拦截器

拦截器一种编程技术，用于在数据请求过程中拦截请求或响应。前端开发同学可能对axios发送请求的拦截器更加的熟悉，而NestJs中的拦截器拦截的是Controller控制器的请求。它是一种设计模式，用于在请求和响应之间插入额外的逻辑。通过使用拦截器，你可以实现诸如日志记录、身份验证、异常处理等功能。

创建拦截器

拦截器是一个实现了 `NestInterceptor` 接口的类，并且内部重写了 `intercept()` 方法，它返回一个代表响应流的 `Obeservable` 对象。

```
import { CallHandler, ExecutionContext, Injectable, NestInterceptor } from "@ne
import { Observable } from "rxjs";

@Injectable()
export class TestInterceptor implements NestInterceptor {
  intercept(context: ExecutionContext, next: CallHandler<any>): Observable<any>
    return next.handle().pipe()
  }
}
```

`intercept()` 方法内必须调用 `next.handle()` 函数，当调用 `next.handle()` 函数时才会发出原始请求，如果你不调用这个方法，那么请求将不会被发送到服务器，也不会产生任何响应。

绑定拦截器

拦截器的绑定可以在指定模块中绑定，也可以在全局绑定。

局部绑定

局部绑定拦截器需要从 `@nestjs/common` 包中导入 `@UseInterceptors()` 装饰器。

```
import {
  Controller, UseInterceptors,
  Get
} from '@nestjs/common'
import { OrderInterceptor } from './order.interceptor'
@UseInterceptors(OrderInterceptor)
@Controller('order')
export class OrderController {
  @Get()
  findAll(){
    return ['order find all handler',12,3]
  }
}
```

全局绑定

全局绑定只需要在 `main.ts` 中使用app提供的全局方法 `useGlobalInterceptors()` 即可

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { TestInterceptor } from './common/test.interceptor'
async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.useGlobalInterceptors(new TestInterceptor())
  await app.listen(3000);
}
bootstrap();
```

更改响应格式

前端同学向服务端发送请求，服务端响应给客户端的结构都是统一的结构方便前端同学进行响应数据的处理。

```
{
    code:200,
    success:true,
    message:'ok',
    data:null
}
```

要统一返回这种格式，服务端不可能向下图一样每个控制器中都写上重复的一串代码，拦截器响应映射就是能很方便的处理统一响应格式。

```
@Controller('order')
export class OrderController {
  @Get()
  findAll(){
    return {
      code:200,
      success:true,
      message:'ok',
      data:{
        list:[]
      }
    }
  }

  @Get('info')
  findInfo(){
    return {
      code:200,
      success:true,
      message:'ok',
      data:null
    }
  }
}
```

在控制器中只需要关注data即可，对上面最简单的那段代码进行更改，这里需要注意的是如果获取控制器中返回的数据。

```
import { CallHandler, ExecutionContext, Injectable, NestInterceptor } from "@ne
import { Observable } from "rxjs";
import { tap,map } from 'rxjs/operators'

@Injectable()
export class TestInterceptor implements NestInterceptor {
  intercept(context: ExecutionContext, next: CallHandler<any>): Observable<any>
    console.log('abc', next)
    return next.handle().pipe(
      map((data)=>{
        return {
          code:200,
          data,
          msg:'success'
        }
      }),
      tap(()=>{
        console.log('aaaaaaa')
      })
    )
  }
}
```

9．守卫Guards

在 NestJS 中，Guard（守卫）是一种用于保护路由端点的机制。类似于中间件，在请求到达路由处理程序之前，Guard 可以对请求进行拦截并在特定条件下允许或拒绝请求。可以将 Guard 想象成一个安全守卫或门卫，它会检查进入系统的请求是否符合特定的要求才能通过。通常情况下，Guard 在路由处理程序之前执行，用于验证请求是否具备执行相应操作的权限或满足其他特定条件。这些条件可以是身份验证、授权、角色检查、IP 地址过滤等。

创建守卫

Guard守卫是一个实现了 `CanActivate` 接口的类，并且内部重写了 `canActivate()` 方法，返回一个布尔值。添加了守卫的路由会触发这个类，如果返回的布尔值是true则继续发起请求，如果返回的布尔值是false则拒绝请求。

