**Angular开发指南**

目录

[**1.** **文件规范** 2](#_Toc44246111)

[**1.1.** **文件结构与约定** 2](#_Toc44246112)

[**1.2.** **单一职责原则** 2](#_Toc44246113)

[**1.2.1** **单一规则** 2](#_Toc44246114)

[**1.2.2** **小函数** 3](#_Toc44246115)

[**2.** **命名规范** 3](#_Toc44246116)

[**2.1** **总体命名原则** 3](#_Toc44246117)

[**2.2** **使用点和横杠来分隔文件名** 3](#_Toc44246118)

[**2.3** **符号名与文件名** 4](#_Toc44246119)

[**2.4** **服务名** 5](#_Toc44246120)

[**2.5** **引导程序** 6](#_Toc44246121)

[**2.6** **组件选择器** 6](#_Toc44246122)

[**2.7** **组件的自定义前缀** 6](#_Toc44246123)

[**2.8** **指令选择器** 7](#_Toc44246124)

[**2.9** **指令的自定义前缀** 7](#_Toc44246125)

[**2.10** **管道名** 7](#_Toc44246126)

[**2.11** **Angular NgModule 命名** 8](#_Toc44246127)

[**3.** **编程规范** 9](#_Toc44246128)

[**3.1** **类** 9](#_Toc44246129)

[**3.2** **常量** 9](#_Toc44246130)

[**3.3** **接口** 10](#_Toc44246131)

[**3.4** **属性和方法** 10](#_Toc44246132)

[**3.5** **导入语句中的空行** 11](#_Toc44246133)

[**3.6** **注释** 11](#_Toc44246134)

[**3.7** **优化作用域链** 12](#_Toc44246135)

[**3.8** **合并http请求** 12](#_Toc44246136)

[**3.9** **业务分离** 12](#_Toc44246137)

[**4.** **应用程序结构与 NgModule** 13](#_Toc44246138)

[**4.1** **总体结构的指导原则** 13](#_Toc44246139)

[**4.2** **按特性组织的目录结构** 13](#_Toc44246140)

[**4.3** **应用的根模块** 14](#_Toc44246141)

[**4.4** **特性模块** 14](#_Toc44246142)

[**4.5** **共享特性模块** 15](#_Toc44246143)

[**4.6** **核心特性模块** 15](#_Toc44246144)

[**4.7** **防止多次导入 CoreModule** 16](#_Toc44246145)

[**4.8** **惰性加载的目录** 17](#_Toc44246146)

[**4.9** **不要直接导入惰性加载的目录** 18](#_Toc44246147)

[**5.** **组件** 18](#_Toc44246148)

[**5.1** **把组件当做元素** 18](#_Toc44246149)

[**5.2** **把模板和样式从组件中分离** 19](#_Toc44246150)

[**5.3** **内联输入和输出属性装饰器** 19](#_Toc44246151)

[**5.4** **避免为输入和输出属性指定别名** 21](#_Toc44246152)

[**5.5** **成员顺序** 22](#_Toc44246153)

[**5.6** **把逻辑放到服务里** 24](#_Toc44246154)

[**5.7** **不要给输出属性加前缀** 25](#_Toc44246155)

[**5.8** **把表现层逻辑放到组件类里** 25](#_Toc44246156)

[**6.** **指令** 27](#_Toc44246157)

[**6.1** **使用指令来增强已有元素** 27](#_Toc44246158)

[**6.2** **HostListener、HostBinding 装饰器 和 组件元数据 host** 27](#_Toc44246159)

[**7.** 服务 29](#_Toc44246160)

[**7.1** **服务总是单例的** 29](#_Toc44246161)

[**7.2** **单一职责** 29](#_Toc44246162)

[**7.3** **提供一个服务** 29](#_Toc44246163)

[**7.4** **使用 @Injectable() 类装饰器** 30](#_Toc44246164)

[**8.** **数据服务** 31](#_Toc44246165)

[**8.1** **通过服务与 Web 服务器通讯** 31](#_Toc44246166)

[**9.** **生命周期钩子** 31](#_Toc44246167)

[**9.1** **实现生命周期钩子接口** 31](#_Toc44246168)

1. **文件规范**
   1. **文件结构与约定**

后述代码例子中，有的文件有一个或多个相似名字的配套文件。（例如 hero.component.ts 和 hero.component.html）。

本文将会使用像 hero.component.ts|html|css|spec 的简写来表示上面描述的多个文件，目的是保持本指南的简洁性，增加描述文件结构时的可读性。

* 1. **单一职责原则**

对所有的组件、服务等等应用单一职责原则，这样可以让应用更干净、更易读、更易维护、更易测试。

1. **单一规则**

坚持每个文件只定义一样东西（例如服务或组件），考虑把文件大小限制在 400 行代码以内

1. 单组件文件非常容易阅读、维护，并能防止在版本控制系统里与团队冲突。
2. 单组件文件可以防止一些隐蔽的程序缺陷，当把多个组件合写在同一个文件中时，可能造成共享变量、创建意外的闭包，或者与依赖之间产生意外耦合等情况。
3. 单独的组件通常是该文件默认的导出，可以用路由器实现按需加载。
4. 最关键的是，可以增强代码可重用性和阅读性，减少出错的可能性。
5. **小函数**

