**[WebAPI前置知识:HTTP与RestfulAPI](https://www.cnblogs.com/UliiAn/p/5353475.html)**

         对HTTP协议的基本了解是能理解并使用RestFul风格API的基础，在了解了这些基础之后，使用各种RestFul的开发框架才能得心应手。我一开始使用WebApi的时候就因为对这些知识缺乏了解，觉得用起来各种不顺手，直到熟悉了这些HTTP的知识后，使用WebApi开发起来才觉得得心应手，我的理解里，RestFul风格的API即是对HTTP协议良好支持，实现HTTP完整语义风格的API。

         在介绍这些知识之前，我需要强调一下很多人存在的一个误区：HTTP的谓词和数据传递方式。绝大多数人接触并使用的HTTP协议都是在网站编写的过程中，在一般的WEB应用中，我们仅使用GET、POST两个谓词，其他谓词并不适用，在这一习惯下很多人有几个奇怪的认知：HTTP协议只适用于网站开发，HTTP仅有两个谓词：GET/POST，HTTP调用数据传递仅使用表单K-V的形式进行；在这种认知下，用这种风格开发的RestApi经常会不伦不类，使用ASP.NET WebAPi也会显得不伦不类，平添麻烦。而我们首先要认识到，网站的数据交互只是HTTP使用的一个场景而已，HTTP可以传递各种形式的数据。

         我们从HTTP的第一行说起：HTTP的第一行包含三个信息：谓词、URL、HTTP协议版本。三个数据使用空格隔开。

         谓词：对于RestFul API来说谓词是非常重要的一个元素，WEB API就是使用谓词作为默认的路由方式，最常用的谓词有：POST\DELETE\PUT\GET，这四个谓词对应了“增、删、改、查”四个动作（POST和PUT谁是增谁是改不同资料总有不同的说法，我其实有略微有点困惑啦……有定义说PUT是幂等操作，而POST不是，那PUT就更偏重于改而POST更偏重于增）。最常用的谓词即为这四个，也有其他谓词拥有不同的语义：

HEAD：仅返回相应头部，不包含Body

TRACE：对数据传输过程进行诊断

OPTIONS：请求 Web 服务器告知其支持的各种功能

还有其他谓词，如果需要可以查询相关文档，但并不常用。

其中，GET，DELETE不包含BODY，PUT,POST可以包含BODY。而如果一个谓词包含了语义之外的操作，例如GET中带BODY，POST用于删除资源这种操作也是被允许的，称之为谓词的重载，虽然HTTP可以支持谓词的重载，但并不建议使用，因为不符合标准语义。

         URL : URL定义了一个资源，例如[www.example.com/person](http://www.example.com/person) 定义了person为一个资源，结合上面所介绍的谓词，我们提供Person一组操作：

         GET www.example/person/1 即获取ID为1的用户的信息

         POST [www.example/person/](http://www.example/person/) (BODY中包含Person的描述) 创建一个Person资源

         PUT [www.example/person/1](http://www.example/person/1) (BODY中包含Person的描述) 更新一个Person资源

         DELETE www.example/person/1 删除ID为1的Person资源

         HTTP版本：

         目前主要使用的是HTTP1.0 和 HTTP1.1协议，HTTP2.0协议正在普及阶段，用的还不是很多。HTTP1.0 和HTTP1.1区别很小，其中的差异对于RestFul来说影响并不是很大。具体的差别大家可以查询相关文档。

HTTP的第一行内容就是这些，接下来会有一个\r\n来进行换行，接下来就是HTTP HEAD部分，HTTP HEAD描述了HTTP请求和响应。我认为HTTP HEAD即为HTTP协议中最重要的部分，他包含了编码、BODY长度、内容协商等信息，你也可以包含一些自定义信息。下面我来为大家介绍几个在RestFul API中常用的HEAD:

         User-Agent:用户代理，是什么客户端发出的请求，如IE、Chrome、Fiddler等

         HOST:域名（HOST一般用于服务器的站点绑定，一般和URL的域名相同，但是在一些自定义的DNS使用方式中，可能会出现HOST和URL中的域名不一致）

         Authorization：验证信息，这个字段可以包含一些用于用户验证的信息，而表示方法为：schema authorinfo，中间使用空格隔开，其中schema代表了验证方法，authorinfo代表了验证信息，常见的schema 如 Base：authorinfo使用用户名+密码，并用Base64进行编码。或者使用Token，类似于Session的方式。

Accept：接受何种序列化方式返回的数据，用MIME表示，用于对响应数据的内容协商，可以包含多个MIME，按优先顺序排列，如application/json，application/xml，text/html；具体服务器可以返回什么类型的数据需要由服务器支持情况而定，有一些标准MIME，可以查到；有时我们也需要一些自定义的MIME，例如bson、protocolbuffer等，我们可以自定义MIME，在服务端开发自己的实现，而这些特的扩展在ASP.NET WebApi中都有相应的扩展点。

