Nestjs官方CLI

快速上手

全局安装CLI

```
$ npm i -g @nestjs/cli
$ nest new project-name
```

或者通过安装官方的starter起步项目：

```
$ git clone https://github.com/nestjs/typescript-starter.git project

$ cd project
$ npm install
$ npm run start
```

> 如果不想安装git仓库的历史依赖，可以使用`degit`：`npm i -g degit`
>
> 然后使用degit来下载仓库：`degit https://github.com/nestjs/typescript-starter.git`

官方的CLINest CLI是一个命令行界面工具，以帮助您初始化、开发和维护 `Nest` 应用程序。它以多种方式提供帮助，包括搭建项目、以开发模式为其提供服务，以及为生产分发构建和打包应用程序。它体现了最佳实践的架构模式，以构建良好的应用程序。

大多命令行工具可以使用`--help`来查看帮助：

```
→ nest --help
Usage: nest <command> [options]

Options:
  -v, --version                                   Output the current version.
  -h, --help                                      Output usage information.

Commands:
  new|n [options] [name]                          Generate Nest application.
  build [options] [app]                           Build Nest application.
  start [options] [app]                           Run Nest application.
  info|i                                          Display Nest project details.
  update|u [options]                              Update Nest dependencies.
  add [options] <library>                         Adds support for an external library to
your project.
  generate|g [options] <schematic> [name] [path]  Generate a Nest element.
    Available schematics:
```

```
┌────────────────┬────────────────┬──────────────────────────────────────────────┐
│ name           │ alias          │ description                                  │
│ application    │ application    │ Generate a new application workspace         │
│ class          │ cl             │ Generate a new class                         │
│ configuration  │ config         │ Generate a CLI configuration file            │
│ controller     │ co             │ Generate a controller declaration            │
│ decorator      │ d              │ Generate a custom decorator                  │
│ filter         │ f              │ Generate a filter declaration                │
│ gateway        │ ga             │ Generate a gateway declaration               │
│ guard          │ gu             │ Generate a guard declaration                 │
│ interceptor    │ in             │ Generate an interceptor declaration          │
│ interface      │ interface      │ Generate an interface                        │
│ middleware     │ mi             │ Generate a middleware declaration            │
│ module         │ mo             │ Generate a module declaration                │
│ pipe           │ pi             │ Generate a pipe declaration                  │
│ provider       │ pr             │ Generate a provider declaration              │
│ resolver       │ r              │ Generate a GraphQL resolver declaration      │
│ service        │ s              │ Generate a service declaration               │
│ library        │ lib            │ Generate a new library within a monorepo     │
│ sub-app        │ app            │ Generate a new application within a monorepo │
│ resource       │ res            │ Generate a new CRUD resource                 │
└────────────────┴────────────────┴──────────────────────────────────────────────┘
```

然后，如果想知道其中某一个子命令的用法，可以使用`nest <command> --help`的形式来进行查看：

```
# 例如：
➜ nest generate --help
```

特别说明：

名称	缩写	描述
application	application	生成一个新的应用工作区
class	cl	生成一个新的class
configuration	config	生成 CLI 配置文件
controller	co	生成一个控制器声明
decorator	d	生成一个自定义的装饰者
filter	f	生成一个过滤器声明
gateway	ga	生成网关
guard	gu	生成守卫
interceptor	in	生成拦截器
interface	interface	生成接口声明
middleware	mi	生成中间件声明
module	mo	生成一个模块声明
pipe	pi	生成管道声明
provider	pr	生成提供者声明
resolver	r	生成GraphQL resolver声明
service	s	生成服务
library	lib	生成一个monorepo库
sub-app	App	生成一个monorepo的应用
resource	Res	生成一个新的CURD资源

