链滴

NetApiStarter- 开始你的 api

作者：loogn

原文链接：https://ld246.com/article/1575100135063

来源网站：链滴

在此之前，写过一篇 给新手的WebAPI实践 ，获得了很多新人的认可，那时还是基于.net mvc，文生成还是自己闹洞大开写出来的，经过这两年的时间，netcore的发展已经势不可挡，自己也在不断学习，公司的项目也转向了netcore。大部分也都是前后分离的架构，后端api开发居多，从中整理了些东西在这里分享给大家。

源码地址：https://gitee.com/loogn/NetApiStarter，这是一个基于netcore mvc 3.0的模板项目，果你使用的netcore 2.x，除了引用不通用外，代码基本是可以复用的。下面介绍一下其中的功能。

登录验证

这里我默认使用了jwt登录验证，因为它足够简单和轻量，在netcore mvc中使用jwt验证非常简单，先在startup.cs文件中配置服务并启用：

ConfigureServices方法中：

```
            var jwtSection = Configuration.GetSection("Jwt");
            services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
                .AddJwtBearer(options =>
                {
                    options.TokenValidationParameters = new TokenValidationParameters
                    {
                        ValidateIssuer = true,
                        ValidateAudience = true,
                        ValidateLifetime = true,
                        ValidateIssuerSigningKey = true,
                        ValidAudience = jwtSection["Audience"],
                        ValidIssuer = jwtSection["Issuer"],
                        IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(jwtSec
ion["SigningKey"]))
                    };
                });
```

Configure方法中，在UseRouting和UseEndpoints方法之前：

```
app.UseAuthorization();
```

上面我们使用到了jwt配置块，对应appsettings.json文件中有这样的配置：

```
{
  "Jwt": {
    "SigningKey": "1234567812345678",
    "Issuer": "NetApiStarter",
    "Audience": "NetApiStarter"
  }
}
```

我们再操作两步来实现登录验证，

一、提供一个接口生成jwt，

二、在客户端请求头部加上Authorization: Bearer {jwt}

我先封装了一个生成jwt的方法

```
    public static class JwtHelper
    {
```

```
        public static string WriteToken(Dictionary<string, string> claimDict, DateTime exp)
        {
            var key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(AppSettings.Instance.Jw
.SigningKey));
            var creds = new SigningCredentials(key, SecurityAlgorithms.HmacSha256);
            var token = new JwtSecurityToken(
                issuer: AppSettings.Instance.Jwt.Issuer,
                audience: AppSettings.Instance.Jwt.Audience,
                claims: claimDict.Select(x => new Claim(x.Key, x.Value)),
                expires: exp,
                signingCredentials: creds);
            var jwt = new JwtSecurityTokenHandler().WriteToken(token);
            return jwt;
        }
}
```

然后在登录服务中调用

```
        /// <summary>
        /// 登录，获取jwt
        /// </summary>
        /// <param name="request"></param>
        /// <returns></returns>
        public ResultObject<LoginResponse> Login(LoginRequest request)
        {
            var user = userDao.GetUser(request.Account, request.Password);
            if (user == null)
            {
                return new ResultObject<LoginResponse>("用户名或密码错误");
            }
            var dict = new Dictionary<string, string>();
            dict.Add("userid", user.Id.ToString());
            var jwt = JwtHelper.WriteToken(dict, DateTime.Now.AddDays(7));
            var response = new LoginResponse { Jwt = jwt };
            return new ResultObject<LoginResponse>(response);
        }
```

在Controller和Action上添加[Authorize]和[AllowAnonymous]两个特性就可以实现登录验证了。

请求响应

这里请求响应的设计依然没有使用restful风格，一是感觉太麻烦，二是真的不太懂(实事求是)，所以求还是以POST方式投递JSON数据，响应当然也是JSON数据这个没啥异议的。

