### .Net Core 依赖注入实现原理-基本特性

相对.Net Framework来说，.Net Core的优势就是跨平台和模块化。其实软件多多少少都会采用模块化的方式来设计，只是.Net Core更加激进一点。.Net Core的所有组件都是细粒度的，采用Nuget包的形式发布。MS几乎完全重写了原先的.Net，并且鼓励开发人员采用以前不常用的开发理念，比这里要说的依赖注入（ID），它在Asp.Net Core里被作为最基础组件来使用。

MS开发的这个轻量级的依赖注入组件，包名为：

Microsoft.Extensions.DependencyInjection

Guthub地址：<https://github.com/aspnet/DependencyInjection>

虽然托管在ASP.NET Core 相关的项目集下面，但是它其实是独立的。说到DI，肯定会想起另一个名字：IoC（控制反转）。对于IoC的理解以及它和ID的关系，可以参考蒋金楠同样关于这个依赖注入组件讲解的系列博客：

IoC：<http://www.cnblogs.com/artech/p/asp-net-core-di-ioc.html>

DI：<http://www.cnblogs.com/artech/p/asp-net-core-di-di.html>

我之前一直以为IoC和ID就是同一个东西的两个不同叫法而已，它们确实很像，现在发现可能错了。蒋大大把“控制反转”理解成：对于“流程的控制”的转移。它是一种设计原则。而DI是IoC的一种实现模式。具体的可以查看上面两篇博客。

读一个项目的源码其实需要方法，一上来就看代码难免找不着北。我自己一般都是先掌握它的基本使用方法，了解它的特性，然后以调试的方式多走几遍流程，等到对里面实现方式有了大概的理解之后，如果这时候想更清楚一点，读代码相对来说就会轻松很多。所以一般讲原理的东西先从基本的使用开始讲（蒋大大一直是这么做的）。这篇文章里的内容和蒋大大博客里的内容大致相同，除此之外加入了我自己的一些总结，对其中有些内容，比如生命周期管理讲得更详细一些。没有站在大牛的肩膀上，很多东西还要自己去琢磨和理解。

进入正题。

1. 接口

这个默认的依赖注入实现主要由下面两个接口暴露功能：

IServiceProvidor：客户端最终调用它来请求对应的服务实现（这个接口在System命名空间下，不属于ID这个组件）；

IServiceCollection：客户端通过它注册服务；

说明：这里的“服务”，具体是指：封装了相关功能的类或对象。

1. 使用方法

构造一个IServiceCollection对象，通过它的一系列扩展方法注册服务，调用IServiceCollection的扩展方法BuildServiceProvider获得IServiceProvidor对象，最后调用IServiceProvidor的GetService方法获得具体的服务对象。

1. 基本特性

之后的代码例子需要的接口（服务）和类（服务实现）：

服务：

public interface IFoo { }

public interface IBar { }

public interface IBiz { }

public interface IGux { }

服务实现：

public class Foo : Base, IFoo { }

public class Foo2 : Base, IFoo { }

public class Bar : Base, IBar { }

public class Biz : Base, IBiz { }

public class Gux : Base, IGux { }

服务基类（封装公共代码）：

public class Base : IDisposable

{

public Base()

{

Console.WriteLine($"Created: {this.GetType()}");

}

public void Dispose()

{

Console.WriteLine($"Disposed: {this.GetType()}");

}

}

1. 服务的注册及获取

此实现给IServiceCollection这个接口扩展了许多扩展方法，可分为三组，分别对应服务的三种生命周期（生命周期管理后面会说到，这里先大概理解）：

AddTransient：注册瞬态服务，即后续每次获取都创建一个新的服务对象；

AddScoped：注册具有作用域的服务，即在一定的作用于范围内获取到服务是同一个对象；

AddSingleton：注册单例的服务，即每次获取的都是同一个服务对象；

如下代码注册机获取服务：

IServiceProvider provider = new ServiceCollection()

.AddTransient<IFoo, Foo>()

.AddScoped<IBar, Bar>()

