第 1 章 初始化 ASP.NET Core 应用程序

本章要点

- WebApplicationBuilder 类与 WebApplication 类的关系
- 运行 ASP.NET Core 应用程序
- 为 Web 服务器设置 URL
- 处理应用生命周期事件

1.1 应用程序的启动过程

ASP.NET Core 应用程序在启动之前，需要经过以下步骤。

（1）创建 WebApplicationBuilder 对象实例。

（2）（可选）向 Services 集合添加所需要的服务类型。

（3）（可选）配置相关的选项类型。

（4）调用 Build 方法创建 WebApplication 实例。

（5）（可选）组建 HTTP 处理管线。

（6）启动应用程序，同时启动内置的 Kestrel 服务器。

要创建基本的 ASP.NET Core 应用程序项目，可以执行 dotnet new web 命令。例如，执行以下命令将创建一个名为 Demo 的 ASP.NET Core 项目。

```
dotnet new web -n Demo -o .
```

其中，new 命令指明即将完成的操作为创建新的应用程序项目；web 参数表示项目类型，此处指 ASP.NET Core 项目。若希望创建控制台应用程序，可以执行以下命令。

```
dotnet new console …
```

-n 参数表示待创建项目的名称，示例中为 Demo；-o 参数指定被创建项目的存放路径，上述示例中，“.”表示存放到当前工作目录下。也可以使用绝对路径，如

```
dotnet new web C:\test\mydir          // Windows
dotnet new web /home/user/apps        // Linux
```

Web 项目模板将生成以下 C#代码。

```
// 创建 WebApplicationBuilder 对象
var builder = WebApplication.CreateBuilder(args);

/* 此处可以向 Services 集合添加服务类，或配置应用程序 */

// 创建 WebApplication 对象
var app = builder.Build();

// 构建 HTTP 管道
app.MapGet("/", () =>"Hello World!");

// 启动应用程序
app.Run();
```

1.2 WebApplicationBuilder 类

WebApplicationBuilder 类没有公共的构造函数，因此不能直接实例化。只能通过 WebApplication 类的 CreateBuilder()方法获得 WebApplicationBuilder 实例。此方法是静态成员，可在代码中直接调用，如

```
WebApplicationBuilder builder = WebApplication.CreateBuilder();
```

如果 ASP.NET Core 应用程序需要通过命令行参数获取配置参数，则应该将传递给程序入口点（Main 方法）的参数继续传递给 CreateBuilder()方法，即

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

获得 WebApplicationBuilder 实例后，就可以使用它配置应用程序了，如向依赖注入容器（ServiceCollection）添加服务类、设置 Kestrel 服务器、添加配置参数的数据源等。CreateBuilder()方法在创建 WebApplicationBuilder 实例的过程中也会添加一些默认的配置，以保证 ASP.NET Core 应用程序的基本代码能够正常运行。这些默认配置包括：

（1）为内置 Web 服务器 Kestrel 设定默认值；

（2）配置应用程序的运行环境；

（3）默认添加的配置，来源有命令行参数、环境变量、appsettings.json 文件；

（4）配置默认的日志输出方案；

（5）将当前目录设置为应用的内容根目录；

（6）配置 IIS（Internet Information Services）集成，以便能在 Windows 操作系统以 IIS 服务器托管 ASP.NET Core 应用程序。

应用程序配置完成后，调用 Build()方法，创建并返回 WebApplication 实例。调用 Build()方法后，依赖注入容器将变为只读状态，无法修改。运行以下代码后会发生错误。

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
/* 配置应用程序 */
```

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

// 错误：此时 Services 无法修改
builder.Services.AddRazorPages();
```

WebApplicationBuilder 对象只是为应用程序的运行准备好相关的配置参数，而不会真正启动 ASP.NET Core 应用程序。要启动应用程序，需要调用 WebApplication 实例的相关方法。

