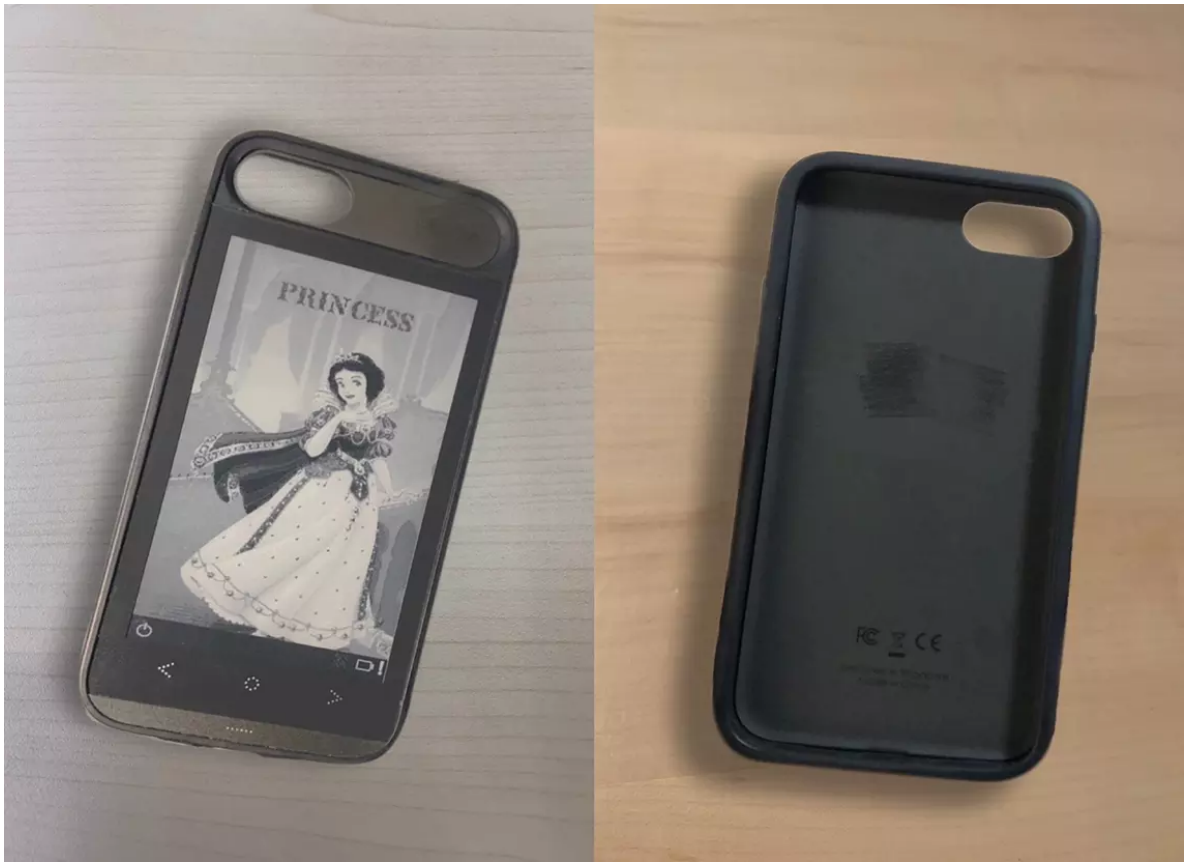
装饰器模式，又名装饰者模式。它的定义是“在不改变原对象的基础上，通过对其进行包装拓展，使原有对象可以满足用户的更复杂需求”。

当然，对于没接触过装饰器的同学来说，这段定义意义不大。我们先借助一个生活中的例子来理解装饰器：

生活中的装饰器

去年有个手机壳在同事里非常流行，我也随大流买了一个，它长这样：



这个手机壳的安装方式和普通手机壳一样，就是卡在手机背面。不同的是它卡上去后会变成一块水墨屏，这样一来我们手机就有了两个屏幕。平时办公或者玩游戏的时候，用正面的普通屏幕；阅读的时候怕伤眼睛，就可以翻过来用背面的水墨屏了。

