### .Net Core 依赖注入实现原理（二）-服务注册

[上一篇基本特性](http://justing.me/article/20160823113540)总结了.Net Core自带的依赖注入组件的基本特性，它实现了服务的注册，解析以及服务生命周期的管理，从这一篇开始将通过源码分析这些特性是怎么实现的，本篇是关于服务注册的。

要注册一个服务，很简单的其实就是调用如下的代码：

IServiceProvider provider = new ServiceCollection()

.AddTransient<IFoo, Foo>()

.BuildServiceProvider();

这样就注册了IFoo服务，指定了它的实现类是Foo，其中涉及到三个核心接口：IServiceProvider，IServiceCollection，ServiceDescriptor。优美的设计首先应该是简单的，这三个接口的职责非常简单，先看IServiceProvider：

public interface IServiceProvider

{

object GetService(Type serviceType);

}

它的职责就是获取一个服务类型对象，返回对应的服务实例。接下来是IServiceCollection：

public interface IServiceCollection : IList<ServiceDescriptor>

{

}

这个接口没有自己的成员，只是简单的继承了IList<ServiceDescriptor>接口，说明这个接口的职责是一个ServiceDescriptor的集合。最后看一下ServiceDescriptor：

public class ServiceDescriptor

{

public ServiceLifetime Lifetime { get; }

public Type ServiceType { get; }

public Type ImplementationType { get; }

public object ImplementationInstance { get; }

public Func<IServiceProvider, object> ImplementationFactory { get; }

}

顾名思义，这个类是用于描述服务的，它存储了和服务相关的信息，包括：

* Lifetime：服务的生命周期类型，是个枚举类，有Transient，Scoped和Singleton；
* ServiceType：服务的类型；
* ImplementationType：服务的实现类型，创建服务实例时类型的一个对象；
* ImplementationInstance：服务实例，创建服务实例时直接返回该对象；
* ImplementationFactory：服务实例的创建工厂，创建服务实例时返回由它创建对象；

服务的注册本质上就是创建一个ServiceDescriptor对象，指定服务的信息，然后将它添加到IServiceCollection里，默认的实现提供了一个默认的IServiceCollection实现类：ServiceCollection，它在命名空间Microsoft.Extensions.DependencyInjection下。所以最直接的注册服务的方式像下面这样（和开头的服务注册效果是一样的）：

IServiceCollection sc = new ServiceCollection();

sc.Add(new ServiceDescriptor(typeof(IFoo), typeof(Foo), ServiceLifetime.Transient));

sc.BuildServiceProvider();

但是为了提供清晰易读的方式，实现给IServiceCollection添加了一系列Add..形式.的扩展方法，也就有了开头的注册服务的方式。

在添加完了一系列的服务信息之后，我们需要得到服务容器，也就是IServiceProvider的一个对象，这一步需要调用IServiceCollection的BuildServiceProvider方法。这个方法同样是个扩展方法，在ServiceCollectionContainerBuilderExtensions类里实现，方法的逻辑其实只是简单的创建了一个ServiceProvider类的一个对象，它是IServiceProvider的默认实现。先来看一下ServiceProvider类（只列出了比较重要的几个字段）：

internal class ServiceProvider : IServiceProvider, IDisposable

{

private readonly ServiceTable \_table;

private List<IDisposable> \_transientDisposables;

internal Dictionary<object, object> ResolvedServices { get; } = new Dictionary<object, object>();

internal ServiceProvider Root { get; }

}

些字段的说明如下：

* ServiceTable类型的\_table字段：承载了服务信息，这是另一种形式的服务信息，它们根据ServiceDescriptor创建；除此之外，这个类以委托的形式缓存了服务的创建方式，用于后续再次解析服务时提高效率（服务的的创建后续会讲）；
* List<IDisposable>类型的\_transientDisposables字段：存储实现IDispose接口并且以Transient形式注册的服务实例，用于生命周期管理；
* Dictionary<object, object>类型的ResolvedServices属性：存储以Singleton或者Scoped形式注册的服务实例，用于生命周期管理；
* ServiceProvider类型的Root属性：对于在子作用域里的服务容器，Root指向了根作用域的服务容器，对于根作用域，Root指向本身，用于生命周期管理；

