عنوان: **پیاده سازی authorization به روش AOP به کمک کتابخانه های SNAP و StructureMap**
نویسنده: کاوه شهبازی
تاریخ: ۱۳۹۲/۰۸/۰۱ ۱۷:۵۵
آدرس: www.dotnettips.info
برچسب‌ها: Authorization, AOP, SNAP, StructureMap

همانطور که پیشتر در این مقاله بحث شده است، بوسیله AOP می‌توان قابلیت‌هایی که قسمت عمده‌ای از برنامه را تحت پوشش قرار می‌دهند، کپسوله کرد. یکی از قابلیت‌هایی که در بخشهای مختلف یک سیستم نرم‌افزاری مورد نیاز است، Authorization یا اعتبارسنجی‌ست. در ادامه به بررسی یک پیاده‌سازی به این روش می‌پردازیم.

کتابخانه SNAP

کتابخانه SNAP به گفته سازنده آن، با یکپارچه‌سازی AOP با IoC Containerهای محبوب، برنامه‌نویسی به این سبک را ساده می‌کند. این کتابخانه هم اکنون علاوه بر structureMap از IoC Providerهای Autofac, Ninject, LinFu و Castle Windsor نیز پشتیبانی میکند.

دریافت SNAP.StructureMap

برای دریافت آن نیاز است دستور پاورشل ذیل را در کنسول نیوگت ویژوال استودیو اجرا کنید:

```
PM> Install-Package snap.structuremap
```

پس از اجرای دستور فوق، کتابخانه SNAP.StructureMap که در زمان نگارش این مطلب نسخه 1.8.0 آن موجود است به همراه کلیه نیازمندی‌های آن که شامل موارد زیر می‌باشد نصب خواهد شد.

```
StructureMap (≥ 2.6.4.1)
CommonServiceLocator.StructureMapAdapter (≥ 1.1.0.3)
SNAP (≥ 1.8)
fasterflect (≥ 2.1.2)
Castle.Core (≥ 3.1.0)
CommonServiceLocator (≥ 1.0)
```

تنظیمات SNAP

از آنجا که تنظیمات SNAP همانند تنظیمات StructureMap تنها باید یک بار اجرا شود، بهترین جا برای آن در یک برنامه وب، Application_Start فایل Global.asax است.

```
namespace Framework.UI.Asp
{
    public class Global : HttpApplication
    {
        void Application_Start(object sender, EventArgs e)
        {
            initSnap();
            initStructureMap();
        }

        private static void initSnap()
        {
            SnapConfiguration.For<StructureMapAspectContainer>(c =>
            {
                // Tell Snap to intercept types under the "Framework.ServiceLayer..." namespace.
                c.IncludeNamespace("Framework.ServiceLayer.*");
                // Register a custom interceptor (a.k.a. an aspect).
                c.Bind<Framework.ServiceLayer.Aspects.AuthorizationInterceptor>()
                .To<Framework.ServiceLayer.Aspects.AuthorizationAttribute>();
            });
        }

        void Application_EndRequest(object sender, EventArgs e)
        {
            ObjectFactory.ReleaseAndDisposeAllHttpScopedObjects();
```

```
        }

        private static void initStructureMap()
        {
            var thread = StructureMap.Pipeline.Lifecycles.GetLifecycle(InstanceScope.HttpSession);
            ObjectFactory.Configure(x =>
            {
                x.For<IUserManager>().Use<EFUserManager>();
                x.For<IAuthorizationManager>().LifecycleIs(thread)
                 .Use<EFAuthorizationManager>().Named("_AuthorizationManager");
                x.For<Framework.DataLayer.IUnitOfWork>()
                 .Use<Framework.DataLayer.Context>();

                x.SetAllProperties(y =>
                {
                    y.OfType<IUserManager>();
                    y.OfType<Framework.DataLayer.IUnitOfWork>();
                    y.OfType<Framework.Common.Web.IPageHelpers>();
                });
            });
        }
    }
}
```

بخش اعظم کدهای فوق در مقاله‌های « استفاده از StructureMap به عنوان یک IoC Container » و « تزریق خودکار وابستگی‌ها در برنامه‌های ASP.NET Web forms » شرح داده شده‌اند، تنها بخش جدید متد initSnap() است، که خط اول آن به snap می‌گوید همه کلاس‌هایی که در فضای نام Framework.ServiceLayer و زیرمجموعه‌های آن هستند را پوشش دهد. خط دوم نیز کلاس AuthorizationInterceptor را به عنوان مرجعی برای handle کردن AuthorizationAttribute معرفی می‌کند.

