ASP.NET Core 6

框架揭秘

第 1 章

编程体验

虽然本书的读者大都是 .NET Core 的开发者，对于.NET Core 及 ASP.NET Core 的基本编程模式也都很熟悉，但是当我们升级到.NET 6，很多东西都发生了改变。很多特性被添加进来，现有一些编程方式也被改进，有的甚至不再推荐使用。尤其是 ASP.NET Core 6 推出的 Minimal API 应用承载方式让程序变得异常简洁，所以本书所有的演示实例将全部采用这种编程模式。本章提供了 20 个极简的实例，它们可以帮助读者对 ASP.NET Core 的基本编程模式有一个大体的认识。

1.1 控制台程序

本章提供的 20 个简单的演示实例涵盖了 ASP.NET Core 基本的编程模式，这些实例不仅用于演示控制台、API、MVC、gRPC 应用的构建与编程，还用于演示 Dapr 在 ASP.NET Core 中的应用。除此之外，这 20 个实例还涵盖了依赖注入、配置选项、日志记录的应用。我们先从最简单的控制台应用开始，不过在此之前先简单了解一下如何构建 ASP.NET Core 6 开发环境。

1.1.1 构建开发环境

.NET 6 的官方网站介绍了在各种操作系统平台（Windows、macOS 和 Linux）上构建开发环境的方式。总体来说，在不同的平台上开发 .NET 应用都需要安装相应的 SDK 和 IDE。成功安装 SDK 之后，我们在本地将自动拥有 .NET 的运行时、基础类库及相应的开发工具。顺便说一下，本书提供的演示实例默认采用的运行环境为 Windows。

dotnet.exe 是 .NET SDK 提供的一个重要的命令行工具。我们在进行 .NET 应用的开发部署时会频繁地使用它。dotnet.exe 提供了很多实用的命令，后续章节涉及相关内容时再进行针对性介绍。当 .NET SDK 安装结束之后，通过执行“dotnet”命令可以确认 .NET SDK 是否安装成功。如图 1-1 所示，执行“dotnet --info”命令可以查看当前安装的 .NET SDK 的基本信息，显示的信息包含 SDK 的版本、运行环境及本机安装的所有运行时版本。从图 1-1 中可以看出本机只安装了一个 6.0.0 版本。

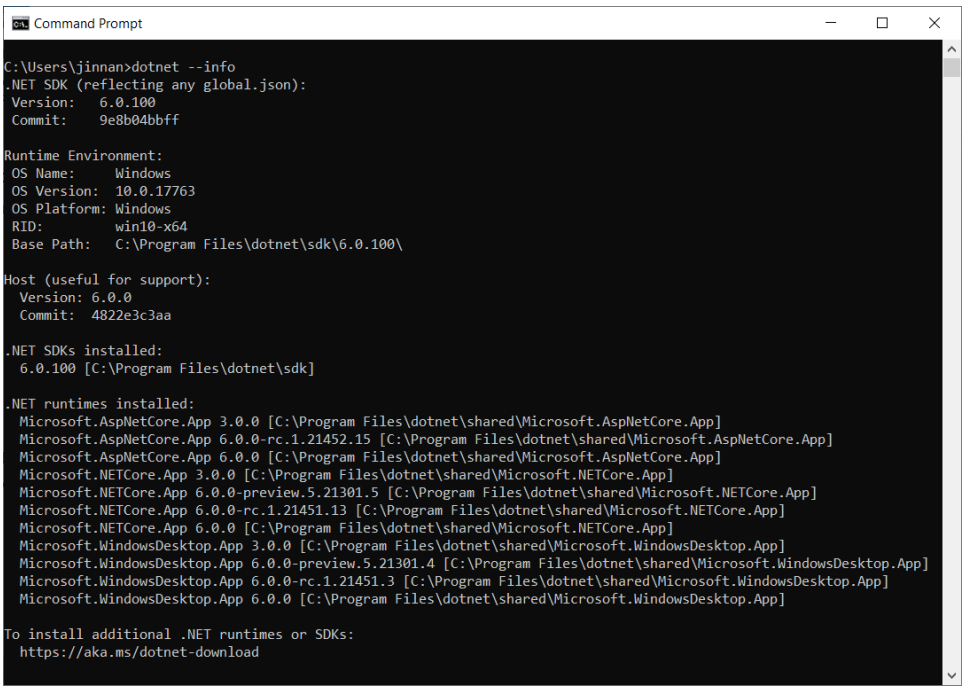

图 1-1　执行“dotnet --info”命令获取 .NET SDK 的基本信息

高效的开发离不开优秀的 IDE，在这方面作为一个 .NET 开发者是幸福的，因为我们拥有强大的 IDE——Visual Studio。虽然 Visual Studio Code 也是一款优秀的产品，但作者推荐使用 Visual Studio，尤其是在 Windows 上进行开发时。开发 ASP.NET Core 6 应用需要使用最新的 Visual Studio 2022。Visual Studio 提供了社区版（Community）、专业版（Professional）和企业版（Enterprise），其中社区版是免费的，专业版和企业版需要付费购买。

除了 Visual Studio 和 Visual Studio Code 这两款由微软自家提供的 IDE，我们还可以使用 Rider。Rider 是 JetBrains 开发的一款针对 .NET 的 IDE，可以利用它开发 ASP.NET、 .NET、Xmarin 及 Unity 应用。和 Visual Studio Code 一样，Rider 也是一款跨平台的 IDE，我们可以同时在 Windows、maxOS 及各种桌面版本的 Linux Distribution 上使用它。但 Rider 不是一款免费的 IDE，对其感兴趣的读者可以在官网方站下载 30 天试用版。

1.1.2　命令行构建 .NET 应用

dotnet.exe 提供了一个用来创建初始应用的“new”命令。在启动开发流程时，我们一般不会从第一行代码开始写起，可以利用这个“new”命令创建一个具有初始结构的应用程序。在开发过程中如果需要添加某种类型的文件（如各种类型的配置文件、MVC 应用的 Controller 类型文件和视图文件等），则可以利用“new”命令来完成。 .NET SDK 在安装时提供了一系列预定义的脚手架模板，可以执行“dotnet new --list”命令列出当前安装的模板，如图 1-2 所示。

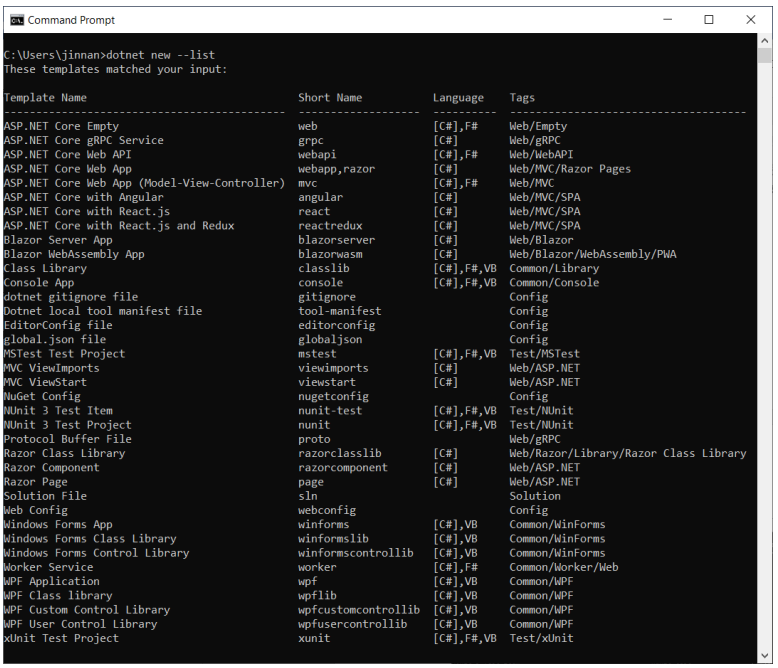

图 1-2　执行“dotnet new --list”命令获取脚手架模板列表

下面执行“dotnet new”命令（dotnet new console -n App）创建一个名为“App”的控制台程序，如图 1-3 所示。具体来说，该命令执行之后会在当前工作目录创建一个由指定应用名称命名的子目录，并将生成的文件存放在里面。和传统的 .NET Framework 及 .NET Core 一样，一个由 C#开发的 ASP.NET Core 项目依然由一个对应的.csproj 文件进行定义，图 1-3 中的 App.csproj 就是这样的一个文件。

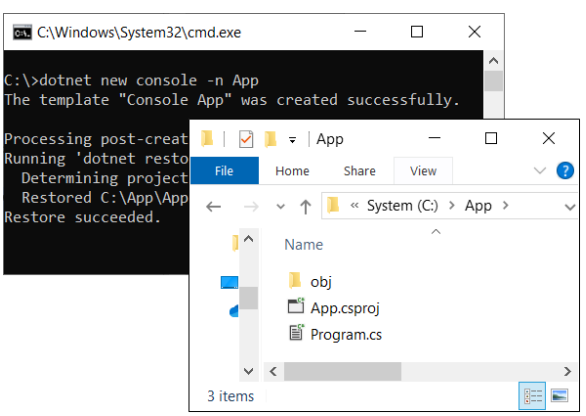

图 1-3　执行“dotnet new”命令创建一个控制台程序

.csproj 文件最终是为 MSBuild 服务的，该文件提供了相关的配置来控制 MSBuild 针对当前

项目的编译和发布行为。下面的代码就是 App.csproj 文件的全部内容，如果你曾经查看过传统 .NET Framework 下的 .csproj 文件，就会惊叹于这个 App.csproj 文件内容的简洁。.NET Core 6 下简洁的项目文件缘于对 SDK 的应用。不同的应用类型会采用不同的 SDK。比如，创建的这个控制台应用采用的 SDK 为“Microsoft.NET.Sdk”，ASP.NET Core 应用会采用另一个名为“Microsoft.NET.Sdk.Web”的 SDK。如果开发用于承载后台服务的应用，则一般会采用“Microsoft.NET.Sdk.Worker”这个 SDK，“第 14 章 服务承载”专门介绍了这个话题。SDK 相当于为某种类型的项目制定了一套面向 MSBuild 的基准配置，如果在项目文件的<Project>根节点设置了具体的 SDK，就意味着直接将这套基准配置继承下来。

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net6.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>
</Project>
```

在上面的代码中，与项目相关的属性可以分组定义在项目文件的<PropertyGroup>节点下。这个 App.csproj 文件定义了 4 个属性，其中，OutputType 属性和 TargetFramework 属性表示编译输出类型与采用的目标框架。由于创建的是一个针对 ASP.NET Core 的可执行控制台应用，所以将 OutputType 和 TargetFramework 的属性分别设置为“Exe”和“net6.0”。

项目的 ImplicitUsings 属性与 C# 10 提供的“全局命名空间”新特性有关。在这个特性被推出之前，用来导入命名空间的 using 语句的作用范围仅限于当前源文件，这就意味着常用的命名空间需要在不同的源文件中进行重复导入。顾名思义，“全局命名空间”就是针对整个项目全局导入的命名空间，它被定义在项目编译后自动生成的一个 .cs 文件中。创建的项目被编译后会在“obj\Debug\net6.0”目录下生成一个名为“App.GlobalUsings.g.cs”的 C#文件，该文件采用如下定义全局导入了一系列命名空间。

```
// <auto-generated/>
global using global::System;
global using global::System.Collections.Generic;
global using global::System.IO;
global using global::System.Linq;
global using global::System.Net.Http;
global using global::System.Threading;
global using global::System.Threading.Tasks;
```

导入的命名空间因采用的 SDK 而有所不同，如果将 SDK 更改为“Microsoft.NET.Sdk.Web”，则会发现 “App.GlobalUsings.g.cs”文件导入了更多的命名空间，这部分多出的命名空间在开发 ASP.NET Core 应用时基本上都会用到。

```
// <auto-generated/>
global using global::Microsoft.AspNetCore.Builder;
global using global::Microsoft.AspNetCore.Hosting;
global using global::Microsoft.AspNetCore.Http;
```

```
global using global::Microsoft.AspNetCore.Routing;
global using global::Microsoft.Extensions.Configuration;
global using global::Microsoft.Extensions.DependencyInjection;
global using global::Microsoft.Extensions.Hosting;
global using global::Microsoft.Extensions.Logging;
global using global::System;
global using global::System.Collections.Generic;
global using global::System.IO;
global using global::System.Linq;
global using global::System.Net.Http;
global using global::System.Net.Http.Json;
global using global::System.Threading;
global using global::System.Threading.Tasks;
```

项目的另一个名为 Nullable 的属性与 C#的一个名为“空值（Null）验证”的特性有关。相信每一个 .NET 开发人员都很熟悉 NullReferenceException 这个异常类型。照理说这是一个不应该出现的异常，因为我们在调用某个对象的某个方法或者将它作为参数传入某个方法时，本就应该进行空值验证。但是现在我们面临的问题是，方法的输入参数是否可以接收 Null 无法通过声明体现出来。方法的返回值也是如此，我们无法确定得到的结果是否为 Null。

为了解决这个问题，从 C# 8.0 版本开始引入了“可空引用类型（Nuallable Reference Type）”的概念。如果方法的参数可以接收 Null，或者返回对象可能是 Null，则对应的类型上应该加上“?”后缀。如果没有“?”后缀，隐含的意思就是参数或者返回值不可能为 Null，Visual Studio 会根据这个特性对程序实施空值检验。以下面这两行代码为例，由于 Type 类型的 GetMethod 方法返回的是可空的 MethodInfo 对象（MethodInfo?），所以针对返回的 MethodInfo 对象的 Invoke 方法的调用，Visual Studio 会产生一个警告，提醒方法调用的目标对象可能为 Null。

```
var method = typeof(Console).GetMethod("WriteLine", new Type[] { typeof(object) });
method.Invoke(null, new object[] { new Foobar("foo", "bar") });
```

但是对于上面这种场景，我们知道 MethodInfo 其实是不可能为 Null 的，此时可以采用如下两种方式消除这个警告。一种是在调用 GetMethod 方法得到的 MethodInfo 对象的后面，添加一个“!”作为后缀，另一种是在调用 Invoke 方法时将此后缀添加到 method 变量后面。这两种方式都是为了表明开发人员明确知道对应的变量不为 Null。本书提供的所有实例将全程开启这个特性，并尽量保证不会有空值警告产生。

```
var method = typeof(Console).GetMethod("WriteLine", new Type[] { typeof(object) })!;
method.Invoke(null, new object[] { new Foobar("foo", "bar") });
```

```
var method = typeof(Console).GetMethod("WriteLine", new Type[] { typeof(object) });
method!.Invoke(null, new object[] { new Foobar("foo", "bar") });
```

执行“dotnet new”命令除了可以创建一个空的控制台程序，还会生成一些初始化代码，下面就是在项目目录下生成的 Program.cs 文件的内容。可以看出整个文件只有两行代码，其中一行还是注释。这唯一的一行代码调用了 Console 类型的静态方法，将字符串“Hello, World!”输出到控制台上。这里体现了 C # 10“顶级语句（Top-level Statements）”的新特性。

```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

对于 C#这门面向对象的编程语言来说，所有的代码都属于一个类型，即使是作为入口的 Main 方法也不例外。如果采用 C# 10，则入口程序的代码可以作为顶级语句独立存在，因为在编译程序时会将它们放到一个自动生成的 Program 类型的 Main 方法中。这个特性对于真正的项目开发并不会带来多大的好处，但是对于作者来说是一个福音，由于采用了这个特性，本书缩减了提供的代码片段的篇幅。

通过执行脚手架命令行创建的应用程序虽然简单，但它是一个完整的 .NET 应用。我们可以在无须任何修改的情况下直接编译和运行它。针对 .NET 应用的编译和运行同样可以执行“dotnet.exe”命令完成。如图 1-4 所示，在将项目根目录作为工作目录后，执行“dotnet build”命令对这个控制台应用实施编译。由于默认采用 Debug 编译模式，所以编译生成的程序集会保存在“\bin\Debug\”目录下。同一个应用可以采用多个目标框架，针对不同目标框架编译生成的程序集被放在不同的目录下。由于创建的是 ASP.NET Core 的应用程序，所以最终生成的程序集被保存在“\bin\Debug\net6.0\”目录下。

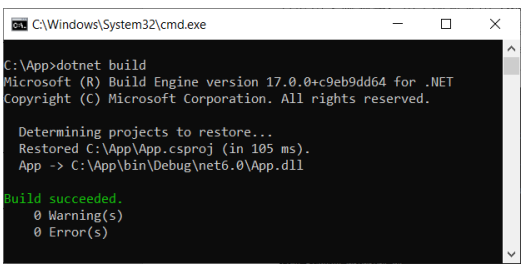

图 1-4　执行“dotnet build”命令编译一个控制台程序

如果查看编译的输出目录，则可以发现两个同名（App）的程序集文件，一个是 App.dll，另一个是 App.exe，后者在体积上会大很多。App.exe 是一个可以直接运行的可执行文件，而 App.dll 只是一个单纯的动态链接库，需要借助“dotnet”命令才能执行。

如图 1-5 所示，当执行“dotnet run”命令后，编译后的程序随即被执行，“Hello, World!”字符串被直接输出到控制台上。执行“dotnet run”命令启动程序之前其实无须显式执行“dotnet build”命令对源代码实施编译，因为该命令会自动触发编译操作。在执行“dotnet”命令启动应用程序集时，也可以直接指定启动程序集的路径（dotnet bin\Debug\net6.0\App.dll）。实际上“dotnet run”命令主要用在开发测试中，dotnet {AppName}.dll 的方式才是部署环境（如 Docker 容器）中采用的启动方式。（S101）[①]

① 解释见附录 A

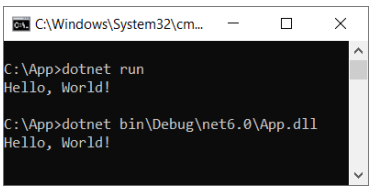

图 1-5　执行“dotnet”命令启动一个控制台程序

1.2　ASP.NET 应用

在前文中，我们利用“dotnet new”命令创建了一个简单的控制台程序，接下来将其改造成一个 ASP.NET 应用。我们在前面已经说过，不同的应用类型会采用不同的 SDK，所以直接修改 App.csproj 文件将 SDK 设置为“Microsoft.NET.Sdk.Web”。由于不需要利用生成的 .exe 文件来启动 ASP.NET 应用，所以应该将 XML 元素<OutputType>Exe</OutputType>从<PropertyGroup>节点中删除。

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>
</Project>
```

如果此时使用 Visual Studio 直接打开项目文件 App.csproj，则会发现项目根目录下自动生成一个名为“Properties”的子目录，launchSettings.json 配置文件自动生成并且被保存在此目录下，如图 1-6 所示。

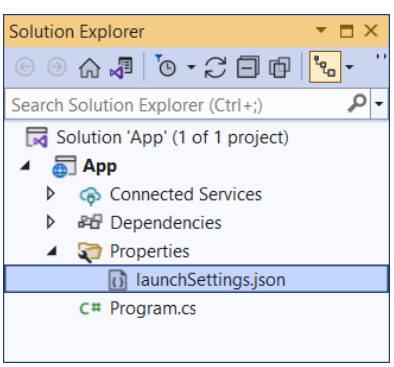

图 1-6　launchSettings.json 配置文件

1.2.1　launchSettings.json

顾名思义，launchSettings.json 是一个在应用启动时自动加载的配置文件，该配置文件可以采用不同的设置启动应用程序。下面使用 Visual Studio 自动创建 launchSettings.json 文件的全部

内容。我们可以看出该配置文件默认添加了 iisSettings 和 profiles 两个节点，iisSettings 节点提供 IIS 相关的配置，profiles 节点定义了一系列针对不同场景的 Profile。

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:49301/",
      "sslPort": 44334
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "App": {
      "commandName": "Project",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE ENVIRONMENT": "Development"
      },
      "applicationUrl": "https://localhost:5001;http://localhost:5000"
    }
  }
}
```

初始的 launchSettings.json 配置文件会默认创建两个 Profile，一个被命名为“IIS Express”，另一个则使用应用名称命名（App）。每个 Profile 相当于定义了应用启动时采用的设置，包括应用启动的方式、环境变量和 URL 等，具体设置如下。

- commandName：启动当前应用程序的命令类型，有效的选项包括 IIS、IISExpress、Executable 和 Project，前 3 个选项分别表示采用 IIS、IISExpress 和指定的可执行文件（.exe）来启动应用程序。如果使用“dotnet run”命令启动应用程序，则需要将对应 Profile 的 commandName 属性设置为 Project。
- executablePath：如果 commandName 属性的值被设置为 Executable，则需要利用 executablePath 属性设置启动可执行文件的路径（绝对路径或相对路径）。
- environmentVariables：该属性用来设置环境变量。由于 launchSettings.json 配置文件只在开发环境中使用，所以默认会添加一个名为“ASPNETCORE_ENVIRONMENT”的环境变量，并将它的值设置为“Development”，ASP.NET Core 应用就是利用这样一个环境变量来表示当前部署环境的。

- commandLineArgs：命令行参数，即传入 Main 方法的参数列表。如果采用“顶级语句”特性，则对应 args 变量。
- workingDirectory：启动当前应用运行的工作目录。
- applicationUrl：应用程序采用的 URL 列表，多个 URL 之间使用分号（;）分隔。
- launchBrowser：一个布尔类型的开关，表示应用程序启动时是否自动启动浏览器。
- launchUrl：如果 launchBrowser 属性的值被设置为 True，则浏览器采用的初始化路径通过 launchUrl 属性进行设置。
- nativeDebugging：是否启动本地代码调试（Native Code Debugging），默认值为 False。
- externalUrlConfiguration：如果该属性的值被设置为 True，就意味着禁用本地的配置，默认值为 False。
- use64Bit：如果 commandName 属性的值被设置为 IIS Express，则 use64Bit 属性决定采用 x64 版本还是 x86 版本，默认值为 False，这样 ASP.NET Core 应用默认采用 x86 版本的 IIS Express。

launchSettings.json 配置文件中的所有设置仅仅针对开发环境，在产品（Production）环境下是不需要这个配置文件的，所以应用发布后生成的文件列表中也不包含该配置文件。该配置文件其实不需要手动编辑，当前项目属性对话框（在解决方案对话框中选择“Properties”选项打开当前项目属性对话框）中“Debug”选项卡下的所有设置最终都会体现在该配置文件上，如图 1-7 所示。

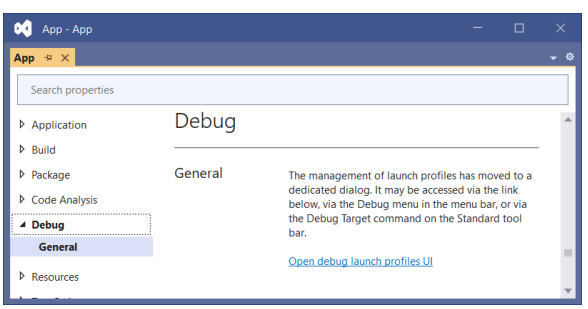

图 1-7　在 Visual Studio 中通过设置调试选项编辑 launchSettings.json 配置文件

如果在 launchSettings.json 配置文件中设置了多个 Profile，则它们会以图 1-8 的形式出现在 Visual Studio 的工具栏中。我们可以选择任意一个 Profile 来启动当前程序。如果在 Profile 中通过设置 launchBrowser 属性决定启动浏览器，则可以选择浏览器的类型。

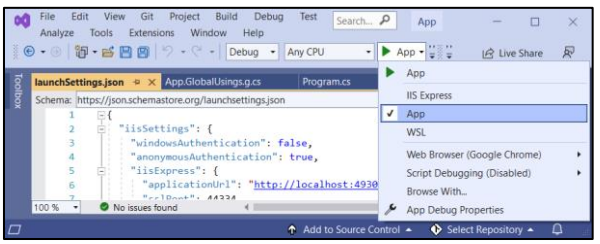

图 1-8　在 Visual Studio 中选择 Profile

如果我们针对项目根目录通过执行“dotnet run”命令启动应用程序，则 launchSettings.json 配置文件默认会被加载。我们可以通过命令行参数“--launch-profile”指定采用的 Profile。如果没有对 Profile 做显式指定，则默认选择配置文件中第一个“commandName”为“Project”的 Profile。如果在执行“dotnet run”命令时不希望加载 launchSettings.json 配置文件，则可以显式指定命令行参数“--no-launch-profile”。

如果不需要生成这个 launchSettings.json 配置文件，则可以修改项目文件并按照如下方式添加一个名为“NoDefaultLaunchSettingsFile”的属性。对于本书后续章节涉及的演示实例，默认都会通过这种方式来抑制这个启动配置文件的生成，这是为了减少演示实例涉及的文件，更重要的是我们针对演示实例运行效果的描述都是基于默认配置的。

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
    <NoDefaultLaunchSettingsFile>true</NoDefaultLaunchSettingsFile>
  </PropertyGroup>
</Project>
```

1.2.2　Minimal API

ASP.NET Core 应用的承载（Hosting）经历了 3 次较大的变迁，由于最新的承载方式提供的 API 最为简洁且依赖最小，所以我们将它称为“Minimal API”。本书提供的大部分演示实例均使用了 Minimal API。下面就是采用这种编程模式编写的第一个 Hello World 程序。

```
RequestDelegate handler = context => context.Response.WriteAsync("Hello, World!");
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
WebApplication app = builder.Build();
app.Run(handler: handler);
app.Run();
```

上面的代码涉及 3 个重要的对象，其中，WebApplication 对象表示承载的应用，Minimal API 采用“构建者（Builder）”模式来构建它，此构建者体现为一个 WebApplicationBuilder 对象。在上面代码中，调用 WebApplication 类型的静态工厂方法 CreateBuilder 创建了一个 WebApplicationBuilder 对象，该方法的参数 args 表示命令行参数数组。在调用该对象的 Build 方法将 WebApplication 对象构建出来后，我们调用了它的 Run 扩展方法并使用一个

RequestDelegate 对象作为其参数。虽然 RequestDelegate 是一个简单的委托类型，但是它在 ASP.NET Core 框架体系中地位非凡。下面先来对它做一个简单的介绍。

当一个 ASP.NET Core 应用启动之后，它会使用注册的服务器绑定到指定的端口进行请求监听。当接收抵达的请求之后，一个通过 HttpContext 表示的上下文对象会被创建出来。我们不仅可以从这个上下文对象中提取所有与当前请求相关的信息，还能直接使用该上下文对象完成对请求的响应。关于这一点完全可以从 HttpContext 这个抽象类的两个核心属性 Request 和 Response 看出来。

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

由于 ASP.NET Core 应用针对请求的处理总是在一个 HttpContext 上下文对象中进行的，所以针对请求的处理器可以表示为一个 Func<HttpContext, Task>类型的委托。由于这样的委托会被广泛地使用，所以 ASP.NET Core 直接定义了一个专门的委托类型，就是我们在程序中使用的 RequestDelegate。从下面 RequestDelegate 类型的定义可以看出，它本质上就是一个 Func <HttpContext, Task>委托对象。

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

再次回到演示程序。首先创建了一个 RequestDelegate 委托对象，对应的目标方法会在响应输出流中写入字符串“Hello, World!”。我们将此委托对象作为参数调用 WebApplication 对象的 Run 扩展方法，这个调用可以理解为将这个委托对象作为所有请求的处理器，接收到的所有请求都将通过这个委托对象来处理。演示程序最后调用 WebApplication 对象的另一个无参 Run 扩展方法是为了启动承载的应用。

在 Visual Studio 下，我们可以直接按 F5 键（或 Ctrl + F5 组合键）启动该程序，当然执行“dotnet run”命令的应用启动方式依然有效，本书提供的演示实例大都会采用这种方式。如图 1-9 所示，我们以命令行方式启动程序后，控制台上出现了 ASP.NET Core 框架输出的日志，日志表明应用程序已经开始在默认的两个终节点（http://localhost:5000 和 https://localhost:5001）监听请求了。我们使用浏览器对这两个终结节发送了两个请求，均得到一致的响应。从响应的内容可以看出应用程序正是利用我们指定的 RequestDelegate 委托对象处理请求的。（S102）

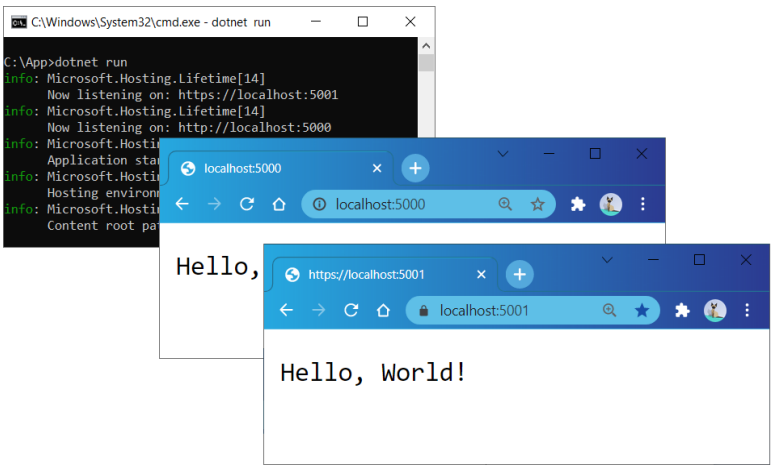

图 1-9　启动应用程序并利用浏览器进行访问

上面演示的应用程序先调用定义在 WebApplication 类型的静态工厂方法 CreateBuilder 创建一个 WebApplicationBuilder 对象，再利用后者构建一个表示承载应用的 WebApplication 对象。WebApplicationBuilder 对象提供了很多用来对构建 WebApplication 对象进行设置的 API，但是演示的应用程序并未使用到它们，此时我们可以直接调用静态工厂方法 Create 将 WebApplication 对象创建出来。在下面的改写应用程序中，我们直接将请求处理器定义为一个本地静态方法 HandleAsync。（S103）

```
var app = WebApplication.Create(args);
app.Run(handler: HandleAsync);
app.Run();

static Task HandleAsync(HttpContext httpContext)
    => httpContext.Response.WriteAsync("Hello, World!");
```

1.2.3　中间件

承载的 ASP.NET Core 应用最终体现为由注册中间件构建的请求处理管道。在服务器接收到请求并成功构建出 HttpContext 上下文对象之后，会将请求交给这个管道进行处理。在管道完成了处理任务之后，控制权再次回到服务器的手中，它会将处理的结果转换为响应发送出去。从应用编程的角度来看，这个管道体现为上述的 RequestDelegate 委托对象，组成它的单个中间件则体现为另一个类型为 Func<RequestDelegate,RequestDelegate>的委托对象，该委托对象的输入和输出都是一个 RequestDelegate 对象，前者表示由后续中间件构建的管道，后者表示将当前中间件纳入此管道后生成的新管道。可能读者目前对此还不能完全地理解，不过没有关系，“第 15 章 应用承载（上）”“第 16 章 应用承载（中）”“第 17 章 应用承载（下）”会全面而深入地阐述这个话题。

在上面的演示实例中，将一个 RequestDelegate 委托对象作为参数调用了 WebApplication 对

象的 Run 扩展方法，当时说这是为应用程序设置一个请求处理器。其实这种说法不够准确，该扩展方法只是注册了一个中间件。说得更加具体一点，这个扩展方法用于注册处于管道末端的中间件。为了让读者体验到中间件和管道对请求的处理，可以对上面的演示实例进行如下修改。（S104）

```
var app = WebApplication.Create(args);
IApplicationBuilder appBuilder = app;
appBuilder
    .Use(middleware: HelloMiddleware)
    .Use(middleware: WorldMiddleware);
app.Run();

static RequestDelegate HelloMiddleware(RequestDelegate next)
    => async httpContext => {
    await httpContext.Response.WriteAsync("Hello,  ");
    await next(httpContext);
};

static RequestDelegate WorldMiddleware(RequestDelegate next)
     => httpContext => httpContext.Response.WriteAsync("World!");
```

由于中间件体现为一个 Func<RequestDelegate,RequestDelegate>委托对象，所以利用上面定义的两个与该委托对象类型具有一致声明的本地静态方法 HelloMiddleware 和 WorldMiddleware 来表示对应的中间件。我们将完整的文本“Hello, World!”拆分为“Hello, ”和“World!”两段，分别由上述两个终节点写入响应输出流。在创建出表示承载应用的 WebApplication 对象之后，将它转换为 IApplicationBuilder 接口类型，并调用其 Use 方法完成对上述两个中间件的注册（由于 WebApplication 类型显式实现了定义在 IApplicationBuilder 接口中的 Use 方法，所以我们不得不进行类型转换）。利用浏览器采用相同地址请求启动的应用程序，我们依然可以得到如图 1-9 所示的响应内容。

虽然中间件最终总是体现为一个 Func<RequestDelegate，RequestDelegate>委托对象，但是我们在开发过程中可以采用各种不同的形式来定义中间件。比如，可以将中间件定义为如下两种类型的委托对象，这两个委托对象分别使用作为输入参数的 RequestDelegate 和 Func<Task>完成对后续管道的调用。

- Func<HttpContext, RequestDelegate, Task>。
- Func<HttpContext, Func<Task>, Task>。

我们现在来演示如何使用 Func<HttpContext, RequestDelegate, Task>委托对象的形式来定义中间件。在下面的代码中，我们将 HelloMiddleware 方法和 WorldMiddleware 方法替换为与 Func<HttpContext, RequestDelegate, Task>委托对象类型具有一致声明的本地静态方法。（S105）

```
var app = WebApplication.Create(args);
app
    .Use(middleware: HelloMiddleware)
    .Use(middleware: WorldMiddleware);
app.Run();
```

```
static async Task HelloMiddleware(HttpContext httpContext, RequestDelegate next)
{
    await httpContext.Response.WriteAsync("Hello, ");
    await next(httpContext);
};

static Task WorldMiddleware(HttpContext httpContext, RequestDelegate next)
    => httpContext.Response.WriteAsync("World!");
```

下面的程序以类似的方式将这两个中间件替换为与 Func<HttpContext, Func<Task>, Task>委托对象类型具有一致声明的本地方法。当调用 WebApplication 对象的 Use 方法将这两种“变体”注册为中间件时，该方法内部会将提供的委托对象转换为 Func<RequestDelegate,RequestDelegate>类型。（S106）

```
var app = WebApplication.Create(args);
app
    .Use(middleware: HelloMiddleware)
    .Use(middleware: WorldMiddleware);
app.Run();

static async Task HelloMiddleware(HttpContext httpContext, Func<Task> next)
{
    await httpContext.Response.WriteAsync("Hello, ");
    await next();
};

static Task WorldMiddleware(HttpContext httpContext, Func<Task> next)
    => httpContext.Response.WriteAsync("World!");
```

当我们试图利用一个自定义中间件来完成某种请求处理功能时，其实很少会将中间件定义为上述这 3 种委托形式，基本上都会将其定义为一个具体的类型。中间件类型有多种定义方式，其中一种是直接实现 IMiddleware 接口。本书将其称为“强类型”的中间件定义方式。现在就采用这样的方式定义一个简单的中间件类型。

无论是定义中间件类型，还是定义其他的服务类型，如果它们具有对其他服务的依赖，则应该采用依赖注入（Dependency Injection）的方式将它们整合在一起。整个 ASP.NET Core 框架就建立在依赖注入框架之上，依赖注入已经成为 ASP.NET Core 最基本的编程方式，针对依赖注入的系统介绍被放在“第 2 章 依赖注入（上）”和“第 3 章 依赖注入（下）”。我们接下来会演示依赖注入在自定义中间件类型中的应用。

在前面演示的实例中，我们利用中间件写入以“硬编码”方式指定的问候语“Hello, World!”，现在选择 IGreeter 接口表示的服务根据指定的时间来提供对应的问候语，Greeter 类型是该接口的默认实现。这里需要提前说明一下，本书提供的所有的演示实例都以“App”命名，独立定义的类型默认会定义在约定的“App”命名空间下。为了节省篇幅，接下来类型定义代码将不再提供所在的命名空间，当启动应用程序出现针对“App”命名空间的导入时希望读者不要感到奇怪。

```
namespace App
{
    public interface IGreeter
    {
        string Greet(DateTimeOffset time);
    }

    public class Greeter : IGreeter
    {
        public string Greet(DateTimeOffset time) => time.Hour switch
        {
            var h when h >= 5 && h < 12      => "Good morning!",
            var h when h >= 12 && h < 17     => "Good afternoon!",
            _                                => "Good evening!"
        };
    }
}
```

我们定义了一个名为 GreetingMiddleware 的中间件类型。在下面代码中，该类型实现了 IMiddleware 接口，请求的处理实现在 InvokeAsync 方法中。我们在 GreetingMiddleware 类型的构造函数中注入了 IGreeter 对象，并利用它在实现的 InvokeAsync 方法中根据当前时间来提供对应的问候语，后者将作为请求的响应内容。

```
public class GreetingMiddleware : IMiddleware
{
    private readonly IGreeter  greeter;
    public GreetingMiddleware(IGreeter greeter)
        => _greeter = greeter;

    public Task InvokeAsync(HttpContext context, RequestDelegate next)
        => context.Response.WriteAsync(_greeter.Greet(DateTimeOffset.Now));
}
```

GreetingMiddleware 中间件的应用体现在如下代码中。我们通过调用了 WebApplication 对象的 UseMiddleware<GreetingMiddleware>扩展方法注册了这个中间件。由于强类型中间件实例是由依赖注入容器实时提供的，所以必须预先将它注册为服务。注册最终会添加到 WebApplicationBuilder 的 Services 属性返回的 IServiceCollection 对象上，我们在得到这个对象后通过调用它的 AddSingleton< GreetingMiddleware >方法将该中间件注册为“单例服务”。由于中间件依赖 IGreeter 服务，所以通过调用 AddSingleton<IGreeter, Greeter>扩展方法对该服务进行了注册。

```
using App;
var builder = WebApplication.CreateBuilder(args);
builder.Services
    .AddSingleton<IGreeter, Greeter>()
    .AddSingleton<GreetingMiddleware>();
var app = builder.Build();
app.UseMiddleware<GreetingMiddleware>();
app.Run();
```

启动该应用程序之后，针对它的请求会得到根据当前时间生成的问候语。如图 1-10 所示，由于目前的时间为晚上 7 点，所以浏览器上显示“Good evening!”。（S107）

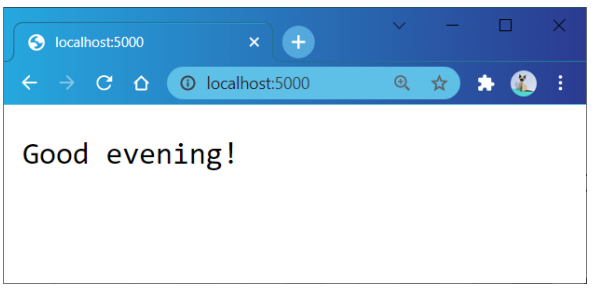

图 1-10　自定义中间件返回的问候语

中间件类型其实并不一定非得实现某个接口，或者继承某个基类，按照既定的约定进行定义即可。按照 ASP.NET Core 的约定，中间件类型需要定义成一个公共实例类型（静态类型无效），其构造函数中可以注入任意的依赖服务，但必须包含一个 RequestDelegate 类型的参数，该参数表示由后续中间件构建的管道，当前中间件利用它将请求分发给后续管道进行进一步处理。请求的处理实现在 InvokeAsync 方法或 Invoke 方法中，这些方法的返回类型都为 Task，第一个参数被绑定为当前的 HttpContext 上下文对象，所以 GreetingMiddleware 中间件类型可以改写成如下形式。

```
public class GreetingMiddleware
{
    private readonly IGreeter _greeter;
    public GreetingMiddleware(RequestDelegate next, IGreeter greeter)
        =>  greeter = greeter;
    public Task InvokeAsync(HttpContext context)
        => context.Response.WriteAsync( greeter.Greet(DateTimeOffset.Now));
}
```

强类型的中间件实例是在对请求进行处理时由依赖注入容器实时提供的，按照约定定义的中间件实例则不同，在注册中间件时就已经利用依赖注入容器将它创建，所以前者可以采用不同的生命周期模式，后者总是一个单例对象。也正是因为这个原因，我们不需要将中间件注册为服务。（S108）

```
using App;
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddSingleton<IGreeter, Greeter>();
var app = builder.Build();
app.UseMiddleware<GreetingMiddleware>();
app.Run();
```

对于按照约定定义的中间件类型，依赖服务不一定非要注入构造函数中，我们可以选择直接注入 InvokeAsync 方法或 Invoke 方法中，所以上面这个 GreetingMiddleware 中间件也可以定义成如下形式。对于按照约定定义的中间件类型，构造函数注入和方法注入并不是等效的，两者之间的差异会在“第 17 章 应用承载（下）”中进行介绍。（S109）

```
public class GreetingMiddleware
{
    public GreetingMiddleware(RequestDelegate next){}
    public Task InvokeAsync(HttpContext context, IGreeter greeter)
        => context.Response.WriteAsync(greeter.Greet(DateTimeOffset.Now));
}
```

1.2.4 配置选项

在开发 ASP.NET Core 应用过程中，我们会广泛使用到配置（Configuration）。ASP.NET Core 采用了一个非常灵活的配置框架。我们可以将存储在任何载体的数据作为配置源，还可以将结构化的配置转换成对应的选项（Options）类型，以强类型的方式来使用它们。针对配置选项的系统介绍被放在“第 5 章 配置选项（上）”和“第 6 章 配置选项（下）”中，我们先在这里“预热”一下。

在前面演示的实例中，Greeter 类型针对指定时间提供的问候语依然是以“硬编码”的方式提供的，现在将它们放到配置文件以方便进行调整。为此我们在项目根目录下添加一个名为“appsettings.json”的配置文件，并将 3 条问候语以如下形式定义在 JSON 文件中。

```
{
  "greeting": {
    "morning": "Good morning!",
    "afternoon": "Good afternoon!",
    "evening": "Good evening!"
  }
}
```

ASP.NET Core 应用中的配置通过 IConfiguration 对象表示。我们可以采用依赖注入的形式“自由”地使用它。对于演示的程序来说，我们首先按照如下方式将 IConfiguration 对象注入 Greeter 类型的构造函数中，然后调用其 GetSection 方法得到定义了上述问候语的配置节（greeting）。在实现的 Greet 方法中，以索引的方式利用指定的 Key（morning、afternoon 和 evening）提取对应的问候语。由于程序启动时会自动加载这个按照约定命名的“appsettings.json”配置文件，所以程序的其他地方不要进行任何修改。（S110）

```
public class Greeter : IGreeter
{
    private readonly IConfiguration  configuration;
    public Greeter(IConfiguration configuration)
        => _configuration = configuration.GetSection("greeting");

    public string Greet(DateTimeOffset time) => time.Hour switch
    {
        var h when h >= 5 && h < 12  =>  configuration["morning"],
        var h when h >= 12 && h < 17 =>  configuration["afternoon"],
        _                            => _configuration["evening"],
    };
}
```

正如前面所说，将结构化的配置转换成对应类型的 Options 对象，以强类型的方式来使用它

们是更加推荐的编程模式。为此我们为 3 条问候语定义了如下 GreetingOptions 配置选项类型。

```
public class GreetingOptions
{
    public string Morning { get; set; }      = default!;
    public string Afternoon { get; set; }    = default!;
    public string Evening { get; set; }      = default!;
}
```

虽然 Options 对象不能直接以依赖服务的形式进行注入，但却可以由注入的 IOptions<TOptions>对象来提供。如下面的代码片段所示，我们在 Greeter 类型的构造函数中注入 IOptions<GreetingOptions>对象，并利用其 Value 属性中得到需要的 GreetingOptions 对象。在得到这个对象之后，实现的 Greet 方法中只需要从对应的属性中获取相应的问候语即可。

