**[Asp.Net Core实战(干货)](https://www.cnblogs.com/sportsky/p/9400419.html)**

**序言**

使用.NET Core，团队可以更容易专注的在.net core上工作。比如核心类库（如System.Collections）的更改仍然需要与.NET Framework相同的活力，但是ASP.NET Core或Entity Framework Core可以更轻松地进行实质性更改，而不受向后兼容性的限制。.NET Core借鉴了.NET Framework的最佳实践，并将软件工程的最新进展结合在一起。

寒暄、扯淡已经完毕，，，下面是我最近时间对.Net Core整理的相关知识，觉得这些在项目中是最基础且最会应用到的，，，，不喜欢扯理论，直接撸码：

1、浅谈Startup类

2、自定义路由

3、跨域设置

4、自定义读取配置文件信息

5、程序集批量依赖注入

6、使用NLog写入文件日志

7、使用NLog写入数据库日志

8、Nlog标签解读

9、启用Session

10、json数据，自定义日期格式

11、json数据，string类型字段返回为null时默认返回空字符串

12、Json数据，返回字段同实体字段大小写一致

**一、浅谈Startup类**

在ASP.NET Core应用程序中，使用一个按约定Startup命名的类Startup，在Program.cs中使用[WebHostBuilderExtensions](https://docs.microsoft.com/en-us/dotnet/api/Microsoft.AspNetCore.Hosting.WebHostBuilderExtensions)[UseStartup <TStartup>](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.hosting.webhostbuilderextensions.usestartup#Microsoft_AspNetCore_Hosting_WebHostBuilderExtensions_UseStartup__1_Microsoft_AspNetCore_Hosting_IWebHostBuilder_)方法指定类，但通常使用系统默认的startup，可以通过startup的构造函数进行依赖注入，startup类中必须包含Configure方法同时可以根据实际情况添加ConfigureServices方法，这两个方法均在应用程序运行时被调用。Startup 类的 执行顺序：构造 -> configureServices ->configure

ConfigureServices方法：主要用于服务配置，比如依赖注入(DI)的配置，使用时该方法必须在Configure方法之前

Configure方法：用于应用程序响应HTTP请求，通过向IApplicationBuilder实例添加中间件组件来配置请求管道

**二、自定义路由**

在Startup类的Configure方法配置

public void Configure(IApplicationBuilder app, IHostingEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

#region 自定义路由配置

app.UseMvc(routes =>

{

// 自定义路由

routes.MapRoute(

name: "default1",

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

defaults: new { controller = "Values", action = "Index" });

// 默认路由

routes.MapRoute(

name: "default",

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

defaults: new { controller = "Values", action = "Index" });

});

#endregion

}

View Code

**三、跨域设置**

在Startup类的ConfigureServices方法配置

public void ConfigureServices(IServiceCollection services)

{

#region 跨域设置

services.AddCors(options =>

{

options.AddPolicy("AppDomain", builder =>

{

builder.AllowAnyOrigin() // Allow access to any source from the host

.AllowAnyMethod() // Ensures that the policy allows any method

.AllowAnyHeader() // Ensures that the policy allows any header

.AllowCredentials(); // Specify the processing of cookie

});

});

#endregion

services.AddMvc();

}

其中“AppDomain”这个名字是自定义的，大家可以根据自己的喜好定义不同的名字，配置完成之后，在控制器上面添加[EnableCors("AppDomain")]特性即可，如果要实现全局的跨域设置，可以在Configure方法里面配置app.UseCors("AppDomain")，即能实现全局的跨域设置

**四、自定义读取配置文件信息**

 这里是写的一个公共方法去读取配置文件appsettings.json

using Microsoft.Extensions.Configuration;

using Microsoft.Extensions.Configuration.Json;

using Microsoft.Extensions.DependencyInjection;

using Microsoft.Extensions.Options;

using System.IO;

public class JsonConfigurationHelper

{

public static T GetAppSettings<T>(string key,string path= "appsettings.json") where T : class, new()

{

var currentClassDir = Directory.GetCurrentDirectory();

IConfiguration config = new ConfigurationBuilder()

.SetBasePath(currentClassDir)

.Add(new JsonConfigurationSource { Path = path, Optional = false, ReloadOnChange = true })