> 练习与作业：最开始使用了一个`new`命令，后面最常用的即是`generator`或`g`（简写）命令，可以对照着上表进行熟悉。

资源合集

官方示例：https://github.com/nestjs/nest/tree/master/sample

Awesome：https://github.com/nestjs/awesome-nestjs

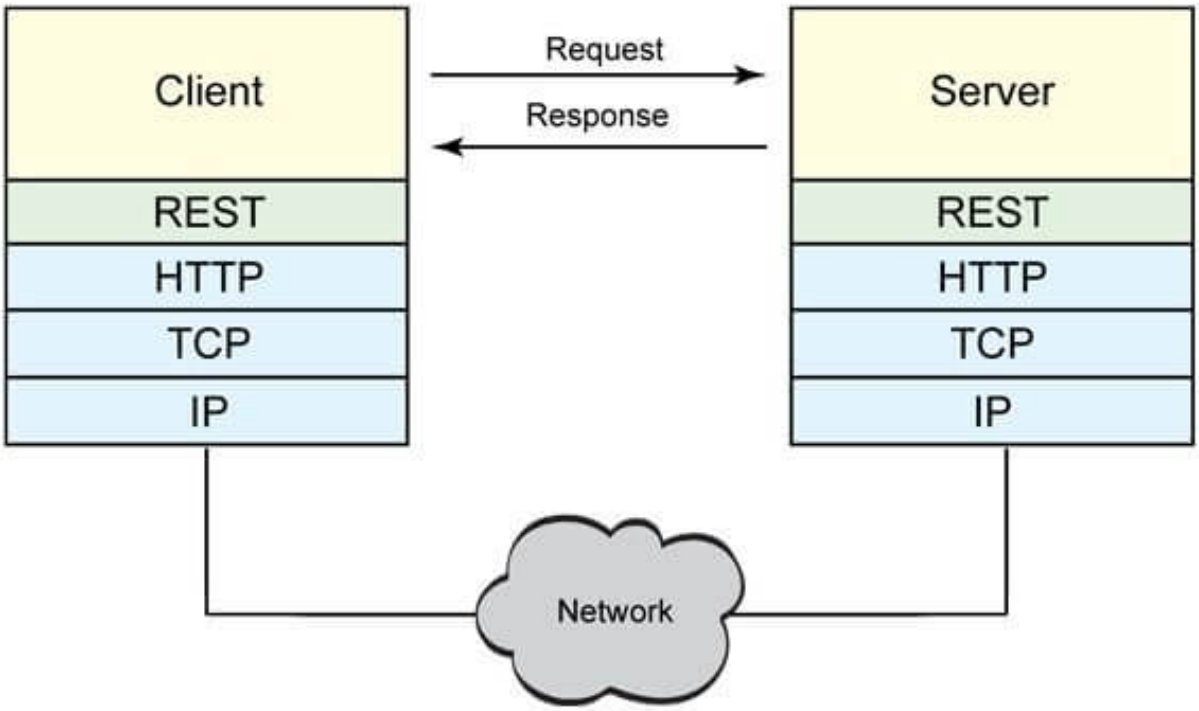
nestjs中文网：https://docs.nestjs.cn/9/introduction

