MediatR让进程内通信如此简单，基于MediatR实现事件订阅发布功能

作者：微信公众号：【架构师老卢】 11-14 7:39

概述：当使用 MediatR 这个.NET库时，你可以实现各种不同的应用方法，包括基础功能的使用方法以及一些高级应用。下面将详细介绍 MediatR 在.NET应用中的各种用法，包括基础用法和高级应用，提供带有中文注释的源代码示例。

当使用 MediatR 这个.NET库时，你可以实现各种不同的应用方法，包括基础功能的使用方法以及一些高级应用。下面将详细介绍 MediatR 在.NET应用中的各种用法，包括基础用法和高级应用，提供带有中文注释的源代码示例。

MediatR 简介

MediatR 是一个.NET库，用于实现 Mediator 模式，它允许你将请求和处理程序解耦，从而提高代码的可维护性和可扩展性。在 Mediator 模式中，消息发送者（请求）不直接与消息处理者（处理程序）通信，而是通过中介者（MediatR）来传递消息。这可以帮助降低代码的复杂度，使应用程序更容易扩展和维护。

基础功能的使用方法

首先，让我们从 MediatR 的基础功能开始，包括请求和处理程序的创建、注册和使用。

1. 创建请求和处理程序

在使用 MediatR 之前，你需要创建请求和处理程序。

```
// 创建一个请求类，它代表一个请求消息
public class MyRequest : IRequest<string>
{
    public string Message { get; set; }
}

// 创建一个处理程序类，用于处理请求
public class MyRequestHandler : IRequestHandler<MyRequest, string>
{
    public async Task<string> Handle(MyRequest request, CancellationToken cancellationToken)
    {
        // 处理请求的逻辑在这里，然后返回结果
        return $"处理请求: {request.Message}";
    }
}
```

2. 注册 MediatR

接下来，你需要在应用程序中注册 MediatR 服务。通常，这是在启动时进行的操作。

```
public void ConfigureServices(IServiceCollection services)
{
    // 注册 MediatR 服务
    services.AddMediatR(Assembly.GetExecutingAssembly());
}
```

这样，MediatR 将能够识别并管理你的请求和处理程序。

3. 发送请求

现在，你可以在你的应用程序中发送请求，MediatR 将负责将请求传递给正确的处理程序，并返回处理结果。

```
public class MyController : ControllerBase
{
    private readonly IMediator _mediator;

    public MyController(IMediator mediator)
    {
        _mediator = mediator;
    }

    [HttpGet]
    public async Task<IActionResult> Get()
    {
        var request = new MyRequest { Message = "Hello, MediatR!" };
        var response = await _mediator.Send(request);

        return Ok(response);
    }
}
```

这个简单示例演示了如何创建请求、处理程序、注册 MediatR 服务以及发送请求。MediatR 将自动路由请求到正确的处理程序，然后返回响应。

高级应用

除了基本功能，MediatR 还提供了一些高级功能，以帮助你更好地组织和扩展你的代码。

4. 中介者管道

MediatR 提供了中介者管道，你可以在请求处理前后执行一些操作，如身份验证、日志记录等。这有助于分离关注点和提高代码的可维护性。

创建中介者管道

首先，我们来创建一个中介者管道，用于记录请求和响应的日志。

```
public class LoggingMiddleware<TRequest, TResponse> : IPipelineBehavior<TRequest, TResponse>
{
    private readonly ILogger<LoggingMiddleware<TRequest, TResponse>> _logger;

    public LoggingMiddleware(ILogger<LoggingMiddleware<TRequest, TResponse>> logger)
    {
        _logger = logger;
    }

    public async Task<TResponse> Handle(TRequest request, CancellationToken cancellationToken, RequestHandlerDelegate<TResponse> ne
    {
        _logger.LogInformation("处理请求: {Request}", request);
        var response = await next();
        _logger.LogInformation("处理结果: {Response}", response);
        return response;
    }
}
```

注册中介者管道

在 `Startup.cs` 文件中，将中介者管道注册到 MediatR。

```
services.AddTransient(typeof(IPipelineBehavior<,>), typeof(LoggingMiddleware<,>));
```

现在，每次发送请求时，LoggingMiddleware 将记录请求和响应信息，帮助你跟踪请求的执行过程。

5. 异常处理

MediatR 还允许你处理请求处理过程中可能发生的异常。你可以创建一个异常处理程序，并在需要时将其注册到 MediatR。

创建异常处理程序

```
public class ExceptionHandlingMiddleware<TRequest, TResponse> : IPipelineBehavior<TRequest, TResponse>
{
    public async Task<TResponse> Handle(TRequest request, CancellationToken cancellationToken, RequestHandlerDelegate<TResponse> ne
    {
        try
        {
            return await next();
        }
        catch (Exception ex)
        {
            // 处理异常，例如记录日志或返回自定义错误信息
            throw;
        }
    }
}
```

注册异常处理程序

在 `Startup.cs` 文件中，将异常处理程序注册到 MediatR。

```
services.AddTransient(typeof(IPipelineBehavior<,>), typeof(ExceptionHandlingMiddleware<,>));
```

现在，当请求处理程序中发生异常时，异常处理程序将捕获并处理它，这有助于提高应用程序的可靠性和健壮性。

6. 多个处理程序

MediatR 允许你将多个处理程序与一个请求相关联，这是一个非常有用的功能，特别是在需要执行多个操作或获取多个不同处理程序的结果时。

创建多个处理程序

假设我们有一个额外的处理程序用于处理相同的请求。

```
public class MySecondRequestHandler : IRequestHandler<MyRequest, string>
{
    public async Task<string> Handle(MyRequest request, CancellationToken cancellationToken)
    {
        return $"第二个处理程序: {request.Message}";
    }
}
```

发送请求到多个处理程序

在控制器中，你可以发送请求到多个处理程序，并获取所有处理程序的响应。

```
public async Task<IActionResult> Get()
{
    var request = new MyRequest { Message = "Hello, MediatR!" };
    var responses = await _medi

ator.Send(request);

    return Ok(responses);
}
```

现在，你将获得一个包含所有处理程序响应的列表，这在某些场景下非常有用。

我们详细介绍了 MediatR 的基础功能和高级应用，包括请求和处理程序的创建、注册和使用，中介者管道的使用，异常处理和多个处理程序的应用。MediatR 是一个非常强大和灵活的库，它可以帮助你更好地组织和解耦你的代码，提高代码的可维护性和可扩展性。