FRXS接口框架技术文档

1.设计思想

芙蓉兴盛API接口框架(V20)采取插件式设计(实现扩展的项目不会对核心项目有任何影响)，对http请求信息进行处理;接口扩展只需继承框架核心基类(ActionBase)即可，新增，修改，删除，迭代接口对其他接口无任何影响，框架默认实现了日志记录器，缓存器，接口访问记录器（方便管理统计），权限校验器，加解密器，作业任务。框架采取插件开发模式，方便功能点的扩展。

2.需要了解的一些概念以及组件使用

接口框架采取了插件式架构开发模式，因此代码里会引入一些设计模式，比如:依赖倒置，控制反转，依赖注入，IOC容器，AOP切面编程等。如果在此之前没有熟悉过，建议可以先了解下，便于理解代码架构设计。另外，框架引用了一些第三方组件，如果之前没有熟悉，可以通过下面给出的网址来了解下其基本用法。

2.1 依赖注入：AutoFac，官网：http://docs.autofac.org/en/latest/

2.2 序列化与反序列：Newtonsoft.Json，官网：http://www.newtonsoft.com/json

2.3 ORM(扩展项目会介绍到)：EntityFramework 6.1.3 ，官网：https://msdn.microsoft.com/zh-cn/data/ef

3.框架主要流程以及核心接口解析

3.1 系统启动

系统启动的时候，会自动调用ApiApplication.Initialize方法，进行系统注册，内部实现的流程为：首先会注册此方法传入的配置文件，如果未传入配置文件，那么系统会自动搜索接口服务根目录下面的~/ApiConfig.cs文件；如果存在此文件系统将会进行配置注册。注册完配置文件后，进行系统所有服务接口注册，具体调用：ServicesContainer.Current.Init()，注册完所有服务后，进行注册所有实现IRouteProvider接口，此接口为路由配置接口，注册路由后将自动启动实现IStartUp接口的所有实现类

3.2 接口搜索器(IActionSelector，DefaultActionSelector)

接口搜索器作为接口框架核心接口，其作用就是查找所有合法的Action接口，那么什么样的Action类才是合法的接口呢？系统框架里定义，只有满足下列2个条件的类，才是合法的Action接口

1. ***继承ActionBase抽象类，且抽象基类第一个参数RequestDto必须继承自RequestDotBase对象***
2. ***Action类可访问类型必须为公开public***

知道了合法的Action定义后，我再来开下 IActionSelector 接口,它定义了3个查找具体实现接口的方法，具体定义请看下面接口定义：

```
/// <summary>
/// 用于搜索所有合法的 Action 接口信息
/// </summary>
public interface IActionSelector
{
    /// <summary>
    /// 获取全部实现的接口（实现里需要实现将有移除特性 ObsoleteAttribute 的 Action 过滤掉）
    /// </summary>
    /// <param name="skipSystemActions">是否需要跳过系统接口</param>
    IEnumerable<ActionDescriptor> GetActionDescriptors(bool skipSystemActions = false);

    /// <summary>
    /// 根据接口名称获取接口信息
    /// </summary>
    /// <param name="actionName">接口名称，具体实现里请实现接口名称大小写不敏感</param>
    /// <returns></returns>
    IEnumerable<ActionDescriptor> GetActionDescriptors(string actionName);

    /// <summary>
    /// 根据接口名称获取接口描述
    /// </summary>
    /// <param name="actionName">接口名称，具体实现里请实现接口名称大小写不敏感</param>
    /// <param name="version">接口版本,如果不指定版本号，请实现类里实现返回版本号最大的接口描述对象</param>
```

```
        /// <returns>返回接口描述，如果不存在则返回 null</returns>
        ActionDescriptor GetActionDescriptor(string actionName, string version);

        }
```

系统框架里同时定义了一个默认的接口查找器实现（DefaultActionSelector），具体实现逻辑为,采取惰性扫描加载bin目录下所有程序集，检测程序集里定义的类是否继承自 ActionBase 抽象类,如果是，则根据接口类型构造出接口描述对象 ActionDescriptor 添加到静态缓存字典；这样在第二次访问时候就无需再次扫描，提高程序加载速度。具体实现为：

```
    /// <summary>
    /// 系统默认的 Action 查找器
    /// </summary>
    public class DefaultActionSelector : IActionSelector
    {
        /// <summary>
        /// 缓存所有的 action 接口，便于快速检索方法，提高接口检索速度;忽略大小写
        /// </summary>
        private static readonly IList<ActionDescriptor> CachedActionDescriptors = new List<ActionDescriptor>();
        /// <summary>
        /// 用于保存对应的接口名称和版本，方便快速的判断接口是否存在
        /// </summary>
        private static readonly IList<string> CachedActionNames = new List<string>();
        private static bool _initializationed = false;
        private static readonly object Locker = new object();
        /// <summary>
        ///
        /// </summary>
        private readonly ITypeFinder _typeFinder;

        /// <summary>
        /// 系统默认的 Action 查找器
        /// </summary>
        /// <param name="typeFinder">类型查找接口</param>
        public DefaultActionSelector(ITypeFinder typeFinder)
        {
            typeFinder.CheckNullThrowArgumentNullException("typeFinder");
            this._typeFinder = typeFinder;
        }

        /// <summary>
        /// 获取所有的合法接口
        /// </summary>
        /// <returns>返回所有合法的接口 ActionDescriptor 集合</returns>
        /// <param name="skipSystemActions">是否跳过系统框架接口</param>
        public virtual IEnumerable<ActionDescriptor> GetActionDescriptors(bool skipSystemActions = false)
        {
            //还未初始化
            if (_initializationed)
            {
                return CachedActionDescriptors;
            }

            lock (Locker)
            {
                if (_initializationed)
                {
                    return CachedActionDescriptors;
                }

                //已经初始化了标志
                _initializationed = true;

                //筛选接口
                this._typeFinder.FindClassesOfType(typeof(IAction))
                    //必须要继承自 ActionBase 抽象基类，将过期的接口直接忽略掉
                    .Where(type => type.IsAssignableToActionBase()).OrderBy(o => o.Name).ToList().ForEach(type =>
                    {
                        //创建一个接口描述对象
                        var actionDescriptor = new ReflectedActionDescriptor(type).GetActionDescriptor();

                        //已经注销
                        if (actionDescriptor.IsObsolete)
                        {
                            return;
                        }

                        //接口名称
                        string actionName = actionDescriptor.ActionName;

                        //查询缓存里是否存在了
                        var existsActionDescriptor = CachedActionDescriptors.FirstOrDefault(o => o.ActionName.Equals(actionName,
StringComparison.OrdinalIgnoreCase) && o.Version == actionDescriptor.Version);

                        //不存在就添加到缓存
                        if (existsActionDescriptor.IsNull())
                        {
                            //添加到全局接口描述缓存
                            CachedActionDescriptors.Add(actionDescriptor);

                            //添加到接口名称缓存，方便快速判断是否存在接口以及指定的版本
```

```
                        CachedActionNames.Add("{0}${1}".With(actionName, actionDescriptor.Version));
                    }
                    else
                    {
                        //还原下操作
                        _initializationed = false;
                        CachedActionDescriptors.Clear();
                        CachedActionNames.Clear();
                        //已经存在了键（实现类配置了相同的接口名称）；直接抛出异常，便于在开发期就发现问题
                        throw new
ApiException(Resource.CoreResource.DefaultActionSelector_FoundMoreThenOneActionName.With(actionName, type.FullName,
existsActionDescriptor.ActionType.FullName, actionDescriptor.Version));
                    }
                });
            }

            //返回接口集合
            return CachedActionDescriptors;
        }

        /// <summary>
        /// 根据接口名称获取接口信息
        /// </summary>
        /// <param name="actionName">接口名称，忽略大小写</param>
        /// <returns>返回指定接口名称下面的所有接口版本</returns>
        public IEnumerable<ActionDescriptor> GetActionDescriptors(string actionName)
        {
            return this.GetActionDescriptors().Where(o => o.ActionName.Equals(actionName,
StringComparison.OrdinalIgnoreCase)).ToList();
        }

        /// <summary>
        /// 根据接口名称获取接口描述对象
        /// </summary>
        /// <param name="actionName">接口名称</param>
        /// <param name="version">接口版本，版本设置为空或者 null，框架将会使用同名接口版本号最大的接口</param>
        /// <returns>如果未找到则返回 null</returns>
        public ActionDescriptor GetActionDescriptor(string actionName, string version)
        {
            //接口不存在直接返回 null
            if (actionName.IsNullOrEmpty())
            {
                return null;
            }

            //获取接口所有版本
            var actionDescriptors = this.GetActionDescriptors(actionName);

            //为找到任何版本信息
            if (actionDescriptors.IsNull() || actionDescriptors.IsEmpty())
            {
                return null;
            }

            //未指定版本号，直接获取指定接口最新的版本
            if (version.IsNullOrEmpty())
            {
                return actionDescriptors.OrderByDescending(o => o.Version).FirstOrDefault();
            }

            //指定了版本号，就直接查找执行版本即可
            return actionDescriptors.FirstOrDefault(o => o.Version == version);
        }
    }
```

知道了Action 接口是怎么被查找出来后，我们再来看一下作为Action接口基类的 ActionBase 抽象类，它定义为一个泛型抽象基类，带2个泛型参数，第一个参数是客户端上送业务参数对应的RequestDto接受对象，此对象必须继承RequestDtoBase。第二个参数是任意类型的参数，是接口处理后返回的真实业务数据对象，此对象对应于ActionResult.Data属性。另外**ActionBase**也是所有Action接口的基类，在开发过程中，如果一个类，没有继承它，系统框架将不会认为类是一个接口（即使实现了IAction接口）,**ActionBase**里定义了一系列方便操作的属性，比如：Logger,CacheManager等等，来方便日志记录和缓存管理。下面我们来看一下其定义：

```
/// <summary>
/// 核心类，接口 action 抽象类；所有外部实现的接口都需要继承此类
/// 注意：上送参数和下送数据不支持字典对象数据类型，能够用简易数据类型表达的，尽量用简易数据类型
/// </summary>
/// <typeparam name="TRequestDto">
/// 客户端上传 Data 参数 JSON 反序列化后对应的对象，此对象必须是继承于 RequestDtoBase 类的一个实体类
/// 命名约定规则为：接口类名+RequestDto。如果需要校验客户端 UserId 和 UserName 是
/// 否提交，实现类里请继承接口：IRequiredUserIdAndUserName，这样系统框架会在执行前先校验参数的
/// 准确性，数据约束规则完全兼容命名空间 System.ComponentModel.DataAnnotations 下是所有特性。可
/// 以方便的在上送参数对象属性上定义特性的方式来进行数据验证
/// </typeparam>
/// <typeparam name="TResponseDto">
/// 输出 ActionResult.Data 对象，就可以是任意的输出 DTO 类型，没有对此类型进行泛型约束
/// 如果此泛型类型指定为：NullResponseDto，系统自动生成 SDK 开发包的时候，将不会生成返回输出类
/// 注意:尽量不要将此类型定义成 object 类型，要不框架无法自动生成 SDK 开发包（客户端需要手工进行处理）
```

```
    /// 集合类型请尽量使用 List 数据类型方便系统框架自动完成 SDK 输出（注意：不支持字典，字典可以使用集合来替代）
    /// </typeparam>
    public abstract class ActionBase<TRequestDto, TResponseDto> : IAction, IActionFilter, IDisposable
        where TRequestDto : RequestDtoBase, new()
    {
        /// <summary>
        /// 默认构造一个空的记录器，并且初始化一个空的日志记录器和一个空的缓存器
        /// </summary>
        protected ActionBase()
        {
            this.Logger = NullLogger.Instance;
            this.CacheManager = NullCacheManager.Instance;
        }

        /// <summary>
        /// 用于记录日志，系统默认使用了空记录日志，如果需要使用其他日志记录器，比如将日志记录到数据库或者其他存储介质
        /// 请在外部实现 ILogger 接口，然后注入覆盖系统默认的记录器
        /// </summary>
        public ILogger Logger { get; set; }

        /// <summary>
        /// API 框架提供的缓存接口，具体实现请在外部实现具体的缓存实现(这里系统框架未实现任何缓存);
        /// 在进行具体的使用过程中，由于有可能会缓存键会引起冲突，建议缓存键使用 this.GetType().FullName 加具体缓存键来实现键的冲突或者在外部定义好预定义的键，供具体实现类里调用
        /// 由于接口层并不知道数据库的实体对象变化，因此建议在接口层使用缓存一般是在预知变化不会太频繁的情况下使用，其他情况下慎用；或者在实体增加，修改，删除的时候，全部做缓存设置，删除操作
        /// </summary>
        public ICacheManager CacheManager { get; set; }

        /// <summary>
        /// 当前 action 请求上下文信息(系统框架会自动赋值)
        /// </summary>
        public RequestContext RequestContext { get; set; }

        /// <summary>
        /// 当前 action 请求描述信息(系统框架会自动赋值)
        /// </summary>
        public ActionDescriptor ActionDescriptor { get; set; }

        /// <summary>
        /// 获取接口描述管理器
        /// </summary>
        protected ActionDocResourceManager ActionDocResourceManager
        {
            get
            {
                return ActionDocResourceManager.Instance;
            }
        }

        /// <summary>
        /// 上送的参数（此参数系统在参加 action 实例的时候，会自动根据上送的参数进行绑定赋值）
        /// 此属性与强类型的 RequestDto 属性数据保持一致
        /// </summary>
        object IAction.RequestDto { get; set; }

        /// <summary>
        /// 上送的参数（此参数系统在参加 action 实例的时候，会自动根据上送的参数进行绑定赋值）
        /// 注意：当上送的 JSON 格式不正确的时候，系统框架会自动抛出异常，无需处理；因此此属性一定有返回值
        /// </summary>
        protected TRequestDto RequestDto
        {
            get
            {
                //获取接口请求参数对象
                var requestDto = (this as IAction).RequestDto;

                //为空直接返回默认值
                if (requestDto.IsNull())
                {
                    return default(TRequestDto);
                }

                //转型成功，直接返回泛型模板对象
                if (requestDto is TRequestDto)
                {
                    return (TRequestDto)requestDto;
                }

                //返回泛型模板类型默认值
                return default(TRequestDto);
            }
        }

        /// <summary>
        /// 校验上送的参数是否正确（默认会校验是否为 null）；失败会直接抛出异常，系统框架会自动捕捉到此异常
        /// 此方法在框架执行 action.Executing()方法里会自动进行调用，如果需要改变上送参数对象数据校验，请在重写类里重写此方法即可，但是一般情况下无需重写
        /// </summary>
        protected virtual void ValidRequestDto()
        {
```

```
            //0.上送的参数对象不能为空
            if (this.RequestDto.IsNull())
            {
                throw new ApiException(Resource.CoreResource.ActionBase_RequestDto_Null);
            }

            //1.如果 RequestDto 继承了 IRequestDtoValidatable 或者添加了特性校验
            var requestDtoValidatorResult = new RequestDtoValidator(this.RequestDto, this.ActionDescriptor).Valid();

            //校验失败，直接抛出异常
            if (!requestDtoValidatorResult.IsValid)
            {
                throw new ApiException(string.Join(" | ", requestDtoValidatorResult.Errors.Select(item =>
"{0}:{1}".With(item.MemberName, item.ErrorMessage))));
            }
        }

        /// <summary>
        /// 获取当前请求获取缓存键信息，方便重写实现类里直接使用
        /// 只要接口名称和提交的数据包不变，生成的那么缓存键就不会变化，因此实现针对不同接口和不同请求数据包进行缓存
        /// 具体计算方式为：ActionName + "." + subCacheKey + "." + Units.MD5(Data + Format).ToUpper();
        /// </summary>
        /// <param name="subCacheKey">同一操作上下文，有可能需要不同的子缓存键；可以增加子缓存键，防止冲突</param>
        /// <returns>返回本次请求缓存键</returns>
        protected string GetRequestCacheKey(string subCacheKey = "")
        {
            return this.RequestContext.GetRequestCacheKey(subCacheKey);
        }

        /// <summary>
        /// 语言资源文件的读取；内部使用 XML 资源文件；
        /// 参数 1 为资源文件的键，自定义的语言资源包请使用接口的 ActionName 来作为节点名称，此委托会自动使用接口名称来构造键；KEY 只要输入
对应接口语言文件 key 节点名称即可
        /// 参数 2 为在资源文件找不到的情况下，默认显示的信息；
        /// 返回值为读取到的键值信息；
        /// 资源文件的格式为：请参见：Host/Config/LanguageResource.xml
        /// 调用如：this.L("Exist_S","删除 {0} 出错，当前状态为：{1}" , "001", "已确定")
        /// </summary>
        protected Localizer L
        {
            get
            {
                return (resourceKey, defaultValue, args) =>
                {
                    //获取语言包
                    var languageResourceManager = LanguageResourceManager.Instance;

                    //语言包不存在
                    if (languageResourceManager.IsNull())
                    {
                        return defaultValue ?? string.Empty;
                    }

                    //key 为空，直接返回默认的说明
                    if (resourceKey.IsNullOrEmpty())
                    {
                        return defaultValue ?? string.Empty;
                    }

                    //构造资源的 key 值;（忽略大小写）
                    string key = "{0}.{1}".With(this.ActionDescriptor.ActionName, resourceKey);

                    //获取资源
                    string value = languageResourceManager.GetLanguageResourceValue(key, defaultValue);

                    //含有参数就进行格式化
                    return args.IsNull() || args.Length == 0 ? value : value.With(args);
                };
            }
        }

        /// <summary>
        /// 框架异常错误的 ActionResult 对象
        /// 对象默认的参数为： Flag = ActionResultFlag.EXCEPTION
        /// </summary>
        /// <param name="info">异常消息</param>
        /// <returns></returns>
        protected ActionResult<TResponseDto> ExceptionActionResult(string info)
        {
            return new ActionResult<TResponseDto>() { Flag = ActionResultFlag.EXCEPTION, Info = info };
        }

        /// <summary>
        /// 直接返回错误的 ActionResult 对象（此方法仅仅是为了实现类里方便调用返回）
        /// 对象默认参数为：Flag = ActionResultFlag.FAIL
        /// </summary>
        /// <param name="info">输出的错误消息</param>
        /// <returns>直接返回一个 Flag = ActionResultFlag.FAIL 的接口返回值对象</returns>
        protected ActionResult<TResponseDto> ErrorActionResult(string info)
        {
```

```
    return new ActionResult<TResponseDto>() { Flag = ActionResultFlag.FAIL, Info = info };
}

/// <summary>
/// 直接返回错误的 ActionResult 对象（此方法仅仅是为了实现类里方便调用返回）
/// </summary>
/// <param name="info">输出的错误消息</param>
/// <param name="data">返回的一些数据(会被序列化成 JSON 或者 XML 格式输出给客户端)</param>
/// <returns>直接返回一个 Flag = ActionResultFlag.FAIL 的接口返回值对象</returns>
protected ActionResult<TResponseDto> ErrorActionResult(string info, TResponseDto data)
{
    return new ActionResult<TResponseDto>() { Flag = ActionResultFlag.FAIL, Info = info, Data = data };
}

/// <summary>
/// 请求参数未提交错误 ActionResult 对象（此方法仅仅是为了方便实现类里方便调用返回）
/// </summary>
/// <param name="argumentName">参数名称</param>
/// <returns>返回一个现实错误的 ActionResult 对象</returns>
protected ActionResult<TResponseDto> ArgumentNullErrorActionResult(string argumentName)
{
    return new ActionResult<TResponseDto>()
    {
        Flag = ActionResultFlag.FAIL,
        Info = Resource.CoreResource.ActionBase_ArgumentNullErrorActionResult_Info.With(argumentName)
    };
}

/// <summary>
/// 此方法仅仅用于返回一个不带任何返回值的 ActionResult 对象
/// 对象默认参数为：Flag = ActionResultFlag.SUCCESS, Info = "OK"
/// </summary>
/// <returns></returns>
protected ActionResult<TResponseDto> SuccessActionResult()
{
    return new ActionResult<TResponseDto>() { Flag = ActionResultFlag.SUCCESS, Info = "OK" };
}

/// <summary>
/// 此方法用于返回一个成功的消息 ActionResult 对象
/// 对象默认参数为：Flag = ActionResultFlag.SUCCESS, Info = "OK"
/// </summary>
/// <param name="data">需要返回给客户端的对象，会格式化成 XML 或者 JSON</param>
/// <returns></returns>
protected ActionResult<TResponseDto> SuccessActionResult(TResponseDto data)
{
    return new ActionResult<TResponseDto>() { Flag = ActionResultFlag.SUCCESS, Info = "OK", Data = data };
}

/// <summary>
/// 此方法用于返回一个成功的消息 ActionResult 对象
/// 对象默认参数为：Flag = ActionResultFlag.SUCCESS
/// </summary>
/// <param name="info">成功消息说明</param>
/// <param name="data">需要返回给客户端的对象，会格式化成 XML 或者 JSON</param>
/// <returns></returns>
protected ActionResult<TResponseDto> SuccessActionResult(string info, TResponseDto data)
{
    return new ActionResult<TResponseDto>() { Flag = ActionResultFlag.SUCCESS, Info = info, Data = data };
}

/// <summary>
/// 显式实现下接口，防止子类里出现，调用的时候出现意外
/// 开始执行接口自定义的业务逻辑前，先执行下框架内部定义的一些判断
/// 此方法在内部调用过来受保护的 ValidRequestDto()方法，用于校验上送参数对象的正确性；
/// </summary>
void IAction.Executing()
{
    this.ValidRequestDto();
}

/// <summary>
/// 显式实现接口，此接口仅仅只调用泛型版本的方法
/// </summary>
/// <returns></returns>
ActionResult<object> IAction.Execute()
{
    //执行外部定义的主方法
    var actionResult = this.Execute();

    //返回执行对象，转型成 object 类型
    return new ActionResult<object>()
    {
        Data = actionResult.Data,
        Flag = actionResult.Flag,
        Info = actionResult.Info
    };
}

/// <summary>
///  泛型执行接口，系统框架执行的时候会自动调用此方法
```

```
        ///  此方法必须在具体接口实现类里重写
        /// </summary>
        /// <returns></returns>
        public abstract ActionResult<TResponseDto> Execute();

        /// <summary>
        /// 执行：Execute() 方法前执行
        /// </summary>
        /// <param name="actionExecutingContext">
        /// 拦截器执行上下文，接口执行前拦截器，如果 ActionExecutingContext.Result 属性不为 null，
        /// 则代表拦截成功(如要需要进行拦截，请在方法体内将 ActionExecutingContext.Result 属性赋值即可)，不会继续执行 Execute()方法
        /// 保存默认值 null，则代表不拦截，继续后续的流程执行
        /// </param>
        protected virtual void OnActionExecuting(ActionExecutingContext actionExecutingContext)
        {
        }

        /// <summary>
        /// 显式实现下执行前触发的事件，方便以后扩展（系统框架里做一些事情）
        /// </summary>
        /// <param name="actionExecutingContext">拦截器执行上下文</param>
        void IActionFilter.OnActionExecuting(ActionExecutingContext actionExecutingContext)
        {
            this.OnActionExecuting(actionExecutingContext);
        }

        /// <summary>
        /// 执行：Execute() 方法后执行
        /// </summary>
        /// <param name="actionExecutedContext">拦截器执行上下文</param>
        protected virtual void OnActionExecuted(ActionExecutedContext actionExecutedContext)
        {
        }

        /// <summary>
        /// 显式实现下执行后触发的事件，方便以后扩展（系统框架里做一些事情）
        /// </summary>
        /// <param name="actionExecutedContext">拦截器执行上下文</param>
        void IActionFilter.OnActionExecuted(ActionExecutedContext actionExecutedContext)
        {
            #region 先处理下需要清理的缓存键

            if (!this.ActionDescriptor.UnloadCacheKeys.IsNull() && this.ActionDescriptor.UnloadCacheKeys.Length > 0)
            {
                //启动匹配缓存键的方式，删除相关缓存键
                foreach (var unloadCacheKey in this.ActionDescriptor.UnloadCacheKeys)
                {
                    try
                    {
                        this.CacheManager.RemoveByPattern(unloadCacheKey);
                        this.Logger.Debug("清理缓存匹配键：{0} 成功".With(unloadCacheKey));
                    }
                    catch (Exception ex)
                    {
                        this.Logger.Error(ex);
                    }
                }
            }

            #endregion

            //执行完所有的事件之后
            this.OnActionExecuted(actionExecutedContext);
        }

        /// <summary>
        /// 基类默认实现下 IDispose
        /// </summary>
        public void Dispose()
        {
            this.Dispose(true);
            //GC.SuppressFinalize(this);
        }

        /// <summary>
        /// 具体接口实现类可以重写此方法，系统框架在执行完接口后，会自动调用此方法
        /// </summary>
        /// <param name="disposing"></param>
        protected virtual void Dispose(bool disposing) { }
    }
```

3.3 接口创建器(IActionFactory，DefaultActionFactory)

通过接口搜索器 IActionSelector 找出所有接口类型后，我们需要将接口类型进行激活， 负责创建接口实例的接口就是 IActionFactory, 接口创建器定义了3个接口方法，2个Create()方法为创建出接口对象实例，ReleaseAction()方法为释放接口资源(谁创建谁负责销毁)。具体定义如下：

```
    /// <summary>
```

```
/// 接口创建器
/// </summary>
public interface IActionFactory
{
    /// <summary>
    /// 根据指定接口名称创建一个调用接口
    /// </summary>
    /// <param name="requestContext">当前请求上下文</param>
    /// <param name="actionName">接口名称，实现类尽量实现大小写不敏感</param>
    /// <param name="version">接口版本，实现类里需要实现，如果未指定接口版本，那么就获取接口最新版本号</param>
    /// <returns></returns>
    IAction Create(RequestContext requestContext, string actionName, string version);

    /// <summary>
    /// 创建一个接口实例
    /// </summary>
    /// <param name="requestContext">当前请求上下文</param>
    /// <param name="actionDescriptor">接口描述对象</param>
    /// <returns></returns>
    IAction Create(RequestContext requestContext, ActionDescriptor actionDescriptor);

    /// <summary>
    /// 释放 action 占用的资源，框架在执行完 IAction.Execute()方法后，执行此方法
    /// </summary>
    /// <param name="action">action 实例</param>
    void ReleaseAction(IAction action);
}
```

同接口查找器一样，系统同样定义了默认实现(DefaultActionFactory)，具体实现的代码如下：

```
/// <summary>
/// 默认的接口激活器实现类，内部使用了缓存提高执行效率
/// </summary>
public class DefaultActionFactory : IActionFactory
{
    /// <summary>
    ///
    /// </summary>
    private readonly IActionActivator _actionActivator;
    private readonly IActionSelector _actionSelector;

    /// <summary>
    /// 日志记录器
    /// </summary>
    public ILogger Logger { get; set; }

    /// <summary>
    /// 方便注入新的激活器
    /// </summary>
    /// <param name="actionSelector">合法的 Action 接口查找器</param>
    /// <param name="actionActivator">需要定义接口激活器</param>
    public DefaultActionFactory(IActionSelector actionSelector, IActionActivator actionActivator)
    {
        //判断 null
        actionActivator.CheckNullThrowArgumentNullException("actionActivator");
        actionSelector.CheckNullThrowArgumentNullException("actionSelector");

        this._actionActivator = actionActivator;
        this._actionSelector = actionSelector;
        this.Logger = NullLogger.Instance;
    }

        /// <summary>
        /// 根据接口名称，创建对应的接口实例
        /// </summary>
        /// <param name="requestContext">当前请求上下文信息</param>
        /// <param name="actionName">大小写不敏感</param>
        /// <param name="version">接口版本</param>
        /// <returns></returns>
        public virtual IAction Create(RequestContext requestContext, string actionName, string version)
        {
            try
            {
                //requestContext 参数不能为 null；系统框架级错误，直接抛出异常
                requestContext.CheckNullThrowArgumentNullException("requestContext");

                //接口名称未提供
                if (actionName.IsNullOrEmpty())
                {
                    return new ErrorAction(new Exception(Resource.CoreResource.DefaultActionFactory_ActionNameIsNullOrString),
requestContext);
                }

                //获取到接口描述
                var actionDescriptor = this._actionSelector.GetActionDescriptor(actionName, version);

                //未找到接口(忽略大小写)
                if (actionDescriptor.IsNull())
                {
                    return new ErrorAction(new
```

```
Exception(Resource.CoreResource.DefaultActionFactory_ActionNameNotFound.With(actionName)), requestContext);
                }

                //进行接口激活，创建接口实例
                return this.Create(requestContext, actionDescriptor);
            }
            catch (Exception ex)
            {
                this.Logger.Error(ex);
                return new ErrorAction(ex, requestContext);
            }
        }

        /// <summary>
        /// 接口描述对象
        /// </summary>
        /// <param name="requestContext">当前请求上下文</param>
        /// <param name="actionDescriptor">接口描述对象</param>
        /// <returns>返回一个接口实例</returns>
        public virtual IAction Create(RequestContext requestContext, ActionDescriptor actionDescriptor)
        {
            requestContext.CheckNullThrowArgumentNullException("requestContext");
            actionDescriptor.CheckNullThrowArgumentNullException("actionDescriptor");

            //进行接口激活，创建接口实例
            var action = (IAction)this._actionActivator.Create(actionDescriptor.ActionType);

            //赋值接口描述对象
            action.ActionDescriptor = actionDescriptor;

            //请求上下文
            action.RequestContext = requestContext;

            //上送参数对象赋值
            action.RequestDto = requestContext.RequestDto;

            //返回接口实例
            return action;
        }

        /// <summary>
        /// 释放资源直接看 action 继承了 IDisposable;如果继承了，直接调用 IDisposable.Dispose()方法
        /// </summary>
        /// <param name="action">action 实例</param>
        public virtual void ReleaseAction(IAction action)
        {
            IDisposable disposable = action as IDisposable;
            if (!disposable.IsNull())
            {
                disposable.Dispose();
            }
        }
    }
```

从上面具体的接口创建器实现可以看出，默认的接口实例创建器，依赖于2个重要的接口，接口查找器（IActionSelector）和接口激活器（IActionActivator），接口的激活并不是直接从此实现里使用反射的方式来进行创建实例，而是从激活器里去激活。ReleaseAction() 方法实现比较简单，只检测接口是否实现了 IDisposable 接口，然后调用 IDisposable.Dispose()方法来进行资源释放.

3.4 接口激活器(IActionActivator,DefaultActionActivator)

顾名思义，接口激活器，即：如何对具体的实现接口进行激活。激活器仅仅定义了一个简单的唯一方法 Create(Type actionType) ，此方法根据接口类型，创建出一个接口实例。具体的定义如下：

```
/// <summary>
/// action 激活器
/// </summary>
public interface IActionActivator
{
    /// <summary>
    /// 通过 action 类型，激活 action
    /// </summary>
    /// <param name="actionType">IAction 类型信息</param>
    /// <returns>Action 对象实例</returns>
    IAction Create(Type actionType);
}
```

同样系统框架定义了一个默认的激活器：DefaultActionActivator 具体的实现代码如下

```
/// <summary>
/// action 激活器实现
/// </summary>
public class DefaultActionActivator : IActionActivator
{
    /// <summary>
    /// 使用属性注入的方式注入日志组件
    /// </summary>
```

```
        public ILogger Logger { get; set; }

        /// <summary>
        /// 为了不让写日志出错（属性注入是惰性注入，因此需要进行构造函数指定一个空的日志实现）
        /// </summary>
        public DefaultActionActivator()
        {
            this.Logger = NullLogger.Instance;
        }

        /// <summary>
        /// 创建指定类型的 Action 对象
        /// </summary>
        /// <param name="actionType">实现 ActionBase 的实现类类型</param>
        /// <returns></returns>
        public virtual IAction Create(Type actionType)
        {
            //actionType 不能为 null
            actionType.CheckNullThrowArgumentNullException("actionType");

            //没有继承自 ActionBase 抽象基类
            if (!actionType.IsAssignableToActionBase())
            {
                throw new ApiException(Resource.CoreResource
                          .DefaultActionActivator_TypeError
                          .With(actionType.FullName));
            }

            //从 IOC 容器里创建一个新的 Action 接口对象
            try
            {
                return (IAction)ServicesContainer.Current.Resolver(actionType);
            }
            catch (Exception exception)
            {
                this.Logger.Error(exception.StackTrace);
                throw new ApiException(exception.Message, exception.InnerException);
            }
        }
    }
```

从具体的实现可以看出，创建接口实例并不是使用 `Activator.CreateInstance()` 方法进行创建的，而是直接从服务容器（IOC）里来进行创建实例，这样可以方便接口实现类注入其他服务

3.5 接口执行(IActionInvoker,DefaultActionInvoker)

当我们通过 IActionFactory 接口创建出接口实例后，系统框架并不是直接使用 IAction 实例里方法进行接口执行，而是将直接接口委托给了 IActionInvoker 接口来执行，这样做的目的就是方便在接口执行前后进行拦截，也方便后续扩展。具体的接口定义如下：

```
    /// <summary>
    /// Action 接口执行器（Action 代理封装），方便对 action 执行方法进行 AOP 横切
    /// </summary>
    public interface IActionInvoker
    {
        /// <summary>
        /// Action 执行器，通过当前 action 对象，调用 IAction.Execute()方法进行接口执行
        /// </summary>
        /// <param name="action">IAction 接口实例</param>
        /// <returns>ActionResult 对象</returns>
        ActionResult Execute(IAction action);
    }
```

同样框架默认实现的接口执行器为： `DefaultActionInvoker` ,其具体实现为：

```
    /// <summary>
    /// Action 接口执行器（对 Action 接口进行代理封装），方便对 action 执行方法进行 AOP 横切
    /// </summary>
    internal class DefaultActionInvoker : IActionInvoker
    {
        /// <summary>
        /// 日志记录器
        /// </summary>
        public ILogger Logger { get; set; }

        /// <summary>
        /// 默认 Action 激活器
        /// </summary>
        public DefaultActionInvoker()
        {
            this.Logger = NullLogger.Instance;
        }

        /// <summary>
        /// 此方法里进行更多的 http 条件判断
        /// </summary>
        /// <param name="action">实现 IAction 的实例</param>
```

```
/// <returns></returns>
public virtual ActionResult Execute(IAction action)
{
    //action 不能为 null
    action.CheckNullThrowArgumentNullException("action");

    //当前 action 的描述信息
    ActionDescriptor actionDescriptor = action.ActionDescriptor;

    //用于保存执行结果
    ActionResult actionResult = null;

    /* 过滤器执行顺序规则
     *
     * Action.Executing() --> Exception --> return ErrResult
     *
     * - A.OnActionExecuting()
     *
     * ----B.OnActionExecuting()
     *
     * --------C.OnActionExecuting()
     *
     * ------------Action.Execute() --> Exception --> return ErrResult
     *
     * --------C.OnActionExecuted()
     *
     * ----B.OnActionExecuted()
     *
     * - A.OnActionExecuted()
     *
     */

    #region 执行 action.Executing()

    try
    {
        //执行下框架判断
        action.Executing();
    }
    catch (Exception ex)
    {
        //抛出异常返回
        return new ActionResult() { Flag = ActionResultFlag.EXCEPTION, Info = ex.Message };
    }

    //先获取到全部的过滤器，包括（接口自身，全局过滤器，特性过滤器）
    var actionFilters = new List<ActionFilterWrapper> { ((IActionFilter)action).AsActionFilterWrapper() };

    //添加特性过滤器和全局过滤器
    actionFilters.AddRange(
        actionDescriptor.ActionFilters.Select(actionFilter => actionFilter.AsActionFilterWrapper()));

    #endregion

    #region 执行 OnActionExecuting

    //执行前拦截
    foreach (var item in
            actionFilters.OrderByDescending(o => o.InternalOrder)
                .ThenByDescending(o => o.Order)
                .Select(o => o.ActionFilter))
    {
        //一个拦截器出现错误，不能影响到其他拦截器的执行
        try
        {
            var actionExecutingContext = new ActionExecutingContext(action);
            item.OnActionExecuting(actionExecutingContext);
            //如果执行完毕后，直接结果不为 null，值直接返回了，不会进行下面的 action 执行了
            if (!actionExecutingContext.Result.IsNull())
            {
                return actionExecutingContext.Result;
            }
        }
        catch (Exception ex)
        {
            this.Logger.Error(ex);
        }
    }

    #endregion

    #region 执行 action.Execute()

    try
    {
        //正式执行业务逻辑
        var executedActionResult = action.Execute();
        //转型下
        actionResult = new ActionResult()
        {
            Data = executedActionResult.Data,
            Flag = executedActionResult.Flag,
```

```
            Info = executedActionResult.Info
        };
    }
    catch (Exception ex)
    {
        string errorMessage =
            Resource.CoreResource.DefaultActionInvoker_ActionExecuteError.With(actionDescriptor.ActionName,
                actionDescriptor.ActionType.FullName, ex.Message);
        //抛出异常返回
        actionResult = new ActionResult()
        {
            Flag = ActionResultFlag.EXCEPTION,
            Info = "Message: {0}, StackTrace: {1}".With(errorMessage, ex.StackTrace)
        };
        //记录下日志
        this.Logger.Error(ex);
    }

    #endregion

    #region 执行 OnActionExecuted

    //执行后拦截
    foreach (var item in actionFilters.OrderBy(o => o.InternalOrder).ThenBy(o => o.Order).Select(o => o.ActionFilter))
    {
        try
        {
            var actionExecutedContext = new ActionExecutedContext(action, actionResult);
            item.OnActionExecuted(actionExecutedContext);
            actionResult = actionExecutedContext.Result;
        }
        catch (Exception ex)
        {
            this.Logger.Error(ex);
        }
    }

    #endregion

    //返回值
    return actionResult;
  }
}
```

从实现的代码我们可以看出，在执行action.Execute()方法的前后，执行器都进行了拦截处理。执行前，执行后拦截器我们后续将讲到，简单来说我们可以继承 ActionFilterBaseAttribute 特性类附加到Action接口实现类上或者直接实现 IActionFilter 接口使用GlobalActionFiltersManager.Filters.Add(new GlobalRequestHanderInterceptors.DefaultGlobalRequestHanderInterceptor());注册全局拦截器，对接口执行进行拦截。同时我们也注意到 ActionBase 抽象类也实现了 IActionFilter 接口，然而，我们具体实现的接口需要继承 ActionBase 也就是说，我们实现的接口其实也是一个拦截器，我们可以通过重写 OnActionExecuting(ActionExecutingContext actionExecutingContext) 和 OnActionExecuted(ActionExecutedContext actionExecutedContext) 方法来进行接口拦截，3种拦截器的执行顺序为：全局优先级最高，接口本身的拦截器次之，特性定义的拦截器最后执行。

3.6 接口返回值ActionResult格式化（IMediaTypeFormatter，IMediaTypeFormatterFactory，DefaultMediaTypeFormatterFactory）

