目　录

致谢

当前文档 《nestjs 5.0 中文文档》 由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2018-05-16。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常生活、工作和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

文档地址: http://www.bookstack.cn/books/nestjs-5.0

书栈官网: http://www.bookstack.cn

书栈开源: https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

README

介绍

Nest是构建高效，可扩展的 Node.js Web 应用程序的框架。 它使用现代的 JavaScript 或 TypeScript（保留与纯 JavaScript 的兼容性），并结合 OOP（面向对象编程），FP（函数式编程）和 FRP（函数响应式编程）的元素。在底层，Nest 使用了 Express，可以方便地使用各种可用的第三方插件。

设计哲学

近几年，由于 Node.js，JavaScript 已经成为 Web 前端和后端应用程序的“通用语言”，并且有了 Angular，React 和 Vue 等令人耳目一新的项目，提高了开发人员的生产力，使得可以快速构建可测试的且可扩展的前端应用程序。 然而，在服务器端，虽然有很多优秀的库、helper 和 Node 工具，但是它们都没有有效地解决主要问题 - 架构。

Nest 旨在提供一个开箱即用的应用程序体系结构，允许轻松创建高度可测试，可扩展，松散耦合且易于维护的应用程序。

来源(书栈小编注)

> *https://github.com/nestcn/docs*

概述

第一步

第一步

在这一组文章中，您将了解 Nest 的核心基础知识。主要是了解基本的 nest 应用程序构建模块。您将构建一个基本的 CRUD 应用程序，其中的功能涵盖了大量的入门基础。

语言

我们爱上了 TypeScript，但最重要的是，我们喜欢 Node.js。 这就是为什么 Nest 兼容 TypeScript 和纯 JavaScript 。 Nest 正利用最新的语言功能，所以要使用简单的 JavaScript 框架，我们需要一个 Babel 编译器。

在文章中，我们主要使用 TypeScript ，但是当它包含一些 Typescript 特定的表达式时，您总是可以将代码片段切换到 JavaScript 版本。

【译者注：由于 nest.js 对 ts 特性支持更好，中文文档只翻译 Typescript】

先决条件

请确保您的操作系统上安装了 Node.js（> = 6.11.0）。

建立

使用 Nest CLI 建立新项目非常简单。 只要确保你已经安装了 npm，然后在你的 OS 终端中使用以下命令：

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

`project` 目录将在 `src` 目录中包含几个核心文件。

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

按照约定，新创建的模块应该有一个专用目录。

main.ts	应用程序入口文件。它使用 `NestFactory` 用来创建 Nest 应用实例。
app.module.ts	定义 `AppModule` 应用程序的根模块。
app.controller.ts	带有单个路由的基本控制器示例。

`main.ts` 包含一个异步函数，它负责引导我们的应用程序：

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

async function bootstrap() {
  const app = await NestFactory.create(ApplicationModule);
  await app.listen(3000);
}
bootstrap();
```

要创建一个 Nest 应用实例，我们使用了 `NestFactory` 。 `create()` 方法返回一个实现 `INestApplication` 接口的对象，并提供一组可用的方法，在后面的章节中将对此进行详细描述。

运行应用程序

安装过程完成后，您可以运行以下命令启动 HTTP 服务器：

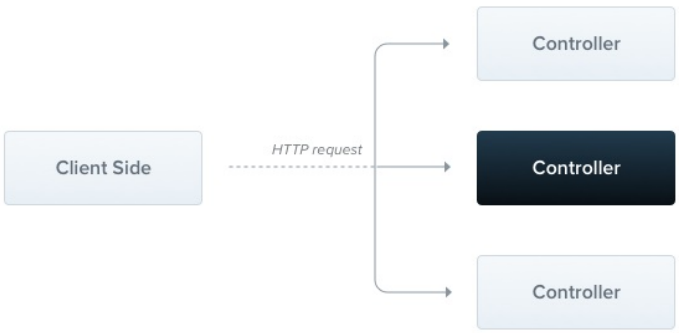
```
$ npm run start
```

此命令在 `src` 目录中的 `main.ts` 文件中定义的端口上启动 HTTP 服务器。在应用程序运行时，打开浏览器并访问 `http//localhost:3000/` 。 你应该看到 `Hello world!` 信息。

控制器

控制器

控制器层负责处理传入的请求， 并返回对客户端的响应。



为了创建一个基本的控制器，我们必须使用 `装饰器` 。多亏了他们， Nest 知道如何将控制器映射到相应的路由。

> *cats.controller.ts*

```
import { Controller, Get } from '@nestjs/common';

@Controller('cats')
export class CatsController {
  @Get()
  findAll() {
    return [];
  }
}
```

装饰器

在上面的例子中，我们使用了定义基本控制器所需的 `@Controller('cats')` 装饰器 。 这个装饰器是强制性的。 `cats` 是每个在类中注册的每个路由的可选前缀。使用前缀可以避免在所有路由共享通用前缀时出现冲突的情况， `findAll()` 方法附近的 `@Get()` 修饰符告诉 Nest 创建此路由路径的端点，并将每个相应的请求映射到此处理程序。由于我们为每个路由（ `cats` ）声明了前缀，所以 Nest 会在这里映射每个 `/cats` 的 GET 请求。

当客户端调用此端点时， Nest 将返回 200 状态码和解析的 JSON，因此在本例中-它只是一个空数组。这怎么可能？

有 `两种` 可能的方法来处理响应：

标准（推荐）	我们对处理程序的处理方式与普通函数相同。当我们返回 JavaScript 对象或数组时，它会自动转换为 JSON。当我们返回字符串， Nest 将只发送一个字符串而不尝试解析它。
	此外， 响应状态代码在默认情况下总是 200，除了 POST 请求外，此时它是 201。我们可以通过在处理程序层添加 `@HttpCode(...)` 装饰器来轻松地更改此行为。
库专用	我们可以使用库特定的响应对象，，例如findAll（@Res（）response）。

我们可以使用「express」响应对象，我们可以在函数签名中使用 `@Res()` 装饰器注入，例如 `findAll (@Res() response)` 。

!> 注意！ 禁止同时使用这两种方法。 Nest 检测处理程序是否正在使用 `@Res ()` 或 `@Next ()` ，如果是这样,此单个路由的「标准」方式将被禁用。

Request 对象

许多端点需要访问客户端的请求细节。 实际上，Nest 正使用 特定的库 请求对象。 我们可以强制 Nest 使用 `@Req ()` 装饰器将请求对象注入处理程序。

`cats.controller.ts`

```
import { Controller, Get, Req } from '@nestjs/common';

@Controller('cats')
export class CatsController {
  @Get()
  findAll(@Req() request) {
    return [];
  }
}
```

「Request」对象表示 HTTP 请求，并具有「Request」查询字符串，参数，HTTP 标头 和 正文的属性（在这里阅读更多），但在大多数情况下，不必手动获取它们。 我们可以使用专用的装饰器，比如 `@Body()` 或 可以直接使用的装饰器 `@Query()` 。 下面是装饰器和普通表达对象的比较。

`@Request()`	`req`
`@Response()`	`res`

`@Next()`	next
`@Session()`	req.session
`@Param(param?: string)`	req.params / req.params[param]
`@Body(param?: string)`	req.body / req.body[param]
`@Query(param?: string)`	req.query / req.query[param]
`@Headers(param?: string)`	req.headers / req.headers[param]

更多端点

我们已经创建了一个端点来获取数据（GET 路由）。 提供创建新记录的方式也是很好的。让我们创建 POST 处理程序：

> *cats.controller.ts*

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

@Controller('cats')
export class CatsController {
  @Post()
  create() {}

  @Get()
  findAll() {
    return [];
  }
}
```

这很容易。 Nest 以同样的方式提供了其他的端点装饰器 - `@Put()`， `@Delete()`， `@Patch()`， `@Options()`， `@Head()` 和 `@All()`。

状态码操作

> 如前面所说，默认情况下，响应的状态码总是*200*，除了 *POST* 请求外，此时它是*201*，我们可以通过在处理程序层添加 `@HttpCode(...)` 装饰器来轻松更改此行为。

`cats.controller.ts`

```
import { Controller, Get, Post, HttpCode } from '@nestjs/common';

@Controller('cats')
export class CatsController {
  @HttpCode(204)
  @Post()
  create() {}

  @Get()
  findAll() {
    return [];
  }
}
```

路由参数

当需要将动态数据作为 URL 的一部分接受时，使用静态路径的路由无法提供帮助。要使用路由参数定义路由，只需在路由的路径中指定路由参数，如下所示。

```
@Get(':id')
findOne(@Param() params) {
  console.log(params.id);
  return {};
}
```

Async / await

我们喜欢现代 JavaScript，而且我们知道数据读取大多是异步的。这就是为什么 Nest 支持异步函数，并且与他们一起工作得非常好。

!> 了解更多关于 `Async / await` 请点击这里！

每个异步函数都必须返回 Promise。这意味着您可以返回延迟值，而 Nest 将自行解析它。让我们看看下面的例子：

> *cats.controller.ts*

```
@Get()
async findAll(): Promise<any[]> {
  return [];
}
```

这是完全有效的。

此外，Nest 路由处理程序更强大。它可以返回一个 Rxjs observable 的流，使得在简单的 web 应用程序和 微服务 之间的迁移更加容易。

> *cats.controller.ts*

```
@Get()
findAll(): Observable<any[]> {
  return of([]);
}
```

没有推荐的方法。 你可以使用任何你想要的。

POST 处理程序

奇怪的是，这个 POST 路由处理程序不接受任何客户端参数。我们至少应该在 `@Body()` 这里添加参数来解决这个问题。

首先，我们需要确定 DTO（数据传输对象）架构。DTO 是一个定义如何通过网络发送数据的对象。我们可以使用 TypeScript 接口或简单的类来完成。令人惊讶的是，我们建议在这里使用类。为什么？这些类是 JavaScript ES6 标准的一部分，所以它们只是简单的函数。

另一方面，TypeScript 接口在编译过程中被删除，Nest 不能引用它们。这一点很重要，因为管道等特性能够在访问变量的元类型时提供更多的可能性。

我们来创建 `CreateCatDto`：

> *create-cat.dto.ts*

```
export class CreateCatDto {
  readonly name: string;
  readonly age: number;
  readonly breed: string;
}
```

它只有三个基本属性。 所有这些都被标记为只读，因为我们应该尽可能使我们的功能尽可能不被污染。

现在我们可以使用 `CatsController` 中的模式：

> *cats.controller.ts*

```
@Post()
async create(@Body() createCatDto: CreateCatDto) {}
```

处理 errors

这里有一个关于处理异常的独立章节。

最后一步

控制器已经准备就绪，可以使用，但是 Nest 不知道 CatsController 是否存在，所以它不会创建这个类的一个实例。我们需要告诉它。

控制器总是属于模块，这就是为什么我们 controllers 在 `@Module()` 装饰器中保存数组。 由于除了根 ApplicationModule，我们没有其他模块，现在就让我们使用它：

> *app.module.ts*

```
import { Module } from '@nestjs/common';
import { CatsController } from './cats/cats.controller';

@Module({
  controllers: [CatsController],
})
export class ApplicationModule {}
```

!> 我们将元数据附加到 module 类，所以现在 Nest 可以很容易地反映出哪些控制器必须被安装。

特定库 方式

到目前为止，我们已经讨论了 Nest 操作响应的标准方式。操作响应的第二种方法是使用特定于库的响应对象。为了注入响应对象，我们需要使用 `@Res()` 装饰器。为了对比差异，我们重写 CatsController：

```
import { Controller, Get, Post, Res, Body, HttpStatus } from
'@nestjs/common';
import { CreateCatDto } from './dto/create-cat.dto';

@Controller('cats')
export class CatsController {
  @Post()
  create(@Res() res, @Body() createCatDto: CreateCatDto) {
    res.status(HttpStatus.CREATED).send();
  }

```

```
11.    @Get()
12.    findAll(@Res() res) {
13.       res.status(HttpStatus.OK).json([]);
14.    }
15.  }
```

从我的角度来看，这种方式是非常不清晰。 我当然更喜欢第一种方法，但为了使 Nest 向下兼容以前的版本，这种方法仍然可用。 而且，响应对象提供了更大的灵活性 - 您可以完全控制响应。

提供者

提供者

几乎所有的东西都可以被认为是提供者 - service, repository, factory, helper 等等。他们都可以注入依赖关系 `constructor`，也就是说，他们可以创建各种关系。但事实上，提供者不过是一个用 `@Injectable()` 装饰器注解的简单类。

控制器应处理 HTTP 请求并将更复杂的任务委托给服务。提供者是纯粹的 JavaScript 类，其 `@Injectable()` 上有装饰器。

?> 由于 Nest 可以以更多的面向对象方式设计和组织依赖性，因此我们强烈建议遵循 SOLID 原则。

我们来创建一个简单的 CatsService 提供者:

> *cats.service.ts*

```
1.  import { Injectable } from '@nestjs/common';
2.  import { Cat } from './interfaces/cat.interface';
3.
4.  @Injectable()
5.  export class CatsService {
6.    private readonly cats: Cat[] = [];
7.
8.    create(cat: Cat) {
```

```
    this.cats.push(cat);
  }

  findAll(): Cat[] {
    return this.cats;
  }
}
```

这是一个 CatsService 基本类，有一个属性和两个方法。唯一的新特点是它使用 `@Injectable()` 装饰器。该 `@Injectable()` 附加的元数据，从而 Nest 知道这个类是一个 Nest 提供者。

?> Cat 上面有一个界面。我们没有提到它，因为这个模式与 CreateCatDto 我们在前一章中创建的类完全相同。

由于我们已经完成了服务类，所以我们在以下内容中使用它 CatsController ：

> *cats.controller.ts*

```
import { Controller, Get, Post, Body } from '@nestjs/common';
import { CreateCatDto } from './dto/create-cat.dto';
import { CatsService } from './cats.service';
import { Cat } from './interfaces/cat.interface';

@Controller('cats')
export class CatsController {
  constructor(private readonly catsService: CatsService) {}

  @Post()
  async create(@Body() createCatDto: CreateCatDto) {
    this.catsService.create(createCatDto);
  }

  @Get()
  async findAll(): Promise<Cat[]> {
    return this.catsService.findAll();
```

```
  }
}
```

在 CatsService 通过类构造函数注入。不要害怕 private readonly 缩短的语法。这意味着我们已经在同一位置创建并初始化了 catsService 成员。

依赖注入

Nest 是建立在强大的设计模式，通常称为依赖注入。我们建议在官方的 Angular文档中阅读关于这个概念的伟大文章。

在 Nest 中，由于 TypeScript 的缘故，管理依赖关系非常简单，因为它们只是按类型解决，然后注入控制器的构造函数中：

```
constructor(private readonly catsService: CatsService) {}
```

定制提供者

Nest 用来解决提供者之间关系的控制反转要比上面描述的要强大得多。 `@Injectable()` 装饰器只是冰山一角，不需要严格定义提供商。相反，您可以使用普通值，类，异步或同步工厂。看看这里找到更多的例子。

最后一步

> *app.module.ts*

```
import { Module } from '@nestjs/common';
import { CatsController } from './cats/cats.controller';
import { CatsService } from './cats/cats.service';

@Module({
```

```
  controllers: [CatsController],
  providers: [CatsService],
})
export class ApplicationModule {}
```

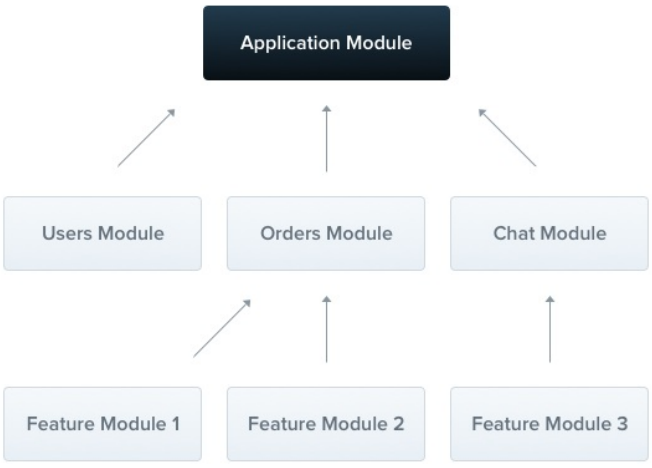
得益于此，Nest 将能够解决 CatsController 类的依赖关系。这就是我们目前的目录结构：

```
src
├── cats
│   ├──dto
│   │   └──create-cat.dto.ts
│   ├── interfaces
│   │       └──cat.interface.ts
│   ├──cats.service.ts
│   └──cats.controller.ts
├──app.module.ts
└──main.ts
```

模块

模块

模块是具有 `@Module()` 装饰器的类。 `@Module()` 装饰器提供了元数据，Nest 用它来组织应用程序结构。



每个 Nest 应用程序至少有一个模块，即根模块。根模块是 Nest 开始安排应用程序树的地方。事实上，根模块可能是应用程序中唯一的模块，特别是当应用程序很小时，但是对于大型程序来说这是没有意义的。在大多数情况下，您将拥有多个模块，每个模块都有一组紧密相关

的功能。

所述 `@Module()` 装饰采用单个对象，其属性描述该模块：

providers	由 Nest 注入器实例化的提供者，并且可以至少在整个模块中共享
controllers	必须创建的一组控制器
imports	导入模块所需的导入模块列表
exports	此模块提供的提供者的子集，并应在其他模块中使用

默认情况下，模块封装提供者。这意味着无法插入既不是当前模块的直接部分，也不能从导入的模块导出提供者。

CatsModule

`CatsController` 和 `CatsService` 属于同一个应用程序域。应该将它们移动到功能模块 `CatsModule`。

> *cats/cats.module.ts*

```
import { Module } from '@nestjs/common';
import { CatsController } from './cats.controller';
import { CatsService } from './cats.service';

@Module({
  controllers: [CatsController],
  providers: [CatsService],
})
export class CatsModule {}
```

我已经创建了 `cats.module.ts` 文件，并把与这个模块相关的所有东西都移到了 `cats` 目录下。我们需要做的最后一件事是将这个模块导入根模块 `(ApplicationModule)`。

> *app.module.ts*

```
import { Module } from '@nestjs/common';
import { CatsModule } from './cats/cats.module';

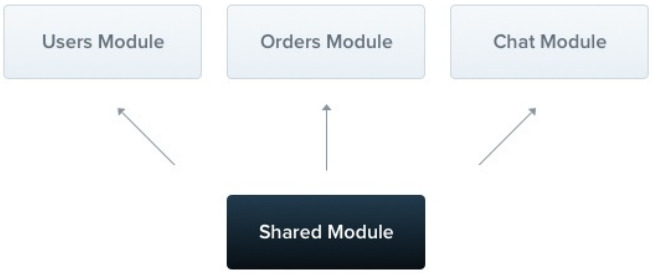
@Module({
  imports: [CatsModule],
})
export class ApplicationModule {
```

现在 Nest 知道除了 `ApplicationModule` 之外，注册 `CatsModule` 也是非常重要的。 这就是我们现在的目录结构：

```
src
├──cats
│    ├──dto
│    │    └──create-cat.dto.ts
│    ├──interfaces
│    │     └──cat.interface.ts
│    ├──cats.service.ts
│    ├──cats.controller.ts
│    └──cats.module.ts
├──app.module.ts
└──main.ts
```

共享模块

在 Nest 中，默认情况下，模块是单例，因此您可以毫不费力地在 2..* 模块之间共享同一个组件实例。



实际上，每个模块都是一个共享模块。 一旦创建就被每个模块重复使用。 假设我们将在几个模块之间共享 `CatsService` 实例。 我们需要把 `CatsService` 放到 `exports` 数组中，如下所示：

> *cats.module.ts*

```
import { Module } from '@nestjs/common';
import { CatsController } from './cats.controller';
import { CatsService } from './cats.service';

@Module({
  controllers: [CatsController],
  providers: [CatsService],
  exports: [CatsService]
})
export class CatsModule {}
```

现在，每个导入 `CatsModule` 的模块都可以访问该模块 ，并将 `CatsService` 与导入该模块的所有模块共享相同的实例。

模块重新导出

模块可以导出他们的提供者。 而且，他们可以再导出自己导入的模块。

```
@Module({
  imports: [CommonModule],
  exports: [CommonModule],
})
export class CoreModule {}
```

依赖注入

模块类也可以注入提供者（例如，用于配置目的）：

> *cats.module.ts*

```
import { Module } from '@nestjs/common';
import { CatsController } from './cats.controller';
import { CatsService } from './cats.service';

@Module({
  controllers: [CatsController],
  providers: [CatsService],
})
export class CatsModule {
  constructor(private readonly catsService: CatsService) {}
}
```

但是，由于循环依赖性，模块类不能由提供者注入。

全局模块

如果你不得不在任何地方导入相同的模块，那可能很烦人。在 Angular 中，提供者是在全局范围内注册的。一旦定义，他们到处可用。另一方面，Nest 封装了模块范围内的组件。您无法在其他地方使用模块提供者而不导入他们。但是有时候，你可能只想提供一组随时可用的东西 - 例如：helper，数据库连接等等。这就是为什么你能够使模块成为全局模块。

```
import { Module, Global } from '@nestjs/common';
import { CatsController } from './cats.controller';
import { CatsService } from './cats.service';

@Global()
@Module({
  controllers: [CatsController],
  providers: [CatsService],
  exports: [CatsService]
})
export class CatsModule {}
```

`@Global` 装饰器使模块成为全局作用域。 全局模块应该只注册一次，最好由根或核心模块注册。 之后， `CatsService` 组件将无处不在，但 `CatsModule` 不会被导入。

!> 使一切全局化并不是一个好的解决方案。 全局模块在这里减少了必要的样板数量。 `imports` 数组仍然是使模块 API 透明的最佳方式。

动态模块

Nest 模块系统带有一个称为动态模块的功能。 它使您能够毫不费力地创建可定制的模块。 让我们来看看 `DatabaseModule` :

```
import { Module, DynamicModule } from '@nestjs/common';
import { createDatabaseProviders } from './database.providers';
import { Connection } from './connection.provider';

@Module({
  providers: [Connection],
})
export class DatabaseModule {
  static forRoot(entities = [], options?): DynamicModule {
    const providers = createDatabaseProviders(options, entities);
```

```
    return {
      module: DatabaseModule,
      providers: providers,
      exports: providers,
    };
  }
}
```

