第 3 章

Web 服务系统

Web 框架开发的业务系统，尽管所在行业、服务内容、规模等存在差异，但其共同点都是基于 Web 服务提供的 HTTP 接口来完成信息交互，因此，Web 服务是 Web 框架最为基础的特性，是实现所有业务功能的前提。

本章将讲述框架实现 Web 服务系统的开发过程，内容涵盖了 Web 服务系统的 3 个重要的层次：Web 服务框架、中间件机制及路由系统，如图 3-1 所示。

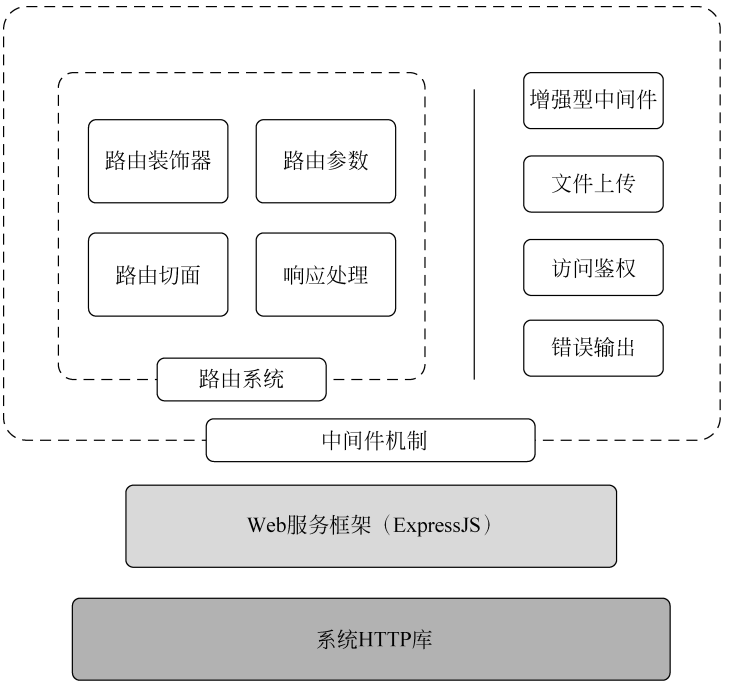

图 3-1　Web 服务系统

Web 服务处在系统底层 HTTP 通信逻辑和上层业务代码之间，它需要对原始的 HTTP 协议数据进行抽象和转换，以便于上层代码的使用，所以通常在开发中还需要引入 Web 服务框架以进行处理。本章引入的服务框架是 Node. js 开发领域较为成熟的 ExpressJS 框架。

注意：类似 ExpressJS 的 Web 服务框架和本书开发的 TypeSpeed 框架之间的区别在 3.1.1 节有详细的讲述。

和 ExpressJS 框架一起引入的还有中间件机制，中间件机制是一种设计模式，它抽象了 Web 服务的请求-响应链的处理过程，提供了编写 Web 服务功能的代码组织方法。

在中间件机制的基础上，框架实现路由装饰器、切面装饰器、参数装饰器等开发者常用的路由系统，同时集成了 Cookie/Session，以及传输压缩、文件上传、JWT 鉴权等多种增强型的中间件，供开发者日常开发使用。

3.1 集成 Web 服务框架

本章将采用第 2 章开发的对象管理机制实现 Web 服务的扩展功能，集成 ExpressJS 框架以支持框架的底层服务。

3.1.1 ExpressJS

ExpressJS 是基于 Node.js 的 Web 服务框架，在 MIT 许可证下作为自由及开放源代码软件发行。ExpressJS 被设计用来开发 Web 应用和 API。目前，它是基于 Node.js 的服务器端框架的事实标准之一。

ExpressJS 通常也被称作 Web 框架。它和本书开发的 TypeSpeed 框架之间的区别如图 3-2 所示。

图 3-2 展示了 Web 框架的层级结构。从底部开始，可以看到底层是 Node.js 的 HTTP 模块，它的主要作用是将操作系统底层的 HTTP 通信协议内容封装成 JavaScript 代码库的形式，供进行底层协议级别的开发。这个层次需要关注的开发细节最多，因为它只提供了协议层面的基础逻辑，在协议之上的部分都需要开发者自行编码实现。例如开发路由功能，需要开发者从 HTTP 协议库的 Request 对象内提取用户访问地址，然后匹配路由规则来指向对应的页面逻辑，中间还要处理诸如协议出错、数据缓冲区转换等烦琐的细节。

在底层 HTTP 模块上开发是较为烦琐的，并且容易忽略一些细节而导致后期系统运行不稳定，所以通常需要对 HTTP 模块进行封装，提供更为稳定、成熟的框架给上层业务系统使用，这些框架可称为 Web 服务框架。常见的 Web 服务框架有 ExpressJS、Koa 等。

Web 服务框架重新定义了 HTTP 模块的使用方式，例如将 HTTP 请求抽象为请求-响应链的中间件机制，提供简单的方法来处理的 HTTP 请求数据，提供各种丰富的输出响应方法等。在这个层次上，需要使用者自行实现的底层功能相对较少，不过由于 Web 服务框架的主要工作是包装底层 HTTP 协议逻辑，简化对请求和响应的操作，缺少对上层业务系统的支持，所以 Web 服务框架只适合在页面较少或者逻辑较简单的 Web 应用中直接使用，更多的场景是为上层 Web 业务框架提供中间件机制的支持。

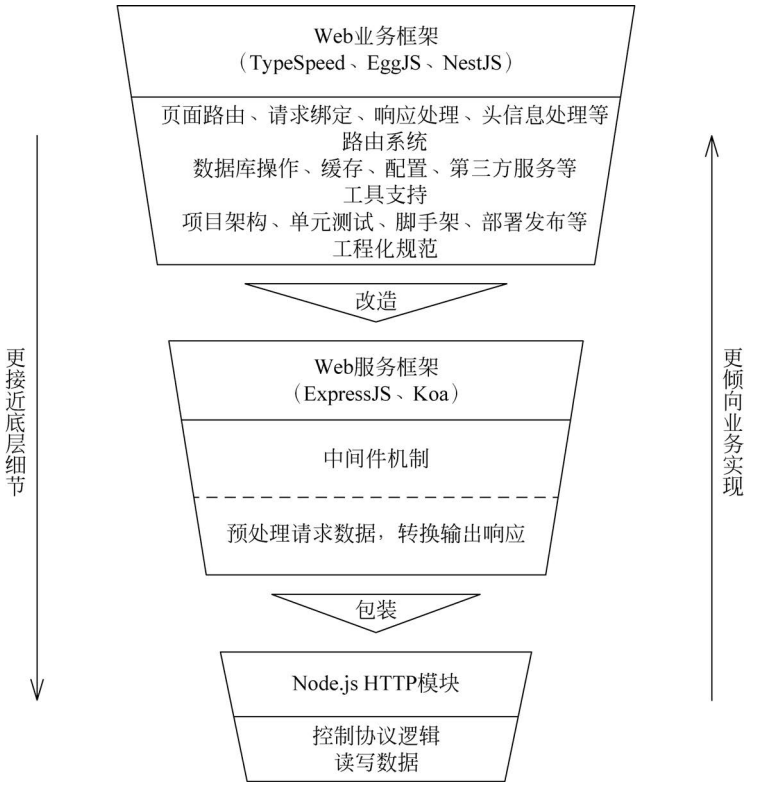

图 3-2　Web 框架的层级结构

Web 业务框架，即本书开发的 TypeSpeed 框架，此外较为知名的还有 EggJS、NestJS 等，这些框架通常简称为 Web 框架。它们基于 Web 服务框架，遵循中间件机制对 HTTP 请求进行再次封装，Web 框架具有以下 3 个特点：

（1）提供功能易用的路由系统，例如 NestJS 和 TypeSpeed 均提供了路由装饰器，使用简单，符合 TypeScript 的开发习惯。

（2）提供支持业务开发的工具支持，例如 TypeSpeed 提供数据库操作、认证鉴权、消息队列等开箱即用的功能，提升业务系统的开发效率。

（3）提供工程化的规范和支持，Web 框架对团队开发和协作提供了创建标准化项目结构的脚手架、单元测试、自动化文档等功能的支持，使其能适用于大规模项目。

3.1.2　中间件机制

ExpressJS 网站对中间件(Middleware)的定义是：中间件函数是在应用程序请求-响应循环中可以访问请求对象、响应对象及下一个中间件的函数。ExpressJS 网站给出了中间件的示例，代码如下：

```
const express = require('express')
const app = express()
```

```
app.get('/', (req, res, next) => {
  res.send('Hello World!')
})

app.listen(3000)
```

常量 app 是 Express 对象,它是当前 Web 程序的实例。它的 get()方法用于接收 HTTP 的 GET 请求,其第 1 个参数指的是当前请求的 URL 路径,而第 2 个参数的匿名函数将处理请求并返回响应信息。这个匿名函数就是中间件,它的 3 个参数 req、res、next 分别对应请求对象、响应对象、下一个中间件实例。

请求对象是对 HTTP 请求数据的封装,在请求数据中比较重要的有请求头信息(Headers)、请求体信息(RequestBody)和客户端 IP、UA 等特征信息。

响应对象对应的是浏览器将要接收的信息,通常情况下响应的是文本内容,如 JSON 数据或 HTML 页面,但有些场景服务器端也会返回特殊的状态码,指示浏览器的下一步行为,例如 302 表示跳转到另一个页面、403 表示权限不足、502 表示服务器端出错等,所以响应对象中比较值得关注的是响应类型和状态码。

下一个中间件实例是指向中间件链条的下一个中间件。用于显示页面的中间件通常不需要调用 next,这代表当前中间件是链条的最后一个中间件,所以它会调用响应对象的输出方法 res.send()或者 res.end()将页面返回浏览器。那么,什么时候需要调用 next 呢?

这就要说到中间件的两个作用:响应请求和对请求进行预处理。

响应请求很好理解,就是将正确的页面返回浏览器显示,而在响应请求之前,可能需要进行一些预处理工作,例如典型的认证鉴权,网站的认证鉴权逻辑要求在用户请求页面之前,先对用户请求携带的认证信息进行验证,只有验证通过后才能进入页面,所以预处理的中间件通常不会直接参与页面显示相关的逻辑,其作用主要是预先对请求数据做一些鉴别或者转换(例如文件上传)等工作。

负责预处理的中间件在完成处理后,需要调用 next()将请求传递给下一个中间件。如果在预处理的工作中出现异常情况,例如用户登录信息不匹配,则可以调用 next(err)来传递异常信息。如果调用输入参数的 next(err),则会跳过下一个正常的中间件,而将直接到达能够处理异常的中间件,以此来给用户返回异常信息。

从以上 3 个参数可以看出,中间件本身具备了处理输入、输出及操纵处理方向的能力,所以可以将其视为 Web 服务处理 HTTP 请求的最小单元。

中间件机制是将这些最小的处理单元串联执行,一个中间件通过 next 传递到下一个中间件,形成一条处理从请求到响应的功能链条,如图 3-3 所示。

图 3-3 展示了一个 HTTP 请求达到 Web 服务后历经的各个环节,例如请求的头信息如果内容包含了文件上传域,则会在上传中间件里对请求体数据进行转换,将上传文件的二进制内容抽离出来并存储成临时文件,并且临时文件信息会附加到 req 请求对象上,继续传递到下一个鉴权中间件,鉴权中间件会读取当前请求头的用户信息进行验证,验证通过后进

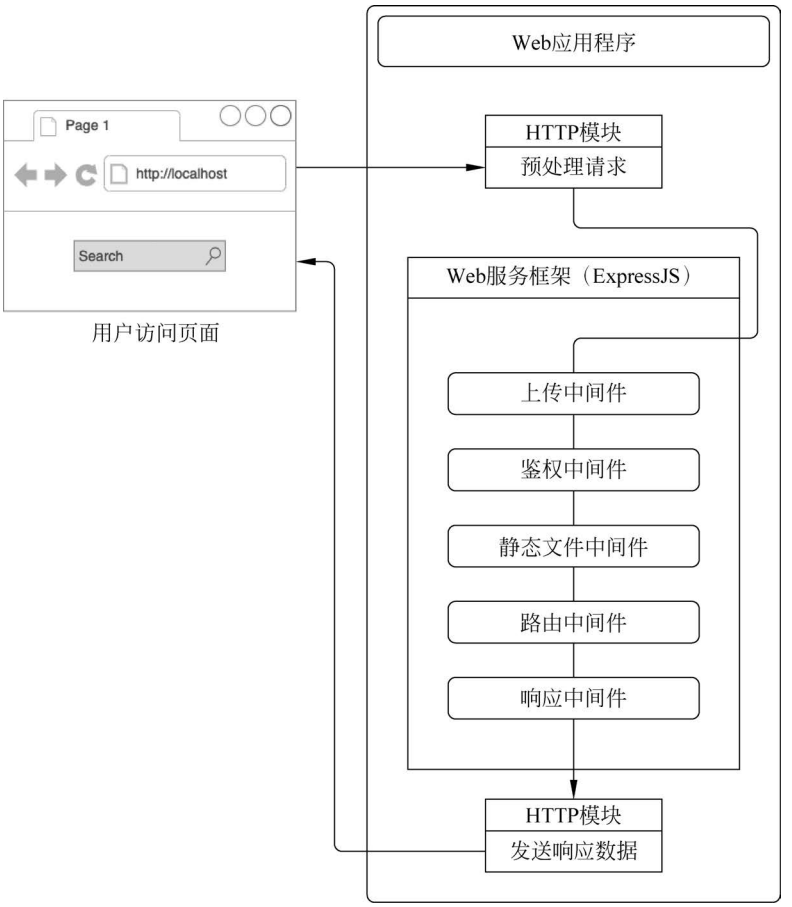

图 3-3　中间件的串联执行

入页面中间件。

整个链条上最重要的是路由中间件，路由中间件通常有很多，分别对应各种 HTTP 请求方法和不同的路径。路由中间件会根据 HTTP 请求路径和方法进行匹配并调用对应的路由中间件进行处理。

路由中间件是页面逻辑代码的所在中间件，这是 Web 应用主要的业务逻辑所在。匹配的路由中间件将对请求数据进行各种业务逻辑的处理，如数据库操作等。最后它会将处理结果传递到响应中间件，响应中间件从 res 响应对象里取得结果的格式和数据，依据格式来发送响应头信息，并把结果发送回浏览器，结束这次处理流程。

从以上过程可以看到单个中间件只完成特定的处理，从而提高了单个中间件的代码可重用性和稳定性，而通过恰当的逻辑将它们串联起来。中间件的串联执行机制，十分恰当地抽象了 Web 请求-响应的本质，让开发者能够更清晰地组织每个请求对应的功能，为整体服务的可扩展性提供良好的基础。

中间件的串联执行机制，同时也是一种名为责任链的设计模式。责任链模式(Chain of

Responsibility)是一种处理请求的模式,它让多个处理单元都有机会处理该请求,直到其中某个处理单元成功为止。责任链模式把多个处理单元串成链,然后让请求在链上传递。责任链的优点正是中间件机制的优势:高可用的处理单元和灵活的组织逻辑。

3.1.3 应用程序入口

在将 ExpressJS 集成到 TypeSpeed 框架之前,需要先解决应用程序入口问题,也就是框架乃至整个 Web 应用程序应该在哪里开始被执行?

应用程序最早被执行的位置被称为应用程序入口。应用程序入口是框架设计的要点之一。

举个例子,Java 语言的程序入口是一个函数签名为 public static void main(String[] args)的方法,通常简称为 main 方法,main 方法是开发者代码的起点。

Node.js 应用程序相对 Java 更灵活,Node.js 可以执行任何 js/ts 文件以作为起点,但依据 Node.js 的包规范,项目的 package.json 配置文件里 main 配置项指定的文件将作为整个程序的入口文件。

Web 框架的应用程序入口,在执行顺序上面,它必须先让框架的核心功能准备就绪,然后开始执行开发者的代码,所以应用程序入口的执行时机是在框架系统的全部逻辑被 import 之后,而在开发者业务代码执行之前。

TypeSpeed 框架的应用程序入口设计如下:

(1) 遵循包规范,入口文件必须是 package.json 的 main 配置项,约定文件名是 main.ts。

(2) 在 main.ts 文件内,被@app 装饰器装饰的类是启动类,而启动类的 main()方法就是应用程序的入口,也就是开发者的第 1 行代码所在。

该设计采用的装饰器为主和约定优于配置的原则。因为框架的大部分逻辑基于装饰器实现,所以程序入口同样不例外。

设计中只采用了类装饰器,而不是类装饰器加方法装饰器的组合。因为在测试中发现,仅需要用类装饰器来标注启动类,然后约定启动方法名称是 main()方法即可定位到初始代码的位置,这样使用起来更简单。

从第 2 章我们了解到,装饰器会在执行当前文件或是被 import 时执行,所以@app 类装饰器将执行框架自身代码的位置,它会在 main.ts 文件执行时被调用。当框架代码就绪之后,再转向执行启动类的 main()方法。@app 类装饰器的代码如下:

```
//chapter03/01 - web - server/src/speed.ts

function app < T extends { new(...args: any[]): {} }>(constructor: T) {
    const srcDir = process.cwd() + "/src";
    const srcFiles = walkSync(srcDir, { globs: ['**/*.ts'] });

    const testDir = process.cwd() + "/test";
    const testFiles = walkSync(testDir, { globs: ['**/*.ts'] });
    //异步转同步执行
```

```
    (async function () {
        try {
            //载入框架的 TS 文件,执行其中的装饰器
            for (let p of srcFiles) {
                let moduleName = p.replace(".d.ts", "").replace(".ts", "");
                await import(srcDir + "/" + moduleName);
            }
            //载入开发者的 TS 文件,执行装饰器并覆盖
            for (let p of testFiles) {
                let moduleName = p.replace(".d.ts", "").replace(".ts", "");
                await import(testDir + "/" + moduleName);
            }
        } catch (err) {
            console.error(err);
        }
        //开启用户程序
        log("main start")
        const main = new constructor();
        main["main"]();
    }());
}
```

@app 类装饰器是泛型函数,使用泛型可以通过继承于 constructor 构造函数来取得当前启动类,使用 new 实例化当前启动类,进而调用启动类的 main()方法。

@app 分为 3 部分,在函数的开始位置定义了 4 个常量:

(1) srcDir 是框架源码所在目录,用 process.cwd()取得命令执行的位置,该位置拼接上 src 即框架所在目录。

(2) srcFiles 是数组,它使用 walkSync 库来搜索 srcDir 目录下所有目录里包含 ts 后缀的文件名。这是为了后面扫描 import 所有框架文件做准备。

(3) testDir 是开发者代码所在目录,和 srcDir 同样是拼接而成的,只是它指向 test 目录。

(4) testFiles 同样是数组,包含开发者目录的所有 ts 文件名。

接下来就是循环 import 载入 srcFiles 文件和 testFiles 文件。这里采用了执行异步代码的写法,代码如下:

```
(async function () {
  ...
  await import(path);
  ...
}());
```

因为 import()载入函数返回的是一个 Promise 对象,实际上这意味着它是一个异步函数,所以必须在它的前面加上 await 进行调用,再在外面包装一层 async function 进行调用,确保这些异步代码按顺序来执行。

另外,注意 import 的文件名都先经过 replace()函数进行过滤,这是因为后缀为 ts 的文件在编译后会变成后缀为 js 的文件,而 import 可以忽略文件后缀,所以这里将后缀去除后

