# 在 Angular 项目中实现 Data Mocking 的几种方式

2021 年 7 月 3 日

**摘要**

在这篇文章中，我们会介绍几种基于 Angular 的一些特性实现的 Data Mocking 方法，通过这些方法，我们可以尽可能真实地模拟后端的开发环境，实现更加方便的前端开发、测试环境。具体地，我们会分别介绍：1) 基于 IoC 设计模式和 DI 技术的 Data Mocking; 2) 基于 HttpInterceptor 的 Data Mocking; 3) 基于工厂模式的配置切换方法。这些方法，解决了如何在 Angular 中灵活优雅地实现 Data Mocking 的问题，它的意义在于：为前端开发人员提供了一种可依赖的开发测试方式。

## 1 引言

所谓 Data Mocking，我们指的是，在前端以各种方式模拟后端，其中最主要的，就是模拟后端的响应（返回值），也能模拟一些简单的后端交互逻辑。通过这样的模拟 (Simulation), 我们能提前知道我们写的程序到底是不是对的，我们能提前知道，我们有没有正确地处理后端返回的数据，以及**假如说**后端返回了正确的数据，我们能否做正确的处理。这些问题的答案很重要，因为当出现问题时，这些问题的答案能帮助我们快速定位问题。

以上说的是简单的模拟，它得出的结果回答了「当后端按正常情况返回数据时，前端的表现是否也正常」的问题，Data Mocking 也能做一些复杂的模拟，比如说模拟各种边界情况，对于一个按照约定本应是字符串的字段，我们可以模拟后端返回一个数字，一个数组或者一个布尔值，甚至模拟这种字段缺失的情况，以此来测试前端对于异常情况的处理如何，或者说前端的鲁棒性如何。压力测试（边界测试）实际上也属于这一类模拟，只不过后端返回的数据都是正常的，但是数据量远远地超出了正常的范围。

往大了说，不管是前端也好，后端也好，在开发过程中，进行足够多的测试是有必要的，这就好比人们会通过打预防针接种疫苗，来获得对一些疾病的免疫能力，这其实就相当于软件开发中的自测，它不能够解决所有的问题，但是做了自测肯定比没做要好。

## 2 控制反转

控制反转 (Inversion of Control, IoC) 是一种开发实践，简单来说，在按照 IoC 思想设计的软件架构中，程序执行的控制权在框架，开发人员要做的，只是写好各个任务模块的代码，而框架则会在需要的时候去调用它们，这于传统的过程式编程的控制流是相反的——在传统的控制流

中，程序员要自己决定先执行什么、后执行什么，而在按照 IoC 思想设计的的软件中，是框架去按需执行各个部分。

## 2.1 依赖注入

依赖注入 (Dependency Injection) 是一种设计模式，它实现了 IoC 开发实践。在 DI 设计模式中，有一个 IoC 容器负责初始化一个对象的各种依赖，开发人员只需把这种依赖声明出来。举例来说，当我们在 Angular 项目中需要发起 Http 请求时，有可能会这样写：

```
export class HeroListComponent implements OnInit {
  constructor(private http: HttpClient) {}

  ngOnInit(): void {
    this.http.get('/api/v1/heroes').subscribe(data => {
      // ...
    });
  }
}
```

这里，`HttpClient` 是一个 Injection Token, 我们只是在构造器中告诉 IoC 容器：我们需要一个 HttpClient, 你去找吧，找来了顺便帮我实例化成 `http` 并且注入到当前 `HeroListComponent` 中供我使用。我们从前到后从来没有自己 `new` 过这个 `HttpClient`, 这都是 IoC 容器帮我们做的。

如果觉得这个例子还是有些不理解的话，且看下一个例子：假设这回我们不打算在 `Component` 里边做，而是希望将获取数据这个过程，交给 Service 来做，在此之前，我们先定好接口，也就是约定 Service 应该给我们获取到什么样的数据：

```
/** 英雄接口 */
export interface IHero {
    /** 英雄的 ID */
    id: number;

    /** 英雄的名字 */
    name: string;
}

/** 英雄数据查询接口 */
export interface IHeroQueryResult {
    /** 总数 */
    totalCounts: number;

    /** 当前页查询结果 */
```

```
16     heroes: IHero[];
17 };
```