当我们通过 IActionInvoker 接口执行完接口后，我们会得到 ActionResult 对象 ，此对象为固定类型，所有接口执行完毕后，都会返回此类型。IMediaTypeFormatter 格式化器就是将返回的 ActionResult 对象进行格式化，格式化成XML或者JSON或者VIEW（视图）进行输出到客户端。格式化器具体定义为：

```
/// <summary>
/// 所有格式化器必须继承此抽象类；具体实现类里只要重写方法 SerializedActionresultToString()即可
/// 由于系统自带的 XML 格式化无法针对匿名对象实现格式化，所以需要在实现类里自己去使用递归方式去探测每一个属性对象，然后进行格式化
/// </summary>
public interface IMediaTypeFormatter
{
    /// <summary>
    /// 对象资源格式化成字符串
    /// </summary>
    /// <param name="requestContext">当前请求上下文</param>
    /// <param name="actionResult">ActionResult 对象</param>
    /// <returns>返回特定的 ActionResult 对象序列化字符串</returns>
    string SerializedActionResultToString(RequestContext requestContext, ActionResult actionResult);
}
```

格式化器具体实现，框架实现了3个格式化器 JsonMediaTypeFormatter,XmlMediaTypeFormatter,ViewMediaTypeFormatter 分别对应格式化成JSON,XML,VIEW；格式化成JSON和XML比较简单，就是利用了Newtonsoft.Json 组件进行序列化。具体实现代码为：

```
/// <summary>
/// JSON 格式化器
/// </summary>
public class JsonMediaTypeFormatter : IMediaTypeFormatter
{
```

```
        /// <summary>
        /// JSON 格式化器
        /// </summary>
        /// <param name="requestContext">当前请求上下文</param>
        /// <param name="actionResult">ActionResult 对象</param>
        /// <returns>输出序列化后的字符串</returns>
        public virtual string SerializedActionResultToString(RequestContext requestContext, ActionResult actionResult)
        {
            //返回格式化数据
            return actionResult.ToJson();
        }
    }
    /// <summary>
    /// XML 格式化器
    /// </summary>
    public class XmlMediaTypeFormatter : IMediaTypeFormatter
    {
        /// <summary>
        /// XML 格式化器
        /// </summary>
        /// <param name="requestContext">当前请求上下文</param>
        /// <param name="actionResult">ActionResult 对象</param>
        /// <returns>输出序列化后的字符串</returns>
        public virtual string SerializedActionResultToString(RequestContext requestContext, ActionResult actionResult)
        {
            return actionResult.ToXml();
        }
    }
```

VIEW的格式化比较复杂点，涉及到视图引擎，API接口框架使用了自定义视图引擎，来支持VIEW类型格式化。这里我们仅仅讲解下VIEW格式化器的基本原理。系统首先会从资源管理器 IResourceFinderManager 里检索指定的键，键即为视图路径，具体体现为：首先搜索~/Views/{AvtionName}.aspx，如果资源管理器里存在则返回，进行源代码编译。如果找不到则再次从资源管理器 IResourceFinderManager 里找路径：{NameSpace}.Views.{ActionName}.aspx ，如果找到则进行编译，返回结果，如果经过上面2轮查找还没有找到视图源码，最后就使用框架提供的Views.T.aspx视图源码来兜底。具体的视图引擎如果进行编译我们再后续再讲专门开一个主题章节来进行阐述。我们先看下实现代码：

```
    /// <summary>
    /// IAction 接口文档生成器；如果是第三方插件内嵌资源视图文件，
    /// </summary>
    public class ViewMediaTypeFormatter : IMediaTypeFormatter
    {
        /// <summary>
        /// 程序集内嵌资源实体查找器
        /// </summary>
        private readonly IResourceFinderManager _viewFinderManager;

        /// <summary>
        /// 内嵌资源路径规则
        /// </summary>
        private const string ManifestResourceViewSearchPath = "{0}.Views.{1}.aspx";

        /// <summary>
        /// 接口框架命名空间
        /// </summary>
        private static readonly string Namespace = Assembly.GetExecutingAssembly().GetName().Name;
        /// <summary>
        /// 换行符
        /// </summary>
        private static readonly string NewLine = Environment.NewLine;
```

```
        /// <summary>
        /// 视图搜索路径地址：如：~/views/{0}.aspx
        /// </summary>
        private string[] ViewLocationFormats { get; set; }

        /// <summary>
        /// 初始化下默认的搜索地址
        /// </summary>
        /// <param name="viewFinderManager">资源查找器</param>
        public ViewMediaTypeFormatter(IResourceFinderManager viewFinderManager)
        {
            viewFinderManager.CheckNullThrowArgumentNullException("viewFinderManager");
            //按照优先级排序下
            this._viewFinderManager = viewFinderManager;
            //请注意先后顺序
            this.ViewLocationFormats = new string[] { "~/Views/{0}.aspx", "~/Views/T.aspx" };
        }

        /// <summary>
        /// 返回待搜索列表
        /// </summary>
        /// <param name="requestContext">当前请求上下文</param>
        /// <returns>返回当前接口需要搜索的资源</returns>
        private IEnumerable<string> GetSearchedViewPath(RequestContext requestContext)
        {
            //用于保存搜索路径

            //接口名称
```

```
        string actionName = requestContext.RawRequestParams.ActionName;

        //合法的视图名称集合
        var viewNames = new string[] { actionName, actionName.Replace(".", "") };

        //循环下看下是否存在自定义的视图文件路径(优先级最高 1)
        IList<string> searchedViewPath = (from locationPath in this.ViewLocationFormats
            from viewName in viewNames
            select requestContext.HttpContext.Server.MapPath(locationPath.With(viewName))).ToList();

        //当前插件所在程序集 Views 文件夹查找（程序集自定义的内嵌资源文件优先级 2）
        if (!requestContext.ActionDescriptor.IsNull())
        {
            //当前接口所在程序集
            Assembly assembly = requestContext.ActionDescriptor.ActionType.Assembly;
            //检测所有视图命名规则是否存在
            foreach (var viewName in viewNames)
            {
                searchedViewPath.Add(ManifestResourceViewSearchPath.With(assembly.GetName().Name, viewName));
            }
        }

        //全局默认（兜底）
        searchedViewPath.Add(ManifestResourceViewSearchPath.With(Namespace, "T"));

        //返回所有的待搜索路径
        return searchedViewPath;
    }

    /// <summary>
    /// 根据上下文获取当前接口视图源代码
    /// </summary>
    /// <param name="requestContext">当前请求上下文</param>
    /// <returns>返回接口视图源文件；注意：在未找到对应实体源文件，会抛出异常</returns>
    private string GetViewSource(RequestContext requestContext)
    {
        //资源模板文件
        string viewSource = null;

        //获取待搜索列表
        var searchedViewPaths = this.GetSearchedViewPath(requestContext);

        //循环搜索列表
        foreach (var searchPath in searchedViewPaths)
        {
            //找到对应的模板源码，直接返回
            viewSource = this._viewFinderManager.GetResource(searchPath);
            if (!viewSource.IsNull())
            {
                break;
            }
            if (!viewSource.IsNull())
            {
                break;
            }
        }

        //都未找到，抛出异常
        if (viewSource.IsNull())
        {
            throw new ApiException(Resource.CoreResource.View_Not_Exists.With(NewLine, string.Join(NewLine,
searchedViewPaths.Distinct().ToArray())));
        }

        //返回源代码
        return viewSource;
    }

    /// <summary>
    /// 搜索视图，读取视图，然后执行视图，返回执行后的视图内容
    /// </summary>
    /// <param name="requestContext">当前请求上下文</param>
    /// <param name="actionResult">IAction 执行结果</param>
    /// <returns>返回格式化后的字符串</returns>
    public string SerializedActionResultToString(RequestContext requestContext, ActionResult actionResult)
    {
        requestContext.CheckNullThrowArgumentNullException("requestContext");
        requestContext.RawRequestParams.CheckNullThrowArgumentNullException("requestContext.RawRequestParams");

        //资源模板文件
        string viewSource = this.GetViewSource(requestContext);
        //找到视图文件，就使用视图输出
        ApiViewEngine viewEngine = new ApiViewEngine(Language.CSharp);
        var viewParameters = new ViewParameterCollection();
        //视图模板文件始终只会包含 RequestContext，ActionResult 这 2 个对象
        viewParameters.Add(new ViewParameter("RequestContext", requestContext));
        //在需要输出复杂类型的时候，所有的数据请挂靠在 ActionResult.Data 属性下，这样外部可以根据此对象来进行视图解析
        viewParameters.Add(new ViewParameter("ActionResult", actionResult));
        //编译视图文件并输出编译执行后的结果
        return viewEngine.CompileByViewSource(viewSource, viewParameters);
    }
```

```
    }
```

3.7 接口格式化后的输出（IResponse）

输出器其定义的方法很简单，接口只定义了一个方法，作用是将 IMediaTypeFormatter 接口格式化后的数据输出到客户端。具体的定义为：

```
    /// <summary>
    /// action 执行结果输出器
    /// </summary>
    public interface IResponse
    {
        /// <summary>
        /// 格式化输出 ActionResult 对象到客户端
        /// </summary>
        /// <param name="requestContext">当前请求上下文</param>
        /// <param name="responseFormat">actionResultString 格式化数据类型；XML/JSON</param>
        /// <param name="serializedActionResultString">执行结果对象</param>
        /// <returns></returns>
        void ResponseSerializedStringToClient(RequestContext requestContext, ResponseFormat responseFormat, string
serializedActionResultString);
    }
```

系统框架默认实现为 DefaultResponse，具体逻辑直接使用 requestContext.HttpContext.Response 方法将格式化后的字符串流输出给客户端，具体实现为：

```
    /// <summary>
    /// 字符串输出器默认实现
    /// </summary>
    internal class DefaultResponse : IResponse
    {
        /// <summary>
        /// 自定义一些输出头信息，实现类无需了解
        /// </summary>
        /// <param name="requestContext">当前请求上下文</param>
        private static void AddCustomerResponseHeaders(RequestContext requestContext)
        {
            try
            {
                ////输出一些头信息到客户端
                //requestContext.HttpContext.Response.Headers.Add("Api-Action-Name",
                //    requestContext.RawRequestParams.ActionName ?? string.Empty);

                ////版本
                //requestContext.HttpContext.Response.Headers.Add("Api-Version",
                //    requestContext.ActionDescriptor.IsNull() ? "" : requestContext.ActionDescriptor.Version);

                ////集群，分布式服务器编号
                //requestContext.HttpContext.Response.Headers.Add("Api-Server-Name", requestContext.SysOptions.ServerName);

                //把上下文保存的一些自定义打点数据输出到客户端头部，方便调试
                //foreach (var item in requestContext.AdditionDatas)
                //{
                //    requestContext.HttpContext.Response.Headers.Add(item.Key, (item.Value is DateTime) ?
((DateTime)item.Value).ToString("yyyy/MM/dd HH:mm:ss.ffffff") : item.Value.ToString());
                //}
            }
            finally { }
        }

        /// <summary>
        /// 输出格式化数据到客户端
        /// </summary>
        /// <param name="requestContext">当前请求上下文</param>
        /// <param name="responseFormat">输出格式化类型</param>
        /// <param name="serializedActionResultString">格式化了的 ActionResult 对象字符串</param>
        public void ResponseSerializedStringToClient(RequestContext requestContext, ResponseFormat responseFormat, string
serializedActionResultString)
        {
            //参数值为 null
            requestContext.CheckNullThrowArgumentNullException("requestContext");
            requestContext.HttpContext.CheckNullThrowArgumentNullException("requestContext.HttpContext");

            //添加一些自定义的 header 头信息
            AddCustomerResponseHeaders(requestContext);

            //字符串格式为 XML
            if (responseFormat == ResponseFormat.XML)
            {
                requestContext.HttpContext.Response.ContentType = "application/xml; charset=utf-8";
            }

            //字符串格式为 JSON
            if (responseFormat == ResponseFormat.JSON)
            {
                requestContext.HttpContext.Response.ContentType = "application/json; charset=utf-8";
            }
```

```
            //HTML
            if (responseFormat == ResponseFormat.VIEW)
            {
                requestContext.HttpContext.Response.ContentType = "text/html; charset=utf-8";
            }

            //输出格式化数据给客户端
            requestContext.HttpContext.Response.Write(serializedActionResultString ?? string.Empty);
        }
    }
```

3.8 值提供器/值提供器管理器（IValueProvider，IValueProvidersManager）

值提供器接口用于对绑定对象提供值，即：所有的对象绑定属性值会从值提供器里获取值来进行属性绑定。值提供器管理用于管理所有值提供器实现，提供了一个更加高层的抽象。注意：值提供器实现注册后，模式为协作模式，即：各个值提供器会根据其优先级来进行值提供。

值提供器的接口定义为：

```
/// <summary>
/// 值提供器；此接口属于协作接口，即：多个注册的实现，会依次根据值提供器优先级来进行获取值
/// </summary>
public interface IValueProvider
{
    /// <summary>
    /// 根据对象属性名称从值提供器里获取值(注意有可能会返回 null，不存在值)
    /// </summary>
    /// <param name="propertyName">键名称，一般对应于绑定对象的属性名称</param>
    /// <returns></returns>
    object GetValue(string propertyName);

    /// <summary>
    /// 获取到所有的键信息
    /// </summary>
    IEnumerable<string> GetAllKeys();

    /// <summary>
    /// 值提供器优先级
    /// </summary>
    int Order { get; }
}
```

值提供器管理器接口定义为：

```
/// <summary>
/// 值提供器管理器；此过滤器将会将所有注册的值提供器进行管理
/// </summary>
public interface IValueProvidersManager
{
    /// <summary>
    /// 值提供器集合
    /// </summary>
    IEnumerable<IValueProvider> ValueProviders { get; }

    /// <summary>
    /// 获取所有的键
    /// </summary>
    /// <returns></returns>
    IEnumerable<string> GetAllKeys();

    /// <summary>
    /// 根据键，后去键所对应的值
    /// </summary>
    /// <param name="propertyName">键名称</param>
    /// <returns>如果键不存在就直接返回 null</returns>
    object GetValue(string propertyName);
}
```

接口框架里默认实现的3种值提供器，分别为：FormValueProvider(基于web form input 的值提供其),QueryStringValueProvider(基于URL查询字符串的值提供其),ServerVariablesValueProvider(基于服务器环境变量)，具体实现为：

```
/// <summary>
/// 基于 web form input 的值提供其
/// </summary>
internal class FormValueProvider : IValueProvider
{
    /// <summary>
    ///
    /// </summary>
    private readonly IDictionary<string, object> _formValueDictionary =
        new Dictionary<string, object>(StringComparer.OrdinalIgnoreCase);

    /// <summary>
    /// Form 表单值提供器
```

```
        /// </summary>
        /// <param name="httpContext"></param>
        public FormValueProvider(HttpContextBase httpContext)
        {
            httpContext.CheckNullThrowArgumentNullException("httpContext");
            httpContext.Request.Unvalidated.Form.AllKeys.ToList().ForEach(key =>
            {
                if (!key.IsNullOrEmpty() && !_formValueDictionary.ContainsKey(key))
                {
                    this._formValueDictionary.Add(key, httpContext.Request.Unvalidated.Form[key]);
                }
            });
        }

        /// <summary>
        ///
        /// </summary>
        /// <returns></returns>
        public IEnumerable<string> GetAllKeys()
        {
            return this._formValueDictionary.Keys;
        }

        /// <summary>
        ///
        /// </summary>
        /// <param name="propertyName"></param>
        /// <returns></returns>
        public object GetValue(string propertyName)
        {
            return (from key in this._formValueDictionary.Keys
                    where key.Equals(propertyName, StringComparison.OrdinalIgnoreCase)
                    select this._formValueDictionary[key]).FirstOrDefault();
        }

        /// <summary>
        ///
        /// </summary>
        public int Order
        {
            get { return 1; }
        }
    }

    /// <summary>
    /// 基于 URL 查询字符串的值提供其
    /// </summary>
    internal class QueryStringValueProvider : IValueProvider
    {
        /// <summary>
        ///
        /// </summary>
        private readonly IDictionary<string, object> _queryStringDictionary =
            new Dictionary<string, object>(StringComparer.OrdinalIgnoreCase);

        /// <summary>
        /// URL 参数值提供器
        /// </summary>
        /// <param name="httpContext"></param>
        public QueryStringValueProvider(HttpContextBase httpContext)
        {
            httpContext.CheckNullThrowArgumentNullException("httpContext");
            httpContext.Request.Unvalidated.QueryString.AllKeys.ToList().ForEach(key =>
            {
                if (!key.IsNullOrEmpty() && !_queryStringDictionary.ContainsKey(key))
                {
                    this._queryStringDictionary.Add(key, httpContext.Request.Unvalidated.QueryString[key]);
                }
            });
        }

        /// <summary>
        ///
        /// </summary>
        /// <returns></returns>
        public IEnumerable<string> GetAllKeys()
        {
            return this._queryStringDictionary.Keys;
        }

        /// <summary>
        ///
        /// </summary>
        /// <param name="propertyName"></param>
        /// <returns></returns>
        public object GetValue(string propertyName)
        {
            return (from key in this._queryStringDictionary.Keys
                    where key.Equals(propertyName, StringComparison.OrdinalIgnoreCase)
                    select this._queryStringDictionary[key]).FirstOrDefault();
        }

        /// <summary>
```

```
        ///
        /// </summary>
        public int Order
        {
            get { return 0; }
        }
    }

    /// <summary>
    /// 服务器环境变量值提供器，定义的实体请将-去掉
    /// </summary>
    public class ServerVariablesValueProvider : IValueProvider
    {
        /// <summary>
        ///
        /// </summary>
        private readonly IDictionary<string, object> _dic = new Dictionary<string, object>();

        /// <summary>
        ///
        /// </summary>
        /// <param name="httpContext"></param>
        public ServerVariablesValueProvider(HttpContextBase httpContext)
        {
            foreach (var key in httpContext.Request.ServerVariables.AllKeys)
            {
                this._dic.Add(key.Replace("-", string.Empty), httpContext.Request.ServerVariables[key]);
            }
        }

        /// <summary>
        ///
        /// </summary>
        /// <returns></returns>
        public IEnumerable<string> GetAllKeys()
        {
            return this._dic.Keys;
        }

        /// <summary>
        ///
        /// </summary>
        /// <param name="propertyName"></param>
        /// <returns></returns>
        public object GetValue(string propertyName)
        {
            return (from key in this._dic.Keys where key.Equals(propertyName, StringComparison.OrdinalIgnoreCase) select
this._dic[key]).FirstOrDefault();
        }

        /// <summary>
        /// 注册的所有只提供都找不到对应的键值，就获取服务器环境变量
        /// </summary>
        public int Order
        {
            get { return int.MinValue; }
        }
    }
```

值提供器管理器框架默认实现为来管理已经注册的所有值提供器，用于提供更高层方法，让调用则无需知道系统框架提供了那些值提供器，也方便值提供器的定义管理，其具体实现如下：

```
    /// <summary>
    /// 默认的值提供器管理器
    /// </summary>
    public class DefaultValueProvidersManager : IValueProvidersManager
    {
        /// <summary>
        /// 保存值提供器集合
        /// </summary>
        private readonly IList<IValueProvider> _valueProviders = new List<IValueProvider>();

        /// <summary>
        /// 使用值提供器集合来构造一个值提供器管理器
        /// </summary>
        /// <param name="valueProviders">值提供器集合</param>
        public DefaultValueProvidersManager(IValueProvider[] valueProviders)
        {
            valueProviders.CheckNullThrowArgumentNullException("valueProviders");
            //安装值提供器优先级（在多个值提供器提供相同键的情况下，优先级高的先取出，后续的值提供器将不会取值）
            foreach (var item in valueProviders.OrderByDescending(o => o.Order))
            {
                this._valueProviders.Add(item);
            }
        }

        /// <summary>
        /// 获取所有注册的值提供器
        /// </summary>
        public IEnumerable<IValueProvider> ValueProviders
        {
```

```
        get { return this._valueProviders; }
    }

    /// <summary>
    /// 获取所有值提供器的键
    /// </summary>
    /// <returns>返回邮件值提供器的键集合</returns>
    public IEnumerable<string> GetAllKeys()
    {
        return this._valueProviders.SelectMany(item => item.GetAllKeys()).ToList();
    }

    /// <summary>
    /// 根据键获取值提供器对应的值
    /// </summary>
    /// <param name="propertyName">键名称</param>
    /// <returns></returns>
    public object GetValue(string propertyName)
    {
        return this._valueProviders.Select(item => item.GetValue(propertyName)).FirstOrDefault(value => !value.IsNull());
    }

}
```

3.9 资源查找器/资源查找器管理器(IResourceFinder，IResourceFinderManager)

资源查找器为系统框架提供资源的组件，比如：视图缓存，模块说明文件，模块图标（会转换成base64字符串存储）。其他服务类可以方便的获取资源。资源查找器管理器提供了所有资源查找器更加高层的抽象接口。方便调用。

资源查找器节点定义如下：

```
/// <summary>
/// 所有接口所在程序集视图文件查找器（注意此接口只找文本类型的，比如：js,css,aspx,asp,cshtml 等）
/// 此接口属于协作接口，即：注册多个资源查找器系统会依次在各个查找器里进行资源查找
/// </summary>
public interface IResourceFinder
{
    /// <summary>
    /// 优先级排序，数字越大，优先级越高
    /// </summary>
    int Order { get; }

    /// <summary>
    /// 返回所有程序集内嵌资源信息；此方法的实现最好能够进行缓存机制，即第一次加载到时候描述所有程序集，后续直接从缓存里读取
    /// key:资源文件名称
    /// value:资源文件源代码
    /// </summary>
    /// <returns></returns>
    IDictionary<string, string> GetResources();

    /// <summary>
    /// 获取到资源文件原始文本
    /// </summary>
    /// <param name="resourceViewFullPath">内嵌资源路径</param>
    /// <returns>内嵌资源原始文件(找不的将返回 null，所以调用的时候需要注意下 null 情况)</returns>
    string GetResource(string resourceViewFullPath);
}
```

资源查找器管理器接口定义如下：

```
    /// <summary>
    /// 资源查找过滤器
    /// </summary>
    public interface IResourceFinderManager
    {
        /// <summary>
        /// 系统框架注册的所有资源查找器
        /// </summary>
        IEnumerable<IResourceFinder> ResourceFinders { get; }

        /// <summary>
        /// 获取资源
        /// </summary>
        /// <param name="resourceName">资源名称（部分路径或者全部路径，如：sys.png 或者 Frxs.ServiceCenter.Api.Core.Resource.sys.png）
</param>
        /// <returns></returns>
        string GetResource(string resourceName);
    }
```

同样，系统框架默认定义2个资源查找器，分别为：AssemblyResourceFinder(模块内嵌视图查找器),LocalFileViewResourceFinder(本地文件视图查找器)

具体实现如下：

```
    /// <summary>
    /// 资源文件读取器
```

```
    /// </summary>
    public abstract class ResourceFinderBase : IResourceFinder
    {
        /// <summary>
        /// 读取内嵌资源，指定扩展名；系统框架默认读取：.aspx，.t ,系统框架不区分后缀大小写
        /// 为了安全调用，此属性只能在重写类里重写（框架采取约定的方式）
        /// </summary>
        protected virtual string[] SupportedFileExtensions
        {
            get
            {
                return new string[] { ".aspx", ".asp", ".ascx", ".master", ".js", ".css", ".cshtml", "vbhtml", ".html", ".htm",
".shtml", ".txt", ".xml" };
            }
        }

        /// <summary>
        /// 获取全部资源
        /// </summary>
        /// <returns>获取所有文本资源信息</returns>
        public abstract IDictionary<string, string> GetResources();

        /// <summary>
        /// 获取文本类型资源文件源码
        /// </summary>
        /// <param name="resourceFullPath">针对内嵌资源或者本地资源路径</param>
        /// <returns></returns>
        public string GetResource(string resourceFullPath)
        {
            //当前接口视图文件忽略大小写
            var viewResource = this.GetResources().FirstOrDefault(o => o.Key.Equals(resourceFullPath,
StringComparison.OrdinalIgnoreCase));

            //不存在直接进行下一个
            if (!viewResource.IsNull() && !viewResource.Key.IsNullOrEmpty())
            {
                return viewResource.Value;
            }

            //获取空
            return null;
        }

        /// <summary>
        /// 优先级；默认 int.Minvalue，最低
        /// </summary>
        public virtual int Order
        {
            get { return int.MinValue; }
        }
    }

    /// <summary>
    /// 模块内嵌视图查找器
    /// </summary>
    public class AssemblyResourceFinder : ResourceFinderBase, IStartUp
    {
        /// <summary>
        /// 用于缓存所有程序集内嵌资源文件信息
        /// key:视图路径，value:视图源代码
        /// </summary>
        private static readonly Dictionary<string, string> CachedManifestResourceNames = new Dictionary<string,
string>(StringComparer.OrdinalIgnoreCase);
        private static bool _initializationed = false;
        private static readonly object Locker = new object();
        private readonly IActionSelector _actionSelector;
        private ICacheManager _cacheManager;

        /// <summary>
        /// 默认的内嵌视图查找器
        /// </summary>
        /// <param name="actionSelector">接口查找器</param>
        /// <param name="cacheManager">缓存器</param>
        public AssemblyResourceFinder(IActionSelector actionSelector, ICacheManager cacheManager)
        {
            actionSelector.CheckNullThrowArgumentNullException("actionSelector");
            cacheManager.CheckNullThrowArgumentNullException("cacheManager");
            this._actionSelector = actionSelector;
            this._cacheManager = cacheManager;
        }

        /// <summary>
        /// 内嵌图片扩展名
        /// </summary>
        private readonly string[] _supportedLogoImageExtensions = new string[] { ".jpg", ".png", ".gif" };

        /// <summary>
        /// 允许读取内嵌的图片信息
        /// </summary>
        protected override string[] SupportedFileExtensions
        {
```

```
        get
        {
            return base.SupportedFileExtensions.Concat(this._supportedLogoImageExtensions).ToArray();
        }
    }

    /// <summary>
    /// 获取所有程序集的内嵌视图文件；惰性加载，第一次获取的时候加载
    /// </summary>
    /// <returns></returns>
    public override IDictionary<string, string> GetResources()
    {
        //还未初始化
        if (_initializationed)
        {
            return CachedManifestResourceNames;
        }
        lock (Locker)
        {
            if (_initializationed)
            {
                return CachedManifestResourceNames;
            }

            //已经初始化了标志
            _initializationed = true;

            //获取所有接口信息
            var actions = this._actionSelector.GetActionDescriptors();

            //获取所有程序集(搜索所有的接口并且使用接口所在程序集来进行归组)
            var assemblys = (from action in actions group action by action.ActionType.Assembly into g select g.Key).ToList();

            //连接下插件所在程序集
            assemblys = assemblys.Concat(ApiPluginManager.GetApiPlugins().Select(o => o.GetType().Assembly)).ToList();

            //筛选排除重复
            assemblys = (assemblys.GroupBy(assembly => assembly).Select(g => g.Key)).OrderBy(o => o.FullName).ToList();

            //循环接口所有程序集
            foreach (var assembly in assemblys)
            {
                //获取程序集内嵌资源名称
                var resourceNames = assembly.GetManifestResourceNames();

                //循环内嵌资源
                foreach (var resourceName in resourceNames)
                {
                    //已经存在缓存
                    if (CachedManifestResourceNames.ContainsKey(resourceName))
                    {
                        continue;
                    }

                    //循环指定的扩展名，不在扩展名之中的，不加载到视图缓存集合
                    foreach (var fileExtension in this.SupportedFileExtensions)
                    {
                        //视图文件必须以指定扩展名结尾
                        if (!resourceName.EndsWith(fileExtension, StringComparison.OrdinalIgnoreCase))
                        {
                            continue;
                        }

                        //读取接口所在程序集内嵌资源文件
                        var resourceStream = assembly.GetManifestResourceStream(resourceName);

                        //读取的资源项目是否为 null
                        if (resourceStream.IsNull())
                        {
                            continue;
                        }

                        //图片资源(图片都是一些小图片)
                        if (this._supportedLogoImageExtensions.Contains(fileExtension, StringComparer.OrdinalIgnoreCase))
                        {
                            var resourceStreamBytes = new byte[resourceStream.Length];
                            resourceStream.Read(resourceStreamBytes, 0, (int)resourceStream.Length);
                            //转换成 base64 保存
                            var viewSource = Convert.ToBase64String(resourceStreamBytes);
                            //将原文件添加到缓存
                            CachedManifestResourceNames.Add(resourceName, viewSource);
                            resourceStream.Close();
                        }
                        else
                        {
                            //读取内嵌资源文件
                            using (var streamReader = new StreamReader(resourceStream))
                            {
                                //获取全部源文件
                                var viewSource = streamReader.ReadToEnd();
                                //将原文件添加到缓存
```

```
                            CachedManifestResourceNames.Add(resourceName, viewSource);
                        }
                    }
                }
            }
        }

        //返回所有视图
        return CachedManifestResourceNames;
    }

    /// <summary>
    /// 预热下，系统启动时候执行一次
    /// </summary>
    void IStartUp.Init()
    {
        this.GetResources();
    }
}

/// <summary>
/// 本地文件视图查找器
/// </summary>
public class LocalFileViewResourceFinder : ResourceFinderBase, IStartUp
{
    /// <summary>
    /// 用于缓存所有系统框架的文本资源
    /// </summary>
    private static readonly Dictionary<string, string> CachedeLocalResourceNames = new Dictionary<string,
string>(StringComparer.OrdinalIgnoreCase);
    private static readonly ReaderWriterLockSlim Locker = new ReaderWriterLockSlim();
    private static bool _initializationed = false;
    private readonly HttpContextBase _httpContext;
    private ICacheManager _cacheManager;

    /// <summary>
    /// 视图文件保存的本地文件夹
    /// </summary>
    private const string ViewDirectory = "~/Views";

    /// <summary>
    /// 本地文件资源查找器
    /// </summary>
    /// <param name="httpContext">当前 http 请求上下文</param>
    /// <param name="cacheManager">缓存接口</param>
    public LocalFileViewResourceFinder(HttpContextBase httpContext, ICacheManager cacheManager)
    {
        this._httpContext = httpContext;
        this._cacheManager = cacheManager;
    }

    /// <summary>
    /// 获取所有查找器
    /// </summary>
    /// <returns></returns>
    public override IDictionary<string, string> GetResources()
    {
        //已经初始化了
        if (_initializationed)
        {
            return CachedeLocalResourceNames;
        }

        using (new WriteLockDisposable(Locker))
        {
            //初始化了
            if (_initializationed)
            {
                return CachedeLocalResourceNames;
            }

            _initializationed = true;

            //获取物理路径
            string physicalPath = this._httpContext.Server.MapPath(ViewDirectory);

            //获取所有文件夹下面的文件
            if (Directory.Exists(physicalPath))
            {
                //找出合法的后缀文件
                var fiels = Directory.GetFiles(physicalPath, "*", SearchOption.AllDirectories)
                    .Where(fileName => this.SupportedFileExtensions.Any(ex => ex.Equals(Path.GetExtension(fileName),
StringComparison.OrdinalIgnoreCase))).ToList();

                //循环读取本地资源
                foreach (var file in fiels)
                {
                    //缓存里已经存在指定文件
                    if (CachedeLocalResourceNames.ContainsKey(file))
                    {
                        continue;
```

```
                }
                //添加到缓存
                using (StreamReader streamReader = new StreamReader(file))
                {
                    CachedeLocalResourceNames.Add(file, streamReader.ReadToEnd());
                }
            }
        }
    }

    //返回缓存本地资源文件
    return CachedeLocalResourceNames;
}

/// <summary>
/// 优先级(重写下是为了系统自定义文件夹配置高于程序集)
/// </summary>
public override int Order
{
    get
    {
        return 0;
    }
}

/// <summary>
/// 预热下，系统启动时候执行一次
/// </summary>
void IStartUp.Init()
    {
        this.GetResources();
    }
}
```

从具体的实现可以看出，不管是你续集内嵌资源查找器，还是本地文件资源查找器，同时还都实现了 IStartUp 接口，此接口会在框架启动的时候，自动执行，也就是说，本地或者内嵌资源查找器都会在程序框架启动的那一刻，执行一次资源扫描，然后将所有资源加载到内存，这样后续访问就会直接走内存，而不需要一次次的扫描执行。另外，本地资源查找器实现采取了约定的模式，即：只会查找宿主Views目录下的所有指定的文本文件，且本地资源查找器的优先级高于程序集内嵌资源查找器，这样才能在扩展的时候覆盖内嵌默认实现。

3.10 上送参数绑定器(IRequestParamsBinder)

上送参数绑定器接口，即：将通过 webform或者url方式提交的参数绑定到接受对象 RequestParams 上。其接口定义为：

```
/// <summary>
/// 上送参数对象绑定器接口
/// </summary>
public interface IRequestParamsBinder
{
    /// <summary>
    /// 获取上送的参数对象
    /// </summary>
    /// <returns></returns>
    RequestParams GetRequestParams();
}
```

同样系统框架默认定义了一个具体实现 DefaultRequestParamsBinder ，其实现代码如下：

```
/// <summary>
/// 上送参数对象绑定器
/// </summary>
internal class DefaultRequestParamsBinder : IRequestParamsBinder
{
    /// <summary>
    ///
    /// </summary>
    readonly IValueProvidersManager _valueProviderManager;

    /// <summary>
    ///
    /// </summary>
    /// <param name="valueProviderManager">值提供器管理器</param>
    public DefaultRequestParamsBinder(IValueProvidersManager valueProviderManager)
    {
        this._valueProviderManager = valueProviderManager;
    }

    /// <summary>
    /// 获取接口框架上送参数对象
    /// </summary>
    /// <returns></returns>
    public RequestParams GetRequestParams()
    {
        return new DefaultModelBinder().Bind<RequestParams>(this._valueProviderManager);
    }
}
```

从实现可以看出，默认的绑定器依赖于值提供器管理器，直接从值提供器里获取属性值来进行绑定，内部调用了默认的

DefaultModelBinder 对象绑定器来进行绑定，对象绑定器的实现代码如下：

```
/// <summary>
/// 默认的对象绑定器； 针对简易对象(属性不包含复杂对象)
/// </summary>
internal class DefaultModelBinder : IModelBinder
{
    /// <summary>
    /// 根据值提供其，绑定对象，自动给参数赋值，使用反射方式
    /// </summary>
    /// <param name="valueProvidersManager">值提供器管理器</param>
    /// <typeparam name="T">待绑定的类型必须要有无参构造函数</typeparam>
    /// <returns></returns>
    public T Bind<T>(IValueProvidersManager valueProvidersManager)
    {
        //参数不能为 null
        valueProvidersManager.CheckNullThrowArgumentNullException("valueProvidersManager");

        //创建对象，并且对属性赋值（如果属性值在值提供器里存在并且可以转型成功）
        T obj = Activator.CreateInstance<T>();

        //获取对象属性
        var propertys = obj.GetType().GetPropertiesInfo();

        //循环属性从值提供器里进行赋值
        foreach (var p in propertys)
        {
            //从对象里获取数据
            var value = valueProvidersManager.GetValue(p.Name);

            //属性是可写并且不是索引方法的并且数据不为 null
            if (!p.CanWrite || p.GetIndexParameters().Length > 0 || value.IsNull())
            {
                continue;
            }
            try
            {
                p.SetValue(obj, Convert.ChangeType(value, p.PropertyType), null);
            }
            catch (Exception)
            {
                // ignored
            }
        }
        return obj;
    }
}
```

绑定的逻辑很简单，就是先通过反射的方式获取到参数对象 RequestParams 的所有公开且可写的属性,，然后通过属性名称从值提供器管理器里取值，取到值就进行赋值绑定。最后我们再开看一下 RequestParams 对象定义：

```
/// <summary>
/// 接口客户端提交的核心参数包
/// </summary>
[Serializable]
public class RequestParams
{
    /// <summary>
    /// 客户端 ID：比如：20009 等
    /// </summary>
    public string AppKey { get; set; }

    /// <summary>
    /// 从请求的信息里获取到请求的接口名称，比如：API.Help
    /// </summary>
    public string ActionName { get; set; }

    /// <summary>
    /// 客户端指定接口服务器返回数据的格式化方式 XML/JSON/VIEW
    /// </summary>
    public string Format { get; set; }

    /// <summary>
    /// 上传的 JSON 数据；就算是不需上送参数，也需要上送"{}"字符串
    /// </summary>
    public string Data { get; set; }

    /// <summary>
    /// 上传时间戳（服务器与服务端到时候进行时间比对）;格式为：yyyy/MM/dd HH:mm:ss
    /// 应用场景：比如，一个接口如果不加这个时间戳的时候，只要有人截获了提交参数以及知道了 URL
    /// 那么截获访问消息的人，完全可以重复提交接口数据，这尤其在针对数据操作的时候，影响比较大，
    /// 因此加上次时间戳，让调用客户端上送客户端时间，然后服务器比对时间戳与服务器时间，
    /// 如果相差时间间隔比较大（比如 1 分钟），那么不允许提交
    /// </summary>
    public string TimeStamp { get; set; }

    /// <summary>
    /// 接口版本（在有多个接口名称一致的情况下；可以根据指定接口版本来选择特定的版本接口）
    /// </summary>
```

```
        public string Version { get; set; }

        /// <summary>
        /// 客户端数据签名（具体的数据签名方式需要在实际业务场景里约定）
        /// </summary>
        public string Sign { get; set; }

        ///// <summary>
        ///// 重写下 2 此请求参数对象是否相同
        ///// </summary>
        ///// <param name="lRequestParams"></param>
        ///// <param name="rRequestParams"></param>
        ///// <returns></returns>
        //public static bool operator ==(RequestParams lRequestParams, RequestParams rRequestParams)
        //{
        //    return lRequestParams.Equals(rRequestParams);
        //}

        ///// <summary>
        ///// 重写下 2 此请求参数对象是否不同
        ///// </summary>
        ///// <param name="lRequestParams"></param>
        ///// <param name="rRequestParams"></param>
        ///// <returns></returns>
        //public static bool operator !=(RequestParams lRequestParams, RequestParams rRequestParams)
        //{
        //    return !(lRequestParams == rRequestParams);
        //}

        ///// <summary>
        ///// 
        ///// </summary>
        ///// <param name="other"></param>
        ///// <returns></returns>
        //protected bool Equals(RequestParams other)
        //{
        //    return string.Equals(AppKey, other.AppKey) && string.Equals(ActionName, other.ActionName) &&
        //           string.Equals(Format, other.Format) && string.Equals(Data, other.Data) &&
        //           string.Equals(TimeStamp, other.TimeStamp) && string.Equals(Version, other.Version) &&
        //           string.Equals(Sign, other.Sign);
        //}

        ///// <summary>
        ///// 
        ///// </summary>
        ///// <param name="obj"></param>
        ///// <returns></returns>
        //public override bool Equals(object obj)
        //{
        //    if (ReferenceEquals(null, obj)) return false;
        //    if (ReferenceEquals(this, obj)) return true;
        //    if (obj.GetType() != this.GetType()) return false;
        //    return Equals((RequestParams)obj);
        //}

        ///// <summary>
        ///// 
        ///// </summary>
        ///// <returns></returns>
        //public override int GetHashCode()
        //{
        //    unchecked
        //    {
        //        var hashCode = (AppKey != null ? AppKey.GetHashCode() : 0);
        //        hashCode = (hashCode * 397) ^ (ActionName != null ? ActionName.GetHashCode() : 0);
        //        hashCode = (hashCode * 397) ^ (Format != null ? Format.GetHashCode() : 0);
        //        hashCode = (hashCode * 397) ^ (Data != null ? Data.GetHashCode() : 0);
        //        hashCode = (hashCode * 397) ^ (TimeStamp != null ? TimeStamp.GetHashCode() : 0);
        //        hashCode = (hashCode * 397) ^ (Version != null ? Version.GetHashCode() : 0);
        //        hashCode = (hashCode * 397) ^ (Sign != null ? Sign.GetHashCode() : 0);
        //        return hashCode;
        //    }
        //}
    }
```