和本篇相关的字段只有上面ServiceTable类型的\_table字段和Root属性，来看一下ServiceProvider的构造函数里的逻辑（删除了非重点代码）：

public ServiceProvider(IEnumerable<ServiceDescriptor> serviceDescriptors)

{

Root = this;

\_table = new ServiceTable(serviceDescriptors);

\_table.Add(typeof(IServiceProvider), new ServiceProviderService());

\_table.Add(typeof(IServiceScopeFactory), new ServiceScopeService());

\_table.Add(typeof(IEnumerable<>), new OpenIEnumerableService(\_table));

}

在创建ServiceProvider的时候，Root属性就自动指向了自身。接下来是初始化\_table字段，ServiceTable的构造函数接受一个ServiceDescriptor类型的集合，然后根据ServiceDescriptor创建另一种类型的服务信息，看一下ServiceTable的构造函数（删除了一些验证性的代码）：

public ServiceTable(IEnumerable<ServiceDescriptor> descriptors)

{

\_services = new Dictionary<Type, ServiceEntry>();

\_genericServices = new Dictionary<Type, List<IGenericService>>();

foreach (var descriptor in descriptors)

{

var serviceTypeInfo = descriptor.ServiceType.GetTypeInfo();

if (serviceTypeInfo.IsGenericTypeDefinition)

{

Add(descriptor.ServiceType, new GenericService(descriptor));

}

else if (descriptor.ImplementationInstance != null)

{

Add(descriptor.ServiceType, new InstanceService(descriptor));

}

else if (descriptor.ImplementationFactory != null)

{

Add(descriptor.ServiceType, new FactoryService(descriptor));

}

else

{

Add(descriptor.ServiceType, new Service(descriptor));

}

}

}

构造函数的逻辑就是初始化\_services和\_genericServices字段，两者分别是存储了IService和IGenericService对象的集合。先看一下IService接口：

internal interface IService

{

IService Next { get; set; }

ServiceLifetime Lifetime { get; }

IServiceCallSite CreateCallSite(ServiceProvider provider, ISet<Type> callSiteChain);

Type ServiceType { get; }

}

* Next属性：当多次注册相同类型的服务的时候，该字段指向了在它后面注册的服务；
* Lifetime：服务的生命周期，来自于ServiceDescriptor；
* ServiceType：服务类型，来自于ServiceDescriptor；
* CreateCallSite：方法创建一个IServiceCallSite类型的对象，用于后续的服务解析；

IGenericService接口：

internal interface IGenericService

{

ServiceLifetime Lifetime { get; }

IService GetService(Type closedServiceType);

}

* Lifetime：服务生命周期，来自于ServiceDescriptor；
* GetService：方法返回泛型服务对应的真是服务，可见IGenericService只是IService的过渡阶段，最终创建服务还会指派到IService；

ServiceTable构造函数的后续逻辑很简单，遍历所有的ServiceDescriptor：对于泛型服务，创建GenericService类型的对象添加到\_genericServices里；对于非泛型服务，按照ImplementationInstance，ImplementationFactory，以及ImplementationType这三个属性的优先级分别创建InstanceService，FactoryService，Service类型的对象添加到\_services字段里，所以当注册一个服务，却指定了以上三个属性里的多个，最后生效的只有一个。

最后，从ServiceProvider的构造函数可以看到，除了我们自己注册的服务，实现还分别添加了这三个服务：

* IServiceProvider：以Transient形式注册，用于返回服务容器自身，对应的IService实现类是ServiceProviderService；
* IServiceScopeFactory：以Scoped形式注册，用于创建作用域，对应的IService实现类型是ServiceScopeService；
* IEnumerable<>：以Transient形式注册，用于支持返回同一个服务的多个实现。，对应的IGenericService类型是OpenIEnumerableService；

待续...