در ادامه به بررسی کلاس AuthorizationInterceptor می‌پردازیم.

```
namespace Framework.ServiceLayer.Aspects
{
    public class AuthorizationInterceptor : MethodInterceptor
    {
        public override void InterceptMethod(IInvocation invocation, MethodBase method, Attribute
attribute)
        {
            var AuthManager = StructureMap.ObjectFactory
.GetInstance<Framework.ServiceLayer.UserManager.IAuthorizationManager>();
            var FullName = GetMethodFullName(method);
            if (!AuthManager.IsActionAuthorized(FullName))
                throw new Common.Exceptions.UnauthorizedAccessException("");

            invocation.Proceed(); // the underlying method call
        }

        private static string GetMethodFullName(MethodBase method)
        {
            var TypeName = (((System.Reflection.MemberInfo)(method)).DeclaringType).FullName;
            return TypeName + "." + method.Name;
        }
    }

    public class AuthorizationAttribute : MethodInterceptAttribute
    { }
```

کلاس مذکور از کلاس MethodInterceptor کتابخانه snap ارث بری کرده و متد InterceptMethod را تحریف میکند. این متد، کار اجرای متد اصلی ای که با این Aspect تزئین شده را بر عهده دارد. بنابراین می‌توان پیش از اجرای متد اصلی، اعتبارسنجی را انجام داد. **کلاس MethodInterceptor**

کلاس MethodInterceptor شامل چندین متد دیگر نیز هست که میتوان برای سایر مقاصد از جمله مدیریت خطا و Event logging از آنها استفاده کرد.

```
namespace Snap
{
```

```
    public abstract class MethodInterceptor : IAttributeInterceptor, IInterceptor, IHideBaseTypes
    {
        protected MethodInterceptor();

        public int Order { get; set; }
        public Type TargetAttribute { get; set; }

        public virtual void AfterInvocation();
        public virtual void BeforeInvocation();
        public void Intercept(IInvocation invocation);
        public abstract void InterceptMethod(IInvocation invocation, MethodBase method, Attribute
attribute);
        public bool ShouldIntercept(IInvocation invocation);
    }
}
```

یک نکته

نکته مهمی که در اینجا پیش می‌آید این است که برای اعتبارسنجی، کد کاربری شخصی که لاگین کرده، باید به طریقی در اختیار متد IsActionAuthorized() قرار بگیرد. برای این کار می‌توان در یک HttpMudole به عنوان مثال همان ماژولی که برای تسهیل در کار تزریق خودکار وابستگی‌ها در سطح فرم‌ها استفاده می‌شود، با استفاده از امکانات structureMap به وهله‌ی ایجاد شده از AuthorizationManager (که با کمک structureMap با طول عمر InstanceScope.HttpSession ساخته شده است) دسترسی پیدا کرده و خاصیت مربوطه را مقداردهی کرد.

```
private void Application_PreRequestHandlerExecute(object source, EventArgs e)
        {
            var page = HttpContext.Current.Handler as BasePage; // The Page handler
            if (page == null)
                return;

            WireUpThePage(page);
            WireUpAllUserControls(page);

            var UsrCod = HttpContext.Current.Session["UsrCod"];
            if (UsrCod != null)
            {
                var _AuthorizationManager = ObjectFactory

.GetNamedInstance<Framework.ServiceLayer.UserManager.IAuthorizationManager>("_AuthorizationManager");

                ((Framework.ServiceLayer.UserManager.EFAuthorizationManager)_AuthorizationManager)
                    .AuditUserId = UsrCod.ToString();
            }
        }
```

روش استفاده

نحوه استفاده از Aspect تعریف شده در کد زیر قابل مشاهده است:

```
namespace Framework.ServiceLayer.UserManager
{
    public class EFUserManager : IUserManager
    {
        IUnitOfWork _uow;
        IDbSet<User> _users;

        public EFUserManager(IUnitOfWork uow)
        {
            _uow = uow;
            _users = _uow.Set<User>();
        }

        [Framework.ServiceLayer.Aspects.Authorization]
        public List<User> GetAll()
        {
            return _users.ToList<User>();
        }
    }
}
```

نظرات خوانندگان

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۰۸/۰۱ ۱۹:۷

با تشکر از شما. چند سؤال: متد AuthManager.IsActionAuthorized چطور تعریف شده؟ و همچنین EFAuthorizationManager حدودا چه تعریفی داره؟

نویسنده: کاوه شهبازی
تاریخ: ۱۳۹۲/۰۸/۰۱ ۱۹:۵۱

۱- متد IsActionAuthorized نام کامل متدی که قرار است اجرا شود را به عنوان پارامتر گرفته و در دیتابیس (در این پیاده سازی به وسیله‌ی EntityFramework) چک میکند که کاربری که Id اش در AuthManager.AuditUserId است (یعنی کاربری که درخواست اجرای متد را داده است) اجازه اجرای این متد را دارد یا نه. بسته به نیازمندی برنامه شما این دسترسی میتواند به طور ساده فقط مستقیما برای کاربر ثبت شود و یا ترکیبی از دسترسی خود کاربر و دسترسی گروه هایی که این کاربر در آن عضویت دارد باشد.