坚持定义简单函数，考虑限制在 75 行之内

1)        简单函数更易于测试，特别是当它们只做一件事，只为一个目的服务时。

2)        简单函数促进代码重用。

3)        简单函数更易于阅读。

4)        简单函数更易于维护。

5)        简单函数可避免易在大函数中产生的隐蔽性错误，例如与外界共享变量、创建意外的闭包或与依赖之间产生意外耦合等。

1. **命名规范**

命名约定对可维护性和可读性非常重要。本文为文件名和符号名制定了一套命名约定。

1. **总体命名原则**

坚持所有符号使用一致的命名规则，坚持遵循同一个模式来描述符号的特性和类型，推荐的模式为 feature.type.ts。

1. 命名约定提供了一致的方式来查找内容，让你一眼就能找到。 项目的一致性是至关重要的。团队内的一致性也很重要。整个公司的一致性会提供惊人的效率。
2. 命名约定帮助你更快得找到想找的代码，也更容易理解它。
3. 目录名和文件名应该清楚的传递它们的意图。例如，app/heroes/hero-list.component.ts 包含了一个用来管理英雄列表的组件。
4. **使用点和横杠来分隔文件名**

坚持在描述性名字中，用横杠来分隔单词。

坚持使用点来分隔描述性名字和类型。

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

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

1. 类型名字提供一致的方式来快速的识别文件中有什么。
2. 利用编辑器或者 IDE 的模糊搜索功能，可以很容易地找到特定文件。
3. 像 .service 这样的没有简写过的类型名字，描述清楚，毫不含糊。 像 .srv, .svc, 和 .serv 这样的简写可能令人困惑。
4. 为自动化任务提供模式匹配。
5. **符号名与文件名**

坚持为所有东西使用一致的命名约定，以它们所代表的东西命名。

坚持使用大写驼峰命名法来命名类。符号名匹配它所在的文件名。

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

坚持在符号名后面追加约定的类型后缀（例如 .component.ts、.directive.ts、.module.ts、.pipe.ts、.service.ts）。

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

1. 遵循一致的约定可以快速识别和引用不同类型的资源。

|  |  |
| --- | --- |
| **符号名** | **文件名** |
| @Component({ ... })  export class AppComponent { } | app.component.ts |
| @Component({ ... })  export class HeroesComponent { } | heroes.component.ts |
| @Component({ ... })  export class HeroListComponent { } | hero-list.component.ts |
| @Component({ ... })  export class HeroDetailComponent { } | hero-detail.component.ts |
| @Directive({ ... })  export class ValidationDirective { } | validation.directive.ts |
| @NgModule({ ... })  export class AppModule | app.module.ts |
| @Pipe({ name: 'initCaps' })  export class InitCapsPipe implements PipeTransform { } | init-caps.pipe.ts |
| @Injectable()  export class UserProfileService { } | user-profile.service.ts |

1. **服务名**

坚持在描述性名字中，用横杠来分隔单词。

坚持使用一致的规则命名服务，以它们的特性来命名。

坚持为服务的类名加上 Service 后缀。 例如，获取数据或图表列表的服务应该命名为 DataService 或 ChartService。

有些词汇显然就是服务，比如那些以“-er”后缀结尾的。比如把记日志的服务命名为 Logger 就比 LoggerService 更好些。需要在你的项目中决定这种特例是否可以接受。 但无论如何，都要尽量保持一致。

1. 提供一致的方式来快速识别和引用服务。
2. 像 Logger 这样的清楚的服务名不需要后缀。
3. 像 Credit 这样的，服务名是名词，需要一个后缀。当不能明显分辨它是服务还是其它东西时，应该添加后缀。

|  |  |
| --- | --- |
| **符号名** | **文件名** |
| @Injectable()  export class HeroDataService { } | hero-data.service.ts |
| @Injectable()  export class CreditService { } | credit.service.ts |
| @Injectable()  export class Logger { } | logger.service.ts |

1. **引导程序**

坚持把应用的引导程序和平台相关的逻辑放到名为 main.ts 的文件里。

坚持在引导逻辑中包含错误处理代码。

避免把应用逻辑放在 main.ts 中，而应放在组件或服务里。

1. 应用的启动逻辑遵循一致的约定。
2. 这是从其它技术平台借鉴的常用约定。
3. **组件选择器**

坚持使用短横线命名法（dashed-case）或叫烤串命名法（kebab-case）来命名组件的元素选择器，让元素名和自定义元素规范保持一致。

1. **组件的自定义前缀**

坚持使用带连字符的小写元素选择器值（例如 admin-users）。

坚持为组件选择器添加自定义前缀。例如，toh 前缀表示 Tour of Heroes（英雄指南），而前缀 admin 表示管理特性区。

坚持使用前缀来识别特性区或者应用程序本身。

1. 防止与其它应用中的组件和原生 HTML 元素发生命名冲突。
2. 更容易在其它应用中推广和共享组件。
3. 组件在 DOM 中更容易被区分出来。

@Component({

selector: 'toh-hero-button',

templateUrl: './hero-button.component.html'

})

export class HeroButtonComponent {}

1. **指令选择器**

坚持使用小驼峰形式命名指令的选择器。

1. 可以让指令中的属性名与视图中绑定的属性名保持一致。
2. Angular 的 HTML 解析器是大小写敏感的，可以识别小驼峰形式。

1. **指令的自定义前缀**

坚持为指令的选择器添加自定义前缀（例如前缀 toh 来自 Tour of Heroes）。

坚持用小驼峰形式拼写非元素选择器，除非该选择器用于匹配原生 HTML 属性。

1. 防止名字冲突。
2. 指令更加容易被识别。

@Directive({

selector: '[tohValidate]'