为啥使用POST+JSON呢，主要是简单，大家都懂，而且规则统一、繁简皆宜，比如什么参数都不需，就传{}，根据ID查询文章{articleId:23}，或者复杂的查询条件和表单提交{ name:'abc', addr:{provic:'HeNan', city:'ZhengZhou'},tags:['骑马','射箭'] } 等等都可以优雅的传递。

这只是我个人的风格，netcore mvc是支持其他的方式的，选自己喜欢的就行了。

下面的内容还是按照POST+JSON来说。

首先提供请求基类：

```
    /// <summary>
```

```
    /// 登录用户请求的基类
    /// </summary>
    public class LoginedRequest
    {
        #region jwt相关用户
        private ClaimsPrincipal _claimsPrincipal { get; set; }

        public ClaimsPrincipal GetPrincipal()
        {
            return _claimsPrincipal;
        }
        public void SetPrincipal(ClaimsPrincipal user)
        {
            _claimsPrincipal = user;
        }

        public string GetClaimValue(string name)
        {
            return _claimsPrincipal?.FindFirst(name)?.Value;
        }
        #endregion

        #region 数据库相关用户 （如果有必要的话）
        //不用属性是因为swagger中会显示出来
        private User _user;
        public User GetUser()
        {
            return _user;
        }
        public void SetUser(User user)
        {
            _user = user;
        }
        #endregion
    }
```

这个类中说白了就是两个手写属性，一个ClaimsPrincipal用来保存从jwt解析出来的用户，一个User来保存数据库中完整的用户信息，为啥不直接使用属性呢，上面注释也提到了，不想在api文档中显出来。这个用户信息是在服务层使用的，而且User不是必须的，比如jwt中的信息够服务层使用，不义User也是可以的，总之这里的信息是为服务层逻辑服务的。

我们还可以定义其他的基类，比如经常用的分页基类：

```
    public class PagedRequest : LoginedRequest
    {
        public int PageIndex { get; set; }
        public int PageSize { get; set; }
    }
```

根据项目的实际情况还可以定义更多的基类来方便开发。

响应类使用统一的格式，这里直接提供json方便查看：

```
{
  "result": {
```

```
    "jwt": "string"
  },
  "success": true,
  "code": 0,
  "msg": "错误信息"
}
```

result是具体的响应对象，如果success为false的话，result一般是null。

ActionFilter

mvc本身是一个扩展性极强的框架，层层有拦截，ActionFilter就是其中之一，IActionFilter接口有两方法，一个是OnActionExecuted，一个是OnActionExecuting，从命名也能看出，就是在Action的后分别执行的方法。我们这里主要重写OnActionExecuting方法来做两件事：

一、将登陆信息赋值给请求对象

二、验证请求对象

这里说的请求对象，其类型就是LoginedRequest或者LoginedRequest的子类，看代码：

```
    [AppService]
    [AttributeUsage(AttributeTargets.Class | AttributeTargets.Method, AllowMultiple = false, In
erited = true)]
    public class MyActionFilterAttribute : ActionFilterAttribute
    {
        /// <summary>
        /// 是否验证参数有效性
        /// </summary>
        public bool ValidParams { get; set; } = true;

        public override void OnActionExecuting(ActionExecutingContext context)
        {
            //由于Filters是套娃模式，使用以下逻辑保证作用域的覆盖 Action > Controller > Global
            if (context.Filters.OfType<MyActionFilterAttribute>().Last() != this)
            {
                return;
            }

            //默认只有一个参数
            var firstParam = context.ActionArguments.FirstOrDefault().Value;
            if (firstParam != null && firstParam.GetType().IsClass)
            {
                //验证参数合法性
                if (ValidParams)
                {
                    var validationResults = new List<ValidationResult>();
                    var validationFlag = Validator.TryValidateObject(firstParam, new ValidationContex
(firstParam), validationResults, false);
                    if (!validationFlag)
                    {
                        var ro = new ResultObject(validationResults.First().ErrorMessage);
                        context.Result = new JsonResult(ro);
                        return;
```

```
                }
            }
        }

        var requestParams = firstParam as LoginedRequest;
        if (requestParams != null)
        {
            //设置jwt用户
            requestParams.SetPrincipal(context.HttpContext.User);
            var userid = requestParams.GetClaimValue("userid");
            //如果有必要，可以每次都获取数据库中的用户
            if (!string.IsNullOrEmpty(userid))
            {
                var user = ((UserService)context.HttpContext.RequestServices.GetService(typeof(
serService))).SingleById(long.Parse(userid));
                requestParams.SetUser(user);
            }
        }

        base.OnActionExecuting(context);
    }
}
```

