# Autofac 8.0 文档 - 中文版

# Autofac 介绍

![fig:](data:text/html; charset=utf-8;base64,PCFET0NUWVBFIEhUTUwgUFVCTElDICItLy9JRVRGLy9EVEQgSFRNTCAyLjAvL0VOIj48aHRtbD48aGVhZD48dGl0bGU+NDAzIEZvcmJpZGRlbjwvdGl0bGU+PC9oZWFkPjxib2R5PjxoMT40MDMgRm9yYmlkZGVuPC9oMT48cD5Zb3UgZG9uJ3QgaGF2ZSBwZXJtaXNzaW9uIHRvIGFjY2VzcyB0aGUgVVJMIG9uIHRoaXMgc2VydmVyLjwvcD48cD5kZW5pZWQgYnkgVUEgQUNMID0gYmxhY2tsaXN0PC9wPjxoci8+UG93ZXJlZCBieSBUZW5naW5lIDxici8+Q0ROIFJlcXVlc3QgSWQ6IGRkODJjMWE0MTcyNjQ5MTU4ODA0Mjc0NDZlPC9ib2R5PjwvaHRtbD4=)

一个令人上瘾的 .NET IoC 容器。

Autofac 是 Microsoft .NET 的 IoC 容器。它管理类之间的依赖关系，以便应用程序随着规模和复杂性的增长而易于更改。这是通过将常规的 .NET 类视为组件来实现的。

# Autofac 快速入门

将 Autofac 集成到应用程序的基本模式如下：

* 考虑采用控制反转（IoC）进行应用架构设计。
* 添加 Autofac 引用。
* 在应用程序启动时...
  + 创建一个 ContainerBuilder。
  + 注册组件 (register component)
  + 构建容器并存储以备后续使用。
* 在应用程序执行期间...
  + 从容器创建一个生命周期作用域。
  + 使用生命周期作用域来解析组件实例。
* Structure your app with *inversion of control* (IoC) in mind.
* Add Autofac references.
* At application startup…
  + Create a **ContainerBuilder**
  + Register components.
  + Build the container and store it for later use.
* During application execution…
  + Create a lifetime scope from the container.
  + Use the lifetime scope to resolve instances of the components.

这个入门指南会逐步引导你完成一个简单的控制台应用程序。掌握了基础知识后，你可以查看其他部分，了解更高级的用法和针对不同类型应用程序（如 WCF、ASP.NET 等）的 集成。

## 应用程序结构

控制反转的核心思想是，而不是让应用程序中的类彼此关联并让类“new”它们的依赖项，而是反转这一过程，使得依赖项在类构造时传递进来。Martin Fowler 写了 [一篇很好的文章](http://martinfowler.com/articles/injection.html) 解释了依赖注入/控制反转， 如果你对此感兴趣，可以阅读这篇文章。

对于我们的示例应用，我们将定义一个写当前日期的类。但不想将其绑定到 Console，因为我们希望以后能测试该类或在没有控制台的地方使用它。

我们甚至可以进一步抽象出写入操作，以便将来轻松替换一个写明天日期的版本。

我们会这样做：

using System;  
  
namespace DemoApp  
{  
 // 这个接口帮助解耦“写入输出”的概念与 Console 类。我们并不关心写入操作如何实现，只关心能否写入。  
 public interface IOutput  
 {  
 void Write(string content);  
 }  
  
 // 这个 IOutput 接口的实现实际上就是向 Console 写入。技术上讲，我们也可以实现 IOutput 来写入 Debug 或 Trace... 或任何其他地方。  
 public class ConsoleOutput : IOutput  
 {  
 public void Write(string content)  
 {  
 Console.WriteLine(content);  
 }  
 }  
  
 // 这个接口解耦了写入日期的概念与实际执行写入的操作。与 IOutput 类似，过程被一个接口抽象出来。  
 public interface IDateWriter  
 {  
 void WriteDate();  
 }  
  
 // 这个 TodayWriter 就是这一切的结合点。注意它接受一个 IOutput 构造参数 - 这使得根据实现，写入器可以写入任何地方。此外，它实现了 WriteDate 方法，使得今天日期会被写出来；你可以有一个写入不同格式或日期的版本。  
 public class TodayWriter : IDateWriter  
 {  
 private IOutput \_output;  
 public TodayWriter(IOutput output)  
 {  
 this.\_output = output;  
 }  
  
 public void WriteDate()  
 {  
 this.\_output.Write(DateTime.Today.ToShortDateString());  
 }  
 }  
}

现在我们有了一个相当合理（尽管有些牵强）的依赖关系集合，让我们开始使用 Autofac！

## 添加 Autofac 引用

第一步是在项目中添加 Autofac 引用。在这个示例中，我们只使用核心的 Autofac。[其他应用程序类型可能需要额外的 Autofac 集成库](https://autofac.readthedocs.io/en/latest/integration/index.html)。

![fig:](data:text/html; charset=utf-8;base64,PCFET0NUWVBFIEhUTUwgUFVCTElDICItLy9JRVRGLy9EVEQgSFRNTCAyLjAvL0VOIj48aHRtbD48aGVhZD48dGl0bGU+NDAzIEZvcmJpZGRlbjwvdGl0bGU+PC9oZWFkPjxib2R5PjxoMT40MDMgRm9yYmlkZGVuPC9oMT48cD5Zb3UgZG9uJ3QgaGF2ZSBwZXJtaXNzaW9uIHRvIGFjY2VzcyB0aGUgVVJMIG9uIHRoaXMgc2VydmVyLjwvcD48cD5kZW5pZWQgYnkgVUEgQUNMID0gYmxhY2tsaXN0PC9wPjxoci8+UG93ZXJlZCBieSBUZW5naW5lIDxici8+Q0ROIFJlcXVlc3QgSWQ6IGRkODJjMWIwMTcyNjQ5MTU4ODIwMDg4NzBlPC9ib2R5PjwvaHRtbD4=)

最简单的方法是通过 NuGet。"Autofac" package 包含了你所需的全部核心功能。

## 应用程序启动

在应用程序启动时，你需要创建一个 ContainerBuilder 并使用它注册你的 组件。组件是一个表达式、.NET 类型或其他代码片段，它公开一或多个 服务 并可以接收其他 依赖项。

简单来说，想象一个实现了接口的 .NET 类，例如：

public class SomeType : IService  
{  
}

你可以使用以下两种方式之一来引用该类型：

* 作为类型本身，SomeType
* 作为接口，IService

在这种情况下，组件是 SomeType，它公开的服务是 SomeType 和 IService。

在 Autofac 中，你会像这样使用 ContainerBuilder 注册它：

// 创建你的 builder 。  
var builder = new ContainerBuilder();  
  
// 通常你只对通过其接口暴露类型感兴趣：  
builder.RegisterType<SomeType>().As<IService>();  
  
// 但是，如果你想同时提供这两种服务（不那么常见）  
// 你可以这样说：  
builder.RegisterType<SomeType>().AsSelf().As<IService>();

对于我们的示例应用，我们需要注册所有组件（类）并暴露它们的服务（接口），以便所有东西都能很好地连接起来。

我们还需要存储容器，以便稍后使用它来解析类型。

using System;  
using Autofac;  
  
namespace DemoApp  
{  
 public class Program  
 {  
 private static IContainer Container { get; set; }  
  
 static void Main(string[] args)  
 {  
 var builder = new ContainerBuilder();  
 builder.RegisterType<ConsoleOutput>().As<IOutput>();  
 builder.RegisterType<TodayWriter>().As<IDateWriter>();  
 Container = builder.Build();  
  
 // WriteDate 方法是我们将利用依赖注入的地方。稍后我们会定义这个方法。  
 WriteDate();  
 }  
 }  
}

现在我们有了一个包含所有组件的 容器，并且它们已经正确地公开了 服务。让我们使用它吧。

## 应用程序执行

在应用程序执行期间，你需要使用已注册的组件。你通过从一个 生命周期作用域 中 解析 它们来做到这一点。容器本身就是一个生命周期作用域，你可以直接从容器解析东西。然而，不建议直接从容器解析。

当你解析一个组件时，根据定义的 实例作用域，会为该对象创建一个新的实例。（解析组件大致相当于调用 new 来实例化一个类。这非常简化了，但从类比的角度来看是可以的。）一些组件可能需要释放（例如，它们实现了 IDisposable）- Autofac 可以在作用域释放时为你处理这些组件的释放。

但是，容器会存在整个应用程序的生命周期。如果你直接从容器解析很多东西，可能会有很多东西等待释放，这可能不是好事（而且你可能会看到 “内存泄漏”）。相反，从容器创建一个 子生命周期作用域，然后从中解析。当你完成解析组件时，释放子作用域，所有东西都会为你清理干净。

（当你使用 Autofac 集成库 时，子作用域的创建在很大程度上为你处理，你无需考虑。）

对于我们的示例应用，我们将实现 WriteDate 方法，从作用域中获取写入器，并在完成后释放作用域。

namespace DemoApp  
{  
 public class Program  
 {  
 private static IContainer Container { get; set; }  
  
 static void Main(string[] args)  
 {  
 // ...前面看到的内容...  
 }  
  
 public static void WriteDate()  
 {  
 // 创建作用域，解析 IDateWriter，使用它，然后释放作用域。  
 using (var scope = Container.BeginLifetimeScope())  
 {  
 var writer = scope.Resolve<IDateWriter>();  
 writer.WriteDate();  
 }  
 }  
 }  
}

现在当你运行程序时...

* WriteDate 方法从作用域中创建一个生命周期作用域，以便避免任何内存泄漏 - 如果 IDateWriter 或其依赖项是可释放的，它们会在作用域释放时自动释放。
* WriteDate 方法手动从生命周期作用域中解析一个 IDateWriter。（这是 “服务定位” 。）内部...
  + Autofac 观察到 IDateWriter 映射到 TodayWriter，所以开始创建一个 TodayWriter。
  + Autofac 观察到 TodayWriter 的构造函数需要一个 IOutput。 ( 这是 “构造注入” 。 )
  + Autofac 观察到 IOutput 映射到 ConsoleOutput，所以创建了一个新的 ConsoleOutput 实例。
  + Autofac 使用新创建的 ConsoleOutput 实例完成 TodayWriter 的构造。
  + Autofac 返回完全构造的 TodayWriter 给 WriteDate 使用。
* 对于 writer.WriteDate() 的调用，会去调用 TodayWriter.WriteDate()，因为这就是解析的结果。
* Autofac 生命周期作用域被释放。从该生命周期作用域解析的所有可释放项也会被释放。

稍后，如果你想让应用程序写不同的日期，你可以实现不同的 IDateWriter，然后在启动时更改注册。你不必更改其他任何类。太棒了，控制反转！

**注意：**一般来说，服务定位通常被认为是反模式 [(参见文章)](https://blog.ploeh.dk/2010/02/03/ServiceLocatorIsAnAntiPattern.aspx)。也就是说，在代码中到处创建作用域并在每个地方散播对容器的使用并不是最佳做法。使用 Autofac 集成库 通常你不必像上面示例应用中那样做。相反，你通常会在应用的中央、顶层 位置进行依赖注入，手动解析的情况非常少见。当然，如何设计你的应用取决于你自己。

## 更进一步

示例应用演示了如何使用 Autofac，但你可以做的更多：

* 查看 [集成库](https://www.koudingke.cn/docs/zh-Hans/autofac-docs/latest/Integration/Index) 列表 ，了解如何将 Autofac 与你的应用集成。
* 学习 [注册组件的方式](https://www.koudingke.cn/docs/zh-Hans/autofac-docs/latest/Register/Index) ，以增加灵活性。
* 学习 [Autofac 配置选项](https://www.koudingke.cn/docs/zh-Hans/autofac-docs/latest/Configuration/Index) ，以便更好地管理组件注册。

## 需要帮助？

* 你可以在 [StackOverflow](https://stackoverflow.com/questions/tagged/autofac) 上提问。
* 可以参与 [Autofac Google 组](https://groups.google.com/forum/#forum/autofac) 讨论。
* 我们在 CodeProject 上有一个 [Autofac 教程](http://www.codeproject.com/KB/architecture/di-with-autofac.aspx) - by Nicholas Blumhardt。
* 如果你想深入了解，可以查看 [高级调试技巧](https://www.koudingke.cn/docs/zh-Hans/autofac-docs/latest/Troubleshooting/Index) 。

## 从源代码构建

源代码和 Visual Studio 项目文件可以在 [GitHub](https://github.com/autofac/Autofac) 上找到。构建说明和贡献指南请参阅 **贡献者指南** 。

# Autofac 注册概念

通过创建一个 ContainerBuilder 并告知 builder 哪些[组件](https://docs.autofac.org/en/latest/glossary.html)暴露哪些服务 ，你可以注册 Autofac 中的 组件。

组件可以通过反射（注册特定的.NET 类型或泛型开放类型）、提供现成的实例（你创建的对象的实例）或使用lambda 表达式（执行以实例化对象的匿名函数）来创建。ContainerBuilder 提供了一系列 Register() 方法，用于设置这些配置。

每个组件都会暴露一个或多个服务，这些服务通过 ContainerBuilder 上的 As() 方法进行绑定。

// 创建用于注册组件和服务的 builder 。  
var builder = new ContainerBuilder();  
  
// 注册暴露接口的类型...  
builder.RegisterType<ConsoleLogger>().As<ILogger>();  
  
// 注册你创建的对象实例...  
var output = new StringWriter();  
builder.RegisterInstance(output).As<TextWriter>();  
  
// 注册执行以创建对象的表达式...  
builder.Register(c => new ConfigReader("mysection")).As<IConfigReader>();  
  
// 构建容器以完成注册并准备对象解析。  
var container = builder.Build();  
  
// 现在你可以使用Autofac解析服务。例如，这行代码将执行注册到IConfigReader服务的lambda表达式。  
using(var scope = container.BeginLifetimeScope())  
{  
 var reader = scope.Resolve<IConfigReader>();  
}

## 反射组件

### 通过类型注册

反射生成的组件通常通过类型注册：

var builder = new ContainerBuilder();  
builder.RegisterType<ConsoleLogger>();  
builder.RegisterType(typeof(ConfigReader));

在使用基于反射的组件时，Autofac 会自动使用具有最多可从容器获取参数的类的构造函数。

例如，假设你有一个有三个构造函数的类：

public class MyComponent  
{  
 public MyComponent() { /\* ... \*/ }  
 public MyComponent(ILogger logger) { /\* ... \*/ }  
 public MyComponent(ILogger logger, IConfigReader reader) { /\* ... \*/ }  
}

现在，如果你像这样在容器中注册组件和服务：

var builder = new ContainerBuilder();  
builder.RegisterType<MyComponent>();  
builder.RegisterType<ConsoleLogger>().As<ILogger>();  
var container = builder.Build();  
  
using(var scope = container.BeginLifetimeScope())  
{  
 var component = scope.Resolve<MyComponent>();  
}

当你解析组件时，Autofac 会看到你已经注册了 ILogger ，但没有注册 IConfigReader 。在这种情况下，会选择第二个构造函数，因为它具有容器中能找到的最多参数。

关于基于反射的组件的一个重要注意事项： 通过 RegisterType 注册的所有组件类型必须是具体类型。虽然组件可以作为 服务 暴露抽象类或接口，但你不能注册抽象类或接口组件。如果你考虑一下，这是有意义的：在幕后，Autofac 正在创建你注册的东西的实例。你不能 new 一个抽象类或接口。你需要有一个实现，对吧？

### 指定构造函数

你可以手动选择特定的构造函数并覆盖自动选择，方法是在使用 UsingConstructor 方法和一个表示构造函数参数类型的类型列表时注册你的组件：

builder.RegisterType<MyComponent>()  
 .UsingConstructor(typeof(ILogger), typeof(IConfigReader));

需要注意的是，你仍然需要在解析时间提供所需的参数，否则尝试解析对象时会出错。你可以 在注册时传递参数 ，也可以 在解析时传递它们 。

注意 有关自定义选择构造函数的高级方法，请参阅 [这里](https://www.koudingke.cn/docs/zh-Hans/autofac-docs/latest/Advanced/Constructor-Selection) 。

### 必需属性

从 Autofac 7.0 开始，在基于反射的组件中，所有 [必需属性](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/required) 都会自动被解析，就像构造函数参数一样。

组件的所有必需属性都必须是可解析的服务（或作为 参数 提供），否则在尝试解析组件时会抛出异常。

例如，考虑一个具有以下属性的类：

public class MyComponent  
{  
 public required ILogger Logger { protected get; init; }  
  
 public required IConfigReader ConfigReader { protected get; init; }  
}

你可以像使用具有构造函数的类那样注册和使用这个类：

var builder = new ContainerBuilder();  
builder.RegisterType<MyComponent>();  
builder.RegisterType<ConsoleLogger>().As<ILogger>();  
builder.RegisterType<ConfigReader>().As<IConfigReader>();  
var container = builder.Build();  
  
using(var scope = container.BeginLifetimeScope())  
{  
 // Logger 和 ConfigReader将被填充。  
 var component = scope.Resolve<MyComponent>();  
}

必需属性也会自动设置在所有基类上（如果存在），这使得在深层次对象结构中使用必需属性很有用，因为这允许你避免在服务集中调用带有服务的基类构造函数；Autofac 会为你设置基类属性。

注意 有关必需属性注入的更多详细信息，请参阅 属性注入文档 中的专门部分。

## 实例组件

在某些情况下，你可能希望预先生成对象的实例并将它添加到容器中供注册的组件使用。你可以使用 RegisterInstance 方法来实现这一点：

var output = new StringWriter();  
builder.RegisterInstance(output).As<TextWriter>();

当你这样做时，需要考虑的是 Autofac 会 自动处理注册组件的生命周期管理 ，你可能更想控制对象的生命周期，而不是让 Autofac 为你调用 Dispose 。在这种情况下，你需要使用 ExternallyOwned 方法注册实例：

var output = new StringWriter();  
builder.RegisterInstance(output)  
 .As<TextWriter>()  
 .ExternallyOwned();

将提供的实例注册到容器中对于将 Autofac 集成到现有应用程序并使容器中的组件使用已存在的单例实例也很方便。与其直接将这些组件绑定到单例，不如将其注册为实例：

builder.RegisterInstance(MySingleton.Instance).ExternallyOwned();

这确保最终可以消除静态单例并替换为由容器管理的一个。

实例提供的默认服务是实例的实际类型。请参阅 “服务与组件” 部分。

## Lambda 表达式组件

当组件创建逻辑超出简单构造函数调用时，反射是一个很好的默认选择。然而，当事情变得复杂时，就会出现问题。

Autofac 可以接受一个委托或 lambda 表达式作为组件创建者：

builder.Register(c => new A(c.Resolve<B>()));

传递给表达式的参数 c 是组件上下文（一个 IComponentContext 对象），组件在此上下文中创建。你可以使用此上下文来从容器中解析其他值，以协助创建组件。使用这个上下文而不是闭包来访问容器非常重要，以便正确支持 确定性垃圾回收 和嵌套容器。

你还可以使用这个上下文参数来满足额外的依赖关系——在示例中，A 需要一个类型为 B 的构造函数参数，该参数可能有额外的依赖项。

除了使用 IComponentContext 在 lambda 表达式中解决依赖项外，你还可以使用泛型 Register 重载，将依赖项作为 lambda 的可变数量类型参数传递，并让 Autofac 为你解决它们：

builder.Register((IDependency1 dep1, IDependency2 dep2) => new Component(dep1, dep2));

如果你需要做出条件选择，或者使用 ResolveNamed 等方法，只需将 IComponentContext 作为 lambda 的第一个参数：

builder.Register((IComponentContext ctxt, IDependency1 dep1) => new Component(dep1, ctxt.ResolveNamed<IDependency2>("value")));

表达式创建的组件的默认服务是表达式返回类型。

下面是通过反射组件创建效果不佳但通过 lambda 表达式很好地解决的一些需求示例。

### 复杂参数

构造函数参数不能总是使用简单的常量值声明。与其在 XML 配置语法中费力地构建某种类型值，不如使用代码：

builder.Register(c => new UserSession(DateTime.Now.AddMinutes(25)));

（当然，会话过期可能是一个你想在配置文件中指定的事情——但你明白意思了；）

### 属性注入

虽然 Autofac 提供了 [更高级的属性注入方法](https://www.koudingke.cn/docs/zh-Hans/autofac-docs/latest/Register/Prop-Method-Injection) ，但你也可以使用表达式和属性初始化器来填充属性：

builder.Register(c => new A(){ MyB = c.ResolveOptional<B>() });

ResolveOptional 方法会尝试解析值，但如果服务未注册，不会抛出异常。（如果服务已注册但无法正确解析，你仍会收到异常。）这是 解析服务 的选项之一。

在大多数情况下，不推荐使用属性注入。 通过 [空对象模式](https://www.koudingke.cn/go?link=http%3a%2f%2fen.wikipedia.org%2fwiki%2fNull_Object_pattern)，重载构造函数或构造函数参数的默认值等替代方法，可以使用构造注入创建更干净、"不可变"的组件，同时支持可选依赖项。

### 根据参数值选择实现

隔离组件创建的一大好处是具体类型可以变化。这通常在运行时而非配置时间完成：

builder.Register<CreditCard>(  
 (c, p) =>  
 {  
 var accountId = p.Named<string>("accountId");  
 if (accountId.StartsWith("9"))  
 {  
 return new GoldCard(accountId);  
 }  
 else  
 {  
 return new StandardCard(accountId);  
 }  
 });

在这个例子中，CreditCard 由 GoldCard 和 StandardCard 两个类实现——具体实例化哪一种取决于运行时提供的账户 ID。

在这个示例中，参数是提供给创建函数的 通过一个名为 p 的可选第二个参数传递参数。

使用这种注册方式如下：

var card = container.Resolve<CreditCard>(new NamedParameter("accountId", "12345"));

你可以使用 Func<X, Y> 关系 ，将参数从命名参数改为类型参数：

builder.Register<CreditCard>(  
 (c, p) =>  
 {  
 // 使用类型参数而不是命名参数以与Func<X, Y>配合  
 var accountId = p.TypedAs<string>();  
 if (accountId.StartsWith("9"))  
 {  
 return new GoldCard(accountId);  
 }  
 else  
 {  
 return new StandardCard(accountId);  
 }  
 });

这样注册看起来像这样：

var cardFactory = container.Resolve<Func<string, CreditCard>>();  
var card = cardFactory("12345");

你也可以使用 [lambda 表达式组件](#Top_of_s3_xhtml) ，使其更简单：

builder.Register<string, CreditCard>(  
 accountId =>  
 {  
 if (accountId.StartsWith("9"))  
 {  
 return new GoldCard(accountId);  
 }  
 else  
 {  
 return new StandardCard(accountId);  
 }  
 });

如果你有一个创建 CreditCard 实例的委托，并使用了 delegate 工厂 ，也可以实现另一种干净且类型安全的语法。Delegate 工厂是支持多个相同类型参数的方法。

## 开放泛型组件

Open generic

Autofac 支持开放泛型类型。使用 RegisterGeneric() builder 方法：

builder.RegisterGeneric(typeof(NHibernateRepository<>))  
 .As(typeof(IRepository<>))  
 .InstancePerLifetimeScope();

当容器请求匹配的服务类型时，Autofac 会映射为等效的**封闭**实现类型：

// Autofac 将返回一个 NHibernateRepository <Task>  
var tasks = container.Resolve<IRepository<Task>>();

注册特定服务类型（如 IRepository）会覆盖开放泛型版本。

如果需要根据自定义行为选择封闭泛型实现，也可以使用委托提供封闭泛型类型：

var builder = new ContainerBuilder();  
  
builder.RegisterGeneric((context, types, parameters) =>  
{  
 // 对选择哪种封闭类型作出决策。  
 if (types.Contains(typeof(string)))  
 {  
 return new StringSpecializedImplementation();  
 }  
  
 return Activator.CreateInstance(typeof(GeneralImplementation<>).MakeGenericType(types));  
}).As(typeof(IService<>));

注意 请注意，使用 RegisterGeneric 的委托形式通常比基于反射的形式性能稍低，因为不能以相同方式缓存封闭泛型类型。

## 服务与组件

当你注册 组件 时，必须告诉 Autofac 该组件暴露哪些 服务 。默认情况下，大多数注册项只自己暴露为注册类型：

// 这个暴露了服务 "CallLogger"  
builder.RegisterType<CallLogger>();

组件只能根据它们公开的服务进行 决定 。在简单示例中这意味着：

// 这将工作，因为组件默认暴露其类型：  
scope.Resolve<CallLogger>();  
  
// 这将不会工作，因为我们没有告诉注册也把 ILogger 接口暴露在 CallLogger 上：  
scope.Resolve<ILogger>();

你可以根据需要为组件暴露任意数量的服务：

builder.RegisterType<CallLogger>()  
 .As<ILogger>()  
 .As<ICallInterceptor>();

一旦暴露了服务，就可以根据该服务解析组件。但是，请注意，一旦将组件作为特定服务公开，将默认服务（组件类型）替换掉：

// 这些都将工作，因为我们注册时提供了适当的服务：  
scope.Resolve<ILogger>();  
scope.Resolve<ICallInterceptor>();  
  
// 这将不再工作，因为我们为组件指定了服务覆盖：  
scope.Resolve<CallLogger>();

如果你想根据一组服务以及使用默认服务公开组件，请使用 AsSelf 方法：

builder.RegisterType<CallLogger>()  
 .AsSelf()  
 .As<ILogger>()  
 .As<ICallInterceptor>();

现在所有这些都将工作：

// 这些都将工作，因为我们注册时提供了适当的服务：  
scope.Resolve<ILogger>();  
scope.Resolve<ICallInterceptor>();  
scope.Resolve<CallLogger>();

## 默认注册

如果有多个组件暴露相同的 服务 ，Autofac 将使用最后一个注册的组件作为该服务的默认提供者：

builder.RegisterType<ConsoleLogger>().As<ILogger>();  
builder.RegisterType<FileLogger>().As<ILogger>();

在这种情况下，FileLogger 将是 ILogger 的默认，因为它是最后注册的。

要更改此行为，请使用 PreserveExistingDefaults() 修饰符：

builder.RegisterType<ConsoleLogger>().As<ILogger>();  
builder.RegisterType<FileLogger>().As<ILogger>().PreserveExistingDefaults();

在这种情况下，ConsoleLogger 将是 ILogger 的默认，因为对 FileLogger 的后续注册使用了 PreserveExistingDefaults() 。

## 条件注册

Conditional registration

**注意** 条件注册是在 Autofac 4.4.0版本中引入的。

大多数情况下，如上所述 "默认注册" 部分所述，重写注册就足以确保在运行时正确解析组件。确保按正确的顺序注册；利用 PreserveExistingDefaults() ；利用 lambda 表达式/委托注册处理更复杂的条件和行为，可以让你走得更远。

然而，在某些场景下，这可能不是你想要的方式：

* 如果有其他功能正在处理，你不想让组件存在于系统中。例如，如果你解析一个 IEnumerable 的服务，所有实现该服务的注册组件都将返回，即使未使用 PreserveExistingDefaults() 。通常这没问题，但在某些边缘情况下可能不希望这样。
* 只有在其他组件未注册时，或者只有在其他组件已注册时，你才想注册组件。你不能从正在构建的容器中解析东西，也不应该更新已经构建好的容器。根据其他注册动态注册组件可能会有所帮助。

有两个注册扩展可以帮助解决这些问题：

* OnlyIf() - 提供一个使用IComponentRegistryBuilder来确定是否应进行注册的 lambda。
* IfNotRegistered() - 简化方法，如果其他服务已注册，则阻止注册发生。

这些扩展在 ContainerBuilder.Build() 时运行，并按实际组件注册的顺序执行。以下是一些示例，展示它们如何工作：

var builder = new ContainerBuilder();  
  
// 只有 ServiceA 将被注册。  
// 注意 IfNotRegistered 检查的是 SERVICE TYPE（As<T>），而不是 COMPONENT TYPE（RegisterType<T>）。  
builder.RegisterType<ServiceA>()  
 .As<IService>();  
builder.RegisterType<ServiceB>()  
 .As<IService>()  
 .IfNotRegistered(typeof(IService));  
  
// HandlerA 将被注册 - 它在 HandlerB 有机会注册之前运行，  
// 所以 IfNotRegistered 检查将找不到它。  
//  
// HandlerC 不会被注册，因为它在 HandlerB 之后运行。注意它可以检查 HandlerB 的类型，  
// 因为 HandlerB 使用了 AsSelf() 而不是仅 As<IHandler>()。再次强调，  
// IfNotRegistered 只能检查 "As" 类型的注册。  
  
builder.RegisterType<HandlerA>()  
 .AsSelf()  
 .As<IHandler>()  
 .IfNotRegistered(typeof(HandlerB));  
builder.RegisterType<HandlerB>()  
 .AsSelf()  
 .As<IHandler>();  
builder.RegisterType<HandlerC>()  
 .AsSelf()  
 .As<IHandler>()  
 .IfNotRegistered(typeof(HandlerB));  
  
// Manager将被注册，因为都注册了IService和HandlerB。OnlyIf谓词允许更大的灵活性。  
builder.RegisterType<Manager>()  
 .As<IManager>()  
 .OnlyIf(reg =>  
 reg.IsRegistered(new TypedService(typeof(IService))) &&  
 reg.IsRegistered(new TypedService(typeof(HandlerB))));  
  
// 对于开放泛型要小心 - IfNotRegistered和IsRegistered仅适用于封闭泛型，因为那是你将要解析的！  
// 注意检查的是封闭泛型，它作为开放泛型的一部分将被注册。  
// 如果你在IfNotRegistered检查中放入一个开放泛型，它总是会显示未注册  
builder.RegisterGeneric(typeof(CommandHandler<>))  
 .As(typeof(ICommandHandler<>))  
 .IfNotRegistered(typeof(ICommandHandler<MyCommand>));  
  
// 这是条件检查实际运行的地方。再次强调，它们按照添加到ContainerBuilder的注册顺序运行。  
var container = builder.Build();

## 注册配置

你可以通过 XML 或程序化配置（ “模块” ）配置同时提供一组注册，或者在运行时更新注册。你还可以 使用 Autofac 模块 来实现动态注册逻辑或条件注册策略。

## 动态注册

Autofac 模块 是引入动态注册逻辑或简单跨切特性最简便的方式。例如，你可以使用模块来 动态地将 log4net 日志实例附加到正在解决的服务上 。

如果你发现需要更动态的行为，比如添加对新 隐式关系类型的支持 ，你可能需要 查看高级概念部分的注册源部分 。

# Autofac 将参数注册到组件

当你使用 注册组件 时，你可以为基于该组件的 服务解决方案 提供一组参数。如果你更愿意在解决服务时提供参数，也可以 这样做。

## 可用的参数类型

Autofac 提供了几种不同的参数匹配策略：

* NamedParameter：按名称匹配目标参数
* TypedParameter：按类型（精确类型匹配）匹配目标参数
* ResolvedParameter：灵活参数匹配

NamedParameter 和 TypedParameter 只能提供常量值。

ResolvedParameter 可以用来动态地从容器中获取值，例如通过按名称解析服务。

## 使用反射组件的参数

当你注册一个基于反射的组件时，组件类型的构造函数可能需要一个无法从容器中解析的参数。你可以在注册时使用参数提供这个值。

假设你有一个需要传递配置节名称的配置读取器：

public class ConfigReader : IConfigReader  
{  
 public ConfigReader(string configSectionName)  
 {  
 // 存储配置节名称  
 }  
  
 // ...根据节名称读取配置。  
}

你可以使用 lambda 表达式组件来实现：

builder.Register(c => new ConfigReader("sectionName")).As<IConfigReader>();

或者为反射组件注册提供参数：

// 使用 NAMED 参数：  
builder.RegisterType<ConfigReader>()  
 .As<IConfigReader>()  
 .WithParameter("configSectionName", "sectionName");  
  
// 使用 TYPED 参数：  
builder.RegisterType<ConfigReader>()  
 .As<IConfigReader>()  
 .WithParameter(TypedParameter.From("sectionName"));  
  
// 使用 RESOLVED 参数：  
builder.RegisterType<ConfigReader>()  
 .As<IConfigReader>()  
 .WithParameter(  
 new ResolvedParameter(  
 (pi, ctx) => pi.ParameterType == typeof(string) && pi.Name == "configSectionName",  
 (pi, ctx) => "sectionName"));

## 使用 lambda 表达式组件的参数

对于 lambda 表达式组件注册，你不是在注册时传递参数值，而是启用在服务解决时传递参数值的能力。（ 更多关于带有参数的解决服务的信息 ）

在组件注册表达式中，你可以利用传入的参数，使用泛型 lambda Register 方法：

builder.Register((MyConfig config) => new Worker(config));

如果你需要访问完整的参数列表，可以通过更改注册委托的签名来实现。不要将参数作为 lambda 的参数，而是接受一个 IComponentContext 和一个 IEnumerable：

// 使用两个参数注册委托：  
// c = 当前的 IComponentContext，用于动态解析依赖项  
// p = 包含传入参数集的 IEnumerable<Parameter>  
builder.Register((c, p) =>  
 new Worker(p.Named<MyConfig>("config")));

当 [使用参数解析](https://www.koudingke.cn/docs/zh-Hans/autofac-docs/latest/Resolve/Parameters) 时，你的 lambda 会使用传递的参数：

var customConfig = new MyConfig  
{  
 SomeValue = "../"  
};  
  
var worker = scope.Resolve<Worker>(new NamedParameter("config", customConfig));

# Autofac 属性和方法注入

虽然构造函数参数注入是向正在构建的组件传递值的首选方法，但也可以使用属性或方法注入来提供值。

属性注入 使用可写属性而不是构造函数参数进行注入。方法注入 通过调用方法设置依赖项。

### 属性注入

如果组件是一个 lambda 表达式组件 ，则使用对象初始化器：

builder.Register(c => new A { B = c.Resolve<B>() });

为了支持 [循环依赖](https://www.koudingke.cn/docs/zh-Hans/autofac-docs/latest/Advanced/Circular-Dependencies)，使用一个 [激活事件处理器](https://www.koudingke.cn/docs/zh-Hans/autofac-docs/latest/Lifetime/Events) ：

builder.Register(c => new A()).OnActivated(e => e.Instance.B = e.Context.Resolve<B>());

#### 必需属性

从 Autofac 7.0 开始，对于 反射组件 ，所有 [必需属性](https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.codeanalysis.setsrequiredmembersattribute) 在对象构造时会自动解析，并通常与强制性的构造函数参数以相同方式处理。

例如，考虑以下类型：

public class MyComponent  
{  
 public required ILogger Logger { protected get; init; }  
  
 public required IConfigReader ConfigReader { protected get; init; }  
}

当组件被解析时，Autofac 将填充 Logger 和 ConfigReader 属性，就像它们是构造函数参数一样。Context 属性将被视为标准属性，不会默认被填充。

你可以使用任何有效的访问修饰符组合在必需属性上，但是 public required ... { protected get; init; } 在这些示例中被使用，因为它提供了类似于构造函数的访问和可见性：属性只在构造时可设置，对其他类不可见。

必需属性注入也自动适用于所有具有必需属性的基类：

public class ComponentBase  
{  
 public required ILogger Logger { protected get; init; }  
}  
  
public class MyComponent : ComponentBase  
{  
 public required IConfigReader ConfigReader { protected get; init; }  
}

在上述示例中，解析 MyComponent 时，将同时在基类和组件本身中填充 Logger 。

**重要** Autofac 并不认为必需属性的类型是否为 nullable 表示某种 “可选” 必需属性。如果属性标记为 required ，那么它是必需的，并且必须注入，或者通过参数提供，无论其是否为 null 。

#### 必需属性与构造函数

如果你想混合使用构造函数和必需属性，可以这样做：

public class MyComponent  
{  
 public MyComponent(ILogger logger)  
 {  
 Logger = logger;  
 }  
  
 private ILogger Logger { get; set; }  
  
 public required IConfigReader ConfigReader { protected get; init; }  
}

如果有多个构造函数可用，默认情况下 Autofac 会选择参数匹配最多的构造函数（除非使用了 自定义构造函数选择 。这种情况保持不变，必需属性集对选择的构造函数没有影响。

Autofac 并不知道你是否在构造函数内部设置了某个必需属性。看这个例子：

public class MyComponent  
{  
 public MyComponent()  
 {  
 }  
  
 public MyComponent(ILogger logger)  
 {  
 Logger = logger;  
 }  
  
 public required ILogger Logger { protected get; init; }  
}

在这种情况下，Autofac 会选择接受 ILogger 参数的构造函数，该构造函数又设置了 Logger 属性。然而，由于 Logger 标记为必需属性，Autofac 将再次解析 ILogger 并将其注入到必需属性中。

要避免这种情况，请使用 [SetsRequiredMembers](https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.codeanalysis.setsrequiredmembersattribute) 属性标记设置所有必需属性的构造函数：

using System.Diagnostics.CodeAnalysis;  
  
public class MyComponent  
{  
 public MyComponent()  
 {  
 }  
  
 [SetsRequiredMembers]  
 public MyComponent(ILogger logger)  
 {  
 Logger = logger;  
 }  
  
 public required ILogger Logger { protected get; init; }  
}

由于构造函数标记为设置了所有必需成员，当使用该构造函数创建组件实例时，Autofac 将不会进行任何必需属性注入。

#### 必需属性与参数

在 注册参数 或 解析参数 时提供的任何 TypedParameter 都将用于注入必需属性。但是，NamedParameter 和 PositionalParameter 不被视为属性注入的有效参数，因为它们被认为是只应用于构造函数参数。

### PropertiesAutowired

你可以在注册时使用 PropertiesAutowired() 修饰符在任何组件上注入属性：

// 默认行为：注入所有公共可写的属性。  
builder.RegisterType<A>().PropertiesAutowired();  
  
// 提供更精细的属性选择器委托。此示例显示注入所有属性，其中属性类型以'I'开头——一种“仅注入接口属性”的方式。委托接收要注入的属性的PropertyInfo描述和注入的对象。  
builder.RegisterType<B>()  
 .PropertiesAutowired(  
 (propInfo, instance) => propInfo.PropertyType.Name.StartsWith("I"));  
  
// 更加复杂，你可以提供自己的IPropertySelector实现，包含所需的任何功能。别忘了这将在每个关联的解决期间运行，因此性能很重要！  
builder.RegisterType<C>().PropertiesAutowired(new MyCustomPropSelector());

### 手动指定属性

如果你只想将一个特定的属性和值连接起来，可以使用 WithProperty() 修饰符：

builder.RegisterType<A>().WithProperty("PropertyName", propertyValue);

### 覆盖必需属性

使用 WithProperty 方法在注册类型时为必需属性提供的任何属性值将覆盖注入该属性的需求，Autofac 将使用提供的值代替：

public class MyComponent  
{  
 public required ILogger Logger { protected get; init; }  
  
 public required IConfigReader ConfigReader { protected get; init; }  
}  
  
var builder = new ContainerBuilder();  
builder.RegisterType<MyComponent>().WithProperty("Logger", new ConsoleLogger());  
  
var container = builder.Build();  
  
// 尽管没有注册ILogger，但这不会抛出异常。Logger属性由WithProperty提供。  
container.Resolve<MyComponent>();

### 在现有对象上注入属性

你还可以仅填充对象上的属性。为此，请使用生命周期范围上的 InjectUnsetProperties 扩展，它将解析并填充所有**公共、可写且尚未设置（null）**的属性：

lifetimeScope.InjectUnsetProperties(myObject);

### 方法注入

最简单的方法是在组件上调用一个方法来设置值，只需使用一个 lambda 表达式组件 并在激活器中处理方法调用：

builder.Register(c => {  
 var result = new MyObjectType();  
 var dep = c.Resolve<TheDependency>();  
 result.SetTheDependency(dep);  
 return result;  
});

如果不能使用注册 lambda，可以在 激活事件处理器 中添加一个：

builder  
 .RegisterType<MyObjectType>()  
 .OnActivating(e => {  
 var dep = e.Context.Resolve<TheDependency>();  
 e.Instance.SetTheDependency(dep);  
 });

# Autofac - Assembly扫描

Autofac 可以使用约定来查找和注册组件。你可以扫描并注册单个类型，也可以专门针对 Autofac 模块 进行扫描。

## 类型扫描

也称为约定驱动注册或扫描，Autofac 可根据用户指定的规则从一个程序集中注册一组类型：

var dataAccess = Assembly.GetExecutingAssembly();  
  
builder.RegisterAssemblyTypes(dataAccess)  
 .Where(t => t.Name.EndsWith("Repository"))  
 .AsImplementedInterfaces();

每次调用 RegisterAssemblyTypes() 都只会应用一套规则 - 如果有多个不同的组件集合需要注册，则需要多次调用 RegisterAssemblyTypes()。

Autofac 还支持使用 RegisterAssemblyOpenGenericTypes() 进行泛型开放类型的程序集扫描。它使用与 RegisterAssemblyTypes() 类似的通用语义：

var dataAccess = Assembly.GetExecutingAssembly();  
 builder.RegisterAssemblyOpenGenericTypes(dataAccess)  
 .Where(t => t.Name.StartsWith("MessageHandler"))  
 .AsImplementedInterfaces();

### 过滤类型

RegisterAssemblyTypes() 和 RegisterAssemblyOpenGenericTypes() 分别接受一个或多个程序集的参数数组。默认情况下，将注册程序集中所有具体的类 ，包括内部和私嵌套类。你可以使用提供的 LINQ 样式的谓词来过滤要注册的类型集。

在 4.8.0 中添加了 PublicOnly() 扩展方法，使数据封装更加方便。如果你只想注册公开类，请使用 PublicOnly()：

builder.RegisterAssemblyTypes(asm)  
 .PublicOnly();

要应用自定义过滤器到注册的类型，使用 Where() 谓词：

builder.RegisterAssemblyTypes(asm)  
 .Where(t => t.Name.EndsWith("Repository"));

要排除类型进行扫描，请使用 Except() 谓词：

builder.RegisterAssemblyTypes(asm)  
 .Except<MyUnwantedType>();

Except() 谓词还允许你为特定排除类型自定义注册：

builder.RegisterAssemblyTypes(asm)  
 .Except<MyCustomizedType>(ct =>  
 ct.As<ISpecial>().SingleInstance());

可以使用多个过滤器，此时它们将按逻辑与应用。

builder.RegisterAssemblyTypes(asm)  
 .PublicOnly()  
 .Where(t => t.Name.EndsWith("Repository"))  
 .Except<MyUnwantedRepository>();

### 指定服务

RegisterAssemblyTypes() 和 RegisterAssemblyOpenGenericTypes() 的注册语法是单个类型注册语法的子集，因此像 As() 这样的方法对程序集也适用：

builder.RegisterAssemblyTypes(asm)  
 .Where(t => t.Name.EndsWith("Repository"))  
 .As<IRepository>();

As() 和 Named() 方法的额外重载接受决定类型将提供哪些服务的 lambda 表达式：

uilder.RegisterAssemblyTypes(asm)  
 .As(t => t.GetInterfaces()[0]);

与常规组件注册一样，多次调用 As() 将组合在一起。

为了更容易建立常见的约定，已添加了一些额外的注册方法：

| 方法 | 描述 | 示例 |
| --- | --- | --- |
| AsImplementedInterfaces() | 将类型注册为提供其公开接口（排除 IDisposable）作为服务。 | builder.RegisterAssemblyTypes(asm).Where(t => t.Name.EndsWith("Repository")).AsImplementedInterfaces(); |
| AsClosedTypesOf(open) | 注册可以赋值给封闭的泛型类型的类型。 | builder.RegisterAssemblyTypes(asm).AsClosedTypesOf(typeof(IRepository<>)); |
| AsSelf() | 默认：将类型注册为自身 - 当与其他服务规范一起覆盖默认行为时很有用。 | builder.RegisterAssemblyTypes(asm).AsImplementedInterfaces().AsSelf(); |

## 模块扫描

模块扫描由 RegisterAssemblyModules() 注册方法执行，顾名思义，它会扫描提供的程序集，创建模块实例，然后将它们注册到当前容器 builder 。

例如，如果下面两个简单的模块类位于同一个程序集中，并且每个都注册一个组件：

public class AModule : Module  
 {  
 protected override void Load(ContainerBuilder builder)  
 {  
 builder.Register(c => new AComponent()).As<AComponent>();  
 }  
 }  
 public class BModule : Module  
 {  
 protected override void Load(ContainerBuilder builder)  
 {  
 builder.Register(c => new BComponent()).As<BComponent>();  
 }  
 }

不接受类型参数的 RegisterAssemblyModules() 重载将注册程序集中实现 IModule 的所有类。在下面的例子中，两个模块都会被注册：

var assembly = typeof(AComponent).Assembly;  
 var builder = new ContainerBuilder();  
 builder.RegisterAssemblyModules(assembly);

带泛型类型参数的 RegisterAssemblyModules() 重载允许你指定模块必须继承的基类型。在下面的例子中，只注册了一个模块，因为扫描受到了限制：

var assembly = typeof(AComponent).Assembly;  
 var builder = new ContainerBuilder();  
 builder.RegisterAssemblyModules<AModule>(assembly);

带类型对象参数的 RegisterAssemblyModules() 重载工作起来像泛型类型参数重载，但它允许你指定可能在运行时确定的类型。在下面的例子中，只注册了一个模块，因为扫描受到了限制：

var assembly = typeof(AComponent).Assembly;  
 var builder = new ContainerBuilder();  
 builder.RegisterAssemblyModules(typeof(AModule), assembly);

## IIS 托管的 Web 应用程序

在使用 IIS 应用程序进行程序集扫描时，根据你的做法可能会遇到一些问题（ 这是我们的常见问题之一 ）。

在 IIS 中托管应用程序时，应用程序首次启动时所有程序集都会加载到 AppDomain 中，但当 IIS 重新启动 AppDomain 时，这些程序集将仅按需加载。

要解决此问题，请使用 [GetReferencedAssemblies()](https://www.koudingke.cn/go?link=https%3a%2f%2fmsdn.microsoft.com%2fen-us%2flibrary%2fsystem.web.compilation.buildmanager.getreferencedassemblies.aspx) 方法从 [System.Web.Compilation.BuildManager](https://www.koudingke.cn/go?link=https%3a%2f%2fmsdn.microsoft.com%2fen-us%2flibrary%2fsystem.web.compilation.buildmanager.aspx) 获取引用的程序集列表：

var assemblies = BuildManager.GetReferencedAssemblies().Cast<Assembly>();

这将强制引用的程序集立即加载到 AppDomain 中，使其可用于模块扫描。

https://findermp.video.qq.com/251/20304/stodownload?encfilekey=Cvvj5Ix3eez3Y79SxtvVL0L7CkPM6dFibFeI6caGYwFHic0uERW3KtYVPQc30eTiaxBcnW4iapc7Stno0MPicGg8cR2JNwbibFSRlqOkbZpWuWUTNUXBWpLm0338yUHiaUt7yw6tIKmUN83adxBBCp0kyXAnQ&dotrans=0&hy=SH&idx=1&m=f27e3814dadbc41c5665979b3bd92f42&uzid=2&X-snsvideoflag=xWT111&token=6xykWLEnztJZATruM6AKVDSPKbctMk7rT2JSJowta8nLxUEDPUH8Micr9picYdleyhpJWnn5FlasjEuEE1xo2E8JeZOBKCQglSunibgwvHnbqpH8IDlCyBo8q4cNDRIVooZ# Autofac 服务解析

Resolving Services

在你已经注册了相应的组件，并且公开了它们提供的服务之后 组件注册，你可以从构建好的容器或者子生命周期范围(lifetime scope)中获取服务。使用Resolve() 方法来实现：

var builder = new ContainerBuilder();  
builder.RegisterType<MyComponent>().As<IService>();  
var container = builder.Build();  
using(var scope = container.BeginLifetimeScope())  
{  
 var service = scope.Resolve<IService>();  
}

注意 例子中是通过生命周期范围而不是直接从容器中获取服务——你也应该这样做。

*虽然可以从根容器中直接获取组件，但在某些情况下这样做可能会导致内存泄漏。****建议你尽可能地从生命周期范围中获取组件，确保服务实例能够被正确地释放和垃圾回收。有关更多内容，请阅读关于*** *控制范围和生命周期\_\_的章节。*

在解析服务时，Autofac会自动处理服务依赖关系树的整个结构，并解决构建服务所需的任何依赖。如果你有 循环依赖而没有得到妥善处理，或者缺少必需的服务，你会得到一个DependencyResolutionException异常。

如果你需要获取一个可能注册也可能未注册的服务，你可以尝试使用 ResolveOptional()或 TryResolve()来进行条件解析：

var service = scope.ResolveOptional<IService>();  
IProvider provider = null;  
if(scope.TryResolve<IProvider>(out provider))  
{  
   
}

ResolveOptional() 和TryResolve() 都围绕着特定服务是否注册的条件性。如果服务已注册，将尝试解析。如果解析失败（例如，由于缺少注册的依赖），你仍然会得到一个 DependencyResolutionException。如果你需要基于服务是否成功解析的条件性解析，可以将Resolve() 调用包裹在一个 try/catch 块中。

你可能还会对查看 [高级主题列表](https://docs.autofac.org/en/latest/advanced/index.html)感兴趣，了解 [命名和键控服务 、与组件元数据一起工作](https://docs.autofac.org/en/latest/advanced/metadata.html)等与服务解析相关的其他话题。

# Autofac参数解析

Passing Parameters to Resolve

### Autofac参数解析

当需要 解析服务 时，你可能会发现需要将参数传递给解析。如果你知道注册时的值，可以使用 在注册时提供这些参数 。

Resolve()方法接受与注册时相同的参数类型，使用可变长度参数列表。另外，委托工厂 和Func暗示关系类型也允许在解析期间传递参数。

## 可用的参数类型

Available Parameter Types

Autofac提供了几种不同的参数匹配策略：

* NamedParameter：通过名称匹配目标参数
* TypedParameter：按类型匹配目标参数（要求精确类型匹配）
* ResolvedParameter：灵活参数匹配

NamedParameter 和TypedParameter 只能提供常量值。

ResolvedParameter 可以用于动态从容器中获取值的方式提供值，例如根据名称解析服务。

## 使用反射组件时的参数

Parameters with Reflection Components

当你解析基于反射的组件时，类型的构造函数可能需要一个依赖于运行时值的参数，这在注册时是不可用的。可以在Resolve()调用中使用参数来提供该值。

假设你有一个配置读取器，它需要传入配置节的名称：

public class ConfigReader : IConfigReader  
{  
 public ConfigReader(string configSectionName)  
 {  
   
 }  
}

你可以这样向 Resolve()调用传递参数：

var reader = scope.Resolve<ConfigReader>(  
 new NamedParameter("configSectionName", "sectionName"));

与注册时的参数类似 ，如果ConfigReader 组件是 通过反射 ，那么NamedParameter 将映射到相应的命名构造函数参数。

如果有多个参数，只需通过Resolve() 方法一起传递它们：

var service = scope.Resolve<AnotherService>(  
 new NamedParameter("id", "service-identifier"),  
 new TypedParameter(typeof(Guid), Guid.NewGuid()),  
 new ResolvedParameter(  
 (pi, ctx) => pi.ParameterType == typeof(ILog) && pi.Name == "logger",  
 (pi, ctx) => LogManager.GetLogger("service")));

## 使用 lambda表达式组件时的参数

对于 lambda表达式组件注册，你需要在 lambda表达式内部处理参数，以便在Resolve()调用中传递它们时，可以利用这些参数。

在组件注册表达式中，可以通过更改注册委托的签名来利用传入的参数。除了接收一个 IComponentContext参数外，还需要接收一个 IComponentContext和一个 IEnumerable：

builder.Register((c, p) =>  
 new ConfigReader(p.Named<string>("configSectionName")))  
 .As<IConfigReader>();

现在当你解析IConfigReader 时，你的 lambda 会使用传递的参数：

var reader = scope.Resolve<IConfigReader>(new NamedParameter(  
 "configSectionName", "sectionName"));

## 不显式调用 Resolve 传递参数

Autofac 支持两种功能，可以自动生成服务 “工厂” ，这些工厂接受强类型参数列表，在解析时使用这些参数。这是一种稍微更干净的方式来创建需要参数的组件实例。

* 委托工厂 允许定义工厂委托方法。
* Func暗示 关系类型 可以提供自动生成的工厂函数。

关于使用这些方法根据参数选择实现的示例，请参阅 注册页面上的示例。

# Autofac隐式关系类型

Implicit Relationship Types

Autofac支持自动解决特定类型，以支持组件和服务之间的特殊关系。要利用这些关系，请正常注册组件，但更改消耗组件的构造函数参数或在Resolve()调用中指定的关系类型。

例如，当 Autofac 注入一个IEnumerable 类型的构造函数参数时，它不会寻找提供IEnumerable 的组件。相反，容器将找到所有实现ITask 的组件，并注入它们。

（不用担心，下面有例子展示了各种类型的使用方法及其含义。）

*注意：为了****覆盖默认行为****，仍然可以显式注册这些类型的实现。*

[本文档的内容基于 Nick Blumhardt 的博客文章 [The Relationship Zoo](http://nblumhardt.com/2010/01/the-relationship-zoo/)]

[关系动物园](https://nblumhardt.com/2010/01/the-relationship-zoo/)

## 支持的关系类型

下表总结了 Autofac中支持的各种关系类型，并显示了你可以使用的 .NET类型来消费它们。每个关系类型后面都有更详细的描述和用例。

| 关系 | 类型 | 意义 |
| --- | --- | --- |
| A 需要 B | B | 直接依赖 |
| A 在未来某个时间需要 B | Lazy<B> | 延迟实例化 |
| A 在未来某个时间需要 B | Owned<B> | 受控生命周期 |
| A 需要创建 B 的实例 | Func<B> | 动态实例化 |
| A 需要为 B 提供参数 X 和 Y | Func<X, Y, B> | 参数化实例化 |
| A 需要所有种类的 B | IEnumerable<B>、IList<B>、ICollection<B> | 列表 |
| A 需要知道关于 B 的 X | Meta<B> 和Meta<B, X> | 元数据查询 |
| A 根据 X 选择 B | IIndex<X, B> | 键控服务查找 |

### 直接依赖 (B)

Direct Dependency (B)

直接依赖是最基础的关系，组件 A需要服务 B。这通过标准构造函数和属性注入自动处理：

public class A  
{  
 public A(B dependency) { ... }  
}

注册A 和B 组件，然后解耦：

var builder = new ContainerBuilder();  
builder.RegisterType<A>();  
builder.RegisterType<B>();  
var container = builder.Build();  
using (var scope = container.BeginLifetimeScope())  
{   
 var a = scope.Resolve<A>();  
}

### 延迟实例化 (Lazy<B>)

Delayed Instantiation (Lazy<B>)

延迟依赖不会在首次使用时实例化。这种关系出现在依赖很少使用或者创建成本较高的情况下。要利用这一点，请在A的构造函数中使用 Lazy<B>：

public class A  
{  
 Lazy<B> \_b;  
 public A(Lazy<B> b) { \_b = b }  
 public void M()  
 {  
   
 \_b.Value.DoSomething();  
 }  
}

如果你有一个需要元数据的延迟依赖，可以使用 Lazy<B, M> 而不是较长的 Meta<Lazy<B>, M>。

### 受控生命周期 (Owned<B>)

Controlled Lifetime (Owned<B>)

受控依赖可以在不再需要时由拥有者释放。受控依赖通常对应于依赖组件执行的一些工作单元。

这种关系类型尤其有趣，当与实现IDisposable的组件一起工作时。Autofac在生命周期范围结束时自动丢弃可丢弃的组件 ，但这可能意味着组件被持有过久；或者你可能只想自己控制对象的释放。在这种情况下，你可以使用一个受控依赖。

public class A  
{  
 Owned<B> \_b;  
 public A(Owned<B> b) { \_b = b; }  
 public void M()  
 {  
   
 \_b.Value.DoSomething();  
   
 \_b.Dispose();  
 }  
}

内部，Autofac在一个小型生命周期范围内为B服务进行解析。当你调用Dispose()时，生命周期范围会被丢弃。这意味着如果依赖项不是共享的（例如单例），那么丢弃B也将同时丢弃它的依赖项。

这也意味着，如果你在一个生命周期范围内注册了InstancePerLifetimeScope()，并且将其作为 Owned<B> 解决，那么你可能不会在同一个生命周期范围内获得与其他地方相同的实例。这个示例显示了问题：

var builder = new ContainerBuilder();  
 builder.RegisterType<A>().InstancePerLifetimeScope();  
 builder.RegisterType<B>().InstancePerLifetimeScope();  
 var container = builder.Build();  
 using (var scope = container.BeginLifetimeScope())  
 {  
   
 var b1 = scope.Resolve<B>();  
 b1.DoSomething();  
   
 var b2 = scope.Resolve<B>();  
 b2.DoSomething();  
   
 var a = scope.Resolve<A>();  
 a.M();  
 }

这是设计上的原因，因为你不希望一个组件在其他所有组件之下丢弃B。然而，如果没有意识到，这可能会导致一些混淆。

如果你始终想自行控制 B的释放，可以将 B 注册为 ExternallyOwned()。

### 动态实例化 (Func<B>)

Dynamic Instantiation (Func<B>)

使用自动生成的工厂可以让你在程序控制流内动态地在程序中resolve B，而无需直接依赖 Autofac 库。如果满足以下条件，请使用此关系类型：

* 需要创建特定服务的多个实例。
* 想要特别控制服务设置的时间。
* 对是否需要服务不确定，希望在运行时决定。

这种关系在 WCF 集成 等场景中也很有用，其中需要在通道故障后创建新的服务代理。

Func<B>行为就像调用Resolve<B>()。这意味着它不仅限于处理无参构造函数的对象，它会连接构造参数、执行属性注入，并遵循 Resolve<B>() 所做的整个生命周期。

此外，生命周期范围受到尊重。如果你将对象注册为InstancePerDependency()，并多次调用Func<B>，每次都会得到一个新的实例；如果你将对象注册为SingleInstance()，并且多次调用Func<B>来解决对象，无论你传递多少次，都将始终获得同一个对象实例。

