**基于.NET的Web API**

* **什么是Web API?**

HTTP不仅仅用于服务网页。HTTP也是构建服务和数据的API的强大平台。HTTP是简单、灵活、无所不在的。几乎任何你可以想到的平台都有一个HTTP库，因此HTTP服务可以达到广泛的客户端，包括浏览器、移动设备和传统桌面应用程序。

Web API可以为服务器应用程序提供对外的服务和数据访问接口。ASP.NET Web API是一种在.NET框架之上构建Web API的框架。

浏览器

Web API

Server

Mobile

Desktop APP

响应

请求

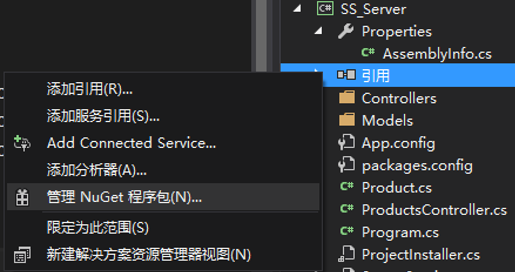
* **Web API有哪些用途？**

通过HTTP通讯协议，Web API可以提供一下的服务：

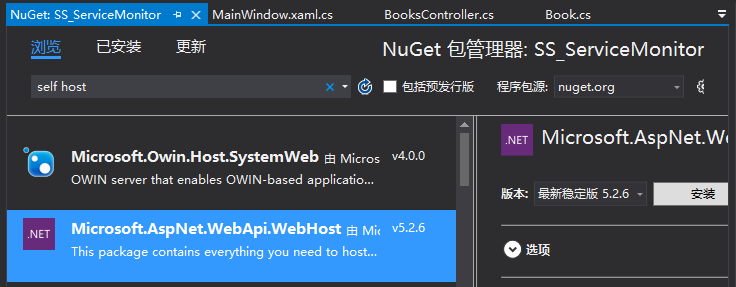
* 以GET方式获取数据
* 以POST方式发送和获取数据
* 以PUT方式添加数据
* 以DELETE方式删除数据
* **创建宿主于Window Service的Web API服务**

Web API可以宿主于任何的应用程序，无论的ASP.NET应用程序、WPF程序还是Window Service，甚至是控制台程序。因为Web API通常作为服务端对外开放的接口，理所应当要继承服务端稳定、长久的运行属性。因此，本教程介绍Web API的创建实例，并教授如何将其宿主于Window Service。关于如何创建使用Window Service应用程序，请详见“Window服务的创建和使用”章节，这里不再赘述。

* 下载并安装NuGet程序包



点击“管理NuGet程序包”，在打开的NuGet管理界面的搜索框中输入“SelfHost”，然后选择“Microsoft ASP.NET Web API 2.2 Self Host”，点击安装。

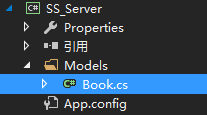


安装的NuGet程序包里面包含着用于HTTP协议传输的类，可用于Web API的开发，主要的命名空间为system.web.Http和system.web.Http.SelfHost。

* 添加Domain Models数据域模型

模型Model是表示应用程序中数据的对象。ASP.NET Web API可以自动将模型序列化为JSON、XML或其他格式，然后将序列化的数据写入HTTP响应消息的正文中。只要客户端可以读取序列化格式，就可以对对象进行反序列化。大多数客户端可以解析XML或JSON。此外，客户端可以通过在HTTP请求消息中设置Access头来指示它想要的格式。

在“解决方案资源管理器”中，创建Models文件夹，从上下文菜单中选择“添加”，然后选择“类”，并命名为“Book”。



用以下代码替换Book.cs中的代码：

namespace SS\_Server.Models

{

public class Book

{

public int Id { get; set; }

public string Author { get; set; }

public double Price { get; set; }

}

}

* 添加Web API Controller控制器

在Web API中，控制器是处理HTTP请求的对象。.NET通过Route路由，根据请求的API方式和参数，获取后台对应的控制器，并调用控制器的处理函数，返回处理后的数据。