3.11 上送业务参数绑定器(IRequestDtoBinder)

上送业务参数绑定器主要赋值将 RequestParams 对象里的Data属性进行反序列化绑定到RequestDto对象，Data属性为具体的业务参数，格式为一个合法的JSON对象。绑定成功后的RequestDto对象，相当于具体接口的一个入参。
其接口定义如下：

```
    /// <summary>
    /// RequestDto 参数获取绑定器
    /// </summary>
    public interface IRequestDtoBinder
    {
        /// <summary>
        /// 绑定获取上送参数(强类型)
        /// </summary>
        /// <param name="requestContext">当前请求上下文</param>
```

```
        /// <param name="actionDescriptor">接口描述信息</param>
        /// <returns>返回上送参数 data 对应的 RequestDto 对象</returns>
        TRequestDto Bind<TRequestDto>(RequestContext requestContext, IActionDescriptor actionDescriptor) where TRequestDto :
IRequestDto;

        /// <summary>
        /// 绑定获取上送参数(弱类型)
        /// </summary>
        /// <param name="requestContext">当前请求上下文</param>
        /// <param name="actionDescriptor">接口描述信息</param>
        /// <returns>返回上送参数 data 对应的 RequestDto 对象</returns>
        object Bind(RequestContext requestContext, IActionDescriptor actionDescriptor);
    }
```

系统框架同样也默认实现了此接口，具体实现类为：DefaultRequestDtoBinder ，我们可以先看下具体实现：

```
    /// <summary>
    /// 默认的上送参数绑定器
    /// </summary>
    internal class DefaultRequestDtoBinder : IRequestDtoBinder
    {
        /// <summary>
        /// 日志记录器
        /// </summary>
        public ILogger Logger { get; set; }

        /// <summary>
        /// 默认的上送参数绑定器
        /// </summary>
        public DefaultRequestDtoBinder()
        {
            this.Logger = NullLogger.Instance;
        }

        /// <summary>
        /// 绑定获取上送参数对象
        /// </summary>
        /// <param name="requestContext">当前请求上下文</param>
        /// <param name="actionDescriptor">接口描述信息</param>
        /// <returns>获取成功就返回上送的对象</returns>
        public TRequestDto Bind<TRequestDto>(RequestContext requestContext, IActionDescriptor actionDescriptor) where
TRequestDto : IRequestDto
        {
            return (TRequestDto)this.Bind(requestContext, actionDescriptor);
        }

        /// <summary>
        /// 绑定上送参数与实体对象
        /// </summary>
        /// <param name="requestContext">当前请求上下文</param>
        /// <param name="actionDescriptor">接口描述信息</param>
        /// <returns>获取成功就返回上送的对象(弱类型)</returns>
        public object Bind(RequestContext requestContext, IActionDescriptor actionDescriptor)
        {
            //不为 null
            requestContext.CheckNullThrowArgumentNullException("requestContext");
            actionDescriptor.CheckNullThrowArgumentNullException("requestDtoType");

            //必须继承自 RequestDtoBase
            if (!actionDescriptor.RequestDtoType.IsAssignableToIRequestDto())
            {
                throw new ApiException("requestDtoType 类型必须继承自 RequestDtoBase 或者实现 IRequestDto");
            }

            try
            {
                //DATA 数据包为空，默认的给一个 JSON 串
                if (requestContext.DecryptedRequestParams.Data.IsNullOrEmpty())
                {
                    requestContext.DecryptedRequestParams.Data = "{}";
                }

                //反序列化上送的 DATA 数据(使用解密后的 Data 数据)
                var requestDto =
requestContext.DecryptedRequestParams.Data.DeserializeJsonStringToObject(actionDescriptor.RequestDtoType);

                //将当前上下文请求对象赋值
                requestContext.RequestDto = requestDto;

                //返回上送数据对象
                return requestDto;
            }
            catch (Exception ex)
            {
                //设置错误消息
                //string errorMessage =
Resource.CoreResource.ActionBase_GetRequestDto_DeserializeObject_Error.With(this.GetType().FullName);

                //记录下日志
                this.Logger.Error(ex, requestContext.DecryptedRequestParams.Data ?? string.Empty);
```

```
                //出现异常直接返回错误
                return null;
            }
        }
    }
```

3.12 上送业务参数校验器(IRequestDtoValidator，IRequestDtoValidatable，IRequiredUserIdAndUserName)

当我们通过IRequestDtoBinder接口将上送业务参数绑定到具体的RequestDto后，系统框架会自动对RequestDto对象数据的合法性进行一次校验。为了不重复手工鞋相同的校验逻辑，比如：字符串的长度，字符串必须是手机号码等等。校验的方式分2种情况,第一:如果定义的RequestDto类型继承了IRequestDtoValidatable接口，可以手工的方式来进行参数校验，第二种：RequestDto类型，附加了实现 ValidationAttribute 属性的特性类，系统框架支持命名空间System.ComponentModel.DataAnnotations下所有的特性校验类。
下面我们先来看一下 IRequestDtoValidatable 接口定义：

```
    /// <summary>
    /// 校验 DTO 参数是否合法（业务合法），如果上送的参数对象实现了此接口，系统框架会自动进行校验
    /// 使用原则：如果是通用的数据校验比如，字符串长度，最小，最大值等，可以使用特性校验 System.ComponentModel.DataAnnotations 下的特性标签;
    /// 此接口在复杂业务数据正确性校验下使用
    /// </summary>
    public interface IRequestDtoValidatable
    {
        /// <summary>
        /// 此方法也许在执行 Valid（）方法先，执行下数据的处理，比如设置默认值操作等
        /// </summary>
        void BeforeValid();

        /// <summary>
        /// 验证方法;如果验证不通过返回错误集合，如果验证通过了，返回一个空的集合
        /// </summary>
        /// <returns>返回校验后的错误集合;如果通过验证，则返回一个空的列表集合</returns>
        IEnumerable<RequestDtoValidatorResultError> Valid();
    }
```

接口定义了2个方法，BeforeValid()方法，在Valid()方法执行前执行，旨在校验前，让开发人员有机会再次修改参数。具体的使用，我们在后续的接口开发实例章节再说。

不管是通过定义特性的方式，还是通过实现接口是方式来进行校验，最后校验都统一到RequestDtoValidator类来实现，我们先看下具体实现：

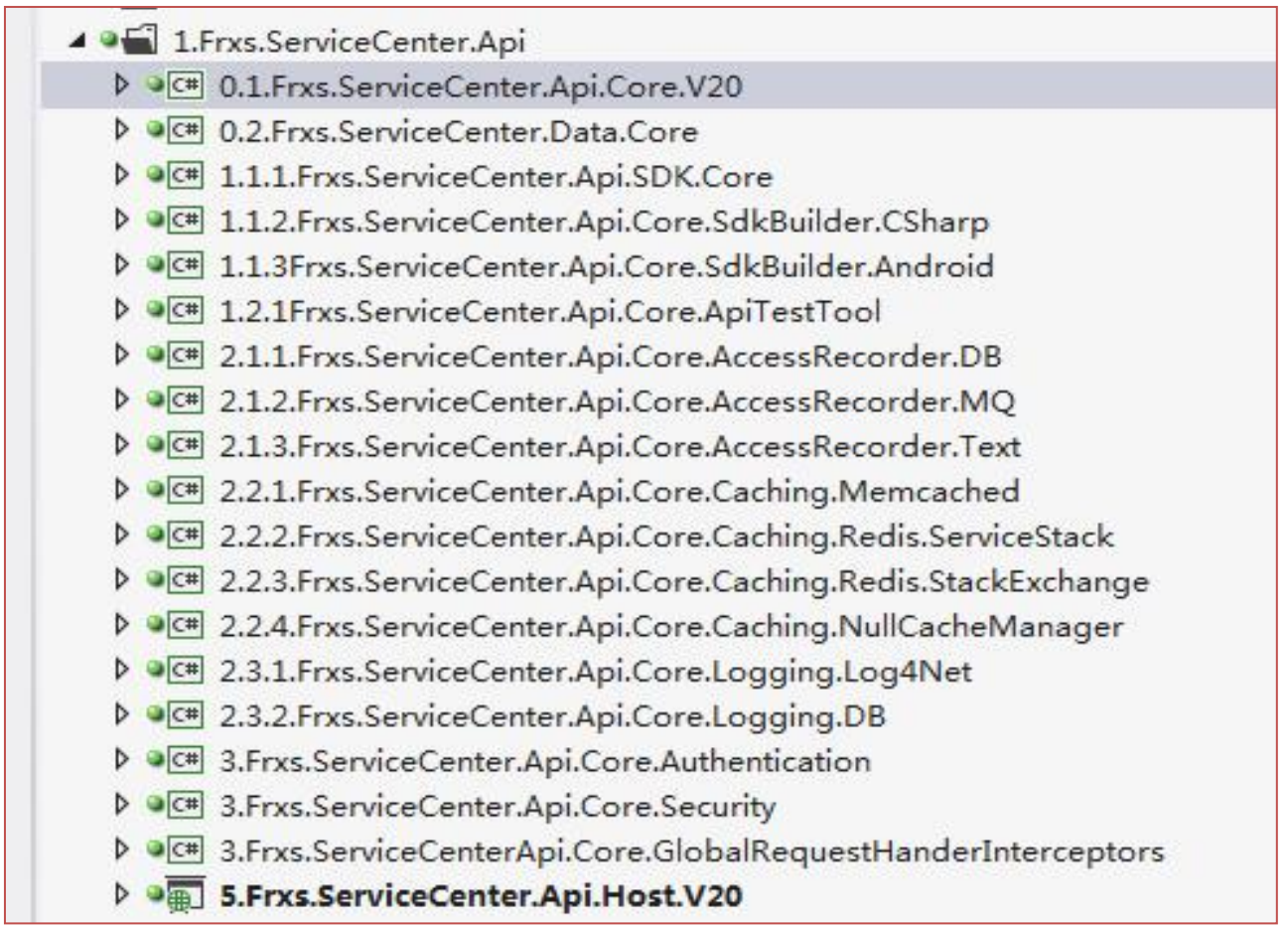
```
    /// <summary>
    /// 如果一个 RequestDto 实现了 IRequestDtoValidatable 接口，将可以使用此验证器来验证业务数据的准确性
    /// 特性验证器完全兼容：System.ComponentModel.DataAnnotations 命名空间特性验证(特性生效一定需要
    /// 上送参数类实现 IRequestDtoValidatable 接口，否则就算定义了校验特性，也不会生效)
    /// </summary>
    public class RequestDtoValidator
    {
        /// <summary>
        /// requestDto 参数上送对象
        /// </summary>
        /// <param name="requestDto">requestDto 参数上送对象</param>
        /// <param name="actionDescriptor">接口描述对象</param>
        public RequestDtoValidator(RequestDtoBase requestDto, ActionDescriptor actionDescriptor)
        {
            this.RequestDto = requestDto;
            this.ActionDescriptor = actionDescriptor;
            this.ValidatableObject = requestDto as IRequestDtoValidatable;
        }

        /// <summary>
        /// 当前输入的 RequestDto 对象
        /// </summary>
        public RequestDtoBase RequestDto { get; private set; }

        /// <summary>
        /// 接口描述对象
        /// </summary>
        public ActionDescriptor ActionDescriptor { get; private set; }

        /// <summary>
        /// 转型后的 RequestDto 对象，有可能为 null
        /// </summary>
        public IRequestDtoValidatable ValidatableObject { get; private set; }

        /// <summary>
        /// 验证实体数据正确性，返回 RequestDtoValidatorResult 对象
        /// </summary>
        public RequestDtoValidatorResult Valid()
        {
            //如果上送对象实现了 IRequestDtoValidatable 或者 IRequiredUserIdAndUserName 就进行校验
            if (this.ValidatableObject.IsNull() && !(this.RequestDto is IRequiredUserIdAndUserName))
            {
                return new RequestDtoValidatorResult();
```

```
            }

            //用于保存验证集合
            var validationResultErrors = new List<RequestDtoValidatorResultError>();

            //校验器给实现类一个改变参数属性值的机会
            this.ValidatableObject.BeforeValid();

            //1.校验定义在参数对象上的特性校验
            var validationContext = new ValidationContext(this.RequestDto);
            var results = new List<ValidationResult>();
            if (!Validator.TryValidateObject(this.RequestDto, validationContext, results, true))
            {
                results.ForEach(o =>
                {
                    validationResultErrors.Add(new RequestDtoValidatorResultError(string.Join(",", o.MemberNames.ToArray()),
o.ErrorMessage));
                });
            }

            //没有通过校验直接返回(防止在业务参数校验的时候，直接取值造成异常抛出)
            if (!validationResultErrors.IsEmpty())
            {
                return new RequestDtoValidatorResult(validationResultErrors);
            }

            //定义了自定义验证接口
            if (!this.ValidatableObject.IsNull())
            {
                //2.进行自定义的数据校验器
                var validationResults = this.ValidatableObject.Valid();

                //校验自定义业务数据是否正确
                validationResultErrors.AddRange(validationResults);
            }

            //3.校验用户名和用户 ID 或者在接口定义了需要校验的特性
            if (this.RequestDto is IRequiredUserIdAndUserName || this.ActionDescriptor.RequiredUserIdAndUserName)
            {
                //验证用户 ID 和用户名称的委托为空弹出消息
                if (SystemOptionsManager.Current.ValidUserIdAndUserNameFun.IsNull())
                {
                    validationResultErrors.Add(new RequestDtoValidatorResultError("ValidUserIdAndUserNameFun",
Resource.CoreResource.ActionBase_ValidUserIdAndUserNameFun_Null_Error));
                }
                //不为空就开始举行校验
                else
                {
                    //检测用户 ID 和用户名称是否提交
                    if (!SystemOptionsManager.Current.ValidUserIdAndUserNameFun(this.RequestDto))
                    {
                        validationResultErrors.Add(new RequestDtoValidatorResultError("UserId,UserName",
Resource.CoreResource.ActionBase_RequiredUserIdAndUserName_Error));
                    }
                }
            }

            //返回校验错误集合
            return new RequestDtoValidatorResult(validationResultErrors);
        }

    }
```

校验的业务逻辑分3块，首先会检测RequestDto对象是否实现了IRequestDtoValidatable 接口，如果实现了，则调用IRequestDtoValidatable.BeforeValid() 方法，更改下对象值。然后校验特性校验，如果特性校验不通过，直接返回校验结果类：RequestDtoValidatorResult ， 如果特性校验通过，再检测RequestDto是否实现了 IRequestDtoValidatable 接口，实现了的话就调用接口 IRequestDtoValidatable.Valid() 方法进行校验， 最后如果RequestDto实现了IRequiredUserIdAndUserName 接口就进行用户校验，注意，接口IRequiredUserIdAndUserName 仅仅是个空接口，如果具体对外接口需要校验上送的用户ID和用户名称，直接实现此接口即可，具体的校验逻辑委托给了外部配置文件，外部若未配置，系统框架内部设计了一个默认的校验逻辑,即：用户ID>0且用户名称不能为空。

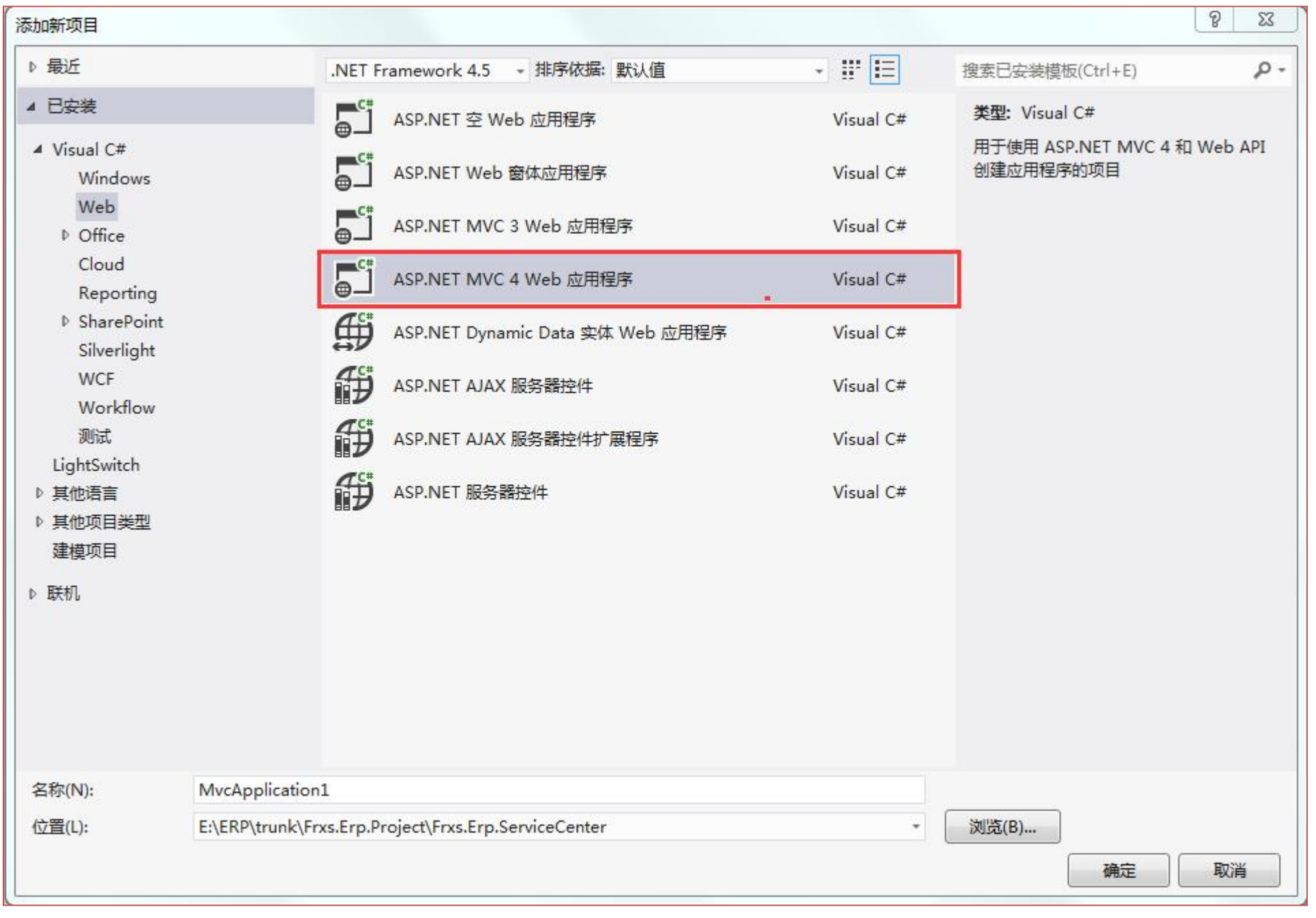
4.开始我们第一个Action接口之旅

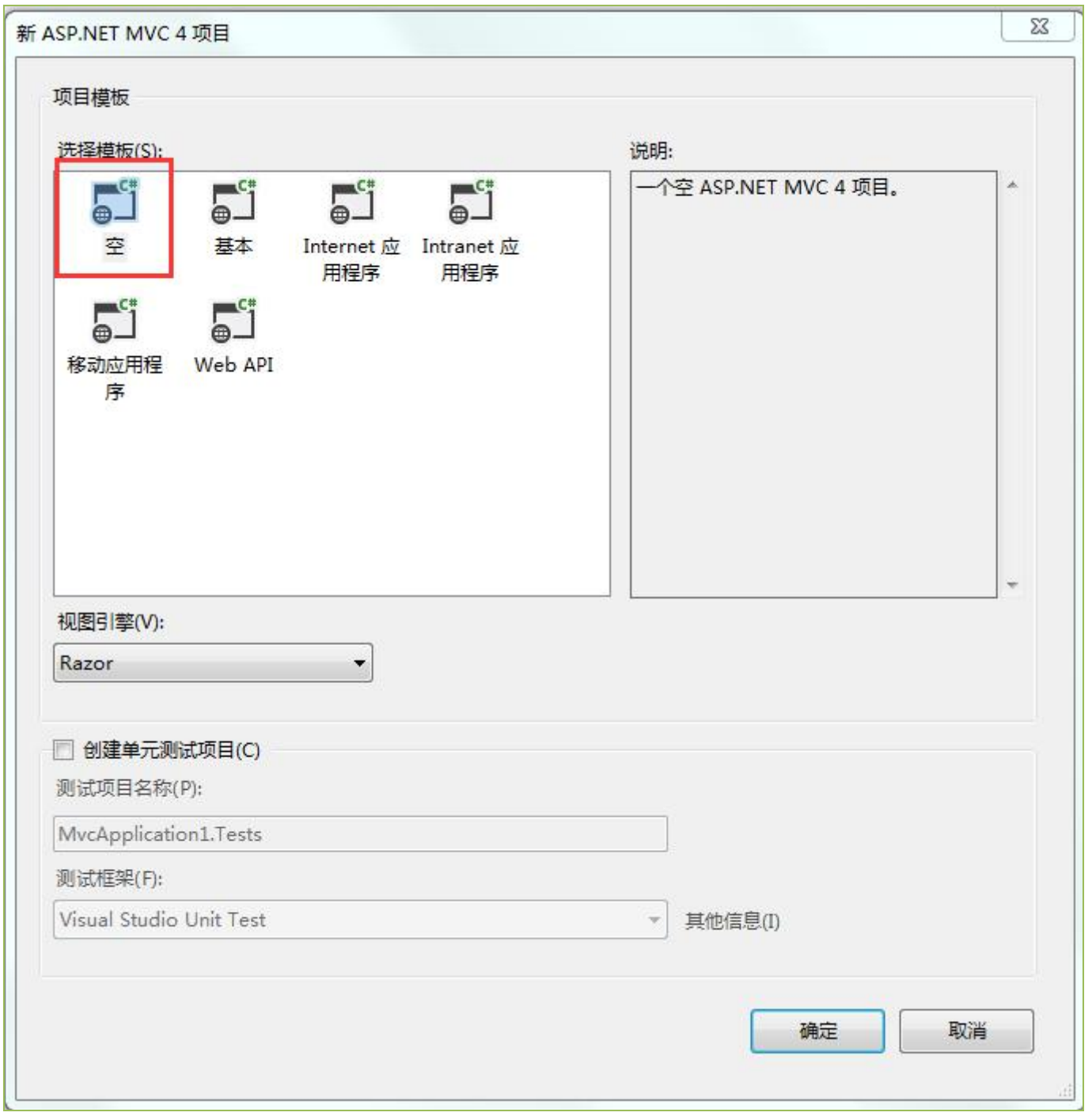
经过上面接口核心执行流程讲述后，我们现在开始我们的接口开发。首先我们来看一下项目的结构：



0.1.Frxs.ServiceCenter.Api.Core.V20　此项目为接口框架核心项目，此项目定义各种接口。开发 Action 接口需要引用此项目,其他项目都是已经基于接口框架的扩展项目（相当于插件，命名原则为：Frxs.ServiceCenter.Api.Core.{扩展名称名称}.{具体实现方式}），需要的时候，只要在 HOST 项目里引用这些扩展项目即可，系统会自动进行加载解析，至于如何进行插件的创建。后续章节会讲述。

首先我们新建一个 MVC 站点，用于作为接口框架的宿主



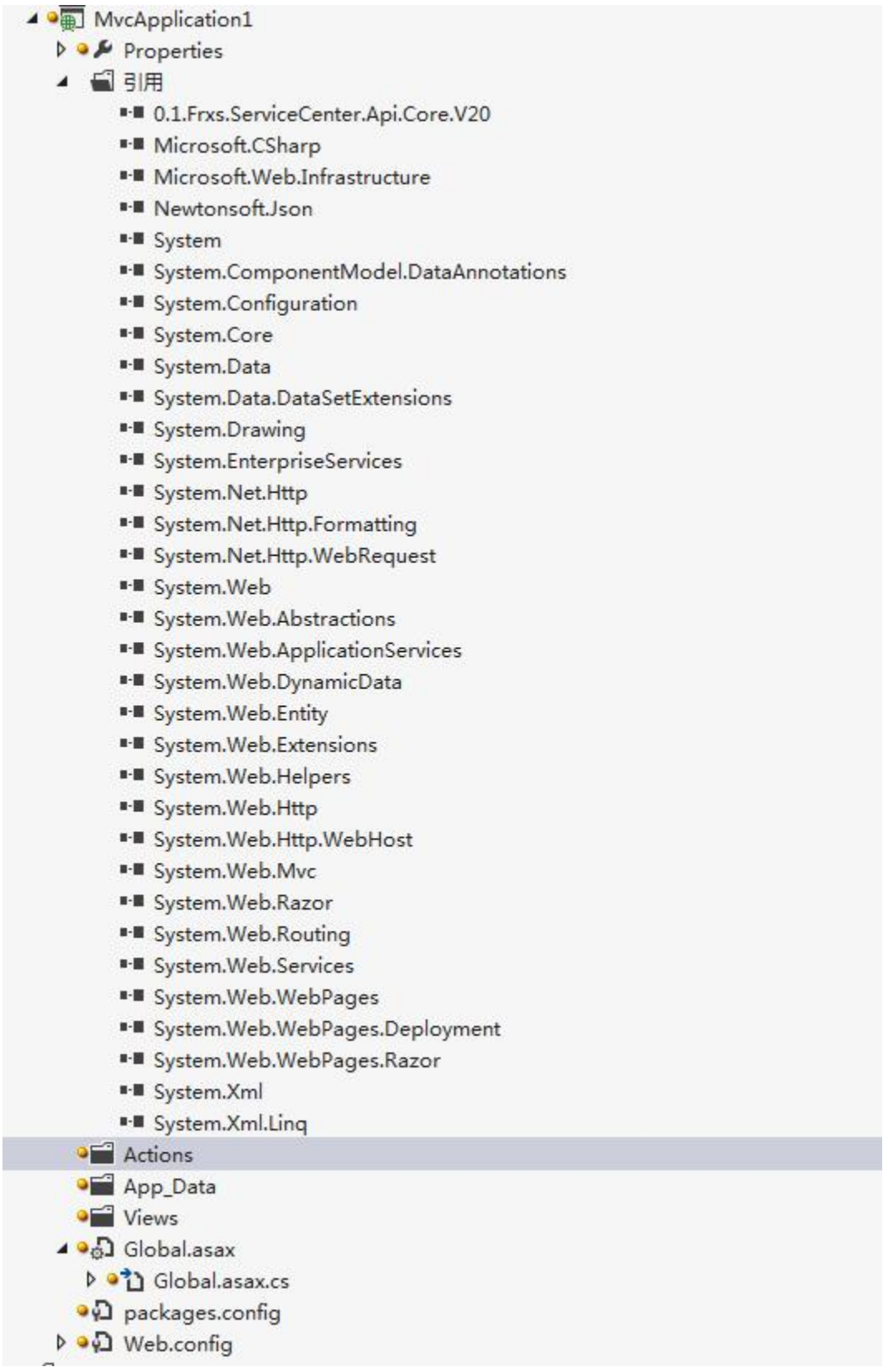


新建项目完成后

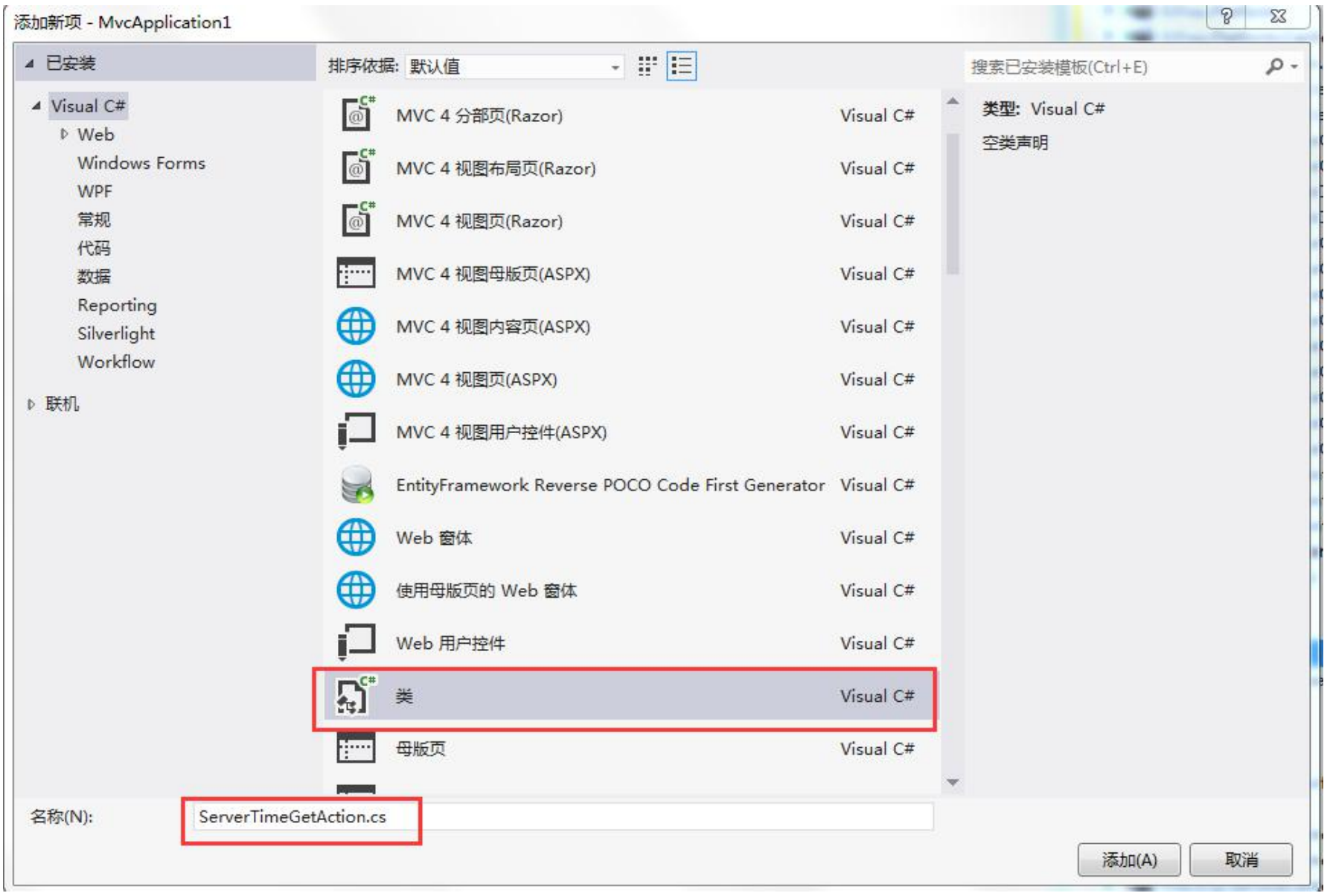
1. 删除掉项目里文件夹：~/App_Start,~/Controllers,~/Models 文件夹，
2. 新增 Actions 文件夹（用于放置接口类，当然我们完全可以重新新建一个类库项目来单独保存接口类，但是实例为了方便，就直接在我们的 HOST 项目里添加一个 Actions 文件夹作为接口类文件夹，接口类放置于哪个程序集对使用没有任何关系，系统框架会自动进行程序集扫描加载），
3. 添加对项目 0.1.Frxs.ServiceCenter.Api.Core.V20 引用
4. 配置 Global.asax,修改成如下

```
/// <summary>
///
/// </summary>
public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        //初始化系统框架
        Frxs.ServiceCenter.Api.Core.ApiApplication.Initialize();
    }
}
```

完成配置后，项目结构变成下面图示：



然后添加一个接口实现类，另外取名为：ServerTimeGetAction.cs ，注意接口类命名我们采取约定的方式。即:{模块命名}+{操作动词}+Action.cs ， 最后的 Action 并不是必须的，但是我们为了在后续开发中，快速知道哪些是接口类，哪些非接口类，加上 Action 后缀。



然后让类继承 ActionBase 抽象基类，TRequestDto 参数我们设置成系统框架内置的 NullRequestDto ，代表无上送参数，TResponseDto 我们设置 string 类型， 然后我们重写 `Execute()` 方法，直接使用 ActionBase 定义的 `SuccessActionResult` 方法，返回一个字符串。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using Frxs.ServiceCenter.Api.Core;

namespace MvcApplication1.Actions
{
    /// <summary>
    /// 获取服务器时间
    /// </summary>
    [ActionName("ServerTime.Get")]
    public class ServerTimeGetAction :ActionBase<NullRequestDto,string>
    {
        /// <summary>
        ///
        /// </summary>
        /// <returns></returns>
        public override ActionResult<string> Execute()
        {
         return this.SuccessActionResult(DateTime.Now.ToString());
        }
    }
}
```

从上面代码可以看出，在 ServerTimeGetAction 类上添加了一个 ActionName 特性,此特性用于指定当前接口对外调用的名称。此特性非必须，如果未指定 ActionName 特性，接口名称默认是类名取掉后缀 Action 后的属于部分。比如:ServerTimeGetAction 类若未指定 ActionName 特性,系统默认的名称就是：ServerTimeGet ，接口上还有如下特性可以附加：

1. ***ActionGroupAttribute（接口归类,不指定系统默认 null）***
2. ***ActionResultCacheAttribute(使用特性设置接口全局缓存)***
3. ***AllowAnonymousAttribute(允许匿名访问，不会进行身份授权校验，即授权器不会对此接口生效)***
4. ***AuthorAttribute(设置接口作者)***
5. ***DisableDataSignatureTransmissionAttribute(是否进行数据加密传输，禁止的话，不会走加解密流程)***
6. ***DisablePackageSdkAttribute(是否允许自动打包到SDK，需要配合客户端 SDK 生成器插件使用)***
7. ***EnableAjaxRequestAttribute(是否允许Ajax 方式访问接口)***
8. ***EnableRecordApiLogAttribute(是否允许日志记录器记录接口访问日志)***
9. ***HttpMethodAttribute(也许客户端HTTP 提交方式POST 或者 GET)***
10. ***RequireHttpsAttribute(是否需要HTTPS 安全连接方式访问接口)***

11. UnloadCachekeysAttribute(设置接口执行完成后，需要从 CacheManager 自动移除的缓存 key，可以模糊，即：只要缓存键里含有此指定关键词的缓存对象都会被批量移除)

12. VersionAttribute(接口版本，不指定，当前接口版本默认 0.0)

接下来我们再来看下，怎么给项目新增一个描述，我们再新增一个 ApiPluginDescriptor.cs 类（此类为非必须类，名称也非必须命名成这样，模块增加此类后，系统框架会自动搜索实现了 IApiPluginDescriptor 接口的类，将模块加载显示到 HOST 站点首页，方便开发查看），实现 IApiPluginDescriptor 接口即可，但是为不重复做事情，系统框架默认提供了一个插件描述对象抽象基类 ApiPluginDescriptorBase，抽象基类 ApiPluginDescriptorBase 实现了 IApiPluginDescriptor 接口并且已经帮我们默认实现大部分方法 ，我们只要继承 ApiPluginDescriptorBase 然后重写 DisplayName(插件名称)和 Author(插件作者)属性即可，当前接口描述对象抽象基类，里有很多属性都可以重写，比如：获取 url，获取 LOGO 地址等。现在我们这个类值重写名称和作者，其他属性都使用抽象类默认实现。代码如下：

```
/// <summary>
/// 插件描述
/// </summary>
[Serializable]
public class ApiPluginDescriptor : ApiPluginDescriptorBase
{
    /// <summary>
    ///
    /// </summary>
    /// <param name="resourceFinderManager"></param>
    public ApiPluginDescriptor(IResourceFinderManager resourceFinderManager)
        : base(resourceFinderManager)
    {
    }

    /// <summary>
    ///
    /// </summary>
    public override string DisplayName
    {
        get { return "接口集（演示接口项目）"; }
    }

    /// <summary>
    ///
    /// </summary>
    public override string Author
    {
        get { return "zhangliang@frxs.com"; }
    }
}
```

为了更好理解创建描述类，我们把系统创建描述对象接口 IApiPluginDescriptor 和框架默认提供的抽象实现基类代码也展示出来，便于理解。

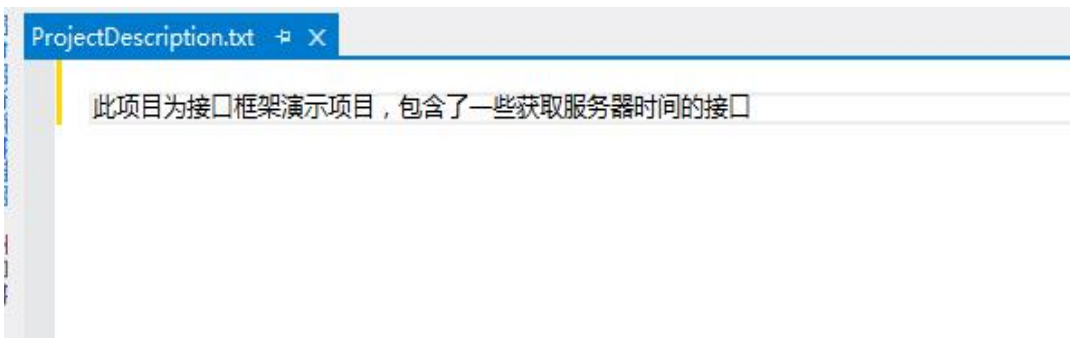
```
/// <summary>
/// 用于描述 API 扩展工具，方便框架搜索并且显示已经加载的扩展工具
/// </summary>
public interface IApiPluginDescriptor
{
    /// <summary>
    /// 显示名称
    /// </summary>
    string DisplayName { get; }

    /// <summary>
    /// 如果含有首页界面，首页访问地址是多少
    /// </summary>
string IndexUrl { get; }

/// <summary>
/// 版本
/// </summary>
string Version { get; }

/// <summary>
/// LOGO，URL 地址
/// </summary>
string Logo { get; }

/// <summary>
/// 作者
/// </summary>
string Author { get; }

/// <summary>
/// 插件描述
/// </summary>
string Description { get; }

/// <summary>
/// 获取依赖项
/// </summary>
IEnumerable<string> ReferencedAssemblies { get; }
```

```
}

/// <summary>
/// 接口创建描述基类
/// </summary>
public abstract class ApiPluginDescriptorBase : IApiPluginDescriptor
{
    /// <summary>
    /// 资源查找器
    /// </summary>
    private readonly IResourceFinderManager _resourceFinderManager;

    /// <summary>
    /// 过滤掉一些系统自带的 dll
    /// </summary>
    private const string AssemblySkipPattern = "^mscorlib|^System,|System.Xml,|System.Core,|System.Web,";

    /// <summary>
    /// 插件默认的 LOGO 地址
    /// </summary>
    private const string LogoUrl = "/GetResource?resourceName={0}";

    /// <summary>
    /// 插件 LOGO 合法的图片后缀集
    /// </summary>
    private static readonly string[] LogoFileExtensions = new string[] { "jpg", "png", "gif" };

    /// <summary>
    /// 组件描述说明文件
    /// </summary>
    private const string ProjectDescriptionFileName = "ProjectDescription.txt";

    /// <summary>
    /// 当前插件所属程序集
    /// </summary>
    private readonly Assembly _currenAssembly;