```
import { Injectable, CanActivate, ExecutionContext } from "@nestjs/common";
import { Observable } from "rxjs";

@Injectable()
export class TestGuard implements CanActivate {
  canActivate(context: ExecutionContext): boolean | Promise<boolean> | Observal
    console.log('触发了守卫')
    return false
  }
}
```

绑定守卫

守卫绑定可以绑定有3种方式：

- 绑定在控制器上：只在当前控制器下的所有路由生效
- 绑定在路由方法上：绑定了该守卫的路由生效
- 绑定在全局：项目所有的路由生效

绑定在控制器和绑定在路由方法上都使用 `UseGuards`

绑定在路由方法上

```
@Controller('user')
export class UserController {
  @Get()
  @UseGuards(TestGuard)
  handleUserInfo(){
    return 'user info'
  }
}
```

绑定在控制器上

```
@Controller('user')
@UseGuards(TestGuard)
export class UserController {
  @Get()
  handleUserInfo(){
    return 'user info'
  }
}
```

全局绑定

全局绑定，调用app应用的 `useGlobalGuards()` 方法

```
async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.useGlobalGuards(new TestGuard())
  await app.listen(3000);
}
```

10. 装饰器

在 NestJS 中，装饰器是一种元编程技术，用于修改类、方法、属性或参数的行为。装饰器通常以 `@` 符号开始，并可以接受一些参数。NestJS 使用装饰器来定义和组织应用程序的架构，使得代码更加简洁且易于维护。

内置装饰器

NestJS内部提供了装饰器，有类装饰器、方法装饰器、属性装饰器

类装饰器:

- `@Controller()`：用于定义一个控制器类。
- `@Middleware()`：用于定义中间件类。
- `@Module()`：用于定义模块类。
- `@Injectable()`：用于定义服务类（也称为提供者）。
- `@Inject()`：用于注入依赖到构造函数中。

方法装饰器:

- `@Get()`, `@Post()`, `@Put()`, `@Delete()` 等: 用于定义 HTTP 请求处理程序。
- `@UseGuards()`：用于应用守卫。
- `@UseInterceptors()`：用于应用拦截器。
- `@UsePipes()`：用于应用管道。
- `@UseFilters()`：用于应用异常过滤器。

属性装饰器:

- `@Body()`
- `@Param()`
- `@Query()`

- `@Req()`
- `@Res()`
- `@Headders()`

自定义装饰器

如果你的参数需要一些额外的处理是NestJS内置装饰器无法完成的功能，比如对参数解密，你可以创建自定义装饰器，使用 `createParamDecorator()` 方法创建参数装饰器。

```
import { createParamDecorator, ExecutionContext } from "@nestjs/common";

export const DecryptParameter = createParamDecorator(
  (data:unknown,context:ExecutionContext)=>{

    return 'abc'
  }
)
```

在Controller中使用， 输出的就是自定义装饰其中返回的值

```
@Get()
handleUserInfo(@DecryptParameter('age') param){
    console.log(param,888)
    return 'user info'
}
```

设置元数据

最常见的设置元数据的场景，是做角色权限时。需要告诉NestJS哪些路由接口只能由指定的角色才能访问，指定角色就可以自定义一个设置元数据的装饰器。

设置元数据使用 `SetMetadata()` 方法，它是一个装饰器工厂函数，用于设置元数据。元数据是在运行时与类、方法或参数相关联的额外信息。

使用 `SetMetadata()` 可以在类或方法级别设置元数据，然后通过反射 API 在运行时访问这些元数据。这对于实现一些高级功能非常有用，比如权限控制、路由映射、请求处理等。

如下：

```
import { SetMetadata } from '@nestjs/common';

export const Custom = (value: string) => SetMetadata('custom', value);
```

在controller中这么使用

```
@Get()
@Custom('this is custom')
@UseGuards(TestGuard)
handleUserInfo(){
    return 'user info'
}
```

在守卫中就是用 `Reflector` 获取设置的 `custom` 数据

