**使用 HttpMessageHandler 实现 HttpClient 请求管道自定义**

2021-04-28

阅读时长: 10 分钟

最近，博主偶然间在 [博客园](https://www.cnblogs.com/) 看到一篇文章：[ASP.NET Core 扩展库之 Http 请求模拟](https://www.cnblogs.com/xfrog/p/14703251.html" \t "_blank)，它里面介绍了一种利用 [HttpMessageHandler](https://docs.microsoft.com/zh-cn/dotnet/api/system.net.http.httpmessagehandler?view=net-5.0" \t "_blank) 来实现 Http 请求模拟的方案。在日常工作中，我们总是不可避免地要和第三方的服务或者接口打交道，尤其是当我们需要面对“**联调**”这样一件事情的时候。通常，我们可以通过类似 [YAPI](https://github.com/ymfe/yapi) 这样的工具来对尚在开发中的接口进行模拟。可是，因为这种方式会让我们的测试代码依赖于一个外部工具，所以，从严格意义上讲，它其实应该属于“**集成测试**”的范畴。在接触前端开发的过程中，对于其中的 [Mock.js](http://mockjs.com/) 印象深刻。故而，当看到 .NET 中有类似实现的时候，好奇心驱使我对其中的核心，即 HttpMessageHandler 产生了浓厚的兴趣。平时，我们更多的是使用 [Moq](https://github.com/moq/moq4" \t "_blank) 这样的库来模拟某一个对象的行为，而对一个 Http 请求进行模拟，可以说是开天辟地头一遭。带着这些问题出发，就有了今天这篇博客，通过 HttpMessageHandler 实现 HttpClient 请求管道的自定义。

**什么是 HttpMessageHandler？**

相信大家读过我提到的文章以后，都能找到这里面最核心的一个点：HttpMessageHandler。于是，我们今天要面对的第一个问题就是，什么是 HttpMessageHandler？此时，我们需要一张历久弥新的示意图，来自 [微软官方](https://www.asp.net/media/4071077/aspnet-web-api-poster.pdf)。这里，我们重点关注的是 DelegatingHandler，它继承自 HttpMessageHandler。通过这张图，我们能够获得哪些信息呢？

我认为，主要有以下几点：**第一，HttpMessageHandler 处于整个 Http 请求管道的第一梯队，每一个路由匹配的请求都会从这里“进入”和“离开”；第二，HttpMessageHandler 可以是全局配置或者针对某个特定的路由，只要这个路由被匹配到就会执行；第三，HttpMessageHandler 可以直接构造 Http 响应并且返回，跳过剩余的管道流程**。不知道大家看到这里会想到什么？坦白讲，我联想到了.NET Core 中的中间件，而唯一不同的地方或许是，中间件是 ASP.NET Core 里的概念，这里则是 ASP.NET Web API 里的概念。尤其是第三点，它对于我们的意义非常重大，因为它，我们才可以做到对一个 Http 请求进行模拟。

[Graphical user interface

Description automatically generated](https://i.loli.net/2021/04/28/AwLZDdqXc5KERky.png)HttpMessageHandler 与 ASP.NET Web API

而事实上，在 ASP.NET Web API 的设计中，它是由一组 HttpMessageHandler 经过“首尾相连”而成，这种管道式的设计使得框架本身具有很高的扩展性。虽然，作为一个服务端框架，ASP.NET Web API 最主要的作用是就是“**处理请求、响应回复**”，可具体采用的处理策略会因具体场景的不同而不同。所以，管道式设计的本质，就是让某一个 Handler 只负责某个单一的消息处理功能，在根据具体场景的不同，选择需要的 Handler 并将其串联成一个完整的消息处理通道。而在这里，这个负责单一的消息处理功能的 Handler 其实就是 HttpMessageHandler，因为它不单单可以对请求消息(**HttpRequestMessage**)进行处理，同时还可以对响应消息(**HttpResponseMessage**)进行处理。此时，我们就不难理解 HttpMessageHandler 的定义：

public abstract class HttpMessageHandler : IDisposable

{

protected HttpMessageHandler();

public void Dispose();

protected virtual void Dispose(bool disposing);

protected internal virtual HttpResponseMessage Send(

HttpRequestMessage request,

CancellationToken cancellationToken

);

protected internal abstract Task<HttpResponseMessage> SendAsync(

HttpRequestMessage request,

CancellationToken cancellationToken

);

