性，它们几乎都具有一个对应的特性，这些特性都对应一个接口。描述原始 HTTP 上下文的特性接口如表 13-3 所示。

表 13-3 描述原始 HTTP 上下文的特性接口

zFeature	属 性	描 述
<code>IHttpRequestFeature</code>	<code>Request</code>	获取描述请求的基本信息
<code>IHttpResponseFeature</code>	<code>Response</code>	控制对请求的响应
<code>IHttpConnectionFeature</code>	<code>Connection</code>	提供描述当前 HTTP 连接的基本信息
<code>IItemsFeature</code>	<code>Items</code>	提供用户存放针对当前请求的对象容器
<code>IHttpRequestLifetimeFeature</code>	<code>RequestAborted</code>	传递请求处理取消通知和中止当前请求处理
<code>IServiceProvidersFeature</code>	<code>RequestServices</code>	提供根据服务注册创建的 <code>ServiceProvider</code>
<code>ISessionFeature</code>	<code>Session</code>	提供描述当前会话的 <code>Session</code> 对象
<code>IHttpRequestIdentifierFeature</code>	<code>TraceIdentifier</code>	为追踪日志（Trace）提供针对当前请求的唯一标识
<code>IHttpWebSocketFeature</code>	<code>WebSockets</code>	管理 Web Socket

对于表 13-3 列举的众多特性接口，后续相关章节中都会涉及，目前只介绍表示请求和响应的 `IHttpRequestFeature` 接口与 `IHttpResponseFeature` 接口。从下面给出的代码片段可以看出，这两个接口具有与抽象类 `HttpRequest` 和 `HttpResponse` 一致的定义。对于 `DefaultHttpContext` 类型来说，它的 `Request` 属性和 `Response` 属性返回的具体类型为 `DefaultHttpRequest` 与 `DefaultHttpResponse`，它们分别利用这两个特性实现了定义在基类（`HttpRequest` 和 `HttpResponse`）的所有抽象成员。

```
public interface IHttpRequestFeature
{
    Stream                Body { get; set; }
    IDictionary           Headers { get; set; }
    string                Method { get; set; }
    string                Path { get; set; }
    string                PathBase { get; set; }
    string                Protocol { get; set; }
    string                QueryString { get; set; }
    string                Scheme { get; set; }
}

public interface IHttpResponseFeature
{
    Stream                Body { get; set; }
    bool                 HasStarted { get; }
    IDictionary           Headers { get; set; }
    string                ReasonPhrase { get; set; }
}
```

```

int                StatusCode { get; set; }

void OnCompleted(Func<object, Task> callback, object state);
void OnStarting(Func<object, Task> callback, object state);
}

```

13.1.3 获取上下文

如果第三方组件需要获取表示当前请求上下文的 `HttpContext` 对象，就可以通过注入 `IHttpContextAccessor` 服务来实现。`IHttpContextAccessor` 对象提供如下所示的 `HttpContext` 属性返回针对当前请求的 `HttpContext` 对象，由于该属性并不是只读的，所以当前的 `HttpContext` 也可以通过该属性进行设置。

```

public interface IHttpContextAccessor
{
    HttpContext HttpContext { get; set; }
}

```

ASP.NET Core 框架提供的 `HttpContextAccessor` 类型可以作为 `IHttpContextAccessor` 接口的默认实现。从如下所示的代码片段可以看出，`HttpContextAccessor` 将提供的 `HttpContext` 对象以一个 `AsyncLocal<HttpContext>` 对象的方式存储起来，所以在整个请求处理的异步处理流程中都可以利用它得到同一个 `HttpContext` 对象。

```

public class HttpContextAccessor : IHttpContextAccessor
{
    private static AsyncLocal<HttpContext> httpContextCurrent
        = new AsyncLocal<HttpContext>();
    public HttpContext HttpContext
    {
        get => _httpContextCurrent.Value;
        set => _httpContextCurrent.Value = value;
    }
}

```

针对 `IHttpContextAccessor/HttpContextAccessor` 的服务注册可以通过如下所示的 `AddHttpContextAccessor` 扩展方法来完成。由于它调用的是 `IServiceCollection` 接口的 `TryAddSingleton<TService, TImplementation>` 扩展方法，所以不用担心多次调用该方法而出现服务的重复注册问题。

```

public static class HttpServiceCollectionExtensions
{
    public static IServiceCollection AddHttpContextAccessor(
        this IServiceCollection services)
    {
        services.TryAddSingleton<IHttpContextAccessor, HttpContextAccessor>();
        return services;
    }
}

```

13.1.4 上下文的创建与释放

利用注入的 `IHttpContextAccessor` 服务的 `HttpContext` 属性得到当前 `HttpContext` 上下文的前提是该属性在此之前已经被赋值，在默认情况下，该属性是通过默认注册的 `IHttpContextFactory` 服务赋值的。管道在开始处理请求前对 `HttpContext` 上下文的创建，以及请求处理完成后对它的回收释放都是通过 `IHttpContextFactory` 对象完成的。`IHttpContextFactory` 接口定义了如下两个方法：`Create` 方法会根据提供的特性集合来创建 `HttpContext` 对象，`Dispose` 方法则负责将提供的 `HttpContext` 对象释放。

```
public interface IHttpContextFactory
{
    HttpContext Create(IFeatureCollection featureCollection);
    void Dispose(HttpContext httpContext);
}
```

ASP.NET Core 框架提供如下所示的 `DefaultHttpContextFactory` 类型作为对 `IHttpContextFactory` 接口的默认实现，作为默认 `HttpContext` 上下文的 `DefaultHttpContext` 对象就是由它创建的。如下面的代码片段所示，在 `IHttpContextAccessor` 服务被注册的情况下，ASP.NET Core 框架将调用第二个构造函数来创建 `HttpContextFactory` 对象。在 `Create` 方法中，它根据提供的 `IFeatureCollection` 对象创建一个 `DefaultHttpContext` 对象，在返回该对象之前，它会将该对象赋值给 `IHttpContextAccessor` 对象的 `HttpContext` 属性。

```
public class DefaultHttpContextFactory : IHttpContextFactory
{
    private readonly IHttpContextAccessor _httpContextAccessor;
    private readonly FormOptions _formOptions;
    private readonly IServiceScopeFactory serviceScopeFactory;

    public DefaultHttpContextFactory(IServiceProvider serviceProvider)
    {
        _httpContextAccessor = serviceProvider.GetService<IHttpContextAccessor>();
        _formOptions = serviceProvider.GetRequiredService<IOptions<FormOptions>>().Value;
        serviceScopeFactory = serviceProvider.GetRequiredService<IServiceScopeFactory>();
    }

    public HttpContext Create(IFeatureCollection featureCollection)
    {
        var httpContext = CreateHttpContext(featureCollection);
        if (httpContextAccessor != null)
        {
            httpContextAccessor.HttpContext = httpContext;
        }
        httpContext.FormOptions = _formOptions;
        httpContext.ServiceScopeFactory = serviceScopeFactory;
        return httpContext;
    }

    private static DefaultHttpContext CreateHttpContext(
```

```

        IFeatureCollection featureCollection)
    {
        if (featureCollection is IDefaultHttpContextContainer container)
        {
            return container.HttpContext;
        }

        return new DefaultHttpContext(featureCollection);
    }

    public void Dispose(HttpContext httpContext)
    {
        if (_httpContextAccessor != null)
        {
            httpContextAccessor.HttpContext = null;
        }
    }
}

```

如上面的代码片段所示，`HttpContextFactory` 在创建出 `DefaultHttpContext` 对象并将它设置到 `IHttpContextAccessor` 对象的 `HttpContext` 属性上之后，它还会设置 `DefaultHttpContext` 对象的 `FormOptions` 属性和 `ServiceScopeFactory` 属性，前者表示针对表单的配置选项，后者是用来创建服务范围的工厂。当 `Dispose` 方法执行的时候，`DefaultHttpContextFactory` 对象会将 `IHttpContextAccessor` 服务的 `HttpContext` 属性设置为 `Null`。

13.1.5 RequestServices

ASP.NET Core 框架中存在两个用于提供所需服务的依赖注入容器：一个针对应用程序，另一个针对当前请求。绑定到 `HttpContext` 上下文 `RequestServices` 属性上针对当前请求的 `IServiceProvider` 来源于通过 `IServiceProvidersFeature` 接口表示的特性。如下面的代码片段所示，`IServiceProvidersFeature` 接口定义了唯一的属性 `RequestServices`，可以利用它设置和获取与请求绑定的 `IServiceProvider` 对象。

```

public interface IServiceProvidersFeature
{
    IServiceProvider RequestServices { get; set; }
}

```

如下所示的 `RequestServicesFeature` 类型是对 `IServiceProvidersFeature` 接口的默认实现。如下面的代码片段所示，当我们创建一个 `RequestServicesFeature` 对象时，需要提供当前的 `HttpContext` 上下文和创建服务范围的 `IServiceScopeFactory` 工厂。`RequestServicesFeature` 对象的 `RequestServices` 属性提供的 `IServiceProvider` 对象来源于 `IServiceScopeFactory` 对象创建的服务范围，在请求处理过程中提供的 `Scoped` 服务实例的生命周期被限定在此范围之内。

```

public class RequestServicesFeature :
    IServiceProvidersFeature, IDisposable, IAsyncDisposable
{
    private readonly IServiceScopeFactory _scopeFactory;
}

```

```
private IServiceProvider      requestServices;
private IServiceScope        _scope;
private bool                  requestServicesSet;
private readonly HttpContext  context;

public RequestServicesFeature(HttpContext context, IServiceScopeFactory scopeFactory)
{
    _context          = context;
    scopeFactory      = scopeFactory;
}

public IServiceProvider RequestServices
{
    get
    {
        if (!_requestServicesSet && _scopeFactory != null)
        {
            context.Response.RegisterForDisposeAsync(this);
            _scope          = _scopeFactory.CreateScope();
            requestServices = scope.ServiceProvider;
            requestServicesSet = true;
        }
        return requestServices;
    }

    set
    {
        _requestServices      = value;
        requestServicesSet    = true;
    }
}

public ValueTask DisposeAsync()
{
    switch (scope)
    {
        case IAsyncDisposable asyncDisposable:
            var vt = asyncDisposable.DisposeAsync();
            if (!vt.IsCompletedSuccessfully)
            {
                return Awaited(this, vt);
            }
            vt.GetAwaiter().GetResult();
            break;
        case IDisposable disposable:
            disposable.Dispose();
            break;
    }

    _scope          = null;
}
```

```

        requestServices      = null;
        return default;

        static async ValueTask Awaited(RequestServicesFeature servicesFeature,
            ValueTask vt)
        {
            await vt;
            servicesFeature._scope = null;
            servicesFeature.requestServices = null;
        }
    }

    public void Dispose() => DisposeAsync().ConfigureAwait(false).GetAwaiter().GetResult();
}

```

为了在完成请求处理之后释放所有非 Singleton 服务实例，我们必须及时释放创建的服务范围。针对服务范围的释放实现在 `DisposeAsync` 方法中，该方法是针对 `IAsyncDisposable` 接口的实现。在服务范围被创建时，`RequestServicesFeature` 对象会调用表示当前响应的 `HttpResponse` 对象的 `RegisterForDisposeAsync` 方法将自身添加到需要释放的对象列表中，当响应完成之后，`DisposeAsync` 方法会自动被调用，进而将针对当前请求的服务范围联通该范围内的服务实例释放。

前面提及，除了创建返回的 `DefaultHttpContext` 对象，`DefaultHttpContextFactory` 对象还会设置用于创建服务范围的工厂（对应如下所示的 `ServiceScopeFactory` 属性）。用来提供基于当前请求依赖注入容器的 `RequestServicesFeature` 特性正是根据 `IServiceScopeFactory` 对象创建的。

```

public sealed class DefaultHttpContext : HttpContext
{
    public override IServiceProvider      RequestServices {get;set;}
    public IServiceScopeFactory          ServiceScopeFactory { get; set; }
}

```

13.2 IServer + IHttpApplication

ASP.NET Core 的请求处理管道由一个服务器和一组中间件构成，但对于面向传输层的服务器来说，它其实没有中间件的概念。当服务器接收到请求之后，会将该请求分发给一个处理器进行处理，对服务器而言，这个处理器就是一个 HTTP 应用，此应用通过 `IHttpApplication<TContext>` 接口来表示。由于服务器是通过 `IServer` 接口表示的，所以可以将 ASP.NET Core 框架的核心视为由 `IServer` 和 `IHttpApplication<TContext>` 对象组成的管道（见图 13-2）。



图 13-2 由 `IServer` 和 `IHttpApplication<TContext>` 对象组成的管道

13.2.1 IServer

由于服务器是整个请求处理管道的“龙头”，所以启动和关闭应用的最终目的是启动和关闭服务器。ASP.NET Core 框架中的服务器通过 `IServer` 接口来表示，该接口具有如下所示的 3 个成

员，其中由服务器提供的特性就保存在其 `Features` 属性表示的 `IFeatureCollection` 集合中。`IServer` 接口的 `StartAsync<TContext>` 方法与 `StopAsync` 方法分别用来启动和关闭服务器。

```
public interface IServer : IDisposable
{
    IFeatureCollection Features { get; }

    Task StartAsync<TContext>(IHttpApplication<TContext> application,
        CancellationToken cancellationToken);
    Task StopAsync(CancellationToken cancellationToken);
}
```

服务器在开始监听请求之前总是绑定一个或者多个监听地址，这个地址是应用程序从外部指定的。具体来说，应用程序指定的监听地址会封装成一个特性，并且在服务器启动之前被添加到它的特性集合中。这个承载了监听地址列表的特性通过如下所示的 `IServerAddressesFeature` 接口来表示，该接口除了有一个表示地址列表的 `Addresses` 属性，还有一个布尔类型的 `PreferHostingUrls` 属性，该属性表示如果监听地址同时设置到承载系统配置和服务器上，是否优先考虑使用前者。

```
public interface IServerAddressesFeature
{
    ICollection<string> Addresses { get; }
    bool PreferHostingUrls { get; set; }
}
```