语义化版本号：https://semver.org/lang/zh-CN/

RESTful API

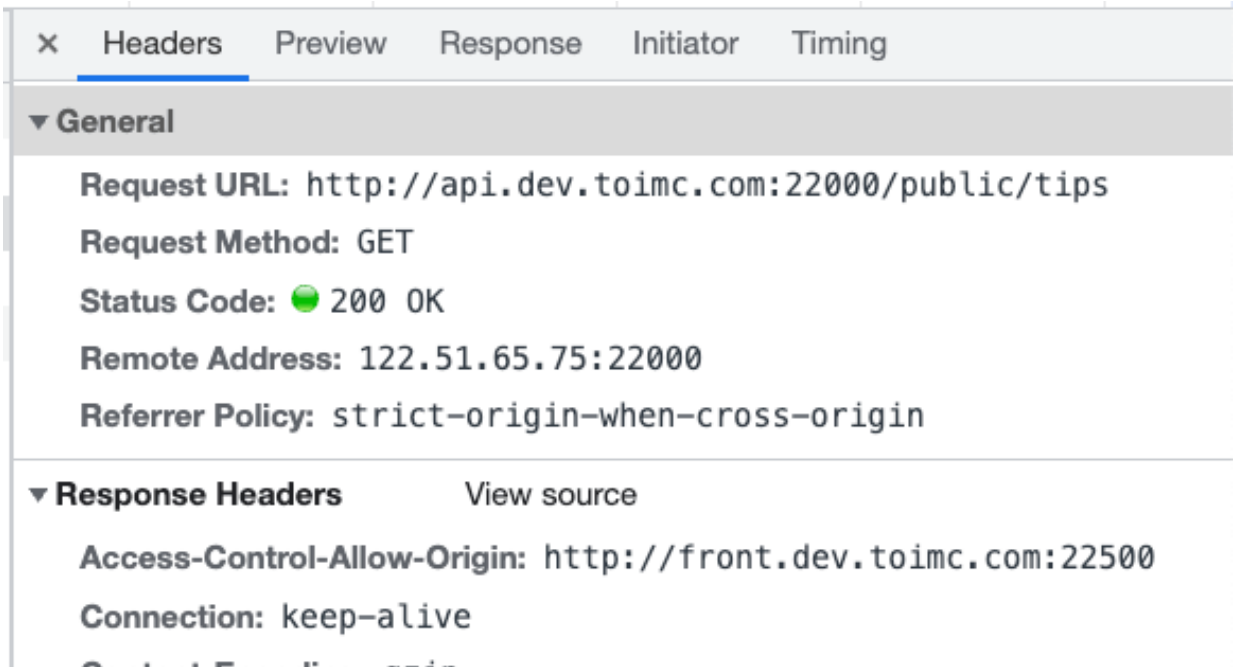
定义

REST -- REpresentational State Transfer 直接翻译：表现层状态转移，**在程序员眼中，它代表着是一种接口风格**。大多数业务应用程序必须与其他内部和第三方应用程序进行通信才能执行各种任务，所以才有一个“共识”的约定，这个约定（规则）就是RESTful API。



应用程序编程接口（API）由一组定义和协议组合而成，可用于构建和集成应用软件。

它长什么样？举例（通过Chrome调试工具NetWork选项卡查看）：



Content-Encoding: gzip
Content-Security-Policy: default-src 'self';base-uri 'self';block-all-mixed-content;font-src 'self' https: data:;frame-ancestors 'self';img-src 'self' data:;object-src 'none';script-src 'self';script-src-attr 'none';style-src 'self' https: 'unsafe-inline';upgrade-insecure-requests
Content-Type: application/json; charset=utf-8
Date: Fri, 12 Aug 2022 07:36:29 GMT
Expect-CT: max-age=0
Referrer-Policy: no-referrer
Strict-Transport-Security: max-age=15552000; includeSubDomains
Transfer-Encoding: chunked
Vary: Accept-Encoding, Origin
X-Content-Type-Options: nosniff
X-DNS-Prefetch-Control: off
X-Download-Options: noopen
X-Frame-Options: SAMEORIGIN
X-Permitted-Cross-Domain-Policies: none
X-XSS-Protection: 0

▼ Request Headers View source

Accept: application/json, text/plain, */*
Accept-Encoding: gzip, deflate
Accept-Language: zh-CN,zh;q=0.9,en;q=0.8
Connection: keep-alive
DNT: 1
Host: api.dev.toimc.com:22000
Origin: http://front.dev.toimc.com:22500
Referer: http://front.dev.toimc.com:22500/
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/104.0.0.0 Safari/537.36

设计

RESTful API需要设计序言、全局（错误码、请求BaseUrl、Proxy等）参数、修改记录以及按照功能划分的接口描述。

下面来介绍一份标准的接口设计中，重要的组成部分：

- 接口描述；
- 请求URL；
- 请求方式：POST/GET/DELETE/PUT；
- 参数：Body 或者 Params 或者 Headers参数（JWT Token）及参数说明；
- 返回示例；
- 返回参数说明；

示例：

简要描述：

- 用户注册接口

请求URL：

- `http://xx.com/api/user/register`

请求方式：

- POST

参数：

参数名	必选	类型	说明
username	是	string	用户名
password	是	string	密码
name	否	string	昵称

返回示例

```
{
  "error_code": 0,
  "data": {
    "uid": "1",
    "username": "12154545",
    "name": "toimc",
    "groupid": 2 ,
    "reg_time": "1436864169",
    "last_login_time": "0",
  }
}
```

返回参数说明

参数名	类型	说明
groupid	int	用户组id，1：超级管理员；2：普通用户

备注

- 更多返回错误代码请看首页的错误代码描述

测试工具

常见的测试工具：

- 客户端：Postman、Apifox、eolink、DOClever
- 网页端：Chrome插件Talend API、Postman Interceptor
- 插件工具：VSCode插件REST Client、HTTP Client

第一个Nestjs应用：Hello World

参考：快速上手 创建初始化项目：

```
# 启动项目

npm run start

# or
yarn start

# or
pnpm start
```

在应用程序运行后, 打开浏览器并访问 `http://localhost:3000/`，应该可能看到 `Hello world!` 信息。

项目的基本目录：

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

文件说明：

文件名	说明
app.controller.ts	带有单个路由的基本控制器示例。
app.controller.spec.ts	对于基本控制器的单元测试样例
app.module.ts	应用程序的根模块。
app.service.ts	带有单个方法的基本服务
main.ts	应用程序入口文件。它使用 `NestFactory` 用来创建 Nest 应用实例。

