动态 API 接口插件实现文档 V1.0

一、设计目的

当我们的应用程序架构是以服务的形式进行分布式访问的时候，我们不可避免的会涉及到一个接口数据交互。我们的数据流向一般为（当然这里仅仅是为了解释动态 API 接口产生的原因，而不会纠结具体的分层架构）:仓储层-->领域业务逻辑层-->应用服务层-->API 接口层-->客户端调用层(展现层)；当我们公开接口的时候，API 接口层会调用应用服务层或者领域业务逻辑层的数据。有的时候，我们的 API 接口层仅仅做了一些简单的数据转发功能。从而造成 API 接口层里的某些接口变成了非常薄的一层，API 接口方法仅仅是做了针对应用服务层或者领域业务逻辑层的简单调用。这样对于开发者来说会造成不必要的重复工作。假若我们能够通过某种方法，在运行时将应用服务层或者领域业务逻辑层里的方法自动映射层对外公开的 API 接口，那么我们将会大大的减少工作量，而且也会减少代码的维护量（当然正常的流程系统框架还是走正常的 API 接口层，只是框架帮我们自动完成了接口映射工作而已）。基于这样的出发点，我们再原来接口框架的基础上扩展出了动态 API 接口插件。我们只要在 API 接口层（HOST）项目引用此动态接口插件程序集。即可实现将应用服务层或者领域业务逻辑层的方法映射成接口，供展现层调用。

项目 SVN 地址：

http://192.168.3.155:8443/svn/frxs_server/ERP/trunk/Frxs.Erp.Project/Frxs.Erp.ServiceCenter/Frxs.ServiceCenter.Core.DynamicApi

演示项目 SVN 地址：

http://192.168.3.155:8443/svn/frxs_server/ERP/trunk/Frxs.Erp.Project/Frxs.Erp.ServiceCenter/Frxs.ServiceCenter.Api.Host.V20

二、架构实现

我们知道，API 接口的所有操作抽象出来都总结为：上送参数(RequestDto)，下送数据(ResponseDto)，数据加工处理（方法处理）,如果我们将应用服务层或者领域业务逻辑层的方法参数映射成接口上送参数，将返回值映射成下送数据，API 数据加工处理方法 Execute 自动调用对应的方法。那么我们就可以完美的解决映射工作。然后我们通过搜索所有合法的应用服务层方法来自动生成 API 接口即可。主要的思想以及流程即上面所说，即大致流程为：

系统框架自动寻找我们约定的需要映射的类-->查找里面合法的待映射成API 接口的方法 -->将方法入参映射成接口的 RequestDto，将方法返回值映射成接口的 ResponseDto，将方法自动放到接口的 Execute 方法里调用-->通过上面的步骤创建一个 API 接口源码-->使用 C#动态编译，编译成动态程序集-->重新调用框架的接口

注册流程，将动态程序集里定义的接口注册到当前系统的 IOC 容器里。

2.1 如何告知系统哪些类需要动态映射 API 接口 IDynamicApiServics 接口

上面的步骤即为核心流程，下面我们来看下代码是如何实现，首先我们要系统框架知道哪些应用服务层或领域业务逻辑层类是需要映射 API 接口方法的类，我们定义了一个 IDynamicApiService 接口，此接口是个空接口，无任何方法属性，仅仅用于标识我们的哪些类需要自动映射成 API 接口。

```
public class TestService : IDynamicApiService
//需要动态映射的服务类，需要继承此接口 IDynamicApiService，此为约定。
//当然，我们可以不继承的方式来实现，但是那样会加重动态接口查询的速度，因此我们采取约定的方式来架构
{
}
```

2.2 动态 API 接口查找器(IDynamicApiSelector,DynamicApiAttribute,NotDynamicApiAttribute)

知道了如何定义一个需要动态映射成 API 接口的类后，那么怎么去搜索合法的 API 接口方法，什么样的方法定义才是合法的 API 接口方法？我们先来看一下动态接口 API 搜索器 IDynamicApiSelector 的接口定义：

```
/* ********************************************************
 * FRXS(ISC) zhangliang4629@163.com 9/7/2016 9:01:26 AM
 * ********************************************************/
using System.Collections.Generic;
using System;
using System.Reflection;

namespace Frxs.ServiceCenter.Api.Core.DynamicApi
{
    /// <summary>
    /// 动态接口查找器，动态查找所有实现了 IDynamicApiService 接口的类
    /// </summary>
    public interface IDynamicApiSelector
    {
        /// <summary>
        /// 获取所有合法动态 API 方法描述对象，一个合法的动态接口方法需要满足
        ///
        /// 1.方法所属的类，必须实现 IDynamicApiService 接口（空接口）
        ///
        /// 2.需要映射成动态 API 的方法，需要定义 DynamicApiAttribute 特性，如果
        ///   为了方便，我们可以将 DynamicApiAttribute 特性定义在方法所属的类上
        ///   这样类里的所有方法都会继承此特性定义
        ///
        /// 3.方法入参必须小于等于 1 个参数，如果参数个数为 1，那么参数必须继承 RequestDtoBase 抽象类
        ///   如果参数为 0 个，系统框架会自动将参数映射成 NullRequestDto 类
        ///
        /// 4.如果方法是一个合法的动态 API，但是我们不想让它对外公开成一个动态 API,我们只要
        ///   在方法上定义特性 NotDynamicApiAttribute 即可。
        /// </summary>
        /// <param name="methodFilter">合法的动态 API 方法筛选器(此过滤器是在满足上面动态 API 搜索的情况下，二次搜索过滤器)</param>
        /// <returns>所有合法的待映射成动态 API 接口的方法描述对象集合，找不到则返回空集合</returns>
        IEnumerable<DynamicApiDescriptor> GetDynamicApiDescriptors(Func<MethodInfo, bool> methodFilter = null);
    }
}
```

