.NET6中一些常用组件的配置及使用记录，持续更新中。。。 - VictorStar - 博客园

.NET6中一些常用组件的配置及使用记录，持续更新中。。。

NET6App

介绍

.NET 6的CoreApp框架，用来学习.NET6的一些变动和新特性，使用EFCore,等一系列组件的运用，每个用单独的文档篇章记录，持续更新文档哦，有什么想要了解的组件啊可以留言或私信我。

如果对您有帮助，点击⭐Star⭐关注 ，感谢支持开源！

软件架构

分为模型层，服务层，接口层来做测试使用

0.如何使用IConfiguration、Environment

直接在builder后的主机中使用。

```
builder.Configuration；
builder.Environment
```

1.如何使用Swagger

.NET 6 自带模板已经默认添加Swagger，直接使用即可。

```
builder.Services.AddSwaggerGen();

if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}
```

2. 如何添加EFCore到.NET 6中

按照EFCore常规使用方法，申明表的Entity及Dbcontext后，在program.cs文件中添加

```
builder.Services.AddDbContext<Service.DataContext>(opt => {
    opt.UseSqlServer(builder.Configuration.GetConnectionString("Default"));
});
```

即可在其他地方注入使用 DataContext

使用Sqlite数据库，需要引用 Microsoft.EntityFrameworkCore.Sqlite，
并在添加服务时，改为

```
opt.UseSqlite(builder.Configuration.GetConnectionString("Default"));
```

包管理控制台数据库结构生成方法：
使用 add-migration 创建迁移
使用 update-database 更新数据结构

3.如何注入一个服务

```
builder.Services.AddScoped<UserIdentyService>();
```

4.如何定义全局的using引用

在根目录下新建一个 cs文件，比如Globalusing.cs，在里面添加你的全局引用，和常规引用不同的是，在using前面添加 global

```
global using Service;
global using Entity;
global using Entity.Dto;
```

5.如何使用Autofac

添加 Nuget 引用

```
Autofac.Extensions.DependencyInjection
```

program.cs文件添加autofac的使用和注入配置

```
builder.Host.UseServiceProviderFactory(new AutofacServiceProviderFactory());
builder.Host.ConfigureContainer<ContainerBuilder>(builder =>
 {
     Assembly assembly = Assembly.Load("Service.dll");
     builder.RegisterAssemblyTypes(assembly)
            //.AsImplementedInterfaces()// 无接口的注入方式
            .InstancePerDependency();
 });
```

此时即可构造函数注入使用。

6.如何使用Log4Net

添加引用

```
Microsoft.Extensions.Logging.Log4Net.AspNetCore
```

新建配置文件 log4net.config；
添加service配置

```
//注入Log4Net
builder.Services.AddLogging(cfg =>
{
    //默认的配置文件路径是在根目录，且文件名为log4net.config
    //cfg.AddLog4Net();
    //如果文件路径或名称有变化，需要重新设置其路径或名称
    //比如在项目根目录下创建一个名为config的文件夹，将log4net.config文件移入其中，并改名为
log4net.config
    //则需要使用下面的代码来进行配置
    cfg.AddLog4Net(new Log4NetProviderOptions()
    {
        Log4NetConfigFileName = "config/log4net.config",
        Watch = true
    });
});
```

即可在需要的地方定义使用

```
_logger = LogManager.GetLogger(typeof(UserController));
```

7.如何使用全局异常过滤器

首先新建 GlobalExceptionFilter 全局异常过滤器，继承于 ExceptionFilter ，用于接收处理抛出的异常

```
public class GlobalExceptionFilter : IExceptionFilter
{
    readonly IWebHostEnvironment hostEnvironment;
    readonly ILog logger;
    public GlobalExceptionFilter(IWebHostEnvironment _hostEnvironment)
    {
        this.hostEnvironment = _hostEnvironment;
        this.logger = LogManager.GetLogger(typeof(GlobalExceptionFilter));
    }
    public void OnException(ExceptionContext context)
    {
        if (!context.ExceptionHandled)//如果异常没有处理
        {
            var result = new ApiResult
            {
                Code = 500,
                IsSuccess = false,
                Message = "服务器发生未处理的异常"
            };

            if (hostEnvironment.IsDevelopment())
            {
                result.Message += "," + context.Exception.Message;
                result.Data = context.Exception.StackTrace;
            }

            logger.Error(result);

            context.Result = new JsonResult(result);
            context.ExceptionHandled = true;//异常已处理
        }
    }
}
```

