一文读懂 @Decorator 装饰器——理解 VS Code 源码的基础

原创 easonruan 腾讯技术工程 2021-08-06 18:02

作者：easonruan，腾讯 CSIG 前端开发工程师

1. 装饰器的样子

我们先来看看 Decorator 装饰器长什么样子，大家可能没在项目中用过 Decorator 装饰器，但多多少少会看过下面装饰器的写法：

```
/* Nest.Js cats.controller.ts */
import { Controller, Get } from '@nestjs/common';

@Controller('cats')
export class CatsController {
  @Get()
  findAll(): string {
    return 'This action returns all cats';
  }
}
```

> 摘自《Nest.Js》官方文档

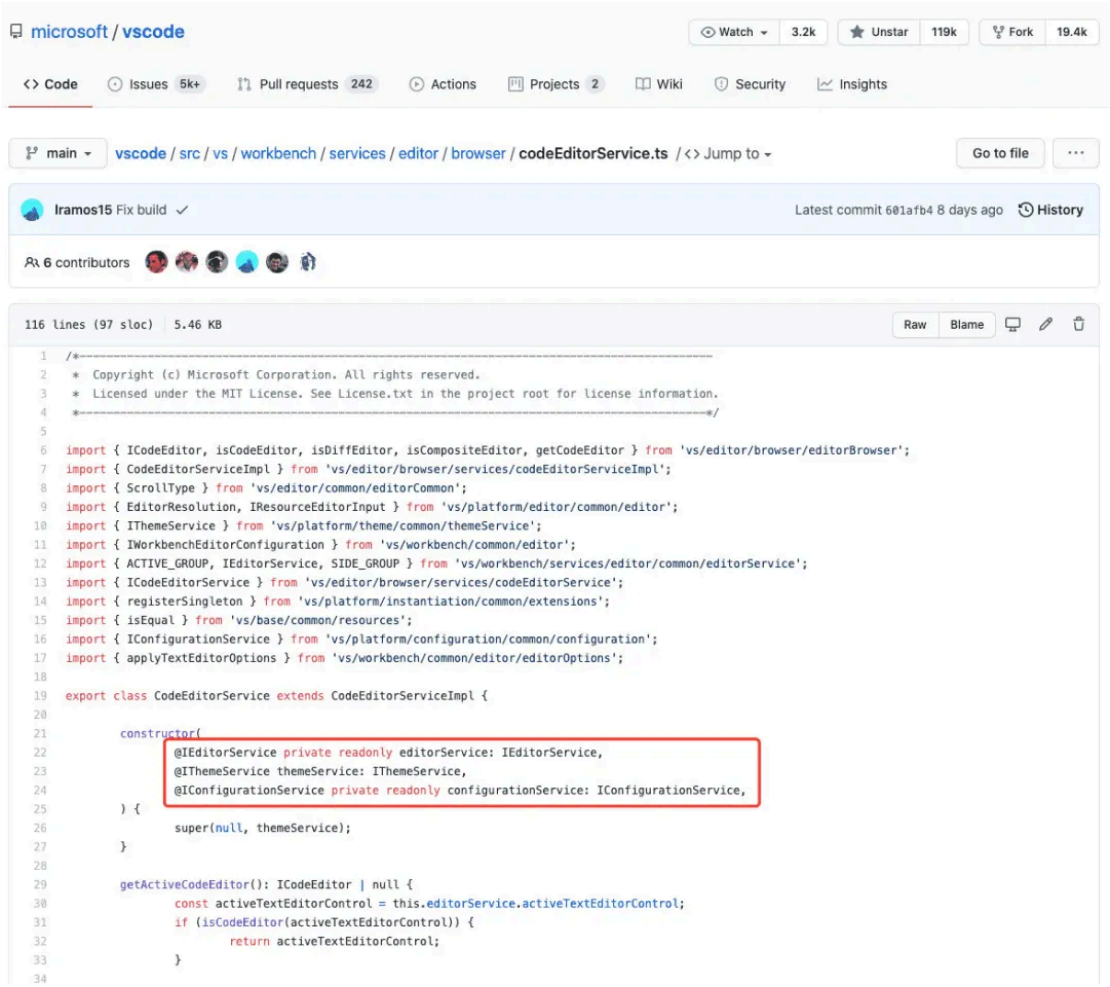
上述代码大家可以不着急去理解，主要是让大家对装饰器有一个初步了解，后面我们会逐一分析 Decorator 装饰器的实现原理以及具体用法。

2. 为什么要理解装饰器

2.1 浅一点来说，理解才能读懂 VS Code 源码

Decorator 装饰器是 **ECMAScript** 的语言提案，目前还处于 **stage-2** 阶段，但是借助 TypeScript 或者 Babel，已经有大量的优秀开源项目深度用上它了，比如：VS Code, Angular, Nest.Js(后端 Node.js 框架), TypeORM, Mobx(5) 等等

举个例子：
https://github.com/microsoft/vscode/blob/main/src/vs/workbench/services/editor/browser/codeEditorService.ts#L22



作为一个有追求的程序员，你可能会问：上面代码的装饰器代表什么含义？去掉装饰器后能不能正常运行？

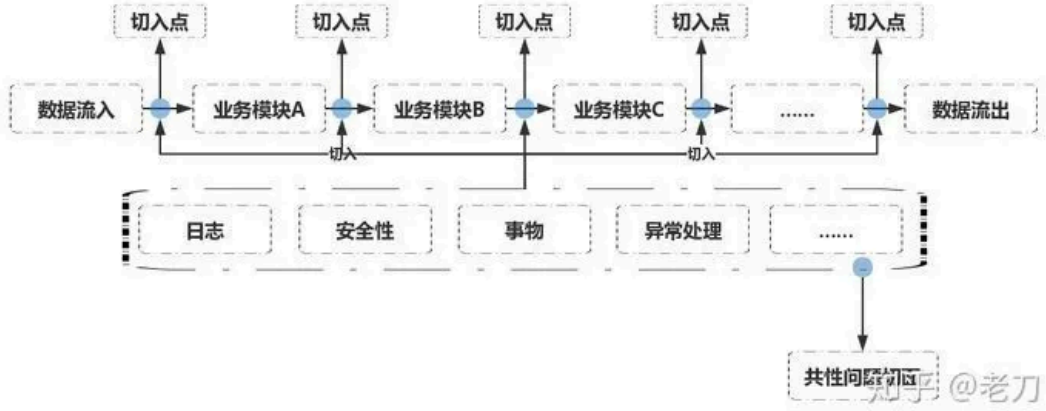
如果没弄懂装饰器，很难读懂 VS Code 这些优秀项目源码的核心思想。所以说你不需要熟练使用装饰器，但一定要理解装饰器的用法。