})

export class ValidateDirective {}

1. **管道名**

坚持为所有管道使用一致的命名约定，用它们的特性来命名。

1. 提供一致方式快速识别和引用管道。

|  |  |
| --- | --- |
| **符号名** | **文件名** |
| @Pipe({ name: 'ellipsis' })  export class EllipsisPipe implements PipeTransform { } | ellipsis.pipe.ts |
| @Pipe({ name: 'initCaps' })  export class InitCapsPipe implements PipeTransform { } | init-caps.pipe.ts |

1. **Angular NgModule 命名**

坚持为符号名添加 Module 后缀。

坚持为文件名添加 .module.ts 扩展名。

坚持用特性名和所在目录命名模块。

1. 提供一致的方式来快速标识和引用模块。
2. 大驼峰命名法是一种命名约定，用来标识可用构造函数实例化的对象。

很容易就能看出这个模块是同名特性的根模块。

坚持为 RoutingModule 类名添加 RoutingModule 后缀。

坚持为 RoutingModule 的文件名添加 -routing.module.ts 后缀。

1. RoutingModule 是一种专门用来配置 Angular 路由器的模块。 “类名和文件名保持一致”的约定使这些模块易于发现和验证。

|  |  |
| --- | --- |
| **符号名** | **文件名** |
| @NgModule({ ... })  export class AppModule { } | app.module.ts |
| @NgModule({ ... })  export class HeroesModule { } | heroes.module.ts |
| @NgModule({ ... })  export class VillainsModule { } | villains.module.ts |
| @NgModule({ ... })  export class AppRoutingModule { } | app-routing.module.ts |
| @NgModule({ ... })  export class HeroesRoutingModule { } | heroes-routing.module.ts |

1. **编程规范**

坚持一致的编程、命名和空格的约定。

1. **类**

坚持使用大写驼峰命名法来命名类。

1. 遵循类命名传统约定。
2. 类可以被实例化和构造实例。根据约定，用大写驼峰命名法来标识可构造的东西。

export class ExceptionService {

constructor() { }

}

1. **常量**

坚持用 const 声明变量，除非它们的值在应用的生命周期内会发生变化。

1. 告诉读者这个值是不可变的。
2. TypeScript 会要求在声明时立即初始化，并阻止再次赋值，以帮助确保你的设计意图。

考虑把常量名拼写为小驼峰格式。

1. 小驼峰变量名 (heroRoutes) 比传统的大写蛇形命名法 (HERO\_ROUTES) 更容易阅读和理解。
2. 把常量命名为大写蛇形命名法的传统源于现代 IDE 出现之前， 以便阅读时可以快速发现那些 const 定义。 TypeScript 本身就能够防止意外赋值。

坚持容许现存的const 常量沿用大写蛇形命名法。

1. 传统的大写蛇形命名法仍然很流行、很普遍，特别是在第三方模块中。 修改它们没多大价值，还会有破坏现有代码和文档的风险。

export const mockHeroes = ['Sam', 'Jill'];

export const heroesUrl = 'api/heroes';

export const VILLAINS\_URL = 'api/villains';

1. **接口**

坚持使用大写驼峰命名法来命名接口。

考虑在接口名字前面加 I 前缀。

考虑在服务和可声明对象（组件、指令和管道）中用类代替接口。

考虑用接口作为数据模型。

1. TypeScript 指导原则不建议使用 “I” 前缀。
2. 单独一个类的代码量小于类+接口。
3. 类可以作为接口使用（只是用 implements 代替 extends 而已）。
4. 在 Angular 依赖注入系统中，接口类（译注：指写成类的形式，但是只当做接口使用）可以作为服务提供商的查找令牌。

[复制代码](javascript:void(0);)

import { Injectable } from '@angular/core';

import { IHero } from './hero.model';

@Injectable()

export class HeroCollectorService {

hero: IHero;

constructor() { }

}

[复制代码](javascript:void(0);)

1. **属性和方法**

坚持使用小写驼峰命名法来命名属性和方法。

避免为私有属性和方法添加下划线前缀。

1. 遵循传统属性和方法的命名约定。
2. JavaScript 不支持真正的私有属性和方法。
3. TypeScript 工具让识别私有或公有属性和方法变得很简单。

[复制代码](javascript:void(0);)

import { Injectable } from '@angular/core';

@Injectable()

export class ToastService {

message: string;

private toastCount: number;

hide() {

this.toastCount--;

this.log();

}

show() {

this.toastCount++;

this.log();

}

private log() {

console.log(this.message);

}

}

[复制代码](javascript:void(0);)

1. **导入语句中的空行**

坚持在第三方导入和应用导入之间留一个空行。

考虑按模块名字的字母顺排列导入行。

考虑在解构表达式中按字母顺序排列导入的东西。

1. 空行可以让阅读和定位本地导入更加容易。
2. 按字母顺序排列可以让阅读和定位本地导入更加容易。

import { HttpClient } from '@angular/common/http';

import { Injectable } from '@angular/core';

import { ExceptionService, SpinnerService, ToastService } from '../../core';

import { Hero } from './hero.model';

1. **注释**

对于函数定义以及比较复杂的逻辑，要写必要的注释。对于函数定义的注释以及函数内部的连续多行注释使用块注释“/\*…\*/”，对于函数内部的单行注释使用行注释“//…”。

