

## 前端常用设计模式(1)--装饰器(decorator)原 荐

kidzhao 2018年05月31日 20:44 浏览(465) 已收藏(27) 评论(8) 分享

关于作者



kidzhao(赵少龙) PCG新闻技术平台部平台前端组员工

作者文章

- 【数途】渐进式数据可视化平台
- 【LEAH全栈】三小时开发一个权限系统
- 从零开发一个node命令行工具
- RxJS不完全指北(入门篇)
- leah-react-app开发记录(二)

导语 从《塞尔传说-荒野之息》谈起

### 一.引子-先来安利一款好游戏

《塞尔达传说-荒野之息》，这款于2017年3月3日由任天堂(“民间高手”)发售在自家主机平台WIIU和SWITCH上的单机RPG游戏，可谓是跨时代的“神作”了。第一次制作“开放类”游戏的任天堂就教科书般的定义了这类游戏应该如何制作。



而这个游戏真正吸引我的地方是他的**细节**，举个栗子，《荒野之息》中的世界有天气和温度两个概念，会下雨打雷，有严寒酷暑，但是这些天气不想大多数游戏一样，只是简单的背景，而是实实在在会影响主角林克(Link)每一个操作。比如，下雨天去爬山会打滑；打雷天如果身上有金属装备会被雷劈(木制装备则没事!)；严寒中会慢慢流失体力(穿上一件保暖衣就解决了)；酷暑中使用爆炸箭则会原地爆炸!等等；





就是这些细节让这个游戏世界显得无比**真实又有趣**。

## 二.问题-如何设计这样的游戏代码？

作为程序猿，玩游戏之余不禁会思考，这样的游戏代码应该如何设计编写？

比如“攀爬”这个动作，需要判断攀爬的位置，林克的装备（有些装备能让你爬的更快），当时的天气，林克的体力等等众多条件，里面肯定参杂的无数if else，更何况这只是其中一个简单的操作，拓展到全部游戏，其复杂的不可想象。

显然这样的设计是不行的。

那我们假设“攀爬”的方法只专心处理攀爬这件事（有体力就能成功，反之失败），其他判断在方法外部执行，比如判断天气，装备，位置等等，这样就符合了程序设计的单一职责和低耦合等原则，并且判断天气的方法还可以拿去别的地方复用，增强了代码的复用度和可测试度，似乎可行！

那应该如何设计这样的代码呢？这就引出了我们今天的主角-装饰器模式。

## 三.主角-装饰器模式 (decorator)

根据GoF在《设计模式：可复用面向对象软件的基础》（以下简称《设计模式》）一书中对装饰器模式定义：装饰器模式又称**包装模式**（“wrapper”），**目的是以对用户透明的方式扩展对象的功能，是继承的一种代替方案。**

一起划重点：

- 1.**对用户透明**：一般指被装饰过的对象的对外接口不变，“攀爬”被怎么装饰都还是“攀爬”。
- 2.**扩展对象的功能**：一般指修改或添加对象功能，比如林克在雪地就可以用盾牌滑雪，平地则没有这个能力。
- 3.**继承的一种代替方案**：熟悉面向对象的同学一定对继承并不陌生，这里我们重点谈谈继承本身的一些缺点：1) 继承中子类 and 超类存在强耦合性，超类的修改会影响全部子类；2) 超类对子类是“白盒复用”，子类必须了解超类的全部实现，破坏了封装性。3) 当项目庞大时，继承会使得子类爆发性增长，比如《荒野之息》中存在料理系统，任意两种食材均可以搭配出一款料理，假定有10中可以使用食材，使用继承的方式就要构建 $10 \times 10 = 100$ 个子类表示料理结果，而装饰器模式仅仅使用 $10 + 1 = 11$ 个子类就可以完成以上工作。（还包括了任意种食材的混合，事实上游戏中的确可以。）

最后，总结一下装饰器模式的特点：不改变对象自身的基础上，在程序运行时给对象添加某种功能，一句话：**锦上添花**。（想想《王者荣耀》中最赚钱的皮肤，怎么全是游戏，喂！）