2.2 深一点来说，理解才能弄懂 AOP , IoC, DI 等优秀编程思想

1.AOP 即面向切面编程 (Aspect Oriented Programming)

AOP 主要意图是将日志记录，性能统计，安全控制，异常处理等代码从业务逻辑代码中划分出来，将它们独立到非指导业务逻辑的方法中，进而改变这些行为的时候不影响业务逻辑的代码。

简而言之，就是“优雅”的把“辅助功能逻辑”从“业务逻辑”中分离，解耦出来。

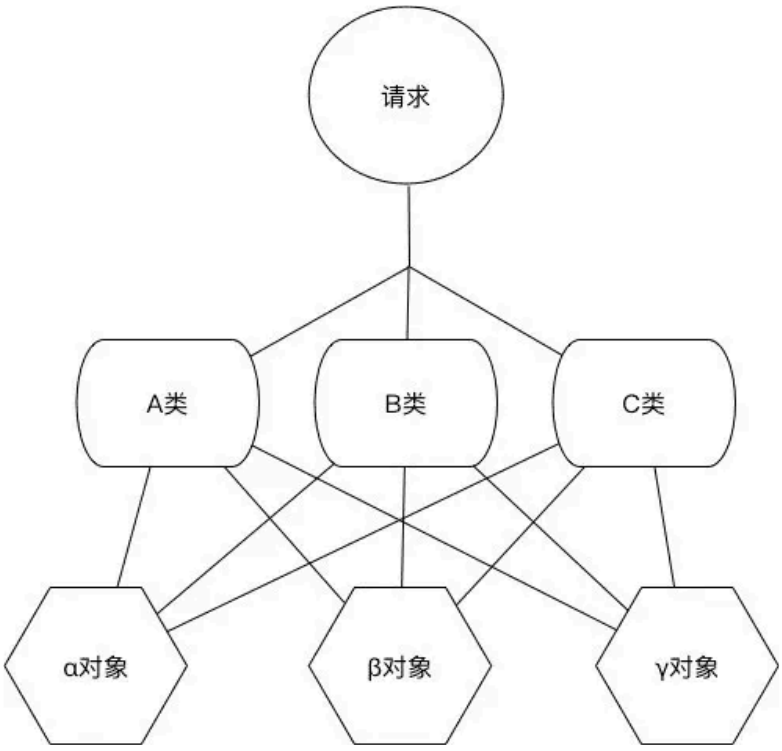


> 图摘自《简谈前端开发中的 AOP(一) -- 前端 AOP 的实现思路》

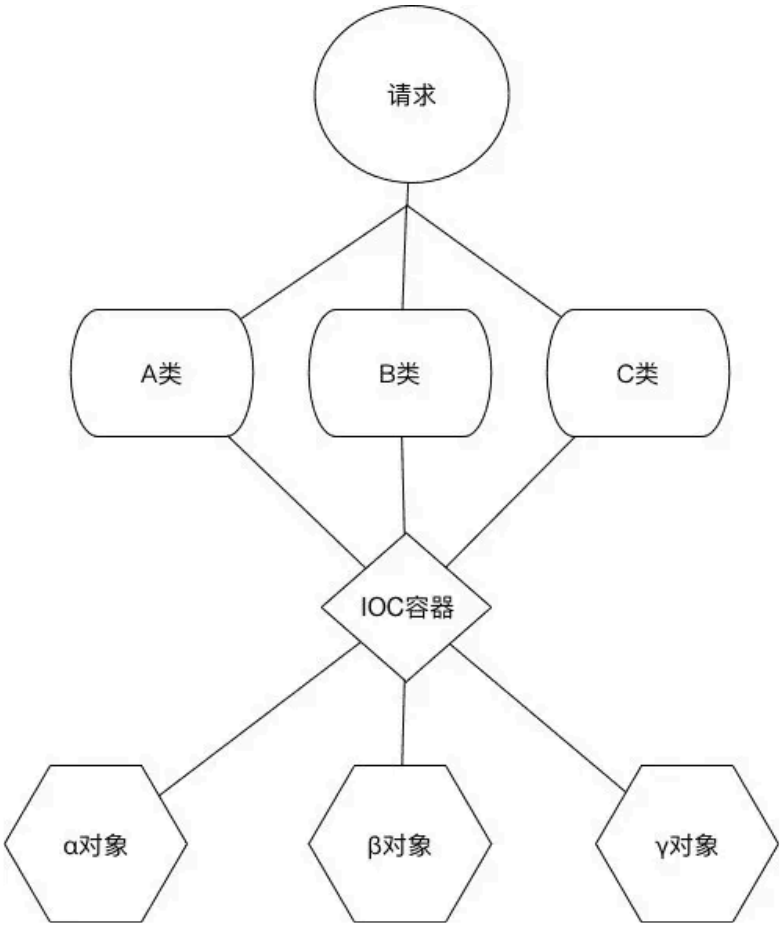
2.IoC 即 控制反转 (Inversion of Control)，是解耦的一种设计理念