模型验证这块使用的是系统自带的，从上面代码也可以看出，如果请求对象定义为LoginedRequest其子类，每次请求会填充ClaimsPrincipal，如果有必要，可以从数据库中读取User信息填充。

请求经过ActionFilter时，模型验证不通过的，直接返回了验证错误信息，通过之后到达Action和Service时，用户信息已经可以直接使用了。

api文档和日志

api文档首选swagger了，aspnetcore 官方文档也是使用的这个，我这里用的是Swashbuckle，首先装引用

```
Install-Package Swashbuckle.AspNetCore -Version 5.0.0-rc4
```

定义一个扩展类，方便把swagger注入容器中：

```
public static class SwaggerServiceExtensions
{
    public static IServiceCollection AddSwagger(this IServiceCollection services)
    {
        //https://github.com/domaindrivendev/Swashbuckle.AspNetCore
        services.AddSwaggerGen(c =>
        {
            c.SwaggerDoc("v1", new OpenApiInfo
            {
                Title = "My Api",
                Version = "v1"
            });
            c.IgnoreObsoleteActions();
            c.IgnoreObsoleteProperties();
            c.DocumentFilter<SwaggerDocumentFilter>();
            //自定义类型映射
            c.MapType<byte>(() => new OpenApiSchema { Type = "byte", Example = new Ope
```

```
ApiByte(0) });
            c.MapType<long>(() => new OpenApiSchema { Type = "long", Example = new Ope
ApiLong(0L) });
            c.MapType<int>(() => new OpenApiSchema { Type = "integer", Example = new Op
nApiInteger(0) });
            c.MapType<DateTime>(() => new OpenApiSchema { Type = "DateTime", Example =
new OpenApiDateTime(DateTimeOffset.Now) });

            //xml注释
            foreach (var file in Directory.GetFiles(AppContext.BaseDirectory, "*.xml"))
            {
                c.IncludeXmlComments(file);
            }

            //Authorization的设置
            c.AddSecurityDefinition("Bearer", new OpenApiSecurityScheme
            {
                In = ParameterLocation.Header,
                Description = "请输入验证的jwt。示例：Bearer {jwt}",
                Name = "Authorization",

                Type = SecuritySchemeType.ApiKey,
            });
        });
        return services;
    }

    /// <summary>
    /// Swagger控制器描述文字
    /// </summary>
    class SwaggerDocumentFilter : IDocumentFilter
    {
        public void Apply(OpenApiDocument swaggerDoc, DocumentFilterContext context)
        {
            swaggerDoc.Tags = new List<OpenApiTag>
            {
                new OpenApiTag{ Name="User", Description="用户相关"},
                new OpenApiTag{ Name="Common", Description="公共功能"},
            };
        }
    }
}
```

主要是验证部分，加上去之后就可以在文档中使用jwt测试了

然后在startup.cs的ConfigureServices方法中

```
services.AddSwagger();
```

Configure方法中：

```
        if (env.IsDevelopment())
        {
            app.UseSwagger();
            app.UseSwaggerUI(options =>
            {
```

```
                options.SwaggerEndpoint("/swagger/v1/swagger.json", "My API V1");
                options.DocExpansion(DocExpansion.None);
            });
        }
```