好了,这样 Service 中实现的数据查询函数只要返给我们关于 `IHeroQueryResult` 的 `Observable` 就可以了，加载的过程是标准化的:

```
1 export class HeroListComponent implements OnInit {
2
3     // ...
4
5     /**
6      * 将英雄载入视图
7      * @param data 查询到的英雄列表数据
8      */
9     load(data: IHeroQueryResult): void {
10        // ...
11    }
12 }
```

我们首先对这样一个实现英雄查询功能的 Service 做个约定:

```
1 /**
2  * 抽象英雄数据服务，由 Component 依赖，由直属 Module 负责提供相应的实现。
3  */
4 @Injectable()
5 export abstract class HeroDataService {
6
7     /**
8      * 查询英雄数据
9      */
10    abstract getHeroes(): Observable<IHeroQueryResult>;
11
12 }
```

然后我们做两个实现，一个是 Mock 的:

```
1 /** 此服务负责产生 Mock 的英雄数据列表 */
2 @Injectable()
3 export class MockHeroDataService {
4   constructor() {}
5
6   /**
```

```
   * 产生 Mock 的英雄数据列表
   * @returns {Observable<IHeroQueryResult>} 英雄列表查询结果
   */
  getHeroes(): Observable<IHeroQueryResult> {
    return of({
        totalCounts: 2,
        heroes: [
            { id: 0, name: 'mockHero0', },
            { id: 1, name: 'mockHero1', },
        ],
    });
  }
}
```

一个是真正向后端发起 Http 请求的:

```
/** 服务器返回的参数 */
type ServerHeroQueryReturn = {
  /** 英雄表总记录数 */
  total_counts: number;

  /** 英雄列表 */
  results: {

    /** 英雄 ID */
    hero_id: number;

    /** 英雄名称 */
    hero_name: string;
  }[];
}

/**
 * 此服务负责以 HTTP 方式向后端请求英雄列表数据，实现了 HeroDataService
 */
@Injectable()
export class HttpHeroDataService implements HeroDataService {

  constructor(
```

```
    private httpClient: HttpClient,
  ) {}

  /**
   * 以 HTTP 协议向后端请求英雄列表数据，接受分页参数
   * @returns {Observable<IHeroQueryResult>} 英雄查询结果
   */
  getHeroes(): Observable<IHeroQueryResult> {
    const apiPath = '/api/v1/heroes';
    return this.httpClient.get<ServerHeroQueryReturn>(apiPath).pipe(
      map(serverReturn => ({
        totalCounts: serverReturn.total_counts,
        heroes: serverReturn.results.map(result => ({
          id: result.hero_id, name: result.hero_name,
        })),
      })),
    );
  }
}
```

在 Module 定义文件中，我们可以这样写：

```
@NgModule({
  imports: [
    CommonModule,
    HeroRoutingModule,
  ],
  providers: [
    {
      provide: HeroDataService, useClass: MockHeroDataService,
    }
  ],
})
export class HeroModule {}
```

也可以这样写：

```
@NgModule({
  imports: [
    CommonModule,
    HeroRoutingModule,
```

```
  ],
  providers: [
    {
      provide: HeroDataService, useClass: HttpHeroDataService,
    }
  ],
})
export class HeroModule {}
```

这样对于在 `HeroListComponent` 中出现的 `HeroDataService` 这个 Injection Token, R3 (Angular 负责实现依赖注入的模块) 会去 HeroModule 中查找到底是使用 `HttpHeroDataService` 的实例来提供，还是使用 `MockHeroDataService` 的实例来提供：

```
export class HeroListComponent implements OnInit {

    constructor(private heroDataSerivce: HeroDataService) {}

    ngOnInit(): void {
        this.heroDataSerivce.getHeroes().subscribe(heroQueryResult => {
            this.load(heroQueryResult);
        });
    }

    load(queryResult: IHeroQueryResult): void {
        // ...
    }
}
```

如果你觉得这个例子太过复杂，我们可以来看两个简单的例子，假设我们的产品有两个名称，一个内部名称，和一个外部名称，那么我们可以这样先定义一个 InjectionToken，以及写好两个产品名称：

```
/** Logo 注入标识 */
export const LOGO = new InjectionToken<string>('Logo');

/** 内部使用 Logo */
export const INTERNAL_LOGO = 'Project 8';

/** 产品 Logo */
export const PROD_LOGO = 'ICar';
```

我们会在 Layout 中依赖它：

```
export class LayoutComponent implements OnInit {

    constructor(
        @Inject(LOGO) private logo: string,
    ) { }

    ngOnInit(): void {
        this.headerLogoText = this.logo;
    }

}
```