    /// <summary>
    /// 当前插件程序集命名空间
    /// </summary>
    private readonly string _currentAssemblyName;

    /// <summary>
    ///
    /// </summary>
    /// <param name="resourceFinderManager">资源查找器</param>
    protected ApiPluginDescriptorBase(IResourceFinderManager resourceFinderManager)
    {
        this._resourceFinderManager = resourceFinderManager;
        this._currenAssembly = this.GetType().Assembly;
        this._currentAssemblyName = this._currenAssembly.GetName().Name;
    }

    /// <summary>
    /// 插件名称
    /// </summary>
    public abstract string DisplayName { get; }

    /// <summary>
    /// 插件首页地址;默认为空
    /// </summary>
    public virtual string IndexUrl
    {
        get { return string.Empty; }
    }

    /// <summary>
    /// 插件版本，默认直接获取当前插件程序集版本
    /// </summary>
    public virtual string Version
    {
        get { return this._currenAssembly.GetName().Version.ToString(); }
    }

    /// <summary>
    /// 插件 LOGO;采取约定方式，默认会搜索 DLL 内嵌资源，根目录 Logo.jpg,Logo.gif,Logo.Png 和 Resource/(Logo.jpg,Logo.gif,Logo.Png)
    /// </summary>
    public virtual string Logo
    {
        get
        {
            //Resource 目录
            foreach (var fileExtension in LogoFileExtensions)
            {
                var resourceName = "{0}.Resource.Logo.{1}".With(this._currentAssemblyName, fileExtension);

                //是否存在 LOGO 资源缓存
                var logoBase64String = this._resourceFinderManager.GetResource(resourceName);

                if (!logoBase64String.IsNullOrEmpty())
                {
```

```
                return LogoUrl.With(resourceName);
            }
        }

        //根目录
        foreach (var fileExtension in LogoFileExtensions)
        {
            var resourceName = "{0}.Logo.{1}".With(this._currentAssemblyName, fileExtension);

            //是否存在 LOGO 资源缓存
            var logoBase64String = this._resourceFinderManager.GetResource(resourceName);

            if (!logoBase64String.IsNullOrEmpty())
            {
                return LogoUrl.With(resourceName);
            }
        }

        //返回默认的
        return LogoUrl.With("SystemPlugin.png");
    }
}

/// <summary>
/// 插件作者，默认为 string.Empty
/// </summary>
public virtual string Author
{
    get { return string.Empty; }
}

/// <summary>
/// 创建描述，使用约定方式，每个插件的根目录放置 ProjectDescription.txt 或者 Resource/ProjectDescription.txt 文件，系统框架自动回搜索读取此说明文件
/// </summary>
public virtual string Description
{
    get
    {
        //资源名称
        var resourceName = "{0}.{1}".With(this._currentAssemblyName, ProjectDescriptionFileName);

        //先找根目录
        var projectDescription = this._resourceFinderManager.GetResource(resourceName);
        if (!projectDescription.IsNullOrEmpty())
        {
            return projectDescription;
        }

        //再找 Resource 文件夹
        resourceName = "{0}.Resource.{1}".With(this._currentAssemblyName, ProjectDescriptionFileName);
        return this._resourceFinderManager.GetResource(resourceName);
    }
}

/// <summary>
/// 创建依赖那些程序集，如果不存在则返回一个空的集合
/// </summary>
public virtual IEnumerable<string> ReferencedAssemblies
{
    get { return FilterAssemblies(this._currenAssembly.GetReferencedAssemblies().Select(o => o.FullName)); }
}

/// <summary>
/// 过滤一些系统级别的程序集
/// </summary>
/// <param name="referencedAssemblies">插件引用的程序集</param>
/// <returns></returns>
private static IEnumerable<string> FilterAssemblies(IEnumerable<string> referencedAssemblies)
{
    return referencedAssemblies.Where(referencedAssembly => !Regex.IsMatch(referencedAssembly, AssemblySkipPattern, RegexOptions.IgnoreCase | RegexOptions.Compiled)).ToList();
}
}
```

到此，我们第一个 Action 接口就已经完成。现在我们允许下 F5，可以看下效果。

可以看到我们刚刚定义的接口 ServerTime.Get 已经出现在左边了。右边显示了我们加载的插件（我们将任何使用框架扩展出来的模块都看成是插件）；到现在我们已经完成了一个基本的接口开发。但是加载的插件并没有显示太多的说明信息，Logo 也是系统框架默认的，那么我们如果让模块调用者看到更多关系此模块的信息呢？现在我们为创建增加一个说明文件：在项目根目录创建一个 ProjectDescription.txt 文件，此文件为系统框架默认的项目说明文件，只要将此文件放置于项目根目录，系统会自动读取（当然，此文件还可以放置于项目 Resource 文件夹里）,然后我们想 ProjectDescription.txt 文件里写一些项目的说明：

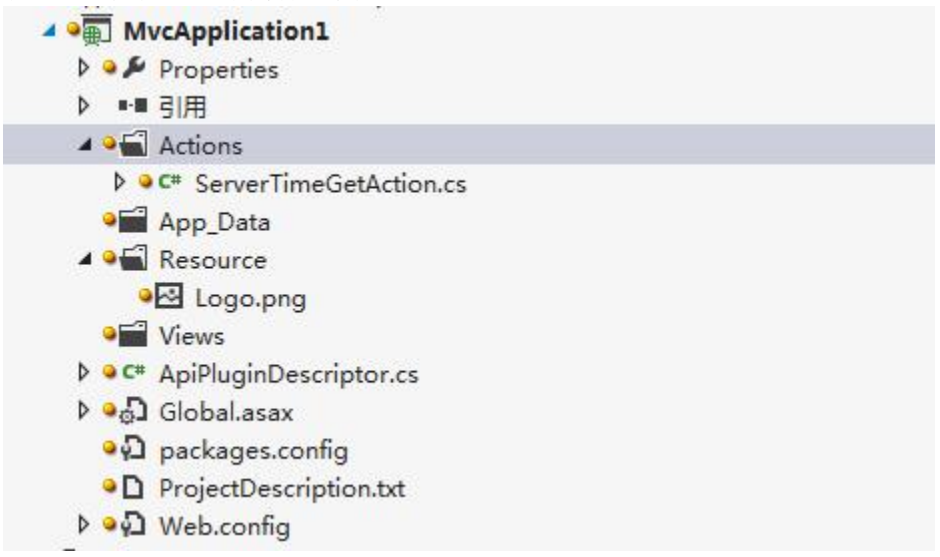


然后右键此文件，打开属性选项，更改下文件生成操作

保存后，重新编译下项目，然后 F5 预留，就可以看到右边插件说明了

下面我们再讲一下如何更改扩展项目模块的LOGO,我们在项目根目录新建一个 Resource 文件夹,然后添加一个图片到 Resource 文件夹，图片命名为 Logo.jpg(或者 Logo.png，Logo.gif)。和 ProjectDescription.txt 文件一样，将生成操作改成“嵌入的资源”，然后保存，编译。我们再次运行，就可以看到项目 LOGO 已经改变

现在整个项目结构看起来如下



接口写完，我们怎么去访问呢？我们回顾下前面的 `RequestParams` 对象,其包含了7个固定的参数,因此我只要通过POST或者GET的方式，提交相应的参数即可。接口完整的访问路径为：http://HOST/APi? 完整的参数如下：

ActionName:	ServerTime.Get	必须
Format:	XML/JSON/VIEW	可选，不传默认为 JSON
AppKey:		可选参数，当需要约定授权的时候，可能需要上传，根据实际项目需要上送
Version:	0.0	可选，不传，系统自动选择同名最高版本
TimeStamp:	2016-06-03 14:03:23	可选，当前调用者客户端时间戳
Sign:	B6E137495BD05167DD001CEABC251B81	可选，如果服务器实现了签名校验，就需要按照约定方式签名
Data:	{}	可选，如果传送了，就必须为合法的 JSON 串

通过上述我们构造出一个GET请求,返回格式化为JSON，如果需要返回XML或者VIEW只要将参数Format替换成Format=XML或者Format=VIEW即可

http://localhost:18769/Api?ActionName=ServerTime.Get&Format=JSON&AppKey=&Version=&TimeStamp=&Sign=&Data={}

返回的数据(JSON)

根据返回JSON或者XML数据，客户端（C#，安卓，IOS等）就可以进行解析，来和接口进行数据交互。但是如果一切工作都由手工的方式去写客户端访问代码的话，既繁琐又容易出错。因此在此思想基础上，我们可以对接口框架进行扩展，分别开发出针对C#或者安卓等客户端使用的SDK代码自动生成器。需要使用C#客户端SDK代码生成器，需要引用在HOST项目引用1.1.2.Frxs.ServiceCenter.Api.Core.SdkBuilder.CSharp项目即可，我们现在就引用下，看下效果：

右侧出了一个C#语言版SDK生成器，我们点击标题就可以打开此生成器，此生成器可以方便输出C#客户端代码访问

5.接口框架扩展点

5.1 身份/权限/正确性校验接口(IAuthentication)

对于一个接口来说，我们很多时候，需要对调用者身份进行校验，对上送的数据完整性进行检测。因此系统框架定义了 `IAuthentication` 接口用于校验用。我们先来看一下该接口的定义：

```
/// <summary>
/// 用于校验 APPKEY 身份，时间戳，上传数据签名，是否正确；具体实现交给外部去实现
/// 注意：此接口校验器为全局，如果想要定义单独接口授权校验器，请实现：ActionAuthenticationBaseAttribute 抽象类
/// 注意此接口，注册为协作类型，即：多个注册实现授权器会按照优先级全部执行一次
/// </summary>
public interface IAuthentication
{
    /// <summary>
    /// 用于排序优先级，越高越先执行;但是全局实现的接口肯定优先于特性接口
    /// </summary>
    int Order { get; }

    /// <summary>
    /// 验证身份是否通过；校验的时候请使用原始的请求参数，即：RequestContext.RawRequestParams 参数进行校验
    /// </summary>
    /// <param name="requestContext">请求参数</param>
    /// <returns>校验成功返回 true,失败返回 false</returns>
    AuthenticationResult Valid(RequestContext requestContext);
}
```

该接口的定义非常简单，Order只读属性用于返回当前校验器的优先级，数字越大，优先级越高。Valid 方法接收一个 RequestContext 当前请求上下文对象，返回一个 AuthenticationResult 对象。AuthenticationResult 对象用于表示一个校验器是否检验结果，其定义如下：

```
/// <summary>
/// 身份校验返回对象;全局校验，在接口类上，如果定义了 AllowAnonymousAttribute 特性类，全局验证将不起作用
/// </summary>
public class AuthenticationResult
{
    /// <summary>
    /// 是否校验通过
    /// </summary>
    public bool IsValid { get; private set; }

    /// <summary>
    /// 错误或者成功返回的消息
    /// </summary>
    public string Message { get; private set; }

    /// <summary>
    ///
    /// </summary>
    /// <param name="isValid">是否校验通过</param>
    /// <param name="message">错误或者成功返回的消息</param>
    public AuthenticationResult(bool isValid, string message)
    {
        this.IsValid = isValid;
        this.Message = message;
    }
}
```

系统框架提供了2种接口校验授权方式

第一种即：直接新建类，实现 IAuthentication 接口，然后将实现类注册到IOC容器，比如我们实现一个 DefaultAuthentication 授权器，代码如下：

```
/// <summary>
/// 全局默认的身份校验器
/// </summary>
public class DefaultAuthentication : IAuthentication
{
    /// <summary>
    /// 默认返回 true，全部通过，任何 APPKEY 都可以访问接口
    /// </summary>
    /// <param name="requestContext">当前请求上下文</param>
    /// <returns>身份信息验证是否通过，通过返回：true，失败返回:false</returns>
    public virtual AuthenticationResult Valid(RequestContext requestContext)
    {
        return new AuthenticationResult(true, "OK");
    }

    /// <summary>
    /// 排序
    /// </summary>
    public int Order
    {
        get { return int.MinValue; }
    }
}
```

然后使用注册文件进行注册，这样系统框架就会使用此校验器进行校验

```
/// <summary>
/// 注册系统默认实现的接口服务类
/// </summary>
public class DependencyRegistar : IDependencyRegistar
{
    /// <summary>
    /// 优先级最低，方便外部程序重写框架里的实现，覆盖掉系统默认的实现
```

```
        /// 最先注册下系统默认的实现；这样外部实现才能覆盖掉原始的实现
        /// </summary>
        public int Order
        {
            get
            {
                return 0
            }
        }

        /// <summary>
        /// 注册特定的类型到容器
        /// </summary>
        /// <param name="containerBuilder">注册容器</param>
        /// <param name="typeFinder">类型查找器</param>
        public void Register(ContainerBuilder containerBuilder, ITypeFinder typeFinder)
        {
            //默认身份验证器（空实现）
            containerBuilder.RegisterType<DefaultAuthentication>()
                .AsImplementedInterfaces()
                .InstancePerLifetimeScope();
        }

    }
```

我们顺便讲一下 `IDependencyRegistar` 这个接口，这个接口为框架级别的注册接口。系统启动的时候，系统框架会自动搜索实现此接口的类，然后进行批量自动注册，优先级越高，越后注册（覆盖签名已经注册的相同实现类）。

第二种方式，使用接口特性类来定义接口校验器（此验证方式区别于上面方式为，此方式为特定接口。上述的为全局），使用特性类来定义校验器，我们需要继承一个 `AuthenticationBaseAttribute` 特性抽象类，此特性抽象类同样实现了 `IAuthentication` 接口。其实现访问如下：

```
/// <summary>
/// 特定接口授权基类；需要实现接口自定义签名，授权校验的，请继承此类，重写 IsValid 方法即可；
/// 然后将自己实现的特性类，附加到接口类上面即可）
/// * ************************************************************************************
/// 为什么要定义此授权过滤器抽象基类，因为接口框架已经定义了一个 IAuthentication 授权接口，原因如下：
/// 1.IAuthentication 授权接口是统一的授权检测，比如参数正确性，加密是否正确等，即全局的授权判断
/// 2.但是有些接口需要一个独自的授权判断，因此定义了一个授权基类来实现不同接口实现不同授权判断的情况
/// </summary>
[AttributeUsage(AttributeTargets.Class, AllowMultiple = true)]
public abstract class AuthenticationBaseAttribute : Attribute, IAuthentication
{
    /// <summary>
    /// 优先级，越高越先执行判断;默认 0
    /// </summary>
    public virtual int Order { get; set; }

    /// <summary>
    /// 默认实现全部返回 true，继承类需要重写此方法，用于实际的业务逻辑授权判断
    /// </summary>
    /// <param name="requestContext">当前请求上下文</param>
    /// <returns>true/false</returns>
    public virtual AuthenticationResult Valid(RequestContext requestContext)
    {
        return new AuthenticationResult(true, "OK");
    }
}
```

现在我们来继承 `AuthenticationBaseAttribute` ，来实现一个自定义的一个叫 `TestAuthenticationAttribute` 的特性校验器。代码如下：

```
/// <summary>
/// 我们继承校验器抽象特性类,实现接口特定校验
/// </summary>
public class TestAuthenticationAttribute : AuthenticationBaseAttribute
{
    /// <summary>
    /// 演示校验数据签名不能为空
    /// </summary>
    /// <param name="requestContext"></param>
    /// <returns></returns>
    public override AuthenticationResult Valid(RequestContext requestContext)
    {
        if(requestContext.DecryptedRequestParams.Sign.IsNullOrEmpty())
        {
            return new AuthenticationResult(false, "数据签名不能为空");
        }

        //当前这里可以对所有的数据进行校验，或者取出 AppKey，读取数据库或者缓存，取出存在性，或者准确性等等

        return base.Valid(requestContext);
    }
 }
```

然后将此特性应用到我们之前写的 `ServerTime.Get` 接口上，代码如下：

```
/// <summary>
/// 获取服务器时间
/// </summary>
[ActionName("ServerTime.Get")]
[TestAuthentication]
public class ServerTimeGetAction :ActionBase<NullRequestDto,string>
{
    /// <summary>
    ///
    /// </summary>
    /// <returns></returns>
    public override ActionResult<string> Execute()
    {
     return this.SuccessActionResult(DateTime.Now.ToString());
    }
 }
```

再次访问此接口我们发现校验器已经生效了

说明：之前说过 AllowAnonymousAttribute 特性可以让定义了 IAuthentication 接口全局校验失效。但是对于单独的附加于Action接口上面的特性校验器，即使Action接口加了 AllowAnonymousAttribute 特性;特性校验还是会起作用。

为了更加详细的了解校验器的工作，我们来大概的讲述下校验流程，首先：在 IActionSelector 查找器在搜索到所有的接口同时，会将全局和特性校验器保存于 IActionDescriptor 属性 Authentications 上，具体的代码片段为

```
/// <summary>
/// 使用反射来获取 action 描述信息
/// </summary>
public class ReflectedActionDescriptor : IActionDescriptor
{
    /// <summary>
    /// 获取单个接口授权，签名校验
    /// </summary>
    public IEnumerable<IAuthentication> Authentications
    {
        get
        {
            //所有的权限验证器（全局的验证器要先执行，特性上面单独的验证器后执行）
            IList<IAuthentication> actionAuthenticationBaseAttribute = new List<IAuthentication>();

            //0.获取全局注册的授权器(需要进行授权访问)
            var authentications = ServicesContainer.Current.ResolverAll<IAuthentication>().OrderByDescending(o => o.Order);
            foreach (var authentication in authentications)
            {
                actionAuthenticationBaseAttribute.Add(authentication);
            }

            //1.获取特性上定义的独立授权特性
            IList<IAuthentication> actionAuthenticationBaseAttributes = this.ActionType.GetCustomAttributes()
                .OfType<AuthenticationBaseAttribute>()
                .Cast<IAuthentication>()
                .ToList();

            //排序后在加入
            foreach (var item in actionAuthenticationBaseAttributes.OrderByDescending(o => o.Order))
            {
                actionAuthenticationBaseAttribute.Add(item);
            }

            //3.返回接口授权集合
            return actionAuthenticationBaseAttribute;
        }
    }
}
```

统一校验使用 AuthenticationManager 管理类来进行，实现代码为：

```
/// <summary>
/// 权限校验管理器
/// </summary>
internal class AuthenticationManager
{
```

```
/// <summary>
/// 针对允许匿名和非匿名进行判断，如果接口允许匿名访问，接口将不走 IAuthentication 接口流程，直接返回校验成功；
/// </summary>
/// <param name="requestContext">当前请求上下文</param>
/// <param name="actionDescriptor">接口描述对象</param>
/// <returns>身份信息验证是否通过，通过返回：true，失败返回:false</returns>
public AuthenticationResult Valid(RequestContext requestContext, IActionDescriptor actionDescriptor)
{
    requestContext.CheckNullThrowArgumentNullException("requestContext");
    actionDescriptor.CheckNullThrowArgumentNullException("actionDescriptor");

    //获取接口所有的授权器，包括特性和全局注册的
    var authentications = actionDescriptor.Authentications;

    //循环授权器，看验证是否通过
    foreach (var authentication in authentications)
    {
        //允许匿名，不进行全局授权校验，但是要进行接口自定义的授权校验
        if (actionDescriptor.AllowAnonymous && !(authentication is AuthenticationBaseAttribute))
        {
            continue;
        }

        //授权校验
        var validResult = authentication.Valid(requestContext);

        //授权校验失败，直接返回，不进行后续的校验
        if (!validResult.IsValid)
        {
            return validResult;
        }
    }

    //直接返回校验成功
    return new AuthenticationResult(true, "OK");
}
}
```

5.2 安全校验器，加解密接口(IApiSecurity)

针对一些安全性需求非常高的接口，开发的时候，可能需要进行加密上送数据或者加密下发数据，针对这样的需求，接口框架定义了一个加解密 `IApiSecurity` 接口，其定义为2个方法，针对上送参数解密和针对下送数据加密。具体定义如下代码：

```
/// <summary>
/// 接口上送参数，下送数据加密解密接口；在实现类里请不要抛出任何异常
/// 注意：此接口为单一注册接口，最后注册的实现会覆盖掉前面注册的实现
/// </summary>
public interface IApiSecurity
{
    /// <summary>
    /// 上送的参数对象进行解密，具体根据对那部分进行解密，需要在实际项目里进行双方约定
    /// </summary>
    /// <param name="rawRequestParams">上送参数对象</param>
    /// <returns>解密后的数据对象</returns>
    RequestParamsDecryptResult RequestParamsDecrypt(RequestParams rawRequestParams);

    /// <summary>
    /// 下送数据加密方法
    /// </summary>
    /// <param name="decryptedRequestParams">
    /// 解密后的上送参数对象，
    /// 为什么要定义此解密后的参数原因：由于有可能会根据不同的 AppKey 生成不同的加密解密方式，
    /// 因此需要将上送的参数对象传入，用于差异化加密</param>
    /// <param name="actionResultString">下送的 JSON 或者 XML 或者 View 数据加密</param>
    /// <returns>加密后的数据</returns>
    string ResponseEncrypt(RequestParams decryptedRequestParams, string actionResultString);
}
```

系统框架里如果需要使用加解密流程，我们可以实现一个类，然后实现接口 `IApiSecurity` 即可。下面我们实现一个加解密器 `DefaultApiSecurity` ，代码片段如下：

```
/// <summary>
/// 系统框架默认的安全接口，此接口什么都没有做，直接原路返回，即上送参数解密，下送参数解密；
/// 具体实现类可以继承此类，对上送参数解密，下送数据加密，已经数据签名进行校验等
/// </summary>
public class DefaultApiSecurity : IApiSecurity
{
    /// <summary>
    /// 获取待签名的键值对，为空或者为 null 的属性不会添加到字典表，已经安装 key 键进行排序(a-z)
    /// </summary>
    /// <param name="rawRequestParams">请求的原始上送参数对象</param>
    /// <returns></returns>
    protected virtual string GetSignRequestParamsString(RequestParams rawRequestParams)
    {
        //根据上送参数进行排序，排除掉 Sign 参数
        var keyvalue = rawRequestParams.GetAttributes(false, false)
            .Where(o => !o.Key.Equals("Sign", StringComparison.OrdinalIgnoreCase))
```

```
            .Select(o => new KeyValuePair<string, string>(o.Key, o.Value.ToString()))
            .ToList();

        //返回待签名的上送数据
        return string.Join("", keyvalue.Select(o => o.Value).ToArray());
    }

    /// <summary>
    /// 根据 APPKEY 获取到数据签名密钥
    /// </summary>
    /// <param name="appKey"></param>
    /// <returns></returns>
    protected virtual string GetAppSecret(string appKey)
    {
        return appKey;
    }

    /// <summary>
    /// 直接返回上送 data 数据
    /// </summary>
    /// <param name="rawRequestParams">上送参数对象</param>
    /// <returns>返回解密对象结果</returns>
    public virtual RequestParamsDecryptResult RequestParamsDecrypt(RequestParams rawRequestParams)
    {
        ////根据 AppKey 的值，查询出对应的签名密钥
        //string appSecrct = this.GetAppSecret(rawRequestParams.AppKey);

        ////签名值
        //var sign = this.MD5Encrypt("{0}{1}{0}".With(appSecrct, this.GetSignRequestParamsString(rawRequestParams)));

        ////判断下数据签名是否正确
        //if (!sign.Equals(rawRequestParams.Sign))
        //{
        //  //  return new RequestParamsDecryptResult(false, "数据签名错误", rawRequestParams,
rawRequestParams.MapTo<RequestParams>());
        //}

        //创建一个新的解密上送参数对象(防止直接应用，修改的时候出现问题)
        var decryptedRequestParams = rawRequestParams.MapTo<RequestParams>();

        //返回默认解密结果
        return new RequestParamsDecryptResult(true, "OK", rawRequestParams, decryptedRequestParams);
    }

    /// <summary>
    /// 直接返回下送数据
    /// </summary>
    /// <param name="actionResultString">actionResult 对象格式化字符串</param>
    /// <param name="decryptedRequestParams">解密后的上送参数对象</param>
    /// <returns></returns>
    public virtual string ResponseEncrypt(RequestParams decryptedRequestParams, string actionResultString)
    {
        //如果每个客户端都是不同的加密方式，可以根据 decryptedRequestParams 参数里的 appKey 来获取特定客户端加密约定，
        //具体怎么获取可以从数据库，从配置文件，从其他存储方式等等

        return actionResultString;
    }

}
```

然后使用注册器，注册到IOC容器，实现我们自定义的加解密器，当然在具体的项目里，我们需要和调用端进行协商，使用那些加解密方式。下面代码片段显示如何将我们自定义的加解密器注册到IOC。

```
/// <summary>
/// 注册系统默认实现的接口服务类
/// </summary>
public class DependencyRegistar : IDependencyRegistar
{
    /// <summary>
    /// 优先级最低，方便外部程序重写框架里的实现，覆盖掉系统默认的实现
    /// 最先注册下系统默认的实现；这样外部实现才能覆盖掉原始的实现
    /// </summary>
    public int Order
    {
        get
        {
            return 1
        }
    }

    /// <summary>
    /// 注册特定的类型到容器
    /// </summary>
    /// <param name="containerBuilder">注册容器</param>
    /// <param name="typeFinder">类型查找器</param>
    public void Register(ContainerBuilder containerBuilder, ITypeFinder typeFinder)
    {
        //接口加密解密器(空实现)
        containerBuilder.RegisterType<DefaultApiSecurity>()
            .AsImplementedInterfaces()
```

```
            .SingleInstance();
        }
    }
```

5.3 日志记录接口(ILogger，ILoggerFactory)

对于一个系统来说，日志的记录非常重要，比如系统框架错误日志，调试日志等等，接口框架提供了接口日志记录器接口，但是没有实现，需要我们外部自己去实现日志记录器。系统框架值定义了一个空实现（NullLogger和NullLoggerFactory），具体实现我们可以实现2个定义的接口即可(我们完全可以根据自己的需求，将日志记录器记录到不同的介质，比如：文本文件，关系数据库或者NoSql等)。定义的接口代码如下：

```
/// <summary>
/// 日志记录器
/// </summary>
public interface ILogger
{
    /// <summary>
    /// 是否启用日志记录器(针对某一级别的)
    /// </summary>
    /// <param name="level"></param>
    /// <returns></returns>
    bool IsEnabled(LogLevel level);

    /// <summary>
    /// 记录日志
    /// </summary>
    /// <param name="level">记录等级</param>
    /// <param name="exception">错误异常</param>
    /// <param name="format">格式化字符串，如：服务器错误{0}.....</param>
    /// <param name="args">格式化字符参数值</param>
    void Log(LogLevel level, Exception exception, string format, params object[] args);
}


/// <summary>
/// 日志记录器创建工厂
/// </summary>
public interface ILoggerFactory
{
    /// <summary>
    /// 创建日志记录器
    /// </summary>
    /// <param name="type">任意的类型；不会影响到 ILogger 的创建；仅仅作为日志记录异常类</param>
    /// <returns></returns>
    ILogger CreateLogger(Type type);
}
```

扩展项目，我们实现了一个基于Log4Net的日志实现。

项目为：2.3.1.Frxs.ServiceCenter.Api.Core.Logging.Log4Net 使用的时候，可以直接在HOST项目里引用此项目。系统就会自动加载，我们这里对实现不做多的说明。我们只将一下在接口里如果使用日志接口，还记得前面讲到 ActionBase 抽象基类么？ 此类上面定义了一个 Logger 属性，其类型就是 ILogger 。我们只要在实现类里直接使用即可。另外，其他服务类，需要使用 ILogger 也非常简单，只要在构造函数里定义此类型参数即可。后续的所有事情，系统框架都会完成，会自动注入相应的Logger实现。我们看一下下面Action接口实现（基于属性注入和构造函数注入）

```
/// <summary>
/// 获取服务器时间
/// </summary>
[ActionName("ServerTime.Get")]
[TestAuthentication]
public class ServerTimeGetAction : ActionBase<NullRequestDto, string>
{
    /// <summary>
    ///
    /// </summary>
    private ILogger _logger;

    /// <summary>
    /// 自动注入日志记录器
    /// </summary>
    /// <param name="logger">日志记录器</param>
    public ServerTimeGetAction(ILogger logger)
    {
        this._logger = logger;
    }

    /// <summary>
    ///
    /// </summary>
    /// <returns></returns>
    public override ActionResult<string> Execute()
    {
        //直接使用基类属性
        this.Logger.Debug("使用属性注入 Logger");

        //使用构造函数
        this._logger.Debug("使用构造函数注入 Logger");
```

```
            //由于日志记录器注册成单例，所以属性 Logger 和构造函数的 Logger 是同一个对象（引用）

            return this.SuccessActionResult(DateTime.Now.ToString());
        }
    }
```

Logger属性和构造函数注入，适用于所有注册到IOC的类，不仅仅针对Action接口，因此我们实现日志记录器，无须关心到底实现是什么，我们只要知道，定义一个属性或者定义一个构造函数，系统框架就会自动注入即可。

另外我们同时注意到，在系统框架里默认实现了一个 NullLogger,NullLoggerFactory 空实现。我们称之为这样的设计为“空实现模式”，虽然它不是一个标准的设计模式，但是在实际开发中，此模式不亚于任何一个经典的设计模式。我们后续的缓存，语言包等设计都使用了此模式。此模式的好处是：1.减少程序中判断的逻辑，因为在类构造函数里，我们一般先判断注入的是否为空对象，如果为null，则默认给予空实现。2.方便截断任意实现，比如缓存，我们使用的是Redis的实现，但是在某个时间点，我们需要覆盖掉Redis实现，而采取空实现，即：直接走数据库检索，而不走缓存检索。那么我们可以在不改变任何以前代码的情况下，直接将空实现的Dll复制到bin目录即可。移除空实现，只要删除对应的Dll即可还原原来的实现。这也满足软件开发的开闭原则。

5.4 接口访问记录接口(IApiAccessRecorder)

接口访问记录接口允许我们是非方便的将客户端访问接口详情记录到不同介质，比如：数据库，文本文件，NOSQL等等存储介质。我们再扩展记录器的时候，只要实现 IApiAccessRecorder 接口即可。下面我们介绍下如何进行扩展。

首先，新建一个类库项目，然后引用 0.1.Frxs.ServiceCenter.Api.Core.V20 项目或者直接引用 Frxs.ServiceCenter.Api.Core.dll。然后新建一个实现类，我们取名叫： TestAccessRecorder.cs ,具体的实现简单逻辑如下：

```
/// <summary>
/// 测试接口访问记录器实现
/// </summary>
public class TestAccessRecorder : IApiAccessRecorder
{
    /// <summary>
    /// 记录器优先记录次序，次序越大，越先记录
    /// </summary>
    public int Priority
    {
        get
        {
            return 9;
        }
    }

    /// <summary>
    /// 记录日志
    /// </summary>
    /// <param name="args"></param>
    public void Record(ApiAccessRecorderArgs args)
    {

        // args.ActionDescriptor.ActionName;

        //此处记录日志处理

        throw new NotImplementedException();
    }
}
```

实现代码后，我们将此实现注册到IOC容器：实现代码如下：

```
/// <summary>
/// API 框架会自动检测到这里的注册类,自动完成注册
/// </summary>
public class DependencyRegistar : IDependencyRegistar
{
    /// <summary>
    /// 系统框架默认的会被覆盖;
    /// </summary>
    /// <param name="containerBuilder"></param>
    /// <param name="typeFinder">类型查找器</param>
    public void Register(ContainerBuilder containerBuilder, ITypeFinder typeFinder)
    {
        containerBuilder.RegisterType<TestAccessRecorder>()
            .As<IApiAccessRecorder>()
            .SingleInstance();
    }

    /// <summary>
    /// 数字越大越后注册
    /// </summary>
    public int Order
    {
        get { return 1; }
    }
}
```

这样就完成了一个记录器的实现，并注册到IOC容器，后续我们HOST项目只要引用此项目即可（或者直接引用其生产的Dll）

说明： IApiAccessRecorder 接口系统框架定义成了多实现协作接口，即：多个实现不是竞态模式（后续注册覆盖前面注册）。这样我们的多个接口访问记录器实现会同时生效，比如：我们注册了一个将访问日志记录到数据库的记录器同时注册了一个将访问记录记录到文本的记录器，那么系统框架会根据记录器实现的优先级(Order属性)来依次进行记录。为了方便我们更加好的理解多个记录器如何工作，我们再来看下具体的记录器协调器管理类实现代码（ApiAccessRecordPublisher.cs）：

```
/// <summary>
/// API 日志记录器发布者，给所有订阅者进行接口信息发布
/// </summary>
internal class ApiAccessRecordPublisher
{
    /// <summary>
    /// 发布日志访问消息；会逐个调用订阅者，进行发布
    /// </summary>
    /// <param name="actionResultString">执行结果的字符串</param>
    /// <param name="requestContext">当前请求上下文</param>
    /// <param name="actionLifeTime">Action 对象的执行时间对象</param>
    public static void Publish(string actionResultString, RequestContext requestContext, ActionLifeTime actionLifeTime)
    {
        //不记录访问日志，直接返回
        if (!SystemOptionsManager.Current.EnableAccessRecorder || !requestContext.ActionDescriptor.EnableRecordApiLog)
        {
            return;
        }
        //获取所有注册的消费者
        var apiAccessRecorders = ServicesContainer.Current.ResolverAll<IApiAccessRecorder>();

        //获取当前操作用户信息
        var currentUserIdentity = requestContext.GetCurrentUserIdentity(() => new UserIdentity() { UserId = -1, UserName =
"unknown" });

        //构造出记录器需要的参数
        var args = new ApiAccessRecorderArgs()
        {
            RequestStartTime = actionLifeTime.StartTime,
            RequestEndTime = actionLifeTime.EndTime,
            RequestUsedTotalMilliseconds = actionLifeTime.UsedTotalMilliseconds,
            HttpMethod = requestContext.HttpContext.Request.HttpMethod,
            Ip = requestContext.HttpContext.Request.GetClientIp(),
            RequestData = requestContext.DecryptedRequestParams.Data ?? string.Empty,
            ResponseData = actionResultString,
            ResponseFormat = requestContext.DecryptedRequestParams.Format,
            UserId = currentUserIdentity.UserId,
            UserName = currentUserIdentity.UserName,
            ActionDescriptor = requestContext.ActionDescriptor,
            RequestParams = requestContext.DecryptedRequestParams
        };

        //获取所有已经注册的接口
        apiAccessRecorders.OrderByDescending(x => x.Priority).ToList().ForEach(item =>
        {
            try
            {
                item.Record(args);
            }
            catch
            {
                // ignored
            }
        });
    }
}
```

从管理器代码可以看出，管理器先从IOC容器里将所有注册的 IApiAccessRecorder 接口实现反转出来，然后根据优先级依次进行访问日志记录。从代码可以看出，此发布器相当于发布/订阅模式的发布者角色，所有的实现相当于订阅者角色。我们没有C#原生的事件(Event)模式来实现，而是使用了IOC容器的方式来实现了发布/订阅。方便了扩展和维护。

另外，我们看到，发布者 ApiAccessRecordPublisher 类，并没有实现多线程的方式来进行发布，所以在具体的订阅者实现类里，为了不影响到接口框架的后续执行，请在具体的实现类要以最快的速度返回，可以在具体实现类里进行多线程操作。

5.5 缓存接口(ICacheManager)

框架缓存接口已经深入到了接口框架的各个部分，是提高接口相应速度的一个重要策略。在资源查找器里，接口搜索器里等等实现类，都有缓存接口的影子。因此这部分我们将会详细的介绍缓存接口。

同其他定义的接口一样，系统框架也仅仅给了缓存接口的定义，具体的实现要交给调用者去实现，但是为了让接口框架正常工作，框架对接口缓存给了特殊待遇。即：接口框架里，默认实现了2个具体的实现（MemoryCacheManager，PerRequestCacheManager）来支撑接口工作。我们先来看一下缓存接口的定义：

```
/// <summary>
/// 系统框架缓存器接口；在实际使用中请引入：ICacheManager.Extensions 扩展来进行使用
/// </summary>
public interface ICacheManager
{
    /// <summary>
    /// 根据缓存键获取缓存实体对象
    /// </summary>
    /// <typeparam name="T">明确的缓存对象</typeparam>
    /// <param name="key">缓存键</param>
    /// <returns>指定缓存类型的对象</returns>
    T Get<T>(string key);

    /// <summary>
    /// 设置缓存；注意这里设置缓存，是否先删除缓存还是不删除已有缓存；请在具体实现里做
    /// </summary>
    /// <param name="key">缓存键</param>
    /// <param name="data">缓存数据</param>
    /// <param name="cacheTime">缓存过期时间,单位为分钟</param>
    void Set(string key, object data, int cacheTime);

    /// <summary>
    /// 根据缓存键判断是否已经有缓存
    /// </summary>
    /// <param name="key">缓存键</param>
    /// <returns>Result</returns>
    bool IsSet(string key);

    /// <summary>
    /// 根据缓存键删除对应缓存
    /// </summary>
    /// <param name="key">缓存键</param>
    void Remove(string key);

    /// <summary>
    /// 根据正则表达式来删除缓存
    /// </summary>
    /// <param name="pattern">正则表达式匹配模式</param>
    void RemoveByPattern(string pattern);

    /// <summary>
    /// 清空全部的缓存键
    /// </summary>
    void Clear();
}
```

从定义可以看出，缓存接口仅仅定义了6个方法。我们的实现只要针对这6个定义来进行实现即可。特别需要注意的是，方法：`void RemoveByPattern(string pattern)` ，此方法十分重要，是我们整个缓存机制的核心。此方法要求具体实现为：输入一个正则匹配模式，来批量删除相关的缓存像。比如：我们传入参数pattern为：_SYS_，那么所有缓存项key值含有_SYS_关键词的缓存项，都需要被移除。或者我们输入：^SYS ，那么所有以：SYS开头的缓存键都需要被移除。所以具体实现的时候需要注意下接口定义的这个方法。

当然，缓存接口仅仅定义上述6个方法，在我们实际开发中足够满足，但是对于调用者来说，6个方法显得比较单薄。调用起来不是非常的方便。因此我们定义基于 `ICacheManager` 接口的一些扩展。具体定义如下：

```
/// <summary>
/// 缓存扩展类
/// </summary>
public static class CacheManagerExtensions
{
    /// <summary>
    /// Variable (lock) to support thread-safe
    /// </summary>
    private static readonly object SyncObject = new object();

    /// <summary>
    /// Get a cached item. If it's not in the cache yet, then load and cache it;
    /// the default cachetime is 30*24*60 minutes 30days
    /// </summary>
    /// <typeparam name="T">Type</typeparam>
    /// <param name="cacheManager">Cache manager</param>
    /// <param name="key">Cache key</param>
    /// <param name="acquire">Function to load item if it's not in the cache yet</param>
    /// <returns>Cached item</returns>
    public static T Get<T>(this ICacheManager cacheManager, string key, Func<T> acquire)
    {
        return Get(cacheManager, key, 30 * 24 * 60, acquire);
    }

    /// <summary>
    /// Get a cached item. If it's not in the cache yet, then load and cache it
    /// </summary>
    /// <typeparam name="T">Type</typeparam>
    /// <param name="cacheManager">Cache manager</param>
    /// <param name="key">Cache key</param>
    /// <param name="cacheTime">Cache time in minutes (0 - do not cache)</param>
    /// <param name="acquire">Function to load item if it's not in the cache yet</param>
    /// <returns>Cached item</returns>
```

