.NET 开发经典名著

# ASP.NET Core 开发实战

[意] 迪诺·埃斯波西托(Dino Esposito) 著
赵利通 译

清华大学出版社
北 京

北京市版权局著作权合同登记号 图字：01-2019-3037



图书在版编目(CIP)数据

ASP.NET Core 开发实战 / (意) 迪诺 • 埃斯波西托(Dino Esposito) 著，赵利通 译. —北京：清华大学出版社，2019

(.NET 开发经典名著)

书名原文：Programming ASP.NET Core

ISBN 978-7-302-52887-6

Ⅰ. ①A… Ⅱ. ①迪… ②赵… Ⅲ. ①网页制作工具－程序设计 Ⅳ. ①TP393.092.2

中国版本图书馆 CIP 数据核字(2019)第 083667 号

责任编辑：王　军
装帧设计：思创景点
责任校对：牛艳敏
责任印制：丛怀宇

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　邮　　编：100084
社 总 机：010-62770175　　邮　　购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者：三河市金元印装有限公司
经　　销：全国新华书店
开　　本：170mm×240mm　　印　　张：24.5　　字　　数：508 千字
版　　次：2019 年 7 月第 1 版　　印　　次：2019 年 7 月第 1 次印刷
定　　价：79.80 元

产品编号：074815-01

# 译 者 序

ASP.NET 是 Web 开发的主要工具之一，构建在.NET Framework 之上。自微软推出这种技术，至今已经有将近二十年的时间。为了紧跟技术和应用环境的最新发展，微软在 ASP.NET 的基础上开发出了轻量级、跨平台的 ASP.NET Core。

ASP.NET Core 是微软新的宠儿，其开源性、跨平台性和轻量级特性意味着它会受到开发社区的欢迎，并且之前由于种种原因没有选择 ASP.NET 的公司和个人，可能从此选择 ASP.NET Core。无论从哪个角度看，ASP.NET Core 很快将会流行起来，并且预计在未来很长一段时期内会是业界的领头羊。因此，从现在开始了解 ASP.NET Core，不仅能接触到 Web 开发的前沿技术，满足开发人员的好奇心，也有助于将来需要使用 ASP.NET Core 进行开发时迅速上手。

选择阅读本书，读者必将受益匪浅。本书由点及面，将 ASP.NET Core 开发的理念和技术进行具体分析，寻根究底，力求深入，而后从生态系统的角度，讲解如何部署自己的 ASP.NET Core 应用程序。

本书作者对.NET 开发有着丰富的经验和深刻的认识，相信读者在阅读本书的过程中，经常会有醍醐灌顶的感觉。例如，作者开宗明义，一开始就解释了为什么在 ASP.NET 技术已经十分成熟的情况下，微软还要开发 ASP.NET Core。理解这个原因后，读者就对 ASP.NET Core 的理念、技术需求和应用场景有了总体认识。这种认识就像指路明灯，不仅使读者在学习 ASP.NET Core 的具体技术和实际应用时不会走入歧途，而且使理解 ASP.NET Core 的技术和具体应用变得容易许多。

之后本书通过一个基本的 ASP.NET Core 项目，帮助读者认识 ASP.NET Core 的结构、特点和基本应用。形成了总体认识，就能够更有目的地学习 ASP.NET Core。于是，在前面内容的基础上，作者分成几个部分讲解 ASP.NET Core 的方方面面。首先讲解 ASP.NET Core 的应用模型，细致探讨了 MVC 模型、控制器和视图部分，并对 Razor 语法进行了介绍。然后，讲解了一些普遍适用的问题，如设计上的一些考虑、应用安全、如何访问应用的数据等。Web 应用不只涉及后台，所以作者还讲解了前端的一些问题，包括如何设计 Web API，以及与客户端有关的一些问题。最后，作者从实用的角度出发，介绍了 ASP.NET Core 的生态系统，如何部署自己的 ASP.NET Core 应用，并分析了如何将现有的应用迁移到 ASP.NET Core。

在本书的翻译过程中，同事翟会昌给予了很大的帮助，另外刘治国、王惠良、韩江华、曹慕云、张海霞和李明明也耐心为我解答了一些有困惑的地方，帮助我保证技术翻译的准确性，在此一并致谢！

# 作 者 简 介

Dino Esposito 是 BaxEnergy 的一名数字策略师，迄今已经撰写了超过 20 本图书和 1000 篇文章。他的编程生涯已有 25 年。大家都公认，他撰写的图书和文章促进了全世界数千名.NET 开发人员和架构师的职业发展。Dino 的编程生涯始于 1992 年，当时他是一名 C 开发人员。他见证了.NET 的问世、Silverlight 的兴衰，以及各种架构模式的起起伏伏。他现在很期待人工智能 2.0 和区块链。他创作了 *The Sabbatical Break*——这是一部戏剧风格的作品，讲述了游历未被污染的想象空间，将软件、文学、科学、体育、技术和艺术融合在一起。可以通过 http://youbiquitous.net 联系他，也可以访问：

http://twitter.com/despos

http://instagram.com/desposofficial

http://facebook.com/desposofficial

# 前　言

ASP.NET Core 发展历程的某些方面让我想起了 15 年前 ASP.NET 刚问世的时候。1999 年秋天，当时还很年轻的 Scott Guthrie——现在担任 Microsoft 的副总裁——在伦敦向一小群 Web 开发人员展示了一个被称为 ASP+的新东西。当时还是 Active Server Pages 居于统治地位的时代，ASP+试图引入一种新语法，将 VBScript 代码放回服务器，并用一种编译语言来表达这种语法。ASP+是一项重大的成就。

Scott 进行展示时，公众还不知道有.NET，它要到第二年夏天才会正式公布。Scott 在一个独立的运行环境中进行演示(演示内容包括一个令人惊叹的 Web Service 示例)，这个运行时环境基于一个能够监听端口 80 的自定义工作进程(一个控制台应用程序)。最早的演示使用了普通的 Visual Basic 和 C++代码，以及 Win32 API。很快，ASP+被吸收到了新的.NET Framework 中，并最终蜕变为 ASP.NET。

