1 过滤器

过滤器的接口是IApiActionFilterAttribute，WebApiClient提供默认ApiActionFilterAttribute抽象基类，比从IApiActionFilterAttribute实现一个过滤器要简单得多。

1.1 TraceFilterAttribute

这是一个用于调试追踪的过滤器，可以将请求与响应内容写入统一日志，统一日志工厂需要在HttpApiConfig的LoggerFactory配置。

接口或方法使用[TraceFilter]

```
[TraceFilter]
public interface IUserApi : IHttpApi
{
    // GET {url}?account={account}&password={password}&something={something}
    [HttpGet]
    [Timeout(10 * 1000)] // 10s超时
    Task<string> GetAboutAsync(
        [Url] string url,
        UserInfo user,
        string something);
}
```

配置统一日志工厂

```
HttpApiFactory.Add<IUserApi>().ConfigureHttpApiConfig(c =>
{
    c.LoggerFactory = new LoggerFactory().AddConsole();
});
```

请求之后输出请求信息

```
info: IUserApi.GetAboutAsync[0]
      [REQUEST] 2018-10-08 23:55:25.775
      GET /webapi/user/about?Account=laojiu&password=123456&BirthDay=2018-01-
01&Gender=1&something=somevalue HTTP/1.1
      Host: localhost:9999

      [RESPONSE] 2018-10-08 23:55:27.047
      This is from NetworkSocket.Http

      [TIMESPAN] 00:00:01.2722715
```

1.2 自定义过滤器

```
[SignFilter]
public interface IUserApi : IHttpApi
{
    ...
}
```

```
class SignFilter : ApiActionFilterAttribute
{
    public override Task OnBeginRequestAsync(ApiActionContext context)
    {
        var sign = DateTime.Now.Ticks.ToString();
        context.RequestMessage.AddUrlQuery("sign", sign);
        return base.OnBeginRequestAsync(context);
    }
}
```

当我们需要为每个请求的url额外的动态添加一个叫sign的参数，这个sign可能和配置文件等有关系，而且每次都需要计算，就可以如上设计与应用一个SignFilter。

2 全局过滤器

全局过滤器的执行优先级比非全局过滤器的要高，且影响全部的请求方法，其要求实现IApiActionFilter接口，并实例化添加到HttpApiConfig的GlobalFilters。像[TraceFilter]等一般过滤器，也是实现了IApiActionFilter接口，也可以添加到GlobalFilters作为全局过滤器。

2.1 自定义全局过滤器

```
class MyGlobalFilter : IApiActionFilter
{
    public Task OnBeginRequestAsync(ApiActionContext context)
    {
        // do something
        return Task.CompletedTask;
    }

    public Task OnEndRequestAsync(ApiActionContext context)
    {
        // do something
        return Task.CompletedTask;
    }
}
```

添加到GlobalFilters

```
HttpApiFactory.Add<IUserApi>().ConfigureHttpApiConfig(c =>
{
    c.GlobalFilters.Add(new MyGlobalFilter());
});
```

2.2 自定义OAuth2全局过滤器

```
/// <summary>
/// 表示提供client_credentials方式的token过滤器
/// </summary>
public class TokenFilter : AuthTokenFilter
{
    /// <summary>
    /// 获取提供Token获取的Url节点
    /// </summary>
    public string TokenEndpoint { get; private set; }

    /// <summary>
    /// 获取client_id
    /// </summary>
    public string ClientId { get; private set; }

    /// <summary>
    /// 获取client_secret
    /// </summary>
    public string ClientSecret { get; private set; }

    /// <summary>
    /// OAuth授权的token过滤器
    /// </summary>
    /// <param name="tokenEndPoint">提供Token获取的Url节点</param>
    /// <param name="client_id">客户端id</param>
    /// <param name="client_secret">客户端密码</param>
    public TokenFilter(string tokenEndPoint, string client_id, string client_secret)
    {
        this.TokenEndpoint = tokenEndPoint ?? throw new
ArgumentNullException(nameof(tokenEndPoint));
        this.ClientId = client_id ?? throw new ArgumentNullException(nameof(client_id));
        this.ClientSecret = client_secret ?? throw new
ArgumentNullException(nameof(client_secret));
    }

    /// <summary>
    /// 请求获取token
    /// 可以使用TokenClient来请求
    /// </summary>
    /// <returns></returns>
    protected override async Task<TokenResult> RequestTokenResultAsync()
    {
        var tokenClient = new TokenClient(this.TokenEndpoint);
        return await tokenClient.RequestClientCredentialsAsync(this.ClientId,
this.ClientSecret);
    }

    /// <summary>
    /// 请求刷新token
    /// 可以使用TokenClient来刷新
    /// </summary>
    /// <param name="refresh_token">获取token时返回的refresh_token</param>
    /// <returns></returns>
    protected override async Task<TokenResult> RequestRefreshTokenAsync(string
refresh_token)
    {
        var tokenClient = new TokenClient(this.TokenEndpoint);
        return await tokenClient.RequestRefreshTokenAsync(this.ClientId, this.ClientSecret,
refresh_token);
    }
}
```

