



下载APP



国庆策划 | 快来看看怎么用原生JavaScript实现手势解锁组件

2020-10-05 月影

跟月影学可视化

[进入课程 >](#)



讲述：月影

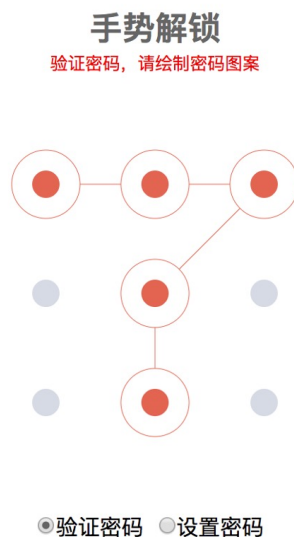
时长 00:46 大小 732.42K



你好，我是月影。前几天，我给你出了一道实操题，不知道你完成得怎么样啦？

今天，我就给你一个 [🔗 参考版本](#)。当然，并不是说这一版就最好，而是说，借助这一版的实现，我们就能知道当遇到这样比较复杂的 UI 需求时，我们应该怎样思考和实现。





首先，组件设计一般来说包括 7 个步骤，分别是理解需求、技术选型、结构（UI）设计、数据和 API 设计、流程设计、兼容性和细节优化，以及工具和工程化。

当然了，并不是每个组件设计的时候都需要进行这些过程，但一个项目总会在其中一些过程里遇到问题需要解决。所以，下面我们来做一个简单的分析。

理解需求

上节课的题目本身只是说设计一个常见的手势密码的 UI 交互，那我们就可以通过选择验证密码和设置密码来切换两种状态，每种状态有自己的流程。

如果你就照着需求把整个组件的状态切换和流程封装起来，或者只是提供了一定的 UI 样式配置能力的话，还远远不够。实际上这个组件如果要给用户使用，我们需要将过程节点开放出来。也就是说，**需要由使用者决定设置密码的过程里执行什么操作、验证密码的过程和密码验证成功后执行什么操作**，这些是组件开发者无法代替使用者来决定的。

复制代码

```
1 var password = '11121323';
2
3
4 var locker = new HandLock.Locker({
5   container: document.querySelector('#handlock'),
6   check: {
```

```
7     checked: function(res){
8         if(res.err){
9             console.error(res.err); //密码错误或长度太短
10            [执行操作...]
11        }else{
12            console.log(`正确, 密码是: ${res.records}`);
13            [执行操作...]
14        }
15    },
16 },
17 update:{
18     beforeRepeat: function(res){
19         if(res.err){
20             console.error(res.err); //密码长度太短
21             [执行操作...]
22         }else{
23             console.log(`密码初次输入完成, 等待重复输入`);
24             [执行操作...]
25         }
26     },
27     afterRepeat: function(res){
28         if(res.err){
29             console.error(res.err); //密码长度太短或者两次密码输入不一致
30             [执行操作...]
31         }else{
32             console.log(`密码更新完成, 新密码是: ${res.records}`);
33             [执行操作...]
34         }
35     },
36 },
37 });
38
39
40 locker.check(password)
```

技术选型

这个问题的 UI 展现的核心是九宫格和选中的小圆点，从技术上来讲，我们有三种可选方案：DOM/Canvas/SVG，三者都是可以实现主体 UI 的。那我们该怎么选择呢？

如果使用 DOM，最简单的方式是使用 flex 布局，这样能够做成响应式的。使用 DOM 的优点是容易实现响应式，事件处理简单，布局也不复杂（但是和 Canvas 比起来略微复杂），但是斜线（demo 里没有画）的长度和斜率需要计算。

除了使用 DOM 外，使用 Canvas 绘制也很方便。用 Canvas 实现有两个小细节，一是要实现响应式，我们可以用 DOM 构造一个正方形的容器。这里，我们使用 padding-

top:100% 撑开容器高度使它等于容器宽度。 代码如下:

[复制代码](#)

```
1 #container {  
2   position: relative;  
3   overflow: hidden;  
4   width: 100%;  
5   padding-top: 100%;  
6   height: 0px;  
7   background-color: white;  
8 }  
9
```

第二个细节是为了在 retina 屏上获得清晰的显示效果, 我们将 Canvas 的宽高增加一倍, 然后通过 transform: scale(0.5) 来缩小到匹配容器宽高。

