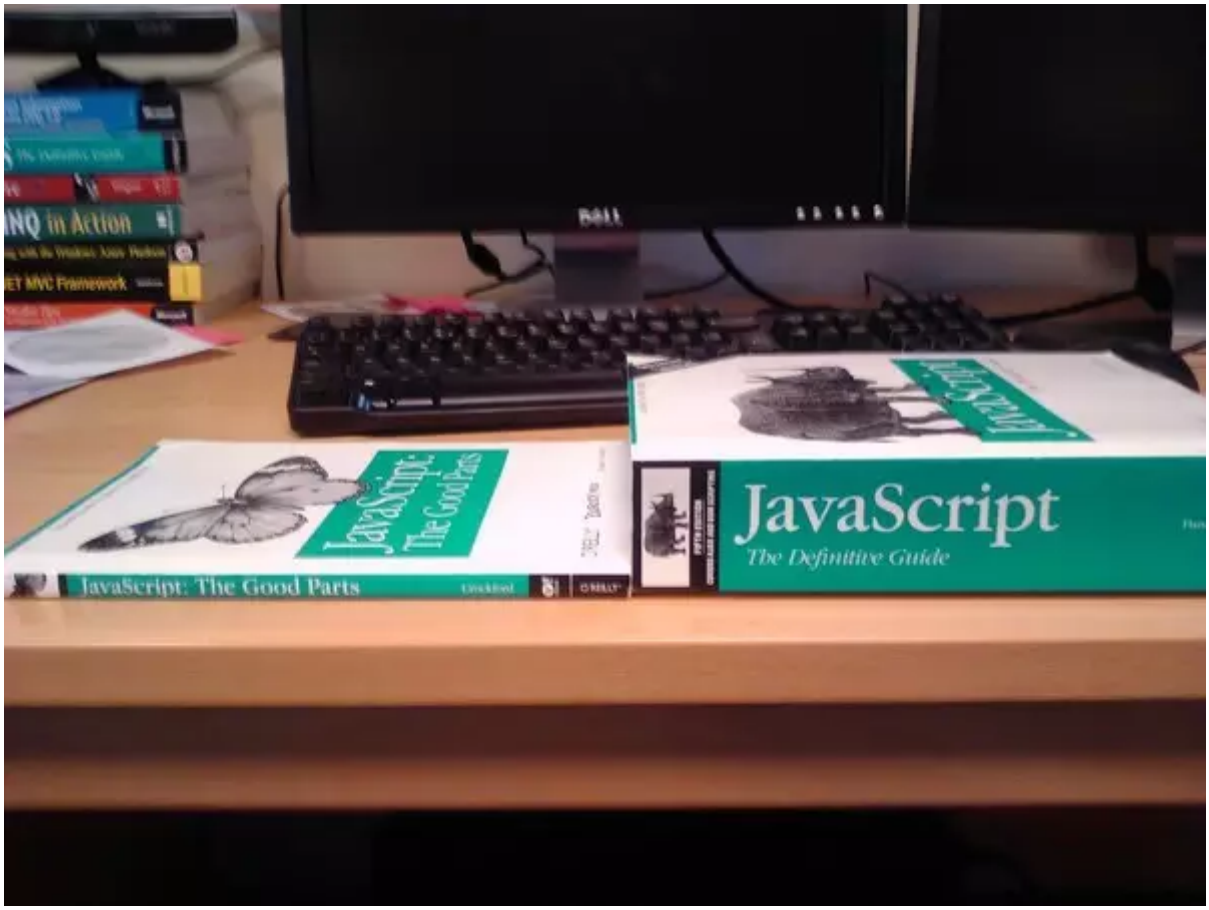
如何写出好的 JavaScript —— 浅谈 API 设计

2017-09-14 前端大全

（点击上方公众号，可快速关注）

> 作者：十年踪迹
> www.h5jun.com/post/how-to-write-better-js-code.html
> 如有好文章投稿，请点击 → 这里了解详情

很多同学觉得写 JavaScript 很简单，只要能写出功能来，效果能实现就好。还有一些培训机构，专门教人写各种“炫酷特效”，以此让许多人觉得这些培训很“牛逼”。然而事实上，能写 JavaScript 和写好 JavaScript 这中间还有很遥远的距离。成为专业前端，注定在 JavaScript 路途上需要一步步扎实的修炼，没有捷径。



看一个简单的例子：

实现一个类似于“交通灯”的效果，让三个不同颜色的圆点每隔 2 秒循环切换。

对应的 HTML 和 CSS 如下：

```
<ul id="traffic" class="wait">
  <li><span></span></li>
```

```
  <li><span></span></li>
  <li><span></span></li>
</ul>
```

```
#traffic > li{
  display: block;
}

#traffic span{
  display: inline-block;
  width: 50px;
  height: 50px;
  background-color: gray;
  margin: 5px;
  border-radius: 50%;
}

#traffic.stop li:nth-child(1) span{
  background-color: #a00;
}

#traffic.wait li:nth-child(2) span{
  background-color: #aa0;
}

#traffic.pass li:nth-child(3) span{
  background-color: #0a0;
}
```

那么这一功能的 JavaScript 该如何实现呢?

