本节我们选取业务开发中最常见的四种代理类型：事件代理、虚拟代理、缓存代理和保护代理来进行讲解。

在实际开发中，代理模式和我们下节要讲的“大 Boss ”观察者模式一样，可以玩出花来。但设计模式这玩意儿就是这样，变体再多、玩得再花，它的核心操作都是死的，套路也是死的——正是这种极强的规律性带来了极高的性价比。相信学完这节后，大家对这点会有更深的感触。

事件代理

事件代理，可能是代理模式最常见的一种应用方式，也是一道实打实的高频面试题。它的场景是一个父元素下有多个子元素，像这样：

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>事件代理</title>
</head>
<body>
  <div id="father">
    <a href="#">链接1号</a>
    <a href="#">链接2号</a>
    <a href="#">链接3号</a>
    <a href="#">链接4号</a>
    <a href="#">链接5号</a>
    <a href="#">链接6号</a>
  </div>
</body>
</html>
```

我们现在的需求是，希望鼠标点击每个 a 标签，都可以弹出“我是xxx”这样的提示。比如点击第一个 a 标签，弹出“我是链接1号”这样的提示。这意味着我们至少要安装 6 个监听函数给 6 个不同的的元素(一般我们会用循环，代码如下所示），如果我们的 a 标签进一步增多，那么性能的开销会更大。

```
// 假如不用代理模式，我们将循环安装监听函数
const aNodes = document.getElementById('father').getElementsByTagName('a')

const aLength = aNodes.length

for(let i=0;i<aLength;i++) {
    aNodes[i].addEventListener('click', function(e) {
        e.preventDefault()
        alert(`我是${aNodes[i].innerText}`)
    })
}
```

考虑到事件本身具有“冒泡”的特性，当我们点击 a 元素时，点击事件会“冒泡”到父元素 div 上，从而被监听到。如此一来，点击事件的监听函数只需要在 div 元素上被绑定一次即可，而不需要在子元素上被绑定 N 次——这种做法就是事件代理，它可以很大程度上提高我们代码的性能。

事件代理的实现

用代理模式实现多个子元素的事件监听，代码会简单很多：

```
// 获取父元素
const father = document.getElementById('father')

// 给父元素安装一次监听函数
father.addEventListener('click', function(e) {
    // 识别是否是目标子元素
    if(e.target.tagName === 'A') {
        // 以下是监听函数的函数体
        e.preventDefault()
        alert(`我是${e.target.innerText}`)
    }
} )
```

在这种做法下，我们的点击操作并不会直接触及目标子元素，而是由父元素对事件进行处理和分发、间接地将其作用于子元素，因此这种操作从模式上划分属于代理模式。

虚拟代理

在《**性能小册的Lazy-Load小节**》，我们介绍了懒加载这种技术，此处强烈建议大家，尤其是近期有校招或跳槽需求的同学，转过头去复习一下这个小节，说不定下一次的面试题里就有原题，这点在该小节的评论区已经有同学佐证了。

我们此处简单地给大家描述一下懒加载是个什么东西：它是针对图片加载时机的优化：在一些图片量比较大的网站，比如电商网站首页，或者团购网站、小游戏首页等。如果我们尝试在用户打开页面的时候，就把所有的图片资源加载完毕，那么很可能会造成白屏、卡顿等现象。

此时我们会采取“先占位、后加载”的方式来展示图片 —— 在元素露出之前，我们给它一个 div 作占位，当它滚动到可视区域内时，再即时地去加载真实的图片资源，这样做既减轻了性能压力、又保住了用户体验。

除了图片懒加载，还有一种操作叫**图片预加载**。预加载主要是为了避免网络不好、或者图片太大时，页面长时间给用户留白的尴尬。常见的操作是先让这个 img 标签展示一个占位图，然后创建一个 Image 实例，让这个 Image 实例的 src 指向真实的目标图片地址、观察该 Image 实例的加载情况 —— 当其对应的真实图片加载完毕后，即已经有了该图片的缓存内容，再将 DOM 上的 img 元素的 src 指向真实的目标图片地址。此时我们直接去取了目标图片的缓存，所以展示速度会非常快，从占位图到目标图片的时间差会非常小、小到用户注意不到，这样体验就会非常好了。

上面的思路，我们可以不假思索地实现如下

```
class PreLoadImage {
    // 占位图的url地址
    static LOADING_URL = 'xxxxxx'

    constructor(imgNode) {
        // 获取该实例对应的DOM节点
        this.imgNode = imgNode
    }

    // 该方法用于设置真实的图片地址
    setSrc(targetUrl) {
        // img节点初始化时展示的是一个占位图
        this.imgNode.src = PreLoadImage.LOADING_URL
        // 创建一个帮我们加载图片的Image实例
        const image = new Image()
```