.Build();

var appconfig = new ServiceCollection()

.AddOptions()

.Configure<T>(config.GetSection(key))

.BuildServiceProvider()

.GetService<IOptions<T>>()

.Value;

return appconfig;

}

}

/// <summary>

/// 读取配置文件

/// </summary>

/// <returns></returns>

[HttpGet]

public dynamic JsonConfig()

{

var jsonStr = JsonConfigurationHelper.GetAppSettings<ConfigDTO>("config");

return Ok(jsonStr);

}

/// <summary>

/// 实体类

/// </summary>

public class ConfigDTO

{

public dynamic name { get; set; }

}

{

"config": {

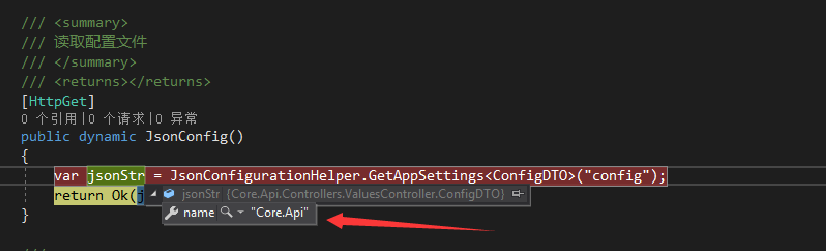
"name": "Core.Api"

}

}

View Code

截图看效果



**五、程序集批量依赖注入**

我们都知道依赖注入主要是为了方便解耦，解除应用程序之间的依赖关系，在我看来DI、IOC这两者差不多是一样的，DI是从应用程序的角度而IOC是从容器的角度，它们主要是对同一件事情的不同角度的描述。然而，，，，，，当我们项目业务比较多的时候，如果要实现多个业务的注入，通常方法是手动一个个的添加注入，这样可能有点太繁琐，所以就想到了利用反射实现批量注入，，，，，，

**方法一**

帮助类

public class RuntimeHelper

{

/// <summary>

/// 获取项目程序集，排除所有的系统程序集(Microsoft.\*\*\*、System.\*\*\*等)、Nuget下载包

/// </summary>

/// <returns></returns>

public static IList<Assembly> GetAllAssemblies()

{

var list = new List<Assembly>();

var deps = DependencyContext.Default;

var libs = deps.CompileLibraries.Where(lib => !lib.Serviceable && lib.Type != "package");//排除所有的系统程序集、Nuget下载包

foreach (var lib in libs)

{

try

{

var assembly = AssemblyLoadContext.Default.LoadFromAssemblyName(new AssemblyName(lib.Name));

list.Add(assembly);

}

catch (Exception)

{

// ignored

}

}

return list;

}

public static Assembly GetAssembly(string assemblyName)

{

return GetAllAssemblies().FirstOrDefault(assembly => assembly.FullName.Contains(assemblyName));

}

public static IList<Type> GetAllTypes()

{

var list = new List<Type>();

foreach (var assembly in GetAllAssemblies())

{

var typeInfos = assembly.DefinedTypes;

foreach (var typeInfo in typeInfos)

{

list.Add(typeInfo.AsType());

}

}

return list;

}

public static IList<Type> GetTypesByAssembly(string assemblyName)

{

var list = new List<Type>();

var assembly = AssemblyLoadContext.Default.LoadFromAssemblyName(new AssemblyName(assemblyName));

var typeInfos = assembly.DefinedTypes;

foreach (var typeInfo in typeInfos)

{

list.Add(typeInfo.AsType());

}

return list;

}

public static Type GetImplementType(string typeName, Type baseInterfaceType)

{

return GetAllTypes().FirstOrDefault(t =>

{

if (t.Name == typeName &&

t.GetTypeInfo().GetInterfaces().Any(b => b.Name == baseInterfaceType.Name))

{

var typeInfo = t.GetTypeInfo();

return typeInfo.IsClass && !typeInfo.IsAbstract && !typeInfo.IsGenericType;

}

return false;

});