1.3　启动应用程序

WebApplicationBuilder.Build()方法返回 WebApplication 实例后，可以适当地安排 HTTP 管线，以便能处理并响应来自客户端的 HTTP 请求，如

```
var app = builder.Build();
// 运行一个简单的中间件
app.Use(async (httpcontext, next) =>
{
    // ...
    await next();
});
// 构造一个终结点，HTTP 管线在此处结束
// 并沿着调用堆栈将响应消息回传给客户端
app.MapGet("/", () =>"Hello World!");
```

最后，调用 WebApplication 实例的 StartAsync()、Run()、RunAsync()等方法正式启动应用程序。

StartAsync()方法完成应用程序启动后会马上返回。此时，开发人员需要想办法保证应用程序能持续运行并循环处理客户端请求，否则应用程序启动之后会立即退出。当需要关闭应用程序时，应调用 StopAsync()方法。

下面的示例演示了如何按 Esc 键退出应用程序。

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();                          // 构建应用程序

app.MapGet("/", () =>"Hello World!");               // HTTP 管线

// 启动应用程序
await app.StartAsync();
// 循环检测用户是否按下了 Esc 键
while(true)
{
    ConsoleKeyInfo key = Console.ReadKey(intercept: true);
    if(key.Key == ConsoleKey.Escape)
    {
        // 跳出循环
        Console.WriteLine("*** 应用程序即将退出 ***");
        break;
```

```
    }
}
await app.StopAsync();                                    // 停止应用程序
```

调用 StartAsync()方法后，只要应用程序顺利启动，该方法立即返回；接着程序进入一个死循环（while 循环的条件永远都是 true，此循环会无限次地执行），在循环代码内部，等待用户的键盘输入。如果用户按下的是 Esc 键，就会跳出 while 循环。

跳出循环后，就会调用 StopAsync()方法，停止应用程序，如图 1-1 所示。

```
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5050
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: G:\Demo\DemoApp\DemoApp\
*** 应用程序即将退出 ***
info: Microsoft.Hosting.Lifetime[0]
      Application is shutting down...
```

图 1-1　按下 Esc 键后应用程序停止

如果调用的是 Run()或 RunAsync()方法，开发人员不需要编写额外的代码等待应用程序退出，因为 Run()或 RunAsync()方法被调用后会处于等待状态，直到应用程序退出才会返回。下面的示例演示 Run()方法的使用。

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

/* 配置应用程序 */

// 构建应用程序实例
var app = builder.Build();

// …

// 启动应用程序
app.Run();
```

如果要调用 RunAsync()方法，就要使用异步等待。

```
await app.RunAsync();
```

应用程序运行后，可以按 Ctrl+C 快捷键退出。

1.4　使用 Host 初始化应用程序

Host（翻译为“主机”）是一种具有特定功能的服务类型，它封装并管理与当前应用程序有关的各种资源，如应用程序的配置数据、依赖注入容器，以及服务容器中注册的所有实现了 IHostedService 接口的对象。当 Host 启动时，它会在服务容器中检索所有实现 IHostedService

接口的类型实例，然后逐个调用它们的 StartAsync()方法。

1.4.1　通用主机

Host 可以用于许多典型的.NET 应用程序中（如控制台应用程序），称为通用主机（Generic Host）。

使用通用主机构建应用程序的大致过程如下。

（1）创建 HostBuilder 实例。可以直接使用该类的构造函数进行实例化，也可以调用 Host 类的 CreateDefaultBuilder()静态方法获取 HostBuilder 实例。

（2）通过调用 HostBuilder 的成员方法或扩展方法完成配置，如调用 ConfigureServices()方法向支持依赖注入的服务容器添加服务类型。

（3）调用 Build()方法创建 Host 实例。

（4）调用 Host 实例的 StartAsync()、Run()、RunAsync()等方法启动主机。

（5）（可选）调用 StopAsync()方法停止通用主机。

1.4.2　示例：简单的通用主机

下面的示例将演示一个简单的使用通用主机的控制台应用程序，并在其中启动两个服务项目。步骤如下。

（1）创建控制台应用程序项目。

（2）执行以下命令，添加 Nuget 包引用。

```
dotnet add package Microsoft.Extensions.Hosting
dotnet add package Microsoft.Extensions.DependencyInjection
```