3.DI 即 依赖注入 (Dependency Injection)，是 IoC 的一种具体实现

使用 IoC 前：



使用 IoC 后：



> 图摘自**《两张图让你理解 IoC (控制反转)》**

IoC 控制反转的设计模式可以大幅度地降低了程序的耦合性。而 Decorator 装饰器在 VS Code 的控制反转设计模式里，其主要作用是实现 DI 依赖注入的功能和精简部分重复的写法。由于该步骤实现较为复杂，我们先从简单的例子为切入点去了解装饰器的基本原理。

3. 装饰器的概念区分

在理解装饰器之前，有必要先对装饰器的 3 个概念进行区分。

3.1 Decorator Pattern (装饰器模式)

是一种**抽象的设计理念**，核心思想是**在不修改原有代码情况下，对功能进行扩展。**

3.2 Decorator (装饰器)

是一种**特殊的装饰类函数**，是一种对装饰器模式理念的具体实现。

3.3 @Decorator (装饰器语法)

是一种**便捷的语法糖**(写法)，通过 @ 来引用，需要编译后才能运行。理解了概念之后可以知道：**装饰器的存在就是希望实现装饰器模式的设计理念。**

说法 1：**在不修改原有代码情况下，对功能进行扩展。**也就是对扩展开放，对修改关闭。

说法 2：**优雅地把“辅助性功能逻辑”从“业务逻辑”中分离，解耦出来。**（AOP 面向切面编程的设计理念）

4. 装饰器的实战：记录函数耗时

现在有一个 关羽(GuanYu) 类，它有两个函数方法： attack(攻击) 和 run(奔跑)

```
class GuanYu {
  attack() {
    console.log('挥了一次大刀')
  }
  run() {
    console.log('跑了一段距离')
  }
}
```

而我们都是优秀的程序员，时时刻刻都有着经营思维 (性能优化)，因此想给 关羽(GuanYu) 的函数方法提前做好准备：记录关羽的每一次 attack(攻击) 和 run(奔跑) 的执行时间，以便于后期做性能优化。

4.1 做法一：复制粘贴，不用思考一把梭就是干

拿到需求，不用多想，立刻在函数前后，添加记录函数耗时的逻辑代码，并复制粘贴到其他地方：

```
class GuanYu {
  attack() {
+   const start = +new Date()
    console.log('挥了一次大刀')
+   const end = +new Date()
+   console.log(`耗时: ${end - start}ms`)
  }
  run() {
+   const start = +new Date()
    console.log('跑了一段距离')
+   const end = +new Date()
+   console.log(`耗时: ${end - start}ms`)
  }
}
```

但是这样直接修改原函数代码有以下几个问题：

1. **理解成本高**

统计耗时的相关代码与函数本身逻辑并无关系，对函数结构造成了破坏性的修改，影响到了对原函数本身的理解

2. **维护成本高**

如果后期还有更多类似的函数需要添加统计耗时的代码，在每个函数中都添加这样的代码非常低效，也大大提高了维护成本

4.2 做法二：装饰器模式，不修改原代码扩展功能

4.2.1 装饰器前置基础知识

在开始用装饰器实现之前必须掌握以下基础：

1. **Object.getOwnPropertyDescriptor()**

返回指定对象上一个自有属性对应的**属性描述符**

```
var a = { b: () => {} }
var descriptor = Object.getOwnPropertyDescriptor(a, 'b')
console.log(descriptor)
/**
 * {
 *   configurable: true,  // 可配置的
 *   enumerable: true,    // 可枚举的
 *   value: () => {},     // 该属性对应的值（数值，对象，函数等）
 *   writable: true,      // 可写入的
 * }
 */
```

这里要注意一个点是：value 可以是 JavaScript 的任意值，比如函数方法，正则，日期等

2. **Object.defineProperty()**

在一个对象上定义或修改一个属性的描述符：

```
const object1 = {};

Object.defineProperty(object1, 'property1', {
  value: 'ThisIsNotWritable',
  writable: false
});

object1.property1 = 'newValue';
// throws an error in strict mode

console.log(object1.property1);
// expected output: 'ThisIsNotWritable'
```

4.2.2 【重点】手写一个装饰器函数

有了上面的两个基础后，我们开始利用装饰器模式的设计理念，用纯函数的形式写一个装饰器，实现记录函数耗时功能。为了让大家更深刻理解装饰器的原理，我们先不用 @Decorator 这个语法糖。

下面代码是本文的重点，大家可以放慢阅读速度，理解后再继续往下看：

```
// 装饰器函数
function decoratorLogTime(target, key) {
  const targetPrototype = target.prototype
  // Step1 备份原来类构造器上的属性描述符 Descriptor
  const oldDescriptor = Object.getOwnPropertyDescriptor(targetPrototype, key)

  // Step2 编写装饰器函数业务逻辑代码
  const logTime = function (...arg) {
    // Before 钩子
    let start = +new Date()
    try {
      // 执行原来函数
      return oldDescriptor.value.apply(this, arg) // 调用之前的函数
    } finally {
      // After 钩子
      let end = +new Date()
      console.log(`耗时: ${end - start}ms`)
    }
  }

  // Step3 将装饰器覆盖原来的属性描述符的 value
  Object.defineProperty(targetPrototype, key, {
    ...oldDescriptor,
    value: logTime
  })
}


class GuanYu {
  attack() {
    console.log('挥了一次大刀')
  }
  run() {
    console.log('跑了一段距离')
  }
}
// Step4 手动执行装饰器函数，装饰 GuanYu 的 attack 函数
decoratorLogTime(GuanYu, 'attack')
// Step4 手动执行装饰器函数，装饰 GuanYu 的 run 函数
decoratorLogTime(GuanYu, 'run')


const guanYu = new GuanYu()
guanYu.attack()
// 挥了一次大刀
// 耗时: 0ms
guanYu.run()
```

```
// 跑了一段距离
// 耗时: 0ms
```

