WebAPI 是什么？

WebAPI 是一种用来开发系统间接口、设备接口 API 的技术，基于 Http 协议，请求和返回格式结果默认是 json 格式。比 WCF 更简单、更通用，比 WebService 更节省流量、更简洁。

比如下面的接口 http://sms.rupeng.cn “可以”使用 WebAPI 来开发。

WebAPI 是开发接口的技术，用户不会直接和 WebAPI 打交道，因此 WebAPI 也不会生成界面，这是 WebAPI 和普通 ASP.Net MVC 的区别。

其实完全使用普通 ASP.Net MVC 甚至 HttpHandler 也可以开发这样的接口，但是 WebAPI 是专门做这个的，更专业。

WebAPI 尽可能让开发人员复用 MVC 的路由、ModelBinder、Filter 等知识，但是只是模仿，类名、命名空间等一般都不一样，用法也有一些差别。比如 ASP.net MVC 的 IAuthorizationFilter 如下：

```
using System.Web.Mvc;
namespace WebApplication1
{
    public class AFilter : IAuthorizationFilter
    {
        public void OnAuthorization(AuthorizationContext filterContext)
        {
        }
    }
}
```

ASP.Net Web API 的 IAuthorizationFilter 如下：

```
using System;
using System.Net.Http;
using System.Threading;
using System.Threading.Tasks;
using System.Web.Http.Controllers;
using System.Web.Http.Filters;

namespace WebApplication24
{
    public class AuthorFilter : IAuthorizationFilter
    {
        public bool AllowMultiple => true;

        public Task<HttpResponseMessage> ExecuteAuthorizationFilterAsync(HttpActionContext actionContext, CancellationToken cancellationToken, Func<Task<HttpResponseMessage>> continuation)
        {
            return  continuation();
        }
    }
```

}

因为 ASP.Net WebAPI 专注于接口开发，所有有如下特点：1）webapi 的 Action 方法返回值直接返回对象，专注于数据；2）webapi 更符合 Restful 的风格 3）有利于独立于 IIS 部署（selfhost、winform、windows service、控制台）。4）Action 可以直接声明为 async。

先讲使用.Net Framework 的 WebAPI，最后也演示一下.net core 中的 WebAPI。

什么是 Restful

传统的 Http 接口怎么设计？
删除：发 Post 请求 http://api.rupeng.com/User/Delete/3，返回 json 报文体：{status:'ok'}或者{status:'idnotfound',msg:'id 没找到'}或者{status:'noperm',msg:'没有权限'}
登录：发 Post 请求 http://api.rupeng.com/User/Login，报文体里带用户名、密码，返回和上面类似。
新增用户：发 Post 请求 http://api.rupeng.com/User/AddNew，报文体里带用户信息，返回和上面类似。
查询用户信息：发 Get 请求 http://api.rupeng.com/User/3，返回{status:'idnotfound',msg:'id 没找到'}或者{status:'ok',data:{username:'rupeng',age:5}};

上面这样设计挺好的，没什么不好！

Http 设计之初是有“谓词语义”的，除了 Get、Post 之外，Http 还有 Put、Delete 等请求。获取数据用 Get 请求、删除数据用 Delete 请求，新增用 Post，修改用 Put。用 Http 的返回状态码表示服务器端的处理结果，比如 http://api.rupeng.com/User/3 找不到就返回 404，http://api.rupeng.com/User/Delete/3 如果没有执行权限就返回 401。把所有操作都抽象成对资源的增删改查的操作。

如果基于“Http 谓词语义”进行通讯协议的设计，带来的好处：
1） 可以为不同类型做不同的权限控制；
2） 不再需要“Delete”、“AddNew”这样的 Action 名字，根据请求的类型就可以判断；
3） 返回报文的格式也确定，不用再约定返回状态码，充分利用 Http 状态码；
4） 有利于系统优化，浏览器可以自动缓存 Get 请求
5） Get 没有副作用，是幂等的，可以重试。

由于很多浏览器只支持 Get、Post，所以在 Web 时代，其他请求类型很少应用。接口时代，接口都是普通语言调用了，因此“Http 谓词语义”又得到重用。

如果接口设计按照 Http 谓词语义设计，那么就可以说这个接口是“restful 风格”。Rest 标准。-ful：***的。grace：态度，graceful 温柔

ASP.Net WebAPI 设计之初就是一个 restful 风格的技术，基于这种 Restful 风格写 WebAPI 方法的时候，如果方法叫 Get 则接受 Get 方式访问，如果方法叫 Delete，则接受 Delete 方式访问。但是这个玩意理论性太强，难度较大（比如登录属于什么操作？怎么通过状态码区分“密码错误”和“AppKey 错误”？怎么通过状态码区分“年龄不能大于 99”这个错误？），

不容易掌握，大公司的 API 都很少有满足 Restful 规范的。因此仅供参考，尽可能遵守，但是别追求极端。

下面的是屁话：你那是没有理解 Rest 的真谛，仔细看看“Roy Thomas Fielding 博士”的论文去。做开发又不是搞科研，研究论文才能搞明白的东西是不能用到业务开发中的。

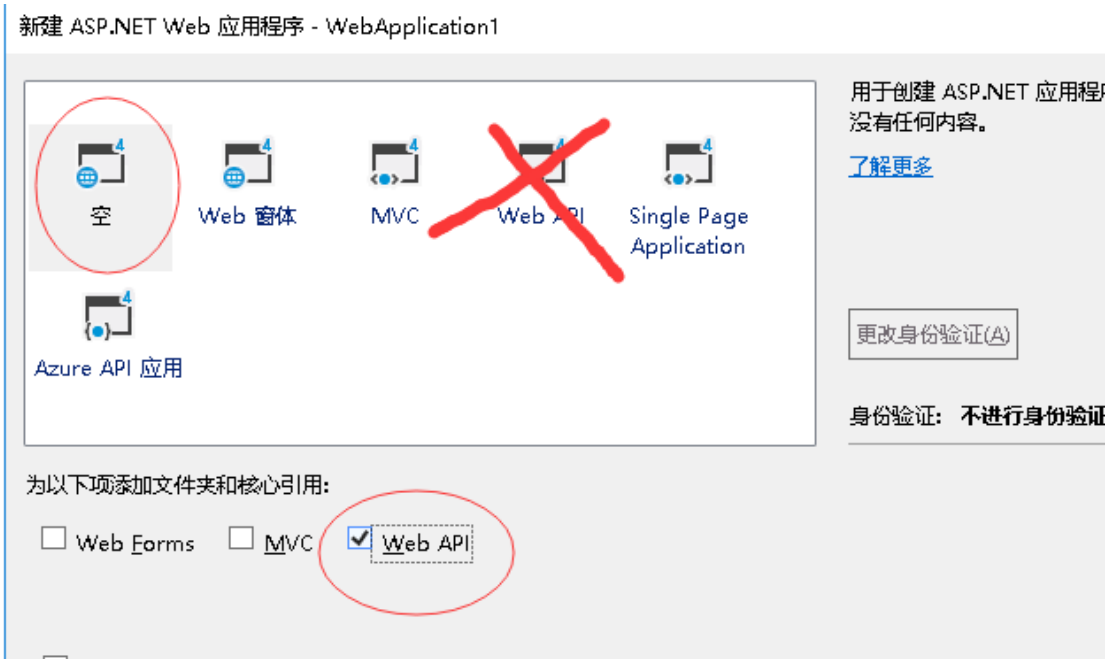
作为一个公司技术掌门人，一定要选择那种“傻 X 都会用的技术”，这样项目才能保质保量按时完成！