۲- EFAuthorizationManager کلاس ساده ایست

```
namespace Framework.ServiceLayer.UserManager
{
    public class EFAuthorizationManager : IAuthorizationManager
    {
        public String AuditUserId { get; set; }
        IUnitOfWork _uow;

        public EFAuthorizationManager(IUnitOfWork uow)
        {
            _uow = uow;
        }

        public bool IsActionAuthorized(string actionName)
        {
            var res = _uow.Set<User>()
            .Any(u => u.Id == AuditUserId &&
                    u.AllowedActions.Any(a => a.Name == actionName));
            return res;
        }

        public bool IsPageAuthorized(string pageURL)
        {
            //TODO: بررسی وجود دسترسی باید پیاده سازی شود
            //فقط برای تست
            return true;
        }
    }
}
```

خلاصه ای از کلاسهای مدل مرتبط را هم در زیر مشاهده می‌کنید:

```
namespace Framework.DataModel
{
    public class User : BaseEntity
    {
        public string UserName { get; set; }
        public string Password { get; set; }

        //...

        [Display(Name = "عملیات مجاز")]
        public virtual ICollection<Action> AllowedActions { get; set; }
    }

    public class Action:BaseEntity
    {
        public string Name { get; set; }
        public Entity RelatedEntity { get; set; }
```

```
        //...

        public virtual ICollection<User> AllowedUsers { get; set; }
    }

    public abstract class BaseEntity
    {
        [Key]
        public int Id { get; set; }
        //...
    }
}
```

عنوان: AOP با استفاده از Microsoft Unity
نویسنده: مسعود پاکدل
تاریخ: ۱۳۹۲/۰۸/۰۶ ۱۳:۵
آدرس: www.dotnettips.info
برچسب‌ها: UnityContainer, AOP

چند روز پیش فرصتی پیش آمد تا بتوانم مروری بر مطلب منتشر شده درباره AOP داشته باشم. به حق مطلب مورد نظر، بسیار خوب و مناسب شرح داده شده بود و همانند سایر مقالات جناب نصیری چیزی کم نداشت. اما امروز قصد پیاده سازی یک مثال AOP، با استفاده از Microsoft Unity Application Block را به عنوان IOC Container دارم. اگر شما هم، مانند من از UnityContainer به عنوان IOC Container در پروژه‌های خود استفاده می‌کنید نگران نباشید. این کتابخانه به خوبی از مباحث Interception پشتیبانی می‌کند. در ادامه طی یک مقاله این مورد را با هم بررسی می‌کنیم.
برای دوستانی که با AOP آشنایی ندارند پیشنهاد می‌شود ابتدا مطلب مورد نظر را یک بار مطالعه نمایند.
برای شروع یک پروژه در VS.Net بسازید و ارجاع به اسمبلی‌های زیر را در پروژه فراموش نکنید:

«Microsoft.Practices.EnterpriseLibrary.Common
«Microsoft.Practices.Unity
«Microsoft.Practices.Unity.Configuration
«Microsoft.Practices.Unity.Interception
«Microsoft.Practices.Unity.Interception.Configuration

یک اینترفیس به نام IMyOperation بسازید:

```
public interface IMyOperation
    {
        void DoIt();
    }
```

کلاسی می‌سازیم که اینترفیس بالا را پیاده سازی نماید:

```
public void DoIt()
  {
     Console.WriteLine( "this is main block of code" );
  }
```

قصد داریم با استفاده از AOP یک سری کد مورد نظر خود(در این مثال کد لاگ کردن عملیات در یک فایل مد نظر است) را به کدهای متدهای مورد نظر تزریق کنیم. یعنی با فراخوانی این متد کدهای لاگ عملیات در یک فایل ذخیره شود بدون تکرار یا فراخوانی دستی متد لاگ.
ابتدا یک کلاس برای لاگ عملیات می‌سازیم:

```
public class Logger
    {
        const string path = @"D:\Log.txt";

        public static void WriteToFile( string methodName )
        {
            object lockObject = new object();
            if ( !File.Exists( path ) )
            {
                File.Create( path );
            }
            lock ( lockObject )
            {
                using ( TextWriter writer = new StreamWriter( path , true ) )
                {
                    writer.WriteLine( string.Format( "{0} at {1}" , methodName , DateTime.Now ) );
                }
            }
        }
    }
```