ASP.NET Core 在一开始被展示时，同样作为一个新的独立框架，这是一个从头编写的框架，将 Microsoft 的 Web 堆栈的可扩展性和性能提升到了新高度。但在这个过程中，ASP.NET Core 的开发团队看到了一个诱人的机会来让 ASP.NET Core 框架在多个平台上可用。为实现这个目标，必须使.NET Framework 的一个子集在目标平台上可用，这意味着必须创建一个新的.NET Framework。最终，一个新的.NET Framework 被开发出来了。

在很长时间内，ASP.NET Core 是一个移动的目标，而移动这个目标的机制没有人清楚，并且没有被及时、有效地沟通。大约 20 年前，我们还没有如今这种社交媒体带来的即时分享的态度。而且，虽然 ASP+很可能也是一个移动的目标，但是 Microsoft 以外的人们(甚至 Microsoft 内没有直接参与 ASP+项目的人们)并不知道这一点。

虽然 ASP.NET 和 ASP.NET Core 的发展过程在关键方面可能看上去是相同的，但是它们的发展环境有很大区别。ASP.NET 之前的 Web 是新生阶段的 Web，可扩展的服务器端技术有限，而且可扩展性并不像今天这样是一个严峻的问题。同时，有大量应用程序需要针对 Web 重写，只是在等待由可靠的供应商提供的一个可靠的平台。

如今，即使不使用 ASP.NET Core，也仍然有许多框架可供使用。但是，ASP.NET Core 并不只是前端技术；它也是后端技术、Web API 以及要独立部署或者部署到 Service Fabric 的小型简洁的 Web(容器化)整体式应用程序。ASP.NET Core 还可以用在多个硬件/软件平台上。

很难说在近期甚至目前，ASP.NET Core 会不会成为每个公司和团队必须使用的技术。但可以肯定，ASP.NET Core 是 ASP.NET 开发人员需要了解的一种技术，是在多种平台上进行 Web 开发时可供使用的另一种全栈解决方案。

## 本书面向的读者对象

完全的新手(至少是对 Web 开发没有一点了解的新手)不适合阅读本书。本书针对的是 ASP.NET 开发人员，尤其是具有 MVC 背景的 ASP.NET 开发人员。同时，本书适合有丰富开发经验的 Web 开发人员，特别是具有 MVC 开发背景但是新接触 ASP.NET 的 Web 开发人员。虽然 ASP.NET Core 是一种全新的框架，但是它与 ASP.NET MVC 有许多共同点，与 Web Forms 也有少量共同点。

如果读者使用 Microsoft 技术或者计划使用 Microsoft 技术，那么对于全栈开发，ASP.NET Core 提供了一个出色的选择，包括与 Azure 云紧密结合起来。

## 本书的假定

本书假定读者对 Microsoft 堆栈(其他平台也可以)上的 Web 开发有基本了解，最好有成熟的理解。

## 本书不适合的读者对象

如果读者是 Web 编程的新手，从来没有听说过 ASP.NET，想要寻找一本 ASP.NET Core 的分步骤指南，那么本书可能不是一个理想选择。

## 本书结构

本书分为 5 个部分。

- 第 I 部分概述 ASP.NET Core 的基础知识，并介绍 hello-world 应用程序。
- 第 II 部分关注 MVC 应用程序模型，并介绍其核心组成，如控制器和视图。
- 第 III 部分介绍一些公共的开发问题，如身份验证、配置和数据访问。

- 第 IV 部分介绍用于构建可用的、有效的表示层的技术和其他框架。
- 第 V 部分介绍运行时管道、部署和迁移策略。

## 系统需求

要完成本书的练习，需要配备下面列出的硬件和软件：

- Windows 7 或更高版本，macOS 10.12 或更高版本。
- 或者，可使用众多 Linux 发行版中的一种，请参考 https://docs.microsoft.com/en-us/dotnet/core/linux-prerequisites。
- Visual Studio 2015 或更高版本的任意版本；Visual Studio Code。
- Internet 连接，以下载软件或者章节示例。

## 代码示例下载

本书中的所有代码，可在 https://aka.ms/ ASPNetCore/downloads 上找到，也可扫描封底二维码获取。

## 勘误、更新和图书支持

我们已经尽最大努力来确保本书及其配套内容的准确性。在以下网址，可以查阅本书的更新列表，其中列举了提交的勘误及对应的更正：

https://aka.ms/ASPNetCore/errata

如果读者发现了列表中没有列出的错误，请在该页面上把错误提交给我们。

如果需要额外的支持，请给 Microsoft Press Book Support 发送邮件，地址为 mspinput@microsoft.com。

请注意，上面列出的地址不提供对 Microsoft 的软件和硬件产品的支持。要想获得关于 Microsoft 的软件和硬件的帮助，请访问 http://support.microsoft.com。

## 保持联系

让我们保持对话！在 Twitter 上可以联系到我们：http://twitter.com/MicrosoftPress。

# 目　录


## 第 I 部分　新 ASP.NET 一览

第 1 章　为什么又开发一个 ASP.NET ····· 3
1.1　.NET 平台现状 ····· 4
1.1.1　.NET 平台的亮点 ····· 4
1.1.2　.NET Framework ····· 4
1.1.3　ASP.NET Framework ····· 5
1.1.4　Web API 框架 ····· 6
1.1.5　对极简 Web 服务的需求 ····· 7
1.2　15 年过去后的.NET ····· 7
1.2.1　更简洁的.NET Framework ····· 8
1.2.2　将 ASP.NET 与宿主解耦 ····· 9
1.2.3　新的 ASP.NET Core ····· 10
1.3　.NET Core 的命令行工具 ····· 10
1.3.1　安装 CLI 工具 ····· 10
1.3.2　dotnet 驱动程序工具 ····· 11
1.3.3　dotnet 的预定义命令 ····· 12
1.4　小结 ····· 13

第 2 章　第一个 ASP.NET Core 项目 ····· 15
2.1　ASP.NET Core 项目的分析 ····· 15
2.1.1　项目结构 ····· 16
2.1.2　与运行时环境交互 ····· 22
2.2　依赖注入子系统 ····· 27
2.2.1　依赖注入一览 ····· 27
2.2.2　ASP.NET Core 中的依赖注入 ····· 29
2.2.3　与外部 DI 库集成 ····· 31
2.3　构建极简网站 ····· 33
2.3.1　创建单端点网站 ····· 34
2.3.2　访问 Web 服务器上的文件 ····· 40
2.4　小结 ····· 44