这里限制了只有在开发环境才显示api文档，如果是需要外部调用的话，可以不做这个限制。

日志组件使用Serilog。

首先也是安装引用

```
Install-Package Serilog
Install-Package Serilog.AspNetCore
Install-Package Serilog.Settings.Configuration
Install-Package Serilog.Sinks.RollingFile
```

然后在appsettings.json中添加配置

```
{
  "Serilog": {
    "WriteTo": [
      { "Name": "Console" },
      {
        "Name": "RollingFile",
        "Args": { "pathFormat": "logs/{Date}.log" }
      }
    ],
    "Enrich": [ "FromLogContext" ],
    "MinimumLevel": {
      "Default": "Debug",
      "Override": {
        "Microsoft": "Warning",
        "System": "Warning"
      }
    }
  },
}
```

更多配置请查看https://github.com/serilog/serilog-settings-configuration

上述配置会在应用程序根目录的logs文件夹下，每天生成一个命名类似20191129.log的日志文件

最后要修改一下Program.cs，代替默认的日志组件

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseConfiguration(new ConfigurationBuilder().SetBasePath(Environme
t.CurrentDirectory).AddJsonFile("appsettings.json").Build());

                webBuilder.UseStartup<Startup>();

                webBuilder.UseSerilog((whbContext, configureLogger) =>
```

```
                {
configureLogger.ReadFrom.Configuration(whbContext.Configuration);
                });
            });
```

文件分块上传

文件上传就像登录验证一样常用，哪个应用还不上传个头像啥的，所以我也打算整合到模板项目中，果是单纯的上传也就没必要说了，这里主要说的是一种大文件上传的解决方法： 分块上传。

分块上传是需要客户端配合的，客户端把一个大文件分好块，一小块一小块的上传，上传完成之后服端按照顺序合并到一起就是整个文件了。

所以我们先定义分块上传的参数：

```
string identifier : 文件标识，一个文件的唯一标识，
int chunkNumber ： 当前块所以，我是从1开始的
int chunkSize ： 每块大小，客户端设置的固定值，单位为byte，一般2M左右就可以了
long totalSize：文件总大小，单位为byte
int totalChunks：总块数
```

这些参数都好理解，在服务端验证和合并文件时需要。

开始的时候我是这样处理的，客户端每上传一块，我会把这块的内容写到一个临时文件中，使用identifier和chunkNumber来命名，这样就知道是哪个文件的哪一块了，当上传完最后一块之后，也就是chunkNumber==totalChunks的时候，我将所有的分块小文件合并到目标文件，然后返回url。

这个逻辑是没什么问题，只需要一个机制保证合并文件的时候所有块都已上传就可以了，为什么要这一个机制呢，主要是因为客户端的上传可能是多线程的，而且也不能完全保证http的响应顺序和请求序是一样的，所以虽然上传完最后一块才会合并，但是还是需要一个机制判断一下是否所有块都上传毕，没有上传完还要等待一下（想一想怎么实现！）。

后来在实际上传过程中发现最后一块响应会比较慢，特别是文件很大的时候，这个也好理解，因为最一块上传会合并文件，所以需要优化一下。

这里就使用到了队列的概念了，我们可以把每次上传的内容都放在队列中，然后使用另一个线程从队中读取并写入目标文件。在这个场景中BlockingCollection是最合适不过的了。

我们定义一个实体类，用于保存入列的数据：

```
public class UploadChunkItem
{
    public byte[] Data { get; set; }
    public int ChunkNumber { get; set; }
    public int ChunkSize { get; set; }
    public string FilePath { get; set; }
}
```

然后定义一个队列写入器

```
public class UploadChunkWriter
{
    public static UploadChunkWriter Instance = new UploadChunkWriter();
    private BlockingCollection<UploadChunkItem> _queue;
```

