.Net Core WebApi Configuration

提到“配置”二字，我想绝大部分.NET开发人员脑海中会立马浮现出两个特殊文件的身影，那就是我们再熟悉不过的app.config和web.config，多年以来我们已经习惯了将结构化的配置信息定义在这两个文件之中。到了.NET Core的时候，很多我们习以为常的东西都发生了改变，其中也包括定义配置的方式。总的来说，新的配置系统显得更加轻量级，并且具有更好的扩展性，其最大的特点就是支持多样化的数据源。我们可以采用内存的变量作为配置的数据源，也可以直接配置定义在持久化的文件甚至数据库中。

针对配置的API涉及三个对象，它们分别是Configuration、ConfigurationBuilder和ConfigurationProvider，配置模型中具有相应的接口来表示它们。这三个对象之间的关系很清晰:

Configuration - 对象承载着在编程过程中使用的配置信息，

ConfigurationProvider - 则是配置信息原始数据源的提供者，两者之间沟通由ConfigurationBuilder来完成，它利用  
 ConfigurationProvider提取源数据将其转换为Configuration对象。

ConfigurationBuilder – ConfigurationProvider 与 Configuration 之间的沟通桥梁，将数据源转化为 Configuration对象

using Microsoft.Extensions.Configuration;

using Microsoft.Extensions.Configuration.Memory;

[**配置模型详解**](https://www.cnblogs.com/artech/p/asp-net-core-config-02.html)

# 一、配置的三种结构

## 逻辑结构 - 配置的逻辑结构就是Configuration对象所体现的结构, 配置在逻辑上呈现为一种树形结构，我们称之为配置树，组成这棵树的某个节点就体现为一个Configuration对象。表现为键值对的原子配置项存储于叶子节点中，而非叶子节点仅仅体现为一个配置节点的逻辑容器，自身不包含具体的配置数据。

public class DateTimeFormatSettings

{

public string LongDatePattern { get; set; }

public string LongTimePattern { get; set; }

public string ShortDatePattern { get; set; }

public string ShortTimePattern { get; set; }

public DateTimeFormatSettings (IConfiguration configuration)

{

this.LongDatePattern = configuration["LongDatePattern"];

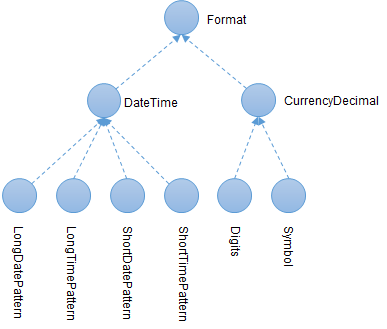
this.LongTimePattern = configuration["LongTimePattern"];

this.ShortDatePattern = configuration["ShortDatePattern"];

this.ShortTimePattern = configuration["ShortTimePattern"];

}

}



public class CurrencyDecimalFormatSettings

{

public int Digits { get; set; }

public string Symbol { get; set; }

public CurrencyDecimalFormatSettings(IConfiguration configuration)

{

this.Digits = int.Parse(configuration["Digits"]);

this.Symbol = configuration["Symbol"];

}

}

public class FormatSettings

{

public DateTimeFormatSettings DateTime { get; set; }

public CurrencyDecimalFormatSettings CurrencyDecimal { get; set; }

public FormatSettings(IConfiguration configuration)

{

this.DateTime = new DateTimeFormatSettings(configuration.GetSection("DateTime"));

this.CurrencyDecimal = new CurrencyDecimalFormatSettings(configuration.GetSection("CurrencyDecimal"));

}

}

## 原始结构 - 配置采用怎样的原始结构取决于我们采用何种方式定义它。最常见的配置源体现为采用某个格式的文本文件，那么配置的原始结构则由文件的格式来决定。

对于上面我们定义的FormatSettings类型，我们可以按照如下的形式以XML和JSON的格式来定义其配置。

XML:

1: <Format>

2: <DateTime>

3: <LongDatePattern>dddd, MMMM d, yyyy</LongDatePattern>

4: <LongTimePattern>h:mm:ss tt</LongTimePattern>

5: <ShortDatePattern>M/d/yyyy</ShortDatePattern>

6: <ShortTimePattern>h:mm tt</ShortTimePattern>

7: </DateTime>

8: <CurrencyDecimal>

9: <Digits>2</Digits>

10: <Symbol>$</Symbol>

11: </CurrencyDecimal>

12: </Format>

JSON:

1: {

2: "format": {

3: "dateTime": {

4: "longDatePattern" : "dddd, MMMM d, yyyy",

5: "longTimePattern" : "h:mm:ss tt",

6: "shortDatePattern" : "M/d/yyyy",

7: "shortTimePattern" : "h:mm tt"

8: },

9: "currencyDecimal": {

10: "digits": "2",

11: "symbol": "$"

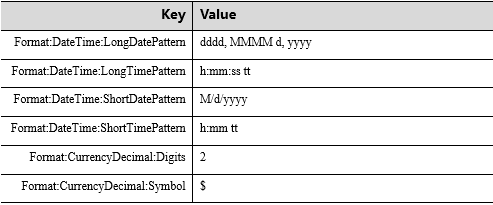
12: }

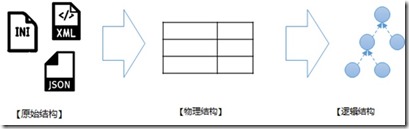
13: }

14: }

## 物理结构 - 配置模型的终极目的就是将配置从原始结构转换成逻辑结构。不过在进行结构转化的时候，它并不会直接将原始的配置数据转换成一个Configuration对象，它们之间由一种被我称为物理结构的中间结构作为过度。配置的物理结构体现为一个简单的数据字典。同样是针对FormatSettings这个类型，对应的配置将具有如下表所示的物理结构。

Dictionary<string, string> - Key 使用冒号 “:” 来分隔树形结构的层级关系





# 二、Configuration

## ConfigurationRoot - 一个ConfigurationRoot对象表示配置数的根节点，如果它被重新加载了，那么这颗配置树承载的所有配置数据均被重新加载了。

public interface IConfigurationRoot : IConfiguration

{

void Reload();

}

## ConfigurationSection - 一个ConfigurationSection对应着配置树中某个非根配置节。IConfigurationSection具有如下三个属性，只读属性Key用来唯一标识多个“同父”配置节，而另一个只读属性Path则表示从根节点到父节点的路径，该路径由ConfigurationSection的Key组成，并采用冒号作为分隔符。Path和Key的组合体现了当前配置节在整个配置树中的位置。

public interface IConfigurationSection : IConfiguration

{

string Path { get; }

string Key { get; }

string Value { get; set; }

}

IConfigurationSection的Value属性表示配置节的值,

只有配置树叶子结点对应的ConfigurationSection对象才具有值，

非叶子节点对应的ConfigurationSection对象仅仅表示一组隶属于它的所有子配置节的逻辑容器，它们的Value一般返回Null。

# 三、ConfigurationProvider

则是配置信息原始数据源的提供者，两者之间沟通由ConfigurationBuilder来完成，它利用ConfigurationProvider提取源数据将其转换为Configuration对象。

public interface IConfigurationProvider

{

IEnumerable<string> GetChildKeys(IEnumerable<string> earlierKeys, string parentPath);

IChangeToken GetReloadToken();

void Load();

void Set(string key, string value);

bool TryGet(string key, out string value);

}

public abstract class ConfigurationProvider : IConfigurationProvider

{

protected ConfigurationProvider();

protected IDictionary<string, string> Data { get; set; }

public virtual IEnumerable<string> GetChildKeys(IEnumerable<string> earlierKeys, string parentPath);

public IChangeToken GetReloadToken();

public virtual void Load();

public virtual void Set(string key, string value);

public virtual bool TryGet(string key, out string value);

protected void OnReload();

}

比较常用的MemoryConfigurationProvider

public class MemoryConfigurationProvider : ConfigurationProvider, IEnumerable<KeyValuePair<string, string>>, IEnumerable

