# 混合式开发框架
# Web API 整合说明书

V1.0

| 序号 | 修改人 | 修改日期 | 修改后版本 | 修改说明 |
|---|---|---|---|---|
| 1 | 伍华聪 | 2015-11-23 | V1.0 | 初稿 |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |

# 目 录

**1. 引言 ..... 3**

1.1. 背景 ..... 3
1.2. 编写目的 ..... 6
1.3. 参考资料 ..... 7
1.4. 术语与缩写 ..... 7

**2. WEB API 的特点 ..... 7**

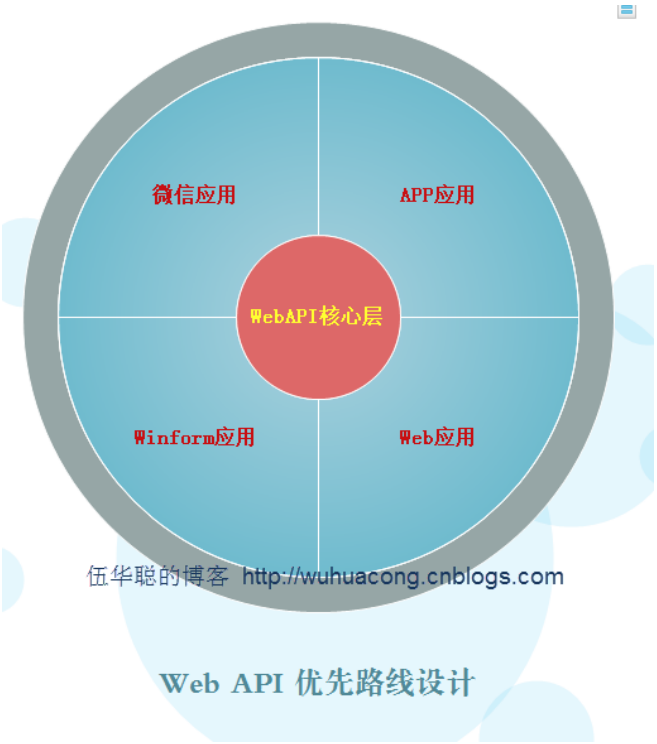
2.1. WEB API 的接口访问分类 ..... 7
2.2. WEB API 使用安全签名的实现 ..... 8
2.3. WEB API 使用安全令牌的实现 ..... 12
2.4. WEB API 设计分析 ..... 14
2.4.1. Web API 的控制器设计分析 ..... 14
2.4.2. Web API 客户端（混合式 Winform 框架模块）的调用 ..... 17
2.5. WINFORM 界面调用 WEBAPI 的过程 ..... 23
2.6. ASP.NET WEB API 的开发 ..... 28
2.6.1. 常规的 Web API 项目开发 ..... 28
2.6.2. 基于混合式框架及代码生成工具的 Web API 项目开发 ..... 30
2.7. WEB API 接口的测试 ..... 34
2.8. WEB API 应用支持 HTTPS ..... 37
2.8.1. HTTPS 基础知识介绍 ..... 37
2.8.2. SSL 基础知识介绍 ..... 38
2.8.3. 支持 SSL 的 CA 证书购买 ..... 39
2.8.4. Web API 应用支持 HTTPS ..... 41
2.9. WEB API 自定义异常结果的处理 ..... 43
2.9.1. 常规的异常处理 ..... 43
2.9.2. 地址接口异常处理 ..... 46
2.10. WEB API 接口设计经验总结 ..... 48
2.10.1. 在接口定义中确定 MVC 的 GET 或者 POST 方式 ..... 48
2.10.2. 动态对象的接口定义 ..... 49
2.10.3. 集合和分页的处理 ..... 50
2.10.4. 混合框架界面整合 Web API 接口 ..... 52
2.11. 开发框架中使用事务进行数据的统一处理 ..... 55
2.11.1. 框架的事务支持 ..... 55

2.11.2. 框架事务接口的使用 .....59
2.11.3. 混合框架中 Web API 的封装和调用.....62

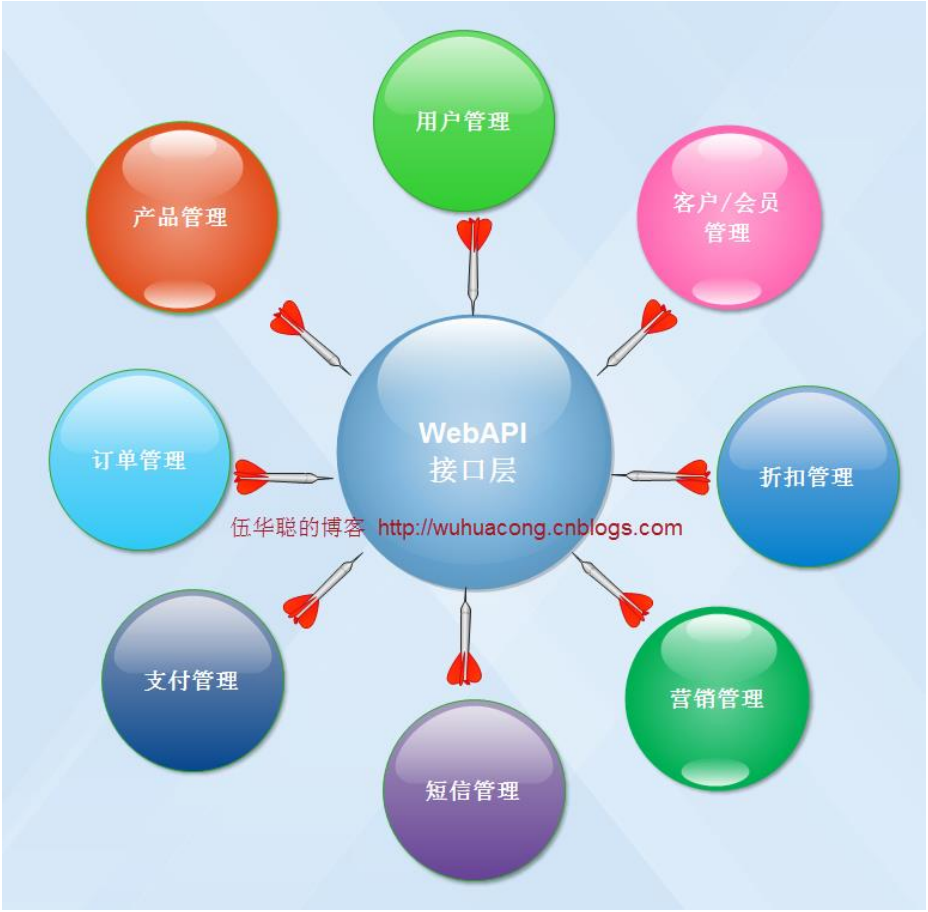
# 1. 引言

## 1.1. 背景

Web API 是一种应用接口框架，它能够构建 HTTP 服务以支撑更广泛的客户端（包括浏览器，手机和平板电脑等移动设备）的框架， ASP.NET Web API 是一种用于在 .NET Framework 上构建 RESTful 应用程序的理想平台。在目前发达的应用场景下，我们往往需要接入 Winform 客户端、APP 程序、网站程序、以及目前热火朝天的微信应用等，这些数据应该可以由同一个服务提供，这个就是我们所需要构建的 Web API 平台。由于 Web API 层作为一个公共的接口层，我们就很好保证了各个界面应用层的数据一致性。
通过上面的了解，我们可以知道，所有外部的应用，其实都可以基于一个相同的 Web API 核心开展的，如下图所示。

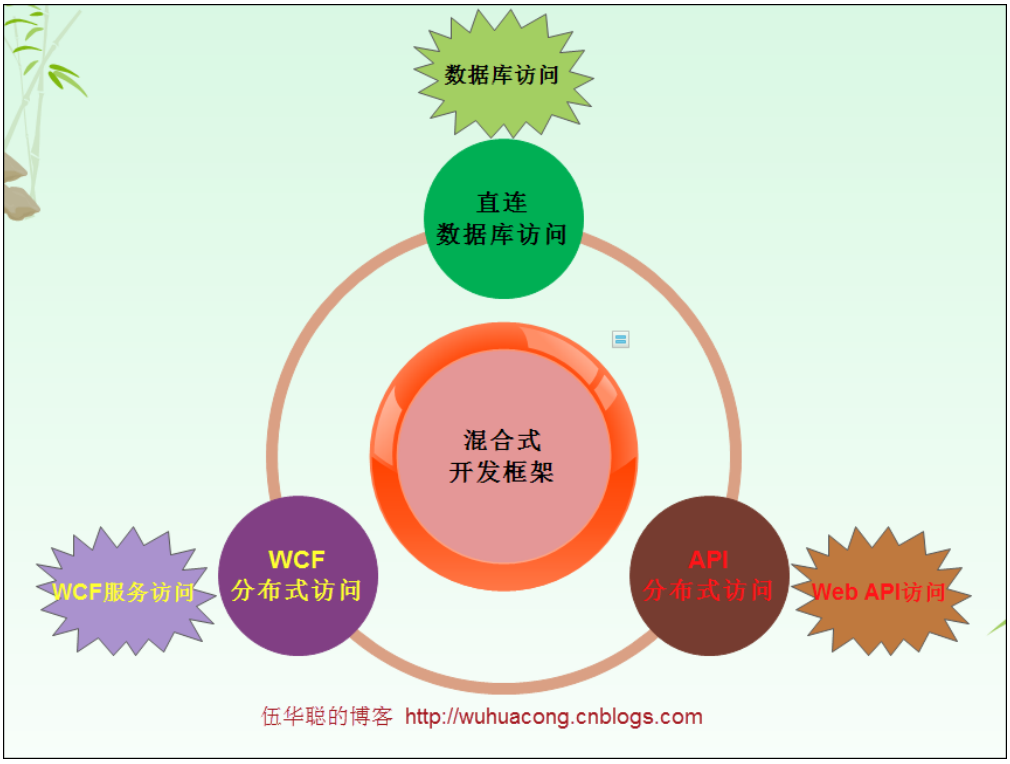



在当前大平台，大应用的背景下，可以基于一个整体的平台，构建很多应用生态链，这样就可以把 Web API 作为核心层，可以在上面开发我们各种企业业务应用了。

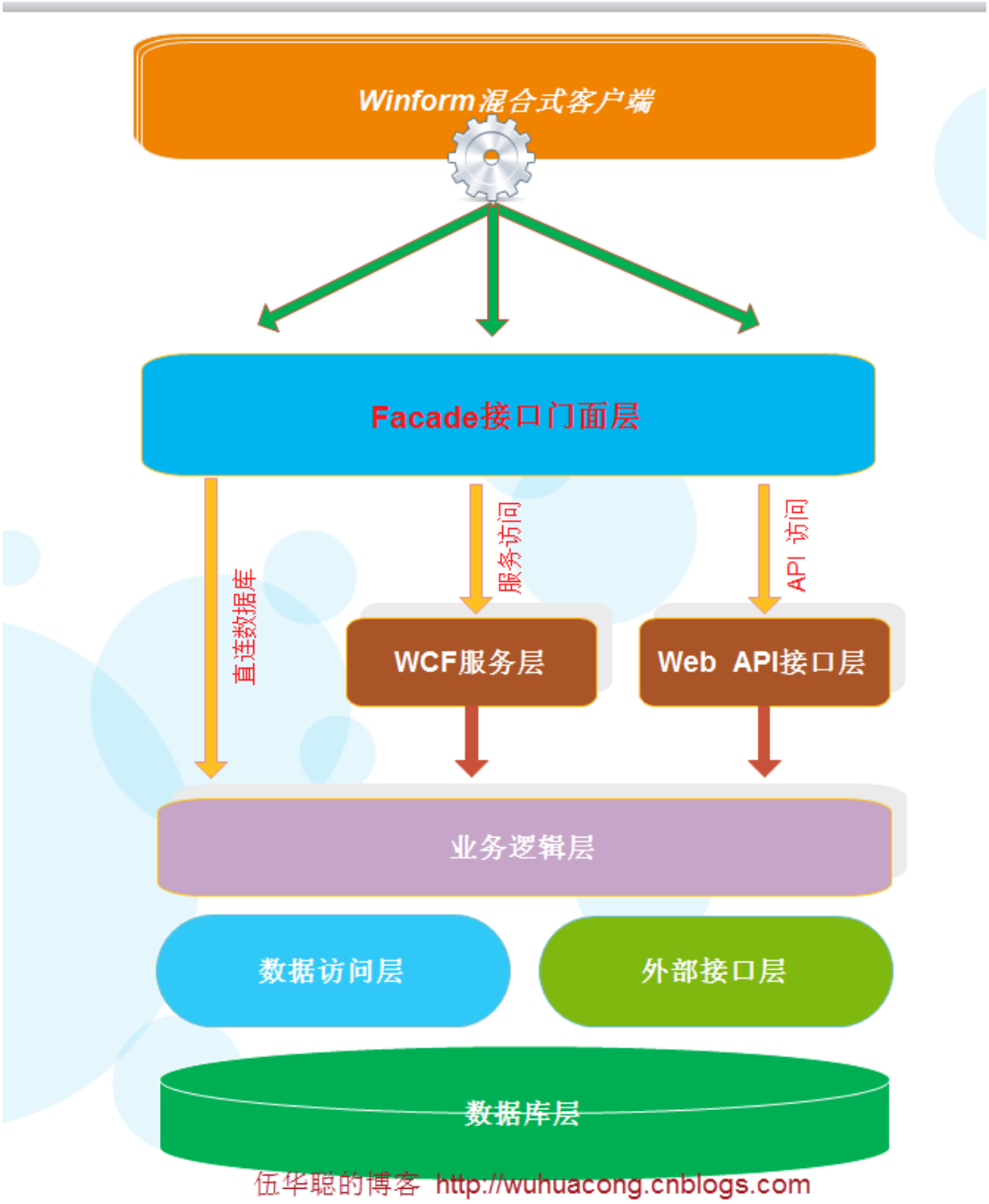



在 Winform 混合式框架的界面里面，我们除了可以利用直接访问数据库方式，以及采用访问分布式 WCF 服务的方式接入，还可以使得它能够访问 Web API 的数据服务，从而构建成一个适应性更加广泛、功能更加强大的混合式开发框架模式；对于 Web API，由于它提供的是一种无状态的接口访问，而且往往 Web API 一般为了多种客户端接入的需要，可能需要发布在公网上进行访问，因此我们需要更加注重 Web API 接口层的安全性。Web API 作为整个纽带的核心，在整个核心层需要考虑到统一性、稳定性、以及安全性等方面因素。

除了直连数据库访问的传统模式，WCF 分布式访问的 WCF 服务访问模式，还可以接入 API 分布式访问的 Web API 接口模式，他们的关系构成了一个完整的 Winform 应用体系，如下图所示。




混合式框架的实现细节，就是通过一个类似开关模式的配置模块，确定是采用直接访问数据库方式，还是访问 WCF 服务的方式，它们两者是统一到一个 Facade 接口门面层上，如果考虑到 Web API 层，基于混合式的架构，也就是在这个 Facade 接口门面层上增加多一个 Web API 的接口的封装成即可。具体整个框架的架构图如下所示。




## 1.2. 编写目的

本文档主要介绍《Winform 混合式开发框架》中应用到的 Web API 的说明、框架整合、框架开发应用方面的操作使用。

## 1.3. 参考资料

| 序号 | 名称 | 版本/日期 | 来源 |
|---|---|---|---|
| 1 | 《Winform 开发框架-架构设计说明书.doc》 | | 内部 |
| 2 | | | 内部 |
| 3 | | | 内部 |
| 4 | | | 内部 |

## 1.4. 术语与缩写

**1** 在本文件中出现的“系统”一词，除非特别说明，均适用于《WCF开发框架》、《Winform开发框架》。

2 在本文安装.NET框架中，除非特别说明，均指.NET 4.0框架。

# 2. Web API 的特点

## 2.1. Web API 的接口访问分类

Web API 接口的访问方式，大概可以分为几类：

1）一个是使用用户令牌，通过 Web API 接口进行数据访问。这种方式，可以有效识别用户的身份，为用户接口返回用户相关的数据，如包括用户信息维护、密码修改、或者用户联系人等与用户身份相关的数据。

2）一种是使用安全签名进行数据提交。这种方式提交的数据，URL 连接的签名参数是经过安全一定规则的加密的，服务器收到数据后也经过同样规则的安全加密，确认数据没有被中途篡改后，再进行数据修改处理。因此我们可以为不同接入方式，如 Web/APP/Winform 等不同接入方式指定不同的加密秘钥，但是秘钥是双方约定的，并不在网络连接上传输，连接传输的一般是这个接入的 AppID，服务器通过这个 AppID 来进行签名参数的加密对比，这种方式，类似微信后台的回调处理机制，它们就是经过这样的处理。

3）一种方式是提供公开的接口调用，不需要传入用户令牌、或者对参数进行加密签名的，这种接口一般较少，只是提供一些常规的数据显示而已。

下面图示就是这几种接入方式的说明和大概应用场景。

## 2.2. Web API 使用安全签名的实现

首先我们为用户注册的时候，需要由我们认可的终端发起，也就是它们需要进行安全签名，后台确认签名有效性，才能正常实现用户注册，否则遭到伪造数据，系统就失去原有的意义了。

```
/// <summary>
/// 注册用户信息接口
/// </summary>
public interface IUserApi
{
    /// <summary>
    /// 注册用户处理，包括用户名，密码，身份证号，手机等信息
    /// </summary>
    /// <param name="json">注册用户信息</param>
    /// <param name="signature">加密签名字符串</param>
    /// <param name="timestamp">时间戳</param>
    /// <param name="nonce">随机数</param>
    /// <param name="appid">应用接入ID</param>
    /// <returns></returns>
    ResultData Add(UserJson json,
        string signature, string timestamp, string nonce, string appid);
}
```

我们一般需要设计 Web API 对象的接口的时候，需要考虑安全性的原因，也就是需要增加更多的一些字段信息了。如可以在增删改这些接口，除了传入 token 信息外（标识具体用户），也还是需要传入签名信息，如下接口所示。

```
/// <summary>
/// 插入指定对象到数据库中
/// </summary>
/// <param name="info">指定的对象</param>
/// <returns>执行操作是否成功。</returns>
public virtual CommonResult Insert(T info, string token, string signature, string timestamp, string nonce, string appid)
```

在接口里面，我们就需要对用户的权限和签名信息进行校验，然后在进行下一步的数据处理，如果校验权限和参数完整性不通过，则会被拦截，不执行数据库的处理了。

```
//如果用户token检查不通过，则抛出MyApiException异常。
//检查用户是否有权限，否则抛出MyDenyAccessException异常
base.CheckAuthorized(AuthorizeKey.InsertKey, token, signature, timestamp, nonce, appid);
```

除了这些对数据修改的特殊性接口，有时候我们还需要查找等类似的，不对数据产生变化的接口，只需要传入令牌即可，如下接口所示。

```
/// <summary>
/// 查询数据库,检查是否存在指定ID的对象
/// </summary>
/// <param name="id">对象的ID值</param>
/// <returns>存在则返回指定的对象,否则返回Null</returns>
[HttpGet]
public virtual T FindByID(string id, string token)
{
    //如果用户token检查不通过，则抛出MyApiException异常。
    //检查用户是否有权限，否则抛出MyDenyAccessException异常
    base.CheckAuthorized(AuthorizeKey.ViewKey, token);

    T info = baseBLL.FindByID(id);
    return info;
}
```

我们可以看到，上面还是会对 token 进行校验，不过少了很多签名所需的日期标识、随机数，完整性校验签名，应用 ID 等参数。

我们会根据用户的 token 进行解析，如果是正常的 token 并可以通过解析，那么获取对应用户的权限，判断是否可以进行下一步处理即可。如果顺利通过，那么访问数据库，把所需的数据返回给调用者即可。