当然，为了让 R3 能找到 LOGO 这个 Injection Token 的「提供者」，还需要在 Module 中显式声明：

```
/** 此 Module 负责导出一个实现全局 Layout 的组件 */
@NgModule({
    declarations: [LayoutComponent],
    imports: [CommonModule, RouterModule],
    exports: [LayoutComponent],
    providers: [
        {
            provide: LOGO,
            useValue: INTERNAL_LOGO,
        },
    ],
})
export class LayoutModule {}
```

这样，LayoutComponent 依赖的那个 LOGO 就会被 R3 用 INTERNAL_LOGO 注入。

# 3 拦截器

Angular 的 http 模块提供了 HTTP_INTERCEPTORS 这个 Injection Token, 它被定义为：

```
export declare const HTTP_INTERCEPTORS: InjectionToken<HttpInterceptor[]>;
```

我们可以自己实现想要的 HttpInterceptor 提供给 HTTP_INTERCEPTORS 来实现 Http 请求拦截功能，这样做，可以在请求发出之前修改请求参数，也可以在请求返回以后修改返回结果。

现在让我们来实现一个简单的 HttpInterceptor, 它拦截符合条件的请求，并且返回 404：

```
@Injectable()
```

```
export class MockHeroResponseInterceptor implements HttpInterceptor {
    intercept(
        req: HttpRequest<unknown>,
        next: HttpHandler
    ): Observable<HttpEvent<unknown>> {
    if (request.url !== '/api/v1/heroes') {
        return next.handle(request);
    }

    return of(new HttpResponse({
        status: 404, body: 'Not Found',
    }));
  }
}
```

通过在 Hero 模块中以这种方式提供 `HttpInterceptor`, 我们可以拦截发往 `/api/v1/heroes` 的请求，并且无条件返回 404:

```
@NgModule({
    declarations: [HeroListComponent],
    imports: [CommonModule, HttpClientModule, HeroRoutingModule],
    providers: [
        {
            provide: HTTP_INTERCEPTORS,
            useClass: MockHeroResponseInterceptor,
            multi: true,
        },
    ],
})
export class HeroModule {}
```

注意到 `multi: true`, 这是因为 `HTTP_INTERCEPTORS` 是一个 `HttpInterceptor[]`, 可以以这种方式被多次提供, 也就是说, 我们可以实现多个 `HttpInterceptor`, 并且都提供给 `HTTP_INTERCEPTORS`, 使得每个 `HttpInterceptor` 都生效。

# 4 工厂模式

现在是，我们在线上和开发测试中使用两套后端，我们希望能实现自动切换。具体地，我们在一个配置文件中定义好后端地址:

```
export type HttpBackendConfig = {
```

```
    server: string;
    path: string;
};

export const HTTP_BACKEND = new InjectionToken<HttpBackendConfig>('Backend');

export const PROD_HTTP_BACKEND: HttpBackendConfig = {
    server: 'https://prod.example.com',
    path: '/api/v1/heroes',
};

export const DEV_HTTP_BACKEND: HttpBackendConfig = {
    server: 'http://localhost:8000',
    path: '/api/v1/heroes',
};
```

假设已经有了一个服务，它能够判断程序当前是运行在开发环境还是生产环境：

```
@Injectable()
export class EnvProbService {

    // ...

    getEnv(): 'dev' | 'prod' | 'unknown' {
        // ...
    }
}
```

那么我们就可以让程序自动地决定应该使用哪个后端：

```
@NgModule({
  declarations: [HeroListComponent],
  imports: [CommonModule, HeroRoutingModule],
  providers: [
    {
        provide: HTTP_BACKEND,
        useFactory: (envProbService: EnvProbService): HttpBackendConfig => {
            const currentEnv = envProbService.getEnv();
            if (currentEnv === "dev") {
                return DEV_HTTP_BACKEND;
            } else if (currentEnv === "prod") {
```

```
12                 return PROD_HTTP_BACKEND;
13             } else {
14                 return DEV_HTTP_BACKEND;
15             }
16         },
17       deps: [EnvProbService],
18     },
19     EnvProbService,
20   ],
21 })
22 export class HeroModule {}
```

当有组件或者服务要用到 `HTTP_BACKEND` 这个 Injection Token 的时候，R3 就会用工厂生产一个 `HttpBackendConfig` 来提供，而这个工厂的原材料则是一个 `EnvProbService`.