最佳实践

工程目录

为了去理解Python的语言设计之美，其实更要理解这样的一句话“约定大于配置”，好的工程化目录（约定）能够很好的提升项目的可维护性。

作者推荐

在官方的issues中，我们可以找到一些提示：Best scalable project structure #2249 这里有作者的回复。

```
- src
  - core
  - common
    - middleware
    - interceptors
    - guards
  - user
      - interceptors (scoped interceptors)
    - user.controller.ts
    - user.model.ts
  - store
    - store.controller.ts
    - store.model.ts
```

- 可以使用monorepo的方法——在一个repo中创建两个项目，并在它们之间共享共同的东西，如库/包。
- 没有模块目录，按照功能进行划分。
- 把通用/核心的东西归为单独的目录：common，比如：拦截器/守卫/管道

参考项目

第一个参考项目

技术栈：Nest + sequelize-typescript + JWT + Jest + Swagger

项目地址：kentloog/nestjs-sequelize-typescript

```
.
├── README.md
├── assets
│   └── logo.png
├── config
│   ├── config.development.ts
│   └── config.production.ts
├── config.ts
├── db
│   ├── config.ts
│   ├── migrations
│   │   └── 20190128160000-create-table-user.js
│   └── seeders-dev
│       └── 20190129093300-test-data-users.js
├── nest-cli.json
├── nodemon-debug.json
├── nodemon.json
├── package-lock.json
├── package.json
```

```
├── src
│   ├── app.module.ts
│   ├── database
│   │   ├── database.module.ts
│   │   └── database.providers.ts
│   ├── main.ts
│   ├── posts
│   │   ├── dto
│   │   │   ├── create-post.dto.ts
│   │   │   ├── post.dto.ts
│   │   │   └── update-post.dto.ts
│   │   ├── post.entity.ts
│   │   ├── posts.controller.ts
│   │   ├── posts.module.ts
│   │   ├── posts.providers.ts
│   │   └── posts.service.ts
│   ├── shared
│   │   ├── config
│   │   │   └── config.service.ts
│   │   ├── enum
│   │   │   └── gender.ts
│   │   └── shared.module.ts
│   ├── swagger.ts
│   └── users
│       ├── auth
│       │   ├── jwt-payload.model.ts
│       │   └── jwt-strategy.ts
│       ├── dto
│       │   ├── create-user.dto.ts
│       │   ├── update-user.dto.ts
│       │   ├── user-login-request.dto.ts
│       │   ├── user-login-response.dto.ts
│       │   └── user.dto.ts
│       ├── user.entity.ts
│       ├── users.controller.ts
│       ├── users.module.ts
│       ├── users.providers.ts
│       └── users.service.ts
├── test
│   ├── app.e2e-spec.ts
│   ├── jest-e2e.json
│   └── test-data.ts
├── tsconfig.build.json
├── tsconfig.json
└── tslint.json
```

特点：

- 项目文档及相关的资源在根目录
- 数据库及项目配置会放在根目录（细节：数据库升级文件）
- `src`中会对功能进行划分建不同的文件夹`users`、`posts`

- 单个功能文件夹中，会包括一个完整CURD的相关文件(dto/controller/module/providers/service)
- 抽离公共配置到`shared`文件夹

