**控制器基类 ControllerBase 和Controller的区别：**

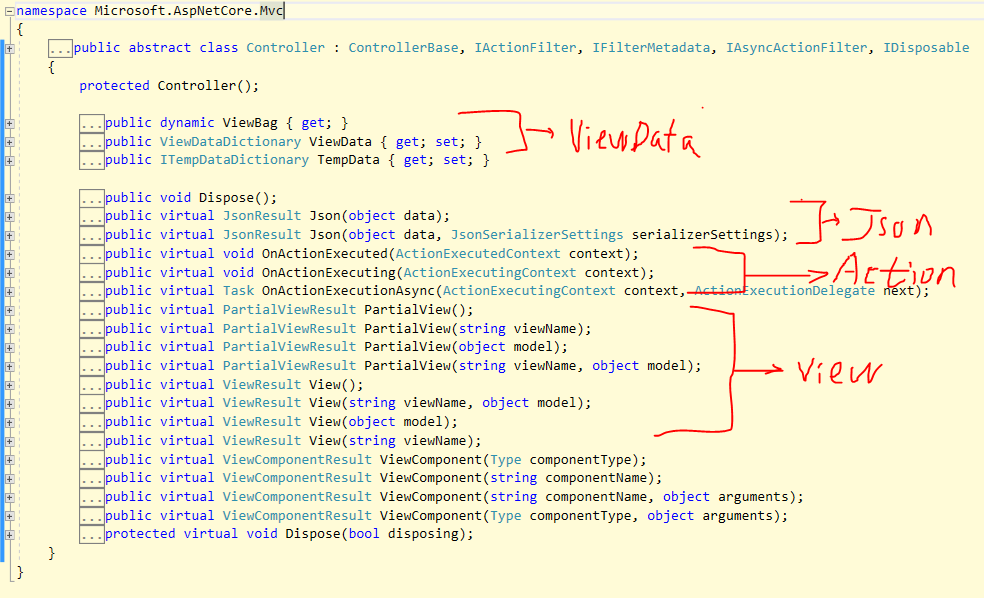
他们都来自于同一个命名空间：using Microsoft.AspNetCore.Mvc;

[**https://docs.microsoft.com/en-us/aspnet/overview**](https://docs.microsoft.com/en-us/aspnet/overview)

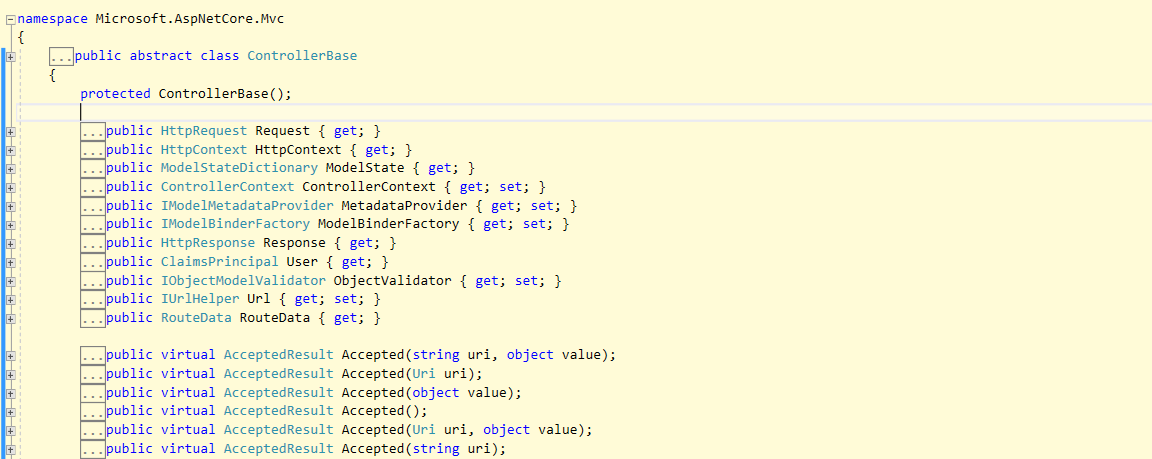
**Controller**

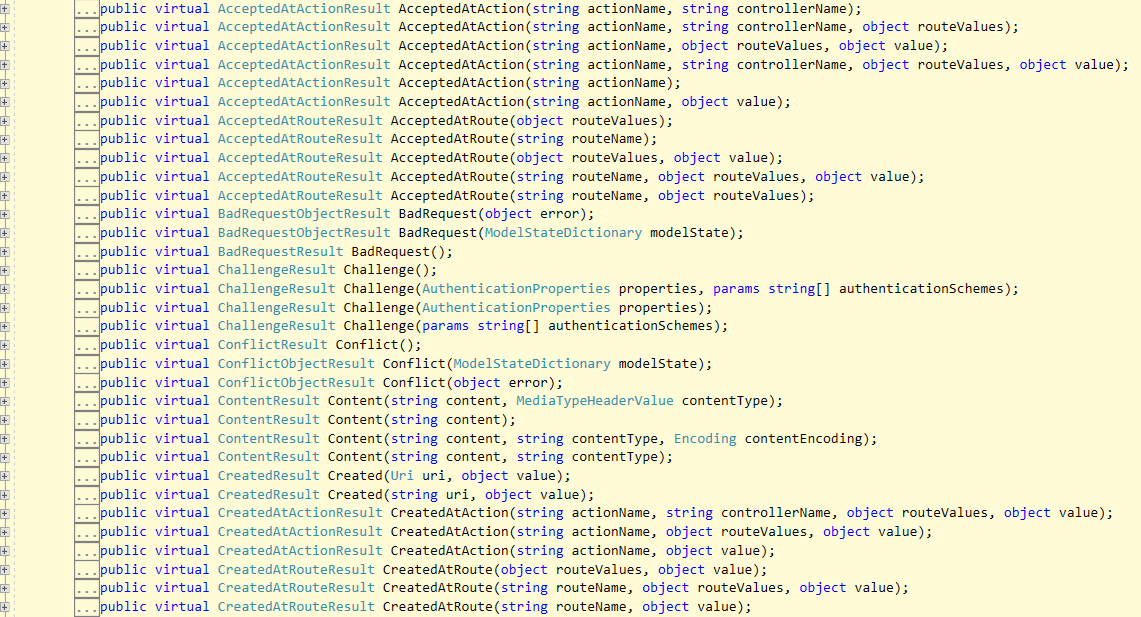
首先，可以看到Controller这个类继承了ControllerBase基类，并提供了对View的支持

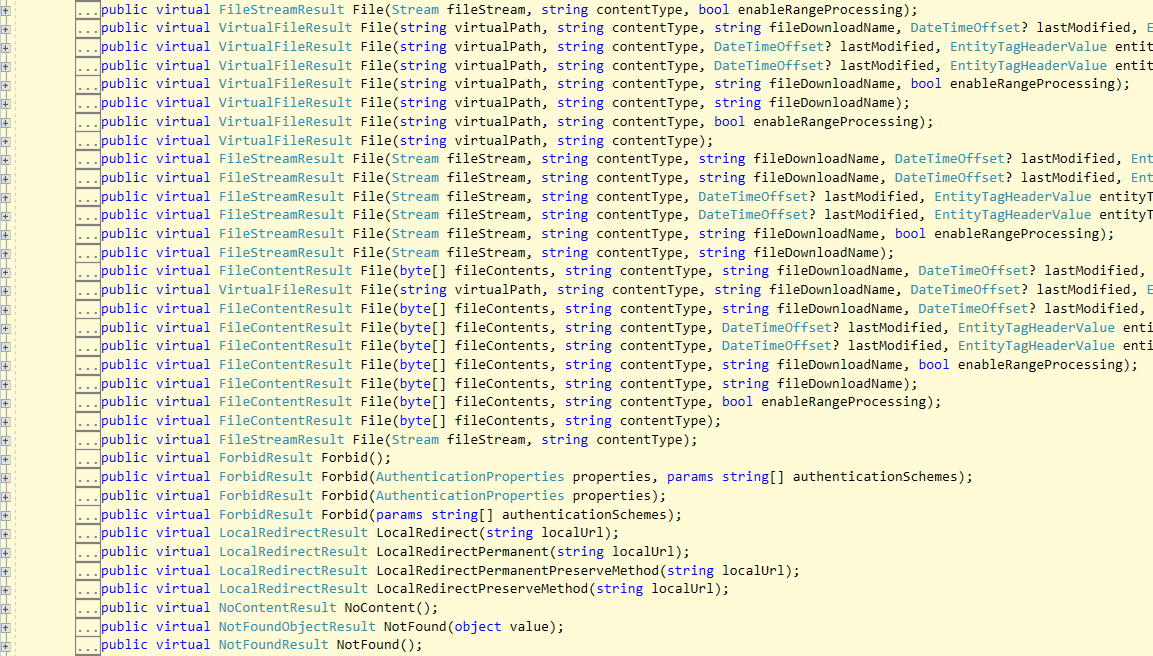
包含有：ViewData, JsonResule, Action Event, View, ViewComponentResult

