ASP.NET Core Web 框架架构设计调研

StardustDL[1)]

[1)](南京大学计算机科学与技术系)

摘 要 随着 Web 开发的火热流行，众多 Web 开发框架层出不穷，在.NET 体系内，ASP.NET Core 是一个新的 Web 开发平台，使用 C# 语言编写，提供了一个可复用和可定制的请求处理管道。本文调研了 ASP.NET Core 这一框架的核心思路与架构设计，从而展示其如何满足高复用性，高定制性的要求，进而适用于众多应用场景。

关键词 网站开发框架，.NET

中图法分类号 TP391 DOI 号 10.11897/SP.J.1016.01.2022.00001

ASP.NET Core Web Framework Architecture Design Study

StardustDL[1)]

[1)](Department of Computer Science and Technology, Nanjing University)

Abstract As Web development has become more and more popular, many Web development frameworks have been developed. In the .NET community, ASP.NET Core is a new Web development platform written in C# that provides a reusable and customizable request processing pipeline. This paper investigates the core idea of ASP.NET Core framework and its architectural design, showing how it can meet the requirements of high reusability and customizability for many application scenarios.

Key words web framework; .NET

1 引言

随着 Web 开发的火热流行，众多 Web 开发框架层出不穷，如面向 Java 的 Spring 与 Spring-boot 框架，面向 Python 的 Django 和 Flask 框架，面向 Ruby 的 Ruby on Rails 框架，面向 JavaScript 的 Express 框架等。在.NET 体系内，ASP.NET Core 是一个新的 Web 开发平台，使用 C# 语言编写，提供了一个可复用和可定制的请求处理管道，如微软在 ASP.NET Core 上构建了模型-视图-控制器（MVC），Razor，Blazor，SignalR，GRPC，Orleans 等多种 Web 框架，以及第三方的 GraphQL 服务器，YARP 代理等框架。本文调研了 ASP.NET Core 这一框架的核心思路与架构设计，从而展示其如何满足适用于众多应用场景的高复用性，高定制性的要求。

在依赖注入等基础框架之上，ASP.NET Core 基于.NET Core 的服务承载系统实现了 Web 服务器，使用基于管道和中间件的设计处理请求，并提供了灵活的配置与日志系统。众多请求处理需求通过管道中间件完成，如路由，异常，缓存，会话，认证授权，本地化等。本文调

收稿日期：2022-07-03；修改日期：2022-07-03 StardustDL，计算机科学与技术硕士学位在读

研了 ASP.NET Core 中的重要系统设计，包括依赖注入，抽象文件系统，配置系统，日志系统，以及其处理请求的核心设计：管道与中间件架构。

2 依赖注入

ASP.NET Core 应用的运行中依赖于各种组件提供服务。依赖注入框架统一了这些服务的注册和访问模式，是整个框架的基础。依赖注入实现了控制反转（Inverse of Control）这一设计思想。在传统编程中，代码调用可重用的库来处理通用任务，但在 IoC 中，代码从一个通用框架中接收控制流，是框架调用了自定义或特定任务的代码。

ASP.NET 使用一个服务提供容器 ServiceCollection 注册服务。并提供了三类服务生命周期，包括临时 Transient，局部 Scoped，单例（全局）Singleton。注册服务实质上是在服务集合中添加了一个 ServiceDescriptor，其包含了如何判断此服务是否是所需服务（如根据服务类型，生命周期，服务名称等信息），以及如何创建此服务（如使用已创建的服务实例，调用函数创建服务）。

在服务注册完成后，ASP.NET Core 会根据 ServiceCollection 创建 ServiceProvider 来提供服务。具体来说，ServiceProvider 提供了诸如 GetService 之类的方法用于获取指定服务，以及 CreateScope 方法用于创建局部服务域，可以通过嵌套局部服务域实现更细粒度的生命周期。ServiceProvider 根据 ServiceCollection 中定义的 ServiceDescriptor 创建服务并管理服务生命周期。创建服务过程中，ServiceProvider 会通过构造函数，属性，方法三种途径进行依赖注入。

应用启动时，ASP.NET Core 会扫描程序集，按照约定和特性标注自动注册服务，开发人员也可手动定制服务注册过程，以注册额外服务。应用运行时，ASP.NET Core 会按照约定创建所需服务，处理到来的请求。