## 第 II 部分　ASP.NET MVC 应用程序模型

第 3 章　启动 ASP.NET MVC ····· 47
3.1　启用 MVC 应用程序模型 ····· 47
3.1.1　注册 MVC 服务 ····· 48
3.1.2　启用传统路由 ····· 50
3.2　配置路由表 ····· 53
3.2.1　路由的剖析 ····· 54
3.2.2　路由的高级方面 ····· 59
3.3　ASP.NET MVC 的机制 ····· 62
3.3.1　操作调用程序 ····· 63
3.3.2　处理操作结果 ····· 64
3.3.3　操作筛选器 ····· 64
3.4　小结 ····· 65

第 4 章　ASP.NET MVC 控制器 ····· 67
4.1　控制器类 ····· 67
4.1.1　发现控制器的名称 ····· 68
4.1.2　继承的控制器 ····· 69
4.1.3　POCO 控制器 ····· 70
4.2　控制器操作 ····· 73

4.2.1 将操作映射到方法 ……73
4.2.2 基于特性的路由 ……77
4.3 实现操作方法 ……80
4.3.1 基本数据获取 ……80
4.3.2 模型绑定 ……82
4.3.3 操作结果 ……88
4.4 操作筛选器 ……91
4.4.1 操作筛选器的剖析 ……91
4.4.2 操作筛选器的小集合 ……94
4.5 小结 ……97

第 5 章 ASP.NET MVC 视图 …… 99
5.1 提供 HTML 内容 ……99
5.1.1 从终止中间件提供 HTML ……100
5.1.2 从控制器提供 HTML ……100
5.1.3 从 Razor 页面提供 HTML ……101
5.2 视图引擎 ……102
5.2.1 调用视图引擎 ……102
5.2.2 Razor 视图引擎 ……103
5.2.3 添加自定义视图引擎 ……108
5.2.4 Razor 视图的结构 ……109
5.3 向视图传递数据 ……113
5.3.1 内置的字典 ……113
5.3.2 强类型视图模型 ……116
5.3.3 通过 DI 系统注入数据 ……118
5.4 Razor 页面 ……118
5.4.1 引入 Razor 页面的理由 ……118
5.4.2 Razor 页面的实现 ……119
5.4.3 从 Razor 页面提交数据 ……120
5.5 小结 ……122

第 6 章 Razor 语法 ……123
6.1 语法元素 ……123
6.1.1 处理代码表达式 ……124
6.1.2 布局模板 ……128
6.1.3 分部视图 ……131
6.2 Razor 标记帮助程序 ……133
6.2.1 使用标记帮助程序 ……133
6.2.2 内置的标记帮助程序 ……135
6.2.3 编写自定义标记帮助程序 ……138
6.3 Razor 视图组件 ……141
6.3.1 编写视图组件 ……142
6.3.2 Composition UI 模式 ……143
6.4 小结 ……145

第 III 部分 跨领域关注点

第 7 章 设计考虑 ……149
7.1 依赖注入基础结构 ……149
7.1.1 进行重构以隔离依赖 ……149
7.1.2 ASP.NET Core DI 系统概述 ……152
7.1.3 DI 容器的各个方面 ……155
7.1.4 在层中注入数据和服务 ……156
7.2 收集配置数据 ……157
7.2.1 支持的数据提供程序 ……158
7.2.2 构建配置文档对象模型 ……160
7.2.3 传递配置数据 ……162
7.3 分层架构 ……164
7.3.1 表示层 ……165
7.3.2 应用层 ……167
7.3.3 领域层 ……167
7.3.4 基础结构层 ……168
7.4 处理异常 ……168
7.4.1 异常处理中间件 ……168
7.4.2 异常筛选器 ……171

7.4.3 记录异常……173
7.5 小结……174
第 8 章 应用程序安全……175
8.1 Web 安全基础结构……175
8.1.1 HTTPS 协议……175
8.1.2 处理安全证书……176
8.1.3 对 HTTPS 应用加密……176
8.2 ASP.NET Core 中的身份验证……176
8.2.1 基于 cookie 的身份验证……177
8.2.2 处理多个身份验证方案……179
8.2.3 建模用户身份……180
8.2.4 外部身份验证……184
8.3 通过 ASP.NET Identity 进行用户身份验证……189
8.3.1 ASP.NET Identity 概述……189
8.3.2 使用 User Manager……193
8.4 授权策略……197
8.4.1 基于角色的授权……198
8.4.2 基于策略的授权……201
8.5 小结……206
第 9 章 访问应用程序数据……207
9.1 创建相对通用的应用程序后端……208
9.1.1 整体式应用程序……208
9.1.2 CQRS 方法……210
9.1.3 基础结构层的构成……211
9.2 .NET Core 中的数据访问……212
9.2.1 Entity Framework 6.x……213
9.2.2 ADO.NET 适配器……215
9.2.3 使用微型 O/RM 框架……217
9.2.4 使用 NoSQL 存储……219
9.3 EF Core 的常见任务……220
9.3.1 建模数据库……221
9.3.2 处理表数据……224
9.3.3 处理事务……229
9.3.4 关于异步数据处理……231
9.4 小结……233
第 IV 部分 前端
第 10 章 设计 Web API……237
10.1 使用 ASP.NET Core 构建 Web API……237
10.1.1 公开 HTTP 端点……238
10.1.2 文件服务器……240
10.2 设计 RESTful 接口……242
10.2.1 REST 简介……242
10.2.2 在 ASP.NET Core 中使用 REST……245
10.3 保护 Web API 的安全……248
10.3.1 只计划真正需要的安全性……249
10.3.2 较为简单的访问控制方法……250
10.3.3 使用身份管理服务器……251
10.4 小结……258
第 11 章 从客户端提交数据……259
11.1 组织 HTML 表单……259
11.1.1 定义 HTML 表单……260
11.1.2 Post-Redirect-Get 模式……263
11.2 通过 JavaScript 提交表单……266
11.2.1 上传表单内容……266
11.2.2 刷新当前屏幕的一部分……270

