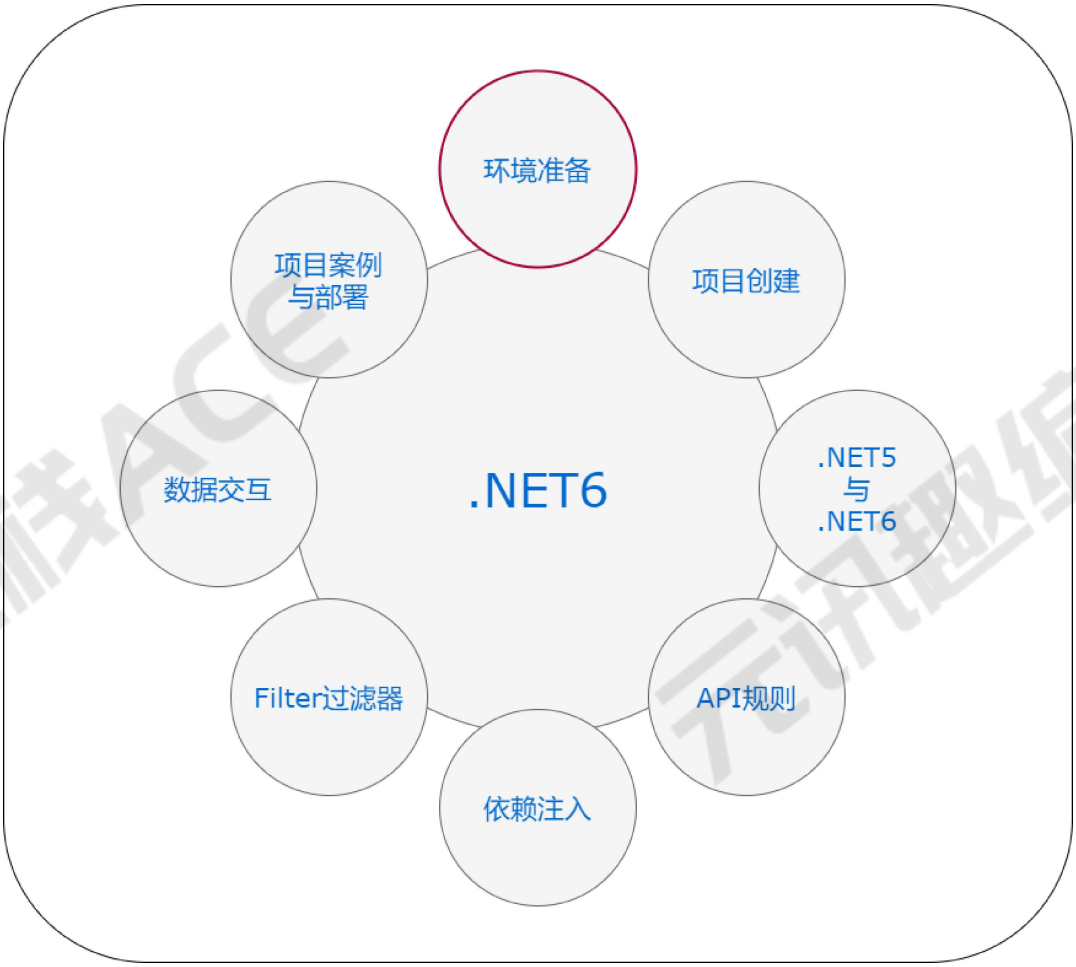
.NET6 专题课程知识点梳理

通过该专题我们将学习.NET6的如下知识点



什么是.NET6

.NET6是2021年11月8日发布的最新的也是最快的.NET框架。它具有跨平台及优秀的运行速度。C#/.NET可以说是目前市面上几乎唯一可以做任何事情的平台。

IEEE Spectrum 2021 编程语言 Top 10 如下：

Rank	Language	Type	Score
1	Python		100.0
2	Java		95.4
3	C		94.7
4	C++		92.4
5	JavaScript		88.1
6	C#		82.4
7	R		81.7
8	Go		77.7
9	HTML		75.4

如何配置.NET6的运行环境

方案一

直接安装Visual Studio 2022 : https://visualstudio.microsoft.com/zh-hans/vs/

这里注意 **社区版** 是免费的，而 **企业版** 和 **专业版** 是付费的

授权后：

方案二

下载.NET6的SDK文件：https://dotnet.microsoft.com/en-us/download/dotnet/6.0

用此种方案可以结合 **vs code** 进行项目开发

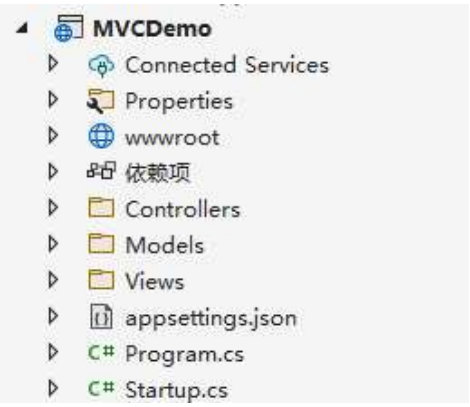
Visual Studio Code 与 Visual Studio 区别

Visual Studio 是一个功能齐全的 IDE，而 Visual Studio Code（以下简称VsCode）是一个代码编辑器。Visual Studio Code 是免费的，它基于开放源代码，在 Windows、macOS 和 Linux 上运行。