```
        private int _writeWorkerCount = 3;
        private Thread _writeThread;
        public UploadChunkWriter()
        {
            _queue = new BlockingCollection<UploadChunkItem>(500);
            _writeThread = new Thread(this.Write);
        }

        public void Write()
        {
            while (true)
            {
                //单线程写入
                //var item = _queue.Take();
                //using (var fileStream = File.Open(item.FilePath, FileMode.Open, FileAccess.Write, F
leShare.ReadWrite))
                //{
                //    fileStream.Position = (item.ChunkNumber - 1) * item.ChunkSize;
                //    fileStream.Write(item.Data, 0, item.Data.Length);
                //    item.Data = null;
                //}

                //多线程写入
                Task[] tasks = new Task[_writeWorkerCount];
                for (int i = 0; i < _writeWorkerCount; i++)
                {
                    var item = _queue.Take();
                    tasks[i] = Task.Run(() =>
                    {
                        using (var fileStream = File.Open(item.FilePath, FileMode.Open, FileAccess.Wri
e, FileShare.ReadWrite))
                        {
                            fileStream.Position = (item.ChunkNumber - 1) * item.ChunkSize;
                            fileStream.Write(item.Data, 0, item.Data.Length);
                            item.Data = null;
                        }
                    });
                }
                Task.WaitAll(tasks);
            }
        }

        public void Add(UploadChunkItem item)
        {
            _queue.Add(item);
        }

        public void Start()
        {
            _writeThread.Start();
        }
    }
```

主要是Write方法的逻辑，调用_queue.Take()方法从队列中获取一项，如果队列中没有数据，这个方

会堵塞当前线程，这也是我们所期望的，获取到数据之后，打开目标文件（在上传第一块的时候会创），根据ChunkNumber 和ChunkSize找到开始写入的位置，然后把本块数据写入。

打开目标文件的时候使用了FileShare.ReadWrite，表示这个文件可以同时被多个线程读取和写入。

文件上传方法也简单：

```
        /// <summary>
        /// 分片上传
        /// </summary>
        /// <param name="formFile"></param>
        /// <param name="chunkNumber"></param>
        /// <param name="chunkSize"></param>
        /// <param name="totalSize"></param>
        /// <param name="identifier"></param>
        /// <param name="totalChunks"></param>
        /// <returns></returns>
        public ResultObject<UploadFileResponse> ChunkUploadfile(IFormFile formFile, int chun
kNumber, int chunkSize, long totalSize,
            string identifier, int totalChunks)
        {
            var appSetting = AppSettings.Instance;
            #region 验证
            if (formFile == null && formFile.Length == 0)
            {
                return new ResultObject<UploadFileResponse>("文件不能为空");
            }
            if (formFile.Length > appSetting.Upload.LimitSize)
            {
                return new ResultObject<UploadFileResponse>("文件超过了最大限制");
            }
            var ext = Path.GetExtension(formFile.FileName).ToLower();
            if (!appSetting.Upload.AllowExts.Contains(ext))
            {
                return new ResultObject<UploadFileResponse>("文件类型不允许");
            }
            if (chunkNumber == 0 || chunkSize == 0 || totalSize == 0 || identifier.Length == 0 || to
alChunks == 0)
            {
                return new ResultObject<UploadFileResponse>("参数错误0");
            }
            if (chunkNumber > totalChunks)
            {
                return new ResultObject<UploadFileResponse>("参数错误1");
            }
            if (totalSize > appSetting.Upload.TotalLimitSize)
            {
                return new ResultObject<UploadFileResponse>("参数错误2");
            }
            if (chunkNumber < totalChunks && formFile.Length != chunkSize)
            {
                return new ResultObject<UploadFileResponse>("参数错误3");
            }
            if (totalChunks == 1 && formFile.Length != totalSize)
            {
```

```
            return new ResultObject<UploadFileResponse>("参数错误4");
        }
        #endregion

        //写入逻辑
        var now = DateTime.Now;
        var yy = now.ToString("yyyy");
        var mm = now.ToString("MM");
        var dd = now.ToString("dd");

        var fileName = EncryptHelper.MD5Encrypt(identifier) + ext;
        var folder = Path.Combine(appSetting.Upload.UploadPath, yy, mm, dd);
        var filePath = Path.Combine(folder, fileName);

