

从零开始实现DDD应用

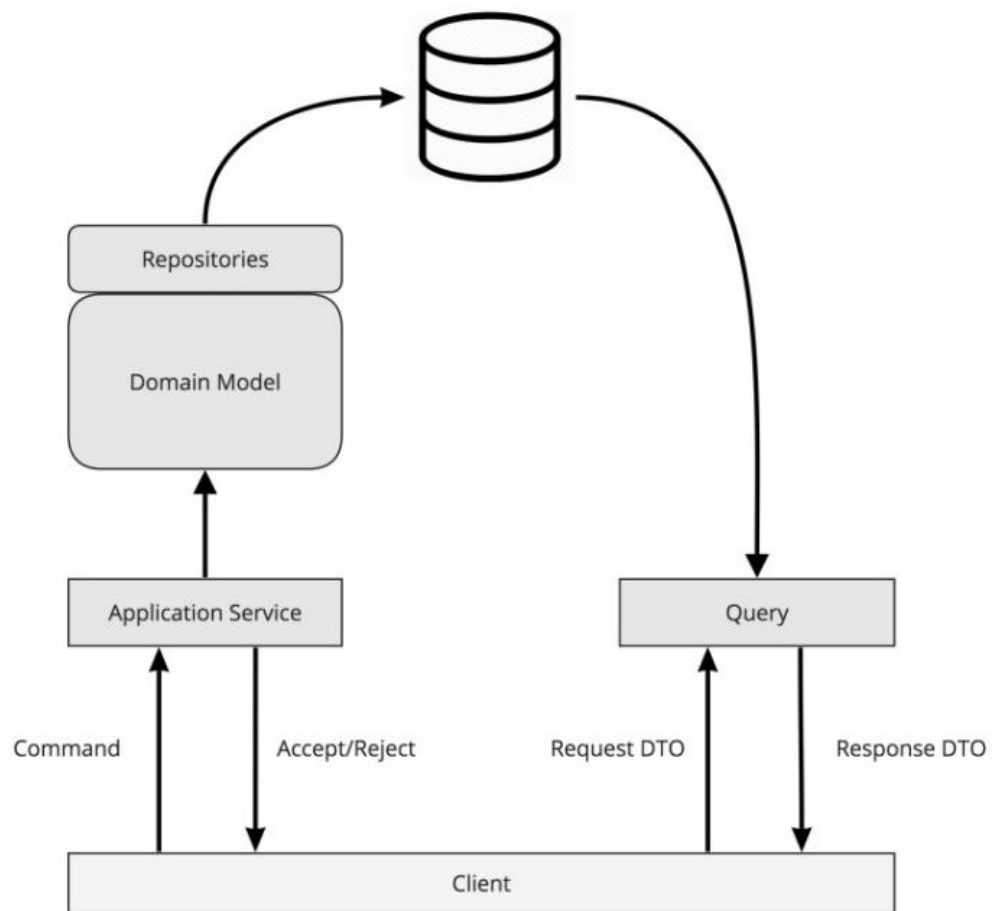
第五章 CQRS

作者：贺传珺

CQRS

CQRS是命令和查询责任分离（Command and Query Responsibility Segregation）的简称。为什么要把查询和命令分离？最直接的原因是绝大多数系统的读写的量是不对等的。某些系统读很多写很少，某些读很少写很多，这时候就得要针对读或者写进行优化，但是优化可以说是万恶之源，要瞻前顾后怕影响到其他的功能。使用CQRS之后，可以更清醒的分离关注点，更方便进行优化。另一方面，使用EFCore等ORM框架生成的SQL语句往往难以优化，一旦出现性能问题很多时候无从下手，在配置上下文的时候也得配置一堆的关系和懒加载项。对于DDD来说，每个聚合对应着一个仓储，在查询多个仓储的时候必然会难以配置关系。CQRS就是为了解决这些问题而存在。

CQRS



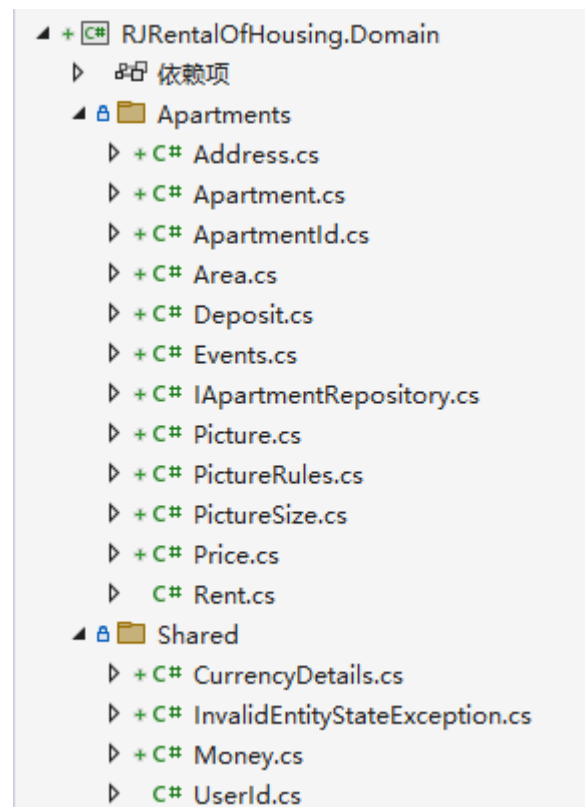
回顾一下第一章开头的这张图。我们已经实现了左边那部分的内容，但是查询几乎没提到，除了一些和业务有关的查询之外。本章就来实现这张图右边的部分。

CQRS

本章会使用到PostgreSQL数据库，ORM框架会同时使用Entity Framework Core和Dapper。使用其它的ORM和数据库也可以，代码会有细微的区别。首先我们将代码整理一下，我们稍后会加入一个新的聚合根。