再载入。

在@app 的最后部分是 new 实例化启动类,并调用启动类的 main()方法。

注意:@app 装饰器的代码逻辑可以参照 2.2.7 节关于入口文件机制的讲解,以加深理解。

现在看如何使用@app 类装饰器,也就是应用程序入口的具体使用,代码如下:

```
//chapter03/01 - web - server/test/main.ts

import ServerFactory from "../src/factory/server - factory.class";
import { app, log, autoware } from "../src/speed";

@app
class Main {

    @autoware
    public server : ServerFactory;

    public main(){
        this.server.start(8080);
        log('start application');
    }
}
```

@app 装饰的 Main 类是启动类,它的 main()方法就是应用程序的入口。该入口程序执行 this.server.start()方法以启动 Web 服务,然后输出应用启动的日志。


13min

3.1.4 集成 ExpressJS

集成 ExpressJS 共分成两个步骤,第 1 步,集成 ExpressJS 的启动代码;第 2 步,对它的路由系统进行改造。这里先完成第 1 步,即集成启动代码。

遵循第 2 章的对象管理机制,首先在 factory 目录建立 Web 服务的父类,代码如下:

```
//chapter03/01 - web - server/src/factory/server - factory.class.ts

export default abstract class ServerFactory {
    protected middlewareList: Array<any> = [];
    public abstract setMiddleware(middleware: any);
    public abstract start(port: number);
}
```

ServerFactory 定义了框架集成 Web 服务的两个抽象方法:

(1) setMiddleware(middleware: any)方法用于增加中间件,新增的中间件存储在 middlewareList 数组里。setMiddleware()方法方便开发者将其自定义的中间件添加到中间件链条里,以方便进行一些额外的操作。

（2）start()方法用于启动 Web 服务，它将被开发者的代码所调用。它的参数 port 是启动 Web 服务的端口号。

由于对象管理拥有覆盖扩展的特性，开发者如果希望用另一个 Web 服务来替代 ExpressJS 框架，则可以在不修改框架源码的情况下，使用@bean 装饰器来创建新的 Web 服务以达到更换默认 Web 服务的目标。

接下来是默认 Web 服务的实现子类，代码如下：

```
//chapter03/01 - web - server/src/default/express - server.class.ts

export default class ExpressServer extends ServerFactory {
    //提供 Web 服务对象
    @bean
    public getSever(): ServerFactory {
        return new ExpressServer();
    }
    //设置中间件
    public setMiddleware(middleware: any) {
        this.middlewareList.push(middleware);
    }
    //启动 Web 服务
    public start(port: number) {
        const app: express.Application = express();
        this.middlewareList.forEach(middleware => {
            app.use(middleware);
        });
        app.listen(port, () => {
            log("server start at port: " + port);
        });
    }
}
```

ExpressServer 类继承于 ServerFactory，并重写了 setMiddleware()和 start()方法。

ExpressServer 的 setMiddleware()方法简单地将传入的中间件参数 push()到 middlewareList 数组里，后期如果有需要，则可以在 setMiddleware()方法中增加一些对加入的中间件进行改造的逻辑。

start()方法是集成 ExpressJS 的第 1 步，它先实例化 ExpressJS 的 app 对象，接着循环 middlewareList 数组，用 app.use()方法把这些中间件设置到 ExpressJS 实例里。这些中间件就会依据被设置的先后顺序来执行。

start()方法最后调用 app.listen()方法来启动 ExpressJS 的 Web 服务，并且在第 2 个参数的匿名函数的内部输出服务启动的日志，如图 3-4 所示。

这样第 1 步（集成）便完成，接下来可以测试它的效果，命令如下：

```
ts - node test/main.ts

decorator bean, the return Type is: ServerFactory
```

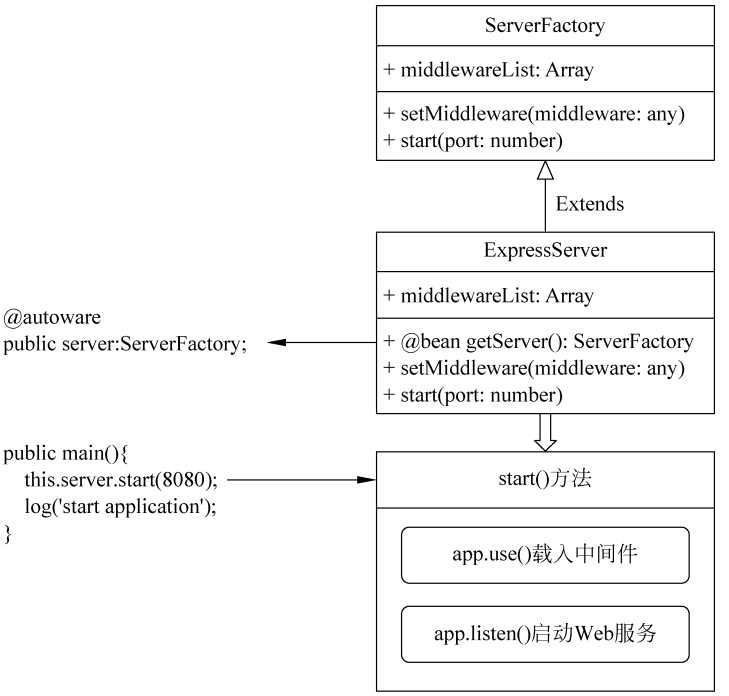

图 3-4　集成 ExpressJS

```
Map(1) { 'ServerFactory' => [Function: getSever] }
decorator bean, the return Type is: LogFactory
Map(2) {
  'ServerFactory' => [Function: getSever],
  'LogFactory' => [Function: createLog]
}
console.log : decorator bean, the return Type is: LogFactory
Map(2) {
  'ServerFactory' => [Function: getSever],
  'LogFactory' => [Function: createLog]
}
[LOG] 2023-10-25 18:10:03 speed.ts:34 (onClass) decorator onClass: TestLog
[LOG] 2023-10-25 18:10:03 speed.ts:27 () main start
[LOG] 2023-10-25 18:10:03 main.ts:13 (Main.main) start application
[LOG] 2023-10-25 18:10:03 express-server.class.ts:19 (Server.<anonymous>) server start
at port: 8080
```

命令先是输出在 BeanFactory 类内部检查当前对象工厂的调试代码。之后输出 start() 方法的启动日志，这样 Web 服务就启动完成了。

打开浏览器访问 http://localhost:8080/，页面显示 Cannot Get /即证明了服务已经正常运行，该提示是因为 Web 服务还没有设置路由中间件，所以无法进行路由转向。

3.1.5 小结

至此，框架已经初步集成了 ExpressJS 的 Web 服务，并成功启动。

本节介绍了 Web 服务框架的概念，而 ExpressJS 是 Web 服务框架里较成熟的框架，也是 TypeSpeed 框架用于支持其 Web 服务的选择。ExpressJS 的优势在于简化了底层 HTTP 模块的烦琐细节及提供了中间件机制。

中间件机制是一种重要的 Web 服务的架构模式。中间件机制十分恰当地抽象了 Web 请求-响应的逻辑，给开发具有可重用性和可扩展性的服务器端代码提供了很清晰的代码组织结构。

在相当一部分知名的 Web 框架里采用中间件机制作为主要的开发架构，举例如下。

(1) Node.js 的 Koa 框架。值得一提的是，Koa 和 ExpressJS 均来自同一作者。

(2) Go 的 Gin、Fiber、Echo 等框架。

(3) PHP 的 Laravel 框架。

(4) Python 的 Flask 框架。

(5) Java 的 Play 框架。

理解中间件机制对于我们广泛地认识各种框架的核心思想很有帮助，后续的章节将会继续介绍各种常用中间件及其使用方法。

3.2 路由装饰器

在 3.1.4 节提到集成 ExpressJS 有两个步骤，即集成启动代码和改造路由系统。第 1 步通过对象管理机制，框架已经成功集成了 ExpressJS 的启动代码并成功运行。本章将继续讲解第 2 步，即关于路由系统的改造。

路由是 Web 服务的主要功能之一，它的作用是将浏览器请求的路径导向对应的页面，它是 Web 应用程序能够以不同的路径提供各种功能接口的基础。

3.2.1 简单的路由实现

首先来简单了解一个 ExpressJS 的路由写法，在 3.1.4 节的 ExpressJS 启动代码的 start()函数里，加入对访问路径/hello 进行处理的路由中间件，代码如下：

```
//chapter03/01 - web - server/src/default/express - server.class.ts

export default class ExpressServer extends ServerFactory {
    @bean
    public getSever(): ServerFactory {
        return new ExpressServer();
    }
    public setMiddleware(middleware: any) {
```

```
            this.middlewareList.push(middleware);
        }
        public start(port: number) {
            const app: express.Application = express();
            this.middlewareList.forEach(middleware => {
                app.use(middleware);
            });
        //在这里加入指向路径/hello的路由中间件
        app.get("/hello", function(req, res){
                res.send("hello");
        });
            app.listen(port, () => {
                log("server start at port: " + port);
            });
        }
    }
```

再次通过 ts-node 命令启动程序，打开浏览器访问 http://localhost:8080/hello，即可看到如图 3-5 所示的内容。

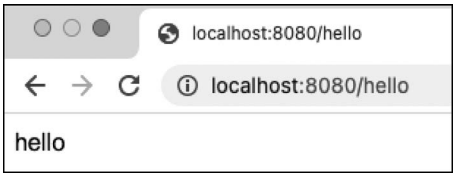

图 3-5　Web 路由显示页面结果

现在来单独观察这段 ExpressJS 设置路由中间件的代码，代码如下：

```
app.get("/hello", function(req, res){
     res.send("hello");
});
```

代码使用 ExpressJS 实例的 get()方法，get()方法为 HTTP 方法之一。HTTP 方法还有 all、post、put、delete、options、head 等 20 多个，详情可参考 ExpressJS 框架的 IRoute 接口的定义。

从名称可以看出，这些 HTTP 方法大体对应了 HTTP 协议的请求方法(Request Method)。值得注意的是，all()方法并不对应 HTTP 请求方法，它指代的是全部的请求方法。

HTTP 方法的作用是根据用户发起的请求类型来选择不同的处理方法，或者不处理。例如上述代码 app.get()只支持 GET 请求/hello，如果使用 POST 将请求发送到/hello，则将返回 404 状态码，表示找不到页面，如图 3-6 所示。

当然，如果将 app.get()改为 app.all()，则不管什么请求方法都能正常请求到/hello 页面，如图 3-7 所示。

图 3-6　使用 POST 请求/hello 返回 404，表示找不到页面

图 3-7　app.all()可以处理任意的请求方法

app.get()方法的第 1 个参数是请求路径，该参数支持 3 种匹配规则，表示任意匹配该路径的请求都将由其后的路由中间件进行处理。关于请求路径的匹配规范，将在 3.2.2 节详细讲解。

app.get()方法的第 2 个参数是路由中间件，是具体处理当前页面逻辑的代码所在。

路由中间件作为 HTTP 请求-响应链条的一环，同样有着 req、res、next 共 3 个参数，但通常 next 参数在路由中间件里很少用到，可参考 3.1.2 节的介绍，这里不再赘述。

请求对象 req，类型是 ExpressJS 的 Request，同时也继承于 HTTP 模块的 IncomingMessage，其作用是提供 HTTP 模块解析请求后的数据，这里是它的一些常用方法和变量：

(1) req.get()方法的别名为 req.header()，其作用是取得请求的头信息。例如 req.get('content-type')取得请求的内容类型，如 text 文本、json 内容、binary 二进制等。

(2) req.accepts()方法，其作用是取得请求里标明可接受的响应类型，在实际开发中可以将其理解成请求期待的响应类型，例如 req.accepts('json')表明浏览器期待收到 JSON 格式的响应数据，req.accepts('html, json')表明期待的类型是 HTML 或者 JSON。

(3) req.param()方法，其作用是取得路径的参数或者 URL 的 Query 部分的参数，是获取请求参数的主要方法，将在 3.4 节具体介绍。

(4) req.ip 变量，其作用是取得浏览器的 IP，在限制 IP 访问网站或根据 IP 显示不同内容的场景下非常有用。

(5) req.body 变量和 req.param()一样，属于请求参数的一种类型，将在 3.4 节具体介绍。

(6) req.cookies 变量，其作用是取得请求中附带的 Cookie 值，将在 3.6.4 节详细介绍。

(7) req.url 变量，其作用是取得请求的整个 URL 网址。

响应对象 res，类型是 Response，也继承于 HTTP 模块的 ServerResponse，其作用是转换响应数据并发回给浏览器，这里是它的一些常用方法：

(1) res.set()方法的别名为 res.header(),其作用是设置响应的头信息。

(2) res.cookie()方法,其作用是设置 Cookie 信息,将在 3.6.4 节详细介绍。

(3) res.type()方法,其作用是设置响应的 Content-Type 类型。例如 res.type('json')、res.type('png')。

(4) res.send()方法的别名为 res.end(),是发送响应内容的主要方法,它的参数可以是字符串、Buffer 对象、JSON 内容、HTML 内容等。在不调用 res.type()设定类型的情况下,send()方法会自动根据参数类型,自动设置对应的 Content-Type 类型。值得注意的是,正是因为 send()方法会自动设置类型等响应头信息,所以 send()方法不能重复调用,否则将出现响应头信息重复发送的错误。

(5) res.status()方法,其作用是设置响应的 HTTP 状态码。

(6) res.sendStatus()方法,其作用是设置响应 HTTP 状态码并发送响应,它等同于 status()加 send(),例如 res.sendStatus(200)相当于 res.status(200).send('OK')。

(7) res.json()方法,其作用是发送 JSON 格式的响应,等同于 res.type('json').send(内容)。

(8) res.download()方法,其作用是发送文件下载,参数是文件的路径,浏览器会接收到下载文件。

(9) res.sendFile()方法,其作用是发送文件响应,参数是文件的路径,和 res.download()的区别是 sendFile()方法不一定会启动浏览器的下载,例如发送图片将在浏览器直接显示图片而不是下载图片文件。

(10) res.redirect()方法,其作用是发送 302 状态码,使浏览器自动跳转到参数中设定的另一个 URL 网址。

(11) res.render()方法,其作用是显示经过模板引擎渲染后的页面,该方法需要先配置模板引擎中间件,将在 3.5 节详细介绍。


10min

3.2.2 路径功能详解

app.get()方法的第 1 个参数是请求路径,这里所说讲述的路径,指的是 URL 网址中域名和问号之间的部分。例如地址 https://localhost/about.html?source=search,路径是/about.html,而问号后面的部分被称为查询字符串(URL Query),并不是路径的一部分。

1. 路径匹配

路径是 HTTP 请求关联到路由中间件的关键。在 3.2.1 节的代码中演示了路径/hello 对应其中间件的例子,匹配像/hello 这样的路径,称为完整路径匹配。

ExpressJS 的请求路径共有以下 3 类匹配方法。

1) 完整路径匹配

请求路径和路径参数必须完全一致,见表 3-1。

表 3-1 完整路径匹配示例

代 码	匹配示例
app.get('/', (req, res) => {})	http://localhost/
app.get('/about', (req, res) => {})	http://localhost/about
app.get('/page.html', (req, res) => {})	http://localhost/page.html

2）字符串表达式匹配

使用?、+、*、( )等字符对路径简单地进行字符串匹配，字符串表达式跟正则表达式相似，但相对后者要简单一些，见表 3-2。

表 3-2 字符串表达式匹配示例

代 码	说 明	匹配示例
app.get('/ab?cd', (req, res) => {})	?问号表示前一字符是 0 个或 1 个	http://localhost/acd http://localhost/abcd
app.get('/ab+cd', (req, res) => {})	+加号表示前一字符是 1 个或多个	http://localhost/abcd http://localhost/abbcd http://localhost/abbbcd …
app.get('/ab*cd.html', (req, res) => {})	*星号表示任何个数的字母或数字	http://localhost/abcd.html http://localhost/abPAGEcd.html http://localhost/ab123cd.html …
app.get('/ab(cd)?e', (req, res) => {})	括号里的字符组合和?、+、*符号一起使用，表示字符组合的数量	http://localhost/abe http://localhost/abcde

3）正则表达式匹配

路径参数要求写成正则表达式，见表 3-3.

表 3-3 正则表达式匹配示例

代 码	说 明	匹配示例
app.get(/a/, (req, res) => {})	表达式/a/可以匹配任何包含 a 的字符串	http://localhost/abe http://localhost/newabe …
app.get(/.*fly$/, (req, res) => {})	表达式/.*fly$/可以匹配任何以 fly 开头或者结尾的字符串，但不匹配中间有 fly 的字符串	http://localhost/butterfly http://localhost/dragonfly …

完整路径匹配性能是最高的，因为它只需检查字符串相等的情况，而字符串表达式匹配使用的是 JavaScript 字符串的匹配功能，比起需要执行词法分析和递归匹配的正则表达式，性能也要高出一些。每次 HTTP 请求都会经过所有的路径匹配，随着系统规模逐渐扩大，性能上微小的差距也会不断放大，从而会对系统性能带来影响。

因此，前两者的匹配场景虽然没正则匹配那么丰富，但在实际应用时，应尽可能地优先

使用完整路径匹配和字符串表达式匹配。

2. 路径参数

路径参数指的是从 URL 网址中取得的参数值，通常 URL 网址包含参数值能使整个 URL 看起来更简洁。

ExpressJS 的路径参数可以用 req.params 对象获取。在匹配路径里，在参数前加入冒号即可将该参数识别成路径参数，见表 3-4。

表 3-4 路径参数识别示例

代 码	地址示例	识别参数
app.get('/item/:productId.html', (req, res) => { res.send(req.params) })	http://localhost/item/492056.html	req.params = { id: '492056' }
app.get('/:productId-:page.html', (req, res) => { res.send(req.params) })	http://localhost/32423-1.html	req.params = { productId: '32423', page: '1' }

路径参数是 Web 业务开发中比较重要的数据之一，是浏览器与服务器端交互的必要手段。在 3.4 节开发请求参数装饰器时将进一步介绍路径参数的使用。

3.2.3 开发路由装饰器

ExpressJS 的路由功能，由 HTTP 方法、路径匹配、对应的路由中间件等 3 部分组成，那么框架集成路由功能，也需要分别对应这 3 部分进行改造。

首先框架的路由功能采用装饰器来设计，以延续框架的编程风格，装饰器对应的是 HTTP 方法，这里主要设计最为常用的 GET、POST 和 ALL，其他的 HTTP 方法都可以使用 ALL 来承接。

路径参数和 HTTP 方法结合得比较紧密，因此将路径参数设计为 HTTP 方法装饰器的参数，而路由中间件则是被装饰器装饰的方法函数，如图 3-8 所示。

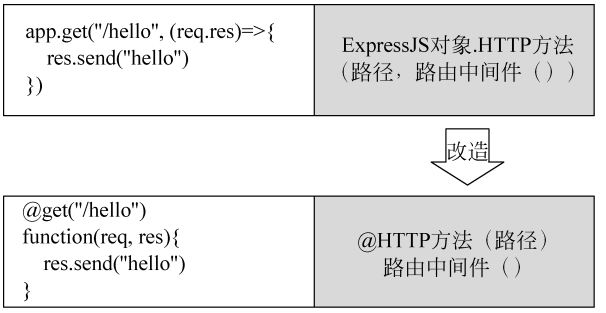

图 3-8 路由装饰器和 Express 路由的对应关系

由于 get 是 TypeScript 的关键字，get 不能作为装饰器名称，所以路由装饰器的名称为@GetMapping、@PostMapping 和@RequestMapping。

