

Module模块

概念介绍

模块是具有 `@Module()` 装饰器的类，该装饰器提供了元数据。每个Nest应用至少有一个模块 `app.module.ts`，即根模块。模块的作用就是用来组织应用程序结构，Controller控制器、Service提供者都需要在模块中注册。

`@Module()` 装饰器接收一个描述模块属性的对象：`metadata`，包含的属性只能是如下图中的四个：`imports`、`controllers`、`providers`、`exports`

```
export interface ModuleMetadata {  
  /**  
   * Optional list of imported modules that export the providers which are  
   * required in this module.  
   */  
  imports?: Array<Type<any> | DynamicModule | Promise<DynamicModule> | ForwardReference>;  
  /**  
   * Optional list of controllers defined in this module which have to be  
   * instantiated.  
   */  
  controllers?: Type<any>[];  
  /**  
   * Optional list of providers that will be instantiated by the Nest injector  
   * and that may be shared at least across this module.  
   */  
  providers?: Provider[];  
  /**  
   * Optional list of the subset of providers that are provided by this module  
   * and should be available in other modules which import this module.  
   */  
  exports?: Array<DynamicModule | Promise<DynamicModule> | string | symbol | Provider | ForwardReference | Abstract<any> | Function>;  
}
```

如果你的Nest应用比较简单，可以将所有不同功能的Controller控制器、Service提供者都在这个根模块中注册。

```
import { Module } from '@nestjs/common';  
import { UserController } from './user/user.controller'  
import { GoodsController } from './goods/goods.controller'  
import { OrderController } from './order/order.controller'  
import { UserService } from './user/user.service'  
import { GoodsService } from './goods/goods.service'  
import { OrderService } from './order/order.service'  
@Module({  
  imports: [],  
  controllers: [UserController, GoodsController, OrderController],  
  providers: [UserService, GoodsService, OrderService],  
})  
export class AppModule {}
```

功能模块

如上在应用程序比较简单时可以将所有的Controller、Service都在跟模块中注册。但是如果你开发的应用程序是大型复杂项目，可以考虑将同属同一功能的Controller、Service在一个module下注册。

如上，有三个功能模块User、Goods、Order，用命令行 `nest g mo name` 生成对应的模块文件：

- user.module.ts
- goods.module.ts
- order.module.ts

以User模块为例：

```
import { Module } from '@nestjs/common';
import { UserController } from './user.controller';
import { UserService } from './user.service';
@Module({
  controllers: [UserController],
  providers: [UserService]
})
export class UserModule {}
```

在根模块文件，导入它们，注册到imports中

```
import { Module } from '@nestjs/common';
import { UserModule } from './user/user.module';
import { OrderModule } from './order/order.module';
import { GoodsModule } from './goods/goods.module';
@Module({
  imports: [UserModule, OrderModule, GoodsModule],
  controllers: [],
  providers: [],
})
export class AppModule {}
```

模块共享

模块与模块之间可以轻松的共享同一个提供者的实例，如果你想在模块A中使用模块B的提供者实例，那么你必须要把B的提供者实例放到 `exports` 数组中，同时你要在模块A中导入模块B。

例如，想把Level功能模块的Service暴露出来，那么 `level.module.ts` 代码如下：

```
import { Module } from '@nestjs/common';
import { LevelService } from '../level.service';
import { LevelController } from '../level.controller';
@Module({
  controllers: [LevelController],
  providers: [LevelService],
  exports:[LevelService]
})
export class LevelModule {}
```

Level功能的Service暴露出来之后，要在User功能模块中使用它，更改 `user.module.ts` 代码如下：

```
import { Module } from '@nestjs/common';
import { UserService } from '../user.service';
import { UserController } from '../user.controller';
import { LevelModule } from '../../level/level.module'
@Module({
  imports:[LevelModule],
  controllers: [UserController],
  providers: [UserService]
})
export class UserModule {}
```

