**<https://msdn.microsoft.com/zh-cn/magazine/mt707534.aspx>**

**必备 .NET - 使用 .NET Core 实现依赖关系注入**

作者 [Mark Michaelis](https://msdn.microsoft.com/zh-cn/magazine/mt149362?author=mark+michaelis) | 2016 年 6 月 | [获取代码](http://GitHub.com/IntelliTect/Articles)

在我的前两篇文章（“使用 .NET Core 的日志记录”([msdn.com/magazine/mt694089](http://msdn.com/magazine/mt694089)) 和“.NET Core 中的配置”([msdn.com/magazine/mt632279](http://msdn.com/magazine/mt632279))）中，我演示了如何从 ASP.NET Core 项目 (project.json) 和更常见的 .NET 4.6 C# 项目 (\*.csproj) 中利用 .NET Core 功能。换句话说，利用新框架并不仅限于编写 ASP.NET Core 项目的那些人。在本专栏中，我将继续深入探讨 .NET Core，并重点探讨 .NET Core 依赖关系注入 (DI) 功能，以及如何利用这些功能启用控制反转 (IoC) 模式。如前所述，可以从“传统的”CSPROJ 文件和新兴的 project.json 类型的项目中利用 .NET Core 功能。对于示例代码，这一次我会使用来自 project.json 项目的 XUnit。

**为什么使用依赖关系注入？**

使用 .NET，通过 new 运算符（即，new MyService 或任何想要实例化的对象类型）调用构造函数即可轻松实现对象实例化。遗憾的是，此类调用会强制实施客户端（或应用程序）代码到已实例化对象的紧密耦合的连接（硬编码的引用），此外还会引用其程序集/NuGet 包。对于常见的 .NET 类型而言，这不是问题。然而，对于提供“服务”（如日志记录、配置、支付、通知或事件 DI）的类型，如果你想切换所用服务的实现，则可能不需要依赖关系。例如，一种方案是，客户端可能将 NLog 用于日志记录，而另一种方案是，客户端可能选择 Log4Net 或 Serilog。而且，使用 NLog 的客户端不喜欢使用 Serilog 打乱其项目，因此，同时引用两种日志记录服务不会令人满意。

为了解决对服务实现的引用进行硬编码的问题，DI 提供了一个间接层，这样与其直接使用 new 运算符实例化服务，倒不如客户端（或应用程序）请求实例的服务集或“工厂”。此外，与其请求特定类型的服务集（例如创建一个紧密耦合的引用），倒不如请求一个接口（如 ILoggerFactory），并期待服务提供程序（本例中为 NLog、Log4Net 或 Serilog）实现该接口。

结果是，当客户端直接引用抽象程序集 (Logging.Abstractions) 时，会同时定义服务接口­，将不需要引用直接实现。

我们将解耦返回到客户端的实际实例的模式称为控制反转。这是因为，与其客户端确定要实例化的对象，就像使用 new 运算符显式调用构造函数时一样，倒不如 DI 确定将返回的内容。DI 注册了由客户端请求的类型（一般为接口）和将返回的类型之间的关联。此外，DI 通常会确定已返回类型的生存期，具体取决于该类型的所有请求之间将有单个共享的实例、每个请求将各有一个新实例，还是介于两者之间。

对 DI 的一个尤为常见的需求体现在单元测试中。考虑相应地取决于付款服务的购物车服务。假设编写利用付款服务的购物车服务，并尝试对购物车服务进行单元测试，而不实际调用真实的付款服务。相反，你想调用的是模拟付款服务。为了使用 DI 实现此目的，你的代码会从 DI 框架请求付款服务接口的实例而不是调用，例如，new PaymentService。然后，只需为单元测试“配置”DI 框架，以返回一个模拟付款服务。

相比之下，生产主机可以配置购物车，以使用（可能很多）付款服务选项之一。也许最重要的是，引用将仅针对付款抽象，而不是针对每个具体的实现。

提供“服务”的实例而不是使客户端直接将其实例化是 DI 的基本原则。事实上，一些 DI 框架允许通过支持基于配置和反射的绑定机制（而不是编译时绑定）从引用实现中对主机进行解耦。这种解耦称为服务定位器模式。

