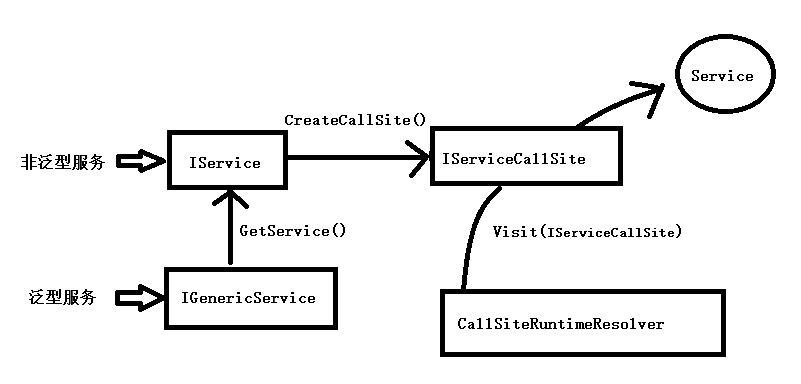
### .Net Core 依赖注入实现原理（三）-服务解析

1. 总体流程

[上一篇服务的注册](http://justing.me/article/20160911002630)里讲到最初注册的服务信息，最终以转化成以两个内部的接口IService和IGenericService的形式存储在了ServiceTable里，后续的服务解析就是在这两个接口的基础上进行的。具体地说，服务的解析涉及到了以下几个类：

* IService；
* IGenericService；
* IServiceCallSite；
* CallSiteRuntimeResolver：负责实际的服务解析和创建工作；

这些类都是内部的，他们的具体关系如下图：



对于非泛型服务，由IService创建IServiceCallSite，对于泛型服务，由IGenericService得到IService进而创建IServiceCallSite，最后由CallSiteRuntimeResolver解析（访问）IServiceCallSite得到最终的服务实例。

二．非泛型服务解析

先来看看对于非泛型服务的解析，上一篇讲到对于非泛型的服务，添加到ServiceTable里的IService有三个实现类：

* InstanceService：针对指定了ServiceDescriptor的ImplementationInstance属性的情况；
* FactoryService： 针对对指定了ServiceDescriptor的ImplementationFactory属性的情况；
* Service：针对对指定了ServiceDescriptor的ImplementationType属性的情况；

根据上图，看一下它们的CreateCallSite方法分别返回什么，首先是InstanceService：

internal class InstanceService : IService, IServiceCallSite

{

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

{

return this;

}

}

它直接返回自身，因为它同时也实现了IServiceCallSite。下面是FactoryService：

internal class FactoryService : IService, IServiceCallSite

{

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

{

return this;

}

}

它也是直接返回自身。最后是Service，由于它里面的代码比较长，所以我这边以注释及伪代码的方式解释里面的大致工作：

internal class Service : IService

{

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

{

// 获取服务实现类型的最佳构造函数。

// 如果不存在这样的构造函数，则抛异常。

var bestConstructor = GetBestConstructor(descriptor.ImplementationType);

if(bestConstructor == null) throw error;

// 根据构造函数的参数来构建用于解析构造函数参数的IServiceCallSite 数组

var parameters = bestConstructor.GetParameters();

var parameterCallSites = PopulateCallSites(parameters);

// 根据构造函数的参数个数来决定创建不同的IServiceCallSite

return parameterCallSites.Length == 0 ?

(IServiceCallSite)new CreateInstanceCallSite(\_descriptor) :

new ConstructorCallSite(bestConstructor, parameterCallSites);

}

}

我们知道默认实现的注入方式是基于构造函数的，当有多个构造函数的时候需要选择一个最合适的构造函数来注入，这个最佳构造函数的选择标准在第一篇里有讲，这里不再赘述。另外由于构造函数的参数也需要由服务容器进行解析，所以每个参数也需要创建一个对应的IServiceCallSite，这个过程和当前的过程一致，所以这是个递归的过程。最后根据构造函数的参数数目的不同，创建不同的IServiceCallSite：CreateInstanceCallSite和ConstructorCallSite，分别针对无参数和有参数的情况。

创建了IServiceCallSite之后需要将它传给CallSiteRuntimeResolver进行解析，这里的解析过程采用了访问者模式，我们先来看CallSiteRuntimeResolver对于InstanceService的访问：

protected override object VisitInstanceService(InstanceService instanceCallSite, ServiceProvider provider)

{

return instanceCallSite.Descriptor.ImplementationInstance;