WebAPI 入门

1、 创建一个最简单的.net Web 项目

创建一个普通的“ASP.Net Web 应用程序（.Net Framework）”

不建议用 WebAPI 这个模板，因为他会强制使用 MVC



选【空】，然后勾选【Web API】

2、 Controller 下新建一个 PersonController 类，类继承自 ApiController（using System.Web.Http），类内容如下：

```
public IEnumerable<string> Get()//用  /api/Person 访问。不要忘了加“/api”这个前缀
{
    return new string[] { "value1", "value2" };
}

public string Get(int id) //用  /api/Person/3  或者/api/Person?id=3 访问
{
    return "value";
}
```

```
public string Get(string name) //用 /api/Person?name=yzk 访问
{
    return name;
}
```

public string GetByPhoneNum(string phoneNum) 这个可以用 /api/Person? phoneNum =110 访问。不能用/api/Person/ GetByPhoneNum? phoneNum =110 访问，其实这个方法叫什么名字无所谓。如果没有上面的 Get(string name)方法，这个方法叫 Get (string phoneNum)也可以

写一个控制台程序，使用 HttpClient 等发 Http 请求访问这几个接口方法，看一下返回。

分析：

1） WebAPI 的 Controller 从 ApiController 继承

2） WebAPI 的 Action 的返回值一般不用写 ActionResult，而是直接给出返回值数据类型；

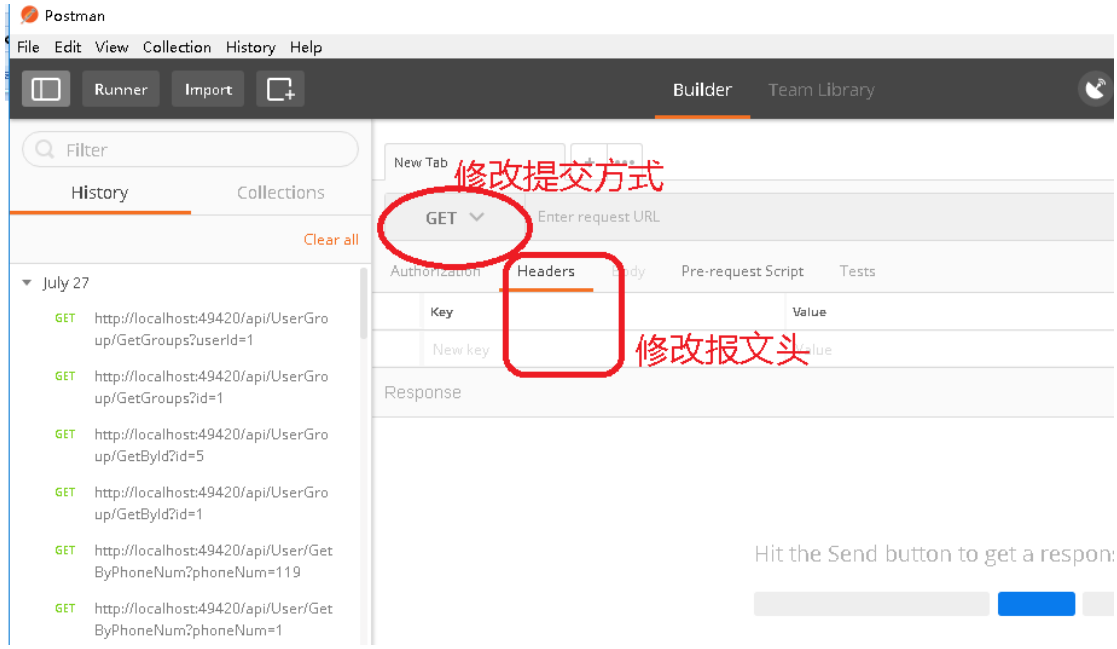
3） 默认，WebAPI 的路径不是/Controller/Action 的方式

4） Action 中如果有调用异步方法，那么只要把 Action 声明成 async、返回值搞成 Task 就可以

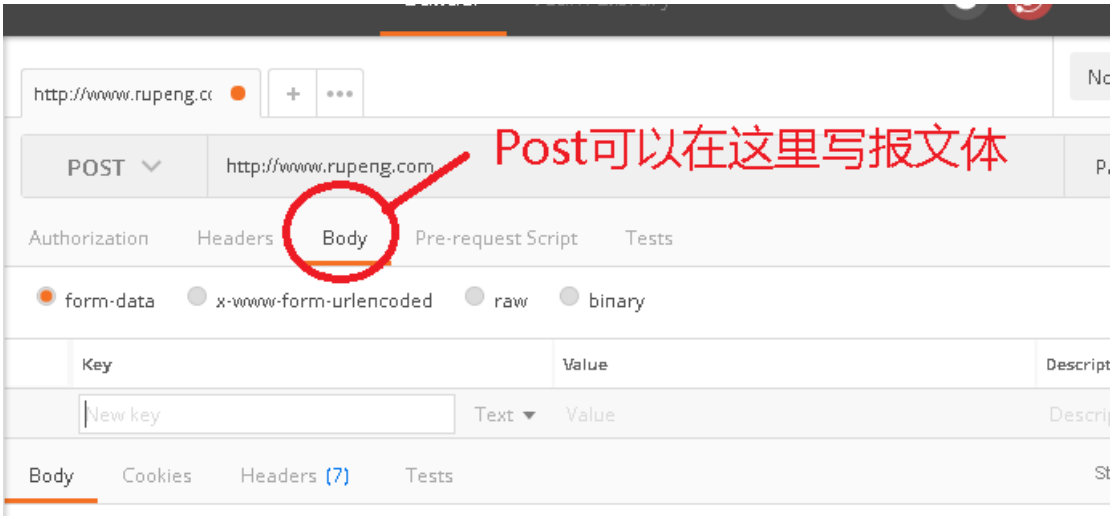
```
public async Task<IEnumerable<string>> Get()
{
    HttpClient hc = new HttpClient();
    string resp = await hc.GetStringAsync("http://www.rupeng.com");
    return new string[] { resp.Length.ToString(), resp.Substring(0,10)};
}
```

3、 开发 WebAPI 接口的时候，如果写程序来发出 Http 接口访问接口进行测试太麻烦。可以借助 Fiddler、PostMan 等工具来帮我们提交 Http 请求。这里介绍使用 PostMan。

PostMan 官网地址：https://www.getpostman.com/



Post 请求



4、 WebAPI 会根据 Http 请求的 Method 访问不同的 Action，给 Controller 增加如下的 Action：

```
// POST api/<controller>
public string Post([FromBody]string value)
{
    return "收到Post，value="+value;
}

// PUT api/<controller>/5
public string Put(int id, [FromBody]string value)
{
    return "收到Put，id="+id+"，value="+value;
}

// DELETE api/<controller>/5
public string Delete(int id)
{
    return "收到Delete，id=" + id;
}
```

