前置知识：ES7 中的装饰器

小册在知识储备上不要求所有同学掌握 ES6+ 语法，所以先带大家一起过一遍装饰器的基本操作~

在 ES7 中，我们可以像写 python 一样通过一个@语法糖轻松地给一个类装上装饰器：

```
// 装饰器函数，它的第一个参数是目标类
function classDecorator(target) {
    target.hasDecorator = true
    return target
}

// 将装饰器“安装”到Button类上
@classDecorator
class Button {
    // Button类的相关逻辑
}

// 验证装饰器是否生效
console.log('Button 是否被装饰了：', Button.hasDecorator)
```

也可以用同样的语法糖去装饰类里面的方法：

```
// 具体的参数意义，在下个小节，这里大家先感知一下操作
function funcDecorator(target, name, descriptor) {
    let originalMethod = descriptor.value
    descriptor.value = function() {
    console.log('我是Func的装饰器逻辑')
    return originalMethod.apply(this, arguments)
  }
  return descriptor
}

class Button {
    @funcDecorator
    onClick() {
        console.log('我是Func的原有逻辑')
    }
}

// 验证装饰器是否生效
const button = new Button()
button.onClick()
```

注：以上代码直接放进浏览器/Node 中运行会报错，因为浏览器和 Node 目前都不支持装饰器语法，需要大家安装 Babel 进行转码：

> 安装 Babel 及装饰器相关的 Babel 插件

```
npm install babel-preset-env babel-plugin-transform-decorators-legacy --save-dev
```

注：在没有任何配置选项的情况下，babel-preset-env 与 babel-preset-latest（或者 babel-preset-es2015，babel-preset-es2016 和 babel-preset-es2017 一起）的行为完全相同。

> 编写配置文件.babelrc：

```
{
  "presets": ["env"],
  "plugins": ["transform-decorators-legacy"]
}
```

最后别忘了下载全局的 Babel 命令行工具用于转码：

```
npm install babel-cli -g
```

执行完这波操作，我们首先是对目标文件进行转码，比如说你的目标文件叫做 `test.js`，想要把它转码后的结果输出到 `babel_test.js`，就可以这么写:

```
babel test.js --out-file babel_test.js
```

> 运行babel_test.js

```
babel_test.js
```

就可以看到你的装饰器是否生效啦~

OK，知道了装饰器长啥样，我们一起看看装饰器的实现细节：

装饰器语法糖背后的故事

所谓语法糖，往往意味着“美好的表象”。正如 class 语法糖背后是大家早已十分熟悉的 ES5 构造函数一样，装饰器语法糖背后也是我们的老朋友，不信我们一起来看看 `@decorator` 都帮我们做了些什么：

Part1：函数传参&调用

上一节我们使用 ES6 实现装饰器模式时曾经将按钮实例传给了 Decorator，以便于后续 Decorator 可以对它进行逻辑的拓展。这也正是装饰器的最最基本操作——定义装饰器函数，将被装饰者“交给”装饰器。这也正是装饰器语法糖首先帮我们做掉的工作 —— 函数传参&调用。

类装饰器的参数

当我们给一个类添加装饰器时：

```
function classDecorator(target) {
    target.hasDecorator = true
    return target
}

// 将装饰器“安装”到Button类上
@classDecorator
class Button {
    // Button类的相关逻辑
}
```

此处的 target 就是被装饰的类本身。

方法装饰器的参数

而当我们给一个方法添加装饰器时：

```
function funcDecorator(target, name, descriptor) {
    let originalMethod = descriptor.value
    descriptor.value = function() {
    console.log('我是Func的装饰器逻辑')
    return originalMethod.apply(this, arguments)
  }
  return descriptor
}

class Button {
    @funcDecorator
    onClick() {
        console.log('我是Func的原有逻辑')
    }
}
```