CQRS

首先在Domain层新建三个文件夹：Apartments、Shard、UserProfiles，然后把文件移动过去。



移动文件夹后记得修改命名空间和Debug，详细代码不在此展示，请查看章节配套代码。

实现用户聚合

在UserProfiles中新建一个Events类。

```
namespace RJRentalOfHousing.Domain.UserProfiles
{
    0 个引用 | 0 项更改 | 0 名作者, 0 项更改
    public static class Events
    {
        0 个引用 | 0 项更改 | 0 名作者, 0 项更改
        public class UserRegistered
        {
            0 个引用 | 0 项更改 | 0 名作者, 0 项更改
            public Guid UserId { get; set; }
            0 个引用 | 0 项更改 | 0 名作者, 0 项更改
            public string FullName { get; set; }
            0 个引用 | 0 项更改 | 0 名作者, 0 项更改
            public string DisplayName { get; set; }
        }

        0 个引用 | 0 项更改 | 0 名作者, 0 项更改
        public class ProfilePhotoUploaded
        {
            0 个引用 | 0 项更改 | 0 名作者, 0 项更改
            public Guid UserId { get; set; }
            0 个引用 | 0 项更改 | 0 名作者, 0 项更改
            public string PhotoUrl { get; set; }
        }
    }
}
```

```
}
0 个引用 | 0 项更改 | 0 名作者, 0 项更改
public class UserFullNameUpdated
{
    0 个引用 | 0 项更改 | 0 名作者, 0 项更改
    public Guid UserId { get; set; }
    0 个引用 | 0 项更改 | 0 名作者, 0 项更改
    public string FullName { get; set; }
}

0 个引用 | 0 项更改 | 0 名作者, 0 项更改
public class UserDisplayNameUpdated
{
    0 个引用 | 0 项更改 | 0 名作者, 0 项更改
    public Guid UserId { get; set; }
    0 个引用 | 0 项更改 | 0 名作者, 0 项更改
    public string DisplayName { get; set; }
}
}
```

实现用户聚合

接下来在UserProfiles文件夹中新建一个FullName类，它是UserProfile中的一个值对象。（为了方便实现，这里先写值对象）

```
namespace RJRentalOfHousing.Domain.UserProfiles
{
    5 个引用 | 0 项更改 | 0 名作者, 0 项更改
    public record FullName
    {
        2 个引用 | 0 项更改 | 0 名作者, 0 项更改
        public string Value { get; internal set; }

        1 个引用 | 0 项更改 | 0 名作者, 0 项更改
        internal FullName(string fullname) => Value = fullname;

        0 个引用 | 0 项更改 | 0 名作者, 0 项更改
        public static FullName FromString(string fullname)
        {
            if (string.IsNullOrEmpty(fullname))
                throw new ArgumentNullException(nameof(fullname));
            return new FullName(fullname);
        }

        public static implicit operator string(FullName fullname) => fullname.Value;

        0 个引用 | 0 项更改 | 0 名作者, 0 项更改
        protected FullName() { }
    }
}
```

实现用户聚合

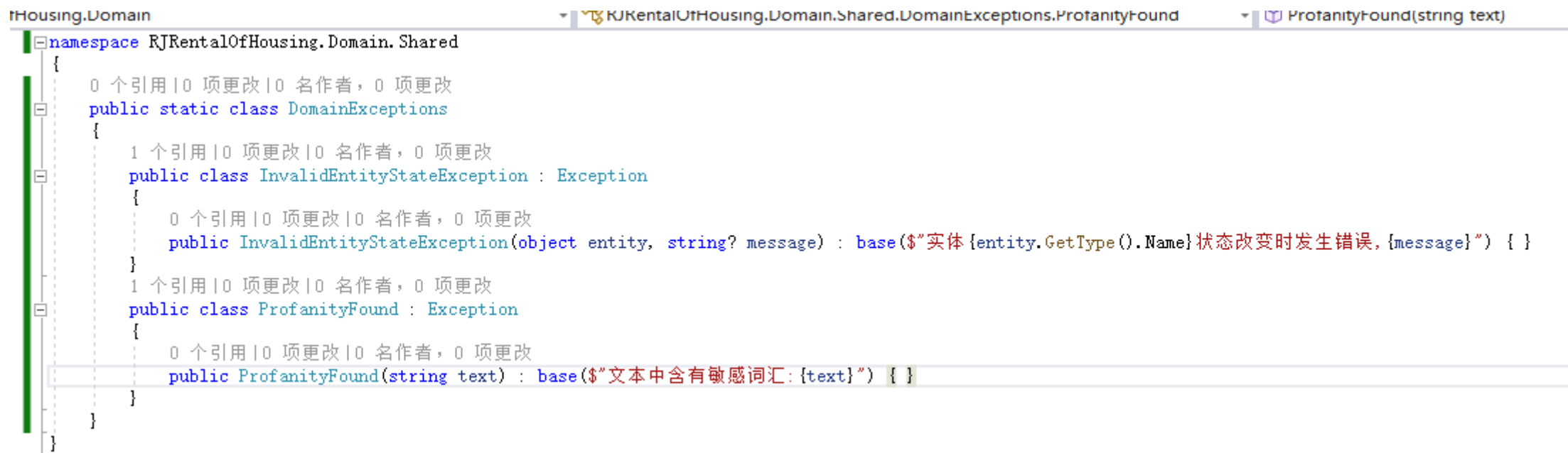
在添加显示名称（DisplayName）之前，我们还有一件事要做。由于显示名称是要公开在页面上的，所以必须注意屏蔽敏感词。在Shard文件夹中新建一个ContentModeration类，创建一个委托用于检测敏感词。

```
namespace RJRentalOfHousing.Domain.Shared
{
    public delegate bool CheckTextForProfanity(string text);
}
```

用委托能实现的功能用接口一定能实现。但是这里选择用委托是因为只有一个方法，而且功能相对简单。

实现用户聚合

现在要新增一个领域中的错误类型，在Shard文件夹中新建一个DomainExceptions类，并把以前实现的InvalidEntityStateException移动过去，然后Debug。



The screenshot shows a Visual Studio code editor with the following C# code:

```
namespace RJRentalOfHousing.Domain.Shared
{
    0 个引用 | 0 项更改 | 0 名作者, 0 项更改
    public static class DomainExceptions
    {
        1 个引用 | 0 项更改 | 0 名作者, 0 项更改
        public class InvalidEntityStateException : Exception
        {
            0 个引用 | 0 项更改 | 0 名作者, 0 项更改
            public InvalidEntityStateException(object entity, string? message) : base($"实体 {entity.GetType().Name} 状态改变时发生错误, {message}") { }
        }
        1 个引用 | 0 项更改 | 0 名作者, 0 项更改
        public class ProfanityFound : Exception
        {
            0 个引用 | 0 项更改 | 0 名作者, 0 项更改
            public ProfanityFound(string text) : base($"文本中含有敏感词汇: {text}") { }
        }
    }
}
```

The code defines a namespace `RJRentalOfHousing.Domain.Shared` containing a static class `DomainExceptions`. Inside `DomainExceptions`, there are two classes: `InvalidEntityStateException` and `ProfanityFound`, both inheriting from `Exception`. The `InvalidEntityStateException` constructor takes an `object` and a `string?` message, and the `ProfanityFound` constructor takes a `string` text. Both constructors use string interpolation to format the error message.