3 抽象文件系统

ASP.NET Core 使用 IFileProvider 实现了具有层次化目录结构的抽象文件系统，其对应的具体文件可能来自操作系统文件系统，数据库，或远程网络，其内容在读取时动态生成，并引入 ChangeToken 响应文件系统更改。通过这一设计，ASP.NET Core 拓展了文件访问的实现方式，不再依赖于操作系统的真实文件系统，使得读取程序集中的嵌入文件，访问来自网络的远程目录文件都成为可能，这为众多文件访问场景（如配置文件读取）带来了良好的扩展性。

4 配置系统

ASP.NET Core 使用了配置系统为应用提供定制化的配置。配置系统包含三个核心对象 ConfigurationSource（Provider），ConfigurationBuilder，Configuration，分别表示配置数据源，配

置构造器，和实际的配置信息。配置构造器是连接配置数据源和实际配置信息的桥梁。配置构造器首先按优先级顺序读取配置源中的数据，这可能来自配置文件，对象实例，环境变量，程序参数等，然后构造器将众多源数据解析成统一的结构化层次数据，从而生成最终的配置信息。

进一步地，ASP.NET Core 使用了 Options 模式实现强类型的配置访问，即将配置信息绑定为一个 POCO 的 Options 对象，并以依赖注入的方式来使用它。Options 模式提供了生命周期和变更监控，从而实现动态配置访问。

5 日志系统

ASP.NET Core 使用了统一日志编程模式，对众多的诊断日志框架（如微软的 Debugger，TraceSource，EventSource，DiagnosticSource，和第三方的 Log4Net，NLog，Serilog 等）进行整合。这一日志编程模型主要通过三个核心对象 Logger，LoggerFactory，LoggerProvider 实现。应用程序通过 LoggerFactory 创建指定的 Logger 对象（如某一日志类别）来记录多种级别（Trace，Debug，Information，Warning，Error，Critical）的日志，LoggerProvider 则在过滤所收到的日志后完成针对相应渠道的日志输出（如控制台，Debugger，数据库，远程日志系统，或第三方日志框架）。

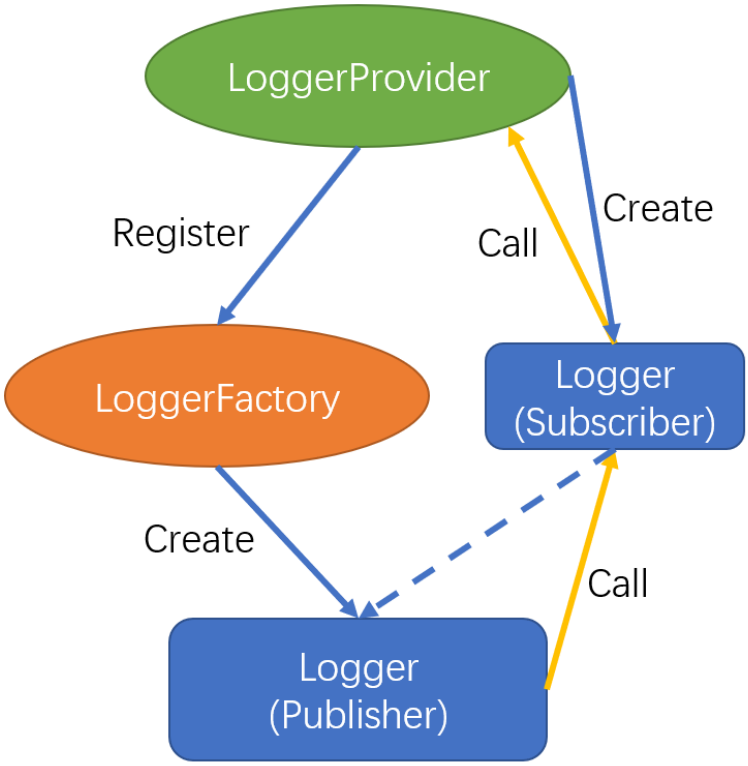

图 1 日志模型实现结构

这一日志模型的实现较为复杂，如图 1。首先，一系列 LoggerProvider 被注册到 Logger-

Factory 中，然后应用调用 LoggerFactory 创建 Logger 对象，在 LoggerFactory 实现内部，这一 Logger 对象由注册到 LoggerFactory 的一系列 LoggerProvider 创建的 Logger 对象组合而成，其将应用创建的日志分发到各个 LoggerProvider 中。故在这一过程中，有两类 Logger 对象，一是 LoggerFactory 创建的 Logger 对象，其被应用访问，是日志的发布者（生产者），二是 LoggerProvider 创建的 Logger 对象，其被 LoggerFactory 创建的 Logger 对象访问，是日志的订阅者（消费者）。LoggerFactory 创建的 Logger 连接了应用和具体的日志系统。