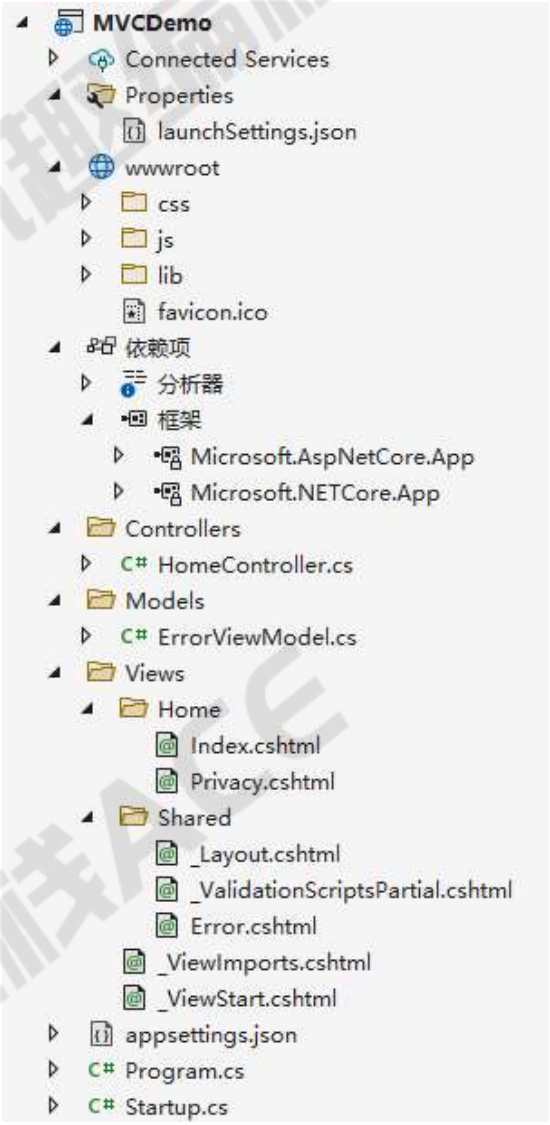
创建项目

用Visual Studio 和 VsCode 分别创建

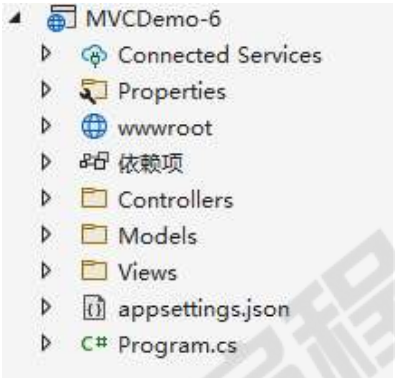
NET5 MVC 的项目结构



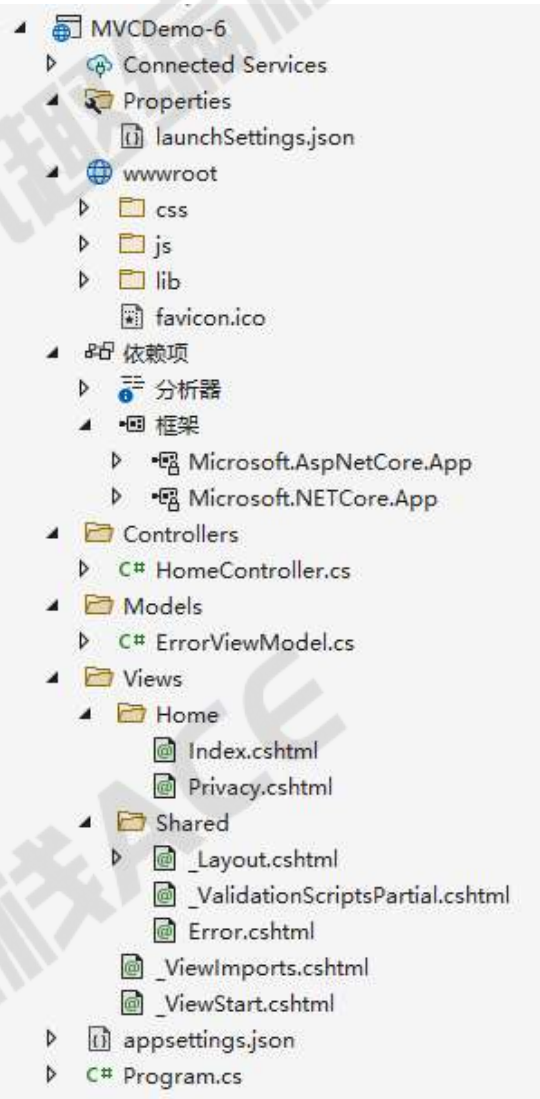
展开查看



.NET6 MVC 的项目



展开结构



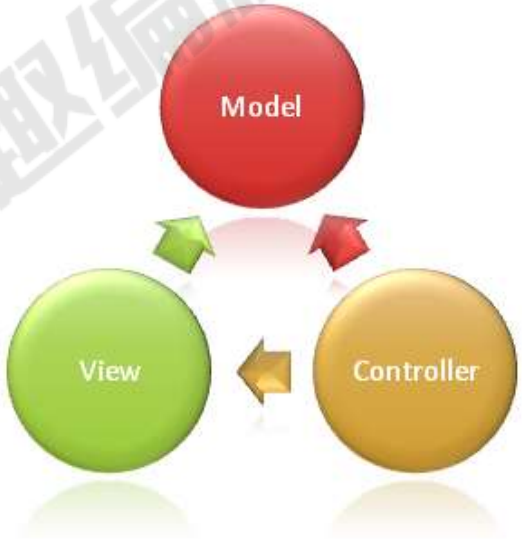
> 从表层看来 5 和 6 的最大区别就在 `Program.cs` 和 `Startup.cs` 上，这块我们未来会详细探讨，现在我们先暂且放过他~

MVC？ Web API?

MVC

模型-视图-控制器 (MVC) 体系结构模式将应用程序分成 3 个主要组件组：模型、视图和控制器。 此模式有助于实现 **关注点分离**。 使用此模式，用户请求被路由到控制器，后者负责使用模型来执行用户操作和/或检索查询结果。 控制器选择要显示给用户的视图，并为其提供所需的任何模型数据。

下图显示 3 个主要组件及其相互引用关系：



这种责任划分有助于根据复杂性缩放应用程序，因为这更易于编码、调试和测试包含单一作业的某个组成部分（模型、视图或控制器）。 但这会加大更新、测试和调试代码的难度，该代码在这 3 个领域的两个或多个领域间存在依赖关系。 例如，用户界面逻辑的变更频率往往高于业务逻辑。 如果将表示代码和业务逻辑组合在单个对象中，则每次更改用户界面时都必须修改包含业务逻辑的对象。 这常常会引发错误，并且需要在每次进行细微的用户界面更改后重新测试业务逻辑。