```
public class Greeter : IGreeter
{
    private readonly GreetingOptions  options;
    public Greeter(IOptions<GreetingOptions> optionsAccessor)
        =>  options = optionsAccessor.Value;

    public string Greet(DateTimeOffset time) => time.Hour switch
    {
        var h when h >= 5 && h < 12          => _options.Morning,
        var h when h >= 12 && h < 17         => _options.Afternoon,
                                             =>  options.Evening
    };
}
```

由于 IOptions<GreetingOptions>对象提供的配置选项不能无中生有（实际上存在于配置中），所以我们需要将对应的配置节（greeting）绑定到 GreetingOptions 对象上。这项工作其实也属于服务注册的范畴，具体可以按照如下形式调用 IServiceCollection 对象的 Configure<TOptions>扩展方法来完成。如下面的代码片段所示，表示应用整体配置的 IConfiguration 对象来源于 WebApplicationBuilder 的 Configuration 属性。（S111）

```
using App;
var builder = WebApplication.CreateBuilder(args);
builder.Services
    .AddSingleton<IGreeter, Greeter>()
    .Configure<GreetingOptions>(builder.Configuration.GetSection("greeting"));
var app = builder.Build();
app.UseMiddleware<GreetingMiddleware>();
app.Run();
```

1.2.5　诊断日志

诊断日志对于纠错排错必不可少。ASP.NET Core 采用的诊断日志框架强大、易用且灵活，“第 7 章 诊断日志（上）”“第 8 章 诊断日志（中）”“第 9 章 诊断日志（下）”会对这个主题进行系统而深入的介绍。现在我们试着为演示程序添加诊断日志的功能。

在演示程序中，Greeter 类型会根据指定的时间返回对应的问候语，现在将时间和对应的问候

语以日志的方式记录下来并看一看两者是否匹配。在前文中曾提过，依赖注入是 ASP.NET Core 应用最基本的编程模式。我们将涉及的功能（无论是与业务相关的还是与业务无关的）进行拆分，最终以具有不同粒度的服务将整个程序化整为零，服务之间的依赖关系直接以注入的方式来解决。前文演示了配置选项的注入，而用来记录日志的 ILogger 对象依然采用注入的方式获得。

如下面的代码片段所示，我们在 Greeter 类型的构造函数中注入了 ILogger<Greeter>对象。在实现的 Greet 方法中，我们调用该对象的 LogInformation 扩展方法记录了一条 Information 等级的日志，日志内容体现了时间与问候语文本之间的映射关系。

```
public class Greeter : IGreeter
{
    private readonly GreetingOptions  options;
    private readonly ILogger          logger;

    public Greeter(IOptions<GreetingOptions> optionsAccessor, ILogger<Greeter> logger)
    {
        _options      = optionsAccessor.Value;
         logger       = logger;
    }

    public string Greet(DateTimeOffset time)
    {
        var message = time.Hour switch
        {
            var h when h >= 5 && h < 12      => _options.Morning,
            var h when h >= 12 && h < 17     =>  options.Afternoon,
                                             =>  options.Evening
        };
         logger.LogInformation(message:"{time} => {message}",time, message);
        return message;
    }
}
```

由于采用 Minimal API 编写的 ASP.NET Core 应用程序会默认将诊断日志整合进来，所以整个演示程序的其他地方都不要修改。当修改后的应用程序启动之后，每一个请求都会通过日志留下“痕迹”。由于控制台是默认开启的日志输出渠道之一，所以日志内容直接会输出到控制台上。图 1-11 所示为以命令行形式启动应用程序，控制台上显示的都是以日志形式输出的内容。在众多系统日志中，我们发现有一条是由 Greeter 对象输出的。（S112）

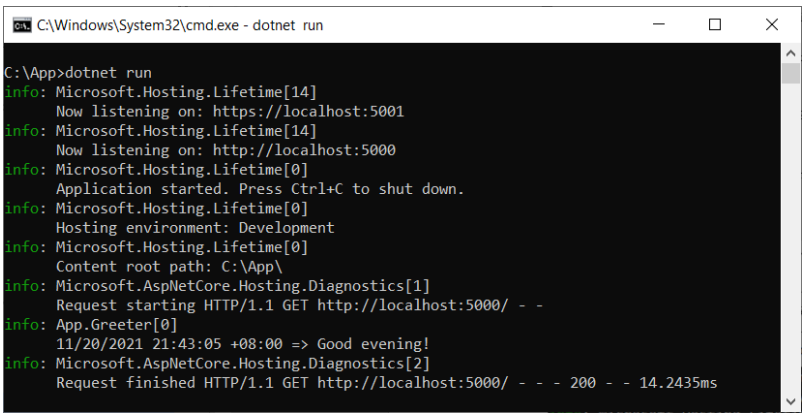

图 1-11　输出到控制台上的日志

1.2.6　路由

ASP.NET Core 的路由是由 EndpointRoutingMiddleware 和 EndpointMiddleware 两个中间件实现的，在所有预定义的中间件类中，它们是比较重要的两个中间件，因为不仅是 MVC 和 gRPC 框架建立在路由系统之上，后面介绍的 Dapr.NET 的发布订阅和 Actor 编程模式也是如此。我们会在“第 20 章 路由”中系统地介绍由这两个中间件构建的路由系统。现在我们放弃前面演示实例中注册的中间件，改用路由的方式来实现类似的功能。

如下面的代码片段所示，我们在利用 WebApplicationBuilder 将表示承载应用的 WebApplication 对象构建出来之后，并没有注册任何中间件，而是调用它的 MapGet 扩展方法注册了一个指向路径“/greet”的路由终节点（Endpoint）。该终节点的处理器是一个指向 Greet 方法的委托对象，这就意味着请求路径为“/greet”的 GET 请求会路由到这个终节点，并最终调用 Greet 方法进行处理。

```
using App;
var builder = WebApplication.CreateBuilder(args);
builder.Services
    .AddSingleton<IGreeter, Greeter>()
    .Configure<GreetingOptions>(builder.Configuration.GetSection("greeting"));
var app = builder.Build();
app.MapGet("/greet", Greet);
app.Run();

static string Greet(IGreeter greeter) => greeter.Greet(DateTimeOffset.Now);
```

ASP.NET Core 的路由系统的强大之处在于，我们可以使用任何类型的委托对象作为注册终节点的处理器，路由系统在调用处理器方法之前会“智能地”提取相应的数据初始化每一个参数。当方法执行之后，它还会针对具体返回的对象来对请求实施响应。对于提供的 Greet 方法来说，路由系统在调用它之前会利用依赖注入容器提供作为参数的 IGreeter 对象。由于返回的是一个字符串，所以文本经过编码后会直接作为响应的主体内容，响应的内容类型（Content-

Type）最终会被设置为“text/plain”。启动程序之后，如果利用浏览器请求“/greet”路径，则针对当前时间解析出来的问候语会以图 1-12 的形式呈现出来。（S113）

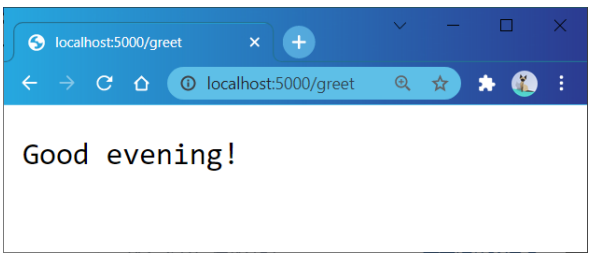

图 1-12　采用路由返回的问候语

1.3　MVC

ASP.NET Core 可以视为一种底层框架，它为我们构建了基于管道的请求处理模型。我们可以进一步在它上面构建某种编程模型的 Web 框架。例如，用于开发 API 和 Web 页面的 MVC 框架，采用 Protocol Buffers 消息编码方式和 HTTP2 传输协议的 gRPC 框架，旨在进行实时交互的 SignalR 框架，采用 Actor 模型的 Orleans 框架等。我们现在就来演示一下如何编写一个极简的 MVC 应用。

1.3.1　定义 Controller

我们直接将上面演示的程序改写成 MVC 应用。MVC 应用以 Controller 为核心，所有的请求总是指向定义在某个 Controller 类型中的某个 Action 方法。当应用接收到请求之后，会激活对应的 Controller 对象，并通过执行对应的 Action 方法处理该请求。按照约定，合法的 Controller 类型必须是以“Controller”作为后缀命名的公共实例类型。我们一般会让定义的 Controller 类型派生自 Controller 基类以“借用”一些有用的 API，但这不是必需的。例如，下面定义的 GreetingController 就没有指定基类。

```
public class GreetingController
{
    [HttpGet("/greet")]
    public string Greet([FromServices] IGreeter greeter)
        => greeter.Greet(DateTimeOffset.Now);
}
```

由于 MVC 框架是建立在路由系统之上的，所以定义在 Controller 类型中的 Action 方法最终会转换成一个或者多个注册到指定路径模板的终节点。对于定义在 GreetingController 类型中 Action 的 Greet 方法来说，通过标注的 HttpGetAttrbute 特性不仅为对应的路由终节点定义了 HTTP 方法的约束（该终节点仅限于处理 GET 请求），还同时指定了绑定的请求路径（“/greet”）。

依赖的服务可以直接注入 Controller 类型中。具体来说，它支持两种注入形式，一种是注入构造函数中，另一种是直接注入 Action 方法中。对于方法注入，对应参数上必须标注一个

FromServiceAttribute 特性。IGreeter 对象就是采用这种方式注入 Greet 方法中的。与路由系统针对返回对象的处理方式一样，MVC 框架针对 Action 方法的返回值也会根据其类型进行针对性的处理。使用 Greet 方法返回的字符串会直接作为响应的主体内容，响应的内容类型（Content-Type）会被设置为“text/plain”。

在完成了 GreetingController 类型的定义之后，我们需要对入口程序进行相应的修改。如下面的代码片段所示，在完成了 IGreeter 服务的注册和 GreetingOptions 配置选项的设置之后，调用同一个 IServiceCollection 对象的 AddControllers 扩展方法注册了与 Controller 相关服务的注册。在 WebApplication 对象被构建出来之后，调用了它的 MapControllers 扩展方法将定义在所有 Controller 类型中的 Action 方法映射为对应的终节点。程序启动之后，如果我们利用浏览器请求“/greet”这个路径，则依然会得到相应的输出结果，如图 1-12 所示。（S114）

```
using App;
var builder = WebApplication.CreateBuilder(args);
builder.Services
    .AddSingleton<IGreeter, Greeter>()
    .Configure<GreetingOptions>(builder.Configuration.GetSection("greeting"))
    .AddControllers();
var app = builder.Build();
app.MapControllers();
app.Run();
```

1.3.2　引入视图

上面改造的 MVC 程序并没有涉及视图，请求的响应内容是由 Action 方法直接提供的，现在利用视图来呈现最终响应的内容。由于上面的实例调用 IServiceCollection 接口的 AddControllers 扩展方法只会注册 Controller 相关的服务，所以将其换成 AddControllersWithViews 扩展方法。顾名思义，新的扩展方法会将视图相关的服务添加进来。

```
using App;
var builder = WebApplication.CreateBuilder(args);
builder.Services
    .AddSingleton<IGreeter, Greeter>()
    .Configure<GreetingOptions>(builder.Configuration.GetSection("greeting"))
    .AddControllersWithViews();
var app = builder.Build();
app.MapControllers();
app.Run();
```

我们对 GreetinigController 进行了改造。如下面的代码片段所示，让它继承 Controller 这个基类。将 Action 方法 Greet 的返回类型修改为 IActionResult 接口，具体返回的是通过 View 方法创建的表示默认视图（针对当前 Action 方法）的 ViewResult 对象。在返回 Action 方法之前，它还利用对 ViewBag 的设置将当前时间传递到呈现的视图中。

```
public class GreetingController : Controller
{
    [HttpGet("/greet")]
```

```
    public IActionResult Greet()
    {
        ViewBag.Time = DateTimeOffset.Now;
        return View();
    }
}
```

ASP.NET Core MVC 采用 Razior 视图引擎，视图被定义成一个后缀名为.cshtml 的文件，这是一个按照 Razor 语法编写的静态 HTML 和动态 C#代码动态交织的文本文件。由于上面为了呈现试图调用的 View 方法没有指定任何参数，所以视图引擎会根据当前 Controller 的名称（Greeting）和 Action 的名称（Greet）去定位定义目标视图的.cshtml 文件。为了迎合默认的视图定位规则，我们需要采用 Action 的名称来命名创建的视图文件（Greet.cshtml），并将其添加到“Views/Greeting”目录下。

```
@using App
@inject IGreeter Greeter;
<html>
    <head>
        <title>Greeting</title>
    </head>
    <body>
        <p>@Greeter.Greet((DateTimeOffset)ViewBag.Time)</p>
    </body>
</html>
```

上面的代码片段就是添加的视图文件（Views/Greeting/Greet.cshtml）的内容。总体来说，这是一个 HTML 文档，除了在主体部分呈现的问候语文本（前置的@字符定义动态执行的 C#表达式）是根据指定时间动态解析出来的，其他内容均为静态的 HTML。借助@inject 指令将依赖的 IGreeter 对象以属性的形式注入，并且将属性名称设置为 Greeter，所以可以在视图中直接调用它的 Greet 方法得到呈现的问候语。调用 Greet 方法指定的时间是 GreetingController 利用 ViewBag 传递过来的，所以可以直接利用它将其提取出来。程序启动之后，利用浏览器请求“/greet”这个路径，虽然浏览器也会呈现出相同的文本（见图 1-13），但是响应的内容是完全不同的。之前响应的仅仅是内容类型为“text/plain”的单纯文本，现在响应的则是一份完整的 HTML 文档，内容类型为“text/html”。（S115）

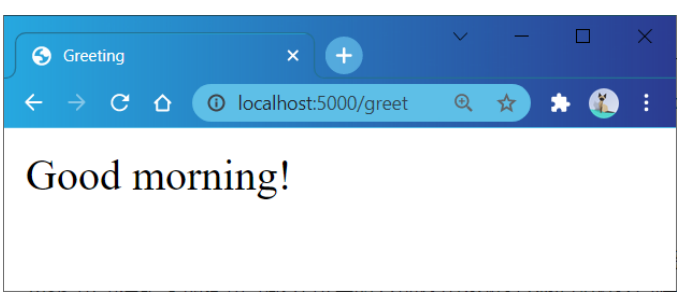

图 1-13　以视图形式返回的问候

1.4　gRPC

gRPC 最开始由 Google（gRPC 的“g”）开发，目前已经成为一款高性能、开源、语言中立的远程调用（RPC）框架。近年来，gRPC 越来越受到业界的欢迎，俨然成为可以代替传统的 Web API 的新势力。gRPC 之所以受到如此热捧，是因为其“高性能”，它采用的 Proto Buffers 数据序列化协议和 HTTP2 传输方式是成就其高性能的主要因素。目前，很多主流的编程语言都提供了 gRPC 支持，而面向 .NET 的 gRPC 框架就建立在 ASP.NET Core 框架上面。下面介绍一下 .NET 针对 gRPC 的编程体验。（S116）

1.4.1　定义服务

虽然 Visual Studio 提供了创建 gRPC 的项目模板，该模板提供的脚手架会自动创建一系列的初始文件，也会对项目做一些初始设置，但这反而是作者不想要的，至少不希望在这里使用这个模板。和前面一样，我们希望演示的实例只包含最本质和必要的元素，所以选择在一个空的解决方案上构建 gRPC 应用。

如图 1-14 所示，我们在一个空的解决方案上添加了 3 个项目。Proto 是一个空的类库项目，我们将会使用它来存放标准的 Proto Buffers 消息和 gRPC 服务的定义；Server 是一个空的 ASP.NET Core 应用，gRPC 服务的实现类型就放在这里，它也是承载 gRPC 服务的应用。Client 是一个控制台程序，用来模拟调用 gRPC 服务的客户端。

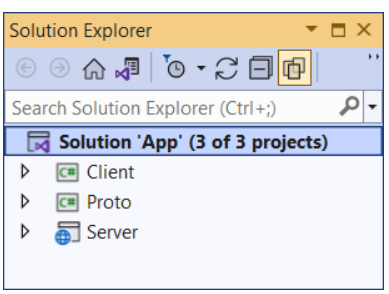

图 1-14　gRPC 解决方案

gRPC 是语言中立的远程调用框架，gRPC 服务契约使用的数据类型都采用标准的定义方式。具体来说，gRPC 传输的数据采用 Proto Buffers 协议进行序列化，Proto Buffers 采用高效紧凑的二进制编码。我们将用于定义数据类型和服务的 Proto Buffers 文件定义在 Proto 项目中，在这之前需要为这个空的类库项目添加“Grpc.AspNetCore”这个 NuGet 包的引用。

不再使用简单的“Hello World”，现在演示的 gPRC 服务指定另一种稍微“复杂”一点的应用场景——用它来完成简单的加、减、乘、除运算。在 Proto 项目中添加一个名为 Calculator.proto 的文本文件，并在其中以如下形式将 Calculator 这个 gRPC 服务进行定义。如下面的代码片段所示，这个服务包含 4 个操作，它们的输入和输出都被定义成 Proto Buffers 消息。作为输入的 InputMessage 消息包含两个整型的数据成员（表示运算的两个操作数）。返回的

OutputMessage 消息除了通过 result 表示计算结果，还具有 status 和 error 两个成员，前者表示计算状态（成功还是失败），后者提供计算失败时的错误消息。

```
syntax = "proto3";
option csharp_namespace = "App";

service Calculator {
  rpc Add (InputMessage) returns (OutputMessage);
  rpc Substract (InputMessage) returns (OutputMessage);
  rpc Multiply (InputMessage) returns (OutputMessage);
  rpc Divide (InputMessage) returns (OutputMessage);
}

message InputMessage {
  int32        x              = 1;
  int32        y              = 2;
}

message OutputMessage {
  int32        status         = 1;
  int32        result         = 2;
  string       error          = 3;
}
```

从上面的定义可以看出，gRPC 的“操作”和 C# 的“方法”不同，它只有输入和输出的概念，没有参数和返回值的概念。或者说它只有唯一的参数和返回值，并且都体现为一个 Proto Buffers 消息。如果读者了解 WCF，就会发现上面的定义和 WCF 的服务契约本质上是一致的。WCF 的服务契约以 XML 的形式定义了每个操作的请求和回复消息的结构，上面的 Calculator 也属于针对服务契约的定义，服务的 4 个操作的输入和输出消息的结构通过对应的 Proto Buffers 消息确定。Proto Buffers 消息的每个数据成员会被赋予一个唯一的“序号”，这个序号将会代替对应的成员名称写入序列化后的二进制内容中。Proto Buffers 消息被序列化之后，生成的二进制文件的体积会很小，这和序列化时采用序号代替成员名称有很大的关系。

创建的 Calculator.proto 文件无法直接被使用，我们需要利用内置的代码生成器将它转换成.cs 代码。具体操作很简单，我们只需要在 Visual Studio 的解决方案窗口中选择这个文件，打开 Calculator.proto 文件属性对话框，如图 1-15 所示。在 Build Action 下拉列表中选择“Protobuf compiler”选项，同时在 gRPC Stub Classes 下拉列表中选择“Client and Server”选项。

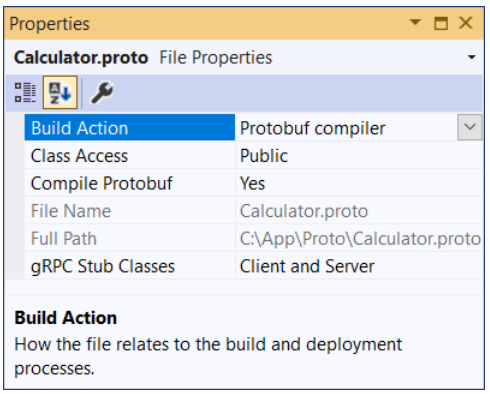

图 1-15　Calculator.proto 文件属性对话框

完成设置之后，无论何时对 Calculator.proto 文件所做的改变都将触发代码的自动生成，具体生成的 .cs 文件会自动保存在 obj 目录下。由于在 gRPC Stub Classes 下拉列表中选择了“Client and Server”选项，所以它不仅会生成服务端用来定义服务实现类型的 Stub 类，还会生成客户端用来调用服务的 Stub 类。上面以可视化形式进行的设置最终会体现在项目文件（Proto.csproj）上，所以直接修改此文件也可以达到相同的目的，如下面的代码就是这个文件的完整内容。

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>
  <ItemGroup>
    <None Remove="Calculator.proto" />
  </ItemGroup>
  <ItemGroup>
    <PackageReference Include="Grpc.AspNetCore" Version="2.40.0" />
  </ItemGroup>
  <ItemGroup>
    <Protobuf Include="Calculator.proto" />
  </ItemGroup>
</Project>
```

1.4.2　实现和承载

Proto 项目中的 Calculator.proto 文件仅仅是按照标准的形式定义的“服务契约”。我们需要在 Server 项目中定义具体的实现类型。在添加了 Proto 项目的引用之后，定义如下名为 CalculatorService 的 gRPC 服务实现类型。让 CalculatorService 类型继承自一个内嵌于 Calculator 中的 CalculatorBase 类型，这个 Calculator 就是根据 Calculator.proto 生成的一个类型。

```
public class CalculatorService : Calculator.CalculatorBase
{
    private readonly ILogger _logger;
```

```
    public CalculatorService(ILogger<CalculatorService> logger) => _logger = logger;

    public override Task<OutputMessage> Add(InputMessage request,
        ServerCallContext context)
        => InvokeAsync((op1, op2) => op1 + op2, request);
    public override Task<OutputMessage> Substract(InputMessage request,
       ServerCallContext context)
        => InvokeAsync((op1, op2) => op1 - op2, request);
    public override Task<OutputMessage> Multiply(InputMessage request,
        ServerCallContext context)
        => InvokeAsync((op1, op2) => op1 * op2, request);
    public override Task<OutputMessage> Divide(InputMessage request,
        ServerCallContext context)
        => InvokeAsync((op1, op2) => op1 / op2, request);

    private Task<OutputMessage> InvokeAsync(Func<int, int, int> calculate,
        InputMessage input)
    {
        OutputMessage output;
        try
        {
            output = new OutputMessage { Status = 0,
                Result = calculate(input.X, input.Y) };
        }
        catch (Exception ex)
        {
             logger.LogError(ex, "Calculation error.");
            output = new OutputMessage { Status = 1, Error = ex.ToString() };
        }
        return Task.FromResult(output);
    }
}
```

MVC 应用定义的 Controller 类型的构造函数中可以注入依赖服务，gRPC 服务实现类型也是如此，CalculatorService 的构造函数中注入的 ILogger<CalculatorService>对象就体现了这一点。Calculator.proto 文件为 Calcultor 服务定义的 4 个操作会转换成 CalculatorBase 类型中对应的虚方法。我们按照上面的方式重写了它们。

在完成了 gRPC 服务实现类型的定义之后，我们需要对承载它的入口程序编写如下代码。由于 gRPC 采用 HTTP2 传输协议，所以在利用 WebApplicationBuilder 的 WebHost 属性得到对应的 IWebHostBuilder 对象之后，调用其 ConfigureKestrel 扩展方法，让默认注册的 Kestrel 服务器监听的终节点默认采用 HTTP2 协议。gRPC 相关的服务通过调用 IServiceCollection 接口的 AddGrpc 扩展方法进行注册。由于 gRPC 也是建立在路由系统之上的，定义在服务中的每个操作最终也会转换成相应的路由终节点，这些终节点的生成和注册是通过调用 WebApplication 对象的 MapGrpcService<TService>扩展方法完成的。

```
using App;
using Microsoft.AspNetCore.Server.Kestrel.Core;
```

```
var builder = WebApplication.CreateBuilder(args);
builder.WebHost.ConfigureKestrel(kestrel => kestrel.ConfigureEndpointDefaults(
    endpoint => endpoint.Protocols =  HttpProtocols.Http2));
builder.Services.AddGrpc();
var app = builder.Build();
app.MapGrpcService<CalculatorService>();
app.Run();
```

1.4.3　调用服务

Calculator.proto 文件生成的代码包含用来调用对应 gRPC 服务的 Stub 类，所以模拟客户端的 Client 项目也需要添加对 Proto 项目的引用。在此之后，我们可以编写如下程序调用 gRPC 服务完成 4 种基本的数学运算。

```
using App;
using Grpc.Core;
using Grpc.Net.Client;

using var channel = GrpcChannel.ForAddress("http://localhost:5000");
var client = new Calculator.CalculatorClient(channel);
var inputMessage = new InputMessage { X = 1, Y = 0 };

await InvokeAsync(input => client.AddAsync(input), inputMessage, "+");
await InvokeAsync(input => client.SubstractAsync(input), inputMessage, "-");
await InvokeAsync(input => client.MultiplyAsync(input), inputMessage, "*");
await InvokeAsync(input => client.DivideAsync(input), inputMessage, "/");

static async Task InvokeAsync(Func<InputMessage, AsyncUnaryCall<OutputMessage>> invoker,
    InputMessage input, string @operator)
{
    var output = await invoker(input);
    if (output.Status == 0)
    {
        Console.WriteLine($"{input.X}{@operator}{input.Y}={output.Result}");
    }
    else
    {
        Console.WriteLine(output.Error);
    }
}
```

如上面的代码片段所示，通过调用 GrpcChannel 类型的静态方法 ForAddress，针对 gRPC 服务的地址“http://localhost:5000”创建了一个 GrpcChannel 对象，该对象表示与服务进行通信的“信道（Channel）”。我们利用它创建了一个 CalculatorClient 对象，作为调用 gRPC 服务的客户端或代理，CalculatorClient 类型同样是内嵌在生成的 Calculator 类型中的。最终我们利用这个代理完成了 4 种基本运算的服务调用，具体的 gRPC 调用实现在 InvokeAsync 本地方法中。

接下来以命令行的方式先后启动 Server 应用和 Client 应用，客户端和服务端的控制台上的输出结果如图 1-16 所示。由于传入的参数分别为 1 和 0，所以除除法运算外，其他三次调用都

会成功返回结果，除法调用则会输出错误信息。由于 CalculatorService 进行了异常处理，并且将异常信息以日志的形式记录下来，所以也将错误信息输出到服务端的控制台上。

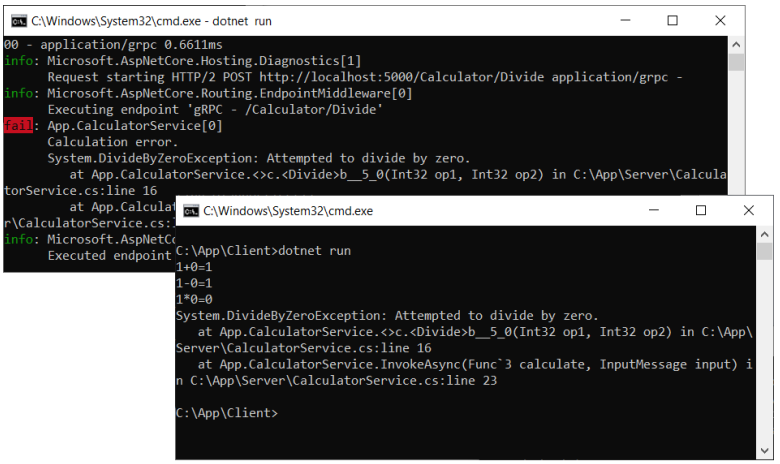

图 1-16　输出结果（1）

1.5　Dapr

有的读者可能没有接触过 Dapr，但是一定对它“有所耳闻”，本书出版期间有很多人都在谈论它。我们从其命名（Dapr 的全称是“分布式应用运行时，Distributed Application Runtime”）可以看出 Dapr 的定位，它并不是分布式应用的开发框架，它提供的是更底层的“运行时”。我们可以使用不同的编程语言，采用不同的开发框架在这个由 Dapr 提供的“运行时”上面构建分布式应用。下面介绍 Dapr 在 .NET 中的开发体验。

1.5.1　构建开发环境

目前，Dapr 支持 3 种典型的部署或承载（Hosting）方式，第一种是采用 Self-Host 的方式部署到本地开发物理机或虚拟机上，第二种是部署到 Kubernetes 集群中，第三种是以 Serverless 的模式部署到云端（如 Azure）。我们选择最简单的 Self-Host 承载模式。

Dapr 开发环境的构建从安装 Dapr 命令行（Dapr-Cli）开始。Dapr 的官方文档提供了在各种环境下安装 Dapr 命令行的方式。在 Windows 下只需要以管理员身份执行如下“PowerShell”命令即可。

```
powershell -Command
"iwr -useb https://raw.githubusercontent.com/dapr/cli/master/install/install.ps1 | iex"
```

安装过程结束之后，可以执行“dapr”命令检验 Dapr 命令行是否安装成功。如果控制台上出现如图 1-17 所示的输出结果，则表示已经成功安装了 Dapr 命令行。接下来就可以执行“dapr init”命令，在本地完成 Dapr 开发和运行环境的初始化工作，该命令需要以管理员身份运行。在这个过程中可能会因为国内的网络问题出现初始化失败，我们可能需要多试几次才能成功。

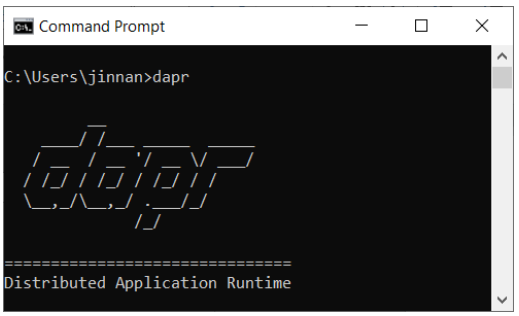

图 1-17　输出结果（2）

在不指定任何参数的情况下执行“dapr init”命令可以构建并启动 3 个 Docker 容器，所以需要确保初始化 Dapr 之前本机安装并启动了 Docker。如图 1-18 所示，Dapr 初始化过程中构建的这 3 个容器分别是用来存储状态的 Redis 容器，用来收集分布式跟踪信息的 Zipkin 容器，以及用来提供 Actor 分布信息的 Placement 容器。

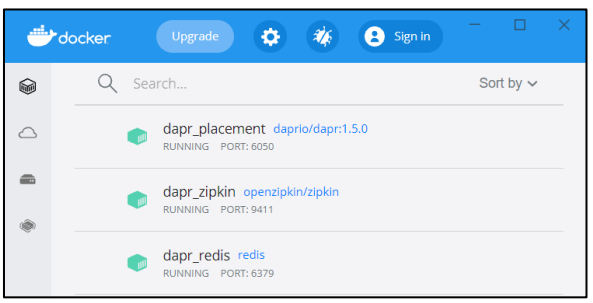

图 1-18　构建并启动 3 个 Docker 容器

1.5.2　服务调用

Dapr 是一个采用 Service Mesh 设计的分布式微服务运行时。每一个部署在 Dapr 上的应用实例（独立进程或容器）都具有一个专属的 Sidecar，具体体现为一个独立的进程（daprd）或容器。应用实例只会与它专属的 Sidecar 进行通信，跨应用通信是在两个应用实例的 Sidecar 之间进行的，具体的传输协议可以采用 HTTP 或 gRPC。正是因为应用实例和 Sidecar 是在各自的进程内独立运行的，所以 Dapr 才对应用开发采用的技术栈没有任何限制。

接下来就通过一个简单的实例演示 Dapr 下的服务调用。我们创建了 Dapr 应用解决方案，如图 1-19 所示。App1 和 App2 表示两个具有依赖关系的应用，App1 会调用 App2 提供的服务。Shared 是一个类库项目，用来提供被 App1 和 App2 共享的数据类型。

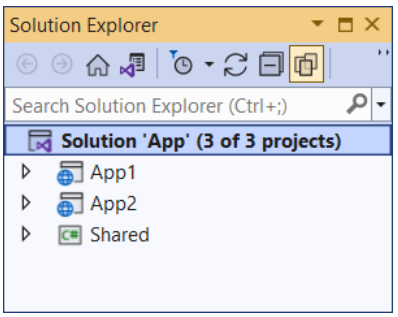

图 1-19　Dapr 应用解决方案

我们依然沿用上面演示的数学运算应用场景，并在 Shared 项目中定义如下两个数据类型。表示输入的 Input 类型提供了两个操作数（X 和 Y），表示输出的 Output 类型除了通过其 Result 属性表示运算结果，还利用 Timestamp 属性返回运算时间戳。

```
public class Input
{
    public int X { get; set; }
    public int Y { get; set; }
}

public class Output
{
    public int                Result { get; set; }
    public DateTimeOffset     Timestamp { get; set; } = DateTimeOffset.Now;
}
```

App2 就是一个简单的 ASP.NET Core 应用，我们采用路由的方式注册了执行数学运算的终节点。如下面的代码片段所示，注册的终节点采用的路径模板为“/{method}”，路由参数“{method}”既表示运算操作类型，也作为 Dapr 服务的方法名。在作为终节点处理器的 Calculate 方法中，请求的主体内容被提取出来，经过反序列化后绑定为 input 参数。在根据提供的输入执行对应的运算并生成 Output 对象后，将其序列化成 JSON 文本，并以此作为响应的内容。

```
using Microsoft.AspNetCore.Mvc;
using Shared;

var app = WebApplication.Create(args);
app.MapPost("{method}", Calculate);
app.Run("http://localhost:9999");

static IResult Calculate(string method, [FromBody] Input input)
{
    var result = method.ToLower() switch
    {
        "add" => input.X + input.Y,
        "sub" => input.X - input.Y,
        "mul" => input.X * input.Y,
```

```
        "div" => input.X / input.Y,
          => throw new InvalidOperationException($"Invalid method {method}")
    };
    return Results.Json(new Output { Result = result });
}
```

在调用 WebApplication 对象的 Run 方法启动应用时，显式指定了监听地址，其目的是将端口（9999）固定下来。App2 目前实际上与 Dapr 毫无关系，我们必须以 Dapr 的方式启动它才能将它部署到本机的 Dapr 环境中，具体来说可以执行“dapr run --app-id app2 --app-port 9999 --dotnet run”命令在启动 Sidecar 的同时以子进程的方式启动应用。提供的命令行参数除了提供应用的启动方式（dotnet run），还提供了应用的表示（--app-id app2）和监听的端口（--app-port 9999）。考虑到每次在控制台输入这些烦琐的命令有点麻烦，我们选择在 launchSettings.json 文件中定义如下 Profile 来以 Dapr 的方式启动应用。由于这种启动方式会将输出目录作为当前工作目录，所以选择指定程序集的方式来启动应用（dotnet App2.dll）。

```
{
  "profiles": {
    "Dapr": {
      "commandName": "Executable",
      "executablePath": "dapr",
      "commandLineArgs": "run --app-id app2 --app-port 9999 -- dotnet App2.dll"
    }
  }
}
```

App1 是一个简单的控制台应用，为了能够采用上述方式来启动它，我们还是将 SDK 从“Microsoft.NET.Sdk”修改为“Microsoft.NET.Sdk.Web”。在 launchSettings.json 文件中定义诸如下面的 Profile，应用的标识被设置为“app1”。由于 App1 仅涉及对其他应用的调用，自身并不提供服务，所以不需要设置端口。

```
{
  "profiles": {
    "Dapr": {
      "commandName": "Executable",
      "executablePath": "dapr",
      "commandLineArgs": "run --app-id app1 -- dotnet App1.dll"
    }
  }
}
```

由于 App1 涉及 Dapr 服务的调用，需要使用 Dapr 客户端 SDK 提供的 API，所以我们为它添加了“Dapr.Client”这个 NuGet 包的引用。具体的服务调用体现在下面的程序中，我们调用 DaprClient 的静态方法 CreateInvokeHttpClient，针对目标服务或应用的标识“app2”创建一个 HttpClient 对象，并利用它完成 4 个服务方法的调用。具体的服务调用是在 InvokeAsync 这个本地方法中实现的，在将作为输入的 Input 对象序列化成 JSON 文本之后，该本地方法会将其作为请求的主体内容。在一个分布式环境下，我们不需要知道目标服务所在的位置，因为这是不确定的，只需确定的是目标服务/应用的标识，所以直接将此标识作为请求的目标地址。在得到调

用结果之后，我们对它进行了简单的格式化并直接输出到控制台上。

```
using Dapr.Client;
using Shared;

HttpClient client = DaprClient.CreateInvokeHttpClient(appId: "app2");
var input = new Input(2, 1);

await InvokeAsync("add", "+");
await InvokeAsync("sub", "-");
await InvokeAsync("mul", "*");
await InvokeAsync("div", "/");

async Task InvokeAsync(string method, string @operator)
{
    var response = await client.PostAsync(method, JsonContent.Create(input));
    var output = await response.Content.ReadFromJsonAsync<Output>();
    Console.WriteLine(
        $"{input.X} {@operator} {input.Y} = {output.Result} ({output.Timestamp})");
}
```

在先后启动 App2 和 App1 之后，两个应用所在的控制台上的输出结果如图 1-20 所示。应用输出的文本会采用“==App==”作为前缀，其余内容为 Sidecar 输出的日志。从 App2 所在控制台（前面）上的输出结果可以看出，它成功地完成了基于 4 种运算的服务调用。当以 Debug 模式启动 App1 时会有“闪退”的现象，如果出现这样的情况，则可以选择非 Debug 模式（在解决方案窗口的项目上右击，在弹出的快捷菜单中选择“Debug => Start Without Debuging”命令）来启动它。（S117）

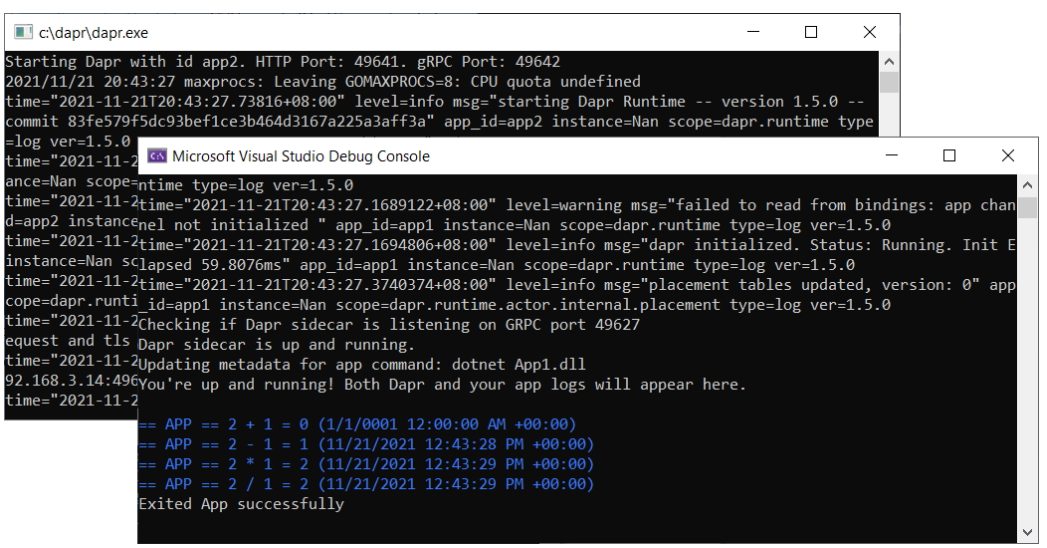

图 1-20　输出结果（3）

1.5.3　状态管理

我们可以借助 Dapr 提供的状态管理组件创建“有状态”的服务。这里的状态并不是存储在应用实例的进程中供其独享，而是存储在独立的存储中（如 Redis）供所有应用实例共享，所以并不会影响水平伸缩的功能。对于上面演示的实例，假设计算服务提供的是 4 个耗时的操作，那么可以将计算结果缓存起来避免重复计算，现在就来实现这样的功能。

为了能够使 API，我们为 App2 添加“Dapr.AspNetCore”这个 NuGet 包的引用。将缓存相关的 3 个操作定义在 IResultCache 接口中。如下面的代码片段所示，IResultCache 接口定义了 3 个方法，GetAsync 方法根据指定的操作/方法名称和输入提取缓存的计算结果，SaveAsync 方法负责将计算结果根据对应的方法名称和输入缓存起来，ClearAsync 方法负责将指定方法的所有缓存结果全部清除掉。

```
public interface IResultCache
{
    Task<Output> GetAsync(string method, Input input);
    Task SaveAsync(string method, Input input, Output output);
    Task ClearAsync(params string[] methods);
}
```

如下所示的 IResultCache 接口的实现类型为 ResultCache。我们在构造函数中注入了 DaprClient 对象，并利用它来完成状态管理的相关操作。计算结果缓存项的 Key 由方法名称和输入参数以“Result_{method}_{X}_{Y}”这样的格式生成，具体的格式化体现在 _keyGenerator 字段返回的委托上。由于涉及对缓存计算结果的清除，我们不得不将所有计算结果缓存项的 Key 也一并缓存起来，该缓存项采用的 Key 为“ResultKeys”。

```
public class ResultCache : IResultCache
{
    private readonly DaprClient                      _daprClient;
    private readonly string                           keyOfKeys = "ResultKeys";
    private readonly string                           storeName = "statestore";
    private readonly Func<string, Input, string>     _keyGenerator;

    public ResultCache(DaprClient daprClient)
    {
         daprClient = daprClient;
        _keyGenerator = (method, input) => $"Result_{method}_{input.X}_{input.Y}";
    }

    public Task<Output> GetAsync(string method, Input input)
    {
        var key =  keyGenerator(method, input);
        return _daprClient.GetStateAsync<Output>(storeName: _storeName, key: key);
    }

    public async Task SaveAsync(string method, Input input, Output output)
    {
        var key = _keyGenerator(method, input);
```

```
        var keys = await _daprClient.GetStateAsync<HashSet<string>>
            (storeName:  storeName,
            key: _keyOfKeys) ?? new HashSet<string>();
        keys.Add(key);

        var operations = new StateTransactionRequest[2];
        var value = Encoding.UTF8.GetBytes(JsonSerializer.Serialize(output));
        operations[0] = new StateTransactionRequest(key: key, value: value,
            operationType: StateOperationType.Upsert);

        value = Encoding.UTF8.GetBytes(JsonSerializer.Serialize(keys));
        operations[1] = new StateTransactionRequest(key: _keyOfKeys, value: value,
            operationType: StateOperationType.Upsert);
        await _daprClient.ExecuteStateTransactionAsync(storeName: _storeName,
            operations: operations);
    }

    public async Task ClearAsync(params string[] methods)
    {
        var keys = await _daprClient.GetStateAsync<HashSet<string>>
            (storeName:  storeName,
            key:  keyOfKeys);
        if (keys != null)
        {
            var selectedKeys = keys
                .Where(it => methods.Any(m => it.StartsWith($"Result_{m}"))).ToArray();
            if (selectedKeys.Length > 0)
            {
                var operations = selectedKeys
                    .Select(it => new StateTransactionRequest(key: it, value: null,
                    operationType: StateOperationType.Delete))
                    .ToList();
                operations.ForEach(it => keys.Remove(it.Key));
                var value = Encoding.UTF8.GetBytes(JsonSerializer.Serialize(keys));
                operations.Add(new StateTransactionRequest(key: _keyOfKeys,
                value: value,
                    operationType: StateOperationType.Upsert));
                await  daprClient.ExecuteStateTransactionAsync(storeName:  storeName,
                    operations: operations);
            }
        }
    }
}
```