正如前面所说，服务器将用来处理由它接收请求的处理器会被视为一个通过 `IHttpApplication<TContext>` 接口表示的应用，所以可以将 ASP.NET Core 的请求处理管道视为 `IServer` 对象和 `IHttpApplication<TContext>` 对象的组合。当调用 `IServer` 对象的 `StartAsync<TContext>` 方法启动服务器时，我们需要提供这个用来处理请求的 `IHttpApplication<TContext>` 对象。`IHttpApplication<TContext>` 采用基于上下文的请求处理方式，泛型参数 `TContext` 代表的就是上下文的类型。在 `IHttpApplication<TContext>` 处理请求之前，它需要先创建一个上下文对象，该上下文会在请求处理结束之后被释放。上下文的创建、释放和自身对请求的处理实现在该接口 3 个对应的方法（`CreateContext`、`DisposeContext` 和 `ProcessRequestAsync`）中。

```
public interface IHttpApplication<TContext>
{
    TContext CreateContext(IFeatureCollection contextFeatures);
    void DisposeContext(TContext context, Exception exception);
    Task ProcessRequestAsync(TContext context);
}
```

13.2.2 HostingApplication

ASP.NET Core 框架利用如下所示的 `HostingApplication` 类型作为 `IHttpApplication<TContext>` 接口的默认实现，它使用一个内嵌的 `Context` 类型来表示处理请求的上下文。一个 `Context` 对象是对一个 `HttpContext` 对象的封装，同时承载了一些与诊断相关的信息。

```
public class HostingApplication : IHttpApplication<HostingApplication.Context>
{
```



```

...
public struct Context
{
    public HttpContext    HttpContext { get; set; }

    public IDisposable    Scope { get; set; }
    public long           StartTimestamp { get; set; }
    public bool           EventLogEnabled { get; set; }
    public Activity       Activity { get; set; }
}
}

```

HostingApplication 对象会在开始和完成请求处理，以及在请求过程中出现异常时发出一些诊断日志事件。具体来说，**HostingApplication** 对象会采用 3 种不同的诊断日志形式，包括基于 **DiagnosticSource** 和 **EventSource** 的诊断日志以及基于 .NET Core 日志系统的日志。**Context** 除 **HttpContext** 外的其他属性都与诊断日志有关。具体来说，**Context** 的 **Scope** 是为 **ILogger** 创建的针对当前请求的日志范围（第 9 章有对日志范围的详细介绍），此日志范围会携带唯一标识每个请求的 ID，如果注册 **ILoggerProvider** 提供的 **ILogger** 支持日志范围，它可以将这个请求 ID 记录下来，那么我们就可以利用这个 ID 将针对同一请求的多条日志消息组织起来做针对性分析。

HostingApplication 对象会在请求结束之后记录当前请求处理的耗时，所以它在开始处理请求时就会记录当前的时间戳，**Context** 的 **StartTimestamp** 属性表示开始处理请求的时间戳。它的 **EventLogEnabled** 属性表示针对 **EventSource** 的事件日志是否开启，而 **Activity** 属性则与针对 **DiagnosticSource** 的诊断日志有关，**Activity** 代表基于当前请求处理的活动。

虽然 ASP.NET Core 应用的请求处理完全由 **HostingApplication** 对象负责，但是该类型的实现逻辑其实是很简单的，因为它将具体的请求处理分发给一个 **RequestDelegate** 对象，该对象表示的正是所有注册中间件组成的委托链。在创建 **HostingApplication** 对象时除了需要提供 **RequestDelegate** 对象，还需要提供用于创建 **HttpContext** 上下文的 **IHttpContextFactory** 对象，以及与诊断日志有关的 **ILogger** 对象和 **DiagnosticListener** 对象，它们被用来创建上面提到过的 **HostingApplicationDiagnostics** 对象。

```

public class HostingApplication : IHttpApplication<HostingApplication.Context>
{
    private readonly RequestDelegate    _application;
    private HostingApplicationDiagnostics    diagnostics;
    private readonly IHttpContextFactory    _httpContextFactory;

    public HostingApplication(RequestDelegate application, ILogger logger,
        DiagnosticListener diagnosticSource, IHttpContextFactory httpContextFactory)
    {
        application            = application;
        _diagnostics           = new HostingApplicationDiagnostics(logger, diagnosticSource);
        _httpContextFactory    = httpContextFactory;
    }

    public Context CreateContext(IFeatureCollection contextFeatures)

```

```

{
    var context = new Context();
    var httpContext = httpContextFactory.Create(context.Features);
    diagnostics.BeginRequest(httpContext, ref context);
    context.HttpContext = httpContext;
    return context;
}

public Task ProcessRequestAsync(Context context)
    => _application(context.HttpContext);

public void DisposeContext(Context context, Exception exception)
{
    var httpContext = context.HttpContext;
    diagnostics.RequestEnd(httpContext, exception, context);
    _httpContextFactory.Dispose(httpContext);
    _diagnostics.ContextDisposed(context);
}
}

```

如上面的代码片段所示，当 `CreateContext` 方法被调用时，`HostingApplication` 对象会利用 `IHttpContextFactory` 工厂创建出当前 `HttpContext` 上下文，并进一步将它封装成一个 `Context` 对象。在返回这个 `Context` 对象之前，它会调用 `HostingApplicationDiagnostics` 对象的 `BeginRequest` 方法记录相应的诊断日志。用来真正处理当前请求的 `ProcessRequestAsync` 方法比较简单，只需要调用代表中间件委托链的 `RequestDelegate` 对象即可。

对于用来释放上下文的 `DisposeContext` 方法来说，它会利用 `IHttpContextFactory` 对象的 `Dispose` 方法来释放创建的 `HttpContext` 对象。换句话说，`HttpContext` 上下文的生命周期是由 `HostingApplication` 对象控制的。完成针对 `HttpContext` 上下文的释放之后，`HostingApplication` 对象会利用 `HostingApplicationDiagnostics` 对象记录相应的诊断日志。`Context` 的 `Scope` 属性表示的日志范围就是在调用 `HostingApplicationDiagnostics` 对象的 `ContextDisposed` 方法时释放的。如果将 `HostingApplication` 对象引入 ASP.NET Core 的请求处理管道，那么完整的管道就体现为图 13-3 所示的结构。

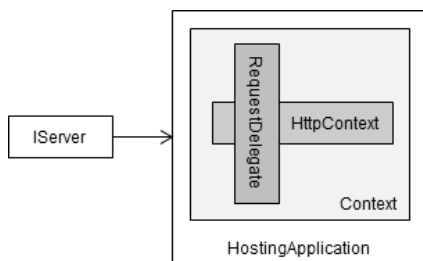


图 13-3 由 `IServer` 和 `HostingApplication` 组成的管道

13.2.3 诊断日志

很多人可能对 ASP.NET Core 框架自身记录的诊断日志并不关心，其实很多时候这些日志对纠错排错和性能监控提供了很有用的信息。例如，假设需要创建一个 APM（Application Performance Management）来监控 ASP.NET Core 处理请求的性能及出现的异常，那么我们完全可以将 `HostingApplication` 对象记录的日志作为收集的原始数据。实际上，目前很多 APM（如 Elastic APM 和 SkyWalking APM 等）针对 ASP.NET Core 应用的客户端都是利用这种方式收集请求调用链信息的。

日志系统

为了确定什么样的信息会被作为诊断日志记录下来，下面介绍一个简单的实例，将 `HostingApplication` 对象写入的诊断日志输出到控制台上。前面提及，`HostingApplication` 对象会将相同的诊断信息以 3 种不同的方式进行记录，其中包含第 9 章介绍的日志系统，所以我们可以通过注册对应 `ILoggerProvider` 对象的方式将日志内容写入对应的输出渠道。

整个演示实例如下面的代码片段所示：首先通过调用 `IWebHostBuilder` 接口的 `ConfigureLogging` 方法注册一个 `ConsoleLoggerProvider` 对象，并开启针对日志范围的支持。我们调用 `IApplicationBuilder` 接口的 `Run` 扩展方法注册了一个中间件，该中间件在处理请求时会利用表示当前请求上下文的 `HttpContext` 对象得到与之绑定的 `IServiceProvider` 对象，并进一步从中提取出用于发送日志事件的 `ILogger<Program>` 对象，我们利用它写入一条 `Information` 等级的日志。如果请求路径为 `“/error”`，那么该中间件会抛出一个 `InvalidOperationException` 类型的异常。

```
public class Program
{
    public static void Main()
    {
        Host.CreateDefaultBuilder()
            .ConfigureLogging(builder => builder.AddConsole(
                options => options.IncludeScopes = true))
            .ConfigureWebHostDefaults(builder => builder
                .Configure(app => app.Run(context =>
                    {
                        var logger = context.RequestServices
                            .GetRequiredService<ILogger<Program>>();
                        logger.LogInformation($"Log for event FooBar");
                        if (context.Request.Path == new PathString("/error"))
                        {
                            throw new InvalidOperationException(
                                "Manually throw exception.");
                        }
                        return Task.CompletedTask;
                    }
                )))
            .Build()
            .Run();
    }
}
```

```
}

```

在启动程序之后，我们利用浏览器采用不同的路径（“/foobar”和“/error”）向应用发送了两次请求，演示程序的控制台上呈现的输出结果如图 13-4 所示。由于我们开启了日志范围的支持，所以被 ConsoleLogger 记录下来的日志都会携带日志范围的信息。日志范围的唯一标识被称为请求 ID（Request ID），它由当前的连接 ID 和一个序列号组成。从图 13-4 可以看出，两次请求的 ID 分别是“0HLO4ON65ALGG:00000001”和“0HLO4ON65ALGG:00000002”。由于采用的是长连接，并且两次请求共享同一个连接，所以它们具有相同的连接 ID（“0HLO4ON65ALGG”）。同一连接的多次请求将一个自增的序列号（“00000001”和“00000002”）作为唯一标识。（S1301）

```

C:\App>dotnet run
Info: Microsoft.Hosting.Lifetime[0]
      Now listening on: http://localhost:5000
Info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
Info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
Info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\App
Info: Microsoft.AspNetCore.Hosting.Diagnostics[1]
      => ConnectionId:0HLO4ON65ALGG => RequestPath:/foobar RequestId:0HLO4ON65ALGG:00000001, ActivityId:|e3fab7e0-4c4487afb681ec68.
      Request starting HTTP/1.1 GET http://localhost:5000/foobar
Info: App.Program[0]
      => ConnectionId:0HLO4ON65ALGG => RequestPath:/foobar RequestId:0HLO4ON65ALGG:00000001, ActivityId:|e3fab7e0-4c4487afb681ec68.
      Log for event Foobar
Info: Microsoft.AspNetCore.Hosting.Diagnostics[2]
      => ConnectionId:0HLO4ON65ALGG => RequestPath:/foobar RequestId:0HLO4ON65ALGG:00000001, ActivityId:|e3fab7e0-4c4487afb681ec68.
      Request finished in 8.8081ms 200
Info: Microsoft.AspNetCore.Hosting.Diagnostics[1]
      => ConnectionId:0HLO4ON65ALGG => RequestPath:/error RequestId:0HLO4ON65ALGG:00000002, ActivityId:|e3fab7e1-4c4487afb681ec68.
      Request starting HTTP/1.1 GET http://localhost:5000/error
Info: App.Program[0]
      => ConnectionId:0HLO4ON65ALGG => RequestPath:/error RequestId:0HLO4ON65ALGG:00000002, ActivityId:|e3fab7e1-4c4487afb681ec68.
      Log for event Foobar
[Error] Microsoft.AspNetCore.Server.Kestrel[13]
      => ConnectionId:0HLO4ON65ALGG => RequestPath:/error RequestId:0HLO4ON65ALGG:00000002, ActivityId:|e3fab7e1-4c4487afb681ec68.
      Connection id "0HLO4ON65ALGG", Request id "0HLO4ON65ALGG:00000002": An unhandled exception was thrown by the application.
System.InvalidOperationException: Manually throw exception.
      at App.Program.<>c.<Main>b__0_4(HttpContext context) in C:\App\Program.cs:line 31
      at Microsoft.AspNetCore.HostFiltering.HostFilteringMiddleware.Invoke(HttpContext context)
      at Microsoft.AspNetCore.Hosting.Internal.HostingApplication.ProcessRequestAsync(Context context)
      at Microsoft.AspNetCore.Server.Kestrel.Core.Internal.Http.HttpProtocol.ProcessRequests[TContext](IHostingApplication`1 application)
Info: Microsoft.AspNetCore.Hosting.Diagnostics[2]
      => ConnectionId:0HLO4ON65ALGG => RequestPath:/error RequestId:0HLO4ON65ALGG:00000002, ActivityId:|e3fab7e1-4c4487afb681ec68.
      Request finished in 43.8156ms 500
  
```

图 13-4 捕捉 HostingApplication 记录的诊断日志

除了用于唯一表示每个请求的请求 ID，日志范围承载的信息还包括请求指向的路径，这也可以从图 13-4 所示的输出接口看出来。另外，上述请求 ID 实际上对应 HttpContext 类型的 TraceIdentifier 属性。如果需要进行跨应用的调用链跟踪，所有相关日志就可以通过共享 TraceIdentifier 属性构建整个调用链。

```

public abstract class HttpContext
{
    public abstract string TraceIdentifier { get; set; }
    ...
}
  
```

对于两次采用不同路径的请求，控制台共捕获了 7 条日志，其中类别为 App.Program 的日志是应用程序自行写入的，HostingApplication 写入日志的类别为“Microsoft.AspNetCore.”