حال نیاز به یک Handler برای مدیریت فراخوانی کدهای تزریق شده داریم. برای این کار یک کلاس می‌سازیم که اینترفیس ICallHandler را پیاده سازی نماید.

```
public class LogHandler : ICallHandler
    {
        public IMethodReturn Invoke( IMethodInvocation input , GetNextHandlerDelegate getNext )
        {
            Logger.WriteToFile( input.MethodBase.Name );

            var methodReturn = getNext()( input , getNext );

            return methodReturn;
        }

        public int Order { get; set; }
    }
```

کلاس بالا یک متد به نام Invoke دارد که فراخوانی متدهای مورد نظر برای تزریق کد را در دست خواهد گرفت. در این متد ابتدا عملیات لاگ در فایل مورد نظر ثبت می‌شود(با استفاده از Logger.WriteToFile). سپس با استفاده از getNext که از نوع GetNextHandlerDelegate است، اجرا را به کدهای اصلی برنامه منتقل می‌کنیم.

```
var methodReturn = getNext()( input , getNext );
```

برای مدیریت بهتر عملیات لاگ یک Attribute می‌سازیم که فقط متد هایی که نیاز به لاگ کردن دارند را مشخص کنیم. به صورت زیر:

```
public class LogAttribute : HandlerAttribute
    {
        public override ICallHandler CreateHandler( Microsoft.Practices.Unity.IUnityContainer container
)
        {
            return new LogHandler();
        }
    }
```

فقط دقت داشته باشید که کلاس مورد نظر به جای ارث بری از کلاس Attribute باید از کلاس HandlerAttribute که در فضای نام Microsoft.Practices.Unity.InterceptionExtension تعبیه شده است ارث ببرد(خود این کلاس از کلاس Attribute ارث برده است). کافیست در متد CreateHandler آن که Override شده است یک نمونه از کلاس LogHandler را برگشت دهیم.
برای آماده سازی Ms Unity جهت عملیات Interception باید کدهای زیر در ابتدا برنامه قرار داده شود:

```
var  unityContainer = new UnityContainer();

 unityContainer.AddNewExtension<Interception>();

  unityContainer.Configure<Interception>().SetDefaultInterceptorFor<IMyOperation>( new
InterfaceInterceptor() );

  unityContainer.RegisterType<IMyOperation, MyOperation>();
```

توضیح چند مطلب:
بعد از نمونه سازی از کلاس UnityContainer باید Interception به عنوان یک Extension به این Container اضافه شود. سپس با استفاده از متد Configure برای اینترفیس IMyOperation یک Interceptor پیش فرض تعیین می‌کنیم. در پایان هم به وسیله متد RegisterType کلاس MyOperation به اینترفیس IMyOperation رجیستر می‌شود. از این پس هر گاه درخواستی برای اینترفیس IMyOperation از unityContainer شود یک نمونه از کلاس MyOperation در اختیار خواهیم داشت.
به عنوان نکته آخر متد DoIt در اینترفیس بالا باید دارای LogAttribute باشد تا عملیات مزین سازی با کدهای لاگ به درستی انجام شود.

یک نکته تکمیلی:
در هنگام مزین سازی متد set خاصیت ها، به دلیل اینکه اینترفیسی برای این کار وجود ندارد باید مستقیما عملیات AOP به خود کلاس اعمال شود. برای این کار باید به صورت زیر عمل نمود:

```
var container = new UnityContainer();
container.RegisterType<Book , Book>();

container.AddNewExtension<Interception>();

 var policy = container.Configure<Interception>().SetDefaultInterceptorFor<Book>( new
VirtualMethodInterceptor() ).AddPolicy( "MyPolicy" );

  policy.AddMatchingRule( new PropertyMatchingRule( "*" , PropertyMatchingOption.Set ) );
  policy.AddCallHandler<Handler.NotifyChangedHandler>();
```

همان طور که مشاهده می‌کنید عملیات Interception مستقیما برای کلاس پیکر بندی می‌شود و به جای InterfaceInterceptor از VirtualMethodInterceptor برای تزریق کد به بدنه متدها استفاده شده است. در پایان نیز با تعریف یک Policy می‌توانیم به راحتی(با استفاده از "*") متد Set تمام خواص کلاس را به NotifyChangedHandler مزین نماییم.