上面提到了用户令牌，用户令牌是一个类似实际生活的通行证，是通过用户名、密码等信息获取到的一个安全令牌，可以在多个接口进行传递的字符串，减少密码参数的传输，提高安全性。

这个用户令牌，一般由单独的接口产生，我们一般放到 AuthController 里面，这个控制器负责用户令牌相关的处理调用。

其实我们获得用户的令牌，也是需要进行用户安全签名认证的，这样我们才有效保证用户身份令牌获取的合法性。上面接口，除了 info 对象为对象创建的参数外，其他几个参数，都是为了安全性的考虑而加入的。

```
/// <summary>
/// 系统认证等基础接口
/// </summary>
public interface IAuthApi
{
    /// <summary>
    /// 注册用户获取访问令牌接口
    /// </summary>
    /// <param name="username">用户登录名称</param>
    /// <param name="password">用户密码</param>
    /// <param name="signature">加密签名字符串</param>
    /// <param name="timestamp">时间戳</param>
    /// <param name="nonce">随机数</param>
    /// <param name="appid">应用接入ID</param>
    /// <returns></returns>
    TokenResult GetAccessToken(string username, string password,
        string signature, string timestamp, string nonce, string appid);
}
```

上面介绍到的参数，我们提及了几个参数，一个是加密签名字符串，一个是时间戳，一个是随机数，一个是应用接入 ID，我们一般的处理规则如下所示。

1）Web API 为各种应用接入，如 APP、Web、Winform 等接入端分配应用 AppID 以及通信密钥 AppSecret，双方各自存储。
2）接入端在请求 Web API 接口时需携带以下参数：signature、 timestamp、nonce、appid，签名是根据几个参数和加密秘钥生成。
3）Web API 收到接口调用请求时需先检查传递的签名是否合法，验证后才调用相关接口。

加密签名在服务端（Web API 端）的验证流程参考微信的接口的处理方式，处理逻辑如下所示。

1）检查 timestamp 与系统时间是否相差在合理时间内，如 10 分钟。
2）将 appSecret、timestamp、nonce 三个参数进行字典序排序
3）将三个参数字符串拼接成一个字符串进行 SHA1 加密

4）加密后的字符串可与 signature 对比，若匹配则标识该次请求来源于某应用端，请求是合法的。

C#端代码校验如下所示。

```
/// <summary>
/// 检查应用接入的数据完整性
/// </summary>
/// <param name="signature">加密签名内容</param>
/// <param name="timestamp">时间戳</param>
/// <param name="nonce">随机字符串</param>
/// <param name="appid">应用接入Id</param>
/// <returns></returns>
public CheckResult ValidateSignature(string signature, string timestamp, string nonce, string appid)
{
    CheckResult result = new CheckResult();
    result.errmsg = "数据完整性检查不通过";

    //根据Appid获取接入渠道的详细信息
    AppInfo channelInfo = BLLFactory<App>.Instance.FindByAppId(appid);
    if (channelInfo != null)
    {
        #region 校验签名参数的来源是否正确
        string[] ArrTmp = { channelInfo.AppSecret, timestamp, nonce };

        Array.Sort(ArrTmp);
        string tmpStr = string.Join("", ArrTmp);

        tmpStr = FormsAuthentication.HashPasswordForStoringInConfigFile(tmpStr, "SHA1");
        tmpStr = tmpStr.ToLower();

        if (tmpStr == signature && ValidateUtil.IsNumber(timestamp))
        {
            DateTime dtTime = timestamp.ToInt32().IntToDateTime();
            double minutes = DateTime.Now.Subtract(dtTime).TotalMinutes;
            if (minutes > timspanExpiredMinutes)
            {
                result.errmsg = "签名时间戳失效";
            }
            else
            {
                result.errmsg = "";
                result.success = true;
                result.channel = channelInfo.Channel;
            }
        }
        #endregion
    }
    return result;
}
```

一旦我们完成对安全签名进行成功认证，也就是我们对数据提交的来源和完整性进行了确认，就可以进行更多和安全性相关的操作了，如获取用户的访问令牌信息的操作如下所示。

第一步是验证用户的签名是否符合要求，符合要求后进行用户信息的比对，并生成用户访问令牌数据 JSON，返回给调用端即可。

```
/// <summary>
/// 系统认证等基础接口实现
/// </summary>
4 个引用
public class AuthApi : BaseApi, IAuthApi
{
    2 个引用
    public TokenResult GetAccessToken(string username, string password,
        string signature, string timestamp, string nonce, string appid)
    {
        TokenResult token = new TokenResult();
        try
        {
            //先校验接入应用的数据完整性
            CheckResult checkResult = ValidateSignature(signature, timestamp, nonce, appid);
            if (checkResult.success)
            {
                检查用户身份
            }
        }
        catch (Exception ex)
        {
            LogTextHelper.Error(ex);
        }
        return token;
    }
```

## 2.3. Web API 使用安全令牌的实现

通过上面的接口，我们获取到的用户访问令牌，以后和用户相关的信息调用，我们就可以通过这个令牌参数进行传递就可以了，这个令牌带有用户的一些基础信息，如用户 ID，过期时间等等，这个 Token 的设计思路来源于 JSON Web Token (JWT)，具体可以参考 http://self-issued.info/docs/draft-ietf-oauth-json-web-token.html，以及 GitHub 上的项目 https://github.com/jwt-dotnet/jwt。

由于 Web API 的调用，都是一种无状态方式的调用方式，我们通过 token 来传递我们的用户信息，这样我们只需要验证 Token 就可以了。JWT 的令牌生成逻辑如下所示

```
//先校验接入应用的数据完整性
CheckResult checkResult = ValidateSignature(signature, timestamp, nonce, appid);
if (checkResult.success)
{                                                          1、检查用户签名
    #region 检查用户身份
    //检验数据完整性后，从数据库检查用户身份
    CommonResult result = BLLFactory<Subscriber>.Instance.VerifyUser(username, password);
    if (result.Success)
    {
        //用户ID                                                2、验证用户身份
        string userId = result.Data1;
        TimeSpan t = (DateTime.UtcNow - new DateTime(1970, 1, 1));
        int times = (int)t.TotalSeconds;

        var payload = new Dictionary<string, object>
            {
                {"iss", userId},
                { "iat", times },
            };
                                                           3、生成JWT令牌
        //生成具体的Token和过期时间
        token.access_token = JsonWebToken.Encode(payload, sharedKey, JwtHashAlgorithm.HS256);
        token.expires_in = expiredDays * 24 * 3600;
    }
    #endregion
}
```

令牌生成后，我们需要在 Web API 调用处理前，对令牌进行校验，确保令牌是正确有效的。

检查的代码，就是把令牌生成的过程逆反过来，获取相应的信息，并且对令牌签发的时间进行有效性判断，一般可以约定一个失效时间，如 1 天或者 7 天，也不用设置太短。

一般来说，访问令牌不能永久有效，对于访问令牌的重新更新问题，可以设置一个规则，只允许最新的令牌使用，并把它存储在接口缓存里面进行对比，应用系统退出的时候，就把内存里面的 Token 移除就可以了。

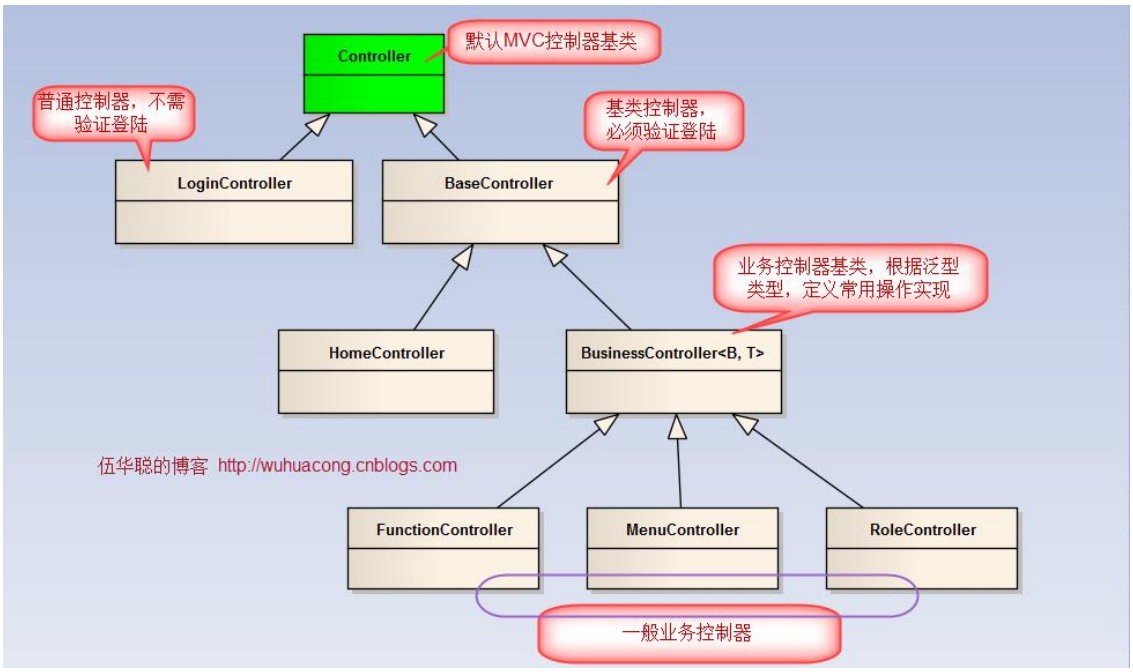
```
/// <summary>
/// 检查用户的Token有效性
/// </summary>
/// <param name="token"></param>
/// <returns></returns>
public CheckResult ValidateToken(string token)
{
    //返回的结果对象
    CheckResult result = new CheckResult();
    result.errmsg = "令牌检查不通过";

    if (!string.IsNullOrEmpty(token))
    {
        try
        {
            string decodedJwt = JsonWebToken.Decode(token, sharedKey);
            if (!string.IsNullOrEmpty(decodedJwt))
            {
                #region 检查令牌对象内容
                dynamic root = JObject.Parse(decodedJwt);
                string username = root.name;
                string userid = root.iss;
                int jwtcreated = (int)root.iat;

                //检查令牌的有效期，7天内有效
                TimeSpan t = (DateTime.UtcNow - new DateTime(1970, 1, 1));
                int timestamp = (int)t.TotalDays;
                if (timestamp - jwtcreated > expiredDays)
                {
                    throw new ArgumentException("用户令牌失效.");
                }

                //成功校验
                result.success = true;
                result.errmsg = "";
                result.userid = userid;
                #endregion
            }
        }
        catch (Exception ex)
        {
            LogTextHelper.Error(ex);
        }
    }
    return result;
}
```

## 2.4. Web API 设计分析

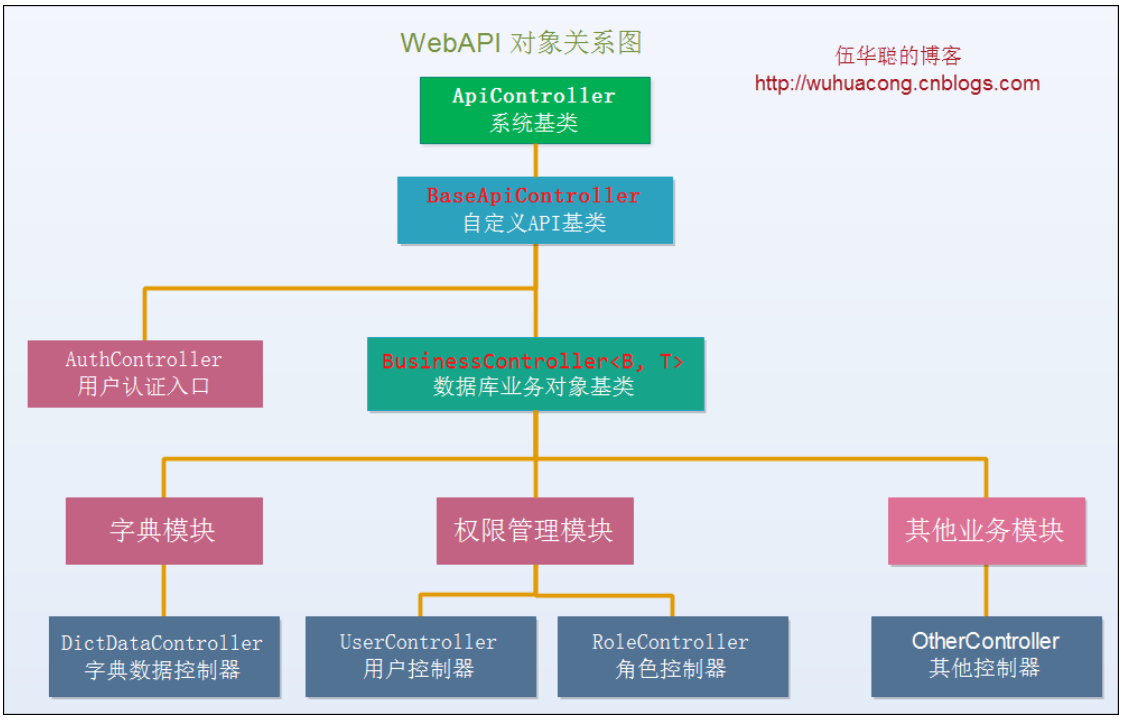
### 2.4.1. Web API 的控制器设计分析

上面介绍了一些 Web API 控制器的职能，一般情况下，我们设计一个架构，还需要考虑到基类对象之间的重用关系，尽可能把接口抽象到基类层面上去，减少子类的开发代码量，降低维护成本。

基于上面的目的，参考了我的 Web 开发框架对于 MVC 控制器的设计思路



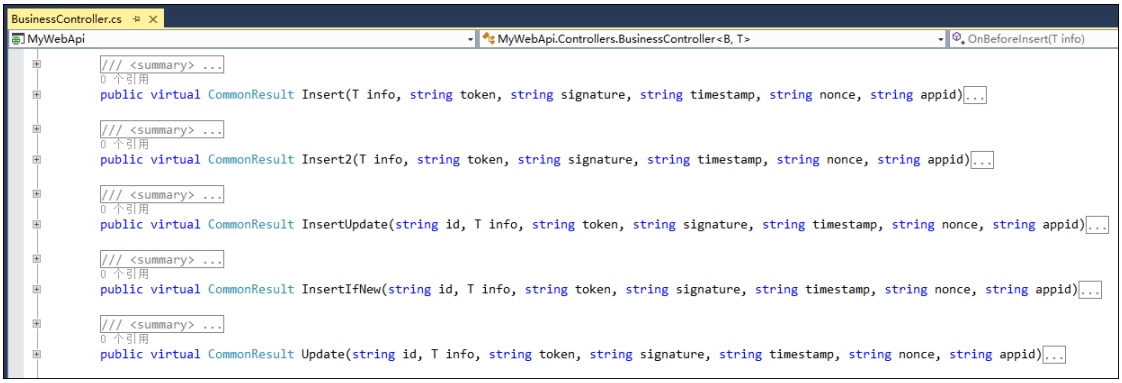

重新整理了 Web API 的控制器设计对象继承关系，如下所示：




我们关键的核心就是设计好 **BusinessController<B, T>**这个基类，里面设计了大量的通用接口，包括常规的增删改查、分页等处理接口，那么子类继承过来就可以直接拥有这些接口了，多方便啊。

```
namespace MyWebApi.Controllers
{
    /// <summary>
    /// 本控制器基类专门为访问数据业务对象而设的基类
    /// </summary>
    /// <typeparam name="B">业务对象类型</typeparam>
    /// <typeparam name="T">实体类类型</typeparam>
    34 个引用
    public class BusinessController<B, T> : BaseApiController
        where B : class
        where T : WHC.Framework.ControlUtil.BaseEntity, new()
    {
        构造函数及常用

        对象添加、修改、查询接口

        返回集合的接口

        基础接口

        其他接口
    }
}
```

部分接口说明如下所示。




上面的 BusinessController 类定义了常规的增删改查、分页等常见的接口以及实现，我们在开发中，往往也需要定义自己的业务接口，这些是无法自动进行生成的，我们可以参考基类代码进行适当的处理。

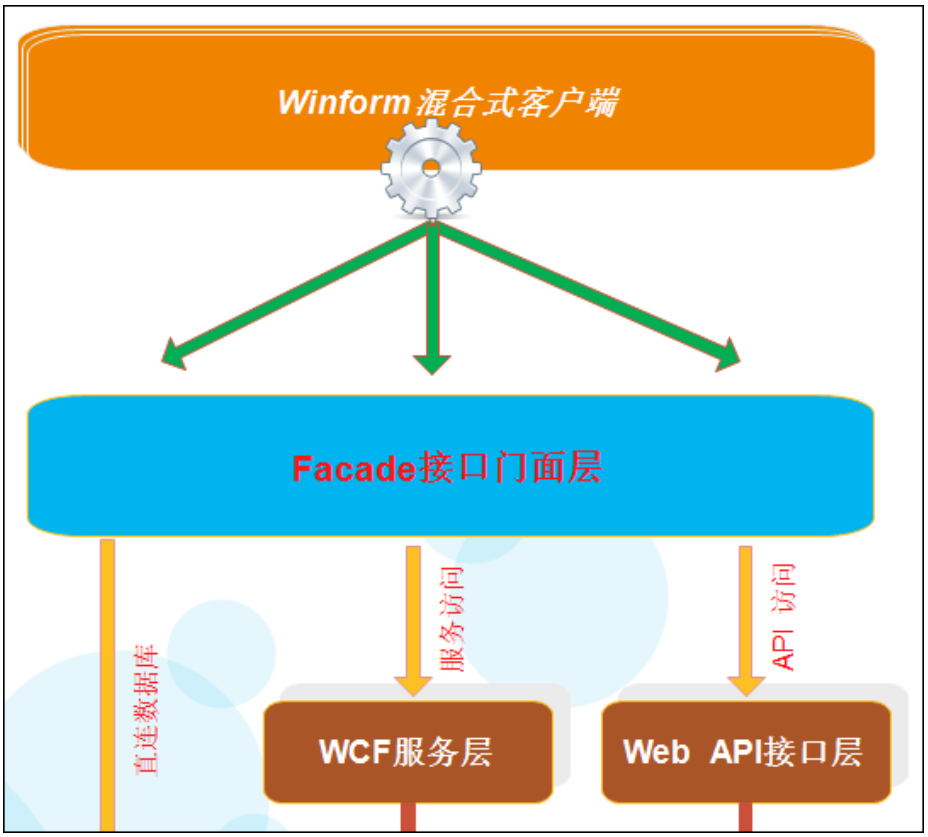
```
15
16  namespace MyWebApi.Controllers
17  {
        0 个引用
18      public class ItemDetailController : BusinessController<ItemDetail, ItemDetailInfo>
19      {
20          写入数据前修改部分属性
31
32
33          #region IItemDetailService 成员
34
35          [HttpGet]
            0 个引用
36          public List<ItemDetailInfo> FindByBigType(string bigType, string token)
37          {
38              //令牌检查,不通过则抛出异常
39              CheckResult checkResult = CheckToken(token);
40
41              return BLLFactory<ItemDetail>.Instance.FindByBigType(bigType);
42          }
43
44          [HttpGet]
            0 个引用
45          public List<ItemDetailInfo> FindByItemType(string itemType, string wareHouse, string token)
46          {
47              //令牌检查,不通过则抛出异常
48              CheckResult checkResult = CheckToken(token);
49
50              return BLLFactory<ItemDetail>.Instance.FindByItemType(itemType, wareHouse);
51          }
```

## 2.4.2. Web API 客户端（混合式 Winform 框架模块）的调用

前面介绍了 Web API 服务端平台的架构设计思路，通过上面的整合，我们减轻了开发重复功能的增删改查等基础功能的控制器代码，把这些接口抽象到接口里面即可实现。

但是我们具体应该如何遵循统一接口层 Facade 层的约定，然后统一调用 WebAPI 层的接口，做到悄无声息的从不同的数据源里面获取数据，展示在客户端里面呢。

上面我们分析到，整个混合式 Winform 框架模块里面，设计方面考虑了数据的获取方面：包含了直接从数据库获取，从 WCF 服务获取，以及 Web API 层的数据获取三部分内容，当然还可以有更多的数据接入模式（如 WebService 等），设计效果如下所示。