添加到GlobalFilters

```
HttpApiFactory.Add<IUserApi>().ConfigureHttpApiConfig(c =>
{
    var tokenFilter = new TokenFilter ("http://localhost/tokenEndpoint","client","secret");
    c.GlobalFilters.Add(tokenFilter);
});
```

3. 自定义特性

WebApiClient内置很多特性，包含接口级、方法级、参数级的，他们分别是实现了IApiActionAttribute接口、IApiActionFilterAttribute接口、IApiParameterAttribute接口、IApiParameterable接口和IApiReturnAttribute接口的一个或多个接口。

3.1 自定义IApiParameterAttribute

例如服务端要求使用x-www-form-urlencoded提交，由于接口设计不合理，目前要求是提交：fieldX= {X}的json文本&fieldY={Y}的json文本 这里{X}和{Y}都是一个多字段的Model，我们对应的接口是这样设计的：

```
[HttpHost("/upload")]
ITask<bool> UploadAsync(
     [FormField][AliasAs("fieldX")] string xJson,
     [FormField][AliasAs("fieldY")] string yJson);
```

显然，我们接口参数为string类型的范围太广，没有约束性，我们希望是这样子

```
[HttpHost("/upload")]
ITask<bool> UploadAsync([FormFieldJson] X fieldX, [FormFieldJson] Y fieldY);
```

[FormFieldJson]将参数值序列化为Json并做为表单的一个字段内容

```
[AttributeUsage(AttributeTargets.Parameter, AllowMultiple = false)]
class FormFieldJson: Attribute, IApiParameterAttribute
{
    public async Task BeforeRequestAsync(ApiActionContext context,
ApiParameterDescriptor parameter)
    {
        var options = context.HttpApiConfig.FormatOptions;
        var json = context.HttpApiConfig.JsonFormatter.Serialize(parameter.Value, options);
        var fieldName = parameter.Name;
        await context.RequestMessage.AddFormFieldAsync(fieldName, json);
    }
}
```

4. 异常处理和重试策略

4.1 try catch异常处理

```
try
{
    var user = await userApi.GetByIdAsync("id001");
    ...
}
catch (HttpStatusFailureException ex)
{
    var error = ex.ReadAsAsync<ErrorModel>();
    ...
}
catch (HttpRequestException ex)
{
    ...
}
```

4.2 Retry重试策略

```
try
{
    var user1 = await userApi
        .GetByIdAsync("id001")
        .Retry(3, i => TimeSpan.FromSeconds(i))
        .WhenCatch<HttpStatusFailureException>();
    ...
}
catch (HttpStatusFailureException ex)
{
    var error = ex.ReadAsAsync<ErrorModel>();
    ...
}
catch (HttpRequestException ex)
{
    ...
}
catch(Exception ex)
{
    ...
}
```

4.3 RX扩展

在一些场景中，你可能不需要使用async/await异步编程方式，WebApiClient提供了Task对象转换为IObservable对象的扩展，使用方式如下：

```
var unSubscriber = userApi.GetByIdAsync("id001")
   .Retry(3, i => TimeSpan.FromSeconds(i))
   .WhenCatch<HttpStatusFailureException>();
   .ToObservable().Subscribe(result =>
   {
       ...
   }, ex =>
   {
        ...
   });
```