**.NET Core Microsoft.Extensions.DependencyInjection**

若要利用 .NET Core DI 框架，你只需引用 Microsoft.Extnesions.DependencyInjection.Abstractions NuGet 包。此包提供了 IServiceCollection 接口的入口，从而公开你可以从中调用 GetService<TService> 的 System.IService­Provider。类型参数 TService 标识要检索的服务的类型（一般为接口），如下应用程序代码获得了一个实例：

ILoggingFactory loggingFactor = serviceProvider.GetService<ILoggingFactory>();

有一些相应的非泛型 GetService 方法将 Type 作为参数（而不是泛型参数）。泛型方法允许直接分配给特定类型的变量，而非泛型版本需要一个显式转换，因为返回类型为 Object。此外，当添加该服务类型时，会有泛型约束，因此使用该类型参数时可以完全避免转换。

如果在调用 GetService 时没有使用收集服务注册任何类型，它将返回 null。这在与 null 传播运算符结合以将可选行为添加到应用时非常有用。类似的 GetRequiredService 方法在没有注册服务类型时会抛出异常。

如你所见，代码非常简单。然而，现在缺少的是如何获得在其上调用 GetService 的服务提供程序的实例。解决方案是首先实例化 ServiceCollection 的默认构造函数，然后再注册你想要服务提供的类型。**图 1** 中显示了一个示例，你可以假设其中的每个类（Host、Application 和 PaymentService）已在单独的程序集中实现。此外，尽管 Host 程序集知道要使用哪个记录器，但是没有在 Application 或 PaymentService 中引用记录器。同样，Host 程序集没有引用 PaymentServices 程序集。接口也在单独的“抽象”程序集中实现了。例如，ILogger 接口是在 Microsoft.Extensions.Logging.Abstractions 程序集中定义的。

图 1 注册和请求来自依赖关系注入的对象

public class Host

{

  public static void Main()

  {

    IServiceCollection serviceCollection = new ServiceCollection();

    ConfigureServices(serviceCollection);

    Application application = new Application(serviceCollection);

    // Run

    // ...

  }

  static private void ConfigureServices(IServiceCollection serviceCollection)

  {

    ILoggerFactory loggerFactory = new Logging.LoggerFactory();

    serviceCollection.AddInstance<ILoggerFactory>(loggerFactory);

  }

}

public class Application

{

  public IServiceProvider Services { get; set; }

  public ILogger Logger { get; set; }

    public Application(IServiceCollection serviceCollection)

  {

    ConfigureServices(serviceCollection);

    Services = serviceCollection.BuildServiceProvider();

    Logger = Services.GetRequiredService<ILoggerFactory>()

            .CreateLogger<Application>();

    Logger.LogInformation("Application created successfully.");

  }

  public void MakePayment(PaymentDetails paymentDetails)

  {

    Logger.LogInformation(

      $"Begin making a payment { paymentDetails }");

    IPaymentService paymentService =

      Services.GetRequiredService<IPaymentService>();

    // ...

  }

  private void ConfigureServices(IServiceCollection serviceCollection)

  {

    serviceCollection.AddSingleton<IPaymentService, PaymentService>();

  }

}

public class PaymentService: IPaymentService

{

  public ILogger Logger { get; }

  public PaymentService(ILoggerFactory loggerFactory)

  {

    Logger = loggerFactory?.CreateLogger<PaymentService>();

    if(Logger == null)

    {

      throw new ArgumentNullException(nameof(loggerFactory));

    }

    Logger.LogInformation("PaymentService created");

  }

}

从概念上讲，可以将 ServiceCollection 类型认为是名称/值对，其中名称是稍后将要检索的对象的类型（一般为接口），而值是实现接口的类型或用于检索该类型的算法（委托）。因此，在**图 1** 的 Host.Configure­Services 方法中调用 AddInstance 可注册 ILoggerFactory 类型的任何请求，该类型返回在 ConfigureServices 方法中创建的相同 LoggerFactory 实例。因此，Application 和 PaymentService 均可以检索 ILoggerFactory，而无需了解实现和配置记录器的知识（或程序集/NuGet 引用）。同样，Application 提供 MakePayment 方法，无需了解关于要使用的付款服务的知识。