```
const value = this.reflector.get('custom',context.getHandler())
console.log(value)
```

Reflector简介

`Reflector` 是 NestJS 提供的一个服务，用于在运行时反射类、方法或参数上的元数据。它提供了几个方法，如 `get()`、 `getAll()` 和 `has()`，可以用来读取和检查元数据。

JWT身份验证

接口权限

NestJS中控制接口访问权限通常是通过Guards守卫来实现，最基本的访问权限就是身份校验，访问需要权限的接口需要带一个标识身份的字符串。通常来说一个项目中大部分的接口都是需要访问权限的，只有少数的接口，比如登录接口不需要。

因为大多数接口需要身份权限，少数接口不需要。那么就可以用一个守卫默认所有的接口都需要校验身份，给不需要校验身份的少数接口添加指定的元数据，在守卫中对这些带有指定元数据的接口放行即可。

案例演示：

准备两个接口路由： `/user/login` 和 `/user/info`

`user.congroller.ts` 代码

```
import { Controller, Post, Body, Get } from "@nestjs/common";

import { UserService } from "./user.service";

@Controller('user')
export class UserController {
  constructor(
    private userService:UserService
  ){}

  @Post('login')
  handleLogin(@Body() loginDto){
    const { username } = loginDto
    return this.userService.findOne(username)
  }

  @Get('info')
  hanleUserinfo(){
    return 'userinfo'
  }
}
```

`user.service.ts` 代码

```
import { Injectable } from "@nestjs/common";

@Injectable()
export class UserService {
  private readonly users = [
    { username:'张三', userid:1, password:'abc12345'},
    { username:'李四', userid:2, password:'abc12345'}
  ]

  async findOne(username){
    return this.users.find(user => user.username = username)
  }
}
```

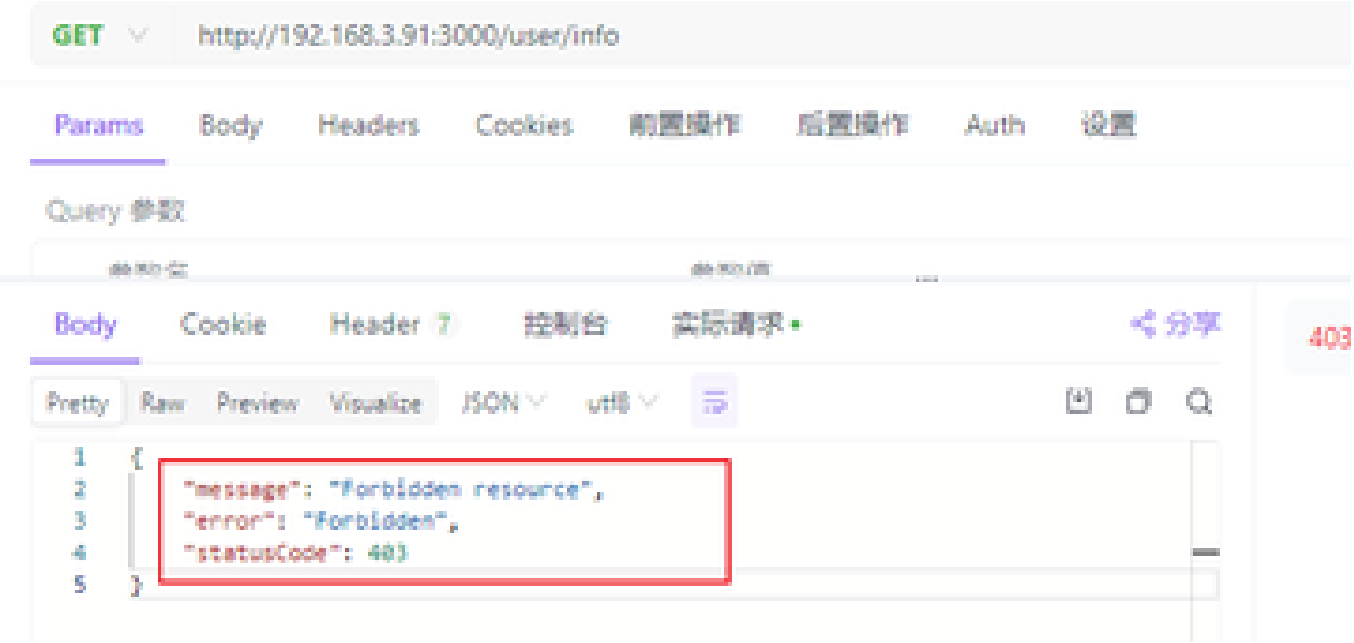
此时访问两个接口路由都可以成功响应，默认这两个接口路由都不可通过，**创建守卫 `auth.guard.ts`**