> 备注
>
> 视图和控制器均依赖于模型。 但是，模型既不依赖于视图，也不依赖于控制器。 这是分离的一个关键优势。 这种分离允许模型独立于可视化展示进行构建和测试。

Web API

ASP.NET Core 支持使用 C# 创建 RESTful 服务，也称为 Web API。 若要处理请求，Web API 使用控制器。 Web API 中的*控制器*是派生自 的类。

我们来看一个网址理解WebAPI：https://api.apihubs.cn/holiday/get?date=20220218

思考

MVC 和 Web API 有何异同?

Web API 入门

路由规则

`Route` 属性用于配置 Web API 路由

神奇的 ApiControllerAttribute

曾几何时，我们在使用Web API 时，常常需要给方法的参数添加特性

特性	绑定源
[FromBody]	请求正文
[FromForm]	请求正文中的表单数据
[FromHeader]	请求标头
[FromQuery]	请求查询字符串参数
[FromRoute]	当前请求中的路由数据
[FromServices]	作为操作参数插入的请求服务

但是自从有了这个属性，不要啦，统统不要啦！这个属性就是用于修饰的控制器配置有功能和行为，以改进开发人员生成 API 的体验为目标

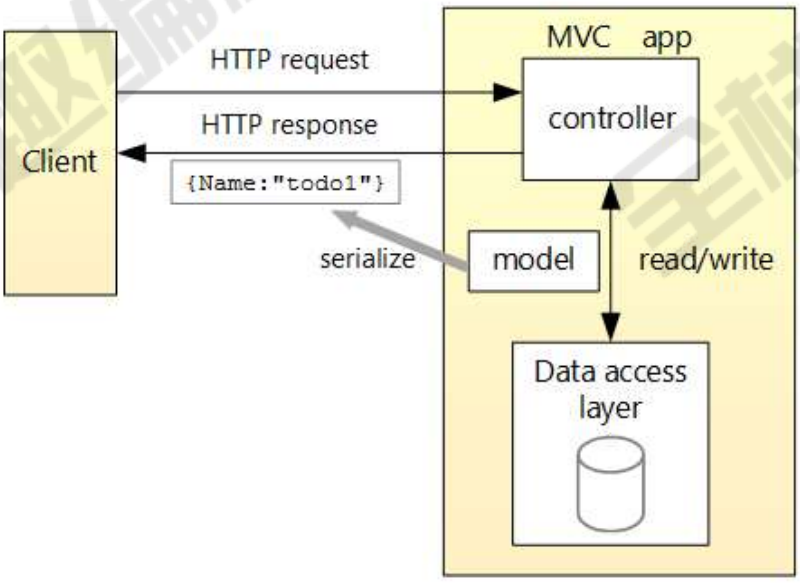
[ApiController] 属性可应用于控制器类，以启用下述 API 特定的固定行为：

