关于 Lind.DDD 框架里 API 框架的技术点说明

讲解：张占岭

主要框架：Lind.DDD

目录

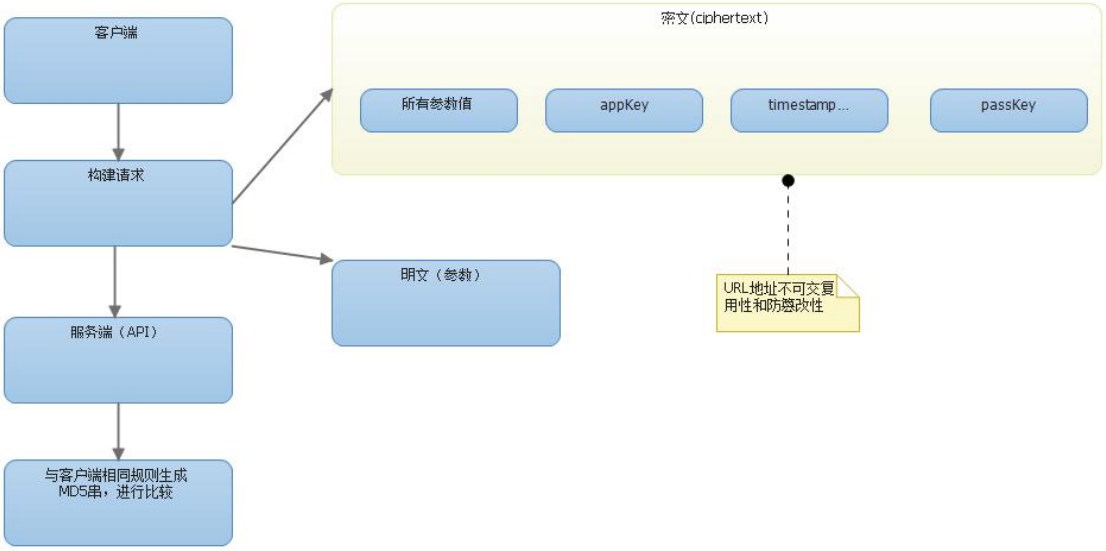
关于 Lind.DDD.Authorization.....2
关于授权的原理.....2
关于 ApiValidateModelConfig.....2
关于 Lind.DDD.CacheConfigFile.....3
如何为你的 API 项目注入授权模块.....4
关于服务端收取过滤器 ApiValiadateFilter.....5
如何在客户端生产加密授权串.....5
关于请求类与响应类.....5
客户端如何做分页.....8

关于 Lind.DDD.Authorization

Lind.DDD 为我们提交了强大的 API 校验组件，只需要在全局或者要授权的 controller 上添加对应的过滤器即可完成授权的过程，这样，你的 API 就安全多了。

关于授权的原理

客户端在向 API 服务端获取数据时，需要先申请一个 **appkey** 作为自己的标识，当然这也是双方约定的，我们可以叫做公钥，而真正做数据校验的不是它，它只是一个唯一标识，对外公开，真正做数据加密的叫 **passKey**，它是保存在双方内部的，不对外公开，所以叫密钥，在客户端向 API 服务端通讯时，需要将这个 **passKey** 连同参数和 **appkey** 传到服务端，由服务端做相同的校验码生产逻辑，最后两者进行比较，相同即验证通过。



关于 ApiValidateModelConfig

`ApiValidateModelConfig` 主要在服务端存储所有被收取的客户端的信息，它是一个列表集合，由 `AppKey,AppName,PassKey,ExpireDate` 等元素组成，它们具体的含义如下：

```
/// <summary>
/// 服务端 - 客户端数据校验模型
/// </summary>
[Serializable]
public class ApiValidateModel
{
    /// <summary>
    /// 项目键，用于网络传输
    /// </summary>
    public string AppKey { get; set; }
    /// <summary>
    /// 项目名称
    /// </summary>
    public string AppName { get; set; }
    /// <summary>
    /// 密钥
    /// </summary>
    public string PassKey { get; set; }
    /// <summary>
    /// 过期时间
    /// </summary>
    public DateTime ExpireDate { get; set; }
}
```

而这个实体在服务端校验时，会从配置文件 XML 中反射出来，以遍进行比较，当然，你的配置文件如果没有修改，它会直接从内存里进行获取，这个逻辑由 CacheConfig 控制。