这个水墨屏手机壳安装后，**不会对手机原有的功能产生任何影响，仅仅是使手机具备了一种新的能力**（多了块屏幕），因此它在此处就是一个标准的装饰器。

PS：手机壳是挺好用的，就是有点厚，用了一个月我就放弃了，目前正在工位一角吃灰。大家谨记理性消费、适度消费。

装饰器的应用场景

按钮是我们平时写业务时常见的页面元素。假设我们的初始需求是：每个业务中的按钮在点击后都弹出「您还未登录哦」的弹框。

那我们可以很轻易地写出这个需求的代码：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>按钮点击需求1.0</title>
</head>
<style>
    #modal {
        height: 200px;
        width: 200px;
        line-height: 200px;
        position: fixed;
        left: 50%;
        top: 50%;
        transform: translate(-50%, -50%);
        border: 1px solid black;
        text-align: center;
    }
</style>
<body>
    <button id='open'>点击打开</button>
    <button id='close'>关闭弹框</button>
</body>
<script>
    // 弹框创建逻辑，这里我们复用了单例模式面试题的例子
    const Modal = (function() {
        let modal = null
        return function() {
            if(!modal) {
                modal = document.createElement('div')
                modal.innerHTML = '您还未登录哦~'
                modal.id = 'modal'
                modal.style.display = 'none'
                document.body.appendChild(modal)
            }
            return modal
        }
    })()

    // 点击打开按钮展示模态框
    document.getElementById('open').addEventListener('click', function() {
        // 未点击则不创建modal实例，避免不必要的内存占用
        const modal = new Modal()
        modal.style.display = 'block'
    })

    // 点击关闭按钮隐藏模态框
    document.getElementById('close').addEventListener('click', function() {
        const modal = document.getElementById('modal')
        if(modal) {
            modal.style.display = 'none'
        }
    })
</script>
</html>
```

按钮发布上线后，过了几天太平日子。忽然有一天，产品经理找到你，说这个弹框提示还不够明显，我们应该在弹框被关闭后把按钮的文案改为“快去登录”，同时把按钮置灰。

听到这个消息，你立刻马不停蹄地翻出之前的代码，找到了按钮的 click 监听函数，手动往里面添加了文案修改&按钮置灰逻辑。但这还没完，因为你司的几乎每个业务里都用到了这类按钮：除了“点击打开”按钮，还有“点我开始”、“点击购买”按钮等各种五花八门的按钮，这意味着你不得不深入到每一个业务的深处去给不同的按钮添加这部分逻辑。

有的业务不在你这儿，但作为这个新功能迭代的 owner，你还需要把需求细节再通知到每一个相关同事（要么你就自己上，去改别人的代码，更恐怖），怎么想怎么麻烦。一个文案修改&按钮置灰尚且如此麻烦，更不要说我们日常开发中遇到的更复杂的需求变更了。

不仅麻烦，直接去修改已有的函数体，这种做法违背了我们的“开放封闭原则”；往一个函数体里塞这么多逻辑，违背了我们的“单一职责原则”。所以说这个事儿，越想越不能这么干。

> 我想一定会有同学质疑说为啥不把按钮抽成公共组件 Button，这样只需要在 Button 组件里修改一次逻辑就可以了。这种想法非常好。但注意，我们楼上的例子没有写组件直接写了 Button 标签是为了简化示例。事实上真要写组件的话，不同业务里必定有针对业务定制的不同 Button 组件，比如 MoreButton 、BeginButton等等，也是五花八门的，所以说我们仍会遇到同样的困境。

讲真，我想任何人去做这个需求的时候，其实都压根**不想去关心它现有的业务逻辑是啥样的**——你说这按钮的旧逻辑是我自己写的还好，理解成本不高；万一碰上是个离职同事写的，那阅读难度谁能预料呢？我不想接锅，我只是想**对它已有的功能做个拓展，只关心拓展出来的那部分新功能如何实现**，对不对？

程序员说：“我不想努力了，我想开挂”，于是便有了装饰器模式。

装饰器模式初相见

为了不被已有的业务逻辑干扰，当务之急就是将旧逻辑与新逻辑分离，**把旧逻辑抽出去**：

