Startup 类

`Startup` 类位于：

- 已配置应用所需的服务。
- 应用的请求处理管道定义为一系列中间件组件。

下面是 `Startup` 类示例：

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddDbContext<RazorPagesMovieContext>(options =>
options.UseSqlServer(Configuration.GetConnectionString("RazorPagesMovieContext")));

        services.AddControllersWithViews();
        services.AddRazorPages();
    }

    public void Configure(IApplicationBuilder app)
    {
        app.UseHttpsRedirection();
        app.UseStaticFiles();

        app.UseRouting();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapDefaultControllerRoute();
            endpoints.MapRazorPages();
        });
    }
}
```

依赖关系注入（服务）

ASP.NET Core 有内置的依赖关系注入 (DI) 框架，可在应用中提供配置的服务。 例如，日志记录组件就是一项服务。

将配置（或注册）服务的代码添加到 `Startup.ConfigureServices` 方法中。 例如：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<RazorPagesMovieContext>(options =>
options.UseSqlServer(Configuration.GetConnectionString("RazorPagesMovieContext")));

    services.AddControllersWithViews();
    services.AddRazorPages();
}

```

通常使用构造函数注入从 DI 解析服务。 通过构造函数注入，有一个类声明请求的类型或接口的构造函数参数。 DI 框架在运行时提供此服务的实例。

以下示例使用构造函数注入从 DI 解析 `RazorPagesMovieContext`：

```
public class IndexModel : PageModel
{
    private readonly RazorPagesMovieContext _context;

    public IndexModel(RazorPagesMovieContext context)
    {
        _context = context;
    }

    // ...

    public async Task OnGetAsync()
    {
        Movies = await _context.Movies.ToListAsync();
```

```
    }
}

```

如果内置控制反转 (IoC) 容器不能满足应用的所有需求，可以改用第三方 IoC。
有关详细信息，请参阅 ASP.NET Core 依赖注入。

中间件

请求处理管道由一系列中间件组件组成。 每个组件在 `HttpContext` 上执行操作，调用管道中的下一个中间件或终止请求。
按照惯例，通过在 `Startup.Configure` 方法中调用 `Use...` 扩展方法，向管道添加中间件组件。 例如，要启用静态文件的呈现，请调用 `UseStaticFiles`。
以下示例配置了请求处理管道：

```
public void Configure(IApplicationBuilder app)
{
    app.UseHttpsRedirection();
    app.UseStaticFiles();

    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapDefaultControllerRoute();
        endpoints.MapRazorPages();
    });
}

```

ASP.NET Core 包含一组丰富的内置中间件。 也可编写自定义中间件组件。
有关详细信息，请参阅 ASP.NET Core 中间件。

主机

ASP.NET Core 应用在启动时构建主机。 主机封装应用的所有资源，例如：

- HTTP 服务器实现
- 中间件组件
- Logging
- 依赖关系注入 (DI) 服务
- Configuration

有两个不同的主机：

- .NET 通用主机
- ASP.NET Core Web 主机

建议使用 .NET 通用主机。 ASP.NET Core Web 主机仅用于支持后向兼容性。

以下示例将创建 .NET 通用主机：C#复制

```
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>();
            });
}

```

`CreateDefaultBuilder` 和 `ConfigureWebHostDefaults` 方法为主机配置一组默认选项，例如：

- 将 Kestrel 用作 Web 服务器并启用 IIS 集成。
- 从 appsettings.json、appsettings.{Environment Name}.json、环境变量、命令行参数和其他配置源中加载配置 。
- 将日志记录输出发送到控制台并调试提供程序。

有关详细信息，请参阅 ASP.NET Core 中的 .NET 通用主机。

非 Web 方案

其他类型的应用可通过通用主机使用横切框架扩展，例如日志记录、依赖项注入 (DI)、配置和应用生命周期管理。 有关详细信息，请参阅 ASP.NET Core 中的 .NET 通用主机 和 在 ASP.NET Core 中使用托管服务实现后台任务。

服务器

ASP.NET Core 应用使用 HTTP 服务器实现侦听 HTTP 请求。 服务器对应用的请求在表面上呈现为一组由 `HttpContext` 组成的请求功能。

- Windows

ASP.NET Core 提供以下服务器实现：

- Kestrel 是跨平台 Web 服务器。 Kestrel 通常使用 IIS 在反向代理配置中运行。 在 ASP.NET Core 2.0 或更高版本中，Kestrel 可作为面向公众的边缘服务器运行，直接向 Internet 公开。
- IIS HTTP 服务器适用于使用 IIS 的 Windows。 借助此服务器，ASP.NET Core 应用和 IIS 在同一进程中运行。
- HTTP.sys是适用于不与 IIS 一起使用的 Windows 的服务器。

有关详细信息，请参阅 ASP.NET Core 中的 Web 服务器实现。

Configuration

ASP.NET Core 提供了配置框架，可以从配置提供程序的有序集中将设置作为名称/值对。 可将内置配置提供程序用于各种源，例如 .json 文件、.xml 文件、环境变量和命令行参数 。 可编写自定义配置提供程序以支持其他源。

默认情况下，ASP.NET Core 应用配置为从 appsettings.json、环境变量和命令行等读取内容。 加载应用配置后，来自环境变量的值将替代来自 appsettings.json 的值。