        //线程安全的创建文件
        if (!File.Exists(filePath))
        {
            lock (lockObj)
            {
                if (!File.Exists(filePath))
                {
                    if (!Directory.Exists(folder))
                    {
                        Directory.CreateDirectory(folder);
                    }
                    File.Create(filePath).Dispose();
                }
            }
        }

        var data = new byte[formFile.Length];
        formFile.OpenReadStream().Read(data, 0, data.Length);

        UploadChunkWriter.Instance.Add(new UploadChunkItem
        {
            ChunkNumber = chunkNumber,
            ChunkSize = chunkSize,
            Data = data,
            FilePath = filePath
        });
        if (chunkNumber == totalChunks)
        {
            //等等写入完成
            int i = 0;
            while (true)
            {
                if (i >= 20)
                {
                    return new ResultObject<UploadFileResponse>
                    {
                        Success = false,
                        Msg = $"上传失败，总大小：{totalSize},实际大小：{new FileInfo(filePath).Leng
h}",
                        Result = new UploadFileResponse { Url = "" }
```

```
                };
            }
            if (new FileInfo(filePath).Length != totalSize)
            {
                Thread.Sleep(TimeSpan.FromMilliseconds(1000));
                i++;
            }
            else
            {
                break;
            }
        }
        var fileUrl = $"{appSetting.RootUrl}{appSetting.Upload.RequestPath}/{yy}/{mm}/{dd}
{fileName}";
        var response = new UploadFileResponse { Url = fileUrl };
        return new ResultObject<UploadFileResponse>(response);
    }
    else
    {
        return new ResultObject<UploadFileResponse>
        {
            Success = true,
            Msg = "uploading...",
            Result = new UploadFileResponse { Url = "" }
        };
    }
}
```

撇开上面的参数验证，主要逻辑也就是三个，一是创建目标文件，二是分块数据加入队列，三是最后块的时候要验证文件的完整性（也就是所有的块都上传了，并都写入到了目标文件）

创建目标文件需要保证线程安全，这里使用了双重检查加锁机制，双重检查的优点是避免了不必要的锁情况。

完整性我只是验证了文件的大小，这只是一种简单的机制，一般是够用了，别忘了我们的接口都是受jt保护的，包括这里的上传文件。如果要求更高的话，可以让客户端传参整个文件的md5值，然后服端验证合并之后文件的md5是否和客户端给的一致。

最后要开启写入线程，可以在Startup.cs的Configure方法中开启：

```
UploadChunkWriter.Instance.Start();
```

经过这样的整改，上传速度溜溜的，最后一块也不用长时间等待啦！

（项目中当然也包含了不分块上传）

其他功能

自从netcore提供了依赖注入，我也习惯了这种写法，不过在构造函数中写一堆注入实在是难看，而既要声明字段接收，又要写参数赋值，挺麻烦的，于是乎自己写了个小组件，已经用于手头所有的项，当然也包含在了NetApiStarter中，不仅解决了属性和字段注入，同时也解决了实现多接口注入的题，以及一个接口多个实现精准注入的问题，详细说明可查看项目文档Autowired.Core。

如果你听过MediatR，那么这个功能不需要介绍了，项目中包含一个应用程序级别的事件发布和订阅

功能，具体使用可查看文档AppEventService。

如果你听过AutoMapper，那么这个功能也不需要介绍了，项目中包含一个SimpleMapper，代码不功能还行，支持嵌套类、数组、IList<>、IDictionary<,>实体映射在多层数据传输的时候可谓是必不少的功能，用法嘛就不说了，只有一个Map方法太简单了

重中之重

如果你感觉这个项目对你、或者其他人（You or others，没毛病）有稍许帮助，请给个Star好吗！

NetApiStarter仓库地址：https://gitee.com/loogn/NetApiStarter