```
public static T Get<T>(this ICacheManager cacheManager, string key, int cacheTime, Func<T> acquire)
{
    if (cacheManager.IsSet(key))
    {
        return cacheManager.Get<T>(key);
    }
    lock (SyncObject)
    {
        var result = acquire();
        if (cacheTime > 0)
        {
            cacheManager.Set(key, result, cacheTime);
        }
        return result;
    }
}

/// <summary>
/// Set cache data; the default cachetime is 30*24*60 minutes
/// </summary>
/// <param name="cacheManager"></param>
/// <param name="key">Cache Key</param>
/// <param name="data">the data wait to Cached</param>
public static void Set(this ICacheManager cacheManager, string key, object data)
{
    lock (SyncObject)
    {
        cacheManager.Set(key, data, 30 * 24 * 60);
    }
}
```

扩展类里定义了3个方便调用的方法，Get方法，是对原始 ICacheManager 接口的一些组合，在传入的缓存key的时候，如果缓存里不存在此缓存键，那么就使用委托从存储介质里获取数据，然后调用Set压入缓存，再返回给调用者。这样我们再使用的时候就非常方便，代码也显得是非紧凑。

下面我们先来看一下系统框架默认实现的2个缓存，具体的逻辑我们不做多的阐述，我们仅仅只看一下实现，来学习下如何实现缓存接口即可：

```
    /// <summary>
    /// ASP.NET 自带内存缓存器(系统框架自带的缓存器实现),注意使用内存缓存，需要 T 类型为可以序列化的类型；建议在实体类上面都加上可以序列化特性
    /// </summary>
    public partial class MemoryCacheManager : ICacheManager
    {
        /// <summary>
        /// Cache object
        /// </summary>
        protected ObjectCache Cache
        {
            get
            {
                return MemoryCache.Default;
            }
        }

        /// <summary>
        /// Gets or sets the value associated with the specified key.
        /// </summary>
        /// <typeparam name="T">Type</typeparam>
        /// <param name="key">The key of the value to get.</param>
        /// <returns>The value associated with the specified key.</returns>
        public virtual T Get<T>(string key)
        {
            return (T)this.Cache[key];
        }

        /// <summary>
        /// Adds the specified key and object to the cache.
        /// </summary>
        /// <param name="key">key</param>
        /// <param name="data">Data</param>
        /// <param name="cacheTime">Cache time</param>
        public virtual void Set(string key, object data, int cacheTime)
        {
            //为空的话，系统将直接返回，不进行设置缓存
            if (data.IsNull())
            {
                return;
            }

            //如果存在就先删除，然后添加新缓存
            if (this.IsSet(key))
            {
                this.Remove(key);
            }

            //绝对过期时间
            var policy = new CacheItemPolicy {AbsoluteExpiration = DateTime.Now + TimeSpan.FromMinutes(cacheTime)};
```

```
            //设置缓存
            this.Cache.Add(new CacheItem(key, data), policy);
        }

        /// <summary>
        /// Gets a value indicating whether the value associated with the specified key is cached
        /// </summary>
        /// <param name="key">key</param>
        /// <returns>Result</returns>
        public virtual bool IsSet(string key)
        {
            return (this.Cache.Contains(key));
        }

        /// <summary>
        /// Removes the value with the specified key from the cache
        /// </summary>
        /// <param name="key">/key</param>
        public virtual void Remove(string key)
        {
            this.Cache.Remove(key);
        }

        /// <summary>
        /// Removes items by pattern
        /// </summary>
        /// <param name="pattern">pattern</param>
        public virtual void RemoveByPattern(string pattern)
        {
            var regex = new Regex(pattern, RegexOptions.Singleline | RegexOptions.Compiled | RegexOptions.IgnoreCase);
            var keysToRemove = (from
                                    item in
                                    this.Cache
                                where
                                    regex.IsMatch(item.Key)
                                select item.Key).ToList();
            foreach (string key in keysToRemove)
            {
                this.Remove(key);
            }
        }

        /// <summary>
        /// Clear all cache data
        /// </summary>
        public virtual void Clear()
        {
            foreach (var item in this.Cache)
            {
                this.Remove(item.Key);
            }
        }

        /// <summary>
        /// 释放，直接释放掉所有的缓存键
        /// </summary>
        public void Dispose()
        {
            //this.Clear();
        }
    }

    /// <summary>
    /// Represents a manager for caching during an HTTP request (short term caching)
    /// </summary>
    public partial class PerRequestCacheManager : ICacheManager
    {
        /// <summary>
        ///
        /// </summary>
        private readonly HttpContextBase _httpContext;

        /// <summary>
        /// Ctor
        /// </summary>
        /// <param name="context">Context</param>
        public PerRequestCacheManager(HttpContextBase context)
        {
            this._httpContext = context;
        }

        /// <summary>
        /// Creates a new instance of the NopRequestCache class
        /// </summary>
        protected virtual IDictionary GetItems()
        {
            return !_httpContext.IsNull() ? _httpContext.Items : null;
        }

        /// <summary>
        /// Gets or sets the value associated with the specified key.
        /// </summary>
```

```
        /// <typeparam name="T">Type</typeparam>
        /// <param name="key">The key of the value to get.</param>
        /// <returns>The value associated with the specified key.</returns>
        public virtual T Get<T>(string key)
        {
            var items = GetItems();
            if (items.IsNull())
            {
                return default(T);
            }
            return (T)items[key];
        }

        /// <summary>
        /// Adds the specified key and object to the cache.
        /// </summary>
        /// <param name="key">key</param>
        /// <param name="data">Data</param>
        /// <param name="cacheTime">Cache time</param>
        public virtual void Set(string key, object data, int cacheTime)
        {
            var items = GetItems();
            if (items.IsNull() || data.IsNull())
            {
                return;
            }
            if (items.Contains(key))
            {
                items[key] = data;
            }
            else
            {
                items.Add(key, data);
            }
        }

        /// <summary>
        /// Gets a value indicating whether the value associated with the specified key is cached
        /// </summary>
        /// <param name="key">key</param>
        /// <returns>Result</returns>
        public virtual bool IsSet(string key)
        {
            var items = GetItems();
            if (items.IsNull())
            {
                return false;
            }
            return (!items[key].IsNull());
        }

        /// <summary>
        /// Removes the value with the specified key from the cache
        /// </summary>
        /// <param name="key">/key</param>
        public virtual void Remove(string key)
        {
            var items = GetItems();
            if (items.IsNull())
            {
                return;
            }
            items.Remove(key);
        }

        /// <summary>
        /// Removes items by pattern
        /// </summary>
        /// <param name="pattern">pattern</param>
        public virtual void RemoveByPattern(string pattern)
        {
            var items = GetItems();
            if (items.IsNull())
            {
                return;
            }

            var enumerator = items.GetEnumerator();
            var regex = new Regex(pattern, RegexOptions.Singleline | RegexOptions.Compiled | RegexOptions.IgnoreCase);
            var keysToRemove = new List<string>();
            while (enumerator.MoveNext())
            {
                if (regex.IsMatch(enumerator.Key.ToString()))
                {
                    keysToRemove.Add(enumerator.Key.ToString());
                }
            }

            foreach (string key in keysToRemove)
            {
                items.Remove(key);
            }
        }
```

```
        /// <summary>
        /// Clear all cache data
        /// </summary>
        public virtual void Clear()
        {
            var items = GetItems();
            if (items.IsNull())
                return;

            var enumerator = items.GetEnumerator();
            var keysToRemove = new List<string>();
            while (enumerator.MoveNext())
            {
                keysToRemove.Add(enumerator.Key.ToString());
            }

            foreach (string key in keysToRemove)
            {
                items.Remove(key);
            }
        }
    }
```

下面我们重点讲解一下，整个接口缓存的使用策略。在前面的介绍里，我们可以看到，ActionBase抽象基类里，我们定义一个 : public ICacheManager CacheManager { get; set; } 属性，代码片段如下：

```
    /// <summary>
    /// 核心类，接口 action 抽象类；所有外部实现的接口都需要继承此类
    /// 注意：上送参数和下送数据不支持字典对象数据类型，能够用简易数据类型表达的，尽量用简易数据类型
    /// </summary>
    /// <typeparam name="TRequestDto">
    /// 客户端上传 Data 参数 JSON 反序列化后对应的对象，此对象必须是继承于 RequestDtoBase 类的一个实体类
    /// 命名约定规则为：接口类名+RequestDto。如果需要校验客户端 UserId 和 UserName 是
    /// 否提交，实现类里请继承接口：IRequiredUserIdAndUserName，这样系统框架会在执行前先校验参数的
    /// 准确性，数据约束规则完全兼容命名空间 System.ComponentModel.DataAnnotations 下是所有特性。可
    /// 以方便的在上送参数对象属性上定义特性的方式来进行数据验证
    /// </typeparam>
    /// <typeparam name="TResponseDto">
    /// 输出 ActionResult.Data 对象，就可以是任意的输出 DTO 类型，没有对此类型进行泛型约束
    /// 如果此泛型类型指定为：NullResponseDto，系统自动生成 SDK 开发包的时候，将不会生成返回输出类
    /// 注意:尽量不要将此类型定义成 object 类型，要不框架无法自动生成 SDK 开发包（客户端需要手工进行处理）
    /// 集合类型请尽量使用 List 数据类型方便系统框架自动完成 SDK 输出（注意：不支持字典，字典可以使用集合来替代）
    /// </typeparam>
    public abstract class ActionBase<TRequestDto, TResponseDto> : IAction, IActionFilter, IDisposable
        where TRequestDto : RequestDtoBase, new()
        {
             /// <summary>
             /// API 框架提供的缓存接口，具体实现请在外部实现具体的缓存实现(这里系统框架未实现任何缓存);
             /// 在进行具体的使用过程中，由于有可能会缓存键会引起冲突，建议缓存键使用 this.GetType().FullName 加具体缓存键来实现键的冲突或者在外部定义好预定义的键，供具体实现类里调用
            /// 由于接口层并不知道数据库的实体对象变化，因此建议在接口层使用缓存一般是在预知变化不会太频繁的情况下使用，其他情况下慎用；或者在实体增加，修改，删除的时候，全部做缓存设置，删除操作
            /// </summary>
            public ICacheManager CacheManager { get; set; }
        }
```

而我们的所有Action接口，都是继承了此抽象基类。因此我们可以非常方便的在实现类里进行缓存操作，比如：

```
    /// <summary>
    /// 获取服务器时间
    /// </summary>
    [ActionName("ServerTime.Get")]
    [TestAuthentication]
    public class ServerTimeGetAction : ActionBase<NullRequestDto, string>
    {
        /// <summary>
        ///
        /// </summary>
        private ILogger _logger;

        /// <summary>
        /// 自动注入日志记录器
        /// </summary>
        /// <param name="logger">日志记录器</param>
        public ServerTimeGetAction(ILogger logger)
        {
            this._logger = logger;
        }

        /// <summary>
        ///
        /// </summary>
        /// <returns></returns>
        public override ActionResult<string> Execute()
        {
            //直接使用基类属性
            this.Logger.Debug("使用属性注入 Logger");

            //使用构造函数
```

```
            this._logger.Debug("使用构造函数注入 Logger");

            //由于日志记录器注册成单例，所以属性 Logger 和构造函数的 Logger 是同一个对象（引用）

            //演示使用缓存
            this.CacheManager.Get(this.GetRequestCacheKey(), () => {

                //当前这里可以进行更加复杂的数据业务逻辑,然后返回

                return DateTime.Now.ToString();
            });


            return this.SuccessActionResult(DateTime.Now.ToString());
        }
    }
```

从代码里可以看出我们无需关系具体的缓存实现是说明，我们只要调用CacheManager属性即可非常方便的使用缓存，至于具体实现的是哪个缓存系统，我们可以通过注入的方式来进行。上面的实现仅仅是利用基类公开的缓存接口来实现，但是很多时候我们需要缓存这个接口返回值，上面的例子仅仅是我们缓存了部分片段。当我们需要对这个接口返回进行缓存的时候，可以使用Action特性的方式来进行，此特性为：ActionResultCacheAttribute 特性，继承与：Attribute 类。现在我们看下此特性的定义：

```
    /// <summary>
    /// 缓存特性；单位分钟，当接口使用此特性后，将会在入口进行缓存全局拦截 ActionResult
    /// </summary>
    [AttributeUsage(AttributeTargets.Class, AllowMultiple = false)]
    public class ActionResultCacheAttribute : Attribute
    {
        /// <summary>
        /// 缓存
        /// </summary>
        /// <param name="prefix">缓存前缀（一般管理一组相关联的缓存体系，方便联动删除）</param>
        /// <param name="cacheTime">缓存时间，单位：分钟，默认 30 天</param>
        /// <param name="ignoreUserIdAndUserName">
        /// 根据上送参数对象生成缓存键的时候是否忽略上送用户 ID 和用户名称
        /// 为什么要设置此开关，因为当同一业务的时候，如果客户端上送了用户 ID 和用户名称，那么针对同一接口将会生成 2 个缓存副本，
        /// 这不是我们所期望的，我们期望同一业务的查询（上送参数一致），不同的人看到的都是相同的缓存
        /// 默认是忽略掉当前操作用户业务参数
        /// </param>
        public ActionResultCacheAttribute(string prefix, int cacheTime = 30*24*60, bool ignoreUserIdAndUserName = true)
        {
            this.Prefix = prefix;
            this.CacheTime = cacheTime;
            this.IgnoreUserIdAndUserName = ignoreUserIdAndUserName;
        }

        /// <summary>
        /// 缓存特性；单位分钟，当接口使用此特性后，将会在入口进行缓存全局拦截 ActionResult
        /// 请注意，添加修改接口不要设置缓存特性，要不 Execute()方法将无法执行
        /// </summary>
        /// <param name="cacheTime">缓存时间，单位：分钟</param>
        public ActionResultCacheAttribute(int cacheTime)
            : this(prefix: string.Empty, cacheTime: cacheTime, ignoreUserIdAndUserName: true)
        {
        }

        /// <summary>
        /// 缓存时间，单位：分钟
        /// </summary>
        public int CacheTime { get; private set; }

        /// <summary>
        /// 缓存前缀，配合缓存接口：ICacheManager.RemoveByPattern(string pattern)方法使用
        /// 也即一组相关的业务接口，可以定义相同的缓存前缀，这样在进行缓存操作的时候，可以批量同时移除相关业务缓存
        /// </summary>
        public string Prefix { get; private set; }

        /// <summary>
        /// 是否忽略上送业务参数里的用户身份信息（进行全局缓存，如果不忽略就是单独用户缓存）
        /// 根据上送参数对象生成缓存键的时候是否忽略上送用户 ID 和用户名称
        /// 为什么要设置此开关，因为当同一业务的时候，如果客户端上送了用户 ID 和用户名称，那么针对同一接口将会生成 2 个缓存副本，
        /// 这不是我们所期望的，我们期望同一业务的查询（上送参数一致），不同的人看到的都是相同的缓存
        /// 默认是忽略掉当前操作用户业务参数
        /// </summary>
        public bool IgnoreUserIdAndUserName { get; private set; }
    }
```

定义了2个构造函数，构造函数的说明，上面的类里有详细的说明，我们现在只说明下prefix参数，这个参数非常重要，可以说是整个缓存自动化的关键。我们前面演示的缓存使用列子里有个片段：

```
//演示使用缓存
this.CacheManager.Get(this.GetRequestCacheKey(), () => {

    //当前这里可以进行更加复杂的数据业务逻辑,然后返回

    return DateTime.Now.ToString();
});
```

此片段开有个获取缓存的方法 GetRequestCacheKey()，此方法是系统框架自动生成的，供 Action 接口级缓存使用的，用于生成缓存键，我们先看下此方法的具体实现：

```
/// <summary>
/// 获取当前请求获取缓存键信息，方便重写实现类里直接使用
/// 只要接口名称和提交的数据包不变，生成的那么缓存键就不会变化，因此实现针对不同接口和不同请求数据包进行缓存
/// 具体计算方式为：ActionName + "." + subCacheKey + "." + Units.MD5(Data + Format).ToUpper();
/// </summary>
/// <param name="subCacheKey">同一操作上下文，有可能需要不同的子缓存键；可以增加子缓存键，防止冲突</param>
/// <returns>返回本次请求缓存键</returns>
protected string GetRequestCacheKey(string subCacheKey = "")
{
    return this.RequestContext.GetRequestCacheKey(subCacheKey);
}
```

而此方法有调用了上下文获取接口缓存key的方法：

```
/// <summary>
/// 获取当前请求获取缓存键信息，方便重写实现类里直接使用
/// 只要接口名称+序列化格式+提交的数据包不变，生成的那么缓存键就不会变化，因此实现针对不同接口和不同请求数据包进行缓存
/// 由于是针对字符串进行缓存，因此在提交不同预期序列化返回（XML,JSON）会保存 2 份不同的缓存
/// 注意：这里的缓存键仅仅是针对同一接口同一参数的缓存键，不是针对这个应用的全局缓存键
/// </summary>
/// <param name="subCacheKey">同一操作上下文，有可能需要不同的子缓存键；可以增加子缓存键，防止冲突</param>
/// <returns>返回本次请求缓存键</returns>
public string GetRequestCacheKey(string subCacheKey = null)
{
    //接口描述不能为空
    this.ActionDescriptor.CheckNullThrowArgumentNullException("ActionDescriptor");
    this.DecryptedRequestParams.CheckNullThrowArgumentNullException("DecryptedRequestParams");

    //默认就是最原始的上送参数
    var data = this.DecryptedRequestParams.Data;

    //忽略掉用户名称和编号
    if (this.ActionDescriptor.Cache.IgnoreUserIdAndUserName)
    {
        //IRequestDto requestObj = this.RequestDto as IRequestDto;
        //忽略掉用户信息 KEY 计算方式，先获取到上送的对象，然后获取到所有参数属性字典，在忽略掉用户 ID 和用户名称，再对字典进行 JSON
串行化

        //如果不忽略掉当前接口操作用户，那么接口缓存将成为了私有缓存，即达不到预期使用全局缓存的目的
        data =
            this.RequestDto.GetAttributes()
                .Where(
                    o => !new string[] { "UserId", "UserName" }.Contains(o.Key, StringComparer.OrdinalIgnoreCase))
                .ToList()
                .SerializeObjectToJosn();
    }

    //生成当前请求接口缓存键
    var apiRequestCacheKey = "{0}.{1}.{2}.{3}".With(this.DecryptedRequestParams.ActionName,
        this.ActionDescriptor.Version,
        subCacheKey ?? "_SYS_", //外部不指定子缓存键，就指定系统默认缓存键，这样可以在一定时候，使用匹配模式，批量全部删除系统缓
存等
        MD5.Encrypt(data + this.DecryptedRequestParams.Format).ToUpper());

    //检测是否定义了前缀
    return this.ActionDescriptor.Cache.Prefix.IsNullOrEmpty()
        ? apiRequestCacheKey
        : "{0}.{1}".With(ActionDescriptor.Cache.Prefix, apiRequestCacheKey);
}
```

从此代码可以看出，针对每个接口上送的每个参数，都会生成不同的缓存key,统一接口只要上送的Data参数不一致，那么生成的缓存key就会不同。具体生成的缓存key逻辑为：={Prefix}.{接口名称}.{接口版本号}.{子缓存键}.{MD5(Data+Format)}。这样计算出来的缓存key就会是唯一的。达到Action级别根据上送参数不同进行缓存的可能。

具体的使用方式为：

```
[ActionResultCache("SYS", 15, true)]
public class ServerTimeGetAction : ActionBase<NullRequestDto, string>
{
    //...................
}
```

上面我们介绍了如果将数据保存到缓存，下面我们讲一下如何移除缓存。移除缓存我们同样采取了手工指定移除和定义特性的方式来移除。我们先来讲解一下手工移除，代码片段如下：

```
        /// <summary>
        ///
        /// </summary>
        /// <returns></returns>
        public override ActionResult<string> Execute()
        {
            //移除单个缓存
            this.CacheManager.Remove("xxx");
            //批量移除
            this.CacheManager.RemoveByPattern("_SYS");

            return this.SuccessActionResult(DateTime.Now.ToString());
        }
```

我们可以直接使用Remove或者RemoveByPattern方式来进行自定义的缓存移除。另外一种的利用特性的方式来进行缓存移除，我们可以使用 UnloadCachekeysAttribute 特性来进行移除。我们先看一下此特性定义：

```
/// <summary>
/// 完成 Action.Execute() 操作后，移除指定的缓存键
/// </summary>
[AttributeUsage(AttributeTargets.Class, AllowMultiple = false)]
public class UnloadCachekeysAttribute : Attribute
{
    /// <summary>
    /// 待移除的缓存键（匹配键）
    /// </summary>
    public string[] UnloadCacheKeys { get; private set; }

    /// <summary>
    /// 移除指定匹配模式的缓存键
    /// </summary>
    /// <param name="unloadPrefixCacheKeys">待移除的缓存键（注意此键会采取正则的方式进行匹配，如果愿意，完全可以使用正则表达式）
</param>
    public UnloadCachekeysAttribute(params string[] unloadPrefixCacheKeys)
    {
        this.UnloadCacheKeys = unloadPrefixCacheKeys ?? new List<string>().ToArray();
    }
}
```

此特性带一个 UnloadCacheKeys 参数，我们可以指定对个用户模糊匹配的缓存key，这样，在接口直接完成后，会自动调用 ActionBase里的OnActionExecuted方法，来进行缓存的自动移除。代码片段如下：

```
/// <summary>
/// 核心类，接口 action 抽象类；所有外部实现的接口都需要继承此类
/// 注意：上送参数和下送数据不支持字典对象数据类型，能够用简易数据类型表达的，尽量用简易数据类型
/// </summary>
/// <typeparam name="TRequestDto">
/// 客户端上传 Data 参数 JSON 反序列化后对应的对象，此对象必须是继承于 RequestDtoBase 类的一个实体类
/// 命名约定规则为：接口类名+RequestDto。如果需要校验客户端 UserId 和 UserName 是
/// 否提交，实现类里请继承接口：IRequiredUserIdAndUserName，这样系统框架会在执行前先校验参数的
/// 准确性，数据约束规则完全兼容命名空间 System.ComponentModel.DataAnnotations 下是所有特性。可
/// 以方便的在上送参数对象属性上定义特性的方式来进行数据验证
/// </typeparam>
/// <typeparam name="TResponseDto">
/// 输出 ActionResult.Data 对象，就可以是任意的输出 DTO 类型，没有对此类型进行泛型约束
/// 如果此泛型类型指定为：NullResponseDto，系统自动生成 SDK 开发包的时候，将不会生成返回输出类
/// 注意:尽量不要将此类型定义成 object 类型，要不框架无法自动生成 SDK 开发包（客户端需要手工进行处理）
/// 集合类型请尽量使用 List 数据类型方便系统框架自动完成 SDK 输出（注意：不支持字典，字典可以使用集合来替代）
/// </typeparam>
public abstract class ActionBase<TRequestDto, TResponseDto> : IAction, IActionFilter, IDisposable
    where TRequestDto : RequestDtoBase, new()
{
       /// <summary>
    /// 显式实现下执行后触发的事件，方便以后扩展（系统框架里做一些事情）
    /// </summary>
    /// <param name="actionExecutedContext">拦截器执行上下文</param>
    void IActionFilter.OnActionExecuted(ActionExecutedContext actionExecutedContext)
    {
        #region 先处理下需要清理的缓存键

        if (!this.ActionDescriptor.UnloadCacheKeys.IsNull() && this.ActionDescriptor.UnloadCacheKeys.Length > 0)
        {
            //启动匹配缓存键的方式，删除相关缓存键
            foreach (var unloadCacheKey in this.ActionDescriptor.UnloadCacheKeys)
            {
                try
                {
                    this.CacheManager.RemoveByPattern(unloadCacheKey);
                    this.Logger.Debug("清理缓存匹配键：{0} 成功".With(unloadCacheKey));
                }
                catch (Exception ex)
                {
                    this.Logger.Error(ex);
                }
```

```
            }
        }

        #endregion

        //执行完所有的事件之后
        this.OnActionExecuted(actionExecutedContext);
    }
}
```

从代码可以看出，当我们执行完Action.Execute()方法后，系统框架会自动调用IActionFilter.OnActionExecuted方法，来进行缓存的移除。具体Action接口加移除特性如下列代码所示：

```
[UnloadCachekeys("_FRXS_")]
public class ServerTimeGetAction : ActionBase<NullRequestDto, string>
{
   //.....................
}
```

这样定义后，在执行完接口后，系统后自动清理包含还有 _FRXS_ 关键词key的所有缓存项，从而达到相关性缓存移除。

使用建议：框架自动包含了缓存处理策略，因此我们可以将模块进行分组。定义一个缓存匹配键，我们现在拿分类来说明：首先我们顶一个模糊匹配键。用于所有分类相关项目的缓存设置和清理。代码如下：

```
/// <summary>
/// 基本分类缓存 key
/// </summary>
internal class CategoriesCacheKey
{
    /// <summary>
    /// 此缓存键用户匹配所有相关联的缓存键（用于匹配删除所有相关缓存）
    /// </summary>
    public const string FRXS_CATEGORIES_PATTERN_KEY = "Frxs.Categories";
}
```

当我们Action接口进行Query操作，定义特性缓存的时候加上此prefix前缀。所有分类Command操作如果牵涉到prefix相关的，都使用特性定义需要移除相关缓存键，这样我们可以非常方便的使用框架自动提供的缓存策略。达到任意数据的缓存(比如：列表分页，单项等等)。系统框架缓存策略是站在一个全局的角度来看待缓存数据而不是针对单一实体DTO对象。因此把人工干预的降低到最少。开发人员能够自己定义缓存的粒度而无需关系缓存键的生成和清理。从而是开发人员更加关注与业务的实现而无需关系技术的底层实现。实现所有数据类型的缓存。大大提高接口的处理速度与开发速度。

5.6 如何对Action接口实现拦截(IActionFilter)

前面的讲述 IActionInvoker 接口已经涉及到了拦截器，没有进行详细的讲解，拦截器作用就是，在接口执行IAction.Execute()前后，进行接口的拦截，在执行前可以拦截IAction.Execute()的执行，执行IAction.Execute()后，可以对返回的ResultAction数据进行修改整理，再返回给客户端。我们先来看一下 IActionFilter 定义：

```
/// <summary>
/// 接口执行过滤器接口
/// </summary>
public interface IActionFilter
{
    /// <summary>
    /// 执行方法前；在适当时候进行接口拦截
    /// </summary>
    /// <param name="actionExecutingContext">执行上下文</param>
    void OnActionExecuting(ActionExecutingContext actionExecutingContext);

    /// <summary>
    /// 执行方法后;可以修改接口执行结果
    /// </summary>
    /// <param name="actionExecutedContext">执行上下文</param>
    void OnActionExecuted(ActionExecutedContext actionExecutedContext);
}
```

定义比较简单，只定义2个方法，用于执行前拦截和执行后拦截，系统框架实现了3中方式来进行接口的拦截。

5.6.1 通过Action类特性来定义Action接口拦截器

现在我们来新建一个自定义的特性拦截器，系统框架为了定义拦截器的顺序，针对特性定义的拦截定义了一个抽象基类，所有特性拦截器都需要继承此基类。其实现代码为：

```
/// <summary>
/// IAction.Execute()方法执行前后自定义执行方法；所有自定义进行接口拦截的类，都需要继承此抽象基类，
/// 接口可以附加多个拦截器，按照定义的优先级先后顺序执行
/// </summary>
[AttributeUsage(AttributeTargets.Class, AllowMultiple = true, Inherited = false)]
public abstract class ActionFilterBaseAttribute : Attribute, IActionFilter
{
    /// <summary>
    /// 执行优先级，数字越大优先级越高
```

```
        /// </summary>
        public virtual int Order { get; set; }

        /// <summary>
        /// 执行方法前；在适当时候进行接口拦截
        /// </summary>
        /// <param name="actionExecutingContext">执行上下文</param>
        public virtual void OnActionExecuting(ActionExecutingContext actionExecutingContext) { }

        /// <summary>
        /// 执行方法后;可以修改接口执行结果
        /// </summary>
        /// <param name="actionExecutedContext">执行上下文</param>
        public virtual void OnActionExecuted(ActionExecutedContext actionExecutedContext) { }
    }
```

我们可以看到抽象基类定义一个 Order 优先级属性并且特性的 AllowMultiple 定义为 true，也就是说，这个特性类是可以重复在 Action 接口上定义多个过滤器的，这也接口框架里为数不多的支持多个特性的类。 现在我们新建一个 TestActionFilterAttribute 拦截器，实现代码如下：

```
    /// <summary>
    /// 自定义一个特性拦截器
    /// </summary>
    public class TestActionFilterAttribute : ActionFilterBaseAttribute
    {
        /// <summary>
        /// 执行前拦截
        /// </summary>
        /// <param name="actionExecutedContext"></param>
        public override void OnActionExecuted(ActionExecutedContext actionExecutedContext)
        {
            if (actionExecutedContext.RequestContext.DecryptedRequestParams.Sign.IsNullOrEmpty())
            {
                //直接返回（不会进行后续的执行）
                actionExecutedContext.Result = new ActionResult()
                {
                    Flag = ActionResultFlag.FAIL,
                    Info = "ERROR",
                    Data = null
                };
            }
        }

        /// <summary>
        /// 执行好拦截
        /// </summary>
        /// <param name="actionExecutingContext"></param>
        public override void OnActionExecuting(ActionExecutingContext actionExecutingContext)
        {
            actionExecutingContext.Result.Info = "执行后对数据修改了";
        }

    }
```

另一完成以后，我们就可以将此特性校验器定义到指定的某一个 Action 接口上面，这样就为 Action 接口指定了一个过滤器，代码如下：

```
    [TestActionFilter]
    public class ServerTimeGetAction : ActionBase<NullRequestDto, string>
    {
        //.....................
    }
```

5.6.2 通过全局注册拦截器

在讲解如何进行全局拦截器用法之前，我们先来了解用于保存全局拦截器的管理容器：IGlobalActionFiltersCollection接口，此接口定义如下：

```
/// <summary>
/// 全局接口过滤器集合接口
/// </summary>
public interface IGlobalActionFiltersCollection
{
    /// <summary>
    /// 添加一个全局接口过滤器
    /// </summary>
    /// <param name="actionFilters">过滤器</param>
    void Add(params IActionFilter[] actionFilters);

    /// <summary>
    /// 清空所有过滤器
    /// </summary>
    void Clear();

    /// <summary>
    /// 全局接口过滤器数
    /// </summary>
    int Count { get; }
```

```
        /// <summary>
        /// 移除特定对象的第一个匹配项。
        /// </summary>
        /// <param name="item">继承自 ActionFilterBaseAttribute 的过滤器</param>
        /// <returns></returns>
        bool Remove(IActionFilter item);

        /// <summary>
        /// 获取所有的全局结果过滤器
        /// </summary>
        IEnumerable<IActionFilter> GetActionFilters();
    }
```

系统框架的实现类为 GlobalActionFiltersCollection 类，作为全局拦截器容器，为了实现方便，直接继承自List数据类型。我们再看下具体实现：

```
    /// <summary>
    /// 接口过滤器集合配置表
    /// </summary>
    public class GlobalActionFiltersCollection : List<IActionFilter>, IGlobalActionFiltersCollection
    {
        /// <summary>
        /// 添加一个新的全局拦截器到管理器里面
        /// </summary>
        /// <param name="actionFilters">全局拦截器</param>
        public void Add(params IActionFilter[] actionFilters)
        {
            //不能为 null
            actionFilters.CheckNullThrowArgumentNullException("actionFilters");

            //添加拦截器到集合，排除掉已经添加的
            foreach (var item in from item in actionFilters
                let actionFilter = this.FirstOrDefault(o => o.GetType() == item.GetType())
                where actionFilter.IsNull()
                select item)
            {
                base.Add(item);
            }
        }

        /// <summary>
        /// 获取所有的全局拦截器
        /// </summary>
        public IEnumerable<IActionFilter> GetActionFilters()
        {
            return this;
        }
    }
```

框架里并不是直接使用这个容器来进行全局管理的，而是使用了一个全局拦截器管理器，里面定义了一个静态的容器，公开了一个静态的集合属性来进行对全局拦截器的管理，实现如下：

```
    /// <summary>
    /// 全局拦截器管理器
    /// </summary>
    public class GlobalActionFiltersManager
    {
        /// <summary>
        /// 全局拦截器表
        /// </summary>
        private static readonly GlobalActionFiltersCollection Instance = new GlobalActionFiltersCollection();

        /// <summary>
        /// 返回全局拦截器配置表
        /// </summary>
        public static GlobalActionFiltersCollection Filters
        {
            get
            {
                return Instance;
            }
        }
    }
```

现在我们来讲解下如何实现全局注册，我们先来实现一个全局拦截器类： TestGlobalActionFilter.cs 类，实现代码如下：

```
    /// <summary>
    /// 实现一个全局拦截器
    /// </summary>
    public class TestGlobalActionFilter : IActionFilter
    {
        /// <summary>
        /// 执行前拦截
        /// </summary>
        /// <param name="actionExecutedContext"></param>
        public void OnActionExecuted(ActionExecutedContext actionExecutedContext)
        {
```

```
            actionExecutedContext.RequestContext.HttpContext.Response.Write("{0}.{1}".With(typeof(TestGlobalActionFilter).Name,
"OnActionExecuted"));
             //throw new NotImplementedException();
        }

        /// <summary>
        /// 执行后拦截
        /// </summary>
        /// <param name="actionExecutingContext"></param>
        public void OnActionExecuting(ActionExecutingContext actionExecutingContext)
        {
            actionExecutingContext.RequestContext.HttpContext.Response.Write("{0}.{1}".With(typeof(TestGlobalActionFilter).Name,
"OnActionExecuting"));
            //throw new NotImplementedException();
        }
    }
```

然后我们再Global.asax里使用全局拦截器管理器进行注册，注册代码如下：

```
/// <summary>
///
/// </summary>
public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        //初始化系统框架
        Frxs.ServiceCenter.Api.Core.ApiApplication.Initialize();
        //注册全局拦截器
        Frxs.ServiceCenter.Api.Core.GlobalActionFiltersManager.Filters.Add(new TestGlobalActionFilter());
    }
}
```

通过这样的流程后，我们就注册成功了一个全局拦截器，注册完成后，系统就会指定此全局拦截器。

5.6.3 通过重写Action接口的:protected virtual void OnActionExecuting(ActionExecutingContext actionExecutingContext)和protected virtual void OnActionExecuted(ActionExecutedContext actionExecutedContext)来实现拦截

从前面的描述我们可以知道，Action接口实现里我们可以重写2个方法来进行拦截器。我们还是以前面获取服务器时间接口作为例子，看演示如何进行拦截器执行,实现如下：

```
public class ServerTimeGetAction : ActionBase<NullRequestDto, string>
{
    /// <summary>
    ///
    /// </summary>
    private ILogger _logger;

    /// <summary>
    /// 自动注入日志记录器
    /// </summary>
    /// <param name="logger">日志记录器</param>
    public ServerTimeGetAction(ILogger logger)
    {
        this._logger = logger;
    }

    /// <summary>
    ///
    /// </summary>
    /// <returns></returns>
    public override ActionResult<string> Execute()
    {
        return this.SuccessActionResult(DateTime.Now.ToString());
    }

    /// <summary>
    ///
    /// </summary>
    /// <param name="actionExecutingContext"></param>
    protected override void OnActionExecuting(ActionExecutingContext actionExecutingContext)
    {
        actionExecutingContext.RequestContext.HttpContext.Response.Write("{0}.{1}".With(typeof(ServerTimeGetAction).Name,
"OnActionExecuting"));
        base.OnActionExecuting(actionExecutingContext);
    }

    /// <summary>
    ///
    /// </summary>
    /// <param name="actionExecutedContext"></param>
    protected override void OnActionExecuted(ActionExecutedContext actionExecutedContext)
    {
        actionExecutedContext.RequestContext.HttpContext.Response.Write("{0}.{1}".With(typeof(ServerTimeGetAction).Name,
"OnActionExecuted"));
        base.OnActionExecuted(actionExecutedContext);
```

```
        }
    }
```

说明：拦截器为多协作策略，即：如果即实现了特性拦截，实现了全局拦截，获取实现了Action自定义的拦截器方法重写，那么所有的实现方式都会被执行，执行顺序为：全局最优先，重写方法次之，特性拦截器最后执行

讲解完所有实现拦截器的方式后，我们再来看下框架内部是如何把这几类拦截器进行合并来执行的。在系统框架内部并不是直接使用各个拦截器，而是先把拦截器都包装包一个 `ActionFilterWrapper` 对象。这个对象对拦截器的包装，旨在提供一个统一的接口来进行拦截器排序，器实现代码如下：

```
/// <summary>
/// 过滤器对象包装器，此包装器主要用户过滤器执行顺序封装
/// </summary>
internal class ActionFilterWrapper
{
    /// <summary>
    /// 优先级排序，优先级越高，越先执行，默认值：0
    /// 过滤器优先级
    /// 3 全局注册的过滤器
    /// 2 接口自身实现的过滤器
    /// 1 接口上附加的特性过滤器
    /// </summary>
    public int Order { get; private set; }

    /// <summary>
/// 内部排序;在执行拦截的时候，注意下
/// 1.拦截接口执行方法前，多个拦截器执行顺序为：InternalOrder,Order 高低来排序执行
/// 2.拦截直接执行方法后，多个拦截器执行顺序为：InternalOrder,Order 低到高来执行
/// 即全局拦截器执行优先于特性定义的拦截器，但是执行后执行顺序却是相反的
/// 在内部，这个字段将会被赋值，全局拦截器设置为 int.MaxValue，自定义的接口特性设置为 int.MinValue
/// </summary>
public int InternalOrder { get; private set; }

/// <summary>
/// 过滤器实例
/// </summary>
public IActionFilter ActionFilter { get; private set; }

/// <summary>
/// 在构造函数里进行各种过滤器的排序值生成
/// </summary>
/// <param name="actionFilter">过滤器</param>
public ActionFilterWrapper(IActionFilter actionFilter)
{
    this.ActionFilter = actionFilter;

    //全局过滤器优先级最高
    this.InternalOrder = 3;

    //自身接口过滤器（优先级中等）
    if (actionFilter is IAction)
    {
        this.InternalOrder = 2;
    }

    //接口特性过滤器(优先级最后)
    if (actionFilter is ActionFilterBaseAttribute)
    {
        this.InternalOrder = 1;
        this.Order = (actionFilter as ActionFilterBaseAttribute).Order;
    }
}
```

为了方便代码调用，我们定义了一个针对 `ActionFilterWrapper` 的扩展 `ActionFilterExtensions`，其代码实现很简单，就是封装下，然后返回一个 `ActionFilterWrapper` 对象。

```
/// <summary>
/// 过滤器接口扩展
/// </summary>
internal static class ActionFilterExtensions
{
    /// <summary>
    /// 将过滤器转换成内部过滤器包装类
    /// </summary>
    /// <param name="actionFilter">接口过滤器</param>
    /// <returns>返回过滤器包装类</returns>
    public static ActionFilterWrapper AsActionFilterWrapper(this IActionFilter actionFilter)
    {
        actionFilter.CheckNullThrowArgumentNullException("actionFilter");
        return new ActionFilterWrapper(actionFilter);
    }
}
```

封装处理在默认Action接口执行器 DefaultActionInvoker 里，代码片段为：

