Ocelot 中文文档三

作者：loogn

原文链接：https://ld246.com/article/1562132016492

来源网站：链滴

转换Headers

Ocelot允许在请求下游服务之前和之后转换头部.目前Ocelot只支持查找和替换.这个功能在Github #10提出.我确定这个功能可以在各个方面发挥作用。

添加到请求

这个功能在GitHub #313被提出。

如果你想在你的上游请求中添加一个头，请在ocelot.json文件的ReRoute中添加如下配置：

```
"UpstreamHeaderTransform": {
    "Uncle": "Bob"
}
```

上面例子中，一个键为Uncle，值为Bob的头将被添加到上游服务中。

也支持占位符（看下面）。

添加到相应

这个功能在GitHub #280被提出。

如果你想在你的下游响应中添加一个头，请在ocelot.json文件的ReRoute中添加如下配置：

```
"DownstreamHeaderTransform": {
    "Uncle": "Bob"
},
```

上面例子中，当请求一个特定ReRoute的时候，Ocelot将返回一个键为Uncle，值为Bob的头。

如果你想返回Butterfly 跟踪id，需要像下面这样...

```
"DownstreamHeaderTransform": {
    "AnyKey": "{TraceId}"
},
```

查找并替换

为了变换头，首先我们指定头的键，然后指定我们想要的变换内容，例如

```
"Test": "http://www.bbc.co.uk/, http://ocelot.com/"
```

上面键是"Test",值是“http://www.bbc.co.uk/,http://ocelot.com/”，这个值的意思是使用http://ocelot.com/替换http://www.bbc.co.uk/，语法是{find},{replace}。希望还算简单明了，更多解释在下例子中。

下游请求之前

在ocelot.json的ReRoute中添加如下配置 ,以便用http://ocelot.com/替换http://www.bbc.co.uk/ .Tst头将被替换并发送到下游服务.

```
 "UpstreamHeaderTransform": {
    "Test": "http://www.bbc.co.uk/, http://ocelot.com/"
},
```

下游请求之后

在ocelot.json的ReRoute中添加如下配置 ,以便用http://ocelot.com/替换http://www.bbc.co.uk/ . Ocelot收到下游服务响应之后将进行替换.

```
"DownstreamHeaderTransform": {
    "Test": "http://www.bbc.co.uk/, http://ocelot.com/"
},
```

占位符

Ocelot允许在头不转换中使用占位符.

{BaseUrl} - 这个是Ocelot的基本url. 例如http://localhost:5000/. {DownstreamBaseUrl} - 这个是游服务的基本url 例如http://localhost:5001/. 目前这个只在DownstreamHeaderTransform中起作用 {TraceId} - 这个是Butterfly的跟踪id.目前这个也只在DownstreamHeaderTransform中起作用.

处理 302 重定向

Ocelot默认会自动遵循重定向，但是如果您想将location头返回给客户端，您可能需要将location更为Ocelot而不是下游服务。 Ocelot可以使用以下配置实现。

```
"DownstreamHeaderTransform": {
    "Location": "http://www.bbc.co.uk/, http://ocelot.com/"
},
 "HttpHandlerOptions": {
    "AllowAutoRedirect": false,
},
```

你也可以使用BaseUrl占位符.

```
"DownstreamHeaderTransform": {
    "Location": "http://localhost:6773, {BaseUrl}"
},
 "HttpHandlerOptions": {
    "AllowAutoRedirect": false,
},
```

最后,如果你使用负载均衡的话,你将得到多个下游基地址,所以像上面那样是不能正常工作的.在这种情下你可以如下配置.

```
"DownstreamHeaderTransform": {
    "Location": "{DownstreamBaseUrl}, {BaseUrl}"
},
 "HttpHandlerOptions": {
    "AllowAutoRedirect": false,
},
```

未来

理想情况下，这个特性能够支持实际上头部可以有多个值的情况。目前只是假设一个。 如果它可以查和替换多个值应该是非常棒的。

```
"DownstreamHeaderTransform": {
    "Location": "[{one,one},{two,two}"
},
 "HttpHandlerOptions": {
    "AllowAutoRedirect": false,
},
```

如果有人想在这一展身手,请自己搞定吧!