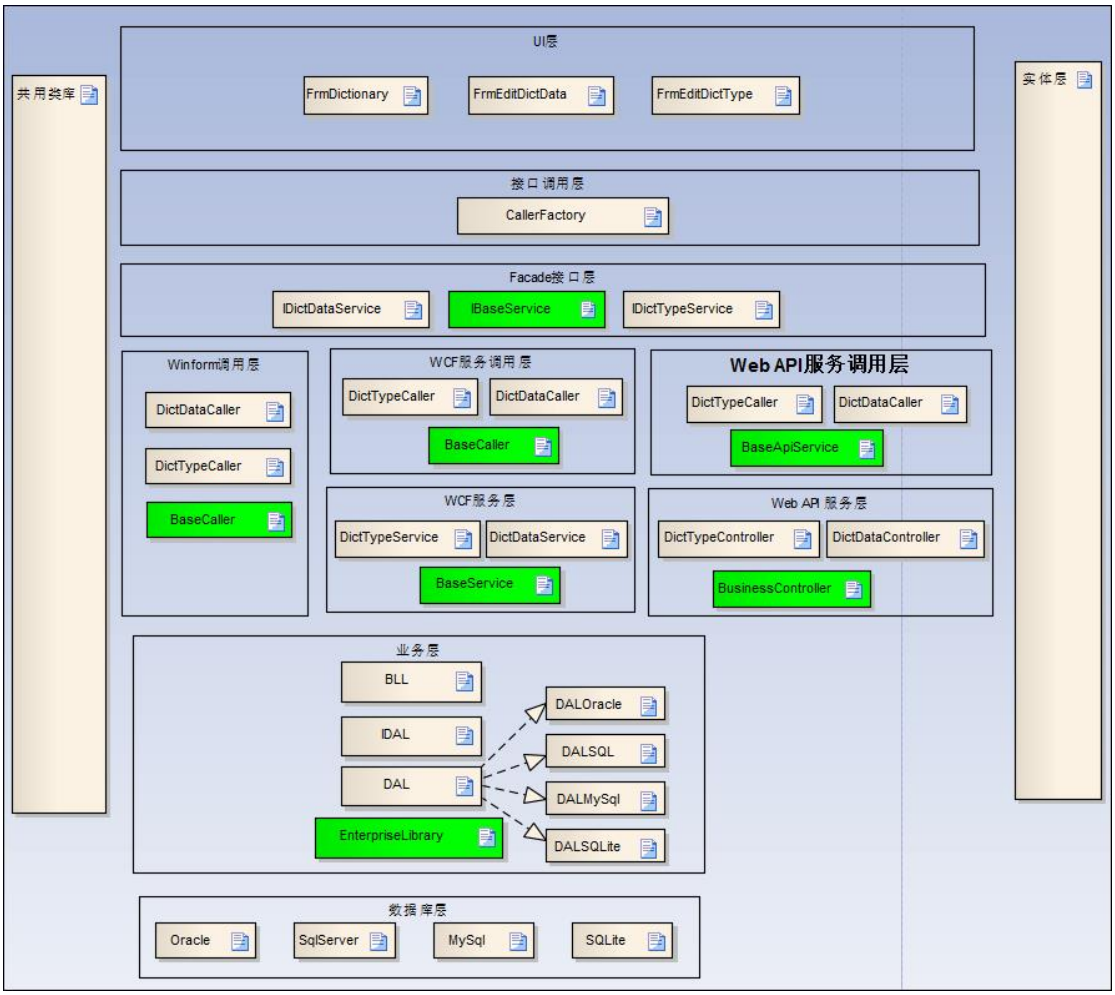



所有的数据接入，我们在 Facade 层都统一到接口里面，客户端的调用也统一到了 CallerFactory<T>这个泛型工厂里面，我们根据配置的不同，从不同的模块里面加载，从而实现不同数据源的动态获取了。

如下逻辑就是 CallerFactory<T>泛型工厂类的加载逻辑，如下所示：

```
if (isWCF)
{
    callerNamespace = rootNamespace + ".ServiceCaller";
}
else if (isAPI)
{
    callerNamespace = rootNamespace + ".WebApiCaller";
}
else
{
    callerNamespace = rootNamespace + ".WinformCaller";
}

#endregion

string assemblyName = typeof(T).Assembly.GetName().Name;//约定Facade接口和实现是同一个程序集
Assembly tempAssembly = Assembly.Load(assemblyName);
```

为了实现简化客户端调用的封装，我们一般也把常规的通用操作封装一下，如下是混合框架里面的设计思路，里面的封装都是通过***Caller 的类来进行数据的访问的，这些类统一实现一定关系的集成封装。

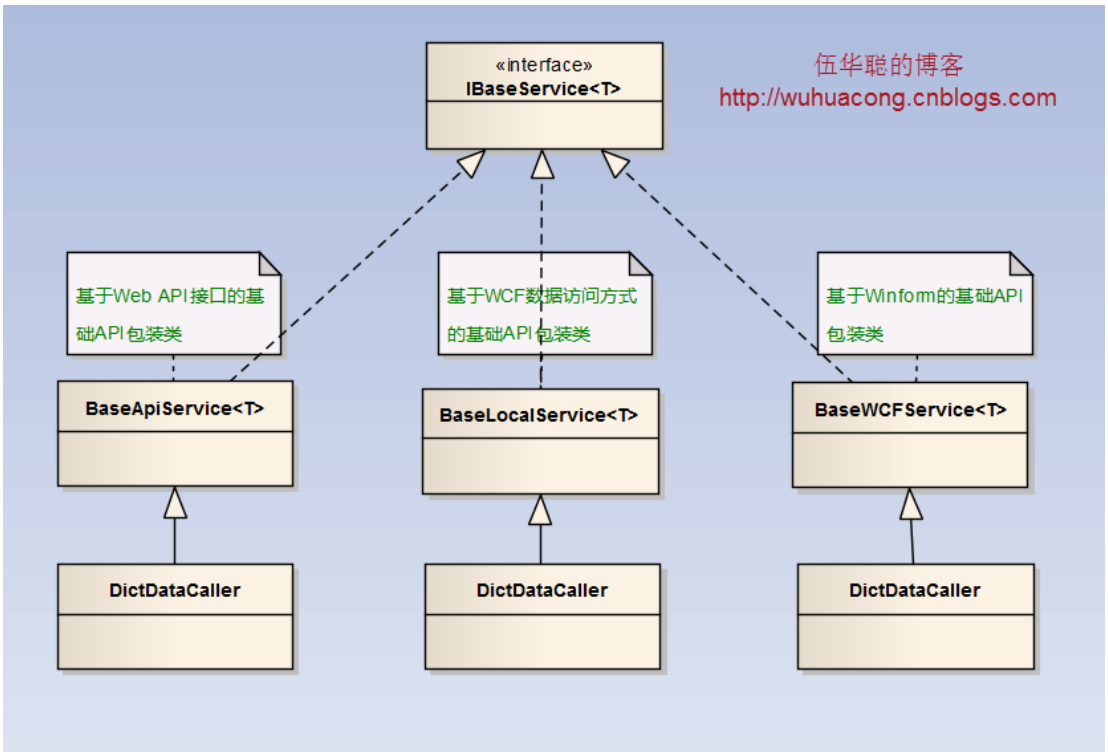



在 Caller 项目里面，就是为客户端提供一致的调用方式，不管是 WCF 方式、直接访问数据库方式，还是目前介绍的 Web API 方式，我们通过 Caller 项目类的封装处理，就能屏蔽他们之间的差异，使得我们的界面调用实现统一性，并且也方便在各个方式的调用中进行切换。

这个项目里面包含了门面层的接口定义层 Facade 接口层，由于我们考虑了 WCF 的接入方式，因此需要增加一些 WCF 的接口标识，如果仅仅是考虑 Web API 的接入，则可以不用[ServiceContract]和[OperationContract]的声明内容。

```
IMemberService.cs
WHC.CloudMember.Caller                    WHC.CloudMember.Facade.IMer
using ...

namespace WHC.CloudMember.Facade
{
    [ServiceContract]
    [ServiceKnownType(typeof(CardStatus))]
    49 个引用
    public interface IMemberService : IBaseService<MemberInfo>
    {
        /// <summary>
        /// 根据公司ID获取公司的相关会员
        /// </summary>
        /// <param name="corpId">公司ID</param>
        /// <returns></returns>
        [OperationContract]
        2 个引用
        List<MemberInfo> FindByCorp(string corpId);
```

为了简化说明调用接口的处理，这里把上面的关系进行了简化，加入了 Web API 的调用封装类的处理后，几种访问模式下的调用端封装继承关系，如下设计图所示。



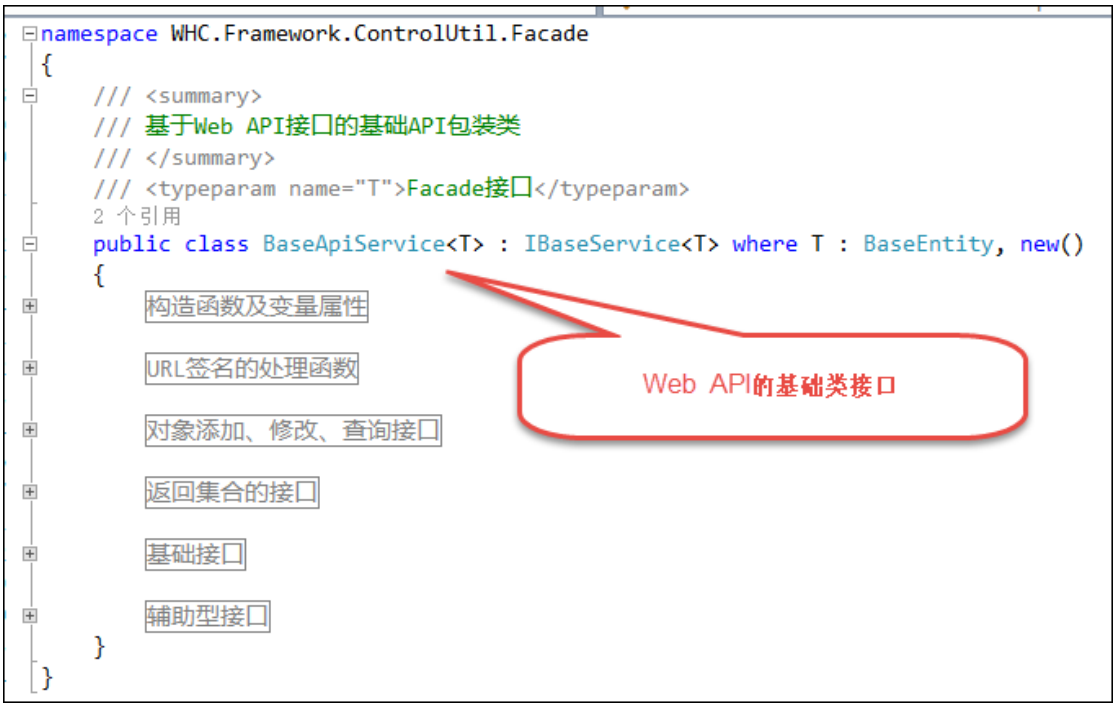

最底层的几个 DictDataCaller 分别是不同访问方式下的接口调用封装类，对于 Web API 来说，它的访问代码就是如下所示。

```
public override bool Delete(string key)
{
    var action = "Delete";
    string url = GetPostUrlWithToken(action) +string.Format("&id={0}", key);

    CommonResult result = JsonHelper<CommonResult>.ConvertJson(url);
    return result.Success;
}

public List<DictDataInfo> FindByTypeID(string dictTypeId)
{
    var action = "FindByTypeID";
    string url = GetTokenUrl(action) + string.Format("&dictTypeId={0}", dictTypeId);

    List<DictDataInfo> result = JsonHelper<List<DictDataInfo>>.ConvertJson(url);
    return result;
}
```

第一个 Delete 函数是基类提供的，这里进行了重写，一般情况下，不需要处理就具备增删改分页等基础接口的调用封装了。

在混合式框架里面，我们已经提供了功能强大的 Web API 调用封装的基类，这些基类默认已经包含了对基础增删改查以及分页等 Web API 接口的封装。因此我们在做 Web API 项目开发的时候，只需要定义和实现我们所需要的额外处理接口即可。

在开发框架里面，我们一般通过继承基类的方式，可以直接使用基类定义的增删改接口的封装处理，这样可以极大简化我们所需的代码，同时降低代码编写的复杂度，常规的功能我们也不用重复编写了。



如我们仓库管理模块里面，备件信息模块的 Web API 接口调用封装类的部分代码如下所示，这些接口的封装就是我们基于自定义 Web API 接口的封装处理代码。

```
namespace WHC.WareHouseMis.WebApiCaller
{
    /// <summary>
    /// ItemDetail, 基于API服务的Facade接口实现类
    /// </summary>
    1 个引用
    public class ItemDetailCaller : BaseApiService<ItemDetailInfo>, IItemDetailService
    {
        0 个引用
        public ItemDetailCaller()  : base()
        {
            this.ConfigurationPath = ApiConfig.ConfigFileName; //Web API配置文件
            this.configurationName = ApiConfig.ItemDetail;
        }

        #region IItemDetailService 成员

        4 个引用
        public List<ItemDetailInfo> FindByBigType(string bigType)
        {
            var action = "FindByBigType";
            string url = GetTokenUrl(action) + string.Format("&bigType={0}", bigType);

            List<ItemDetailInfo> result = JsonHelper<List<ItemDetailInfo>>.ConvertJson(url);
            return result;
        }

        4 个引用
        public List<ItemDetailInfo> FindByItemType(string itemType, string wareHouse)
        {
            var action = "FindByItemType";
            string url = GetTokenUrl(action) + string.Format("&itemType={0}&wareHouse={1}", itemType, wareHouse);

            List<ItemDetailInfo> result = JsonHelper<List<ItemDetailInfo>>.ConvertJson(url);
            return result;
        }
```

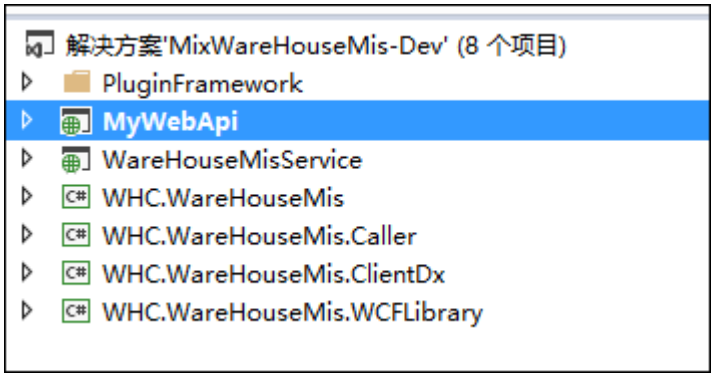
我们在开发混合框架模块的时候，一般都保留直接调用数据库的包装类项目代码【WinformCaller】，这样我们可以在开发的时候，先着重调试好直接访问数据库的模式，然后在调试基于 Web API 的方式，这样可以避免很多低层次的错误，同时也可以快速调试跟踪代码，为我们后面的 Web API 接入排除更多的错误，提高开发效率和代码质量。

由于所有的实现类都实现继承了统一的 Facade 层的接口，那么统一调用也就是自然而然的事情了。所以在 Winform 界面里面，所有的调用都是使用 CallerFactory<T>进行了统一的处理，数据访问（不管 WCF，还是 Web API 等方式）的不同不影响接口的处理， 三种方式的数据调用，统一都是下面的代码进行处理。

```
DictDataInfo info = CallerFactory<IDictDataService>.Instance.FindByID(ID);

if (info != null)
{
    SetInfo(info);

    try
    {
        bool succeed = CallerFactory<IDictDataService>.Instance.Update(info, info.ID.ToString());
        return succeed;
    }
    catch (Exception ex)
    {
        LogTextHelper.Error(ex);
        MessageDxUtil.ShowError(ex.Message);
    }
}
```

## 2.5. Winform 界面调用 WebAPI 的过程

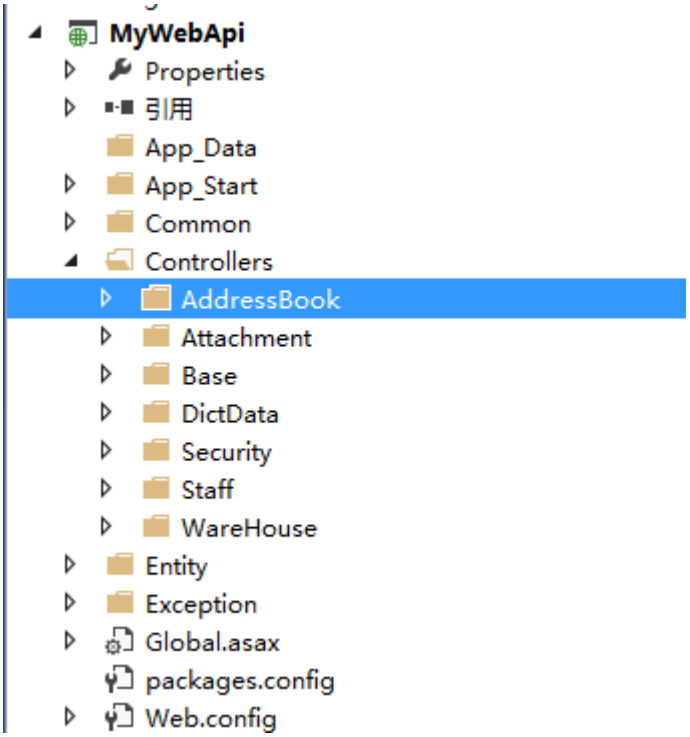
我以实际项目的结构，也就是以 Web API 方式改造的会员管理系统项目结构进行展示，以实践的项目开发过程来阐述整个开发过程，希望读者对这些有更好的了解和更直观的认识。整个混合框架系统项目结构如下所示。