11.2.3 将文件上传到 Web 服务器……272
11.3 小结……275
第 12 章 客户端数据绑定……277
12.1 通过 HTML 刷新视图……277
12.1.1 准备工作……278
12.1.2 定义可刷新区域……278
12.1.3 综合运用……278
12.2 通过 JSON 刷新视图……284
12.2.1 Mustache.JS 库简介……284
12.2.2 KnockoutJS 库简介……288
12.3 构建 Web 应用程序的 Angular 方法……293
12.4 小结……294
第 13 章 构建设备友好的视图……295
13.1 根据实际设备调整视图…295
13.1.1 HTML 5 在开发设备应用方面的优势……296
13.1.2 特征检测……298
13.1.3 客户端设备检测……300
13.1.4 Client Hints 即将问世……303
13.2 对设备友好的图片……303
13.2.1 PICTURE 元素……303
13.2.2 ImageEngine 平台……305
13.2.3 自动调整图片大小……305
13.3 面向设备的开发策略……307
13.3.1 以客户端为中心的策略……307
13.3.2 以服务器为中心的策略……311
13.4 小结……312
第 V 部分 ASP.NET Core 生态系统
第 14 章 ASP.NET Core 的运行时环境……315
14.1 ASP.NET Core 的宿主……315
14.1.1 WebHost 类……316
14.1.2 自定义宿主设置……319
14.2 内置的 HTTP 服务器……324
14.2.1 选择 HTTP 服务器……324
14.2.2 配置反向代理……326
14.2.3 Kestrel 的配置参数……329
14.3 ASP.NET Core 的中间件……331
14.3.1 管道架构……331
14.3.2 编写中间件组件……333
14.3.3 打包中间件组件……337
14.4 小结……339
第 15 章 部署 ASP.NET Core 应用程序……341
15.1 发布应用程序……341
15.1.1 在 Visual Studio 内发布应用程序……342
15.1.2 使用 CLI 工具发布应用程序……347
15.2 部署应用程序……348
15.2.1 部署到 IIS……349
15.2.2 部署到 Microsoft Azure……351
15.2.3 部署到 Linux……355
15.3 Docker 容器……357
15.3.1 容器与虚拟机……357
15.3.2 从容器到微服务架构……358
15.3.3 Docker 与 Visual Studio 2017……358

15.4　小结 …… 359

第 16 章　迁移和采用策略 ……361

16.1　寻找商业价值 …… 361

16.1.1　寻找益处 ……362

16.1.2　brownfield 开发 ……366

16.1.3　greenfield 开发 ……367

16.2　yellowfield 策略概述 …… 370

16.2.1　处理缺失的依赖 ……370

16.2.2　.NET Portability Analyzer ……370

16.2.3　Windows Compatibility Pack ……372

16.2.4　推迟跨平台挑战 ……372

16.2.5　走向微服务架构 ……373

16.3　小结 ……375

# 第 I 部分

# 新 ASP.NET 一览

欢迎来到 ASP.NET Core 的世界。

自 Microsoft 开发出 ASP.NET 和.NET Framework，已有超过 15 年的时间。在此期间，Web 开发已经发生了天翻地覆的变化。开发人员在这段时期学到了很多，客户则想让开发人员以新的方式把截然不同的解决方案交付到新的设备上。ASP.NET Core 反映了这些变化，而且考虑到了未来有可能发生的变化。本书第 I 部分依托背景介绍 ASP.NET Core，将帮助读者快速入门。

第 1 章解释了 ASP.NET Core 出现的原因、读者(特别是 ASP.NET MVC 开发人员)可能感到熟悉的地方，以及截然不同的地方。我们将在简洁的、模块化的、开源的、跨平台的.NET Core Framework 环境中探索 ASP.NET Core，并了解 ASP.NET Core 如何为极简 Web 服务和完整网站提供更好的支持。另外，第 1 章还会简单介绍 ASP.NET Core 的命令行接口(Command-Line Interface，CLI)开发工具。

在第 2 章，我们将快速创建自己的第一个应用程序。一些东西似乎从不会改变，如第一个开发的应用程序是 Hello World。我们将延续这个大家熟知的传统。但是即便如此，读者也会了解到 ASP.NET Core 令人惊讶的极简性，以及是什么在支持着这种极简风格。

# 第 1 章

# 为什么又开发一个 ASP.NET

如果我们希望事情保持原状，就到了必须发生改变的时候了。

——朱塞佩·托马西·迪·兰佩杜萨，《豹》

时间大概要回到1999年夏天。当时，为Windows操作系统编写软件需要具备C/C++技能和一些庞大的库，如 Microsoft Foundation Classes(MFC)和 ActiveX Template Library(ATL)，帮助简化开发。组件对象模型(Component Object Model，COM)正在成为 Windows 系统上运行的所有应用程序的根本基础。所有应用程序功能(包括数据访问)都将被重新设计为符合且能够感知COM。但是，选择什么编程语言和开发工具仍然是重要的考虑因素，如果数据访问或复杂的用户界面在要开发的Windows应用程序中必不可少，就尤为如此。如果选择了 Visual Basic，那么数据库访问是很容易实现的，也能够快速实现美观的用户界面，但缺点是不能使用函数指针，也不能访问(至少不能很容易、很可靠地访问)Windows SDK 的所有函数。另一方面，如果选择了 C 或 C++，就没有高层的数据访问工具可用，而且相比 Visual Basic，构建菜单或工具栏就困难多了。

对软件从业人员来说，当时的环境并不轻松，但是我们最终都设法找到了适合自己的领地，并能够运营壮大自己的业务。然而，突然间，.NET 出现了，一切都变得更好了。

# 1.1　.NET 平台现状

.NET 平台于 2000 年夏天公布，一年后进入第二个 beta 阶段。2002 年初，.NET 1.0 发布，不过从软件行业的角度看，那已经是几个地质时代之前了。

## 1.1.1　.NET 平台的亮点

.NET 平台由一个类框架和一个名为公共语言运行时(Common Language Runtime，CLR)的虚拟机组成。CLR 在本质上是一个执行环境，负责执行在概念上用类似于 Java 字节码的中间语言(Intermediate Language，IL)编写的代码。CLR 为运行代码提供了各种服务，例如内存管理和垃圾回收、异常处理、安全、版本管理、调试和分析。最重要的是，CLR 能以一种跨语言的方式提供这些服务。

在 CLR 之上是语言编译器和“托管语言”。托管语言即存在对应编译器的一种普通的编程语言；编译器能生成 IL 代码供 CLR 执行。所有.NET 编译器都会生成 IL 代码，但是 IL 代码不能直接在 Windows 操作系统上运行。因而，另一个工具被开发出来：实时(just-in-time)编译器。这种编译器将 IL 代码转换成能够直接在特定硬件/软件平台上运行的二进制代码。