此处的 target 变成了 `Button.prototype` ，即类的原型对象。这是因为 onClick 方法总是要依附其实例存在的，修饰 onClik 其实是修饰它的实例。但我们的装饰器函数执行的时候，Button 实例还**并不存在**。为了确保实例生成后可以顺利调用被装饰好的方法，装饰器只能去修饰 Button 类的原型对象。

装饰器函数调用的时机

装饰器函数执行的时候，Button 实例还并不存在。这是因为实例是在我们的代码**运行时**动态生成的，而装饰器函数则是在**编译阶段**就执行了。所以说装饰器函数真正能触及到的，就只有类这个层面上的对象。

Part2：将“属性描述对象”交到你手里

在编写类装饰器时，我们一般获取一个target参数就足够了。但在编写方法装饰器时，我们往往需要至少三个参数：

```
function funcDecorator(target, name, descriptor) {
    let originalMethod = descriptor.value
    descriptor.value = function() {
    console.log('我是Func的装饰器逻辑')
    return originalMethod.apply(this, arguments)
  }
  return descriptor
}
```

第一个参数的意义，前文已经解释过。第二个参 数name，是我们修饰的目标属性属性名，也没啥好讲的。关键就在这个 descriptor 身上，它也是我们使用频率最高的一个参数，它的真面目就是“属性描述对象”（attributes object）。这个名字大家可能不熟悉，但 `Object.defineProperty` 方法我想大家多少都用过，它的调用方式是这样的：

```
Object.defineProperty(obj, prop, descriptor)
```

此处的descriptor和装饰器函数里的 descriptor 是一个东西，它是 JavaScript 提供的一个内部数据结构、一个对象，专门用来描述对象的属性。它由各种各样的属性描述符组成，这些描述符又分为数据描述符和存取描述符：

- 数据描述符：包括 value（存放属性值，默认为默认为 undefined）、writable（表示属性值是否可改变，默认为true）、enumerable（表示属性是否可枚举，默认为 true）、configurable（属性是否可配置，默认为true）。

- 存取描述符：包括 `get` 方法（访问属性时调用的方法，默认为 undefined），`set`（设置属性时调用的方法，默认为 undefined ）

很明显，拿到了 descriptor，就相当于拿到了目标方法的控制权。通过修改 descriptor，我们就可以对目标方法为所欲为的逻辑进行拓展了~

在上文的示例中，我们通过 descriptor 获取到了原函数的函数体（originalMethod），把原函数推迟到了新逻辑（console）的后面去执行。这种做法和我们上一节在ES5中实现装饰器模式时做的事情一狗一样，所以说装饰器就是这么回事儿，换汤不换药~

生产实践

装饰器在前端世界的应用十分广泛，即便是在 ES7 未诞生的那些个蛮荒年代，也没能阻挡我们用装饰器开挂的热情。要说优秀的生产实践，可以说是两天两夜也说不完。但有一些实践，我相信大家可能都用过，或者说至少见过、听说过，只是当时并不清楚这个是装饰器模式。此处为了强化大家脑袋里已有的经验与设计模式知识之间的关联，更为了趁热打铁、将装饰器模式常见的用法给大家加固一下，我们一起来看几个不错的生产实践案例：

React中的装饰器：HOC

> 高阶组件就是一个函数，且该函数接受一个组件作为参数，并返回一个新的组件。

HOC (Higher Order Component) 即高阶组件。它是装饰器模式在 React 中的实践，同时也是 React 应用中非常重要的一部分。通过编写高阶组件，我们可以充分复用现有逻辑，提高编码效率和代码的健壮性。

我们现在编写一个高阶组件，它的作用是把传入的组件**丢进一个有红色边框的容器里**（拓展其样式）。

```
import React, { Component } from 'react'

const BorderHoc = WrappedComponent => class extends Component {
  render() {
    return <div style={{ border: 'solid 1px red' }}>
      <WrappedComponent />
    </div>
  }
}
export default borderHoc
```

