INTRODUCTION

尽管 .NET 在 2000 年被发布，但它没有变成一个过时的技术。相反，.NET 对开发者的吸引力持续增长，自从它开源并且不仅可以在 Windows 使用还能在 Linux 上使用。.NET 还可以在客户端的浏览器中运行——不需要下载扩展——通过使用 WebAssembly 标准。

随着对 C# 和 .NET 的新增强功能不断推出，其关注点不仅在于性能提升，还在于易用性。.NET 越来越成为新开发者的选择。

C# 对长期开发者同样有吸引力。每年，Stack Overflow 询问开发者他们的最爱，最害怕和最想要的变成语言和框架。C# 已经持续几年在“最爱的语言”排列前 10。ASP.NET Core 现在是“最爱的web框架”首位。.NET Core 在“最爱的其他框架/库/工具”类别中第一。详见 `https://insights.stackoverflow.com/survey/2020`

当你使用 C# 和 ASP.NET Core，你可以创建 web 应用程序和服务（包括微服务），运行在 Windows，Linux 和 Mac 上。你可以用 Windows 运行时去创建本地 Windows 程序，使用 C#，XAML 和 .NET。你可以创建类库，它可以在 ASP.Net Core，Windows 程序和 .Net MAUI 中共享。

大多数本书中的例子是构建和运行在 Windows 或 Linux 系统上的。例外情况是仅在 Windows 平台上运行的 Windows 应用程序示例。你可以使用 Visual Studio，Visual Studio Code，或 Visual Studio for the Mac 作为开发环境；只有 Windows 程序示例需要 Visual Studio。

.NET 的世界

.NET 有长的历史；第一个版本被发布于 2002 年。完全重写的新一代的 .NET（.NET Core 1.0 在 2016 年）是非常年轻的。最近，很多来自旧 .NET 版本的功能已经被带到了 .NET Core 去简化迁移体验。

当创建新的应用程序，没有理由不迁移到新的 .NET 版本。旧的程序是否应该呆在旧版本的 .NET 还是迁移到新版本取决于使用的功能，迁移的难度和迁移后你能获得什么优势。这个问题最好的选择需要通过逐个应用程序的分析来考虑。

新 .NET 提供了容易的方式去创建 Windows 和 web 程序和服务。你可以创建微服务在 Kubernetes 集群中的 Docker 容器中运行；创建 web 程序返回 HTML，JavaScript 和 CSS；创建 web 程序返回 HTML，JavaScript 和 .NET 二进制文件，他使用 WebAssembly 以安全且标准的方式运行在客户端浏览器里。你可以以传统的方式创建 Windows 程序使用 WPF 和 Windows Forms 并利用现代的 XAML 功能和控件，它们支持用 WinUI 实现流畅的设计，以及通过.NET MAUI 开发移动应用程序。

.NET 使用现代模式。依赖注入被构建进核心服务，比如 ASP.NET Core 和 EF Core，这不仅使单元测试更容易，而且允许开发人员容易的增强和更改这些技术的功能。

.NET 能运行在多平台上。除了 Windows 和 macOS，许多 Linux 环境也被支持，比如 Alpine，CentOS，Debian，Febora，openSUSE，Red Hat，SLES 和 Ubuntu。

.NET 是开源的（`https://github.com/dotnet`）并且免费获得。你可以找到 C# 编译器的会议记录（`https://github.com/dotnet/csharplang`），C# 编译器的源代码（`https://github.com/dotnet/Roslyn`），.NET 运行时和类库的源代码（`https://github.com/dotnet/runtime`），和 ASP.NET Core的源代码（`https://github.com/dotnet/aspnetcore`），包括 Razor Pages，Blazor 和 SignalR。

这里是一些对于新 .NEt 功能特点的总结：

- .NET 是开源的。
- .NET 使用现代模式。
- .NET 支持在多平台上开发
- ASP.NET Core 可以允许在 Windows 和 Linux 上。

C# 的世界

当 C# 在 2002 年被发布，他是一种为了 .NET Framework 开发的语言。C# 的设计思想来自 C++，Java 和 Pascal。Anders Hejlsberg 从 Borland 来到微软并带来了Delphi语言开发的经验。在 Microsoft，Hejlsberg 工作于 Microsoft 版的 Java，叫做 J++，在创造 C# 前。

注意 今天，Anders Hejlsberg 已经转移工作到 TypeScript（虽然他依然影响 C#），并且 Mads Torgersen 是 C# 的项目领头人。C# 的改进在 `https://github.com/dotnet/csharplang` 上公开讨论，而且你可以阅读 C# 语言的建议和会议记录。你也可以提交自己的 C# 建议。