```
//先获取到全部的过滤器，包括（接口自身，全局过滤器，特性过滤器）
var actionFilters = new List<ActionFilterWrapper> { ((IActionFilter)action).AsActionFilterWrapper() };

//添加特性过滤器和全局过滤器
actionFilters.AddRange(
       actionDescriptor.ActionFilters.Select(actionFilter => actionFilter.AsActionFilterWrapper()));
```

逻辑是，先将Action接口自身转型成 IActionFilter 接口，然后将特性和全局拦截器加入到集合，后续就是根据拦截器集合进行拦截器的执行。

5.7 本地语言资源包(LanguageResource)

在接口返回值的时候，有时候，我们开发人员定义的返回措辞在实际环境里并不能满足运营的需求，比如：开发人员定义的返回说明可能是比较专业化的，运营人员需要返回更加符合用户可读性和可理解度，但是项目又正式上线，这种情况下去直接修改硬编码的代码，再去发布。不光时间上不划算，还有可能由于不小心出现改错的情况发生，因此就有了本地语言包的概念。有了它我们不必修改代码，而是通过语言包的形式来更改返回说明文字。ActionBase抽象基类里，我们可以看到它定义了一个L的属性，此属性是个委托，旨在返回一个语言包委托，获取特定KEY值对应的语言说明。代码片段如下：

```
public abstract class ActionBase<TRequestDto, TResponseDto> : IAction, IActionFilter, IDisposable
    where TRequestDto : RequestDtoBase, new()
    {
        /// <summary>
        /// 语言资源文件的读取；内部使用 XML 资源文件；
        /// 参数 1 为资源文件的键，自定义的语言资源包请使用接口的 ActionName 来作为节点名称，此委托会自动使用接口名称来构造键；KEY 只要输入对应接口语言文件 key 节点名称即可
        /// 参数 2 为在资源文件找不到的情况下，默认显示的信息；
        /// 返回值为读取到的键值信息；
        /// 资源文件的格式为：请参见：Host/Config/LanguageResource.xml
        /// 调用如：this.L("Exist_S","删除 {0} 出错，当前状态为：{1}" , "001", "已确定")
        /// </summary>
        protected Localizer L
        {
            get
            {
                return (resourceKey, defaultValue, args) =>
                {
                    //获取语言包
                    var languageResourceManager = LanguageResourceManager.Instance;

                    //语言包不存在
                    if (languageResourceManager.IsNull())
                    {
                        return defaultValue ?? string.Empty;
                    }

                    //key 为空，直接返回默认的说明
                    if (resourceKey.IsNullOrEmpty())
                    {
                        return defaultValue ?? string.Empty;
                    }

                    //构造资源的 key 值;（忽略大小写）
                    string key = "{0}".With(resourceKey);

                    //如果没有以接口名称开头，就自动将接口名称附加到前缀
                    if (!resourceKey.StartsWith(this.ActionDescriptor.ActionName, StringComparison.OrdinalIgnoreCase))
                    {
                        key = "{0}.{1}".With(this.ActionDescriptor.ActionName, resourceKey);
                    }

                    //获取资源
                    string value = languageResourceManager.GetLanguageResourceValue(key, defaultValue);

                    //含有参数就进行格式化
                    return args.IsNull() || args.Length == 0 ? value : value.With(args);
                };
            }
        }
```

从代码可以看出，此属性返回的是一个 Localizer 委托，内部通过构造出一个匿名委托来返回我们需要的委托对象，我们先来看一下定义：

```
namespace Frxs.ServiceCenter.Api.Core
{
    /// <summary>
    /// 语言包读取委托
    /// </summary>
    /// <param name="resourceKey">语言包键(key)</param>
    /// <param name="defaultValue">指定语言包键不存在，返回默认值</param>
    /// <returns>返回指定键的语言包</returns>
    public delegate string Localizer(string resourceKey, string defaultValue, params object[] args);
```

```
}
```

参数resourceKey为：语言包键key值，我们拿上面的语言包来说，调用ERR_01对应的Value值，键应该为：ServerTime.Get.ERR_01。但是我们基类ActionBase属性L返回匿名委托里已经做了处理，只传入key键，不传ActionName也可以。

匿名委托内部使用了语言包资源器，用于保存，获取语言。具体的逻辑比较简单，就是通过反序列化XML文件，获得语言包对象。然后根据KEY值，获取对应的Value。代码如下：

```
    /// <summary>
    /// ACTION 自定义接口使用的语言包对象管理器
    /// </summary>
    public class LanguageResourceManager
    {
        /// <summary>
        /// 用于缓存，忽略大小写
        /// </summary>
        private readonly IDictionary<string, string> _languageResourceKeyValues = new Dictionary<string,
string>(StringComparer.OrdinalIgnoreCase);
        private static readonly object Locker = new object();
        private static LanguageResourceManager _instance;

        /// <summary>
        /// 获取接口语言包管理对象
        /// </summary>
        public static LanguageResourceManager Instance
{
    get
    {
        if (!_instance.IsNull())
        {
            return _instance;
        }
        lock (Locker)
        {
            if (_instance.IsNull())
            {
                _instance = new LanguageResourceManager(SystemOptionsManager.Current.LanguageResourcePath);
            }
        }
        return _instance;
    }
}

/// <summary>
/// ACTION 自定义接口使用的语言包对象管理器
/// </summary>
/// <param name="languageResourcePath">语言包路径</param>
private LanguageResourceManager(string languageResourcePath)
{
    this.Init(this.GetLanguageResource(languageResourcePath));
}

/// <summary>
/// 获取用户自定义的语言资源包
/// </summary>
/// <param name="languageResourcePath">语言资源包路径</param>
/// <returns>获取 xml 语言包对象，当语言包路径不存在情况下，返回 null</returns>
private LanguageResource GetLanguageResource(string languageResourcePath)
{
    //语言包路径不存在直接返回空的集合
    if (languageResourcePath.IsNullOrEmpty() || !File.Exists(languageResourcePath))
    {
        return null;
    }
    LanguageResource languageResource = null;
    //读取语言资源包
    using (var streamReader = new StreamReader(languageResourcePath))
    {
        try
        {
            languageResource = (LanguageResource)new XmlSerializer(typeof(LanguageResource)).Deserialize(streamReader);
        }
        catch
        {
            // ignored
        }
    }

    return languageResource;
}

/// <summary>
/// 初始化语言包对象获取对应的字典
/// </summary>
private void Init(LanguageResource languageResource)
{
    if (languageResource.IsNull() || languageResource.Actions.IsNull())
    {
        return;
```

```
        }

        foreach (var action in languageResource.Actions)
        {
            if (action.Items.IsNull() || action.Items.IsEmpty())
            {
                continue;
            }

            foreach (var item in action.Items)
            {
                //获取键信息
                string key = "{0}.{1}".With(action.Name, item.Key);

                //已经存在键，直接抛出异常，防止在正式环境出现错误
                if (this._languageResourceKeyValues.ContainsKey(key))
                {
                    this._languageResourceKeyValues.Clear();
                    throw new ApiException("语言资源包存在相同的键，键：{0},值：{1}".With(key, item.Value));
                }
                this._languageResourceKeyValues.Add(key, item.Value);
            }
        }
    }

    /// <summary>
    /// 完整的键值名称
    /// </summary>
    /// <param name="fullKey">资源键：完整的键名称，比如：Api.Core.System.ERR_01</param>
    /// <param name="defaultValue">资源文件不存在对应的键，就直接返回 defaultValue</param>
    /// <returns>从字典里返回指定键的值，不存在就返回默认值 defaultValue</returns>
    public string GetLanguageResourceValue(string fullKey, string defaultValue)
    {
        //没有指定键
        if (fullKey.IsNullOrEmpty())
        {
            return defaultValue ?? string.Empty;
        }

        //不存在指定的键
        if (!this._languageResourceKeyValues.ContainsKey(fullKey))
        {
            return defaultValue ?? string.Empty;
        }

        //返回指定的键值
        return this._languageResourceKeyValues[fullKey];
    }
}
```

下面我们来看下在接口里怎么使用语言包功能，我们还是以获取服务器时间接口来做说明：

```
public class ServerTimeGetAction : ActionBase<NullRequestDto, string>
{
    /// <summary>
    ///
    /// </summary>
    private ILogger _logger;

    /// <summary>
    /// 自动注入日志记录器
    /// </summary>
    /// <param name="logger">日志记录器</param>
    public ServerTimeGetAction(ILogger logger)
    {
        this._logger = logger;
    }

    /// <summary>
    ///
    /// </summary>
    /// <returns></returns>
    public override ActionResult<string> Execute()
    {
        //获取定义的语言包
        var info = this.L("ERR_01", "商品：{0}不存在", typeof(ServerTimeGetAction).Name);

        //返回
        return this.SuccessActionResult(info, DateTime.Now.ToString());
    }
}
```

然后我们来看一下外部配置文件的定义：外部资源包为一个XML格式的文件，每个Action接口一个节点，这样我们每个KEY定义到如下配置文件，这样系统框架会自动默认忽略掉代码里硬编码的默认值，而使用配置文件里的ERR_O1错误消息。

```
<?xml version="1.0"?>
<resource>
    <actions>
        <action name="ServerTime.Get">
            <items>
                <item key="ERR_01" value="商品编号:{0}不存在，请检查" />
                <item key="ERR_02" value="提交的数据格式错误，请检查" />
                <item key="ERR_03" value="提交的数据格式错误，请检查" />
                <item key="ERR_04" value="提交的数据格式错误，请检查" />
            </items>
        </action>
    </actions>
</resource>
```

5.8 外部服务模块如何使用资源查找管理器/如何扩展自己的资源查找器(IResourceFinderManager)

在前面的流程描述里，我们讲解了资源管理器的实现，以及系统框架里默认的2个资源查找器(AssemblyResourceFinder,LocalFileViewResourceFinder)，现在我们来自己扩展一个自己定义的资源查找器，由前面的章节我们知道，实现自己资源查找器需要实现 IResourceFinder 接口。此接口定义也非常简单，我们先看下接口定义：

```
/// <summary>
/// 所有接口所在程序集视图文件查找器（注意此接口只找文本类型的，比如：js,css,aspx,asp,cshtml 等）
/// 此接口属于协作接口，即：注册多个资源查找器系统会依次在各个查找器里进行资源查找
/// </summary>
public interface IResourceFinder
{
    /// <summary>
    /// 优先级排序，数字越大，优先级越高
    /// </summary>
    int Order { get; }

    /// <summary>
    /// 返回所有程序集内嵌资源信息；此方法的实现最好能够进行缓存机制，即第一次加载到时候描述所有程序集，后续直接从缓存里读取
    /// key:资源文件名称
    /// value:资源文件源代码
    /// </summary>
    /// <returns></returns>
    IDictionary<string, string> GetResources();

    /// <summary>
    /// 获取到资源文件原始文本
    /// </summary>
    /// <param name="resourceViewFullPath">内嵌资源路径</param>
    /// <returns>内嵌资源原始文件(找不的将返回 null，所以调用的时候需要注意下 null 情况)</returns>
    string GetResource(string resourceViewFullPath);
}
```

现在我们来实现一个固定值的资源查找器。首先我们先新建一个名叫 ： TestResourceFinder.cs 的类。让后让其继承IResourceFinder接口。代码如下：

```
/// <summary>
/// 实现固定的自定义查找器
/// </summary>
public class TestResourceFinder : IResourceFinder
{
    /// <summary>
    /// 让其优先于系统框架，优先级最高
    /// </summary>
    public int Order
    {
        get
        {
            return int.MaxValue;
        }
    }

    /// <summary>
    /// 根据指定的 key 获取资源值
    /// </summary>
    /// <param name="resourceViewFullPath"></param>
    /// <returns></returns>
    public string GetResource(string resourceViewFullPath)
    {
        var keyValue = this.GetResources().FirstOrDefault(o => o.Key == resourceViewFullPath);
        if (keyValue.IsNull() || keyValue.Key.IsNullOrEmpty())
        {
            return null;
        }
        return keyValue.Value;
    }

    /// <summary>
```

```
        /// 获取当前资源查找器所有的键-值对
        /// </summary>
        /// <returns></returns>
        public IDictionary<string, string> GetResources()
        {
            return new Dictionary<string, string>() {
                { "Test.001", "test001"},
                { "Test.002", "test002"},
                { "Test.003", "test003"},
            };
        }
    }
```

然后我们将此资源查找器注册到IOC容器，注册代码如下：

```
    /// <summary>
    /// API 框架会自动检测到这里的注册类,自动完成注册
    /// </summary>
    public class DependencyRegistar : IDependencyRegistar
    {
        /// <summary>
        /// 系统框架默认的会被覆盖;
        /// </summary>
        /// <param name="containerBuilder"></param>
        /// <param name="typeFinder">类型查找器</param>
        public void Register(ContainerBuilder containerBuilder, ITypeFinder typeFinder)
        {
            containerBuilder.RegisterType<TestResourceFinder>()
                .AsImplementedInterfaces()
                .SingleInstance();       }

        /// <summary>
        /// 数字越大越后注册
        /// </summary>
        public int Order
        {
            get { return 1; }
        }
    }
```

这样我们就完成了一个自定义的资源查找器，当我们再使用资源查找管理器获取资源：Test.001 的时候，会优先使用刚刚我们自定义的查找器。返回："test001" 值。

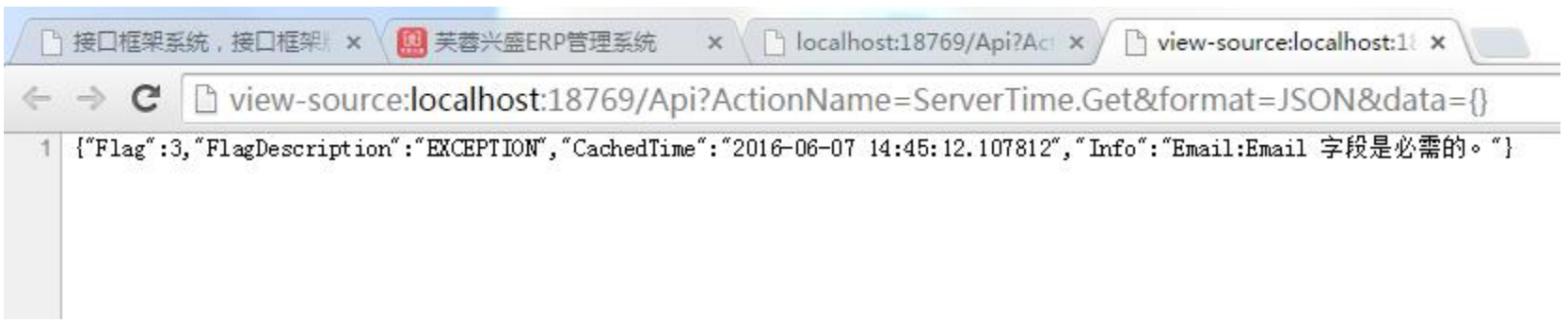
5.9 如何自定义一个RequestDto参数校验特性(ValidationAttribute)

当我们使用反序列化，将客户端上送的业务参数Data数据JSON字符串反序列化成RequestDto后，后端使用的时候，我们很多时候都需要判断下客户是否真实上送了正确的参数。比如：联系电话是否提交？联系电话格式是否准确？邮件是否必须提交？邮件格式是否准确？如果我们使用手工的方式，一个参数去校验的话，那将是一件很枯燥，重复的事情，也不利于代码的复用。因此接口框架针对RequestDto的属性专门做了一个校验器。这样我们只要在RequestDto属性上加上特定特性标签就可以实现数据的校验。我们来看一下简单的例子：

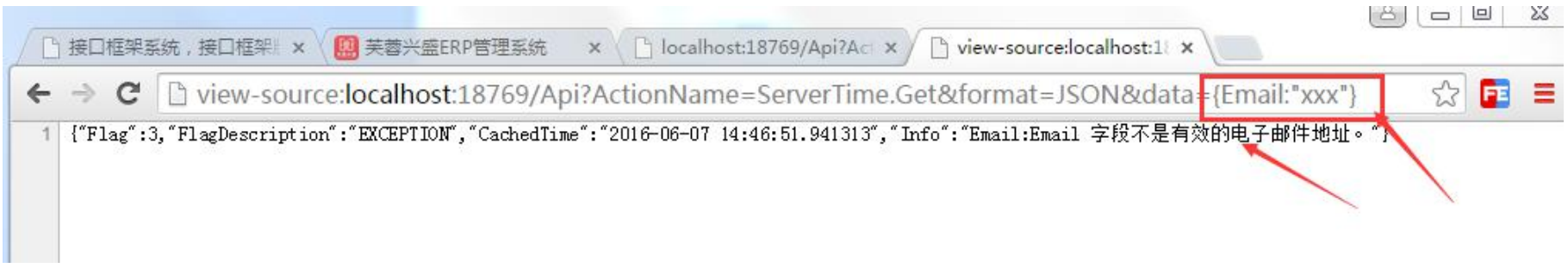
```
    /// <summary>
    /// 获取服务器时间
    /// </summary>
    [ActionName("ServerTime.Get")]
    public class ServerTimeGetAction : ActionBase<ServerTimeGetAction.ServerTimeGetActionRequestDto, string>
    {
        /// <summary>
        /// 上送参数对象
        /// </summary>
        public class ServerTimeGetActionRequestDto : RequestDtoBase
        {

            /// <summary>
            /// 电子邮件
            /// </summary>
            [Required, EmailAddress]
            public string Email { get; set; }

            /// <summary>
            /// 校验前给个机会修改数据
            /// </summary>
            public override void BeforeValid()
            {
                base.BeforeValid();
            }

            /// <summary>
            /// 非常复杂的业务校验，交给开发手工去指定
            /// </summary>
            /// <returns></returns>
            public override IEnumerable<RequestDtoValidatorResultError> Valid()
            {
                //比如数据必须在某个指定范围，或者年龄大于小于摸个范围，等等

                return base.Valid();
            }
```

```
        }

        /// <summary>
        ///
        /// </summary>
        public ServerTimeGetAction()
        { }

        /// <summary>
        ///
        /// </summary>
        /// <returns></returns>
        public override ActionResult<string> Execute()
        {
            return this.SuccessActionResult(DateTime.Now.ToString());
        }
    }
```

我们将上送的参数对象定义为成了一个内部类（当前此类放置于哪个地方不是非常重要，我们这里为看方便理解和查看所以定义成了内部内），我们看到**Email**属性我们定义成了必须填写并且格式必须是邮件格式。现在我们保存上面代码，然后运行下程序，首先我们**Data**参数上面都不传：



然后提交**Email**参数，但是格式不准确，看返回什么信息：



通过上面行为我们可以看到我们仅仅只是定义属性特性，就能实现参数准确性的校验，这样我们再开发的时候就非常方便。系统框架属性校验完全兼容：**System.ComponentModel.DataAnnotations** 命名空间下的所有特性校验类。另外，系统框架里还扩展了**5**个参数校验类：

1. ***GreaterThanAttribute（数字必须大于指定值）***
2. ***InAttribute（输入的值必须包含在指定的范围值域内）***
3. ***NotEmptyAttribute（集合元素不能为空(即：元素必须大于1个)***
4. ***NotEqualAttribute（比较指定属性值，检测是否与当前属性相等）***
5. ***NotNullAttribute（可为空的值类型，或者object不能为null，如果需要使字符串类型不为null，请使用RequiredAttribute特性）***

那么我们如何扩展 *System.ComponentModel.DataAnnotations* 校验行为呢？我们拿**InAttribute**这个扩展来看。实现所有的校验扩展必须继承 **ValidationAttribute** 抽象类，然后重写 **IsValid(object value)** 方法或者 **IsValid(object value, ValidationContext validationContext)** 方法即可，我们来先下代码：

```
    /// <summary>
    /// 输入的值必须包含在指定的范围值域内
    /// </summary>
    public class InAttribute : ValidationAttribute
    {
        /// <summary>
        /// 值域范围
        /// </summary>
        public object[] Values { get; private set; }

        /// <summary>
        ///
        /// </summary>
        /// <param name="values">值域范围，比如：[1,2,3,4]或者["z","x","Y"]</param>
        public InAttribute(params object[] values)
        {
            values.CheckNullThrowArgumentNullException("values");
            this.Values = values;
            this.ErrorMessage = "值取值范围为:{0}".With(string.Join(",", values));
        }
```

```
        /// <summary>
        ///
        /// </summary>
        /// <param name="value"></param>
        /// <returns></returns>
        public override bool IsValid(object value)
        {
            return value.IsNull() || this.Values.Contains(value);
        }
    }
```

这样我们就扩展了一个属性值必须在指定范围内的校验扩展，根据上面的规则，我们可以扩展出各种各样自己需要的数据校验。

从上面的ServerTime.Get接口还可以看出，我们上送参数对象里还有2个可以重写的方法，此2个方法定义在RequestDtoBase里 `void BeforeValid()` 用于在执行校验前，开发人员有机会修改参数值或者对参数重新指定一个值。
`IEnumerable<RequestDtoValidatorResultError> Valid()` 方法用户在特性校验无法满足的情况下使用，这样框架参数校验器执行的时候，会先校验特性校验器，然后在校验手工指定的校验。如果都通过校验了，才能继续往下执行接口。为了更加深刻的理解下整个校验流程，我们来看一下校验器的实现：

```
/// <summary>
/// 如果一个 RequestDto 实现了 IRequestDtoValidatable 接口，将可以使用此验证器来验证业务数据的准确性
/// 特性验证器完全兼容：System.ComponentModel.DataAnnotations 命名空间特性验证(特性生效一定需要
/// 上送参数类实现 IRequestDtoValidatable 接口，否则就算定义了校验特性，也不会生效)
/// </summary>
public class RequestDtoValidator
{
    /// <summary>
    /// requestDto 参数上送对象
    /// </summary>
    /// <param name="requestDto">requestDto 参数上送对象</param>
    /// <param name="actionDescriptor">接口描述对象</param>
    public RequestDtoValidator(RequestDtoBase requestDto, ActionDescriptor actionDescriptor)
    {
        this.RequestDto = requestDto;
        this.ActionDescriptor = actionDescriptor;
        this.ValidatableObject = requestDto as IRequestDtoValidatable;
    }

    /// <summary>
    /// 当前输入的 RequestDto 对象
    /// </summary>
    public RequestDtoBase RequestDto { get; private set; }

    /// <summary>
    /// 接口描述对象
    /// </summary>
    public ActionDescriptor ActionDescriptor { get; private set; }

    /// <summary>
    /// 转型后的 RequestDto 对象，有可能为 null
    /// </summary>
    public IRequestDtoValidatable ValidatableObject { get; private set; }

    /// <summary>
    /// 验证实体数据正确性，返回 RequestDtoValidatorResult 对象
    /// </summary>
    public RequestDtoValidatorResult Valid()
    {
        //如果上送对象实现了 IRequestDtoValidatable 或者 IRequiredUserIdAndUserName 就进行校验
        if (this.ValidatableObject.IsNull() && !(this.RequestDto is IRequiredUserIdAndUserName))
        {
            return new RequestDtoValidatorResult();
        }

        //用于保存验证集合
        var validationResultErrors = new List<RequestDtoValidatorResultError>();

        //校验器给实现类一个改变参数属性值的机会
        this.ValidatableObject.BeforeValid();

        //1.校验定义在参数对象上的特性校验
        var validationContext = new ValidationContext(this.RequestDto);
        var results = new List<ValidationResult>();
        if (!Validator.TryValidateObject(this.RequestDto, validationContext, results, true))
        {
            results.ForEach(o =>
            {
                validationResultErrors.Add(new RequestDtoValidatorResultError(
                    string.Join(",", o.MemberNames.ToArray()), o.ErrorMessage));
            });
        }

        //没有通过校验直接返回(防止在业务参数校验的时候，直接取值造成异常抛出)
        if (!validationResultErrors.IsEmpty())
        {
            return new RequestDtoValidatorResult(validationResultErrors);
        }
```

```
            //定义了自定义验证接口
            if (!this.ValidatableObject.IsNull())
            {
                //2.进行自定义的数据校验器
                var validationResults = this.ValidatableObject.Valid();

                //校验自定义业务数据是否正确
                validationResultErrors.AddRange(validationResults);
            }

            //3.校验用户名和用户 ID 或者在接口定义了需要校验的特性
            if (this.RequestDto is IRequiredUserIdAndUserName || this.ActionDescriptor.RequiredUserIdAndUserName)
            {
                //验证用户 ID 和用户名称的委托为空弹出消息
                if (SystemOptionsManager.Current.ValidUserIdAndUserNameFun.IsNull())
                {
                    validationResultErrors.Add(new RequestDtoValidatorResultError("ValidUserIdAndUserNameFun",
                        Resource.CoreResource.ActionBase_ValidUserIdAndUserNameFun_Null_Error));
                }
                //不为空就开始举行校验
                else
                {
                    //检测用户 ID 和用户名称是否提交
                    if (!SystemOptionsManager.Current.ValidUserIdAndUserNameFun(this.RequestDto))
                    {
                        validationResultErrors.Add(new RequestDtoValidatorResultError("UserId,UserName",
                            Resource.CoreResource.ActionBase_RequiredUserIdAndUserName_Error));
                    }
                }

                //返回校验错误集合
                return new RequestDtoValidatorResult(validationResultErrors);
            }

        }
```

除了可以使用系统自带的使用特性方式来进行数据校验外，系统框架还扩展了一个基于FluentValidation校验组件的项目，项目为：1.5.1Frxs.ServiceCenter.Api.Core.FluentValidation 。

5.10 接口版本快速迭代以及接口版本控制（Version）

当我们一个接口开发完成后，项目已经上线。这个时候，如果我们想对现有接口进行升级改造，但是我们不能下线现有的Action接口，而是要以增量的方式，来覆盖现有接口。这个时候接口版本控制就显得尤为重要。接口框架系统提供了2种版本控制。着需要我们再实际开发种根据情况来选择。

1.让客户端指定接口版本。还记得我们接受参数对象 RequestParams 有个Version属性吗？这个属性就是让客户端，来选择同名接口的哪个版本。
2.客户端不指定版本号，让服务器自动选择最新版本的版本号（或者服务器指定接口版本号，服务器可以通过配置文件来下线接口，禁用接口）

具体的使用，需要根据项目需要来选择。这里不做多说明。代码的实现我们可以看一下 IActionSelector 接口的实现类：DefaultActionSelector 。

我们现在来说明下，我们如何制定接口的版本，当我们写Action接口的时候，如果未指定 VersionAttribute 特性，那么系统框架默认接口版本为 0.0 ，如果我们指定了接口版本，那么就会返回我们指定的版本。具体使用看下面代码：

```
/// <summary>
/// 获取服务器时间
/// </summary>
[ActionName("ServerTime.Get")]
[Version(1,0)]
public class ServerTimeGetAction : ActionBase<ServerTimeGetAction.ServerTimeGetActionRequestDto, string>
{
    //.....................
}
```

上面代码片段，我们指定了 ServerTime.Get 接口版本为1.0，注意一个同名的接口，可以有不同的版本。

5.11 IP白名单

考虑到接口的安全性，我们很多接口可能只想外部固定的一些IP客户端才能访问，除了IP白名单外的所有IP地址都禁止访问。这样的需求我们可以通过框架的白名单功能来限制。为了更加好的理解白名单的功能，我们先来看下实现，然后我们来说一下怎么去使用。首先看下实现，实现很简单，保存白名单的使用了集合：

```
/// <summary>
/// 白名单系统配置表
/// </summary>
public class WhiteIpCollection : Collection<string>
{
```

```
        /// <summary>
        /// 添加一批白名单
        /// </summary>
        /// <param name="ips">白名单</param>
        public void Add(params string[] ips)
        {
            if (ips.IsNull())
            {
                return;
            }
            foreach (var ip in ips.Where(ip => !this.Contains(ip)))
            {
                base.Add(ip);
            }
        }

        /// <summary>
        /// 删除一批白名单
        /// </summary>
        /// <param name="ips">白名单</param>
        public void Remove(params string[] ips)
        {
            if (ips.IsNull())
            {
                return;
            }
            foreach (var item in ips)
            {
                base.Remove(item);
            }
        }

        /// <summary>
        /// 检测指定 IP 是否有权限访问接口系统
        /// </summary>
        /// <param name="ip">待检测 IP 地址</param>
        /// <returns>IP 地址是否在白名单里</returns>
        public bool IsValid(string ip)
        {
            //设置了白名单，需要判断是否在定义的白名单里面
            return 0 == this.Count || this.Contains(ip);
        }
    }
```

然后通过一个 `WhiteIpManager` 管理器来进行白名单的管理：

```
    /// <summary>
    /// 白名单集合；一旦定义了白名单，那么只能在白名单里面的 IP 地址才能访问，如果未定义，那么全部 IP 都可以访问
    /// 一般配置在 Global.asax 文件里，应用程序启动的时候就加载，在运行时，最好不要添加白名单，可能会涉及到并发问题
    /// </summary>
    public class WhiteIpManager
    {
        /// <summary>
        /// 将白名单保存在静态全局缓存里
        /// </summary>
        private static readonly WhiteIpCollection Instance = new WhiteIpCollection();

        /// <summary>
        /// 返回接口配置表
        /// </summary>
        public static WhiteIpCollection Ips
        {
            get
            {
                return Instance;
            }
        }
    }
```

知道了实现后，假设我们只允许 `192.168.0.23` 这个IP来访问我们的服务，我们再Global.asax里进行白名单配置：

```
    /// <summary>
    ///
    /// </summary>
    public class MvcApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            //初始化系统框架
            Frxs.ServiceCenter.Api.Core.ApiApplication.Initialize();

            //注册白名单
            Frxs.ServiceCenter.Api.Core.WhiteIpManager.Ips.Add("192.168.0.32");
        }
    }
```

注意：如果我们未配置白名单，默认任意IP客户端都可以访问。

5.12 使用配置文件来控制接口行为

现在我们来说一下使用配置文件来控制接口行为。配置文件不是系统框架必须的。但是有时候，我们需要在正式环境里去修改接口行为，比如：下线，禁用，设置缓存，清理缓存等。我们再不更改源代码的情况下，使得我们运维有方法去做更改，这个功能对于我们运维来说非常重要。还记得文档最开始我们说的~/ApiConfig.cs文件吗？现在我们重点来说一下这个文件的作用。此文件实现为一个C#类文件，需要注意的是，我们在开发的时候，将此文件：属性->生成操作：编译(使用cs文件而不用XML配置文件，出发点是可以利用VS的提示来写配置，而无需记忆XML文件节点)。我们在发布项目的时候，将此文件属性->生成操作：内容，这样此文件发布的时候就不会被打包到dll里。我们现在来看一下此文件的内容：

```
/// <summary>
/// 动态编译并注册接口框架配置
/// 注意：此文件需要单独放置于网站根目录；系统框架将会在启动的时候自动进行注册配置；
/// 开发的时候此文件生成操作设置成：编译；发布的时候，请将此文件生成操作设置为：内容
/// </summary>
public class ApiConfig : IDynamicCompiledDependencyRegistar
{
    /// <summary>
    /// 注册各种需要初始化的数据
    /// </summary>
    public void Register()
    {
        // 1.注册全局配置信息
        SystemOptionsManager.Current = new SystemOptions()
        {
            // 是否开启接口访问记录器，此开关为全局，不管是否实现了具体的接口访问记录器，如果此开关关闭，都将无法记录(系统框架默认开启
false)
            EnableAccessRecorder = true,

            // 服务器名称；正式环境里，请将不同的 API 服务器名称分别配置，便于在分布式接口中跟踪调试服务器
            ServerName = string.Format("frxs-server-{0}", new System.Random().Next(1, 100)),

            // ACTION 项目语言资源包地址(可以为空，系统将使用默认的语言包，如果配置了，就必须保证文件是存在的)
            LanguageResourcePath = System.Web.HttpContext.Current.Server.MapPath("~/Config/LanguageResource.xml"),

            // 接口描述文档(针对多个接口项目分开开发的情况下，请指定多个注释文档即可，正式环境里可以将此配置标注掉)
            ActionDocResourcePaths = new string[] {
                //请注意，文件必须存在，要不框架调用会出现，如果不存在，可以标注掉此配置即可
                System.Web.HttpContext.Current.Server.MapPath("~/bin/Frxs.ServiceCenter.Api.Host.XML"),
                System.Web.HttpContext.Current.Server.MapPath("~/config/Frxs.Erp.ServiceCenter.ID.Actions.XML"),
                System.Web.HttpContext.Current.Server.MapPath("~/config/Frxs.Erp.ServiceCenter.Product.Actions.XML"),
                System.Web.HttpContext.Current.Server.MapPath("~/config/Frxs.Erp.ServiceCenter.Product.Model.XML"),
                System.Web.HttpContext.Current.Server.MapPath("~/config/Frxs.ServiceCenter.Api.Core.XML")
            },

            // 配置 SDK 开发包命名空间
            SdkNamespace = "Frxs.ServiceCenter.Api.SDK.V20",

            // 检测用户 ID 和用户名称是否提交(框架里只定义接口，具体不实现。交给外部配置来检测用户名称和用户 ID 是否提交了)
            ValidUserIdAndUserNameFun = (requestDto) =>
            {
                //皮之不存毛将焉附?
                if (requestDto.IsNull())
                {
                    return false;
                }
                return 0 <= requestDto.UserId && !string.IsNullOrWhiteSpace(requestDto.UserName);
            }
        };

        // 自定义数据（其他第三方扩展项目可以进行数据的配置）
        SystemOptionsManager.Current.AdditionDatas.Add("Api-System-StartTime", DateTime.Now);

        // 比如现在有个针对接口框架扩展出来的项目，需要进行数据配置，可以进行如下配置
        //SystemOptionsManager.Current.AdditionDatas.Add("System-{AppName}-{ParamName}", "参数值");

        // 2.注册 IP 白名单(正式环境根据安全性需要来进行配置)
        //WhiteIpManager.Ips.Add("192.168.0.11", "192.168.3.24", "192.168.2.20", "192.168.3.122");

        // 3.注册全局拦截器
        //GlobalActionFiltersManager.Filters.Add(new
GlobalRequestHanderInterceptors.DefaultGlobalRequestHanderInterceptor());

        // 4.如果需要给接口配置不同的策略，可以按照下面方式来配置注意先后顺序，后注册的配置文件会覆盖掉前面注册的属性配置
        ApiConfigManager.Configs

        .Register(new ActionConfigItem()
        {
            //框架级别全局接口配置，全部接口都会生效，同时会覆盖掉对应接口框架默认的参数设置(兜底设置)

            //缓存过期时间设置（>0 为启用缓存，时间单位为：分钟）
            //CacheTime                    = 0,

            //是否允许 Ajax 访问
            //EnableAjaxRequest             = false,

            //允许客户端提交方式
            //HttpMethod                   = HttpMethod.POST | HttpMethod.GET,
```

```
    //接口是否过期
    //Obsolete                    = false,

    //是否需要安全连接访问
    //RequireHttps                = false,

    //是否进行全局校验
    //AllowAnonymous              = true,

    //是否允许打包到 SDK
    //CanPackageToSdk             = true,

    //是否允许记录日志
    //EnableRecordApiLog          = true,

    //上送参数是否需要进行用户 ID 和用户名称校验
    //RequiredUserIdAndUserName     = false
})

// 接口注销(正式环境下需要将将界面显示的接口注销掉)
    // .Obsolete("API.BuildSDK", "Api.Doc", "Api.Doc.Builder", "API.Logs.Get", "API.Logs.List", "API.TestTool")
    // .Obsolete("API.BuildSDK", "1.0")
    // .ObsoleteSystemActions()

.Register("API.ServerTime.Get", new ActionConfigItem()
{
    // 单一接口未指定版本全局配
    AllowAnonymous = true,
    EnableAjaxRequest = false,
    EnableRecordApiLog = true,
    HttpMethod = HttpMethod.POST | HttpMethod.GET,
    Obsolete = false,
    RequireHttps = false
})

// 匿名方式进行注册，只要匿名对象属性与配置对象参数一致既可(参数不区分大小写)
.Register("API.ServerTime.Get", new { RequireHttps = false, CacheTime = 10 })

//接口分组
.Group("API.ServerTime.Get", "公共接口")

// 注册 1.0
.Register("API.ServerTime.Get", "1.0", new ActionConfigItem()
{
    //是否忽略掉操作用户
    HttpMethod = HttpMethod.POST | HttpMethod.GET

    // 1.0 版本值配置了缓存配置，但是系统框架会自动将此属性赋值到此接口未指定版本号的配置上面，
    // 这样此接口配置的真实配置为

    // ********************************************************
    // AllowAnonymous      = true
    // CacheTime           = 100
    // EnableAjaxRequest    = false
    // EnableRecordApiLog   = true
    // HttpMethod          = HttpMethod.POST | HttpMethod.GET
    // Obsolete            = false
    // RequireHttps         = false
    // ********************************************************

})
        .Register("API.ServerTime.Get", "1.0", new ActionConfigItem()
        {
            // 特定版本配置会覆盖全局配置
            RequiredUserIdAndUserName = false,

            //不走加解密流程
            DataSignatureTransmission = false,

            //执行完事件后，会自动执行此缓存键清理
            UnloadCacheKeys = new[] { "API.Logs" }
        })

        .Register("API.ServerTime.Get", "1.1", new ActionConfigItem()
        {
            // 特定版本配置会覆盖全局配置
            RequireHttps = false,

            // 配置前缀
            CachePrefix = "API.ServerTime",

            //忽略掉操作用户
            CacheKeyIgnoreUserIdAndUserName = true,

            // 重新配置缓存为 10 分钟
            CacheTime = 10
        });
    }
}
```

上面的配置是是非完整的配置，注意的是，此文件内容必须是一个合法的C#类，并且需要实现 IDynamicCompiledDependencyRegistar 接口。系统框架在启动的时候，会自动对此文件读取->然后进行动态编译。然后在执行里面的 void Register() 方法，进行系统注册。

5.13 如何实现接口与接口之间调用（不走http流程）

从Action接口的定义可以看出，其实现了IAction接口，Execute() 都是返回了统一的 ActionResult 对象。这就使得我们可以将接口当成一个单独的服务类，当某些接口实现了的功能，另外一个接口需要同样的业务逻辑。我们可以不重复定义，而直接使用已经定义的接口。比如项目 1.1.2.Frxs.ServiceCenter.Api.Core.SdkBuilder.CSharp 里，有个生成接口文档的功能，当我们再界面点击接口名称的时候，查看的是单独的接口文档，但是我们需要批量生成的时候，里面的业务逻辑一模一样，仅仅多的是，后面的批量接口调用前面的单一接口。做一次循环。具体调用我们看下代码即可以明白，后续逻辑我们不做多的说明(同时这2个接口也可以作为接口开发的范例，业务使用的特性比较全)：

单独的显示接口文档Action接口：

```
/// <summary>
/// 接口在线文档生成器接口
/// </summary>
[ActionName("Api.Doc"), DisablePackageSdk, EnableRecordApiLog(false), Version(0, 0)]
public class ApiDocAction : ActionBase<ApiDocAction.ApiDocActionRequestDto, ApiDocAction.ApiDocActionResponseDto>
{
    /// <summary>
    /// 上送的参数对象
    /// </summary>
    public class ApiDocActionRequestDto : RequestDtoBase
    {
        /// <summary>
        /// 接口名称
        /// </summary>
        [Required]
        public string ActionName { get; set; }