这种关系的一个示例如下：

public class B  
{  
 public B() {}  
 public void DoSomething() {}  
}  
  
public class A  
{  
 Func<B> \_bFactory;  
 public A(Func<B> b) { \_bFactory = b; }  
 public void M()  
 {  
   
 var b = \_bFactory();  
 b.DoSomething();  
 }  
}

注册A和B组件 ，然后解耦：

var builder = new ContainerBuilder();  
 builder.RegisterType<A>();  
 builder.RegisterType<B>();  
 var container = builder.Build();  
 using (var scope = container.BeginLifetimeScope())  
 {  
   
 var a = scope.Resolve<A>();  
   
 a.M();  
   
 a.M();  
 a.M();  
 }

生命周期范围受到尊重，因此你可以利用这一点。

var builder = new ContainerBuilder();  
builder.RegisterType<A>();  
builder.RegisterType<B>().InstancePerLifetimeScope();  
var container = builder.Build();  
using (var scope = container.BeginLifetimeScope())  
{  
 var a = scope.Resolve<A>();  
 a.M();  
 a.M();  
 a.M();  
}

### 参数化实例化 (Func<X, Y, B>)

Parameterized Instantiation (Func<X, Y, B>)

你可以使用自动生成的工厂来创建对象实例时提供参数，此时对象的构造函数需要额外的一些参数。虽然 Func<B>关系类似于Resolve<B>()，但Func<X, Y, B>关系则像是调用Resolve<B>(TypedParameter.From(x), TypedParameter.From(y)) ——一个具有类型参数的解析操作。这提供了与 在注册时传递参数 或手动 解析时传递参数 时不同的选择：

public class B  
 {  
 public B(string someString, int id) {}  
 public void DoSomething() {}  
 }  
 public class A  
 {  
   
 Func<int, string, B> \_bFactory;  
 public A(Func<int, string, B> b) { \_bFactory = b }  
 public void M()  
 {  
 var b = \_bFactory(42, "http://hell.owor.ld");  
 b.DoSomething();  
 }  
 }

请注意，由于我们正在解析实例而不是直接调用构造函数，所以无需按照构造函数定义中参数出现的顺序声明它们，也不需要提供构造函数中列出的所有参数。如果构造函数的一部分参数可以由作用域解决，则可以从 Func<X, Y, B>的签名中省略这些参数。你只需要列出作用域无法解决的类型即可。

另一种方法是，如果你已经有了一个具体的实例，可以使用此方法覆盖构造函数参数，即使这个参数原本是要从容器中自动解析的。

示例：

public class B  
 {  
 public B(int id, P peaDependency, Q queueDependency, R ourDependency) {}  
 public void DoSomething() {}  
 }  
 public class A  
 {  
   
 Func<int, P, B> \_bFactory;  
 public A(Func<int, P, B> bFactory) { \_bFactory = bFactory }  
 public void M(P existingPea)  
 {  
   
 var b = \_bFactory(42, existingPea);  
 b.DoSomething();  
 }  
 }

Autofac 根据类型（就像 TypedParameter）确定构造参数的值。由此产生的后果是，自动生成的函数工厂不能在输入参数列表中有重复的类型。 例如，假设你有一个这样的类型：

public class DuplicateTypes  
 {  
 public DuplicateTypes(int a, int b, string c)  
 {  
   
 }  
 }

你可能想要注册该类型，并为其提供一个自动生成的函数工厂。你可以成功地解析函数，但无法执行它。你可以尝试使用每个类型的其中一个参数来解析一个工厂，这将正常工作，但所有相同类型的构造参数都将使用相同的输入值。

var funcWithDuplicates = scope.Resolve<Func<int, int, string, DuplicateTypes>>();  
 var obj1 = funcWithDuplicates(1, 2, "three");  
 var funcWithoutDuplicates = container.Resolve<Func<int, string, DuplicateTypes>>();  
 var obj2 = funcWithoutDuplicates(1, "three");

如果你确实需要多个相同类型的参数，可以查看 委托工厂 。

作用域的生命周期受到这种关系类型的尊重，就像使用 Func<B> 或 委托工厂时一样。如果你将对象注册为InstancePerDependency()，并且多次调用Func<X, Y, B>，每次都会得到一个新的实例。然而，如果你将对象注册为SingleInstance()，并且多次调用Func<X, Y, B> 来解析对象，无论你传递不同的参数，你都会得到同一个对象实例。仅仅传递不同的参数不会打破对作用域生命周期的尊重：

var builder = new ContainerBuilder();  
 builder.RegisterType<A>();  
 builder.RegisterType<B>();  
 builder.RegisterType<Q>();  
 builder.RegisterType<R>();  
 var container = builder.Build();  
 using(var scope = container.BeginLifetimeScope())  
 {  
   
 var a = scope.Resolve<A>();  
 var p = new P();  
   
 a.M(p);  
   
 a.M(p);  
 a.M(p);  
 }

这展示了无论参数如何，作用域的生命周期是如何受到尊重的：

var builder = new ContainerBuilder();  
 builder.RegisterType<B>().InstancePerLifetimeScope();  
 builder.RegisterType<Q>();  
 builder.RegisterType<R>();  
 var container = builder.Build();  
 using(var scope = container.BeginLifetimeScope())  
 {  
   
 var factory = scope.Resolve<Func<int, P, B>>();  
   
 var b1 = factory(10, new P());  
   
 var b2 = factory(17, new P());  
   
 Assert.Same(b1, b2);  
 }

委托工厂允许你为工厂函数提供自定义委托签名，以克服像 Func<X, Y, B>这样的关系带来的挑战，比如支持多个相同类型的参数。委托工厂可能是生成工厂的强大替代方案——请参阅 [高级主题部分](https://docs.autofac.org/en/latest/advanced/delegate-factories.html)中的这一特性。

### 序列（IEnumerable<B>, IList<B>, ICollection<B>）

Enumeration (IEnumerable<B>, IList<B>, ICollection<B>)

依赖于 可枚举类型 提供同一服务（接口）的多个实现。这对于消息处理器等场景很有帮助，其中一条消息进来，注册了多个处理器来处理消息。

假设你有一个依赖接口定义如下：

public interface IMessageHandler  
 {  
 void HandleMessage(Message m);  
 }

此外，你有一个消费者依赖，它需要注册多个并接收所有注册的依赖：

public class MessageProcessor  
 {  
 private IEnumerable<IMessageHandler> \_handlers;  
 public MessageProcessor(IEnumerable<IMessageHandler> handlers)  
 {  
 this.\_handlers = handlers;  
 }  
 public void ProcessMessage(Message m)  
 {  
 foreach(var handler in this.\_handlers)  
 {  
 handler.HandleMessage(m);  
 }  
 }  
 }

你可以轻松地使用隐式序列关系类型来实现这一点。只需注册所有依赖和消费者，当你解析消费者时，所有匹配的依赖项集将被解析为枚举。

var builder = new ContainerBuilder();  
 builder.RegisterType<FirstHandler>().As<IMessageHandler>();  
 builder.RegisterType<SecondHandler>().As<IMessageHandler>();  
 builder.RegisterType<ThirdHandler>().As<IMessageHandler>();  
 builder.RegisterType<MessageProcessor>();  
 var container = builder.Build();  
 using(var scope = container.BeginLifetimeScope())  
 {  
   
 var processor = scope.Resolve<MessageProcessor>();  
 processor.ProcessMessage(m);  
 }

如果容器中没有注册任何匹配项，序列支持将返回一个空集合。也就是说，使用上面的例子，如果没有注册任何IMessageHandler 实现，这将会失败：

scope.Resolve<IMessageHandler>();

但这是可行的：

scope.Resolve<IEnumerable<IMessageHandler>>();

这可能会造成一些“陷阱”，你可能会认为如果使用这种关系注入某物，你会得到 null。实际上，你会得到一个空列表。

### 元数据查询（Meta<B>, Meta<B, X>）

Metadata Interrogation (Meta<B>, Meta<B, X>)

Autofac的 元数据功能 允许你将任意数据与服务关联起来，在解决依赖时可以使用这些数据来做出决策。如果你希望在消费组件中做出这些决策，可以使用Meta<B>关系，它会为你提供一个包含所有对象元数据的字符串/对象字典：

public class A  
 {  
 Meta<B> \_b;  
 public A(Meta<B> b) { \_b = b; }  
 public void M()  
 {  
 if (\_b.Metadata["SomeValue"] == "yes")  
 {  
 \_b.Value.DoSomething();  
 }  
 }  
 }

你还可以使用 强类型元数据 ，通过指定Meta<B, X>关系中的元数据类型：

public class A  
 {  
 Meta<B, BMetadata> \_b;  
 public A(Meta<B, BMetadata> b) { \_b = b; }  
 public void M()  
 {  
 if (\_b.Metadata.SomeValue == "yes")  
 {  
 \_b.Value.DoSomething();  
 }  
 }  
 }

如果你需要为懒加载依赖项获取元数据，可以使用Lazy<B, M>而不是更长的Meta<Lazy<B>, M>。

### 键控服务查找（IIndex<X, B>）

Keyed Service Lookup (IIndex<X, B>)

当你有很多相同类型的服务（如 IEnumerable<B> 关系），但希望根据 服务键 选择一个服务时，可以使用IIndex<X, B> 关系。首先，使用键注册你的服务：

var builder = new ContainerBuilder();  
 builder.RegisterType<DerivedB>().Keyed<B>("first");  
 builder.RegisterType<AnotherDerivedB>().Keyed<B>("second");  
 builder.RegisterType<A>();  
 var container = builder.Build();

然后使用IIndex<X, B>获取带有键的服务字典：

public class A  
 {  
 IIndex<string, B> \_b;  
 public A(IIndex<string, B> b) { \_b = b; }  
 public void M()  
 {  
 var b = this.\_b["first"];  
 b.DoSomething();  
 }  
 }

## 组合关系类型

关系类型可以组合，因此：

IEnumerable<Func<Owned<ITask>>>

会被正确解释为：

* 所有实现
* 工厂，返回
* [所有实例](https://www.koudingke.cn/docs/zh-Hans/autofac-docs/latest/Advanced/Owned-Instances)
* ITask 服务

## 关系类型与容器独立性

基于标准 .NET 类型的 Autofac 自定义关系类型不会强制你的应用程序更紧密地绑定到 Autofac。它们为你提供了一种与编写其他组件一致的容器配置编程模型（而不是必须了解很多特定的容器扩展点和 API，这些可能还会集中你的配置）。

例如，你仍然可以在核心模型中创建自定义ITaskFactory，但如果需要，可以提供一个基于Func<Owned> 的AutofacTaskFactory 实现。

请注意，有些关系是基于 Autofac 中的类型（如IIndex<X, B>）。使用这些关系类型确实会将你绑定到至少引用 Autofac，即使你选择使用不同的 IoC容器来实际解析服务。

# Autofac 控制作用域和生命周期

你可能还记得在 注册主题 中提到的，你向容器添加实现服务的组件。然后你会 解析服务 并使用这些服务实例来完成你的工作。不过，你仍然可能会想知道：

* 组件何时实例化？
* 组件何时被丢弃？
* 如何确保单例在我的应用程序中正确共享？
* 如何控制这些？

*注意：这里大部分内容基于* Nick Blumhardt 的文章 - [Autofac 生命周期入门](https://nblumhardt.com/2011/01/an-autofac-lifetime-primer/)*。虽然随着时间的推移 Autofac 的一些功能有所变化，但其中描述的概念仍然有效，并有助于理解生命周期的作用域。*

## 基本概念和术语

服务的生命周期是指它在应用程序中的存在时间，从最初的实例化到 丢弃 (Disposal)。例如，如果你创建了一个实现了 IDisposable接口的对象，并稍后调用 Dispose()方法，那么该对象的生命周期就是从实例化开始，直到 Dispose()被调用（或者如果没有主动丢弃，垃圾回收器会处理）。

using (var component = new DisposableComponent())  
 {  
   
   
 }

服务的作用域是该服务可以在其中与其他使用它的组件共享的区域。例如，在你的应用程序中，你可能有一个全局静态单例——这个全局对象实例的作用域将是整个应用程序。另一方面，你可能在一个for循环中创建一个局部变量，该变量使用全局单例——局部变量的作用域远小于全局。

public static string Singleton = "single-instance";  
 using (var component = new DisposableComponent())  
 {  
 for (var i = 0; i < 10; i++)  
 {   
 component.DoWork(Singleton, i);  
 }  
 }

Autofac中的生命周期作用域概念结合了这两个概念。实际上，生命周期作用域等同于应用程序中的一个工作单元。一个工作单元可能会开始一个生命周期作用域，然后根据需要从该作用域中获取所需的其他服务。当你从作用域中解析服务时，Autofac会跟踪可丢弃（IDisposable）组件，并在工作单元结束时自动清理它们。

生命周期作用域控制的主要两件事是共享和丢弃。

* 生命周期作用域是嵌套的，它们控制组件如何共享。例如，一个"单例" 服务可能从根生命周期作用域中解析，而单独的工作单元可能需要其自己的其他服务实例。你可以通过 在注册时设置组件实例作用域 来确定组件如何共享。
* 生命周期作用域跟踪可丢弃的对象，并在作用域被丢弃时自动处理它们。例如，如果你有一个实现 IDisposable的组件，并且从生命周期作用域中解析它，作用域将持有它，并为你自动处理丢弃，这样你的服务消费者就不必了解底层实现。你可以选择控制这种行为或添加新的丢弃行为。

当你在应用程序中工作时，记住这些概念可以帮助你最有效地利用资源。

始终从生命周期作用域而不是根容器解析服务非常重要。由于生命周期作用域的丢弃跟踪特性，如果你从容器（根生命周期作用域）大量解析可丢弃组件，你可能会无意中导致内存泄漏。根容器将一直持有这些可丢弃组件的引用，直到应用结束（通常情况下，这是整个应用程序的生命周期），以便它可以丢弃它们。你可以选择更改丢弃行为\_\_，但最好只从作用域中解析。如果 Autofac 检测到使用单例或共享组件，它会自动将其放入适当的跟踪作用域中。

## 作用域和层次结构

可视化生命周期作用域的最简单方法就像一棵树。你从根容器（即根生命周期作用域）开始，每个工作单元（如 Web请求等）——每个子生命周期作用域——从那里分支出来。

![fig:](data:text/html; charset=utf-8;base64,PCFET0NUWVBFIEhUTUwgUFVCTElDICItLy9JRVRGLy9EVEQgSFRNTCAyLjAvL0VOIj48aHRtbD48aGVhZD48dGl0bGU+NDAzIEZvcmJpZGRlbjwvdGl0bGU+PC9oZWFkPjxib2R5PjxoMT40MDMgRm9yYmlkZGVuPC9oMT48cD5Zb3UgZG9uJ3QgaGF2ZSBwZXJtaXNzaW9uIHRvIGFjY2VzcyB0aGUgVVJMIG9uIHRoaXMgc2VydmVyLjwvcD48cD5kZW5pZWQgYnkgVUEgQUNMID0gYmxhY2tsaXN0PC9wPjxoci8+UG93ZXJlZCBieSBUZW5naW5lIDxici8+Q0ROIFJlcXVlc3QgSWQ6IGRkODJjMWFmMTcyNjQ5MTU4ODQzNTQ5NjRlPC9ib2R5PjwvaHRtbD4=)

当你构建 Autofac容器时，你创建的就是那个根容器/生命周期作用域。集成包 或应用程序代码可以从容器创建子生命周期作用域，甚至可以从其他子作用域创建子作用域。

生命周期作用域有助于确定依赖关系来自何处。一般来说，组件会尝试从解析它的作用域中获取依赖项。例如，如果你在一个子生命周期作用域中尝试解析某个东西，Autofac会尝试从子作用域中获取组件的所有依赖项。

影响这一机制的是 “生命周期”方面的 “生命周期作用域”。有些组件，如单例，需要跨多个作用域共享。这会影响依赖项的定位。基本规则如下：

* 子生命周期作用域可以从父作用域获取依赖项，但父作用域不能深入到子作用域中。（你可以通过 “向上移动”在树中查找，但不能 “向下移动”。）
* 组件将从拥有组件的作用域获取其依赖项，即使组件是由树中较深的作用域解析的。我们将在下面的单例生命周期示例中对此进行说明。

生命周期作用域的部分工作是 处理你从作用域中解析的组件的丢弃 。当你解析一个实现IDisposable的组件时，拥有该组件的作用域将持有对该组件的引用，以便在作用域被丢弃时正确地丢弃它。如果你想深入了解如何处理丢弃，你可以考虑一些基本事项：

* 如果你从根生命周期作用域（容器）解析 IDisposable项，它们将被保留，直到容器被丢弃（通常在应用结束时）。这可能导致内存泄漏。总是尝试从子生命周期作用域中解析并处理完作用域后丢弃它们。
* 丢弃父作用域并不会自动丢弃子作用域。以上图为例，如果你丢弃根生命周期作用域，它不会丢弃外面的四个子作用域。负责正确丢弃作用域的责任在于创建作用域的你。
* 如果你丢弃了父作用域但仍继续使用子作用域，事情就会失败。你不能从已丢弃的作用域中解析依赖项。建议按照创建的顺序反向丢弃作用域。

你可以阅读更多关于 与生命周期作用域一起工作（包括更多代码示例！），组件丢弃和可用的不同 实例作用域 的内容。

## 示例：单例生命周期

早些时候我们提到，组件将从拥有组件的作用域获取其依赖项。让我们通过一个例子深入研究：单例。

当你声明单例时，它由声明它的作用域所有。

* 如果你在构建容器时声明单例，它将由根生命周期作用域持有。当你从这种方式注册的单例中解析时，所有依赖项都将来自根作用域。
* 当创建子生命周期作用域时，你可以在其中添加单例——这些将由它们注册的作用域持有。当你从它那里解析时，所有依赖项都将来自该子生命周期作用域。

这样做可以确保你不会在单例之下丢弃依赖项；并且不会因为子作用域被丢弃后仍然持有引用而导致内存泄漏。

假设你有以下几类：

public class Component  
 {  
 private readonly Dependency \_dep;  
 public string Name => this.\_dep.Name;  
 public Component(Dependency dep)  
 {  
 this.\_dep = dep;  
 }  
 }  
 public class Dependency  
 {  
 public string Name { get; }  
 public Dependency(string name)  
 {  
 this.Name = name;  
 }  
 }

现在假设你有一些代码来构建容器并使用这些类。

var builder = new ContainerBuilder();  
 builder.RegisterType<Component>().SingleInstance();  
 builder.Register(ctx => new Dependency("root"));  
 var container = builder.Build();  
 var rootComp = container.Resolve<Component>();  
 Console.WriteLine(rootComp.Name);  
 using (var child1 = container.BeginLifetimeScope(  
 b => b.Register(ctx => new Dependency("child1"))))  
 {  
 child1Comp = child1.Resolve<Component>();  
 Console.WriteLine(child1Comp.Name);  
 }  
 using (var child2 = container.BeginLifetimeScope(  
 b => {  
 b.RegisterType<Component>().SingleInstance();  
 b.Register(ctx => new Dependency("child2"));  
 }))  
 {  
 var child2Comp = child2.Resolve<Component>();  
   
 Console.WriteLine(child2Comp.Name);  
   
 Debug.Assert(rootComp != child2Comp);  
 using (var child2SubScope = child2.BeginLifetimeScope(  
 b => b.Register(ctx => new Dependency("child2SubScope"))))  
 {  
 var child2SubComp = child2SubScope.Resolve<Component>();  
   
 Console.WriteLine(child2SubComp.Name);  
   
 Debug.Assert(child2Comp == child2SubComp);  
 }  
 }

以图片形式表示，看起来像这样：

![fig:](data:text/html; charset=utf-8;base64,PCFET0NUWVBFIEhUTUwgUFVCTElDICItLy9JRVRGLy9EVEQgSFRNTCAyLjAvL0VOIj48aHRtbD48aGVhZD48dGl0bGU+NDAzIEZvcmJpZGRlbjwvdGl0bGU+PC9oZWFkPjxib2R5PjxoMT40MDMgRm9yYmlkZGVuPC9oMT48cD5Zb3UgZG9uJ3QgaGF2ZSBwZXJtaXNzaW9uIHRvIGFjY2VzcyB0aGUgVVJMIG9uIHRoaXMgc2VydmVyLjwvcD48cD5kZW5pZWQgYnkgVUEgQUNMID0gYmxhY2tsaXN0PC9wPjxoci8+UG93ZXJlZCBieSBUZW5naW5lIDxici8+Q0ROIFJlcXVlc3QgSWQ6IGRkODJjMWIxMTcyNjQ5MTU4ODY2OTg4OTdlPC9ib2R5PjwvaHRtbD4=)

如你所见，生命周期作用域不仅决定了组件存活的时间，还决定了它从哪里获取依赖项。在设计应用程序时，你需要考虑这一点，以避免遇到以为自己重写了值，但实际上被组件声明的生命周期所阻碍的问题。更多关于组件实例作用域的信息可以在 我们的实例作用域文档中 找到。

## 示例：Web应用程序

让我们以一个更具体的 Web应用程序为例，说明如何使用生命周期作用域。假设你有以下场景：

* 你有一个全局单例日志服务。
* 同时有两个请求进入 Web应用程序。
* 每个请求是一个逻辑 “工作单元”，每个都需要自己的订单处理服务。
* 每个控制器都需要将信息日志到日志服务。

在这种情况下，你将有一个包含单例日志服务的根生命周期作用域，每个请求有一个子生命周期作用域，每个子作用域都有自己的控制器实例。

从类注册的角度来看，可能如下所示：

var builder = new ContainerBuilder();  
 builder.RegisterType<Logger>().As<ILogger>().SingleInstance();  
 builder.RegisterType<Controller>().InstancePerLifetimeScope();  
 var container = builder.Build();

当 Web请求进来时，生命周期作用域可能会像这样：

![fig:](data:text/html; charset=utf-8;base64,PCFET0NUWVBFIEhUTUwgUFVCTElDICItLy9JRVRGLy9EVEQgSFRNTCAyLjAvL0VOIj48aHRtbD48aGVhZD48dGl0bGU+NDAzIEZvcmJpZGRlbjwvdGl0bGU+PC9oZWFkPjxib2R5PjxoMT40MDMgRm9yYmlkZGVuPC9oMT48cD5Zb3UgZG9uJ3QgaGF2ZSBwZXJtaXNzaW9uIHRvIGFjY2VzcyB0aGUgVVJMIG9uIHRoaXMgc2VydmVyLjwvcD48cD5kZW5pZWQgYnkgVUEgQUNMID0gYmxhY2tsaXN0PC9wPjxoci8+UG93ZXJlZCBieSBUZW5naW5lIDxici8+Q0ROIFJlcXVlc3QgSWQ6IGRkODJjMWFlMTcyNjQ5MTU4ODg3OTU5MzdlPC9ib2R5PjwvaHRtbD4=)

对于像 ASP.NET Core 这样的 Web 应用程序框架，大致的事件顺序可能是这样的：

1. 当 Web请求进来时，Web应用程序框架创建一个子生命周期作用域—— “请求生命周期作用域”。
2. Web应用程序框架从请求生命周期作用域中解析控制器实例。由于控制器注册为 “按生命周期作用域实例”，因此该实例将在该请求中的任何组件之间共享，但不会与其他请求共享。
3. 应用级别注册为单例的日志服务作为依赖项注入到每个控制器实例中。
4. 在每个 Web请求结束时，请求生命周期作用域将被丢弃，控制器将被垃圾回收。日志服务将继续存活并存储在根生命周期作用域中，以便在整个应用程序生命周期中继续注入。
5. 在 Web应用程序结束（在关闭期间）时，Web应用程序框架应丢弃根容器并释放日志服务。

# Autofac处理生命周期范围

**Working with Lifetime Scopes**

## 创建新的生命周期范围

你可以通过调用任何现有生命周期范围的 BeginLifetimeScope()方法来创建一个新范围，从根容器开始。生命周期范围是可丢弃的，并且会跟踪组件的卸载，请确保始终调用 "Dispose()" 或使用 using语句包裹它们。

using (var scope = container.BeginLifetimeScope())  
 {  
   
 var service = scope.Resolve<IService>();  
   
 using (var unitOfWorkScope = scope.BeginLifetimeScope())  
 {  
 var anotherService = unitOfWorkScope.Resolve<IOther>();  
 }  
 }

## 标记生命周期范围

有些情况下，你希望在工作单元之间共享服务，但又不想像单例那样全局共享这些服务。一个常见的例子是在 Web应用程序中的 “每个请求”生命周期（有关更多信息，请参阅 “实例范围”主题 ）。在这种情况下，你希望标记你的生命周期范围，并将服务注册为InstancePerMatchingLifetimeScope()。

例如，假设你有一个发送电子邮件的组件。在系统中，一个逻辑事务可能需要发送多封邮件，因此可以在各个逻辑事务片段之间共享该组件。然而，你不希望电子邮件组件成为全局单例。你的设置可能看起来像这样：

var builder = new ContainerBuilder();  
 builder.RegisterType<EmailSender>()  
 .As<IEmailSender>()  
 .InstancePerMatchingLifetimeScope("transaction");  
 builder.RegisterType<OrderProcessor>()  
 .As<IOrderProcessor>();  
 builder.RegisterType<ReceiptManager>()  
 .As<IReceiptManager>();  
 var container = builder.Build();  
 using (var transactionScope = container.BeginLifetimeScope("transaction"))  
 {  
 using (var orderScope = transactionScope.BeginLifetimeScope())  
 {  
   
 var op = orderScope.Resolve<IOrderProcessor>();  
 op.ProcessOrder();  
 }  
 using (var receiptScope = transactionScope.BeginLifetimeScope())  
 {  
   
 var rm = receiptScope.Resolve<IReceiptManager>();  
 rm.SendReceipt();  
 }  
 }

再次强调，有关更多关于带标签范围和每个请求范围的信息，请参阅 “实例范围”主题 。

## 向生命周期范围添加注册项

Autofac允许你在创建生命周期范围时动态添加注册项。这在你需要进行某种 “临时焊接”有限注册重写或通常只是需要在范围中添加一些不想全局注册的东西时很有帮助。你通过传递一个接受ContainerBuilder的 Lambda表达式给BeginLifetimeScope()来完成这一点。

using (var scope = container.BeginLifetimeScope(  
 builder =>  
 {  
 builder.RegisterType<Override>().As<IService>();  
 builder.RegisterModule<MyModule>();  
 }))  
 {  
   
 }

# Autofac实例范围

Instance Scope

实例范围决定了同一个服务的请求之间如何共享一个实例。请注意，你应该了解 生命周期范围的概念 ，以便更好地理解这里发生的事情。

当请求一个服务时，Autofac可以返回一个单独的实例（单例范围）、一个新的实例（每个依赖范围）或在一个上下文中（如线程或 HTTP请求）中的单个实例（按生命周期范围）。

这同样适用于从显式Resolve()调用返回的实例，以及容器内部创建的实例，以满足其他组件的依赖。

*注意****选择正确的生命周期范围将有助于避免*** ***被囚禁的依赖****以及其他组件生存时间过长或过短的陷阱。开发者需要为应用程序的每个组件做出正确的选择。*

## 按依赖范围实例化

Instance Per Dependency

在其他容器中也称为 “瞬态”或 “工厂”。使用按依赖范围的实例化，对服务的每次请求都会返回一个唯一的实例。

这是默认设置，如果没有指定其他选项。

var builder = new ContainerBuilder();  
 builder.RegisterType<Worker>();  
 builder.RegisterType<Worker>().InstancePerDependency();

当你为按依赖范围的组件进行解析时，每次都会得到一个新的实例。

using(var scope = container.BeginLifetimeScope())  
{  
for(var i = 0; i < 100; i++)  
{  
   
   
 var w = scope.Resolve<Worker>();  
 w.DoWork();  
}  
}

## 单例实例

Single Instance

这也被称为“单例” 。使用单例范围，所有根级和嵌套范围内的请求都将返回一个实例。

var builder = new ContainerBuilder();  
 builder.RegisterType<Worker>().SingleInstance();

当你解析单例组件时，无论在哪里请求，你总是会得到相同的实例。

var root = container.Resolve<Worker>();  
 using(var scope1 = container.BeginLifetimeScope())  
 {  
 for(var i = 0; i < 100; i++)  
 {  
 var w1 = scope1.Resolve<Worker>();  
 using(var scope2 = scope1.BeginLifetimeScope())  
 {  
 var w2 = scope2.Resolve<Worker>();  
 }  
 }  
 }

## 按生命周期范围实例化

Instance Per Lifetime Scope

此范围适用于嵌套的生命周期。具有按生命周期范围实例化的组件最多在每个嵌套生命周期范围内有一个实例。

这对于特定于单个工作单元的对象很有用，这些对象可能需要嵌套更多的逻辑工作单元。每个嵌套生命周期范围将获得注册依赖项的一个新实例。

var builder = new ContainerBuilder();  
 builder.RegisterType<Worker>().InstancePerLifetimeScope();

当你解析按生命周期范围的实例化组件时，你会在每个嵌套范围（例如，按工作单元）内得到一个单一实例。

using(var scope1 = container.BeginLifetimeScope())  
 {  
 for(var i = 0; i < 100; i++)  
 {  
   
 var w1 = scope1.Resolve<Worker>();  
 }  
 }  
 using(var scope2 = container.BeginLifetimeScope())  
 {  
 for(var i = 0; i < 100; i++)  
 {  
   
   
 var w2 = scope2.Resolve<Worker>();  
 }  
 }  
 using(var scope3 = container.BeginLifetimeScope())  
 {  
 var w3 = scope3.Resolve<Worker>();  
 using(var scope4 = scope3.BeginLifetimeScope())  
 {  
   
 var w4 = scope4.Resolve<Worker>();  
 }  
 }  
 var w5 = container.Resolve<Worker>();  
 using(var scope5 = container.BeginLifetimeScope())  
 {  
   
 var w6 = scope5.Resolve<Worker>();  
 }

## 按匹配生命周期范围实例化

Instance Per Matching Lifetime Scope

这类似于上面的“按生命周期范围实例化”的概念，但提供了更精确的实例共享控制。

当你创建一个嵌套生命周期范围时，你可以 “标记”或 “命名”范围。具有按匹配生命周期范围的组件最多在一个具有给定名称的嵌套生命周期范围内有一个实例。这允许你在没有全局共享实例的情况下创建一种 “按范围的单例”，其他嵌套生命周期范围可以共享组件实例。

这对于特定于单个工作单元的对象很有用，例如 HTTP请求，因为可以为每个工作单元创建一个嵌套生命周期。如果为每个 HTTP请求创建了一个嵌套生命周期，那么任何具有按生命周期范围的组件都将有一个 HTTP请求的实例。（关于按请求生命周期范围的更多信息，请参阅下面。）

在大多数应用中，只需要一层容器嵌套就足以表示工作的范围。如果需要更多的嵌套层次（例如：全局 >请求 >事务），可以在层次结构中的特定级别配置组件共享，使用标签。

var builder = new ContainerBuilder();  
 builder.RegisterType<Worker>().InstancePerMatchingLifetimeScope("my-request");

提供的标签值与你启动范围时关联的生命周期范围相关联。如果你试图解析一个按匹配生命周期范围的组件，而没有匹配名称的正确范围，你会收到一个异常。

using(var scope1 = container.BeginLifetimeScope("my-request"))  
 {  
 for(var i = 0; i < 100; i++)  
 {  
 var w1 = scope1.Resolve<Worker>();  
 using(var scope2 = scope1.BeginLifetimeScope())  
 {  
 var w2 = scope2.Resolve<Worker>();  
   
   
 }  
 }  
 }  
 using(var scope3 = container.BeginLifetimeScope("my-request"))  
 {  
 for(var i = 0; i < 100; i++)  
 {  
   
 var w3 = scope3.Resolve<Worker>();  
 using(var scope4 = scope3.BeginLifetimeScope())  
 {  
 var w4 = scope4.Resolve<Worker>();  
   
   
 }  
 }  
 }  
 using(var noTagScope = container.BeginLifetimeScope())  
 {  
   
 var fail = noTagScope.Resolve<Worker>();  
 }

## 按请求实例化

Instance Per Request

某些应用类型自然适合 “请求”类型的语义，例如 ASP.NET的 Web Forms和 MVC应用。在这种应用类型中，能够有一个 “每个请求的单例”很有帮助。

按请求实例化建立在按匹配生命周期范围实例化之上，提供了一个众所周知的生命周期范围标签、注册便利方法和常见应用类型的集成。但背后还是按匹配生命周期范围实例化。

这意味着如果你尝试解析注册为按请求实例化的组件，但是没有当前的请求...你会收到一个异常。

有关如何处理按请求生命周期的信息详细FAQ

var builder = new ContainerBuilder();  
 builder.RegisterType<Worker>().InstancePerRequest();

ASP.NET Core 使用按生命周期范围实例化而不是按请求实例化。有关更多信息，请参阅 ASP.NET Core 集成文档。

## 按所有者实例化

Instance Per Owned

InstancePerOwned

Owned\_ 隐式关系类型 创建新的嵌套生命周期范围。可以使用按所有者实例化注册来将依赖项绑定到所有者实例上。

var builder = new ContainerBuilder();  
 builder.RegisterType<MessageHandler>();  
 builder.RegisterType<ServiceForHandler>().InstancePerOwned<MessageHandler>();

在这个例子中，ServiceForHandler 服务将绑定到所有者的生命周期MessageHandler 实例上。

using(var scope = container.BeginLifetimeScope())  
{  
  
var h1 = scope.Resolve<Owned<MessageHandler>>();  
h1.Dispose();  
}

## 线程范围

Autofac可以强制执行一个线程绑定的对象不会满足另一个线程绑定组件的依赖项。虽然没有便捷的方法实现这一点，但你可以使用生命周期范围来实现。

var builder = new ContainerBuilder();  
 builder.RegisterType<MyThreadScopedComponent>()  
 .InstancePerLifetimeScope();  
 var container = builder.Build();

然后，每个线程都有自己的生命周期范围：

void ThreadStart()  
{  
 using (var threadLifetime = container.BeginLifetimeScope())  
 {  
 var thisThreadsInstance = threadLifetime.Resolve<MyThreadScopedComponent>();  
 }  
}

**重要提示**：考虑到多线程场景，你需要非常小心，确保父范围不会在创建的子线程下被释放。如果你在创建线程后释放父范围，可能会导致组件无法解析。

通过ThreadStart() 执行的每个线程都会获得MyThreadScopedComponent 的一个实例——这本质上是生命周期范围内的单例。由于范围实例永远不会提供给外部范围，因此更容易保持线程组件的隔离。

你可以在创建线程的代码中注入父生命周期范围，通过参数接收ILifetimeScope。Autofac 知道自动注入当前生命周期范围，并可以从此创建嵌套范围。

public class ThreadCreator  
 {  
 private ILifetimeScope \_parentScope;  
 public ThreadCreator(ILifetimeScope parentScope)  
 {  
 this.\_parentScope = parentScope;  
 }  
 public void ThreadStart()  
 {  
 using (var threadLifetime = this.\_parentScope.BeginLifetimeScope())  
 {  
 var thisThreadsInstance = threadLifetime.Resolve<MyThreadScopedComponent>();  
 }  
 }  
 }

如果你想更严格地限制这个规则，可以使用与匹配生命周期范围关联的实例（如上所述），将线程范围组件与内部生命周期关联起来（它们仍然会从外部容器中的工厂/单例组件接收依赖项）。这种方法的效果如下：

![fig:](data:text/html; charset=utf-8;base64,PCFET0NUWVBFIEhUTUwgUFVCTElDICItLy9JRVRGLy9EVEQgSFRNTCAyLjAvL0VOIj48aHRtbD48aGVhZD48dGl0bGU+NDAzIEZvcmJpZGRlbjwvdGl0bGU+PC9oZWFkPjxib2R5PjxoMT40MDMgRm9yYmlkZGVuPC9oMT48cD5Zb3UgZG9uJ3QgaGF2ZSBwZXJtaXNzaW9uIHRvIGFjY2VzcyB0aGUgVVJMIG9uIHRoaXMgc2VydmVyLjwvcD48cD5kZW5pZWQgYnkgVUEgQUNMID0gYmxhY2tsaXN0PC9wPjxoci8+UG93ZXJlZCBieSBUZW5naW5lIDxici8+Q0ROIFJlcXVlc3QgSWQ6IGRkODJjMTk2MTcyNjQ5MTU4OTAzMzM4MzdlPC9ib2R5PjwvaHRtbD4=)

图中的 “上下文”指的是使用BeginLifetimeScope()创建的容器。

# Autofac被囚禁的依赖

Captive Dependencies

被俘获的依赖

A "captive dependency" occurs when a component intended to live for a short amount of time gets held by a component that lives for a long time. This [blog article from Mark Seemann](http://blog.ploeh.dk/2014/06/02/captive-dependency/) does a good job of explaining the concept.

某个组件的生命周期被另一个组件不当地延长了。在软件工程中，这通常指的是一个长期存在的组件（如单例）持有一个应该短期存在的组件（如请求范围的组件），导致后者无法在预期的时间内被释放。

当一个设计用于短期存在时间的组件被一个长期存在的组件所持有时，就会发生 “被囚禁的依赖”现象。[Mark Seemann在这篇文章](https://www.koudingke.cn/go?link=http%3a%2f%2fblog.ploeh.dk%2f2014%2f06%2f02%2fcaptive-dependency%2f)中很好地解释了这个概念。

Autofac 并不会阻止你创建被囚禁的依赖。你可能会因为被囚禁的依赖的设置方式而得到解决异常，但并不总是这样。停止产生被囚禁的依赖是开发者的责任。

## 一般规则

避免被囚禁依赖的一般规则：

消费组件的生命周期应该小于或等于它所消费的服务的生命周期。

简单来说，不要让单例依赖一个“按请求实例”的服务，因为它会被持有太长时间。

## 简单示例

假设你有一个 Web应用程序，它使用传入请求的信息来决定应该连接到哪个数据库。你可能有以下组件：

* 一个接收当前请求和数据库连接工厂的 *存储库*。
* 类似于HttpContext的 *当前请求*，可以用来帮助确定业务逻辑。
* 接收某种参数并返回正确数据库连接的 *数据库连接工厂*。

在这个例子中，考虑你想为每个组件使用的 生命周期范围 。*当前请求上下文*是一个明显的例子——你想要 “按请求实例”。其他组件呢？

对于 *存储库*，假设你选择 “单例”。单例只会在应用生命周期中创建一次并缓存。如果你选择 “单例”，请求上下文会被传递进来，并且在应用整个生命周期中都会被持有——即使那个当前请求已经结束，过时的请求上下文也会被持有。*存储库*生命周期较长，但它持有一个寿命较短的组件。这就是一个被囚禁的依赖。

但是，如果你将 *存储库*改为 “按请求实例”，现在它的生存期与当前请求相同，不再更长。这正好与它所需的请求上下文一样长，所以现在它不是被囚禁的。*存储库*和请求上下文将在同一时间释放（在请求结束时），一切都会好起来。

再进一步，假设你将 *存储库*改为 “按依赖实例”以每次获取一个新的。这仍然是可以的，因为它打算生存的时间比当前请求要短。它不会长时间持有请求，因此没有被囚禁。

数据库连接工厂也会进行类似的思考过程，但可能需要考虑不同的因素。也许工厂的初始化成本较高，或者需要维护一些内部状态才能正常工作。你可能不希望它是 “按请求实例”或 “按依赖实例”。实际上，你可能确实需要它作为一个单例。

短生命周期的依赖可以持有长生命周期的依赖。如果你的 *存储库*是 “按请求实例”或 “按依赖实例”，你仍然会没事。数据库连接工厂故意活得更久。

## 代码示例

下面的单元测试展示了如何强制创建一个被囚禁的依赖。在这个例子中，使用了一个 “规则管理器”来处理一组 “规则”，这些规则在整个应用程序中被使用。

public class RuleManager  
 {  
 public RuleManager(IEnumerable<IRule> rules)  
 {  
 this.Rules = rules;  
 }  
 public IEnumerable<IRule> Rules { get; private set; }  
 }  
 public interface IRule { }  
 public class SingletonRule : IRule { }  
 public class InstancePerDependencyRule : IRule { }  
 [Fact]  
 public void CaptiveDependency()  
 {  
 var builder = new ContainerBuilder();  
   
 builder.RegisterType<RuleManager>()  
 .SingleInstance();  
   
 builder.RegisterType<InstancePerDependencyRule>()  
 .As<IRule>();  
   
 builder.RegisterType<SingletonRule>()  
 .As<IRule>()  
 .SingleInstance();  
 using (var container = builder.Build())  
 using (var scope = container.BeginLifetimeScope("request"))  
 {   
 var manager = scope.Resolve<RuleManager>();  
 }  
 }

请注意，上面的例子并没有直接显示，但如果你在container.BeginLifetimeScope() 调用中动态添加规则注册，那么这些动态注册将不会包含在解析出的RuleManager 中。规则管理器作为单例，从包含动态注册的根容器中解析。

另一个代码示例显示了如何在创建一个错误地绑定到子生命周期范围的被囚禁依赖时，可能会得到一个异常。

public class RuleManager  
 {  
 public RuleManager(IEnumerable<IRule> rules)  
 {  
 this.Rules = rules;  
 }  
 public IEnumerable<IRule> Rules { get; private set; }  
 }  
 public interface IRule { }  
 public class SingletonRule : IRule  
 {  
 public SingletonRule(InstancePerRequestDependency dep) { }  
 }  
 public class InstancePerRequestDependency : IRule { }  
 [Fact]  
 public void CaptiveDependency()  
 {  
 var builder = new ContainerBuilder();  
   
 builder.RegisterType<RuleManager>()  
 .SingleInstance();  
   
 builder.RegisterType<SingletonRule>()  
 .As<IRule>()  
 .SingleInstance();  
   
 builder.RegisterType<InstancePerRequestDependency>()  
 .As<IRule>()  
 .InstancePerMatchingLifetimeScope("request");  
 using (var container = builder.Build())  
 using (var scope = container.BeginLifetimeScope("request"))  
 {  
 Assert.Throws<DependencyResolutionException>(() => scope.Resolve<RuleManager>());  
 }  
 }

## 规则的例外

考虑到应用程序开发者最终负责确定是否可以接受被囚禁的依赖，开发者可能会确定单例（例如）可以接受一个 “按依赖实例”的服务。

例如，也许你有一个故意设置为只存活于消费组件生命周期的缓存类。如果消费者是单例，缓存可以存储整个应用生命周期中的数据；如果消费者是“按请求实例”，它只存储单个网络请求的数据。在这种情况下，你可能会有意无意地让一个生命周期较长的组件依赖于一个生命周期较短的组件。

只要应用程序开发者理解以这样的生命周期设置事物的后果，这就可以接受。也就是说，如果你要这样做，请有意识地去做，而不是意外地。

# Autofac 释放

Disposal

disposable: 有一次性的意思，比如: disposable chopsticks/cup

在单个工作单元中获取的资源（数据库连接、事务、已验证的会话、文件句柄等）应在工作完成时进行dispose。.NET 提供了IDisposable 接口来帮助实现这种更确定的dispose方式。

对于某些 IoC 容器，需要明确地告诉它们dispose某个特定实例，例如通过ReleaseInstance() 方法。这使得确保使用正确的dispose语义变得非常困难。

* 从非dispose组件切换到dispose组件可能需要修改客户端代码。
* 使用共享实例时可能会忽略dispose操作的客户端代码，在切换到非共享实例时几乎肯定会忘记清理。

Autofac 通过生命周期范围 解决这些问题，以便在单个工作单元中dispose所有创建的组件。

using (var scope = container.BeginLifetimeScope())  
{  
 scope.Resolve<DisposableComponent>().DoSomething();  
   
}

在工作单元开始时创建一个生命周期范围，当工作单元完成时，嵌套容器可以dispose范围内所有超出作用域的实例。

## 注册组件

虽然Autofac可以自动dispose一些组件，但你也可以手动指定dispose机制。

组件必须注册为InstancePerDependency()（默认）或InstancePerLifetimeScope() 的某种变体（如InstancePerMatchingLifetimeScope() 或InstancePerRequest()）。

如果你注册了单例组件（作为SingleInstance()），它们将随着容器的存在而存在。由于容器的生命周期通常与应用程序生命周期相同，这意味着组件将在应用程序结束时才被dispose。

### 自动 dispose

Automatic Disposal

要利用自动确定性的dispose，你的组件必须实现IDisposable。然后，你可以根据需要注册组件，并在每次使用生命周期范围解析组件后，组件的Dispose()方法会被调用。

var builder = new ContainerBuilder();  
 builder.RegisterType<SomeDisposableComponent>();  
 var container = builder.Build();

### 异步disposal 支持

Asynchronous Disposal Support

如果你的组件的dispose行为需要一些 I/O操作，例如将缓冲区的内容写入文件或将数据包发送到网络以关闭连接，那么你可能需要考虑实现新的 .NET接口 [IAsyncDisposable](https://www.koudingke.cn/go?link=https%3a%2f%2fdocs.microsoft.com%2fen-us%2fdotnet%2fapi%2fsystem.iasyncdisposable%3fview%3dnetstandard-2.1)。

在 Autofac 5.0 中，增加了对IAsyncDisposable 接口的支持，现在可以异步dispose生命周期范围：

class MyComponent : IDisposable, IAsyncDisposable  
 {  
 INetworkResource myResource;  
 public void Dispose()  
 {  
 myResource.Close();  
 }  
 public async ValueTask DisposeAsync()  
 {  
 await myResource.CloseAsync();  
 }  
 }  
 await using (var scope = container.BeginLifetimeScope())  
 {  
 var service = scope.Resolve<MyComponent>();  
   
 }

当一个生命周期范围被异步dispose时，任何注册的实现了IAsyncDisposable 的组件除了IDisposable 还会调用DisposeAsync() 方法，而不是Dispose() 方法。

如果组件仅实现了同步的Dispose() 方法，则在异步dispose生命周期范围时仍会调用它。

当使用Autofac 与 ASP.NET Core 集成时，所有按请求的生命周期范围都会被异步dispose。

**重要:**

尽管实现IAsyncDisposable 不强制要求实现IDisposable ，但我们强烈建议你这样做。

如果你的组件只实现了IAsyncDisposable ，但有人同步dispose范围，Autofac 将被迫使用同步阻塞的dispose，会发出一个诊断警告：

*AUTOFAC: 试图进行同步dispose，但跟踪的对象类型为* ***AsyncComponent****只实现了* ***IAsyncDisposable****。这会导致效率低下的阻塞dispose。考虑在* ***AsyncComponent****上实现* ***IDisposable****，或者使用* ***DisposeAsync****来dispose范围/容器。*

### 指定dispose

如果你的组件不实现IDisposable，但在生命周期范围结束时仍需要一些清理操作，你可以使用OnRelease生命周期事件 。

var builder = new ContainerBuilder();  
builder.RegisterType<SomeComponent>()  
 .OnRelease(instance => instance.CleanUp());  
var container = builder.Build();

请注意，OnRelease() 覆盖了IDisposable.Dispose() 的默认处理。如果你的组件同时实现了IDisposable 和需要其他清理方法，你可能需要在OnRelease() 中手动调用Dispose()，或者更新你的类，使其在Dispose() 中调用清理方法。

### 禁用dispose

默认情况下，组件由容器所有并会在适当的时候dispose。要禁用此行为，请将组件注册为外部所有：

builder.RegisterType<SomeComponent>().ExternallyOwned();

容器永远不会调用外部所有注册对象的 Dispose() 或 DisposeAsync()。dispose此类组件的责任在于你。

另一种禁用dispose的替代方案是使用隐式关系Owned 和 拥有实例 。在这种情况下，你在消费代码中不再依赖类型T，而是依赖Owned。消费代码将负责dispose。

public class Consumer  
 {  
 private Owned<DisposableComponent> \_service;  
 public Consumer(Owned<DisposableComponent> service)  
 {  
 \_service = service;  
 }  
 public void DoWork()  
 {  
   
 \_service.Value.DoSomething();  
   
 \_service.Dispose();  
 }  
 }

有关Owned的更多信息，请参阅 拥有实例 话题。

## 从生命周期范围解析组件

生命周期范围通过调用BeginLifetimeScope() 创建。最简单的方法是在using 块中。使用生命周期范围来解析你的组件，然后在工作单元完成时dispose范围。

using (var lifetime = container.BeginLifetimeScope())  
{  
 var component = lifetime.Resolve<SomeComponent>();  
   
}

注意，在Autofac集成库 中，标准的工作单元生命周期范围会自动为你创建和dispose。例如，在Autofac的 ASP.NET MVC 集成 中，每个 Web请求开始时会为你创建一个生命周期范围，通常会从那里解析所有组件。在 Web请求结束时，范围会自动dispose，无需你额外创建范围。如果你正在使用集成库 ，请了解可用的自动创建范围。

你可以在此处了解更多关于 创建生命周期范围的信息。

## 子范围不会自动dispose

尽管生命周期范围本身实现了IDisposable，但你创建的生命周期范围 不会自动为你dispose。如果你创建了一个生命周期范围，你负责调用Dispose()清理它并触发组件的自动dispose。这可以通过using语句轻松完成，但如果在没有using的情况下创建范围，别忘了在完成时dispose它。

重要的是区分 你创建的范围和 集成库为你创建的范围。你无需管理集成范围（如 ASP.NET请求范围），这些将为你处理。然而，如果你手动创建自己的范围，将负责清理它。

## 提供的实例

如果你向Autofac提供 实例注册 ，Autofac将假设对实例的所有权，并处理其dispose。

var output = new StringWriter();  
 builder.RegisterInstance(output)  
 .As<TextWriter>();

如果你想自己控制实例的dispose，需要将实例注册为ExternallyOwned()。

var output = new StringWriter();  
 builder.RegisterInstance(output)  
 .As<TextWriter>()  
 .ExternallyOwned();

## 更高级别的层次结构

上述演示的最简单且最推荐的资源管理场景是两层的：有一个单一的 “根” 容器，并为每个工作单元创建一个生命周期范围。然而，可以使用 标记生命周期范围 创建更复杂的容器和组件层次结构。

# Autofac 生命周期事件

Autofac 提供了可以在实例生命周期各个阶段挂钩的事件。这些事件在组件注册（或通过附加到IComponentRegistration 接口）时订阅。

## OnPreparing

OnPreparing 事件在需要创建组件的新实例之前触发，但在此之前不会调用OnActivating。

此事件可用于指定 Autofac 在创建组件新实例时会考虑的自定义参数信息。

该事件的主要用途是模拟或拦截 Autofac 通常作为参数传递给组件激活的服务，方法是将提供的PreparingEventArgs 参数的Parameters 属性设置为任何自定义参数。

**提示** 在使用此事件设置参数之前，请考虑是否更适合作为注册时间进行定义，使用*参数注册*。

## OnActivating

OnActivating 事件在使用组件之前触发。在这里，你可以：

在某些情况下，如RegisterType() ，注册的实际类型用于类型解析并由ActivatingEventArgs 使用。例如，以下代码会导致运行时类转换异常：

builder.RegisterType<TConcrete>()   
 .As<TInterface>()   
 .OnActivating(e => e.ReplaceInstance(new TInterfaceSubclass()));

一个简单的解决方案是分两步进行注册：

builder.RegisterType<TConcrete>().AsSelf();  
 builder.Register<TInterface>(c => c.Resolve<TConcrete>())  
 .OnActivating(e => e.ReplaceInstance(new TInterfaceSubclass()));

## OnActivated

OnActivated 事件在组件完全构建后触发。在这里，你可以执行依赖于组件完全构建的应用级任务 - 这应该是非常罕见的。

## OnRelease

OnRelease 事件取代了组件的 标准清理行为 。实现了IDisposable 的组件的标准清理行为是调用Dispose() 方法。未实现IDisposable 的组件或标记为外部拥有的组件的标准清理行为是无操作 - 不做任何事情。OnRelease 使用提供的实现替换这种行为。

# Autofac在容器构建时运行代码

Autofac提供了组件在构建容器时被通知或自动激活的能力。

有三种自动激活机制可用：

在所有情况下，当容器被构建时，组件将被激活。

**注意** **避免过度使用启动逻辑**：在容器构建时运行启动逻辑可能会感觉也很适合用于协调一般的应用程序启动逻辑。**应用程序启动是依赖管理的另一个关注点**。鉴于你可能遇到的顺序和其他挑战，建议你将应用程序启动逻辑与依赖管理逻辑分开。

## 可启动组件

可启动组件是指在容器首次构建时由容器激活，并具有特定方法调用来启动组件操作的组件。

关键在于实现Autofac.IStartable 接口。当容器构建时，组件将被激活，IStartable.Start() 方法将被调用。

这只会发生在每个组件的单个实例第一次构建容器时。手动解析可启动组件不会导致其 Start()方法被调用。不推荐将可启动组件注册为除 SingleInstance()之外的任何内容。

需要像Start()方法那样每次激活时都调用的方法的组件应该使用 生命周期事件 ，如OnActivated。

要创建一个可启动组件，请实现Autofac.IStartable：

public class StartupMessageWriter : IStartable  
 {  
 public void Start()  
 {  
 Console.WriteLine("App is starting up!");  
 }  
 }

然后注册你的组件，并确保将其指定为IStartable，否则操作将不会执行：

var builder = new ContainerBuilder();  
 builder  
 .RegisterType<StartupMessageWriter>()  
 .As<IStartable>()  
 .SingleInstance();

当容器构建时，类型将被激活，并且IStartable.Start()方法将被调用。在这个示例中，将向控制台写入一条消息。

组件启动的顺序未定义，但自 Autofac 4.7.0 以来，如果一个实现了IStartable 的组件依赖于另一个IStartable 组件，则在依赖组件被激活之前，Start() 方法将保证已被调用：

static void Main(string[] args)  
 {  
 var builder = new ContainerBuilder();  
 builder.RegisterType<Startable1>().AsSelf().As<IStartable>().SingleInstance();  
 builder.RegisterType<Startable2>().As<IStartable>().SingleInstance();  
 builder.Build();  
 }  
 class Startable1 : IStartable  
 {  
 public Startable1()  
 {  
 Console.WriteLine("Startable1 activated");  
 }  
 public void Start()  
 {  
 Console.WriteLine("Startable1 started");  
 }  
 }  
 class Startable2 : IStartable  
 {  
 public Startable2(Startable1 startable1)  
 {  
 Console.WriteLine("Startable2 activated");  
 }  
 public void Start()  
 {  
 Console.WriteLine("Startable2 started");  
 }  
 }

输出如下：

Startable1 activated  
Startable1 started  
Startable2 activated  
Startable2 started

## 自动激活组件

自动激活组件是指在容器构建时仅需激活一次的组件。这是一种 “预热” 行为，不需要在组件上调用任何方法，也不需要实现任何接口——只需解析组件实例，而无需持有该实例的引用。

要注册一个自动激活组件，请使用AutoActivate() 注册扩展。

var builder = new ContainerBuilder();  
 builder  
 .RegisterType<TypeRequiringWarmStart>()  
 .AsSelf()  
 .AutoActivate();

*注意：如果你在注册* ***AutoActivate()****组件时****省略*** *AsSelf()****或*** *As()****服务注册调用****，则组件将****仅注册为自动激活****，并且在容器构建后不一定可以按自身进行解析。*

## 构建回调

你可以注册任何任意动作在容器或生命周期范围构建时间发生。构建回调是一个Action ，在返回容器之前，它将在容器构建之前执行。构建回调按照注册的顺序执行：

var builder = new ContainerBuilder();  
 builder  
 .RegisterBuildCallback(c => c.Resolve<DbContext>());  
 var container = builder.Build();

你可以使用构建回调作为另一种自动在容器构建时启动/预热对象的方式。通过它们与生命周期事件OnActivated和SingleInstance注册结合使用来实现这一点。

一个冗长且复杂的单元测试形式示例：

public class TestClass  
{   
 private class Dependency1  
 {  
 public Dependency1(ITestOutputHelper output)  
 {  
 output.WriteLine("Dependency1.ctor");  
 }  
 }  
 private class Dependency2  
 {  
 private ITestOutputHelper output;  
 public Dependency2(ITestOutputHelper output, Dependency1 dependency)  
 {  
 this.output = output;  
 output.WriteLine("Dependency2.ctor");  
 }  
 public void Initialize()  
 {  
 this.output.WriteLine("Dependency2.Initialize");  
 }  
 }  
 private class Dependency3  
 {  
 private ITestOutputHelper output;  
 public Dependency3(ITestOutputHelper output, Dependency1 dependency)  
 {  
 this.output = output;  
 output.WriteLine("Dependency3.ctor");  
 }  
 public void Initialize()  
 {  
 this.output.WriteLine("Dependency3.Initialize");  
 }  
 }  
 private class Dependency4  
 {  
 private ITestOutputHelper output;  
 public Dependency4(ITestOutputHelper output, Dependency2 dependency2, Dependency3 dependency3)  
 {  
 this.output = output;  
 output.WriteLine("Dependency4.ctor");  
 }  
 public void Initialize()  
 {  
 this.output.WriteLine("Dependency4.Initialize");  
 }  
 }  
   
 private ITestOutputHelper \_output;  
 public TestClass(ITestOutputHelper output)  
 {  
 this.\_output = output;  
 }  
 [Fact]  
 public void OnActivatedDependencyChain()  
 {  
 var builder = new ContainerBuilder();  
 builder.RegisterInstance(this.\_output).As<ITestOutputHelper>();  
 builder.RegisterType<Dependency1>().SingleInstance();  
   
 builder.RegisterType<Dependency2>().SingleInstance().OnActivated(args => args.Instance.Initialize());  
 builder.RegisterType<Dependency3>().SingleInstance().OnActivated(args => args.Instance.Initialize());  
 builder.RegisterType<Dependency4>().SingleInstance().OnActivated(args => args.Instance.Initialize());  
   
 builder.RegisterBuildCallback(c => c.Resolve<Dependency4>());  
 builder.RegisterBuildCallback(c => c.Resolve<Dependency2>());  
 builder.RegisterBuildCallback(c => c.Resolve<Dependency1>());  
 builder.RegisterBuildCallback(c => c.Resolve<Dependency3>());  
   
 var container = builder.Build();  
   
 container.Resolve<Dependency1>();  
 container.Resolve<Dependency2>();  
 container.Resolve<Dependency3>();  
 container.Resolve<Dependency4>();  
 }  
}

这个示例单元测试将生成以下输出：

Dependency1.ctor  
Dependency2.ctor  
Dependency3.ctor  
Dependency4.ctor  
Dependency2.Initialize  
Dependency3.Initialize  
Dependency4.Initialize

