第 1 章

概　　览

欢迎来到第1章，本章将介绍ASP.NET Core及其安装和设置开发环境，让读者对这个强大的Web开发框架有一个全面的了解。

1.1　ASP.NET Core 简介

ASP.NET Core是一个强大而灵活的Web应用程序开发框架，它为开发人员提供了丰富的功能和工具，帮助他们构建高性能、可扩展的Web应用程序。ASP.NET Core具有许多令人激动的特性和优势，让我们一起来看看吧。

1.1.1　为什么选择 ASP.NET Core

首先，ASP.NET Core是跨平台的，这意味着可以在不同的操作系统上运行应用程序，如Windows、macOS和Linux。这为我们提供了更大的灵活性和可扩展性，使得应用程序可以在多个平台上运行而无须重写代码。这是一个非常重要的特性，尤其是在现代的多平台世界中。

其次，ASP.NET Core是开源的，这意味着我们可以访问它的源代码并参与到框架的开发和改进中。开源社区为ASP.NET Core提供了丰富的资源和支持，我们可以从中获得宝贵的知识和经验，并与其他开发人员进行交流和合作。这使得ASP.NET Core成为一个活跃且充满活力的开发生态系统。

1.1.2　ASP.NET Core 的核心特性

ASP.NET Core具有许多令人印象深刻的核心特性，下面是其中一些重要的特性：

（1）高性能：ASP.NET Core经过优化，具有出色的性能表现。它采用了一些先进的技术和策略，如异步编程模型和内存管理，来提供高吞吐量和低延迟的响应。

（2）可扩展性：ASP.NET Core提供了强大的可扩展性，使得应用程序能够应对不断增长的用户需求。我们可以轻松地扩展应用程序，以适应高流量和大规模的用户访问。

（3）依赖关系注入：ASP.NET Core内置了依赖关系注入（Dependency Injection）容器，帮助我们管理和解耦应用程序中的各个组件。依赖关系注入可以提高代码的可测试性和可维护性，使得开发和维护应用程序变得更加轻松。

（4）轻量级和模块化：ASP.NET Core的设计理念是轻量级和模块化，它避免了不必要的复杂性和冗余。我们可以根据需求选择和使用需要的组件，以构建精简且高效的应用程序。

（5）安全性：ASP.NET Core提供了许多内置的安全功能，帮助我们保护应用程序免受常见的安全威胁。它支持身份验证、授权、数据保护等关键安全功能，使得我们可以构建安全可靠的应用程序。

这些仅仅是ASP.NET Core的一些核心特性，它还有许多其他强大的功能等待我们去探索和应用。无论是构建简单的网站还是复杂的企业级应用程序，ASP.NET Core都能满足需求。

1.1.3 ASP.NET Core 的架构

ASP.NET Core的架构采用了模块化的设计，由许多不同的组件和模块组成。核心组件包括：

（1）HTTP服务器：ASP.NET Core可以使用各种HTTP服务器来处理HTTP请求和响应。常用的HTTP服务器包括Kestrel、IIS（Internet Information Services）等。

（2）Web主机：Web主机是一个宿主环境，用于启动和运行ASP.NET Core应用程序。它负责处理应用程序的生命周期、配置和管理。

（3）中间件：中间件是ASP.NET Core的一个重要概念，它是请求处理管道中的一个组件。中间件可以对请求和响应进行处理，并将它们传递给下一个中间件或终端处理程序。

（4）路由：路由是决定如何将请求映射到处理程序的机制。ASP.NET Core提供了灵活而强大的路由系统，使得我们可以自定义路由规则。

（5）控制器和视图：控制器是处理用户请求的核心组件，它接收请求并生成相应的响应。视图负责呈现用户界面，并与控制器进行交互。

这些组件共同工作，使得ASP.NET Core能够处理请求、生成响应，并构建出完整的Web应用程序。

1.1.4 ASP.NET Core 的应用场景

ASP.NET Core适用于各种应用场景。无论是构建简单的静态网站还是复杂的企业级应用程序，ASP.NET Core都能胜任。

以下是一些适合使用ASP.NET Core的应用场景的示例：

（1）Web应用程序：ASP.NET Core可以用于构建各种类型的Web应用程序，如电子商务网站、社交媒体平台、新闻门户等。

（2）API服务：ASP.NET Core提供了强大的Web API功能，可以用于构建RESTful API服务，供其他应用程序和移动应用程序使用。

（3）实时通信：ASP.NET Core具有实时通信的功能，可以用于构建聊天应用程序、实时协作工具等。

（4）微服务：ASP.NET Core适合构建微服务架构，将应用程序拆分为小而自治的服务单元，以提高可扩展性和灵活性。

（5）云原生应用程序：ASP.NET Core可以在云环境中部署和运行，如Microsoft Azure、Amazon Web Services等。

这只是ASP.NET Core的一些应用场景示例，读者可以根据自己的需求和项目要求，选择合适的方式来应用ASP.NET Core。

1.2　ASP.NET Core 的演变历程

ASP.NET Core是一个不断演变和发展的Web应用程序开发框架。它的演变历程是一个精心设计和不断改进的过程，让我们一起来了解一下吧！

1.2.1　早期的 ASP.NET 框架

要了解ASP.NET Core的演变历程，首先要回顾一下早期的ASP.NET框架。ASP.NET框架是从ASP（Active Server Pages）技术发展而来的，它在Web开发领域起到了一定的革命性作用。

ASP.NET框架的出现使得Web开发更加简单和高效。它引入了Web Forms，这是一种基于事件模型的编程方式，使得开发人员可以通过拖放控件和编写事件处理程序来构建Web应用程序。Web Forms提供了一种类似于Windows Forms的开发体验，使得开发人员可以更轻松地构建交互性强、功能丰富的Web应用程序。

然而，随着时间的推移，ASP.NET框架也暴露出一些不足之处，其中一些问题包括复杂的页面生命周期、大量的视图状态数据、对前端技术的限制等。此外，ASP.NET框架在某种程度上与特定的操作系统和Web服务器绑定，限制了它在不同平台上的应用。

1.2.2　ASP.NET Core 的诞生

为了应对上述问题和新的需求，微软决定重新设计和重构ASP.NET，于是ASP.NET Core诞生了。ASP.NET Core的目标是提供一个现代化、跨平台和高性能的Web应用程序开发框架。

ASP.NET Core在设计上采用了模块化和轻量级的原则，摒弃了一些旧有的概念和依赖。它经过精心设计，以满足当今Web开发的需求。下面是ASP.NET Core的一些重要特点和改进：

（1）跨平台支持：ASP.NET Core可以在多个操作系统上运行，包括Windows、macOS和Linux。这使得开发人员可以在不同的平台上使用相同的代码和技术栈开发应用程序。

（2）高性能：ASP.NET Core经过优化，具有出色的性能表现。它采用了异步编程模型和新的处理管道，以提供高吞吐量和低延迟的响应。

（3）模块化和灵活性：ASP.NET Core采用了模块化的设计，我们可以选择性地使用和配置需要的组件和功能。这样可以使应用程序变得更轻巧、更高效，并且可以更好地满足特定需求。

（4）开源和社区支持：ASP.NET Core是开源的，拥有一个活跃的开源社区。开源社区提供了丰富的资源、工具和支持，使得ASP.NET Core变得更强大和可靠。

（5）支持新的前端技术：ASP.NET Core对前端技术提供了更好的支持，如支持使用现代的JavaScript框架、前端构建工具和RESTful API等。

通过不断改进和演变，ASP.NET Core成为一个先进、灵活且功能丰富的Web应用程序开发框架。它提供了强大的工具和功能，使得开发人员能够构建出高性能、可扩展的Web应用程序，并在不同的平台上运行。

1.2.3 迁移到 ASP.NET Core

如果读者目前正在使用早期版本的ASP.NET框架，或者使用其他的Web开发框架，那么可以考虑迁移到ASP.NET Core。迁移到ASP.NET Core有许多好处，包括跨平台支持、更好的性能、更灵活的开发体验等。

在迁移到ASP.NET Core之前，需要评估应用程序和相关依赖，了解迁移的复杂性和可能的挑战。微软提供了详细的迁移指南和工具，可以帮助我们顺利地将应用程序迁移到ASP.NET Core。

1.3 安装和设置开发环境

在开始使用ASP.NET Core之前，需要进行一些安装和设置工作，这样才能确保我们能够顺利地进行ASP.NET Core应用程序的开发和调试。本节就让我们一起来了解如何安装和设置开发环境吧！

