在 Angular 项目中使用装饰器来模拟请求延迟和请求进度更新

2021 年 8 月 28 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 TypeScript 装饰器模式的请求延迟模拟方法；2) 基于 TypeScript 装饰器模式和 RxJS 库的请求进度更新模拟方法；这些方法，解决了如何在 Angular 中实现请求延迟模拟和进度更新模拟的问题，它的意义在于：为前端开发人员提供了一种更加真实的数据模拟和 Demo 模拟后端环境的搭建方法，并且，灵活地运用装饰器，能够使得项目中的代码的重复程度得到减轻，使得 DRY 原则 (Dont't Repear Yourself) 得到更好的贯彻。

1 引言

当我们想要开发一个 Demo 时，我们需要实现这个 Demo 的后端，这个时候有两种选项：1）和后端约定接口，让专业后端工程师去做专业的事情；2）我们前端利用前端框架中的请求拦截机制（Vue 和 Angular 都有）去模拟一个简单的后端。在本文中，我们采用后一种方法，即我们前端自己模拟一个简单的后端。

这带来另外一个好处就是，模拟后端的实现和前端的实现是非常同步的，由于都是一个人做，沟通成本和协调成本为 0, 我们能够根据 Demo 业务逻辑灵活自由的随时实现和更新这个「假」后端来达到我们想要的效果：这其中就包括，我们可以模拟请求的延迟和进度更新，更具体地举个例子：我们可以模拟请求一张表格的某一页数据的延迟为两秒钟，并且可以模拟一个请求的进度，比如说在某时某刻请求完成了 80%, 大概还有 20% 的进度等等。

2 装饰器

对于有 Python 背景，或者是函数式开发背景的同学来说，装饰器是不陌生的，简单来说，当一个编程语言把函数当成「一等公民」之后，函数就和对象没什么差别，函数某种程度上作为一个原子或者一个独立单元或者一个独立的对象，可以作为参数传给另外一个函数，也可以由某个函数返回，简单来说就是函数的「一等公民」身份使得高阶函数的实现（函数的函数）变得容易。

更通俗来说，一个装饰器就是一个关于函数的函数，这个函数的输入是一个函数，输出也是一个函数。

在 TypeScript 中，相比 Python，所谓「装饰器」更多强调的是「装饰」，而不是向 Python 那样作为一个高阶函数的存在。在 TypeScript 中，一个装饰器是一个函数，它接受三个参数，第一个是 `target`，如果被装饰的成员是 `static` 的，那么 `target` 就是构造器函数，否则就是类的原型；第二个是 `propertyKey`，也就是成员/属性的名称，比如被修饰的一个成员叫做 `getAge`，那么 `propertyKey` 的值就是 `"getAge"`；第三个参数是 `descriptor`，这个 `descriptor` 和 `Object.defineProperty` 中的 `descriptor` 参数是一模一样的。你还可以使用范型 `TypedPropertyDescriptor<T>` 来作为 `descriptor` 的类型，这样就相当于可以为 `descriptor` 的值指定类型 `T`.

下面我们通过一个具体的例子，来说明装饰器怎么用。

2.1 给返回的 Observable 增加延迟

假设我们有一个 Angular Component 叫做 `UserComponent` 希望通过 `MockedUserService` 来获取模拟的用户数据，`MockedUserService` 的 `getUsers` 方法返回 `Observable<IUserQueryResult>`，一种办法是，去使用 RxJS 提供的 `delay` 算子：

```
export class MockedUserService {
  getUsers(): Observable<IUserQueryResult> {
    // 获得了 users$

    return users$.pipe(delay(2000));
  }
}
```

通过这种方法可以获得 2000 的模拟延迟。

另外一种方法是，我们去定义一个装饰器，把增加延迟的逻辑和在 `MockedUserService` 的方法里边实现的业务逻辑解耦，装饰器这样调用：

```
export class MockedUserService {
  @IncreaseDelay(2000)
  getUsers(): Observable<IUserQueryResult> {
    // 获得了 users$

    return users$;
  }
}
```