Hosting.Diagnostics”。对于第一次请求的 3 条日志消息，第一条是在 HostingApplication 开始处理请求时写入的，我们利用这条日志获知请求的 HTTP 版本（HTTP/1.1）、HTTP 方法（GET）和请求 URL。对于包含主体内容的请求，请求主体内容的媒体类型（Content-Type）和大小（Content-Length）也会一并记录下来。当 HostingApplication 对象处理完请求后会写入第三条日志，日志承载的信息包括请求处理耗时（67.877 6 毫秒）和响应状态码（200）。如果响应具有主体内容，对应的媒体类型同样会被记录下来。（S1301）

对于第二次请求，由于我们人为抛出了一个异常，所以异常的信息被写入日志。但是如果足够仔细，就会发现这条等级为 Error 的日志并不是由 HostingApplication 对象写入的，而是作为服务器的 KestrelServer 写入的，因为该日志采用的类别为“Microsoft.AspNetCore.Server.Kestrel”。换句话说，HostingApplication 对象利用 ILogger 记录的日志中并不包含应用的异常信息。

DiagnosticSource 诊断日志

HostingApplication 采用的 3 种日志形式还包括基于 DiagnosticSource 对象的诊断日志，所以我们可以通过注册诊断监听器来收集诊断信息。如果通过这种方式获取诊断信息，就需要预先知道诊断日志事件的名称和内容荷载的数据结构。通过查看 HostingApplication 类型的源代码，我们会发现它针对“开始请求”、“结束请求”和“未处理异常”这 3 类诊断日志事件对应的名称，具体如下。

- 开始请求：Microsoft.AspNetCore.Hosting.BeginRequest。
- 结束请求：Microsoft.AspNetCore.Hosting.EndRequest。
- 未处理异常：Microsoft.AspNetCore.Hosting.UnhandledException。

至于针对诊断日志消息的内容荷载（Payload）的结构，上述 3 类诊断事件具有两个相同的成员，分别是表示当前请求上下文的 HttpContext 和通过一个 Int64 整数表示的当前时间戳，对应的数据成员的名称分别为 httpContext 和 timestamp。对于未处理异常诊断事件，它承载的内容荷载还包括一个额外的成员，那就是表示抛出异常的 Exception 对象，对应的成员名称为 exception。

既然我们已经知道事件的名称和诊断承载数据的成员，所以可以定义如下所示的 DiagnosticCollector 类型作为诊断监听器（需要针对 NuGet 包“Microsoft.Extensions.DiagnosticAdapter”的引用）。针对上述 3 类诊断事件，我们在 DiagnosticCollector 类型中定义了 3 个对应的方法，各个方法通过标注的 DiagnosticNameAttribute 特性设置了对应的诊断事件。我们根据诊断数据承载的结构定义了匹配的参数，所以 DiagnosticSource 对象写入诊断日志提供的诊断数据将自动绑定到对应的参数上。如果读者对如何监听并捕捉由 DiagnosticSource 对象发出的诊断日志事件不熟悉，可参阅第 8 章的相关内容。

```
public class DiagnosticCollector
{
    [DiagnosticName("Microsoft.AspNetCore.Hosting.BeginRequest")]
    public void OnRequestStart(HttpContext httpContext, long timestamp)
    {
```

```

        var request = httpContext.Request;
        Console.WriteLine($"Request starting {request.Protocol} {request.Method}
            {request.Scheme}://{request.Host}{request.PathBase}{request.Path}");
        httpContext.Items["StartTimestamp"] = timestamp;
    }

    [DiagnosticName("Microsoft.AspNetCore.Hosting.EndRequest")]
    public void OnRequestEnd(HttpContext httpContext, long timestamp)
    {
        var startTimestamp = long.Parse(httpContext.Items["StartTimestamp"].ToString());
        var timestampToTicks = TimeSpan.TicksPerSecond / (double)Stopwatch.Frequency;
        var elapsed = new TimeSpan((long)(timestampToTicks *
            (timestamp - startTimestamp)));
        Console.WriteLine($"Request finished in {elapsed.TotalMilliseconds}ms
            {httpContext.Response.StatusCode}");
    }

    [DiagnosticName("Microsoft.AspNetCore.Hosting.UnhandledException")]
    public void OnException(HttpContext httpContext, long timestamp, Exception exception)
    {
        OnRequestEnd(httpContext, timestamp);

        Console.WriteLine($"{exception.Message}\nType:{exception.GetType()}\nStacktrace:
            {exception.StackTrace}");
    }
}

```

可以在针对“开始请求”诊断事件的 `OnRequestStart` 方法中输出当前请求的 HTTP 版本、HTTP 方法和 URL。为了能够计算整个请求处理的耗时，我们将当前时间戳保存在 `HttpContext` 上下文的 `Items` 集合中。在针对“结束请求”诊断事件的 `OnRequestEnd` 方法中，我们将这个时间戳从 `HttpContext` 上下文中提取出来，结合当前时间戳计算出请求处理耗时，该耗时和响应的状态码最终会被写入控制台。针对“未处理异常”诊断事件的 `OnException` 方法则在调用 `OnRequestEnd` 方法之后将异常的消息、类型和跟踪堆栈输出到控制台上。

如下面的代码片段所示，在注册的 `Startup` 类型中，我们在 `Configure` 方法注入 `Diagnostic Listener` 服务，并调用它的 `SubscribeWithAdapter` 扩展方法将上述 `DiagnosticCollector` 对象注册为诊断日志的订阅者。与此同时，我们调用 `IApplicationBuilder` 接口的 `Run` 扩展方法注册了一个中间件，该中间件会在请求路径为“/error”的情况下抛出一个异常。

```

public class Program
{
    public static void Main()
    {
        Host.CreateDefaultBuilder()
            .ConfigureLogging(builder => builder.ClearProviders())
            .ConfigureWebHostDefaults(builder => builder.UseStartup<Startup>())
            .Build()
            .Run();
    }
}

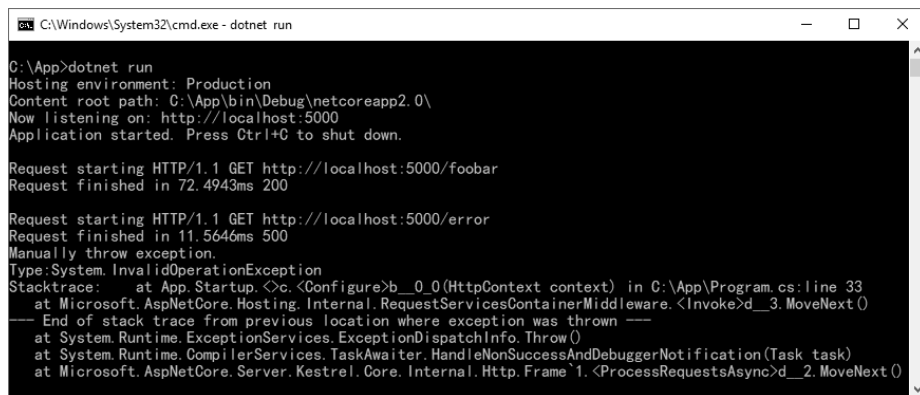
```

```

public class Startup
{
    public void Configure(IApplicationBuilder app, DiagnosticListener listener)
    {
        listener.SubscribeWithAdapter(new DiagnosticCollector());
        app.Run(context =>
        {
            if (context.Request.Path == new PathString("/error"))
            {
                throw new InvalidOperationException("Manually throw exception.");
            }
            return Task.CompletedTask;
        });
    }
}

```

待演示实例正常启动后，可以采用不同的路径（“/foobar”和“/error”）对应用程序发送两个请求，服务端控制台会以图 13-5 所示的形式输出 DiagnosticCollector 对象收集的诊断信息。如果我们试图创建一个针对 ASP.NET Core 的 APM 框架来监控请求处理的性能和出现的异常，可以采用这样的方案来收集原始的诊断信息。（S1302）



```

C:\Windows\System32\cmd.exe - dotnet run
C:\App>dotnet run
Hosting environment: Production
Content root path: C:\App\bin\Debug\netcoreapp2.0\
Now listening on: http://localhost:5000
Application started. Press Ctrl+C to shut down.

Request starting HTTP/1.1 GET http://localhost:5000/foobar
Request finished in 72.4943ms 200

Request starting HTTP/1.1 GET http://localhost:5000/error
Request finished in 11.5646ms 500
Manually throw exception.
Type: System.InvalidOperationException
Stacktrace:
   at App.Startup.<>c.<Configure>b__0_0(HttpContext context) in C:\App\Program.cs:line 33
   at Microsoft.AspNetCore.Hosting.Internal.RequestServicesContainerMiddleware.<Invoke>d__3.MoveNext()
--- End of stack trace from previous location where exception was thrown ---
   at System.Runtime.ExceptionServices.ExceptionDispatchInfo.Throw()
   at System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task task)
   at Microsoft.AspNetCore.Server.Kestrel.Core.Internal.Http.Frame`1.<ProcessRequestsAsync>d__2.MoveNext()

```

图 13-5 利用注册的诊断监听器获取诊断日志

EventSource 事件日志

除了上述两种日志形式，HostingApplication 对象针对每个请求的处理过程中还会利用 EventSource 对象发出相应的日志事件。除此之外，在启动和关闭应用程序（实际上就是启动和关闭 IWebHost 对象）时，同一个 EventSource 对象还会被使用。这个 EventSource 类型采用的名称为 Microsoft.AspNetCore.Hosting，上述 5 个日志事件对应的名称如下。

- 启动应用程序：HostStart。
- 开始处理请求：RequestStart。
- 请求处理结束：RequestStop。
- 未处理异常：UnhandledException。

- 关闭应用程序：HostStop。

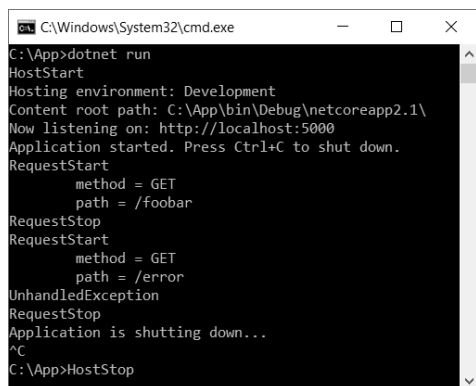
我们可以通过如下所示的实例来演示如何利用创建的 `EventListener` 对象来监听上述 5 个日志事件。如下面的代码片段所示，我们定义了派生于抽象类 `EventListener` 的 `DiagnosticCollector`。在启动应用前，我们创建了这个 `DiagnosticCollector` 对象，并通过注册其 `EventSourceCreated` 事件开启了针对目标名称为 `Microsoft.AspNetCore.Hosting` 的 `EventSource` 的监听。在注册的 `EventWritten` 事件中，我们将监听到的事件名称的负载内容输出到控制台上。

```
public class Program
{
    private sealed class DiagnosticCollector : EventListener {}
    static void Main()
    {
        var listener = new DiagnosticCollector();
        listener.EventSourceCreated += (sender, args) =>
        {
            if (args.EventSource.Name == "Microsoft.AspNetCore.Hosting")
            {
                listener.EnableEvents(args.EventSource, EventLevel.LogAlways);
            }
        };
        listener.EventWritten += (sender, args) =>
        {
            Console.WriteLine(args.EventName);
            for (int index = 0; index < args.PayloadNames.Count; index++)
            {
                Console.WriteLine($"{args.PayloadNames[index]} = {args.Payload[index]}");
            }
        };

        Host.CreateDefaultBuilder()
            .ConfigureLogging(builder => builder.ClearProviders())
            .ConfigureWebHostDefaults(builder => builder
                .Configure(app => app.Run(context =>
                    {
                        if (context.Request.Path == new PathString("/error"))
                        {
                            throw new InvalidOperationException("Manually throw exception.");
                        }
                        return Task.CompletedTask;
                    }
                )))
            .Build()
            .Run();
    }
}
```

以命令行的形式启动这个演示程序后，从图 13-6 所示的输出结果可以看到名为 `HostStart` 的事件被发出。然后采用目标地址“`http://localhost:5000/foobar`”和“`http:// http://localhost:5000/error`”对应用程序发送两个请求，从输出结果可以看出，应用程序针对前者的处理过程会发出

RequestStart 事件和 RequestStop 事件，针对后者的处理则会因为抛出的异常发出额外的事件 UnhandledException。输入“Ctrl+C”关闭应用后，名称为 HostStop 的事件被发出。对于通过 EventSource 发出的 5 个事件，只有 RequestStart 事件会将请求的 HTTP 方法（GET）和路径（“/foobar”和“/error”）作为负载内容，其他事件都不会携带任何负载内容。（S1303）



```

C:\Windows\System32\cmd.exe
C:\App>dotnet run
HostStart
Hosting environment: Development
Content root path: C:\App\bin\Debug\netcoreapp2.1\
Now listening on: http://localhost:5000
Application started. Press Ctrl+C to shut down.
RequestStart
  method = GET
  path = /foobar
RequestStop
RequestStart
  method = GET
  path = /error
UnhandledException
RequestStop
Application is shutting down...
^C
C:\App>HostStop
  
```

图 13-6 利用注册 EventListener 监听器获取诊断日志

13.3 中间件委托链

ASP.NET Core 应用默认的请求处理管道是由注册的 `IServer` 对象和 `HostingApplication` 对象组成的，后者利用一个在创建时提供的 `RequestDelegate` 对象来处理 `IServer` 对象分发给它的请求。而 `RequestDelegate` 对象实际上是由所有的中间件按照注册顺序创建的。换句话说，这个 `RequestDelegate` 对象是对中间件委托链的体现。如果将 `RequestDelegate` 替换成原始的中间件，那么 ASP.NET Core 应用的请求处理管道体现为图 13-7 所示的形式。

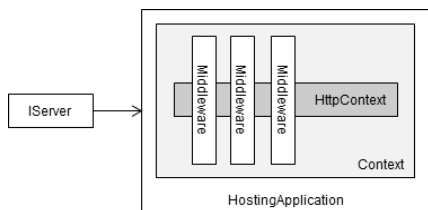


图 13-7 完整的请求处理管道

13.3.1 IApplicationBuilder

对于一个 ASP.NET Core 应用来说，它对请求的处理完全体现在注册的中间件上，所以“应用”从某种意义上讲体现在通过所有注册中间件创建的 `RequestDelegate` 对象上。正因为如此，ASP.NET Core 框架才将构建这个 `RequestDelegate` 对象的接口命名为 `IApplicationBuilder`。`IApplicationBuilder` 是 ASP.NET Core 框架中的一个核心对象，我们将中间件注册在它上面，并且最终利用它来创建代表中间件委托链的 `RequestDelegate` 对象。

如下所示的代码片段是 `IApplicationBuilder` 接口的定义。该接口定义了 3 个属性：`ApplicationServices` 属性代表针对当前应用程序的依赖注入容器，`ServerFeatures` 属性则返回服务器提供的特性集合，`Properties` 属性返回的字典则代表一个可以用来存放任意属性的容器。

```
public interface IApplicationBuilder
{
    IServiceProvider ApplicationServices { get; set; }
    IFeatureCollection ServerFeatures { get; }
    IDictionary<string, object> Properties { get; }

    IApplicationBuilder Use(Func<RequestDelegate, RequestDelegate> middleware);
    RequestDelegate Build();
    IApplicationBuilder New();
}
```