- 属性路由要求
- 自动 HTTP 400 响应
- 绑定源参数推理
- Multipart/form-data 请求推理
- 错误状态代码的问题详细信息

创建Restful风格的API

此处创建以下 API：

API	描述	请求正文	响应正文
`GET /api/todoitems`	获取所有待办事项	None	待办事项的数组
`GET /api/todoitems/{id}`	按 ID 获取项	None	待办事项
`POST /api/todoitems`	添加新项	待办事项	待办事项
`PUT /api/todoitems/{id}`	更新现有项	待办事项	None
`DELETE /api/todoitems/{id}`	删除项	无	None



返回值类型

ASP.NET Core 提供以下 Web API 控制器操作返回类型选项：

- 特定类型
- IActionResult
- ActionResult

特定类型

最简单的操作返回基元或复杂数据类型（如 `string` 或自定义对象类型）。 请参考以下操作，该操作返回自定义 `Product` 对象的集合：

```
[HttpGet]
public List<Product> Get() =>
    _repository.GetProducts();
```

思考：这是一个正常返回的值，但是我们在开发中常常看到只返回一个状态码的情况，比如 `404` 抑或是 `200` ，那这种情况我们怎么去实现呢？这些状态码又属于什么返回值类型了？

IActionResult 类型

当操作中可能有多个 `ActionResult` 返回类型时，适合使用 IActionResult 返回类型。 `ActionResult` 类型表示多种 HTTP 状态代码。 此类别中的某些常见返回类型为 BadRequestResult (400)、NotFoundResult (404) 和 OkObjectResult (200)。

其中有两种可能的返回类型：

```
[HttpGet("{id:int}")]
public IActionResult GetById(int id)
{
    if (id != 1)
    {
        return NotFound();
    }

    return Ok(product);
}
```

在上述操作中：

- 当 `id` 代表的产品不在基础数据存储中时，则返回 404 状态代码。 NotFound 便利方法作为 `return new NotFoundResult();` 的简写调用。
- 如果产品确实存在，则返回状态代码 200 及 `Product` 对象。 Ok 便利方法作为 `return new OkObjectResult(product);` 的简写调用。

思考：好嘛！问题又来了，如果我既要返回普通类型的返回值，又要返回指定状态码，我又该怎么办呢?

ActionResult 类型

ASP.NET Core 包括面向 Web API 控制器操作的 ActionResult 返回类型。 这种返回值类型既能返回普通类型的返回值，又能返回指定状态码。

```
[HttpGet("{id}")]
public ActionResult<Product> GetById(int id)
{
    if (id != 1)
    {
        return NotFound();
    }
    return product;
}
```

Minimal APIs

最小的 API 包括：

- 新的承载 API
- WebApplication和 WebApplicationBuilder
- 新的路由 API

代码：

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.Run();
```

可以通过命令行上的 `dotnet new web` 或在 Visual Studio 中选择“空 Web”模板来创建前面的代码。

以下代码创建 WebApplication(`app`)，而无需显式创建 WebApplicationBuilder：

```
var app = WebApplication.Create(args);

app.MapGet("/", () => "This is a GET");
app.MapPost("/", () => "This is a POST");
app.MapPut("/", () => "This is a PUT");
app.MapDelete("/", () => "This is a DELETE");

app.Run();
```

> 从以上案例可以发现，方法体其实是一个委托，那么既然是委托，ACE就有点小心思了~往下看。

静态方法

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/", HelloWorld.Hello);

string Hi() => "Hello Ace";
app.MapGet("/hello", Hi);

app.MapGet("/user/{userName}/age/{age}",
    (int userName, int age) => $"{userName}今年{age}岁");

app.Run();

class HelloWorld
{
    public static string Hello()
    {
        return "Hello World";
    }
}
```

路由约束

路由约束限制路由的匹配行为。

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/todos/{text}", (string text) => text;
app.MapGet("/posts/{slug:regex(^[a-z0-9_-]+$)}", (string slug) => $"Post
{slug}");