1.3.1 安装.NET Core SDK

首先，需要安装.NET Core SDK（Software Development Kit）。SDK是用于开发和运行ASP.NET Core应用程序所需的工具和资源。我们可以按照以下步骤来安装.NET Core SDK：

步骤 01 访问.NET 官方网站（https://dotnet.microsoft.com/download），如图 1-1 所示。

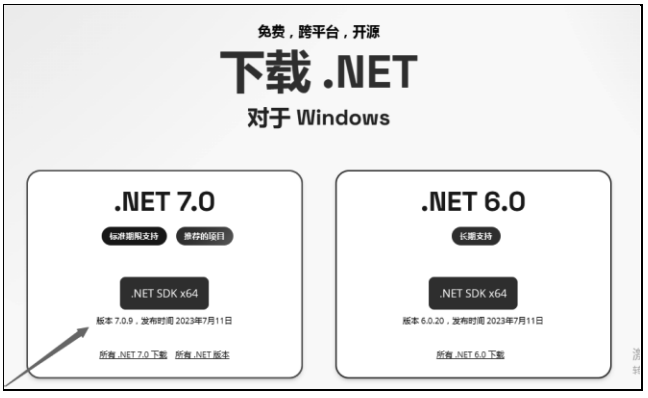

图 1-1 下载.NET 页面

步骤 02 选择适合我们操作系统的.NET Core SDK 版本，通常可以选择最新的稳定版本，如图 1-2 所示。

图 1-2　选择版本页面

步骤 03 下载并运行安装程序，按照安装程序的指示完成安装过程，如图 1-3 所示。

图 1-3　安装程序

步骤 04 安装完成后，在命令行中运行“dotnet –version”命令，以验证.NET Core SDK 是否成功安装并可用，如图 1-4 所示。

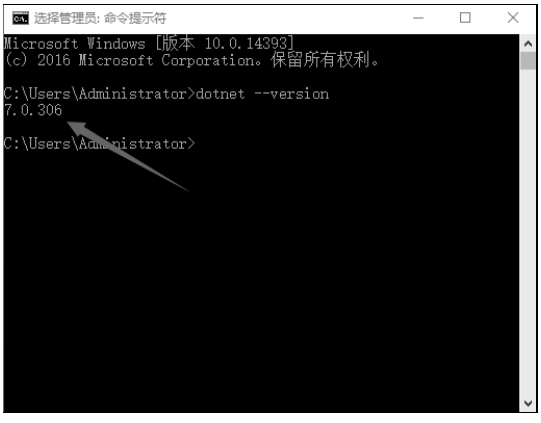

图 1-4　检查.NET Core SDK 是否可用

1.3.2 选择开发工具

安装了.NET Core SDK后，我们可以选择合适的开发工具来编写ASP.NET Core应用程序。以下是一些常用的开发工具选项：

1. Visual Studio

Visual Studio是一个强大的集成开发环境（IDE），提供了丰富的功能和工具，适用于ASP.NET Core开发。我们可以下载并安装Visual Studio Community Edition（免费版本）或Professional/Enterprise Edition。

2. Visual Studio Code

Visual Studio Code是一个轻量级的文本编辑器，也是一个功能强大的开发工具。它支持ASP.NET Core开发，并提供了许多有用的扩展和插件。

3. 命令行工具

如果读者喜欢使用命令行进行开发，那么可以使用.NET Core CLI（命令行界面）。CLI提供了许多命令，用于创建、构建和运行ASP.NET Core应用程序。

选择合适的开发工具取决于个人偏好和项目需求。无论选择哪种工具，都可以顺利地进行ASP.NET Core开发。

1.3.3 创建 ASP.NET Core 项目

安装了开发工具后，可以开始创建ASP.NET Core项目了。一些常见的创建一个新的ASP.NET Core项目的方法如下：

1. 使用 Visual Studio

打开Visual Studio，选择创建新项目（New Project）选项。在项目模板中选择ASP.NET Core Web应用程序，然后按照向导的指示进行项目设置和配置。

2. 使用 Visual Studio Code

在Visual Studio Code中，可以使用命令面板（Ctrl+Shift+P）或终端（Terminal）来创建ASP.NET Core项目。首先使用dotnet new命令创建一个新的项目模板，然后使用dotnet run命令来运行应用程序。

3. 使用命令行工具

如果使用命令行工具进行开发，可以使用.NET Core CLI来创建和运行ASP.NET Core项目。首先使用dotnet new命令创建一个新的项目模板，然后使用dotnet run命令来运行应用程序。

创建项目时，我们可以选择不同的项目模板，如Web API、MVC、Razor Pages等。选择适合需求的项目模板，并根据需要进行配置和设置。

1.3.4　运行和调试应用程序

创建了ASP.NET Core项目后，就可以运行和调试应用程序了。以下是一些常用的运行和调试方式：

1. Visual Studio

在Visual Studio中，可以通过单击“开始调试”（Start Debugging）按钮或按F5键来运行和调试应用程序。还可以设置断点、监视变量，并使用调试工具来查找和修复代码中的问题。

2. Visual Studio Code

在Visual Studio Code中，可以使用调试面板（Debug Panel）来配置和运行调试会话。还可以设置断点、启动调试会话，并使用调试工具来检查应用程序的运行情况。

3. 命令行工具

如果使用命令行工具进行开发，那么可以使用.NET Core CLI提供的命令来构建和运行应用程序。使用dotnet build命令构建项目，使用dotnet run命令运行应用程序。

通过运行和调试应用程序，我们可以验证代码是否正常工作，并进行必要的修复和改进。

1.4　小　　结

在本章中，我们对ASP.NET Core进行了简单的介绍。

首先介绍了ASP.NET Core是一个强大而灵活的Web应用程序开发框架，它具有跨平台、高性能、开源等特点，适用于各种应用场景。接着，介绍了ASP.NET Core的演变历程，ASP.NET Core是从早期的ASP.NET框架发展而来的，经过重新设计和重构，以满足当今Web开发的需求。然后，介绍了安装和设置ASP.NET Core开发环境的步骤。我们安装了.NET Core SDK，选择了合适的开发工具，并创建了一个新的ASP.NET Core项目。最后，介绍了如何运行和调试应用程序，以验证代码的正确性。

现在，我们已经为学习ASP.NET Core打下了坚实的基础。接下来的章节将深入探讨ASP.NET Core的各个方面，帮助读者成为一个熟练的ASP.NET Core开发人员。

第 2 章

基 础 知 识

本章将深入介绍ASP.NET Core的基础知识，包括ASP.NET Core的核心概念和关键组件，为构建强大和可扩展的应用程序打下坚实的基础。

2.1 Razor Pages 介绍

在本节中，我们将介绍ASP.NET Core中的Razor Pages。Razor Pages是一种用于构建Web应用程序的编程模型，它可以帮助我们快速开发具有丰富交互和动态内容的网页。通过使用Razor Pages，我们可以轻松地将数据和逻辑代码与视图组合在一起，以创建功能强大的Web应用程序。

2.1.1 什么是 Razor Pages

Razor Pages是ASP.NET Core中的一种新的页面编程模型，它与传统的MVC（模型-视图-控制器）模式相比，更加简单直观。Razor Pages旨在提供一种易于学习和使用的方式来构建Web应用程序。

在Razor Pages中，每个页面都有一个对应的.cshtml文件，其中包含了HTML标记和C#代码。这使得在一个文件中组织视图和相关的代码变得非常容易和直观。

Razor Pages基于Razor语法，这是一种结合了HTML和C#的强大模板引擎。Razor语法允许我们在HTML中嵌入C#代码，并且可以根据需要动态生成HTML内容。

2.1.2 Razor Pages 和 MVC

在介绍Razor Pages之前，让我们先来了解一下传统的MVC模式。

MVC是一种应用广泛的Web开发模式，它将应用程序分为模型（Model）、视图（View）和控制器（Controller）3个部分。模型负责处理数据，视图负责呈现界面，控制器负责处理用

户请求和协调模型与视图之间的交互。

与MVC相比，Razor Pages更加简洁。在Razor Pages中，每个页面都是一个独立的单元，包含了视图和相关的代码，不需要显式定义控制器。这种紧密集成的设计使得开发更加高效，并且适用于构建小到中型规模的应用程序。

在某些情况下，MVC仍然是一个更适合的选择，特别是当我们需要更复杂的路由和控制器之间的细粒度控制时。但对于大多数应用程序来说，Razor Pages提供了一个更简单、更直观的开发模式。