```
import { Injectable, CanActivate, ExecutionContext } from "@nestjs/common";
import { Observable } from "rxjs";

@Injectable()
  canActivate(context: ExecutionContext): boolean | Promise<boolean> | Observal

    return false
  }
}
```

全局绑定守卫，**来到** `app.module.ts` **文件**，用注册 `Providers` 的方式注册绑定全局守卫

```
import { Module } from '@nestjs/common';
import { APP_GUARD } from '@nestjs/core';
import { UserModule } from './user/user.module';
import { AuthGuard } from './guards/auth.guard';
@Module({
  imports: [
    UserModule
  ],
  providers:[
    { provide:APP_GUARD, useClass:AuthGuard }
  ]
})
export class AppModule {}
```

那么此时访问两个接口路由就会被拒绝访问



但是希望 `/user/login` 接口可以通过请求，使用 `SetMetadata()` 给 `/user/login` 接口路由添加元数据，这里做成一个 `Public` 自定义装饰器。

创建 `public.decorator.ts` 文件并填充代码

```
import { SetMetadata } from "@nestjs/common";

export const IS_PUBLICK_KEY = 'isPublic'
export const Public = ()=> SetMetadata(IS_PUBLICK_KEY,true)
```

哪个接口路由需要开放，就把这个装饰器给哪个用

```
@Post('login')
@Public()
handleLogin(@Body() loginDto){
    const { username } = loginDto
    return this.userService.findOne(username)
}
```

同时还要**更改auth守卫的代码**，添加上条件语句

```
import { Injectable, CanActivate, ExecutionContext } from "@nestjs/common";
import { Reflector } from "@nestjs/core";
import { Observable } from "rxjs";
import { IS_PUBLICK_KEY } from "src/decorators/public.decorator";

@Injectable()
export class AuthGuard implements CanActivate {
  constructor(
    private reflector:Reflector
  ){}

  canActivate(context: ExecutionContext): boolean | Promise<boolean> | Observal
    const isPublic = this.reflector.get(IS_PUBLICK_KEY,context.getHandler())

    if(isPublic){
      return true
    }

    return false
  }
}
```

那么现在可以访问 `/user/login` 而不可以访问 `/user/info`

JWT的使用

我希望 `/user/info` 接口路由也可以在通过某种规则校验之后也能正常发起请求，通常会是在login接口中生成一个token返回，前端再调其它接口时把这个token携带上给到服务端，服务端解析校验，解析校验成功则正常请求。

需要安装一个库 `jsonwebtoken`

JSON Web Token（JWT）是一种开放标准，用于安全地在各方之间传输信息作为 JSON 对象。通常用于用户认证和授权。

安装

```
yarn add jsonwebtoken
```

生成token

JWT提供了一个 `sign()` 方法基于指定的数据，指定的secretKey生成token

```
const {sign} = require('jsonwebtoken')

async findOne(username){
    console.log(sign({username},'qingniu'))
    return this.users.find(user => user.username = username)
}
```

验证token

JWT提供了一个 `verify()` 方法验证生成的token，从token中可以解析出源数据

```
const payload = verify(token,'qingniu')
```

过期时间

如果不手动添加过期时间，那么sign生成的token是没有过期时间的。

- 在payload数据中添加 `exp` 属性指定过期时间，单位是秒

```
const payload = {
    username:'张三',
    exp:60 * 5        // 5分钟后过期
}
const token = sign(payload,'secretKey')
```

- 在sign的第三个参数中配置过期时间，设置 `expiresIn` 属性的值

```
const payload = {
  username: '张三',
};
const token = jwt.sign(payload, 'secretKey', { expiresIn: '1h' });
```

过期时间接收的值

`expiresIn` 选项可以接受一个数字或字符串作为值。这个值表示token的有效期长度：

- 如果你提供一个数字，它会被解释为以秒为单位的时间长度。
- 如果你提供一个字符串，你需要确保在字符串中包含时间单位（如"days"、"hours"等），否则默认使用毫秒作为单位。

加上身份验证

在登录接口生成token返回，再在访问 `/user/info` 接口时token塞入请求头

更改 `user.service.ts` 代码

```
async findOne(username){
    const user = this.users.find(user => user.username = username)
    if(user){
      const token = sign({username},'qingniu')
      return token
    }
}
```

更改 `auth.guard.ts` 代码

```
import { Injectable, CanActivate, ExecutionContext, UnauthorizedException } fro
import { verify } from "jsonwebtoken";
...