从输出中可以看出，回调和OnActivated方法按依赖顺序执行。如果你必须使初始化和启动都在依赖顺序（不仅仅是激活/解析）下发生，这是解决办法。

请注意，如果不使用SingleInstance ，则OnActivated 将为每个依赖项的新实例调用一次。由于预热对象通常是单例，并且创建起来很昂贵，这通常正是你想要的。

## 生命周期范围

使用SingleInstance 或InstancePerDependency 以外的方式注册IStartable 或AutoActivate 可能不如你期望的工作。

例如，如果使用 InstancePerLifetimeScope注册，这并不会导致在你创建的每个生命周期范围内都启动一个新的可启动组件。相反，可启动组件将在容器构建时运行。

此外，你不能在命名生命周期范围中使用IStartable 或AutoActivate。在命名范围创建时注册将不会启动组件；相反，它会在尝试启动组件时引发异常，因为命名范围不存在。

static void Main(string[] args)  
 {  
 var builder = new ContainerBuilder();  
   
 builder.RegisterType<Startable1>()  
 .As<IStartable>()  
 .InstancePerMatchingLifetimeScope("unitOfWork");  
 builder.Build();  
 }

如果你需要在特定生命周期范围中启动某些内容，你需要在创建该范围时（即，在BeginLifetimeScope 调用中）进行注册。

static void Main(string[] args)  
 {  
 var builder = new ContainerBuilder();  
 var container = builder.Build();  
 using (var uow = container.BeginLifetimeScope("unitOfWork", b => b.RegisterType<Startable1>().As<IStartable>()))  
 {  
   
 }  
 }

构建回调将在容器级别和范围级别上工作。它们会在指定的级别上运行。

var builder = new ContainerBuilder();  
 builder.RegisterBuildCallback(  
 c => Console.WriteLine("这发生在容器构建时。"));  
 using var container = builder.Build();  
 using var scope = container.BeginLifetimeScope(  
 b => b.RegisterBuildCallback(  
 c => Console.WriteLine("这发生在范围构建时。")));

## 小提示

*尽可能避免启动逻辑*：在容器构建时运行启动逻辑非常方便，但依赖注入容器主要用于对象的连接，而不是应用程序的启动。尽可能地将这些关注点分开是个好主意。

**顺序**：通常，启动逻辑的执行顺序是 IStartable.Start()、AutoActivate和构建回调。然而，这并不是强制性的。例如，如 IStartable文档中所述，事情会按照依赖顺序而不是注册顺序发生。此外，Autofac 保留更改此顺序的权利（例如，重构对 IStartable.Start()和 AutoActivate\_\_的调用到构建回调）。如果你需要控制初始化逻辑的确切执行顺序，最好自己编写初始化逻辑，以便可以控制顺序。

避免在 IStartable.Start或 AutoActivate中创建生命周期范围：如果你的启动逻辑包括从其中解析组件的生命周期范围的创建，那么这个范围还没有执行所有的启动器。通过创建范围，你迫使了一个竞争条件。这种逻辑更适合在容器构建后执行自定义逻辑，而不是作为 IStartable的一部分。  
考虑使用 OnActivated和 SingleInstance进行懒加载：与其使用构建回调或启动逻辑，不如考虑使用生命周期事件 OnActivated以及 SingleInstance\_\_注册，这样初始化可以在对象上发生，但不受容器构建顺序的约束。

# Autofac 使用 JSON 或 XML 配置

大多数 IoC 容器都提供了程序接口以及基于 JSON 或 XML 的配置支持，Autofac 也不例外。

Autofac 通过ContainerBuilder 类鼓励使用程序化配置。通过程序化接口是容器设计的核心。当在编译时无法选择或配置具体类时，推荐使用 JSON 或 XML。

深入研究 JSON或 XML配置之前，请务必阅读 模块 部分，它解释了 JSON或 XML组件注册无法处理的复杂场景。JSON或 XML配置并不是程序化配置功能对功能的替代，因此复杂的场景可能需要结合 JSON和模块一起使用。

## 使用 Microsoft 配置 (4.0+)

*注意****Microsoft 配置适用于 Autofac.Configuration 4.0 及更高版本。****它不适用于该包的早期版本。*

随着 [Microsoft.Extensions.Configuration](https://www.koudingke.cn/go?link=https%3a%2f%2fwww.nuget.org%2fpackages%2fMicrosoft.Extensions.Configuration)的发布，以及 Autofac.Configuration 4.0.0，Autofac 利用了过去受限于应用程序配置文件（如app.config 或web.config）时无法获得的更灵活的配置模型。如果你之前使用过这些配置文件，你需要将配置迁移到新格式，并更新与应用程序容器设置配置的方式。

### 快速入门

设置应用程序配置的基本步骤如下：

1. 设置可以在Microsoft.Extensions.Configuration 中读取的 JSON 或 XML 文件。

* JSON 配置使用Microsoft.Extensions.Configuration.Json
* XML 配置使用Microsoft.Extensions.Configuration.Xml

1. 使用Microsoft.Extensions.Configuration.ConfigurationBuilder 构建配置。
2. 创建一个Autofac.Configuration.ConfigurationModule，并将构建好的Microsoft.Extensions.Configuration.IConfiguration 传递给它。
3. 将Autofac.Configuration.ConfigurationModule 注册到你的容器中。

包含一些简单注册的配置文件看起来像这样：

{  
 "defaultAssembly": "Autofac.Example.Calculator",  
 "components": [  
 {  
 "type": "Autofac.Example.Calculator.Addition.Add, Autofac.Example.Calculator.Addition",  
 "services": [  
 {  
 "type": "Autofac.Example.Calculator.Api.IOperation"  
 }  
 ],  
 "injectProperties": true  
 },  
 {  
 "type": "Autofac.Example.Calculator.Division.Divide, Autofac.Example.Calculator.Division",  
 "services": [  
 {  
 "type": "Autofac.Example.Calculator.Api.IOperation"  
 }  
 ],  
 "parameters": {  
 "places": 4  
 }  
 }  
 ]  
 }

JSON更干净、易读，但如果你更喜欢 XML，同样的配置会看起来像这样：

<?xml version="1.0" encoding="utf-8"?>  
 <autofac defaultAssembly="Autofac.Example.Calculator">  
 <components name="0">  
 <type>Autofac.Example.Calculator.Addition.Add, Autofac.Example.Calculator.Addition</type>  
 <services name="0" type="Autofac.Example.Calculator.Api.IOperation" />  
 <injectProperties>true</injectProperties>  
 </components>  
 <components name="1">  
 <type>Autofac.Example.Calculator.Division.Divide, Autofac.Example.Calculator.Division</type>  
 <services name="0" type="Autofac.Example.Calculator.Api.IOperation" />  
 <injectProperties>true</injectProperties>  
 <parameters>  
 <places>4</places>  
 </parameters>  
 </components>  
 </autofac>

**请注意** XML 中组件和服务的序号命名——这是由于 **Microsoft.Extensions.Configuration**处理有序集合（数组）的方式造成的。

按照以下方式将配置构建并注册到 AutofacContainerBuilder：

var config = new ConfigurationBuilder();  
 config.AddJsonFile("autofac.json");  
 var module = new ConfigurationModule(config.Build());  
 var builder = new ContainerBuilder();  
 builder.RegisterModule(module);

### 默认程序集

你可以在配置中指定一个“默认程序集”选项，以帮助你以更短的形式写类型。如果在类型或接口引用中未指定完整的程序集限定类型名称，将假设其位于默认程序集中。

{  
 "defaultAssembly": "Autofac.Example.Calculator"  
}

### 组件

组件是最常见的注册项。你可以为每个组件指定从生命周期范围到参数的各种内容。

组件在配置中的顶级components 元素中添加。其中包含要注册的所有组件。

这个示例显示了一个组件，它包含了所有选项，只是为了说明语法。在实际组件注册中，你不会在每个注册中使用所有这些选项。

{  
 "components": [  
 {  
 "type": "Autofac.Example.Calculator.Addition.Add, Autofac.Example.Calculator.Addition",  
 "services": [  
 {  
 "type": "Autofac.Example.Calculator.Api.IOperation"  
 },  
 {  
 "type": "Autofac.Example.Calculator.Api.IAddOperation",  
 "key": "add"  
 }  
 ],  
 "autoActivate": true,  
 "injectProperties": true,  
 "instanceScope": "per-dependency",  
 "metadata": [  
 {  
 "key": "answer",  
 "value": 42,  
 "type": "System.Int32, mscorlib"  
 }  
 ],  
 "ownership": "external",  
 "parameters": {  
 "places": 4  
 },  
 "properties": {  
 "DictionaryProp": {  
 "key": "value"  
 },  
 "ListProp": [1, 2, 3, 4, 5]  
 }  
 }  
 ]  
 }

| 元素名称 | 描述 | 有效值 |
| --- | --- | --- |
| type | 唯一必需的。组件的 concrete 类（如果在非默认程序集中的话，需要指定完整的程序集限定类型名称）。 | 可以通过反射创建的任何 .NET类型名称。 |
| services | 组件公开的服务的数组。每个服务必须有type，可选地指定key。 | 可以通过反射创建的任何 .NET类型名称。 |
| autoActivate | 一个布尔值，指示组件是否应该 自动激活 。 | true,false |
| injectProperties | 一个布尔值，指示是否应 为组件启用属性注入 。 | true,false |
| instanceScope | 组件的 实例范围 。 | singleinstance,perlifetimescope,perdependency,perrequest |
| metadata | 一个 元数据值数组 ，用于关联组件。每个条目指定name、type 和value。 | 任何 元数据值 。 |
| ownership | 允许你控制 组件的生命周期范围是否负责释放组件，还是由你的代码负责 。 | lifetimescope,external |
| parameters | 一个名值字典，其中每个元素的名称是构造函数参数的名称，值是要注入的值。 | 组件类型的构造函数中的任意参数。 |
| properties | 一个名值字典，其中每个元素的名称是属性的名称，值是要注入的值。 | 组件类型上的任何可设置属性。 |

注意，parameters 和properties 都支持字典和枚举值。你可以在 JSON 结构的示例中看到如何指定这些内容。

### 模块

在使用 Autofac时，可以将模块与组件一起使用配置来注册。模块作为配置中的顶级modules元素内的一个数组添加。

以下示例显示了一个具有所有选项的模块，只是为了说明语法。实际上，在每个模块注册时，你不会使用所有这些选项。

{  
 "modules": [  
 {  
 "type": "Autofac.Example.Calculator.OperationModule, Autofac.Example.Calculator",  
 "parameters": {  
 "places": 4  
 },  
 "properties": {  
 "DictionaryProp": {  
 "key": "value"  
 },  
 "ListProp": [1, 2, 3, 4, 5]  
 }  
 }  
 ]  
}

| 元素名称 | 描述 | 有效值 |
| --- | --- | --- |
| type | 必须填写。模块的具体类（如果在默认以外的其他程序集中，需要提供完整的类型名）。 | 任何从Autofac.Module 派生且可以通过反射创建的.NET 类型。 |
| parameters | 名值字典，其中每个元素的名称是构造函数参数的名称，值是要注入的值。 | 模块类型构造函数中的任何参数。 |
| properties | 名值字典，其中每个元素的名称是属性的名称，值是要注入的值。 | 模块类型的任何可设置属性。 |

请注意，parameters 和properties 都支持字典和枚举值。上面的 JSON 结构中已经展示了如何指定这些。

如果愿意，你可以根据不同的参数/属性集多次注册同一个模块。

### 类型名称

在所有情况下，只要看到类型名称（组件类型、服务类型、模块类型），预期它将是标准的、完整类型名（[MSDN 文档](https://www.koudingke.cn/go?link=https%3a%2f%2fmsdn.microsoft.com%2fen-us%2flibrary%2fyfsftwz6(v%3dvs.110).aspx)），通常可以传递给Type.GetType(string typename)。如果类型在defaultAssembly 中，你可以省略 assembly 名称，但无论如何添加它都没有坏处。

完整的类型名包含命名空间、逗号和 assembly 名称，例如Autofac.Example.Calculator.OperationModule, Autofac.Example.Calculator。在这种情况下，Autofac.Example.Calculator.OperationModule 是类型，它位于Autofac.Example.Calculator assembly 中。

泛型稍微复杂一些。配置不支持开放泛型，因此也需要指定每个泛型参数的完全限定名称。

例如，假设在ConfigWithGenericsDemo assembly 中有一个泛型接口IRepository，并且有一个名为StringRepository 的类实现了IRepository。要在配置中注册它，如下所示：

{  
 "components": [  
 {  
 "type": "ConfigWithGenericsDemo.StringRepository, ConfigWithGenericsDemo",  
 "services": [  
 {  
 "type": "ConfigWithGenericsDemo.IRepository`1[[System.String, mscorlib]], ConfigWithGenericsDemo"  
 }  
 ]  
 }  
 ]  
 }

如果你难以确定类型名称，可以在代码中使用类似的方法：

System.Diagnostics.Debug.WriteLine(typeof(IRepository<string>).AssemblyQualifiedName);

### 与旧版配置的差异

从基于app.config的旧版（版本 4.0之前）配置迁移到新格式时，需要注意以下几点：

* 没有 ConfigurationSettingsReader。Microsoft.Extensions.Configuration 完全取代了旧的 XML 格式配置。旧版配置文档不适用于 4.0 及更高版本的配置包。
* 多个配置文件处理方式不同。旧版配置有一个files元素，可以自动同时加载几个文件进行配置。现在使用Microsoft.Extensions.Configuration.ConfigurationBuilder 来实现这一点。
* 自动激活已支持。现在可以指定 自动激活组件，这是以前配置中不可用的功能。
* XML 使用子元素而非属性。这有助于保持 XML 和 JSON 解析器的兼容性，以便正确组合 XML 和 JSON 配置源。
* 使用XML 需要为组件和服务命名并使用数字。Microsoft.Extensions.Configuration 要求每个配置项都有名称和值。它支持有序集合（数组）的方式是为集合中的无名元素自动分配数字名称（如 "0"、"1" 等）。如果你不使用 JSON，需要留意这个要求，否则可能无法得到预期结果。
* 支持按请求生命周期范围。以前无法配置元素以具有 每个请求生命周期范围。现在可以。
* 名称/值中的破折号已移除。XML 元素名称以前包含破折号，如inject-properties，现在为了与 JSON 配置格式兼容，它们使用驼峰命名法，如injectProperties。
* 服务在子元素中指定。旧版配置允许直接在组件顶部声明服务。新系统要求所有服务都在services 集合中。

### 更多提示

新的Microsoft.Extensions.Configuration 机制提供了很多灵活性。你可以利用以下功能：

* 环境变量支持。可以使用Microsoft.Extensions.Configuration.EnvironmentVariables 来根据环境进行配置更改。快速调试、修补或修复代码而无需更改代码的一个方法是根据环境切换 Autofac 注册。
* 易于配置合并。ConfigurationBuilder 允许你从多个来源创建配置并合并它们。如果有很多配置，考虑扫描你的配置文件并动态构建配置，而不是硬编码路径。
* 自定义配置源。你可以实现自己的Microsoft.Extensions.Configuration.ConfigurationProvider，它背后不仅仅依赖于文件。如果你想集中配置，可以考虑使用数据库或 REST API 支持的配置源。

## 使用应用程序配置（旧版预 4.0）

**注意** 下面描述的旧版应用程序配置适用于 3.x 版本及更早的 Autofac.Configuration。它不适用于 4.0 及更高版本的包。

在 [Microsoft.Extensions.Configuration](https://www.koudingke.cn/go?link=https%3a%2f%2fwww.nuget.org%2fpackages%2fMicrosoft.Extensions.Configuration)和更新的配置模型发布之前，Autofac 与标准 .NET 应用程序配置文件（app.config /web.config）集成。在 3.x 版本的 Autofac.Configuration 包中，这是配置的主要方式。

### 设置

使用旧版配置机制，你需要在配置文件的顶部附近声明一个节处理器：

<?xml version="1.0" encoding="utf-8"?>  
 <configuration>  
 <configSections>  
 <section name="autofac" type="Autofac.Configuration.SectionHandler, Autofac.Configuration"/>  
 </configSections>

然后，提供一个描述组件的节：

<autofac defaultAssembly="Autofac.Example.Calculator.Api">  
 <components>  
 <component  
 type="Autofac.Example.Calculator.Addition.Add, Autofac.Example.Calculator.Addition"  
 service="Autofac.Example.Calculator.Api.IOperation" />  
 <component  
 type="Autofac.Example.Calculator.Division.Divide, Autofac.Example.Calculator.Division"  
 service="Autofac.Example.Calculator.Api.IOperation" >  
 <parameters>  
 <parameter name="places" value="4" />  
 </parameters>  
 </component>  
 </components>  
 </autofac>

defaultAssembly属性是可选的，允许使用命名空间限定而非完全限定的类型名称，这可以节省一些混乱和打字，特别是如果你使用一个配置文件对应一个程序集（参见下面的附加配置文件）。

### 组件

组件是最常见的注册项。你可以为每个组件指定许多内容，从生命周期范围到参数。

#### 组件属性

以下属性可用于component 元素（默认值与程序化 API 相同）：

| 属性名称 | 描述 | 有效值 |
| --- | --- | --- |
| type | 必需属性。组件的实现类（如果在非默认命名空间的程序集中，则使用完全限定名）。 | 可以通过反射创建的任何.NET类型名称。 |
| service | 组件公开的服务。对于多个服务，请使用嵌套的services 元素。 | 类似于type 。 |
| instance-scope | 实例作用域 -参见 实例作用域。 | per-dependency、single-instance 或per-lifetime-scope |
| instance-ownership | 容器对实例的所有权 - 参见InstanceOwnership 枚举。 | lifetime-scope 或external |
| name | 组件的字符串名称。 | 任何非空字符串值。 |
| inject-properties | 启用组件的属性（设置器）注入。 | yes、no |

#### 组件子元素

| 元素 | 描述 |
| --- | --- |
| services | 包含组件公开服务类型的列表（参见service 属性）的service 元素。 |
| parameters | 显式构造参数列表，用于设置实例（如上所示的示例）。 |
| properties | 显式属性值列表（与parameters 语法相同）。 |
| metadata | 包含name、value 和type 属性的item 节点列表。 |

XML配置语法中缺少了一些可通过编程 API提供的功能，例如注册泛型。在这种情况下，建议使用模块。

### 模块

使用组件精细配置容器可能会变得繁琐。Autofac支持将组件封装到 模块 中，以封装实现并提供灵活的配置。

通过类型注册模块：

<modules>  
 <module type="MyModule" />

与上述组件类似，你可以在模块注册中添加嵌套的parameters 和properties 。

### 额外配置文件

你可以使用以下方式包含其他配置文件：

<files>  
 <file name="Controllers.config" section="controllers" />

### 配置容器

首先，你的项目必须引用 Autofac.Configuration.dll。

使用ConfigurationSettingsReader 初始化容器配置，其中包含你给 XML 配置节的名称：

var builder = new ContainerBuilder();  
builder.RegisterModule(new ConfigurationSettingsReader("mycomponents"));

容器设置读取器会覆盖已注册的默认组件；你可以编写应用程序，使其在默认情况下运行良好，然后只针对特定部署重写必要的组件注册。

### 多个文件或节

在同一个容器中可以使用多个设置读取器，以读取不同的节，甚至如果提供了文件名给ConfigurationSettingsReader构造函数，还可以读取不同的配置文件。

# Autofac模块

## 简介

IoC使用 组件 作为应用程序的基本构建块。通过提供对组件的构造参数和属性的访问，常常用作实现 部署时配置 的方式。

这通常是一个可疑的做法，原因如下：

* 构造函数可能会更改：组件的构造函数签名或属性的更改可能会破坏已部署的App.config文件——这些问题可能在开发过程的后期才会出现。
* JSON/XML 难以维护：大量组件的配置文件可能变得难以维护。
* 代码开始出现在配置中：暴露类的属性和构造参数是对应用程序内部“封装”的不愉快侵犯——这些细节不属于配置文件。

这就是模块发挥作用的地方。

模块是一个小类，可以用来将一组相关组件包装在一个“外观”后面，以简化配置和部署。模块故意限制了配置参数的数量，这些配置参数可以根据实现模块的组件独立地更改。

模块内的组件仍然在组件/服务级别使用依赖关系来从其他模块访问组件。

模块本身不会经历依赖注入。它们用于配置容器，而不是像其他组件那样注册和解决。例如，如果你的模块接受一个构造参数，你需要自己传递它。它不会来自容器。

## 模块的优势

### 减少配置复杂性

通过 IoC配置应用程序时，常常需要在多个组件之间设置参数。模块将相关配置项组合到一个地方，减少了查找正确设置所对应组件的负担。

模块的实现者确定如何将模块的配置参数映射到内部组件的属性和构造参数。

### 明确的配置参数

直接通过组件配置应用程序会产生一个庞大的表面区域，升级应用程序时需要考虑。如果可以通过配置文件设置任何类的任何属性，并且每个站点都不同，那么重构就不再安全。

创建模块限制了用户可以配置的配置参数，并使维护程序员明确知道这些参数是什么。

你还可以避免在编写良好程序元素与编写良好配置参数之间的权衡。

### 隔离内部应用程序架构

通过组件配置应用程序意味着配置需要根据诸如使用枚举与创建策略类等事情而有所不同。使用模块隐藏了应用程序结构的这些细节，保持配置简洁。

### 更好的类型安全性

当应用程序由可变的类组成时，总是存在一定程度的类型安全性损失。然而，通过 XML配置大量组件会加剧这个问题。

模块是通过程序化方式构建的，所以其中的所有组件注册逻辑都可以在编译时检查。

### 动态配置

模块内的组件配置是动态的：模块的行为可以根据运行环境变化。如果只通过组件配置，这几乎是不可能的。

### 高级扩展

模块可用于不仅仅是简单的类型注册——你还可以附加到组件解析事件，并扩展参数如何解析或其他扩展。log4net 集成模块示例 展示了这样一个模块。

## 示例

在 Autofac 中，模块实现了Autofac.Core.IModule 接口。通常，它们会从Autofac.Module 抽象类派生。

这个模块提供了IVehicle 服务：

public class CarTransportModule : Module  
 {  
 public bool ObeySpeedLimit { get; set; }  
 protected override void Load(ContainerBuilder builder)  
 {  
 builder.RegisterType<Car>(c => c.Resolve<IDriver>()).As<IVehicle>();  
 if (ObeySpeedLimit)  
 builder.RegisterType<SaneDriver>().As<IDriver>();  
 else  
 builder.RegisterType<CrazyDriver>().As<IDriver>();  
 }  
 }

### 封装配置

我们的CarTransportModule 提供了ObeySpeedLimit 配置参数，而不暴露这是通过选择理智或疯狂驾驶员实现的事实。使用模块的客户端可以通过声明其意图来使用它：

builder.RegisterModule(new CarTransportModule() {  
 ObeySpeedLimit = true  
 });

或者在Microsoft.Extensions.Configuration 的 configuration 格式 中：

{  
 "modules": [  
 {  
 "type": "MyNamespace.CarTransportModule, MyAssembly",  
 "properties": {  
 "ObeySpeedLimit": true  
 }  
 }  
 ]  
 }

这很有价值，因为模块的实现可以根据需要而变化，而不会产生连锁效应。毕竟，这就是封装的概念。

### 覆盖灵活性

尽管CarTransportModule 的客户端可能主要关心IVehicle 服务，但模块也向容器注册其IDriver 依赖项。这确保了配置仍然可以在部署时以与独立注册组件相同的方式被覆盖。

使用 Autofac 时，推荐的做法是在程序化配置之后添加任何 XML 配置，例如：

builder.RegisterModule(new CarTransportModule());  
builder.RegisterModule(new ConfigurationSettingsReader());

这样，“紧急” 覆盖可以在 配置文件 中进行：

{  
 "components": [  
 {  
 "type": "MyNamespace.LearnerDriver, MyAssembly",  
 "services": [  
 {  
 "type": "MyNamespace.IDriver, MyAssembly"  
 }  
 ]  
 }  
 ]  
 }

因此，模块增加了封装性，但并不妨碍你在必要时调整其内部。

## 适应部署环境

模块可以是动态的——也就是说，它们可以根据执行环境自定义配置。

加载模块时，它可以做一些有趣的事情，比如检查环境：

protected override void Load(ContainerBuilder builder)  
 {  
 if (Environment.OSVersion.Platform == PlatformID.Unix)  
 RegisterUnixPathFormatter(builder);  
 else  
 RegisterWindowsPathFormatter(builder);  
 }

## 模块的常见用例

* 配置提供子系统的相关服务，例如使用NHibernate 的数据访问
* 包装可选的应用程序功能作为 “插件”
* 提供预构建的包以与系统集成，例如会计系统
* 注册经常一起使用的相似服务，例如一组文件格式转换器
* 新的或定制的容器配置机制，例如使用模块实现JSON/XML配置；可以通过这种方式添加使用属性的配置

# Autofac OWIN 集成

[OWIN (Open Web Interface for .NET)](https://www.koudingke.cn/go?link=http%3a%2f%2fowin.org%2f)是一个更简单的模型，用于构建无需将应用程序绑定到 Web 服务器的基于 Web 的应用程序。为此，它使用了一个名为“中间件”（middleware）的概念，通过该概念可以创建一个请求通过其传递的管道。

由于 OWIN处理应用程序管道的方式（检测请求何时开始和结束等）与更传统的 ASP.NET应用程序有所不同，因此将 Autofac集成到 OWIN应用程序的方法略有不同。[你可以在这篇概述中了解有关OWIN和其工作原理的信息。](https://www.koudingke.cn/go?link=http%3a%2f%2fwww.asp.net%2faspnet%2foverview%2fowin-and-katana%2fan-overview-of-project-katana)

需要记住的重要一点是，OWIN 中间件注册的顺序很重要。中间件按注册顺序处理，就像链一样，所以你需要先注册基础的东西（如 Autofac中间件）。

## 快速入门

要利用 Autofac 在你的 OWIN 管道中：

1. 从 NuGet 引用Autofac.Owin 包。
2. 构建你的 Autofac 容器。
3. 将 Autofac 中间件注册到 OWIN 并传入容器。

public class Startup  
{  
 public void Configuration(IAppBuilder app)  
 {  
 var builder = new ContainerBuilder();  
   
 var container = builder.Build();  
   
 app.UseAutofacMiddleware(container);  
   
 }  
}

请参阅各个 ASP.NET 集成库 页面，以了解不同应用类型以及它们如何支持 OWIN。

## 中间件中的依赖注入

通常，当你使用应用程序注册 OWIN中间件时，会使用随中间件提供的扩展方法。例如，对于 Web API，有app.UseWebApi(config); 这样的扩展方法。在这种方式下注册的中间件是静态定义的，不会注入依赖项。

对于自定义中间件，你可以允许 Autofac将依赖项注入到中间件中，而不是将其注册为静态扩展。

var builder = new ContainerBuilder();  
builder.RegisterType<MyCustomMiddleware>();  
var container = builder.Build();  
app.UseAutofacMiddleware(container);

当你调用app.UseAutofacMiddleware(container); 时，Autofac 中间件本身将被添加到管道中，之后容器中注册的所有Microsoft.Owin.OwinMiddleware 类也将被添加到管道中。

这种方式注册的中间件将为每个通过 OWIN 管道传递的请求在请求生命周期范围内进行解析。

## 控制中间件顺序

对于简单场景，app.UseAutofacMiddleware(container); 将处理向 OWIN 请求范围添加 Autofac 生命周期以及将使用 Autofac 注册的中间件添加到管道中。

如果你希望更多地控制何时将启用依赖注入的中间件添加到管道中，可以使用UseAutofacLifetimeScopeInjector 和UseMiddlewareFromContainer 扩展方法。

var builder = new ContainerBuilder();  
builder.RegisterType<MyCustomMiddleware>();  
var container = builder.Build();  
app.UseAutofacLifetimeScopeInjector(container);  
app.UseWebApi(config);  
app.UseMiddlewareFromContainer<MyCustomMiddleware>();

## 示例

有关 Web API 和 OWIN 自托管示例，请参阅 [Autofac 示例存储库](https://www.koudingke.cn/go?link=https%3a%2f%2fgithub.com%2fautofac%2fExamples%2ftree%2fmaster%2fsrc%2fWebApiExample.OwinSelfHost)中的项目。

# Autofac MVC 集成

Autofac 总是会紧跟最新的 ASP.NET MVC 版本更新，因此文档也会同步更新到最新版本。通常情况下，各个版本之间的集成方式保持相对一致。

要将 ASP.NET MVC 与 Autofac 集成，需要引用 [Autofac.Mvc5 NuGet 包](https://www.koudingke.cn/go?link=https%3a%2f%2fwww.nuget.org%2fpackages%2fAutofac.Mvc5%2f)。

MVC 集成提供了控制器、模型绑定器、动作过滤器和视图的依赖注入支持，并且添加了 生命周期支持 。

此页面解释了 ASP.NET 经典 MVC 的集成。如果你使用的是 ASP.NET Core，请查看 ASP.NET Core 集成页面 。

## 快速入门

为了将 Autofac 与 MVC 集成，你需要引用 MVC 集成的 NuGet 包，注册控制器并设置依赖解析器。还可以选择启用其他功能。

protected void Application\_Start()  
{  
 var builder = new ContainerBuilder();  
   
 builder.RegisterControllers(typeof(MvcApplication).Assembly);  
   
 builder.RegisterModelBinders(typeof(MvcApplication).Assembly);  
 builder.RegisterModelBinderProvider();  
   
 builder.RegisterModule<AutofacWebTypesModule>();  
   
 builder.RegisterSource(new ViewRegistrationSource());  
   
 builder.RegisterFilterProvider();  
   
 builder.InjectActionInvoker();  
   
 var container = builder.Build();  
 DependencyResolver.SetResolver(new AutofacDependencyResolver(container));  
}

下面的部分将详细介绍这些功能以及如何使用它们。

## 注册控制器

在应用程序启动时，在构建 Autofac容器时，应注册你的 MVC控制器及其依赖项。这通常发生在 OWIN启动类或Global.asax的Application\_Start方法中。

var builder = new ContainerBuilder();  
builder.RegisterControllers(typeof(MvcApplication).Assembly);  
builder.RegisterType<HomeController>().InstancePerRequest();

*注意，ASP.NET MVC 通过具体类型请求控制器，所以使用* ***As()****注册是不正确的。如果手动注册控制器并选择指定寿命，必须使用* ***InstancePerDependency()****或* ***InstancePerRequest()****进行注册——****尝试为多个请求重用控制器实例时，ASP.NET MVC 会抛出异常****。*

## 设置依赖解析器

构建容器后，将容器传递给AutofacDependencyResolver 的新实例。使用静态DependencyResolver.SetResolver 方法，让 ASP.NET MVC 知道应该使用AutofacDependencyResolver 来查找服务。这是 Autofac 实现的IDependencyResolver 接口的实现。

var container = builder.Build();  
DependencyResolver.SetResolver(new AutofacDependencyResolver(container));

## 注册模型绑定器

可选步骤之一是为模型绑定器启用依赖注入。与控制器类似，可以在应用程序启动时在容器中注册模型绑定器。可以使用RegisterModelBinders() 方法完成此操作。同时别忘了使用RegisterModelBinderProvider() 扩展方法注册AutofacModelBinderProvider 。这是 Autofac 实现的IModelBinderProvider 接口的实现。

builder.RegisterModelBinders(Assembly.GetExecutingAssembly());  
builder.RegisterModelBinderProvider();

由于RegisterModelBinders() 扩展方法使用 Assembly 扫描添加所需的模型绑定器，因此需要使用ModelBinderTypeAttribute 指定模型绑定器（实现了IModelBinder 的类）应注册的类型。

这样操作：

[ModelBinderType(typeof(string))]  
public class StringBinder : IModelBinder  
{  
 public override object BindModel(ControllerContext controllerContext, ModelBindingContext bindingContext)  
 {  
   
 }  
}

如果一个类需要为多个类型注册，可以在类上添加多个ModelBinderTypeAttribute 。

## 注册 Web 抽象

MVC 集成包含一个 Autofac 模块，它将为 web 抽象类添加 HTTP 请求生命周期 注册，允许你在类中将 web 抽象作为依赖项，并在运行时获取正确的值。

以下包括的抽象类：

* HttpContextBase
* HttpRequestBase
* HttpResponseBase
* HttpServerUtilityBase
* HttpSessionStateBase
* HttpApplicationStateBase
* HttpBrowserCapabilitiesBase
* HttpFileCollectionBase
* RequestContext
* HttpCachePolicyBase
* VirtualPathProvider
* UrlHelper

要使用这些抽象类，只需将AutofacWebTypesModule 添加到容器中，使用标准RegisterModule() 方法。

builder.RegisterModule<AutofacWebTypesModule>();

## 启用视图页属性注入

要使你的 MVC 视图支持属性注入，可以在构建应用程序容器之前将ViewRegistrationSource 添加到ContainerBuilder 。

builder.RegisterSource(new ViewRegistrationSource());

你的视图页必须继承 MVC支持创建视图的基类。使用 Razor视图引擎时，这将是WebViewPage类。

public abstract class CustomViewPage : WebViewPage  
{  
 public IDependency Dependency { get; set; }  
}

当使用 Web Forms 视图引擎时，ViewPage、ViewMasterPage 和ViewUserControl 类是支持的。

public abstract class CustomViewPage : ViewPage  
{  
 public IDependency Dependency { get; set; }  
}

确保实际的视图页从自定义基类继承。对于 Razor视图引擎，可以在.cshtml文件中的@inherits指令中实现：

@inherits Example.Views.Shared.CustomViewPage

对于 Web Forms 视图引擎，你需要在.aspx 文件的@ Page 指令的Inherits 属性中指定。

<%@ Page Language="C#" MasterPageFile="~/Views/Shared/Site.Master" Inherits="Example.Views.Shared.CustomViewPage" %>

由于 ASP.NET MVC 内部的一个问题，属性注入不可用于 Razor 布局页。 view 页 工作正常，但布局页不会。有关更多详细信息，请参阅 [issue#349](https://github.com/autofac/Autofac/issues/349#issuecomment-33025529)。

## 启用动作过滤器属性注入

要使用动作过滤器的属性注入，只需在构建容器之前调用RegisterFilterProvider() 方法，并将容器提供给AutofacDependencyResolver 。

builder.RegisterFilterProvider();

这允许你在过滤器属性上添加属性，并将容器中注册的任何匹配依赖项注入到属性中。

例如，下面的动作过滤器将在容器中注入ILogger 实例（假设已注册ILogger ）。请注意，过滤器本身不需要在容器中注册。

public class CustomActionFilter : ActionFilterAttribute  
{  
 public ILogger Logger { get; set; }  
 public override void OnActionExecuting(ActionExecutingContext filterContext)  
 {  
 Logger.Log("OnActionExecuting");  
 }  
}

其他过滤器属性类型（如授权属性）也可以使用相同简单的方法。

public class CustomAuthorizeAttribute : AuthorizeAttribute  
{  
 public ILogger Logger { get; set; }  
 protected override bool AuthorizeCore(HttpContextBase httpContext)  
 {  
 Logger.Log("AuthorizeCore");  
 return true;  
 }  
}

按照常规将这些属性应用到动作上后，就完成了工作。

[CustomActionFilter]  
[CustomAuthorizeAttribute]  
public ActionResult Index()  
{  
}

## 启用操作参数的注入

虽然不常见，但有些人希望在调用动作方法时由 Autofac填充参数。建议你在控制器上使用构造函数注入而不是动作方法注入，但如果愿意，可以启用动作方法注入：

builder.RegisterType<ExtensibleActionInvoker>().As<IActionInvoker>();  
 builder.InjectActionInvoker();

你也可以使用InjectActionInvoker() 机制与自定义调用者一起使用。

builder.RegisterType<MyCustomActionInvoker>().As<IActionInvoker>();  
builder.InjectActionInvoker();

## OWIN 集成

如果你使用 MVC 作为 OWIN 应用程序的一部分，则需要执行以下操作：

public class Startup  
{  
 public void Configuration(IAppBuilder app)  
 {  
 var builder = new ContainerBuilder();  
   
   
 builder.RegisterControllers(typeof(MvcApplication).Assembly);  
   
   
   
 var container = builder.Build();  
 DependencyResolver.SetResolver(new AutofacDependencyResolver(container));  
   
   
 app.UseAutofacMiddleware(container);  
 app.UseAutofacMvc();  
 }  
}

小问题：MVC 并不能完全运行在OWIN 管道中。它仍然需要HttpContext.Current和一些非 OWIN的东西。在应用程序启动时，当 MVC注册路由时，它会实例化一个IControllerFactory，最终创建两个请求生命周期范围。这只会在应用启动时的路由注册期间发生，而不是在开始处理请求时，但这是需要了解的事情。这是两个管道被搅和在一起的遗留物。[我们曾尝试过寻找解决方法](https://www.koudingke.cn/go?link=https%3a%2f%2fgithub.com%2fautofac%2fAutofac.Mvc%2fissues%2f5)，但未能以一种干净的方式做到这一点。

## 使用“插件”程序集

如果你在一个未被主应用程序引用的“插件程序集”中拥有控制器，[你需要将控制器插件程序集与 ASP.NET BuildManager 注册](https://www.koudingke.cn/go?link=http%3a%2f%2fwww.paraesthesia.com%2farchive%2f2013%2f01%2f21%2fputting-controllers-in-plugin-assemblies-for-asp-net-mvc.aspx)。

你可以通过配置或编程方式来实现。

如果你选择配置，你需要将插件程序集添加到/configuration/system.web/compilation/assemblies 列表中。如果你的插件程序集不在bin 文件夹中，还需要更新/configuration/runtime/assemblyBinding/probing 路径。

<?xml version="1.0" encoding="utf-8"?>  
 <configuration>  
 <runtime>  
 <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">  
   
 <probing privatePath="bin;bin\plugins" />  
 </assemblyBinding>  
 </runtime>  
 <system.web>  
 <compilation>  
 <assemblies>  
 <add assembly="The.Name.Of.Your.Plugin.Assembly.Here" />  
 </assemblies>  
 </compilation>  
 </system.web>  
 </configuration>

如果你选择程序化注册，你需要在 ASP.NETBuildManager 启动之前，在应用程序启动前进行操作。

创建一个初始化类，用于扫描加载程序集并使用BuildManager 进行注册：

using System.IO;  
 using System.Reflection;  
 using System.Web.Compilation;  
 namespace MyNamespace  
 {  
 public static class Initializer  
 {  
 public static void Initialize()  
 {  
 var pluginFolder = new DirectoryInfo(HostingEnvironment.MapPath("~/plugins"));  
 var pluginAssemblies = pluginFolder.GetFiles("\*.dll", SearchOption.AllDirectories);  
 foreach (var pluginAssemblyFile in pluginAssemblyFiles)  
 {  
 var asm = Assembly.LoadFrom(pluginAssemblyFile.FullName);  
 BuildManager.AddReferencedAssembly(asm);  
 }  
 }  
 }  
 }

然后确保使用一个特性来注册你的应用程序启动前代码：

[assembly: PreApplicationStartMethod(typeof(Initializer), "Initialize")]

## 使用当前 Autofac 依赖解析器

一旦你将 MVC 的DependencyResolver 设置为AutofacDependencyResolver ，你可以使用AutofacDependencyResolver.Current 作为获取当前依赖解析器并将其转换为AutofacDependencyResolver 的快捷方式。

不幸的是，关于使用AutofacDependencyResolver.Current 有一些问题，可能导致某些功能无法正常工作。通常这些问题是由使用像 [Glimpse](https://www.koudingke.cn/go?link=http%3a%2f%2fgetglimpse.com%2f)或 [Castle DynamicProxy](https://www.koudingke.cn/go?link=http%3a%2f%2fwww.castleproject.org%2fprojects%2fdynamicproxy%2f)这样的产品引起的，它们“包装”或“装饰”依赖解析器以添加功能。如果当前的依赖解析器被装饰或以其他方式包装/代理，你就不能将其转换为AutofacDependencyResolver ，也没有单一的方法来“解包”它或获取实际的解析器。

在 Autofac MVC集成的 3.3.3版本之前，我们通过动态添加它到请求生命周期范围来跟踪当前的依赖解析器。这让我们绕过了无法从代理中解包AutofacDependencyResolver的问题……但这意味着AutofacDependencyResolver.Current只能在请求生命周期内工作——你不能在后台任务或应用程序启动时使用它。

从 3.3.3版本开始，查找AutofacDependencyResolver.Current逻辑的变化首先尝试将当前依赖解析器转换为类型；然后特别查找使用 [Castle DynamicProxy](https://www.koudingke.cn/go?link=http%3a%2f%2fwww.castleproject.org%2fprojects%2fdynamicproxy%2f)包装的迹象，并通过反射解包它。如果失败……我们找不到当前的AutofacDependencyResolver，因此会抛出一个InvalidOperationException，消息类似于：

*依赖解析器的类型是****'Some.Other.DependencyResolver'****，但预期是类型* ***Autofac.Integration.Mvc.AutofacDependencyResolver****。它看起来也不像是由****Castle Project*** *的****DynamicProxy*** *包裹的。这个问题可能是由****DynamicProxy*** *实现的更改或使用不同的代理库包裹依赖解析器导致的。*

这种问题最常出现在使用ContainerBuilder.RegisterFilterProvider()的 action过滤器提供程序时。过滤器提供程序需要访问 Autofac依赖解析器，并使用AutofacDependencyResolver.Current来执行它。

如果你看到这个，这意味着你在以无法解包的方式装饰解析器，依赖于AutofacDependencyResolver.Current 的功能将失败。目前的解决方案是不要装饰依赖解析器。

## Glimpse 集成

使用 Autofac 的 MVC 应用程序与 Glimpse 的集成与其他集成基本相同。但是，如果你使用动作方法参数注入（例如，通过 builder.InjectActionInvoker()），则 Glimpse 的执行检查会失败。

你可以通过在 Glimpse配置中添加以下内容来解决这个问题：

<glimpse defaultRuntimePolicy="On" endpointBaseUri="~/Glimpse.axd">  
 <inspectors>  
 <ignoredTypes>  
 <add type="Glimpse.Mvc.Inspector.ExecutionInspector, Glimpse.Mvc"/>  
 </ignoredTypes>  
 </inspectors>  
 <tabs>  
 <ignoredTypes>  
 <add type="Glimpse.Mvc.Tab.Execution, Glimpse.Mvc"/>  
 </ignoredTypes>  
 </tabs>  
 </glimpse>

再次强调，只有在使用动作参数注入时才需要这样做。这是推荐使用控制器构造函数注入而不是动作方法参数注入的众多原因之一。

有关更多信息（包括来自 Glimpse的相关信息链接），请参阅 [此问题](https://www.koudingke.cn/go?link=https%3a%2f%2fgithub.com%2fautofac%2fAutofac.Mvc%2fissues%2f7)。

## 单元测试

当你在一个使用 Autofac的 ASP.NET MVC应用程序中进行单元测试，而你注册了InstancePerRequest组件时，你会在尝试解决这些组件时遇到异常，因为在单元测试中没有 HTTP请求生命周期。

每一次请求生命周期范围 主题概述了测试和调试请求范围组件的策略。

## 示例

在 Autofac 示例仓库中有一个展示 ASP.NET MVC 集成的示例项目 [此处](https://www.koudingke.cn/go?link=https%3a%2f%2fgithub.com%2fautofac%2fExamples%2ftree%2fmaster%2fsrc%2fMvcExample)。

# Autofac Web API 集成

Web API 2 集成需要使用 [Autofac.WebApi2 NuGet 包](https://www.koudingke.cn/go?link=https%3a%2f%2fwww.nuget.org%2fpackages%2fAutofac.WebApi2)。而 Web API 1.x 集成则需要使用 [Autofac.WebApi NuGet 包](https://www.koudingke.cn/go?link=https%3a%2f%2fwww.nuget.org%2fpackages%2fAutofac.WebApi%2f)。Web API 集成提供了控制器、模型绑定器和动作过滤器的依赖注入支持，并且添加了 按请求生命周期支持 。

此页面解释了 ASP.NET 经典 Web API 集成。如果你正在使用 ASP.NET Core，请参阅 ASP.NET Core 集成页面。

## 快速入门

要使 Autofac 与 Web API 集成，你需要引用 Web API 集成的 NuGet 包，注册控制器，并设置依赖解析器。你可以选择性地启用其他功能。

protected void Application\_Start()  
 {  
 var builder = new ContainerBuilder();  
   
 var config = GlobalConfiguration.Configuration;  
   
 builder.RegisterApiControllers(Assembly.GetExecutingAssembly());  
   
 builder.RegisterWebApiFilterProvider(config);  
   
 builder.RegisterWebApiModelBinderProvider();  
   
 var container = builder.Build();  
 config.DependencyResolver = new AutofacWebApiDependencyResolver(container);  
 }

下面的各节将详细介绍这些功能以及如何使用它们。

## 获取 HttpConfiguration

在 Web API 中，设置应用程序需要更新HttpConfiguration 对象的属性并设置值。根据你的应用程序部署方式，获取这个配置的方式可能不同。在文档中，我们将提到“你的HttpConfiguration ”，你需要决定如何获取它。

对于标准 IIS 托管，HttpConfiguration 是GlobalConfiguration.Configuration 。

var builder = new ContainerBuilder();  
var config = GlobalConfiguration.Configuration;  
builder.RegisterApiControllers(Assembly.GetExecutingAssembly());  
var container = builder.Build();  
config.DependencyResolver = new AutofacWebApiDependencyResolver(container);

对于自托管，HttpConfiguration 是你的HttpSelfHostConfiguration 实例。

var builder = new ContainerBuilder();  
var config = new HttpSelfHostConfiguration("http://localhost:8080");  
builder.RegisterApiControllers(Assembly.GetExecutingAssembly());  
var container = builder.Build();  
config.DependencyResolver = new AutofacWebApiDependencyResolver(container);

对于 OWIN 集成，HttpConfiguration 是你在应用启动类中创建并传递给 Web API 中间件的那个。

var builder = new ContainerBuilder();  
 var config = new HttpConfiguration();  
 builder.RegisterApiControllers(Assembly.GetExecutingAssembly());  
 var container = builder.Build();  
 config.DependencyResolver = new AutofacWebApiDependencyResolver(container);

## 注册控制器

在应用程序启动时，在构建 Autofac容器时，你应该注册 Web API控制器及其依赖项。这通常发生在 OWIN启动类或Global.asax文件的Application\_Start方法中。

默认情况下，实现IHttpController且名字以Controller结尾的类型会被注册。

var builder = new ContainerBuilder();  
 builder.RegisterApiControllers(Assembly.GetExecutingAssembly());  
 builder.RegisterType<ValuesController>().InstancePerRequest();

如果你的控制器不遵循标准命名约定，可以选择使用RegisterApiControllers 方法的重载来提供自定义后缀。

builder.RegisterApiControllers("MyCustomSuffix", Assembly.GetExecutingAssembly());

## 设置依赖解析器

构建容器后，将容器传递给AutofacWebApiDependencyResolver 的新实例。将新的解析器附加到HttpConfiguration.DependencyResolver ，以便 Web API 知道应该使用AutofacWebApiDependencyResolver 来查找服务。这是 Autofac 实现的IDependencyResolver 接口。

var container = builder.Build();  
config.DependencyResolver = new AutofacWebApiDependencyResolver(container);

## 通过依赖注入提供过滤器

由于属性是通过反射 API创建的，你不能自己调用构造函数。除了属性注入外，你没有其他选择。Autofac的 Web API集成提供了一种机制，让你可以创建实现过滤器接口（如IAutofacActionFilter、IAutofacContinuationActionFilter、IAutofacAuthorizationFilter和IAutofacExceptionFilter）的类，并使用容器 builder 的注册语法将它们与所需的控制器或动作方法关联起来。

### 注册过滤器提供程序

你需要注册 Autofac过滤器提供程序的实现，因为它负责根据注册来连接过滤器。这是通过在容器 builder 上调用RegisterWebApiFilterProvider方法并提供一个HttpConfiguration实例来完成的。

var builder = new ContainerBuilder();  
builder.RegisterWebApiFilterProvider(config);

### 实现过滤器接口

使用现有的 Web API过滤器属性创建类时，你的类应实现集成中定义的适当过滤器接口。

#### 标准动作过滤器接口

IAutofacActionFilter 接口让你可以在执行动作前后定义过滤器，就像你从ActionFilterAttribute 派生一样。

下面的过滤器是一个动作过滤器，它实现了IAutofacActionFilter 而不是System.Web.Http.Filters.IActionFilter 。

public class LoggingActionFilter : IAutofacActionFilter  
 {  
 readonly ILogger \_logger;  
 public LoggingActionFilter(ILogger logger)  
 {  
 \_logger = logger;  
 }  
 public Task OnActionExecutingAsync(HttpActionContext actionContext, CancellationToken cancellationToken)  
 {  
 \_logger.Write(actionContext.ActionDescriptor.ActionName);  
 return Task.FromResult(0);  
 }  
 public Task OnActionExecutedAsync(HttpActionExecutedContext actionExecutedContext, CancellationToken cancellationToken)  
 {  
 \_logger.Write(actionContext.ActionDescriptor.ActionName);  
 return Task.FromResult(0);  
 }  
 }

请注意，示例中没有实际的异步代码运行，所以返回Task.FromResult(0)，这是一种常见的返回“空任务”的方式。如果过滤器确实需要异步代码，你可以返回一个真正的Task对象，或者像其他异步方法那样使用async /await。

#### 继续动作过滤器接口

除了上面的常规IAutofacActionFilter，还有一个IAutofacContinuationActionFilter 。该接口也作为动作过滤器工作，但不使用OnActionExecutingAsync 和OnActionExecutedAsync 方法，而是采用延续风格，只有一个接受回调的方法ExecuteActionFilterAsync ，用于运行链中的下一个过滤器。

你可能会选择使用IAutofacContinuationActionFilter 而不是IAutofacActionFilter ，例如如果你想将整个请求包装在一个using 块中，比如你想为请求分配一个TransactionScope ，如下所示：

public class TransactionScopeFilter : IAutofacContinuationActionFilter  
 {  
 public async Task<HttpResponseMessage> ExecuteActionFilterAsync(  
 HttpActionContext actionContext,  
 CancellationToken cancellationToken,  
 Func<Task<HttpResponseMessage>> next)  
 {  
 using (new TransactionScope(TransactionScopeAsyncFlowOption.Enabled))  
 {  
 return await next();  
 }  
 }  
 }

*注意****常规的*** *IAutofacActionFilter****在延续过滤器内部运行，因此在*** *OnActionExecutingAsync****、动作方法本身和过滤器的*** *OnActionExecutedAsync\_\_之间，异步上下文也会保留。*

### 注册过滤器

为了执行过滤器，你需要将其注册到容器中，并告知它应针对哪个控制器（或控制器），以及可选的动作，进行操作。这通过使用容器 builder 扩展方法完成，每个过滤器类型都有相应的扩展方法：

* ActionFilter
* ActionFilterOverride
* AuthenticationFilter
* AuthenticationFilterOverride
* AuthorizationFilter
* AuthorizationFilterOverrideW
* ExceptionFilter
* ExceptionFilterOverride

对于每个过滤器类型，都有几种注册方法：

· AsWebApi{FilterType}ForAllControllers  
将此过滤器注册到所有控制器的所有动作方法上，就像注册全局 Web API 过滤器一样。

· AsWebApi{FilterType}For()

将过滤器注册到指定控制器上，就像在控制器级别放置基于属性的过滤器一样。

指定基控制器类会导致该过滤器应用于所有继承自它的控制器。

此方法接受一个可选的 lambda表达式，指示控制器上的特定方法，就好像你在应用一个动作过滤器到特定动作一样。

在下面的例子中，动作过滤器被应用于ValuesController 的Get 动作方法。

var builder = new ContainerBuilder();  
 builder.RegisterType<LoggingActionFilter>()  
 .AsWebApiActionFilterFor<ValuesController>(c => c.Get(default(int)))  
 .InstancePerRequest();

当应用于需要参数的动作方法时，使用default关键字和参数的数据类型作为 lambda表达式的强类型占位符。例如，上面示例中的Get 动作方法需要一个int 类型的参数，并在 lambda 表达式中使用default(int) 作为占位符。

· AsWebApi{FilterType}Where()\*Where 方法允许你指定一个谓词，以便对哪些动作和/或控制器进行更高级的自定义决策。

以下示例中，将异常过滤器应用到所有 POST方法上：

var builder = new ContainerBuilder();  
builder.Register(c => new LoggingExceptionFilter(c.Resolve<ILogger>()))  
 .AsWebApiExceptionFilterWhere(action => action.SupportedHttpMethods.Contains(HttpMethod.Post))  
 .InstancePerRequest();

还有一个接受ILifetimeScope 的谓词版本，你可以在此内部使用服务：

var builder = new ContainerBuilder();  
 builder.Register(c => new LoggingExceptionFilter(c.Resolve<ILogger>()))  
 .AsWebApiExceptionFilterWhere((scope, action) => scope.Resolve<IFilterConfig>().ShouldFilter(action))  
 .InstancePerRequest();

*注意****过滤器谓词对于每个动作****/\_\_过滤器组合只会调用一次；它们不会在每次请求上都调用。*

你可以应用任意数量的过滤器。注册一种类型的过滤器不会移除或替换之前已注册的过滤器。

你可以将过滤器注册链接在一起，以针对多个控制器应用过滤器，如下所示：

builder.Register(c => new LoggingActionFilter(c.Resolve<ILogger>()))  
 .AsWebApiActionFilterFor<LoginController>()  
 .AsWebApiActionFilterFor<ValuesController>(c => c.Get(default(int)))  
 .AsWebApiActionFilterFor<ValuesController>(c => c.Post(default(string)))  
 .InstancePerRequest();

### 过滤器覆盖

注册过滤器时，有基本的注册方法（如AsWebApiActionFilterFor() ），以及覆盖注册方法（如AsWebApiActionFilterOverrideFor() ）。

覆盖方法的目的是提供一种确保某些过滤器优先执行的方式。你可以有任意数量的覆盖——这些不是替换过滤器，只是优先执行的过滤器。

过滤器将以以下顺序执行：

* 控制器范围的覆盖
* 动作范围的覆盖
* 控制器范围的过滤器
* 动作范围的过滤器

### 在 Autofac动作过滤器中设置响应

与标准 Web API 过滤器类似，你可以在动作过滤器的OnActionExecutingAsync 方法中设置HttpResponseMessage 。

class RequestRejectionFilter : IAutofacActionFilter  
{  
 public async Task OnActionExecutingAsync(HttpActionContext actionContext, CancellationToken cancellationToken)  
 {  
   
 actionContext.Response = actionContext.Request.CreateErrorResponse(HttpStatusCode.BadRequest, "Request not valid");  
 await Task.FromResult(0);  
 }  
 public void Task OnActionExecutedAsync(HttpActionExecutedContext actionExecutedContext, CancellationToken cancellationToken)  
 {  
 }  
}

要匹配标准 Web API行为，如果你设置了Response属性，则后续的动作过滤器将不会被调用。但是，已经调用过的任何动作过滤器都将调用OnActionExecutedAsync方法，并填充适当的响应。

### 标准 Web API过滤器特性是单例

你可能会注意到，如果你使用标准 Web API过滤器，则无法使用InstancePerRequest依赖项。

与 MVC中的过滤器提供程序不同，Web API中的过滤器提供程序不允许你指定过滤器实例不应缓存。这意味着所有Web API 的过滤器特性本质上都是整个应用程序生命周期内的单例实例。

如果要在过滤器中获取按请求的服务，你会发现只有使用 Autofac过滤器接口时才能实现。使用标准 Web API过滤器时，依赖项将在过滤器首次解决时注入一次，之后将不再注入。

现有的 Web API过滤器特性的单例性质是我们需要自定义过滤器接口的原因。

如果你无法使用 Autofac接口，并且在过滤器中需要按请求或依赖项实例的服务，请使用服务定位。幸运的是，Web API使获取当前请求范围变得非常容易——它随HttpRequestMessage一起提供。

以下是使用 Web API的IDependencyScope服务定位的过滤器示例，以获取按请求的服务：

public class ServiceCallActionFilterAttribute : ActionFilterAttribute  
{  
 public override void OnActionExecuting(HttpActionContext actionContext)  
 {  
   
 var requestScope = actionContext.Request.GetDependencyScope();  
   
 var service = requestScope.GetService(typeof(IMyService)) as IMyService;  
   
 service.DoWork();  
 }  
}

### 实例过滤器不会注入

在设置过滤器时，你可能想手动向集合中添加过滤器，如下所示：

config.Filters.Add(new MyActionFilter());

Autofac 不会为通过这种方式注册的过滤器的属性注入。这有点像你使用 RegisterInstance将预构造的对象放入 Autofac 时的情况——Autofac 不会注入或修改预构造的对象。对于预先构造并添加到过滤器集合中的过滤器实例也是如此。与属性过滤器（如上所述）一样，你可以使用服务定位而不是属性注入来解决这个问题。

## 通过依赖注入提供模型绑定器

Autofac与 Web API的集成提供了通过依赖注入解决你的模型绑定器的能力，并使用一个流式接口将绑定器与类型关联起来。

### 注册绑定器提供程序

为了在需要时能解决任何注册的IModelBinder 实现，你需要注册 Autofac 模型绑定器提供程序。这是通过在容器 builder 上调用RegisterWebApiModelBinderProvider 方法完成的。

var builder = new ContainerBuilder();  
builder.RegisterWebApiModelBinderProvider();

### 注册模型绑定器

一旦实现了System.Web.Http.ModelBinding.IModelBinder 来处理绑定问题，将其与 Autofac 注册，并告诉 Autofac 应使用哪个绑定器处理哪些类型。

builder  
 .RegisterType<AutomobileBinder>()  
 .AsModelBinderForTypes(typeof(CarModel), typeof(TruckModel));

### 为参数标记 ModelBinderAttribute

即使你已注册模型绑定器，但仍需将参数标记为[ModelBinder] 属性，以便 Web API 知道使用模型绑定器而不是媒体类型格式化器来绑定模型。你不再需要指定模型绑定器类型，但需要标记参数。[有关此内容，参阅 Web API 文档](https://www.koudingke.cn/go?link=https%3a%2f%2fdocs.microsoft.com%2fen-us%2faspnet%2fweb-api%2foverview%2fformats-and-model-binding%2fparameter-binding-in-aspnet-web-api)。

public HttpResponseMessage Post([ModelBinder] CarModel car) { ... }

## 按控制器类型的特定服务

Web API 具有一个有趣的功能，允许你通过添加实现IControllerConfiguration 接口的属性来配置按控制器类型的 Web API 服务（例如IActionValueBinder ）。