然后在Service中添加全局异常过滤器

```
builder.Services.AddControllers(option =>
    {
        option.Filters.Add<GlobalExceptionFilter>();
    }
);
```

添加控制器方法完成测试

```
[HttpGet("exception")]
public ApiResult ExceptionAction()
{
    throw new NotImplementedException();
}
```

8.如何使用redis做缓存

使用 StackExchange.Redis 作为缓存组件（其他组件类似的使用方式）。nuget 安装 StackExchange.Redis.Extensions.Core
首先，先建立一个类 RedisClient ，用于管理redis的连接和操作，再建立一个

RedisClientFactory 类，用于创建 redis的连接；

```
public class RedisClient{...}
public class RedisClientFactory{...}
```

appsettings.json 中添加redis的配置

```
"RedisConfig": {
    "Redis_Default": {
      "Connection": "127.0.0.1:6379",
      "InstanceName": "Redis1:"
    },
    "Redis_6": {
      "Connection": "127.0.0.1:6379",
      "DefaultDatabase": 6,
      "InstanceName": "Redis2:"
    }
  }
```

service中添加 redis客户端的引用

```
//添加redis的使用
builder.Services.AddSingleton<RedisClient>(_=>
RedisClientFactory.GetInstance(builder.Configuration));
```

一顿操作后，就可以在你想要使用redis的地方引用了

```
RedisClient redisClient
...
this.redisDb = redisClient.GetDatabase("Redis_Default");
redisDb.StringSet("clientId", "clientId", TimeSpan.FromSeconds(10));
```

要使用redis做分布式缓存，先引用 Microsoft.Extensions.Caching.StackExchangeRedis

```
//将Redis分布式缓存服务添加到服务中
builder.Services.AddStackExchangeRedisCache(options =>
    {
        //用于连接Redis的配置
Configuration.GetConnectionString("RedisConnectionString")读取配置信息的串
        options.Configuration = "Redis_6";//
Configuration.GetConnectionString("RedisConnectionString");
        //Redis实例名RedisDistributedCache
        options.InstanceName = "RedisDistributedCache";
    });
```

> 引用自 "分布式 Redis 缓存"

9. 如何添加使用定时任务组件

此处使用 Hangfire 定时任务组件，轻便，可持久化，还有面板。
引用 Hangfire 后，即可新增定时任务。

```
//启用Hangfire服务.
builder.Services.AddHangfire(x => x.UseStorage(new MemoryStorage()));
builder.Services.AddHangfireServer();

...

//启用Hangfire面板
app.UseHangfireDashboard();
//开启一个定时任务
RecurringJob.AddOrUpdate("test",() => Console.WriteLine("Recurring!"),
Cron.Minutely());
```

访问 https://localhost:7219/hangfire 即可看到任务面板

10. 如何使用业务锁锁住下单或者支付操作

首先，做这个事需要能先构建出一个锁出来，这个锁有个锁的标识key，可以根据这个key判定key对应的锁是否存在，
这样的话，在某个用户支付或者下单减库存啥的时候，就可以按照这个key先上锁，后面有用户走其他渠道进行同样的操作的时候，就可以根据是否上锁了，来判断操作能否继续。

比如一个支付订单的业务，可以在手机上操作，也可以在电脑上操作，这个时候就可以给支付接口上锁，只要一个支付过程存在着，并且没有超时，那就不能在其他渠道进行操作。
我们上面已经使用了redis，下面就用redis构建个锁来模拟这个操作,具体看代码：

```
/// <summary>
    /// 测试业务锁
    /// </summary>
    /// <returns></returns>
    [HttpGet("lockhandle")]
    public async Task<ApiResult> LockHandle(int userId)
    {
        var key = "user";
        var token = $"ID:{userId}";
        try
        {
            if (redisDb.LockTake(key, token, TimeSpan.FromSeconds(50)))
            {
                await Task.Delay(30 * 1000);
                return await Task.FromResult(ApiResult.Success($"ID:{userId} 获取到
锁了，操作正常,connectId:{Request.HttpContext.Connection.Id}"));
            }
            else
            {
                return await Task.FromResult(ApiResult.Fail($"有正在操作的
锁,connectId:{Request.HttpContext.Connection.Id}"));
            }
        }
        catch (Exception)
        {
            throw;
        }
        finally
        {
            redisDb.LockRelease(key, token);
        }
    }
```

