自定义提供者

在前面几章中，我们讨论了依赖注入(DI)的各个方面，以及如何在 Nest 中使用它。其中一个例子是基于构造函数的依赖注入，用于将实例(通常是服务提供者)注入到类中。当您了解到依赖注入是以一种基本的方式构建到 Nest 内核中时，您不会感到惊讶。到目前为止，我们只探索了一个主要模式。随着应用程序变得越来越复杂，您可能需要利用 DI 系统的所有特性，因此让我们更详细地研究它们。

依赖注入

依赖注入是一种控制反转（ IoC ）技术，您可以将依赖的实例化委派给 IoC 容器（在我们的示例中为 NestJS 运行时系统），而不是必须在自己的代码中执行。 让我们从"提供者"一章中检查此示例中发生的情况。

首先，我们定义一个提供者。 @Injectable() 装饰器将 CatsService 类标记为提供者。

> cats.service.ts

```typescript
import { Injectable } from '@nestjs/common';
import { Cat } from './interfaces/cat.interface';

@Injectable()
export class CatsService {
  private readonly cats: Cat[] = [];

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

然后，我们要求 Nest 将提供程序注入到我们的控制器类中：

> cats.controller.ts

```typescript
import { Controller, Get } from '@nestjs/common';
import { CatsService } from './cats.service';
import { Cat } from './interfaces/cat.interface';

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

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

最后，我们在 Nest IoC 容器中注册提供程序

> app.module.ts

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

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

这个过程有三个关键步骤:

1. 在 cats.service.ts 中 @Injectable() 装饰器声明 CatsService 类是一个可以由 Nest IoC 容器管理的类。

2. 在 cats.controller.ts 中 CatsController 声明了一个依赖于 CatsService 令牌(token)的构造函数注入:

```typescript
constructor(private readonly catsService: CatsService)
```

3. 在 app.module.ts 中，我们将标记 CatsService 与 cats.service.ts 文件中的 CatsService 类相关联。我们将在下面确切地看到这种关联（也称为注册）的发生方式。

当 Nest IoC 容器实例化 CatsController 时，它首先查找所有依赖项*。当找到 CatsService 依赖项时，它将对 CatsService 令牌(token)执行查找，并根据上述步骤（上面的＃3）返回 CatsService 类。假定单例范围（默认行为），Nest 然后将创建 CatsService 实例，将其缓存并返回，或者如果已经缓存，则返回现有实例。

这个解释稍微简化了一点。我们忽略的一个重要方面是，分析依赖项代码的过程非常复杂，并且发生在应用程序引导期间。一个关键特性是依赖关系分析(或“创建依赖关系图”)是可传递的。在上面的示例中，如果 CatsService 本身具有依赖项，那么那些依赖项也将得到解决。依赖关系图确保以正确的顺序解决依赖关系-本质上是“自下而上”。这种机制使开发人员不必管理此类复杂的依赖关系图。

标准提供者

让我们仔细看看 @Module() 装饰器。在中 app.module ，我们声明:

```typescript
@Module({
  controllers: [CatsController],
  providers: [CatsService],
})
```

providers属性接受一个提供者数组。到目前为止，我们已经通过一个类名列表提供了这些提供者。实际上，该语法 providers: [CatsService] 是更完整语法的简写:

```typescript
providers: [
  {
    provide: CatsService,
```

```
    useClass: CatsService,
  },
];
```

现在我们看到了这个显式的构造，我们可以理解注册过程。在这里，我们明确地将令牌 `CatsService` 与类 `CatsService` 关联起来。简写表示法只是为了简化最常见的用例，其中令牌用于请求同名类的实例。

自定义提供者

当您的要求超出标准提供商所提供的要求时，会发生什么？这里有一些例子：

- 您要创建自定义实例，而不是让 `Nest` 实例化（或返回其缓存实例）类
- 您想在第二个依赖项中重用现有的类
- 您想使用模拟版本覆盖类进行测试

`Nest` 可让您定义自定义提供程序来处理这些情况。它提供了几种定义自定义提供程序的方法。让我们来看看它们。

值提供者 (useValue)