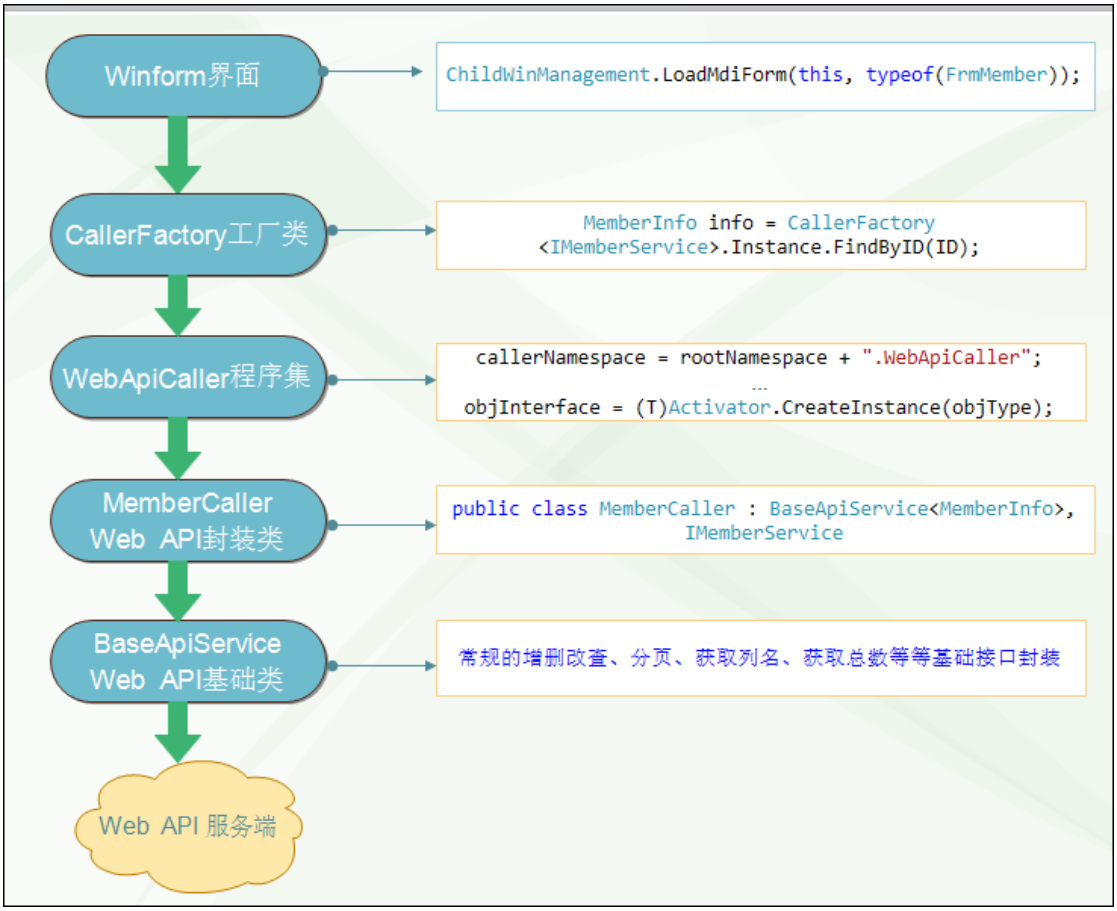

项目结构的总体说明如下所示。

| 项目名称 | 项目说明 |
| --- | --- |
| WHC.WareHouseMis.ClientDx | 基于 Web API 的 WInform 界面项目 |
| WHC.WareHouseMis.Caller | 接口调用封装类模块 |
| MyWebApi | Web API 的服务发布项目 |
| WHC. WareHouseMis | 业务逻辑模块的项目，包含数据访问层、接口、实体类等 |

在混合框架里面，Web API 服务项目 MyWebApi，它的详细内容包括服务接口的定义、异常的处理、令牌权限认证及识别、Web API 客户端的调用的知识点。




Winform 调用 Web API 的过程，这个过程可以通过下面这个图示进行讲解。



**1）**首先我们在界面一般是通过定义一个 Winform 窗体，并在其中放置相应的控件来承载信息的，这个和普通的 Winform 是一样的，例如我们定义一个窗体对象 FrmMember，以及 FrmEditMember。

**2）**在主体界面里面，我们需要调用 FrmMember 这个窗体，可以通过对话框的方式，或者是多文档的方式进行调用显示。

```
FrmEditMember dlg = new FrmEditMember();
dlg.ShowDialog();
```

或者多文档界面展示

```
ChildWinManagement.LoadMdiForm(this, typeof(FrmMember));
```

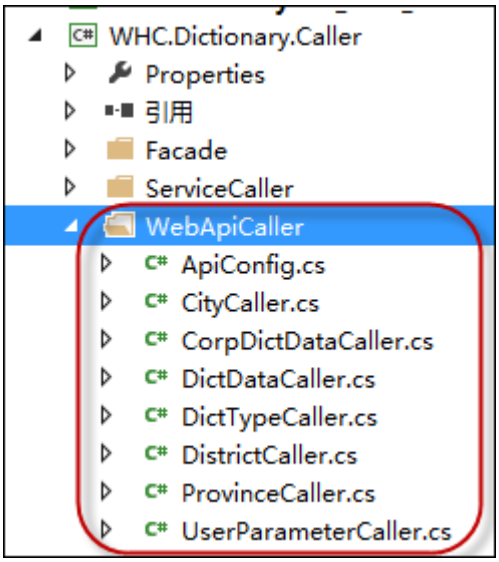
**3）**在界面里面，我们需要调用接口对象（Web API 的客户端包装类）进行获取对应的信息，这里使用到了接口工厂 CallerFactory<T>这种方式进行调用。

```
MemberInfo info =
CallerFactory<IMemberService>.Instance.FindByID(ID);
```

**4）**上面这个工厂类 CallerFactory<T>是负责获取到对应的接口实现类并创建对象，方便我们进行调用处理。它的逻辑主要是通过 IMemberService 接口所在的程序集（例如 WHC.CloudMember.WebApiCaller），然后获取对应接口的实现类，并构建一个这样的接口实例出来使用的。

```
/// <summary>
/// 混合式框架中针对不同调用方式的工厂类（Web API、WCF或者Win方式调用）
/// </summary>
/// <typeparam name="T">接口类型</typeparam>
0 个引用
public class CallerFactory<T>
{
    private static Hashtable objCache = new Hashtable();
    private static object syncRoot = new Object();
    private static string callerNamespace = null;//Facade接口实现类的命名空间

    /// <summary>
    /// 创建或者从缓存中获取对应接口的实例
    /// </summary>
    0 个引用
    public static T Instance...

    /// <summary>
    /// 根据配置参数CallerType的值，创建不同的调用方式实现接口
    /// </summary>
    /// <returns></returns>
    1 个引用
    private static T CreateObject()...

    /// <summary>
    /// 创建程序集对象
    /// </summary>
    1 个引用
    private static Assembly LoadAssembly()...
```

例如字典模块，混合框架里面，他们的各个模块的实现类是放在程序集里面的，我们的目标就是根据接口的名称，从对应的部分获取相应的 Web API 接口调用包装类进行使用。




**5**）我们构建的 Web API 接口调用包装类（WebApiCaller 里面的内容），为了实现更加方便的调用，我们为它进行了一定的封装，使它在基于泛型的基础上具有基础增删改查、分页等功能的调用处理。

```
namespace WHC.CloudMember.WebApiCaller
{
    /// <summary>
    /// 基于WebAPI方式的Facade接口实现类
    /// </summary>
    1 个引用
    public class MemberCaller : BaseApiService<MemberInfo>, IMemberService
    {
        0 个引用
```

从这个类的定义里面，我们可以看到 Web API 的调用包装类 MemberCaller 是继承自 BaseApiService<MemberInfo>这样的泛型基类的。这个 BaseApiService<MemberInfo>就具有对特定对象的增删改查、分页等基础调用功能了。

例如在基类 BaseApiService 里面的查找对应对象的接口代码如下所示

```
        /// <summary>
        /// 查询数据库,检查是否存在指定 ID 的对象(用于字符型主键)
        /// </summary>
        /// <param name="key">对象的 ID 值</param>
        /// <returns>存在则返回指定的对象,否则返回 Null</returns>
        public virtual T FindByID(string key)
        {
            var action = "FindByID";
            string url = GetTokenUrl(action) + string.Format("&id={0}", key);
```

```
            return JsonHelper<T>.ConvertJson(url);
        }
```

这里面的逻辑就是构建一个带有 token（用户身份标识）的连接字符串和参数字符串，从而获取 HTML 内容后把它转换为具体对象的处理了。

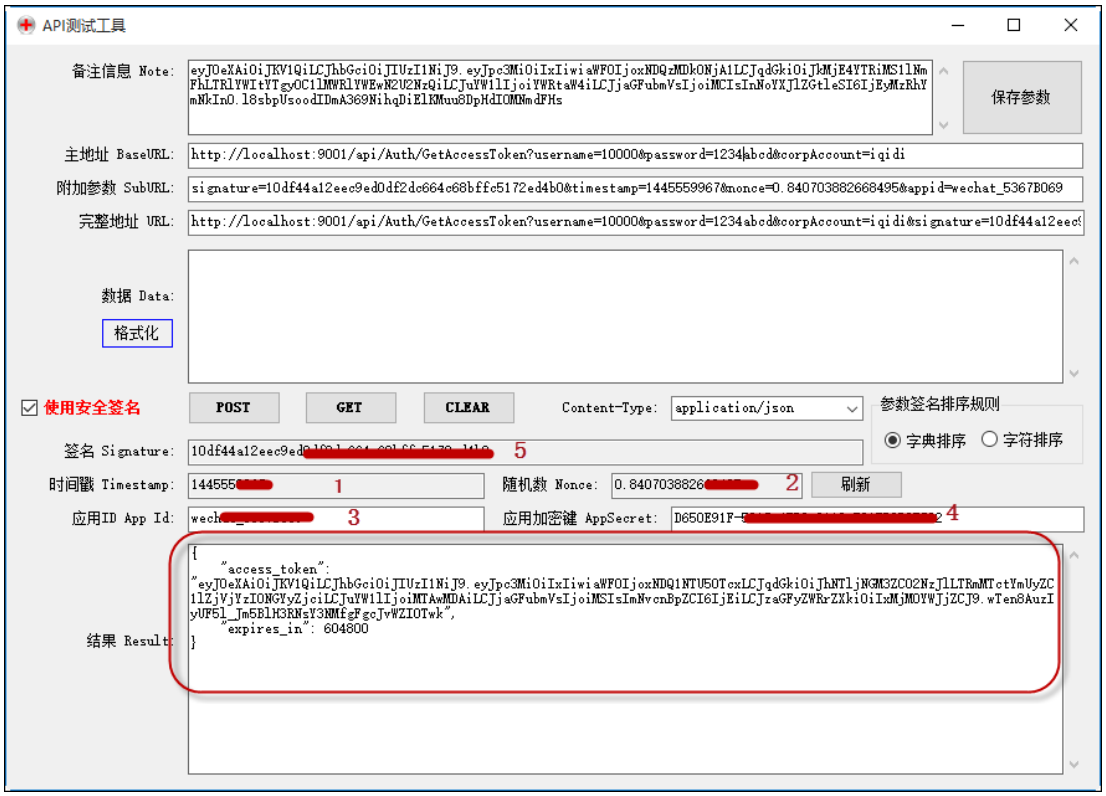
其中转换的代码就是利用了 Newtonsoft.Json 的对象的转换，具体代码如下所示。

```
        /// <summary>
        /// 转换 Json 字符串到具体的对象
        /// </summary>
        /// <param name="url">返回 Json 数据的链接地址</param>
        /// <returns></returns>
        public static T ConvertJson(string url)
        {
            HttpHelper helper = new HttpHelper();
            helper.ContentType = "application/json";

            string content = helper.GetHtml(url);
            VerifyErrorCode(content);

            T result = JsonConvert.DeserializeObject<T>(content);
            return result;
        }
```

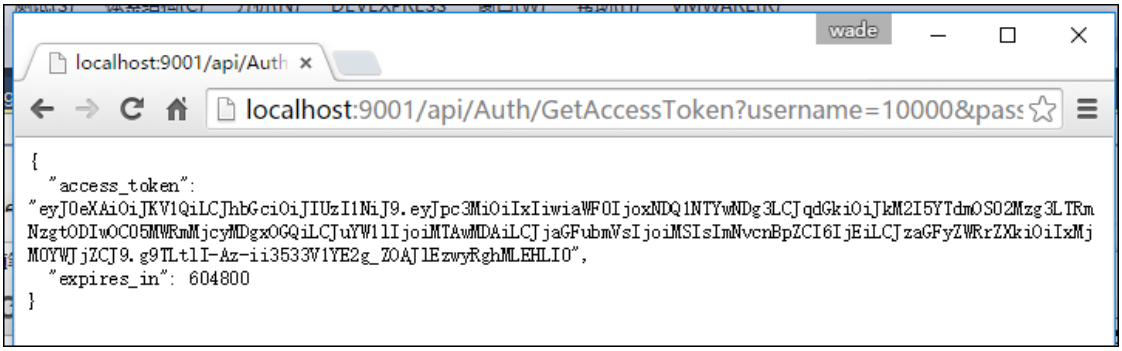
**6）用户的访问令牌（Token 信息）**

当然我们调用这个接口前，我们需要获取到对应的 Token（用户令牌）然后才能进行 API 的调用了。这个 Token 的机制采用了 JWT 的令牌生成方式，具有很好的通用性。

例如我使用自己的 Web API 调试工具，获取到对应的 token 方式如下所示。下面的 1-5 的标识就是获取 token 所需要的签名数据，当然连接还带有几个账号认证所需要的信息了，如账号密码、所在公司等信息。



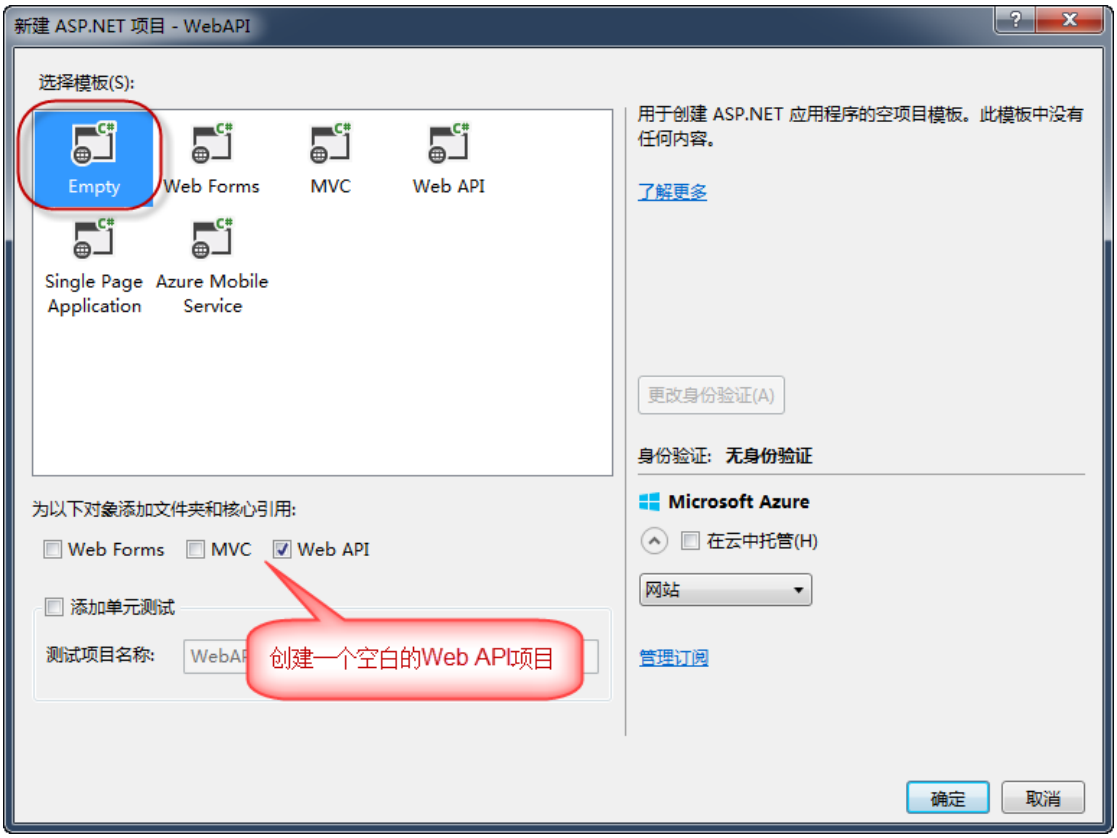

当然我们也可以使用浏览器进行测试获取 Token 的信息，只是没有那么方便而已。




## 2.6. ASP.NET Web API 的开发

### 2.6.1. 常规的 Web API 项目开发

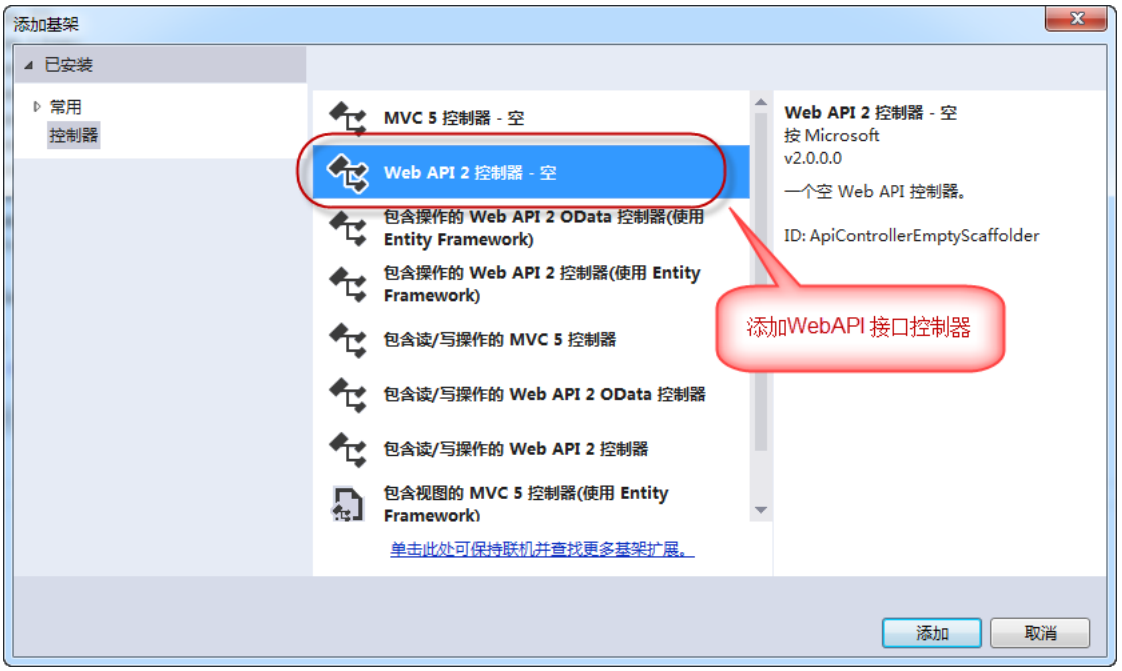
上面我们定义了一般的 Web API 接口，以及实现相应的业务实现，如果我们需要创建 Web API 层，那么还需要构建一个 Web API 项目的，常规的 Web API 项目开发流程如下所示。




创建好相应的项目后，可以为项目添加一个 Web API 基类，方便控制共同的接口。

```
5 个引用
public class BaseApiController : ApiController
{
    辅助方法
}
```

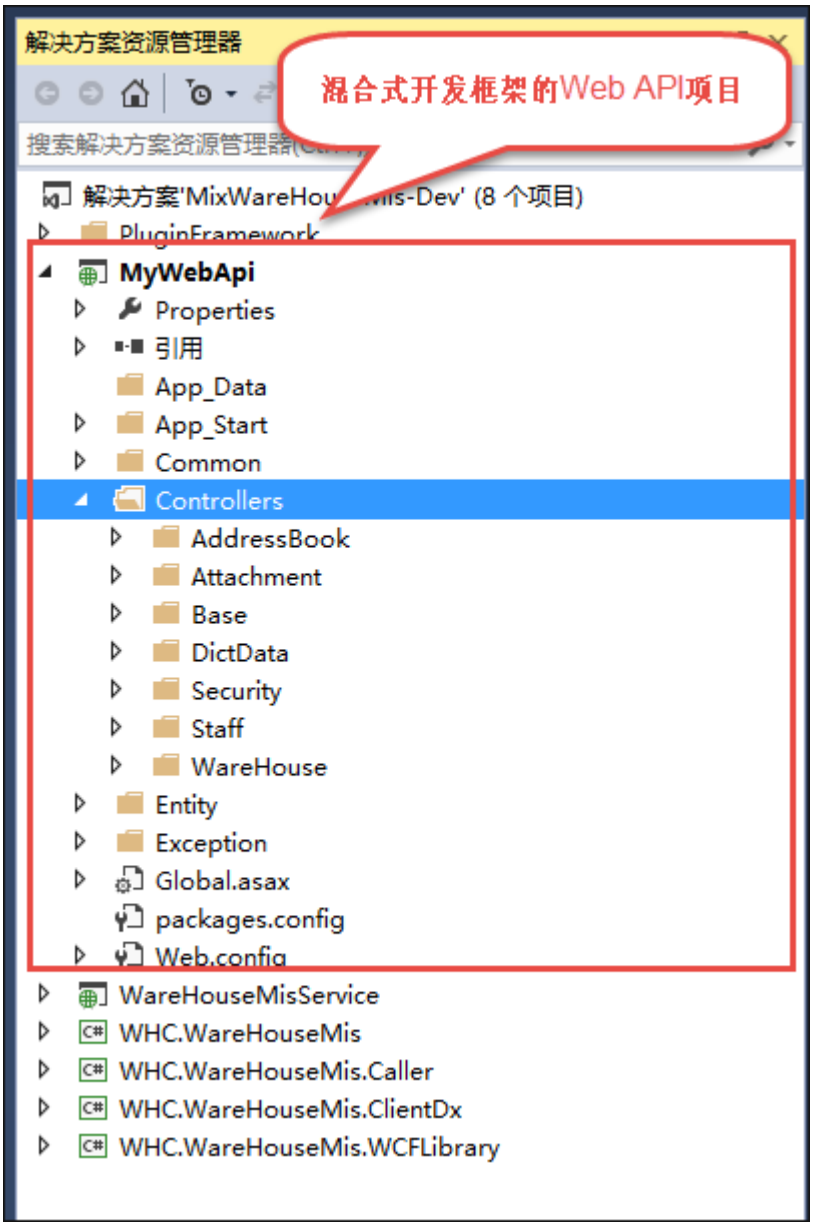
然后我们就可以在 Controller 目录上创建更多的应用 API 控制器了。




