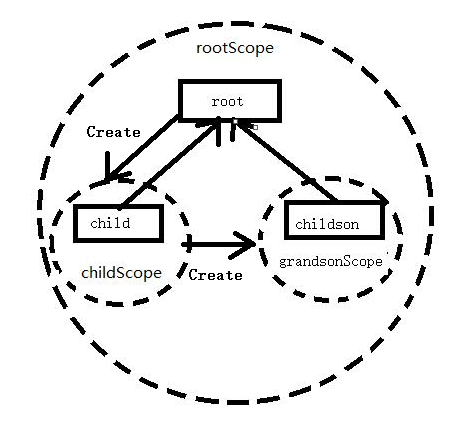
### .Net Core 依赖注入实现原理（四）-生命周期管理

一．作用域结构

在[第一篇](http://justing.me/article/20160823113540)里讲到，创建一个子作用域，需要向当前的服务容器IServiceProvider请求一个IServiceScopeFactory的服务，然后调用CreateScope方法得到一个IServiceScope对象，这个IServiceScope对象有一个ServiceProvider属性表示子作用域对应的服务容器。然后我们还提到作用域之间只有两层的父子关系，所有的子作用域都是同级的（无论它是通过根作用域创建的还是通过其他子作用域创建的），再次附上之前的图：



我们来看一下创建子作用域的过程是怎么样的。在[服务的注册](http://justing.me/article/20160911002630)那篇讲到IServiceScopeFactory是内部注册的服务，它对应的IService在ServiceTable里存储的实现类是ServiceScopeService，所以我们来看一下CallSiteRuntimeResolver是如何对他进行访问（解析服务）的：

protected override object VisitServiceScopeService(ServiceScopeService serviceScopeService, ServiceProvider provider)

{

return new ServiceScopeFactory(provider);

}

可以看到他直接返回了一个ServiceScopeFactory对象，并且将当前作用域下的服务容器传给了它。接着来看一下ServiceScopeFactory的CreateScope方法的实现：

internal class ServiceScopeFactory : IServiceScopeFactory

{

private readonly ServiceProvider \_provider;

public ServiceScopeFactory(ServiceProvider provider)

{

\_provider = provider;

}

public IServiceScope CreateScope()

{

return new ServiceScope(new ServiceProvider(\_provider));

}

}

它创建了一个ServiceScope对象作为子作用域，还创建了一个ServiceProvider对象，将它作为子作用域下的服务容器。注意到新的服务容器的构造函数接受父作用域里的服务容器作为参数，这个构造函数的逻辑如下：

internal ServiceProvider(ServiceProvider parent)

{

Root = parent.Root;

\_table = parent.\_table;

}

新创建的服务容器的Root属性被指向了根作用域，由此可见所有的子作用域下的服务容器它们都引用了根作用域下的服务容器，所以才有了上面的作用域结构图。这个结构方便了服务生命周期的管理，因为我们知道对于Singleton形式的服务它的生命周期由根服务容器管理，无论在哪个作用域下请求它们，都会把解析的工作指派给根服务容器。

1. 生命周期管理

我们根据服务的生命周期类型分别讨论它们的生命周期管理策略。[上一篇](http://justing.me/article/20160911010617)对于服务的创建工作涉及了不同类型的IServiceCallSite，然后利用CallSiteRuntimeResolver对IServiceCallSite进行访问最终得到服务实例。对于生命周期的管理逻辑，同样也是交给了不同的IServiceCallSite，三种生命周期类型分别对应了三个不同的IServiceCallSite：

TransientCallSite，ScopedCallSite和SingletonCallSite，他们的创建逻辑在ServiceProvider的GetResolveCallSite方法里：

internal IServiceCallSite GetResolveCallSite(IService service, ISet<Type> callSiteChain)

{

IServiceCallSite serviceCallSite = service.CreateCallSite(this, callSiteChain);

if (service.Lifetime == ServiceLifetime.Transient)

{

return new TransientCallSite(serviceCallSite);

}

else if (service.Lifetime == ServiceLifetime.Scoped)