此模块默认定义了 `Connection` 提供者，但另外 - 根据传递的 `options` 和 `entities` - 创建一个提供者集合，例如存储库。 实际上，动态模块扩展了模块元数据。 当您需要动态注册组件时，这个实质特性非常有用。 然后你可以通过以下方式导入 `DatabaseModule`：

```
import { Module } from '@nestjs/common';
import { DatabaseModule } from './database/database.module';
import { User } from './users/entities/user.entity';

@Module({
  imports: [
    DatabaseModule.forRoot([User]),
  ],
})
export class ApplicationModule {}
```

为了导出动态模块，可以省略函数调用部分：

```
import { Module } from '@nestjs/common';
import { DatabaseModule } from './database/database.module';
import { User } from './users/entities/user.entity';

@Module({
  imports: [
    DatabaseModule.forRoot([User]),
  ],
  exports: [DatabaseModule]
```

```
10. })
11. export class ApplicationModule {}
```

中间件

中间件

中间件是一个在路由处理器之前被调用的函数。 中间件功能可以访问请求和响应对象，以及应用程序请求响应周期中的下一个中间件功能。下一个中间件函数通常由名为 `next` 的变量表示。

Nest 中间件实际上等价于 express 中间件。 从 Express 官方文档复制的中间件功能有很多：

> 中间件函数可以执行以下任务：
>
> - 执行任何代码。
> - 对请求和响应对象进行更改。
> - 结束请求-响应周期。
> - 调用堆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。

Nest 中间件可以是一个函数，也可以是一个带有 `@Injectable()` 装饰器的类。 这个类应该实现 `NestMiddleware` 接口。 我们来创建一个例子， `LoggerMiddleware` 类：

> *logger.middleware.ts*

```
1.  import { Injectable, NestMiddleware, MiddlewareFunction } from
    '@nestjs/common';
2.
3.  @Injectable()
4.  export class LoggerMiddleware implements NestMiddleware {
5.    resolve(...args: any[]): MiddlewareFunction {
6.      return (req, res, next) => {
7.        console.log('Request...');
8.        next();
9.      };
10.   }
11. }
```

该 `resolve()` 方法必须返回常规的特定库的中间件 `(req, res, next) => any`

依赖注入

说到中间件， 也不例外。与提供者和控制器相同， 它们能够注入属于统一模块的依赖项（通过constructor）。

中间件放在哪里

中间件不能在 `@Module()` 装饰器中列出。我们必须使用 `configure()` 模块类的方法来设置它们。包含中间件的模块必须实现 `NestModule` 接口。让我们设置 `LoggerMiddleware` 在 `ApplicationModule` 关卡上。

> *app.module.ts*

```
import { Module, NestModule, MiddlewareConsumer } from
'@nestjs/common';
import { LoggerMiddleware } from
'./common/middlewares/logger.middleware';
import { CatsModule } from './cats/cats.module';

@Module({
  imports: [CatsModule],
})
export class ApplicationModule implements NestModule {
  configure(consumer: MiddlewareConsumer) {
    consumer
      .apply(LoggerMiddleware)
      .forRoutes('/cats');
  }
}
```

在上面的例子中，我们已经设置了 `LoggerMiddleware` 的 `/cats` 的路由处理程序 CatsController 。MiddlewareConsumer 是一个帮助类。它提供了几种使用中间件的方法。他们都可以简单地链接。让我们来看看这些方法。

在 `forRoutes()` 可采取单个对象，多个对象，控制器类和甚至多个控制器类。在大多数情况下，你可能只是通过控制器，并用逗号分隔。以下是单个控制器的示例

> *app.module.ts*

```
import { Module, NestModule, MiddlewareConsumer } from
'@nestjs/common';
import { LoggerMiddleware } from
'./common/middlewares/logger.middleware';
import { CatsModule } from './cats/cats.module';

@Module({
  imports: [CatsModule],
```

```
})
export class ApplicationModule implements NestModule {
  configure(consumer: MiddlewareConsumer): void {
    consumer
      .apply(LoggerMiddleware)
      .forRoutes(CatsController);
  }
}
```

?> 该 `apply()` 方法可以采用单个中间件或一组中间件。

将参数传递给中间件

有时中间件的行为取决于自定义值，例如用户角色数组，选项对象等。我们可以将其他参数传递给 `resolve()` 来使用 `with()` 方法。看下面的例子：

> *app.moudle.ts*

```
import { Module, NestModule, MiddlewareConsumer } from
'@nestjs/common';
import { LoggerMiddleware } from
'./common/middlewares/logger.middleware';
import { CatsModule } from './cats/cats.module';
import { CatsController } from './cats/cats.controller';

@Module({
  imports: [CatsModule],
})
export class ApplicationModule implements NestModule {
  configure(consumer: MiddlewareConsumer): void {
    consumer
      .apply(LoggerMiddleware)
      .with('ApplicationModule')
      .forRoutes(CatsController);
  }
}
```

我们已经通过了一个自定义字符串 - `ApplicationModule` 的 `with()` 方法。现在我们必须调整 `LoggerMiddleware` 的 `resolve()` 方法。

> *logger.middleware.ts*

```
import { Injectable, NestMiddleware, MiddlewareFunction } from '@nestjs/common';

@Injectable()
export class LoggerMiddleware implements NestMiddleware {
  resolve(name: string): MiddlewareFunction {
    return (req, res, next) => {
      console.log(`[${name}] Request...`); // [ApplicationModule] Request...
      next();
    };
  }
}
```

该 `name` 的属性值将是 `ApplicationModule` 。

异步中间件

从 `resolve()` 方法中返回异步函数没有禁忌。所以， `resolve()` 方法也可以写成 `async` 的。这种模式被称为 *延迟中间件* 。

> *logger.middleware.ts*

```
import { Injectable, NestMiddleware, MiddlewareFunction } from '@nestjs/common';

@Injectable()
export class LoggerMiddleware implements NestMiddleware {
  async resolve(name: string): Promise<MiddlewareFunction> {
```

```
    await someAsyncJob();

    return async (req, res, next) => {
      await someAsyncJob();
      console.log(`[${name}] Request...`); // [ApplicationModule] Request...
      next();
    };
  }
}
```

函数式中间件

`LoggerMiddleware` 很短。它没有成员，没有额外的方法，没有依赖关系。为什么我们不能只使用一个简单的函数？这是一个很好的问题，因为事实上 - 我们可以做到。这种类型的中间件称为函数式中间件。让我们把 `logger` 转换成函数。

> *logger.middleware.ts*

```
export function logger(req, res, next) {
  console.log(`Request...`);
  next();
};
```

现在在 `ApplicationModule` 中使用它。

> *app.module.ts*

```
import { Module, NestModule, MiddlewareConsumer } from '@nestjs/common';
import { logger } from './common/middlewares/logger.middleware';
import { CatsModule } from './cats/cats.module';
import { CatsController } from './cats/cats.controller';

@Module({
```

```
  imports: [CatsModule],
})
export class ApplicationModule implements NestModule {
  configure(consumer: MiddlewareConsumer) {
    consumer
      .apply(logger)
      .forRoutes(CatsController);
  }
}
```

?> 当您的中间件没有任何依赖关系时，我们可以考虑使用函数式中间件。

多个中间件

如前所述，为了绑定顺序执行的多个中间件，我们可以在 `apply()` 方法内用逗号分隔它们。

```
export class ApplicationModule implements NestModule {
  configure(consumer: MiddlewareConsumer) {
    consumer
      .apply(cors(), helmet(), logger)
      .forRoutes(CatsController);
  }
}
```

全局中间件

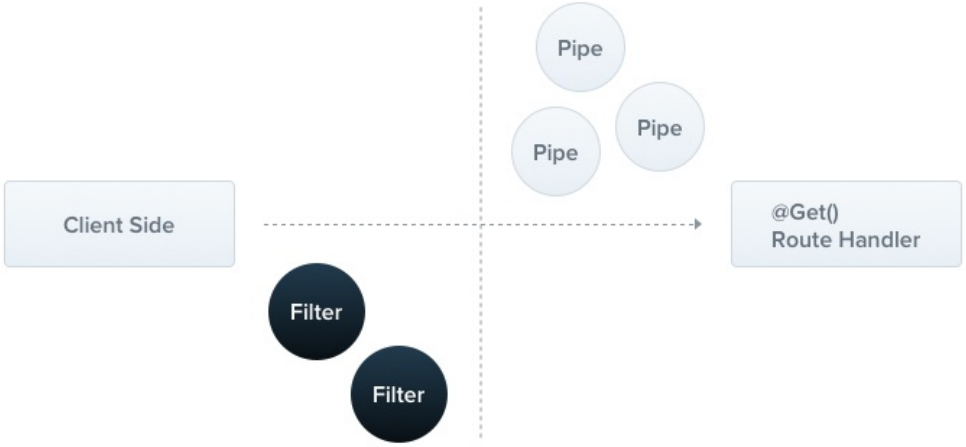
为了一次将中间件绑定到每个注册路由，我们可以利用实例 `use()` 提供的方法 `INestApplication`：

```
const app = await NestFactory.create(ApplicationModule);
app.use(logger);
await app.listen(3000);
```

异常过滤器

异常过滤器

内置的 异常层负责处理整个应用程序中的所有抛出的异常。当捕获到未处理的异常时，最终用户将收到适当的用户友好响应。



每个异常都由全局异常筛选器处理， 当无法识别时 （既不是 `HttpException` 也不是继承的类 `HttpException` ），用户将收到以下 JSON 响应：

```
{
    "statusCode": 500,
    "message": "Internal server error"
}
```

HttpException

包 `HttpException` 内 有一个内置的类 `@nestjs/common` 。核心异常处理程序与此类会很好地工作。当你抛出 `HttpException` 对象时，它将被处理程序捕获并转换为相关的 JSON 响应。

在 `CatsController` ，我们有一个 `create()` 方法（ `POST` 路由）。让我们假设这个路由处理器由于某种原因会抛出一个异常。我们要强制编译它：

> *cats.controller.ts*

```
@Post()
async create(@Body() createCatDto: CreateCatDto) {
  throw new HttpException('Forbidden', HttpStatus.FORBIDDEN);
}
```

!> 我在这里用了 `HttpStatus` 。这是从 `@nestjs/common` 包中导入的助手枚举。

现在当客户端调用这个端点时，响应如下所示：

```
{
    "statusCode": 403,
    "message": "Forbidden"
}
```

`HttpException` 构造函数采用 `string | object` 作为第一个参数。如果您要传递「对象」而不是「字符串」， 则将完全覆盖响应体。

> *cats.controller.ts*

```
@Post()
async create(@Body() createCatDto: CreateCatDto) {
  throw new HttpException({
    status: HttpStatus.FORBIDDEN,
    error: 'This is a custom message',
  }, 403);
}
```

这就是响应的样子：

```
{
    "status": 403,
    "error": "This is a custom message"
}
```

异常层次（Exceptions Hierarchy）

好的做法是创建自己的异常层次结构。这意味着每个 HTTP 异常都应从基 `HttpException` 类继承。因此，Nest 将识别您的所有异常，并将充分注意错误响应。

> *forbidden.exception.ts*

```
export class ForbiddenException extends HttpException {
  constructor() {
    super('Forbidden', HttpStatus.FORBIDDEN);
  }
}
```

既然 `ForbiddenException` 扩展了基础 `HttpException`，它将和核心异常处理程序一起工作，所以我们可以在这个 `create()` 方法中使用这个类。

> *cats.controller.ts*

```
@Post()
async create(@Body() createCatDto: CreateCatDto) {
  throw new ForbiddenException();
}
```

HTTP exceptions

为了减少样板代码，Nest 提供了一系列扩展核心的可用异常 `HttpException` 。所有这些都可以在 `@nestjs/common` 包中找到：

- BadRequestException
- UnauthorizedException
- NotFoundException
- ForbiddenException
- NotAcceptableException
- RequestTimeoutException
- ConflictException
- GoneException
- PayloadTooLargeException
- UnsupportedMediaTypeException
- UnprocessableException
- InternalServerErrorException
- NotImplementedException
- BadGatewayException
- ServiceUnavailableException当
- GatewayTimeoutException

异常过滤器（Exception Filters）

基本异常处理程序很好，但有时您可能想要完全控制异常层，例如添加一些日志记录或使用不同的 JSON 模式。我们喜欢通用的解决方案，使您的生活更轻松，这就是为什么称为异常过滤器的功能被创建 的原因

我们要创建过滤器，它的职责是捕获 `HttpException` 类实例异常，并为它们设置自定义响应逻辑。

> *http-exception.filter.ts*

```
1.  import { ExceptionFilter, Catch, ArgumentsHost } from
    '@nestjs/common';
2.  import { HttpException } from '@nestjs/common';
3.
4.  @Catch(HttpException)
5.  export class HttpExceptionFilter implements ExceptionFilter {
6.    catch(exception: HttpException, host: ArgumentsHost) {
7.      const ctx = host.switchToHttp();
8.      const response = ctx.getResponse();
9.      const request = ctx.getRequest();
10.
11.     response
12.       .status(status)
13.       .json({
14.         statusCode: exception.getStatus(),
15.         timestamp: new Date().toISOString(),
16.         path: request.url,
17.       });
18.   }
19. }
```

?> 每个异常过滤器都应该实现这个 `ExceptionFilter` 接口。它强制你提供具有正确特征的 `catch()` 的方法。

所述 `@Catch()` 装饰结合所需的元数据到异常过滤器。它告诉 Nest，这个过滤器正在寻找 `HttpException` 。 `@Catch()` 的参数是

无限个数的，所以你可以为多个类型的异常设置该过滤器，只需要用逗号将它们分开。

该 `exception` 属性是一个当前处理的异常，同时 `host` 也是一个 `ArgumentsHost` 对象。ArgumentsHost 是传递给原始处理程序的参数的一个包装 ，它根据应用程序的类型在底层包含不同的参数数组。

```
export interface ArgumentsHost {
  getArgs<T extends Array<any> = any[]>(): T;
  getArgByIndex<T = any>(index: number): T;
  switchToRpc(): RpcArgumentsHost;
  switchToHttp(): HttpArgumentsHost;
  switchToWs(): WsArgumentsHost;
}
```

在 `ArgumentsHost` 有一组有用的方法，有助于从底层数组挑选正确的参数为我们提供。换句话说， ArgumentsHost 除了一系列参数之外别无他法 。例如，当过滤器在 HTTP 应用程序上下文中使用时，ArgumentsHost 将包含 [request, response] 数组。但是，当前上下文是一个 Web 套接字应用程序时，该数组将等于 [client, data]。此设计决策使您能够访问任何将最终传递给相应处理程序的参数。

让我们配合 HttpExceptionFilter 使用 `create()` 方法。

> *cats.controller.ts*

```
@Post()
@UseFilters(new HttpExceptionFilter())
async create(@Body() createCatDto: CreateCatDto) {
  throw new ForbiddenException();
}
```

?> `@UseFilters()` 装饰器从 `@nestjs/common` 包导入。

我们在 `@UseFilters()` 这里使用了装饰器。与 `@Catch()` 相同，它需要无限的参数。

这个实例 `HttpExceptionFilter` 已经被直接创建了。另一种可用的方式是传递类（不是实例），让框架实例化责任并启用依赖注入。

> *cats.controller.ts*

```
@Post()
@UseFilters(HttpExceptionFilter)
async create(@Body() createCatDto: CreateCatDto) {
  throw new ForbiddenException();
}
```

?> 提示如果可能，最好使用类而不是实例。它可以减少内存使用量，因为 Nest 可以轻松地在整个应用程序中重复使用同一类的实例。

在上面的例子中，`HttpExceptionFilter` 仅适用于单个 `create()` 路由处理程序，但它不是唯一的方法。实际上，异常过滤器可以是方法范围的，控制器范围的，也可以是全局范围的。

> *cats.controller.ts*

```
@UseFilters(new HttpExceptionFilter())
export class CatsController {}
```

此结构为 `CatsController` 中的每个路由处理程序设置 `HttpExceptionFilter`。它是控制器范围的异常过滤器的示例。最后一个可用范围是全局范围的异常过滤器。

> *main.ts*

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

```
}
bootstrap();
```

!> 该 `useGlobalFilters()` 方法不会为网关和微服务设置过滤器。

全局过滤器用于整个应用程序，每个控制器，每个路由处理程序。就依赖注入而言，从任何模块外部注册的全局过滤器（如上面的示例中那样）不能注入依赖关系，因为它们不属于任何模块。为了解决这个问题，您可以使用以下构造直接从任何模块设置过滤器 ：

> *app.module.ts*

```
import { Module } from '@nestjs/common';
import { APP_FILTER } from '@nestjs/core';

@Module({
  providers: [
    {
      provide: APP_FILTER,
      useClass: HttpExceptionFilter,
    },
  ],
})
export class ApplicationModule {}
```

?> 提示另一种选择是使用执行上下文功能。另外，useClass 并不是处理自定义提供商注册的唯一方法。在这里了解更多。

抓住一切

为了处理每个发生的异常（无论异常类型如何），可以将括号留空（ `@Catch()` ）：

> *any-exception.filter.ts*
> ```typescript
> import { ExceptionFilter, Catch, ArgumentsHost } from
> ```

```
  '@nestjs/common';

@Catch()
export class AnyExceptionFilter implements
ExceptionFilter {
catch(exception: any, host: ArgumentsHost) {
const ctx = host.switchToHttp();
const response = ctx.getResponse();
const request = ctx.getRequest();
```

```
1.  response
2.    .status(status)
3.    .json({
4.      statusCode: exception.getStatus(),
5.      timestamp: new Date().toISOString(),
6.      path: request.url,
7.    });
```

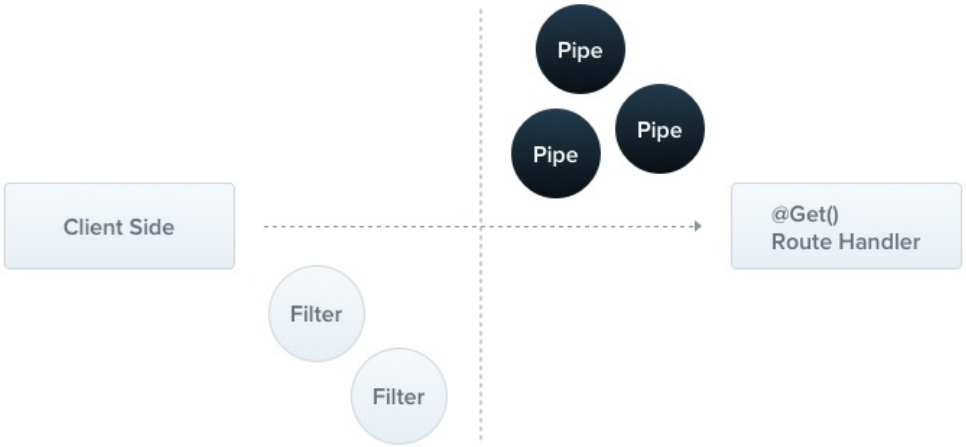
```
}
}
```

在上面的例子中，过滤器会捕获已经抛出的每个异常，而不会将自身限制为一组特定的类。

管道

管道

管道是具有 `@Pipe()` 装饰器的类。管道应实现 `PipeTransform` 接口。



管道将输入数据转换为所需的输出。 另外，它可以处理验证，因为当数据不正确时可能会抛出异常。

?> 提示管道在异常区域内运行。 这意味着当抛出异常时，它们由核心异常处理程序处理，异常过滤器应用于当前上下文。

内置管道

`Nest` 自带两个可用的管道，即 `ValidationPipe` 和 `ParseIntPipe`。 他们从 `@nestjs/common` 包中导出。 为了更好地理解它们是如何工作的，我们将从头开始构建它们。

它是什么样子的？

我们从 `ValidationPipe` 开始。 现在它只取得一个类型的值，并且会返回同样类型的值。

> *validation.pipe.ts*

```
1. import { PipeTransform, Pipe, ArgumentMetadata } from
   '@nestjs/common';
2.
3. @Pipe()
4. export class ValidationPipe implements PipeTransform<any> {
5.   transform(value: any, metadata: ArgumentMetadata) {
6.     return value;
7.   }
8. }
9. }
```

!> `ValidationPipe` 仅适用于 `TypeScript`。 如果你使用普通的 `JavaScript`，我建议使用 `Joi` 库。

每个管道必须提供 `transform()` 方法。 这个方法有两个参数：

- value
- metadata

`value` 是当前处理的参数，而 `metadata` 是其元数据。 元数据对象包含一些属性：

```
export interface ArgumentMetadata {
    type: 'body' | 'query' | 'param' | 'custom';
    metatype?: new (...args) => any;
    data?: string;
}
```

这里有一些属性描述参数：

参数	描述
type	告诉我们该属性是一个 `body @Body()`，`query @Query()`，`param @Param()` 还是自定义参数 在这里阅读更多。
metatype	属性的元类型，例如 `String`。 如果在函数签名中省略类型声明，则不确定。
data	传递给装饰器的字符串，例如 `@Body('string')`。 如果您将括号留空，则不确定。

!> `TypeScript` 接口在编译期间消失，所以如果你使用接口而不是类，那么元类型的值将是一个 `Object`。

它是如何工作的？

我们来关注一下 `CatsController` 的 `create()` 方法：

> *cats.controler.ts*

```
@Post()
async create(@Body() createCatDto: CreateCatDto) {
  this.catsService.create(createCatDto);
}
```

一个 `CreateCatDto` 参数

> *create-cat.dto.ts*

```
export class CreateCatDto {
```

```
  readonly name: string;
  readonly age: number;
  readonly breed: string;
}
```

这个对象总是要正确的，所以我们必须验证这三个成员。我们可以在路由处理程序方法中做到这一点，但是我们会打破单个责任规则（SRP）。第二个想法是创建一个验证器类并在那里委托任务，但是每次在方法开始的时候我们都必须使用这个验证器。那么验证中间件呢？这是一个好主意，但不可能创建一个通用的中间件，可以在整个应用程序中使用。

这是第一个用例，当你应该考虑使用管道。

类验证器

本节仅适用于 `TypeScript` 。

`Nest` 与类验证器一起工作良好，这个惊人的库允许您使用基于装饰器的验证。 由于我们可以访问处理过的属性的元类型，所以基于装饰器的验证对于管道功能是非常强大的。 让我们添加一些装饰到 `CreateCatDto` 。

> *create-cat.dto.ts*

```
import { IsString, IsInt } from 'class-validator';

export class CreateCatDto {
  @IsString()
  readonly name: string;

  @IsInt()
  readonly age: number;

  @IsString()
```

```
  readonly breed: string;
}
```