[复制代码](#)

```
1 #container canvas{  
2   position: absolute;  
3   left: 50%;  
4   top: 50%;  
5   transform: translate(-50%, -50%) scale(0.5);  
6 }
```

由于 Canvas 的定位是 absolute, 它本身的默认宽高并不等于容器的宽高, 需要通过 JavaScript 设置。

[复制代码](#)

```
1 let width = 2 * container.getBoundingClientRect().width;  
2 canvas.width = canvas.height = width;  
3
```

使用上面的代码, 我们就可以通过在 Canvas 上绘制实心圆和连线来实现 UI 了。具体的方法, 我下面会详细来讲。

最后, 我们来看一下使用 SVG 的绘制方法。不过, 由于 SVG 原生操作的 API 不是很方便, 我们可以使用了 [Snap.svg 库](#), 实现起来和使用 Canvas 大同小异, 我就不详细来

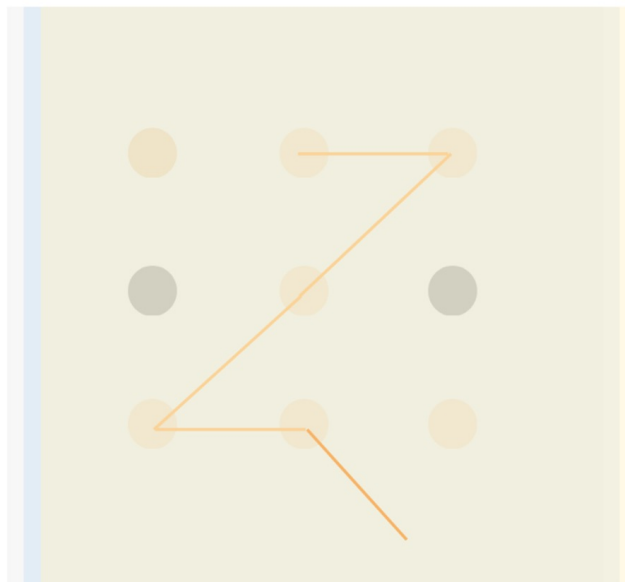
说了。但是，SVG 的问题是移动端兼容性不如 DOM 和 Canvas 好，所以综合上面三者的情况，我最终选择使用 Canvas 来实现。

结构设计

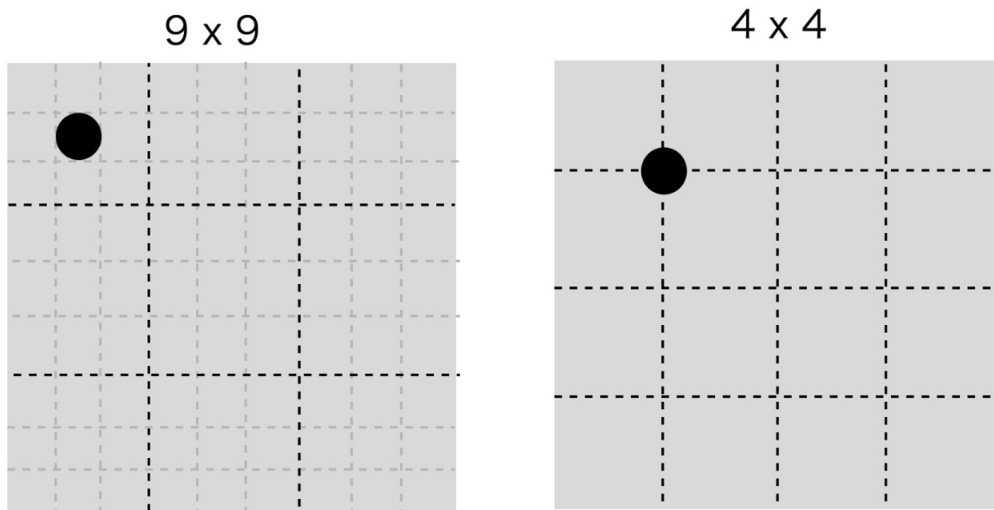
使用 Canvas 实现的话，DOM 结构就比较简单了。为了实现响应式，我们需要实现一个自适应宽度的正方形容器，方法前面已经讲过了，然后我们在容器中创建 Canvas。

这里需要注意的一点是，我们应当把 Canvas 分层。这是因为 Canvas 的渲染机制里，要更新画布的内容，需要刷新要更新的区域重新绘制。因此我们有必要把频繁变化的内容和基本不变的内容分层管理，这样能显著提升性能。