app.Run();
```

传参形式

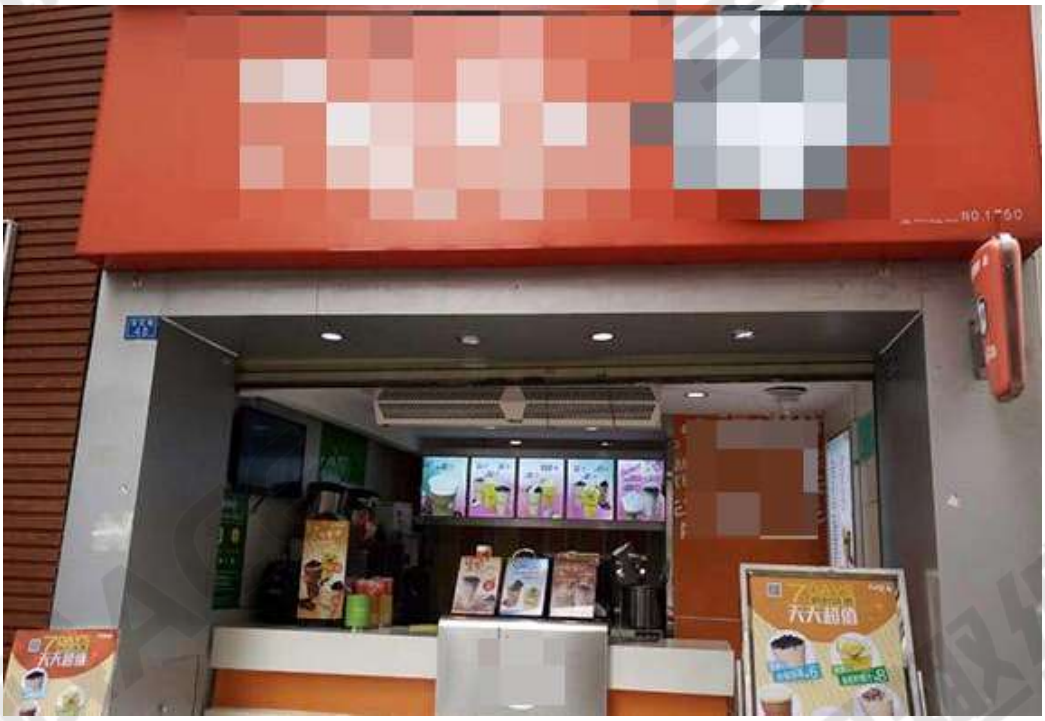
常见的请求方式的参数形式与Restful保持一致。

> 特别强调
>
> Minimal APIs有一种依赖注入的传参形式，这个我们学完一拉注入再说

依赖注入

理解 IOC

IOC（Inversion of Control）即控制反转，那什么又叫控制反转呢？我们来看下面两张图。



问：那么理解了控制反转，那么他和依赖注入又是什么关系呢？

答：其实依赖注入是实现控制反转的一种手段或者方式。

概念

依赖注入（DI: Dependency Injection）我们首先要理清是 **谁依赖谁，为什么需要依赖，谁注入谁，注入了什么**

- 谁依赖于谁：当然是应用程序依赖于IOC容器；
- 为什么需要依赖：应用程序需要IOC容器来提供对象需要的外部资源；
- 谁注入谁：很明显是IOC容器注入应用程序某个对象，应用程序依赖的对象；
- 注入了什么：就是注入某个对象所需要的外部资源（包括对象、资源、常量数据）

ASP.NET Core6 IOC容器

上文中多次提到IOC容器，那么IOC容器是什么?

在 ASP NET Core 6 中的IOC容器就是 `ServiceCollection`，从字面意思理解就是 **服务收集器** 或者叫 **服务集合** 。他就是一个存放服务的容器罐子而已。而这里所谓的服务就是指在开发中需要使用的各种类的统称。注意 `ServiceCollection` 不光只能在Web项目中使用哦!

服务生命周期

服务生命周期分三类

- Transient 瞬时生命周期
- Singleton 单例生命周期
- Scoped 作用域生命周期

注册服务

针对以上三种提到的三种服务的生命周期，自然就对应了三种服务注册的方式，分别是

- AddTransient()
- AddSingleton()
- AddScoped()

具体注册方式如下:

```
services.AddTransient<IUser,User>()
services.AddTransient(typeof(IUser), typeof(User))

services.AddScoped<IUser,User>()
services.AddScoped(typeof(IUser), typeof(User))

services.AddSingleton<IUser,User>()
services.AddSingleton(typeof(IUser), typeof(User))
services.AddSingleton<IUser>(user)
```

思考

在这里我们陡然发现，为什么在单例注册时和另外两个有所不同呢?

服务使用

在 ASP.NET Core6 中，他的原生容器只能使用构造函数注入，具体方式如下:

```
public class IocController
{
    private IUser _user;
    public IocController(IUser user)
```