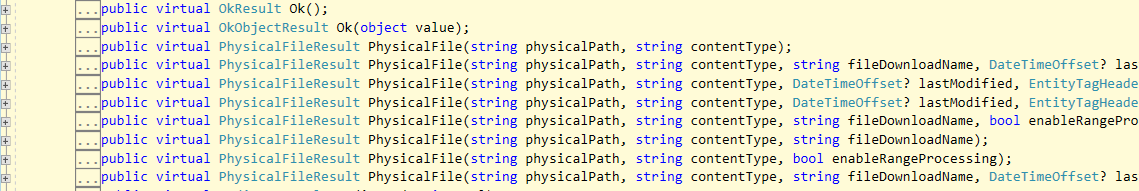


**ControllerBase :** 并不提供对 View 的支持， 只包含请求信息头信息等， 和返回的数据格式与结果













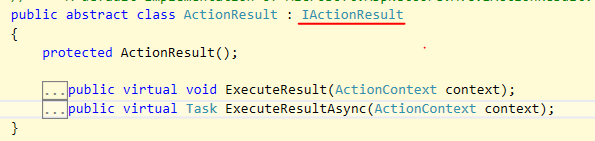
总结： 如果是 MVC 框架则使用 Controller, 如果是WebApi 用于数据提供则使用 ControllerBase

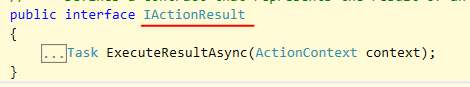
Controller继承自ControllerBase , 凡是ControllerBase有的特性, Controller 都有

**ControllerBase:**

ActionResult - 是其他返回类型的基础

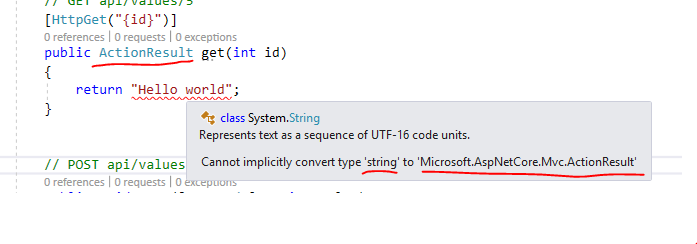
public abstract class ActionResult : IActionResult





ActionResult 作为 WebApi 返回类型：

1. 通常习惯使用 IActionResult 来替代 ActionResult 作为返回类型
2. ActionResult 不能直接返回类型, 需要调用其他的辅助方法来返回数据。



1. ActionResult 是其他返回类型的基础类型：并其它是抽象类。把它作为WebApi的返回类型，只能通过调用其他的辅助方法来返回不同类型 (这些类型都是直接或多层继承自ActionResult）

ObjectResult - StatusCode()

AcceptedResult - Accepted()

StatusCodeResult – StatusCode()

ContentResult – Content()

OkResult – Ok()

OkObjectResult – Ok()

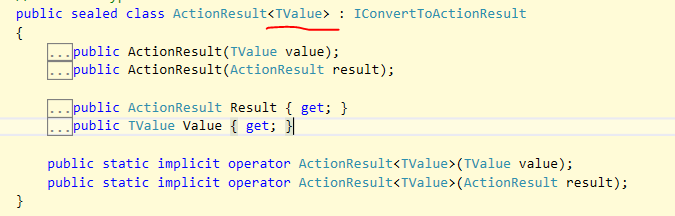
BadRequestResult – BadRequest()

BadRequestObjectResult - BadRequest()

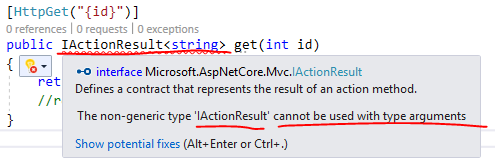
ForbidResult – Forbid()

SignInResult - SignIn()

1. 如果我们使用辅助方法来返回数据不方便，可以使用ActionResult<TValue> 这样可以直接返回所需要的类型。



注意： 泛类型没有 IActionResult<TValue>, 只有ActionResult<TValue>



使用泛类型的缺点， 不能很好的处理错误：

[HttpGet("{id}")]

public ActionResult<int> get(int id)

{

try

{

int a = 100;

int b = 0;

int c = a / b;

return c;

}

catch(Exception err)

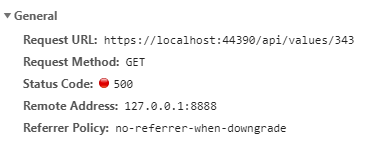
{

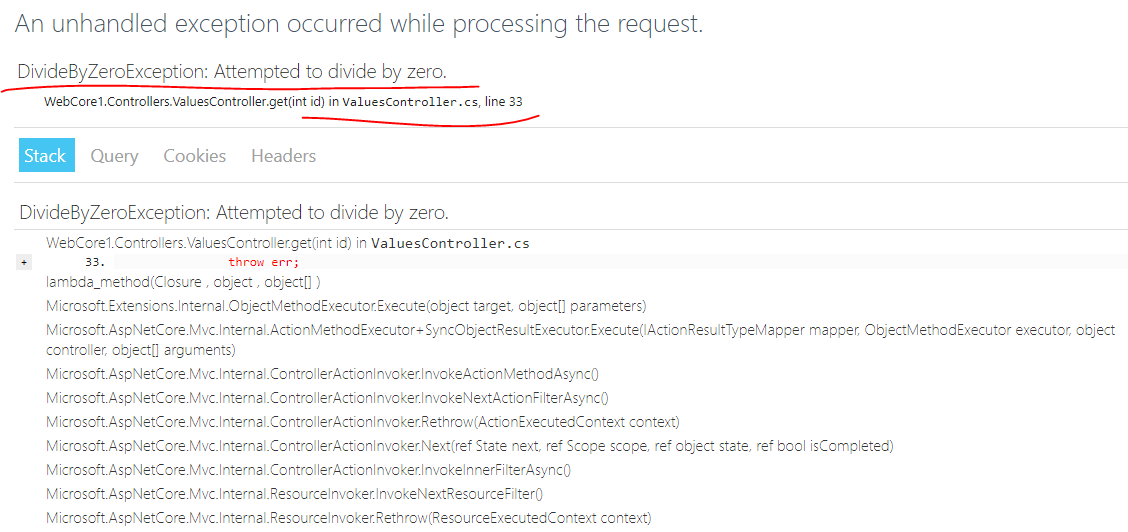
throw err;

}

}

StatusCode: 500 Internal Server Error





关于WebApi 的环境依赖注入问题：

WebApi Controller 的构造函数本身可以自动绑定

[Route("api/[controller]")]

[ApiController]

public class CoursesController : ControllerBase

{

public CourseDB DB { get; set; }

public CoursesController(

IHostingEnvironment env,

IOptions<**SQLConfig**> sqlConfig,

IConfiguration conf

)

{

this.Env = env;

this.DB = new CourseDB(sqlConfig.Value);

}

[HttpGet]

public IActionResult Get()

{

}

}

StartUp.cs

public class Startup

{

public Startup(IConfiguration configuration)

{

Configuration = configuration;

}

public IConfiguration Configuration { get; }

// This method gets called by the runtime. Use this method to add services to the container.

public void ConfigureServices(IServiceCollection services)

