<https://indepth.dev/posts/1118/insiders-guide-into-interceptors-and-httpclient-mechanics-in-angular>

首先，我们将使用文档中描述的方法来注册两个拦截器，这些拦截器将自定义标头添加到请求中。然后，我们将执行相同的操作，但是将使用自定义中间件链，而不是使用Angular定义的机制。最后，我们将研究HttpClient请求方法如何构建可观察的流HttpEvents以及[对不变性](https://angular.io/guide/http" \l "immutability)的[需求](https://angular.io/guide/http" \l "immutability)。

*样例*

首先，让我们实现两个简单的拦截器，每个拦截器都使用文档中描述的方法向发送的请求添加标头。对于每个拦截器，我们声明一个实现intercept方法的类。在此方法内，我们通过添加Custom-Header-1和Custom-Header-2来修改请求：

@Injectable()

export class I1 implements HttpInterceptor {

intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

const modified = req.clone({setHeaders: {'Custom-Header-1': '1'}});

return next.handle(modified);

}

}

@Injectable()

export class I2 implements HttpInterceptor {

intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

const modified = req.clone({setHeaders: {'Custom-Header-2': '2'}});

return next.handle(modified);

}

}

如您所见，每个拦截器都将下一个处理程序作为第二个参数。我们需要调用它以将控制权传递给中间件链中的下一个拦截器。我们很快就会发现您打电话时会发生什么，next.handle以及为什么有时您不需要这样做。另外，如果您一直想知道为什么需要clone()在请求上调用method，那么您很快就会得到答案。

实施后，我们需要使用HTTP\_INTERCEPTORS令牌注册它们：

@NgModule({

imports: [BrowserModule, HttpClientModule],

declarations: [AppComponent],

providers: [

{

provide: HTTP\_INTERCEPTORS,

useClass: I1,

multi: true

},

{

provide: HTTP\_INTERCEPTORS,

useClass: I2,

multi: true

}

],

bootstrap: [AppComponent]

})

export class AppModule {}

然后执行一个简单的请求，检查是否已添加标题：

@Component({

selector: 'my-app',

template: `

<div><h3>Response</h3>{{response|async|json}}</div>

<button (click)="request()">Make request</button>`

,

})

export class AppComponent {

response: Observable<any>;

constructor(private http: HttpClient) {}

request() {

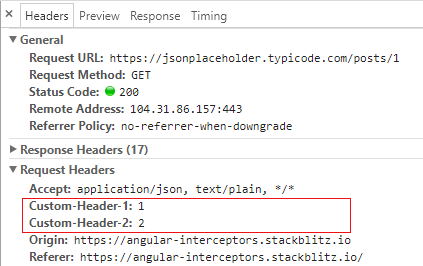
const url = 'https://jsonplaceholder.typicode.com/posts/1';

this.response = this.http.get(url, {observe: 'body'});

}

}

如果我们正确地完成了所有操作，则在检查Network选项卡时，我们应该会看到将标头发送到服务器：



*实施自定义中间件链*

我们的任务是将拦截器手动集成到请求处理逻辑中，而无需使用所提供的方法HttpClient。在这样做的同时，我们将完全像Angular在后台构建一个处理程序链。

处理请求

在现代浏览器中，使用XmlHttpRequest或Fetch API来实现AJAX功能。同样，库通常使用JSONP技术，有时会导致与更改检测有关的意外后果。因此，自然地，Angular需要一种使用上述方法之一向服务器发出请求的服务。这种服务被称为后端在文档上HttpClient，例如：

在拦截器中，next始终表示链中的下一个拦截器（如果有），或者在没有更多拦截器的情况下表示最终后端

在HttpClient通过角提供的模块有这种服务的两种实现方法- HttpXhrBackend使用的XmlHttpRequest API和JsonpClientBackend使用JSONP技术。HttpXhrBackend在中默认使用HttpClient。

Angular定义了一个称为HTTP（请求）处理程序的抽象，它负责处理请求。一个处理请求的中间件链包括HTTP处理程序，该处理程序将请求传递到链中的下一个处理程序，直到其中一个处理程序返回可观察到的流。处理程序的接口由抽象类HttpHandler定义：

export abstract class HttpHandler {

abstract handle(req: HttpRequest<any>): Observable<HttpEvent<any>>;