我们希望这个 `IncreaseDelay` 能够带来的效果是 `getUsers` 返回的东西延迟指定的时长再触发，那么这个装饰器可以这样子实现：

```
type ObservableFn<T> = (...args: any[]) => Observable<T>;
```

```
2
3 function IncreaseDelay<T>(delayMS: number) {
4   return (
5     target: any,
6     propertyKey: string,
7     descriptor: TypedPropertyDescriptor<ObservableFn<T>>,
8   ) => {
9     const originalFn = descriptor.value as ObservableFn<T>;
10
11    descriptor.value = function (...args: any[]): Observable<T> {
12      return originalFn.call(this, ...args).pipe(delay(delayMS));
13    };
14  };
15 }
```

通过修改 `descriptor.value`, 我们把旧的函数换成了新的，新的函数会在执行时调旧的函数，并且对于旧的函数的返回值做 `delay` 处理。

除此之外，这个 `IncreaseDelay` 还是可复用的，并且是和业务逻辑分开的，但凡一个成员函数返回的是 `Observable` 并且需要增加这个 `Observable` 的地方，都可以用这个装饰器。

2.2 模拟一个假的进度条

在现实情况中，网络连接往往是不稳定的，这和网络链路是通过 4G, 5G, WiFi 还是光纤无关。由此会造成请求的延时会不稳定，比如说请求一个很短的列表可能要 2, 3 秒，而一个几千行的列表有时候也可能比较快比如说 1 秒不到。

类似于一种「安慰剂」的浏览器视图顶部进度条现如今被许多网站采用，并且在细节上各自有着不同的实现方式。本文不讨论这样一种视图顶栏进度条是如何具体实现的，我们把它当成一个组件，把它当成一个接受一个 $[0, 1]$ 内的实数，并且输出是相应显示效果的 UI 组件就可以了。

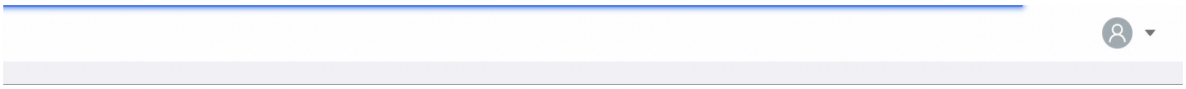

图 1: 注意顶栏上面蓝色细长的进度条

这种假进度条能够一定程度上缓解用户在等待内容加载期间产生的焦虑感。

那么，现在既然把这个进度条抽象成了一个可调用的组件，我们接下来只需在每时每刻计算一下当前最新进度值就可以了。同样地，我们可以将它与业务逻辑耦合在一起实现：

```
export class MockedUserService {
  getUsers(): Observable<IUserQueryResult> {
    // 获得了 users$

    // 每 100 毫秒计算一次进度
    const tickPeriodMS = 100;

    // 预估一个请求需要 2 秒钟响应
    const estimatedResponseDelayMS = 2000;

    // 记录下开始请求时间点
    const startPoint = new Date().valueOf();

    // 记录下进度更新订阅，当请求完成后取关
    let tickSubscription = interval(tickPeriodMS).subscribe(() => {

      // 计算请求已经等待的时间
      const elapsed = new Date().valueOf() - startPoint;

      // 计算进度
      let progress = elapsed / estimatedResponseDelayMS;

      // 当进度条超过 0.9 后减慢速度
      if (progress > 0.9) {
        const slowFactor = 100;
        progress = progress + (progress - 0.9) / slowFactor;
      }

      // 如果估计时间小于实际时间则会出现这种情况
      if (progress > 1) {
        // 这时我们要将进度修正到正确值
        progress = 1;
      }

      // 广播进度条进度，好让组件接收到
      progress$.next({ showProgressBar: true, progress });
    })

```

```
39      // 当请求完成时
40      return users$.pipe(delay(2000)).pipe(tap(() => {
41        // 停止进度刷新
42        tickSubscription.unsubscribe();
43
44        // 进度推至 100%
45        progress$.next({ showProgressBar: true, progress: 1 });
46
47        // 500 毫秒后关闭进度条
48        timer(500).subscribe(() => {
49          progress$.next({ showProgressBar: false });
50        });
51      }));
52    }
53 }
```

经过上面这一段代码,我们基本能够做出一个假的进度条,但问题是,难道每当我们需要 Mock 一个接口，就需要将上面这些代码再复制粘贴一遍吗?

