

.NET 开发经典名著

C#高级编程

(第 11 版)

C# 7 & .NET Core 2.0

[美] 克里斯琴·内格尔(Christian Nagel) 著

李 铭 译

清华大学出版社

北 京

Christian Nagel

Professional C# 7 and .NET Core 2.0

EISBN: 978-1-119-44927-0

Copyright © 2018 by John Wiley & Sons, Inc.

All Rights Reserved. This translation published under license.

Trademarks: Wiley, the Wiley logo, Wrox, the Wrox logo, Programmer to Programmer, and related trade dress are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates, in the United States and other countries, and may not be used without written permission. All other trademarks are the property of their respective owners. John Wiley & Sons, Inc., is not associated with any product or vendor mentioned in this book.

本书中文简体字版由 Wiley Publishing, Inc. 授权清华大学出版社出版。未经出版者书面许可, 不得以任何方式复制或抄袭本书内容。

北京市版权局著作权合同登记号 图字: 01-2018-4100

Copies of this book sold without a Wiley sticker on the cover are unauthorized and illegal.

本书封面贴有 Wiley 公司防伪标签, 无标签者不得销售。

版权所有, 侵权必究。侵权举报电话: 010-62782989 13701121933

图书在版编目(CIP)数据

C#高级编程: 第11版: C# 7 & .NET Core 2.0 / (美)克里斯琴·内格尔(Christian Nagel) 著; 李铭 译. —北京: 清华大学出版社, 2019

(.NET 开发经典名著)

书名原文: Professional C# 7 and .NET Core 2.0

ISBN 978-7-302-52256-0

I. ①C… II. ①克… ②李… III. ①C 语言—程序设计 IV. ①TP312.8

中国版本图书馆 CIP 数据核字(2019)第 016880 号

责任编辑: 王 军 于 平

装帧设计: 孔祥峰

责任校对: 成凤进

责任印制:

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 刷 者:

装 订 者:

经 销: 全国新华书店

开 本: 190mm×260mm 印 张: 67.25 字 数: 2273 千字

版 次: 2019 年 3 月第 1 版 印 次: 2019 年 3 月第 1 次印刷

定 价: 198.00 元

产品编号:

C#是微软公司在 2000 年 6 月发布的一种面向对象的、运行于 .NET Framework 之上的高级程序设计语言，由 Anders Hejlsberg 主持开发，它是第一个面向组件的编程语言，其源码会编译成 MSIL 再运行。C#是微软公司 .NET Framework 的主角。C#是一种安全的、稳定的、简单的、优雅的、由 C 和 C++ 衍生出来的编程语言。它在继承 C 和 C++ 强大功能的同时，去掉了它们的一些复杂特性，使程序员可以快速地编写各种基于微软 .NET 平台的应用程序。

C#是兼顾系统开发和应用开发的最佳实用语言，很有可能成为编程语言历史上的第一个“全能”型语言。它提供了以下软件工程要素的支持：强类型检查、数组维度检查、未初始化的变量引用检测、自动垃圾收集。目前，C#已经发布到 7.2 版本，其中：

- C# 1.0：纯粹的面向对象。
- C# 2.0：增加了泛型编程新概念。
- C# 3.0：率先实现了 LINQ 的语言。
- C# 4.0：支持动态编程。
- C# 5：支持异步编程。
- C# 6 和 .NET Core 1.0 提供了代码的共享，.NET Core 运行在 Windows、Linux 和 Mac 操作系统上。因此自从 .NET Core 推出以来，就可以在任何操作系统上构建程序。
- C# 7 和 .NET Core 2.0 提供了函数式编程。

随着微软加快了发布速度，同时在每次更新中都提供了更重要的改进，对新工具和新特性的快速处理就变得前所未有的重要。《C#高级编程(第 11 版) C#7&.NET Core 2.0》就是为了达到这个目的而设计的，关于 C#的一切都在这里。

本书为有经验的程序员提供了他们需要与世界领先的编程语言有效合作的信息。最新的 C 语言增加了许多新的特性并更新了一些特性，帮助你在更短的时间内完成更多的工作，本书是快速入门的理想指南。C# 7 重点关注代码简化和性能，对本地函数、元组类型、记录类型、模式匹配、非可空引用类型、不可变类型提供了新的支持。Visual Studio 的改进将给 C 开发人员与空间交互的方式带来重大改变，将 .NET 引入非微软平台，并将 Docker、GULP 和 NPM 等工具与其他平台结合起来。

本书分四个部分，介绍 C#语言及其在各个领域中的应用。第 I 部分给出 C#语言的良好背景知识。第 II 部分介绍独立于应用程序类型的 .NET Core 和 Windows Runtime。第 III 部分论述 Web 应用程序和服务。第 IV 部分介绍如何使用 XAML 构建应用程序，包括 Universal Windows 应用程序和 Xamarin 应用程序。

无论你是 C#新手，还是刚刚迁移到 C# 7，如果希望对最新特性有一个扎实的掌握，能够利用该语言的全部功能来创建健壮的、高质量的应用程序，本书都是你需要知道的所有内容的一站式指南。

在这里要感谢清华大学出版社的编辑，他们为本书的翻译投入了巨大的热情，并付出了很多心血。没有你们的帮助和鼓励，本书不可能顺利付梓。本书主要章节由李铭翻译，参与翻译的还有陈妍、何美英、陈宏波、熊晓磊、管兆昶、潘洪荣、曹汉鸣、高娟妮、王燕、谢李君、李珍珍、王璐、王华健、柳松洋、曹晓松、陈彬、洪妍、刘芸、邱培强、高维杰、张素英、颜灵佳、方峻、顾永湘、孔祥亮。

对于这本经典之作，译者本着“诚惶诚恐”的态度，在翻译过程中力求“信、达、雅”，但是鉴于译者水平有限，错误和失误在所难免，如有任何意见和建议，请不吝指正。

译者

作者简介



Christian Nagel 是 Visual Studio 和开发技术方向的 Microsoft MVP, 担任微软开发技术代言人(Microsoft Regional Director)已经超过 15 年。Christian 是 CN innovation 公司的创始人, CN innovation 公司提供指导、培训、代码评审, 并协助使用微软技术设计和开发解决方案。他拥有超过 25 年的软件开发经验。

Christian Nagel 最初在 Digital Equipment 公司通过 PDP 11 和 VAX / VMS 系统开始他的计算机职业生涯, 接触过各种语言和平台。在 2000 年, .NET 只有一个技术概览版时, 他就开始使用各种技术建立.NET 解决方案。目前, 他主要指导人们开发和设计 Windows 应用程序、ASP.NET Core Web 应用程序和 Xamarin, 并帮助他们使用 Microsoft Azure 服务产品。

在软件开发领域工作多年以后, Christian 仍然热爱学习和使用新技术, 并通过多种形式教别人如何使用新技术。他的 Microsoft 技术知识非常渊博, 编写了很多书, 拥有微软认证培训师(MCT)和微软认证解决方案开发专家(MCSD)认证。Christian 经常在国际会议(如 Microsoft Ignite、BASTA! 和 TechDays)上发言。他创立了 INETA Europe 来支持.NET 用户组。他的联系网站是 www.cninnovation.com, 博客是 <https://csharp.christiannagel.com>, 其 Twitter 账号是@christiannagel。

技术编辑简介



István Novák 是 SoftwArt 的合伙人和首席技术顾问, SoftwArt 是匈牙利的一家小型 IT 咨询公司。István 是一名软件架构师和社区的传教士。在过去 25 年里,他参加了五十多个企业软件开发项目。2002 年,他在匈牙利与他人合作出版了第一本关于 .NET 开发的图书。2007 年,他获得微软最有价值专家(MVP)头衔。2011 年,他成为微软开发技术代言人(Microsoft Regional Director)。István 与他人合作出版了 *Visual Studio 2010 and .NET 4 Six-in-One*(Wiley, 2010)和《Windows 8 应用开发入门经典》(Wiley, 2012),独立撰写了《Visual Studio 2010 LightSwitch 开发入门经典》(Wiley, 2011)。

István 从匈牙利的布达佩斯技术大学获得硕士学位和软件技术的博士学位。他与妻子和两个女儿居住在匈牙利的 Dunakeszi。他是一个充满激情的潜水员,在一年的任何季节都会去红海潜水。

致 谢

我要感谢 Charlotte Kughen，他使本书的文本更具可读性。.NET Core 在不断演变，Charlotte 为我提供了巨大的帮助，使本书阐述的.NET 技术能与时俱进。她还利用许多周末的时间帮助本书快速出版。也特别感谢 István Novák，他作为技术编辑，校正了本书中的一些错误。

.NET Core 的飞速发展和我在书中使用的临时构建存在一些问题，István 向我挑战，改进了代码示例，让读者更容易理解。谢谢你们：Charlotte 和 István，你们让本书的质量上了一个大台阶。

我还要感谢.NET Core 团队的 Richard Lander。我们就 C#高级编程第 11 版的内容和方向进行了热烈的讨论。Rich 还抽出时间给我提供了一些关于本书中几个章节的好建议。

我也要感谢 Kenyon Brown，以及 Wiley 出版社帮助出版本书的其他人。我还要感谢我的妻子和孩子，为了撰写本书，我花费了大量的时间，包括晚上、周末和冬季假日，但你们很理解并支持我。Angela、Stephanie、Matthias 和 Katharina，你们是我深爱的人。没有你们，本书不可能顺利出版。

许多年过去了，.NET 有了新的发展势头。.NET Framework 有一个年轻的兄弟.NET Core！

.NET Framework 是封闭的源代码，只能在 Windows 系统上使用。现在.NET Core 是开源的，可以在 Linux 上使用，并且使用现代模式。在.NET 生态系统中有很多巨大的改进。

注意：

由于最近的变化，C#在最受欢迎的编程语言中排名前十，而.NET Core 在最受欢迎的框架中排名第三。在 Web 和桌面开发人员中，C#在最流行的语言中排名第三。详情请登录 <https://insights.stackoverflow.com/survey/2017>。

使用 C#和 ASP.NET Core，可以创建运行在 Windows、Linux 和 Mac 上的 Web 应用程序和服务。使用 Windows Runtime(Windows 运行库)，可以通过 C#和 XAML 以及.NET Core 创建本机 Windows 应用程序(也称为通用 Windows 平台，UWP)。通过 Xamarin，使用 C#和 XAML 可以创建运行在 Android 和 iOS 设备上的应用程序。在.NET Standard 的帮助下，可以创建能在 ASP.NET Core、Windows 应用、Xamarin 中共享的库，还可以创建传统的 Windows Forms 和 WPF 应用程序。所有这些都在本书中介绍。

本书大部分示例都建立在带有 Visual Studio 的 Windows 系统上。许多示例也在 Linux 上进行了测试，并在 Linux 和 Mac 上运行。除了 Windows 应用程序示例之外，还可以使用 Mac 的 Visual Studio Code 或 Visual Studio for the Mac 作为开发环境。

0.1 .NET Core 的世界

.NET 有很长的历史，但是.NET Core 很年轻。.NET Core 2.0 从.NET Framework 中获得了许多新的 API，使其更容易将现有的.NET Framework 应用程序迁移到.NET Core 的新世界。

一个简单的步骤是，可以创建使用.NET Standard 2.0 的库，这些库可以在.NET Framework 4.6.1 及以上版本的应用程序、.NET Core 2.0 应用程序和 Windows Build 16299 以上版本的应用程序中使用。

现在，没有理由不在后端使用 ASP.NET Core。随着迁移到.NET Standard 的简化，越来越多的库可以在.NET Core 中使用。总体来看，ASP.NET Core MVC 与 ASP.NET MVC 非常相似。然而 ASP.NET Core MVC 要灵活得多，使用.NET Core 模式时更容易操作，也更容易扩展。

对于创建新的 Web 应用程序，可能只需要使用新技术 Razor Pages。如果应用程序增长，Razor Pages 可以很容易地扩展到使用 ASP.NET Core MVC 的模型-视图-控制器模式。

在撰写本文时，用于实时通信的技术 SignalR 的.NET Core 版本即将发布。

ASP.NET Core 与 JavaScript 技术(如 Angular 和 React/Redux)的结合非常有效，甚至还有模板使用这些技术以及用于后端服务的 ASP.NET Core 创建项目。

注意：

可以通过 <https://github.com/dotnet/corefx> 访问.NET Core 的源代码。.NET Core 命令行可以在 <https://github.com/dotnet/cli> 上使用。在 <https://github.com/aspnet> 上有许多 ASP.NET Core 的存储库。其中包括 ASP.NET Core MVC、Razor、SignalR、EntityFrameworkCore 等。

下面是对 .NET Core 部分特性的总结：

- .NET Core 是开源的。
- .NET Core 使用现代模式。
- .NET Core 支持在多个平台上开发。
- ASP.NET Core 可以在 Windows 和 Linux 上运行。

使用 .NET Core 时，会发现这项技术是 .NET 自从第一个版本以来最大的改变。 .NET Core 是一个新的开始，从这里可以继续我们的旅程，快速进行新的开发。

0.2 C#的重要性

C# 在 2002 年发布时，是一个用于 .NET Framework 的开发语言。C# 的设计思想来自 C++、Java 和 Pascal。Anders Hejlsberg 从 Borland 来到微软公司，带来了开发 Delphi 语言的经验。Hejlsberg 在微软公司开发了 Java 的 Microsoft 版本 J++，之后创建了 C#。

注意：

Anders Hejlsberg 现在已经转移到 *TypeScript* (而他仍在影响 C#)，Mads Torgersen 是 C# 的项目负责人。C# 的改进可以在 <https://github.com/dotnet/csharplang> 上公开讨论。在这里，可以阅读 C# 语言建议和会议记录，也可以提交自己的 C# 建议。

C# 一开始不仅作为一种面向对象的通用编程语言，而且是一种基于组件的编程语言，支持属性、事件、特性(注解)和构建程序集(包括元数据的二进制文件)。

随着时间的推移，C# 增强了泛型、语言集成查询(Language Integrated Query, LINQ)、lambda 表达式、动态特性和更简单的异步编程。C# 编程语言并不简单，它提供了很多功能，而且实际使用的功能在不断进化。因此，C# 不仅是面向对象或基于组件的语言，它还包括函数式编程的理念，开发各种应用程序的通用语言会实际应用这些理念。

在 C# 6 中，编译器的源代码完全重写了。不仅新的编译器管道可以在自定义程序中使用，微软还获得了一些新的资源，使变更不会破坏程序的其他部分。因此，增强编译器变得容易了。

C# 7 再次添加了许多具有函数编程背景的新特性，如本地函数、元组和模式匹配。

0.3 C# 7 的新特性

C# 6 扩展包括 `static using`、基于表达式体的方法和属性、自动实现的属性初始化器、只读自动属性、`nameof` 操作符、空条件运算符、字符串插入、字典初始化器、异常过滤器以及 `catch` 中等待。C# 7 的新特性是什么？

0.3.1 数字分隔符

数字分隔符使代码更具可读性。在声明变量时可以给单独的数字添加 `_`。编译器只是删除 `_`。下面的代码片段在 C# 7 中看起来更具可读性：

C# 6

```
long n1 = 0x1234567890ABCDEF;
```

C# 7

```
long n2 = 0x_1234_5678_90AB_CDEF;
```

在 C# 7.2 中，也可以把 “`_`” 放在开头。

C# 7.2

```
long n2 = 0x_1234_5678_90AB_CDEF;
```

数字分隔符在第 2 章介绍。

0.3.2 二进制字面值

C# 7 为二进制提供了一个新的字面值。二进制的值只能是 0 和 1。现在数字分隔符变得尤为重要：

C# 7

```
uint binary1 = 0b1111_0000_1010_0101_1111_0000_1010_0101;
```

二进制字面值在第 2 章介绍。

0.3.3 表达式体的成员

C# 6 允许使用表达式体的方法和属性。在 C# 7 中，表达式体可以与构造函数、析构函数、本地函数、属性访问器等一起使用。这里可以看到属性访问器在 C# 6 和 C# 7 之间的区别：

C# 6

```
private string _firstName;
public string FirstName
{
    get { return _firstName; }
    set { Set(ref _firstName, value); }
}
```

C# 7

```
private string _firstName;
public string FirstName
{
    get => _firstName;
    set => Set(ref _firstName, value);
}
```

表达式体的成员在第 3 章介绍。

0.3.4 out 变量

在 C# 7 之前，out 变量必须在使用之前声明。而在 C# 7 中，代码减少了一行，因为变量可以在使用时声明：

C# 6

```
string n = "42";
int result;
if (string.TryParse(n, out result))
{
    Console.WriteLine($"Converting to a number was successful: {result}");
}
```

C# 7

```
string n = "42";
if (string.TryParse(n, out var result))
{
    Console.WriteLine($"Converting to a number was successful: {result}");
}
```

这个特性在第 3 章介绍。

0.3.5 不拖尾的命名参数

C# 支持可选参数需要的命名参数，但在任何情况下都可以支持可读性。C# 7.2 支持不拖尾的命名参数。参

数名可以添加到 C# 7.2 的任何参数中：

C# 7.0

```
if (Enum.TryParse(weekdayRecommendation.Entity, ignoreCase: true,
result: out DayOfWeek weekday))
{
    reservation.Weekday = weekday;
}
```

C# 7.2

```
if (Enum.TryParse(weekdayRecommendation.Entity, ignoreCase: true,
out DayOfWeek weekday))
{
    reservation.Weekday = weekday;
}
```

命名参数在第 3 章介绍。

0.3.6 只读结构

结构应该是只读的(有一些例外)。在 C# 7.2 中，可以使用 `readonly` 修饰符声明结构体，因此编译器可以验证结构体没有更改。编译器也可以使用此保证，不复制作为参数传递给它的结构，但把它传递为引用：

C# 7.2

```
public readonly struct Dimensions
{
    public double Length { get; }
    public double Width { get; }

    public Dimensions(double length, double width)
    {
        Length = length;
        Width = width;
    }

    public double Diagonal => Math.Sqrt(Length * Length + Width * Width);
}
```

只读结构在第 3 章介绍。

0.3.7 in 参数

C# 7.2 还允许给参数使用 `in` 修饰符。这就保证了所传递的值类型不会更改，并可以通过引用来传递，以避免复制：

C# 7.2

```
static void CantChange(in AStruct s)
{
    // s can't change
}
```

`ref`、`in` 和 `out` 修饰符在第 3 章介绍。

0.3.8 Private Protected

C# 7.2 添加了一个新的访问修饰符 `private protected`。如果某成员用于同一程序集中的类型，或派生自类的另一个程序集的类型，那么访问修饰符 `protected internal` 允许访问它。使用 `private protected` 时，上述两个条件使用 AND 而不是 OR——只有当类派生自基类，且位于同一程序集中时，才允许访问。

访问修饰符在第 4 章中介绍。

0.3.9 目标类型的 default

在 C# 7.1 中，定义了 `default` 字面值，与 `default` 操作符相比，它允许使用更短的语法。`default` 操作符总是需要类型的重复，现在不再需要了。这适用于复杂类型：

C# 7.0

```
int x = default(int);
ImmutableArray<int> arr = default(ImmutableArray<int>);
```

C# 7.1

```
int x = default;
ImmutableArray<int> arr = default;
```

`default` 字面值在第 5 章介绍。

0.3.10 本地函数

在 C# 7 之前，不可能在方法中声明函数。但可以创建一个 `lambda` 表达式并调用它，如 C# 6 代码片段所示：

C# 6

```
public void SomeFunStuff()
{
    Func<int, int, int> add = (x, y) => x + y;

    int result = add(38, 4);
    Console.WriteLine(result);
}
```

在 C# 7 中，可以在方法中声明一个本地函数。本地函数只能在方法的作用域内访问：

C# 7

```
public void SomeFunStuff()
{
    int add(int x, int y) => x + y;

    int result = add(38, 4);
    Console.WriteLine(result);
}
```

本地函数在第 13 章解释。本书的几个章节介绍了它的不同用途。

0.3.11 元组

元组允许组合不同类型的对象。在 C# 7 之前，元组是 .NET Framework 中的 `Tuple` 类。可以使用 `Item1`、`Item2`、`Item3` 等访问元组的成员。在 C# 7 中，元组是该语言的一部分，可以定义成员的名称：

C# 6

```
var t1 = Tuple.Create(42, "astring");
int i1 = t1.Item1;
string s1 = t1.Item2;
```

C# 7

```
var t1 = (n: 42, s: "magic");
int i1 = t1.n;
string s1 = t1.s;
```

除此之外，新的元组是值类型(`ValueTuple`)，而 `Tuple` 类型是引用类型。元组的所有更改都包含在第 13 章中。

0.3.12 推断的元组名

C# 7.1 通过自动推断元组名称来扩展元组，类似于匿名类型。在 C# 7.0 中，元组的成员总是需要命名。如果元组成员的名称应该与分配给它的属性或字段相同，那么在 C# 7.1 中，如果不提供名称，它就与分配给它的成员的名称相同：

C# 7.0

```
var t1 = (FirstName: racer.FirstName, Wins: racer.Wins);
int wins = t1.Wins;
```

C# 7.1

```
var t1 = (racer.FirstName, racer.Wins);
int wins = t1.Wins;
```

0.3.13 拆解

不，这不是打印错误。拆解不是析构函数。元组可以拆解成独立的变量，例如：

C# 7

```
(int n, string s) = (42, "magic");
```

如果定义了 `Deconstruct` 方法，也可以拆解 `Person` 对象：

C# 7

```
var p1 = new Person("Tom", "Turbo");
(string firstName, string lastName) = p1;
```

拆解在第 13 章介绍。

0.3.14 模式匹配

使用模式匹配，`is` 操作符和 `switch` 语句增强为三种模式：`const` 模式、类型模式和 `var` 模式。下面的代码片段显示了使用 `is` 操作符的模式。第一个检查匹配常数 42，第二个检查 `Person` 对象，第三个用 `var` 模式检查每个对象。使用类型和 `var` 模式，可以为强类型访问声明一个变量：