{

public MemoryConfigurationProvider(MemoryConfigurationSource source);

public void Add(string key, string value);

public IEnumerator<KeyValuePair<string, string>> GetEnumerator();

}

如何使用：

Dictionary<string, string> ms = new Dictionary<string, string> {

{ "good:0:monday", "Mon1"},

{"good:0:tue", "Tues1"},

{"good:0:wed", "Wed1"},

{"good:0:thu", "Thu1"},

{"good:1:monday", "Mon2"},

{"good:1:tue", "Tues2"},

{"good:1:wed", "Wed2"},

{"good:1:thu", "Thu2"},

{"good:2:monday", "Mon3"},

{"good:2:tue", "Tues3"},

{"good:2:wed", "Wed3"},

{"good:2:thu", "Thu3"},

{"good:3:monday", "Mon4"},

{"good:3:tue", "Tues4"},

{"good:3:wed", "Wed4"},

{"good:3:thu", "Thu4"}

};

方法一：

MemoryConfigurationSource mmsource = new MemoryConfigurationSource();

mmsource.InitialData = ms;

IConfigurationProvider icp = mmsource.Build(new ConfigurationBuilder());

string thu = string.Empty;

icp.TryGet("good:3:thu", out thu);

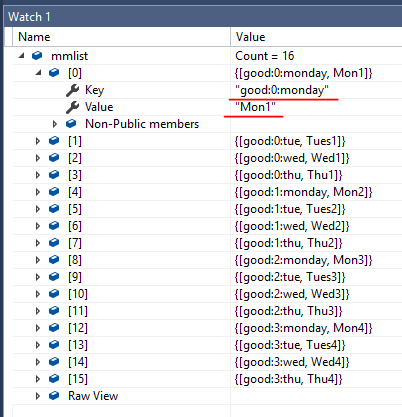
方法二：

MemoryConfigurationSource mmsource = new MemoryConfigurationSource();

mmsource.InitialData = ms;

MemoryConfigurationProvider mmprovider = new MemoryConfigurationProvider(mmsource);

var mmlist = mmprovider.ToList();

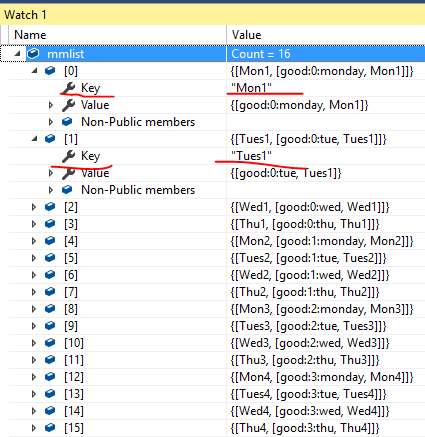
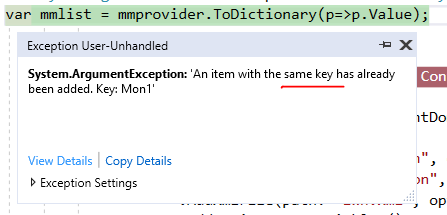


MemoryConfigurationSource mmsource = new MemoryConfigurationSource();

mmsource.InitialData = ms;

MemoryConfigurationProvider mmprovider = new MemoryConfigurationProvider(mmsource);

var mmlist = mmprovider.ToDictionary(p=>p.Value); 如果 p.Value 有重复值，则会抛出错误

单独使用MemoryConfigurationSource 和IConfigurationProvider 或者 MemoryConfigurationProvider 没有什么意义，和直接操作 List<KeyPairValue<string, string>> 没有区别

# 四、ConfigurationBuilder

ConfigurationProvider 与 Configuration 之间的沟通桥梁，将数据源转化为 Configuration对象

public interface IConfigurationBuilder

{

IDictionary<string, object> Properties { get; }

IList<IConfigurationSource> Sources { get; }

IConfigurationBuilder Add(IConfigurationSource source);