通过第 12 章的介绍可知，ASP.NET Core 应用的中间件体现为一个 `Func<RequestDelegate, RequestDelegate>` 对象，而针对中间件的注册则通过调用 `IApplicationBuilder` 接口的 `Use` 方法来完成。`IApplicationBuilder` 对象最终的目的就是根据注册的中间件创建作为代表中间件委托链的 `RequestDelegate` 对象，这个目标是通过调用 `Build` 方法来完成的。`New` 方法可以帮助我们创建一个新的 `IApplicationBuilder` 对象，除了已经注册的中间件，创建的 `IApplicationBuilder` 对象与当前对象具有相同的状态。

具有如下定义的 `ApplicationBuilder` 类型是对 `IApplicationBuilder` 接口的默认实现。`ApplicationBuilder` 类型利用一个 `List<Func<RequestDelegate, RequestDelegate>>` 对象来保存注册的中间件，所以 `Use` 方法只需要将指定的中间件添加到这个列表中即可，而 `Build` 方法只需要逆序调用这些注册的中间件对应的 `Func<RequestDelegate, RequestDelegate>` 对象就能得到我们需要的 `RequestDelegate` 对象。值得注意的是，`Build` 方法会在委托链的尾部添加一个额外的中间件，该中间件会将响应状态码设置为 404，所以应用在默认情况下会回复一个 404 响应。

```
public class ApplicationBuilder : IApplicationBuilder
{
    private readonly IList<Func<RequestDelegate, RequestDelegate>> middlewares
        = new List<Func<RequestDelegate, RequestDelegate>>();

    public IDictionary<string, object> Properties { get; }
    public IServiceProvider ApplicationServices
    {
        get { return GetProperty<IServiceProvider>("application.Services"); }
        set { SetProperty<IServiceProvider>("application.Services", value); }
    }

    public IFeatureCollection ServerFeatures
    {
        get { return GetProperty<IFeatureCollection>("server.Features"); }
    }

    public ApplicationBuilder(IServiceProvider serviceProvider)
    {
```

```

        Properties = new Dictionary<string, object>();
        ApplicationServices = serviceProvider;
    }

    public ApplicationBuilder(IServiceCollection serviceProvider, object server)
        : this(serviceProvider)
        => SetProperty("server.Features", server);

    public IApplicationBuilder Use(Func<RequestDelegate, RequestDelegate> middleware)
    {
        middlewares.Add(middleware);
        return this;
    }

    public IApplicationBuilder New()
        => new ApplicationBuilder(this);

    public RequestDelegate Build()
    {
        RequestDelegate app = context =>
        {
            context.Response.StatusCode = 404;
            return Task.FromResult(0);
        };
        foreach (var component in middlewares.Reverse())
        {
            app = component(app);
        }
        return app;
    }

    private ApplicationBuilder(ApplicationBuilder builder)
    {
        this.Properties = new CopyOnWriteDictionary<string, object>(
            builder.Properties, StringComparer.Ordinal);
    }

    private T GetProperty<T>(string key)
    {
        object value;
        return Properties.TryGetValue(key, out value) ? (T)value : default(T);
    }

    private void SetProperty<T>(string key, T value)
    {
        Properties[key] = value;
    }
}

```

由上面的代码片段可以看出，不论是通过 `ApplicationServices` 属性返回的 `IService`

Provider 对象，还是通过 `ServerFeatures` 属性返回的 `IFeatureCollection` 对象，它们实际上都保存在通过 `Properties` 属性返回的字典对象上。`ApplicationBuilder` 具有两个公共构造函数重载，其中一个构造函数具有一个类型为 `Object` 的 `server` 参数，但这个参数并不是表示服务器，而是表示服务器提供的 `IFeatureCollection` 对象。`New` 方法直接调用私有构造函数创建一个新的 `ApplicationBuilder` 对象，属性字典的所有元素会复制到新创建的 `ApplicationBuilder` 对象中。

ASP.NET Core 框架使用的 `IApplicationBuilder` 对象是通过注册的 `IApplicationBuilderFactory` 服务创建的。如下面的代码片段所示，`IApplicationBuilderFactory` 接口具有唯一的 `CreateBuilder` 方法，它会根据提供的特性集合创建相应的 `IApplicationBuilder` 对象。具有如下定义的 `ApplicationBuilderFactory` 类型是对该接口的默认实现，前面介绍的 `ApplicationBuilder` 对象正是由它创建的。

```
public interface IApplicationBuilderFactory
{
    IApplicationBuilder CreateBuilder(IFeatureCollection serverFeatures);
}

public class ApplicationBuilderFactory : IApplicationBuilderFactory
{
    private readonly IServiceProvider serviceProvider;

    public ApplicationBuilderFactory(IServiceProvider serviceProvider)
        => serviceProvider = serviceProvider;

    public IApplicationBuilder CreateBuilder(IFeatureCollection serverFeatures)
        => new ApplicationBuilder(this.serviceProvider, serverFeatures);
}
```

13.3.2 弱类型中间件

虽然中间件最终体现为一个 `Func<RequestDelegate, RequestDelegate>` 对象，但是在大部分情况下我们总是倾向于将中间件定义成一个 `POCO` 类型。通过第 11 章的介绍可知，中间件类型的定义具有两种形式：一种是按照预定义的约定规则来定义中间件类型，即弱类型中间件；另一种则是直接实现 `IMiddleware` 接口，即强类型中间件。下面介绍基于约定的中间件类型的定义方式，这种方式定义的中间件类型需要采用如下约定。

- 中间件类型需要有一个有效的公共实例构造函数，该构造函数必须包含一个 `RequestDelegate` 类型的参数，当前中间件通过执行这个委托对象将请求分发给后续中间件进行处理。这个构造函数不仅可以包含任意其他参数，对参数 `RequestDelegate` 出现的位置也不做任何约束。
- 针对请求的处理实现在返回类型为 `Task` 的 `Invoke` 方法或者 `InvokeAsync` 方法中，该方法的第一个参数表示当前请求对应的 `HttpContext` 上下文，对于后续的参数，虽然约定并未对此做限制，但是由于这些参数最终是由依赖注入框架提供的，所以相应的服务注册必须存在。

如下所示的代码片段就是一个典型的按照约定定义的中间件类型。我们在构造函数中注入了一个必需的 `RequestDelegate` 对象和一个 `IFoo` 服务。在用于请求处理的 `InvokeAsync` 方法中，除了包含表示当前 `HttpContext` 上下文的参数，我们还注入了一个 `IBar` 服务，该方法在完成自身请求处理操作之后，通过构造函数中注入的 `RequestDelegate` 对象可以将请求分发给后续的中间件。

```
public class FoobarMiddleware
{
    private readonly RequestDelegate _next;
    private readonly IFoo _foo;

    public FoobarMiddleware(RequestDelegate next, IFoo foo)
    {
        next = next;
        _foo = foo;
    }

    public async Task InvokeAsync(HttpContext context, IBar bar)
    {
        ...
        await _next(context);
    }
}
```

采用上述方式定义的中间件最终是通过调用 `IApplicationBuilder` 接口如下所示的两个扩展方法进行注册的。当我们调用这两个方法时，除了指定具体的中间件类型，还可以传入一些必要的参数，它们将作为调用构造函数的输入参数。对于定义在中间件类型构造函数中的参数，如果有对应的服务注册，ASP.NET Core 框架在创建中间件实例时可以利用依赖注入框架来提供对应的参数，所以在注册中间件时是不需要提供构造函数的所有参数的。

```
public static class UseMiddlewareExtensions
{
    public static IApplicationBuilder UseMiddleware<TMiddleware>(
        this IApplicationBuilder app, params object[] args);
    public static IApplicationBuilder UseMiddleware(this IApplicationBuilder app,
        Type middleware, params object[] args);
}
```

由于 ASP.NET Core 应用的请求处理管道总是采用 `Func<RequestDelegate, RequestDelegate>` 对象来表示中间件，所以无论采用什么样的中间件定义方式，注册的中间件总是会转换成委托对象。那么上述两个扩展方法是如何实现这样的转换的？为了解决这个问题，我们采用极简的形式自行定义了第二个非泛型的 `UseMiddleware` 方法。

```
public static class UseMiddlewareExtensions
{
    private static readonly MethodInfo GetServiceMethod = typeof(IServiceProvider)
        .GetMethod("GetService", BindingFlags.Public | BindingFlags.Instance);
    public static IApplicationBuilder UseMiddleware(this IApplicationBuilder app,
        Type middlewareType, params object[] args)
    {

```

```

...
var invokeMethod = middlewareType
    .GetMethods(BindingFlags.Instance | BindingFlags.Public)
    .Where(it => it.Name == "InvokeAsync" || it.Name == "Invoke")
    .Single();
Func<RequestDelegate, RequestDelegate> middleware = next =>
{
    var arguments = (object[])Array.CreateInstance(typeof(object),
        args.Length + 1);
    arguments[0] = next;
    if (args.Length > 0)
    {
        Array.Copy(args, 0, arguments, 1, args.Length);
    }
    var instance = ActivatorUtilities.CreateInstance(app.ApplicationServices,
        middlewareType, arguments);
    var factory = CreateFactory(invokeMethod);
    return context => factory(instance, context, app.ApplicationServices);
};

return app.Use(middleware);
}

private static Func<object, HttpContext, IServiceProvider, Task>
CreateFactory(MethodInfo invokeMethod)
{
    var middleware = Expression.Parameter(typeof(object), "middleware");
    var httpContext = Expression.Parameter(typeof(HttpContext), "httpContext");
    var serviceProvider = Expression.Parameter(typeof(IServiceProvider),
        "serviceProvider");

    var parameters = invokeMethod.GetParameters();
    var arguments = new Expression[parameters.Length];
    arguments[0] = httpContext;
    for (int index = 1; index < parameters.Length; index++)
    {
        var parameterType = parameters[index].ParameterType;
        var type = Expression.Constant(parameterType, typeof(Type));
        var getService = Expression.Call(serviceProvider, GetServiceMethod, type);
        arguments[index] = Expression.Convert(getService, parameterType);
    }
    var converted = Expression.Convert(middleware, invokeMethod.DeclaringType);
    var body = Expression.Call(converted, invokeMethod, arguments);
    var lambda = Expression.Lambda<
        Func<object, HttpContext, IServiceProvider, Task>>(
        body, middleware, httpContext, serviceProvider);

    return lambda.Compile();
}
}

```

由于请求处理的具体实现定义在中间件类型的 `Invoke` 方法或者 `InvokeAsync` 方法上，所以注册这样一个中间件需要解决两个核心问题：其一，创建对应的中间件实例；其二，将针对中间件实例的 `Invoke` 方法或者 `InvokeAsync` 方法调用转换成 `Func<RequestDelegate, RequestDelegate>` 对象。由于存在依赖注入框架，所以第一个问题很好解决，从上面给出的代码片段可以看出，我们最终调用静态类型 `ActivatorUtilities` 的 `CreateInstance` 方法创建出中间件实例。

由于 ASP.NET Core 框架对中间件类型的 `Invoke` 方法和 `InvokeAsync` 方法的声明并没有严格限制，该方法返回类型为 `Task`，它的第一个参数为 `HttpContext` 上下文，所以针对该方法的调用比较烦琐。要调用某个方法，需要先传入匹配的参数列表，有了 `IServiceProvider` 对象的帮助，针对输入参数的初始化就显得非常容易。我们只需要从表示方法的 `MethodInfo` 对象中解析出方法的参数类型，就能够根据类型从 `IServiceProvider` 对象中得到对应的参数实例。

如果有表示目标方法的 `MethodInfo` 对象和与之匹配的输入参数列表，就可以采用反射的方式来调用对应的方法，但是反射并不是一种高效的手段，所以 ASP.NET Core 框架采用表达式树的方式来实现针对 `InvokeAsync` 方法或者 `Invoke` 方法的调用。基于表达式树针对中间件实例的 `InvokeAsync` 方法或者 `Invoke` 方法的调用实现在前面提供的 `CreateFactory` 方法中，由于实现逻辑并不复杂，所以不需要再对提供的代码做详细说明。

13.3.3 强类型中间件

通过调用 `IApplicationBuilder` 接口的 `UseMiddleware` 扩展方法注册的是一个按照约定规则定义的中间件类型，由于中间件实例是在应用初始化时创建的，这样的中间件实际上是一个与当前应用程序具有相同生命周期的 `Singleton` 对象。但有时我们希望中间件对象采用 `Scoped` 模式的生命周期，即要求中间件对象在开始处理请求时被创建，在完成请求处理后被回收释放。

如果需要后面这种类型的中间件，就需要让定义的中间件类型实现 `IMiddleware` 接口。如下面的代码片段所示，`IMiddleware` 接口定义了唯一的 `InvokeAsync` 方法，用来实现对请求的处理。对于实现该方法的中间件类型来说，它可以利用输入参数得到针对当前请求的 `HttpContext` 上下文，还可以得到用来向后续中间件分发请求的 `RequestDelegate` 对象。

```
public interface IMiddleware
{
    Task InvokeAsync(HttpContext context, RequestDelegate next);
}
```

实现了 `IMiddleware` 接口的中间件是通过依赖注入的形式提供的，所以在调用 `IApplicationBuilder` 接口的 `UseMiddleware` 扩展方法注册中间件类型之前需要做相应的服务注册。在一般情况下，我们只会在需要使用 `Scoped` 生命周期时才会采用这种方式来定义中间件，所以在进行服务注册时一般将生命周期模式设置为 `Scoped`，设置成 `Singleton` 模式也未尝不可，这就与按照约定规则定义的中间件没有本质区别。读者可能会有疑问，注册中间件服务时是否可以将生命周期模式设置为 `Transient`？实际上这与 `Scoped` 是没有区别的，因为中间件在同一个请求上下文中只会被创建一次。

对实现了 `IMiddleware` 接口的中间件的创建与释放是通过注册的 `IMiddlewareFactory` 服务来完成的。如下面的代码片段所示，`IMiddlewareFactory` 接口提供了如下两个方法：`Create` 方法会根据指定的中间件类型创建出对应的实例，`Release` 方法则负责释放指定的中间件对象。

```
public interface IMiddlewareFactory
{
    IMiddleware Create(Type middlewareType);
    void Release(IMiddleware middleware);
}
```