## 1.1.2　.NET Framework

在当时，.NET 最令我感到震撼的是其在同一个项目中混合不同编程语言的能力。例如，可以用 Visual Basic 创建一个库，然后在使用其他任何托管语言编写的代码中调用这个库。另外，.NET 还提供了一个新的、极为强大的语言，也就是如今普遍使用的 C#语言，它在 Java 语言的灰烬上浴火重生。

总的来看，对开发人员来说，最大的变化是能用类访问底层 Windows SDK 的大部分功能。这些类构成了基类库(Base Class Library，BCL)，是任何.NET 应用程序都能够使用的公共基础代码。BCL 是与 CLR 紧密集成在一起的可重用类型的集合，包括基本类型、LINQ，以及对常见操作有帮助的类和类型，如 I/O、日期、集合和诊断。

BCL 得到了额外的一组针对性很强的库的补充，如用于数据库访问的 ADO.NET、用于桌面 Windows 应用程序的 Windows Forms、用于 Web 应用的 ASP.NET，以及 XML 等。这些额外的库逐渐壮大，吸收了一些庞大的框架，例如 Windows Presentation Foundation (WPF)、Windows Communication Foundation(WCF)和 Entity Framework(EF)。

BCL 与这些额外的框架一起构成了.NET Framework。

## 1.1.3　ASP.NET Framework

1999 年秋天，Microsoft 揭开了一个新 Web 框架的面纱，计划用其取代 Active Server Pages(ASP)。在最初的公开演示中，这个新框架被称为 ASP+，基于其自己的 C/C++引擎，后来被纳入.NET 平台，成为如今的 ASP.NET。

ASP.NET 框架包含 Internet Information Server(IIS)的一个扩展，能够捕捉传入的 HTTP 请求，并通过 ASP.NET 的运行时环境处理它们。在运行时框架中，通过找到能够处理该请求的特定组件，然后为浏览器准备一个 HTTP 响应包，来解析请求。运行时环境的结构就像一个管道：请求进入这个管道，经历不同的阶段，直到被完整处理，之后其响应被写回到输出流中。

与竞争对手不同的是，ASP.NET 提供一个有状态的、基于事件的编程模型，允许隐含的上下文从一个请求传递到另一个请求。桌面应用程序的开发人员很熟悉这种模型，而这种模型也将 Web 编程世界向诸多只有有限 HTML 和 JavaScript 技能(甚至完全不具备这些技能)的开发人员打开。由于最初的 ASP.NET 在 HTTP 和 HTML 之上添加了厚厚的抽象层，因此它吸引了众多 Visual Basic、Delphi、C/C++甚至 Java 程序员。

### 1. Web Forms 模型

ASP.NET 运行时环境在最初设计时有两个主要目标：

- 第一个目标是提供一个编程模型，尽可能使开发人员不必接触 HTML 和 JavaScript。Web Forms 模型受到经典的客户机/服务器请求的影响，应用效果极好，创建出了一个既包含免费服务器组件，又包含商用服务器组件的生态系统，提供了越来越高级的功能，如智能数据网格、输入表单、向导、日期选取器等。
- 第二个目标是尽量将 ASP.NET 和 IIS 混合到一起。ASP.NET 被设想为 IIS 的左膀右臂，而不只是一个插件，其运行时环境将成为 IIS 结构的一部分。2008 年发布的 IIS 7 是一个里程碑。在 IIS 7 及更高版本的集成管道(Integrated Pipeline)工作模式下，IIS 和 ASP.NET 共享相同的管道。请求在进入 IIS 时采取的路径与在 ASP.NET 中采取的路径相同。ASP.NET 代码只是负责处理请求，以及按照自己的需要拦截和预处理任何请求。

大约在 2009 年，Web Forms 编程模型得到 ASP.NET MVC 的补充。ASP.NET MVC 的出现受到与 ASP.NET 最初设想的目标完全不同的一种原则的启发。在 Web Forms 模型中，ASP.NET 页面通过服务器控件生成自己的 HTML，这也是 ASP.NET 能够取得成功并被快速接受的主要原因。这些服务器控件是黑盒组件(以声明的方式或者编程的方式配置)，为浏览器生成 HTML 和 JavaScript。但是，开发人员对于生

成的 HTML 只有程度有限的控制，而人们的需求在随着时间改变。

2. ASP.NET MVC 模型

ASP.NET MVC 是全新设计的，旨在更加接近 HTTP 协议工作；ASP.NET MVC 没有尝试隐藏 HTTP 的任何功能，而是要求开发人员熟悉 HTTP 请求和响应的机制。理想情况下，使用 ASP.NET MVC 的开发人员应该具备 JavaScript 和 CSS 技能。ASP.NET MVC 是在新的跨领域需求(例如关注点隔离、模块化和可测试性)的推动下，对编程模型进行重新设计的结果。

ASP.NET MVC 做出了一个也许并不容易的决定：不建立自己的运行时环境，而是作为现有 ASP.NET 运行时的一个插件。这既是一个好消息，也是一个坏消息。说是好消息，是因为可以选择通过 Web Forms 模型或者 ASP.NET MVC 模型来处理传入的请求，这样一来就很容易在一开始建立一个 Web Form 应用程序，然后逐渐将其演化成一个 ASP.NET MVC 应用程序。说是坏消息，是因为这样一来就不能解决 ASP.NET 在结构上的(按照现代需求来说)缺点。例如，ASP.NET MVC 团队尽力做到了能够模拟整个 HTTP 上下文，却无法在框架中建立完整的、规范的依赖注入基础结构。

对于处理必须返回 HTML 内容的 Web 请求，ASP.NET MVC 编程模型是最灵活、最容易理解的方式。但是，从某个时间开始，伴随着移动空间的爆炸式发展，HTML 不再是 HTTP 请求唯一可能的输出。

### 1.1.4　Web API 框架

移动设备出现后，能够请求 Web 端点，向任意类型的客户端提供任意类型的内容，如 JSON、XML、图片和 PDF。任何能够发出 HTTP 请求的一段代码都是 Web 端点的潜在客户端。而且，某些解决方案的可扩展性变得至关重要。