11. 如何配置跨域

此处主要记录全局跨域，不包括指定api跨域。先增加一个配置 "Cors": "http:127.0.0.1:5001"，配置可以跨域的url，也可以使用默认跨域配置。host配置以下服务,按需使用：

```
builder.Services.AddCors(delegate (CorsOptions options)
{
    options.AddPolicy("CorsPolicy", delegate (CorsPolicyBuilder corsBuilder)
    {
        //指定url跨域
        corsBuilder.WithOrigins(builder.Configuration.GetValue<string>
("Cors").Split(','));
        //默认跨域
        corsBuilder.SetIsOriginAllowed((string _) =>
true).AllowAnyMethod().AllowAnyHeader()
            .AllowCredentials();
    });
});
```

12. 如何使用NewtonsoftJson

.NET6 默认的系列化库是内置的 System.Text.Json，使用中如果有诸多不熟悉的地方，那肯定是想换回 NewtonsoftJson，需要nuget 引用 Microsoft.AspNetCore.Mvc.NewtonsoftJson 来配置使用，
常用配置包括日期格式、大小写规则、循环引用配置。。。等，下面是一个配置

```
builder.Services.AddControllers(option =>
    {
        option.Filters.Add<GlobalExceptionFilter>();
    }
).AddNewtonsoftJson(options =>
{
    options.SerializerSettings.ContractResolver = new
CamelCasePropertyNamesContractResolver(); //序列化时key为驼峰样式
    options.SerializerSettings.DateTimeZoneHandling = DateTimeZoneHandling.Local;
    options.SerializerSettings.DateFormatString = "yyyy-MM-dd HH:mm:ss";
    options.SerializerSettings.ReferenceLoopHandling =
ReferenceLoopHandling.Ignore;//忽略循环引用
});
```

13. 如何使用SignalR

首先添加一个 ChatHub 作为 交互中心处理器

```
public class ChatHub : Hub
    {
        public async Task SendMessage(string user, string message)
        {
            await Clients.All.SendAsync("ReceiveMessage", user, message);
        }
    }
```

在主机中使用服务

```
builder.Services.AddSignalR();
...
app.UseEndpoints(endpoints =>
{
    endpoints.MapHub<ChatHub>("/chatHub");
});
```

14. 如何使用Dapper

Dapper是大家常用的一个数据库连接扩展组件，下面介绍下，如何使用常规扩展，来在.net Core中使用Dapper。
首先，建立一个DbComponent ，来获取由 .netCore 提供的 Configuration 配置文件，并用 DbProviderFactories 工厂，创建数据库连接，此类只管创建连接，又其他使用类进行销毁。

```
/// <summary>
/// 创建连接处理
/// </summary>
public class DbComponent
{
    /// 数据库连接配置
    private static ConnectionStringSettings connectionSetting;

    public static void InitDapper(ConnectionStringSettings
connectionStringSettings)
    {
        connectionSetting = connectionStringSettings;
    }

    //通过工厂模式创建Connection连接 此连接已打开
    public static IDbConnection GetConnection()
    {
        get {
            var cnnection =
DbProviderFactories.GetFactory(connectionSetting.ProviderName).CreateConnection();
            if (cnnection == null)
                throw new Exception("数据库链接获取失败!");
            cnnection.ConnectionString = connectionSetting.ConnectionString;
            cnnection.Open();
            return cnnection;
        }
    }
}
```

使用前，需要在program中初始化一下组件

```
/// <summary>
/// 初始化Dapper组件
/// </summary>
DbProviderFactories.RegisterFactory("Microsoft.Data.Sqlite",
Microsoft.Data.Sqlite.SqliteFactory.Instance);
DbComponent.InitDapper(new System.Configuration.ConnectionStringSettings
{
    ConnectionString = builder.Configuration.GetConnectionString("Default"),
    ProviderName = "Microsoft.Data.Sqlite"
});
```

程序启动后，就可以在需要的地方使用

```
public class UserIdentyService
{
    public ApiResult DapperList()
    {
        using (var connect = DbComponent.Connection)
        {
            var users= connect.Query<User>("SELECT * FROM Users").ToList();
            return ApiResult.Success(users);
        }
    }
}
```