版本一

有的同学说，这个实现还不简单嘛？直接用几个定时器一下切换不就好了：

```
const traffic = document.getElementById("traffic");

(function reset(){
  traffic.className = "wait";

  setTimeout(function(){
```

```
    traffic.className = "stop";
    setTimeout(function(){
      traffic.className = "pass";
      setTimeout(reset, 2000)
    }, 2000)
  }, 2000);
})();
```

没错，就这个功能本身，这样实现就 OK 了。但是这样实现有什么问题呢?

首先是**过程耦合**，状态切换是wait->stop->pass 循环，在上面的设计里，实际上操作顺序是耦合在一起的，要先 ‘wait’ ，然后等待 2000 毫秒再 ‘stop’ ，然后再等待 2000 毫秒在 ‘pass’ ，这中间的顺序一旦有调整，需求有变化，代码都需要修改。

其次，这样的异步嵌套是会产生 **callback hell** 的，如果需求不是三盏灯，而是五盏灯、十盏灯，代码的嵌套结构就很深，看起来就很难看了。

所以我们说，版本一方法虽然直接，但因为抽象程度很低（几乎没有提供任何抽象 API），它的扩展性很不好，因为异步问题没处理，代码结构也很不好。如果只能写这样的代码，是不能说就写好了 JavaScript 的。

版本二

要解决版本一的**过程耦合**问题，最简单的思路是将状态['wait','stop','pass']抽象出来:

```
const traffic = document.getElementById("traffic");

var stateList = ["wait", "stop", "pass"];

var currentStateIndex = 0;

setInterval(function(){
  var state = stateList[currentStateIndex];
  traffic.className = state;
  currentStateIndex = (currentStateIndex + 1) % stateList.length;
}, 2000);
```

这是一种数据抽象的思路，应用它我们得到了上面的这个版本。

这一版本比前一版本要好很多，但是它也有问题，最大的问题就是**封装性很差**，它把 stateList 和 currentStateIndex 都暴露出来了，而且以全局变量的形式，这么做很不好，需要优化。

版本三

版本三是中规中矩的一版，也是一般我们在工作中比较常用的思路。应该将暴露出来的 API 暴露出来（本例中的 stateList）。将不应该暴露出来的数据或状态隐藏（本例中的 currentStateIndex）。

有许多同学觉得说写出这一版本来已经很不错的。的确，应该也还不错，但这一版的抽象程度其实也不是很高，或者说，如果考虑适用性，这版已经很好了，但是如果考虑可复用性的话，这版依然有改进空间。

我们再看一个思路上较有意思的版本。

版本四

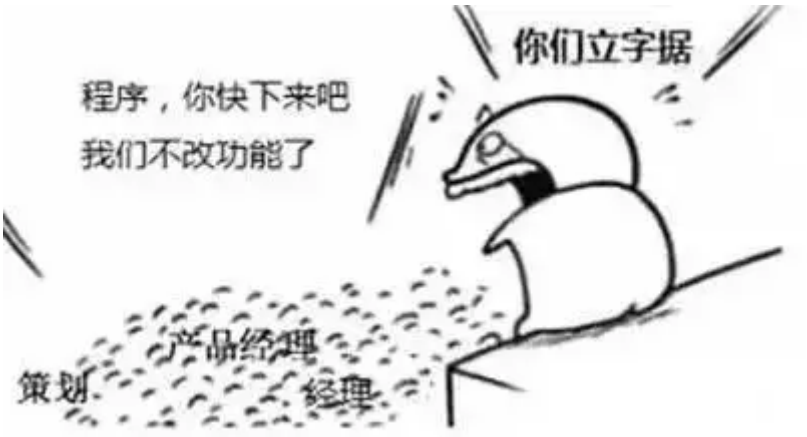
```
const traffic = document.getElementById("traffic");

function poll(...fnList){
  let stateIndex = 0;

  return function(...args){
    let fn = fnList[stateIndex++ % fnList.length];
    return fn.apply(this, args);
  }
}

function setState(state){
  traffic.className = state;
}

let trafficStatePoll = poll(setState.bind(null, "wait"),
                            setState.bind(null, "stop"),
                            setState.bind(null, "pass"));

setInterval(trafficStatePoll, 2000);
```

这一版用的是**过程抽象**的思路，而过程抽象，是**函数式编程**的基础。在这里，我们抽象出了一个 `poll(...fnList)` 的高阶组合函数，它将一个函数列表组合起来，每次调用时依次轮流执行列表里的函数。
我们说，程序设计的本质是抽象，而**过程抽象**是一种与**数据抽象**对应的思路，它们是两种不同的抽象模型。数据抽象比较基础，而过程抽象相对高级一些，也更灵活一些。数据抽象是研究函数如何操作数据，而过程抽象则在此基础上研究函数如何操作函数。所以说如果把抽象比作数学，那么数据抽象是初等数学，过程抽象则是高等数学。同一个问题，既可以用初等数学来解决，又可以用高等数学来解决。用什么方法解决，取决于问题的模型和难度等等。