在 ASP.NET 的空间中，并没有多少余地来扩展其基础结构，进而适应新的场景：高度可扩展性、云和平台无关性。Web API 框架的出现旨在提供一个临时解决方案，应对瘦服务器的高需求，这种服务器能够提供 RESTful 接口，并与任意 HTTP 客户端进行对话，而不作任何假定或限制。Web API 框架是另外一组类，用于创建只知道完整的 HTTP 语法和语义的 HTTP 端点。Web API 框架提供的编程接口与 ASP.NET MVC 几乎相同，包括控制器、路由和模型绑定，但是在一个全新的运行时环境中运行它们。

在 ASP.NET Web API 中，让创建的 Web 框架与 Web 服务器解除耦合的观念开始生根，导致了 Open Web Interface for .NET(OWIN)标准被定义出来。OWIN 是一套规范，设定了 Web 服务器与 Web 应用程序的互操作规则。随着 OWIN 的问世，ASP.NET 最初的第二个目标(即 Web 宿主与 Web 应用程序的强耦合)成为往事。

任何遵守 OWIN 标准的应用程序都能够成为 Web API 的潜在宿主，但是 Web API 要想做到有用，就必须托管到 IIS 中，这就需要有一个 ASP.NET 应用程序。在 ASP.NET 应用程序(无论是 Web Forms 还是 MVC)中使用 Web API 会导致应用程序使用的内存量增加，因为使用了两个运行时环境。

### 1.1.5　对极简 Web 服务的需求

近年来，软件行业出现了另一个重要的变化，开始需要极简 Web 服务，也就是包围一段业务逻辑的一个薄薄的 Web 服务器层。

极简 Web 服务器是一个 HTTP 端点，客户端可调用这个端点来获取基本的、主要基于文本的内容。这样的 Web 服务器不需要运行复杂的、定制的管道，而是只需要接受 HTTP 请求，根据情况进行处理，然后返回一个 HTTP 响应。这个过程不应该有开销，或者应该只有上下文要求的开销。对客户端编程模型(如 Angular)的运用进一步增加了对这种 Web 服务的需求。

ASP.NET 及其所有运行时环境都不是针对类似的场景设计的。虽然 ASP.NET 运行时(既支持 Web Forms 应用，也支持 MVC 应用)在一定程度上可定制(禁用会话、输出缓存甚至身份验证)，但是并没有达到如今的一些业务场景所要求的粒度和控制级别。例如，ASP.NET 几乎无法转变为有效的静态文件服务器。

## 1.2　15 年过去后的.NET

对于任何软件来说，15 年都不是一个很短的时间，.NET Framework 也不例外。ASP.NET 是在 20 世纪 90 年代后期设计出来的，而 Web 的变化非常迅速。大约在 2014 年，ASP.NET 团队开始计划一个新的 ASP.NET，并按照 OWIN 规范，设计了一个全新的运行时环境。

该团队的主要目的是移除对旧有的 ASP.NET 运行时—— system.web 程序集是其象征的依赖。不过，该团队还有一个关键目标：使开发人员能够完全控制管道，从而既能够构建极简 Web 服务，又能够构建完整的网站。在这个过程中，该团队面临着一个难题：确保吞吐量，并在保证成本较低的前提下通过云平台有效地提供任意解决方案，使应用的内存占用显著减少。不止如此，当时的.NET Framework 还必须接受特殊处理，以实现减重。

新的 ASP.NET 的指导原则可归结如下：

- 使 ASP.NET 既能够访问完整的现有.NET Framework，又能够访问其精简的版本，这种版本去除了所有很少使用的、用途也不大的依赖。
- 使新的 ASP.NET 环境与宿主 Web 服务器解耦。

然而，当实现了这个计划后，又出现了其他许多问题和机遇。机遇如此诱人，让人不能白白错过。

## 1.2.1　更简洁的.NET Framework

新的 ASP.NET 的设计伴随着一个新的.NET Framework，后者最终被命名为.NET Core Framework。可以把这个新的框架视为原来的.NET Framework 的一个子集，它被专门设计成更加细粒度、更加精简，更重要的是，被设计为能够支持跨平台使用。这个设计目标通过两种方式实现：移除一些功能并重写其他功能，以提高在某些情况下的有效性，补偿对所移除功能的依赖。

.NET Core Framework 主要被设计为用于 ASP.NET 应用程序。这个因素最终引导着在.NET Core Framework 中包含哪些库和丢弃哪些库。.NET Core Framework 为执行应用程序提供了一个新的运行时，称为 CoreCLR。CoreCLR 的布局和架构与目前的.NET CLR 相同，负责加载 IL 代码，编译成机器代码，以及回收垃圾。CoreCLR 不支持目前的 CLR 的某些功能，例如应用程序域和代码访问安全，这些功能被证明并非是必要的，或者是专门针对 Windows 平台的，所以难以移植到其他平台。不止如此，.NET Core Framework 的类库用包的形式提供，而包的粒度很小，比目前的.NET Framework 小得多。

.NET Core 平台是完全开源的。表 1-1 给出了相关存储库的链接。

表 1-1　.NET Core 源代码的 Github 链接

| 平台 | 描述 | 链接 |
| --- | --- | --- |
| CoreCLR | CLR 及相关工具 | http://github.com/dotnet/coreclr |
| CoreFX | .NET Core Framework | http://github.com/dotnet/corefx |

简言之，完整的.NET Framework 与.NET Core Framework 之间的区别可归结如下：

- .NET Core Framework 更加精简，模块化程度更高。
- .NET Core Framework(及相关工具)是开源的。
- .NET Core Framework 只能用来编写 ASP.NET 和控制台应用程序。
- .NET Core Framework 可与应用程序一同部署，而完整的.NET Framework 只能安装到目标机器上，由所有应用程序共享。可以看到，这一点为版本管理带来了很大的问题。

去掉了平台依赖性以后，就能够修改新的、更加精简的.NET Framework，使其能够在其他操作系统上工作。这是.NET Core Framework 与现有的.NET Framework 的又一大区别。.NET Core Framework 可用于编写跨平台的应用程序，让它们也能运行在 Linux 和 Mac 操作系统上。

注意：

.NET Core 2.0 发布后，完整的.NET Framework 和.NET Core Framework 之间的功能差异正在缩小，因为 Core Framework 中已经移植了更多的类和名称空间(如 System.Drawing 和数据表类)。但是，并不能认为.NET Core Framework 是完整的.NET Framework 的复制品。.NET Core Framework 是从头开始重新设计的一个新框架，看上去与完整的.NET Framework 很类似，但是能够跨平台工作。