在这里我把 UI 分别绘制在 3 个图层里，对应 3 个 Canvas。最上层只有随着手指头移动的那个线段，中间是九个点，最下层是已经绘制好的线。之所以这样分，是因为随手指头移动的那条线需要不断刷新，底下两层都不用频繁更新，但是把连好的线放在最底层是因为我要做出圆点把线的一部分遮挡住的效果。



接着，我们确定圆点的位置。



圆点的位置有两种定位法，第一种是九个九宫格，圆点在小九宫格的中心位置。认真的同学肯定已经发现了，在前面 DOM 方案里，我们就是采用这样的方式。这个时候，圆点的直径为 11.1%。第二种方式是用横竖三条线把宽高四等分，圆点在这些线的交点处。

在 Canvas 里我们采用第二种方法来确定圆点（代码里的 $n = 3$ ）。

[复制代码](#)

```
1 let range = Math.round(width / (n + 1));
2
3
4 let circles = [];
5
6
7 //drawCircleCenters
8 for(let i = 1; i <= n; i++){
9   for(let j = 1; j <= n; j++){
10     let y = range * i, x = range * j;
11     drawSolidCircle(circleCtx, fgColor, x, y, innerRadius);
12     let circlePoint = {x, y};
13     circlePoint.pos = [i, j];
14     circles.push(circlePoint);
15   }
16 }
```


最后一点严格说不属于结构设计，但因为我们的 UI 是通过触屏操作，所以我们需要考虑 Touch 事件处理和坐标的转换。

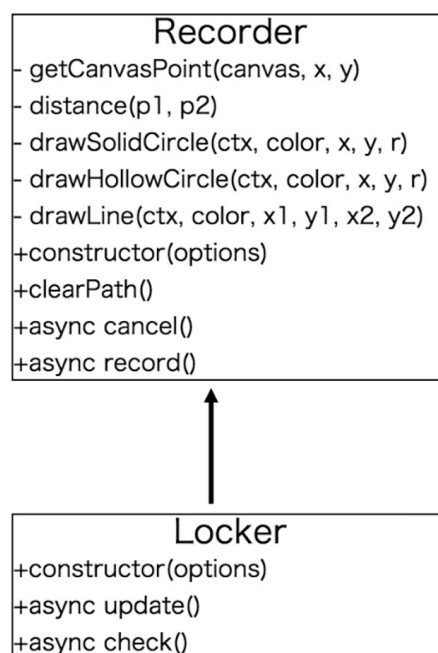
[复制代码](#)

```
1 function getCanvasPoint(canvas, x, y){
2   let rect = canvas.getBoundingClientRect();
3   return {
4     x: 2 * (x - rect.left),
5     y: 2 * (y - rect.top),
6   };
7 }
8
```

我们将 Touch 相对于屏幕的坐标转换为 Canvas 相对于画布的坐标。代码里的 2 倍是因为我们前面说了要让 retina 屏下清晰，我们将 Canvas 放大为原来的 2 倍。


API 设计

接下来我们需要设计给使用者使用的 API 了。在这里，我们将组件功能分解一下，独立出一个单纯记录手势的 Recorder。将组件功能分解为更加底层的组件，是一种简化组件设计的常用模式。



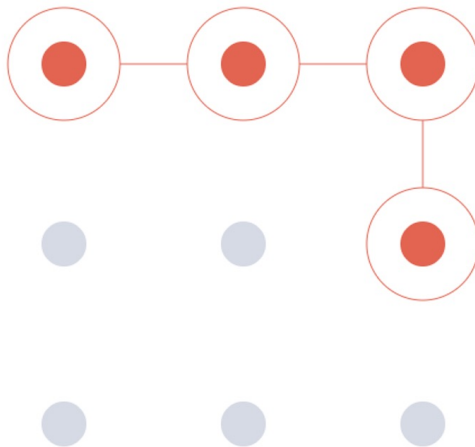
我们抽取出底层的 Recorder，让 Locker 继承 Recorder，Recorder 负责记录，Locker 管理实际的设置和验证密码的过程。

我们的 Recorder 只负责记录用户行为，由于用户操作是异步操作，我们将它设计为 Promise 规范的 API，它可以以如下方式使用：

 复制代码

```
1 var recorder = new HandLock.Recorder({
2   container: document.querySelector('#main')
3 });
4
5
6 function recorded(res){
7   if(res.err){
8     console.error(res.err);
9     recorder.clearPath();
10    if(res.err.message !== HandLock.Recorder.ERR_USER_CANCELED){
11      recorder.record().then(recorded);
12    }
13  }else{
14    console.log(res.records);
15    recorder.record().then(recorded);
16  }
17 }
18
19
20 recorder.record().then(recorded)
```