具体搜索合法的方法上面接口定义注释写的比较清楚了。下面在来看下具体的

DefaultDynamicApiSelector 实现：

```
/* ************************************************************
 * FRXS(ISC) zhangliang4629@163.com 9/7/2016 11:18:50 AM
 * ************************************************************/
using System;
using System.Collections.Generic;
using System.Linq;
using System.Reflection;

namespace Frxs.ServiceCenter.Api.Core.DynamicApi
{
    /// <summary>
    /// 默认的动态接口查找器，为了扩展，我们将方法定义成了可重写的。
    /// 便于根据不同的项目，实现不同的动态 API 接口查找方式
    /// </summary>
    public class DefaultDynamicApiSelector : IDynamicApiSelector
    {
        /// <summary>
        ///
        /// </summary>
        private readonly ITypeFinder _typeFinder;

        /// <summary>
        ///
        /// </summary>
        /// <param name="typeFinder">类型查找器</param>
        public DefaultDynamicApiSelector(ITypeFinder typeFinder)
        {
            typeFinder.CheckNullThrowArgumentNullException("typeFinder");
            this._typeFinder = typeFinder;
        }

        /// <summary>
        /// 获取接口请求上送参数类型
        /// </summary>
        /// <param name="methodInfo">待生成接口的方法信息</param>
        /// <returns>Type，方法的参数类型</returns>
        private Type GetRequestDtoType(MethodInfo methodInfo)
        {
            //获取接口所有参数
            var parameters = methodInfo.GetParameters();

            //方法不带参数，默认给一个系统框架自带的 NullRequestDto 参数
            if (!parameters.Any())
            {
                return typeof(NullRequestDto);
            }

            //含有参数，获取第一个参数类型
            return parameters.FirstOrDefault().ParameterType;
        }

        /// <summary>
        /// 获取接口返回对象数据类型,内部做了处理，如果无返回值，则默认输出 NullResponseDto
        /// </summary>
        /// <param name="methodInfo">待生成接口的方法信息</param>
        /// <returns>Type 方法的返回值类型</returns>
        private Type GetResponseDtoType(MethodInfo methodInfo)
        {
            //无返回值
            //if (typeof(void).Equals(methodInfo.ReturnType))
            //{
            //    return typeof(NullResponseDto);
            //}

            var returnType = methodInfo.ReturnType;
```

```
    //泛型集合
    //if (returnType.IsGenericType && returnType.GetGenericTypeDefinition() == typeof(IEnumerable<>))
    //{
    //    return typeof(IList<>).MakeGenericType(returnType.GetGenericArguments()[0]);
    //}

    //泛型字典

    //有返回值
    return returnType;
}

/// <summary>
/// 公开的，并且参数只有一个，并且入参继承 RequestDtoBase 基类的，我们认为是一个需要公开的接口
/// </summary>
/// <param name="methodInfo">待生成接口的方法信息</param>
/// <returns>true/false</returns>
protected virtual bool IsDynamicApi(MethodInfo methodInfo)
{
    //IsSpecialName：是否是属性 ，属性会自动生成：get_属性名称这样的方法，因此需要排除掉
    if (methodInfo.IsSpecialName)
    {
        return false;
    }

    //指定不为动态 API 接口
    if (methodInfo.IsDefined(typeof(NotDynamicApiAttribute), false))
    {
        return false;
    }

    //类是否定义了动态接口特性，如果定义了，那么下面的所有实例方法都变成了动态接口
    if (!methodInfo.DeclaringType.IsDefined(typeof(DynamicApiAttribute), false)
        && !methodInfo.IsDefined(typeof(DynamicApiAttribute), false))
    {
        return false;
    }

    //不能是泛型方法(在业务逻辑层，定义成泛型方法的做法不是很好，因为业务领域了一般都是针对领域分析的结果
    //一般不会出现通用的业务逻辑，就算是有，那么肯定可以放在其他层，而不应该放在领域业务层里面)
    if (methodInfo.IsGenericMethod)
    {
        return false;
    }

    //获取接口所有参数
    var parameters = methodInfo.GetParameters();

    //如果接口参数大于 1，我们不认为是动态接口
    if (parameters.Length > 1)
    {
        return false;
    }

    //如果参数数量==1，但是参数没有继承 RequestDtoBase 我们也不认为是合法的动态接口
    if (parameters.Length == 1 && !typeof(RequestDtoBase).IsAssignableFrom(parameters[0].ParameterType))
    {
        return false;
    }

    //方法是合法的动态接口
    return true;
}

/// <summary>
/// 获取所有带生成的合法的服务层方法接口
```

```
        /// </summary>
        /// <param name="methodFilter">接口方法过滤器</param>
        /// <returns>获取当前应用程序域里的所有合法的待生成动态 API 接口的描述对象集合</returns>
        public virtual IEnumerable<DynamicApiDescriptor> GetDynamicApiDescriptors(Func<MethodInfo, bool> methodFilter = null)
        {
            //用于临时保存接口描述对象集合
            var dynamicApiDescriptors = new List<DynamicApiDescriptor>();

            //查找所有制定定义了动态 API 接口的 services 类
            var dynamicApiServiceTypes = this._typeFinder.FindClassesOfType<IDynamicApiService>();

            //获取所有的符合接口的方法
            foreach (var dynamicApiServiceType in dynamicApiServiceTypes)
            {
                //所有的方法集合(必须为公开且为实例方法)
                var methods = dynamicApiServiceType.GetMethods(BindingFlags.Public | BindingFlags.Instance)
                    .Where(m => this.IsDynamicApi(m));

                //自定义了过滤器，但是也必须在框架定义的基础之上，再次进行过滤
                if (!methodFilter.IsNull())
                {
                    methods = methods.Where(methodFilter);
                }

                //获取合法的方法
                foreach (var method in methods)
                {
                    //默认使用类名+方法名称作为接口名称
                    var actionName = "{0}.{1}".With(dynamicApiServiceType.Name, method.Name);

                    //方法定义了接口名称
                    var actionNameAttributeType = typeof(ActionNameAttribute);
                    if (method.IsDefined(actionNameAttributeType, false))
                    {
                        actionName = method.GetCustomAttributes(actionNameAttributeType, false)
                            .Cast<ActionNameAttribute>().FirstOrDefault().Name;
                    }

                    //构造动态 API 描述对象
                    var dynamicApiDescriptor = new DynamicApiDescriptor()
                    {
                        DeclaringType = dynamicApiServiceType,
                        RequestDtoType = this.GetRequestDtoType(method),
                        ResponseDtoType = this.GetResponseDtoType(method),
                        MethodInfo = method,
                        ActionName = actionName
                    };

                    //将找到的可以映射动态 api 接口的方法描述对象保存到集合
                    dynamicApiDescriptors.Add(dynamicApiDescriptor);
                }
            }

            //返回所有找到的动态 API 描述对象
            return dynamicApiDescriptors;
        }
    }
}
```