## 四.场景-面向切片编程 (AOP)

说到装饰器，最经典的应用场景就是**面向切片编程**（Aspect Oriented Programming，以下简称AOP），AOP适合某些具有**横向逻辑**（可切片）的应用，比如提交表单，点击提交按钮以后执行的逻辑是：上报点击 -> 校验数据 -> 提交数据 -> 上报结果。可以看到，首尾的上报日志功能和核心业务逻辑并没有直接关系，并且几乎所有表单提交都需要上报日志的功能，因此，上报日志，这个功能就可以单独**抽象**出来，最后在程序运行（或编译）时动态织入业务逻辑

中。类似的功能还有：数据校验，权限控制，异常处理，缓存管理等等。

AOP的优点是可以保持业务逻辑模块的**纯净**和**高内聚**，同时方便功能**复用**，通过装饰器就可以很方便的把功能模块装饰到主业务逻辑中去。

## 五.应用-前端开发中的应用

接下来我们一起看看具体装饰器模式是如何在前端开发中应用的。

Talk is cheap, show me the code! (屁话少说，放码过来!)

在JS中**改变一个对象**再简单不过了。

```
2 var obj = {
3   name: '林克',
4   address: "海拉尔",
5   showProps(props) {
6     console.log(` ${props}: ${this[props]} `);
7   }
8 };
9
10 obj.showProps("address"); // address: 海拉尔
11 obj.address += " 初始之地";
12 obj.showProps("address"); // address: 海拉尔 初始之地
```

得益于JS是一门基于原型的弱类型语言，给对象添加或修改功能都十分容易，因此传统的面向对象中的装饰器模式在JS中的应用并不太多（ES6正式提出class以后场景有所增加）。

我们先简单模拟一下**面向对象中的装饰器模式**。

假设我们要开发一个飞机大战的游戏，飞机可以切换装备的武器，发射不同的子弹。

```
var Plane = function() {};
Plane.prototype.fire = function() {
  console.log("发射普通子弹! ");
};
```

我们先实现一个飞机的类，并实现一个fire方法。

接着，我们实现一个发射导弹的装饰器类

```
// 导弹装饰器
var MissileDecorator = function(plane) {
  this.plane = plane;
};
MissileDecorator.prototype.fire = function() {
  // 先调用原有对象的方法
  this.plane.fire();
  console.log("发射导弹! ");
};
```

这个类接收一个飞机实例，并且重新实现了fire方法，在方法内部先调用原来实例的fire方法，接着扩展此方法，增加了发射导弹的功能。

类似的我们再实现一个发射原子弹的装饰器。

```
// 原子弹装饰器
var AtomDecorator = function(plane) {
  this.plane = plane;
};
AtomDecorator.prototype.fire = function() {
  // 先调用原有对象的方法
  this.plane.fire();
  console.log("发射原子弹! ");
};
```

最后我们看一下应该如何使用这两个装饰器。

```
var plane = new Plane();
plane = new MissileDecorator(plane);
```

```
plane = new AtomDecorator(plane);
plane.fire();
```

可以看到，经过两个装饰器装饰后的plane实例，再调用fire方法时，就可以同时发射三种子弹了。而装饰器本身并没有直接改写Plane类，只是增强了它的fire方法，对plane实例的使用者也是透明的。

接下来我们看一看如何应用装饰器在JS中实现AOP编程。

首先我们扩展一下函数的原型，让每个函数都可以被装饰。我们给函数增加一个before和after方法，这两个方法各自接收一个新的函数，并保证新函数在原函数之前（before）或之后（after）执行。

```
// 给函数加上before和after
// 接收新函数作为参数
Function.prototype.before = function (beforeFn) {
  const __self = this; // 保存原函数引用
  // 返回包含原函数和新函数的“代理”函数
  return function () {
    // 执行新函数，在新函数之前执行
    // 保证this不被劫持，参数和原函数相同
    beforeFn.apply(this, arguments);
    // 执行原函数，返回执行结果
    return __self.apply(this, arguments);
  };
};
Function.prototype.after = function (afterFn) {
  const __self = this;
  return function () {
    const ret = __self.apply(this, arguments);
    afterFn.apply(this, arguments);
    return ret;
  };
};
```