现在是完成 `ValidationPipe` 类的时候了。

> *validation.pipe.ts*

```
import { PipeTransform, Pipe, ArgumentMetadata, BadRequestException } from '@nestjs/common';
import { validate } from 'class-validator';
import { plainToClass } from 'class-transformer';

@Pipe()
export class ValidationPipe implements PipeTransform<any> {
    async transform(value, metadata: ArgumentMetadata) {
      const { metatype } = metadata;
      if (!metatype || !this.toValidate(metatype)) {
          return value;
      }
      const object = plainToClass(metatype, value);
      const errors = await validate(object);
      if (errors.length > 0) {
          throw new BadRequestException('Validation failed');
      }
      return value;
    }

    private toValidate(metatype): boolean {
      const types = [String, Boolean, Number, Array, Object];
      return !types.find((type) => metatype === type);
    }
}
```

!> 我已经使用了类转换器库。它是由同一个作者作为类验证器库，所以他们一起玩。

我们来看看这个代码。首先，请注意 `transform()` 函数是异步的。这

是可能的，因为Nest支持同步和异步管道。另外，还有一个辅助函数 `toValidate()` 。它负责从验证过程中排除原生 `JavaScript` 类型。最后一个重要的是我们必须返回相同的价值。这个管道是一个验证特定的管道，所以我们需要返回完全相同的属性以避免重写。

最后一步是设置 `ValidationPipe` 。管道，与异常过滤器相同，它们可以是方法范围的，控制器范围的和全局范围的。另外，管道可以是参数范围的。我们可以直接将管道实例绑定到路由参数装饰器，例如 `@Body()` 。让我们来看看下面的例子：

> *cats.controler.ts*

```
1. @Post()
2. async create(@Body(new ValidationPipe()) createCatDto:
   CreateCatDto) {
3.   this.catsService.create(createCatDto);
4. }
```

当验证逻辑仅涉及一个指定的参数时，参数范围的管道是有用的。 要在方法级别设置管道，您需要使用 `UsePipes()` 装饰器。

> *cats.controler.ts*

```
1. @Post()
2. @UsePipes(new ValidationPipe())
3. async create(@Body() createCatDto: CreateCatDto) {
4.   this.catsService.create(createCatDto);
5. }
```

!> `@UsePipes()` 修饰器是从 `@nestjs/common` 包中导入的。

由于 `ValidationPipe` 被创建为尽可能通用，所以我们将把它设置为一个全局作用域的管道，用于整个应用程序中的每个路由处理器。

> *main.ts*

```
async function bootstrap() {
  const app = await NestFactory.create(ApplicationModule);
  app.useGlobalPipes(new ValidationPipe());
  await app.listen(3000);
}
bootstrap();
```

!> `useGlobalPipes()` 方法不会为网关和微服务设置管道。

变压器管道

验证不是唯一的用例。 在本章的开始部分，我已经提到管道也可以将输入数据转换为所需的输出。 这是真的，因为从 `transform` 函数返回的值完全覆盖了参数的前一个值。 有时从客户端传来的数据需要经过一些修改。 此外，有些部分可能会丢失，所以我们必须应用默认值。 变压器管道填补了客户端和请求处理程序的请求之间的空白。

> *parse-int.pipe.ts*

```
import { PipeTransform, Pipe, ArgumentMetadata, HttpStatus,
BadRequestException } from '@nestjs/common';

@Pipe()
export class ParseIntPipe implements PipeTransform<string> {
  async transform(value: string, metadata: ArgumentMetadata) {
    const val = parseInt(value, 10);
    if (isNaN(val)) {
      throw new BadRequestException('Validation failed');
    }
    return val;
  }
}
```

这是一个 `ParseIntPipe` ，它负责将一个字符串解析为一个整数值。现在让我们将管道绑定到选定的参数：

```
@Get(':id')
async findOne(@Param('id', new ParseIntPipe()) id) {
  return await this.catsService.findOne(id);
}
```

全局管道

全局管道不属于任何范围。 他们生活在模块之外，因此，他们不能注入依赖。 我们需要立即创建一个实例。 但有时，全局管道依赖于其他对象。 我们如何解决这个问题？

解决方案非常简单。 实际上，每个 `Nest` 应用程序实例都是一个创建的Nest上下文。 `Nest` 上下文是 `Nest` 容器的一个包装，它包含所有实例化的类。 我们可以直接使用应用程序对象从任何导入的模块中获取任何现有的实例。

假设我们在 `SharedModule` 中注册了一个 `ValidationPipe` 。 这个 `SharedModule` 被导入到根模块中。 我们可以使用以下语法选择 `ValidationPipe` 实例：

```
const app = await NestFactory.create(ApplicationModule);
const validationPipe = app
  .select(SharedModule)
  .get(ValidationPipe);

app.useGlobalPipes(validationPipe);
```

要获取 `ValidationPipe` 实例，我们必须使用2个方法，在下表中有详细描述：

参数	描述
`get()`	使得可以检索已处理模块中可用的组件或控制器的实例。
`select()`	允许您浏览模块树，例如，从所选模块中提取特定实例。

!> 默认情况下选择根模块。 要选择任何其他模块，您需要遍历整个模块堆栈（一步一步）。

看守器

看守器

看守器是一个使用 `@Guard()` 装饰器的类。 看守器应该使用 `CanActivate` 接口。

看守器有一个单独的责任。它们确定请求是否应该由路由处理程序处理。到目前为止， 访问限制逻辑大多在中间件内。这样很好， 因为诸如 token 验证或将req对象附加属性与特定路由没有强关联。

但中间件是非常笨的。它不知道调用 next() 函数后应该执行哪个处理程序。另一方面， 看守器可以访问 ExecutionContext 对象， 所以我们确切知道将要评估什么。

?> 看守器是在每个中间件之后执行的， 但在管道之前。

授权看守器

最好的用例之一就是认证逻辑，因为只有当调用者具有足够的权限（例如管理员角色）时才能使用特定的路由。我们有一个计划要创建的 AuthGuard 将依次提取和验证在请求标头中发送的 token。

> *auth.guard.ts*

```
1.  import { Injectable, CanActivate, ExecutionContext } from
    '@nestjs/common';
2.  import { Observable } from 'rxjs';
3.
4.  @Injectable()
5.  export class AuthGuard implements CanActivate {
6.    canActivate(
7.      context: ExecutionContext,
8.    ): boolean | Promise<boolean> | Observable<boolean> {
9.      const request = context.switchToHttp().getRequest();
10.     return validateRequest(request);
11.   }
12. }
```

不管 validateRequest() 函数背后的逻辑是什么，主要的一点是要展示如何简单地利用看守器。每个看守器都提供一个 canActivate() 功能。看守器可能通过（Promise 或 Observable）同步地或异步地返回它的布尔答复。返回的值控制 Nest 行为：

- 如果返回 true，将处理用户调用。
- 如果返回 false，则 Nest 将忽略当前处理的请求。

canActivate() 函数采用单参数 ExecutionContext 实例。ExecutionContext 从 ArgumentsHost 继承（这里首先提到）。ArgumentsHost 是围绕已传递给原始处理程序的参数的包装，它包含基于应用程序类型的引擎下的不同参数数组。

```
export interface ArgumentsHost {
  getArgs<T extends Array<any> = any[]>(): T;
  getArgByIndex<T = any>(index: number): T;
  switchToRpc(): RpcArgumentsHost;
  switchToHttp(): HttpArgumentsHost;
  switchToWs(): WsArgumentsHost;
}
```

ArgumentsHost 为我们提供了一套有用的方法，帮助从基础数组中选取正确的参数。换言之，ArgumentsHost 只是一个参数数组而已。例如，当在 HTTP 应用程序上下文中使用该保护程序时，ArgumentsHost 将在内部包含 [request, response] 数组。但是，当当前上下文是 web 套接字应用程序时，此数组将等于 [client, data]]。通过此设计，您可以访问最终传递给相应处理程序的任何参数。

ExecutionContext 提供多一点。它扩展了 ArgumentsHost，而且还提供了有关当前执行过程的更多细节。

getHandler() 返回对当前处理的处理程序的引用，而 getClass() 返回此特定处理程序所属的控制器类的类型。换句话说，如果用户指向在 CatsController 中定义和注册的 `create()` 方法，则 getHandler() 将返回对 `create()` 方法和 `getClass()` 的引用，在这种情况下，将只返回一个 CatsController 类型（不是实例）。

基于角色的认证

一个更详细的例子是一个 RolesGuard 。这个看守器只允许具有特定角色的用户访问。我们要从一个基本的看守器模板开始：

> *roles.guard.ts*

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

@Injectable()
export class RolesGuard implements CanActivate {
  canActivate(
    context: ExecutionContext,
  ): boolean | Promise<boolean> | Observable<boolean> {
    return true;
  }
}
```

看守器可以是控制器范围的，方法范围的和全局范围的。为了建立看守器，我们使用 `@UseGuards()` 装饰器。这个装饰器可以带来无数的参数。也就是说，你可以传递几个看守器并用逗号分隔它们。

> *cats.controller.ts*

```
@Controller('cats')
@UseGuards(RolesGuard)
export class CatsController {}
```

!> `@UseGuards()` 装饰器是从 `@nestjs/common` 包中导入的。

我们已经通过了 RolesGuard 类型而不是实例，使框架成为实例化责任并启用依赖项注入。另一种可用的方法是传递立即创建的实例：

> *cats.controller.ts*

```
@Controller('cats')
@UseGuards(new RolesGuard())
export class CatsController {}
```

上面的构造将守卫附加到此控制器声明的每个处理程序。如果我们决定只限制其中一个， 我们只需要设置在方法级别的看守器。为了绑定全局

看守器，我们使用 Nest 应用程序实例的 `useGlobalGuards()` 方法：

```
const app = await NestFactory.create(ApplicationModule);
app.useGlobalGuards(new RolesGuard());
```

!> 该 useGlobalGuards() 方法没有设置网关和微服务的看守器。

全局看守器用于整个应用程序，每个控制器和每个路由处理程序。在依赖注入方面，从任何模块外部注册的全局看守器（如上面的示例中所示）不能插入依赖项，因为它们不属于任何模块。为了解决此问题，您可以使用以下构造直接从任何模块设置一个看守器：

> *app.module.ts*

```
import { Module } from '@nestjs/common';
import { APP_GUARD } from '@nestjs/core';

@Module({
  providers: [
    {
      provide: APP_GUARD,
      useClass: RolesGuard,
    },
  ],
})
export class ApplicationModule {}
```

?> 另一种选择是使用执行上下文功能。另外，useClass并不是处理自定义提供商注册的唯一方法。在这里了解更多

反射器

看守器在正常工作，但我们仍然没有利用最重要的看守器的特征，即执行上下文。

现在， `RolesGuard` 是不可重用的。 我们如何知道处理程序需要处理哪些角色？ `CatsController` 可以有很多。 有些可能只适用于管理员，一些适用于所有人。

这就是为什么与看守器一起， `Nest` 提供了通过 `@ReflectMetadata()` 装饰器附加自定义元数据的能力。

> *cats.controller.ts*

```
@Post()
@ReflectMetadata('roles', ['admin'])
async create(@Body() createCatDto: CreateCatDto) {
  this.catsService.create(createCatDto);
}
```

!> `@ReflectMetadata()` 装饰器是从 `@nestjs/common` 包中导入的。

通过上面的构建，我们将 roles 元数据(roles 是一个关键，虽然['admin']是一个特定的值)附加到 `create()` 方法。 直接使用 `@ReflectMetadata()` 并不是一个好习惯。 相反，你应该总是创建你自己的装饰器。

> *roles.decorator.ts*

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

export const Roles = (...roles: string[]) =>
ReflectMetadata('roles', roles);
```

这样更简洁。 由于我们现在有一个 `@Roles()` 装饰器，所以我们可以在 `create()` 方法中使用它。

> *cats.controller.ts*

```
@Post()
@Roles('admin')
```

```
3.  async create(@Body() createCatDto: CreateCatDto) {
4.    this.catsService.create(createCatDto);
5.  }
```

我们再来关注一下 `RolesGuard` 。 现在，它立即返回 `true` ，允许请求继续。 为了反映元数据，我们将使用在 `@nestjs/core` 中提供的反射器 helper 类。

> *roles.guard.ts*

```
1.  import { Injectable, CanActivate, ExecutionContext } from
    '@nestjs/common';
2.  import { Observable } from 'rxjs';
3.  import { Reflector } from '@nestjs/core';
4.
5.  @Injectable()
6.  export class RolesGuard implements CanActivate {
7.    constructor(private readonly reflector: Reflector) {}
8.
9.    canActivate(context: ExecutionContext): boolean {
10.     const roles = this.reflector.get<string[]>('roles',
    context.getHandler());
11.     if (!roles) {
12.       return true;
13.     }
14.     const request = context.switchToHttp().getRequest();
15.     const user = request.user;
16.     const hasRole = () => user.roles.some((role) =>
    roles.includes(role));
17.     return user && user.roles && hasRole();
18.   }
19. }
```

?> 在 `node.js` 世界中，将授权用户附加到 `request` 对象是一种常见的做法。 这就是为什么我们假定 `request.user` 包含用户对象。

反射器允许我们通过指定的键很容易地反映元数据。 在上面的例子

中，我们反映了处理程序，因为它是对路由处理函数的引用。 如果我们也添加控制器反射部分，我们可以使这个警卫更通用。 为了提取控制器元数据，我们只是使用 `context.getClass()` 而不是 `getHandler()` 函数。

```
1. const roles = this.reflector.get<string[]>('roles',
   context.getClass());
```

现在，当用户尝试调用没有足够权限的 `/cat` `POST` 端点时， `Nest` 会自动返回以下响应：

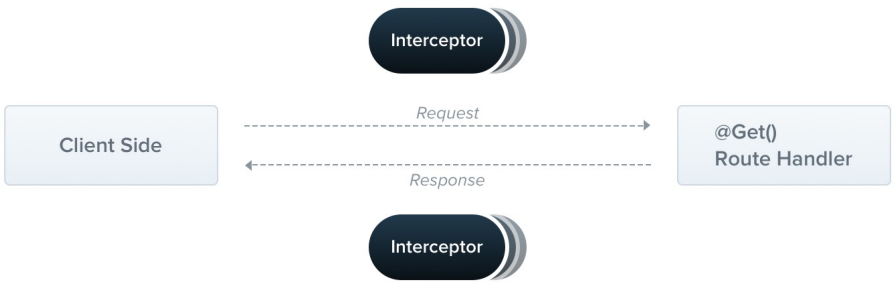
```
1. {
2.   "statusCode": 403,
3.   "message": "Forbidden resource"
4. }
```

实际上，返回 `false` 的守护器强制 `Nest` 抛出一个 `HttpException` 异常。如果您想要向最终用户返回不同的错误响应，则应该引发异常.这个异常可以被异常过滤器捕获。

拦截器

拦截器

拦截器是 `@Interceptor()` 装饰器注解的类。拦截器应该实现 `NestInterceptor` 接口。



拦截器具有一系列有用的功能，这些功能受面向切面编程（AOP）技术的启发。它们可以：

- 在函数执行之前/之后绑定额外的逻辑
- 转换从函数返回的结果
- 转换从函数抛出的异常

- 根据所选条件完全重写函数 (例如, 缓存目的)

基础

每个拦截器都有 `intercept()` 方法，它许2个参数的方法。 第一个是 `ExecutionContext` 实例（与看守器完全相同的对象）。 在 ExecutionContext 从继承 ArgumentsHost（第一次提到在这里）。 ArgumentsHost 是传递给原始处理程序的参数的一个包装，它根据应用程序的类型在引擎下包含不同的参数数组。

```
1. export interface ArgumentsHost {
2.   getArgs<T extends Array<any> = any[]>(): T;
3.   getArgByIndex<T = any>(index: number): T;
4.   switchToRpc(): RpcArgumentsHost;
5.   switchToHttp(): HttpArgumentsHost;
6.   switchToWs(): WsArgumentsHost;
7. }
```

ArgumentsHost 为我们提供了一套有用的方法, 帮助从基础数组中选取正确的参数。换言之, ArgumentsHost 只是一个参数数组而已。例如, 当在 HTTP 应用程序上下文中使用该保护程序时, ArgumentsHost 将在内部包含 [request, response] 数组。但是, 当当前上下文是 web 套接字应用程序时, 此数组将等于 [client, data]。通过此设计决策, 您可以访问最终传递给相应处理程序的任何参数。

ExecutionContext 提供多一点点。它扩展了 ArgumentsHost, 而且还提供了有关当前执行过程的更多细节。

```
1. export interface ExecutionContext extends ArgumentsHost {
2.   getClass<T = any>(): Type<T>;
3.   getHandler(): Function;
4. }
```

所述 getHandler() 返回一个参考当前处理的处理程序，而 getClass() 返回的类型的 Controller 此特定处理程序属于类别。使用换句话说，如果用户指向 create() 方法被定义和内注册 CatsController时，getHandler() 将返回一个 create() 参考方法和 getClass()，在这种情况下，将简单地返回一个 CatsController 类型（未实例）。

第二个参数是一个 `call$` ，一个 Observable 流。如果你不返回这个流，主处理程序将不会被评估。这是什么意思？基本上，这 `call$` 是一个推迟最终处理程序执行的流。比方说，有人提出了 POST `/cats` 请求。这个请求指向在 create() 里面定义的处理程序 CatsController 。如果 `call$` 一直拦截不会返回流，create() 则不会计算该方法。只有当 `call$` 流返回时，最终的方法才会被触发。为什么？因为 Nest 订阅到返回的流，并使用此流生成的值为最终用户创建单个响应或多个响应。此外，正如前面提到的， `call$` 是一个 Observable，也就是说，它为我们提供了一组非常强大的操作符，可以帮助处理例如响应操作。

截取之前/之后

第一个用例是使用拦截器在函数执行之前或之后添加额外的逻辑。当我们记录与应用程序的交互时，例如存储用户调用，异步调度事件或计算时间戳，这很有用。作为一个例子，我们来创建一个简单的例子 LoggingInterceptor。

> *logging.interceptor.ts*

```
1. import { Injectable, NestInterceptor, ExecutionContext } from
   '@nestjs/common';
2. import { Observable } from 'rxjs';
```

```
import { tap } from 'rxjs/operators';

@Injectable()
export class LoggingInterceptor implements NestInterceptor {
  intercept(
    context: ExecutionContext,
    call$: Observable<any>,
  ): Observable<any> {
    console.log('Before...');

    const now = Date.now();
    return call$.pipe(
      tap(() => console.log(`After... ${Date.now() - now}ms`)),
    );
  }
}
```

?> 拦截器的作用与控制器、组件、看守器和中间件相同，它们可以通过构造函数来插入依赖项。

由于 `stream$` 是一个 `RxJS` Observable,我们可以使用很多不同的操作符来操纵 stream 流。以上例子，使用了 `tap()` 运算符，它可以调用该函数观察序列的正常执行或异常终止。
要设置拦截器，我们使用从 `@nestjs/common` 包导入的 `@UseInterceptors()` 装饰器。与看守器一样，拦截器可以是控制器范围内的，方法范围内的或者全局范围内的。

> *cats.controller.ts*

```
@UseInterceptors(LoggingInterceptor)
export class CatsController {}
```

?> @UseInterceptors() 装饰器从 @nestjs/common 导入包。

由此，每个定义的路由处理器 CatsController 都会使用

LoggingInterceptor。当有人调用 GET `/cats` 端点时，您将在控制台窗口中看到以下输出：

```
Before...
After... 1ms
```

请注意，我们通过 LoggingInterceptor 类型而不是实例，使框架实例化责任并启用依赖注入。另一种可用的方法是通过立即创建的实例：

cats.controller.ts

```
@UseInterceptors(new LoggingInterceptor())
export class CatsController {}
```

如上所述，上面的构造将拦截器附加到此控制器声明的每个处理程序。如果我们决定只限制其中一个，我们只需在方法级别设置拦截器。为了绑定全局拦截器，我们使用 Nest 应用程序实例的 useGlobalInterceptors() 方法：

```
const app = await NestFactory.create(ApplicationModule);
app.useGlobalInterceptors(new LoggingInterceptor());
```

全局拦截器用于整个应用程序、每个控制器和每个路由处理程序。在依赖注入方面，从任何模块外部注册的全局拦截器（如上面的示例中所示）无法插入依赖项，因为它们不属于任何模块。为了解决此问题，您可以使用以下构造直接从任何模块设置一个看守器：

app.module.ts

```
import { Module } from '@nestjs/common';
import { APP_INTERCEPTOR } from '@nestjs/core';

@Module({
```

```
  providers: [
    {
      provide: APP_INTERCEPTOR,
      useClass: LoggingInterceptor,
    },
  ],
})
export class ApplicationModule {}
```

?> 另一种选择是使用执行上下文功能。另外，useClass 并不是处理自定义提供商注册的唯一方法。在这里了解更多。

响应映射

我们已经知道，call$ 是一个 Observable。此对象包含从路由处理程序返回的值，因此我们可以使用 `map()` 运算符轻松地对其进行改变。

?> 响应映射不适用于快速响应策略（无法直接使用 `@Res()` 对象）。

让我们创建一个 TransformInterceptor，它将打包响应并将其分配给 data 属性。

> *transform.interceptor.ts*

```
import { Injectable, NestInterceptor, ExecutionContext } from '@nestjs/common';
import { Observable } from 'rxjs';
import { map } from 'rxjs/operators';

export interface Response<T> {
  data: T;
}

@Injectable()
```

```
export class TransformInterceptor<T>
  implements NestInterceptor<T, Response<T>> {
  intercept(
    context: ExecutionContext,
    call$: Observable<T>,
  ): Observable<Response<T>> {
    return call$.pipe(map(data => ({ data })));
  }
}
```

?> 嵌套拦截器的工作就像一个具有异步 intercept() 函数的魅力，意思是，你可以毫不费力地切换你的函数，如果必须异步。

之后，当有人调用GET `/cats` 端点时，请求将如下所示（我们假设路由处理程序返回一个空 arry[]）：

```
{
    "data": []
}
```

拦截器在创建用于整个应用程序的可重用解决方案时具有巨大的潜力。例如，我们假设我们需要将每个发生的null值转换为空字符串''。我们可以使用一行代码并将拦截器绑定为全局代码。由于这一点，它会被每个注册处理程序自动重用。

```
import { Injectable, NestInterceptor, ExecutionContext } from
'@nestjs/common';
import { Observable } from 'rxjs';
import { map } from 'rxjs/operators';

@Injectable()
export class ExcludeNullInterceptor implements NestInterceptor {
  intercept(
    context: ExecutionContext,
    call$: Observable<any>,
  ): Observable<any> {
```

```
    return call$.pipe(map(value => value === null ? '' : value ));
  }
}
```