访问 Post 方法的时候要使用 Post 请求，报文体是“=hello”，注意不是“value=hello”，[FromBody]表示把报文体中没有 name 的这个值作为值赋值，最多只能有一个[FromBody]参数

访问 Put 方法的时候要使用 Put 请求，请求 http://localhost:57723/api/Person/1 报文体还要是“=hello”

访问 Delete 方法的时候使用 Delete 请求，请求 http://localhost:57723/api/Person/1

由于项目的默认路由是 routeTemplate: "api/{controller}/{id}"，因此 Action 的名字是不起作用的，只是根据请求的类型来找到对应的方法，如果带最后一个/5，还会匹配 id 参数。除此之外再写其他重载方法也不认，这是初学 WebAPI 常遇到的难题。

重要：WebAPI 的路由默认规则和 MVC 不一样，在 WebApiConfig 中，没有{action}，所以不根据 Url 区分方法，而是根据请求方式和参数来对应方法。这种玩意实用价值不大，还容易把

开发者搞晕。因此建议 **routeTemplate 改成"api/{controller}/{action}/{id}"，后续课程我们都这样配置。这样就是遵守 MVC 的 Action 对应规则：Action 方法不能重载，除非通过[HttpPost]、[HttpGet]等来标注，尽量不使用[HttpPost]、[HttpGet]之外的方法。强制要求方法都标注[HttpPost]、[HttpGet]、[HttpDelete]等。**

当然工作中有 Restful 学究使用纯正的 Rest 用 WebAPI，要么忍，要么滚！

Web API 的参数

除了路由之外，WebAPI 还有一个难点，就是参数，跟 MVC 差别很大，特别怪异！

1、 Get 方法的参数。api/User/Login?phoneNum=123&password=123

```
[HttpGet]
public string Login(string phoneNum,string password)
```

这样写就行

如果用模型对象，则必须标注[FromUri]

```
[HttpGet]
public string Login2([FromUri]LoginModel model)
```

2、 Post方法的参数，如果提交的请求体需要是name=rupeng&age=5这样的格式。如果用string AddNew(string phoneNum, string password)这种普通参数会有很多的坑（参考《C#进阶系列——WebApi 接口参数不再困惑：传参详解》），所以不要用。都用模型对象，public string AddNew2(LoginModel model)，也可以参数标注[FromBody]：public string AddNew2([FromBody]LoginModel model)。（只能有一个参数标注FromBody）。如果参数很少，每次都要写一个类，特别麻烦。

3、 如果Post请求提交的数据是ContentType=”application/json”方式提交，那么报文体用{userName:"admin3",password:"123"}这样的json格式提交，那么参数也可以用模型对象。

4、 通过自定义路由规则来捕获参数，

WebAPI可以通过[Route]和[RoutePrefix]来自定义路由，[RoutePrefix]作用于Controller，[Route]作用于Action，一旦一个Controller或者Action设置了[Route]、[RoutePrefix]那么设置的routeTemplate讲在这个Controller或者Action中不起作用。

比如把某个Action标注为

```
[Route("GetById2")]
[HttpGet]
public string GetById(long id)
```

则就要用http://****/GetById2?id=1访问，连http://***/GetById2/1都不行，因为这个也是routeTemplate中定义的。如果要想继续用http://****/api/Person/GetById2?id=1访问，则必须使用[Route("api/Person/GetById2")]

当然每个Action都设定api/Person很麻烦，可以在Controller上标注[RoutePrefix(“api/Person”)]（路径不能以/结尾），则Action主要标注[Route("GetById2")]就可以了。

当然上面只是举例，谁也不会闲的这么搞。[Route]、[RoutePrefix]有两个用途，一个是多版本管理（后面讲），一个就是可以从url捕获参数(无论get、post等都可以)

```
[Route("Login/{phoneNum}/{password}")]
[HttpPost]
public string Login(string phoneNum, string password)
```

然后就可以通过/api/Dog/Login/33/44访问

路由中还是可以进行数据格式限制
https://docs.microsoft.com/en-us/aspnet/web-api/overview/web-api-routing-and-actions/attribute-routing-in-web-api-2 也可以自定义限制规则，还支持可选参数

总结：如果使用 Post 请求，那么就需要每个方法都声明一个参数对应的类，即使只有一个参数，参数前标注[FromBody]，参数少的话麻烦；如果使用 Get 请求，那么参数比较自由，普通参数、类参数都可以，但是需要注意防范缓存（使用 ajax 的话加上一个当前时间或者随机数的参数，使用 HttpClient 等需要禁用缓存）。没有完美。

如果用 Get 方式，参数既可以用普通参数也可以用模型类参数，需要注意防止缓存；如果用 Post 方式，参数一定要用模型类，客户端既可以用 ContentType=”application/x-www-form-urlencoded”提交表单，也可以用 ContentType=”application/json”提交。

Web API 的返回值

1、 Action 的返回值可以是普通类型，这是最常用的。
2、 Action 的返回值可以为 void，这样客户端会得到 204 的状态码，尽量别这样干。
3、 Action 的返回值可以为 IHttpActionResult 类型，可以调用 ApiController 中的 Ok()、NotFound()、Json()、Content()、Redirect()等方法。如果需要用 Http 状态码来表示处理结果，那么可以用这种方法。参考资料：http://www.cnblogs.com/landeanfen/p/5501487.html
4、 HttpResponseMessage 可以做更精细化的返回内容控制，比如返回二进制文件、设置返回报文头

```
[HttpPost]
public HttpResponseMessage Test6()
{
 HttpContent httpContent = new StringContent("hello");
 return new HttpResponseMessage { Content= httpContent ,StatusCode= HttpStatusCode.OK};
}
```

再演示使用 StreamContent 返回文件（接口尽量不要直接返回二进制流）。

WebAPI 的多版本管理

什么是 API 的多版本问题？Android 等 App 存在着多版本客户端共存的问题：App 最新

版已经升级到了 5.0 了，但是有的用户手机上还运行着 4.8、3.9 甚至 2.2 版本的 App，由于早期没有内置升级机制、用户不会升级、用户拒绝升级等原因，造成这些旧版本 App 也在运行。开发新版本 App 的时候，要给接口增加新的功能或者修改以前接口的规范，会造成旧版本 App 无法使用，因此在一定情况下会“保留旧接口的运行、新功能用新接口”，这样就会存在多版本接口共存的问题。

通常的做法是：旧版接口做一个代码分支，除了进行 bug 修改外，旧版本接口不再做改动；新接口代码继续演化升级。在客户端请求的时候带着要请求的接口版本号，在服务器端选择合适的版本代码进行处理。

技术处理方法：

1、（最推荐）不同版本用不同的域名：v1.api.rupeng.com、v2.api.rupeng.com、v3……；

2、在 url、报文头等中带不同的版本信息，用 Nginx 等做反向代理服务器，然后将 http://api.rupeng.com/api/v1/User/1 和 http://api.rupeng.com/api/v2/User/1 转到不同的服务器处理。