IConfigurationRoot Build();

}

# [多样性的配置来源[上篇]](https://www.cnblogs.com/artech/p/asp-net-core-config-4-1.html)

* MemoryConfigurationProvider
* EnvironmentVariablesConfigurationProvider
* CommandLineConfigurationProvider
* JsonConfigurationProvider
* XmlConfiguationProvider
* IniConfigurationProvider
* 自定义ConfigurationProvider

如果添加

IConfigurationBuilder 提供了许多 AddXXX 方法:

MemoryConfigurationSource mmsource = new MemoryConfigurationSource();

mmsource.InitialData = ms;

IConfigurationBuilder builder = new ConfigurationBuilder();

builder.SetBasePath(AppDomain.CurrentDomain.BaseDirectory)

.AddInMemoryCollection(ms)

.Add(mmsource)

.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 configRoot = builder.Build();

services.AddSingleton<IConfiguration>(configRoot);

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"

}

}

LWH.XML

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

<servers>

<database>

<host>D3Server</host>

<user>SA</user>

<trust>true</trust>

<yeep>XML Yeep</yeep>

</database>

</servers>

最终生成的Configuration对象是合并的结果，相同键值是后面的覆盖前面的值。

获取数据的方法：

var website = configRoot.GetSection("website").Value; - 是叶子节点，所以Value = www.d3security.com

var ftp = configRoot.GetSection("ftp").Value; - null 因为是非叶子节点，是容器

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Name** | **Value** | **Type** |
|  | website | "www.d3security.com" | string |
|  | ftp | null | string |

var website = configRoot.GetSection("database").GetValue(typeof(string), "host");

var ftp = configRoot.GetSection("ftp").GetValue<string>("server");

string ftp = configRoot.GetValue<string>("ftp:server");

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Name** | **Value** | **Type** |
|  | website | "D3Server" | object {string} |
|  | ftp | "ftp://qa.d3security.com" | string |

可以使用强类型获取数据，通过定义数据模型来获取

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 string memory { 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; }

}

Database db = configRoot.GetSection("database").Get<Database>();

Mftp ftp = configRoot.GetSection("ftp").Get<Mftp>();

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Name** | **Value** | **Type** |
| ◢ | db | {WebCore1.Database} | WebCore1.Database |
|  | flag | "good" | string |
|  | host | "D3Server" | string |
|  | memory | "64GB" | string |
|  | pwd | 889 | int |
|  | user | "SA" | string |
|  | yeep | "XML Yeep" | string |
| ◢ | ftp | {WebCore1.Mftp} | WebCore1.Mftp |
|  | password | "d3456" | string |
|  | server | "ftp://qa.d3security.com" | string |
|  | trans | "Vancouver" | string |
|  | uid | 998 | int |

可以通过类型的构建函数来解析数据：

public class iDB {

public string host { get; set; }

public string user { get; set; }

public int pwd { get { return \_pwd; } set { \_pwd = value; } }

private int \_pwd;

public string flag { get; set; }

public string yeep { get; set; }

public iDB(IConfiguration con)

{

this.host = con["host"];

this.user = con["user"];

int.TryParse(con["pwd"], out \_pwd);

this.flag = con["pwd"];

this.yeep = con["yeep"];

}

}

iDB db1 = new iDB(configRoot.GetSection("database"));

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Name** | **Value** | **Type** |
| ◢ | db1 | {WebCore1.iDB} | WebCore1.iDB |
|  | \_pwd | 889 | int |
|  | flag | "889" | string |
|  | host | "D3Server" | string |
|  | pwd | 889 | int |
|  | user | "SA" | string |
|  | yeep | "XML Yeep" | string |

public class iDB {

public string host { get; set; }

public string user { get; set; }

public int pwd { get { return \_pwd; } set { \_pwd = value; } }

private int \_pwd;

public string flag { get; set; }

public string yeep { get; set; }

public iDB(IConfiguration con)