第二个参考项目

技术栈：具有AWS Lambda，DynamoDB，DynamoDB Streams的完全无服务器生产应用程序

项目地址：International-Slackline-Association/Rankings-Backend

```
.
├── LICENSE
├── README.md
├── docs
│   ├── AWS_Architecture.png
│   ├── Development\ Notes.md
│   └── GourceOutput.png
├── jest.config.js
├── package-lock.json
├── package.json
├── serverless
│   ├── environment.yml
│   └── secrets.example.yml
├── serverless.yml
├── src
│   ├── api
│   │   ├── admin
│   │   │   ├── api.module.ts
│   │   │   ├── athlete
│   │   │   ├── contest
│   │   │   ├── database.module.ts
│   │   │   ├── index.ts
│   │   │   ├── results
│   │   │   └── submit
│   │   └── webapp
│   │       ├── api.module.ts
│   │       ├── athlete
│   │       ├── contest
│   │       ├── country
│   │       ├── database.module.ts
│   │       ├── index.ts
│   │       ├── nestjsTest.controller.ts
│   │       └── rankings
│   ├── core
│   │   ├── athlete
│   │   │   ├── athlete.service.ts
│   │   │   ├── entity
│   │   │   ├── interfaces
│   │   │   └── rankings.service.ts
│   │   ├── aws
│   │   │   ├── aws.module.ts
```

```
│   │   │   ├── aws.services.interface.ts
│   │   │   └── aws.services.ts
│   │   ├── category
│   │   │   └── categories.service.ts
│   │   ├── contest
│   │   │   ├── contest.service.ts
│   │   │   ├── entity
│   │   │   └── points-calculator.service.ts
│   │   └── database
│   │       ├── database.module.ts
│   │       ├── database.service.ts
│   │       ├── dynamodb
│   │       ├── redis
│   │       └── test
│   ├── cron-job
│   │   ├── cron-job.module.ts
│   │   ├── cron-job.service.ts
│   │   ├── cron-job.spec.ts
│   │   ├── database.module.ts
│   │   └── index.ts
│   ├── dynamodb-streams
│   │   ├── athlete
│   │   │   ├── athlete-contest-record.service.ts
│   │   │   ├── athlete-details-record.service.ts
│   │   │   └── athlete-records.module.ts
│   │   ├── contest
│   │   │   ├── contest-record.service.ts
│   │   │   └── contest-records.module.ts
│   │   ├── database.module.ts
│   │   ├── dynamodb-streams.module.ts
│   │   ├── dynamodb-streams.service.ts
│   │   ├── index.ts
│   │   ├── test
│   │   │   ├── contest-modifications.spec.ts
│   │   │   └── lambda-trigger.ts
│   │   └── utils.ts
│   ├── image-resizer
│   │   ├── S3Events.module.ts
│   │   ├── S3Events.service.ts
│   │   ├── database.module.ts
│   │   ├── index.ts
│   │   ├── test
│   │   │   ├── lambda-trigger.ts
│   │   │   └── s3-image-put.spec.ts
│   │   └── thumbnail-creator
│   │       ├── imagemagick.ts
│   │       ├── s3.service.ts
│   │       ├── thumbnail-creator.module.ts
│   │       └── thumbnail-creator.service.ts
│   └── shared
│       ├── constants.ts
│       ├── decorators
```

```
│       │   └── roles.decorator.ts
│       ├── enums
│       │   ├── contestType-utility.ts
│       │   ├── discipline-utility.ts
│       │   ├── enums-utility.ts
│       │   └── index.ts
│       ├── env_variables.ts
│       ├── exceptions
│       │   ├── api.error.ts
│       │   └── api.exceptions.ts
│       ├── extensions.ts
│       ├── filters
│       │   └── exception.filter.ts
│       ├── generators
│       │   └── id.generator.ts
│       ├── guards
│       │   └── roles.guard.ts
│       ├── index.ts
│       ├── logger.ts
│       ├── pipes
│       │   └── JoiValidation.pipe.ts
│       ├── types
│       │   ├── express.d.ts
│       │   ├── extensions.d.ts
│       │   └── shared.d.ts
│       └── utils.ts
├── test
│   ├── jest-e2e.json
│   └── test-setup.ts
├── tsconfig.json
├── tslint.json
└── webpack
    ├── webpack.config.Dev.js
    ├── webpack.config.Prod.js
    ├── webpack.config.Test.js
    └── webpack.config.base.js
```

特点：

- 根目录中存放webpack、微服务配置 + 项目文档
- `src`中会对功能进行划分建不同的文件夹：`api`、`core`、`dynamodb-stream`、`image-resizer`
- 在核心模块中，按照功能模块进划分，与之相关的entity、service放置在同一文件夹中
- 抽离公共配置到`shared`文件夹：常量、自定义的装饰器、统一错误处理、过滤器、生成器、守卫、日志服务