实现用户聚合

现在来实现显示名称的值对象，在UserProfiles中新增一个DisplayName类。

```
using RJRentalOfHousing.Domain.Shared;

namespace RJRentalOfHousing.Domain.UserProfiles
{
    5 个引用 | 0 项更改 | 0 名作者, 0 项更改
    public record DisplayName
    {
        2 个引用 | 0 项更改 | 0 名作者, 0 项更改
        public string Value { get; internal set; }

        1 个引用 | 0 项更改 | 0 名作者, 0 项更改
        internal DisplayName(string displayName) => Value = displayName;

        0 个引用 | 0 项更改 | 0 名作者, 0 项更改
        public static DisplayName FromString(string displayName, CheckTextForProfanity hasProfanity)
        {
            if (string.IsNullOrEmpty(displayName))
                throw new ArgumentNullException(nameof(displayName));
            if (hasProfanity(displayName))
                throw new DomainExceptions.ProfanityFound(displayName);
            return new DisplayName(displayName);
        }

        public static implicit operator string(DisplayName displayName) => displayName.Value;

        0 个引用 | 0 项更改 | 0 名作者, 0 项更改
        protected DisplayName() { }
    }
}
```

实现用户聚合

在UserProfiles新增一个UserProfile类，这个类就是用户信息的聚合根。

```
using RJRentalOfHousing.Domain.Shared;
using RJRentalOfHousing.Framework;

namespace RJRentalOfHousing.Domain.UserProfiles
{
    11 个引用 | 0 项更改 | 0 名作者, 0 项更改
    public class UserProfile : AggregateRoot<UserId>
    {
        0 个引用 | 0 项更改 | 0 名作者, 0 项更改
        protected UserProfile() { }
        2 个引用 | 0 项更改 | 0 名作者, 0 项更改
        public Guid UserProfileId { get; internal set; }
        3 个引用 | 0 项更改 | 0 名作者, 0 项更改
        public FullName? FullName { get; internal set; }
        3 个引用 | 0 项更改 | 0 名作者, 0 项更改
        public DisplayName? DisplayName { get; internal set; }
        1 个引用 | 0 项更改 | 0 名作者, 0 项更改
        public string PhotoUrl { get; internal set; }

        1 个引用 | 0 项更改 | 0 名作者, 0 项更改
        public UserProfile(UserId id, FullName fullName, DisplayName displayName)
            => Apply(new Events.UserRegistered
            {
                UserId = id,
                FullName = fullName,
                DisplayName = displayName
            });
    }
}
```

实现用户聚合

0 个引用 | 0 项更改 | 0 名作者, 0 项更改

```
public void UpdateFullName(FullName fullName)
=> Apply(new Events.UserFullNameUpdated
{
    UserId = Id,
    FullName = fullName
});
```

0 个引用 | 0 项更改 | 0 名作者, 0 项更改

```
public void UpdateDisplayName(DisplayName displayName)
=> Apply(new Events.UserDisplayNameUpdated
{
    UserId = Id,
    DisplayName = displayName
});
```

0 个引用 | 0 项更改 | 0 名作者, 0 项更改

```
public void UpdateProfilePhoto(Uri photoUrl)
=> Apply(new Events.ProfilePhotoUploaded
{
    UserId = Id,
    PhotoUrl = photoUrl.ToString()
});
```

2 个引用 | 0 项更改 | 0 名作者, 0 项更改

```
protected override void EnsureValidState() { }
```

```
protected override void When(object @event)
{
    switch (@event)
    {
        case Events.UserRegistered e:
            Id = new UserId(e.UserId);
            FullName = new FullName(e.FullName);
            DisplayName = new DisplayName(e.DisplayName);
            UserProfileId = e.UserId;
            break;
        case Events.UserFullNameUpdated e:
            FullName = new FullName(e.FullName);
            break;
        case Events.UserDisplayNameUpdated e:
            DisplayName = new DisplayName(e.DisplayName);
            break;
        case Events.ProfilePhotoUploaded e:
            PhotoUrl = e.PhotoUrl;
            break;
    }
}
```

实现用户聚合

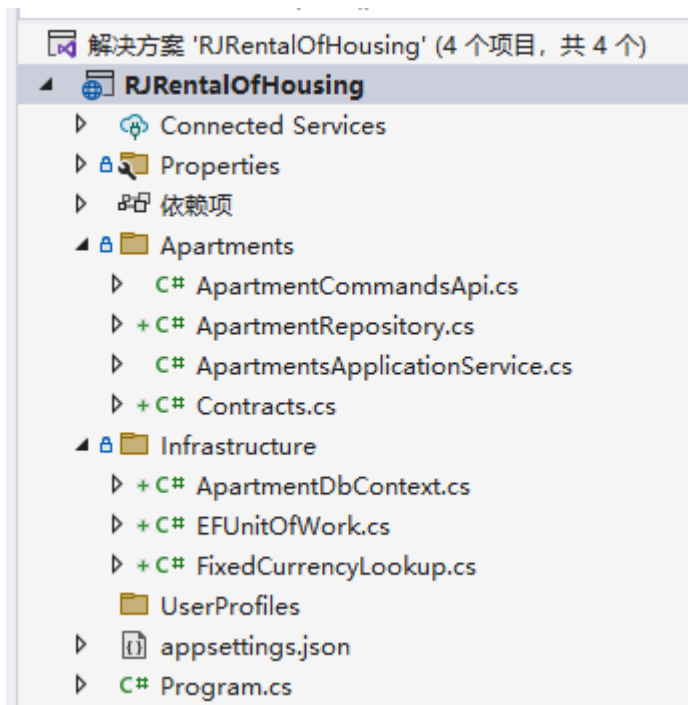
和Apartment一样，用户信息也需要一个仓储和ORM对接。在UserProfiles中新建一个IUserProfileRepository接口。

```
using RJRentalOfHousing.Domain.Shared;

namespace RJRentalOfHousing.Domain.UserProfiles
{
    0 个引用 | 0 项更改 | 0 名作者, 0 项更改
    public interface IUserProfileRepository
    {
        0 个引用 | 0 项更改 | 0 名作者, 0 项更改
        Task<UserProfile> Load(UserId id);
        0 个引用 | 0 项更改 | 0 名作者, 0 项更改
        Task Add(UserProfile entity);
        0 个引用 | 0 项更改 | 0 名作者, 0 项更改
        Task<bool> Exists(UserId id);
    }
}
```