3、多个版本的 Controller 共处在一个项目中，然后使用 [RoutePrefix] 或者 IHttpControllerSelector 根据报文头、路径等选择不同的 Controller 执行。下面主要讲这两种方法

处理方法 1：[Route]、[RoutePrefix]

```
[RoutePrefix("api/v1/Person")]
public class PersonController : ApiController
{
    [Route("{id}")]
    public string Get(int id)
    {
        return "我是旧版"+id;
    }
}
[RoutePrefix("api/v2/Person")]
public class PersonV2Controller : ApiController
{
    [Route("{id}")]
    public string Get(int id)
    {
        return "我是V2版" + id;
    }
}
```

处理方法 2（推荐）：自定义 IHttpControllerSelector

1） WebApiConfig 中的路由改成如下：

```
config.Routes.MapHttpRoute(
    name: "DefaultApiv1",
    routeTemplate: "api/v1/{controller}/{id}",
    defaults: new { id = RouteParameter.Optional }
```

```
);
config.Routes.MapHttpRoute(
        name: "DefaultApiv2",
        routeTemplate: "api/v2/{controller}/{id}",
        defaults: new { id = RouteParameter.Optional }
);
```

当然如果还是想用{controller}/{action}的方式，那么改就是了

2） 不同版本的 Controller 放到不同的 namespace 下

```
    namespace WebApplication27.Controllers.v2
    {
        public class PersonController : ApiController
        {
            public string Get(int id)
            {
                return "我是V2版" + id;
            }
        }
    }
    namespace WebApplication27.Controllers.v1
    {
        public class PersonController : ApiController
        {
            public string Get(int id)
            {
                return "我是旧版"+id;
            }
        }
    }
```

3） 编写一个类

```
using System.Collections.Generic;
using System.Linq;
using System.Net.Http;
using System.Text.RegularExpressions;
using System.Web.Http;
using System.Web.Http.Controllers;
using System.Web.Http.Dispatcher;

namespace WebApplication27
{
    public class VersionControllerSelector : DefaultHttpControllerSelector
    {
        private HttpConfiguration _config;
```

```
        public VersionControllerSelector(HttpConfiguration config) : base(config)
        {
            _config = config;
        }

        public override IDictionary<string, HttpControllerDescriptor> GetControllerMapping()
        {
            Dictionary<string, HttpControllerDescriptor> dict
                = new Dictionary<string, HttpControllerDescriptor>();
            foreach (var asm in _config.Services.GetAssembliesResolver().GetAssemblies())
            {
                //获取所有继承自ApiController的非抽象类
                var controllerTypes =asm.GetTypes()
                    .Where(t => !t.IsAbstract &&
typeof(ApiController).IsAssignableFrom(t)).ToArray();
                foreach(var ctrlType in controllerTypes)
                {
                    //从namespace中提取出版本号
                    var match = Regex.Match(ctrlType.Namespace,
                        @"WebApplication27.Controllers.v(\d+)");
                    if(match.Success)
                    {
                        string verNum = match.Groups[1].Value;//获取版本号
                        string ctrlName =
Regex.Match(ctrlType.Name,"(.+)Controller").Groups[1].Value;//从PersonController中拿到Person
                        string key = ctrlName + "v" + verNum;//Personv2为key
                        dict[key] = new HttpControllerDescriptor(_config,ctrlName,ctrlType);
                    }
                }
            }
            return dict;
        }

        //设计就是返回HttpControllerDesriptor的过程
        public override System.Web.Http.Controllers.HttpControllerDescriptor
SelectController(HttpRequestMessage request)
        {
            //获取所有的controller键值集合
            var controllers = GetControllerMapping();
            //获取路由数据
            var routeData = request.GetRouteData();
            //从路由中获取当前controller的名称
            var controllerName = (string)routeData.Values["controller"];
            //从url中获取到版本号
```

```
                string verNum =
Regex.Match(request.RequestUri.PathAndQuery,@"api/v(\d+)").Groups[1].Value;
                string key = controllerName + "v"+verNum;//获取Personv2
                if(controllers.ContainsKey(key))//获取HttpControllerDescriptor
                {
                    return controllers[key];
                }
                else
                {
                    return null;
                }
            }
        }
}
```

4） 在 WebApiConfig 的 Register 中添加

```
config.Services.Replace(typeof(IHttpControllerSelector),
                    new VersionControllerSelector(config));
```

除了通过 Url 来分辨版本之外，还能在报文头中加入版本号、在 queryString 中加入版本号

下面比如通过在请求报文头中加入 ApiVersion 来指定版本

1） WebApiConfig 中的路由改成原始的样子如下

```
config.Routes.MapHttpRoute(
                    name: "DefaultApiv1",
                    routeTemplate: "api/{controller}/{id}",
                    defaults: new { id = RouteParameter.Optional }
                );
```

当然如果还是想用{controller}/{action}的方式，那么改就是了

2） 编写和上面一样的两个 Controller

3） 编写 VersionControllerSelector，其实和上面差不多，只不过改成从报文头读取版本号

只有红色部分和上面的不一样

```
using System.Collections.Generic;
using System.Linq;
using System.Net.Http;
using System.Text.RegularExpressions;
using System.Web.Http;
using System.Web.Http.Controllers;
using System.Web.Http.Dispatcher;

namespace WebApplication27
{
    public class VersionControllerSelector : DefaultHttpControllerSelector
    {
```

```
        private HttpConfiguration _config;

        public VersionControllerSelector(HttpConfiguration config) : base(config)
    {
            _config = config;
        }

        public override IDictionary<string, HttpControllerDescriptor> GetControllerMapping()
        {
            Dictionary<string, HttpControllerDescriptor> dict
                = new Dictionary<string, HttpControllerDescriptor>();
            foreach (var asm in _config.Services.GetAssembliesResolver().GetAssemblies())
            {
                //获取所有继承自ApiController的非抽象类
                var controllerTypes =asm.GetTypes()
                    .Where(t => !t.IsAbstract &&
typeof(ApiController).IsAssignableFrom(t)).ToArray();
                foreach(var ctrlType in controllerTypes)
                {
                    //从namespace中提取出版本号
                    var match = Regex.Match(ctrlType.Namespace,
                        @"WebApplication27.Controllers.v(\d+)");
                    if(match.Success)
                    {
                        string verNum = match.Groups[1].Value;//获取版本号
                        string ctrlName =
Regex.Match(ctrlType.Name,"(.+)Controller").Groups[1].Value;//从PersonController中拿到Person
                        string key = ctrlName + "v" + verNum;//Personv2位key
                        dict[key] = new HttpControllerDescriptor(_config,ctrlName,ctrlType);
                    }
                }
            }
            return dict;
        }

        //设计就是返回HttpControllerDesriptor的过程
        public override System.Web.Http.Controllers.HttpControllerDescriptor
SelectController(HttpRequestMessage request)
        {
            //获取所有的controller键值集合
            var controllers = GetControllerMapping();
            //获取路由数据
            var routeData = request.GetRouteData();
            //从路由中获取当前controller的名称
```

```
            var controllerName = (string)routeData.Values["controller"];
            //从url中获取到版本号
            string verNum = request.Headers.GetValues("ApiVersion").Single();//唯一不一样的

            string key = controllerName + "v"+verNum;//获取Personv2
            if(controllers.ContainsKey(key))//获取HttpControllerDescriptor
            {
                return controllers[key];
            }
            else
            {
                return null;
            }
        }
    }
}
```