（3）在代码文件中引入以下命名空间。

```
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.DependencyInjection;
```

（4）定义两个服务类（DemoService1 和 DemoService2），它们都实现 IHostedService 接口。这两个类将作为服务运行在通用主机上。

```
// 第 1 个服务类
public sealed class DemoService1 : IHostedService
{
    public Task StartAsync(CancellationToken cancellationToken)
    {
        Console.WriteLine("服务 1：开始运行");
        return Task.CompletedTask;
    }

    public Task StopAsync(CancellationToken cancellationToken)
    {
        Console.WriteLine("服务 1：停止运行");
        return Task.CompletedTask;
```

```
    }
}

// 第 2 个服务类
public sealed class DemoService2 : IHostedService
{
    public Task StartAsync(CancellationToken cancellationToken)
    {
        Console.WriteLine("服务 2：开始运行");
        return Task.CompletedTask;
    }

    public Task StopAsync(CancellationToken cancellationToken)
    {
        Console.WriteLine("服务 2：停止运行");
        return Task.CompletedTask;
    }
}
```

主要是实现两个方法：StartAsync()方法在服务被启动时调用；同理，当服务被停止时则调用 StopAsync()方法。本示例中仅调用 Console.WriteLine()方法输出简单的文本，以便从控制台输出得知服务的运行状态。

（5）创建 HostBuilder 实例。

```
var hostBuilder = Host.CreateDefaultBuilder(args);
```

（6）调用 ConfigureServices()方法，将上述步骤中实现的两个服务类注册到服务容器中。

```
hostBuilder.ConfigureServices(services =>
{
    services.AddSingleton<IHostedService, DemoService1>();
    services.AddSingleton<IHostedService, DemoService2>();
});
```

AddSingleton()方法表示这两个服务类在整个应用程序生命周期内是单个实例的——它们的构造函数只调用一次。

（7）创建通用主机实例。

```
var host = hostBuilder.Build();
```

（8）启动通用主机。

```
await host.StartAsync();
Console.WriteLine("通用主机已启动，按 Enter 键退出");
Console.ReadLine();
await host.StopAsync();
```

调用 StartAsync()方法启动通用主机，随后 Console.ReadLine()方法会等待用户的键盘输入。只要按 Enter 键，ReadLine()方法会立即返回，然后调用 StopAsync()方法停止通用主机。

运行示例程序后，若看到屏幕上输出“服务 1：开始运行”和“服务 2：开始运行”，则表明托管在通用主机上的两个服务已顺利启动；此时按 Enter 键，通用主机停止，托管在其中

的两个服务也会停止，并在屏幕上输出相关信息，如图 1-2 所示。

```
服务1：开始运行
服务2：开始运行
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Production
info: Microsoft.Hosting.Lifetime[0]
      Content root path: G:\Demo\DemoApp\DemoApp\bin\Debug\net6.0
通用主机已启动，按Enter键退出

info: Microsoft.Hosting.Lifetime[0]
      Application is shutting down...
服务2：停止运行
服务1：停止运行
```

图 1-2　在通用主机上运行两个服务示例

1.4.3　Web 主机

对于 ASP.NET Core 项目，新版本推荐使用前文所介绍的初始化方法，即使用 WebApplication-Builder 类型实例构建 WebApplication 对象。而使用 Web 主机仅用于对旧版本的兼容，或者同时使用通用主机和 Web 主机的项目。

Web 主机有两种使用方法。

第 1 种方法是先创建用于通用主机的 HostBuilder 实例，然后再配置 Web 主机，最后创建通用主机实例并启动主机。请思考下面的示例代码。

```
// 创建通用主机的 Builder
var hostBuilder = Host.CreateDefaultBuilder(args);
// 添加 Web 主机配置
hostBuilder.ConfigureWebHostDefaults(webhostBuilder =>
{
    // 添加服务
    webhostBuilder.ConfigureServices(services =>
    {
        services.AddControllers();
    })
    // 构建 HTTP 管线
    .Configure(appBuilder =>
    {
        appBuilder.Use(async (ctx, next) =>
        {
            ctx.Response.Headers.Add("Access-Mode", "Read");
            await next();
        })
        .UseRouting()
        .UseEndpoints(endPoints =>
        {
            endPoints.Map("/", () =>"示例应用程序");
        });
```

```
        });
    });

    // 构建主机实例
    var host = hostBuilder.Build();
    // 启动主机
    await host.RunAsync();
```