@RequestMapping 对应的是 ExpressJS 的 ALL 请求方法，它可以接受任意 HTTP 方法的请求。

路由装饰器的开发思路是由路由装饰器收集路径和对应的方法函数，然后在特定的时机将收集到的内容赋值给 ExpressJS 对象，完成路由装饰器的装配。

简单来讲就是分为收集路由信息和设置路由中间件两个步骤，代码如下：

```
//chapter03/02 - routes/src/route - mapping.decorator.ts

const routerMapper = {
  "get" : {},
  "post" : {},
  "all" : {}
}

function setRouter(app: express.Application) {
  for (let key in routerMapper["get"]) {
    app.get(key, routerMapper["get"][key]);
  }
  for (let key in routerMapper["post"]) {
    app.post(key, routerMapper["post"][key]);
  }
  for (let key in routerMapper["all"]) {
    app.all(key, routerMapper["all"][key]);
  }
}

//GET 路由装饰器
function GetMapping(value: string) {
  return function (target, propertyKey: string) {
    routerMapper["get"][value] = target[propertyKey];
  };
}
//POST 路由装饰器
function PostMapping(value: string) {
  return function (target, propertyKey: string) {
    routerMapper["post"][value] = target[propertyKey];
  };
}
//通用路由装饰器
function RequestMapping(value: string) {
  return function (target, propertyKey: string) {
    routerMapper["all"][value] = target[propertyKey];
  };
}
export { GetMapping, PostMapping, RequestMapping, setRouter };
```

route-mapping.decorator.ts 文件导出的@GetMapping、@PostMapping 和@RequestMapping 是路由装饰器。3 个装饰器的代码基本相同，这里以@GetMapping 为例讲解。

常量 routerMapper 有 3 个属性,分别对应 3 个 HTTP 方法名称。routerMapper 用于收集装饰器对应的信息。

@GetMapping 装饰器把装饰器参数 value 作为键,把被装饰器方法的 target 和 propertyKey 两个参数作为值,这对键值被赋值到 routerMapper["get"]对象,完成对路由信息的收集。

接下来 setRouter()方法作为给 ExpressJS 对象设置路由的主方法,它的参数是 ExpressJS 对象。setRouter()方法内部对 routerMapper["get"]等 3 个对象进行循环,每次循环都取出一对键值,分别用 app.get()、app.post()、app.all()设置路由中间件,完成设置路由的步骤。

route-mapping.decorator.ts 文件最后对 setRouter()方法进行导出以供 ExpressServer 使用。ExpressServer 改动比较少,只需在 app.listen()之前调用 setRouter()方法并传入 app 对象,代码如下:

```
//chapter-3/02-routes/src/default/express-server.class.ts


public start(port: number) {
    const app: express.Application = express();
    this.middlewareList.forEach(middleware => {
        app.use(middleware);
    });
    //传入 app 对象即可对路由进行设置
    setRouter(app);
    app.listen(port, () => {
        log("server start at port: " + port);
    });
}
```

3.2.4 测试路由装饰器

路由装饰器的效果跟 app.get()方法的效果是否一致呢? 可以创建页面对其进行测试,代码如下:

```
//chapter03/02-routes/test/first-page.class.ts

import { log } from "../src/speed";
import { GetMapping } from "../src/route-mapping.decorator";
export default class FirstPage {

    @GetMapping("/first")
    public index(req: any, res: any) {
        log("FirstPage index running");
        res.send("FirstPage index running");
    }

}
```

用 ts-node 命令启动服务后，打开浏览器访问 http://localhost:8080/first 可以看到输出的内容，表明路由装饰器可正常运作，如图 3-9 所示。

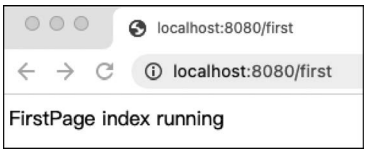

图 3-9　路由装饰器的显示效果

3.2.5　优化路由装饰器

观察 route-mapping.decorator.ts 文件的 3 个装饰器，它们的结构是类似的，只有名称不同，因此，可以对其进行重构整合。

首先将 3 个装饰器的方法独立出来作为公共方法，公共方法 mapperFunction 里面的方法名参数是 get、post、all 中的一种，代码如下：

```
//chapter03/05 - template/src/route - mapping.decorator.ts


function mapperFunction(method: string, value: string) {
  return (target: any, propertyKey: string) => {
    routerMapper[method][value] = (...args) => {
      const getBean = BeanFactory.getBean(target.constructor);
      return getBean[propertyKey](...args);
    }
  }
}
```

mapperFunction 函数代替了原有的功能，3 个装饰器可以直接使用 mapperFunction 函数，只是参数不同，代码如下：

```
//chapter03/05 - template/src/route - mapping.decorator.ts


const GetMapping = (value: string) => mapperFunction("get", value);
const PostMapping = (value: string) => mapperFunction("post", value);
const RequestMapping = (value: string) => mapperFunction("all", value);

export { GetMapping, PostMapping, RequestMapping, setRouter };
```

重构后的代码比较整洁、清晰，方便后续进行开发扩展。

3.2.6　小结

本节详细讲解了 ExpressJS 的路由功能，并且把 ExpressJS 的路由功能改造成 TypeScript 装饰器。

只要把路由装饰器放在任意类的成员方法上，即可让该方法具备路由功能。对比原来的 ExpressJS 路由方法，路由装饰器更易用、灵活，并且在编码风格上更能体现 TypeScript 语言的优势。

3.3 路由切面功能

在 3.2 节框架已将 ExpressJS 的路由方法改造成路由装饰器，解决了访问路径与页面程序的对应问题，接下来考虑一个问题：怎样在页面程序执行前或者执行后，动态地加入一些额外功能呢？例如用户成功登录系统后，访问的每个页面都必须进行权限的检查，或者页面程序对执行的时间进行记录统计，检查是否存在页面性能低下的情况。

3.3.1 面向切面编程

在回答上面的问题之前，先来看一个计算页面执行时间的例子，代码如下：

```
@GetMapping("/hello")
Function record(req, res){
  //开始记录当前时间
  console.time('hello - page');
  //一些繁重的处理,如以下的循环
  let counter = 0;
  for (let i = 0; i < 100000000; i++) {
    counter++;
  }
  //结束记录时间,并输出执行时间
  console.timeEnd('hello - page')
  res.send("hello");
}
```

访问该页面会输出页面的执行时长，例如 hello-page：95.721ms，结果取决于系统性能。

像这样的时长统计如果直接放到各个页面的代码里，则会产生以下两个问题：

(1) 重复代码过多，影响页面本来代码逻辑的阅读。

(2) 难以维护，当需要改动这些代码时，每个页面都必须进行修改，工作量太大并且容易遗漏。

以上两个问题该怎么解决？根据 3.1.2 节学过的中间件机制，页面的请求会经过一条请求-响应的链条，链条上串联着各种中间件，页面的逻辑处在路由中间件的位置，如图 3-10 所示。

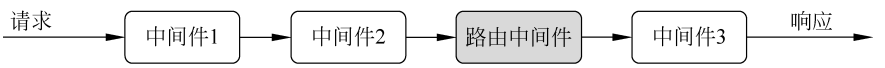

图 3-10 单个请求-响应链条

那么要解决上述问题，可以在请求-响应链条上，在路由中间件的前后分别动态地加上两个处理程序，如图 3-11 所示。

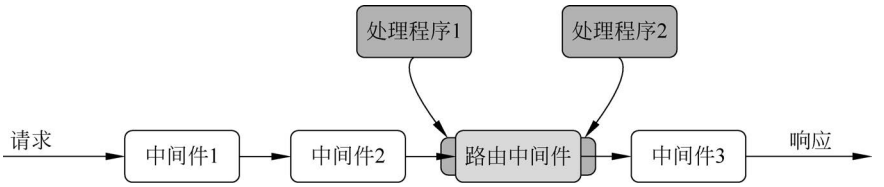

图 3-11　在请求-响应链条上加入处理程序

页面请求的次数当然不止一次，如果从多次页面访问的请求-响应链条来观察，则像是用刀子在链条上的路由中间件两边的垂直位置切上两刀，然后在刀切的位置加入额外的处理程序，这就是服务器端开发领域的一个重要概念：面向切面编程，如图 3-12 所示。

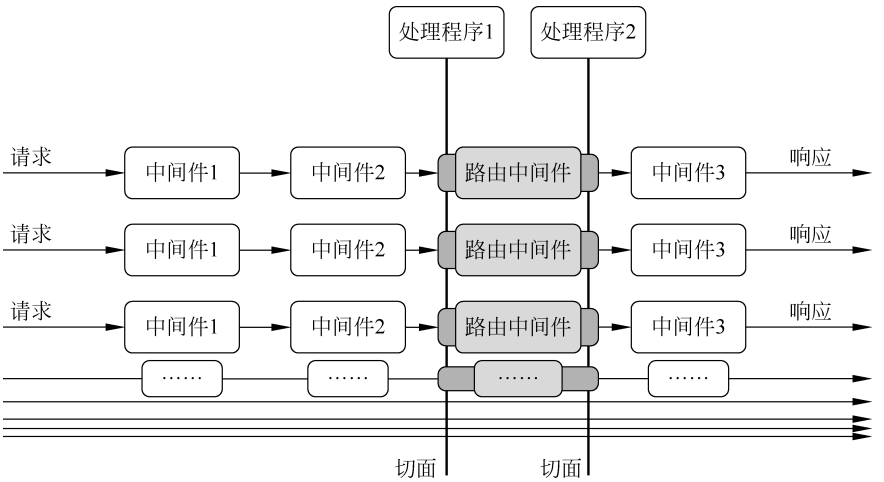

图 3-12　在多个请求-响应链条上切入处理程序

注意：为什么要插入处理程序，而不是直接在路由中间件前后再加两个中间件呢？实际上，处理程序和路由中间件是一体的，处理程序需要取得路由中间件的数据或进行一些操作，而中间件从代码逻辑上是相互独立的，这就造就了中间件机制的灵活性，但也使中间件没有办法轻易地取得前后其他中间件的数据。

面向切面程序设计(Aspect-Oriented Programming，AOP)是计算机科学中的一种程序设计思想，旨在将交叉切入关注点与作为业务主体的核心关注点进行分离，以提高程序代码的模块化程度。通过在现有代码的基础上增加额外的通告(Advice)机制，能够对被声明为“点切入”(Pointcut)的代码块进行统一管理与装饰，例如“对所有方法名以 set * 开头的方法添加后台日志”。该思想使开发人员能够将与代码核心业务逻辑关系不那么密切的功能(如日志功能)添加到程序中，同时又不降低业务代码的可读性。

上述定义揭示了面向切面编程的两个目标：

(1) 为程序增加与业务逻辑关系并不密切的额外功能。

(2) 在增加额外功能的同时，不降低业务代码的可读性。在代码层面，切入点的代码和功能都是与业务主体代码分离的，原来的代码不需要做任何的修改。

面向切面编程的实现，选择的是可以在既有代码上增加额外能力的语言特性，例如Java 的注解，或 TypeScript 的装饰器等。既能指定切面代码的执行位置，又不需要将这些逻辑配置入侵到现有代码的内部，从而使代码仍然清晰可读。

3.3.2 设计切面程序功能

框架的路由切面程序仍采用装饰器实现。

前置切面装饰器@before 标记在目标方法前需要执行的程序，后置切面装饰器@after 标记在目标方法之后需要执行的程序，而两个切面装饰器的参数可指定目标方法。

以前置切面装饰器的使用举例，代码如下：

```
//chapter03/03 - aop/test/aop - test.class.ts

import { before, log, onClass, after } from "../src/speed";
import FirstPage from "./first - page.class";

@onClass
export default class AopTest {

    @before(FirstPage, "index")
    public FirstIndex() {
        log("Before FirstPage index run, at AopTest FirstIndex.");
        return "FirstIndex";
    }
}
```

@before 装饰的 FirstIndex()方法将在目标方法执行之前运行，@before 的两个参数可指定目标方法，第 1 个参数代表目标方法所在的类是 FirstPage，第 2 个参数代表 FirstPage 类的 index()方法就是目标方法。@before 与目标方法的关系如图 3-13 所示。

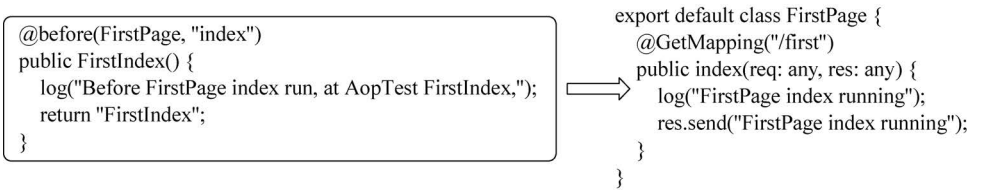

图 3-13 前置切面装饰器在目标方法前执行

开发切面装饰器的思路是当执行@before 装饰器时，从它的参数里获得目标方法，用一个新方法来“代替”目标方法，新方法的代码先执行@before 装饰的方法，再来调用目标方法。这就相当于给目标方法设置了一个“代理人”，将目标方法交给代理人来执行，代理人可

以在其中插入所需的额外代码，该编程技巧就是设计模式的代理模式。

代理模式：为其他对象提供一种代理以控制对这个对象的访问。在某些情况下，一个对象不适合或者不能直接引用另一个对象，而代理对象可以在客户端和目标对象之间起到中介的作用。

代理模式的优势是代理程序和目标方法的接口是完全相同的，任何调用目标方法的代码都能不作修改而直接替换为代理程序，并且替换的过程对开发者而言是无感知的。

使用代理模式实现路由切面功能有以下两个要点：

(1) 创建切面装饰器，其装饰的代理方法包含切面代码和目标方法。

(2) 找到路由装饰器的方法实例，在程序运行时对目标方法进行替换。

3.3.3　@before 切面装饰器

开发切面程序，很适合使用测试先行的开发方式。

先建立切面功能测试类 AopTest 类，切面要实现的前置和后置功能都在里面，代码如下：

```
//chapter03/03-aop/test/aop-test.class.ts

import { before, log, onClass, after } from "../src/speed";
import FirstPage from "./first-page.class";

@onClass
export default class AopTest {

    //前置切面装饰器，在 FirstPage 的 index()方法调用前执行
    @before(FirstPage, "index")
    public FirstIndex() {
        log("Before FirstPage index run, at AopTest FirstIndex.");
        log("AopTest FirstIndex run over." + this.getWordsFromAopTest());
        return "FirstIndex";
    }

    //在测试切面装饰器里再调用其他方法
    public getWordsFromAopTest() {
        return "getWordsFromAopTest";
    }

    //前置切面装饰器，在 FirstPage 的 getTestFromFirstPage()方法调用前执行
    @before(FirstPage, "getTestFromFirstPage")
    public testGetTestFromFirstPage() {
        log("AopTest testGetTestFromFirstPage run over.");
    }
}
```

@before 装饰 AopTest 的 FirstIndex()方法，@before 参数用于指向 FirstPage 页面类的 index 页面。期待的结果是在 index()页面被浏览器访问时，程序先执行这里的 FirstIndex()方法，然后执行原来的 index()方法。

接下来要找到 FirstPage 类的实例,这样才能对其 index()方法进行替换。由于 FirstPage 的 index()页面方法是通过路由装饰器@GetMapping 来执行的,因此可以改动@GetMapping 装饰器的代码,将 index()方法存储到对象工厂。

当@before 执行时,@before 装饰的 FirstIndex()方法就会替换原来的 index()方法。最后当 Web 服务访问该页面时,将执行替换后的 FirstIndex()方法,从而实现切面开发的意图,代码如下:

```
//chapter03/03 - aop/src/route - mapping.decorator.ts


function GetMapping(value: string) {
  return function (target, propertyKey: string) {
    routerMapper["get"][value] = () => {
      let getBean = BeanFactory.getBean(target.constructor);
      if(getBean === undefined) {
        BeanFactory.putBean(target.constructor, target);
        getBean = target;
      }
      return getBean[propertyKey];
    }
  };
}
```

接着开发@before 路由切面装饰器,它的参数表示指向哪个页面。装饰器根据参数从 BeanFactory 取出页面实例,替换成被装饰的方法后再存放回去,代码如下:

```
//chapter03/03 - aop/src/speed.ts


function before(constructorFunction, methodName: string) {
    const targetBean = BeanFactory.getBean(constructorFunction);
    return function ( target, propertyKey: string) {
        const currentMethod = targetBean[methodName];
        targetBean[methodName] = (...args) => {
            target[propertyKey]();
            return currentMethod(...args);
        }
        BeanFactory.putBean(constructorFunction, targetBean);
    };
}
```

完成上述改动后,启动服务进行测试,命令如下:

```
ts - node test/main.ts

[LOG] 2022 - 12 - 04 15:52:06 aop - test.class.ts:9 (Object.FirstIndex) Before FirstPage index
run, at AopTest FirstIndex.
[LOG] 2022 - 12 - 04 15:52:06 first - page.class.ts:9 (index) FirstPage index running
```

当服务成功启动后打开浏览器访问 http://localhost/first，命令行输出信息表明，AopTest 类 FirstIndex()方法在页面 index()方法之前被执行了。

切面装饰器的效果达到了预期吗？做个简单实验，在 FirstPage 的 index()页面尝试调用 FirstPage 类的其他方法，代码如下：

```
//chapter03/03 - aop/test/first - page.class.ts


export default class FirstPage {
    @GetMapping("/first")
    public index(req: any, res: any) {
        log("FirstPage index running" + this.getTestFromFirstPage());
        res.send("FirstPage index running");
    }

    public getTestFromFirstPage() {
        return "getTestFromFirstPage";
    }
}
```

服务启动后访问页面，发现命令输出了错误信息：TypeError: Cannot read properties of undefined (reading 'getTestFromFirstPage')，提示 this 没有 getTestFromFirstPage()方法，为什么？

追踪代码，发现@GetMapping 装饰器存入的路由方法只是单独的函数方法，在 setRouter()函数被 app.get()使用时，只是进行了一次函数调用，并没有调用 FirstPage 实例的 index()方法，因此才会提示 this 为空，因为 this 并没有指向 FirstPage 实例。

注意：方法调用者的 this 指向错误问题在 JavaScript 和 TypeScript 编程中较为常见。当遇到 TypeError: Cannot read properties of undefined 的错误提示时，必须对 this 的指向进行检查。

问题的成因是当@GetMapping 存入对象时，index()方法仅是一个函数，类 FirstPage 并没有被实例化。解决方法就是要先将 FirstPage 实例化，再来调用实例的 index()方法，代码如下：

```
//chapter03/03 - aop/src/route - mapping.decorator.ts


function GetMapping(value: string) {
  return function (target, propertyKey: string) {
    routerMapper["get"][value] = (...args) => {
      let getBean = BeanFactory.getBean(target.constructor);
      if(getBean === undefined) {
        BeanFactory.putBean(target.constructor, target);
        getBean = target;
```

```
        }
        let targetObject = new target.constructor();
        return targetObject[propertyKey](...args);
      }
    };
}
```

@GetMapping 装饰的 FirstPage 类就是上述代码中的 target,这里对其进行实例化,再来调用它的 propertyKey 方法,propertyKey 方法实际的值是 index()方法,随后返回调用 index()方法的结果,供 app.get()使用。

重新执行程序,访问/first 地址,发现 FirstPage 里 this.getTestFromFirstPage()能够正常使用,也不报错,但程序也没有执行 AopTest 类 FirstIndex()方法。

代码中,@GetMapping 对 FirstPage 进行了一次 new 实例化操作,因此@GetMapping 取得的 index()方法和@before 取得的 index()方法,它们的实例不是同一个 FirstPage,所以这时@before 里的 index()方法就不会被执行了。那么,有没有办法让两者从同一个位置来取得 FirstPage 实例呢?