}

由于像HttpXhrBackend这样的后端服务可以通过发出网络请求来处理请求，因此它是HTTP处理程序的一个示例。通过与后端服务器通信来处理请求是最常见的处理形式，但不是唯一的一种。替代请求处理的一个常见示例是在不向服务器发出请求的情况下提供来自本地缓存的请求。使得能够处理的请求的任何服务应该实现handle其中，根据函数签名，返回一个可观察到的事件，诸如HTTP的方法HttpProgressEvent，HttpHeaderResponse或HttpResponse。因此，如果我们想提供一些自定义请求处理逻辑，则需要创建实现HttpHandler接口的服务。

*使用后端作为HTTP处理程序*

HttpClient服务将在HttpHandler令牌下注入在DI容器中注册的全局HTTP处理程序。然后，它通过触发handle方法使用它来发出请求：

export class HttpClient {

constructor(private handler: HttpHandler) {}

request(...): Observable<any> {

...

const events$: Observable<HttpEvent<any>> =

of(req).pipe(concatMap((req: HttpRequest<any>) => this.handler.handle(req)));

...

}

}

默认情况下，全局HTTP处理程序是HttpXhrBackend后端。它已在HttpBackend令牌中的令牌中注册：

@NgModule({

providers: [

HttpXhrBackend,

{ provide: HttpBackend, useExisting: HttpXhrBackend }

]

})

export class HttpClientModule {}

您可能会猜到HttpXhrBackend实现了HttpHandler接口：

export abstract class HttpHandler {

abstract handle(req: HttpRequest<any>): Observable<HttpEvent<any>>;

}

export abstract class HttpBackend implements HttpHandler {

abstract handle(req: HttpRequest<any>): Observable<HttpEvent<any>>;

}

export class HttpXhrBackend implements HttpBackend {

handle(req: HttpRequest<any>): Observable<HttpEvent<any>> {}

}

<>

由于默认的XHR后端是在HttpBackend令牌下注册的，因此我们可以自己注入它，并有效地代替使用HttpClient来进行请求。因此，不要使用HttpClient：

export class AppComponent {

response: Observable<any>;

constructor(private http: HttpClient) {}

request() {

const url = 'https://jsonplaceholder.typicode.com/posts/1';

this.response = this.http.get(url, {observe: 'body'});

}

}

<>

让我们直接使用默认的XHR后端，如下所示：

export class AppComponent {

response: Observable<any>;

constructor(private backend: HttpXhrBackend) {}

request() {

const req = new HttpRequest('GET', 'https://jsonplaceholder.typicode.com/posts/1');

this.response = this.backend.handle(req);

}

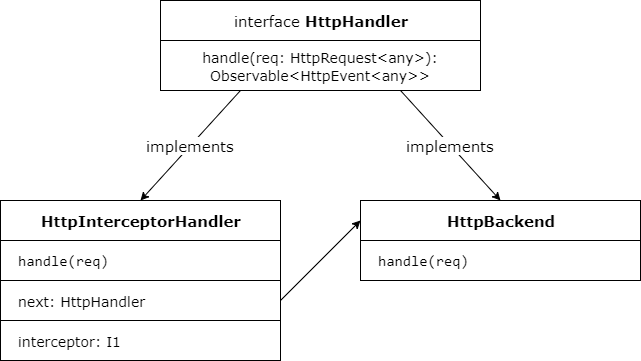
}

<>

这是演示。该示例中需要注意的几件事。首先，我们需要HttpRequest手动构建。其次，由于后端处理程序返回HTTP事件流，因此您将在屏幕上看到不同的对象闪烁，并最终呈现整个http响应对象。

*添加拦截器*

因此，我们设法直接使用了后端实现，但是由于尚未运行拦截器，因此尚未将标头添加到请求中。拦截器包含处理请求的逻辑，但是要与请求器一起使用，HttpClient则需要将其包装到实现HttpHandler接口的服务中。我们可以通过执行拦截器并将对链中下一个处理程序的引用传递给该拦截器的方式来实现此服务。这将使拦截器有可能触发下一个处理程序，该处理程序通常是后端。为此，每个自定义处理程序将保留对链中下一个处理程序的引用，并将其与请求一起传递给拦截器。所以我们想要这样的东西：