@Injectable()
export class AuthGuard implements CanActivate {
  ...

  canActivate(context: ExecutionContext): boolean | Promise<boolean> | Observat
    ...
    const request = context.switchToHttp().getRequest()
    const token = this.extractTokenFromHeader(request)
    if(!token){
      throw new UnauthorizedException()
    }
    try {
      const payload = verify(token,'qingniu')
      request['user'] = payload
    } catch (error) {
      throw new UnauthorizedException()
    }
    return true
  }

  private extractTokenFromHeader(request){
    const [type, token] = request.headers.authorization?.split(' ') ?? [];
    return type === 'Bearer' ? token : undefined;
  }
}
```

验证token成功后会把信息添加到 `request` 对象上去，那在访问 `/user/info` 接口时可以获取到

```
@Get('info')
hanleUserinfo(@Req() req){
    console.log(req.user,555555)
    return 'userinfo'
}
```

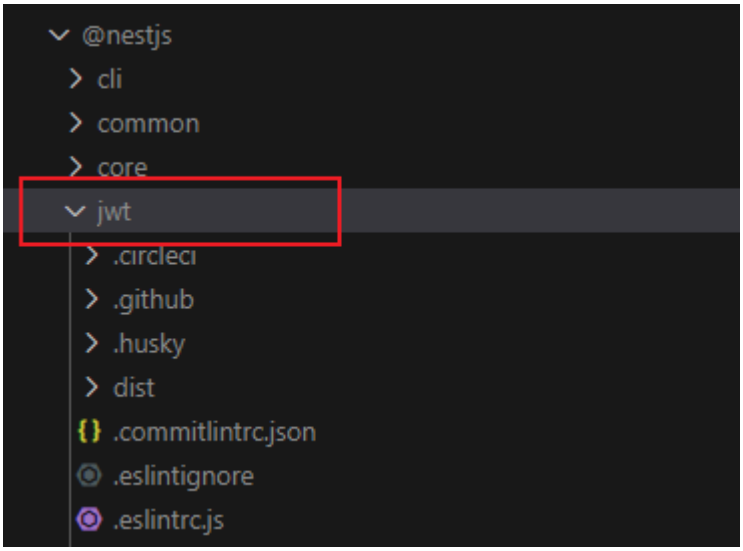
@nestjs/jwt

`@nestjs/jwt` 是NestJS框架中的一个模块，提供了与JSON Web Token（JWT）相关的工具函数和策略类。这个模块可以让你在NestJS项目中轻松地生成和解析JWT。

运行安装命令

```
yarn add @nestjs/jwt
```

安装完成后你可以在 `node_module` 中的 `@nestjs` 文件目录中找到它



`@nestjs/jwt` 的主要功能包括：

- **JWT服务**：提供了一个封装了jsonwebtoken库的JWT服务，用于生成和解析JWT。
- **JwtModule**：是一个可以全局配置JWT选项的模块，例如设置密钥、过期时间等。
- **JwtStrategy**：是一个实现了Passport.js策略接口的类，用于从请求中提取并验证JWT。

改用 `@nestjs/jwt` 创建生成token

在前面我们是直接使用 `jsonwebtoken` 创建生成的token，现在我们用 `JwtModule` 模块来全局配置JWT选项，调用 `JwtService` 的方法生成token。

使用JwtModule必须在要在使用的那个模块中导入注册，来到 `user.module.ts` 文件

```
import { Module } from "@nestjs/common";
import { JwtModule } from "@nestjs/jwt";
import { UserController } from "./user.controller";
import { UserService } from "./user.service";

@Module({
  imports:[
    JwtModule.register({
      global:true,
      secret:'QING NIU QIANDUAN',
      signOptions:{
        expiresIn:60
      }
    })
  ],
  controllers:[UserController],
  providers:[UserService],
  exports:[UserService]
})
export class UserModule {}
```

只需调用 `JwtModule` 的 `register()` 方法注册并全局配置jwt，然后就可以把 `JwtService` 服务注入到 `user.service.ts` 中使用 `signAsync()` 方法