1. **优化作用域链**

循环内部引用的对象，如果层级很深，要在循环外部定义在循环内部直接使用的层级变量，以便提高JS引擎对循环内的标识符解析速度。比如有如下对象：

[复制代码](javascript:void(0);)

const fruit = {

name: 'Apple',

color: 'red',

provider: {

name: 'Jaffray',

phones: [

'01088888888',

'18812345678'

]

}

};

let dummy;

//方法一，效率较低

for(let phone of fruit.provider.phones) {

dummy = phone;

}

//方法二，效率较高

const phones = fruit.provider.phones;

for(let phone of phones) {

dummy = phone;

}

[复制代码](javascript:void(0);)

1. **合并http请求**

对于一些性质相似且同一时期会并发访问的http请求，应与接口开发者沟通，将接口合并，以减少发送请求次数、加快并发请求执行，进而提高性能。

比如分别获取性别和年龄的请求，合并在一个接口完成更高效；获取省份和市区的请求，合并在一个接口完成更高效。

1. **业务分离**

在构建组件的过程中，要将业务逻辑从组件的渲染中分离出来，组件只管对指定格式的数据进行渲染，程序根据业务需要，创建不同的服务来完成组件需要的数据的获取和组装等处理，最终将处理好的数据传递给组件去渲染，以提高组件的可复用性能和可移植性能。

1. **应用程序结构与 NgModule**

坚持一致的编程、命名和空格的约定。

所有应用程序的源代码都放到名叫 src 的目录里。 所有特性区都在自己的文件夹中，带有它们自己的 NgModule。

所有内容都遵循每个文件一个特性的原则。每个组件、服务和管道都在自己的文件里。所有第三方程序包保存到其它目录里，而不是 src 目录，因为你几乎不会修改它们，所以不希望它们弄乱你的应用程序。使用本规范介绍的文件命名约定。

1. **总体结构的指导原则**

坚持把所有源代码都放到名为 src 的目录里。

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

1. 在早期阶段能够帮助保持应用的结构小巧且易于维护和移植，这样当应用增长时就容易进化了。
2. 组件通常有四个文件 (\*.html、 \*.css、 \*.ts 和 \*.spec.ts)，它们很容易把一个目录弄乱。
3. **按特性组织的目录结构**

坚持根据特性区命名目录。

1. 开发人员可以快速定位代码，扫一眼就能知道每个文件代表什么，目录尽可能保持扁平，既没有重复也没有多余的名字。
2. LIFT 原则中包含了所有这些。
3. 遵循 LIFT 原则精心组织内容，避免应用变得杂乱无章。
4. 当有很多文件时（例如 10 个以上），在专用目录型结构中定位它们会比在扁平结构中更容易。

坚持为每个特性区创建一个 NgModule。

1. NgModule 使惰性加载可路由的特性变得更容易。
2. NgModule 隔离、测试和复用特性更容易。
3. **应用的根模块**

坚持在应用的根目录创建一个 NgModule（例如 /src/app）。

1. 每个应用都至少需要一个根 NgModule。

考虑把根模块命名为 app.module.ts。

1. 能让定位和识别根模块变得更容易。

app/app.module.ts：

[复制代码](javascript:void(0);)

import { NgModule } from '@angular/core';

import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';

import { HeroesComponent } from './heroes/heroes.component';

@NgModule({

imports: [

BrowserModule,

],

declarations: [

AppComponent,

HeroesComponent

],

exports: [ AppComponent ],

entryComponents: [ AppComponent ]

})

export class AppModule {}

[复制代码](javascript:void(0);)

1. **特性模块**

坚持为应用中每个明显的特性创建一个 NgModule。

坚持把特性模块放在与特性区同名的目录中（例如 app/heroes）。

坚持特性模块的文件名应该能反映出特性区的名字和目录（例如 app/heroes/heroes.module.ts）。

坚持特性模块的符号名应该能反映出特性区、目录和文件名（例如在 app/heroes/heroes.module.ts 中定义 HeroesModule）。

1. 特性模块可以对其它模块暴露或隐藏自己的实现。
2. 特性模块标记出组成该特性分区的相关组件集合。
3. 方便路由到特性模块 —— 无论是用主动加载还是惰性加载的方式。
4. 特性模块在特定的功能和其它应用特性之间定义了清晰的边界。
5. 特性模块帮助澄清开发职责，以便于把这些职责指派给不同的项目组。
6. 特性模块易于隔离，以便测试。
7. **共享特性模块**

坚持在 shared 目录中创建名叫 SharedModule 的特性模块（例如在 app/shared/shared.module.ts 中定义 SharedModule）。

坚持在共享模块中声明那些可能被特性模块引用的可复用组件、指令和管道。

考虑把可能在整个应用中到处引用的模块命名为 SharedModule。

考虑 不要在共享模块中提供服务。服务通常是单例的，应该在整个应用或一个特定的特性模块中只有一份。 不过也有例外，比如，在下面的范例代码中，注意 SharedModule 提供了 FilterTextService。这里可以这么做，因为该服务是无状态的，也就是说，该服务的消费者不会受到这些新实例的影响。