2.3 动态 API 接口源码生成器(CollectionAction)

通过接口查找器找到了所有合法的需要映射成动态 API 接口的方法后，我们开看一下，如何将搜索到的合法待映射成动态 API 接口的方法输出合法的接口类：

```
/* ****************************************************
 * FRXS(ISC) zhangliang4629@163.com 9/7/2016 2:22:24 PM
```

```
 * ********************************************************/
using Autofac;
using System;
using System.CodeDom.Compiler;
using System.Collections.Generic;
using System.IO;
using System.Linq;

namespace Frxs.ServiceCenter.Api.Core.DynamicApi
{
    /// <summary>
    /// 检索出待自动生成接口的业务代码
    /// </summary>
    [DisablePackageSdk, DisableDataSignatureTransmission, AllowAnonymous]
    internal class CollectionAction : ActionBase<NullRequestDto, IEnumerable<DynamicApiDescriptor>>
    {
        /// <summary>
        ///
        /// </summary>
        private readonly IDynamicApiSelector _dynamicApiSelector;
        private readonly IMediaTypeFormatterFactory _mediaTypeFormatterFactory;
        private readonly IActionSelector _actionSelector;
        private readonly ITypeFinder _typeFinder;
        private readonly DynamicApiConfig _dynamicApiConfig;
        private readonly IMachineNameProvider _machineNameProvider;

        /// <summary>
        ///
        /// </summary>
        /// <param name="dynamicApiSelector">动态 API 方法查找器</param>
        /// <param name="mediaTypeFormatterFactory">内容输出格式化器创建器</param>
        /// <param name="actionSelector">API 接口查找器</param>
        /// <param name="typeFinder">类型查找器</param>
        /// <param name="dynamicApiConfig">动态 API 接口配置信息</param>
        /// <param name="machineNameProvider">当前实例信息提供器</param>
        public CollectionAction(
            IDynamicApiSelector dynamicApiSelector,
            IMediaTypeFormatterFactory mediaTypeFormatterFactory,
            IActionSelector actionSelector,
            ITypeFinder typeFinder,
            DynamicApiConfig dynamicApiConfig,
            IMachineNameProvider machineNameProvider)
        {
            this._dynamicApiSelector = dynamicApiSelector;
            this._mediaTypeFormatterFactory = mediaTypeFormatterFactory;
            this._actionSelector = actionSelector;
            this._typeFinder = typeFinder;
            this._dynamicApiConfig = dynamicApiConfig;
            this._machineNameProvider = machineNameProvider;
        }

        /// <summary>
        /// 返回所有业务层指定需要映射的接口信息
        /// </summary>
        /// <returns></returns>
        public override ActionResult<IEnumerable<DynamicApiDescriptor>> Execute()
        {
            return this.SuccessActionResult(this._dynamicApiSelector.GetDynamicApiDescriptors(m => true));
        }

        /// <summary>
        /// 自动将业务层指定的方法输出为接口
        /// </summary>
        /// <param name="actionExecutedContext">直接执行完毕后触发的方法执行上下文</param>
        protected override void OnActionExecuted(ActionExecutedContext actionExecutedContext)
        {
            //请求的内部接口
```

```
            var actionName = "Frxs.DynamicApi.Generator";

            //临时保存源代码 Key:源代码文件名称，Value:源代码
            IList<KeyValuePair<string, string>> sourceClassStrings = new List<KeyValuePair<string, string>>();

            //循环当前注册的所有模型
            foreach (var item in (IEnumerable<DynamicApiDescriptor>)actionExecutedContext.Result.Data)
            {
                //原始请求参数
                var requestParams = new RequestParams()
                {
                    ActionName = actionName,
                    Data = "{}",
                    Format = "View",
                };

                //构造请求上下文
                var requestContext = new RequestContext(
                            httpContext: this.RequestContext.HttpContext,
                            systemOptions: SystemOptionsManager.Current,
                            requestDto: null,
                            actionDescriptor: this.RequestContext.ActionDescriptor,
                            rawRequestParams: requestParams,
                            decryptedRequestParams: requestParams.MapTo<RequestParams>());

                //格式化器，默认使用 view
                var mediaTypeFormatter = this._mediaTypeFormatterFactory.Create(ResponseFormat.VIEW);

                //接口源代码
                var serializedActionResultToString = mediaTypeFormatter.SerializedActionResultToString(requestContext, new
ActionResult()
                {
                    Data = item,
                    Flag = ActionResultFlag.SUCCESS,
                    Info = "OK"
                });

                //接口保存的文件名称
                string actionFileName = "{0}.Action.cs".With(item.ActionName);

                //添加到集合
                sourceClassStrings.Add(new KeyValuePair<string, string>(actionFileName, serializedActionResultToString));
            }

            //开发模式，保存到本地磁盘
            if ((this._dynamicApiConfig.WorkMode & WorkMode.Develop) == WorkMode.Develop)
            {
                //创建文件夹
                this.CreateDynamicApiDirectory();

                //保存原代码
                this.SaveSourceToDisk(sourceClassStrings);
            }

            //工作默认为自动映射
            if ((this._dynamicApiConfig.WorkMode & WorkMode.Dynamic) == WorkMode.Dynamic)
            {
                this.CompileAssemblyFromSource(sourceClassStrings.ToArray());
            }
        }