答案是用类装饰器,类装饰器的主要作用就是收集实例。

这里用的是@onClass 类装饰器,它直接对 FirstPage 类进行实例化,存入对象工厂。

@before 装饰器从对象工厂取得 FirstPage 实例,用 Object.assign()函数执行替换方法的操作,而@GetMapping 装饰器也从对象工厂取得同一个 FirstPage 实例,调用其 index()方法返回,代码如下:

```
//chapter03/03-aop/src/speed.ts


function before(constructorFunction, methodName: string) {
    const targetBean = BeanFactory.getBean(constructorFunction);
    return function (target, propertyKey: string) {
        const currentMethod = targetBean[methodName];
        Object.assign(targetBean, {
            [methodName]: function (...args) {
                target[propertyKey](...args);
                log("========== before ==========");
                currentMethod.apply(targetBean, args);
            }
        })
    };
}
```

重启服务并访问页面,发现问题得以解决,AopTest 类如预期在 index()页面执行前输出了内容。

3.3.4 @after 切面装饰器

接下来是后置切面装饰器@after,和前置切面不同的地方是后置切面装饰器需要接收

页面方法的返回值，方便对页面方法的输出进行后置处理，代码如下：

```
//chapter03/03 - aop/src/speed.ts


function after(constructorFunction, methodName: string) {
    const targetBean = BeanFactory.getBean(constructorFunction);
    return function (target, propertyKey: string) {
        const currentMethod = targetBean[methodName];
        Object.assign(targetBean, {
            [methodName]: function (...args) {
                const result = currentMethod.apply(targetBean, args);
                const afterResult = target[propertyKey](result);
                log(" ========== after ========== ");
                return afterResult ?? result;
            }
        })
    };
}
```

最后，在 AopTest 类加入@after 的测试代码，代码如下：

```
//chapter03/03 - aop/test/aop - test.class.ts

import { before, log, onClass, after } from "../src/speed";
import FirstPage from "./first - page.class";

@onClass
export default class AopTest {

    //前置切面装饰器，在 FirstPage 的 index()方法调用前执行
    @before(FirstPage, "index")
    public FirstIndex() {
        log("Before FirstPage index run, at AopTest FirstIndex.");
        log("AopTest FirstIndex run over." + this.getWordsFromAopTest());
        return "FirstIndex";
    }

    //在测试切面装饰器里再调用其他方法
    public getWordsFromAopTest() {
        return "getWordsFromAopTest";
    }

    //前置切面装饰器，在 FirstPage 的 getTestFromFirstPage()方法调用前执行
    @before(FirstPage, "getTestFromFirstPage")
    public testGetTestFromFirstPage() {
        log("AopTest testGetTestFromFirstPage run over.");
    }

    //后置切面装饰器，在 FirstPage 的 index()方法调用后执行
```

```
    @after(FirstPage, "index")
    public testFirstIndexAfter(result) {
        log("AopTest testFirstIndexAfter run over, result: " + result);
        log(result);
    }
}
```

至此,路由切面装饰器已经开发完成,然而它们并不完善,例如它们只能对单一的页面方法配置额外的操作,但实际情况可能是一系列的页面方法都需要做类似的操作,这就要求切面装饰器支持类似通配符进行切面。本章限于篇幅不详述,读者可以自行尝试。

3.3.5 小结

本节介绍了面向切面编程的概念。面向切面编程是一种在现有代码的基础上添加额外功能,但不会影响代码可读性的设计模式。

从开发意图看,面向切面编程和2.2.3节的依赖注入相似,但面向切面更关注的是“切入点”,因此当需要为现有代码添加额外功能时需要关注以下两点:

(1) 如果关注点是增强现有代码,使其具备更广泛的能力,则应当选择依赖注入,直接改变现有代码的适用范围,方法在2.2.3节有详细介绍。

(2) 如果关注点是给现有代码提供额外的与业务核心逻辑关系不密切的功能,尤其是在核心逻辑前后加入重复性功能时,就应当采用面向切面编程。


18min

3.4 请求参数装饰器

本节在路由系统上实现参数装饰器,以方便开发者取得具体的请求参数。

被@GetMapping装饰的页面方法接受两个参数req、res,分别是ExpressJS的请求对象和响应对象,代码如下:

```
@GetMapping("/first")
public index(req: any, res: any) {
    log("FirstPage index running");
    res.send("FirstPage index running");
}
```

直接取得上述的请求数据,虽然可以满足开发的需要,但是单就index()函数而言,从函数入参是无法确定其内部使用的是哪些请求数据。这样会产生两个问题:

(1) 无法确保参数的有效性。因为程序直接调用req整个对象,具体的参数获取代码被写在index()方法内部,也就无法利用TypeScript类型系统对参数类型进行校验,只能期待代码本身会对参数进行限制和校验。

(2) 函数无法进行测试。单元测试要求函数有清晰的入参和明确的返回值。如果入参是req这样内容非常复杂的请求对象,则在测试之前构建req对象用来测试就变得十分困

难，并且函数的代码的可读性也很低。

因此，开发请求参数装饰器，给路由函数标识各种请求参数，一方面可以方便参数类型的校验和测试，另一方面可以通过标识获得准确的数据内容。

3.4.1　设计请求参数装饰器

回顾 2.1.2 节装饰器的知识，参数装饰器必须配合方法装饰器使用。它的关键在于第 3 个参数 parameterIndex 变量。

parameterIndex 表示标记参数所处的位置（位置从 0 开始计算），方便找到具体参数。举个例子，@reqBody 装饰器的作用是取得请求体数据，请求体的格式通常是 JSON，代码如下：

```
@postMapping("/request/body")
testBody(@res res, @reqBody body: object) {
    log("body: " + JSON.stringify(body));
    res.send("test body");
}
```

上述代码@reqBody 装饰参数的 parameterIndex 为 1，表示此参数在 testBody()方法的第 2 个参数位置，当对 testBody()方法进行赋值时，即可将其第 2 个参数值设置为 req.body。这样就能实现@reqBody()装饰器的功能。

本节框架将开发一系列请求参数装饰器，这些装饰器的基本实现逻辑都是类似的，找到它们的位置，然后在路由方法调用时便可在对应位置输入适当的请求数据。

请求参数装饰器有以下这些：

（1）@reqBody，获取请求体数据，通常浏览器提交的 JSON 数据会被存放在请求体里。

（2）@reqParam，获取请求的路径参数，例如从请求路径/request/param/56 里取得 56 这个值。

（3）@reqQuery，获取请求的附加参数，即网址的问号后的参数值。

（4）@reqForm，获取提交 HTML 表单域数据。

（5）@req，中间件的 Request 对象。

（6）@request，@req 的别名。

（7）@res，中间件的 Response 对象。

（8）@response，@res 的别名。

（9）@next，中间件的 next 参数。

上面的@request 和@response 是别名，在导出装饰器时用 as 关键字即可，代码如下：

```
export { next, reqBody, reqQuery, reqForm, reqParam,
    req, req as request, res, res as response …
```

3.4.2　请求参数装饰器的实现

除了@request 和@response 之外，其他 7 个装饰器的基本实现思路是相同的，即装饰

器负责收集数据,存入全局变量 routerParams 供路由系统使用,代码如下:

```
//chapter03/04 - page - params/src/route.decorator.ts


const routerParams = {};

//Request 对象的参数装饰器
function req(target: any, propertyKey: string, parameterIndex: number) {
  const key = [target.constructor.name, propertyKey, parameterIndex].toString();
  routerParams[key] = (req, res, next) => req;
}

//Response 对象的参数装饰器
function res(target: any, propertyKey: string, parameterIndex: number) {
  const key = [target.constructor.name, propertyKey, parameterIndex].toString();
  routerParams[key] = (req, res, next) => res;
}

//Next 函数的参数装饰器
function next(target: any, propertyKey: string, parameterIndex: number) {
  const key = [target.constructor.name, propertyKey, parameterIndex].toString();
  routerParams[key] = (req, res, next) => next;
}

//取得请求体内容的参数装饰器
function reqBody(target: any, propertyKey: string, parameterIndex: number) {
  const key = [target.constructor.name, propertyKey, parameterIndex].toString();
  routerParams[key] = (req, res, next) => req.body;
}

//取得请求属性的参数装饰器
function reqParam(paramName: string) {
  return (target: any, propertyKey: string, parameterIndex: number) => {
    const key = [target.constructor.name, propertyKey, parameterIndex].toString();
    routerParams[key] = (req, res, next) => req.params[paramName];
  }
}

//取得 Query 属性值的参数装饰器
function reqQuery(paramName: string) {
  return (target: any, propertyKey: string, parameterIndex: number) => {
    const key = [target.constructor.name, propertyKey, parameterIndex].toString();
    routerParams[key] = (req, res, next) => req.query[paramName];
  }
}

//取得表单属性值的参数装饰器
function reqForm(paramName: string) {
```

```
    return (target: any, propertyKey: string, parameterIndex: number) => {
      const key = [target.constructor.name, propertyKey, parameterIndex].toString();
      routerParams[key] = (req, res, next) => req.body[paramName];
    }
  }

  export { next, reqBody, reqQuery, reqForm, reqParam,
      req, req as request, res, res as response ……
```

上述 7 个装饰器以是否带参数来区分，可分成两类：

(1) 不带参数的装饰器@req、@res、@next、@reqBody。

(2) 带参数的装饰器@reqParam、@reqQuery、@reqForm。

不带参数的参数装饰器以@req 举例，代码先将 target 对象、propertyKey 被装饰的方法名及 parameterIndex 参数位置这 3 个信息拼装成唯一键，其值为(req,res,next)=> req 中间件，键值存入 routerParams，其他不带参数的装饰器与此类似，只是值不一样，例如@reqBody 的值为(req,res,next)=> req. body。

带参数的装饰器的逻辑基本相同，多出来的输入参数作为获取请求数据的名称。例如@ reqQuery 的值为(req,res,next)=> req. query[paramName]。这里的 paramName 即装饰器参数。

接着，路由统一方法 mapperFunction()函数即可使用 routerParams 的数据，对页面方法进行赋值，代码如下：

```
//chapter03/04 - page - params/src/route.decorator.ts

…
let args = [req, res, next];
if(paramTotal > 0) {
  for(let i = 0; i < paramTotal; i++) {
    if(routerParams[[target.constructor.name, propertyKey, i].toString()]){
      args[i] = routerParams[[target.constructor.name, propertyKey, i].toString()](req,
res, next);
    }
  }
}
const testResult = await routerBean[propertyKey].apply(routerBean, args);
…
```

mapperFunction()函数修改了页面的调用方式。页面方法的参数 args 的初始值是 req、res、next 这 3 个对象的数组，然后根据页面方法的参数个数进行循环，在 parameterIndex 位置被赋值成 routerParams 对应的值。

按这样的赋值方法，当页面方法部分参数被标记参数装饰器，但有些参数并没有被标记时，没有标记的参数还是按原有 req、res、next 来使用，代码如下：

```
@getMapping("/request/query")
testQuery(req, res, @reqQuery("id") id: number) {
    log("id: " + id);
    res.send("test query");
}
```

在上述代码中 req 和 res 仍是 Request 和 Response 对象,只是第 3 个参数被覆盖为@reqQuery 的 id 值。

mapperFunction()函数最后使用 apply()函数来赋值这些参数并调用页面方法。

```
const testResult = await routerBean[propertyKey].apply(routerBean, args);
```

注意:apply() 方法是 JavaScript 执行函数的方法之一,它可以用数组格式的参数来执行函数。

mapperFunction()函数的代码的全局变量 routerParamsTotal 记录了每个路由方法的参数的个数,方便对参数进行循环赋值,而在使用 3.3 节实现的@before 和@after 两个切面装饰器时,routerParamsTotal 对参数个数记录不准确,因为切面装饰器已经替换了原来的页面方法,所以取得的参数个数是替换后的新方法的参数的个数。

这里需要对切面装饰器进行修改,在它们的内部记录原页面方法的参数的个数,代码如下:

```
//chapter03/04-page-params/src/route.decorator.ts

//前置切面装饰器
function before(constructorFunction, methodName: string) {
  const targetBean = getComponent(constructorFunction);
  return function (target, propertyKey: string) {
      const currentMethod = targetBean[methodName];
      //检查方法的参数的个数,并记录以备参数装饰器使用
      if(currentMethod.length > 0){
           routerParamsTotal [[ constructorFunction. name, methodName ]. toString ( )] =
currentMethod.length;
      }
      Object.assign(targetBean, {
          [methodName]: function (...args) {
              target[propertyKey](...args);
              return currentMethod.apply(targetBean, args);
          }
      })
  };
}
//@after 类似,代码省略
```

完成 7 个参数装饰器和 mapperFunction()函数的修改后,接着对其进行测试,代码如下:

```
//chapter03/04 - page - params/test/test - request.class.ts


@component
export default class TestRequest {

    //测试用装饰器获取 Request 和 Response 对象
    @getMapping("/request/res")
    testRes(@req req, @res res) {
        res.send("test res");
    }

    //测试用装饰器获取 Query 参数 id
    @getMapping("/request/query")
    async testQuery(req, res, @reqQuery id: number): Promise<MutilUsers> {
        log("id: " + id);
        return Promise.resolve(new MutilUsers("group", [new UserDto(1, "name"), new UserDto
(2, "name")]));
    }

    //测试用装饰器获取请求体对象
    @postMapping("/request/body")
    testBody(@res res, @reqBody body: object):MutilUsers {
        log("body: " + JSON.stringify(body));
        return new MutilUsers("group", [new UserDto(1, "name"), new UserDto(2, "name")]);
    }

    //测试用装饰器获取表单参数
    @postMapping("/request/form")
    testForm(@res res, @reqForm("name") name: string) {
        log("form: " + JSON.stringify(name));
        res.send("test form");
    }

    //测试用装饰器获取路径参数 id
    @getMapping("/request/param/:id")
    testParam(@res res, @reqParam id: number) {
        log("id: " + id);
        res.send("test param");
    }
}
```

上述代码的测试过程较为简单，可参考注释说明。

3.4.3　用 toString()优化装饰器

参数装饰器@reqQuery、@reqParam 在使用时通常请求参数和参数名是相同的，例如@reqParam("id") id: number，请求参数是 id，方法上面的参数名同样也是 id。那么是否可以优化成只需一个 id，也就是@reqParam id: number 即可取得请求参数 id。

注意：@reqForm 的情况与此类似，不过在实际开发中@reqForm 代表 HTML 表单域的名称，清晰写出表单域名称有助于清楚地表达页面方法和表单之间的对应关系，所以不做缩略。

那么，问题在于如何取得页面方法的参数名称。通常可以考虑反射机制，在 TypeScript 里使用 Reflect Metadata 来取得运行时信息。

但是 Reflect Metadata 对类方法的信息，只提供了 design:paramtypes、design:type 两条信息，两者都是针对参数类型的，并没有取得参数名称的方法，代码如下：

```
Reflect.getMetadata("design:paramtypes", target, propertyKey);
Reflect.getMetadata("design:type", target, propertyKey);
```

注意：实际上在很多编程语言里没有直接取得方法/函数的参数名称的方案。语言的设计者总是认为参数名称不重要。例如 Java 语言编译后的参数会被重新命名，要取得参数名称只能修改其编译过程。

TypeScript、JavaScript 有个特殊的原型方法 toString()，通常用于显示变量的类型内容，但由于在 TypeScript、JavaScrip 里函数是“一等公民”，故函数也可以调用 toString()，而且函数调用 toStirng()可以取得函数本身的代码，十分神奇！

下面通过 ts-node 命令尝试使用 toString()，ts-node 命令加-p -e 标识，可以直接运行代码，命令如下：

```
ts-node -p -e '(function compute(a: number, b: number): number {return a + b}).toString()'

function compute(a, b) { return a + b; }
```

从 ts-node 命令的结果可以看到 toString()输出了 compute()函数本身的代码，正如所期待的那样，这里的参数 a 和 b 保持了原样。只是代码里面的 number 类型被抹除了，这是因为 TypeScript 编译成 JavaScript 后类型信息会被抹除。

借助 toString()获取整个函数源码的能力，使用正则匹配即可取得参数的名称，代码如下：

```
//chapter03/04-page-params/src/route.decorator.ts


function getParamInFunction(fn: Function, index: number) {
  const code = fn.toString().replace(/((\/\/.*$)|(\/\*[\s\S]*?\*\/))/mg, '').replace
(/=>.*$/mg, '').replace(/=[^,]+/mg, '');
  const result = code.slice(code.indexOf('(') + 1, code.indexOf(')')).match(/([^\s,]+)/g);
  return result[index] || null;
}
```

然后把@reqParam、@reqQuery 修改成不带参数的装饰器，并使用 getParamInFunction() 来取得参数名称，代码如下：

```
//chapter03/04 - page - params/src/route.decorator.ts


function reqQuery(target: any, propertyKey: string, parameterIndex: number) {
  const key = [target.constructor.name, propertyKey, parameterIndex].toString();
  const paramName = getParamInFunction(target[propertyKey], parameterIndex);
  routerParams[key] = (req, res, next) => req.query[paramName];
}
//@reqParam 类似，这里忽略
```

完成了对@reqParam、@reqQuery 的优化后，测试页面可以写得简单一些，直接省略参数名称，代码如下：

```
//chapter03/04 - page - params/test/test - request.class.ts


@getMapping("/request/param/:id")
testParam(@res res, @reqParam id: number) {
    log("id: " + id);
    res.send("test param");
}
```

3.4.4　小结

本节实现了请求参数装饰器。请求参数装饰器对页面方法的参数进行标识，以便取得所需的参数信息，比起只有 Request 和 Response 对象的常规路由方法，前者明确了页面的输入信息，有利于保证参数的有效性和可测试性。

本节装饰器的开发过程体现了实现参数装饰器的两个关键点：

(1) 参数位置 parameterIndex 是定位参数的关键信息。

(2) 用全局变量将参数装饰器和对应的方法装饰器关联起来，进而实现所需功能。

此外，值得注意的是 toString()函数，它具有获取函数源代码的能力。借助 toString() 的能力，简化了@reqParam 和@reqQuery 两个装饰器的参数名，方便开发者使用。

3.5　响应处理与模板引擎

响应处理处在请求-响应链条里的最后一环。本节将讲解它是如何对数据按逻辑进行处理的，将处理结果返回浏览器，完成一次请求的全过程。模板引擎是响应处理中最为复杂的一种，它也是 Web 开发的重点知识之一。

响应处理跟显示内容直接相关，因此它对应了 MVC 模式中的视图层。

3.5.1 MVC 设计模式

MVC 模式(Model-View-Controller)是软件工程中的一种软件架构模式,把软件系统分为模型(Model)、视图(View)和控制器(Controller)等 3 个层级,如图 3-14 所示。

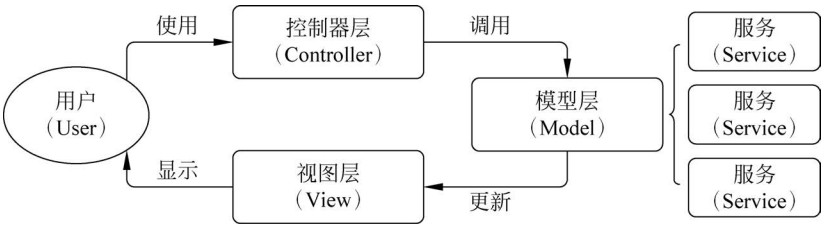

图 3-14 MVC 层级间的关系

(1) 控制器层提供用户操作的外部接口,Web 服务里通常指的是请求路径对应的页面方法。控制器层接收到用户请求后,对数据进行初步校验、转换等处理,然后传递给模型层进一步地进行业务处理。