实现用户聚合

整理一下API项目的代码，新建Apartments和UserProfiles两个文件夹，然后移动文件，并且把Apartments合约的类名改为Contracts和修改对应的命名空间，具体过程略，请查看配套代码。



实现用户聚合

在API项目中的UserProfiles文件夹中新建一个Contracts类。

```
namespace RJRentalOfHousing.UserProfiles
{
    0 个引用 | 0 项更改 | 0 名作者, 0 项更改
    public static class Contracts
    {
        0 个引用 | 0 项更改 | 0 名作者, 0 项更改
        public static class V1
        {
            0 个引用 | 0 项更改 | 0 名作者, 0 项更改
            public class RegisterUser
            {
                0 个引用 | 0 项更改 | 0 名作者, 0 项更改
                public Guid UserId { get; set; }
                0 个引用 | 0 项更改 | 0 名作者, 0 项更改
                public string FullName { get; set; }
                0 个引用 | 0 项更改 | 0 名作者, 0 项更改
                public string DisplayName { get; set; }
            }

            0 个引用 | 0 项更改 | 0 名作者, 0 项更改
            public class UpdateUserFullName
            {
                0 个引用 | 0 项更改 | 0 名作者, 0 项更改
                public Guid UserId { get; set; }
                0 个引用 | 0 项更改 | 0 名作者, 0 项更改
                public string FullName { get; set; }
            }
        }
    }
}
```

```
0 个引用 | 0 项更改 | 0 名作者, 0 项更改
public class UpdateDisplayName
{
    0 个引用 | 0 项更改 | 0 名作者, 0 项更改
    public Guid UserId { get; set; }
    0 个引用 | 0 项更改 | 0 名作者, 0 项更改
    public string DisplayName { get; set; }
}

0 个引用 | 0 项更改 | 0 名作者, 0 项更改
public class UpdateUserProfilePhoto
{
    0 个引用 | 0 项更改 | 0 名作者, 0 项更改
    public Guid UserId { get; set; }
    0 个引用 | 0 项更改 | 0 名作者, 0 项更改
    public string PhotoUrl { get; set; }
}
```

实现用户聚合

为新的聚合新增应用服务，在API项目中的UserProfiles文件夹中新建一个UserProfileApplicationService类。

```
using RJRentalOfHousing.Domain.Shared;
using RJRentalOfHousing.Domain.UserProfiles;
using RJRentalOfHousing.Framework;
using static RJRentalOfHousing.UserProfiles.Contracts;

namespace RJRentalOfHousing.UserProfiles
{
    1 个引用 | 0 项更改 | 0 名作者, 0 项更改
    public class UserProfileApplicationService : IApplicationService
    {
        private readonly IUserProfileRepository _repository;
        private readonly IUnitOfWork _unitOfWork;
        private readonly CheckTextForProfanity _checkTextForProfanity;

        0 个引用 | 0 项更改 | 0 名作者, 0 项更改
        public UserProfileApplicationService(IUserProfileRepository repository, IUnitOfWork unitOfWork, CheckTextForProfanity checkTextForProfanity)
        {
            _repository = repository;
            _unitOfWork = unitOfWork;
            _checkTextForProfanity = checkTextForProfanity;
        }
    }
}
```


实现用户聚合

```
private async Task HandleUpdate(Guid userProfileId, Action<UserProfile> operation)
{
    var apartment = await _repository.Load(new UserId(userProfileId));
    if (apartment == null)
        throw new InvalidOperationException($"{userProfileId} 不存在");
    operation(apartment);
    await _unitOfWork.Commit();
}
```

1 个引用 | 0 项更改 | 0 名作者, 0 项更改

```
public async Task Handle(object command)
{
    switch(command)
    {
        case V1.RegisterUser cmd:
            if (await _repository.Exists(new UserId(cmd.UserId)))
                throw new InvalidOperationException($"{cmd.UserId} 已存在");
            var userProfile = new UserProfile(new UserId(cmd.UserId), FullName.FromString(cmd.FullName), DisplayName.FromString(cmd.DisplayName, _checkTextForProfanity));
            await _repository.Add(userProfile);
            await _unitOfWork.Commit();
            break;
        case V1.UpdateUserFullName cmd:
            await HandleUpdate(cmd.UserId, profile => profile.UpdateFullName(FullName.FromString(cmd.FullName)));
            break;
        case V1.UpdateDisplayName cmd:
            await HandleUpdate(cmd.UserId, profile => profile.UpdateDisplayName(DisplayName.FromString(cmd.DisplayName, _checkTextForProfanity)));
            break;
        case V1.UpdateUserProfilePhoto cmd:
            await HandleUpdate(cmd.UserId, profile => profile.UpdateProfilePhoto(new Uri(cmd.PhotoUrl)));
            break;
        default:
            throw new InvalidOperationException($"{cmd.GetType().FullName} 未知的命令类型");
    }
}
```