.AddSingleton<IBiz, Biz>()

.BuildServiceProvider();

IFoo foo = provider.GetService<IFoo>();

IBar bar = provider.GetService<IBar>();

IBiz biz = provider.GetService<IBiz>();

IBiz biz2 = provider.GetService<IBiz>();

Console.WriteLine(foo);

Console.WriteLine(bar);

Console.WriteLine(biz);

Console.WriteLine($"biz == biz2:{ ReferenceEquals(biz, biz2)}");

输出为：

Created: DependencyInjectionDemo.Foo

Created: DependencyInjectionDemo.Bar

Created: DependencyInjectionDemo.Biz

DependencyInjectionDemo.Foo

DependencyInjectionDemo.Bar

DependencyInjectionDemo.Biz

biz == biz2:True

针对每种生命周期，IServiceCollection提供了不同的重载，主要区别是创建服务对象的方式不同：

提供服务实现类的Type对象：运行时通过反射创建服务对象；

提供服务实现类的对象：运行时直接返回这个对象，只能用于单例的情况；

提供一个创建服务对象的委托：运行时调该委托创建服务对象；

具体代码如下：

IServiceProvider provider = new ServiceCollection()

.AddSingleton<IFoo, Foo>() // 提供类型

.AddSingleton<IFoo>(\_ => new Foo()) // 提供委托

.AddSingleton<IFoo>(new Foo()) // 提供服务对象

.BuildServiceProvider();

1. 对泛型的支持

这里的泛型其实是指开放类型（未指定泛型参数的类型模板），比如：

IEnumerable<>,IDictionary<,>。具体使用如下面的代码：、

IServiceProvider provider = new ServiceCollection()

.AddTransient(typeof(IGeneric<,>), typeof(Generic<,>))

.BuildServiceProvider();

Console.WriteLine(provider.GetService<IGeneric<IFoo, IBar>>());

上面获取到的服务类型是Generic<IFoo, IBar>，可见获取的服务实现类型其实就是：拿IGeneric<IFoo, IBar>的两个泛型参数，以Generic<,>为模板创建的类型。

注意：

1. 这里的AddTransient之所以没有用泛型版本，是因为泛型参数必须是封闭类型，所以这里把类型信息作为参数传递进去；
2. 注册的时候，服务类型（IGeneric<,>）和服务实现类型（Generic<,>）必须都是开放类型，而且泛型参数数目要相同；
3. 获取服务的时候指定一个封闭类型（IGeneric<IFoo, IBar>），并且它对应的开放类型（IGeneric<,>）必须和注册时的服务类型（IGeneric<,>）相同；
4. 注册的服务类型（IGeneric<,>）和服务实现类型（Generic<,>）不一定要有继承关系，比如上面的IGeneric<,>和Generic<,>并没有继承关系，但是实际得到的服务实现类型（Generic<IFoo, IBar>）必须和获取时指定的服务类型（IGeneric<IFoo, IBar>）有继承关系；
5. 多次注册相同的服务

“相同的服务”是指：注册时的服务类型相同。具体包括下面几种情况：

生命周期相同，服务实现类型不同；

生命周期不同，服务实现类型相同；

生命周期不同，服务实现类型也不同；

无论是上面那种情况，获取服务时，以最后一次注册的为准。如下面的测试代码：

IServiceProvider provider = new ServiceCollection()

.AddTransient<IFoo, Foo>()

.AddTransient<IFoo, Foo2>()

.AddSingleton<IFoo, Foo>()

.BuildServiceProvider();

IFoo foo1 = provider.GetService<IFoo>();

IFoo foo2 = provider.GetService<IFoo>();

Console.WriteLine($"foo1: {foo1.GetType()}");

Console.WriteLine($"foo1 == foo2: {ReferenceEquals(foo1, foo2)}");

上面代码输出：

Created: DependencyInjectionDemo.Foo

foo1 == foo2: True

为了获取到注册的所有服务实现，可以调用IServiceProvider的GetServices扩展方法，比如上面的例子要获取IFoo对应的所有的服务实现可以这样：

foreach (var foo in provider.GetServices<IFoo>())