}

Copy

也许，你会忍不住问这样一个问题：DelegatingHandler 和 HttpMessageHandler 的区别是什么？ 其实，只要你稍微仔细一点，你就会发现，两者最大的区别是 DelegatingHandler 里新增一个叫做 InnerHandler 的成员，它本身就是一个 HttpMessageHandler。所以，聪明的你又联想到什么呢？我想，或许是一个叫做 RequestDelegate 的委托，还记得我们写中间件是一直都少不了的 Next 吗？不得不说，这里越来越有中间件的味道了。你可以立马想到的一件事情是，除了最后一个 Handler 是 HttpMessageHandler 以外，剩下的前面的所有的 Handler 都是 DelegatingHandler。为什么这样说呢？因为前面的 n-1 个 Handler 都需要串联下一个 Handler，只有第 n 个 Handler可以允许短路，所以，大概就相当于 Use() 和 Run() 的区别？

public abstract class DelegatingHandler : HttpMessageHandler

{

protected DelegatingHandler();

protected DelegatingHandler(HttpMessageHandler innerHandler);

// InnerHandler是实现管道式设计的关键

public HttpMessageHandler? InnerHandler { get; set; }

protected override void Dispose(bool disposing);

protected internal override HttpResponseMessage Send(

HttpRequestMessage request,

CancellationToken cancellationToken

);

protected internal override Task<HttpResponseMessage> SendAsync(

HttpRequestMessage request,

CancellationToken cancellationToken

);

}

Copy

所以，此时此刻，你能否为 HttpMessageHandler 下一个清晰的定义呢？我想，或许可以这样理解，一种可以对 请求消息(**HttpRequestMessage**) 和 响应消息(**HttpResponseMessage**) 进行处理，同时多个 HttpMessageHandler 可以组成一个完整的消息处理通道的中间件。屏幕前的你又是如何理解的呢？欢迎大家在评论区留言，留下你对于 HttpMessageHandler 的想法或者认识。

**实现自定义请求管道**

好了，搞清楚 HttpMessageHandler 是什么以后，我们就可以考虑自定义请求管道的实现啦！让我们从一个最简单的示例开始，假设我们这里定义了两个自定义的 Handler，它们分别是： HandlerA 和 HandlerB，我们应该如何将其应用到具体的 HttpClient上呢？

// Handler A

public class HandlerA : DelegatingHandler

{

private readonly ILogger<HandlerA> \_logger;

public HandlerA(ILogger<HandlerA> logger) { \_logger = logger; }

protected override Task<HttpResponseMessage> SendAsync(

HttpRequestMessage request,

CancellationToken cancellationToken

)

{

\_logger.LogInformation("This is Handler A");

return base.SendAsync(request, cancellationToken);

}

}

// Handler B

public class HandlerB : DelegatingHandler

{

private readonly ILogger<HandlerB> \_logger;

public HandlerB(ILogger<HandlerB> logger) { \_logger = logger; }

protected override Task<HttpResponseMessage> SendAsync(

HttpRequestMessage request,

CancellationToken cancellationToken

)

{

\_logger.LogInformation("This is Handler B");

return base.SendAsync(request, cancellationToken);

}

}

Copy

这里，我们考虑两种场景，依赖注入 和 非依赖注入。对于依赖注入的场景，我们只需要调用AddHttpMessageHandler()方法按顺序注册即可，不需要处理InnerHandler，这里遵循先注册后使用的原则；对于非依赖注入的场景，需要处理InnerHandler，并在构造HttpClient的时候作为参数传入。

// 依赖注入

var services = new ServiceCollection();

services.AddTransient<HandlerA>();

services.AddTransient<HandlerB>();

services.AddHttpClient("MyClient", options => {

options.BaseAddress = new Uri("https://blog.yuanpei.me/");

})

.AddHttpMessageHandler<HandlerA>()