(2) 模型层是业务逻辑与数据库打交道的部分,通常这部分代码会被写成多个服务(Service),每个服务都对应着相关功能的业务逻辑,例如 Order Service 对应的是订单逻辑,User Service 对应的是用户逻辑等。在业务规模较大的场景里,模型层的架构可以扩展为独立的分布式系统,也就是微服务架构。

(3) 视图层定义了系统如何显示数据,即本节的响应处理。响应处理主要有两种形式,即 JSON 格式和模板引擎。

3.5.2 JSON 格式输出

著名的框架 Spring Boot 有个非常方便的特性,当页面方法的返回值是对象时,这个对象会被自动转换成 JSON 格式并返回浏览器,无须在各个页面响应时都进行对象到 JSON 的格式转换,从而减少开发者编写的重复性编码。

对比在页面方法里用 Response 对象的 send()方法输出结果,方法返回值即为输出结果,这显然更符合函数输入/输出的开发认知。

因而本节框架将实现类似的对象自动转 JSON 输出功能。

注意:广泛接触各种 Web 框架,了解它们的特性,发现一些非常方便的特性或者对开发思路有所启发的功能,可以考虑将其在框架里实现。

3.2.5 节的 mapperFunction()函数统一了 3 个路由装饰器的逻辑,因此自动转换 JSON 的功能可以在 mapperFunction()函数实现,而且作为默认输出,当返回值是字符串时,该函数也将输出文本响应,代码如下:

```
//chapter-03/05-template/src/route-mapping.decorator.ts


function mapperFunction(method: string, value: string) {
  return (target: any, propertyKey: string) => {
    routerMapper[method][value] = (req, res, next) => {
      const routerBean = BeanFactory.getBean(target.constructor);
      const testResult = routerBean[propertyKey](req, res, next);
      if (typeof testResult === "object") {
        res.json(testResult);
      } else if (typeof testResult !== "undefined") {
        res.send(testResult);
      }
      return testResult;
    }
  }
}
```

mapperFunction()函数增加对 testResult 变量的检查。testResult 是路由方法的返回结果。

当 testResult 的类型是对象时，使用 json()方法输出，而如果 testResult 的类型是非对象值，则统一当作文本类型处理，使用 send()方法输出。接下来增加两个页面来测试该特性，代码如下：

```
//chapter03/05-tempate/test/first-page.class.ts


@GetMapping("/first/sendJson")
public sendJson() {
  log("FirstPage sendJson running");
  return {
    "from" : "sendJson",
    "to" : "Browser"
  }
}

@GetMapping("/first/sendResult")
public sendResult() {
  log("FirstPage sendResult running");
  return "sendResult";
}
```

启动服务，用浏览器访问 http://localhost:8080/first/sendJson 地址，打开浏览器的开发者工具，可以看到请求响应的类型是 Content-Type: application/json; charset=utf-8，页面响应是 JSON 格式，如图 3-15 所示。

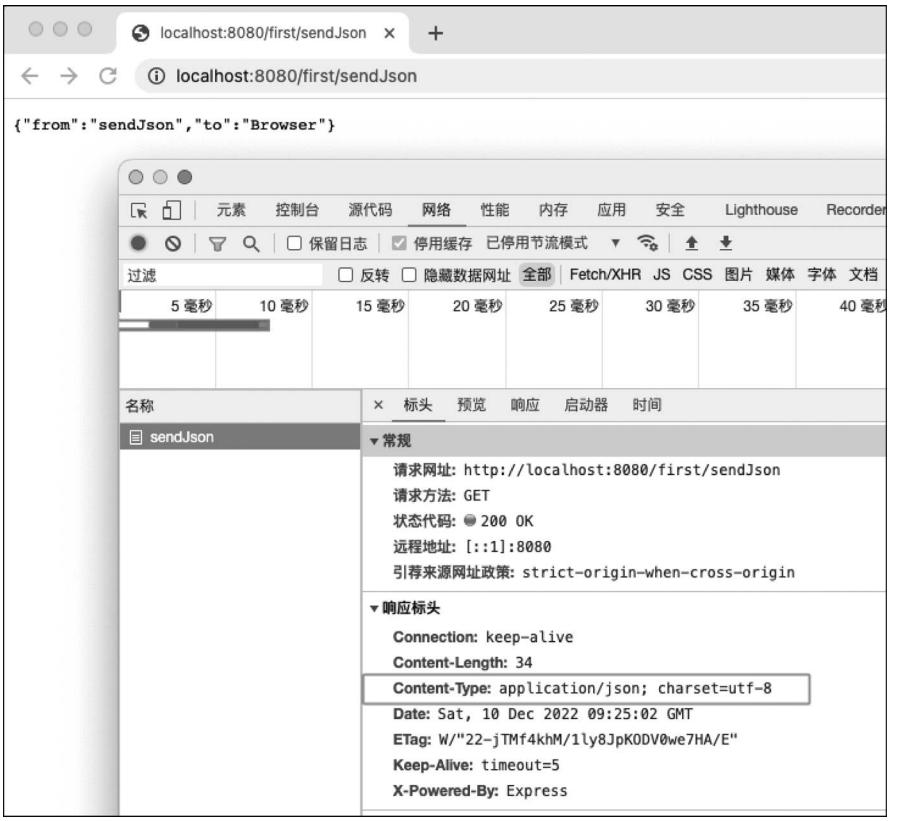

图 3-15 JSON 格式的响应内容


19min

3.5.3 模板引擎是什么

模板引擎(Template Engine)在 Web 框架里的使用由来已久,尤其是在前后端分离成为主流开发方案之前,基于模板引擎的 Web 应用程序是最主要的开发方案。相对前后端分离的前端页面渲染,服务器端框架直接显示页面被称为服务器端渲染。

虽然目前模板引擎已不再是 Web 开发的主流方案,但服务器端渲染还是有不少应用场景,它有以下几点优势:

(1) 对搜索引擎优化(SEO)比较友好。服务器端渲染将页面内容直接输出给浏览器,搜索引擎爬虫收到内容后无须再次渲染,即可识别到内容,因此内容会更容易被搜索引擎收录。

(2) 在不需要太多交互的页面场景,例如博客的文章页,使用服务器端渲染的显示速度更快,更流畅。

服务器端渲染对于前端开发人员的技能要求相对较低。在早期 Web 项目的人员分工里,页面的显示和处理主要由服务器端开发人员使用模板引擎技术进行处理。尤其当这部分开发人员具备部分前端开发能力时,开发团队甚至无须配备专职的前端开发人员。

注意：模板引擎技术在早期也有过一段蓬勃发展的时期，是彼时 Web 开发必备的技能。当时模板引擎的重要程度相当于现在的 Vue 和 React 等前端框架。著名的模板引擎有 PHP 的 Smarty，Spring 框架的 Thymeleaf 等。

从实现原理看，模板引擎是基于字符串替换的技术。模板引擎将输入的数据在模板里面进行字符串替换，最终输出页面。这些进行字符串替换处理的页面被称为模板，如图 3-16 所示。

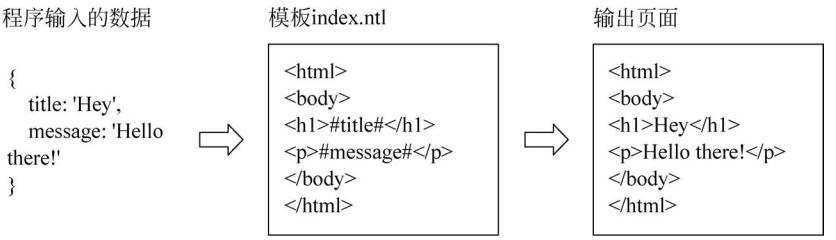

图 3-16　模板引擎是一种字符串替换技术

模板引擎定义了一些简单的语法来编写简单的逻辑，使页面的显示具备一些简单的程序处理能力。模板引擎支持的语法有 if 条件判断、for 循环等。例如著名的 PHP 模板引擎 Smarty 支持 if-elseif-else 形式的条件判断，代码如下：

```
{if $name eq 'Fred'}
    Welcome Sir.
{elseif $name eq 'Wilma'}
    Welcome Ma'am.
{else}
    Welcome, whatever you are.
{/if}
```

模板引擎大部分的功能集中在如何显示页面，它是 MVC 模式的 View 部分。

3.5.4　ExpressJS 的模板引擎

作为成熟的 Web 服务框架，ExpressJS 也内置了对模板引擎的支持。在 3.2.1 节提到过 Response 对象的 render()方法，此方法就是 ExpressJS 使用模板引擎的渲染方法。

ExpressJS 官网展示了一个模板引擎的简单例子，下面将该例子的代码放到 ExpressServer 的 start()方法里执行，以观察其用法，代码如下：

```
//chapter03/05-template/src/default/express-server.class.ts

//开启 Web 服务测试
public start(port: number) {
  const app: express.Application = express();
  this.middlewareList.forEach(middleware => {
```

```
    app.use(middleware);
  });
  setRouter(app);

  //使用 fs 库
  const fs = require('fs')
  //配置模板引擎
  app.engine('ntl', (filePath, options, callback) => {
    //通过 fs 库读取模板文件
    fs.readFile(filePath, (err, content) => {
      if (err) return callback(err)
      //进行简单的替换操作
      const rendered = content.toString()
        .replace('#title#', `<title>${options["title"]}</title>`)
        .replace('#message#', `<h1>${options["message"]}</h1>`)
      return callback(null, rendered)
    })
  })
  //配置模板目录和模板文件后缀
  app.set('views', './test')          //specify the views directory
  app.set('view engine', 'ntl')       //register the template engine

  app.listen(port, () => {
    log("server start at port: " + port);
  });
}
```

上述代码的 app.engine()函数是 ExpressJS 集成模板引擎的主函数。它的第 1 个参数代表模板页面的文件后缀名,只有该后缀的文件才会被识别成模板。

第 2 个参数表示模板引擎的处理函数,处理函数的 3 个参数分别如下。

(1) filePath:模板文件的路径。

(2) options:传给模板的替换数据,即页面方法的数据结果。

(3) callback:模板引擎输出结果的回调函数。callback 的第 1 个参数如果不是 null,则意味着模板引擎执行失败,第 2 个参数是经过模板引擎处理的页面内容。

上述代码演示了一个自定义模板引擎,只做了简单的处理:

(1) 使用 Node.js 文件库,根据 filePath 参数读取模板文件的内容。

(2) 用 options 参数的输入数据,替换了模板内容里 title 和 message 两个字符串,然后调用 callback 返回替换后的内容。

对应的模板文件 index.ntl,代码如下:

```
//chapter03/05 - template/test/views/index.ntl

<html>
<head>
#title#
```

```
</head>
<body>
#message#
</body>
</html>
```

在 FirstPage 类的 index()方法输出该页面，代码如下：

```
//chapter03/05-template/test/first-page.class.ts


@GetMapping("/first")
public index(req: any, res: any) {
  log("FirstPage index running" + this.getTestFromFirstPage());
  res.render('index', { title: 'Hey', message: 'there' })
}
```

修改完毕后启动服务，打开浏览器访问 http://localhost:8080/first 即可看到替换后的页面。

在 FirstPage 类中用于输出的 res.render()，底层调用的就是 app.engine()设置的自定义模板引擎。

3.5.5 模板引擎的选型

上述的自定义引擎仅作为简单的演示，不能真正提供给开发者，这时就需要选择一款成熟的模板引擎，作为 TypeSpeed 框架内置的模板引擎。

之所以选择现成的模板引擎，而不是自行设计开发，是因为模板引擎的选型有以下几个优势：

(1) 现有的模板引擎比较成熟、完成度较高、集成到其他程序的能力也很齐备，例如 Mustache 模板引擎支持在 40 多种编程语言中集成和使用。

(2) 模板引擎的职责比较单一。

(3) 开源的模板引擎数量极多，NPM 库 template engine 标签下的模板引擎有上千个。

此外，模板引擎数量众多也带来一个很现实的问题，几乎每个模板引擎自行实现了一套特殊的语法。这些语法各有千秋，很难说孰高孰低，但数量众多的语法会给开发者带来一定的学习成本。

从框架设计的视角看，如果有一套能兼容常见模板引擎的方案就好了。所幸，真有这样的方案：Consolidate 多模板引擎集成库。

Consolidate 可以适配十多种常见的 Node.js 模板引擎，统一了这些模板引擎的配置和使用接口，因此 TypeSpeed 框架选择 Consolidate 这个极具性价比的方案。

注意：Consolidate 库的作者是 T.J. Holowaychuk。他在 JavaScript 和 Go 开发领域都非常高产，他是 ExpressJS、Koa(与前者齐名的 Web 框架)等数百个优质开源项目的作者。本书第 6 章在介绍框架脚手架时还会用到他的另一个作品：Commander.js。

3.5.6 集成多模板引擎库

接下来框架将集成 Consolidate 作为模板引擎的支持,同时集成的还有 Mustache 模板引擎。Mustache 是一个十分精简的模板引擎,以简单的语法和广泛的语言支持著称。

首先需要安装上述两个库,命令如下:

```
npm install consolidate mustache
```

Consolidate 能够适配各种模板引擎,但是它本身并不是模板引擎,所以具体使用某个引擎时仍需要同时安装此模板引擎。

Consolidate 的作用是连接框架和 Mustache,如果开发者希望使用其他的模板引擎,则可以通过修改配置的方式,让 Consolidate 接入其他的模板引擎。

模板引擎的集成应该是在路由方法设置之前,即在 ExpressServer 类的 setRouter()调用之前,加入 app.engine()方法的调用即可,代码如下:

```
//chapter03/05-template/src/default/express-server.class.ts

import * as express from "express";
import * as consolidate from "consolidate";
import ServerFactory from "../factory/server-factory.class";
import { setRouter } from "../route-mapping.decorator";
import { bean, log } from "../speed";

export default class ExpressServer extends ServerFactory {
    //提供 Web 服务对象
    @bean
    public getSever(): ServerFactory {
        const server = new ExpressServer();
        server.app = express();
        return server;
    }

    //设置中间件
    public setMiddleware(middleware: any) {
        this.middlewareList.push(middleware);
    }

    //启动服务
    public start(port: number) {
        this.middlewareList.forEach(middleware => {
            this.app.use(middleware);
        });
        this.setDefaultMiddleware();
        this.app.listen(port, () => {
            log("server start at port: " + port);
        });
    }
```

```
        //设置默认中间件
        private setDefaultMiddleware() {
            //模板配置
            const viewConfig = {
                "engine": "mustache",
                "path": "/test/views",
                "suffix": "html"
            };
            this.app.engine(viewConfig["suffix"], consolidate[viewConfig["engine"]]);
            this.app.set('view engine', viewConfig["suffix"]);
            this.app.set('views', process.cwd() + viewConfig["path"]);

            setRouter(this.app);
        }
    }
```

上述代码把 setRouter()的调用从 start()方法里抽离出来，用 setDefaultMiddleware()方法来代替，而 setDefaultMiddleware()方法的内容用于设置模板引擎和 setRouter()，这样的处理方便未来可以给 Web 服务增加更多的内置服务，而不影响 start()方法的可读性。

viewConfig 是模板引擎的配置，其配置项有 3 个，分别如下。

(1) engine：模板引擎名称，该名称将被输入 Consolidate 库来取得适配的模板引擎。

(2) path：模板目录路径，使用 app.set('views',path)设置模板目录的路径。

(3) suffix：模板文件后缀。

FirstPage 类增加/first/renderTest 页面来测试上述的集成结果，代码如下：

```
//chapter03/05 - template/test/first - page.class.ts

@GetMapping("/first/renderTest")
public renderTest(req: any, res: any) {
    res.render("index", {name:"zzz"});
}
```

启动服务，打开浏览器访问 http://localhost:8080/first/renderTest 即可看到 Mustache 渲染处理的页面。

3.5.7　小结

本节讲解了 Web 开发领域重要的设计模式之一：MVC 模式，而响应处理是 MVC 模式的视图层，负责处理页面的显示。

响应处理有两种主要类型，即 JSON 格式输出和模板引擎。

JSON 格式输出能够将页面方法返回的对象自动转换为 JSON 数据输出，方便开发者使用，并且页面方法作为一个有着清晰的输入/输出类型的函数，也提升了其可测试性。

模板引擎是 Web 开发领域早期较为流行的开发技术之一，本节也介绍了 ExpressJS 模板引擎的使用，以及实现框架集成多模板引擎适配库 Consolidate。


24min

3.6 使用中间件增强框架功能

ExpressJS的设计非常精妙,它具有最小化的内核系统,仅提供路由方法和中间件机制。一个ExpressJS应用程序可以看作一系列中间件的聚合程序。

注意:ExpressJS的文档将中间件按作用和来源划分为应用级、路由级、错误处理、内置和第三方等5类中间件,这5类中间件在开发上几乎没有区别。

本节将在TypeSpeed框架里集成以下与路由相关的常用功能,这些功能都是使用第三方中间件进行整合而成的。

(1) 静态资源服务。

(2) 站点图标功能。

(3) GZip传输压缩。

(4) Cookie。

(5) Session。

同时,框架还会配合第1章实现的@value配置装饰器来对这些功能进行配置。默认情况下这些功能不会自动开启,开发者可按需进行配置使用。

3.6.1 静态资源服务

静态资源也是前端资源,例如HTML文件、前端JavaScript、图片、字体等。

通常前后端分离的项目会将前端资源单独存放到另一个服务器和域名上,用Nginx服务器来提供资源访问,而Web框架本身也可以作为资源服务器使用,开发者可以将项目前后端文件放到同一个Web服务器上,方便管理。

express.static()是ExpressJS内置的中间件,它能够提供简单的静态资源服务。

注意:express.static是ExpressJS内置的中间件,不需要用npm命令安装依赖。

TypeSpeed的静态资源服务基于express.static()进行集成,功能配置设计用节点static作为静态资源的配置项,其值是静态资源的文件目录,代码如下:

```
//chapter03/06 - middleware/test/config.json


{
    …
    "static": "/static",
    …
}
```

上述配置/static是静态资源存放的目录。在/static目录下已保存了一张用于演示的图片k.jpg。

在ExpressServer类增加static成员变量，用@value装饰器给该变量注入配置值，代码如下：

```
//chapter03/05-template/src/default/express-server.class.ts


@value("static")
private static: string;
```

接着在setDefaultMiddleware()方法中加入设置express.static()中间件的代码，使用前会先检查配置是否存在，代码如下：

```
//chapter03/05-template/src/default/express-server.class.ts


if(this.static) {
    const staticPath = process.cwd() + this.static;
    this.app.use(express.static(staticPath))
}
```

启动程序，使用浏览器访问http://localhost:8080/k.jpg，可以看到图片被显示出来了。

图3-17　站点图标

3.6.2　站点图标功能

站点图标可以显示在浏览器的窗口标签和收藏夹里，站点图标能提升一个网站的用户认知度，尤其是在打开过多的浏览器窗口后，用户只能根据标签上显示的站点图标来切换窗口，如图3-17所示。

站点图标的文件名是favicon.ico，浏览器在默认情况下会在域名根目录取得该图标，例如https://www.baidu.com/favicon.ico。

注意：将站点图标存放在3.6.1节静态资源目录也可以正常显示。

站点图标功能使用了serve-favicon中间件，安装serve-favicon的命令如下：

```
npm install serve-favicon
```