        /// <summary>
        /// 创建文件夹，如果不存在就创建(当一个站点运行在一个服务器，但是配置了多个 WEB-frame 的时候，可能会出现错误)
        /// </summary>
        private void CreateDynamicApiDirectory()
        {
            //生成存放 DLL 的目录
```

```
        var dllSaveDirectory = this.RequestContext.HttpContext.Server
                                .MapPath(this._dynamicApiConfig.DynamicDirectory);
        if (!Directory.Exists(dllSaveDirectory))
        {
            Directory.CreateDirectory(dllSaveDirectory);
        }
        //存在文件夹，就先清空文件
        else
        {
            var files = Directory.GetFiles(dllSaveDirectory, "*");
            foreach (var file in files)
            {
                File.Delete(file);
            }
        }
    }

    /// <summary>
    /// 保存源代码到本地磁盘
    /// </summary>
    /// <param name="sources">文件名-源代码</param>
    private void SaveSourceToDisk(IEnumerable<KeyValuePair<string, string>> sources)
    {
        //循环保存源代码到本地磁盘
        foreach (var source in sources)
        {
            using (var streamWriter = new StreamWriter(this.RequestContext.HttpContext.Server
                .MapPath("{0}/{1}".With(this._dynamicApiConfig.DynamicDirectory, source.Key))))
            {
                streamWriter.WriteLine(source.Value);
            }
        }
    }

    /// <summary>
    /// 编译源代码到 DLL
    /// </summary>
    /// <param name="sources">待编译的源代码</param>
    private void CompileAssemblyFromSource(IEnumerable<KeyValuePair<string, string>> sources)
    {
        //动态 API 配置文件
        var httpServerUtility = this.RequestContext.HttpContext.Server;

        //生成存放 DLL 的目录
        var dllSaveDirectory = httpServerUtility.MapPath(this._dynamicApiConfig.DynamicDirectory);

        //dll 名
        string dllname = "{0}.dll".With(this._dynamicApiConfig.ActionNameSpace);

        //bin 目录
        string binDirectoryPath = httpServerUtility.MapPath("~/bin");

        //输出 dll 保存地址
        string assemblySavePath = Path.Combine(dllSaveDirectory, dllname);

        //保存 DLL 注释文件 XML 路径
        string docSavePath = Path.Combine(dllSaveDirectory, "{0}.XML".With(this._dynamicApiConfig.ActionNameSpace));

        //获取代码编译器
        CodeDomProvider provider = CodeDomProvider.CreateProvider("C#");
        CompilerParameters compilerparams = new CompilerParameters();
        //指定编译选项，是类库、并且生成 XML 注释文档
        compilerparams.CompilerOptions = "/target:library /optimize";
        //compilerparams.CompilerOptions = "/target:library /optimize /doc:{0}".With(docSavePath);
        compilerparams.GenerateInMemory = true;
        compilerparams.GenerateExecutable = false;
        //compilerparams.IncludeDebugInformation = true;
```

```
compilerparams.IncludeDebugInformation = false;
compilerparams.TreatWarningsAsErrors = false;
//compilerparams.OutputAssembly = assemblySavePath;

//加载默认系统 dll 文件
string[] systemDlls = new string[] { "System.dll",
                                     "System.Core.dll",
                                     "System.Xml.Linq.dll",
                                     "System.Configuration.dll",
                                     "System.Data.DataSetExtensions.dll",
                                     "Microsoft.CSharp.dll",
                                     "System.Data.dll",
                                     "System.Xml.dll",
                                     "System.Web.dll" };

systemDlls.ToList().ForEach(assembly =>
{
    compilerparams.ReferencedAssemblies.Add(assembly);
});

//获取 bin 目录下面的所有 dll 文件 排除当前生成 dll
Directory.GetFiles(binDirectoryPath, "*.dll", SearchOption.TopDirectoryOnly)
         .Select(path => Path.GetFileName(path))
         .Where(t => t != dllname).ToList()
         .ForEach(assembly =>
         {
             compilerparams.ReferencedAssemblies.Add(Path.Combine(binDirectoryPath, assembly));
         });

//编译源代码
var compilerResults = provider.CompileAssemblyFromSource(compilerparams, sources.Select(o => o.Value).ToArray());

//编译源码出现错误
if (compilerResults.Errors.HasErrors)
{
    IList<string> errors = new List<string>();
    foreach (CompilerError error in compilerResults.Errors)
    {
        errors.Add(String.Format("Error on line {0}: {1}", error.Line, error.ErrorText));
    }

    //记录下错误日志
    this.Logger.Error(errors.JoinToString(Environment.NewLine));

    //出错直接返回
    return;
}

// 获取动态的程序集
// var dynamicAssembly = Assembly.LoadFile(assemblySavePath);

// 加载程序集到当前域
// AppDomain.CurrentDomain.Load(dynamicAssembly.GetName());

//重新刷新 API 接口缓存器
this._actionSelector.Reset();

//由于执行的先后原因，我们再次刷新下所有 API 接口 IOC 注册
var containerBuilder = new ContainerBuilder();
//框架自动搜索程序集，注册所有实现了 IAction 接口的类；
this._typeFinder.FindClassesOfType(typeof(IAction), new[] { compilerResults.CompiledAssembly })
                .Where(type => type.IsAssignableToActionBase()).ToList().ForEach(type =>
                {
                    containerBuilder.RegisterType(type).PropertiesAutowired().InstancePerLifetimeScope();
                });
containerBuilder.Update(ServicesContainer.Current.Container);
```

```
            //记录下日志
            this.Logger.Information("动态 API 生成成功，程序集：{0}，当前 HOST 工作进程：{1}"
                    .With(compilerResults.CompiledAssembly.FullName, this._machineNameProvider.GetMachineName()));
        }
    }
}
```