{

}

上面的代码输出：

Created: DependencyInjectionDemo.Foo

Created: DependencyInjectionDemo.Foo2

Created: DependencyInjectionDemo.Foo

4.注入方式

服务之间往往存在相互依赖，比如服务IA有一个服务类型为IB的属性B，那么获取服务IA时，服务容器应该要负责初始化IA的属性B。这个初始化过程就是（属性）注入，注入的方式一般有构造器注入，属性注入和方法注入。此实现只支持构造器注入。比如我修改类型Gux给他添加一个构造器：

public class Gux : Disposable, IGux

{

public Gux(IFoo foo)

{

Console.WriteLine("Gux(IFoo)");

}

}

然后注册并获取服务IGux：

new ServiceCollection()

.AddTransient<IFoo, Foo>()

.AddTransient<IGux, Gux>()

.BuildServiceProvider()

.GetService<IGux>();

上面的代码输出：

Created: DependencyInjectionDemo.Foo

Created: DependencyInjectionDemo.Gux

Gux(IFoo)

可见服务容器自动调用了Gux的构造器，传入了已事先创建好的IFoo服务对象。

5.构造器选择策略

上面说到，服务容器默认通过构造器的方式来创建服务实例，它所选择的构造器必须满足下面两个条件：

1.服务容器必须能够提供所有的参数；

2.在满足条件1的所有构造器中，选出一个参数最多的，所有其他构造函数的参数类型列表都是该构造函数参数类型列表的子集；

如果不满足上面2个条件，就抛InvalidOperationException异常。

为了证实这两点，先修改Gux类：

public class Gux : Disposable, IGux

{

public Gux(IFoo foo)

{

Console.WriteLine("Gux(IFoo)");

}

public Gux(IFoo foo, IBar bar)

{

Console.WriteLine("Gux(IFoo, IBar)");

}

public Gux(IFoo foo, IBar bar, IBiz biz)

{

Console.WriteLine("Gux(IFoo, IBar, IBaz)");

}

}

然后注册并获取IGux服务：

new ServiceCollection()

.AddTransient<IFoo, Foo>()

.AddTransient<IBar, Bar>()

.AddTransient<IBiz, Biz>()

.AddTransient<IGux, Gux>()

.BuildServiceProvider()

.GetService<IGux>();

输出：

Created: DependencyInjectionDemo.Foo

Created: DependencyInjectionDemo.Bar

Created: DependencyInjectionDemo.Biz

Created: DependencyInjectionDemo.Gux

Gux(IFoo, IBar, IBiz)

现在把构造函数public Gux(IFoo foo, IBar bar, IBiz biz) 改成：

public Gux(IFoo foo, IBiz biz)，同样运行上面的代码后，输出：

Unable to activate type 'DependencyInjectionDemo.Gux'. The following constructors are ambigious:

Void .ctor(DependencyInjectionDemo.IFoo, DependencyInjectionDemo.IBar)

Void .ctor(DependencyInjectionDemo.IFoo, DependencyInjectionDemo.IBiz)

6.服务的生命周期管理

前面说到注册服务时，可以以三种方式进行注册，具体的区别体现在后续服务实例的创建方式，以及它们的生命周期上面：

AddTransient方式注册的服务，由当前作用域的服务容器创建，每次获取都创建一个新的服务实例，它的生命周期由当前的作用域的服务容器维护；

AddScoped方式注册的服务，由当前作用域的服务容器创建，在相同作用域内获取到的是同一个服务实例，它的生命周期由当前作作用域的服务容器维护；

AddSingleton方式注册的服务，由根作用域的服务容器创建，每次获取到的都是同一个服务实例，它的生命周期由根作用域的服务容器维护；

首先解释作用域和服务容器的关系，到现在为止，我们的代码创建的IServiceProvider其实是在根作用域下的服务容器，但是它所在的根作用于域并没有一个实际的对象来表示，这个根作用域在整个应用程序期间一直存在。那么如何创建一个作用域呢？可以通过向服务容器IServiceProvider请求