然后把准备好的图标文件favicon.ico存放在程序的根目录，将favicon设定为站点图标，代码如下：

```
//chapter03/06-middleware/test/config.json


{
```

```
    …
    "favicon" : "/favicon.ico",
    …
}
```

在 ExpressServer 类增加变量以读取配置：

```
//chapter03/05 - template/src/default/express - server.class.ts


@value("favicon")
private favicon: string;
```

在 setDefaultMiddleware()方法里增加对应的代码，同样先对配置进行检查，然后用 app.use()载入 serveFavicon 中间件，代码如下：

```
//chapter03/05 - template/src/default/express - server.class.ts


if(this.favicon) {
    const faviconPath = process.cwd() + this.favicon;
    this.app.use(serveFavicon(faviconPath));
}
```

3.6.3 传输压缩实现

优化页面加载速度是 Web 开发工作时常遇到的一个挑战。优化页面速度的方法通常围绕资源的加载和页面渲染两个方面进行。资源的压缩传输便是其中一种常规的优化手段。

基于 HTTP 协议的资源压缩传输方案有 gzip、deflate，Brotli 等，其中 gzip 传输压缩最为普遍。

gzip 采用 zip 压缩算法，其原理是将准备传输的内容，在服务器端先进行 zip 压缩再传输，浏览器接收到内容后进行 zip 解压再显示。

经过压缩的传输内容比压缩前体积减小了很多，传输因此而快不少，尤其是文本文件的 zip 压缩率较高，诸如 HTML、JavaScript 等文件的传输速度优化效果明显。

gzip 压缩中间件选用的是 compression 库，安装命令如下：

```
npm install compression
```

压缩功能的配置项为 compression，与前面两个中间件不同，compression 的配置是对象，其 level 项表示压缩率，取值为 0～9，9 为最大压缩比，代码如下：

```
//chapter03/06 - middleware/test/config.json


{
    …
```

```
    "compression": {
      "level" : 9
    },
    …
}
```

ExpressServer 类增加配置并用 app.use()载入中间件，代码如下：

```
//chapter03/05 - template/src/default/express - server.class.ts


@value("compression")
private compression: object;

…

if(this.compression) {
  this.app.use(compression(this.compression));
}
```

启动程序后测试，使用浏览器访问 http://localhost:8080/first。打开 Chrome 浏览器的开发者工具，进入网络(Network)一栏，发现本次请求的响应标头并没有 gzip 字样，表示这次请求并没有使用 gzip 压缩进行传输。为什么 gzip 压缩没有生效？

检查 compression 库源码，发现它要求页面输出至少 1024 字节，才会进行 gzip 的压缩处理。

为了测试 1024 字节的页面输出，测试页面增加了内容，当内容增加到 1023 字节时，响应标头依然没有 gzip 字样，这时页面的传输大小为 1.3KB，如图 3-18 所示。

而当输出的内容达到 1024 字节时，终于可以看到响应标头 Content-Encoding：gzip，该字段代表响应页面已使用 gzip 压缩，要求浏览器进行 zip 解压。此时页面的传输大小也变成了 383B，只有之前的三分之一左右，如图 3-19 所示。

图 3-18　页面为 1023 字节的响应标头

图 3-19　页面超过 1024 字节的响应标头

3.6.4 Cookie

Cookie是浏览器保存用户信息的常用方案之一,相比其他浏览器保存信息的方案,Cookie有着简单易用,浏览器兼容性高的优点。

Cookie是HTTP协议的头信息(Header)的字段,它既可用于浏览器发往服务器端的请求,也可用于服务器端返回浏览器的响应。

Cookie字段的格式是键-值对(Key-Value)字符串,各键-值对之间用分号分隔,键和值之间用等号分隔,如图3-20所示。

图3-20 请求标头的Cookie信息

Cookie功能使用的中间件是cookie-parser库,安装命令如下:

```
npm install cookie-parser
```

Cookie的配置设定为Cookie项,其配置同样是对象格式,代码如下:

```
//chapter03/06-middleware/test/config.json


{
    …
    "cookie": {
        "secret": "catme",
        "options": {}
    },
    …
}
```

配置的secret字段是加密字符串,要求每个网站都设置不同的secret随机字符串。options用于配置Cookie的存放路径和统一过期时间等,如果options为空,则遵循浏览器对Cookie的默认设定。

ExpressServer类使用app.use()方法载入cookie-parser中间件,并进行配置,代码如下:

```
//chapter03/05 - template/src/default/express - server.class.ts

@value("cookie")
private CookieConfig: object;


this.app.use(CookieParser(this.cookieConfig["secret"] || undefined, this.cookieConfig
["options"] || {}));
```

Cookie 可以存在于请求或者响应的头信息，它具有赋值和读取两个功能，因此这里用了两个页面进行测试，代码如下：

```
//chapter03/05 - template/test/second - page.class.ts


@onClass
export default class SecondPage {

    @GetMapping("/second/setCookie")
    setCookiePage(req, res) {
        res.cookie("name", "zzz");
        return "setCookie";
    }

    @GetMapping("/second/getCookie")
    getCookiePage(req, res) {
        const CookieName = req.cookies.name;
        return "getCookie: " + CookieName;
    }
}
```

页面/second/setCookie 里，用 res.cookie()方法给 Cookie 进行赋值，该页面的 Cookie 是响应给浏览器的，因此赋值操作的对象是 Response。

而/second/getCookie 页面使用 Request 对象来读取请求中的 Cookie 信息，代码是 req.cookies.name，其中 name 对应的是前面赋值 Cookie 的名称。

接下来启动服务进行测试，访问 http://localhost:8080/second/setCookie 可以看到浏览器收到的响应头信息已经带上 Set-Cookie 字段，该字段的内容对应的就是 res.cookie()方法的赋值，如图 3-21 所示。

然后访问读取 Cookie 页面 http://localhost:8080/second/getCookie，可以看到在请求头信息里有前面赋值的 name 值，而且/second/getCookie 页面也显示了这个值，如图 3-22 所示。

至此，框架已经支持对 Cookie 的赋值和读取。接下来将讲解经常和 Cookie 一起出现的 Session。

图 3-21 setCookie 页面的响应标头

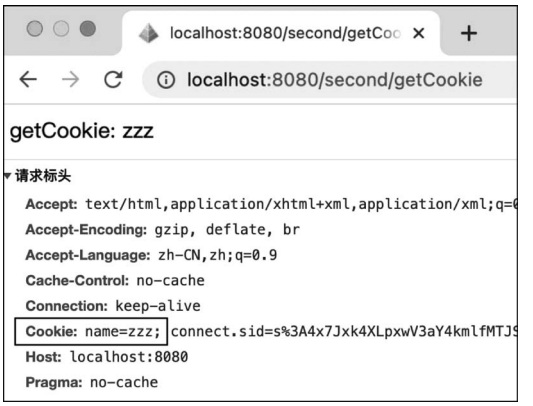

图 3-22 getCookie 页面的请求标头

3.6.5 Session

HTTP 协议是无状态(Stateless)的网络协议。无状态协议的特点是把每个请求都看作一次独立的事务,与之前的请求无关。无状态就意味着服务器端在协议层面无法确定任何两次请求是不是由同一个浏览器发出的。

因此,服务器端程序就需要从协议内容入手,采取一些手段辨别请求是否属于同一个浏览器或同一个访问用户。

注意:这里提到协议本身没有记录状态,但不代表协议传输的内容里没有记录这类信息。

最常见的辨别请求的方法就是 Cookie 加 Session 的组合,辨别的过程如下:

(1) 当用户在浏览器填写完登录信息后将信息发送给服务器端,服务器端将对资料进行检查,确认登录信息合法后会生成一串密钥,先把用户信息和密钥存到 Session 里,再把密钥设置到响应头的 Cookie 字段下发给浏览器。

(2) 浏览器获得 Cookie 字段的密钥后,接下来的每次请求都会在 Cookie 字段带上密钥信息。

(3) 当服务器端接收这些请求时,从 Cookie 字段取得密钥并与 Session 存储的值进行比对,如果比对成功,则确认这次的请求是该登录用户发起的,从 Session 里取得用户信息来处理业务逻辑。

通常,像这样的一系列请求-响应过程被称为一次会话。

从实现原理讲,Session 是一个运行在服务器端的小型数据库,它存储的格式是键-值对形式,每次登录操作成功后都会将用户信息存入,信息的键就是密钥,密钥会随着 Cookie 发送到浏览器并存储,而后的请求都带上该 Cookie 信息以供服务器端 Session 进行比对。

注意：本节讲解的是Session中间件的使用，因此Session是直接通过变量共享存储的，而在日常开发中，更常见的是通过Redis、Memcache等服务器来存储Session。在本书的5.3.5节将介绍使用Redis存储Session的方案。

Session功能使用的中间件是express-session库，安装命令如下：

```
npm install express - session
```

Session的配置项是session，配置格式是对象，代码如下：

```
//chapter03/06 - middleware/test/config.json

{
    …
    "session": {
        "trust proxy": false,
        "secret": "keyboard cat",
        "resave": false,
        "saveUninitialized": true,
        "cookie": {
            "secure": false
        }
    }
        …
}
```

Session的配置比较多，较为关键的有以下两项：

(1) trust proxy表示是否信任网关，如果Web程序使用了Nginx等反向代理，则需要将trust proxy设置为true。

(2) secret表示加密密钥，开发者必须给每个应用程序都设置不同的随机密钥，以确保Session不会在浏览器端被暴力破解。

ExpressServer类使用app.use()方法载入express-session中间件，代码如下：

```
//chapter03/05 - template/src/default/express - server.class.ts


@value("session")
private session: object;


if(this.session) {
    const sessionConfig = this.session;
    if(sessionConfig["trust proxy"] === 1){
        this.app.set('trust proxy', 1);
    }
    this.app.use(expressSession(sessionConfig));
}
```

上述代码有个细节,如果将 trust proxy 配置为 true,则还需要对 ExpressJS 进行设置,即需要将 trust proxy 设置为 1。

测试页面用 Session 来做个简单的计数器,每次访问该页面都会给计数器加一,并显示计数器的当前值,代码如下:

```
//chapter03/05 - template/test/second - page.class.ts

export default class SecondPage {
    @GetMapping("/second/testSession")
    testForSession(req, res) {
        req.session.view = req.session.view ? req.session.view + 1 : 1;
        return "testForSession: " + req.session.view;
    }
}
```

运行程序,每次刷新浏览器访问 http://localhost:8080/second/testSession 时页面显示的数字都会加一,即便是分开两个浏览器访问,该数字也是累计的。这也证明了 Session 可以跨多浏览器使用的特点,如图 3-23 所示。

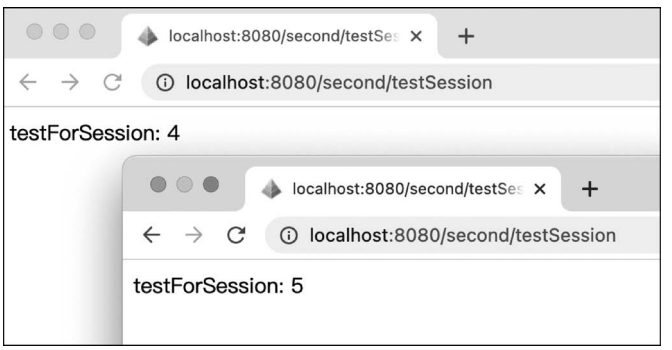

图 3-23 访问页面数字会不断累加

3.6.6 小结

本节介绍了如何将多个路由相关中间件集成到框架中的过程,其中涵盖了许多 Web 程序开发的基础知识,掌握这些知识可以让开发者更高效地开发 Web 程序,提升开发效率。

3.7 文件上传

文件上传是 Web 项目较常见的开发需求之一,本节将介绍文件上传的原理,以及如何将文件上传功能集成到框架里。

3.7.1 文件上传原理

文件上传是通过 HTTP 协议进行传输的。文件上传的过程粗略分为以下 4 步:

(1) 用户单击页面的上传表单域并选择文件,表单域取得了上传文件的路径。

(2) 表单提交时,根据路径读取文件二进制内容并附加到 HTTP 请求的请求体里,将请求发送到服务器端程序。

(3) 服务器端程序发现请求头类型为 Content-Type: multipart/form-data,便会解析请求体内容,将上传文件的二进制重新写成文件存放到临时目录里。

(4) 业务程序读取临时文件并做相应的业务处理。

从上述过程可以看出,上传文件的请求和普通 HTTP 请求的区别在于请求头类型及附加了文件内容。

请求头类型在 HTML 的 form 表单域设置,代码如下:

```
<form action = "/upload" enctype = "multipart/form - data" method = "post">
    < input type = "text" name = "title">< br >
    < input type = "file" name = "upload" multiple = "multiple">< br >
    < input type = "submit" value = "Upload">
</form >
```

请求体里文件内容的结构是依据请求头的 boundary 字段来区分的,代码如下:

```
Content - Type: multipart/form - data; boundary = ----
WebKitFormBoundaryOejulrdFZFEOU4el
```

boundary 字段表示这是一个表单,它是这个表单的分隔符,如图 3-24 所示。

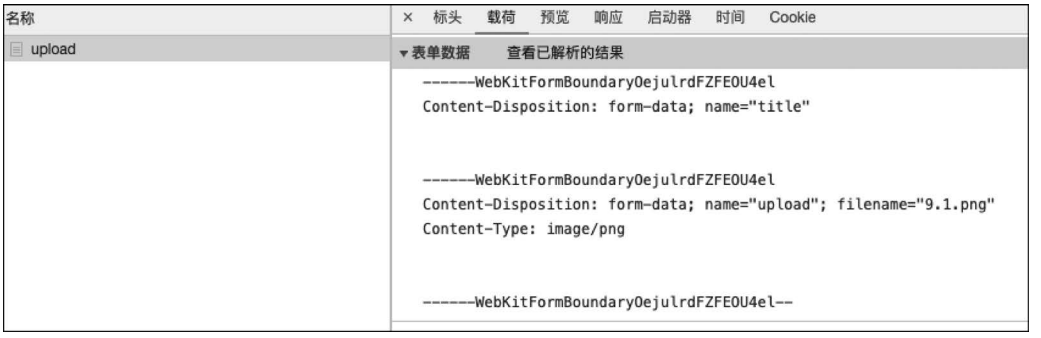

图 3-24　上传文件的 HTTP 请求体

从图 3-24 可以看到,boundary 的值被分隔成 3 段内容:

(1) 输入框,名称为 title。

(2) 文件域,名称是 upload,文件名为 9.1.png,文件类型是 image/png。

(3) 该 PNG 文件的二进制内容。

服务器端程序收到此次请求,解析请求体内容并将二进制内容保存成临时文件,得到文件上传的数据,代码如下:

```
{
  fieldName: 'upload',
  originalFilename: '9.1.png',
```

```
    path:
    '/var/folders/_d/czpf83d16610sd0_1tqzm09h0000gn/T/eL4nhdPZ1BPFRFAj - u1QGvjm.png',
    headers: {
        'content - disposition': 'form - data; name = "upload"; filename = "9.1.png"',
        'content - type': 'image/png'
      },
      size: 27485
  }
```

3.7.2 使用文件上传库

文件上传的原理是依据请求头 boundary 字段,对请求体的表单内容进行解析,因此,在框架内集成文件上传功能,也就是选用文件解析库并且将其内置到 Web 服务里。

这里选择的文件上传解析库是 multiparty。在 NPM 网站上类似的解析库有很多,multiparty 相对成熟,使用简单。

测试文件上传的 HTML 页面,可以用 3.3 节的模板引擎功能来渲染,代码如下:

```
//chapter03/07 - upload/test/views/upload.html

< html >
< head >
</head >
< body >
    < form action = "/upload" enctype = "multipart/form - data" method = "post">
        < input type = "text" name = "title">< br >
        < input type = "file" name = "upload" multiple = "multiple">< br >
        < input type = "submit" value = "Upload">
    </form >
</body >
</html >
```

SecondPage 增加了 form()方法,用于显示 upload.html 上传表单页面,还增加了 upload()方法,用于接收文件并上传。在 upload()方法使用 multiparty 获取上传文件的信息,代码如下:

```
//chapter03/07 - upload/test/second - page.class.ts


@PostMapping("/upload")
public upload(req, res) {
    const form = new multiparty.Form();

    form.parse(req, (err, fields, files) => {
        res.writeHead(200, { 'content - type': 'text/plain' });
        res.write('received upload:\n\n');
        log(files);
        res.end(util.inspect({ fields: fields, files: files }));
    });
```

```
}

@GetMapping("/form")
form(req, res) {
    res.render("upload");
}
```

运行程序，使用浏览器访问 http://localhost:8080/form 以显示上传表单，当选择一个文件上传时，即可看到命令行输出了上传文件的信息，如图 3-25 所示。

图 3-25　文件上传表单和结果输出

3.7.3　实现文件上传装饰器

文件上传过程是使用解析库来解析请求，提供文件上传信息以供开发者使用。那么，是否能开发一个@upload 装饰器，让开发者有选择性地在需要上传解析时装饰在页面方法上，只对当前页面的请求进行解析，既节省系统开销，也符合框架的设计风格。

按需解析上传的逻辑是将 form.parse()这段代码移到路由装饰器的位置来执行，而@upload 装饰器的主要作用是对页面方法进行标注，从而可以判断是否需要执行解析，代码如下：

```
//chapter03/07 - upload/src/route - mapping.decorator.ts

const uploadMapper = {}
//收集上传页面
function mapperFunction(method: string, value: string) {
  return (target: any, propertyKey: string) => {
    routerMapper[method][value] = (req, res, next) => {
      if (uploadMapper[target.constructor.name + "#" + propertyKey]) {
        uploadMapper[target.constructor.name + "#" + propertyKey](req, res, next);
      }

...
```

```
//对上传请求进行解析处理
function upload(target: any, propertyKey: string) {
  uploadMapper[target.constructor.name + "#" + propertyKey] = (req, res, next) => {
    if (req.method === 'POST') {
      const form = new multiparty.Form();
      log("upload start");
      form.parse(req, function (err, fields, files) {
        req.files = files;
        log("upload end");
        next();
      });
    };
  }
}
```

@upload 装饰器将当前的类和方法作为键,将带有 form.parse()代码的中间件作为值,存到全局变量 uploadMapper 中,这也是多种方法装饰器共同装饰一种方法时常用的技巧。

存入的中间件和 3.7.2 节的代码类似,用 form.parse 对 Request 对象进行解析,取得上传文件的数据并赋值给 req.files,之后调用 next()进入下一个中间件,下一个中间件即页面方法,这样就能取得上传文件信息了。

@upload 装饰器和 res.files 的测试页面,代码如下:

```
//chapter03/07-upload/test/second-page.class.ts


@PostMapping("/upload")
@upload
public upload(req, res) {
    const files = req.files;
    log(files);
    res.send("upload success");
}
```

执行程序,使用浏览器访问 http://localhost:8080/form,选择文件并提交后会发现命令行输出的 req.files 值是 undefined,也就是没有取得上传信息。

先检查 form.parse()代码是否对上传请求进行了解析。在 form.parse()代码里加入对 files 的打印输出,代码如下:

```
form.parse(req, function (err, fields, files) {
    req.files = files;
    log(files);
    log("upload end");
    next();
});
```