通过传递给IControllerConfiguration.Initialize 方法的HttpControllerSettings 参数上的Services 属性，你可以覆盖全局设置的服务。这种基于属性的方法似乎鼓励你直接实例化服务对象，然后覆盖全局注册的服务。Autofac允许你通过容器配置这些按控制器类型的特定服务，而不是将它们隐藏在没有依赖注入支持的属性中。

### 添加控制器配置属性

由于 Web API定义了扩展点，因此无法避免在控制器上添加用于应用配置的属性。Autofac集成包含一个AutofacControllerConfigurationAttribute，你可以在 Web API控制器上应用该属性，以表示它们需要按控制器类型的配置。

关键是要记住的是，实际的配置信息将由你构建容器时确定，而无需在实际属性中实现任何配置。在这种情况下，该属性可以被视为纯粹的标记，指示容器将定义配置信息并提供服务实例。

[AutofacControllerConfiguration]  
public class ValuesController : ApiController  
{  
   
}

### 支持的服务

支持的服务可以分为单个样式或多样式服务。例如，你只能有一个IHttpActionInvoker ，但可以有多个ModelBinderProvider 服务。

你可以为以下单个样式服务使用依赖注入：

* IHttpActionInvoker
* HttpActionSelector
* ActionValueBinder
* IBodyModelValidator
* IContentNegotiator
* IHttpControllerActivator
* ModelMetadataProvider

以下多样式服务受支持：

* ModelBinderProvider
* ModelValidatorProvider
* ValueProviderFactory
* MediaTypeFormatter

在多样式服务列表中，MediaTypeFormatter 实际上是与众不同的。从技术上讲，它实际上不是一个服务，而是添加到HttpControllerSettings 实例的MediaTypeFormatterCollection 中，而不是ControllerServices 容器。我们考虑过不支持MediaTypeFormatter 实例的依赖注入，但确保它们也可以按控制器类型从容器中解决。

### 服务注册

下面是一个将自定义的IHttpActionSelector 实现为InstancePerApiControllerType() 的例子，用于ValuesController。当应用到控制器类型时，所有衍生控制器也将接收相同的配置。AutofacControllerConfigurationAttribute 被衍生控制器类型继承，并且容器中对注册的服务的行为相同。当你为单例风格的服务注册时，它总是会替换全局级别配置的默认服务。

builder.Register(c => new CustomActionSelector())  
 .As<IHttpActionSelector>()  
 .InstancePerApiControllerType(typeof(ValuesController));

### dispose现有服务

默认情况下，多个风格的服务会被附加到全局级别已配置的服务集中。当你使用容器注册多个风格的服务时，可以选择dispose现有的服务集，以便只使用你注册为InstancePerApiControllerType() 的服务。这可以通过在InstancePerApiControllerType() 方法上设置clearExistingServices 参数为true 来完成。如果任何多个风格服务的注册表示希望发生这种情况，将移除该类型的现有服务。

builder.Register(c => new CustomModelBinderProvider())  
 .As<ModelBinderProvider>()  
 .InstancePerApiControllerType(  
 typeof(ValuesController),  
 clearExistingServices: true);

### 按控制器类型的服务限制

如果你使用按控制器类型的服务，就不能依赖注册为InstancePerRequest()的其他服务。问题在于 Web API会缓存这些服务，并不是每次创建该类型的控制器时都会从容器中请求它们。在不引入依赖注入集成的关键概念（即控制器类型的键）的情况下，Web API很可能无法轻松添加这种支持，而这意味着所有容器都需要支持键值对服务。

## 批处理

如果你选择使用 [Web API批处理功能](https://www.koudingke.cn/go?link=https%3a%2f%2fblogs.msdn.microsoft.com%2fwebdev%2f2013%2f11%2f01%2fintroducing-batch-support-in-web-api-and-web-api-odata%2f)，请注意，向批处理端点发送的初始多部分请求是 Web API创建请求生命周期范围的地方。批处理中的子请求都发生在内存中，并共享同一个请求生命周期范围 -批处理中的每个子请求不会得到单独的生命周期范围。

这是因为 Web API中的批处理处理方式会从父请求复制属性到子请求。ASP.NET Web API 框架有意从父请求复制到子请求的一个属性就是请求生命周期范围。对此没有工作绕过方法，也不受 Autofac 控制。

## OWIN 集成

如果你的 Web API 是作为 OWIN 应用程序的一部分 使用的，你需要：

public class Startup  
 {  
 public void Configuration(IAppBuilder app)  
 {  
 var builder = new ContainerBuilder();  
   
   
 var config = new HttpConfiguration();  
   
 builder.RegisterApiControllers(Assembly.GetExecutingAssembly());  
   
 var container = builder.Build();  
 config.DependencyResolver = new AutofacWebApiDependencyResolver(container);  
   
   
 app.UseAutofacMiddleware(container);  
 app.UseAutofacWebApi(config);  
 app.UseWebApi(config);  
 }  
 }

在 OWIN 集成中常见的错误是使用GlobalConfiguration.Configuration。在 OWIN 中，你需要从头开始创建配置。使用 OWIN 集成时，不应在任何地方引用GlobalConfiguration.Configuration。

## 单元测试

当你正在单元测试使用 Autofac 的 ASP.NET Web API 应用程序，其中注册了InstancePerRequest 组件时，尝试解决这些组件时会遇到异常，因为在单元测试中没有 HTTP 请求生命周期。

请求生命周期范围 主题概述了测试和调试按请求范围组件的策略。

## 示例

在 [Autofac 示例仓库](https://www.koudingke.cn/go?link=https%3a%2f%2fgithub.com%2fautofac%2fExamples%2ftree%2fmaster%2fsrc%2fWebApiExample.OwinSelfHost)中，有一个示例项目展示了 Web API 与 OWIN 自托管的结合。

# Autofac SignalR 集成

SignalR 集成需要使用[Autofac.SignalR NuGet 包](https://www.koudingke.cn/go?link=https%3a%2f%2fnuget.org%2fpackages%2fAutofac.SignalR%2f)。

SignalR 集成提供了对 SignalR 中心的依赖注入集成。由于 SignalR 的内部实现，不支持按请求生命周期的依赖。

除了这篇针对 Autofac的文档外，你可能还对 [微软关于SignalR和依赖注入的文档](https://www.koudingke.cn/go?link=http%3a%2f%2fwww.asp.net%2fsignalr%2foverview%2fadvanced%2fdependency-injection)感兴趣。

## 快速入门

要将 Autofac 与 SignalR 集成，你需要引用 SignalR 集成 NuGet 包，注册你的中心，并设置依赖解析器。

protected void Application\_Start()  
 {  
 var builder = new ContainerBuilder();  
   
 builder.RegisterHubs(Assembly.GetExecutingAssembly());  
   
 var container = builder.Build();  
 GlobalHost.DependencyResolver = new AutofacDependencyResolver(container);  
 }

下面的各部分会详细介绍这些功能如何工作以及如何使用它们。

## 注册中心

在应用程序启动时，在构建 Autofac容器时，应注册你的 SignalR中心及其依赖项。这通常发生在 OWIN启动类中或Global.asax的Application\_Start方法中。

var builder = new ContainerBuilder();  
 builder.RegisterHubs(Assembly.GetExecutingAssembly());  
 builder.RegisterType<ChatHub>().ExternallyOwned();

如果手动注册单个中心，请确保它们已注册为ExternallyOwned() 。这样可以确保 SignalR 被允许控制中心的销毁，而不是 Autofac。

## 设置依赖解析器

在构建容器后，将其传递给AutofacDependencyResolver 类的新实例。将新的解析器附加到GlobalHost.DependencyResolver（如果你使用 OWIN，则附加到HubConfiguration.Resolver ）以便让 SignalR 知道它应该使用AutofacDependencyResolver 来查找服务。这是 Autofac 实现IDependencyResolver 接口的方式。

var container = builder.Build();  
 GlobalHost.DependencyResolver = new AutofacDependencyResolver(container);

## 管理依赖生命周期

由于不支持按请求的依赖，为SignalR 中心解析的所有依赖项都来自根容器。

* 如果你有IDisposable组件，它们将在应用程序的整个生命周期内存在，因为 Autofac会在 直到生命周期范围/容器被丢弃 时保持它们。你应该将这些注册为ExternallyOwned() 。
* 任何注册为InstancePerLifetimeScope() 的组件实际上都是单例。鉴于只有一个根生命周期范围，你只会得到一个实例。

为了使管理你的中心依赖生命周期更容易，可以在构造函数中将根生命周期范围注入到中心中。接下来，创建一个子生命周期范围，你可以在中心调用期间使用它来解析所需的项。最后，确保当中心由 SignalR销毁时，也销毁子生命周期。（这类似于服务定位，但这是唯一获得 “按中心”类型范围的方法。是的，它并不出色。）

public class MyHub : Hub  
 {  
 private readonly ILifetimeScope \_hubLifetimeScope;  
 private readonly ILogger \_logger;  
 public MyHub(ILifetimeScope lifetimeScope)  
 {  
   
 \_hubLifetimeScope = lifetimeScope.BeginLifetimeScope();  
   
 \_logger = \_hubLifetimeScope.Resolve<ILogger>();  
 }  
 public void Send(string message)  
 {  
   
 \_logger.Write("Received message: " + message);  
 Clients.All.addMessage(message);  
 }  
 protected override void Dispose(bool disposing)  
 {  
   
 if (disposing && \_hubLifetimeScope != null)  
 {  
 \_hubLifetimeScope.Dispose();  
 }  
 base.Dispose(disposing);  
 }  
 }

如果这是应用程序中的常见模式，你可能要考虑创建一个基类/抽象类，其他中心可以从该基类派生以节省所有复制粘贴创建/销毁范围的操作。

将生命周期范围注入到你的中心并不会给你按请求的生命周期范围。它只是让你以比从根容器中解析所有内容更主动的方式来管理依赖生命周期。即使使用这个解决方案，使用 InstancePerRequest 仍然会失败。有关更多信息，请参阅 关于按请求范围的 FAQ 。

## OWIN 集成

如果你正在使用 SignalR 作为 OWIN 应用程序的一部分 ，则需要：

public class Startup  
 {  
 public void Configuration(IAppBuilder app)  
 {  
 var builder = new ContainerBuilder();  
 var config = new HubConfiguration();  
   
 builder.RegisterHubs(Assembly.GetExecutingAssembly());  
   
 var container = builder.Build();  
 config.Resolver = new AutofacDependencyResolver(container);  
 app.UseAutofacMiddleware(container);  
 app.MapSignalR("/signalr", config);  
   
 var hubPipeline = config.Resolver.Resolve<IHubPipeline>();  
 hubPipeline.AddModule(new MyPipelineModule());  
 }  
 }

OWIN 集成中的常见错误是使用 GlobalHost 。在 OWIN 中，你从头开始创建配置。

使用 OWIN 集成时，不应在任何地方引用 GlobalHost 。[微软有关于此和其他 IoC 集成问题的文档。](https://www.koudingke.cn/go?link=http%3a%2f%2fwww.asp.net%2fsignalr%2foverview%2fadvanced%2fdependency-injection)

# Autofac Web Forms 集成

ASP.NET Web Forms 集成需要使用 [Autofac.Web NuGet 包](https://www.koudingke.cn/go?link=https%3a%2f%2fwww.nuget.org%2fpackages%2fAutofac.Web%2f)。

Web Forms 集成提供了代码背后类的依赖注入集成。它还添加了 基于请求的生命周期支持。

此页面解释了 ASP.NET 经典 Web Forms 集成。如果你正在使用 ASP.NET Core，请 查看 ASP.NET Core 集成页面。

## 快速入门

要将 Autofac 与 Web Forms 集成，你需要引用 Web Forms 集成的 NuGet 包，并在web.config 中添加模块，同时在Global 应用类上实现IContainerProviderAccessor 。

在web.config 中添加模块：

<configuration>  
 <system.web>  
 <httpModules>  
   
 <add  
 name="ContainerDisposal"  
 type="Autofac.Integration.Web.ContainerDisposalModule, Autofac.Integration.Web"/>  
 <add  
 name="PropertyInjection"  
 type="Autofac.Integration.Web.Forms.PropertyInjectionModule, Autofac.Integration.Web"/>  
 </httpModules>  
 </system.web>  
 <system.webServer>  
   
 <modules>  
 <add  
 name="ContainerDisposal"  
 type="Autofac.Integration.Web.ContainerDisposalModule, Autofac.Integration.Web"  
 preCondition="managedHandler"/>  
 <add  
 name="PropertyInjection"  
 type="Autofac.Integration.Web.Forms.PropertyInjectionModule, Autofac.Integration.Web"  
 preCondition="managedHandler"/>  
 </modules>  
 </system.webServer>  
 </configuration>

实现IContainerProviderAccessor ：

public class Global : HttpApplication, IContainerProviderAccessor  
 {  
   
 static IContainerProvider \_containerProvider;  
   
 public IContainerProvider ContainerProvider  
 {  
 get { return \_containerProvider; }  
 }  
 protected void Application\_Start(object sender, EventArgs e)  
 {  
   
 var builder = new ContainerBuilder();  
 builder.RegisterType<SomeDependency>();  
   
   
 \_containerProvider = new ContainerProvider(builder.Build());  
 }  
 }

以下各节详细介绍了这些功能的工作原理以及如何使用它们。

## 在 Web.config 中添加模块

Autofac 通过使用 [IHttpModule](https://www.koudingke.cn/go?link=https%3a%2f%2fmsdn.microsoft.com%2fen-us%2flibrary%2fsystem.web.ihttpmodule.aspx)实现来管理组件寿命并将其集成到 ASP.NET 管道中。你需要在web.config 中配置这些模块。

以下片段显示了已配置的模块：

<configuration>  
 <system.web>  
 <httpModules>  
   
 <add  
 name="ContainerDisposal"  
 type="Autofac.Integration.Web.ContainerDisposalModule, Autofac.Integration.Web"/>  
 <add  
 name="PropertyInjection"  
 type="Autofac.Integration.Web.Forms.PropertyInjectionModule, Autofac.Integration.Web"/>  
 </httpModules>  
 </system.web>  
 <system.webServer>  
   
 <modules>  
 <add  
 name="ContainerDisposal"  
 type="Autofac.Integration.Web.ContainerDisposalModule, Autofac.Integration.Web"  
 preCondition="managedHandler"/>  
 <add  
 name="PropertyInjection"  
 type="Autofac.Integration.Web.Forms.PropertyInjectionModule, Autofac.Integration.Web"  
 preCondition="managedHandler"/>  
 </modules>  
 </system.webServer>  
 </configuration>

注意，虽然有两部分分别用于 IIS6和 IIS7，但强烈建议你同时使用这两个部分。ASP.NET开发服务器即使目标部署环境是 IIS7也会使用 IIS6设置。如果你使用 IIS Express，它将使用 IIS7 设置。

你在那里看到的模块执行了一些有趣的操作：

* ContainerDisposalModule让 Autofac 在请求完成后立即处理在请求处理期间创建的任何组件。
* PropertyInjectionModule在页面生命周期执行之前将依赖项注入到页面中。还提供了替代的UnsetPropertyInjectionModule ，它只会将 Web 表单/控件的属性设置为 null 值。 (仅使用一个或另一个，但不能两者都用。)

## 在 Global.asax 中实现 IContainerProviderAccessor

依赖注入模块期望HttpApplication 实例支持IContainerProviderAccessor 。完整的全局应用程序类如下所示：

public class Global : HttpApplication, IContainerProviderAccessor  
 {  
   
 static IContainerProvider \_containerProvider;  
   
 public IContainerProvider ContainerProvider  
 {  
 get { return \_containerProvider; }  
 }  
 protected void Application\_Start(object sender, EventArgs e)  
 {  
   
 var builder = new ContainerBuilder();  
 builder.RegisterType<SomeDependency>();  
   
   
 \_containerProvider = new ContainerProvider(builder.Build());  
 }  
 }

Autofac.Integration.Web.IContainerProvider 提供了两个有用的属性：ApplicationContainer 和RequestLifetime 。

* ApplicationContainer 是启动时构建的应用程序容器。
* RequestLifetime是一个基于应用程序容器的组件 生命周期范围 ，将在当前 Web请求结束时被销毁。在需要手动依赖解析/服务查找时可以使用它。它包含的组件（除了单例）将是特定于当前请求的（这是 基于请求的生命周期依赖项 在这里解决的地方）。

## 技巧和窍门

### 结构化页面和用户控件以进行 DI

为了向 Web 表单页面(System.Web.UI.Page 实例)或用户控件(System.Web.UI.UserControl 实例)注入依赖项，你必须公开允许设置的依赖项作为公共属性。这使PropertyInjectionModule 可以为你填充这些属性。

确保在应用程序启动时注册所需的依赖项。

var builder = new ContainerBuilder();  
 builder.RegisterType<Component>().As<IService>().InstancePerRequest();  
 \_containerProvider = new ContainerProvider(builder.Build());

然后，在页面的代码背后，为所需的依赖项创建公共的 get/set 属性：

public partial class MyPage : Page  
 {  
   
 public IService MyService { get; set; }  
 protected void Page\_Load(object sender, EventArgs e)  
 {  
   
 label1.Text = this.MyService.GetMessage();  
 }  
 }

同样的公共属性注入过程也适用于用户控件，只需在应用程序启动时注册组件并为依赖项提供公共 get/set属性即可。

重要的是要注意对于用户控件，只有当控件在页面请求生命周期的预加载阶段由页面创建并添加到Controls 集合时，才会自动注入属性。动态创建的控件，无论是通过代码还是通过模板（如 Repeater），在这一点上都不会可见，必须手动注入其属性。

### 手动属性注入

在某些情况下，如在程序性创建用户控件或其他对象时，你可能需要手动在对象上设置属性。为此，你需要：

* 获取当前应用程序实例。
* 将其转换为Autofac.Integration.Web.IContainerProviderAccessor 。
* 从应用程序实例中获取容器提供程序。
* 从IContainerProvider 获取RequestLifetime ，并使用InjectProperties() 方法在对象上注入属性。

在代码中，这看起来像这样：

var cpa = (IContainerProviderAccessor)HttpContext.Current.ApplicationInstance;  
 var cp = cpa.ContainerProvider;  
 cp.RequestLifetime.InjectProperties(objectToSet);

请注意，你需要Autofac 和Autofac.Integration.Web 命名空间才能使属性注入工作，因为InjectProperties() 是Autofac 命名空间中的扩展方法。

### 通过属性明确注入

在向现有应用程序添加依赖注入时，有时希望区分哪些 Web 表单页面将注入依赖项，哪些不会。Autofac.Integration.Web 中的InjectPropertiesAttribute 和AttributedInjectionModule 有助于实现这一目标。

如果你选择使用 AttributedInjectionModule，除非它们标记有特殊属性，否则默认情况下不会自动注入public 属性的依赖项。

首先，从web.config 文件中删除PropertyInjectionModule ，然后用AttributedInjectionModule 替换它：

<configuration>  
 <system.web>  
 <httpModules>  
   
 <add  
 name="ContainerDisposal"  
 type="Autofac.Integration.Web.ContainerDisposalModule, Autofac.Integration.Web"/>  
 <add  
 name="AttributedInjection"  
 type="Autofac.Integration.Web.Forms.AttributedInjectionModule, Autofac.Integration.Web"/>  
 </httpModules>  
 </system.web>  
 <system.webServer>  
   
 <modules>  
 <add  
 name="ContainerDisposal"  
 type="Autofac.Integration.Web.ContainerDisposalModule, Autofac.Integration.Web"  
 preCondition="managedHandler"/>  
 <add  
 name="AttributedInjection"  
 type="Autofac.Integration.Web.Forms.AttributedInjectionModule, Autofac.Integration.Web"  
 preCondition="managedHandler"/>  
 </modules>  
 </system.webServer>  
 </configuration>

配置完成后，页面和控件将不会默认注入依赖项。相反，它们必须标记为Autofac.Integration.Web.Forms.InjectPropertiesAttribute 或Autofac.Integration.Web.Forms.InjectUnsetPropertiesAttribute。两者之间的区别：

* InjectPropertiesAttribute 将始终为页面/控件设置关联到 Autofac 注册的组件的公共属性。
* InjectUnsetPropertiesAttribute 只有当它们为 null 并且关联的组件已注册时，才会设置页面/控件的公共属性。

[InjectProperties]  
 public partial class MyPage : Page  
 {  
   
 public IService MyService { get; set; }  
   
 }

### 通过基页面类进行依赖注入

如果你不想通过模块（例如之前提到的AttributedInjectionModule 或PropertyInjectionModule）自动注入属性，可以以更手动的方式集成 Autofac，方法是创建一个在页面请求生命周期的PreInit 阶段执行手动属性注入的基页面类。

如果只有少数页面需要依赖注入，这样做可能比较合适，而不需要在管道中包含AttributedInjectionModule（但仍然需要ContainerDisposalModule）。如果你有超过少量的页面，考虑通过属性进行显式注入可能是有益的。

protected void Page\_PreInit(object sender, EventArgs e)  
 {  
 var cpa = (IContainerProviderAccessor)HttpContext.Current.ApplicationInstance;  
 var cp = cpa.ContainerProvider;  
 cp.RequestLifetime.InjectProperties(this);  
 }

### 定制依赖注入模块

如果提供的 *属性*、*未设置属性* 和 *带有注解* 的依赖注入模型不适合你的需求，创建自定义注入行为非常容易。只需继承Autofac.Integration.Web.DependencyInjectionModule，并在Web.config中使用结果即可。

要实现的一个抽象成员如下：

protected abstract IInjectionBehavior GetInjectionBehaviorForHandlerType(Type handlerType);

返回的IInjectionBehavior可以是预定义的NoInjection、PropertyInjection或UnsetPropertyInjection属性之一；也可以是IInjectionBehavior接口的自定义实现。

## 示例

有关 ASP.NET Web 表单集成的示例项目，请参阅 [Autofac 示例仓库](https://www.koudingke.cn/go?link=https%3a%2f%2fgithub.com%2fautofac%2fExamples%2ftree%2fmaster%2fsrc%2fWebFormsExample)。

# Autofac .NET Core 集成

.NET Core 配备了一个符合规范的容器，即 [Microsoft.Extensions.DependencyInjection](https://www.koudingke.cn/go?link=https%3a%2f%2fgithub.com%2faspnet%2fDependencyInjection)。Autofac.Extensions.DependencyInjection 包实现了这些抽象，以通过 Autofac 提供依赖注入。

与 ASP.NET Core 的集成非常相似，因为整个框架已经统一了围绕依赖注入的抽象。我们的 ASP.NET Core 集成文档中有关于 ASP.NET Core（以及托管应用程序）使用特定主题的更多信息。

## 快速入门

要通过 Microsoft.Extensions.DependencyInjection 包在 .NET Core 应用程序中利用 Autofac：

* 从 NuGet 引用 Autofac.Extensions.DependencyInjection 包。
* 在应用程序启动时（例如，在 Program 或 Startup 类中）...

o 使用框架扩展方法在 IServiceCollection 中注册服务。

o 将注册的服务填充到 Autofac 中。

o 添加 Autofac 注册和覆盖。

o 构建容器。

o 使用容器创建 AutofacServiceProvider。

public class Program  
 {  
 public static void Main(string[] args)  
 {  
 var serviceCollection = new ServiceCollection();  
   
 serviceCollection.AddLogging();  
 var containerBuilder = new ContainerBuilder();  
   
 containerBuilder.Populate(serviceCollection);   
   
 containerBuilder.RegisterType<MessageHandler>().As<IHandler>();  
   
 var container = containerBuilder.Build();  
 var serviceProvider = new AutofacServiceProvider(container);  
 }  
 }

**你不必使用 Microsoft.Extensions.DependencyInjection。** 如果你不是编写需要它的 .NET Core 应用程序，或者你不使用其他库提供的任何依赖注入扩展，你可以直接使用 Autofac。你可能只需要调用 Populate() 方法，而不需要 AutofacServiceProvider。根据你的应用需求选择合适的部分。

## 作为根使用的子范围

Using a Child Scope as a Root

在复杂的应用中，你可能希望将使用 Populate()注册的服务保留在子生存周期范围中。例如，一个自托管 ASP.NET Core 组件的应用可能希望将 MVC 注册项等隔离到主容器之外。Populate()方法提供重载版本，允许你指定一个标记的子生存周期范围，该范围应作为包含项目的 “容器”。

**注意**  
 *如果你使用此功能，将无法使用 ASP.NET Core 支持的* *IServiceProviderFactory{TContainerBuilder}****（****ConfigureContainer* *支持）。这是因为* *IServiceProviderFactory{TContainerBuilder}* *假设它在根级别工作。*

public class Program  
{  
 private const string RootLifetimeTag = "MyIsolatedRoot";  
 public static void Main(string[] args)  
 {  
 var serviceCollection = new ServiceCollection();  
 serviceCollection.AddLogging();  
 var containerBuilder = new ContainerBuilder();  
 containerBuilder.RegisterType<MessageHandler>().As<IHandler>();  
 var container = containerBuilder.Build();  
 using (var scope = container.BeginLifetimeScope(RootLifetimeTag, b =>  
 {  
 b.Populate(serviceCollection, RootLifetimeTag);  
 }))  
 {  
 var serviceProvider = new AutofacServiceProvider(scope);  
 }  
 }  
}

# Autofac ASP.NET Core 集成

ASP.NET Core（以前称为 ASP.NET 5）改变了以前的依赖注入框架如何集成到 ASP.NET 执行中。以前，每个功能（如 MVC、Web API 等）都有自己的 “依赖解析器” 机制，只是稍微不同的集成方式。ASP.NET Core 引入了[符合规范的容器](https://www.koudingke.cn/go?link=http%3a%2f%2fblog.ploeh.dk%2f2014%2f05%2f19%2fconforming-container%2f)机制，通过 Microsoft.Extensions.DependencyInjection ，包括统一的请求生命周期范围、服务注册等功能。

此外，从 ASP.NET Core 3.0 开始，有一个通用的应用托管机制，可用于非 ASP.NET Core 应用。

**此页面解释了 ASP.NET Core 和通用.NET Core 托管的集成。如果你正在使用** **ASP.NET** **经典，请参阅** **ASP.NET** **经典集成页面** **。**

如果你使用的是 .NET Core 但不使用 ASP.NET Core（或不使用通用托管），则可以查看 此处有一个更简单的示例 ，展示这种集成。

## 快速入门

* 从 NuGet 引用 Autofac.Extensions.DependencyInjection 包。
* 在 Program.Main 方法中，将托管机制附加到 Autofac。（参见下面的示例。）
* 在 Startup 类的 ConfigureServices 方法中，使用其他库提供的扩展方法向 IServiceCollection 注册内容。
* 在 Startup 类的 ConfigureContainer 方法中，直接向 Autofac 的 ContainerBuilder 注册内容。

IServiceProvider 会自动为你创建，你只需进行**注册**即可。

### ASP.NET Core 1.1 - 2.2

这个示例展示了**ASP.NET Core 1.1 - 2.2**的用法，你在 WebHostBuilder 上调用 services.AddAutofac() 。**这不是用于 ASP.NET Core 3+**或.NET Core 3+的通用托管支持——ASP.NET Core 3 要求你直接指定服务提供者工厂，而不是将其添加到服务集合中。

public class Program  
{  
 public static void Main(string[] args)  
 {  
 var host = new WebHostBuilder()  
 .UseKestrel()  
 .ConfigureServices(services => services.AddAutofac())  
 .UseContentRoot(Directory.GetCurrentDirectory())  
 .UseIISIntegration()  
 .UseStartup<Startup>()  
 .Build();  
 host.Run();  
 }  
}

这个示例展示了**ASP.NET Core 1.1 - 2.2**的用法，你从 ConfigureServices(IServiceCollection services) 委托中返回一个 IServiceProvider 。**这不是用于 ASP.NET Core 3+** 或.NET Core 3+的通用托管支持——ASP.NET Core 3 已弃用从 ConfigureServices 返回服务提供者的功能。

public class Startup  
 {  
 public Startup(IHostingEnvironment env)  
 {  
   
 var builder = new ConfigurationBuilder()  
 .SetBasePath(env.ContentRootPath)  
 .AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)  
 .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true)  
 .AddEnvironmentVariables();  
 this.Configuration = builder.Build();  
 }  
 public IConfigurationRoot Configuration { get; private set; }  
 public ILifetimeScope AutofacContainer { get; private set; }  
   
   
 public IServiceProvider ConfigureServices(IServiceCollection services)  
 {  
   
 services.AddOptions();  
   
 var builder = new ContainerBuilder();  
   
 builder.Populate(services);  
   
 builder.RegisterModule(new MyApplicationModule());  
 AutofacContainer = builder.Build();  
   
 return new AutofacServiceProvider(AutofacContainer);  
 }  
   
 public void Configure(  
 IApplicationBuilder app,  
 ILoggerFactory loggerFactory)  
 {  
 loggerFactory.AddConsole(this.Configuration.GetSection("Logging"));  
 loggerFactory.AddDebug();  
 app.UseMvc();  
 }  
 }

### ASP.NET Core 3.0+和通用托管

**在 ASP.NET Core 3.0 中，托管方式发生了变化，需要不同的集成。**你不能再从 ConfigureServices 返回 IServiceProvider ，也不能将服务提供者工厂添加到服务集合中。

这是针对 ASP.NET Core 3+和.NET Core 3+的通用托管支持：

public class Program  
 {  
 public static void Main(string[] args)  
 {  
 var host = Host.CreateDefaultBuilder(args)  
 .UseServiceProviderFactory(new AutofacServiceProviderFactory())  
 .ConfigureWebHostDefaults(webHostBuilder => {  
 webHostBuilder  
 .UseContentRoot(Directory.GetCurrentDirectory())  
 .UseIISIntegration()  
 .UseStartup<Startup>();  
 })  
 .Build();  
 host.Run();  
 }  
 }

### Startup 类

在你的 Startup 类（适用于所有 ASP.NET Core 版本）中，然后使用 ConfigureContainer 访问 Autofac 的 ContainerBuilder 并直接向 Autofac 注册内容。

public class Startup  
 {  
 public Startup(IConfiguration configuration)  
 {  
   
 this.Configuration = configuration;  
 }  
 public IConfiguration Configuration { get; private set; }  
 public ILifetimeScope AutofacContainer { get; private set; }  
   
 public void ConfigureServices(IServiceCollection services)  
 {  
   
 services.AddOptions();  
 }  
   
   
 public void ConfigureContainer(ContainerBuilder builder)  
 {  
   
 builder.RegisterModule(new MyApplicationModule());  
 }  
   
 public void Configure(  
 IApplicationBuilder app,  
 ILoggerFactory loggerFactory)  
 {  
   
 this.AutofacContainer = app.ApplicationServices.GetAutofacRoot();  
 loggerFactory.AddConsole(this.Configuration.GetSection("Logging"));  
 loggerFactory.AddDebug();  
 app.UseMvc();  
 }  
 }

## 配置方法命名约定

Configure、ConfigureServices 和 ConfigureContainer 方法都支持基于 IHostingEnvironment.EnvironmentName 的环境特定命名约定。默认情况下，名称分别为 Configure 、 ConfigureServices 和 ConfigureContainer 。如果你想要环境特定的设置，可以在 Configure 部分后面加上环境名，如 ConfigureDevelopment 、 ConfigureDevelopmentServices 和 ConfigureDevelopmentContainer 。如果没有匹配环境的方法，则使用默认方法。

这意味着你不一定需要使用 Autofac 配置 来在开发和生产环境中切换配置；你可以通过 Startup中的程序性设置来实现。

public class Startup  
 {  
 public Startup(IHostingEnvironment env)  
 {  
   
 }  
   
 public void ConfigureServices(IServiceCollection services)  
 {  
   
 }  
   
 public void ConfigureDevelopmentServices(IServiceCollection services)  
 {  
   
 }  
   
 public void ConfigureContainer(ContainerBuilder builder)  
 {  
   
 }  
   
 public void ConfigureProductionContainer(ContainerBuilder builder)  
 {  
   
 }  
   
 public void Configure(IApplicationBuilder app, ILoggerFactory loggerFactory)  
 {  
   
 }  
   
 public void ConfigureStaging(IApplicationBuilder app, ILoggerFactory loggerFactory)  
 {  
   
 }  
 }

这是 ASP.NET Core 应用程序主持的一项功能 - 它不是 Autofac 的行为。在 ASP.NET Core 中，启动过程是由 [StartupLoader 类](https://www.koudingke.cn/go?link=https%3a%2f%2fgithub.com%2faspnet%2fHosting%2fblob%2frel%2f1.1.0%2fsrc%2fMicrosoft.AspNetCore.Hosting%2fInternal%2fStartupLoader.cs) 负责定位需要调用的方法的。如果你想更深入地了解这是如何工作的，可以查看那个类。

## 依赖注入钩子

**Dependency Injection Hooks**

与 ASP.NET 经典集成 不同，ASP.NET Core 是专门为依赖注入设计的。这意味着如果你试图了解如何将服务注入到 MVC 视图（ [这里](https://www.koudingke.cn/go?link=https%3a%2f%2fdocs.asp.net%2fen%2flatest%2fmvc%2fviews%2fdependency-injection.html)有更多说明），现在这些都由 ASP.NET Core 控制并文档化了，你只需要按照上面的说明设置你的服务提供者，而无需做任何特定于 Autofac 的操作。

以下是一些有关 ASP.NET Core 集成的链接，它们提供了关于 DI 整合的特定见解：

## 从 ASP.NET 经典迁移时的主要差异

如果你已经使用过其他 ASP.NET 集成 ，你可能会对迁移 ASP.NET Core 时的关键差异感兴趣。

* **使用 InstancePerLifetimeScope 替代 InstancePerRequest。** 在以前的 ASP.NET 集成中，你可以将依赖项注册为 InstancePerRequest，这将确保每个 HTTP 请求只创建一个依赖项实例。这之所以可行，是因为 Autofac 负责设置 “每个请求的生命周期作用域” 。随着 Microsoft.Extensions.DependencyInjection 的引入，创建每个请求和其他子生命周期作用域现在是框架提供的符合规范的容器的一部分，因此所有子生命周期作用域都被平等对待 - 没有特殊的“请求级别作用域”。相反，你应该将依赖项注册为 InstancePerLifetimeScope，而不是 InstancePerRequest，以获得相同的行为。请注意，如果你在 Web 请求期间创建自己的生命周期作用域，那么在这些子作用域中会得到一个新的实例。
* **不再有 DependencyResolver。** 其他 ASP.NET 集成机制需要在多个位置设置基于 Autofac 的自定义依赖注入解析器。现在通过 Microsoft.Extensions.DependencyInjection 和 Startup.ConfigureServices 方法，只需返回 IServiceProvider，即可实现“魔法”。在控制器、类等中，如果需要手动进行服务定位，请获取 IServiceProvider。
* **没有特殊的中间件。** 在以前的 OWIN 集成 中，需要注册一个特殊的 Autofac 中间件来管理请求生命周期。现在 Microsoft.Extensions.DependencyInjection 承担了繁重的工作，所以无需额外的中间件进行注册。
* **不再需要手动注册控制器。**以前，为了使 DI 工作，你需要将所有控制器注册到 Autofac 中。现在 ASP.NET Core 框架会自动将所有控制器通过服务解析，所以你不再需要这样做。
* **不再需要通过依赖注入扩展来调用中间件。**在 OWIN 集成 中，有诸如 UseAutofacMiddleware() 的扩展，允许将 DI 引入到中间件。现在，通过组合自动注入构造参数和动态解析到中间件的 Invoke 方法的参数，这一切都自动完成。ASP.NET Core 框架处理所有这些。
* **MVC 和 Web API 是一回事。**以前，根据你是使用 MVC 还是 Web API，会有不同的方式来接入 DI。在 ASP.NET Core 中，这两个东西合并在一起，所以只需要设置一个依赖注入解析器，维护一个配置文件即可。
* **控制器不会从容器中解析；只有控制器构造函数参数会被解析。**这意味着控制器生命周期、属性注入和其他事情都不会由 Autofac 管理，而是由 ASP.NET Core 管理。你可以使用 AddControllersAsServices() 来改变这一点，下面会详细讨论。

## 控制器作为服务

默认情况下，ASP.NET Core 会从容器解析控制器的 *参数*，但并不会实际从容器中解析 *控制器本身*。这通常不是问题，但它意味着：

* *控制器* 的生命周期由框架而非请求生命周期管理。
* *控制器构造函数参数* 的生命周期由请求生命周期管理。
* 你可能在注册控制器时做的特殊绑定（如设置属性注入）将不起作用。

要更改这一点，当你使用服务集合注册 MVC 时，可以指定 AddControllersAsServices()。这样当服务提供者工厂调用 builder.Populate(services) 时，将自动将控制器类型注册到 IServiceCollection 中。

public class Startup  
 {  
   
 public void ConfigureServices(IServiceCollection services)  
 {  
   
 services.AddMvc().AddControllersAsServices();  
 }  
 public void ConfigureContainer(ContainerBuilder builder)  
 {  
   
 builder.RegisterType<MyController>().PropertiesAutowired();  
 }  
 }

有关更详细的教程，包括步行游览，请参阅 [Filip Woj博客上的文章](https://www.koudingke.cn/go?link=http%3a%2f%2fwww.strathweb.com%2f2016%2f03%2fthe-subtle-perils-of-controller-dependency-injection-in-asp-net-core-mvc%2f)。那里的一位评论者还发现了一些关于 RC2 如何处理控制器作为服务的更改 [见此处](https://www.koudingke.cn/go?link=http%3a%2f%2fwww.strathweb.com%2f2016%2f03%2fthe-subtle-perils-of-controller-dependency-injection-in-asp-net-core-mvc%2f%23comment-2702995712)。

## 多租户支持

由于 ASP.NET Core 对生成请求生命周期作用域的急切性，它使得多租户支持无法直接工作。有时用于租户识别的常见 IHttpContextAccessor 也可能来不及设置。为此，添加了名为 Autofac.AspNetCore.Multitenant 的包来修复这个问题。

启用多租户支持：

* 添加对 Autofac.AspNetCore.Multitenant 包的引用。
* 在 Program.Main 中创建 Web 主机时，包含一个调用 UseServiceProviderFactory 扩展，并使用 AutofacMultitenantServiceProviderFactory。提供一个回调，用于配置你的租户。
* 在 Startup.ConfigureServices 和 Startup.ConfigureContainer 中，注册那些不会特定于租户的东西，这些属于 **根容器**。
* 在回调（例如 Startup.ConfigureMultitenantContainer）中构建多租户容器。

以下是 Program.Main 中的部分示例代码：

public class Program  
 {  
 public static async Task Main(string[] args)  
 {  
 var host = Host  
 .CreateDefaultBuilder(args)  
 .UseServiceProviderFactory(new AutofacMultitenantServiceProviderFactory(Startup.ConfigureMultitenantContainer))  
 .ConfigureWebHostDefaults(webHostBuilder => webHostBuilder.UseStartup<Startup>())  
 .Build();  
 await host.RunAsync();  
 }  
 }

Startup 类的结构如下：

public class Startup  
 {  
   
 public void ConfigureServices(IServiceCollection services)  
 {  
   
 services.AddMvc();  
   
 services.AddAutofacMultitenantRequestServices();  
 }  
 public void ConfigureContainer(ContainerBuilder builder)  
 {  
   
 builder.RegisterType<Dependency>().As<IDependency>();  
 }  
 public static MultitenantContainer ConfigureMultitenantContainer(IContainer container)  
 {  
   
 var strategy = new MyTenantIdentificationStrategy();  
 var mtc = new MultitenantContainer(strategy, container);  
 mtc.ConfigureTenant("a", cb => cb.RegisterType<TenantDependency>().As<IDependency>());  
 return mtc;  
 }  
 }

## 使用子作用域作为根

Using a Child Scope as a Root

在复杂应用中，你可能希望将服务划分开来，让根容器在整个应用程序的不同部分之间共享，但使用托管部分（如 ASP.NET Core）的子生存期上下文。例如：

标准 ASP.NET Core 集成和通用托管应用程序支持中，你可以使用 AutofacChildLifetimeScopeServiceProviderFactory替换标准 AutofacServiceProviderFactory。这允许你在特定命名生存期上下文中附加配置操作，而不是构建好的容器。

public class Program  
{  
 public static async Task Main(string[] args)  
 {   
 var containerBuilder = new ContainerBuilder();  
 builder.RegisterType<SomeGlobalDependency>()  
 .As<ISomeGlobalDependency>()  
 .InstancePerLifetimeScope();  
 var container = containerBuilder.Build();  
   
 var hostOne = Host  
 .CreateDefaultBuilder(args)  
 .UseServiceProviderFactory(new AutofacChildLifetimeScopeServiceProviderFactory(container.BeginLifetimeScope("root-one")))  
 .ConfigureWebHostDefaults(webHostBuilder =>  
 {  
 webHostBuilder  
 .UseContentRoot(AppContext.BaseDirectory)   
 .UseUrls("http://localhost:5000")  
 .UseStartup<StartupOne>();  
 })  
 .Build();  
   
 var hostTwo = Host  
 .CreateDefaultBuilder(args)  
 .UseServiceProviderFactory(new AutofacChildLifetimeScopeServiceProviderFactory(container.BeginLifetimeScope("root-two")))  
 .ConfigureWebHostDefaults(webHostBuilder =>  
 {  
 webHostBuilder  
 .UseContentRoot(AppContext.BaseDirectory)   
 .UseUrls("http://localhost:5001")  
 .UseStartup<StartupTwo>();  
 })  
 .Build();  
 await Task.WhenAll(hostOne.RunAsync(), hostTwo.RunAsync())  
 }  
}

这将改变你的 Startup 类的工作方式 - 你不再直接在 ConfigureContainer 中使用 ContainerBuilder，现在它是 AutofacChildLifetimeScopeConfigurationAdapter：

public class StartupOne  
 {  
   
 public Startup(IWebHostEnvironment env)  
 {  
   
 }  
 public void ConfigureServices(IServiceCollection services)  
 {  
   
 }  
   
 public void ConfigureContainer(AutofacChildLifetimeScopeConfigurationAdapter config)  
 {  
 config.Add(builder => builder.RegisterModule(new AutofacHostOneModule()));  
 }  
 public void Configure(  
 IApplicationBuilder app,  
 ILoggerFactory loggerFactory)  
 {  
   
 }  
 }  
 public class StartupTwo  
 {  
   
 public Startup(IWebHostEnvironment env)  
 {  
   
 }  
 public void ConfigureServices(IServiceCollection services)  
 {  
   
 }  
   
 public void ConfigureContainer(AutofacChildLifetimeScopeConfigurationAdapter config)  
 {  
 config.Add(builder => builder.RegisterModule(new AutofacHostTwoModule()));  
 }  
 public void Configure(  
 IApplicationBuilder app,  
 ILoggerFactory loggerFactory)  
 {  
   
 }  
 }

如果你不使用服务提供程序工厂，Populate()方法提供了一个重载，允许你指定应作为"容器"使用的带有标签的子生存期上下文。

.NET Core集成文档还展示了使用子生存期上下文作为根的示例 。

使用子生存期上下文作为根与多租户支持不兼容。你必须选择一个，不能两者兼得。

## 示例

在 [Autofac 示例存储库](https://www.koudingke.cn/go?link=https%3a%2f%2fgithub.com%2fautofac%2fExamples%2ftree%2fmaster%2fsrc%2fAspNetCoreExample) 中有一个演示 ASP.NET Core 集成的 [示例项目](https://github.com/autofac/Examples/tree/master/src/AspNetCoreExample)。

# Autofac Azure Functions 集成

Azure Functions 支持使用 Microsoft 的依赖注入框架进行依赖注入，但你可以通过添加一些引导代码使其与 Autofac 一起工作。

我们建议阅读官方 Microsoft 文档 [《Azure Functions 中的依赖注入概述》](https://www.koudingke.cn/go?link=https%3a%2f%2fdocs.microsoft.com%2fen-us%2fazure%2fazure-functions%2ffunctions-dotnet-dependency-injection) ，以了解 Azure Functions 中依赖注入的基本概念。

## 步骤概览

1. 从 NuGet 安装 Autofac、Autofac.Extensions.DependencyInjection 和 Microsoft.Azure.Functions.Extensions 。
2. 添加一个基于 Autofac 的工作器激活器，用于创建你的函数类实例。
3. 创建一个 Startup类，在其中注册你的组件，并替换默认的工作器激活器。

## Autofac 工作器激活器

工作器激活器负责实例化包含你的函数的类。在项目中添加以下代码——它是一个基于 Autofac 生命周期范围的工作器激活器，用于从生命周期范围中获取适当的类。接下来我们将实现 LifetimeScopeWrapper和 LoggerModule。

internal class AutofacJobActivator : IJobActivatorEx  
 {  
 public T CreateInstance<T>()  
 {  
   
 throw new NotSupportedException();  
 }  
 public T CreateInstance<T>(IFunctionInstanceEx functionInstance)  
 where T : notnull  
 {  
 var lifetimeScope = functionInstance.InstanceServices  
 .GetRequiredService<LifetimeScopeWrapper>()  
 .Scope;  
   
 var loggerFactory = functionInstance.InstanceServices.GetRequiredService<ILoggerFactory>();  
 lifetimeScope.Resolve<ILoggerFactory>(  
 new NamedParameter(LoggerModule.LoggerFactoryParam, loggerFactory)  
 );  
 lifetimeScope.Resolve<ILogger>(  
 new NamedParameter(LoggerModule.FunctionNameParam, functionInstance.FunctionDescriptor.LogName)  
 );  
 return lifetimeScope.Resolve<T>();  
 }  
 }

接下来，实现 LifetimeScopeWrapper 。这个类从 IServiceCollection 中解析，允许我们在函数执行完毕后释放 Autofac 生命周期范围。

internal sealed class LifetimeScopeWrapper : IDisposable  
 {  
 public ILifetimeScope Scope { get; }  
 public LifetimeScopeWrapper(IContainer container)  
 {  
 Scope = container.BeginLifetimeScope();  
 }  
 public void Dispose()  
 {  
 Scope.Dispose();  
 }  
 }

为了能够解析 ILogger，我们需要特殊处理，因为某些日志工厂直到 Startup类运行后才会初始化。我们可以通过添加以下代码来解决这个问题。

internal class LoggerModule : Module  
 {  
 public const string LoggerFactoryParam = "loggerFactory";  
 public const string FunctionNameParam = "functionName";  
 protected override void Load(ContainerBuilder builder)  
 {  
 builder.Register((ctx, p) => p.Named<ILoggerFactory>(LoggerFactoryParam))  
 .SingleInstance();  
 builder.Register((ctx, p) =>  
 {  
 var factory = ctx.Resolve<ILoggerFactory>();  
 var functionName = p.Named<string>(FunctionNameParam);  
 return factory.CreateLogger(Microsoft.Azure.WebJobs.Logging.LogCategories.CreateFunctionUserCategory(functionName));  
 })  
 .InstancePerLifetimeScope();  
 }  
 }

即使你没有直接使用 ILogger ，也应该将 LoggerModule 包含在项目中，因为 Microsoft 的许多 NuGet 包都引用了此接口。

**Startup** **类**

最后，添加一个 Startup 类将所有内容连接起来。这个类在概念上类似于 ASP.NET Core 项目的 Startup 类。

FunctionsStartup 基类由 Microsoft.Azure.Functions.Extensions NuGet 包提供。

[assembly: FunctionsStartup(typeof(MyFunctionApp.Startup))]  
 namespace MyFunctionApp;  
 internal class Startup : FunctionsStartup  
 {  
 public override void Configure(IFunctionsHostBuilder builder)  
 {  
   
 builder.Services.AddDataProtection();  
 builder.Services.AddSingleton(GetContainer(builder.Services));  
   
 builder.Services.AddScoped<LifetimeScopeWrapper>();  
 builder.Services.Replace(ServiceDescriptor.Singleton(typeof(IJobActivator), typeof(AutofacJobActivator)));  
 builder.Services.Replace(ServiceDescriptor.Singleton(typeof(IJobActivatorEx), typeof(AutofacJobActivator)));  
 }  
 private static IContainer GetContainer(IServiceCollection serviceCollection)  
 {  
 var containerBuilder = new ContainerBuilder();  
 containerBuilder.Populate(serviceCollection);  
 containerBuilder.RegisterModule<LoggerModule>();  
   
 containerBuilder.RegisterAssemblyTypes(typeof(Startup).Assembly)  
 .InNamespaceOf<Function1>();  
   
 return containerBuilder.Build();  
 }  
 }

就这样！现在你的函数类将从 Autofac 中解析。

## 示例函数

这是一个使用依赖注入服务的 HTTP 触发函数示例。请注意，类和 Run方法不是静态的。

public class Function1  
 {  
 private readonly IRandomNumberService \_randomNumberService;  
 public Function1(IRandomNumberService randomNumberService)  
 {  
 \_randomNumberService = randomNumberService;  
 }  
   
 [FunctionName("Function1")]  
 public IActionResult Run(  
 [HttpTrigger(AuthorizationLevel.Function, "get", Route = null)]  
 HttpRequest request  
 )  
 {  
 var number = \_randomNumberService.GetDouble();  
 return new OkObjectResult($"Your random number is {number}.");  
 }  
 }

## 致谢

本指南受到了社区 NuGet 包 [Autofac.Extensions.DependencyInjection.AzureFunctions](https://www.koudingke.cn/go?link=https%3a%2f%2fgithub.com%2fjunalmeida%2fautofac-azurefunctions) 的启发。如果你更喜欢使用 NuGet 包而不是这里介绍的自定义方法，可以尝试一下 Autofac.Extensions.DependencyInjection.AzureFunctions 。

# Autofac Blazor 集成

[ASP.NET Core Blazor](https://www.koudingke.cn/go?link=https%3a%2f%2fdocs.microsoft.com%2fen-gb%2faspnet%2fcore%2fblazor%2f) 使用 ASP.NET Core 3+ 中的通用应用托管，但两种 [托管模型](https://www.koudingke.cn/go?link=https%3a%2f%2fdocs.microsoft.com%2fen-gb%2faspnet%2fcore%2fblazor%2fhosting-models) 的集成略有不同。

**服务器端** 实现与任何其他 ASP.NET Core 3 应用程序的配置方式完全相同。

**客户端** 注入稍微受到限制，因为需要 [WebAssembly](https://www.koudingke.cn/go?link=https%3a%2f%2fwebassembly.org) 托管的要求。

此 WebAssembly 示例适用于 2021 年 3 月 30 日的 .NET 5。示例：

public class Program  
 {  
 public static async Task Main(string[] args)  
 {  
 var builder = WebAssemblyHostBuilder.CreateDefault(args);  
 builder.ConfigureContainer(new AutofacServiceProviderFactory(ConfigureContainer));  
 builder.RootComponents.Add<App>("#app");  
 await builder.Build().RunAsync();  
 }  
 private static void ConfigureContainer(ContainerBuilder builder)  
 {  
   
 }  
 }

一旦注册，Blazor 组件可以通过 [依赖注入](https://www.koudingke.cn/go?link=https%3a%2f%2fdocs.microsoft.com%2fen-gb%2faspnet%2fcore%2fblazor%2fdependency-injection)使用 [标准@inject Razor指令](https://www.koudingke.cn/go?link=https%3a%2f%2fdocs.microsoft.com%2fen-us%2faspnet%2fcore%2fblazor%2fdependency-injection%3fview%3daspnetcore-3.0%23request-a-service-in-a-component)。

# Windows Communication Foundation (WCF)

WCF 集成对于客户端和服务端都依赖于 [Autofac.Wcf NuGet 包](https://www.koudingke.cn/go?link=https%3a%2f%2fwww.nuget.org%2fpackages%2fAutofac.Wcf%2f) 。

**WCF 集成提供了服务端和服务代理的依赖注入整合。由于 WCF 的内部实现，对于每个请求的生命周期依赖没有明确的支持。**

## 客户端

在你的服务客户端应用中使用 Autofac，有以下几点好处：

* **确定性释放资源**：自动释放由 ChannelFactory.CreateChannel() 创建的代理消耗的资源。
* **方便的服务代理注入**：对于使用服务的类型，你可以轻松地为服务接口类型注入依赖项。

在应用程序启动时，针对每个服务注册一个 ChannelFactory 和一个使用工厂打开通道的函数：

var builder = new ContainerBuilder();  
 builder  
 .Register(c => new ChannelFactory<ITrackListing>(  
 new BasicHttpBinding(),  
 new EndpointAddress("http://localhost/TrackListingService")))  
 .SingleInstance();  
 builder  
 .Register(c => c.Resolve<ChannelFactory<ITrackListing>>().CreateChannel())  
 .As<ITrackListing>()  
 .UseWcfSafeRelease();  
 builder.RegisterType<AlbumPrinter>();  
 var container = builder.Build();

在这个例子中...