{

return new ScopedCallSite(service, serviceCallSite);

}

else

{

return new SingletonCallSite(service, serviceCallSite);

}

}

可以看到这三个IServiceCallSite都包装了由IService创建的IServiceCallSite。同样的，生命周期的具体管理逻辑需要看一下CallSiteRuntimeResolver对于这三个IServiceCallSite的不同访问方式。

1. TransientCallSite

对于TransientCallSite的访问逻辑如下：

protected override object VisitTransient(TransientCallSite transientCallSite, ServiceProvider provider)

{

var service = VisitCallSite(transientCallSite.Service, provider);

return provider.CaptureDisposable(service);

}

首先需要在当前作用域下的服务容器创建服务实例，然后调用服务容器的CaptureDisposable方法，这个方法的逻辑是这样的：如果创建的服务容器实现了IDispose接口，那么就将它添加到当前的服务容器里。下面是CaptureDisposable的代码大致逻辑（注意服务是添加到\_transientDisposables字段里的）：

internal object CaptureDisposable(object service)

{

var disposable = service as IDisposable;

if (disposable != null)

{

if (\_transientDisposables == null)

{

\_transientDisposables = new List<IDisposable>();

}

\_transientDisposables.Add(disposable);

}

return service;

}

1. ScopedCallSite

对于TransientCallSite ，首先判断在当前服务容器下有没有该类型的服务实例（之前是否已被创建过），如果没有，则创建并添加到当前的服务容器里，代码逻辑如下（注意服务是添加到ResolvedServices属性里的）：

protected override object VisitScoped(ScopedCallSite scopedCallSite, ServiceProvider provider)

{

object resolved;

lock (provider.ResolvedServices)

{

if (!provider.ResolvedServices.TryGetValue(scopedCallSite.Key, out resolved))

{

resolved = VisitCallSite(scopedCallSite.ServiceCallSite, provider);

provider.ResolvedServices.Add(scopedCallSite.Key, resolved);

}

}

return resolved;

}

1. SingletonCallSite

前面说到，对于Singleton类型的服务，解析工作会指派到根服务容器下，所以它的逻辑很简单：

protected override object VisitSingleton(SingletonCallSite singletonCallSite, ServiceProvider provider)

{

return VisitScoped(singletonCallSite, provider.Root);

}

解析逻辑和Scopeed类型的服务一样，唯一的不同是显式指定了根服务容器，这样该服务的生命周期管理及创建工作都交给了根服务容器。

综上所述，我们回过头来看一下[第一篇](http://justing.me/article/20160823113540)总结的三条结论：

* 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；
* 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；
* 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；

1. 服务的销毁

服务的生命周期结束于服务的销毁，一般来说我们不会主动调用服务的Dispose方法，相反我们要销毁服务，往往是销毁接管它生命周期的作用域，也就是IServiceScope对象。当我们调用IServiceScope对象的Dispose方法时，他会继而调用对应的服务容器的Dispose方法（我们一般不会直接通过IServiceProvider调用，因为这个接口不实现IDispose），而服务容器又会进一步销毁所有由它引用的之前创建的服务实例，下面是ServiceProvider的Dispose方法大致逻辑：

public void Dispose()

{

if (\_transientDisposables != null)

{

foreach (var disposable in \_transientDisposables)

{

disposable.Dispose();

}

\_transientDisposables.Clear();

}

foreach (var entry in ResolvedServices)

{

(entry.Value as IDisposable)?.Dispose();

}

ResolvedServices.Clear();

}

我们又看到了这两个熟悉的ServiceProvider成员：\_transientDisposables和ResolvedServices，前者引用了所有Transient类型并且实现了IDispose接口的服务实例，后者引用了所有Scoped类型的服务实例。到此为止，我们再来看一下[第一篇](http://justing.me/article/20160823113540)总结的服务使用的准则：

* 如果服务是以Transient方式注册的，并且服务实例实现了IDisposable，那么应该在子作用域内创建，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；
* 如果服务是以Scoped方式注册的，那么应该在子作用域内进行创建，否则它会以单例的形式一直存在根服务容器中；
* 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；