在实现的 GetAsync 方法中，我们根据指定的方法名称和输入生成对应缓存项的 Key，并调用 DaprClient 对象的 GetStateAsync<TValue>以提取对应缓存项的值。将 Key 作为该方法的第二个参数，第一个参数表示状态存储（State Store）组件的名称。Dapr 在初始化过程中会默认设置一个针对 Redis 的状态存储组件，并将其命名为“statestore”，ResultCache 使用的正是这个状态存储组件。

对单一状态值进行设置只需要调用 DaprClient 对象的 SaveStateAsync<TValue>方法即可，但是我们实现的 SaveAsync 方法除了需要缓存计算结果，还需要修正“ResultKeys”这个缓存项的值。为了确保两者的一致性，最好在同一个事务中进行两个缓存项的更新，为此需调用 DaprClient 对象的 ExecuteStateTransactionAsync 方法。我们为此创建了两个 StateTransactionRequest 对象来描述对这两个缓存项的更新，具体来说，需要设置缓存项的 Key、Value 和操作类型（Upsert）。这里设置的值必须是最原始的二进制数组，由于状态管理组件默认采用 JSON 的序列化方式和 UTF-8 编码，所以我们按照这样的规则生成了作为缓存值的二进制数组。另一个实现的 ClearAsync 方法采用类似的方式来删除指定方法的计算结果缓存，并修正了“ResultKeys”缓存项的值。

接下来，需要对计算服务的处理方法 Calculate 进行必要的修改。如下面的代码片段所示，直接在 Calculate 方法中注入 IResultCache 对象，如果能够利用该对象提取缓存的计算结果，则直接将它返回给客户端。只有在对应计算结果尚未缓存的情况下，我们才能真正实施计算。在返回计算结果之前，我们会对计算结果实施缓存。Calculate 方法中注入 IResultCache 对象和 DaprClient 对象对应的服务并在 WebApplication 被构建之前进行了服务的注册。

```
using App2;
using Microsoft.AspNetCore.Mvc;
using Shared;
var builder = WebApplication.CreateBuilder(args);
builder.Services
    .AddSingleton<IResultCache, ResultCache>()
    .AddDaprClient();
var app = builder.Build();
app.MapPost("/{method}", Calculate);
app.Run("http://localhost:9999");

static async Task<IResult> Calculate(string method, [FromBody] Input input,
    IResultCache resultCache)
{
    var output = await resultCache.GetAsync(method, input);
    if (output == null)
    {
        var result = method.ToLower() switch
        {
            "add" => input.X + input.Y,
            "sub" => input.X - input.Y,
            "mul" => input.X * input.Y,
            "div" => input.X / input.Y,
            _ => throw new InvalidOperationException($"Invalid operation {method}")
        };
        output = new Output { Result = result };
        await resultCache.SaveAsync(method, input, output);
    }
    return Results.Json(output);
}
```

为了验证利用 Dapr 状态管理组件缓存计算结果的效果，我们对 App1 的程序也进行了相应

的修改。如下面的代码片段所示，对计算服务的 4 个操作进行了两轮调用，并将它们之间的时间间隔设置为 5 秒。

```
using Dapr.Client;
using Shared;

using (var client = DaprClient.CreateInvokeHttpClient(appId: "app2"))
{
    var input = new Input { X = 2, Y = 1 };

    await InvokeAsync();
    await Task.Delay(5000);
    Console.WriteLine();
    await InvokeAsync();

    async Task InvokeAsync()
    {
        await InvokeCoreAsync("add", "+");
        await InvokeCoreAsync("sub", "-");
        await InvokeCoreAsync("mul", "*");
        await InvokeCoreAsync("div", "/");
    }

    async Task InvokeCoreAsync(string method, string @operator)
    {
        var response = await client.PostAsync(method, JsonContent.Create(input));
        var output = await response.EnsureSuccessStatusCode()
            .Content.ReadFromJsonAsync<Output>();
        Console.WriteLine(
            $"{input.X} {@operator} {input.Y} = {output!.Result} ({output.Timestamp})");
    }
}
```

由于两轮服务调用使用相同的输入，如果服务端对计算结果进行了缓存，那么针对同一个方法的调用应该具有相同的时间戳。图 1-21 中的输出结果证实了这一点。（S118）

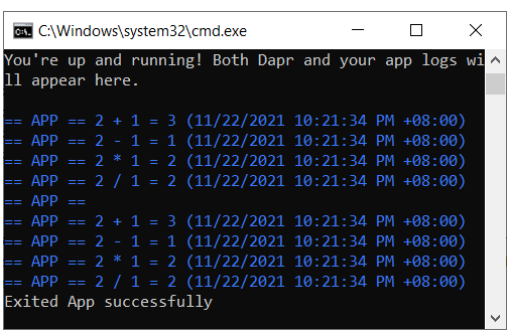

图 1-21　利用状态管理缓存计算结果

1.5.4　发布订阅

Dapr 提供了“开箱即用”的发布订阅（Pub/Sub）模块，我们可以将其作为消息队列来用。上面演示的实例利用状态管理组件缓存了计算结果，现在我们采用发布订阅的方法将指定方法的计算结果缓存清除。具体来说，我们在 App2 中订阅“删除缓存”的主题（Topic），当接收到发布的对应主题的消息时，从消息中提取待删除的方法列表，并将对应的计算结果缓存清除。至于“删除缓存”的主题发布，我们将它交给 App1 来完成。

我们为此对 App2 再次进行修改。如下面的代码片段所示，我们针对路径“clear”注册了一个作为“删除缓存”主题的订阅终节点，它对应的处理方法为 ClearAsync，通过标注在该方法上的 TopicAttribute 来对订阅的主题进行相应设置。该特性构造函数的第一个参数为采用的发布订阅组件名称，我们采用的是初始化 Dapr 时设置的基于 Redis 的发布订阅组件，该组件命名为“pubsub”；第二个参数表示订阅主题的名称，将其设置为“clearresult”。

```
using App2;
using Dapr;
using Microsoft.AspNetCore.Mvc;
using Shared;
var builder = WebApplication.CreateBuilder(args);
builder.Services
    .AddSingleton<IResultCache, ResultCache>()
    .AddDaprClient();
var app = builder.Build();

app.UseCloudEvents();
app.MapPost("clear", ClearAsync);
app.MapSubscribeHandler();

app.MapPost("/{method}", Calculate);
app.Run("http://localhost:9999");

[Topic(pubsubName:"pubsub", name:"clearresult")]
static Task ClearAsync(IResultCache cache, [FromBody] string[] methods)
    => cache.ClearAsync(methods);

static async Task<IResult> Calculate(string method, [FromBody]Input input,
    IResultCache resultCache)
{
    var output = await resultCache.GetAsync(method, input);
    if (output == null)
    {
        var result = method.ToLower() switch
        {
            "add" => input.X + input.Y,
            "sub" => input.X - input.Y,
            "mul" => input.X * input.Y,
            "div" => input.X / input.Y,
```

```
            _ => throw new InvalidOperationException($"Invalid operation {method}")
        };
        output = new Output { Result = result };
        await resultCache.SaveAsync(method, input, output);
    }
    return Results.Json(output);
}
```

使用 ClearAsync 方法定义了两个参数，第一个参数会默认绑定为注册的 IResultCache 服务，第二个参数表示待删除的方法列表。上面标注的 FromBodyAttribute 特性将指导路由系统通过提取请求主体内容来绑定对应参数值。但是 Dapr 的发布订阅组件默认采用 Cloud Events 消息格式，如果请求的主体为具有如此结构的消息，则按照默认的绑定规则，针对 input 参数的绑定将会失败。为此调用 WebApplication 对象的 UseCloudEvents 扩展方法额外注册一个 CloudEventsMiddleware 中间件，该中间件会提取请求数据部分的内容，并使用它将整个请求主体部分的内容替换，那么针对 methods 参数的绑定就能成功了。

我们还调用 WebApplication 对象的 MapSubscribeHandler 扩展方法注册了一个额外的终节点。在启动应用时，Sidecar 会利用这个终节点收集当前应用提供的所有订阅处理器的元数据信息，其中包括发布订阅组件和主题名称，以及调用的路由或路径（对于本实例来说就是“clear”）。当 Sidecar 接收到发布消息后，会根据这组元数据选择匹配的订阅处理器，并利用其提供的路径完成对它的调用。

我们针对发布者的角色对 App1 进行了相应的修改。如下面的代码片段所示，我们利用创建的 DaprClientBuilder 构建了一个 DaprClient 对象。在两轮计算服务的调用之间，调用 DaprClient 的 PublishEventAsync 方法发布了一个名为“clearresult”的消息。从提供的第三个参数可以看出，仅清除了“加法”和“减法”两个方法的计算结果缓存。

```
using Dapr.Client;
using Shared;

var daprClient = new DaprClientBuilder().Build();
using (var client = DaprClient.CreateInvokeHttpClient(appId: "app2"))
{
    var input = new Input { X = 2, Y = 1 };

    await InvokeAsync();
    await daprClient.PublishEventAsync(pubsubName: "pubsub", topicName: "clearresult",
        data: new string[] { "add", "sub" });
    await Task.Delay(5000);
    Console.WriteLine();
    await InvokeAsync();

    async Task InvokeAsync()
    {
        await InvokeCoreAsync("add", "+");
        await InvokeCoreAsync("sub", "-");
        await InvokeCoreAsync("mul", "*");
```

```
        await InvokeCoreAsync("div", "/");
    }

    async Task InvokeCoreAsync(string method, string @operator)
    {
        var response = await client.PostAsync(method, JsonContent.Create(input));
        var output = await response.EnsureSuccessStatusCode()
          .Content.ReadFromJsonAsync<Output>();
        Console.WriteLine(
            $"{input.X} {@operator} {input.Y} = {output!.Result} ({output.Timestamp})");
    }
}
```

图 1-22 所示为利用发布订阅组件删除计算结果缓存。对于两轮间隔为 5 秒的服务调用，由于缓存被清除，加法和减法的计算结果具有不同的时间戳，但乘法和除法的计算结果依旧是相同的。（S119）

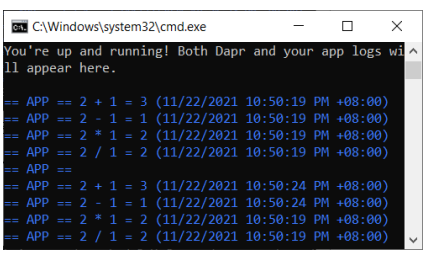

图 1-22　利用发布订阅组件删除计算结果缓存

1.5.5　Actor 模型

如果分布式系统待解决的功能可以分解成若干个很小且状态独立的逻辑单元，则可以考虑使用 Actor 模型（Model）进行设计。具体来说，将上述这些状态逻辑单元定义成单个的 Actor，并在它们之间采用消息驱动的通信方法完成整个工作流程。每个 Actor 只需要考虑对接收的消息进行处理，并将后续的操作转换成消息分发给另一个 Actor 即可。由于每个 Actor 以单线程模式执行，所以我们无须考虑多线程并发和同步的问题。由于 Actor 之间的交互是完全无阻塞的，所以这样做一般能够提高系统整体的吞吐量。

接下来，依然通过对上面演示实例的修改来演示 Dapr 的 Actor 模型在 .NET 下的应用。这次我们将一个具有状态的累加计数器设计成 Actor。在 Shared 项目中为这个 Actor 定义了一个接口，如下面的代码片段所示，这个名为 IAccumulator 的接口派生于 IActor，由于后者来源于“Dapr.Actors”这个 NuGet 包，所以需要添加对应的包引用。IAccumulator 接口定义了两个方法，IncreaseAsync 方法根据指定的数值进行累加并返回当前的值，ResetAsync 方法用于将累加数值重置归零。

```
public interface IAccumulator: IActor
{
    Task<int> IncreaseAsync(int count);
```

```
    Task ResetAsync();
}
```

将 IAccumulator 接口的实现类型 Accumulator 定义在 App2 中。如下面的代码片段所示，除了实现对应的接口，Accumulator 类型还继承了 Actor 这个基类。由于每个 Actor 提供当前累加的值，所以它们是有状态的。但是不能利用 Accumulator 实例的属性来维持这个状态，我们使用从基类继承下来的 StateManager 属性返回的 IActorStateManager 对象来管理当前 Actor 的状态。具体来说，调用 TryGetStateAsync 方法提取当前 Actor 针对指定名称（__counter）的状态值，新的状态值通过调用它的 SetStateAsync 方法进行设置。由于 IActorStateManager 对象的 SetStateAsync 方法对状态进行的更新都是本地操作，最终还需要调用 Actor 对象自身的 SaveStateAsync 方法来提交所有的状态更新。Actor 的状态依旧是通过 Dapr 的状态管理组件进行存储的。

```
public class Accumulator : Actor, IAccumulator
{
    private readonly string  stateName = "  counter";
    public Accumulator(ActorHost host) : base(host)
    {
    }
    public async Task<int> IncreaseAsync(int count)
    {
        var counter = 0;
        var existing = await StateManager.TryGetStateAsync<int>(stateName:  stateName);
        if(existing.HasValue)
        {
            counter = existing.Value;
        }
        counter+= count;
        await StateManager.SetStateAsync(stateName:  stateName, value:counter);
        await SaveStateAsync();
        return counter;
    }
    public async Task ResetAsync()
    {
        await StateManager.TryRemoveStateAsync(stateName:  stateName);
        await SaveStateAsync();
    }
}
```

承载 Actor 相关的 API 由“Dapr.Actors.AspNetCore”这个 NuGet 包提供，所以需要添加该包的引用。Actor 的承载方式与 MVC 框架的承载方式类似，它们都是建立在路由系统上，使用 MVC 框架将所有 Controller 类型转换成注册的终节点，而 Actor 的终节点由 WebApplication 的 MapActorsHandlers 扩展方法进行注册。在注册中间件之前，还需要调用 IServiceCollection 接口的 AddActors 扩展方法将注册的 Actor 类型添加到 ActorRuntimeOptions 配置选项上。

```
using App2;
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddActors(options => options.Actors.RegisterActor<Accumulator>());
```

```
var app = builder.Build();
app.MapActorsHandlers();
app.Run("http://localhost:9999");
```

我们在 App1 中编写程序来演示 Actor 的调用。如下面的代码片段所示，调用 ActorProxy 的静态方法 Create<TActor>创建了两个 IAccumulator 对象。在创建 Actor 对象（其实是调用 Actor 的代理）时需要指定唯一标识 Actor 的 ID（001 和 002）和对应的类型（Accumulator）。

```
using Dapr.Actors;
using Dapr.Actors.Client;
using Shared;

var accumulator1 = ActorProxy.Create<IAccumulator>(new ActorId("001"), "Accumulator");
var accumulator2 = ActorProxy.Create<IAccumulator>(new ActorId("002"), "Accumulator");

while (true)
{
    var counter1 = await accumulator1.IncreaseAsync(1);
    var counter2 = await accumulator2.IncreaseAsync(2);
    await Task.Delay(5000);
    Console.WriteLine($"001: {counter1}");
    Console.WriteLine($"002: {counter2}\n");

    if (counter1 > 10)
    {
        await accumulator1.ResetAsync();
    }
    if (counter2 > 20)
    {
        await accumulator2.ResetAsync();
    }
}
```

Actor 对象创建出来后，在一个循环中采用不同的步长（1 和 2）调用它们的 IncreaseAsync 方法以实施累加操作。在计数器数值达到上限（10 和 20）时，调用它们的 ResetAsync 方法重置计数器。在先后启动 App2 和 App1 之后，App1 所在控制台上将会输出两个累加计数器提供的计数，如图 1-23 所示。（S120）

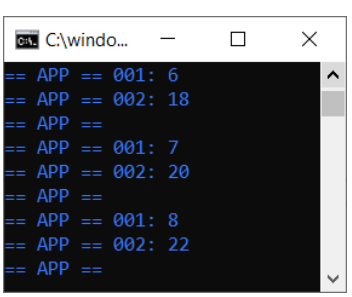

图 1-23　利用 Actor 模式实现的累加计数器

第14章

服务承载

借助 .NET 提供的服务承载（Hosting）系统，我们可以将一个或者多个长时间运行的后台服务寄宿或者承载在创建的应用中。任何需要在后台长时间运行的操作都可以定义成标准化的服务并利用该系统来承载，ASP.NET 应用最终也体现为这样一个承载服务。本章主要介绍“泛化”的服务承载系统，不会涉及任何关于 ASP.NET 的内容。

14.1 服务承载

一个 ASP.NET 应用本质上是一个需要长时间在后台运行的服务，除了这种典型的承载服务，还有很多其他的服务承载需求。接下来就通过一个简单的实例来演示如何承载一个服务来收集当前执行环境的性能指标。

14.1.1 性能指标收集服务

承载服务的项目一般会采用“Microsoft.NET.Sdk.Worker”这个 SDK。服务承载模型涉及的接口和类型基本上定义在“Microsoft.Extensions.Hosting.Abstractions”这个 NuGet 包，而具体实现由“Microsoft.Extensions.Hosting”这个 NuGet 包来提供。下面演示的承载服务会定时采集当前进程的性能指标并将其分发出去。我们只关注处理器使用率、内存使用量和网络吞吐量这 3 种典型的指标，为此定义了如下 PerformanceMetrics 类型。我们并不会实现真正的性能指标收集，定义的 Create 静态方法会利用随机生成的指标来创建 PerformanceMetrics 对象。

```
public class PerformanceMetrics
{
    private static readonly Random  random = new();

    public int         Processor { get; set; }
    public long        Memory { get; set; }
    public long        Network { get; set; }

    public override string ToString()
        => @$"CPU: {Processor * 100}%; Memory: {Memory / (1024* 1024)}M;
```

```
        Network: {Network / (1024 * 1024)}M/s";

    public static PerformanceMetrics Create() => new()
    {
        Processor     =  random.Next(1, 8),
        Memory        = _random.Next(10, 100) * 1024 * 1024,
        Network       = _random.Next(10, 100) * 1024 * 1024
    };
}
```

承载服务通过 IHostedService 接口表示，该接口定义的 StartAsync 方法和 StopAsync 方法可以启动与关闭服务。我们将性能指标采集服务定义成如下 PerformanceMetricsCollector 类型。在实现的 StartAsync 方法中，一个定时器每隔 5 秒调用 Create 静态方法创建一个 PerformanceMetrics 对象，并将它承载的性能指标输出到控制台上。作为定时器的 Timer 对象会在 StopAsync 方法中被释放。

```
public sealed class PerformanceMetricsCollector : IHostedService
{
    private IDisposable?  scheduler;
    public Task StartAsync(CancellationToken cancellationToken)
    {
        _scheduler = new Timer(Callback, null, TimeSpan.FromSeconds(5),
            TimeSpan.FromSeconds(5));
        return Task.CompletedTask;

        static void Callback(object? state)=>
          Console.WriteLine($"[{DateTimeOffset.Now}]{PerformanceMetrics.Create()}");
    }

    public Task StopAsync(CancellationToken cancellationToken)
    {
         scheduler?.Dispose();
        return Task.CompletedTask;
    }
}
```

服务承载系统通过 IHost 接口表示承载服务的宿主，IHost 对象在应用启动过程中采用 Builder 模式由对应的 IHostBuilder 对象来构建。HostBuilder 类型是对 IHostBuilder 接口的默认实现，所以采用如下方式创建一个 HostBuilder 对象，并调用其 Build 方法来提供作为宿主的 IHost 对象。在调用 Build 方法构建 IHost 对象之前，先调用 ConfigureServices 方法将 PerformanceMetricsCollector 注册成针对 IHostedService 接口的服务，并将生命周期模式设置成 Singleton。

```
using App;
new HostBuilder()
    .ConfigureServices(svcs => svcs
        .AddSingleton<IHostedService, PerformanceMetricsCollector>())
    .Build()
    .Run();
```

再调用 Run 方法启动通过 IHost 对象表示的承载服务宿主，进而启动由它承载的 PerformanceMetricsCollector 服务，该服务每隔 5 秒在控制台上输出“采集”的性能指标，如图 14-1 所示。（S1401）[①]

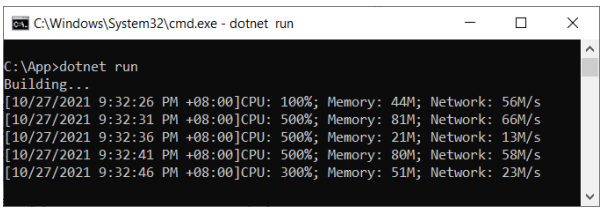

图 14-1　承载指标采集服务

除了采用一般的服务注册方式，我们还可以按照如下方式调用 IServiceCollection 接口的 AddHostedService<THostedService>扩展方法来对承载服务 PerformanceMetricsCollector 进行注册。我们一般也不会通过调用构造函数的方式创建 IHostBuilder 对象，而是使用定义在 Host 类型中的 CreateDefaultBuilder 工厂方法创建 IHostBuilder 对象。

```
using App;
Host.CreateDefaultBuilder(args)
    .ConfigureServices(svcs => svcs.AddHostedService<PerformanceMetricsCollector>())
    .Build()
    .Run();
```

14.1.2　依赖注入

服务承载系统整合依赖注入框架，针对承载服务的注册实际上就是将它注册到依赖注入框架中。既然承载服务实例最终是通过依赖注入容器提供的，那么它自身所依赖的服务当然也可以进行注册。接下来将 PerformanceMetricsCollector 提供的性能指标收集功能分解到由 4 个接口表示的服务中，IProcessorMetricsCollector 接口、IMemoryMetricsCollector 接口和 INetworkMetricsCollector 接口表示的服务分别用于收集 3 种对应的性能指标，而 IMetricsDeliverer 接口表示的服务则用于将收集的性能指标发送出去。

```
public interface IProcessorMetricsCollector
{
    int GetUsage();
}
public interface IMemoryMetricsCollector
{
    long GetUsage();
}
public interface INetworkMetricsCollector
{
    long GetThroughput();
}
```

① 解释见附录 B

```
public interface IMetricsDeliverer
{
    Task DeliverAsync(PerformanceMetrics counter);
}
```

所定义的 MetricsCollector 类型实现了 3 个性能指标采集接口，采集的性能指标直接来源于通过 Create 静态方法创建的 PerformanceMetrics 对象。MetricsDeliverer 类型实现了 IMetricsDeliverer 接口，实现的 DeliverAsync 方法直接将 PerformanceMetrics 对象承载的性能指标输出到控制台上。

```
public class MetricsCollector :
    IProcessorMetricsCollector,
    IMemoryMetricsCollector,
    INetworkMetricsCollector
{
    long INetworkMetricsCollector.GetThroughput()
    => PerformanceMetrics.Create().Network;

    int IProcessorMetricsCollector.GetUsage()
    => PerformanceMetrics.Create().Processor;

    long IMemoryMetricsCollector.GetUsage()
    => PerformanceMetrics.Create().Memory;
}

public class MetricsDeliverer : IMetricsDeliverer
{
    public Task DeliverAsync(PerformanceMetrics counter)
    {
        Console.WriteLine($"[{DateTimeOffset.UtcNow}]{counter}");
        return Task.CompletedTask;
    }
}
```

由于整个性能指标的采集工作被分解到 4 个接口表示的服务中，所以可以采用如下所示的方式重新定义承载服务类型 PerformanceMetricsCollector。在构造函数中注入 4 个依赖服务，StartAsync 方法利用注入的 IProcessorMetricsCollector 对象、IMemoryMetricsCollector 对象和 INetworkMetricsCollector 对象采集对应的性能指标，并利用 IMetricsDeliverer 对象将其发送出去。

```
public sealed class PerformanceMetricsCollector : IHostedService
{
    private readonly IProcessorMetricsCollector           _processorMetricsCollector;
    private readonly IMemoryMetricsCollector               memoryMetricsCollector;
    private readonly INetworkMetricsCollector              networkMetricsCollector;
    private readonly IMetricsDeliverer                    _MetricsDeliverer;
    private IDisposable?                                   scheduler;

    public PerformanceMetricsCollector(
```

```
        IProcessorMetricsCollector processorMetricsCollector,
        IMemoryMetricsCollector memoryMetricsCollector,
        INetworkMetricsCollector networkMetricsCollector,
        IMetricsDeliverer MetricsDeliverer)
    {
        _processorMetricsCollector   = processorMetricsCollector;
        _memoryMetricsCollector      = memoryMetricsCollector;
         networkMetricsCollector     = networkMetricsCollector;
        _MetricsDeliverer            = MetricsDeliverer;
    }

    public Task StartAsync(CancellationToken cancellationToken)
    {
       _scheduler = new Timer(Callback, null, TimeSpan.FromSeconds(5),
           TimeSpan.FromSeconds(5));
       return Task.CompletedTask;

        async void Callback(object? state)
        {
            var counter = new PerformanceMetrics
            {
                Processor     =  processorMetricsCollector.GetUsage(),
                Memory        = _memoryMetricsCollector.GetUsage(),
                Network       =  networkMetricsCollector.GetThroughput()
            };
            await _MetricsDeliverer.DeliverAsync(counter);
        }
    }

    public Task StopAsync(CancellationToken cancellationToken)
    {
        _scheduler?.Dispose();
        return Task.CompletedTask;
    }
}
```

在调用 IHostBuilder 接口的 Build 方法将 IHost 对象创建出来之前，包括承载服务在内的所有服务都可以通过它的 ConfigureServices 方法进行注册。修改后的程序启动之后同样会在控制台上输出图 14-1 所示的结果。（S1402）

```
using App;
var collector = new MetricsCollector();
Host.CreateDefaultBuilder(args)
    .ConfigureServices(svcs => svcs
        .AddHostedService<PerformanceMetricsCollector>()
        .AddSingleton<IProcessorMetricsCollector>(collector)
        .AddSingleton<IMemoryMetricsCollector>(collector)
        .AddSingleton<INetworkMetricsCollector>(collector)
        .AddSingleton<IMetricsDeliverer, MetricsDeliverer>())
    .Build()
    .Run();
```

14.1.3　配置选项

真正的应用开发基本都会使用配置选项，如演示程序中性能指标采集的时间间隔就应该采用配置选项来指定。由于涉及对性能指标数据的发送，所以最好将发送的目标地址定义在配置选项中。如果有多种传输协议可供选择，就可以定义相应的配置选项。 .NET 应用推荐采用 Options 模式来使用配置选项，所以可以定义如下 MetricsCollectionOptions 类型来承载 3 种配置选项。

```
public class MetricsCollectionOptions
{
    public TimeSpan           CaptureInterval { get; set; }
    public TransportType      Transport { get; set; }
    public Endpoint           DeliverTo { get; set; }
}

public enum TransportType
{
    Tcp,
    Http,
    Udp
}

public class Endpoint
{
    public string     Host { get; set; }
    public int        Port { get; set; }
    public override string ToString() => $"{Host}:{Port}";
}
```

传输协议和目标地址被使用在 MetricsDeliverer 服务中，所以对它进行了相应的修改。如下面的代码片段所示，在构造函数中利用注入的 IOptions<MetricsCollectionOptions>服务来提供上面的两个配置选项。在实现的 DeliverAsync 方法中，将采用的传输协议和目标地址输出到控制台上。

```
public class MetricsDeliverer : IMetricsDeliverer
{
    private readonly TransportType    transport;
    private readonly Endpoint         deliverTo;

    public MetricsDeliverer(IOptions<MetricsCollectionOptions> optionsAccessor)
    {
        var options    = optionsAccessor.Value;
         transport     = options.Transport;
         deliverTo     = options.DeliverTo;
    }

    public Task DeliverAsync(PerformanceMetrics counter)
    {
        Console.WriteLine($"[{DateTimeOffset.Now}]Deliver performance counter {counter}
```

```
to
            { deliverTo} via { transport}");
        return Task.CompletedTask;
    }
}
```

承载服务类型 PerformanceMetricsCollector 同样应该采用这种方式来提取表示性能指标采集频率的配置选项。如下所示的代码片段为 PerformanceMetricsCollector 采用配置选项后的完整定义。

```
public sealed class PerformanceMetricsCollector : IHostedService
{
    private readonly IProcessorMetricsCollector            processorMetricsCollector;
    private readonly IMemoryMetricsCollector               _memoryMetricsCollector;
    private readonly INetworkMetricsCollector               networkMetricsCollector;
    private readonly IMetricsDeliverer                      metricsDeliverer;
    private readonly TimeSpan                              _captureInterval;
    private IDisposable?                                    scheduler;

    public PerformanceMetricsCollector(
        IProcessorMetricsCollector processorMetricsCollector,
        IMemoryMetricsCollector memoryMetricsCollector,
        INetworkMetricsCollector networkMetricsCollector,
        IMetricsDeliverer metricsDeliverer,
        IOptions<MetricsCollectionOptions> optionsAccessor)
    {
         processorMetricsCollector   = processorMetricsCollector;
        _memoryMetricsCollector      = memoryMetricsCollector;
         networkMetricsCollector     = networkMetricsCollector;
         metricsDeliverer            = metricsDeliverer;
        _captureInterval             = optionsAccessor.Value.CaptureInterval;
    }

    public Task StartAsync(CancellationToken cancellationToken)
    {
       _scheduler = new Timer(Callback, null, TimeSpan.FromSeconds(5),
           _captureInterval);
       return Task.CompletedTask;

        async void Callback(object? state)
        {
            var counter = new PerformanceMetrics
            {
                Processor     =  processorMetricsCollector.GetUsage(),
                Memory        = _memoryMetricsCollector.GetUsage(),
                Network       =  networkMetricsCollector.GetThroughput()
            };
            await _metricsDeliverer.DeliverAsync(counter);
        }
    }
```

```
    public Task StopAsync(CancellationToken cancellationToken)
    {
        _scheduler?.Dispose();
        return Task.CompletedTask;
    }
}
```

由于配置文件是配置选项的常用来源，所以在根目录下添加了一个名为 appsettings.json 的配置文件，并在其中定义如下内容来提供上述 3 个配置选项。由 Host 类型的 CreateDefaultBuilder 工厂方法创建的 IHostBuilder 对象会自动加载这个配置文件。

```
{
  "MetricsCollection": {
    "CaptureInterval": "00:00:05",
    "Transport": "Udp",
    "DeliverTo": {
      "Host": "192.168.0.1",
      "Port": 3721
    }
  }
}
```

接下来对演示程序进行相应的修改。之前针对依赖服务的注册是通过调用 IHostBuilder 对象的 ConfigureServices 方法利用作为参数的 Action<IServiceCollection>对象完成的，IHostBuilder 接口还有一个 ConfigureServices 重载方法，它的参数类型为 Action<HostBuilderContext, IServiceCollection>，作为输入的 HostBuilderContext 上下文可以提供表示应用配置的 IConfiguration 对象。

```
using App;
var collector = new MetricsCollector();
Host.CreateDefaultBuilder(args)
    .ConfigureServices((context, svcs) => svcs
        .AddHostedService<PerformanceMetricsCollector>()
        .AddSingleton<IProcessorMetricsCollector>(collector)
        .AddSingleton<IMemoryMetricsCollector>(collector)
        .AddSingleton<INetworkMetricsCollector>(collector)
        .AddSingleton<IMetricsDeliverer, MetricsDeliverer>()
        .Configure<MetricsCollectionOptions>(context.Configuration
            .GetSection("MetricsCollection")))
    .Build()
    .Run();
```

我们利用提供的 Action<HostBuilderContext, IServiceCollection>委托对象通过调用 IServiceCollection 接口的 Configure<TOptions>扩展方法从提供的 HostBuilderContext 对象中提取配置，并对 MetricsCollectionOptions 配置选项进行了绑定。修改后的程序运行之后，控制台上的输出结果如图 14-2 所示。（S1403）

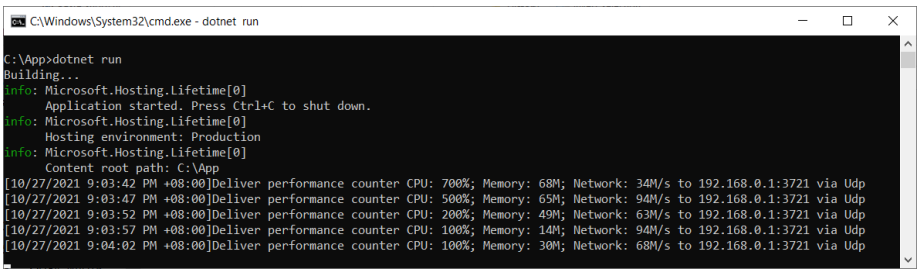

图 14-2　引入配置选项

14.1.4　承载环境

应用程序总是针对某个具体环境进行部署的，开发（Development）、预发（Staging）和产品（Production）是 3 种典型的部署环境，这里的部署环境在服务承载系统中统称为“承载环境”（Hosting Environment）。一般来说，不同的承载环境往往具有不同的配置选项。下面将演示如何为不同的承载环境提供相应的配置选项。“第 5 章　配置选项（上）”已经演示了如何提供针对具体环境的配置文件，具体的做法很简单：将共享或者默认的配置定义在基础配置文件（如 appsettings.json）中，将差异化的部分定义在具体环境的配置文件（如 appsettings.staging. json 和 appsettings.production.json）中。对于演示的实例来说，我们可以采用图 14-3 所示的方式添加额外的两个配置文件来提供针对预发环境和产品环境的差异化配置。

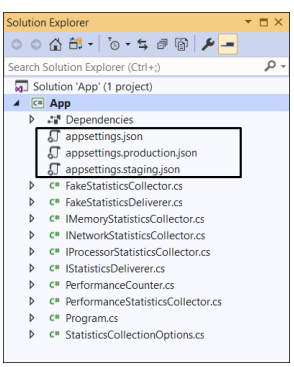

图 14-3　针对承载环境的配置文件

对于演示实例提供的 3 个配置选项来说，假设针对承载环境的差异化配置仅限于发送的目标终节点（IP 地址和端口），就可以采用如下方式将它们定义在针对预发环境的 appsettings. staging.json 和针对产品环境的 appsettings.production.json 中。

appsettings.staging.json：

```
{
  "MetricsCollection": {
    "DeliverTo": {
      "Host": "192.168.0.2",
      "Port": 3721
```

```
    }
  }
}
```

appsettings.production.json：

```
{
  "MetricsCollection": {
    "DeliverTo": {
      "Host": "192.168.0.3",
      "Port": 3721
    }
  }
}
```

由于在调用 Host 的 CreateDefaultBuilder 方法时传入了命令行参数（args），所以默认创建的 IHostBuilder 会将其作为配置源。也正因为如此，可以采用命令行参数的形式设置当前的承载环境（对应配置名称为“environment”）。如图 14-4 所示，分别指定不同的承载环境先后 4 次运行应用程序，从输出的 IP 地址可以看出，应用程序确实是根据当前承载环境加载对应的配置文件的。输出结果还体现了另一个细节，那就是默认采用的是产品环境。（S1404）

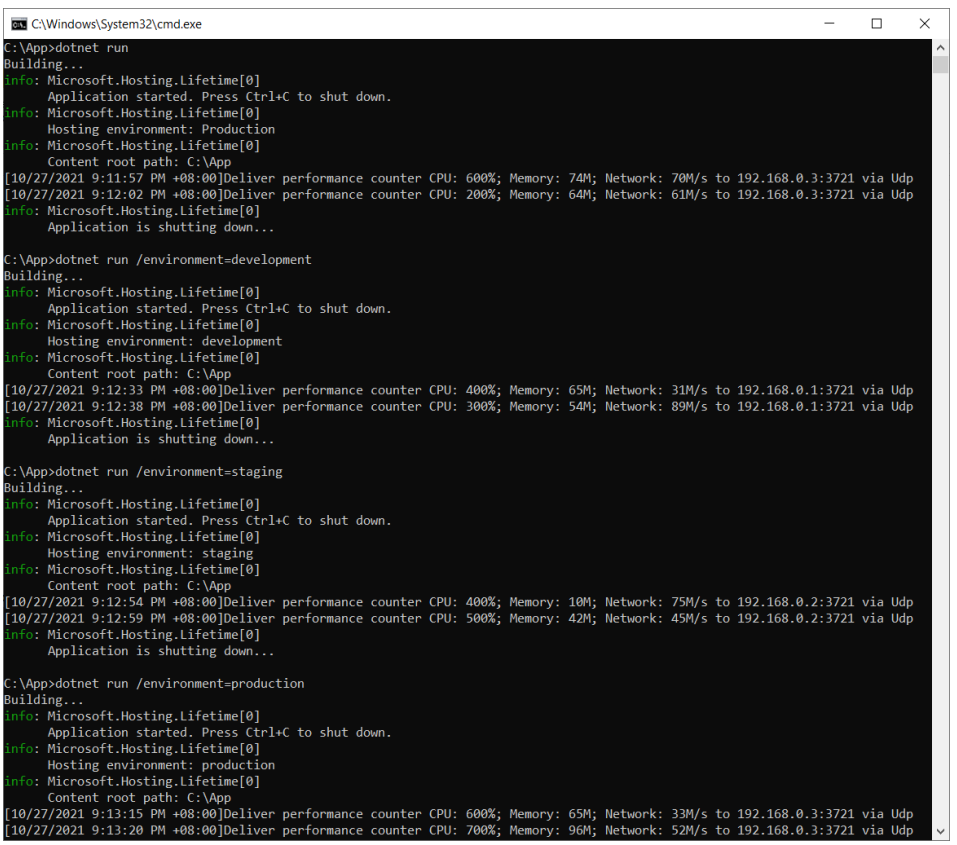

图 14-4　针对承载环境加载配置文件

14.1.5 日志

应用开发中不可避免地会涉及很多针对“诊断日志”的应用，第 7～9 章对这个主题进行了系统而详细的介绍。接下来演示承载服务如何记录日志。对于演示实例来说，用于发送性能指标的 MetricsDeliverer 对象会将收集的指标数据输出到控制台上。下面将这段文字以日志的形式输出，为此我们将这个类型进行了如下修改。

```
public class MetricsDeliverer : IMetricsDeliverer
{
    private readonly TransportType    transport;
    private readonly Endpoint        _deliverTo;
    private readonly ILogger          logger;
    private readonly Action<ILogger, DateTimeOffset, PerformanceMetrics, Endpoint,
        TransportType, Exception?> _logForDelivery;

    public MetricsDeliverer(
        IOptions<MetricsCollectionOptions>          optionsAccessor,
        ILogger<MetricsDeliverer>           logger)
    {
        var options           = optionsAccessor.Value;
         transport            = options.Transport;
         deliverTo            = options.DeliverTo;
        _logger               = logger;
         logForDelivery       = LoggerMessage.Define<DateTimeOffset, PerformanceMetrics,
            Endpoint, TransportType>(LogLevel.Information, 0,
            "[{0}]Deliver performance counter {1} to {2} via {3}");
    }

    public Task DeliverAsync(PerformanceMetrics counter)
    {
         logForDelivery( logger, DateTimeOffset.Now, counter,  deliverTo,  transport,
            null);
        return Task.CompletedTask;
    }
}
```

如上面的代码片段所示，我们利用构造函数中注入的 ILogger<MetricsDeliverer>对象来记录日志。“第 8 章 诊断日志（中）”已经提到，为了避免对同一个消息模板的重复解析，我们可以使用 LoggerMessage 类型定义的委托对象来输出日志，这也是 MetricsDeliverer 采用的编程模式。运行修改后的程序会在控制台上输出图 14-5 所示的结果。由输出结果可以看出，这些文字是由注册的 ConsoleLoggerProvider 提供的 ConsoleLogger 对象输出到控制台上的。由于承载系统自身在进行服务承载过程中也会输出一些日志，所以它们也会输出到控制台上。（S1405）

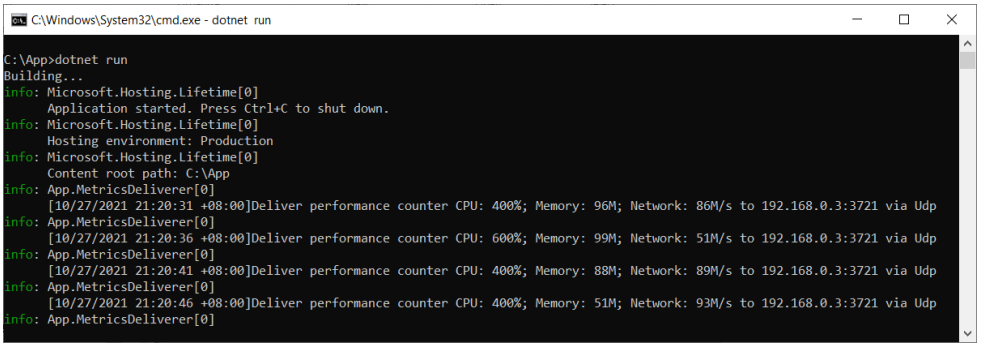

图 14-5　将日志输出到控制台上

如果需要对输出的日志进行过滤，则可以将过滤规则定义在配置文件中。为了避免在产品环境中因输出过多的日志影响性能，可以在 appsettings.production.json 配置文件中以如下形式将类型前缀为“Microsoft.”的日志（最低）等级设置为 Warning。

```
{
  "MetricsCollection": {
    "DeliverTo": {
      "Host": "192.168.0.3",
      "Port": 3721
    }
  },
  "Logging": {
    "LogLevel": {
      "Microsoft": "Warning"
    }
  }
}
```

如果此时分别针对开发环境和产品环境以命令行的形式运行修改后的程序，则可以发现针对开发环境控制台输出类型前缀为“Microsoft.”的日志，但是在针对产品环境的控制台上却找不到它们的踪影，如图 14-6 所示。（S1406）

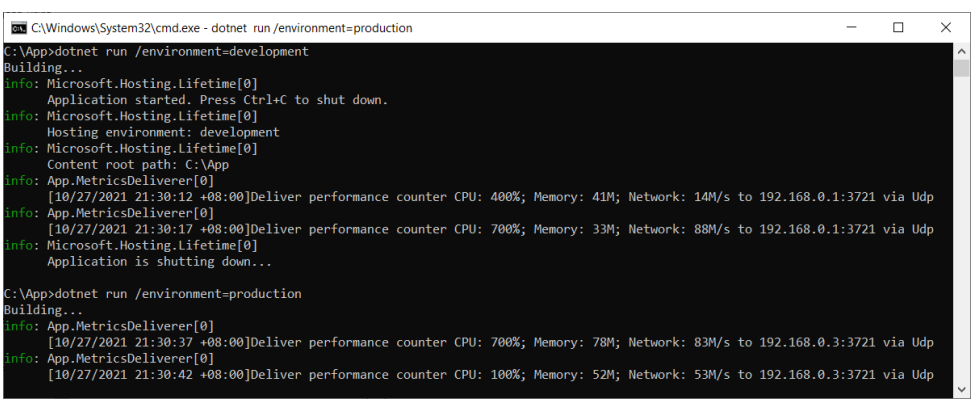

图 14-6　根据承载环境过滤日志

14.2 服务承载模型

服务承载模型主要由 3 个核心对象组成，如图 14-7 所示。多个通过 IHostedService 接口表示的服务被承载（或者寄宿、托管）于通过 IHost 接口表示的宿主上，IHostBuilder 接口表示 IHost 对象的构建者。