**NOTE:**

**如果我们已经使用ASP.NET MVC，你已经熟悉了控制器。Web API控制器类似于MVC控制器，但继承了ApiController类，而不是Controller类。**

在“解决方案资源管理器”中，创建Controllers文件夹，从上下文菜单中选择“添加”，然后选择“类”，并命名为“BooksController”。

用以下代码替换BooksController.cs中的代码：

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Web.Http;

using SS\_Server.Models;

namespace SS\_Server.Controllers

{

public class BooksController : ApiController

{

Book[] books = new Book[]

{

new Book { Id = 0, Author = "Joke Fuli", Price = 13.23 },

new Book { Id = 1, Author = "Make", Price = 12.00 },

new Book { Id = 2, Author = "Boli", Price = 16.87 }

};

public Book[] GetAll()

{

return books;

}

}

}

* 添加Route路由

如果我们熟悉ASP.NET MVC，我们会发现Web API路由和MVC路由是很相似的。最大的不同是Web API使用HTTP方式，而不是URI路径来选择调用的行为。我们也可以在Web API里面使用MVC风格的路由，但这里只是讲述Web API路由。

* 路由表

在ASP.NET Web API中，控制器是用于处理HTTP请求的类。在类里面的public方法被称为行为方法或者单一行为。当Web API框架接收到一个请求时，它会将该请求路由给特定的行为。

为了明确哪个行为被调用，该框架使用一个路由表：

routes.MapHttpRoute(

name: "API Default",

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

defaults: new { id = RouteParameter.Optional }

);

如果是纯粹的Web API项目，VS会将该路由表保存在“App\_Start”目录里面名为“WebApiConfig.cs”的类文件。

因为本教程是使用宿主Web API，所以必须将路由表在HttpSelfHostConfiguration对象里面设置。

HttpSelfHostConfiguration config = new HttpSelfHostConfiguration("http://localhost:8080");

config.MapHttpRoute(

name: "API Default",

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

defaults: new { id = RouteParameter.Optional }

);

首先在Web API宿主对象的构造函数里面配置宿主IP。

接着，在默认路由表路由模板中，我们可以看到有三个变量：

* api:

api是路由模板中字面路径声明，它只是为了区别MVC 路由。我们通过/api/controller..和/controller..来区分是使用Web API还是使用MVC API来调用后台API。

* controller:

controller是占位符变量，它表示要路由的控制器，比如要路由控制器“BooksController”，则需要使用<http://localhost:8080/Books/...>调用API。

* id:

id也是占位符变量，它表示控制器中的Action方法的参数名称，它必需对应于行为方法中的参数，否则，.NET将无法路由到正确的Action，认为调用的API是不存在的。当然，我们可以使用“defaults: new { id = RouteParameter.Optional }”将id配置为可选配置项，因此，上面的路由表可以匹配两个同名Action，一个带id参数，一个无参。

路由表将引导宿主API对象去匹配使用哪种风格的API:MVC API or Web API，然后匹配哪个控制器controller，最后匹配基于参数的Action。但是，路由表还未能正确匹配Action，因为控制器里面有多个Action的参数一样时，比如以下两个Action:

public class BooksController : ApiController

{

Book[] books = new Book[]

{

new Book { Id = 0, Author = "Joke Fuli", Price = 13.23 },

new Book { Id = 1, Author = "Make", Price = 12.00 },

new Book { Id = 2, Author = "Boli", Price = 16.87 }

};

public Book[] GetAll() //Action

{

return books;

}

public Book PostOne() //Action

{

return books[0];

}

}

这两个Action都能匹配路由表的配置，所以，为了能够正确的调用想要的Action， 需要通过配合HTTP的请求方式加以区分。

* HTTP请求方式匹配Action

为了能够找到正确的Action，Web API会查看HTTP的请求方式：GET、POST、PUT、Delete，然后寻找那些以该请求方式开头的Action。例如，在一个GET请求中，Web API会寻找以“Get...”开头的Action，比如“GetOne”或者“GetAll”。这种惯例只适用于基于GET、POST、PUT、Delete的HTTP请求。当然，我们也可以通过配置属性路由来匹配其他HTTP方式的Action。

public class BooksController : ApiController

{

public Book[] GetAll() //Action

{

return books;