```
    {
        _user = user;
    }

    [HttpGet]
    public string Get()
    {
        return _user.GetName();
    }
}
```

Filter 过滤器

要了解Filter过滤器，我们首先要知道什么是AOP。AOP即面向切面编程，何为面向切面呢，我们来看以下这个小案例。

小案例：去外地女友家做作业

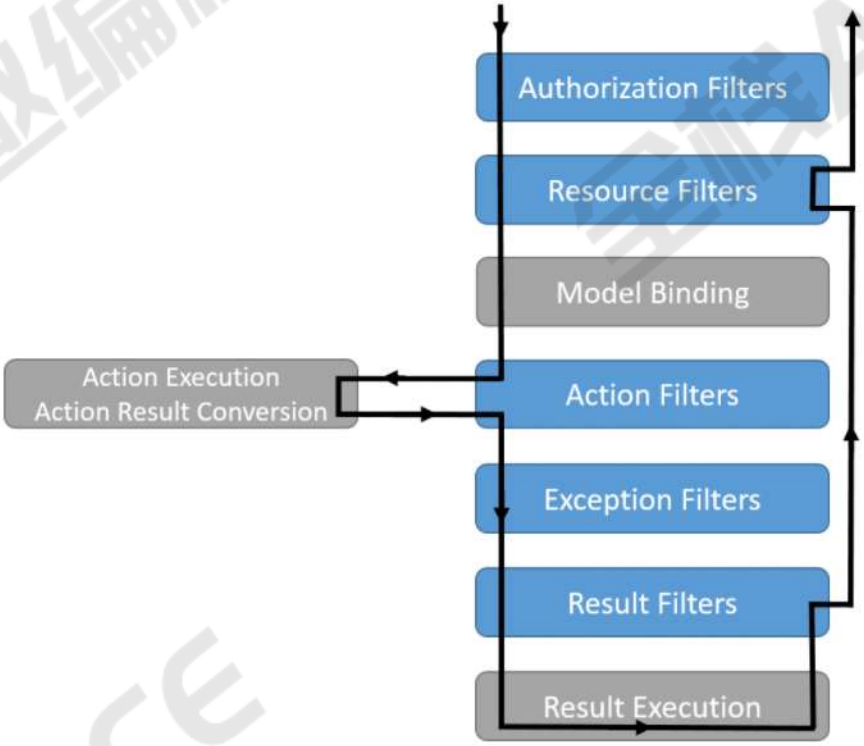
小明放假在家觉得无聊，想去外地的女朋友家交作业。于是他买了上午8:00等G5656次列车车票。但是他由于过于兴奋起早了，六点就到了高铁站。他发现6:10分有一班高铁D5858次到他女友有所在的城市，于是他决定冒险进站，不了在检票口被告知车次与车票不符，车票只限坐当日当次列车，于是小明只有乖乖等到八点，坐上了D5656次列车。请模拟检票口检票的业务，写一个小明拿票进站的案例。

Filter的类型

从以上案例中我们理解了什么是AOP。那么Filter就是实现AOP的一种方式。在ASP .NET Core 中一共有5大Filter

- AuthorizationFilter 授权过滤器
- IResourceFilter 资源管理过滤器
- IActionFilter 行为过滤器
- IExceptionFilter 异常过滤器
- IResultFilter 结果过滤器

那他们的执行顺序又是如何呢?



注册方式

注册方式分为

- 方法注册
- 类注册
- 全局注册

方法注册

```
[HttpGet]
[ActionFilter]
public IEnumerable<WeatherForecast> Get()
{
    return Enumerable.Range(1, 5).Select(index => new WeatherForecast
    {
        Date = DateTime.Now.AddDays(index),
        TemperatureC = Random.Shared.Next(-20, 55),
        Summary = Summaries[Random.Shared.Next(Summaries.Length)]
    })
    .ToArray();
}
```

类注册

```
[ApiController]
[Route("[controller]")]
[ActionFilter]
public class WeatherForecastController : BaseApiController<IUserService>
{
}
```

全局注册

```
builder.Services.AddControllers(o=>o.Filters.Add(typeof(CtmActionFilterAttribute
)));
```

授权过滤器

- 是过滤器管道中运行的第一个过滤器。
- 控制对操作方法的访问。
- 具有在它之前的执行的方法，但没有之后执行的方法。

自定义授权过滤器需要自定义授权框架。 建议配置授权策略或编写自定义授权策略，而不是编写自定义过滤器。 内置授权过滤器：

- 调用授权系统。
- 不授权请求。

不会在授权过滤器中引发异常：

- 不会处理异常。
- 异常过滤器不会处理异常。

在授权过滤器出现异常时请小心应对。

资源过滤器

- 实现 IResourceFilter 或 IAsyncResourceFilter 接口。
- 执行会覆盖过滤器管道的绝大部分。
- 只有授权过滤器在资源过滤器之前运行。

```
public class ShortCircuitingResourceFilterAttribute : Attribute, IResourceFilter
{
    public void OnResourceExecuting(ResourceExecutingContext context)
    {
    }
    public void OnResourceExecuted(ResourceExecutedContext context)
    {
    }
}
```

如果要使大部分**管道短路**，资源过滤器会很有用。 例如，如果缓存命中，则缓存过滤器可以绕开管道的其余阶段。

管道断路器

- 可以防止模型绑定访问表单数据。
- 用于上传大型文件，以防止表单数据被读入内存。

```
public class ShortCircuitingResourceFilterAttribute : Attribute, IResourceFilter
{
    public void OnResourceExecuting(ResourceExecutingContext context)
    {
        context.Result = new ContentResult
        {
            Content = nameof(ShortCircuitingResourceFilterAttribute)
        };
    }

    public void OnResourceExecuted(ResourceExecutedContext context) { }
}
```

操作过滤器

- 实现 IActionFilter 或 IAsyncActionFilter 接口。
- 它们的执行围绕着操作方法的执行。

以下代码显示示例操作过滤器：

