

## 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

```
1  /**
2   * 不用代理B
3   */
4  var Flower = function(){};
5
6  var xiaoming = {
7      sendFlower: function(target){
8          var flower = new Flower();
9          target.receiveFlower(flower);
10     }
11 }
12
13 var A = {
14     receiveFlower: function(flower){
15         console.log('收到花' + flower)
16     }
17 }
18
19 xiaoming.sendFlower(A);
20
21 /**
22 * 加入代理B
23 */
24
25 // ...
26 var B = {
27     receiveFlower: function(flower){
28         A.receiveFlower(flower);
29     }
30 }
```

```
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。通常做法先设置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：

```
1  /**
2   * 图片预加载
3   */
4
5  var myImage = (function(){
6      var imgNode = document.createElement('img');
7      document.body.appendChild(imgNode);
8
9      return {
10         setSrc: function(src){
11             imgNode.src = src;
12         }
13     }
14 })()
15
16 // 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
17
18 var proxyImage = (function(){
19     var img = new Image;
20     img.onload = function(){
21         myImage.setSrc(img.src);
22     }
23     return {
24         setSrc: function(src){
25             // 加载本地占位
26             myImage.setSrc('file:///C:/Users/loading.gif');
27             // Image对象去异步加载真正的图片-加载完成后通过onload顶替myImage
28             img.src = src;
29         }
30     }
31 })()
32
33 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

## 虚拟代理合并Http请求

在Web开发中，也许最大的开销就是网络请求了，在诸如网页指标监控上报，批量选中目标并发送请求等场景中我们如果不进行优化会造成非常频繁的网络请求。

一种解决方案是我们可以通过一个代理函数proxySynchronousFile来收集一段时间内的请求，然后一次性发送给服务器，在对实时性要求不高的业务和系统中，几秒的延迟并不会会有副作用，但是可以大大减轻服务器的压力。

```

1  /**
2   * 代理对象合并http请求
3   */
4
5  var synchronousFile = function(id){
6      console.log('开始同步文件,id为:' + id);
7  };
8
9  var proxySynchronousFile = (function(){
10     var cache = [], timer;
11     return function(id){
12         // 保证不会覆盖已经启动的定时器
13         if(timer){
14             return;
15         }
16
17         timer = setTimeout(function(){
18             synchronousFile(id);
19             clearTimeout(timer);
20             timer = null;
21             cache.length = 0;
22         },2000)
23     }
24 })()
25
26 var checkbox = document.getElementsByTagName('input');
27 for(var i = 0, c; c = checkbox[i++];){
28     c.onclick = function(){
29         if(this.checked === true){
30             proxySynchronousFile(this.id)
31         }
32     }
33 }

```

## 虚拟代理在惰性加载中应用

很多时候我们项目会加载一些工具JS文件，这些JS文件没有必要一开始就加载进来，因为不是所有用户都需要里面的功能，希望再用户要使用时候再去加载。

在JS加载之前，为了能够让用户正常使用里面的API，通常我们解决方案是提供一个占位的JS代理对象，提供的接口跟真正的JS文件一致。

例如一个Console调试台工具的例子：

```

1  /**
2   * 惰性加载console.js工具类
3   */
4  var cache = [];
5
6  // 未加载真正console.js之前的代码
7  var miniConsole = {
8      log: function(){
9          var args = arguments;
10         cache.push(function(){
11             return miniConsole.log.apply(miniConsole, args);
12         })
13     }
14 }
15
16 miniConsole.log(1);
17
18 // 用户按下F2, 开始加载真正的miniConsole.js
19 document.body.addEventListener('keydown', (ev) => {
20     if(ev.code === 113){
21         var script = document.createElement('script');
22         script.onload = function(){
23             for(var i = 0, fn; fn = cache[i++];){
24                 fn();
25             }
26         }
27         script.src = 'miniConsole.js';
28         document.getElementsByTagName('head')[0].appendChild(script);
29     }
30 }, false);
31
32 // 真正的miniConsole.js代码
33 miniConsole = {
34     log: function(){
35         // 真正的代码略
36         console.log(Array.prototype.join.call(arguments));
37     }
38 }

```

## 缓存代理

缓存代理可以为一些开销大的运算结果提供暂时的存储, 如果下次运算时的入参一致, 则直接返回缓存结果, 节约计算耗时, 这里我们可以尝试和高阶函数相结合, 为各种算法函数提供缓存代理支

持：

```
1  /**
2   * 缓存代理(脑补耗时计算过程)
3   */
4
5  // 乘积算法
6  var mult = function(){
7      var a = 1;
8      for(var i = 0, l = arguments.length; i < l; i++){
9          a = a*arguments[i];
10     }
11     return a;
12 }
13
14 // 累加算法
15 var plus = function(){
16     var a = 1;
17     for(var i = 0, l = arguments.length; i < l; i++){
18         a = a+arguments[i];
19     }
20     return a;
21 }
22
23 // 缓存代理包裹
24 var createProxyFactory = function(fn){
25     var cache = {};
26     return function(){
27         var args = Array.prototype.join.call(arguments, ',');
28         if(args in cache){
29             return cache[args];
30         }
31         return cache[args] = fn.apply(this, arguments);
32     }
33 }
34
35 var proxyMult = createProxyFactory(mult);
36 var proxyPlus = createProxyFactory(plus);
```

## 其他代理模式

代理模式的变体很多，这里简单介绍下：

- 防火墙代理：控制网络资源的访问，保护主体不让坏人接近
- 远程代理：为一个对象在不同的地址空间提供代理，可能是另一个远程机的对象
- 保护代理：用于对象不同访问权限的情况
- 智能引用代理：取代了简单的指针，在访问对象时可以执行一些附加操作

## 小结

在JS开发中最常用的代理模式是虚拟代理和缓存代理，注意在我们编写业务时，往往不需要先去猜测是否使用代理模式，当真正发现不方便直接访问某个对象的时候，再编写代理也不迟