2.4 系统启动自动生成动态接口(StartUp)

为了让系统在启动的时候，自动映射合法的动态 API 接口，我们在插件里定义了一个实现 IStartUp 接口的启动类。这样系统在启动的时候，会自动调用 CollectionAction 来生成 API 接口

```
/* ********************************************************
 * FRXS(ISC) zhangliang4629@163.com 9/7/2016 11:25:19 AM
 * ********************************************************/
using System;
using System.Linq;
using System.Linq.Expressions;
using System.Reflection;
using System.Web;
using Frxs.ServiceCenter.Api.Core.ViewEngine;

namespace Frxs.ServiceCenter.Api.Core.DynamicApi
{
    /// <summary>
    /// 启动的时候先预热下，将所有合法的待生成的 API 的方法查找下
    /// </summary>
    public class StartUp : IStartUp
    {
        /// <summary>
        ///
        /// </summary>
        private readonly IDynamicApiSelector _dynamicApiSelector;
        private readonly IMediaTypeFormatterFactory _mediaTypeFormatterFactory;
        private readonly ILogger _logger;
        private readonly IActionSelector _actionSelector;
        private readonly ITypeFinder _typeFinder;
        private readonly IMachineNameProvider _machineNameProvider;
        private readonly IResourceFinderManager _resourceFinderManager;
        private readonly DynamicApiConfig _dynamicApiConfig;

        /// <summary>
        ///
        /// </summary>
        /// <param name="dynamicApiSelector">动态接口方法查找器</param>
        /// <param name="mediaTypeFormatterFactory">内容输出格式化创建器</param>
        /// <param name="actionSelector">API 接口查找器</param>
        /// <param name="typeFinder">类型查找器</param>
        /// <param name="logger">日志记录器</param>
        /// <param name="machineNameProvider">当前实例信息提供器</param>
        /// <param name="resourceFinderManager">资源查找器</param>
        /// <param name="viewEngineManager">视图引擎管理器</param>
        /// <param name="dynamicApiConfig">动态 API 接口配置信息</param>
        public StartUp(
            IDynamicApiSelector dynamicApiSelector,
            IMediaTypeFormatterFactory mediaTypeFormatterFactory,
            IActionSelector actionSelector,
            ITypeFinder typeFinder,
            ILogger<StartUp> logger,
            IMachineNameProvider machineNameProvider,
            IResourceFinderManager resourceFinderManager,
            IViewEngineManager viewEngineManager,
            DynamicApiConfig dynamicApiConfig)
        {
            this._dynamicApiSelector = dynamicApiSelector;
            this._mediaTypeFormatterFactory = mediaTypeFormatterFactory;
            this._actionSelector = actionSelector;
```

```
        this._typeFinder = typeFinder;
        this._logger = logger ?? GenericNullLogger<StartUp>.Instance;
        this._machineNameProvider = machineNameProvider;
        this._resourceFinderManager = resourceFinderManager;
        this._dynamicApiConfig = dynamicApiConfig;
    }

    /// <summary>
    /// 初始化(整个什么周期只执行一次，在 IOC 容器注册完成后进行)
    /// </summary>
    public void Init()
    {
        try
        {
            //原始请求参数
            var requestParams = new RequestParams()
            {
                ActionName = "Frxs.DynamicApi.Collection",
                Data = "{}",
                Format = "View"
            };

            //接口描述对象
            var actionDescriptor = new ReflectedActionDescriptor(typeof(CollectionAction)).GetActionDescriptor();

            //构造请求上下文
            var requestContext = new RequestContext(
                       httpContext: new HttpContextWrapper(HttpContext.Current),
                       systemOptions: SystemOptionsManager.Current,
                       requestDto: new NullRequestDto(),
                       actionDescriptor: actionDescriptor,
                       rawRequestParams: requestParams,
                       decryptedRequestParams: requestParams.MapTo<RequestParams>());

            //创建出生成动态 API 接口实例
            var collectionAction = new CollectionAction(this._dynamicApiSelector,
                                                        this._mediaTypeFormatterFactory,
                                                        this._actionSelector,
                                                        this._typeFinder,
                                                        this._dynamicApiConfig,
                                                        this._machineNameProvider);
            collectionAction.ActionDescriptor = actionDescriptor;
            collectionAction.RequestContext = requestContext;
            collectionAction.Logger = this._logger;
            ((IAction)collectionAction).RequestDto = requestContext.RequestDto;

            //执行返回结果
            var result = collectionAction.Execute();

            //执行结果出来方法上下文构造
            var actionExecutedContext = new ActionExecutedContext(collectionAction, new ActionResult()
            {
                Flag = result.Flag,
                Info = result.Info,
                Data = result.Data
            });

            //获取 OnActionExecuted 方法，业务系统框架里定义成了收保护的方法，因此需要使用反射的方式执行
            var methodInfo = collectionAction.GetType()
                .GetMethods(BindingFlags.NonPublic | BindingFlags.Public | BindingFlags.Instance)
                .FirstOrDefault(x => !x.IsSpecialName && x.Name == "OnActionExecuted");

            //执行方法
            // m.Invoke(collectionAction, new object[] { actionExecutedContext });
            this.Execute(methodInfo)(collectionAction, actionExecutedContext);
        }
        //扩展，不属于核心流程，所以此流程出错，对整个框架来说不构成影响，因此我们只记录下错误日志
```

```
            catch (Exception exc)
            {
                this._logger.Error(exc);
            }
        }

        /// <summary>
        /// 这里我们使用表达式树的方式来执行接口方法，来加快执行速度
        /// 生成表达式树:(action, actionExecutedContext) => action.OnActionExecuted(actionExecutedContext)
        /// </summary>
        /// <param name="methodInfo">待执行的方法信息</param>
        /// <returns>动态创建的表达式树对应的委托</returns>
        private Action<CollectionAction, ActionExecutedContext> Execute(MethodInfo methodInfo)
        {
            var target = Expression.Parameter(typeof(CollectionAction), "action");
            var argParam = Expression.Parameter(typeof(ActionExecutedContext), "actionExecutedContext");
            var callMethodInvoker = Expression.Call(target, methodInfo, argParam);
            return Expression.Lambda<Action<CollectionAction, ActionExecutedContext>>(callMethodInvoker, target,
argParam).Compile();
        }
    }
}
```