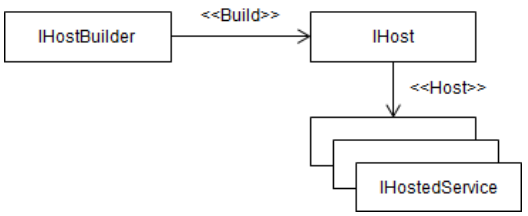

图 14-7 服务承载模型

14.2.1 IHostedService

承载的服务总是会被定义成 IHostedService 接口的实现类型。如下面的代码片段所示，该接口仅定义了两个用来启动和关闭自身服务的方法。当作为宿主的 IHost 对象被启动时，它会激活每个注册的 IHostedService 服务实例，并调用 StartAsync 方法来启动它们。当服务承载应用程序关闭之前，作为服务宿主的 IHost 对象会被关闭，承载的每个 IHostedService 服务对象的 StopAsync 方法也会被调用。

```
public interface IHostedService
{
    Task StartAsync(CancellationToken cancellationToken);
    Task StopAsync(CancellationToken cancellationToken);
}
```

承载系统无缝集成了依赖注入框架，服务承载所需的依赖服务，包括承载服务和它所依赖的服务均由此依赖注入容器提供，承载服务注册的本质就是注册 IHostedService 服务的过程。由于承载服务大多需要长时间运行直到应用被关闭，对应服务注册一般会采用 Singleton 生命周期模式。如下所示的 AddHostedService<THostedService>扩展方法通过调用 TryAddEnumerable 扩展方法来对承载服务进行注册，所以不会出现服务重复注册的问题。

```
public static class ServiceCollectionHostedServiceExtensions
{
    public static IServiceCollection AddHostedService<THostedService>(
        this IServiceCollection services) where THostedService: class, IHostedService
    {
        services.TryAddEnumerable(
            ServiceDescriptor.Singleton<IHostedService, THostedService>());
        return services;
    }
}
```

自定义的承载服务除了直接实现 IHostedService 接口，也可以派生于 BackgroundService 抽

象类型。如下面的代码片段所示，BackgroundService 实现了 IHostedService 接口，实现的 StartAsync 方法会调用自身定义的 ExecuteAsync 抽象方法，所以 BackgroundService 的派生类只需要将具体的承载操作定义在重写的 ExecuteAsync 方法中。

```
public abstract class BackgroundService : IHostedService, IDisposable
{
    private Task                      _executeTask;
    private CancellationTokenSource    stoppingCts;

    public virtual Task ExecuteTask => _executeTask;

    protected abstract Task ExecuteAsync(CancellationToken stoppingToken);

    public virtual Task StartAsync(CancellationToken cancellationToken)
    {
        _stoppingCts = CancellationTokenSource
          .CreateLinkedTokenSource(cancellationToken);
         executeTask = ExecuteAsync( stoppingCts.Token);
        if (_executeTask.IsCompleted)
        {
            return _executeTask;
        }
        return Task.CompletedTask;
    }

    public virtual async Task StopAsync(CancellationToken cancellationToken)
    {
        if ( executeTask == null)
        {
            return;
        }

        try
        {
            _stoppingCts.Cancel();
        }
        finally
        {
            await Task.WhenAny(_executeTask,
                Task.Delay(Timeout.Infinite, cancellationToken)).ConfigureAwait(false);
        }
    }

    public virtual void Dispose() => _stoppingCts?.Cancel();
}
```

14.2.2　IHost

通过 IHostedService 接口表示的服务最终被承载于 IHost 接口表示的宿主上。一般来说，一

个服务承载应用在整个生命周期内只会创建一个 IHost 对象，启动和关闭应用程序本质上就是启动和关闭作为宿主的 IHost 对象。如下面的代码片段所示，IHost 派生于 IDisposable 接口，当它被关闭之后，应用程序还会调用其 Dispose 方法做一些额外的资源释放工作。

```
public interface IHost : IDisposable
{
    IServiceProvider Services { get; }
    Task StartAsync(CancellationToken cancellationToken = default);
    Task StopAsync(CancellationToken cancellationToken = default);
}
```

IHost 接口的 Services 属性返回作为依赖注入容器的 IServiceProvider 对象，该对象提供了服务承载过程中所需的服务实例，其中就包括需要承载的 IHostedService 服务。定义在 IHost 接口中的 StartAsync 方法和 StopAsync 方法完成了针对服务宿主的启动与关闭。

1．IHostApplicationLifetime

前面演示的实例在利用 HostBuilder 对象构建出 IHost 对象之后，并没有调用其 StartAsync 方法启动它，而是调用另一个名为 Run 的扩展方法，该扩展方法涉及服务承载应用程序的生命周期管理。如果要充分理解该扩展方法的本质，就需要先来了解表示承载应用程序生命周期的 IHostApplicationLifetime 对象。如下面的代码片段所示，IHostApplicationLifetime 接口提供了 3 个 CancellationToken 类型的属性，它们都被用来接收应用程序开启与关闭的通知。该接口还提供了一个 StopApplication 方法来关闭应用程序。

```
public interface IHostApplicationLifetime
{
    CancellationToken ApplicationStarted { get; }
    CancellationToken ApplicationStopping { get; }
    CancellationToken ApplicationStopped { get; }

    void StopApplication();
}
```

如下所示的 ApplicationLifetime 类型是对 IHostApplicationLifetime 接口的默认实现。我们可以看到 3 个属性返回的 CancellationToken 对象来源于 3 个对应的 CancellationTokenSource 对象，后者的 Cancle 方法分别在 NotifyStarted 方法、StopApplication 方法和 NotifyStopped 方法中被调用。也就是说，当这 3 个方法将应用程序启动和关闭的通知发送出去后，该通知就能通过 3 个对应的 CancellationToken 对象接收。

```
public class ApplicationLifetime : IHostApplicationLifetime
{
    private readonly ILogger<ApplicationLifetime>            logger;
    private readonly CancellationTokenSource                _startedSource;
    private readonly CancellationTokenSource                 stoppedSource;
    private readonly CancellationTokenSource                 stoppingSource;

    public ApplicationLifetime(ILogger<ApplicationLifetime> logger)
    {
        _startedSource        = new CancellationTokenSource();
```

```
        _stoppedSource        = new CancellationTokenSource();
         stoppingSource       = new CancellationTokenSource();
        _logger               = logger;
    }

    private void ExecuteHandlers(CancellationTokenSource cancel)
    {
        if (!cancel.IsCancellationRequested)
        {
            cancel.Cancel(false);
        }
    }

    public void NotifyStarted()
    {
        try
        {
            ExecuteHandlers( startedSource);
        }
        catch (Exception exception)
        {
             logger.ApplicationError(6, "An error occurred starting the application",
                exception);
        }
    }

    public void NotifyStopped()
    {
        try
        {
            ExecuteHandlers( stoppedSource);
        }
        catch (Exception exception)
        {
            _logger.ApplicationError(8, "An error occurred stopping the application",
                exception);
        }
    }

    public void StopApplication()
    {
        lock ( stoppingSource)
        {
            try
            {
                ExecuteHandlers(_stoppingSource);
            }
            catch (Exception exception)
            {
                _logger.ApplicationError(7,
```

```
                    "An error occurred stopping the application", exception);
            }
        }
    }

    public CancellationToken ApplicationStarted     => _startedSource.Token;
    public CancellationToken ApplicationStopped     => _stoppedSource.Token;
    public CancellationToken ApplicationStopping    =>  stoppingSource.Token;
}
```

接下来通过一个简单的实例演示如何利用IHostApplicationLifetime服务来关闭整个承载应用程序。我们在一个控制台应用程序中定义了如下承载服务类型 FakeHostedService，并在其构造函数中注入了 IHostApplicationLifetime 服务。在得到其 3 个属性返回的 CancellationToken 对象之后，分别在它们上面注册了一个回调并在控制台输出相应的文字。

```
public sealed class FakeHostedService : IHostedService
{
    private readonly IHostApplicationLifetime        lifetime;
    private IDisposable?                             _tokenSource;

    public FakeHostedService(IHostApplicationLifetime lifetime)
    {
         lifetime = lifetime;
         lifetime.ApplicationStarted.Register(() => Console.WriteLine(
            "[{0}]Application started", DateTimeOffset.Now));
         lifetime.ApplicationStopping.Register(() => Console.WriteLine(
            "[{0}]Application is stopping.", DateTimeOffset.Now));
        _lifetime.ApplicationStopped.Register(() => Console.WriteLine(
            "[{0}]Application stopped.", DateTimeOffset.Now));
    }

    public Task StartAsync(CancellationToken cancellationToken)
    {
        _tokenSource = new CancellationTokenSource(TimeSpan.FromSeconds(5))
            .Token.Register( lifetime.StopApplication);
        return Task.CompletedTask;
    }

    public Task StopAsync(CancellationToken cancellationToken)
    {
         tokenSource?.Dispose();
        return Task.CompletedTask;
    }
}
```

在实现的 StartAsync 方法中，我们采用如上方式在等待 5 秒之后调用 IHostApplicationLifetime 对象的 StopApplication 方法关闭应用程序。FakeHostedService 服务最后采用如下方式承载于当前应用程序中。

```
using App;
Host.CreateDefaultBuilder(args)
```

```
    .ConfigureServices(svcs => svcs.AddHostedService<FakeHostedService>())
    .Build()
    .Run();
```

该程序运行之后，控制台上输出的结果如图 14-8 所示，从 3 条消息输出的时间间隔可以确定当前应用程序正是承载 FakeHostedService 通过调用 IHostApplicationLifetime 服务的 StopApplication 方法关闭的。（S1407）

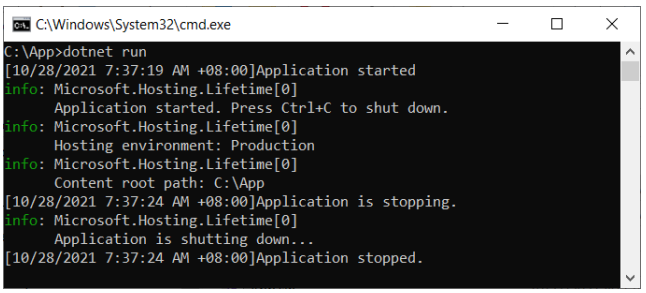

图 14-8　调用 IHostApplicationLifetime 服务关闭应用程序

2．Run 方法

IHost 接口的 Run 方法会在内部调用 IHost 对象的 StartAsync 方法并持续等待。直到接收到来自 IHostApplicationLifetime 服务发出的关闭应用程序通知后，IHost 对象才会调用自身的 StopAsync 方法，此时才会返回 Run 方法的调用。启动 IHost 对象直到应用程序关闭体现在如下 WaitForShutdownAsync 方法上。

```
public static class HostingAbstractionsHostExtensions
{
    public static async Task WaitForShutdownAsync(this IHost host,
        CancellationToken token = default)
    {
        var applicationLifetime = host.Services.GetService<IHostApplicationLifetime>();
        token.Register(state =>((IHostApplicationLifetime)state).StopApplication(),
            applicationLifetime);

        var waitForStop = new TaskCompletionSource<object>(
            TaskCreationOptions.RunContinuationsAsynchronously);
        applicationLifetime.ApplicationStopping.Register(state =>
        {
            var tcs = (TaskCompletionSource<object>)state;
            tcs.TrySetResult(null);
        }, waitForStop);

        await waitForStop.Task;
        await host.StopAsync();
    }
}
```

如下所示的 WaitForShutdown 方法是上面 WaitForShutdownAsync 方法的同步版本。同步的 Run 方法和异步的 RunAsync 方法的实现也体现在下面的代码片段中。下面的代码片段还提供了

Start 方法和 StopAsync 方法的定义，前者可以作为 StartAsync 方法的同步版本，后者可以在关闭 IHost 对象时指定一个超时时限。

```
public static class HostingAbstractionsHostExtensions
{
    public static void WaitForShutdown(this IHost host)
        => host.WaitForShutdownAsync().GetAwaiter().GetResult();

    public static void Run(this IHost host)
        => host.RunAsync().GetAwaiter().GetResult();

    public static async Task RunAsync(this IHost host, CancellationToken token = default)
    {
        try
        {
            await host.StartAsync(token);
            await host.WaitForShutdownAsync(token);
        }
        finally
        {
            host.Dispose();
        }
    }

    public static void Start(this IHost host)
        => host.StartAsync().GetAwaiter().GetResult();

    public static Task StopAsync(this IHost host, TimeSpan timeout)
        => host.StopAsync(new CancellationTokenSource(timeout).Token);
}
```

14.2.3 IHostBuilder

在了解了作为服务宿主的 IHost 对象之后，下面介绍作为构建者的 IHostBuilder 对象。如下面的代码片段所示，IHostBuilder 接口的核心方法 Build 用来提供由它构建的 IHost 对象。它还有一个字典类型的只读属性 Properties，该属性被作为一个共享的数据字典。

```
public interface IHostBuilder
{
    IDictionary<object, object> Properties { get; }
    IHost Build();
    ...
}
```

作为一个典型的设计模式，Builder 模式在最终提供给由它构建的对象之前，一般允许进行相应的前期设置，所以 IHostBuilder 接口提供了一系列的方法为最终构建的 IHost 对象进行相应的设置。具体的设置主要涵盖两个方面：针对配置的设置和针对依赖注入框架的设置。

1．配置

IHostBuilder 接口对配置的设置体现在 ConfigureHostConfiguration 方法和 ConfigureAppConfiguration 方法上，前者涉及的配置主要在服务承载过程中使用，所以是针对服务“宿主（Host）”的配置；后者涉及的配置主要供承载的 IHostedService 服务使用，所以是针对“应用（App）”的配置。针对宿主的配置会被针对应用的配置“继承”下来，应用程序最终得到的配置实际上是两者合并的结果。

```
public interface IHostBuilder
{
    IHostBuilder ConfigureHostConfiguration(
        Action<IConfigurationBuilder> configureDelegate);
    IHostBuilder ConfigureAppConfiguration(
        Action<HostBuilderContext, IConfigurationBuilder> configureDelegate);
    ...
}
```

ConfigureHostConfiguration 方法提供了一个 Action<IConfigurationBuilder>委托对象作为参数。我们可以利用它注册不同的配置源或者实施其他相关的设置（如设置配置文件所在目录的基础路径）。ConfigureAppConfiguration 方法的参数则是 Action<HostBuilderContext, IConfigurationBuilder>，作为第一个参数的 HostBuilderContext 对象携带了与服务承载相关的上下文信息。我们可以利用该上下文信息对配置系统进行针对性设置。

HostBuilderContext 携带的上下文信息主要包含两部分：一是通过 Configuration 属性表示的针对宿主的配置；二是通过 HostingEnvironment 属性表示的承载环境。HostBuilderContext 类型同样具有一个作为共享数据字典的 Properties 属性。

```
public class HostBuilderContext
{
    public IConfiguration                   Configuration { get; set; }
    public IHostEnvironment                 HostingEnvironment { get; set; }
    public IDictionary<object, object>      Properties { get; }

    public HostBuilderContext(IDictionary<object, object> properties);
}
```

ConfigureAppConfiguration 方法使我们可以就当前承载上下文对应用配置进行针对性设置，如针对前期提供承载配置，或者之前添加到 Properties 字典中的某个属性，以及最常见的针对当前的承载环境。如果针对配置系统的设置与当前承载上下文无关，则可以调用如下这个同名的扩展方法，该扩展方法提供的参数依旧是一个 Action<IConfigurationBuilder>对象。

```
public static class HostingHostBuilderExtensions
{
    public static IHostBuilder ConfigureAppConfiguration(this IHostBuilder hostBuilder,
        Action<IConfigurationBuilder> configureDelegate)
    => hostBuilder.ConfigureAppConfiguration((context, builder) =>
        configureDelegate(builder));
}
```

2．承载环境

承载环境通过 IHostEnvironment 接口表示，HostBuilderContext 类型的 HostingEnvironment 属性返回的就是一个 IHostEnvironment 对象。如下面的代码片段所示，除了表示环境名称的 EnvironmentName 属性，IHostEnvironment 接口还定义了一个表示当前应用名称的 ApplicationName 属性。

```
public interface IHostEnvironment
{
    string            EnvironmentName { get; set; }
    string            ApplicationName { get; set; }
    string            ContentRootPath { get; set; }
    IFileProvider     ContentRootFileProvider { get; set; }
}
```

很多的应用程序会涉及一些静态文件，比较典型的就是 Web 应用的 JavaScript、CSS 和图片，这些静态文件被称为“内容文件”（Content File）。IHostEnvironment 接口的 ContentRootPath 属性表示存放这些内容文件的根目录所在的路径。ContentRootFileProvider 属性对应的是指向该路径的 IFileProvider 对象。我们可以利用它获取目录的层次结构，也可以直接利用它来读取文件的内容。

开发、预发和产品是 3 种典型的承载环境，如果严格采用“Development”“Staging”“Production”来对环境进行命名，针对这 3 种承载环境的判断就可以利用 IsDevelopment、IsStaging 和 IsProduction 这 3 个扩展方法来完成。如果需要判断指定的 IHostEnvironment 对象是否属于某个指定的环境，则可以直接调用 IsEnvironment 扩展方法。针对环境名称的比较是不区分字母大小写的。

```
public static class HostEnvironmentEnvExtensions
{
    public static bool IsDevelopment(this IHostEnvironment hostEnvironment)
        =>hostEnvironment.IsEnvironment(Environments.Development);
    public static bool IsStaging(this IHostEnvironment hostEnvironment)
        =>hostEnvironment.IsEnvironment(Environments.Staging);
    public static bool IsProduction(this IHostEnvironment hostEnvironment)
        =>hostEnvironment.IsEnvironment(Environments.Production);

    public static bool IsEnvironment(this IHostEnvironment hostEnvironment,
        string environmentName)
        =>string.Equals(hostEnvironment.EnvironmentName, environmentName,
        StringComparison.OrdinalIgnoreCase);
}

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

IHostEnvironment 对象承载的 3 个属性都是通过配置的形式提供的，对应的配置项名称为“environment”“contentRoot”“applicationName”，它们对应 HostDefaults 类型中的 3 个静态只读字段。我们可以调用 IHostBuilder 接口的 UseEnvironment 方法和 UseContentRoot 方法来设置环境名称与内容文件的根目录的路径。从下面的代码片段可以看出，UseEnvironment 方法调用的依旧是 ConfigureHostConfiguration 方法。如果没有对应用名称做显式设置，入口程序集的名称就会作为当前应用名称。由于一些组件或者框架会假定当前应用名称就是应用所在项目编译后的程序集名称，所以我们一般不会对应用名称进行设置。

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

public static class HostingHostBuilderExtensions
{
    public static IHostBuilder UseEnvironment(this IHostBuilder hostBuilder,
        string environment)
    {
        return hostBuilder.ConfigureHostConfiguration(configBuilder =>
        {
            configBuilder.AddInMemoryCollection(new[]
            {
               new KeyValuePair<string, string>(HostDefaults.EnvironmentKey,environment)
            });
        });
    }
     public static IHostBuilder UseContentRoot(this IHostBuilder hostBuilder,
        string contentRoot)
    {
        return hostBuilder.ConfigureHostConfiguration(configBuilder =>
        {
            configBuilder.AddInMemoryCollection(new[]
            {
                new KeyValuePair<string, string>(HostDefaults.ContentRootKey,
                    contentRoot)
            });
        });
    }
}
```

3．依赖注入

由于包括承载服务（IHostedService）在内的所有依赖服务都由依赖注入框架提供，所以 IHostBuilder 接口提供了更多的方法来注册依赖服务。绝大部分用来注册服务的方法最终都会调用 IHostBuilder 接口的 ConfigureServices 方法，由于该方法提供的参数是一个

Action<HostBuilderContext, IServiceCollection>委托对象，这就意味着服务可以就当前的承载上下文进行针对性注册。如果注册的服务与当前承载上下文无关，就可以调用如下这个同名的扩展方法，该扩展方法提供的参数是一个 Action<IServiceCollection>委托对象。

```
public interface IHostBuilder
{
    IHostBuilder ConfigureServices(
        Action<HostBuilderContext, IServiceCollection> configureDelegate);
    ...
}

public static class HostingHostBuilderExtensions
{
    public static IHostBuilder ConfigureServices(this IHostBuilder hostBuilder,
        Action<IServiceCollection> configureDelegate)
        => hostBuilder.ConfigureServices((context, collection) =>
            configureDelegate(collection));
}
```

IHostBuilder 接口提供了如下两个 UseServiceProviderFactory<TContainerBuilder>重载方法。我们可以利用第一个重载方法注册的 IServiceProviderFactory<TContainerBuilder>对象实现对第三方依赖注入框架的整合。IHostBuilder 接口还定义了 ConfigureContainer<TContainerBuilder>方法来对提供的依赖注入容器进行进一步设置。

```
public interface IHostBuilder
{
    IHostBuilder UseServiceProviderFactory<TContainerBuilder>(
        IServiceProviderFactory<TContainerBuilder> factory);
    IHostBuilder UseServiceProviderFactory<TContainerBuilder>(
        Func<HostBuilderContext, IServiceProviderFactory<TContainerBuilder>> factory);
    IHostBuilder ConfigureContainer<TContainerBuilder>(
        Action<HostBuilderContext, TContainerBuilder> configureDelegate);
}
```

我们认为原生依赖注入框架已经能够满足绝大部分项目的需求，与第三方依赖注入框架的整合其实并没有太大的必要。原生的依赖注入框架利用 DefaultServiceProviderFactory 来提供作为依赖注入容器的 IServiceProvider 对象，针对它的注册由如下两个 UseDefaultServiceProvider 扩展方法来完成。

```
public static class HostingHostBuilderExtensions
{
    public static IHostBuilder UseDefaultServiceProvider(this IHostBuilder hostBuilder,
        Action<ServiceProviderOptions> configure)
       => hostBuilder.UseDefaultServiceProvider((context, options) => configure(options));

    public static IHostBuilder UseDefaultServiceProvider(this IHostBuilder hostBuilder,
        Action<HostBuilderContext, ServiceProviderOptions> configure)
    {
        return hostBuilder.UseServiceProviderFactory(context =>
        {
```

```
            var options = new ServiceProviderOptions();
            configure(context, options);
            return new DefaultServiceProviderFactory(options);
        });
    }
}
```

定义在 IHostBuilder 接口的 ConfigureContainer<TContainerBuilder>方法提供的参数是一个 Action<HostBuilderContext, TContainerBuilder>委托对象。如果针对 TContainerBuilder 的设置与当前承载上下文无关，则可以调用如下简化的 ConfigureContainer<TContainerBuilder>扩展方法，它提供一个 Action<TContainerBuilder>委托对象作为参数。

```
public static class HostingHostBuilderExtensions
{
    public static IHostBuilder ConfigureContainer<TContainerBuilder>(
        this IHostBuilder hostBuilder, Action<TContainerBuilder> configureDelegate)
    {
        return hostBuilder.ConfigureContainer<TContainerBuilder>((context, builder) =>
            configureDelegate(builder));
    }
}
```

4．创建并启动宿主

IHostBuilder 接口还定义了如下 StartAsync 扩展方法，该扩展方法同时完成了 IHost 对象的创建和启动工作，IHostBuilder 接口的另一个 Start 方法是 StartAsync 方法的同步版本。

```
public static class HostingAbstractionsHostBuilderExtensions
{
    public static async Task<IHost> StartAsync(this IHostBuilder hostBuilder,
        CancellationToken cancellationToken = default)
    {
        var host = hostBuilder.Build();
        await host.StartAsync(cancellationToken);
        return host;
    }

    public static IHost Start(this IHostBuilder hostBuilder)
        => hostBuilder.StartAsync().GetAwaiter().GetResult();
}
```

14.3　服务承载流程

上一节着重介绍了组成服务承载模型的 3 个核心对象，接下来从抽象转向具体，介绍服务承载系统模型是如何实现的。要想了解服务承载模型的默认实现，只需要了解 IHost 接口和 IHostBuilder 接口的默认实现类型。由图 14-9 可以看出，IHost 接口和 IHostBuilder 接口的默认实现类型分别是 Host 与 HostBuilder，这两个类型是本节介绍的重点。

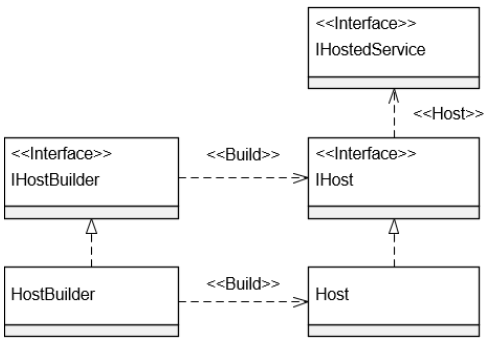

图 14-9　完整的服务承载模型

14.3.1　服务宿主

Host 是对 IHost 接口的默认实现，它仅仅是定义在“Microsoft.Extensions.Hosting”这个 NuGet 包中的一个内部类型。由于承载系统还提供了另一个同名的公共静态类型，在容易出现混淆的地方，我们会将它称为“实例类型 Host”以示区别。在正式介绍 Host 类型的具体实现之前，我们先来认识两个与之相关的类型，其中一个是与承载相关的配置选项 HostOptions，另一个是 IHostLifetime 接口。

如下所示的 HostOptions 类型仅包含 ShutdownTimeout 和 BackgroundServiceExceptionBehavior 两个属性。ShutdownTimeout 属性表示关闭 Host 对象的超时时限，该属性可以通过配置来提供，对应配置节名称为“shutdownTimeoutSeconds”。BackgroundServiceExceptionBehavior 属性表示返回一个同名的枚举，当某个承载服务执行过程中抛出未被处理的异常时，这个属性将用来决定当前承载应用是忽略此异常并继续运行（Ignore）还是立即终止运行。

```
public class HostOptions
{
    public TimeSpan                                    ShutdownTimeout { get; set; }
    public BackgroundServiceExceptionBehavior       BackgroundServiceExceptionBehavior
        { get; set; }

    internal void Initialize(IConfiguration configuration)
    {
        var str = configuration["shutdownTimeoutSeconds"];
        if (!string.IsNullOrEmpty(str) && int.TryParse(
           str, NumberStyles.None, CultureInfo.InvariantCulture, out int num))
        {
            ShutdownTimeout = TimeSpan.FromSeconds(num);
        }
    }
}

public enum BackgroundServiceExceptionBehavior
{
    StopHost,
```

```
    Ignore
}
```

前文已经介绍了一个与承载应用生命周期相关的 IHostApplicationLifetime 接口，Host 类型还涉及另一个与生命周期相关的 IHostLifetime 接口。调用 StartAsync 方法将 Host 对象启动之后，该方法会先调用 IHostLifetime 服务的 WaitForStartAsync 方法。Host 对象的 StopAsync 方法在执行过程中，如果它成功关闭了所有承载的服务，则注册 IHostLifetime 服务的 StopAsync 方法也会被调用。

```
public interface IHostLifetime
{
    Task WaitForStartAsync(CancellationToken cancellationToken);
    Task StopAsync(CancellationToken cancellationToken);
}
```

在前面演示的日志实例中，程序运行后控制台上会输出 3 条级别为 Information 的日志，其中第 1 条日志的内容为“Application started. Press Ctrl+C to shut down.”，后面两条日志内容则是当前承载环境的信息和存放内容文件的根目录路径。当应用程序关闭之前，控制台上还会出现一条内容为“Application is shutting down...”的日志。上述这 4 条日志在控制台上的输出结果如图 14-10 所示。

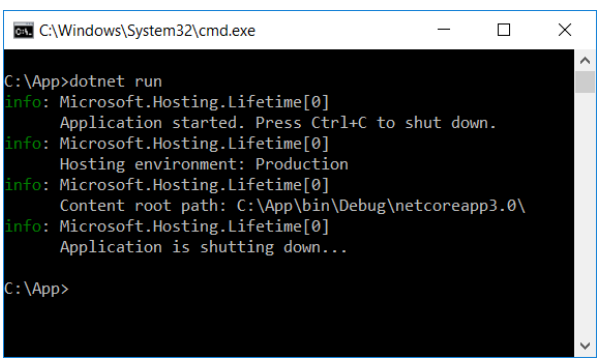

图 14-10　由 ConsoleLifetime 对象输出的日志

图 14-10 所示的 4 条日志都是通过如下 ConsoleLifetime 对象输出的，ConsoleLifetime 类型是对 IHostLifetime 接口的实现。除了以日志的形式输出与当前承载应用程序相关的状态信息，针对 Cancel 按键（Ctrl+C 组合键）的捕捉及随后关闭当前应用程序的功能也在 ConsoleLifetime 类型中实现。ConsoleLifetime 采用的配置选项定义在 ConsoleLifetimeOptions 类型中，该类型唯一的属性 SuppressStatusMessages 用来决定上述 4 条日志是否需要输出，如果不在控制台上输出这些日志，则可以显式将此属性设置为 True。

```
public class ConsoleLifetime : IHostLifetime, IDisposable
{
    public ConsoleLifetime(IOptions<ConsoleLifetimeOptions> options,
        IHostEnvironment environment, IHostApplicationLifetime applicationLifetime);
    public ConsoleLifetime(IOptions<ConsoleLifetimeOptions> options,
        IHostEnvironment environment, IHostApplicationLifetime applicationLifetime,
```

```
        ILoggerFactory loggerFactory);

    public Task StopAsync(CancellationToken cancellationToken);
    public Task WaitForStartAsync(CancellationToken cancellationToken);
    public void Dispose();
}

public class ConsoleLifetimeOptions
{
    public bool SuppressStatusMessages { get; set; }
}
```

下面的代码片段展示的是经过简化的 Host 类型的定义（如忽略了针对承载服务的异常处理）。Host 类型的构造函数中注入了一系列依赖服务，包括作为依赖注入容器的 IServiceProvider 对象、用来记录日志的 ILogger<Host>对象、提供配置选项的 IOptions <HostOptions>对象，以及两个与生命周期相关的 IHostApplicationLifetime 对象和 IHostLifetime 对象。这里提供的 IHostApplicationLifetime 对象的类型必须是 ApplicationLifetime，因为它需要调用 NotifyStarted 方法和 NotifyStopped 方法在应用程序启动与关闭之后向订阅者发送通知，这两个方法并没有定义在 IHostApplicationLifetime 接口中。

```
internal class Host : IHost
{
    private readonly ILogger<Host>            logger;
    private readonly IHostLifetime            hostLifetime;
    private readonly ApplicationLifetime     _applicationLifetime;
    private readonly HostOptions              options;
    private IEnumerable<IHostedService>       hostedServices;

    public IServiceProvider Services { get; }

    public Host(IServiceProvider services, IHostApplicationLifetime applicationLifetime,
        ILogger<Host> logger, IHostLifetime hostLifetime, IOptions<HostOptions> options)
    {
        Services                    = services;
         applicationLifetime        = (ApplicationLifetime)applicationLifetime;
        _logger                     = logger;
        _hostLifetime               = hostLifetime;
         options                    = options.Value);
    }

    public async Task StartAsync(CancellationToken cancellationToken = default)
    {
        await  hostLifetime.WaitForStartAsync(cancellationToken);
        cancellationToken.ThrowIfCancellationRequested();
        _hostedServices = Services.GetService<IEnumerable<IHostedService>>();
        foreach (var hostedService in  hostedServices)
        {
            await hostedService.StartAsync(cancellationToken).ConfigureAwait(false);
        }
```

```
        _applicationLifetime?.NotifyStarted();
    }

    public async Task StopAsync(CancellationToken cancellationToken = default)
    {
        using (var cts = new CancellationTokenSource(_options.ShutdownTimeout))
        using (var linkedCts = CancellationTokenSource.CreateLinkedTokenSource(
            cts.Token, cancellationToken))
        {
            var token = linkedCts.Token;
             applicationLifetime?.StopApplication();
            foreach (var hostedService in _hostedServices.Reverse())
            {
                await hostedService.StopAsync(token).ConfigureAwait(false);
            }

            token.ThrowIfCancellationRequested();
            await  hostLifetime.StopAsync(token);
             applicationLifetime?.NotifyStopped();
         }
    }

    public void Dispose()=>(Services as IDisposable)?.Dispose();
}
```

实现的 StartAsync 方法先调用了 IHostLifetime 对象的 WaitForStartAsync 方法。如果注册的服务类型为 ConsoleLifetime，则输出前面提及的 3 条日志。与此同时，ConsoleLifetime 对象还会注册控制台的按键事件，其目的在于确保在用户按下 Ctrl+C 组合键后应用程序能够被正常关闭。

Host 对象会利用作为依赖注入容器的 IServiceProvider 对象提取表示承载服务的所有 IHostedService 对象，并通过调用其 StartAsync 方法来启动它们。当所有承载的服务正常启动之后，ApplicationLifetime 对象的 NotifyStarted 方法会被调用，此时订阅者会接收到应用程序启动的通知。需要着重指出的是，表示承载服务的 IHostedService 对象是“逐个”（不是并发）被启动的，而且只有等待所有承载服务全部被启动之后，应用程序才算是启动成功。在整个启动过程中，如果利用作为参数的 CancellationToken 接收到取消请求，则启动操作中止。

当 Host 对象的 StopAsync 方法被调用时，它会调用 ApplicationLifetime 对象的 StopApplication 方法对外发出应用程序即将被关闭的通知，此后它会调用每个 IHostedService 对象的 StopAsync 方法。当所有承载服务被成功关闭之后，该方法先后调用 IHostLifetime 对象的 StopAsync 方法和 ApplicationLifetime 对象的 NotifyStopped 方法。在关闭 Host 过程中，如果超出了通过 HostOptions 配置选项设定的超时时限，或者利用作为参数的 CancellationToken 对象接收到取消请求，则整个过程立即中止。

14.3.2 服务承载设置

HostBuilder 类型是对 IHostBuilder 接口的默认实现，上述的 Host 对象就是由它构建的。在实现时旨在对配置进行设置的 ConfigureHostConfiguration 方法和 ConfigureAppConfiguration 方法中，HostBuilder 将提供的委托对象暂存在_configureHostConfigActions 字段和_configureAppConfigActions 字段表示的集合中，它们都将在 Build 方法中被启用。

```
public class HostBuilder : IHostBuilder
{
    private List<Action<IConfigurationBuilder>>  configureHostConfigActions
        = new List<Action<IConfigurationBuilder>>();
    private List<Action<HostBuilderContext, IConfigurationBuilder>>
         configureAppConfigActions = new
        List<Action<HostBuilderContext, IConfigurationBuilder>>();

    public IDictionary<object, object> Properties { get; }
        = new Dictionary<object, object>();

    public IHostBuilder ConfigureHostConfiguration(
        Action<IConfigurationBuilder> configureDelegate)
    {
         configureHostConfigActions.Add(configureDelegate);
        return this;
    }

    public IHostBuilder ConfigureAppConfiguration(
        Action<HostBuilderContext, IConfigurationBuilder> configureDelegate)
    {
        _configureAppConfigActions.Add(configureDelegate);
        return this;
    }
    ...
}
```

与针对配置的设置一样，在 HostBuilder 类型实现的 ConfigureServices 方法中，用来注册依赖服务的 Action<HostBuilderContext, IServiceCollection>委托对象暂存在_configureServicesActions 字段表示的集合中。

```
public class HostBuilder : IHostBuilder
{
    private List<Action<HostBuilderContext, IServiceCollection>>  configureServicesActions
        = new List<Action<HostBuilderContext, IServiceCollection>>();

    public IHostBuilder ConfigureServices(
        Action<HostBuilderContext, IServiceCollection> configureDelegate)
    {
         configureServicesActions.Add(configureDelegate);
        return this;
    }
```

```
    ...
}
```

除了直接调用 IHostBuilder 接口的 ConfigureServices 方法进行服务注册，还可以调用如下扩展方法完成某些特殊服务的注册。两个 ConfigureLogging 重载扩展方法用于注册与日志框架相关的服务，两个 UseConsoleLifetime 重载扩展方法添加的是针对 ConsoleLifetime 服务的注册，两个 RunConsoleAsync 重载扩展方法在此基础上进一步构建并启动作为宿主的 IHost 对象。两个 ConfigureHostOptions 重载扩展方法完成针对 HostOptions 配置选项的设置。

```
public static class HostingHostBuilderExtensions
{
    public static IHostBuilder ConfigureLogging(this IHostBuilder hostBuilder,
        Action<HostBuilderContext, ILoggingBuilder> configureLogging)
    {
        return hostBuilder.ConfigureServices((context, collection) =>
            collection.AddLogging(builder => configureLogging(context, builder)));
    }

    public static IHostBuilder ConfigureLogging(this IHostBuilder hostBuilder,
        Action<ILoggingBuilder> configureLogging)
    {
        return hostBuilder.ConfigureServices((context, collection) =>
            collection.AddLogging(builder => configureLogging(builder)));
    }

    public static IHostBuilder UseConsoleLifetime(this IHostBuilder hostBuilder)
    {
        return hostBuilder.ConfigureServices((context, collection) =>
            collection.AddSingleton<IHostLifetime, ConsoleLifetime>());
    }

    public static IHostBuilder UseConsoleLifetime(this IHostBuilder hostBuilder,
        Action<ConsoleLifetimeOptions> configureOptions)
    {
        return hostBuilder.ConfigureServices((context, collection) =>
        {
            collection.AddSingleton<IHostLifetime, ConsoleLifetime>();
            collection.Configure(configureOptions);
        });
    }

    public static Task RunConsoleAsync(this IHostBuilder hostBuilder,
        CancellationToken cancellationToken = default)
    {
        return hostBuilder.UseConsoleLifetime().Build().RunAsync(cancellationToken);
    }

    public static Task RunConsoleAsync(this IHostBuilder hostBuilder,
        Action<ConsoleLifetimeOptions> configureOptions,
        CancellationToken cancellationToken = default)
```

```
    {
        return hostBuilder.UseConsoleLifetime(configureOptions).Build()
           .RunAsync(cancellationToken);
    }

    public static IHostBuilder ConfigureHostOptions(this IHostBuilder hostBuilder,
        Action<HostBuilderContext, HostOptions> configureOptions) => hostBuilder
        .ConfigureServices((context, collection) => collection.Configure<HostOptions>(
        options => configureOptions(context, options)));

    public static IHostBuilder ConfigureHostOptions(this IHostBuilder hostBuilder,
        Action<HostBuilderContext, HostOptions> configureOptions)
        => hostBuilder.ConfigureServices((context, collection)=> collection
            .Configure<HostOptions>(options => configureOptions(context, options)));
}
```

作为依赖注入容器的 IServiceProvider 对象总是由 IServiceProviderFactory <TContainerBuilder> 工厂创建。由于这是一个泛型对象，所以 HostBuilder 会将它转换成 IServiceFactoryAdapter 接口作为适配。从该接口的定义可以看出，TContainerBuilder 对象仅仅被转换成了 Object 类型。ServiceFactoryAdapter<TContainerBuilder>类型是对 IServiceFactoryAdapter 接口的默认实现。

```
internal interface IServiceFactoryAdapter
{
    object CreateBuilder(IServiceCollection services);
    IServiceProvider CreateServiceProvider(object containerBuilder);
}

internal class ServiceFactoryAdapter<TContainerBuilder> : IServiceFactoryAdapter
{
    private IServiceProviderFactory<TContainerBuilder>       serviceProviderFactory;
    private readonly Func<HostBuilderContext>               _contextResolver;
    private Func<HostBuilderContext, IServiceProviderFactory<TContainerBuilder>>
         factoryResolver;

    public ServiceFactoryAdapter(
        IServiceProviderFactory<TContainerBuilder> serviceProviderFactory)
        =>_serviceProviderFactory = serviceProviderFactory;

    public ServiceFactoryAdapter(Func<HostBuilderContext> contextResolver,
        Func<HostBuilderContext, IServiceProviderFactory<TContainerBuilder>>
        factoryResolver)
    {
         contextResolver = contextResolver;
         factoryResolver = factoryResolver;
    }

    public object CreateBuilder(IServiceCollection services)
        => _serviceProviderFactory?? _factoryResolver(_contextResolver())
           .CreateBuilder(services);
```

```
    public IServiceProvider CreateServiceProvider(object containerBuilder)
        =>  serviceProviderFactory
        .CreateServiceProvider((TContainerBuilder)containerBuilder);
}
```

如下所示的是 HostBuilder 实现的用来注册 IServiceProviderFactory<TContainerBuilder>的两个 UseServiceProviderFactory<TContainerBuilder>方法，它们提供的 IServiceProviderFactory <TContainerBuilder>委托对象和 Func<HostBuilderContext, IServiceProviderFactory <TContainerBuilder>> 委托对象被转换成上面定义的 ServiceFactoryAdapter<TContainerBuilder>类型后通过_serviceProviderFactory 字段暂存起来。如果这两个方法并没有被调用，那么 _serviceProviderFactory 字段返回的将是根据 DefaultServiceProviderFactory 对象创建的 ServiceFactoryAdapter <IServiceCollection>对象，服务承载系统默认使用原生的依赖注入框架就体现在这里。

```
public class HostBuilder : IHostBuilder
{
    private List<IConfigureContainerAdapter>  configureContainerActions =
        new List<IConfigureContainerAdapter>();
    private IServiceFactoryAdapter _serviceProviderFactory =
        new ServiceFactoryAdapter<IServiceCollection>(new DefaultServiceProviderFactory());

    public IHostBuilder UseServiceProviderFactory<TContainerBuilder>(
        IServiceProviderFactory<TContainerBuilder> factory)
    {
        _serviceProviderFactory = new ServiceFactoryAdapter<TContainerBuilder>(factory);
        return this;
    }

    public IHostBuilder UseServiceProviderFactory<TContainerBuilder>(
        Func<HostBuilderContext, IServiceProviderFactory<TContainerBuilder>> factory)
    {
         serviceProviderFactory = new ServiceFactoryAdapter<TContainerBuilder>(
          () =>  hostBuilderContext, factory );
        return this;
    }
}
```

注册的 IServiceProviderFactory<TContainerBuilder>工厂提供的 TContainerBuilder 对象可以通过 ConfigureContainer<TContainerBuilder>方法由提供的 Action<HostBuilderContext, TContainerBuilder>委托对象进行进一步设置。这个泛型的委托对象采用类似的方式转换成 IConfigureContainerAdapter 对象进行适配，如下所示的 ConfigureContainerAdapter <TContainerBuilder>类型是对这个接口的实现。

```
internal interface IConfigureContainerAdapter
{
    void ConfigureContainer(HostBuilderContext hostContext, object containerBuilder);
}
```

```
internal class ConfigureContainerAdapter<TContainerBuilder> : IConfigureContainerAdapter
{
    private Action<HostBuilderContext, TContainerBuilder> _action;

    public ConfigureContainerAdapter(Action<HostBuilderContext, TContainerBuilder> action)
        => _action = action;
    public void ConfigureContainer(HostBuilderContext hostContext, object containerBuilder)
        =>  action(hostContext, (TContainerBuilder)containerBuilder);
}
```

如下所示的代码片段为 ConfigureContainer<TContainerBuilder>方法的实现，该方法会将提供的 Action<HostBuilderContext, TContainerBuilder>对象转换成 ConfigureContainerAdapter<TContainerBuilder>对象，并添加到_configureContainerActions 字段表示的集合中暂存起来。

```
public class HostBuilder : IHostBuilder
{
    private List<IConfigureContainerAdapter> _configureContainerActions =
        new List<IConfigureContainerAdapter>();

    public IHostBuilder ConfigureContainer<TContainerBuilder>(
        Action<HostBuilderContext, TContainerBuilder> configureDelegate)
    {
        _configureContainerActions.Add(
            new ConfigureContainerAdapter<TContainerBuilder>(configureDelegate));
        return this;
    }
    ...
}
```