}

public Book[] PostAll() //Action

{

return books

}

public Book[] PutAll() //Action

{

return books;

}

public Book[] DeleteAll() //Action

{

return books[0];

}

}

上面的4个Action可以通过HTTP的请求方式进行匹配。实际上，匹配过程是api->controller->http method->params。

* 使用Atrribute属性匹配Action

习惯上，我们往往不以HTTP的method为开头命名Action，这个时候必须为Action添加特性。

[HttpGet]

public Book[] FindAll() //Action

我们可以让Action匹配其他HTTP method:

[AcceptVerbs("HEAD")]

public Book[] FindAll() //Action

当然，我们可以让同一个Action匹配多个HTTP method:

[AcceptVerbs("GET", "POST","HEAD","DELETE")]

public Book[] FindAll() //Action

* 通过Action的方法名进行路由

在前面讲述的路由过程中，我们使用的是默认的路由表，它通过HTTP的method来匹配Action。当然，我们也可以修改路由表，把Action的方法名做匹配条件。

config.MapHttpRoute(

name: "Action API ",

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

defaults: new { id = RouteParameter.Optional }

);

使用该模板时，在控制器里面的Action需要通过添加属性来设置匹配请求的方式：

[HttpGet]

Public string Details() //Action

而且，我们还可以重载Action方法，例如：

[HttpGet]

[ActionName("Details")]

Public string DetailsbyGET() //Action

[HttpPOST]

[ActionName("Details")]

Public string DetailsbyPOST() //Action

这样，我们可以使用“api/Books/Details”+ HTTP method来匹配Action。

* Non-Actions

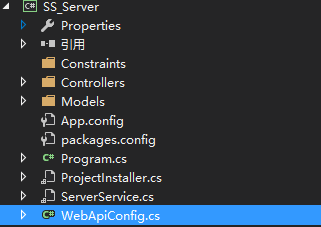
如果在控制器中定义的其他方法不是作为路由的Action，比如一个方法，存粹提供给其他Action调用，这里，我们需要为这个方法添加下面的属性来阻止其被路由匹配：

[NonAction]

Private string GetData() //Non-Action

* 添加Web API配置文件WebApiConfig.cs

当我们完成控制器的定义后，便可在WebApiConfig.cs文件中配置路由。



内容如下:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Web.Http;

namespace SS\_Server

{

public static class WebApiConfig

{

public static void Register(HttpConfiguration config)

{

// Attribute routing.

config.MapHttpAttributeRoutes();

// Convention-based routing.

config.Routes.MapHttpRoute(

name: "DefaultApi",

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

defaults: new { id = RouteParameter.Optional }

);

}

}

}

* 创建Web API宿主实例

最后，我们在Window Service的启动事件中创建Web API宿主实例，并让其异步监听客户端请求：

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Diagnostics;

using System.Linq;

using System.ServiceProcess;

using System.Text;

using System.Threading.Tasks;

using System.Xml;

using System.IO;

using Newtonsoft;

using System.Web.Http;

using System.Web.Http.SelfHost;

namespace SS\_Server

{

partial class ServerService : ServiceBase

{

private static HttpSelfHostServer server = null;

public ServerService()

{

InitializeComponent();

}

protected override void OnStart(string[] args)

{

// TODO: 在此处添加代码以启动服务。

HttpSelfHostConfiguration config = new

HttpSelfHostConfiguration("http://localhost:8080");

WebApiConfig.Register(config);

server = new HttpSelfHostServer(config);

server.OpenAsync().Wait();

}

protected override void OnStop()

{

//TODO: 在此处添加代码以执行停止服务所需的关闭操作。

}

}

}