C# 最初不仅是一门面向对象的通用编程语言，更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（包含元数据的二进制文件）。

随着时间的推移，C# 被增强，通过加入泛型、语言集成查询（LINQ）、lambda 表达式、动态特性以及更简便的异步编程。C# 不是一个简单的编程语言，因为它提供了很多特功能性，但它也在持续发展，不断加入实用的新功能。由此，C# 不仅仅是一门面向对象或基于组件的语言；他还包含了函数式编程的理念——这些对于开发各类应用的通用语言来说都具有实用价值。

如今，每年都会发布一个新版本的 C#，C# 8 加入了可空引用类型，C# 9加入了记录类型（records）等等，C# 10 和 .NET 6 在 2021 年发布，C# 11 和 .NET 7 在 2022 年发布。因为如今变更频繁，查看本书的 GitHub 仓库（详见“源代码”一节）以获取持续更新。

C# 的新东西

每年，一个新版本的 C# 被发布，伴随着许多新功能特性可以获得。最新的版本包括很多功能特性：可空引用类型，去减少 NullableReferenceException 异常，作为替换，让编译器提供更多帮助；提高生成效率的功能，如索引（indices）和范围（ranges）；switch 表达式，它可以让 switch 语句显得过时；using 声明等简化语法；模式匹配的提升。顶级语句允许使小型应用程序减少源代码的行数，记录类型（record）——一种类，可让编译器自动生成用于相等性比较、解构和 with 表达式的模板代码。代码生成器允许自动创建代码当编译器允许。所有这些新特性都在本书中涵盖。

ASP.NET CORE 的新东西

ASP.NET Core 现在包含新技术用于创建 web 应用：Blazor Server 和 Blazor WebAssembly。使用 Blazor，你有一个用 C# 代码编写客户端和服务端的全栈选择。使用 Blazor Server，你创建的包含 HTML 和 C# 代码的 Razor 组件会在服务器上运行。使用 Blazor WebAssembly，用 C# 和 HTML 编写的 Razor 组件会在客户端运行——借助 HTML5 标准的 WebAssembly 技术，该技术允许在浏览器中运行二进制代码，且为所有现代 Web 浏览器所支持。

对于创建服务，你现在可以使用 ASP.NET Core 的 gRPC，实现服务间的二进制通信。如果需要传输大量数据，这是服务到服务通信的绝佳选择，以减少带宽需要，同时降低 CPU 和内存占用。

0.1 Windows 的新东西

第一部分

C#语言

第一章　反射，元数据和源生成器

这章中有什么？

- 使用自定义特性
- 使用反射在运行时检查元数据
- 使用dynamic类型
- 用ExpandoObject创建动态对象
- 用源生成器编译代码

本章的代码下载

本章的源代码可在本书位于 `http://www.wiley.com` 的页面上获取。点击下载链接即可。你也可以在 `https://github.com/ProfessionalCSharp/ProfessionalCSharp2021` 的 1_CS/ReflectionAndSourceGenerators 目录下找到这些代码。

本章的代码分为以下几个主要示例：

- LookupWhatsNew
- TypeView
- VectorClass
- WhatsNewAttributes
- Dynamic
- DynamicFileReader
- CodeGenerationSample

所有的样例项目都启用了可空引用类型

1.1 运行时代码检查和动态编程

本节聚焦于自定义特性，反射，动态编程，和借助 C#9 源生成器在构建过程中的源代码生成。自定义特性是一些机制，它允许你去联系自定义元数据和程序元素。这些元数据在编译时被创建，并且被嵌入到一个程序集。反射是一个通用术语，它描述了在运行时检查和操控程序元素的能力。比如，反射允许你去做以下事情：

- 枚举一个类型中的成员
- 实例化一个新对象
- 执行一个对象中的成员
- 找出一个类型的信息
- 找出一个程序集的信息
- 检查被应用到一个类型的自定义特性
- 创建并编译一个新程序集

这个列表代表了很多的功能并且包含一些被 .NET 提供的最强大且最复杂的能力。因为一章中没有包含所有反射能力的空间，所以我聚焦于哪些你很可能去用的最频繁的元素。

为了去展示自定义特性和反射，在这章中，你将首先开发一个例子，基于一个频繁发布他的软件升级并且想要获得关于这些被自动记录的升级细节的公司。在这个例子中，你将定义指明程序元素被最后一次修改时的日期和被修改了什么的自定义特性。然后你使用反射开发一个程序，可以在一个程序集中寻找这些特性并且自动显示从给定日期开始的对软件升级的所有细节。