我们在“第 2 章 依赖注入（上）”中创建了一个名为 Cat 的简易版依赖注入框架，并在“第 3 章 依赖注入（下）”中为其创建了一个 IServiceProviderFactory<TContainerBuilder>实现类型，具体类型为 CatServiceProvider。接下来演示一下如何通过注册 CatServiceProvider 实现与 Cat 这个第三方依赖注入框架的整合。在创建的演示程序中，我们定义了 3 个服务（Foo、Bar 和 Baz）和对应的接口（IFoo、IBar 和 IBaz），并在服务类型上标注 MapToAttribute 特性来定义服务注册信息。

```
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 { }
```

如下所示的 FakeHostedService 类型表示承载的服务。在构造函数中注入 IFoo 对象、IBar 对

象和 IBaz 对象，构造函数提供的调试断言用于验证上述 3 个服务是否被成功注册。

```
public sealed class FakeHostedService: IHostedService
{
    public FakeHostedService(IFoo foo, IBar bar, IBaz baz)
    {
        Debug.Assert(foo != null);
        Debug.Assert(bar != null);
        Debug.Assert(baz != null);
    }
    public Task StartAsync(CancellationToken cancellationToken) => Task.CompletedTask;
    public Task StopAsync(CancellationToken cancellationToken) => Task.CompletedTask;
}
```

在如下演示程序中创建了一个 IHostBuilder 对象，先调用其 ConfigureServices 方法注册了需要承载的 FakeHostedService 服务后，再调用它的 UseServiceProviderFactory 方法完成了对 CatServiceProvider 的注册。随后调用 CatBuilder 的 Register 方法完成了入口程序集的批量服务注册。调用 IHostBuilder 的 Build 方法构建出作为宿主的 IHost 对象并启动它之后，承载的 FakeHostedService 服务将被自动创建并启动。（S1408）

```
using App;
using System.Reflection;

Host.CreateDefaultBuilder()
    .ConfigureServices(svcs => svcs.AddHostedService<FakeHostedService>())
    .UseServiceProviderFactory(new CatServiceProviderFactory())
    .ConfigureContainer<CatBuilder>(
        builder => builder.Register(Assembly.GetEntryAssembly()!))
    .Build()
    .Run();
```

14.3.3　创建宿主

HostBuilder 对象并没有在实现的 Build 方法中调用构造函数来创建 Host 对象，Host 对象是利用作为依赖注入容器的 IServiceProvider 对象创建的。为了可以采用依赖注入容器来提供构建的 Host 对象，HostBuilder 对象必须完成前期的服务注册工作。HostBuilder 对象针对 Host 对象的创建大体可以划分为如下 4 个步骤。

- 创建 HostBuilderContext 上下文对象：首先创建宿主配置的 IConfiguration 对象和表示承载环境的 IHostEnvironment 对象，然后利用两者创建表示承载上下文的 HostBuilderContext 对象。
- 创建应用的配置：创建表示应用配置的 IConfiguration 对象，并用它替换 HostBuilderContext 上下文对象的配置。
- 注册依赖服务：注册依赖服务包括应用程序通过调用 ConfigureServices 方法提供的服务注册和其他一些确保服务承载正常执行的默认服务注册。
- 构建 IServiceProvider 对象，并利用它提供 Host 对象：利用注册的 IServiceProviderFactory

<TContainerBuilder>工厂创建作为依赖注入容器的 IServiceProvider 对象，并利用此对象提供作为宿主的 Host 对象。

1．创建 HostBuilderContext 上下文对象

一个 HostBuilderContext 上下文对象由承载宿主配置的 IConfiguration 对象和描述当前承载环境的 IHostEnvironment 对象组成，后者提供的环境名称、应用名称和内容文件根目录路径可以通过前者来指定，具体的配置项名称定义在如下 HostDefaults 静态类型中。

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

下面通过一个简单的实例演示如何利用配置的方式来指定上述 3 个与承载环境相关的属性。我们定义了一个名为 FakeHostedService 的承载服务，并在构造函数中注入 IHostEnvironment 对象。FakeHostedService 派生于抽象类 BackgroundService，在 ExecuteAsync 方法中将与承载环境相关的环境名称、应用名称和内容文件根目录路径输出到控制台上。

```
public class FakeHostedService : BackgroundService
{
    private readonly IHostEnvironment _environment;
    public FakeHostedService(IHostEnvironment environment)
        =>  environment = environment;
    protected override Task ExecuteAsync(CancellationToken stoppingToken)
    {
        Console.WriteLine("{0,-15}:{1}", nameof( environment.EnvironmentName),
            _environment.EnvironmentName);
        Console.WriteLine("{0,-15}:{1}", nameof( environment.ApplicationName),
             environment.ApplicationName);
        Console.WriteLine("{0,-15}:{1}", nameof(_environment.ContentRootPath),
             environment.ContentRootPath);
        return Task.CompletedTask;
    }
}
```

FakeHostedService 采用如下形式进行承载。为了避免输出日志的“干扰”，调用 IHostBuilder 接口的 ConfigureLogging 扩展方法将注册的 ILoggerProvider 对象全部清除。如果调用 Host 静态类型的 CreateDefaultBuilder 方法时传入当前的命令行参数，则创建的 IHostBuilder 对象会将其作为配置源，以命令行参数的形式来指定承载上下文对象的 3 个属性。

```
using App;
Host.CreateDefaultBuilder(args)
    .ConfigureLogging(logging=>logging.ClearProviders())
    .ConfigureServices(svcs => svcs.AddHostedService<FakeHostedService>())
    .Build()
    .Run();
```

我们采用命令行的方式启动应用程序，并利用传入的命令行参数指定环境名称、应用名称

和内容文件根目录路径（确保路径确实存在）。图 14-11 所示的输出结果表明，应用程序当前的承载环境与基于宿主的配置是一致的。（S1409）

图 14-11　利用配置来初始化承载环境

HostBuilder 针对 HostBuilderContext 上下文对象的创建体现在如下 CreateBuilderContext 方法中。如下面的代码片段所示，该方法创建了一个 ConfigurationBuilder 对象并调用 AddInMemoryCollection 扩展方法注册了内存变量的配置源。接下来它会将这个 ConfigurationBuilder 对象作为参数调用 ConfigureHostConfiguration 方法提供的所有 Action <IConfigurationBuilder>委托对象。ConfigurationBuilder 对象生成的 IConfiguration 对象将作为 HostBuilderContext 上下文对象的配置。

```
public class HostBuilder: IHostBuilder
{
    private List<Action<IConfigurationBuilder>> _configureHostConfigActions ;
    private IConfiguration                       hostConfiguration;

    public IHost Build()
    {
        var buildContext = CreateBuilderContext();
        ...
    }

    private HostBuilderContext CreateBuilderContext()
    {
        //Create Configuration
        var configBuilder = new ConfigurationBuilder().AddInMemoryCollection();
        foreach (var buildAction in _configureHostConfigActions)
        {
            buildAction(configBuilder);
        }
         hostConfiguration = configBuilder.Build();

        //Create HostingEnvironment
        var contentRoot =  hostConfig [HostDefaults.ContentRootKey];
        var contentRootPath = string.IsNullOrEmpty(contentRoot)
            ? AppContext.BaseDirectory
            : Path.IsPathRooted(contentRoot)
            ? contentRoot
```

```
            : Path.Combine(Path.GetFullPath(AppContext.BaseDirectory), contentRoot);
        var hostingEnvironment = new HostingEnvironment()
        {
            ApplicationName = hostConfig [HostDefaults.ApplicationKey],
            EnvironmentName = hostConfig [HostDefaults.EnvironmentKey]
                ?? Environments.Production,
            ContentRootPath = contentRootPath,
        };
        if (string.IsNullOrEmpty(hostingEnvironment.ApplicationName))
        {
            hostingEnvironment.ApplicationName =
                Assembly.GetEntryAssembly()?.GetName().Name;
        }
        hostingEnvironment.ContentRootFileProvider =
            new PhysicalFileProvider(hostingEnvironment.ContentRootPath);

        //Create HostBuilderContext
        return new HostBuilderContext(Properties)
        {
            HostingEnvironment       = hostingEnvironment,
            Configuration            =  hostConfiguration
        };
    }
    ...
}
```

在 HostBuilderContext 上下文对象的配置创建出来后，CreateBuilderContext 方法会根据该配置创建表示承载环境的 HostingEnvironment 对象。如果应用名称的配置不存在，则入口程序集名称将被设置为应用名称。如果内容文件根目录路径对应的配置不存在，当前应用的基础路径就会作为内容文件根目录路径。如果指定的是一个相对路径，HostBuilder 就会根据基础路径生成一个绝对路径作为内容文件根目录路径。CreateBuilderContext 方法最终会根据创建的 HostingEnvironment 对象和之前创建的 IConfiguration 对象将 Host BuilderContext 上下文对象构建出来。

2．构建应用的配置

到目前为止，作为承载上下文的 Host BuilderContext 对象携带的是通过调用 ConfigureHostConfiguration 方法初始化的配置。接下来调用 ConfigureAppConfiguration 方法初始化的配置将与之合并，具体的逻辑体现在 BuildAppConfiguration 方法上。

如下面的代码片段所示，BuildAppConfiguration 方法会创建一个 ConfigurationBuilder 对象，并调用其 AddConfiguration 方法合并现有的配置。与此同时，内容文件根目录的路径被设置为配置文件所在目录的基础路径。BuildAppConfiguration 方法最后会将之前创建的 HostBuilderContext 对象和 ConfigurationBuilder 对象作为参数调用在 ConfigureAppConfiguration 方法提供的每一个 Action<HostBuilderContext, IConfigurationBuilder>委托对象，它们共同完成应用配置的初始化工作。利用 ConfigurationBuilder 对象最终创建的 IConfiguration 对象成为 HostBuilderContext 上下文对象的新配置。

```
public class HostBuilder: IHostBuilder
{
    private List<Action<HostBuilderContext, IConfigurationBuilder>>
        _configureAppConfigActions;

    public IHost Build()
    {
        var buildContext = CreateBuilderContext();
         buildContext.Configuration = BuildAppConfiguration(buildContext);
        ...
    }

    private IConfiguration BuildAppConfiguration(HostBuilderContext buildContext)
    {
        var configBuilder = new ConfigurationBuilder()
            .SetBasePath(buildContext.HostingEnvironment.ContentRootPath)
            .AddConfiguration(buildContext.Configuration,true);
        foreach (var action in  configureAppConfigActions)
        {
            action(_hostBuilderContext, configBuilder);
        }
        return configBuilder.Build();
    }
}
```

3．注册依赖服务

当 HostBuilderContext 上下文对象被创建并初始化后，HostBuilder 需要完成服务注册，其实该服务注册现体现在 ConfigureAllServices 方法中。如下面的代码片段所示，ConfigureAllServices 方法在将 HostBuilderContext 上下文对象和 ServiceCollection 对象作为参数调用 ConfigureServices 方法提供的每一个 Action<HostBuilderContext, IServiceCollection>委托对象之前，它还会注册一些额外的系统服务。ConfigureAllServices 方法最终返回包含所有服务注册的 IServiceCollection 对象。

```
public class HostBuilder: IHostBuilder
{
    private List<Action<HostBuilderContext,IServiceCollection>> _configureServicesActions;
    private IConfiguration                                       hostConfiguration;

    public IHost Build()
    {
        var buildContext = CreateBuilderContext();
        buildContext.Configuration = BuildAppConfiguration(buildContext);
        var services = ConfigureAllServices (buildContext);
        ...
    }

    private IServiceCollection ConfigureAllServices(HostBuilderContext buildContext)
    {
```

```
        var services = new ServiceCollection();
        services.AddSingleton(buildContext);
        services.AddSingleton(buildContext.HostingEnvironment);
        services.AddSingleton(_ => buildContext.Configuration);
        services.AddSingleton<IHostApplicationLifetime, ApplicationLifetime>();
        services.AddSingleton<IHostLifetime, ConsoleLifetime>();
        services.AddSingleton<IHost,Host>();
        services.AddOptions();
        services.Configure<HostOptions>(
            options => options.Initialize( hostConfiguration);
        services.AddLogging();

        foreach (var configureServicesAction in  configureServicesActions)
        {
            configureServicesAction( hostBuilderContext, services);
        }
        return services;
    }
}
```

对于 ConfigureAllServices 方法默认注册的这些服务，可以直接注入承载服务进行消费。由于其中包含了 IHost/Host 的服务注册，所以最终构建的 IServiceProvider 对象可以提供作为服务宿主的 Host 对象。

4．创建 IServiceProvider 对象，并利用它提供 Host 对象

目前，我们已经拥有了所有的服务注册，接下来的任务就是利用它们创建作为依赖注入容器的 IServiceProvider 对象，并由该对象提供构建的 Host 对象。IServiceProvider 对象的创建体现在如下所示的 CreateServiceProvider 方法中。

如下面的代码片段所示，使用 CreateServiceProvider 方法会先得到_serviceProviderFactory 字段表示的 IServiceFactoryAdapter 对象，该对象是根据 UseServiceProviderFactory<TContainerBuilder>方法提供的 IServiceProviderFactory<TContainerBuilder>工厂创建的，调用它的 CreateBuilder 方法可以得到由注册的 IServiceProviderFactory<TContainerBuilder>工厂创建的 TContainerBuilder 对象。

```
public class HostBuilder: IHostBuilder
{
    private List<IConfigureContainerAdapter> _configureContainerActions;
    private IServiceFactoryAdapter _serviceProviderFactory

    public IHost Build()
    {
        var buildContext             = CreateBuilderContext();
        buildContext.Configuration   = BuildAppConfiguration(buildContext);
        var services                 = ConfigureServices(buildContext);
        var serviceProvider          = CreateServiceProvider(buildContext, services);
         return serviceProvider.GetRequiredService<IHost>();
    }

    private IServiceProvider CreateServiceProvider(
```

```
        HostBuilderContext builderContext,IServiceCollection services)
    {
        var containerBuilder = _serviceProviderFactory.CreateBuilder(services);
        foreach (var containerAction in _configureContainerActions)
        {
            containerAction.ConfigureContainer(builderContext, containerBuilder);
        }
        return  serviceProviderFactory.CreateServiceProvider(containerBuilder);
    }
}
```

使用 CreateServiceProvider 方法将 _configureContainerActions 字段集合中每个 IConfigureContainerAdapter 对象提取出来，这里的 IConfigureContainerAdapter 对象是根据 ConfigureContainer<TContainerBuilder>方法提供的 Action<HostBuilderContext, TContainerBuilder> 对象创建的。该方法先将这个 TContainerBuilder 对象作为参数调用它的 ConfigureContainer 方法进行初始化之后，再将它作为参数调用 IServiceFactoryAdapter 对象的 CreateServiceProvider 方法将表示依赖注入容器的 IServiceProvider 对象创建出来。Build 方法最后利用 IServiceProvider 来提供作为宿主的 Host 对象。

14.3.4　静态类型 Host

如果直接利用 Visual Studio 的项目模板来创建一个 ASP.NET Core 应用，就会发现生成的程序采用如下所示的服务承载方式。用来创建宿主的 IHostBuilder 对象是间接地调用静态类型 Host 的 CreateDefaultBuilder 方法创建的，那么这个方法究竟会提供一个什么样的 IHostBuilder 对象呢？

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

如下所示的代码片段是定义在静态类型 Host 中的两个 CreateDefaultBuilder 重载方法的定义，它们最终提供的仍然是一个 HostBuilder 对象，但是在返回该对象之前，调用 ConfigureDefaults 扩展方法完成一些默认初始化工作。

```
public static class Host
{
    // Methods
```

```
    public static IHostBuilder CreateDefaultBuilder() =>
        CreateDefaultBuilder(null);

    public static IHostBuilder CreateDefaultBuilder(string[] args) =>
        new HostBuilder().ConfigureDefaults(args);
}
```

静态类型 Host 调用的 ConfigureDefaults 扩展方法定义如下，该扩展方法会自动将当前目录作为内容文件根目录。它还会调用 HostBuilder 对象的 ConfigureHostConfiguration 方法注册环境变量的配置源，对应环境变量名称的前缀被设置为“DOTNET_”。如果提供了命令行参数，则 ConfigureDefaults 方法还会注册命令行参数的配置源。

```
public static class HostingHostBuilderExtensions
{
    public static IHostBuilder ConfigureDefaults(this IHostBuilder builder,
      string[] args)
    {
        builder.UseContentRoot(Directory.GetCurrentDirectory());

        // 宿主配置
        builder.ConfigureHostConfiguration(config =>
        {
            config.AddEnvironmentVariables(prefix: "DOTNET ");
            if (args is { Length: > 0 })
            {
                config.AddCommandLine(args);
            }
        });

        // 应用配置
        builder.ConfigureAppConfiguration((hostingContext, config) =>
        {
            IHostEnvironment env = hostingContext.HostingEnvironment;
            bool reloadOnChange = GetReloadConfigOnChangeValue(hostingContext);

            config
                .AddJsonFile("appsettings.json", optional: true,
                     reloadOnChange: reloadOnChange)
                .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true,
                     reloadOnChange: reloadOnChange);

            if (env.IsDevelopment() && env.ApplicationName is { Length: > 0 })
            {
                var appAssembly = Assembly.Load(new AssemblyName(env.ApplicationName));
                if (appAssembly is not null)
                {
                    config.AddUserSecrets(appAssembly, optional: true,
                        reloadOnChange: reloadOnChange);
                }
            }
```

```
        config.AddEnvironmentVariables();

        if (args is { Length: > 0 })
        {
            config.AddCommandLine(args);
        }
    });

    // 日志
    builder.ConfigureLogging((hostingContext, logging) =>
    {
        bool isWindows = OperatingSystem.IsWindows();
        if (isWindows)
        {
            logging.AddFilter<EventLogLoggerProvider>(
                level => level >= LogLevel.Warning);
        }

        logging.AddConfiguration(hostingContext.Configuration
          .GetSection("Logging"));
        if (!OperatingSystem.IsBrowser())
        {
            logging.AddConsole();
        }
        logging.AddDebug();
        logging.AddEventSourceLogger();

        if (isWindows)
        {
            logging.AddEventLog();
        }

        logging.Configure(options =>
        {
            options.ActivityTrackingOptions =
                ActivityTrackingOptions.SpanId |
                ActivityTrackingOptions.TraceId |
                ActivityTrackingOptions.ParentId;
        });

    });

    // 依赖注入
    builder.UseDefaultServiceProvider((context, options) =>
    {
        bool isDevelopment = context.HostingEnvironment.IsDevelopment();
        options.ValidateScopes = isDevelopment;
        options.ValidateOnBuild = isDevelopment;
    });
```

```
        return builder;

        static bool GetReloadConfigOnChangeValue(HostBuilderContext hostingContext)
            => hostingContext.Configuration.GetValue(
               "hostBuilder:reloadConfigOnChange", defaultValue: true);
    }
}
```

设置“宿主”配置后，ConfigureDefaults 方法调用 HostBuilder 对象的 ConfigureAppConfiguration 方法设置“应用”配置，配置源包括 JSON 配置文件（appsettings.json 和 appsettings.{environment}.json）、环境变量（没有前缀限制）和命令行参数（如果提供了表示命令行参数的字符串数组）。在注册 JSON 配置文件时，ConfigureDefaults 方法会利用宿主配置“hostBuilder:reloadConfigOnChange”决定是否在文件发生变化之后自动加载新的配置。如果没有提供此项配置，则此项特性是自动开启的。

在完成了配置设置后，ConfigureDefaults 方法还会调用 HostBuilder 对象的 ConfigureLogging 扩展方法进行一些与日志相关的设置，其中包括与应用日志相关的配置（对应配置节名称为“Logging”），以及注册针对控制台（如果不是以“Web Assemby”方式运行）、调试器和 EventSource 与 EventLog（针对 Windows）的日志输出渠道。ConfigureDefaults 方法还通过设置 ActivityTrackingOptions 配置选项对基于活动跟踪的日志范围进行相应设置，作为日志范围被捕捉的内容包括 Activity 的 SpanId、TraceId 和 ParentId。

ConfigureDefaults 方法最后调用 HostBuilder 对象的 UseDefaultServiceProvider 扩展方法对 DefaultServiceProviderFactory 进行了注册。如果当前为开发环境，则 ServiceProviderOptions 配置选项的 ValidateScopes 属性和 ValidateOnBuild 属性均被开启。所以在开发环境中，当作为依赖注入容器的 IServiceProvider 对象被创建之后，系统不仅会进行服务范围的验证，还会验证提供的服务注册最终能否有效地提供具体的实例。由于这两项验证是以牺牲性能为代价的，所以仅限于开发环境。

第15章

应用承载（上）

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 旨在提供极具扩展功能的请求处理管道。我们可以利用管道的定制在它上面构建采用不同编程模式的开发框架。由于这部分内容是本书的核心，所以分为 3 章（第 15 ~ 17 章）对请求处理管道进行全方面介绍。

15.1 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：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进行相应的处理。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，我们先从编程的角度了解 ASP.NET Core 管道式的请求处理方式。

15.1.1 承载方式的变迁

ASP.NET Core 应用本质上就是一个由中间件构成的管道，承载系统将应用承载于一个托管进程中运行，其核心任务就是构建这个管道。从设计模式的角度来讲，“管道”是构建者（Builder）模式最典型的应用场景，所以 ASP.NET Core 先后采用的 3 种承载方式都是应用这种模式。

1．IWebHostBuilder/IWebHost

ASP.NET Core 1.X/2.X 采用的承载模型以 IWebHostBuilder 和 IWebHost 为核心，如图 15-1 所示。IWebHost 对象表示承载 Web 应用的宿主（Host），管道随着 IWebHost 对象的启动被构建。IWebHostBuilder 对象作为宿主对象的构建者，针对管道构建的设置都应用在该对象上面。

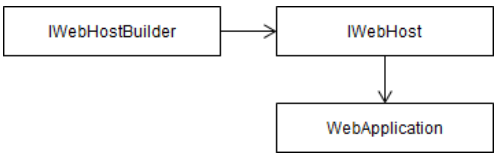

图 15-1　基于 IWebHostBuilder 和 IWebHost 的承载方式

这种“原始”的应用承载方式依然被保留了下来，如下 Hello World 应用程序就是采用的这种承载方式。先创建一个实现了 IWebHostBuilder 接口的 WebHostBuilder 对象，再调用其 UseKestrel 扩展方法注册了一个 Kestrel 服务器。接下来调用它的 Configure 方法利用提供的 Action<IApplicationBuilder>委托对象注册了一个中间件，该中间件将指定的“Hello World”文本作为响应内容。调用 IWebHostBuilder 对象的 Build 方法将作为宿主的 IWebHost 对象构建后，调用其 Run 方法将它启动。此时注册的服务器和中间件组成的管道被构建，服务器开始监听、接收请求，在将请求交给后续的中间件进行处理后，它会将响应回复给客户端。（S1501）

```
new WebHostBuilder()
    .UseKestrel()
    .Configure(app => app.Run(context => context.Response.WriteAsync("Hello World!")))
    .Build()
    .Run();
```

按照“面向接口编程”的原则，其实不应该调用构造函数创建一个“空”的 WebHostBuilder 对象并自行完成针对该对象的所有设置，而是选择按照如下方式调用定义在静态类型 WebHost 中的 CreateDefaultBuilder 工厂方法创建一个具有默认设置的 IWebHostBuilder 对象。由于 Kestrel 服务器的配置就属于“默认设置”的一部分，所以不需要调用 UseKestrel 扩展方法。

```
using Microsoft.AspNetCore;

WebHost.CreateDefaultBuilder()
    .Configure(app => app.Run(context => context.Response.WriteAsync("Hello World!")))
    .Build()
    .Run();
```

如果管道涉及过多的中间件需要注册，则还可以将“中间件注册”这部分工作实现在一个按照约定定义的 Startup 类型中。由于 ASP.NET Core 建立在依赖注入框架之上，所以应用程序往往需要涉及很多服务注册，一般也会将“服务注册”的工作放在这个 Startup 类型中。最终只需要按照如下方式将 Startup 注册到创建的 IWebHostBuilder 对象上。（S1502）

```
using Microsoft.AspNetCore;

WebHost.CreateDefaultBuilder()
    .UseStartup<Startup>()
    .Build()
    .Run();

public class Startup
{
```

```
    public void ConfigureServices(IServiceCollection services)
        => services.AddSingleton<IGreeter, Greeter>();
    public void Configure(IApplicationBuilder app, IGreeter greeter)
        => app.Run(context => context.Response.WriteAsync(greeter.Greet()));
}

public interface IGreeter
{
    string Greet();
}

public class Greeter : IGreeter
{
    public string Greet() => "Hello World!";
}
```

2．IHostBuilder/IHost

除了承载 Web 应用，我们还有很多针对后台服务（比如很多批处理任务）的承载需求，为此微软推出了以 IHostBuilder/IHost 为核心的服务承载系统，“第 14 章　服务承载”已经对该系统进行了详细的介绍。Web 应用本身实际上就是一个长时间运行的后台服务，我们完全可以将应用定义成一个 IHostedService 服务，该类型就是图 15-2 中的 GenericWebHostService。如果将上面介绍的称为第一代应用承载方式，此处介绍的就是第二代应用承载方式。

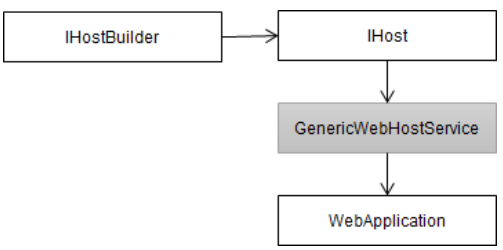

图 15-2　基于 IHostBuilder/IHost 的承载方式

IHostBuilder 接口定义的很多方法（其中很多是扩展方法）旨在完成两个方面的设置：第一，为创建的 IHost 对象及承载的 IHostedService 服务注册依赖服务；第二，为服务承载和应用提供相应的配置。如果采用基于 IWebHostBuilder/IWebHost 的承载方式，则上述这两个方面的设置由 IWebHostBuilder 对象来完成，后者在此基础上还提供了针对中间件的注册。

虽然 IWebHostBuilder 接口提供的除了中间件注册的其他设置基本可以调用 IHostBuilder 接口相应的方法来完成，但是由于 IWebHostBuilder 承载的很多配置都是以扩展方法的形式提供的，所以有必要提供 IWebHostBuilder 接口的兼容。基于 IHostBuilder/IHost 的承载系统中提供对 IWebHostBuilder 接口的兼容是通过如下所示的 ConfigureWebHost 扩展方法完成的，GenericWebHostService 承载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进行一些默认设置（如 KestrelServer）。

```
public static class GenericHostWebHostBuilderExtensions
{
```

```
    public static IHostBuilder ConfigureWebHost(this IHostBuilder builder,
        Action<IWebHostBuilder> configure);
    public static IHostBuilder ConfigureWebHost(this IHostBuilder builder,
        Action<IWebHostBuilder> configure,
        Action<WebHostBuilderOptions> configureWebHostBuilder);
    public static IHostBuilder ConfigureWebHostDefaults(this IHostBuilder builder,
        Action<IWebHostBuilder> configure)
}
```

如果采用基于 IHostBuilder/IHost 的承载方式，则上面演示的“Hello World”应用程序可以被修改成如下形式。在调用 Host 的 CreateDefaultBuilder 工厂方法创建出具有默认设置的 IHostBuilder 对象之后，调用它的 ConfigureWebHostDefaults 扩展方法针对承载 ASP.NET Core 应用的 GenericWebHostService 进行进一步设置。该扩展方法提供的 Action<IApplicationBuilder>委托对象完成了 Startup 类型的注册。（S1503）

```
Host.CreateDefaultBuilder()
    .ConfigureWebHostDefaults(webHostBuilder => webHostBuilder.UseStartup<Startup>())
    .Build()
    .Run();

public class Startup
{
    public void ConfigureServices(IServiceCollection services)
        => services.AddSingleton<IGreeter, Greeter>();
    public void Configure(IApplicationBuilder app, IGreeter greeter)
        => app.Run(context => context.Response.WriteAsync(greeter.Greet()));
}

public interface IGreeter
{
    string Greet();
}

public class Greeter : IGreeter
{
    public string Greet() => "Hello World!";
}
```

3．Minimal API

ASP.NET Core 应用通过 GenericWebHostService 这个承载服务被整合到基于 IHostBuilder/IHost 的服务承载系统中之后，也许微软意识到 Web 应用和后台服务的承载方式还是应该加以区分，而且它们采用的 SDK 都不一样（ASP.NET Core 应用采用的 SDK 为“Microsoft.NET.Sdk.Web”，后台服务采用的 SDK 一般为“Microsoft.NET.Sdk.Worker”），于是推出了基于 WebApplicationBuilder/ WebApplication 的承载方式。但这一次并非又回到了起点，因为底层的承载方式其实没有改变，它只是在上面再封装了一层。

新的应用承载方式依然采用“构建者（Builder）”模式，核心的两个对象分别为 WebApplication 和 WebApplicationBuilder，表示承载应用的 WebApplication 对象由

WebApplicationBuilder 对象构建。我们可以将其称为“第三代应用承载方式”或“Minimal API”。第二代应用承载方式需要提供 IWebHostBuilder 接口的兼容，作为第三代应用承载方式的 Minimal API 则需要同时提供 IWebHostBuilder 接口和 IHostBuilder 接口的兼容，此兼容性是通过这两个接口的实现类型 ConfigureWebHostBuilder 和 ConfigureHostBuilder 完成的。

WebApplicationBuilder 类型的 WebHost 属性和 Host 属性返回了 ConfigureWebHostBuilder 和 ConfigureHostBuilder 这两个对象，之前定义在 IWebHostBuilder 接口和 IHostBuilder 接口上的绝大部分 API（并非所有 API）借助它们得以复用。也正是因为如此，我们会发现相同的功能具有两到三种不同的编程方式。例如，IWebHostBuilder 接口和 IHostBuilder 接口都提供了注册服务的方法，而 WebApplicationBuilder 类型利用 Services 属性直接将存储服务注册的 IServiceCollection 对象暴露，所以任何的服务注册都可以利用这个属性来完成。

```
public sealed class WebApplicationBuilder
{
    public ConfigureWebHostBuilder   WebHost { get; }
    public ConfigureHostBuilder      Host { get; }

    public IServiceCollection        Services { get; }
    public ConfigurationManager      Configuration { get; }
    public ILoggingBuilder           Logging { get; }

    public IWebHostEnvironment       Environment { get; }

    public WebApplication Build();
}

public sealed class ConfigureWebHostBuilder : IWebHostBuilder, ISupportsStartup
public sealed class ConfigureHostBuilder : IHostBuilder, ISupportsConfigureWebHost
```

IWebHostBuilder 接口和 IHostBuilder 接口都提供了设置配置和日志的方法，这两个方面的设置都可以利用 WebApplicationBuilder 的 Configuration 和 Logging 暴露出来的 ConfigurationManager 和 ILoggingBuilder 对象来实现。既然我们采用了 Minimal API，那么应该尽可能使用 WebApplicationBuilder 类型提供的 API。

如果采用这种全新的承载方式，则前面演示的 Hello World 应用程序可以被修改成如下形式。调用定义在 WebApplication 类型中的 CreateBuilder 静态工厂方法根据指定的命令行参数（args）创建一个 WebApplicationBuilder 对象，并调用其 Build 方法构建表示承载 Web 应用的 WebApplication 对象。（S1504）

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();
app.Run(context => context.Response.WriteAsync("Hello World! "));
app.Run();
```

接下来调用它的两个 Run 方法，调用第一个 Run 方法是 IApplicationBuilder 接口（WebApplication 类型实现了该接口）的扩展方法，其目的是注册中间件，调用第二个 Run 方法才是启动 WebApplication 对象表示的应用。由于并没有在 WebApplicationBuilder 对象上进行任

何设置，所以可以按照如下方式调用 WebApplication 的 Create 静态方法将 WebApplication 对象创建出来。

```
var app = WebApplication.Create(args);
app.Run(context => context.Response.WriteAsync("Hello World! "));
app.Run();
```

值得一提的是，之前的两种承载方式都倾向于将初始化操作定义在注册的 Startup 类型中，这种编程在 Mininal API 中不再被支持，所以如下应用程序虽然可以被成功编译，但是在运行时会抛出异常。由于 Minima API 是本书推荐的编程方式，所以后续将不再介绍 Startup 类型的编程模式。

```
var builder = WebApplication.CreateBuilder(args);
builder.WebHost.UseStartup<Startup>();
var app = builder.Build();
app.Run();
```

15.1.2 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，其中位于“龙头”的服务器负责请求的监听、接收、分发和最终的响应，中间件用来完成针对请求的处理。如果读者希望对请求处理管道有一个深刻的认识，就需要对中间件有一定程度的了解。

1．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“双工”管道。服务器将接收的请求消息注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什么请求与响应消息，而是一个通过 HttpContext 表示的上下文对象，我们利用这个上下文对象不仅可以获取当前请求的所有信息，还可以直接完成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 上下文对象，那么一个 Func<HttpContext,Task>委托对象可以表示处理 HttpContext 的操作，或者用于处理 HTTP 请求的处理器（Handler）。由于这个委托对象非常重要，所以 ASP.NET Core 专门定义了如下一个名为 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”，那么中间件能否通过该委托对象来表示呢？

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

2．Func<RequestDelegate, RequestDelegate>

实际上组成请求处理管道的中间件体现为一个 Func<RequestDelegate, RequestDelegate>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可以表示为一个 RequestDelegate 委托对象。对于图 15-3 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 委托对象，该委托对象会作为中间件 Foo 输入，中间件 Foo 借助这个委托对象将当前 HttpContext 分发给后续管道进行进一步处理。中间件的输出依然是一个 RequestDelegate 委托对象，它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 委托对象体现的就是由 Foo、Bar 和 Baz 组成的请求处理管道。

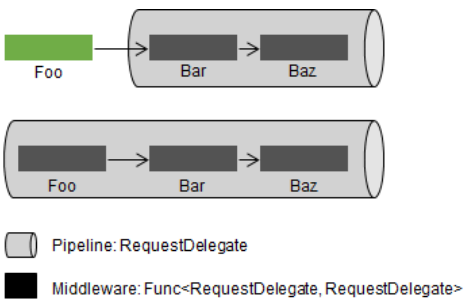

图 15-3　中间件

既然原始的中间件是通过一个 Func<RequestDelegate, RequestDelegate>委托对象来表示的，我们就可以直接注册这样一个委托对象作为中间件。将如下 IApplicationBuilder 接口定义的 Use 方法提供的 Func<RequestDelegate, RequestDelegate>委托对象注册为中间件。表示承载应用的 WebApplication 类型实现了一系列的接口中就有 IApplicationBuilder 接口，这就意味着我们可以将中间件直接注册到 WebApplication 对象上。

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

public sealed class WebApplication :
    IHost,
    IDisposable,
    IApplicationBuilder,
    IEndpointRouteBuilder,
    IAsyncDisposable
{
    IApplicationBuilder IApplicationBuilder.Use(
        Func<RequestDelegate, RequestDelegate> middleware);
    ...
}
```

IApplicationBuilder 接口还定了如下两个 Use 方法，在这里注册的中间件被表示成类型为

Func<HttpContext, Func<Task>, Task>和 Func<HttpContext, RequestDelegate, Task>的委托对象。从两个 Use 方法的实现来看，传入的委托对象最终还是转换成 Func<RequestDelegate, RequestDelegate>对象。

```
public static class UseExtensions
{
    public static IApplicationBuilder Use(this IApplicationBuilder app,
        Func<HttpContext, Func<Task>, Task> middleware)
    {
        return app.Use(next =>
        {
            return context =>
            {
                Func<Task> simpleNext = () => next(context);
                return middleware(context, simpleNext);
            };
        });
    }

    public static IApplicationBuilder Use(this IApplicationBuilder app,
        Func<HttpContext, RequestDelegate, Task> middleware)
    {
        return app.Use(next => context => middleware(context, next));
    }
}
```

如下所示的演示程序，先创建了两个 Func<RequestDelegate, RequestDelegate>委托对象，它们会在响应中写入两个字符串（“Hello”和“World!”）。在创建了表示承载应用的 WebApplication 对象，并将其转换成 IApplicationBuilder 接口后（IApplicationBuilder 接口的 Use 方法在 WebApplication 类型中是显式实现的，所以不得不做这样的类型转换），调用其 Use 方法将这两个委托对象注册为中间件。

```
var app = WebApplication.Create(args);
IApplicationBuilder applicationBuilder = app;
applicationBuilder
    .Use(Middleware1)
    .Use(Middleware2);
app.Run();

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

运行该程序后，我们利用浏览器对应用监听地址（http://localhost:5000）发送请求，两个中间件写入的字符串会以图 15-4 所示的形式呈现出来。（S1505）

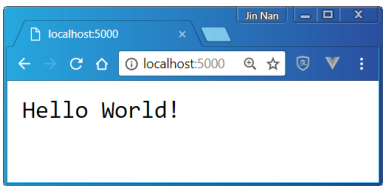

图 15-4　利用注册的中间件处理请求

对于前两代应用承载方式来说，针对中间件的注册可以调用 IWebHostBuilder 接口的 Configure 扩展方法来完成。但是这种方式在 Minimal API 中不再被支持，如果将上面演示的程序修改为如下形式，则针对 Configure 扩展方法的调用将会抛出异常。

```
var builder = WebApplication.CreateBuilder();
builder.WebHost.Configure(app => app
    .Use(Middleware1)
    .Use(Middleware2));
builder.Build().Run();
```

虽然我们可以直接采用原始的 Func<RequestDelegate, RequestDelegate>委托对象来定义中间件，但是在大部分情况下，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。

- 强类型定义：自定义的中间件类型显式实现 IMiddleware 接口，并在实现的 InvokeAsync 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。

3．Run 方法的本质

在演示的 Hello World 应用程序中，调用 IApplicationBuilder 接口的 Run 方法注册了一个 RequestDelegate 对象来处理请求，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“向后传递”，如果调用 IApplicationBuilder 接口的 Run 方法后又注册了其他的中间件，则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);
}
```

15.1.3　定义强类型中间件

如果采用强类型中间件类型定义方式，则只需要实现如下 IMiddleware 接口。该接口定义了唯一的 InvokeAsync 方法来处理请求。这个 InvokeAsync 方法定义了两个参数，前者表示当前 HttpContext 上下文对象，后者表示一个 RequestDelegate 委托对象，它也表示后续中间件组成的管道。如果当前中间件需要将请求分发给后续中间件进行处理，则只需要调用这个委托对象，否则会停止对请求的处理。

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

如下所示的演示程序定义了一个实现 IMiddleware 接口的 StringContentMiddleware 中间件类型，实现的InvokeAsync方法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，所以需要预先对它注册为服务。用于存储服务注册的IServiceCollection对象可以通过 WebApplicationBuilder 的 Services 属性获得，演示程序利用它完成了StringContentMiddleware 的服务注册。由于表示承载应用的 WebApplication 类型实现了IApplicationBuilder 接口，所以直接调用它的 UseMiddleware<TMiddleware>方法来注册中间件类型。（S1506）

```
var builder = WebApplication.CreateBuilder();
builder.Services.AddSingleton<StringContentMiddleware>(
    new StringContentMiddleware("Hello World!"));
var app = builder.Build();
app.UseMiddleware<StringContentMiddleware>();
app.Run();

public 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);
}
```

如下面的代码片段所示，在注册中间件类型时可以将中间件类型设置为泛型参数，也可以调用另一个非泛型的 UseMiddleware 方法将中间件类型作为参数。这两个方法均提供了一个 args 参数，但是该参数是为注册“基于约定的中间件”而设计的，当注册一个实现了 IMiddleware 接口的强类型中间件时是不能指定该参数的。运行该程序后，利用浏览器访问监听地址依然可以得到图15-4 所示的输出结果。

```
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);
}
```

15.1.4 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，其实这种方式有时显得约束过重，不够灵活，基于约定来定义中间件类型更常用。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，而只需要遵循如下这些约定。

- 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，当中间件实例被创建时，表示后续中间件管道的 RequestDelegate 对象将与这个参数进行绑定。构造函数可以包含任意其他参数，RequestDelegate 参数出现的位置也没有限制。
- 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，它们的第一个参数为 HttpContext 上下文对象。约定并未对后续参数进行限制，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。

利用这种方式定义的中间件依然通过前面介绍的 UseMiddleware 方法和 UseMiddleware<TMiddleware>方法进行注册。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时显式指定。

演示实例定义了如下 StringContentMiddleware 类型，它的 InvokeAsync 方法会将预先指定的字符串作为响应内容。StringContentMiddleware 的构造函数定义了 contents 参数和 forewardToNext 参数，前者表示响应内容，后者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>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 forewardToNext 之所以没有每次都显式指定，是因为默认值的存在。（S1507）

```
var app = WebApplication.CreateBuilder().Build();
app
    .UseMiddleware<StringContentMiddleware>("Hello")
    .UseMiddleware<StringContentMiddleware>(" World!", false);
app.Run();

public sealed class StringContentMiddleware
{
    private readonly RequestDelegate  next;
    private readonly string _contents;
    private readonly bool  forewardToNext;

    public StringContentMiddleware(RequestDelegate next, string contents,
        bool forewardToNext = true)
    {
        _next                = next;
         forewardToNext      = forewardToNext;
        _contents            = contents;
    }

    public async Task Invoke(HttpContext context)
    {
        await context.Response.WriteAsync( contents);
        if (_forewardToNext)
        {
            await _next(context);
```

```
        }
    }
}
```

运行该程序后，利用浏览器访问监听地址依然可以得到图 15-4 所示的输出结果。对于前面介绍的定义中间件的方式，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上。强类型方式定义的中间件采用的生命周期取决于对应的服务注册，但是按照约定定义的中间件则总是一个单例对象。

15.2 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）都由表示依赖注入容器的 IServiceProvider 对象提供。在定义承载服务时，我们也可以采用依赖注入方式来消费它所依赖的服务。依赖注入容器能否提供我们需要的服务实例取决于必要的服务注册是否存在。

15.2.1 服务注册

服务注册有 3 种方式。WebApplicationBuilder 的 Host 属性和 WebHost 属性分别用于返回 IHostBuilder 对象和 IWebHostBuilder 对象，可以调用它们的 ConfigureServices 方法进行服务注册。IHostBuilder 接口和 IWebHostBuilder 接口还定义了很多用于服务注册的扩展方法，它们对于 Minimal API 来说绝大部分都是可用的。针对 IHostBuilder 接口的服务注册已经在“第 14 章 服务承载”进行了详细介绍，如下所示为 ConfigureServices 方法在 IWebHostBuilder 接口中的定义。但是既然我们推荐采用 Minimal API，为什么不直接利用 WebApplicationBuilder 的 Services 属性来进行服务注册呢？

```
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; }
}
```