难怪Angular中已经存在这种包装处理程序的实现，并且称为HttpInterceptorHandler。因此，让我们使用它来包装其中一个拦截器。不幸的是，Angular不会将其导出为公共API，因此我们仅从源代码中复制基本实现：

export class HttpInterceptorHandler implements HttpHandler {

constructor(private next: HttpHandler, private interceptor: HttpInterceptor) {}

handle(req: HttpRequest<any>): Observable<HttpEvent<any>> {

// execute an interceptor and pass the reference to the next handler

return this.interceptor.intercept(req, this.next);

}

}

并像这样使用它包装我们的第一个拦截器：

export class AppComponent {

response: Observable<any>;

constructor(private backend: HttpXhrBackend) {}

request() {

const req = new HttpRequest('GET', 'https://jsonplaceholder.typicode.com/posts/1');

const handler = new HttpInterceptorHandler(this.backend, new I1());

this.response = handler.handle(req);

}

}

现在，一旦发出请求，我们就可以看到已将其Custom-Header-1添加到请求中。这是演示。通过上述实现，我们将一个拦截器包裹在其中HttpInterceptorHandler，该拦截器引用了下一个处理程序，即XHR后端。这已经是一连串的处理程序。

让我们在包装第二个拦截器的链中添加另一个处理程序：

export class AppComponent {

response: Observable<any>;

constructor(private backend: HttpXhrBackend) {}

request() {

const req = new HttpRequest('GET', 'https://jsonplaceholder.typicode.com/posts/1');

const i1Handler = new HttpInterceptorHandler(this.backend, new I1());

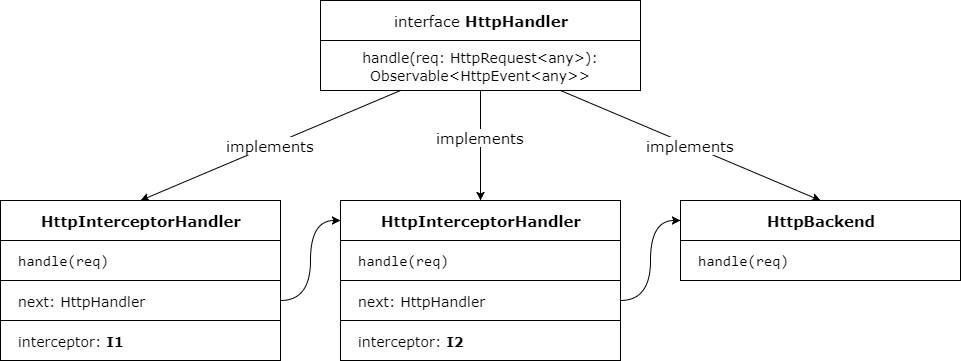
const i2Handler = new HttpInterceptorHandler(i1Handler, new I2());

this.response = i2Handler.handle(req);

}

}

在此处查看演示，现在一切正常，就像HttpClient在示例应用程序中使用时一样。我们要做的是，我们刚刚构建了处理程序的中间件链，其中每个处理程序执行一个拦截器，并将对下一个处理程序的引用传递给它。这是链图：



当我们next.handle(modified)在拦截器中执行该语句时，我们会将控制权传递给链中的下一个处理程序：

export class I1 implements HttpInterceptor {

intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

const modified = req.clone({setHeaders: {'Custom-Header-1': '1'}});

// passing control to the handler in the chain

return next.handle(modified);

}

}

最终，该控件将传递给最后一个后端处理程序，该后端处理程序将向服务器执行请求。

*自动包装拦截器*

与其通过逐个链接拦截器来手动构建链，我们可以通过向所有已注册的拦截器注入HTTP\_INTERCEPTORS令牌并使用reduceRight链接它们来自动完成链。让我们这样做：

export class AppComponent {

response: Observable<any>;

constructor(

private backend: HttpBackend,

@Inject(HTTP\_INTERCEPTORS) private interceptors: HttpInterceptor[]) {}

request() {

const req = new HttpRequest('GET', 'https://jsonplaceholder.typicode.com/posts/1');

const i2Handler = this.interceptors.reduceRight(

(next, interceptor) => new HttpInterceptorHandler(next, interceptor), this.backend);

this.response = i2Handler.handle(req);