要在 HostBuilder 上添加 Web 主机的配置，可以调用 ConfigureWebHost()方法，也可以调用 ConfigureWebHostDefaults()方法。上述示例调用了 ConfigureWebHostDefaults()方法，它会为 Web 主机设置常用的默认参数，如使用内置的 Kestrel 服务器、IIS 集成等。

ConfigureServices()方法用于向依赖注入容器添加服务类型，定义应用程序需要用到的功能；Configure()方法用于构建 HTTP 管线，处理客户端的 HTTP 请求。

最后，调用 Build()方法创建通用主机实例，并调用 RunAsync()方法启动主机，ASP.NET Core 应用程序开始运行。

使用 Web 主机的第 2 种方法是调用 WebHost 类的 CreateDefaultBuilder()静态方法创建 WebHostBuilder 实例，接着配置服务容器和 HTTP 管线，最后调用 Build()方法创建 Web 主机实例并运行它。示例代码如下。

```
    // 创建 WebHostBuilder 实例
    var webhostBuilder = WebHost.CreateDefaultBuilder(args);
    // 配置依赖注入容器
    webhostBuilder.ConfigureServices(services =>
    {
        services.AddControllers();
    });
    // 构建 HTTP 管线
    webhostBuilder.Configure(app =>
    {
        app.UseRouting();
        app.UseEndpoints(endPoints =>
        {
            endPoints.MapControllers();
            endPoints.Map("/", () =>"示例应用程序");
        });
    });

    // 创建 Web 主机实例
    var host = webhostBuilder.Build();
    // 启动 Web 主机
    host.Run();
```

除了使用 WebHost.CreateDefaultBuilder()方法，还可以直接实例化 WebHostBuilder 对象，并手动配置 Web 主机，如

```
    var webhostBuilder = new WebHostBuilder();
    // 使用内置的 Kestrel 服务器和 IIS 集成功能
```

```
webhostBuilder.UseKestrel().UseIISIntegration();
// 添加应用程序的配置源
webhostBuilder.ConfigureAppConfiguration(cfg =>
{
    // 命令行参数
    cfg.AddCommandLine(args);
    // 环境变量
    cfg.AddEnvironmentVariables();
    // JSON 文件
    cfg.AddJsonFile("appsettings.json");
});
```

1.5　设置应用程序的 URL

Web 服务器需要绑定一个或多个有效的统一资源定位符（Uniform Resource Locator，URL），并在这些 URL 上侦听连接。客户端（通常是 Web 浏览器）通过 URL 找到要访问的 Web 应用程序。因此，URL 可以标识应用程序的唯一性。

为 ASP.NET Core 应用程序设置 URL 有多种方法，大体上可分为两类。

（1）硬编码。即直接把要使用的 URL 写到程序代码中，其缺点是一旦修改了 URL，就需要重新编译应用程序。

（2）应用程序配置。此方法比较灵活，修改 URL 后不需要重新编译应用程序。参数的配置方案比较多，如命令行参数、环境变量、配置文件等。

1.5.1　调用 UseUrls()方法

调用 WebApplication.CreateBuilder()方法创建 WebApplicationBuilder 实例后，就可以通过它的 WebHost 属性调用 UseUrls()方法为 ASP.NET Core 应用程序指定 URL。该方法是 IWebHostBuilder 接口类型的扩展方法，由于 WebApplicationBuilder 类实现了 IWebHostBuilder 接口，所以能够调用 UseUrls()方法。