请注意，ServiceCollection 不直接提供 GetService 或 GetRequiredService 方法。而是由 ServiceCollection.BuildServiceProvider 方法返回的 IServiceProvider 提供这些方法。此外，仅由提供程序提供的服务是调用 BuildServiceProvider 之前添加的服务。

Microsoft.Framework.DependencyInjection.Abstractions 还包括称为 ActivatorUtilities 的静态帮助程序类，该类提供了一些有用的方法，用于处理未使用 IServiceProvider（自定义的 ObjectFactory 委托）注册的构造函数参数，或者在想要创建默认实例的情况下，调用 GetService 时返回 null（请参阅 [bit.ly/1WIt4Ka#ActivatorUtilities](http://bit.ly/1WIt4Ka#ActivatorUtilities)）。

**服务生存期**

在**图 1** 中，我调用了 IServiceCollection AddInstance<TService>(TService implementationInstance) 扩展方法。Instance 是 .NET Core DI 附带的四个不同的 TService 生存期选项之一。它规定不仅 GetService 的调用将返回 TService 类型的对象，而且将返回使用 AddInstance 注册的特定 implementationInstance 实例。换句话说，使用 AddInstance 进行注册可以保存特定的 implementationInstance 实例，因此每次使用 AddInstance 方法的 TService 类型参数调用 GetService（或 GetRequiredService）时均可以返回该实例。

相反，IServiceCollection AddSingleton<TService> 扩展方法没有实例参数，而是依赖于通过构造函数进行实例化的 TService。默认的构造函数有效，Microsoft.Extensions.Dependency­Injection 也支持注册了参数的非默认构造函数。例如，你可以调用：

IPaymentService paymentService = Services.GetRequiredService<IPaymentService>()

而且，在实例化需要其构造函数中的 ILoggingFactory 的 PaymentService 类时，DI 将负责检索具体的 ILoggingFactory 实例并利用该实例。

如果 TService 类型中没有此类方法可用，则可以重载 AddSingleton 扩展方法，该方法采用了 Func<IServiceProvider, TService> implementationFactory（用于实例化 TService 的工厂方法）类型的委托。无论你是否提供工厂方法，服务收集实现都会确保将仅创建一个 TService 类型的实例，从而确保存在单一实例。在第一次调用触发 TService 实例的 GetService 后，在服务收集的生存期内将始终返回同一实例。

IServiceCollection 还包括 AddTransient(Type serviceType, Type implementationType) 和 AddTransient(Type serviceType, Func<IServiceProvider, TService> implementationFactory) 扩展方法。这些方法类似于 AddSingleton，不同的是每次调用这些方法时都会返回一个新实例，从而确保你始终拥有 TService 类型的新实例。

最后，有几个 AddScoped 类型的扩展方法。这些方法设计为在给定的上下文中返回同一实例，并且每当上下文（也称为作用域）更改时都会创建新实例。从概念上讲，ASP.NET Core 的行为映射到作用域生存期。从本质上讲，新实例是针对每个 HttpContext 实例创建的，而且每当在相同的 HttpContext 内调用 GetService 时，都会返回完全相同的 TService 实例。

总之，有四个生存期选项，用于从服务收集实现返回的对象： Instance、Singleton、Transient 和 Scoped。最后三个是在 ServiceLifetime 枚举中定义的 ([bit.ly/1SFtcaG](http://bit.ly/1SFtcaG))。但是，缺少 Instance，因为它是 Scoped（在其中无法更改上下文）的特殊用例。

之前我提到过 ServiceCollection 在概念上就像一个名称/值对，它将 TService 类型用于查找。ServiceCollection 类型的实际实现在 ServiceDescription 类中完成（请参阅 [bit.ly/1SFoDgu](http://bit.ly/1SFoDgu)）。该类为实例化 TService（即，ServiceType (TService)）、Implementation­Type 或 ImplementationFactory 委托以及 ServiceLifetime 所需的信息提供了一个容器。除了 ServiceDescriptor 构造函数，ServiceDescriptor 上还有许多静态工厂方法，可帮助实例化 ServiceDescriptor 本身。