.AddHttpMessageHandler<HandlerB>();

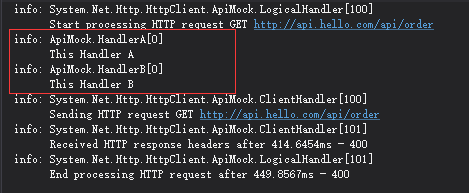
// 非依赖注入

var handler = new HandlerA() { InnerHandler = new HandlerB() };

var client = new HttpClient(handler)

Copy

此时，我们就可以得到下面的结果，可以注意到的是，两个Handler的执行顺序与注册顺序一致：

[](https://i.loli.net/2021/04/29/URNWavrVgyzMAxe.png)Handler执行顺序与注册顺序

好了，热身环节到此结束！下面，我们来开始实战，这里展示的是 HttpMessageHandler 在日志记录、请求重试 和 接口模拟等方面的应用。

**日志记录**

对于 Http 请求的日志，我们希望记录请求的Url、Http动词、请求时长等信息，而这一点，在一个大量接入第三方接口的系统或者是以 Http 驱动的微服务架构中，常常是不可或缺的一环，对于我们排查故障、监控服务非常有用。

protected override async Task<HttpResponseMessage> SendAsync(HttpRequestMessage request, CancellationToken cancellationToken)

{

var correlationId = GetCorrelationId(request);

using (\_logger.BeginScope($"correlationId={correlationId}"))

{

var sw = Stopwatch.StartNew();

\_logger.LogInformation($"Start Processing HTTP Request {request.Method} {request.RequestUri} [Correlation: {correlationId}]");

var response = base.Send(request, cancellationToken);

\_logger.LogInformation($"End Processing HTTP Request in {sw.ElapsedMilliseconds}ms {response.StatusCode}, [Correlation: {correlationId}]");

return response;

}

}

// GetCorrelationId

private string GetCorrelationId(HttpRequestMessage request)

{

if (request.Headers.TryGetValues("X-Correlation-ID", out var values))

return values.First();

var correlationId = Guid.NewGuid().ToString();

request.Headers.Add("X-Correlation-ID", correlationId);

return correlationId;

}

Copy

此时，我们可以得到下面的结果：

[Text

Description automatically generated](https://i.loli.net/2021/04/29/aORFS3ZQEw8pNbT.png)HttpMessageHandler 实现日志记录

**请求重试**

我们知道，一个系统中接入的外部因素越多，则整个系统的稳定性越低。而国内的产品通常都喜欢"大而全"的"万物互联"，所以，最实际的问题，其实就是调用一个第三方的接口，如何保证其可靠性。所以，考虑请求的故障恢复就显得非常有意义，为此，我们可以引入Polly，在实现SendAsync()方法的时候，通过Polly中的超时、重试等机制对其做一层包装：

public class RetryableHttpMessageHandler : DelegatingHandler

{

private readonly ILogger<RetryableHttpMessageHandler> \_logger;

private readonly IAsyncPolicy<HttpResponseMessage> \_retryPolicy;

public RetryableHttpMessageHandler(

ILogger<RetryableHttpMessageHandler> logger

)

{

\_logger = logger;

\_retryPolicy = Policy<HttpResponseMessage>

.Handle<HttpRequestException>()

.Or<TimeoutException>()

.OrResult(x => (int)x.StatusCode >= 400)

.RetryAsync(3, (ret, index) =>

{

\_logger.LogInformation($"调用接口异常：{ret.Exception?.Message}，状态码：{ret.Result.StatusCode}, 正在进行第{index}次重试");

});

}

protected override Task<HttpResponseMessage> SendAsync(

HttpRequestMessage request,

CancellationToken cancellationToken

)

{

return \_retryPolicy.ExecuteAsync(() => base.SendAsync(request, cancellationToken));