异常映射

另一个有趣的用例是利用 `catchError()` 操作符来覆盖抛出的异常：

> *exception.interceptor.ts*

```
import {
  Injectable,
  NestInterceptor,
  ExecutionContext,
  HttpStatus,
} from '@nestjs/common';
import { HttpException } from '@nestjs/common';
import { Observable } from 'rxjs';
import { catchError } from 'rxjs/operators';
import { _throw } from 'rxjs/observable/throw';

@Injectable()
export class ErrorsInterceptor implements NestInterceptor {
  intercept(
    context: ExecutionContext,
    call$: Observable<any>,
  ): Observable<any> {
    return call$.pipe(
      catchError(err =>
        _throw(new HttpException('Message', HttpStatus.BAD_GATEWAY)),
      ),
    );
  }
}
```

stream 重写

有时我们可能希望完全防止调用处理程序并返回不同的值（例如，由于性能问题而导致缓存），这是有多种原因的。一个很好的例子是缓存拦截器，它将缓存的响应存储在一些 TTL 中。不幸的是，这个功能需要更多的代码和由于简化，我们将提供一个基本的例子，应该简要解释的主要概念。

> *cache.interceptor.ts*

```
import { Injectable, NestInterceptor, ExecutionContext } from '@nestjs/common';
import { Observable } from 'rxjs';
import { of } from 'rxjs/observable/of';

@Injectable()
export class CacheInterceptor implements NestInterceptor {
  intercept(
    context: ExecutionContext,
    call$: Observable<any>,
  ): Observable<any> {
    const isCached = true;
    if (isCached) {
      return of([]);
    }
    return call$;
  }
}
```

这里有一个 CacheInterceptor 与硬编码的 isCached 变量和硬编码的 `response[]` 。我们在这里通过运算符创建返回了一个新的流，因此路由处理程序根本不会被调用。当有人调用使用 CacheInterceptor 的端点时，响应（硬编码的空数组）将返回 immedietely。为了创建通用解决方案，您可以利用反射器并创建自

定义修饰符。该反射器是很好地描述在看守器章。

返回流的可能性给了我们许多可能性。让我们考虑另一个常见的用例。假设您想处理超时。当端点在一段时间后没有返回任何内容时， 我们希望响应错误响应。

> *timeout.interceptor.ts*

```
import { Injectable, NestInterceptor, ExecutionContext } from '@nestjs/common';
import { Observable } from 'rxjs';
import { timeout } from 'rxjs/operators';

@Injectable()
export class TimeoutInterceptor implements NestInterceptor {
  intercept(
    context: ExecutionContext,
    call$: Observable<any>,
  ): Observable<any> {
    return call$.pipe(timeout(5000))
  }
}
```

5秒后，请求处理将被取消。

自定义装饰器

自定义路由参数装饰器

Nest 是基于 `装饰器` 这种语言特性而创建的。在很多常用的编程语言中 `装饰器` 都是一个很大众的概念，但在 JavaScript 语言中这个概念却比较新。所以为了更好地理解装饰器是如何工作的，你应该看看 这篇 文章。下面给出一个简单的定义：

ES2016 的装饰器是一个可以将目标对象，名称和属性描述符作为被修饰方法（returns function）的参数的表达式。你可以通过装饰器前缀 `@` 来使用它，并且把它放在你试图装饰的最上面。装饰器可以被定义为一个类或是属性。

Nest 提供了一组有用的参数装饰器，可以和 HTTP 路由处理器（route handlers）一起使用。下面是一组装饰器和普通表达式对象的对照。

`@Request()`	`req`
`@Response()`	`res`
`@Next()`	`next`
`@Session()`	`req.session`
`Param(param?: string)`	`req.params / req.params[param]`
`@Body(param?: string)`	`req.body / req.body[param]`
`@@Query(param?: string)`	`req.query / req.query[param]`
`@Headers(param?: string)`	`req.headers / req.headers[param]`

另外，你还可以创建你自己的自定义装饰器。为什么它很有用呢？

在 node.js 的世界中，把属性值附加到 `request` 对象中是一种很常见的做法。然后你可以在任何时候在路由处理程器（route handlers）中手动取到它们，例如，使用下面这个构造：

```
1. const user = req.user;
```

为了更加方便和透明地做到这一点，我们可以创建 `@User()` 装饰器并且在所有控制器中重复利用它。

> *user.decorator.ts*

```
1. import { createRouteParamDecorator } from '@nestjs/common';
2.
3. export const User = createRouteParamDecorator((data, req) => {
4.   return req.user;
5. });
```

现在你可以在任何你想要的地方很方便地使用它。

```
1. @Get()
2. async findOne(@User() user: UserEntity) {
3.   console.log(user);
4. }
```

传递数据（Passing data）

当你的装饰器的行为依赖于某些条件时，你可以使用 `data` 给装饰器的工厂函数传参。例如，下面的构造：

```
1. @Get()
2. async findOne(@User('test') user: UserEntity) {
3.   console.log(user);
```

```
}
```

可以通过 `data` 访问传进来的 `test` 字符串：

> *user.decorator.ts*

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

export const User = createParamDecorator((data, req) => {
  console.log(data); // test
  return req.user;
});
```

管道（Pipes）

Nest 对待自定义的路由参数装饰器和这些内置的装饰器（@Body()，@Param() 和 @Query()）一样。这意味着管道也会因为自定义注释参数（在本例中为 `user` 参数）而被执行。此外，你还可以直接将管道应用到自定义装饰器上：

```
@Get()
async findOne(@User(new ValidationPipe()) user: UserEntity) {
  console.log(user);
}
```

基础

用户提供商

当你想直接绑定到控制容器的Nest反转时，有很多场景。例如，任何常量值，基于当前环境创建的配置对象，外部库或预先计算的值（取决于其他几个定义的提供程序）。此外，您可以覆盖默认实现，例如在需要时使用不同的类或使用各种测试双打（用于测试目的）。

你应该始终牢记的一件重要事情就是Nest使用**tokens**(口令)来标识依赖关系。通常，自动生成的标记等于类。如果你想创建一个自定义提供者，你需要选择一个令牌。大多数情况下，自定义令牌都由纯字符串表示。遵循最佳实践，您应该在分隔文件中保存这些标记，例如，在 `constants.ts` 中。

我们来看看可用的选项。

使用值

`useValue` 语法在定义常量值，将外部库放入Nest容器或用模拟对象替换实际实现时非常有用。

```
import { connection } from './connection';

const connectionProvider = {
  provide: 'Connection',
  useValue: connection,
};

@Module({
  providers: [connectionProvider],
})
export class ApplicationModule {}
```

为了注入自定义用户提供，我们使用 `@Inject()` 装饰器。这个装饰器接受一个参数 - 令牌。

```
@Injectable()
class CatsRepository {
  constructor(@Inject('Connection') connection: Connection) {}
}
```

!> `@Inject()` 装饰器从 `@nestjs/common` 包中导入。

当您想要覆盖默认提供者的值时，比方说，您想强制Nest使用模拟 `CatsService` 以进行测试，您可以简单地使用现有类作为标记。

```
import { CatsService } from './cats.service';

const mockCatsService = {};
const catsServiceProvider = {
    provide: CatsService,
    useValue: mockCatsService,
```

```
};

@Module({
    imports: [CatsModule],
    providers: [catsServiceProvider],
})
export class ApplicationModule {}
```

在上面的例子中， `CatsService` 将被传递的 `mockCatsService` 模拟对象覆盖。这意味着，Nest不是手动创建 `CatsService` 实例，而是将此提供者视为已解决，并使用 `mockCatsService` 作为其代表值。

使用类

`useClass` 语法允许您对每个选定的因素使用不同的类。例如，我们有一个抽象的（或默认的） `ConfigService` 类。根据当前环境，Nest应该使用不同的配置服务实现。

```
const configServiceProvider = {
  provide: ConfigService,
  useClass: process.env.NODE_ENV === 'development'
  ? DevelopmentConfigService
  : ProductionConfigService,
};

@Module({
    providers: [configServiceProvider],
})
export class ApplicationModule {}
```

!>我们使用了 `ConfigService` 类，而不是自定义标记，因此我们已经覆盖了默认的实现。

在这种情况下，即使任何类依赖于 `ConfigService` ，Nest也会注入提供的类的实例

（ `DevelopmentConfigService` 或 `ProductionConfigService` ）。

使用工厂

`useFactory` 是一种动态创建提供程序的方式。实际提供者将等于工厂函数的返回值。工厂功能可以依靠几个不同的提供商或保持完全独立。这意味着工厂可以接受参数，Nest将在实例化过程中解析并传递参数。另外，这个函数可以异步返回值。这里点击 更多详解。当你的提供者不得不被动态的计算或者为了解决异步操作时可以使用它。

```
1.  const connectionFactory = {
2.    provide: 'Connection',
3.    useFactory: (optionsProvider: OptionsProvider) => {
4.      const options = optionsProvider.get();
5.      return new DatabaseConnection(options);
6.    },
7.    inject: [OptionsProvider],
8.  };
9.
10. @Module({
11.   providers: [connectionFactory],
12. })
13. export class ApplicationModule {}
```

?> 如果您的工厂需要其他提供者，则必须将其标记传入 `inject` 数组中。 Nest会以相同顺序将实例作为函数的参数传递。

导出自定义提供者

为了导出自定义提供者，我们可以使用令牌或整个对象。以下示例显示了一个标记案例：

```
1.  const connectionFactory = {
2.    provide: 'Connection',
```

```
  useFactory: (optionsProvider: OptionsProvider) => {
    const options = optionsProvider.get();
    return new DatabaseConnection(options);
  },
  inject: [OptionsProvider],
};

@Module({
  providers: [connectionFactory],
  exports: ['Connection'],
})
export class ApplicationModule {}
```

但是你也可以使用整个对象：

```
const connectionFactory = {
  provide: 'Connection',
  useFactory: (optionsProvider: OptionsProvider) => {
    const options = optionsProvider.get();
    return new DatabaseConnection(options);
  },
  inject: [OptionsProvider],
};

@Module({
  providers: [connectionFactory],
  exports: [connectionFactory],
})
export class ApplicationModule {}
```

支持我们

Nest是一个MIT许可的开源项目。它可以发展得益于这些令人敬畏的人们的支持。如果你想加入他们，请阅读 更多。

异步提供者

例如，在完成一些异步任务之前，应用程序启动必须被延迟，直到与数据库的连接建立，您应该考虑使用异步提供程序。为了创建一个 `异步` 提供者，我们使用 `useFactory` 。工厂必须返回 `Promise` （因此 `异步` 函数也适合）。

```
{
  provide: 'AsyncDbConnection',
  useFactory: async () => {
    const connection = await createConnection(options);
    return connection;
  },
},
```

?> 在这里了解 更多关于定制提供商的语法。

异步提供者可以通过它们的标记（在上述情况下，通过 `AsyncDbConnection` 标记）简单地注入其他组件。一旦异步提供程序已解析，每个依赖于异步提供程序的类都将被实例化。

以上示例用于演示目的。如果你正在寻找更详细的，请看 这里。

循环依赖

例如，当A类需要B类，而B类也需要A类时，就会产生循环依赖。Nest允许在提供者和模块之间创建循环依赖关系，但我们建议您尽可能避免。有时候避免这种关系真的很难，这就是为什么我们提供了一些方法来解决这个问题。

正向引用

正向引用允许Nest引用目前尚未被定义的参考。当 `CatsService` 和 `CommonService` 相互依赖时，关系的两端需要使

用 `@Inject()` 和 `forwardRef()` 实用工具，否则Nest不会实例化它们，因为所有基本元数据都不可用。让我们看看下面的代码片段：

> *cats.service.ts*

```
@Injectable()
export class CatsService {
  constructor(
    @Inject(forwardRef(() => CommonService))
    private readonly commonService: CommonService,
  ) {}
}
```

> *?>* *forwardRef()* 函数是从 *@nestjs/common* 包中导入的。

这是第一层的关系。现在让我们对 `CommonService` 进行相同的操作：

> *common.service.ts*

```
@Injectable()
export class CommonService {
  constructor(
    @Inject(forwardRef(() => CatsService))
    private readonly catsService: CatsService,
  ) {}
}
```

> *!>* 你不能保证哪个构造函数会被首先调用。

为了在模块之间创建循环依赖关系，必须在模块关联的两个部分上使用相同的 `forwardRef()` 实用程序：

> *common.module.ts*

```
@Module({
  imports: [forwardRef(() => CatsModule)],
})
export class CommonModule {}
```

> 模块参考

Nest提供了可以简单地注入到任何组件中的 `ModuleRef` 类。

> *cats.service.ts*
> *```typescript*
> *@Injectable()*
> *export class CatsService implements OnModuleInit {*
> *private service: Service;*
> *constructor(private readonly moduleRef: ModuleRef) {}*

```
onModuleInit() {
this.service = this.moduleRef.get(Service);
}
}
```

```
?> `ModuleRef`类是从`@nestjs/core`包中导入的。


模块引用有一个`get()`方法，它允许检索当前模块中可用的任何组件。
## 平台不可知论

Nest的整个观点是作为一个平台不可知的框架。平台独立性使得**创建可重用的逻辑部分**成为可能，人们可以利用多种不同类型的应用程序。框架的架构专注于适用于任何类型的服务器端解决方案。

**概述**类别主要指HTTP服务器(REST APIs)。但是，所有这些构建模块都可以轻松用于不同的传输层(`microservices`或`websockets`包)。此外，Nest还配备了专用的[GraphQL]($5.0-graphql?id=快速开始)模块。最后但并非最不重要的一点是,执行上下文功能有助于创建在Nest顶部的Node.js上运行的所有内容。

Nest鼓励成为一个完整的平台，为您的应用带来更高级别的可重用性。建造一次，随处使用！

## 测试


自动测试是**全功能软件产品**的重要组成部分。这对于至少覆盖系统中最敏感的部分非常
```

```
    重要。为了实现这个目标，我们产生了一系列不同的测试，例如集成测试，单元测试，e2e
    测试等等。 Nest提供了一系列改进测试体验的测试实用程序。
17.
18. 通常，您可以使用您喜欢的任何**测试框架**。我们不强制模具，选择适合您要求的任何东
    西。主要的Nest应用程序启动程序与Jest框架集成在一起，以减少开始编写测试时的开
    销，但仍然可以轻松删除它并使用任何其他工具。
19.
20. ### 安装
21.
22. 首先，我们需要安装所需的软件包：
23. ```bash
24. $ npm i --save-dev @nestjs/testing
```

单元测试

在下面的例子中，我们有两个不同的类，分别是 `CatsController` 和 `CatsService` 。如前所述，Jest被用作一个完整的测试框架。该框架的行为像一个测试运行者，并提供断言函数和测试双工实用程序，以帮助模拟，间谍等。我们已经手动强制执行 `catsService.findAll()` 方法来返回结果变量，一旦被调用。由此，我们可以测试 `catsController.findAll()` 是否返回预期的结果。

> *cats.controller.spec.ts*
> ```typescript
> *import { CatsController } from ‘./cats.controller’;*
> *import { CatsService } from ‘./cats.service’;*

```
describe(‘CatsController’, () => {
let catsController: CatsController;
let catsService: CatsService;

beforeEach(() => {
catsService = new CatsService();
catsController = new CatsController(catsService);
```

```
});

describe('findAll', () => {
it('should return an array of cats', async () => {
const result = ['test'];
jest.spyOn(catsService,
'findAll').mockImplementation(() => result);
```

```
    expect(await catsController.findAll()).toBe(result);
});
```

```
});
});
```

```
?> 保持你的测试文件在附近的测试类。测试文件应该有`.spec`或`.test`后缀。

到目前为止，我们没有使用任何现有的Nest测试工具。由于我们手动处理实例化测试类，因此上面的测试套件与Nest无关。这种类型的测试称为**隔离测试**。

### 测试工具
`@nestjs/testing`包给了我们一套提升测试过程的实用程序。让我们重写前面的例子，但现在使用暴露的`Test`类。

> cats.controller.spec.ts
```typescript
import { Test } from '@nestjs/testing';
import { CatsController } from './cats.controller';
import { CatsService } from './cats.service';

describe('CatsController', () => {
  let catsController: CatsController;
  let catsService: CatsService;

  beforeEach(async () => {
    const module = await Test.createTestingModule({
        controllers: [CatsController],
```
```

```
21.         providers: [CatsService],
22.       }).compile();
23.
24.     catsService = module.get<CatsService>(CatsService);
25.     catsController = module.get<CatsController>(CatsController);
26.   });
27.
28.   describe('findAll', () => {
29.     it('should return an array of cats', async () => {
30.       const result = ['test'];
31.       jest.spyOn(catsService, 'findAll').mockImplementation(() =>
    result);
32.
33.       expect(await catsController.findAll()).toBe(result);
34.     });
35.   });
36. });
```

`Test` 类有一个 `createTestingModule()` 方法，该方法将模块元数据（与在 `@Module()` 装饰器中传递的对象相同的对象）作为参数。这个方法创建了一个 `TestingModule` 实例，它反过来提供了一些方法，但是当涉及到单元测试时，它们中只有一个是有用的- `compile()` 。该功能是异步的，因此必须等待。一旦模块编译完成，您可以使用 `get()` 方法检索任何实例。

为了模拟一个真实的实例，你可以用自定义的提供者覆盖现有的 用户提供者。

端到端测试

当应用程序增长时，很难手动测试每个API端点的行为。端到端测试帮助我们确保一切工作正常并符合项目要求。为了执行e2e测试，我们使用与单元测试相同的配置，但另外我们利用允许模拟HTTP请求的超类库。

> *cats.e2e-spec.ts*
> *```typescript*
> *import * as request from ‘supertest’;*
> *import { Test } from ‘@nestjs/testing’;*
> *import { CatsModule } from ‘../../src/cats/cats.module’;*
> *import { CatsService } from ‘../../src/cats/cats.service’;*
> *import { INestApplication } from ‘@nestjs/common’;*

```
describe(‘Cats’, () => {
let app: INestApplication;
let catsService = { findAll: () => [‘test’] };

beforeAll(async () => {
const module = await Test.createTestingModule({
imports: [CatsModule],
})
.overrideProvider(CatsService)
.useValue(catsService)
.compile();
```

```
1.  app = module.createNestApplication();
2.  await app.init();
```

```
});

it( /GET cats , () => {
return request(app.getHttpServer())
.get(‘/cats’)
.expect(200)
.expect({
data: catsService.findAll(),
});
});
```

```
afterAll(async () => {
await app.close();
});
});
```

`` ?> 将您的 e2e 测试文件保存在 .e2e-spec 或 .e2e-test`后缀。

`cats.e2e-spec.ts` 测试文件包含一个HTTP端点测试(`/ cats`)。我们使用 `app.getHttpServer()` 方法来获取在Nest应用程序的后台运行的底层HTTP服务器。请注意， `TestingModule` 实例提供了 `overrideProvider()` 方法，因此我们可以覆盖导入模块声明的现有提供程序。另外，我们可以分别使用相应的方法， `overrideGuard()` ， `overrideInterceptor()` ， `overrideFilter()` 和 `overridePipe()` 来相继覆盖守卫，拦截器，过滤器和管道。

编译好的模块有几种在下表中详细描述的方法：

`createNestInstance()`	基于给定模块创建一个Nest实例（返回 `INestApplication` ），请注意，需要使用 `init()` 方法手动初始化应用程序
`createNestMicroservice()`	基于给定模块创建Nest微服务实例（返回 `INestMicroservice)`
`get()`	检索应用程序上下文中可用的控制器或提供程序（包括警卫，过滤器等）的实例
`select()`	例如，浏览模块树，从所选模块中提取特定实例（与启用严格模式一起使用）

技术

认证（Authentication）

身份验证是大多数现有应用程序的重要组成部分。有许多不同的方法、策略和方法来处理用户授权。我们最终决定使用什么取决于特定的应用程序要求，并且与它们的需求密切相关。

passport 是目前最流行的 node.js 认证库，为社区所熟知，并相继应用于许多生产应用中。将此工具与 Nest 框架集成起来非常简单。为了演示，我们将设置 passport-http-bearer 和 passport-jwt 策略。

安装

```
$ npm install --save @nestjs/passport passport passport-jwt
passport-http-bearer jsonwebtoken
```

承载

首先，我们将实现 passport-http-bearer 库。让我们从创建 `AuthService` 类开始，它将公开一个方法 `validateUser()`， 该方法的责任是通过提供的承载令牌查询用户。

> *auth.service.ts*

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

```
import { UsersService } from '../users/users.service';

@Injectable()
export class AuthService {
  constructor(private readonly usersService: UsersService) {}

  async validateUser(token: string): Promise<any> {
    return await this.usersService.findOneByToken(token);
  }
}
```

`validateUser()` 方法将 `token` 作为参数。此 token 是从与HTTP请求一起传递的授权标头中提取的。`findOneByToken()` 方法的职责是验证传递的 token 是否确实存在，并与数据库中的所有注册帐户关联。

完成 `AuthService` 后，我们必须创建相应的策略，passport 将使用该策略来验证请求。

> *http.strategy.ts*

```
import { BearerStrategy } from 'passport-http-bearer';
import { PassportStrategy } from '@nestjs/passport';
import { Injectable, UnauthorizedException } from '@nestjs/common';
import { AuthService } from './auth.service';

@Injectable()
export class HttpStrategy extends PassportStrategy(BearerStrategy)
{
  constructor(private readonly authService: AuthService) {
    super();
  }

  async validate(token: any, done: Function) {
    const user = await this.authService.validateUser(token);
    if (!user) {
      return done(new UnauthorizedException(), false);
```

```
    }
    done(null, user);
  }
}
```

`HttpStrategy` 使用 `AuthService` 来验证 token。当 token 有效时，passport 允许进行进一步的请求处理。否则，用户将收到 `401 (Unauthorized)` 响应。

然后，我们可以创建 `AuthModule`。

> *auth.module.ts*

```
import { Module } from '@nestjs/common';
import { AuthService } from './auth.service';
import { HttpStrategy } from './http.strategy';
import { UsersModule } from '../users/users.module';

@Module({
  imports: [UsersModule],
  providers: [AuthService, HttpStrategy],
})
export class AuthModule {}
```

?> 为了使用 `UsersService`，`AuthModule` 导入了 `UsersModule`。内部实现在这里并不重要。

然后，您可以在想要启用身份验证的任何位置使用 `AuthGuard`。

```
@Get('users')
@UseGuards(AuthGuard('bearer'))
findAll() {
  return [];
}
```

?> `AuthGuard` 是 `@nestjs/passport` 包中提供的。

`bearer` 是 passport 将使用的策略的名称。此外，`AuthGuard` 还接受第二个参数，`options` 对象，您可以通过该对象来确定 passport 行为。

JWT

第二种描述的方法是使用 JSON web token (JWT) 对端点进行身份验证。首先，让我们关注 AuthService 类。我们需要从token验证切换到基于负载的验证逻辑，并提供一种方法来为特定用户创建 JWT 令牌，然后可用于对传入请求进行身份验证。

> *auth.service.ts*

