**<https://www.cnblogs.com/CreateMyself/p/4813004.html>**

**[Web APi入门之Self-Host寄宿及路由原理（二）](http://www.cnblogs.com/CreateMyself/p/4813004.html)**

**前言**

刚开始表面上感觉Web API内容似乎没什么，也就是返回JSON数据，事实上远非我所想，不去研究不知道，其中的水还是比较深，那又如何，一步一个脚印来学习都将迎刃而解。

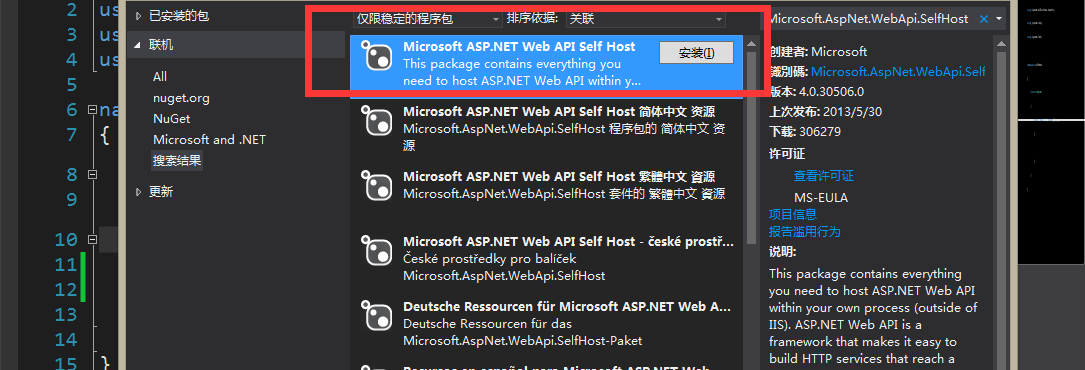
**Self-Host**

**我们知道Web API它可以快速为HTTP客户端提供API来创建Web服务，为何如此这样说呢？因为我们可以将其作为主机也就是一个服务器来用完全不需要IIS，这就是我们下面要讲的第一个内容Self-Host，实现对Web API寄宿的方式有多种并且都是独立于ASP.NET框架之外，如下Self-Host寄宿是存在于Web API 1中的，而在Web API 2中实现寄宿是采用Web-Host来进行寄宿（通过程序包packages中的Web-Host以及Web Client可得知），因为Web API本身是无法提供【请求-响应】的机制，所以需要寄宿来实现，即通过具体的应用程序来为Web API运行提供一个环境。下面且听我娓娓道来。既然是进行交互必然有服务器和客户端，下面我们将从建立控制台应用程序开始来进行了解。**

**Web API服务器**

**第一步**

**建立一个SelfHost的控制台应用程序，添加【Microsoft.AspNet.WebApi.SelfHost】程序包，搜索时会出现多个包注意不是【AspNetWebApi.SelfHost.】如下：**



**第二步**

**添加类，如下：**

[复制代码](javascript:void(0);)

public class Product

{

public int Id { get; set; }

public string Name { get; set; }

public string Category { get; set; }

public decimal Price { get; set; }

}

[复制代码](javascript:void(0);)

**第三步**

**添加派生自APiController控制的类以及要演示的数据，如下：**

[复制代码](javascript:void(0);)

public class ProductsController : ApiController

{

Product[] products = new Product[]

{

new Product { Id = 1, Name = "Tomato Soup", Category = "Groceries", Price = 1 },

new Product { Id = 2, Name = "Yo-yo", Category = "Toys", Price = 3.75M },

new Product { Id = 3, Name = "Hammer", Category = "Hardware", Price = 16.99M }

};

public IEnumerable<Product> GetAllProducts()

{

return products;

}

public Product GetProductById(int id)

{

var product = products.FirstOrDefault((p) => p.Id == id);

if (product == null)

{

throw new HttpResponseException(HttpStatusCode.NotFound); //如果未找到数据并返回状态码404

}

return product;

}

public IEnumerable<Product> GetProductsByCategory(string category)

{

return products.Where(p => string.Equals(p.Category, category,

StringComparison.OrdinalIgnoreCase));

}

}

[复制代码](javascript:void(0);)

**第四步**

**在控制台主程序中配置服务器以及添加路由**

[复制代码](javascript:void(0);)

var config = new HttpSelfHostConfiguration("http://localhost:8080"); //配置主机

config.Routes.MapHttpRoute( //配置路由

"API Default", "api/{controller}/{id}",

new { id = RouteParameter.Optional });

using (HttpSelfHostServer server = new HttpSelfHostServer(config)) //监听HTTP

{

server.OpenAsync().Wait(); //开启来自客户端的请求

Console.WriteLine("Press Enter to quit.");

Console.ReadLine();

}

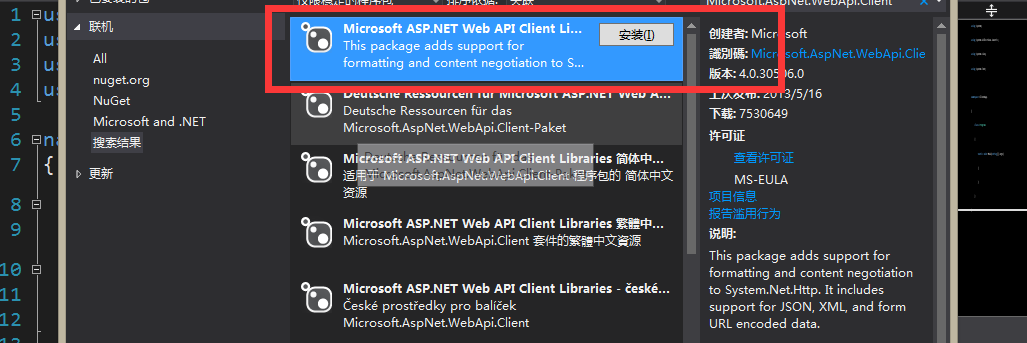
[复制代码](javascript:void(0);)

以上就是关于Web API关于主机的设置，接下来就是建立一个客户端来访问此服务器上的资源。

**Web API客户端**

**第一步**

**同理建立一个ClientApp的控制台应用程序，同时添加Web API客户端程序包【Microsoft.AspNet.WebApi.Client】如下：**



**第二步**

**既然是要访问服务器上的资源，自然要添加对Web API服务器（SelfHost）的引用了。**

**第三步**

**接下来就是建立客户端并读取服务器上的资源**

[复制代码](javascript:void(0);)

static HttpClient client = new HttpClient(); //利用此对象进行对Web API的调用

static void ListAllProducts()

{

HttpResponseMessage resp = client.GetAsync("api/products").Result;

resp.EnsureSuccessStatusCode();

var products = resp.Content.ReadAsAsync<IEnumerable<SelfHost.Product>>().Result;

foreach (var p in products)

{

Console.WriteLine("{0} {1} {2} ({3})", p.Id, p.Name, p.Price, p.Category);

}

}

static void ListProduct(int id)

{

var resp = client.GetAsync(string.Format("api/products/{0}", id)).Result;

resp.EnsureSuccessStatusCode();

var product = resp.Content.ReadAsAsync<SelfHost.Product>().Result;

Console.WriteLine("ID {0}: {1}", id, product.Name);

}

static void ListProducts(string category)

{

Console.WriteLine("Products in '{0}':", category);

string query = string.Format("api/products?category={0}", category);

var resp = client.GetAsync(query).Result;

resp.EnsureSuccessStatusCode();

var products = resp.Content.ReadAsAsync<IEnumerable<SelfHost.Product>>().Result;

foreach (var product in products)

{

Console.WriteLine(product.Name);

}

}

[复制代码](javascript:void(0);)

* **通过调用HttpClient.GetAsync来发出一个Get请求来请求服务器上的Uri资源。**
* **通过调用HttpResponseMessage.EnsureSuccessStatusCode方法来确定请求是否成功，若失败即返回错误状态码则抛出一个异常。**
* **通过调用ReadAsAsync<T>将HTTP中响应的数据类型进行反序列化，该方法为一个扩展方法。**

GetAsync和ReadAsync为异步方法，直到获得结果值即Result属性的值操作完成，否则将一直阻塞线程。

**第四步**

**最后在客户端控制台主程序中建立客户端与服务器端的通信服务即可**

[复制代码](javascript:void(0);)

client.BaseAddress = new Uri("http://localhost:8080");

ListAllProducts();

ListProduct(1);

ListProducts("toys");

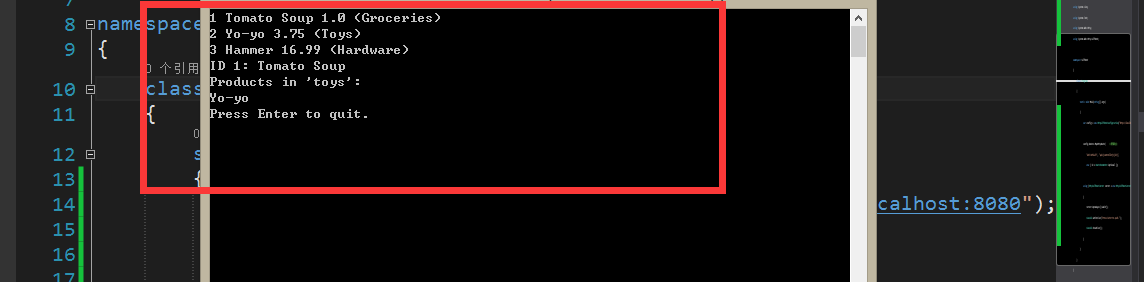
Console.WriteLine("Press Enter to quit.");

Console.ReadLine();

[复制代码](javascript:void(0);)

接下来就是启动Web API服务器程序，通过Web API客户端来访问服务器并获得其请求的资源。【注意】windows 8系统启动服务器必须以管理员身份运行，否则报错。

访问资源成功，如下：



**路由原理**

 如果对MVC框架中路由熟悉的话，Web API的路由原理和其相似，但是不同的是Web API是使用的HTTP方法来选择Action方法而不是通过URI路径来选择Action方法。

**路由表**

**在Web API中处理HTTP请求的是一个控制器类，控制器中的公有方法叫做Action方法，当Web API框架获得一个请求时，会根据请求路由到一个Action方法上。为了决定调用哪个Action方法，Web API框架利用路由表来决定，当创建项目模板时将在App\_Start文件下的 WebApiConfig 创建一个默认的路由，如下：**

config.Routes.MapHttpRoute(

name: "DefaultApi",

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

defaults: new { id = RouteParameter.Optional }

);

当我们需要将Web API作为服务器时此时必须直接在HttpSelfHostConfiguration上设置路由表【上述关于SelfHost已经演示】

**注册路由**

在Web API路由配置文件中有一个 Register 方法，该方法中有一个 HttpConfiguration 实例参数，该HttpConfiguration是消息管道中的全局对象，我们可以通过其对管道某一个行为作出相关操作，从而达到我们所需。通过HttpConfiguration上的属性\_routes即 HttpRouteCollection 对象中的 CreateRoute 方法再依据默认的路由模板、默认值以及相关约束来创建实现了 IHttpRoute 接口的路由对象即 HttpRoute ，并通过HttpConfiguration中的Routes对象中的扩展方法MapHttpRoute方法来达到注册路由映射的目的，当然我们也可以直接通过调用路由集合HttpRouteCollection中的 Add 方法来注册路由到路由表中。

**路由表中每一条都包含一个对应的路由模板，Web API默认的路由模板是api/{controller}/{id} ，在此模板下，api是一个路径段，而{controller}和{id}是占位符。当Web API收到HTTP请求时，它尝试去匹配路由表中的路由模板之一的URI，若没匹配上，则返回404。**

例如，用下面的URI来匹配默认的路由

/api/contacts

/api/contacts/1

/api/products/gizmo1

【注意】为何在Web API的控制器中以API开头的原因是避免和MVC框架中的路由冲突，如/contacts会进入到MVC路由中，而/api/contacts会进入到Web API框架中，当然我们可以通过改变默认的路由表来改变约定，但是不建议这么做。

一旦一个匹配的路由被找到，Web API根据默认约定来选择Controller和Action：

* **Web API会用Controller替代占位符{controller}的值中来找到控制器。**
* **Web API着眼根据HTTP方法来找到Action方法，然后找到一个以一个HTTP方法的名称开始的Action方法，例如，对于一个Get请求，Web API会找到一以Get开头的Action方法，如：GetContact或者GetAllContact的Action方法。这种约定同样也适用于GET、POST、PUT以及DELETE方法。我们通过在控制器上使用特性来启用其他的方法。**
* **在路由模板中的其他的占位符，如{id}被映射到action方法参数上。**

**查找控制器**

当一个URI资源请求过来时Web API框架会查找已经注册的路由表中的路由并进行匹配，如果匹配通过，此时将创建一个包含每个占位符的值的字典，字典的键为占位符的名称，当然不包括大括号，字典的值为请求过来的URI中的值或者是路由模板中的默认值，然后这个字典将会保存在 IHttpRouteData 对象中的字典中。该对象还存在一个路由对象IHttpRoute。当请求的URI过来时，此时会通过请求信息即 HttpRequestMessage  对象中的 RquestUri 属性来获得其URI，接下来HttpRoute会接受到HttpRequestMessage中的URI并进行解析，然后将通过调用 GetRouteData 等方法来封装路由数据给实现了IHttpRouteData接口的 HttpRouteData ，通过HttpRouteData中的构造函数中的路由对象来获取传递过来的路由对象HttpRoute（该路由对象也根据请求过来的路由变量来绑定到路由模板中最终生成一个完整的URL），同时因为实现了IHttpRouteData接口则此时该接口的字典将传递给HttpRouteData构造函数中的路由字典即 HttpRouteValueDictionary 。【注意】GetRouteData方法中的参数有一个为virtualPathRoot即虚拟根路径，当执行此方法时得到的是相对路径，也就是说通过路由模板进行匹配是根据相对路径来进行匹配的。

查找控制器是通过 IHttpControllerSelector 接口上的 SelectController  ，此方法参数为HttpRequestMessage的实例并返回一个 HttpControllerDescriptor ，此接口的默认实现是通过 DefaultHttpControllerSelector  类实现，此类实现的算法有如下三点：

* **在上述路由字典即HttpRouteValueDictionary中查找controller的键。**
* **获取键的值并将字符串Controller作为控制器的类型名。**
* **用此类型名来查找一个Web API控制器。**

**查找Action**

在查找到控制器之后，框架将会通过 IHttpActionSelector 接口中的 SelectAction 方法来查找Action方法，该方法参数要获取一个控制器上下文即 HttpControllerContext 返回一个 HttpActionDescriptor 对象实例。其默认实现是通过 APiControllerActionSelector 来提供的，要查找到Action要经过如下三点：

* **请求的HTTP方法。**
* **在路由模板中占位符{action}对应的值。**
* **在控制器上Action方法的参数。**

【注意】如何确定在控制上的方法是Action方法呢？当查找Action方法时，仅仅只着眼于控制上的公有的实例方法（不包括从APiController上继承的特殊名称的方法，如：重载、事件、构造等等）并且是从APiController类上继承的方法。

**总结**

综上所述，对于Web API上的一个路由系统总共有三个阶段

* **匹配一个URI到一个路由模板。**
* **选择一个控制器。**
* **选择一个Action方法。**

当然你可以对路由模板进行自定义以及相关参数利用正则表达式进行约束等，这就不再详细描述。

接下来我们就上述叙述来进行相关例子

[复制代码](javascript:void(0);)

public class ProductsController : ApiController

{

public void GetAllProducts() { }

public IEnumerable<Product> GetProductById(int id) { }

public HttpResponseMessage DeleteProduct(int id){ }

}

[复制代码](javascript:void(0);)

对于以下可能的HTTP请求，会对于每个请求对应应该被调用的Action方法。

|  |  |  |  |
| --- | --- | --- | --- |
| HTTP Method | URI PATH | Action | Parameter |
| GET | api/products | GetAllProducts | (none) |
| GET | api/products/4 | GetProductById | 4 |
| DELETE | api/products/4 | DeleteProduct | 4 |
| POST | api/products | (no match) |  |

【注意】在上述中的URI的{id}，如果存在则会被映射到Action方法中的id参数中，如上述中两种Get方法，一个有id参数一个没有没有id参数。同时也应注意POST请求将会失败，因为在控制器中没有定义一个POST....方法。

**根据约定来进行映射Action方法注意**

假设Web API控制器名称ProductController并继承APiController，在该控制器下有如下两个方法：

[复制代码](javascript:void(0);)

public Product GetProductById(int id)

{

var product = products.FirstOrDefault((p) => p.Id == id);

if (product == null)

{

throw new HttpResponseException(HttpStatusCode.NotFound);

}

return product;

}

public string Get(int id)

{

return "value";

}

[复制代码](javascript:void(0);)

**当发出Get（如：/api/product/1）请求时将会出错，出错原因为匹配到多个路由，默认能够匹配到GetProductById方法就不用说了，此时同样能匹配到Get方法，所以此时需要将Get标记为【NonAction】或者采用其他方法来区别这两个路由。**

**HTTP方法**

除了使用约定的HTTP方法之外，我们也可以使用特性HttpGet、HttpPost、HttpPut以及HttpDelete来修饰Action方法来显式指定一个Action的HTTP方法。

**在下面的例子中，FindProduct方法会被映射到GET请求。**

public class ProductsController : ApiController

{

[HttpGet]

public Product FindProduct(id) {}

}

我们可以使用 AcceptVerbs 特性对一个Action使用多个HTTP方法或者说是使用HTTP方法而不是POST、DELETE、GET以及PUT，如下：

**多个请求到同一Action**

[复制代码](javascript:void(0);)

[AcceptVerbs("GET", "Post")]

public Product FindProduct(int id)

{

}

或者

[AcceptVerbs(HttpVerbs.Get | HttpVerbs.Post)]

public Product FindProduct(int id)

{

}

[复制代码](javascript:void(0);)

**根据请求不同来响应相同Action，如下：**

[复制代码](javascript:void(0);)

public Product FindProduct(int id)

{

}

[AcceptVerbs(HttpVerbs.Post]

public Product FindProduct(string guid) { }

[复制代码](javascript:void(0);)

**通过Action名称配置路由**

由于默认的路由模板，Web API使用HTTP方法来选择Action，但是我们可以创建一个包括在URI中的Action名称的路由。如下：

routes.MapHttpRoute(

name: "ActionApi",

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

defaults: new { id = RouteParameter.Optional }

);

在上述路由模板中，{action} 参数名称命名了在控制器上的Action方法，在此种路由下，可以使用特性来指定HTTP方法，例如，允许控制器有如下方法：

public class ProductsController : ApiController

{

[HttpGet]

public string Details(int id);

}

在此种情况下，对于api/products/details的GET请求会映射到Details方法，这种路由风格和MVC相似。

**通过使用ActionName特性来覆盖Action的名称，例如下列例子，有两种映射到如api/products/thumbnail/id的方法，一种支持GET请求，一种支持POST请求**

[复制代码](javascript:void(0);)

public class ProductsController : ApiController

{

[HttpGet]

[ActionName("Thumbnail")]

public HttpResponseMessage GetThumbnailImage(int id);

[HttpPost]

[ActionName("Thumbnail")]

public void AddThumbnailImage(int id);

}

[复制代码](javascript:void(0);)

**Non-Actions**

**为了阻止一个方法作为Action方法被调用，通过使用Non-Actions特性来将其标记为不是一个Action方法，即使这个方法匹配到了路由规则**

// Not an action method.

[NonAction]

public string GetPrivateData() { ... }