2.1.3　创建一个 Razor Page

下面通过一个简单的示例来展示如何创建一个Razor Page。具体步骤如下：

步骤 01　创建一个名为“Index.cshtml”的文件，并将代码清单 2-1 中代码复制到该文件中。

代码清单 2-1

```
@page
@model IndexModel

<h1>Welcome to Razor Pages</h1>

<p>Current time: @DateTime.Now</p>

<p>Message from model: @Model.Message</p>
```

在代码清单2-1中，我们使用@page指令指定这个文件是一个Razor Page。然后使用@model指令指定了与该页面相关联的模型类。接着，我们可以在页面中使用HTML标记和Razor语法来构建页面的内容。这里展示了一个标题和一些简单的文本内容。注意，我们还使用了@DateTime.Now来显示当前时间，使用了@Model.Message来显示来自模型的消息。

步骤 02　接下来，创建一个名为 Index.cshtml.cs 的文件，并将代码清单 2-2 中的代码复制到该文件。

代码清单 2-2

```
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace lesson02 1.Pages
{
    public class IndexModel : PageModel
    {
        public string? Message { get; set; }

        public void OnGet()
        {
            Message = "Hello, Razor Pages!";
        }

    }
}
```

在代码清单2-2中，我们创建了一个继承自PageModel的模型类IndexModel，在该类中定义了一个公共属性Message，并在OnGet方法中设置了消息的值。

步骤03 现在，运行应用程序并在浏览器中导航到/Index 路径，我们将看到 Razor Page 的内容，包括当前时间和来自模型的消息，如图 2-1 所示。

图 2-1 呈现的 Razor Pages 内容

2.2 MVC 介绍

在本节中，我们将介绍MVC模式。MVC是一种应用广泛的Web开发模式，它帮助我们更好地组织和管理Web应用程序的结构和逻辑。通过了解MVC的原理和组成部分，我们将能够构建更加可维护、可扩展和可测试的应用程序。

2.2.1 什么是 MVC 模式

MVC模式是一种软件设计模式，用于将应用程序分解为3个主要组件：模型（Model）、视图（View）和控制器（Controller）。每个组件都有自己的职责和功能，通过彼此之间的协作来实现完整的应用程序。

1. 模型

模型负责处理应用程序的数据和业务逻辑。它是应用程序的核心部分，用于表示和管理数据、执行数据操作、处理业务规则等。模型通常是与数据库、文件系统或外部服务进行交互的接口。

2. 视图

视图负责呈现用户界面和展示数据。它是应用程序的外观部分，负责将模型中的数据可视化并向用户显示。视图通常是HTML、CSS和JavaScript等前端技术的组合，用于创建交互性和可视化效果。

3. 控制器

控制器负责处理用户请求和协调模型与视图之间的交互。它接收用户输入，根据请求调

用相应的模型逻辑，并决定将哪个视图呈现给用户。控制器还可以处理验证、身份验证和其他与请求处理相关的任务。

通过将应用程序划分为这3个组件，MVC模式提供了一种结构化的方式来组织和管理应用程序的逻辑。这种分离使得每个组件都可以独立开发、测试和维护，从而提高了代码的可重用性和可维护性。

2.2.2 MVC 模式的工作流程

下面让我们来看一下MVC模式的工作流程，以便更好地理解各个组件之间的交互。

（1）用户发起请求：用户在浏览器中输入URL或与应用程序交互，触发一个请求。

（2）控制器接收请求：控制器接收请求并根据请求的类型和内容决定下一步的操作。

（3）控制器调用模型：控制器根据请求的内容调用适当的模型方法，进行数据处理、业务逻辑操作等。

（4）模型处理数据：模型接收控制器传递的数据，并执行相应的操作，例如从数据库中检索数据、更新数据等。

（5）控制器获取结果：控制器接收模型处理后的结果，并根据需要进行后续处理。

（6）控制器选择视图：控制器选择要呈现给用户的视图，并将模型的数据传递给视图。

（7）视图渲染内容：视图接收控制器传递的数据，并使用合适的模板引擎将数据和静态内容组合成最终的HTML页面。

（8）响应返回给用户：生成的HTML页面作为响应返回给用户的浏览器，用户在浏览器中看到呈现的页面。

通过这种方式，MVC模式将应用程序的逻辑分离为不同的组件，每个组件都有自己的职责，使得代码更具可读性、可维护性和可测试性。

2.2.3 在 ASP.NET Core 中使用 MVC

ASP.NET Core框架提供了内置的支持来构建基于MVC模式的Web应用程序。下面是一个简单的示例，展示如何在ASP.NET Core中使用MVC。

步骤01 确保已经安装了 ASP.NET Core 开发环境，并创建一个新的 ASP.NET Core Web 应用程序项目。

步骤02 在项目中创建一个控制器类，例如 HomeController，如代码清单 2-3 所示。

代码清单 2-3

```
using Microsoft.AspNetCore.Mvc;

namespace lesson02_2.Controllers
{
    public class HomeController : Controller
    {

        public IActionResult Index()
```

```
        {
            return View();
        }
    }
}
```

在代码清单2-3中，创建了一个名为“HomeController”的控制器类，它继承自Controller基类。在控制器中，我们定义了一个名为“Index”的动作方法，并在方法中返回一个视图。

步骤03 创建一个与控制器方法对应的视图文件，例如 Index.cshtml，如代码清单 2-4 所示。

代码清单 2-4

```
<h1>Welcome to MVC</h1>
<p>This is the home page of our application.</p>
```

在代码清单2-4中，创建了一个简单的视图，用于显示欢迎消息和应用程序的首页内容。

步骤04 配置路由，以便将请求映射到控制器和动作方法。在 Program.cs 文件的 Main 方法中添加代码清单 2-5 中的代码。

代码清单 2-5

```
app.UseRouting();
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

在代码清单2-5中，使用了默认的路由模板"{controller=Home}/{action=Index}/{id?}"，它将请求映射到名为“Home”的控制器的Index方法中。

步骤05 运行应用程序并在浏览器中导航到“/”路径，我们将看到 MVC 模式下的视图内容，如图 2-2 所示。

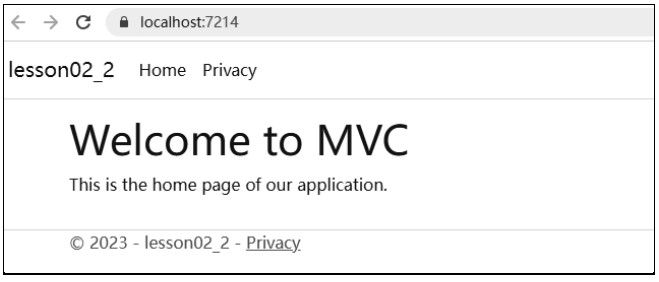

图 2-2　MVC 模式下的视图内容

2.3　Web API 介绍

在本节中，我们将介绍Web API的概念和用途。Web API是一种用于构建和提供Web服务的技术，它可以帮助我们构建灵活、可扩展和跨平台的应用程序。通过了解Web API的原理和

用法，我们将能够创建强大的API，并与其他应用程序进行数据交换和集成。

2.3.1 什么是 Web API

Web API是一种用于构建和提供Web服务的技术。它允许应用程序通过HTTP协议与外部世界进行通信，并传递和接收数据。Web API可以返回各种格式的数据，例如JSON、XML等，使得数据交换和集成变得更加灵活和可扩展。

Web API通常用于构建面向移动应用程序、前端应用程序、第三方开发者等的后端服务。它提供了一种可靠和标准的方式来暴露应用程序的功能和数据，使其他应用程序能够与之进行交互。

2.3.2 RESTful API

在Web API的世界中，一种常见的设计风格是REST（Representational State Transfer，表现层状态转换）。RESTful API是基于REST原则设计的API，它使用统一的URL结构和HTTP方法来表示资源和执行操作。

RESTful API的设计原则包括：

（1）资源导向：将应用程序的功能和数据表示为资源，每个资源都有一个唯一的URL来标识。

（2）无状态：每个请求都应该包含足够的信息来处理该请求，不依赖于之前的请求或状态。

（3）使用HTTP方法：使用HTTP协议提供的方法（如GET、POST、PUT、DELETE）来表示对资源的操作。

（4）使用HTTP状态码：使用合适的HTTP状态码来表示请求的结果和状态。

通过遵循RESTful API的设计原则，可以使API更加清晰、易于理解和可扩展。

2.3.3 创建一个 Web API

下面让我们通过一个简单的示例来展示如何在ASP.NET Core中创建一个Web API。

步骤01 创建一个新的 ASP.NET Core Web 应用程序项目，并选择 Web API 模板。

步骤02 在项目中创建一个控制器类，例如 ProductsController，如代码清单 2-6 所示。

代码清单 2-6