{

services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version\_2\_1);

services.Configure<**SQLConfig**>(Configuration.GetSection("SQLServer"));

services.AddSpaStaticFiles(configuration =>

{

configuration.RootPath = "ClientApp/dist";

});

services.AddDistributedMemoryCache();

services.AddSession(options =>

{

// Set a short timeout for easy testing.

options.IdleTimeout = TimeSpan.FromSeconds(3600 \* 24);

options.Cookie.HttpOnly = true;

});

}

public void Configure(IApplicationBuilder app, IHostingEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

else

{

app.UseExceptionHandler("/Error");

app.UseHsts();

}

app.UseHttpsRedirection();

app.UseStaticFiles();

app.UseSpaStaticFiles();

app.UseSession();

app.UseMvc(routes =>

{

routes.MapRoute(

name: "default",

template: "{controller=Home}/{action=Index}/{id?}");

});

app.UseSpa(spa =>

{

// To learn more about options for serving an Angular SPA from ASP.NET Core,

// see https://go.microsoft.com/fwlink/?linkid=864501

spa.Options.SourcePath = "ClientApp";

if (env.IsDevelopment())

{

spa.UseAngularCliServer(npmScript: "start");

}

});

}

}

# Use the Angular project template with ASP.NET Core

<https://docs.microsoft.com/en-us/aspnet/core/client-side/spa/angular?view=aspnetcore-2.1&tabs=visual-studio>

* 传统的ASP.NET MVC 的路由

protected void Application\_Start()

{

AreaRegistration.RegisterAllAreas();

FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);

RouteConfig.RegisterRoutes(RouteTable.Routes);

BundleConfig.RegisterBundles(BundleTable.Bundles);

}

public static void RegisterRoutes(RouteCollection routes)

{

routes.IgnoreRoute("{resource}.axd/{\*pathInfo}");

routes.MapRoute(

name: "Default",

url: "{controller}/{action}/{id}",

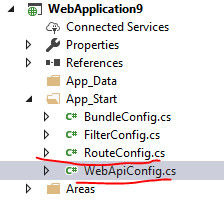
defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }

);

}

传统的MVC + WEBAPI路由机制：

激活 [RoutePrefix()] 和 [Route()] 必须添加MapHttpAttributeRoutes()



MVC 的路由配置

public class RouteConfig

{

public static void RegisterRoutes(RouteCollection routes)

{

routes.**MapMvcAttributeRoutes**(); - 这是激活 MVC 的[RoutePrefix()] 和 [Route()]

routes.IgnoreRoute("{resource}.axd/{\*pathInfo}");

routes.MapRoute(

name: "Default",

url: "{controller}/{action}/{id}",

defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }

);

}

}

有了上面的正确配置：

[RoutePrefix("sales/home")]

public class HomeController : Controller

{

[Route("haha")]

public ActionResult Index()

{

ViewBag.Title = "Home Page";

return View();

}

}

WebApi 的路由配置

public static class WebApiConfig

{

public static void Register(HttpConfiguration config)

{

config.MapHttpAttributeRoutes(); - 激活WebApi的 [RoutePrefix()] 和 [Route()]

config.Routes.MapHttpRoute(

name: "DefaultApi",

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

defaults: new { id = RouteParameter.Optional }

);

}

}

有了上面的正确配置：

[RoutePrefix("sales/order")]

public class ValuesController : ApiController

{

[Route("detail")]

// GET api/values

public IEnumerable<string> Get()

{

return new string[] { "value1", "value2" };

}

}

* ASP.Net Core 的路由机制:

默认的设置：

public void ConfigureServices(IServiceCollection services)

{

services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version\_2\_1);

}

如果选择的是 ASP.Net Core Web Application:

public void Configure(IApplicationBuilder app, IHostingEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

else

{

app.UseExceptionHandler("/Error");

app.UseHsts();

}

app.UseHttpsRedirection();

app.UseStaticFiles();

app.UseCookiePolicy();

app.**UseMvc**();

}

1. app.**UseMvc**() 并没有指定路由规则，为什么可以自动路由WebApi 的请求

app.UseMvc() – 并没有设置路由规则，只是启用路由功能，WebApi 之所以能被路由

是因为: 有指定[Route]

[**Route**("api/[controller]")]

[ApiController]

public class ValuesController : ControllerBase

{

public IHostingEnvironment Env;

public IConfiguration Config;

public ValuesController(IHostingEnvironment env, IConfiguration config)

{

this.Env = env;

this.Config = config;

}

// GET api/values

[HttpGet]

public ActionResult<IEnumerable<string>> Get()

{

int a = 100;

int b = 15;

int c = a / b;

return new string[] { $"a={a}", $"b={b}", $"a/b={c}" };

}

}

那么对于Asp.Net Core 里有 MVC 应用：

<https://localhost:44364/home/index>

public class HomeController : Controller

{

public IActionResult Index()

{

return View();

}

public IActionResult Error ()

{

return View();

}

public IActionResult Contact([FromRoute] string myurl)

{

var con = RouteData.Values["CONTROLLER"];

var act = RouteData.Values["ACTion"];

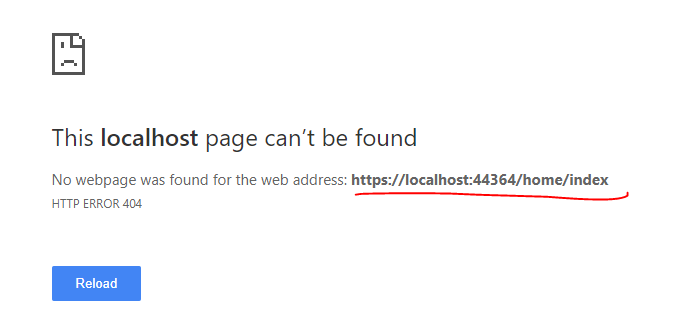
var myu = RouteData.Values["MYURL"];

ViewBag.myu = myurl;

return View();

}

}



[Route("HOME")]

public class HomeController : Controller

{

[Route("Haha")] - 这里有定义路由

public IActionResult Index()

{

return View();

}

public IActionResult Error ()

{

return View();

}

public IActionResult Contact([FromRoute] string myurl) - 这里没有定义路由

{

var con = RouteData.Values["CONTROLLER"];

var act = RouteData.Values["ACTion"];

var myu = RouteData.Values["MYURL"];

ViewBag.myu = myurl;

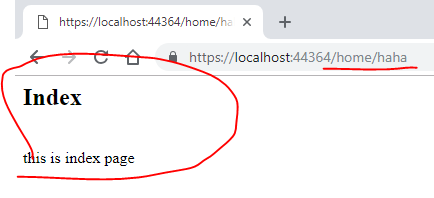
return View();

}

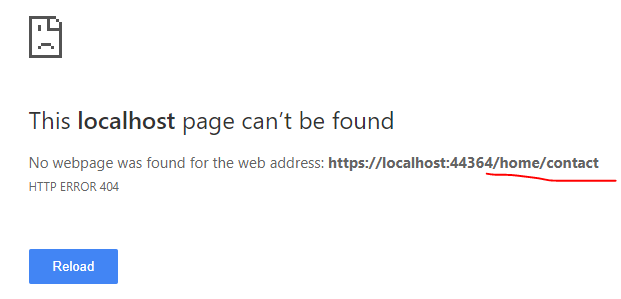
}

注意只有定义路由的Action 才能被执行

<https://localhost:44364/home/haha> - 这个可以路由到此页面

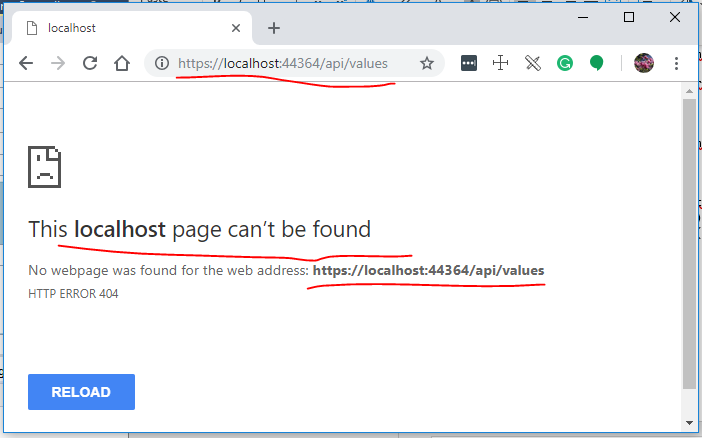


<https://localhost:44364/home/contact> - 由于Action 上没有定义路由，所以找不到



1. 如果在 Configure 里不指定： app.UseMvc() 则无法路由所有的请求

<https://localhost:44364/api/values>



1. 在 Configure 里指定： MVC 的路由

如果选择的是 ASP.Net Core Web Application (MVC)

public void Configure(IApplicationBuilder app, IHostingEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

else

{

app.UseExceptionHandler("/Home/Error");

app.UseHsts();