以上即动态 API 接口核心的类已经实现流程。下面我们来看下怎么使用

三、使用方法

使用方法很简单，我们只要在我们的接口 HOST 站点，引用接口插件程序集，然后将定于有需要动态映射 API 接口的服务类也引用到 HOST 站点即可，系统框架会自动完成所有动作。我们下面来看一个演示服务类：

```
/* ********************************************************
 * FRXS(ISC) zhangliang4629@163.com 9/7/2016 3:56:55 PM
 * ********************************************************/
using Frxs.ServiceCenter.Api.Core;
using Frxs.ServiceCenter.Api.Core.Data;
using Frxs.ServiceCenter.Api.Core.DynamicApi;
using System;
using System.Collections.Generic;
using System.Linq;

namespace Frxs.ServiceCenter.Api.Host
{
    /// <summary>
    /// 动态接口演示服务类(Services 类)
    /// 针对需要动态生成的服务类，注册类型到 IOC 容器的时候，需要将自身类型也注册进去，不能只注册对应的接口
    /// </summary>
    [DynamicApi] //如果为了省事，类上定义的，所有符合动态 API 的方法都会被映射
    public class TestService : IDynamicApiService
    //需要动态映射的服务类，需要继承此接口 IDynamicApiService，此为约定。
    //当然，我们可以不继承的方式来实现，但是那样会加重动态接口查询的速度，因此我们采取约定的方式来架构
    {
        /// <summary>
        ///
        /// </summary>
        private readonly ITypeFinder _typeFinder;
        private readonly IRepository<Data.Domain.Shelf> _shelfRepository;
        private readonly IRepository<Data.Domain.Warehouse> _warehouseRepository;
        private readonly IUnitOfWork _unitOfWork;
        private readonly IDbContext _dbContext;
        private readonly IDataProvider _dataProvider;

        /// <summary>
        /// 日志记录器(只要定义了此属性，系统框架会自动创建当前实现的日志记录器)
        /// </summary>
        public ILogger Logger { get; set; }
```

```
/// <summary>
/// 缓存记录器，任意类，只要定义了此属性，系统框架会自动处理，
/// 当然我们也可以在强制需要使用缓存的服务类采取构造函数注入的方式
/// </summary>
public ICacheManager CacheManager { get; set; }

/// <summary>
/// 显示构造函数注入服务类
/// </summary>
/// <param name="typeFinder"></param>
/// <param name="shelfRepository"></param>
/// <param name="warehouseRepository"></param>
/// <param name="unitOfWork"></param>
/// <param name="dbContext"></param>
/// <param name="dataProvider"></param>
public TestService(
    ITypeFinder typeFinder,
    IRepository<Data.Domain.Shelf> shelfRepository,
    IRepository<Data.Domain.Warehouse> warehouseRepository,
    IUnitOfWork unitOfWork,
    IDbContext dbContext,
    IDataProvider dataProvider)
{
    this._typeFinder = typeFinder;
    this._shelfRepository = shelfRepository;
    this._warehouseRepository = warehouseRepository;
    this._unitOfWork = unitOfWork;
    this._dbContext = dbContext;
    this._dataProvider = dataProvider;
    this.Logger = GenericNullLogger<TestService>.Instance;
    this.CacheManager = NullCacheManager.Instance;
}

/// <summary>
/// 符合动态 API 接口，但是定义了 NotDynamicApi ，排除映射成动态 API
/// </summary>
/// <param name="request"></param>
/// <returns></returns>
//[NotDynamicApi]
public IDictionary<string, string> Method0(NullRequestDto request)
{
    return new Dictionary<string, string>() { };
}

/// <summary>
/// 非动态 API 服务层方法
/// </summary>
/// <param name="Id"></param>
/// <returns></returns>
public Data.Domain.Shelf EntityQuery(NullRequestDto Id)
{
    return this._shelfRepository.Table.FirstOrDefault();
}

/// <summary>
/// 演示原始 SQL 语句，以及无参动态接口定义
/// </summary>
[DynamicApi]
public object DynamicSqlQuery()
{
    return this._dbContext.DynamicSqlQuery("Select * from  Shelf where id>@p0", 10);
}

/// <summary>
/// 演示使用 SQL 直接查询
/// </summary>
```

```
        /// <returns></returns>
        public IEnumerable<Data.Domain.Warehouse> SqlQuery()
        {
            return this._dbContext.SqlQuery<Data.Domain.Warehouse>("select top 1000 * from Warehouses order by Id desc").OrderBy(o
=> o.Id).ToList();
        }

        /// <summary>
        /// 演示聚合根，以及单条数据插入
        /// </summary>
        /// <returns></returns>
        [DynamicApi]
        public object Insert()
        {
            //聚合根的方式，插入一个聚合根，相关对象会自动插入
            var warehouse = new Data.Domain.Warehouse()
            {
                Id = Guid.NewGuid().ToString("N"),
                WHName = "演示" + Guid.NewGuid().ToString("N"),
                Shelfs = new List<Data.Domain.Shelf> { new Data.Domain.Shelf()
            {
                ModifyTime = DateTime.Now,
                ModifyUserName = "system",
                ModifyUserId = "0",
                ShelfAreaID = 100,
                ShelfAreaName = "A",