```
1.  import * as jwt from 'jsonwebtoken';
2.  import { Injectable } from '@nestjs/common';
3.  import { UsersService } from '../users/users.service';
4.  import { JwtPayload } from './interfaces/jwt-payload.interface';
5.
6.  @Injectable()
7.  export class AuthService {
8.    constructor(private readonly usersService: UsersService) {}
9.
10.   async createToken() {
11.     const user: JwtPayload = { email: 'user@email.com' };
12.     return jwt.sign(user, 'secretKey', { expiresIn: 3600 });
13.   }
14.
15.   async validateUser(payload: JwtPayload): Promise<any> {
16.     return await this.usersService.findOneByEmail(payload.email);
17.   }
18. }
```

?> 在最佳情况下，`jwt` package 和 token configuration (密钥和到期时间)应注册为 custom providers。

为了简化一个示例，我们创建了一个假用户。此外，到期时间和 `secretKey` 是硬编码的(在实际应用中，您应该考虑使用环境变量)。第二步是创建相应的 `JwtStrategy` 。

> *jwt.strategy.ts*

```
import { ExtractJwt, Strategy } from 'passport-jwt';
import { AuthService } from './auth.service';
import { PassportStrategy } from '@nestjs/passport';
import { Injectable, UnauthorizedException } from '@nestjs/common';
import { JwtPayload } from './interfaces/jwt-payload.interface';

@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy) {
  constructor(private readonly authService: AuthService) {
    super({
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      secretOrKey: 'secretKey',
    });
  }

  async validate(payload: JwtPayload, done: Function) {
    const user = await this.authService.validateUser(payload);
    if (!user) {
      return done(new UnauthorizedException(), false);
    }
    done(null, user);
  }
}
```

`JwtStrategy` 使用 `AuthService` 来验证解码的有效负载。有效负载有效(用户存在)时，passport允许进一步处理请求。否则，用户将收到 `401 (Unauthorized)` 响应

之后，我们可以转到 `AuthModule` 。

> *auth.module.ts*

```
import { Module } from '@nestjs/common';
import { AuthService } from './auth.service';
import { JwtStrategy } from './jwt.strategy';
import { UsersModule } from '../users/users.module';

@Module({
  imports: [UsersModule],
  providers: [AuthService, JwtStrategy],
})
export class AuthModule {}
```

?> 为了使用 `UsersService`，`AuthModule` 导入了 `UsersModule`。内部实现在这里并不重要。

然后，您可以在想要启用身份验证的任何位置使用 `AuthGuard`。

```
@Get('users')
@UseGuards(AuthGuard('jwt'))
findAll() {
  return [];
}
```

?> `AuthGuard` 是 `@nestjs/passport` 包中提供的。

`jwt` 是 passport 将使用的策略的名称。此外，`AuthGuard` 还接受第二个参数，`options` 对象，您可以通过该对象来确定 passport 行为。这里提供了一个完整的工作示例。

数据库（TypeORM）

为了减少开始与数据库进行连接所需的样板，Nest 提供了随时可用的 `@nestjs/typeorm` 软件包。我们选择了 TypeORM，因为它绝对是 Node.js 中可用的最成熟的对象关系映射器（ORM）。由于它是用 TypeScript编写的，所以它在Nest框架下运行得非常好。

首先，我们需要安装所有必需的依赖关系：

```
$ npm install --save @nestjs/typeorm typeorm mysql
```

?> 在本章中，我们将使用MySQL数据库，但TypeORM提供了许多不同的支持，如PostgreSQL，SQLite甚至MongoDB（NoSQL）。

一旦安装完成，我们可以将其 TypeOrmModule 导入到根目录中 ApplicationModule 。

> *app.module.ts*

```
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';

@Module({
  imports: [
    TypeOrmModule.forRoot({
      type: 'mysql',
      host: 'localhost',
      port: 3306,
      username: 'root',
      password: 'root',
      database: 'test',
      entities: [__dirname + '/../**/*.entity{.ts,.js}'],
      synchronize: true,
    }),
  ],
})
export class ApplicationModule {}
```

`forRoot()` 方法接受与 TypeORM 包中的 `createConnection()` 相同的配置对象。此外，我们可以在项目根目录中创建一个 `ormconfig.json` 文件，而不是将任何内容传递给它。

> *ormconfig.json*

```
{
  "type": "mysql",
  "host": "localhost",
  "port": 3306,
  "username": "root",
  "password": "root",
  "database": "test",
  "entities": ["src/**/**.entity{.ts,.js}"],
  "synchronize": true
}
```

现在我们可以简单地将圆括号留空：

> *app.module.ts*

```
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';

@Module({
  imports: [TypeOrmModule.forRoot()],
})
export class ApplicationModule {}
```

之后，`Connection` 和 `EntityManager` 将可用于注入整个项目（无需导入任何其他模块），例如以这种方式：

> *app.module.ts*

```
import { Connection } from 'typeorm';

@Module({
  imports: [TypeOrmModule.forRoot(), PhotoModule],
})
export class ApplicationModule {
  constructor(private readonly connection: Connection) {}
}
```

存储库模式

该TypeORM支持库的设计模式，使每个实体都有自己的仓库。这些存储库可以从数据库连接中获取。

首先，我们至少需要一个实体。我们将重用 `Photo` 官方文档中的实体。

> *photo/photo.entity.ts*

```
1.  import { Entity, Column, PrimaryGeneratedColumn } from 'typeorm';
2.
3.  @Entity()
4.  export class Photo {
5.    @PrimaryGeneratedColumn()
6.    id: number;
7.
8.    @Column({ length: 500 })
9.    name: string;
10.
11.   @Column('text')
12.   description: string;
13.
14.   @Column()
15.   filename: string;
16.
17.   @Column('int')
18.   views: number;
19.
20.   @Column()
21.   isPublished: boolean;
22. }
```

该 `Photo` 实体属于该 `photo` 目录。这个目录代表了 `PhotoModule` 。这是你决定在哪里保留你的模型文件。从我的观点来看，最好的方法是将它们放在他们的域中， 放在相应的模块目录中。

让我们看看 `PhotoModule`：

> *photo/photo.module.ts*

```
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';
import { PhotoService } from './photo.service';
import { PhotoController } from './photo.controller';
import { Photo } from './photo.entity';

@Module({
  imports: [TypeOrmModule.forFeature([Photo])],
  providers: [PhotoService],
  controllers: [PhotoController],
})
export class PhotoModule {}
```

此模块使用 `forFeature()` 方法定义定义哪些存储库应在当前范围内注册。

现在，我们可以使用 `@InjectRepository()` 修饰器向 `PhotoService` 注入 `PhotoRepository`：

> *photo/photo.service.ts*

```
import { Injectable, Inject } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';
import { Photo } from './photo.entity';

@Injectable()
export class PhotoService {
  constructor(
    @InjectRepository(Photo)
    private readonly photoRepository: Repository<Photo>,
  ) {}

  async findAll(): Promise<Photo[]> {
```

```
    return await this.photoRepository.find();
  }
}
```

?> 不要忘记将 `PhotoModule` 导入根 `ApplicationModule`。

多个数据库

某些项目可能需要多个数据库连接。幸运的是，这也可以通过本模块实现。要使用多个连接，首先要做的是创建这些连接。在这种情况下，连接命名成为必填项。

假设你有一个 `Person` 实体和一个 `Album` 实体，每个实体都存储在他们自己的数据库中。

```
@Module({
  imports: [
    TypeOrmModule.forRoot({
      type: 'postgres',
      host:  'photo_db_host',
      port: 5432,
      username: 'user',
      password: 'password',
      database: 'db',
      entities: [Photo],
      synchronize: true
    }),
    TypeOrmModule.forRoot({
      type: 'postgres',
      name: 'personsConnection',
      host:  'person_db_host',
      port: 5432,
      username: 'user',
      password: 'password',
      database: 'db',
      entities: [Person],
```

```
      synchronize: true
    }),
    TypeOrmModule.forRoot({
      type: 'postgres',
      name: 'albumsConnection',
      host:  'album_db_host',
      port: 5432,
      username: 'user',
      password: 'password',
      database: 'db',
      entities: [Album],
      synchronize: true
    })
  ]
})
export class ApplicationModule {}
```

?> 如果未为连接设置任何 `name` ，则该连接的名称将设置为 `default` 。请注意，不应该有多个没有名称或同名的连接，否则它们会被覆盖。

此时，您的 `Photo` 、 `Person` 和 `Album` 实体中的每一个都已在各自的连接中注册。通过此设置，您必须告诉 `TypeOrmModule.forFeature()` 函数和 `@InjectRepository()` 装饰器应该使用哪种连接。如果不传递任何连接名称，则使用 `default` 连接。

```
@Module({
  // ...
  TypeOrmModule.forFeature([Photo]),
  TypeOrmModule.forFeature([Person], 'personsConnection'),
  TypeOrmModule.forFeature([Album], 'albumsConnection')
})
export class ApplicationModule {}
```

您也可以为给定的连接注入 `Connection` 或 `EntityManager`：

```
@Injectable()
export class PersonService {
  constructor(
    @InjectConnection('personsConnection')
    private readonly connection: Connection,
    @InjectEntityManager('personsConnection')
    private readonly entityManager: EntityManager
  ) {}
}
```

测试

在单元测试我们的应用程序时，我们通常希望避免任何数据库连接，从而使我们的测试适合于独立，并使它们的执行过程尽可能快。但是我们的类可能依赖于从连接实例中提取的存储库。那是什么？解决方案是创建假存储库。为了实现这一点，我们应该设置 custom providers。事实上，每个注册的存储库都由 `entitynamereposition` 标记表示，其中 `EntityName` 是实体类的名称。

`@nestjs/typeorm` 包提供了基于给定实体返回准备好token的 `getRepositoryToken()` 函数。

```
@Module({
  providers: [
    PhotoService,
    {
      provide: getRepositoryToken(Photo),
      useValue: mockRepository,
    },
  ],
})
export class PhotoModule {}
```

现在，将使用硬编码 `mockRepository` 作为 `PhotoRepository`。每当任何提供程序使用 `@InjectRepository()` 修饰器请求 `PhotoRepository` 时，Nest 会使用注册的 `mockRepository` 对象。

这儿有一个可用的例子。

Mongo

文件上传

为了处理文件上传，Nest使用了multer中间件。这个中间件是完全可配置的，您可以根据您的应用需求调整其行为。

基本实例

当我们要上传单个文件时，我们只需将 `FileInterceptor ()` 与处理程序绑定在一起，然后使用 `@UploadedFile ()` 装饰器从 `request` 中取出 `file`。

```
@Post('upload')
@UseInterceptors(FileInterceptor('file'))
uploadFile(@UploadedFile() file) {
  console.log(file);
}
```

?> `FileInterceptor()` 和 `@UploadedFile()` 装饰都是 `@nestjs/common` 包提供的。

`FileInterceptor()` 接收两个参数，一个 `fieldName`（指向包含文件的 HTML 表单的字段）和可选 `options` 对象。这些 `MulterOptions` 等效于传入 multer 构造函数（此处有更多详细信息）

多个文件

为了同时上传多个文件，我们使用 `FilesInterceptor()` 。这个拦截器需要三个参数。 `fieldName` (保持不变)、可同时上载的最大文件数 `maxCount` 以及可选的 `MulterOptions` 对象。此外，要从 `request` 对象中选择文件，我们使用 `@UploadedFiles()` 装饰器

```
1. @Post('upload')
2. @UseInterceptors(FilesInterceptor('files'))
3. uploadFile(@UploadedFiles() files) {
4.   console.log(files);
5. }
```

?> `FilesInterceptor()` 和 `@UploadedFiles()` 装饰都是 `@nestjs/common` 包提供的。

日志记录（Logger）

Nest附带了一个默认的内部 `Logger` 实现，它在实例化过程中使用，并且在几个不同的情况下使用，例如 occurred exception 等。但有时，您可能希望完全禁用日志记录，或者提供自定义实现并自行处理消息。为了关闭 logger，我们使用 Nest 的 options 对象。

```
1. const app = await NestFactory.create(ApplicationModule, {
2.   logger: false,
3. });
4. await app.listen(3000);
```

不过，我们可能希望在底层使用不同的 logger，而不是禁用整个日志机制。为了达到这个目的，我们必须传递一个满足 `LoggerService` 接口的对象。例如，可以是内置的 `console` 。

```
const app = await NestFactory.create(ApplicationModule, {
  logger: console,
});
await app.listen(3000);
```

但这不是个好主意。但是，我们可以轻松创建自己的记录器。

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

export class MyLogger implements LoggerService {
  log(message: string) {}
  error(message: string, trace: string) {}
  warn(message: string) {}
}
```

然后，我们可以直接应用 `MyLogger` 实例：

```
const app = await NestFactory.create(ApplicationModule, {
  logger: new MyLogger(),
});
await app.listen(3000);
```

CORS

跨源资源共享（CORS）是一种允许从另一个域请求资源的机制。在引擎盖下，Nest 使用了 cors 软件包，该软件包提供了一些选项，您可以根据自己的要求进行自定义。为了启用CORS，你必须调用 `enableCors()` 方法。

```
const app = await NestFactory.create(ApplicationModule);
app.enableCors();
await app.listen(3000);
```

而且，你可以传递一个配置对象作为这个函数的参数。可用的属性在官

方的 cors 仓库中详细描述。另一种方法是使用Nest选项对象：

```
1. const app = await NestFactory.create(ApplicationModule, { cors:
   true });
2. await app.listen(3000);
```

您也可以使用cors配置对象，而不是传递布尔值。

Configuration

用于在不同的环境中运行的应用程序。根据环境的不同，应该使用各种配置变量。例如，很可能本地环境会针对特定数据库凭证进行中继，仅对本地数据库实例有效。为了解决这个问题，我们过去利用了 `.env` 包含键值对的文件，每个键代表一个特定的值，因为这种方法非常方便。

安装

为了解析我们的环境文件，我们将使用一个 dotenv 软件包。

```
1. $ npm i --save dotenv
```

Service

首先，我们来创建一个 `ConfigService` 类。

```
1. import * as dotenv from 'dotenv';
2. import * as fs from 'fs';
3.
4. export class ConfigService {
5.   private readonly envConfig: { [prop: string]: string };
6.
7.   constructor(filePath: string) {
8.     this.envConfig = dotenv.parse(fs.readFileSync(filePath))
```

```
  }

  get(key: string): string {
    return this.envConfig[key];
  }
}
```

这个类只有一个参数，`filePath` 是你的 `.env` 文件的路径。提供 `get()` 方法以启用对私有 `envConfig` 对象的访问，该对象包含在环境文件中定义的每个属性。

最后一步是创建一个 `ConfigModule`。

```
import { Module } from '@nestjs/common';
import { ConfigService } from './config.service';

@Module({
  providers: [
    {
      provide: ConfigService,
      useValue: new ConfigService(`${process.env.NODE_ENV}.env`),
    },
  ],
  exports: [ConfigService],
})
export class ConfigModule {}
```

`ConfigModule` 会注册 `ConfigService` 并将其导出。此外，我们还传递了 `.env` 文件的路径。此路径将因实际执行环境而异。现在，您可以简单地在任何位置插入 `ConfigService`，并根据传递的密钥提取特定值Sample。`.env` 文件可能如下所示：

> *development.env*

```
DATABASE_USER=test
DATABASE_PASSWORD=test
```

使用ConfigService

要从 `ConfigService` 访问环境变量，我们需要注入它。因此我们首先需要导入该模块。

> *app.module.ts*

```
@Module({
  imports: [ConfigModule],
  ...
})
```

之后，您可以使用注入标记来注入它。默认情况下，标记等于类名（在我们的例子中 `ConfigService` ）。

> *app.service.ts*

```
@Injectable()
export class AppService {
  private isAuthEnabled: boolean;
  constructor(config: ConfigService) {
    // Please take note that this check is case sensitive!
    this.isAuthEnabled = config.get('IS_AUTH_ENABLED') === 'true' ? true : false;
  }
}
```

您也可以将 `ConfigModule` 声明为全局模块，而不是在所有模块中重复导入 `ConfigModule` 。

高级配置（可选）

我们刚刚实现了一个基础 `ConfigService` 。但是，这种方法有几个缺点，我们现在将解决这些缺点：

- 缺少环境变量的名称和类型（无智能感知）
- 缺少提供对 `.env` 文件的验证
- env文件将布尔值作为string（‘`true`‘),提供，因此每次都必须将它们转换为 `boolean`

验证

我们将从验证提供的环境变量开始。如果未提供所需的环境变量或者它们不符合您的预定义要求，则可以抛出错误。为此，我们将使用npm包Joi。通过Joi，您可以定义一个对象模式（schema）并根据它来验证JavaScript对象。

安装Joi和它的类型（用于TypeScript用户）：

```
$ npm install --save joi
$ npm install --save-dev @types/joi
```

安装软件包后，我们就可以转到 `ConfigService`。

> *config.service.ts*

```
import * as Joi from 'joi';
import * as fs from 'fs';

export interface EnvConfig {
  [prop: string]: string;
}

export class ConfigService {
  private readonly envConfig: EnvConfig;

  constructor(filePath: string) {
    const config = dotenv.parse(fs.readFileSync(filePath));
    this.envConfig = this.validateInput(config);
  }
```

```
15.
16.   /**
17.    * Ensures all needed variables are set, and returns the
    validated JavaScript object
18.    * including the applied default values.
19.    */
20.   private validateInput(envConfig: EnvConfig): EnvConfig {
21.     const envVarsSchema: Joi.ObjectSchema = Joi.object({
22.       NODE_ENV: Joi.string()
23.         .valid(['development', 'production', 'test', 'provision'])
24.         .default('development'),
25.       PORT: Joi.number().default(3000),
26.       API_AUTH_ENABLED: Joi.boolean().required(),
27.     });
28.
29.     const { error, value: validatedEnvConfig } = Joi.validate(
30.       envConfig,
31.       envVarsSchema,
32.     );
33.     if (error) {
34.       throw new Error(`Config validation error: ${error.message}`);
35.     }
36.     return validatedEnvConfig;
37.   }
38. }
```

由于我们为 `NODE_ENV` 和 `PORT` 设置了默认值，因此如果不在环境文件中提供这些变量，验证将不会失败。然而，我们需要明确提供 `API_AUTH_ENABLED` 。如果我们的.env文件中的变量不是模式（schema）的一部分，则验证也会引发错误。此外，Joi 还会尝试将env字符串转换为正确的类型。

类属性

对于每个配置属性，我们必须添加一个getter方法。

> *config.service.ts*

```
get isApiAuthEnabled(): boolean {
  return Boolean(this.envConfig.API_AUTH_ENABLED);
}
```

用法示例

现在我们可以直接访问类属性。

> *config.service.ts*

```
@Injectable()
export class AppService {
  constructor(config: ConfigService) {
    if (config.isApiAuthEnabled) {
      // Authorization is enabled
    }
  }
}
```

HTTP模块

Axios 是丰富功能的 HTTP 客户端，广泛应用于许多应用程序中。这就是为什么Nest包装这个包，并公开它默认为内置 `HttpModule`。`HttpModule` 导出 `HttpService`，它只是公开了基于 axios 的方法来执行 HTTP 请求，而且还将返回类型转换为 `Observables`。

为了使用 `httppservice`，我们需要导入 `HttpModule`。

```
@Module({
  imports: [HttpModule],
  providers: [CatsService],
})
```

```
export class CatsModule {}
```

？> `HttpModule` 是 `@nestjs/common` 包提供的

然后，你可以注入 `HttpService` 。这个类可以从 `@nestjs/common` 包中获取。

```
@Injectable()
export class CatsService {
  constructor(private readonly httpService: HttpService) {}

  findAll(): Observable<AxiosResponse<Cat[]>> {
    return this.httpService.get('http://localhost:3000/cats');
  }
}
```

MVC

Nest 默认使用 Express 库，因此有关 Express 中的 MVC（模型 - 视图 - 控制器）模式的每个教程都与 Nest 相关。首先，让我们使用CLI工具搭建一个简单的Nest应用程序：

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

为了创建一个简单的MVC应用程序，我们必须安装一个模板引擎：

```
$ npm install --save hbs
```

我们决定使用 `hbs` 引擎，但您可以使用任何符合您要求的内容。安装过程完成后，我们需要使用以下代码配置快速实例：

> *main.ts*

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

async function bootstrap() {
  const app = await NestFactory.create(ApplicationModule);

  app.useStaticAssets(__dirname + '/public');
  app.setBaseViewsDir(__dirname + '/views');
  app.setViewEngine('hbs');

  await app.listen(3000);
}
bootstrap();
```

我们告诉express，该 `public` 目录将用于存储静态文件，`views` 将包含模板，并且 `hbs` 应使用模板引擎来呈现 HTML 输出。

现在，让我们在该文件夹内创建一个 `views` 目录和一个 `index.hbs` 模板。在模板内部，我们将打印从控制器传递的 `message`：

> *index.hbs*

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>App</title>
</head>
<body>
  {{ message }}
</body>
</html>
```

然后，打开 `app.controller` 文件，并用以下代码替换 `root()` 方

法：

> *app.controller.ts*

```
import { Get, Controller, Render } from '@nestjs/common';

@Controller()
export class AppController {
  @Get()
  @Render('index')
  root() {
    return { message: 'Hello world!' };
  }
}
```

?> 事实上，当 Nest 检测到 `@Res()` 装饰器时，它会注入 `response` 对象。在这里了解更多关于它的能力。

在应用程序运行时，打开浏览器访问 `http://localhost:3000/` 你应该看到这个 Hello world! 消息。

这里有一个可用的例子

Fastify

如本章所述，我们可以将任何兼容的HTTP提供程序与Nest一起使用。其中一个是fastify库。为了创建一个具有fastify的MVC应用程序，我们必须安装以下软件包：

```
$ npm i --save fastify point-of-view handlebars
```

接下来的步骤几乎涵盖了与express库相同的内容(差别很小)。安装过程完成后，我们需要打开 `main.ts` 文件并更新其内容：

> *main.ts*

```
import { NestFactory } from '@nestjs/core';
import { ApplicationModule } from './app.module';
import { FastifyAdapter } from '@nestjs/core/adapters/fastify-adapter';
import { join } from 'path';

async function bootstrap() {
  const app = await NestFactory.create(ApplicationModule, new FastifyAdapter());
  app.useStaticAssets({
    root: join(__dirname, 'public'),
    prefix: '/public/',
  });
  app.setViewEngine({
    engine: {
      handlebars: require('handlebars'),
    },
    templates: join(__dirname, 'views'),
  });
  await app.listen(3000);
}
bootstrap();
```