5） 还是在 WebApiConfig 的 Register 中添加

```
config.Services.Replace(typeof(IHttpControllerSelector),
                new VersionControllerSelector(config));
```

WebAPI 的 Filter

1、

IAuthorizationFilter，报文头中的 UserName 为 admin 则验证通过

```
public class MyAuthorFilter：IAuthorizationFilter //如果项目也添加了对 MVC 程序集的引用，一定要用 System.Web.Http.Filters 下的 IAuthorizationFilter
{
    public bool AllowMultiple => true;

    public async Task<HttpResponseMessage>ExecuteAuthorizationFilterAsync(HttpActionContext actionContext,
        CancellationToken cancellationToken, Func<Task<HttpResponseMessage>> continuation)
    {
        IEnumerable<string> values;
        if(actionContext.Request.Headers.TryGetValues("UserName", out values))
        {
            string userName = values.FirstOrDefault();
            if (userName != "admin")
            {
```

```
                return new HttpResponseMessage(System.Net.HttpStatusCode.Unauthorized);
            }
        }
        else
        {
            return new HttpResponseMessage(System.Net.HttpStatusCode.Unauthorized);
        }

        return await continuation();
    }
}
```

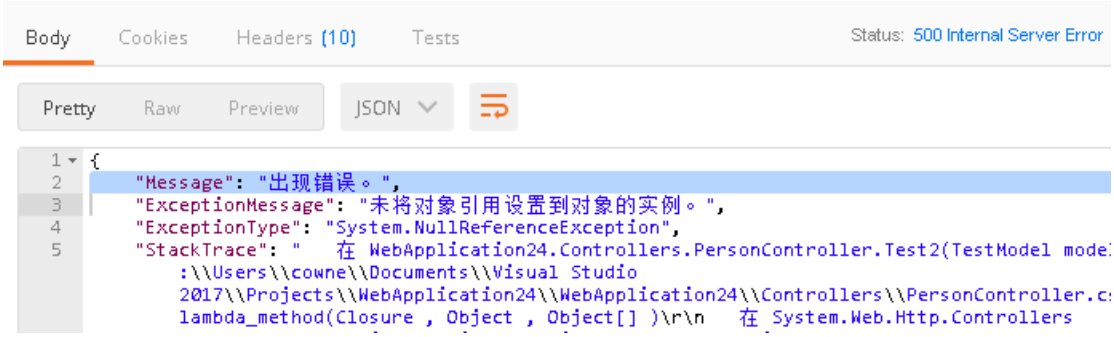
然后 WebApiConfig 中 config.Filters.Add(new MyAuthorFilter());

一旦 ExecuteAuthorizationFilterAsync 内的方法 return await continuation();就是继续向下执行，return new HttpResponseMessage 则是不支持，直接以 HttpResponseMessage 作为执行结果。

2、IActionFilter 用法类似

WebAPI 的异常处理

1、ASP.Net Web API 错误处理：对于 Action 中的异常，默认就是返回 500 状态码，报文体是 Json 格式：



这样也是最好的。

2、对于“Id 不存在”、“年龄不合法”等这类的错误既可以通过自动以状态码的方式返回(不太够用)，也可以自定义下面这种类型作为返回值：

```
public class ApiResult<T>
{
    public int Code { get; set; }
    public string Message { get; set; }
    public T Value { get; set; }
}
public ApiResult<int> Test4()
```

然后在文档中约定：0 代表成功、1 代表用户名不能为空、2 代表金额超限。

3、未处理异常的处理

自定义类实现 IExceptionFilter 接口

```
public class ExceptionFilter : IExceptionFilter
{
    public bool AllowMultiple => false;

    public async Task ExecuteExceptionFilterAsync(HttpActionExecutedContext actionExecutedContext, CancellationToken cancellationToken)
    {
        using(StreamWriter writer = File.AppendText("d:/err.txt"))
        {
            await writer.WriteLineAsync(actionExecutedContext.Exception.ToString());
        }
    }
}
```

实际项目是使用 Log4Net 等组件记录到日志文件。

WebApiConfig 中 config.Filters.Add(new ExceptionFilter());

接口的安全控制

如果你的接口是开放的，谁都可以成功调用，那么会非常危险。因此除非你真的想做开放式服务，否则要对用户的请求做权限控制。

举例，新浪微博开放了微博的接口，所有人可以调用这些接口“发微博”、“看微博”等。当然不是随便调用，而是要到新浪微博的开放平台后台申请一个 AppKey（或者 AppId），这个 AppKey 就是供“如鹏版新浪微博”这个 App 用的，新浪微博的用户也可以用同样的用户名密码登录“如鹏版新浪微博”这个 App 发微博、看微博，和官方版本的新浪微博是互通的。

哪一天新浪发现“如鹏版新浪微博”的开发者用接口干坏事了，比如在用户的每一条发的微博后面都自动加上了如鹏网的广告，那么就可以封杀“如鹏版新浪微博”的 AppKey，这样大家都不能使用“如鹏版新浪微博”了，但是可以换官方版本的微博继续使用。

哪一天新浪发现某个用户大量发违法的微博了，那么就可以封杀这个用户，那么这个用户无论用“如鹏版新浪微博”app 还是“官方版本的微博”app 就都不能发微博了。

所以说通俗一点：AppKey 是控制“如鹏版新浪微博”这个客户端能不能调用接口的，用户名是用来控制微博用户的。

还可以基于 AppKey 进行很多的控制：比如限制一个 AppKey 调用的次数，避免接口被滥用，或者按照调用次数收费等。

如果接口只供自己系统内部使用，可以不做 AppKey 机制，但是建议使用 AppKey 机制，因为也许以后要给内部其他系统调用甚至对外开发，提前做好准备省的以后麻烦。如果接口是暴露给 App 等由客户设备发送请求的终端时，做认证就是特别有必要了，这样即使被人抓包了，那么只要封堵这个 AppKey 就可以了。

无论是认证还是授权，都需要传递认证信息，对于认证来讲就是 AppKey、AppSecret，

对于授权来讲就是用户的账号、密码。

接口传输这些认证信息方法有很多，常用的有：

1） 每次请求，直接把“用户名/AppKey”、“密码/AppSecret”通过表单、QueryString 或者报文头传递；这种安全性比较差，因为随时可以被截获。