```
using Microsoft.AspNetCore.Mvc;

namespace lesson02_3.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    public class ProductsController : ControllerBase
    {
        private static List<string> _products = new List<string>
```

```
        {
            "Product 1",
            "Product 2",
            "Product 3"
        };

        [HttpGet]
        public ActionResult<IEnumerable<string>> Get()
        {
            return _products;
        }
    }
}
```

在代码清单2-6中，创建了一个名为“ProductsController”的控制器类，并使用[ApiController]和 [Route("api/[controller]")]属性进行了标记。在控制器中定义了一个GET方法，用于处理HTTP GET请求，并返回产品列表。

步骤03 运行应用程序，并通过浏览器或 API 测试工具访问“/api/products”路径。我们将获得一个包含产品列表的 JSON 响应，如代码清单 2-7 所示。

代码清单 2-7

```
[
  "Product 1",
  "Product 2",
  "Product 3"
]
```

2.3.4 使用其他 HTTP 方法

除了GET方法之外，Web API还支持其他常见的HTTP方法，例如POST、PUT和DELETE，用于创建、更新和删除资源。

下面让我们在ProductsController中添加一些额外的方法，如代码清单2-8所示。

代码清单 2-8

```
[HttpPost]
public IActionResult Post(string product)
{
    _products.Add(product);
    return Ok();
}

[HttpPut("{index}")]
public IActionResult Put(int index, string product)
{
    if (index >= 0 && index < _products.Count)
    {
        _products[index] = product;
```

```
            return Ok();
        }
        else
        {
            return NotFound();
        }
    }

    [HttpDelete("{index}")]
    public IActionResult Delete(int index)
    {
        if (index >= 0 && index < _products.Count)
        {
            _products.RemoveAt(index);
            return Ok();
        }
        else
        {
            return NotFound();
        }
    }
```

在代码清单2-8中，添加了一个POST方法用于创建新的产品，一个PUT方法用于更新现有的产品，以及一个DELETE方法用于删除产品。

现在，我们可以使用相应的HTTP方法和适当的URL来测试这些API方法，并观察它们的行为。

2.4 应用启动

在ASP.NET Core中，应用启动是构建Web应用程序的重要部分。它涉及配置应用程序的服务、中间件和其他必要的组件，以及定义应用程序的请求处理管道。本节将介绍如何正确启动ASP.NET Core应用程序。

2.4.1 配置应用程序的启动类

在ASP.NET Core中，可以使用一个特殊的启动类来配置应用程序。这个启动类通常命名为“Startup”，并包含一个名为“ConfigureServices”和一个名为“Configure”的方法。

1. ConfigureServices 方法

在ConfigureServices方法中，可以配置应用程序的服务。服务是应用程序中的可重用组件，可以通过依赖关系注入系统在整个应用程序中进行访问。例如，我们可以在ConfigureServices方法里注册数据库上下文、存储库和其他服务。

下面是一个ConfigureServices方法的示例，如代码清单2-9所示。

代码清单 2-9

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

在代码清单2-9中，我们使用AddDbContext方法注册了一个数据库上下文，并配置它使用SQL Server作为数据库提供程序。另外，我们还使用AddControllers方法注册了Controller服务。

2. Configure 方法

在Configure方法中，可以配置应用程序的请求处理管道。请求处理管道由一系列中间件组成，用于处理传入的HTTP请求并生成响应。我们可以通过添加、删除或重新排序中间件来满足应用程序的需求。

下面是一个Configure方法的示例，如代码清单2-10所示。

代码清单 2-10

```
    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
           if (env.IsDevelopment())
           {
               app.UseSwagger();
               app.UseSwaggerUI();
           }

           app.UseHttpsRedirection();
           app.UseRouting();
           app.UseAuthorization();

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

在这个示例中，我们根据应用程序的环境配置使用了不同的中间件。在开发环境中，开发人员异常页中间件在默认情况下处于启用状态。在请求处理管道中，我们还使用了其他一些常用的中间件，如UseHttpsRedirection（将HTTP请求重定向到HTTPS）、UseRouting（启用路由）等。

2.4.2 启动应用程序

要启动ASP.NET Core应用程序，需要在Main方法中调用CreateHostBuilder方法，并指定Startup类。

下面是一个Main方法的示例，如代码清单2-11所示。

代码清单 2-11

```
using codelist0209_0211;

var builder = WebApplication.CreateBuilder(args);

var startup = new Startup(builder.Configuration);
startup.ConfigureServices(builder.Services);

var app = builder.Build();

startup.Configure(app, app.Environment);

app.Run();
```

在这个示例中，我们使用CreateBuilder方法创建一个默认的WebApplicationBuilder实例，并通过builder.Build()生成用于配置HTTP管道和路由的Web应用程序，通过app.Run()运行应用程序并阻止调用线程，直至主机关闭。在new Startup(builder.Configuration)中，我们指定了Startup类来配置应用程序。

一旦应用程序启动，它将开始监听传入的HTTP请求，并根据Startup类中定义的请求处理管道进行处理。

2.5 依赖关系注入

在本节中，我们将介绍依赖关系注入的概念和作用。依赖关系注入是一种设计模式，用于解耦组件之间的依赖关系，提高代码的可测试性、可维护性和可扩展性。通过了解DI的原理和用法，我们将能够更好地管理应用程序中的对象和依赖关系。

2.5.1 什么是依赖关系注入

依赖关系注入是一种设计模式，用于在应用程序中管理对象之间的依赖关系。它通过将对象的创建和依赖项的提供从对象本身解耦出来，使得对象可以专注于自身的功能而不需要关注如何创建或获取依赖项。

在DI中，对象不再自己创建或获取所需的依赖项，而是通过外部机制将依赖项注入对象中。这种外部机制通常由DI容器负责，它会自动解析和提供所需的依赖项。

DI的好处包括：

（1）松耦合：对象不再直接依赖于特定的实现细节，而是依赖于抽象接口。这使得对象之间的耦合度降低，更容易进行代码重构和更换依赖项。

（2）可测试性：通过将依赖项注入对象中，我们可以轻松地模拟和替换依赖项，从而使单元测试变得更容易和可靠。

（3）可维护性：DI使得代码的结构更清晰和可读，因为依赖项的创建和传递逻辑被提取到DI容器中，使得代码更易于理解和维护。

2.5.2 在 ASP.NET Core 中使用依赖关系注入

ASP.NET Core框架内置了强大的依赖关系注入容器，使得在应用程序中使用DI变得非常简单。

下面通过一个简单的示例来演示如何在ASP.NET Core中使用DI。

步骤01 假设我们有一个服务接口 IMyService 和它的实现 MyService，如代码清单 2-12 所示。

代码清单 2-12

```
public interface IMyService
{
    void DoSomething();
}

public class MyService : IMyService
{
    public void DoSomething()
    {
        Console.WriteLine("Doing something...");
    }
}
```

步骤02 现在，我们希望在控制器中使用这个服务。我们可以通过 DI 将 IMyService 注入控制器中，如代码清单 2-13 所示。

代码清单 2-13

```
public class MyController : Controller
{
    private readonly IMyService _myService;

    public MyController(IMyService myService)
    {
        _myService = myService;
    }

[HttpGet]
    public void Index()
    {
        _myService.DoSomething();
    }
}
```

在代码清单2-13中，我们在控制器的构造函数中接收了一个IMyService参数，并将它赋值给私有字段_myService。这样，就可以在控制器的其他方法中使用_myService来调用服务的方法了。

ASP.NET Core的DI容器会自动解析控制器的依赖关系，并在需要时创建和提供IMyService的实例。

步骤 03 在 Startup.cs 文件的 ConfigureServices 方法中，需要注册服务和其对应的实现，如代码清单 2-14 所示。

代码清单 2-14

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddScoped<IMyService, MyService>();
    // 其他服务的注册
}
```

在代码清单2-14中，我们使用AddScoped方法注册了IMyService和MyService之间的依赖关系。这意味着DI容器将在每个HTTP请求范围内创建一个MyService的实例，并将它提供到需要的地方。

现在，当我们访问MyController的Index方法时，DI容器会自动创建MyController的实例，并自动注入IMyService的实例。我们可以在Index方法中调用服务的方法，而无须手动创建或获取服务的实例。

2.5.3　生命周期管理

ASP.NET Core的DI容器提供了多种生命周期选项，以控制对象的生命周期和作用域。以下是一些常见的生命周期选项：

- Singleton：在整个应用程序生命周期内只创建一个实例。
- Scoped：在每个HTTP请求范围内创建一个实例。
- Transient：每次请求或获取时都创建一个新的实例。

通过选择适当的生命周期选项，我们可以更好地管理对象的生命周期和资源的使用，提高应用程序的性能和资源利用率。

在注册服务时，可以使用相应的方法来指定生命周期，例如AddSingleton、AddScoped和AddTransient，如代码清单2-15所示。

代码清单 2-15

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSingleton<IMySingletonService, MySingletonService>();
    services.AddScoped<IMyScopedService, MyScopedService>();
    services.AddTransient<IMyTransientService, MyTransientService>();
    // 其他服务的注册
}
```