}

}

Copy

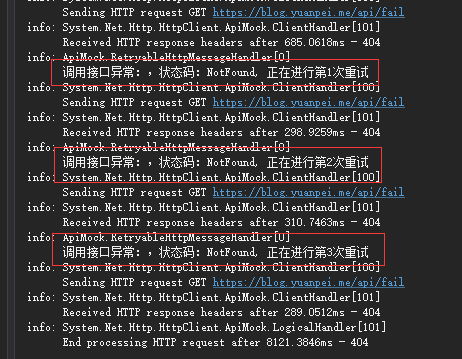
同样地，我们这里通过HttpClient来请求指定的接口。因为，下面的接口实际上是不存在的。所以，理论上它会返回404这个状态码。而我们的重试策略是，在发生HttpRequestException或者TimeoutException异常以及 Http 响应的状态码大于 400 时，自动触发 3 次重试。

var client = \_clientFactory.CreateClient("ApiMock");

var response = await client.GetAsync("/api/fail");

Copy

此时，我们可以得到下面的结果：

[](https://i.loli.net/2021/04/30/OaUyhNF7XYsA8mI.png)HttpMessageHandler 实现请求重试

可以发现，不多不少刚好是 3 次。除了重试以外，Polly还支持类似超时、断路器等等不同的策略，甚至可以将它们组合起来使用，这些都属于[Polly](https://github.com/App-vNext/Polly" \t "_blank)的内容，不作为本文的重点内容来讲解，感兴趣的朋友可以查阅这篇文章：[.NET 开源项目 Polly 介绍](https://www.cnblogs.com/willick/p/polly.html" \t "_blank)。需要说明的是，微软官方提供的 Microsoft.Extensions.Http.Polly，它在IHttpClientBuilder上添加了一个名为AddPolicyHandler()的扩展方法，这里的例子可以被简化为下面这样，它和我们这里举的例子是完全一致的：

// 定义重试策略

var retryPolicy = Policy<HttpResponseMessage>

.Handle<HttpRequestException>()

.Or<TimeoutException>()

.OrResult(x => (int)x.StatusCode >= 400)

.RetryAsync(3, (ret, index) =>

{

Console.WriteLine($"调用接口异常：{ret.Exception?.Message}，状态码：{ret.Result.StatusCode}, 正在进行第{index}次重试");

});

// 注册HttpClient并指定重试策略

services.AddHttpClient("ApiMock", options => {

options.BaseAddress = new Uri("https://blog.yuanpei.me");

})

.AddPolicyHandler(retryPolicy);

Copy

**接口模拟**

在集成第三方接口时，在双方确定好接口以后，接口消费方会有一段时间的“黒写”时期。因为在接口提供方的接口没有正式提供前，接口消费方始终只能通过“**模拟**”的方式来进行测试。考虑到单元测试对 [YAPI](https://github.com/ymfe/yapi" \t "_blank) 存在耦合，所以，接口模拟同样是一件意义非凡的事情。这里的思路是利用 HttpMessageHandler 的“**短路**”功能，即构造一个 HttpResponseMessage 并返回。

首先，我们定义一个MockItem类型，它含有两个委托类型的属性RouteSelector和Executor。其中，前者用来匹配路由，而后者则用来处理接口返回值。

public class MockItem

{

public Func<HttpRequestMessage, bool> RouteSelector { get; set; }

public Func<HttpRequestMessage, HttpResponseMessage, Task> Executor { get; set; }

}

Copy

接下来，我们需要定义相应的Handler，这里是ApiMockHttpMessageHandler：

public class ApiMockHttpMessageHandler: DelegatingHandler

{

private readonly ILogger<ApiMockHttpMessageHandler> \_logger;

private readonly IEnumerable<MockItem> \_routes;

public ApiMockHttpMessageHandler(

ILogger<ApiMockHttpMessageHandler> logger,

IEnumerable<MockItem> routes)

{

\_logger = logger;

\_routes = routes;

}

protected override async Task<HttpResponseMessage> SendAsync(

HttpRequestMessage request,

CancellationToken cancellationToken

)

{

// 匹配路由并调用其Executor属性

var route = \_routes.FirstOrDefault(x => x.RouteSelector?.Invoke(request));

if (route != null)

{

var response = new HttpResponseMessage();

await route.Executor?.Invoke(request, response);

return response;

}

return base.Send(request, cancellationToken);