这里需要注意的是新函数和原函数具有相同this和参数。

有了两个方法，以前很多复杂的需求就变得很简单了。

### 栗子一：挂载多个onload函数

通常情况下，window.onload只能挂载一个回调函数，重复声明回调函数，后面的会把之前声明的覆盖掉，有了after以后，这个麻烦解决了。

```
// 应用1: onload
window.onload = function () {
  console.log(1);
};
// 挂载多个onload函数
window.onload = (window.onload || function () { })
  .after(() => console.log(2))
  .after(() => console.log(3))
  .after(() => console.log(4))
```

### 栗子二：日志上报

```
// 应用2: 数据上报
let clickHeadle = () => alert('点击成功!')
const sendLog = (e) => console.log('点击元素为:', e.target.innerHTML)
clickHeadle = clickHeadle.after(sendLog)
export const Buttons = () => (
  <div>
    <Button onClick={clickHeadle}>log1</Button>
    <Button onClick={clickHeadle}>log2</Button>
  </div>
)
```

### 栗子三：追加（改变）参数

比如，为了增加安全性，给所有接口都增加一个token参数，如果不实用AOP，我们只能改



ajax方法了。但是有了AOP，就可以像下面这样操作。

```
// 应用3, 改变函数参数
// 给请求加上公共参数
const ajax = (url, params) => console.log(params)
const getToken = () => 'TOKEN'
const ajaxWithToken = ajax.before((url, params) => params.token = getToken() )

ajax( 'url1', {name: 'link'} ) // {name: 'link'}
ajaxWithToken( 'url1', {name: 'link'} ) // {name: 'link', token: 'TOKEN' }
```

原理就是before函数和原函数接收相同的this和参数，并且before会在原函数之前执行。

其实AOP在前端项目中的应用场景还很多，比如校验表单参数，异常处理，数据缓存，本地持久化等，这里不在一一举例了。

有些同学对直接改写函数的原型比较抵触，这里我们也给出函数式的before实现。

```
// 不修改Function.prototype
const before = function(fn, beforeFn) {
  return function() {
    beforeFn.apply(this, arguments)
    return fn.apply(this, arguments)
  }
}

let a = before(() => console.log('n3'), () => console.log('n2'))
a = before(a, () => console.log('n1'))
a() // n1,n2,n3
```

## 六.ES7-@decorator语法

在JS未来的标准（ES7）中，装饰器也已被加入到了提案中。

前端同学都知道jQuery最大的特点就是它链式调用的API设计，其核心是每个方法都返回this，也就是jQuery对象实例，我们不妨先实现一个高阶函数，用于实现链式调用。

```
// 流式调用
function fluent(fn) {
  return function () {
    fn.apply(this, arguments)
    return this
  }
}
```

fluent函数接收一个函数fn作为参数，返回一个新的函数，在新函数内部通过apply调用fn，并最终返回上下文this。有了这个函数，我们就可以很方便的给任意对象的方法添加链式调用。

```
class Person {
  setName = fluent(function (first, last) {
    this.first = first
    this.last = last
  })

  sayName = fluent(function () {
    console.log(`My first name is ${this.first}, my last name is ${this.last}`)
  })
}

const person = new Person()
person.setName('Harry', 'Poter').sayName().setName('Tom', 'Cat').sayName()
```

接下来，我们看看如何使用ES7的@decorator语法来简化上面的代码，先来看一下结果。

```
class Person2 {
  @fluentDecorate
  setName(first, last) {
    this.first = first
    this.last = last
  }
  @fluentDecorate
  sayName() {
    console.log(`My first name is ${this.first}, my last name is ${this.last}`)
  }
}
```

```

    sayName() {
      console.log(`My first name is ${this.first}, my last name is ${this.last}`)
    }
  }
  const person2 = new Person2();
  person2.setName('Jane', 'Doe').sayName().setName('John', 'Wang').sayName()

```

熟悉Java的同学一眼就看出这不是注解写法么，没错，ES7中的@decorator正是参考了Python和Java语法设计出来的。@后面的fluentDecorate是一个装饰器函数，这个函数接收三个参数，分别是target，name和descriptor，这三个参数和Object.defineProperty方法的参数完全相同，实际上@decorator也正是这个方法语法糖而已。