在代码清单2-15中，使用了不同的生命周期选项来注册不同类型的服务。这样，当我们注入这些服务时，DI容器将根据相应的生命周期选项来创建和提供服务的实例。

2.6 中 间 件

本节将介绍中间件的概念和作用。中间件是ASP.NET Core中的一个关键概念，它允许我们在请求处理管道中插入自定义的组件，实现各种功能，例如路由、日志记录、异常处理等。通过了解中间件的原理和用法，我们将能够更好地控制和定制应用程序的请求处理流程。

2.6.1 什么是中间件

中间件是ASP.NET Core请求处理管道中的组件，用于处理HTTP请求和生成HTTP响应。中间件位于请求管道的特定位置，每个中间件都有机会处理请求或传递请求给下一个中间件。

中间件可以执行各种任务，例如身份验证、授权、路由、异常处理、日志记录等。通过使用中间件，我们可以轻松地构建复杂的请求处理流程，并实现不同的功能。

2.6.2 中间件的工作原理

下面介绍一下中间件的工作原理，以便更好地理解它在请求处理管道中的作用。

ASP.NET Core的请求处理管道由一系列中间件组成，并按照特定的顺序依次执行。当一个请求到达应用程序时，它首先通过管道的起始点，然后按照中间件的顺序依次经过每个中间件。中间件的执行顺序非常重要，因为它决定了请求的处理流程。

以下是一个简化的示例，展示了中间件的执行顺序，如代码清单2-16所示。

代码清单 2-16

```
请求 --> 中间件1 --> 中间件2 --> ... --> 中间件N --> 响应
```

在代码清单2-16中，请求首先经过中间件1，然后传递给中间件2，以此类推，直到达到最后一个中间件。每个中间件都可以根据需要对请求进行处理或修改，并将请求传递给下一个中间件。

2.6.3 创建自定义中间件

在ASP.NET Core中，我们可以创建自定义的中间件来满足特定的需求。

下面以一个简单的日志记录中间件为例，演示如何创建自定义中间件。

步骤 01 创建一个新的类来实现中间件逻辑，如代码清单 2-17 所示。

代码清单 2-17

```
using Microsoft.AspNetCore.Http;
using System;
using System.Threading.Tasks;

public class LoggingMiddleware
{
```

```
    private readonly RequestDelegate _next;

    public LoggingMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public async Task InvokeAsync(HttpContext context)
    {
        // 在请求处理之前执行的逻辑
        Console.WriteLine($"Request: {context.Request.Path}");

        await _next(context);

        // 在请求处理之后执行的逻辑
        Console.WriteLine($"Response: {context.Response.StatusCode}");
    }
}
```

在代码清单2-17中，创建了一个名为“LoggingMiddleware”的类，并实现了一个InvokeAsync方法作为中间件的入口点。在InvokeAsync方法中，我们可以在请求处理之前和之后执行自定义的逻辑。这里我们简单地记录了请求的路径和响应的状态码。

步骤 02　接下来，在 Startup.cs 文件的 Configure 方法中使用自定义中间件，如代码清单 2-18 所示。

代码清单 2-18

```
public void Configure(IApplicationBuilder app)
{
    app.UseMiddleware<LoggingMiddleware>();

    // 其他中间件的配置
}
```

在代码清单2-18中，使用UseMiddleware方法将LoggingMiddleware添加到请求处理管道中。因此，当应用程序收到请求时，LoggingMiddleware将记录请求的路径和响应的状态码。

2.6.4　内置中间件

ASP.NET Core还提供了许多内置的中间件，用于常见的任务和功能。以下是一些常用的内置中间件：

- UseDeveloperExceptionPage：开发人员异常页中间件，用于报告应用运行时错误。
- UseDatabaseErrorPage：数据库错误页中间件，用于报告数据库运行时错误。
- UseExceptionHandler：异常处理程序中间件，用于捕获以下中间件中引发的异常。
- UseHsts：HTTP 严格传输安全协议中间件，用于添加Strict-Transport-Security标头。
- UseHttpsRedirection：HTTPS重定向中间件，用于将HTTP请求重定向到HTTPS。
- UseStaticFiles：静态文件中间件，用于提供静态文件（如CSS、JavaScript、图像等）。
- UseCookiePolicy：Cookie策略中间件。

- UseRouting：用于路由请求的路由中间件。
- UseAuthentication：身份验证中间件，用于尝试对用户进行身份验证，然后才会允许用户访问安全资源。
- UseAuthorization：用于授权用户访问安全资源的授权中间件。
- UseSession：会话中间件，用于建立和维护会话状态。如果应用使用会话状态，则在Cookie策略中间件之后和 MVC 中间件之前调用会话中间件。

通过使用这些内置中间件，我们可以轻松地添加常见的功能到应用程序的请求处理管道中。

2.7 Web 主机

本节将介绍Web主机的概念和作用。Web主机是ASP.NET Core应用程序的宿主环境，负责启动应用程序、处理HTTP请求和管理应用程序的生命周期。通过了解Web主机的原理和用法，我们将能够更好地理解和管理ASP.NET Core应用程序。

2.7.1 什么是 Web 主机

Web主机是ASP.NET Core应用程序的宿主环境。它负责启动应用程序、处理HTTP请求和管理应用程序的生命周期。Web主机提供了必要的基础设施来运行ASP.NET Core应用程序，并与HTTP服务器进行通信。

创建和配置 Web 主机

下面通过一个简单的示例来演示如何创建和配置通用主机。

步骤 01 创建一个新的 ASP.NET Core 应用程序项目，并选择通用主机模板。

步骤 02 在 Program.cs 文件中找到 CreateDefaultBuilder 方法，该方法用于创建和配置 Web 主机，如代码清单 2-19 所示。

代码清单 2-19

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

在代码清单2-19中，使用CreateDefaultBuilder方法创建了一个默认的主机构建器，并使用ConfigureWebHostDefaults方法配置了Web主机。

在webBuilder.UseStartup<Startup>()中，指定了一个名为“Startup”的类作为应用程序的启动类。Startup类用于配置应用程序的服务和中间件。

2.7.2　配置 Web 主机选项

Web主机可以通过主机构建器的选项进行配置，以下是一些常见的选项：

- UseUrls：指定应用程序监听的URL。
- UseEnvironment：指定应用程序的环境名称（如Development、Production等）。
- UseConfiguration：使用自定义的配置。
- UseContentRoot：指定应用程序的内容根目录。

我们可以根据应用程序的需求选择适当的选项，并在CreateHostBuilder方法中进行配置。

2.7.3　运行 Web 主机

要运行Web主机，我们需要在Main方法中调用Run方法，如代码清单2-20所示。

代码清单 2-20

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

在代码清单2-20中，首先通过CreateHostBuilder方法创建主机，并使用Build方法构建主机。然后调用Run方法来启动主机并开始监听HTTP请求。

现在，我们可以运行应用程序并访问配置的URL，这样就可以与应用程序进行交互了。

2.8　HTTP 服务器

本节将介绍HTTP服务器的概念和作用。HTTP服务器是负责处理和响应HTTP请求的软件程序。在ASP.NET Core中，HTTP服务器是Web应用程序和客户端之间的桥梁，它负责接收和处理HTTP请求，并返回相应的HTTP响应。通过了解HTTP服务器的原理和用法，我们将能够更好地理解和控制ASP.NET Core应用程序与客户端之间的通信。

2.8.1　什么是 HTTP 服务器

HTTP服务器是负责处理和响应HTTP请求的软件程序。它充当Web应用程序和客户端之间的桥梁，负责接收和处理客户端发送的HTTP请求，并返回相应的HTTP响应。

在ASP.NET Core中，可以使用各种HTTP服务器来承载和运行应用程序，例如Kestrel、IIS和Apache等。不同的服务器可能具有不同的特性和配置选项，但它们都遵循HTTP协议，以实现与客户端之间的通信。

2.8.2 Kestrel HTTP 服务器

Kestrel是ASP.NET Core的默认HTTP服务器，它是一个跨平台的、轻量级的服务器。Kestrel使用libuv作为其底层网络库，具有高性能和可扩展性。