سورس کامل مثال بالا

عنوان: **AOP و پردازش فراخوانی‌های تو در تو**
نویسنده: مهتدی حسن‌پور
تاریخ: ۱۳۹۲/۰۸/۲۸ ۱۳:۱۵
آدرس: www.dotnettips.info
برچسب‌ها: AOP, StructureMap

هنگامی‌که از روش AOP استفاده می‌کنیم گاهی نیاز است متد تزیین‌شده را از متدی درون خود کلاس فراخوانی کنیم و می‌خواهیم aspectهای آن متد نیز فراخوانی شوند.

پیش‌نیاز : دوره‌ی AOP

(برای سادگی کار از تعریف attribute خودداری کردم. شما می‌توانید با توجه به آموزش، attributeهای دلخواه را به متدها بیافزایید).

Interface و کلاس پیاده‌سازی‌شده‌ی آن در لایه سرویس:

```
public interface IMyService
{
  void foo();
  void bar();
}

public class MyService : IMyService
{
  public void foo()
  {
    Console.Write("foo");
    bar();
  }

  public void bar()
  {
    Console.Write("bar");
  }
}
```

نام متد در خروجی نوشته می‌شود. همچنین می‌خواهیم پیش از فراخوانی این متدها، متنی در خروجی نوشته شود.

آماده‌سازی Interceptor

یک interceptor ساده که نام متد را در خروجی می‌نویسد.

```
//using Castle.DynamicProxy;

public class Interceptor : IInterceptor
{
  public void Intercept(IInvocation invocation)
  {
    Console.WriteLine("Intercepted: " + invocation.Method.Name);
    invocation.Proceed();
  }
}
```

معرفی Interceptor به سیستم

همانند قبل:

```
//using System;
//using Castle.DynamicProxy;
//using StructureMap;

class Program
    {
        static void Main(string[] args)
        {
            ObjectFactory.Initialize(x =>
            {
                var dynamicProxy = new ProxyGenerator();
                x.For<IMyService>()
                 .EnrichAllWith(myTypeInterface =>
                                dynamicProxy.CreateInterfaceProxyWithTarget(myTypeInterface, new
Intercept()))
                 .Use<MyService>();
```

```
            });

            var myService = ObjectFactory.GetInstance<IMyService>();
            myService.foo();
        }
    }
```

انتظار ما این است که خروجی زیر تولید شود:

```
Intercepted foo
foo
Intercepted bar
bar
```

اما نتیجه این می‌شود که دلخواه ما نیست:

```
Intercepted foo
foo
bar
```

راه‌حل

برای حل این مشکل دو کار باید انجام داد:
-1 متد تزیین‌شده باید **virtual** باشد.

```
public class MyService : IMyService
{
  public virtual void foo()
  {
    Console.Write("foo");
    bar();
  }

  public virtual void bar()
  {
    Console.Write("foo");
    bar();
  }
}
```

-2 شیوه معرفی متد به سیستم باید به روش زیر باشد:

```
// جایگزین روش پیشین در متد Main
x.For<IMyService>()
                    .EnrichAllWith(myTypeInterface  => dynamicProxy.CreateClassProxy<MyService>(new
Intercept()))
                    .Use<MyService>();
```

دلیل این مسئله به دو روش proxy برمی‌گردد که برای اطلاع بیشتر به مستندات پروژه Castle مراجعه کنید.
در این‌جا روش Inheritance-based به کار رفته است. در این روش، تنها متدهای **virtual** را می‌توان intercept کرد. در روش پیشین(Composition-based) برای همه‌ی متدها عملیات intercept انجام می‌شد (کلاس proxy پیاده‌سازی‌شده‌ی interface ما بود) که در این‌جا این‌گونه نیست و می‌تواند به سرعت برنامه کمک کند.

عنوان: **پیاده سازی INotifyPropertyChanged با استفاده از Unity Container**
نویسنده: برات جوادی
تاریخ: ۱۳۹۴/۰۱/۲۴ ۱۴:۲۵
آدرس: www.dotnettips.info
گروه‌ها: WPF, UnityContainer, AOP

AOP چیست

AOP یکی از فناوری‌های مرتبط با توسعه نرم افزار محسوب می‌شود که توسط آن می‌توان اعمال مشترک و متداول موجود در برنامه را در یک یا چند ماژول مختلف قرار داد (که به آن‌ها Aspects نیز گفته می‌شود) و سپس آن‌ها را به مکان‌های مختلفی در برنامه متصل ساخت. عموما Aspects، قابلیت‌هایی را که قسمت عمده‌ای از برنامه را تحت پوشش قرار می‌دهند، کپسوله می‌کنند. اصطلاحا به این نوع قابلیت‌های مشترک، تکراری و پراکنده مورد نیاز در قسمت‌های مختلف برنامه، Cross cutting concerns نیز گفته می‌شود؛ مانند اعمال ثبت وقایع سیستم، امنیت، مدیریت تراکنش‌ها و امثال آن. با قرار دادن این نیازها در Aspects مجزا، می‌توان برنامه‌ای را تشکیل داد که از کدهای تکراری عاری است.