实现用户聚合

现在实现WebAPI，在此之前我们把一些等会要用到的公共代码移出来。在Infrastructure文件夹中新建一个RequestHandler类。

```
using Microsoft.AspNetCore.Mvc;

namespace RJRentalOfHousing.Infrastructure
{
    0 个引用 | 0 项更改 | 0 名作者, 0 项更改
    public static class RequestHandler
    {
        0 个引用 | 0 项更改 | 0 名作者, 0 项更改
        public static async Task<IActionResult> HandleRequest<T>(T request, Func<T, Task> handler, Serilog.ILogger log)
        {
            try
            {
                log.Debug("正在处理类型为 {type} 的 HTTP 请求", typeof(T).Name);
                await handler(request);
                return new OkResult();
            }
            catch (Exception e)
            {
                log.Error("处理请求失败", e);
                return new BadRequestObjectResult(new { error = e.Message, stackTrace = e.StackTrace });
            }
        }
    }
}
```

实现用户聚合

实现WebAPI，在UserProfiles文件夹中新建UserProfileCommandsApi类，类型为API控制器-空。

```
using Microsoft.AspNetCore.Mvc;
using RJRentalOfHousing.Infrastructure;
using static RJRentalOfHousing.UserProfiles.Contracts;

namespace RJRentalOfHousing.UserProfiles
{
    [Route("/profile")]
    2 个引用 | 0 项更改 | 0 名作者, 0 项更改
    public class UserProfileCommandsApi : Controller
    {
        private readonly UserProfileApplicationService _applicationService;
        private static readonly Serilog.ILogger Log = Serilog.Log.ForContext<UserProfileCommandsApi>();

        0 个引用 | 0 项更改 | 0 名作者, 0 项更改
        public UserProfileCommandsApi(UserProfileApplicationService applicationService) => _applicationService = applicationService;

        [HttpPost]
        0 个引用 | 0 项更改 | 0 名作者, 0 项更改
        public Task<IActionResult> Post(V1.RegisterUser request) => RequestHandler.HandleRequest(request, _applicationService.Handle, Log);

        [Route("fullname")]
        [HttpPut]
        0 个引用 | 0 项更改 | 0 名作者, 0 项更改
        public Task<IActionResult> Put(V1.UpdateUserFullName request) => RequestHandler.HandleRequest(request, _applicationService.Handle, Log);
    }
}
```

实现用户聚合

```
[Route("displayname")]
```

```
[HttpPut]
```

```
0 个引用 | 0 项更改 | 0 名作者, 0 项更改
```

```
public Task<IActionResult> Put(V1.UpdateDisplayName request) => RequestHandler.HandleRequest(request, _applicationService.Handle, Log);
```

```
[Route("photo")]
```

```
[HttpPut]
```

```
0 个引用 | 0 项更改 | 0 名作者, 0 项更改
```

```
public Task<IActionResult> Put(V1.UpdateUserProfilePhoto request) => RequestHandler.HandleRequest(request, _applicationService.Handle, Log);
```

```
}
```

```
}
```

实现用户聚合

还记得第一章提到的领域服务吗？这里还得实现一个，刚才我们说到显示名称要过滤敏感词，我找到了一个免费的接口。现在用这个来实现需求。在Infrastructure文件夹中新建PurgomalumClient类。

```
using Microsoft.AspNetCore.WebUtilities;

namespace RJRentalOfHousing.Infrastructure
{
    2 个引用 | 0 项更改 | 0 名作者, 0 项更改
    public class PurgomalumClient
    {
        private readonly HttpClient _httpClient;

        0 个引用 | 0 项更改 | 0 名作者, 0 项更改
        public PurgomalumClient() { }

        0 个引用 | 0 项更改 | 0 名作者, 0 项更改
        public PurgomalumClient(HttpClient httpClient) => _httpClient = httpClient;

        0 个引用 | 0 项更改 | 0 名作者, 0 项更改
        public async Task<bool> CheckForProfanity(string text)
        {
            var result = await _httpClient.GetAsync(QueryHelpers.AddQueryString("https://www.purgomalum.com/service/containsprofanity", "text", text));
            var value = await result.Content.ReadAsStringAsync();
            return bool.Parse(value);
        }
    }
}
```

实现用户聚合

修改DbContext，加入UserProfile的配置。现在DbContext不只有Apartments的配置，所以类名得改一下，改成RentalDbContext。

3 个引用 | 0 项更改 | 0 名作者，0 项更改

```
public DbSet<Apartment> Apartments { get; set; }
```

0 个引用 | 0 项更改 | 0 名作者，0 项更改

```
public DbSet<UserProfile> UserProfiles { get; set; }
```

0 个引用 | 0 项更改 | 0 名作者，0 项更改

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
```

```
{  
    optionsBuilder.UseLoggerFactory(_loggerFactory);  
    optionsBuilder.EnableSensitiveDataLogging();  
}
```

0 个引用 | 0 项更改 | 0 名作者，0 项更改

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
```

```
{  
    modelBuilder.ApplyConfiguration(new ApartmentEntityTypeConfiguration());  
    modelBuilder.ApplyConfiguration(new PictureEntityTypeConfiguration());  
    modelBuilder.ApplyConfiguration(new UserProfileEntityTypeConfiguration());  
}
```

1 个引用 | 0 项更改 | 0 名作者，0 项更改

```
public class UserProfileEntityTypeConfiguration : IEntityTypeConfiguration<UserProfile>
```

```
{
```

0 个引用 | 0 项更改 | 0 名作者，0 项更改

```
public void Configure(EntityTypeBuilder<UserProfile> builder)
```

```
{
```

```
    builder.HasKey(x => x.UserProfileId);  
    builder.OwnsOne(x => x.Id);  
    builder.OwnsOne(x => x.DisplayName);  
    builder.OwnsOne(x => x.FullName);  
}
```

```
}
```

实现用户聚合

用EFCore来实现仓储接口，在UserProfiles文件夹中新建UserProfileRepository类。

```
using RJRentalOfHousing.Domain.Shared;
using RJRentalOfHousing.Domain.UserProfiles;
using RJRentalOfHousing.Infrastructure;

namespace RJRentalOfHousing.UserProfiles
{
    1 个引用 | 0 项更改 | 0 名作者, 0 项更改
    public class UserProfileRepository : IUserProfileRepository
    {
        private readonly RentalDbContext _dbContext;

        0 个引用 | 0 项更改 | 0 名作者, 0 项更改
        public UserProfileRepository(RentalDbContext dbContext) => _dbContext = _dbContext;

        2 个引用 | 0 项更改 | 0 名作者, 0 项更改
        public async Task Add(UserProfile entity) => await _dbContext.UserProfiles.AddAsync(entity);

        2 个引用 | 0 项更改 | 0 名作者, 0 项更改
        public async Task<bool> Exists(UserId id) => await _dbContext.UserProfiles.FindAsync(id.Value) != null;

        2 个引用 | 0 项更改 | 0 名作者, 0 项更改
        public async Task<UserProfile> Load(UserId id) => await _dbContext.UserProfiles.FindAsync(id.Value);
    }
}
```