Kestrel可以独立运行，也可以与其他HTTP服务器（如IIS）配合使用。它可以通过配置文件或代码进行配置，并支持SSL/TLS加密、HTTP/2协议、反向代理等功能。

以下是一个简单的示例，展示如何在ASP.NET Core应用程序中使用Kestrel作为HTTP服务器，如代码清单2-21所示。

代码清单 2-21

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

在代码清单2-21中，我们使用UseKestrel方法配置了Kestrel作为HTTP服务器。然后，使用UseStartup方法指定了一个名为Startup的类作为应用程序的启动类。

通过这样的配置，我们可以在应用程序中使用Kestrel作为默认的HTTP服务器。

2.8.3 其他 HTTP 服务器

除了Kestrel之外，ASP.NET Core还支持其他常见的HTTP服务器，如HTTP.sys、IIS、Apache等。这些服务器通常与特定的操作系统或托管环境相关联，并具有各自的配置和使用方式。

要使用其他HTTP服务器，需要进行适当的配置，并将应用程序部署到相应的服务器上。

2.9 配　　置

本节将介绍配置的概念和作用。配置是ASP.NET Core中用于管理应用程序配置的强大机制。它提供了一种统一的方式来读取和使用应用程序的配置数据，使得配置信息可以轻松地进行管理和修改。通过了解配置的原理和用法，我们将能够更好地配置和定制ASP.NET Core应用程序。

2.9.1 什么是配置

配置是ASP.NET Core中的一个重要组件，用于管理应用程序的配置信息。它提供了一种统一的方式来读取和使用配置数据，包括应用程序的参数、连接字符串、密钥、标志等。

配置支持多种配置源，如JSON文件、环境变量、命令行参数等。通过使用配置，我们可以轻松地读取配置数据，并在应用程序中使用这些数据。

2.9.2　配置源

配置支持多种配置源，我们可以根据应用程序的需求来选择适当的配置源。以下是一些常见的配置源：

- JSON文件：JSON文件是一种常见的配置源，它使用简单的键值对结构来存储配置数据。
- 环境变量：环境变量是操作系统中的全局变量，可以用来存储配置信息。通过读取环境变量，我们可以将配置数据传递给应用程序。
- 命令行参数：命令行参数是在应用程序启动时传递的参数，可以用来覆盖默认的配置值。
- 密钥管理器：密钥管理器是用于存储和管理敏感数据（如密码、密钥等）的工具。配置可以与密钥管理器集成，以安全地存储和读取敏感配置数据。

我们可以根据应用程序的需求组合使用不同的配置源，并将其配置为应用程序的配置提供者。

2.9.3　读取配置数据

配置提供了简单而强大的方式来读取配置数据。通过注入IConfiguration接口，我们可以轻松地读取和使用配置数据。

以下是一个简单的示例，展示如何读取配置数据，如代码清单2-22所示。

代码清单 2-22

```
public class MyService
{
    private readonly IConfiguration  configuration;

    public MyService(IConfiguration configuration)
    {
        configuration = configuration;
    }

    public void DoSomething()
    {
        var settingValue = _configuration["SettingKey"];
        Console.WriteLine($"Setting value: {settingValue}");
    }
}
```

在代码清单2-22中，我们在MyService类的构造函数中注入了IConfiguration接口，然后就可以使用_configuration对象来读取配置数据了。

通过使用配置键（例如SettingKey），我们可以访问相应的配置值。

2.9.4　配置文件

配置支持使用配置文件来存储和组织配置数据。常见的配置文件格式包括JSON、XML、

INI等。

以下是一个JSON配置文件的示例，如代码清单2-23所示。

代码清单 2-23

```
{
  "SettingKey": "Hello",
  "Logging": {
    "LogLevel": "Information"
  }
}
```

在应用程序中，我们可以使用appsettings.json文件作为默认的配置文件，并通过配置构建器进行加载和使用，如代码清单2-24所示。

代码清单 2-24

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureAppConfiguration((hostingContext, config) =>
        {
            config.SetBasePath(Directory.GetCurrentDirectory());
            config.AddJsonFile("appsettings.json", optional: true);
        })
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
        });
```

在代码清单2-24中，我们使用ConfigureAppConfiguration方法来配置构建器，并加载appsettings.json文件。

通过这样的配置，我们可以在应用程序中使用IConfiguration接口来读取和使用配置数据。

2.10 选 项 模 式

选项模式（Options Pattern）是ASP.NET Core中的一种用于配置和选项管理的模式。通过选项模式，我们可以将应用程序的设置和配置信息进行统一管理，使其更易于维护和扩展。

2.10.1 为什么需要选项模式

在开发ASP.NET Core应用程序时，通常需要配置一些参数和选项，例如数据库连接字符串、日志级别、缓存设置等。在过去，我们可能会使用配置文件、环境变量或者直接在代码中硬编码这些参数。然而，这种做法存在一些问题：

（1）硬编码的参数不易维护：将配置信息硬编码在代码中，会导致在修改参数时需要修改代码，并重新编译应用程序，增加了维护成本。

（2）不便于配置文件管理：虽然可以使用配置文件来管理参数，但是手动解析和读取配置文件的代码逻辑往往相对烦琐，而且不够直观。

（3）难以进行动态配置：有时候我们希望能够在应用程序运行时动态修改参数，而硬编码和配置文件方式都不太适合实现动态配置。

为了解决这些问题，ASP.NET Core 提供了选项模式，它提供了一种统一的方式来管理应用程序的配置和选项。

2.10.2　如何使用选项模式

使用选项模式的第一步是定义选项类（Options Class）。选项类是一个普通的C#类，用于存储应用程序的配置信息。以下是一个定义选项类的示例，如代码清单2-25所示。

代码清单 2-25

```
public class MyConfigOptions
{
    public string Key1 { get; set; }
    public int Key2 { get; set; }
    // 其他配置项
}
```

在选项类中，可以定义需要的各种配置项属性，例如示例中的数据库连接字符串和日志级别。

接下来，需要在应用程序的启动代码中注册选项，并将配置信息绑定到选项类上。这可以通过调用services.Configure<TOptions>(configuration)方法来实现，其中TOptions是定义的选项类类型，configuration是应用程序的配置对象，如代码清单2-26所示。

代码清单 2-26

```
public void ConfigureServices(IServiceCollection services)
{
    // ...

    services. AddOptions<MyConfigOptions>()
         .Bind(Configuration.GetSection("MyConfig"));
    // ...
}
```

在代码清单2-26中，我们将MyConfigOptions类型的选项注册到服务容器中，并将配置对象Configuration绑定到该选项类上。

最后，在需要使用配置信息的地方通过依赖注入来获取选项对象，如代码清单2-27所示。

代码清单 2-27

```
public class HomeController : ControllerBase
{
    private readonly MyConfigOptions  myConfig;
```

```
    public HomeController(IOptions<MyConfigOptions> myConfig)
    {
        _myConfig = myConfig.Value;
    }

    [HttpGet]
    public void Index()
    {
        Console.WriteLine("Key1:" + _myConfig.Key1);
        Console.WriteLine("Key2:" + _myConfig.Key2);
    }

}
```

在上述示例中，我们通过构造函数注入IOptions<MyConfigOptions>对象，并在需要使用配置信息的地方通过_myConfig.Value获取选项对象，然后就可以直接访问选项对象的属性，如示例中的数据库连接字符串和日志级别。

2.10.3 选项验证和默认值

在使用选项模式时，还可以进行选项验证和设置默认值。例如，我们可以在选项类中添加验证逻辑，以确保配置信息的有效性，并应用若干DataAnnotations规则，如代码清单2-28所示。

代码清单 2-28

```
public class MyConfigOptions
{
    [RegularExpression(@"^[a-zA-Z''-'\s]{1,40}$")]
    public string Key1 { get; set; }

    [Range(0, 100, ErrorMessage = "Value for {0} must be between {1} and {2}.")]
    public int Key2 { get; set; }
}
```

接下来，通过调用ValidateDataAnnotations以使用DataAnnotations启用验证，如代码清单2-29所示。

代码清单 2-29

```
public void ConfigureServices(IServiceCollection services)
{
    // ...

    services. AddOptions<MyConfigOptions>()
         .Bind(Configuration.GetSection("MyConfig"))
         .ValidateDataAnnotations();

    // ...
}
```

在代码清单2-29中，我们使用MyConfigOptions类来验证选项的有效性。

另外，还可以在选项类中设置默认值，以防止配置文件中没有指定相应的参数，如代码清单2-30所示。

代码清单 2-30