API略有不同，但这些方法调用背后的想法保持不变。此外，我们还必须确保传递到 `@Render()` 装饰器中的模板名称包含文件扩展名。

> *app.controller.ts*

```
import { Get, Controller, Render } from '@nestjs/common';

@Controller()
export class AppController {
  @Get()
  @Render('index.hbs')
  root() {
    return { message: 'Hello world!' };
  }
}
```

在应用程序运行时，打开浏览器并导航至 `http://localhost:3000/` 。你应该看到这个Hello world!消息。

这里有 一个可用的例子。

性能（Fastify）

在底层，Nest使用了Express，但如前所述，它提供了与各种其他库的兼容性，例如 Fastify。它是怎么工作的？事实上，Nest需要使用您最喜欢的库，它是一个兼容的适配器，它主要将相应的处理程序代理到适当的库特定的方法。此外，您的库必须至少提供与express类似的请求-响应周期管理。

Fastify非常适合这里，因为它以与express类似的方式解决设计问题。然而，fastify的速度要快得多，达到了几乎两倍的基准测试结果。问题是，为什么Nest仍然使用express作为默认的HTTP提供程序？因为express是应用广泛、广为人知的，而且拥有一套庞大的兼容中间件。

但我们并没有将人们锁定在单一的模式中。我们让他们使用任何他们需要的东西。如果您关心真正出色的性能，Fastify是一个更好的选择，这就是为什么我们提供内置 `FastifyAdapter` 有助于将此库与Nest整合在一起的原因。

安装

首先，我们需要安装所需的软件包：

```
1.  $ npm i --save fastify fastify-formbody
```

适配器（Adapter）

安装fastify后，我们可以使用 `FastifyAdapter` 。

```
1. import { NestFactory } from '@nestjs/core';
2. import { FastifyAdapter } from '@nestjs/core/adapters';
3. import { ApplicationModule } from './app.module';
4.
5. async function bootstrap() {
6.   const app = await NestFactory.create(ApplicationModule, new
   FastifyAdapter());
7.   await app.listen(3000);
8. }
9. bootstrap();
```

就这样。此外，您还可以通过 `FastifyAdapter` 构造函数将选项传递到fastify构造函数中。请记住，Nest现在使用fastify作为HTTP提供程序，这意味着在express上转发的每个配方都将不再起作用。您应该使用fastify等效软件包。

热重载（Webpack）

对应用程序的引导过程影响最大的是TypeScript编译。但问题是，每次发生变化时，我们是否必须重新编译整个项目？一点也不。这就是为什么 webpack HMR（Hot-Module Replacement）大大减少了实例化您的应用程序所需的时间。

安装

首先，我们安装所需的软件包：

```
1. $ npm i --save-dev webpack webpack-cli webpack-node-externals
```

配置（Configuration）

然后，我们需要创建一个 `webpack.config.js` ，它是webpack的一个配置文件，并将其放入根目录。

```
const webpack = require('webpack');
const path = require('path');
const nodeExternals = require('webpack-node-externals');

module.exports = {
  entry: ['webpack/hot/poll?100', './src/main.ts'],
  watch: true,
  target: 'node',
  externals: [
    nodeExternals({
      whitelist: ['webpack/hot/poll?100'],
    }),
  ],
  module: {
    rules: [
      {
        test: /.tsx?$/,
        use: 'ts-loader',
        exclude: /node_modules/,
      },
    ],
  },
  mode: 'development',
  resolve: {
    extensions: ['.tsx', '.ts', '.js'],
  },
  plugins: [new webpack.HotModuleReplacementPlugin()],
  output: {
    path: path.join(__dirname, 'dist'),
    filename: 'server.js',
  },
};
```

此配置告诉webpack关于我们的应用程序的一些基本内容。其中有一个

入口文件，应该使用哪个目录来保存编译后的文件，以及为了编译源文件我们要使用哪种加载程序。基本上，你不应该担心太多，你根本不需要理解这个文件的内容。

热模块更换

为了启用HMR，我们必须打开Nest应用程序入口文件（这是 `main.ts` ）并添加一些关键的事情。

```
declare const module: any;

async function bootstrap() {
  const app = await NestFactory.create(ApplicationModule);
  await app.listen(3000);

  if (module.hot) {
    module.hot.accept();
    module.hot.dispose(() => app.close());
  }
}
bootstrap();
```

就这样。为了简化执行过程，请将这两行添加到 `package.json` 文件的脚本中。

```
"start": "node dist/server",
"webpack": "webpack --config webpack.config.js"
```

现在只需打开你的命令行并运行下面的命令：

```
$ npm run webpack
```

webpack开始监视文件后，在另一个命令行窗口中运行另一个命令：

```
$ npm run start
```

这里有一个可用的例子

GraphQL

GraphQL

快速开始

GraphQL 是一种用于 API 的查询语言。这是 GraphQL 和 REST

之间一个很好的比较（译者注： GraphQL 替代 REST 是必然趋势）。在这组文章中，我们不会解释什么是 GraphQL，而是演示如何使用 `@nestjs/GraphQL` 模块。

GraphQLModule 只不过是 Apollo 服务器的包装器。我们没有造轮子，而是提供一个现成的模块，这让 GraphQL 和 Nest 有了比较简洁的融合方式。

安装

首先，我们需要安装所需的软件包：

```
$ npm i --save @nestjs/graphql apollo-server-express graphql-tools graphql
```

译者注： fastify 请参考：https://github.com/coopnd/fastify-apollo

Apollo 中间件

安装软件包后，我们可以应用 `apollo-server-express` 软件包提供的 GraphQL 中间件 ：

> *app.module.ts*

```
import { Module, MiddlewareConsumer, NestModule } from '@nestjs/common';
import { graphqlExpress } from 'apollo-server-express';
import { GraphQLModule } from '@nestjs/graphql';

@Module({
  imports: [GraphQLModule],
})
export class ApplicationModule implements NestModule {
  configure(consumer: MiddlewareConsumer) {
```

```
10.     consumer
11.       .apply(graphqlExpress(req => ({ schema: {}, rootValue: req
    })))
12.       .forRoutes('/graphql');
13.   }
14. }
```

就这样。我们传递一个空对象作为 GraphQL `schema` 和 `req`（请求对象）`rootValue`。此外，还有其他一些可用的 `graphqlExpress` 选项，你可以在这里阅读它们。

Schema

为了创建 schema，我们使用的是 `GraphQLFactory` 它是 `@nestjs/graphql` 包的一部分。此组件提供了一个createSchema（）方法，该方法接受与makeexecumbleschema（）函数相同的对象，这里详细描述。

schema 选项对象至少需要 `resolvers` 和 `typeDefs` 属性。您可以手动传递类型定义，或者使用 `GraphQLFactory` 的实用程序 `mergeTypesByPaths()` 方法。让我们看一下下面的示例：

> *app.module.ts*

```
1. import { Module, MiddlewareConsumer, NestModule } from
   '@nestjs/common';
2. import { graphqlExpress } from 'apollo-server-express';
3. import { GraphQLModule, GraphQLFactory } from '@nestjs/graphql';
4.
5. @Module({
6.   imports: [GraphQLModule],
7. })
8. export class ApplicationModule implements NestModule {
9.   constructor(private readonly graphQLFactory: GraphQLFactory) {}
10.
```

```
11.   configure(consumer: MiddlewareConsumer) {
12.     const typeDefs =
    this.graphQLFactory.mergeTypesByPaths('./**/*.graphql');
13.     const schema = this.graphQLFactory.createSchema({ typeDefs });
14.
15.     consumer
16.       .apply(graphqlExpress(req => ({ schema, rootValue: req })))
17.       .forRoutes('/graphql');
18.   }
19. }
```

?> 在此处了解关于GraphQL Schema 的更多信息。

在这种情况下， `GraphQLFactory` 将遍历每个目录，并合并具有 `.graphql` 扩展名的文件。之后，我们可以使用这些特定的类型定义来创建一个 `schema` 。 `resolvers` 将自动反映出来。

在这里， 您可以更多地了解解析器映射的实际内容。

解析器映射

当使用 `graphql-tools` 时，您必须手动创建解析器映射。以下示例是从Apollo文档复制并粘贴的，您可以在其中阅读更多内容：

```
1.  import { find, filter } from 'lodash';
2.
3.  // example data
4.  const authors = [
5.    { id: 1, firstName: 'Tom', lastName: 'Coleman' },
6.    { id: 2, firstName: 'Sashko', lastName: 'Stubailo' },
7.    { id: 3, firstName: 'Mikhail', lastName: 'Novikov' },
8.  ];
9.  const posts = [
10.   { id: 1, authorId: 1, title: 'Introduction to GraphQL', votes: 2
    },
11.   { id: 2, authorId: 2, title: 'Welcome to Meteor', votes: 3 },
```

```
  { id: 3, authorId: 2, title: 'Advanced GraphQL', votes: 1 },
  { id: 4, authorId: 3, title: 'Launchpad is Cool', votes: 7 },
];

const resolverMap = {
  Query: {
    author(obj, args, context, info) {
      return find(authors, { id: args.id });
    },
  },
  Author: {
    posts(author, args, context, info) {
      return filter(posts, { authorId: author.id });
    },
  },
};
```

使用该 `@nestjs/graphql` 包，解析器映射是使用元数据自动生成的。我们用等效的 Nest-Way 代码重写上面的例子。

```
import { Query, Resolver, ResolveProperty } from '@nestjs/graphql';
import { find, filter } from 'lodash';

// example data
const authors = [
  { id: 1, firstName: 'Tom', lastName: 'Coleman' },
  { id: 2, firstName: 'Sashko', lastName: 'Stubailo' },
  { id: 3, firstName: 'Mikhail', lastName: 'Novikov' },
];
const posts = [
  { id: 1, authorId: 1, title: 'Introduction to GraphQL', votes: 2 },
  { id: 2, authorId: 2, title: 'Welcome to Meteor', votes: 3 },
  { id: 3, authorId: 2, title: 'Advanced GraphQL', votes: 1 },
  { id: 4, authorId: 3, title: 'Launchpad is Cool', votes: 7 },
];

```

```
@Resolver('Author')
export class AuthorResolver {
  @Query()
  author(obj, args, context, info) {
    return find(authors, { id: args.id });
  }

  @ResolveProperty()
  posts(author, args, context, info) {
    return filter(posts, { authorId: author.id });
  }
}
```

该 `@Resolver()` 装饰并不影响查询和变更。它只告诉 Nest 每个 `@ResolveProperty()` 都有一个父级，在本例中是 `Author` 。

?> 如果您使用的是 `@Resolver()` 修饰器，则不必将类标记为 `@Injectable()` ，否则，它将是必需的。

通常，我们会使用类似 `getAuthor()` 或 `getPosts()` 作为方法名。我们也可以轻松地做到这一点：

```
import { Query, Resolver, ResolveProperty } from '@nestjs/graphql';
import { find, filter } from 'lodash';

// example data
const authors = [
  { id: 1, firstName: 'Tom', lastName: 'Coleman' },
  { id: 2, firstName: 'Sashko', lastName: 'Stubailo' },
  { id: 3, firstName: 'Mikhail', lastName: 'Novikov' },
];
const posts = [
  { id: 1, authorId: 1, title: 'Introduction to GraphQL', votes: 2 },
  { id: 2, authorId: 2, title: 'Welcome to Meteor', votes: 3 },
  { id: 3, authorId: 2, title: 'Advanced GraphQL', votes: 1 },
  { id: 4, authorId: 3, title: 'Launchpad is Cool', votes: 7 },
```

```
];

@Resolver('Author')
export class AuthorResolver {
  @Query('author')
  getAuthor(obj, args, context, info) {
    return find(authors, { id: args.id });
  }

  @ResolveProperty('posts')
  getPosts(author, args, context, info) {
    return filter(posts, { authorId: author.id });
  }
}
```

?> `@Resolver()` 装饰可以在方法级别被使用。

重构

上述代码背后的想法是为了展示 Apollo 和 Nest-way 之间的区别，以便简单地转换代码。现在，我们要做一个小的重构来利用 Nest 架构的优势，使其成为一个真实的例子。

```
@Resolver('Author')
export class AuthorResolver {
  constructor(
    private readonly authorsService: AuthorsService,
    private readonly postsService: PostsService,
  ) {}

  @Query('author')
  async getAuthor(obj, args, context, info) {
    const { id } = args;
    return await this.authorsService.findOneById(id);
  }

  @ResolveProperty('posts')
```

```
15.   async getPosts(author, args, context, info) {
16.     const { id } = author;
17.     return await this.postsService.findAll({ authorId: id });
18.   }
19. }
```

现在我们必须在某个地方注册 `AuthorResolver`，例如在新创建的 `AuthorsModule` 中。

```
1. @Module({
2.   imports: [PostsModule],
3.   providers: [AuthorsService, AuthorResolver],
4. })
5. export class AuthorsModule {}
```

`GraphQLModule` 将负责反映「元数据」，并自动将类转换为正确的解析器映射。你必须做的唯一一件事就是在某个地方导入此模块，因此 Nest 将知道 `AuthorsModule` 存在。

类型定义

最后一个缺失的部分是类型定义（阅读更多）文件。让我们在解析器类附近创建它。

> *author-types.graphql*

```
1. type Author {
2.   id: Int!
3.   firstName: String
4.   lastName: String
5.   posts: [Post]
6. }
7.
8. type Post {
9.   id: Int!
10.   title: String
```

```
  votes: Int
}

type Query {
  author(id: Int!): Author
}
```

就这样。我们创建了一个 `author(id: Int!)` 查询。

?> 此处了解有关 GraphQL 查询的更多信息。

变更（Mutations）

在 GraphQL 中，为了修改服务器端数据，我们使用了变更（Mutations）。了解更多

Apollo 官方文献中有一个 `upvotePost ()` 变更的例子。这种变更允许增加后 `votes` 属性值。

```
Mutation: {
  upvotePost: (_, { postId }) => {
    const post = find(posts, { id: postId });
    if (!post) {
      throw new Error(`Couldn't find post with id ${postId}`);
    }
    post.votes += 1;
    return post;
  },
}
```

为了在 Nest-way 中创建等价的变更，我们将使用 `@Mutation()` 装饰器。让我们扩展在上一节（解析器映射）中使用的 `AuthorResolver`。

```
import { Query, Mutation, Resolver, ResolveProperty } from
```

```
    '@nestjs/graphql';
import { find, filter } from 'lodash';

// example data
const authors = [
  { id: 1, firstName: 'Tom', lastName: 'Coleman' },
  { id: 2, firstName: 'Sashko', lastName: 'Stubailo' },
  { id: 3, firstName: 'Mikhail', lastName: 'Novikov' },
];
const posts = [
  { id: 1, authorId: 1, title: 'Introduction to GraphQL', votes: 2
},
  { id: 2, authorId: 2, title: 'Welcome to Meteor', votes: 3 },
  { id: 3, authorId: 2, title: 'Advanced GraphQL', votes: 1 },
  { id: 4, authorId: 3, title: 'Launchpad is Cool', votes: 7 },
];

@Resolver('Author')
export class AuthorResolver {
  @Query('author')
  getAuthor(obj, args, context, info) {
    return find(authors, { id: args.id });
  }

  @Mutation()
  upvotePost(_, { postId }) {
    const post = find(posts, { id: postId });
    if (!post) {
      throw new Error(`Couldn't find post with id ${postId}`);
    }
    post.votes += 1;
    return post;
  }

  @ResolveProperty('posts')
  getPosts(author) {
    return filter(posts, { authorId: author.id });
  }
```

```
38. }
```

重构

我们要做一个小的重构来利用Nest架构的优势，将其变为一个 真实的例子.

```
1.  @Resolver('Author')
2.  export class AuthorResolver {
3.    constructor(
4.      private readonly authorsService: AuthorsService,
5.      private readonly postsService: PostsService,
6.    ) {}
7.
8.    @Query('author')
9.    async getAuthor(obj, args, context, info) {
10.     const { id } = args;
11.     return await this.authorsService.findOneById(id);
12.   }
13.
14.   @Mutation()
15.   async upvotePost(_, { postId }) {
16.     return await this.postsService.upvoteById({ id: postId });
17.   }
18.
19.   @ResolveProperty('posts')
20.   async getPosts(author) {
21.     const { id } = author;
22.     return await this.postsService.findAll({ authorId: id });
23.   }
24. }
```

就这样。业务逻辑被转移到了 `PostsService` 。

类型定义

最后一步是将我们的变更添加到现有的类型定义中。

> *author-types.graphql*

```
type Author {
  id: Int!
  firstName: String
  lastName: String
  posts: [Post]
}

type Post {
  id: Int!
  title: String
  votes: Int
}

type Query {
  author(id: Int!): Author
}

type Mutation {
  upvotePost(postId: Int!): Post
}
```

该 `upvotePost(postId: Int!): Post` 变更后现在可用！

订阅（Subscriptions）

订阅只是查询和变更等另一种 GraphQL 操作类型。它允许通过双向传输层创建实时订阅，主要通过 websockets 实现。阅读更多的订阅。

以下是 `commentAdded` 订阅示例，可直接从官方 Apollo 文档复制和粘贴：

```
Subscription: {
  commentAdded: {
    subscribe: () => pubSub.asyncIterator('commentAdded')
  }
}
```

?> `pubsub` 是一个 `PubSub` 类的实例。在这里阅读更多

为了以Nest方式创建等效订阅，我们将使用 `@Subscription()` 装饰器。让我们扩展 `AuthorResolver` 在解析器映射部分中的使用。

```
import { Query, Resolver, Subscription, ResolveProperty } from '@nestjs/graphql';
import { find, filter } from 'lodash';
import { PubSub } from 'graphql-subscriptions';

// example data
const authors = [
  { id: 1, firstName: 'Tom', lastName: 'Coleman' },
  { id: 2, firstName: 'Sashko', lastName: 'Stubailo' },
  { id: 3, firstName: 'Mikhail', lastName: 'Novikov' },
];
const posts = [
  { id: 1, authorId: 1, title: 'Introduction to GraphQL', votes: 2 },
  { id: 2, authorId: 2, title: 'Welcome to Meteor', votes: 3 },
  { id: 3, authorId: 2, title: 'Advanced GraphQL', votes: 1 },
  { id: 4, authorId: 3, title: 'Launchpad is Cool', votes: 7 },
];

// example pubsub
const pubSub = new PubSub();

@Resolver('Author')
export class AuthorResolver {
  @Query('author')
  getAuthor(obj, args, context, info) {
```

```
25.     return find(authors, { id: args.id });
26.   }
27.
28.   @Subscription()
29.   commentAdded() {
30.     return {
31.       subscribe: () => pubSub.asyncIterator('commentAdded'),
32.     };
33.   }
34.
35.   @ResolveProperty('posts')
36.   getPosts(author) {
37.     return filter(posts, { authorId: author.id });
38.   }
39. }
```

重构

我们在这里使用了一个本地 `PubSub` 实例。相反，我们应该将 `PubSub` 定义为一个组件，通过构造函数（使用 `@Inject()` 装饰器）注入它，并在整个应用程序中重用它。您可以在此了解有关嵌套自定义组件的更多信息。

类型定义

最后一步是更新类型定义（阅读更多）文件。

> *author-types.graphql*

```
1. type Author {
2.   id: Int!
3.   firstName: String
4.   lastName: String
5.   posts: [Post]
6. }
7.
```

```
type Post {
  id: Int!
  title: String
  votes: Int
}

type Query {
  author(id: Int!): Author
}

type Comment {
  id: String
  content: String
}

type Subscription {
  commentAdded(repoFullName: String!): Comment
}
```

就这样。我们创建了一个 `commentAdded(repoFullName: String!): Comment` 订阅。另外，我们应该创建一个发送和订阅事件的 Web sockets 服务器，但为了保持这个示例尽可能简单，我们省略了这部分。尽管如此，你可以在这里找到完整的示例实现。

标量

为了定义一个自定义标量（在这里阅读更多关于标量的信息），我们必须创建一个类型定义和一个专用的解析器。在这里（如在官方文档中），我们将采取 `graphql-type-json` 包用于演示目的。这个npm包定义了一个 `JSON` GraphQL标量类型。首先，让我们安装包：

```
$ npm i --save graphql-type-json
```

然后，我们必须将自定义解析器传递给 `createSchema()` 函数：

```
const resolvers = { JSON: GraphQLJSON };
const schema = this.graphQLFactory.createSchema({ typeDefs,
resolvers });
```

?> `GraphQLJSON` 是从 `graphql-type-json` 包中导入的

现在，我们可以在类型定义中使用 `JSON` 标量：

```
scalar JSON

type Foo {
  field: JSON
}
```

看守器和拦截器

在 GraphQL 中，许多文章抱怨如何处理诸如身份验证或操作的副作用之类的问题。我们应该把它放在业务逻辑里面吗？我们是否应该使用高阶函数来增强查询和变更，例如使用授权逻辑？没有单一的答案。

Nest 生态系统正试图利用现有的功能，如看守器和拦截器来帮助解决这个问题。其背后的想法是减少冗余，并为您提供工具，帮助创建结构良好，可读性强且一致的应用程序。

使用

您可以像在简单的 REST 应用程序中一样使用看守器和拦截器。它们的行为等同于在 `graphqlExpress` 中间件中作为 `rootValue` 传递请求。让我们看一下下面的代码：

```
@Query('author')
@UseGuards(AuthGuard)
async getAuthor(obj, args, context, info) {
  const { id } = args;
```

```
5.    return await this.authorsService.findOneById(id);
6.  }
```

由于这一点，您可以将您的身份验证逻辑移至看守器(guard)，甚至可以复用与 REST 应用程序中相同的看守器(guard)类。拦截器的工作方式完全相同：

```
1.  @Mutation()
2.  @UseInterceptors(EventsInterceptor)
3.  async upvotePost(_, { postId }) {
4.    return await this.postsService.upvoteById({ id: postId });
5.  }
```

写一次，随处使用：)

Schema 拼接

Schema 拼接功能允许从多个底层GraphQL API创建单个GraphQL模式。你可以在这里阅读更多。

代理（Proxying）

要在模式之间添加代理字段的功能，您需要在它们之间创建额外的解析器。我们来看看 Apollo 文档中的例子：

```
1.  mergeInfo => ({
2.    User: {
3.      chirps: {
4.        fragment: `fragment UserFragment on User { id }`,
5.        resolve(parent, args, context, info) {
6.          const authorId = parent.id;
7.          return mergeInfo.delegate(
8.            'query',
9.            'chirpsByAuthorId',
```

```
          {
            authorId,
          },
          context,
          info,
        );
      },
    },
  }
})
```