IServiceScopeFactory类型的服务，继而用它创建一个根作用域下的一个子作用域：

IServiceProvider root = new ServiceCollection()

.BuildServiceProvider();

IServiceScope childScope = root.GetService<IServiceScopeFactory>().CreateScope();

其实IServiceScopeFactory是根服务容器内部注册的一个服务。通过IServiceScope我们可以得到和这个作用域对应的服务容器：

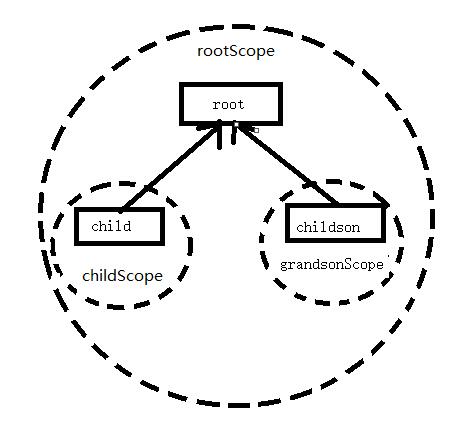
IServiceProvider child = childScope.ServiceProvider;

这样一来，两个服务容器root和child就具有了父子关系。同样，我们可以通过child来创建一个作用域和对应的服务容器：

IServiceScope grandsonScope = child.GetService<IServiceScopeFactory>().CreateScope();

IServiceProvider grandson = grandsonScope.ServiceProvider;

但是这里要注意的是，虽然从代码逻辑上来看，三个作用域和服务容器具有父，子，孙的关系，但是从物理上来说childScope和grandsonScope，以及child和grandson并没有父子关系，它们只和root及根作用域有父子关系，也就是说这里的层级关系只有2层，而不是3层。childScope和grandsonScope以及child和grandson是同级的。如下图所示：



下面演示作用域是如何管理服务生命周期的，我先用三种方式在根服务容器中注册服务：

IServiceProvider root = new ServiceCollection()

.AddTransient<IFoo, Foo>()

.AddScoped<IBar, Bar>()

.AddSingleton<IBiz, Biz>()

.BuildServiceProvider();

然后创建两个子作用域及其对应的服务容器：

IServiceScope childScope = root.GetService<IServiceScopeFactory>().CreateScope();

ServiceProvider child = childScope.ServiceProvider;

IServiceScope grandsonScope = child.GetService<IServiceScopeFactory>().CreateScope();

IServiceProvider grandson = grandsonScope.ServiceProvider;

首先测试以单例方式注册的服务：

IBiz biz1 = root.GetService<IBiz>();

IBiz biz2 = child.GetService<IBiz>();

IBiz biz3 = grandson.GetService<IBiz>();

Console.WriteLine($"biz1 == biz2: {biz1 == biz2}");

Console.WriteLine($"biz2 == biz3: {biz2 == biz3}");

上面代码输出：

Created: DependencyInjectionDemo.Biz

biz1 == biz2: True

biz2 == biz3: True

可见服务IBiz只在根容器请求的时候创建一次，后续无论在哪里请求都只返回同一个服务实例。

测试以Scope方式创建的服务：

IBar bar1 = root.GetService<IBar>();

IBar bar2 = child.GetService<IBar>();

IBar bar3 = grandson.GetService<IBar>();

IBar bar4 = grandson.GetService<IBar>();

Console.WriteLine($"bar1 == bar2: {bar1 == bar2}");

Console.WriteLine($"bar2 == bar3: {bar2 == bar3}");

Console.WriteLine($"bar1 == bar3: {bar1 == bar3}");

Console.WriteLine($"bar3 == bar4: {bar3 == bar4}");