پیاده سازی INotifyPropertyChanged یکی از این مسائل می‌باشد که می‌توان آن را در یک Aspect محصور و در ماژول‌های مختلف استفاده کرد.

مسئله:

کلاس زیر مفروض است:

```
public class Foo
{
        public virtual int Id { get; set; }

        public virtual string Name { get; set; }
}
```

اکنون می‌خواهیم کلاس Foo را به INotifyPropertyChanged مزین، و یک Subscriber به قسمت set پراپرتی‌های کلاس تزریق کنیم.

راه حل:

ابتدا پکیج‌های Unity را از Nuget دریافت کنید:

```
PM> Install-Package Unity.Interception
```

این پکیج وابستگی‌های خود را که Unity و CommonServiceLocator هستند نیز دریافت می‌کند.

حال یک Interceptor که اینترفیس IInterceptionBehavior را پیاده سازی می‌کند، می‌نویسیم:

```
namespace NotifyPropertyChangedInterceptor.Interceptions
{
    using System;
    using System.Collections.Generic;
    using System.ComponentModel;
    using System.Reflection;
    using Microsoft.Practices.Unity.InterceptionExtension;

    class NotifyPropertyChangedBehavior : IInterceptionBehavior
    {
        private event PropertyChangedEventHandler PropertyChanged;

        private readonly MethodInfo _addEventMethodInfo =
            typeof(INotifyPropertyChanged).GetEvent("PropertyChanged").GetAddMethod();

        private readonly MethodInfo _removeEventMethodInfo =
            typeof(INotifyPropertyChanged).GetEvent("PropertyChanged").GetRemoveMethod();


        public IMethodReturn Invoke(IMethodInvocation input, GetNextInterceptionBehaviorDelegate
getNext)
        {
            if (input.MethodBase == _addEventMethodInfo)
            {
                return AddEventSubscription(input);
            }
```

```
            if (input.MethodBase == _removeEventMethodInfo)
            {
                return RemoveEventSubscription(input);
            }

            if (IsPropertySetter(input))
            {
                return InterceptPropertySet(input, getNext);
            }

            return getNext()(input, getNext);
        }

        public bool WillExecute
        {
            get { return true; }
        }

        public IEnumerable<Type> GetRequiredInterfaces()
        {
            yield return typeof(INotifyPropertyChanged);
        }

        private IMethodReturn AddEventSubscription(IMethodInvocation input)
        {
            var subscriber = (PropertyChangedEventHandler)input.Arguments[0];
            PropertyChanged += subscriber;

            return input.CreateMethodReturn(null);
        }

        private IMethodReturn RemoveEventSubscription(IMethodInvocation input)
        {
            var subscriber = (PropertyChangedEventHandler)input.Arguments[0];
            PropertyChanged -= subscriber;

            return input.CreateMethodReturn(null);
        }

        private bool IsPropertySetter(IMethodInvocation input)
        {
            return input.MethodBase.IsSpecialName && input.MethodBase.Name.StartsWith("set_");
        }

        private IMethodReturn InterceptPropertySet(IMethodInvocation input,
GetNextInterceptionBehaviorDelegate getNext)
        {
            var propertyName = input.MethodBase.Name.Substring(4);

            var subscribers = PropertyChanged;
            if (subscribers != null)
            {
                subscribers(input.Target, new PropertyChangedEventArgs(propertyName));
            }

            return getNext()(input, getNext);
        }
    }
}
```

متد Invoke : این متد Behavior مورد نظر را پردازش می‌کند (در اینجا، تزریق یک Subscriber در قسمت set پراپرتی ها). **متد GetRequiredInterfaces** : یک روش است برای یافتن کلاس هایی که با اینترفیس IInterceptionBehavior مزین شده‌اند.
پراپرتی WillExecute : ابن پراپرتی به Unity می‌گوید که این Behavior اعمال شود یا نه. اگر مقدار برگشتی آن false باشد، متد Invoke اجرا نخواهد شد.
همانطور که در متد Invoke مشاهد می‌کنید، شرط هایی برای افزودن و حذف یک Subscriber و چک کردن متد set نوشته شده و در غیر این صورت کنترل به متد بعدی داده می‌شود.