ASP.NET Core 提供如下所示的 `MiddlewareFactory` 类型作为 `IMiddlewareFactory` 接口的默认实现，上面提及的中间件针对依赖注入的创建方式就体现在该类型中。如下面的代码片段所示，`MiddlewareFactory` 直接利用指定的 `IServiceProvider` 对象根据指定的中间件类型来提供对应的实例。由于依赖注入框架自身具有针对提供服务实例的生命周期管理策略，所以 `MiddlewareFactory` 的 `Release` 方法不需要对提供的中间件实例做具体的释放操作。

```
public class MiddlewareFactory : IMiddlewareFactory
{
    private readonly IServiceProvider _serviceProvider;

    public MiddlewareFactory(IServiceProvider serviceProvider)
        => _serviceProvider = serviceProvider;
    public IMiddleware Create(Type middlewareType)
        => serviceProvider.GetRequiredService(this.serviceProvider, middlewareType)
        as IMiddleware;
    public void Release(IMiddleware middleware) {}
}
```

了解了作为中间件工厂的 `IMiddlewareFactory` 接口之后，下面介绍 `IApplicationBuilder` 用于注册中间件的 `UseMiddleware` 扩展方法是如何利用它来创建并释放中间件的，为此我们编写了如下这段简写的代码来模拟相关的实现。如下面的代码片段所示，如果注册的中间件类型实现了 `IMiddleware` 接口，`UseMiddleware` 方法会直接创建一个 `Func<RequestDelegate, RequestDelegate>` 对象作为注册的中间件。

```
public static class UseMiddlewareExtensions
{
    public static IApplicationBuilder UseMiddleware(this IApplicationBuilder app,
        Type middlewareType, params object[] args)
    {
        if (typeof(IMiddleware).IsAssignableFrom(middlewareType))
        {
            if (args.Length > 0)
            {
                throw new NotSupportedException(
                    "Types that implement IMiddleware do not support explicit arguments.");
            }
            app.Use(next =>
            {
                return async context =>
                {
```



```

        var middlewareFactory = context.RequestServices
            .GetRequiredService<IMiddlewareFactory>();
        var middleware = middlewareFactory.Create(middlewareType);
        try
        {
            await middleware.InvokeAsync(context, next);
        }
        finally
        {
            middlewareFactory.Release(middleware);
        }
    };
});
}
...
}
}

```

当作为中间件的委托对象被执行时，它会从当前 `HttpContext` 上下文的 `RequestServices` 属性中获取针对当前请求的 `IServiceProvider` 对象，并由它来提供 `IMiddlewareFactory` 对象。在利用 `IMiddlewareFactory` 对象根据注册的中间件类型创建出对应的中间件对象之后，中间件的 `InvokeAsync` 方法被调用。在当前及后续中间件针对当前请求的处理完成之后，`IMiddlewareFactory` 对象的 `Release` 方法被调用来释放由它创建的中间件。

`UseMiddleware` 方法之所以从当前 `HttpContext` 上下文的 `RequestServices` 属性获取 `IServiceProvider`，而不是直接使用 `IApplicationBuilder` 的 `ApplicationServices` 属性返回的 `IServiceProvider` 来创建 `IMiddlewareFactory` 对象，是出于生命周期方面的考虑。由于后者采用针对当前应用程序的生命周期模式，所以不论注册中间件类型采用的生命周期模式是 `Singleton` 还是 `Scoped`，提供的中间件实例都是一个 `Singleton` 对象，所以无法满足我们针对请求创建和释放中间件对象的初衷。

上面的代码片段还反映了一个细节：如果注册了一个实现了 `IMiddleware` 接口的中间件类型，我们是不允许指定任何参数的，一旦调用 `UseMiddleware` 方法时指定了参数，就会抛出一个 `NotSupportedException` 类型的异常。

13.3.4 注册中间件

在 ASP.NET Core 应用请求处理管道构建过程中，`IApplicationBuilder` 对象的作用就是收集我们注册的中间件，并最终根据注册的先后顺序创建一个代表中间件委托链的 `RequestDelegate` 对象。在一个具体的 ASP.NET Core 应用中，利用 `IApplicationBuilder` 对象进行中间件的注册主要体现为如下 3 种方式。

- 调用 `IWebHostBuilder` 的 `Configure` 方法。
- 调用注册 `Startup` 类型的 `Configure` 方法。
- 利用注册的 `IStartupFilter` 对象。

如下所示的 `IStartupFilter` 接口定义了唯一的 `Configure` 方法，它返回的 `Action<IApplication`

Builder>对象将用来注册所需的中间件。作为该方法唯一输入参数的 Action<IApplicationBuilder>对象，则用来完成后续的中间件注册工作。IStartupFilter 接口的 Configure 方法比 IStartup 的 Configure 方法先执行，所以可以利用前者注册一些前置或者后置的中间件。

```
public interface IStartupFilter
{
    Action<IApplicationBuilder> Configure(Action<IApplicationBuilder> next);
}
```

13.4 应用的承载

到目前为止，我们知道 ASP.NET Core 应用的请求处理管道是由一个 IServer 对象和 IHttpApplication 对象构成的。我们可以根据需要进行注册不同类型的服务器，但在默认情况下，IHttpApplication 是一个 HostingApplication 对象。一个 HostingApplication 对象由指定的 RequestDelegate 对象来完成所有的请求处理工作，而后者代表所有中间件按照注册的顺序串联而成的委托链。所有的这一切都被 GenericWebHostService 整合在一起，在对这个承载 Web 应用的服务做进一步介绍之前，下面先介绍与它相关的配置选项。

13.4.1 GenericWebHostServiceOptions

GenericWebHostService 这个承载服务的配置选项类型为 GenericWebHostServiceOptions。如下面的代码片段所示，这个内部类型有 3 个属性，其核心配置选项由 WebHostOptions 属性承载。GenericWebHostServiceOptions 类型的 ConfigureApplication 属性返回的 Action<IApplicationBuilder> 对象用来注册中间件，启动过程中针对中间件的注册最终都会转移到这个属性上。

```
internal class GenericWebHostServiceOptions
{
    public WebHostOptions WebHostOptions { get; set; }
    public Action<IApplicationBuilder> ConfigureApplication { get; set; }
    public AggregateException HostingStartupExceptions { get; set; }
}
```

第 11 章提出，可以利用一个外部程序集中定义的 IHostingStartup 实现类型来完成初始化任务，而 GenericWebHostServiceOptions 类型的 HostingStartupExceptions 属性返回的 AggregateException 对象就是对这些初始化任务执行过程中抛出异常的封装。一个 WebHostOptions 对象承载了与 IWebHost 相关的配置选项，虽然在基于 IHost/IHostBuilder 的承载系统中，IWebHost 接口作为宿主的作用已经不存在，但是 WebHostOptions 这个配置选项依然被保留下来。

```
public class WebHostOptions
{
    public string ApplicationName { get; set; }
    public string Environment { get; set; }
    public string ContentRootPath { get; set; }
    public string WebRoot { get; set; }
    public string StartupAssembly { get; set; }
    public bool PreventHostingStartup { get; set; }
    public IReadOnlyList<string> HostingStartupAssemblies { get; set; }
}
```

```

public IReadOnlyList<string>      HostingStartupExcludeAssemblies { get; set; }
public bool                      CaptureStartupErrors { get; set; }
public bool                      DetailedErrors { get; set; }
public TimeSpan                 ShutdownTimeout { get; set; }

public WebHostOptions() => ShutdownTimeout = TimeSpan.FromSeconds(5.0);
public WebHostOptions(IConfiguration configuration);
public WebHostOptions(IConfiguration configuration, string applicationNameFallback);
}

```

一个 `WebHostOptions` 对象可以根据一个 `IConfiguration` 对象来创建，当我们调用这个构造函数时，它会根据预定义的配置键从该 `IConfiguration` 对象中提取相应的值来初始化对应的属性。

```

public static class WebHostDefaults
{
    public static readonly string ApplicationKey           = "applicationName";
    public static readonly string StartupAssemblyKey     = "startupAssembly";
    public static readonly string DetailedErrorsKey     = "detailedErrors";
    public static readonly string EnvironmentKey        = "environment";
    public static readonly string WebRootKey            = "webroot";
    public static readonly string CaptureStartupErrorsKey = "captureStartupErrors";
    public static readonly string ServerUrlsKey         = "urls";
    public static readonly string ContentRootKey        = "contentRoot";
    public static readonly string PreferHostingUrlsKey  = "preferHostingUrls";
    public static readonly string PreventHostingStartupKey = "preventHostingStartup";
    public static readonly string ShutdownTimeoutKey    = "shutdownTimeoutSeconds";

    public static readonly string HostingStartupAssembliesKey
        = "hostingStartupAssemblies";
    public static readonly string HostingStartupExcludeAssembliesKey
        = "hostingStartupExcludeAssemblies";
}

```

这些预定义的配置键作为静态只读字段被定义在静态类 `WebHostDefaults` 中，其中大部分在第 11 章已有相关介绍，本节只对此进行总结。`WebHostOptions` 属性列表如表 13-4 所示。值得注意的是，对于布尔类型的属性值（如 `PreventHostingStartup` 和 `CaptureStartupErrors`），配置项的值“True”（不区分大小写）和“1”将转换为 `True`，其他的值将转换成 `False`。这个将配置项的值转换成布尔值的逻辑实现在 `WebHostUtilities` 的静态方法 `ParseBool` 中，如果我们有类似的需求可以直接调用这个方法。

表 13-4 WebHostOptions 属性列表

属 性	配 置 键	说 明
ApplicationName	applicationName	应用名称。如果调用 <code>IWebHostBuilder</code> 接口的 <code>Configure</code> 方法注册中间件，那么提供的 <code>Action<IApplicationBuilder></code> 对象指向的目标方法所在的程序集名称将作为应用名称。如果调用 <code>IWebHostBuilder</code> 接口的 <code>UseStartup</code> 扩展方法，指定的 <code>Startup</code> 类型所在的程序集名称会作为应用名称
Environment	environment	应用当前的部署环境。如果没有显示指定，默认的环境名称为 <code>Production</code>

续表

属 性	配 置 键	说 明
ContentRootPath	contentRoot	存放静态内容文件的根目录。如果未做显式设置，默认为当前程序域的基础目录，对应 AppDomain 的 BaseDirectory 属性，静态类 AppContext 的 BaseDirectory 属性返回的也是这个目录
WebRoot	webroot	存放静态 Web 资源文件的根目录。如果未做显式设置，并且 ContentRootPath 目录下存在一个名为 wwwroot 的子目录，那么该目录将作为 Web 资源文件的根目录
StartupAssembly	startupAssembly	注册的 Startup 类型所在的程序集名称。如果调用 IWebHostBuilder 接口的 UseStartup 扩展方法，指定的 Startup 类型所在的程序集名称会作为该属性的值
PreventHostingStartup	preventHostingStartup	是否允许执行其他程序集中的初始化程序。如果这个开关并没有显式关闭，就可以在一个单独的程序集中利用 HostingStartupAttribute 特性注册一个实现了 IHostingStartup 接口的类型，它可以在应用启动时执行一些初始化操作
HostingStartupAssemblies	hostingStartupAssemblies	承载初始化程序的程序集列表，配置中的程序集名称之间采用分号分隔。ApplicationName 属性代表的程序集名称默认被添加到这个列表中
HostingStartupExcludeAssemblies	hostingStartupExcludeAssemblies	HostingStartupAssemblies 属性表示初始化程序的程序集列表中需要被排除的程序集
CaptureStartupErrors	captureStartupErrors	是否需要捕捉应用启动过程中出现的未处理异常。如果这个属性被显式设置为 True，出现的未处理异常并不会阻止应用的正常启动，但是这样的应用在接收到请求之后会返回一个状态码为 500 的响应
DetailedErrors	detailedErrors	如果 CaptureStartupErrors 属性被显式设置为 True，该属性表示是否需要在响应消息中输出详细的错误信息
ShutdownTimeout	shutdownTimeoutSeconds	应用关闭的超时时限，默认时限为 5 秒

13.4.2 GenericWebHostService

从如下所示的代码片段可以看出，GenericWebHostService 的构造函数中会注入一系列的依赖服务或者对象，其中包括用来提供配置选项的 IOptions<GenericWebHostServiceOptions>对象、作为管道“龙头”的服务器、用来创建 ILogger 对象的 ILoggerFactory 对象、用来发送相应诊断事件的 DiagnosticListener 对象、用来创建 HttpContext 上下文的 IHttpContextFactory 对象、用来创建 IApplicationBuilder 对象的 IApplicationBuilderFactory 对象、注册的所有 IStartupFilter 对象、承载当前应用配置的 IConfiguration 对象和代表当前承载环境的 IWebHostEnvironment 对象。在 GenericWebHostService 构造函数中注入的对象或者由它们创建的对象（如由 ILoggerFactory 对象创建的 ILogger 对象）最终会存储在对应的属性上。

```
internal class GenericWebHostService : IHostedService
{
    public GenericWebHostServiceOptions Options { get; }
    public IServer Server { get; }
```

```

public ILogger                Logger { get; }
public ILogger                LifetimeLogger { get; }
public DiagnosticListener     DiagnosticListener { get; }
public IHttpContextFactory   HttpContextFactory { get; }
public IApplicationBuilder   ApplicationBuilderFactory { get; }
public IEnumerable<IStartupFilter> StartupFilters { get; }
public IConfiguration         Configuration { get; }
public IWebHostEnvironment   HostingEnvironment { get; }

public GenericWebHostService(IOptions<GenericWebHostServiceOptions> options,
    IServer server, ILoggerFactory loggerFactory,
    DiagnosticListener diagnosticListener, IHttpContextFactory httpContextFactory,
    IApplicationBuilder applicationBuilderFactory,
    IEnumerable<IStartupFilter> startupFilters, IConfiguration configuration,
    IWebHostEnvironment hostingEnvironment);

public Task StartAsync(CancellationToken cancellationToken);
public Task StopAsync(CancellationToken cancellationToken);
}