第三个参考项目：

技术栈：使用 NestJS 的 Blog/CMS， RESTful API 服务端应用

项目地址：[surmon-china/nodepressTemplate]

```
.
├── API_DOC.md
├── CHANGELOG.md
├── LICENSE
├── README.md
├── classified
├── cspell.json
├── dbbackup
├── logo.png
├── logo.psd
├── nest-cli.json
├── package.json
├── scripts
│   ├── README.md
│   ├── dbbackup.sh
│   ├── dbrecover.sh
│   └── deploy.sh
├── src
│   ├── app.config.ts
│   ├── app.controller.spec.ts
│   ├── app.controller.ts
│   ├── app.environment.ts
│   ├── app.module.ts
│   ├── constants
│   │   ├── cache.constant.ts
│   │   ├── meta.constant.ts
│   │   ├── system.constant.ts
│   │   └── text.constant.ts
│   ├── decorators
│   │   ├── cache.decorator.ts
│   │   ├── http.decorator.ts
│   │   └── query-params.decorator.ts
│   ├── errors
│   │   ├── bad-request.error.ts
│   │   ├── custom.error.ts
│   │   ├── forbidden.error.ts
│   │   ├── unauthorized.error.ts
│   │   └── validation.error.ts
│   ├── filters
│   │   └── error.filter.ts
│   ├── guards
│   │   ├── auth.guard.ts
│   │   └── humanized-auth.guard.ts
│   ├── interceptors
│   │   ├── cache.interceptor.ts
│   │   ├── error.interceptor.ts
│   │   ├── logging.interceptor.ts
│   │   └── transform.interceptor.ts
│   ├── interfaces
│   │   ├── http.interface.ts
│   │   ├── mongoose.interface.ts
│   │   └── state.interface.ts
```

```
│   ├── main.ts
│   ├── middlewares
│   │   ├── cors.middleware.ts
│   │   └── origin.middleware.ts
│   ├── models
│   │   └── extend.model.ts
│   ├── modules
│   │   ├── announcement
│   │   │   ├── announcement.controller.ts
│   │   │   ├── announcement.model.ts
│   │   │   ├── announcement.module.ts
│   │   │   └── announcement.service.ts
│   │   ├── article
│   │   │   ├── article.controller.ts
│   │   │   ├── article.model.ts
│   │   │   ├── article.module.ts
│   │   │   └── article.service.ts
│   │   ├── auth
│   │   │   ├── auth.controller.ts
│   │   │   ├── auth.interface.ts
│   │   │   ├── auth.model.ts
│   │   │   ├── auth.module.ts
│   │   │   ├── auth.service.ts
│   │   │   └── jwt.strategy.ts
│   │   ├── category
│   │   │   ├── category.controller.ts
│   │   │   ├── category.model.ts
│   │   │   ├── category.module.ts
│   │   │   └── category.service.ts
│   │   ├── comment
│   │   │   ├── comment.controller.ts
│   │   │   ├── comment.model.ts
│   │   │   ├── comment.module.ts
│   │   │   └── comment.service.ts
│   │   ├── expansion
│   │   │   ├── expansion.controller.ts
│   │   │   ├── expansion.module.ts
│   │   │   ├── expansion.service.dbbackup.ts
│   │   │   └── expansion.service.statistic.ts
│   │   ├── like
│   │   │   ├── like.controller.ts
│   │   │   ├── like.module.ts
│   │   │   └── like.service.ts
│   │   ├── option
│   │   │   ├── option.controller.ts
│   │   │   ├── option.model.ts
│   │   │   ├── option.module.ts
│   │   │   └── option.service.ts
│   │   ├── syndication
│   │   │   ├── syndication.controller.ts
│   │   │   ├── syndication.module.ts
│   │   │   └── syndication.service.ts
```

```
│   │   └── tag
│   │       ├── tag.controller.ts
│   │       ├── tag.model.ts
│   │       ├── tag.module.ts
│   │       └── tag.service.ts
│   ├── pipes
│   │   └── validation.pipe.ts
│   ├── processors
│   │   ├── cache
│   │   │   ├── cache.config.service.ts
│   │   │   ├── cache.module.ts
│   │   │   └── cache.service.ts
│   │   ├── database
│   │   │   ├── database.module.ts
│   │   │   └── database.provider.ts
│   │   └── helper
│   │       ├── helper.module.ts
│   │       ├── helper.service.akismet.ts
│   │       ├── helper.service.cs.ts
│   │       ├── helper.service.email.ts
│   │       ├── helper.service.google.ts
│   │       ├── helper.service.ip.ts
│   │       └── helper.service.seo.ts
│   └── transformers
│       ├── codec.transformer.ts
│       ├── error.transformer.ts
│       ├── model.transformer.ts
│       ├── mongoose.transformer.ts
│       └── urlmap.transformer.ts
├── test
│   ├── app.e2e-spec.ts
│   └── jest-e2e.json
├── tsconfig.build.json
├── tsconfig.json
├── tsconfig.spec.json
└── yarn.lock
```