以上就是装饰器的具体实现方法，其核心思路是：

1. **Step1 备份原来类构造器 (Class.prototype) 的属性描述符 (Descriptor)**

利用 Object.getOwnPropertyDescriptor 获取

2. **Step2 编写装饰器函数业务逻辑代码**

利用执行原函数前后钩子，添加耗时统计逻辑

3. **Step3 用装饰器函数覆盖原来属性描述符的 value**

利用 Object.defineProperty 代理

4. **Step4 手动执行装饰器函数，装饰 Class(类) 指定属性**

从而实现在不修改原代码的前提下，执行额外逻辑代码

5. @Decorator 装饰器语法糖

但上一步 4.2.2 手写的装饰器函数存在两个可优化的点：

1. **是否可以让装饰器函数更关注业务逻辑?**

Step1, Step2 是通用逻辑的，每个装饰器都需要实现，简单来说就是可复用的。

2. **是否可以让装饰器写法更简单?**

纯函数实现的装饰器，每装饰一个属性都要手动执行装饰器函数，详见 Step4 步骤。针对上述优化点，装饰器草案中有一颗特别甜的语法糖，也就是 @Decorator ，它能够帮你省去很多繁琐的步骤来用上装饰器。

只需要在想使用的装饰器前加上 @ 符号，装饰器就会被应用到目标上。

5.1 @Decorator 语法糖的便捷性

下面我们用 @Decorator 的写法，来实现同样的功能，看看代码量可以精简多少：

```
// Step2 编写装饰器函数业务逻辑代码
function logTime(target, key, descriptor) {
  const oldMethed = descriptor.value
  const logTime = function (...arg) {
    let start = +new Date()
    try {
      return oldMethed.apply(this, arg) // 调用之前的函数
    } finally {
      let end = +new Date()
      console.log(`耗时: ${end - start}ms`)
```

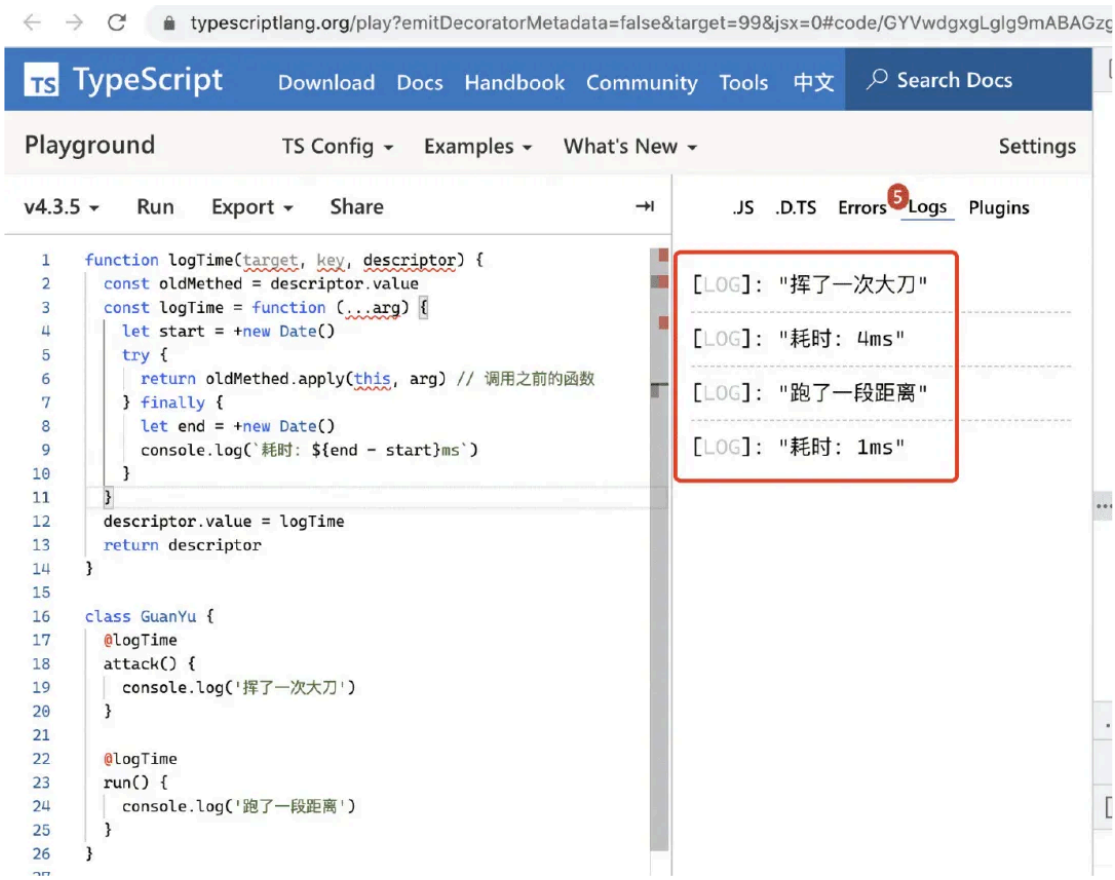
```
    }
  }
  descriptor.value = logTime
  return descriptor
}

class GuanYu {
  // Step4 利用 @ 语法糖装饰指定属性
  @logTime
  attack() {
    console.log('挥了一次大刀')
  }

  // Step4 利用 @ 语法糖装饰指定属性
  @logTime
  run() {
    console.log('跑了一段距离')
  }
}

const guanYu = new GuanYu()
guanYu.attack()
// [LOG]: 挥了一次大刀
// [LOG]: 耗时: 3ms
guanYu.run()
// [LOG]: 跑了一段距离
// [LOG]: 耗时: 3ms
```

为了让更直观了解上述代码是否可以编译后正常执行，

我们可以从 **TypeScript Playground** 直接看到编译后的代码以及运行结果，

> 注意！为了方便理解，记得关闭配置 emitDecoratorMetadata 禁止输出元数据，元数据是另一个比较复杂的知识点，我们本篇文章先跳过关闭后编译的代码会更简单