UseUrls()方法接受一个或多个字符串类型的参数，每个参数值表示一个有效的 URL。示例代码如下。

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

// 为应用程序设置 3 个 URL
builder.WebHost.UseUrls(
"http://localhost:4252",
"http://localhost:6703",
"http://localhost:5361");

var app = builder.Build();
…
```

运行应用程序后，在浏览器地址栏中输入上述 3 个 URL 中的任意一个，均可以访问，如图 1-3 所示。

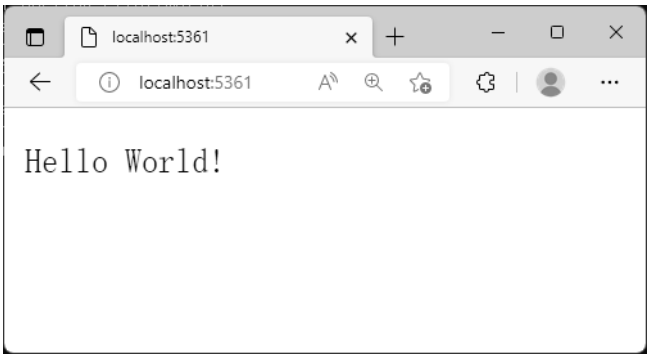

图 1-3　使用指定的 URL 访问 Web 应用

也可以使用 UseSetting()方法配置 URL，如

```
builder.WebHost.UseSetting(WebHostDefaults.ServerUrlsKey,
"http://localhost:12345;http://localhost:54218;http://*:8080");
```

WebHostDefaults 类定义了一组默认的应用配置字段，其中 ServerUrlsKey 表示 URL 的配置键，即 urls。其实，UseUrls()方法内部也是调用了 UseSetting(WebHostDefaults.ServerUrlsKey, …)方法，封装为 UseUrls()方法调用起来更简单。

1.5.2　使用 WebApplication 类的 Urls 属性

Urls 属性是一个字符串类型的集合对象，可以调用 Add()方法添加一个 URL，或者调用 Remove()方法删除一个 URL。

下面的代码通过访问 Urls 属性为应用程序添加两个 URL。

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

app.Urls.Add("http://localhost:5050");
app.Urls.Add("http://localhost:6060");
…
```

Urls 属性是 WebApplication 类的实例成员，允许在调用 WebApplicationBuilder.Build()方法之后进行修改。

1.5.3　调用 Run()方法时传递 URL

在调用 WebApplication 对象的 Run()方法（或 RunAsync()方法）时，可以向 url 参数传递一个有效的 URL，该 URL 将被应用到应用程序上。此处只能设置一个 URL，而且会将应用程序之前配置的所有 URL 全部删除。

请思考以下示例。

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

…

// 配置 3 个 URL
app.Urls.Add("http://localhost:8472");
app.Urls.Add("http://192.168.0.107:13055");
app.Urls.Add("http://127.0.0.1:9288");

app.Run("http://127.0.0.1:16249");
```

上述代码中，先是通过 Urls 属性添加 3 个 URL。接着，在调用 app.Run()方法时设置了第 4 个 URL。

运行示例程序后，只有 http://127.0.0.1:16249 这个地址才能访问应用程序，如图 1-4 所示，Urls 属性中添加的 3 个 URL 被删除。不管前面的代码通过何种方式设置了 URL，最终只有调用 Run()或 RunAsync()方法所指定的 URL 有效。

```
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://127.0.0.1:16249
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
```

图 1-4　只有一个 URL 可用

1.5.4　通过 ServerAddressesFeature 对象设置 URL

ServerAddressesFeature 类（位于 Microsoft.AspNetCore.Hosting.Server.Features 命名空间）是 IServerAddressesFeature 接口的默认实现类。在应用程序代码中一般通过 IServerAddressesFeature 接口来引用。其实，WebApplication 类的 Urls 属性、Run()方法等成员内部也是使用 ServerAddressesFeature 类设置 URL 的，只是做了一层封装而已。

ServerAddressesFeature 类公开一个集合类型的属性——Addresses，在代码中可以调用 Add()方法添加 URL。示例代码如下。

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

// 获取服务实例
IServer server = app.Services.GetRequiredService<IServer>();
// 获取 Feature
IServerAddressesFeature? feat = server.Features
.Get<IServerAddressesFeature>();
if(feat != null)
```

```
    {
        // 设置 URL
        feat.Addresses.Add("http://127.0.0.1:8210");
        feat.Addresses.Add("http://localhost:7466");
        feat.Addresses.Add("http://localhost:23051");
        feat.Addresses.Add("http://localhost:16620");
    }

    app.MapGet("/", () =>"Hello World!");

    app.Run();
```