在这章中的另一个例子考虑了一个应用程序，它可以读出或写入一个数据库，并使用自定义特性标记哪些类或属性对应于哪些表或列。通过在运行时从程序集中读取这些特性，这个程序可以自动检索或写入数据到数据库中的恰当位置，而无需对于每个表或列编写特定逻辑。

这一章的第二个方面就是动态编程，自从 dynamic 类型被添加到 C#4，动态编程就成为 C# 语言的一部分。虽然 C# 是一个静态类型语言，但动态编程的加入给了 C# 语言从 C# 内部调用脚本函数的能力。

在本章中，你将了解 dynamic 类型及其使用规则。你还将看到 DynamicObject 的实现长什么样并且如何使用它。DynamicObject 的实现之一 ExpandoObject 也将被涵盖。

本章的第三个大方面是一个 C#9 的增强——源生成器。使用源生成器，代码可以在你开始构建流程时被生成。你写的源代码可以被增强，同时还可以利用其他数据源来生成 C# 源代码。在本章中，你将看到源生成器如何检查特性，从而在编译时生成代码，而无需在运行时依赖反射。

1.2 自定义特性

你已经在这本书中看到如何在你的程序中定义多个特性。这些特性已经被 Microsoft 作为 .NET 的一部分被定义，并且许多特性都受到了 C# 编译器的特殊支持。这意味着对于那行特定的特性，编译器可以按特定方式自定义编译过程——比如，根据 StructLayout 特性中的细节在内存中给结构体布局。

.NET 还允许你去定义特性。默认情况下自定义特性不会对编译过程有任何影响，因为编译器本身并不会识别它们（一会你将看到源生成器，在那儿自定义特性对编译过程有影响）。当这些特性应用于程序元素时，它们会被当作元数据输出到编译后的程序集中。

单独来看，这些元数据可能对文档编写有帮助，但让特性变得真正强大的是：通过使用反射，你的代码可以在运行时读取这些元数据并做出相应的决策。这意味着你定义的自定义特性可以直接影响你代码的运行方式。例如，对于自定义权限类，自定义特性可以被用于去允许声明式代码访问安全检查，自定义特性还可以去关联信息和程序元素，以便之后被测试工具使用，或者在开发允许加载插件或模块的可扩展框架时使用。

1.2.1 编写自定义特性

为了理解如何编写自定义特性，了解当编译器遇到一个你代码中的元素，并且这个元素带有一个被实现了的自定义特性时做了什么是有用的。假定你

有一个 C# 属性定义如下：

```
[FieldName("SocialSecurityNumber")]
public string SocialSecurityNumber
{
    get {
        // ...
```

编译器期望找到一个带有这个名字的类，并且期望这个类直接或间接继承自 System.Attribute。编译器还期望这个类包括了控制特性使用的信息。特别的，这个特性类需要具体说明一下内容：

- 可以应用该特性的程序元素的类型（类、结构体、属性、方法等）
- 将特性多次应用于同一程序元素是否合法
- 当特性应用于类或接口时，是否被派生类和子接口继承
- 特性用到的必选和可选参数

如果编译器不能找到一个相应的特性类或找到了但是你使用特性的方式不符合特性类的信息，编译器将引发编译错误。

继续这个例子，假定你已经定义了 FieldName 特性如下：

```
[AttributeUsage(AttributeTargets.Property,
    AllowMultiple=false, Inherited=false)]
public class FieldNameAttribute: Attribute
{
    private string _name;
    public FieldNameAttribute(string name) => _name = name;
}
```

下一节将讨论这个定义中的每一个元素

详述 AttributeUsage 特性

首先要注意的是特性类被标记有一个特性——System.AttributeUsage 特性。这是一个被微软定义的特性，用来给编译器提供特殊支持。AttributeUsage 的首要目的是去识别可以应用你自定义特性的程序元素的类型。这个信息

被 AttributeUsage 的第一个参数提供。这个参数是强制性的，并且是可枚举类型：AttributeTargets 。在之前的例子中，你已经指示了 FieldName 特性仅仅可以被应用与属性，这是好的，因为这正是你在前面的代码片段中应用它的地方。AttributeTargets 枚举类型定义了一些成员去应用特性到程序集，类，构造函数，字段，事件，方法，接口，结构体，返回值等等。