}

app.UseHttpsRedirection();

app.UseStaticFiles();

app.UseCookiePolicy();

app.UseMvc(routes =>

{

routes.MapRoute(

name: "default",

template: "{controller=Home}/{action=Index}/{id?}");

});

}

也可以使用默认 defaults: new { controller= “”, action=”” },

app.UseMvc(routes =>

{

routes.MapRoute(

name: "default",

template: "{controller}/{action}/{id?}",

defaults: new { controller = "home", action = "index" }

);

});

Controller, Action 大小写无所谓

app.UseMvc(routes =>

{

routes.MapRoute(

name: "default",

template: "{controller}/{action}/{id?}",

defaults: new { Controller="Home", Action="contact" }

);

});

也可以往路由里添加其他默认 key => value

app.UseMvc(routes =>

{

routes.MapRoute(

name: "default",

template: "{controller}/{action}/{myurl}",

defaults: new { Controller="Home", Action="contact", myurl="lwh.cshtml" }

);

});

我们可以从WebApi 的具体 Action 里取回路由值：RouteData.Values[ keyName ] keyName 大小写无所谓

public class HomeController : Controller

{

public IActionResult Index()

{

return View();

}

public IActionResult Error ()

{

return View();

}

public IActionResult Contact([FromRoute] string myurl)

{

var con = RouteData.Values["CONTROLLER"];

var act = RouteData.Values["ACTion"];

var myu = RouteData.Values["MYURL"];

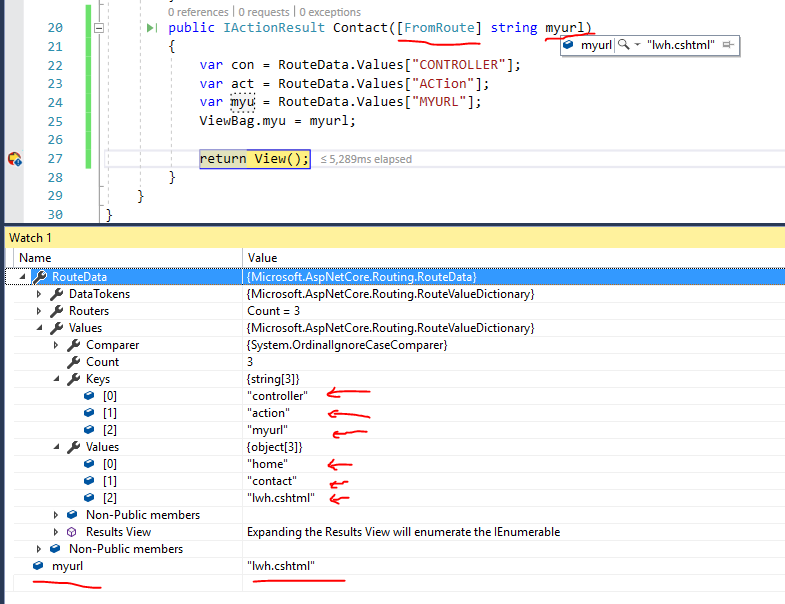
ViewBag.myu = myurl;

return View();

}

}

因为全局已经指定了路由， 所以 MVC 里的 controller 和 action 可以不需要指定路由。



1. 在 Controller 里指定： WebApi 的路由

[Route("api/[controller]")]

[ApiController]

public class ValuesController : ControllerBase

{

public IHostingEnvironment Env;

public IConfiguration Config;

public ValuesController(IHostingEnvironment env, IConfiguration config)

{

this.Env = env;

this.Config = config;

}

// GET api/values

[HttpGet("take")]

public ActionResult<IEnumerable<string>> Get()

{

int a = 100;

int b = 15;

int c = a / b;

return new string[] { $"a={a}", $"b={b}", $"a/b={c}" };

}

// GET api/values/5

[HttpGet("{id}")]

public ActionResult<string> get(int id)

{

return $"Get id = {id}";

}

// POST api/values

[HttpPost]

public IActionResult Post([FromBody] StuModel stu)

{

return Ok(stu);

}

// PUT api/values/5

[HttpPut("{id}")]

public string Put(int id)

{

int a = 100;

int b = 20;

int c = a / b;

return $"Result: {a / b} id={id}";

}

// DELETE api/values/5

[Route("delete/{id}")]

public async Task<string> Delete(int id)

{

return await Task.FromResult<string>($"Delete id={id}");

}

}

注意事项：

1. WebApi 只涉及数据的返回，不需要考虑 View, 所以使用 ControllerBase 即可
2. 必须要使用 [ApiController], 以更自动映射Method => Action (Get, Post, Put, Delete), 否则不会映射

[Route("api/[controller]")] - 必须指定以区分其他的路由

[ApiController] – 必须指定否则不会映射到对应的Action

1. 关于 [HttpPost], [HttpGet], [HttpPut], [HttpDelete], [Route] 都可以重定义路由的 URL

[HttpPost], [HttpGet], [HttpPut], [HttpDelete], [Route] 都继承自 IRouteTemplateProvider

所以都可以改写路由URL

1. 在 .Net Core WebApi 没有 [RoutePrefix()], 只能在全局里定义

public static class MvcOptionsExtensions

{

public static void UseCentralRoutePrefix(this MvcOptions opts, IRouteTemplateProvider routeAttribute)

{

// 添加我们自定义 实现IApplicationModelConvention的RouteConvention

opts.Conventions.Insert(0, new RouteConvention(routeAttribute));

}

}

public class RouteConvention : IApplicationModelConvention

{

private readonly AttributeRouteModel \_centralPrefix;

public RouteConvention(IRouteTemplateProvider routeTemplateProvider)

{

\_centralPrefix = new AttributeRouteModel(routeTemplateProvider);

}

//接口的Apply方法

public void Apply(ApplicationModel application)

{

//遍历所有的 Controller

foreach (var controller in application.Controllers)

{

// 已经标记了 RouteAttribute 的 Controller

var matchedSelectors = controller.Selectors.Where(x => x.AttributeRouteModel != null).ToList();

if (matchedSelectors.Any())

{

foreach (var selectorModel in matchedSelectors)

{

// 在 当前路由上 再 添加一个 路由前缀

selectorModel.AttributeRouteModel = AttributeRouteModel.CombineAttributeRouteModel(\_centralPrefix,

selectorModel.AttributeRouteModel);

}

}

// 没有标记 RouteAttribute 的 Controller

var unmatchedSelectors = controller.Selectors.Where(x => x.AttributeRouteModel == null).ToList();

if (unmatchedSelectors.Any())

{

foreach (var selectorModel in unmatchedSelectors)

{

// 添加一个 路由前缀

selectorModel.AttributeRouteModel = \_centralPrefix;

}

}

}

}

}

public void ConfigureServices(IServiceCollection services)

{

//services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version\_2\_1);

services.AddMvc(opt=> {

opt.UseCentralRoutePrefix(new RouteAttribute("myapp/v2"));

}).SetCompatibilityVersion(CompatibilityVersion.Version\_2\_1);