坚持在 SharedModule 中导入所有模块都需要的资产（例如 CommonModule 和 FormsModule）。

1. SharedModule 中包含的组件、指令和管道可能需要来自其它公共模块的特性（例如来自 CommonModule 中的 ngFor 指令）。

坚持在 SharedModule 中声明所有组件、指令和管道。

坚持从 SharedModule 中导出其它特性模块所需的全部符号。

1. SharedModule 的存在，能让常用的组件、指令和管道在很多其它模块的组件模板中都自动可用。

避免在 SharedModule 中指定应用级的单例服务提供商。如果是刻意要得到多个服务单例也行，不过还是要小心。

1. 惰性加载的特性模块如果导入了这个共享模块，会创建一份自己的服务副本，这可能会导致意料之外的后果。
2. 对于单例服务，你不希望每个模块都有自己的实例。 而如果 SharedModule 提供了一个服务，那就有可能发生这种情况。
3. **核心特性模块**

考虑把那些数量庞大、辅助性的、只用一次的类收集到核心模块中，让特性模块的结构更清晰简明。

坚持把那些“只用一次”的类收集到 CoreModule 中，并对外隐藏它们的实现细节。简化的 AppModule 会导入 CoreModule，并且把它作为整个应用的总指挥。

坚持在 core 目录下创建一个名叫 CoreModule 的特性模块（例如在 app/core/core.module.ts 中定义 CoreModule）。

坚持把要共享给整个应用的单例服务放进 CoreModule 中（例如 ExceptionService 和 LoggerService）。

坚持导入 CoreModule 中的资产所需要的全部模块（例如 CommonModule 和 FormsModule）。

1. CoreModule 提供了一个或多个单例服务。Angular 使用应用的根注入器注册这些服务提供商，让每个服务的这个单例对象对所有需要它们的组件都是可用的，而不用管该组件是通过主动加载还是惰性加载的方式加载的。
2. CoreModule 将包含一些单例服务。而如果是由惰性加载模块来导入这些服务，它就会得到一个新实例，而不是所期望的全应用级单例。

坚持把应用级、只用一次的组件收集到 CoreModule 中。 只在应用启动时从 AppModule 中导入它一次，以后再也不要导入它（例如 NavComponent 和 SpinnerComponent）。

1. 真实世界中的应用会有很多只用一次的组件（例如加载动画、消息浮层、模态框等），它们只会在 AppComponent 的模板中出现。 不会在其它地方导入它们，所以没有共享的价值。 然而它们又太大了，放在根目录中就会显得乱七八糟的。

避免在 AppModule 之外的任何地方导入 CoreModule。

1. 如果惰性加载的特性模块直接导入 CoreModule，就会创建它自己的服务副本，并导致意料之外的后果。
2. 主动加载的特性模块已经准备好了访问 AppModule 的注入器，因此也能取得 CoreModule 中的服务。

坚持从 CoreModule 中导出 AppModule 需导入的所有符号，使它们在所有特性模块中可用。

1. CoreModule 的存在就让常用的单例服务在所有其它模块中可用。
2. 你希望整个应用都使用这个单例服务。 你不希望每个模块都有这个单例服务的单独的实例。 然而，如果 CoreModule 中提供了一个服务，就可能偶尔导致这种后果。

AppModule 变得更小了，因为很多应用根部的类都被移到了其它模块中。 AppModule 变得稳定了，因为你将会往其它模块中添加特性组件和服务提供者，而不是这个 AppModule。 AppModule 把工作委托给了导入的模块，而不是亲力亲为。 AppModule 聚焦在它自己的主要任务上：作为整个应用的总指挥。

1. **防止多次导入 CoreModule**

应该只有 AppModule 才允许导入 CoreModule。

坚持防范多次导入 CoreModule，并通过添加Guards逻辑来尽快失败。

1. Guards可以阻止对 CoreModule 的多次导入。
2. Guards会禁止创建单例服务的多个实例。

app/core/module-import-guard.ts：

export function throwIfAlreadyLoaded(parentModule: any, moduleName: string) {

if (parentModule) {

throw new Error(`${moduleName} has already been loaded. Import Core modules in the AppModule only.`);

}

}

app/core/core.module.ts：

[复制代码](javascript:void(0);)

import { NgModule, Optional, SkipSelf } from '@angular/core';

import { CommonModule } from '@angular/common';

import { LoggerService } from './logger.service';

import { NavComponent } from './nav/nav.component';

import { throwIfAlreadyLoaded } from './module-import-guard';

@NgModule({

imports: [

CommonModule // we use ngFor

],

exports: [NavComponent],

declarations: [NavComponent],

providers: [LoggerService]

})

export class CoreModule {

constructor( @Optional() @SkipSelf() parentModule: CoreModule) {

throwIfAlreadyLoaded(parentModule, 'CoreModule');

}

}

[复制代码](javascript:void(0);)

1. **惰性加载的目录**

某些边界清晰的应用特性或工作流可以做成惰性加载或按需加载的，而不用总是随着应用启动。

坚持把惰性加载特性下的内容放进惰性加载目录中。 典型的惰性加载目录包含路由组件及其子组件以及与它们有关的那些资产和模块。

1. 这种目录让标识和隔离这些特性内容变得更轻松。
2. **不要直接导入惰性加载的目录**

避免让兄弟模块和父模块直接导入惰性加载特性中的模块。

1. 直接导入并使用此模块会立即加载它，而原本的设计意图是按需加载它。
2. **组件**
3. **把组件当做元素**

避免让兄弟模块和父模块直接导入惰性加载特性中的模块。

1. 直接导入并使用此模块会立即加载它，而原本的设计意图是按需加载它。

考虑给组件一个元素选择器，而不是属性或类选择器。

1. 组件有很多包含 HTML 以及可选 Angular 模板语法的模板。 它们显示内容。开发人员会把组件像原生 HTML 元素和 WebComponents 一样放进页面中。
2. 查看组件模板的 HTML 时，更容易识别一个符号是组件还是指令。

