6 代理模式

代理模式是为一个对象提供一个代用品或者占位符，以便控制对它的访问

这种模式非常有意义，例如现实生活中的明星经纪人作为代理，客户实际上访问的是替身对象，由替身对象做出一些处理之后再转交给本体对象

第一个例子-小明追MM的故事

小明有一个女神A，然后打听到A和他有一个共同的朋友B，决定让B代替自己完成送花这件事情

```
/**
 * 不用代理B
 */
var Flower = function(){};

var xiaoming = {
    sendFlower: function(target){
        var flower = new Flower();
        target.receiveFlower(flower);
    }
}

var A = {
    receiveFlower: function(flower){
        console.log('收到花' + flower)
    }
}

xiaoming.sendFlower(A);

/**
 * 加入代理B
 */

// ...
var B = {
    receiveFlower: function(flower){
        A.receiveFlower(flower);
    }
}
```

```
31
32 xiaoming.sendFlower(B);
```

上述代理B看上去毫无用处，只是简单转发了一下，但是实际情况下代理B可能更了解女神A，B会监听A的心情变化并在适当的时候转交花，提高小明的成功率：

```
1  /**
2   * 代理B发挥用处了
3   */
4  var A = {
5      receiveFlower: function(flower){
6          console.log('收到花' + flower)
7      },
8      listenGoodMood: function(fn){
9          // 假设10s条件后心情变化
10         setTimeout(() => {fn()}, 10000)
11     }
12 }
13
14 var B = {
15     receiveFlower: function(flower){
16         A.listenGoodMood(() => {
17             A.receiveFlower(flower);
18         })
19     }
20 }
21
22 xiaoming.sendFlower(B);
```

这里是不是能体会到一点proxy的价值了呢？他更懂目标，也将小明和女神解耦开来了

保护代理和虚拟代理

- 保护代理：如果上述代理B的工作是过滤一些不合理或者无用的对A的请求，那么就叫做保护代理
  - 例如年龄太大或者太穷的表白对象
  - JS中我们无法判断谁访问了某个对象，出发加上业务标记
- 虚拟代理：假设之前new Flower的代价比较大，我们可以将new的职责交给B，在A心情好的时候再去花费开销new Flower，延迟到真正需要它的时候再进行
  - 本章主要讨论的是虚拟代理

真实示例-虚拟代理实现图片预加载

Web开发中，图片预加载是常用的技术，因为直接给img的src设置图片链接，有时候会因为网络或者图片过大等原因造成空白。通常做法线设置loading图片展位，然后异步加载图片，之后再回填到img节点，这个场合就很适合虚拟代理：

```
/**
 * 图片预加载
 */

var myImage = (function(){
    var imgNode = document.createElement('img');
    document.body.appendChild(imgNode);

    return {
        setSrc: function(src){
            imgNode.src = src;
        }
    }
})()

// 引入代理对象proxy，避免页面空白 - 图片被加载好之前会出现菊花图loading.gif占位

var proxyImage = (function(){
    var img = new Image;
    img.onload = function(){
        myImage.setSrc(img.src);
    }
    return {
        setSrc: function(src){
            // 加载本地占位
            myImage.setSrc('file://C:/Users/loading.gif');
            // Image对象去异步加载真正的图片-加载完成后通过onload顶替myImage
            img.src = src;
        }
    }
})()

proxyImage.setSrc('http://xxx/x.png');
```

当然还可以在proxy的处理过程当中增加一些额外的操作，为真正的对象访问做好提前处理

代理的意义

看了上面的例子，你可以想问代理模式究竟有什么好处呢?

这里我们先引入一个面向对象设计的原则 - 单一职责原则：

> ❤️ 一个类/对象/函数而言，应该仅有一个引起它变化的原则，如果一个对象承担了多项职责，那么引起它变化的原因就会有多个，所以鼓励将行为分布到细粒度的对象之中。

之前代码中的MyImage对象除了给img节点设置src之外，可能还要负责处理Loading预览图，这时的强耦合性就可能影响另外一个职责的实现，例如某时候我们需要去掉Loading逻辑，那么就必须更改MyImage对象了。

这个时候代理的作用就有了，我们就可以将锦上添花的预加载图片功能交由代理负责，本体仍然负责真正src的处理 ---> 同时也符合开放/封闭原则

代理和本体接口的一致性

上一节提到一点：某一天不需要Loading逻辑了，可以选择直接调用本体的setSrc方法

这里的关键是proxy和本体都拥有setSrc方法，从用户侧看来大家都是一致的，这样的处理有这些好处：

- 用户无需额外的心智负担
- 任何使用本体的地方都可以使用代理来增强

Java中需要proxy和本体都显式的实现同一个接口，而在JS这种动态语言当中，我们有时会通过鸭子类型来检测代理和本体是否都实现了setSrc方法

> 👍 在Typescript当中，我们可以通过定义type/interface以及extends等方式来规范化proxy和本体的使用接口，Javascript可能只能凭自觉和鸭子校验了