`useValue` 语法对于注入常量值、将外部库放入 `Nest` 容器或使用模拟对象替换实际实现非常有用。假设您希望强制 `Nest` 使用模拟 `CatsService` 进行测试。

typescript

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

const mockCatsService = {
  /* mock implementation
  ...
  */
};

@Module({
  imports: [CatsModule],
  providers: [
    {
      provide: CatsService,
      useValue: mockCatsService,
    },
```

```
  ],
})
export class AppModule {}
```

在本例中， CatsService 令牌将解析为 mockCatsService 模拟对象。 useValue 需要一个值——在本例中是一个文字对象，它与要替换的 CatsService 类具有相同的接口。由于 TypeScript 的结构类型化，您可以使用任何具有兼容接口的对象，包括文本对象或用 new 实例化的类实例。

到目前为止，我们已经使用了类名作为我们的提供者标记（ providers 数组中列出的提供者中的 Provide 属性的值）。 这与基于构造函数的注入所使用的标准模式相匹配，其中令牌也是类名。（如果此概念尚不完全清楚，请参阅DI基础知识，以重新学习令牌）。 有时，我们可能希望灵活使用字符串或符号作为 DI 令牌。 例如：

typescript

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

@Module({
  providers: [
    {
      provide: 'CONNECTION',
      useValue: connection,
    },
  ],
})
export class AppModule {}
```

在本例中，我们将字符串值令牌('CONNECTION')与从外部文件导入的已存在的连接对象相关联。

> 除了使用字符串作为令牌之外，还可以使用 JavaScript 符号。

我们前面已经看到了如何使用基于标准构造函数的注入模式注入提供者。此模式要求用类名声明依赖项。 'CONNECTION' 自定义提供程序使用字符串值令牌。让我们看看如何注入这样的提供者。为此，我们使用 @Inject() 装饰器。这个装饰器只接受一个参数——令牌。

typescript

```
@Injectable()
export class CatsRepository {
```

```
    constructor(@Inject('CONNECTION') connection: Connection) {}
  }
```

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

虽然我们在上面的例子中直接使用字符串 `'CONNECTION'` 来进行说明，但是为了清晰的代码组织，最佳实践是在单独的文件（例如 `constants.ts` ）中定义标记。 对待它们就像对待在其自己的文件中定义并在需要时导入的符号或枚举一样。

类提供者 (useClass)

`useClass` 语法允许您动态确定令牌应解析为的类。 例如，假设我们有一个抽象（或默认）的 `ConfigService` 类。 根据当前环境，我们希望 `Nest 提供配置服务的不同实现。 以下代码实现了这种策略。

typescript

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

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

让我们看一下此代码示例中的一些细节。 您会注意到，我们首先定义对象 `configServiceProvider` ，然后将其传递给模块装饰器的 `providers` 属性。 这只是一些代码组织，但是在功能上等同于我们到目前为止在本章中使用的示例。

另外，我们使用 `ConfigService` 类名称作为令牌。 对于任何依赖 `ConfigService` 的类， `Nest` 都会注入提供的类的实例（ `DevelopmentConfigService` 或 `ProductionConfigService` ），该实例将覆盖在其他地方已声明的任何默认实现（例如，使用 `@Injectable()` 装饰器声明的 `ConfigService` ）。

工厂提供者 (useFactory)

`useFactory` 语法允许动态创建提供程序。实工厂函数的返回实际的 `provider` 。工厂功能可以根据需要简单或复杂。一个简单的工厂可能不依赖于任何其他的提供者。更复杂的工厂可以自己注入它需要的其他提供者来计算结果。对于后一种情况，工厂提供程序语法有一对相关的机制:

1. 工厂函数可以接受(可选)参数。
2. `inject` 属性接受一个提供者数组，在实例化过程中， `Nest` 将解析该数组并将其作为参数传递给工厂函数。这两个列表应该是相关的: `Nest` 将从 `inject` 列表中以相同的顺序将实例作为参数传递给工厂函数。

下面示例演示:

typescript

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

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

别名提供者 (useExisting)