一般情况下，我们创建的空白 Web API 控制，里面的继承关系，以及所有的接口代码，都需要我们进行代码编写的，因此这种方式效率会比较低。

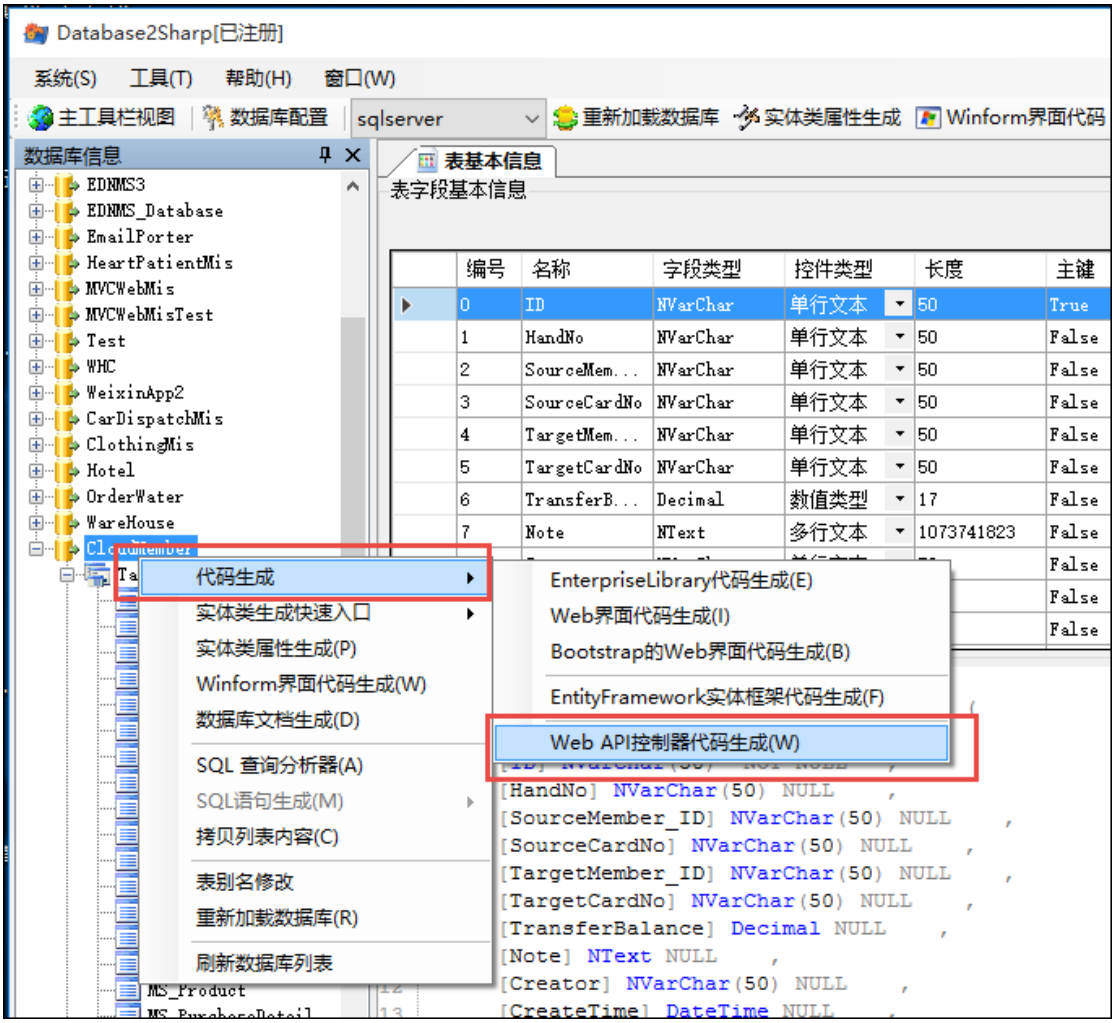
## 2.6.2. 基于混合式框架及代码生成工具的 Web API 项目开发

基于混合式开发框架的基础上做增量的开发工作，是一件非常方便的事情。混合式开发框架本身带有集成 Web API 接口的实现，因此它本身也带有一个 Web API 的项目。如果我们需要做增量开发，只需要在这个基础上添加我们我们所需要的控制器类文件即可。如下面是混合式的 Web API 项目。

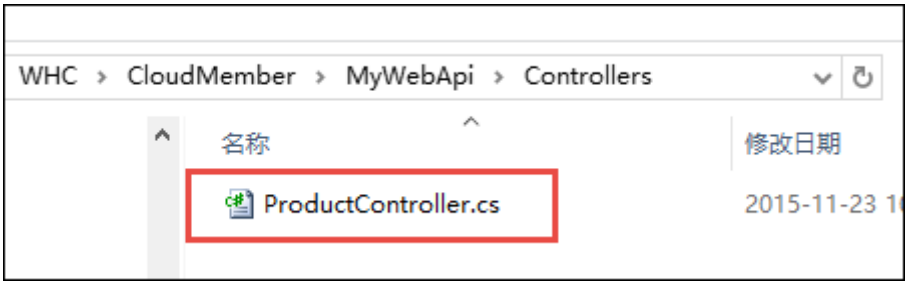



当然，这个 Web API 的项目，我们也是做了很好的提炼和整合的，因此这个 Web API 项目中关键的控制器类代码，我们也是可以使用功能强大的代码生成工具进行开发的。

在代码生成工具里面，展开左边的数据库树列表后，在右键菜单中选择【代码生成】【Web API 控制器代码生成】即可生成所需要表的 Web API 控制器类文件了。




生成文件或，代码工具自动打开所在目录，如下所示。




生成的代码自动创建好对应的继承关系、命名空间的引用关系，并且代码为了方便参考使用，自动生成了一些代表性接口的代码案例，以帮助我们快速创建各种类型的接口。

ProductController.cs

(杂项文件) MyWebApi.Controllers.ProductController

```
1  using ...
17
18 namespace MyWebApi.Controllers
19 {
20     public class ProductController : BusinessController<Product, ProductInfo>
21     {
22         写入数据前修改部分属性
33
34         #region 示例代码
35
36         /*
37         /// <summary>
38         /// 获取对应记录的名称
39         /// </summary>
40         /// <param name="id">记录ID</param>
41         /// <param name="token">访问令牌</param>
42         /// <returns></returns>
43         [HttpGet]
44         public string GetName(string id, string token)
45         {
46             //令牌检查,不通过则抛出异常
47             CheckResult checkResult = CheckToken(token);
48             return BLLFactory<Product>.Instance.GetName(id);
49         }
50
51         /// <summary>
52         /// 给指定角色添加用户
53         /// </summary>
54         /// <param name="param">包含多个属性的对象</param>
55         /// <param name="token">访问令牌</param>
56         [HttpPost]
57         public CommonResult AddUser(JObject param, string token)
58         {
59             //令牌检查,不通过则抛出异常
60             CheckResult checkResult = CheckToken(token);
61             dynamic obj = param;
62             if (obj != null)
63             {
64                 string userID = obj.userID;
65                 string roleID = obj.roleID;
66
67                 bool result = BLLFactory<Role>.Instance.AddUser(userID, roleID);
68                 return new CommonResult(result);
```

上面例子代码包含了常规的 HTTPGet 调用获取数据接口，以及基于 HTTPPost 方式的数据提交接口，这里面的提交采用了动态的 JSON 参数，非常方便使用。

另外，代码还带有一些需要安全签名参数的数据处理，以及数据分页的接口案例，如下代码所示。从中可见，我们是希望客户能够尽可能的快速、高效的处理好各个接口、项目之间的关系，尽可能为开发人员考虑好各种关系，以及提供方便高效的项目开发。

```
/// <summary>
/// 验证用户身份有效性
/// </summary>
/// <param name="param">包含多个属性的对象</param>
/// <returns></returns>
[HttpPost]
public CommonResult VerifyUser(JObject param, string signature, string timestamp, string nonce, string appid)
{
    //如果用户签名检查不通过，则抛出MyApiException异常。
    base.CheckSignature(signature, timestamp, nonce, appid);

    dynamic obj = param;
    if (obj != null)
    {
        string account = obj.account;
        string password = obj.password;
        string corpAccount = obj.corpAccount;
        string ip = obj.ip;
        string macAddr = obj.macAddr;

        return BLLFactory<User>.Instance.VerifyUser(account, password, corpAccount, ip, macAddr);
    }
    else
    {
        throw new MyApiException("传递参数错误");
    }
}

/// <summary>
/// 根据条件查询数据库,并返回对象集合(用于分页数据显示)
/// </summary>
/// <returns>指定对象的集合</returns>
[HttpPost]
public virtual PagedList<ProductInfo> FindWithPager(string condition, PagerInfo pagerInfo, string token)
{
    //令牌检查,不通过则抛出异常
    CheckResult checkResult = CheckToken(token);
    List<ProductInfo> list = BLLFactory<Product>.Instance.FindWithPager(condition, pagerInfo);

    //构造成Json的格式传递
    var result = new PagedList<ProductInfo>() { total_count = pagerInfo.RecordCount, list = list };
    return result;
}
```

在混合框架的开发过程中，Web API 开发只是其中的一环，Web API 接口操作相对于 WCF 来说，具有比较轻型、高效的特点，所以分布式推荐采用 Web API 的数据获取方式，如果重点采用 Web API 的数据接口获取数据，那么我们开发过程，就可以针对性的对 Web API 进行相关的开发工作了。

**一般为了方便调试，在混合式框架的开发过程中，我们一般还需要保留传统直接访问数据库的方式**。这样方便我们快速进行调试工作，检查开发过程中接口的错误问题，因为直接访问的方式，在混合框架里面实现统一的 Façade 门面层接口，只是对 BLL 层进行一层很薄的封装，因此实现起来最为方便。所以开发的时候，我们可以先实现直接访问数据库的方式，同时逐步实现 Web API 或者 WCF 获取数据的方式。

## 2.7. Web API 接口的测试

接下来我们要做的就是需要增加业务接口，以便进行具体的测试了，建议使用 Winform 项目，对每个接口进行一个测试，或者也可以考虑使用单元测试的方式，看个人喜好吧。

例如我们如果要测试用户登陆的接口的话，我们的测试代码如下所示。

```
/// <summary>
/// 生成签名字符串
/// </summary>
/// <param name="appSecret">接入秘钥</param>
/// <param name="timestamp">时间戳</param>
/// <param name="nonce">随机数</param>
private string SignatureString(string appSecret, string timestamp, string nonce)
{
    string[] ArrTmp = { appSecret, timestamp, nonce };

    Array.Sort(ArrTmp);
    string tmpStr = string.Join("", ArrTmp);

    tmpStr = FormsAuthentication.HashPasswordForStoringInConfigFile(tmpStr, "SHA1");
    return tmpStr.ToLower();
}

private TokenResult GetTokenResult()
{
    string timestamp = DateTime.Now.DateTimeToInt().ToString();
    string nonce = new Random().NextDouble().ToString();
    string signature = SignatureString(appSecret, timestamp, nonce);

    string appended = string.Format("&signature={0}&timestamp={1}&nonce={2}&appid={3}", signature, timestamp, nonce, appId);
    string queryUrl = url + "Auth/GetAccessToken?username=test&password=123456" + appended;

    HttpHelper helper = new HttpHelper();
    string token = helper.GetHtml(queryUrl);
    Console.WriteLine(token);
    TokenResult tokenResult = JsonConvert.DeserializeObject<TokenResult>(token);
    return tokenResult;
}
```

如果我们已经获得了令牌，我们根据令牌传递参数给连接，并获取其他数据的测试处理代码如下所示。

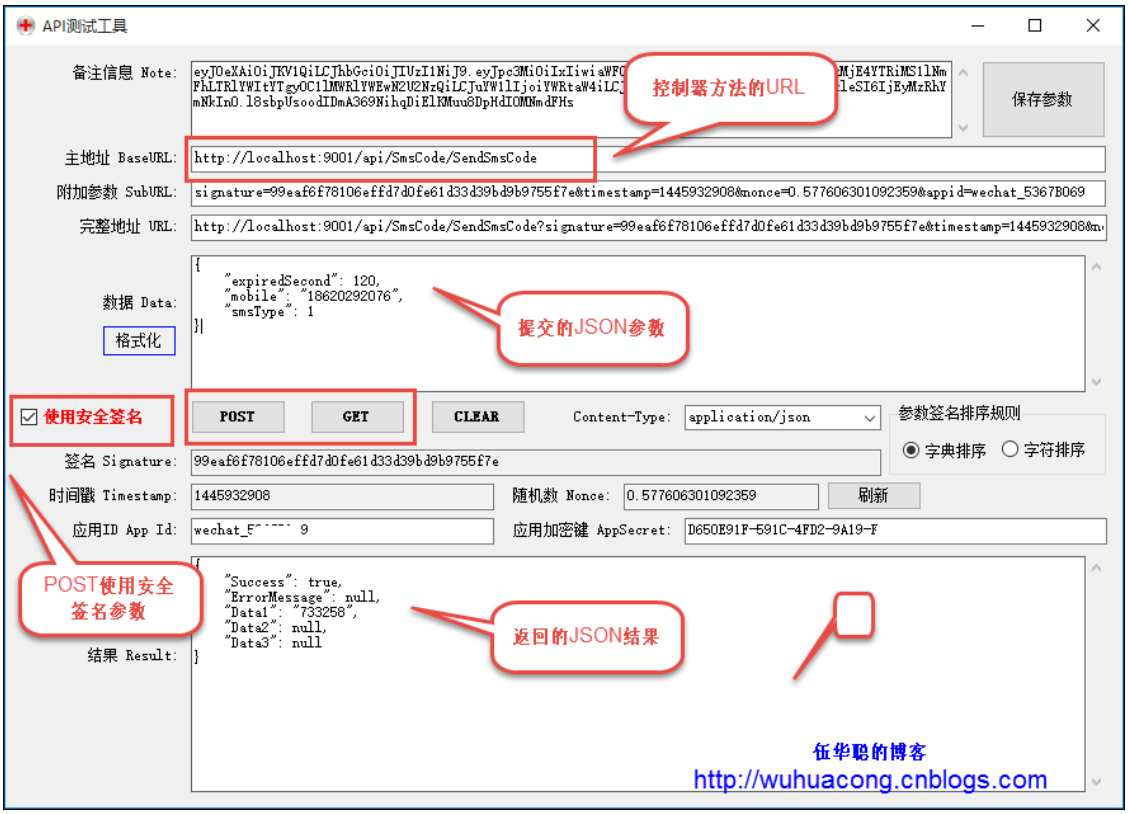
```
//获取访问令牌
TokenResult tokenResult = GetTokenResult();

string queryUrl = url + "/Contact/get?token=" + tokenResult.access_token;
HttpHelper helper = new HttpHelper();
string result = helper.GetHtml(queryUrl);
Console.WriteLine(result);
```

如果需要 POST 数据的话，那么调用代码如下所示。

```
//使用POST方式
var data = new
{
    name = "张三",
    certno = "123456789",
};
var postData = data.ToJson();

queryUrl = url + "/Contact/Add?token=" + tokenResult.access_token;
helper = new HttpHelper();
helper.ContentType = "application/json";
result = helper.GetHtml(queryUrl, postData, true);
Console.WriteLine(result);
```

Web API 后台，会自动把 POST 的 JSON 数据转换为对应的对象的。如果是 GET 方式，我们可能可以直接通过浏览器进行调试，如果是 POST 方式，我们需要使用一些协助工具，如 Fiddler 等处理工具，但是最好的方式是自己根据需要弄一个测试工具，方便测试。

以下就是我为了自己 Web API 接口开发的需要，专门弄的一个调试工具，可以自动组装相关的参数，包括使用安全签名的参数，还可以把所有参数数据进行存储。



# 2.8. Web API 应用支持 HTTPS

在前面介绍的 Web API 内容里面，介绍了 Web API 的架构设计、项目开发方面的内容，其中提出了现在流行的 Web API 优先的路线，这种也是我们开发多应用（APP、微信、微网站、商城、以及 Winform 等方面的整合）的时候值得考虑的线路之一。

一般情况下，由于 HTTP 协议的安全性，传递的参数容易被拦截，从而可能导致潜在的危险，所以一般 WebAPI 接口层都采用了 HTTPS 协议的，也就是采用 SSL 层来对数据进行安全性的加密的。

## 2.8.1. HTTPS 基础知识介绍

### 1） HTTPS

HTTPS（全称：Hypertext Transfer Protocol over Secure Socket Layer），是以安全为目标的 HTTP 通道，简单讲是 HTTP 的安全版。即 HTTP 下加入 SSL 层，HTTPS 的安全基础是 SSL，因此加密的详细内容就需要 SSL。 它是一个 URI scheme（抽象标识符体系），句法类同 http: 体系。用于安全的 HTTP 数据传输。https:URL 表明它使用了 HTTPS，但 HTTPS 存在不同于 HTTP 的默认端口及一个加密/身份验证层（在 HTTP 与 TCP 之间）。这个系统的最初研发由网景公司进行，提供了身份验证与加密通讯方法，现在它被广泛用于万维网上安全敏感的通讯，例如交易支付方面。