读取相关配置值的首选方法是使用选项模式。 有关详细信息，请参阅使用选项模式绑定分层配置数据。

为了管理密码等机密配置数据，.NET Core 提供了机密管理器。 对于生产机密，建议使用 Azure 密钥保管库。

有关详细信息，请参阅 ASP.NET Core 中的配置

环境

执行环境（例如 `Development`、`Staging` 和 `Production`）是 ASP.NET Core 中的高级概念。 通过设置 `ASPNETCORE_ENVIRONMENT` 环境变量来指定应用的运行环境。 ASP.NET Core 在应用启动时读取该环境变量，并将该值存储在 `IWebHostEnvironment` 实现中。 通过依赖关系注入 (DI)，可以在应用中任何位置实现此操作。

使用以下示例配置应用，应用在 `Development` 环境中运行时将提供详细错误信息：

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();

    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapDefaultControllerRoute();
        endpoints.MapRazorPages();
    });
}

```

有关详细信息，请参阅 在 ASP.NET Core 中使用多个环境。

Logging

ASP.NET Core 支持适用于各种内置和第三方日志记录提供程序的日志记录 API。 可用的提供程序包括：

- 控制台
- 调试
- Windows 事件跟踪

- Windows 事件日志
- TraceSource
- Azure 应用服务
- Azure Application Insights

若要创建服务，请从依赖关系注入 (DI) 解析 ILogger<TCategoryName> 服务，并调用 LogInformation 等日志记录方法。 例如：

```
public class TodoController : ControllerBase
{
    private readonly ILogger _logger;

    public TodoController(ILogger<TodoController> logger)
    {
        _logger = logger;
    }

    [HttpGet("{id}", Name = "GetTodo")]
    public ActionResult<TodoItem> GetById(string id)
    {
        _logger.LogInformation(LoggingEvents.GetItem, "Getting item {Id}", id);

        // Item lookup code removed.

        if (item == null)
        {
            _logger.LogWarning(LoggingEvents.GetItemNotFound, "GetById({Id}) NOT FOUND", id);
            return NotFound();
        }

        return item;
    }
}

```

`LogInformation` 等日志记录方法支持任意数量的字段。 这些字段通常用于构造消息 `string`，但某些日志记录提供程序会将它们作为独立字段发送到数据存储。 此功能使日志提供程序可以实现语义日志记

录，也称为结构化日志记录。
有关详细信息，请参阅 .NET Core 和 ASP.NET Core 中的日志记录。

路由

路由是映射到处理程序的 URL 模式。 处理程序通常是 Razor 页面、MVC 控制器中的操作方法或中间件。 借助 ASP.NET Core 路由，可以控制应用使用的 URL。
有关详细信息，请参阅 ASP.NET Core 中的路由。

错误处理

ASP.NET Core 具有用于处理错误的内置功能，例如：

- 开发人员异常页
- 自定义错误页
- 静态状态代码页
- 启动异常处理

有关详细信息，请参阅 处理 ASP.NET Core 中的错误。

发出 HTTP 请求

`IHttpClientFactory` 的实现可用于创建 `HttpClient` 实例。 工厂可以：

- 提供一个中心位置，用于命名和配置逻辑 HttpClient 实例。 例如，注册并配置 github 客户端以访问 GitHub。 注册并配置默认客户端以实现其他目的。
- 支持多个委托处理程序的注册和链接，以生成出站请求中间件管道。 此模式类似于 ASP.NET Core 的入站中间件管道。 此模式提供了一种用于管理 HTTP 请求相关问题的机制，包括缓存、错误处理、序列化以及日志记录。
- 与 Polly 集成，这是用于瞬时故障处理的常用第三方库。
- 管理基础 HttpClientHandler 实例的池和生存期，避免手动管理 HttpClient 生存期时可能出现的常见 DNS 问题。通过 ILogger 添加可配置的日志记录体验，用于记录通过工厂创建的客户端发送的所有请求。

有关详细信息，请参阅 在 ASP.NET Core 中使用 IHttpClientFactory 发出 HTTP 请求。

内容根

内容根目录是指向以下内容的基路径：

- 托管应用的可执行文件 (.exe)。
- 构成应用程序的已编译程序集 (.dll)。
- 应用使用的内容文件，例如：
  - Razor 文件（.cshtml、.razor）
  - 配置文件（.json、.xml）
  - 数据文件 (.db)
- Web 根目录
- ，通常是 wwwroot 文件夹。

在开发中，内容根目录默认为项目的根目录。 此目录还是应用内容文件和 Web 根目录的基路径。 在构建主机时设置路径，可指定不同的内容根目录。 有关详细信息，请参阅内容根。

Web 根

Web 根目录是公用静态资源文件的基路径，例如：

- 样式表 (.css)
- JavaScript (.js)
- 图像（.png、.jpg）

默认情况下，静态文件仅从 Web 根目录及其子目录提供。 Web 根目录路径默认为 **{content root}/wwwroot**。 在构建主机时设置路径，可指定不同的 Web 根目录。 有关详细信息，请参阅 Web 根目录。

防止使用项目文件中的 <Content> 项目项在 wwwroot 中发布文件。 下面的示例会阻止在 wwwroot/local 及其子目录中发布内容：XML复制

```
<ItemGroup>
  <Content Update="wwwroot\local\**\*.*" CopyToPublishDirectory="Never" />
</ItemGroup>

```

在 Razor .cshtml 文件中，波形符-斜线 (~/) 指向 Web 根。 以 ~/ 开头的路径称为虚拟路径。

有关详细信息，请参阅 ASP.NET Core 中的静态文件。