`useExisting` 语法允许您为现有的提供程序创建别名。这将创建两种访问同一提供者的方法。在下面的示例中，(基于 `string`)令牌 `'AliasedLoggerService'` 是(基于类的)令牌 `LoggerService` 的别名。假设我们有两个不同的依赖项，一个用于 `'AlilasedLoggerService'` ，另一个用于 `LoggerService` 。如果两个依赖项都用单例作用域指定，它们将解析为同一个实例。

typescript

```
@Injectable()
class LoggerService {
  /* implementation details */
```

```
}

const loggerAliasProvider = {
  provide: 'AliasedLoggerService',
  useExisting: LoggerService,
};

@Module({
  providers: [LoggerService, loggerAliasProvider],
})
export class AppModule {}
```

非服务提供者

虽然提供者经常提供服务，但他们并不限于这种用途。提供者可以提供任何值。例如，提供程序可以根据当前环境提供配置对象数组，如下所示:

typescript

```
const configFactory = {
  provide: 'CONFIG',
  useFactory: () => {
    return process.env.NODE_ENV === 'development'
      ? devConfig
      : prodConfig;
  },
};

@Module({
  providers: [configFactory],
})
export class AppModule {}
```

导出自定义提供者

与任何提供程序一样，自定义提供程序的作用域仅限于其声明模块。要使它对其他模块可见，必须导出它。要导出自定义提供程序，我们可以使用其令牌或完整的提供程序对象。

以下示例显示了使用 `token` 的例子:

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

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

但是你也可以使用整个对象：

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

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

异步提供者

在完成一些异步任务之前，应用程序必须等待启动状态, 例如，在与数据库的连接建立之前，您可能不希望开始接受请求。 在这种情况下你应该考虑使用异步 `provider` 。

其语法是使用 `useFactory` 语法的 `async/await` 。工厂返回一个承诺，工厂函数可以等待异步任务。在实例化依赖于(注入)这样一个提供程序的任何类之前， `Nest` 将等待承诺的解决。

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

> 在这里了解 更多自定义 provider 的相关方法。

注入

与任何其他提供程序一样，异步提供程序通过其令牌被注入到其他组件。在上面的示例中，您将使用结构 `@Inject('ASYNC_CONNECTION')`。

实例

以上示例用于演示目的。如果你正在寻找更详细的例子，请看 这里。

动态模块

模块一章介绍了 `Nest` 模块的基础知识，并简要介绍了动态模块。本章扩展了动态模块的主题。完成后，您应该对它们是什么以及如何以及何时使用它们有很好的了解。

简介

文档概述部分中的大多数应用程序代码示例都使用了常规或静态模块。模块定义像**提供者**和**控制器**这样的组件组，它们作为整个应用程序的模块部分组合在一起。它们为这些组件提供了执行上下文或范围。例如，模块中定义的提供程序对模块的其他成员可见，而不需要导出它们。当提供者需要在模块外部可见时，它首先从其主机模块导出，然后导入到其消费模块。

首先，我们将定义一个 UsersModule 来提供和导出 UsersService 。 UsersModule 是 UsersService 的主模块。

typescript

```typescript
import { Module } from '@nestjs/common';
import { UsersService } from './users.service';

@Module({
  providers: [UsersService],
  exports: [UsersService],
})
export class UsersModule {}
```

接下来，我们将定义一个 AuthModule ，它导入 UsersModule ，使 UsersModule 导出的提供程序在 AuthModule 中可用:

typescript

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

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

这些构造使我们能够注入 UsersService 例如 AuthService 托管在中的 AuthModule ：

typescript

```typescript
import { Injectable } from '@nestjs/common';
import { UsersService } from '../users/users.service';

@Injectable()
export class AuthService {
  constructor(private readonly usersService: UsersService) {}
  /*
    Implementation that makes use of this.usersService
  */
}
```

我们将其称为静态模块绑定。 Nest 在主模块和消费模块中已经声明了连接模块所需的所有信息。让我们来看看这个过程中发生了什么。 Nest 通过以下方式使 UsersService 在 AuthModule 中可用:

1. 实例化 UsersModule ，包括传递导入 UsersModule 本身使用的其他模块，以及传递的任何依赖项(参见自定义提供程序)。
2. 实例化 AuthModule ，并将 UsersModule 导出的提供程序提供给 AuthModule 中的组件(就像在 AuthModule 中声明它们一样)。

在 AuthService 中注入 UsersService 实例。

动态模块实例

使用静态模块绑定，消费模块不会机会影响来自主机模块的提供者的配置方式。为什么这很重要?考虑这样一种情况:我们有一个通用模块，它需要在不同的用例中有不同的行为。这类似于许多系统中的**插件**概念，在这些系统中，一般功能需要一些配置才能供使用者使用。

Nest 的一个很好的例子是配置模块。 许多应用程序发现使用配置模块来外部化配置详细信息很有用。 这使得在不同部署中动态更改应用程序设置变得容易：例如，开发人员的开发数据库，测试环境的数据库等。通过将配置参数的管理委派给配置模块，应用程序源代码保持独立于配置参数。

主要在于配置模块本身，因为它是通用的(类似于 '插件')，需要由它的消费模块进行定制。这就是动态模块发挥作用的地方。使用动态模块特性，我们可以使配置模块成为动态的，这样消费模块就可以使用 API 来控制配置模块在导入时是如何定制的。

换句话说，动态模块提供了一个 API ，用于将一个模块导入到另一个模块中，并在导入模块时定制该模块的属性和行为，而不是使用我们目前看到的静态绑定。

配置模块示例

在本节中，我们将使用示例代码的基本版本。 截至本章末尾的完整版本在此处可用作工作示例。

我们的要求是使 ConfigModule 接受选项对象以对其进行自定义。 这是我们要支持的功能。 基本示例将 .env 文件的位置硬编码为项目根文件夹。 假设我们要使它可配置，以便您可以在您选择的任何文件夹中管理 .env 文件。 例如，假设您想将各种 .env 文件存储在项目根目录下名为 config 的文件夹中（即 src 的同级文件夹）。 在不同项目中使用 ConfigModule 时，您希望能够选择其他文件夹。

动态模块使我们能够将参数传递到要导入的模块中，以便我们可以更改其行为。 让我们看看它是如何工作的。 如果我们从最终目标开始，即从使用模块的角度看，然后向后工作，这将很有帮助。 首先，让我们快速回顾一下静态导入 `ConfigModule` 的示例（即，一种无法影响导入模块行为的方法）。 请密切注意 `@Module()` 装饰器中的 `imports` 数组：

typescript

```typescript
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { ConfigModule } from './config/config.module';

@Module({
  imports: [ConfigModule],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

让我们考虑一下动态模块导入是什么样子的，我们在其中传递了一个配置对象。比较这两个例子之间的导入数组的差异:

typescript

```typescript
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { ConfigModule } from './config/config.module';

@Module({
  imports: [ConfigModule.register({ folder: './config' })],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

让我们看看在上面的动态示例中发生了什么。变化的部分是什么?

1. `ConfigModule` 是一个普通类，因此我们可以推断它必须有一个名为 `register()` 的静态方法。我们知道它是静态的，因为我们是在 `ConfigModule` 类上调用它，而不是在类的实例上。注意:我们将很快创建的这个方法可以有任意名称，但是按照惯例，我们应该调用它 `forRoot()` 或 `register()` 方法。

2. `register()` 方法是由我们定义的，因此我们可以接受任何我们喜欢的参数。在本例中，我们将接受具有适当属性的简单 `options` 对象，这是典型的情况。

3. 我们可以推断 `register()` 方法必须返回类似模块的内容，因为它的返回值出现在熟悉的导入列表中，到目前为止，我们已经看到该列表包含了一个模块列表。

实际上，我们的 `register()` 方法将返回的是 `DynamicModule` 。 动态模块无非就是在运行时创建的模块，它具有与静态模块相同属性，外加一个称为模块的附加属性。 让我们快速查看一个示例静态模块声明，并密切注意传递给装饰器的模块选项：

```typescript
@Module({
  imports: [DogsService],
  controllers: [CatsController],
  providers: [CatsService],
  exports: [CatsService]
})
```