```

由于 ASP.NET Core 应用是由 `GenericWebHostService` 服务承载的，所以启动应用程序本质上就是启动这个承载服务。承载 `GenericWebHostService` 在启动过程中的处理流程基本上体现在如下所示的 `StartAsync` 方法中，该方法中刻意省略了一些细枝末节的实现，如输入验证、异常处理、诊断日志事件的发送等。

```

internal class GenericWebHostService : IHostedService
{
    public Task StartAsync(CancellationToken cancellationToken)
    {
        //1. 设置监听地址
        var serverAddressesFeature = Server.Features?.Get<IServerAddressesFeature>();
        var addresses = serverAddressesFeature?.Addresses;
        if (addresses != null && !addresses.IsReadOnly && addresses.Count == 0)
        {
            var urls = Configuration[WebHostDefaults.ServerUrlsKey];
            if (!string.IsNullOrEmpty(urls))
            {
                serverAddressesFeature.PreferHostingUrls = WebHostUtilities.ParseBool(
                    Configuration, WebHostDefaults.PreferHostingUrlsKey);

                foreach (var value in urls.Split(new[] { ';' },
                    StringSplitOptions.RemoveEmptyEntries))
                {
                    addresses.Add(value);
                }
            }
        }

        //2. 构建中间件管道
        var builder = ApplicationBuilderFactory.CreateBuilder(Server.Features);
    }
}

```

```

        Action<IApplicationBuilder> configure = Options.ConfigureApplication;
        foreach (var filter in StartupFilters.Reverse())
        {
            configure = filter.Configure(configure);
        }
        configure(builder);
        var handler = builder.Build();

        //3. 创建 HostingApplication 对象
        var application = new HostingApplication(handler, Logger, DiagnosticListener,
            HttpContextFactory);

        //4. 启动服务器
        return Server.StartAsync(application, cancellationToken);
    }
}

```

我们将实现在 `GenericWebHostService` 类型的 `StartAsync` 方法中用来启动应用程序的流程划分为如下 4 个步骤。

- **设置监听地址：**服务器的监听地址是通过 `IServerAddressesFeature` 接口表示的特性来承载的，所以需要将配置提供的监听地址列表和相关的 `PreferHostingUrls` 选项（表示是否优先使用承载系统提供地址）转移到该特性中。
- **构建中间件管道：**通过调用 `IWebHostBuilder` 对象和注册的 `Startup` 类型的 `Configure` 方法针对中间件的注册会转换成一个 `Action<IApplicationBuilder>` 对象，并复制给配置选项 `GenericWebHostServiceOptions` 的 `ConfigureApplication` 属性。`GenericWebHostService` 承载服务会利用注册的 `IApplicationBuilderFactory` 工厂创建出对应的 `IApplicationBuilder` 对象，并将该对象作为参数调用这个 `Action<IApplicationBuilder>` 对象就能将注册的中间件转移到 `IApplicationBuilder` 对象上。但在此之前，注册 `IStartupFilter` 对象的 `Configure` 方法会优先被调用，`IStartupFilter` 对象针对前置中间件的注册就体现在这里。代表注册中间件管道的 `RequestDelegate` 对象最终通过调用 `IApplicationBuilder` 对象的 `Build` 方法返回。
- **创建 `HostingApplication` 对象：**在得到代表中间件管道的 `RequestDelegate` 之后，`GenericWebHostService` 对象进一步利用它创建出 `HostingApplication` 对象，该对象对于服务器来说就是用来处理由它接收请求的应用程序。
- **启动服务器：**将创建出的 `HostingApplication` 对象作为参数调用作为服务器的 `IServer` 对象的 `StartAsync` 方法后，服务器随之被启动。此后，服务器绑定到指定的地址监听抵达的请求，并为接收的请求创建出对应的 `HttpContext` 上下文，后续中间件将在这个上下文中完成各自对请求的处理任务。请求处理结束之后，生成的响应最终通过服务器回复给客户端。

关闭 `GenericWebHostService` 服务之后，只需要按照如下方式关闭服务器即可。除此之外，`StopAsync` 方法还会利用 `EventSource` 的形式发送相应的事件，我们在前面针对诊断日志的演示可以体验此功能。

```
internal class GenericWebHostService : IHostedService
{
    public async Task StopAsync(CancellationToken cancellationToken)
        => Server.StopAsync(cancellationToken);
}
```

13.4.3 GenericWebHostBuilder

要承载一个 ASP.NET Core 应用，只需要将 `GenericWebHostService` 服务注册到承载系统中即可。但 `GenericWebHostService` 服务具有针对其他一系列服务的依赖，所以在注册该承载服务之前需要先完成对这些依赖服务的注册。针对 `GenericWebHostService` 及其依赖服务的注册是借助 `GenericWebHostBuilder` 对象来完成的。

在传统的基于 `IWebHost/IWebHostBuilder` 的承载系统中，`IWebHost` 对象表示承载 Web 应用的宿主，它由对应的 `IWebHostBuilder` 对象构建而成，`IWebHostBuilder` 针对 `IWebHost` 对象的构建体现在它的 `Build` 方法上。由于通过该方法构建 `IWebHost` 对象是利用依赖注入框架提供的，所以 `IWebHostBuilder` 接口定义了两个 `ConfigureServices` 方法重载来注册这些依赖服务。如果注册的服务与通过 `WebHostBuilderContext` 对象表示的承载上下文（承载环境和配置）无关，我们一般调用第一个 `ConfigureServices` 方法重载，第二个方法可以帮助我们完成基于承载上下文的服务注册，我们也可以根据当前承载环境和提供的配置来动态地注册所需的服务。

```
public interface IWebHostBuilder
{
    IWebHost Build();

    string GetSetting(string key);
    IWebHostBuilder UseSetting(string key, string value);
    IWebHostBuilder ConfigureAppConfiguration(Action<WebHostBuilderContext,
        IConfigurationBuilder> configureDelegate);

    IWebHostBuilder ConfigureServices(Action<IServiceCollection> configureServices);
    IWebHostBuilder ConfigureServices(
        Action<WebHostBuilderContext, IServiceCollection> configureServices);
}
```

在基于 `IWebHost/IWebHostBuilder` 的承载系统中，`WebHostBuilder` 是对 `IWebHostBuilder` 接口的默认实现。如果采用基于 `IHost/IHostBuilder` 的承载系统，默认实现的 `IWebHostBuilder` 类型为 `GenericWebHostBuilder`。这个内部类型除了实现 `IWebHostBuilder` 接口，还实现了如下所示的两个内部接口（`ISupportsStartup` 和 `ISupportsUseDefaultServiceProvider`）。前面介绍 `Startup` 时提到过 `ISupportsStartup` 接口，它定义了一个用于注册中间件的 `Configure` 方法和一个用来注册 `Startup` 类型的 `UseStartup` 方法。`ISupportsUseDefaultServiceProvider` 接口则定义了唯一的 `UseDefaultServiceProvider` 方法，该方法用来对默认使用的依赖注入容器进行设置。

```
internal interface ISupportsStartup
{
    IWebHostBuilder Configure(
        Action<WebHostBuilderContext, IApplicationBuilder> configure);
}
```

```

    IWebHostBuilder UseStartup(Type startupType);
}

internal interface ISupportsUseDefaultServiceProvider
{
    IWebHostBuilder UseDefaultServiceProvider(
        Action<WebHostBuilderContext, ServiceProviderOptions> configure);
}

```

服务注册

下面通过简单的代码来模拟 `GenericWebHostBuilder` 针对 `IWebHostBuilder` 接口的实现。首先介绍用来注册依赖服务的 `ConfigureServices` 方法的实现。如下面的代码片段所示，`GenericWebHostBuilder` 实际上是对一个 `IHostBuilder` 对象的封装，针对依赖服务的注册是通过调用 `IHostBuilder` 接口的 `ConfigureServices` 方法实现的。

```

internal class GenericWebHostBuilder :
    IWebHostBuilder,
    ISupportsStartup,
    ISupportsUseDefaultServiceProvider
{
    private readonly IHostBuilder _builder;

    public GenericWebHostBuilder(IHostBuilder builder)
    {
        builder = builder;
        ...
    }

    public IWebHostBuilder ConfigureServices(Action<IServiceCollection> configureServices)
        => ConfigureServices( (_, services) => configureServices(services));

    public IWebHostBuilder ConfigureServices(
        Action<WebHostBuilderContext, IServiceCollection> configureServices)
    {
        _builder.ConfigureServices((context, services)
            => configureServices(GetWebHostBuilderContext(context), services));
        return this;
    }

    private WebHostBuilderContext GetWebHostBuilderContext(HostBuilderContext context)
    {
        if (!context.Properties.TryGetValue(typeof(WebHostBuilderContext), out var value))
        {
            var options = new WebHostOptions(context.Configuration,
                Assembly.GetEntryAssembly()?.GetName().Name);
            var webHostBuilderContext = new WebHostBuilderContext
            {
                Configuration = context.Configuration,
                HostingEnvironment = new HostingEnvironment(),
            };
        }
    }
}

```



```

    };
    webHostBuilderContext.HostingEnvironment
        .Initialize(context.HostingEnvironment.ContentRootPath, options);
    context.Properties[typeof(WebHostBuilderContext)] = webHostBuilderContext;
    context.Properties[typeof(WebHostOptions)] = options;
    return webHostBuilderContext;
}

var webHostContext = (WebHostBuilderContext)value;
webHostContext.Configuration = context.Configuration;
return webHostContext;
}
}
}

```

IHostBuilder 接口的 ConfigureServices 方法提供针对当前承载上下文的服务注册，通过 HostBuilderContext 对象表示的承载上下文包含两个元素，分别是表示配置的 IConfiguration 对象和表示承载环境的 IHostEnvironment 对象。而 ASP.NET Core 应用下的承载上下文是通过 WebHostBuilderContext 对象表示的，两个上下文之间的不同之处体现在针对承载环境的描述上，WebHostBuilderContext 上下文中的承载环境是通过 IWebHostEnvironment 对象表示的。GenericWebHostBuilder 在调用 IHostBuilder 对象的 ConfigureServices 方法注册依赖服务时，需要调用 GetWebHostBuilderContext 方法将提供的 WebHostBuilderContext 上下文转换成 HostBuilderContext 类型。

GenericWebHostBuilder 对象在构建时会以如下方式调用 ConfigureServices 方法注册一系列默认的依赖服务，其中包括表示承载环境的 IWebHostEnvironment 服务、用来发送诊断日志事件的 DiagnosticSource 服务和 DiagnosticListener 服务（它们返回同一个服务实例）、用来创建 HttpContext 上下文的 IHttpContextFactory 工厂、用来创建中间件的 IMiddlewareFactory 工厂、用来创建 IApplicationBuilder 对象的 IApplicationBuilderFactory 工厂等。除此之外，GenericWebHostBuilder 构造函数中还完成了针对 GenericWebHostServiceOptions 配置选项的设置，承载 ASP.NET Core 应用的 GenericWebHostService 服务也是在这里注册的。

```

internal class GenericWebHostBuilder :
    IWebHostBuilder,
    ISupportsStartup,
    ISupportsUseDefaultServiceProvider
{
    private readonly IHostBuilder _builder;
    private AggregateException hostingStartupErrors;

    public GenericWebHostBuilder(IHostBuilder builder)
    {
        _builder = builder;
        _builder.ConfigureServices((context, services)=>
        {
            var webHostBuilderContext = GetWebHostBuilderContext(context);
            services.AddSingleton(webHostBuilderContext.HostingEnvironment);
            services.AddHostedService<GenericWebHostService>();
        });
    }
}

```

```

DiagnosticListener instance = new DiagnosticListener("Microsoft.AspNetCore");
services.TryAddSingleton(instance);
services.TryAddSingleton<DiagnosticSource>(instance);
services.TryAddSingleton<IHttpContextFactory, DefaultHttpContextFactory>();
services.TryAddScoped<IMiddlewareFactory, MiddlewareFactory>();
services.TryAddSingleton
    <IApplicationBuilderFactory, ApplicationBuilderFactory>();

var webHostOptions = (WebHostOptions)context
    .Properties[typeof(WebHostOptions)];
services.Configure<GenericWebHostServiceOptions>(options=>
{
    options.WebHostOptions = webHostOptions;
    options.HostingStartupExceptions = _hostingStartupErrors;
});
});
...
}
}

```

配置的读写

除了两个 `ConfigureServices` 方法重载, `IWebHostBuilder` 接口的其他方法均与配置有关。基于 `IHost/IHostBuilder` 的承载系统涉及两种类型的配置: 一种是在服务承载过程中供作为宿主的 `IHost` 对象使用的配置, 另一种是供承载的服务或者应用消费的配置, 前者是后者的子集。这两种类型的配置分别由 `IHostBuilder` 接口的 `ConfigureHostConfiguration` 方法和 `ConfigureAppConfiguration` 方法进行设置, `GenericWebHostBuilder` 针对配置的设置最终会利用这两个方法来完成。

`GenericWebHostBuilder` 提供的配置体现在它的字段 `_config` 上, 以键值对形式设置和读取配置的 `UseSetting` 方法与 `GetSetting` 方法操作的是 `_config` 字段表示的 `IConfiguration` 对象。静态 `Host` 类型的 `CreateDefaultBuilder` 方法创建的 `HostBuilder` 对象会默认将前缀为“DOTNET_”的环境变量作为配置源, ASP.NET Core 应用则选择将前缀为“ASPNETCORE_”的环境变量作为配置源, 这一点体现在如下所示的代码片段中。

```

internal class GenericWebHostBuilder :
    IWebHostBuilder,
    ISupportsStartup,
    ISupportsUseDefaultServiceProvider
{
    private readonly IHostBuilder    _builder;
    private readonly IConfiguration   config;

    public GenericWebHostBuilder(IHostBuilder builder)
    {
        _builder = builder;
        config = new ConfigurationBuilder()
            .AddEnvironmentVariables(prefix: "ASPNETCORE_")

```

```

        .Build();
        _builder.ConfigureHostConfiguration(config => config.AddConfiguration(_config));
        ...
    }
    public string GetSetting(string key) => _config[key];

    public IWebHostBuilder UseSetting(string key, string value)
    {
        config[key] = value;
        return this;
    }
}