```
public class MyConfigOptions
{
    public string Key1 { get; set; } = "hello";
    public int Key2 { get; set; } = 30;
}
```

在代码清单2-30中，我们为Key1和Key2设置了默认值，如果配置文件中没有指定相应的参数，将会使用这些默认值。

2.11　执 行 环 境

本节将介绍执行环境的概念和作用。执行环境是ASP.NET Core中用于确定应用程序运行的上下文信息的机制。它提供了访问和使用与应用程序部署和执行环境相关的信息的方式。通过了解执行环境的原理和用法，我们将能够更好地配置和适应ASP.NET Core应用程序。

2.11.1　什么是执行环境

执行环境是ASP.NET Core中的一个重要概念，用于确定应用程序当前运行的上下文信息。执行环境提供了访问与应用程序部署和执行环境相关的信息的方式，例如应用程序的环境名称、操作系统、主机、配置等。

执行环境可以帮助我们根据当前的上下文信息进行不同的配置和适应。例如，在开发环境下可以启用详细的日志记录和调试信息，而在生产环境下可以启用性能优化和错误处理。

2.11.2　执行环境的类型

在ASP.NET Core中，有两种常见的执行环境类型：

- 开发环境：开发环境是在开发和调试应用程序时使用的环境。它通常具有更详细的日志记录，更灵活的错误处理和调试功能。
- 生产环境：生产环境是应用程序部署和运行的环境。它通常具有更高的性能，更严格的错误处理和安全性。

除了这两种常见的执行环境类型，还可以根据应用程序的需求自定义其他的执行环境。

2.11.3　访问执行环境信息

在ASP.NET Core中，可以通过注入IWebHostEnvironment接口来访问执行环境的信息。

以下是一个简单的示例，展示如何使用IWebHostEnvironment接口，如代码清单2-31所示。

代码清单 2-31

```
public class MyService
{
    private readonly IWebHostEnvironment  environment;

    public MyService(IWebHostEnvironment environment)
    {
        environment = environment;
    }

    public void DoSomething()
    {
        var environmentName = _environment.EnvironmentName;
        Console.WriteLine($"Current environment: {environmentName}");
    }
}
```

在这个示例中，我们在MyService类的构造函数中注入了IWebHostEnvironment接口，然后使用_environment对象来访问执行环境的信息。通过EnvironmentName属性，我们可以获取当前的环境名称（如Development、Production等）。

2.11.4 配置执行环境

ASP.NET Core应用程序的执行环境是通过设置应用程序的环境变量来确定的。可以在不同的部署环境中设置不同的环境变量值，以指定应用程序应使用的执行环境。

以下是一些常用的设置执行环境的方式：

（1）通过启动配置文件：可以在启动配置文件中设置ASPNETCORE_ENVIRONMENT环境变量的值。

（2）通过操作系统环境变量：可以在部署环境的操作系统中设置ASPNETCORE_ENVIRONMENT环境变量的值。

（3）通过命令行参数：可以在应用程序启动时通过命令行参数（如--environment）来设置执行环境。

根据应用程序的需求，选择适当的方式来配置执行环境，并确保应用程序在不同的环境中正确地适应和运行。

2.12 日 志 记 录

本节将介绍日志记录的概念和作用。日志记录是在应用程序中记录和存储运行时信息的机制。它对于应用程序的调试、故障排除和性能分析非常重要。通过了解日志记录的原理和用

法，我们将能够更好地管理和分析ASP.NET Core应用程序。

2.12.1　为什么需要日志记录

日志记录是应用程序开发和维护过程中至关重要的一部分，它提供了以下几个重要的好处：

- 故障排除：日志记录可以帮助我们追踪和定位应用程序中的错误和异常。当应用程序发生故障时，我们可以查看日志以了解问题产生的根本原因。
- 性能分析：通过记录关键操作和事件的性能指标，我们可以使用日志来分析和优化应用程序的性能。日志记录可以帮助我们识别潜在的性能瓶颈和热点。
- 安全审计：日志记录可以记录应用程序的关键操作和访问事件，以进行安全审计和合规性检查。它可以帮助我们跟踪和审计敏感数据的访问和修改。

2.12.2　ASP.NET Core 的日志记录

在ASP.NET Core中，日志记录是通过内置的日志记录器（Logger）来实现的。日志记录器是用于记录和存储应用程序运行时信息的组件。

ASP.NET Core提供了一个统一的接口（ILogger）来使用日志记录器。通过注入ILogger接口，我们可以在应用程序中记录各种类型的日志消息。

以下是一个简单的示例，展示如何在ASP.NET Core应用程序中使用日志记录，如代码清单2-32所示。

代码清单 2-32

```
public class MyService
{
    private readonly ILogger<MyService> _logger;

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

    public void DoSomething()
    {
        _logger.LogInformation("Doing something...");

        try
        {
            // 执行操作
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "An error occurred while doing something.");
        }
    }
```

```
    }
```

在代码清单2-32中，我们在MyService类的构造函数中注入了ILogger<MyService>接口，然后使用_logger对象来记录不同级别的日志消息。

通过使用不同的日志级别（如LogInformation和LogError），我们可以记录不同类型的日志消息，并在应用程序中使用它们。

2.12.3 配置日志记录

在ASP.NET Core中，可以通过配置文件或代码来配置日志记录。配置日志记录可以包括设置日志级别、选择日志输出目标（如控制台、文件、数据库等）、格式化日志消息等。

以下是一个示例，展示如何在应用程序的配置文件中配置日志记录，如代码清单2-33所示。

代码清单 2-33

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "MyNamespace.MyClass": "Debug"
    },
    "Console": {
      "Enabled": true
    }
  }
}
```

在代码清单2-33中，我们在配置文件中定义了不同的日志记录设置。我们可以为不同的命名空间、类或默认值设置不同的日志级别，并选择不同的输出目标。

通过配置日志记录，我们可以根据应用程序的需求和环境进行灵活的日志记录设置。

2.12.4 日志记录最佳实践

在进行日志记录时，以下是一些值得注意的最佳实践：

- 选择适当的日志级别：选择适当的日志级别，以平衡详细的日志记录和性能需求。
- 提供有意义的日志消息：确保日志消息清晰、有意义，能够帮助识别问题。
- 结构化日志记录：使用结构化日志记录格式，以便更轻松地查询和分析日志数据。
- 安全敏感信息：避免在日志中记录敏感信息，如密码、密钥等。

2.13 路　　由

本节将介绍路由的概念和作用。路由是ASP.NET Core中用于映射HTTP请求到相应处理程

序的机制。通过了解路由的原理和用法，我们将能够更好地理解和控制应用程序的URL结构和请求处理。

2.13.1 什么是路由

路由是将传入的HTTP请求映射到相应的处理程序或动作的过程。在ASP.NET Core中，路由是负责解析URL路径和参数，并将请求分发到相应的处理程序或控制器的机制。

通过路由，我们可以定义应用程序的URL结构，并确定由哪个处理程序来处理特定的HTTP请求。

2.13.2 路由模板

在ASP.NET Core中，路由是通过路由模板（Route Template）来定义的。路由模板是一种特殊的语法，用于指定URL路径和参数的模式。

以下是一个简单的示例，展示如何定义一个基本的路由模板，如代码清单2-34所示。

代码清单 2-34

```
app.MapGet("/hello", async context =>
{
    await context.Response.WriteAsync("Hello, World!");
});
```

在代码清单2-34中，我们使用MapGet方法将“/hello”路径映射到一个处理程序。当客户端发送一个GET请求到“/hello”时，将执行该处理程序，并返回“Hello, World!”作为响应。

通过路由模板，我们可以定义不同类型的路由，包括静态路由、参数化路由、区域路由等。

2.13.3 路由参数

路由参数允许我们从URL路径中提取变量值，并将其传递给处理程序。通过路由参数，我们可以根据URL的不同部分来动态地处理请求。

以下示例展示如何在路由模板中使用参数，如代码清单2-35所示。

代码清单 2-35

```
app.MapGet("/users/{id}", async context =>
{
    var id = context.Request.RouteValues["id"];
    await context.Response.WriteAsync($"User ID: {id}");
});
```

在代码清单2-35中，我们使用{id}作为路由模板的一部分。当客户端发送一个GET请求到类似“/users/123”的URL时，我们可以从路由值中提取ID，并在响应中使用它。