特点：

- 项目文档及相关的资源在根目录
- `src`中`modules`会对功能进行划分建不同的文件夹
- 单个功能文件夹中，会包括一个完整CURD的相关文件(model/controller/module/service)
- 把公共的代码（按照nestjs逻辑分层）拆成单独的文件夹`guards`、`filters`、`decorators`、`interceptors`、`interfaces`、`errors`

项目：[CatsMiaow/node-nestjs-structure]

下面的项目结构：

```
+-- bin                  // Custom tasks
+-- dist                 // Source build
+-- public               // Static Files
+-- src
|   +-- config           // Environment Configuration
|   +-- entity           // TypeORM Entities generated by `typeorm-model-generator` module
|   +-- auth             // Authentication
|   +-- common           // Global Nest Module
|   |   +-- constants    // Constant value and Enum
|   |   +-- controllers  // Nest Controllers
|   |   +-- decorators   // Nest Decorators
|   |   +-- dto          // DTO (Data Transfer Object) Schema, Validation
|   |   +-- filters      // Nest Filters
|   |   +-- guards       // Nest Guards
|   |   +-- interceptors // Nest Interceptors
|   |   +-- interfaces   // TypeScript Interfaces
|   |   +-- middleware   // Nest Middleware
|   |   +-- pipes        // Nest Pipes
|   |   +-- providers    // Nest Providers
|   |   +-- *            // models, repositories, services...
|   +-- shared           // Shared Nest Modules
|   +-- gql              // GraphQL Structure Sample
|   +-- *                // Other Nest Modules, non-global, same as common structure above
+-- test                 // Jest testing
+-- typings              // Modules and global type definitions
```

如果是功能模块：

```
// Module structure
// Add folders according to module scale. If it's small, you don't need to add folders.
+-- src/greeter
|   +-- *                // folders
|   +-- greeter.constant.ts
|   +-- greeter.controller.ts
|   +-- greeter.service.ts
|   +-- greeter.module.ts
|   +-- greeter.*.ts
|   +-- index.ts
```

特点：

- 项目文档及相关的资源在根目录，包括`typings`、`test`、`bin`
- `src`中会对功能进行划分建不同的文件夹
- 抽离公共代码到`common`文件夹，配置文件放在`config`文件夹，实体类放置在`entity`中
- 鉴权相关的逻辑放在`auth`
- 把同类的`guards`、`filters`、`decorators`、`interceptors`、`interfaces`、`errors`存放在`common`文件夹中

代码规范（风格指南）

我们对Angular风格指南进行了摘抄，如下：

> 参考：Angular风格指南

总则

坚持每个文件只定义一样东西（例如服务或组件）

考虑把文件大小限制在 400 行代码以内

坚持定义简单函数

考虑限制在 75 行之内

命名

坚持所有符号使用一致的命名规则

坚持遵循同一个模式来描述符号的特性和类型

使用点和横杠来分隔文件名

坚持 在描述性名字中，用横杠来分隔单词。

坚持使用点来分隔描述性名字和类型。

坚持遵循先描述组件特性，再描述它的类型的模式，对所有组件使用一致的类型命名规则。推荐的模式为 `feature.type.ts`。

坚持使用惯用的后缀来描述类型，包括 `*.service`、`*.component`、`*.pipe`、`.module`、`.directive`。 必要时可以创建更多类型名，但必须注意，不要创建太多。

符号名与文件名

坚持为所有东西使用一致的命名约定，以它们所代表的东西命名。

坚持使用**大写驼峰命名法来命名类**

坚持匹配符号名与它所在的文件名

坚持在符号名后面追加约定的类型后缀（例如 `Component`、`Directive`、`Module`、`Pipe`、`Service`）。

坚持在文件名后面追加约定的类型后缀（例如 `.component.ts`、`.directive.ts`、`.module.ts`、`.pipe.ts`、`.service.ts`）

坚持使用*中线命名法（dashed-case）*或叫*烤串命名法（kebab-case）*来命名组件的元素选择器。

服务名&管道名

坚持使用一致的规则命名服务，以它们的特性来命名

坚持为服务的类名加上 `Service` 后缀。 例如，获取数据或英雄列表的服务应该命名为 `DataService` 或 `HeroService`

坚持为所有管道使用一致的命名约定，用它们的特性来命名。 管道类名应该使用 UpperCamelCase（类名的通用约定），而相应的 `name` 字符串应该使用 lowerCamelCase。 `name` 字符串中不应该使用中线（“中线格式”或“烤串格式”）。例如：

`ellipsis.pipe.ts`

```
@Pipe({ name: 'ellipsis' })
export class EllipsisPipe implements PipeTransform { }
```