上面代码输出：

Created: DependencyInjectionDemo.Bar

Created: DependencyInjectionDemo.Bar

Created: DependencyInjectionDemo.Bar

bar1 == bar2: False

bar2 == bar3: False

bar1 == bar3: False

bar3 == bar4: True

可见对于Scope方式创建的服务，在不同的作用域内是独立创建的，在同一个作用域内只创建一次，之后都返回同一个服务实例。

测试以Transient方式创建的服务：

IFoo foo1 = root.GetService<IFoo>();

IFoo foo2 = child.GetService<IFoo>();

IFoo foo3 = grandson.GetService<IFoo>();

IFoo foo4 = grandson.GetService<IFoo>();

Console.WriteLine($"foo1 == foo2: {foo1 == foo2}");

Console.WriteLine($"foo2 == foo3: {foo2 == foo3}");

Console.WriteLine($"foo1 == foo3: {foo1 == foo3}");

Console.WriteLine($"foo3 == foo4: {foo3 == foo4}");

上面的代码输出：

Created: DependencyInjectionDemo.Foo

Created: DependencyInjectionDemo.Foo

Created: DependencyInjectionDemo.Foo

Created: DependencyInjectionDemo.Foo

foo1 == foo2: False

foo2 == foo3: False

foo1 == foo3: False

foo3 == foo4: False

可见每次在请求服务的时候都会创建新的服务实例。

关于生命周期的管理，上面测试只说明了问题的一方面：创建。还有另一方面是：销毁。我们向服务容器获取到服务对象后，容器都有可能保持对服务的引用，除非我们及时地销毁服务容器（调用Dispose方法），进而销毁之前创建的服务对象，否则的话就要等到下一次垃圾回收，由垃圾回收器回收服务容器，这样对内存并不友好。

下面的情况服务容器会保持对服务对象的引用：

单例模式的服务在整个应用程序期间都由根服务容器保持对它的引用；

Scope模式的服务由当前当前作用域的服务容器保持对它的引用；

Transient模式的服务，如果服务实例实现了IDisposable，那么当前作用域的服务容器就会保持对它的引用，否则不会；

作用域的好处就是可以对服务的使用进行隔离和有效管理服务的生命周期，对于Scope模式的服务，我们可以在特定的作用域内创建并使用同一个，并不会和其他的代码产生冲突。对于Transient模式的服务，如果没有作用域，它就有可能被根作用域引用，结果就是它可能始终得不到回收，因为根服务容器的生命周期往往和应用程序的生命周期是相同的。所以下面总结了服务及容器的使用方式：

1. 如果服务是单例的，那么你应该明确它是线程安全的，并且它的生命周期和应用程序的生命周期是一致的；
2. 如果服务是以Scope方式注册的，那么应该在子作用域内进行创建和访问，否则它会以单例的形式一直存在根服务容器中；
3. 如果服务是以Transient方式注册的，并且服务实例实现了IDisposable，那么也应该在子作用域内创建和访问，否则创建的服务将始终被根服务容器引用，从而引起内存泄漏；

下面是创建子作用域进行服务访问的推荐代码：

using (IServiceScope childScope = root.GetService<IServiceScopeFactory>().CreateScope())

{

IServiceProvider child = childScope.ServiceProvider;

//...

}

如下代码：

IServiceProvider root = new ServiceCollection()

.AddTransient<IFoo, Foo>()

.AddScoped<IBar, Bar>()

.AddSingleton<IBiz, Biz>()

.BuildServiceProvider();

using (var scope = root.GetService<IServiceScopeFactory>().CreateScope())

{

var child = scope.ServiceProvider;

IFoo foo = child.GetService<IFoo>();

IBar bar = child.GetService<IBar>();

IBiz biz = child.GetService<IBiz>();

}

Created: DependencyInjectionDemo.Foo

Created: DependencyInjectionDemo.Bar

Created: DependencyInjectionDemo.Biz

Disposed: DependencyInjectionDemo.Foo

Disposed: DependencyInjectionDemo.Bar

看到IFoo和IBar都随着当前作用域scope的销毁而销毁，而IBiz因为是单例的，它的生命周期由根服务容器控制，所以并不受当前作用域的影响。