```
public class CtmActionFilter : IActionFilter
{
    public void OnActionExecuting(ActionExecutingContext context)
    {
        // Do something before the action executes.
    }

    public void OnActionExecuted(ActionExecutedContext context)
    {
        // Do something after the action executes.
    }
}
```

ActionExecutingContext 提供以下属性：

- ActionArguments - 用于读取操作方法的输入。
- Controller - 用于处理控制器实例。
- Result - 设置 `Result` 会使操作方法和后续操作过滤器的执行短路。

在操作方法中引发异常：

- 防止运行后续过滤器。
- 与设置 `Result` 不同，结果被视为失败而不是成功。

ActionExecutedContext 提供 `Controller` 和 `Result` 以及以下属性：

- Canceled - 如果操作执行已被另一个过滤器设置短路，则为 true。
- Exception - 如果操作或之前运行的操作过滤器引发了异常，则为非 NULL 值。 将此属性设置为 null：

    - 有效地处理异常。
    - 执行 `Result`，从操作方法中将它返回。

该框架提供一个可子类化的抽象 ActionFilterAttribute。

`OnActionExecuting` 操作过滤器可用于：

- 验证模型状态。
- 如果状态无效，则返回错误。

```
public class ValidateModelAttribute : ActionFilterAttribute
{
    public override void OnActionExecuting(ActionExecutingContext context)
    {
        if (!context.ModelState.IsValid)
        {
            context.Result = new BadRequestObjectResult(context.ModelState);
        }
    }
}
```

> 备注
>
> 使用 `[ApiController]` 属性注释的控制器会自动验证模型状态并返回 400 响应。

`OnActionExecuted` 方法在操作方法之后运行：

- 可通过 Result 属性查看和处理操作结果。
- 如果操作执行已被另一个过滤器设置短路，则 Canceled 设置为 true。
- 如果操作或后续操作过滤器引发了异常，则将 `Exception` 设置为非 NULL 值。
    - 有效地处理异常。
    - 执行 `ActionExecutedContext.Result`，从操作方法中将它正常返回。

异常过滤器

- 实现 IExceptionFilter 或 IAsyncExceptionFilter。
- 可用于实现常见的错误处理策略。

下面的异常过滤器示例显示在开发应用时发生的异常的相关详细信息：

```
public class CtmExceptionFilter : IExceptionFilter
{
    public void OnException(ExceptionContext context)
    {
        context.Result = new ContentResult
        {
            Content = context.Exception.ToString()
        };
    }
}
```

以下代码测试异常过滤器：

```
[CtmExceptionFilter]
public class ExceptionController : ControllerBase
{
    [HttpGet]
    public IActionResult Get()
    {
        throw new Exception();
    }
}
```

异常过滤器：

- 没有之前和之后的事件。
- 实现 OnException 或 OnExceptionAsync。
- 处理 Razor 页面或控制器创建、Razor、操作过滤器或操作方法中发生的未经处理的异常。
- 请不要捕获资源过滤器、结果过滤器或 MVC 结果执行中发生的异常。

异常过滤器：

- 非常适合捕获发生在操作中的异常。
- 并不像错误处理中间件那么灵活。

建议使用中间件处理异常。 基于所调用的操作方法，仅当错误处理不同时，才使用异常过滤器。 例如，应用可能具有用于 API 终结点和视图/HTML 的操作方法。 API 终结点可能返回 JSON 形式的错误信息，而基于视图的操作可能返回 HTML 形式的错误页。

结果过滤器

- 实现接口：
  - IResultFilter 或 IAsyncResultFilter
  - IAlwaysRunResultFilter 或 IAsyncAlwaysRunResultFilter
- 它们的执行围绕着操作结果的执行。

IResultFilter 和 IAsyncResultFilter

以下代码显示示例结果过滤器：

```
public class CtmResultFilter : IResultFilter
{
    public void OnResultExecuting(ResultExecutingContext context)
    {
        // Do something before the result executes.
    }

    public void OnResultExecuted(ResultExecutedContext context)
    {
        // Do something after the result executes.
    }
}
```

仅当操作或操作过滤器生成操作结果时，才会执行结果过滤器。 不会在以下情况下执行结果过滤器：

- 授权过滤器或资源过滤器使管道短路。
- 异常过滤器通过生成操作结果来处理异常。

如果在 `IResultFilter.OnResultExecuting` 中引发异常，则会导致：

- 阻止操作结果和后续过滤器的执行。
- 结果被视为失败而不是成功。

IAlwaysRunResultFilter 和 IAsyncAlwaysRunResultFilter

IAlwaysRunResultFilter 和 IAsyncAlwaysRunResultFilter 接口声明了一个针对所有操作结果运行的 IResultFilter 实现。 这包括由以下对象生成的操作结果：

- 设置短路的授权过滤器和资源过滤器。
- 异常过滤器。

例如，以下过滤器始终运行并在内容协商失败时设置具有“422 无法处理的实体”ObjectResult状态代码的操作结果 (ObjectResult)：

```
public class UnprocessableResultFilter : IAlwaysRunResultFilter
{
    public void OnResultExecuting(ResultExecutingContext context)
    {
        if (context.Result is StatusCodeResult statusCodeResult
            && statusCodeResult.StatusCode ==
StatusCodes.Status415UnsupportedMediaType)
        {
            context.Result = new ObjectResult("Unprocessable")
            {
                StatusCode = StatusCodes.Status422UnprocessableEntity
            };
        }
    }

    public void OnResultExecuted(ResultExecutedContext context) { }
}
```

思考

在过滤器中是否可以使用依赖注入呢?

数据交互 EntityFrameworkCore

Entity Framework (EF) Core 是轻量化、可扩展、开源和跨平台版的常用 Entity Framework 数据访问技术。

EF Core 可用作对象关系映射程序 (O/RM)，这可以实现以下两点：

- 使 .NET 开发人员能够使用 .NET 对象处理数据库。
- 无需再像通常那样编写大部分数据访问代码。

EF Core 支持多个数据库引擎，在使用时需要引入Nuget包