{

this.host = con.GetValue<string>("host");

this.user = con.GetValue<string>("user");

this.pwd = con.GetValue<int>("pwd");

this.flag = con.GetValue<string>("pwd");

this.yeep = con.GetValue<string>("yeep");

}

}

处理列表类型：

public class GOOD

{

public string monday { get; set; }

public string tue { get; set; }

public string wed { get; set; }

public string thu { get; set; }

}

IList<GOOD> glist = configRoot.GetSection("good").Get<IList<GOOD>>();

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Name** | **Value** | **Type** |
| ◢ | glist | Count = 4 | System.Collections.Generic.IList<WebCore1.GOOD> {System.Collections.Generic.List<WebCore1.GOOD>} |
|  | ◢ [0] | {WebCore1.GOOD} | WebCore1.GOOD |
|  | monday | "Mon1" | string |
|  | thu | "Thu1" | string |
|  | tue | "Tues1" | string |
|  | wed | "Wed1" | string |
|  | ◢ [1] | {WebCore1.GOOD} | WebCore1.GOOD |
|  | monday | "Mon1" | string |
|  | thu | "Thu2" | string |
|  | tue | "Tues2" | string |
|  | wed | "Wed2" | string |
|  | ◢ [2] | {WebCore1.GOOD} | WebCore1.GOOD |
|  | monday | "Mon3" | string |
|  | thu | "Thu3" | string |
|  | tue | "Tues3" | string |
|  | wed | "Wed3" | string |
|  | ◢ [3] | {WebCore1.GOOD} | WebCore1.GOOD |
|  | monday | "Mon4" | string |
|  | thu | "Thu4" | string |
|  | tue | "Tues4" | string |
|  | wed | "Wed4" | string |

处理字典类型：

Dictionary<string, string> ms = new Dictionary<string, string> {

{"database:memory", "64GB" },

{ "good:A:monday", "Mon1"},

{"good:A:tue", "Tues1"},

{"good:A:wed", "Wed1"},

{"good:A:thu", "Thu1"},

{"good:B:monday", "Mon1"},

{"good:B:tue", "Tues2"},

{"good:B:wed", "Wed2"},

{"good:B:thu", "Thu2"},

{"good:C:monday", "Mon3"},

{"good:C:tue", "Tues3"},

{"good:C:wed", "Wed3"},

{"good:C:thu", "Thu3"},

{"good:D:monday", "Mon4"},

{"good:D:tue", "Tues4"},

{"good:D:wed", "Wed4"},

{"good:D:thu", "Thu4"}

};

Dictionary<string, GOOD> glist = configRoot.GetSection("good").Get<Dictionary<string, GOOD>>();

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Name** | **Value** | **Type** |
| ◢ | glist | Count = 4 | System.Collections.Generic.Dictionary<string, WebCore1.GOOD> |
|  | ◢ [0] | {[A, WebCore1.GOOD]} | System.Collections.Generic.KeyValuePair<string, WebCore1.GOOD> |
|  | Key | "A" | string |
|  | ▶ Value | {WebCore1.GOOD} | WebCore1.GOOD |
|  | ▶ Non-Public members |  |  |
|  | ◢ [1] | {[B, WebCore1.GOOD]} | System.Collections.Generic.KeyValuePair<string, WebCore1.GOOD> |
|  | Key | "B" | string |
|  | ◢ Value | {WebCore1.GOOD} | WebCore1.GOOD |
|  | monday | "Mon1" | string |
|  | thu | "Thu2" | string |
|  | tue | "Tues2" | string |
|  | wed | "Wed2" | string |
|  | ▶ Non-Public members |  |  |
|  | ◢ [2] | {[C, WebCore1.GOOD]} | System.Collections.Generic.KeyValuePair<string, WebCore1.GOOD> |
|  | Key | "C" | string |
|  | ◢ Value | {WebCore1.GOOD} | WebCore1.GOOD |
|  | monday | "Mon3" | string |
|  | thu | "Thu3" | string |
|  | tue | "Tues3" | string |
|  | wed | "Wed3" | string |
|  | ▶ Non-Public members |  |  |
|  | ◢ [3] | {[D, WebCore1.GOOD]} | System.Collections.Generic.KeyValuePair<string, WebCore1.GOOD> |
|  | Key | "D" | string |
|  | ◢ Value | {WebCore1.GOOD} | WebCore1.GOOD |
|  | monday | "Mon4" | string |
|  | thu | "Thu4" | string |
|  | tue | "Tues4" | string |
|  | wed | "Wed4" | string |
|  | ▶ Non-Public members |  |  |
|  | ▶ Raw View |  |  |