}

}

public static class ServiceExtension

{

/// <summary>

/// 用DI批量注入接口程序集中对应的实现类。

/// </summary>

/// <param name="service"></param>

/// <param name="interfaceAssemblyName"></param>

/// <returns></returns>

public static IServiceCollection RegisterAssembly(this IServiceCollection service, string interfaceAssemblyName)

{

if (service == null)

throw new ArgumentNullException(nameof(service));

if (string.IsNullOrEmpty(interfaceAssemblyName))

throw new ArgumentNullException(nameof(interfaceAssemblyName));

var assembly = RuntimeHelper.GetAssembly(interfaceAssemblyName);

if (assembly == null)

{

throw new DllNotFoundException($"the dll \"{interfaceAssemblyName}\" not be found");

}

//过滤掉非接口及泛型接口

var types = assembly.GetTypes().Where(t => t.GetTypeInfo().IsInterface && !t.GetTypeInfo().IsGenericType);

foreach (var type in types)

{

var implementTypeName = type.Name.Substring(1);

var implementType = RuntimeHelper.GetImplementType(implementTypeName, type);

if (implementType != null)

service.AddSingleton(type, implementType);

}

return service;

}

/// <summary>

/// 用DI批量注入接口程序集中对应的实现类。

/// </summary>

/// <param name="service"></param>

/// <param name="interfaceAssemblyName">接口程序集的名称（不包含文件扩展名）</param>

/// <param name="implementAssemblyName">实现程序集的名称（不包含文件扩展名）</param>

/// <returns></returns>

public static IServiceCollection RegisterAssembly(this IServiceCollection service, string interfaceAssemblyName, string implementAssemblyName)

{

if (service == null)

throw new ArgumentNullException(nameof(service));

if (string.IsNullOrEmpty(interfaceAssemblyName))

throw new ArgumentNullException(nameof(interfaceAssemblyName));

if (string.IsNullOrEmpty(implementAssemblyName))

throw new ArgumentNullException(nameof(implementAssemblyName));

var interfaceAssembly = RuntimeHelper.GetAssembly(interfaceAssemblyName);

if (interfaceAssembly == null)

{

throw new DllNotFoundException($"the dll \"{interfaceAssemblyName}\" not be found");

}

var implementAssembly = RuntimeHelper.GetAssembly(implementAssemblyName);

if (implementAssembly == null)

{

throw new DllNotFoundException($"the dll \"{implementAssemblyName}\" not be found");

}

//过滤掉非接口及泛型接口

var types = interfaceAssembly.GetTypes().Where(t => t.GetTypeInfo().IsInterface && !t.GetTypeInfo().IsGenericType);

foreach (var type in types)

{

//过滤掉抽象类、泛型类以及非class

var implementType = implementAssembly.DefinedTypes

.FirstOrDefault(t => t.IsClass && !t.IsAbstract && !t.IsGenericType &&

t.GetInterfaces().Any(b => b.Name == type.Name));

if (implementType != null)

{

service.AddSingleton(type, implementType.AsType());

}

}

return service;

}

}

View Code

 在Startupl类的ConfigureServices方法中添加

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

public void ConfigureServices(IServiceCollection services)

{

#region 程序集批量依赖注入

services.RegisterAssembly("Core.BLL");

#endregion

services.AddMvc();

}

View Code

调用(Ps:Core.BLL这个类库里面分别有一个接口IAccountService和一个类AccountService，AccountService类去继承接口IAccountService并实现接口里面的方法)

public interface IAccountService

{

int GetLst();

}

public class AccountService: IAccountService

{

public int GetLst()

{

return 1;

}

}

public class ValuesController : Controller

{

private readonly IAccountService \_accountService;

public ValuesController(IAccountService accountService)

{

\_accountService = accountService;

}

[HttpGet]

public dynamic GetAccount()

{

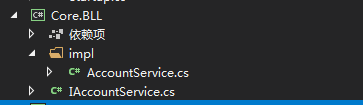
var result = this.\_accountService.GetLst();

return Ok();

}

}

View Code



**方法二**

public static class InjectionExtension

{

/// <summary>

/// 批量注入接口程序集中对应的实现类(接口和实现类在同一个程序集时)

/// </summary>

/// <param name="services">services</param>

/// <param name="assemblyName">程序集名称</param>

public static void BatchAddScoped(this IServiceCollection services, string assemblyName)