* 直到从容器中请求 ITrackListing 时，才执行 CreateChannel() 调用。
* UseWcfSafeRelease() 配置选项确保在释放客户端通道时不会丢失异常信息。

当消费服务时，像通常那样添加构造函数依赖项。例如，这个示例展示了一个通过远程 ITrackListing服务将专辑列表打印到控制台的应用。它通过 AlbumPrinter类实现：

public class AlbumPrinter  
 {  
 readonly ITrackListing \_trackListing;  
 public AlbumPrinter(ITrackListing trackListing)  
 {  
 \_trackListing = trackListing;  
 }  
 public void PrintTracks(string artist, string album)  
 {  
 foreach (var track in \_trackListing.GetTracks(artist, album))  
 Console.WriteLine("{0} - {1}", track.Position, track.Title);  
 }  
 }

当你从生命周期作用域中解析 AlbumPrinter类时，服务接口的通道会自动注入。

注意，鉴于 服务代理是dispose(可释放)的 ，你应该从子作用域而不是根容器中解析它。因此，如果必须手动解析（无论出于何种原因），确保你是从其中创建子作用域的地方进行操作：

using(var lifetime = container.BeginLifetimeScope())  
 {  
 var albumPrinter = lifetime.Resolve<AlbumPrinter>();  
 albumPrinter.PrintTracks("The Shins", "Wincing the Night Away");  
 }

## 服务端

### 快速入门

要让 Autofac 与 WCF 在服务端集成，你需要引用 WCF 集成的 NuGet 包，注册你的服务，并设置依赖解析器。你还需要更新 .svc文件，引用 Autofac 服务主机工厂。

这里有一个示例应用启动块：

protected void Application\_Start()  
 {  
 var builder = new ContainerBuilder();  
   
 builder.RegisterType<TestService.Service1>();  
   
 var container = builder.Build();  
 AutofacHostFactory.Container = container;  
 }

而 .svc文件看起来像这样：

<%@ ServiceHost  
 Service="TestService.Service1, TestService"  
 Factory="Autofac.Integration.Wcf.AutofacServiceHostFactory, Autofac.Integration.Wcf" %>

下面的各节将详细说明这些功能以及如何使用它们。

### 注册服务实现

你可以通过类型、接口或名称三种方式之一在容器中注册服务类型。

#### 通过类型注册

最常见的方式是直接在容器中注册服务实现类型，并在 .svc文件中指定该实现类型。

在应用程序启动时，代码可能如下所示：

var builder = new ContainerBuilder();  
 builder.RegisterType<TestService.Service1>();  
 AutofacHostFactory.Container = builder.Build();

.svc文件会指定适当的服务实现类型和主机工厂，如下所示：

<%@ ServiceHost  
 Service="TestService.Service1, TestService"  
 Factory="Autofac.Integration.Wcf.AutofacServiceHostFactory, Autofac.Integration.Wcf" %>

注意在 .svc 文件中需要使用服务的完全限定名，即 Service="Namespace.ServiceType, AssemblyName" 。

#### 通过接口注册

第二种选择是将合同类型注册到容器中，并在 .svc 文件中指定合同。如果你不想改变 .svc 文件，但想更改处理请求的实现类型，这很有用。

应用程序启动时的代码可能是这样的：

var builder = new ContainerBuilder();  
 builder.RegisterType<TestService.Service1>()  
 .As<TestService.IService1>();  
 AutofacHostFactory.Container = builder.Build();

.svc文件会指定服务合同类型和主机工厂，如下所示：

<%@ ServiceHost  
 Service="TestService.IService1, TestService"  
 Factory="Autofac.Integration.Wcf.AutofacServiceHostFactory, Autofac.Integration.Wcf" %>

同样，在 .svc 文件中需要使用合同的完全限定名，即 Service="Namespace.IContractType, AssemblyName" 。

#### 通过名称注册

第三种选择是在容器中注册一个命名的服务实现，并在 .svc 文件中指定该服务名称。如果你希望进一步抽象出 .svc 文件，这非常有用。

在应用程序启动时，代码可能如下所示：

var builder = new ContainerBuilder();  
 builder.RegisterType<TestService.Service1>()  
 .Named<object>("my-service");  
 AutofacHostFactory.Container = builder.Build();

请注意，服务实现类型作为对象注册——这一点很重要。如果你注册了一个命名的服务，但没有将其注册为对象，那么将找不到它。

.svc文件会指定你注册的服务名称和主机工厂，如下所示：

<%@ ServiceHost  
 Service="my-service"  
 Factory="Autofac.Integration.Wcf.AutofacServiceHostFactory, Autofac.Integration.Wcf" %>

### 选择合适的主机工厂

WCF 提供了两种服务主机工厂。Autofac 有对应于每个的实现。

如果你在.svc文件中使用ServiceHostFactory，请更新为AutofacServiceHostFactory。这是 Autofac 和 WCF 最常见的用法。

如果你在.svc文件中使用WebServiceHostFactory，请更新为AutofacWebServiceHostFactory。

### 不使用 .svc 的服务

如果你想不使用 .svc 文件使用服务，Autofac 可以支持。

如上所述，只需在容器中注册服务。

var builder = new ContainerBuilder();  
 builder.RegisterType<Service1>();  
 AutofacHostFactory.Container = builder.Build();

为了使用不带 .svc 的服务，需要在 web.config 文件的 serviceActivation 元素下添加一个工厂条目。这样可以确保使用 AutofacServiceHostFactory 激活服务。

<serviceHostingEnvironment aspNetCompatibilityEnabled="true" multipleSiteBindingsEnabled="true">  
 <serviceActivations>  
 <add factory="Autofac.Integration.Wcf.AutofacServiceHostFactory, Autofac.Integration.Wcf"  
 relativeAddress="~/Service1.svc"  
 service="TestService.Service1, TestService" />  
 </serviceActivations>  
 </serviceHostingEnvironment>

### 扩展式服务

如果你想使用扩展式服务，按照上述方式在容器中注册服务。

var builder = new ContainerBuilder();  
 builder.RegisterType<Service1>();  
 AutofacHostFactory.Container = builder.Build();

然后使用 AutofacServiceHostFactory 和服务实现类型定义一个新的 ServiceRoute 。

RouteTable.Routes.Add(new ServiceRoute("Service1", new AutofacServiceHostFactory(), typeof(Service1)));

最后，将 UrlRoutingModule 添加到 web.config 文件。

<system.webServer>  
 <modules runAllManagedModulesForAllRequests="true">  
 <add name="UrlRoutingModule" type="System.Web.Routing.UrlRoutingModule, System.Web, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a" />  
 </modules>  
 <handlers>  
 <add name="UrlRoutingHandler" preCondition="integratedMode" verb="\*" path="UrlRouting.axd" />  
 </handlers>  
 </system.webServer>

配置完 IIS 后，你就可以通过 http://hostname/appname/Service1访问 WCF 服务了。

### WAS（Windows Activation Service）托管和非 HTTP 激活

在使用 WAS (Windows Activation Service) 部署 WCF 服务时，你无法在 Global.asax 文件的 Application\_Start 事件中构建容器，因为 WAS 不使用标准的 ASP.NET 管道。

另一种方法是将包含名为 AppInitialize 的静态方法的代码文件放在 App\_Code 文件夹中。

namespace MyNamespace  
 {  
 public static class AppStart  
 {  
 public static void AppInitialize()  
 {  
   
 }  
 }  
 }

更多关于 AppInitialize() 的信息，请参阅 "[如何初始化托管的 WCF 服务](https://www.koudingke.cn/go?link=https%3a%2f%2fdocs.microsoft.com%2fen-us%2farchive%2fblogs%2fwenlong%2fhow-to-initialize-hosted-wcf-services)" 。

### 自托管

要使用集成进行自托管 WCF 服务，关键在于在服务主机上使用 AddDependencyInjectionBehavior() 扩展。设置你的容器并进行注册，但**不要设置全局容器**。相反，将容器应用到你的服务主机上。

ContainerBuilder builder = new ContainerBuilder();  
 builder.RegisterType<Service1>();  
 using (var container = builder.Build())  
 {  
 Uri address = new Uri("http://localhost:8080/Service1");  
 ServiceHost host = new ServiceHost(typeof(Service1), address);  
 host.AddServiceEndpoint(typeof(IEchoService), new BasicHttpBinding(), string.Empty);  
   
 host.AddDependencyInjectionBehavior<IService1>(container);  
 host.Description.Behaviors.Add(new ServiceMetadataBehavior {HttpGetEnabled = true, HttpGetUrl = address});  
 host.Open();  
 Console.WriteLine("The host has been opened.");  
 Console.ReadLine();  
 host.Close();  
 Environment.Exit(0);  
 }

**处理** **InstanceContextMode.Single** **服务**

从可扩展性角度来看，使用 InstanceContextMode.Single 并不是一个好主意。允许多个调用者使用 ConcurrencyMode.Multiple 访问单个实例意味着你还需要注意任何共享状态被多个线程访问的问题。如果可能，应创建 InstanceContextMode.PerCall 的服务。

#### IIS/WAS 部署

AutofacServiceHostFactory 会识别标记为 InstanceContextMode.Single 的 WCF 服务，并确保可以从容器提供服务主机的单例实例。如果容器中的服务未使用 SingleInstance() 生命周期范围进行注册，将抛出异常。对于未标记为 InstanceContextMode.Single 的 WCF 服务，在容器中注册 SingleInstance() 服务也是无效的。

#### 自托管

当自托管服务时，可以手动为标记为 InstanceContextMode.Single 的服务进行构造函数注入。这是通过从容器中获取一个 SingleInstance() 服务，然后将其传递给手动创建的 ServiceHost 构造函数来实现的。

var service = container.Resolve<IService1>();  
 var host = new ServiceHost(service, new Uri("http://localhost:8080/Service1"));

### 模拟请求生命周期范围

**如前所述，由于** **WCF** **内部原因，WCF** **对于按请求生命周期依赖没有明确的支持。**

Autofac 通过使用 [实例提供程序](https://www.koudingke.cn/go?link=https%3a%2f%2fmsdn.microsoft.com%2fen-us%2flibrary%2fsystem.servicemodel.dispatcher.iinstanceprovider(v%3dvs.110).aspx)来与 WCF 集成，该提供程序会根据服务实例上下文跟踪服务和依赖项的生命周期范围。

总结来说：基于服务的 [实例上下文模式](https://www.koudingke.cn/go?link=https%3a%2f%2fmsdn.microsoft.com%2fen-us%2flibrary%2fsystem.servicemodel.servicebehaviorattribute.instancecontextmode(v%3dvs.110).aspx)，会为每个服务创建一个生命周期范围。

[如果不设置，默认为“会话”。](https://www.koudingke.cn/go?link=https%3a%2f%2fmsdn.microsoft.com%2fen-us%2flibrary%2fsystem.servicemodel.servicebehaviorattribute.instancecontextmode(v%3dvs.110).aspx)当客户端调用时，会为你的服务创建一个实例，后续来自同一客户端的调用将得到相同的实例。

但是，如果你想模拟按请求的生命周期范围，你可以：

执行这两步后，每次调用都会为一个新的生命周期范围创建一个新的实例（因为 WCF 实例上下文希望每次调用都创建一个新的服务实例）。然后，服务和依赖项只会在该实例上下文生命周期范围内一次性解决——实际上实现了按请求的生命周期。

请注意，如果你的服务实现中有共享的依赖项，且这些依赖项既存在于按调用的服务（InstanceContextMode.PerCall）中，又存在于按会话或单例服务（InstanceContextMode.PerSession 或 InstanceContextMode.Single）中，这可能会产生反效果：对于按调用的服务，不会为每次调用创建新的服务实例，这意味着共享的依赖项（注册为 InstancePerLifetimeScope）在整个服务生命周期内都是单例的。你可能需要尝试将依赖项注册为 InstancePerCall 或 InstancePerLifetimeScope，以获得期望的效果。

### 使用装饰器与服务

标准的 Autofac 服务托管几乎适用于所有情况，但如果你的 WCF 服务实现上使用了 装饰器（不是依赖项的装饰器，而是实际服务实现的装饰器），则需要使用 **多租户 WCF 服务托管机制**，而不是标准的 Autofac 服务主机。

你不需要使用多租户容器、传递租户 ID 或使用其他多租户选项，但确实需要使用多租户服务主机。

这是因为 WCF 托管（.NET 内部）要求主机使用具体类型（而非抽象或接口）初始化，一旦提供了类型，就不能更改。当使用装饰器时，我们直到首次解引用时才会知道最终类型（在将所有装饰器连接在一起等操作之后）。多租户托管机制通过添加另一个动态代理来解决这个问题——一个空的、无目标的具体类，实现了服务接口。当 WCF 主机需要实现时，其中一个动态代理会被启动，而实际的实现（在这种情况下，你的装饰过的 WCF 实现）将是目标。

再次强调，只有当你装饰服务实现类本身时才需要这样做。如果你仅装饰或适应服务实现的依赖项，无需使用多租户主机。标准托管将正常工作。

### 示例

Autofac 示例存储库有一个 [WCF服务实现示例](https://www.koudingke.cn/go?link=https%3a%2f%2fgithub.com%2fautofac%2fExamples%2ftree%2fmaster%2fsrc%2fWcfExample)，以及一个 [作为该服务客户端的MVC应用程序](https://www.koudingke.cn/go?link=https%3a%2f%2fgithub.com%2fautofac%2fExamples%2ftree%2fmaster%2fsrc%2fMvcExample)。

还有示例展示了 [一个多租户WCF服务](https://www.koudingke.cn/go?link=https%3a%2f%2fgithub.com%2fautofac%2fExamples%2ftree%2fmaster%2fsrc%2fMultitenantExample.WcfService)和 [关联的客户端](https://www.koudingke.cn/go?link=https%3a%2f%2fgithub.com%2fautofac%2fExamples%2ftree%2fmaster%2fsrc%2fMultitenantExample.MvcApplication)，说明了如何使用 **多租户服务托管** 。

# Autofac 服务 Fabric 集成

[Autofac.ServiceFabric](https://www.nuget.org/packages/Autofac.ServiceFabric) 包允许将 Autofac 与 [Service Fabric](https://azure.microsoft.com/en-us/services/service-fabric/) 集成。

## 快速入门

在 Main 程序方法中，使用 Autofac 扩展构建容器并注册服务。这将从容器和 ServiceRuntime 中附加服务注册。在应用关闭时，释放容器。

using System;  
 using System.Diagnostics;  
 using System.Reflection;  
 using System.Threading;  
 using Autofac;  
 using Autofac.Integration.ServiceFabric;  
 namespace DemoService  
 {  
 public static class Program  
 {  
 private static void Main()  
 {  
 try  
 {   
 var builder = new ContainerBuilder();  
   
 builder.RegisterModule(new LoggerModule(ServiceEventSource.Current.Message));  
   
 builder.RegisterServiceFabricSupport();  
   
 builder.RegisterStatelessService<DemoStatelessService>("DemoStatelessServiceType");  
   
 using (builder.Build())  
 {  
 ServiceEventSource.Current.ServiceTypeRegistered(  
 Process.GetCurrentProcess().Id,  
 typeof(DemoStatelessService).Name);  
   
 Thread.Sleep(Timeout.Infinite);  
 }  
 }  
 catch (Exception e)  
 {  
 ServiceEventSource.Current.ServiceHostInitializationFailed(e.ToString());  
 throw;  
 }  
 }  
 }  
 }

## 按请求的范围

可以利用 Autofac 支持的 隐式关系 来实现一种 “按请求” 风格的范围机制。

例如，如果你的无状态服务实际上是单例，则想要使用 Func或 Func<Owned>关系（分别对应非可丢弃和可丢弃组件）将自动生成的工厂注入到服务中。然后，你的服务可以根据需要解析依赖项。

例如，假设你有一个无状态的用户服务，它需要读取不应作为单例的后端存储。假设后端存储是 IDisposable的，你想使用 Func<Owned>并像这样注入它：

public class UserService : IUserService  
 {  
 private readonly Func<Owned<IUserStore>> \_userStoreFactory;  
 public UserService(Func<Owned<IUserStore>> userStoreFactory)  
 {  
 \_userStoreFactory = userStoreFactory;  
 }  
 public async Task<string> GetNameAsync(int id)  
 {  
 using (var userStore = \_userStoreFactory())  
 {  
 return await userStore.Value.GetNameAsync(id);  
 }  
 }  
 }

虽然没有关于特定于请求处理的“内置”语义，但你可以利用 隐式关系 做很多事情，因此熟悉它们是值得的。

## 示例

有关服务 Fabric 集成的示例项目，请参阅 [Autofac 示例仓库](https://www.koudingke.cn/go?link=https%3a%2f%2fgithub.com%2fautofac%2fExamples%2ftree%2fmaster%2fsrc%2fServiceFabricDemo) 。

# Autofac 管理扩展框架 (MEF)

Autofac 的 MEF 集成允许你使用 [管理扩展框架](https://www.koudingke.cn/go?link=https%3a%2f%2fmsdn.microsoft.com%2fen-us%2flibrary%2fdd460648(v%3dvs.100).aspx) 在应用程序中暴露扩展点。

要在 Autofac 应用程序中使用 MEF，你必须引用 .NET 框架的 System.ComponentModel.Composition.dll 组件，并从 NuGet 获取 [Autofac.Mef](https://www.koudingke.cn/go?link=https%3a%2f%2fwww.nuget.org%2fpackages%2fAutofac.Mef%2f) 包。

**注意：这是一个单向操作** MEF 集成允许 Autofac 解决在 MEF 中注册的项，但不允许 MEF 解决在 Autofac 中注册的项。

## 在 Autofac 中使用 MEF 扩展

Autofac/MEF 集成允许将 MEF 目录注册到 ContainerBuilder，然后使用 RegisterComposablePartCatalog() 扩展方法。

var builder = new ContainerBuilder();  
 var catalog = new DirectoryCatalog(@"C:\MyExtensions");  
 builder.RegisterComposablePartCatalog(catalog);

支持所有 MEF 目录类型：

* TypeCatalog
* AssemblyCatalog
* DirectoryCatalog

一旦注册了 MEF 目录，其中的导出项可以通过 Autofac 容器或注入到其他组件来解决。例如，假设你有一个使用 MEF 属性定义导出类型的类：

[Export(typeof(IService))]  
 public class Component : IService { }

使用 MEF 目录，你可以注册该类型。Autofac 将找到导出的接口并提供服务。

var catalog = new TypeCatalog(typeof(Component));  
 builder.RegisterComposablePartCatalog(catalog);  
 var container = builder.Build();  
 var obj = container.Resolve<IService>();

## 将 Autofac 组件提供给 MEF 扩展

Autofac 组件不会自动提供给 MEF 扩展导入。也就是说，如果你使用 Autofac 解决一个使用 MEF 注册的组件，只允许使用 MEF 注册的其他服务来满足其依赖关系。

要将 Autofac 组件提供给 MEF，请使用 Exported() 扩展方法：

builder.RegisterType<Component>()  
 .Exported(x => x.As<IService>().WithMetadata("SomeData", 42));

再次强调，这是一次性操作。它允许 Autofac 为在 Autofac 中注册的 MEF 组件提供依赖项 - 它不会将 Autofac 注册项导出到 MEF 目录中以供解决。

## 使用元数据

Autofac MEF 集成为现有的 Lazy 支持添加了 Lazy<T, TMetadata> 关系支持。

例如，假设你有一个定义元数据的接口：

public interface IAgeMetadata  
 {  
 int Age { get; }  
 }

你可以注册 Autofac 服务并使用 Lazy<T, TMetadata> 关系，方法是添加 MEF 元数据注册源：

var builder = new ContainerBuilder();  
 builder.RegisterMetadataRegistrationSources();  
 builder.RegisterType<Component>()  
 .WithMetadata<IAgeMetadata>(m => m.For(value => value.Age, 42));  
 var container = builder.Build();

然后，你可以从此处解决 Lazy<T, TMetadata>：

using (var scope = container.BeginLifetimeScope())  
{  
 var lazy = scope.Resolve<Lazy<Component, IAgeMetadata>>();   
}

**已知问题**：如果你在 MEF 的 [Imports] 上有一个 Lazy<T, TMetadata> 值，此时对象 T**不会延迟实例化**。[在Autofac.Mef仓库中已为此问题提交了一个问题。](https://www.koudingke.cn/go?link=https%3a%2f%2fgithub.com%2fautofac%2fAutofac.Mef%2fissues%2f1)如果你想帮忙，我们很乐意接受一个 PR！

## 已知问题/陷阱

* **Autofac 和 MEF 的集成是单向的。** 它不允许 MEF 组合容器访问在 Autofac 中注册的东西。相反，它基本上采用 MEF 注册语义，并帮助填充一个 Autofac 容器。之后，你期望从 Autofac 解决东西，而不是从 MEF 容器。
* **懒加载元数据导入不工作。**如果你有一个 MEF 的 [Imports] 在一个 Lazy<T, TMetadata> 值上，此时对象 T**不会延迟实例化**。[在 Autofac.Mef 仓库中已为此问题提交了一个问题。](https://www.koudingke.cn/go?link=https%3a%2f%2fgithub.com%2fautofac%2fAutofac.Mef%2fissues%2f1)
* **不支持开放泛型导出。** 如果你在一个 MEF 组件上有一个像 [Export(typeof(A<>))]这样的属性，Autofac 将无法正确处理这个导出，解决此类类型的对象将失败。[在Autofac.Mef仓库中已为此问题提交了一个问题。](https://www.koudingke.cn/go?link=https%3a%2f%2fgithub.com%2fautofac%2fAutofac.Mef%2fissues%2f4)

# Autofac Moq

[Moq](https://www.koudingke.cn/go?link=https%3a%2f%2fgithub.com%2fMoq%2fmoq4)集成包允许你在单元测试中使用 Autofac 容器自动为具体实例和 mock 抽象实例创建依赖项。你可以从 [NuGet](https://www.koudingke.cn/go?link=https%3a%2f%2fwww.nuget.org%2fpackages%2fAutofac.Extras.Moq)获取 Autofac.Extras.Moq包。

## 入门

假设你有一个要测试的系统和一个依赖项：

public class SystemUnderTest  
 {  
 public SystemUnderTest(IDependency dependency)  
 {  
 }  
 }  
 public interface IDependency  
 {  
 }

在编写单元测试时，使用 Autofac.Extras.Moq.AutoMock类来实例化系统。这样会自动为你构造函数注入一个模拟的依赖项。当你创建 AutoMock 工厂时，你可以指定默认的模拟行为：

* AutoMock.GetLoose() - 使用松散模拟行为创建自动模拟。
* AutoMock.GetStrict() - 使用严格模拟行为创建自动模拟。
* AutoMock.GetFromRepository(repo) - 根据现有配置的存储库创建模拟。

[Test]  
 public void Test()  
 {  
 using (var mock = AutoMock.GetLoose())  
 {  
   
 var sut = mock.Create<SystemUnderTest>();  
 }  
 }

## 配置 Moq

你可以像使用 Moq 一样配置自动模拟并/或对其进行断言。

[Test]  
 public void Test()  
 {  
 using (var mock = AutoMock.GetLoose())  
 {   
 mock.Mock<IDependency>().Setup(x => x.GetValue()).Returns("expectedValue");  
 var sut = mock.Create<SystemUnderTest>();  
   
 var actual = sut.DoWork();  
   
 mock.Mock<IDependency>().Verify(x => x.GetValue());  
 Assert.AreEqual("expectedValue", actual);  
 }  
 }  
 public class SystemUnderTest  
 {  
 private readonly IDependency dependency;  
 public SystemUnderTest(IDependency strings)  
 {  
 this.dependency = strings;  
 }  
 public string DoWork()  
 {  
 return this.dependency.GetValue();  
 }  
 }  
 public interface IDependency  
 {  
 string GetValue();  
 }

## 配置特定依赖项

你可以使用 GetLoose、GetStrict 或 GetFromRepository 方法的 beforeBuild 回调参数来配置 AutoMock ，为特定服务类型提供特定的实例（或应用其他注册行为），这类似于配置新的生命周期范围：

[Test]  
 public void Test()  
 {  
 var dependency = new Dependency();  
 using (var mock = AutoMock.GetLoose(cfg => cfg.RegisterInstance(dependency).As<IDependency>()))  
 {  
   
 var dep = mock.Create<IDependency>();  
   
 var underTest = mock.Create<SystemUnderTest>();  
   
 }  
 }

传递给回调的 cfg 参数是一个普通的 Autofac ContainerBuilder 实例，所以你可以像在正常设置中那样进行任何注册行为。

你还可以通过 RegisterMock 扩展方法配置 AutoMock 使用任何现有的模拟：

[Test]  
public void Test()  
 {  
 var mockA = new Mock<IServiceA>();  
 mockA.Setup(x => x.RunA());  
   
 using (var mock = AutoMock.GetLoose(cfg => cfg.RegisterMock(mockA)))  
 {  
   
 var component = mock.Create<TestComponent>();  
   
 }  
}

# Autofac FakeItEasy

[Autofac.Extras.FakeItEasy](https://www.koudingke.cn/go?link=https%3a%2f%2ffakeiteasy.github.io) 集成包允许你在单元测试中使用 Autofac 容器自动为具体和模拟的抽象实例创建假依赖项。

数组类型、IEnumerable 类型和具体类型将通过底层容器创建，该容器已自动配置了 AnyConcreteTypeNotAlreadyRegisteredSource；而其他接口和抽象类将被创建为 FakeItEasy 模拟对象。

你可以在 [NuGet](https://www.koudingke.cn/go?link=https%3a%2f%2fnuget.org%2fpackages%2fAutofac.Extras.FakeItEasy) 上获取 Autofac.Extras.FakeItEasy 包。

## 开始使用

假设你有一个要测试的系统及其依赖项：

public class SystemUnderTest  
 {  
 public SystemUnderTest(IDependency dependency)  
 {  
 }  
 }  
 public interface IDependency  
 {  
 }

在编写单元测试时，使用 Autofac.Extras.FakeItEasy.AutoFake类来实例化要测试的系统。这样会在构造函数中为你自动注入一个假依赖项。

[Test]  
 public void Test()  
 {  
 using (var fake = new AutoFake())  
 {  
   
 var sut = fake.Resolve<SystemUnderTest>();  
 }  
 }

## 配置模拟对象

你可以像平时使用 FakeItEasy 那样配置自动模拟对象，并/或对它们进行断言。

[Test]  
 public void Test()  
 {  
 using (var fake = new AutoFake())  
 {  
   
 A.CallTo(() => fake.Resolve<IDependency>().GetValue()).Returns("expected value");  
 var sut = fake.Resolve<SystemUnderTest>();  
   
 var actual = sut.DoWork();  
   
 A.CallTo(() => fake.Resolve<IDependency>().GetValue()).MustHaveHappened();  
 Assert.AreEqual("expected value", actual);  
 }  
 }  
 public class SystemUnderTest  
 {  
 private readonly IDependency dependency;  
 public SystemUnderTest(IDependency strings)  
 {  
 this.dependency = strings;  
 }  
 public string DoWork()  
 {  
 return this.dependency.GetValue();  
 }  
 }  
 public interface IDependency  
 {  
 string GetValue();  
 }

## 配置特定依赖项

你可以配置 AutoFake为给定服务类型的特定实例提供服务：

[Test]  
 public void Test()  
 {  
 using (var fake = new AutoFake())  
 {  
 var dependency = new Dependency();  
 fake.Provide(dependency);  
   
 }  
 }

你也可以为给定服务类型配置特定实现类型：

[Test]  
 public void Test()  
 {  
 using (var fake = new AutoFake())  
 {  
   
 fake.Provide<IDependency, Dependency>();  
   
 fake.Provide<IOtherDependency, OtherDependency>(  
 new NamedParameter("id", "service-identifier"),  
 new TypedParameter(typeof(Guid), Guid.NewGuid()));  
   
 }  
 }

## 模拟选项

你可以使用 AutoFake的可选构造函数参数指定模拟对象的创建选项：

using (var fake = new AutoFake(  
 strict: true,  
 callsBaseMethods: true,   
 onFakeCreated: f => { ... }))  
 {  
   
 }

混合这些选项时要小心。指定 callsBaseMethods与其他任何选项一起使用没有意义，因为它会覆盖它们。当同时指定 onFakeCreated和 strict时，onFakeCreated提供的配置将根据需要覆盖 strict。

# 最佳实践与推荐

Best Practices and Recommendations

你总是可以在 [StackOverflow](https://www.koudingke.cn/go?link=https%3a%2f%2fstackoverflow.com%2fquestions%2ftagged%2fautofac)上使用 autofac标签或在 [讨论组](https://www.koudingke.cn/go?link=https%3a%2f%2fgroups.google.com%2fforum%2f%23forum%2fautofac)中寻求 Autofac 的使用指导，但这些快速提示能帮助你开始。

## 总是从嵌套生命周期中解析依赖项

Always Resolve Dependencies from Nested Lifetimes

Autofac 设计用于为你 跟踪和释放资源 。为了确保这一点，确保长运行的应用程序被划分为工作单元（请求或事务），并且服务通过工作单元级别的生命周期作用域进行解析。ASP.NET 中的 请求级生命周期作用域支持 就是一个例子。

## 通过模块结构化配置

Structure Configuration with Modules

Autofac 模块 为容器配置提供了结构，并允许在部署时注入设置。与其单独使用 XML 配置 ，不如考虑使用模块以获得更灵活的方法。模块始终可以与 XML 配置结合使用，以获得两全其美的体验。

## 在委托注册中使用 As()

Use As() in Delegate Registrations

Autofac 根据你用来注册组件的表达式推断实现类型：

builder.Register(c => new Component()).As<IComponent>();

...使得 Component 类型成为组件的 LimitType 。以下其他类型转换机制等效，但不提供正确的 LimitType：

builder.Register(c => (IComponent)new Component());  
 builder.Register<IComponent>(c => new Component());

## 使用构造注入

Use Constructor Injection

使用构造注入为必需依赖项和属性注入可选依赖项的概念相当知名。然而，另一种方法是使用 ["空对象模式"](https://www.koudingke.cn/go?link=http%3a%2f%2fen.wikipedia.org%2fwiki%2fNull_object_pattern)或 ["特殊情况模式"](https://www.koudingke.cn/go?link=http%3a%2f%2fmartinfowler.com%2feaaCatalog%2fspecialCase.html)，为可选服务提供默认的、无操作的实现。这防止了在组件实现中的特殊案例代码（例如 if (Logger != null) Logger.Log("message");）的可能性。

## 使用关系类型，而不是服务定位器

Use Relationship Types, Not Service Locators

让组件访问容器，将其存储在公共静态属性中，或将像 Resolve()这样的函数添加到全局"IoC"类上，会违背使用依赖注入的目的。此类设计与 [服务定位器模式](https://www.koudingke.cn/go?link=http%3a%2f%2fmartinfowler.com%2farticles%2finjection.html%23UsingAServiceLocator)有更多共同点。

如果组件对容器（或生命周期作用域）有依赖，看看它们是如何使用容器获取服务的，然后将这些服务添加到组件（依赖注入）构造参数中。

对于需要实例化其他组件或以更高级方式与容器交互的组件，请使用 关系类型 。

## 按最不特殊到最的特殊的顺序去注册组件

Register Components from Least-to-Most Specific

先 general, 再特殊

Autofac **默认情况下会覆盖(override)**组件注册。这意味着应用程序可以注册所有默认组件，然后读取一个关联的配置文件来覆盖任何已针对部署环境自定义的组件。

## 使用性能检查器

**Use Profilers for Performance Checking**

在进行任何性能优化或假设潜在内存泄漏之前，**始终运行性能检查器**，如 [SlimTune](https://www.koudingke.cn/go?link=http%3a%2f%2fcode.google.com%2fp%2fslimtune%2f) ，[dotTrace](https://www.koudingke.cn/go?link=http%3a%2f%2fwww.jetbrains.com%2fprofiler%2f) 或 [ANTS](https://www.koudingke.cn/go?link=http%3a%2f%2fwww.red-gate.com%2fproducts%2fdotnet-development%2fants-performance-profiler%2f) ，查看时间真正花费在哪里。可能并非你想象的那样。

## 一次注册，多次解析

**Register Once, Resolve Many**

如果可以避免的话，不要在工作单元期间注册组件；注册组件的成本高于解析一个组件。使用嵌套生命周期作用域和适当的 实例作用域 来保持每个工作单元实例的独立性。

## 使用 lambda 注册频繁使用的组件

Register Frequently-Used Components with Lambdas

如果你确实需要从 Autofac 中挤出额外的性能，你的最佳选择是找出创建最多的组件，并使用表达式而不是类型进行注册，例如：

builder.RegisterType<Component>();

变成：

builder.Register(c => new Component());

这可以将 Resolve() 调用的速度提高高达 10 倍，但只对出现在许多对象图中的组件有意义。有关 lambda 组件的更多内容，请参阅 注册文档 。

## 考虑容器为不可变的

Consider a Container as Immutable

从 Autofac 5.x 开始，容器是不可变的。在构建后更新容器可能存在一些潜在风险。例如：

* 已自动启动的组件可能已经运行并使用了你在更新过程中覆盖的注册。这些自动启动的组件不会重新运行。
* 已经解析的服务可能基于添加的内容引用错误的依赖项。
* 已经解析的可丢弃组件可能已经被解决，并且会一直存在，直到拥有其生命周期作用域被释放——即使新的注册暗示应该不使用可丢弃组件。
* 注册订阅生命周期事件的组件可能在更新后订阅了错误的事件——不是所有的事件在更新过程中都会重新初始化。

为了防止这些风险成为问题，不再可以选择在构造后更新容器。

**相反，考虑在子生命周期作用域中注册更新或更改。** **有关这方面的示例，请** **参阅生命周期作用域文档** **。**

## 优化或避免诊断

Optimize or Avoid Diagnostics

System.Diagnostics.DiagnosticSource 的诊断集成性能相当不错，但附加的追踪器可能会影响整体性能。例如，任何追踪器（如 DefaultDiagnosticTracer ）都会跟踪完整的解析操作链，这会分配内存并使用资源来保留操作数据，直到完整解析操作完成。

如果没有诊断监听器，整体性能会更好，但如果有的话，考虑使用非常快速且低分配的诊断监听器。例如，当操作发生时，不需要额外的内存或跟踪就可以记录一条消息。这可能比在完成前构建健壮的跟踪所需的开销更低。

# 高级话题

Advanced Topics

# 注册源

Registration Sources

一个 *注册源* 是一种动态向 Autofac 组件上下文（如容器或生命周期范围）添加注册的方式。

注册源是通过实现 IRegistrationSource接口创建的。许多 Autofac 功能都是通过这种方式实现的——例如，隐式关系类型 是通过注册源添加的。（你不会真的认为我们手动为每个单例类型都向容器中注册了它们吧？）[Nick Blumhardt有一篇关于这如何工作的很棒的博客文章。](https://www.koudingke.cn/go?link=http%3a%2f%2fnblumhardt.com%2f2010%2f01%2fdeclarative-context-adapters-autofac2%2f)

当你没有可以添加到容器中的有限数量的注册时，注册源非常有用。很多时候，程序集扫描 或者 模块使用 可以解决动态注册问题……但当其他方法都失败或者这些方式无法满足你的需求时，可能需要使用注册源。

## 任何未注册的具体类型源

“Any Concrete Type Not Already Registered” Source

AnyConcreteTypeNotAlreadyRegisteredSource，我们称之为 ACTNARS，是随附于 Autofac 的一个示例注册源，它允许你根据是否已明确注册，从 Autofac 容器中解析任何具象类型。从其他控制反转容器迁移到 Autofac 的人可能会习惯这种行为，因此 ACTNARS 是一种弥合这种差距的方法。

你可以使用 Autofac.Features.ResolveAnything.AnyConcreteTypeNotAlreadyRegisteredSource，将其添加到你的容器中。

var builder = new ContainerBuilder();  
builder.RegisterSource(new AnyConcreteTypeNotAlreadyRegisteredSource());  
var container = builder.Build();

## 可逆注册源

Contravariant Registration Source

ContravariantRegistrationSource 对于注册需要在 [可逆上下文中](https://www.koudingke.cn/go?link=https%3a%2f%2fdocs.microsoft.com%2fen-us%2fdotnet%2fstandard%2fgenerics%2fcovariance-and-contravariance)（使用比原始指定更通用/不那么衍生的类型）后来解析的类型很有帮助。

**如果使用此源，请先进行注册。** 正如标准注册一样，注册源按注册顺序进行评估。如果你首先注册一个开放泛型（它也是内部的注册源），然后注册 ContravariantRegistrationSource，则不会得到预期的结果。

在处理器模式中使用 ContravariantRegistrationSource 很常见：

public interface IHandler<in TCommand>  
{  
 void Handle(TCommand command);  
}  
  
public class BaseCommandHandler : IHandler<BaseCommand>  
{  
 public void Handle(BaseCommand command)  
 {  
 Console.WriteLine(command.GetType().Name);  
 }  
}  
  
public class BaseCommand  
{  
}  
  
public class DerivedCommand : BaseCommand  
{  
}  
  
var builder = new ContainerBuilder();  
builder.RegisterSource(new ContravariantRegistrationSource());  
builder.RegisterType<BaseCommandHandler>().As<IHandler<BaseCommand>>();  
var container = builder.Build();  
var a = container.Resolve<IHandler<BaseCommand>>();  
a.Handle(new BaseCommand());   
var b = container.Resolve<IHandler<DerivedCommand>>();  
b.Handle(new DerivedCommand());

## 实现1个注册源

Implementing a Registration Source

通过一个简单的示例来展示如何实现注册源是最好的方法。

假设你有一个工厂，负责生成某种事件处理器类。你需要通过工厂而不是通过 Autofac 生成它们，因此处理器本身不会与 Autofac 注册。同时，也没有好的方式说“当有人请求任何事件处理器时，通过这个特殊工厂生成它”。这是一个很好的例子，说明何时可以使用注册源。

对于示例，让我们定义一个简单的事件处理器基/抽象类和几个实现。

public abstract class BaseHandler  
 {  
 public virtual string Handle(string message)  
 {  
 return "Handled: " + message;  
 }  
 }  
 public class HandlerA : BaseHandler  
 {  
 public override string Handle(string message)  
 {  
 return "[A] " + base.Handle(message);  
 }  
 }  
 public class HandlerB : BaseHandler  
 {  
 public override string Handle(string message)  
 {  
 return "[B] " + base.Handle(message);  
 }  
 }

现在，让我们创建一个工厂接口和实现。

public interface IHandlerFactory  
 {  
 T GetHandler<T>() where T : BaseHandler;  
 }  
 public class HandlerFactory : IHandlerFactory  
 {  
 public T GetHandler<T>() where T : BaseHandler  
 {  
 return (T)Activator.CreateInstance(typeof(T));  
 }  
 }

最后，让我们创建一些使用处理器的消费者类。

public class ConsumerA  
 {  
 private HandlerA \_handler;  
 public ConsumerA(HandlerA handler)  
 {  
 this.\_handler = handler;  
 }  
 public void DoWork()  
 {  
 Console.WriteLine(this.\_handler.Handle("ConsumerA"));  
 }  
 }  
 public class ConsumerB  
 {  
 private HandlerB \_handler;  
 public ConsumerB(HandlerB handler)  
 {  
 this.\_handler = handler;  
 }  
 public void DoWork()  
 {  
 Console.WriteLine(this.\_handler.Handle("ConsumerB"));  
 }  
 }

现在我们有了服务和消费者，让我们创建一个注册源。在示例源中，我们将……

1. 检查是否正在请求 BaseHandler类型。如果不是，则源不会提供任何注册来满足解析请求。
2. 为请求的具体 BaseHandler子类构建动态注册，这将包括调用提供者/工厂获取实例的 lambda。
3. 将动态注册返回给解析操作，以便它完成工作。

这是注册源的代码。

using Autofac;  
 using Autofac.Core;  
 using Autofac.Core.Activators.Delegate;  
 using Autofac.Core.Lifetime;  
 using Autofac.Core.Registration;  
 public class HandlerRegistrationSource : IRegistrationSource  
 {  
 public IEnumerable<IComponentRegistration> RegistrationsFor(  
 Service service,  
 Func<Service, IEnumerable<ServiceRegistration>> registrationAccessor)  
 {  
 var swt = service as IServiceWithType;  
 if(swt == null || !typeof(BaseHandler).IsAssignableFrom(swt.ServiceType))  
 {  
   
 return Enumerable.Empty<IComponentRegistration>();  
 }  
   
 var registration = new ComponentRegistration(  
 Guid.NewGuid(),  
 new DelegateActivator(swt.ServiceType, (c, p) =>  
 {  
   
 var provider = c.Resolve<IHandlerFactory>();  
   
 var method = provider.GetType().GetMethod("GetHandler").MakeGenericMethod(swt.ServiceType);  
   
 return method.Invoke(provider, null);  
 }),  
 new CurrentScopeLifetime(),  
 InstanceSharing.None,  
 InstanceOwnership.OwnedByLifetimeScope,  
 new [] { service },  
 new Dictionary<string, object>());  
 return new IComponentRegistration[] { registration };  
 }  
 public bool IsAdapterForIndividualComponents { get{ return false; } }  
 }

最后一步是将所有内容注册到 Autofac——注册源、工厂和消费者类。但是请注意，我们不必注册实际的处理器，因为注册源会处理这些。

var builder = new ContainerBuilder();  
 builder.RegisterType<HandlerFactory>().As<IHandlerFactory>();  
 builder.RegisterSource(new HandlerRegistrationSource());  
 builder.RegisterType<ConsumerA>();  
 builder.RegisterType<ConsumerB>();  
 var container = builder.Build();

现在，当你解析处理器消费者之一时，你将获得正确的处理器。

using (var scope = container.BeginLifetimeScope())  
{  
 var consumer = scope.Resolve<ConsumerA>();   
 consumer.DoWork();  
}

# 适配器和装饰器

Adapters and Decorators

## 适配器

Adapters

[适配器模式](http://en.wikipedia.org/wiki/Adapter_pattern)将一个服务合同适配（就像包装一样）到另一个。

这篇 [入门文章](http://nblumhardt.com/2010/04/lightweight-adaptation-coming-soon/) 描述了一个适配器模式的具体示例，以及如何在 Autofac 中使用它。

Autofac 提供了内置的适配器注册，因此你可以注册一组服务，并使它们自动适应不同的接口。

var builder = new ContainerBuilder();  
 builder.RegisterType<SaveCommand>()  
 .As<ICommand>()  
 .WithMetadata("Name", "Save File");  
 builder.RegisterType<OpenCommand>()  
 .As<ICommand>()  
 .WithMetadata("Name", "Open File");  
 builder.RegisterAdapter<Meta<ICommand>, ToolbarButton>(  
 cmd => new ToolbarButton(cmd.Value, (string)cmd.Metadata["Name"]));  
 var container = builder.Build();  
 var buttons = container.Resolve<IEnumerable<ToolbarButton>>();

## 装饰器

[装饰器模式](https://www.koudingke.cn/go?link=https%3a%2f%2fen.wikipedia.org%2fwiki%2fDecorator_pattern)与适配器模式类似，其中一个服务“包裹”另一个服务。然而，与适配器相反，装饰器暴露的是它们装饰的 *相同服务*。使用装饰器的目的是在不改变对象签名的情况下向对象添加功能。

Autofac 提供了内置的装饰器注册，因此你可以注册服务并自动用装饰类包裹它们。

### 简化语法

从 Autofac 4.9.0 开始，它提供了一种简化装饰器 [语法](https://www.koudingke.cn/go?link=https%3a%2f%2falexmg.com%2fposts%2fupcoming-decorator-enhancements-in-autofac-4-9)，可以作为经典语法（下面所示）的替代方案。它使用起来更简单，比早期机制更具灵活性。

var builder = new ContainerBuilder();  
 builder.RegisterType<SaveCommandHandler>()  
 .As<ICommandHandler>();  
 builder.RegisterType<OpenCommandHandler>()  
 .As<ICommandHandler>();  
 builder.RegisterDecorator<LoggingDecorator, ICommandHandler>();  
 builder.RegisterDecorator<DiagnosticDecorator, ICommandHandler>();  
 var container = builder.Build();  
 var handlers = container.Resolve<IEnumerable<ICommandHandler>>();

如果你事先不知道类型，你可以手动指定，而不是使用泛型：

builder.RegisterDecorator(typeof(LoggingDecorator), typeof(ICommandHandler));  
 builder.RegisterDecorator(typeof(DiagnosticDecorator), typeof(ICommandHandler));

如果你想手动实例化你的装饰器或执行更复杂的装饰器创建，这也是可能的。

builder.RegisterDecorator<ICommandHandler>(  
 (context, parameters, instance) => new ComplexDecorator(instance)  
 );

在 Lambda 中，context 是正在发生的决议的 IComponentContext（因此，如果需要，你可以解决其他东西）；parameters 是一个 IEnumerable，其中包含传递的所有参数；而 instance 是正在被装饰的服务实例。请记住，如果存在多个装饰器链，则 instance可能是一个 *装饰器实例* 而不是正在被装饰的根/基础事物。

开放泛型支持装饰。

var builder = new ContainerBuilder();  
 builder.RegisterGeneric(typeof(CommandHandler<>)  
 .As(ICommandHandler<>);  
 builder.RegisterGenericDecorator(typeof(LoggingDecorator<>), typeof(ICommandHandler<>));  
 builder.RegisterGenericDecorator(typeof(DiagnosticDecorator<>), typeof(ICommandHandler<>));  
 var container = builder.Build();  
 var handler = container.Resolve<ICommandHandler<Save>>();

装饰可以有条件。会为注册提供一个上下文对象，允许你决定是否应用装饰器：

builder.RegisterDecorator<ErrorHandlerDecorator, ICommandHandler>(  
 context => !context.AppliedDecorators.Any());  
 builder.RegisterGenericDecorator(  
 typeof(ErrorHandlerDecorator<>),  
 typeof(ICommandHandler<>),  
 context => !context.AppliedDecorators.Any());

这些 Lambda 中的 context是一个 IDecoratorContext，其中包含有关已应用装饰器列表、实际正在解决的服务类型等信息。

你可以使用这个上下文在你的装饰器中做出决策，如果需要的话，可以将其注入到装饰器的构造函数参数中。

public class ErrorHandlerDecorator : ICommandHandler  
 {  
 private readonly ICommandHandler \_decorated;  
 private readonly IDecoratorContext \_context;  
 public ErrorHandlerDecorator(ICommandHandler decorated, IDecoratorContext context)  
 {  
 this.\_decorated = decorated ?? throw new ArgumentNullException(nameof(decorated));  
 this.\_context = context ?? throw new ArgumentNullException(nameof(context));  
 }  
 public void HandleCommand(Command command)  
 {  
 if(this.\_context.ImplementationType.GetCustomAttribute<SkipHandlingAttribute>() != null)  
 {  
   
 }  
 else  
 {  
   
 }  
 }  
 }

**不能为装饰器指定生命周期作用域。** 装饰器的生命周期与其所装饰的事物的生命周期相关联。服务及其所有装饰器都在同一时间销毁。如果你装饰一个单例，所有装饰器也将是单例。如果你装饰一个按请求实例化的东西（例如，在 Web 应用程序中），装饰器也会在整个请求期间存在。

### 经典语法

自 Autofac 2.4 起，经典语法就已经存在，并且至今仍在工作。它比新语法更复杂，但如果你有一些现有的使用它的代码，该代码将继续有效。

这篇 [文章](https://www.koudingke.cn/go?link=http%3a%2f%2fnblumhardt.com%2f2011%2f01%2fdecorator-support-in-autofac-2-4%2f)详细介绍了 Autofac 中装饰器的工作原理。

var builder = new ContainerBuilder();  
 builder.RegisterType<SaveCommandHandler>()  
 .Named<ICommandHandler>("handler");  
 builder.RegisterType<OpenCommandHandler>()  
 .Named<ICommandHandler>("handler");  
 builder.RegisterDecorator<ICommandHandler>(  
 (c, inner) => new CommandHandlerDecorator(inner),  
 fromKey: "handler");  
 var container = builder.Build();  
 var handlers = container.Resolve<IEnumerable<ICommandHandler>>();

你也可以使用开放的装饰器注册。

var builder = new ContainerBuilder();  
 builder.RegisterGeneric(typeof(CommandHandler<>))  
 .Named("handler", typeof(ICommandHandler<>));  
 builder.RegisterGenericDecorator(  
 typeof(CommandHandlerDecorator<>),  
 typeof(ICommandHandler<>),  
 fromKey: "handler");  
 var container = builder.Build();  
 var mailHandlers = container.Resolve<IEnumerable<ICommandHandler<EmailCommand>>>();

如果你在一个 WCF 服务实现类上使用装饰器，有关 WCF 集成的一些特殊注意事项。

# Autofac 循环依赖

Circular Dependencies

循环依赖是指组件之间的运行时相互依赖。

## 属性/属性依赖

当一个类 ( DependsByProperty1 ) 依赖于另一个类型的属性 ( DependsByProperty2 )，而这种依赖反过来又存在于 ( DependsByProperty2 ) 的属性中时，就会出现这种情况。

在这种情况下，请记住以下几点：

* **使属性依赖可设置**。属性必须是可写入的。
* 使用 **PropertiesAutowired** 注册类型。 确保允许循环依赖行为。

*·* **两种类型都不能注册为 InstancePerDependency** 。

*如果任何类型设置为工厂作用域，你将无法得到期望的结果（即两个类型相互引用）。你可以按任意方式进行作用域设置——如* *SingleInstance* 、*InstancePerLifetimeScope* *或其他任何作用域，但不能是工厂作用域。*

示例：

class DependsByProp1  
 {  
 public DependsByProp2 Dependency { get; set; }  
 }  
 class DependsByProp2  
 {  
 public DependsByProp1 Dependency { get; set; }  
 }  
 var cb = new ContainerBuilder();  
 cb.RegisterType<DependsByProp1>()  
 .InstancePerLifetimeScope()  
 .PropertiesAutowired(PropertyWiringOptions.AllowCircularDependencies);  
 cb.RegisterType<DependsByProp2>()  
 .InstancePerLifetimeScope()  
 .PropertiesAutowired(PropertyWiringOptions.AllowCircularDependencies);

## 构造函数/属性依赖

当一个类 ( DependsByCtor ) 通过构造函数依赖于另一个类型 ( DependsByProperty ) ，而这种依赖反过来又存在于 ( DependsByProperty ) 的属性中时，就会出现这种情况。

在这种情况下，请记住以下几点：

* **使属性依赖可设置。** *具有属性依赖的类型上的属性必须是可写入的。*
* **使用 PropertiesAutowired 注册具有属性依赖的类型** 。确保允许循环依赖行为。
* **两种类型都不能注册为InstancePerDependency** **。**如果任何类型设置为工厂作用域，你将无法得到期望的结果（即两个类型相互引用）。你可以按任意方式进行作用域设置——如 *SingleInstance* *、InstancePerLifetimeScope* *或其他任何作用域，但不能是工厂作用域。*

示例：

class DependsByCtor  
 {  
 public DependsByCtor(DependsByProp dependency) { }  
 }  
 class DependsByProp  
 {  
 public DependsByCtor Dependency { get; set; }  
 }  
 var cb = new ContainerBuilder();  
 cb.RegisterType<DependsByCtor>()  
 .InstancePerLifetimeScope();  
 cb.RegisterType<DependsByProp>()  
 .InstancePerLifetimeScope()  
 .PropertiesAutowired(PropertyWiringOptions.AllowCircularDependencies);

## 构造函数/构造函数依赖

具有**循环构造函数依赖**的两个类型 **不被支持**。尝试使用这种方式注册类型时，会抛出异常。

你可以尝试使用 DynamicProxy2 扩展和一些创造性编程来解决这个问题。

# Autofac 组件元数据/属性元数据

**如果你熟悉管理扩展框架** **(Managed Extensibility Framework, MEF)，你可能已经见过使用组件元数据的例子。**

元数据是关于组件的信息，与组件一起存储，可以在不创建组件实例的情况下访问。

## 在组件注册时添加元数据

描述元数据的值在组件注册时关联到组件。每个元数据项是一个名称/值对：

builder.Register(c => new ScreenAppender())  
 .As<ILogAppender>()  
 .WithMetadata("AppenderName", "screen");

同样的内容也可以用在 部署时配置 中：

{  
 "components": [  
 {  
 "type": "MyApp.Components.Logging.ScreenAppender, MyApp",  
 "services": [  
 {  
 "type": "MyApp.Services.Logging.ILogAppender, MyApp"  
 }  
 ],  
 "metadata": [  
 {  
 "key": "AppenderName",  
 "value": "screen",  
 "type": "System.String, mscorlib"  
 }  
 ]  
 }  
 ]  
 }

## 消费元数据

与常规属性不同，元数据项独立于组件本身。

这使得它在基于运行时条件选择多个组件或元数据不是组件实现固有属性的情况下很有用。元数据可以代表 ITask应该运行的时间，或者 ICommand的按钮标题。

其他组件可以通过 Meta类型来消费元数据。

public class Log  
 {  
 readonly IEnumerable<Meta<ILogAppender>> \_appenders;  
 public Log(IEnumerable<Meta<ILogAppender>> appenders)  
 {  
 \_appenders = appenders;  
 }  
 public void Write(string destination, string message)  
 {  
 var appender = \_appenders.First(a => a.Metadata["AppenderName"].Equals(destination));  
 appender.Value.Write(message);  
 }  
 }

要不创建目标组件的情况下消费元数据，可以使用 Meta<Lazy>或.NET 4 的 Lazy<T, TMetadata>类型，如下面所示。

## 强类型元数据

为了避免使用字符串键来描述元数据，可以定义一个包含每个元数据项的公共读写属性的元数据类：

public class AppenderMetadata  
 {  
 public string AppenderName { get; set; }  
 }

在注册时，可以使用重载的 WithMetadata方法将值关联起来：

builder.Register(c => new ScreenAppender())  
 .As<ILogAppender>()  
 .WithMetadata<AppenderMetadata>(m =>  
 m.For(am => am.AppenderName, "screen"));

注意使用了强类型的 AppenderName属性。

注册和消费元数据是分开的，因此强类型元数据可以通过弱类型技术消费，反之亦然。

你还可以通过 DefaultValue 属性提供默认值：

public class AppenderMetadata  
 {  
 [DefaultValue("screen")]  
 public string AppenderName { get; set; }  
 }

如果你的解决方案能够引用 System.ComponentModel.Composition ，可以使用 System.Lazy<T, TMetadata> 类型从强类型元数据类中消费值：

public class Log  
 {  
 readonly IEnumerable<Lazy<ILogAppender, LogAppenderMetadata>> \_appenders;  
 public Log(IEnumerable<Lazy<ILogAppender, LogAppenderMetadata>> appenders)  
 {  
 \_appenders = appenders;  
 }  
 public void Write(string destination, string message)  
 {  
 var appender = \_appenders.First(a => a.Metadata.AppenderName == destination);  
 appender.Value.Write(message);  
 }  
 }

另一个巧妙的技巧是将元数据字典传递给元数据类的构造函数：

public class AppenderMetadata  
 {  
 public AppenderMetadata(IDictionary<string, object> metadata)  
 {  
 AppenderName = (string)metadata["AppenderName"];  
 }  
 public string AppenderName { get; set; }  
 }

## 接口为基础的元数据

如果你能访问 System.ComponentModel.Composition 并包含对 Autofac.Mef 包的引用，你可以使用接口而不是类作为元数据。

接口应定义具有每个元数据项的可读属性：

public interface IAppenderMetadata  
 {  
 string AppenderName { get; }  
 }

在将元数据注册到接口类型之前，还必须调用 ContainerBuilder 上的 RegisterMetadataRegistrationSources 方法。

builder.RegisterMetadataRegistrationSources();

在注册时，可以使用接口和重载的 WithMetadata 方法将值关联起来：

builder.Register(c => new ScreenAppender())  
 .As<ILogAppender>()  
 .WithMetadata<IAppenderMetadata>(m =>  
 m.For(am => am.AppenderName, "screen"));

元数据的获取方式与基于类的元数据相同。

## 属性为基础的元数据

Autofac.Extras.AttributeMetadata包使你可以通过属性指定元数据。Autofac 核心支持允许组件根据属性过滤传入的依赖项。

要在解决方案中使带有属性的元数据工作，请按照以下步骤操作：

**1.** **创建元数据属性**

**2.** **应用元数据属性**

**3.** **在消费中使用元数据过滤**

**4.** **确保容器使用你的属性**

### 创建元数据属性

元数据属性是一个实现了 System.Attribute 的 System.ComponentModel.Composition.MetadataAttributeAttribute 的类。

该属性上的任何公开可读属性将成为名称/值属性对 - 元数据名称将是属性名称，值将是属性值。

在下面的示例中，AgeMetadataAttribute 将提供一个元数据名称/值对，其中名称将是 Age（属性名称），值将是构造期间指定的属性值。

[MetadataAttribute]  
 public class AgeMetadataAttribute : Attribute  
 {  
 public int Age { get; private set; }  
 public AgeMetadataAttribute(int age)  
 {  
 Age = age;  
 }  
 }

### 应用元数据属性

一旦有了元数据属性，就可以将其应用于组件类型以提供元数据。

public interface IArtwork  
 {  
 void Display();  
 }  
 [AgeMetadata(100)]  
 public class CenturyArtwork : IArtwork  
 {  
 public void Display() { ... }  
 }

### 在消费中使用元数据过滤

除了通过属性提供元数据外，你还可以设置自动过滤组件。这将帮助根据提供的元数据设置构造函数参数。

可以根据 服务键 或注册元数据进行过滤。这种基于属性的过滤无需自定义元数据属性即可完成。

KeyFilterAttribute、MetadataFilterAttribute 和 WithAttributeFiltering 扩展方法可以在核心 Autofac 包的 Autofac.Features.AttributeFilters 命名空间中找到。

#### KeyFilterAttribute

KeyFilterAttribute 允许你选择特定键的服务进行消费。

以下示例显示了一个需要具有特定键的组件的类：

public class ArtDisplay : IDisplay  
 {  
 public ArtDisplay([KeyFilter("Painting")] IArtwork art) { ... }  
 }

要使用的组件需要注册具有指定名称的关键服务。还需要为容器注册组件，以便容器知道要查找它。

var builder = new ContainerBuilder();  
 builder.RegisterType<MyArtwork>().Keyed<IArtwork>("Painting");  
 builder.RegisterType<ArtDisplay>().As<IDisplay>().WithAttributeFiltering();  
 var container = builder.Build();

MetadataFilterAttribute 允许你根据特定元数据值过滤组件。

以下示例显示了一个需要具有特定元数据值的组件的类：

public class ArtDisplay : IDisplay  
 {  
 public ArtDisplay([MetadataFilter("Age", 100)] IArtwork art) { ... }  
 }

要使用的组件需要注册具有指定名称/值对的元数据。可以使用前面示例中看到的带属性元数据类，或者在注册时手动指定元数据。还需要为容器注册组件，以便容器知道要查找它。

var builder = new ContainerBuilder();  
 builder.RegisterModule<AttributedMetadataModule>();  
 builder.RegisterType<CenturyArtwork>().As<IArtwork>();  
 builder.RegisterType<ArtDisplay>().As<IDisplay>().WithAttributeFiltering();  
 var container = builder.Build();

### 确保容器使用你的属性

你创建的元数据属性不会默认使用。为了告诉容器你正在使用元数据属性，需要将 AttributedMetadataModule注册到容器中。

var builder = new ContainerBuilder();  
 builder.RegisterModule<AttributedMetadataModule>();  
 builder.RegisterType<CenturyArtwork>().As<IArtwork>();  
 var container = builder.Build();

如果你在构造函数中使用元数据过滤器（例如 KeyFilterAttribute 或 WithAttributeFiltering），则需要使用 WithAttributeFiltering 扩展来注册这些组件。请注意，如果你仅使用过滤器而不用属性元数据，实际上并不需要 AttributedMetadataModule。元数据过滤器可以独立使用。

var builder = new ContainerBuilder();  
 builder.RegisterType<ArtDisplay>().As<IDisplay>().WithAttributeFiltering();  
 var container = builder.Build();

# Autofac 命名和键控服务

Named and Keyed Services

Autofac 提供了三种常见的服务标识方法。最常见的就是按类型识别：

builder.RegisterType<OnlineState>().As<IDeviceState>();

在这个例子中，将 IDeviceState 类型的 OnlineState 组件关联起来。你可以使用 Resolve() 方法通过服务类型获取组件实例：

var r = container.Resolve<IDeviceState>();

然而，你也可以通过字符串名称或对象键来识别服务。

## 命名服务

Named Services

使用服务名称可以进一步标识服务。在这种情况下，Named()注册方法取代了 As()。

builder.RegisterType<OnlineState>().Named<IDeviceState>("online");

要获取命名服务，可以使用 ResolveNamed()方法：

var r = container.ResolveNamed<IDeviceState>("online");

命名服务实际上就是使用字符串作为键的键入服务，因此接下来描述的技术同样适用于命名服务。

## 键控服务

Keyed Services

在某些情况下，使用字符串作为组件名称很方便，但在其他情况下我们可能希望使用其他类型的键。键入服务提供了这种能力。

例如，在我们的示例中，枚举可能描述设备的不同状态：

public enum DeviceState { Online, Offline }

每个枚举值对应于服务的一个实现：

public class OnlineState : IDeviceState { }

然后可以像下面这样使用枚举值作为实现的键进行注册。

var builder = new ContainerBuilder();  
 builder.RegisterType<OnlineState>().Keyed<IDeviceState>(DeviceState.Online);  
 builder.RegisterType<OfflineState>().Keyed<IDeviceState>(DeviceState.Offline);

### 显式解析

Resolving Explicitly

可以使用 ResolveKeyed() 显式解析键入服务：

var r = container.ResolveKeyed<IDeviceState>(DeviceState.Online);

但这会导致使用容器作为服务定位器，这是不推荐的。为了替代这种模式，Autofac 提供了 IIndex 类型。

### 使用索引解析

Resolving with an Index

Autofac.Features.Indexed.IIndex<K, V> 是 Autofac 自动实现的一种关系类型<../resolve/relationships>。根据键选择服务实现的组件可以通过接受一个类型为 IIndex<K, V> 的构造函数参数来做到这一点。

public class Modem : IHardwareDevice  
 {  
 IIndex<DeviceState, IDeviceState> \_states;  
 IDeviceState \_currentState;  
 public Modem(IIndex<DeviceState, IDeviceState> states)  
 {  
 \_states = states;  
 SwitchOn();  
 }  
 void SwitchOn()  
 {  
 \_currentState = \_states[DeviceState.Online];  
 }  
 }

在 SwitchOn() 方法中，使用索引来查找注册时使用 DeviceState.Online 键的 IDeviceState 实现。

### 使用属性解析

Resolving with Attributes

Autofac 的 [metadata特性](https://docs.autofac.org/en/latest/advanced/metadata.html) 提供了一个 KeyFilterAttribute ，允许你标记带有指定键应使用的键入服务的构造函数参数。该属性的用法如下：

public class ArtDisplay : IDisplay  
 {  
 public ArtDisplay([KeyFilter("Painting")] IArtwork art) { ... }  
 }

当你注册需要属性过滤的组件时，需要确保启用它。查询属性并执行过滤操作会带来轻微但非零的性能开销，所以它不会自动发生。

var builder = new ContainerBuilder();  
 builder.RegisterType<Painting>().Keyed<IArtwork>("Painting");  
 builder.RegisterType<ArtDisplay>().As<IDisplay>().WithAttributeFiltering();