```

如上面的代码片段所示，`GenericWebHostBuilder` 对象在构造过程中会创建一个 `ConfigurationBuilder` 对象，并将前缀为“ASPNETCORE_”的环境变量作为配置源。在利用 `ConfigurationBuilder` 对象创建 `IConfiguration` 对象之后，该对象体现的配置通过调用 `IHostBuilder` 对象的 `ConfigureHostConfiguration` 方法被合并到承载系统的配置中。由于 `IHostBuilder` 接口和 `IWebHostBuilder` 接口的 `ConfigureAppConfiguration` 方法具有相同的目的，所以 `GenericWebHostBuilder` 类型的 `ConfigureAppConfiguration` 方法直接调用 `IHostBuilder` 的同名方法。

```

internal class GenericWebHostBuilder :
    IWebHostBuilder,
    ISupportsStartup,
    ISupportsUseDefaultServiceProvider
{
    private readonly IHostBuilder _builder;

    public IWebHostBuilder ConfigureAppConfiguration(
        Action<WebHostBuilderContext, IConfigurationBuilder> configureDelegate)
    {
        builder.ConfigureAppConfiguration((context, builder)
            => configureDelegate(GetWebHostBuilderContext(context), builder));
        return this;
    }
}

```

默认依赖注入框架配置

原生的依赖注入框架被直接整合到 ASP.NET Core 应用中，源于 `GenericWebHostBuilder` 类型 `ISupportsUseDefaultServiceProvider` 接口的实现。如下面的代码片段所示，在实现的 `UseDefaultServiceProvider` 方法中，`GenericWebHostBuilder` 会根据 `ServiceProviderOptions` 对象承载的配置选项完成对 `DefaultServiceProviderFactory` 工厂的注册。

```

internal class GenericWebHostBuilder :
    IWebHostBuilder,
    ISupportsStartup,
    ISupportsUseDefaultServiceProvider
{

```

```

public IWebHostBuilder UseDefaultServiceProvider(
    Action<WebHostBuilderContext, ServiceProviderOptions> configure)
{
    builder.UseServiceProviderFactory(context =>
    {
        var webHostBuilderContext = GetWebHostBuilderContext(context);
        var options = new ServiceProviderOptions();
        configure(webHostBuilderContext, options);
        return new DefaultServiceProviderFactory(options);
    });

    return this;
}
}

```

Startup

在大部分应用开发场景下，通常将应用启动时需要完成的初始化操作定义在注册的 `Startup` 中，按照约定定义的 `Startup` 类型旨在完成如下 3 个任务。

- 利用 `Configure` 方法或者 `Configure{EnvironmentName}` 方法注册中间件。
- 利用 `ConfigureServices` 方法或者 `Configure{EnvironmentName}Services` 方法注册依赖服务。
- 利用 `ConfigureContainer` 方法或者 `Configure{EnvironmentName}Container` 方法对第三方依赖注入容器做相关设置。

上述 3 个针对 `Startup` 的设置最终都需要应用到基于 `IHost/IHostBuilder` 的承载系统上。由于 `Startup` 类型是注册到 `GenericWebHostBuilder` 对象上的，而 `GenericWebHostBuilder` 对象本质上是对 `IHostBuilder` 对象的封装，这些设置可以借助这个被封装的 `IHostBuilder` 对象被应用到承载系统上，具体的实现体现在如下几点。

- 将针对中间件的注册转移到 `GenericWebHostServiceOptions` 这个配置选项的 `ConfigureApplication` 属性上。
- 调用 `IHostBuilder` 对象的 `ConfigureServices` 方法来完成真正的服务注册。
- 调用 `IHostBuilder` 对象的 `ConfigureContainer<TContainerBuilder>` 方法完成对依赖注入容器的设置。

上述 3 个在启动过程执行的初始化操作由 3 个对应的 `Builder` 对象（`ConfigureBuilder`、`ConfigureServicesBuilder` 和 `ConfigureContainerBuilder`）辅助完成，其中 `Startup` 类型的 `Configure` 方法或者 `Configure{EnvironmentName}` 方法对应如下所示的 `ConfigureBuilder` 类型。`ConfigureBuilder` 对象由 `Configure` 方法或者 `Configure{EnvironmentName}` 方法对应的 `MethodInfo` 对象创建而成，最终赋值给 `GenericWebHostServiceOptions` 配置选项 `ConfigureApplication` 属性的就是这个委托对象。如下所示的代码片段是 `ConfigureBuilder` 类型简化后的定义。

```

public class ConfigureBuilder
{
    public MethodInfo MethodInfo { get; }
    public ConfigureBuilder(MethodInfo configure)

```

```

=> MethodInfo = configure;

public Action<IApplicationBuilder> Build(object instance)
=> builder => Invoke(instance, builder);

private void Invoke(object instance, IApplicationBuilder builder)
{
    using (var scope = builder.ApplicationServices.CreateScope())
    {
        var serviceProvider = scope.ServiceProvider;
        var parameterInfos = MethodInfo.GetParameters();
        var parameters = new object[parameterInfos.Length];
        for (var index = 0; index < parameterInfos.Length; index++)
        {
            var parameterInfo = parameterInfos[index];
            parameters[index] =
                parameterInfo.ParameterType == typeof(IApplicationBuilder)
                ? builder
                : serviceProvider.GetRequiredService(parameterInfo.ParameterType);
        }
        MethodInfo.InvokeWithoutWrappingExceptions(instance, parameters);
    }
}
}
}

```

如下所示的 `ConfigureServicesBuilder` 和 `ConfigureContainerBuilder` 类型是简化后的版本，前者对应 `Startup` 类型的 `ConfigureServices/Configure{EnvironmentName}Services` 方法，后者对应 `ConfigureContainer` 方法或者 `Configure{EnvironmentName}Container` 方法。针对对应方法的调用会反映在 `Build` 方法返回的委托对象上。

```

public class ConfigureServicesBuilder
{
    public MethodInfo MethodInfo { get; }
    public ConfigureServicesBuilder2(MethodInfo configureServices)
        => MethodInfo = configureServices;
    public Func<IServiceCollection, IServiceProvider> Build(object instance)
        => services => Invoke(instance, services);
    private IServiceProvider Invoke(object instance, IServiceCollection services)
        => MethodInfo.InvokeWithoutWrappingExceptions(instance, new object[] { services })
        as IServiceProvider;
}

public class ConfigureContainerBuilder
{
    public MethodInfo MethodInfo { get; }
    public ConfigureContainerBuilder(MethodInfo configureContainerMethod)
        => MethodInfo = configureContainerMethod;
    public Action<object> Build(object instance) => container
        => Invoke(instance, container);
    private void Invoke(object instance, object container)

```

```

=> MethodInfo.InvokeWithoutWrappingExceptions(instance,
new object[] { container });
}

```

通过第 11 章的介绍可知, `Startup` 类型的构造函数中是可以注入依赖服务的, 但是可以在这里注入的依赖服务仅限于组成当前承载上下文的两个元素, 即表示承载环境的 `IHostEnvironment` 对象或者 `IWebHostEnvironment` 对象和表示配置的 `IConfiguration` 对象。这一个特性是如下这个特殊的 `IServiceProvider` 实现类型决定的。

```

internal class GenericWebHostBuilder :
    IWebHostBuilder,
    ISupportsStartup,
    ISupportsUseDefaultServiceProvider
{
    private class HostServiceProvider : IServiceProvider
    {
        private readonly WebHostBuilderContext context;
        public HostServiceProvider(WebHostBuilderContext context)
        {
            context = context;
        }

        public object GetService(Type serviceType)
        {
            if (serviceType == typeof(IWebHostEnvironment)
                || serviceType == typeof(IHostEnvironment))
            {
                return context.HostingEnvironment;
            }
            if (serviceType == typeof(IConfiguration))
            {
                return context.Configuration;
            }
            return null;
        }
    }
}

```

如下所示的代码片段是 `GenericWebHostBuilder` 的 `UseStartup` 方法简化版本的定义。可以看出, `Startup` 类型是通过调用 `IHostBuilder` 对象的 `ConfigureServices` 方法来注册的。如下面的代码片段所示, `GenericWebHostBuilder` 对象会根据指定 `Startup` 类型创建出 3 个对应的 `Builder` 对象, 然后利用上面的 `HostServiceProvider` 创建出 `Startup` 对象, 并将该对象作为参数调用对应的 3 个 `Builder` 对象的 `Build` 方法构建的委托对象完成对应的初始化任务。

```

internal class GenericWebHostBuilder :
    IWebHostBuilder,
    ISupportsStartup,
    ISupportsUseDefaultServiceProvider
{
    private readonly IHostBuilder _builder;

```

```

private readonly object startupKey = new object();

public IWebHostBuilder UseStartup(Type startupType)
{
    _builder.Properties["UseStartup.StartupType"] = startupType;
    builder.ConfigureServices((context, services) =>
    {
        if (_builder.Properties.TryGetValue("UseStartup.StartupType",
            out var cachedType) && (Type)cachedType == startupType)
        {
            UseStartup(startupType, context, services);
        }
    });

    return this;
}

private void UseStartup(Type startupType, HostBuilderContext context,
    IServiceCollection services)
{
    var webHostBuilderContext = GetWebHostBuilderContext(context);
    var webHostOptions = (WebHostOptions)context.Properties[typeof(WebHostOptions)];

    ExceptionDispatchInfo startupError = null;
    object instance = null;
    ConfigureBuilder configureBuilder = null;

    try
    {
        instance = ActivatorUtilities.CreateInstance(
            new HostServiceProvider(webHostBuilderContext), startupType);
        context.Properties[ startupKey ] = instance;
        var environmentName = context.HostingEnvironment.EnvironmentName;
        BindingFlags bindingFlags = BindingFlags.Public | BindingFlags.Instance;

        //ConfigureServices
        var configureServicesMethod = startupType.GetMethod(
            $"Configure{environmentName}Services", bindingFlags)
            ?? startupType.GetMethod("ConfigureServices", bindingFlags);
        if (configureServicesMethod != null)
        {
            var configureServicesBuilder =
                new ConfigureServicesBuilder(configureServicesMethod);
            var configureServices = configureServicesBuilder.Build(instance);
            configureServices(services);
        }

        //ConfigureContainer
        var configureContainerMethod = startupType
            .GetMethod($"Configure{environmentName}Container", bindingFlags)

```

```

        ?? startupType.GetMethod("ConfigureContainer", bindingFlags);
    if (configureContainerMethod != null)
    {
        var configureContainerBuilder =
            new ConfigureBuilder(configureServicesMethod);
        builder.Properties[typeof(ConfigureContainerBuilder)]
            = configureContainerBuilder;
        var containerType = configureContainerBuilder.MethodInfo
            .GetParameters()[0].ParameterType;
        var actionType = typeof(Action<,>)
            .MakeGenericType(typeof(HostBuilderContext), containerType);
        var configure = GetType().GetMethod(nameof(ConfigureContainer),
            BindingFlags.NonPublic | BindingFlags.Instance)
            .MakeGenericMethod(containerType)
            .CreateDelegate(actionType, this);

        //IHostBuilder.ConfigureContainer<TContainerBuilder>(
        //Action<HostBuilderContext, TContainerBuilder> configureDelegate)
        typeof(IHostBuilder).GetMethods().First(m => m.Name ==
            nameof(IHostBuilder.ConfigureContainer))
            .MakeGenericMethod(containerType)
            .Invoke(_builder, BindingFlags.DoNotWrapExceptions, null,
                new object[] { configure }, null);
    }

    var configureMethod = startupType.GetMethod($"Configure{environmentName}",
        bindingFlags)
        ?? startupType.GetMethod("Configure", bindingFlags);
    configureBuilder = new ConfigureBuilder(configureMethod);
}
catch (Exception ex) when (webHostOptions.CaptureStartupErrors)
{
    startupError = ExceptionDispatchInfo.Capture(ex);
}

//Configure
services.Configure<GenericWebHostServiceOptions>(options =>
{
    options.ConfigureApplication = app =>
    {
        startupError?.Throw();
        if (instance != null && configureBuilder != null)
        {
            configureBuilder.Build(instance)(app);
        }
    };
});
});
}

//用于创建 IHostBuilder.ConfigureContainer<TContainerBuilder>(

```



```
//Action<HostBuilderContext, TContainerBuilder> configureDelegate)方法的参数
private void ConfigureContainer<TContainer>(HostBuilderContext context,
    TContainer container)
{
    var instance = context.Properties[_startupKey];
    var builder = (ConfigureContainerBuilder)context
        .Properties[typeof(ConfigureContainerBuilder)];
    builder.Build(instance)(container);
}
}
```

除了上面的 `UseStartup` 方法，`GenericWebHostBuilder` 还实现了 `ISupportsStartup` 接口的 `Configure` 方法。如下面的代码片段所示，该方法会将指定的用于注册中间件的 `Action<WebHostBuilderContext, IApplicationBuilder>` 复制给作为配置选项的 `GenericWebHostServiceOptions` 对象的 `ConfigureApplication` 属性。由于注册 `Startup` 类型的目的也是通过设置 `GenericWebHostServiceOptions` 对象的 `ConfigureApplication` 属性来注册中间件，如果在调用了 `Configure` 方法时又注册了一个 `Startup` 类型，系统会采用“后来居上”的原则。

```
internal class GenericWebHostBuilder :
    IWebHostBuilder,
    ISupportsStartup,
    ISupportsUseDefaultServiceProvider
{
    public IWebHostBuilder Configure(
        Action<WebHostBuilderContext, IApplicationBuilder> configure)
    {
        _builder.ConfigureServices((context, services) =>
        {
            services.Configure<GenericWebHostServiceOptions>(options =>
            {
                var webhostBuilderContext = GetWebHostBuilderContext(context);
                options.ConfigureApplication =
                    app => configure(webhostBuilderContext, app);
            });
        });
        return this;
    }
}
```

除了直接调用 `UseStartup` 方法注册一个 `Startup` 类型，还可以利用配置注册 `Startup` 类型所在的程序集。`GenericWebHostBuilder` 对象在初始化过程中会按照约定的规则定位和加载 `Startup` 类型。通过第 11 章的介绍可知，`GenericWebHostBuilder` 对象会按照如下顺序从指定的程序集类型列表中筛选 `Startup` 类型。