少数情况下，你要为组件使用属性选择器，比如你要加强某个内置元素时。 比如，Material Design 组件库就会对 <button mat-button> 使用这项技术。不过，你不应该在自定义组件上使用这项技术。

反例：

app/heroes/hero-button/hero-button.component.ts：

@Component({

selector: '[tohHeroButton]',

templateUrl: './hero-button.component.html'

})

export class HeroButtonComponent {}

app/app.component.html：

<div tohHeroButton></div>

正例：

app/heroes/hero-button/hero-button.component.ts：

@Component({

selector: 'toh-hero-button',

templateUrl: './hero-button.component.html'

})

export class HeroButtonComponent {}

app/app.component.html：

<toh-hero-button></toh-hero-button>

1. **把模板和样式从组件中分离**

坚持当超过 3 行时，把模板和样式提取到一个单独的文件。

坚持把模板文件命名为 [component-name].component.html，其中，[component-name] 是组件名。

坚持把样式文件命名为 [component-name].component.css，其中，[component-name] 是组件名。

坚持指定相对于模块的 URL ，给它加上 ./ 前缀。

1. 巨大的、内联的模板和样式表会遮盖组件的意图和实现方式，削弱可读性和可维护性。
2. 在多数编辑器中，编写内联的模板和样式表时都无法使用语法提示和代码片段功能。 Angular 的 TypeScript 语言服务（即将到来）可以帮助那些编辑器在编写 HTML 模板时克服这一缺陷，但对 CSS 样式没有帮助。
3. 当你移动组件文件时，相对于组件的 URL 不需要修改，因为这些文件始终会在一起。
4. “./ ” 前缀是相对 URL 的标准语法，不必依赖 Angular 的特殊处理，如果没有前缀则不行。
5. **内联输入和输出属性装饰器**

坚持 使用 @Input() 和 @Output()，而非 @Directive 和 @Component 装饰器的 inputs 和 outputs 属性。

坚持把 @Input() 或者 @Output() 放到所装饰的属性的同一行。

1. 易于在类里面识别哪些属性是输入属性或输出属性。
2. 如果需要重命名与 @Input 或者 @Output 关联的属性或事件名，你可以在一个位置修改。
3. 依附到指令的元数据声明会比较简短，更易于阅读。
4. 把装饰器放到同一行可以精简代码，同时更易于识别输入或输出属性。

反例：

app/heroes/shared/hero-button/hero-button.component.ts：

[复制代码](javascript:void(0);)

@Component({

selector: 'toh-hero-button',

template: `<button></button>`,

inputs: [

'label'

],

outputs: [

'change'

]

})

export class HeroButtonComponent {

change = new EventEmitter<any>();

label: string;

}

[复制代码](javascript:void(0);)

正例：

app/heroes/shared/hero-button/hero-button.component.ts：

[复制代码](javascript:void(0);)

@Component({

selector: 'toh-hero-button',

template: `<button>{{label}}</button>`

})

export class HeroButtonComponent {

@Output() change = new EventEmitter<any>();

@Input() label: string;

}

[复制代码](javascript:void(0);)

1. **避免为输入和输出属性指定别名**

避免除非有重要目的，否则不要为输入和输出指定别名。

1. 同一个属性有两个名字（一个对内一个对外）很容易导致混淆。
2. 如果指令名也同时用作输入属性，而且指令名无法准确描述这个属性的用途时，应该使用别名。

反例：

app/heroes/shared/hero-button/hero-button.component.ts：

[复制代码](javascript:void(0);)

@Component({

selector: 'toh-hero-button',

template: `<button>{{label}}</button>`

})

export class HeroButtonComponent {

@Output('changeEvent') change = new EventEmitter<any>();

@Input('labelAttribute') label: string;

}

[复制代码](javascript:void(0);)

app/app.component.html：

<toh-hero-button labelAttribute="OK" (changeEvent)="doSomething()">

</toh-hero-button>

正例：

app/heroes/shared/hero-button/hero-button.component.ts：

[复制代码](javascript:void(0);)

@Component({

selector: 'toh-hero-button',

template: `<button>{{label}}</button>`

})

export class HeroButtonComponent {

@Output() change = new EventEmitter<any>();

@Input() label: string;

}

[复制代码](javascript:void(0);)

app/heroes/shared/hero-button/hero-highlight.directive.ts

[复制代码](javascript:void(0);)

import { Directive, ElementRef, Input, OnChanges } from '@angular/core';

@Directive({ selector: '[heroHighlight]' })

export class HeroHighlightDirective implements OnChanges {

@Input('heroHighlight') color: string;

constructor(private el: ElementRef) {}

ngOnChanges() {

this.el.nativeElement.style.backgroundColor = this.color || 'yellow';

}

}

[复制代码](javascript:void(0);)

app/app.component.html：

<toh-hero-button label="OK" (change)="doSomething()">

</toh-hero-button>

<h3 heroHighlight="skyblue">The Great Bombasto</h3>

1. **成员顺序**

坚持把属性成员放在前面，方法成员放在后面。

坚持先放公共成员，再放私有成员，并按照字母顺序排列。

1. 把类的成员按照统一的顺序排列，易于阅读，能立即识别出组件的哪个成员服务于何种目的。

反例：

app/shared/toast/toast.component.ts：

[复制代码](javascript:void(0);)

export class ToastComponent implements OnInit {

private defaults = {

title: '',

message: 'May the Force be with you'