无论使用哪种生存期注册 TService，TService 本身必须是一个引用类型，而不是值类型。每当你将类型参数用于 TService（而不是作为参数传递 Type）时，编译器都会使用泛型类约束进行验证。然而，编译器不会验证是否使用的是对象类型 TService。你一定要避免这种情况，以及任何其他非独特的接口（或许如 IComparable）。原因是，如果你注册了对象类型的内容，无论你在 GetService 调用中指定哪种类型的 TService，将始终返回注册为 TService 类型的对象。

**DI 实现的依赖关系注入**

ASP.NET 利用 DI 的程度之深，事实上，你可以在 DI 框架本身内实现 DI。换句话说，你不限于使用在 Microsoft.Extensions.DependencyInjection 中发现的 DI 机制的 ServiceCollection 实现。相反，只要你有实现 IServiceCollection（在 Microsoft.Extensions.DependencyInjection.Abstractions 中定义，请参阅 [bit.ly/1SKdm1z](http://bit.ly/1SKdm1z)）或IServiceProvider（在 .NET Core lib 框架的 System 命名空间内定义）的类，你就可以替代自己的 DI 框架或利用另外一个完善的 DI 框架，其中包括 Ninject（[ninject.org](http://ninject.org/)，经过数年的努力维护 @IanfDavis 呼之欲出）和 Autofac ([autofac.org](http://autofac.org/))。

**浅谈 ActivatorUtilities**

Microsoft.Framework.DependencyInjection.Abstractions 还包括静态帮助程序类，该类提供了一些有用的方法，用于处理未使用 IServiceProvider（自定义的 ObjectFactory 委托）注册的构造函数参数，或者在想要创建默认实例的情况下，调用 GetService 时返回 null。你可以找到一些在 MVC 框架和 SignalR 库中使用此实用工具类的示例。在第一种情况下，存在一个带有 CreateInstance<T>(IServiceProvider provider, params object[] parameters) 签名的方法，允许你针对未注册的参数使用 DI 框架将构造函数参数传入到注册的类型中。你可能还会有性能需求，lambda 函数需要生成已编译的 lambda 类型。返回 ObjectFactory 的 CreateFactory(Type instanceType, Type[] argumentTypes) 方法在这种情况下可能有用。第一个参数是用户寻求的类型，而第二个参数是所有的构造函数类型，以匹配你希望使用的第一个类型的构造函数。在其[实现](https://github.com/aspnet/DependencyInjection/blob/1.0.0-beta4/src/Microsoft.Framework.DependencyInjection.Interfaces/ActivatorUtilities.cs" \l "L71)中，这些片段都精简到已编译的 lambda，多次调用后，性能会相当高。最后，GetServiceOrCreateInstance<T>(IServiceProvider provider) 方法提供了一个简单方式，用于提供可能已选择在其他地方注册的类型的默认实例。这在调用之前允许 DI 的情况下尤为有用，但是，如果未发生这种情况，你会获得一个回退实现。

**总结**

与 .NET Core 日志记录和配置一样，.NET Core DI 机制提供了一个相对简单的功能实现。虽然你不可能找到其他一些框架的更高级的 DI 功能，但 .NET Core 版本是轻量级的，并且是一个很好的入门方式。此外（再如日志记录和配置），.NET Core 实现可以被一个更成熟的实现替代。因此，你可能会考虑利用 .NET Core DI 框架作为一个“包装器”，通过它，将来你可以根据需要插入其他 DI 框架。通过这种方式，你不必定义自己的“自定义”DI 包装器，但可以利用 .NET Core 的包装器作为标准，任何客户端/应用程序都可以为标准的包装器插入自定义的实现。

关于 ASP.NET Core 需要注意的是，它自始至终都在利用 DI。这无疑是一个重大实践，在单元测试中尝试替代库的模拟实现时，如果你需要它，它会尤为重要。缺点是，并非简单的调用带有 new 运算符的构造函数，DI 注册和 GetService 调用的复杂性是必要的。我不禁想知道，C# 语言是否可以简化这种复杂性，但是，基于目前的 C# 7.0 设计，要实现这一点并不容易。