### 2）HTTPS 和 HTTP 的区别

一、https 协议需要到 ca 申请证书，一般免费证书很少，需要交费。

二、http 是超文本传输协议，信息是明文传输，https 则是具有安全性的 ssl 加密传输协议。

三、http 和 https 使用的是完全不同的连接方式，用的端口也不一样，前者是 80，后者是 443。

四、http 的连接很简单，是无状态的；HTTPS 协议是由 SSL+HTTP 协议构建的可进行加密传输、身份认证的网络协议，比 http 协议安全。

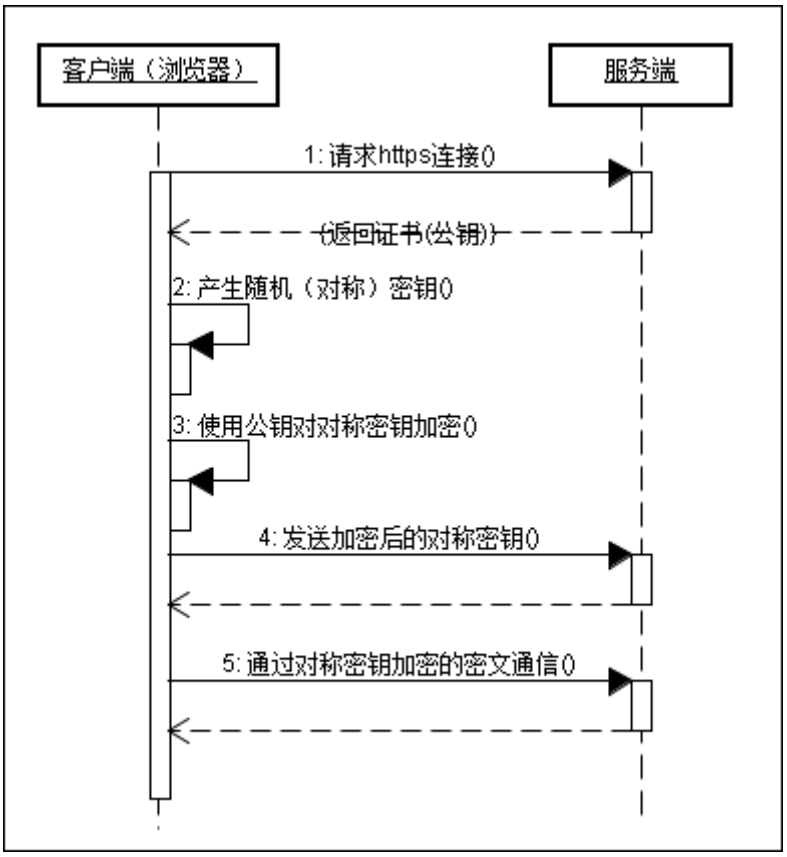
### 3）https 的实现原理

有两种基本的加解密算法类型

1)对称加密：密钥只有一个，加密解密为同一个密码，且加解密速度快，典型的对称加密算法有 DES、AES 等；

2)非对称加密：密钥成对出现（且根据公钥无法推知私钥，根据私钥也无法推知公钥），加密解密使用不同密钥（公钥加密需要私钥解密，私钥加密需要公钥解密），相对对称加密速度较慢，典型的非对称加密算法有 RSA、DSA 等。

https 的通信过程：




### 4） https 通信的优点

1）客户端产生的密钥只有客户端和服务器端能得到；
2）加密的数据只有客户端和服务器端才能得到明文；
3）客户端到服务端的通信是安全的。

## 2.8.2. SSL 基础知识介绍

**1）SSL 安全套接层协议(Secure Socket Layer)**

为 Netscape 所研发，用以保障在 Internet 上数据传输之安全，利用数据加密(Encryption)技术，可确保数据在网络上之传输过程中不会被截取及窃听。目前一般通用之规格为 40 bit 之安全标准，美国则已推出 128 bit 之更高安全标准，但限制出境。只要 3.0 版本以上之 IE 或 Netscape 浏览器即可支持 SSL。

当前版本为 3.0。它已被广泛地用于 Web 浏览器与服务器之间的身份认证和加密数据传输。

SSL 协议位于 TCP/IP 协议与各种应用层协议之间，是一种国际标准的加密及身份认证通信协议,为 TCP 提供一个可靠的端到端的安全服务，为两个通讯个体之间提供保密性和完整性(身份鉴别)。SSL 协议可分为两层：SSL 记录协议（SSL Record Protocol）：它建立在可靠的传输协议（如 TCP）之上，为高层协议提供数据封装、压缩、加密等基本功能的支持。SSL 握手协议（SSL Handshake Protocol）：它建立在 SSL 记录协议之上，用于在实际的数据传输开始前，通讯双方进行身份认证、协商加密算法、交换加密密钥等。

**2）SSL 协议特点**

1）SSL 协议可用于保护正常运行于 TCP 之上的任何应用协议，如 HTTP、FTP、SMTP 或 Telnet 的通信，最常见的是用 SSL 来保护 HTTP 的通信。
2）SSL 协议的优点在于它是与应用层协议无关的。高层的应用协议（如 HTTP、FTP、Telnet 等）能透明地建立于 SSL 协议之上。
3）SSL 协议在应用层协议之前就已经完成加密算法、通信密钥的协商以及服务器的认证工作。在此之后应用层协议所传送的数据都会被加密，从而保证通信的安全性。
4）SSL 协议使用通信双方的客户证书以及 CA 根证书，允许客户/服务器应用以一种不能被偷听的方式通信，在通信双方间建立起了一条安全的、可信任的通信通道。
5）该协议使用密钥对传送数据加密，许多网站都是通过这种协议从客户端接收信用卡编号等保密信息。常用于交易过程中。

**3）SSL 功能**
1)客户对服务器的身份认证:
SSL 服务器允许客户的浏览器使用标准的公钥加密技术和一些可靠的认证中心（CA）的证书，来确认服务器的合法性。

2)服务器对客户的身份认证:
也可通过公钥技术和证书进行认证，也可通过用户名，password 来认证。

3)建立服务器与客户之间安全的数据通道:
SSL 要求客户与服务器之间的所有发送的数据都被发送端加密、接收端解密，同时还检查数据的完整性。

## 2.8.3. 支持 SSL 的 CA 证书购买

上面介绍一些基础知识，我们可以简单的概括一下，就是引入了 HTTPS，可以很好解决接口参数的安全性问题。这些在很多大厂商的接口里面，都是使用 HTTPS 协议的，如腾讯微信、支付宝等接口，特别对于支付内容，使用 HTTPS 是必须的。
对于支持 HTTPS，核心的问题就是解决证书的问题，必须是由第三方权威机构颁发的证书，这样才能达到不被伪造的可能性，一般我们采用的是 CA 证书，这些证书是需要通过付费购买的（天下没有免费的午餐），证书的购买的供应商网站有很多，可以选择自己合适的进行购买。

**1）国外的 GoDaddy 平台购买证书过程**
这个参考博客园站长 dudu 的介绍内容，这个 goDaddy 是支持支付宝用美元购买的，使用平

台是英文，可以购买域名、证书等服务，是一个影响力较大的厂商。
1） 打开 godaddy.com 网站，通过菜单进入 Products -> SSL&Security -> SSL Certificates，选择 Protect All Subdomains("Wildcard")， 在 Pick your plan type 中，选择 Standard(Validates domain ownership)，然后完成购买。
2）进入 My Account -> SSL CERTIFICATES，创建证书（Certificate），创建时将之前得到的 CSR 内容复制到“CSR 文本框”中。
3）接入来进入 GoDaddy 的审批流程，在审批过程中需要验证域名的所有者（dns 填加记录或上传 html 文件至网站目录），验证成功后很快就会生成 CA 证书。
4）下载 CA 证书文件至生成 CSR 的服务器上。

普通证书费用是 63 美元左右，可以通过支付宝进行交易。

**2）国内沃通平台购买证书过程**

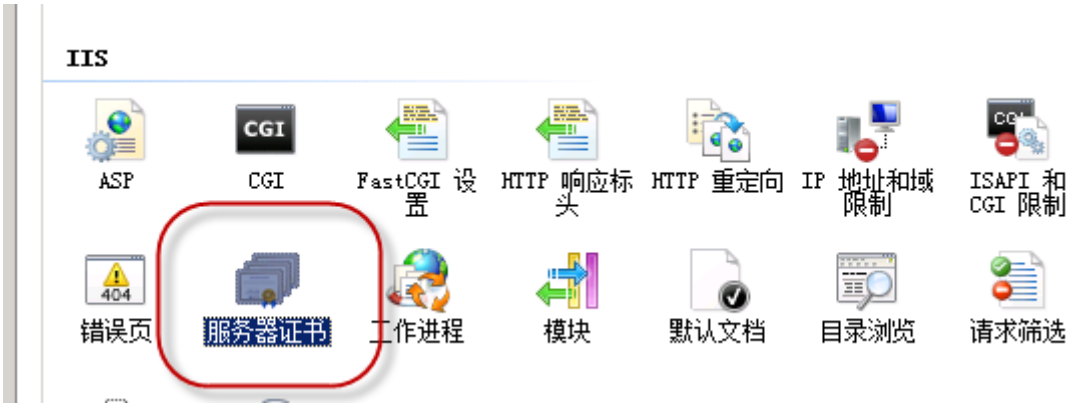
这个是纯中文版本的平台，比较方便使用，费用也比上面的贵一些，分的级别好像也多一些，相对上面那个国外的 GoDaddy 的三个种类证书，这个产品线分了 6 个类型。最低的也要接近 500 块，相对 GoDaddy 来说，费用要多一些了。由于是初步使用，也就购买了这个使用了。
申请购买也很简单，一步步按提示操作填写内容即可，大致分为这几步：

1）输入购买的证书类型，以及年限等资料；
2）输入域名的信息，以及需要域名对应的邮件进行验证；
3）提交购买订单后，使用在线支付或者使用公司账号汇款到指定账号；
4）客服验证后，技术人员提供证书生成操作，我们在列表里面尽快使用证书密码下载证书。

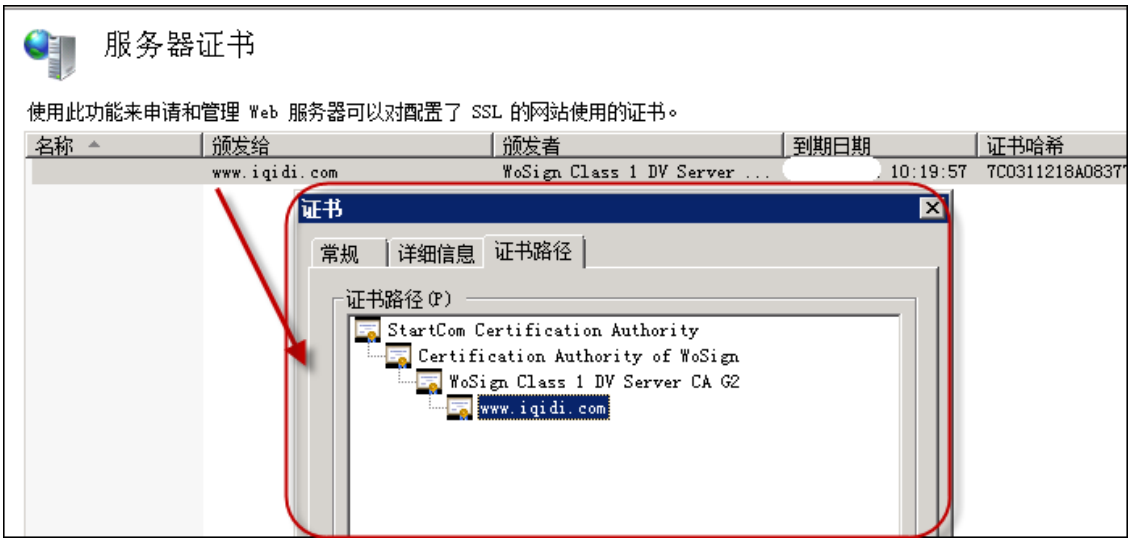
最后成功后，在订单信息里面，有这样的列表，提供了两个 SHA1 和 SHA2 两种加密协议的证书，官方建议使用 SHA2。

## 2.8.4. Web API 应用支持 HTTPS

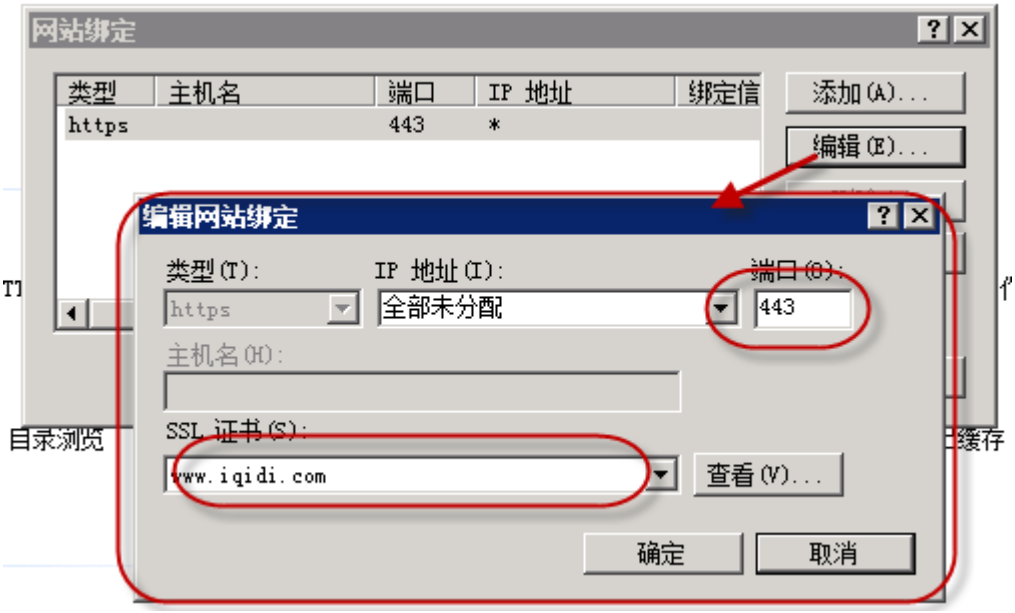
有了证书，我们在云服务器上（如阿里云）的 IIS 里面的证书模块里面，可以导入已有的证书。



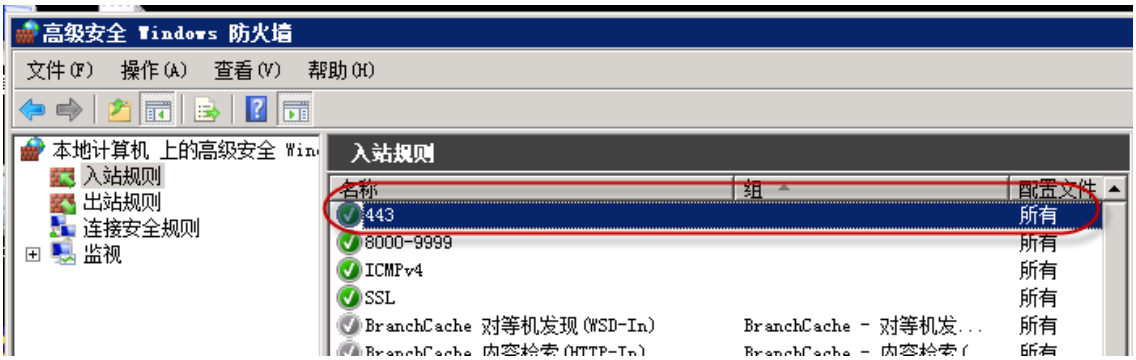
导入证书成功后，我们可以看到列表里面有具体的证书了，同时双击可以查看证书详细信息。




创建一个网站，并指定使用这个证书，端口采用 443 即可。




一般情况下，我们创建这个步骤后，你可以使用类似这样地址：https://www.iqidi.com/测试下具体的网站是否已经开通了，如果可以正常访问，那说明你的 443 端口已经启动侦听了，如果没有，可能是没有启动，或者是防火墙禁止了，一般情况下，我们还是需要在防火墙里面增加 443 端口的入站规则，设置为允许，否则可能被屏蔽了。



如果都设置了，可以通过 DOS 命令行来检查端口是否已经开始侦听了。

**查询 443 端口监听**
**netstat -ano | find "443"**

如果还是有问题，可以请求 CA 证书的供应商或者云供应商的技术人员帮你看看，一般都是会配合你解决的，我的开始也是折腾了他们一会，最后也莫名其妙的可以了。
最后成功后，在 Chrome 浏览器里面打开地址就有绿色的标志了。

## 2.9. Web API 自定义异常结果的处理

### 2.9.1. 常规的异常处理

统一的异常处理，把正确的信息返回给调用者很重要，可以让接口开发人员或者用户，了解具体的原因所在，这样可以得到有效的错误处理。

参考微信 API 的处理，微信 API，对于调用都有一个错误信息返回，不会直接裸露未经处理的异常，因此它们都是经过了一定的拦截处理，然后把错误信息包装提供给接口调用方的。如下是微信的一些接口处理错误。错误时微信会返回错误码等信息，JSON 数据包示例如下（该示例为 AppID 无效错误）：

```
{"errcode":40013,"errmsg":"invalid appid"}
```

我们根据自己的需要，定义了一个统一的错误信息实体类，如下所示。

```
/// <summary>
/// 接口返回的错误信息
/// </summary>
public class BaseResultJson
{
    /// <summary>
    /// 错误代码
    /// </summary>
    public int errcode { get; set; }

    /// <summary>
    /// 如果不成功，返回的错误信息
    /// </summary>
    public string errmsg { get; set; }

    /// <summary>
    /// 是否成功
    /// </summary>
    public bool success { get; set; }
}
```

这样我们就可以把拦截到的错误信息，转换为这样一个方便使用的实体类信息了。

拦截 Web API 的调用异常，一般可以结合 Try Catch 的方法，以及异常拦截器进行处理，如下是主动抛出的一些异常信息处理。

```
//如果没有通过，则抛出异常，由异常过滤器统一处理
if (!result.success)
{
    throw new MyApiException(result.errmsg, result.errcode);
}
```

其中 MyApiException 是自定义的一个异常信息，用来承载自定义错误信息的异常类。

异常拦截器，我们在 Web API 里面可以通过 Attribute 这种标签特性进行处理，如下是我在 Web API 的基类里面定义了一个异常处理器。

```
/// <summary>
/// 所有接口基类
/// </summary>

[ExceptionHandling]
public class BaseApiController : ApiController
```

这个特性对象的定义，它的代码如下所示。

```
/// <summary>
/// API自定义错误过滤器属性
/// </summary>
public class ExceptionHandlingAttribute : ExceptionFilterAttribute
{
    /// <summary>
    /// 统一对调用异常信息进行处理，返回自定义的异常信息
    /// </summary>
    /// <param name="context">HTTP上下文对象</param>
    public override void OnException(HttpActionExecutedContext context)
    {
        //自定义异常的处理
        MyApiException ex = context.Exception as MyApiException;
        if (ex != null)
        {
            throw new HttpResponseException(new HttpResponseMessage(HttpStatusCode.InternalServerError)
            {
                //封装处理异常信息，返回指定JSON对象
                Content = new StringContent(new BaseResultJson(ex.Message, false, ex.errcode).ToJson()),
                ReasonPhrase = "Exception"
            });

        }

        //记录关键的异常信息
        Debug.WriteLine(context.Exception);

        //常规异常的处理
        string msg = string.IsNullOrEmpty(context.Exception.Message) ? "接口出现了错误，请重试或者联系管理员" : context.Exception.Message;
        throw new HttpResponseException(new HttpResponseMessage(HttpStatusCode.InternalServerError)
        {
            Content = new StringContent(msg),
            ReasonPhrase = "Critical Exception"
        });
    }
}
```

根据这些代码，我们就可以实现对调用异常的统一封装处理，让它给我们返回统一的对象信息了，如下是其中一个调用异常，转换为自定义异常信息后的结果输出。

```
{"errcode":404,"errmsg":"请求的资源不支持 http 方法“POST”。
","success":false}
```

这样我们在处理 Web API 的返回结果的时候，可以先处理它的异常信息，具体的处理代码如下所示。

```
HttpHelper helper = new HttpHelper();
helper.ContentType = "application/json";

string content = helper.GetHtml(url, postData, true);
VerifyErrorCode(content);

T result = JsonConvert.DeserializeObject<T>(content);
return result;
```

我们在上面红色部分的代码就是先处理异常定义信息，如果有这些异常，我们可以在界面中进行异常处理显示了。

例如，如果自定义异常存在，我们转换后，把对应的信息显示出来，重新抛出异常即可。

```
BaseResultJson errorResult = JsonConvert.DeserializeObject<BaseResultJson>(content);
//非成功操作才记录异常，因为有些操作是返回正常的结果（{"errcode": 0, "errmsg": "ok"}）
if (errorResult != null && !errorResult.success)
{
    string error = string.Format("请求发生错误！错误代码：{0}，说明：{1}", (int)errorResult.errcode, errorResult.errmsg);
    LogTextHelper.Error(errorResult.ToJson());

    throw new Exception(error);//抛出错误
}
```