实现用户聚合

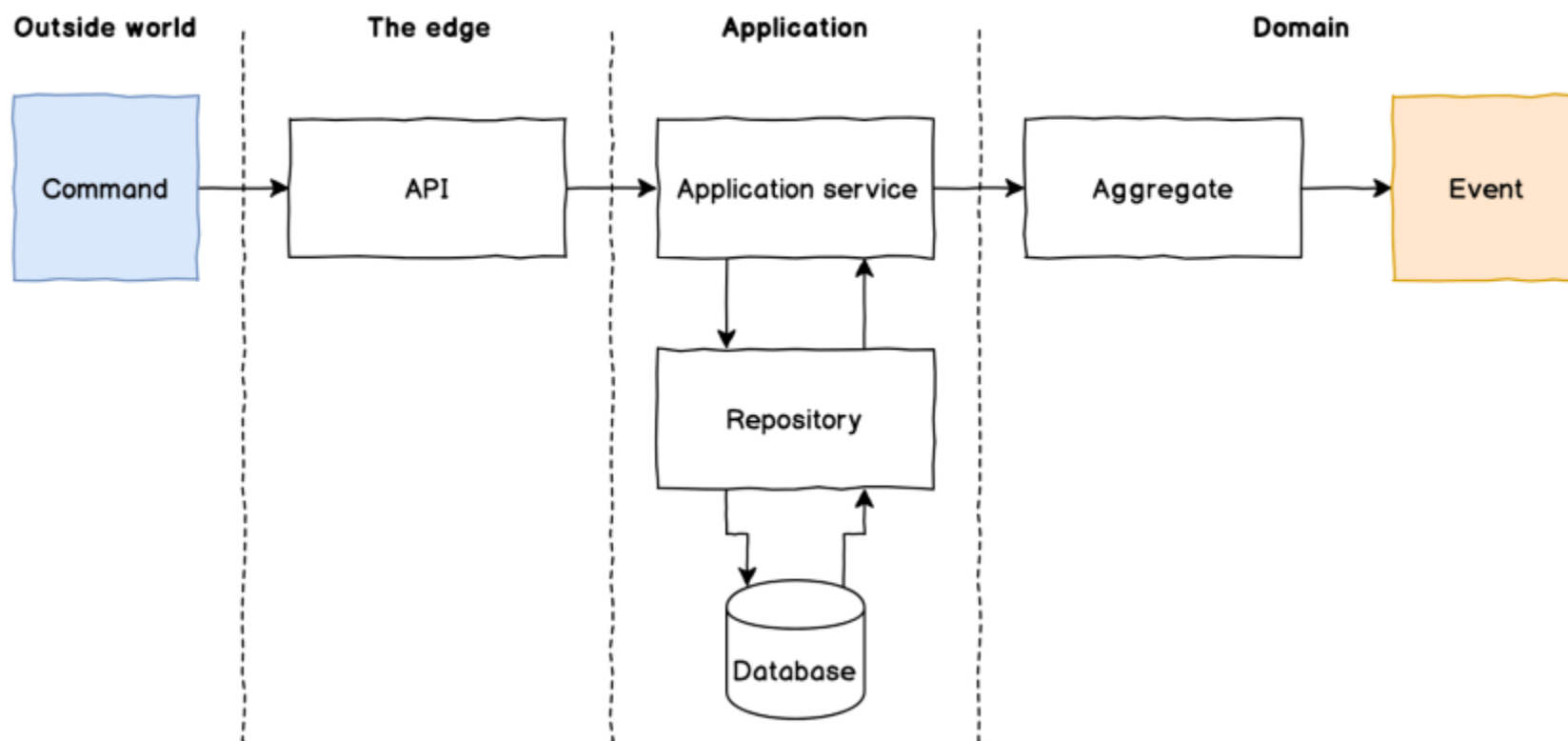
现在还剩最后一步，在Program里注册刚才实现的服务，这里只列出需要改变的地方。

```
builder.Services.AddSwaggerGen();

var purgomalumClient = new PurgomalumClient();
builder.Services.AddEntityFrameworkNpgsql().AddDbContext<RentalDbContext>(options => options.UseNpgsql(connectingString), ServiceLifetime.Singleton);
builder.Services.AddSingleton<IUnitOfWork, EFUnitOfWork>();
builder.Services.AddScoped<ICurrencyLookup, FixedCurrencyLookup>();
builder.Services.AddScoped<IApartmentRepository, ApartmentRepository>();
builder.Services.AddScoped<IUserProfileRepository, UserProfileRepository>();
builder.Services.AddScoped<ApartmentsApplicationService>();
builder.Services.AddScoped<UserProfileApplicationService>
    (c => new UserProfileApplicationService(
        c.GetService<IUserProfileRepository>(),
        c.GetService<IUnitOfWork>(),
        text => purgomalumClient.CheckForProfanity(text).GetAwaiter().GetResult()));
var app = builder.Build();
```