值得注意的是@decorator不止可以作用在对象或类的方法上面，还可以直接作用在类（class）上，区别是装饰函数的第一个参数target不同，当作用在方法上时，target指向对象本身，而当作用在类时target指向类（class），并且name和descriptor都是undefined。

以下给出fluentDecorate函数的完整实现。

```

// 使用装饰器装饰对象方法
// target: 表示当前调用这个方法的对象，也就是上面new出来的person对象。
// name: 表示的是当前要修饰的方法名，例如这里是: 'sayName' 或者 'setName'。
// descriptor: 表示的是当前这个方法的描述符对象，
// 一个方法或者属性的描述符对象可以通过ES5的Object.defineProperty方法进行设置，
// 详见JavaScript高级程序设计的139页。
function fluentDecorate(target, name, descriptor) {
  // console.log(target, name, descriptor)
  const fn = descriptor.value;
  descriptor.value = function (...args) {
    fn.apply(target, args)
    return target
  }
}

```

通常我们可以把这个装饰函数再抽象一下，让他成为一个高阶函数，可以接收我们最开始定义的fluent函数或者其他函数（比如节流函数等），然后返回一个用这个函数装饰的新装饰函数，更具有通用型。

```

// 抽象decorate
function decorateWith(decorator) {
  return (target, name, descriptor) => {
    console.log(target, name, descriptor)
    descriptor.value = decorator.call(target, descriptor.value)
  };
}

```

```

class Person3 {
  @decorateWith(fluent)
  setName(first, last) {
    this.first = first
    this.last = last
  }
  @decorateWith(fluent)
  sayName() {
    console.log(`My first name is ${this.first}, my last name is ${this.last}`)
  }
}

const person3 = new Person3()
person3.setName('Jane', 'Doe').sayName().setName('John', 'Doe').sayName()

```

@decorator到目前为止还只是个提案，没有任何浏览器支持了这个语法，但是好在可以使用Babel以插件（transform-decorators-legacy）的形式在自己的项目中体验。

注意，@decorator只能作用于类和类的方法上，不能用于普通函数，因为函数存在变量提升，而类是不会提升的。

## 七. 组件-装饰器在React项目中的应用

最后结合目前前端最火的框架React，来看看装饰器是如何在组件上使用的。

回到最开始的假设，如何开发出《荒野之息》这样细节丰富的游戏，下面我们就使用React搭配装饰器来模拟一下游戏中的细节实现。

我们先实现一个Person组件，用来代指游戏的主角，这个组件可以接收名字，生命值，攻击类等初始化参数，并在一个卡片中展示这些参数，当生命值为0时，会提示“游戏结束”。并且在卡片中放置一个“JUMP”按钮，用点击按钮模拟主角跳跃的交互。

```
class Person extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      name: 'Link',
      hp: 1,
      atk: 1,
      def: 1
    }
    this.onJump = this.onJump.bind(this)
  }
  shouldComponentUpdate(nextProps, nextState) {
    if (nextState.hp === this.state.hp) return false
    return true
  }
  componentDidMount() {
    this.init(this.props)
  }
  componentWillReceiveProps(nextProps) {
    this.init(nextProps)
  }
  init = (props) => this.setState({ ...this.state, ...props })
  onJump() {
    message.success('jump success!')
  }
  render() {
    if (this.state.hp <= 0) setTimeout(() => alert('game over!'), 200)
    return (
      <Card>
        {Object.keys(this.state).map(key => <div key={key}>{` ${key}: ${this.state[key]} `}</div>)}
        <Button onClick={this.onJump}>JUMP</Button>
      </Card>
    );
  }
}
```

组件调用：

```
<Person name="塞尔达" hp="100" />
```

实现结果如下，是不是很抽象？哈哈！



接下来我们想要模拟游戏中的天气和温度变化，需要实现一个“自然环境”的组件Natural，这个组件自身有天气（wat）和温度（tep）两个状态（state），并且可以通过输入改变这两个状态，我们之前创建的Person组件作为后代插入这个组件中，并且接收Natural的wat和tep状态作为属性。

