NestJs基础

1．NestJs简介

NestJS 是一个用于构建高效、可扩展的 Node.js 服务器端应用程序的开发框架。它使用现代 JavaScript，完全支持 TypeScript，并结合了多种编程范式，包括面向对象编程 (OOP)、函数式编程 (FP) 和函数式反应式编程 (FRP) 的元素。

- 支持TypeScript：NestJs默认使用Typescript，提供了静态类型检查和更好的代码组织结构
- 模块化：NestJs使用基于装饰器的依赖注入系统，允许开发者轻松的组织和管理代码模块
- 面向切面编程：提供了一种优雅的方式来处理常见的编程任务，如日志记录、事务管理和缓存
- 支持中间件：NestJS 可以无缝集成 Express 或 Fastify 等流行的 Node.js 框架
- 微服务架构： 支持构建微服务架构的应用程序，使得大型项目能够分解为独立的服务，每个服务都可以单独部署和扩展

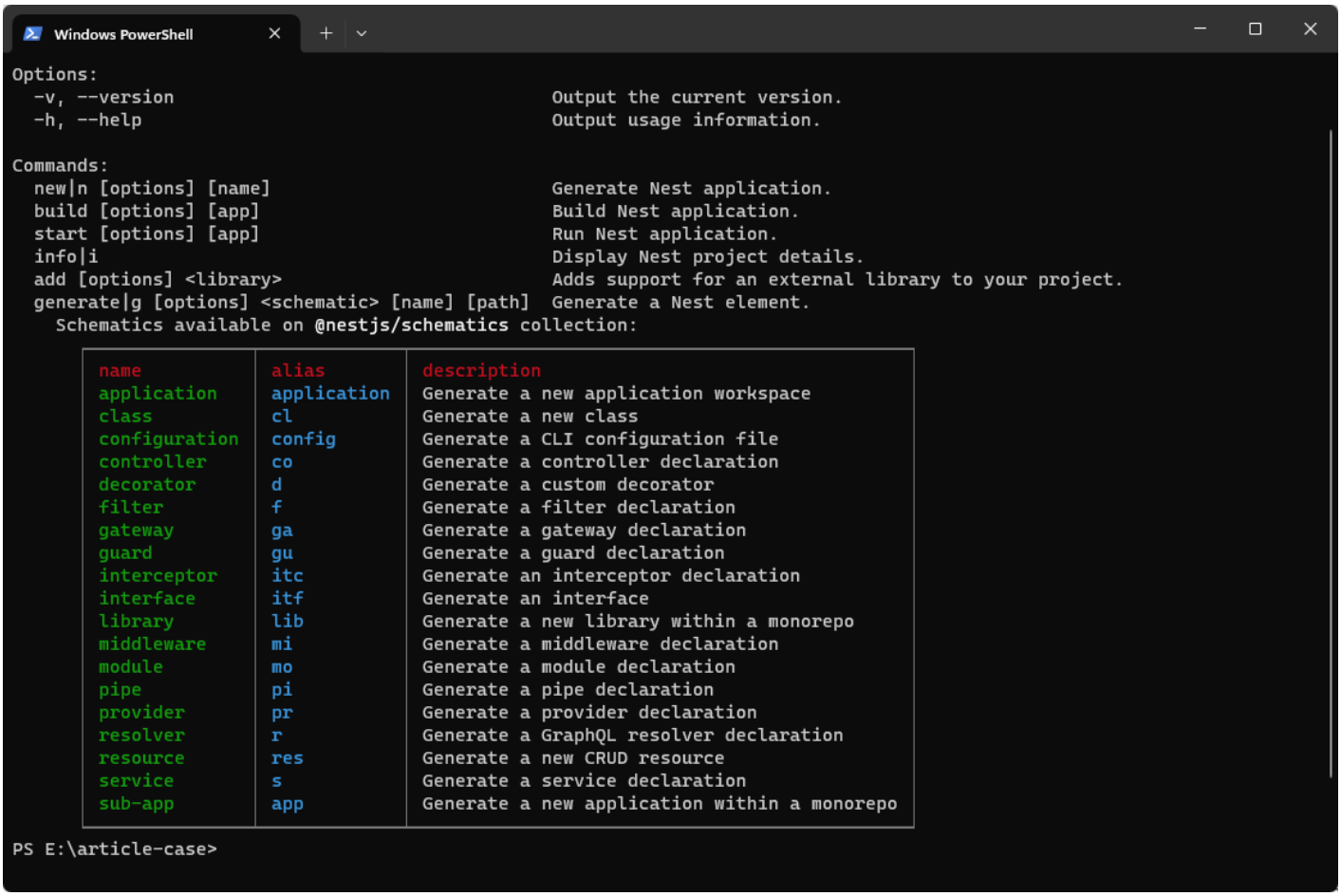
安装脚手架

确保您的操作系统上安装了Nodejs，安装版本必须 `>=16` 。

全局安装 `Nest Cli` 脚手架

```
npm i -g @nestjs/cli
```

安装完成运行 `nest -h` 命令验证是否安装成功，在控制台输出如下图所示则成功了



创建项目

用脚手架直接创建nest项目，运行命令

```
nest new project
```

创建成功后，打开项目代码，项目结构如下， `src` 目录中是核心文件。

```
src
 ├── app.controller.spec.ts
 ├── app.controller.ts
 ├── app.module.ts
 ├── app.service.ts
 └── main.ts
package.json
tsconfig.json
```

运行命令如下命令启动项目

```
npm run start
```

在开发时，运行热更新命令，已检测项目文本改动，自动更新服务

```
npm run start:dev
```

2．Controller控制器

在 NestJS 中，Controller（控制器）是应用程序的入口点，负责处理客户端发送的 HTTP 请求并返回响应。Controller 负责定义路由、接收请求参数以及调用服务层来处理业务逻辑。

Controller的主要作用包括：

- 定义路由：使用装饰器（如 `@Get()`，`@Post()`，`@Put()`，`@Delete()` 等）来指定与特定 URL 和 HTTP 方法关联的处理函数。
- 接受和验证请求参数：通过注入 `@Req()` 参数或使用 `@Body()`，`@Query()`，`@Param()` 等装饰器来获取请求中的信息，并进行必要的验证。
- 调用服务层方法：将请求参数传递给相应的服务类来执行业务逻辑。