}

<https://www.cnblogs.com/savorboard/p/dontnet-IApplicationModelConvention.html>

注意： 只对 MVC 起作用， WebApi 则工作不正常， 不是很好的解决方案，用处不大

1. 扩展默认路由

app.UseMvc(routes =>

{

routes.MapRoute(

name: "Contact",

template: "about",

defaults: new { Controller="home", Action="about", yeep="Sohu"}

);

routes.MapRoute(

name: "default",

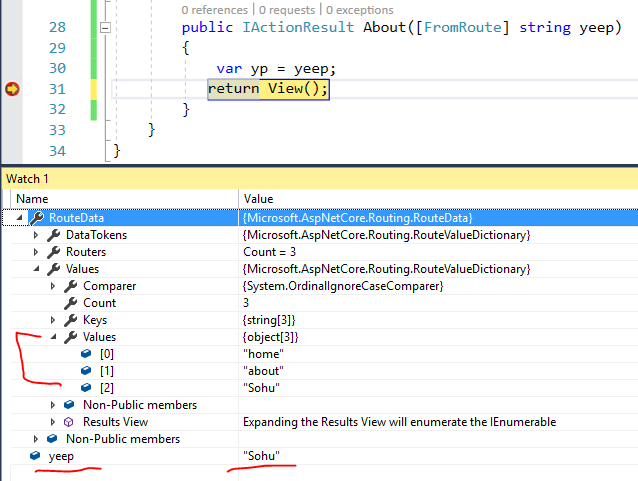
template: "{controller}/{action}/{myurl}",

defaults: new { Controller="Home", Action="contact", myurl="lwh.cshtml" }

);

});

https://localhost:44364/about



如果在Controller 里有定义路由，则覆盖全局的路由定义

[Route("[controller]/[action]/{myurl}")]

public class HomeController : Controller

{

public IActionResult Index()

{

return View();

}

public IActionResult Error ()

{

return View();

}

public IActionResult Contact([FromRoute] string myurl)

{

var con = RouteData.Values["CONTROLLER"];

var act = RouteData.Values["ACTion"];

var myu = RouteData.Values["MYURL"];

ViewBag.myu = myurl;

return View();

}

public IActionResult About([FromRoute] string yeep)

{

var yp = yeep;

return View();

}

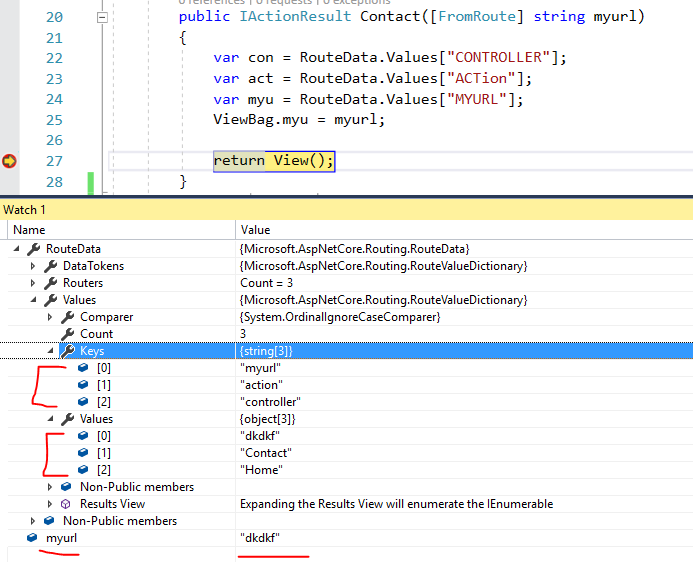
}

<https://localhost:44364/about> - 找不到

<https://localhost:44364/home/index> - 找不到

<https://localhost:44364/home/index/dkdkf> - 找到匹配，多出一个路由值 myurl

<https://localhost:44364/home/contact/dkdkf> - 找到匹配



1. [Route("[controller]")] 和 [Route("[action]")] 都是引用的声明的控制器名称和方法名称，即Home控制器和Index方法.可选参数 {xxx?} 加问号

[Route("[action]/{yeep=Good}")] 也可以带默认值，如果有默认值，不带参数也可以被匹配到，参数使用默认值。

其他可选路由参数：

[Route("[action]/{myu**?**}")]

public IActionResult Index([FromRoute] string myu)

{

var url = myu;

return View();

}

<https://localhost:44364/home/index> - 路由匹配， myu = null

<https://localhost:44364/home/index/jackwu> - 路由匹配， myu = jackwu

[Route("[action]/{myu?}")]

public IActionResult Index([FromRoute] int myu)

{

var url = myu;

return View();

}

<https://localhost:44364/home/index> - 路由匹配， myu = 0

<https://localhost:44364/home/index/jackwu> - 路由匹配， myu = 0

<https://localhost:44364/home/index/16.89883> - 路由匹配， myu = 0

<https://localhost:44364/home/index/899> - 路由匹配， myu = 899

[Route("[action]/{myu}")]

public IActionResult Index([FromRoute] DateTime myu)

{

var url = myu;

return View();

}

<https://localhost:44364/home/index> - 路由不匹配，路由参数必须

<https://localhost:44364/home/index/993> - 路由匹配，myu=日期最小值

1. Route Attributes

public class OrdersController : ApiController

{

[Route("customers/{customerId}/orders")]

[HttpGet]

public IEnumerable<Order> FindOrdersByCustomer(int customerId) { ... }

}

[Route("customers/{customerId}/orders/{orderId}")]

public Order GetOrderByCustomer(int customerId, int orderId) { ... }

1. including non-standard methods, use the **AcceptVerbs** attribute

// WebDAV method

[Route("api/books")]

[AcceptVerbs("MKCOL")]

public void MakeCollection() { }

1. Route Constraints

[Route("users/{id:int}")]

public User GetUserById(int id) { ... }

[Route("users/{name}")]

public User GetUserByName(string name) { ... }

[Route("users/{id:int:min(1)}")]

public User GetUserById(int id) { ... }

The following table lists the constraints that are supported.

| **Constraint** | **Description** | **Example** |
| --- | --- | --- |
| alpha | Matches uppercase or lowercase Latin alphabet characters (a-z, A-Z) | {x:alpha} |
| bool | Matches a Boolean value. | {x:bool} |
| datetime | Matches a **DateTime** value. | {x:datetime} |
| decimal | Matches a decimal value. | {x:decimal} |
| double | Matches a 64-bit floating-point value. | {x:double} |
| float | Matches a 32-bit floating-point value. | {x:float} |
| guid | Matches a GUID value. | {x:guid} |
| int | Matches a 32-bit integer value. | {x:int} |
| length | Matches a string with the specified length or within a specified range of lengths. | {x:length(6)} {x:length(1,20)} |
| long | Matches a 64-bit integer value. | {x:long} |
| max | Matches an integer with a maximum value. | {x:max(10)} |
| maxlength | Matches a string with a maximum length. | {x:maxlength(10)} |
| min | Matches an integer with a minimum value. | {x:min(10)} |
| minlength | Matches a string with a minimum length. | {x:minlength(10)} |
| range | Matches an integer within a range of values. | {x:range(10,50)} |
| regex | Matches a regular expression. | {x:regex(^\d{3}-\d{3}-\d{4}$)} |

1. Optional URI Parameters and Default Values

public class BooksController : ApiController

{

[Route("api/books/locale/{lcid:int?}")]

public IEnumerable<Book> GetBooksByLocale(int lcid = 1033) { ... }

}

1. Route Names

public class BooksController : ApiController

{

[Route("api/books/{id}", Name="GetBookById")]

public BookDto GetBook(int id)

{

// Implementation not shown...

}

[Route("api/books")]

public HttpResponseMessage Post(Book book)

{

// Validate and add book to database (not shown)

var response = Request.CreateResponse(HttpStatusCode.Created);

// Generate a link to the new book and set the Location header in the response.

string uri = Url.Link("GetBookById", new { id = book.BookId });

response.Headers.Location = new Uri(uri);

return response;

}

}

1. Route Order

[RoutePrefix("orders")]

public class OrdersController : ApiController

{

[Route("{id:int}")] // constrained parameter

public HttpResponseMessage Get(int id) { ... }

[Route("details")] // literal

public HttpResponseMessage GetDetails() { ... }

[Route("pending", RouteOrder = 1)]

public HttpResponseMessage GetPending() { ... }

[Route("{customerName}")] // unconstrained parameter

public HttpResponseMessage GetByCustomer(string customerName) { ... }

[Route("{\*date:datetime}")] // wildcard

public HttpResponseMessage Get(DateTime date) { ... }

}

These routes are ordered as follows.