我们打开**上面代码的在线 Playground 链接**，点击 Run 运行按钮，即可看到其正常运行和输出结果：



对比纯手写的装饰器，用 @Decorator 语法糖可以省去 2 个重复的步骤：

- **Step1 备份原来类构造器 (Class.prototype) 的属性描述符 (Descriptor)**

```
const oldDescriptor = Object.getOwnPropertyDescriptor(targetPrototype, key)
```

- **Step3 用装饰器函数覆盖原来属性描述符的 value**

```
Object.defineProperty(targetPrototype, key, {
  ...oldDescriptor,
  value: logTime
})
```

开发者仅需两步即可实现装饰器的功能，可以更专注于装饰器本身的业务逻辑：

- **Step2 编写装饰器函数业务逻辑代码**

```
function logTime(target, key, descriptor) {
  const oldMethed = descriptor.value
  const logTime = function (...arg) {
    let start = +new Date()
    try {
      return oldMethed.apply(this, arg) // 调用之前的函数
    } finally {
      let end = +new Date()
```

```
    console.log(`耗时: ${end - start}ms`)
   }
  }
  descriptor.value = logTime
  return descriptor
}
```

- **Step4 利用 @ 语法糖装饰指定属性**

```
@logTime
attack() {
  console.log('挥了一次大刀')
}
```

5.2 【重点】分析 @Decorator 语法糖编译后的代码

@Decorator 语法糖很甜，但却不能直接食用。因为装饰器目前仅仅是 **ECMAScript** 的语言提案，还处于 **stage-2** 阶段，无论是最新版的 Chrome 浏览器还是 Node.js 都不能直接运行带有 @Decorator 语法糖的代码。我们需要借助 TypeScript 或者 Babel 的能力，将源码编译后才能正常运行。而在 **TypeSciprt Playground** 上，我们可以直接看到编译后代码。

为了更清晰容易理解，我们把编译后的业务代码先注释掉，只看装饰器实现的相关代码：

```
"use strict";
// Part1 装饰器工具函数(__decorate)的定义
var __decorate = (this && this.__decorate) || function (decorators, target, key, desc) {
    var c = arguments.length, r = c < 3 ? target : desc === null ? desc = Object.getOwnProperty
    if (typeof Reflect === "object" && typeof Reflect.decorate === "function") r = Reflect.deco
    else for (var i = decorators.length - 1; i >= 0; i--) if (d = decorators[i]) r = (c < 3 ? d
    return c > 3 && r && Object.defineProperty(target, key, r), r;
};

function logTime(target, key, descriptor) {
    // ...
}

class GuanYu {
    // ...
}

// Part2 装饰器工具函数(__decorate)的执行
__decorate([logTime], GuanYu.prototype, "attack", null);
__decorate([logTime], GuanYu.prototype, "run", null);

// ...
```

上述代码核心点是两个部分，一个是定义，一个是执行。定义部分较为复杂，我们先从执行入手：Part2 装饰器工具函数(__decorate)的执行会传入以下 4 个参数：

1. **装饰器业务逻辑函数**
2. **类的构造器**
3. **类的构造器属性名**
4. **属性描述符(可以为 null)**

为了方便理解 Part1 装饰器工具函数 `__decorate` 的定义，我们需要精简 `__decorate` 的函数代码，让它变成最简单的样子，而精简代码的前提是收集条件：

- **条件 1 `(this && this.__decorate)` 可删除**

这里的 this 是指 window 对象，这一步的含义是避免重复定义 __decorate 函数，属于辅助代码，可删掉。

- **条件 2 `c < 3 === false`**

Part1 的 c = arguments.length 代表参数的个数，由 Part2 我们知道工具函数会传入 4 个参数，因此在本次案例中 c < 3 参数个数小于 3 的情况不存在，即 c < 3 === false，

- **条件 3 `c > 3 === true`**

本次案例中 c > 3 参数大于 3 的情况存在，即 c > 3 === true 。

- **条件 4 `desc === null`**

同时在 Part1 我们知道第四个参数 desc 传入的值就是 null ，即 desc === null

- **条件 5 `typeof Reflect !== "object"`**

Reflect 反射是 ES6 的语法，本文为了更容易理解，暂不引入新的 ES6 特性和语法，让环境默认为 ES5，即不存在 Reflect 对象，即 typeof Reflect !== "object"，有了上述条件后，我们可以进一步精简 `__decorate` 的方法

- 代码片段 1：

```
r = c < 3 ? target : desc === null ? desc = Object.getOwnPropertyDescriptor(target, key) : desc

// 根据 c < 3 === false , desc === null 条件
// 精简后

r = desc = Object.getOwnPropertyDescriptor(target, key)
// r 和 desc 此时代表的是属性的描述符 Descriptor
```

- 代码片段 2：

```
if (d = decorators[i]) r = (c < 3 ? d(r) : c > 3 ? d(target, key, r) : d(target, key)) || r;
```

```
// 根据 c < 3 === false , c > 3 === true 条件
// 精简后

if (d = decorators[i]) r = d(target, key, r) || r;
```

- 代码片段 3：

```
if (typeof Reflect === "object" && typeof Reflect.decorate === "function") r = Reflect.decorate

// 为了方便理解，本案例暂认为 Reflect 不存在
// 精简后

// 空
```

- 代码片段 4：

```
return c > 3 && r && Object.defineProperty(target, key, r), r;

// 根据 c > 3 === true, r 是属性描述符，必定存在
// 精简后

Object.defineProperty(target, key, r)
return r;
```

- **精简后整体代码：**

```
var __decorate = function (decorators, target, key, desc) {
    var c = arguments.length;
    // Step1 备份原来类构造器 (Class.prototype) 的属性描述符 (Descriptor)
    var r = desc = Object.getOwnPropertyDescriptor(target, key),
    var d;

    for (var i = decorators.length - 1; i >= 0; i--) {
      // d 为装饰器业务逻辑函数
      if (d = decorators[i]) {
        // 执行 d，并传入 target 类构造器，key 属性名，r 属性描述符
        r = d(target, key, r) || r;
      }
    }

    // Step3 用装饰器函数覆盖原来属性描述符
    Object.defineProperty(target, key, r)
    return r;
};
```