转换Claims

Ocelot允许用户访问claims并把它们转换到头部,请求字符串参数和其他claims中.这仅在用户通过身验证后才可用。

用户通过身份验证之后,我们运行claims转换中间件.这个中间件允许在授权中间件调用之前转换claims当用户身份验证之后,首先会调用claims转换到头的中间件,最后调用claims转换到查询字符串的中间件.

执行转换的语法对于每个处理都是相同的。在ReRoute配置中，使用特定名称AddClaimsToRequest AddHeadersToRequest，AddQueriesToRequest添加一个json字典。

注意,我不是一个编程专家,所以不知道这个语法是否好...

在词典中，这些条目指定了Ocelot应该如何转换！ 字典的键将成为claim,头,查询参数的键。

条目的值将被解析成转换的逻辑.首先指定了一个字典访问器,例如Claims[CustomerId].意思是我们想问claims并获取键为CustomerId的claim类型.然后一个大于号(>)用于分隔.下一个条目是值或带索引的值.如果指定了单个值，Ocelot将取该值并将其添加到变换中。如果该值有一个索引器，Ocelot将找在另一个大于符号后面提供的分隔符。 然后，Ocelot会使用分隔符将值分开，并将所需的索引添到转换中。

Claims 到 Claims 的转换

下面是一个Claims到Claims转换的例子

```
    "UserType": "Claims[sub] > value[0] > |",
    "UserId": "Claims[sub] > value[1] > |"
}
```

这显示了Ocelot查看用户的sub声明并将其转换为UserType和UserId声明的转换。 假设sub声明看来像这样“usertypevalue | useridvalue”。

Claims到头的转换

下面是一个Claims到头转换的例子

```
"AddHeadersToRequest": {
    "CustomerId": "Claims[sub] > value[1] > |"
}
```

这显示了Ocelot查看用户的sub声明并将其转换为CustomerId头的转换。 假设sub声明看起来像这

“usertypevalue | useridvalue” 。

Claims 到查询字符串参数的转换

下面是一个Claims到查询字符串参数转换的例子

```
"AddQueriesToRequest": {
    "LocationId": "Claims[LocationId] > value",
}
```

这显示了Ocelot查看用户的LocationId声明并将其作为发往下游服务的查询字符串参数LocationId的换。

日志

目前，Ocelot使用标准的日志记录接口ILoggerFactory/ILogger <T>。 在IOcelotLogger / IOcelotoggerFactory中提供了标准的asp.net core日志记录的一个实现。 因为Ocelot在日志中添加了一些外的信息，如请求ID（如果已配置的话）。

这有个全局的错误处理程序，可以捕获所有异常并作为错误记录他们。

最后，如果日志记录设置为跟踪级别，Ocelot将记录开始，结束和任何抛出异常的中间件，这些异常能非常有用。

不是使用标准框架的日志记录的原因是，我无法覆盖将IncludeScopes设置为true时记录的请求标识。

警告

如果您记录日志到控制台，您将获得糟糕的性能。 我遇到过很多关于Ocelot性能的问题，它始终记调试级别的日志，并记录到控制台 :) 所有请确保您生产中记录了正确的东西 :)

跟踪

Ocelot使用一个杰出的项目Butterfly 提供了跟踪功能。

为了使用跟踪，请阅读Butterfly的文档。

在Ocelot中如果你想跟踪一个ReRoute，你需要做如下事情：

在ConfigureServices方法中

```
services
    .AddOcelot()
    .AddOpenTracing(option =>
    {
        //this is the url that the butterfly collector server is running on...
        option.CollectorUrl = "http://localhost:9618";
        option.Service = "Ocelot";
    });
```

然后在ocelot.json文件中，添加如下配置到你想要跟随的ReRoute中。

```
"HttpHandlerOptions": {
    "UseTracing": true
 },
```

现在，当这个ReRoute被调用的时候，Ocelot会发生跟踪信息到Butterfly。

请求Id和关联Id

Ocelot支持一个客户端以头的形式发送requestid。 如果设置了，一旦中间件管道中可用，Ocelot便使用这个requestid进行日志记录。 Ocelot也会使用指定头将requireid转发给下游服务。

如果在日志配置中你设置IncludeScopes为true,你还可以在日志中获取asp.net core的请求id。

为了是用requestid,有两种选择。

全局

在ocelot.json的GlobalConfiguration配置块中如下设置。这样所有进入Ocelot的请求都会起作用。

```
 "GlobalConfiguration": {
  "RequestIdKey": "OcRequestId"
}
```