1. orders/details
2. orders/{id}
3. orders/{customerName}
4. orders/{\*date}
5. orders/pending

* ASP.Net Core: json 配置文件的注入问题：

1. 注入整个IConfiguration

public Startup(IConfiguration configuration)

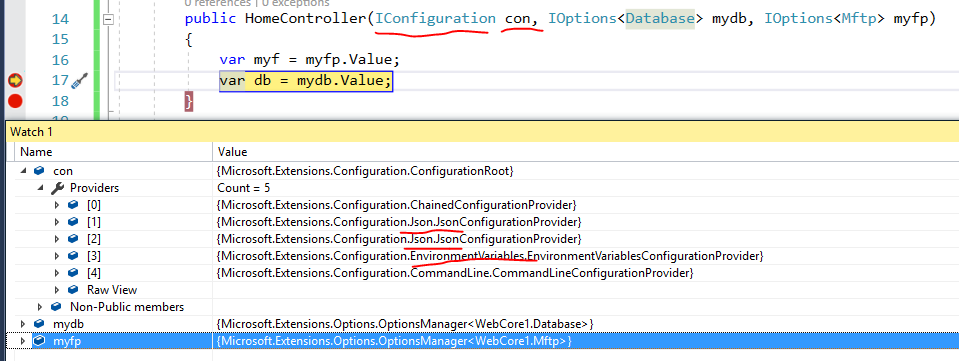
{

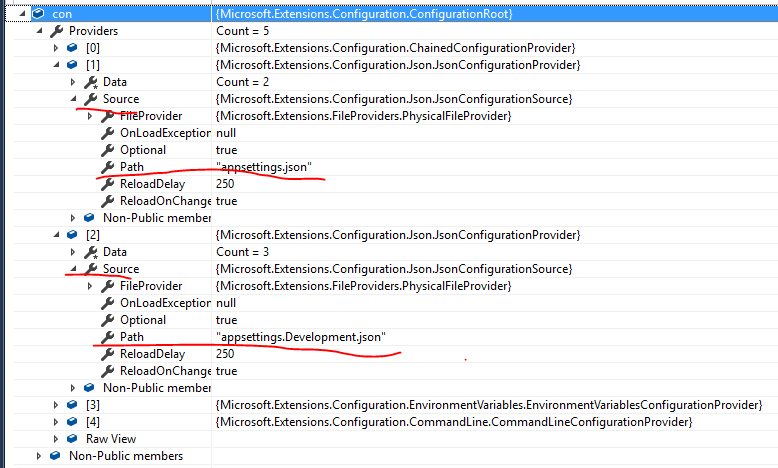
Configuration = configuration;

}

默认的IConfiguration 注入了5 个 项目

其中两个 JSON 文件： appsettings.json 和 appsettings.Development.json





我们可以自定义IConfiguration 并注入到所有Controller

IConfigurationBuilder builder = new ConfigurationBuilder();

builder.SetBasePath(AppDomain.CurrentDomain.BaseDirectory)

.AddXmlFile(path: "lwh.xml", optional: false, reloadOnChange: true)

.AddJsonFile(path: "lwh.json", optional: false, reloadOnChange: true)

.AddJsonFile(path: "lwh1.json", optional: false, reloadOnChange: true)

.AddEnvironmentVariables();

IConfigurationRoot mycon = builder.Build();

services.AddSingleton<IConfiguration>(mycon);

lwh.xml

<?xml version="1.0" encoding="utf-8" ?>

<servers>

<database>

<host>Localhost:333</host>

<user>SA</user>

<trust>true</trust>

<yeep>XML Yeep</yeep>

</database>

</servers>

Lwh.json

{

"database": {

"host": "localhost",

"user": "dba",

"pwd": "889"

},

"website": "www.d3security.com",

"ftp": {

"server": "ftp://qa.d3security.com",

"uid": "567",

"password": "d3456"

}

}

Lwh1.json

{

"database": {

"user": "superman",

"flag": "good"

},

"website": "www.d3security.com",

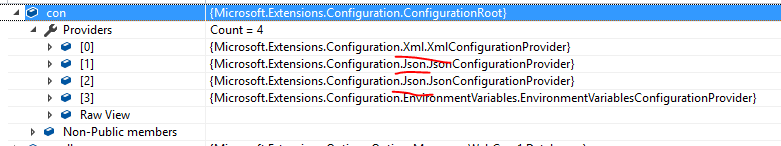
"ftp": {

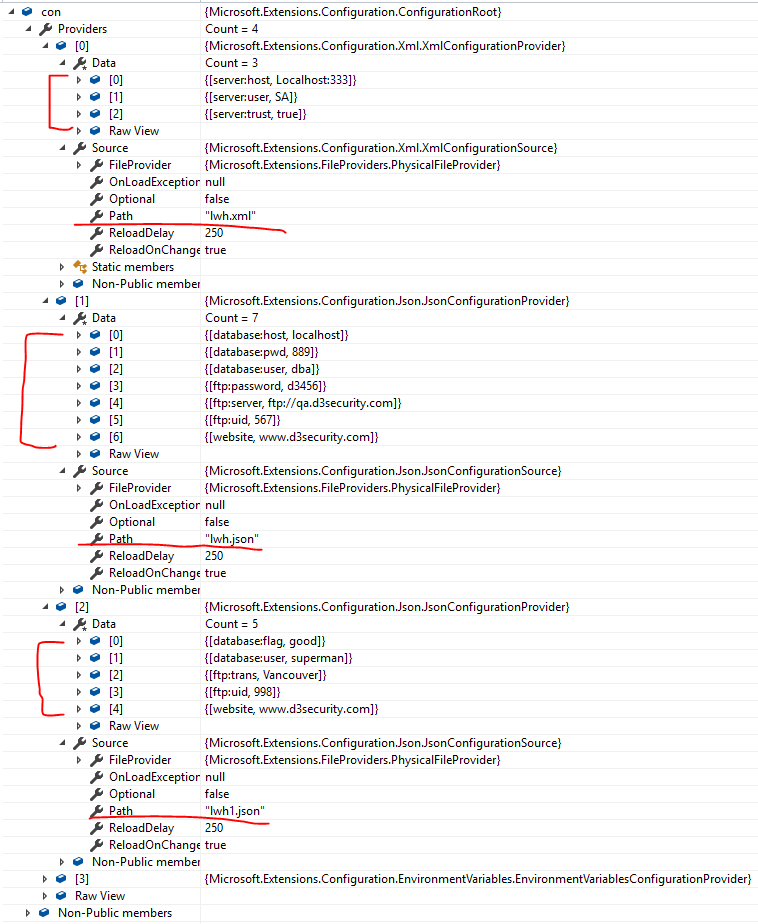
"uid": "998",

"trans": "Vancouver"

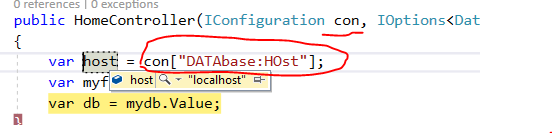
}

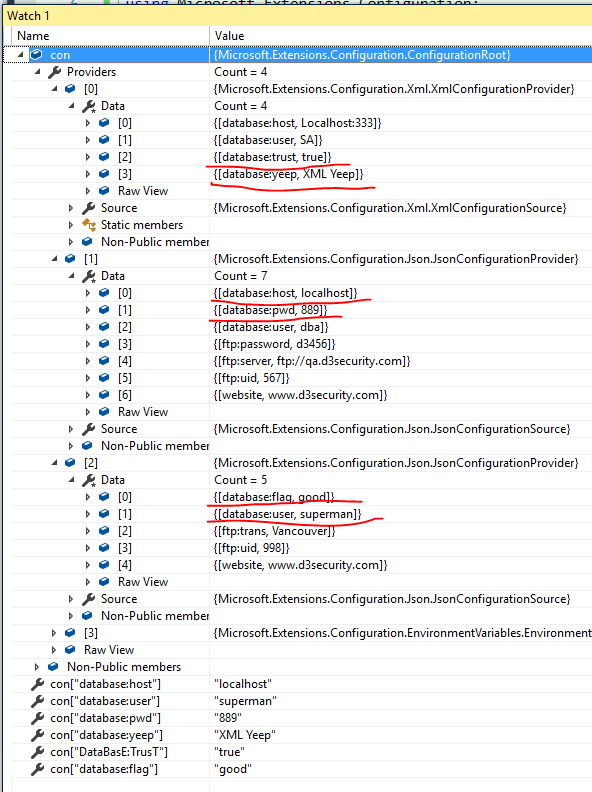
}





如何读取IConfiguration的值呢：

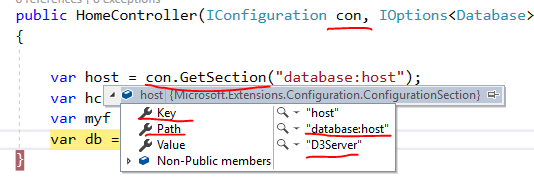




总结：

1. 我们可以看到所有的Config的键值都放在一个字典里，关键字大小写无所谓： host, Host, HOst
2. 无论我们添加多少个文件，最终都会自动合并，相同的键值后面的覆盖前面的。
3. 所有值都是字符串，需要自行转化成自己想要的值。
4. GetSession 方法是返回一个字典项目，和获取字典方法是一样 con[“key”]

con.GetSection("database:host")