用它来装饰目标组件

```
import React, { Component } from 'react'
import BorderHoc from './BorderHoc'

// 用BorderHoc装饰目标组件
@BorderHoc
class TargetComponent extends React.Component {
  render() {
    // 目标组件具体的业务逻辑
  }
}

// export出去的其实是一个被包裹后的组件
export default TargetComponent
```

可以看出，高阶组件从实现层面来看其实就是上文我们提到的类装饰器。在高阶组件的辅助下，我们不必因为一个小小的拓展而大费周折地编写新组件或者把一个新逻辑重写 N 多次，只需要轻轻 @ 一下装饰器即可。

使用装饰器改写 Redux connect

Redux 是热门的状态管理工具。在 React 中，当我们想要引入 Redux 时，通常需要调用 connect 方法来把状态和组件绑在一起：

```
import React, { Component } from 'react'
import { connect } from 'react-redux'
import { bindActionCreators } from 'redux'
import action from './action.js'

class App extends Component {
  render() {
    // App的业务逻辑
  }
}

function mapStateToProps(state) {
  // 假设App的状态对应状态树上的app节点
  return state.app
}

function mapDispatchToProps(dispatch) {
  // 这段看不懂也没关系，下面会有解释。重点理解connect的调用即可
  return bindActionCreators(action, dispatch)
}

// 把App组件与Redux绑在一起
export default connect(mapStateToProps, mapDispatchToProps)(App)
```

这里给没用过 redux 的同学解释一下 connect 的两个入参： `mapStateToProps` 是一个函数，它可以建立组件和状态之间的映射关系； `mapDispatchToProps` 也是一个函数，它用于建立组件和 `store.dispatch` 的关系，使组件具备通过 dispatch 来派发状态的能力。

总而言之，我们调用 connect 可以返回一个**具有装饰作用的函数**，这个函数可以接收一 个React 组件作为参数，使这个目标组件和 Redux 结合、具备 Redux 提供的数据和能力。既然有装饰作用，既然是**能力的拓展**，那么就一定能用装饰器来改写： 把 connect 抽出来：

```
import { connect } from 'react-redux'
import { bindActionCreators } from 'redux'
import action from './action.js'

function mapStateToProps(state) {
  return state.app
}

function mapDispatchToProps(dispatch) {
  return bindActionCreators(action, dispatch)
}

// 将connect调用后的结果作为一个装饰器导出
export default connect(mapStateToProps, mapDispatchToProps)
```

在组件文件里引入connect：

```
import React, { Component } from 'react'
import connect from './connect.js'

@connect
export default class App extends Component {
  render() {
    // App的业务逻辑
  }
}
```

这样一来，我们的代码结构是不是清晰了很多？可维护性、可读性都上升了一个level，令人赏心悦目~

> Tips： 回忆一下上面一个小节的讲解，对号入座看一看，connect装饰器从实现和调用方式上来看，是不是同时也是一个高阶组件呢？

优质的源码阅读材料——core-decorators

前面都在教大家怎么写装饰器模式，这里来聊聊怎么**用好**装饰器模式。

装饰器模式的优势在于其极强的灵活性和可复用性——它本质上是一个函数，而且往往不依赖于任何逻辑而存在。这一点提醒了我们，当我们需要用到某个反复出现的拓展逻辑时，比起自己闷头搞，不如去看一看团队（社区）里有没有现成的实现，如果有，那么贯彻“拿来主义”，直接@就可以了。所以说装饰器模式是个好同志，它可以帮我们省掉大量复制粘贴的时间。

这里就要给大家推荐一个非常赞的装饰模式库 —— core-decorators 。core-decorators 帮我们实现好了一些使用频率较高的装饰器，比如 `@readonly` (使目标属性只读)、 `@deprecate` (在控制台输出警告，提示用户某个指定的方法已被废除)等等等等。这里强烈建议大家把 core-decorators 作为自己的源码阅读材料，你能收获的或许比你想象中更多~

（阅读过程中有任何想法或疑问，或者单纯希望和笔者交个朋友啥的，欢迎大家添加我的微信xyalinode与我交流哈~）