关于 Lind.DDD.CacheConfigFile

它在早期的大叔框架里就已经出来了，主要用于做配置文件缓存的，当缓存文件被修改后，它的信息将重新被加载，否则将从内在中来进行获取，这个文件需要管理员在服务端进行维护，在添加和删除配置时，需要做修改，当然，我们也完全可以把它持久化到其它数据库里，如 sqlserver,redis 等介质。

```
<?xml version="1.0" encoding="utf-8" ?>
- <ApiValidateModelList>
  - <ApiValidateModel>
      <AppKey>lind</AppKey>
      <AppName>大叔</AppName>
      <PassKey>lind123</PassKey>
      <ExpireDate>2016-12-12</ExpireDate>
    </ApiValidateModel>
  - <ApiValidateModel>
      <AppKey>zzl</AppKey>
      <AppName>占岭</AppName>
      <PassKey>zzl123</PassKey>
      <ExpireDate>2016-12-12</ExpireDate>
    </ApiValidateModel>
  </ApiValidateModelList>
```

如何为你的 API 项目注入授权模块

为 API 项目注入授权功能很是容易，直接在对应的 controller 上添加过滤器 Lind.DDD.Authorization.Api.ApiValiadateFilter 即可。

```
/// <summary>
/// 有用户收取的接口
/// </summary>
/// <returns></returns>
[ApiValiadateFilter]
public PagedList<SOA.DTO.ResponseUserInfo> Get(int validate)
{
    return Get();
}
```

或者在 Global.asax 里添加全局的过滤，也是可以的，值得注意的是，如果你添加的是全局过滤器，如果希望有一些 Controller 不被授权，即可以被匿名访问，那你也只需要为指定的控制器添加 AllowAnonymous 特性即可。

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        config.Routes.MapHttpRoute(
            name: "DefaultApi",
            routeTemplate: "api/{controller}/{id}",
            defaults: new { id = RouteParameter.Optional }
        );

        config.Filters.Add(new Lind.DDD.Filters.ExceptionErrorLoggerAttribute());
        //日志拦截器
        //config.Filters.Add(new Lind.DDD.Filters.ActionLoggerAttribute());
        //校验拦截器
        config.Filters.Add(new Lind.DDD.Authorization.Api.ApiValiadateFilter());

    }
```

关于服务端收取过滤器 ApiValiadateFilter

ApiValiadateFilter 是服务端的收取核心组件，它会拦截指定的 api 控制器，然后进行授权检查，如果没有被标示 AllowAnonymous，它将会进行校验，具体就是将请求参数进行排序，组件，并连同 passkey(由客户端传来的 appKey 进行查询，得到的 ApiValidateModel 实体)生成新的 MD5 加密串，与客户端传过来的密钥进行对比，匹配即有效，否则返回 403 无权访问，最新的 api 校验的新功能如下：

一、统一校验模块

二、统一参数组合的生成

三、UTC 时间戳的引入，参数有效性校验（1 小时有效）

四、双方约定的密钥，请求的防伪造

如何在客户端生产加密授权串

Lind.DDD 框架为我们提供了生成请求密钥的方法，你需要做的只是将所有参数添加到字典，然后调用对应的方法即可，这对于.net 开发人员来说，绝对是个福音！

```
using (var http = new HttpClient())
{
    var param = new NameValueCollection();
    param.Add("validate", "1");
    param.Add("AppKey", "lind");
    string ciphertext = ApiValidateHelper.GenerateCipherText(param, "lind1234");
    var response = http.GetAsync("http://localhost:24334/api/UserApi?"
        + param.ToUrl()
        + "&ciphertext="
        + ciphertext).Result;

    if (response.StatusCode != HttpStatusCode.OK)
        return Content(response.ReasonPhrase);

    var model = JsonConvert.DeserializeObject<PagedList<SOA.DTO.ResponseUserInfo>>
        (response.Content.ReadAsStringAsync().Result);

    return View("UserList", model);
}
```

关于请求类与响应类

API 服务端与客户端约定了请求与响应的模型，它们都是抽象类，提供最基础的功能，下面简单说一下：

- 请求对象

对应于 DTO 请求类，它继承自抽象类 RequestBase，它提供了传输标示，分页，排序，筛选字段等功能。

```
/// <summary>
/// DTO请求体基类
/// </summary>
public abstract class RequestBase
{
    /// <summary> ...
    public RequestBase()...

    /// <summary> ...
    private Func<PropertyInfo, bool> queuePredicate;

    #region 公用属性，不进行参数过滤
    /// <summary> ...
    public string GuidKey { get; set; }
    /// <summary> ...
    public string Page { get; set; }
    /// <summary> ...
    public string Sort { get; set; }
    /// <summary> ...
    public string ContainFields { get; set; }
    /// <summary> ...
    public virtual bool IsValid...


    #endregion

    #region Methods
    /// <summary> ...
    public IEnumerable<KeyValuePair<string, object>> GetProperyiesDictionary()...
    /// <summary> ...
    public Paging.PageParameters GetPageParameters()...
    /// <summary> ...
    public IEnumerable<KeyValuePair<string, object>> GetSortDictionary()...
    /// <summary> ...
    public IEnumerable<RuleViolation> GetRuleViolations()...