再次执行程序后上传文件,发现 form.parse()位置输出的 files 日志确实是有内容的,

如图 3-26 所示。

```
[LOG] 2022-12-18 10:51:49 route-mapping.decorator.ts:46 () {
  upload: [
    {
      fieldName: 'upload',
      originalFilename: 'k.jpg',
      path: '/var/folders/jv/z3x8y3k95r5473y9jq7c0t6r0000gn/T/afrdKci30EdNY0Z5ucsLlHpE.jpg',
      headers: [Object],
      size: 1380
    }
  ]
}
```

图 3-26　上传文件的检查输出

从输出看，日志顺序似乎有点问题，继续增加日志输出来检查，代码如下：

```
function upload(target: any, propertyKey: string) {
  uploadMapper[target.constructor.name + "#" + propertyKey] = (req, res, next) => {
    if (req.method === 'POST') {
      const form = new multiparty.Form();
      log("upload start");
      form.parse(req, function (err, fields, files) {
        req.files = files;
        log(files);
        log("upload end");
        next();
        log("upload next end");
      });
    };
  }
}

...

@PostMapping("/upload")
@upload
public upload(req, res) {
    log("page start");
    const files = req.files;
    log(files);
    res.send("upload success");
    log("page end");
}
```

增加更多输出后的日志，如图 3-27 所示。

从输出可以看到，form.parse()里的日志输出在页面方法执行结束后，这就意味在 form.parse()里调用的 next()没有起到作用，所以执行的顺序错乱了。

经过进一步的测试，发现 next()只有写在中间件里才能正常运作，而前面的写法并不是写在中间件里，而只是将 form.parse()当作页面逻辑的一部分，所以 next()没有生效。

```
[LOG] 2022-12-18 10:59:08 route-mapping.decorator.ts:43 (Object.uploadMapper.<computed> [as SecondPage#upload]) upload start
[LOG] 2022-12-18 10:59:08 second-page.class.ts:29 (SecondPage.upload) page start
[LOG] 2022-12-18 10:59:08 second-page.class.ts:31 (SecondPage.upload) undefined
[LOG] 2022-12-18 10:59:08 second-page.class.ts:33 (SecondPage.upload) page end
[LOG] 2022-12-18 10:59:08 route-mapping.decorator.ts:46 () {
  upload: [
    {
      fieldName: 'upload',
      originalFilename: 'k.jpg',
      path: '/var/folders/jv/z3x8y3k95r5473y9jq7c0t6r0000gn/T/i-itD6b1mxLv12vVaLg4Ztlr.jpg',
      headers: [Object],
      size: 1380
    }
  ]
}
[LOG] 2022-12-18 10:59:08 route-mapping.decorator.ts:47 () upload end
[LOG] 2022-12-18 10:59:08 route-mapping.decorator.ts:49 () upload next end
```

图 3-27　详细的上传文件日志输出

注意：在使用 JavaScript/TypeScript 编程时，next()的写法是避免回调顺序错乱的技巧。只要 next()在适当的位置(例如回调函数或者 Promise)使用，理论上就能解决顺序问题。

解决的方案是将上传逻辑抽离出来，使其变成中间件。这里先来了解 ExpressJS 中间件的另一种用法：将一个或多个中间件写在路由方法的参数里，这些中间件只针对当前的路由请求生效，代码如下：

```
//中间件针对特定路由生效的伪代码
app.get("/index", 中间件 1, 中间件 2, ..., function(req, res){
    ...
});
```

和前面的中间件的用法做一个对比，app.use()载入的中间件是全局范围生效的，可以理解成任意的请求都会经过 app.use()的中间件进行处理，代码如下：

```
//普通中间件的伪代码
app.use(中间件 1);
app.use(中间件 2);
...
```

现在，将 form.parse()部分代码抽离为独立中间件 uploadMiddleware()，并配置在路由方法的参数里，查看它的 next()是否生效，代码如下：

```
function uploadMiddleware(req, res, next) {
  const form = new multiparty.Form();
  form.parse(req, (err, fields, files) => {
    req.files = files["upload"] || undefined;
    next();
  });
}
```

而@upload 装饰器用来标识哪个路由方法需要使用 uploadMiddleware()，代码如下：

```
function upload(target: any, propertyKey: string) {
  uploadMapper.push(target.constructor.name + "#" + propertyKey)
}
```

接着 setRouter()方法匹配是否启用 uploadMiddleware()，代码如下：

```
if (method === "post" && uploadMapper.includes(rounterFunction["name"])) {
    app[method](key, uploadMiddleware, rounterFunction["invoker"]);
} else {
    app[method](key, rounterFunction["invoker"]);
}
```

修改程序并运行，再对上传文件进行测试。发现这次上传文件的信息可以在/upload 页面正常输出，如图 3-28 所示。

```
[LOG] 2022-12-18 14:08:05 second-page.class.ts:30 (SecondPage.upload) [
  {
    fieldName: 'upload',
    originalFilename: 'k.jpg',
    path: '/var/folders/jv/z3x8y3k95r5473y9jq7c0t6r0000gn/T/WQPYyUwPHAtKaeT_TuGvLs8-.jpg',
    headers: {
      'content-disposition': 'form-data; name="upload"; filename="k.jpg"',
      'content-type': 'image/jpeg'
    },
    size: 1380
  }
]
[LOG] 2022-12-18 14:08:05 second-page.class.ts:31 (SecondPage.upload) uploaded
```

图 3-28　上传文件的正确输出

3.7.4　小结

本节实现文件上传装饰器@upload，它使用了 app.use()之外另一种 ExpressJS 中间件的使用方法，这种方法将中间件放到路由方法的参数列表，中间件的作用范围仅限于该路由方法内生效。

@upload 在实现时使用了全局变量 uploadMapper 来收集装饰器对应的方法，提供给@GetMapping 等路由装饰器进行判定使用，这是多种方法装饰器协作的编程方法，读者务必留意。

3.8　Web 服务鉴权

Web 服务鉴权是针对请求的访问权限进行检查的能力，也就是通常所讲的网站登录和权限验证等相关功能。

3.8.1 实现基本访问认证

Basic Authentication 是最基础的权限验证方案,是早期的 HTTP 协议内置的权限认证方案,因此 Basic Authentication 在 HTTP 协议有专用的状态码 401,401 状态码指示下一步操作必须提供 Basic Authentication 验证。

Basic Authentication 的定义:在 HTTP 协议中,基本认证(Basic Access Authentication)是允许 HTTP 用户浏览器在请求时提供用户名和密码的一种方式。在进行基本认证的过程里,请求的 HTTP 头字段会包含 Authorization 字段,形式为 Authorization: Basic <凭证>,该凭证是用户和密码组合的 Base64 编码字符串。

Basic Authentication 的中间件库是 express-basic-auth,安装命令如下:

```
npm install express - basic - auth
```

在 3.1.4 节实现了给 ExpressJS 设置中间件的 setMiddleware()方法,这里通过 setMiddleware()方法来集成 Basic Authentication 中间件,代码如下:

```
//chapter03/02 - routes/test/main.ts


public main(){
    this.server.setMiddleware(basicAuth({
        users: { 'admin': 'supersecret' }
    }));
    this.server.start(8080);
    log('start application');
}
```

执行程序,访问任意页面都可正常显示,没有发现 401 状态码或者其他错误提示。显然,Basic Authentication 没有生效。

在 start()方法输出检查 this.middlewareList,检查 setMiddleware()方法是否正常设置了 Basic Authentication 中间件,代码如下:

```
//chapter03/02 - routes/test/main.ts


public start(port: number) {
    log(this.middlewareList);
    this.middlewareList.forEach(middleware => {
        this.app.use(middleware);
    });
    this.setDefaultMiddleware();

    ...
```

执行程序,观察输出,此时可发现 this.middlewareList 是空数组,setMiddleware()执行前后两个 this.server 指代的对象不同,出现了 3.3.3 节相同的问题:this 指向错误,因此才

会出现 setMiddleware()无效的情况。

引起该问题的原因是 BeanFactory 存入对象的构造方法函数，只有再取出时才进行调用实例化，因此每次取得的都是重新创建的对象，从而导致使用时上下文的 this 不能指向同一个对象。

至此，对象工厂需要进行一些优化，BeanFactory 增加两种方法，即 putObject() 和 getObject()，代码如下：

```
//chapter03/08-auth/src/bean-factory.class.ts


export default class BeanFactory {
    private static beanMapper: Map<string, any> = new Map<string, any>();
    private static objectMapper: Map<string, any> = new Map<string, any>();

    public static putBean(mappingClass: Function, beanClass: any): any {
        this.beanMapper.set(mappingClass.name, beanClass);
    }

    public static getBean(mappingClass: Function): any {
        return this.beanMapper.get(mappingClass.name);
    }

    public static putObject(mappingClass: Function, beanClass: any): any {
        this.objectMapper.set(mappingClass.name, beanClass);
    }
    public static getObject(mappingClass: Function): any {
        return this.objectMapper.get(mappingClass.name);
    }
}
```

至此 BeanFactory 拥有以下 4 个静态方法。

(1) putBean()：存入的参数由类名 target、方法名 propertyKey、函数执行结果 factory 等三者组成。用于存入对象标识，但未实例化。

(2) getBean()：获取 putBean()存入的内容。

(3) putObject()：存入实例化后的对象。

(4) getObject()：获取 putObject()保存的对象。

这样对象工厂就能分别应对两类场景：

(1) 取出对象再进行实例化。

(2) 直接取用已实例化的对象。

改进 BeanFactory 后执行程序，访问任意页面都会提示 HTTP 401 错误，401 错误意味着需要进行 Basic Authentication 验证。

测试验证可以使用 Postman 工具，在 Postman 请求设置 Basic Auth 认证，填入前面 main.ts 代码内的用户信息。单击发起请求可以发现通过验证，正常返回页面，如图 3-29 所示。

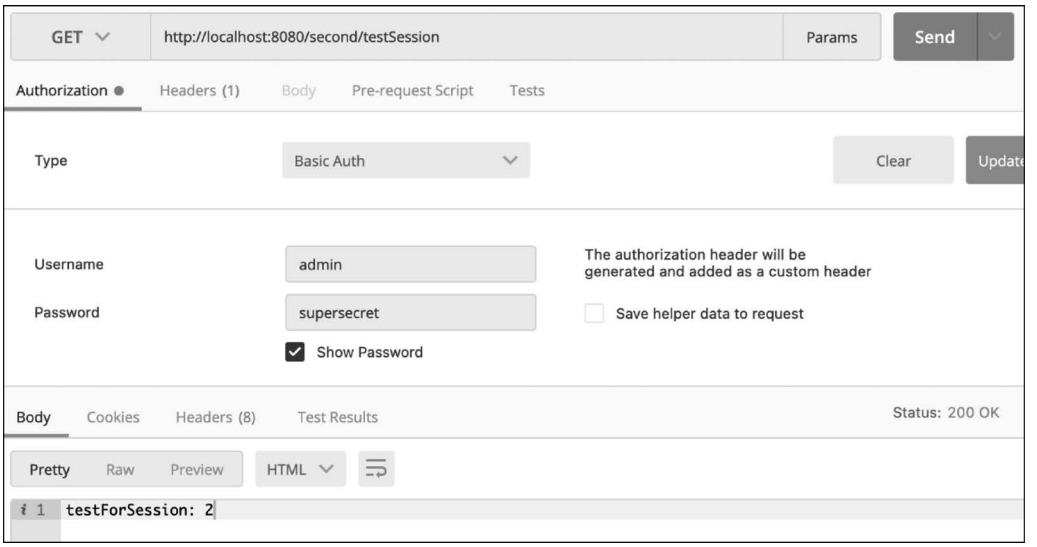

图 3-29　Postman 测试 Basic Authentication 验证

注意：Postman 可以对接口进行调试，对 API 发起 GET、POST 等测试请求，支持编辑头信息和参数、上传文件等功能。同类工具还有 Apifox、PAW、JMeter 等。

3.8.2　实现验证装饰器

JWT(JSON Web Token)是基于 JSON 格式的一种网络程序认证方案，其遵循的标准为 RFC7519。对比 Cookie 等验证方案，JWT 有两个突出的优点：携带信息相对较多、具备独立认证特性。

(1) Cookie 以明文方式传输信息，其加密信息只能是少量的(如标识符等)关键信息，而 JWT 是密文传输，因此携带的信息可包含用户标识、用户角色和权限等较多的信息。

(2) Cookie 的用户信息等内容必须在服务器端通过标识符查询取得，而 JWT 自身携带了认证信息，因此 JWT 可以独立使用，不需要依赖于服务器存储额外的用户信息。

基于以上特性，JWT 特别适用于分布式网站的验证场景。

JWT 协议格式由 3 部分构成：

(1) 头部(Header)包含 JWT 声明类型和加密算法名称。

(2) 数据(Payload)存放有效信息，主体是令牌字符串和开发者自定义的字段信息，如用户 ID、权限标识等。

(3) 签名(Signature)对上述两部分数据分别用 Base64 编码后再使用加密算法加密得到的字符串，该字段确保了内容的有效性和完整性。

框架在实现 JWT 功能时使用 express-jwt 中间件。@jwt 装饰器和 3.7 节的@upload 类似，同样是通过全局变量进行记录，代码如下：

```
//chapter03/08-auth/src/route-mapping.decorator.ts


function jwt(jwtConfig) {
  return (target: any, propertyKey: string) => {
    if (routerMiddleware[target.constructor.name + "#" + propertyKey]) {
      routerMiddleware[target.constructor.name + "#" + propertyKey].push(expressjwt
(jwtConfig));
    } else {
      routerMiddleware[target.constructor.name + "#" + propertyKey] = [expressjwt
(jwtConfig)];
    }
  }
}
```

与@upload 稍有不同的是，@jwt 装饰器需要输入参数，JWT 要用到参数进行配置。例如@jwt({ secret: "shhhhhhared-secret", algorithms: ["HS256"] })，这里 secret 是加密密钥，而 algorithms 是加密算法。

注意：JWT 方案最机密的信息是密钥，密钥可以解密程序中所有的 JWT 密文，因此每个应用程序必须有单独的、妥善保存的、唯一的密钥。

此时需要优化 3.7 节在路由中间件参数设置@upload 中间件的写法，直接在中间件参数上面硬编码，现在增加 JWT 中间件就需要重新修改框架，因此，这里采用了 JavaScript 内置的 apply()函数，apply()函数在调用函数时动态地增减参数，代码如下：

```
//chapter03/08-auth/src/route-mapping.decorator.ts


let rounterFunction = routerMapper[method][key];
if (routerMiddleware[rounterFunction["name"]]) {
     let args: Array < any > = [key, ...routerMiddleware[rounterFunction["name"]],
rounterFunction["invoker"]];
    app[method].apply(app, args);
} else {
    app[method](key, rounterFunction["invoker"]);
}
```

至此，@jwt 收集启用验证的页面，统一了@jwt 和@upload 设置中间件的写法，在 setRouter()方法进行批量调用的方法。

接下来在 SecondPage 测试页面修改原有的/form 页面，使只有带有 JWT 验证信息的请求才能打开该页面，代码如下：

```
//chapter03/08-auth/test/second-page.class.ts

@jwt({ secret: "shhhhhhared-secret", algorithms: ["HS256"] })
```

```
@GetMapping("/form")
form(req, res) {
    res.render("upload");
}
```

运行程序,访问 http://localhost:8080/form 时会出现未验证的错误提示,如图 3-30 所示。

图 3-30　要求 JWT 验证访问的页面

要进行 JWT 验证,首先需要生成 JWT 验证字符串,这里使用 jsonwebtoken 库,安装命令如下:

```
npm install jsonwebtoken
```

FirstPage 测试页面/login 假定是一个登录页面,它将输出验证字符串,代码如下:

```
//chapter03/08-auth/test/first-page.class.ts


@GetMapping("/login")
login() {
    const token = jwttoken.sign({ foo: 'bar' }, 'shhhhhhared-secret');
    return token;
}
```

jwttoken.sign()方法的第 2 个参数密钥和前面/form 页面的@jwt 参数密钥一致。这时/form 页面将@GetMapping 改为@PostMapping。

执行程序,访问 http://localhost:8080/login 可以得到一串验证字符串,如图 3-31 所示。

图 3-31　取得验证字符串

在 Postman 将请求方法设置为 POST，在 Headers 增加 Authorization 项，值为 Bearer＋空格＋验证字符串，如图 3-32 所示。

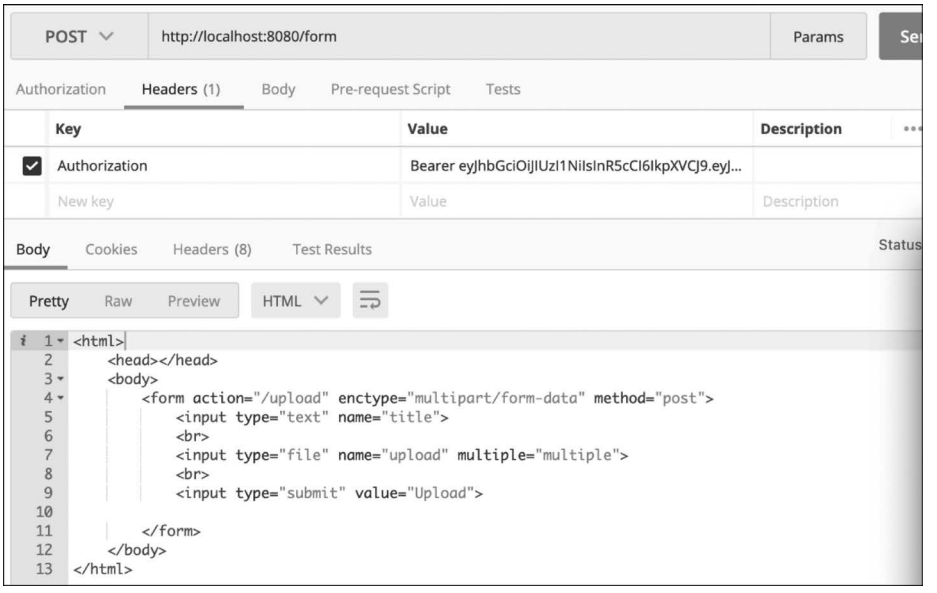

图 3-32　测试 JWT 验证

图 3-32 中请求已经成功地通过了 JWT 验证，并输出了 3.7 节的上传表单 HTML。

3.8.3　拦截器

Basic Authentication 和 JWT 装饰器都是针对单个页面方法的鉴权方式，在少量页面需要鉴权的网站使用比较方便，但是在网站大量页面需要统一鉴权的情景（如站点管理后台）就显得力不从心了。

页面单点鉴权采用的是黑名单逻辑，只对特定的页面进行检查，而像管理后台这种应用，要求全部页面都必须鉴权，只保留少数页面（如登录页，注册页等）不进行检查就是白名单逻辑，被称为全局访问鉴权。

按照全局访问鉴权的特性来设计框架的全局鉴权有以下两个要点：

（1）只有在白名单的页面才能豁免权限检查。

（2）其他任何页面，一律进行权限检查。

设想在任意页面方法前插入一个特定中间件对权限进行检查，期间跳过白名单的页面就能符合上述两点，从而实现全局鉴权。

在任意页面方法前执行的操作逻辑被称为拦截器（Interceptor）。

3.8.4　开发全局拦截器机制

拦截器机制是一个在页面方法前的代码执行阶段，其代码主要完成的工作有解析请求

参数、转换数据赋值、执行权限检查、调试程序异常等。

因此,框架需要提供这样的代码执行时机,让开发者能够自行实现各种拦截逻辑,其中就包括基于白名单的权限检查。

拦截器的实现沿用对象管理机制,方便开发者自行继承及扩展,代码如下:

```
//chapter03/08 - jwt/src/factory/authentication - factory.class.ts


import * as express from 'express';
export default abstract class AuthenticationFactory {
      public preHandle (req: express.Request, res: express.Response, next: express.
NextFunction): void{
         next();
    }
     public afterCompletion (req: express.Request, res: express.Response, next: express.
NextFunction): void{
         next();
    }
}
```