                ShelfCode = "A2",
                ShelfID = 1,
                ShelfType = "WH",
                Status = "S",
                StatusStr = "SSS",
                WName = "星沙完成",
                WID = "A200",
                IsDeleted = false,
                CreatedTime = DateTime.Now,
                CreatedUserId = "0",
                CreatedUserName = "system"
            } }
            };

            this._warehouseRepository.Insert(warehouse);

            //提交工作单元（事务支持）
            this._unitOfWork.Commit();

            //记录下日志
            this.Logger.Information(this.Logger.GetType().FullName);

            //返回编号
            return warehouse.Shelfs.FirstOrDefault().Id;
        }

        /// <summary>
        /// 演示更新操作
        /// </summary>
        public void Update()
        {
            var shelf = this._shelfRepository.Table.FirstOrDefault(o => o.Id == 200);
            if (shelf != null)
            {
                //修改字段
                shelf.ShelfCode = "A0001" + Guid.NewGuid().ToString();

                //更新
                this._shelfRepository.Update(shelf);

                //提交事务到数据库
```

```
            this._unitOfWork.Commit();
        }
    }

    /// <summary>
    /// 演示删除操作
    /// </summary>
    public void Delete()
    {
        //查询出待删除的数据
        var shelfs = this._shelfRepository.Table.Where(o => o.Id < 200);

        //循环移除
        foreach (var item in shelfs)
        {
            this._shelfRepository.Delete(item);
        }

        //提交事务到数据库
        this._unitOfWork.Commit();
    }

    /// <summary>
    /// 演示调用存储过程 1
    /// </summary>
    /// <returns></returns>
    public object ExecuteStoredProcedure1()
    {
        var idParameter = this._dataProvider.GetParameter();
        idParameter.ParameterName = "Id";
        idParameter.Value = 1;

        //专门的存储过程执行方法，比下面的调用方式简单点，在进行存储过程使用的时候
        //请先判断下 this._dataProvider.StoredProceduredSupported 是否支持存储过程
        //因为有些数据库不执行存储过程
        var q = this._dbContext.ExecuteStoredProcedure<Data.Domain.Shelf>("SP", idParameter);

        return q;

    }

    /// <summary>
    /// 演示调用存储过程 2
    /// </summary>
    /// <returns></returns>
    public object ExecuteStoredProcedure2()
    {
        var idParameter = this._dataProvider.GetParameter();
        idParameter.ParameterName = "Id";
        idParameter.Value = 1;
        //idParameter.Direction = System.Data.ParameterDirection.Output;

        // 执行存储过程,执行语句里需要自己定义存储参数
        var q = this._dbContext.SqlQuery<Data.Domain.Shelf>("EXEC [SP] @Id", idParameter).ToList();

        return q;
    }

    /// <summary>
    /// 演示导航属性
    /// </summary>
    /// <returns></returns>
    public object NavigationProperty()
    {
        //导航属性演示
        var x = this._shelfRepository.Table.Where(o => o.Id == 20).OrderBy(o => o.Id).Skip(0).Take(10).Select(o => new
        {
```

```
                WId = o.Warehouse.Id,
                WName = o.Warehouse.WHName,
                Id = o.Id
            }).ToList();

            return x;
        }

        /// <summary>
        /// 非动态 API 接口
        /// </summary>
        /// <param name="x"></param>
        /// <returns></returns>
        public int Method2(int x)
        {
            return 0;
        }
    }
}
```

将插件程序集和服务类程序集都引用到接口 HOST 站点后，我们将会看到我们定义在 TestService 里的方法已经自动映射成 API 接口了

DTO生成器(DtoGenerator)(By:芙蓉兴盛社区网络股份有限公司，Version:1.0.0.0)

Description:

直接根据自定义的SQL查询语句，输出对应的DTO对象。

Dependencies:

- Autofac, Version=3.0.0.0, Culture=neutral, PublicKeyToken=17863af14b0044da
- EntityFramework, Version=6.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
- Frxs.ServiceCenter.Api.Core, Version=2.8.0.16881, Culture=neutral, PublicKeyToken=null
- Frxs.ServiceCenter.Api.Core.Data, Version=3.0.0.0, Culture=neutral, PublicKeyToken=null
- System, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
- System.Data, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
- System.Web, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a
- System.Web.Mvc, Version=4.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35
- System.Xml, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089