اتصال Interceptor به کلاس ها

در ادامه Unity را برای ساخت یک نمونه از کلاس پیکربندی می‌کنیم:

```
var container = new UnityContainer();

container.RegisterType<Foo, Foo>(
                new AdditionalInterface<INotifyPropertyChanged>(),
                new Interceptor<VirtualMethodInterceptor>(),
                new InterceptionBehavior<NotifyPropertyChangedBehavior>())
                .AddNewExtension<Interception>();
```

توسط متد RegisterType یک Type را با پیکربندی دلخواه به Unity معرفی می‌کنیم. در اینجا به ازای درخواست Foo (اولین پارامتر جنریک)، یک Foo (دومین پارامتر جنریک) برگشت داده می‌شود. این متد تعدادی InjetctionMember (بصورت params) دریافت می‌کند که در این مثال سه InjetctionMember به آن پاس داه شده است:

Interceptor : اطلاعاتی در مورد IInterceptor و نحوه‌ی Intercept یک شیء را نگه داری می‌کند. در اینجا از VirtualMethodInterceptor برای تزریق کد استفاده شده.

InterceptionBehavior : این کلاس Behavior مورد نظر را به کلاس تزریق می‌کند.

AddintionalInterface : کلاس target را مجبور به پیاده سازی اینترفیس دریافتی از پارامتر می‌کند. اگر کلاس behavior، متد GetRequiredInterfaces اینترفیس INotifyPropertyChanged را برمی گرداند، نیازی نیست از AddintionalInterface در پارامتر متد فوق استفاده کنید.

نکته : کلاس VirtualMethodInterceptor فقط اعضای virtual را تحت تاثیر قرار می‌دهد.

اکنون نحوه‌ی ساخت یک نمونه از کلاس Foo به شکل زیر است:

```
var foo = container.Resolve<Foo>();
(foo as INotifyPropertyChanged).PropertyChanged += FooPropertyChanged;
private void FooPropertyChanged (object sender, PropertyChangedEventArgs e)
 {
      // Do some things.......
 }
```

نکته‌ی تکمیلی

طبق مستندات MSDN ، کلاس VirtualMethodInterceptor یک کلاس جدید مشتق شده از کلاس target (در اینجا Foo) می‌سازد. بنابراین اگر کلاس‌های شما دارای Data annotation و یا در کلاس‌های Mapper یک ORM استفاده شده‌اند (مانند کلاس‌های لایه Domain)، بجای VirtualMethodInterceptor از TransparentProxyInterceptor استفاده کنید. سرعت اجرای VirtualMethodInterceptor سریعتر است ؛ اما به یاد داشته که برای استفاده از TransparentProxyInterceptor باید کلاس target از کلاس MarshalByRefObject ارث بری کند. دریافت مثال کامل این مقاله

نظرات خوانندگان

نویسنده: جلال
تاریخ: ۱۳۹۴/۰۱/۲۴ ۲۰:۲۳

این روش به همه‌ی Property Setterهای کلاس بدون در نظر گرفتن نیازهای کاربر/برنامه نویس، فراخوانی PropertyChanged رو اضافه می‌کنه. همینطور ممکنه کاربر بخواد با فراخوانی یه PropertyChanged برای یه Property، بعدش مجددا این رویداد رو برای یه Property دیگه فراخوانی کنه. به نظرم بهتره از روش‌های Attribute Base مثل این استفاده بشه.

نویسنده: برات جوادی
تاریخ: ۱۳۹۴/۰۱/۲۵ ۸:۵۴

- این Interceptor فقط کار تزریق یک Subscriber برای PropertyChanged را به عهده دارد و به سایر نیازها کاری ندارد. ضمن اینکه نیازهای کاربر/برنامه نویس اینجا کمی نامفهوم است!

- هنگام تشخیص متد set در Interceptor میتوان یک شرط دیگر گذاشت و اینکار را انجام داد.
- بسته به سناریو می‌توان از attribute هم استفاده کرد. در اینجا قصدم تزریق برای همه پراپرتی‌ها بوده، درصورتی که تزریق فقط برای برخی از انها باشد، میشه Attribute هم تعریف کرد .