除了可以采用上述 3 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必要的服务，这些公共服务除了供框架使用，也可以供应用程序使用。那么应用程序运行后究竟预先注册了哪些服务？我们编写了如下简单的程序来回答这个问题。

```
using System.Text;

var builder = WebApplication.CreateBuilder();
var app = builder.Build();
app.Run(InvokeAsync);
app.Run();

Task InvokeAsync(HttpContext httpContext)
{
    var sb = new StringBuilder();
    foreach (var service in builder.Services)
    {
        var serviceTypeName = GetName(service.ServiceType);
        var implementationType = service.ImplementationType
            ?? service.ImplementationInstance?.GetType()
            ?? service.ImplementationFactory
            ?.Invoke(httpContext.RequestServices)?.GetType();
        if (implementationType != null)
        {
           sb.AppendLine(@$"{service.Lifetime,-15}{GetName(service.ServiceType),-60}
              { GetName(implementationType)}");
        }
    }
    return httpContext.Response.WriteAsync(sb.ToString());
}

static 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)}>";
}
```

演示程序调用 WebApplication 对象的 Run 方法注册了一个中间件，它会将每个服务对应的声明类型、实现类型和生命周期作为响应内容进行输出。运行这段程序后，系统注册的所有公共服务会以图 15-5 所示的方式输出请求。（S1508）

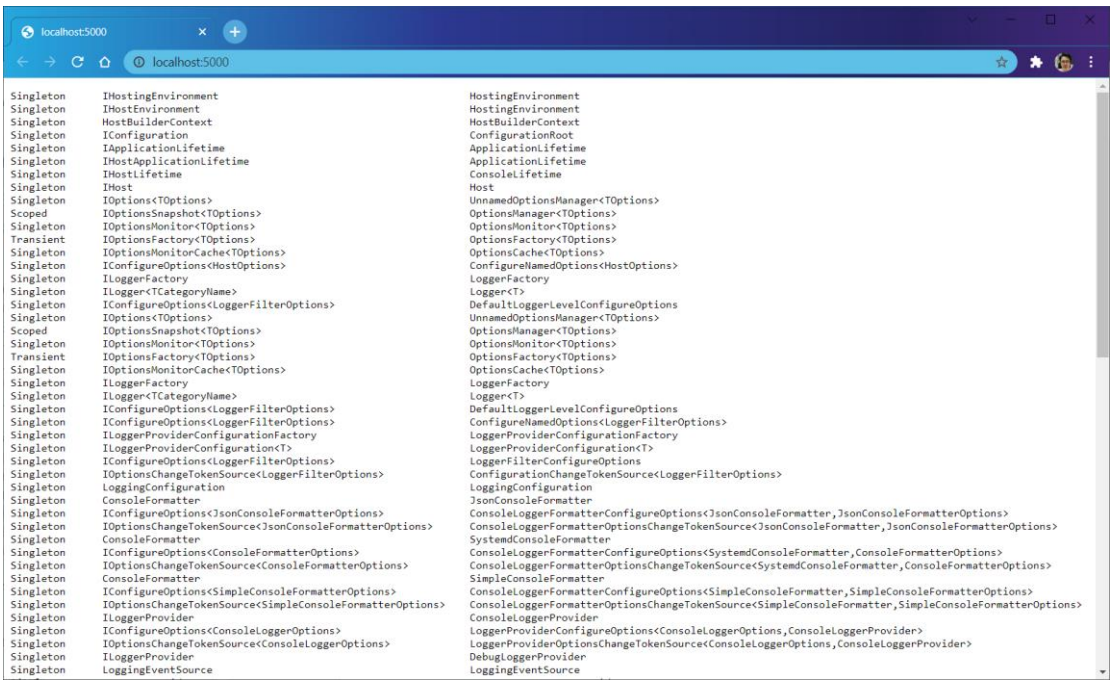

图 15-5　ASP.NET Core 框架注册的公共服务

15.2.2　服务注入

在构造函数或者约定的方法中注入依赖服务对象是主要的服务消费方式。对于以处理管道为核心的 ASP.NET Core 框架来说，依赖注入主要体现在中间件的定义上。由于 ASP.NET Core 框架在创建中间件对象并利用它们构建整个管道时，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采用如下方式注入构造函数中。（S1509）

```
using System.Diagnostics;

var builder = WebApplication.CreateBuilder(args);
builder.Services
    .AddSingleton<FoobarMiddleware>()
    .AddSingleton<Foo>()
    .AddSingleton<Bar>();
var app = builder.Build();
app.UseMiddleware<FoobarMiddleware>();
app.Run();

public class FoobarMiddleware : IMiddleware
{
    public FoobarMiddleware(Foo foo, Bar bar)
    {
        Debug.Assert(foo != null);
        Debug.Assert(bar != null);
```

```
    }

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

public class Foo {}
public class Bar {}
```

上面演示的是强类型中间件的定义方式，如果采用约定方式来定义中间件类型，则依赖服务还可以采用如下方式注入用于处理请求的 InvokeAsync 方法或者 Invoke 方法中。（S1510）

```
using System.Diagnostics;

var builder = WebApplication.CreateBuilder(args);
builder.Services
    .AddSingleton<Foo>()
    .AddSingleton<Bar>();
var app = builder.Build();
app.UseMiddleware<FoobarMiddleware>();
app.Run();

public class FoobarMiddleware
{
    private readonly RequestDelegate _next;
    public FoobarMiddleware(RequestDelegate next) =>  next = next;
    public Task InvokeAsync(HttpContext context, Foo foo, Bar bar)
    {
        Debug.Assert(context != null);
        Debug.Assert(foo != null);
        Debug.Assert(bar != null);
        return  next(context);
    }
}

public class Foo {}
public class Bar {}
```

中间件类型的 InvokeAsync 方法或者 Invoke 方法还具有一个约定，那就是 HttpContext 上下文对象必须作为方法的第一个参数，所以如下中间件类型 FoobarMiddleware 的定义是错误的。与其说是一个约定，还不如说是一个限制，这限制在我们看来毫无意义。对于基于约定的中间件来说，构造函数注入与方法注入在生命周期上存在巨大的差异。由于中间件是一个单例对象，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 方法或者 Invoke 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。

```
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);
}
```

15.2.3 生命周期

当调用 IServiceCollection 相关方法注册服务时，总是会指定一种生命周期。由“第 3 章 依赖注入（下）”的介绍可知，作为依赖注入容器的多个 IServiceProvider 对象通过 IServiceScope 对象表示的服务范围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 服务实例与需要回收释放的 Transient 服务实例则保存在当前 IServiceProvider 对象中，只有不需要回收的 Transient 服务才会用完后就被丢弃。

至于服务实例是否需要回收释放，取决于服务实例的类型是否实现 IDisposable 接口，服务实例的回收释放由保存它的 IServiceProvider 对象负责。当 IServiceProvider 对象自身的 Dispose 方法被调用时，它会调用自身维护的所有待释放实例的 Dispose 方法。对于一个非根容器的 IServiceProvider 对象来说，其生命周期决定于当前的服务范围对象，表示服务范围的 IServiceScope 对象的 Dispose 方法完成对当前范围内的 IServiceProvider 对象的释放。

1．两个 IServiceProvider 对象

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 采用针对应用程序的生命周期，而 Scoped 采用针对请求的生命周期。Singleton 服务的生命周期会一直延续到应用程序被关闭的那一刻，而 Scoped 服务的生命周期仅仅与当前请求绑定在一起，那么这样的生命周期模式是如何实现的呢？

在应用程序正常运行后会创建一个作为根容器的 IServiceProvider 对象，它被称为 ApplicationServices。如果应用程序在处理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 对象创建一个表示服务范围的 IServiceScope 对象，后者会创建一个 IServiceProvider 对象作为子容器。请求处理过程中所需的服务实例均由子容器来提供，它被称为 RequestServices。

针对当前请求的 IServiceScope 对象的 Dispose 方法会在完成请求处理之后被调用，与当前请求绑定的 RequestServices 得以释放。此时由它保存的 Scoped 服务实例和实现了 IDisposable 接口的 Transient 服务实例将变得“无所依托”。在它们变成垃圾对象供 GC 回收之前，实现了 IDisposable 接口的 Scoped 和 Transient 服务实例的 Dispose 方法在 RequestServices 被释放时调用。如下面的代码片段所示，HttpContext 上下文对象的 RequestServices 属性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; }
    ...
}
```

下面的实例使读者对注入服务的生命周期具有更加深刻的认识。首先，定义 Foo、Bar 和 Baz 这 3 个服务类，它们的基类 Base 实现了 IDisposable 接口。然后，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机。

```
var builder = WebApplication.CreateBuilder(args);
builder.Logging.ClearProviders();
builder.Services
    .AddSingleton<Foo>()
    .AddScoped<Bar>()
    .AddTransient<Baz>();

var app = builder.Build();
app.Run(InvokeAsync);
app.Run();

static Task InvokeAsync(HttpContext httpContext)
{
    var path = httpContext.Request.Path;
    var requestServices = httpContext.RequestServices;
    Console.WriteLine($"Receive request to {path}");

    requestServices.GetRequiredService<Foo>();
    requestServices.GetRequiredService<Bar>();
    requestServices.GetRequiredService<Baz>();

    requestServices.GetRequiredService<Foo>();
    requestServices.GetRequiredService<Bar>();
    requestServices.GetRequiredService<Baz>();

    if (path == "/stop")
    {
        requestServices.GetRequiredService<IHostApplicationLifetime>()
            .StopApplication();
    }
    return httpContext.Response.WriteAsync("OK");
}

public class Base : IDisposable
{
    public Base() => Console.WriteLine($"{GetType().Name} is created.");
    public void Dispose() => Console.WriteLine($"{GetType().Name} is disposed.");
}
public class Foo : Base {}
public class Bar : Base {}
public class Baz : Base {}
```

我们采用不同的生命周期对这 3 个服务进行了注册，并将请求处理实现在 InvokeAsync 这个本地方法中。该方法会从 HttpContext 上下文对象中提取 RequestServices，并利用它“两次”提取 3 个服务对应的实例。如果请求路径为“/stop”，则它会采用相同的方式提取 IHostApplicationLifetime 对象，并通过调用其 StopApplication 方法将应用程序关闭。

我们首先采用命令行的形式运行该应用程序，然后利用浏览器依次向该应用程序发送两个请求，采用的路径分别为 “/index”和“ /stop”，控制台上的输出结果如图 15-6 所示。由于 Foo 服务采用的生命周期模式为 Singleton，所以在整个应用程序的生命周期内 Foo 对象只会被创建一次。对于每个接收的请求，虽然 Bar 和 Baz 都被使用了两次，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 Foo 对象会在应用程序关闭时被释放，而生命周期模式分别为 Scoped 和 Transient 的 Bar 对象与 Baz 对象都会在应用程序处理完当前请求之后被释放。（S1511）

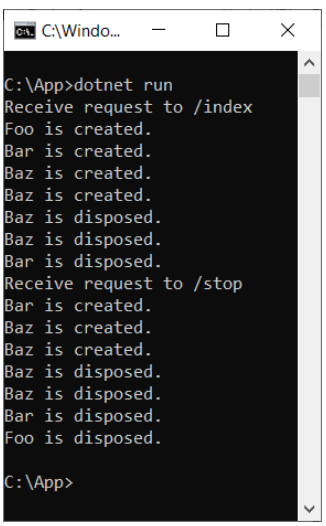

图 15-6　服务实例的生命周期

2．基于服务范围的验证

由“第 3 章 依赖注入（下）”的介绍可知，Scoped 服务既不应该由 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被及时释放。如果忽视了这个问题，就容易造成内存泄漏。下面是一个典型的实例。

下面的演示程序使用的 FoobarMiddleware 中间件需要从数据库中加载由 Foobar 类型表示的数据。这里采用 Entity Framework Core 从 SQL Server 中提取数据，所以我们为实体类型 Foobar 定义了 DbContext（FoobarDbContext），调用 IServiceCollection 接口的 AddDbContext<TDbContext>扩展方法对它以 Scoped 生命周期模式进行了注册。

```
using Microsoft.EntityFrameworkCore;
using System.ComponentModel.DataAnnotations;
```

```
var builder = WebApplication.CreateBuilder(args);
builder.Host.UseDefaultServiceProvider(options => options.ValidateScopes = false);
builder.Services.AddDbContext<FoobarDbContext>(
    options => options.UseSqlServer("{your connection string}"));
var app = builder.Build();
app.UseMiddleware<FoobarMiddleware>();
app.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单例对象，而且它是在应用程序运行时由 ApplicationServices 创建的。由于 FoobarMiddleware 的构造函数中注入了 FoobarDbContext 对象，所以该对象自然也成为一个单例对象，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。

```
using Microsoft.EntityFrameworkCore;
using System.ComponentModel.DataAnnotations;

var builder = WebApplication.CreateBuilder(args);
builder.Host.UseDefaultServiceProvider(options => options.ValidateScopes = true);
builder.Services.AddDbContext<FoobarDbContext>(
    options => options.UseSqlServer("{your connection string}"));
var app = builder.Build();
```

```
app.UseMiddleware<FoobarMiddleware>();
app.Run();
...
```

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，则希望服务实例真正采用基于请求的生命周期模式。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。由“第 14 章　服务承载”的介绍可知，针对服务范围的检验开关可以调用 IHostBuilder 接口的 UseDefaultServiceProvider 扩展方法进行设置。如果采用上面的方式开启针对服务范围的验证，则运行该程序之后会出现图 15-7 所示的异常。由于此验证会影响性能，所以在默认情况下此开关只有在开发环境下才会被开启。（S1512）

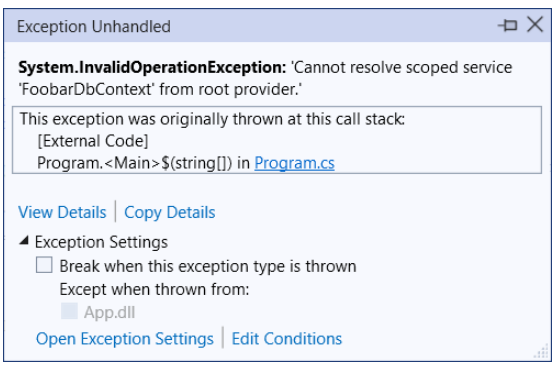

图 15-7　针对 Scoped 服务的验证

如果确实需要在中间件中注入 Scoped 服务，则可以采用强类型定义方式，并对中间件类型以 Scoped 模式进行注册。如果采用基于约定的中间件定义方式，则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方式在 InvokeAsync 方法中利用 RequestServices 来提供依赖服务。

```
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);
    }
}
```

第二种解决方案是按照如下方式直接在 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到，使用 InvokeAsync 方法注入的服务就是由基于当前请求的 RequestServices 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);
    }
}
```

15.3　配置

与前面介绍的服务注册一样，针对配置的设置同样可以采用 3 种不同的编程模式。第一种是利用 WebApplicationBuilder 的 Host 属性返回的 IHostBuilder 对象，它可以设置面向宿主和应用的配置，“第 14 章 服务承载”已经对此进行了详细介绍。IWebHostBuilder 接口上面同样提供了一系列用来对配置进行设置的方法，我们可以将这些方法应用到 WebApplicationBuilder 的 WebHost 属性返回的 IWebHostBuilder 对象上。需要注意的是，既然推荐使用 Mininal API，最好还是采用最新的编程方式。

```
public sealed class WebApplicationBuilder
{
    public ConfigurationManager Configuration { get; }
    ...
}

public sealed class WebApplication :
    IHost, IDisposable, IApplicationBuilder, IEndpointRouteBuilder, IAsyncDisposable
{
    public IConfiguration Configuration { get; }
    ...
}
```

WebApplicationBuilder 的 Configuration 属性用于返回一个 ConfigurationManager 对象。通过“第 5 章 配置选项（上）”中针对配置系统的介绍，我们知道 ConfigurationManager 类型同时实现了 IConfigurationBuilder 接口和 IConfiguration 接口。作为一个 IConfigurationBuilder 对象，它可以被用来注册配置源。作为一个 IConfiguration 对象，它也反映了当前实时的配置状态。WebApplication 对象被 WebApplicationBuilder 构建出来后，将完整的配置固定下来并转移到它的 Configuration 属性上。

15.3.1　初始化配置

当应用程序运行时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，但它只会选择以“ASPNETCORE_”为前缀的环境变量（通过 Host 静态类型的 CreateDefaultBuilder 方法创建的 HostBuilder 默认选择的是以“DOTNET_”为前缀的环境变量）。在演示环境变量的初始化配置之前，需要先解决配置的使用问题，即如何获取配置数据。

如下面的代码片段所示，我们设置两个环境变量，它们的名称分别为“ASPNETCORE_

FOO”和“ASPNETCORE_BAR”。在调用 WebApplication 的 CreateBuilder 方法创建 WebApplicationBuilder 对象后，提取它的 Configuration 属性。经过调试断言后我们可以看出这两个环境变量被成功转移到配置中。表示承载应用的 WebApplication 构建出来后，其 Configuration 属性返回的 IConfiguration 对象上同样包含相同的配置。（S1513）

```
using System.Diagnostics;

Environment.SetEnvironmentVariable("ASPNETCORE_FOO", "123");
Environment.SetEnvironmentVariable("ASPNETCORE BAR", "456");

var builder = WebApplication.CreateBuilder(args);
IConfiguration configuration = builder.Configuration;
Debug.Assert(configuration["foo"] == "123");
Debug.Assert(configuration["bar"] == "456");

var app = builder.Build();
configuration = app.Configuration;
Debug.Assert(configuration["foo"] == "123");
Debug.Assert(configuration["bar"] == "456");
```

15.3.2 以“键-值”对形式读取和修改配置

“第 5 章 配置选项（上）”已经对配置模型进行了深入介绍。我们知道 IConfiguration 对象是以字典的结构来存储配置数据的，可以利用该对象提供的索引以“键-值”对的形式读取和修改配置。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);
    ...
}
```

如下面的代码片段所示，我们可以利用 WebApplicationBuilder 的 WebHost 属性将对应的 IWebHostBuilder 对象提取出来，先通过调用其 GetSetting 方法将以环境变量设置的配置提取出来，再通过调用其 UseSetting 方法将提供的“键-值”对保存到应用的配置中。配置最终的状态被固定下来后转移到构建的 WebApplication 对象上。（S1514）

```
using System.Diagnostics;

var builder = WebApplication.CreateBuilder(args);
builder.WebHost.UseSetting("foo", "abc");
builder.WebHost.UseSetting("bar", "xyz");

Debug.Assert(builder.WebHost.GetSetting("foo") == "abc");
Debug.Assert(builder.WebHost.GetSetting("bar") == "xyz");

IConfiguration configuration = builder.Configuration;
```

```
Debug.Assert(configuration["foo"] == "abc");
Debug.Assert(configuration["bar"] == "xyz");

var app = builder.Build();
configuration = app.Configuration;
Debug.Assert(configuration["foo"] == "abc");
Debug.Assert(configuration["bar"] == "xyz");
```

15.3.3　注册配置源

配置系统最大的特点是可以注册不同的配置源。针对配置源的注册同样可以利用不同的编程方式来实现，其中一种就是利用 WebApplicationBuilder 的 Host 属性返回的 IHostBuilder 对象，并调用其 ConfigureHostConfiguration 方法和 ConfigureAppConfiguration 方法完成宿主和应用的配置，其中包含配置源的注册。IWebHostBuilder 接口也提供如下等效的 ConfigureAppConfiguration 方法。该方法提供的参数是一个 Action<WebHostBuilderContext, IConfigurationBuilder>委托对象，这就意味着我们可以承载上下文对象并对配置进行针对性设置。如果提供的设置与当前承载上下文对象无关，则可以调用另一个参数类型为 Action <IConfigurationBuilder>的 ConfigureAppConfiguration 重载方法。

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

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

我们可以利用 WebApplicationBuilder 的 WebHost 属性返回对应的 IWebHostBuilder 对象，并采用如下方式利用 IWebHostBuilder 对象注册配置源。（S1515）

```
using System.Diagnostics;

var builder = WebApplication.CreateBuilder(args);
builder.WebHost.ConfigureAppConfiguration(config
    => config.AddInMemoryCollection(new Dictionary<string, string>
    {
        ["foo"] = "123",
        ["bar"] = "456"
    }));
var app = builder.Build();
Debug.Assert(app.Configuration["foo"] == "123");
Debug.Assert(app.Configuration["bar"] == "456");
```

由于 WebApplicationBuilder 的 Configuration 属性返回的 ConfigurationManager 自身就是一个 IConfigurationBuilder 对象，所以直接按照如下方式将配置源注册到它上面，这也是我们推荐的编程方式。值得一提的是，如果调用 WebApplication 类型的 CreateBuilder 方法或者 Create 方法时传入了命令行参数，则会自动添加命令行参数的配置源。（S1516）

```
using System.Diagnostics;

var builder = WebApplication.CreateBuilder(args);
builder.Configuration.AddInMemoryCollection(new Dictionary<string, string>
{
    ["foo"] = "123",
    ["bar"] = "456"
});
var app = builder.Build();
Debug.Assert(app.Configuration["foo"] == "123");
Debug.Assert(app.Configuration["bar"] == "456");
```

15.4 承载环境

基于 IHostBuilder/IHost 的服务承载系统采用 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储内容文件的根目录路径。Web 应用需要更多的承载环境信息，额外的信息被定义在 IWebHostEnvironment 接口中。

15.4.1 IWebHostEnvironment

如下面的代码片段所示，派生于 IHostEnvironment 接口的 IWebHostEnvironment 接口定义了 WebRootPath 属性和 WebRootFileProvider 属性，前者表示存储 Web 资源文件根目录的路径，后者表示返回该路径对应的 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），则只需要将它存储在 WebRootPath 属性表示的目录下。当前承载环境之间反映在 WebApplicationBuilder 类型的 Environment 属性中。表示承载应用的 WebApplication 类型同样具有这样一个属性。

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

public sealed class WebApplicationBuilder
{
    public IWebHostEnvironment Environment { get; }
    ...
}

public sealed class WebApplication
{
    public IWebHostEnvironment Environment { get; }
```

```
    ...
}
```

我们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 属性表示当前应用的名称，它的默认值为入口程序集的名称。EnvironmentName 属性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”。ASP.NET Core 应用会将所有的内容文件都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，存储此类型内容文件的绝对路径通过 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的工作目录。如果该目录下存在一个名为“wwwroot”的子目录，那么它将用来存储 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 属性将返回 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。（S1517）

```
using System.Diagnostics;
using System.Reflection;

var builder = WebApplication.CreateBuilder();
var environment = builder.Environment;

Debug.Assert(Assembly.GetEntryAssembly()?.GetName().Name ==
  environment.ApplicationName);
var currentDirectory = Directory.GetCurrentDirectory();

Debug.Assert(Equals( environment.ContentRootPath,  currentDirectory));
Debug.Assert(Equals(environment.ContentRootPath, currentDirectory));

var wwwRoot = Path.Combine(currentDirectory, "wwwroot");
if (Directory.Exists(wwwRoot))
{
    Debug.Assert(Equals(environment.WebRootPath, wwwRoot));
}
else
{
    Debug.Assert(environment.WebRootPath == null);
}

static bool Equals(string path1, string path2)
    =>string.Equals(path1.Trim(Path.DirectorySeparatorChar),
```

```
    path2.Trim(Path.DirectorySeparatorChar),StringComparison.OrdinalIgnoreCase);
```

15.4.2 通过配置定制承载环境

IWebHostEnvironment 对象承载的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”“webroot”。静态类 WebHostDefaults 为它们定义了对应的属性。通过“第 14 章 服务承载”可知，前 3 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";
}
```

由于应用初始化过程中的很多操作都与当前的承载环境有关，所以承载环境必须在运行应用最初的环境就被确定下来，并在整个应用生命周期内都不能改变。如果我们希望采用配置的方式来控制当前应用的承载环境，则相应的设置必须在 WebApplicationBuilder 对象创建之前执行，在之后试图修改相关的配置都会抛出异常。按照这个原则，我们可以采用命令行参数的方式对承载环境进行设置。

```
var app = WebApplication.Create(args);
var environment = app.Environment;

Console.WriteLine($"ApplicationName:{environment.ApplicationName}");
Console.WriteLine($"ContentRootPath:{environment.ContentRootPath}");
Console.WriteLine($"WebRootPath:{environment.WebRootPath}");
Console.WriteLine($"EnvironmentName:{environment.EnvironmentName}");
```

上面的演示程序利用命令行参数的方式控制承载环境的 4 个属性。我们首先将命令行参数传入 WebApplication 类型的 Create 方法创建了一个 WebApplication 对象，然后从中提取表示承载环境的 IWebHostEnvironment 对象并将其携带信息输出到控制台上。利用命令行参数的方式运行该程序，并指定了与承载环境相关的 4 个参数，如图 15-8 所示。（S1518）

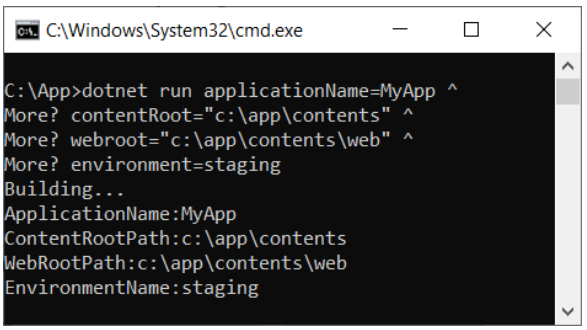

图 15-8　利用命令行参数定义承载环境

除了命令行参数，使用环境变量同样能达到相同的目的，当时应用的名称无法通过对应的配置进行设置。对于上面创建的这个演示程序，现在换一种方式运行它。如图 15-9 所示，在执行“dotnet run”命令运行程序之前，我们为承载环境的 4 个属性设置了对应的环境变量。从输出的结果可以看出，除应用名称依然是入口程序集名称外，承载环境的其他 3 个属性与我们设置的环境变量是一致的。

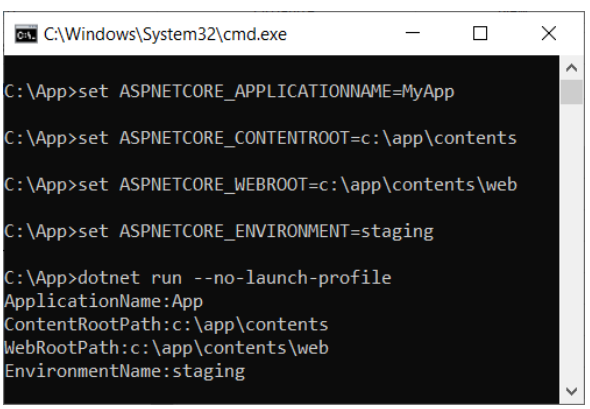

图 15-9　利用环境变量定义承载环境

承载环境除了可以采用利用上面演示的两种方式进行设置，我们也可以使用 WebApplicationOptions 配置选项。如下面的代码片段所示，WebApplicationOptions 定义了 4 个属性，分别表示命令行参数数组、环境名称、应用名称和内容根目录路径。WebApplicationBuilder 具有参数类型为 WebApplicationOptions 的 CreateBuilder 方法。

```
public class WebApplicationOptions
{
    public string[]    Args { get; set; }
    public string      EnvironmentName { get; set; }
    public string      ApplicationName { get; set; }
    public string      ContentRootPath { get; set; }
}

public sealed class WebApplication
```

```
{
    public static WebApplicationBuilder CreateBuilder(WebApplicationOptions options);
    ...
}
```

如果利用 WebApplicationOptions 来对应用所在的承载环境进行设置，则上面演示的程序可以修改成如下形式。由于 WebApplicationOptions 并不包含 WebRootPath 对应的配置选项，如果程序运行后则会发现承载环境的这个属性为空。由于 IWebHostEnvironment 服务提供的应用名称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。（S1519）

```
var options = new WebApplicationOptions
{
    Args              = args,
    ApplicationName   = "MyApp",
    ContentRootPath   = Path.Combine(Directory.GetCurrentDirectory(), "contents"),
    EnvironmentName   = "staging"
};
var app = WebApplication.CreateBuilder(options).Build();
var environment = app.Environment;
Console.WriteLine($"ApplicationName:{environment.ApplicationName}");
Console.WriteLine($"ContentRootPath:{environment.ContentRootPath}");
Console.WriteLine($"WebRootPath:{environment.WebRootPath}");
Console.WriteLine($"EnvironmentName:{environment.EnvironmentName}");
```

IWebHostBuilder 接口中如下 3 个对应的扩展方法用来对承载环境的环境名称、内容文件根目录和 Web 资源根目录进行设置。通过“第 14 章 服务承载”的介绍可知，IHostBuilder 接口也有类似的扩展方法，但是这些扩展方法将无法在 Minima API 中使用。

```
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);
}
```

需要注意的是，针对承载环境的设置必须在创建 WebApplicationBuilder 对象之前进行操作。由于 IWebHostBuilder 对象是通过 WebApplicationBuilder 对象的 WebHost 属性提供的，所以我们自然无法利用它改变已经固定下来的环境设置。如果我们试图这样做，则程序将会抛出一

个类型为 NotSupportedException 的异常。图 15-10 所示为当试图再次修改环境名称时，Visual Studio 出现的异常。

图 15-10　在创建 WebApplicationBuilder 对象后修改环境设置出现的异常

15.4.3　设置监听地址

上面介绍了 IWebHostBuilder 用于设置承载环境的 3 个扩展方法，下面介绍为服务器设置监听地址的 UseUrls 方法。和上面这些方法一样，使用 UseUrls 方法设置的 URL 列表会以“;”作为分隔符连接成一个字符串并写入配置，对应配置项的名称为“urls”，WebHostDefaults 类型同样定义了对应的静态只读 ServerUrlsKey 属性。应用最终采用的监听地址会保存在创建的 WebApplication 对象的 Urls 属性中。由于设置的监听地址被保存在配置中，所以可以利用命令行参数、环境变量或者直接修改对应配置项的方式来指定它们。

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

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

public sealed class WebApplication
{
    public ICollection<string> Urls { get;}
}
```

Minimal API 提供了两种设置监听地址的方式，一种是将监听地址添加到 WebApplication 对象的 Urls 属性中，另一种是直接将监听地址作为参数传入 WebApplication 对象的 Run 方法或者 RunAsync 方法。下面的代码片段演示了这两种编程方式。如果通过这两种方式注册了监听地址，则上面通过配置形式执行的监听地址将会被忽略。由于监听终节点可以直接注册到服务器上，这里还涉及它们之间取舍问题，具体的策略将在“第 18 章 服务器”中进行详细介绍。

```
var app = WebApplication.Create();
app.Urls.Add("http://0.0.0.0:80/");
...
app.Run();
```

```
var app = WebApplication.Create();
...
app.Run("http://0.0.0.0:80/");
```

15.4.4 针对环境的编程

对于同一个 ASP.NET Core 应用来说，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 IsEnvironment 扩展方法，用于确定当前是否为指定的部署环境。IHostEnvironment 接口还提供了额外的 3 个扩展方法并对 3 种典型部署环境（开发、预发和产品）进行判断，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";
}
```

1. 注册服务

前文已经提到，ASP.NET Core 目前支持 3 种服务注册形式，它们都提供了承载环境的动态选择机制。由于可以直接通过 WebApplicationBuilder 的 Environment 属性得到当前的承载环境，所以采用如下方式在不同的环境下为同一个接口注册不同的实现类型，这也是最为简单直接的编程方式。

```
var builder = WebApplication.CreateBuilder(args);
if (builder.Environment.IsDevelopment())
{
    builder.Services.AddSingleton<IFoobar, Foo>();
}
else
{
    builder.Services.AddSingleton<IFoobar, Bar>();
}
```

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

如果调用 IHostBuilder 接口的 ConfigureServices 方法进行服务注册，则可以按照如下服务注册方式达到相同的目的。“14 章　服务承载”已经对这种承载环境的服务注册方式进行了详细的介绍。IWebHostBuilder 接口也有一个类似的 ConfigureServices 方法，所以将 WebApplicationBuilder 的 Host 属性替换成 WebHost 属性，最终的效果也是一样的。

```
var builder = WebApplication.CreateBuilder(args);
builder.Host.ConfigureServices((context, services) =>
{
    if (context.HostingEnvironment.IsDevelopment())
    {
        services.AddSingleton<IFoobar, Foo>();
    }
    else
    {
        services.AddSingleton<IFoobar, Bar>();
    }
});

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

```
var builder = WebApplication.CreateBuilder(args);
builder.WebHost.ConfigureServices((context, services) =>
{
    if (context.HostingEnvironment.IsDevelopment())
    {
        services.AddSingleton<IFoobar, Foo>();
    }
    else
    {
        services.AddSingleton<IFoobar, Bar>();
    }
});

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

2．注册中间件

针对不同的环境注册对应的中间件也是一个常见的需求。如果采用之前的应用承载方式，则可以调用 IWebHostBuilder 接口的 Configure 方法或者利用注册的 Startup 类型来完成中间件的注册，它们均提供了基于承载环境进行针对性中间件注册的功能。由于这两种编程模式在 Minimal API 下均不再支持，所以本书不对这部分内容进行介绍。由于 WebApplicationBuilder 的 Environment 属性直

接提供当前的承载环境，所以针对不同环境注册针对性的中间件变得简单而直接。

```
var app = WebApplication.Create(args);
if (app.Environment.IsDevelopment())
{
    app.UseMiddleware<FooMiddleware>();
}
else
{
    app
        .UseMiddleware<BarMiddleware>()
        .UseMiddleware<BazMiddleware>();
}
app.Run();
```

3. 配置

与前面介绍的服务注册一样，针对应用配置的设置同样具有 3 种不同的编程模式，而且它们都支持针对不同承载环境的针对性设置。JSON 文件是承载配置最常见的形式。我们可以将配置内容分配到多个文件中。如图 15-11 所示，我们可以将与承载环境无关的配置定义在 Appsettings.json 文件中，针对环境的差异化配置定义在以“Appsettings.{EnvironmentName}.json”形式命名（Appsettings.Development.json、Appsettings.Staging.json 和 Appsettings.Production.json）的文件中。

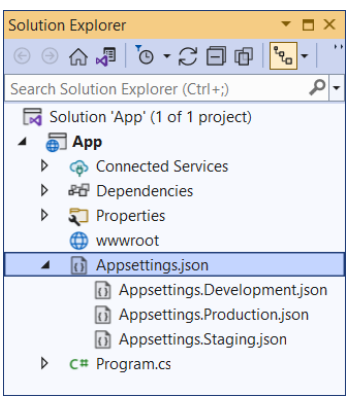

图 15-11 针对承载环境的配置文件

由于 WebApplicationBuilder 利用其 Configuration 属性和 Environment 属性直接提供了构建配置的 ConfigurationManager 对象和当前的承载环境，所以可以直接采用如下方式对这些配置文件进行注册。这也是我们推荐的编程方式。

```
var builder = WebApplication.CreateBuilder(args);
builder.Configuration
    .AddJsonFile(path: "AppSettings.json", optional: false)
    .AddJsonFile(path: $"AppSettings.{builder.Environment.EnvironmentName}.json",
        optional: true);
var app = builder.Build();
```

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

通过“14 章 服务承载”可知，IHostBuilder 接口的 ConfigureAppConfiguration 方法也实现了类似的功能，所以也可以按照如下方式利用 WebApplicationBuilder 的 Host 属性返回的 IHostBuilder 对象将这个方法“借用”过来。IWebHostBuilder 接口同样定义了类似的方式，所以将 Host 属性替换成 WebHost 属性，最终也会达到一样的效果。

```
var builder = WebApplication.CreateBuilder(args);
builder.Host.ConfigureAppConfiguration((context, configBuilder)=> configBuilder
    .AddJsonFile(path: "AppSettings.json", optional: false)
    .AddJsonFile(path:
$"AppSettings.{context.HostingEnvironment.EnvironmentName}.json",
        optional: true));
var app = builder.Build();
...
app.Run();
```

```
var builder = WebApplication.CreateBuilder(args);
builder.WebHost.ConfigureAppConfiguration((context, configBuilder)=> configBuilder
    .AddJsonFile(path: "AppSettings.json", optional: false)
    .AddJsonFile(path:
$"AppSettings.{context.HostingEnvironment.EnvironmentName}.json",
        optional: true));
var app = builder.Build();
...
app.Run();
```

第16章

应用承载（中）

“第 15 章　应用承载（上）”利用一系列实例演示了 ASP.NET Core 应用的编程模式，并借此来体验基于管道的请求处理流程。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，实际隐藏了很多细节。将管道对于 ASP.NET Core 框架的地位拔得多高都不过分，为了使读者对此有深刻的认识，在介绍真实管道的构建之前，我们先介绍一个 Mini 版的 ASP.NET Core 框架。

16.1　中间件委托链

“第 17 章 应用承载（下）”将会详细介绍 ASP.NET 请求处理管道的构建及它对请求的处理流程，作为对这一部分内容的铺垫，作者提取管道最核心的部分并构建一个 Mini 版的 ASP.NET Core 框架。与真正的框架相比，虽然模拟框架要简单很多，但是它们采用完全一致的设计，在定义接口或者类型时采用真实的名称，但是在 API 的定义和实现上进行了最大限度的简化。（S1601）

16.1.1　HttpContext

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对象的方式实现的。HttpContext 就是这个共享的上下文对象。

如图 16-1 所示，当服务器接收到请求之后，它会创建一个 HttpContext 上下文对象，所有中间件都在这个上下文对象中完成请求的处理工作。那么这个 HttpContext 上下文对象究竟会携带什么样的上下文信息呢？我们知道一个 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 上下文对象承载的最核心的上下文信息。

图 16-1　中间件共享上下文对象

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 上下文对象是针对请求和响应的封装，那么应用程序就可以利用它得到当前请求所有的输入信息，也可以借助它完成所需的所有输出工作。我们为 ASP.NET Core 模拟框架定义了如下极简版本的 HttpContext 类型。

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

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

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

如上面的代码片段所示，我们可以利用 Request 属性返回的 HttpRequest 对象得到当前请求的地址、报头集合和主体内容。我们利用 Response 属性返回的 HttpResponse 对象，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。

16.1.2　中间件

所有针对请求的处理是在当前 HttpContext 上下文对象中完成的，所以一个 Action<HttpContext>委托对象可以用来表示请求处理器（Handler）。但 Action<HttpContext>委托对象仅仅是请求处理器针对“同步”编程模式的表现形式，面向 Task 的异步编程模式的处理器应该表示为 Func<HttpContext,Task>委托对象。由于这个委托对象具有非常广泛的应用，所以专门定义了如下 RequestDelegate 类型，可以看出它就是对 Func<HttpContext,Task>委托对象的表达。由于管道（剔除服务器）本质上就是一个请求处理器，自然可以通过一个 RequestDelegate 委托对象来表示，那么组成管道的中间件又如何表示呢？

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

组成管道的中间件体现为一个 Func<RequestDelegate, RequestDelegate>委托对象，它的输入与输出都是一个 RequestDelegate 委托对象。我们可以这样来理解：对于管道中的某个中间件（图 16-2 中的第一个中间件），后续中间件组成的管道体现为一个 RequestDelegate 委托对象，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管道进行进一步处理，所以它需要将后续管道的 RequestDelegate 作为输入对象。

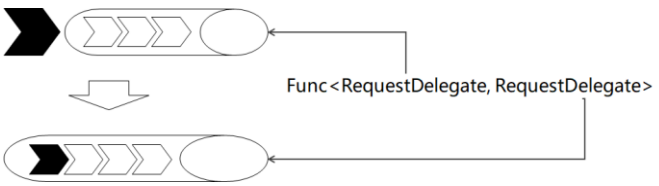

图 16-2　中间件

当表示当前中间件的委托对象执行之后，会将它自己“纳入”这个作为输入的管道，那么表示新管道的 RequestDelegate 就成为中间件委托的输出对象，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>委托对象。如果我们依次注册了多个中间件，则只需要按照它们在管道中的相反的顺序执行对应的委托对象，最终创建作为管道的 RequestDelegate 委托对象。

16.1.3　中间件管道的构建

从事软件行业近 20 年，作者对框架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作者总是会不断地问自己一个问题“还能再简单点吗”。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“能否再简单点吗”，其实是可以的，因为中间件管道本质上就是一个通过 RequestDelegate 委托对象表示的请求处理器，所以图 16-3 所示的请求处理管道将具有更加简单的表达式“Pipeline = Server + Handler（RequestDelegate）”。

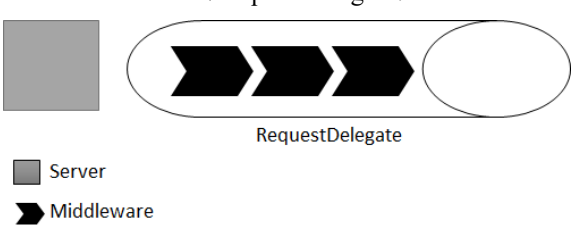

图 16-3　Pipeline = Server + Handler(RequestDelegate)

表示中间件的多个 Func<RequestDelegate, RequestDelegate>委托对象向 RequestDelegate 委托对象的转换是通过 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。由于 Web 应用的本质就是一个请求处理器，所以将中间件管道视为“应用”，使用 IApplicationBuilder 对象构建的“应用”就是由注册中间件构成

的管道，最终体现为一个 RequestDelegate 委托对象。

```
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 方法则将所有的中间件按照注册的顺序组装成一个 RequestDelegate 委托对象。如下所示的 ApplicationBuilder 类型是对 IApplicationBuilder 接口的默认实现，它采用“逆序”执行中间件委托的方式将 RequestDelegate 委托对象构建出来。给出的代码片段还体现了这样一个细节：管道的尾端额外添加了一个返回 404 响应的处理器。这就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;
    }
}
```

16.2　服务器

服务器在管道中的功能非常明确，那就是负责 HTTP 请求的监听、接收和最终的响应。启动后的服务器会绑定到指定的一个或者多个终节点监听请求。请求被服务器接收后用来创建 HttpContext 上下文对象，此上下文对象将作为参数调用表示中间件管道的 RequestDelegate 委托

对象来完成对请求的处理，最后服务器将生成的响应回复给客户端。

16.2.1 IServer

在模拟的 ASP.NET Core 框架中，我们将服务器定义成极度简化的 IServer 接口。如下面的代码片段所示，该接口定义了唯一的 StartAsync 方法用来启动自身的服务器。服务器最终需要将接收的请求分发给表示中间件管道的 RequestDelegate 委托对象，该委托对象体现为 handler 参数。

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

16.2.2 针对服务器的适配

面向应用层的 HttpContext 上下文对象是对请求和响应的封装与抽象，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抽象的 HttpContext 上下文对象，但是却可以注册不同类型的服务器，如何解决两者之间的适配问题呢？在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，就再加一层。抽象的 HttpContext 上下文对象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。

HttpContext 上下文对象与服务器之间的这层抽象体现为定义的一系列“特性”（Feature）。如图 16-4 所示，HttpContext 上下文对象提供的状态和表现出来的能够以特性的方式抽象出来，具体的服务器为这些抽象的特性提供具体的实现。抽象的特性一般都对应一个接口，在系统提供的众多特性接口中，最重要的莫过于提供请求和完成响应的 IRequestFeature 接口和 IResponseFeature 接口。