然后在控制器 `user.controller.ts` 中使用

```
import { Controller, Get } from '@nestjs/common';
import { UserService } from '../user.service';
import { LevelService } from '../../level/level.service'
@Controller('user')
export class UserController {
  constructor(
    private readonly userService: UserService,
    private readonly levelService: LevelService
  ) {}
  @Get()
  findAll() {
    return this.levelService.findAll();
  }
}
```

全局模块

如果一个功能模块A的提供者实例需要在很多个不同的模块中使用，用上面共享模块的方式需要在各个不同的模块中重复导入模块A的步骤。为了方便使用，可以将模块A设置为全局模块。

装饰器 `@Global()` 可以使模块成为全局模块，只需要在根模块注册一次即可在所有需要使用到他的地方直接使用,如下代码将Level模块变成全局模块

```
import { Module, Global } from '@nestjs/common';
import { LevelService } from './level.service';
import { LevelController } from './level.controller';
@Global()
@Module({
  controllers: [LevelController],
  providers: [LevelService],
  exports:[LevelService]
})
export class LevelModule {}
```

5. 中间件的使用

中间件是处于接收到请求到执行路由处理程序之间的函数，中间件函数可以访问请求和响应对象，因此在中间件函数中可以处理请求对象，获取到请求头中的token进行校验等逻辑。



中间件就是一个函数，在这个函数中可以执行以下任务：

- 执行任何代码
- 对请求和响应对象进行更改
- 结束请求
- 调用对栈中的下一个中间件函数
- 如果当前的中间件函数没有结束请求-响应周期，它必须调用 `next()` 将控制传递给下一个中间件函数。否则，请求将被挂起。

实现中间件类

中间件就是一个具有装饰器 `@Injectable()` 的类，这个类需要实现 `NestMiddleware` 接口。

```
import {
  Injectable, NestMiddleware
} from '@nestjs/common'
import { Request, Response, NextFunction } from 'express'
@Injectable()
export class LoggerMiddleware implements NestMiddleware{
  use(req:Request, res:Response, next:NextFunction){
    console.log(req)
    next()
  }
}
```

使用中间件

中间件不能像Controller、Providers那样在 `@Module()` 装饰器中列出，使用这个中间件的模块需要实现 `NestModule` 接口，代码如下：

```
import { Module, NestModule, MiddlewareConsumer } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';

import { LoggerMiddleware } from './common/logger.middleware';

@Module({
  imports: [],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule implements NestModule{
  configure(consumer: MiddlewareConsumer){
    consumer.apply(LoggerMiddleware).forRoutes(AppController)
  }
}
```

路由匹配

forRoutes()用于匹配指定哪些控制器、哪些路由可以使用这个中间件。

- 单个字符串：指定这一个路由可以使用这个中间件
- 多个字符串：指定多个路由可以使用这个中间件

- 单个控制器：指定这一个控制器下所有的路由都可以使用这个中间件
- 多个控制器：每个控制器以逗号分隔，指定的控制器下所有的路由都可使用
- RouteInfo对象：包含path、method的对象

指定 `/user/info` 路由可以使用中间件

```
export class AppModule implements NestModule{
  configure(consumer: MiddlewareConsumer){
    consumer.apply(LoggerMiddleware).forRoutes('user/info')
  }
}
```

指定 `/user/info` 路由和 `/user/abc` 路由可以使用中间件

```
export class AppModule implements NestModule{
  configure(consumer: MiddlewareConsumer){
    consumer.apply(LoggerMiddleware).forRoutes('user/info','user/abc')
  }
}
```

所有以 `/user` 开头的路由都可以使用中间件

```
export class AppModule implements NestModule{
  configure(consumer: MiddlewareConsumer){
    consumer.apply(LoggerMiddleware).forRoutes('user')
  }
}
```

只有 `AppController` 下的所有路由才可以使用中间件

```
export class AppModule implements NestModule{
  configure(consumer: MiddlewareConsumer){
    consumer.apply(LoggerMiddleware).forRoutes(AppController)
  }
}
```

只有 `AppController` 和 `ProductController` 下的所有路由才可以使用中间件

```
export class AppModule implements NestModule{
  configure(consumer: MiddlewareConsumer){
    consumer.apply(LoggerMiddleware).forRoutes(AppController,ProdutController)
  }
}
```