我建议使用GlobalConfiguration，除非你真的需要它是指定ReRoute的。

ReRoute

如果你想覆盖全局设置，在ocelot.json的特定ReRoute中添加如下设置。

```
"RequestIdKey": "OcRequestId"
```

一旦Ocelot识别出与ReRoute对象匹配的请求，它将根据ReRoute的配置来设置requestid。

这可能会导致一下困惑。如果你在GlobalConfiguration中设置了requestid，可能在ReRoute被匹配是一个，匹配后是另一个，因为requestid的key会变。这是因为设计如此，而且这是我目前能想到的好的解决方案了。在这种情况下OcelotLogger会在日志中记录当前requestid和上一个requestid。

下面的例子是debug级别下一个正常请求的日志记录。

```
dbug: Ocelot.Errors.Middleware.ExceptionHandlerMiddleware[0]
      requestId: asdf, previousRequestId: no previous request id, message: ocelot pipeline start
d,
dbug: Ocelot.DownstreamRouteFinder.Middleware.DownstreamRouteFinderMiddleware[0]
      requestId: asdf, previousRequestId: no previous request id, message: upstream url path is
upstreamUrlPath},
dbug: Ocelot.DownstreamRouteFinder.Middleware.DownstreamRouteFinderMiddleware[0]
      requestId: asdf, previousRequestId: no previous request id, message: downstream templat
 is {downstreamRoute.Data.ReRoute.DownstreamPath},
dbug: Ocelot.RateLimit.Middleware.ClientRateLimitMiddleware[0]
      requestId: asdf, previousRequestId: no previous request id, message: EndpointRateLimitin
 is not enabled for Ocelot.Values.PathTemplate,
dbug: Ocelot.Authorisation.Middleware.AuthorisationMiddleware[0]
      requestId: 1234, previousRequestId: asdf, message: /posts/{postId} route does not require
user to be authorised,
```

```
dbug: Ocelot.DownstreamUrlCreator.Middleware.DownstreamUrlCreatorMiddleware[0]
      requestId: 1234, previousRequestId: asdf, message: downstream url is {downstreamUrl.Da
a.Value},
dbug: Ocelot.Request.Middleware.HttpRequestBuilderMiddleware[0]
      requestId: 1234, previousRequestId: asdf, message: setting upstream request,
dbug: Ocelot.Requester.Middleware.HttpRequesterMiddleware[0]
      requestId: 1234, previousRequestId: asdf, message: setting http response message,
dbug: Ocelot.Responder.Middleware.ResponderMiddleware[0]
      requestId: 1234, previousRequestId: asdf, message: no pipeline errors, setting and returni
g completed response,
dbug: Ocelot.Errors.Middleware.ExceptionHandlerMiddleware[0]
      requestId: 1234, previousRequestId: asdf, message: ocelot pipeline finished,
```

中间件注入和重写

警告！请谨慎使用。 如果您在中间件管道中看到任何异常或奇怪的行为，并且正在使用以下任何一种为。删除它们，然后重试！

当在Startup.cs中配置Ocelot的时候，可以添加或覆盖中间件。如下所示：

```
var configuration = new OcelotPipelineConfiguration
{
   PreErrorResponderMiddleware = async (ctx, next) =>
   {
      await next.Invoke();
   }
  app.UseOcelot(configuration);
};
```

在上面的例子中，提供的函数将在第一个Ocelot中间件之前运行。 这允许用户在Ocelot管道运行之和之后提供他们想要的任何行为。 这意味着你可以打破一切，你开心就好！

用户可以针对以下内容设置功能。

- PreErrorResponderMiddleware - 上面已经解释过了.
- PreAuthenticationMiddleware - 这个允许用户执行预认证逻辑，然后再调用 Ocelot的认证中间。
- AuthenticationMiddleware - 可以重写Ocelot的认证中间件。
- PreAuthorisationMiddleware - 这个允许用户执行预授权逻辑，然后再调用 Ocelot的授权中间。
- AuthorisationMiddleware - 可以重写Ocelot的授权中间件。
- PreQueryStringBuilderMiddleware - 这允许用户在传递给Ocelot请求创建器之前在http请求上理查询字符串。