## 2.9.2. 地址接口异常处理

对于常规的异常，我们通过上面的处理方式，就可以很好进行拦截并处理了，如果接口异常是全局性的，如访问地址簿正确，或者参数多了几个信息，那么调用的接口就不是有效的地址，这样的话，返回的信息就不会被上面的拦截器进行处理了。

如我们给一个无效的 API 调用路径，在浏览器中获得下面错误结果。

上面结果就无法被我们的常规异常拦截器所捕获，因此不会输出经过封装好的异常信息。

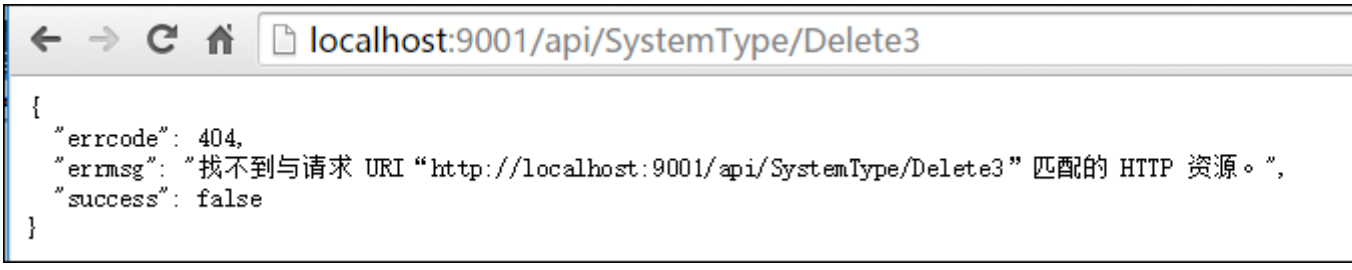
如果需要拦截，我们需要增加自己的消息代理处理，用来捕获这些特殊的异常信息。

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {

        ..............

        config.MessageHandlers.Add(new CustomErrorMessageDelegatingHandler());
```

上面红色部分就是我们自己添加的消息代理处理类，用来处理一些特殊的异常信息，如下代码所示。

```
/// <summary>
/// API自定义错误消息处理委托类。
/// 用于处理访问不到对应API地址的情况，对错误进行自定义操作。
/// </summary>
public class CustomErrorMessageDelegatingHandler : DelegatingHandler
{
    protected override Task<HttpResponseMessage> SendAsync(HttpRequestMessage request, CancellationToken cancellationToken)
    {
        return base.SendAsync(request, cancellationToken).ContinueWith<HttpResponseMessage>((responseToCompleteTask) =>
        {
            HttpResponseMessage response = responseToCompleteTask.Result;
            HttpError error = null;
            if (response.TryGetContentValue<HttpError>(out error))
            {
                //添加自定义错误处理
                //error.Message = "Your Customized Error Message";
            }

            if (error != null)
            {
                //获取抛出自定义异常，有拦截器统一解析
                throw new HttpResponseException(new HttpResponseMessage(HttpStatusCode.NotFound)
                {
                    //封装处理异常信息，返回指定JSON对象
                    Content = new StringContent(new BaseResultJson(error.Message, false, 404).ToJson()),
                    ReasonPhrase = "Exception"
                });
            }
            else
            {
                return response;
            }
        });
    }
}
```

经过了上面的处理后，我们进一步测试一下不存在的地址的异常处理结果，可以看到输出的内容是经过了自定义对象的转换了。




常规的调用，如果接口不对应，那么错误也是类似下面的消息

{"errcode":404,"errmsg":"找不到与请求 URI “http://localhost:9001/api/SystemType/Delete?signature=72f8d706c79dc14d70fc3f080d4706748d754021&timestamp=1443194061&nonce=0.650359861855563&appid=website_9A39C2A8&token=eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiIxIiwiaWFOIjoxNDQzMTk0MDM4LCJqdGkiOiI1YmEyYmE5NiOyZTA4LTQ1ZTgtYTAwNyO1MmY3OTkzYTg2NzEiLCJuYW1lIjoiYWRtaW4iLCJjaGFubmVsIjoiMCIsInNoYXJlZGtleSI6IjEyMzRhYmNkIn0.RRXQmmSCJzDK5Or6rmBL5wjd-YIJoEQFcOpOzqhR6IU” 匹配的 HTTP 资源。","success":false}

有了这些信息，我们就可以统一我们的调用规则，并进行异常记录和显示了，非常方便。

## 2.10. Web API 接口设计经验总结

### 2.10.1. 在接口定义中确定 MVC 的 GET 或者 POST 方式

由于我们整个 Web API 平台是基于 MVC 的基础上进行的 API 开发，因此整个 Web API 的接口，在定义的时候，一般需要显示来声明接口是[HttpGet]或者[HttpPost]，虽然有些接口也可以不用声明，但是避免出现类似下面的错误信息，显式声明还是有好处的。

```
请求的资源不支持 http 方法“POST
```

例如在基类定义的查找对象接口如下所示。

```
        /// <summary>
        /// 查询数据库,检查是否存在指定 ID 的对象
        /// </summary>
        /// <param name="id">对象的 ID 值</param>
        /// <returns>存在则返回指定的对象,否则返回 Null</returns>
        [HttpGet]
        public virtual T FindByID(string id, string token)
```

如果是增删改的接口，一般需要声明为 POST 方式提交数据，而且基于安全性的考虑，需要携带更多的参数。

```
    /// <summary>
    /// 插入指定对象到数据库中
    /// </summary>
    /// <param name="info">指定的对象</param>
    /// <returns>执行操作是否成功。</returns>
    [HttpPost]
    public virtual CommonResult Insert(T info, string token, string signature, string timestamp, string nonce, string appid)
```

## 2.10.2. 动态对象的接口定义

在一般的 Web API 接口里面，我们可能都会碰到很多简单类型的参数，但是又想让它们以 POST 方式提交数据，那么我们就可以有两种方法来处理，一种是定义一个类来放置这些参数，一种是采用动态的 JObject 参数，前者有很多不方便的地方，因为我们不可能为每个接口参数定义多一个实体类，这样可能会有很多难以管理的类定义。如下面是微信 API 的调用接口案例，我们也需要设置这样的处理规则。

```
接口调用请求说明
http 请求方式: POST（请使用 https 协议）
https://api.weixin.qq.com/cgi-bin/groups/update?access_token=ACCESS_TOKEN
POST 数据格式：json
POST 数据例子：{"group":{"id":108,"name":"test2_modify2"}}
```

那么我们采用 JObject 是这么样的呢，我们来看接口的定义和处理代码。JObject 是 Newtonsoft.Json.Linq 命名空间下的一个对象。

```
/// <summary>
/// 修改用户密码
/// </summary>
/// <param name="param">包含userName和userPassword的复合对象</param>
/// <param name="token">用户访问令牌</param>
/// <returns></returns>
[HttpPost]
public CommonResult ModifyPassword(JObject param,  string token)
{
    //令牌检查,不通过则抛出异常
    CheckResult checkResult = CheckToken(token);
    dynamic obj = param;
    if (obj != null)
    {
        string userName = obj.userName;
        string userPassword = obj.userPassword;

        bool success = BLLFactory<User>.Instance.ModifyPassword(userName, userPassword);
        return new CommonResult(success);
    }
    else
    {
        throw new MyApiException("传递参数出现错误");
    }
}
```

其中我们把 JObject 对象转换为我们所需要的对象的时候，因为我们没有定义具体的实体类，因此采用了 dynamic 语法，声明这是一个动态对象，由运行时获取对应的属性。

```
dynamic obj = param;
```

这样我们就可以在调用的时候，动态 POST 对应的 JSON 对象给 Web API 接口，而不需要预先定义各种接口参数的类了。

```
/// <summary>
/// 调用Web API接口，修改用户密码
/// </summary>
/// <param name="userName">用户名称</param>
/// <param name="userPassword">修改的密码</param>
/// <returns>如果修改成功返回true，否则返回false</returns>
public bool ModifyPassword(string userName, string userPassword)
{
    var action = "ModifyPassword";
    var postData = new
    {
        userName = userName,
        userPassword = userPassword
    }.ToJson();

    string url = GetTokenUrl(action);

    CommonResult result = JsonHelper<CommonResult>.ConvertJson(url, postData);
    return (result != null) ? result.Success : false;
}
```

其中 GetTokenUrl 是根据 token 和 API 的地址等参数，构建一个完整的提交地址。我们在上面代码通过

```
var postData = new
{
    userName = userName,
    userPassword = userPassword
}.ToJson();
```

就可以动态创建一个对象，并生成它的 JSON 字符串，把数据 POST 提交到对应的 API 接口里面即可，然后对结果进行对象的转换就算完成了。

## 2.10.3. 集合和分页的处理

在很多接口里面，我们都需要用到分页的处理，Web API 也不例外，这样可以提交数据检索效率，减少服务器数据处理的压力，同时也提交客户端的数据显示速度。

一般的集合接口定义如下所示（通用性基类接口）。

```
    /// <summary>
    /// 返回数据库所有的对象集合
    /// </summary>
    /// <returns>指定对象的集合</returns>
    [HttpGet]
    public virtual List<T> GetAll(string token)
    {
        //检查用户是否有权限，否则抛出MyDenyAccessException异常
        base.CheckAuthorized(AuthorizeKey.ListKey, token);

        List<T> list = baseBLL.GetAll();
        return list;
    }
```

但是这样的返回记录会比较多，一般情况下需要分页，那么分页的处理接口定义如下所示。

```
/// <summary>
/// 根据条件查询数据库,并返回对象集合(用于分页数据显示)
/// </summary>
/// <returns>指定对象的集合</returns>
[HttpPost]
public virtual PagedList<T> FindWithPager(string condition, PagerInfo pagerInfo, string token)
```

分页接口，在这里返回的结果里面，用了一个 PageList 的泛型类，这个方便我们获取当前的记录及总数，它的定义如下所示。

```
/// <summary>
/// 分页集合
/// </summary>
/// <typeparam name="T">对象</typeparam>
public class PagedList<T>
{
    /// <summary>
    /// 返回记录的总数
    /// </summary>
    public int total_count { get; set; }

    /// <summary>
    /// 列表集合
    /// </summary>
    public List<T> list { get; set; }
}
```

最后整个分页的处理 Web API 接口实现如下所示。

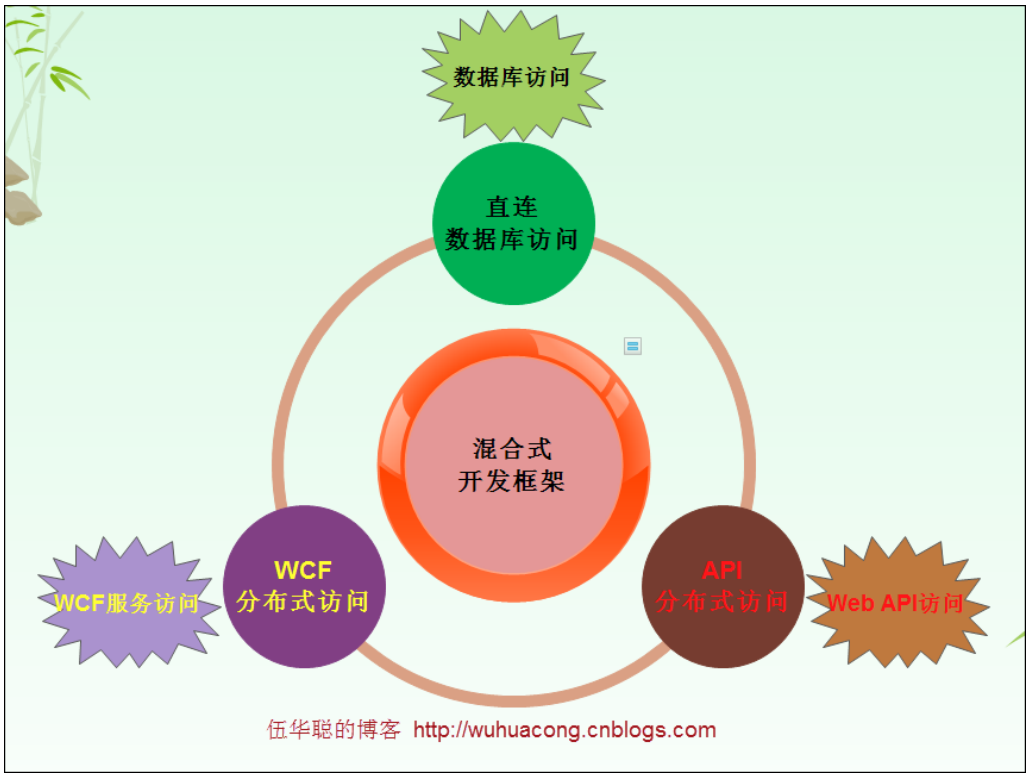
```
/// <summary>
/// 根据条件查询数据库,并返回对象集合(用于分页数据显示)
/// </summary>
/// <returns>指定对象的集合</returns>
[HttpPost]
public virtual PagedList<T> FindWithPager(string condition, PagerInfo pagerInfo, string token)
{
    //检查用户是否有权限，否则抛出MyDenyAccessException异常
    base.CheckAuthorized(AuthorizeKey.ListKey, token);

    List<T> list = baseBLL.FindWithPager(condition, pagerInfo);

    //构造成Json的格式传递
    var result = new PagedList<T>() { total_count = pagerInfo.RecordCount, list = list };
    return result;
}
```

客户端调用分页的 Web API 代码如下所示。

```
/// <summary>
/// 根据条件查询数据库,并返回对象集合(用于分页数据显示)
/// </summary>
/// <param name="condition">查询的条件</param>
/// <param name="pagerInfo">分页实体</param>
/// <returns>指定对象的集合</returns>
public virtual List<T> FindWithPager(string condition, ref PagerInfo pagerInfo)
{
    var action = "FindWithPager";
    string url = GetTokenUrl(action) + string.Format("&condition={0}", condition);
    var postData = pagerInfo.ToJson();

    List<T> result = new List<T>();
    PagedList<T> list = JsonHelper<PagedList<T>>.ConvertJson(url, postData);
    if (list != null)
    {
        pagerInfo.RecordCount = list.total_count;//修改总记录数
        result = list.list;
    }
    return result;
}
```

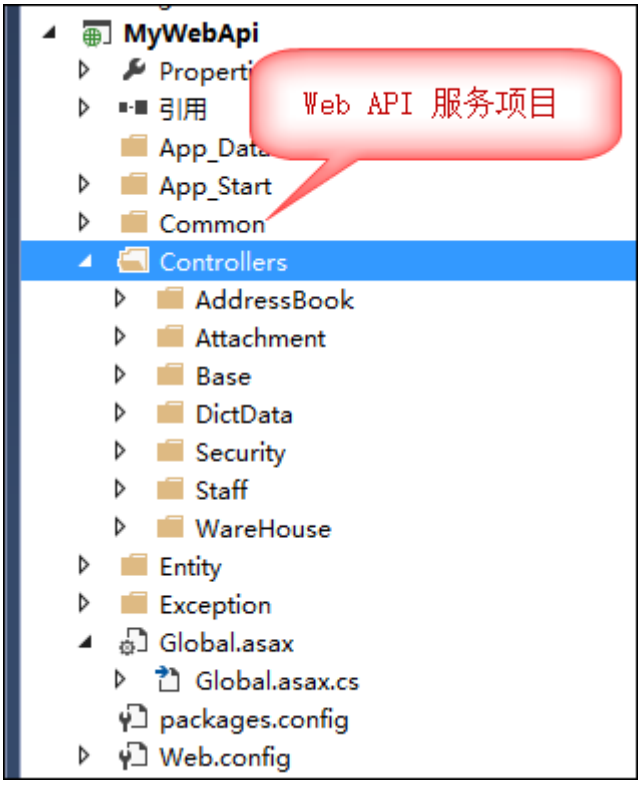
## 2.10.4. 混合框架界面整合 Web API 接口

在整个 Web API 的平台构建以及在混合框架的整合过程中，我把各个模块还是遵循相对独立的方式进行开发和整合，它们实现了从直接访问数据库、以 WCF 服务获取数据，以及通过 WebAPI 调用方式获取数据几种方式的统一，从而实现了整个混合框架的高度整合。

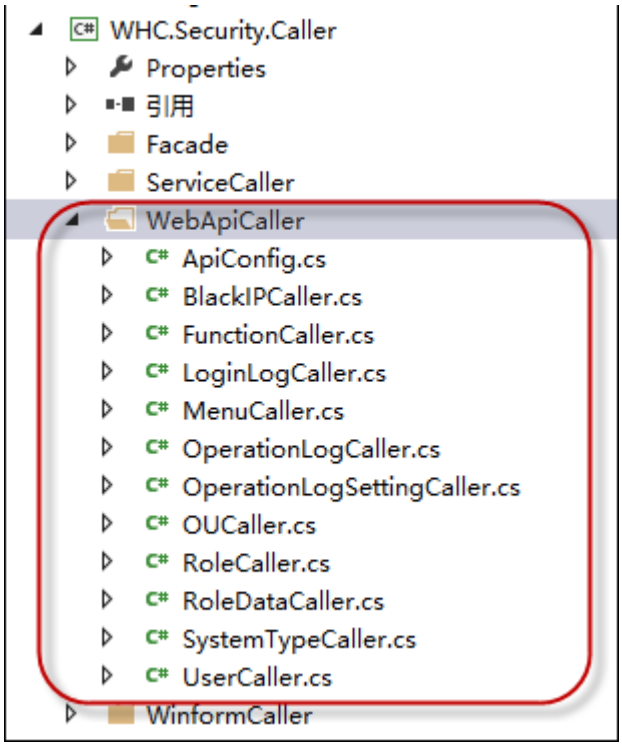



整个混合框架的核心是以相对独立的方式，整合各个可重用的模块，我们可以遵循一定的基础上，快速构建统一的应用平台。

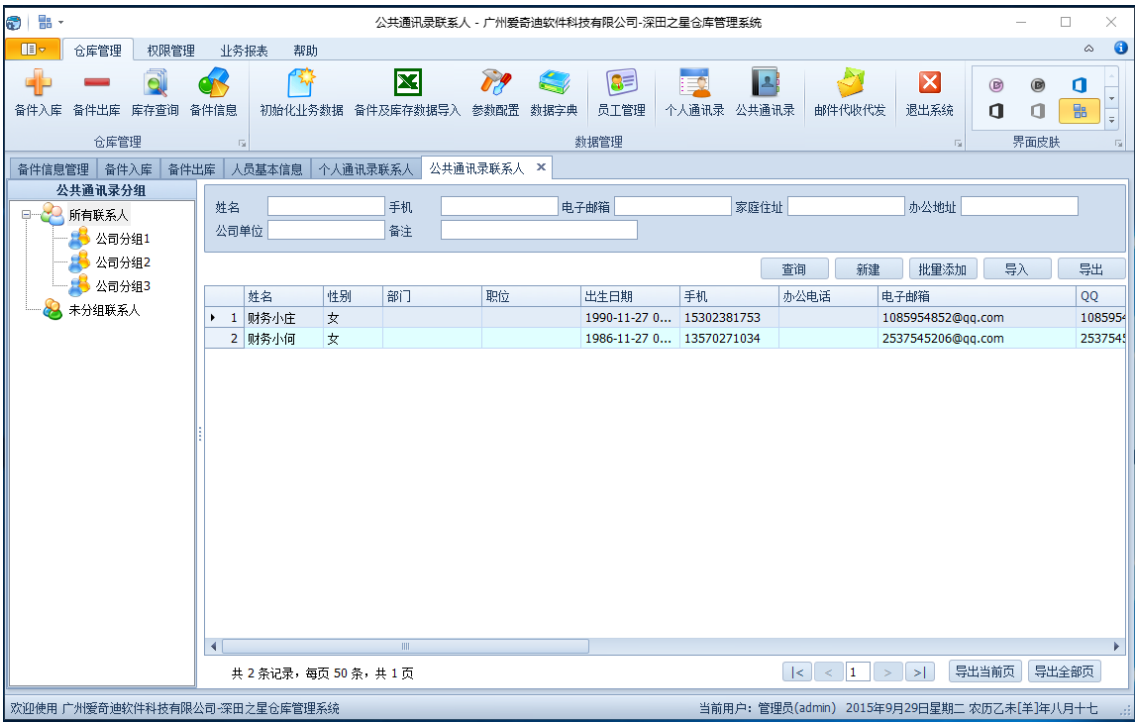
搭建完毕的整个 WebAPI 平台，其中包括了服务端内容，以 API 控制器的方式，发布了对应的 Web API 接口。




在每个混合框架的独立模块里面，我们封装了对应的 Web API 客户端调用处理，从而实现了 Web API 的调用方式。




在 Win10 下，使用 Web API 模式运行混合框架，获得的主体界面效果如下所示。




# 2.11. 开发框架中使用事务进行数据的统一处理

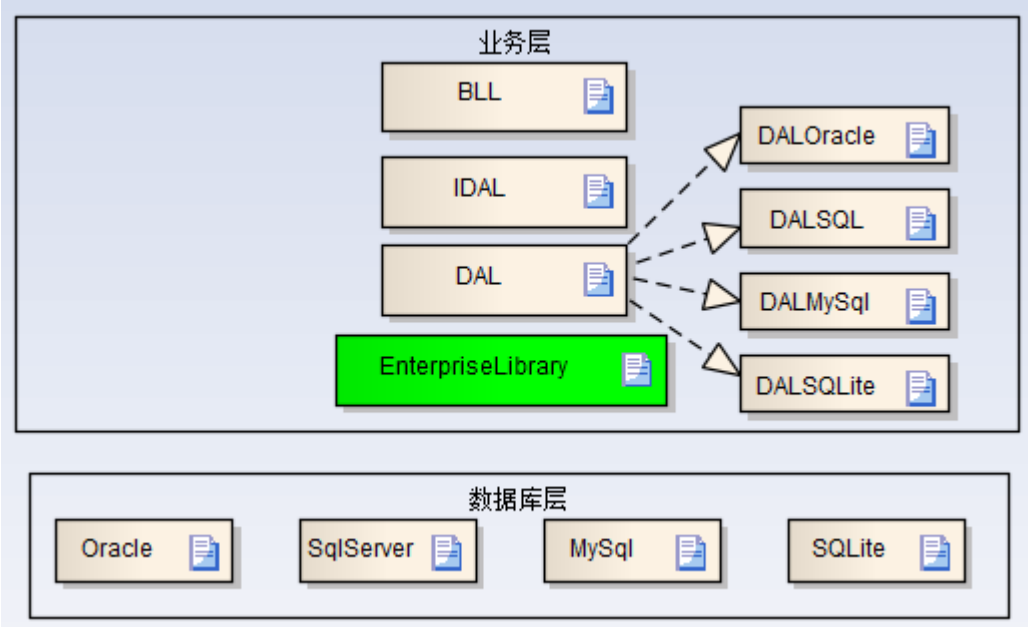
## 2.11.1. 框架的事务支持

在很多业务数据处理的场合，整条数据链的数据完整性是非常重要的，因为我们在系统里面，往往需要同时更新或者写入一些数据，如果其中任何一环处理错误，都应该逐条滚回，这种原子性的确保就是通过事务来进行的，本文介绍的这个事务处理，适用于我的所有开发框架，如 Winform 开发框架、混合式开发框架、Web 框架等。

由于上面介绍的这些框架都是基于业务逻辑层 BLL 层之上的，如复杂一点的混合式框架，在 BLL 层之上还有一个 WCF 服务层、或者 Web API 的数据提供层，因此为了适应多种框架的适用性，我们建议把业务规则封装在 BLL 层，这样各种应用框架使用的时候，代码就不用修改很多，而且业务逻辑统一，也很方便理解。

上面我提到的几个框架，在底层我们主要是采用了微软的 Enterprise Library 的数据库访问模块，因此它能够很好抽象各种数据库的事务，以适应各种不同数据库的事务处理。使用微软的 Enterprise Library 模块，可以很好支持 SQLSever、Oracle、Mysql、Access、SQLite 等数据库。

开发框架，常见的分层模式，可以分为 UI 层、BLL 层、DAL 层、IDAL 层、Entity 层、公用类库层等等

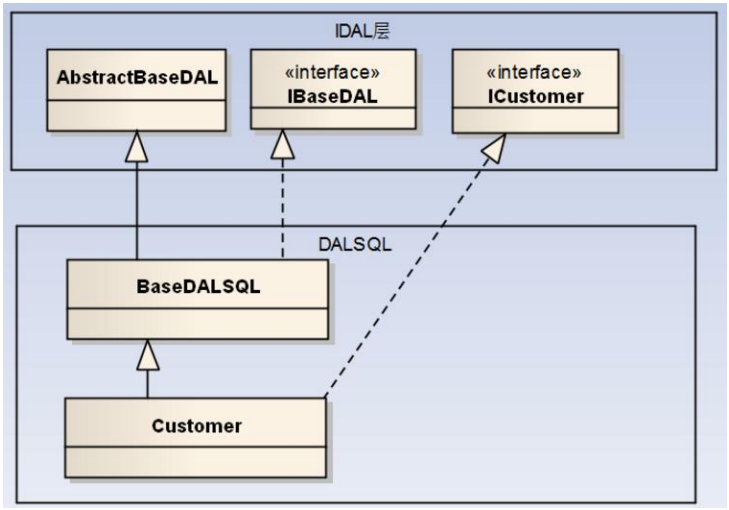



框架的基类我们封装了大量的通用性处理函数，包括数据访问层、业务逻辑层的基类，所有的基类函数基本上都带有一个 **DbTransaction trans = null** 的定义，就是我们可以采用事务，也可以默认不采用事务，是一个可选性的事务参数。

如数据访问类的部分接口定义如下所示。