AuthenticationFactory 的两种方法 preHandle()和 afterCompletion()分别对应在页面方法前执行的前置中间件和在页面方法后执行的后置中间件。两种方法的默认实现都是不进行任何操作,调用 next()进入下一个中间件,因此,AuthenticationFactory 的默认对象非常简单,只需提供@bean 装饰器初始化,代码如下:

```
//chapter03/08 - jwt/src/default/default - authentication.class.ts


export default class DefaultAuthentication extends AuthenticationFactory{
    @bean
    public getAuthentication(): AuthenticationFactory {
        return new DefaultAuthentication();
    }
}
```

实现在页面方法前后起作用,即将 AuthenticationFactory 插入 setRouter()的代码前后,代码如下:

```
//chapter03/08 - jwt/src/default/express - server.class.ts

export default class ExpressServer extends ServerFactory {
      …
    @autoware
    public authentication: AuthenticationFactory;

    private setDefaultMiddleware() {
            …
            //配置前置处理器
```

```
            this.app.use(this.authentication.preHandle);

        if (this.static) {
            const staticPath = process.cwd() + this.static;
            this.app.use(express.static(staticPath))
        }
        setRouter(this.app);
        //配置后置处理器
        this.app.use(this.authentication.afterCompletion);

        ...
```

@autoware 装饰器先取得 AuthenticationFactory 对象，然后在 setRouter()的前后位置调用 AuthenticationFactory 的 preHandle()、afterCompletion()方法，从而实现拦截器逻辑。

preHandle()方法放置在静态文件中间件 express.static()的前面，使拦截器可以对图片、JS 等访问请求进行拦截，确保静态文件也在权限保护之下。

至此，框架完成了拦截器机制的开发，此时拦截器仅仅是默认实现，在实际使用中还需要扩展新的拦截器。

3.8.5　实现 JWT 全局拦截器

实现自定义的全局拦截器只要先继承于 AuthenticationFactory，然后用@bean 提供实例化对象即可，代码如下：

```
//chapter03/08-jwt/app/src/jwt-authentication.class.ts

import { AuthenticationFactory, bean, config } from "../../src/typespeed";
import express from "express";
import { expressjwt, GetVerificationKey } from "express-jwt";
import * as jwt from 'jsonwebtoken';

const jwtConfig: {
    secret: jwt.Secret | GetVerificationKey;
    algorithms: jwt.Algorithm[];
    ignore: string[];
} = config("jwt");

export default class JwtAuthentication extends AuthenticationFactory {
    //提供拦截器对象
    @bean
    public getJwtAuthentication(): AuthenticationFactory {
        return new JwtAuthentication();
    }
    //拦截前置处理器
      public preHandle(req: express.Request, res: express.Response, next: express.
NextFunction): void {
```

```
            if(!jwtConfig.ignore.includes(req.path)) {
                const jwtMiddleware = expressjwt(jwtConfig);
                //用 JWT 中间件进行检验
                jwtMiddleware(req, res, (err) => {
                    if (err) {
                        //next(err);
                    }
                });
            }
            next();
        }
    }
```

getJwtAuthentication()方法提供了 AuthenticationFactory 对象,框架的对象管理会用它覆盖 AuthenticationFactory 的默认实现,成为在 ExpressJS 路由方法使用的拦截器中间件。

JwtAuthentication 的 preHandle()方法首先判断当前访问路径 req.path 是否在 JWT 配置的白名单里,如果在白名单里,则略过权限检查并放行。JWT 配置的白名单 jwtConfig.ignore 是数组,内容可配置多个需要放行的路径地址。

if 判断的内部是权限判断的演示代码,用 express-jwt 库检查请求的 JWT 字符串信息。在实际使用时,还需要从 JWT 信息取出当前用户的其他信息,示例代码如下:

```
    public preHandle(req: express.Request, res: express.Response, next: express.NextFunction):
    void {
        if(!jwtConfig.ignore.includes(req.path)) {
            const jwtMiddleware = expressjwt(jwtConfig);
            jwtMiddleware(req, res, (err) => {
                if (err) {
                    next(err);
                }
                const checkIsUser = checkFromDatabase(req.auth?.user, req.auth?.token);
                if(checkIsUser){
                    req["user"] = req.auth?.user;
                }
            });
        }
        next();
    }
```

checkFromDatabase()示范使用数据库校验用户名和认证信息,用户名来自 JWT 存储的 auth 字段,如果校验成功,则对 req 赋值用户信息 req.user,其他页面即可用 req.user 作为用户信息进行业务处理。

上述 JWT 的配置内容用 config("jwt")方法取得,配置变量 jwtConfig 有 secret、algorithms、ignore 字段,分别表示密钥、加密算法和白名单数组。

3.8.6 小结

本节介绍了两类 Web 服务鉴权,即单个页面鉴权及全局拦截器。单个页面鉴权适用于

限制黑名单页面访问的场景，采用@jwt 装饰器来标记特定的机密页面，只有合法的请求才能访问这些页面。

相对而言，全局拦截器适用场景采用的是白名单逻辑，如后台管理系统。白名单逻辑只允许少量白名单页面能够自由访问系统，如登录页、注册页等，其余页面都需要进行鉴权。框架采用对象管理机制实现全局拦截器逻辑，方便开发者进行扩展，而 JWT 全局拦截器示范了利用全局拦截器进行 JWT 验证的例子。

3.9　服务器端错误输出

在 3.8.2 节介绍权限验证时，出现了如图 3-30 所示的验证错误提示，该错误提示较为原始，由于暴露了程序的堆栈信息，存在安全隐患，因此本节将介绍框架如何收敛这些错误提示，让用户只能看到友好的页面提示，并且让开发者能自行实现所需的提示内容。

3.9.1　捕捉常见错误

服务器端常见的错误提示有 404 和 500 错误。这里 404 和 500 指的是 HTTP 状态码。

(1) 404 Page Not Found，一般因为浏览器访问的页面不存在、路由没有正确配置或者页面引用了错误的图片和资源地址。

(2) 500 Internal Server Error，起因是服务器端程序出现错误情况，被动或者主动地将情况返回浏览器。

被动和主动的区别是服务器端程序是否针对错误进行处理。服务器端代码如果没有主动捕获错误而使程序崩溃信息直接输出到浏览器，则是被动情况，如图 3-30 所示的提示，而主动意味着服务器端捕获了错误并给浏览器返回友好的信息，这时 HTTP 状态码仍为 500，表示当前服务确实存在问题。

在中间件链上捕获 400 和 500 错误需要建立两个特殊的中间件，代码如下：

```
//chapter03/09-error/src/default/express-server.class.ts


setRouter(this.app);

this.app.use((req, res, next) => {
    res.status = 404;
    res.write("404 Not Found");
    res.end();
});

this.app.use((err, req, res, next) => {
    if (!err) {
        next();
    }
```

```
        res.status(err.status || 500);
        res.send("500 Server Error");
    });
```

在 setRouter()方法之后的第 1 个中间件捕获的是 400 错误。setRouter()可以匹配所有正确配置了页面路径的地址,当某个请求路径无法被 setRouter()匹配到时,请求的地址就是不正确的,即找不到页面。

这时程序会执行到 setRouter()后的第 1 个中间件,这个中间件特殊之处是它没有路径参数,因此它能捕获所有的错误路径,给浏览器返回 404 状态码和 Page Not Found 提示,如图 3-33 所示。

接下来第 2 个中间件是 500 错误处理,相比其他中间件,500 错误处理中间件有 4 个参数(err, req, res, next),其中第 1 个参数 err 是错误处理的关键。

中间件的错误处理规则是,当中间件链条上的任何一个中间件出错时都会直接跳过后续所有没有使用 err 参数的中间件,落在第 1 个带有 4 个参数的中间件,这时 err 参数带有具体的出错信息。

500 错误处理中间件会检查 err 值,如果发现 err 非空,则输出 err 携带的错误码,或者 500 错误码,并且给浏览器返回 500 Server Error 提示。

测试 500 错误处理,需准备一个抛出错误的页面,代码如下:

```
//chapter03/09 - error/test/second - page.class.ts


@GetMapping("/second/testError")
testError(req, res) {
    throw new Error('Test Error');
}
```

运行程序,访问 http://localhost:8080/second/testError 即可看到错误提示,如图 3-34 所示。

图 3-33 404 错误提示

图 3-34 500 错误提示

3.9.2 错误日志输出

在 500 错误测试例子里,抛出错误使用 throw new Error('Test Error')方式,这样的错误会被输出直接输出到系统命令行里,但无法被日志系统取得,因此,在日常开发中异常处理更多地会使用预定义的错误输出方法。

异常处理是编程的重点课题之一，异常指的是逻辑上存在的问题，但该问题并不影响系统的运作。例如，上传文件时发现上传的文件格式不正确，这时需要给浏览器返回预先定义的错误信息，同时在服务器端记录错误日志，使异常也有迹可循。

注意：通常在系统运行过程里，当同类的异常多次发生时，极有可能是代码有不符合期待的情况存在，这是值得重视的预兆，因此对错误日志的统一收集极其有必要。

框架设计了 log()函数作为一般信息的输出方法，下面增加 error()函数来作为错误日志统一记录的方法，代码如下：

```
//chapter03/09 - error/src/speed.ts


public error(message?: any, ...optionalParams: any[]) : void {
    this.logger.error(message, ...optionalParams);
}
```

error()函数使用 LogFactory 日志类的 error()方法，因此在 LogFactory 和它的子类 LogDefault、CustomLog 中也需要增加 error()方法，代码如下：

```
//chapter03/09 - error/src/default/log - default.class.ts


export default class LogDefault extends LogFactory {

    @bean
    createLog(): LogFactory {
        return new LogDefault();
    }

    public log(message?: any, ...optionalParams: any[]): void {
        console.log(message, ...optionalParams);
    }
    //增加错误捕获日志
    public error(message?: any, ...optionalParams: any[]) : void{
        console.error(message, ...optionalParams);
    }

}
```

在上述的 404 错误处理中间件测试 error()方法，观察其生效情况，代码如下：

```
this.app.use((req, res, next) => {
    error("404 not found, for page: " + req.url);
    res.status = 404;
    res.write("404 Not Found");
    res.end();
});
```

测试 404 页面可以看到 error()在命令行输出 ERROR 类型的日志，如图 3-35 所示。

```
[LOG] 2023-11-08 18:11:30 speed.ts:46 (onClass) decorator onClass: SecondPage
[LOG] 2023-11-08 18:11:30 speed.ts:46 (onClass) decorator onClass: TestLog
[LOG] 2023-11-08 18:11:30 test-log.class.ts:7 (new TestLog) TestLog constructor
[LOG] 2023-11-08 18:11:30 speed.ts:39 () main start
[LOG] 2023-11-08 18:11:30 express-server.class.ts:43 (ExpressServer.start) []
[LOG] 2023-11-08 18:11:30 main.ts:16 (Main.main) start application
[LOG] 2023-11-08 18:11:30 express-server.class.ts:49 (Server.<anonymous>) server start at port: 8080
[ERROR] 2023-11-08 18:11:45 express-server.class.ts:88 () 404 not found, for page: /unkown
```

图 3-35　ERROR 类型的日志

3.9.3　美化内置错误页面

在 3.9.1 节介绍了 404 和 500 错误的捕获，其页面输出比较粗糙，在实际开发中开发者还需要自行设计页面对 3.9.1 节中的页面进行替换，因此，本节将介绍框架如何提供美观的默认错误提示页面，方便开发者开箱即用。

1. 目录结构

首先框架选用了两个开源的错误提示页面，页面的样式、图片等均被内嵌在 HTML 文件中，能够直接显示，无须依托其他的外部资源。这两个页面放置在/static/error-page 目录，目录结构如图 3-36 所示。

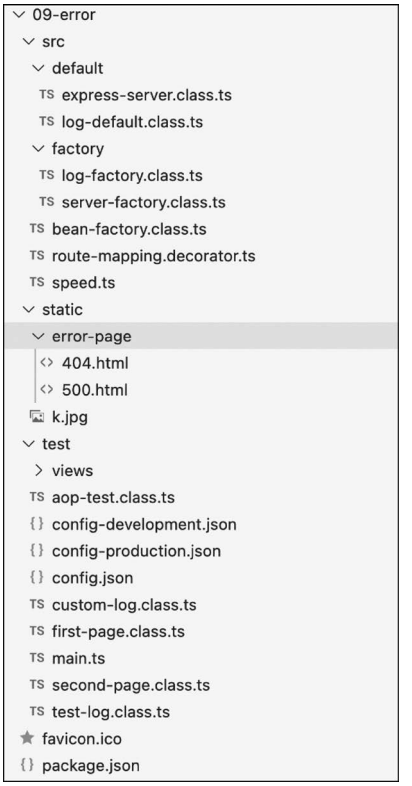

图 3-36　错误页面的目录结构

注意：这两个错误页面的源码都是基于 MIT 开源协议的，允许免费商用，具体协议可参考页面文件头部的注释内容。

2. 404 页面美化

修改 setDefaultMiddleware()方法 404 处理中间件，使其输出 404.html 文件，代码如下：

```
//chapter03/09-error/src/default/express-server.class.ts


//404 页面
this.app.use((req, res, next) => {
    error("404 not found, for page: " + req.url);
    if (req.accepts('html')) {
        res.render(process.cwd() + "/static/error-page/404.html");
    } else if (req.accepts('json')) {
        //当浏览器需要 JSON 格式时，返回 JSON 错误提示
        res.json({ error: 'Not found' });
    } else {
        //默认情况返回文本类型错误提示
        res.type('txt').send('Not found');
    }
});
```

注意，404 中间件用 req.accepts()方法获得当前请求预期的返回类型。现行前后端开发大量采用的是 JSON 格式交互的 API，因此只有在 req.accepts()方法明确预期是 HTML 时，才能用 res.render()方法给浏览器显示 HTML 页面。

非 HTML 请求则需要再检查是否是 JSON 请求并使用 req.json()方法返回信息，如果前两者均不支持，则认为浏览器端仅需要通用的文本类型返回。

404 页面的显示效果如图 3-37 所示。

3. 500 页面美化

500 页面也采用了开源的错误页面，如图 3-38 所示。

在 setDefaultMiddleware()方法中，同样要注意用 req.accepts()方法检查是否支持 HTML 格式或者 JSON 格式，采用对应的输出方法，代码如下：

```
//chapter03/09-error/src/default/express-server.class.ts


//500 页面
this.app.use((err, req, res, next) => {
    if (!err) {
        next();
    }
    error(err);
```

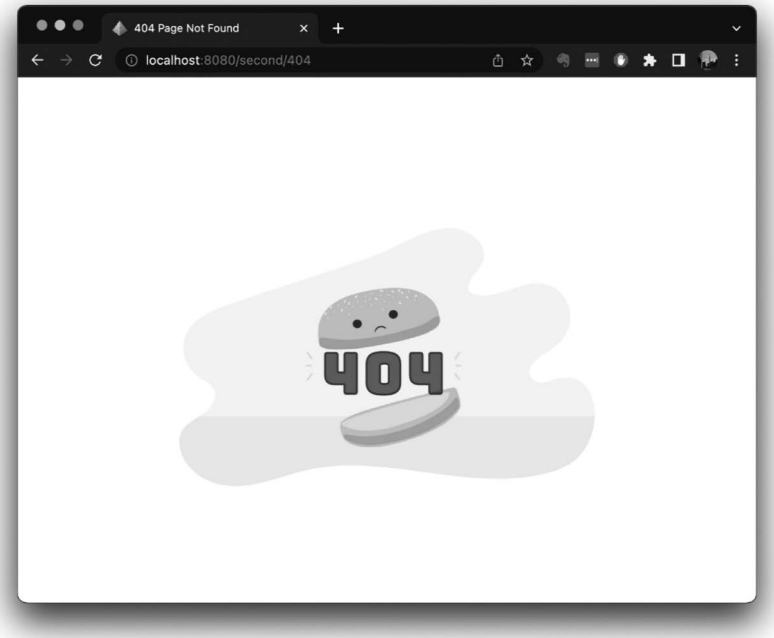

图 3-37　404 页面显示效果

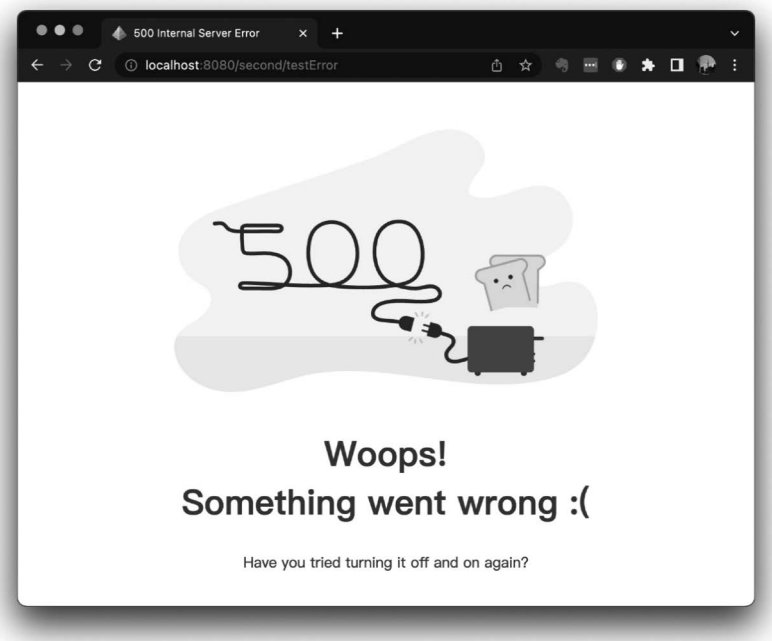

图 3-38　500 页面显示效果

```
        res.status(err.status || 500);
        if (req.accepts('html')) {
            res.render(process.cwd() + "/static/error-page/500.html");
        } else if (req.accepts('json')) {
            //当浏览器需要 JSON 格式时,返回 JSON 错误提示
            res.json({ error: 'Internal Server Error' });
        } else {
            //默认情况返回文本类型错误提示
            res.type('txt').send('Internal Server Error');
        }
    });
```

3.9.4　小结

本节讲解了框架对状态码为 404 和 500 两种服务错误的处理。找不到资源或路由配置错误的 404 错误,以及服务器端程序出错导致的 500 错误都是最常见的网络错误情况。

两类错误的捕获,比较巧妙地运用了 ExpressJS 中间件的特性,它们是特殊的中间件:

(1) 404 错误处理中间件紧跟着 setRouter()方法,当请求路径无法匹配 setRouter()里的所有页面路径时,不管是请求路径本身写错,还是页面的路由配置错误都能解释成找不到页面的错误,那么该请求将由 404 中间件进行处理。

(2) 500 错误处理中间件遵循了 ExpressJS 中间件的错误流转规则。当中间件链条任意中间件抛出异常或者给 err 参数赋值都会被视为出现服务错误。那么请求会跳过链条上后续的中间件,直接交给有 4 个参数的 500 错误中间件捕获处理。

值得注意的是,在 3.1.2 节介绍的中间件机制是名为责任链的设计模式,而当某个环节出错时,略过后续环节而直接进入错误处理程序的方式,有效地解决了链条过长出错难以捕获的问题,这是责任链模式的标准做法。