很明显，您只能在调用app.UseOcelot()之前添加中间件，而不能在它之后，因为Ocelot不会调用下个中间件。

负载均衡

Ocelot能通过可用的下游服务对每个ReRoute进行负载平衡。 这意味着您可以扩展您的下游服务，且Ocelot可以有效地使用它们。

可用的负载均衡器的类型是：

LeastConnection - 最少连接，跟踪哪些服务正在处理请求，并把新请求发送到现有请求最少的服务。该算法状态不在整个Ocelot集群中分布。

RoundRobin - 轮询可用的服务并发送请求。 该算法状态不在整个Ocelot集群中分布。

NoLoadBalancer - 不负载均衡，从配置或服务发现提供程序中取第一个可用的下游服务。

CookieStickySessions - 使用cookie关联所有相关的请求到制定的服务。下面有更多信息。

你必须在你的配置中选择使用哪种负载均衡方式。

配置

下面展示了如何使用ocelot.json给一个ReRoute设置多个下游服务，并选择LeadConnection负载均器。这是设置负载均衡最简单的方法。

```
{
   "DownstreamPathTemplate": "/api/posts/{postId}",
   "DownstreamScheme": "https",
   "DownstreamHostAndPorts": [
         {
            "Host": "10.0.1.10",
            "Port": 5000,
         },
         {
            "Host": "10.0.1.11",
            "Port": 5000,
         }
      ],
   "UpstreamPathTemplate": "/posts/{postId}",
   "LoadBalancerOptions": {
      "Type": "LeastConnection"
   },
   "UpstreamHttpMethod": [ "Put", "Delete" ]
}
```

服务发现

下面展示了如何使用服务发现设置一个ReRoute，并选择LeadConnection负载均衡器。

```
{
   "DownstreamPathTemplate": "/api/posts/{postId}",
   "DownstreamScheme": "https",
   "UpstreamPathTemplate": "/posts/{postId}",
   "UpstreamHttpMethod": [ "Put" ],
   "ServiceName": "product",
   "LoadBalancerOptions": {
      "Type": "LeastConnection"
   },
   "UseServiceDiscovery": true
}
```

设置此操作时，Ocelot将从服务发现提供程序查找下游主机和端口，并通过所有可用服务来负载均衡求。 如果您向服务发现提供程序（consul）添加和删除服务，那么Ocelot将遵循这一点，停止调用被删除的服务并开始调用已添加的服务。

CookieStickySessions

我已经实现了一个非常基本的粘性会话类型的负载平衡器。 它意味着支持的场景是你有一堆不共享会状态的下游服务器，如果你为其中一台服务器获得多个请求，那么它应该每次都去同一个盒子，否则用户的会话状态可能不正确。 这个特性在问题＃322中有被提出，尽管用户想要的比只是粘性会话更复杂：），无论如何，我认为这是个不错的功能！

为了设置CookieStickySessions负载均衡器，你需要做如下事情。

```
{
   "DownstreamPathTemplate": "/api/posts/{postId}",
   "DownstreamScheme": "https",
   "DownstreamHostAndPorts": [
         {
            "Host": "10.0.1.10",
            "Port": 5000,
         },
         {
            "Host": "10.0.1.11",
            "Port": 5000,
         }
      ],
   "UpstreamPathTemplate": "/posts/{postId}",
   "LoadBalancerOptions": {
      "Type": "CookieStickySessions",
      "Key": "ASP.NET_SessionId",
      "Expiry": 1800000
   },
   "UpstreamHttpMethod": [ "Put", "Delete" ]
}
```

LoadBalancerOptions的Type需要是CookieStickySessions，Key是您希望用于粘性会话的cookie名称，Expiry是您希望会话被粘合的时间，以毫秒为单位。 请记住，每次请求都会刷新，这意味着会仿会话的工作方式（滑动过期--译者注）。

如果您有多个具有相同LoadBalancerOptions的ReRoutes，那么所有这些ReRoutes将为随后的请求用相同的负载均衡器。 这意味着会话将会跨ReRoute进行粘合。

请注意，如果您提供多个DownstreamHostAndPort或者您正在使用Consul等服务发现提供程序，且返回多个服务，则CookieStickySessions将使用循环选择下一个服务器。 目前是硬编码，但可以变。

委托处理程序

Ocelot允许用户将委托处理程序添加到HttpClient传输中。 这个功能在github #208中提出，我确定会以各种方式被使用。之后我们在GitHub＃264中进行了扩展。