## 1.2.2　将 ASP.NET 与宿主解耦

为了使 Web 应用程序模型既能够用于编写极简 Web 服务，又能够用于编写完整的网站，将 ASP.NET 与 IIS 解耦被证明是必要的一步。OWIN 的理念(参见 http://owin.org)是：

- 将 Web 服务器的功能与 Web 应用程序的功能隔离开。
- 鼓励为.NET Web 开发设计出更简单的模块，当这些模块结合起来时，能够实现真实网站的强大力量。

图 1-1 显示了 OWIN 的整体架构。

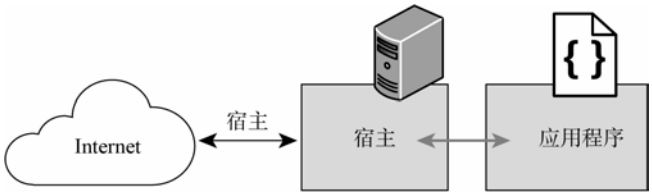


图 1-1　开放 Web 接口架构

在基于 OWIN 的架构中，宿主 Web 服务器不再必须是 IIS。而且，宿主接口可用控制台应用程序或者 Windows 服务实现。在满足了这些限制后，采用 OWIN 开放接口的 Web 应用程序模型的真正威力就会展现：同一个应用程序可以在任何符合 OWIN 的 Web 服务器上托管，系统平台是什么并不重要。

HTTP 协议是平台无关的，所以当构建出一个新的.NET Framework 版本，使其不再紧密依赖特定的平台(如 Windows)时，构建一个能够跨平台工作的 Web 应用程序模型就成了一个可行的、很有吸引力的项目。

重要

当 IIS 在 2008 年开始支持集成管道模式时，Microsoft 对 Web 的观念与如今全然不同。在某种程度上，环境也不同了。按照集成管道的观念，IIS 和 ASP.NET 需要密切合作，看起来就像一个统一的引擎。为新的 ASP.NET 构建的模型推翻了集成管道的观念，认为 ASP.NET 是一个独立的环境，可以在任何 Web 服务器上托管。这种模型认为，在某些情况下，这种独立的环境甚至在直接呈现给外界时也能够工作。

### 1.2.3　新的 ASP.NET Core

ASP.NET Core 是一个新的框架，用于构建多种基于 Internet 的应用程序，主要是(但不限于)Web 应用程序。事实上，可以把特殊的 Web 应用程序看成内置 IoT 的服务器和面向 Web 的服务，例如移动应用程序的后台。

在编写 ASP.NET Core 应用程序时，可使其针对.NET Core Framework，也可以使其针对完整的.NET Framework。ASP.NET Core 被设计为跨平台的，使开发人员能够创建运行在 Windows、Mac 和 Linux 上的应用程序。ASP.NET Core 包含一个内置的 Web 服务器和一个运行应用程序代码的运行时环境。应用程序代码是用稍作调整的 ASP.NET MVC 框架编写的，并依赖于一组系统模块。这些系统模块被设计为极小的模块，从而提供更多的机会来构建只要最小开销就能运行的应用程序。图 1-2 显示了 ASP.NET Core 的整体架构。


图 1-2　ASP.NET Core 的整体架构

**注意：**

并不是严格需要 Web 服务器(如 IIS 或 Apache)，因为内置的 Web 服务器(Kestrel)能够被直接公开给外界。是否需要一个独立的 Web 服务器主要取决于 Kestrel 能否满足需要。

新的 ASP.NET 依赖于.NET Core SDK 的工具来构建和运行应用程序。下一节将详细介绍.NET SDK 和命令行工具。第 14 章将详细介绍 ASP.NET Core 运行时。

## 1.3　.NET Core 的命令行工具

在.NET Core 中，所有的基本开发工具(即用于构建、测试、运行和发布应用程序的工具)也作为命令行应用程序提供。这些应用程序统称为.NET Core 命令行接口(CLI)。

### 1.3.1　安装 CLI 工具

在能够开发和部署.NET Core 应用程序的所有平台上都可以使用 CLI 工具。CLI 工具通常提供了针对具体平台定制的安装包，例如 Linux 上的 RPM 或 DEB 包，或

者 Windows 上的 MSI 包。运行安装程序后，CLI 工具将安全地存储到磁盘上一个可全局访问的位置。图 1-3 显示了一台 Windows 计算机上 CLI 工具的存储文件夹。

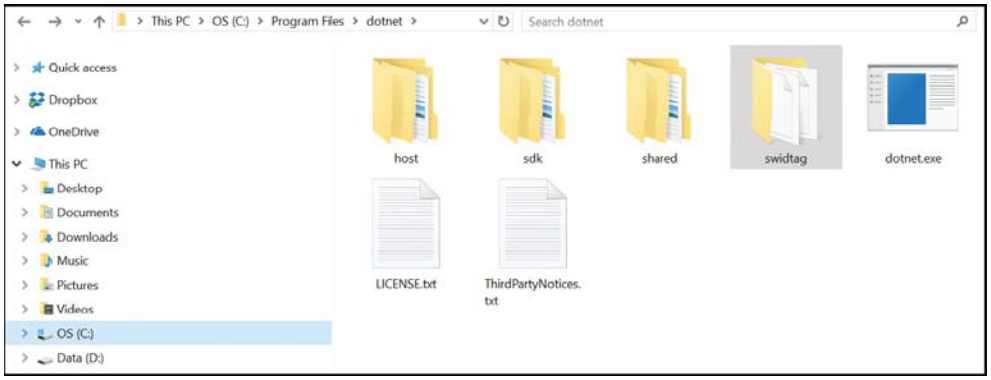


图 1-3　已安装的 CLI 工具

注意，可以同时运行多个版本的 CLI 工具。当安装多个版本的时候，默认情况下运行的是最新版本。

## 1.3.2　dotnet 驱动程序工具

CLI 一般被称为一组工具，但实际上，它是由一个宿主工具(称为驱动程序)运行的一组命令。这个宿主工具就是 dotnet.exe(参见图 1-3)。命令行指令的格式如下所示：

```
dotnet [host-options] [command] [arguments] [common-options]
```