上述代码先从服务容器中取出实现了 IServer 接口的默认服务器对象(通常是 KestrelServer 类), 再从服务器对象的 Features 集合中查找出实现 IServerAddressesFeature 接口的对象。最后通过 IServerAddressesFeature. Addresses 属性添加所需 URL。

本示例演示的方法比较复杂，建议直接使用封装好的 WebApplication.Urls 属性。

1.5.5 使用命令行参数

前面几个示例使用的 URL 配置方法均属于硬编码方式，其缺点是一旦 URL 需要变动，就要修改代码并重新编译应用程序，在实际开发中不太方便。而通过传递命令行参数配置 URL 的方式则相对灵活，只要在运行应用程序时提供名为 urls 的参数即可。

在调用 WebApplication 类的 CreateBuilder()方法创建 WebApplicationBuilder 实例时，应用程序的配置源中已默认添加对命令参数的支持，因此，开发人员不需要编写额外的处理代码。

下面给出一个简单的示例。

```
    // 在屏幕上打印命令行参数
    Console.WriteLine("命令行参数: {0}", string.Join(", ", args));

    var builder = WebApplication.CreateBuilder(args);
    // 创建应用程序
    var app = builder.Build();
    app.MapGet("/", () =>"你好，世界");
    // 启动 Web 应用程序
    await app.RunAsync();
```

其中，args 是程序入口点——Main 方法的参数，传递给应用程序的命令行参数会保存在 args 参数中。请注意，CreateBuilder()方法有多个重载，若希望接收传递给应用程序的命令行参数，就必须调用带 args 参数的重载，即

```
    public static WebApplicationBuilder CreateBuilder(string[] args);
```

如果调用的是无参数的 CreateBuilder 方法，那么应用程序是不会读取命令行参数的。还可以通过 WebApplicationOptions 类引用命令行参数，它公开了一个名为 Args 的属性，可用于设置命令行参数，如

```
…
var builder = WebApplication.CreateBuilder(new WebApplicationOptions
{
    Args = args
});
// 创建应用程序
var app = builder.Build();
…
```

假设该示例生成的可执行文件为 DemoApp，那么，在执行应用程序时可以通过以下方法设置 urls 参数。

```
DemoApp --urls http://localhost:32085
DemoApp --urls=http://127.0.0.1:14520
DemoApp /urls http://*:15992
DemoApp /urls=http://127.0.0.1:8088
```

其中，http://*:15992 表示侦听本地所有 IP 地址的 15992 端口。如果要指定多个 URL，需要用分号（必须是英文符号）分隔，如

```
DemoApp --urls http://localhost:1788;http://127.0.0.1:48160
```

如果应用程序是以程序集的方式生成（即生成.dll 文件），也可以通过 dotnet 命令运行，如

```
dotnet DemoApp.dll --urls=http://localhost:44225
```

1.5.6　使用配置文件

默认情况下，ASP.NET Core 应用程序会将名为 appsettings.json（包括 appsettings.Development.json 等文件）的配置文件添加到配置源中。因此，通过该文件配置 URL 也非常方便，而且当 URL 需要变动时直接修改配置文件即可。

打开 appsettings.json 文件，加入字段名为 urls 的配置。

```
{
    …
    "urls": "http://*:59148;http://localhost:23804;http://localhost:6731"
}
```

多个 URL 使用分号（英文）分隔。每个 URL 的开头不能有空格，但结尾可以有空格。

例如，下面的配置是无效的，应用程序运行后会发生错误，因为第 2 个 URL（http://localhost:23804）的开始处存在空格。

```
"urls": "http://*:59148; http://localhost:23804;http://localhost:6731"
```