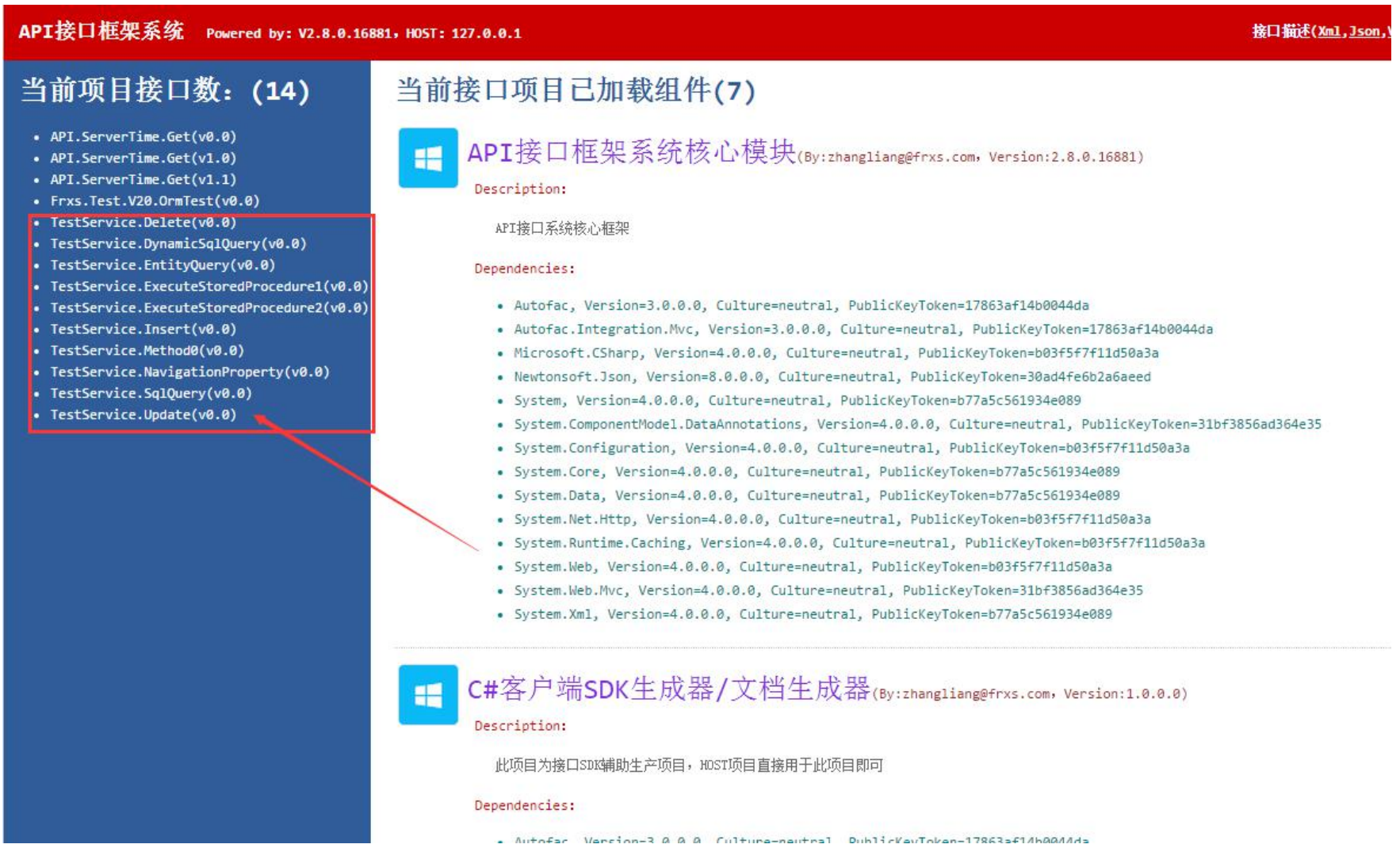
动态接口(DynamicApi)(By:芙蓉兴盛社区网络股份有限公司，Version:1.0.0.0)

Description:

此插件可以直接将业务层(Services)，映射成接口API。

Dependencies:

- Autofac, Version=3.0.0.0, Culture=neutral, PublicKeyToken=17863af14b0044da
- Frxs.ServiceCenter.Api.Core, Version=2.8.0.16881, Culture=neutral, PublicKeyToken=null
- System, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
- System.Core, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
- System.Web, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a


API接口框架系统 Powered by: V2.8.0.16881, HOST: 127.0.0.1
接口描述(Xml,Json,
当前项目接口数：(14)
API.ServerTime.Get(v0.0)
API.ServerTime.Get(v1.0)
API.ServerTime.Get(v1.1)
Frxs.Test.V20.OrmTest(v0.0)
TestService.Delete(v0.0)
TestService.DynamicSqlQuery(v0.0)
TestService.EntityQuery(v0.0)
TestService.ExecuteStoredProcedure1(v0.0)
TestService.ExecuteStoredProcedure2(v0.0)
TestService.Insert(v0.0)
TestService.Method0(v0.0)
TestService.NavigationProperty(v0.0)
TestService.SqlQuery(v0.0)
TestService.Update(v0.0)
当前接口项目已加载组件(7)
API接口框架系统核心模块(By:zhangliang@frxs.com, Version:2.8.0.16881)
Description:
API接口系统核心框架
Dependencies:
Autofac, Version=3.0.0.0, Culture=neutral, PublicKeyToken=17863af14b0044da
Autofac.Integration.Mvc, Version=3.0.0.0, Culture=neutral, PublicKeyToken=17863af14b0044da
Microsoft.CSharp, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a
Newtonsoft.Json, Version=8.0.0.0, Culture=neutral, PublicKeyToken=30ad4fe6b2a6aeed
System, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
System.ComponentModel.DataAnnotations, Version=4.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35
System.Configuration, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a
System.Core, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
System.Data, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
System.Net.Http, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a
System.Runtime.Caching, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a
System.Web, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a
System.Web.Mvc, Version=4.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35
System.Xml, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
C#客户端SDK生成器/文档生成器(By:zhangliang@frxs.com, Version:1.0.0.0)
Description:
此项目为接口SDK辅助生产项目，HOST项目直接用于此项目即可
Dependencies: