自从项目上采用敏捷开发的流程以后，我们的开发任务中出现了不少“联调”的任务，而所谓的“联调”任务，完全是拜前后端分离所赐。通常来讲，按照前后端分离的思想，我们的团队会被分成前端和后端两个组，前端负责页面内数据的展示，后端负责提供相关服务的接口。这样听起来非常合理，对吧？可问题在于，后端常常在等前端联调这些接口，因为后端不知道具体有哪些异常需要处理；同样，前端常常在等后端接口稳定，因为一旦出现问题，就会导致接口发生变更。虽然在此之前，我们早已花了一周左右的时间去讨论接口，接口文档早已伴随着API部署到线上，可我们依然需要大量的时间去沟通每个接口的细节。用一种什么样的语言来描述这种状态呢？大概就是人们并不是真的需要接口文档，因为真的不会有人去看这东西。

从敏捷开发到产品架构

为什么会出现这种情况呢？我想，可以从三个方面来考虑，即设计不当、进度不一、沟通不畅。有时候集思广益去讨论一个接口，可能并不是一件好事，因为考虑的因素越多，问题就会变得越复杂，相应地妥协的地方就会越多。我并非不懂得做人需要适当妥协，事实是从妥协的那一刻起，我们的麻烦越来越多。有人问怎么能消灭Bug，我说消灭需求就可以了。现代人被各种各样的社交网络包围着，以至于隐私都被赤裸裸地暴露在空气中，可你很难想象人与人之间的沟通会越来越困难，难道是因为社交网络加剧了人类本身的孤独？没有人是一座孤岛，可前后端分离好像加剧了这种界限。现在动辄讲究全栈，可当你把精力都耗费在这些联系上去，你如何去追求全栈？相反，我们像电话接线员一样，在不停地切换上下文，因为我们要“敏捷”起来，可作为工程师就会知道，切换上下文需要付出相应的代价。

我之所以提到这样一个场景，是出于对当前项目的一种整体回顾。我们的项目是一个客户端产品，但是它依然体现了前后端分离的思想。受业务背景限制，这个客户端采用了Native + Web的技术架构。如果你了解整个互联网产品形态的演变历程，就会对这种技术架构非常的了解，从曾经的Native和Web之争，到所谓的Hybrid App，再到如今的React Native及小程序，这种技术架构其实一直都存在，譬如Electron、Atom、Node-Webkit、Cordova、Ionic、VSCode等等，其实都是非常相近的技术。对应到我们的项目，我们提供了一个JSBridge来完成Native层和Web层之间的通信，而客户端的渲染实际上是由前端来完成的，所以你可以想到，我们通过一个WebView来加载页面，而平台相关的交互由C++/C#来完成，所以，理论上客户端是是一个和Electron类似的壳子(Shell)，它可以展示来自任何页面的内容。

从客户端的角度来讲，它是Native层接口的提供者，连接着平台相关的API，并集成了第三方的硬件设备，所以，理论上它是和具体业务无关的。可实际上，因为Web层不能直接和文件系统交互，所以，像上传、下载这样本该由前端调用的接口，部分地转移到了客户端这边，所以，客户端无可避免地受到后端API变化的影响，因为业务上需求存在差异，上

传接口前后共发生了三次变化，所以，客户端中存在三个版本的上传，当然，我相信这是一个设计上的问题，通过改进设计可以得到完美的解决。关于上传为什么会这么复杂，感兴趣的朋友可以通过留言来一起交流。这里我想说的是什么呢？因为客户端希望与具体业务无关，所以，客户端注定是以功能来划分服务，然后通过JSBridge暴露给Web层。可是对后端的微服务架构而言，它的服务是以业务为主导的，它的一个业务就是一个接口。由此导致一个问题，后端接口的数量不断增加，客户端面临频繁地改动。