通过路由参数，我们可以构建具有动态URL结构的应用程序。

2.13.4 路由约束

路由约束允许我们对路由参数的格式和取值进行限制。通过使用路由约束，可以确保参数满足特定的要求，从而更好地控制请求的处理。

以下示例展示如何使用路由约束，如代码清单2-36所示。

代码清单 2-36

```
app.MapGet("/users/{id:int}", async context =>
{
    var id = context.Request.RouteValues["id"];
    await context.Response.WriteAsync($"User ID: {id}");
});
```

在代码清单2-36中，我们使用“:int”作为路由模板的一部分，并指定了参数的类型约束为整数。这样，只有满足整数格式的参数才会被匹配并处理。

通过路由约束，我们可以确保参数满足特定的格式要求，从而提高应用程序的安全性和稳定性。

2.13.5 路由属性

ASP.NET Core还提供了一种更简洁的方式来定义路由，即使用路由属性（Route Attribute）。通过在控制器或处理程序的类或方法上应用路由属性，我们可以直接指定与之关联的路由模板。

以下示例展示如何在控制器类和方法上使用路由属性，如代码清单2-37所示。

代码清单 2-37

```
[Route("api/[controller]")]
[ApiController]
public class UsersController : ControllerBase
{
    [HttpGet("{id}")]
    public void GetUser(int id)
    {
        Console.WriteLine(id);
    }
}
```

在代码清单2-37中，我们使用[Route]属性在控制器类上指定了路由模板，表示该控制器的所有动作都将使用以“/api/[controller]”开头的URL。同时，在HttpGet方法上使用[HttpGet("{id}")]属性指定了路由参数。

通过路由属性，我们可以更直观地定义路由，并将它与相关的控制器和动作关联起来。

2.14　错 误 处 理

本节将介绍错误处理的概念和作用。错误处理是应对应用程序中出现的错误和异常的机制。通过了解错误处理的原理和用法，我们将能够更好地管理和响应应用程序中的错误情况。

2.14.1　为什么需要错误处理

错误处理是应用程序开发和维护过程中至关重要的一部分，它可以帮助我们识别、捕获和处理应用程序中出现的错误和异常情况。

错误处理的好处如下：

- 提供友好的错误信息：错误处理可以帮助我们向用户提供有意义和友好的错误信息，从而增强用户体验。
- 避免应用程序崩溃：通过捕获和处理错误，可以避免应用程序因错误而崩溃或不可用。
- 追踪和记录错误：错误处理可以帮助我们追踪和记录应用程序中出现的错误，从而帮助排除故障和分析问题。

2.14.2　全局错误处理

在ASP.NET Core中，可以使用全局错误处理机制来统一处理应用程序中的错误和异常。全局错误处理允许我们定义一个中间件来捕获并处理应用程序中发生的所有错误。

以下示例展示如何配置全局错误处理中间件，如代码清单2-38所示。

代码清单 2-38

```
    app.UseExceptionHandler("/error");

    app.MapGet("/error", async context =>
    {
        var exceptionHandlerPathFeature =
context.Features.Get<IExceptionHandlerPathFeature>();
        var exception = exceptionHandlerPathFeature?.Error;
        // 处理错误，生成自定义的错误响应
    });
```

在代码清单2-38中，我们使用UseExceptionHandler方法来配置全局错误处理中间件，并指定错误处理的路径为“/error”。当应用程序发生错误时，将执行中间件中指定的处理逻辑。

在处理逻辑中，我们可以获取异常信息，并根据需要生成自定义的错误响应。

通过全局错误处理，我们可以集中处理应用程序中的错误，并提供一致的错误处理机制。

2.14.3　异常筛选器

除了全局错误处理之外，ASP.NET Core还提供了异常筛选器（Exception Filters）的机制，

用于在特定条件下处理错误。

异常筛选器可以应用于控制器或动作方法，以捕获和处理特定类型的异常。

以下示例展示如何使用异常筛选器，如代码清单2-39所示。

代码清单 2-39

```
public class CustomExceptionFilter : IExceptionFilter
{
    public void OnException(ExceptionContext context)
    {
        // 处理特定类型的异常
    }
}

[ServiceFilter(typeof(CustomExceptionFilter))]
public class MyController : Controller
{
    // 控制器逻辑
}
```

在代码清单2-39中，我们定义了一个实现IExceptionFilter接口的异常筛选器，然后在控制器上使用[ServiceFilter]属性来应用该异常筛选器。

当控制器中的动作方法发生特定类型的异常时，异常筛选器的OnException方法将被执行，我们可以在该方法中处理异常情况。

通过异常筛选器，我们可以对特定类型的异常进行处理，并根据需要采取相应的措施。

2.14.4 状态码和错误页面

在错误处理过程中，状态码和错误页面起着重要的作用。状态码是HTTP响应的一部分，用于表示请求的处理结果。错误页面是向用户展示有关错误的页面，以提供更好的用户体验。

在ASP.NET Core中，我们可以通过配置来定义状态码和错误页面的行为。

以下示例展示如何在应用程序中定义状态码和错误页面，如代码清单2-40所示。

代码清单 2-40

```
app.UseStatusCodePagesWithRedirects("/error/{0}");

app.UseExceptionHandler("/error");

app.MapGet("/error", async context =>
{
    var code = context.Request.RouteValues["code"];
        // 根据状态码生成自定义的错误页面
});
```

在代码清单2-40中，我们使用UseStatusCodePagesWithRedirects方法来配置状态码处理中间件，并指定错误页面的路径模板。当应用程序返回指定状态码时，中间件将重定向到指定路径模板，从而显示自定义的错误页面。

通过配置状态码和错误页面，可以提供更好的用户体验，并向用户显示有关错误的相关信息。

2.15　静 态 文 件

本节将介绍如何在ASP.NET Core应用程序中处理和提供静态文件。静态文件通常是应用程序中的样式表、脚本文件、图像和其他静态资源。通过有效地处理静态文件，可以提高应用程序的性能和加载速度。

2.15.1　配置静态文件中间件

在ASP.NET Core中，我们使用静态文件中间件来处理和提供静态文件。要在应用程序中启用静态文件中间件，需要在Startup类的Configure方法中进行配置。

下面示例演示如何在Configure方法中启用静态文件中间件，如代码清单2-41所示。

代码清单 2-41

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

    // 其他中间件配置
}
```

在代码清单2-41中，我们使用了UseStaticFiles方法来启用静态文件中间件。这将允许我们在应用程序中访问静态文件，例如wwwroot文件夹中的文件。

2.15.2　创建静态文件

要在应用程序中使用静态文件，需要将它们放置在wwwroot文件夹中。wwwroot文件夹是默认的静态文件根目录，可以通过在ConfigureServices方法中调用UseWebRoot方法来更改该目录。

以下是一个示例目录结构，展示wwwroot文件夹中的常见静态文件，如代码清单2-42所示。

代码清单 2-42

```
wwwroot/
├── css/
│   └── styles.css
├── js/
│   └── script.js
└── images/
    └── logo.png
```

在上面的示例中，有一个css文件夹，包含名为“styles.css”的CSS文件；一个js文件夹，

包含名为“script.js”的JavaScript文件；一个images文件夹，包含名为“logo.png”的图像文件。

2.15.3 访问静态文件

启用静态文件中间件后，我们可以通过URL来访问静态文件。默认情况下，静态文件中间件会处理URL中的路径，并根据文件系统中的对应文件提供相应的静态文件。

例如，如果有一个styles.css文件位于wwwroot/css目录下，那么可以通过以下URL来访问它，如代码清单2-43所示。

代码清单 2-43

```
http://localhost:5000/css/styles.css
```

类似地，如果有一个logo.png图像位于wwwroot/images目录下，那么可以通过以下URL来访问它，如代码清单2-44所示。

代码清单 2-44

```
http://localhost:5000/css/styles.css
```

静态文件中间件还提供了一些其他功能，例如默认文档处理、目录浏览和响应缓存等，可以通过在UseStaticFiles方法中传递StaticFileOptions来配置这些功能。

2.16 小　　结

本章深入介绍了ASP.NET Core的基础知识。

首先探讨了Razor Pages、MVC和Web API的概念，了解到这些是构建ASP.NET Core应用程序的不同方式，可以根据需求选择适合的架构。接着介绍了应用启动的重要性，了解了默认的应用启动方式和如何自定义启动行为。然后介绍了依赖关系注入的概念和用法，以及中间件的作用和配置方式。

最后深入研究了Web主机、HTTP服务器和配置，了解了如何管理应用程序的执行环境和记录日志，还学习了路由和错误处理的重要性，以及如何处理静态文件。

本章介绍的概念和技术将为读者构建功能强大、可扩展和可靠的Web应用程序奠定基础。