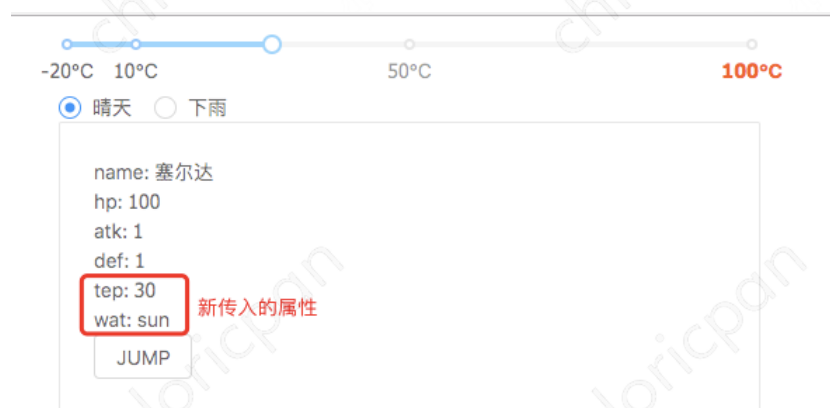
```
class Natural extends React.Component {
  state = {
    tep: 30, // 温度
    wat: 'sun' // 天气
  }
  render() {
    const { tep, wat } = this.state
    return (
      <div style={{ width: '500px', margin: '10px auto' }} >
        <Slider marks={marks} value={tep} onChange={value => this.setState({ tep: value })} />
        <RadioGroup value={wat} onChange={e => this.setState({ wat: e.target.value })} >
          <Radio value='sun'>晴天</Radio>
          <Radio value='rain'>下雨</Radio>
        </RadioGroup>
      </div>
    );
  }
}
```

```

    <Person name="塞尔达" hp="100" tep={tep} wat={wat} />
  </div>
);
}
}

```

好了，我们的实验页面就完成了，最终效果如下，上面可以通过进度条和单选按钮改变天气和温度，改变后的结果通过props传递给游戏主角。



但是现在改变温度和天气对主角并不会造成任何影响，接下来我们想在**不改变原有Person组件**的前提下，实现两个功能：**第一，当温度大于50度或者小于10度的时候，主角生命值慢慢下降；第二当天气是雨天的时候，主角每跳跃3次就失败1次。**

先来实现第一个功能，温度过高和过低时，主角生命值慢慢减少。我们的思路是实现一个装饰器，用这个装饰器在外部装饰Person组件，使得这个组件可以感知温度变化。先给出实现：

```

const decorateTep = (WapperedComponent) => {
  // console.log(WapperedComponent)
  return class extends React.Component {
    state = {
      hp: this.props.hp,
      tep: this.props.tep
    }

    componentDidMount() {
      setInterval(this.checkTep, 1000)
    }

    componentWillReceiveProps(nextProps) {
      const { tep } = nextProps
      this.setState({ tep })
    }

    checkTep = () => {
      const { hp, tep } = this.state
      if (tep > 50 || tep < 10) {
        const nhp = hp - 10
        this.setState({ hp: nhp > 0 ? nhp : 0 })
      }
    }

    render() {
      const { hp, ...ext } = this.props
      return <WapperedComponent hp={this.state.hp} {...ext} />
    }
  }
}

```

仔细观察decorateTep函数，它接收一个组件（A）作为参数，返回一个新的React组件（B），在B内部维护了一个hp和tep状态，在tep处于临界值时，改变B的hp，最后render时用B的hp代替原来的hp属性传递给A组件。

这不是就是高阶组件（HOC）么？！没错，当装饰器去装饰一个组件时，它的实现和高阶组件完全一致。通过返回一个新组件的方式去增强原有组件的能力，这也符合React提倡的组件组合的设计模式（注意不是mixin或者继承），decorateTep的使用方法很简单，一行代码搞定：



```
@decorateTep
class Person extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      name: 'Link',
      hp: 1,