我们把加载自定义文件的顺序改变一下：

IConfigurationBuilder builder = new ConfigurationBuilder();

builder.SetBasePath(AppDomain.CurrentDomain.BaseDirectory)

.AddJsonFile(path: "lwh.json", optional: false, reloadOnChange: true)

.AddJsonFile(path: "lwh1.json", optional: false, reloadOnChange: true)

//这次把 XML 内容放到最后

.AddXmlFile(path: "lwh.xml", optional: false, reloadOnChange: true)

.AddEnvironmentVariables();

IConfigurationRoot mycon = builder.Build();

services.AddSingleton<IConfiguration>(mycon);

lwh.xml内容：

<?xml version="1.0" encoding="utf-8" ?>

<servers>

<database>

<host>D3Server</host>

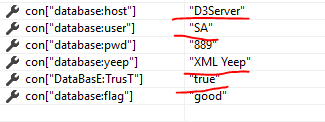
<user>SA</user>

<trust>true</trust>

<yeep>XML Yeep</yeep>

</database>

</servers>



我们看到， 合并：相同的键值后面的覆盖前面的

1. 在程序启动阶段，添加自定义配置文件：

public static void Main(string[] args)

{

CreateWebHostBuilder(args).Build().Run();

}

public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>

WebHost.CreateDefaultBuilder(args)

.ConfigureAppConfiguration((hostingContext, config) =>

{

config.SetBasePath(Directory.GetCurrentDirectory())

.AddJsonFile(path: "lwh.json", optional: false, reloadOnChange: true)

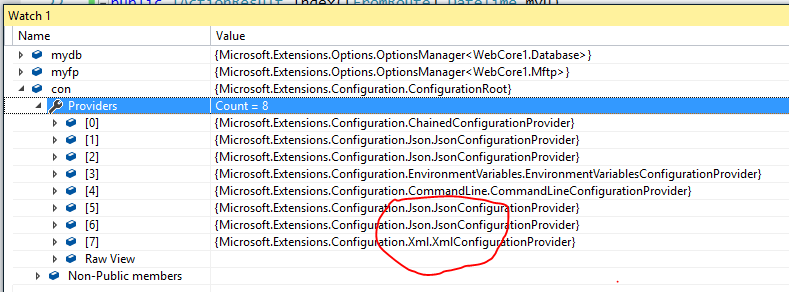
.AddJsonFile(path: "lwh1.json", optional: false, reloadOnChange: true)

.AddXmlFile(path: "lwh.xml", optional: false, reloadOnChange: true);

})

.UseStartup<Startup>();

并不会覆盖默认的，而是添加到默认IConfiguration后面, 如下图：



1. 注入IOptions<OptionSample类>

先定义一个OptionSample类需要实现IOptions接口

public class Database

{

public string host { get; set; }

public string user { get; set; }

public int pwd { get; set; }

public string flag { get; set; }

public string yeep { get; set; }

}

public class Mftp

{

public string server { get; set; }

public int uid { get; set; }

public string password { get; set; }

public string trans { get; set; }

}

public void ConfigureServices(IServiceCollection services)

{

services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version\_2\_1);

IConfigurationBuilder builder = new ConfigurationBuilder();

builder.SetBasePath(AppDomain.CurrentDomain.BaseDirectory)

.AddJsonFile(path: "lwh.json", optional: false, reloadOnChange: true)

.AddJsonFile(path: "lwh1.json", optional: false, reloadOnChange: true)

.AddXmlFile(path: "lwh.xml", optional: false, reloadOnChange: true)

.AddEnvironmentVariables();

IConfigurationRoot mycon = builder.Build();

services.Configure<Database>(mycon.GetSection("database"));

services.Configure<Mftp>(mycon.GetSection("ftp"));

}

# 这样做的好处，不会把默认的 IConfiguration 覆盖掉，而且是以强命名的对象读取数据

# 

public static IWebHost BuildWebHost(string[] args) =>

WebHost.CreateDefaultBuilder(args)

.ConfigureAppConfiguration((context, builder) =>

{

IHostingEnvironment env = context.HostingEnvironment;

builder.AddJsonFile("appsettings.json", optional:false, reloadOnChange:true)

.AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional:true, reloadOnChange:true);

})

.UseStartup<Startup>()

.Build();

# 加环境变量来加载配置文件

# ASP.NET Core 依赖注入：

using Microsoft.Extensions.DependencyInjection;

## 直接注入IServiceProvider的方式

因为在Core里面，IServiceProvider是允许直接注入到具体应用里的，所以就有了直接通过IServiceProvider来获取具体实现的方式。

1. 通过GetService方式

首先注入具体实现

services.AddTransient<LogicSericeImpt1>();

services.AddTransient<LogicSericeImpt2>();

然后在构造函数中通过如下方式获取具体实现

public TodoController(IServiceProvider serviceProvider)

{

var s1FromProvider = serviceProvider.GetService<LogicSericeImpt1>();

var s2FromProvider = serviceProvider.GetService<LogicSericeImpt2>();

}

2. 通过GetServices方式

首先注入interface及具体实现

services.AddTransient<ILogicService, LogicSericeImpt1>();

services.AddTransient<ILogicService, LogicSericeImpt2>();

然后在构造函数中通过如下方式获取具体实现

public TodoController(IServiceProvider serviceProvider)

{

var logicAllRegs = serviceProvider.GetServices<ILogicService>().ToList();

var s1FromAllRegs = logicAllRegs[0];

var s2FromAllRegs = logicAllRegs[1];

}

3.直接注入集合的方式

这种方式其实就是省去了注入IServiceProvider的过程，直接将GetServices获取的结果进行注入。首先注入interface及具体实现

services.AddTransient<ILogicService, LogicSericeImpt1>();

services.AddTransient<ILogicService, LogicSericeImpt2>();

然后在构造函数中通过如下方式获取具体实现

public TodoController(IEnumerable<ILogicService> services)

{

var s1FromServices = services.First();

var s2FromServices = services.Skip(1).First();

}

4.直接注入Func工厂的方式

在这种方式下，我们并不是直接注入具体实现，而是注入了Func这么一个工厂实现。首先我们还是需要注入具体实现

services.AddTransient<LogicSericeImpt1>();

services.AddTransient<LogicSericeImpt2>();

然后我们继续注入Func这个工厂，这里我们按int来返回不同的实现，当然你也可以采用其他方式比如string

services.AddSingleton(provider =>

{

Func<int, ILogicService> func = n =>

{

switch (n)

{

case 1:

return provider.GetService<LogicSericeImpt1>();

case 2:

return provider.GetService<LogicSericeImpt2>();

default:

throw new NotSupportedException();

}

};

return func;

});

然后在构造函数中通过如下方式获取具体实现

public TodoController(Func<int, ILogicService> funcFactory)

{

var s1FromFunc = funcFactory(1);

var s2FromFunc = funcFactory(2);

}

public void ConfigureServices(IServiceCollection services)

{

services.AddDbContext<TeacherDB>(options=> {

options.UseSqlServer(Configuration.GetSection("Database:MSSQL:ConnectionString").Value);