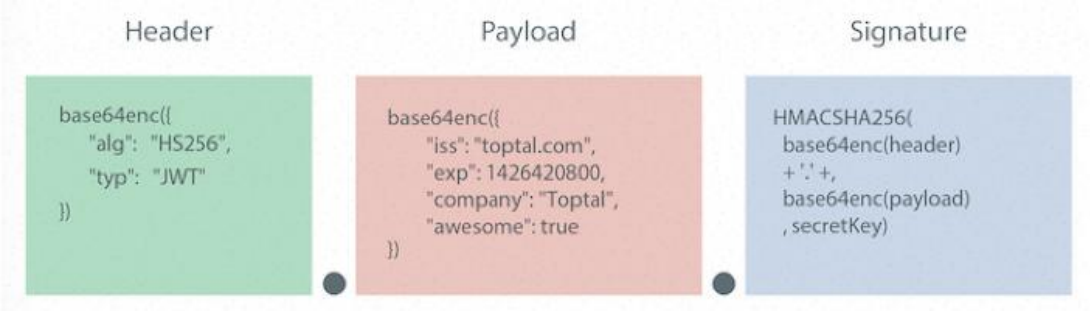
2） 首次先使用“用户名/AppKey”、“密码/AppSecret”获取 Access_Token（相当于 SessionId，在服务器端生成 guid，用 guid 做 key，用用户名做 value 保存到 redis、memcached 等地方），以后的请求都带着 Access_Token，Access_Token 存在有效期，过时后要重新获取 Access_Token。缺点是 Access_Token 有过期重新登录的问题，而移动端 app 经常需要一段时间不用打开还要能直接用。需要有一个类似于 Session 的中心回话服务器。WebAPI 也可以使用 asp.net 的 Session，但是不建议使用。

3） 登录时，服务器端把“用户名/AppKey”、“密码/AppSecret”采用 JWT 等的加密后返回给客户端，客户端以后发送请求的时候把 JWT 加密的内容放到请求中，服务端再解密获取用户名。优点是不需要会话状态服务器，有利于分布式部署，还有可以避免 Session 的过期问题，可以一直能用；缺点是一旦加密解密算法泄露，会带来安全性问题。

JWT 算法

http://www.cnblogs.com/ldybyz/p/6943827.html
http://www.jianshu.com/p/576dbf44b2ae
http://www.jianshu.com/p/180a870a308a

JWT 由三块组成，可以把用户名、用户 Id 等保存到 Payload 部分



注意 Payload 和 Header 部分都是 Base64 编码，可以轻松的 Base64 解码回来。因此 Payload 部分约等于是明文的，因此不能在 Payload 中保存不能让别人看到的机密信息。虽然说 Payload 部分约等于是明文的，但是不用担心 Payload 被篡改，因为 Signature 部分是根据 header+payload+secretKey 进行加密算出来的，如果 Payload 被篡改，就可以根据 Signature 解密时候校验。

1） 加密

```
Install-Package JWT          //需要.net 4.6.2，旧版要求低
var payload = new Dictionary<string, object>
```

```
{
    { "UserId", 123 },
    { "UserName", "admin" }
};
var secret = "GQDstcKsx0NHjPOuXOYg5MbeJ1XT0uFiwDVvVBrk";//不要泄露

IJwtAlgorithm algorithm = new HMACSHA256Algorithm();
IJsonSerializer serializer = new JsonNetSerializer();
IBase64UrlEncoder urlEncoder = new JwtBase64UrlEncoder();
IJwtEncoder encoder = new JwtEncoder(algorithm, serializer, urlEncoder);

var token = encoder.Encode(payload, secret);
Console.WriteLine(token);
```

2） 解密

```
var token = "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJVc2VySWQiOjEyMywiVXNlck5hbWUiOiJhZG1pbiJ9.Qjw1epD5P6p4Yy2yju3-fkq28PddznqRj3ESfALQy_U";
var secret = "GQDstcKsx0NHjPOuXOYg5MbeJ1XT0uFiwDVvVBrk";
try
{
    IJsonSerializer serializer = new JsonNetSerializer();
    IDateTimeProvider provider = new UtcDateTimeProvider();
    IJwtValidator validator = new JwtValidator(serializer, provider);
    IBase64UrlEncoder urlEncoder = new JwtBase64UrlEncoder();
    IJwtDecoder decoder = new JwtDecoder(serializer, validator, urlEncoder);

    var json = decoder.Decode(token, secret, verify: true);
    Console.WriteLine(json);
}
catch (TokenExpiredException)
{
    Console.WriteLine("Token has expired");
}
catch (SignatureVerificationException)
{
    Console.WriteLine("Token has invalid signature");
}
```

试着篡改一下 Payload 部分。

3） 过期时间

在 payload 中增加一个名字为 exp 的值，值为过期时间和 1970/1/1 00:00：00 相差的秒数

```
double exp = (DateTime.UtcNow.AddSeconds(10) - new DateTime(1970, 1, 1)).TotalSeconds;
```

接口安全传输

1） https 可以增加被抓包的难度，所以只要是部署到客户端，必须启用 https
2） 篡改请求，比如用户应该只能查看/user?id=5，但是用户截获请求后修改 id 从而能够查看其它数据。把所有请求参数和 value 按照名字排序后拼接到一起，加上 AppSerect 计算散列值作为 sign，传过去，服务器端再校验一遍。
3） 请求重放，重复的发放请求。每次请求的时候都带着当前时间（时间戳），服务器端比较一下如果这个时间和当前时间相差超过一定时间，则失效。因此最多被重放一段时间，这个要求客户端的时间和服务器端的时间要保持相差不大。

sign、时间戳也是可以通过表单、QueryString 或者报文头等传递

没有绝对安全。终极的安全还是通过业务流程来保证安全，比如后端再次校验权限、重要操作要短信验证等。

项目：用户中心 OpenAPI

开发一个实现了用户、用户组的 OpenAPI，可以查询用户组、获取用户组下用户、管理用户组中的用户、登录检查等，这样各个系统可以共用这个用户中心，实现用户数据集中处理。也是后续项目的一部分。

一、数据库设计：

T_AppInfos（App 表）：

字段名	类型	是否允许为空
Id	bigint	NO
Name	nvarchar(100)	NO
AppKey	nvarchar(100)	NO
AppSecret	nvarchar(100)	NO
CreateDateTime	datetime	NO
IsEnabled	bool	NO

T_Users（用户表）：

字段名	类型	是否允许为空
Id	bigint	NO
PhoneNum	nvarchar(50)	NO
NickName	nvarchar(20)	NO
PasswordHash	nvarchar(100)	NO
PasswordSalt	nvarchar(20)	NO
CreateDateTime	datetime	NO

T_UserGroups（用户组表）

字段名	类型	是否允许为空
Id	bigint	NO
Name	nvarchar(50)	NO
CreateDateTime	datetime	NO

T_GroupUsers（用户-用户组关系表）

字段名	类型	是否允许为空
UserId	bigint	NO
GroupId	bigint	NO

二、项目结构

UserCenter.Services	Service 项目、DbContext、实体类等
UserCenter.IServices	Service 接口
UserCenter.DTO	DTO
UserCenter.OpenAPI	WebAPI 主项目

三、UserCenter.DTO 设计

```
public class AppInfoDTO
{
    public long Id { get; set; }
    public string Name { get; set; }
    public string AppKey { get; set; }
    public string AppSecret { get; set; }
    public bool IsEnabled{get;set;}
}
public class UserDTO
{
    public long Id { get; set; }
    public string PhoneNum { get; set; }
    public string NickName { get; set; }
}
public class UserGroupDTO
{
    public long Id { get; set; }
    public string Name { get; set; }
}
```