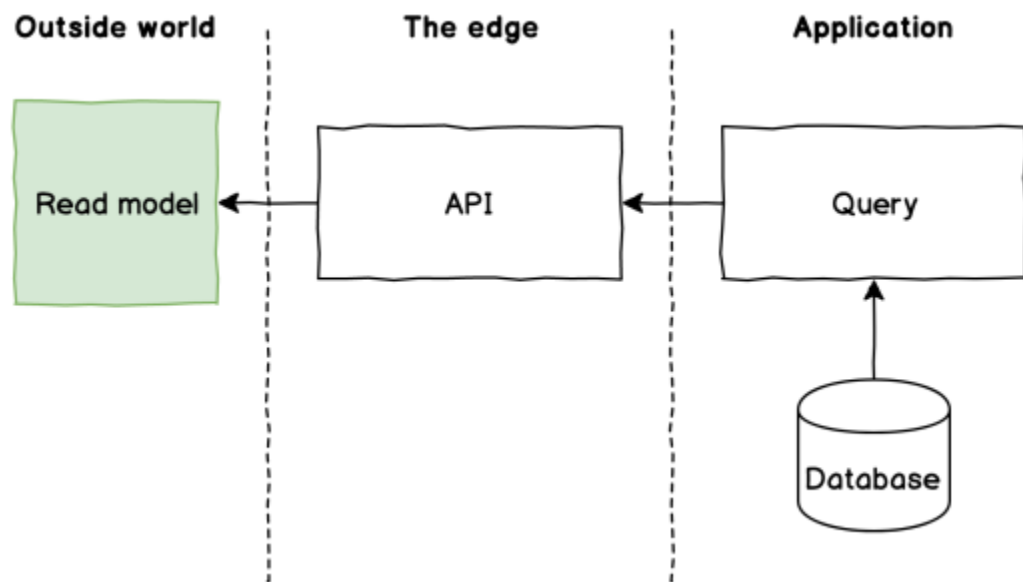

实现查询端

在开始写代码之前，我们再回顾一下CQRS中的流程。首先是命令也就是增、删、改等写的操作。



实现查询端

然后是读的流程。



读的流程非常简单，原因本章开头的时候已经讲过。

实现查询端

首先从定义读取模型开始，在API项目的Apartments文件夹中新建ReadModels类。

```
namespace RJRentalOfHousing.Apartments
{
    1 个引用 | 0 项更改 | 0 名作者, 0 项更改
    public static class ReadModels
    {
        0 个引用 | 0 项更改 | 0 名作者, 0 项更改
        public class ApartmentDetails
        {
            0 个引用 | 0 项更改 | 0 名作者, 0 项更改
            public Guid ApartmentId { get; set; }
            0 个引用 | 0 项更改 | 0 名作者, 0 项更改
            public string Address { get; set; }
            0 个引用 | 0 项更改 | 0 名作者, 0 项更改
            public decimal Areas { get; set; }
            0 个引用 | 0 项更改 | 0 名作者, 0 项更改
            public decimal Rent { get; set; }
            0 个引用 | 0 项更改 | 0 名作者, 0 项更改
            public string RentCurrencyCode { get; set; }
            0 个引用 | 0 项更改 | 0 名作者, 0 项更改
            public decimal Deposit { get; set; }
            0 个引用 | 0 项更改 | 0 名作者, 0 项更改
            public string DepositCurrencyCode { get; set; }
            0 个引用 | 0 项更改 | 0 名作者, 0 项更改
            public string OwnerDisplayName { get; set; }
            0 个引用 | 0 项更改 | 0 名作者, 0 项更改
            public string[] PhotoUrls { get; set; }
            0 个引用 | 0 项更改 | 0 名作者, 0 项更改
            public string Remark { get; set; }
        }
    }
}
```

```
public class ApartmentListItem
{
    0 个引用 | 0 项更改 | 0 名作者, 0 项更改
    public Guid ApartmentId { get; set; }
    0 个引用 | 0 项更改 | 0 名作者, 0 项更改
    public string Address { get; set; }
    0 个引用 | 0 项更改 | 0 名作者, 0 项更改
    public decimal Areas { get; set; }
    0 个引用 | 0 项更改 | 0 名作者, 0 项更改
    public decimal Rent { get; set; }
    0 个引用 | 0 项更改 | 0 名作者, 0 项更改
    public string RentCurrencyCode { get; set; }
    0 个引用 | 0 项更改 | 0 名作者, 0 项更改
    public decimal Deposit { get; set; }
    0 个引用 | 0 项更改 | 0 名作者, 0 项更改
    public string DepositCurrencyCode { get; set; }
}
```

实现查询端

然后定义我们的查询模型，在Apartments文件夹中新建QueryModels类。

```
housing
└─ namespace RJRentalOfHousing.Apartments
    {
        0 个引用 | 0 项更改 | 0 名作者, 0 项更改
        public static class QueryModels
        {
            0 个引用 | 0 项更改 | 0 名作者, 0 项更改
            public class GetAllApartmentsByPage
            {
                0 个引用 | 0 项更改 | 0 名作者, 0 项更改
                public int Page { get; set; }
                0 个引用 | 0 项更改 | 0 名作者, 0 项更改
                public int PageSize { get; set; }
            }

            0 个引用 | 0 项更改 | 0 名作者, 0 项更改
            public class GetOwnersApartment
            {
                0 个引用 | 0 项更改 | 0 名作者, 0 项更改
                public Guid OwnerId { get; set; }
                0 个引用 | 0 项更改 | 0 名作者, 0 项更改
                public int Page { get; set; }
                0 个引用 | 0 项更改 | 0 名作者, 0 项更改
                public int PageSize { get; set; }
            }
        }
    }
}
```

```
0 个引用 | 0 项更改 | 0 名作者, 0 项更改
public class GetPublishedApartment
{
    0 个引用 | 0 项更改 | 0 名作者, 0 项更改
    public int Page { get; set; }
    0 个引用 | 0 项更改 | 0 名作者, 0 项更改
    public int PageSize { get; set; }
}

0 个引用 | 0 项更改 | 0 名作者, 0 项更改
public class GetApartmentDetailById
{
    0 个引用 | 0 项更改 | 0 名作者, 0 项更改
    public Guid ApartmentId { get; set; }
}
}
```

实现查询端

现在开始实现与数据库交互的部分，这部分内容需要使用到Dapper，安装过程略。在Apartments文件夹中新建Queries类，并且实现一个查询。

```
using System.Data.Common;
using static RJRentalOfHousing.Apartments.ReadModels;
using static RJRentalOfHousing.Apartments.QueryModels;
using Dapper;
using RJRentalOfHousing.Domain.Apartments;

namespace RJRentalOfHousing.Apartments
{
    0 个引用 | 0 项更改 | 0 名作者, 0 项更改
    public static class Queries
    {
        1 个引用 | 0 项更改 | 0 名作者, 0 项更改
        private static int Offset(int page, int pageSize) => page * pageSize;

        0 个引用 | 0 项更改 | 0 名作者, 0 项更改
        public static Task<IEnumerable<ApartmentListItem>> Query(this DbConnection connection, GetCreatedApartment query)
            => connection.QueryAsync<ApartmentListItem>("SELECT \"ApartmentId\", \"Areas_Value\", \"Address_Value\", \"Rent_Amount\", \"Rent_Currency_CurrencyCode\", \"Deposit_Amount\", \"Deposit_Currency_CurrencyCode\" FROM \"Apartments\" Where State=@State LIMIT @PageSize OFFSET @Offset",
                new
                {
                    State = (int)Apartment.ApartmentState.Created,
                    PageSize = query.PageSize,
                    Offset = Offset(query.Page, query.PageSize)
                });
    }
}
```