- 返回响应：根据服务层返回的结果构建适当的响应对象，并将其返回给客户端。

创建控制器

每个控制器都是一个Class类，要让一个Class类成为一个控制器则必须使用 `@Controller()` 装饰器指定

```
@Controller()
export class AppController {}
```

装饰器 `@Controller()` 支持接收参数作为路由地址的前缀，如果没传参则默认是根路由 `/`

内部添加代码如下， 则可访问 `localhost:3000/` 在页面会显示字符串 `hello world`

```
@Controller()
export class AppController {
  @Get()
  getHello(): string {
    return 'hello world';
  }
}
```

如装饰器接收参数，如写成这样 `@Controller('/abc')` ，则需要访问 `localhost:3000/abc`

如何添加路由

NestJs的路由系统是随着控制器的创建自动形成，NestJs提供了如下装饰器生成路由：

- `@Get()` ：创建处理GET请求的路由地址
- `@Post()` ：创建处理POST请求的路由地址
- `@Put()` ：创建处理PUT请求的路由地址
- `@Delete()` ：创建处理PUT请求的路由地址
- `@Patch()` ：创建处理PATCH请求的路由地址
- `@Head()` ：创建处理HEAD请求的路由地址
- `@Options()` ：创建处理OPTIONS请求的路由地址

获取请求参数

- 获取GET请求的参数，使用 **`@Query()`** 装饰器

```
@Get('mGet')
handleGet(@Query() query){
    return query
}
```

- 获取POST请求的参数，使用 **`@Body()`** 装饰器

```
@Post('mPost')
handlePost(@Body() body){
    return body
}
```

- 获取PUT请求的参数，put请求的参数也是在请求体中，所以也使用 **`@Body()`** 装饰器

```
@Put('mPut')
handlePut(@Body() body){
    return body
}
```

- 获取DELETE请求的参数，请求参数作为路由的一部分，使用 `@Param()` 装饰器

```
@Delete('mDelete/:id')
handleDelete(@Param() body){
    return body
}
```

- 获取PATCH请求的参数，也使用 `@Body()` 装饰器

```
@Patch('mPatch')
handlePatch(@Body() body){
   return body
}
```