不做平庸的ApiCaller

有很多人说，今天的编程工作变得越来越简单，对于这一点我非常认同。因为，无论是无论是语言、工具、生态、平台，都获得空前的繁荣，所以，我们大多数人的工作，可能就是调用现成的API，而少数人的工作，可能就是提供友好的API，甚至连代码你都可以在Google上找到，你唯一要做的就是Ctrl + C & Ctrl + V。当初想要改变世界的你我，突然有一天就变成了ApiCaller，甚至大多数的框架，你连底层细节都无从得知。可你真的打算做一个平庸的ApiCaller吗？至少我是不愿意的，因为在我看来，调用后端提供的API，大多数情况下都是换个URL，或者换个参数，这样的代码你写一次以后，剩下的基本就是复制和粘贴了，你可能会非常鄙视我的这种行为，可事实就是这样的，不单单我在复制，连我身边的同事都在复制。可这能怎么办啊，只要后端提供了新接口，或者是对接口进行了调整，而这些接口必须由客户端封装，我们的工作就永远不会停止，可这不过调用后端的API 而已啊！

有时候，我们会说工作经验和工作时间未必是正相关的，因为如果我们十年都在做一件事情，那么其实和一年是没有区别的。为了避免成为一个平庸的ApiCaller，你必须思考那些真正重要的事情。怎么能降低后端API变化对客户端的影响呢？降低耦合度。怎么降低耦合度呢？依赖抽象而非依赖具体。想想WebService，它通过WSDL来对服务进行描述，而通过WSDL就可以在客户端创建代理类，一旦WebService发生变更，重新生成代理类就好。再回想一下，调用后端API会遇到那些问题？设置Header、设置Cookie 、拼接URL、拼接参数、URLEncode、SSL、JSON序列化、FormData、上传文件、编码/解码等等，是不是每一次都在处理这些问题？看到项目里用HttpWebRequest去构造Mulitpartfile结构，我忽然间觉得绝望。既然每次都是翻来覆去这些东西，为什么要用手来写？API文档构建工具可以帮助用户生成curl以及常见语言对应的代码，所以，我有理由相信，我们需要一个东西来帮助我们完成这个工作，就像WebService生成代理类一样。那么，有没有这样一个东西呢？这就是本文的主角——基于声明式的RESTful风格的客户端：WebApiClient。

.NET下的Retrofit：WebApiClient

WebApiClient是.NET平台下的Retrofit。要理解这句话，首先要理解Retrofit。什么是Retrofit呢？Retrofit是一个Android/Java下的网络通信库，其本身基于okHttp，熟悉Android开发的朋友，对这个库应该不会感到陌生。Retrofit帮助我们解决了上文中提到的，在请求一个Web API时会遇到的问题，并通过注解这种技术，以一种声明式的方式来定义接口。简单来说，所有你想要调用Web API都是接口中的一个方法，你通过注解来告诉Retrofit，该方法会请求哪一个Web API，参数会以什么样的形式传递过去，结果会以什么样的形式返回回来，你完全不必去写那些底层HTTP通信相关的东西，因为Retrofit会帮你在运行时实现这个接口。所以，我们说Retrofit是一种声明式的HTTP客户端。声明式我们见过相当多啦，Java里的注解，C#里的Attribute、Python里的装饰器、JavaScript里的修饰器，以及如今各种各样的双向绑定框架。下面，我们来一起看看WebApiClient这个库。

现在，我假设你手里已经有可供调用的Web API，并且你真实地了解这些Web API是如何工作的。至此，我们需要完成的工作主要都集中在客户端，这里我们编写一个控制台应用来完成这一工作。首先，需要在项目中引入WebApiClient这个库，我们直接通过Nuget来完成安装即可(注：这里共有Laojiu.WebApiClient、WebApiClient.JIT和WebApiClient.AOT三个版本，本文使用的是Laojiu.WebApiClient)。使用WebApiClient的基本流程是：首先，定义一个继承自IHttpApiClient的接口并在接口中声明相关方法；其次，通过Attribute对接口中的方法和参数进行修饰以完成和Web API的绑定；最后，通过WebApiClient生成该接口的一个实例，而通过调用相应的实例方法就可以得到结果。这是不是和代理类的感觉非常像呢？像博主这样懒惰的人，或许连接口都不愿意亲自去写，因为我相信越是严谨的规则，就越是适合应用到自动化上面去。这样说可能无法让大家形成对WebApiClient的直观印象，那么让我们从一个简单的例子开始吧！