代码经过精简之后核心原理还是和我们 **4.2.2 手写一个装饰器函数**的原理是一样的。

1. **Step1 备份原来类构造器 (Class.prototype) 的属性描述符 (Descriptor)**

利用 Object.getOwnPropertyDescriptor 获取

2. **Step3 用装饰器函数覆盖原来属性描述符的 value **

利用 Object.defineProperty 代理

TypeScript 对装饰器编译后的代码，只不过是把装饰器可复用的逻辑抽离成一个工具函数，方便复用而已。分析到这里，是不是对 @Decorator 装饰器最根本的实现有了更深入的了解？从上面的例子，我们也进一步验证了：

1. Decorator Pattern 装饰器模式的设计理念：**在不修改原有代码情况下，对功能进行扩展**
2. Decorator 装饰器的具体实现，本质是函数，参数有 target, key, descriptor
3. @Decoretor 是装饰器的一种语法糖，只是一种便捷写法，编译后本质还是一个函数

6. 带参数的装饰器：装饰器工厂函数

在上面的「记录函数耗时」例子中，如果我们希望在日志前面加个可变的标签，如何实现？

答案是使用带参数的装饰器

重点：**logTime 是个高阶函数，可以理解成装饰器工厂函数，其接收参数执行后，返回一个装饰器函数**

```
function logTime(tag) { // 这是一个装饰器工厂函数


  return function(target, key, descriptor) {  // 这是装饰器
    const oldMethed = descriptor.value
    const logTime = function (...arg) {
      let start = +new Date()
      try {
        return oldMethed.apply(this, arg)
      } finally {
        let end = +new Date()
        console.log(`【${tag}】耗时: ${end - start}ms`)
      }
    }
    descriptor.value = logTime
    return descriptor
  }
}

class GuanYu {
  @logTime('攻击')
  attack() {
    console.log('挥了一次大刀')
  },
```

```
  @logTime('奔跑')
  run() {
    console.log('跑了一段距离')
  }
}

// ...
```

编译后：

```
// ...

__decorate([logTime('攻击')], GuanYu.prototype, "attack", null);
__decorate([logTime('奔跑')], GuanYu.prototype, "run", null);

// ...
```

看了编译后的代码，我们就很容易知道带参数装饰器的具体实现原理，无非是直接先执行装饰器工厂函数，此时传入对应参数，然后返回一个新的装饰器业务逻辑的函数。

7. 五种装饰器：类、属性、方法、参数、访问器

我们上面学了那么多装饰器的内容，其实只学了一种装饰器：方法装饰器，而装饰器一共有 5 种类型可被我们使用：

1. 类装饰器
2. 属性装饰器
3. 方法装饰器
4. 访问器装饰器
5. 参数装饰器

先来个全家福，然后我们逐一攻破

```
// 类装饰器
@classDecorator
class GuanYu {

  // 属性装饰器
  @propertyDecorator
  name: string;

  // 方法装饰器
  @methodDecorator
  attack (
    // 参数装饰器
    @parameterDecorator
      meters: number
  ) {}
```

```
  // 访问器装饰器
  @accessorDecorator
  get horse() {}
}
```

7.1 类装饰器

类型声明：

```
// 类装饰器
function classDecorator(target: any) {
  return // ...
};
```

- @参数：只接受一个参数

target：类的构造器

- @返回：如果类装饰器返回了一个值，她将会被用来代替原有的类构造器的声明

  因此，类装饰器适合用于继承一个现有类并添加一些属性和方法。例如我们可以添加一个 addToJsonString 方法给所有的类来新增一个 toString 方法

```
function addToJSONString(target) {
  return class extends target {
    toJSONString() {
      return JSON.stringify(this);
    }
  };
}

@addToJSONString
class C {
  public foo = "foo";
  public num = 24;
}

console.log(new C().toJSONString())
// [LOG]: "{"foo":"foo","num":24}"
```

7.2 方法装饰器

已经在上面章节介绍过利用方法装饰器来实现「记录函数耗时」功能，现在我们重新复习下

类型声明：

```
// 方法装饰器
function methodDecorator(target: any, propertyKey: string, descriptor: PropertyDescriptor) {
  return // ...
};
```

- @参数：
  1. target：对于静态成员来说是类的构造器，对于实例成员来说是类的原型链
  2. propertyKey：属性的名称
  3. descriptor：属性的**描述器**
- @返回：如果返回了值，它会被用于替代属性的描述器。

利用方法装饰器我们可以实现更多的具体场景，比如「打印 Request 的请求参数和结果」功能：

```
function loggerParamsResult(target, propertyKey, descriptor) {
  const original = descriptor.value;

  descriptor.value = async function (...args) {

    let result
    let error
    try {
      result = await original.call(this, ...args);
    } catch(e) {
      error = new Error(e)
    }
    if (error) {
      console.error('请求失败! ')
      console.error('请求参数: ', ...args)
      console.error('失败原因: ', error)
    } else {
      console.log('请求成功! ')
      console.log('请求参数', ...args)
      console.log('请求结果: ', result)
    }
    return result;
  }
}

class App {
  @loggerParamsResult
  request(data) {
    return new Promise((resolve, reject) => {
      const random = Math.random() > 0.5
      if (random) {
        resolve(random)
      } else {
```

```
        reject(random)
      }
    })
  }
}

const app = new App();
app.request({ url: 'https://www.tencent.com'});

// [LOG]: "请求成功！"
// [LOG]: "请求参数",  {
//   "url": "https://www.tencent.com"
// }
// [LOG]: "请求结果: ",  true


// [ERR]: "请求失败！"
// [ERR]: "请求参数: ",  {
//   "url": "https://www.tencent.com"
// }
// [ERR]: "失败原因: ",  false
```