```
Microsoft.EntityFrameworkCore
Microsoft.EntityFrameworkCore.SqlServer
Microsoft.EntityFrameworkCore.SqlServer.Design
Microsoft.EntityFrameworkCore.Tools
```

模型生成方式

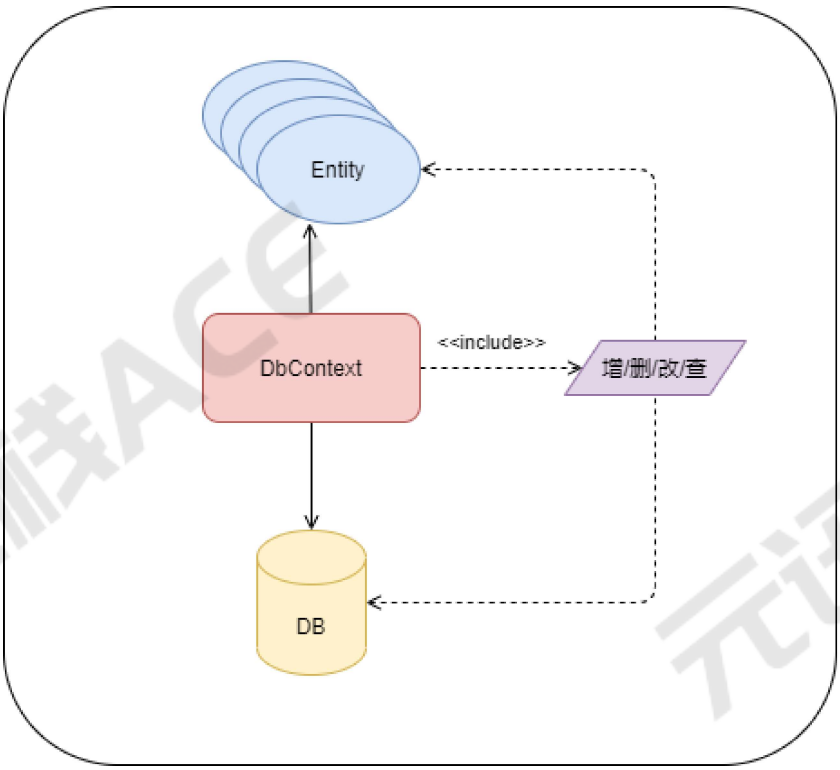
对于 EF Core，使用模型执行数据访问。 模型由实体类和表示数据库会话的上下文对象构成。

EF 支持以下模型开发方法：

- **DbFirst** 从现有数据库生成模型。
- **CodeFirst** 创建模型后，使用 EF 迁移从模型创建数据库。 模型发生变化时，迁移可让数据库不断演进。

什么是上下文

上下文对象用于关联数据库与实体之间的关系，通过实体对数据库进行查询并保存数据。



DbFirst

```
Scaffold-DbContext "Server=localhost;database=TestDb;uid=sa;pwd=1qaz2wsx"
Microsoft.EntityFrameworkCore.SqlServer -OutputDir Entities -ContextDir Entities

-Context TestDbContext -Force

-OutputDir    实体文件所存放的文件目录
-ContextDir    DbContext文件存放的目录
-Context    DbContext文件名
-Schemas    需要生成实体数据的数据表所在的模式
-Tables    需要生成实体数据的数据表的集合
-Force 强制执行，重写已经存在的实体文件
```

查询

使用语言集成查询 (LINQ) 从数据库检索实体类的实例。

```
using (var db = new TestDbContext())
{
    var users = db.User
        .Where(b => b.Age > 15)
        .OrderBy(b => b.UserName)
        .ToList();
}
```

保存数据

使用实体类的实例在数据库中创建、删除和修改数据。

```
using (var db = new TestDbContext())
{
    var user = new User { UserName = "Ace" };
    db.User.Add(user);
    db.SaveChanges();
}
```

CodeFirst

编写上下文

```
using System.Collections.Generic;
using Microsoft.EntityFrameworkCore;

namespace Intro
{
    public class BloggingContext : DbContext
    {
        public DbSet<User> Users { get; set; }

        protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
        {
            optionsBuilder.UseSqlServer(
                @"Server=
(localdb)\mssqllocaldb;Database=TestDb;Trusted_Connection=True");
        }
    }

    public class Users
    {
        public int Id { get; set; }
        public string UserName { get; set; }
        public int Age { get; set; }
    }
}
```

迁移命令

```
add-migration CreateDb   添加迁移文件
update-database   根据最新的迁移文件生成数据库
```

项目案例与部署

以下内容自行发挥

开发登录功能

项目部署

Windows部署

Linux部署