C# 7

```
public void PatternMatchingWithIsOperator(object o)
{
    if (o is 42)
    {
    }
    if (o is Person p)
    {
    }
    if (o is var v1)
    {
    }
}
```

通过 `switch` 语句，可以对 `case` 子句使用相同的模式。如果模式匹配，则可以将变量声明为强类型。也可以在以下条件下使用 `when` 过滤模式：

C# 7

```
public void PatternMatchingWithSwitchStatement(object o)
{
    switch (o)
    {
        case 42:
            break;
    }
}
```

```

        case Person p when p.FirstName == "Katharina":
            break;
        case Person p:
            break;
        case var v:
            break;
    }
}

```

模式匹配在第 13 章中介绍。

0.3.15 Throw 表达式

抛出异常只有在语句中才可行，在表达式中是不可行的。因此，当使用构造函数接收参数时，需要对 `null` 进行额外的检查，才能抛出 `ArgumentNullException` 异常。在 C# 7 中，可以在表达式中抛出异常，因此可以使用合并操作符在左侧为 `null` 时抛出 `ArgumentNullException` 异常。

C# 6

```

private readonly IBooksService _booksService;
public BookController(BooksService booksService)
{
    if (booksService == null)
    {
        throw new ArgumentNullException(nameof(b));
    }
    _booksService = booksService;
}

```

C# 7

```

private readonly IBooksService _booksService;
public BookController(BooksService booksService)
{
    _booksService = booksService ?? throw new ArgumentNullException(nameof(b));
}

```

Throw 表达式在第 14 章中介绍。

0.3.16 异步 Main()方法

在 C# 7.1 之前，Main()方法总是需要声明为类型 `void`。在 C# 7.1 中，Main()方法也可以是 `Task` 类型，使用 `async` 和 `await` 关键字：

C# 7.0

```

static void Main()
{
    SomeMethodAsync().Wait();
}

```

C# 7.1

```

async static Task Main()
{
    await SomeMethodAsync();
}

```

异步编程在第 15 章介绍。

0.3.17 引用语义

.NET Core 主要关注性能的提高。为引用语义添加 C# 特性有助于提高性能。在 C# 7 之前，`ref` 关键字可以与参数一起使用，通过引用传递值类型。现在也可以对返回类型和本地变量使用 `ref` 关键字。

下面的代码片段声明方法 `GetNumber`，以返回对 `int` 的引用。这样，调用者可以直接访问数组中的元素，并可以更改其内容：

C# 7.0

```
int[] _numbers = { 3, 7, 11, 15, 21 };
public ref int GetNumber(int index)
{
    return ref _numbers[index];
}
```

在 C# 7.2 中, **readonly** 修饰符可以添加到 **ref** 返回值上。这样, 调用者不能更改返回值的内容, 但仍然使用引用语义, 并可以避免在返回结果时复制值类型。调用方收到引用, 但不允许更改引用:

C# 7.2

```
int[] _numbers = { 3, 7, 11, 15, 21 };
public ref readonly int GetNumber(int index)
{
    return ref _numbers[index];
}
```

在 C# 7.2 之前, C# 可以创建引用类型(类)和值类型(结构)。然而, 装箱时, 结构体也可以存储在堆中。在 C# 7.2 中, 可以声明一个类型 **ref struct**, 该类型只允许放在堆栈上:

C# 7.2

```
ref struct OnlyOnTheStack
{
}
```

引用的新特性在第 17 章中介绍。

0.4 ASP.NET Core 中的新特性

在 .NET Core 和 Visual Studio 2017 中, 有一个新的项目文件。在 Visual Studio 2015 中是预览版本的 .NET Core 工具, 在 Visual Studio 2017 中已经发布了。该工具使用 **csproj** 文件切换到 MSBuild 环境, 所以现在有了 **csproj** 文件用于 .NET Framework 和 .NET Core 应用程序。然而, 这并不是前几代的 **csproj** 文件。现在的 **csproj** 文件要短得多, 简化了许多, 也可以使用简单的文本编辑器来修改它们。

.NET Core 2.0 通过 .NET Standard 2.0 中定义的类和方法得到增强, 这更便于将现有的 .NET Framework 应用程序迁移到 .NET Core。

创建 ASP.NET Core 项目, 不仅简化了 **csproj** 文件, 而且简化了 C# 源代码。使用默认的 **WebHostBuilder** 时, 会预定义更多的内容。添加配置和日志提供程序时, 不需要自己添加它们。在 ASP.NET Core MVC 中, 有了一些小的改进——例如, 视图组件现在可以在标记辅助程序中使用。

还有一种新的技术——**Razor** 页面, 比 ASP.NET Core MVC 更容易学习。有些应用程序不需要来自模型-视图-控制器模式的抽象, 此时就可以使用 **Razor** 页面。

0.5 UWP 的新特性

Windows 10 一年更新两次(如果在 Windows Insiders 程序中, 就会更频繁地得到更新, 但这对大多数用户来说并不正常)。每次 Windows 更新都会发布一个新的 SDK。最新的两项更新是 **Creators Update**(构建号 15063, 2017 年 3 月)和 **Fall Creators Update**(构建号 16299, 2017 年 10 月)。

微软继续提供集成在 Windows 控件中的新设计特性。新的设计命名为 **Fluent Design**, 它集成到标准控件中, 也可以直接访问——例如, 使用 **acrylic** 和 **reveal brushes**。在应用程序中通过 **ParallaxView** 添加了视差效果。

添加特性也提高了生产力。可以使用 **Windows Template Studio** (Visual Studio 的一个扩展) 中的模板编辑器创建多个页面, 并使用预先生成的服务。

XAML 通过有条件的 XAML 进行了增强, 使其更容易支持多个 Windows 10 版本, 但使用旧版本 Windows 10 中没有的新功能。

InkCanvas 控件提供了新的标尺，可以轻松地集成到应用程序中。NavigationView 可以很容易地创建带有汉堡包按钮和 SplitView 的自适应菜单。本书的IV部分将详细介绍所有这些新特性以及更多的内容。

0.6 编写和运行 C#代码的环境

.NET Core 运行在 Windows、Linux 和 Mac 操作系统上。使用 Visual Studio Code(<https://code.visualstudio.com>)，可以在任何操作系统上创建和构建程序。

最好用的开发工具是 Visual Studio 2017，也是本书使用的工具。可以使用 Visual Studio Community 2017 版(<https://www.visualstudio.com>)，但本书介绍的一些功能只有 Visual Studio 的企业版提供。需要企业版时会提到。Visual Studio 2017 需要 Windows 10 构建号 1507 或更高版本，要求使用 Windows 8.1、Windows Server 2012 R2 或 Windows 7 SP1。要构建和运行本书中的 Windows 应用程序(UWP)，需要 Windows 10。

要为 iOS 创建和构建 Xamarin 应用程序，还需要用于构建系统的 Mac。没有 Mac，仍然可以为 Windows 和 Android 开发 Xamarin 应用程序。

在Mac上开发应用程序，可以使用Visual Studio for Mac: <https://www.visualstudio.com/vs/Visual-Studio-Mac/>。可以使用这个工具创建ASP.NET Core和Xamarin应用程序，但是不能创建和测试Windows应用程序。

0.7 本书内容

本书首先在第 1 章介绍.NET 的整体体系结构，给出编写托管代码所需要的背景知识。此后概述不同的应用程序类型，学习如何用新的开发环境 CLI 编译程序，介绍在 Visual Studio 中开始学习的最重要部分。之后本书分几部分介绍 C#语言及其在各个领域中的应用。

第 I 部分——C#语言

该部分介绍 C#语言的良好背景知识。尽管这一部分假定读者是有经验的编程人员，但它没有假设读者拥有任何特定语言的知识。首先介绍 C#的基本语法和数据类型，再介绍 C#的面向对象特性，之后介绍 C#中的一些高级编程主题，如委托、lambda 表达式和语言集成查询(Language Integrated Query, LINQ)。

由于 C#包含许多函数式编程的特性，因此需要了解元组和模式匹配的函数式编程基础，讨论异步编程和引用语义的新语言特性。本部分最后探讨 Visual Studio 2017 的许多特性，还将了解 Docker 的基础知识，以及 Visual Studio 2017 支持 Docker 的方式。

第 II 部分——.NET Core 与 Windows Runtime

第 19~29 章介绍独立于应用程序类型的.NET Core 和 Windows Runtime(运行库)。本部分在第 19 章中介绍如何创建库和 NuGet 包，学习如何以最好的方式使用.NET 标准。

依赖注入(Dependency Injection, DI)与.NET Core 一起使用：服务注入 Entity Framework Core 和 ASP.NET Core。ASP.NET Core MVC 使用了数百个服务。DI 便于跨 WPF、UWP 和 Xamarin 使用相同的代码。第 20 章专门介绍 DI 的基础，还可以在 DI 容器 Microsoft.Extensions.DependencyInjection 中学到高级特性，包括调整非微软容器。其他许多章节也使用 DI。

第 21 章介绍使用任务并行库(Task Parallel Library, TPL)进行并行编程，以及用于同步的各种对象。

第 22 章学习如何访问文件系统，读取文件和目录，了解如何使用 System.IO 名称空间中的流和 Windows 运行库中的流来编写 Windows 应用程序。

第 23 章学习使用套接字和使用更高级别的抽象(如 HttpClient)的联网的核心基础。

第 24 章利用流来了解安全性，以及如何加密数据，允许进行安全的转换。本章还讨论了创建 Web 应用程序时需要了解的一些主题，如 SQL 注入和跨站点请求伪造攻击的问题。

第 25 和 26 章展示了如何访问数据库。第 25 章使用 ADO.NET 直接解释事务，并涵盖了使用.NET 核心的环境事务。第 26 章介绍了 Entity Framework Core 2.0 提供的所有新特性。EF Core 2.0 有旧的 Entity Framework 6.x

无法提供的许多特性。

第 27 章介绍使用本地化的技术本地化应用程序，该技术对 Windows 和 Web 应用程序都非常重要。

用 C# 代码创建功能时，不要跳过创建单元测试的步骤。一开始需要更多的时间，但随着时间的推移，添加功能和维护代码时，就会看到其优势。第 28 章介绍如何创建单元测试，包括 Visual Studio 2017 中的实时单元测试、网络测试和编码的 UI 测试。

第 29 章介绍了 .NET Core 的日志功能，以及使用 Visual Studio AppCenter 获取分析信息。

第 III 部分——Web 应用程序和服务

本部分将介绍 Web 应用程序和服务。本部分从第 30 章开始，提供了 ASP.NET Core 的基础。使用 MVC 模式创建 Web 应用程序，包括新技术 Razor 页面，在第 31 章中介绍。第 32 章涵盖了 ASP.NET Core 的 REST 服务特性：Web API。

第 IV 部分——应用程序

本部分介绍如何使用 XAML 构建应用程序——Universal Windows 应用程序和 Xamarin。了解 Windows 应用程序的基础，包括第 33 章中 XAML 的基础，其中包含 XAML 语法、依赖属性和标记扩展，可以在其中创建自己的 XAML 语法。本章介绍了 Windows 控件的不同类别以及与 XAML 绑定数据的基础。

第 34 章主要关注 MVVM(模型-视图-视图模型)模式。该章将学习如何利用基于 XAML 的应用程序的数据绑定特性，这些特性允许在 Windows 应用程序、WPF 和 Xamarin 之间共享大量代码。也可以分享许多为 iOS 和 Android 平台开发的代码。创建 WPF 应用程序并没有在本书中介绍——这一技术在最近几年并没有得到多少改进，应该考虑转向通用 Windows 平台，如果使用第 34 章学到的知识，就更容易实现这一点。WPF 应用程序仍然需要维护。要想更深入地了解 WPF，应该阅读本书的上一版。

第 35 章介绍基于 XAML 的应用程序的样式化。第 36 章介绍了用通用 Windows 平台创建 Windows 应用程序的高级功能。展示了应用程序服务、上墨(inking)、AutoSuggest 控件、高级编译绑定特性等。

第 37 章帮助启动 Windows、Android 和 iPhone 的 Xamarin 开发，并展示幕后发生的事情。学习 Xamarin.Android、Xamarin.iOS 和 Xamarin.Forms 之间的不同之处。了解 Xamarin.Forms 控件与 Windows 控件的不同之处，更快地从 Windows 开发转向 Xamarin。本章的较大示例使用与第 34 章中的 Windows 应用程序相同的 MVVM 库。

附加的章节

附加的第 1 章讨论了 Microsoft Composition，它允许创建容器和部件之间的独立性。附加的第 2 章论述如何将对象序列化到 XML 和 JSON 中，以及用于读取和编写 XML 的不同技术。

Web 应用程序的发布和订阅技术使用 ASP.NET Core 技术 WebHooks 和 SignalR 的形式，在附加的第 3 章中讨论。附加的第 4 章对使用 Bot 服务和 Azure 认知服务创建应用程序有了新的认识。

附加的第 5 章涵盖了一些与 Windows 应用程序相关的额外主题：使用相机、地理定位来访问当前的位置信息，以各种格式显示地图的 MapControl，以及几个传感器(比如提供光线信息和测量重力的传感器)。

可以扫描封底二维码查看附加的 5 章内容。

0.8 如何下载本书的示例代码

在读者学习本书中的示例时，可以手工输入所有的代码，也可以使用本书附带的源代码文件。本书使用的所有源代码都可以从本书合作站点 <http://www.wrox.com/> 上下载。登录到站点 <http://www.wrox.com/>，使用 Search 工具或书名列表就可以找到本书。接着单击本书细目页面上的 Download Code 链接，就可以获得所有的源代码，也可以扫描封底的二维码获取本书的源代码。

注意：

许多图书的书名都很相似, 所以通过 ISBN 查找本书是最简单的, 本书英文版的 ISBN 是 978-1-119-44927-0。

在下载代码后, 只需要用自己喜欢的解压缩软件对它进行解压缩即可。另外, 也可以进入 <http://www.wrox.com/dynamic/books/download.aspx> 上的 Wrox 代码下载主页, 查看本书和其他 Wrox 图书的所有代码。

0.9 GitHub

源代码也可以在 GitHub 上提供, 网址是 <https://www.github.com/ProfessionalCSharp/ProfessionalCSharp7>。在 GitHub 中, 还可以使用 Web 浏览器打开每个源代码文件。使用这个网站时, 可以把完整的源代码下载到一个 zip 文件。还可以将源代码复制到系统上的本地目录。只需要安装 git 工具, 为此可以使用 Visual Studio 或者从 <https://git-scm.com/downloads> 下载 Windows、Linux 和 Mac 的 git 工具。要将源代码复制到本地目录, 请使用 git clone:

```
> git clone https://www.github.com/ProfessionalCSharp/ProfessionalCSharp7
```

使用此命令, 把完整的源代码复制到子目录 ProfessionalCSharp7。之后就可以开始处理源文件了。

Visual Studio 的更新可用, SignalR 库发布后, 源代码将在 GitHub 上更新。如果在复制源代码之后源代码发生了变化, 就可以在将当前目录更改为源代码目录后, 提取最新的更改:

```
> git pull
```

如果对源代码做了一些更改, git pull 可能会导致错误。如果发生这种情况, 可以把更改隐藏起来, 然后再取出来:

```
> git stash
> git pull
```

git 命令的完整列表可以在 <https://git-scm.com/docs> 上找到。

如果源代码有问题, 可以在存储库中报告问题。在浏览器中打开 <https://github.com/ProfessionalCSharp/ProfessionalCSharp7>, 单击 Issues 选项卡, 单击 New Issue 按钮。这将打开一个编辑器, 如图 1 所示, 尽可能详细地描述问题。

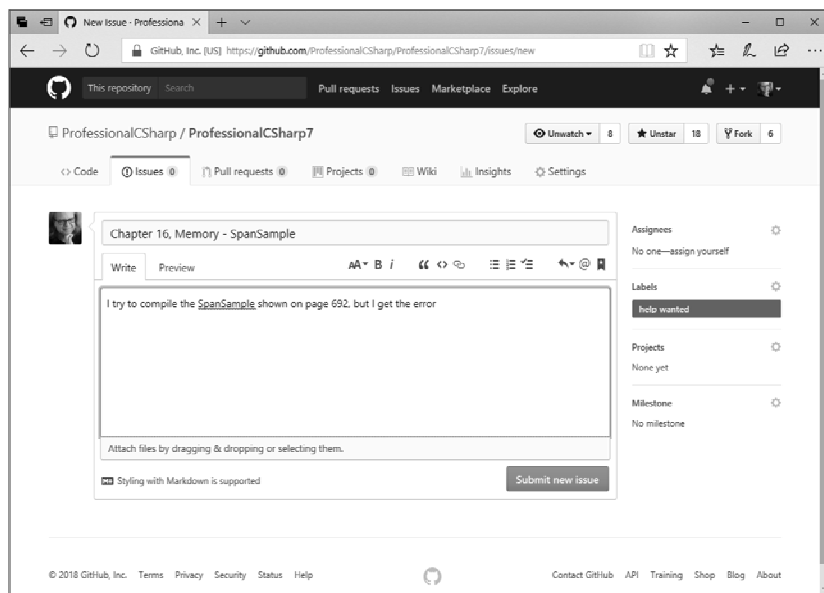


图 1

为了报告问题，需要一个 GitHub 账户。如果有一个 GitHub 账户，也可以将源代码存储库分叉到账户上。有关使用 GitHub 的更多信息，请查看 <https://guides.github.com/activities/hello-world>。

注意：

可以读取源代码和相关问题，并在不加入 GitHub 的情况下在本地复制存储库。要在 GitHub 上发布问题并创建自己的存储库，需要自己的 GitHub 账户。

0.10 勘误表

尽管我们已经尽力保证不出现错误，但错误总是难免的，如果你在本书中找到了错误，如拼写错误或代码错误，请告诉我们，我们将非常感激。通过勘误表，可以让其他读者避免受挫，当然，这还有助于提供更高质量的信息。

要在网站上找到本书的勘误表，可以登录 <http://www.wrox.com>，通过 Search 工具或书名列表查找本书，然后在本书的细目页面上，单击 Book Errata 链接。在这个页面上可以查看 Wrox 编辑已提交和粘贴的所有勘误项。完整的图书列表还包括每本书的勘误表，网址是 www.wrox.com/misc-pages/booklist.shtml。

如果没有在 Book Errata 页面上发现自己找到的错误，请访问 www.wrox.com/contact/techsupport.shtml，填好表单，将你找到的错误发送给我们。我们将检查信息，如果合适的话，将消息发送到该书的勘误页面，并在该书的后续版本中修复问题。

目 录

第 I 部分 C# 语言

第 1 章 .NET 应用程序和工具	3
1.1 选择技术	3
1.2 回顾.NET 历史	4
1.2.1 C# 1.0——一种新语言	4
1.2.2 带有泛型的 C# 2 和.NET 2	6
1.2.3 .NET 3.0——Windows Presentation Foundation	6
1.2.4 C# 3 和.NET 3.5——LINQ	6
1.2.5 C# 4 和.NET 4.0——dynamic 和 TPL	7
1.2.6 C# 5 和异步编程	7
1.2.7 C# 6 和.NET Core 1.0	8
1.2.8 C# 7 和.NET Core 2.0	8
1.2.9 选择技术, 继续前进	9
1.3 .NET 术语	10
1.3.1 .NET Framework	11
1.3.2 .NET Core	11
1.3.3 .NET Standard	11
1.3.4 NuGet 包	12
1.3.5 名称空间	12
1.3.6 公共语言运行库	13
1.3.7 Windows 运行库	13
1.4 用.NET Core CLI 编译	14
1.4.1 设置环境	14
1.4.2 创建应用程序	15
1.4.3 构建应用程序	16
1.4.4 运行应用程序	16
1.4.5 创建 Web 应用程序	17
1.4.6 发布应用程序	17
1.5 使用 Visual Studio 2017	19
1.6 应用程序类型和技术	24
1.6.1 数据访问	24
1.6.2 Windows 应用程序	24
1.6.3 Xamarin	24
1.6.4 Web 应用程序	25

1.6.5 Web API	25
1.6.6 WebHooks 和 SignalR	25
1.6.7 Microsoft Azure	25
1.7 开发工具	26
1.7.1 Visual Studio Community	27
1.7.2 Visual Studio Professional	27
1.7.3 Visual Studio Enterprise	27
1.7.4 Visual Studio for Mac	27
1.7.5 Visual Studio Code	27
1.8 小结	27
第 2 章 核心 C#	29
2.1 C#基础	29
2.2 变量	31
2.2.1 初始化变量	31
2.2.2 类型推断	32
2.2.3 变量的作用域	33
2.2.4 常量	34
2.3 预定义数据类型	35
2.3.1 值类型和引用类型	35
2.3.2 .NET 类型	36
2.3.3 预定义的值类型	36
2.3.4 预定义的引用类型	40
2.4 程序流控制	42
2.4.1 条件语句	42
2.4.2 循环	44
2.4.3 跳转语句	47
2.5 名称空间	47
2.5.1 using 语句	48
2.5.2 名称空间的别名	49
2.6 Main()方法	49
2.7 使用注释	50
2.7.1 源文件中的内部注释	50
2.7.2 XML 文档	51
2.8 C#预处理器指令	52
2.8.1 #define 和#undef	52

2.8.2	#if、#elif、#else 和#endif	52	4.3.3	隐藏方法	89
2.8.3	#warning 和#error	53	4.3.4	调用方法的基类版本	90
2.8.4	#region 和#endregion	53	4.3.5	抽象类和抽象方法	90
2.8.5	#line	53	4.3.6	密封类和密封方法	91
2.8.6	#pragma	54	4.3.7	派生类的构造函数	91
2.9	C#编程准则	54	4.4	修饰符	93
2.9.1	关于标识符的规则	54	4.4.1	访问修饰符	93
2.9.2	用法约定	55	4.4.2	其他修饰符	94
2.10	小结	58	4.5	接口	94
第 3 章	对象和类型	59	4.5.1	定义和实现接口	95
3.1	创建及使用类	60	4.5.2	派生的接口	97
3.2	类和结构	60	4.6	is 和 as 运算符	98
3.3	类	61	4.7	小结	99
3.3.1	字段	61	第 5 章	泛型	100
3.3.2	只读字段	61	5.1	泛型概述	100
3.3.3	属性	62	5.1.1	性能	101
3.3.4	匿名类型	65	5.1.2	类型安全	102
3.3.5	方法	66	5.1.3	二进制代码的重用	102
3.3.6	构造函数	69	5.1.4	代码的扩展	102
3.4	结构	73	5.1.5	命名约定	102
3.4.1	结构是值类型	74	5.2	创建泛型类	103
3.4.2	只读结构	75	5.3	泛型类的功能	105
3.4.3	结构和继承	75	5.3.1	默认值	106
3.4.4	结构的构造函数	75	5.3.2	约束	106
3.4.5	ref 结构	76	5.3.3	继承	108
3.5	按值和按引用传递参数	76	5.3.4	静态成员	108
3.5.1	ref 参数	77	5.4	泛型接口	109
3.5.2	out 参数	77	5.4.1	协变和逆变	109
3.5.3	in 参数	78	5.4.2	泛型接口的协变	110
3.6	可空类型	79	5.4.3	泛型接口的逆变	111
3.7	枚举类型	79	5.5	泛型结构	111
3.8	部分类	81	5.6	泛型方法	113
3.9	扩展方法	83	5.6.1	泛型方法示例	113
3.10	Object 类	83	5.6.2	带约束的泛型方法	114
3.11	小结	84	5.6.3	带委托的泛型方法	115
第 4 章	继承	85	5.6.4	泛型方法规范	115
4.1	面向对象	85	5.7	小结	116
4.2	继承的类型	85	第 6 章	运算符和类型强制转换	117
4.2.1	多重继承	86	6.1	运算符和类型转换	117
4.2.2	结构和类	86	6.2	运算符	118
4.3	实现继承	86	6.2.1	运算符的简化操作	119
4.3.1	虚方法	87	6.2.2	运算符的优先级和关联性	125
4.3.2	多态性	88	6.3	使用二进制运算符	126

6.3.1 位的移动	128	7.11 数组池	167
6.3.2 有符号数和无符号数	128	7.11.1 创建数组池	168
6.4 类型的安全性	129	7.11.2 从池中租用内存	168
6.4.1 类型转换	130	7.11.3 将内存返回给池	168
6.4.2 装箱和拆箱	132	7.12 小结	169
6.5 比较对象的相等性	133	第 8 章 委托、lambda 表达式和事件	170
6.5.1 比较引用类型的相等性	133	8.1 引用方法	170
6.5.2 比较值类型的相等性	134	8.2 委托	170
6.6 运算符重载	135	8.2.1 声明委托	171
6.6.1 运算符的工作方式	135	8.2.2 使用委托	172
6.6.2 运算符重载的示例: Vector 结构	136	8.2.3 简单的委托示例	174
6.6.3 比较运算符的重载	139	8.2.4 Action<T>和 Func<T>委托	175
6.6.4 可以重载的运算符	140	8.2.5 BubbleSorter 示例	176
6.7 实现自定义的索引运算符	141	8.2.6 多播委托	177
6.8 用户定义的类型强制转换	142	8.2.7 匿名方法	180
6.8.1 实现用户定义的类型强制转换	143	8.3 lambda 表达式	181
6.8.2 多重类型强制转换	147	8.3.1 参数	181
6.9 小结	150	8.3.2 多行代码	181
第 7 章 数组	151	8.3.3 闭包	182
7.1 相同类型的多个对象	151	8.4 事件	182
7.2 简单数组	152	8.4.1 事件发布程序	182
7.2.1 数组的声明	152	8.4.2 事件侦听器	184
7.2.2 数组的初始化	152	8.5 小结	185
7.2.3 访问数组元素	153	第 9 章 字符串和正则表达式	186
7.2.4 使用引用类型	153	9.1 System.String 类	187
7.3 多维数组	154	9.1.1 构建字符串	188
7.4 锯齿数组	155	9.1.2 StringBuilder 成员	190
7.5 Array 类	156	9.2 字符串格式	190
7.5.1 创建数组	156	9.2.1 字符串插值	191
7.5.2 复制数组	156	9.2.2 日期时间和数字的格式	192
7.5.3 排序	157	9.2.3 自定义字符串格式	193
7.6 数组作为参数	159	9.3 正则表达式	194
7.7 数组协变	159	9.3.1 正则表达式概述	194
7.8 枚举	160	9.3.2 RegularExpressionsPlayground 示例	195
7.8.1 IEnumerator 接口	160	9.3.3 显示结果	197
7.8.2 foreach 语句	160	9.3.4 匹配、组和捕获	198
7.8.3 yield 语句	161	9.4 字符串和 Span	200
7.9 结构比较	164	9.5 小结	201
7.10 Span	165	第 10 章 集合	202
7.10.1 创建切片	166	10.1 概述	202
7.10.2 使用 Span 改变值	166	10.2 集合接口和类型	203
7.10.3 只读的 Span	167		

10.3	列表	203	12.2.5	排序	253
10.3.1	创建列表	204	12.2.6	分组	254
10.3.2	只读集合	210	12.2.7	LINQ 查询中的变量	255
10.4	队列	210	12.2.8	对嵌套的对象分组	255
10.5	栈	213	12.2.9	内连接	256
10.6	链表	214	12.2.10	左外连接	258
10.7	有序列表	217	12.2.11	组连接	260
10.8	字典	219	12.2.12	集合操作	262
10.8.1	字典初始化器	219	12.2.13	合并	263
10.8.2	键的类型	219	12.2.14	分区	264
10.8.3	字典示例	220	12.2.15	聚合操作符	264
10.8.4	Lookup 类	223	12.2.16	转换操作符	266
10.8.5	有序字典	223	12.2.17	生成操作符	267
10.9	集	224	12.3	并行 LINQ	267
10.10	性能	225	12.3.1	并行查询	268
10.11	小结	227	12.3.2	分区器	268
第 11 章	特殊的集合	228	12.3.3	取消	269
11.1	概述	228	12.4	表达式树	269
11.2	处理位	228	12.5	LINQ 提供程序	271
11.2.1	BitArray 类	229	12.6	小结	272
11.2.2	BitVector32 结构	230	第 13 章	C#函数式编程	273
11.3	可观察的集合	232	13.1	概述	273
11.4	不变的集合	233	13.1.1	避免状态突变	274
11.4.1	使用构建器和不变的集合	235	13.1.2	函数作为第一个类	275
11.4.2	不变集合类型和接口	235	13.2	表达式体的成员	275
11.4.3	使用 LINQ 和不变的数组	236	13.3	扩展方法	276
11.5	并发集合	236	13.4	using static 声明	277
11.5.1	创建管道	237	13.5	本地函数	278
11.5.2	使用 BlockingCollection	239	13.5.1	本地函数与 yield 语句	279
11.5.3	使用 ConcurrentDictionary	240	13.5.2	递归本地函数	281
11.5.4	完成管道	241	13.6	元组	282
11.6	小结	242	13.6.1	元组的声明和初始化	282
第 12 章	LINQ	243	13.6.2	元组解构	283
12.1	LINQ 概述	243	13.6.3	元组的返回	283
12.1.1	列表和实体	244	13.6.4	幕后的原理	284
12.1.2	LINQ 查询	246	13.6.5	ValueTuple 与元组的兼容性	285
12.1.3	扩展方法	246	13.6.6	推断出元组名称	285
12.1.4	推迟查询的执行	248	13.6.7	元组与链表	286
12.2	标准的查询操作符	249	13.6.8	元组和 LINQ	286
12.2.1	筛选	250	13.6.9	解构	287
12.2.2	用索引筛选	251	13.6.10	解构与扩展方法	288
12.2.3	类型筛选	252	13.7	模式匹配	288
12.2.4	复合的 from 子句	252	13.7.1	模式匹配与 is 运算符	288

13.7.2 模式匹配与 switch 语句	290	15.5.2 切换到 UI 线程	324
13.7.3 模式匹配与泛型	291	15.5.3 使用 IAsyncOperation	325
13.8 小结	291	15.5.4 避免阻塞情况	325
第 14 章 错误和异常	292	15.6 小结	325
14.1 简介	292	第 16 章 反射、元数据和动态编程	326
14.2 异常类	293	16.1 在运行期间检查代码和动态编程	326
14.3 捕获异常	294	16.2 自定义特性	327
14.3.1 异常和性能	296	16.2.1 编写自定义特性	327
14.3.2 实现多个 catch 块	296	16.2.2 自定义特性示例:	
14.3.3 在其他代码中捕获异常	299	WhatsNewAttributes	330
14.3.4 System.Exception 属性	299	16.3 反射	331
14.3.5 异常过滤器	299	16.3.1 System.Type 类	332
14.3.6 重新抛出异常	300	16.3.2 TypeView 示例	333
14.3.7 没有处理异常时发生的情况	303	16.3.3 Assembly 类	335
14.4 用户定义的异常类	303	16.3.4 完成 WhatsNewAttributes 示例	336
14.4.1 捕获用户定义的异常	304	16.4 为反射使用动态语言扩展	339
14.4.2 抛出用户定义的异常	305	16.4.1 创建 Calculator 库	339
14.4.3 定义用户定义的异常类	307	16.4.2 动态实例化类型	339
14.5 调用者信息	309	16.4.3 用 Reflection API 调用成员	340
14.6 小结	310	16.4.4 使用动态类型调用成员	340
第 15 章 异步编程	311	16.5 dynamic 类型	341
15.1 异步编程的重要性	311	16.6 DynamicObject 和 ExpandoObject	
15.2 异步编程的 .NET 历史	312	概述	344
15.2.1 同步调用	312	16.6.1 DynamicObject	344
15.2.2 异步模式	313	16.6.2 ExpandoObject	345
15.2.3 基于事件的异步模式	314	16.7 小结	347
15.2.4 基于任务的异步模式	314	第 17 章 托管和非托管内存	348
15.2.5 异步 Main() 方法	315	17.1 内存	348
15.3 异步编程的基础	315	17.2 后台内存管理	349
15.3.1 创建任务	316	17.2.1 值数据类型	349
15.3.2 调用异步方法	316	17.2.2 引用数据类型	350
15.3.3 使用 Awaiter	317	17.2.3 垃圾收集	352
15.3.4 延续任务	317	17.3 强引用和弱引用	354
15.3.5 同步上下文	318	17.4 处理非托管的资源	354
15.3.6 使用多个异步方法	318	17.4.1 析构函数或终结器	355
15.3.7 使用 ValueTasks	319	17.4.2 IDisposable 接口	356
15.3.8 转换异步模式	320	17.4.3 using 语句	356
15.4 错误处理	320	17.4.4 实现 IDisposable 接口和析构函数	357
15.4.1 异步方法的异常处理	321	17.4.5 IDisposable 和终结器的规则	358
15.4.2 多个异步方法的异常处理	321	17.5 不安全的代码	358
15.4.3 使用 AggregateException 信息	322	17.5.1 用指针直接访问内存	358
15.5 异步与 Windows 应用程序	322	17.5.2 指针示例: PointerPlayground	364
15.5.1 配置 await	323	17.5.3 使用指针优化性能	367

17.6	引用的语义	369
17.6.1	传递 ref 和返回 ref	371
17.6.2	ref 和数组	371
17.7	Span<T>	373
17.7.1	Span 引用托管堆	373
17.7.2	Span 引用栈	373
17.7.3	Span 引用本机堆	374
17.7.4	Span 扩展方法	374
17.8	平台调用	375
17.9	小结	378
第 18 章	Visual Studio 2017	379
18.1	使用 Visual Studio 2017	379
18.1.1	Visual Studio 的版本	382
18.1.2	Visual Studio 设置	382
18.2	创建项目	383
18.2.1	面向多个版本的.NET	384
18.2.2	选择项目类型	385
18.3	浏览并编写项目	388
18.3.1	Solution Explorer	388
18.3.2	使用代码编辑器	394
18.3.3	学习和理解其他窗口	399
18.3.4	排列窗口	402
18.4	构建项目	402
18.4.1	构建、编译和生成代码	403
18.4.2	调试版本和发布版本	403
18.4.3	选择配置	404
18.4.4	编辑配置	404
18.5	调试代码	406
18.5.1	设置断点	407
18.5.2	使用数据提示和调试器可视化 工具	407
18.5.3	Live Visual Tree	408
18.5.4	监视和修改变量	409
18.5.5	异常	409
18.5.6	多线程	410
18.6	重构工具	411
18.7	诊断工具	412
18.8	通过 Docker 创建和使用容器	415
18.8.1	Docker 简介	416
18.8.2	在 Docker 容器中运行 ASP.NET Core	416
18.8.3	创建 Dockerfile	417
18.8.4	使用 Visual Studio	418

18.9	小结	420
------	----	-----

第 II 部分 .NET Core 与 Windows Runtime

第 19 章	库、程序集、包和 NuGet	423
19.1	库的地狱	423
19.2	程序集	425
19.3	创建库	426
19.3.1	.NET 标准	427
19.3.2	创建.NET 标准库	428
19.3.3	解决方案文件	428
19.3.4	引用项目	429
19.3.5	引用 NuGet 包	429
19.3.6	NuGet 的来源	430
19.3.7	使用.NET Framework 库	431
19.4	使用共享项目	433
19.5	创建 NuGet 包	435
19.5.1	NuGet 包和命令行	435
19.5.2	支持多个平台	435
19.5.3	NuGet 包与 Visual Studio	436
19.6	小结	438
第 20 章	依赖注入	439
20.1	依赖注入的概念	439
20.1.1	使用没有依赖注入的服务	440
20.1.2	使用依赖注入	441
20.2	使用.NET Core DI 容器	442
20.3	服务的生命周期	443
20.3.1	使用单例和临时服务	445
20.3.2	使用 Scoped 服务	446
20.3.3	使用自定义工厂	448
20.4	使用选项初始化服务	449
20.5	使用配置文件	450
20.6	创建平台独立性	452
20.6.1	.NET 标准库	452
20.6.2	WPF 应用程序	453
20.6.3	UWP 应用程序	454
20.6.4	Xamarin 应用程序	455
20.7	使用其他 DI 容器	456
20.8	小结	457
第 21 章	任务和并行编程	458
21.1	概述	459
21.2	Parallel 类	460
21.2.1	使用 Parallel.For()方法循环	460
21.2.2	提前中断 Parallel.For	462

21.2.3	Parallel.For()方法的初始化	462	22.4	使用流	504
21.2.4	使用 Parallel.ForEach()方法循环	463	22.4.1	使用文件流	505
21.2.5	通过 Parallel.Invoke()方法调用 多个方法	464	22.4.2	读取流	508
21.3	任务	464	22.4.3	写入流	508
21.3.1	启动任务	464	22.4.4	复制流	509
21.3.2	Future——任务的结果	466	22.4.5	随机访问流	509
21.3.3	连续的任务	467	22.4.6	使用缓存的流	510
21.3.4	任务层次结构	468	22.5	使用读取器和写入器	511
21.3.5	从方法中返回任务	468	22.5.1	StreamReader 类	511
21.3.6	等待任务	468	22.5.2	StreamWriter 类	512
21.4	取消架构	470	22.5.3	读写二进制文件	512
21.4.1	Parallel.For()方法的取消	470	22.6	压缩文件	513
21.4.2	任务的取消	471	22.6.1	使用压缩流	514
21.5	数据流	472	22.6.2	使用 Brotli	514
21.5.1	使用动作块	472	22.6.3	压缩文件	515
21.5.2	源和目标数据块	473	22.7	观察文件的更改	515
21.5.3	连接块	474	22.8	使用内存映射的文件	516
21.6	Timer 类	475	22.8.1	使用访问器创建内存映射文件	517
21.7	线程问题	477	22.8.2	使用流创建内存映射文件	518
21.7.1	争用条件	477	22.9	使用管道通信	520
21.7.2	死锁	479	22.9.1	创建命名管道服务器	520
21.8	lock 语句和线程安全	480	22.9.2	创建命名管道客户端	521
21.9	Interlocked 类	483	22.9.3	创建匿名管道	522
21.10	Monitor 类	484	22.10	通过 Windows 运行库使用 文件和流	523
21.11	SpinLock 结构	485	22.10.1	Windows App 编辑器	523
21.12	WaitHandle 基类	485	22.10.2	把 Windows Runtime 类型映射为 .NET 类型	525
21.13	Mutex 类	485	22.11	小结	526
21.14	Semaphore 类	486	第 23 章	网络	527
21.15	Events 类	487	23.1	概述	527
21.16	Barrier 类	490	23.2	HttpClient 类	528
21.17	ReaderWriterLockSlim 类	492	23.2.1	发出异步的 Get 请求	528
21.18	Lock 和 await	494	23.2.2	抛出异常	529
21.19	小结	496	23.2.3	传递标题	529
第 22 章	文件和流	497	23.2.4	访问内容	531
22.1	概述	498	23.2.5	用 HttpResponseMessage 自定义请求	531
22.2	管理文件系统	498	23.2.6	使用 SendAsync 创建 HttpRequestMessage	532
22.2.1	检查驱动器信息	499	23.2.7	使用 HttpClient 和 Windows Runtime	532
22.2.2	使用 Path 类	500	23.3	使用 WebListener 类	534
22.2.3	创建文件和文件夹	500	23.4	使用实用工具类	536
22.2.4	访问和修改文件属性	501	23.4.1	URI	537
22.2.5	使用 File 执行读写操作	502			
22.3	枚举文件	503			

23.4.2	IPAddress	538	25.2.2	连接池	585
23.4.3	IPHostEntry	538	25.2.3	连接信息	585
23.4.4	Dns	539	25.3	命令	587
23.5	使用 TCP	540	25.3.1	ExecuteNonQuery()方法	587
23.5.1	使用 TCP 创建 HTTP 客户程序	540	25.3.2	ExecuteScalar()方法	588
23.5.2	创建 TCP 侦听器	541	25.3.3	ExecuteReader()方法	589
23.5.3	创建 TCP 客户端	547	25.3.4	调用存储过程	590
23.5.4	TCP 和 UDP	550	25.4	异步数据访问	591
23.6	使用 UDP	550	25.5	事务	592
23.6.1	建立 UDP 接收器	550	25.6	事务和 System.Transaction	595
23.6.2	创建 UDP 发送器	551	25.6.1	可提交的事务	597
23.6.3	使用多播	553	25.6.2	依赖事务	598
23.7	使用套接字	554	25.6.3	环境事务	599
23.7.1	使用套接字创建侦听器	554	25.6.4	嵌套作用域和环境事务	601
23.7.2	使用 NetworkStream 和套接字	556	25.7	小结	602
23.7.3	通过套接字使用读取器和写入器	557	第 26 章	Entity Framework Core	604
23.7.4	使用套接字实现接收器	557	26.1	Entity Framework 简史	605
23.8	小结	559	26.2	EF Core 简介	606
第 24 章	安全性	560	26.2.1	创建模型	607
24.1	概述	560	26.2.2	约定、注释和流利 API	607
24.2	验证用户信息	561	26.2.3	创建上下文	608
24.2.1	使用 Windows 标识	561	26.2.4	创建数据库	608
24.2.2	Windows Principal	562	26.2.5	删除数据库	609
24.2.3	使用声称	562	26.2.6	写入数据库	609
24.3	加密数据	564	26.2.7	读取数据库	610
24.3.1	创建和验证签名	565	26.2.8	更新记录	610
24.3.2	实现安全的数据交换	567	26.2.9	删除记录	611
24.3.3	使用 RSA 签名和散列	569	26.2.10	日志记录	611
24.4	保护数据	571	26.3	使用依赖注入	612
24.4.1	实现数据保护	571	26.4	创建模型	614
24.4.2	用户机密	573	26.4.1	创建关系	614
24.5	资源的访问控制	575	26.4.2	数据注释	614
24.6	Web 安全性	577	26.4.3	流利 API	615
24.6.1	编码	577	26.4.4	自包含类型的配置	616
24.6.2	SQL 注入	579	26.4.5	在数据库中搭建模型	617
24.6.3	跨站点请求伪造	580	26.4.6	映射到字段	618
24.7	小结	581	26.4.7	阴影属性	619
第 25 章	ADO.NET 和事务	582	26.5	查询	621
25.1	ADO.NET 概述	582	26.5.1	基本查询	621
25.1.1	示例数据库	583	26.5.2	客户端和服务端求值	622
25.1.2	NuGet 包和名称空间	583	26.5.3	原始 SQL 查询	623
25.2	使用数据库连接	584	26.5.4	已编译查询	624
25.2.1	管理连接字符串	585	26.5.5	全局查询过滤器	624

26.5.6	EF.Functions	625	27.3.2	使用资源文件生成器	665
26.6	关系	625	27.3.3	通过 ResourceManager 使用 资源文件	666
26.6.1	使用约定的关系	625	27.3.4	System.Resources 名称空间	666
26.6.2	显式加载相关数据	627	27.4	使用 ASP.NET Core 本地化	667
26.6.3	即时加载相关数据	628	27.4.1	注册本地化服务	667
26.6.4	使用注释的关系	628	27.4.2	注入本地化服务	668
26.6.5	使用流利 API 的关系	629	27.4.3	区域性提供程序	668
26.6.6	根据约定的每个层次结构的表	630	27.4.4	在 ASP.NET Core 中使用资源	669
26.6.7	使用流利 API 的每个层次 结构中的表	632	27.4.5	使用控制器和视图进行本地化	670
26.6.8	表的拆分	633	27.5	本地化 UWP	674
26.6.9	拥有的实体	634	27.5.1	给 UWP 使用资源	674
26.7	保存数据	636	27.5.2	使用多语言应用程序工具集 进行本地化	675
26.7.1	用关系添加对象	636	27.6	小结	677
26.7.2	对象的跟踪	637	第 28 章	测试	678
26.7.3	更新对象	638	28.1	概述	678
26.7.4	更新未跟踪的对象	638	28.2	使用 MSTest 进行单元测试	679
26.7.5	批处理	639	28.2.1	使用 MSTest 创建单元测试	679
26.8	冲突的处理	640	28.2.2	运行单元测试	681
26.8.1	最后一个更改获胜	640	28.2.3	使用 MSTest 预期异常	682
26.8.2	第一个更改获胜	641	28.2.4	测试全部代码路径	683
26.9	上下文池	644	28.2.5	外部依赖	683
26.10	使用事务	644	28.3	使用 xUnit 进行单元测试	685
26.10.1	使用隐式的事务	644	28.3.1	使用 xUnit 和 .NET Core	686
26.10.2	创建显式的事务	646	28.3.2	创建 Fact 属性	686
26.11	迁移	647	28.3.3	创建 Theory 特性	687
26.11.1	准备项目文件	647	28.3.4	使用 Mocking 库	688
26.11.2	利用 ASP.NET Core MVC 托管 应用程序	648	28.4	实时单元测试	690
26.11.3	托管 .NET Core 控制台应用程序	648	28.5	使用 EF Core 进行单元测试	692
26.11.4	创建迁移	649	28.6	使用 Windows 应用程序进行 UI 测试	693
26.11.5	以编程方式应用迁移	651	28.7	Web 集成、负载和性能测试	697
26.11.6	应用迁移的其他方法	652	28.7.1	ASP.NET Core 集成测试	697
26.12	小结	652	28.7.2	创建 Web 测试	698
第 27 章	本地化	653	28.7.3	运行 Web 测试	700
27.1	全球市场	653	28.8	小结	702
27.2	System.Globalization 名称空间	654	第 29 章	跟踪、日志和分析	703
27.2.1	Unicode 问题	654	29.1	诊断概述	703
27.2.2	区域性和区域	655	29.2	使用 EventSource 跟踪	704
27.2.3	使用区域性	658	29.2.1	EventSource 的简单用法	705
27.2.4	排序	663	29.2.2	跟踪工具	706
27.3	资源	664	29.2.3	派生自 EventSource	707
27.3.1	资源读取器和写入器	664			

29.2.4	使用注释和 EventSource	709
29.2.5	创建事件清单模式	710
29.2.6	使用活动 ID	712
29.3	创建自定义侦听器	714
29.4	使用 ILogger 接口编写日志	715
29.4.1	配置提供程序	716
29.4.2	使用作用域	717
29.4.3	过滤	718
29.4.4	配置日志记录	718
29.4.5	使用没有依赖注入的 ILogger	719
29.5	使用 Visual Studio App Center 进行分析	720
29.6	小结	722

第 III 部分 Web 应用程序和服务

第 30 章	ASP.NET Core	727
30.1	概述	727
30.2	Web 技术	728
30.2.1	HTML	728
30.2.2	CSS	729
30.2.3	JavaScript 和 TypeScript	729
30.2.4	脚本库	729
30.3	ASP.NET Web 项目	730
30.3.1	启动	733
30.3.2	示例应用程序	735
30.4	添加客户端内容	736
30.4.1	为客户端内容使用工具	737
30.4.2	通过 Bower 使用客户端库	738
30.4.3	使用 JavaScript 包管理器 npm	739
30.4.4	捆绑	739
30.4.5	用 webpack 打包	740
30.5	请求和响应	741
30.5.1	请求标题	742
30.5.2	查询字符串	744
30.5.3	编码	745
30.5.4	表单数据	745
30.5.5	cookie	746
30.5.6	发送 JSON	747
30.6	依赖注入	747
30.6.1	定义服务	748
30.6.2	注册服务	748
30.6.3	注入服务	748
30.6.4	调用控制器	749
30.7	简单的路由	749

30.8	创建自定义的中间件	750
30.9	会话状态	751
30.10	用 ASP.NET Core 配置	752
30.10.1	读取配置	753
30.10.2	修改配置提供程序	755
30.10.3	基于环境的不同配置	756
30.11	小结	757

第 31 章 ASP.NET Core MVC

31.1	为 ASP.NET Core MVC 建立服务	758
31.2	定义路由	760
31.2.1	添加路由	760
31.2.2	使用路由约束	761
31.3	创建控制器	761
31.3.1	理解动作方法	762
31.3.2	使用参数	762
31.3.3	返回数据	762
31.3.4	使用 Controller 基类和 POCO 控制器	763
31.4	创建视图	765
31.4.1	向视图传递数据	765
31.4.2	Razor 语法	766
31.4.3	创建强类型视图	766
31.4.4	定义布局	767
31.4.5	用部分视图定义内容	770
31.4.6	使用视图组件	773
31.4.7	在视图中使用依赖注入	774
31.4.8	为多个视图导入名称空间	775
31.5	从客户端提交数据	775
31.5.1	模型绑定器	777
31.5.2	注解和验证	778
31.6	使用 HTML Helper	779
31.6.1	简单的 Helper	779
31.6.2	使用模型数据	779
31.6.3	定义 HTML 特性	780
31.6.4	创建列表	780
31.6.5	强类型化的 Helper	781
31.6.6	编辑器扩展	782
31.6.7	实现模板	782
31.7	Tag Helper	783
31.7.1	激活 Tag Helper	783
31.7.2	使用锚定 Tag Helper	783
31.7.3	使用 Label Tag Helper	784
31.7.4	使用 Input Tag Helper	785

32.5.1	使用 EF Core	835
32.5.2	创建数据访问服务	836
32.6	用 OpenAPI 或 Swagger 创建元数据	837
32.7	创建和使用 OData 服务	841
32.7.1	创建数据模型	842
32.7.2	创建数据库	843
32.7.3	OData 启动代码	844
32.7.4	创建 OData 控制器	844
32.7.5	OData 查询	845
32.8	使用 Azure Function	847
32.8.1	创建 Azure Function	847
32.8.2	使用依赖注入容器	848
32.8.3	实现 GET、POST 和 PUT 请求	849
32.8.4	运行 Azure Function	851
32.9	小结	852
第 IV 部分 应用程序		
33 章	Windows 应用程序	855
33.1	Windows 应用程序简介	855
33.1.1	Windows 运行库	856
33.1.2	Hello, Windows	856
33.1.3	应用程序清单文件	857
33.1.4	应用程序启动	859
33.1.5	主页	859
33.2	XAML	861
33.2.1	XAML 标准	861
33.2.2	将元素映射到类	861
33.2.3	通过 XAML 使用定制的.NET 类	862
33.2.4	将属性用作特性	863
33.2.5	将属性用作元素	863
33.2.6	依赖属性	864
33.2.7	创建依赖属性	864
33.2.8	值变更回调和事件	865
33.2.9	路由事件	866
33.2.10	附加属性	867
33.2.11	标记扩展	868
33.2.12	自定义标记扩展	869
33.2.13	条件 XAML	870
33.3	控件	871
33.3.1	框架派生的 UI 元素	872
33.3.2	控件派生的控件	875
33.3.3	范围控件	881
33.3.4	内容控件	882
33.3.5	按钮	883

第 IV 部分 应用程序

第 33 章	Windows 应用程序	855
33.1	Windows 应用程序简介	855
33.1.1	Windows 运行库	856
33.1.2	Hello, Windows	856
33.1.3	应用程序清单文件	857
33.1.4	应用程序启动	859
33.1.5	主页	859
33.2	XAML	861
33.2.1	XAML 标准	861
33.2.2	将元素映射到类	861
33.2.3	通过 XAML 使用定制的.NET 类	862
33.2.4	将属性用作特性	863
33.2.5	将属性用作元素	863
33.2.6	依赖属性	864
33.2.7	创建依赖属性	864
33.2.8	值变更回调和事件	865
33.2.9	路由事件	866
33.2.10	附加属性	867
33.2.11	标记扩展	868
33.2.12	自定义标记扩展	869
33.2.13	条件 XAML	870
33.3	控件	871
33.3.1	框架派生的 UI 元素	872
33.3.2	控件派生的控件	875
33.3.3	范围控件	881
33.3.4	内容控件	882
33.3.5	按钮	883

33.3.6	项控件	884	34.7.1	使用 IEditableObject	923
33.3.7	Flyout 控件	884	34.7.2	视图模型的具体实现	924
33.4	数据绑定	884	34.7.3	命令	925
33.4.1	用 INotifyPropertyChanged 更改通知	885	34.7.4	服务、ViewModel 和依赖注入	926
33.4.2	创建图书列表	886	34.8	视图	927
33.4.3	列表绑定	887	34.8.1	从视图模型中打开对话框	930
33.4.4	把事件绑定到方法	887	34.8.2	页面之间的导航	931
33.4.5	使用数据模板和数据模板选择器	888	34.8.3	自适应用户界面	933
33.4.6	绑定简单对象	890	34.8.4	显示进度信息	935
33.4.7	值的转换	891	34.8.5	使用列表项中的操作	936
33.5	导航	892	34.9	使用事件传递消息	938
33.5.1	导航回最初的页面	892	34.10	使用框架	939
33.5.2	重写 Page 类的导航	893	34.11	小结	940
33.5.3	在页面之间导航	894	第 35 章	样式化 Windows 应用程序	941
33.5.4	后退按钮	895	35.1	样式设置	941
33.5.5	Hub	896	35.2	形状	942
33.5.6	Pivot	898	35.3	几何图形	944
33.5.7	NavigationView	899	35.3.1	使用段的几何图形	944
33.6	布局	902	35.3.2	使用 PathMarkup 的几何图形	945
33.6.1	StackPanel	902	35.4	变换	945
33.6.2	Canvas	903	35.4.1	缩放	945
33.6.3	Grid	903	35.4.2	平移	946
33.6.4	VariableSizedWrapGrid	904	35.4.3	旋转	946
33.6.5	RelativePanel	906	35.4.4	倾斜	946
33.6.6	自适应触发器	906	35.4.5	组合变换和复合变换	946
33.6.7	XAML 视图	909	35.4.6	使用矩阵的变换	947
33.6.8	延迟加载	909	35.5	画笔	947
33.7	小结	910	35.5.1	SolidColorBrush	947
第 34 章	模式和 XAML 应用程序	911	35.5.2	LinearGradientBrush	947
34.1	使用 MVVM 的原因	911	35.5.3	ImageBrush	948
34.2	定义 MVVM 模式	912	35.5.4	AcrylicBrush	948
34.3	共享代码	913	35.5.5	RevealBrush	949
34.3.1	使用 API 协定和通用 Windows 平台	913	35.6	样式和资源	949
34.3.2	使用共享项目	915	35.6.1	样式	949
34.3.3	使用 .NET 标准库	916	35.6.2	资源	951
34.4	示例解决方案	917	35.6.3	从代码中访问资源	952
34.5	模型	918	35.6.4	资源字典	952
34.5.1	实现变更通知	918	35.6.5	主题资源	953
34.5.2	使用 Repository 模式	919	35.7	模板	954
34.6	服务	920	35.7.1	控件模板	955
34.7	视图模型	921	35.7.2	数据模板	958
			35.7.3	样式化 ListView	959
			35.7.4	ListView 项的数据模板	960

35.7.5	项容器的样式	960	36.9	自动建议	1011
35.7.6	项面板	961	36.10	小结	1013
35.7.7	列表视图的控件模板	961	第 37 章 Xamarin.Forms		1015
35.8	动画	962	37.1	Xamarin 开发入门	1015
35.8.1	时间轴	962	37.1.1	用 Android 架构 Xamarin	1016
35.8.2	缓动函数	964	37.1.2	用 iOS 架构 Xamarin	1016
35.8.3	关键帧动画	968	37.1.3	Xamarin.Forms	1017
35.8.4	过渡	969	37.2	Xamarin 开发工具	1018
35.9	可视化状态管理器	971	37.2.1	Android	1018
35.9.1	用控件模板预定义状态	972	37.2.2	iOS	1019
35.9.2	定义自定义状态	973	37.2.3	Visual Studio 2017	1019
35.9.3	设置自定义的状态	973	37.2.4	Visual Studio for Mac	1019
35.10	小结	974	37.2.5	Visual Studio App Center	1020
第 36 章 高级 Windows 应用程序		975	37.3	Android 基础	1020
36.1	概述	975	37.3.1	活动	1021
36.2	应用程序的生命周期	976	37.3.2	资源	1022
36.2.1	应用程序的执行状态	976	37.3.3	显示列表	1022
36.2.2	在页面之间导航	976	37.3.4	显示消息	1024
36.3	导航状态	978	37.4	iOS 基础	1025
36.3.1	暂停应用程序	979	37.4.1	iOS 应用程序结构	1025
36.3.2	激活暂停的应用程序	980	37.4.2	故事板	1026
36.3.3	测试暂停	980	37.4.3	控制器	1028
36.3.4	页面状态	981	37.4.4	显示消息	1028
36.4	共享数据	983	37.5	Xamarin.Forms 应用程序	1029
36.4.1	共享源	983	37.5.1	托管 Xamarin 的 Windows 应用程序	1029
36.4.2	共享目标	986	37.5.2	托管 Xamarin 的 Android	1030
36.5	应用程序服务	991	37.5.3	托管 Xamarin 的 iOS	1031
36.5.1	创建模型	991	37.5.4	共享的项目	1031
36.5.2	为应用程序服务连接创建后台任务	992	37.6	使用公共库	1032
36.5.3	注册应用程序服务	993	37.7	控件层次结构	1032
36.5.4	调用应用程序服务	994	37.8	页面	1033
36.6	高级的编译绑定	996	37.9	导航	1034
36.6.1	已编译数据绑定的生命周期	996	37.10	布局	1035
36.6.2	绑定到方法上	997	37.11	视图	1037
36.6.3	用 x:Bind 分阶段	998	37.12	数据绑定	1037
36.7	使用文本	1002	37.13	命令	1038
36.7.1	使用字体	1002	37.14	ListView 和 ViewCell	1038
36.7.2	内联和块元素	1004	37.15	小结	1039
36.7.3	使用溢出区域	1005			
36.8	上墨	1008			

附赠章节电子版(请扫描封底二维码获取)

第 1 章 Composition	1
BC1.1 概述	1
BC1.2 Composition 库的体系结构	2
BC1.2.1 使用特性的 Composition	3
BC1.2.2 基于约定的部件注册	8
BC1.3 定义协定	10
BC1.4 导出部件	13
BC1.4.1 创建部件	13
BC1.4.2 使用部件的部件	17
BC1.4.3 导出元数据	17
BC1.4.4 使用元数据进行惰性加载	19
BC1.5 导入部件	19
BC1.5.1 导入连接	22
BC1.5.2 部件的惰性加载	23
BC1.5.3 读取元数据	23
BC1.6 小结	25
第 2 章 XML 和 JSON	26
BC2.1 数据格式	26
BC2.1.1 XML	27
BC2.1.2 .NET 支持的 XML 标准	28
BC2.1.3 在框架中使用 XML	28
BC2.1.4 JSON	29
BC2.2 读写流格式的 XML	30
BC2.2.1 使用 XmlReader 类读取 XML	31
BC2.2.2 使用 XmlWriter 类	33
BC2.3 在 .NET 中使用 DOM	34
BC2.3.1 使用 XmlDocument 类读取	35
BC2.3.2 遍历层次结构	35
BC2.3.3 使用 XmlDocument 插入节点	36
BC2.4 使用 XPathNavigator 类	37
BC2.4.1 XPathDocument 类	37
BC2.4.2 XPathNavigator 类	37
BC2.4.3 XPathNodeIterator 类	38
BC2.4.4 使用 XPath 导航 XML	38
BC2.4.5 使用 XPath 评估	39
BC2.4.6 用 XPath 修改 XML	39
BC2.5 在 XML 中序列化对象	40
BC2.5.1 序列化简单对象	40
BC2.5.2 序列化一个对象树	42
BC2.5.3 没有特性的序列化	44

BC2.6 LINQ to XML	46
BC2.6.1 XDocument 对象	46
BC2.6.2 XElement 对象	47
BC2.6.3 XNamespace 对象	47
BC2.6.4 XComment 对象	49
BC2.6.5 XAttribute 对象	49
BC2.6.6 使用 LINQ 查询 XML 文档	50
BC2.6.7 查询动态的 XML 文档	50
BC2.6.8 转换为对象	52
BC2.6.9 转换为 XML	52
BC2.7 JSON	53
BC2.7.1 创建 JSON	53
BC2.7.2 转换对象	54
BC2.7.3 序列化对象	55
BC2.7.4 遍历 JSON 节点	55
BC2.8 小结	56
第 3 章 WebHooks 和 SignalR	57
BC3.1 概述	57
BC3.2 WebSockets	58
BC3.2.1 WebSockets 服务器	58
BC3.2.2 WebSockets 客户端	60
BC3.3 使用 SignalR 的简单聊天程序	62
BC3.3.1 创建集线器	62
BC3.3.2 用 HTML 和 JavaScript 创建客户端	63
BC3.3.3 创建 SignalR .NET 客户端	65
BC3.4 分组连接	68
BC3.4.1 用分组扩展集线器	68
BC3.4.2 用分组扩展 Windows 客户端	69
BC3.5 WebHooks 的体系结构	71
BC3.6 创建 Dropbox 和 GitHub 接收器	72
BC3.6.1 创建 Web 应用程序	73
BC3.6.2 为 Dropbox 和 GitHub 配置 WebHooks	73
BC3.6.3 实现处理程序	73
BC3.6.4 用 Dropbox 和 GitHub 配置应用程序	76
BC3.6.5 运行应用程序	77
BC3.7 小结	77

第 4 章 机器人和认知服务.....	79	BC5.2 相机.....	98
BC4.1 机器人的定义.....	79	BC5.3 Geolocation 和 MapControl	99
BC4.2 创建对话框机器人.....	80	BC5.3.1 使用 MapControl.....	99
BC4.2.1 配置状态服务	81	BC5.3.2 使用 Geolocator 定位信息	102
BC4.2.2 接收机器人消息	82	BC5.3.3 街景地图.....	103
BC4.2.3 定义对话框.....	83	BC5.3.4 继续请求位置信息.....	104
BC4.2.4 使用 PromptDialog	85	BC5.4 传感器.....	105
BC4.3 为对话框使用 Form Flow	88	BC5.4.1 光线.....	106
BC4.4 创建英雄卡	89	BC5.4.2 罗盘	107
BC4.5 机器人和 LUIS.....	91	BC5.4.3 加速计.....	107
BC4.5.1 定义意图和话语	92	BC5.4.4 倾斜计.....	108
BC4.5.2 访问 LUIS 中的建议	95	BC5.4.5 陀螺仪.....	109
BC4.5.3 使用带有活动检查的 Form Flow	96	BC5.4.6 方向	109
BC4.6 小结	96	BC5.4.7 Rolling Marble 示例.....	110
第 5 章 Windows 应用程序的更多特性	97	BC5.5 小结.....	112
BC5.1 概述	97		

第 I 部分

C# 语言

- 第 1 章 .NET 应用程序和工具
- 第 2 章 核心 C#
- 第 3 章 对象和类型
- 第 4 章 继承
- 第 5 章 泛型
- 第 6 章 运算符和类型强制转换
- 第 7 章 数组
- 第 8 章 委托、lambda 表达式和事件
- 第 9 章 字符串和正则表达式
- 第 10 章 集合
- 第 11 章 特殊的集合
- 第 12 章 LINQ
- 第 13 章 C#函数式编程
- 第 14 章 错误和异常

- 第 15 章 异步编程
- 第 16 章 反射、元数据和动态编程
- 第 17 章 托管和非托管的资源
- 第 18 章 Visual Studio 2017

第 1 章

.NET 应用程序和工具

本章要点

- 回顾 .NET 的历史
- 理解 .NET Framework 和 .NET Core 之间的差异
- NuGet 包
- 公共语言运行库
- Windows 运行库的特性
- 编写 “Hello World!” 程序
- .NET Core 命令行界面
- Visual Studio 2017
- 通用 Windows 平台
- 创建 Windows 应用程序的技术
- 创建 Web 应用程序的技术

本章源代码下载：

单击 www.wrox.com 的 Download Code 选项卡可下载本章源代码。源代码也可以在 HelloWorld 目录的 <https://github.com/ProfessionalCSharp/ProfessionalCSharp7> 中找到。

本章代码分为以下几个主要的示例文件：

- HelloWorld
- WebApp
- SelfContained HelloWorld

1.1 选择技术

.NET 是在 Windows 平台上创建应用程序的杰出技术。但现在，.NET 是在 Windows、Linux 和 Mac 上创建应用程序的杰出技术。

.NET Core 是 .NET 自其发明以来最大的变化。目前 .NET 代码是开源的代码，还可以为其他平台创建应用程序。.NET 使用现代模式。.NET Core 和 NuGet 包允许微软公司以更短的更新周期提供新特性。应该使用什么

技术创建应用程序并不容易决定，本章将提供这方面的帮助。其中包含用于创建 Windows、Web 应用程序和服务的不同技术的信息，指导选择什么技术进行数据库访问，凸显了 .NET Framework 和 .NET Core 之间的差异。

1.2 回顾.NET 历史

要更好地理解 .NET 和 C# 的可用功能，最好先了解它的历史。表 1-1 显示了 .NET Framework 的版本、对应的公共语言运行库(Common Language Runtime, CLR)的版本、C# 的版本以及 Visual Studio 的版本，并指出相应版本的发布年份。除了知道使用什么技术之外，最好也知道不推荐使用什么技术，因为这些技术会被替代。