对于输出结果，我们简单用选中圆点的行列坐标拼接起来得到一个唯一的序列。例如“11121323” 就是如下选择图形：



为了让 UI 显示具有灵活性，我们还可以将外观配置抽取出来。

复制代码

```
1  const defaultOptions = {
2    container: null, //创建canvas的容器，如果不填，自动在 body 上创建覆盖全屏的层
3    focusColor: '#e06555', //当前选中的圆的颜色
4    fgColor: '#d6dae5', //未选中的圆的颜色
5    bgColor: '#fff', //canvas背景颜色
6    n: 3, //圆点的数量： n x n
7    innerRadius: 20, //圆点的内半径
8    outerRadius: 50, //圆点的外半径，focus 的时候显示
9    touchRadius: 70, //判定touch事件的圆半径
10   render: true, //自动渲染
11   customStyle: false, //自定义样式
12   minPoints: 4, //最小允许的点数
13 };
14
```

这样，我们实现完整的 Recorder 对象，核心代码如下：

复制代码

```
1  [...] //定义一些私有方法
2
3
4  const defaultOptions = {
5    container: null, //创建canvas的容器，如果不填，自动在 body 上创建覆盖全屏的层
```

```
6   focusColor: '#e06555', //当前选中的圆的颜色
7   fgColor: '#d6dae5',    //未选中的圆的颜色
8   bgColor: '#fff',       //canvas背景颜色
9   n: 3, //圆点的数量: n x n
10  innerRadius: 20, //圆点的内半径
11  outerRadius: 50, //圆点的外半径, focus 的时候显示
12  touchRadius: 70, //判定touch事件的圆半径
13  render: true, //自动渲染
14  customStyle: false, //自定义样式
15  minPoints: 4, //最小允许的点数
16 };
17
18
19 export default class Recorder{
20   static get ERR_NOT_ENOUGH_POINTS(){
21     return 'not enough points';
22   }
23   static get ERR_USER_CANCELED(){
24     return 'user canceled';
25   }
26   static get ERR_NO_TASK(){
27     return 'no task';
28   }
29   constructor(options){
30     options = Object.assign({}, defaultOptions, options);
31
32
33     this.options = options;
34     this.path = [];
35
36
37     if(options.render){
38       this.render();
39     }
40   }
41   render(){
42     if(this.circleCanvas) return false;
43
44
45     let options = this.options;
46     let container = options.container || document.createElement('div');
47
48
49     if(!options.container && !options.customStyle){
50       Object.assign(container.style, {
51         position: 'absolute',
52         top: 0,
53         left: 0,
54         width: '100%',
55         height: '100%',
56         lineHeight: '100%',
57         overflow: 'hidden',
```

```
58     backgroundColor: options.bgColor
59   });
60   document.body.appendChild(container);
61 }
62 this.container = container;
63
64
65 let {width, height} = container.getBoundingClientRect();
66
67
68 //画圆的 canvas, 也是最外层监听事件的 canvas
69 let circleCanvas = document.createElement('canvas');
70
71
72 //2 倍大小, 为了支持 retina 屏
73 circleCanvas.width = circleCanvas.height = 2 * Math.min(width, height);
74 if(!options.customStyle){
75   Object.assign(circleCanvas.style, {
76     position: 'absolute',
77     top: '50%',
78     left: '50%',
79     transform: 'translate(-50%, -50%) scale(0.5)',
80   });
81 }
82
83
84 //画固定线条的 canvas
85 let lineCanvas = circleCanvas.cloneNode(true);
86
87
88 //画不固定线条的 canvas
89 let moveCanvas = circleCanvas.cloneNode(true);
90
91
92 container.appendChild(lineCanvas);
93 container.appendChild(moveCanvas);
94 container.appendChild(circleCanvas);
95
96
97 this.lineCanvas = lineCanvas;
98 this.moveCanvas = moveCanvas;
99 this.circleCanvas = circleCanvas;
100
101
102 this.container.addEventListener('touchmove',
103   evt => evt.preventDefault(), {passive: false});
104
105
106 this.clearPath();
107 return true;
108 }
109 clearPath(){
```

```
110     if(!this.circleCanvas) this.render();
111
112
113     let {circleCanvas, lineCanvas, moveCanvas} = this,
114         circleCtx = circleCanvas.getContext('2d'),
115         lineCtx = lineCanvas.getContext('2d'),
116         moveCtx = moveCanvas.getContext('2d'),
117         width = circleCanvas.width,
118         {n, fgColor, innerRadius} = this.options;
119
120
121     circleCtx.clearRect(0, 0, width, width);
122     lineCtx.clearRect(0, 0, width, width);
123     moveCtx.clearRect(0, 0, width, width);
124
125
126     let range = Math.round(width / (n + 1));
127
128
129     let circles = [];
130