15. 如何添加自定义配置文件

有时候我们不想把配置全部放在 appsettings.json ，我们想自己建立一个文件夹来存储其他配置文件，比如config/...json之类的，咋整呢，
我们新建个文件夹 config，下面建立一个配置文件app.json，里面存几个配置以便验证。

使用前添加如下代码即可

```
builder.Configuration.AddJsonFile("config/app.json");
Console.WriteLine(builder.Configuration.GetValue<string>("weixin"));
```

16. 如何简单上传文件

上传文件是每个api框架都会实现的功能，我们先实现一个简单的文件上传。
首先做个配置文件，存储上传的文件存储位置、大小及格式限制等的配置

```
public class UploadConfig
{
    /// <summary>
    /// 最大值
    /// </summary>
    public int MaxSize { get; set; } = 1024 * 1024 * 1024;
    /// <summary>
    /// 存储路径
    /// </summary>
    public string UploadDir { get; set; } = @"D://Upload";
    /// <summary>
    /// 站点名称
    /// </summary>
    public string WebSite { get; set; }
}
```

添加测试action,完成文件上传，并返回文件访问路径

```
/// <summary>
/// 上传文件测试
/// </summary>
/// <param name="files"></param>
/// <returns></returns>
[HttpPost("upload")]
public async Task<ApiResult> Upload([FromForm(Name ="file")] List<IFormFile>
files)
{
    var config = configuration.GetSection("UploadConfig").Get<UploadConfig>();
    if (files.Count == 0)
    {
        return ApiResult.Fail("没有需要上传的文件");
    }
    var dir = Path.Combine(AppDomain.CurrentDomain.BaseDirectory,
config.UploadDir);

    if (!Directory.Exists(dir)) Directory.CreateDirectory(dir);

    //验证大小或者格式之类
    foreach (var file in files)
    {
        var fileName =
ContentDispositionHeaderValue.Parse(file.ContentDisposition).FileName;
        var fileSize = file.Length;
        if (fileSize > config.MaxSize)
        {
            return ApiResult.Fail($"{fileName}文件过大");
        }
    }
    //存储文件
    var result = new List<string>();
    foreach (var file in files)
    {
        var fileName = file.FileName;
        using (var stream = System.IO.File.Create(Path.Combine(dir, fileName)))
        {
            await file.CopyToAsync(stream);
        }
        result.Add(string.Join('/',config.WebSite,"upload/view" fileName));
    }
    return ApiResult.Success(result);
}
```

上述文件访问路径需要配置静态目录来进行访问

```
//启动www静态目录
app.UseStaticFiles();
//启动上传文件目录
app.UseStaticFiles(new StaticFileOptions {
    FileProvider =  new
PhysicalFileProvider(builder.Configuration.GetValue<string>
("UploadConfig:UploadDir")),
    RequestPath = "/upload/view"
});
```