于是装饰器出场了，装饰器可以解决这里遇到的代码复用问题:

```
1 /**
2  * 进度更新参数类型
3  *
4  * @property {boolean} showProgressBar - 是否展示进度条
5  * @property {number} progress - 进度（范围 [0, 1] 内的浮点数）
6  */
7 type IProgressOption = { showProgressbar: boolean, progress: number }
8
9 /**
10  * 进度更新装饰器，用在一个返回 Observable 的成员上
11  *
12  * @param {number} estimatedDelayMS - 预估请求延迟（单位：毫秒）
13  * @param {Subject<IProgressOption>} progress$ - 用于接受进度更新的 Subject
14  */
15 function ProgressUpdateSimulation(
16   estimatedDelayMS: number,
17   progress$: Subject<IProgressOption>,
18 ) {
19
```

```
  // 返回一个装饰器
  return (
    target: any,
    propertyKey: string,
    descriptor: TypedPropertyDescriptor<ObservableFn>,
  ) => {
    const originalFn = descriptor.value;
    if (!originalFn) {
      return;
    }

    // 用新函数代替原来的旧函数，并且当新函数被调用时:
    descriptor.value = function (...args: any[]): Observable<any> {

      // 每 100 毫秒计算一次进度
      const tickPeriodMS = 100;

      // 预估一个请求需要 2 秒钟响应
      const estimatedResponseDelayMS = 2000;

      // 记录下开始请求时间点
      const startPoint = new Date().valueOf();

      // 记录下进度更新订阅，当请求完成后取关
      let tickSubscription = interval(tickPeriodMS).subscribe(() => {

        // 计算请求已经等待的时间
        const elapsed = new Date().valueOf() - startPoint;

        // 计算进度
        let progress = elapsed / estimatedResponseDelayMS;

        // 当进度条超过 0.9 后减慢速度
        if (progress > 0.9) {
          const slowFactor = 100;
          progress = progress + (progress - 0.9) / slowFactor;
        }

```

```
58          // 如果估计时间小于实际时间则会出现这种情况
59          if (progress > 1) {
60            // 这时我们要将进度修正到正确值
61            progress = 1;
62          }
63
64          // 广播进度条进度，好让组件接收到
65          progress$.next({ showProgressBar: true, progress });
66        })
67
68        // 当请求完成时
69        return originalFn.call(this, ...args).pipe(tap(() => {
70          // 停止进度刷新
71          tickSubscription.unsubscribe();
72
73          // 进度推至 100%
74          progress$.next({ showProgressBar: true, progress: 1 });
75
76          // 500 毫秒后关闭进度条
77          timer(500).subscribe(() => {
78            progress$.next({ showProgressBar: false });
79          });
80        }));
81
82      };
83    };
84  }
```

我们只需这样使用即可：

```
const progress$ = new Subject<IProgressOption>();

export class MockedUserService {
  progressSubscription?: Subscription;

  constructor(private progressBarService: ProgressBarService) {}

  ngOnInit(): void {
    this.progressSubscription = progress$.subscribe((progress) =>
      this.progressBarService.update(progress)
    );
  }

  @ProgressUpdateSimulation(2000, progress$)
  @IncreaseDelay(2000)
  getUsers(): Observable<IUserQueryResult> {
    // ...
  }

  ngOnDestroy(): void {
    this.progressSubscription?.unsubscribe();
  }
}
```

另外要注意装饰器的添加顺序，先用 `@IncreaseDelay(2000)` 去装饰 `getUsers` 可以给 `getUsers` 返回的 `Observable` 增加 `2000` 毫秒的延时，而再 `ProgressUpdateSimulation(2000)` 去装饰则是能够在此基础上去模拟等待过程中的进度更新。