        /// <summary>
        /// 接口版本（不传入的话，会显示版本号最大的同名接口）
        /// </summary>
        public string Version { get; set; }
    }

    /// <summary>
    /// 下送的数据
    /// </summary>
    public class ApiDocActionResponseDto
    {
        /// <summary>
        /// 接口描述对象
        /// </summary>
        public ActionDescriptor ActionDescriptor { get; set; }

        /// <summary>
        /// 请求参数 data 数据 JSON 串
        /// </summary>
        public string RequestDtoJson { get; set; }

        /// <summary>
        ///
        /// </summary>
        public IDictionary<Type, IEnumerable<ComplexObjPropertyTypeDescriptor>> RequestTypes { get; set; }

        /// <summary>
        ///
        /// </summary>
        public IDictionary<Type, IEnumerable<ComplexObjPropertyTypeDescriptor>> ResponseTypes { get; set; }

        /// <summary>
        /// 输出 JSON 串
        /// </summary>
        public string ResponseDtoJson { get; set; }

        /// <summary>
        /// 输出 XML 串
        /// </summary>
        public string ResponseDtoXml { get; set; }

    }

    /// <summary>
    ///
    /// </summary>
    private readonly IActionSelector _actionSelector;
    private readonly IApiDocBuilder _apiDocBuilder;
    private readonly IMediaTypeFormatterFactory _mediaTypeFormatterFactory;

    /// <summary>
    ///
```

```
        /// </summary>
        /// <param name="actionSelector">接口查找器</param>
        /// <param name="apiDocBuilder">接口仿真数据提供器</param>
        /// <param name="mediaTypeFormatterFactory">格式化器工厂</param>
        public ApiDocAction(
            IActionSelector actionSelector,
            IApiDocBuilder apiDocBuilder,
            IMediaTypeFormatterFactory mediaTypeFormatterFactory)
        {
            this._actionSelector = actionSelector;
            this._apiDocBuilder = apiDocBuilder;
            this._mediaTypeFormatterFactory = mediaTypeFormatterFactory;
        }

        /// <summary>
        /// 执行业务逻辑
        /// </summary>
        /// <returns></returns>
        public override ActionResult<ApiDocActionResponseDto> Execute()
        {
            //接口描述对象
            var actionDescriptor = this._actionSelector.GetActionDescriptor(this.RequestDto.ActionName, this.RequestDto.Version);

            //上送的数据
            string requestDtoJson = this._apiDocBuilder.CreateInstance(actionDescriptor.RequestDtoType).SerializeObjectToJosn();

            //下送的数据
            var actionResult = new ActionResult()
            {
                Data = this._apiDocBuilder.CreateInstance(actionDescriptor.ResponseDtoType),
                Flag = ActionResultFlag.SUCCESS,
                Info = "OK"
            };

            //代码生成器基类
            CodeGeneratorBase codeGeneratorBase = (CodeGeneratorBase)this._apiDocBuilder;

            //上送的对象信息
            var requestTypes = new List<Type> { actionDescriptor.RequestDtoType };
            codeGeneratorBase.GetComplexObjTypes(actionDescriptor.RequestDtoType, requestTypes);
            var requestTypeDescriptors = new Dictionary<Type, IEnumerable<ComplexObjPropertyTypeDescriptor>>();
            requestTypes.ForEach(o =>
            {
                requestTypeDescriptors.Add(o, codeGeneratorBase.GetComplexObjPropertyTypeDescriptors(o));
            });

            //下送对象信息
            var responseTypes = new List<Type>();
            if (codeGeneratorBase.IsComplexType(actionDescriptor.ResponseDtoType)
&& !codeGeneratorBase.IsCollection(actionDescriptor.ResponseDtoType))
            {
                responseTypes.Add(actionDescriptor.ResponseDtoType);
            }
            codeGeneratorBase.GetComplexObjTypes(actionDescriptor.ResponseDtoType, responseTypes);
            var responseTypeDescriptors = new Dictionary<Type, IEnumerable<ComplexObjPropertyTypeDescriptor>>();
            responseTypes.ForEach(o =>
            {
                responseTypeDescriptors.Add(o, codeGeneratorBase.GetComplexObjPropertyTypeDescriptors(o));
            });

            //返回
            return this.SuccessActionResult(new ApiDocActionResponseDto()
            {
                ActionDescriptor = actionDescriptor,
                RequestDtoJson = requestDtoJson,
                RequestTypes = requestTypeDescriptors,
                ResponseTypes = responseTypeDescriptors,
                ResponseDtoJson =
this._mediaTypeFormatterFactory.Create(ResponseFormat.JSON).SerializedActionResultToString(this.RequestContext, actionResult),
                ResponseDtoXml =
this._mediaTypeFormatterFactory.Create(ResponseFormat.XML).SerializedActionResultToString(this.RequestContext, actionResult)
            });
        }
    }
```

批量生成接口文档的Action接口：

```
    /// <summary>
    /// 生成 API 文档接口，此项目同时也示例，接口可以直接进行调用；返回值为 HTML 接口文档下载地址
    /// </summary>
    [ActionName("Api.Doc.Builder"), DisablePackageSdk, EnableRecordApiLog(false)]
    public class ApiDocBuilderAction : ActionBase<NullRequestDto, string>
    {
        /// <summary>
        ///
        /// </summary>
        private readonly IActionSelector _actionSelector;
        private readonly IApiDocBuilder _simulation;
        private readonly IMediaTypeFormatterFactory _mediaTypeFormatterFactory;

        /// <summary>
```

```
        ///
        /// </summary>
        /// <param name="actionSelector">接口搜索器</param>
        /// <param name="simulation">接口仿真数据输出器</param>
        /// <param name="mediaTypeFormatterFactory">格式化器工厂</param>
        public ApiDocBuilderAction(
            IActionSelector actionSelector,
            IApiDocBuilder simulation,
            IMediaTypeFormatterFactory mediaTypeFormatterFactory)
        {
            this._actionSelector = actionSelector;
            this._simulation = simulation;
            this._mediaTypeFormatterFactory = mediaTypeFormatterFactory;
        }

        /// <summary>
        /// 执行业务逻辑
        /// </summary>
        /// <returns></returns>
        public override ActionResult<string> Execute()
        {
            //获取所有的接口信息
            var actionDescriptors = this._actionSelector.GetActionDescriptors().Where(o => o.CanPackageToSdk
&& !o.IsObsolete).ToList();

            //调用输出器接口
            const string actionName = "API.Doc";
            //zip 保存地址
            const string zipSaveFile = "~/App_Data/apidoc/apidoc.zip";
            //保存文件夹
            const string htmlSaveDirectory = "~/App_Data/apidoc/temp";

            //保存物理路径
            var physicalHtmlSaveDirectory = this.RequestContext.HttpContext.Server.MapPath(htmlSaveDirectory);
            var physicalZipSaveFile = this.RequestContext.HttpContext.Server.MapPath(zipSaveFile);

            //删除临时文件夹
            if (Directory.Exists(physicalHtmlSaveDirectory))
            {
                Directory.Delete(physicalHtmlSaveDirectory, true);
            }
            if (File.Exists(physicalZipSaveFile))
            {
                File.Delete(physicalZipSaveFile);
            }

            //生成临时文件夹
            Directory.CreateDirectory(physicalHtmlSaveDirectory);

            //循环所有接口
            foreach (var actionDescriptor in actionDescriptors)
            {
                //上送业务参数对象
                var requestDto = new ApiDocAction.ApiDocActionRequestDto()
                {
                    ActionName = actionDescriptor.ActionName,
                    Version = actionDescriptor.Version
                };

                //原始请求参数
                var requestParams = new RequestParams()
                {
                    ActionName = actionName,
                    Data = requestDto.SerializeObjectToJosn(),
                    Format = "View",
                    AppKey = "",
                    Sign = "",
                    TimeStamp = DateTime.Now.ToString(CultureInfo.InvariantCulture),
                    Version = ""
                };

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

                //获取生成接口
                IAction apiDocAction = new ApiDocAction(this._actionSelector, this._simulation, this._mediaTypeFormatterFactory);
                apiDocAction.RequestDto = requestDto;
                apiDocAction.RequestContext = requestContext;
                apiDocAction.ActionDescriptor = (ActionDescriptor)requestContext.ActionDescriptor;

                try
                {
                    //执行文档生成器接口（当成服务使用）
                    var actionResult = apiDocAction.Execute();
```

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

                //格式化后的数据
                var serializedActionResultToString = mediaTypeFormatter.SerializedActionResultToString(requestContext, new
ActionResult()
                {
                    Data = actionResult.Data,
                    Flag = ActionResultFlag.SUCCESS,
                    Info = "OK"
                });

                //替换掉内部的 JS 引用
                serializedActionResultToString = serializedActionResultToString
                    .Replace("/GetResource?resourceName=jquery-1.9.1.min.js",
                        "http://apps.bdimg.com/libs/jquery/1.6.4/jquery.min.js");

                //输出到文件
                //HostingEnvironment.MapPath
                using (var streamWriter = new StreamWriter(this.RequestContext.HttpContext.Server.MapPath("{0}/{1}.{2}.html"
                            .With(htmlSaveDirectory, actionDescriptor.ActionName, actionDescriptor.Version))))
                {
                    streamWriter.WriteLine(serializedActionResultToString);
                }
            }
            catch (Exception ex)
            {
                this.Logger.Error("接口名称：{0}，错误:{1}".With(apiDocAction.RequestDto.SerializeObjectToJosn(),
ex.StackTrace));
            }
        }

        //打包文档
        using (var zipOutputStream = new ZipOutputStream(File.OpenWrite(physicalZipSaveFile)))
        {
            byte[] buffer = new byte[4096];
            zipOutputStream.SetLevel(9);
            foreach (var file in Directory.GetFiles(physicalHtmlSaveDirectory))
            {
                var entry = new ZipEntry(Path.GetFileName(file));
                zipOutputStream.PutNextEntry(entry);
                using (var fileStream = File.OpenRead(file))
                {
                    StreamUtils.Copy(fileStream, zipOutputStream, buffer);
                }
            }
        }

        //输出到客户端
        try
        {
            this.RequestContext.HttpContext.Response.ContentType = "application/zip";
            this.RequestContext.HttpContext.Response.AddHeader("Content-Disposition",
                "attachment; filename=apidoc_{0}.zip".With(DateTime.Now.ToString("yyMMddHHmmss")));
            this.RequestContext.HttpContext.Response.WriteFile(physicalZipSaveFile);
            this.RequestContext.HttpContext.Response.End();
        }
        catch (Exception)
        {
            // ignored
        }

        //返回
        return new ActionResult<string>()
        {
            Flag = ActionResultFlag.SUCCESS,
            Info = "批量生成 API 接口文档成功",
            Data = zipSaveFile
        };

    }
}
```

5.14 MVC里Controller构造函数服务注入

我们拿系统框架里提供统一资源管理器的Controller来说，此控制器属于系统级别，所有插件开发界面层都可以调用，我们先来看下代码：

```
/// <summary>
/// 获取资源类（方便插件资源文件获取），插件 view 视图里获取资源文件可以直接使用此控制器
/// 比如，想获取插件本身自己内嵌的资源 JS 文件，直接使用下面方式既可以
/// <![CDATA[
/// <script type="text/javascript" src="/GetResource?resourceName=jquery-1.9.1.min.js"></script>
/// <script type="text/javascript" src="/GetResource/jquery-1.9.1.min.js"></script>
/// ]]>
/// </summary>
public class ResourceController : ApiControllerBase
{
    /// <summary>
```

```
/// 程序集内嵌资源查找器
/// </summary>
private readonly IResourceFinderManager _resourceFinderManager;

/// <summary>
/// 构造函数
/// </summary>
/// <param name="resourceFinderManager">资源查找管理器</param>
public ResourceController(IResourceFinderManager resourceFinderManager)
{
    resourceFinderManager.CheckNullThrowArgumentNullException("resourceFinders");
    this._resourceFinderManager = resourceFinderManager;
}

/// <summary>
/// 获取资源  /GetResource?resourceName=jquery-1.9.1.min.js 或者 /GetResource/jquery-1.9.1.min.js
/// </summary>
/// <param name="resourceName">资源名称，请注意此地方仅仅是判断资源名称和资源查找器里集合尾部相同的资源；</param>
/// <returns>返回指定内嵌资源文件文本</returns>
public System.Web.Mvc.ActionResult GetResource(string resourceName)
{
    //获取资源
    var resource = this._resourceFinderManager.GetResource(resourceName);

    //存在资源就显示资源文本到客户端
    if (resource.IsNull()) return this.Content(string.Empty);

    //js
    if (resourceName.EndsWith(".js", StringComparison.OrdinalIgnoreCase))
    {
        return this.Content(resource, "application/javascript");
    }

    //css
    if (resourceName.EndsWith(".css", StringComparison.OrdinalIgnoreCase))
    {
        return this.Content(resource, "text/css");
    }

    //html
    if (resourceName.EndsWith(".html", StringComparison.OrdinalIgnoreCase))
    {
        return this.Content(resource, "text/html");
    }

    //gif
    if (resourceName.EndsWith(".gif", StringComparison.OrdinalIgnoreCase))
    {
        return this.File(Convert.FromBase64String(resource), "image/gif");
    }

    //jpg
    if (resourceName.EndsWith(".jpg", StringComparison.OrdinalIgnoreCase))
    {
        return this.File(Convert.FromBase64String(resource), "image/jpeg");
    }

    //png
    if (resourceName.EndsWith(".png", StringComparison.OrdinalIgnoreCase))
    {
        return this.File(Convert.FromBase64String(resource), "image/png");
    }

    //否则返回文本类型
        return this.Content(resource, "text/plain");
    }
}
```

我们从代码可以看出，控制器的构造函数里带了一个资源管理器 IResourceFinderManager 接口，我们不用管此接口的具体实现是什么，我们只要定义此参数，系统框架会自动将我们实现的注册到IOC的具体实现注入到控制器里。另外我们为此控制器定义了专门的路由：

```
/// <summary>
/// 路由注册，系统框架会自动注册此路由
/// </summary>
internal class RouteProvider : IRouteProvider
{
    /// <summary>
    /// 注册接口路由设置
    /// </summary>
    /// <param name="routes"></param>
    public void RegisterRoutes(RouteCollection routes)
    {
        //所有资源获取类  /Resource?resourceName=jquery-1.9.1.min.js
        routes.MapRoute(
            name: "ServiceCenter.Api.Core.V20_GetResource",
            url: "GetResource",
            defaults: new { controller = "Resource", action = "GetResource" },
            namespaces: new string[] { "Frxs.ServiceCenter.Api.Core.Host" });
```

```
            //获取资源的另外一种方式：/Resource/jquery-1.9.1.min.js
            routes.MapRoute(
                name: "ServiceCenter.Api.Core.V20_GetResource_01",
                url: "Resource/{*resourceName}",
                defaults: new { controller = "Resource", action = "GetResource" },
                    namespaces: new string[] { "Frxs.ServiceCenter.Api.Core.Host" });
    }
```

上面的路由接口也是系统框架提供的，系统启动的时候，会自动搜索所有实现此接口的类，然后进行批量进行路由注册，VIEW界面插件调用的话可以如下：

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>接口框架系统，接口框架版本：2.2.60531.28953</title>
    <script type="text/javascript" src="/GetResource?resourceName=jquery-1.9.1.min.js"></script>
    <script type="text/javascript" src="/GetResource?resourceName=jquery.poshytip.js"></script>
    <link rel="stylesheet" href="/GetResource?resourceName=tip-yellow.css" type="text/css" />
</head>
```

5.15 API接口框架(VIEW)视图引擎介绍

接口框架里返回数据格式化系统定义了3中格式化方式：JSON/XML/VIEW；前2种格式化返，我们前面已经讲述到，现在我们讲述下VIEW格式的返回。VIEW数据返回的简单流程为：首先我们实现了一个VIEW格式化器，和JSON和XML一样，VIEW格式化器同样实现了IMediaTypeFormatter 接口，我们来看一下具体实现：

```
/// <summary>
/// IAction 接口文档生成器；如果是第三方插件内嵌资源视图文件，
/// </summary>
public class ViewMediaTypeFormatter : IMediaTypeFormatter
{
    /// <summary>
    /// 程序集内嵌资源实体查找器
    /// </summary>
    private readonly IResourceFinderManager _viewFinderManager;

    /// <summary>
    /// 内嵌资源路径规则
    /// </summary>
    private const string ManifestResourceViewSearchPath = "{0}.Views.{1}.aspx";

    /// <summary>
    /// 接口框架命名空间
    /// </summary>
    private static readonly string Namespace = Assembly.GetExecutingAssembly().GetName().Name;
    /// <summary>
    /// 换行符
    /// </summary>
    private static readonly string NewLine = Environment.NewLine;

    /// <summary>
    /// 视图搜索路径地址：如：~/views/{0}.aspx
    /// </summary>
    private string[] ViewLocationFormats { get; set; }

    /// <summary>
    /// 初始化下默认的搜索地址
    /// </summary>
    /// <param name="viewFinderManager">资源查找器</param>
    public ViewMediaTypeFormatter(IResourceFinderManager viewFinderManager)
    {
        viewFinderManager.CheckNullThrowArgumentNullException("viewFinderManager");
        //按照优先级排序下
        this._viewFinderManager = viewFinderManager;
        //请注意先后顺序
        this.ViewLocationFormats = new string[] { "~/Views/{0}.aspx", "~/Views/T.aspx" };
    }

    /// <summary>
    /// 返回待搜索列表
    /// </summary>
    /// <param name="requestContext">当前请求上下文</param>
    /// <returns>返回当前接口需要搜索的资源</returns>
    private IEnumerable<string> GetSearchedViewPath(RequestContext requestContext)
    {
        //用于保存搜索路径

        //接口名称
        string actionName = requestContext.RawRequestParams.ActionName;

        //合法的视图名称集合
        var viewNames = new string[] { actionName, actionName.Replace(".", "") };

        //循环下看下是否存在自定义的视图文件路径(优先级最高1)
        IList<string> searchedViewPath = (from locationPath in this.ViewLocationFormats
```

```
            from viewName in viewNames
            select requestContext.HttpContext.Server.MapPath(locationPath.With(viewName))).ToList();

        //当前插件所在程序集 Views 文件夹查找（程序集自定义的内嵌资源文件优先级 2）
        if (!requestContext.ActionDescriptor.IsNull())
        {
            //当前接口所在程序集
            Assembly assembly = requestContext.ActionDescriptor.ActionType.Assembly;
            //检测所有视图命名规则是否存在
            foreach (var viewName in viewNames)
            {
                searchedViewPath.Add(ManifestResourceViewSearchPath.With(assembly.GetName().Name, viewName));
            }
        }

        //全局默认（兜底）
        searchedViewPath.Add(ManifestResourceViewSearchPath.With(Namespace, "T"));

        //返回所有的待搜索路径
        return searchedViewPath;
    }

    /// <summary>
    /// 根据上下文获取当前接口视图源代码
    /// </summary>
    /// <param name="requestContext">当前请求上下文</param>
    /// <returns>返回接口视图源文件；注意：在未找到对应实体源文件，会抛出异常</returns>
    private string GetViewSource(RequestContext requestContext)
    {
        //资源模板文件
        string viewSource = null;

        //获取待搜索列表
        var searchedViewPaths = this.GetSearchedViewPath(requestContext);

        //循环搜索列表
        foreach (var searchPath in searchedViewPaths)
        {
            //找到对应的模板源码，直接返回
            viewSource = this._viewFinderManager.GetResource(searchPath);
            if (!viewSource.IsNull())
            {
                break;
            }
            if (!viewSource.IsNull())
            {
                break;
            }
        }

        //都未找到，抛出异常
        if (viewSource.IsNull())
        {
            throw new ApiException(Resource.CoreResource.View_Not_Exists.With(NewLine, string.Join(NewLine,
searchedViewPaths.Distinct().ToArray())));
        }

        //返回源代码
        return viewSource;
    }

    /// <summary>
    /// 搜索视图，读取视图，然后执行视图，返回执行后的视图内容
    /// </summary>
    /// <param name="requestContext">当前请求上下文</param>
    /// <param name="actionResult">IAction 执行结果</param>
    /// <returns>返回格式化后的字符串</returns>
    public string SerializedActionResultToString(RequestContext requestContext, ActionResult actionResult)
    {
        requestContext.CheckNullThrowArgumentNullException("requestContext");
        requestContext.RawRequestParams.CheckNullThrowArgumentNullException("requestContext.RawRequestParams");

        //资源模板文件
        string viewSource = this.GetViewSource(requestContext);
        //找到视图文件，就使用视图输出
        ApiViewEngine viewEngine = new ApiViewEngine(Language.CSharp);
        var viewParameters = new ViewParameterCollection();
        //视图模板文件始终只会包含 RequestContext，ActionResult 这 2 个对象
        viewParameters.Add(new ViewParameter("RequestContext", requestContext));
        //在需要输出复杂类型的时候，所有的数据请挂靠在 ActionResult.Data 属性下，这样外部可以根据此对象来进行视图解析
        viewParameters.Add(new ViewParameter("ActionResult", actionResult));
        //编译视图文件并输出编译执行后的结果
        return viewEngine.CompileByViewSource(viewSource, viewParameters);
    }
}
```

从上面点可以看出，VIEW格式化器传入了资源查找器管理器，利用资源查找器根据约定的查找路径，我们先获取到视图源文件({接口名称}.aspx)，然后利用视图引擎：来动态编译视图成一个cs类文件，然后利用 CodeDomProvider 类对生成的cs文件进行动态编译，执行里面的方法，返回执行后的数据流(Stream)，然后再对数据流转化成字符串。最后利用IResponse接口对

返回的字符串输出到客户端。因此重点是我们怎么来解析视图文件。

说明：系统框架视图文件为什么选择.aspx结尾作为视图模板文件，我们其实可以完全不要考虑视图文件后缀，使用.aspx作为视图文件，主要是想利用VS的自动提示，方便我们视图文件的编写。

我们先来看一个系统框架提供的通用模板T.aspx（接口在找不到自己约定的视图的时候，会使用此视图文件来兜底，防止错误）

```
<!--
 ******************************************************
 * FRXS(ISC) zhangliang4629@163.com <%:DateTime.Now.ToString("yyyy\\/MM\\/dd HH:mm:ss.fff") %>
 * ******************************************************
-->

<%@ Page Language="C#" %>

<%@ Assembly Name="Frxs.ServiceCenter.Api.Core" %>
<%@ Import Namespace="Frxs.ServiceCenter.Api.Core" %>
<%@ Import Namespace="System.Collections.Specialized" %>
<%@ Import Namespace="System.Collections" %>
<%@ Import Namespace="System.Web" %>

<script runat="server">
    public Frxs.ServiceCenter.Api.Core.RequestContext RequestContext;
    public Frxs.ServiceCenter.Api.Core.ActionResult ActionResult;
</script>

<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title><%:this.RequestContext.DecryptedRequestParams.ActionName %></title>
</head>
<body style="padding: 10px; font-family: 微软雅黑;">

    <b>ActionName</b>
    <ul>
        <li><%:this.RequestContext.DecryptedRequestParams.ActionName %></li>
    </ul>

    <%if (null != this.RequestContext.ActionDescriptor)
      { %>
    <b>ActionDescriptor</b>
    <ul>
        <%foreach (var item in this.RequestContext.ActionDescriptor.GetAttributes())
          { %>
        <li><%:item.Key %>: <%:item.Value.SerializeObjectToJosn() %></li>
        <%} %>
    </ul>
    <%} %>

    <b>AdditionDatas:</b>
    <br />
    <ul>
        <%foreach (var item in this.RequestContext.AdditionDatas)
          { %>
        <li><%:item.Key %>: <%: item.Value is DateTime? ((DateTime)item.Value).ToString("yyyy/MM/dd HH:mm:ss.ffffff"):
item.Value %></li>
        <%} %>
    </ul>

    <b>RequestCacheKey:</b>
    <br />
    <ul>
        <li><%:this.RequestContext.GetRequestCacheKey() %></li>
    </ul>
    <b>SysOptions:</b>
    <br />
    <ul>
        <%foreach (var item in this.RequestContext.SysOptions.GetAttributes())
          { %>
        <%if (item.Value.GetType() == typeof(string[]))
          { %>
        <li><%:item.Key%>: <%:string.Join("<br />", (string[])item.Value)%></li>
        <%}
          else if (item.Value.GetType() == typeof(Dictionary<string, object>))
          { %>
        <li><%:item.Key%>:
         <%foreach (var kv in (Dictionary<string, object>)item.Value)
           { %>
            <br />
            <%:kv.Key%>: <%:kv.Value %></li>
        <%} %>
        <%}
          else
          { %>
        <li><%:item.Key%>: <%:item.Value%></li>
        <%}
          } %>
    </ul>
```

```
<b>Headers:</b>
<br />
<ul>
    <%foreach (string key in this.RequestContext.HttpContext.Response.Headers.AllKeys)
      { %>
    <li><%:key %>: <%:this.RequestContext.HttpContext.Response.Headers[key] %></li>
    <%} %>
</ul>

<b>ActionConfigCollection:</b>
<br />
<ul>
    <%foreach (var item in ApiConfigManager.Configs.GetConfigs())
      { %>
    <li><b><%:item.Key.IsNullOrEmptyForDefault(()=>"全局",s=>s) %>: </b>
        <% foreach (var p in item.Value.GetAttributes(false))
           {
        %>
        <br />
          <%:p.Key %>: <%: (p.Value is IEnumerable) ? p.Value.SerializeObjectToJosn() : p.Value %>
        <% } %>
    </li>
    <%} %>
</ul>

<b>RequestParams:</b>
<br />
<ul>
    <%foreach (var kv in this.RequestContext.RawRequestParams.GetAttributes())
      { %>
    <li><%:kv.Key %>: <%:kv.Value %></li>
    <%} %>
</ul>

<b>ActionResult:</b>
<br />
<div style="padding: 10px;">
    <%:this.ActionResult.SerializeObjectToJosn() %>
</div>
</body>
</html>
```

视图引擎的关键是，如何把上面的视图文件转换成C#可以识别的类，即第一步如何解析此视图文件，将此视图文件转换成一个C#类。由于代码比较长，我们先来简单的讲解下执行流程：通过此模板文件可以看出，我们把模板拆分成5个部分

1. <%@ Assembly Name="Frxs.ServiceCenter.Api.Core" %> 这一部分是系统编译此模板需要导入的 dll 文件
2. <%@ Import Namespace="Frxs.ServiceCenter.Api.Core" %> 这部分是系统编译此模板需要导入的命名空间
3. <script runat="server">
   public Frxs.ServiceCenter.Api.Core.RequestContext RequestContext;
   public Frxs.ServiceCenter.Api.Core.ActionResult ActionResult;
   </script>
   这一部分是会生成 C#类的属性

4. <% %> 这部分会生成类的执行代码
5. 除上面之外的所有文本会转换成文本常量，在执行类的时候，会字符赋值

经过上面的流程转换后会生成一个完整的 C#类，具体生成的类文件为：

```
using System;
using System.Text;
using System.Collections.Generic;
using Frxs.ServiceCenter.Api.Core;
using System.Collections.Specialized;
using System.Collections;
using System.Web;

public class Page
{
    public System.IO.StreamWriter Response;
    public string SectionText7;
    public string SectionText9;
    public string SectionText11;
    public string SectionText13;
    public string SectionText15;
    public string SectionText17;
    public string SectionText19;
    public string SectionText21;
    public string SectionText23;
    public string SectionText25;
    public string SectionText27;
    public string SectionText29;
    public string SectionText31;
    public string SectionText33;
    public string SectionText35;
    public string SectionText37;
    public string SectionText39;
    public string SectionText41;
    public string SectionText43;
    public string SectionText45;
    public string SectionText47;
    public string SectionText49;
```

```
        public string SectionText51;
        public string SectionText53;
        public string SectionText55;
        public string SectionText57;
        public string SectionText59;
        public string SectionText61;
        public string SectionText63;
        public string SectionText65;
        public string SectionText67;
        public string SectionText69;
        public string SectionText71;
        public string SectionText73;
        public string SectionText75;
        public string SectionText77;
        public string SectionText79;
        public string SectionText81;
        public string SectionText83;
        public string SectionText85;
        public string SectionText87;
        public string SectionText89;
        public string SectionText91;
        public string SectionText93;
        public string SectionText95;
        public Frxs.ServiceCenter.Api.Core.RequestContext RequestContext;
        public Frxs.ServiceCenter.Api.Core.ActionResult ActionResult;
        public Page() { }

        public void RenderPage()
        {
            Response.Write(SectionText7);
            Response.Write(DateTime.Now.ToString("yyyy\\/MM\\/dd HH:mm:ss.fff"));
            Response.Write(SectionText9);
            Response.Write(this.RequestContext.DecryptedRequestParams.ActionName);
            Response.Write(SectionText11);
            Response.Write(this.RequestContext.DecryptedRequestParams.ActionName);
            Response.Write(SectionText13);
            if (null != this.RequestContext.ActionDescriptor)
            {
                Response.Write(SectionText15);
                foreach (var item in this.RequestContext.ActionDescriptor.GetAttributes())
                {
                    Response.Write(SectionText17);
                    Response.Write(item.Key);
                    Response.Write(SectionText19);
                    Response.Write(item.Value.SerializeObjectToJosn());
                    Response.Write(SectionText21);
                }
                Response.Write(SectionText23);
            }
            Response.Write(SectionText25);
            foreach (var item in this.RequestContext.AdditionDatas)
            {
                Response.Write(SectionText27);
                Response.Write(item.Key);
                Response.Write(SectionText29);
                Response.Write(item.Value is DateTime ? ((DateTime)item.Value).ToString("yyyy/MM/dd HH:mm:ss.ffffff") :
item.Value);
                Response.Write(SectionText31);
            }
            Response.Write(SectionText33);
            Response.Write(this.RequestContext.GetRequestCacheKey());
            Response.Write(SectionText35);
            foreach (var item in this.RequestContext.SysOptions.GetAttributes())
            {
                Response.Write(SectionText37);
                if (item.Value.GetType() == typeof(string[]))
                {
                    Response.Write(SectionText39);
                    Response.Write(item.Key);
                    Response.Write(SectionText41);
                    Response.Write(string.Join("<br />", (string[])item.Value));
                    Response.Write(SectionText43);
                }
                else if (item.Value.GetType() == typeof(Dictionary<string, object>))
                {
                    Response.Write(SectionText45);
                    Response.Write(item.Key);
                    Response.Write(SectionText47);
                    foreach (var kv in (Dictionary<string, object>)item.Value)
                    {
                        Response.Write(SectionText49);
                        Response.Write(kv.Key);
                        Response.Write(SectionText51);
                        Response.Write(kv.Value);
                        Response.Write(SectionText53);
                    }
                    Response.Write(SectionText55);
                }
                else
                {
                    Response.Write(SectionText57);
                    Response.Write(item.Key);
                    Response.Write(SectionText59);
```

```
            Response.Write(item.Value);
            Response.Write(SectionText61);
        }
    }
    Response.Write(SectionText63);
    foreach (string key in this.RequestContext.HttpContext.Response.Headers.AllKeys)
    {
        Response.Write(SectionText65);
        Response.Write(key);
        Response.Write(SectionText67);
        Response.Write(this.RequestContext.HttpContext.Response.Headers[key]);
        Response.Write(SectionText69);
    }
    Response.Write(SectionText71);
    foreach (var item in ApiConfigManager.Configs.GetConfigs())
    {
        Response.Write(SectionText73);
        Response.Write(item.Key.IsNullOrEmptyForDefault(() => "全局", s => s));
        Response.Write(SectionText75);
        foreach (var p in item.Value.GetAttributes(false))
        {
            Response.Write(SectionText77);
            Response.Write(p.Key);
            Response.Write(SectionText79);
            Response.Write((p.Value is IEnumerable) ? p.Value.SerializeObjectToJosn() : p.Value);
            Response.Write(SectionText81);
        }
        Response.Write(SectionText83);
    }
    Response.Write(SectionText85);
    foreach (var kv in this.RequestContext.RawRequestParams.GetAttributes())
    {
        Response.Write(SectionText87);
        Response.Write(kv.Key);
        Response.Write(SectionText89);
        Response.Write(kv.Value);
        Response.Write(SectionText91);
    }
    Response.Write(SectionText93);
    Response.Write(this.ActionResult.SerializeObjectToJosn());
    Response.Write(SectionText95);
    }
}
```

经过引擎解析器解析后，得到上面的类，然后试图引擎会编译上面的类，得到动态程序集(Assembly)，然后通过得到的动态程序集创建出Page类，在执行上述类 public void RenderPage() 方法。

上面就是试图引擎对视图的解析与执行简单流程，具体的实现由于代码比较多我们可以看项目 0.1.Frxs.ServiceCenter.Api.Core.V20 里文件夹 ViewEngine 所有实现即可。我们这里仅仅给出现编译过程中的核心类 ApiViewEngine：

```
/// <summary>
/// 接口框架视图引擎；使用 webform 语法
/// </summary>
public class ApiViewEngine
{
    /// <summary>
    /// 编译源码的语言
    /// </summary>
    private readonly string _language = "C#";

    /// <summary>
    /// 编译源文件需要的系统框架 dll
    /// </summary>
    private readonly string[] _systemDlls = new string[] { "System.dll", "System.Core.dll", "System.Web.dll",
"System.Linq.dll" };

    /// <summary>
    /// 缓存视图文件便于后的程序集，提高执行性能
    /// </summary>
    private static readonly IDictionary<string, KeyValuePair<SectionCollection, Assembly>> CacheAssemblys = new
Dictionary<string, KeyValuePair<SectionCollection, Assembly>>();

    /// <summary>
    /// 默认使用 C#语言编译器
    /// </summary>
    public ApiViewEngine() { }

    /// <summary>
    /// 指定 VIEW 视图使用的语言，C#或者 VB
    /// </summary>
    /// <param name="language">编译器语言</param>
    public ApiViewEngine(Language language)
    {
        this._language = language.ToString();
    }

    /// <summary>
    /// 编译视图也需要引入的第三方 dll 程序集
    /// </summary>
```

```
        public string[] Assemblies { get; set; }

        /// <summary>
        /// 编译视图需要引入的命名空间
        /// </summary>
        public string[] Namespaces { get; set; }

        /// <summary>
        /// 一除掉空行
        /// </summary>
        /// <param name="streamReader"></param>
        /// <returns></returns>
        private string RemoveEmptyLine(TextReader streamReader)
        {
            var lines = new List<string>();
            string line = null;
            do
            {
                line = streamReader.ReadLine();
                if (!line.IsNullOrEmpty())
                {
                    lines.Add(line);
                }
            } while (!line.IsNull());

            return string.Join(Environment.NewLine, lines.ToArray());
        }

        /// <summary>
        /// 根据视图文件原始文件内容，编译视图文件到程序集
        /// </summary>
        /// <param name="viewSource">视图文件源码</param>
        /// <param name="assemblies">引用的程序集集合</param>
        /// <param name="namespaces">需要添加的命名空间集合</param>
        /// <param name="parameters">视图定义的需要输入的参数集合</param>
        /// <param name="response">将执行后的视图保存到数据流</param>
        /// <returns>the comiled assembly</returns>
        private void CompileByViewSource(string viewSource, string[] assemblies, string[] namespaces, ViewParameterCollection
parameters, StreamWriter response)
        {
            //源码为空，直接抛出异常
            viewSource.CheckNullThrowArgumentNullException("viewSource");

            //计算源文件的 MD5 值，具有相同 MD5 值的源文件，必须会有相同的编译结果；
            //仅仅不同的是，属性参数的不同
            var viewSourceMd5 = MD5.Encrypt(viewSource);

            //检测缓存系统里是否存在已经编译的视图模板文件
            if (CacheAssemblys.ContainsKey(viewSourceMd5))
            {
                var item = CacheAssemblys[viewSourceMd5];
                //直接执行已经编译的视图文件
                item.Key.Process(item.Value, parameters, response);
                return;
            }

            //引用的程序集
            this.Assemblies = assemblies;

            //引用的命名空间
            this.Namespaces = namespaces;

            //分析视图原文件
            var sections = ViewParser.ParsePage(viewSource, string.Empty);

            //获取代码编译器
            CodeDomProvider provider = CodeDomProvider.CreateProvider(this._language);
            CompilerParameters compilerparams = new CompilerParameters
            {
                GenerateInMemory = true,
                GenerateExecutable = false,
                IncludeDebugInformation = false
            };
            //system dll
            foreach (var sysDll in this._systemDlls)
                compilerparams.ReferencedAssemblies.Add(sysDll);
            //setup references assemblies
            if (!this.Assemblies.IsNull())
            {
                foreach (string assembly in this.Assemblies)
                {
                    compilerparams.ReferencedAssemblies.Add(assembly);
                }
            }

            //引入编译需要的第三方 DLL
            foreach (Section section in sections)
            {
                if (section.Type == SectionType.Directive)
                {
                    if (section.Values.Directive.Equals("assembly", StringComparison.OrdinalIgnoreCase))
```

```
                {
                    string assembly;
                    if (section.Values.TryGetValue("name", out assembly))
                    {
                        compilerparams.ReferencedAssemblies.Add(Path.Combine(HostHelper.GetBinDirectory(), assembly +
".dll"));
                    }
                }
            }
        }

        //获取源代码
        string sourceClassString = sections.ExtractSource(namespaces);

        //源代码为空，直接返回
        if (sourceClassString.Length == 0)
        {
            return;
        }

        //编译源代码
        var compilerResults = provider.CompileAssemblyFromSource(compilerparams, new string[] { sourceClassString });

        //编译错误，直接将错误信息输出
        if (compilerResults.Errors.HasErrors)
        {
            //先输出下原始的视图源文件
            response.WriteLine(sourceClassString);
            //将错误信息输出到源文件最后
            foreach (CompilerError error in compilerResults.Errors)
            {
                response.WriteLine("Error on line {0}: {1}", error.Line, error.ErrorText);
            }
        }
        //编译成功，执行编译结果
        else
        {
            //压入到缓存，方便下次快速访问，无需再次编译（多次编译在执行很多次的情况下，内存会暴增，因为前面编译的程序集，并没有释放）
            if (!CacheAssemblys.ContainsKey(viewSourceMd5))
                CacheAssemblys.Add(viewSourceMd5, new KeyValuePair<SectionCollection, Assembly>(sections,
compilerResults.CompiledAssembly));
            //执行一次编译
            sections.Process(compilerResults.CompiledAssembly, parameters, response);
        }
    }

    /// <summary>
    /// 根据提供的视图文件路径，编译视图文件到程序集
    /// </summary>
    /// <param name="viewPath">视图文件路径，请输入绝对路径比如：g:\\temp\t.aspx</param>
    /// <param name="assemblies">视图引用类型需要用到的程序集</param>
    /// <param name="namespaces">视图引用类型需要用到的命名空间（如果类全部是完整的输入，不需要命名空间，但是需要引用所属的程序集）
</param>
    /// <param name="parameters">视图对外公开的参数（即输入模型对象）</param>
    /// <param name="response">将视图执行结果输出到流</param>
    private void CompileByViewPath(string viewPath, string[] assemblies, string[] namespaces, ViewParameterCollection
parameters, StreamWriter response)
    {
        //路径不能为空
        if (viewPath.IsNullOrEmpty())
        {
            throw new ArgumentNullException("viewPath");
        }