表 1-1

.NET Framework	CLR	C#	Visual Studio
1.0	1.0	1.0	2002
1.1	1.1	1.2	2003
2.0	2.0	2.0	2005
3.0	2.0	2.0	2005+扩展版
3.5	2.0	3.0	2008
4.0	4.0	4.0	2010
4.5	4.0	5.0	2012
4.5.1	4.0	5.0	2013
4.6	4.0	6	2015
4.7	4.0	7	2017

当使用 .NET Core 创建应用程序时，了解支持级别的时间框架非常重要。LTS(Long Time Support, 长时间支持)的支持长度比 Current 长，但 Current 会更快地获得新特性。LTS 在发行后或下一个 LTS 版本发布的 12 个月得到 3 年的支持，以较短的版本为准。因此，如果下一个 LTS 版本在 2018 年 6 月 27 日之前没有发布，那么 .NET Core 1.0 将支持到 2019 年 6 月 27 日。如果下一个 LTS 版本在更早的时候发布，那么在下一个 LTS 发布后的一年，.NET Core 1.0 将得到支持。

.NET Core 1.1 最初是一个 Current 版本，但它变成了 LTS，它得到的支持长度与 .NET Core 1.0 相同。

.NET Core 2.0 是一个 Current 支持级别的版本。这意味着它将得到 3 年的支持，下一个 LTS 发布后的 12 个月，或发布下一个 Current 版本后的 3 个月——以较短的时间为准。可以假定，最后一个选项是视情况而定，.NET Core 2.0 将在 .NET Core 2.1 可用后 3 个月得到支持。

表 1-2 列出了 .NET Core 版本、发布日期和支持级别。

表 1-2

.NET Core 版本	发布日期	支持级别
1.0	2016 年 6 月 27 日	LTS
1.1	2016 年 11 月 16 日	LTS*
2.0	2017 年 8 月 14 日	Current

下面各小节详细介绍这两个表，以及 C# 和 .NET 的发展。

1.2.1 C# 1.0 —— 一种新语言

C# 1.0 是一种全新的编程语言，用于 .NET Framework。开发它时，.NET Framework 由大约 3000 个类和 CLR 组成。

创建 Java 的 Sun 公司申请法庭判决不允许微软公司更改 Java 代码后, Anders Hejlsberg 设计了 C#。Hejlsberg 为微软公司工作之前, 在 Borland 公司设计了 Delphi 编程语言(一种 Object Pascal 语言)。Hejlsberg 在微软公司负责 J++(Java 编程语言的微软版本)。鉴于 Hejlsberg 的背景, C#编程语言主要受到 C++、Java 和 Pascal 的影响。

因为 C#的创建晚于 Java 和 C++, 所以微软公司分析了其他语言中典型的编程错误, 完成了一些不同的工作来避免这些错误。这些不同的工作包括:

- 在 if 语句中, 布尔(Boolean)表达式是必需的(C++也允许在这里使用整数值)。
- 允许使用 struct 和 class 关键字创建值类型和引用类型(Java 只允许创建自定义引用类型; 在 C++中, struct 和 class 之间的区别只是访问修饰符的默认值不同)。
- 允许使用虚拟方法和非虚拟方法 (这类似于 C++; Java 总是创建虚拟方法)。

当然, 阅读本书, 会看到更多的变化。

现在, C#是一种纯粹的面向对象编程语言, 具备继承、封装和多态性等特性。C#也提供了基于组件的编程改进, 如委托和事件。

在.NET 和 CLR 推出之前, 每种编程语言都有自己的运行库。在 C++中, C++运行库与每个 C++程序链接起来。Visual Basic 6 有自己的运行库 VBRun。Java 的运行库是 Java 虚拟机(Java Virtual Machine, JVC)——可以与 CLR 相媲美。CLR 是每种.NET 编程语言都使用的运行库。推出 CLR 时, 微软公司提供了 JScript .NET、Visual Basic .NET、Managed C++ 和 C#。JScript .NET 是微软公司的 JavaScript 编译器, 与 CLR 和.NET 类一起使用。Visual Basic .NET 是提供.NET 支持的 Visual Basic, 现在再次简称为 Visual Basic。Managed C++是混合了本地 C++代码与 Managed .NET 代码的语言。今天与.NET 一起使用的新 C++语言是 C++/ CLR。

.NET 编程语言的编译器生成中间语言(Intermediate Language, IL)代码。IL 代码看起来像面向对象的机器码, 使用工具 ildasm.exe 可以打开包含.NET 代码的 DLL 或 EXE 文件来检查 IL 代码。CLR 包含一个即时(Just-In-Time, JIT)编译器, 当程序开始运行时, JIT 编译器会从 IL 代码中生成本地代码。

注意:

IL 代码也称为托管代码。

CLR 的其他部分是垃圾收集器(GC)、调试器扩展和线程实用工具。垃圾收集器负责清理不再引用的托管内存, 这个安全机制使用代码访问安全性来验证允许代码做什么。调试器扩展允许在不同的编程语言之间启动调试会话 (例如, 在 Visual Basic 中启动调试会话, 在 C#库内继续调试)。线程实用工具负责在底层平台上创建线程。

.NET Framework 的第 1 版已经很大了。类在名称空间内组织, 以便于导航可用的 3000 个类。使用名称空间组织类, 允许在不同的名称空间中有相同的类名, 以解决冲突。.NET Framework 的第 1 版允许使用 Windows Forms(名称空间 System.Windows.Forms)创建 Windows 桌面应用程序, 使用 ASP.NET Web Forms (System.Web) 创建 Web 应用程序, 使用 ASP.NET Web Services 与应用程序和 Web 服务通信, 使用.NET Remoting 在.NET 应用程序之间更迅速地通信, 使用 Enterprise Services 创建运行在应用程序服务器上的 COM +组件。

ASP.NET Web Forms 是创建 Web 应用程序的技术, 其目标是开发人员不需要了解 HTML 和 JavaScript。服务器端控件会创建 HTML 和 JavaScript, 这些控件的工作方式类似于 Windows Forms 本身。

C# 1.2 和.NET 1.1 主要是错误修复版本, 改进较小。

注意:

继承在第 4 章中讨论, 委托和事件在第 8 章中讨论。

注意:

.NET 的每个新版本都有 Professional C#图书的新版本。对于.NET 1.0, 这本书已经是第 2 版了, 因为第 1 版是以.NET 1.0 的 Beta 2 为基础出版的。目前, 本书是第 11 版。

1.2.2 带有泛型的 C# 2 和 .NET 2

C# 2 和 .NET 2 是一个巨大的更新。在这个版本中，改变了 C# 编程语言，建立了 IL 代码，所以需要新的 CLR 来支持 IL 代码的增加。一个大的变化是泛型。泛型允许创建类型，而不需要知道使用什么内部类型。所使用的内部类型在实例化(即创建实例)时定义。

C# 编程语言中的这个改进也导致了 Framework 中多了许多新类型，例如 `System.Collections.Generic` 名称空间中新的泛型集合类。有了这个类，1.0 版本定义的旧集合类就很少用在新应用程序中了。当然，旧类现在仍然在工作，甚至在新的 .NET Core 版本中也是如此。

注意：

本书一直在使用泛型，详见第 5 章。第 10 章介绍了泛型集合类。

1.2.3 .NET 3.0 —— Windows Presentation Foundation

发布 .NET 3.0 时，不需要新版本的 C#。3.0 版本只提供了新的库，但它发布了大量新的类型和名称空间。Windows Presentation Foundation(WPF)可能是新框架最大的一部分，用于创建 Windows 桌面应用程序。Windows Forms 包括本地 Windows 控件，且基于像素；而 WPF 基于 DirectX，独立绘制每个控件。WPF 中的矢量图形允许无缝地调整任何窗体的大小。WPF 中的模板还允许完全自定义外观。例如，用于苏黎世机场的应用程序可以包含看起来像一架飞机的按钮。因此，应用程序的外观可以与之前开发的传统 Windows 应用程序不同。`System.Windows` 名称空间下的所有内容都属于 WPF，但 `System.Windows.Forms` 除外。有了 WPF，用户界面可以使用 XML 语法 XAML(XML for Applications Markup Language)设计。

.NET 3.0 推出之前，ASP.NET Web Services 和 .NET Remoting 用于应用程序之间的通信。Message Queuing 是用于通信的另一个选择。各种技术有不同的优点和缺点，它们都用不同的 API 进行编程。典型的企业应用程序必须使用一个以上的通信 API，因此必须学习其中的几项技术。WCF(Windows Communication Foundation)解决了这个问题。WCF 把其他 API 的所有选项结合到一个 API 中。然而，为了支持 WCF 提供的所有功能，需要配置 WCF。

.NET 3.0 版本的第三大部分是 Windows WF(Workflow Foundation)和名称空间 `System.Workflow`。微软公司不是为几个不同的应用程序创建自定义的工作流引擎(微软公司本身为不同的产品创建了几个工作流引擎)，而是把工作流引擎用作 .NET 的一部分。

有了 .NET 3.0，Framework 的类从 .NET 2.0 的 8 000 个增加到约 12 000 个。

注意：

要学习 WPF 和 WCF，需要阅读本书的前一版本《C# 高级编程(第 10 版) C# 6 & .NET Core 1.0》。

1.2.4 C# 3 和 .NET 3.5——LINQ

.NET 3.5 和新版本 C# 3 一起发布。主要改进是使用 C# 定义的查询语法，它允许使用相同的语法来过滤和排序对象列表、XML 文件和数据库。语言增强不需要对 IL 代码进行任何改变，因为这里使用的 C# 特性只是语法糖。所有的增强也可以用旧的语法实现，只是需要编写更多的代码。C# 语言很容易进行这些查询。有了 LINQ 和 lambda 表达式，就可以使用相同的查询语法来访问对象集合、数据库和 XML 文件。

为了访问数据库并创建 LINQ 查询，LINQ to SQL 发布为 .NET 3.5 的一部分。在 .NET 3.5 的第一个更新中，发布了 Entity Framework 的第一个版本。LINQ to SQL 和 Entity Framework 都提供了从层次结构到数据库关系的映射和 LINQ 提供程序。Entity Framework 更强大，但 LINQ to SQL 更简单。随着时间的推移，LINQ to SQL 的特性在 Entity Framework 中实现了，并且 Entity Framework 会一直保留这些特性。Entity Framework 的新版本 Entity Framework Core (EF Core) 看起来与第一版非常不同。

另一种引入为.NET 3.5 一部分的技术是 `System.AddIn` 名称空间，它提供了插件模型。这个模型提供了甚至在过程外部运行插件的强大功能，但它使用起来也很复杂。

注意：

LINQ 详见第 12 章，Entity Framework 的最新版本与.NET 3.5 版本有很大差别，参见第 26 章。

1.2.5 C# 4 和.NET 4.0——dynamic 和 TPL

C# 4 的主题是动态集成脚本语言，使其更容易使用 COM 集成。C#语法扩展为使用 `dynamic` 关键字、命名参数和可选参数，以及用泛型增强的协变和逆变。

其他改进在.NET Framework 中进行。有了多核 CPU，并行编程就变得越来越重要。任务并行库(Task Parallel Library, TPL)使用 `Task` 类和 `Parallel` 类抽象出线程，更容易创建并行运行的代码。

因为用.NET 3.0 创建的工作流引擎没有履行自己的诺言，所以全新的 Windows Workflow Foundation 成为.NET 4.0 的一部分。为了避免与旧工作流引擎冲突，新的工作流引擎是在 `System.Activity` 名称空间中定义的。

C# 4 的增强还需要一个新版本的运行库。运行库从版本 2 跳到版本 4。

发布 Visual Studio 2010 时，附带了一项创建 Web 应用程序的新技术：ASP.NET MVC 2.0。与 ASP.NET Web Forms 不同，该技术关注于模型-视图-控制器(MVC)模式，该模式由项目结构强制执行。这项技术也关注于编程 HTML 和 JavaScript。HTML 和 JavaScript 通过 HTML 5 的发布在开发者社区中获得了巨大的推动。由于这一技术非常新颖，而且是到 Visual Studio 的带外(Out Of Band, OOB)，因此 ASP.NET MVC 是定期更新的。

注意：

C# 4 的 `dynamic` 关键字参见第 16 章。任务并行库参见第 21 章。ASP.NET 的新一代 ASP.NET Core 参见第 30 章，ASP.NET Core MVC 参见第 31 章。

1.2.6 C# 5 和异步编程

C# 5 只有两个新的关键字：`async` 和 `await`，然而，它大大简化了异步方法的编程。在 Windows 8 中，触摸变得更加重要，不阻塞 UI 线程也变得更加重要。用户使用鼠标，习惯于花些时间滚动屏幕。然而，在触摸界面上使用手势时，反应不及时很不好。

Windows 8 还为 Windows Store 应用程序(也称为 Modern 应用程序、Metro 应用程序、通用 Windows 应用程序，最近称为 Windows 应用程序)引入了一个新的编程接口：Windows 运行库。这是一个本地运行库，看起来像是使用语言投射的.NET。许多 WPF 控件都为新的运行库重写了，.NET Framework 的一个子集可以使用这样的应用程序。

`System.AddIn` 框架过于复杂、缓慢，所以用.NET 4.5 创建了一个新的合成框架：Managed Extensibility Framework 和名称空间 `System.Composition`。

独立于平台的通信的新版本是由 ASP.NET Web API 提供的。WCF 提供有状态和无状态的服务，以及许多不同的网络协议，而 ASP.NET Web API 则简单得多，它是基于 Representational State Transfer(REST)软件架构风格的。

注意：

C# 5 的 `async` 和 `await` 关键字在第 15 章中详细讨论，其中也介绍.NET 在不同时期使用的不同异步模式。

MEF 参见网上附加第 1 章。Windows 应用程序参见第 33~36 章，Web API 和 ASP.NET Core MVC 参见第 32 章。

1.2.7 C# 6 和 .NET Core 1.0

C# 6 没有由泛型、LINQ 和异步带来的巨大改进，但有许多小而实用的语言增强，可以在几个地方减少代码的长度。很多改进都通过新的编译器引擎 Roslyn 或 .NET Compiler Platform 实现。

完整的 .NET Framework 并不是近年来使用的唯一 .NET Framework。有些场景需要较小的框架。2007 年，发布了 Microsoft Silverlight 的第一个版本(代码名为 WPF/E，即 WPF Everywhere)。Silverlight 是一个 Web 浏览器插件，支持动态内容。Silverlight 的第一个版本只支持通过 JavaScript 编程。第 2 个版本包含 .NET Framework 的子集。当然，不需要服务器端库，因为 Silverlight 总是在客户端运行，但附带 Silverlight 的 Framework 也删除了核心特性中的类和方法，使其更简洁，便于移植到其他平台。用于桌面的 Silverlight 版本(第 5 版)在 2011 年 12 月发布。Silverlight 也用于 Windows Phone 的编程。Silverlight 8.1 进入 Windows Phone 8.1，但这个版本的 Silverlight 不同于桌面版本。

在 Windows 桌面上，有如此巨大的 .NET 框架，需要更快的开发节奏，也需要较大的改进。在 DevOps 中，开发人员和操作员一起工作，甚至是同一个人不断地为用户提供应用程序和新特性，需要使新特性快速可用。由于框架巨大，且有许多依赖关系，创建新的特性或修复缺陷是一项不容易完成的任务。

有了几个较小的 .NET 版本(如 Silverlight、用于 Windows Phone 的 Silverlight)，在 .NET 的桌面版本和较小版本之间共享代码就很重要。在不同 .NET 版本之间共享代码的一项技术是可移植库。随着时间的推移，有了许多不同的 .NET Framework 和版本，可移植库的管理已成为一场噩梦。

为了解决所有这些问题，需要 .NET 的新版本(是的，的确需要解决这些问题)。Framework 的新版本命名为 .NET Core。 .NET Core 较小，是开源的，带有模块化的 NuGet 包，以及分布给每个应用程序的运行库，不仅可用于 Windows 的桌面版，也可用于许多不同的 Windows 设备，以及 Linux 和 OS X。

为了创建 Web 应用程序，完全重写了 ASP.NET，得到了 ASP.NET Core 1.0。这个版本不完全向后兼容老版本，需要对现有的 ASP.NET MVC(和 ASP.NET Core MVC)代码进行一些修改。然而，与旧版本相比，它也有很多优点，例如每一个网络请求的开销较低，性能更好，也可以在 Linux 上运行。ASP.NET Web Forms 不包含在这个版本中，因为 ASP.NET Web Forms 不是专为最佳性能而设计的，它基于 Windows Forms 应用程序开发人员熟悉的模式来提高对开发人员的友好性。

当然，并不是所有应用程序都很容易改为使用 .NET Core。所以这个巨大的框架也会进行改进——即使这些改进的完成速度没有 .NET Core 那么快，也是要改进的。 .NET Framework 完整的新版本是 4.6。ASP.NET Web Forms 的小更新包在完整的 .NET 上可用。

注意：

C# 语言的变化参见第 I 部分中所有的语言章节，例如，只读属性参见第 3 章，nameof 运算符和空值传播参见第 6 章，字符串插值参见第 9 章，异常过滤器参见第 14 章。

1.2.8 C# 7 和 .NET Core 2.0

C# 更新具有更快的速度。主要版本 7.0 在 2017 年 3 月发布，次级版本 7.1 和 7.2 分别在 2017 年 8 月和 12 月后不久发布。通过项目设置可以选择要使用的编译器版本。

C# 7 引入了许多新特性。这些特性中最重要的部分来自函数式编程：模式匹配和元组。

注意：

模式匹配和元组参见第 13 章。

.NET Core 2.0 的重点是更容易将使用 .NET Framework 编写的现有应用程序引入 .NET Core。以前不能用于 .NET Core、但仍在许多 .NET Framework 应用程序和库中使用的类型，现在可以用于 .NET Core。在 .NET Core 2.0 中添加了两万多个 API。例如，二进制序列化和 DataSet 又回来了，还可以在 Linux 上使用这些特性。另一

个有助于将旧应用程序引入 .NET Core 的特性是 Windows Compatibility Pack (Microsoft.Windows.Compatibility)。这个 NuGet 包定义了用于 WCF、注册表访问、加密、目录服务、绘图等的 API。当前状态参见 <https://github.com/dotnet/designs/blob/master/accepted/compat-pack/compat-pack.md>。

.NET Standard 是一个规范，它定义了在任何支持该标准的平台上应该使用哪些 API。标准版本越高，可用的 API 就越多。.NET Standard 2.0 将标准扩展了两万多个 API，并得到了 .NET Framework 4.6.1、.NET Core 2.0 和通用 Windows 平台(Windows 应用程序)的支持，开始于构建版 16299 (Windows 10 的 Fall Creators Update)。

注意：

第 19 章详细介绍了 .NET Standard。

要检查应用程序是否可以轻松地移植到 .NET Core 中，可以使用 .NET Portability Analyzer (.NET 可移植性分析器)。此工具可以安装为 Visual Studio 的扩展。它会分析二进制文件。还可以为希望获得的版本和框架配置可移植性信息，为 .NET Core、.NET Framework、.NET Standard、Mono、Silverlight、Windows、Xamarin 等提供可移植性信息。结果可以是 JSON、HTML 和 Excel。

图 1-1 显示了在选择 .NET Framework 二进制文件后的总结报告，其中，该二进制文件与 .NET Framework 100% 兼容，与 .NET Core 96.67% 兼容，与 Windows 应用程序 69.7% 兼容。图 1-2 显示了有问题的 API 的详细信息。

Assembly	Target Framework	.NET Core + Platform Extensions	.NET Core	.NET Framework	.NET Standard	Windows	Xamarin Android	Xamarin iOS
SecureTransfer	.NETFramework,Version=v4.7.1	96.97	96.97	100	96.97	69.7	96.97	96.97