动态模块必须返回具有完全相同接口的对象，外加一个称为模块的附加属性。 模块属性用作模块的名称，并且应与模块的类名相同，如下例所示。

> 对于动态模块，模块选项对象的所有属性都是可选的，模块除外。

静态 `register()` 方法呢？ 现在我们可以看到它的工作是返回具有 `DynamicModule` 接口的对象。 当我们调用它时，我们实际上是在导入列表中提供一个模块，类似于在静态情况下通过列出模块类名的方式。 换句话说，动态模块 `API` 只是返回一个模块，而不是固定 `@Modules` 装饰器中的属性，而是通过编程方式指定它们。

仍然有一些细节需要详细了解：

1. 现在我们可以声明 `@Module()` 装饰器的 `imports` 属性不仅可以使用一个模块类名(例如， `imports: [UsersModule]`) ，还可以使用一个返回动态模块的函数(例如， `imports: [ConfigModule.register(...)]`)。

2. 动态模块本身可以导入其他模块。 在本示例中，我们不会这样做，但是如果动态模块依赖于其他模块的提供程序，则可以使用可选的 `imports` 属性导入它们。 同样，这与使用 `@Module()` 装饰器为静态模块声明元数据的方式完全相似。

有了这种理解，我们现在可以看看动态 `ConfigModule` 声明必须是什么样子。 让我们来看一下。

typescript

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

@Module({})
export class ConfigModule {
  static register(): DynamicModule {
    return {
      module: ConfigModule,
      providers: [ConfigService],
      exports: [ConfigService],
    };
  }
}
```

现在应该清楚各个部分是如何联系在一起的了。调用 `ConfigModule.register(...)` 将返回一个 `DynamicModule` 对象，该对象的属性基本上与我们通过 `@Module()` 装饰器提供的元数据相同。

> `DynamicModule` 需要从 `@nestjs/common` 包导入。

然而，我们的动态模块还不是很有趣，因为我们还没有引入任何我们想要配置它的功能。让我们接下来解决这个问题。

模块配置

定制 `ConfigModule` 行为的显而易见的解决方案是在静态 `register()` 方法中向其传递一个 `options` 对象，如我们上面所猜测的。让我们再次看一下消费模块的 `imports` 属性：

typescript

```typescript
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { ConfigModule } from './config/config.module';

@Module({
  imports: [ConfigModule.register({ folder: './config' })],
  controllers: [AppController],
  providers: [AppService],
```

```
})
export class AppModule {}
```

这很好地处理了将一个 options 对象传递给我们的动态模块。那么我们如在何 ConfigModule 中使用 options 对象呢?让我们考虑一下。我们知道，我们的 ConfigModule 基本上是一个提供和导出可注入服务(ConfigService)供其他提供者使用。实际上我们的 ConfigService 需要读取 options 对象来定制它的行为。现在让我们假设我们知道如何将 register() 方法中的选项获取到 ConfigService 中。有了这个假设，我们可以对服务进行一些更改，以便基于 options 对象的属性自定义其行为。(注意:目前，由于我们还没有确定如何传递它，我们将只硬编码选项。我们将在一分钟内解决这个问题)。

typescript

```
import { Injectable } from '@nestjs/common';
import * as dotenv from 'dotenv';
import * as fs from 'fs';
import { EnvConfig } from './interfaces';

@Injectable()
export class ConfigService {
  private readonly envConfig: EnvConfig;

  constructor() {
    const options = { folder: './config' };

    const filePath = `${process.env.NODE_ENV || 'development'}.env`;
    const envFile = path.resolve(__dirname, '../../', options.folder, filePat
    this.envConfig = dotenv.parse(fs.readFileSync(envFile));
  }

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

现在，我们的 ConfigService 知道如何在选项指定的文件夹中查找 .env 文件。

我们剩下的任务是以某种方式将 register() 步骤中的 options 对象注入 ConfigService 。当然，我们将使用依赖注入来做到这一点。这是一个关键点，所以一定要理解它。我们的 ConfigModule 提供 ConfigService 。而 ConfigService 又依赖于只在运行时提供的 options 对象。因此，在运行时，我们需要首先将 options 对象绑定到 Nest IoC 容器，然