} ,ServiceLifetime.Scoped);

services.AddDbContext<StudentDB>(options => {

options.UseSqlServer(Configuration.GetSection("Database:MSSQL:ConnectionString").Value);

}, ServiceLifetime.Scoped);

services.AddScoped<IDatabaseAction, StudentRepository>();

services.AddScoped<IDatabaseAction, TeacherRepository>();

services.AddCors(option => {

option.AddPolicy("enbabled", builder =>

builder.AllowAnyOrigin().AllowAnyHeader().AllowAnyMethod().AllowCredentials().Build()

);

});

services.Configure<CookiePolicyOptions>(options =>

{

options.CheckConsentNeeded = context => true;

options.MinimumSameSitePolicy = SameSiteMode.None;

});

services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version\_2\_1);

}

public TeacherRepository TDB { get; set; }

public StudentRepository SDB { get; set; }

public TeacherController(**IServiceProvider** serviceProvider) {

var services = serviceProvider.GetServices<IDatabaseAction>().ToList();

this.SDB = services[0] as StudentRepository;

this.TDB = services[1] as TeacherRepository;

}

## 服务生命周期

 下面是服务在ASP.NET Core依赖注入中的生命周期：

1. **Transient**类型的服务在每次注入或请求的时候被创建。
2. **Scoped**类型的服务按照作用域被创建。在Web程序中，每个Web请求都会创建新的隔离的服务作用域。这意味着**Scoped**类型的服务通常会根据Web请求创建。
3. **Singleton**类型的服务由DI容器创建。这通常意味着它们根据应用程序仅仅被创建一次，然后用于应用程序的整个生命周期。

DI容器会持续跟踪所有已经被解析的服务。当服务的生命周期终止时，它们会被释放并销毁：

* 如果服务还有依赖，它们同样会被自动释放并销毁。
* 如果服务实现了 **IDisposable** 接口，Dispose 方法会在服务释放时自动被调用。

## 实践指南：

* 尽可能地将你的服务注册为 **Transient** 类型。因为设计Transient服务是简单的。你通常不用关心**多线程问题**和**内存泄漏问题**，并且你知道这类服务只有很短的生存期。
* **谨慎使用 Scoped** 类型服务生命周期，因为如果你创建了子服务作用域或者由非Web程序使用这些服务，那么它会变得诡异复杂。
* **谨慎使用Singleton** 类型的生命周期，因为你需要**处理多线程问题**和潜在的**内存泄漏问题**。
* **不要在Singleton服务上依赖** Transient类型或者 Scoped类型的服务。因为当单例服务注入的时候，Transient服务也会变成单例实例。并且如果Transient服务不是设计用于支持这样的场景的话则可能会导致一些问题。ASP.NET Core的默认DI容器在这种情况下会**抛出异常**。

# Repository 设计模型：

public interface IRepo<T> where T : class

{

IEnumerable<T> GetAll();

T GetByKey(object key);

T Add(T t);

T Update(T t);

bool Save();

}

public class Repo<T> : IRepo<T> where T : class

{

private DbContext DB;

public Repo(DbContext db) => DB = db;

public T Add(T t)

{

this.DB.Set<T>().Add(t);

return t;

}

public IEnumerable<T> GetAll()

{

return this.DB.Set<T>().ToList();

}

public T GetByKey(object key)

{

return this.DB.Find<T>();

}

public bool Save()

{

try

{

this.DB.SaveChanges();

return true;

}

catch (Exception err)

{

return false;

}

}

public T Update(T t)

{

this.DB.Set<T>().Update(t);

return t;

}

}

// Student Repository

public interface IStudentRepo: IRepo<Student>

{

}

public class StudentRepository: Repo<Student>, IStudentRepo

{

private StudentDB DB { get; set; }

public StudentRepository(StudentDB db):base(db) => DB = db;

}

//Teacher Repository

public interface ITeacherRepo: IRepo<Teacher>

{

}

public class TeacherRepository : Repo<Teacher>, ITeacherRepo

{

private TeacherDB DB { get; set; }

public TeacherRepository(TeacherDB db):base(db) => DB = db;

}

注意：

Repository 设计模型的缺点, 如果**DbContext**包含有多个不同类型的 **TEntity** **DbSet**<**TEntity**>,

则只能指定一个 TEntity 作为主要操作对象。

原因是：

public class Repo<T> : IRepo<T> where T : class {

public T Add(T t)

{

this.DB.Set<T>().Add(t);

return t;

}

}

this.DB.Set<T>() – 类型 T 必须是 类的对象必须明确的类型

public T Add<T> (T t) - 如果是 Add<T> - this.DB.Set<T>() 则会编译出错。

{

this.DB.Set<T>().Add(t);

return t;

}

public void ConfigureServices(IServiceCollection services)

{

services.AddDbContext<TeacherDB>(options=> {

options.UseSqlServer(Configuration.GetSection("Database:MSSQL:ConnectionString").Value);

} ,ServiceLifetime.Scoped);

services.AddDbContext<StudentDB>(options => {

options.UseSqlServer(Configuration.GetSection("Database:MSSQL:ConnectionString").Value);

}, ServiceLifetime.Scoped);

services.AddScoped<**ITeacherRepo**, **TeacherRepository**>();

services.AddScoped<**IStudentRepo**, **StudentRepository**>();

services.AddCors(option => {

option.AddPolicy("enbabled", builder =>

builder.AllowAnyOrigin().AllowAnyHeader().AllowAnyMethod().AllowCredentials().Build()

);

});

services.Configure<CookiePolicyOptions>(options =>

{

options.CheckConsentNeeded = context => true;

options.MinimumSameSitePolicy = SameSiteMode.None;

});

services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version\_2\_1);

}

[Route("api/[controller]")]

[ApiController]

public class TRepoController : ControllerBase

{

public **ITeacherRepo** DB { get; set; }

public TRepoController(ITeacherRepo db) => DB = db;

[HttpGet]

public IActionResult Get()

{

return Ok(this.DB.GetAll());

}

[HttpPost]

public IActionResult Post(Teacher t)

{

this.DB.Add(t);

this.DB.Save();

return Ok(t);

}

[HttpPut]

public IActionResult Update(Teacher t)

{

this.DB.Update(t);

this.DB.Save();

return Ok(t);

}

}

[Route("api/[controller]")]

[ApiController]

public class SRepoController : ControllerBase

{

public **IStudentRepo** DB { get; set; }

public SRepoController(**IStudentRepo** db) => DB = db;

[HttpGet]

public IActionResult Get()

{

return Ok(this.DB.GetAll());

}

[HttpPost]

public IActionResult Post(Student t)

{

this.DB.Add(t);

this.DB.Save();

return Ok(t);

}

[HttpPut]

public IActionResult Update(Student t)

{

this.DB.Update(t);

this.DB.Save();

return Ok(t);

}

}

* 常见问题：

# [.net core UseHttpsRedirection() 正式环境无效](https://www.cnblogs.com/lfzm/p/9723994.html) ：无法重定向

public void ConfigureServices(IServiceCollection services)  
{  
 services.AddHttpsRedirection(opt => opt.HttpsPort = 443);  
}

[Route("[action]/{myu}")]

public IActionResult Index([FromRoute] DateTime myu)

{

var url = myu;

if(myu == DateTime.MinValue)

{

return RedirectToAction("About");

}

return View();

}

[Route("[action]/{yeep=Good}")]

public IActionResult About([FromRoute] string yeep)

{

var yp = yeep;

return View();

}

## [Response.Redirect()和Response.RedirectPermanent()区别](https://www.cnblogs.com/lyl6796910/p/3793588.html)

Response.Redirect()方法发的是个 HTTP 302 Found (临时转向) 回复，会在用户尝试访问

老的URL时，导致多余的HTTP往返。搜索引擎一般不会跟随多个重新转向跳转，意味着使用一个临时转向会负面影响你的网页排名。

随着ASP.NET的发展，在ASP.NET 4中引进了一个新的Response.RedirectPermanent(string url)辅助方法，可以用来做一个HTTP 301 (永久性重定向)重新定向。

通常用Response.RedirectPermanent() 方法来进行SEO网站优化。