};

message: string;

title: string;

private toastElement: any;

ngOnInit() {

this.toastElement = document.getElementById('toh-toast');

}

// private methods

private hide() {

this.toastElement.style.opacity = 0;

window.setTimeout(() => this.toastElement.style.zIndex = 0, 400);

}

activate(message = this.defaults.message, title = this.defaults.title) {

this.title = title;

this.message = message;

this.show();

}

private show() {

console.log(this.message);

this.toastElement.style.opacity = 1;

this.toastElement.style.zIndex = 9999;

window.setTimeout(() => this.hide(), 2500);

}

}

[复制代码](javascript:void(0);)

正例：

app/shared/toast/toast.component.ts：

[复制代码](javascript:void(0);)

export class ToastComponent implements OnInit {

// public properties

message: string;

title: string;

// private fields

private defaults = {

title: '',

message: 'May the Force be with you'

};

private toastElement: any;

// public methods

activate(message = this.defaults.message, title = this.defaults.title) {

this.title = title;

this.message = message;

this.show();

}

ngOnInit() {

this.toastElement = document.getElementById('toh-toast');

}

// private methods

private hide() {

this.toastElement.style.opacity = 0;

window.setTimeout(() => this.toastElement.style.zIndex = 0, 400);

}

private show() {

console.log(this.message);

this.toastElement.style.opacity = 1;

this.toastElement.style.zIndex = 9999;

window.setTimeout(() => this.hide(), 2500);

}

}

[复制代码](javascript:void(0);)

1. **把逻辑放到服务里**

坚持在组件中只包含与视图相关的逻辑。所有其它逻辑都应该放到服务中。

坚持把可重用的逻辑放到服务中，保持组件简单，聚焦于它们预期目的。

1. 当逻辑被放置到服务里，并以函数的形式暴露时，可以被多个组件重复使用。
2. 在单元测试时，服务里的逻辑更容易被隔离。当组件中调用逻辑时，也很容易被模拟。
3. 从组件移除依赖并隐藏实施细节。
4. 保持组件苗条、精简和聚焦。
5. **不要给输出属性加前缀**

坚持命名事件时，不要带前缀 on。

坚持把事件处理器方法命名为 on 前缀之后紧跟着事件名。

1. 与内置事件命名一致，例如按钮点击。
2. Angular 允许另一种备选语法 on-\*。如果事件的名字本身带有前缀 on，那么绑定的表达式可能是 on-onEvent。

反例：

app/heroes/hero.component.ts：

[复制代码](javascript:void(0);)

@Component({

selector: 'toh-hero',

template: `...`

})

export class HeroComponent {

@Output() onSavedTheDay = new EventEmitter<boolean>();

}

[复制代码](javascript:void(0);)

app/app.component.html：

<toh-hero (onSavedTheDay)="onSavedTheDay($event)"></toh-hero>

正例：

app/heroes/hero.component.ts：

export class HeroComponent {

@Output() savedTheDay = new EventEmitter<boolean>();

}

app/app.component.html：

<toh-hero (savedTheDay)="onSavedTheDay($event)"></toh-hero>

1. **把表现层逻辑放到组件类里**

坚持把表现层逻辑放进组件类中，而不要放在模板里。

1. 逻辑应该只出现在一个地方（组件类里）而不应分散在两个地方。
2. 将组件的表现层逻辑放到组件类而非模板里，可以增强测试性、维护性和重复使用性。

反例：

app/heroes/hero-list/hero-list.component.ts：

[复制代码](javascript:void(0);)

@Component({

selector: 'toh-hero-list',

template: `

<section>

Our list of heroes:

<hero-profile \*ngFor="let hero of heroes" [hero]="hero">

</hero-profile>

Total powers: {{totalPowers}}<br>

Average power: {{totalPowers / heroes.length}}

</section>

`

})

export class HeroListComponent {

heroes: Hero[];

totalPowers: number;

}

[复制代码](javascript:void(0);)

正例：

app/heroes/hero-list/hero-list.component.ts：

[复制代码](javascript:void(0);)

@Component({

selector: 'toh-hero-list',

template: `

<section>

Our list of heroes:

<toh-hero \*ngFor="let hero of heroes" [hero]="hero">

</toh-hero>

Total powers: {{totalPowers}}<br>

Average power: {{avgPower}}

</section>

`

})

export class HeroListComponent {

heroes: Hero[];

totalPowers: number;

get avgPower() {

return this.totalPowers / this.heroes.length;

}

}

[复制代码](javascript:void(0);)

1. **指令**
2. **使用指令来增强已有元素**

坚持当你需要有表现层逻辑，但没有模板时，使用属性型指令。

1. 属性型指令没有模板。
2. 一个元素可以使用多个属性型指令。

范例：

app/shared/highlight.directive.ts：

[复制代码](javascript:void(0);)

@Directive({

selector: '[tohHighlight]'

})

export class HighlightDirective {

@HostListener('mouseover') onMouseEnter() {

// do highlight work

}

}

[复制代码](javascript:void(0);)

app/app.component.html：

<div tohHighlight>Bombasta</div>

1. **HostListener、HostBinding 装饰器 和 组件元数据 host**

考虑优先使用 @HostListener 和 @HostBinding，而不是 @Directive 和 @Component 装饰器的 host 属性。

坚持让你的选择保持一致。

1. 对于关联到 @HostBinding 的属性或关联到 @HostListener 的方法，要修改时，只需在指令类中的一个地方修改。 如果使用元数据属性 host，你就得在组件类中修改属性声明的同时修改相关的元数据。