```
import { Injectable } from "@nestjs/common";
import { JwtService } from "@nestjs/jwt";
@Injectable()
export class UserService {
  ...
  constructor(
    private jwtService:JwtService
  ){}

  async findOne(username){
    const user = this.users.find(user => user.username = username)
    if(user){
      const token = this.jwtService.signAsync(user)
      return token
    }
  }
}
```

在守卫中也改成调用 `JwtService` 的 `verifyAsync()` 方法

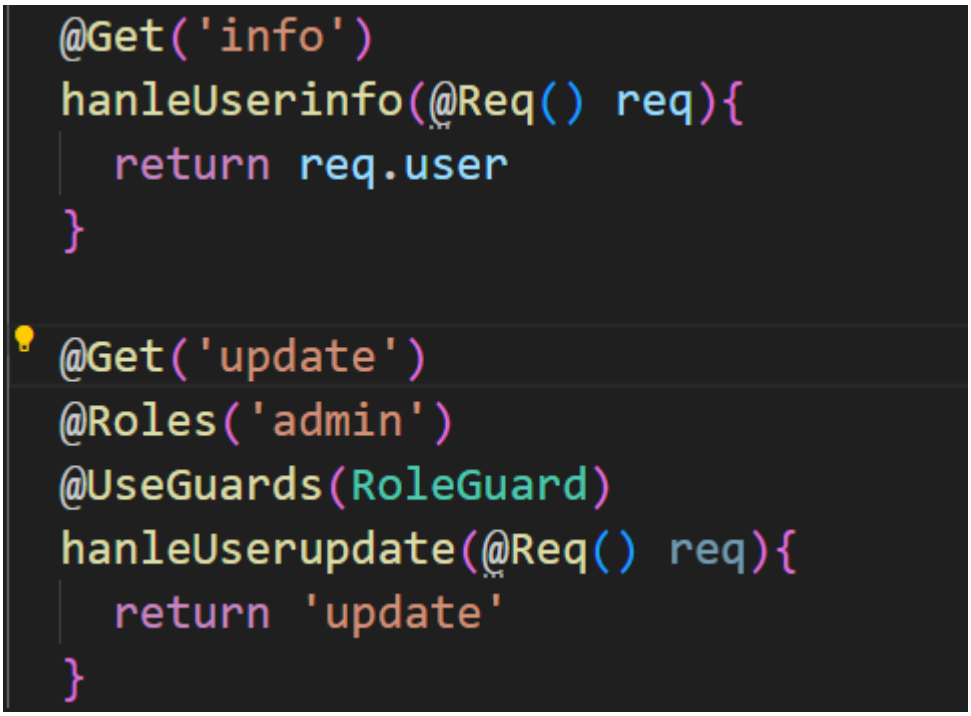
```
import { Injectable, CanActivate, ExecutionContext, UnauthorizedException } fro
import { JwtService } from "@nestjs/jwt";
..

@Injectable()
export class AuthGuard implements CanActivate {
  constructor(
    private reflector:Reflector,
    private jwtService:JwtService
  ){}

  canActivate(context: ExecutionContext): boolean | Promise<boolean> | Observab
    ...
    try {
      const payload = this.jwtService.verifyAsync(token,{
        secret:'QING NIU QIANDUAN'
      })
      request['user'] = payload
    } catch (error) {
      throw new UnauthorizedException()
    }
    return true
  }
  ...
}
```

根据角色限制

有时候在后台管理系统项目中，需要根据登录账号的角色来对访问的接口路由进行授权。比如只有超级管理员账号才能够请求密码修改、添加管理员账号等接口，那就需要做角色权限控制。

给指定的路由添加元数据，告诉NestJS这个路由需要什么角色才能访问，然后再添加角色守卫。

比如有两个接口，一个接口只需要身份认证，也就是只要有可用的token就可以了；另一个接口需要限定admin账号才能访问。



```
@Get('info')
hanleUserinfo(@Req() req){
  return req.user
}

@Get('update')
@Roles('admin')
@UseGuards(RoleGuard)
hanleUserupdate(@Req() req){
  return 'update'
}
```

如上给 `update` 接口使用装饰器添加角色限制，还要使用角色守卫进行拦截

创建一个 `role.decorator.ts` 自定义装饰器