在这里，我们将 `User` 的 `chirps` 属性委托给另一个 GraphQL API。为了在 Nest-way 中实现相同的结果，我们使用 `@DelegateProperty()` 装饰器。

```

@Resolver('User')
@DelegateProperty('chirps')
findChirpsByUserId() {
  return (mergeInfo: MergeInfo) => ({
    fragment: `fragment UserFragment on User { id }`,
    resolve(parent, args, context, info) {
      const authorId = parent.id;
      return mergeInfo.delegate(
        'query',
        'chirpsByAuthorId',
        {
          authorId,
        },
        context,
        info,
      );
    },
  });
}
```

?> `@Resolver()` 装饰器在这里用于方法级，但也可以在顶级

(class) 级别使用它。

然后，让我们再回到 `graphqlExpress` 中间件。我们需要合并我们的架构并在它们之间添加委托。要创建委托，我们使用 `GraphQLFactory` 类的 `createDelegates()` 方法。

> *app.module.ts*

```
configure(consumer) {
  const typeDefs = this.graphQLFactory.mergeTypesByPaths('./**/*.graphql');
  const localSchema = this.graphQLFactory.createSchema({ typeDefs });
  const delegates = this.graphQLFactory.createDelegates();
  const schema = mergeSchemas({
    schemas: [localSchema, chirpSchema, linkTypeDefs],
    resolvers: delegates,
  });

  consumer
    .apply(graphqlExpress(req => ({ schema, rootValue: req })))
    .forRoutes('/graphql');
}
```

为了合并 schema ，我们使用了 `mergeSchemas()` 函数（阅读更多）。此外，您可能会注意到 `chirpsSchema` 和 `linkTypeDefs` 变量。他们是直接从 Apollo 文档复制和粘贴的。

```
import { makeExecutableSchema } from 'graphql-tools';

const chirpSchema = makeExecutableSchema({
  typeDefs: `
    type Chirp {
      id: ID!
      text: String
      authorId: ID!
```

```
    }

    type Query {
      chirpById(id: ID!): Chirp
      chirpsByAuthorId(authorId: ID!): [Chirp]
    }
  `
});
const linkTypeDefs = `
  extend type User {
    chirps: [Chirp]
  }

  extend type Chirp {
    author: User
  }
`;
```

就这样。

IDE

最受欢迎的 GraphQL 浏览器 IDE 称为 GraphiQL。要在您的应用程序中使用 GraphiQL，您需要设置一个中间件。这个特殊的中间件附带了 `apollo-server-express` ，我们必须安装。它的名字是 `graphiqlExpress()` 。

为了建立一个中间件，我们需要再次打开一个 `app.module.ts` 文件：

> *app.module.ts*

```
import { Module, MiddlewareConsumer, NestModule } from
'@nestjs/common';
import { graphqlExpress, graphiqlExpress } from 'apollo-server-
express';
import { GraphQLModule, GraphQLFactory } from '@nestjs/graphql';
```

```
4.
5.  @Module({
6.    imports: [GraphQLModule],
7.  })
8.  export class ApplicationModule implements NestModule {
9.    constructor(private readonly graphQLFactory: GraphQLFactory) {}
10.
11.   configure(consumer: MiddlewareConsumer) {
12.     const typeDefs =
    this.graphQLFactory.mergeTypesByPaths('./**/*.graphql');
13.     const schema = this.graphQLFactory.createSchema({ typeDefs });
14.
15.     consumer
16.       .apply(graphiqlExpress({ endpointURL: '/graphql' }))
17.       .forRoutes('/graphiql')
18.       .apply(graphqlExpress(req => ({ schema, rootValue: req })))
19.       .forRoutes('/graphql');
20.   }
21. }
```

?> `graphiqlExpress()` 提供了一些其他选项，请在此处阅读更多信息。

现在，当你打开 http://localhost:PORT/graphiql 你应该看到一个图形交互式 GraphiQL IDE。

WEBSOCKETS

网关

网关是用 `@WebSocketGateway()` 装饰器注解的类。默认情况下，网关使用 `socket.io`包，但也提供了与广泛的其他库的兼容性，包括本地web套接字实现（阅读更多）。

?> `提示` 网关的行为与简单的 `提供者` 相同，因此它可以毫不费力地通过构造函数注入依赖关系。另外，网关也可以由其他类（提供者和控制器）注入。

安装

首先，我们需要安装所需的软件包：

```
$ npm i --save @nestjs/websockets
```

基本

一般来说，除非你的应用程序不是Web应用程序，或者您已手动更改端口，否则每个网关都会在**HTTP**服务器运行时监听相同的端口。我们可以通过将参数传递给 `@WebSocketGateway(81)` 装饰器来改变这种行为，其中 `81` 是一个选定的端口号。另外，您可以使用以下构造来设置此网关使用的命名空间：

```
@WebSocketGateway(81, { namespace: 'events' })
```

!> 警告只有将网关放入 `提供` 程序数组中，网关才会启动。

`命名空间` 不是唯一可用的选项。您可以传递此处提及的任何其他财产。这些属性将在实例化过程中传递给套接字构造函数。好的，网关现在正在监听，但我们目前尚未订阅收到的消息。让我们创建一个处理程序，它将订阅 `事件` 消息并使用完全相同的数据响应用户。

> *events.gateway.ts*

```
@SubscribeMessage('events')
onEvent(client, data: any): WsResponse<any> {
  const event = 'events';
  return { event, data };
}
```

?> 提示 `WsResponse` 接口和 `@SubscribeMessage()` 装饰器都从 `@nestjs/common` 包中导入。

`onEvent()` 函数有2个参数。第一个是库特定的套接字实例，第二个是从客户端接收的数据。从函数返回的对象必须有2个属性。 `事件` 是发出事件的名称以及必须转发给客户端的 `数据` 。此外，可以使用特定于

库的方法发送消息，例如，通过使用 `client.emit()` 方法。但是，在这种情况下，您无法使用拦截器。如果你不想回应用户，只是不要返回任何东西（或明确返回“falsy”值，例如， `未定义` ）。

现在，当客户端以下列方式发出消息时：

```
socket.emit('events', { name: 'Nest' });
```

`onEvent()` 方法将被执行。此外，为了侦听从上述处理程序中发出的消息，客户端必须附加相应的侦听器：

```
socket.on('events', (data) => console.log(data));
```

异步响应

每个消息处理程序可以是同步的或异步的（ `异步` ），因此您可以返回 `Promise` 。此外，你可以返回RxJS `Observable` ，这意味着你可以返回多个值（它们将被发射，直到流完成）。

events.gateway.ts

```
@SubscribeMessage('events')
onEvent(client, data: any: Observable<WsResponse<number>> {
  const event = 'events';
  const response = [1, 2, 3];

  return from(response).pipe(
    map(data => ({ event, data })),
  );
}
```

上面的消息处理程序将响应3次（从 `响应` 数组中的每个项目按顺序）。

生命周期挂钩

有3个有用的生命周期挂钩。它们都有相应的接口，并在下表中进行描述：

`OnGatewayInit`	强制执行 `afterInit()` 方法。将特定于库的服务器实例作为参数
`OnGatewayConnection`	强制执行 `handleConnection()` 方法。将特定于库的客户端套接字实例作为参数。
`OnGatewayDisconnect`	强制执行 `handleDisconnect()` 方法。将特定于库的客户端套接字实例作为参数。

?>提示每个生命周期接口都来自 `@nestjs/websockets` 包。

特定库的服务器实例

偶尔，您可能希望直接访问本地 `特定库` 的服务器实例。此对象的引用作为参数传递给 `afterInit()` 方法（ `OnGatewayInit` 接口）。第二种方法是使用 `@WebSocketServer()` 装饰器。

```
1.  @WebSocketServer() server;
```

?>注意 `@WebSocketServer()` 装饰器是从 `@nestjs/websockets` 包中导入的。

当它准备好使用时，Nest会自动将服务器实例分配给该属性。

这里有一个可用的例子

异常过滤器

websockets的异常处理层工作原理与prime层完全相同。唯一的区别是不要抛出 `HttpException` ，你应该抛出 `WsException` 。

```
1.  throw new WsException('Invalid credentials.');
```

!>注意 `WsException` 类是从 `@ nestjs / websockets` 包中导入的。

Nest会处理这个异常并用下列数据发出异常消息：

```
{
  status: 'error',
  message: 'Invalid credentials.'
}
```

异常过滤器

异常过滤器也是非常类似的，并且工作方式与主过滤器完全相同。

> *ws-exception.filter.ts*

```
import { Catch, WsExceptionFilter } from '@nestjs/common';
import { WsException } from '@nestjs/websockets';

@Catch(WsException)
export class ExceptionFilter implements WsExceptionFilter {
  catch(exception: WsException, client) {
    client.emit('exception', {
      status: 'error',
      message: `It's a message from the exception filter`,
    });
  }
}
```

!>注意全局设置websockets异常过滤器是不可能的。

管道

websockets管道和普通管道没有区别。唯一应该注意的是，不要抛出 `HttpException` ，而应该使用 `WsException` 。

!>提示 `WsException` 类在 `@socketjs / websockets` 包中可用。

看守器

常规看守器和websockets看守器之间有一个区别。websockets guard将从客户端传递的 `数据` 而不是expressjs请求对象作为 `canActivate()` 函数的参数。此外，当警卫返回 `false` 时，它会抛出 `WsException` （而不是 `HttpException` ）。

!>提示 `WsException` 类在 `@socketjs / websockets` 包中可用。

拦截器

常规拦截器和websockets拦截器之间有一个区别。 Websockets拦截器将从客户端传递的 `数据` 而不是expressjs请求对象作为 `intercept()` 函数的参数。

适配器

Nest websockets模块基于socket.io，但您可以使用 `WebSocketAdapter` 接口来引入自己的库。该界面强制实施下表中描述的几种方法：

`create`	将套接字实例连接到指定的端口
`bindClientConnect`	绑定客户端连接
`bindMessageHandlers`	将传入的消息绑定到适当的消息处理程序

另外，还有两种可选的方法：

`createWithNamespace`	将套接字实例附加到指定的端口和名称空间（如果您的库支持空间）
`bindClientDisconnect`	绑定客户端断开连接事件

出于演示目的，我们将把ws库与Nest应用程序集成在一起。

> *ws-adapter.ts*

```
import * as WebSocket from 'ws';
import { WebSocketAdapter } from '@nestjs/common';
import { MessageMappingProperties } from '@nestjs/websockets';
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/observable/fromEvent';
import 'rxjs/add/observable/empty';
import 'rxjs/add/operator/switchMap';
import 'rxjs/add/operator/filter';

export class WsAdapter implements WebSocketAdapter {
  create(port: number) {
    return new WebSocket.Server({ port });
  }

  bindClientConnect(server, callback: (...args: any[]) => void) {
    server.on('connection', callback);
  }

  bindMessageHandlers(client: WebSocket, handlers:
MessageMappingProperties[], process: (data) => Observable<any>) {
    Observable.fromEvent(client, 'message')
      .switchMap((buffer) => this.bindMessageHandler(buffer,
handlers, process))
      .filter((result) => !!result)
      .subscribe((response) =>
client.send(JSON.stringify(response)));
  }

  bindMessageHandler(buffer, handlers: MessageMappingProperties[],
process: (data) => Observable<any>): Observable<any> {
    const data = JSON.parse(buffer.data);
    const messageHandler = handlers.find((handler) =>
handler.message === data.type);
    if (!messageHandler) {
      return Observable.empty();
    }
    const { callback } = messageHandler;
    return process(callback(data));
```

```
  }
}
```

由于 `WsAdapter` 类可以使用，我们可以使用 `useWebSocketAdapter()` 方法设置适配器：

> *main.ts*

```
const app = await NestFactory.create(ApplicationModule);
app.useWebSocketAdapter(new WsAdapter());
```

现在Nest会使用我们的 `WsAdapter` 而不是默认的WsAdapter。

微服务

微服务

基本

Nest 微服务是一种使用与HTTP不同的传输层的应用程序。

安装

首先，我们需要安装所需的软件包：

```
$ npm i --save @nestjs/microservices
```

概述

通常，Nest支持一系列内置的传输器。它们基于 请求-响应 范式，整个通信逻辑隐藏在抽象层之后。多亏了这一点，您可以轻松地在传输器之间切换，而无需更改任何代码行。我们不支持具有基于日志的持久性的流平台，例如 Kafka或 NATS流，因为它们是为解决不同范围的问题而创建的。但是，您仍然可以通过使用执行上下文功能将它们与Nest一起使用。

为了创建微服务，我们使用 `NestFactory` 类的 `createMicroservice()` 方法。默认情况下，微服务通过 **TCP协议** 监听消息。

> *main.ts*

```
import { NestFactory } from '@nestjs/core';
import { Transport } from '@nestjs/microservices';
import { ApplicationModule } from './app.module';

```

```
async function bootstrap() {
  const app = await NestFactory.createMicroservice(ApplicationModule, {
    transport: Transport.TCP,
  });
  app.listen(() => console.log('Microservice is listening'));
}
bootstrap();
```

`createMicroservice()` 方法的第二个参数是options对象。此对象可能有两个成员：

`transport`	指定传输器
`options`	确定传输器行为的传输器特定选项对象

`options` 对象根据所选的传送器而不同。TCP传输器暴露了下面描述的几个属性。

`host`	连接主机名
`port`	连接端口
`retryAttempts`	连接尝试的总数
`retryDelay`	连接重试延迟（ms）

模式（patterns）

微服务通过 *模式* 识别消息。模式是一个普通值，例如对象、字符串或甚至数字。为了创建模式处理程序，我们使用从 `@nestjs/microservices` 包导入的 `@MessagePattern()` 装饰器。

> *math.controller.ts*

```
import { Controller } from '@nestjs/common';
import { MessagePattern } from '@nestjs/microservices';

@Controller()
```

```
export class MathController {
  @MessagePattern({ cmd: 'sum' })
  sum(data: number[]): number {
    return (data || []).reduce((a, b) => a + b);
  }
}
```

`sum ()` 处理程序正在监听符合 `cmd :'sum'` 模式的消息。模式处理程序采用单个参数，即从客户端传递的 `data` 。在这种情况下，数据是必须累加的数字数组。

异步响应

每个模式处理程序都能够同步或异步响应。因此，支持 `async` (异步)方法。

> *math.controller.ts*

```
@MessagePattern({ cmd: 'sum' })
async sum(data: number[]): Promise<number> {
  return (data || []).reduce((a, b) => a + b);
}
```

此外，我们能够返回 RxJS `Observable` ，因此这些值将被发出，直到流完成。

> *math.controller.ts*

```
@MessagePattern({ cmd: 'sum' })
sum(data: number[]): Observable<number> {
  return from([1, 2, 3]);
}
```

以上消息处理程序将响应3次(对数组中的每个项)。

客户端

为了连接 Nest 的微服务，我们使用 `ClientProxy` 类，该类实例通过 `@Client()` 装饰器分配给属性。这个装饰器只需要一个参数。即与 Nest 微服务选项对象相同的对象。

```
@Client({ transport: Transport.TCP })
client: ClientProxy;
```

!> `@Client()` 和 `ClientProxy` 需要引入 `@nestjs/microservices` 。

`ClientProxy` 公开了一个 `send()` 方法。此方法旨在调用微服务并将其响应返回给 `Observable` ，这意味着，我们可以轻松订阅发送的值。

```
@Get()
call(): Observable<number> {
  const pattern = { cmd: 'sum' };
  const data = [1, 2, 3, 4, 5];

  return this.client.send<number>(pattern, data);
}
```

`send()` 方法接受两个参数， `pattern` 和 `data` 。 `pattern` 必须与 `@MessagePattern()` 装饰器中定义的模式相同，而 `data` 是要传输到另一个微服务的消息。

Redis

第二个内置传输器基于 Redis 数据库。此传输器利用发布/订阅功能。

安装

在开始之前，我们必须安装所需的软件包：

```
$ npm i --save redis
```

概述

为了从TCP传输策略切换到Redis **pub/sub**，我们需要更改传递给 `createMicroservice()` 方法的选项对象。

main.ts

```
const app = await NestFactory.createMicroservice(ApplicationModule, {
  transport: Transport.REDIS,
  options: {
    url: 'redis://localhost:6379',
  },
});
```

选项

有许多可用的选项可以确定传输器的行为。

`url`	连接网址
`retryAttempts`	连接尝试的总数
`retryDelay`	连接重试延迟（ms）

NATS

NATS 是一个简单、高性能的开源消息传递系统。

安装

在开始之前，我们必须安装所需的软件包：

```
$ npm i --save nats
```

概述

为了切换到 **NATS** 传输器，我们需要修改传递到 `createMicroservice()` 方法的选项对象。

main.ts

```
const app = await NestFactory.createMicroservice(ApplicationModule, {
  transport: Transport.NATS,
  options: {
    url: 'nats://localhost:4222',
  },
});
```

选项

有许多可用的选项可以确定传输器的行为。它们在 这里 有很好的描述。

gRPC

gRPC 是一个高性能、开源的通用RPC框架。

安装

在开始之前，我们必须安装所需的软件包：

```
$ npm i --save grpc
```

传输器

为了切换到 **gRPC** 传输器，我们需要修改传递到 `createMicroservice()` 方法的选项对象。

main.ts

```
const app = await NestFactory.createMicroservice(ApplicationModule, {
  transport: Transport.GRPC,
  options: {
    package: 'hero',
    protoPath: join(__dirname, './hero/hero.proto'),
  },
});
```

!> `join()` 函数是从 `path` 包导入的。

选项

`url`	连接网址
`protoPath`	`.proto` 文件的绝对路径
`package`	`protobuf` 包名
`credentials`	服务器证书(阅读更多)

概述

通常，`package` 属性设置 `protobuf` 包名称，而 `protoPath` 是 `.proto` 文件的路径。`hero.proto` 文件是使用协议缓冲区语言构建的。

hero.proto

```
syntax = "proto3";

package hero;

service HeroService {
  rpc FindOne (HeroById) returns (Hero) {}
}

message HeroById {
  int32 id = 1;
}

message Hero {
  int32 id = 1;
  string name = 2;
}
```

在上面的示例中，我们定义了一个 `HeroService` ，它暴露了一个 `FindOne()` gRPC处理程序，该处理程序期望 `HeroById` 作为输入并返回一个 `Hero` 消息。为了定义一个能够实现这个protobuf定义的处理程序，我们必须使用 `@GrpcRoute()` 装饰器。之前的 `@MessagePattern()` 将不再有用。

> *hero.controller.ts*

```
@GrpcRoute('HeroService', 'FindOne')
findOne(data: HeroById, metadata: any): Hero {
  const items = [{ id: 1, name: 'John' }, { id: 2, name: 'Doe' }];
  return items.find(({ id }) => id === data.id);
}
```

!> `@GrpcRoute()` 需要引入 `@nestjs/microservices` 。

`HeroService` 是服务的名称，而 `FindOne` 指向 `FindOne()` gRPC处理程序。对应的 `findOne()` 方法接受两个参数，即从调用方传递

的 `data` 和存储gRPC请求元数据的 `metadata` 。

客户端

为了创建客户端实例，我们需要使用 `@Client()` 装饰器。

> *hero.controller.ts*

```
@Client({
  transport: Transport.GRPC,
  options: {
    package: 'hero',
    protoPath: join(__dirname, './hero/hero.proto'),
  },
})
private readonly client: ClientGrpc;
```

与前面的例子相比有一点差别。我们使用提供 `getService()` 方法的 `ClientGrpc` ，而不是 `ClientProxy` 类。 `getService()` 泛型方法将服务的名称作为参数，并返回其实例(如果可用)。

> *hero.controller.ts*

```
onModuleInit() {
  this.heroService = this.client.getService<HeroService>
('HeroService');
}
```

`heroService` 对象暴露了 `.proto` 文件中定义的同一组方法。注意，所有这些都是 小写 (为了遵循自然惯例)。基本上，我们的gRPC `HeroService` 定义包含 `FindOne()` 函数。这意味着 `heroService` 实例将提供 `findOne()` 方法。

```
interface HeroService {
  findOne(data: { id: number }): Observable<any>;
```

```
3. }
```

所有服务的方法都返回 `Observable` 。由于 Nest 支持 RxJS 流并且与它们很好地协作，所以我们也可以在HTTP处理程序中返回它们。

> *hero.controller.ts*

```
1. @Get()
2. call(): Observable<any> {
3.   return this.heroService.findOne({ id: 1 });
4. }
```

这里 提供了一个完整的示例。

异常过滤器 (Exception filters)

HTTP异常过滤器层和相应的微服务层之间的唯一区别在于，不要抛出 `HttpException` ，而应该使用 `RpcException` 。

?> `RpcException` 类是从 `@nestjs/microservices` 包导入的。

Nest将处理引发的异常，并因此返回具有以下结构的 `error` 对象：

```
1. {
2.   status: 'error',
3.   message: 'Invalid credentials.'
4. }
```

过滤器

异常过滤器 的工作方式与主过滤器相同，只有一个小的区别。 `catch()` 方法必须返回一个 `Observable` 。

> *rpc-exception.filter.ts*

```
import { Catch, RpcExceptionFilter, ArgumentsHost } from '@nestjs/common';
import { Observable, throwError } from 'rxjs';
import { RpcException } from '@nestjs/microservices';

@Catch(RpcException)
export class ExceptionFilter implements RpcExceptionFilter {
  catch(exception: RpcException, host: ArgumentsHost): Observable<any> {
    return throwError(exception.getError());
  }
}
```

!> 不能设置全局的微服务异常过滤器。

下面是一个使用手动实例化 方法作用域 过滤器(也可以使用类作用域)的示例:

```
@UseFilters(new ExceptionFilter())
@MessagePattern({ cmd: 'sum' })
sum(data: number[]): number {
  return (data || []).reduce((a, b) => a + b);
}
```

管道 (Pipes)

微服务管道和普通管道没有区别。唯一需要注意的是，不要抛出 `HttpException` ，而应该使用 `RpcException` 。

?> `RpcException` 类是从 `@nestjs/microservices` 包导入的。

下面是一个使用手动实例化 方法作用域 管道(也可以使用类作用域)的示例:

```
@UsePipes(new ValidationPipe())
```

```
@MessagePattern({ cmd: 'sum' })
sum(data: number[]): number {
  return (data || []).reduce((a, b) => a + b);
}
```