后让 Nest 将其注入 ConfigService 。请记住，在**自定义提供者**一章中，提供者可以包含任何值，而不仅仅是服务，所以我们可以使用依赖项注入来处理简单的 options 对象。

让我们首先处理将 options 对象绑定到 IoC 容器的问题。我们在静态 register() 方法中执行此操作。请记住，我们正在动态地构造一个模块，而模块的一个属性就是它的提供者列表。因此，我们需要做的是将 options 对象定义为提供程序。这将使它可注入到 ConfigService 中，我们将在下一个步骤中利用它。在下面的代码中，注意 provider 数组:

typescript

```
import { DynamicModule, Module } from '@nestjs/common';

import { ConfigService } from './config.service';

@Module({})
export class ConfigModule {
  static register(options): DynamicModule {
    return {
      module: ConfigModule,
      providers: [
        {
          provide: 'CONFIG_OPTIONS',
          useValue: options,
        },
        ConfigService,
      ],
      exports: [ConfigService],
    };
  }
}
```

现在，我们可以通过将 'CONFIG_OPTIONS' 提供者注入 ConfigService 来完成这个过程。回想一下，当我们使用非类令牌定义提供者时，我们需要使用这里描述的 @Inject() 装饰器。

typescript

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

import * as dotenv from 'dotenv';
import * as fs from 'fs';

import { EnvConfig } from './interfaces';

@Injectable()
```

```typescript
export class ConfigService {
  private readonly envConfig: EnvConfig;

  constructor(@Inject('CONFIG_OPTIONS') private options) {
    const filePath = `${process.env.NODE_ENV || 'development'}.env`;
    const envFile = path.resolve(__dirname, '../../', options.folder, filePat
    this.envConfig = dotenv.parse(fs.readFileSync(envFile));
  }

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

最后一点:为了简单起见，我们使用了上面提到的基于字符串的注入标记(`'CONFIG_OPTIONS'`)，但是最佳实践是将它定义为一个单独文件中的常量(或符号)，然后导入该文件。例如:

```typescript
export const CONFIG_OPTIONS = 'CONFIG_OPTIONS';
```

实例

本章代码的完整示例可以在这里找到。

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(provider)和模块(module)之间创建循环依赖关系，但我们建议您尽可能避免。但是有时候难以避免，所以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引用。当 CatsService 和 CommonService 相互依赖时，关系的双方都需要使用 @Inject() 和 forwardRef() ，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

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

forwardRef() 需要从 @nestjs/common 包中导入的。

这只是关系的一方面。现在让我们对 CommonService 做同样的事情:

common.service.ts

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

实例化的顺序是不确定的。不能保证哪个构造函数会被先调用。

模块引用

为了在模块(module)之间创建循环依赖，必须在模块关联的两个部分上使用相同的 forwardRef() ：

> common.module.ts

typescript

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

Nest 提供 ModuleRef 类来导航内部提供者列表，并通过类名获取对任何提供者的引用。ModuleRef 可以通过正常的方式注入到一个类中:

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 类从 @nestjs/core 包中导入。

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。另外, 你可以使用非严格模式(non-strict mode), 保证你可以在整个应用中的任何地方获得该提供者。

typescript

```
this.moduleRef.get(Service, { strict: false });
```

注入作用域

对于使用不同语言的人来说，在 Nest 中几乎所有内容都在传入请求之间共享，这可能会很尴尬。我们有到数据库的连接池，全局状态的单例服务等等。通常， Node.js 不遵循 request/response 多线程无状态模型，其中每个请求都由单独的线程处理。因此，对于我们的应用程序来说，使用单例实例是完全**安全**的。

但是，存在基于请求的控制器生命周期可能是有意行为的边缘情况，例如 GraphQL 应用程序中的每请求缓存，请求跟踪或多租户。我们怎么处理它们?