عنوان: ساده سازی و بالا بردن سرعت عملیات Reflection با استفاده از Dynamic Proxy
نویسنده: وحید نصیری
تاریخ: ۱۳۹۴/۰۳/۰۹ ۱۲:۴۰
آدرس: www.dotnettips.info
گروه‌ها: Reflection, Performance, AOP

فرض کنید یک چنین کلاسی طراحی شده‌است:

```
public class NestedClass
{
    private int _field2;
    public NestedClass()
    {
        _field2 = 12;
    }
}

public class MyClass
{
    private int _field1;
    private NestedClass _nestedClass;

    public MyClass()
    {
        _field1 = 1;
        _nestedClass = new NestedClass();
    }

    private string GetData()
    {
        return "Test";
    }
}
```

می‌خواهیم از طریق Reflection مقادیر فیلدها و متدهای مخفی آن‌را بخوانیم.
حالت متداول دسترسی به فیلد خصوصی آن از طریق Reflection، یک چنین شکلی را دارد:

```
var myClass = new MyClass();

var field1Obj = myClass.GetType().GetField("_field1", BindingFlags.NonPublic | BindingFlags.Instance);
if (field1Obj != null)
{
    Console.WriteLine(Convert.ToInt32(field1Obj.GetValue(myClass)));
}
```

و یا دسترسی به مقدار خروجی متد خصوصی آن، به نحو زیر است:

```
var getDataMethod = myClass.GetType().GetMethod("GetData", BindingFlags.NonPublic |
BindingFlags.Instance);
if (getDataMethod != null)
{
    Console.WriteLine(getDataMethod.Invoke(myClass, null));
}
```

در اینجا دسترسی به مقدار فیلد مخفی NestedClass، شامل مراحل زیر است:

```
var nestedClassObj = myClass.GetType().GetField("_nestedClass", BindingFlags.NonPublic |
BindingFlags.Instance);
if (nestedClassObj != null)
{
    var nestedClassFieldValue = nestedClassObj.GetValue(myClass);
    var field2Obj = nestedClassFieldValue.GetType()
        .GetField("_field2", BindingFlags.NonPublic | BindingFlags.Instance);
    if (field2Obj != null)
    {
        Console.WriteLine(Convert.ToInt32(field2Obj.GetValue(nestedClassFieldValue)));
    }
}
```

البته این مقدار کد فقط برای دسترسی به دو سطح تو در تو بود.

چقدر خوب بود اگر می‌شد بجای این همه کد، نوشت:

```
myClass._field1
myClass._nestedClass._field2
myClass.GetData()
```

نه؟!
برای این مشکل راه حلی معرفی شده‌است به نام Dynamic Proxy که در ادامه به معرفی آن خواهیم پرداخت.

معرفی Dynamic Proxy

Dynamic Proxy یکی از مفاهیم AOP است. به این معنا که توسط آن یک محصور کننده‌ی نامرئی، اطراف یک شیء تشکیل خواهد شد. از این غشای نامرئی عموما جهت مباحث ردیابی اطلاعات، مانند پروکسی‌های Entity framework، همانجایی که تشخیص می‌دهد کدام خاصیت به روز شده‌است یا خیر، استفاده می‌شود و یا این غشای نامرئی کمک می‌کند که در حین دسترسی به خاصیت یا متدی، بتوان منطق خاصی را در این بین تزریق کرد. برای مثال فرآیند تکراری logging سیستم را به این غشای نامرئی منتقل کرد و به این ترتیب می‌توان به کدهای تمیزتری رسید.
یکی دیگر از کاربردهای این محصور کننده یا غشای نامرئی، ساده سازی مباحث Reflection است که نمونه‌ای از آن در پروژه‌ی EntityFramework.Extended بکار رفته‌است.
در اینجا، کار با محصور سازی نمونه‌ای از کلاس مورد نظر با Dynamic Proxy شروع می‌شود. سپس کل عملیات Reflection فوق در همین چند سطر ذیل به نحوی کاملا عادی و طبیعی قابل انجام است:

```
// Accessing a private field
dynamic myClassProxy = new DynamicProxy(myClass);
dynamic field1 = myClassProxy._field1;
Console.WriteLine((int)field1);

// Accessing a nested private field
dynamic field2 = myClassProxy._nestedClass._field2;
Console.WriteLine((int)field2);

// Accessing a private method
dynamic data = myClassProxy.GetData();
Console.WriteLine((string)data);
```

خروجی Dynamic Proxy از نوع dynamic دات نت 4 است. پس از آن می‌توان در اینجا هر نوع خاصیت یا متد دلخواهی را به شکل dynamic تعریف کرد و سپس به مقادیر آن‌ها دسترسی داشت.

بنابراین با استفاده از Dynamic Proxy فوق می‌توان به دو مهم دست یافت:
1) ساده سازی و زیبا سازی کدهای کار با Reflection
2) استفاده‌ی ضمنی از مباحث Fast Reflection . در کتابخانه‌ی Dynamic Proxy معرفی شده، دسترسی به خواص و متدها، توسط کدهای IL بهینه سازی شده‌است و در دفعات آتی کار با آن‌ها، دیگر شاهد سربار بالای Reflection نخواهیم بود.

کدهای کامل این مثال را از اینجا می‌توانید دریافت کنید:
DynamicProxyTests.zip