图 1-1

Target type	Target member	Assembly	.NET Core + Platform Extensions	.NET Core	.NET Framework	.NET Standard	Windows	Xamarin Android
T:System.Security.Cryptography.CngAlgorithm	T:System.Security.Cryptography.CngAlgorithm	SecureTransfer	Supported: 1.0+	Supported: 1.0+	Supported: 3.5+	Supported: 1.6+	Not supported	Supported: 1.0
T:System.Security.Cryptography.CngAlgorithm	M:System.Security.Cryptography.CngAlgorithm	SecureTransfer	Supported: 1.0+	Supported: 1.0+	Supported: 3.5+	Supported: 1.6+	Not supported	Supported: 1.0
T:System.Console	T:System.Console	SecureTransfer	Supported: 1.0+	Supported: 1.0+	Supported: 1.1+	Supported: 1.3+	Not supported	Supported: 1.0
T:System.Console	M:System.Console.ReadLine	SecureTransfer	Supported: 1.0+	Supported: 1.0+	Supported: 1.1+	Supported: 1.3+	Not supported	Supported: 1.0
T:System.Console	M:System.Console.WriteLine	SecureTransfer	Supported: 1.0+	Supported: 1.0+	Supported: 1.1+	Supported: 1.3+	Not supported	Supported: 1.0
T:System.Console	M:System.Console.WriteLine(System.String)	SecureTransfer	Supported: 1.0+	Supported: 1.0+	Supported: 1.1+	Supported: 1.3+	Not supported	Supported: 1.0
T:System.Security.Cryptography.CngKey	T:System.Security.Cryptography.CngKey	SecureTransfer	Supported: 1.0+	Supported: 1.0+	Supported: 3.5+	Supported: 1.6+	Not supported	Supported: 1.0
T:System.Security.Cryptography.CngKey	M:System.Security.Cryptography.CngKey	CreateSecureTransfer	Supported: 1.0+	Supported: 1.0+	Supported: 3.5+	Supported: 1.6+	Not supported	Supported: 1.0
T:System.Security.Cryptography.CngKey	M:System.Security.Cryptography.CngKey	ExportSecureTransfer	Supported: 1.0+	Supported: 1.0+	Supported: 3.5+	Supported: 1.6+	Not supported	Supported: 1.0
T:System.Security.Cryptography.CngKey	M:System.Security.Cryptography.CngKey	ImportSecureTransfer	Supported: 1.0+	Supported: 1.0+	Supported: 3.5+	Supported: 1.6+	Not supported	Supported: 1.0
T:System.Security.Cryptography.AesCryptoService	T:System.Security.Cryptography.AesCryptoService	SecureTransfer	Supported: 2.0+	Supported: 2.0+	Supported: 3.5+	Supported: 2.0+	Not supported	Supported: 1.0
T:System.Security.Cryptography.AesCryptoService	M:System.Security.Cryptography.AesCryptoService	SecureTransfer	Supported: 2.0+	Supported: 2.0+	Supported: 3.5+	Supported: 2.0+	Not supported	Supported: 1.0
T:System.Security.Cryptography.SymmetricAlgorithm	T:System.Security.Cryptography.SymmetricAlgorithm	SecureTransfer	Supported: 1.0+	Supported: 1.0+	Supported: 1.1+	Supported: 1.3+	Not supported	Supported: 1.0
T:System.Security.Cryptography.SymmetricAlgorithm	M:System.Security.Cryptography.SymmetricAlgorithm	SecureTransfer	Supported: 2.0+	Supported: 2.0+	Supported: 1.1+	Supported: 2.0+	Not supported	Supported: 1.0
T:System.Security.Cryptography.SymmetricAlgorithm	M:System.Security.Cryptography.SymmetricAlgorithm	SecureTransfer	Supported: 1.0+	Supported: 1.0+	Supported: 1.1+	Supported: 1.3+	Not supported	Supported: 1.0
T:System.Security.Cryptography.SymmetricAlgorithm	M:System.Security.Cryptography.SymmetricAlgorithm	SecureTransfer	Supported: 1.0+	Supported: 1.0+	Supported: 1.1+	Supported: 1.3+	Not supported	Supported: 1.0

图 1-2

1.2.9 选择技术，继续前进

知道框架内技术相互竞争的原因后，就更容易选择用于编写应用程序的技术。例如，如果创建新的 Windows 应用程序，使用 Windows Forms 就不是好的解决方案，而应该使用基于 XAML 的技术，例如 Universal Windows Platform (UWP)。当然，使用其他技术仍然有很好的理由。需要支持 Windows 7 客户端吗？在这种情况下，UWP 不合适，但 WPF 合适。仍然可以以一种易于切换到其他技术的方式创建 WPF 应用程序，例如 UWP 和 Xamarin。

注意：

请阅读第 34 章，了解如何设计应用程序，以便在 WPF、UWP 和 Xamarin 之间共享尽可能多的代码。

如果创建 Web 应用程序，肯定应使用 ASP.NET Core 与 ASP.NET Core MVC。做这个选择时要排除 ASP.NET Web Forms。如果访问数据库，就应该使用 Entity Framework Core，应该选择 Managed Extensibility Framework 而不是 System.AddIn。

旧应用程序仍在使用 Windows Forms、ASP.NET Web Forms 和其他一些旧技术。只为改变现有的应用程序而使用新技术是没有意义的。进行修改必须有巨大的优势，例如，维护代码已经是一个噩梦，需要大量的重构以缩短客户要求的发布周期，或者使用一项新技术可以减少更新包的编码时间。根据旧有应用程序的类型，使用新技术可能不值得。可以允许应用程序仍使用旧技术，因为在未来的许多年仍将支持 Windows Forms 和 ASP.NET Web Forms。

本书的内容以新技术为基础，展示创建新应用程序的最佳技术。如果仍然需要维护旧应用程序，可以参考《C#高级编程(第 10 版) C#6 & .NET Core 1.0》，其中介绍了 ASP.NET Web Forms、WCF、Windows Forms、System.AddIn、Workflow Foundation 和其他仍然在 .NET Framework 中可用的旧技术。

1.3 .NET 术语

什么是当前的 .NET 技术？图 1-3 给出了 .NET Framework、.NET Core 和 Mono 相互关联的总体情况。所有的 .NET Framework 应用程序、.NET Core 应用程序和 Xamarin 应用程序如果是用 .NET Standard 构建的，就都可以使用相同的库。这些技术共享相同的编译器平台、编程语言和运行库组件。它们不共享相同的运行库，但是它们在运行时共享组件。例如，.NET Framework 和 .NET Core 使用即时(JIT)编译器 RyuJIT。

使用 .NET Framework，可以创建 Windows Forms、WPF 和在 Windows 上运行的旧 ASP.NET 应用程序。

使用 .NET Core，可以创建在不同平台上运行的 ASP.NET Core 和控制台应用程序。.NET Core 也由通用 Windows 平台(UWP)使用，但这并不能使 UWP 在 Linux 上可用。UWP 还利用了 Windows 运行库，它只能在 Windows 上使用。

Xamarin 提供了 Xamarin.iOS 和 Xamarin.Android 库，它们可以为 iPhone 和 Android 开发 C# 应用程序。有了 Xamarin.Forms，就可以在两个移动平台之间共享用户界面。Xamarin 目前仍然基于 Mono 框架，Mono 框架是由 Xamarin 开发的 .NET 变体。在某种程度上，这可能会变为 .NET Core。然而，重要的是所有这些技术都可以使用为 .NET 标准创建的相同的库。

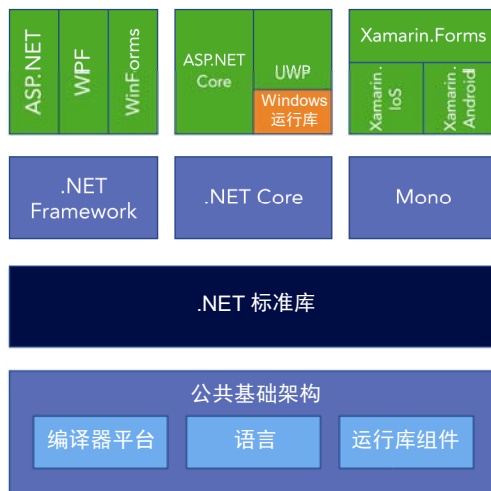


图 1-3

在图 1-3 的下半部分, 可以看到 .NET Framework、.NET Core 和 Mono 之间也有一些共享。运行库组件的代码是共享的, 例如垃圾收集器和 RyuJIT(这是一个新的 JIT 编译器, 可以将 IL 代码编译为本地代码)。垃圾收集器由 CLR、CoreCLR 和 .NET Native 使用。CLR 和 CoreCLR 使用 RyuJIT 即时编译器。所有这些平台都使用 .NET Compiler Platform(也称为 Roslyn)和编程语言。

1.3.1 .NET Framework

.NET Framework 4.7 是 .NET Framework 在过去 15 年不断增强的结果。1.2 节讨论的许多技术都基于这个框架。这个框架用于创建 Windows Forms 和 WPF 应用程序。.NET Framework 4.7 还提供了 Windows Forms 的增强功能, 比如对 High DPI 的支持。

如果希望继续使用 ASP.NET Web Forms, 就应选择 ASP.NET 4.7 和 .NET Framework 4.7。否则, 就需要重写一些代码, 以移动到 .NET Core。根据源代码的质量和添加新特性的需要, 重写代码可能是值得的。

1.3.2 .NET Core

.NET Core 是新的 .NET, 所有新技术都使用它, 是本书的一大关注点。这个框架是开源的, 可以在 <http://www.github.com/dotnet> 上找到它。运行库是 CoreCLR 库; 包含集合类的框架、文件系统访问、控制台和 XML 等都在 CoreFX 库中。

.NET Framework 要求必须在系统上安装应用程序需要的特定版本, 而在 .NET Core 1.0 中, 框架(包括运行库)是与应用程序一起交付的。以前, 把 ASP.NET Web 应用程序部署到共享服务器上有时可能有问题, 因为提供程序安装了旧版本的 .NET。这种情况已经一去不复返了。现在可以同时提交应用程序和运行库, 而不依赖服务器上安装的版本。

.NET Core 以模块化的方式设计。该框架分成数量很多的 NuGet 包。这样就不必处理所有的包了, 而可以用元包来引用一起工作的小包。使用 .NET Core 2.0 和 ASP.NET Core 2.0, 甚至可以改进元包。通过 ASP.NET Core 2.0, 只需要引用 Microsoft.AspNetCore.All, 就可以得到 ASP.NET Core web 应用程序通常需要的所有包。

.NET Core 可以很快更新。即使更新运行库, 也不影响现有的应用程序, 因为运行库与应用程序一起安装。现在, 微软公司可以增强 .NET Core, 包括运行库, 发布周期更短。

注意:

为了使用 .NET Core 开发应用程序, 微软公司创建了新的命令行实用程序 .NET Core Command Line (CLI)。这些工具参见本章后面的内容。

1.3.3 .NET Standard

.NET Standard 不是一个实现, 而是一个协定。本协定规定了需要实现哪些 API。 .NET Framework、.NET Core 和 Xamarin 实现了这个标准。

标准是有版本的。在每个版本中都添加了额外的 API。根据需要的 API, 可以选择库的标准版本。需要检查所选平台是否符合所需版本的标准。

可以在 <https://docs.microsoft.com/en-us/dotnet/standard/net-standard> 中找到 .NET Standard 的平台支持表。以下是需要了解的最重要的部分:

- .NET Core 1.1 支持 .NET Standard 1.6, .NET Core 2.0 支持 .NET Standard 2.0
- .NET Framework 4.6.1 支持 .NET Standard 2.0。
- UWP 构建了 16299, 后来支持 .NET Standard 2.0; 旧版本只支持 .NET Standard 1.4。
- 通过 Xamarin 使用 .NET Standard 2.0, 需要 Xamarin.iOS 10.14 和 Xamarin.Android 8.0。

注意:

请阅读第 19 章中关于 .NET Standard 的详细信息。

1.3.4 NuGet 包

在早期, 程序集是应用程序的可重用单元。添加对程序集的一个引用, 以使用自己代码中的公共类型和方法, 此时, 仍可以这样使用(一些程序集必须这样使用)。然而, 使用库可能不仅意味着添加一个引用并使用它。使用库也意味着一些配置更改, 或者可以通过脚本来利用一些特性。这是在 NuGet 包中打包程序集的一个原因。

NuGet 包是一个 zip 文件, 其中包含程序集(或多个程序集)、配置信息和 PowerShell 脚本。

使用 NuGet 包的另一个原因是, 它们很容易找到, 它们不仅可以从微软公司找到, 也可以从第三方找到。NuGet 包很容易在 NuGet 服务器 <http://www.nuget.org> 上获得。

在 Visual Studio 项目的引用中, 可以打开 NuGet 包管理器(NuGet Package Manager, 见图 1-4), 在该管理器中可以搜索包, 并将其添加到应用程序中。这个工具允许搜索还没有发布的包(包括预发布选项), 定义应该在哪个 NuGet 服务器中搜索包。搜索包的一个地方是自己的共享目录, 其中放置了内部使用的包。

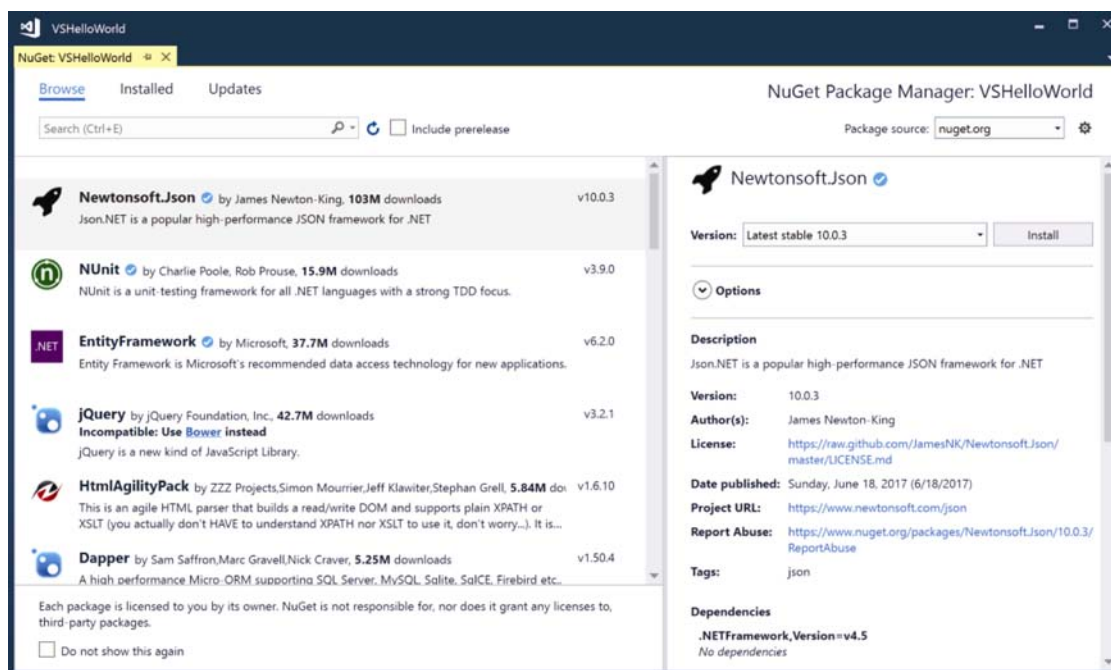


图 1-4

注意:

使用 NuGet 服务器中的第三方包时, 如果一个包以后才能使用, 就总是有风险。还需要检查包的支持可用性。使用包之前, 总要检查项目的链接信息。对于包的来源, 可以选择 Microsoft and .NET, 只获得微软公司支持的包。第三方包也包括在 Microsoft and .NET 部分中, 但它们是微软公司支持的第三方包。

注意:

NuGet 包管理器的更多信息参见第 18 章。

1.3.5 名称空间

可用于 .NET 的类组织在名称以 System 开头的名称空间中。表 1-3 描述的名称空间提供了层次结构的思路。

表 1-3

名 称 空 间	说 明
System.Collections	这是集合的根名称空间。子名称空间也包含集合，如 System.Collections.Concurrent 和 System.Collections.Generic
System.Data	这是访问数据库的名称空间。System.Data.SqlClient 包含访问 SQL Server 的类
System.Diagnostics	这是诊断信息的根名称空间，如事件记录和跟踪(在 System.Diagnostics.Tracing 名称空间中)
System.Globalization	该名称空间包含的类用于全球化和本地化应用程序
System.IO	这是文件 IO 的名称空间，其中的类访问文件和目录，包括读取器、写入器和流
System.Net	这是核心网络的名称空间，比如访问 DNS 服务器，用 System.Net.Sockets 创建套接字
System.Threading	这是线程和任务的根名称空间。任务在 System.Threading.Tasks 中定义

注意：

一些新的.NET 类使用名称以 Microsoft 开头而不是以 System 开头的名称空间，比如用于 Entity Framework Core 的 Microsoft.EntityFrameworkCore，用于新的依赖关系注入框架的 Microsoft.Extensions.DependencyInjection。

1.3.6 公共语言运行库

UWP 利用 Native .NET 通过 AOT Compiler 把 IL 编译成本地代码。这与 Xamarin.iOS 类似。在所有其他场景中，使用 .NET Framework 的应用程序和使用 .NET Core 1.0 的应用程序都需要 CLR(Common Language Runtime，公共语言运行库)。然而，.NET Core 使用 CoreCLR，而 .NET Framework 使用 CLR。那么，CLR 的作用是什么？

在 CLR 执行应用程序之前，编写好的源代码(使用 C#或其他语言编写的代码)都需要编译。在 .NET 中，编译分为两个阶段：

- (1) 将源代码编译为 Microsoft 中间语言(Intermediate Language，IL)。
- (2) CLR 把 IL 编译为平台专用的本地代码。

IL 代码在 .NET 程序集中可用。在运行时，JIT 编译器编译 IL 代码，创建特定于平台的本地代码。

新的 CLR 和 CoreCLR 包括一个新的 JIT 编译器 RyuJIT。新的 JIT 编译器不仅比以前的版本快，还在用 Visual Studio 调试时更好地支持 Edit & Continue 特性。Edit & Continue 特性允许在调试时编辑代码，可以继续调试会话，而不需要停止并重新启动过程。

CLR 还包括一个带有类型加载器的类型系统，类型加载器负责从程序集中加载类型。类型系统中的安全基础设施验证是否允许使用某些类型系统结构，如继承。

创建类型的实例后，实例还需要销毁，内存也需要回收。CLR 的另一个功能是垃圾收集器。垃圾收集器从托管堆中清除不再引用的内存。

CLR 还负责线程的处理。在 C#中创建托管的线程不一定来自底层操作系统。线程的虚拟化和管理由 CLR 负责。

注意：

如何在 C#中创建和管理线程参见第 21 章和 22 章。第 17 章介绍了垃圾收集器和清理内存的方法。

1.3.7 Windows 运行库

从 Windows 8 开始，Windows 操作系统提供了另一种框架：Windows 运行库(Windows Runtime)。这个运行库由 WUP(Windows Universal Platform，Windows 通用平台)使用，Windows 8 使用第 1 版，Windows 8.1 使用第 2 版，Windows 10 使用第 3 版。

与 .NET Framework 不同，这个框架是使用本地代码创建的。当它用于 .NET 应用程序时，所包含的类型

和.NET类似。在语言投射的帮助下，Windows 运行库可以用于 JavaScript、C++和.NET 语言，它看起来像编程环境的本地代码。不仅方法因区分大小写而行为不同，方法和类型也可以根据所处的位置有不同的名称。

Windows 运行库提供了一个对象层次结构，它在以 Windows 开头的名称空间中组织。这些类没有复制.NET Framework 的很多功能；相反，提供了额外的功能，用于在 UWP 上运行的应用程序。如表 1-4 所示。

表 1-4

名 称 空 间	说 明
Windows. ApplicationModel	这个名称空间及其子名称空间(如 Windows.ApplicationModel.Contracts)定义了类，用于管理应用程序的生命周期，与其他应用程序通信
Windows.Data	Windows.Data 定义了子名称空间，来处理文本、JSON、PDF 和 XML 数据
Windows.Devices	地理位置、智能卡、服务设备点、打印机、扫描仪等设备可以用 Windows.Devices 子名称空间访问
Windows.Foundation	Windows.Foundation 定义了核心功能。集合的接口用名称空间 Windows.Foundation.Collections 定义。这里没有具体的集合类。相反，.NET 集合类型的接口映射到 Windows 运行库类型
Windows.Media	Windows.Media 是播放、捕获视频和音频、访问播放列表和语音输出的根名称空间
Windows.Networking	这是套接字编程、数据后台传输和推送通知的根名称空间
Windows.Security	Windows.Security.Credentials 中的类提供了密码的安全存储区，Windows.Security.Credentials.UI 提供了一个选择器，用于从用户处获得凭据
Windows.Services. Maps	这个名称空间包含用于定位服务和路由的类
Windows.Storage	有了 Windows.Storage 及其子名称空间，就可以访问文件和目录，使用流和压缩
Windows.System	Windows.System 名称空间及其子名称空间提供了系统和用户的信息，也提供了一个启动其他应用程序的启动器
Windows.UI.Xaml	在这个名称空间中，可以找到很多用于用户界面的类型

1.4 用.NET Core CLI 编译

在本书的许多章节中并不需要 Visual Studio，而可以使用任何编辑器和命令行。要创建和编译应用程序，可以使用.NET Core 命令行接口(Command Line Interface, CLI)。下面看看如何设置系统，以及如何使用这个工具。

1.4.1 设置环境

安装了 Visual Studio 2017 和最新的更新包后，就可以立即启动 CLI 工具。可以在没有 Visual Studio 2017 的情况下建立一个系统，还可以在 Linux 和 OS X 上使用大部分示例。为了下载环境的应用程序，只需要访问 <https://dot.net> 并单击 Get Started 按钮。从这里，可以下载 Windows、Linux 和 macOS 的 .NET SDK。

对于 Windows，可以下载安装 SDK 的可执行文件。使用 Linux，需要选择 Linux 发行版来获得相应的命令：

- 通过 Red Hat 和 CentOS，使用 yum 安装.NET SDK
- 通过 Ubuntu 和 Debian，使用 apt-get
- 通过 Fedora，使用 dnf install
- 通过 SLES/openSUSE，使用 zipper install
- 要在 Mac 上安装.NET SDK，可以下载.pkg 文件。

在 Windows 上，不同版本的.NET Core 运行库以及 NuGet 包安装在用户配置文件中。使用.NET 时，这个文件夹的大小会增加。随着时间的推移，会创建多个项目，NuGet 包不再存储在项目中，而是存储在这个用户专用的文件夹中。这样做的优势在于，不需要为每个不同的项目下载 NuGet 包。这个 NuGet 包下载后，它就在

系统上。因为不同版本的 NuGet 包和运行库都是可用的，所有的不同版本都存储在这个文件夹中。不时地检查这个文件夹，删除不再需要的旧版本，可能很有趣。

安装 .NET Core CLI 工具，要把 dotnet 工具作为入口点来启动所有这些工具。只需要启动：

```
> dotnet --help
```

会看到，dotnet 工具的所有不同选项都可用。许多选项都有简化符号。要获得帮助，可以输入：

```
> dotnet -h
```

1.4.2 创建应用程序

dotnet 工具提供了一种简单的方法创建“Hello World!”应用程序。输入如下命令：

```
> dotnet new console --output HelloWorld
```

这个命令创建一个新的 HelloWorld 目录并添加源代码文件 Program.cs 和项目文件 HelloWorld.csproj。从 .NET Core 2.0 开始，这个命令还包括 dotnet restore，所有的 NuGet 包都会下载。要查看应用程序使用的库的依赖项和版本列表，可以检查 obj 子目录中的文件 project.assets.json。如果不使用选项--output (或-o 速记符号)，文件就在当前目录中生成。

生成的源代码如下所示(代码文件 HelloWorld/Program.cs)：

```
using System;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

自从 20 世纪 70 年代 Brian Kernighan 和 Dennis Ritchie 撰写了《C 编程语言》一书，使用“Hello World”应用程序开始，学习编程语言就变成一种传统。使用 .NET Core CLI，这个程序会自动生成。

下面看看这个程序的语法。Main()方法是 .NET 应用程序的入口点。CLR 在启动时调用静态 Main()方法。Main()方法需要放到一个类中。这里，这个类命名为 Program，但是可以给它指定任何名称。

Console.WriteLine 调用 Console 类的 WriteLine()方法。Console 类在 System 名称空间中。不需要编写 System.Console.WriteLine 调用该方法；System 名称空间在源文件的顶部使用 using 声明打开。

在编写源代码之后，需要编译代码来运行它。

创建的项目配置文件名为 HelloWorld.csproj。与较老版本的 csproj 文件相比，新项目文件减少为几行，有几个默认值：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>
</Project>
```

对于项目文件，OutputType 定义了输出的类型。对于控制台应用程序，该类型是 Exe。TargetFramework 指定了用于构建应用程序的框架和版本。在样例项目中，应用程序是使用 .NET Core 2.0 构建的。可以将此元素更改为 TargetFramework，并指定多个框架，如 netcoreapp2.0;net47 用于为 .NET Framework 4.7 和 .NET Core 2.0 构建应用程序(项目文件 HelloWorld/HelloWorld.csproj)：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFrameworks>netcoreapp2.0;net47</TargetFrameworks>
  </PropertyGroup>
</Project>
```

sdk 属性指定项目使用的 SDK。微软有两个主要的 SDK：Microsoft.NET.Sdk 用于控制台应用程序，Microsoft.NET.Sdk.Web 用于 ASP.NET Core web 应用程序。

不需要向项目添加源文件。在编译时，会自动添加同一目录和子目录下扩展名为 cs 的文件。扩展名为 resx 的资源文件是自动添加的，用于嵌入资源。可以更改默认行为，并显式排除/包含文件。

也不需要添加.NET Core包。通过指定目标框架 netcoreapp2.0, 引用许多其他包的元包 Microsoft.NETCore.App 会自动包含在内。

1.4.3 构建应用程序

要构建应用程序，需要将当前目录更改为应用程序的目录，并启动 dotnet build。为.NET Core 2.0 和.NET Framework 4.7 编译时，输出如下：

```
> dotnet build
Microsoft (R) Build Engine version 15.5.179.9764 for .NET Core
Copyright (C) Microsoft Corporation. All rights reserved.