Get请求接口

```
[HttpHost("http://localhost:8000")]
public interface IValuesApiCaller : IHttpApiClient
{
    //GET http://localhost:8000/values1
    [HttpGet("/values1")]
    [OAuth2Filter]
    ITask<string> GetValues();

    //GET http://localhost:8000/values1/{id}
    [HttpGet("/values1/{id}")]
```

```
    [OAuth2Filter]
    ITask<string> GetValue(int id);
}
```

在这个示例中，我们展示了WebApiClient是如何处理带参数以及不带参数的Get请求的。通过HttpGet特性，我们分别为GetValues()和GetValue()两个方法指定了请求的URL。虽然在这里我们指定一个完整的URL，可是考虑到我们Web API通常都是分布在不同的域名下，所以我们可以通过HttpHost特性来配置一个BaseURL。接口的返回值为ITask，我们可以通过我们的需要指定相应的类型，在这里我们以ITask为例，特别说明的是，如果服务器返回的是标准的JSON格式，那么我们可以将其映射为相应的实体结构，这就需要使用JsonReturn标特性对方法进行修饰。我们知道Get请求可以通过QueryString形式来进行传参，那么这一点在WebApiClient中如何实现呢？这就用到所谓的“平铺参数”，即我们在方法中声明的参数会被WebApiClient自动地追加到URL上面去，再不需要去手动地拼接这些参数；同理，这些参数可以用一个包装类封装起来，具体大家参考官方文档。

OK，现在来看看如何调用IValuesApiCaller这个接口。我们在前面说过，WebApiClient会帮助我们生成一个IValuesApiCaller的实例，所以我们调用一个Web API的时候，关注点已然从之前的过程实现转变为接口实现，这正是我们渴望看到的局面。一个非常简洁的调用示例：

```
//调用Values Service
using (var client = HttpApiClient.Create<IValuesApiCaller>())
{
    Console.WriteLine("-----Invoke Values Service-----");
    var results = await client.GetValues().InvokeAsync();
    Console.WriteLine($"results is {results}");
    var result = await client.GetValue(10).InvokeAsync();
    Console.WriteLine($"result is {result}");
}
```

Post请求接口

接下来，我们再来说说Post请求接口。同样的，这里我们使用博主编写好的一个Service，我们称之为Student Service。它使用了EF Core来完成数据库的读写，它提供了一组和Student实体相关的API，这里我们使用它来作为Post请求接口的示例实现。因此，我们首先定义一个接口IStudentApiCaller：

```
[HttpHost("http://localhost:8000")]
public interface IStudentApiCaller : IHttpApiClient
{
    //GET http://localhost:8000/student
    [HttpGet("/student")]
    [OAuth2Filter]
    [JsonReturn]
    ITask<List<Student>> GetAllStudents();


    //POST http://localhost:8000/student
    [HttpPost("/student")]
    [OAuth2Filter]
    ITask<string> NewStudent([JsonContent] Student student);
}
```