```
        // 监听目标图片加载的情况，完成时再将DOM上的img节点的src属性设置为目标图片的url
        image.onload = () => {
            this.imgNode.src = targetUrl
        }
        // 设置src属性，Image实例开始加载图片
        image.src = srcUrl
    }
}
```

这个 `PreLoadImage` 乍一看没问题，但其实违反了我们设计原则中的**单一职责原则。**`PreLoadImage` 不仅要负责图片的加载，还要负责 DOM 层面的操作（img 节点的初始化和后续的改变）。这样一来，就**出现了两个可能导致这个类发生变化的原因。**

好的做法是将两个逻辑分离，让 `PreLoadImage` 专心去做 DOM 层面的事情（真实 DOM 节点的获取、img 节点的链接设置），再找一个对象来专门来帮我们搞加载——这两个对象之间缺个媒婆，这媒婆非代理器不可：

```
class PreLoadImage {
    constructor(imgNode) {
        // 获取真实的DOM节点
        this.imgNode = imgNode
    }

    // 操作img节点的src属性
    setSrc(imgUrl) {
        this.imgNode.src = imgUrl
    }
}

class ProxyImage {
    // 占位图的url地址
    static LOADING_URL = 'xxxxxx'

    constructor(targetImage) {
        // 目标Image，即PreLoadImage实例
        this.targetImage = targetImage
    }

    // 该方法主要操作虚拟Image，完成加载
    setSrc(targetUrl) {
       // 真实img节点初始化时展示的是一个占位图
        this.targetImage.setSrc(ProxyImage.LOADING_URL)
        // 创建一个帮我们加载图片的虚拟Image实例
        const virtualImage = new Image()
        // 监听目标图片加载的情况，完成时再将DOM上的真实img节点的src属性设置为目标图片的url
        virtualImage.onload = () => {
            this.targetImage.setSrc(targetUrl)
        }
        // 设置src属性，虚拟Image实例开始加载图片
        virtualImage.src = targetUrl
    }
}
```

`ProxyImage` 帮我们调度了预加载相关的工作，我们可以通过 `ProxyImage` 这个代理，实现对真实 img 节点的间接访问，并得到我们想要的效果。

在这个实例中，`virtualImage` 这个对象是一个“幕后英雄”，它始终存在于 JavaScript 世界中、代替真实 DOM 发起了图片加载请求、完成了图片加载工作，却从未在渲染层面抛头露面。因此这种模式被称为“虚拟代理”模式。

缓存代理

缓存代理比较好理解，它应用于一些计算量较大的场景里。在这种场景下，我们需要“用空间换时间”——当我们需要用到某个已经计算过的值的时候，不想再耗时进行二次计算，而是希望能从内存里去取出现成的计算结果。这种场景下，就需要一个代理来帮我们在进行计算的同时，进行计算结果的缓存了。

一个比较典型的例子，是对传入的参数进行求和：

```
// addAll方法会对你传入的所有参数做求和操作
const addAll = function() {
    console.log('进行了一次新计算')
    let result = 0
    const len = arguments.length
    for(let i = 0; i < len; i++) {
        result += arguments[i]
    }
    return result
}

// 为求和方法创建代理
const proxyAddAll = (function(){
    // 求和结果的缓存池
    const resultCache = {}
    return function() {
        // 将入参转化为一个唯一的入参字符串
        const args = Array.prototype.join.call(arguments, ',')

        // 检查本次入参是否有对应的计算结果
        if(args in resultCache) {
            // 如果有，则返回缓存池里现成的结果
            return resultCache[args]
        }
        return resultCache[args] = addAll(...arguments)
    }
})()
```

我们把这个方法丢进控制台，尝试同一套入参两次，结果喜人：

```
> proxyAddAll(1,2,3,4,5,6)
进行了一次新计算
< 21
> proxyAddAll(1,2,3,4,5,6)
< 21
```

我们发现 proxyAddAll 针对重复的入参只会计算一次，这将大大节省计算过程中的时间开销。现在我们有 6 个入参，可能还看不出来，当我们针对大量入参、做反复计算时，缓存代理的优势将得到更充分的凸显。

保护代理

保护代理，其实在我们上个小节大家就见识过了。此处我们仅作提点，不作重复演示。

开婚介所的时候，为了保护用户的私人信息，我们会在同事哥访问小美的年龄的时候，去校验同事哥是否已经通过了我们的实名认证；为了确保婚介所的利益同事哥确实是一位有诚意的男士，当他想获取小美的联系方式时，我们会校验他是否具有VIP 资格。所谓“保护代理”，就是在访问层面做文章，在 getter 和 setter 函数里去进行校验和拦截，确保一部分变量是安全的。

值得一提的是，上节中我们提到的 Proxy，它本身就是为拦截而生的，所以我们目前实现保护代理时，考虑的首要方案就是 ES6 中的 Proxy。

小结

代理模式行文至此，相信大家都已经做到了心中有数。在本节，我们看到代理模式的目的是十分多样化的，既可以是为了加强控制、拓展功能、提高性能，也可以仅仅是为了优化我们的代码结构、实现功能的解耦。无论是出于什么目的，这种模式的套路就只有一个—— A 不能直接访问 B，A 需要借助一个帮手来访问 B，这个帮手就是代理器。需要代理器出面解决的问题，就是代理模式发光发热的应用场景。

从本小节开始，结构型模式的讲解也将告一段落。下个小节，我们将开始最后的征程，进入行为型模式的世界。

（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~）