实现查询端

在RequestHandler类中添加一个方法，用来处理查询。

```
0 个引用 | 0 项更改 | 0 名作者， 0 项更改  
public static async Task<IActionResult> HandleQuery<TModel>(Func<Task<TModel>> query, Serilog.ILogger log)  
{  
    try  
    {  
        return new OkObjectResult(await query());  
    }  
    catch (Exception e)  
    {  
        log.Error(e, "查询时发生错误");  
        return new BadRequestObjectResult(new { error = e.Message, stackTrace = e.StackTrace });  
    }  
}
```

实现查询端

我们先实现这一个查询，现在在Apartments文件夹中新建一个ApartmentsQueryApi类，它是一个WebApi控制器。

```
using Microsoft.AspNetCore.Mvc;
using RJRentalOfHousing.Infrastructure;
using Serilog;
using System.Data.Common;

namespace RJRentalOfHousing.Apartments
{
    [Route("/apartment")]
    [ApiController]
    2 个引用 | 0 项更改 | 0 名作者, 0 项更改
    public class ApartmentsQueryApi : Controller
    {
        private readonly DbConnection _dbConnection;
        private static Serilog.ILogger _log = Log.ForContext<ApartmentsQueryApi>();

        0 个引用 | 0 项更改 | 0 名作者, 0 项更改
        public ApartmentsQueryApi(DbConnection dbConnection) => _dbConnection = dbConnection;

        [HttpGet]
        [Route("createlist")]
        0 个引用 | 0 项更改 | 0 名作者, 0 项更改
        public Task<ActionResult> Query([FromQuery] QueryModels.GetCreatedApartment request)
            => RequestHandler.HandleQuery(() => _dbConnection.Query(request), _log);
    }
}
```

实现查询端

修改一下Program中的连接字符串，并新增一个DBConnect的注入。然后运行项目，通过WebAPI在插入一些数据。

```
using
using RJRentalOfHousing.Domain.Shared;
using RJRentalOfHousing.Domain.UserProfiles;
using RJRentalOfHousing.Framework;
using RJRentalOfHousing.Infrastructure;
using RJRentalOfHousing.UserProfiles;
using System.Data.Common;

var builder = WebApplication.CreateBuilder(args);

const string connectionString = "Host=127.0.0.1;Database=Chapter5;Username=postgres;Password=123456";
// Add services to the container.

builder.Services.AddControllers();
// Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

var purgomalumClient = new PurgomalumClient();
builder.Services.AddScoped<DbConnection>(c => new NpgsqlConnection(connectionString));
builder.Services.AddEntityFrameworkNpgsql().AddDbContext<RentalDbContext>(options => options.UseNpgsql(connectionString), ServiceLifetime.Singleton);
builder.Services.AddSingleton<IUnitOfWork, EFUnitOfWork>();
builder.Services.AddScoped<ICurrencyLookup, FixedCurrencyLookup>();
builder.Services.AddScoped<IApartmentRepository, ApartmentRepository>();
builder.Services.AddScoped<IUserProfileRepository, UserProfileRepository>();
builder.Services.AddScoped<ApartmentsApplicationService>();
builder.Services.AddScoped<UserProfileApplicationService>
(c => new UserProfileApplicationService(
    c.GetService<IUserProfileRepository>(),
    c.GetService<IUnitOfWork>(),
    text => purgomalumClient.CheckForProfanity(text).GetAwaiter().GetResult()));
var app = builder.Build();
```


实现查询端

调用创建接口之后，表里的数据大概如下：

对象 无标题 - 查询 Apartments @Chapter5.public (本机P...							
开始事务 文本 筛选 排序 导入 导出							
ApartmentId	Areas_Value	Address_Value	Rent_Amount	Rent_Currency_CurrencyC	Rent_Currency_InUse	Rent_Currency_DecimalPlz	Deposit_A
▶ 3fa85f64-5717-4562-b3fc-2c963f66afa6	140	广西南宁市青秀区桃源路80号	5000	CNY	f	0	

实现查询端

现在运行我们的查询接口，看看能不能正常查询出数据。

Parameters

Cancel

Name	Description
Page	<input type="text" value="0"/>
integer(\$int32)	(query)
PageSize	<input type="text" value="10"/>
integer(\$int32)	(query)

Execute

Clear

Responses

Curl

curl -X 'GET' \n'https://localhost:7294/apartment/createlist?Page=0&PageSize=10' \n-H 'accept: */*'

Request URL

https://localhost:7294/apartment/createlist?Page=0&PageSize=10

Server response

Code	Details
200	<div><div>Response body</div><div><pre>{ "apartmentId": "3fa85f64-5717-4562-b3fc-2c963f66afa6", "address": "广西南宁市青秀区桃海路88号", "areas": 140, "rent": 5000, "rentCurrencyCode": "CNY", "deposit": 0, "depositCurrencyCode": null }</pre></div><div><div>Download</div></div></div>

Response headers

```
content-type: application/json; charset=utf-8
date: Sun, 27 Nov 2022 09:05:22 GMT
server: Kestrel
```

查询接口能正常查询出数据。有人会问为什么查询不做单元测试？因为对查询做单元测试的意义不大，查询始终要依赖基础设施（数据库），而且在CQRS中，查询和业务无关，对查询做单元测试实际上只是对Json序列化做测试而已。剩下的查询不再重复演示，各位可以查看章节配套代码。

结束语

本章讲解了CQRS和实现查询端，以及为什么不用EFCore和通过仓储实现查询的原因。整篇教程到此结束。接下来还有什么内容？这篇教程讲解的只是DDD实现的一部分。接下来还有Event Store、ESB、微服务框架等。