{

if (services == null)

throw new ArgumentNullException(nameof(services));

if (assemblyName == null)

throw new ArgumentNullException(nameof(assemblyName));

// 排除所有的系统程序集，Nuget下载包

var libs = DependencyContext.Default.CompileLibraries.Where(lib => !lib.Serviceable && lib.Type != "package");

var serviceLib = libs.Where(c => c.Name.Contains(assemblyName)).FirstOrDefault();

var assembly = AssemblyLoadContext.Default.LoadFromAssemblyName(new AssemblyName(serviceLib.Name));

var serviceClassList = assembly.GetTypes().Where(x=>x.IsInterface).ToList();

foreach (var item in serviceClassList)

{

var implementName = item.Name.Substring(1,item.Name.Length-1);

var implementType= assembly.GetTypes().Where(c => c.IsClass && c.Name == implementName).FirstOrDefault();

if (implementType == null) continue;

services.AddScoped(item, implementType);

}

}

/// <summary>

/// 批量注入接口程序集中对应的实现类(接口和实现类在不同程序集时)

/// </summary>

/// <param name="services"></param>

/// <param name="interfaceAssemblyName"></param>

/// <param name="implementAssemblyName"></param>

public static void BatchAddScoped(this IServiceCollection services, string interfaceAssemblyName, string implementAssemblyName)

{

if (services == null)

throw new ArgumentNullException(nameof(services));

if (string.IsNullOrEmpty(interfaceAssemblyName))

throw new ArgumentNullException(nameof(interfaceAssemblyName));

if (string.IsNullOrEmpty(implementAssemblyName))

throw new ArgumentNullException(nameof(implementAssemblyName));

// 排除所有的系统程序集，Nuget下载包

var libs = DependencyContext.Default.CompileLibraries.Where(lib => !lib.Serviceable && lib.Type != "package");

var serviceInterfaceLib = libs.Where(c => c.Name.Contains(interfaceAssemblyName)).FirstOrDefault();

var interfaceAssembly = AssemblyLoadContext.Default.LoadFromAssemblyName(new AssemblyName(serviceInterfaceLib.Name));

// 过滤非接口

var serviceInterfaceList = interfaceAssembly.GetTypes().Where(x => x.IsInterface).ToList();

var serviceImplementLib = libs.Where(c => c.Name.Contains(implementAssemblyName)).FirstOrDefault();

var implementAssembly = AssemblyLoadContext.Default.LoadFromAssemblyName(new AssemblyName(serviceImplementLib.Name));

// 过滤抽象类、泛型类以及非class

foreach (var item in serviceInterfaceList)

{

var implementName = item.Name.Substring(1, item.Name.Length - 1);

var implementType = implementAssembly.GetTypes().Where(c => c.IsClass && c.Name == implementName).FirstOrDefault();

if (implementType == null) continue;

services.AddScoped(item, implementType);

}

}

}

public void ConfigureServices(IServiceCollection services)

{

#region 程序集批量依赖注入

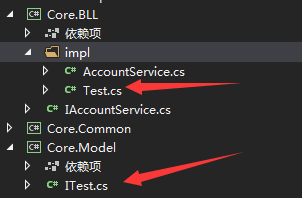
services.BatchAddScoped("Core.BLL"); // 接口和实现类在同一个程序集

services.BatchAddScoped("Core.Model", "Core.BLL");// 接口和实现类在不同程序集

#endregion

}

View Code



**六、使用NLog写入文件日志**

新建配置文件命名为Nlog.config

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

<nlog xmlns="http://www.nlog-project.org/schemas/NLog.xsd" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

<targets>

<!--写入文件-->

<target

xsi:type="File"

name="DebugFile"

fileName="Logs\Debug\${shortdate}.log"

layout="日志时间：${longdate}${newline}日志来源：${callsite}${newline}日志级别：${uppercase:${level}}${newline}消息内容：${message}${newline}异常信息：${exception}${newline}==============================================================${newline}" >

</target>

<target

xsi:type="File"

name="InfoFile"

fileName="Logs\Info\${shortdate}.log"

layout="日志时间：${longdate}${newline}日志来源：${callsite}${newline}日志级别：${uppercase:${level}}${newline}消息内容：${message}${newline}异常信息：${exception}${newline}==============================================================${newline}" >