但下面的配置是有效的，因为空格出现在 URL 的末尾，在应用程序中是允许的。

```
"urls": "http://localhost:23804 ;http://localhost:6731"
```

1.5.7　使用环境变量

ASP.NET Core 应用程序项目默认会读取环境变量所提供的配置，而用于设置 URL 的环境

变量名为“<前缀>_URLS”，预设的前缀为 ASPNETCORE，即在运行应用程序前可以通过设置 ASPNETCORE_URLS 环境变量指定 URL。

下面的示例将为应用程序指定两个 URL 地址，然后运行应用程序（假设应用程序名为 DemoApp）。

```
set ASPNETCORE_URLS=http://localhost:7554;http://localhost:19265
dotnet DemoApp.dll
```

1.5.8 使用 launchSettings.json 文件

在应用程序的开发阶段，launchSettings.json 文件仅用于本地测试，通常位于 Properties 目录下。ASP.NET Core 应用项目模板默认会创建此文件，在 Visual Studio 开发环境中，可以在项目属性窗口中以图形化方式编辑 launchSettings.json 文件。

launchSettings.json 文件为 ASP.NET Core 项目配置多个运行环境，在运行应用程序时可以通过配置名称选择需要的环境。

其中，位于配置节点下的 applicationUrl 字段，可以用来配置应用程序的 URL，如

```
"<环境名称>": {
…
"applicationUrl": "http://localhost:6322",
…
      }
```

如果有多个 URL，请使用分号（英文）分隔，如

```
"<环境名称>": {
…
"applicationUrl": "http://localhost:6322;http://localhost:16378",
…
      }
```

1.5.9 Kestrel 服务器的侦听地址

ASP.NET Core 内置的 Kestrel 服务器也支持 URL 配置，不过该配置是基于 Socket 层面的，因此只能使用 IP 地址和端口号确定服务器要侦听的 URL。

KestrelServerOptions 类定义了 3 个公共方法，用来配置 Web 服务器的侦听地址。

（1）Listen()方法：指定要侦听的 IP 地址和端口号。

（2）ListenAnyIP()方法：仅指定服务器端口号，应用程序会侦听绑定到网络接口上的所有 IP 地址，包括 IPv4 地址和 IPv6 地址。该方法优先侦听 IPv6 地址，如果 IPv6 地址不可用，就侦听 IPv4 地址。

（3）ListenLocalhost()方法：指定服务器端口号，仅侦听本机地址（如 127.0.0.1）。

请看下面的示例。

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

// 配置 Kestrel 服务器
builder.WebHost.ConfigureKestrel(options =>
{
    // 设置侦听地址
    options.ListenAnyIP(10072);
});

var app = builder.Build();
…
```

上述代码指定 Kestrel 服务器绑定到任意 IP 地址，侦听端口为 10072。

1.5.10　通过 HTTP.sys 配置 URL

与 Kestrel 一样，HTTP.sys 也是 ASP.NET Core 内部已实现的服务器类型。HTTP.sys 只能运行在 Windows 平台，不支持跨平台。

要在 ASP.NET Core 项目中使用 HTTP.sys 服务器，需要调用 UseHttpSys()方法。也可以结合 HttpSysOptions 类进行相关配置。其中，UrlPrefixes 属性用于设置服务器的 URL 地址。

通过 HTTP.sys 配置 URL 的示例代码如下。

```
var builder = WebApplication.CreateBuilder(args);
// 使用 HTTP.sys
builder.WebHost.UseHttpSys(option =>
{
    // 设置 URL
    option.UrlPrefixes.Add("http://*:12005");
});
…
app.Run();
```

其中，http://*:12005 表示侦听本机上所有地址上传入的请求，并绑定到 12005 端口。

1.5.11　PreferHostingUrls()方法的作用

当 ASP.NET Core 应用程序同时使用 IWebHostBuilder 和 IServer 接口配置 URL 时，通过 IServer 所设置的 URL 会覆盖由 IWebHostBuilder 配置的 URL。

请思考下面的代码。

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

// 配置第 1 个 URL
builder.WebHost.UseUrls("http://localhost:6500");

// 配置第 2 个 URL
builder.WebHost.ConfigureKestrel(option =>
```

```
{
    option.ListenLocalhost(6502);
});

var app = builder.Build();
…
```