}

}

我们需要在reduceRight这里使用从最后一个注册的拦截器开始构建链。使用上面的代码，我们得到与手动构建时相同的处理程序链。由返回的值reduceRight是对链中第一个处理程序的引用。

实际上，我上面编写的代码是使用拦截器功能在Angular中实现的。以下是消息来源中对此的评论：

构造一个HttpHandler，将一堆HttpInterceptor

应用于请求，然后再将其传递给指定的HttpBackend。

打算在`HttpClientModule`中用作工厂函数。

现在我们知道了它是如何做到的，因为我们在构建链时使用了完全相同的代码。HTTP处理程序中间件链的概况中的最后一点是，此功能已注册为默认值HttpHandler：

@NgModule({

providers: [

{

provide: HttpHandler,

useFactory: interceptingHandler,

deps: [HttpBackend, [@Optional(), @Inject(HTTP\_INTERCEPTORS)]],

}

]

})

export class HttpClientModule {}

因此，HttpClient服务将注入并使用执行该功能的结果，该结果是对链中第一个处理程序的引用。

*构建处理程序链的可观察流*

好的，现在我们知道我们有一堆处理程序，每个处理程序执行一个关联的拦截器并调用链中的下一个处理程序。通过调用此链返回的值是的可观察流HttpEvents。该流通常但并非总是由最后一个处理程序生成，这是后端的具体实现。其他处理程序通常只是返回该流。这是拦截器大多数实现的最后一条语句：

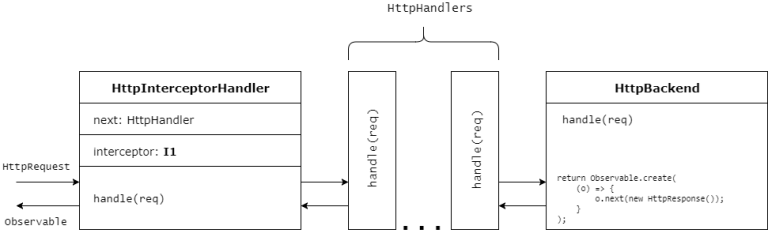
intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

...

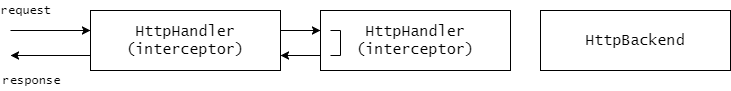
return next.handle(authReq);

}

因此，您可以这样表示逻辑：



但是，由于任何拦截器都可以返回可观察到的信息流，因此HttpEvents您有很多定制机会。例如，您可以实现自己的后端并将其注册为拦截器。或者实现一种[缓存机制](https://angular.io/guide/http" \l "caching)，如果找到该[缓存机制](https://angular.io/guide/http" \l "caching)，则立即返回缓存的值，而无需实际进行下一个处理程序的处理：



但是，由于任何拦截器都可以返回可观察到的信息流，因此HttpEvents您有很多定制机会。例如，您可以实现自己的后端并将其注册为拦截器。或者实现一种缓存机制，如果找到该缓存机制，则立即返回缓存的值，而无需实际进行下一个处理程序的处理：

同样，由于每个拦截器都可以访问下一个拦截器（通过next.handler()）调用返回的可观察流，因此拦截器可以通过RxJs运算符添加自定义逻辑来修改返回的流。

*构造HttpClient的可观察流*

如果您已经认真地阅读了前面的部分，您可能现在想知道处理程序链创建的HTTP事件流是否与调用HttpClient诸如get或的方法返回的流完全相同post。是的，实现不是很有趣。

HttpClient使用创建RxJs运算符使用请求对象启动其自己的可观察流，of并在调用的HTTP请求方法时将其返回HttpClient。处理程序链作为该流的一部分被同步处理，并且使用concatMapoperator展平链返回的可观察对象。该实施的要点是在request方法，因为所有的API的方法，如get，post和delete周围只是包装request：

const events$: Observable<HttpEvent<any>> = of(req).pipe(

concatMap((req: HttpRequest<any>) => this.handler.handle(req))

);