总结：

无论是「记录函数耗时」还是「打印 Request 的请求参数和结果」，本质都是在实现 **Before / After 钩子**，因此我们只需要记住**方法装饰器**可以实现与 **Before / After 钩子** 相关的场景功能。

课后题：

除了上述两个例子，大家还能想到方法装饰器有什么好的应用场景吗？

7.3 属性装饰器

类型声明：

```
// 属性装饰器
function propertyDecorator(target: any, propertyKey: string) {

}
```

- @参数: 只接受两个参数，少了 descriptor **描述器**
  1. target：对于静态成员来说是类的构造器，对于实例成员来说是类的原型链
  2. propertyKey：属性的名称
- @返回: 返回的结果将被忽略

利用属性装饰器，我们可以实现一个非常简单的属性监听功能，当属性改变时触发指定函

数：

```
function observable(fnName) {  // 装饰器工厂函数
  return function (target: any, key: string): any {  // 装饰器
    let prev = target[key];
    Object.defineProperty(target, key, {
      set(next) {
        target[fnName](prev, next);
        prev = next;
      }
    })
  }
}

class Store {
  @observable('onCountChange')
  count = -1;

  onCountChange(prev, next) {
    console.log('>>> count has changed!')
    console.log('>>> prev: ', prev)
    console.log('>>> next: ', next)
  }
}

const store = new Store();
store.count = 10

// [LOG]: ">>> count has changed!"
// [LOG]: ">>> prev: ",  undefined
// [LOG]: ">>> next: ",  -1
// [LOG]: ">>> count has changed!"
// [LOG]: ">>> prev: ",  -1
// [LOG]: ">>> next: ",  10
```

7.4 访问器装饰器

访问器装饰器总体上讲和方法装饰器很接近，唯一的区别在于第三个参数 descriptor **描述器**中有的 key 不同：

方法装饰器的描述器的 key 为：

- value
- writable
- enumerable
- configurable

访问器装饰器的描述器的 key 为：

- get

- set
- enumerable
- configurable

类型声明：

```
// 访问器装饰器
function methodDecorator(target: any, propertyKey: string, descriptor: PropertyDescriptor) {
  return // ...
};
```

例如，我们可以将某个属性在赋值的时候做一层代理，额外相加一个值：

```
function addExtraNumber(num) {  // 装饰器工厂函数
  return function (target, propertyKey, descriptor) { // 装饰器
    const original = descriptor.set;

    descriptor.set = function (value) {
      const newObj = {}
      Object.keys(value).forEach(key => {
        newObj[key] = value[key] + num
      })
      return original.call(this, newObj)
    }
  }
}

class C {
  private _point = { x: 0, y: 0 }

  @addExtraNumber(2)
  set point(value: { x: number, y: number }) {
    this._point = value;
  }

  get point() {
    return this._point;
  }
}

const c = new C();
c.point = { x: 1, y: 1 };

console.log(c.point)

// [LOG]: {
//   "x": 3,
//   "y": 3
// }
```

7.5 参数装饰器

类型声明：

```
// 参数装饰器
function parameterDecorator(target: any, methedKey: string, parameterIndex: number) {

}
```

- @参数：接收三个参数
  1. target：对于静态成员来说是类的构造器，对于实例成员来说是类的原型链
  2. methedKey：方法的名称，**注意！是方法的名称，而不是参数的名称**
  3. parameterIndex：参数在方法中所处的位置的下标
- @返回：返回的值将会被忽略

单独的参数装饰器能做的事情很有限，它一般都被用于记录可被其它装饰器使用的信息。

```
function Log(target, methedKey, parameterIndex) {
  console.log(`方法名称 ${methedKey}`);
  console.log(`参数顺序 ${parameterIndex}`);
}

class GuanYu {
  attack(@Log person, @Log dog) {
    console.log(`向 ${person} 挥了一次大刀`)
  }
}

// [LOG]: "方法名称 attack"
// [LOG]: "参数顺序 0"
```

7.6 装饰器参数总结

装饰目标	参数列表	描述符属性
类装饰器	(target)	x
方法装饰器	(target, propertyKey, descriptor)	configurable: true enumerable: true writable: true initializer: [Function]
属性装饰器	(target, propertyKey)	x
访问器装饰器	(target, propertyKey, descriptor)	configurable: true enumerable: true get: [Function] set: [Function]
参数装饰器	(target, methedKey, parameterIndex)	x

8. 装饰器顺序

8.1 同种装饰器组合顺序：洋葱模型

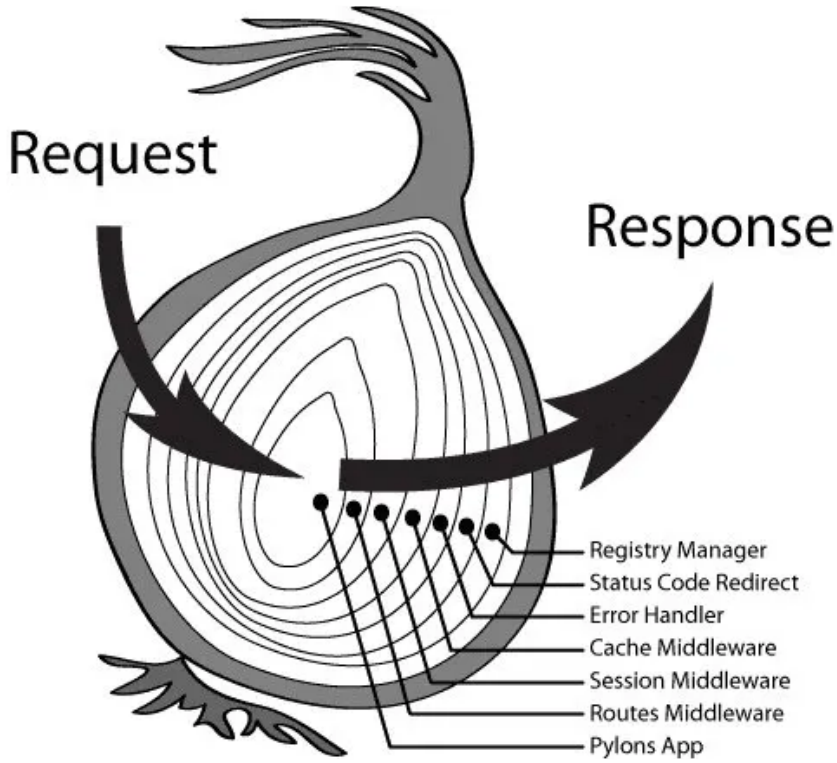
如果同一个方法有多个装饰器，其执行顺序是怎样的?