作用域

基本上，每个提供者都可以作为一个单例，被请求范围限定，并切换到瞬态模式。请参见下表，以熟悉它们之间的区别。

SINGLETON	每个提供者可以跨多个类共享。提供者生命周期严格绑定到应用程序生命周期。一旦应用程序启动，所有提供程序都已实例化。默认情况下使用单例范围。
REQUEST	在请求处理完成后，将为每个传入请求和垃圾收集专门创建提供者的新实例
TRANSIENT	临时提供者不能在提供者之间共享。每当其他提供者向 Nest 容器请求特定的临时提供者时，该容器将创建一个新的专用实例

> 使用单例范围始终是推荐的方法。请求之间共享提供者可以降低内存消耗，从而提高应用程序的性能(不需要每次实例化类)。

使用 (Usage)

为了切换到另一个注入范围，您必须向 @Injectable() 装饰器传递一个参数

typescript

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

@Injectable({ scope: Scope.REQUEST })
export class CatsService {}
```

在自定义提供者的情况下，您必须设置一个额外的范围属性

typescript

```typescript
{
  provide: 'CACHE_MANAGER',
  useClass: CacheManager,
  scope: Scope.TRANSIENT,
}
```

当涉及到控制器时，传递 `ControllerOptions` 对象

typescript

```typescript
@Controller({
  path: 'cats',
  scope: Scope.REQUEST,
})
export class CatsController {}
```

> 网关永远不应该依赖于请求范围的提供者，因为它们充当单例。一个网关封装了一个真正的套接字，不能多次被实例化

所有请求注入

必须非常谨慎地使用请求范围的提供者。请记住， `scope` 实际上是在注入链中冒泡的。如果您的控制器依赖于一个请求范围的提供者，这意味着您的控制器实际上也是请求范围。

想象一下下面的链: `CatsController <- CatsService <- CatsRepository` 。如果您的 `CatsService` 是请求范围的(从理论上讲，其余的都是单例)，那么 `CatsController` 也将成为请求范围的(因为必须将请求范围的实例注入到新创建的控制器中)，而 `CatsRepository` 仍然是单例的。

> 在这种情况下，循环依赖关系将导致非常痛苦的副作用，因此，您当然应该避免创建它们

请求提供者

在 HTTP 应用程序中，使用请求范围的提供者使您能够注入原始请求引用

```typescript
import { Injectable, Scope, Inject } from '@nestjs/common';
import { REQUEST } from '@nestjs/core';
import { Request } from 'express';

@Injectable({ scope: Scope.REQUEST })
export class CatsService {
  constructor(@Inject(REQUEST) private readonly request: Request) {}
}
```

但是，该功能既不能用于微服务，也不能用于 GraphQL 应用程序。在 GraphQL 应用程序中，可以注入 CONTEXT 。

```typescript
import { Injectable, Scope, Inject } from '@nestjs/common';
import { CONTEXT } from '@nestjs/graphql';

@Injectable({ scope: Scope.REQUEST })
export class CatsService {
  constructor(@Inject(CONTEXT) private readonly context) {}
}
```

然后，您可以配置您的 context 值(在 GraphQLModule 中)，以包含请求作为其属性。

性能

使用请求范围的提供者将明显影响应用程序性能。即使 Nest 试图缓存尽可能多的元数据，它仍然必须为每个请求创建类的实例。因此，它将降低您的平均响应时间和总体基准测试结果。如果您的提供者不一定需要请求范围，那么您应该坚持使用单例范围。

生命周期

生命周期事件

所有应用程序元素都有一个由 Nest 管理的生命周期。 Nest 提供了**生命周期钩子**，提供了对关键生命时刻的可见性，以及在关键时刻发生时采取行动的能力。

生命周期序列

通过调用构造函数创建注入/控制器后， Nest 在特定时刻按如下顺序调用生命周期钩子方法。

OnModuleInit	初始化主模块后调用
OnApplicationBootstrap	在应用程序完全启动并引导后调用
OnModuleDestroy	在Nest销毁主模块(app.close() 方法之前进行清理)
OnApplicationShutdown	响应系统信号(当应用程序关闭时，例如 SIGTERM)

使用

所有应用周期的钩子都有接口表示，接口在技术上是可选的，因为它们在 TypeScript 编译之后就不存在了。尽管如此，为了从强类型和编辑器工具中获益，使用它们是一个很好的实践。

typescript

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

@Injectable()
export class UsersService implements OnModuleInit {
  onModuleInit() {
    console.log(`The module has been initialized.`);
  }
}
```