范例：

app/shared/validator.directive.ts：

[复制代码](javascript:void(0);)

import { Directive, HostBinding, HostListener } from '@angular/core';

@Directive({

selector: '[tohValidator]'

})

export class ValidatorDirective {

@HostBinding('attr.role') role = 'button';

@HostListener('mouseenter') onMouseEnter() {

// do work

}

}

[复制代码](javascript:void(0);)

不推荐的范例：

[复制代码](javascript:void(0);)

import { Directive } from '@angular/core';

@Directive({

selector: '[tohValidator2]',

host: {

'[attr.role]': 'role',

'(mouseenter)': 'onMouseEnter()'

}

})

export class Validator2Directive {

role = 'button';

onMouseEnter() {

// do work

}

}

[复制代码](javascript:void(0);)

1. 服务
2. **服务总是单例的**

坚持在同一个注入器内，把服务当做单例使用。用它们来共享数据和功能。

1. 服务是在特性范围或应用内共享方法的理想载体。
2. 服务是共享状态性内存数据的理想载体。

范例：

app/heroes/shared/hero.service.ts：

[复制代码](javascript:void(0);)

export class HeroService {

constructor(private http: Http) { }

getHeroes() {

return this.http.get('api/heroes').pipe(

map((response: Response) => <Hero[]>response.json()));

}

}

[复制代码](javascript:void(0);)

1. **单一职责**

坚持创建单一职责的服务，用职责封装在它的上下文中。

坚持当服务成长到超出单一用途时，创建一个新服务。

1. 当服务有多个职责时，它很难被测试。
2. 当某个服务有多个职责时，每个注入它的组件或服务都会承担这些职责的全部开销。
3. **提供一个服务**

坚持在服务的 @Injectable 装饰器上指定通过应用的根注入器提供服务。

1. Angular 注入器是层次化的。
2. 当你在根注入器上提供该服务时，该服务实例在每个需要该服务的类中是共享的。当服务要共享方法或状态时，这是最理想的选择。
3. 当你在服务的 @Injectable 中注册服务时，Angular CLI 生产环境构建时使用的优化工具可以进行摇树优化，从而移除那些你的应用中从未用过的服务。
4. 当不同的两个组件需要一个服务的不同的实例时，上面的方法这就不理想了。在这种情况下，对于需要崭新和单独服务实例的组件，最好在组件级提供服务。

范例：

src/app/treeshaking/service.ts：

@Injectable({

providedIn: 'root',

})

export class Service {

}

1. **使用 @Injectable() 类装饰器**

坚持当使用类型作为令牌来注入服务的依赖时，使用 @Injectable() 类装饰器，而非 @Inject() 参数装饰器。

1. Angular 的 DI 机制会根据服务的构造函数参数的声明类型来解析服务的所有依赖。
2. 当服务只接受类型令牌相关的依赖时，比起在每个构造函数参数上使用 @Inject()，@Injectable() 的语法简洁多了。

反例：

app/heroes/shared/hero-arena.service.ts：

export class HeroArena {

constructor(

@Inject(HeroService) private heroService: HeroService,

@Inject(Http) private http: Http) {}

}

正例：

app/heroes/shared/hero-arena.service.ts：

[复制代码](javascript:void(0);)

@Injectable()

export class HeroArena {

constructor(

private heroService: HeroService,

private http: Http) {}

}

[复制代码](javascript:void(0);)

1. **数据服务**
2. **通过服务与 Web 服务器通讯**

坚持把数据操作和与数据交互的逻辑重构到服务里。

坚持让数据服务来负责 XHR 调用、本地储存、内存储存或者其它数据操作。

1. 组件的职责是为视图展示或收集信息。它不应该关心如何获取数据，它只需要知道向谁请求数据。把如何获取数据的逻辑移动到数据服务里，简化了组件，让其聚焦于视图。
2. 在测试使用数据服务的组件时，可以让数据调用更容易被测试（模拟或者真实）。
3. 数据管理的详情，比如头信息、方法、缓存、错误处理和重试逻辑，不是组件和其它的数据消费者应该关心的事情。
4. 数据服务应该封装这些细节。这样，在服务内部修改细节，就不会影响到它的消费者。并且更容易通过实现一个模拟服务来对消费者进行测试。
5. **生命周期钩子**
6. **实现生命周期钩子接口**

坚持实现生命周期钩子接口。

1. 如果使用强类型的方法签名，编译器和编辑器可以帮你揪出拼写错误。

反例：

app/heroes/shared/hero-button/hero-button.component.ts：

[复制代码](javascript:void(0);)

@Component({

selector: 'toh-hero-button',

template: `<button>OK<button>`

})

export class HeroButtonComponent {

onInit() { // misspelled

console.log('The component is initialized');

}

}

[复制代码](javascript:void(0);)

正例：

app/heroes/shared/hero-button/hero-button.component.ts：

[复制代码](javascript:void(0);)

@Component({

selector: 'toh-hero-button',

template: `<button>OK</button>`

})

export class HeroButtonComponent implements OnInit {

ngOnInit() {

console.log('The component is initialized');

}

}

[复制代码](javascript:void(0);)