Restore completed in 19.8 ms for
  C:\procsharp\Intro\HelloWorld\HelloWorld.csproj.
HelloWorld -> C:\procsharp\Intro\HelloWorld\bin\Debug\net47\HelloWorld.exe
HelloWorld ->
  C:\procsharp\Intro\HelloWorld\bin\Debug\netcoreapp2.0\HelloWorld.dll

Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:01.58
```

注意：

命令 dotnet new 和 dotnet build 现在包括恢复 NuGet 包。使用 dotnet restore 还可以显式地恢复 NuGet 包。

编译过程的结果是在 bin/debug/[netcoreapp2.0|net47]文件夹中的程序集包含 Program 类的 IL 代码。如果比较.NET Core 与.NET 4.7 的构建结果，会发现一个包含了 IL 代码和.NET Core 的 DLL，以及一个包含了 IL 代码和.NET 4.7 的 EXE。为.NET Core 生成的程序集有一个对 System.Console 程序集的依赖项，而.NET 4.6 程序集在 mscorlib 程序集中找到 Console 类。

要构建发布代码，就需要指定选项--configuration Release (简写为-c Release)：

```
> dotnet build --configuration Release
```

以下章节中的一些代码示例使用了 C# 7.1 或 C# 7.2 提供的功能。默认情况下，使用编译器的最新主版本，即 C# 7.0。要启用新版本的 C#，需要在项目文件中指定这一点，如下面的项目文件部分所示。这里，配置了 C#编译器的最新版本。

```
<PropertyGroup>
  <LangVersion>latest</LangVersion>
</PropertyGroup>
```

1.4.4 运行应用程序

要运行应用程序，可以使用 dotnet run 命令。

```
> dotnet run
```

如果项目文件面向多个框架，就需要通过--framework 选项告诉 dotnet run 命令，使用哪个框架来运行应用程序。这个框架必须通过 csproj 文件来配置。使用样例应用程序，可以在恢复信息之后得到如下输出：

```
> dotnet run --framework netcoreapp2.0
Microsoft (R) Build Engine version 15.5.179.9764 for .NET Core
Copyright (C) Microsoft Corporation. All rights reserved.

Restore completed in 20.65 ms for
  C:\procsharp\Intro\HelloWorld\HelloWorld.csproj.
```

Hello World!

在生产系统中，不使用 `dotnet run` 运行应用程序，而可以使用 `dotnet` 和库的名称：

```
> dotnet bin/debug/netcoreapp2.0/HelloWorld.dll
```

还可以创建可执行文件，但可执行文件是特定于平台的。

注意：

前面是在 Windows 上构建和运行“Hello World!”应用程序，而 `dotnet` 工具在 Linux 和 OS X 上的工作方式是相同的。可以在这两个平台上使用相同的 `dotnet` 命令。

本书的重点是 Windows，因为 Visual Studio 2017 提供了一个比其他平台更强大的开发平台，但本书的许多代码示例是基于 .NET Core 的，所以也能够其他平台上运行。还可以使用 Visual Studio Code(一个免费的开发环境)，直接在 Linux 和 OS X 上开发应用程序，参见 1.7 节，了解 Visual Studio 不同版本的更多信息。

1.4.5 创建 Web 应用程序

还可以使用 .NET Core CLI 创建 Web 应用程序。启动 `dotnet new` 时，可以看到可用的模板列表(参见图 1-5)。

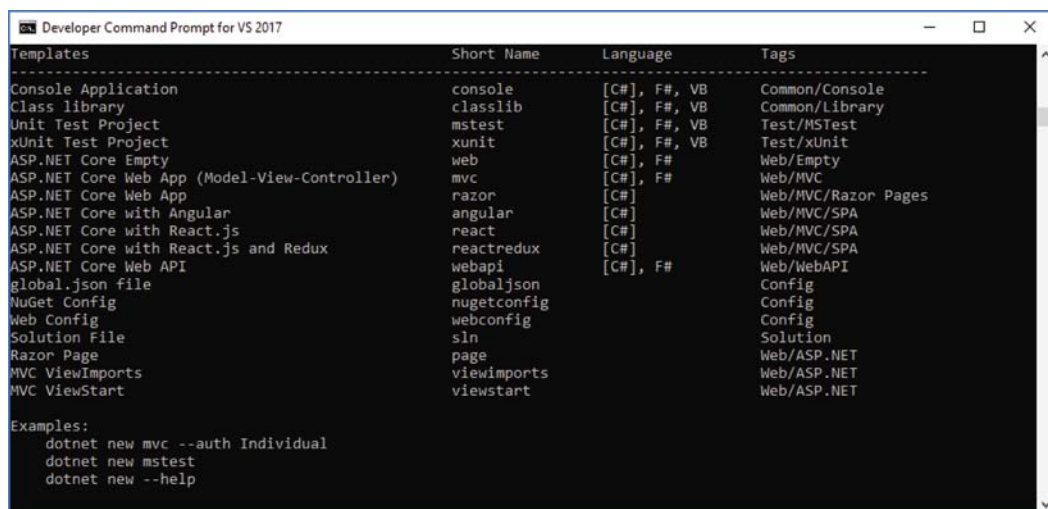


图 1-5

下面的命令

```
> dotnet new mvc -o WebApp
```

会使用 ASP.NET Core MVC 创建新的 ASP.NET Core web 应用程序。切换到 WebApp 文件夹之后，使用下述命令构建和运行程序：

```
> dotnet build
> dotnet run
```

运行上述代码会启动 ASP.NET Core 的 Kestrel 服务器，来监听端口 5000。可以打开浏览器访问从这个服务器返回的页面，如图 1-6 所示。

1.4.6 发布应用程序

使用 `dotnet` 工具，可以创建一个 NuGet 包并发布应用程序来进行部署。首先创建应用程序的框架依赖部署。这减少了发布所需的文件。

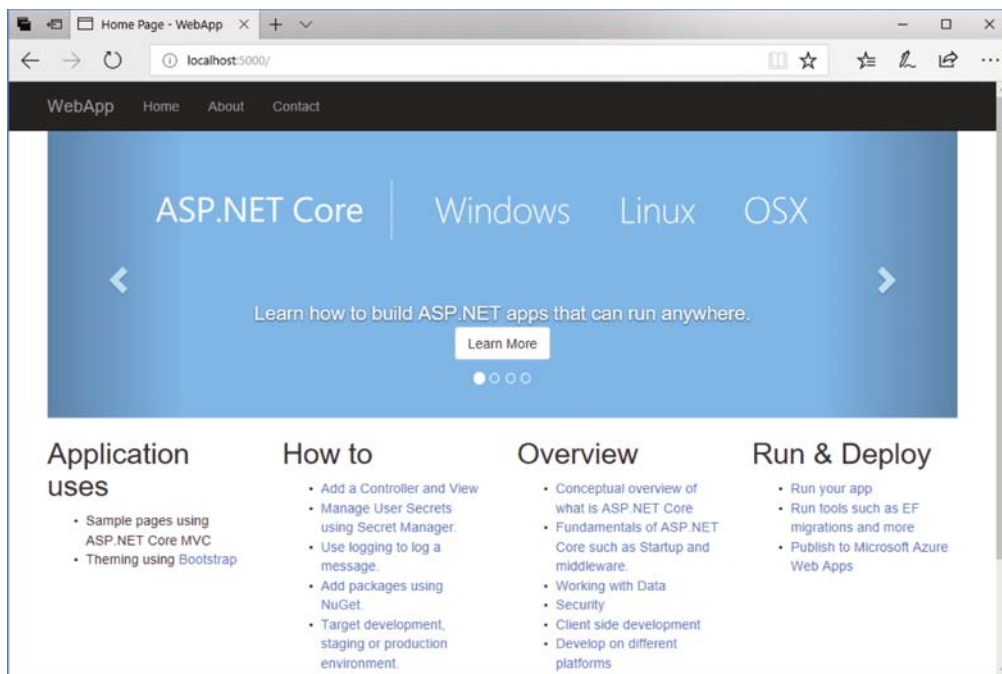


图 1-6

使用之前创建的控制台应用程序，只需要以下命令来创建发布所需的文件。使用-f 选择框架，使用-c 选择版本配置。

```
> dotnet publish -f netcoreapp2.0 -c Release
```

发布所需的文件放在 `bin/Release/netcoreapp2.0/publish` 目录中。

在目标系统上使用这些文件进行发布，也需要运行库。在 <https://www.microsoft.com/net/download/> 上可以找到运行库的下载和安装说明。

在 .NET Framework 中，相同的安装运行库可以由不同的 .NET Framework 版本使用(例如，.NET Framework 4.0 运行库和更新包可以在 .NET Framework 4.7、4.6、4.5、4.0 等应用程序中使用)，与 .NET Framework 相反，对于 .NET Core，要运行应用程序，就需要相同的运行库版本。

注意：

如果应用程序使用了额外的 NuGet 包，这些就需要在 csproj 文件中引用，并且库需要与应用程序一起交付。阅读第 19 章，了解更多信息。

自包含部署

应用程序不需要在目标系统上安装运行库，而是可以用它交付运行库。这就是所谓的自包含部署。

平台不同，运行库就不同。因此，对于自包含部署，需要通过在项目文件中指定 `RuntimeIdentifiers`，来指定支持的平台，如下面的项目文件所示。这里，指定了 Windows 10、macOS 和 Ubuntu Linux 的运行库标识符(项目文件 `SelfContainedHelloWorld/SelfContainedHelloWorld.csproj`)：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>
  <PropertyGroup>
    <RuntimeIdentifiers>
      win10-x64;ubuntu-x64;osx.10.11-x64;
    </RuntimeIdentifiers>
  </PropertyGroup>
</Project>
```

注意：

在 <https://docs.microsoft.com/en-us/dotnet/core/rid-catalog> 的 .NET Core Runtime Identifier(RID)类别中可以获取不同平台和版本的所有运行库标识符。

现在可以为所有不同的平台创建发布文件：

```
> dotnet publish -c Release -r win10-x64
> dotnet publish -c Release -r osx.10.11-x64
> dotnet publish -c Release -r ubuntu-x64
```

在运行这些命令之后，可以在 Release/[win10-x64|osx.10.11-x64|ubuntu-x64]/publish 目录中找到发布所需要的文件。随着 .NET Core 2.0 的规模越来越大，发布的规模也越来越大。在这些目录中，可以找到特定于平台的可执行文件，可以在不使用 dotnet 命令的情况下直接启动它。

1.5 使用 Visual Studio 2017

接下来使用 Visual Studio 2017 代替命令行。本节将介绍 Visual Studio 中最重要的部分来开始工作。Visual Studio 的更多特性在第 18 章中介绍。

1. 安装 Visual Studio 2017

Visual Studio 2017 提供了一个新的安装程序，它可以更容易安装需要的产品。使用安装程序，可以选择开发应用程序所需的工作负载(参见图 1-7)。为了涵盖本书的所有章节，安装这些工作负载：

- UWP 开发
- .NET Desktop 开发
- ASP.NET 和 Web 开发
- Azure 开发
- 移动开发与 .NET
- .NET Core 跨平台开发

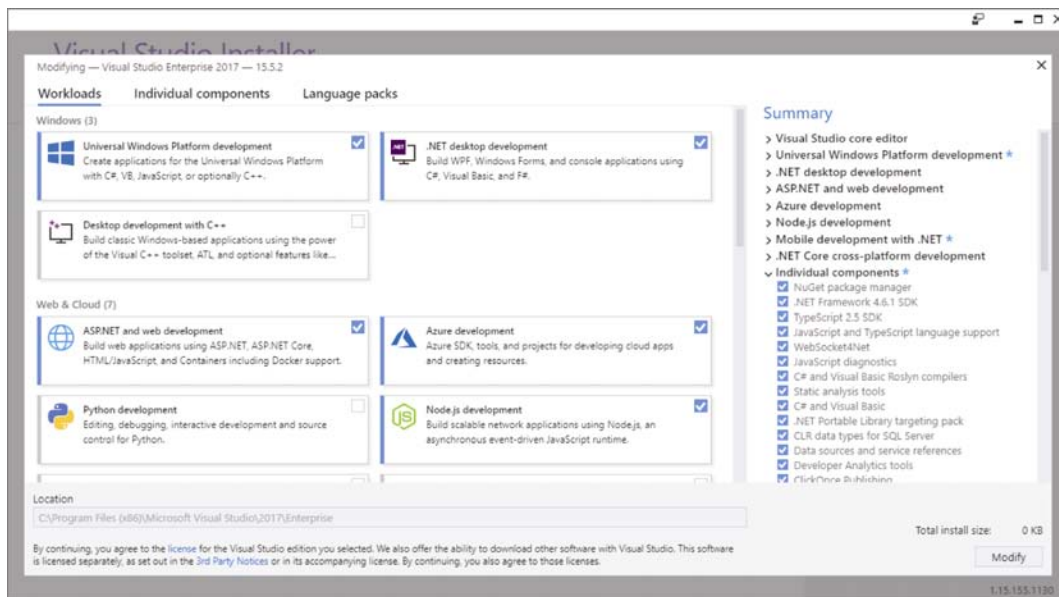


图 1-7

2. 创建一个项目

你可能会被大量的菜单项和 Visual Studio 中的许多选项所淹没。要在本书的第 1 章中创建简单的应用程序，

只需要使用 Visual Studio 的一小部分特性。同样，这本完整的参考书只涵盖了可以用 Visual Studio 完成的一部分操作。Visual Studio 的许多特性是为遗留应用程序以及其他编程语言提供的。

在启动 Visual Studio 之后，首先要创建一个新项目。选择菜单 File | New | Project。打开如图 1-8 所示的对话框，其中列出了可以用来创建新项目的项目项的列表。

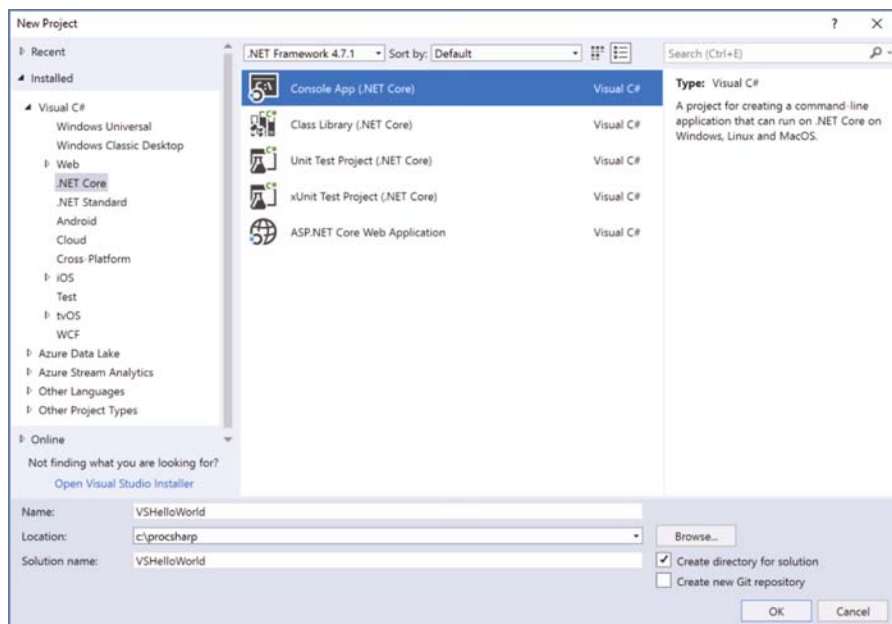


图 1-8

本书主要讨论的是 Visual C#项目项的一个子集。在本章中，选择.NET Core 类别和项目模板 Console App (.NET Core)。在如图 1-8 所示的对话框顶部，选择了.NET Framework 版本。不要混淆，这个选择并不适用于.NET Core 项目。

在此对话框的下半部分，可以输入应用程序的名称，选择存储项目的文件夹，并为解决方案输入一个名称。解决方案可以包含多个项目。

单击 OK 按钮，创建“Hello World!”应用程序。

3. 使用 Solution Explorer

在 Solution Explorer 中(参见图 1-9)，可以看到解决方案、属于解决方案的项目以及项目中的文件。可以选择能进入类和类成员的源代码文件。

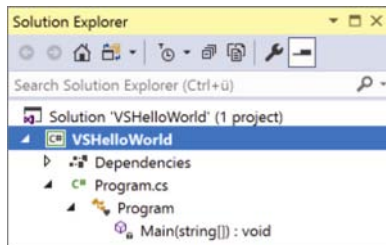


图 1-9

在 Solution Explorer 中选择一项，并单击鼠标右键或按下键盘上的应用程序键时，会打开该项的上下文菜单，如图 1-10 所示。可用的菜单取决于选择的项，以及与 Visual Studio 一起安装的特性。

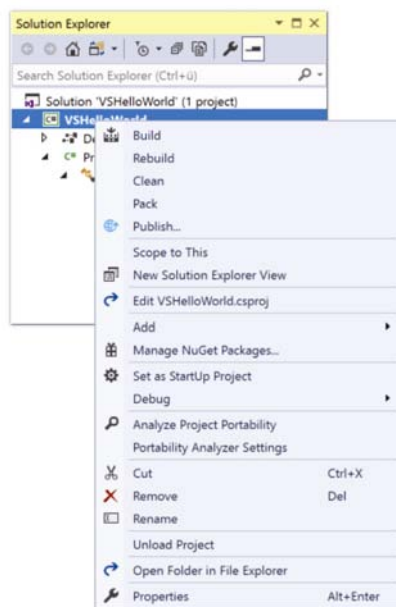


图 1-10

打开项目的上下文菜单时，其中有一个菜单项是用于编辑项目文件的。此选项会打开项目文件 `VSHelloWorld.csproj`，其内容与使用 .NET Core CLI 时看到的内容相同。

4. 配置项目属性

为了配置项目属性，应在 Solution Explorer 中单击项目的上下文菜单，再单击 `Properties`，或选择 `Project | VSHelloWorld Properties`。打开如图 1-11 所示的视图。在这里，可以配置项目的不同设置，如要使用的 .NET Core 版本(假定已经安装了多个框架)、构建设置、在构建过程中应该调用的命令、包配置以及在调试应用程序时使用的参数和环境变量。如前所述，对于一些代码示例，C# 7.0 是不够的。可以使用 `Build` 类别配置 C# 编译器的不同版本。单击 `Advanced` 按钮将打开 `Advanced Build Settings` 对话框(参见图 1-12)。在这里，可以配置 C# 编译器的版本。这个选择会进入 `csproj` 项目配置文件。