好了，上面我们有了四个版本，那么是否考虑了这些版本就足够了呢？

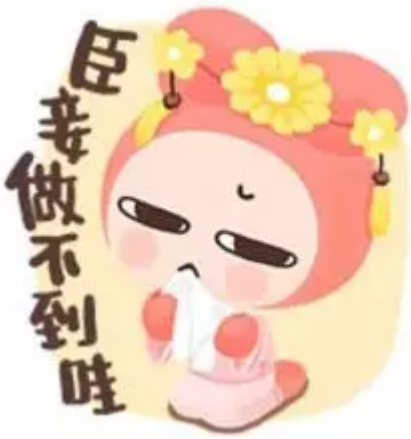
并不是。因为需求是会变更的。假设现在需求变化了：

需求变更：让 wait、stop、pass 状态的持续时长不相等，分别改成 1秒、2秒、3秒。



那么，我们发现 ——

除了版本一之外，版本二、三、四全都跪了……



那是否意味着我们要**回归到版本一**呢？

当然并不是。

版本五

```
const traffic = document.getElementById("traffic");

function wait(time){
  return new Promise(resolve => setTimeout(resolve, time));
}
```

```
function setState(state){
  traffic.className = state;
}

function reset(){
  Promise.resolve()
    .then(setState.bind(null, "wait"))
    .then(wait.bind(null, 1000))
    .then(setState.bind(null, "stop"))
    .then(wait.bind(null, 2000))
    .then(setState.bind(null, "pass"))
    .then(wait.bind(null, 3000))
    .then(reset);
}

reset();
```

版本五的思路是，**既然我们需要考虑不同的持续时间，那么我们需要将等待时间抽象出来**：

```
function wait(time){
  return new Promise(resolve => setTimeout(resolve, time));
}
```

这一版本里我们用了 Promise 来处理回调问题，当然对 ES6 之前的版本，可以用 shim 或 polyfill、第三方库，也可以选择不用 Promise。

版本五抽象出的 wait 方法也还比较通用，可以用在其他地方。这是版本五好的一点。

版本六

我们还可以进一步抽象，设计出版本六，或者类似的**对象模型**：

```
const trafficEl = document.getElementById("traffic");

function TrafficProtocol(el, reset){
  this.subject = el;
  this.autoReset = reset;
  this.stateList = [];
}

TrafficProtocol.prototype.putState = function(fn){
  this.stateList.push(fn);
```

```
}

TrafficProtocol.prototype.reset = function(){
  let subject = this.subject;

  this.statePromise = Promise.resolve();
  this.stateList.forEach((stateFn) => {
    this.statePromise = this.statePromise.then(()=>{
      return new Promise(resolve => {
        stateFn(subject, resolve);
      });
    });
  });
  if(this.autoReset){
    this.statePromise.then(this.reset.bind(this));
  }
}

TrafficProtocol.prototype.start = function(){
  this.reset();
}

var traffic = new TrafficProtocol(trafficEl, true);

traffic.putState(function(subject, next){
  subject.className = "wait";
  setTimeout(next, 1000);
});

traffic.putState(function(subject, next){
  subject.className = "stop";
  setTimeout(next, 2000);
});

traffic.putState(function(subject, next){
  subject.className = "pass";
  setTimeout(next, 3000);
});

traffic.start();
```

这一版本里，我们设计了一个 TrafficProtocol 类，它有 putState、reset、start 三个方法：

- putState 接受一个函数作为参数，这个函数自身有两个参数，一个是 subject，是由 TrafficProtocol 对象初始化时设定的 DOM 元素，一个是 next，是一个函数，表示结束当前 state，进入下一个 state。
- reset 结束当前状态循环，开始新的循环。
- start 开始执行循环，这里的实现是直接调用 reset。

看一下 reset 的实现思路：

```
TrafficProtocol.prototype.reset = function(){
  let subject = this.subject;

  this.statePromise = Promise.resolve();
  this.stateList.forEach((stateFn) => {
    this.statePromise = this.statePromise.then(()=>{
      return new Promise(resolve => {
        stateFn(subject, resolve);
      });
    });
  });
  if(this.autoReset){
    this.statePromise.then(this.reset.bind(this));
  }
}
```

在这里我们创建一个 statePromise，然后将 stateList 中的方法（通过 putState 添加的）依次绑定到 promise 上。如果设置了 autoReset，那么我们在 promise 的最后绑定 reset 自身，这样就实现了循环切换。

有了这个模型，我们要添加新的状态，只需要通过 putState 添加一个新的状态就好了。这一模型不仅仅可以用在这个需求里，还可以用在任何需要顺序执行异步请求的地方。

最后，我们看到，版本六用到了面向对象、过程抽象、Promise等模式，它的优点是 API 设计灵活，通用性和扩展性好。但是版本六也有缺点，它的实现复杂度比前面的几个版本都高，我们在做这样的设计时，也需要考虑是否有**过度设计**的嫌疑。

总结

- 设计是把双刃剑，繁简需要权衡，尺度需要把握。
- 写代码简单，程序设计不易，需要走心。