        //指定的视图文件不存在
        if (!File.Exists(viewPath))
        {
            throw new Exception("视图文件：{0} 未找到".With(viewPath));
        }

        //读取视图文件
        var viewSource = File.ReadAllText(viewPath);

        //编译源文件，生成对应的程序集
        this.CompileByViewSource(viewSource, assemblies, namespaces, parameters, response);
    }

    /// <summary>
    /// 编译视图文件并执行视图
    /// </summary>
    /// <param name="viewPath">视图文件路径，请输入绝对路径比如：g:\\temp\t.aspx</param>
    /// <param name="parameters">视图定义的参数集合</param>
    /// <param name="encode">视图文件文件编码</param>
    /// <returns>返回视图执行结果字符串</returns>
    public string CompileByViewPath(string viewPath, ViewParameterCollection parameters, Encoding encode)
    {
        MemoryStream memoryStream = new MemoryStream();
        StreamWriter response = new StreamWriter(memoryStream, encode) { AutoFlush = true };
        this.CompileByViewPath(viewPath, null, null, parameters, response);
        memoryStream.Position = 0;
```

```
        StreamReader streamReader = new StreamReader(memoryStream, encode);
        var responseString = this.RemoveEmptyLine(streamReader);
        streamReader.Dispose();
        return responseString;
    }

    /// <summary>
    /// 编译视图文件并执行视图
    /// </summary>
    /// <param name="viewSource">视图文件源码</param>
    /// <param name="parameters">视图定义的参数集合</param>
    /// <param name="encode">视图文件文件编码</param>
    /// <returns>编译视图源代码，并将视图执行结果返回</returns>
    public string CompileByViewSource(string viewSource, ViewParameterCollection parameters, Encoding encode)
    {
        MemoryStream memoryStream = new MemoryStream();

        StreamWriter response = new StreamWriter(memoryStream, encode) { AutoFlush = true };
        this.CompileByViewSource(viewSource, null, null, parameters, response);
        memoryStream.Position = 0;
        StreamReader streamReader = new StreamReader(memoryStream, encode);
        var responseString = this.RemoveEmptyLine(streamReader);

        streamReader.Dispose();

        return responseString;
    }

    /// <summary>
    /// 编译视图文件并执行视图，默认使用 UTF-8 编译
    /// </summary>
    /// <param name="viewPath">视图文件路径，请输入绝对路径比如：g:\\temp\t.aspx</param>
    /// <param name="parameters">视图定义的参数集合</param>
    /// <returns>返回视图执行结果字符串</returns>
    public string CompileByViewPath(string viewPath, ViewParameterCollection parameters)
    {
        return this.CompileByViewPath(viewPath, parameters, Encoding.UTF8);
    }

    /// <summary>
    /// 编译视图文件并执行视图，默认使用 UTF-8 编译
    /// </summary>
    /// <param name="viewPath">视图文件路径，请输入绝对路径比如：g:\\temp\t.aspx</param>
    /// <param name="parameters">视图定义的参数集合</param>
    /// <returns></returns>
    public string CompileByViewPath(string viewPath, params ViewParameter[] parameters)
    {
        //构造参数集合
        ViewParameterCollection viewParameters = new ViewParameterCollection();

        //指定了参数
        if (!parameters.IsNull())
        {
            viewParameters.AddRange(parameters);
        }

        //便于并执行原文件
        return this.CompileByViewPath(viewPath, viewParameters);
    }

    /// <summary>
    /// 编译视图文件并执行视图，默认使用 UTF-8 编译
    /// </summary>
    /// <param name="viewSource">视图文件源码</param>
    /// <param name="parameters">视图定义的参数集合</param>
    /// <returns>编译视图源代码，并将视图执行结果返回</returns>
    public string CompileByViewSource(string viewSource, ViewParameterCollection parameters)
    {
        return this.CompileByViewSource(viewSource, parameters, Encoding.UTF8);
    }

    /// <summary>
    /// 编译视图文件并执行视图，默认使用 UTF-8 编译
    /// </summary>
    /// <param name="viewSource">视图文件源码</param>
    /// <param name="parameters">视图定义的参数集合</param>
    /// <returns></returns>
    public string CompileByViewSource(string viewSource, params ViewParameter[] parameters)
    {
        //构造参数集合
        ViewParameterCollection viewParameters = new ViewParameterCollection();

        //指定了参数
        if (!parameters.IsNull())
        {
            viewParameters.AddRange(parameters);
        }

        return this.CompileByViewSource(viewSource, viewParameters);
    }
}
```

5.16 作业任务支持(ITask,ITaskSchedulerRegistar)

系统框架支持作业任务扩展，与前面的所有实现一样，我们只要新建实现类，实现ITask接口即可以实现作业任务，然后注册好作业任务调度器就可以了。下面我们看一下ITask定义：

```
/// <summary>
/// 外部作业任务都需要实现此类
/// 注意外部的实现类里不能注入：
/// HttpContextBase,HttpRequestBase，
/// HttpResponseBase，
/// HttpServerUtilityBase，
/// HttpSessionStateBase：因为在新开的线程里无法使用 http 管道
/// 其他的 IOC 容器里的注入类都可以
/// </summary>
public interface ITask
{
    /// <summary>
    /// 执行作业任务
    /// </summary>
    /// <param name="taskExecuteContext">执行作业任务上下文</param>
        void Execute(TaskExecuteContext taskExecuteContext);
}
```

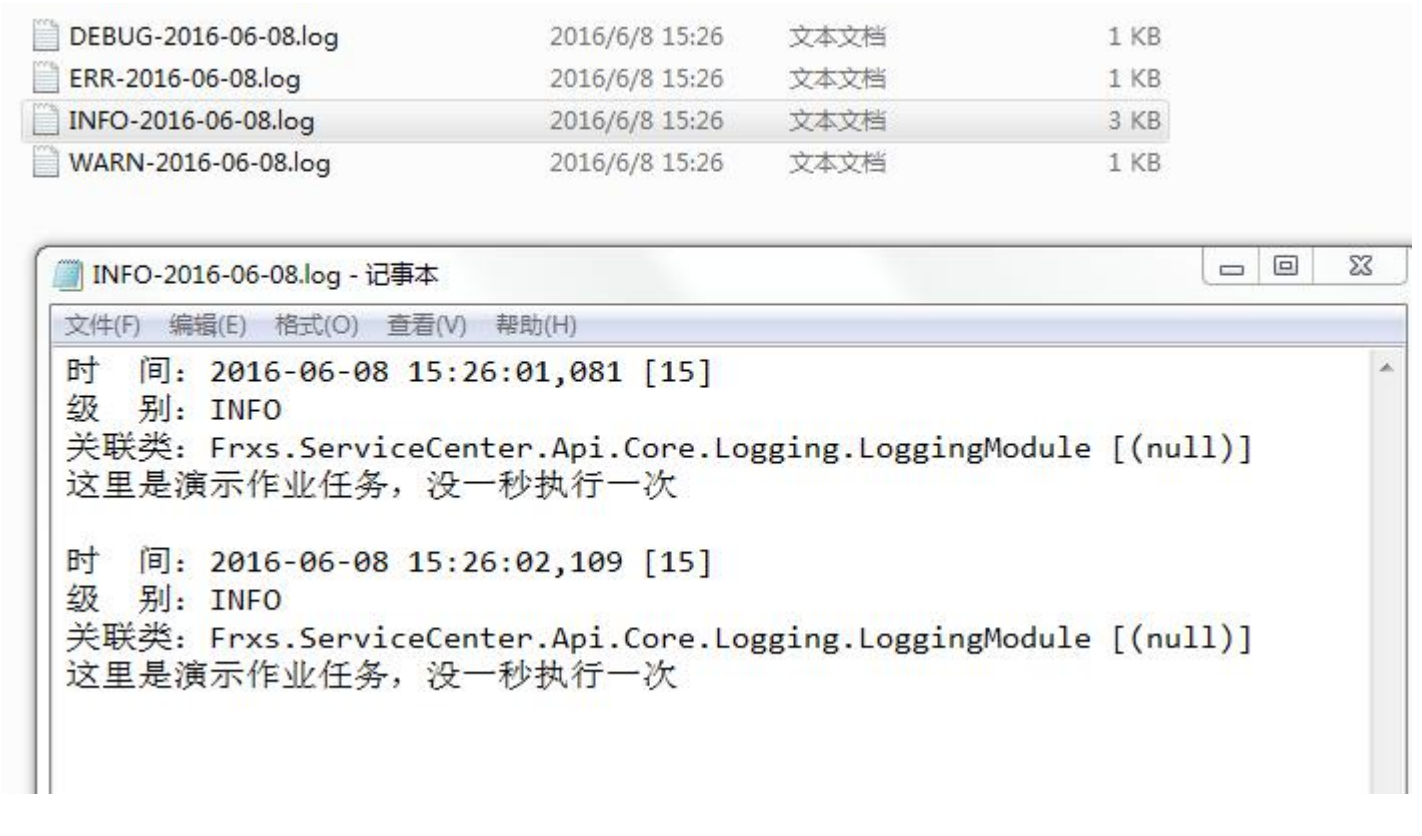
作业接口定义非常简单，只定义了一个Execute方法，我们就不多做阐述，我们现在来看怎么实现一个作业任务。我们新建一个名叫：TestTask.cs的类，然后实现ITask接口，代码如下：

```
/// <summary>
/// 演示作业任务
/// </summary>
public class TestTask : ITask
{
    /// <summary>
    ///
    /// </summary>
    private ILogger _logger;

    /// <summary>
    ///
    /// </summary>
    /// <param name="logger"></param>
    public TestTask(ILogger logger)
    {
         this._logger = logger ?? NullLogger.Instance;
    }

    /// <summary>
    ///
    /// </summary>
    public void Execute(TaskExecuteContext taskExecuteContext)
    {
        this._logger.Information("这里是演示作业任务，没一秒执行一次");
    }
}
```

然后新建一个实现了 ITaskSchedulerRegistar 接口类用于注册作业任务调度管理，代码如下：

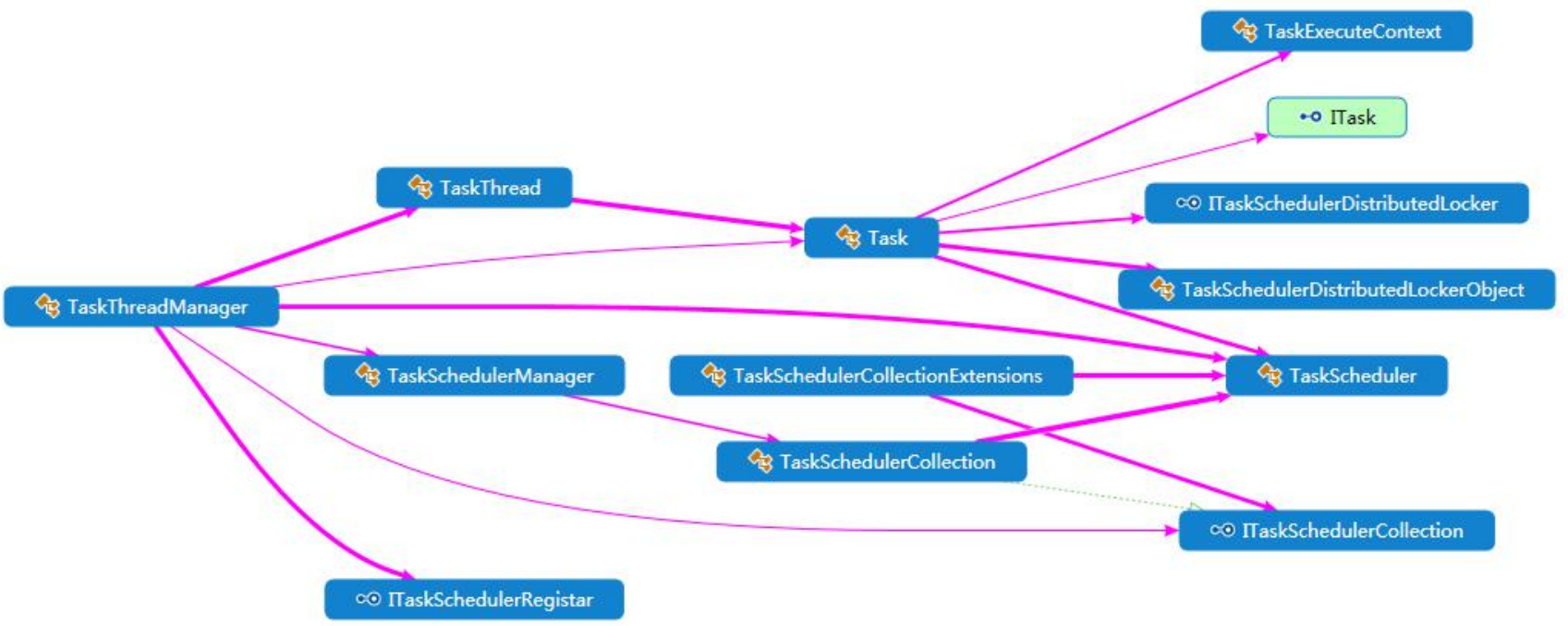
```
/// <summary>
/// 任务作业调度器注册
/// </summary>
public class TaskSchedulerRegistar : ITaskSchedulerRegistar
{
    /// <summary>
    /// 优先级
    /// </summary>
    public int Order
    {
        get {  return 0 ;}
    }

    /// <summary>
    ///
    /// </summary>
    /// <param name="taskSchedulerCollection"></param>
    public void Register(ITaskSchedulerCollection taskSchedulerCollection)
    {
        //注册作业任务的任务调度器（每秒执行一次）
        taskSchedulerCollection.Register<TestTask>(name: "SYS.KeyAliveTask"", seconds: 1); //当然 Name 参数可以是任意字符
        //更新任务调度器(将前面注册的调度器更新为 2 秒执行一次)
        taskSchedulerCollection.Update(name: "SYS.KeyAliveTask", seconds: 2);
    }
}
```

上面作业任务每个一秒钟向日志记录器里记录一个日志，我们保存上面类，然后运行HOST项目，在看我们的日志Info文件。



我们看到系统已经执行了我们定义的作业任务。

我们已经知道了怎么去创建一个作业任务，接下来我们详细讲解下作业任务系统的实现原理和核心代码，作业任务模块抽象出了几个概念：作业线程(TaskThread)，作业调度器(TaskScheduler)，调度任务管理器(TaskSchedulerManager)，作业线程管理器(TaskThreadManager)，作业任务封装对象(Task)，作业接口(ITask)。我们先看一张各个类之间的调用关系图：



现在我们来介绍下各个类之间的关系。前面我们已经知道当我们实现一个作业任务后，为了让其正常工作，我们需要将其注册到作业调度器里，我们先来看下作业任务调度器：

```
/// <summary>
/// 调度器管理器
/// </summary>
public class TaskSchedulerManager
{
    /// <summary>
    /// 保存作业任务调度器
    /// </summary>
    private static readonly TaskSchedulerCollection
        Instance = new TaskSchedulerCollection();

    /// <summary>
    /// 获取调度器列表
    /// </summary>
    public static ITaskSchedulerCollection TaskSchedulers
    {
        get
        {
            return Instance;
        }
    }
}
```

其实现很简单，使用了一个静态的 TaskSchedulerCollection 来保存注册信息，其实现了 ITaskSchedulerCollection

接口。我们再来看下 ITaskSchedulerCollection 接口定义：

```
/// <summary>
/// 作业任务调度器集合类
/// </summary>
public interface ITaskSchedulerCollection
{
    /// <summary>
    /// 注册一个作业任务调度器，实现的时候，直接返回当前调度器集合，方便链式调用
    /// </summary>
    /// <param name="taskScheduler">调度器</param>
    void Register(TaskScheduler taskScheduler);

    /// <summary>
    /// 更新一个作业任务调度器
    /// </summary>
    /// <param name="taskScheduler"></param>
    void Update(TaskScheduler taskScheduler);

    /// <summary>
    /// 根据调度器名称删除调度器
    /// </summary>
    /// <param name="name">调度器名称</param>
    void Remove(string name);

    /// <summary>
    /// 获取作业任务调度器
    /// </summary>
    /// <param name="name"></param>
    /// <returns></returns>
    TaskScheduler Get(string name);

    /// <summary>
    /// 获取所有的调度器
    /// </summary>
    /// <returns></returns>
    IEnumerable<TaskScheduler> GetAll();
}
```

此接口定义了一系列调度器操作的方法，我们再来看下其内部实现 TaskSchedulerCollection 类的定义：

```
/// <summary>
/// 作业任务调度器集合类
/// </summary>
internal class TaskSchedulerCollection : List<TaskScheduler>, ITaskSchedulerCollection
{
    /// <summary>
    /// 获取任务调度器
    /// </summary>
    /// <param name="name"></param>
    /// <returns></returns>
    public TaskScheduler Get(string name)
    {
        return this.FirstOrDefault(o => o.Name.Equals(name, StringComparison.OrdinalIgnoreCase));
    }

    /// <summary>
    /// 获取所有注册的任务调度器
    /// </summary>
    /// <returns></returns>
    public IEnumerable<TaskScheduler> GetAll()
    {
        return this;
    }

    /// <summary>
    /// 注册一个任务调度器
    /// </summary>
    /// <param name="taskScheduler"></param>
    public void Register(TaskScheduler taskScheduler)
    {
        taskScheduler.CheckNullThrowArgumentNullException("taskScheduler");
        taskScheduler.Name.CheckNullThrowArgumentNullException("name");
        taskScheduler.TaskType.CheckNullThrowArgumentNullException("taskType");

        //获取作业任务实现类的类型
        Type taskType = Type.GetType(taskScheduler.TaskType);
        taskType.CheckNullThrowArgumentNullException("taskType");

        //类型必须实现 ITask 接口
        if (!typeof(ITask).IsAssignableFrom(taskType))
        {
            throw new ApiException("注册的作业类型必须要实现 ITask 接口");
        }

        if (this.IsRegisted(taskScheduler.Name))
        {
            throw new ApiException("已经存在了名称为：{0} 的调度器".With(taskScheduler.Name));
        }
```

```
            //add
            this.Add(taskScheduler);
        }

        /// <summary>
        /// 删除作业任务调度
        /// </summary>
        /// <param name="name">调度器名称</param>
        public void Remove(string name)
        {
            var item = this.Get(name);
            if (!item.IsNull())
            {
                ((List<TaskScheduler>)this).Remove(item);
            }
        }

        /// <summary>
        /// 更新作业任务调度器
        /// </summary>
        /// <param name="taskScheduler"></param>
        public void Update(TaskScheduler taskScheduler)
        {
            taskScheduler.CheckNullThrowArgumentNullException("taskScheduler");
            taskScheduler.Name.CheckNullThrowArgumentNullException("name");

            //是否存在调度器
            var item = this.Get(taskScheduler.Name);
            //存在就更新
            if (!item.IsNull())
            {
                item.Seconds = taskScheduler.Seconds;
                item.Enabled = taskScheduler.Enabled;
                item.TaskType = taskScheduler.TaskType;
                item.StopOnError = taskScheduler.StopOnError;
                item.RunOnOneWebFarmInstance = taskScheduler.RunOnOneWebFarmInstance;
            }
        }

        /// <summary>
        /// 检测任务调度器是否重复添加
        /// </summary>
        /// <param name="name"></param>
        /// <returns></returns>
        private bool IsRegisted(string name)
        {
            return !this.Get(name).IsNull();
        }
    }
```

调度器保存集合类实现类我们直接继承了**List**集合，来实现。在最前面的如何做里，我们正是调用了这些方法将作业任务和调度器注册到这里的。

知道了如果注册调度器后，那么我们是如果进行任务的执行呢？系统里我们定义了一个 TaskThread 类，作业任务线程类里保存多个作业任务（具有相同作业间隔执行时间放置于同一作业线程类）。我们来看下其具体的实现代码：

```
    /// <summary>
    /// 作业线程（每个线程可能包含多个作业任务）
    /// </summary>
    public class TaskThread : IDisposable
    {
        private Timer _timer;
        private bool _disposed;
        private readonly Dictionary<string, Task> _tasks;

        /// <summary>
        ///
        /// </summary>
        internal TaskThread()
        {
            this._tasks = new Dictionary<string, Task>();
            this.Seconds = 10 * 60;
            this.Started = DateTime.Now;
        }

        /// <summary>
        /// 真正执行作业线程里的所有作业任务
        /// </summary>
        private void Run()
        {
            if (Seconds <= 0)
                return;

            this.LastRuned = DateTime.Now;
            this.IsRunning = true;

            foreach (Task task in this._tasks.Values)
            {
                task.Execute();
```

```
        }

        this.IsRunning = false;
    }

    /// <summary>
    /// timer 每次触发的时候执行的委托方法
    /// </summary>
    /// <param name="state"></param>
    private void TimerHandler(object state)
    {
        //停止 timer 触发
        this._timer.Change(-1, -1);

        //执行作业任务
        this.Run();

        if (this.RunOnlyOnce)
        {
            this.Dispose();
        }
        else
        {
            this._timer.Change(this.Interval, this.Interval);
        }
    }

    /// <summary>
    /// 释放作业线程
    /// </summary>
    public void Dispose()
    {
        if ((this._timer != null) && !this._disposed)
        {
            lock (this)
            {
                this._timer.Dispose();
                this._timer = null;
                this._disposed = true;
            }
        }
    }

    /// <summary>
    /// 初始化作业线程（创建 timer）
    /// </summary>
    public void InitTimer()
    {
        if (this._timer == null)
        {
            this._timer = new Timer(new TimerCallback(this.TimerHandler), null, this.Interval, this.Interval);
        }
    }

    /// <summary>
    /// 添加一个封装后的作业任务到作业线程
    /// </summary>
    /// <param name="task">作业</param>
    public void AddTask(Task task)
    {
        if (!this._tasks.ContainsKey(task.Name))
        {
            this._tasks.Add(task.Name, task);
        }
    }

    /// <summary>
    /// 作业执行间隔，单位：秒
    /// </summary>
    public int Seconds { get; set; }

    /// <summary>
    /// 作业线程启动时间，一旦启动器时间不会变化
    /// </summary>
    public DateTime Started { get; private set; }

    /// <summary>
    /// 最后一次执行作业任务时间
    /// </summary>
    public DateTime LastRuned { get; private set; }

    /// <summary>
    /// 显示当前作业线程是否正在执行作业
    /// </summary>
    public bool IsRunning { get; private set; }

    /// <summary>
    /// 获取所有作业任务列表（封装后的作业任务）
    /// </summary>
    public IList<Task> Tasks
    {
        get
```

```
            {
                var list = new List<Task>();
                foreach (var task in this._tasks.Values)
                {
                    list.Add(task);
                }
                return new ReadOnlyCollection<Task>(list);
            }
        }

        /// <summary>
        /// 执行作业任务间隔，单位：毫秒，内部使用
        /// </summary>
        private int Interval
        {
            get
            {
                return this.Seconds * 1000;
            }
        }

        /// <summary>
        /// 作业任务线程是否值运行一次，而不重复间隔执行
        /// </summary>
        public bool RunOnlyOnce { get; set; }
    }
```

从实现代码可以看出，我们使用了 System.Threading.Timer 类，作为我们的定时器。具体的实现逻辑为，通过Timer的间隔触发，触发事件来执行委托方法 TimerHandler(object state)，然后此方法又调用了void Run() 方法来循环执行作业任务。但是，我们可以看到这里使用的并不直接是实现 ITask 的类，而是 Task 类，我们先来看下Task类

```
    /// <summary>
    /// 作业任务封装类
    /// </summary>
    public class Task
    {
        /// <summary>
        /// 作业任务调度器
        /// </summary>
        private TaskScheduler _taskScheduler;

        /// <summary>
        /// Ctor for Task
        /// </summary>
        /// <param name="taskScheduler">作业任务调度器</param>
        public Task(TaskScheduler taskScheduler)
        {
            this._taskScheduler = taskScheduler;
            this.Name = taskScheduler.Name;
            this.Enabled = taskScheduler.Enabled;
            this.StopOnError = taskScheduler.StopOnError;
            this.Seconds = taskScheduler.Seconds;
            this.TaskType = taskScheduler.TaskType;
            this.RunOnOneWebFarmInstance = taskScheduler.RunOnOneWebFarmInstance;
        }

        /// <summary>
        /// 执行作业任务
        /// </summary>
        /// <param name="throwException">遇到错误是否直接抛出异常</param>
        /// <param name="dispose">执行完作业任务后，是否释放资源</param>
        public void Execute(bool throwException = false, bool dispose = true)
        {
            //对象生命作用域
            var scope = ServicesContainer.Current.Scope();

            //获取分布式协调器
            var distributedLocker = ServicesContainer.Current
                .Resolver<ITaskSchedulerDistributedLocker>(scope);

            //获取协调器保存的调度信息
            var taskSchedulerLocker = distributedLocker.Get(this.Name) ??
                new TaskSchedulerDistributedLockerObject()
                {
                    TaskSchedulerName = null,
                    LeasedUntil = new DateTime(1900, 1, 1),
                    LeasedByMachineName = null
                };

            try
            {
                //检测是否只能一个运行实例可以作业
                if (this.RunOnOneWebFarmInstance)
                {
                    //获取当前运行实例信息
                    var machineNameProvider = ServicesContainer.Current.Resolver<IMachineNameProvider>(scope);
                    var machineName = machineNameProvider.GetMachineName();

                    //必须要存在实例
                    if (String.IsNullOrEmpty(machineName))
```

```
{
    throw new ApiException("运行实例名称必须存在");
}

//调度器被其他运行实例锁住了
if (taskSchedulerLocker.LeasedUntil.HasValue
    && taskSchedulerLocker.LeasedUntil.Value >= DateTime.Now
    && taskSchedulerLocker.LeasedByMachineName != machineName)
    return;

//没有设置过期时间，或者租期已经超期，重新竞争设置
if (taskSchedulerLocker.LeasedUntil.Value < DateTime.Now)
{
    //设置租期 10 分钟，太长的话怕实例出现问题，其他实例一致没有机会去执行
    //也就是说，在分布式的情况下，一个作业任务最大出现问题的机会只会有 10 分钟
    //这样防止作业任务一致不起作用
    DateTime leasedUntil = DateTime.Now.AddMinutes(10);
    //构造一个锁对象
    taskSchedulerLocker = new TaskSchedulerDistributedLockerObject()
    {
        LeasedUntil = leasedUntil,
        TaskSchedulerName = this.Name,
        LeasedByMachineName = machineName
    };
    //锁定当前运行实例为此任务调度器执行宿主
    distributedLocker.Lock(taskSchedulerLocker);
    //设置当前属性
    this.LeasedUntil = leasedUntil;
    this.LeasedByMachineName = machineName;
}
}

//当前正在作业
this.IsRunning = true;

//获取作业任务类型
Type taskType = Type.GetType(this._taskScheduler.TaskType);

//创建作业任务实例
var instance = (ITask)ServicesContainer.Current.ResolverUnregistered(taskType, scope);

//执行作业任务
if (!instance.IsNull())
{
    instance.Execute(new TaskExecuteContext() { TaskScheduler = this._taskScheduler });
}

}
catch (Exception exc)
{
this.Enabled = !this.StopOnError;

//log error
var logger = NullLogger.Instance;

object _logger = null;
if (ServicesContainer.Current.TryResolver(typeof(ILogger), scope, out _logger))
{
    logger = (ILogger)_logger;
}

logger.Error(string.Format("运行调度错误: '{0}' 错误信息: {1}", this.Name, exc.Message), exc);

if (throwException)
{
    throw;
}
}

//dispose all resources
if (dispose)
{
    scope.Dispose();
}

this.IsRunning = false;
}

/// <summary>
/// 当前作业任务是否正在执行中....
/// </summary>
public bool IsRunning { get; private set; }

/// <summary>
/// 遇到错误是否定制作业任务执行
/// </summary>
public bool StopOnError { get; private set; }

/// <summary>
/// 获取作业任务名称
/// </summary>
public string Name { get; private set; }
```

```
        /// <summary>
        /// 获取当前任务是否允许运行
        /// </summary>
        public bool Enabled { get; private set; }

        /// <summary>
        /// 多少秒执行一次
        /// </summary>
        public int Seconds { get; private set; }

        /// <summary>
        /// ITask 作业任务的类型
        /// </summary>
        public string TaskType { get; private set; }

        /// <summary>
        /// 在分布式站点中，确保一个作业任务只运行一个实例，防止并发重复作业
        /// </summary>
        public bool RunOnOneWebFarmInstance { get; private set; }

        /// <summary>
        /// 作业调度器再特定运行实例里运行的到期时间
        /// 租约到期时间(一般默认租约到期时间为 30 天)
        /// </summary>
        public DateTime? LeasedUntil { get; set; }

        /// <summary>
        /// 作业调度器再特东运行实例里运行的实例名称
        /// 租约机器（在分布式执行的时候，因为每个 IIS 都会有一套还行任务，
        /// 因此我们先签约第一个执行的机器来执行，其他分布式机器比较此租
        /// 约机器名称来判断是否执行）
        /// </summary>
        public string LeasedByMachineName { get; set; }
    }
```

从代码可以看出，Task类是对一个调度器的封装，而调度器里带有一个TaskType的属性，我们来看下此类一个唯一的Execute方法，它首先使用了一个分布式锁服务来获取分布式锁，如果分布式锁属于当前工作实例，那么就通过TaskType创建出具体的作业任务，最后调用ITask.Execute方法来进行执行作业任务。

下面我们来看一下分布式情况下，我们怎么来实现控制作业任务运行。在分布式系统下，我们有时候将站点运行在多个实例上，这样每个站点都是独立的，每台作业任务也是独立的，这种情况下，很多时候我们希望作业任务每次都只能在一台实例上运行，防止同时作业任务统一时间被多次执行。因此我们定义了一个分布式作业任务锁：ITaskSchedulerDistributedLocker 接口，用于控制协调各运行实例作业调度的运行，此接口定义了2个方法，定义如下：

```
    /// <summary>
    /// 此接口用于分布式作业任务锁接口，当我们将应用程序部署在多台机上
    /// 或者 IIS 站点启动多进行工作模式的时候，我们需要一个机制，
    /// 确认某些作业任务只工作在一台进程里，所以需要一个分布式锁来确认正常工作
    /// 因此实现此类的外部保存环境必须能够被所有运行实例访问，不能放置于运行实例缓存里面等
    /// </summary>
    public interface ITaskSchedulerDistributedLocker
    {
        /// <summary>
        /// 获取调度器分布式锁对象信息
        /// </summary>
        /// <param name="taskSchedulerName">作业任务调度器名称</param>
        TaskSchedulerDistributedLockerObject Get(string taskSchedulerName);

        /// <summary>
        /// 设置调度器分布式锁
        /// </summary>
        /// <param name="locker"></param>
        void Lock(TaskSchedulerDistributedLockerObject locker);
    }
```

系统框架里会自动调用其实现类，首先使用Get方法，检测当前作业任务调度器是否被其他运行实例锁定了。如果锁定了，那么当前实例上的作业任务将不会运行。如果没有锁定，那么就将当前实例作为作业任务实例锁定，直到过期时间。Lock 方法参数定义如下：

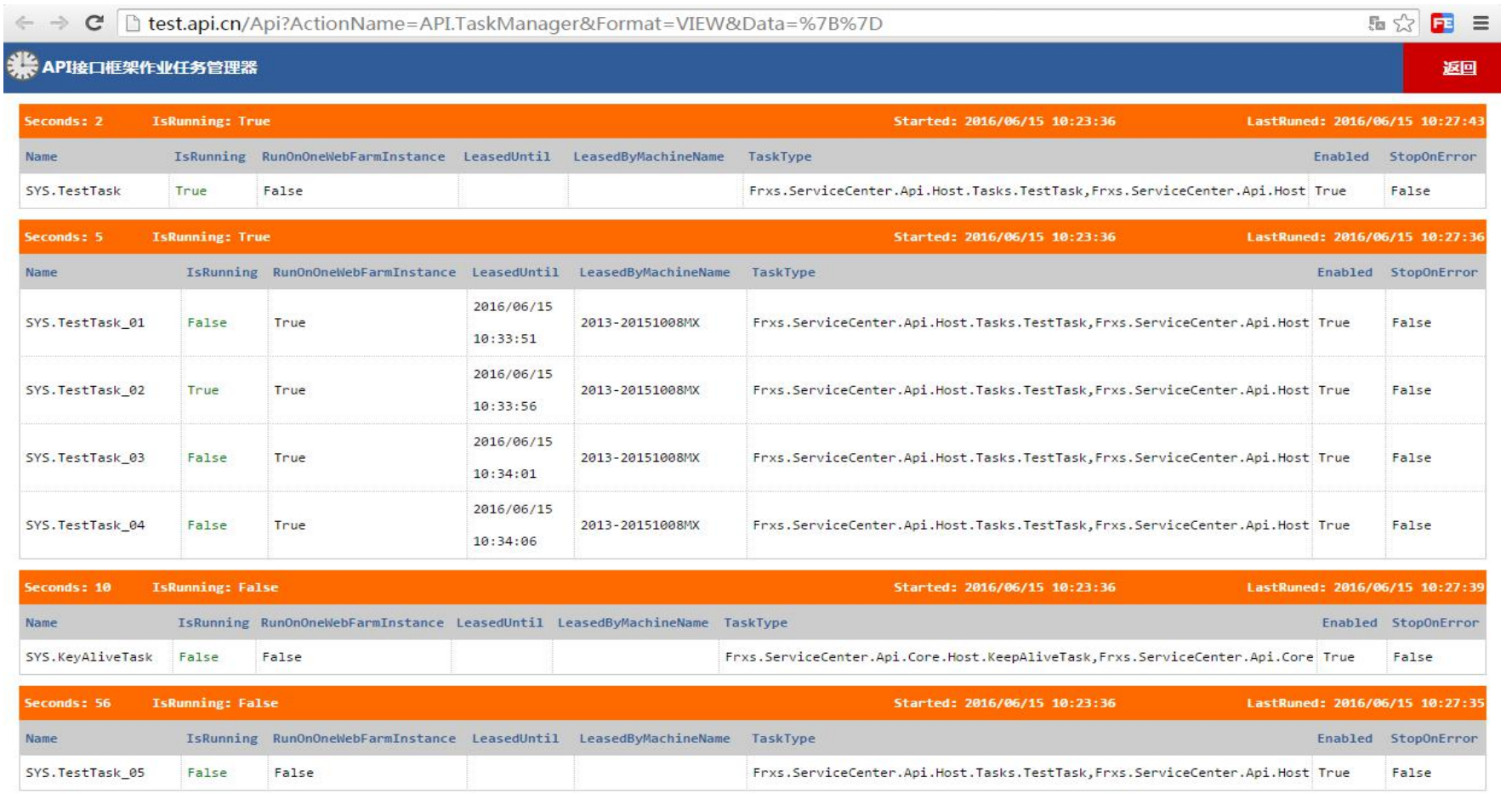
```
    /// <summary>
    /// 用于分布式锁
    /// </summary>
    public class TaskSchedulerDistributedLockerObject
    {
        /// <summary>
        /// 作业任务调度器名称，全局唯一
        /// </summary>
        public string TaskSchedulerName { get; set; }

        /// <summary>
        /// 作业调度器再特定运行实例里运行的到期时间
        /// 租约到期时间(一般默认租约到期时间为 30 天)
        /// </summary>
        public DateTime? LeasedUntil { get; set; }
```

```
        /// <summary>
        /// 作业调度器再特东运行实例里运行的实例名称
        /// 租约机器（在分布式执行的时候，因为每个 IIS 都会有一套还行任务，
        /// 因此我们先签约第一个执行的机器来执行，其他分布式机器比较此租
        /// 约机器名称来判断是否执行）
        /// </summary>
        public string LeasedByMachineName { get; set; }
    }
```

具体属性说明上面注释已经很清楚了，就不在说明。我们现在来看下系统框架默认的分布式锁实现

```
    /// <summary>
    /// 默认的分布式协调器
    /// </summary>
    internal class DefaultTaskSchedulerDistributedLocker : ITaskSchedulerDistributedLocker
    {
        /// <summary>
        /// 用于保存分布式锁的 key
        /// </summary>
        private const string Key = "Task.Scheduler.{0}";

        /// <summary>
        /// 缓存器
        /// </summary>
        private ICacheManager _cacheManager;

        /// <summary>
        /// 默认使用缓存来保存锁定
        /// </summary>
        /// <param name="cacheManager">缓存器必须为单独的公共缓存服务器，不能使用单独的内存缓存器</param>
        public DefaultTaskSchedulerDistributedLocker(ICacheManager cacheManager)
        {
            this._cacheManager = cacheManager;
        }

        /// <summary>
        ///
        /// </summary>
        /// <param name="taskSchedulerName">作业任务调度器名称</param>
        /// <returns></returns>
        public TaskSchedulerDistributedLockerObject Get(string taskSchedulerName)
        {
            return this._cacheManager.Get<TaskSchedulerDistributedLockerObject>(Key.With(taskSchedulerName), () => null);
        }

        /// <summary>
        ///
        /// </summary>
        /// <param name="locker"></param>
        public void Lock(TaskSchedulerDistributedLockerObject locker)
        {
            this._cacheManager.Set(Key.With(locker.TaskSchedulerName), locker, 60);
        }
    }
```

从代码可以看出，我们使用了缓存来作为分布式锁，当然我们还可以使用数据库或者其他存储介质来实现（具体实现不能依赖于运行实例进程）。为了方便查看作业任务，我们专门扩展了一个作业任务管理器项目。具体实现项目为：**1.3.1Frxs.ServiceCenter.Api.Core.Task.Management** 使用方式，只要我们HOST项目引用了此项目（或者dll包）就可以使用。界面如下：



5.17 SDK全自动打包生成器（C#，安卓，IOS）/接口文档自动生成器

为了方便C#客户端对接接口，我们开发了一个专门针对C#语言的SDK代码生成器，项目地址为：
1.1.2.Frxs.ServiceCenter.Api.Core.SdkBuilder.CSharp 具体的实现可以去看下此项目，此项目也是完全按照插件的方式来进行开发，可以利用此项目来了解怎么来扩展接口框架。要使用此代码生成器，只要在HOST项目里引用此项目即可。运行HOST项目后，系统会自动加载此扩展项目，安卓和IOS代码生成器有兴趣的可以扩展下。此项目包含3个命令，我们可以直接使用快捷命令方式下载DLL，源代码，文档：

1. http://HOST/CSharpDownSdk 下载C#客户端调用代码dll包
2. http://HOST/CSharpDownSource 下载C#客户端调用源码包，可以在VS里直接打开
3. http://HOST/DocBuilder 下载接口文档，上送下送参数说明等

6.开发建议以及命名约定

6.1 Action接口命名建议

实际接口开发，系统框架没有强制定义命名规则，但是有些比较好的实践，我们可以来参考下，框架建议一个良好的接口类命名规则为，{模块名称}{操作}Action.cs。我们把操作动词放置了模块名称后面，是为了在我们查找接口的时候能方便的将相同模块的接口放置在一起。

6.2 RequestDto上送参数接受对象命名建议

6.3 ResponseDto下送数据对象命名建议

7.代码注释

系统框架支持代码注释自动生成文档，所以在开发扩展接口项目的时候，类,属性的注释代码填写清楚系统框架就会自动生成说明文档