用法

为了将委托处理程序添加到HttpClient传输中，有两件重要的事情要做。

首先，为了创建一个可以用于委托处理程序的类，它必须如下所示。 我们将在asp.net core容器中注这些处理程序，以便您可以将您已注册的其他服务注入到处理程序的构造函数中。

```
public class FakeHandler : DelegatingHandler
{
    protected override async Task<HttpResponseMessage> SendAsync(HttpRequestMessage r
quest, CancellationToken cancellationToken)
    {
        //do stuff and optionally call the base handler..
        return await base.SendAsync(request, cancellationToken);
    }
}
```

Next you must add the handlers to Ocelot’ s container either as singleton like follows..

其次，您必须将处理程序添加到Ocelot的容器，要么作为单例注册。

```
services.AddOcelot()
        .AddSingletonDelegatingHandler<FakeHandler>()
        .AddSingletonDelegatingHandler<FakeHandlerTwo>()
```

要么注册为临时的...

```
services.AddOcelot()
        .AddTransientDelegatingHandler<FakeHandler>()
        .AddTransientDelegatingHandler<FakeHandlerTwo>()
```

这两个Add方法都有一个名为global的参数，它默认为false。 如果它是false，那么DelegatingHandlr需要通过ocelot.json设置特定的ReRoutes（稍后更多）。 如果设置为true，则它将成为全局处理程，并将应用于所有ReRoutes。

例如：

```
services.AddOcelot()
        .AddSingletonDelegatingHandler<FakeHandler>(true)
```

或者临时注册

```
services.AddOcelot()
        .AddTransientDelegatingHandler<FakeHandler>(true)
```

最后，如果你想要ReRoute指定DelegatingHandlers为你的特定DelegatingHandlers，或全局（稍会详细介绍）DelegatingHandlers，那么你必须将下面的json添加到ocelot.json中的特定ReRoute。 数组中的名称必须与您的DelegatingHandlers类名匹配，以便Ocelot将它们匹配在一起。

```
"DelegatingHandlers": [
    "FakeHandlerTwo",
    "FakeHandler"
]
```

你可以有多个DelegatingHandlers，他们的运行顺序如下：

1. 所有在服务中并且不在ocelot.json的DelegatingHandlers数组中的全局处理程序按加入的顺序排。

2. 所有非全局的DelegatingHandlers以及来自ocelot.json的DelegatingHandlers数组中的所有全局量都按照它们在DelegatingHandlers数组中的顺序排列。

3. 如果启用了跟踪，那么这一步是跟踪DelegatingHandler(看跟踪文档).

4. 如果启用了QoS，那么这一步是QoS DelegatingHandler (看服务质文档).

5. HttpClient发生HttpRequestMessage.

希望其他人会洞悉这个功能是很有用的!

Raft(实验功能不能用于生产环境)

Ocelot最近整合了Rafty，这是我在去年一直研究的Raft的一个实现。 这个项目实验性非常强，所以我认为它没问题之前，请不要在生产环境中使用Ocelot的这个功能。

Raft是一种分布式一致性算法，它允许一组服务器（Ocelots）保持本地状态，而不需要一个集中式据库（例如SQL Server）存储状态。

为了在Ocelot中启用Rafty，您必须对Startup.cs进行以下改动。

```
public virtual void ConfigureServices(IServiceCollection services)
{
    services
      .AddOcelot()
      .AddAdministration("/administration", "secret")
      .AddRafty();
}
```

除此之外，您还必须在您的主项目中添加名为peers.json的文件，其内容看起来如下所示：

```
{
   "Peers": [{
         "HostAndPort": "http://localhost:5000"
      },
      {
         "HostAndPort": "http://localhost:5002"
      },
      {
         "HostAndPort": "http://localhost:5003"
      },
      {
         "HostAndPort": "http://localhost:5004"
      },
      {
         "HostAndPort": "http://localhost:5001"
      }
   ]
}
```

Ocelot的每个实例必须在数组中有它的地址，以便它们可以使用Rafty进行通信。

完成这些配置更改后，您必须使用peers.json文件中的地址部署和启动Ocelot的每个实例。 然后服务应该开始彼此通信！ 您可以通过发布配置更新来检测一切是否正常工作，并通过配置来检查它是否已制到所有服务器。