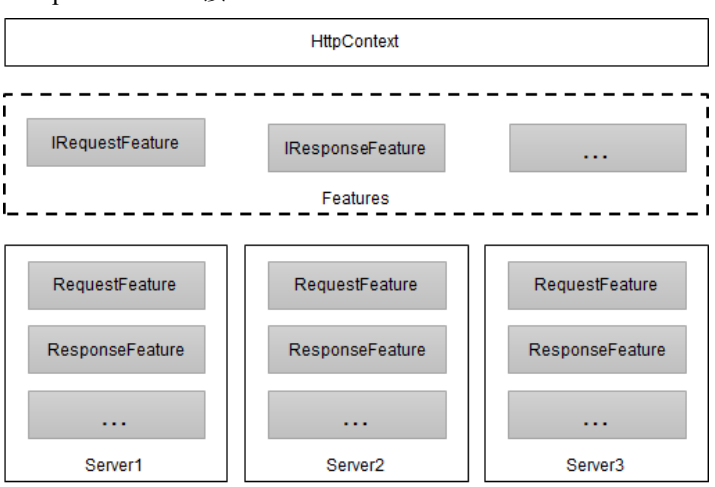

图 16-4　利用特性实现对不同服务器类型的适配

我们定义了如下 IFeatureCollection 接口用来表示存储特性的集合。这是一个将 Type 和

Object 作为 Key 与 Value 的字典，Key 表示注册 Feature 所采用的类型，而 Value 表示 Feature 对象本身，也就是说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>扩展方法和 Get<T>扩展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) where T : class
        => features.TryGetValue(typeof(T), out var value) ? (T?)value : default;

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

我们为提供请求和完成响应的特性定义了 IHttpRequestFeature 接口和 IHttpResponseFeature 接口。如下面的代码片段所示，这两个接口与前面定义的 HttpRequest 类型和 HttpResponse 类型具有一致的成员。

```
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; }
}
```

前面给出了 HttpContext 类型的成员定义，现在为其提供具体的实现。如下面的代码片段所示，表示请求和响应的 HttpRequest 对象与 HttpResponse 对象分别是由对应的特性（IHttpRequestFeature 对象和 IHttpResponseFeature 对象）创建的。HttpContext 上下文对象自身是通过一个表示特性集合的 IFeatureCollection 对象创建的，它会在初始化过程中从这个集合提取对应的特性来创建 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>()
        ?? throw new InvalidOperationException("IHttpRequestFeature does not exist.");
}

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>()
        ?? throw new InvalidOperationException("IHttpResponseFeature does not exist.");

}
```

我们利用 HttpContext 上下文对象的 Request 属性提取的请求信息最初来源于 IHttpRequestFeature 特性，利用它的 Response 属性针对响应所做的任意操作最终都会落到 IHttpResponseFeature 特性上。这两个特性由注册的服务器提供，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。

16.2.3 HttpListenerServer

在对服务器的功能和它与 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。将它命名为 HttpListenerServer，因为它对请求的监听、接收和响应是由一个

HttpListener 对象来完成的。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 HttpContext 上下文对象，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。

当 HttpListener 对象在接收到请求之后同样会创建一个 HttpListenerContext 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，就意味着所有的请求信息都来源于这个原始的 HttpListenerContext 上下文对象。该上下文对象用来完成请求的响应。HttpListenerServer 对应特性所起的作用实际上就是在 HttpListenerContext 和 HttpContext 这两个上下文对象之间搭建起一座桥梁，如图 16-5 所示。

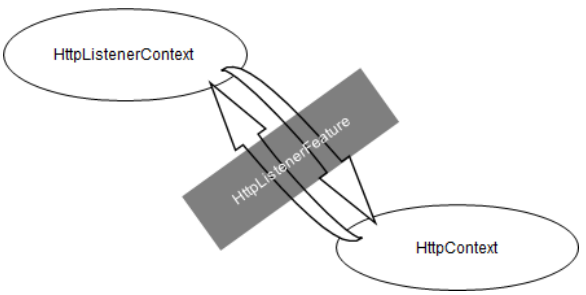

图 16-5　利用 HttpListenerFeature 适配 HttpListenerContext 和 HttpContext

图 16-5 中用来在 HttpListenerContext 和 HttpContext 这两个上下文对象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，我们使 HttpListenerFeature 类型同时实现了 IHttpRequestFeature 接口和 IHttpResponseFeature 接口。在创建 HttpListenerFeature 特性时需要提供一个 HttpListenerContext 上下文对象，IHttpRequestFeature 接口的实现成员所提供的请求信息来源于这个上下文对象，IHttpResponseFeature 接口的实现成员针对响应的操作最终也转移到这个上下文对象。

```
public class HttpListenerFeature : IHttpRequestFeature, IHttpResponseFeature
{
    private readonly HttpListenerContext  context;
    public HttpListenerFeature(HttpListenerContext 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;
    }
}
```

如下所示的代码片段提供了 HttpListenerServer 类型的完整定义。我们在创建 HttpListenerServer 对象时可以显式提供一组监听地址，如果没有提供，则监听地址被默认设置为“localhost:5000”。在实现的 StartAsync 方法中，启动了在构造函数中创建的 HttpListener 对象，并在一个无限循环中调用其 GetContextAsync 方法实现了请求的监听和接收。

```
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 Task StartAsync(RequestDelegate handler)
    {
        Array.ForEach( urls, url =>  httpListener.Prefixes.Add(url));
         httpListener.Start();
        while (true)
        {
              = ProcessAsync(handler);
        }

        async Task ProcessAsync(RequestDelegate handler)
        {
            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 上下文对象，此时只需要利用它创建一个 HttpListenerFeature 特性，并且分别以 IHttpRequestFeature 接口和 IHttpResponseFeature 接口的形式将其注册到创建的 FeatureCollection 集合上。最终利用 FeatureCollection 集合将 HttpContext 上下文对象创建出来，将它作为参数调用表示中间件管道的 RequestDelegate 委托对象，中间件管道将接管并处理该请求。

16.3　承载服务

由于 ASP.NET Core 应用最终是作为一个需要长时间运行的后台服务承载于“服务承载”系统中的，所以还需要为它定义一个 IHostedService 接口的实现类型，这就是接下来着重介绍的 WebHostedService 类型。

16.3.1　WebHostedService

服务器是整个请求处理管道的“龙头”，启动一个 ASP.NET Core 应用就是为了启动服务器，这项工作实现在作为承载服务的 WebHostedService 类型中。如下面的代码片段所示，我们在创建一个 WebHostedService 对象时需要提供表示服务器的 IServer 对象和表示中间件管道的 RequestDelegate 对象，服务器的启动由实现的 StartAsync 方法完成。简单来说，实现的 StopAsync 方法中什么都没做。

```
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;
}
```

我们基本上完成了所有的核心工作，如果能够将一个 WebHostedService 实例注册到服务承载系统中，它就能够启动一个 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。

16.3.2　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 对象和表示中间件管道的 RequestDelegate 对象。WebHostBuilder 提供了更加“便利”和“优雅”的方法来完成 IServer 对象和 RequestDelegate 对象的注册。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构建管道的 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 类型定义了 UseHttpListenerServer 和 Configure 两个扩展方法，前者用来注册 HttpListenerServer，后者利用提供的 Action<IApplicationBuilder>委托对象来注册任意中间件。

```
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;
    }
}
```

ASP.NET Core 应用是以 WebHostedService 这个承载服务的形式注册到服务承载系统中的，针对 WebHostedService 对象创建和注册实现在 ConfigureWebHost 扩展方法上。如下面的代码片段所示，该扩展方法定义了一个 Action<WebHostBuilder>类型的参数，利用它可以注册服务器、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;
    }
}
```

首先利用 ConfigureWebHost 扩展方法创建一个 ApplicationBuilder 对象，然后利用它和当前的 IHostBuilder 对象创建一个 WebHostBuilder 对象，将这个 WebHostBuilder 对象作为参数调用了指定的 Action<WebHostBuilder>委托对象。该扩展方法随后调用 IHostBuilder 接口的 ConfigureServices 方法注册了构建 WebHostedService 对象的工厂，对于由该工厂创建的 WebHostedService 对象来说，它的服务器来源于依赖注入容器，表示中间件管道的 RequestDelegate 对象则由 ApplicationBuilder 对象根据注册的中间件创建。

16.3.3　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，接下来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，我们首先调用静态类型 Host 的 CreateDefaultBuilder 方法创建一个 IHostBuilder 对象，然后调用 ConfigureWebHost 扩展方法并利用提供的 Action<WebHostBuilder>委托对象注册 HttpListenerServer 服务器和 3 个中间件。在调用 Build 方法创建了作为服务宿主的 IHost 对象之后，调用其 Run 方法启动所有承载的 IHostedSerivce 服务。

```
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 Func<RequestDelegate, RequestDelegate>委托对象，所以我们可以利用与之匹配的 FooMiddleware 方法、BarMiddleware 方法和 BazMiddleware 方法来定义对应的中间件，它们调用如下 WriteAsync 扩展方法在响应中输出了一段文字。

```
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);
    }
}
```

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，则输出结果如图 16-6 所示。浏览器上呈现的文字正是由注册的 3 个中间件写入的。

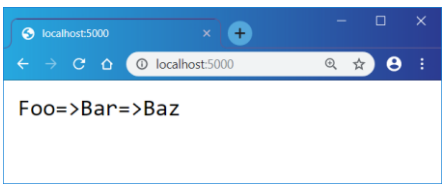

图 16-6　在模拟框架上构建的 ASP.NET Core 应用

第17章

应用承载（下）

在“第16章 应用承载（中）”中，我们利用极少的代码模拟了ASP.NET Core框架的实现，这相当于搭建了一副“骨架”，现在我们将余下的“筋肉”补上，还原一个完整的框架体系。本章主要介绍真实管道的构建流程和应用承载的原理，以及Minimal API背后的“故事“。

17.1 共享上下文对象

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，所有中间件都在由服务器构建的 HttpContext 上下文对象中完成对请求的处理。如果说 HttpContext 是整个ASP.NET Core体系最重要的一个类型，那么相信没有人会有异议。

17.1.1 HttpContext

第16章创建的模拟框架定义了一个简易版的HttpContext类型，它只包含表示请求和响应的两个属性，实际上真正的HttpContext类型拥有更加丰富的成员。除了描述请求和响应的Request属性与 Response 属性，我们还可以从 HttpContext 上下文对象中获取与当前请求相关的很多信息，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。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 HttpContext 上下文对象的 RequestAborted 属性返回的 CancellationToken 对象接收通知。如果需要在请求范围内共享某些数据，则可以将它保存在 Items 属性中。HttpContext 上下文对象的 RequestServices 属性返回的是当前请求的 IServiceProvider 对象。表示请求和响应的 Request 属性与 Response 属性依然是 HttpContext 上下文对象两个重要的成员，前者由如下 HttpRequest 抽象类表示。

```
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 long?                           ContentLength { get; set; }
    public abstract Stream                          Body { get; set; }
    public virtual PipeReader                       BodyReader { get; set; }
    public abstract bool                            HasFormContentType { get; }
    public abstract IFormCollection                 Form { get; set; }
    public virtual RouteValueDictionary             RouteValues { get; }

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

如上面的代码片段所示，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、地址等，也可以获取表示请求的 HTTP 消息的首部和主体。表 17-1 详细描述了定义在 HttpRequest 类型中的主要属性/方法的含义。

表 17-1　定义在 HttpRequest 类型中的主要属性/方法的含义

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）

续表

属性/方法	含　义
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 分别表示 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 分别表示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 分别表示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）。该属性返回的是一个 HostString 对象，它是对主机名称和端口的封装
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 表示针对 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 分别表示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 属性值表示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）
Body/BodyReader	用来读取请求主体内容的 Stream 和 PipeReader
RouteValues	用来存储路由参数的字典
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 分别表示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，接下来介绍另一个与之相对的用于描述响应的 HttpResponse 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，任何形式的响应都可以利用它来实现。

```
public abstract class HttpResponse
{
    public abstract HttpContext             HttpContext { get; }
    public abstract int                     StatusCode { get; set; }
    public abstract IHeaderDictionary       Headers { get; }
    public abstract Stream                  Body { get; set; }
    public virtual PipeWriter               BodyWriter { get; }
```

```
    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 RegisterForDisposeAsync(IAsyncDisposable disposable);
    public virtual void OnCompleted(Func<Task> callback);
    public virtual void Redirect(string location);
    public abstract void Redirect(string location, bool permanent);
}
```

在利用 HttpContext 上下文对象得到表示响应的 HttpResponse 对象之后，可以完成各种类型的响应工作，如设置响应状态码、添加响应报头和写入主体内容等。表 17-2 详细描述了定义在 HttpResponse 类型中的主要属性/方法的含义。

表 17-2　定义在 HttpResponse 类型中的主要属性/方法的含义

属性/方法	含　义
Body	响应主体输出流
BodyWriter	将主体内容写入输出流的 PipeWriter 对象
ContentLength	响应消息主体内容的长度（字节数）
ContentType	响应内容采用的媒体类型/MIME 类型
Cookies	返回一个用于设置（添加或者删除）响应 Cookie（对应响应消息的 Set-Cookie 首部）的 ResponseCookies 对象
HasStarted	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的，所以这个属性表示响应首部是否开始发送
Headers	响应消息的首部集合。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 分别表示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 Value 是一个字符串列表
StatusCode	响应状态码
OnCompleted	注册一个回调操作，以便在响应消息发送结束时自动执行
OnStarting	注册一个回调操作，以便在响应消息开始发送时自动执行
Redirect	发送一个针对指定目标地址的重定向响应消息。permanent 参数表示重定向类型，即状态码为“302”表示暂时重定向或者状态码为“301”表示永久重定向
RegisterForDispose/ RegisterForDisposeAsync	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 IDisposable 接口或者 IAsyncDisposable 接口，所谓的释放体现在对其 Dispose/DisposeAsync 方法的调用

17.1.2　服务器适配

中间件管道总是借助抽象的 HttpContext 上下文对象提取请求和完成响应，但是请求的接收和响应的回复是由服务器完成的，所以必须解决统一抽象的 HttpContext 上下文对象和不同服务器类型之间的适配问题。通过“16 章　应用承载（中）”提供的模拟框架，我们知道这里的适配是借助 “特性”（Feature）完成的。如图 17-1 所示，我们不仅利用特性来提供相应的请求状

态，也赋予特性对请求予以响应的功能。HttpContext 上下文对象被创建在一系列抽象的特性之上，服务器为特性提供了具体的实现。

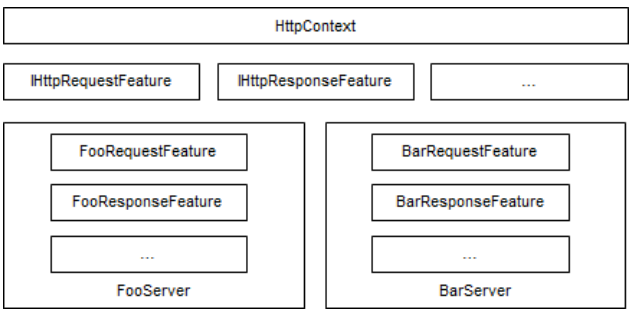

图 17-1　服务器与 HttpContext 上下文对象之间针对特性的适配

由服务器提供的特性集合通过 IFeatureCollection 接口表示。如下面的代码片段所示，一个 IFeatureCollection 对象本质是一个 KeyValuePair<Type, object>对象的集合，由于作为 Key 的类型基本上不会重复，所以它本质上就是一个字典。特性的读/写分别通过 Get<TFeature>方法和 Set<TFeature>方法或者索引来完成。如果 IsReadOnly 属性返回 True，就意味着集合被“冻结”，特性不能被覆盖或者修改。整数类型的只读属性 Revision 可以被视为 IFeatureCollection 对象的版本，针对特性的更改都将改变该属性的值。

```
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; }
}
```

具有如下定义的 FeatureCollection 类型是对 IFeatureCollection 接口的默认实现。它具有两个重载构造函数，默认无参构造函数用于创建一个空的特性集合，另一个构造函数需要指定一个 IFeatureCollection 对象作为后备存储。FeatureCollection 类型的 IsReadOnly 属性总是返回 False，所以它永远是可读可写的。使用无参构造函数创建的 FeatureCollection 对象的 Revision 属性默认返回零，根据指定后备存储创建的 FeatureCollection 对象将提供的 IFeatureCollection 对象的版本作为初始版本。无论采用何种形式改变了注册的特性，此 Revision 属性都将自动递增。

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

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抽象 HttpContext 上下文对象之间的适配而设计的，但是它

的作用不限于此。由于注册的特性采用基于请求的生命周期，所以可以将任何基于请求的状态和功能以特性的方式“附着”在 HttpContext 上下文对象上，其实起到了与 Items 属性类似的作用。由于特性一般都被定义成接口，与相对“随意”的 Items 字段相比，特性更加“正式”一点。

```
public abstract class HttpContext
{
    public abstract IFeatureCollection       Features { get; }
    …
}

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

DefaultHttpContext 对象是 HttpContext 这个抽象类的默认实现。如上面的代码片段所示，该对象就是由指定的 IFeatureCollection 对象构建的。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。表 17-3 列出了这些属性和对应特性接口之间的映射关系。

表 17-3　属性和对应特性接口之间的映射关系

Feature	属　性	含　义
IHttpRequestFeature	Request	获取描述请求的基本信息
IHttpResponseFeature	Response	控制对请求的响应
IHttpResponseBody	Response.Body/BodyWriter	响应主体内容输出流和对应的 PipeWriter
IHttpConnectionFeature	Connection	提供描述当前 HTTP 连接的基本信息
IItemsFeature	Items	提供用户存储针对当前请求的对象容器
IHttpRequestLifetimeFeature	RequestAborted	传递请求处理取消通知和中止当前请求处理
IServiceProvidersFeature	RequestServices	提供根据服务注册创建的 ServiceProvider
ISessionFeature	Session	提供描述当前会话的 Session 对象
IHttpRequestIdentifierFeature	TraceIdentifier	为追踪日志（Trace）提供当前请求的唯一标识
IHttpWebSocketFeature	WebSockets	管理 Web Socket

表 17-3 列举的众多特性接口在后续相关章节中都会涉及，目前我们只关心表示请求和响应的 IHttpRequestFeatureIHttpResponseFeature 接口和 IHttpResponseBodyFeature 接口。从下面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。DefaultHttpContext 的 Request 属性和 Response 属性返回的真实类型分别为 DefaultHttpRequest 与 DefaultHttpResponse，它们分别利用上述这两个特性完成对定义在基类（HttpRequest 和 HttpResponse）的所有抽象成员的实现，但是表示响应主体的输出流（Body 属性）和对应的 PipeWriter（BodyWriter）来源于 IHttpResponseBodyFeature 特性。

```
public interface IHttpRequestFeature
{
    IHeaderDictionary         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; }
    IHeaderDictionary         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);
}

public interface IHttpResponseBodyFeature
{
    Stream            Stream { get; }
    PipeWriter        Writer { get; }

    void DisableBuffering();
    Task StartAsync(CancellationToken cancellationToken = default(CancellationToken));
    Task SendFileAsync(string path, long offset, long? count,
        CancellationToken cancellationToken = default(CancellationToken));
    Task CompleteAsync();
}
```

17.1.3　获取上下文对象

当前请求的 HttpContext 上下文对象可以利用注入 IHttpContextAccessor 对象来获取。如下面的代码片段所示，这个上下文对象体现在 IHttpContextAccessor 接口的 HttpContext 属性上，并且这个属性是可读可写的。

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

HttpContextAccessor 类型是对 IHttpContextAccessor 接口的默认实现。从下面的代码片段可以看出，它将提供的 HttpContext 上下文对象存储在一个 AsyncLocal<HttpContext>对象上，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当前请求的 HttpContext 上下文对象。

```
public class HttpContextAccessor : IHttpContextAccessor
{
    private static AsyncLocal<HttpContext> _httpContextCurrent
```

```
        = new AsyncLocal<HttpContext>();
    public HttpContext HttpContext
    {
        get => _httpContextCurrent.Value;
        set =>  httpContextCurrent.Value = value;
    }
}
```

IHttpContextAccessor/HttpContextAccessor 的服务注册由如下 AddHttpContextAccessor 扩展方法来完成。由于它通过调用 TryAddSingleton<TService, TImplementation>扩展方法的方式来注册服务，所以不用担心多次调用该扩展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;
    }
}
```

17.1.4 上下文对象的创建与释放

ASP.NET Core 应用在开始处理请求前对 HttpContext 上下文对象的创建，以及请求处理完成后对它的回收释放都是通过 IHttpContextFactory 工厂完成的。IHttpContextFactory 接口定义了 Create 和 Dispose 两个方法，前者根据提供的特性集合来创建 HttpContext 上下文对象，后者负责释放回收提供的 HttpContext 上下文对象。

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

DefaultHttpContextFactory 类型是对 IHttpContextFactory 接口的默认实现，DefaultHttpContext 对象就是由它创建的。如下面的代码片段所示，DefaultHttpContextFactory 利用注入的 IServiceProvider 对象得到了 IHttpContextAccessor 对象、用来创建服务范围的 IServiceScopeFactory 工厂和与表单相关的 FormOptions 配置选项。实现的 Create 方法根据提供的特性集合将 DefaultHttpContext 对象创建出来，并将其复制给 IHttpContextAccessor 对象的 HttpContext 属性，此后在当前请求的异步调用链中就可以利用 IHttpContextAccessor 对象得到 HttpContext 上下文对象。

```
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，Create 方法在返回创建的 DefaultHttpContext 对象之前，它还会设置 DefaultHttpContext 对象的 FormOptions 属性和 ServiceScopeFactory 属性。当执行 Dispose 方法时，它会将 IHttpContextAccessor 对象的 HttpContext 属性设置为 Null。

17.1.5　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 上下文对象 RequestServices 属性上的容器来源于 IServiceProvidersFeature 特性。如下面的代码片段所示，该特性接口定义了唯一的 RequestServices 属性。

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

RequestServicesFeature 类型是对 IServiceProvidersFeature 接口的默认实现。如下面的代码片段所示，当创建一个 RequestServicesFeature 对象时需要提供当前的 HttpContext 上下文对象和创建服务范围的 IServiceScopeFactory 工厂。当第一次从 RequestServicesFeature 对象的 RequestServices 提取基于当前请求的依赖注入容器时，它会利用 IServiceScopeFactory 工厂创建一个服务范围，并返回该范围内的 IServiceProvider 对象。我们已经多次强调依赖注入的服务范围在 ASP.NET Core 应用下是指基于请求的“范围”，其本质就体现在这里。

```
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，我们必须及时将创建的服务范围释放。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。在读取 RequestServices 属性时，如果涉及针对服务范围的创建，则 RequestServicesFeature 对象会调用表示当前响应的 HttpResponse 对象的 RegisterForDisposeAsync 将 DisposeAsync 方法注册为回调，此回调的注册确保了创建的服务范围在完成响应之后被终结。除了创建返回的 DefaultHttpContext 对象，DefaultHttpContextFactory 对象还会设置创建服务范围的工厂（对应 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。

17.2　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，它面对的是一个 IHttpApplication<TContext>对象。所以管道可以视为由 IServer 和 IHttpApplication<TContext>对象组成，如图 17-2 所示。

图 17-2　由 IServer 和 IHttpApplication<TContext>对象组成的管道

17.2.1　IServer

由 IServer 接口表示的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。该接口具有如下 3 成员，其中由服务器提供的特性就保存在其 Features 属性返回的 IFeatureCollection 集合中，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，这个地址是从外部指定的。具体来说，服务器采用的监听地址会被封装成一个 IServerAddressesFeature 特性，并在启动服务器之前被添加到它的特性集合中。如下面的代码片段所示，该特性接口除了定义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，那么是否优先考虑使用前者。

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

对服务器而言，IHttpApplication<TContext>对象就是将要接管并处理请求的整个应用，从该接口命名也可以看出来。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 IHttpApplication<TContext>对象。IHttpApplication<TContext>接口的泛型参数 TContext 表示整个请求处理构建的上下文类型，这个上下文类型由 IHttpApplication<TContext>对象的 CreateContext 方法构建，在此上下文类型中针对请求的处理体现在 ProcessRequestAsync 方法中，上下文类型在请求处理结束后由 DisposeContext 方法释放。

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

17.2.2　HostingApplication

如下 HostingApplication 类型是 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。Context 是对一个 HttpContext 上下文对象的封装，它同时提供一些额外的“诊断”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以不同的形式（DiagnosticSource 诊断日志、EventSource 事件日志和 .NET 日志系统）输出一些诊断日志。Context 除 HttpContext 外的其他属性都与诊断日志有关。它的 Scope 属性返回为当前请求创建的日志范围，此范围会携带请求的唯一 ID，如果注册的 ILoggerProvider 对象支持日志范围，提供的 ILogger 对象就可以将这个请求 ID 记录下来，这就意味着可以根据此 ID 将同一个请求的多条日志提取出来构成一组完整的跟踪记录。

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，该时间戳体现在 Context 的 StartTimestamp 属性上。它的 EventLogEnabled 属性表示是否开启 EventSource 事件日志，而 Activity 属性返回表示整个请求处理操作的 Activity 对象。

如下所示为 HostingApplication 类型的完整定义，我们在创建此对象时需要提供表示中间件管道的 RequestDelegate 对象和 IHttpContextFactory 工厂，提供的 ILogger 对象、DiagnosticListener 对象和 ActivitySource 对象被用来创建输出诊断信息的 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,
        ActivitySource activitySource,IHttpContextFactory httpContextFactory)
    {
       _application = application;
        diagnostics = new HostingApplicationDiagnostics(logger, diagnosticSource,
           activitySource);
       _httpContextFactory = httpContextFactory;
    }
```

```
    public Context CreateContext(IFeatureCollection contextFeatures)
    {
        var context = new Context();
        var httpContext =  httpContextFactory.Create(contextFeatures);
        _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);
    }
}
```

HostingApplication 的 CreateContext 方法利用 IHttpContextFactory 工厂创建当前 HttpContext 上下文对象并将它封装成 Context 对象。在返回这个对象之前，它会调用 HostingApplicationDiagnostics 对象的 BeginRequest 方法输出“开始处理请求”事件的日志。ProcessRequestAsync 方法仅仅调用了表示中间件管道的 RequestDelegate 对象便可以完成请求的处理。用于释放上下文的 DisposeContext 方法直接调用 IHttpContextFactory 工厂的 Dispose 方法来释放 HttpContext 上下文对象。可以看出 HttpContext 上下文对象的生命周期是由 HostingApplication 控制的。完成 HttpContext 上下文对象的释放之后，HostingApplication 利用 HostingApplicationDiagnostics 对象输出“完成请求处理”时间的日志。Context 对象的 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，则完整的管道体现为图 17-3 所示的结构。

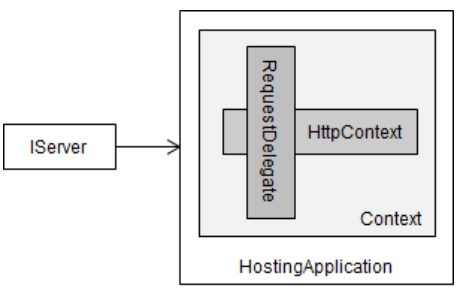

图 17-3　由 IServer 和 HostingApplication 组成的管道

17.2.3　诊断日志

很多人可能对 ASP.NET Core 框架记录的诊断日志并不关心，其实这些日志对纠错、排错和性能监控提供了很有用的信息。如果需要创建一个 APM（Application Performance Management）系统来监控 ASP.NET Core 应用处理请求的性能及出现的异常，则完全可以将 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 系统（如 OpenTelemetry.NET 、Elastic APM 和 SkyWalking APM 等）都是利用这种方式收集分布式跟踪日志的。

1．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，我们通过一个简单的实例演示将 HostingApplication 对象写入的诊断日志输出到控制台上。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 8 章 诊断日志（中）”介绍的日志系统。如下演示程序利用 WebApplicationBuilder 的 Logging 属性得到返回的 ILoggingBuilder 对象，并调用它的 AddSimpleConsole 扩展方法为默认注册的 ConsoleLoggerProvider 开启了针对日志范围的支持。最后调用 IApplicationBuilder 接口的 Run 方法注册一个中间件，该中间件在处理请求时会利用依赖注入容器提取用于发送日志事件的 ILogger<Program>对象，并利用它写入一条 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。

```
var builder = WebApplication.CreateBuilder(args);
builder.Logging.AddSimpleConsole(options => options.IncludeScopes = true);
var app = builder.Build();
app.Run(HandleAsync);
app.Run();

static Task HandleAsync(HttpContext httpContext)
{
    var logger = httpContext.RequestServices.GetRequiredService<ILogger<Program>>();
    logger.LogInformation($"Log for event Foobar");
    if (httpContext.Request.Path == new PathString("/error"))
    {
        throw new InvalidOperationException("Manually throw exception.");
    }
    return Task.CompletedTask;
}
```

在运行程序之后，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，控制台上会输出 7 条日志，如图 17-4 所示。由于开启了日志范围的支持，所以输出的日志都会携带日志范围的信息，日志范围提供了很多有用的分布式跟踪信息，如 Trace ID、Span ID、Parent Span ID，以及请求的 ID 和路径等。请求 ID（Request ID）由当前的连接 ID 和一个序列号组成。从图 17-4 可以看出，两次请求的 ID 分别是“0HMDS8HHE6GD2:00000002”和“0HMDS8HHE6GD2:00000003”。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（0HMCT012M2D9E）。同一个连接的多次请求将一个自增的序列号（00000002 和 00000003）作为唯一标识。（S1701）

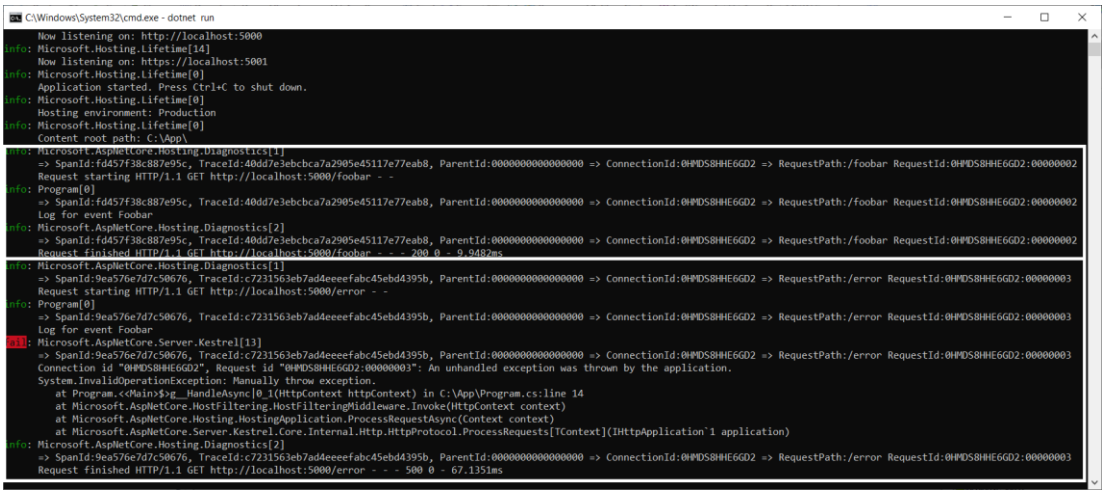

图 17-4　捕捉 HostingApplication 记录的诊断日志

对于两次请求输出的 7 条日志，类别为“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 1 条是在开始处理请求时写入的，利用这条日志获取请求的 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。当请求处理结束后输出第 3 条日志，日志承载的信息包括请求处理耗时（9.9482 毫秒）和响应状态码（200）。如果响应具有主体内容，则对应的媒体类型同样被记录下来。

对于第二次请求，由于人为抛出了异常，所以异常的信息被写入日志。如果我们足够仔细，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”。

2．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载荷的数据结构。我们通过查看 HostingApplication 类型的源代码，就会发现它针对“开始请求”“结束请求”“未处理异常”这 3 类诊断日志事件采用如下命名方式。

- 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。

至于诊断日志消息的内容载荷（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对象的 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，它承载的内容载荷还包括抛出异常，对应的成员名称为“exception”。下面的演示程序定义了 DiagnosticCollector 类型作为诊断监听器，再利用它定义上述 3 类诊断事件的监听方法。

```
public class DiagnosticCollector
{
    [DiagnosticName("Microsoft.AspNetCore.Hosting.BeginRequest")]
    public void OnRequestStart(HttpContext httpContext, long timestamp)
    {
        var request = httpContext.Request;
        Console.WriteLine($"\nRequest 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}");
    }
}
```

“开始请求”事件的 OnRequestStart 方法输出了当前请求的 HTTP 版本、HTTP 方法和 URL。为了能够计算整个请求处理的耗时，它将当前时间戳保存在 HttpContext 上下文对象的 Items 集合中。“结束请求”事件的 OnRequestEnd 方法将这个时间戳从 HttpContext 上下文对象中提取出来，结合当前时间戳计算请求处理耗时，该耗时和响应的状态码最终会被写入控制台。“未处理异常”诊断事件的 OnException 方法在调用 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。如下所示的演示程序中利用 WebApplication 的 Services 提供的依赖注入容器提取注册的 DiagnosticListener 对象，并调用它的 SubscribeWithAdapter 扩展方法将 DiagnosticCollector 对象注册为订阅者。调用 Run 方法注册了一个中间件，该中间件会在请求路径为“/error”的情况下抛出异常。

```
using App;
using System.Diagnostics;
```

```
var builder = WebApplication.CreateBuilder(args);
builder.Logging.ClearProviders();
var app = builder.Build();
var listener = app.Services.GetRequiredService<DiagnosticListener>();
listener.SubscribeWithAdapter(new DiagnosticCollector());
app.Run(HandleAsync);
app.Run();

static Task HandleAsync(HttpContext httpContext)
{
    var listener =
      httpContext.RequestServices.GetRequiredService<DiagnosticListener>();
    if (httpContext.Request.Path == new PathString("/error"))
    {
        throw new InvalidOperationException("Manually throw exception.");
    }
    return Task.CompletedTask;
}
```

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，控制台会输出 DiagnosticCollector 对象收集的诊断信息，如图 17-5 所示。（S1702）

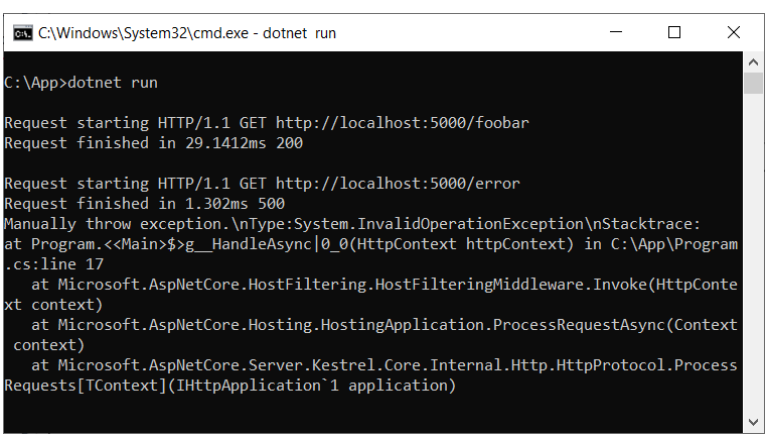

图 17-5　利用注册的诊断监听器获取诊断日志

3．EventSource 事件日志

HostingApplication 在处理每个请求的过程中还会利用名称为“Microsoft.AspNetCore.Hosting”的 EventSource 对象发出相应的日志事件。这个 EventSource 对象来回在启动和关闭应用程序时发出相应的事件，该对象涉及的 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。

演示程序利用创建的 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 类型，并在启动应用程序前创建了 EventListener 对象，通过注册该对象的 EventSourceCreated 事件来开启针对上述 EventSource 的监听。注册的 EventWritten 事件会将监听到的事件名称的负载内容输出到控制台上。

```
using System.Diagnostics.Tracing;

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($"\t{args.PayloadNames[index]} = {args.Payload?[index]}");
    }
};


var builder = WebApplication.CreateBuilder(args);
builder.Logging.ClearProviders();
var app = builder.Build();
app.Run(HandleAsync);
app.Run();

static Task HandleAsync(HttpContext httpContext)
{
    if (httpContext.Request.Path == new PathString("/error"))
    {
        throw new InvalidOperationException("Manually throw exception.");
    }
    return Task.CompletedTask;
}

public class DiagnosticCollector : EventListener { }
```

首先以命令行的形式启动这个演示程序后，从图 17-6 所示的输出结果可以看到，名为 HostStart 的事件被发送。然后采用目标地址“http://localhost:5000/foobar”和“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送 RequestStart 事件和 RequestStop 事件，而针对后者的处理则会因为抛出的异常发送额外的事件 UnhandledException。按 Ctrl+C 组合键关闭应用程序后，名称为 HostStop 的事件

被发送。对于通过 EventSource 发送的 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。（S1703）

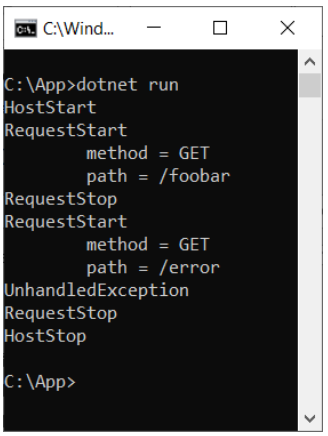

图 17-6　利用注册 EventListener 监听器获取诊断日志

17.3　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 RequestDelegate 委托对象来处理 IServer 对象分发给它的请求。这个 RequestDelegate 委托对象由所有的中间件按照注册顺序构建，它是对中间件委托链的体现。如果将这个 RequestDelegate 委托对象替换成原始的中间件，则 ASP.NET Core 应用的请求处理管道体现为图 17-7 所示的结构。

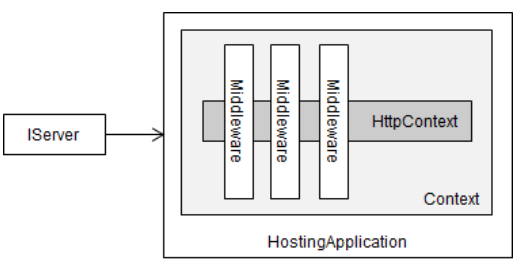

图 17-7　完整的请求处理管道

17.3.1　IApplicationBuilder

ASP.NET Core 应用对请求的处理完全体现在注册的中间件上，所以“应用”从某种意义上是指由注册中间件构建的 RequestDelegate 委托对象。正因为如此，构建 RequestDelegate 委托对象的接口才被命名为“IApplicationBuilder”。IApplicationBuilder 是 ASP.NET Core 框架中的一个核心对象，可以将中间件注册在它上面，并且利用它来创建表示中间件委托链的

RequestDelegate 委托对象。IApplicationBuilder 接口定义了如下 3 个属性，ApplicationServices 属性表示针对当前应用程序的依赖注入容器，ServerFeatures 属性表示返回服务器提供的特性集合，Properties 属性表示提供一个可以用来存储任意属性的字典。

```
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();
}
```

Func<RequestDelegate, RequestDelegate>委托对象的中间件通过调用 IApplicationBuilder 接口的 Use 方法进行注册。RequestDelegate 委托对象的构建体现在 Build 方法上，它的另一个 New 方法用于创建一个新的 IApplicationBuilder 对象。如下这个作为 IApplicationBuilder 接口默认实现的 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，Build 方法采用逆序调用这些 Func<RequestDelegate, RequestDelegate>委托对象便将 RequestDelegate 委托对象构建出来。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。

```
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(IServiceProvider 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)
    {
        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;
    }
}
```

从上面的代码片段可以看出，无论是通过 ApplicationServices 属性返回的 IServiceProvider 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 属性返回的字典中。ApplicationBuilder 具有两个重载公共构造函数，其中一个构造函数定义了一个名为“server”的参数（Object 类型），但这个参数并不是表示服务器，而是表示服务器提供的特性集合。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被复制到该对象中。

ASP.NET Core 框架使用的 IApplicationBuilder 对象是由 IApplicationBuilderFactory 工厂创建的。如下面的代码片段所示，IApplicationBuilderFactory 接口定义了唯一的 CreateBuilder 方法，它会根据提供的特性集合创建相应的 IApplicationBuilder 对象。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);
}
```

17.3.2　弱类型中间件

虽然中间件最终体现为一个 Func<RequestDelegate, RequestDelegate>委托对象，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具体的类型。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，它被称为“弱类型中间件”；另一种是直接实现 IMiddleware 接口，它被称为“强类型中间件”。弱类型中间件会按照如下约定定义。

- 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，ASP.NET Core 框架在创建中间件对象时会将表示后续中间件管道的 RequestDelegate 委托对象绑定为这个参数。构造函数不仅可以包含任意其他参数，而且对参数 RequestDelegate 出现的位置不会进行任何约束。
- 请求的处理实现在返回类型为 Task 的 Invoke 方法或者 InvokeAsync 方法中。这两个方法的第一个参数类型必须是 HttpContext，将自动绑定为当前 HttpContext 上下文对象。对于后续的参数，虽然约定并未对此进行限制，但是由于这些参数最终是由依赖注入容器提供的，所以相应的服务注册必须存在。

如下 FoobarMiddleware 类型就是采用约定定义的弱类型中间件。我们在构造函数中注入了后续中间件管道的 RequestDelegate 对象和 IFoo 对象。用于请求处理的 InvokeAsync 方法除了定义与当前 HttpContext 上下文对象绑定的参数，还注入了一个 IBar 对象，该方法在完成自身请求处理操作之后，通过在构造函数中注入的 RequestDelegate 委托对象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);
    }
}
```

中间件类型通过调用如下 IApplicationBuilder 接口的两个扩展方法进行注册。当调用这两个扩展方法时，除了指定具体的中间件类型，还可以传入一些必要的参数，它们都将作为调用构造函数的输入参数。由于中间件实例是由依赖注入容器构建的，容器会尽可能地利用注册的服务来提供所需的参数，所以指定的参数列表用来提供无法由容器提供或者需要显式指定的参数。

```
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> 委托对象来表示中间件，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这样一个委托对象。如下 UseMiddleware 方法模拟了中间件类型向 Func<RequestDelegate, RequestDelegate>类型转换的逻辑。

```
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();
    }
}
```

由于请求处理实现在中间件类型的 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，将中间件实例的 Invoke 方法或者 InvokeAsync 方法调用转换成 Func<RequestDelegate, RequestDelegate>委托对象。借助于依赖注入框架，第一个问题很好解决，上面的 UseMiddleware 方法是调用 ActivatorUtilities 类型的 CreateInstance 方法将中间件实例创建出来的。

由于中间件类型的 Invoke 方法和 InvokeAsync 方法要求其返回类型和第一个参数类型分别 Task 和 HttpContext，所以针对这两个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，有了依赖注入容器的帮助，初始化输入参数就显得非常容易。我们只需要从表示方法的 MethodInfo 对象中解析出对应的参数类型，就能够根据该类型从容器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 Invoke 方法或者 InvokeAsync 方法的调用。基于表达式树针对中间件实例的 Invoke 方法或者 InvokeAsync 方法的调用，实现在前面提供的 CreateFactory 方法中。

17.3.3 强类型中间件

弱类型中间件对象在应用初始化时就被创建，所以它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，在这种情况下只能定义强类型中间件。强类型中间件需要实现如下 IMiddleware 接口，该接口定义了唯一的 InvokeAsync 方法用来处理请求。中间件可以利用该方法的输入参数得到当前的 HttpContext 上下文对象和表示后续中间件管道的 RequestDelegate 委托对象。

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

由于强类型中间件是在处理请求时由当前请求对应的依赖注入容器（RequestServices）提供的，所以必须将中间件类型注册为服务，当进行服务注册时指定希望采用的生命周期模式。我们一般只会在需要使用 Scoped 生命周期模式时才会采用这种方式来定义中间件，当然设置成 Singleton 生命周期模式也未尝不可。读者可能会问：能否采用 Transient 生命周期模式呢？实际上这与 Scoped 生命周期模式是没有区别的，因为中间件针对同一个请求只会使用一次。强类型中间件对象的创建与释放是通过 IMiddlewareFactory 工厂来完成的。如下面的代码片段所示，IMiddlewareFactory 接口提供了 Create 和 Release 两个方法，前者根据指定的中间件类型创建对应的实例，后者负责释放指定的中间件对象。

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

MiddlewareFactory 是 IMiddlewareFactory 接口的默认实现。如下面的代码片段所示，它直接利用指定的 IServiceProvider 对象根据指定的中间件类型来提供对应的实例。由于依赖注入框架具有针对提供服务实例的生命周期管理策略，所以实现的 Release 方法不需要执行任何操作。