这里重点关注接口中的第二个方法。首先，它是一个Post请求；其次，它接受一个JSON格式的文本作为它的请求体，所以我们这里使用了JsonContent特性。前面我们提到过，接口返回类型ITask，可以映射为对应的实体结构。注意到GetAllStudtents()这个方法中绑定的API，它负责从数据库中查询所有的Student信息并以JSON形式返回，所以这里我们将其映射为List。与此同时，你会注意到JsonReturn特性，这是在告诉WebApiClient，你希望将返回的结果映射为强类型的模型；同理，你可以使用XmlReturn特性来处理返回值为Xml的情形。除此之外，你还可以使用FormContent特性来修饰方法参数，其作用是将模型参数以key1=value1&key2=value2……的形式写入请求体中，对应于x-www-form-urlencode；更一般地，你可以使用FormField特性修饰方法参数，以form-data的形式写入请求体中。Mulitpart是最为讨厌的一种数据格式，请大家自己去看官方文档。

过滤器与OAuth2

无论如何，请允许我说，这是我最喜欢的一个特性。大家会注意到，在我的示例代码中，有一个东西一直没有去说，这就是OAuth2Filter，这其实是自己扩展的一个特性，这意味着在请求该API前，需要通过OAuth2授权以获得身份令牌。对于这一点，我想大家都是清楚的，因为在微服务架构中，Web API是作为一种受保护的资源而存在的，所以鉴权和授权是非常重要的点。以博主的项目组为例，我们做到第三个项目的时候，整个后端的OAuth2认证服务终于实现了统一，可即使如此，每一次这种基础设施都需要联调，都要考

虑到底使用哪一种授权模式。譬如，客户端是考虑把token存放在全局静态类里，而前端是考虑把token存放在Cookie里，甚至在此之前，我们连refresh_token都没有，客户端在调用Web API时天天担心token过期，于是在调用Web API时主动去刷新一次token。你问我为什么不判断一下token有没有过期，因为后端没有提供这个接口呀。其实，我想说的只有一句话，基础设施请交给框架去处理。

WebApiClient提供了用于请求管道中的过滤器，可以让我们在请求前、请求后搞点事情。譬如，我们这里希望在请求前获取token，并将其追加到当前请求的Header里，或者是在请求前判断下token是否过期(假如后端愿意开发这个接口的话)，如果过期了就自动刷新下token，该怎么做呢？首先，我们定义一个IAuthApiCaller的接口，它负责从认证服务器上获取token，这里选择客户端模式：

```
[HttpHost("http://localhost:28203")]
public interface IAuthApiCaller : IHttpApiClient
{
    [HttpPost("/oauth2/token")]
    ITask<string> GetToken([FormField] string client_id,[FormField] string
client_secret,[FormField] string grant_type = "client_credentials");
}
```

接下来，我们继承ApiActionFilterAttribute来编写OAuth2FilterAttribute，显然，它会在请求前调用IAuthApiCaller接口实例，这里我们将client_id和client_secret硬编码到代码里，单单是为了演示如何去印证这个想法，实际项目中大家可以考虑通过配置或者是传参来实现：

```
[AttributeUsage(AttributeTargets.Method)]
public class OAuth2FilterAttribute : ApiActionFilterAttribute
{
    public override Task OnBeginRequestAsync(ApiActionContext context)
    {
        using (var client = HttpApiClient.Create<IAuthApiCaller>())
        {
            var client_id = "578c06935d7f4c9897316ed50b00c19d";
            var client_secret = "d851c10e1897482eb6f476e359984b27";
            var result = client.GetToken(client_id, client_secret).InvokeAsync().Result;
```

```
            var token = json["access_token"].Value<string>();
            context.RequestMessage.Headers.Authorization = new
AuthenticationHeaderValue("Bearer",token);
            return base.OnBeginRequestAsync(context);
        }
    }
}
```

至此，我们只需要给需要需要授权的API添加OAuth2Filter特性即可，全然不需要考虑这个token如何储存的问题。我对静态类和静态方法没有误解，仅仅是因为它是反模式的，任何全局内可以修改的成员，不管有没有人会去修改，它始终都是不安全的。在此我要表扬一下前端的同事，他们通过扩展ajax方法原型，实现了和这里类似的东西。所以说，你要多尝试去看看不同领域里的东西，抓住那些相同或者相似的本质，而不是被那些“旧酒换新瓶”的概念所迷惑，技术圈子的热闹有两种，一种是发明新的技术，一种是发明新的概念，我本人更喜欢第一种，你呢？