Get() 和 GetValue(key) 区别, 以及 Get<T>() 和 GetValue<T>(key) 的区别

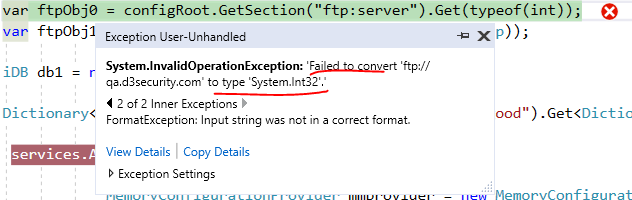
1. Get() – 如果 IConfigurationSection 存在值，则返回值， 如果值为null, 那么会自动递归绑定属性到指定的类型

.Get(Type) – 返回的是 object 类型, Type 是将值或者自动绑定的属性 转化为 Type 类型，如果转化失败则抛出错误

var ftpObj0 = configRoot.GetSection("ftp:server").Get(typeof(string));

var ftpObj1 = configRoot.GetSection("ftp").Get(typeof(Mftp));

var ftpObj0 = configRoot.GetSection("ftp:server").Get(typeof(int)); - string要转化为int,出错



.Get<Type>() - 返回类型是 Type 类型， 但是如果转化失败也是会抛出错误的, 用这个方法更加简洁直观

string ftpObj0 = configRoot.GetSection("ftp:server").Get<string>();

Mftp ftpObj1 = configRoot.GetSection("ftp").Get<Mftp>();

我再来看看， 递归绑定的例子

{

"ftp": {

"server": {

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

"ip": "192.168.1.200",

"port": 4530

},

"uid": "567",

"password": "d3456"

}

}

public class Mftp

{

public Server { get; set; }

public int uid { get; set; }

public string password { get; set; }

public string trans { get; set; }

}

public class Server

{

public string host { get; set; }

public string ip { get; set; }

public int port { get; set; }

}

Server ftpObj0 = configRoot.GetSection("ftp:server").Get<Server>();

Mftp ftpObj1 = configRoot.GetSection("ftp").Get<Mftp>();

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Name** | **Value** | **Type** |
| ◢ | ftpObj0 | {WebCore1.Server} | WebCore1.Server |
|  | host | "ftp://qa.d3security.com" | string |
|  | ip | "192.168.1.200" | string |
|  | port | 4530 | int |
| ◢ | ftpObj1 | {WebCore1.Mftp} | WebCore1.Mftp |
|  | password | "d3456" | string |
|  | ◢ server | {WebCore1.Server} | WebCore1.Server |
|  | host | "ftp://qa.d3security.com" | string |
|  | ip | "192.168.1.200" | string |
|  | port | 4530 | int |
|  | trans | "Vancouver" | string |
|  | uid | 998 | int |

Dictionary<string, GOOD> glist = configRoot.GetSection("good").Get<Dictionary<string, GOOD>>();

IList<GOOD> glist = configRoot.GetSection("good").Get<IList<GOOD>>();

1. GetValue() – 只是针对IConfigurationSection 的 Value 来转化， 如果是非叶子节点，值为null, 则返回值也是null. 并不会自动递归绑定属性到指定类型

string ftp0 = configRoot.GetValue<string>("ftp:server:host");

string ftp1 = configRoot.GetSection("ftp:server").GetValue<string>("notNull");

string ftp2 = configRoot.GetSection("ftp").GetSection("server").GetValue<string>("host");

int ftp3 = configRoot.GetValue<int>("ftp:server");