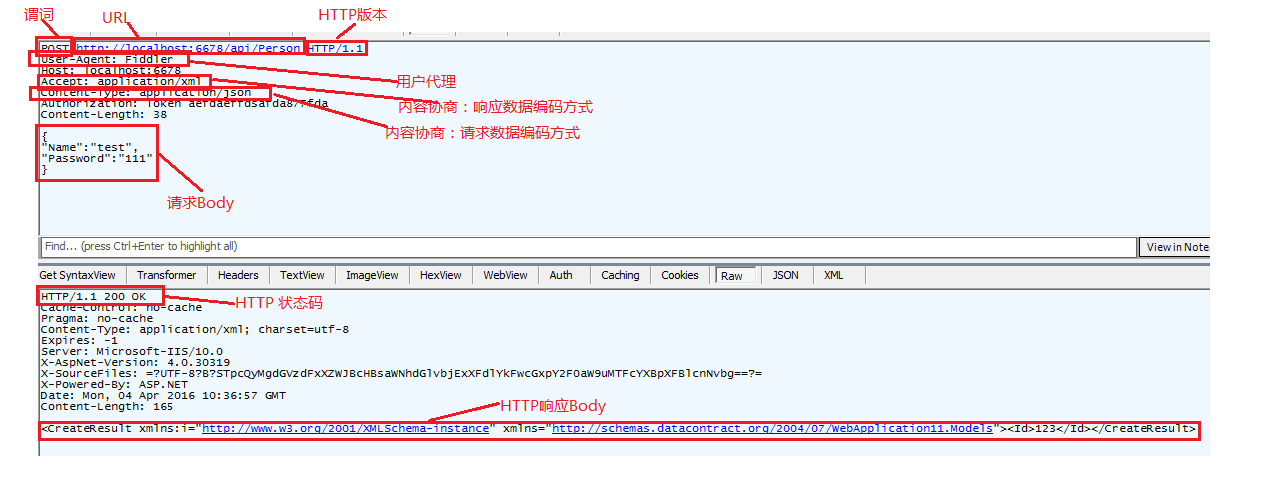
         Content-Type:使用一个MIME表示，表示所发送请求的Body的序列化方式，常见的如application/json，还有WEB交互最常使用的application/x-www-form-urlencoded，都表示了你的body部分的序列化方式，在请求、响应中都会出现

         HTTP HEAD部分我认为是HTTP协议中最核心的部分，其中可配置、使用的地方实在太多太多，而且有太多的细节，以上为我列出的在我的工作中最常用的部分，介绍这些内容的资料全部列出来足够完成一本书了，大家有兴趣可以查找相关资料，在Rest API中，内容协商经常让一开始学习使用Rest的人很迷惑，一定要记住Accept，Content-Type两个头的作用和区别，Accept表示希望接受什么样的数据，Content-Type表示当前请求中Body的编码方式。在ASP.NET WEBAPI中，如果请求中有Content-Type，而没有ACCEPT，则默认使用Content-Type中的内容作为响应的内容协商。

         响应部分也分为头部和Body，响应头部和请求头部最大的不同在于响应首行存在一个HTTP Code，HTTP Code作为API的调用状态的展示，也很重要，在REST API中最常用的状态码一般为2XX,4XX,5XX三个段，而1XX表示工作还要继续，3XX一般表示重定向，在REST API中使用的并不多。而在最常用的三个Status 段中，2XX表示执行成功，4XX表示客户端数据错误(例如参数校验不通过)，5XX表示服务器端处理错误，例如有未处理的异常(如数据库连接错误)，根据这些状态码可以初步判断API调用的执行状态。

         在首部之后有一个空行（\r\n）接下来就是Content，这里有具体的业务数据，根据不同的Content-Type使用不同的序列化方式表示，例如JSON，XML，甚至HTML。各位在学习HTTP API时可以认为网页应用也是HTTP 的一种应用，只是交互方式一般使用application/x-www-form-urlencoded 作为请求、 text/html作为响应的方式进行交互。而RestAPI可以使用其他很多种编码方式进行交互，支持的更广，网页应用只是使用HTTP传输的一种应用场景，RestAPI和网页是可以不分开的。我觉得这一点Nancy比ASP.NET做得更好，Nancy并没有把RestAPI和网页割裂开来，而ASP.NET用MVC和WEBAPI将两者割裂了；请求一个数据，我可以要求Accept为application/json时返回Json数据，而使用text/html时返回一个网页；当然，将这两种应用方式切割或合并起来都各有优劣。

         我所写的这些对于HTTP协议而言实在太少太少，大家有兴趣的可以自行查找相关资料，我只是写出了WEB API中常用的部分，下面我们来用一张图为大家展示一下这些知识：



[**WEB API系列（一）：WEB API的适用场景、第一个实例**](https://www.cnblogs.com/UliiAn/p/5373854.html)

       在我[前一篇博客](http://www.cnblogs.com/UliiAn/p/5353475.html" \o "WebAPI前置知识:HTTP与RestfulAPI" \t "_blank)中已经给各位简单介绍了HTTP协议与RestFul API的关系，以及一些基本的HTTP协议知识，在这些知识的铺垫下，今天，我们一起来讨论一下WEB API的适用场景，然后写我们第一个WEB API接口，并演示如何对其进行简单调用。

       很多人都很迷惑，既然有了WCF为什么还要有WEB API？WEB API会不会取代WCF？

       就我的看法，WCF提供的是一种RPC实现的集合，WCF的设计更多地考虑了SOA的场景，以及各种RPC的问题。很多人也会说，RestFul API也是一种RPC啊，并且WCF中也有关于RestFul 的实现啊。很多资料中RPC和RestFul在风格概念上是有一些区别的，其实我觉得这两者的区别比较主观，过度纠结这些就学院派了；我主要关注了实际使用上的一些问题，在WCF中，支持的协议很多，WS-\*系列协议，以及一些更简洁的协议，其中提供了一些专用通信协议的性能是非常高的，并且WCF还提供了服务发现等功能，我认为WCF更适合内部系统间的高性能调用，社区中也有其他一些RPC方案可以选择，例如gRPC，Avor，thrift都是和WCF定位相同的产品；而WEB API是关注于HTTP RestFul风格的产品，在此基础上，任何语言、任何终端都能非常容易地进行对接，并且能利用非常成熟的各种HTTP基础设施和解决方案来进行开发、调试、负载均衡、内容分发。所以，WEB API是一种针对HTTP的，偏重于快速开发RestFul风格开放式API的开发框架。目前看来，他并不能取代WCF，他们各有适合的场景，不能认为WEB API是WCF的替代产品。

       OK，现在我们来开发第一组WEB API接口！使用VS2012以后的版本都有现成的WEB API创建模板，大家跟着创建就好了，创建出来后，项目中会有MVC、WEB API的项目，WEB API对MVC有依赖，不能单独创建！而WEB API和MVC都是利用类似的路由机制，所以在默认路由中，WEB API 使用

/api/{controller}/{id}

       作为路由，添加了/api/节以区分MVC和web api。

       接下来，我们添加一个WEB API的Controller，取名为PersonController，他继承于ApiController；在创建这个Controller的时候，我们就定义了一种资源：Person，在PersonController里的所有操作均围绕着Person这个资源来的。接下来我们开始定义一组增删改查操作。

       在Web API中，默认路由采用了一种约定：根据谓词来进行路由，而方法名的前缀就是调用该方法对应使用的HTTP谓词。代码示例如下：

[复制代码](javascript:void(0);)

/// <summary>

/// Person 为资源，对Person进行的一组操作

/// </summary>

public class PersonController : ApiController

{

private static List<Person> \_personLst = new List<Person>();

/// <summary>

/// 获取一个Person

/// </summary>

/// <param name="id">Person的ID</param>

/// <returns>Person</returns>

public Person GetPerson(long id)

{

return \_personLst.Find(x => x.Id == id);

}

/// <summary>

/// 添加一个Person

/// </summary>

/// <param name="person">Person</param>

public void PostAddPerson(Person person)

{

\_personLst.Add(person);

}

/// <summary>

/// 修改一个

/// </summary>

/// <param name="id">Person Id</param>

/// <param name="person">新</param>

public void PutModifyPerson(long id, Person person)

{

var p = \_personLst.Find(x => x.Id == id);

p.Age = person.Age;

p.Name = person.Name;

p.Sex = person.Sex;

}

/// <summary>

/// 删除一个Person

/// </summary>

/// <param name="id">Person ID</param>

public void DeletePerson(long id)

{

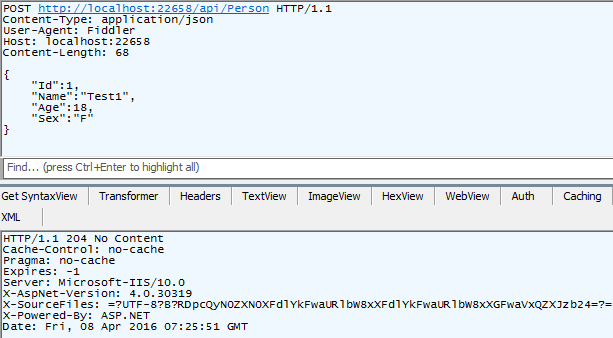
\_personLst.RemoveAll(x => x.Id == id);

}

}

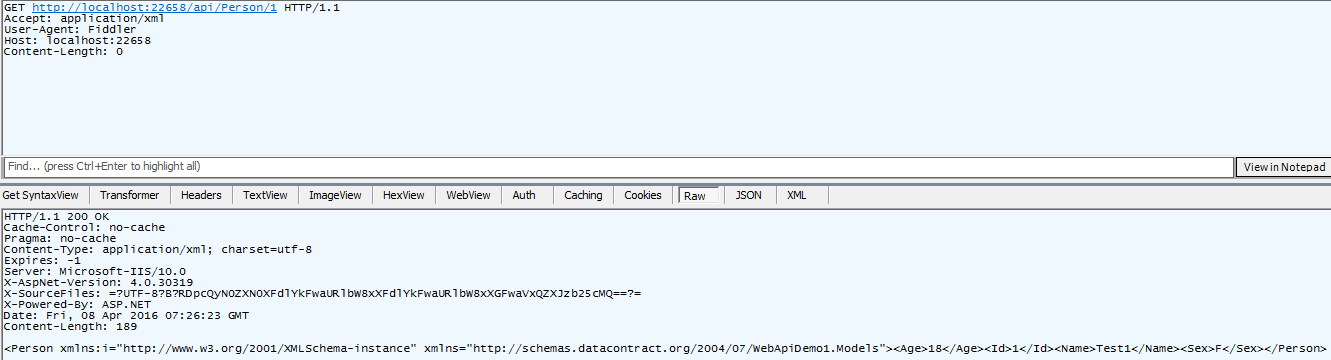
[复制代码](javascript:void(0);)

一个简单的针对资源的CRUD操作的API就好了，不用解析输入，不用拼接输出，就是那么简单！让我们来遛一遛！



发送请求：谓词为POST，语义创建Person，Person描述在Body里，head中声明了Body通过Json序列化。

收到响应：响应码204，属于2XX类型执行成功，Body里没有数据



发送请求：谓词为GET，语义为查询Person资源，Id为1的，head中声明希望接收使用XML序列化的数据

收到响应：响应码为200，执行成功，Body中有数据，数据使用XML序列化



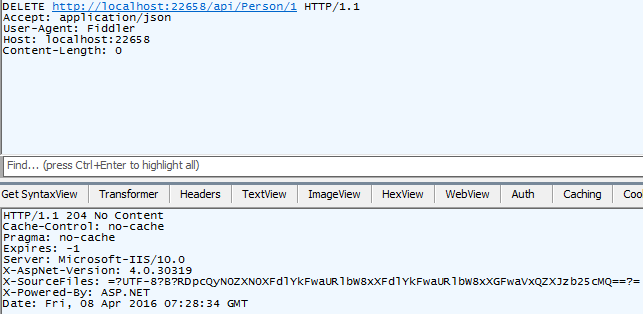
发送请求：谓词为PUT，语义为修改ID为1的Person资源，修改内容在Body中，Content-Type标明Body使用Json序列化，在Body中我们将Name修改为Test1Changed

收到响应，响应码为204，执行成功



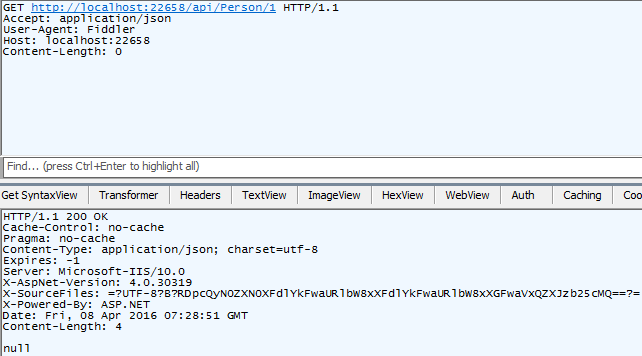
发送请求：谓词为GET，语义为查询ID为1的Person资源，Accept标明希望接收到Json数据

收到响应：可以看到Body为使用Json序列化的内容，Name属性已经变更为Test1Changed



发送请求：谓词为DELETE，语义为删除ID为1的Person资源

收到响应：响应码204，执行成功



发送请求：谓词为GET，语义为查询ID为1的Person资源，Accept标明希望接收到Json数据

收到响应：响应码为200，执行成功，响应内容为null，资源已删除

这就是我用Fiddler来发送、调用的一组RestFul接口，大家可以看到，整个调用过程使用到了HTTP的语义，用到了谓词路由、内容协商。在增、删、改操作中，我都是使用void作为返回值，根据HTTP Code 判断，大家也可以自定义一些返回数据来做出更进一步的操作描述。

在写了这些API后，我们需要在程序中调用，我以C#为例写一组对这些接口调用的实现。在C#中，传统调用HTTP接口一般有两种办法： WebRequest/WebResponse组合的方法调用和WebClient类进行调用。第一种方法抽象程度较低，使用较为繁琐；而WebClient主要面向了WEB网页场景，在模拟Web操作时使用较为方便，但用在RestFul场景下却比较麻烦，在Web API发布的同时，.NET提供了两个程序集：System.Net.Http和System.Net.Http.Formatting。这两个程序集中最核心的类是HttpClient。在.NET4.5中带有这两个程序集，而.NET4需要到Nuget里下载Microsoft.Net.Http和Microsoft.AspNet.WebApi.Client这两个包才能使用这个类，更低的.NET版本就只能表示遗憾了只能用WebRequest/WebResponse或者WebClient来调用这些API了。

       在使用中，System.Net.Http这个程序集提供了HttpClient类以及相关的HTTP调用，而System.Net.Http.Formatting提供了一些针对HttpClient的帮助扩展，更好地支持了内容协商、Content创建等功能。下面我就和大家一起写一下这个例子：

       我们新建一个控制台程序：

       代码如下：

[复制代码](javascript:void(0);)

public class Person

{

public long Id { get; set; }

public string Name { get; set; }

public int Age { get; set; }

public string Sex { get; set; }

public override string ToString()

{

return $"Id={Id} Name={Name} Age={Age} Sex={Sex}";

}

}

class Program

{

static void Main(string[] args)

{

var client = new HttpClient();

client.BaseAddress = new Uri("http://localhost:22658/"); //基本的API URL

client.DefaultRequestHeaders.Accept.Add(new MediaTypeWithQualityHeaderValue("application/json")); //默认希望响应使用Json序列化

Run(client);

Console.ReadLine();

}

static async void Run(HttpClient client)

{

var result = await AddPerson(client);

Console.WriteLine($"添加结果：{result}"); //添加结果：true

var person = await GetPerson(client);

Console.WriteLine($"查询结果：{person}"); //查询结果：Id=1 Name=test Age=10 Sex=F

result = await PutPerson(client);

Console.WriteLine($"更新结果：{result}"); //更新结果：true

result = await DeletePerson(client);

Console.WriteLine($"删除结果：{result}"); //删除结果：true

}

static async Task<bool> AddPerson(HttpClient client)

{

return await client.PostAsJsonAsync("api/Person", new Person() { Age = 10, Id = 1, Name = "test", Sex = "F" }) //向Person发送POST请求，Body使用Json进行序列化

.ContinueWith(x => x.Result.IsSuccessStatusCode); //返回请求是否执行成功，即HTTP Code是否为2XX

}

static async Task<Person> GetPerson(HttpClient client)

{

return await await client.GetAsync("api/Person/1") //向Person发送GET请求

.ContinueWith(x => x.Result.Content.ReadAsAsync<Person>( //获取返回Body，并根据返回的Content-Type自动匹配格式化器反序列化Body

new List<MediaTypeFormatter>() {new JsonMediaTypeFormatter()/\*这是Json的格式化器\*/

,new XmlMediaTypeFormatter()/\*这是XML的格式化器\*/}));

}

static async Task<bool> PutPerson(HttpClient client)

{

return await client.PutAsJsonAsync("api/Person/1", new Person() { Age = 10, Id = 1, Name = "test1Change", Sex = "F" }) //向Person发送PUT请求，Body使用Json进行序列化

.ContinueWith(x => x.Result.IsSuccessStatusCode); //返回请求是否执行成功，即HTTP Code是否为2XX

}

static async Task<bool> DeletePerson(HttpClient client)

{

return await client.DeleteAsync("api/Person/1") //向Person发送DELETE请求

.ContinueWith(x => x.Result.IsSuccessStatusCode); //返回请求是否执行成功，即HTTP Code是否为2XX

}

}

[复制代码](javascript:void(0);)

    这就完成了这组API的调用，是不是非常简单方便？HTTPClient使用全异步的方法，并且他有良好的扩展性，我会在之后的博客中再聊这个问题。

    OK，到此为止一组简单的Restful API和C#的调用客户端就完成了，但这只是开始，Web API是一个很强大的框架，他的扩展点非常丰富，这些扩展能为我们的开发提供很多的帮助，下一篇博文我将为大家带来WEB API中Filter的使用。

[**WEB API 系列（二） Filter的使用以及执行顺序**](https://www.cnblogs.com/UliiAn/p/5402146.html)

　　在WEB Api中，引入了面向切面编程（AOP）的思想，在某些特定的位置可以插入特定的Filter进行过程拦截处理。引入了这一机制可以更好地践行DRY(Don’t Repeat Yourself)思想，通过Filter能统一地对一些通用逻辑进行处理，如：权限校验、参数加解密、参数校验等方面我们都可以利用这一特性进行统一处理，今天我们来介绍Filter的开发、使用以及讨论他们的执行顺序。

一、Filter的开发和调用

         在默认的WebApi中，框架提供了三种Filter，他们的功能和运行条件如下表所示：

|  |  |  |
| --- | --- | --- |
| **Filter 类型** | **实现的接口** | **描述** |
| **Authorization** | IAuthorizationFilter | 最先运行的Filter，被用作请求权限校验 |
| **Action** | IActionFilter | 在Action运行的前、后运行 |
| **Exception** | IExceptionFilter | 当异常发生的时候运行 |

       首先，我们实现一个AuthorizatoinFilter可以用以简单的权限控制：

[复制代码](javascript:void(0);)

public class AuthFilterAttribute : AuthorizationFilterAttribute

{

public override void OnAuthorization(HttpActionContext actionContext)

{

//如果用户方位的Action带有AllowAnonymousAttribute，则不进行授权验证

if (actionContext.ActionDescriptor.GetCustomAttributes<AllowAnonymousAttribute>().Any())

{

return;

}

var verifyResult = actionContext.Request.Headers.Authorization!=null && //要求请求中需要带有Authorization头

actionContext.Request.Headers.Authorization.Parameter == "123456"; //并且Authorization参数为123456则验证通过

if (!verifyResult)

{

//如果验证不通过，则返回401错误，并且Body中写入错误原因

actionContext.Response = actionContext.Request.CreateErrorResponse(HttpStatusCode.Unauthorized,new HttpError("Token 不正确"));

}

}  
 }

[复制代码](javascript:void(0);)

    一个简单的用于用户验证的Filter就开发完了，这个Filter要求用户的请求中带有Authorization头并且参数为123456，如果通过则放行，不通过则返回401错误，并在Content中提示Token不正确。下面，我们需要注册这个Filter，注册Filter有三种方法：

第一种：在我们希望进行权限控制的Action上打上AuthFilterAttribute这个Attribute:

[复制代码](javascript:void(0);)

public class PersonController : ApiController

{

[AuthFilter]

public CreateResult Post(CreateUser user)

{

return new CreateResult() {Id = "123"};

}

}

[复制代码](javascript:void(0);)

这种方式适合单个Action的权限控制。

第二种，找到相应的Controller，并打上这个Attribute：

[复制代码](javascript:void(0);)

[AuthFilter]

public class PersonController : ApiController

{

public CreateResult Post(CreateUser user)

{

return new CreateResult() {Id = "123"};

}

}

[复制代码](javascript:void(0);)

这种方式适合于控制整个Controller，打上这个Attribute以后，整个Controller里所有Action都获得了权限控制。

第三种，找到App\_Start\WebApiConfig.cs，在Register方法下加入Filter实例：

[复制代码](javascript:void(0);)

public static void Register(HttpConfiguration config)

{  
 config.MapHttpAttributeRoutes();  
　　 //注册全局Filter

config.Filters.Add(new AuthFilterAttribute());

config.Routes.MapHttpRoute(

name: "DefaultApi",

routeTemplate: "api/{controller}/{id}",

defaults: new { id = RouteParameter.Optional }

);

}

[复制代码](javascript:void(0);)

用这种方式适合于控制所有的API，任意Controller和任意Action都接受了这个权限控制。

在大多数场景中，每个API的权限验证逻辑都是一样的，在这样的前提下使用全局注册Filter的方法最为简单便捷，可这样存在一个显而易见的问题：如果某几个API是不需要控制的（例如登录）怎么办？我们可以在这样的API上做这样的处理：

[AllowAnonymous]

public CreateResult PostLogin(LoginEntity entity)

{

//TODO:添加验证逻辑

return new CreateResult() {Id = "123456"};

}

我为这个Action打上了AllowAnonymousAttribute，验证逻辑就放过了这个API而不进行权限校验。

    在实际的开发中，我们可以设计一套类似Session的机制，通过用户登录来获取Token，在之后的交互HTTP请求中加上Authorization头并带上这个Token，并在自定义的AuthFilterAttribute中对Token进行验证，一套标准的Token验证流程就可以实现了。

    接下来我们介绍ActionFilter:

　　ActionFilterAttrubute提供了两个方法进行拦截：OnActionExecuting和OnActionExecuted，他们都提供了同步和异步的方法。

　　OnActionExecuting方法在Action执行之前执行，OnActionExecuted方法在Action执行完成之后执行。

　　我们来看一个应用场景：使用过MVC的同学一定不陌生MVC的模型绑定和模型校验，使用起来非常方便，定义好Entity之后，在需要进行校验的地方可以打上相应的Attribute，在Action开始时检查ModelState的IsValid属性，如果校验不通过直接返回View，前端可以解析并显示未通过校验的原因。而Web API中也继承了这一方便的特性，使用起来更加方便：

[复制代码](javascript:void(0);)

public class CustomActionFilterAttribute : ActionFilterAttribute

{

public override void OnActionExecuting(HttpActionContext actionContext)

{

if (!actionContext.ModelState.IsValid)

{

actionContext.Response = actionContext.Request.CreateErrorResponse(HttpStatusCode.BadRequest, actionContext.ModelState);

}

}

}

[复制代码](javascript:void(0);)

    这个Filter就提供了模型校验的功能，如果未通过模型校验则返回400错误，并把相关的错误信息交给调用者。他的使用方法和AuthFilterAttribute一样，可以针对Action、Controller、全局使用。我们可以用下面一个例子来验证：

代码如下：

[复制代码](javascript:void(0);)

public class LoginEntity

{

[Required(ErrorMessage = "缺少用户名")]

public string UserName { get; set; }

[Required(ErrorMessage = "缺少密码")]

public string Password { get; set; }

}

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

[AllowAnonymous]

[CustomActionFilter]

public CreateResult PostLogin(LoginEntity entity)

{

//TODO:添加验证逻辑

return new CreateResult() {Id = "123456"};

}

[复制代码](javascript:void(0);)



当然，你也可以根据自己的需要解析ModelState然后用自己的格式将错误信息通过Request.CreateResponse()返回给用户。

　　OnActionExecuted方法我在实际工作中使用得较少，目前仅在一次部分响应数据加密的场景下进行过使用，使用方法一样，读取已有的响应，并加密后再给出加密后的响应赋值给actionContext.Response即可。

我给大家一个Demo:

[复制代码](javascript:void(0);)

public override async Task OnActionExecutedAsync(HttpActionExecutedContext actionExecutedContext, CancellationToken cancellationToken)

{

var key = 10;

var responseBody = await actionExecutedContext.Response.Content.ReadAsByteArrayAsync(); //以Byte数组方式读取Content中的数据

for (int i = 0; i < responseBody.Length; i++)

{

responseBody[i] = (byte)(responseBody[i] ^ key); //对每一个Byte做异或运算

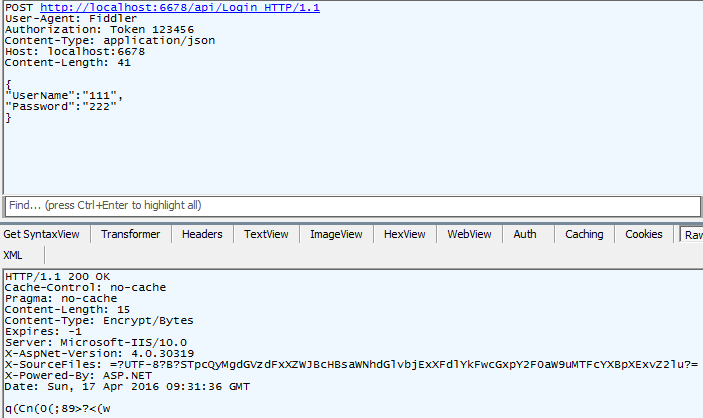
}

actionExecutedContext.Response.Content = new ByteArrayContent(responseBody); //将结果赋值给Response的Content

actionExecutedContext.Response.Content.Headers.ContentType = MediaTypeHeaderValue.Parse("Encrypt/Bytes"); //并修改Content-Type

}

[复制代码](javascript:void(0);)



　　通过这个方法我们将响应的Content每个Byte都做了一个异或运算，对响应内容进行了一次简单的加密，大家可以根据自己的需要进行更可靠的加密，如AES、DES或者RSA…通过这个方法可以灵活地对某个Action的处理后的结果进行处理，通过Filter进行响应内容加密有很强的灵活性和通用性，他能获取当前Action的很多信息，然后根据这些信息选择加密的方式、获取加密所需的参数等等。如果加密所使用参数对当前执行的Action没有依赖，也可以采取HttpMessageHandler来进行处理，在之后的教程中我会进行介绍。

    最后一个Filter：ExceptionFilter

    顾名思义，这个Filter是用来进行异常处理的，当业务发生未处理的异常，我们是不希望用户接收到黄页或者其他用户无法解析的信息的，我们可以使用ExceptionFilter来进行统一处理：

[复制代码](javascript:void(0);)

public class ExceptionFilter : ExceptionFilterAttribute

{

public override void OnException(HttpActionExecutedContext actionExecutedContext)

{

//如果截获异常为我们自定义，可以处理的异常则通过我们自己的规则处理

if (actionExecutedContext.Exception is DemoException)

{

//TODO:记录日志

actionExecutedContext.Response = actionExecutedContext.Request.CreateResponse(

HttpStatusCode.BadRequest, new {Message = actionExecutedContext.Exception.Message});

}

else

{

//如果截获异常是我没无法预料的异常，则将通用的返回信息返回给用户，避免泄露过多信息，也便于用户处理

//TODO:记录日志

actionExecutedContext.Response =

actionExecutedContext.Request.CreateResponse(HttpStatusCode.InternalServerError,

new {Message = "服务器被外星人拐跑了！"});

}

}

}

[复制代码](javascript:void(0);)

    我们定义了一个ExceptoinFilter用于处理未捕获的异常，我们将异常分为两类：一类是我们可以预料的异常：如业务参数错误，越权等业务异常；还有一类是我们无法预料的异常：如数据库连接断开、内存溢出等异常。我们通过HTTP Code告知调用者以及用相对固定、友好的数据结构将异常信息告诉调用者，以便于调用者记录并处理这样的异常。

[复制代码](javascript:void(0);)

[CustomerExceptionFilter]

public class TestController : ApiController

{

public int Get(int a, int b)

{

if (a < b)

{

throw new DemoException("A必须要比B大！");

}

if (a == b)

{

throw new NotImplementedException();

}

return a\*b;

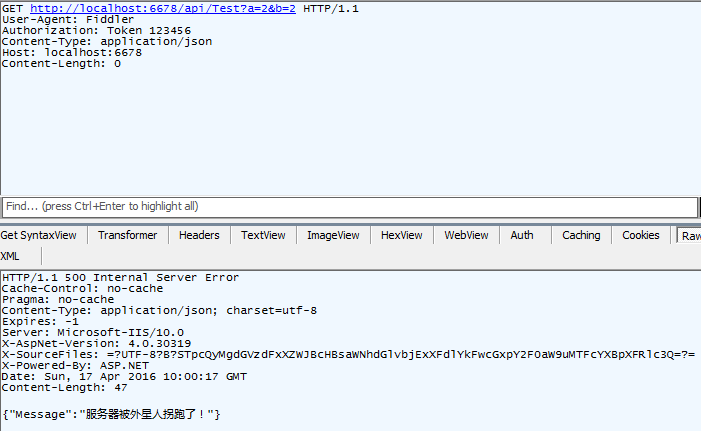
}

}

[复制代码](javascript:void(0);)

    我们定义了一个Action：在不同的情况下会抛出不同的异常，其中一个异常是我们能够预料并认为是调用者传参出错的，一个是不能够处理的，我们看一下结果：





    在这样的RestApi中，我们可以预先定义好异常的表现形式，让调用者可以方便地判断什么情况下是出现异常了，然后通过较为统一的异常信息返回方式让调用者方便地解析异常信息，形成统一方便的异常消息处理机制。

**但是，ExceptionFilter只能在成功完成了Controller的初始化以后才能起到捕获、处理异常的作用，而在Controller初始化完成之前（例如在Controller的构造函数中出现了异常）则ExceptionFilter无能为力。对此WebApi引入了ExceptionLogger和ExceptionHandler处理机制，我们将在之后的文章中进行讲解。**

二、Filter的执行顺序

    在使用MVC的时候，ActionFilter提供了一个Order属性，用户可以根据这个属性控制Filter的调用顺序，而Web API却不再支持该属性。Web API的Filter有自己的一套调用顺序规则：

    所有Filter根据注册位置的不同拥有三种作用域：Global、Controller、Action：

通过HttpConfiguration类实例下Filters.Add()方法注册的Filter（一般在App\_Start\WebApiConfig.cs文件中的Register方法中设置）就属于Global作用域；

通过Controller上打的Attribute进行注册的Filter就属于Controller作用域；

通过Action上打的Attribute进行注册的Filter就属于Action作用域；

他们遵循了以下规则：

1、在同一作用域下，AuthorizationFilter最先执行，之后执行ActionFilter

2、对于AuthorizationFilter和ActionFilter.OnActionExcuting来说，如果一个请求的生命周期中有多个Filter的话，执行顺序都是Global->Controller->Action；

3、对于ActionFilter，OnActionExecuting总是先于OnActionExecuted执行；

4、对于ExceptionFilter和ActionFilter.OnActionExcuted而言执行顺序为Action->Controller->Global；

5、对于所有Filter来说，如果阻止了请求:即对Response进行了赋值，则后续的Filter不再执行。

关于默认情况下的Filter相关知识我们就讲这么一些，如果在文章中有任何不正确的地方或者疑问，欢迎大家为我指出。

[**Web API系列（三） 异常处理**](https://www.cnblogs.com/UliiAn/p/5440746.html)

         在上一篇教程中我为大家介绍了Web API中Filter的开发使用，其中讲到ExceptionFilter时留了一个坑：ExceptionFilter只能截获并处理Action执行过程中发生的异常，在Action执行过程之外如果出现异常，ExceptionFilter是无能为力的。

      这些异常包括：

　　1、  Controller构造方法中出现的异常

　　2、  MessageHandlers中出现的异常

　　3、  路由过程中出现的异常

　　4、  Body在序列化/反序列化过程中出现的异常

　　由此可以看出，ExceptionFilter只能解决ApiControler成功实例化后并执行Action期间出现的异常；为了解决这一个问题，在WEB API中除了ExceptionFilter外还引入了两个针对异常记录、处理的扩展点：

IExceptionLogger 和IExceptionHandler。

         而这两个扩展是作为Web API的管道组件进行注册管理的，并且，他们有不同的分工：

         IExceptionLogger作为异常日志记录组件，负责异常发生后的日志记录，他贯穿于整个Web API的生命周期中，在Web API框架里，任何一个请求周期中出现任何一个未被捕获/处理的异常都会首先进入这个异常日志记录管道进行异常Log记录，在Web API中可以注册多个IExceptionLogger实例负责不同的异常处理。

         IExceptionHandler作为异常处理组件，负责异常发生后的处理工作，他处于异常处理管道的最末端，当IExceptionLogger组件进行一场记录完毕、没有相关的ExceptoinFilter进行异常处理时，才会最终调用ExceptionHandler进行异常处理，在Web API中，有且仅有一个ExceptionHandler进行异常的处理。

         在Web API框架中给出了两个基类：ExceptionLogger和ExceptionHandler，在使用ExceptionLogger基类时，他提供了ShouldLog虚方法，该方法在基类中被调用，其作用在于避免同一个异常被同一个ExceptionLogger实例重复记录（如当后续的管道中该异常又被抛出，或者同一个ExceptionLogger对象不小心被注册了两次就会出现重复记录的可能）我们也能复写ShouldLog方法加入我们自己的异常记录判断逻辑以针对不同的场景进行不同的ExceptionLogger调用。如果有兴趣可以反编译一下ExceptionLogger基类看看，他使用了显示接口实现，挺有意思的一个技巧。下面我们来看一个ExceptionLogger使用的例子：

[复制代码](javascript:void(0);)

public class ErroLogger : ExceptionLogger

{

public async Task LogAsync(ExceptionLoggerContext context, CancellationToken cancellationToken)

{

var sb = new StringBuilder();

//获取Log组件

ILogger log = LogManager.GetCurrentClassLogger();

var request = context.Request;

sb.AppendLine("URL:");

//获取URL

var url = request.RequestUri.ToString();

sb.AppendLine(url);

log.Error(context.Exception,sb.ToString(),"");

}

public override bool ShouldLog(ExceptionLoggerContext context)

{

return context.Exception is DemoException && base.ShouldLog(context);

}

}

[复制代码](javascript:void(0);)

    在这个例子中，我们重写了ShouldLog，保证了这个ExceptionLogger只记录DemoException这个类型的异常，并且也调用了基类方法，保证不会重复记录同一个异常。在LogAsync方法中，我通过Log组件记录了导致异常的请求URL，也记录了异常信息。

接下来我们要对这个组件进行注册：

在App\_Start/WebApiConfig.cs文件中的Register方法中写入

config.Services.Add(typeof(IExceptionLogger),new ErroLogger());

    这样，一个针对DemoException的异常记录组件就开发完成并注册完成了，当Web API执行管道中出现未处理的DemoException异常，均会调用则个组件进行记录。

    接下来我们来写一个ExceptionHandler，在整个Web API框架中，ExceptionHandler只能提供一个实例，与ExceptionLogger一样，我们可以继承ExceptionHandler基类来简化异常处理，在ExceptionHandler中也提供了ShouldHandle方法来判断该异常是否应该处理，避免重复处理管道中其他环节重复抛出的异常。我们也同样提供一个例子：

[复制代码](javascript:void(0);)

public class ErrorHandler : ExceptionHandler

{

public override async Task HandleAsync(ExceptionHandlerContext context, CancellationToken cancellationToken)

{

if (context.Exception is DemoException)

{

context.Result = new ResponseMessageResult(context.Request.CreateResponse(HttpStatusCode.BadRequest,new {Message=context.Exception.Message}));

}

else

{

context.Result = new ResponseMessageResult(context.Request.CreateResponse(HttpStatusCode.InternalServerError,new {Message = "服务器已被外星人绑架"}));

}

}

}

[复制代码](javascript:void(0);)

    在这个例子中，我们判断了异常的类型，并根据不同的异常返回客户端不同的响应内容和不同的HTTP状态码。

    然后在配置中注册这个异常处理模块，在App\_Start/WebApiConfig.cs文件中的Register方法中写入

config.Services.Replace(typeof(IExceptionHandler),new ErrorHandler());

    这样就替换了系统默认的ExceptionHandler，可以使用我们自定义的Handler进行异常的处理了。

    在异常记录、处理过程中，我们都碰到相应的异常上下文参数，我们能通过这个参数获取当前请求的上下文，获取请求、响应（小心有时会为空哦）、捕获到该异常的catch块信息等内容，我们可以利用这些信息更好地描述、记录、处理异常。

    到这里ExceptionLogger组件和ExceptionHandler组件简单的开发就完成了。在开发的过程中我们可以看到，ExceptionLogger负责了全局的异常记录，在Web API框架管道下出现未处理的异常ExceptionLogger都会进行捕获、记录。而ExceptionHandler和ExceptionFilter功能是有重叠的，那何时使用ExceptionHandler何时使用ExceptionFilter呢？我们可以将两者的区别列出如下的表格:

|  |  |  |
| --- | --- | --- |
|  | ExceptionFilter | ExceptionHandler |
| 作用域 | Controller、Action | 全局 |
| 实例个数 | 无限制 | 全局唯一 |
| 作用条件 | Controller实例化成功之后 | Web API成功加载之后 |

　　经过上面的表我们可以看出，如果处理颗粒度细致到Controller、Action级别时，ExceptionFilter处理起来会更得心应手，他已经能精确定位到某个Action，然后可以针对当前Action做定制开发。而ExceptionHandler作用域远大于ExceptionFilter，他处理全局更有优势。

　　关于Web API异常处理的相关内容我就讲这么一些，如果在文章中有任何不正确的地方或者疑问，欢迎大家为我指出