注意，当应用一个特性到你的程序元素时，你直接将特性放到元素前的方括号中。不过，之前列表中有两个值和任何程序元素都没有关联：Assembly 和 Module。特性可以作为一个整体应用于程序集或模块，而不是你代码中的元素；在这个情况下，特性可以被放到你源代码的任何地方，但是他必须有 assembly 或 module 前缀：

```
[assembly:SomeAssemblyAttribute(Parameters)]
[module:SomeAssemblyAttribute(Parameters)]
```

当指定有效的目标元素给自定义特性时，你可以用位运算符 OR 组合这些值。例如，如果你想去指示你的 FieldName 特性可以被应用到属性和字段上，你将使用下面代码：

```
[AttributeUsage(AttributeTargets.Property
    | AttributeTargets.Field,
    AllowMultiple=false, Inherited=false)]
public class FieldNameAttribute: Attribute
```

你也可以用 AttributeTargets.All 去表明你的特性可以被应用到所有程序元素类型。AttributeUsage 特性还包含了两个其他参数：AllowMultiple 和 Inherited。它们被规定使用 <ParameterName>=<ParameterValue> 语法而不是仅仅指定这些参数的值。这些参数是可选的——你可以省略他们。

AllowMultiple 参数表明了是否一个特性可以被多于一次的应用与同一项。当它被设为 false 时，编译器应该引发错误，当看到类似这样的情况：

```
[FieldName("SocialSecurityNumber")]
[FieldName("NationalInsuranceNumber")]
public string SocialSecurityNumber
{
    // ...
```

如果 Inherited 参数被设为 true，应用于类或接口的特性会自动应用于所有派生类或子接口。如果特性被应用与一个方法或属性，他将会自动应用于任何对那个方法或属性的重写（override），以此类推。

详述特性的参数

这一部分演示了你如何给自定义特性指定参数。当编译器遇到如下的语句时，它会检查传递给特性的参数——一个字符串——并且寻找该特性中能精确匹配这些参数的构造函数。

```
[FieldName("SocialSecurityNumber")]
public string SocialSecurityNumber
{
    // ...
```

如果编译器找到了一个匹配的构造函数，它就会将指定的元数据写入程序集。如果编译器没有找到一个匹配的构造函数，一个编译时错误将发生。如本章稍后所述，反射涉及从程序集读取元数据（特性），并实例化这些元数据所代表的特性类。因此，编译器必须保证存在一个匹配的构造函数，去允许指定特性在运行时实例化。

在这个例子中，你只提供了一个构造函数给 FieldNameAttribute，并且这个构造函数需要一个字符串参数。因此，当应用 FieldName 特性到一个属性时，你必须提供一个字符串作为一个参数，就像上面代码展示的那样。

为了去允许特性选择需要提供的参数类型，你可以提供不同的构造函数重载，尽管通常做法是只提供一个构造函数，并且使用属性去定义其他可选参数，就像接下来讲述的。

详述特性参数

就像 AttributeUsage 特性展示的那样，一个给特性添加可选参数的替代语法。这个语法需要指定可选参数的名字和值。他通过 public 属性或字段在特性类工作。例如，假定你修改 SocialSecurityNumber 属性的定义如下：

```
[FieldName("SocialSecurityNumber",
```

```
    Comment="This is the primary key field")]
public string SocialSecurityNumber { get; set; }
{
    // ...
```

在这个例子中，对于第二个参数，编译器识别 <ParameterName>=<ParameterValue> 语法，并且不会试图去匹配这个参数到 FieldNameAttribute 构造函数。相反，它寻找一个对应名字的 public 属性或字段（公共字段并不被认为是好的编程实践，所以你通常使用属性）用来设置这个参数的值。如果你想让之前的代码工作，你必须添加一些代码到 FieldNameAttribute：

```
[AttributeUsage(AttributeTargets.Property,
    AllowMultiple=false, Inherited=false)]
public string FieldNameAttribute : Attribute
{
    public string Comment { get; set; }
    private string _fieldName;
    public FieldNameAttribute(string fieldName)
    {
        _fieldName = fieldName;
    }
    // ...
}
```

1.2.2 自定义特性例子：WhatsNewAttributes

在这部分中，你开始开发本章最开始提到的例子。WhatsNewAttributes 提供了一个用于表明程序元素最后修改时间的特性。这个例子的代码比其他例子更加复杂，因为它由三个独立项目组成：

- WhatsNewAttributes——这个类库包含了特性类的定义：LastModifiedAttribute 和 SupportsWhatsNewAttribute。
- VectorClass——这个类库使用了自定义特性。类型和成员都会通过这些特性进行标注。
- LookupWhatsNew——这个可执行程序通过反射读取这些特性。

WhatsNewAttributes 类库

这部分开始核心的 WhatsNewAttributes .NET 类库。