```
// 将展示Modal的逻辑单独封装
function openModal() {
    const modal = new Modal()
    modal.style.display = 'block'
}
```

编写新逻辑：

```
// 按钮文案修改逻辑
function changeButtonText() {
    const btn = document.getElementById('open')
    btn.innerText = '快去登录'
}

// 按钮置灰逻辑
function disableButton() {
    const btn =  document.getElementById('open')
    btn.setAttribute("disabled", true)
}

// 新版本功能逻辑整合
function changeButtonStatus() {
    changeButtonText()
    disableButton()
```

```
}
```

然后把三个操作逐个添加open按钮的监听函数里：

```
document.getElementById('open').addEventListener('click', function() {
    openModal()
    changeButtonStatus()
})
```

如此一来，我们就实现了“只添加，不修改”的装饰器模式，使用changeButtonStatus的逻辑装饰了旧的按钮点击逻辑。以上是ES5中的实现，ES6中，我们可以以一种更加面向对象化的方式去写：

```
// 定义打开按钮
class OpenButton {
    // 点击后展示弹框（旧逻辑）
    onClick() {
        const modal = new Modal()
        modal.style.display = 'block'
    }
}

// 定义按钮对应的装饰器
class Decorator {
    // 将按钮实例传入
    constructor(open_button) {
        this.open_button = open_button
    }

    onClick() {
        this.open_button.onClick()
        // “包装”了一层新逻辑
        this.changeButtonStatus()
    }

    changeButtonStatus() {
        this.changeButtonText()
        this.disableButton()
    }

    disableButton() {
        const btn =  document.getElementById('open')
        btn.setAttribute("disabled", true)
    }

    changeButtonText() {
        const btn = document.getElementById('open')
        btn.innerText = '快去登录'
    }
}

const openButton = new OpenButton()
const decorator = new Decorator(openButton)

document.getElementById('open').addEventListener('click', function() {
    // openButton.onClick()
    // 此处可以分别尝试两个实例的onClick方法，验证装饰器是否生效
    decorator.onClick()
})
```

大家这里需要特别关注一下 ES6 这个版本的实现，这里我们把按钮实例传给了 Decorator，以便于后续 Decorator 可以对它为所欲为进行逻辑的拓展。在 ES7 中，Decorator 作为一种语法被直接支持了，它的书写会变得更加简单，但背后的原理其实与此大同小异。在下一节，我们将一起去探究一下 ES7 中 Decorator 背后的故事。

值得关注的细节

结束了装饰器的感性认知之旅，下一节我们将直奔ES7装饰器原理&装饰器优秀案例教学。在此之前，我们对本节的一个小细节进行复盘：

单一职责原则

大家可能刚刚没来得及注意，按钮新逻辑中，文本修改&按钮置灰这两个变化，被我封装在了两个不同的方法里，并以组合的形式出现在了最终的目标方法 `changeButtonStatus` 里。这样做的目的是为了强化大家脑中的“单一职责”意识。将不同的职责分离，可以做到每个职责都能被灵活地复用；同时，不同职责之间无法相互干扰，不会出现因为修改了 A 逻辑而影响了 B 逻辑的狗血剧情。

但是，**设计原则并非是板上钉钉的教条。**在此处，我们的代码总共只有两行、且比较简单，逻辑分离的诉求并不特别强，分开最好，不分影响也不大（此处我们选择了拆散两段逻辑，更多地是为了强化大家的意识）。在日常开发中，当遇到两段各司其职的代码逻辑时，我们首先要有“尝试拆分”的敏感，其次要有“该不该拆”的判断——当逻辑粒度过小时，盲目拆分会导致你的项目里存在过多的零碎的小方法，这反而不会使我们的代码变得更好。

OK，真正的战斗才刚刚开始，大家梳理一下思路，一起进入下一节的学习吧~

（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~）