上传与下载

其实，上传应该是Post请求的一种类型，可是考虑到下载的时候，接口的返回类型应该是数据流，所以我决定将这两个内容一起来讲。这里我们就考虑单纯的上传，不考虑由文件和键值对混合组成的MulitpartFormDataContent，因为这种结构让我觉得厌恶。这里，我们直接通过ASP.NET Core编写了一个文件上传/下载的Service，同样地，我们首先定义IFilesApiCaller接口：

```
[HttpHost("http://localhost:8000")]
public interface IFilesApiCaller : IHttpApiClient
{
    //Post http://localhost:8000/files/upload
    [HttpPost("/files/upload")]
    [OAuth2Filter]
    [JsonReturn]
    ITask<string> Upload([HttpContent]List<MulitpartFile> files);

    //Get http://localhost:8000/files/download/{fileId}
    [HttpGet("/files/download/{fileId}")]
    [OAuth2Filter]
```

```
    ITask<HttpResponseMessage> Download(string fileId);
}
```

在这里，上传我使用了ASP.NET Core中的IFormFile接口，并且在Postman测试通过，可是在网页上用type为file的input标签进行测试时，发现页面一直无法正常响应，不知道具体是什么原因(后来发现它完全和Postman中的请求体一样，好吧��)，我一直不太理解ajax上传和表单上传的区别，曾经项目上用HttpWebRequest去做文件的上传，里面需要大量的字符串拼接动作去构造MulitpartFormData，只要后端上传的API发生变更，这段代码几乎就会变成不可维护的代码，幸运的是，在经过几次迭代以后，他们终于意识到了这个问题，在我的建议下，他们使用HttpClient重构了代码。在这里你会看到Download()方法的返回值类型为ITask，这是HttpClient中使用的数据结构。为什么我推荐大家使用这套API，因为它和ASP.NET中的数据结构是一致的，而事实是上，WebApiClient正是在HttpClient的基础上完成的，所以这里你能够想到，我将通过HttpResponseMessage来获取返回的数据流，进而完成文件的下载。一起来看下面的示例：

```
//调用Files Service
using (var client = HttpApiClient.Create<IFilesApiCaller>())
{
    Console.WriteLine("-----Invoke File Service-----");
    var files = new string[]
    {
        @"C:\Users\PayneQin\Videos\Rec 0001.mp4",
        @"C:\Users\PayneQin\Videos\Rec 0002.mp4",
    }
    .Select(f=>new MulitpartFile(f))
    .ToList();
    var result = await client.Upload(files).InvokeAsync();
    Console.WriteLine(result);

    var json = JArray.Parse(result);
    var fileId = ((JObject)json.First)["fileId"].Value<string>();
    var fileName = Path.Combine(Environment.CurrentDirectory,
"Output/Video001.mp4");
    var filePath = Path.GetDirectoryName(fileName);
    if (!Directory.Exists(filePath)) Directory.CreateDirectory(filePath);
```

```
using (var fileStram = new FileStream(fileName, FileMode.Create))
{
    var stream = await client.Download(fileId).InvokeAsync();
    stream.Content.ReadAsStreamAsync().Result.CopyToAsync(fileStram);
}
```