答案:

以方法装饰器为例，同种装饰器组合后，其顺序会像剥洋葱一样，

先从外到内进入，然后由内向外执行。和 Koa 的中间件顺序类似。

```
function dec(id){
  console.log('装饰器初始化', id);
  return function (target, property, descriptor) {
    console.log('装饰器执行', id);
  }
}

class Example {
  @dec(1)
  @dec(2)
  method(){}
}

// 装饰器初始化 1
// 装饰器初始化 2
// 装饰器执行 2
// 装饰器执行 1
```



I2b3cL.png

其原理，看编译后的代码就非常清楚：

重点：

1. **dec(1), dec(2) 初始化时就执行**
2. **for (var i = decorators.length - 1; i >= 0; i--) 是从右向左，倒叙执行**

```
// 由于本段代码不存在 c < 3 (参数少于3个) 的情况，为了方便理解已精简了部分不可能执行的代码
var __decorate = function (decorators, target, key, desc) {
    var c = arguments.length,
        r = desc = Object.getOwnPropertyDescriptor(target, key),
        d;
    for (var i = decorators.length - 1; i >= 0; i--)
      if (d = decorators[i]) r = d(target, key, r) || r;
    Object.defineProperty(target, key, r)
    return r;
};

function dec(id) {
    console.log('装饰器初始化', id);
    return function (target, property, descriptor) {
        console.log('装饰器执行', id);
    };
}
class Example {
    method() { }
}
__decorate([
    dec(1),
    dec(2)
], Example.prototype, "method", null);

// 装饰器初始化 1
// 装饰器初始化 2
// 装饰器执行 2
// 装饰器执行 1
```

8.2 不同类型装饰器顺序：有规则有规律

1. 实例成员：(参数 > 方法) / 访问器 / 属性 装饰器 (按顺序)
2. 静态成员：(参数 > 方法) / 访问器 / 属性 装饰器 (按顺序)
3. 构造器：参数装饰器
4. 类装饰器

多种装饰器优先级为：

实例成员最高，内部成员里面的装饰器则按定义顺序执行，

依次排下来，类装饰器最低

```
function f(key: string): any {
  // console.log("初始化: ", key);
  return function () {
    console.log("执行: ", key);
  };
}

@f("8. 类")
class C {
  @f("4. 静态属性")
  static prop?: number;

  @f("5. 静态方法")
  static method(@f("6. 静态方法参数") foo) {}

  constructor(@f("7. 构造器参数") foo) {}

  @f("2. 实例方法")
  method(@f("1. 实例方法参数") foo) {}

  @f("3. 实例属性")
  prop?: number;
}

// "执行: ",  "1. 实例方法参数"
// "执行: ",  "2. 实例方法"
// "执行: ",  "3. 实例属性"
// "执行: ",  "4. 静态属性"
// "执行: ",  "6. 静态方法参数"
// "执行: ",  "5. 静态方法"
// "执行: ",  "7. 构造器参数"
// "执行: ",  "8. 类"
```

9. 装饰器总结

9.1 应用场景

装饰器很像是组合一系列函数，类似于高阶函数和类。

合理利用装饰器对一些非内部逻辑相关的代码进行封装提炼，

能够帮助我们快速完成重复性的工作，节省时间，极大提高开发效率。

1. 类装饰器

可添加额外的方法和属性，比如：扩展 toJSONString 方法

2. 方法装饰器

可实现 Before / After 钩子功能，比如：记录函数耗时，打印 request 参数结果，节流防抖

3. 属性装饰器

可监听属性改变触发其他事件，比如：实现 count 监听器

4. 访问器装饰器

5. 参数装饰器

当然，还有更多可以使用装饰器的场景等着我们去发现

- 运行时类型检查
- 依赖注入

9.2 优点

- **在不修改原有代码情况下，对功能进行扩展。**也就是对扩展开放，对修改关闭。
- **优雅地把“辅助性功能逻辑”从“业务逻辑”中分离，解耦出来。**（AOP 面向切面编程的设计理念）
- 装饰类和被装饰类可以独立发展，不会相互耦合
- 装饰模式是 Class 继承的一个替代模式，可以理解成组合

9.3 缺点

但是糖再好吃，也不要吃太多，容易坏牙齿的，滥用过多装饰器会导致很多问题：

- 理解成本：过多带业务功能的装饰器会使代码本身逻辑变得扑朔迷离
- 调试成本：装饰器层次增多，会增加调试成本，很难追溯到一个 Bug 是在哪一层包装导致的

9.4 注意事项

1. **装饰器的功能逻辑代码一定是辅助性的**

比如日志记录，性能统计等，这样才符合 AOP 面向切面编程的思想，如果把过多的业务逻辑写在了装饰器上，效果会适得其反。

2. **装饰器语法尚未定案以及未被纳入 ES 标准，标准化的过程还需要很长时间**

由于装饰器语法未来制定的标准可能与当前的装饰器实现方案有所不同，Mobx 6 出于兼容性的考虑，放弃了装饰器的用法，并建议使用 **makeObservable / makeAutoObservable** 代替。

详情请查看：https://zh.mobx.js.org/enabling-decorators.html

装饰器提案目前进度：https://github.com/tc39/proposal-decorators