DateTime ftp4 = configRoot.GetValue<DateTime>("ftp:server:port");

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Name** | **Value** | **Type** |
|  | ftp0 | "ftp://qa.d3security.com" | string |
|  | ftp1 | null | string |
|  | ftp2 | "ftp://qa.d3security.com" | string |
|  | ftp3 | 0 | int |
| ▶ | ftp4 | {1/1/0001 12:00:00 AM} | System.DateTime |

GetValue() – 如果转化失败，则使用默认值 string=null, int = 0, datetime=minvalue, 并不抛出错误

Server ftpObj0 = configRoot.GetSection("ftp").GetValue<Server>("server");-非叶子节点Value=null

Server ftpObj1 = configRoot.GetSection("ftp:server").Get<Server>();- 会自动递归绑定

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Name** | **Value** | **Type** |
|  | ftpObj0 | null | WebCore1.Server |
| ◢ | ftpObj1 | {WebCore1.Server} | WebCore1.Server |
|  | host | "ftp://qa.d3security.com" | string |
|  | ip | "192.168.1.200" | string |
|  | port | 4530 | int |

上面的例子，返回值可以知道 Get() 与 GetValue() 最重要的区别

# [多样性的配置来源](https://www.cnblogs.com/artech/p/asp-net-core-config-4-1.html)

可以在.Net Core 启动程序里添加配置文件源

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

WebHost.CreateDefaultBuilder(args)

.ConfigureAppConfiguration((hostingContext, config) =>

{

var env = hostingContext.HostingEnvironment;

config.SetBasePath(env.ContentRootPath)

.AddInMemoryCollection( new List<KeyValuePair<string, string>>

{

new KeyValuePair<string, string>("database.ssl", "true"),

new KeyValuePair<string, string>("database.port", "443"),

new KeyValuePair<string, string>("database.persit", "Permission"),

new KeyValuePair<string, string>("database.transport", "Stream")

})

.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>();

}

# 五、自定义数据源：

public abstract class ConfigurationProvider : IConfigurationProvider

{

protected ConfigurationProvider();

protected IDictionary<string, string> Data { get; set; }

public virtual IEnumerable<string> GetChildKeys(IEnumerable<string> earlierKeys, string parentPath);

public IChangeToken GetReloadToken();

public virtual void Load();

public virtual void Set(string key, string value);

public virtual bool TryGet(string key, out string value);

protected void OnReload();

}

public interface IConfigurationSource

{

IConfigurationProvider Build(IConfigurationBuilder builder);

}

// create data provider

public class MyProvider : ConfigurationProvider

{

public MyProvider() { }

public override void Load() // 加载数据

{

this.Data = new Dictionary<string, string>

{

{"Yahoo", "Yahoo.com" },

{"Google", "Google.com" },

{"Youtube", "Youtube.com" }

};

this.Set("MyKey", "www.mysite.com");

this.Set("HisKey", "www.yes.com");

base.Load();

}

}

// create data source

public class MySource: IConfigurationSource //通过 Build(IConfigurationBuilder builder)来连接数据提供者

{

public MyProvider myp { get; set; }

public MySource() { }

public MySource(MyProvider myp)

{

this.myp = myp;

}

IConfigurationProvider IConfigurationSource.Build(IConfigurationBuilder builder)

{

if (myp != null)

return myp;

else

return new MyProvider();

}

}

// Extend Configuration Builder and Add Data Source to config

public static class ConfigureBuilderExtension

{

public static IConfigurationBuilder **AddCustomSource**(this IConfigurationBuilder builder, MySource msrc)

{

return builder.Add(msrc);

}

}

// Main Program

MyProvider myp = new MyProvider();

myp.Set("yourkey", "www.key.com");

MySource mys = new MySource(myp);

IConfigurationBuilder builder = new ConfigurationBuilder();

builder.SetBasePath(AppDomain.CurrentDomain.BaseDirectory)

.AddInMemoryCollection(ms)

.**AddCustomSource**(mys)

.Add(mmsource)