四、UserCenter.IServices 接口设计

```
public interface IAppInfoService:IServiceTag
{
    Task<AppInfoDTO> GetByAppKeyAsync(string appKey);
}
public interface IUserGroupService : IServiceTag
{
    Task<UserGroupDTO> GetByIdAsync(long id);
```

```
        Task<UserGroupDTO[]> GetGroupsAsync(long userId);
        Task<UserDTO[]> GetGroupUsersAsync(long userGroupId);
        Task AddUserToGroupAsync(long userGroupId, long userId);
        Task RemoveUserFromGroupAsync(long userGroupId, long userId);
}
public interface IUserService: IServiceTag
{
        Task<long> AddNewAsync(string phoneNum, string nickName, string password);
        Task<bool> UserExistsAsync(string phoneNum);
        Task<bool> CheckLoginAsync(string phoneNum, string password);
        Task<UserDTO> GetByIdAsync(long id);
        Task<UserDTO> GetByPhoneNumAsync(string phoneNum);
}
```

五、UserCenter.Services

开发实体类、开发实体类的 FluentAPI 配置、开发 DbContext

编写服务接口的实现类

从已经写好的项目中拷出来，不再自己写。

项目目前只有 v1 一个版本，但是考虑以后多版本，因此把 Selector、v1 什么都配置好了

六、UserCenter.OpenAPI 基本实现

1、注册 AutoFac

Install-Package Autofac.WebApi2

不要错误的安装 Autofac.Mvc5

也不要错误的安装 Autofac.WebApi，因为 Autofac.WebApi 是给 webapi1 的，否则会报错：重写成员“Autofac.Integration.WebApi.AutofacWebApiDependencyResolver.BeginScope()”时违反了继承安全

然后Global中：using Autofac;using Autofac.Integration.WebApi;

在Global中编写InitAutoFac

```
private void InitAutoFac()
{
        var configuration = GlobalConfiguration.Configuration;
        var builder = new ContainerBuilder();

        // Register API controllers using assembly scanning.
        builder.RegisterApiControllers(Assembly.GetExecutingAssembly()).PropertiesAutowired();
        builder.RegisterWebApiFilterProvider(configuration);

        var services = Assembly.Load("UserCenter.Services");
        builder.RegisterAssemblyTypes(services)
                .Where(type => !type.IsAbstract && typeof(IServiceTag).IsAssignableFrom(type))
                .AsImplementedInterfaces().SingleInstance().PropertiesAutowired();

        var container = builder.Build();
        // Set the WebApi dependency resolver.
```

```
        var resolver = new AutofacWebApiDependencyResolver(container);
        configuration.DependencyResolver = resolver;
}
```

因为 WebApiConfig 中会用到注入，所以在 GlobalConfiguration.Configure(WebApiConfig.Register); 之前调用 InitAutoFac。

2、WebApiConfig 中配置 routeTemplate: "api/{controller}/{action}/{id}"

3、实现 UserController、UserGroupController 基本全是转发这些 Service。全部都按照 Get 实现请求，参数都是普通类型参数。

4、先不开发 AppKey 验证的部分，把接口调好之后再加。使用工具把接口调通。

七、UserCenter.OpenAPI 增加 AppKey 校验

对于请求参数按照名字正序排序，然后用&连接“name=value”，然后加上 AppSecret，然后计算 Md5 值（nuget 包：RuPeng.Common）作为 Sign。

```
    //不要System.Web.Mvc下的IAuthorizationFilter
    //而是用System.Web.Http.Filters下的
```

编写 AuthorizationFilter

```
using RuPeng.Common;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net.Http;
using System.Threading;
using System.Threading.Tasks;
using System.Web.Http.Controllers;
using System.Web.Http.Filters;
using UserCenter.IServices;

namespace UserCenter.OpenAPI.Filters
{
    public class UCAuthorizationFilter : IAuthorizationFilter
    {
        //一个对象必须是IOC容器创建出来的，IOC容器才会自动帮我们注入
        public IAppInfoService appInfoService { get; set; }

        public bool AllowMultiple => true;

        public async Task<HttpResponseMessage> ExecuteAuthorizationFilterAsync(HttpActionContext actionContext, CancellationToken cancellationToken, Func<Task<HttpResponseMessage>> continuation)
        {
            //获得报文头中的AppKey和Sign
            IEnumerable<string> appKeys;
            if(!actionContext.Request.Headers.TryGetValues("AppKey",out appKeys))
```

```
                {
                    return new HttpResponseMessage(System.Net.HttpStatusCode.Unauthorized)
{ Content=new StringContent("报文头中的AppKey为空")};
                }
                IEnumerable<string> signs;
                if (!actionContext.Request.Headers.TryGetValues("Sign", out signs))
                {
                    return new HttpResponseMessage(System.Net.HttpStatusCode.Unauthorized) { Content
= new StringContent("报文头中的Sign为空") };
                }
                string appKey = appKeys.First();
                string sign = signs.First();
                var appInfo = await appInfoService.GetByAppKeyAsync(appKey);
                if(appInfo==null)
                {
                    return new HttpResponseMessage(System.Net.HttpStatusCode.Unauthorized) { Content
= new StringContent("不存在的AppKey") };
                }
                if(!appInfo.IsEnabled)
                {
                    return new HttpResponseMessage(System.Net.HttpStatusCode.Forbidden) { Content =
new StringContent("AppKey已经被封禁") };
                }
                //计算用户输入参数的连接+AppSecret的Md5值
                //orderedQS就是按照key（参数的名字）进行排序的QueryString集合
                var orderedQS = actionContext.Request.GetQueryNameValuePairs().OrderBy(kv =>
kv.Key);
                var segments =    orderedQS.Select(kv => kv.Key + "=" + kv.Value);//拼接key=value的数组
                string qs = string.Join("&", segments);//用&符号拼接起来
                string computedSign = MD5Helper.ComputeMd5(qs + appInfo.AppSecret);//计算qs+secret
的md5值

                //用户传进来md5值和计算出来的比对一下，就知道数据是否有被篡改过
                if(sign.Equals(computedSign,StringComparison.CurrentCultureIgnoreCase))
                {
                    return await continuation();
                }
                else
                {
                    return new HttpResponseMessage(System.Net.HttpStatusCode.Unauthorized) { Content
= new StringContent("sign验证失败") };
                }
            }
        }
```

```
}
```

为了让Ioc容器帮我们注入，我们需要在InitAutoFac中加入：

```
builder.RegisterType(typeof(UCAuthorizationFilter)).PropertiesAutowired();
```

并且需要不是config.Filters.Add(new UCAuthorizationFilter());

而是

```
UCAuthorizationFilter authorFilter =
(UCAuthorizationFilter)GlobalConfiguration.Configuration.DependencyResolver.GetService(typeof(UCAuthorizationFilter));
config.Filters.Add(authorFilter);
```