和

`init-caps.pipe.ts`

```
@Pipe({ name: 'initCaps' })
export class InitCapsPipe implements PipeTransform { }
```

坚持在模块中只包含模块间的依赖关系，所有其它逻辑都应该放到服务中

坚持把可复用的逻辑放到服务中，保持组件简单，聚焦于它们预期目的

坚持在同一个注入器内，把服务当做单例使用，用它们来共享数据和功能

坚持创建封装在上下文中的单一职责的服务

坚持当服务成长到超出单一用途时，创建一个新服务

坚持把数据操作和与数据交互的逻辑重构到服务里。

引导

坚持把应用的引导程序和平台相关的逻辑放到名为 `main.ts` 的文件里

坚持在引导逻辑中包含错误处理代码

避免把应用逻辑放在 `main.ts` 中，而应放在组件或服务里

测试文件名

单元测试：

坚持测试规格文件名与被测试组件文件名相同

坚持测试规格文件名添加 `.spec` 后缀

端到端的测试：

坚持端到端测试规格文件和它们所测试的特性同名，添加 `.e2e-spec` 后缀，或者放在特定的文件夹中。

其他原则

- 定位：

  坚持直观、简单和快速地定位代码。

- 识别：

  坚持命名文件到这个程度：看到名字立刻知道它包含了什么，代表了什么。

  坚持文件名要具有说明性，确保文件中只包含一个组件。

  避免创建包含多个组件、服务或者混合体的文件。

- 扁平

  坚持尽可能保持扁平的目录结构。

  考虑当同一目录下达到 7 个或更多个文件时创建子目录。

  考虑配置 IDE，以隐藏无关的文件，例如生成出来的 `.js` 文件和 `.js.map` 文件等。

- T-DRY

  坚持 DRY（Don't Repeat Yourself，不重复自己）。

  避免过度 DRY，以致牺牲了阅读性

- 代码结构

  坚持从零开始，但要考虑应用程序接下来的路往哪儿走

  坚持有一个近期实施方案和一个长期的愿景

  坚持把所有源代码都放到名为 `src` 的目录里

  坚持如果组件具有多个伴生文件 (`.ts`、`.html`、`.css` 和 `.spec`)，就为它创建一个文件夹

- ESLint

  坚持使用VSCode等IDE、配合ESLint + Prettier等工具来整理代码格式、检查代码风格问题。

示例

```
.
├── README.md
├── dist
├── docker-compose.yml
├── nest-cli.json
├── package.json
├── pnpm-lock.yaml
├── prisma
│   ├── migrations
│   └── schema.prisma
├── src
│   ├── app.module.ts
│   ├── auth
│   │   ├── auth.controller.ts
```

```
│   │   ├── auth.module.ts
│   │   ├── auth.service.ts
│   │   ├── decorator
│   │   │   ├── get-user.decortor.ts
│   │   │   └── index.ts
│   │   ├── dto
│   │   │   ├── auto.dto.ts
│   │   │   └── index.ts
│   │   ├── guard
│   │   │   ├── index.ts
│   │   │   └── jwt.guard.ts
│   │   └── strategy
│   │       ├── index.ts
│   │       └── jwt.strategy.ts
│   ├── bookmark
│   │   ├── bookmark.controller.ts
│   │   ├── bookmark.module.ts
│   │   ├── bookmark.service.ts
│   │   └── dto
│   │       ├── create-bookmark.dto.ts
│   │       ├── edit-bookmark.dto.ts
│   │       └── index.ts
│   ├── main.ts
│   ├── prisma
│   │   ├── prisma.module.ts
│   │   └── prisma.service.ts
│   ├── repl.ts
│   └── user
│       ├── dto
│       │   ├── edit-user.dto.ts
│       │   └── index.ts
│       ├── user.controller.ts
│       ├── user.module.ts
│       └── user.service.ts
├── test
│   ├── app.e2e-spec.ts
│   └── jest-e2e.json
├── tsconfig.build.json
└── tsconfig.json
```

CLI初试

目标：

创建一个`GET`请求的接口`/range`，参数是params的`num`，当传递`num`为`5`时，响应客户端，一组数据：

```
{
  code: 0,
  msg: "请求成功！",
  data: ["1", "2", "3", "4", "5"]
}
```

注意：

- 第三方库产生的arr；
- arr里面是`string`；
- 加分：判断`num`参数传递是否正确；
- 加分：代码文件名、方法名等