</target>

<target

xsi:type="File"

name="ErrorFile"

fileName="Logs\Error\${shortdate}.log"

layout="日志时间：${longdate}${newline}日志来源：${callsite}${newline}日志级别：${uppercase:${level}}${newline}消息内容：${message}${newline}异常信息：${exception}${newline}==============================================================${newline}" >

</target>

<rules>

<logger name="FileLogger" minlevel="Debug" maxLevel="Debug" writeTo="DebugFile" />

<logger name="FileLogger" minlevel="Info" maxLevel="Info" writeTo="InfoFile" />

<logger name="FileLogger" minlevel="Error" maxLevel="Error" writeTo="ErrorFile" />

</rules>

</nlog>

View Code

 在Startup类Configure方法中添加配置

public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

#region NLog配置

loggerFactory.AddNLog(); // 添加NLog

loggerFactory.ConfigureNLog($"{Directory.GetCurrentDirectory()}\\Nlog.config"); // 添加Nlog.config配置文件

loggerFactory.AddDebug();

#endregion

}

View Code

写入日志到文件

public class ValuesController : Controller

{

private readonly Logger \_logger;

public ValuesController()

{

\_logger = LogManager.GetLogger("FileLogger");

}

/// <summary>

/// 写入文件日志

/// </summary>

/// <returns></returns>

[HttpGet]

public dynamic WriteLogToFile()

{

\_logger.Info("写入Info文件");

\_logger.Debug("写入Debug文件");

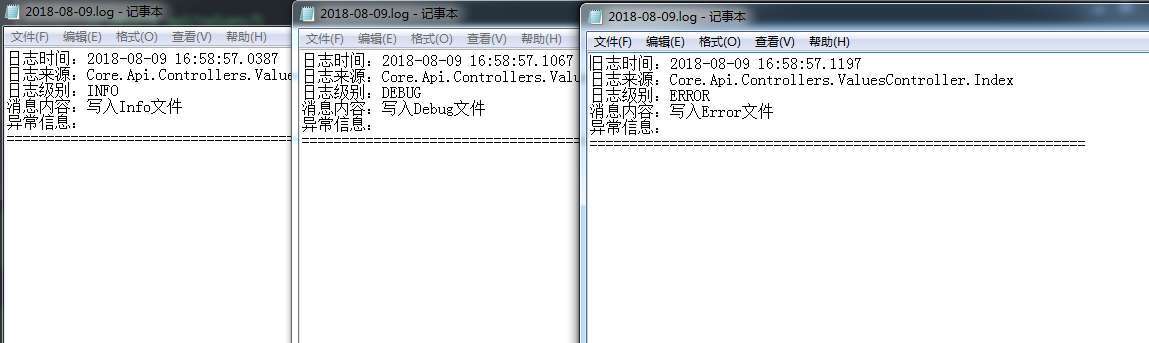
\_logger.Error("写入Error文件");

return Ok();

}

}

View Code



**七、使用NLog写入数据库日志**

添加依赖项：Microsoft.Extensions.Logging和NLog.Extensions.Logging

新建配置文件命名为Nlog.config

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

<nlog xmlns="http://www.nlog-project.org/schemas/NLog.xsd" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

<targets>

<!--写入数据库-->

<target xsi:type="Database" name="Database"

connectionString="Data Source=.;Initial Catalog=MyDb;Persist Security Info=True;User ID=sa;Password=123456"

commandText="insert into NLog\_Log([CreateOn],[Origin],[LogLevel], [Message], [Exception],[StackTrace],[Desc]) values (getdate(), @origin, @logLevel, @message,@exception, @stackTrace,@desc)">

<!--日志来源-->

<parameter name="@origin" layout="${callsite}"/>

<!--日志等级-->

<parameter name="@logLevel" layout="${level}"/>

<!--日志消息-->

<parameter name="@message" layout="${message}"/>

<!--异常信息-->

<parameter name="@exception" layout="${exception}" />

<!--堆栈信息-->

<parameter name="@stackTrace" layout="${stacktrace}"/>

<!--自定义消息内容-->

<parameter name="@desc" layout="${event-context:item=Desc}"/>

</target>

</targets>

<rules>

<logger name="DbLogger" levels="Trace,Debug,Info,Error" writeTo="Database"/>

</rules>

</nlog>

View Code

同第六项代码一样，也是在Configure方法设置，写入日志到数据库

/// <summary>

/// 将日志写入数据库

/// </summary>

/// <returns></returns>

[HttpGet]

public dynamic WriteLogToDb()