这里说明的是，非常遗憾，这里的上传接口并没有被成功调用，可能我还是被MulitpartFormDataContent这种东西所困惑着，尽管我使用了WebApiClient中提供的MulitpartFile类，并且使用HttpContent特性对参数进行了修饰。(后来发现是因为我使用JsonReturn特性，可我的Action的确是返回了JSON啊，所以，我不暂时理解不了这一点��)。我了解到的一点信息是，Spring Cloud中的Feign，一个和Retrofit极其相似的HTTP客户端，其本身并没有实现文件上传的功能，需要借助插件来实现相关功能，所以，这是否说明HTTP协议中的上传实现本身就是一个错误，因为它和form-data搅和在一起，试图用键值对的形式去描述一个文件，我们的业务中需要给文件增加备注关联相关信息，坦白讲，这种数据结构令人非常痛苦，所以，上传这块会有三个不同的版本，我一直希望上传可以和具体的业务解耦，即使需要给文件增加备注或者是关联相关信息，应该交给新的Service去做这件事情啊，这简直教人头疼啊。

可配置与动态化

我知道许多人对特性这种”配置“方式并不感冒，因为他们觉得通过配置文件就可以做到不修改代码。我曾经帮助组里写了一个非常简洁的配置方案，后来这个方案在Code Review的时候被拒绝，因为我和别人写得不一样。直到前几天我看到ASP.NET Core里全新的配置方式，我瞬间意识到这种配置方式和我之前的想法不谋而合，这个世界上聪明的人的想法总是如此一致。我相信人们看到这篇文章里出现的各种特性，都会认为像Host、URL等等这些东西都被硬编码了，说得好像你们的代码不需要随着配置文件变化而变化似的，说得好像你们的代码每次都不需要重新编译似的。我曾经考虑到这一点，在开发一个库的时候，充分考虑到了可配置化，事实是大家都不喜欢写配置文件，从那以后，我就变成了坚定的“约定大于配置“主义。

回到WebApiClient这个话题，如果你不喜欢这种基于特性的配置方式，那么你可以通过HttpApiConfig这个类，动态地对诸如Host、URL等参数进行配置，并在WebApiClient创建接口实例的时候传入这些配置。下面是一个简单的示例：

```
//手动创建配置
var config = new HttpApiConfig()
```

```
{
    HttpHost = new Uri("http://www.yourdomain.com"),
};

//调用Values Service
using (var client = HttpApiClient.Create<IValuesApiCaller>(config))
{
    Console.WriteLine("-----Invoke Values Service-----");
    var results = await client.GetValues().InvokeAsync();
    Console.WriteLine($"results is {results}");
    var result = await client.GetValue(10).InvokeAsync();
    Console.WriteLine($"result is {result}");
}
```

我知道杠精们绝对还有话要说，如果我连请求的URL都是动态地该怎么办呢？此时，你总不能让我再让我去配置URL了吧！对于这个问题，WebApiClient提供了Url特性，该特性可以修饰参数，表明这是一个URL，需要注意的是，该参数必须放在第一位，具体可以参考官方文档。

```
[HttpGet]
ITask<string> Login([Url] string url, string username, string password);
```

本文小结

有时候，我会一直在想，前后端分离到底分离的是什么？在我看来，找出这种界限是最重要的，即前端与后端各自的职责是什么。我们想分离的其实是职责，可惜这种想法极其容易演变为前后端人员的分离。而这种人员上的分离，则让接口的设计和沟通充满了坎坷。前后端分离不在于项目是否由两个或者更多的人完成，而在于你是否可以意识到前后端代码里的界限。在这种前提下，博主通过项目上前后端分离的实践经验，配合产品本身的技术架构体系，引申出一个话题，即前端/客户端如何应对后端API快速扩增带来的影响，并由此提出，通过代理类来调用后端API的想法，这一想法借鉴了WebService。接下来，我们介绍了.NET平台下的Retrofit：WebApiClient，它可以让我们以一种“契约式”思想来声明接口，而不必关心这个接口该如何去实现，因为WebApiClient会帮助你实现具体功能。更改接口的代价永远比实现接口要小，所以，我相信这种声明式的HTTP客户端，可以让你更快速地应对来自后端的影响。在Java的世界里有Retrofit、有Feign，为了不被超越 太多，我们只能迎头赶上。