6 管道与中间件

HTTP 协议的设计决定了任何一个 Web 应用的工作模式都是监听、接收并处理 HTTP 请求，并且最终对请求予以响应。这一过程是管道式设计的典型应用场景：根据具体应用需求构建一个管道，接收的 HTTP 请求流入这个管道，组成管道的各个部分依次对其做相应的处理。

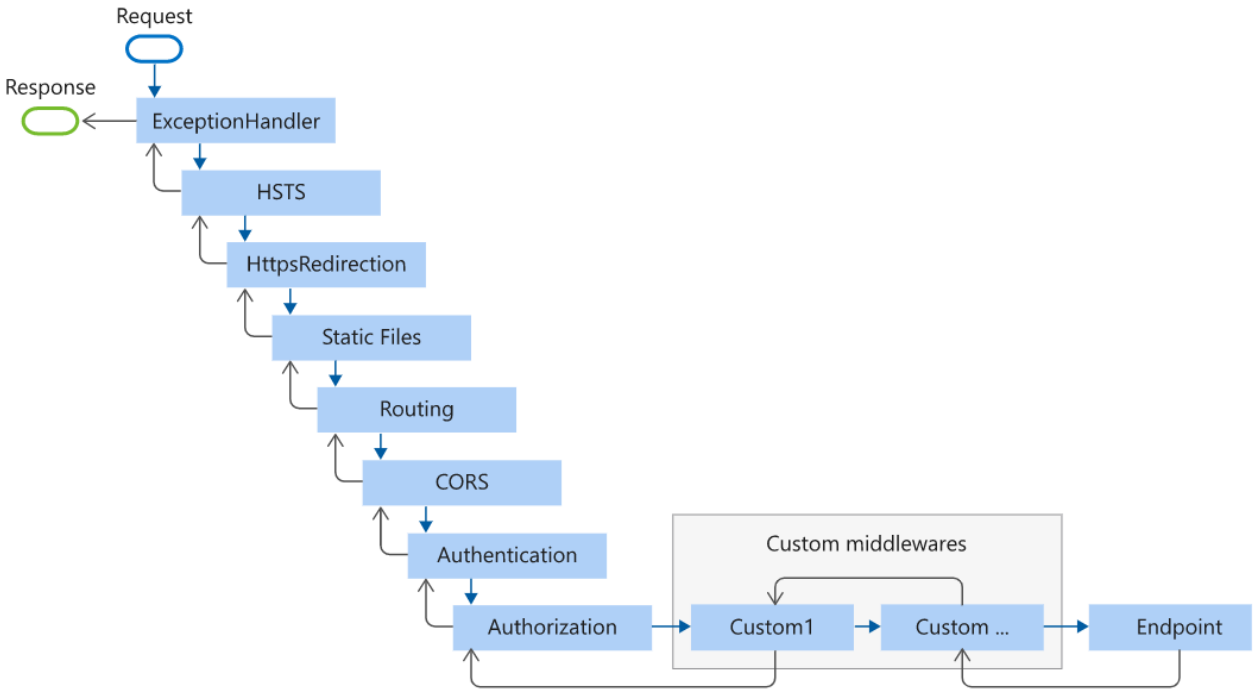

图 2 管道与中间件示例

具体来说，如图 2，应用启动后，定制化的请求处理管道便被构建出来，HTTP 请求一旦到达，服务器首先将其标准化，然后分发给管道后续的节点，即中间件。每个中间件具有各自独立的功能，如路由，认证，授权等。在管道中流动的对象为 HttpContext，包含了请求对象和响应对象，每个中间件可通过此上下文访问请求和修改响应。中间件的依次处理模式构成了一个委托链，每个中间件决定接下来是传递给下一个中间件还是拦截后续处理过程并直接回到上一中间件。