```

接下来我们来实现第二个功能，下雨时跳跃会偶尔失败，这里我们换一个策略，不再装饰Person组件，而是装饰组件内部的onJump跳跃方法。代码如下：

```
const decorateWat = (target, key, descriptor) => {
  // console.log(target, key, descriptor)
  let i = 0
  var method = descriptor.value
  descriptor.value = function (...args) {
    if (this.state.wat !== 'rain') return method.apply(this, args)

    i++
    if (i === 4) {
      message.error('jump fail!')
      i = 0
    } else {
      return method.apply(this, args)
    }
  }
  return descriptor
}
```

区别之前的decorateTep，这个decorateWat装饰器的重点是**第三个参数descriptor**，之前提到，descriptor参数是被装饰方法的描述对象，它的value属性指向的就是**原方法(onJump)**，这里我们用变量method保存原方法，同时使用i记录点击次数，通过**闭包**延长这两个变量的生命周期，最后实现一个新的方法代替原方法，在新方法内部通过**apply**调用原方法并重置变量i，**注意decorateWat最后返回的是改变以后的descriptor对象。**

经过装饰器装饰过的onJump方法如下：

```
init = (props) => this.setState({ ...this.state, ...props })

@decorateWat
onJump() {
  message.success('jump success!')
}

render() {
  if (this.state.hp <= 0) setTimeout(() => alert('game over!'), 200)
  return (

```

好了，接下来就是见证奇迹的时刻！

## 八.轮子-常用装饰器库

事实上现在已经有非常多开源装饰器的库可以拿来使用，以下是质量较好的轮子，希望可以给大家提供帮助。

[core-decorators](#)

[lodash-decorators](#)

[react-decoration](#)

## 九.参考-相关资料阅读

[全部演示源代码](#)

[五分钟让你明白为什么塞尔达可以夺得年度游戏](#)

[《荒野之息》中46个精彩的小细节](#)

[日亚上一位玩家对《荒野之息》的评价](#)

[面向切片编程](#)

《JavaScript 设计模式与开发实践》曾探；人民邮电出版社

《JavaScript 高级程序设计（第三版）》Zakas；人民邮电出版社

《ES 6 标准入门（第二版）》阮一峰；电子工业出版社

最后，如有不对的地方，欢迎各位小伙伴留言拍砖，你们的支持是我继续的最大动力！

谢谢大家！

最后更新于 2019-08-26 02:06

1人已赞赏



仅供内部学习与交流，未经授权切勿外传

标签：[前端\(1\)](#) [设计模式\(1\)](#) [react\(1\)](#) [ES6\(1\)](#) [装饰器\(1\)](#)



本文专属二维码，扫一扫还能分享朋友圈  
想要微信公众号推广本文章？[点击获取链接](#)



我顶 (14)



已收藏(27)

选中文章内容可快速反馈

分享到

转载 (5) 收录 评论 (8) 反馈

大家评论



yajieliu

2018-05-31 20:51:15



👍 顶    💬 回复



erxiaowu

2018-05-31 21:13:46



👍 顶    💬 回复



jjixu

2018-06-01 13:32:28



👍 顶    💬 回复



chunpengliu

2018-06-01 14:15:50

棒

👍 顶    💬 回复




sannyli

2018-06-04 11:10:46

好厉害啊!

👍 顶    💬 回复



sannyli

2018-06-04 15:15:54


有个问题想请教一下：

装饰器使用的时候是在类 (Person) 的定义上面加上 at decorator，那如果在另外一个地方要对 Person 使用另外一个装饰器，是不是也要把 Person 类的定义重复一遍？例如这样：

```
at decorator1
class Person extends React.Component {}

at decorator2
class Person extends React.Component {}
```

👍 顶    💬 回复 (2)



kidzhao (楼主)

2018-06-04 16:37:10

hi, 感谢提问。

很遗憾，@decorator 语法只支持在类定义时候对类或类中的方法去装饰。

如果想实现题主的要求，可以使用HOC的语法，单独导出Person，在使用时去装饰，  
const Person1 = decorator1(Person); const Person2 = decorator2(Person);

事实上，HOC在这里就是装饰器的一种实现，设计模式这个东西很多时候不要关注他的实现方式（语法），而是关注他的意图和目的。

希望这个回答可以忙到你哈！

💬 回复




sannyli

2018-06-04 17:35:22

明白，谢谢~

💬 回复



😊 切换到更多功能

发表评论