.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 configRoot = builder.Build();

# 

# IConfigurationSource – 不可以混合其他不同类型的数据源

public class MySource: IConfigurationSource

{

public MyProvider myp { get; set; }

public MySource()

{

}

public MySource(MyProvider myp)

{

this.myp = myp;

}

IConfigurationProvider IConfigurationSource.Build(IConfigurationBuilder builder)

{

// 这段代码会导致抛出错误: 不可以使用 builder 来添加数据源

builder.AddInMemoryCollection(new List<KeyValuePair<string, string>> {

new KeyValuePair<string, string>("D3Web","www.d3security.com"),

new KeyValuePair<string, string>("Port", "3434")

});

//-----------

if (myp != null)

return myp;

else

return new MyProvider();

}

}

数据来源于数据库：

public class MyDB: DbContext

{

public MyDB() { }

public MyDB(DbContextOptions dbo):base(dbo) { }

public DbSet<Lang> Words { get; set; }

}

[Table("Language")]

public class Lang

{

[Key, DatabaseGenerated(DatabaseGeneratedOption.Identity)]

public int Id { get; set; }

public string Keyword { get; set; }

[Column("lang\_cn")]

public string Word { get; set; }

public byte Status { get; set; }

public byte Deleted { get; set; }

}

public class MyProvider : ConfigurationProvider

{

public Action<DbContextOptionsBuilder> DBBuilderAction { get; set; }

public MyProvider(Action<DbContextOptionsBuilder> dbBuildAction) {

this.DBBuilderAction = dbBuildAction;

}

public override void Load()

{

DbContextOptionsBuilder dbbuilder = new DbContextOptionsBuilder();

this.DBBuilderAction(dbbuilder); - 这里调用Action方法，传递DbContextOptionsBuilder对象，通过Action给对象赋值

MyDB = new MyDB(dbbuilder.Options);

this.Data = myDB.Words.Where(p => p.Status == 1 && p.Deleted == 0).ToDictionary(t => t.Keyword, t => t.Word);

base.Load();

}

}

public class MySource: IConfigurationSource

{

public Action<DbContextOptionsBuilder> DBBuilderAction { get; set; }

public MySource(Action<DbContextOptionsBuilder> dbBuildAction)

{

this.DBBuilderAction = dbBuildAction;

}

public IConfigurationProvider Build(IConfigurationBuilder builder)

{

return new MyProvider(this.DBBuilderAction); - 传递 Action

}

}

public static class ConfigureBuilderExtension

{

public static IConfigurationBuilder **AddCustomSource**(

this IConfigurationBuilder builder,

Action<DbContextOptionsBuilder> dbBuilderAction)

{

return builder.Add(new MySource(dbBuilderAction));

}

}

# 主应用程序：

IConfigurationBuilder builder = new ConfigurationBuilder();

builder.**AddCustomSource**(

p=>p.UseSqlServer("Server=xxx;Catalog=db;UserID=dbuser;Password=xx;")

);

IConfigurationRoot configRoot = builder.Build();

services.AddSingleton<IConfiguration>(configRoot);

# 程序要点：

# 主要通过 Action<DbContextOptionsBuilder> 来传递数据库连接信息

# 方法有很多种，这只是一种参考方法而已，关键是数据

protected IDictionary<string, string> Data { get; set; }

# 六、参考文章

<https://blog.csdn.net/n9nzjx57bf/article/details/72356669?utm_source=blogxgwz4>

<http://www.cnblogs.com/tianma3798/p/6970114.html>

注入整个 Iconfiguration

<https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration/?view=aspnetcore-2.1>

<https://www.jianshu.com/p/b9416867e6e6>

配置文件来自于数据库：

<https://www.cnblogs.com/artech/p/asp-net-core-config-4-3.html>

最全的 .Net Core 配置文件介绍：

<http://www.cnblogs.com/artech/p/asp-net-core-config-01.html>

配置文件来源于数据库： 正确的做法

<https://blog.csdn.net/sD7O95O/article/details/78096325>