此外， OnModuleInit 和 OnApplicationBootstrap 钩子都允许您延迟应用程序初始化过程(返回一个 Promise 或将方法标记为 async)。

typescript

```
async onModuleInit(): Promise<void> {
  await this.fetch();
}
```

OnApplicationShutdown

OnApplicationShutdown 响应系统信号(当应用程序通过 SIGTERM 等方式关闭时)。使用此钩子可以优雅地关闭 Nest 应用程序。这一功能通常用于 Kubernetes 、 Heroku 或类似的服务。

要使用此钩子，必须激活侦听器，侦听关闭信号。

typescript

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

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  // Starts listening to shutdown hooks
  app.enableShutdownHooks();
  await app.listen(3000);
}
bootstrap();
```

如果应用程序接收到一个信号，它将调用 onApplicationShutdown 函数，并将相应的信号作为第一个参数 Injectable 。如果函数确实返回了一个 promise ，那么在 promise 被解析或拒绝之前，它不会关闭 Nest 应用程序。

typescript

```
@Injectable()
class UsersService implements OnApplicationShutdown {
  onApplicationShutdown(signal: string) {
    console.log(signal); // e.g. "SIGINT"
  }
}
```

跨平台

Nest 的作为一个跨平台的框架。平台独立性使得**创建可重用的逻辑部分**成为可能，人们可以利用这种逻辑部件跨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。

一次编译, 各处运行

概览主要涉及 HTTP 服务器(REST APIs)。但是，所有这些构建的模块都可以轻松用于不同的传输层(microservices 或 websockets)。此外， Nest 还配备了专用的 GraphQL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通过 Nest 创建在 Node.js 上运行的所有应用。

Nest 希望成为 Node.js 应用程序的完整平台，为您的应用程序带来更高级别的模块化和可重用性。一次构建，可在任何地方使用！

测试

自动化测试是成熟**软件产品**的重要组成部分。对于覆盖系统中关键的部分是极其重要的。为了实现这个目标，我们产生了一系列不同的测试首单，例如集成测试，单元测试， e2e 测试等。 Nest 提供了一系列改进测试体验的测试实用程序。

通常，您可以使用您喜欢的任何**测试框架**，选择任何适合您要求的工具。 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样板代码，但你仍然可以删除它, 使用任何其他测试框架。

安装

首先，我们需要安装所需的 npm 包:

```bash
$ 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。该框架是test runner, 并提供断言函数和提升测试实用工具，以帮助 mock ， spy 等。一旦被调用, 我们已经手动强制执行 catsService.findAll() 方法来返回结果。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);
    });
  });
});
```

保持你的测试文件测试类附近。测试文件必须以 `.spec` 或 `.test` 结尾

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工具。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],
        providers: [CatsService],
      }).compile();

    catsService = module.get<CatsService>(CatsService);
    catsController = module.get<CatsController>(CatsController);
  });

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

`Test` 类有一个 `createTestingModule()` 方法，该方法将模块的元数据（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，该实例提供了一些方法，但是当涉及到单元测试时，这些方法中只有 `compile()` 是有用的。这个方式是**异步**的，因此必须等待执行完成。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用户提供者覆盖现有的提供者。

E2E

当应用程序代码变多时，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，但另外我们使用 supertest模拟 HTTP 请求。

> 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();

    app = module.createNestApplication();
    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 目录下, 并且以 .e2e-spec 或 .e2e-test 结尾。

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。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，必须使用 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)

译者署名

用户名	头像	职能	签名

用户名	头像	职能	签名
@zuohuadong		翻译	专注于 caddy 和 nest，@zuohuadong at Github
@Drixn		翻译	专注于 nginx 和 C++，@Drixn
@Armor		翻译	专注于 Java 和 Nest，@Armor
@gaoyangy		校正	专注于Vue，TS/JS

End