    /// <summary> ...
    public virtual string GetRuleViolationMessages()...
    #endregion
}
```

- 响应对象

对应于 DTO 响应类，它继承自抽象类 ResponseBase，它提供了 RequestBase 里的传输标示，标明了是否为同一个请求，响应的字段等。

```
/// <summary>
/// 响应体基类
/// </summary>
public abstract class ResponseBase
{
    /// <summary>
    /// 标示码，与RequestBase里的GuidKey对应
    /// </summary>
    public string GuidKey { get; set; }
    /// <summary>
    /// 相应的字段
    /// </summary>
    public string SerializableFields { get; set; }
    /// <summary>
    /// 初始化ResponseMessage
    /// </summary>
    public ResponseBase()
        : this(null)
    { }
    /// <summary>
    /// 初始化ResponseMessage
    /// </summary>
    /// <param name="serializableFields">希望返回的字段</param>
    public ResponseBase(string serializableFields)
    {
        this.SerializableFields = serializableFields;
    }

}
```

- 响应返回对象

对应于 DTO 的返回结果，它由密封类 ResponseMessage 提供，返回它的实例即可，它会提供返回状态码，唯一标识，返回对象，错误码和错误信息等。

```
/// <summary>
/// 返回的相应对象
/// Result: 分页返回
/// Result: 集合返回
/// Result: 实体返回
/// </summary>
public sealed class ResponseMessage
{
    string _serializableFields;
    /// <summary>
    /// 初始化ResponseMessage
    /// </summary>
    public ResponseMessage()
        : this(null)
    { }
    /// <summary>
    /// 初始化ResponseMessage
    /// </summary>
    /// <param name="serializableFields">希望返回的字段</param>
    public ResponseMessage(string serializableFields)
    {
        this._serializableFields = serializableFields;
        this.Status = 100;
    }
    /// <summary>
    /// 标示码，与RequestBase里的GuidKey对应
    /// </summary>
    public string GuidKey { get; set; }
    /// <summary>
    /// 状态码
    /// 100成功，200失败
    /// </summary>
    public int Status { get; set; }
    /// <summary>
    /// 业务错误代码
    /// </summary>
    public string ErrorCode { get; set; }
    /// <summary>
    /// 错误消息
    /// </summary>
    public string ErrorMessage { get; set; }
    /// <summary>
    /// 只写属性，返回的对象，它不被序列化，只在服务端内存临时存储
    /// 它通常是一个ReponseBase对象或者集合
```

客户端如何做分页

分页对于每个项目来说都是必要的，对于面向服务的 API 来说了是必须的，我们 Lind.DDD 对分页一定进行了封装，在这里为大家简单说一下。

统一的 DTO 请求基类 RequestBase，它主要实现请求方调用时的字段过滤（ContainFields），分页控制（Page），传输标示(GuidKey)，按字段排序时（Sort），请求方只需要传输相应的参数即可，代码如下：

```
/// <summary>
/// DTO请求体基类
/// </summary>
public abstract class RequestBase
{
    /// <summary> ...
    public RequestBase()...

    /// <summary>
    /// 以属性作为查询条件,去掉为空的属性和公用属性
    /// </summary>
    private Func<PropertyInfo, bool> queuePredicate;

    #region 公用属性，不进行参数过滤
    /// <summary>
    /// 本次请求唯一标示
    /// </summary>
    public string GuidKey { get; set; }
    /// <summary>
    /// 分页参数，页码和每页显示的记录数
    /// 例：Page=1,5，表示获取第一页，每页显示5条
    /// </summary>
    public string Page { get; set; }
    /// <summary>
    /// 排序相关 1 升序,-1 降序
    /// 例：Sort=email|1,username-1
    /// </summary>
    public string Sort { get; set; }
    /// <summary>
    /// 需要返回的字段，其它字段将不会被序列化，这些字段使用,分开
    /// 例：ContainFields=username,realname,email
    /// </summary>
    public string ContainFields { get; set; }
```

在服务端进行分页方法实现时，返回统一的 Lind.DDD.Paging.PagedList<T>对象，而在业务方法处理时，只需要调用请求类的 GetPageParameters()方法即可拿到客户端传来的分页对象，而从数据库查出来的对象也可以使用 MapToPage<T>()这个方法来映射成指的 DTO 对象，代码如下：

```
public PagedList<ResponseUserInfo> GetUserPages(RequestUserInfo request)
{
    request = request ?? new RequestUserInfo();

    var sort = request.GetSortDictionary();

    var linq = userRepository.GetModel()
                             .OrderByDescending(i => i.Id)
                             .ToPagedList(request.GetPageParameters())
                             .MapToPaged<UserInfo, DTO.ResponseUserInfo>();
    return linq;
}
```