[command]占位符代表将在驱动程序工具中执行的命令，而[arguments]代表传递给该命令的参数。稍后将介绍宿主选项和公共选项。

当安装了 CLI 的多个版本，但是不想运行最新的版本时，可以在应用程序所在的文件夹中创建一个 global.json 文件，在其中至少添加下面的内容。

```
{
 "sdk": {
   "version": "2.0.0"
 }
}
```

version 属性的值决定了使用哪个版本的 CLI 工具。

注意：

CLI 工具的版本不同于应用程序使用的.NET Core 运行时的版本。运行时的版本是在项目文件中指定的，也可以在使用的 IDE 界面内进行编辑。如果想手动编辑这个项目文件，那么只需要编辑.csproj XML 文件，修改 TargetFramework 元素的值。

这个值的名称就代表了版本(如 netcoreapp2.0)。

### 1. 宿主选项

在 dotnet 工具的命令行中，宿主选项代表的是 dotnet 工具的配置。先传递了宿主选项之后，才会传递命令。宿主选项支持 3 个值，分别用于获得关于 CLI 工具和运行时环境的常规信息、获得 CLI 的版本号，以及启用诊断(参见表 1-2)。

表 1-2　CLI 的宿主选项

| 平台 | 描述 |
| --- | --- |
| -d 或--diagnostics | 启用诊断输出 |
| --info | 显示运行时环境和.NET CLI 的信息 |
| --version | 显示.NET CLI 的版本号 |

### 2. 公共选项

表 1-3 中的公共 CLI 选项是所有命令共有的选项，例如获得帮助和启用详细输出。

表 1-3　CLI 的公共选项

| 平台 | 描述 |
| --- | --- |
| -v 或--verbose | 启用详细输出 |
| -h 或--help | 显示关于如何使用 dotnet 工具的常规帮助 |

## 1.3.3　dotnet 的预定义命令

默认情况下，安装 CLI 工具后，就可以使用表 1-4 中列出的命令。注意，表 1-4 尽量按照真实使用这些命令的顺序介绍它们。

表 1-4　常用的 CLI 命令

| 命令 | 描述 |
| --- | --- |
| new | 使用某个可用的模板创建一个新的.NET Core 应用程序。默认的模板包括控制台应用程序，以及 ASP.NET MVC 应用程序、测试项目和类库。还有额外的选项可指定目标语言和项目的名称 |
| restore | 恢复项目的所有依赖。从项目文件读取依赖后，将其恢复为从配置好的源使用的 NuGet 包 |
| build | 构建项目及其全部依赖。应在项目文件中指定编译器的参数(如构建的是库还是应用程序) |

(续表)

| 命令 | 描述 |
| --- | --- |
| run | 编译源代码(如有必要)，生成可执行文件并执行。执行此命令前，必须先执行 build 命令 |
| test | 使用配置好的测试运行程序在项目内执行单元测试。单元测试是依赖于特定单元测试框架及其运行程序的类库 |
| publish | 编译应用程序(如有必要)，从项目文件中读取依赖列表，然后将得到的一组文件发布到一个输出目录 |
| pack | 使用项目的二进制文件创建一个 NuGet 包 |
| migrate | 将原来的基于 project.json 的项目迁移为基于 msbuild 的项目 |
| clean | 清理项目的输出文件夹 |

要详细了解如何调用上述命令，可在命令行中输入下面的命令：

```
dotnet <command> --help
```

通过在项目中引用可移植的控制台应用程序或者将可执行文件复制到 PATH 环境变量关联着的某个目录中(可全局使用)，可添加更多命令。

# 1.4　小结

.NET 平台问世已经超过 15 年了，在这段时间内，它吸引了大量投入，变得非常流行。然而，世界总在不断地变化，朱塞佩·托马西·迪·兰佩杜萨的小说《豹》中有一句话说得再准确不过：如果我们希望事情保持原状，就到了必须发生改变的时候了。因而，原来的.NET 平台——那个围绕着一个庞大的类库和一些应用程序模型(ASP.NET、Windows Forms 和 WPF)设计出来的.NET 平台——如今也正在经历着深刻的重新设计。之所以说“正在经历”，是因为重新设计的工作在 2014 年开始，到版本 2.0 实现第一个里程碑，但是在未来仍将继续进行下去。

从业务上来说，你可能想也可能还不想现在就拥抱这个新的平台，但是我相信，在几年内，新的平台将成为首选的平台和要迁移到的平台。新平台的亮点在于高度模块化和跨平台的本质。基于.NET Core 的任何代码将能够运行在 Linux、Mac 或 Windows 平台上，只是需要不同的运行时。由于非常侧重跨平台的开发，因此操作对应平台的所有核心工具(生成、运行、测试和发布)都作为命令行工具提供，在这些命令行工具的基础上可构建 IDE。.NET Core 的命令行接口被称为 CLI 工具。

在第 2 章，我们将开始介绍本书的核心主题：ASP.NET 和 Web 开发。

# 第 2 章

# 第一个 ASP.NET Core 项目

所有动物生来平等，但有些动物比其他动物更平等。

——乔治·奥威尔，《动物庄园》

ASP.NET Core 是基于.NET Core 平台的一个面向 Web 的应用程序模型。虽然名称中带有熟悉的 ASP.NET 字样，但是 ASP.NET Core 与前一个 ASP.NET 版本并不相同。最重要的是，ASP.NET Core 有一个全新的运行时环境，只支持一种应用程序模型：ASP.NET MVC。这意味着这个新的 Web 框架与 Web Forms 全然不同，甚至与 Web API 也不完全相同。ASP.NET Core 是全新的框架，只有 ASP.NET MVC 编程模型(控制器、视图和路由)中的一部分代码和技能能够在这个框架中重用。


在本章和本书剩余内容中，我们将提到非.NET Core ASP.NET(包括 Web Forms、ASP.NET MVC 和 Web API)的特点和实现细节，并将之与 ASP.NET Core 的特点进行比较。为避免混淆，我们使用术语“经典 ASP.NET”指代 ASP.NET Core 出现之前的 ASP.NET 中可用的任意应用程序模型。

## 2.1 ASP.NET Core 项目的分析

有几种不同的方式可创建一个新的 ASP.NET Core 项目。首先，可以使用 Visual Studio 中提供的某个标准项目模板。其次，可在 CLI 工具中使用 New 命令。如果使