```
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) {}
}
```

UseMiddleware 方法模拟了强/弱类型中间件的注册。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，则 UseMiddleware 方法会直接创建一个 Func<RequestDelegate, RequestDelegate>委托对象作为注册的中间件。当该委托对象被执行时，它会从当前 HttpContext 上下文对象的 RequestServices 属性中获取当前请求的依赖注入容器，并由它来提供 IMiddlewareFactory 工厂。在利用它根据中间件类型将对应实例创建出来后，直接调用其 InvokeAsync 方法来处理请求。在请求处理结束后，IMiddlewareFactory 工厂的 Release 方法被用来释放此中间件。

```
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);
                    }
                };
            });
        }
```

```
    }
    ...
}
```

UseMiddleware 方法之所以从当前 HttpContext 的 RequestServices 属性而不是 IApplicationBuilder 的 ApplicationServices 属性来获取依赖注入容器，是因为生命周期方面的考虑。由于后者是与应用具有相同生命周期的根容器，无论中间件服务注册的生命周期模式是 Singleton 还是 Scoped，提供的中间件实例都是一个 Singleton 对象，所以无法满足针对请求创建和释放中间件对象的初衷。如果注册的是实现了 IMiddleware 接口的中间件类型，则不允许指定任何参数。

17.3.4 注册中间件

中间件总是注册到 IApplicationBuilder 对象上。对于我们推荐的 Minimal API 应用承载方式来说，表示承载应用的 WebApplication 类型实现了 IApplicationBuilder 接口，所以只需要直接将中间件注册到这个对象上。中间件还可以采用 IStartupFilter 对象的方式注册。如下面的代码片段所示，IStartupFilter 接口定义了唯一的 Configure 方法，它返回的 Action<IApplicationBuilder>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，用来完成后续的中间件注册工作。当我们希望将某个中间件置于管道首尾两端时，往往会采用这种方式。

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

17.4 应用的承载

ASP.NET Core 应用最终会作为一个长时间运行的后台服务托管在服务承载系统中，采用的承载服务类型为 GenericWebHostService，将它与上面介绍的这一切整合在一起。在介绍这个承载服务类型之前，我们先来认识一下对应的 GenericWebHostServiceOptions 配置选项。

17.4.1 GenericWebHostServiceOptions

如下 GenericWebHostServiceOptions 配置选项类型定义了 3 个属性，其核心配置选项集中在 WebHostOptions 属性上。它的 ConfigureApplication 属性返回一个 Action<IApplicationBuilder>委托对象，应用初始化过程中针对中间件的注册最终都会转移到这个委托对象上。我们可以采用“Hosting Startup”的形式注册一个外部程序集来完成一些初始化的工作。它的 HostingStartupExceptions 属性返回的 AggregateException 就是对这些初始化任务执行过程中抛出异常的封装。

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

```
    public AggregateException               HostingStartupExceptions { get; set; }
}
```

一个 WebHostOptions 对象承载了与 IWebHost 相关的配置选项，在“第 15 章 应用承载（上）”介绍的“三代”承载方式中，IWebHost 对象在初代承载方式中表示承载 Web 应用的“宿主”（Host）。虽然在基于 IHost/IHostBuilder 的承载系统中，IWebHost 接口已经没有任何意义，但是 WebHostOptions 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，当调用 WebHostOptions 这个构造函数时，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。这些预定义的配置键作为静态只读字段被定义在 WebHostDefaults 静态类中，其中大部分在第 16 章已有相关介绍，本节只对此进行总结。

```
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";
```

```
}
```

表 17-4 列出了定义在 WebHostOptions 配置选项中的属性。值得一提的是，对于布尔类型的属性值（如 PreventHostingStartup 和 CaptureStartupErrors），配置项的值“True”（不区分字母大小写）和“1”都将转换为 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 ParseBool 静态方法中，如果有类似的需求则可以直接调用这个静态方法。

表 17-4　定义在 WebHostOptions 配置选项中的属性

属　性	配 置 键	说　明
ApplicationName	applicationName	应用名称。如果调用 IWebHostBuilder 接口的 Configure 方法注册中间件，则提供的 Action<IApplicationBuilder> 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 IWebHostBuilder 接口的 UseStartup 扩展方法，则指定的 Startup 类型所在的程序集名称将作为应用名称
Environment	environment	应用当前的部署环境。如果没有显示指定，则默认的环境名称为 Production
ContentRootPath	contentRoot	存储静态内容文件的根目录。如果未做显式设置，则默认为当前工作目录
WebRoot	webroot	存储静态 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，则该目录将作为 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，则指定的 Startup 类型所在的程序集名称将作为该属性的值
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，则可以在一个单独的程序集中利用 HostingStartupAttribute 特性注册一个实现了 IHosting Startup 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 属性表示的程序集名称默认被添加到这个列表中
HostingStartupExclude Assemblies	hostingStartupExclude Assemblies	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，则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，则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	应用关闭时的超时时限，默认时限为 5 秒

17.4.2　GenericWebHostService

如下面的代码片段所示，在 GenericWebHostService 类型的构造函数中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 工厂、用来触发诊断事件的 DiagnosticListener 对象、用来创建 Activity 的 ActivitySource 对象，用来创建 HttpContext 上下文对象的 IHttpContextFactory 工厂、用来创建 IApplicationBuilder 对象的 IApplicationBuilderFactory 工厂、注册的所有 IStartupFilter 对象、承载当前应用配置的 IConfiguration 对象和表示当前承载环境的 IWebHostEnvironment 对象。

```
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 IApplicationBuilderFactory        ApplicationBuilderFactory { get; }
    public IEnumerable<IStartupFilter>       StartupFilters { get; }
    public IConfiguration                    Configuration { get; }
    public IWebHostEnvironment               HostingEnvironment { get; }
    public ActivitySource                    ActivitySource { get; }

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

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

由于 ASP.NET Core 应用是作为一个后台服务由 GenericWebHostService 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、异常处理和日志输出等。作为服务器的 IServer 对象被 StartAsync 方法开启之后，又被 StopAsync 方法关闭。

```
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);
    }
    public async Task StopAsync(CancellationToken cancellationToken)
        => Server.StopAsync(cancellationToken);
}
```

我们将实现在 GenericWebHostService 类型中的 StartAsync 方法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 特性来提供的，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：两种针对中间件的注册（调用 IWebHostBuilder 对象的 Configure 方法和注册的 Startup 类型的 Configure 方法）会转换成一个 Action<IApplicationBuilder>委托对象，并将其作为 GenericWebHostServiceOptions 配置选项的 ConfigureApplication 属性。GenericWebHostService 会利用注册的 IApplicationBuilderFactory 工厂创建对应的

IApplicationBuilder 对象，并将该对象作为参数调用这个 Action<IApplicationBuilder>委托对象，就能将注册的中间件转移到 IApplicationBuilder 对象上。在此之前，注册 IStartupFilter 对象的 Configure 方法会被优先调用。表示注册中间件管道的 RequestDelegate 委托对象最终通过调用 IApplicationBuilder 对象的 Build 方法构建。

- 创建 HostingApplication 对象：在得到表示中间件管道的 RequestDelegate 之后，GenericWebHostService 进一步利用它将 HostingApplication 对象创建出来。
- 启动服务器：将 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。

17.4.3　GenericWebHostBuilder

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。GenericWebHostService 及其依赖服务的注册是借助 GenericWebHostBuilder 对象来完成的。在第一代基于 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 对象通过 Build 方法构建。IWebHostBuilder 接口定义了两个 ConfigureServices 重载方法来注册服务。

```
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);
}
```

GenericWebHostBuilder 同时实现了 IWebHostBuilder 接口和 ISupportsUseDefaultServiceProvider 接口。后者定义了一个唯一的 UseDefaultServiceProvider 方法，我们可以利用作为参数的 Action<WebHostBuilderContext, ServiceProviderOptions>委托对象对默认使用的依赖注入容器进行设置。

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

1．服务注册

接下来利用简单的代码来模拟 GenericWebHostBuilder 针对 IWebHostBuilder 接口的实现。我们先来看一看用来注册依赖服务的 ConfigureServices 方法是如何实现的。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。IHostBuilder 接口的 ConfigureServices 方法提供了当前承载上下文的服务注册，承载上下文由承载上下文类型来表示，ASP.NET Core 应用的承载上下文则体现为一个 WebHostBuilderContext 对象。两者的不同之处体现在承载环境的描述上，对应的接口分别为 IHostEnvironment 和 IWebHostEnvironment。ConfigureServices 方法需要调用 GetWebHostBuilderContext 方法将提供的 WebHostBuilderContext 上下文对象转换成 HostBuilderContext 类型。

```
internal class GenericWebHostBuilder :
    IWebHostBuilder,
    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;
    }
}
```

在创建 GenericWebHostBuilder 对象时会以如下方式调用 ConfigureServices 方法注册一系列默认的服务，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（它们都返回同一个服务实例）、与分布式跟踪有关的 ActivitySource 服务和 DistributedContextPropagator 服务（前者用来创建表示跟踪操作的 Activity，后者用来在应用之间传递跟踪上下文），以及分别用来创建 HttpContext 上下文对象、IApplicationBuilder 对象和中间件对象的 IHttpContextFactory、IApplicationBuilderFactory 和 IMiddlewareFactory。它的构造函数中还完成了 GenericWebHostServiceOptions 配置选项的设置，承载 ASP.NET Coer 应用的 GenericWebHostService 服务也是在这里注册的。

```
internal class GenericWebHostBuilder :
    IWebHostBuilder,
    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>();

            services.TryAddSingleton(
              sp => new DiagnosticListener("Microsoft.AspNetCore"));
            services.TryAddSingleton<DiagnosticSource>(
                sp => sp.GetRequiredService<DiagnosticListener>());
           services.TryAddSingleton(sp => new ActivitySource("Microsoft.AspNetCore"));
            services.TryAddSingleton(DistributedContextPropagator.Current);
           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;
            });
            ...
        });
        ...
    }
}
```

2．配置的读/写

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。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_config 字段返回的 IConfiguration 对象，以“键-值”对形式设置和读取配置的 UseSetting 方法与 GetSetting 方法的操作都是这个对象。由静态 Host 类型的 CreateDefaultBuilder 方法创建的 HostBuilder 对象默认将前缀为“DOTNET_”的环境变量作为配置源，ASP.NET Core 应用选择将前缀为“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;
    }
}
```

GenericWebHostBuilder 对象在构造过程中会创建一个 ConfigurationBuilder 对象，并将前缀为“ASPNETCORE_”的环境变量注册为配置源。在利用 ConfigurationBuilder 对象将 IConfiguration 对象构建后，调用 IHostBuilder 对象的 ConfigureHostConfiguration 方法将其合并到承载系统的配置中。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;
    }
}
```

3．默认依赖注入框架配置

GenericWebHostBuilder 通过对 ISupportsUseDefaultServiceProvider 接口的实现将依赖注入框架整合到 ASP.NET Core 应用中。如下面的代码片段所示，实现的 UseDefaultServiceProvider 方法中会根据 ServiceProviderOptions 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;
    }
}
```

4．Hosting Startup

Hosting Startup 是 ASP.NET Core 提供的一个很有用的功能，它使我们可以注册一个独立的程序集来完成一些初始化的工作。具体来说，注册的程序集提供了如下 IHostingStartup 接口的实现类型，并将初始化工作定义在实现的 Configure 方法。此程序集通过标注 HostingStartupAttribute 特性对该类型进行注册。

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

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

WebHostOptions 配置选项提供了如下 3 个与 Hosting Startup 相关的属性。第一个布尔类型的 PreventHostingStartup 属性是此功能的总开关，如果想关闭 Hosting Startup 功能，则只需要将此属性设置为 True。注册的程序集名称需要添加到 HostingStartupExcludeAssemblies 属性中，另一个 HostingStartupExcludeAssemblies 属性则提供了需要排除的程序集。

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

当调用 IHostingStartup 对象的 Configure 方法时需要传入一个 IWebHostBuilder 对象作为参数，这个对象的类型并非 GenericWebHostBuilder，而是如下 HostingStartupWebHostBuilder 类型。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会转移到封装的 GenericWebHostBuilder 对象上。

```
internal class HostingStartupWebHostBuilder :
    IWebHostBuilder,
    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);
    ...
}
```

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，就意味着关闭了此特性。如果 Hosting Startup 特性未被关闭，则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，并得到出注册的 IHostingStartup 实现类型。在通过反射的方式创建对应的 IHostingStartup 对象之后，上面介绍的 HostingStartupWebHostBuilder 对象会被创建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/IHostBuilder 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.");
    ...
}
```

17.4.4　ConfigureWebHostDefaults

虽然 ASP.NET Core 6 推荐使用 Minimal API 的方式来承载 ASP.NET 应用，但是底层采用的依旧是基于 IHost/IHostBuilder 的承载系统。如果利用 Visual Studio 采用传统的模板来创建一个 ASP.NET Core 应用，则生成如下所示的代码。调用静态类型 Host 的 CreateDefaultBuilder 方法在具有默认配置的 IHostBuilder 对象之后，调用了后者的 ConfigureWebHostDefaults 扩展方法，那么这个扩展方法究竟做了些什么呢？

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
```

```
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

ConfigureWebHostDefaults扩展方法内部会调用如下ConfigureWebHost扩展方法，该扩展方法针对承载的 ASP.NET Core 应用所做的设置全部由提供的 Action<IWebHostBuilder>来完成，执行该委托对象传入的参数就是上面介绍的 GenericWebHostBuilder 对象。该对象相当于 IHostBuilder 对象的代理，所以执行 Action<IWebHostBuilder>委托对象产生的结果全部都会转移到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 扩展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 扩展方法中。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 模块的配置都是在 ConfigureWebDefaults 扩展方法中完成的。

```
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, ctx.Configuration);
            }
```

```
        });
        builder.UseKestrel((builderContext, options) =>
        {
            options.Configure(builderContext.Configuration.GetSection("Kestrel"),
                reloadOnChange: true);
        })
        .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>();
            services.AddTransient<IStartupFilter, ForwardedHeadersStartupFilter>();
            services.AddTransient<IConfigureOptions<ForwardedHeadersOptions>,
                ForwardedHeadersOptionsSetup>();

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

17.5　Minimal API

Minimal API 只是在基于 IHost/IHostBuilder 的服务承载系统上进行了封装，它利用 WebApplication 和 WebApplicationBuilder 这两个类型提供了更加简洁的 Minimal API，同时提供了与现有 Minimal API 的兼容。对于由 WebApplication 和 WebApplicationBuilder 构建的承载模型，我们没有必要了解其实现的每一个细节，只需要知道其大致的设计和实现原理，所以本节会采用最简洁的代码模拟这两个类型的实现。

如图 17-8 所示，表示承载应用的 WebApplication 对象是对一个 IHost 对象的封装，而且该类型自身也实现了 IHost 接口，WebApplication 对象还是作为一个 IHost 对象被启动的。作为构建这个 WebApplicationBuilder 则是对一个 IHostBuilder 对象的封装，它对 WebApplication 对象的构建体现在利用封装的 IHostBuilder 对象构建一个对应的 IHost 对象，最终利用 IHost 对象创建 WebApplication 对象。

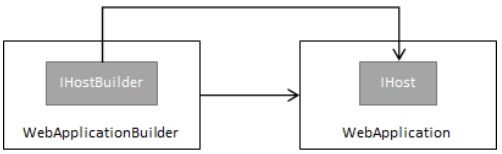

图 17-8　完整的请求处理管道

17.5.1　WebApplication

WebApplication 类型不仅实现了 IHost 接口，还实现 IApplicationBuilder 接口，所以中间件可以直接注册到这个对象上的。该类型还实现了 IEndpointRouteBuilder 接口，所以还能利用它进行路由注册，在第 20 章才会涉及路由，所以现在先忽略该接口的实现。下面的代码模拟了 WebApplication 类型的实现。WebApplication 的构造函中数定义了一个 IHost 类型的参数，并完成了对 IHost 接口所有成员的实现。IApplicationBuilder 接口成员的实现利用创建的 ApplicationBuilder 对象来完成。WebApplication 还提供了一个 BuildRequestDelegate 方法，该方法利用 ApplicationBuilder 对象完成了对中间件管道的构建。

```
public class WebApplication : IApplicationBuilder, IHost
{
    private readonly IHost                    host;
    private readonly ApplicationBuilder      _app;

    public WebApplication(IHost host)
    {
         host  = host;
         app   = new ApplicationBuilder(host.Services);
    }

    IServiceProvider IHost.Services => _host.Services;
    Task IHost.StartAsync(CancellationToken cancellationToken)
        =>  host.StartAsync(cancellationToken);
    Task IHost.StopAsync(CancellationToken cancellationToken)
        =>  host.StopAsync(cancellationToken);

    IServiceProvider IApplicationBuilder.ApplicationServices
        { get =>  app.ApplicationServices; set =>  app.ApplicationServices = value; }
    IFeatureCollection IApplicationBuilder.ServerFeatures
        => _app.ServerFeatures;
    IDictionary<string, object?> IApplicationBuilder.Properties
        =>  app.Properties;
    RequestDelegate IApplicationBuilder.Build()
        =>  app.Build();
    IApplicationBuilder IApplicationBuilder.New()
        => _app.New();
    IApplicationBuilder IApplicationBuilder.Use(
        Func<RequestDelegate, RequestDelegate> middleware)
        => _app.Use(middleware);
```

```
    void IDisposable.Dispose() => _host.Dispose();
    public IServiceProvider Services =>  host.Services;

    internal RequestDelegate BuildRequestDelegate() => _app.Build();
    ...
}
```

WebApplication 额外定义了如下 RunAsync 方法和 Run 方法，它们分别以异步和同步方式启动承载的应用。在调用这两个方法时可以指定监听地址，指定的地址被添加到 IServerAddressesFeature 特性中，而服务器正是利用这个特性来提供监听地址的。

```
public class WebApplication : IApplicationBuilder, IHost
{
    private readonly IHost _host;

    public ICollection<string> Urls
        => _host.Services.GetRequiredService<IServer>().Features
        .Get<IServerAddressesFeature>()?.Addresses ??
        throw new InvalidOperationException("IServerAddressesFeature is not found.");

    public Task RunAsync(string? url = null)
    {
        Listen(url);
        return HostingAbstractionsHostExtensions.RunAsync(this);
    }

    public void Run(string? url = null)
    {
        Listen(url);
        HostingAbstractionsHostExtensions.Run(this);
    }

    private void Listen(string? url)
    {
        if (url is not null)
        {
            var addresses = _host.Services.GetRequiredService<IServer>().Features
               .Get<IServerAddressesFeature>()?.Addresses
               ?? throw new InvalidOperationException(
               "IServerAddressesFeature is not found.");
            addresses.Clear();
            addresses.Add(url);
        }
    }
    ...
}
```

17.5.2　WebApplication 的创建

要创建一个 WebApplication 对象，只需要提供一个对应的 IHost 对象。IHost 对象是通过

IHostBuilder 对象构建的，所以 WebApplicationBuilder 需要一个 IHostBuilder 对象，具体来说是一个 HostBuilder 对象。我们针对 WebApplicationBuilder 对象所做的一切设置最终都需要转移到 HostBuilder 对象上才能生效。

为了提供更加简洁的 Minimal API，WebApplicationBuilder 类型提供了一系列的属性。例如，它利用 Services 属性提供了可以直接进行服务注册的 IServiceCollection 集合，利用 Environment 属性提供了表示当前承载环境的 IWebHostEnvironment 对象，利用 Configuration 属性提供的 ConfigurationManager 对象不仅可以作为 IConfigurationBuilder 对象完成对配置系统的一切设置，它自身也可以作为 IConfiguration 对象为我们提供配置。

WebApplicationBuitder 还定义了 Host 属性和 WebHost 属性，对应类型为 ConfigureHostBuilder 和 ConfigureWebHostBuilder，它们分别实现了 IHostBuilder 接口和 IWebHostBuilder 接口，其目的是复用 IHostBuilder 接口和 IWebHostBuilder 接口承载的 Minimal API（主要是扩展方法）。为了尽可能使用现有方法对 IHostBuilder 对象进行初始化设置，它还使用了一个实现 IHostBuilder 接口的 BootstrapHostBuilder 类型。由这些对象组成了 WebApplicationBuilder 针对 HostBuilder 的构建模型。

如图 17-9 所示，WebApplicationBuilder 的所有工作都是为了构建它封装的 HostBuilder 对象。当 WebApplicationBuilder 初始化时，它除了创建 HostBuilder 对象，还创建存储服务注册的 IServiceCollection 对象，以及用来对配置进行设置的 ConfigurationManager 对象。接下来创建一个 BootstrapHostBuilder 对象，将它参数调用相应的方法（如 ConfigureWebHostDefaults 方法）和初始化设置收集起来，并将收集的服务注册和配置系统的设置分别转移到创建的 IServiceCollection 对象和 ConfigurationManager 对象中，其他设置直接应用到封装的 HostBuilder 对象上。

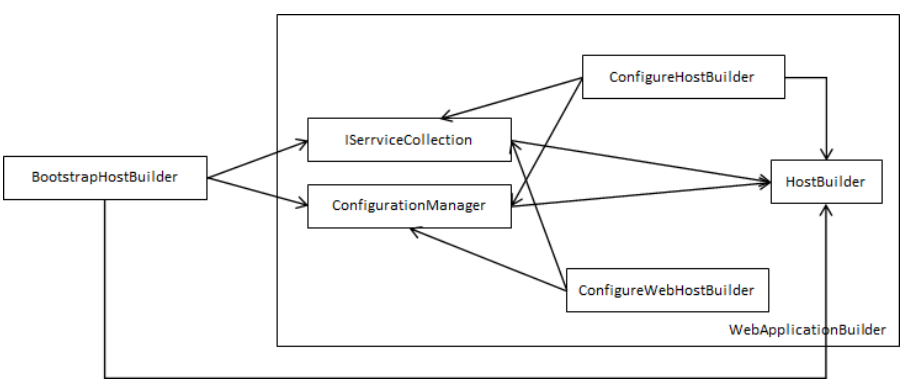

图 17-9　HostBuilder 构建模型

WebApplicationBuilder 在此之后会创建表示承载环境的 IWebHostEnvironment 对象，并对 Environment 属性进行初始化。在得到表示承载上下文的 WebHostBuilderContext 对象之后，上述的 ConfigureHostBuilder 对象和 ConfigureWebHostBuilder 对象被创建，并赋值给 Host 属性和 WebHost 属性。与 BootstrapHostBuilder 作用类似，我们利用这两个对象所做的设置最终都会转

移到上述的 3 个对象中。

当利用 WebApplicationBuilder 进行 WebApplication 对象创建时，IServiceCollection 对象存储的服务注册和 ConfigurationManager 对象承载配置最终转移到 HostBuilder 对象上。此时再利用后者创建对应的 IHost 对象，表示承载应用的 WebApplication 对象最终由 IHost 对象创建。

1．BootstrapHostBuilder

如下所示为模拟 BootstrapHostBuilder 类型的定义。正如上面所说，BootStrapHostBuilder 的作用是收集初始化 IHostBuilder 对象提供的设置并将它们分别应用到指定的 IServiceCollection 对象、ConfigurationManager 对象和 IHostBuilder 对象上。这个使命体现在 BootstrapHostBuilder 的 Apply 方法上，该方法还通过一个输出参数返回创建的 HostBuilderContext 上下文对象。

```
public class BootstrapHostBuilder : IHostBuilder
{
    private readonly List<Action<IConfigurationBuilder>>
         configureHostConfigurations = new();
    private readonly List<Action<HostBuilderContext, IConfigurationBuilder>>
         configureAppConfigurations = new();
    private readonly List<Action<HostBuilderContext, IServiceCollection>>
        _configureServices = new();
    private readonly List<Action<IHostBuilder>>  others = new();

    public IDictionary<object, object> Properties { get; }
        = new Dictionary<object, object>();
    public IHost Build()=> throw new NotImplementedException();
    public IHostBuilder ConfigureHostConfiguration(
        Action<IConfigurationBuilder> configureDelegate)
    {
         configureHostConfigurations.Add(configureDelegate);
        return this;
    }
    public IHostBuilder ConfigureAppConfiguration(
        Action<HostBuilderContext, IConfigurationBuilder> configureDelegate)
    {
         configureAppConfigurations.Add(configureDelegate);
        return this;
    }
    public IHostBuilder ConfigureServices(
        Action<HostBuilderContext, IServiceCollection> configureDelegate)
    {
         configureServices.Add(configureDelegate);
        return this;
    }
    public IHostBuilder UseServiceProviderFactory<TContainerBuilder>(
        IServiceProviderFactory<TContainerBuilder> factory)
    {
         others.Add(builder => builder.UseServiceProviderFactory(factory));
        return this;
    }
```

```
public IHostBuilder UseServiceProviderFactory<TContainerBuilder>(
    Func<HostBuilderContext, IServiceProviderFactory<TContainerBuilder>> factory)
{
    _others.Add(builder => builder.UseServiceProviderFactory(factory));
    return this;
}
public IHostBuilder ConfigureContainer<TContainerBuilder>(
    Action<HostBuilderContext, TContainerBuilder> configureDelegate)
{
     others.Add(builder => builder.ConfigureContainer(configureDelegate));
    return this;
}

internal void Apply(IHostBuilder hostBuilder,  ConfigurationManager configuration,
     IServiceCollection services, out HostBuilderContext builderContext)
{
    // 初始化宿主配置
    var hostConfiguration = new ConfigurationManager();
     configureHostConfigurations.ForEach(it => it(hostConfiguration));

    // 创建承载环境
    var environment = new HostingEnvironment()
    {
        ApplicationName = hostConfiguration[HostDefaults.ApplicationKey],
        EnvironmentName = hostConfiguration[HostDefaults.EnvironmentKey]
           ?? Environments.Production,
        ContentRootPath = HostingPathResolver
            .ResolvePath(hostConfiguration[HostDefaults.ContentRootKey])
    };
    environment.ContentRootFileProvider
        = new PhysicalFileProvider(environment.ContentRootPath);

    // 创建 HostBuilderContext 上下文对象
    var hostContext = new HostBuilderContext(Properties)
    {
        Configuration = hostConfiguration,
        HostingEnvironment = environment,
    };

    // 将宿主配置添加到 ConfigurationManager 中
    configuration.AddConfiguration(hostConfiguration, true);

    // 初始化应用配置
     configureAppConfigurations.ForEach(it => it(hostContext, configuration));

    // 收集服务注册
     configureServices.ForEach(it => it(hostContext, services));

    // 将依赖注入容器设置应用到指定的 IHostBuilder 对象上
    _others.ForEach(it => it(hostBuilder));
```

```
        // 将自定义属性转移到指定的 IHostBuilder 对象上
        foreach (var kv in Properties)
        {
            hostBuilder.Properties[kv.Key] = kv.Value;
        }

        builderContext = hostContext;
    }
}
```

除了 Build 方法，IHostBuilder 接口中定义的所有方法的参数都是委托对象，所以实现的这些方法将提供的委托对象收集起来。在 Apply 方法中，我们通过执行这些委托对象，将初始化设置应用到指定的 IServiceCollection 对象、ConfigurationManager 对象和 IHostBuilder 对象上，并根据初始化宿主配置创建表示承载环境的 HostingEnvironment 对象。Apply 方法最后根据承载环境结合配置将 HostBuilderContext 上下文对象创建出来，并以输出参数的形式返回。

```
internal static class HostingPathResolver
{
    public static string ResolvePath(string? contentRootPath)
        => ResolvePath(contentRootPath, AppContext.BaseDirectory);
    public static string ResolvePath(string? contentRootPath, string basePath)
        => string.IsNullOrEmpty(contentRootPath)
        ? Path.GetFullPath(basePath)
        : Path.IsPathRooted(contentRootPath)
        ? Path.GetFullPath(contentRootPath)
        : Path.GetFullPath(Path.Combine(Path.GetFullPath(basePath), contentRootPath));
}
```

2．ConfigureHostBuilder

ConfigureHostBuilder 对象是在应用了 BootstrapHostBuilder 收集的初始化设置之后创建的，在创建该对象时提供了 HostBuilderContext 上下文对象、ConfigurationManager 对象和 IServiceCollection 对象。将提供的服务注册直接添加到 IServiceCollection 对象中，针对配置的设置已经应用到 ConfigurationManager 对象，直接针对 IHostBuilder 对象的设置则利用_configureActions 字段暂存起来。

```
public class ConfigureHostBuilder : IHostBuilder
{
    private readonly ConfigurationManager            configuration;
    private readonly IServiceCollection             _services;
    private readonly HostBuilderContext             _context;
    private readonly List<Action<IHostBuilder>>      configureActions = new();

    internal ConfigureHostBuilder(HostBuilderContext context,
        ConfigurationManager configuration, IServiceCollection services)
    {
        _configuration        = configuration;
         services             = services;
        _context              = context;
```

```
    }

    public IDictionary<object, object> Properties => _context.Properties;
    public IHost Build() => throw new NotImplementedException();
    public IHostBuilder ConfigureAppConfiguration(
        Action<HostBuilderContext, IConfigurationBuilder> configureDelegate)
        => Configure(() => configureDelegate(_context, _configuration));

    public IHostBuilder ConfigureHostConfiguration(
        Action<IConfigurationBuilder> configureDelegate)
    {
        var applicationName   = _configuration[HostDefaults.ApplicationKey];
        var contentRoot       =  context.HostingEnvironment.ContentRootPath;
        var environment       = _configuration[HostDefaults.EnvironmentKey];

        configureDelegate( configuration);

        // 与环境相关的 3 个配置不允许改变
        Validate(applicationName, HostDefaults.ApplicationKey,
            "Application name cannot be changed.");
        Validate(contentRoot, HostDefaults.ContentRootKey,
            "Content root cannot be changed.");
        Validate(environment, HostDefaults.EnvironmentKey,
            "Environment name cannot be changed.");

        return this;

        void Validate(string previousValue, string key, string message)
        {
            if (!string.Equals(previousValue,  configuration[key],
                StringComparison.OrdinalIgnoreCase))
            {
                throw new NotSupportedException(message);
            }
        }
    }

    public IHostBuilder ConfigureServices(
      Action<HostBuilderContext, IServiceCollection> configureDelegate)
      => Configure(() => configureDelegate(_context, _services));

    public IHostBuilder UseServiceProviderFactory<TContainerBuilder>(
      IServiceProviderFactory<TContainerBuilder> factory)
      => Configure(() =>  configureActions.Add(
      b => b.UseServiceProviderFactory(factory)));

    public IHostBuilder UseServiceProviderFactory<TContainerBuilder>(
        Func<HostBuilderContext, IServiceProviderFactory<TContainerBuilder>> factory)
        => Configure(
            () => _configureActions.Add(b => b.UseServiceProviderFactory(factory)));
```

```
    public IHostBuilder ConfigureContainer<TContainerBuilder>(
        Action<HostBuilderContext, TContainerBuilder> configureDelegate)
        => Configure(
            () =>  configureActions.Add(b => b.ConfigureContainer(configureDelegate)));

    private IHostBuilder Configure(Action configure)
    {
        configure();
        return this;
    }

    internal void Apply(IHostBuilder hostBuilder)
        => _configureActions.ForEach(op => op(hostBuilder));
}
```

WebApplicationBuilder 对象一旦被创建后，针对承载环境的配置是不能改变的，所以 ConfigureHostBuilder 的 ConfigureHostConfiguration 方法针对此添加了相应的验证。两个 UseServiceProviderFactory 方法和 ConfigureContainer 方法针对依赖注入容器的设置最终需要应用到 IHostBuilder 对象上，所以我们将方法中提供的委托对象利用_configureActions 字段暂存起来，并最终利用 Apply 方法应用到指定的 IHostBuilder 对象上。

3．ConfigureWebHostBuilder

ConfigureWebHostBuilder 对象同样是在应用了 BootstrapHostBuilder 提供的初始化设置后创建的，在创建该对象时能够提供 WebHostBuilderContext 上下文对象和承载配置与服务注册的 ConfigurationManager 对象及 IServiceCollection 对象。由于 IWebHostBuilder 接口定义的方法只涉及服务注册和配置的设置，所以由方法提供的委托对象可以直接应用到这两个对象上。

```
public class ConfigureWebHostBuilder : IWebHostBuilder, ISupportsStartup
{
    private readonly WebHostBuilderContext   _builderContext;
    private readonly IServiceCollection       services;
    private readonly ConfigurationManager     configuration;

    public ConfigureWebHostBuilder(WebHostBuilderContext builderContext,
         ConfigurationManager configuration, IServiceCollection services)
    {
         builderContext       = builderContext;
        _services             = services;
         configuration        = configuration;
    }

    public IWebHost Build()=> throw new NotImplementedException();
    public IWebHostBuilder ConfigureAppConfiguration(
        Action<WebHostBuilderContext, IConfigurationBuilder> configureDelegate)
        => Configure(() => configureDelegate( builderContext,  configuration));
    public IWebHostBuilder ConfigureServices(
      Action<IServiceCollection> configureServices)
```

```
        => Configure(() => configureServices(_services));
    public IWebHostBuilder ConfigureServices(
        Action<WebHostBuilderContext, IServiceCollection> configureServices)
        => Configure(() => configureServices(_builderContext, _services));
    public string? GetSetting(string key) =>  configuration[key];
    public IWebHostBuilder UseSetting(string key, string? value)
        => Configure(() => _configuration[key] = value);

    IWebHostBuilder ISupportsStartup.UseStartup(Type startupType)
        => throw new NotImplementedException();
    IWebHostBuilder ISupportsStartup.UseStartup<TStartup>(
        Func<WebHostBuilderContext, TStartup> startupFactory)
    => throw new NotImplementedException();
    IWebHostBuilder ISupportsStartup.Configure(Action<IApplicationBuilder> configure)
        => throw new NotImplementedException();
    IWebHostBuilder ISupportsStartup.Configure(
        Action<WebHostBuilderContext, IApplicationBuilder> configure)
        => throw new NotImplementedException();

    private IWebHostBuilder Configure(Action configure)
    {
        configure();
        return this;
    }
}
```

前文已经提到，传统承载方式将初始化操作定义在注册的 Startup 类型的编程方式已经不被 Minima API 支持，所以 WebApplicationBuilder 本不该实现 ISupportsStartup 接口，但是希望用户在采用这种编程方式时得到显式提醒，所以依然让它实现该接口，并在实现的方法中抛出 NotImplementedException 类型的异常。

4．WebApplicationBuilder

如下代码片段模拟了 WebApplicationBuilder 针对 WebApplication 的构建。利用它的构造函数创建一个 BootstrapHostBuilder 对象，调用它的 ConfigureDefaults 扩展方法和 ConfigureWebHostDefaults 扩展方法将初始化设置收集起来。ConfigureWebHostDefaults 扩展方法利用提供的 Action<IWebHostBuilder>委托对象进行中间件的注册，由于中间件的注册被转移到 WebApplication 对象上，并且它提供了一个 BuildRequestDelegate 方法返回由注册中间件组成的管道，所以在这里只需调用创建 WebApplication 对象（通过_application 字段表示，此时 WebApplication 对象尚未被创建，当中间件真正被注册时会被创建出来）的方法，并将返回的 RequestDelegate 对象作为参数调用 IApplicationBuilder 接口的 Run 方法将中间件管道注册为请求处理器。

```
public class WebApplicationBuilder
{
    private readonly HostBuilder      _hostBuilder = new HostBuilder();
    private WebApplication            _application;
```

```
public ConfigurationManager      Configuration { get; } =
    new ConfigurationManager();
public IServiceCollection        Services { get; } = new ServiceCollection();
public IWebHostEnvironment       Environment { get; }
public ConfigureHostBuilder      Host { get; }
public ConfigureWebHostBuilder   WebHost { get; }
public ILoggingBuilder           Logging { get; }

public WebApplicationBuilder(WebApplicationOptions options)
{
  //创建 BootstrapHostBuilder 并利用它收集初始化过程中设置的配置、服务和依赖注入容器的设置
    var args = options.Args;
    var bootstrap = new BootstrapHostBuilder();
    bootstrap
        .ConfigureDefaults(null)
        .ConfigureWebHostDefaults(webHostBuilder => webHostBuilder.Configure(
            app => app.Run(_application.BuildRequestDelegate())))
        .ConfigureHostConfiguration(config => {
            // 添加命令行配置源
            if (args?.Any() == true)
            {
                config.AddCommandLine(args);
            }

            // 将 WebApplicationOptions 配置选项转移到配置中
            Dictionary<string, string>? settings = null;
            if (options.EnvironmentName is not null) (settings ??= new())
                [HostDefaults.EnvironmentKey] = options.EnvironmentName;
            if (options.ApplicationName is not null) (settings ??= new())
                [HostDefaults.ApplicationKey] = options.ApplicationName;
            if (options.ContentRootPath is not null) (settings ??= new())
                [HostDefaults.ContentRootKey] = options.ContentRootPath;
            if (options.WebRootPath is not null) (settings ??= new())
               [WebHostDefaults.WebRootKey] = options.EnvironmentName;
            if (settings != null)
            {
                config.AddInMemoryCollection(settings);
            }
        });

    // 将 BootstrapHostBuilder 收集的配置和服务转移到 Configuration 和 Services 上
    // 将应用到 BootstrapHostBuilder 上针对依赖注入容器的设置转移到_hostBuilder 字段上
    // 得到 BuilderContext 上下文对象
    bootstrap.Apply( hostBuilder, Configuration, Services, out var builderContext);

    // 如果提供了命令行参数，则在 Configuration 上添加对应的配置源
    if (options.Args?.Any() == true)
    {
        Configuration.AddCommandLine(options.Args);
```

```
        }

        // 创建 WebHostBuilderContext 上下文对象
        // 初始化 Host 属性、WebHost 属性和 Logging 属性
        var webHostContext = (WebHostBuilderContext)builderContext
            .Properties[typeof(WebHostBuilderContext)];
        Environment = webHostContext.HostingEnvironment;
        Host = new ConfigureHostBuilder(builderContext, Configuration, Services);
        WebHost = new ConfigureWebHostBuilder(webHostContext, Configuration, Services);
           Logging = new LogginigBuilder(Services);
    }

    public WebApplication Build()
    {
        // 将 ConfigurationManager 的配置转移到 hostBuilder 字段上
         hostBuilder.ConfigureAppConfiguration(builder =>
        {
            builder.AddConfiguration(Configuration);
            foreach (var kv in ((IConfigurationBuilder)Configuration).Properties)
            {
                builder.Properties[kv.Key] = kv.Value;
            }
        });

        // 将添加的服务注册转移到 hostBuilder 字段上
        _hostBuilder.ConfigureServices((_, services) =>
        {
            foreach (var service in Services)
            {
                services.Add(service);
            }
        });

        // 将应用到 Host 属性上的设置转移到 hostBuilder 字段上
        Host.Apply(_hostBuilder);

        // 利用_hostBuilder 字段创建的 IHost 对象创建 WebApplication
        return  application = new WebApplication( hostBuilder.Build());
    }
}
```

接下来 BootstrapHostBuilder 的 ConfigureHostConfiguration 方法被调用。我们利用它将提供的 WebApplicationOptions 配置选项转移到 BootstrapHostBuilder 针对宿主的配置上。将 IHostBuilder 初始化设置应用到 BootstrapHostBuilder 对象上之后，调用其 Apply 方法将这些设置分别转移到承载服务注册和配置的 IServiceCollection 对象和 ConfigurationManager 对象，以及封装的 HostBuilder 对象上。

Apply 方法利用输出参数提供了 HostBuilderContext 上下文对象，并从中提取 WebHostBuilderContext 上下文对象（GenericWebHostBuilder 会将创建的 WebHostBuilderContext

上下文对象置于 HostBuilderContext 上下文对象的属性字典中）。我们利用这个上下文对象将 ConfigureHostBuilder 对象和 ConfigureWebHostBuilder 对象创建出来，并作为 Host 属性和 WebHost 属性。用于对日志进行进一步设置的 Logging 属性也在这里被初始化，返回的 LoggingBuilder 对象只是对 IServiceCollection 对象的简单封装。

构建 WebApplication 对象的 Build 方法分别调用 ConfigureAppConfiguration 方法和 ConfigureServices 方法将 ConfigurationManager 对象和 IServiceCollection 对象承载的配置与服务注册转移到 HostBuilder 对象上。接下来提取 Host 属性返回的 ConfigureHostBuilder 对象，并调用其 Apply 方法将应用在该对象上的依赖注入容器的设置转移到 HostBuilder 对象上。至此所有的设置全部转移到 HostBuilder 对象上，调用其 Build 方法创建对应的 IHost 对象后，利用 IHost 对象创建代码承载应用的 WebApplication 对象。我们将这个对象赋值到_application 字段上，前面调用 ConfigureWebHostDefaults 扩展方法提供的委托对象会将它的 BuildRequestDelegate 方法构建的中间件管道作为请求处理器。

17.5.3　工厂方法

表示承载应用的 WebApplication 对象是由 WebApplicationBuilder 创建的，但是我们一般不会通过调用构造函数的方式来创建 WebApplicationBuilder 对象，这违背了“面向接口”的编程原则，所以我们都会使用 WebApplication 类型提供的静态工厂方法来创建它。WebApplication 除了提供了 3 个用于创建 WebApplicationBuilder 对象的 CreateBuilder 重载方法，还提供了一个直接创建 WebApplication 对象的 Create 方法。

```
public sealed class WebApplication
{
    public static WebApplicationBuilder CreateBuilder() =>
        new WebApplicationBuilder(new WebApplicationOptions());

    public static WebApplicationBuilder CreateBuilder(string[] args)
    {
        var options = new WebApplicationOptions();
        options.Args = args;
        return new WebApplicationBuilder(options);
    }

    public static WebApplicationBuilder CreateBuilder(WebApplicationOptions options) =>
        new WebApplicationBuilder(options, null);

    public static WebApplication Create(string[]? args = null)
    {
        var options = new WebApplicationOptions();
        options.Args = args;
        return new WebApplicationBuilder(options).Build();
    }
}
```

本节通过 WebApplication 和 WebApplicationBuilder 这两个类型的实现模拟来介绍 Minimal API 的实现原理。一方面为了让讲解更加清晰，另一方面也出于篇幅的限制，不得不省略很多细枝末节的内容，但是设计思想和实现原理别无二致。上面提供的源代码也不是伪代码，如下所示为“模拟的 Minimal API”构建的 ASP.NET Core 应用，它是可以正常运行的。如果读者对真实的 Minimal API 实现感兴趣，则可以将本节作为一个“向导”去探寻“真实的 Minimal API”。（S1704）

```
var app = App.WebApplication.Create();
app.Run(httpContext => httpContext.Response.WriteAsync("Hello World!"));
app.Run();
```