有关如何使用属性和过滤的更多信息，请参阅 metadata文档 。

# Autofac 委托工厂

Delegate Factories

默认情况下，Autofac 支持 Func和 Func<X,Y,T>的 隐式关系。如果你需要在运行时自动创建一个用于解决某些东西的工厂，这些是最佳选择。

自动生成工厂的一个缺点是它们不支持 多个相同类型的参数 。此外，有时你可能希望提供特定的委托类型作为工厂，而不是使用 Func<X,Y,T>并希望人们总是正确地传递参数。

在这种情况下，可能需要考虑使用委托工厂。

**生命周期范围也会被尊重**，无论是在使用 Func或参数化的 Func<X,Y,T>关系时，还是使用委托工厂。如果你将一个对象注册为 InstancePerDependency()，并且多次调用委托工厂，每次都会得到一个新的实例。然而，如果你将一个对象注册为 SingleInstance()，并且多次通过委托工厂来解决该对象，无论你传递不同的参数，你都将始终获得**同一个对象实例**。只是传递不同的参数不会破坏对生命周期范围的尊重。

通过本页，我们将使用股票投资组合的例子，其中持有不同的股票，并可能需要从远程服务获取当前报价。

## 创建委托

Create a Delegate

设置委托工厂的第一步是创建一个将用于从生命周期范围动态解析值的委托。这将代替 Func或 Func<X,Y,T>的隐式关系。

这里有一个单个持股示例 - 持有一股股票和数量。消费者不再直接实例化这个类，而是使用委托工厂。

public class Shareholding  
{  
   
 public delegate Shareholding Factory(string symbol, uint holding);  
 public Shareholding(string symbol, uint holding)  
 {  
 Symbol = symbol;  
 Holding = holding;  
 }  
 public string Symbol { get; private set; }  
 public uint Holding { get; set; }  
}

Shareholding 类声明了一个构造函数，但还提供了可以间接通过从生命周期范围中解析它们来创建 Shareholding 实例的委托类型。

Autofac 可以利用这一点自动生成一个工厂：

var builder = new ContainerBuilder();  
 builder.RegisterType<Shareholding>();  
 using var container = builder.Build();  
 using var scope = container.BeginLifetimeScope();  
 var createHolding = scope.Resolve<Shareholding.Factory>();  
 var holding = createHolding("ABC", 1234);

工厂是一个标准的委托，可以通过 Invoke()或如上所示的函数语法进行调用。

**默认情况下，Autofac** **会根据名称匹配委托的参数到构造函数的参数。**如果你使用泛型 Func关系，Autofac 将切换到按类型匹配参数。名称匹配很重要 - 它允许你在需要时提供多个相同类型的参数，这是 Func隐式关系无法支持的。然而，这也意味着如果你更改构造函数参数的名称，你也必须在委托中更改这些名称。

## 使用委托

Consume the Delegate

一旦你注册了工厂，其他组件就可以消费它。

public class Portfolio  
 {  
 private readonly Shareholding.Factory \_shareHoldingFactory;  
 private readonly List<Shareholding> \_holdings = new List<Shareholding>();  
 public Portfolio(Shareholding.Factory shareholdingFactory)  
 {  
 \_shareHoldingFactory = shareholdingFactory;  
 }  
 public void Add(string symbol, uint holding)  
 {  
 \_holdings.Add(\_shareHoldingFactory(symbol, holding));  
 }  
 }

要设置此关系，Portfolio类会在构建之前用以下方式注册到容器中：

builder.RegisterType<Portfolio>();

然后你可以从生命周期范围请求 Portfolio 实例：

var portfolio = scope.Resolve<Portfolio>();  
 portfolio.Add("DEF", 4324);

## 添加已解析的构造函数参数

Add Resolved Constructor Parameters

这是使用委托工厂的主要好处：你可以添加更多从生命周期范围中解析的构造参数，而不会影响委托！想象一个远程股票报价服务：

public interface IQuoteService  
 {  
 decimal GetQuote(string symbol);  
 }

在 Shareholding 类中，我们想要通过获取报价来计算股票的当前价值。我们可以在 Shareholding 类中添加对 IQuoteService 的依赖，这将由 Autofac 自动填充 - 并且**你不必将其添加到委托工厂**！然后我们可以添加一个 CurrentValue() 方法来使用新服务。

public class Shareholding  
 {  
   
 public delegate Shareholding Factory(string symbol, uint holding);  
 private readonly IQuoteService \_quoteService;  
   
 public Shareholding(string symbol, uint holding, IQuoteService quoteService)  
 {  
 Symbol = symbol;  
 Holding = holding;  
 \_quoteService = quoteService;  
 }  
 public string Symbol { get; private set; }  
 public uint Holding { get; set; }  
 public decimal CurrentValue()  
 {  
   
 return \_quoteService.GetQuote(Symbol) \* Holding;  
 }  
 }

IQuoteService 的实现者也应注册到容器中：

builder.RegisterType<WebQuoteService>().As<IQuoteService>();

现在 Portfolio 类可以利用新的 CurrentValue() 方法，而不了解任何关于报价服务的事情。

public class Portfolio  
 {  
 private readonly Shareholding.Factory \_shareHoldingFactory;  
 private readonly List<Shareholding> \_holdings = new List<Shareholding>();  
 public Portfolio(Shareholding.Factory shareholdingFactory)  
 {  
 \_shareHoldingFactory = shareholdingFactory;  
 }  
 public void Add(string symbol, uint holding)  
 {  
   
 \_holdings.Add(\_shareHoldingFactory(symbol, holding));  
 }  
 public decimal CurrentValue()  
 {  
   
 return \_holdings.Aggregate(0m, (agg, holding) => agg + holding.CurrentValue());  
 }  
 }

## 生命周期范围和释放

Lifetime Scopes and Disposal

就像使用Func关系或直接调用Resolve()一样，使用委托工厂是从生命周期范围中解决某个东西。如果你解决的是可释放的（disposable），那么生命周期范围将跟踪它，并在范围释放时释放它<../lifetime/disposal>。如果直接从容器或在使用可释放组件的长时间存在的生命周期范围中解决，可能会导致内存泄漏，因为范围持有所有已解决的可释放组件的引用。

## 注册 GeneratedFactory（已废弃）

RegisterGeneratedFactory (Obsolete)

**重要** 自 Autofac 7.0 开始，*RegisterGeneratedFactory* 现在标记为废弃。委托工厂和函数关系已经取代了这个功能。

处理松散耦合场景的旧方法是通过使用 RegisterGeneratedFactory() 。这与委托工厂的工作方式非常相似，但需要显式的注册操作。

public delegate DuplicateTypes FactoryDelegate(int a, int b, string c);

然后使用 RegisterGeneratedFactory() 注册该委托：

builder.RegisterType<DuplicateTypes>();  
 builder.RegisterGeneratedFactory<FactoryDelegate>(new TypedService(typeof(DuplicateTypes)));

现在函数可以工作：

var func = scope.Resolve<FactoryDelegate>();  
 var obj = func(1, 2, "three");

# Autofac 自有实例

Owned Instances

## 生命周期和作用域

Lifetime and Scope

Autofac 使用明确界定的范围来控制生命周期。例如，提供 S 服务及其所有依赖项的组件将在 using 块结束时被释放：

IContainer container =   
 using (var scope = container.BeginLifetimeScope())  
 {  
 var s = scope.Resolve<S>();  
 s.DoSomething();  
 }

在 *IoC* 容器中，释放和销毁组件之间通常存在微妙的区别：释放自有组件不仅仅是销毁组件本身。组件的所有依赖项也会被销毁。释放共享组件通常是无操作的，因为其他组件将继续使用其服务。

## 关系类型

Autofac 有一个名为 关系类型 的系统，可以用于以声明性方式提供容器的功能。与上面的例子不同，关系类型允许组件以最小的、声明性的方式精确指定所需的容器服务。

**自有实例** 通过 Owned关系类型进行消耗。

### 自有类型 T

当不再需要时，拥有者可以释放一个自有依赖项。自有依赖项通常对应于依赖组件执行的一些工作单元。

public class Consumer  
 {  
 private Owned<DisposableComponent> \_service;  
 public Consumer(Owned<DisposableComponent> service)  
 {  
 \_service = service;  
 }  
 public void DoWork()  
 {  
   
 \_service.Value.DoSomething();  
 \_service.Dispose();  
 }  
 }

当 Consumer 由容器创建时，它所依赖的 Owned 将在其自身的生命周期范围内创建。当 Consumer 完成使用 DisposableComponent 时，释放 Owned 引用将结束包含 DisposableComponent 的生命周期范围。这意味着 DisposableComponent的所有非共享、可丢弃的依赖项也将被释放。

### 与 Func 结合使用

通常会将自有实例与Func关系结合使用，以便可以根据需要动态开始和结束工作单元。

interface IMessageHandler  
 {  
 void Handle(Message message);  
 }  
 class MessagePump  
 {  
 Func<Owned<IMessageHandler>> \_handlerFactory;  
 public MessagePump(Func<Owned<IMessageHandler>> handlerFactory)  
 {  
 \_handlerFactory = handlerFactory;  
 }  
 public void Go()  
 {  
 while (true)  
 {  
 var message = NextMessage();  
 using (var handler = \_handlerFactory())  
 {  
 handler.Value.Handle(message);  
 }  
 }  
 }  
 }

### 自有和标签

Owned 创建的生命周期使用作为 ILifetimeScope.Tag 存在的标记功能。Owned 生命周期应用的标签将是 new TypedService(typeof(T)) ，即生命周期的标签反映了其入口点。

# Autofac 池化实例

Pooled Instances

通常，应用程序会发现它们有昂贵的初始化组件（比如数据库或某种外部服务连接），而你更希望复用这些实例，而不是每次需要时都创建新的。

这通常被称为维护一个“对象池”。当你想要一个对象的新实例时，从池中获取一个，使用完毕后，再将其返回到池中。

Autofac 可以帮助你在应用程序中实现对象池，而无需自己编写池化实现，让这些池化的组件在依赖注入的世界中感觉更加自然。

**注意**  
 *在继续之前，值得一提的是，许多.NET 标准类型（如* *HttpClient* *或 ADO.NET 的* *SqlConnection* *）已经在幕后为你实现了池化，因此在这些类型上添加 Autofac 池化没有任何益处。*

## 开始使用

要开始创建 Autofac 的池化注册，首先请添加对 [Autofac.Pooling NuGet 包](https://nuget.org/packages/Autofac.Pooling) 的引用。

有了这个，你可以开始定义和使用池化注册，使用新的生命周期配置方法 PooledInstancePerLifetimeScope 和 PooledInstancePerMatchingLifetimeScope：

var builder = new ContainerBuilder();  
 builder.RegisterType<MyCustomConnection>()  
 .As<ICustomConnection>()  
 .PooledInstancePerLifetimeScope();  
 var container = builder.Build();  
 using (var scope = container.BeginLifetimeScope())  
 {  
   
 var instance = scope.Resolve<ICustomConnection>();  
 instance.DoSomething();  
 }  
 using (var scope2 = container.BeginLifetimeScope())  
 {  
   
 var instance = scope.Resolve<ICustomConnection>();  
 instance.DoSomething();  
 }

像任何其他依赖项一样，你可以使用这些服务在构造函数中注入池化的实例：

public class WorkOperation  
 {  
   
 public WorkOperation(ICustomConnection customConnection)  
 {  
   
 }  
 }

当前的 生命周期范围 结束时，从池中获取的实例将返回到池中。

## 在解析之间重置池化实例

对于池化的组件，通常需要在从池中获取或返回到池时进行一些工作来重置对象。

Autofac 允许组件在从池中获取或返回到池时知道这一点，通过实现 IPooledComponent 接口：

public class PoolAwareComponent : IPooledComponent  
 {  
 public void OnGetFromPool(IComponentContext context, IEnumerable<Parameter> parameters)  
 {  
   
 }  
 public void OnReturnToPool()  
 {  
   
 }  
 }

OnGetFromPool 方法会传递当前解析操作的临时 IComponentContext ，以及传递给解析的所有参数。

**警告**

从提供的 IComponentContext 中解析的任何服务将来自 **当前访问池化组件的作用域**。这意味着在 OnReturnToPool 中应该丢弃从该 IComponentContext 中解析的任何实例，以防止内存泄漏。

如果你不能修改你要池化的组件，但需要类似这样的自定义行为，你可以 [实现自定义池策略](https://www.yuque.com/legege007/csharp/dwt7z4hzolkukval#Top_of_s29_xhtml) 。

## 池容量

每个池化注册都有池容量的概念。默认值为 Environment.ProcessorCount \* 2，但可以很容易地使用扩展方法的重载进行自定义：

builder.RegisterType<MyCustomConnection>()  
 .As<ICustomConnection>()  
 .PooledInstancePerLifetimeScope(100);

**重要的是要理解池的容量不会限制它分配/激活的实例数量，或者在任何时候可以使用的实例数量；相反，它限制了池保留的实例数量。**

在实际情况下，这意味着如果你的池容量为 100，而你当前有 100 个实例在使用，那么解析另一个实例将只是激活组件的新实例，而不是阻塞或失败。

然而，如果你有 101 个该组件的实例在使用，下一个返回到池中的实例将被丢弃，而不是保留。在这种情况下，IPooledComponent上的 OnReturnToPool方法仍然会被调用，然后实例会被立即丢弃。

当池丢弃一个实例时，如果对象实现了 IDisposable，Dispose将被调用。

如果你确实希望你的池具有阻止资源可用直到获得的行为，你可以 [实现自定义池策略](https://www.yuque.com/legege007/csharp/dwt7z4hzolkukval#Top_of_s29_xhtml)。

**注意:**

* Autofac 池化行为建立在 [Microsoft.Extensions.ObjectPool](https://www.koudingke.cn/go?link=https%3a%2f%2fdocs.microsoft.com%2fen-us%2faspnet%2fcore%2fperformance%2fobjectpool) 库提供的[对象池](https://www.koudingke.cn/go?link=https%3a%2f%2fdocs.microsoft.com%2fen-us%2faspnet%2fcore%2fperformance%2fobjectpool) 实现之上。
* 该池的行为影响了许多 Autofac.Pooling 的行为。

## 匹配生命周期范围

与配置常规注册以 匹配生命周期范围 的方式相同，也可以配置池化注册以以相同的方式进行匹配：

builder.RegisterType<MyCustomConnection>()  
 .As<ICustomConnection>()  
 .PooledInstancePerMatchingLifetimeScope("tag");

具有匹配生命周期范围的池化注册会导致每个标记的范围从池中获取自己的实例，子范围共享相同的池化实例。

当标记的生命周期范围被丢弃时，实例将返回到池中。

## 池策略

如果你需要在实例从池中获取或返回到池时执行一些自定义行为，你可以实现 IPooledRegistrationPolicy ，或者覆盖 DefaultPooledRegistrationPolicy 。

以下是一个简单的策略示例，它会在可用容量耗尽时阻止进一步请求池化的实例：

public class BlockingPolicy<TPooledObject> : IPooledRegistrationPolicy<TPooledObject>  
 where TPooledObject : class  
 {  
 private readonly SemaphoreSlim \_semaphore;  
 public BlockingPolicy(int maxConcurrentInstances)  
 {  
   
 \_semaphore = new SemaphoreSlim(maxConcurrentInstances);  
   
 MaximumRetained = maxConcurrentInstances;  
 }  
   
 public int MaximumRetained { get; }  
   
 public TPooledObject Get(IComponentContext context, IEnumerable<Parameter> parameters, Func<TPooledObject> getFromPool)  
 {   
 \_semaphore.Wait();  
   
 return getFromPool();  
 }  
   
 public bool Return(TPooledObject pooledObject)  
 {   
 \_semaphore.Release();  
 return true;  
 }  
 }

然后，你可以在注册池时使用此策略：

builder.RegisterType<MyCustomConnection>()  
 .As<ICustomConnection>()  
 .PooledInstancePerLifetimeScope(new BlockingPolicy<MyCustomConnection>(100));

# Autofac 自定义构造函数选择

Custom Constructor Selection

大多数情况下，在 注册反射组件 时，可以安全地将选择正确的构造函数的工作交给 Autofac，或者如果需要的话，可以通过 UsingConstructor在注册中指定一个明确的构造函数。

对于高级用例，你可以实现自定义行为来选择类型可用的构造函数集以及使用其中的哪个构造函数。

## FindConstructorsWith & IConstructorFinder

注册上的 FindConstructorsWith 方法允许你指定如何确定注册的**可用**构造函数集。这可以通过委托从 Type 中检索构造函数实现：

builder.RegisterType<ComponentWithInternalConstructors>()  
 .FindConstructorsWith(type => type.GetDeclaredConstructors());

或者通过实现 IConstructorFinder 接口，这使得更容易为了性能考虑缓存找到的构造函数：

public class AllConstructorsFinder : IConstructorFinder  
 {  
 private static readonly ConcurrentDictionary<Type, ConstructorInfo[]> ConstructorCache = new();  
 public ConstructorInfo[] FindConstructors(Type targetType)  
 {  
 var retval = ConstructorCache.GetOrAdd(targetType, t => t.GetDeclaredConstructors());  
 if (retval.Length == 0)  
 {  
 throw new NoConstructorsFoundException(targetType);  
 }  
 return retval;  
 }  
 }  
 builder.RegisterType<ComponentWithInternalConstructors>()  
 .FindConstructorsWith(new AllConstructorsFinder());

**注意**  
 *对于泛型注册，传递给* *FindConstructorsWith* *委托或* *IConstructorFinder* *的* *Type* *将是具体\类型，而不是泛型。*

## IConstructorSelector

一旦确定了可用的构造函数集，每次组件被解析时，必须从中选择一个构造函数。

如果只有一个可用构造函数，我们只需使用它；但如果有多于一个可用的构造函数，我们需要决定哪个构造函数最合适。

为此，我们可以实现 IConstructorSelector 接口。Autofac 的此接口的默认实现（ MostParametersConstructorSelector ）会选择在解析时可以从容器获取参数最多的构造函数。

当默认的 Autofac 行为不适合时，你可以使用自定义 IConstructorSelector 实现。

这里是一个抽象示例，说明了一个允许参数强制使用“第一个”构造函数的构造函数选择器。

public class FirstConstructorOverrideSelector : IConstructorSelector  
 {  
 private IConstructorSelector \_autofacDefault = new MostParametersConstructorSelector();  
 public BoundConstructor SelectConstructorBinding(BoundConstructor[] constructorBindings, IEnumerable<Parameter> parameters)  
 {  
 if (parameters.Any(x => x is ConstantParameter p && string.Equals(p.Value, "use-first")))  
 {  
 return constructorBindings.First();  
 }  
 return \_autofacDefault.SelectConstructorBinding(constructorBindings, parameters);  
 }  
 }

然后，你可以在组件上注册该选择器：

builder.RegisterType<MyComponent>()  
 .UsingConstructor(new FirstConstructorOverrideSelector());

**注意:**  
 IConstructorSelector 实现仅在给定组件具有多个可用构造函数时才会被调用。

# Autofac 处理并发

Handling Concurrency

Autofac 是为高度并发应用设计的。下面的指导将帮助你在这些场景中取得成功。

## 组件注册

ContainerBuilder 和 ComponentRegistryBuilder**不是线程安全的** ，并且仅在应用程序启动时使用单个线程。这是最常见的场景，适用于大多数应用。

## 服务解析

**所有容器操作在多个线程之间是安全的。**

为了减少锁定开销，每个 Resolve操作都在一个提供了容器依赖解决功能的 "上下文" 中进行。这是传递给组件注册委托的参数。

**上下文对象是单线程的** ，除依赖解析操作期间外，**不应** 使用它们。

避免将上下文存储在组件注册中：

builder.Register(c => new MyComponent(c));

在上述示例中，将 "c" 参数（类型为 IComponentContext）提供给了 MyComponent（它依赖于 IComponent）。这段代码是不正确的，因为临时的 "c" 参数会被重用。

相反，从 "c" 中获取 IComponentContext 来访问非临时上下文：

builder.Register(c =>  
 {  
 IContext threadSpecificContext = c.Resolve<IComponentContext>();   
 return new MyComponent(threadSpecificContext);  
 }

还要注意不要初始化具有对 "c" 参数闭包的组件，因为任何 "c" 的重用都会导致问题。

容器层次结构机制进一步减少了锁定，通过维护任何工厂/容器组件的局部注册副本。一旦初始注册副本创建完毕，使用 "内" 容器的线程可以创建或访问此类组件，而不会阻塞其他线程。

## 生命周期事件

当使用可用的生命周期事件时，请不要在 Preparing、Activating 或 Activated 事件处理器中回调到容器：而是使用提供的 IComponentContext。

## 线程特定的服务

你可以使用 Autofac 注册特定于线程的服务。有关更多信息，请参阅 实例生命周期范围 页面。

## 内部机制

考虑到上述指导原则，这里提供了一些关于 Autofac 中线程安全和锁定的更具体信息。

## 线程安全类型

以下类型对多个线程并发访问是安全的：

* Container
* Disposer（IDisposer 的默认实现）
* LifetimeScope（ILifetimeScope 的默认实现）

这些类型涵盖了几乎所有的运行时/解析场景。

以下类型旨在在配置时间用于单线程访问：

* ContainerBuilder
* ComponentRegistryBuilder（IComponentRegistryBuilder 的默认实现）

因此，正确的 Autofac 应用程序会在启动时使用单个线程上的 ContainerBuilder 创建容器。后续使用容器可以在任何线程上进行。

## 避免死锁

Autofac 的设计方式使得在正常使用情况下不会发生死锁。这部分指南是为维护者或扩展编写者准备的。

可能会获得以下顺序的锁：

* 拥有以下任何锁的线程可能无法获取更多锁：
* 拥有 LifetimeScope 锁的线程随后可以获取：

o 其父 LifetimeScope 的锁

o 上述列出的任何项目

# Autofac 多租户应用

Multitenant Applications

Autofac.Multitenant 提供了多租户依赖注入支持。（在 v4.0.0 之前，该包的名称为 Autofac.Extras.Multitenant。）

## 什么是多租户

**多租户应用** 是一种部署一次但允许多个客户（或“租户”）以自己的方式访问应用的应用程序。

例如，考虑一个托管的在线商店应用——你作为“租户”，租赁该应用，设置一些配置值。当用户使用自定义域名访问应用时，它看起来就像你的公司。其他租户也可能租赁该应用，但应用只在一个中央托管服务器上部署一次，并根据访问它的租户（或租户的最终用户）进行行为调整。

多租户环境中的许多更改都是通过简单的配置来完成的。例如，UI 显示的颜色或字体是可以通过简单配置插件实现的，而无需实际改变应用的行为。

**在更复杂的场景中，你可能需要根据租户进行业务逻辑的更改。** **例如，特定租户可能希望使用一些复杂自定义逻辑来更改某个值的计算方式。如何为应用注册默认行为/依赖项，并允许特定租户覆盖它？**

这就是 Autofac.Multitenant力求解决的问题。

## 一般原则

一般来说，多租户应用在依赖解析方面需要执行以下四个任务：

**1.** **引用 NuGet 包**

**2.** **注册依赖项**

**3.** **租户识别**

**4.** **解析依赖项**

本节概述了这四个步骤的工作原理。后续章节将扩展这些主题，包括如何将这些原则与特定应用类型集成。

### 引用 NuGet 包

任何想要使用多租户功能的应用都需要添加对以下 NuGet 包的引用...

* Autofac
* Autofac.Multitenant

对于 WCF 应用，还需要 Autofac.Multitenant.Wcf。

### 注册依赖项

Autofac.Multitenant 引入了一个名为 Autofac.Multitenant.MultitenantContainer 的新容器类型。这个容器用于管理应用级别的默认设置和特定租户的覆盖。

整体注册过程如下：

1. **创建应用级别的默认容器。** 这个容器用于注册应用的默认依赖项。如果租户没有为依赖项类型提供其他覆盖，则这里注册的依赖项将被使用。
2. **实例化租户标识策略。** 租户标识策略用于根据执行上下文确定当前租户的 ID。关于这一点稍后会详细介绍。
3. **创建多租户容器。** 多租户容器负责跟踪应用默认设置和特定租户的覆盖。
4. **注册特定租户的覆盖。** 对于每个希望覆盖依赖项的租户，通过传递租户 ID 和配置 Lambda 注册适当的覆盖。

通用用法如下所示：

var builder = new ContainerBuilder();  
 builder.RegisterType<Consumer>().As<IDependencyConsumer>().InstancePerDependency();  
 builder.RegisterType<BaseDependency>().As<IDependency>().SingleInstance();  
 var appContainer = builder.Build();  
 var tenantIdentifier = new MyTenantIdentificationStrategy();  
 var mtc = new MultitenantContainer(tenantIdentifier, appContainer);  
 mtc.ConfigureTenant('1', b => b.RegisterType<Tenant1Dependency>().As<IDependency>().InstancePerDependency());  
 mtc.ConfigureTenant('2', b => b.RegisterType<Tenant2Dependency>().As<IDependency>().SingleInstance());

**如果你有一个需要为每个租户提供一个实例的组件，** **可以在容器级别使用** **InstancePerTenant()** **注册扩展方法。**

var builder = new ContainerBuilder();  
 builder.RegisterType<SomeType>().As<ISomeInterface>().InstancePerTenant();

注意，**你只能为租户配置一次。** 之后，你不能再更改租户的覆盖。此外，如果为租户解析依赖项，他们的生命周期范围可能不会更改。在应用启动时配置租户覆盖是一个好习惯，以避免出现任何问题。如果需要根据业务逻辑“构建”租户配置，可以使用 Autofac.Multitenant.ConfigurationActionBuilder。

var builder = new ContainerBuilder();  
 var appContainer = builder.Build();  
 var tenantIdentifier = new MyTenantIdentificationStrategy();  
 var mtc = new MultitenantContainer(tenantIdentifier, appContainer);  
 var actionBuilder = new ConfigurationActionBuilder();  
 if(SomethingIsTrue())  
 {  
 actionBuilder.Add(b => b.RegisterType<AnOverride>().As<ISomething>());  
 }  
 actionBuilder.Add(b => b.RegisterType<SomeClass>());  
 if(AnotherThingIsTrue())  
 {  
 actionBuilder.Add(b => b.RegisterModule<MyModule>());  
 }  
 mtc.ConfigureTenant('1', actionBuilder.Build());

### 识别租户

为了解析特定租户的依赖项，Autofac 需要知道哪个租户正在发出解析请求。也就是说，“对于当前执行上下文，哪个租户正在解析依赖项？”

Autofac.Multitenant 包含一个 ITenantIdentificationStrategy 接口，你可以实现它来提供这样的机制。这允许你在应用程序中适当的地方获取租户 ID：环境变量、当前用户角色、传入请求值，或其他任何地方。

以下是一个简单的 Web 应用可能使用的示例 ITenantIdentificationStrategy。

using System;  
 using System.Web;  
 using Autofac.Multitenant;  
 namespace DemoNamespace  
 {  
   
   
   
 public class RequestParameterStrategy : ITenantIdentificationStrategy  
 {  
 public bool TryIdentifyTenant(out object tenantId)  
 {  
 tenantId = null;  
 try  
 {  
 var context = HttpContext.Current;  
 if(context != null && context.Request != null)  
 {  
 tenantId = context.Request.Params["tenant"];  
 }  
 }  
 catch(HttpException)  
 {  
   
 }  
 return tenantId != null;  
 }  
 }  
 }

在这个示例中，一个 Web 应用程序使用传入的请求参数来获取租户 ID。（请注意，这只是示例，不推荐使用，因为它会允许系统上的任何用户非常容易地切换租户。它也没有处理发生在 Web 请求之外的情况。）

在自定义策略实现中，你可能会选择将租户 ID 表示为 GUID、整数或其他自定义类型。这里的策略是将值从执行上下文解析为强类型对象，并根据值是否存在以及是否可以解析为适当类型来成功/失败。

Autofac.Multitenant 使用 System.Object 作为整个系统的租户 ID 类型，以最大限度地提高灵活性。

**在租户识别中性能很重要。** 每次解析组件、开始新的生命周期范围等时都会进行租户识别。因此，确保租户识别策略快速非常重要。例如，你不希望在租户识别期间进行服务调用或数据库查询。

**在租户识别中要妥善处理错误。**特别是在 ASP.NET 应用程序启动等情况下，你可能使用某种上下文机制（如 HttpContext.Current.Request）来确定租户 ID，但如果在租户 ID 策略被调用时该上下文信息不可用，你需要能够处理这种情况。在上面的示例中，它不仅检查当前 HttpContext，还检查 Request。检查所有内容并处理异常（如解析异常），否则你可能会遇到难以调试的奇怪行为。

**对于你的租户 ID 策略还有更多提示**，请参阅 tenant*id*strategy\_tips 部分。

### 解析特定租户的依赖项

MultitenantContainer 工作方式是，系统中的每个租户都有自己的 Autofac.ILifetimeScope 实例，其中包含应用默认设置和特定租户的覆盖。这样做...

var builder = new ContainerBuilder();  
 builder.RegisterType<BaseDependency>().As<IDependency>().SingleInstance();  
 var appContainer = builder.Build();  
 var tenantIdentifier = new MyTenantIdentificationStrategy();  
 var mtc = new MultitenantContainer(tenantIdentifier, appContainer);  
 mtc.ConfigureTenant('1', b => b.RegisterType<Tenant1Dependency>().As<IDependency>().InstancePerDependency());

这非常类似于使用标准的 ILifetimeScope.BeginLifetimeScope(Action)，如下所示：

var builder = new ContainerBuilder();  
 builder.RegisterType<BaseDependency>().As<IDependency>().SingleInstance();  
 var appContainer = builder.Build();  
 using(var scope = appContainer.BeginLifetimeScope(  
 b => b.RegisterType<Tenant1Dependency>().As<IDependency>().InstancePerDependency())  
 {  
   
 }

当你使用 MultitenantContainer 来解析依赖项时，它会在幕后调用你的 ITenantIdentificationStrategy 来识别租户，找到具有配置覆盖的租户的生命周期范围，并从该范围中解析依赖项。所有这些都在透明的方式下完成，所以你可以像使用其他容器一样使用多租户容器。

var dependency = mtc.Resolve<IDependency>();

这里的关键在于，所有的工作都是在幕后透明地进行的。对 Resolve、BeginLifetimeScope、Tag、Disposer 或 IContainer 接口上的其他方法/属性的任何调用都会经过租户识别过程，并且结果将是特定于租户的。

如果你需要直接访问租户的生命周期范围或应用程序容器，MultitenantContainer 提供了以下功能：

* ApplicationContainer：获取应用程序容器。
* GetCurrentTenantScope：识别当前租户并返回他们的特定生命周期范围。
* GetTenantScope：允许你提供一个特定的租户 ID，以便获取该租户的生命周期范围。

## ASP.NET 集成

ASP.NET 集成与其他 标准 ASP.NET 应用集成 并没有太大不同。真正不同的是，你需要设置应用程序的 Autofac.Integration.Web.IContainerProvider 或 System.Web.Mvc.IDependencyResolver 或其他什么，使用 Autofac.Multitenant.MultitenantContainer 而不是由 ContainerBuilder 构建的标准容器。由于 MultitenantContainer 以透明的方式处理多租户，所以 “一切正常” 。

### ASP.NET 应用程序启动

以下是一个 ASP.NET MVC 的 Global.asax 示例，说明了它是多么简单：

namespace MultitenantExample.MvcApplication  
 {  
 public class MvcApplication : HttpApplication  
 {  
 public static void RegisterRoutes(RouteCollection routes)  
 {  
   
 }  
 protected void Application\_Start()  
 {  
   
 var tenantIdStrategy = new RequestParameterTenantIdentificationStrategy("tenant");  
 var builder = new ContainerBuilder();  
 builder.RegisterType<BaseDependency>().As<IDependency>();  
   
 builder.RegisterType<HomeController>();  
   
 var mtc = new MultitenantContainer(tenantIdStrategy, builder.Build());  
 mtc.ConfigureTenant("1",  
 b =>  
 {  
 b.RegisterType<Tenant1Dependency>().As<IDependency>().InstancePerDependency();  
 b.RegisterType<Tenant1Controller>().As<HomeController>();  
 });  
   
 DependencyResolver.SetResolver(new AutofacDependencyResolver(mtc));  
   
 AreaRegistration.RegisterAllAreas();  
 RegisterRoutes(RouteTable.Routes);  
 }  
 }  
 }

如你所见，**它几乎就像常规 MVC Autofac 集成一样**。你设置应用程序容器、租户 ID 策略、多租户容器和租户覆盖，就像前面在 register*dependencies 和 tenant*identification 中所示。然后当你设置 DependencyResolver 时，给它一个多租户容器。其他一切都正常工作。

**对于其他 Web 应用，这种相似性仍然适用**。当设置 Web 表单的 IContainerProviderAccessor 时，使用多租户容器而不是标准容器。当设置 Web API 的 DependencyResolver 时，使用多租户容器而不是标准容器。

请注意，在示例中，控制器是分开注册的，而不是使用一次注册所有控制器的 builder.RegisterControllers(Assembly.GetExecutingAssembly()); 风格。请参阅下面的内容了解原因。

### 特定于租户的控制器

在 MVC 应用中，你可能会选择让租户覆盖控制器。这是可能的，但需要一些考虑。

首先，**特定于租户的控制器必须继承它们要覆盖的控制器。**例如，如果你有一个 HomeController，而一个租户想要创建他们自己的实现，他们需要从它派生，如下所示...

public class Tenant1HomeController : HomeController  
 {  
   
 }

其次，**如果你的应用程序控制器和特定于租户的控制器在同一命名空间中，你不能一次性注册所有的控制器**。你可能在标准 ASP.NET MVC 集成 中看到过一行 builder.RegisterControllers(Assembly.GetExecutingAssembly());，用来一次性注册命名空间中的所有控制器。不幸的是，如果在这个命名空间中有特定于租户的控制器，那么如果你这样做，它们都将被注册为应用程序级别。相反，你需要逐个注册每个应用程序控制器，然后以相同的方式配置租户特定的覆盖。

上面的 Global.asax 示例展示了这种逐个注册控制器的模式。

当然，如果你将特定于租户的控制器保存在其他命名空间中，你可以一次性使用 builder.RegisterControllers(Assembly.GetExecutingAssembly()); 注册所有应用程序控制器，而且它会正常工作。请注意，如果特定于租户的控制器库没有被主应用程序引用（例如，它们是动态注册的插件，使用装配探测或其他方式）你需要使用 ASP.NET BuildManager 来注册它们。

最后，当注册特定于租户的控制器时，应将它们“注册为”基控制器类型。在上面的示例中，可以看到默认控制器在应用程序容器中像这样注册：

var builder = new ContainerBuilder();  
 builder.RegisterType<HomeController>();

然后当租户在他们的租户配置中覆盖控制器时，看起来像这样：

var mtc = new MultitenantContainer(tenantIdStrategy, builder.Build());  
 mtc.ConfigureTenant("1", b => b.RegisterType<Tenant1Controller>().As<HomeController>());

**由于这个相对复杂性，可能更好的做法是将业务逻辑隔离到外部依赖项中，这些依赖项会传递到控制器，以便租户可以提供替代依赖项，而不是覆盖控制器。**

## ASP.NET Core 集成

ASP.NET Core 改变了许多东西。多租户集成已在 我们的 ASP.NET Core 集成页面上进行了概述 。

## WCF 集成

WCF 集成与 标准 WCF 集成 稍有不同，因为你需要使用不同的服务主机工厂，而且需要一些额外的配置。

此外，识别租户也更难——客户端需要以某种方式向服务传递租户 ID，而服务需要知道如何解释传递的租户 ID。Autofac.Multitenant为此提供了一个简单的解决方案，即在消息头中传递相关信息的行为。

### WCF 集成的参考包

**对于消费多租户服务（客户端应用），请添加引用到...**

* Autofac
* Autofac.Multitenant

**对于提供多租户服务（服务应用），请添加引用到...**

* Autofac
* Autofac.Integration.Wcf
* Autofac.Multitenant
* Autofac.Multitenant.Wcf

### 通过行为传递租户 ID

如前所述（tenant\_identification），为了使多租户工作，你必须确定哪个租户正在执行某个调用，以便你可以解析适当的依赖项。在服务环境中，一个挑战是租户通常在客户端应用程序端建立，而需要将租户 ID 传播到服务，以便服务能够正确行事。

解决这个问题的一个常见方案是在消息头中传播租户 ID。客户端在发出的消息中添加一个特殊的头，其中包含租户 ID。服务解析该头，读取租户 ID，然后使用该 ID 来确定其功能。

在 WCF 中，将这些“动态”头附加到消息并在接收时读取它们的方式是通过行为。你在客户端和服务端都应用这种行为，以便使用相同的头信息（类型、URN 等）。

Autofac.Multitenant 提供了简单的租户 ID 传播行为，即 Autofac.Multitenant.Wcf.TenantPropagationBehavior 。在客户端侧使用时，它使用租户 ID 策略检索上下文租户 ID，并将其插入到传出消息的消息头中。在服务器端使用时，它查找此入站头并解析租户 ID，将其放入操作上下文扩展中。

wcf\_startup部分显示了在客户端和服务端应用此行为的示例。

如果你使用此行为，将提供一个相应的服务器端租户识别策略。请参阅 operationcontext\_id，下面的内容。

### 从 OperationContext 获取租户标识

无论你选择是否使用提供的 Autofac.Multitenant.Wcf.TenantPropagationBehavior在消息头中从客户端传播到服务器（参见上方的 behavior\_id），在整个操作生命周期中存储租户 ID 的好地方是 OperationContext。

Autofac.Multitenant.Wcf 为此提供了 Autofac.Multitenant.Wcf.TenantIdentificationContextExtension ，作为 WCF OperationContext 的扩展。

在操作生命周期的早期阶段（通常在 System.ServiceModel.Dispatcher.IDispatchMessageInspector.AfterReceiveRequest() 实现中），你可以将 TenantIdentificationContextExtension 添加到当前 OperationContext，以便轻松识别租户。下面的示例 AfterReceiveRequest() 实现展示了这一点：

public object AfterReceiveRequest(ref Message request, IClientChannel channel, InstanceContext instanceContext)  
 {  
   
 var tenantId = request.Headers.GetHeader<TTenantId>(TenantHeaderName, TenantHeaderNamespace);  
   
 OperationContext.Current.Extensions.Add(new TenantIdentificationContextExtension() { TenantId = tenantId });  
 return null;  
 }

一旦将租户 ID 附加到上下文，你可以根据需要使用适当的 ITenantIdentificationStrategy 来检索它。

**如果你使用** **TenantIdentificationContextExtension，那么提供的** **Autofac.Multitenant.Wcf.OperationContextTenantIdentificationStrategy** **会自动工作，从** **OperationContext** **中获取租户 ID。**

### 部署多租户服务

在 WCF 服务应用程序中，服务实现可能是特定于租户的，但共享相同的 Service Contract。这允许你将服务合同分离到特定于租户的开发人员可使用的单独程序集中，并允许他们实现自定义逻辑，而无需共享默认实现的任何内部细节。

为了实现这一点，已经实现了多租户服务定位的自定义策略：Autofac.Multitenant.Wcf.MultitenantServiceImplementationDataProvider。

在你的服务的 .svc 文件中，必须指定以下内容：

* **服务合同接口的完整类型名称。**在常规 WCF 集成 中，Autofac 允许你使用类型或命名服务。对于多租户，你必须使用基于服务合同接口的类型服务。
* **Autofac 主机工厂的完整类型名称。**这可以让托管环境知道要使用哪个工厂。（这就像标准的 Autofac WCF 集成 一样。）

一个示例 .svc文件如下所示：

<%@ ServiceHost  
 Service="MultitenantExample.WcfService.IMultitenantService, MultitenantExample.WcfService"  
 Factory="Autofac.Integration.Wcf.AutofacServiceHostFactory, Autofac.Integration.Wcf" %>

当使用 Autofac 容器注册服务实现时，必须像这样将实现注册为合同接口：

builder.RegisterType<BaseImplementation>().As<IMultitenantService>();

然后，可以使用接口类型注册特定于租户的实现：

mtc.ConfigureTenant("1", b => b.RegisterType<Tenant1Implementation>().As<IMultitenantService>());

别忘了在应用程序启动时，在设置容器的附近，你需要告诉 Autofac 你在做多租户：

AutofacHostFactory.ServiceImplementationDataProvider =  
 new MultitenantServiceImplementationDataProvider();

#### 管理服务属性

在 XML 配置（如 web.config）中配置 WCF 服务时，WCF 会自动推断期望的服务元素名称，该名称来自具体的实现类型。例如，在单租户实现中，你的 MyNamespace.IMyService 服务接口可能有一个名为 MyNamespace.MyService 的实现，这就是 WCF 在 web.config 中期望查找的：

<system.serviceModel>  
 <services>  
 <service name="MyNamespace.MyService" />  
 </services>  
 </system.serviceModel>

然而，使用多租户服务主机时，实现接口的具体服务类型是一个动态生成的代理类型，因此服务配置名称变为自动生成的类型名称，如下所示：

<system.serviceModel>  
 <services>  
 <service name="Castle.Proxies.IMyService\_1" />  
 </services>  
</system.serviceModel>

为了简化这个过程，Autofac.Multitenant.Wcf 提供了 Autofac.Multitenant.Wcf.ServiceMetadataTypeAttribute ，你可以使用它来创建一个“元数据伙伴类”（类似于 System.ComponentModel.DataAnnotations.MetadataTypeAttribute ），并标记它以添加类型级别的元数据，从而修改动态代理的行为。

在这种情况下，你需要动态代理具有 System.ServiceModel.ServiceBehaviorAttribute ，以便定义预期的 ConfigurationName 。

首先，将服务接口标记为 ServiceMetadataTypeAttribute ：

using System;  
 using System.ServiceModel;  
 using Autofac.Multitenant.Wcf;  
 namespace MyNamespace  
 {  
 [ServiceContract]  
 [ServiceMetadataType(typeof(MyServiceBuddyClass))]  
 public interface IMyService  
 {  
   
 }  
 }

接下来，创建在属性中指定的伙伴类，并添加适当的元数据。

using System;  
 using System.ServiceModel;  
 namespace MyNamespace  
 {  
 [ServiceBehavior(ConfigurationName = "MyNamespace.IMyService")]  
 public class MyServiceBuddyClass  
 {  
 }  
 }

现在，你可以在 XML 配置文件中使用在伙伴类上指定的配置名称：

<system.serviceModel>  
 <services>  
 <service name="MyNamespace.IMyService" />  
 </services>  
</system.serviceModel>

**关于元数据的重要注意事项：**

* **只有类型级别的元数据被复制。**目前，仅复制伙伴类上的类型级别的元数据到动态代理。如果你有在属性级别或方法级别复制元数据的需求，请提交问题。
* **并非所有元数据都会产生预期的效果。**例如，如果你使用 ServiceBehaviorAttribute来定义与寿命相关的信息，如 InstanceContextMode，则服务不会遵循该指令，因为 Autofac 正在管理寿命，而不是标准的服务主机。指定元数据时要谨慎——如果不起作用，别忘了你没有使用标准的服务寿命管理功能。
* **元数据是应用程序级的，不是按租户的。**元数据伙伴类的信息将在应用程序级别生效，不能被每个租户覆盖。

#### 特定于租户的服务实现

如果你正在部署多租户服务（ hosting），你可以提供特定于租户的服务实现。这允许你提供服务的基础实现，并与租户共享服务合同，以便他们可以开发自定义服务实现。

**你必须将服务合同实现为单独的接口。**你不能将服务实现标记为 ServiceContractAttribute。然后，服务实现必须实现该接口。这通常也是好习惯，但是多租户服务主机不允许直接由具体类型定义合同。

特定于租户的服务实现不需要从基础实现派生；它们只需要实现服务接口。

你可以在应用程序启动（见 wcf\_startup）时注册特定于租户的服务实现。

### WCF 应用程序启动

应用程序启动通常与其他多租户应用程序（ register\_dependencies）相同，但对客户端有一些小事情要做，对服务有一些主机设置。

#### WCF 客户端应用程序启动

**在** **WCF** **客户端应用程序中**，当你注册服务客户端时，你需要注册将租户 ID 传播到服务的行为。如果你遵循 标准 WCF 集成指南 ，那么注册服务客户端看起来像这样：

var tenantIdStrategy = new MyTenantIdentificationStrategy();  
 var builder = new ContainerBuilder();  
 builder.RegisterType<BaseDependency>().As<IDependency>();  
 builder.Register(c =>  
 new ChannelFactory<IMultitenantService>(  
 new BasicHttpBinding(),  
 new EndpointAddress("http://server/MultitenantService.svc"))).SingleInstance();  
 builder.Register(c =>  
 {  
 var factory = c.Resolve<ChannelFactory<IMultitenantService>>();  
 factory.Opening += (sender, args) => factory.Endpoint.Behaviors.Add(new TenantPropagationBehavior<string>(tenantIdStrategy));  
 return factory.CreateChannel();  
 });  
 var mtc = new MultitenantContainer(tenantIdStrategy, builder.Build());

#### WCF 服务应用程序启动

在 **WCF 服务应用程序** 中，你注册默认值和租户特定的覆盖项，就像通常一样（register\_dependencies），但还需要：

* 设置服务主机的行为以期待接收一个用于识别租户的头信息（behavior\_id）。
* 将服务主机工厂容器设置为 MultitenantContainer。

下面的示例中，我们使用的是 **Autofac.Multitenant.Wcf.AutofacHostFactory** 而不是标准的 Autofac 容器工厂（如前面所述）。

namespace MultitenantExample.WcfService  
{  
 public class Global : System.Web.HttpApplication  
 {  
 protected void Application\_Start(object sender, EventArgs e)  
 {  
   
 var tenantIdStrategy = new OperationContextTenantIdentificationStrategy();  
   
 var builder = new ContainerBuilder();  
 builder.RegisterType<BaseImplementation>().As<IMultitenantService>();  
 builder.RegisterType<BaseDependency>().As<IDependency>();  
   
 var mtc = new MultitenantContainer(tenantIdStrategy, builder.Build());  
 mtc.ConfigureTenant("1",  
 b =>  
 {  
 b.RegisterType<Tenant1Dependency>().As<IDependency>().InstancePerDependency();  
 b.RegisterType<Tenant1Implementation>().As<IMultitenantService>();  
 });  
   
 AutofacHostFactory.HostConfigurationAction =  
 host =>  
 host.Opening += (s, args) =>  
 host.Description.Behaviors.Add(new TenantPropagationBehavior<string>(tenantIdStrategy));  
   
 AutofacHostFactory.ServiceImplementationDataProvider =  
 new MultitenantServiceImplementationDataProvider();  
   
 AutofacHostFactory.Container = mtc;  
 }  
 }  
}

## 租户 ID 策略提示

* **性能是关键。** 租户 ID 策略将在多租户容器中的每个解决操作和每个生命周期范围创建时执行。尽可能使其高效 - 每次都缓存而不是进行数据库查找，减少内存分配等。
* **妥善处理错误。** 如果由于任何原因租户 ID 策略出现问题，可能很难调试。确保检查空指针并处理异常。从性能角度来看，确保使用 TryGet 或 TryParse 类似的操作，而不是 try/catch 并让异常控制流程。
* **使租户** **ID** **策略成为单例。** 多租户容器存储了租户 ID 策略的实例。如果在基（非多租户）容器中注册该策略，请确保将其注册为单例。此外，确保租户 ID 策略可能消耗的依赖项也是单例... 或允许它们具有单独的实例缓存。
* **如果可以，先创建租户** **ID** **策略，然后注册它**，而不是简单地注册类型到 Autofac 并让它解决。很诱人“过度 DI”。租户 ID 策略相当基础，你希望确保它正常工作；调试为什么某些内容没有注入到策略可能会很痛苦。此外，很容易“意外”地使租户 ID 策略成为工厂范围或生命周期范围的东西，这将不起作用。还容易忽略事实，即放入租户 ID 策略中的内容在整个应用程序的生命周期内被缓存，而不是针对每个请求进行填充。如果你实际上“新创建”策略并将其实例与 Autofac 注册，而不是注册单例类型，这可以强化这种纪律并帮助你避免问题。
* **注意线程问题！** 租户 ID 策略由多租户容器持有，并跨所有操作使用。它是单例！如果缓存内容，请确保缓存操作是线程安全的。如果存储状态，请确保使用字典之类的结构，其中可以为系统中的每个租户存储状态（或以不依赖于租户的方式缓存查找）。如果你有一个名为 "tenant" 的实例变量，你会遇到麻烦 - 这个 "tenant" 在每个线程和每个解决操作中都是不同的。

## 示例

Autofac 示例仓库有一个 [多租户WCF服务](https://www.koudingke.cn/go?link=https%3a%2f%2fgithub.com%2fautofac%2fExamples%2ftree%2fmaster%2fsrc%2fMultitenantExample.WcfService)和 [相关的MVC客户端应用程序](https://www.koudingke.cn/go?link=https%3a%2f%2fgithub.com%2fautofac%2fExamples%2ftree%2fmaster%2fsrc%2fMultitenantExample.MvcApplication)，展示了如何 [多租户服务托管](https://github.com/autofac/Examples/tree/master/src/MultitenantExample.MvcApplication) 的工作。

还有一个非常简单的 [控制台应用程序示例](https://www.koudingke.cn/go?link=https%3a%2f%2fgithub.com%2fautofac%2fExamples%2ftree%2fmaster%2fsrc%2fMultitenantExample.ConsoleApplication)。

# AssemblyLoadContext 与生命周期范围

在 .NET Core 中，引入了 [AssemblyLoadContext](https://www.koudingke.cn/go?link=https%3a%2f%2flearn.microsoft.com%2fen-us%2fdotnet%2fcore%2fdependency-loading%2funderstanding-assemblyloadcontext) ，使得开发者能够动态地从应用程序加载和卸载程序集。这对于编写基于插件架构的应用程序的开发者来说非常有用。

为了在程序集不再需要时从 AssemblyLoadContext 卸载程序集，外部不能有任何对该程序集中类型的引用。这包括 Autofac，默认情况下，Autofac 会为已注册的类型保留多种内部引用和缓存。

自 Autofac 7.0 起，我们添加了支持，用于向 Autofac 指示给定生命周期范围表示为特定 AssemblyLoadContext 加载的类型；当为特定 AssemblyLoadContext 创建的生命周期范围被卸载时，Autofac 将尽最大努力尝试移除我们为加载上下文中类型所持有的所有引用，以便能够卸载 AssemblyLoadContext。

你可以通过新的 BeginLoadContextLifetimeScope 方法来指示生命周期范围是针对 AssemblyLoadContext 的。下面是一个完整的示例：

public interface IPlugin  
 {  
 void DoSomething();  
 }  
  
 public class MyPlugin : IPlugin  
 {  
 public void DoSomething()  
 {  
 Console.WriteLine("Hello World");  
 }  
 }  
   
 var builder = new ContainerBuilder();  
 builder.RegisterType<DefaultComponent>();  
 var container = builder.Build();  
 var loadContext = new AssemblyLoadContext("PluginContext", isCollectible: true);  
 using (var scope = container.BeginLoadContextLifetimeScope(loadContext, builder =>  
 {  
 var pluginAssembly = loadContext.LoadFromAssemblyPath("plugins/MyPlugin.dll");  
 builder.RegisterAssemblyTypes(pluginAssembly).AsImplementedInterfaces();  
 }))  
 {   
 var plugin = scope.Resolve<IPlugin>();  
 plugin.DoSomething();  
 }  
 loadContext.Unload();

**注意：**

*如果你在* *Autofac* 之外捕获任何已解析组件或加载程序集中的任何类型的引用，很可能将无法卸载你的加载上下文。  
无论是否使用 Autofac，要确保每次都能卸载 *AssemblyLoadContext* 都很复杂。如果你遇到问题，请参阅 dotnet 文档中关于[解决卸载性问题](https://learn.microsoft.com/en-us/dotnet/standard/assembly/unloadability#troubleshoot-unloadability-issues) 的部分。

你可以使用常规的 BeginLifetimeScope 方法从“加载上下文范围”创建额外的生命周期范围，而无需进一步跟踪加载上下文。

这意味着你可以加载一个插件，然后插件可以解析 ILifetimeScope并创建新的作用域，所有程序集元数据都被隔离到最初的“加载上下文作用域”中。

# Autofac 解析管道

Resolve Pipelines

# Autofac 解析管道

在 Autofac（从 6.0 版本开始）中，实际根据服务请求创建实例的逻辑被实现为一个 **管道** ，由多个 **中间件** 组成。每个单独的中间件代表构建或定位实例所需过程的一部分，并将其返回给你。

对于高级自定义场景，Autofac 允许你在管道中添加自己的中间件，以拦截、短路或扩展现有的解析行为。

## 服务管道与注册管道

Service Pipelines vs Registration Pipelines

每个 服务 都有自己的服务管道，而每个 注册 也有自己的注册管道。

让我们看看典型服务的“默认”执行管道：

![fig:](data:text/html; charset=utf-8;base64,PCFET0NUWVBFIEhUTUwgUFVCTElDICItLy9JRVRGLy9EVEQgSFRNTCAyLjAvL0VOIj48aHRtbD48aGVhZD48dGl0bGU+NDAzIEZvcmJpZGRlbjwvdGl0bGU+PC9oZWFkPjxib2R5PjxoMT40MDMgRm9yYmlkZGVuPC9oMT48cD5Zb3UgZG9uJ3QgaGF2ZSBwZXJtaXNzaW9uIHRvIGFjY2VzcyB0aGUgVVJMIG9uIHRoaXMgc2VydmVyLjwvcD48cD5kZW5pZWQgYnkgVUEgQUNMID0gYmxhY2tsaXN0PC9wPjxoci8+UG93ZXJlZCBieSBUZW5naW5lIDxici8+Q0ROIFJlcXVlc3QgSWQ6IGRkODJjMWEzMTcyNjQ5MTU4OTI5MTY5OTFlPC9ib2R5PjwvaHRtbD4=)

服务管道附着在要解决的服务上，即你用来解决问题的东西。无论实际注册提供实例的方式如何，这些对服务的所有解析都是通用的。

注册管道附着在每个单独的注册上，并适用于所有调用该注册的解析，无论使用哪个服务进行解析。

我们可以利用这种分离管道的概念，将行为附加到给定服务的所有调用（装饰器 这样做），或者附加到单个注册（例如，向管道添加 生命周期事件）。

## 管道阶段

Pipeline Phases

当我们向管道添加中间件时，需要指定中间件应运行的 **阶段**。

通过指定阶段，我们可以允许在管道内部对中间件进行排序，从而不依赖于添加中间件的实际顺序。

以下是可用的管道阶段，分为服务阶段和注册阶段。

| **服务管道阶段** | **描述** |
| --- | --- |
| ResolveRequestStart | 解析请求的开始。在这一阶段添加的自定义中间件会在检测循环依赖之前执行。 |
| ScopeSelection | 在这一阶段，选择生命周期范围。如果某些中间件需要更改范围来针对其解析，它会在这里发生（但请注意，注册的 Autofac 生命周期仍然有效）。 |
| Decoration | 在这个阶段，将对实例进行装饰（在管道的输出方向）。 |
| Sharing | 在这一阶段结束时，如果共享实例满足请求，管道将停止执行并退出。在此阶段添加自定义中间件以选择你自己的共享实例。 |
| ServicePipelineEnd | 此阶段发生在服务管道结束（即将开始注册管道）之前。 |
| **注册管道阶段** | **描述** |
| RegistrationPipelineStart | 在注册管道开始时发生。 |
| ParameterSelection | 在激活之前运行此阶段，是推荐的参数替换点，如果需要的话。 |
| Activation | 激活阶段是管道的最后一个阶段，在此阶段创建组件的新实例。 |

**注意**

如果你尝试在添加注册中间件时指定服务管道阶段（反之亦然），你将收到错误。你需要根据要添加到的管道使用适当的阶段。

## 添加注册中间件

Adding Service Middleware

让我们看看如何在创建注册时向注册管道插入我们自己的中间件，使用一个简单的 “Hello World” lambda 中间件，它将在控制台打印一些信息：

var builder = new ContainerBuilder();  
 builder.RegisterType<MyImplementation>().As<IMyService>().ConfigurePipeline(p =>  
 {  
   
 p.Use(PipelinePhase.RegistrationPipelineStart, (context, next) =>  
 {  
 Console.WriteLine("Before Activation - request {0}", context.Service);  
   
 next(context);  
 Console.WriteLine("After Activation - instanctation {0}", context.Instance);  
 });  
 });

你可以看到，我们通过提供给 next回调的方法调用管道中的下一个中间件，从而允许解析操作继续。

在 next返回后，你可以访问创建的实例。这是因为在调用 next时，它会调用管道中的下一个中间件，然后再次调用 next，直到管道结束，实例被激活。

如果不调用 next回调，管道将结束，我们将返回到调用者。

### 定义中间件类

除了通过 lambda 函数提供中间件外，你还可以定义自己的中间件类，并将这些类的实例添加到管道中：

class MyCustomMiddleware : IResolveMiddleware  
 {  
 public PipelinePhase Phase => PipelinePhase.RegistrationPipelineStart;  
 public void Execute(ResolveRequestContext context, Action<ResolveRequestContext> next)  
 {  
 Console.WriteLine("Before Activation - 请求 {0}", context.Service);  
   
 next(context);  
 Console.WriteLine("After Activation - 实例化 {0}", context.Instance);  
 }  
 }  
 builder.RegisterType<MyImplementation>().As<IMyService>().ConfigurePipeline(p =>  
 {  
 p.Use(new MyCustomMiddleware());  
 });

两种添加中间件的方式行为相同，但对于复杂的中间件，定义一个类可能会有所帮助。

### 将中间件添加到所有注册

Adding Middleware to all Registrations

如果你想为所有注册添加一段中间件，可以像添加其他共享注册行为一样使用 Registered 事件：

builder.ComponentRegistryBuilder.Registered += (sender, args) =>  
 {  
   
 args.ComponentRegistration.PipelineBuilding += (sender2, pipeline) =>  
 {  
 pipeline.Use(new MyCustomMiddleware());  
 };  
 };