只有以 `user` 开头且是GET请求的路由才可以使用中间件

```
export class AppModule implements NestModule{
  configure(consumer: MiddlewareConsumer){
    consumer.apply(LoggerMiddleware).forRoutes(
      { path:'user', method:RequestMethod.GET }
    )
  }
}
```

排除路由

可以使用 `exclude()` 排除某些路由，指定哪些路由不经过这个中间件

- 单个字符串：指定的路由不使用该中间件
- 多个字符串：指定多个路由不使用该中间件
- RouteInfo对象：包含path、method的对象

用法与 `forRoutes()` 类似，只是不能使用控制器，其它参数接收一致。功能与 `forRoutes()` 相反

函数式中间件

除了上面那种类中间件的方式来书写中间件，还可以使用更简便的函数式的方式，以上面的为例，就在要使用中间件的module中定义一个中间件函数。

```
import { Module, Global, NestModule, MiddlewareConsumer } from '@nestjs/common'

const logger = (req, res, next) => {
  console.log(req)
  next()
}

@Module()
export class LevelModule implements NestModule{
  configure(consumer: MiddlewareConsumer){
    consumer.apply(logger).forRoutes()
  }
}
```

6. 异常过滤器

Nest 带有一个内置的 **异常层**，它负责处理应用中所有未处理的异常。当你的应用代码未处理异常时，该层会捕获该异常，然后自动发送适当的用户友好响应。

抛出异常

NestJs中提供了一个内置的 `HttpException` 类，这个类是异常基础类，从 `@nestjs/common` 包中导入，在发生某些错误时发送标准HTTP响应对象。

使用 `throw` 关键字抛出异常

```
throw new HttpException('forbidden', HttpStatus.BAD_REQUEST)
```

响应的是标准的错误信息

```
{
  "statusCode": 400,
  "message": "forbidden"
}
```

自定义异常响应

通常不希望响应的异常错误是默认的格式内容，NestJS允许开发者自定义异常响应的格式及内容。

使用 `@UseFilters()` 装饰器，在控制器或特定路由中应用异常过滤器。异常过滤器可以捕获并处理在控制器方法执行过程中抛出的异常。

自定义一个异常过滤器类。这个类需要实现 `ExceptionHandler` 接口，并重写其 `catch()` 方法。当使用 `@UseFilters()` 装饰器给控制器添加上过滤器，抛出异常时就会触发这个过滤器。

```
// 自定义过滤器类
import { ExceptionFilter, Catch } from "@nestjs/common";

@Catch()
export class HttpExceptionFilter implements ExceptionFilter {
  catch(){
    console.log('catch')
  }
}

import {
  Controller,
  Get, Query,
  HttpException,
  HttpStatus,
  UseFilters
} from '@nestjs/common';
import { HttpExceptionFilter } from './common/http-exception.filter';
@Controller('user')
@UseFilters(HttpExceptionFilter)
export class AppController {
  @Get('info')
  handleInfo(@Query() query){
    throw new HttpException('forbidden', HttpStatus.BAD_REQUEST)
    return query
  }
}
```

访问 `user/info` 抛出异常，会触发过滤器 在终端会输出字符串 `catch`。

重写catch方法

异常过滤器类需要实现 `ExceptionHandler` 类，并且需要重写 `catch()` 方法，这个方法接收两个参数：

- **exception**：表示当前正在处理的异常对象。



当控制器或中间件中的代码抛出一个异常时，NestJS 会捕获这个异常，并将它传递给相应的异常过滤器链。每个过滤器都有机会处理这个异常，直到找到一个能够成功处理它的过滤器为止。

- **host**: 是一个 `ArgumentsHost` 类, 表示调用者上下文。这个类封装了请求的详细信息, 包括 HTTP 请求对象、HTTP 响应对象和 Node.js 的原始请求和响应对象。



`ArgumentsHost` 提供了一些方法来切换到不同的上下文类型, 并从中提取相应的请求和响应对象。这些方法包括:

- `switchToHttp()`: 返回一个包含 Node.js HTTP 请求和响应对象的 `HttpContext` 对象。
- `switchToRpc()`: 返回一个包含 RPC 请求和响应对象的 `RpcContext` 对象。
- `switchToWs()`: 返回一个包含 WebSocket 请求和响应对象的 `WsContext` 对象。

通常切换到 `HttpContext` 对象获取 `HttpRequest` 对象, 切换到请求对象后可以调用如下方法

- `getRequest()`
- `getResponse()`
- `getNext()`

```
import { ExceptionFilter, Catch, ArgumentsHost, HttpException } from '@nestjs/core';
import { Request, Response } from 'express';
```

```
@Catch(HttpException)
export class HttpExceptionFilter implements ExceptionFilter {
  catch(exception: HttpException, host: ArgumentsHost) {
    const ctx = host.switchToHttp();
    const response = ctx.getResponse<Response>();
    const request = ctx.getRequest<Request>();
    const status = exception.getStatus();

    response
      .status(status)
      .json({
        statusCode: status,
        timestamp: new Date().toISOString(),
        path: request.url,
      });
  }
}
```

全局绑定过滤器

在 `main.ts` 中使用 `useGlobalFilters()` 方法

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';

import { HttpExceptionFilter } from './common/http-exception.filter';

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

7. Pipe管道

NestJs中管道是用来对Controller控制器的路由处理程序进行处理，Nest在调用路由处理程序之前插入一个管道，这个管道就会拦截方法的调用参数进行转换或者验证处理，然后用转换或验证好的参数调用原方法。

因此NestJs有两种管道类型：

- 转换：管道将输入数据转换为所需的数据输出
- 验证：对输入数据进行验证，验证成功继续传递，验证失败则抛出异常

参数转换

如果要对接收到的参数进行转换，NestJS提供了一些内置的转换Pipe管道。

比如接收到的参数是字符串类型的，要转成数值类型可以使用 `ParseIntPipe`。

请求地址： `/user/info?age=12`

```
@Get('info')
handleInfo(@Query('age', ParseIntPipe) age){
  console.log(typeof age, age)
  return age
}
```

比如接收到的参数转换成布尔类型，可以使用 `ParseBoolPipe`，但是只对将布尔字符串转换。即参数值只能是字符串"false"或者"true"，数值0或者空字符串不能转换，会抛出错误

```
@Post('mPost')
handlePost(@Body('show',ParseBoolPipe) show){
  console.log(typeof show,show)
  return show
}
```

比如接收到的参数以逗号(,)分隔，要转成数组可以使用 `ParseArrayPipe`。使用 `ParseArrayPipe` 必须要安装两个包：`class-validator` 和 `class-transformer`

```
@Post('mPost')
handlePost(@Body('labels',ParseArrayPipe) labels){
  console.log(typeof labels,labels)
  return labels
}
```

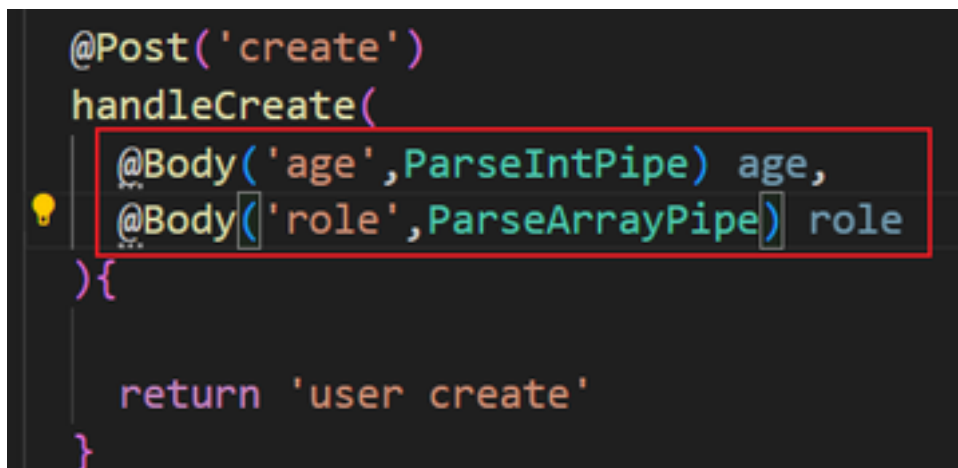
除了以上3个，还有其它6个开箱即用的内置管道：

- ValidationPipe
- ParseFloatPipe
- ParseUUIDPipe
- ParseEnumPipe
- DefaultValuePipe
- ParseFilePipe