- `Startup{EnvironmentName}`（全名匹配）。
- `Startup`（全名匹配）。
- `{StartupAssembly}.Startup{EnvironmentName}`（全名匹配）。
- `{StartupAssembly}.Startup`（全名匹配）。

- Startup{EnvironmentName} (任意命名空间)。
- Startup (任意命名空间)。

从指定启动程序集中加载 Startup 类型的逻辑体现在如下所示的 FindStartupType 方法中。在执行构造函数的最后阶段，如果 WebHostOptions 选项的 StartupAssembly 属性被设置了一个启动程序集，定义在该程序集中的 Startup 方法会被加载出来，并作为参数调用上面定义的 UseStartup 方法完成对它的注册。如果在此过程中抛出异常，并且将 WebHostOptions 选项的 CaptureStartupErrors 属性设置为 True，那么捕捉到的异常会通过设置 GenericWebHostServiceOptions 对象的 ConfigureApplication 属性的方式重新抛出来。

```
internal class GenericWebHostBuilder :
    IWebHostBuilder,
    ISupportsStartup,
    ISupportsUseDefaultServiceProvider
{
    public GenericWebHostBuilder(IHostBuilder builder)
    {
        ...
        if (!string.IsNullOrEmpty(webHostOptions.StartupAssembly))
        {
            try
            {
                var startupType = FindStartupType(webHostOptions.StartupAssembly,
                    webhostContext.HostingEnvironment.EnvironmentName);
                UseStartup(startupType, context, services);
            }
            catch (Exception ex) when (webHostOptions.CaptureStartupErrors)
            {
                var capture = ExceptionDispatchInfo.Capture(ex);
                services.Configure<GenericWebHostServiceOptions>(options =>
                {
                    options.ConfigureApplication = app => capture.Throw();
                });
            }
        }
    }
}

private static Type FindStartupType(string startupAssemblyName, string environmentName)
{
    var assembly = Assembly.Load(new AssemblyName(startupAssemblyName))
        ?? throw new InvalidOperationException(
            string.Format("The assembly '{0}' failed to load.", startupAssemblyName));
    var startupNameWithEnv = $"Startup{environmentName}";
    var startupNameWithoutEnv = "Startup";

    var type =
        assembly.GetType(startupNameWithEnv) ??
        assembly.GetType(startupAssemblyName + "." + startupNameWithEnv) ??

```

```

        assembly.GetType(startupNameWithoutEnv) ??
        assembly.GetType(startupAssemblyName + "." + startupNameWithoutEnv);
    if (null != type)
    {
        return type;
    }

    var types = assembly.DefinedTypes.ToList();
    type = types.Where(info => info.Name.Equals(startupNameWithEnv,
        StringComparison.OrdinalIgnoreCase)).FirstOrDefault()
        ?? types.Where(info => info.Name.Equals(startupNameWithoutEnv,
        StringComparison.OrdinalIgnoreCase)).FirstOrDefault();
    return type?? throw new InvalidOperationException(
        string.Format(
            "A type named '{0}' or '{1}' could not be found in assembly '{2}'.",
            startupNameWithEnv,
            startupNameWithoutEnv,
            startupAssemblyName));
}
}

```

Hosting Startup

Startup 类型可定义在任意程序集中，并通过配置的方式注册到 ASP.NET Core 应用中。**Hosting Startup** 与之类似，我们可以将一些初始化操作定义在任意程序集中，在无须修改应用程序任何代码的情况下利用配置的方式实现对它们的注册。两者的不同之处在于：整个应用最终只会使用到一个 **Startup** 类型，但是采用 **Hosting Startup** 注册的初始化操作都是有效的。**Hosting Startup** 类型提供的方式将一些工具“附加”到一个 ASP.NET Core 应用中。

通过第 11 章的介绍可知，以 **Hosting Startup** 方法实现的初始化操作必须实现在 **IHostingStartup** 接口的实现类型中，该类型最终以 **HostingStartupAttribute** 特性的方式进行注册。**Hosting Startup** 相关的配置最终体现在 **WebHostOptions** 如下的 3 个属性上。

```

public class WebHostOptions
{
    public bool PreventHostingStartup { get; set; }
    public IReadOnlyList<string> HostingStartupAssemblies { get; set; }
    public IReadOnlyList<string> HostingStartupExcludeAssemblies { get; set; }
}

```

定义在 **IHostingStartup** 实现类型上的初始化操作会作用在一个 **IWebHostBuilder** 对象上，但最终对象并不是 **GenericWebHostBuilder**，而是如下所示的 **HostingStartupWebHostBuilder** 对象。从给出的代码片段可以看出，**HostingStartupWebHostBuilder** 对象实际上是对 **GenericWebHostBuilder** 对象的进一步封装，针对它的方法调用最终还是转移到封装的 **GenericWebHostBuilder** 对象上。

```

internal class HostingStartupWebHostBuilder :
    IWebHostBuilder, ISupportsStartup, ISupportsUseDefaultServiceProvider
{
    private readonly GenericWebHostBuilder _builder;
}

```

```
private Action<WebHostBuilderContext, IConfigurationBuilder> configureConfiguration;
private Action<WebHostBuilderContext, IServiceCollection> _configureServices;

public HostingStartupWebHostBuilder(GenericWebHostBuilder builder)
    => _builder = builder;

public IWebHost Build()
    => throw new NotSupportedException();

public IWebHostBuilder ConfigureAppConfiguration(
    Action<WebHostBuilderContext, IConfigurationBuilder> configureDelegate)
{
    _configureConfiguration += configureDelegate;
    return this;
}

public IWebHostBuilder ConfigureServices(
    Action<IServiceCollection> configureServices)
    => ConfigureServices((context, services) => configureServices(services));

public IWebHostBuilder ConfigureServices(
    Action<WebHostBuilderContext, IServiceCollection> configureServices)
{
    configureServices += configureServices;
    return this;
}

public string GetSetting(string key) => builder.GetSetting(key);

public IWebHostBuilder UseSetting(string key, string value)
{
    _builder.UseSetting(key, value);
    return this;
}

public void ConfigureServices(WebHostBuilderContext context,
    IServiceCollection services) => configureServices?.Invoke(context, services);

public void ConfigureAppConfiguration(WebHostBuilderContext context,
    IConfigurationBuilder builder) => _configureConfiguration?.Invoke(context, builder);

public IWebHostBuilder UseDefaultServiceProvider(
    Action<WebHostBuilderContext, ServiceProviderOptions> configure)
    => _builder.UseDefaultServiceProvider(configure);

public IWebHostBuilder Configure(
    Action<WebHostBuilderContext, IApplicationBuilder> configure)
    => builder.Configure(configure);

public IWebHostBuilder UseStartup(Type startupType) => _builder.UseStartup(startupType);
}
```

Hosting Startup 的实现体现在如下所示的 ExecuteHostingStartups 方法中，该方法会根据当前的配置和作为应用名称的入口程序集名称创建一个新的 WebHostOptions 对象，如果这个配置选项的 PreventHostingStartup 属性返回 True，就意味着人为关闭了 Hosting Startup 特性。在 Hosting Startup 特性没有被显式关闭的情况下，该方法会利用配置选项的 HostingStartupAssemblies 属性和 HostingStartupExcludeAssemblies 属性解析出启动程序集名称，并从中解析出注册的 IHostingStartup 类型。在通过反射的方式创建出对应的 IHostingStartup 对象之后，上面介绍的 HostingStartupWebHostBuilder 对象会被创建出来，并作为参数调用这些 IHostingStartup 对象的 Configure 方法。

```
internal class GenericWebHostBuilder :
    IWebHostBuilder,
    ISupportsStartup,
    ISupportsUseDefaultServiceProvider
{
    private readonly IHostBuilder          builder;
    private readonly IConfiguration        config;

    public GenericWebHostBuilder(IHostBuilder builder)
    {
        _builder          = builder;
        _config           = new ConfigurationBuilder()
            .AddEnvironmentVariables(prefix: "ASPNETCORE ")
            .Build();

        builder.ConfigureHostConfiguration(config =>
        {
            config.AddConfiguration( config);
            ExecuteHostingStartups();
        });
    }

    private void ExecuteHostingStartups()
    {
        var options = new WebHostOptions(
            _config, Assembly.GetEntryAssembly()?.GetName().Name);
        if (options.PreventHostingStartup)
        {
            return;
        }

        var exceptions = new List<Exception>();
        hostingStartupWebHostBuilder = new HostingStartupWebHostBuilder(this);

        var assemblyNames = options.HostingStartupAssemblies
            .Except(options.HostingStartupExcludeAssemblies,
                StringComparer.OrdinalIgnoreCase)
            .Distinct(StringComparer.OrdinalIgnoreCase);
    }
}
```

```

        foreach (var assemblyName in assemblyNames)
        {
            try
            {
                var assembly = Assembly.Load(new AssemblyName(assemblyName));
                foreach (var attribute in
                    assembly.GetCustomAttributes<HostingStartupAttribute>())
                {
                    var hostingStartup = (IHostingStartup)Activator
                        .CreateInstance(attribute.HostingStartupType);
                    hostingStartup.Configure(_hostingStartupWebHostBuilder);
                }
            }
            catch (Exception ex)
            {
                exceptions.Add(new InvalidOperationException(
                    $"Startup assembly {assemblyName} failed to execute. See the inner
                    exception for more details.", ex));
            }
        }
        if (exceptions.Count > 0)
        {
            hostingStartupErrors = new AggregateException(exceptions);
        }
    }
}

```

由于调用 `IHostingStartup` 对象的 `Configure` 方法传入的 `HostingStartupWebHostBuilder` 对象是对当前 `GenericWebHostBuilder` 对象的封装，而这个 `GenericWebHostBuilder` 对象又是对 `IHostBuilder` 的封装，所以以 `Hosting Startup` 注册的初始化操作最终还是应用到了以 `IHost/IHost Builder` 为核心的承载系统中。虽然 `GenericWebHostBuilder` 类型实现了 `IWebHostBuilder` 接口，但它仅仅是 `IHostBuilder` 对象的代理，其自身针对 `IWebHost` 对象的构建需求不复存在，所以它的 `Build` 方法会直接抛出异常。

```

internal class GenericWebHostBuilder :
    IWebHostBuilder,
    ISupportsStartup,
    ISupportsUseDefaultServiceProvider
{
    public IWebHost Build() => throw new NotSupportedException(
        $"Building this implementation of {nameof(IWebHostBuilder)} is not supported.");
    ...
}

```

13.4.4 ConfigureWebHostDefaults

演示实例中针对 `IWebHostBuilder` 对象的应用都体现在 `IHostBuilder` 接口的 `ConfigureWebHostDefaults` 扩展方法上，它最终调用的其实是如下所示的 `ConfigureWebHost` 扩展方法。如下面的代码片段所示，`ConfigureWebHost` 方法会将当前 `IHostBuilder` 对象创建的

`GenericWebHostBuilder` 对象作为参数调用指定的 `Action<IWebHostBuilder>` 委托对象。由于 `GenericWebHostBuilder` 对象相当于 `IHostBuilder` 对象的代理，所以这个委托中完成的所有操作最终都会转移到 `IHostBuilder` 对象上。

```
public static class GenericHostWebHostBuilderExtensions
{
    public static IHostBuilder ConfigureWebHost(
        this IHostBuilder builder, Action<IWebHostBuilder> configure)
    {
        var webhostBuilder = new GenericWebHostBuilder(builder);
        configure(webhostBuilder);
        return builder;
    }
}
```

顾名思义，`ConfigureWebHostDefaults` 方法会帮助我们做默认设置，这些设置实现在静态类型 `WebHost` 的 `ConfigureWebDefaults` 方法中。如下所示的 `ConfigureWebDefaults` 方法的实现，该方法提供的默认设置包括将定义在 `Microsoft.AspNetCore.StaticWebAssets.xml` 文件（物理文件或者内嵌文件）作为默认的 Web 资源、注册 `KestrelServer`、配置关于主机过滤（Host Filter）和 `Http Overrides` 相关选项、注册路由中间件，以及对用于集成 IIS 的 `AspNetCoreModule` 模块的配置。

```
public static class GenericHostBuilderExtensions
{
    public static IHostBuilder ConfigureWebHostDefaults(this IHostBuilder builder,
        Action<IWebHostBuilder> configure)
        => builder.ConfigureWebHost(webHostBuilder =>
        {
            WebHost.ConfigureWebDefaults(webHostBuilder);
            configure(webHostBuilder);
        });
}

public static class WebHost
{
    internal static void ConfigureWebDefaults(IWebHostBuilder builder)
    {
        builder.ConfigureAppConfiguration((ctx, cb) =>
        {
            if (ctx.HostingEnvironment.IsDevelopment())
            {
                StaticWebAssetsLoader.UseStaticWebAssets(ctx.HostingEnvironment);
            }
        });
        builder.UseKestrel((builderContext, options) =>
        {
            options.Configure(builderContext.Configuration.GetSection("Kestrel"));
        })
        .ConfigureServices((hostingContext, services) =>
```

```
{
    services.PostConfigure<HostFilteringOptions>(options =>
    {
        if (options.AllowedHosts == null || options.AllowedHosts.Count == 0)
        {
            var hosts = hostingContext
                .Configuration["AllowedHosts"]?.Split(new[] { ';' },
                StringSplitOptions.RemoveEmptyEntries);
            options.AllowedHosts = (hosts?.Length > 0 ? hosts : new[] { "*" });
        }
    });
    services.AddSingleton<IOptionsChangeTokenSource<HostFilteringOptions>>(
        new ConfigurationChangeTokenSource<HostFilteringOptions>(
            hostingContext.Configuration));

    services.AddTransient<IStartupFilter, HostFilteringStartupFilter>();

    if (string.Equals("true",
        hostingContext.Configuration["ForwardedHeaders_Enabled"],
        StringComparison.OrdinalIgnoreCase))
    {
        services.Configure<ForwardedHeadersOptions>(options =>
        {
            options.ForwardedHeaders =
                ForwardedHeaders.XForwardedFor | ForwardedHeaders.XForwardedProto;

            options.KnownNetworks.Clear();
            options.KnownProxies.Clear();
        });

        services.AddTransient<IStartupFilter, ForwardedHeadersStartupFilter>();
    }

    services.AddRouting();
})
.UseIIS()
.UseIISIntegration();
}
```