看守器（Guards）

微服看守器和普通看守器没有区别。唯一需要注意的是，不要抛出 `HttpException` ，而应该使用 `RpcException` 。

?> `RpcException` 类是从 `@nestjs/microservices` 包导入的。

下面是一个使用 方法作用域 看守器(也可以使用类作用域)的示例：

```
@UseGuards(AuthGuard)
@MessagePattern({ cmd: 'sum' })
sum(data: number[]): number {
  return (data || []).reduce((a, b) => a + b);
}
```

拦截器（Interceptors）

常规拦截器和微服务拦截器之间没有区别。下面是一个使用手动实例化方法作用域 拦截器(也可以使用类作用域)的示例：

```
@UseInterceptors(new TransformInterceptor())
@MessagePattern({ cmd: 'sum' })
sum(data: number[]): number {
  return (data || []).reduce((a, b) => a + b);
}
```

执行上下文

执行上下文

有几种安装 Nest 应用程序的方法。您可以创建一个 Web 应用程序，微服务或只是一个 Nest 执行上下文 。 Nest 上下文是 Nest 容器的一个包装，它包含所有实例化的类。我们可以直接使用应用程序对象从任何导入的模块中获取现有实例。由于这一点，您可以充分利用 Nest 框架的优势，包括 **CRON** 任务，甚至可以在其上构建 **CLI** 。

为了创建一个 Nest 应用程序上下文，我们使用下面的语法：

```
async function bootstrap() {
  const app = await NestFactory.createApplicationContext(ApplicationModule);
  // logic...
}
bootstrap();
```

之后，Nest 允许您选择在 Nest 应用程序中注册的任何实例。假设我们在 `TasksModule` 中有一个 `TasksController` 。这个类提供了一组我们想从 CRON 任务中调用可用的方法。

```
const app = await NestFactory.create(ApplicationModule);
const tasksController = app.get(TasksController);
```

就是这样。要获取 `TasksController` 实例，我们必须使用 `get()` 方法。我们不必遍历整个模块树， `get()` 方法就像 查询 一样，自动在每个注册模块中搜索实例。但是，如果您更喜欢严格的上下文检查，则可以使用 `strict: true` 选项对象作为 `get()` 方法的第二个参

数传递给它。然后，您必须通过所有模块从选定的上下文中选取特定的实例。

```
const app = await NestFactory.create(ApplicationModule);
const tasksController =
app.select(TasksModule).get(TasksController, { strict: true });
```

`get()`	检索应用程序上下文中可用的控制器或提供者（包括看守器，过滤器等）的实例。
`select()`	浏览模块树，例如，从所选模块中提取特定实例（与启用严格模式一起使用）。

?> 默认情况下，根模块处于选中状态。要选择任何其他模块，您需要遍历整个模块树(逐步)。

秘籍

秘籍

SQL（TypeORM）

!> 在本文中，您将学习如何使用自定义提供者机制从零开始创建基于 **TypeORM** 包的 `DatabaseModule` 。由于该解决方案包含许多开销，因此您可以使用开箱即用的 `@nestjs/typeorm` 软件包。要了解更多信息，请参阅 此处。

TypeORM 无疑是 node.js 世界中最成熟的对象关系映射器（ORM）。由于它是用 TypeScript 编写的，所以它在 Nest 框架下运行得非常好。在开始使用这个库前，我们必须安装所有必需的依赖关系：

```
$ npm install --save typeorm mysql
```

我们需要做的第一步是使用从 `typeorm` 包导入的 `createConnection()` 函数建立与数据库的连接。`createConnection()`

函数返回一个 `Promise` ，因此我们必须创建一个异步提供者。

> *database.providers.ts*

```
import { createConnection } from 'typeorm';

export const databaseProviders = [
  {
    provide: 'DbConnectionToken',
    useFactory: async () => await createConnection({
      type: 'mysql',
      host: 'localhost',
      port: 3306,
      username: 'root',
      password: 'root',
      database: 'test',
      entities: [
          __dirname + '/../**/*.entity{.ts,.js}',
      ],
      synchronize: true,
    }),
  },
];
```

?> 按照最佳实践，我们在分离的文件中声明了自定义提供者，该文件带有 `*.providers.ts` 后缀。

然后，我们需要导出这些提供者，以便应用程序的其余部分可以 访问它们。

> *database.module.ts*

```
import { Module } from '@nestjs/common';
import { databaseProviders } from './database.providers';

@Module({
  providers: [...databaseProviders],
  exports: [...databaseProviders],
```

```
})
export class DatabaseModule {}
```

现在我们可以使用 `@Inject()` 装饰器注入 `Connection` 对象。依赖于 `Connection` 异步提供者的每个类都将等待解析 `Promise` 。

存储库模式

TypeORM 支持存储库设计模式，因此每个实体都有自己的存储库。这些资料库可以从数据库连接中获取。

但首先，我们至少需要一个实体。我们将重用官方文档中的 `Photo` 实体。

> *photo/photo.entity.ts*

```
import { Entity, Column, PrimaryGeneratedColumn } from 'typeorm';

@Entity()
export class Photo {
  @PrimaryGeneratedColumn()
  id: number;

  @Column({ length: 500 })
  name: string;

  @Column('text')
  description: string;

  @Column()
  filename: string;

  @Column('int')
  views: number;

  @Column()
  isPublished: boolean;
```

```
22. }
```

`Photo` 实体属于 `photo` 目录。此目录代表 `PhotoModule` 。现在，让我们创建一个 存储库 提供者：

> *photo.providers.ts*

```
1.  import { Connection, Repository } from 'typeorm';
2.  import { Photo } from './photo.entity';
3.
4.  export const photoProviders = [
5.    {
6.      provide: 'PhotoRepositoryToken',
7.      useFactory: (connection: Connection) =>
    connection.getRepository(Photo),
8.      inject: ['DbConnectionToken'],
9.    },
10. ];
```

!> 请注意，在实际应用程序中，您应该避免使用魔术字符串。 `PhotoRepositoryToken` 和 `DbConnectionToken` 都应保存在分离的 `constants.ts` 文件中。

现在我们可以使用 `@Inject()` 装饰器将 `PhotoRepository` 注入到 `PhotoService` 中：

> *photo.service.ts*

```
1.  import { Injectable, Inject } from '@nestjs/common';
2.  import { Repository } from 'typeorm';
3.  import { Photo } from './photo.entity';
4.
5.  @Injectable()
6.  export class PhotoService {
7.    constructor(
8.      @Inject('PhotoRepositoryToken')
9.      private readonly photoRepository: Repository<Photo>,
```

```
  ) {}

  async findAll(): Promise<Photo[]> {
    return await this.photoRepository.find();
  }
}
```

数据库连接是 异步 的，但 Nest 使最终用户完全看不到这个过程。`PhotoRepository` 正在等待数据库连接时，`PhotoService` 被延迟，直到存储库准备好使用。整个应用程序可以在每个类实例化时启动。

这是一个最终的 `PhotoModule` ：

> *photo.module.ts*

```
import { Module } from '@nestjs/common';
import { DatabaseModule } from '../database/database.module';
import { photoProviders } from './photo.providers';
import { PhotoService } from './photo.service';

@Module({
  imports: [DatabaseModule],
  providers: [
    ...photoProviders,
    PhotoService,
  ],
})
export class PhotoModule {}
```

!> 不要忘记将 `PhotoModule` 导入到根 `ApplicationModule` 中。

CQRS

最简单的 **CRUD** 应用程序的流程可以使用以下步骤来描述：

1. 控制器层处理HTTP请求并将任务委派给服务。
2. 服务层是正在执行大部分业务逻辑的地方。
3. 服务使用存储库或 DAOs 来更改/保留实体。
4. 实体是我们的模型 - 只有容器的值，setters 和 getters 。

这就是为什么 Nest 提供了一个轻量级的 CQRS 模块，这些模块的组件在下面有详细描述。

Commands

为了使应用程序更易于理解，每个更改都必须以 **Command** 开头。当任何命令被分派时，应用程序必须对其作出反应。命令可以从服务中分派并在相应的 **Command** 处理程序 中使用。

> *heroes-game.service.ts*

```
@Injectable()
export class HeroesGameService {
  constructor(private readonly commandBus: CommandBus) {}

  async killDragon(heroId: string, killDragonDto: KillDragonDto) {
    return await this.commandBus.execute(
      new KillDragonCommand(heroId, killDragonDto.dragonId)
    );
  }
}
```

这里有一个示例服务，它调度 `KillDragonCommand` 。让我们来看看这个命令：

> *kill-dragon.command.ts*

```
export class KillDragonCommand implements ICommand {
  constructor(
    public readonly heroId: string,
```

```
    public readonly dragonId: string,
  ) {}
}
```

这个 `CommandBus` 是一个命令 「流」 。它将命令委托给等效的处理程序。每个命令必须有相应的命令处理程序：

> *kill-dragon.handler.ts*

```
@CommandHandler(KillDragonCommand)
export class KillDragonHandler implements ICommandHandler<KillDragonCommand> {
  constructor(private readonly repository: HeroRepository) {}

  async execute(command: KillDragonCommand, resolve: (value?) => void) {
    const { heroId, dragonId } = command;
    const hero = this.repository.findOneById(+heroId);

    hero.killEnemy(dragonId);
    await this.repository.persist(hero);
    resolve();
  }
}
```

现在，每个应用程序状态更改都是 **Command** 发生的结果。逻辑被封装在处理程序中。我们可以简单地在这里添加日志，甚至我们可以在数据库中保留我们的命令（例如，用于诊断目的）。

为什么我们需要 `resolve()` 函数？有时，我们可能希望从处理程序返回消息到服务。此外，我们可以在 `execute()` 方法的开头调用此函数，因此应用程序首先返回到服务中，然后将响应反馈给客户端，然后 异步 返回到这里处理发送的命令。

事件（Events）

由于我们在处理程序中封装了命令，所以我们阻止了它们之间的交互-应用程序结构仍然不灵活，不具有响应性。解决方案是使用事件。

> *hero-killed-dragon.event.ts*

```
export class HeroKilledDragonEvent implements IEvent {
  constructor(
    public readonly heroId: string,
    public readonly dragonId: string) {}
}
```

事件是异步的。他们由模型调用。模型必须扩展这个 `AggregateRoot` 类。

> *hero.model.ts*

```
export class Hero extends AggregateRoot {
  constructor(private readonly id: string) {
    super();
  }

  killEnemy(enemyId: string) {
    // logic
    this.apply(new HeroKilledDragonEvent(this.id, enemyId));
  }
}
```

`apply()` 方法尚未发送事件，因为模型和 `EventPublisher` 类之间没有关系。如何辨别 `publisher` 的模型？ 我们需要在我们的命令处理程序中使用一个 publisher `mergeObjectContext()` 方法。

> *kill-dragon.handler.ts*

```
@CommandHandler(KillDragonCommand)
export class KillDragonHandler implements
ICommandHandler<KillDragonCommand> {
  constructor(
```

```
    private readonly repository: HeroRepository,
    private readonly publisher: EventPublisher,
  ) {}

  async execute(command: KillDragonCommand, resolve: (value?) => void) {
    const { heroId, dragonId } = command;
    const hero = this.publisher.mergeObjectContext(
      await this.repository.findOneById(+heroId),
    );
    hero.killEnemy(dragonId);
    hero.commit();
    resolve();
  }
}
```

现在，一切都按我们预期的方式工作。注意，我们需要 commit() 事件，因为他们没有立即调用。当然，一个对象不一定已经存在。我们也可以轻松地合并类型上下文：

```
const HeroModel = this.publisher.mergeContext(Hero);
new HeroModel('id');
```

就是这样。模型现在能够发布事件。我们得处理他们。

每个事件都可以有许多事件处理程序。他们不必知道对方。

> *hero-killed-dragon.handler.ts*

```
@EventsHandler(HeroKilledDragonEvent)
export class HeroKilledDragonHandler implements IEventHandler<HeroKilledDragonEvent> {
  constructor(private readonly repository: HeroRepository) {}

  handle(event: HeroKilledDragonEvent) {
    // logic
  }
```

```
8. }
```

现在，我们可以将写入逻辑移动到事件处理程序中。

Sagas

这种类型的 事件驱动架构 可以提高应用程序的 反应性 和 可伸缩性 。现在， 当我们有了事件， 我们可以简单地以各种方式对他们作出反应。**sagas** 是从建筑学的观点来看的最后积木。

sagas 是一个非常强大的功能。单身传奇可以监听听 1.. 事件。它可以组合，合并，过滤事件流。*RxJS* 库是魔术的来源地。简单地说，每个 *sagas* 都必须返回一个包含命令的*Observable*。此命令是 * 异步 调用的。

> *heroes-game.saga.ts*

```
1. @Component()
2. export class HeroesGameSagas {
3.   dragonKilled = (events$: EventObservable<any>):
   Observable<ICommand> => {
4.     return events$.ofType(HeroKilledDragonEvent)
5.       .map((event) => new DropAncientItemCommand(event.heroId,
   fakeItemID));
6.   }
7. }
```

我们宣布一个规则，当任何英雄杀死龙-它应该得到古老的项目。然后， `DropAncientItemCommand` 将被适当的处理程序调度和处理。

建立（Setup）

最后一件事， 我们要处理的是建立整个机制。

> *heroes-game.module.ts*

```
export const CommandHandlers = [KillDragonHandler, DropAncientItemHandler];
export const EventHandlers =  [HeroKilledDragonHandler, HeroFoundItemHandler];

@Module({
  imports: [CQRSModule],
  controllers: [HeroesGameController],
  providers: [
    HeroesGameService,
    HeroesGameSagas,
    ...CommandHandlers,
    ...EventHandlers,
    HeroRepository,
  ]
})
export class HeroesGameModule implements OnModuleInit {
  constructor(
    private readonly moduleRef: ModuleRef,
    private readonly command$: CommandBus,
    private readonly event$: EventBus,
    private readonly heroesGameSagas: HeroesGameSagas,
  ) {}

  onModuleInit() {
    this.command$.setModuleRef(this.moduleRef);
    this.event$.setModuleRef(this.moduleRef);

    this.event$.register(EventHandlers);
    this.command$.register(CommandHandlers);
    this.event$.combineSagas([
      this.heroesGameSagas.dragonKilled,
    ]);
  }
}
```

概要

`CommandBus` 和 `EventBus` 都是 **Observables** 。这意味着您可以轻松地订阅整个「流」，并通过 **Event Sourcing** 丰富您的应用程序。

完整的源代码在 这里 可用。

CLI

CLI

概述

为了帮助用户管理他们的项目，已经创建了 CLI 工具。它同时在很多方面都有帮助，从搭建项目到构建结构良好的应用程序。嵌套CLI基于 @ angular - devkit 软件包。此外，还有专门用于嵌套开发 @ nestjs / schematics 的特殊示意图

安装

使用 **NPM** 安装 CLI:

```
1.  $ npm install -g @nestjs/cli
```

使用 **Docker Hub** 安装CLI:

```
1.  $ docker pull nestjs/cli:[version]
```

更多详细信息可在 Docker Hub 中找到。

使用

为了提高用户体验，CLI命令共享相同的命令架构

```
$ nest [command] [...options]
```

选项

每个命令都接受下面列出的一组选项：

- **–dry-run:** 允许模拟命令执行，以验证它将如何影响您的工作目录

new (alias: n)

new 命令生成基于 typescript-starter上的Nest项目以及安装所需的软件包。CLI会询问您是否缺少创建项目的信息，例如，您想使用哪个程序包管理器来安装依赖项。

Option	Description	Required	Default value
name	你的应用名称	false	nest-app-name
description	你的应用程序描述	false	description
version	你的应用版本	false	1.0.0
author	您的应用作者	false	‘’

示例用法：

```
$ nest new my-awesome-app
OR
$ nest n my-awesome-app
```

generate (alias: g)

generate 命令生成嵌套体系结构组件。

Option	Description	Required	Default value

schematic	下面列表中的示意图名称。	true	N/A
name	generateNest架构组件的名称	true	N/A
path	generateNest架构组件的路径	false	src

可用体系结构组件的列表:

- class (alias: cl)
- controller (alias: co)
- decorator (alias: d)
- exception (alias: e)
- filter (alias: f)
- gateway (alias: ga)
- guard (alias: gu)
- interceptor (alias: i)
- middleware (alias: mi)
- module (alias: mo)
- pipe (alias: pi)
- provider (alias: pr)
- service (alias: s)

  示例用法:

```
$ nest generate service users
$ nest g s users
```

info (alias: i)

info 命令将显示您的项目信息

```
$ nest info
 _   _             _      ___  _____  _____  _     _____
| \ | |           | |    |_  |/  ___|/  __ \| |   |_   _|
|  \| |  ___  ___ | |_     | |\ `--. | /  \/| |     | |
| . ` | / _ \/ __|| __|    | | `--. \| |    | |     | |
| |\  ||  __/\__ \| |_ /\__/ //\__/ /| \__/\| |_____| |_
\_| \_/ \___||___/ \__|\____/ \____/  \____/\_____/\___/

[System Information]
OS Version     : macOS High Sierra
NodeJS Version : v8.9.0
YARN Version    : 1.5.1
[Nest Information]
microservices version : 5.0.0
websockets version    : 5.0.0
testing version       : 5.0.0
common version        : 5.0.0
core version          : 5.0.0
```

FAQ

FAQ

Express 实例

有时，您可能希望完全控制 express 实例生命周期。这很容易，因为 NestFactory.create() 方法将 express 实例作为第二个参数。

```
1. const server = express();
2. const app = await NestFactory.create(ApplicationModule, server);
```

全局路由前缀

要为应用程序中的每个路由设置前缀，让我们使用 INestApplication 对象的 setGlobalPrefix() 方法。

```
1. const app = await NestFactory.create(ApplicationModule);
2. app.setGlobalPrefix('v1');
```

生命周期事件

有2个模块生命周期事件，`OnModuleInit` 和 `OnModuleDestroy`。将它们用于所有初始化的东西并避免将任何东西直接放入构造函数是一种很好的做法。构造函数只能用于初始化类成员并注入所需的依赖项。

```
import { Injectable, OnModuleInit, OnModuleDestroy } from
'@nestjs/common';

@Injectable()
export class UsersService implements OnModuleInit, OnModuleDestroy
{
  onModuleInit() {
    console.log(`Initialization...`);
  }

  onModuleDestroy() {
    console.log(`Cleanup...`);
  }
}
```

混合应用

混合应用程序的应用程序与连接的 `microservice/s`。可以将 `INestApplication` 与 `INestMicroservice` 实例的无限计数结合起来。

```
const app = await NestFactory.create(ApplicationModule);
const microservice = app.connectMicroservice({
    transport: Transport.TCP,
});

await app.startAllMicroservicesAsync();
await app.listen(3001);
```

HTTPS 和多服务器

由于您可以完全控制 express 实例生命周期，因此创建几个同时运行的多个服务器（例如，HTT P和 HTTPS）并不难。

```
const httpsOptions = {
  key: fs.readFileSync("./secrets/private-key.pem"),
  cert: fs.readFileSync("./secrets/public-certificate.pem")
};

const server = express();
const app = await NestFactory.create(ApplicationModule, server);
await app.init();

http.createServer(server).listen(3000);
https.createServer(httpsOptions, server).listen(443)
```

样例

更多例子参考

迁移指南

迁移指南

本文提供了一套从v4迁移到最新v5版本的指导原则。在开发过程中，我们花了很多时间试图避免任何重大改变。尽管如此，为了简化它的使用，API必须在一堆地方进行更改。此外，以前的版本由于已经做出的决定而受到限制。

模板

为了减少Nest和Angular之间的差异数量，根据 `@Module()` 装饰器进行了很少的更改。

- `模块` 属性现在已被弃用，改用 `导入`
- `组件` 属性现在已被弃用，改为使用 `提供者`

装饰器

`@Component()` ， `@Middleware()` ， `@Interceptor()` ， `@Pipe()` 和 `@Gua`

`rd()` 现在已弃用。使用 `@Injectable()` 来代替。

中间件

我们不再完全支持传统的快递中间件模型，也就是说，无论请求方式如何，每个中间件现在都只限于一条路径。而且，不会有更多的错误中间件。这一变化有助于我们在可移植性和兼容性之间找到中间立场。

```
// Before
consumer.apply(LoggerMiddleware).forRoutes(
  { path: '/cats', method: RequestMethod.GET },
  { path: '/cats', method: RequestMethod.POST },
);

// After
consumer.apply(LoggerMiddleware).forRoutes('cats');
```

但是，如果您仍然需要将中间件绑定到特定的请求方法，则可以直接使用快速实例。

```
const app = await NestFactory.create(ApplicationModule);
app.get('cats', logger);
```

但推荐的方法是使用管道，警卫或拦截器。

消费者

`MiddlewaresConsumer` 类已更改为 `MiddlewareConsumer` 。

过滤器

异常过滤器不再作为单一范式被锁定。以前，异常过滤器可以访问 `响应` 对象。与传入的发行版一起， `catch()` 方法取而代之使用 `ArgumentsHost` 实例。

```
// Before
@Catch(HttpException)
export class HttpExceptionFilter implements ExceptionFilter {
  catch(exception: HttpException, response) {}
}

// After
@Catch(HttpException)
export class HttpExceptionFilter implements ExceptionFilter {
  catch(exception: HttpException, host: ArgumentsHost) {
    const response = host.switchToHttp().getResponse();
    // ...
  }
}
```

看守器

与过滤器一样，看守器现在更加灵活。访问增强的 `ExecutionContext` 让守卫更加超级强大，并且所有这些都是在简化的API基础上构建的。

```
// Before
@Guard()
export class RolesGuard implements CanActivate {
  canActivate(
    dataOrRequest,
    context: ExecutionContext,
  ): boolean | Promise<boolean> | Observable<boolean> {
    return true;
  }
}

// After
@Injectable()
export class RolesGuard implements CanActivate {
  canActivate(
    context: ExecutionContext,
  ): boolean | Promise<boolean> | Observable<boolean> {
```

```
    // const request = context.switchToHttp().getRequest();
    return true;
  }
}
```

拦截器

拦截器API的演变方式与等效的看守器API完全相同。

```
// Before
@Interceptor()
export class TransformInterceptor implements NestInterceptor {
  intercept(
    dataOrRequest,
    context: ExecutionContext,
    stream$: Observable<any>,
  ): Observable<any> {
    return stream$.map((data) => ({ data }));
  }
}

// After
@Injectable()
export class TransformInterceptor implements NestInterceptor {
  intercept(
    context: ExecutionContext,
    call$: Observable<T>,
  ): Observable<Response<T>> {
    // const request = context.switchToHttp().getRequest();
    return call$.pipe(map(data => ({ data })));
  }
}
```

自定义装饰器

`createRouteParamDecorator()` 函数现在已被弃用。现在使

用 `createParamDecorator()` 。