**Mark Michaelis** *是 IntelliTect 的创始人，担任首席技术架构师和培训师。在近二十年的时间里，他一直是 Microsoft MVP，并且自 2007 年以来一直担任 Microsoft 区域总监。Michaelis 还是多个 Microsoft 软件设计评审团队（包括 C#、Microsoft Azure、SharePoint 和 Visual Studio ALM）的成员。他在开发者会议上发表了演讲，并撰写了大量书籍，包括最新的“必备 C# 6.0（第 5 版）”(itl.tc/­EssentialCSharp)。可通过他的 Facebook* [*facebook.com/Mark.Michaelis*](http://facebook.com/Mark.Michaelis)*、博客* [*IntelliTect.com/Mark*](http://intellitect.com/Mark)*、Twitter* [*@markmichaelis*](https://twitter.com/@markmichaelis) *或电子邮件* [*mark@IntelliTect.com*](mailto:mark@IntelliTect.com) *与他取得联系。*

感谢以下 IntelliTect 技术专家对本文的审阅： Kelly Adams、Kevin Bost、Ian Davis 和 Phil Spokas

<a href="http://ox-d.101m3.com/w/1.0/rc?cs=4f2320cbb2378&amp;cb=INSERT\_RANDOM\_NUMBER\_HERE"> <img alt="" border="0" src="https://ox-d.101m3.com/w/1.0/ai?auid=139281&amp;cs=4f2320cbb2378&amp;cb=INSERT\_RANDOM\_NUMBER\_HERE" xmlns="http://www.w3.org/1999/xhtml" /> </a>

<a href="http://ox-d.101m3.com/w/1.0/rc?cs=4f2320971b57d&amp;cb=INSERT\_RANDOM\_NUMBER\_HERE"> <img alt="" border="0" src="http://ox-d.101m3.com/w/1.0/ai?auid=139280&amp;cs=4f2320971b57d&amp;cb=INSERT\_RANDOM\_NUMBER\_HERE" xmlns="http://www.w3.org/1999/xhtml" /> </a>

<a href="http://ox-d.101m3.com/w/1.0/rc?cs=d364f7da2e&amp;cb=INSERT\_RANDOM\_NUMBER\_HERE"> <img alt="" border="0" src="https://ox-d.101m3.com/w/1.0/ai?auid=538012782&amp;cs=d364f7da2e&amp;cb=INSERT\_RANDOM\_NUMBER\_HERE" xmlns="http://www.w3.org/1999/xhtml" /> </a>

[**MSDN Magazine Blog**](http://blogs.msdn.com/msdnmagazine/)

[14 Top Features of Visual Basic 14: The Q&A](https://blogs.msdn.microsoft.com/msdnmagazine/2015/01/07/14-top-features-of-visual-basic-14-the-qa/)

Wednesday, 1月 7

[Big Start to the New Year at MSDN Magazine](https://blogs.msdn.microsoft.com/msdnmagazine/2015/01/02/big-start-to-the-new-year-at-msdn-magazine/)

Friday, 1月 2

[More MSDN Magazine Blog entries >](http://blogs.msdn.com/msdnmagazine/)

Current Issue

[](https://msdn.microsoft.com/zh-cn/magazine/mt846709)

[Browse All MSDN Magazines](https://msdn.microsoft.com/zh-cn/magazine/ee310108.aspx)

[[Subscribe to the MSDN Flash newsletter](https://msdn.microsoft.com/en-us/aa570311.aspx?ocid=msdn_magazine)Subscribe to MSDN Flash newsletter](https://msdn.microsoft.com/en-us/aa570311.aspx?ocid=msdn_magazine)

Receive the MSDN Flash e-mail newsletter every other week, with news and information personalized to your interests and areas of focus.

<a href="http://ox-d.101m3.com/w/1.0/rc?cs=e4360aeffd&amp;cb=INSERT\_RANDOM\_NUMBER\_HERE"> <img alt="" border="0" src="https://ox-d.101m3.com/w/1.0/ai?auid=538012781&amp;cs=e4360aeffd&amp;cb=INSERT\_RANDOM\_NUMBER\_HERE" xmlns="http://www.w3.org/1999/xhtml" /> </a>