因为一个对象必须是IOC容器创建出来的，IOC容器才会自动帮我们进行属性或者构造函数参数的注入

服务器部署到 IIS 上，方便测试，否则要开启两个 VS 调试项目。

写接口文档：

网站的接口服务器地址：http://127.0.0.1:8888

地址格式：http://127.0.0.1/api/v1/+具体请求地址

需要把 AppKey 放入请求报文头，请求全部使用 Get 请求

Sign 算法：把 queryString 的键值对，按照 name 升序排序，用&拼接，再加上 AppSecret 计算 32 位 MD5 值。

把 Sign 值也放到请求报文头。

举例：

所有的接口方法都要按照上面的方法传递 AppKey、Sign，否则服务器端会返回 401 错误码。这一点后续不再赘述。

1、 用户管理接口

a) 新增用户，请求：User/AddNew?userName=admin&nickName=tom&password=123，返回{5}表示新增用户的 Id

b) 判断一个手机号是否存在，请求：User/UserExists?phoneNum=18918918189，返回{true}或者{false}

2、

八、开发 SDK 包

编写各个语言的 SDK 包，供各个语言调用，这样省的接口的调用者自己写 Http 请求。

SDKClient 通讯基础类

```
using RuPeng.Common;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net.Http;
using System.Text;
```

```
using System.Threading.Tasks;

namespace UserCenter.NETSDK
{
    class SDKClient
    {
        private string appKey;
        private string appSecret;
        private string serverRoot;//http://127.0.0.1:8888/api/v1/
        public SDKClient(string appKey, string appSecret, string serverRoot)
        {
            this.appKey = appKey;
            this.appSecret = appSecret;
            this.serverRoot = serverRoot;
        }

        /// <summary>
        ///
        /// </summary>
        /// <param name="url">要请求的地址</param>
        /// <param name="queryStringData">querystring参数键值对</param>
        /// <returns></returns>
        public async Task<SDKResult> GetAsync(string url,IDictionary<string,object> queryStringData)
        {
            // var orderedQS =    queryStringData.OrderBy(kv => kv.Key);
            //var qsItems = orderedQS.Select(kv=>kv.Key+"="+kv.Value);
            if(queryStringData==null)
            {
                throw new ArgumentNullException("queryStringData不能为null");
            }
            var qsItems = queryStringData.OrderBy(kv => kv.Key)
                .Select(kv => kv.Key + "=" + kv.Value);
            var queryString = string.Join("&", qsItems);
            string sign = MD5Helper.ComputeMd5(queryString + appSecret);

            using (HttpClient hc = new HttpClient())
            {
                hc.DefaultRequestHeaders.Add("AppKey", appKey);
                hc.DefaultRequestHeaders.Add("Sign", sign);
                var resp = await hc.GetAsync(serverRoot+url + "?" + queryString);
                SDKResult sdkResult = new SDKResult();
                sdkResult.Result = await resp.Content.ReadAsStringAsync();
                sdkResult.StatusCode = resp.StatusCode;
                return sdkResult;
```

```
            }
        }
    }
}
    public class SDKResult
    {
        public string Result { get; set; }
        public HttpStatusCode StatusCode { get; set; }
    }
```

这样再写对 API 方法的进一步封装

九、编写控制台程序测试 SDK

todo：把别的 api 都封装好

.net core 的 webapi

1、 Controller 继承自和 MVC 一样的 Controller；路由配置是，在 Controller 上标注[Route("api/[controller]")]，在方法上标注[HttpGet]、[HttpPost]等，也可以使用[HttpGet("{id}")]这样的格式。可以使用[HttpPost("Test")]这样的方式标注在方法上。这样用 http://localhost:54689/api/values/Test 访问。

2、 Action 支持 IActionResult 做返回值，不支持 HttpResponseMessage 做返回值

3、 不再支持 IHttpControllerSelector，用 IApplicationModelConvention 实现多版本

```
namespace WebApplication32.Controllers.v1
{
    public class DefaultController : Controller
    {
        [HttpGet(nameof(Test))]
        public string Test()
        {
            return "v1";
        }
    }
}
namespace WebApplication32.Controllers.v2
{
    public class DefaultController : Controller
    {
        [HttpGet(nameof(Test))]
        public string Test()
        {
            return "v2";
        }
    }
```

```
}
```

编写 NameSpaceVersionRoutingConvention

```
using System.Linq;
using System.Text.RegularExpressions;
using Microsoft.AspNetCore.Mvc.ApplicationModels;

namespace WebApplication32
{
    public class NameSpaceVersionRoutingConvention : IApplicationModelConvention
    {
        private readonly string apiPrefix;
        private const string urlTemplate = "{0}/{1}/{2}";
        public NameSpaceVersionRoutingConvention(string apiPrefix = "api")
        {
            this.apiPrefix = apiPrefix;
        }

        public void Apply(ApplicationModel application)
        {
            foreach (var controller in application.Controllers)
            {
                //如果Controller标注了[Route]，则不这个不起作用
                var hasRouteAttribute = controller.Selectors
                    .Any(x => x.AttributeRouteModel != null);
                if (hasRouteAttribute)
                {
                    continue;
                }
                var nameSpaces = controller.ControllerType.Namespace.Split('.');
                //获取namespace中版本号部分
                var version = nameSpaces.FirstOrDefault(x => Regex.IsMatch(x, @"^v(\d+)$"));
                if (string.IsNullOrEmpty(version))
                {
                    continue;
                }
                string template = string.Format(urlTemplate, apiPrefix, version,
controller.ControllerName);
                controller.Selectors[0].AttributeRouteModel = new AttributeRouteModel()
                {
                    Template = template
                };
            }
        }
    }
```

```
}
```

下面是老师上课写的版本：

```
using Microsoft.AspNetCore.Mvc.ApplicationModels;
using System.Linq;
using System.Text.RegularExpressions;

namespace CoreWebAPITest1
{
    public class NameSpaceVersionRoutingConvention : IApplicationModelConvention
    {
        private string prefix;

        public NameSpaceVersionRoutingConvention(string prefix)
        {
            this.prefix = prefix;
        }

        public void Apply(ApplicationModel application)
        {
            var controllers = application.Controllers;
            foreach(var controller in controllers)
            {
                //判断Controller上是否标注了[Route]
                var hasRouteAttribute =
                    controller.Selectors.Any(s=>s.AttributeRouteModel!=null);
                //如果标注了，则不由我处理
                if(hasRouteAttribute)
                {
                    continue;
                }
                var matchVer = Regex.Match(controller.ControllerType.Namespace,
                    @".v(\d+)");
                if(!matchVer.Success)
                {
                    continue;
                }
                string verNum = matchVer.Groups[1].Value;//1
                //计算这个Controller对应的路由路径
                string template = this.prefix + "/v" + verNum + "/" + controller.ControllerName;
                controller.Selectors[0].AttributeRouteModel = new AttributeRouteModel()
{ Template=template};
            }
        }
    }
```

```
}
```

Startup 的 ConfigureServices 中

```
services.AddMvc(opts=> {
    opts.Conventions.Insert(0, new NameSpaceVersionRoutingConvention());
});
```

然后访问 http://localhost:57799/api/v1/Default/Test、http://localhost:57799/api/v2/Default/Test

经过调试发现，NameSpaceVersionRoutingConvention 只是在程序启动的时候执行一遍，不像 IHttpControllerSeelct 一样，不需要每次请求访问的时候都运行一遍，因此效率更高！

4、 Filter 和 asp.net mvc core 共享，Microsoft.AspNetCore.Mvc.Filters 下有同步异步两种接口。建议用异步的。然后如下注册

```
services.AddMvc(opts=> {
    opts.Filters.Add(new AuthorFilter());
});
```