}

}

Copy

我们的思路是，对于所有注入到Ioc容器中的MockItem，检查其路由是否匹配，如果路由匹配，则通过其指定的Executor对HttpResponseMessage进行加工并返回。为了更加方便地在Ioc容器中进行注入，我们为IServiceCollection编写了相应的扩展方法：

public static IServiceCollection AddMock<TReturn>(

this IServiceCollection services,

string url, HttpMethod method, TReturn @return

)

{

var mockItem = new MockItem();

mockItem.Executor = BuildExecutor<TReturn>(@return);

mockItem.RouteSelector = BuildRouteSelector(url, method);

return services.AddTransient<MockItem>(sp => mockItem);

}

public static IServiceCollection AddMock<TReturn>(

this IServiceCollection services,

Func<HttpRequestMessage, bool> routeSelector,

Func<HttpRequestMessage, HttpResponseMessage, Task> executor

)

{

var mockItem = new MockItem();

mockItem.Executor = executor;

mockItem.RouteSelector = routeSelector;

return services.AddTransient<MockItem>(sp => mockItem);

}

private static Func<HttpRequestMessage, bool> BuildRouteSelector(

string url, HttpMethod method

)

{

Func<HttpRequestMessage, bool> selector = request =>

{

if (url == "\*") return true;

return url.ToLower() == res.RequestUri.AbsolutePath.ToLower() && method == res.Method;

};

return selector;

}

private static Func<HttpRequestMessage, HttpResponseMessage, Task> BuildExecutor<TReturn>(TReturn @return)

{

Func<HttpRequestMessage, HttpResponseMessage, Task> executor = (request, response) =>

{

response.StatusCode = System.Net.HttpStatusCode.OK;

if (@return is HttpStatusCode)

response.StatusCode = (HttpStatusCode)Enum.Parse(

typeof(HttpStatusCode),

@return.ToString()

);

else if (@return is Exception)

throw @return as Exception;

else if (@return is string)

response.Content = new StringContent(@return as string);

else

response.Content = new StringContent(@return == null ?

"" : JsonConvert.SerializeObject(@return)

);

return Task.CompletedTask;

};

return executor;

}

Copy

此时，我们就可以在单元测试中对接口进行模拟，这样就实现了真正意义上的单元测试：

var services = new ServiceCollection();

// 添加 HttpClient并注册ApiMockHttpMessageHandler

services.AddHttpClient("ApiMock", options => {

options.BaseAddress = new Uri("https://blog.yuanpei.me");

})

.AddHttpMessageHandler<ApiMockHttpMessageHandler>();

// 添加3个模拟接口

services.AddMock("/api/status", HttpMethod.Get, HttpStatusCode.OK);

services.AddMock("/api/query", HttpMethod.Post, new Exception("帅哥你谁啊"));

services.AddMock("/api/order", HttpMethod.Get, new {

OrderId = "OR09874",

CreatedBy = "张三"

});

var serviceProvider = services.BuildServiceProvider();

var httpClientFactory = serviceProvider.GetRequiredService<IHttpClientFactory>();

var httpClient = httpClientFactory.CreateClient("ApiMock");

// 调用/api/order接口

var response = await httpClient.GetAsync("/api/order");

Copy

下图是模拟接口返回的结果，与我们期望的完全一致：

[Graphical user interface, text, application, email

Description automatically generated](https://i.loli.net/2021/04/30/k9lX12aSpcr8ReV.png)HttpMessageHandler 实现接口模拟

**本文小结**

古人云：**他山之石，可以攻玉**。原本被接口模拟(**Mock**)所吸引的博主，意外地收获了 HttpMessageHandler 这个令人兴奋的知识点。博主认为，它是一种可以对 请求消息(**HttpRequestMessage**) 和 响应消息(**HttpResponseMessage**) 进行处理，同时多个 HttpMessageHandler 可以组成一个完整的消息处理通道的中间件。在此基础上，我们实现了诸如**日志记录**、**请求重试**、**接口模拟**等等的扩展性功能。除此以外，它还可以应用到 **Http认证头处理** 、**客户端负载均衡**等方面。

https://blog.yuanpei.me/posts/2070070822/