## 解析请求上下文

ResolveRequestContext

传递给所有中间件的上下文对象是 ResolveRequestContext的一个实例。这个对象存储了解析请求的初始属性，以及解析执行过程中更新的任何属性。

你可以使用此上下文：

* 检查正在解析的服务，使用 Service 属性。
* 检查提供服务的注册。
* 使用 Instance 属性获取或设置解析操作的结果。
* 使用 Parameters 属性访问请求参数，并使用 ChangeParameters 方法更改参数。
* 使用任何正常的解析方法解析另一个服务。

**注意:**  
**ResolveRequestContext**是一个抽象基类。如果你想为中间件编写单元测试，可以创建一个模拟对象并将模拟对象传递给中间件实现。

## 添加服务中间件

服务中间件附着在服务上，而不是特定的注册。因此，当我们添加服务中间件时，我们可以为服务的所有解析添加行为，而不关心提供实例的注册是什么。

直接将服务中间件添加到 ContainerBuilder 上：

var builder = new ContainerBuilder();  
 builder.RegisterServiceMiddleware<IMyService>(PipelinePhase.ResolveRequestStart, (context, next) =>  
 {  
 Console.WriteLine("Request service：{0}", context.Service);  
 next(context);  
 });

与注册中间件类似，你可以注册中间件类而不是 lambda 函数：

builder.RegisterServiceMiddleware<IMyService>(new MyServiceMiddleware());

# 服务中间件源

Service Middleware Sources

类似于 注册源 ，如果你想在运行时动态地在运行时添加服务中间件，可以添加一个 **服务中间件源**。

这对于像开放泛型服务这样的情况特别有用，其中我们不知道实际的服务类型直到运行时。

通过实现 IServiceMiddlewareSource 并将其注册到 ContainerBuilder 来定义服务中间件源。

class MyServiceMiddlewareSource : IServiceMiddlewareSource  
 {  
 public void ProvideMiddleware(Service service, IComponentRegistryServices availableServices, IResolvePipelineBuilder pipelineBuilder)  
 {  
   
 pipelineBuilder.Use(PipelinePhase.Sharing, (context, next) =>  
 {  
 Console.WriteLine("我出现在每个服务上！");  
 next(context);  
 });  
 }  
 }  
   
 builder.RegisterServiceMiddlewareSource(new MyServiceMiddlewareSource());

# Autofac 聚合服务

Aggregate Services

## 引言

当你需要将一组依赖项视为一个依赖项时，聚合服务非常有用。当一个类依赖于多个构造注入的服务，或者拥有多个属性注入的服务时，将这些服务移动到单独的类中可以简化 API。

例如，在超类和子类中，超类有一个或多个构造注入的依赖项。子类通常必须继承这些依赖项，即使它们对超类可能只具有用。通过使用聚合服务，可以将超类构造函数参数合并为一个参数，从而减少子类中的重复性。另一个重要副作用是，子类现在可以防止超类依赖性的更改，这意味着在超类中引入新的依赖项只需更改聚合服务定义即可。

模式和这个示例都在 [这里进行了进一步的阐述](https://www.koudingke.cn/go?link=http%3a%2f%2fpeterspattern.com%2fdependency-injection-and-class-inheritance)。

可以通过手动创建一个具有构造注入依赖项的类并公开这些属性来实现聚合服务。然而，编写和维护聚合服务类及其伴随的测试可能会很快变得乏味。Autofac 的 AggregateService 扩展允许你直接从接口定义生成聚合服务，而无需编写任何实现。

## 必要的引用

你可以使用 [Autofac.Extras.AggregateService NuGet 包](https://nuget.org/packages/Autofac.Extras.AggregateService) 或手动添加对以下程序集的引用来向项目添加聚合服务支持：

* Autofac.dll
* Autofac.Extras.AggregateService.dll
* Castle.Core.dll ([来自Castle 项目](https://www.koudingke.cn/go?link=http%3a%2f%2fwww.castleproject.org%2fdownload%2f)*)*

## 开始使用

假设我们有一个具有多个构造注入依赖项的类，这些依赖项我们私有存储以供稍后使用：

public class SomeController  
 {  
 private readonly IFirstService \_firstService;  
 private readonly ISecondService \_secondService;  
 private readonly IThirdService \_thirdService;  
 private readonly IFourthService \_fourthService;  
 public SomeController(  
 IFirstService firstService,  
 ISecondService secondService,  
 IThirdService thirdService,  
 IFourthService fourthService)  
 {  
 \_firstService = firstService;  
 \_secondService = secondService;  
 \_thirdService = thirdService;  
 \_fourthService = fourthService;  
 }  
 }

为了聚合这些依赖项，我们将这些依赖项移到单独的接口定义，并改为依赖该接口。

public interface IMyAggregateService  
 {  
 IFirstService FirstService { get; }  
 ISecondService SecondService { get; }  
 IThirdService ThirdService { get; }  
 IFourthService FourthService { get; }  
 }  
 public class SomeController  
 {  
 private readonly IMyAggregateService \_aggregateService;  
 public SomeController(IMyAggregateService aggregateService)  
 {  
 \_aggregateService = aggregateService;  
 }  
 }

最后，我们注册聚合服务接口。

using Autofac;  
 using Autofac.Extras.AggregateService;  
 var builder = new ContainerBuilder();  
 builder.RegisterAggregateService<IMyAggregateService>();  
 builder.Register().As<IFirstService>();  
 builder.Register().As<ISecondService>();  
 builder.Register().As<IThirdService>();  
 builder.Register().As<IFourthService>();  
 builder.RegisterType<SomeController>();  
 var container = builder.Build();

聚合服务的接口将自动为你生成实现，依赖项将按照预期填充。

## 聚合服务的解析方式

## 属性

只读属性模拟了常规构造注入依赖项的行为。每个属性的类型将在构建聚合服务实例时解析并缓存。

以下是功能等效的示例：

class MyAggregateServiceImpl : IMyAggregateService  
 {  
 private IMyService \_myService;  
 public MyAggregateServiceImpl(IComponentContext context)  
 {  
 \_myService = context.Resolve<IMyService>();  
 }  
 public IMyService MyService  
 {  
 get { return \_myService; }  
 }  
 }

## 方法

方法将像工厂委托一样工作，并会在每次调用时转化为对每个方法的 Resolve调用。方法的返回类型将被解析，并将任何参数传递给 Resolve调用。

这是方法调用的功能等效样本：

class MyAggregateServiceImpl : IMyAggregateService  
 {  
 public ISomeThirdService GetThirdService(string data)  
 {  
 var dataParam = new TypedParameter(typeof(string), data);  
 return \_context.Resolve<ISomeThirdService>(dataParam);  
 }  
 }

## 属性设置器和无返回类型的函数

在聚合服务中，属性设置器和无返回类型的函数没有意义。尽管它们在聚合服务接口中存在，但不会阻止代理生成。然而，调用这样的方法会抛出异常。

## 工作原理

在内部，AggregateService 使用 [Castle 项目的 DynamicProxy2](https://www.koudingke.cn/go?link=http%3a%2f%2fcastleproject.org) 。给定一个接口（将服务聚合到一个接口），将生成一个实现该接口的代理。代理将翻译对属性和方法的调用为对 Autofac 上下文的Resolve调用。

## 性能考虑

由于聚合服务中的方法调用通过动态代理进行，每次方法调用都会有一些小但非零的开销。关于城堡动态代理框架与其他框架的性能研究可以在 [这里](https://www.koudingke.cn/go?link=http%3a%2f%2fkozmic.pl%2f2009%2f03%2f31%2fdynamic-proxy-frameworks-comparison-update%2f)找到。

# Autofac 类型拦截器

Type Interceptors

Castle.Core 是 [Castle 框架](https://www.koudingke.cn/go?link=http%3a%2f%2fcastleproject.org) 的一部分，它提供了一个名为 "DynamicProxy" 的方法拦截框架。

Autofac.Extras.DynamicProxy 集成包允许对 Autofac 组件的方法调用进行拦截，常见的使用场景包括事务处理、日志记录和声明式安全。你可以使用 Autofac.Extras.DynamicProxy2 与 Autofac 4.0.0 之前的版本配合使用。

## 启用拦截

要使 DynamicProxy 集成工作，基本步骤如下：

* create\_interceptors
* register\_interceptors
* enable*type*interception
* associate\_interceptors

### 创建拦截器

拦截器实现 Castle.DynamicProxy.IInterceptor 接口。下面是一个简单的拦截器示例，用于记录方法调用，包括输入和输出：

public class CallLogger : IInterceptor  
 {  
 TextWriter \_output;  
 public CallLogger(TextWriter output)  
 {  
 \_output = output;  
 }  
 public void Intercept(IInvocation invocation)  
 {  
 \_output.Write("Calling method {0} with parameters {1}... ",  
 invocation.Method.Name,  
 string.Join(", ", invocation.Arguments.Select(a => (a ?? "").ToString()).ToArray()));  
 invocation.Proceed();  
 \_output.WriteLine("Done: result was {0}.", invocation.ReturnValue);  
 }  
 }

### 注册拦截器

拦截器必须在容器中注册。你可以将其注册为类型服务或命名服务。如果你选择命名服务，它们必须命名为 IInterceptor注册项。

选择哪种方式取决于你如何决定将拦截器与被拦截的类型关联起来。

builder.Register(c => new CallLogger(Console.Out))  
 .Named<IInterceptor>("log-calls");  
 builder.Register(c => new CallLogger(Console.Out));

### 为类型启用拦截

当你注册一个被拦截的类型时，需要在注册时标记该类型，以便 Autofac 知道要为此设置拦截。你可以使用 EnableInterfaceInterceptors()和 EnableClassInterceptors()注册扩展来实现这一点。

var builder = new ContainerBuilder();  
 builder.RegisterType<SomeType>()  
 .As<ISomeInterface>()  
 .EnableInterfaceInterceptors();  
 builder.Register(c => new CallLogger(Console.Out));  
 var container = builder.Build();  
 var willBeIntercepted = container.Resolve<ISomeInterface>();

背后，EnableInterfaceInterceptors() 会创建一个执行拦截的接口代理，而 EnableClassInterceptors() 则会动态子类化目标组件，以拦截虚拟方法。

这两种技术都可以与 Assembly 扫描功能结合使用，因此你可以使用相同的方法配置一组组件。

**特殊情况：WCF 代理和远程对象** 虽然 WCF 代理对象看起来像接口，但 EnableInterfaceInterceptors() 机制无法工作，因为 .NET 在幕后实际上是使用了行为像接口的 System.Runtime.Remoting.TransparentProxy 对象。如果要在 WCF 代理上启用拦截，请使用 InterceptTransparentProxy() 方法。

var cb = new ContainerBuilder();  
 cb.RegisterType<TestServiceInterceptor>();  
 cb.Register(c => CreateChannelFactory()).SingleInstance();  
 cb  
 .Register(c => c.Resolve<ChannelFactory<ITestService>>().CreateChannel())  
 .InterceptTransparentProxy(typeof(IClientChannel))  
 .InterceptedBy(typeof(TestServiceInterceptor))  
 .UseWcfSafeRelease();

### 将拦截器与待拦截类型关联

为了选择与你的类型关联的拦截器，你有两种选择。

你的第一种选择是为类型添加一个属性，如下所示：

[Intercept(typeof(CallLogger))]  
 public class First  
 {  
 public virtual int GetValue()  
 {  
   
 }  
 }  
 [Intercept("log-calls")]  
 public class Second  
 {  
 public virtual int GetValue()  
 {  
   
 }  
 }

当使用属性将拦截器关联起来时，你不需要在注册时指定拦截器。只需启用拦截，拦截器类型将自动被发现。

var builder = new ContainerBuilder();  
 builder.RegisterType<First>()  
 .EnableClassInterceptors();  
 builder.Register(c => new CallLogger(Console.Out));  
 var builder = new ContainerBuilder();  
 builder.RegisterType<Second>()  
 .EnableClassInterceptors();  
 builder.Register(c => new CallLogger(Console.Out))  
 .Named<IInterceptor>("log-calls");

第二种选择是在 Autofac 注册时间声明拦截器。你可以使用 InterceptedBy()注册扩展来实现：

var builder = new ContainerBuilder();  
 builder.RegisterType<SomeType>()  
 .EnableClassInterceptors()  
 .InterceptedBy(typeof(CallLogger));  
 builder.Register(c => new CallLogger(Console.Out));

## 提示

Tips

### 使用 public 接口

接口拦截要求接口是公开的（至少对动态生成的代理程序集可见）。非公共接口类型无法被拦截。

如果你想代理 internal 接口，必须在包含接口的程序集中添加注解 [assembly: InternalsVisibleTo("DynamicProxyGenAssembly2")]。

### 使用虚方法

virtual method

类拦截要求被拦截的方法是虚方法，因为它使用子类化作为代理技术。

### 与表达式一起使用

使用表达式创建的组件或作为实例注册的组件，不能被 DynamicProxy2 引擎子类化。在这种情况下，需要使用基于接口的代理。

### 接口注册

要通过接口启用代理，组件必须只通过接口提供其服务。为了获得最佳性能，所有此类服务接口都应包含在注册中，即包含在 As()子句中。

### WCF 代理

如前所述，WCF 代理和其他远程类型是特殊情况，不能使用标准的接口或类拦截。你需要为这些类型使用 InterceptTransparentProxy()。

**类拦截器和** **UsingConstructor**

如果你通过 EnableClassInterceptors()使用类拦截器，请避免同时使用 UsingConstructor()。当启用类拦截时，生成的代理会添加一些新的构造函数，这些构造函数也会接受你想要使用的拦截器集合。当你指定 UsingConstructor()时，会跳过此逻辑，导致你的拦截器未被使用。

## 已知问题

### 异步方法拦截

Castle 拦截器仅提供了同步方法拦截的机制——没有显式的 async/await 支持。然而，由于 async/await 只是返回 Task 对象的语法糖，你可以在拦截器中使用 Task 和 ContinueWith() 类似的方法。[这个issue](https://www.koudingke.cn/go?link=https%3a%2f%2fgithub.com%2fcastleproject%2fCore%2fissues%2f107)中有一个示例。另一种选择是使用 [辅助库](https://www.koudingke.cn/go?link=https%3a%2f%2fgithub.com%2fJSkimming%2fCastle.Core.AsyncInterceptor)，它们可以使异步操作更方便。

### Castle.Core 版本管理

从 Castle.Core 4.2.0 开始，Castle.Core 的 NuGet 包版本 更新了，但 Assembly 版本 没有更新。此外，Castle.Core 4.1.0 的 Assembly 版本 与包（4.1.0.0）匹配，而 4.2.0 包回退到 4.0.0.0。在完整 .NET 框架项目中，关于 Castle.Core 版本的任何混淆可以通过添加一个 assembly 绑定重定向来解决，强制使用 Castle.Core 4.0.0.0。

不幸的是，.NET Core 不支持 assembly 绑定重定向。如果你直接依赖 Castle.Core，同时又通过库（如 Autofac.Extras.DynamicProxy）间接依赖 Castle.Core，你可能会看到类似这样的错误：

System.IO.FileLoadException: 无法加载文件或程序集 'Castle.Core, Version=4.1.0.0, Culture=neutral, PublicKeyToken=407dd0808d44fbdc'。已加载的程序集与程序集中引用的程序集的定义不匹配。 (异常来自 HRESULT: 0x80131040)

这是因为回退的 assembly。

**确保你使用了最新版的 Autofac.Extras.DynamicProxy**。我们会尽最大努力从 Autofac 方面解决问题。更新该库或 Castle.Core 可能会有帮助。

如果这还不行，有两个解决方案：

一是删除你的直接 Castle.Core 引用。间接引用应该会自行解决。

二是如果你不能删除直接引用，或者删除后不起作用……所有直接依赖项都需要更新到 Castle.Core 4.2.0 或更高版本。你需要向这些项目报告问题；这不是 Autofac 可以为你解决的问题。

这里有关于这一挑战的 [Castle.Core issue。](https://github.com/castleproject/Core/issues/288)

# 跨平台和原生应用

Xamarin 和 .NET Native 等工具让 .NET 代码能够编译到特定平台。然而，由于 .NET 反射不一定在所有原生平台上 “正常工作” ，并且参数绑定和对象构造很大程度上依赖于反射，因此有时需要额外的工作才能使 Autofac 和依赖注入（DI）正常工作。

## Xamarin

使用 Xamarin 创建 iOS 或 Android 应用时，如果启用了链接器，可能需要明确指定需要反射支持的类型。[Xamarin自定义链接器配置](https://www.koudingke.cn/go?link=https%3a%2f%2fdeveloper.xamarin.com%2fguides%2fcross-platform%2fadvanced%2fcustom_linking%2f) 文档解释了如何通知链接器保留某些类型，而不会从最终产品中移除它们。具体来说...

* 使用 [Preserve] 属性标记你拥有的类型
* 在构建过程中包含自定义 XML 链接描述文件

一个简单的链接描述文件如下所示：

<linker>  
 <assembly fullname="mscorlib">  
 <type fullname="System.Convert" />  
 </assembly>  
 <assembly fullname="My.Own.Assembly">  
 <type fullname="Foo" preserve="fields">  
 <method name=".ctor" />  
 </type>  
 <namespace fullname="My.Own.Namespace" />  
 <type fullname="My.Other\*" />  
 </assembly>  
 <assembly fullname="Autofac" preserve="all"/>  
 </linker>

Autofac 在 lambda 表达式中使用 System.Convert.ChangeType 方法进行类型转换，因此需要将其包含在链接器定义中。有关进一步讨论，请参阅 [issue #842](https://www.koudingke.cn/go?link=https%3a%2f%2fgithub.com%2fautofac%2fAutofac%2fissues%2f842)。

关于如何组织 Xamarin 自定义链接器配置文件以及如何将其包含在构建中，可以查看 [Xamarin 文档](https://www.koudingke.cn/go?link=https%3a%2f%2fdeveloper.xamarin.com%2fguides%2fcross-platform%2fadvanced%2fcustom_linking%2f)。

Xamarin 链接器可能不会认为 Autofac 是“链接器安全”的。如果链接器过于激进，可能会看到类似以下异常：

The type 'Autofac.Features.Indexed.KeyedServiceIndex'2' does not implement the interface 'Autofac.Features.Indexed.IIndex'2'

[此 StackOverflow 回答]([https://stackoverflow.com/questions/58114288/autofac-build-throws-exception-on-latest-xamarin-ios-when-linker-configured-to](https://www.koudingke.cn/go?link=https%3a%2f%2fstackoverflow.com%2fquestions%2f58114288%2fautofac-build-throws-exception-on-latest-xamarin-ios-when-linker-configured-to)) 指出可以采取以下措施之一：

* 将链接器设置为 Don't link 或 Link Framework SDKs Only（这将增加应用程序大小）
* 在 iOS 项目的属性中，在“iOS Build”下的“Additional mtouch arguments”中添加 --linkskip=Autofac 参数。
* 使用上述链接器 XML，并确保包含 preserve="all" 的 Autofac 行。

## .NET Native

[.NET Native](https://www.koudingke.cn/go?link=https%3a%2f%2fmsdn.microsoft.com%2fen-us%2flibrary%2fdn584397(v%3dvs.110).aspx) 是一种将 .NET 可执行文件编译为原生代码的方法。它用于 Universal Windows 平台 (UWP) 和 Windows 商店应用等场景。

当使用 [.NET Native 进行反射](https://www.koudingke.cn/go?link=https%3a%2f%2fmsdn.microsoft.com%2fen-us%2flibrary%2fdn600640(v%3dvs.110).aspx) 时，如果编译器删除了你需要的类型的反射元数据，可能会遇到 MissingMetadataException 等异常。

你可以使用 [运行时指令 (rd.xml) 文件](https://www.koudingke.cn/go?link=https%3a%2f%2fmsdn.microsoft.com%2fen-us%2flibrary%2fdn600639(v%3dvs.110).aspx) 来配置 .NET Native 编译。一个简单的指令文件如下所示：

<Directives xmlns="http://schemas.microsoft.com/netfx/2013/01/metadata">  
 <Application>  
 <Assembly Name="\*Application\*" Dynamic="Required All" />  
 </Application>  
 </Directives>

该指令文件告诉编译器保留整个应用程序包中所有内容的所有反射数据。这是一种“核选项”——如果你想减小应用程序包的大小，可以更具体地指定要包括的内容。有关更多详细信息，请参考 [MSDN文档](https://www.koudingke.cn/go?link=https%3a%2f%2fmsdn.microsoft.com%2fen-us%2flibrary%2fdn600639(v%3dvs.110).aspx)。

# 调试与故障排查

Debugging and Troubleshooting

如果你遇到严重问题，且在 [StackOverflow](https://stackoverflow.com/questions/tagged/autofac)上没有找到答案，可以尝试自行进行更深入的调试和故障排查。这里有一些建议供你参考。

## 异常处理

Autofac 生成的异常通常会指引你找到问题可能出在哪里。

**不要惊慌！停下来仔细阅读它告诉你的内容。** 以下是一些阅读 Autofac 异常的技巧，可以帮助你节省时间：

* **阅读完整信息。** 异常通常非常具体，会明确指出缺少或错误的地方。然而，过于具体可能导致信息冗长。不要略过消息，真正阅读它！
* **查看嵌套异常。** 在大型对象图中，有时失败的原因可能在堆栈的深处。Autofac 会尽力将重要信息推送到顶部，但有时这并不容易实现。确保查看嵌套异常信息——它们并非无关紧要！
* **注意堆栈跟踪。** 在某些情况下，可能会看起来是 Autofac 导致了错误，但问题可能出在注册的构造函数逻辑或委托上。跟随整个堆栈跟踪，找出问题的确切位置。

## 日志诊断

从 Autofac 6.0 开始，我们引入了基于 [System.Diagnostics.DiagnosticSource](https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.diagnosticsource?view=netcore-3.1) 的日志诊断支持。这允许你拦截 Autofac 发出的日志事件，以便进行更深入的调试。

我们有一篇 [详细说明如何接入日志诊断并进行故障排查的页面](https://docs.autofac.org/en/latest/troubleshooting/diagnostics.html) 。

## Support

我们有一篇 [如何获取支持的概述页面](https://autofac.readthedocs.io/en/latest/support.html)。

## 异常

Exceptions

我们在这里有一些关于常见异常及其解决方法：

### 1.请求的服务未注册

The requested service has not been registered

#### 示例信息

The requested service 'MyApp.IMyDependency' has not been registered. To  
avoid this exception, either register a component to provide the service,  
check for service registration using IsRegistered(), or use the  
ResolveOptional() method to resolve an optional dependency.

请求的服务 MyApp.IMyDependency 未在容器/生命周期作用域中注册。要避免此异常，请使用以下方法之一来解析服务：注册一个组件以提供服务；使用 IsRegistered() 方法检查服务注册情况；或使用 ResolveOptional() 方法来解决可选依赖。

这个错误意味着 Autofac 尝试根据应用程序请求解析服务，但该服务并未在容器/生命周期作用域中注册。消息指出了无法找到的服务。例如，在上面的示例中，未在生命周期作用域中找到 MyApp.IMyDependency服务。

#### 故障排除

Troubleshooting

重要的是要记住**服务**和**组件**之间的区别：**服务**是正在被暴露的接口或类，而**组件**则是为了实现该服务而创建的具体类。

在这样的注册中...

builder.RegisterType<MyComponent>().As<IMyService>();

...MyComponent 是 *组件*，而 IMyService 是 *服务*。

异常告诉你哪个 *服务* 未找到。查看堆栈跟踪，你应该能够看到应用程序中试图解析该服务的代码。这可能出现在更长的解决链路中，例如当你有一个构造函数需要 IMyService的对象，并尝试解析该对象时。

**这个问题的常见原因是注册遗漏。** 确保已注册服务。尤其是使用 程序集扫描 时，容易忽略某些内容。

还可能发生更微妙的问题，即具有相似名称但不同命名空间的 **服务** 可能已注册。例如，如果你的应用程序有 FirstNamespace.IMyService和 SecondNamespace.IMyService，在类文件顶部添加一些 using语句并注册 IMyService，可能没有意识到实际注册了哪个服务。代码看起来像是正确注册的，但实际上并非如此。

同样罕见的是，这可能发生在动态加载程序集并扫描依赖项的“插件系统”中：两个不同的程序集可能具有相同的命名空间和接口名称。实际上注册的是哪个？

最后，如果你使用 AnyConcreteTypeNotAlreadyRegisteredSource来节省时间，可能会意外地将容器中不需要注册或不使用的项自动包含进来。这些额外的项也可能实现你试图解析的接口，因此在解析服务时，由于缺少依赖项，它会生成此类错误，这可能会非常令人困惑。尝试移除 ACTNARS 使用，更明确地注册组件，可能使用程序集扫描代替。

### 2.找不到构造函数

No constructors can be found

#### 示例信息

No constructors on type 'MyApp.MyComponent' can be found with the  
constructor finder 'Autofac.Core.Activators.Reflection.DefaultConstructorFinder'.

在类型 'MyApp.MyComponent' 上找不到构造函数，使用的查找器是 'Autofac.Core.Activators.Reflection.DefaultConstructorFinder'。

**这意味着:**

Autofac 尝试通过反射创建组件，但用于定位组件构造函数的服务表示该组件没有可用的构造函数。

在示例消息中，我们可以看到：

* 正在创建的类是 MyApp.MyComponent。
* MyApp.MyComponent 类正在被 Autofac.Core.Activators.Reflection.DefaultConstructorFinder 查找构造函数。
* Autofac.Core.Activators.Reflection.DefaultConstructorFinder 返回了 0 个可用构造函数。

#### 故障排除

这种情况通常发生在 Autofac 注册了一个没有公共构造函数的组件，例如只有内部或私有构造函数的对象。默认情况下，Autofac 只支持公共构造函数。

如果自定义了 IConstructorFinder 实例，错误信息告诉你自定义构造函数查找器的 FindConstructors() 方法返回了空的构造函数数组。这种场景（创建自定义 IConstructorFinder）并不常见，但确实会发生。在这种情况下，你会看到组件注册带有 FindConstructorsWith() 调用，以附上自定义构造函数查找器。

检查异常中提到的组件（类），看看是否有任何构造函数。如果没有构造函数，那么应该存在编译器生成的默认无参构造函数（它是公开的）。如果类中有构造函数，那么编译器不会生成那个默认的公开无参构造函数。如果没有任何公开构造函数，那就是问题所在。

有关构造函数选择的更多信息，请阅读 这里 。

### 3.找到的构造函数都无法调用

None of the constructors found can be invoked

#### 示例信息

None of the constructors found on type 'MyApp.MyComponent' can be invoked with the available services and parameters:   
Cannot resolve parameter 'MyApp.IMyDependency myDep' of constructor 'Void .ctor(MyApp.IMyDependency)'.

这个错误意味着 Autofac 尝试使用构造函数来实例化一个组件，但构造函数中列出的依赖项中，容器无法从其中解析出任何。消息中会告诉你哪些参数无法被解析。

在示例信息中，我们能看到：

* 被创建的类是 MyApp.MyComponent。
* MyApp.MyComponent 类正在使用签名为 Void .ctor(MyApp.IMyDependency) 的构造函数（单参数构造函数）进行创建。
* 类型为 MyApp.IMyDependency 的参数没有在 Autofac 容器中注册。

#### 解决问题

这种情况通常有以下原因：

首先，**参数类型可能代表了一个尚未正确注册到容器中的服务。** 在上面的例子中，可能是 MyApp.IMyDependency没有被注册，因此无法作为其他组件的依赖。试着在测试中手动解决该服务。例如，你可以尝试调用 container.Resolve<MyApp.IMyDependency>()来查看情况。你可能认为它已经注册了，但实际上并没有。

其次，**在服务的依赖项解析过程中出现了错误。** 在上述例子中，可能是 MyApp.IMyDependency构造函数中的逻辑遇到了问题。阅读内部异常以获取更多关于问题的确切原因的信息。

最后，**你的构造函数中可能存在一个实际上不是服务的参数，容器中不会包含它。** 例如，通常不会将纯字符串或整数值注册到容器中，但如果构造函数需要这样的值，它将无法运行。在这种情况下，可以在注册时指定 Parameter 提供额外参数。我们还有一篇关于 注入配置参数（如连接字符串或环境变量）的 FAQ 可能对你有所帮助。

如果组件有多个构造函数，Autofac 默认会尝试使用能够从容器中满足最多参数的那个。如果你想，可以 指定一个不包含问题参数的构造函数 。

### 4.没有匹配标签的作用域

#### 示例消息

No scope with a tag matching 'AutofacWebRequest' is visible from the scope  
in which the instance was requested.  
  
If you see this during execution of a web application, it generally  
indicates that a component registered as per-HTTP request is being  
requested by a SingleInstance() component (or a similar scenario). Under  
the web integration always request dependencies from the dependency  
resolver or the request lifetime scope, never from the container itself.

没有匹配标签 "AutofacWebRequest" 的可见范围，该范围中请求了实例。

如果您在运行 Web 应用程序时看到此消息，通常表示注册为按 HTTP 请求的组件被单例（或类似场景）组件所请求。在集成 Web 应用程序时，始终从依赖解析器或请求生命周期范围请求依赖项，而不是直接从容器本身请求。

这个错误意味着 Autofac 尝试解决一个已注册为 InstancePerMatchingLifetimeScope 的服务，但没有找到匹配的生命周期范围。

#### 故障排除

* **在 ASP.NET 经典 Web 应用程序（如 ASP.NET MVC、Web 表单等）中**，可能是因为缺少了 [请求生命周期范围](https://www.koudingke.cn/docs/zh-Hans/autofac-docs/latest/Faq/Per-Request-Scope) 。例如，试图在应用程序启动时运行但使用按请求注册的服务的代码将失败。 更多有关如何调试此类问题的信息，请参阅 [关于处理请求生命周期范围的 FAQ](https://www.koudingke.cn/docs/zh-Hans/autofac-docs/latest/Faq/Per-Request-Scope) 。
* **在 ASP.NET Core 应用程序中**，可能是您已将某些内容注册为 InstancePerRequest。ASP.NET Core 没有名为请求生命周期范围的命名范围 - 相反，请使用 InstancePerLifetimeScope。有关更多详细信息，请参阅 [ASP.NET Core 集成文档](https://www.koudingke.cn/docs/zh-Hans/autofac-docs/latest/Integration/AspNetCore) 。
* **如果您在自己的应用程序中创建了自定义的按请求语义**，类似于 InstancePerRequest，那么可能是有一些代码将某些内容注册为 InstancePerRequest，但服务是在请求生命周期范围之外进行解析的。这可能发生在试图在具有按请求生命周期范围和不具有此类范围（如 Web 应用程序和后台处理任务）之间共享注册模块的情况下，或者在应用程序启动之前尝试使用按请求对象的代码运行时。
* **如果注册了一些组件与匹配的生命周期范围，而另一些则没有**，可能会导致问题。例如，假设您注册了一个 IMyService 作为单例，然后又注册了第二个 IMyService 为 InstancePerMatchingLifetimeScope。如果在不匹配的生命周期范围内尝试解析 IEnumerable<IMyService>，您会收到错误。尽管其中一个 IMyService 可能可以解析，但并非所有 IMyService 都可以。InstancePerMatchingLifetimeScope 并不是一个允许某些实例 有时被解析的过滤器。

## 诊断

Diagnostics

Autofac 6.0 引入了诊断支持，形式为 [System.Diagnostics.DiagnosticSource](https://www.koudingke.cn/go?link=https%3a%2f%2fdocs.microsoft.com%2fen-us%2fdotnet%2fapi%2fsystem.diagnostics.diagnosticsource%3fview%3dnetcore-3.1) 。 这样可以让你拦截 Autofac 的诊断事件。

**注意：**  
**诊断并非免费的。如果你不将诊断监听器附加到容器，性能会更好。**此外，如 DefaultDiagnosticTracer 这样的跟踪器在生成操作完整跟踪时会增加内存和资源使用量，因为它们必须在整个解析操作期间保留数据以生成完整的跟踪。建议你在非生产环境中使用诊断；或者使用只处理个别事件而不跟踪完整操作的诊断监听器。

### 快速入门

要开始使用诊断，最简单的方法是使用 Autofac.Diagnostics.DefaultDiagnosticTracer 类。此跟踪器将生成可用于调试的解析操作的层次结构化跟踪。

var containerBuilder = new ContainerBuilder();  
 containerBuilder.RegisterType<Component>().As<IService>();  
 var container = containerBuilder.Build();  
 var tracer = new DefaultDiagnosticTracer();  
 tracer.OperationCompleted += (sender, args) =>  
 {  
 Trace.WriteLine(args.TraceContent);  
 };  
 container.SubscribeToDiagnostics(tracer);  
 using var scope = container.BeginLifetimeScope();  
 scope.Resolve<IService>();

如果你无法直接访问容器（例如，在 ASP.NET Core 中），可以使用构建回调来注册追踪器。

public void ConfigureContainer(ContainerBuilder builder)  
 {  
   
 builder.RegisterModule(new AutofacModule());  
   
 var tracer = new DefaultDiagnosticTracer();  
 tracer.OperationCompleted += (sender, args) =>  
 {  
 Console.WriteLine(args.TraceContent);  
 };  
 builder.RegisterBuildCallback(c =>  
 {  
 var container = c as IContainer;  
 container.SubscribeToDiagnostics(tracer);  
 });  
 }

### 默认诊断追踪器

Default Diagnostic Tracer

上面的快速入门演示了如何使用 Autofac.Diagnostics.DefaultDiagnosticTracer。

当 OperationCompleted 事件被触发时，你会收到事件参数，这些参数提供了：

* Operation - 完成的实际解析操作，以便在需要时进行检查。
* OperationSucceeded - 一个布尔值，指示包含的跟踪是成功还是失败的操作。
* TraceContent - 具有完整解析操作跟踪的构建字符串。

假设你有一个简单的 lambda，它注册一个字符串。

var builder = new ContainerBuilder();  
 builder.Register(ctx => "HelloWorld");  
 var container = builder.Build();

如果从该容器解析字符串，跟踪看起来像这样：

Resolve Operation Starting  
 {  
 Resolve Request Starting  
 {  
 Service: System.String  
 Component: λ:System.String  
 Pipeline:  
 -> CircularDependencyDetectorMiddleware  
 -> ScopeSelectionMiddleware  
 -> SharingMiddleware  
 -> RegistrationPipelineInvokeMiddleware  
 -> ActivatorErrorHandlingMiddleware  
 -> DisposalTrackingMiddleware  
 -> λ:System.String  
 <- λ:System.String  
 <- DisposalTrackingMiddleware  
 <- ActivatorErrorHandlingMiddleware  
 <- RegistrationPipelineInvokeMiddleware  
 <- SharingMiddleware  
 <- ScopeSelectionMiddleware  
 <- CircularDependencyDetectorMiddleware  
 }  
 Resolve Request Succeeded; result instance was HelloWorld  
 }  
 Operation Succeeded; result instance was HelloWorld

如你所见，跟踪非常详细 - 可以看到解析操作经过的整个 中间件管道 ，可以看到激活器（在这种情况下是一个委托），还可以看到结果实例。

这在尝试解决复杂的解析问题时非常有帮助，尽管跟踪越复杂，信息量越大，可能会让人感到压力。

错误跟踪将包括错误发生的位置并表明失败：

Resolve Operation Starting  
 {  
 Resolve Request Starting  
 {  
 Service: System.String  
 Component: λ:System.String  
 Pipeline:  
 -> CircularDependencyDetectorMiddleware  
 -> ScopeSelectionMiddleware  
 -> SharingMiddleware  
 -> RegistrationPipelineInvokeMiddleware  
 -> ActivatorErrorHandlingMiddleware  
 -> DisposalTrackingMiddleware  
 -> λ:System.String  
 X- λ:System.String  
 X- DisposalTrackingMiddleware  
 X- ActivatorErrorHandlingMiddleware  
 X- RegistrationPipelineInvokeMiddleware  
 X- SharingMiddleware  
 X- ScopeSelectionMiddleware  
 X- CircularDependencyDetectorMiddleware  
 }  
 Resolve Request FAILED  
 System.DivideByZeroException: Attempted to divide by zero.  
 at MyProject.MyNamespace.MyMethod.<>c.<GenerateSimpleTrace>b\_\_6\_0(IComponentContext x) in /path/to/MyCode.cs:line 39  
 at Autofac.RegistrationExtensions.<>c\_\_DisplayClass39\_0`1.<Register>b\_\_0(IComponentContext c, IEnumerable`1 p)  
 at Autofac.Builder.RegistrationBuilder.<>c\_\_DisplayClass0\_0`1.<ForDelegate>b\_\_0(IComponentContext c, IEnumerable`1 p)  
 at Autofac.Core.Activators.Delegate.DelegateActivator.ActivateInstance(IComponentContext context, IEnumerable`1 parameters)  
 at Autofac.Core.Activators.Delegate.DelegateActivator.<ConfigurePipeline>b\_\_2\_0(ResolveRequestContext ctxt, Action`1 next)  
 at Autofac.Core.Resolving.Middleware.DelegateMiddleware.Execute(ResolveRequestContext context, Action`1 next)  
 at Autofac.Core.Resolving.Pipeline.ResolvePipelineBuilder.<>c\_\_DisplayClass14\_0.<BuildPipeline>b\_\_1(ResolveRequestContext ctxt)  
 at Autofac.Core.Resolving.Middleware.DisposalTrackingMiddleware.Execute(ResolveRequestContext context, Action`1 next)  
 at Autofac.Core.Resolving.Pipeline.ResolvePipelineBuilder.<>c\_\_DisplayClass14\_0.<BuildPipeline>b\_\_1(ResolveRequestContext ctxt)  
 at Autofac.Core.Resolving.Middleware.ActivatorErrorHandlingMiddleware.Execute(ResolveRequestContext context, Action`1 next)  
 }  
 Operation FAILED

**注意** 返回到中间件的行程如何变为 X-？我们知道错误发生在执行 lambda 时。你可以使用这些提示确切地看到问题出在哪里。

### DOT 图形追踪器

除了 DefaultDiagnosticTracer，我们还提供了 Autofac.Diagnostics.DotGraph 包中的图形追踪器。

如果你添加对这个包的引用，你将能够使用 [DOT 语言](https://www.koudingke.cn/go?link=https%3a%2f%2fgraphviz.org%2fdoc%2finfo%2flang.html) 以视觉方式追踪完整的依赖树。然后，你可以使用像 [Graphviz](https://www.koudingke.cn/go?link=https%3a%2f%2fgraphviz.org%2f) 这样的工具渲染图像。

首先，就像使用 DefaultDiagnosticTracer 一样，将它注册到你的容器中。这次，跟踪输出将是 DOT 图形格式。

var containerBuilder = new ContainerBuilder();  
 containerBuilder.RegisterType<Component>().As<IService>();  
 var container = containerBuilder.Build();  
 var tracer = new DotDiagnosticTracer();  
 tracer.OperationCompleted += (sender, args) =>  
 {  
   
 var path = Path.Combine(Path.GetTempPath(), $"{Guid.NewGuid()}.dot");  
 using var file = new StreamWriter(path);  
 file.WriteLine(args.TraceContent);  
 };  
 container.SubscribeToDiagnostics(tracer);  
 using var scope = container.BeginLifetimeScope();  
 scope.Resolve<IService>();

假设你有一个简单的 lambda，它注册一个字符串。

var builder = new ContainerBuilder();  
 builder.Register(ctx => "HelloWorld");  
 var container = builder.Build();

DOT 图形追踪器的输出看起来像这样（确实很乱）：

digraph G {  
 label=<string<br/><font point-size="8">Operation #1>;  
 labelloc=t  
 na58baa0161f74ca8a74d3481aff7d182 [shape=component,label=<  
 <table border='0' cellborder='0' cellspacing='0'>  
 <tr><td port='nb569aeb076c94321a3c17b56bf16fd2c'>string</td></tr>  
 <tr><td><font point-size="10">Component: λ:string</td></tr>  
 </table>  
 >];  
 }

不过，假设你将这些信息保存到文件中，然后用 Graphviz 将其转换成 PNG：

dot -Tpng -O my-trace.dot

输出的图形看起来像这样：

![fig:](data:text/html; charset=utf-8;base64,PCFET0NUWVBFIEhUTUwgUFVCTElDICItLy9JRVRGLy9EVEQgSFRNTCAyLjAvL0VOIj48aHRtbD48aGVhZD48dGl0bGU+NDAzIEZvcmJpZGRlbjwvdGl0bGU+PC9oZWFkPjxib2R5PjxoMT40MDMgRm9yYmlkZGVuPC9oMT48cD5Zb3UgZG9uJ3QgaGF2ZSBwZXJtaXNzaW9uIHRvIGFjY2VzcyB0aGUgVVJMIG9uIHRoaXMgc2VydmVyLjwvcD48cD5kZW5pZWQgYnkgVUEgQUNMID0gYmxhY2tsaXN0PC9wPjxoci8+UG93ZXJlZCBieSBUZW5naW5lIDxici8+Q0ROIFJlcXVlc3QgSWQ6IGRkODJjMWI3MTcyNjQ5MTU4OTQ5NTEyNTdlPC9ib2R5PjwvaHRtbD4=)

现在看起来有些意思了。我们可以看到这是对一个字符串的解析，并且是由一个 lambda 来完成的。

但是，对于更复杂的场景呢？以下是复杂解析图的一个例子。

![fig:](data:text/html; charset=utf-8;base64,PCFET0NUWVBFIEhUTUwgUFVCTElDICItLy9JRVRGLy9EVEQgSFRNTCAyLjAvL0VOIj48aHRtbD48aGVhZD48dGl0bGU+NDAzIEZvcmJpZGRlbjwvdGl0bGU+PC9oZWFkPjxib2R5PjxoMT40MDMgRm9yYmlkZGVuPC9oMT48cD5Zb3UgZG9uJ3QgaGF2ZSBwZXJtaXNzaW9uIHRvIGFjY2VzcyB0aGUgVVJMIG9uIHRoaXMgc2VydmVyLjwvcD48cD5kZW5pZWQgYnkgVUEgQUNMID0gYmxhY2tsaXN0PC9wPjxoci8+UG93ZXJlZCBieSBUZW5naW5lIDxici8+Q0ROIFJlcXVlc3QgSWQ6IGRkODJjMTk1MTcyNjQ5MTU4OTcwNDEwNjBlPC9ib2R5PjwvaHRtbD4=)

从这个图中，我们可以获取很多信息：

* 需要解析 IHandler 和 IService1 ，它们都需要 IService2 ，并且使用了一个单例实例来满足需求。这意味着它可能是单例，也可能是每个生命周期范围一个实例。
* IService1 和 IService2 都需要 IService3 ，并且每个实例都会创建一个新的 IService3 实例。
* IService3被装饰了——看它如何向下链接到看起来更像一个盒子的节点。这表明有装饰器在起作用。你可以在这个框中看到组件（装饰器）和目标（被装饰的对象）。
* IService3 的构造函数参数需要 ILifetimeScope 。

最后一个参数 —— ILifetimeScope —— 意味着 IService3可能会在代码内部进行服务定位（手动解析）。如果你真的想知道完整的链路，可能需要将这个图与其他图关联起来。但是怎么做呢？

**注意** 顶部有一个 “操作 #1” 计数器——每次通过追踪器的解析操作都会增加这个计数器。你可以查找计数值较大的跟踪，然后做一些手动关联。不幸的是，这就是我们能做到的极限，因为每个解析都是独立的——服务定位会打断链路。你不能假设与生命周期范围关联的所有解析是相关的，例如，可能整个应用程序的所有解析都来自同一个范围。

错误也会被高亮显示，以便你能看到错误发生在哪里。

![fig:](data:text/html; charset=utf-8;base64,PCFET0NUWVBFIEhUTUwgUFVCTElDICItLy9JRVRGLy9EVEQgSFRNTCAyLjAvL0VOIj48aHRtbD48aGVhZD48dGl0bGU+NDAzIEZvcmJpZGRlbjwvdGl0bGU+PC9oZWFkPjxib2R5PjxoMT40MDMgRm9yYmlkZGVuPC9oMT48cD5Zb3UgZG9uJ3QgaGF2ZSBwZXJtaXNzaW9uIHRvIGFjY2VzcyB0aGUgVVJMIG9uIHRoaXMgc2VydmVyLjwvcD48cD5kZW5pZWQgYnkgVUEgQUNMID0gYmxhY2tsaXN0PC9wPjxoci8+UG93ZXJlZCBieSBUZW5naW5lIDxici8+Q0ROIFJlcXVlc3QgSWQ6IGRkODJjMWExMTcyNjQ5MTU4OTkxNTUwNDFlPC9ib2R5PjwvaHRtbD4=)

在这种情况下，你可以看到失败的地方，红色粗体突出显示。你还可以看到异常类型和消息。

### 自定义追踪器

使用 System.Diagnostics.DiagnosticSource，Autofac 允许你创建自定义追踪器，处理各种事件并生成你感兴趣的任何数据。

整体管道中的事件按照以下顺序发生：

中间件可能会启动额外的解析请求；而且管道中有多个中间件项。有关更多详细信息，请参阅 管道 页面。

如果你想追踪整个操作，就像 DefaultDiagnosticTracer 一样，可以从 Autofac.Diagnostics.OperationDiagnosticTracerBase 类开始。DefaultDiagnosticTracer 就是基于这个类构建的。它有意地监听所有解析事件，从头到尾，一次跟踪一个完整操作。你最好的例子是查看 DefaultDiagnosticTracer 的 [源代码](https://www.koudingke.cn/go?link=https%3a%2f%2fgithub.com%2fautofac%2fAutofac%2fblob%2fdca791ca0dbd1aa1cb0ad821539381df403d6d52%2fsrc%2fAutofac%2fDiagnostics%2fDefaultDiagnosticTracer.cs) 。由于要处理的事件很多，需要捕获的数据也很多。

你可以稍微控制一些，只追踪某些事件，使用 Autofac.Diagnostics.DiagnosticTracerBase 。这是一个 DiagnosticListener ，它为事件添加了一些强类型解析，帮助你编写较少的代码。以下是一个在解析操作开始时将日志写入控制台的追踪器示例：

下面是一个追踪完整操作并只保留类似 DefaultDiagnosticTracer 的简单数据堆栈的示例，但没有那么花哨。

public class ConsoleOperationTracer : DiagnosticTracerBase  
 {  
 public ConsoleOperationTracer()  
 : base()  
 {  
 EnableBase("Autofac.Operation.Start");  
 }  
 protected override void OnOperationStart(OperationStartDiagnosticData data)  
 {  
 Console.WriteLine("Operation starting.");  
 }  
 }

现在你可以使用你的自定义追踪器。它不会引发任何事件，但会记录你想要的内容。

var containerBuilder = new ContainerBuilder();  
 containerBuilder.RegisterType<Component>().As<IService>();  
 var container = containerBuilder.Build();  
 container.SubscribeToDiagnostics<ConsoleOperationTracer>();

如果你想要**更大的控制权**，你可以利用 System.Diagnostics.DiagnosticListener 默认使用的 IObserver<KeyValuePair<string, object>> 支持。以下是同样的控制台日志监听器的这种格式：

public class ConsoleOperationTracer : IObserver<KeyValuePair<string, object>>  
 {  
 public void OnCompleted()  
 {  
 }  
 public void OnError(Exception error)  
 {  
 }  
 public void OnNext(KeyValuePair<string, object> value)  
 {  
   
   
   
 Console.WriteLine("Operation starting.");  
 }  
 }

如你所见，如果你深入底层，可以编写非常紧密、性能良好的代码。

当你达到这个程度时，你可以独立于追踪器控制事件订阅。你必须直接将追踪器注册到容器的 DiagnosticSource。

var tracer = new ConsoleOperationTracer();  
 container.DiagnosticSource.Subscribe(tracer, e => e == "Autofac.Operation.Start");

### 符号和源代码

Symbols and Sources

Autofac 包已更新以使用 [Source Link](https://github.com/dotnet/sourcelink)，以便你可以直接从代码中调试到 Autofac 源代码。包可能包含符号直接在里面，也可能在 [NuGet符号服务器](https://docs.microsoft.com/en-us/nuget/create-packages/symbol-packages-snupkg#nugetorg-symbol-server)中。

**在 Visual Studio 中**，有启用搜索 NuGet 符号服务器的选项。[请参阅微软文档，了解如何配置 Visual Studio 以使符号服务器工作。](https://docs.microsoft.com/en-us/visualstudio/debugger/specify-symbol-dot-pdb-and-source-files-in-the-visual-studio-debugger)

**在 VS Code 中**，你可能需要在 settings.json 或 launch.json 中设置调试选项。

要在单元测试调试中启用符号，settings.json的块看起来像这样：

{  
 "csharp.unitTestDebuggingOptions": {  
 "symbolOptions": {  
 "searchMicrosoftSymbolServer": true,  
 "searchNuGetOrgSymbolServer": true  
 }  
 }  
 }

要使用符号启动应用程序，launch.json可能看起来像这样：

{  
 "configurations": [  
 {  
 "console": "internalConsole",  
 "cwd": "${workspaceFolder}/src/MyProject",  
 "env": {  
 "ASPNETCORE\_ENVIRONMENT": "Development",  
 "ASPNETCORE\_URLS": "https://localhost:5000",  
 "COMPlus\_ReadyToRun": "0",  
 "COMPlus\_ZapDisable": "1"  
 },  
 "justMyCode": false,  
 "name": "Launch with SourceLink (Development)",  
 "preLaunchTask": "build",  
 "program": "${workspaceFolder}/src/MyProject/bin/Debug/net6.0/MyProject.dll",  
 "request": "launch",  
 "serverReadyAction": {  
 "action": "openExternally",  
 "pattern": "\\bNow listening on:\\s+(https?://\\S+)",  
 "uriFormat": "%s"  
 },  
 "stopAtEntry": false,  
 "suppressJITOptimizations": true,  
 "symbolOptions": {  
 "searchMicrosoftSymbolServer": true,  
 "searchNuGetOrgSymbolServer": true  
 },  
 "type": "coreclr"  
 }  
 ],  
 "version": "0.2.0"  
 }

# 示例

## 示例 repo

Autofac 提供了一个 [GitHub 仓库](https://github.com/autofac/Examples)，展示了许多不同的集成工作方式。本网站的文档在可能的情况下链接到特定的相关示例。

[查看该仓库，以便更好地理解 Autofac 的实际应用。](https://github.com/autofac/Examples)

## log4net 集成中间件

log4net Integration Middleware

尽管没有专门针对 log4net 的特定库，但你可以通过简单的中间件和一个小型自定义模块轻松注入 log4net.ILog对象。Log4NetMiddleware还是使用 管道中间件 的一个好例子。

以下是根据激活组件类型注入 ILog参数的示例中间件。此中间件处理构造函数和属性注入。

public class Log4NetMiddleware : IResolveMiddleware  
{  
 public PipelinePhase Phase => PipelinePhase.ParameterSelection;  
 public void Execute(ResolveRequestContext context, Action<ResolveRequestContext> next)  
 {  
   
 context.ChangeParameters(context.Parameters.Union(  
 new[]  
 {  
 new ResolvedParameter(  
 (p, i) => p.ParameterType == typeof(ILog),  
 (p, i) => LogManager.GetLogger(p.Member.DeclaringType)  
 ),  
 }));  
   
 next(context);  
   
 if (context.NewInstanceActivated)  
 {  
 var instanceType = context.Instance.GetType();  
 var properties = instanceType  
 .GetProperties(BindingFlags.Public | BindingFlags.Instance)  
 .Where(p => p.PropertyType == typeof(ILog) && p.CanWrite && p.GetIndexParameters().Length == 0);  
   
 foreach (var propToSet in properties)  
 {  
 propToSet.SetValue(context.Instance, LogManager.GetLogger(instanceType), null);  
 }  
 }  
 }  
}

**性能提示**：当前，调用 LogManager.GetLogger(type)会带来轻微的性能影响，因为内部日志管理器会锁定日志器集合以获取合适的日志器。对于中间件的增强可以添加围绕日志实例的缓存，这样在 LogManager调用中避免锁的开销。

接下来是一个简单的 MiddlewareModule（不特定于日志），它为每个注册项添加一个单一的中间件实例到管道中。

public class MiddlewareModule : Autofac.Module  
 {  
 private readonly IResolveMiddleware middleware;  
 public MiddlewareModule(IResolveMiddleware middleware)  
 {  
 this.middleware = middleware;  
 }  
 protected override void AttachToComponentRegistration(IComponentRegistryBuilder componentRegistryBuilder, IComponentRegistration registration)  
 {  
   
 registration.PipelineBuilding += (sender, pipeline) =>  
 {  
   
 pipeline.Use(middleware);  
 };  
 }  
 }

# Autofac 常见问题

Frequently Asked Questions

Faq

## 1.如何处理每个请求的生命周期？

How do I work with per-request lifetime scope?

对于具有请求/响应语义的应用（如 ASP.NET MVC或 Web API），你可以注册依赖项为 “每个请求一个实例” ，这意味着在应用程序处理的每个请求中，都会得到给定依赖项的一个实例，并且该实例与单个请求的生命周期相关联。

要理解每个请求的生命周期，首先需要了解 依赖项生命周期范围的一般工作原理 。一旦你明白了依赖项生命周期范围的工作方式，处理每个请求的生命周期就很简单了。

### 关于 ASP.NET Core

如 ASP.NET Core 集成文档 所述，**ASP.NET Core 没有特定的每个请求的生命周期。**所有内容都注册为 InstancePerLifetimeScope()，而不是 InstancePerRequest()。

### 注册按请求的依赖项

当你想要注册为按请求的依赖项时，使用 InstancePerRequest() 注册扩展方法：

var builder = new ContainerBuilder();  
 builder.RegisterType<ConsoleLogger>()  
 .As<ILogger>()  
 .InstancePerRequest();  
 var container = builder.Build();

每当你的应用程序收到一个入站请求时，你都会得到组件的新实例。处理请求级生命周期范围的创建和清理通常通过应用类型的 Autofac 应用集成库 来完成。

### 如何处理每个请求的生命周期

每个请求的生命周期利用了 标记生命周期范围和“根据匹配生命周期范围的实例”机制 。Autofac 应用集成库会针对不同的应用类型进行挂钩，在入站请求时，它们会创建一个带有标识其为请求生命周期范围的嵌套生命周期范围：

+--------------------------+  
 | Autofac Container |  
 | |  
 | +----------------------+ |  
 | | Tagged Request Scope | |  
 | +----------------------+ |  
 +--------------------------+

当你将组件注册为 InstancePerRequest()时，你告诉 Autofac 在标记为请求范围的生命周期范围内查找并从那里解析组件。这样，如果在单个请求期间存在单元工作生命周期范围，那么按请求的依赖项将在请求期间共享：

+----------------------------------------------------+  
 | Autofac Container |  
 | |  
 | +------------------------------------------------+ |  
 | | Tagged Request Scope | |  
 | | | |  
 | | +--------------------+ +--------------------+ | |  
 | | | Unit of Work Scope | | Unit of Work Scope | | |  
 | | +--------------------+ +--------------------+ | |  
 | +------------------------------------------------+ |  
 +----------------------------------------------------+

请求范围被标记为常量值 Autofac.Core.Lifetime.MatchingScopeLifetimeTags.RequestLifetimeScopeTag，它等于字符串 AutofacWebRequest。如果找不到请求生命周期范围，你会得到一个 DependencyResolutionException，告诉你找不到请求生命周期范围。

有关此异常的调试提示，请参阅下面的 “调试” 部分。

### 不需要请求的跨应用共享依赖项

你可能会遇到一个常见情况，即有一个单独的 Autofac 模块 ，它执行一些依赖项注册，并希望在两个应用程序之间共享这个模块——一个是支持按请求的（如 Web API 应用），另一个不支持（如控制台应用程序或 Windows 服务）。

**How do you register dependencies as per-request and allow registration sharing?**

**如何按请求注册依赖项，并允许注册被共享？**

这个问题有几个潜在的解决方案。

**选项 1：** 将 InstancePerRequest() 注册更改为 InstancePerLifetimeScope()。大多数应用程序不会在任何地方创建自己的子生命周期范围；相反，唯一真正创建的子生命周期范围是请求生命周期。如果你的应用程序就是这样，那么 InstancePerRequest() 和 InstancePerLifetimeScope() 实际上会变得等效。你会得到相同的行为。在不支持按请求语义的应用程序中，你可以根据需要为组件共享创建子生命周期范围。

var builder = new ContainerBuilder();  
 builder.RegisterType<ConsoleLogger>()  
 .As<ILogger>()  
 .InstancePerLifetimeScope();  
 var container = builder.Build();

**选项** **2：** 设置注册模块，以便根据参数指示使用哪种生命周期范围注册类型。

public class LoggerModule : Module  
 {  
 private bool \_perRequest;  
 public LoggerModule(bool supportPerRequest)  
 {  
 this.\_perRequest = supportPerRequest;  
 }  
 protected override void Load(ContainerBuilder builder)  
 {  
 var reg = builder.RegisterType<ConsoleLogger>().As<ILogger>();  
 if(this.\_perRequest)  
 {  
 reg.InstancePerRequest();  
 }  
 else  
 {  
 reg.InstancePerLifetimeScope();  
 }  
 }  
 }

**选项** **3：** 更复杂但更复杂的第三个选项是在自然不支持这些语义的应用程序中实现自定义的按请求语义。例如，Windows 服务不一定有按请求的语义，但如果它自己托管了一个接受请求并提供响应的自定义服务，你可以为每个请求添加按请求的生命周期范围，并启用对按请求依赖项的支持。更多关于这一点的信息，请参阅 “自定义语义” 部分。

### 测试带有按请求依赖项的应用

如果你的应用程序注册了按请求的依赖项，你可能希望在单元测试中重用注册逻辑来设置依赖项。当然，你会发现你的单元测试没有请求生命周期范围，因此你会遇到一个 DependencyResolutionException，表明找不到 AutofacWebRequest范围。如何在测试环境中使用这些注册？

**选项** **1：** 为每个特定测试套件创建自定义注册。特别是在单元测试环境中，你可能不应该为测试设置整个真实的运行时环境——你应该为外部所需的依赖项提供测试双。考虑模拟依赖项，而不是在单元测试环境中执行完整的共享注册。

**选项** **2：** 查看“共享依赖项”部分中的注册共享选择。你的单元测试可以被视为“不支持按请求注册的应用程序”，因此使用允许在不同应用程序类型之间共享的机制可能是合适的。

**选项** **3：** 实现一个测试用的“请求”。这里的意图是，在测试运行之前创建一个真正的 Autofac 生命周期范围，并带有 AutofacWebRequest标签，运行测试，然后释放假的“请求”范围——就像实际运行了一个完整请求一样。这有点复杂，而且方法取决于应用程序类型。

### 模拟 MVC 请求范围

**Autofac ASP.NET MVC 集成** 使用 ILifetimeScopeProvider 实现以及 AutofacDependencyResolver 动态创建按需请求范围。要模拟 MVC 请求范围，你需要提供一个测试 ILifetimeScopeProvider，它不涉及实际的 HTTP 请求。一个简单的版本可能如下所示：

public class SimpleLifetimeScopeProvider : ILifetimeScopeProvider  
 {  
 private readonly IContainer \_container;  
 private ILifetimeScope \_scope;  
 public SimpleLifetimeScopeProvider(IContainer container)  
 {  
 this.\_container = container;  
 }  
 public ILifetimeScope ApplicationContainer  
 {  
 get { return this.\_container; }  
 }  
 public void EndLifetimeScope()  
 {  
 if (this.\_scope != null)  
 {  
 this.\_scope.Dispose();  
 this.\_scope = null;  
 }  
 }  
 public ILifetimeScope GetLifetimeScope(Action<ContainerBuilder> configurationAction)  
 {  
 if (this.\_scope == null)  
 {  
 this.\_scope = (configurationAction == null)  
 ? this.ApplicationContainer.BeginLifetimeScope(MatchingScopeLifetimeTags.RequestLifetimeScopeTag)  
 : this.ApplicationContainer.BeginLifetimeScope(MatchingScopeLifetimeTags.RequestLifetimeScopeTag, configurationAction);  
 }  
 return this.\_scope;  
 }  
 }

当你从构建的应用容器创建 AutofacDependencyResolver时，你需要手动指定简单的生命周期范围提供程序。确保在测试运行之前设置了解析器，然后在测试运行后清理假请求范围。在 NUnit 中，它看起来像这样：

private IDependencyResolver \_originalResolver = null;  
 private ILifetimeScopeProvider \_scopeProvider = null;  
 [TestFixtureSetUp]  
 public void TestFixtureSetUp()  
 {  
   
 this.\_scopeProvider = new SimpleLifetimeScopeProvider(container);  
 var resolver = new AutofacDependencyResolver(container, provider);  
 this.\_originalResolver = DependencyResolver.Current;  
 DependencyResolver.SetResolver(resolver);  
 }  
 [TearDown]  
 public void TearDown()  
 {  
   
 this.\_scopeProvider.EndLifetimeScope();  
 }  
 [TestFixtureTearDown]  
 public void TestFixtureTearDown()  
 {  
   
 DependencyResolver.SetResolver(this.\_originalResolver);  
 }

### 模拟 Web API 请求范围

在 Web API 中，请求的生命周期实际上作为 HttpRequestMessage 对象的一部分，作为 ILifetimeScope 对象在整个系统中移动。要模拟一个请求生命周期，你只需要从正在处理的测试消息中获取 ILifetimeScope。

在测试设置期间，你应该像在应用程序中那样构建依赖注入解析器，并将其与 HttpConfiguration对象关联起来。在每个测试中，根据正在测试的用例创建适当的 HttpRequestMessage，然后使用内置的 Web API 扩展方法将配置附加到消息并从消息中获取请求生命周期。

在 NUnit 中，它可能看起来像这样：

private HttpConfiguration \_configuration = null;  
 [TestFixtureSetUp]  
 public void TestFixtureSetUp()  
 {  
   
 this.\_configuration = new HttpConfiguration  
 {  
 DependencyResolver = new AutofacWebApiDependencyResolver(container)  
 }  
 }  
 [TestFixtureTearDown]  
 public void TestFixtureTearDown()  
 {  
   
 this.\_configuration.Dispose();  
 }  
 [Test]  
 public void MyTest()  
 {  
   
 using (var message = CreateTestHttpRequestMessage())  
 {  
 message.SetConfiguration(this.\_configuration);  
   
 }  
 }

### 解决每个请求依赖问题的技巧

在处理每个请求依赖时，有一些需要注意的地方。这里有一些故障排查帮助。

#### 没有匹配 'AutofacWebRequest' 标签的范围

当人们开始使用每个请求的生命周期时，一个非常常见的异常是：

No scope with a tag matching 'AutofacWebRequest' is visible from the scope  
in which the instance was requested.  
  