至此，文件上传及访问已添加完成

17. 如何添加验证码

验证码是常用的一个api功能，需要完成2步，先生成验证码字符串值及存储，后需要输出验证码图片到前端，即验证码功能需要2个api，1.获取验证码，2.验证验证码。
先实现一个类，用于获取随机值及生成图片
为了方便部署到unix系统，使用 ImageSharp 相关类库

```
public class CaptchaHelper
{
   private const string Letters =
"1,2,3,4,5,6,7,8,9,A,B,C,D,E,F,G,H,J,K,L,M,N,P,Q,R,S,T,U,V,W,X,Y,Z";

   /// <summary>
   /// 生成验证码随机值
   /// </summary>
   /// <param name="codeLength"></param>
   /// <returns></returns>
   public Task<string> GenerateRandomCaptchaAsync(int codeLength = 4)
   {
       var array = Letters.Split(new[] { ',' });
       var random = new Random();
       var temp = -1;
       var captcheCode = string.Empty;
       for (int i = 0; i < codeLength; i++)
       {
           if (temp != -1)
               random = new Random(i * temp * unchecked((int)DateTime.Now.Ticks));

           var index = random.Next(array.Length);
           if (temp != -1 && temp == index)
               return GenerateRandomCaptchaAsync(codeLength);

           temp = index;
           captcheCode += array[index];
       }
       return Task.FromResult(captcheCode);
   }

   /// <summary>
   /// 生成验证码及图片
   /// </summary>
   /// <param name="captchaCode"></param>
   /// <param name="width"></param>
   /// <param name="height"></param>
   /// <returns></returns>
   public Task<(string code, MemoryStream ms)> GenerateCaptchaImageAsync(string
captchaCode, int width = 0, int height = 30)
   {
       //验证码颜色集合
       Color[] colors = { Color.Black, Color.Red, Color.DarkBlue, Color.Green,
Color.Orange, Color.Brown, Color.DarkCyan, Color.Purple };
       //验证码字体集合
       string[] fonts = { "Verdana", "Microsoft Sans Serif", "Comic Sans MS",
"Arial" };
       var r = new Random();

       if(width == 0) { width = captchaCode.Length * 25; }
       //定义图像的大小，生成图像的实例
       using var image = new Image<Rgba32>(width, height);

       // 字体
       var font = SystemFonts.CreateFont(SystemFonts.Families.First().Name, 25,
FontStyle.Bold);

       image.Mutate(ctx =>
       {
```

```
            // 白底背景
            ctx.Fill(Color.White);

            // 画验证码
            for (int i = 0; i < captchaCode.Length; i++)
            {
                ctx.DrawText(captchaCode[i].ToString()
                    , font
                    , colors[r.Next(colors.Length)]
                    , new PointF(20 * i + 10, r.Next(2, 12)));
            }

            // 画干扰线
            for (int i = 0; i < 10; i++)
            {
                var pen = new Pen(colors[r.Next(colors.Length)], 1);
                var p1 = new PointF(r.Next(width), r.Next(height));
                var p2 = new PointF(r.Next(width), r.Next(height));

                ctx.DrawLines(pen, p1, p2);
            }

            // 画噪点
            for (int i = 0; i < 80; i++)
            {
                var pen = new Pen(colors[r.Next(colors.Length)], 1);
                var p1 = new PointF(r.Next(width), r.Next(height));
                var p2 = new PointF(p1.X + 1f, p1.Y + 1f);

                ctx.DrawLines(pen, p1, p2);
            }
        });
        using var ms = new MemoryStream();

        // gif 格式
        image.SaveAsGif(ms);
        return Task.FromResult((captchaCode, ms));
    }
}
```

测试代码

```
/// <summary>
/// 测试验证码
/// </summary>
/// <returns></returns>
[HttpGet("captcha")]
public async Task<IActionResult> GetCaptcha()
{
    var captchaHelper = new CaptchaHelper();
    var captcha = await captchaHelper.GenerateCaptchaImageAsync();
    this.HttpContext.Session.SetString("captcha", captcha.code);
    return File(captcha.ms.ToArray(), "image/gif");
}
```

18. 如何发布到windows

发布到windows比较简单，一步步选择发布，只要注意是按照框架依赖还是独立发布就好了，框架依赖的意思是， 你的服务器已经安装好了个.NetCore的运行时，你就可以直接框架依赖来发布就好了，如果你的服务器没有安装运行时， 或者是需要兼容以前的老的运行时不能更新啥的，你就使用独立发布。

19. 如何发布到linux

发布到linux其实也比较简单，首先，需要有待发布的程序文件，和windows一样，做好框架依赖和独立发布的安装包，目标运行时记得选择linux.64 或者linux-arm，编译发布，获得程序文件。
服务器运行时，和windows一样，框架依赖需要安装下.NET Core的运行时，注意匹配版本号，运行时安装好之后，把程序通过xshell 或者 scp压缩包上传到linux服务器你想要的目录下面，完成程序文件部署，在运行时版本没问题的情况下，dotnet xxx.dll 即可启动站点，还有一些其他的配置可能需要注意，比如后台运行，比如开放防火墙端口之类的，照着整就可以了。

在时间轴上留下点什么，作为个程序员，就在cnblog上吧，我不是技术大牛，也不是IT狂热者，说不上是我选了它还是它选了我，结果是现在我是一名程序员，我对它很感兴趣，不为以后做业务大佬、行业顶尖，只想在自己能力范围内，把事情做到最好，在博客园记录下自己的历程吧，工作，学习，感情，生活。

分类: C#, .NET CORE
标签: C#, .NET Core
« 上一篇： 容器扩展属性 IExtenderProvider 实现WinForm通用数据验证组件