Nest.js遵循RESTful API设计原则，因此建议将资源的标识符作为路由的一部分，而不是将其放在请求体中。

响应数据给客户端

在Nest.js中，控制器（Controller）负责处理来自客户端的请求并返回响应。你可以直接从控制器的方法中返回数据，Nest.js会自动将其转换为HTTP响应。

```
@Controller()
export class AppController {
  @Get()
  findAll(): object {
    return { message: 'This action returns all cats' };
  }
}
```

在路由处理方法中直接使用 `return` 返回，会直接作为响应数据给到客户端。

如果需要更精细的控制响应，可以使用 `@Res()` 装饰器注入Response对象，可以使用这个对象来设置状态码、头部或发送自定义数据。

```
import { Controller, Get, Res } from '@nestjs/common';
import { Response } from 'express';
@Controller()
export class AppController {
  @Get()
  findAll(@Res() res: Response): void {
    res.status(200).json({ message: 'This action returns all cats' });
  }
}
```

3．Provider的用法

在NestJS中，Provider 是一个核心概念，它是一个被框架用来提供依赖注入（Dependency Injection）的组件。简而言之，Provider 是任何可以被框架实例化并注入到其他类中的对象。

几乎所有的NestJS类都可以被视为**Provider**，包括但不限于：

- Services服务
- Middleware中间件
- Filters过滤器
- Pipes管道
- Guards守卫
- Interceptors拦截器

标记为Provider

一个类要标记为Provider必须使用 `@Injectable()` 装饰器修饰。

比如声明一个User类，把它标记为Provider在User类中添加一点代码

```
import { Injectable } from '@nestjs/common';
@Injectable()
export class User {
  handleUserInfo(){
    return 'user info'
  }
}
```

将User这个provider注入依赖，在Controller控制器中通过构造函数注入

```
import {
  Controller, Get
} from '@nestjs/common';
import { User } from './user'
@Controller('user')
export class UserController {
  constructor(
    private readonly user:User
  ){}
  @Get()
  find(){
    return this.user.handleUserInfo()
  }
}
```

上面这种注入方式，改成如下方式更易懂，它们两种写法是等价的

```
import {
  Controller, Get
} from '@nestjs/common';
import { User } from './user'
@Controller('user')
export class UserController {
  private user: User;
  constructor(){
    this.user = new User()
  }
  @Get()
  find(){
    return this.user.handleUserInfo()
  }
}
```

最后，这个provider需要在 `Nest IoC` 容器中注册，也就是要把它塞入到 `app.module.ts` 的providers数组中

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { UserController } from './user/user.controller';
import { User } from './user/user';
@Module({
  imports: [],
  controllers: [AppController, UserController],
  providers: [AppService, User],
})
export class AppModule {}
```

那么当客户端向 `/user` 发起请求，会触发 **find()** 处理程序，find()的结果来自于provider的 **handleUserInfo()** 方法返回的内容。

自定义Provider

提供者需要在module中注册，也就是providers中

```
@Module({
    providers:[UserService]
})
```

providers接收的是一个提供者数组，它的完整写法如下：

```
@Module({
  providers: [{
    provide:UserService,
    useClass:UserService
  }],
})
```

- provide：是注入的令牌，即提供者名称
- useClass：注册的提供者类

自定义名称

提供者的名称可以自定义，设置了自定义名称，则在Controller中注入依赖时必须使用 `@Inject()` 装饰器来指定，否则无法找到对应的提供者。

如下代码，自定义provide名称为‘abc’

```
@Module({
  providers: [{
    provide:'abc',
    useClass:UserService
  }],
})
```

则在Controller中注册依赖要稍加改动

```
constructor(@Inject('abc') private readonly userService:UserService){}
```

自定义值

注册提供者，对象中还有一个 `useValue` 属性，用来注入常量值、将外部库放入 `Nest` 容器。

如下示例，注入一个常量信息，在Controller中使用

app.module.ts

```
@Module({
  providers: [{
    provide:'injectValue',
    useValue:{
      title:'注入常量数据',
      time:'2022-12-06'
    }
  }],
})
```

user.controller.ts

```
import {
  Controller, Get, Inject
} from '@nestjs/common';
@Controller('user')
export class UserController {
  constructor(
    @Inject('injectValue') private readonly info:{}
  ){}
  @Get()
  findAll(){
    return this.info
  }
}
```