If you see this during execution of a web application, it generally  
indicates that a component registered as per-HTTP request is being  
requested by a SingleInstance() component (or a similar scenario). Under  
the web integration always request dependencies from the dependency  
resolver or the request lifetime scope, never from the container itself.

没有与 'AutofacWebRequest' 标签匹配的范围可见于实例请求的范围。  
 如果在执行 Web 应用程序时看到此异常，通常表示注册为 HTTP 请求的组件正在被 `SingleInstance()` 组件（或其他类似场景）请求。在 Web 集成中，始终从依赖解析器或请求生命周期范围内请求依赖，而不是直接从容器本身请求。

这意味着应用程序试图解决一个已注册为 InstancePerRequest() 的依赖项，但没有任何请求生命周期。

常见原因包括：

* 应用程序注册被跨应用程序类型共享。
* 单元测试运行时使用实际应用程序注册，但没有模拟每个请求的生命周期。
* 有一个组件 *存活时间超过一个请求*，但它依赖一个 *只存活一个请求* 的组件。例如，单例组件依赖一个按请求注册的服务。
* 在 ASP.NET 应用程序启动时（如 Global.asax 中）运行的代码，或者在没有活跃请求的情况下使用依赖解析。
* 在没有请求语义的“后台线程”（如 ASP.NET MVC 的 DependencyResolver 进行服务定位）上运行的代码。

追踪问题的根源可能很困难。在很多情况下，你可能会查看正在解决的内容，并看到被解决的组件 *不是按请求注册的*，而且该组件使用的依赖项 *也不是按请求注册的*。在这种情况下，你可能需要沿着依赖链一路追踪。异常可能是来自依赖链深处的某个地方。通常，仔细检查堆栈跟踪可以帮助你。如果你正在使用 动态程序集(assembly)扫描 来定位 模块 进行注册，那么问题注册的来源可能不会立即明显。

在分析问题依赖链中的注册时，查看它们注册的生命周期范围。如果你有一个注册为 SingleInstance()的组件，但它（可能是间接地）消费了一个注册为 InstancePerRequest()的组件，那就是一个问题。SingleInstance()组件会在第一次解决时获取其依赖项，并且永远不会释放。如果这发生在应用程序启动时或在没有当前请求的后台线程上，你会看到这个异常。你可能需要调整一些组件的生命周期范围。再次强调，了解 依赖生命周期范围的一般工作原理 是非常有用的。

无论如何，在某个点上，*某些东西*正在寻找一个请求生命周期范围，但找不到。

如果你试图跨应用程序类型共享注册，请参阅 sharing-dependencies 部分。

如果你试图使用每个请求依赖进行单元测试，testing 和 sharing-dependencies 部分可以给你一些提示。

如果你的 ASP.NET MVC 应用程序中存在应用程序启动代码或后台线程尝试使用 DependencyResolver.Current - AutofacDependencyResolver 需要在 Web 上下文中进行依赖解决。当你尝试从解析器中解决问题时，它会尝试启动一个按请求生命周期范围并将其存储在当前 HttpContext旁边。如果没有当前上下文，事情就会失败。访问 AutofacDependencyResolver.Current 并不能绕过这个问题 - 当前解析器属性的工作方式是，它从当前 Web 请求范围中查找自身（这样做是为了允许与 Glimpse 等其他仪器机制一起工作）。

对于应用程序启动代码或后台线程，你可能需要考虑使用不同的服务定位机制，如 公共服务定位器 来避免按请求范围的需求。如果你这样做，你也需要查看 sharing-dependencies部分来更新你的组件注册，以便它们不一定需要按请求范围。

#### Web API 中的按请求过滤器依赖

如果你使用 Web API 集成 和 AutofacWebApiFilterProvider将依赖注入到动作过滤器中，你可能会注意到 **过滤器中的依赖项只被一次性解决，而不是按请求基础**。

这是 Web API 的一个局限性。Web API 内部创建过滤器实例并缓存它们，永远不会重新创建。这移除了任何可能存在的按请求基础的“挂钩”，可以在过滤器中做任何事情。

如果你需要在过滤器中按请求做某事，你需要使用服务定位，并手动从上下文在过滤器中获取请求生命周期范围。例如，一个 ActionFilterAttribute可能看起来像这样：

public class LoggingFilterAttribute : ActionFilterAttribute  
 {  
 public override void OnActionExecuting(HttpActionContext context)  
 {  
 var logger = context.Request.GetDependencyScope().GetService(typeof(ILogger)) as ILogger;  
 logger.Log("Executing action.");  
 }  
 }

使用这种服务定位机制，你甚至不需要 AutofacWebApiFilterProvider - 即使不使用 Autofac 也可以做到这一点。

### 实现自定义按请求语义

你可能有一个自定义的应用程序处理请求 - 类似于接收请求、执行一些工作并提供输出的 Windows 服务应用程序。在这种情况下，如果你正确地组织你的应用程序，你可以实现一个自定义机制，提供按请求注册和解决依赖的能力。你采取的步骤与其他支持按请求语义的自然应用程序类型中的步骤完全相同。

* **在应用程序启动时构建容器。** 进行注册，构建容器，并存储对全局容器的引用，供稍后使用。
* **当接收到逻辑请求时，创建一个请求生命周期范围。**请求生命周期范围应使用标签 Autofac.Core.Lifetime.MatchingScopeLifetimeTags.RequestLifetimeScopeTag 标记，这样你就可以使用标准注册扩展方法，如 InstancePerRequest()。这还将使你能够跨应用程序类型共享注册模块，如果你愿意的话。
* **将请求生命周期范围与请求关联。** 这意味着你需要在请求内部获取请求范围的能力，而不是有一个静态的全局变量保存“请求范围” - 这是一个线程问题。你需要像 ASP.NET 中的 HttpContext.Current或 WCF 中的 OperationContext.Current这样的构造，或者你需要将请求生命周期与实际传入请求信息（如 Web API）一起存储。
* **请求完成后释放请求生命周期。** 在处理完请求并发送响应后，需要调用 IDisposable.Dispose()在请求生命周期范围内，以确保内存清理和服务实例释放。
* **在应用程序结束时释放容器。** 当应用程序关闭时，调用全局应用程序容器的 IDisposable.Dispose()以确保所有托管资源得到正确释放，数据库等连接关闭。

如何实现取决于你的应用程序，因此无法提供一个“示例”。查看不同应用类型的集成库（如 MVC 和 Web API）的源代码来了解这些是如何做的，这是一个好方法。然后你可以采用这些模式并相应地进行调整，以适应你的应用程序需求。

**这是一个非常高级的过程。** 如果你不适当地释放东西，可能会引入内存泄漏；如果错误地将请求生命周期与请求关联起来，可能会导致线程问题。如果你走这条路，一定要做大量测试和性能分析，确保一切按预期工作。

## 2.如何根据上下文选择服务实现？

How do I pick a service implementation by context?

**如何根据上下文选择服务实现？**

有时，你可能需要注册多个 组件 ，它们都提供了相同的 服务 ，但希望在不同实例中选择使用哪个组件。让我们以一个简单的订单处理系统为例：

* **发货处理器**：负责将订单内容物理寄送。
* **通知处理器**：当订单状态改变时，向用户发送警报。

在这个系统中，发货处理器可能需要支持不同的“插件”来实现不同的配送方式，如邮政、UPS、FedEx 等。通知处理器可能也需要不同的 “插件” 来支持不同的通知方式，如电子邮件或短信。

初始设计可能如下所示：

public interface ISender  
 {  
 void Send(Destination dest, Content content);  
 }  
 public class PostalServiceSender : ISender { ... }  
 public class EmailNotifier : ISender { ... }  
 public class ShippingProcessor  
 {  
 public ShippingProcessor(ISender shippingStrategy) { ... }  
 }  
 public class CustomerNotifier  
 {  
 public CustomerNotifier(ISender notificationStrategy) { ... }  
 }

当你在 Autofac 中进行注册时，可能会像这样：

var builder = new ContainerBuilder();  
 builder.RegisterType<PostalServiceSender>().As<ISender>();  
 builder.RegisterType<EmailNotifier>().As<ISender>();  
 builder.RegisterType<ShippingProcessor>();  
 builder.RegisterType<CustomerNotifier>();  
 var container = builder.Build();

**问题来了：**如何确保发货处理器使用邮政服务策略，而客户通知器使用电子邮件策略？

### 方法 1：重新设计接口

Option 1: Redesign Your Interfaces

当你遇到多个组件实现相同服务但不能同等对待的情况时，**这通常是一个接口设计问题**。

从面向对象的角度来看，你应该遵循 [里氏替换原则](https://www.koudingke.cn/go?link=http%3a%2f%2fen.wikipedia.org%2fwiki%2fLiskov_substitution_principle)，这种情况就违反了这一原则。

换个角度看，假设我们有一个动物类的简单例子。有各种各样的动物，我们想创建一个专门表示鸟笼的特殊类，它只能容纳小型鸟类：

public abstract class Animal  
 {  
 public abstract string MakeNoise();  
 public abstract AnimalSize Size { get; }  
 }  
 public enum AnimalSize  
 {  
 Small, Medium, Large  
 }  
 public class HouseCat : Animal  
 {  
 public override string MakeNoise() { return "Meow!"; }  
 public override AnimalSize { get { return AnimalSize.Small; } }  
 }  
 public abstract class Bird : Animal  
 {  
 public override string MakeNoise() { return "Chirp!"; }  
 }  
 public class Parakeet : Bird  
 {  
 public override AnimalSize { get { return AnimalSize.Small; } }  
 }  
 public class BaldEagle : Bird  
 {  
 public override string MakeNoise() { return "Screech!"; }  
 public override AnimalSize { get { return AnimalSize.Large; } }  
 }

如果直接设计鸟笼类，可能会这样：

public class BirdCage  
{  
 public BirdCage(Animal animal)  
 {  
 if (!(animal is Bird) || animal.Size != AnimalSize.Small)  
 {  
   
 throw new NotSupportedException();  
 }  
 }  
}

**让鸟笼接受任何动物的设计并不合理。**至少应该限制为 “鸟”：

public class BirdCage  
 {  
 public BirdCage(Bird bird)  
 {  
 if (bird.Size != AnimalSize.Small)  
 {  
   
 throw new NotSupportedException();  
 }  
 }  
 }

通过稍微调整设计，我们可以使其更容易，并且只允许正确类型的鸟被使用：

public abstract class Bird : Animal  
 {  
 public override string MakeNoise() { return "Chirp!"; }  
 }  
 public abstract class PetBird : Bird  
 {  
   
 public sealed override AnimalSize { get { return AnimalSize.Small; } }  
 }  
 public class Parakeet : PetBird { }  
 public class BaldEagle : Bird  
 {  
 public override string MakeNoise() { return "Screech!"; }  
 public override AnimalSize { get { return AnimalSize.Large; } }  
 }

现在我们可以很容易地设计出只支持小型宠物鸟的鸟笼。我们只需在构造函数中使用正确的基类：

public class BirdCage  
{  
 public BirdCage(PetBird bird) { }  
}

这个例子虽然有些牵强附会，但原则依然适用——通过重新设计接口，我们可以确保鸟笼只接收预期的东西，而不会接收其他内容。

回到订单处理系统，虽然每个配送机制看起来只是“发送某物”，但其实它们发送的是不同类型的东西。也许有一个基础接口用于一般的“发送”，但可能需要一个中间层来区分发送的不同类型：

public interface ISender  
 {  
 void Send(Destination dest, Content content);  
 }  
 public interface IOrderSender : ISender { }  
 public interface INotificationSender : ISender { }  
 public class PostalServiceSender : IOrderSender { ... }  
 public class EmailNotifier : INotificationSender { ... }  
 public class ShippingProcessor  
 {  
 public ShippingProcessor(IOrderSender shippingStrategy) { ... }  
 }  
 public class CustomerNotifier  
 {  
 public CustomerNotifier(INotificationSender notificationStrategy) { ... }  
 }

**通过重新设计接口，我们无需“根据上下文选择依赖项”** ——我们使用类型来区分，并利用自动绑定的魔力在 解析过程 中发生。

**如果能对解决方案进行调整，这是推荐的方法。**

### 方法 2：换一种注册方式

Option 2: Change the Registrations

在 Autofac 中注册组件时，你可以使用 lambda 表达式而不是类型。你可以手动在那个上下文中将适当的类型与消费组件关联起来：

var builder = new ContainerBuilder();  
 builder.Register(ctx => new ShippingProcessor(new PostalServiceSender()));  
 builder.Register(ctx => new CustomerNotifier(new EmailNotifier()));  
 var container = builder.Build();

如果你想让发送者由 Autofac 自动解决，你可以同时暴露它们作为接口类型和它们自身的类型，然后在 lambda 表达式中解决它们：

var builder = new ContainerBuilder();  
 builder.RegisterType<PostalServiceSender>()  
 .As<ISender>()  
 .AsSelf();  
 builder.RegisterType<EmailNotifier>()  
 .As<ISender>()  
 .AsSelf();  
 builder.Register(ctx => new ShippingProcessor(ctx.Resolve<PostalServiceSender>()));  
 builder.Register(ctx => new CustomerNotifier(ctx.Resolve<EmailNotifier>()));  
 var container = builder.Build();

如果使用 lambda 机制感觉太 “手动” ，或者处理器对象需要大量参数，你可以 手动为注册项附加参数 ：

var builder = new ContainerBuilder();  
 builder.RegisterType<PostalServiceSender>()  
 .As<ISender>()  
 .AsSelf();  
 builder.RegisterType<EmailNotifier>()  
 .As<ISender>()  
 .AsSelf();  
 builder.RegisterType<ShippingProcessor>()  
 .WithParameter(  
 new ResolvedParameter(  
 (pi, ctx) => pi.ParameterType == typeof(ISender),  
 (pi, ctx) => ctx.Resolve<PostalServiceSender>()));  
 builder.RegisterType<CustomerNotifier>();  
 .WithParameter(  
 new ResolvedParameter(  
 (pi, ctx) => pi.ParameterType == typeof(ISender),  
 (pi, ctx) => ctx.Resolve<EmailNotifier>()));  
 var container = builder.Build();

使用参数方法，你在创建发送者和处理器时仍能获得 “自动绑定” 的好处，但在那些特定情况下，你可以指定非常具体的重写。

**如果你无法更改接口并且希望保持简单，这是推荐的方法。**

### 方法 3：使用键控服务

Option 3: Use Keyed Services

或许你可以修改注册，但你也使用了 模块 来注册许多不同的组件，无法通过类型直接将它们关联在一起。解决这个问题的一个简单方法是使用 键控服务 。

在这种情况下，Autofac 允许你为服务注册分配一个 “键” 或 “名称” ，并在另一个注册中根据该键进行解析。在注册发送者的模块中，你会为每个发送者分配适当的键；在注册处理器的模块中，你会在注册项中应用参数，以获取适当的键依赖项。

在注册发送器的模块中添加键名称：

public class SenderModule : Module  
 {  
 protected override void Load(ContainerBuilder builder)  
 {  
 builder.RegisterType<PostalServiceSender>()  
 .As<ISender>()  
 .Keyed<ISender>("order");  
 builder.RegisterType<EmailNotifier>()  
 .As<ISender>()  
 .Keyed<ISender>("notification");  
 }  
 }

在注册处理器的模块中，添加使用已知键的参数：

public class ProcessorModule : Module  
 {  
 protected override void Load(ContainerBuilder builder)  
 {  
 builder.RegisterType<ShippingProcessor>()  
 .WithParameter(  
 new ResolvedParameter(  
 (pi, ctx) => pi.ParameterType == typeof(ISender),  
 (pi, ctx) => ctx.ResolveKeyed<ISender>("order")));  
 builder.RegisterType<CustomerNotifier>();  
 .WithParameter(  
 new ResolvedParameter(  
 (pi, ctx) => pi.ParameterType == typeof(ISender),  
 (pi, ctx) => ctx.ResolveKeyed<ISender>("notification")));  
 }  
 }

现在，当处理器被解析时，它们会搜索键化的服务注册，并注入正确的实现。

*你可以为同一个键有多个服务*，所以如果你的发送器通过隐式支持的关系接受 IEnumerable ，这种情况也可以工作。只需在处理器注册中将参数设置为 ctx.ResolveKeyed<IEnumerable>("order") ，并使用适当的键注册每个发送器。

如果你有能力更改注册，并且不必对所有注册进行程序集(assembly)扫描，这是推荐的选择。

### 选项 4：使用元数据

Option 4: Use Metadata

如果你需要比 键控服务 更灵活的东西，或者你没有能力直接影响注册，你可能想要考虑使用 注册元数据 功能将合适的服务连接在一起。

你可以直接为注册关联元数据：

public class SenderModule : Module  
 {  
 protected override void Load(ContainerBuilder builder)  
 {  
 builder.RegisterType<PostalServiceSender>()  
 .As<ISender>()  
 .WithMetadata("SendAllowed", "order");  
 builder.RegisterType<EmailNotifier>()  
 .As<ISender>()  
 .WithMetadata("SendAllowed", "notification");  
 }  
 }

然后，你可以在消费者注册上使用元数据作为参数：

public class ProcessorModule : Module  
 {  
 protected override void Load(ContainerBuilder builder)  
 {  
 builder.RegisterType<ShippingProcessor>()  
 .WithParameter(  
 new ResolvedParameter(  
 (pi, ctx) => pi.ParameterType == typeof(ISender),  
 (pi, ctx) => ctx.Resolve<IEnumerable<Meta<ISender>>>()  
 .First(a => a.Metadata["SendAllowed"].Equals("order")).Value));  
 builder.RegisterType<CustomerNotifier>();  
 .WithParameter(  
 new ResolvedParameter(  
 (pi, ctx) => pi.ParameterType == typeof(ISender),  
 (pi, ctx) => ctx.Resolve<IEnumerable<Meta<ISender>>>()  
 .First(a => a.Metadata["SendAllowed"].Equals("notification")).Value));  
 }  
 }

（是的，这比使用键化服务稍微复杂一些，但你可能希望利用 元数据设施提供的灵活性 。）

**如果你无法更改发送者组件的注册，但允许更改对象定义，**你可以使用“属性元数据”机制向组件添加元数据。首先，创建自定义元数据属性：

[MetadataAttribute]  
 public class SendAllowedAttribute : Attribute  
 {  
 public string SendAllowed { get; set; }  
 public SendAllowedAttribute(string sendAllowed)  
 {  
 this.SendAllowed = sendAllowed;  
 }  
 }

然后，你可以将自定义元数据属性应用于发送者组件：

[SendAllowed("order")]  
 public class PostalServiceSender : IOrderSender { ... }  
 [SendAllowed("notification")]  
 public class EmailNotifier : INotificationSender { ... }

当你注册发送者时，请确保注册AttributedMetadataModule：

public class SenderModule : Module  
 {  
 protected override void Load(ContainerBuilder builder)  
 {  
 builder.RegisterType<PostalServiceSender>().As<ISender>();  
 builder.RegisterType<EmailNotifier>().As<ISender>();  
 builder.RegisterModule<AttributedMetadataModule>();  
 }  
 }

消费组件可以像平常一样使用元数据——属性属性的名称将成为元数据中的名称：

public class ProcessorModule : Module  
 {  
 protected override void Load(ContainerBuilder builder)  
 {  
 builder.RegisterType<ShippingProcessor>()  
 .WithParameter(  
 new ResolvedParameter(  
 (pi, ctx) => pi.ParameterType == typeof(ISender),  
 (pi, ctx) => ctx.Resolve<IEnumerable<Meta<ISender>>>()  
 .First(a => a.Metadata["SendAllowed"].Equals("order"))));  
 builder.RegisterType<CustomerNotifier>()  
 .WithParameter(  
 new ResolvedParameter(  
 (pi, ctx) => pi.ParameterType == typeof(ISender),  
 (pi, ctx) => ctx.Resolve<IEnumerable<Meta<ISender>>>()  
 .First(a => a.Metadata["SendAllowed"].Equals("notification"))));  
 }  
 }

对于你的消费组件，如果你不介意在参数定义中添加自定义 Autofac 属性，也可以使用属性元数据：

public class ShippingProcessor  
 {  
 public ShippingProcessor([WithMetadata("SendAllowed", "order")] ISender shippingStrategy) { ... }  
 }  
 public class CustomerNotifier  
 {  
 public CustomerNotifier([WithMetadata("SendAllowed", "notification")] ISender notificationStrategy) { ... }  
 }

如果你的消费组件使用了属性，你需要使用 WithAttributeFilter 注册它们：

public class ProcessorModule : Module  
 {  
 protected override void Load(ContainerBuilder builder)  
 {  
 builder.RegisterType<ShippingProcessor>().WithAttributeFilter();  
 builder.RegisterType<CustomerNotifier>().WithAttributeFilter();  
 }  
 }

再次强调，元数据机制非常灵活。你可以混合搭配将元数据与组件和服务消费者关联的方式——属性、参数等。有关 注册元数据 、注册参数 、解析参数 和 隐式支持的关系 （如 Meta关系）的信息，请参阅各自页面。

**如果你已经使用元数据或需要元数据提供的灵活性，这是推荐的选择。**

## 3.如何在 Web 应用中创建基于会话的生命周期范围？

How do I create a session-based lifetime scope in a web application?

在 ASP.NET 中，“每个请求一个实例”的概念是内置的，但你可能希望为某些对象创建 “每个会话一个实例” 。**这条路充满了危险，并且完全不被支持。** 由于这个问题经常被问到，我们提供了一些关于如何让它正常工作的信息，[这是基于StackOverflow上的一个答案](https://www.koudingke.cn/go?link=https%3a%2f%2fstackoverflow.com%2fquestions%2f11721919%2fmanaging-autofac-lifetime-scopes-per-session-and-request-in-asp-net-mvc-3%2f11726210%2311726210)，但如果它对你不起作用，或者你需要额外的支持来实现它，**你将只能自己解决**。

此外，**这些信息适用于经典 ASP.NET MVC，而不是 ASP.NET Core**，但仍然会遇到相同的挑战。很可能不会为 ASP.NET Core 更新这些内容。它也可能不会为 Web API、web 表单或其他任何其他集成更新。你需要根据需要从这里获取想法并进行调整。

### 这是一个坏主意的原因

在你开始之前，这里是你使用会话范围生命周期会遇到的一些挑战：

#### 内存占用

你最终会为系统中的每个用户都创建一个生命周期范围。虽然请求级别的范围很快出现并消失，但会话级别的范围可能会持续很长时间。如果你有很多会话范围内的项目，每个用户的内存使用量都会相当大。如果人们没有正确注销就“放弃”他们的会话，那么这些东西将会活得更久。

#### 不适合分布式环境

生命周期范围及其内容不可序列化。[看看LifetimeScope代码](https://www.koudingke.cn/go?link=https%3a%2f%2fgithub.com%2fautofac%2fAutofac%2fblob%2fdevelop%2fsrc%2fAutofac%2fCore%2fLifetime%2fLifetimeScope.cs)，它并没有标记为[Serializable]……即使标记了，存储在其中的解析对象也不一定都被标记为可序列化。这很重要，因为这意味着你的会话级别生命周期范围可能在一个具有内存会话的单个盒子上工作，但是**如果你部署到具有** **SQL** **会话或会话服务的农场，事情就会崩溃**，因为会话无法序列化你存储的范围。

如果你选择不序列化范围，那么在机器之间，每个用户都有不同的范围——这也是一个问题。

#### 会话不一致使用

会话并不总是重新激活。如果访问的处理器（例如，web 表单）没有实现 IRequiresSessionState，则不会重新激活会话（无论是否在进程内）。Web 表单和 MvcHandler默认实现了这一点，所以你不会遇到任何问题，但如果你有需要注入的自定义处理器，你会遇到一些问题，因为这些请求中的“会话”不存在。对于明确标记不需要会话的处理器（例如，出于性能原因），你也会有麻烦。

#### 不可靠的释放

Session*End 并不总是触发。*[*根据SessionStateModule.End文档*](https://www.koudingke.cn/go?link=https%3a%2f%2fmsdn.microsoft.com%2fen-us%2flibrary%2fsystem.web.sessionstate.sessionstatemodule.end.aspx)*，如果你使用进程外会话状态，实际上你不会收到 Session*End 事件，因此无法清理。

#### 如何做到这一点

假设你已经阅读了所有这些内容，并且想要这样做。

**至少在经典** **ASP.NET MVC** **中，你需要实现自己的Autofac.Integration.Mvc.ILifetimeScopeProvider。这个接口决定了何时/在哪里生成请求生命周期范围。**

**具体如何实现将取决于你自己。** 这是因为上述所有挑战。例如，你将在哪里保存会话级别的生命周期范围？它是附加到实际会话（由于序列化而存在问题）吗？还是在某个静态字典中？还是你想在其他地方保存这些引用？这些问题都不能在这里回答——这很大程度上都是“读者的练习”。

默认的 ILifetimeScopeProvider 实现，Autofac.Integration.Mvc.RequestLifetimeScopeProvider，负责按请求创建、管理和维护生命周期范围。你可以在这里浏览 RequestLifetimeScopeProvider 的代码：[这里](https://www.koudingke.cn/go?link=https%3a%2f%2fgithub.com%2fautofac%2fAutofac.Mvc%2fblob%2fdevelop%2fsrc%2fAutofac.Integration.Mvc%2fRequestLifetimeScopeProvider.cs) ，如果你打算这样做，这是最好的示例代码，展示了此类提供程序的责任。

ILifetimeScopeProvider的实现将是你……

* 为用户找到（或创建）会话生命周期范围
* 将请求生命周期范围作为会话生命周期范围的子范围创建
* 在请求结束时释放请求生命周期范围

这可能也是从设计角度看，你想要在何处释放会话生命周期范围，但它可能很棘手，因为提供程序不会自动接收到会话结束事件。

一旦你有了 ILifetimeScopeProvider ，你就可以在设置依赖注入解析器时使用它。

var scopeProvider = new MyCustomLifetimeScopeProvider(container, configAction);  
 var resolver = new AutofacDependencyResolver(container, scopeProvider);  
 DependencyResolver.SetResolver(resolver);

**你还需要挂钩到Session\_End事件**（例如，在 Global.asax 或 MvcApplication 类中）来释放会话范围。再次，具体如何做完全取决于你，因为 ILifetimeScopeProvider 不会接收任何与会话相关的事件。

## 4.为什么我的程序集在 IIS 重启后没有被扫描？

Why aren't my assemblies getting scanned after IIS restart?

有时你希望使用 程序集(assembly)扫描 功能来加载 IIS 托管应用中的插件。

当在 IIS 中托管应用程序时，应用程序首次启动时所有程序集都会加载到 AppDomain中。然而，**当** **IIS** **重新编译** **AppDomain时，这些程序集将只在需要时加载。**

要解决这个问题，请使用 System.Web.Compilation.BuildManager 类上的 [GetReferencedAssemblies()](https://www.koudingke.cn/go?link=https%3a%2f%2fmsdn.microsoft.com%2fen-us%2flibrary%2fsystem.web.compilation.buildmanager.getreferencedassemblies.aspx) 方法获取引用的程序集列表：

var assemblies = BuildManager.GetReferencedAssemblies().Cast<Assembly>();

这会迫使引用的程序集立即加载到 AppDomain 中，从而使它们可供模块扫描使用。

或者，而不是使用 AppDomain.CurrentDomain.GetAssemblies()进行扫描，**手动从文件系统加载程序集**。手动加载会强制它们进入 AppDomain，然后你可以开始扫描。

## 5.如何条件性注册组件？

How do I conditionally register components?

有时候你可能希望在运行时决定容器中注册哪些内容，可能是基于环境变量或应用程序参数。这里有一些选择方案...

### 使用配置

**Use Configuration**

Autofac 提供了 配置机制 ，允许你通过配置文件指定注册项。你可以设置不同的配置文件以供不同环境和/或不同参数使用，然后在应用启动时选择合适的配置文件进行读取并注册。

如果你选择使用 Microsoft.Extensions.Configuration 抽象（Autofac 4.0+），甚至可以直接将配置表示为环境变量。有关如何在环境变量中表示配置的文档，请参阅 Microsoft.Extensions.Configuration 文档。

### 使用模块

**Use Modules**

Autofac 模块是一种程序化配置机制，它将注册项打包在一起。你可以向模块添加参数（如构造函数参数或属性），使得根据提供的值（从运行时环境读取）模块注册不同的内容或行为有所不同。

关于此的示例可以在 **Autofac 模块** 文档中找到。

### 使用 lambda 表达式注册

Lambda Registrations

你可以使用 lambda 表达式 来 注册组件 ，并在注册时直接做出运行时决策。需要注意的是，这可能会对性能产生影响，具体取决于运行时检查的开销以及执行频率，但它是一个选择。

builder.RegisterType(c =>  
 {  
 var environment = Environment.GetEnvironmentVariable("environment\_name");  
 if (environment == "DEV")  
 {  
 return new DevelopmentObject();  
 }  
 else  
 {  
 return new ProductionObject();  
 }  
 })  
 .As<IMyObject>()  
 .SingleInstance();

## 6.如何跨应用程序类型共享组件注册？

How do I share component registrations across application types?

有人想在网络应用程序中使用按请求生命周期作用域，但又想在其他类型的应用程序中使用其他作用域，这就是问题所在。

## 7.如何使 Autofac 与应用程序隔离？

How do I keep Autofac references isolated away from my app?

有时，你可能希望尽量减少对 Autofac 库和 IoC 容器的引用。根据你的应用程序结构，可能获得不同程度的成功。这里有一些关于如何组织应用程序以最小化 Autofac 引用的提示，以及如果你选择这样做，需要注意的一些事项。

**没有一个方法能解决所有问题。** 你需要根据应用程序的结构和目标进行混合和匹配。

### 应用程序启动

应用程序启动时，会构建 IoC 容器并进行注册。这也是 IoC 容器与应用程序的 [组合根](https://www.koudingke.cn/go?link=http%3a%2f%2fblog.ploeh.dk%2f2011%2f07%2f28%2fCompositionRoot%2f)集成的地方。对于控制台应用，这是 Main方法；对于 ASP.NET 应用，这是 Startup类或 Global.asax；对于其他应用，有其他入口点。

不应试图将 IoC 容器与应用程序的这一部分分离。这是它具体连接到事物的地方。如果你尝试从包含以下代码的模块（如 Global.asax或 Startup类）中移除 Autofac 包/库引用，**节省一下时间，别这么做**。你可能会花很多时间编写抽象和包装，但只会复制大量特定于 Autofac 的语法和功能。

### 组件注册

Autofac 与应用程序连接的大部分地方是你将组件注册到容器的地方。有几种方法可以限制这里与 Autofac 的连接。

### Assembly 扫描

Assembly Scanning

Autofac 支持通过 Assembly 扫描 注册内容。如果你的应用程序有一个已知的插件集或多个可执行文件，并且想要注册实现特定接口或遵循特定命名规则的所有类型，这很有帮助。

然而，在使用模块扫描时，很难配置单个组件。例如，你不能真正地说：“注册所有这些类型，但是只有这少数几个需要单例，而其余的必须按依赖关系实例化。”

如果你发现想使用模块扫描，但需要这种级别的控制，你可能需要使用 .NET 反射并在自己的代码中处理额外功能。例如，如果你想通过自定义属性指定生命周期范围，你可以创建该自定义属性，并相应地编写一组方法来遍历一组模块，根据属性值进行注册。

### 模块

使用 Autofac 模块 是将相关类型注册分组的好方法，但它将实现 Autofac.Module接口的类型与 Autofac 库绑定在一起。

解决这个问题的一种方法是将 Autofac 模块放入单独的库中。这样，对于特定功能，你将有两个库：

* 产品库：此库包含实际代码，不引用 Autofac。
* 模块库：此库引用产品库以及 Autofac。这就是你放置 Autofac 模块的地方。

然后，你的应用程序可以使用模块扫描来查找所有“模块库”，并在其中注册模块。（一种简化扫描的方法是采用模块库的命名约定，然后只在这些库上运行扫描。）

### 配置

Autofac 配置 系统完全基于配置文件工作，并允许你在应用程序启动代码中指定无需引用其他位置的注册项。它不如 Autofac 模块 功能全面，因此你可能无法全部使用这种方式注册，但它对少量注册很有帮助。

你也可以使用配置系统注册 Autofac 模块——这可以让你在 “模块库”（见上文）中指定模块，跳过模块扫描。

### IStartable

Autofac 提供了 IStartable 接口，你可以使用它来自动 解析一个组件并执行代码 。

如果你的 IStartable 注册为单例（通常应该是），你可以利用 AutoActivate() 方法以及 OnActivated 处理器来替换它并移除对 Autofac 的依赖：

var builder = new ContainerBuilder();  
 builder  
 .RegisterType<TypeRequiringWarmStart>()  
 .AutoActivate()  
 .OnActivating(e => e.Instance.Start());

### 服务解析

Service Resolution

在大多数 DI/IoC 使用情况下，你不应该引用 IoC 容器——相反，你会有构造参数和/或可以通过容器设置的属性。

然而，你可能会发现自己在以下几个领域与 Autofac 相关联...

#### 服务定位

Service Location

一些框架缺乏在应用程序启动级别启用所有依赖注入钩子的组合根挂钩。例如，经典的 ASP.NET HttpModules就是这种情况——通常没有允许你向模块注入依赖项的钩子。在这种情况下，你可能会发现使用服务定位很有用（尽管在可能的情况下应尽量减少服务定位，因为它是反模式 [参见](https://www.koudingke.cn/go?link=http%3a%2f%2fblog.ploeh.dk%2f2010%2f02%2f03%2fServiceLocatorisanAnti-Pattern%2f)）。

在需要服务定位但又不想绑定到 Autofac 的情况下，请考虑使用像 [CommonServiceLocator](https://www.koudingke.cn/go?link=https%3a%2f%2fwww.nuget.org%2fpackages%2fCommonServiceLocator%2f) 或 [Microsoft.Extensions.DependencyInjection](https://www.koudingke.cn/go?link=https%3a%2f%2fwww.nuget.org%2fpackages%2fMicrosoft.Extensions.DependencyInjection%2f) 这样的抽象。在 ASP.NET MVC 应用程序中，你已经提供了用于服务定位的 DependencyResolver；其他应用程序类型可能已经提供了类似抽象。通过使用其中一个抽象，你可以移除对 Autofac 的引用……尽管你仍需要引用抽象。

#### 隐式关系

Implicit Relationships

Autofac 支持多种 隐式关系 ，如 IEnumerable、Lazy 和 Func。大部分关系基于核心 .NET 类型。然而，如果你使用以下内容，你将绑定到 Autofac：

* IIndex<X, B>（索引集合）
* Meta 和 Meta<B, X>（强类型元数据）

对象的实例名称/键或元数据没有替代品或解决方案。如果你需要这种功能，你只能使用这些关系。

但是，你可以通过代码来潜在地减少对这些关系的使用：

* 创建工厂：你可以在工厂中包装对这些关系的使用。在应用程序代码库中定义工厂接口，并在允许引用 Autofac 的单独库中定义实现。
* 使用 lambda 注册：你可以使用 lambda注册组件，并在 lambda 中直接解析值。这有点像将工厂放在 lambda 注册中，而不是为其定义单独的接口。这会在应用程序代码中添加一点，将其放入注册（例如，使用元数据和服务键），但这可能是可接受的妥协。

## 8.为什么框架的“旧版本”（如 System.Core 2.0.5.0）被引用？

Why are "old versions" of the framework (e.g., System.Core 2.0.5.0) referenced?

**Autofac（3.x 版本）是一个可移植类库（Portable Class Library，PCL），它针对多个平台进行了优化。**

作为 PCL，如果你使用 Reflector、dotPeek 等工具打开 Autofac，你会看到对一些系统库（如 System.Core）2.0.5.0 版本的引用。实际上，2.0.5.0 版本是 Silverlight 版本的.NET 框架。**这是预期的，并不是问题。** 在运行时，所有东西都会正常工作。Autofac 将会正确绑定到你正在使用的框架版本——无论是.NET 4.5、Silverlight 还是 Windows Phone。[关于PCL，你可以参考MSDN上的更多内容。](https://www.koudingke.cn/go?link=https%3a%2f%2fmsdn.microsoft.com%2fen-us%2flibrary%2fgg597391.aspx)

在使用 Autofac 作为 PCL 时，你可能会遇到类似以下异常：

Test 'MyNamespace.MyFixture.MyTest' failed: System.IO.FileLoadException : Could not load file or assembly 'System.Core, Version=2.0.5.0, Culture=neutral, PublicKeyToken=7cec85d7bea7798e, Retargetable=Yes' or one of its dependencies. The given assembly name or codebase was invalid. (Exception from HRESULT: 0x80131047)  
 at Autofac.Builder.RegistrationData..ctor(Service defaultService)  
 at Autofac.Builder.RegistrationBuilder`3..ctor(Service defaultService, TActivatorData activatorData, TRegistrationStyle style)  
 at Autofac.RegistrationExtensions.RegisterInstance[T](ContainerBuilder builder, T instance)  
 MyProject\MyFixture.cs(49,0): at MyNamespace.MyFixture.MyTest()

测试 'MyNamespace.MyFixture.MyTest' 失败：System.IO.FileLoadException: 无法加载文件或程序集 'System.Core, Version=2.0.5.0, Culture=neutral, PublicKeyToken=7cec85d7bea7798e, Retargetable=Yes' 或其依赖项。给定的程序集名称或代码基无效。 (异常来自 HRESULT: 0x80131047)  
 在 Autofac.Builder.RegistrationData..ctor(Service defaultService)  
 在 Autofac.Builder.RegistrationBuilder`3..ctor(Service defaultService, TActivatorData activatorData, TRegistrationStyle style)  
 在 Autofac.RegistrationExtensions.RegisterInstance[T](ContainerBuilder builder, T instance)  
 MyProject\MyFixture.cs(49,0): 在 MyNamespace.MyFixture.MyTest()

**请确保你的.NET** **框架已更新。** Microsoft 已经发布了补丁来允许 PCL 正确找到合适的运行时环境（[KB2468871](https://www.koudingke.cn/go?link=https%3a%2f%2fsupport.microsoft.com%2fkb%2f2468871)）。如果你遇到上述异常（或其他类似错误），那意味着你缺少最新的.NET 框架补丁。

[这篇博客文章对这些内容以及使用PCL时可能遇到的其他问题有很好的概述。](https://www.koudingke.cn/go?link=https%3a%2f%2fwww.paraesthesia.com%2farchive%2f2013%2f03%2f29%2fportable-class-library-answers.aspx)

从 Autofac 4.x 开始，它不再支持 PCL，而是转向.NET Standard。你应该不会在 Autofac 4.x 及更高版本中看到这类错误。

## 9.为什么不是所有包都使用最新的 Autofac 核心？

Why don't all Autofac packages target the latest Autofac core?

Autofac 有很多 集成包 和扩展。你会发现，并非所有这些包都直接引用最新版的 Autofac 核心。

**除非有技术原因需要将其中一个包的最低版本要求提高，否则我们会保持版本不变。**

我们这么做是因为一般来说，我们不想强迫任何人更新他们的 Autofac 核心版本，除非他们必须这样做。这对任何库集来说都是一个不错的做法——如果用户不必更新，就不应该强制他们更新。

**最简单的解决方案是在你的应用程序/项目中添加对所需** **Autofac** **核心版本的直接引用。**

**在** **.NET Core** **中，“这就可以工作”。** 项目系统会自动识别项目中（即直接引用）的最新 Autofac 版本，并重定向所有绑定到该版本。无需额外的麻烦，就像魔法一样。

**但在** **.NET** **全框架项目中，这会导致需要使用类型绑定重定向。**[这是官方支持的方式](https://www.koudingke.cn/go?link=https%3a%2f%2fmsdn.microsoft.com%2fen-us%2flibrary%2fvstudio%2f2fc472t2.aspx)，用来告诉 .NET 运行时需要将对强命名 assembly 的某个版本的请求重定向到该 assembly 的后续版本。这种情况很常见，因此 NuGet 和 Visual Studio 在许多情况下都会自动将这些配置添加到你的配置文件中。

以下是一个类型绑定重定向的例子：

<?xml version="1.0" encoding="utf-8"?>  
 <configuration>  
 <runtime>  
 <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">  
 <dependentAssembly>  
 <assemblyIdentity name="Autofac"  
 publicKeyToken="17863af14b0044da"  
 culture="neutral" />  
 <bindingRedirect oldVersion="0.0.0.0-3.5.0.0"  
 newVersion="3.5.0.0" />  
 </dependentAssembly>  
 <dependentAssembly>  
 <assemblyIdentity name="Autofac.Extras.CommonServiceLocator"  
 publicKeyToken="17863af14b0044da"  
 culture="neutral" />  
 <bindingRedirect oldVersion="0.0.0.0-3.1.0.0"  
 newVersion="3.1.0.0" />  
 </dependentAssembly>  
 <dependentAssembly>  
 <assemblyIdentity name="Autofac.Extras.Multitenant"  
 publicKeyToken="17863af14b0044da"  
 culture="neutral" />  
 <bindingRedirect oldVersion="0.0.0.0-3.1.0.0"  
 newVersion="3.1.0.0" />  
 </dependentAssembly>  
 <dependentAssembly>  
 <assemblyIdentity name="Autofac.Integration.Mvc"  
 publicKeyToken="17863af14b0044da"  
 culture="neutral" />  
 <bindingRedirect oldVersion="0.0.0.0-3.3.0.0"  
 newVersion="3.3.0.0" />  
 </dependentAssembly>  
 </assemblyBinding>  
 </runtime>  
 </configuration>

类型绑定重定向是 [assembly 强命名](https://www.koudingke.cn/go?link=https%3a%2f%2fmsdn.microsoft.com%2fen-us%2flibrary%2fwd40t7ad.aspx)的不幸副产品。如果 assembly 不是强命名的，你就不需要绑定重定向；但在某些环境中需要 assembly 强命名，所以 Autofac 继续为 assembly 强命名。

**即使 Autofac 总是保持所有引用是最新的，你仍然无法摆脱类型绑定重定向。** 例如，Web API 集成 这样的 Autofac 集成包依赖于其他强命名包，而这些包又有自己的依赖关系。比如，[Microsoft Web API 包](https://www.koudingke.cn/go?link=https%3a%2f%2fwww.nuget.org%2fpackages%2fMicrosoft.AspNet.WebApi.Client%2f)依赖于 [Newtonsoft.Json](https://www.koudingke.cn/go?link=https%3a%2f%2fwww.nuget.org%2fpackages%2fNewtonsoft.Json%2f)，但它们并不总是保持与最新版本同步。相反，它们会指定一个兼容的最低版本。**如果你更新了本地的** **Newtonsoft.Json** **版本……你就会得到一个绑定重定向。**

**与其试图对抗绑定重定向，不如接受它作为** **.NET** **世界中的“商业成本”可能更好。** 它确实会在应用程序配置文件中增加一些“混乱”，但在我们能够从方程中移除强命名之前，这是不可避免的需求。

## 10.如何注入配置、环境或上下文参数？

How do I inject configuration, environment, or context parameters?

有时候，你需要解析一个依赖链中某个位置的 服务 ，而该服务消费了一个 组件 ，并且该组件需要从配置、环境或其他运行时上下文位置传递给它的参数。

对于这个问题，让我们想象一个简单的电子邮件通知系统：

public interface INotifier  
 {  
 void Send(string address, string message);  
 }  
 public class Notifier : INotifier  
 {  
 private IEmailServer \_server;  
 public Notifier(IEmailServer server)  
 {  
 this.\_server = server;  
 }  
 public void Send(string address, string message)  
 {  
 this.\_server.SendMessage(address, "from@domain.com", message);  
 }  
 }  
 public interface IEmailServer  
 {  
 void SendMessage(string toAddress, string fromAddress, message);  
 }  
 public class EmailServer : IEmailServer  
 {  
 private string \_serverAddress;  
 public EmailServer(string serverAddress)  
 {  
 this.\_serverAddress = serverAddress;  
 }  
 public void SendMessage(string toAddress, string fromAddress, message)  
 {  
   
 }  
 }

在 Autofac 中注册内容时，你可能会有如下这样的注册：

var builder = new ContainerBuilder();  
 builder.RegisterType<Notifier>().As<INotifier>();  
 builder.RegisterType<EmailServer>().As<IEmailServer>();  
 var container = builder.Build();

你只知道电子邮件服务器地址是在运行时 - 可能是通过上下文或环境参数，也可能是通过配置。

**如何在解析通知器时将配置/环境/上下文参数传递给电子邮件服务器？**

### 选项 1：使用 Lambda 注册

Option 1: Register Using a Lambda

在这个选项中，而不是直接注册电子邮件服务器类型，使用 Lambda 表达式 注册。这允许你从容器中解析内容，或者使用环境获取值。

var builder = new ContainerBuilder();  
 builder.Register(ctx =>  
 {  
 var address = Environment.GetEnvironmentVariable("SERVER\_ADDRESS");  
 return new EmailServer(address);  
 }).As<IEmailServer>();

作为这一过程的一部分，你可能希望围绕如何获取服务器地址创建某种抽象。例如，它可能是作为 HTTP 请求的一部分获取的，并且你将其存储在 HttpContext中。你可以创建一个地址提供者，如下所示：

public interface IServerAddressProvider  
 {  
 string GetServerAddress();  
 }  
 public class ContextServerAddressProvider : IServerAddressProvider  
 {  
 private HttpContextBase \_context;  
 public ContextServerAddressProvider(HttpContextBase context)  
 {  
 this.\_context = context;  
 }  
 public string GetServerAddress()  
 {  
 return (string)this.\_context.Items["EMAIL\_SERVER\_ADDRESS"];  
 }  
 }

一旦有了提供者，你可以在容器中注册它，并在与 Lambda 结合使用时使用它。

var builder = new ContainerBuilder();  
 builder.RegisterType<ContextServerAddressProvider>()  
 .As<IServerAddressProvider>()  
 .InstancePerRequest();  
 builder.Register(ctx =>  
 {  
 var address = ctx.Resolve<IServerAddressProvider>().GetServerAddress();  
 return new EmailServer(address);  
 }).As<IEmailServer>();

**如果需要传递一个字符串参数或无法修改代码，这是推荐的选项。**

### 选项 2：使用Provider

Option 2: Use a Provider

扩展第 1 选项中描述的提供者机制：通常最大的问题是，需要传递的参数是一个基本类型，如整数或字符串。如果你可以将其切换为使用提供者和强类型接口参数，那么注册会变得更加容易。

例如，你可能可以从像这样 的 Web 请求上下文中获取参数。

public interface IServerAddressProvider  
 {  
 string GetServerAddress();  
 }  
 public class ContextServerAddressProvider : IServerAddressProvider  
 {  
 private HttpContextBase \_context;  
 public ContextServerAddressProvider(HttpContextBase context)  
 {  
 this.\_context = context;  
 }  
 public string GetServerAddress()  
 {  
 return (string)this.\_context.Items["EMAIL\_SERVER\_ADDRESS"];  
 }  
 }

然后，你可以重构电子邮件服务器代码，使其接受提供者而不是地址字符串：

public class EmailServer : IEmailServer  
 {  
 private IServerAddressProvider \_serverAddressProvider;  
 public EmailServer(IServerAddressProvider serverAddressProvider)  
 {  
 this.\_serverAddressProvider = serverAddressProvider;  
 }  
 public void SendMessage(string toAddress, string fromAddress, message)  
 {  
 var address = this.\_serverAddressProvider.GetServerAddress();  
   
 }  
 }

现在，你可以简单地注册类型：

var builder = new ContainerBuilder();  
 builder.RegisterType<ContextServerAddressProvider>()  
 .As<IServerAddressProvider>()  
 .InstancePerRequest();  
 builder.RegisterType<EmailServer>().As<IEmailServer>();

**如果可以**修改代码**，这是推荐的选项。**

## 11.如何在解析链的中间传递组件参数？

How do I pass a parameter to a component in the middle of a resolve chain?

有时，你可能需要在一个依赖链中某个位置解析一个 服务 ，而这个服务又需要一个 组件 。这个组件需要被传递一个参数 。

假设我们有一个简单的电子邮件通知系统：

public interface INotifier  
 {  
 void Send(string address, string message);  
 }  
 public class Notifier : INotifier  
 {  
 private IEmailServer \_server;  
 public Notifier(IEmailServer server)  
 {  
 this.\_server = server;  
 }  
 public void Send(string address, string message)  
 {  
 this.\_server.SendMessage(address, "from@domain.com", message);  
 }  
 }  
 public interface IEmailServer  
 {  
 void SendMessage(string toAddress, string fromAddress, message);  
 }  
 public class EmailServer : IEmailServer  
 {  
 private string \_serverAddress;  
 public EmailServer(string serverAddress)  
 {  
 this.\_serverAddress = serverAddress;  
 }  
 public void SendMessage(string toAddress, string fromAddress, message)  
 {  
   
 }  
 }

在 Autofac 注册时，你可能会有如下注册：

var builder = new ContainerBuilder();  
 builder.RegisterType<Notifier>().As<INotifier>();  
 builder.RegisterType<EmailServer>().As<IEmailServer>();  
 var container = builder.Build();

服务器地址通常在运行时才能知道，可能是来自上下文或环境参数，也可能是配置。

**如何在解析通知器时将服务器地址作为参数传递给电子邮件服务器？**

### 这是一个设计问题的原因

在回答这个问题之前，考虑一下这在很大程度上表明了一个 **设计问题**。

从技术角度讲，你在解析一个 INotifier - 一个不需要知道运行时参数（即电子邮件服务器地址）的组件。INotifier的实现可以改变。你可以为测试注册一个占位符，或者更改发送方式，使其不再需要知道地址。

将服务器地址作为参数传递给 INotifier破坏了基于接口的开发和控制反转提供的解耦，因为它假设你知道整个依赖链是如何被解析的。

**解决问题的关键是打破这种“知识”，而不是传递参数。**

### 解决方案

**不要试图传递参数，而是** **反转问题** **-** **理解如何在运行时确定参数，并将其包装在提供者或** **lambda** **表达式注册中。**

这样，问题就变成了另一个 FAQ，我们将逐步给出答案：如何注入配置、环境或上下文参数？

## 12.为什么没有内置的容器注册分析？

Why isn't container registration analysis built in?

在使用 Autofac 容器时，如果注册不完整或有误，解析时出现运行时异常会让人感到非常沮丧。为什么不内置一种分析机制，以便在容器构建后验证其有效性呢？

虽然某些容器确实提供了这种功能，但 Autofac 在依赖项解析方面的灵活性（根据运行时参数和动态功能进行处理）使得提供有用的容器有效性检查变得困难。

考虑以下示例代码：

var builder = new ContainerBuilder();  
 builder.RegisterType<ProdConfiguration>().AsSelf();  
 builder.Register(ctx => {  
 var env = Environment.GetEnvironmentVariable("ENVIRONMENT");  
 switch (env)  
 {  
 case "Development":  
 return new TestConfiguration();  
 case "Production":  
 return ctx.Resolve<ProdConfiguration>(new NamedParameter("connStr", connectionString));  
 default:  
 throw new NotSupportedException("未知的环境名称。");  
 }).As<IConfiguration>();  
 });

这样的容器配置是完全有效的，但也引发了一些问题：

* 如果 ProdConfiguration需要一个未在容器中注册的连接字符串参数，你的容器有效吗？如果你使用服务定位并在解析时传递该字符串（参阅 这里），容器如何知道？
* 如果 IConfiguration 依赖于特定的环境参数，而在部署时存在但在单元测试时不存在，你的容器有效吗？

这些都是相对简单的情况。再考虑一下其他情况，如...

这些并非罕见场景，而且这还不是支持的所有动态功能的完整列表。

尽管有可能在将来添加一个非常简单的分析机制来检测少数问题，但更有可能的是增强 Autofac 的诊断和跟踪功能，以更快地定位和解决遇到的运行时挑战。（如果你有兴趣参与这项工作，请 [告诉我们！](https://www.koudingke.cn/go?link=https%3a%2f%2fgithub.com%2fautofac%2fAutofac%2fissues)）

## 13.为什么我的 Xamarin 应用出现异常？

Why are things in my Xamarin app misbehaving?

**Autofac 针对.NET Framework 和.NET Standard。** 这使得代码在多个平台之间具有相当的可移植性。它可用于 Mono、Xamarin 和通用 Windows 平台应用等。

[Xamarin提供了一个跨平台编译器](https://www.koudingke.cn/go?link=https%3a%2f%2fdocs.microsoft.com%2fen-us%2fxamarin%2fcross-platform%2fget-started%2fintroduction-to-mobile-development)，它能够将 C# 和 .NET 代码编译成原生应用。从文档中可以了解到：

*在* *iOS* 上，Xamarin 的预编译器（AOT）会直接将 Xamarin.iOS 应用程序编译为 ARM 原生汇编代码。在 *Android* 上，Xamarin 的编译器会编译为中间语言（**IL**），然后在应用启动时即时编译为原生汇编。

其中的一个挑战是，并非所有最终平台（如 iOS 或 Android）都支持所有的 .NET 特性。例如，链接器可能会移除它认为未使用的类型，但实际上这些类型在反射和依赖注入中是必需的。这样做是为了加快应用速度并减小总体大小——移除不认为会被使用到的类型和方法。这种转换和优化可能会导致原本在 .NET 中运行正常的应用，在 Xamarin 编译后的原生应用中表现出略微不同的行为。

**Autofac 并未特别针对 Xamarin 或.NET Native 应用进行构建或测试。** 在针对 .NET Standard 的情况下，确保与其他 .NET Standard 代码兼容的任务就交给了跨平台编译器和链接器。

我们从社区收集了一些关于让 Xamarin 和 .NET Native 应用正常工作的提示。详细了解。

希望这些能帮到你。如果你有其他可能帮助他人的提示或文章，请告诉我们，我们会很高兴将其加入。有时，在 *Xamarin* 中使用反射正确设置编译器和链接器指令可能会是一项挑战。

如果在查看了提示并做了研究后仍然遇到问题，请在 StackOverflow 上提问。 使用标签 xamarin，这样熟悉 Xamarin 的人就能收到通知并帮助回答问题：[相关问题](https://stackoverflow.com/questions/tagged/xamarin)。