{

Logger \_dblogger = LogManager.GetLogger("DbLogger");

LogEventInfo ei = new LogEventInfo();

ei.Properties["Desc"] = "我是自定义消息";

\_dblogger.Info(ei);

\_dblogger.Debug(ei);

\_dblogger.Trace(ei);

return Ok();

}

USE [MyDb]

GO

/\*\*\*\*\*\* Object: Table [dbo].[NLog\_Log] Script Date: 08/09/2018 17:13:20 \*\*\*\*\*\*/

SET ANSI\_NULLS ON

GO

SET QUOTED\_IDENTIFIER ON

GO

CREATE TABLE [dbo].[NLog\_Log](

[ID] [int] IDENTITY(**1**,**1**) NOT NULL,

[Origin] [nvarchar](**500**) NULL,

[LogLevel] [nvarchar](**500**) NULL,

[Message] [nvarchar](**500**) NULL,

[Desc] [nvarchar](**500**) NULL,

[Exception] [nvarchar](**500**) NULL,

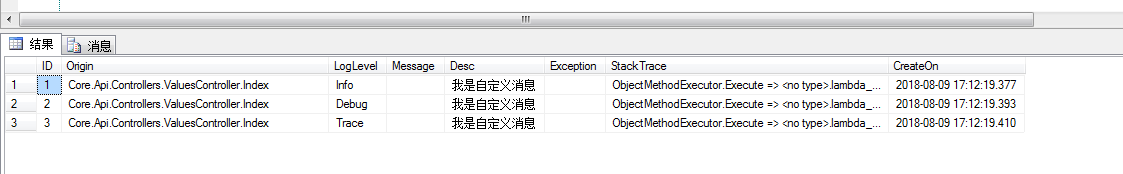
[StackTrace] [nvarchar](**500**) NULL,

[CreateOn] [datetime] NULL

) ON [PRIMARY]

GO

View Code



**八、Nlog标签解读**

NLog的使用方式基本上和其它的Log库差不多，用于输出日志的级别包括：Trace,Debug,Info,Warn,Error,Fatal

<nlog>标签

autoReload 修改配置文件后是否允许自动加载无须重启程序

throwExceptions 内部日志系统抛出异常

internalLogLevel 可选Trace|Debug|Info|Warn|Error|Fatal决定内部日志的级别 Off 关闭

internalLogFile 把内部的调试和异常信息都写入指定文件里

建议throwExceptions的值设为“false”，这样由于日志引发的问题不至于导致应用程序的崩溃。

<targets>标签

<target />区域定义了日志的目标或者说输出 ，,在这里可以按需设置文件名称和格式，输出方式。

name：自定义该target的名字,可供rule规则里使用

type： 定义类型,官方提供的可选类型有:

Chainsaw|ColoredConsole |Console |Database|Debug|Debugger|EventLog|File|LogReceiverService|Mail|Memory|MethodCall|Network |NLogViewer|Null |OutputDebugString|PerfCounter|Trace|WebService

不过常用的还是 File \Database \Colored Console\ Mail

layouts 用来规定布局样式,语法“${属性}”,可以把上下文信息插入到日志中，更多布局渲染器可参考[https://github.com/nlog/NLog/wiki/Layout%20Renderers](https://github.com/nlog/NLog/wiki/Layout%20Renderers" \t "_blank)

<rules>标签

各种规则配置在logger里

name - 记录者的名字

minlevel - 最低级别

maxlevel - 最高级别

level - 单一日志级别

levels - 一系列日志级别，由逗号分隔。

writeTo - 规则匹配时日志应该被写入的一系列目标，由逗号分隔。

**九、启用Session**

Net Core默认情况下是没有启用Session的，需要在Startup文件ConfigureServices方法配置启动，否则无法使用（需要在UseMvc之前，否则会报错）