}

针对InstanceService的访问直接就返回了Descriptor.ImplementationInstance属性作为最终解析到的服务实例。再来看一下对于FactoryService的访问：

protected override object VisitFactoryService(FactoryService factoryService, ServiceProvider provider)

{

return factoryService.Descriptor.ImplementationFactory(provider);

}

同样也很简单，它直接返回由Descriptor.ImplementationFactory服务工厂创建的服务实例。再来看对构造函数注入并且无参情况的CreateInstanceCallSite的访问：

protected override object VisitCreateInstance(CreateInstanceCallSite createInstanceCallSite, ServiceProvider provider)

{

return Activator.CreateInstance(createInstanceCallSite.Descriptor.ImplementationType);

}

由于不需要解析构造函数参数，所以直接返回了通过反射构造函数创建的服务实例。最后是针对有参构造函数情况的ConstructorCallSite的访问：

protected override object VisitConstructor(ConstructorCallSite constructorCallSite, ServiceProvider provider)

{

object[] parameterValues = new object[constructorCallSite.ParameterCallSites.Length];

for (var index = 0; index < parameterValues.Length; index++)

{

parameterValues[index] = VisitCallSite(constructorCallSite.ParameterCallSites[index], provider);

}

return constructorCallSite.ConstructorInfo.Invoke(parameterValues);

}

可以看到这边首先需要先解析到构造函数的参数，所以这边递归访问了构造函数参数对应的IServiceCallSite，得到所有参数对应的服务实例后，最终通过反射的方式创建了最终的服务实例。

三．解析泛型服务

IGenericService的默认实现类型是GenericService，它的GetService方法如下：

public IService GetService(Type closedServiceType)

{

Type[] genericArguments = closedServiceType.GetTypeInfo().GenericTypeArguments;

Type closedImplementationType =

\_descriptor.ImplementationType.MakeGenericType(genericArguments);

var closedServiceDescriptor = new ServiceDescriptor(closedServiceType, closedImplementationType, Lifetime);

return new Service(closedServiceDescriptor);

}

上面的代码印证了第一篇讲到的对于泛型服务的解析方式，比如我们注册了IGeneric<,>并且指定他的服务实现类是Generic<,>，当我们请求IGeneric<IFoo,IBar>类型的服务的时候，那么它的服务实现类性对应的就是Generic<IFoo,IBar>，得到这个对应关系之后，就像我们最初注册服务的方式创建了一个ServiceDescriptor对象，进而得到一个Service的对象。所以泛型服务IGenericService并不是真正意义上的服务，它的作用相当于创建IService的模板。得到IService之后，后续的解析工作就交给了IService，之后的流程就和非泛型服务的解析一样了。

1. 服务不能循环依赖

从上面的服务解析流程看到，当通过构造函数进行注入的时候，首先需要解析构造函数的参数对应的服务，这其实是一个递归的过程，这就限制了服务和服务之间不能循环依赖。比如，IBar的服务实现类Bar有个构造函数需要接收一个IFoo类型的参数，而IFoo的服务实现类Foo的构造函数又有一个IBar类型的参数这就构成了循环依赖。这种情况会导致递归永远进行下去，默认的实现当检测到这种情况的时候会抛出一个异常。

1. 性能

服务的解析过程中涉及到反射，比如查找服务实现类的最佳构造函数，这个过程相对来说耗性能，为了提高效率，只有在第一次解析的时候才需要进行这个工作，之后会以委托的形式缓存创建服务的方式，也就是流程图中最后一个阶段的过程。具体创建这个委托的代码在ServiceProvider的RealizeService方法里：

internal static Func<ServiceProvider, object> RealizeService(ServiceTable table, Type serviceType, IServiceCallSite callSite)

{

var callCount = 0;

return provider =>

{

if (Interlocked.Increment(ref callCount) == 2)

{

Task.Run(() =>

{

var realizedService = new CallSiteExpressionBuilder(\_callSiteRuntimeResolver)

.Build(callSite);

table.RealizedServices[serviceType] = realizedService;

});

}

return \_callSiteRuntimeResolver.Resolve(callSite, provider);

};

}

创建好的委托最终会还存在ServiceTabble的\_realizedServices里。我们还注意到最后一个环节创建服务实例也需要通过反射，为了进一步提高性能，当同一个服务被请求2次以上时，创建服务实例的方式会进行替换（这个过程是异步进行的）：使用动态代码生成来替换原来的委托，避免使用反射。这个机制也体现在上面的代码中（注意RealizeService方法返回的是一个闭包函数）。