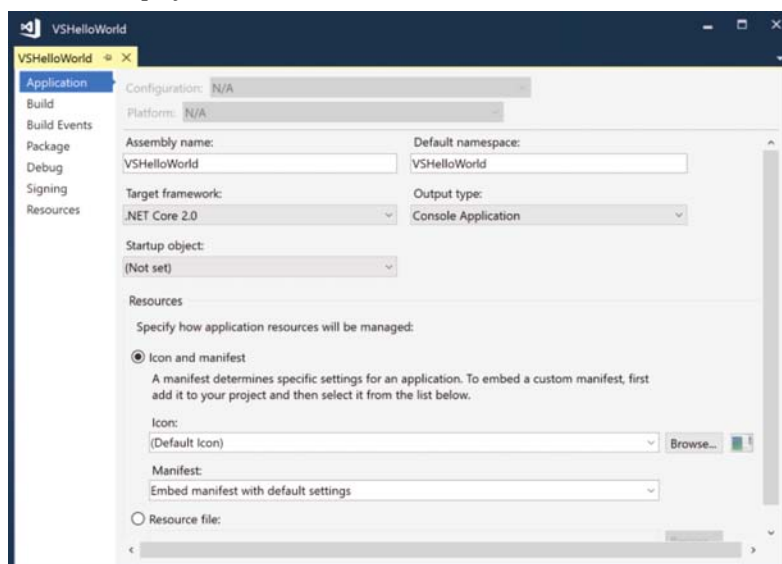


图 1-11

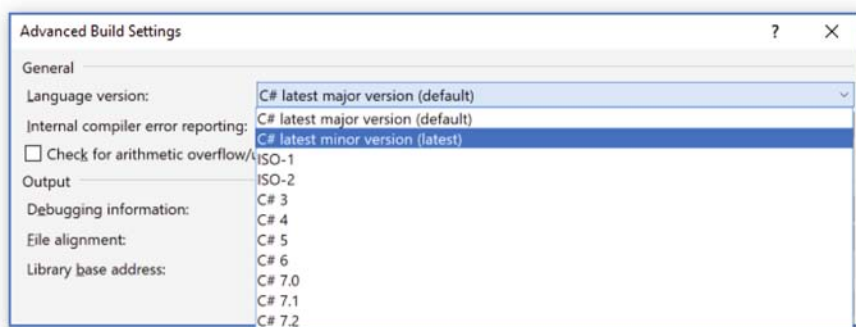


图 1-12

注意：

在对项目属性进行更改时，需要确保在对话框顶部选择正确的配置。如果只使用 Debug 配置更改 C# 编译器的版本，那么当使用新的 C# 语言特性时，构建版本代码就会失败。对于想要的所有配置设置，请选择配置 AllConfigurations。

5. 了解编辑器

Visual Studio 编辑器非常强大。它提供了智能感知功能，该功能可以在按下 Tab 键时，调用方法和属性，并完成输入。在键入时进行编译，可以立即看到带有下划线代码的语法错误。将鼠标指针悬停在下划线的文本上，会弹出一个包含错误描述的小框。

代码编辑器的一个重要的高效特性是代码片段。它们会减少需要输入的内容。只要在编辑器中输入 `cw`，再按 Tab 键，编辑器就会创建 `Console.WriteLine()`。Visual Studio 附带了许多代码片段，选择 Tools | Code Snippets Manager，打开 Code Snippets Manager 对话框(参见图 1-13)，可以看到这些代码片段。在这里，可以在 Language 字段中给用 C# 语言定义的代码片段选择 CSharp，选择组 Visual C# 可以查看为 C# 预定义的所有代码片段。

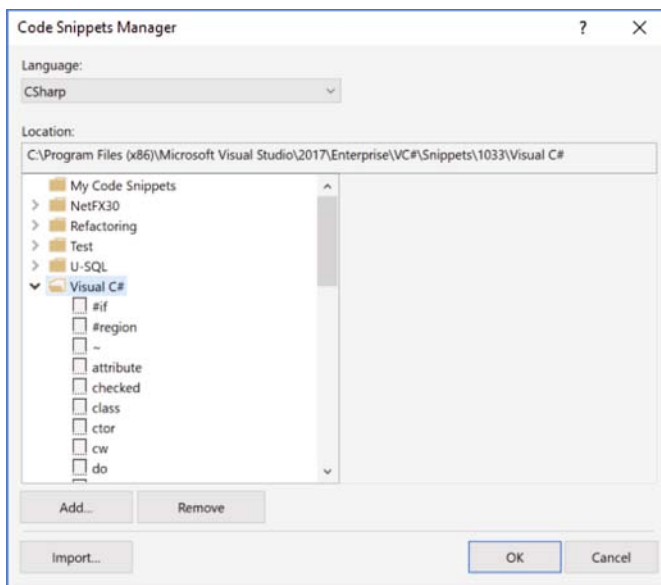


图 1-13

6. 构建项目

从菜单 Build | Build Solution 中编译项目。如果出现错误，Error List 窗口会显示错误和警告。但是，Output 窗口(参见图 1-14)比 Error List 窗口更可靠。有时，Error List 窗口包含了较老的缓存信息，或者当列表较大时，查找错误并不容易。Output 窗口通常为许多不同的工具提供了很好的信息。选择 View | Output，就可以打开 Output 窗口。

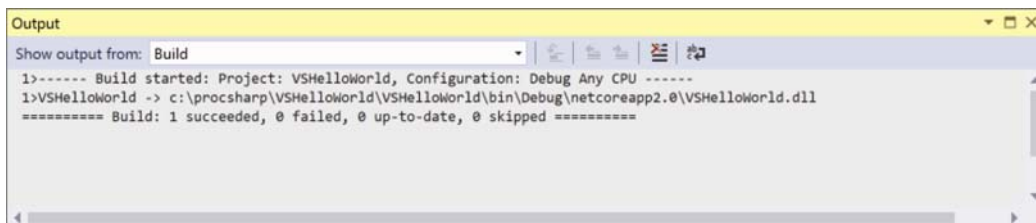


图 1-14

7. 运行应用程序

要运行应用程序，选择 Debug | Start Without Debugging。这将启动应用程序，并保持控制台窗口打开，直到关闭它为止。

请记住，可以选择 Debug 类别，在 Project Properties 中配置应用程序参数。

8. 调试

要调试应用程序，可以单击编辑器中的左侧灰色区域来创建断点(参见图 1-15)。有断点之后，就可以通过选择 Debug | Start Debugging 启动调试器。到达一个断点时，可以使用 Debug 工具栏(参见图 1-16)，以进入、结束或退出方法，也可以使用显示下一条语句。将鼠标悬停在变量上，可以查看当前值。还可以在 Locals 和 Watch 窗口中检查变量设置，也可以在应用程序运行时更改值。

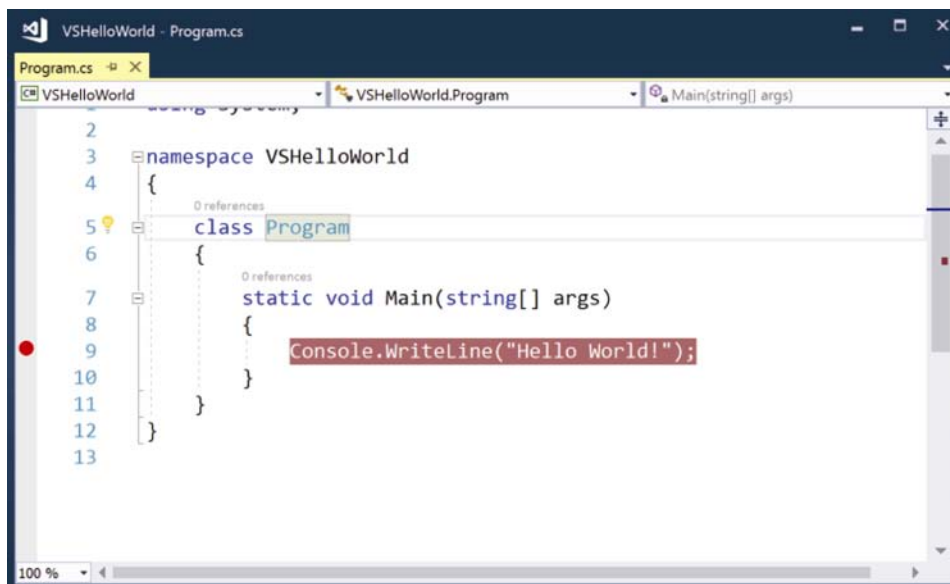


图 1-15



图 1-16

前面介绍的 Visual Studio 部分是帮助领悟本书第 1 章的最重要因素。第 18 章对 Visual Studio 2017 进行了深入的研究。

1.6 应用程序类型和技术

可以使用 C# 创建控制台应用程序，本章的大多数示例都是控制台应用程序。对于实际的程序，控制台应用程序并不常用。使用 C# 创建的应用程序可以使用与 .NET 相关的许多技术。本节概述可以用 C# 编写的不同类型的程序。

1.6.1 数据访问

在介绍应用程序类型之前，先看看所有应用程序类型都使用的技术：数据访问。

文件和目录可以使用简单的 API 调用来访问，但简单的 API 调用对于有些场景而言不够灵活。使用流 API 有很大的灵活性，流提供了更多的特性，例如加密或压缩。阅读器和写入器简化了流的使用。所有可用的不同选项都包含在第 22 章中。也可能以 XML 或 JSON 格式序列化完整的对象。网上附加第 2 章讨论了这些选项。

为了读取和写入数据库，可以直接使用 ADO.NET(参见第 25 章)，也可以使用抽象层：Entity Framework Core(参见第 26 章)。Entity Framework Core 提供了从对象层次结构到数据库关系的映射。

Entity Framework Core 1.0 是 Entity Framework 的完全重新设计，新名称反映了这一点。代码需要更改，把应用程序从 Entity Framework 的旧版本迁移到新版本。旧的映射变体，如 Database First 和 Model First 已被删除，因为 Code First 是更好的选择。完全重新设计也支持关系数据库和 NoSQL。Entity Framework Core 2.0 有一长串的新特性，本书将介绍这些内容。

1.6.2 Windows 应用程序

对于创建 Windows 应用程序，选择的技术应该是 UWP(通用 Windows 平台)。当然，当这个选项无法使用时，会有一些限制——例如，如果仍然需要支持 Windows 7 这样的旧 O/S 版本。此时，可以使用 Windows Presentation Foundation (WPF)。本书不介绍 WPF，但是可以阅读《C# 高级编程(第 10 版) C# 6 & .NET Core 1.0》，它有 5 个章节专门介绍 WPF，其他章节也介绍了 WPF。

本书的一个重点是：通过 UWP 开发应用程序。与 WPF 相比，UWP 提供了更现代的 XAML 来创建用户界面。例如，数据绑定提供了一个编译后的绑定变体，在编译时会得到错误，而不是显示绑定的数据。应用程序在运行于客户端系统之前被编译为本机代码。它提供了现代的设计，现在称为微软中的流畅设计。

注意：

第 33 章介绍了 UWP 应用程序的创建，并介绍了 XAML、不同的 XAML 控件和应用程序的生命周期。通过支持 MVVM 模式，可以使用 WPF、UWP 和 Xamarin，利用尽可能多的公共代码来创建应用程序。这一模式在第 34 章中介绍。为了给应用程序创建酷炫的外观和风格，请务必阅读第 35 章。第 36 章深入介绍了 UWP 的一些高级特性。

1.6.3 Xamarin

如果 Windows 在移动电话市场上扮演更大的角色，那就太好了。然后，UWA(通用 Windows 应用程序)也会在移动电话上运行。但事实并非如此，移动电话上运行 Windows 已经是过去的事情了。然而，通过 Xamarin，你可以使用 C# 和 XAML 在 iPhone 和 Android 上创建应用程序。Xamarin 提供了可以在 Android 上创建应用程序的 API，以及使用熟悉的 C# 代码在 iPhone 上创建应用程序的库。

对于 Android，使用 Android Callable Wrappers (ACW) 和 Managed Callable Wrappers (MCW) 的映射层可以用于在 .NET 代码和 Android 的 Java 运行库之间进行交互操作。在 iOS 中，Ahead of Time (AOT) 编译器将托管代码编译为本地代码。

Xamarin.Forms 提供了 XAML 代码来创建用户界面，并在 Android、iOS、Windows 和 Linux 之间共享尽可能多的用户界面。XAML 只提供可以映射到所有平台的 UI 控件。为了使用平台上的特定控件，可以创建特定

于平台的渲染器。

注意：

用 Xamarin 和 Xamarin.Forms 开发的内容参见第 37 章。

1.6.4 Web 应用程序

最初引入 ASP.NET，从根本上改变了 Web 编程模型。ASP.NET Core 再次改变了它，允许使用 .NET Core 提高性能和可伸缩性。这个新版本也可以在 Windows 和 Linux 系统上运行。

在 ASP.NET Core 中，不再包含 ASP.NET Web Forms(它仍然可以使用，在 .NET 4.7 中更新)。

ASP.NET Core MVC 基于著名的 MVC(模型-视图-控制器)模式，更容易进行单元测试。它还允许把编写用户界面代码与 HTML、CSS、JavaScript 清晰地分离，它只在后台使用 C#。

注意：

第 30 章介绍了 ASP.NET Core 的基础，第 31 章继续使用 ASP.NET Core MVC 框架加固基础。

1.6.5 Web API

过去，SOAP 和 WCF 完成了任务，就不再需要它们了。现代应用程序利用 REST (Representational State Transfer) 和 Web API。使用 ASP.NET Core 创建 Web API 是一种更容易进行通信的选项，它满足了分布式应用程序 90% 以上的需求。这项技术是基于 REST 的，它为无状态、可伸缩的 Web 服务定义了指导方针和最佳实践。

客户端可以接收 JSON 或 XML 数据。JSON 和 XML 也可以格式化来使用 Open Data(ODa)规范。

这个 API 的新特性更容易在 Web 客户端上使用 JavaScript、UWP 和 Xamarin。

创建 Web API 是构建微服务的好方法。构建微服务的方法定义了更小的服务，这些服务可以独立地运行和部署，可以自己控制数据存储。

为了描述服务，定义了一个新的标准：OpenAPI (<https://www.openapis.org>)。这个标准植根于 Swagger (<https://swagger.io/>)。

注意：

ASP.NET Core Web API、Swagger 和更多关于微服务的信息参见第 32 章。

1.6.6 WebHooks 和 SignalR

对于实时 Web 功能以及客户端和服务端之间的双向通信，可以使用的 ASP.NET Core 和 .NET Core 2.1 技术是 WebHooks 和 SignalR。

只要信息可用，SignalR 就允许将信息尽快推送给连接的客户。SignalR 使用 WebSocket 技术推送信息。

WebHooks 可以集成公共服务，这些服务可以调用公共 ASP.NET Core 创建的 Web API 服务。WebHooks 技术从 GitHub、Dropbox 和其他服务中接收推送通知。

注意：

SignalR 连接管理、连接的分组，以及 WebHooks 的授权和集成的基础知识参见网上附加第 3 章。

1.6.7 Microsoft Azure

现在，在考虑开发图景时不能忽视云。虽然没有专门的章节讨论云技术，但在本书的几章中都引用了 Microsoft Azure。

Microsoft Azure 提供了软件即服务(Software as a Service, SaaS)、基础设施即服务(Infrastructure as a Service, IaaS)、平台即服务(Platform as a Service, PaaS) 和函数即服务(Functions as a Service, FaaS)。有时产品介于这些类别之间。下面介绍这些 Microsoft Azure 产品。

1. SaaS

SaaS 提供了完整的软件，不需要处理服务器的管理和更新等。Office 365 是一个 SaaS 产品，它通过云产品使用电子邮件和其他服务。与开发人员相关的 SaaS 产品是 Visual Studio Team Server。Visual Studio Team Server 是云中的 Foundation Server，可以用作私人代码库，跟踪错误和工作项，以及构建和测试服务。第 18 章介绍了 Visual Studio 中可用的 DevOps 特征。

2. IaaS

另一个服务产品是 IaaS。这个服务产品提供了虚拟机。用户负责管理操作系统，维护更新。当创建虚拟机时，可以决定不同的硬件产品，从共享核心开始，到最多 128 核(编写本书时的数据，但这个数据会很快改变)。128 核、2TB 的 RAM 和 4TB 的本地 SSD 属于计算机的“M 系列”。

对于预装的操作系统，可以在 Windows、Windows Server、Linux 和预装了 SQL Server、BizTalk Server、SharePoint 和 Oracle 等的操作系统之间选择。

笔者经常给一周只需要几个小时的环境使用虚拟机，因为虚拟机按小时支付费用。如果想尝试在 Linux 上编译和运行 .NET Core 程序，但没有 Linux 计算机，在 Microsoft Azure 上安装这样一个环境是很容易的。

3. PaaS

对于开发人员来说，Microsoft Azure 最相关的部分是 PaaS。可以为存储和读取数据而访问服务，使用应用程序服务的计算和联网功能，在应用程序中集成开发者服务。

为了在云中存储数据，可以使用关系数据存储 SQL Database。SQL Database 与 SQL Server 的本地版本大致相同。也有一些 NoSQL 解决方案，例如，Cosmos DB 有不同的存储选项，如 JSON 数据、关系或表格存储，Azure Storage 存储 blob(如图像或视频)。

应用程序服务可以用于驻留通过 ASP.NET Core 创建的 Web 应用程序和 API 应用程序。

Microsoft 还在 Microsoft Azure 中提供了 Developer Services。Developer Services 的一部分是 Visual Studio Team Services。Visual Studio Team Services 允许管理源代码，自动构建、测试和部署 CI(持续集成)。

Developer Services 的另一部分是 Application Insights。它的发布周期更短，对于获得用户如何使用应用程序的信息越来越重要。用户因为可能找不到哪些菜单，而从未使用过它们？用户在应用程序中使用什么路径来完成任务？在 Application Insights 中，可以得到良好的匿名用户信息，找出用户关于应用程序的问题，使用 DevOps 可以快速解决这些问题。

还可以使用 Cognitive Services 提供的功能处理图像，使用 Bing Search APIs，利用语言服务理解用户的看法等。

4. FaaS

FaaS 是云服务的一个新概念，也称为无服务器计算技术。当然，幕后总是有一个服务器。不需要为保留的 CPU 和内存付费，就像在 Web 应用程序中使用的应用程序服务一样。相反，支付的金额是基于消费的，即对活动所需要的内存的调用次数和时间付费，且有一些限制。Azure 函数是一种可以使用 FaaS 进行部署的技术。

注意：

第 29 章介绍了跟踪特性以及如何使用 Microsoft Azure 的 Application Insights 产品。第 32 章不仅说明了如何使用 ASP.NET Core MVC 创建 Web API，也展示了如何在 Azure 函数中使用相同的服务功能。网上附加第 4 章介绍了 Microsoft Bot 服务和 Cognitive Services。

1.7 开发工具

第 2 章会讨论很多 C# 代码，而本章的最后一部分介绍开发工具和 Visual Studio 2017 的版本。

1.7.1 Visual Studio Community

这个版本的 Visual Studio 是免费的，具备以前 Professional 版的功能。使用时间有许可限制。它对开源项目和培训、学术和小型专业团队是免费的。Visual Studio Express 版本以前是免费的，但该产品允许在 Visual Studio 中使用扩展。

1.7.2 Visual Studio Professional

这个版本比 Community 版包括更多功能，例如 CodeLens 和 Team Foundation Server，来进行源代码管理和团队协作。有了这个版本，也会得到 MSDN 订阅，其中包括微软公司的几个服务器产品，用于开发和测试。

1.7.3 Visual Studio Enterprise

与 Professional 版不同，这个版本包含很多测试工具，如 Web 负载和性能测试、使用 Microsoft Fakes 进行单元测试隔离，以及编码的 UI 测试(单元测试是所有 Visual Studio 版本的一部分)。通过 Code Clone 可以找到解决方案中的代码克隆。Visual Studio Enterprise 版还包含架构和建模工具，以分析和验证解决方案体系结构。

注意：

有了 Visual Studio 订阅，就有权免费使用 Microsoft Azure，每月具体的数量视 MSDN 订阅的类型而定。

注意：

第 18 章详细介绍了 Visual Studio 2017 几个特性的使用。第 28 章阐述单元测试、Web 测试和创建编码的 UI 测试。

注意：

本书中的一些功能，如编码的 UI 测试，需要 Visual Studio Enterprise 版。使用 Visual Studio Community 版可以完成本书的大部分内容。

1.7.4 Visual Studio for Mac

Visual Studio for Mac 起源于 Xamarin Studio，但现在它提供的比之前的产品多得多。例如，编辑器与 Visual Studio 共享代码，因此用户很快就会熟悉它。有了 Visual Studio for Mac，不仅可以创建 Xamarin 应用程序，还可以创建在 Windows、Linux 和 Mac 上运行的 ASP.NET Core 应用程序。在本书的许多章节中，都可以使用 Visual Studio for Mac。但介绍 UWP 的章节例外，它要求用 Windows 运行和开发应用程序。

1.7.5 Visual Studio Code

与其他 Visual Studio 版本相比，Visual Studio Code 是一个完全不同的开发工具。Visual Studio 2017 提供了基于项目的特性以及一组丰富的模板和工具，而 Visual Studio Code 是一个代码编辑器，几乎不支持项目管理。然而，Visual Studio Code 不仅在 Windows 上运行，也在 Linux 和 OS X 上运行。

对于本书的许多章节，可以使用 Visual Studio Code 作为开发编辑器。但不能创建 UWP 或 Xamarin 应用程序，也无法获得第 18 章介绍的特性。Visual Studio Code 代码可以用于 .NET Core 控制台应用程序，以及使用 .NET Core 的 ASP.NET Core 1.0 Web 应用程序。

可以从 <http://code.visualstudio.com> 下载 Visual Studio Code。

1.8 小结

本章涵盖了很多重要的技术和技术的变化。了解一些技术的历史，有助于确定新的应用程序应该使用哪些

技术，现有的应用程序应该如何处理。

.NET Framework 和 .NET Core 是有差异的。本章讨论了如何在这两种环境中创建并运行 “Hello World!” 应用程序，但没有使用 Visual Studio。

本章阐述了公共语言运行库(CLR)的功能，介绍了用于访问数据库和创建 Windows 应用程序的技术。论述了 ASP.NET Core 的优点。

第 2 章开始讨论 C#语法，学习变量，实现程序流，把代码组织到名称空间中内容。