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttps://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

public void ConfigureServices(IServiceCollection services)

{

// 启用Session

services.AddSession();

services.AddMvc();

}

View Code

**十、json数据，自定义日期格式**

在webapi数据返回中，经常碰到json数据日期带“T”的问题，可以在Startup类ConfigureServices方法中做如下全局配置

public void ConfigureServices(IServiceCollection services)

{

services.AddMvc().AddJsonOptions(options =>

{

options.SerializerSettings.DateFormatString = "yyyy-MM-dd HH:mm:ss"; // 日期格式化

});

}

View Code

**十一、json数据，string类型字段返回为null时默认返回空字符串**

帮助类

public sealed class NullWithEmptyStringResolver : DefaultContractResolver

{

protected override IList<JsonProperty> CreateProperties(Type type, MemberSerialization memberSerialization)

{

return type.GetProperties()

.Select(p =>

{

var jp = base.CreateProperty(p, memberSerialization);

jp.ValueProvider = new NullToEmptyStringValueProvider(p);

return jp;

}).ToList();

}

/// <summary>

/// 将所有返回字段转换为小写

/// </summary>

/// <param name="propertyName"></param>

/// <returns></returns>

//protected override string ResolvePropertyName(string propertyName)

//{

// return propertyName.ToLower();

//}

}

public class NullToEmptyStringValueProvider : IValueProvider

{

PropertyInfo \_MemberInfo;

public NullToEmptyStringValueProvider(PropertyInfo memberInfo)

{

\_MemberInfo = memberInfo;

}

public object GetValue(object target)

{

object result = \_MemberInfo.GetValue(target);

if (result == null)

{

var type = \_MemberInfo.PropertyType;

if (type == typeof(string)) result = "";

//else if (type == typeof(DateTime?))

// result = new DateTime(1, 1, 1);

}

return result;

}

public void SetValue(object target, object value)

{

\_MemberInfo.SetValue(target, value);

}

}

View Code

在Startup类ConfigureServices方法中做如下全局配置

public void ConfigureServices(IServiceCollection services)

{

services.AddMvc().AddJsonOptions(options =>

{

options.SerializerSettings.Formatting = Formatting.Indented; // 返回数据格式缩进(按需配置)

options.SerializerSettings.ContractResolver = new NullWithEmptyStringResolver(); // 字段为字符串返回为null时，默认返回空

});

}

View Code

api后台代码

public class ValuesController : Controller

{

[HttpGet]

public dynamic Index()

{

List<userinfo> list = new List<userinfo>()

{

new userinfo(){ UserName=null }

};

return list;

}

public class userinfo

{

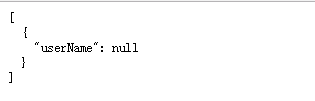
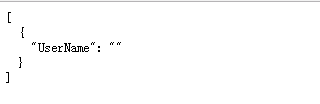
public string UserName { get; set; }

}

}

View Code

配置前和配置之后数据在浏览器中返回效果截图

大家看效果图有没有发现一个问题，我在没有配置时，实体里面的“UserName”字段默认被转换成了小写，这显然不符合我们的要求，当我配置之后就可以返回同实体里面的字段大小写格式一致了，同时为null的字段默认返回了空字符串，是不是美滋滋，，，，

可能有朋友会说，我只想让返回的数据字段同实体的数据字段一致，而字段为null的值依然还是让他返回null，其实这样也可以，看第十二项配置操作即可。

**十二、Json数据，返回字段同实体字段大小写一致**

在.net core中，webapi返回的数据字段，首字母默认被转换成了小写，在Startup类ConfigureServices方法中配置：

public void ConfigureServices(IServiceCollection services)

{

services.AddMvc().AddJsonOptions(options =>

{

options.SerializerSettings.ContractResolver = new Newtonsoft.Json.Serialization.DefaultContractResolver();

});

}

public class ValuesController : Controller

{

[HttpGet]

public dynamic Index()

{

List<userinfo> list = new List<userinfo>()

{

new userinfo(){ UserName=null }

};

return list;

}

public class userinfo

{

public string UserName { get; set; }

}

}

View Code

配置前和配置之后数据在浏览器中返回效果截图