```
/// <summary>
/// 数据访问层的接口
/// </summary>
public interface IBaseDAL<T> where T : BaseEntity
{
    /// <summary>
    /// 插入指定对象到数据库中
    /// </summary>
    /// <param name="obj">指定的对象</param>
    /// <param name="trans">事务对象</param>
    /// <returns>执行成功返回True</returns>
    bool Insert(T obj, DbTransaction trans = null);

    /// <summary>
    /// 根据指定对象的ID,从数据库中删除指定对象
    /// </summary>
    /// <param name="key">指定对象的ID</param>
    /// <param name="trans">事务对象</param>
    /// <returns>执行成功返回<c>true</c>，否则为<c>false</c>。</returns>
    bool Delete(object key, DbTransaction trans = null);

    /// <summary>
    /// 更新对象属性到数据库中
    /// </summary>
    /// <param name="obj">指定的对象</param>
    /// <param name="primaryKeyValue">主键的值</param>
    /// <param name="trans">事务对象</param>
    /// <returns>执行成功返回<c>true</c>，否则为<c>false</c>。</returns>
    bool Update(T obj, object primaryKeyValue, DbTransaction trans = null);

    /// <summary>
    /// 查询数据库,检查是否存在指定ID的对象
    /// </summary>
    /// <param name="key">对象的ID值</param>
    /// <param name="trans">事务对象</param>
    /// <returns>存在则返回指定的对象,否则返回Null</returns>
    T FindByID(object key, DbTransaction trans = null);

    ..................//其他操作
```

数据访问接口和数据访问类的实现图示如下所示。




因此在最高级的抽象基类 AbstractBaseDAL 的数据访问层里面，都有大量关于事务的接口可以使用，同样在 BLL 层也一样，里面大多数就是基于对 AbstractBaseDAL 的包装处理，同样预留了 **DbTransaction trans = null** 的参数方便我们进行事务性的处理。

```
AbstractBaseDAL.cs
WHC.Framework.ControlUtil
WHC.Framework.ControlUtil.AbstractBaseDAL<T>
FindByID(object key, D
 16      /// <summary>
 17      /// 数据访问层的超级基类，所有数据库的数据访问基类都继承自这个超级基类，包括Oracle、SqlServer、Sqlite、MySql、Access等
 18      /// </summary>
         15 个引用
 19      public abstract class AbstractBaseDAL<T> where T : BaseEntity, new()
 20      {
 21          构造函数
194
195          通用操作方法
551
552          #region 对象添加、修改
553
554          /// <summary> ...
             4 个引用
560          public virtual bool Insert(T obj, DbTransaction trans = null)...
569
570          /// <summary> ...
             2 个引用
576          public virtual int Insert2(T obj, DbTransaction trans = null)...
585
586          /// <summary> ...
             3 个引用
593          public virtual bool Update(T obj, object primaryKeyValue, DbTransaction trans = null)...
602
603          /// <summary> ...
             2 个引用
610          public virtual bool Update(CommandType commandType, string sql, DbTransaction trans = null)...
622
623          /// <summary> ...
             2 个引用
630          public virtual bool InsertUpdate(T obj, object primaryKeyValue, DbTransaction trans = null)...
640
641          /// <summary> ...
             2 个引用
648          public virtual bool InsertIfNew(T obj, object primaryKeyValue, DbTransaction trans = null)...
663
664          #endregion
665
666          #region 返回实体类操作
667
668          /// <summary> ...
             4 个引用
674          public virtual T FindByID(object key, DbTransaction trans = null)...
```

同样，在 BLL 层的基类 BaseBLL 里面的接口，也同样包含了这样的事务参数的，因此在框架里面整合事务是一件非常方便的事情，因为所有接口都预留了使用事务的可能，方便我们整合各个方法。

```
/// <summary>
/// 业务基类对象
/// </summary>
/// <typeparam name="T">业务对象类型</typeparam>
3 个引用
public class BaseBLL<T> where T : BaseEntity, new()
{
    构造函数

    #region 对象添加、修改、查询接口

    /// <summary> ...
    1 个引用
    public virtual bool Insert(T obj, DbTransaction trans = null)...

    /// <summary> ...
    1 个引用
    public virtual int Insert2(T obj, DbTransaction trans = null)...

    /// <summary> ...
    1 个引用
    public virtual bool Update(T obj, object primaryKeyValue, DbTransaction trans = null)...

    /// <summary> ...
    0 个引用
    public virtual bool Update(CommandType commandType, string sql, DbTransaction trans = null)...

    /// <summary> ...
    0 个引用
    public virtual bool Update(object id, Hashtable recordField, string targetTable, DbTransaction trans)...

    /// <summary> ...
    1 个引用
    public virtual bool InsertUpdate(T obj, object primaryKeyValue, DbTransaction trans = null)...

    /// <summary> ...
    1 个引用
    public virtual bool InsertIfNew(T obj, object primaryKeyValue, DbTransaction trans = null)...
```

## 2.11.2. 框架事务接口的使用

在框架的 DAL 层和 BLL 层，我们都定义了很多公用的、带有事务参数的接口，如果我们在业务处理里面，使用事务的话，那么也是很方便的事情。如在

事务的处理，除了包含对数据库的更新修改需要使用外，如果在事务的处理中，查询数据等接口也要一并采用事务处理，否则就容易发生锁住堵塞的情况，因此，如果使用事务，那么事务参数是全程需要传入来进行数据的检索、修改等操作的。

事务的标准使用过程，一般是先创建事务对象，然后在各个接口里面使用事务处理，如下代码示例所示。

```
DbTransaction trans = base.CreateTransaction();
if (trans != null)
{
    string sql = ............;
    base.SqlExecute(sql, trans);

    sql = ............;
    base.SqlExecute(sql, trans);

    try
    {
        trans.Commit();
        result = true;
    }
    catch
    {
        trans.Rollback();
        throw;
    }
}
```

如果我们定义的处理接口，只是其中事务处理的一个环节，那么我们就需要在接口定义的时候，预留有 **DbTransaction trans = null** 的参数，示例代码如下所示。

```
/// <summary>
/// 添加积分
/// </summary>
/// <param name="id">会员ID</param>
/// <param name="points">积分（正数为加，负数为减）</param>
/// <returns></returns>
public bool AddPoints(string id, decimal points, DbTransaction trans = null)
{
    string sql = string.Format("Update {0} Set TotalPoints = TotalPoints + {1} Where ID='{2}' ", tableName, points, id);
    return SqlExecute(sql, trans) > 0;
}

/// <summary>
/// 添加消费金额
/// </summary>
/// <param name="id">会员ID</param>
/// <param name="amount">消费金额（正数为加，负数为减）</param>
/// <returns></returns>
public bool AddConsumption(string id, decimal amount, DbTransaction trans = null)
{
    //消费的时候，同时修改累计消费金额和最后到店时间
    string sql = string.Format("Update {0} Set TotalConsumption = TotalConsumption + {1}, LastConsumptionDate=getdate() Where ID='{2}' ", tableName, amount, id);
    return SqlExecute(sql, trans) > 0;
}
```

在 BLL 层，我们同样可以对这些接口预留事务接口，对数据访问的 DAL 层进行封装的。

```
/// <summary>
/// 添加积分
/// </summary>
/// <param name="id">会员ID</param>
/// <param name="points">积分（正数为加，负数为减）</param>
/// <returns></returns>
public bool AddPoints(string id, decimal points, DbTransaction trans = null)
{
    IMember dal = baseDal as IMember;
    return dal.AddPoints(id, points, trans);
}

/// <summary>
/// 添加消费金额
/// </summary>
/// <param name="id">会员ID</param>
/// <param name="amount">消费金额（正数为加，负数为减）</param>
/// <returns></returns>
public bool AddConsumption(string id, decimal amount, DbTransaction trans = null)
{
    IMember dal = baseDal as IMember;
    return dal.AddConsumption(id, amount, trans);
}
```

预留这样的事务参数，对于我们在整合各个处理过程有很大的帮助，否则任何一环没有事务参数，处理就可能出现问题，这也是为什么框架的基类参数，都提供了一个事务参数的可选参数，就是这个道理。

如整合上面积分和金额的处理过程，就是会员系统里面必要的一环，而且这些数据需要连贯在一起进行修改，不能再中间任何一个环节发生数据部分丢失或者不成功的操作，因此我们必须使用事务处理过程。

例如，我在会员消费的过程处理中，其中包含了对消费商品主表、明细表的记录，还包括对库存的调整、还有会员消费、积分赠送等等细粒度的处理，因此我们在事务里面可以按照下面的方式进行处理。

```
        DbTransaction trans = base.CreateTransaction();
        if (trans != null)
        {
            try
            {
                bool success = BLLFactory<MemberConsumption>.Instance.Insert(info, trans);
                if (success)
                {
                    //保存消费明细记录
                    foreach (ConsumptionDetailInfo detail in detailList)
                    {
                        BLLFactory<ConsumptionDetail>.Instance.Insert(detail, trans);

                        //减少对应库存
                        string note = "会员消费";
                        double quantity = (-1) * detail.Quantity;
                        BLLFactory<Stock>.Instance.ModifyQuantity(info.Corp_ID, info.Shop_ID, info.Creator, detail.ProductNo, quantity, note, trans);
                    }

                    //增加消费金额
                    BLLFactory<Member>.Instance.AddConsumption(info.Member_ID, info.Amount, trans);
                    //添加积分（消费一元积一分）
                    BLLFactory<Member>.Instance.AddPoints(info.Member_ID, info.Amount, trans);
                    //减少余额
                    decimal subBalance = (-1) * info.Amount;
                    BLLFactory<Member>.Instance.AddBalance(info.Member_ID, subBalance, trans);
                }

                trans.Commit();
                result.Success = true;
            }
            catch (Exception ex)
            {
                LogTextHelper.Error(ex);
                trans.Rollback();
                result.ErrorMessage = ex.Message;
            }
        }
```

上面的方法就是一个完整的事务处理过程，就是在会员消费的情况下发生的，如果我们需要考虑多种应用框架的封装，如 Web、Web API、Winform、WCF 等方式的调用，那么我们就把它放到了业务逻辑层 BLL 层进行封装，如下就是 BLL 层的方法。

```
        /// <summary>
        /// 保存消费记录，同时修改库存
        /// </summary>
        /// <param name="info">消费主记录</param>
        /// <param name="detailList">消费明细列表</param>
        /// <returns></returns>
        public CommonResult SaveConsumption(MemberConsumptionInfo info, List<ConsumptionDetailInfo> detailList)
```

这样我们在 Web API 的调用的时候，就可以不在使用这个特定的 **DbTransaction trans = null** 参数了，因此上面的方法体就是一个最小的操作单元了。

## 2.11.3. 混合框架中 Web API 的封装和调用

在我的混合式开发框架基础上，我们服务提供可以是传统 Winform、WCF，以及 WebAPI 的方式，对于 Web API 模式，我们对 BLL 业务逻辑层进行了封装，它的 APIController 的方法也就是说如下所示。

```
/// <summary>
/// 保存消费记录，同时修改库存
/// </summary>
/// <param name="info">消费主记录</param>
/// <param name="detailList">消费明细列表</param>
/// <returns></returns>
[HttpPost]
public CommonResult SaveConsumption(JObject param, string token, string signature, string timestamp, string nonce, string appid)
{
    //如果用户签名检查不通过，则抛出MyApiException异常。
    base.CheckTokenAndSignatrue(token, signature, timestamp, nonce, appid);

    dynamic obj = param;
    if (obj != null)
    {
        MemberConsumptionInfo info = obj.info;
        List<ConsumptionDetailInfo> detailList = obj.detailList;

        return BLLFactory<MemberConsumption>.Instance.SaveConsumption(info, detailList);
    }
    else
    {
        throw new MyApiException("传递参数错误");
    }
}
```

这里面的方法有很多参数，第一个 JObject param 是一个动态对象的定义参数，具体可以参考“Web API 接口设计经验总结”里面介绍的“动态对象的接口定义”节点了解。
由于这个处理过程是对数据进行了修改等重要的处理，因此参数需要增加签名数据，以及 Token 身份的标识，而且整个接口是公布在 HTTPS 协议的基础上，因此 接口的安全性是得到了非常好的保证。

对于在混合框架中，访问 Web API 的接口安全性方面，整个框架里面，最有保证、最方便的、适应最广泛应用的就是基于 Web API 的方式接入了，它不仅可以在桌面程序进行处理，也可以在移动端（包括 APP 和微信公众号等），使用 Web API 的接口进行数据的获取和提交。WCF 方式虽然功能强大，但是相对显得笨重一些，而且数据安全性，需要服务端和客户端采用 X509 证书进行通讯，对移动端则是很难的一件事情。

言归正传，上面的事务处理，在它的基础上 Web API 接口进行了封装，我们调用 Web API 就不需要进行事务的参数传递了，因为它已经是一个操作的整体了，要么成功，要么全部失败滚回即可。

为了适应在混合框架的 Winform 里面进行调用，我们还是需要对刚才的 Web API 接口进行了客户端的封装，给 Web API 传递对应的参数（通过 POST 方式提交 JSON 参数给 Web API 接口），具体的代码如下所示。

```
public CommonResult SaveConsumption(MemberConsumptionInfo info, List<ConsumptionDetailInfo> detailList)
{
    var action = "SaveConsumption";
    var postData = new
    {
        info = info,
        detailList = detailList
    }.ToJson();
    string url = GetPostUrlWithToken(action);

    return JsonHelper<CommonResult>.ConvertJson(url, postData);
}
```

这个过程就是封装了对 Web API 的调用，并通过 JSON 数据返回的方式，把他们转换为对应的结果对象，这里的结果是一个通用的结果集对象 CommonResult 。

这样我们在 Winform 的客户端里面就有了统一的调用方式了，非常方便简洁，代码如下所示。

```
//获取消费明细
List<ConsumptionDetailInfo> detailList = GetConsumptionDetail();
//保存消费明细，以及在后台利用事务处理各种关系的修改
CommonResult result = CallerFactory<IMemberConsumptionService>.Instance.SaveConsumption(info, detailList);
```