自定义管道

如上例子中，要转换请求体中的某个字段属性的值，需要请求体中单独取出那个字段对其进行转换后单独使用。

如果请求体中有多个字段属性，那需要把请求体拆成多个转换，如下图：



```
@Post('create')
handleCreate(
  @Body('age',ParseIntPipe) age,
  @Body('role',ParseArrayPipe) role
){
  return 'user create'
}
```

这不是一种好的方式，NestJS支持自定义管道。自定义管道是一个实现了 `PipeTransform` 并且重写了内部的 `transform` 的类。

```
import { PipeTransform, Injectable, ArgumentMetadata } from "@nestjs/common";
@Injectable()
export class UserbodyPipe implements PipeTransform{
  transform(value: any, metadata: ArgumentMetadata) {
    const { role, age } = value
    value.role = role.split(',')
    value.age = parseInt(age)
    return value
  }
}
```

在Controller中用 `@UsePipes()` 装饰器使用这个自定义管道，如下body内的role和age字段属性就转换完成了

```
@Post('create')
@UsePipes(UserbodyPipe)
handleCreate(@Body() body){
  console.log(body)
  return 'user create'
}
```

`UserbodyPipe` 类内部重写了 `transform()` 方法，该方法接收两个参数：

- value: 要处理的原始值，此处代表的是请求体参数body
- metadata: 这是一个包含元数据的对象，它提供了关于正在被处理的值的一些上下文信息。这个对象有两个属性：
 - type: 这是一个字符串，表示了正在被处理的值的位置。它可以是以下几种之一：
 - 'body': 表示值来自于请求体。
 - 'query': 表示值来自于查询参数。
 - 'param': 表示值来自于路径参数。
 - 'custom': 表示值来自于自定义管道。
 - metatype: 是一个构造函数，表示了正在被处理的值的预期类型。

参数校验

参数校验可以使用 `ValidationPipe` 管道，需要配合 `class-validator` 库一起使用。

`class-validator` 是一个用于 TypeScript 和 JavaScript 的装饰器模式的验证库。它提供了一系列装饰器，可以用来指定类属性或方法参数的验证规则。

同时还需要创建一个DTO类，用DTO类来接收请求体重的数据，并且让 `ValidationPipe` 自动进行验证。

创建一个DTO类 内部用 `class-validator` 中的装饰器给字段参数指定校验格式。

```
import { IsString, IsNumber } from "class-validator";

export class CreateDto {
  @IsString()
  name:string;

  @IsNumber()
  age:number;
}
```

在Controller中使用

使用装饰器 `@UsePipes()` 局部验证

```
@UsePipes(ValidationPipe)
handleCreate(@Body() body:CreateDto){
  console.log(body)
  return 'user create'
}
```

也可以在装饰器 `@Body()` 中

```
@Post('create')
handleCreate(@Body(new ValidationPipe()) body:CreateDto){
  console.log(body)
  return 'user create'
}
```

全局启用验证管道

你需要对所有的参数进行校验，可以将 `ValidationPipe` 全局启用。在Module中添加提供者 Provider

```
import { APP_PIPE } from '@nestjs/core';  
@Module({  
  imports: [],  
  controllers: [AppController],  
  providers: [  
    {  
      provide: APP_PIPE,  
      useClass: ValidationPipe  
    },  
    AppService  
  ],  
})
```

- provide: 代表要提供的服务是 APP_PIPE
- useClass: 掉膘的是服务实际使用的类的类型