第 1 个 URL 是通过 UseUrls()方法配置的，第 2 个 URL 则是通过 Kestrel 选项配置的。由于 Kestrel 选项所配置的 URL 具有更高的优先级，使得应用程序选择了第 2 个 URL。即应用程序会侦听 http://localhost:6502。

调用 PreferHostingUrls()方法会改变此优先级。若将 preferHostingUrls 参数设置为 true，那么通过 IWebHostBuilder 配置的 URL 的优先级会更高，并被应用程序使用；若 preferHostingUrls 参数设置为 false，则 IWebHostBuilder 所配置的 URL 会被 Kestrel 选项所配置的 URL 替代。

在上面的示例代码中加入 PreferHostingUrls()方法的调用。

```
builder.WebHost.PreferHostingUrls(true);

// 配置第 1 个 URL
builder.WebHost.UseUrls("http://localhost:6500");

// 配置第 2 个 URL
builder.WebHost.ConfigureKestrel(option =>
{
    option.ListenLocalhost(6502);
});

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

此时应用程序会优先使用 IWebHostBuilder 接口配置的 URL，即 http://localhost:6500。

1.6 应用程序生命周期事件

ASP.NET Core 应用程序在初始化过程中会向服务容器注册 IHostApplicationLifetime 接口类型的服务。该接口描述了几个与应用程序生命周期相关的事件，具体如下。

（1）ApplicationStarted：只有在应用程序完全启动之后才会触发。

（2）ApplicationStopping：应用程序正在停止运行，但尚未完成操作。

（3）ApplicationStopped：当应用程序正常停止（此过程未发生错误）后触发。

IHostApplicationLifetime 接口的默认实现类是 ApplicationLifetime（位于 Microsoft.Extensions.Hosting.Internal 命名空间）。调用 app.Services.GetService<IHostApplicationLifetime>方法后返回的对象正是 ApplicationLifetime 实例。不过，此处通过 IHostApplicationLifetime 接口访问 ApplicationLifetime 对象的成员即可，并不需要将类型强制转换为 ApplicationLifetime。目前，.NET 平台并不支持自定义实现 IHostApplicationLifetime 接口，因此开发人员无法将自己编写的类型添加到服务容器中（应用程序运行后会发生错误）。

下面的示例将演示应用程序生命周期事件的简单处理过程。

```
// 从服务容器中获取 IHostApplicationLifetime 服务
IHostApplicationLifetime? lifetime = app.Services.GetService
<IHostApplicationLifetime>();
if(lifetime != null)
{
    // 在应用程序启动后触发
    lifetime.ApplicationStarted.Register(() =>
    {
        app.Logger.LogInformation("应用程序已启动");
    });
    // 当应用程序正在停止时触发
    lifetime.ApplicationStopping.Register(() =>
    {
        app.Logger.LogInformation("应用程序正在停止……");
    });
    // 在应用程序完全停止后触发
    lifetime.ApplicationStopped.Register(() =>
    {
        app.Logger.LogInformation("应用程序已停止");
    });
}
```

ApplicationStarted、ApplicationStopping 等属性类型都是 CancellationToken 结构体（位于 System.Threading 命名空间），需要通过 Register()方法注册一个委托实例，才能在与委托绑定的方法中响应事件。本示例只是将文本信息写入日志记录中。在实际开发中，可根据具体情况作出响应。例如，在应用程序启动后创建一些数据文件，当应用程序停止后将这些文件删除。

运行示例程序后，会在控制台界面看到输出的日志信息，如图 1-5 所示。

```
info: DemoApp[0]
      应用程序已启动
info: Microsoft.Hosting.Lifetime[0]
      Application is shutting down...
info: DemoApp[0]
      应用程序正在停止……
info: DemoApp[0]
      应用程序已停止
```

图 1-5　输出应用程序生命周期信息