<>

在上面的代码段中，我call将新的替换了实例运算符的旧技术pipe。如果您仍然感到困惑，请concatMap阅读学习将RxJs序列与超直观的交互式图表结合。有趣的是，有一个原因使处理程序的链在以开头的可观察流中执行，of并在注释中进行了解释：

从初始请求Observable.of（）开始，然后在concatMap（）中运行处理程序（包括所有拦截器）。这样，处理程序在Observable链中运行，这会导致拦截器在每个订阅上都重新运行（这也使重试重新运行处理程序，包括拦截器）。

*处理`observe`请求选项*

通过创建的初始观察到的流HttpClient发射的所有HTTP事件，例如HttpProgressEvent，HttpHeaderResponse或HttpResponse。但是从文档中我们知道，我们可以使用以下observe选项指定对哪些事件感兴趣：

request() {

const url = 'https://jsonplaceholder.typicode.com/posts/1';

this.response = this.http.get(url, {observe: 'body'});

}

<>

与{observe: 'body'}从返回的可观察到的流get的方法将仅发射body响应的。的其他可能选项observe是events，response默认为后者。在探讨处理程序链的实现时，我指出了由链调用返回的流将发出所有HTTP事件。的责任是HttpClient根据observe参数过滤这些事件。

这意味着HttpClient我在上一节中演示的返回流的实现需要一些调整。我们可以做的就是过滤这些事件，并根据observe参数值将它们映射到不同的值。这是一个经过简化的实现，它确实做到了：

const events$: Observable<HttpEvent<any>> = of(req).pipe(...)

if (options.observe === 'events') {

return events$;

}

const res$: Observable<HttpResponse<any>> =

events$.pipe(filter((event: HttpEvent<any>) => event instanceof HttpResponse));

if (options.observe === 'response') {

return res$;

}

if (options.observe === 'body') {

return res$.pipe(map((res: HttpResponse<any>) => res.body));

}

<>

在这里您可以找到原始的实现。

*不变性的需求*

文档页面上的不变性有一个有趣的段落，如下所示：

存在拦截器来检查和更改传出的请求和传入的响应。但是，得知HttpRequest和HttpResponse类在很大程度上是不可变的，可能令人惊讶。这是有原因的：因为应用程序可能重试请求，所以拦截器链可能会多次处理单个请求。如果请求是可变的，则重试的请求将与原始请求不同。不变性确保拦截器每次尝试都看到相同的请求。

让我详细说明一下。当您调用HttpClient请求对象上的任何HTTP请求方法时，都会创建该对象。正如我在前面的部分中解释的那样，此请求用于启动可观察的$events序列，并在订阅时将其通过处理程序链传递。但是$events可以重试流，这意味着可以使用序列外部创建的原始请求对象再次触发序列。但是拦截器应始终从原始请求开始。如果请求是可变的，并且可以在拦截器运行期间进行修改，则此条件对于下一轮拦截器将不成立。因此，由于对同一请求对象的引用用于多次启动可观察序列的请求及其所有组成部分，例如HttpHeaders和HttpParams 应该是一成不变的。

<https://indepth.dev/posts/1051/top-10-ways-to-use-interceptors-in-angular>

https://github.com/melcor76/interceptors

*拦截器101*

HttpInterceptor是Angular 4.3引入的。它提供了一种在传递HTTP请求和响应之前转换或处理它们的方法。

尽管拦截器能够更改请求和响应，但是HttpRequest和HttpResponse实例属性仍然可以read-only使它们保持不变。—角度文档

这是因为如果一开始没有成功，我们可能想重试一个请求。不变性确保拦截器链可以多次重新处理同一请求。

您可以使用多个拦截器，但请记住以下几点：

Angular按您提供拦截器的顺序进行应用。如果提供拦截器A，然后是B，然后是C，则请求将在A-> B-> C中流动，而响应将在C-> B-> A中流出。

您以后不能更改顺序或删除拦截器。如果您需要动态启用和禁用拦截器，则必须将该功能内置到拦截器本身中。—角度文档

在示例应用程序中，我们提供了所有拦截器，但一次只能使用一个。这是通过检查路径来完成的。如果不是我们要查找的请求，我们可以将其传递给下一个拦截器next.handle(req).