```
import { SetMetadata } from "@nestjs/common";

export const ROLE_KEY = 'role'
export const Roles = (value)=> SetMetadata(ROLE_KEY,value)
```

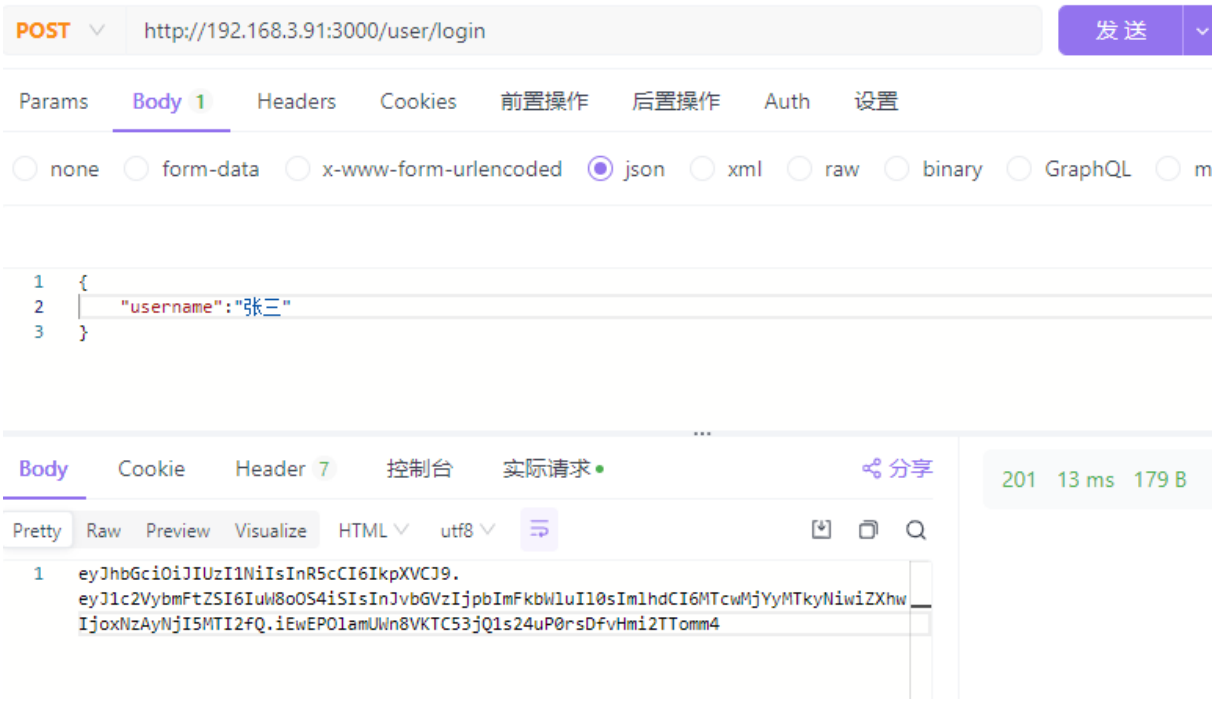
再创建一个 `role.guard.ts` 做角色守卫

```
import { Injectable, CanActivate, ExecutionContext, UnauthorizedException } fro
import { Reflector } from "@nestjs/core";
import { Observable } from "rxjs";
import { ROLE_KEY } from "src/decorators/role.decorator";

@Injectable()
export class RoleGuard implements CanActivate{
  constructor(
    private reflector:Reflector
  ){}
  canActivate(context: ExecutionContext): boolean | Promise<boolean> | Observat
    const role = this.reflector.get(ROLE_KEY,context.getHandler())

    if(!role) return true

    const request = context.switchToHttp().getRequest()
    const user = request.user

    return user&&user.roles.some(r=>r===role)
  }
}
```

在生成token时，需要把账号的 role也作为信息之一

```
async findOne(username){
    const user = this.users.find(user => user.username === username)
    if(user){
      const payload = {
        username:user.username,
        roles:user.role.split(',')
      }
      const token = this.jwtService.sign(payload)
      return token
    }
}
```

请求接口测试，先请求 `/user/login` 接口生成token



然后再把得到的token放到请求头中，继续请求 `/user/update`