Design Patterns: Chain of Responsibility Pattern in TypeScript

Welcome to the Design Patterns in TypeScript series, which introduces some useful design patterns in web development using TypeScript.

Previous articles are as follows:

Strategy Pattern in TypeScript

Chain of Responsibility Pattern in TypeScript

Observer Pattern in TypeScript

Template Method Pattern in TypeScript

Adapter Pattern in TypeScript

Factory Method Pattern in TypeScript

Abstract Factory Pattern in TypeScript

Design patterns are very important for web developers and we can write better code by mastering them. In this article, I will use TypeScript to introduce the Chain of Responsibility Pattern.

The chain of responsibility pattern is a way to avoid coupling between the sender and receiver of a request by giving multiple objects the opportunity to handle the request. In the chain of responsibility pattern, many objects are connected by a reference from each object to its next object to form a chain. Requests are passed along the chain until one of the objects in the chain decides to handle the request.

Different positions in the company have different responsibilities and authorities. Take our company’s off process as an example, when I ask for a day off, it only needs to be approved by the team leader, and there is no need to transfer it to the supervisor and director. If a link in the chain of responsibility cannot process the current request, if there is a next link, the request will be forwarded to the next link for processing.

In the process of software development, for the chain of responsibility, a common application scenario is middleware. Let’s take a look at how to use the chain of responsibility to process requests.

In order to better understand the following code, let’s first look at the corresponding UML diagram:

In the above figure, we define a Handler interface. The following two methods are defined in this interface:

use(h: Handler): Handler => Used to register handler (middleware)

get(url: string, callback: (data: any) => void): void => Register a get request handler

Handler interface

Then we define an AbstractHandler abstract class, which encapsulates the processing logic of the chain of responsibility. That is, combine different handlers to form a reference chain.

AbstractHandler abstract class

Based on the AbstractHandler abstract class, we define AuthMiddleware and LoggerMidddleware respectively. The AuthMiddleware middleware is used to handle user authentication and the LoggerMidddleware middleware is used to output request logs.

AuthMiddleware class

LoggerMiddleware class

With the AuthMiddleware and LoggerMidddleware middleware, let’s define a Route class to register these middlewares.

Route class

After defining the Route class, we can use it and register middlewares in the following way:

When you successfully run the above code, the corresponding output is shown in the following figure:

Finally, let’s summarize the usage scenarios of the chain of responsibility pattern:

Want to submit a request to one of multiple objects without explicitly specifying the recipient.

There are multiple objects that can handle a request, and which object handles the request is automatically determined at runtime, and the client only needs to submit the request to the chain.

If you have any questions, please feel free to leave me a message. I will continue to introduce other patterns later, if you are interested, you can follow me on Medium or Twitter.