131
132     //drawCircleCenters
133     for(let i = 1; i <= n; i++){
134         for(let j = 1; j <= n; j++){
135             let y = range * i, x = range * j;
136             drawSolidCircle(circleCtx, fgColor, x, y, innerRadius);
137             let circlePoint = {x, y};
138             circlePoint.pos = [i, j];
139             circles.push(circlePoint);
140         }
141     }
142
143
144     this.circles = circles;
145 }
146 async cancel(){
147     if(this.recordingTask){
148         return this.recordingTask.cancel();
149     }
150     return Promise.resolve({err: new Error(Recorder.ERR_NO_TASK)});
151 }
152 async record(){
153     if(this.recordingTask) return this.recordingTask.promise;
154
155
156     let {circleCanvas, lineCanvas, moveCanvas, options} = this,
157         circleCtx = circleCanvas.getContext('2d'),
158         lineCtx = lineCanvas.getContext('2d'),
159         moveCtx = moveCanvas.getContext('2d');
```

```
162     circleCanvas.addEventListener('touchstart', ()=>{
163         this.clearPath();
164     });
165
166
167     let records = [];
168
169
170     let handler = evt => {
171         let {clientX, clientY} = evt.changedTouches[0],
172             {bgColor, focusColor, innerRadius, outerRadius, touchRadius} = optio
173             touchPoint = getCanvasPoint(moveCanvas, clientX, clientY);
174
175
176         for(let i = 0; i < this.circles.length; i++){
177             let point = this.circles[i],
178                 x0 = point.x,
179                 y0 = point.y;
180
181
182             if(distance(point, touchPoint) < touchRadius){
183                 drawSolidCircle(circleCtx, bgColor, x0, y0, outerRadius);
184                 drawSolidCircle(circleCtx, focusColor, x0, y0, innerRadius);
185                 drawHollowCircle(circleCtx, focusColor, x0, y0, outerRadius);
186
187
188                 if(records.length){
189                     let p2 = records[records.length - 1],
190                         x1 = p2.x,
191                         y1 = p2.y;
192
193
194                     drawLine(lineCtx, focusColor, x0, y0, x1, y1);
195                 }
196
197
198                 let circle = this.circles.splice(i, 1);
199                 records.push(circle[0]);
200                 break;
201             }
202         }
203
204
205         if(records.length){
206             let point = records[records.length - 1],
207                 x0 = point.x,
208                 y0 = point.y,
209                 x1 = touchPoint.x,
210                 y1 = touchPoint.y;
211
212
213             moveCtx.clearRect(0, 0, moveCanvas.width, moveCanvas.height);
```

```
214     drawLine(moveCtx, focusColor, x0, y0, x1, y1);
215   }
216 };
217
218
219
220
221 circleCanvas.addEventListener('touchstart', handler);
222 circleCanvas.addEventListener('touchmove', handler);
223
224
225 let recordingTask = {};
226 let promise = new Promise((resolve, reject) => {
227   recordingTask.cancel = (res = {}) => {
228     let promise = this.recordingTask.promise;
229
230
231     res.err = res.err || new Error(Recorder.ERR_USER_CANCELED);
232     circleCanvas.removeEventListener('touchstart', handler);
233     circleCanvas.removeEventListener('touchmove', handler);
234     document.removeEventListener('touchend', done);
235     resolve(res);
236     this.recordingTask = null;
237
238
239     return promise;
240   }
241
242
243   let done = evt => {
244     moveCtx.clearRect(0, 0, moveCanvas.width, moveCanvas.height);
245     if(!records.length) return;
246
247
248     circleCanvas.removeEventListener('touchstart', handler);
249     circleCanvas.removeEventListener('touchmove', handler);
250     document.removeEventListener('touchend', done);
251
252
253     let err = null;
254
255
256     if(records.length < options.minPoints){
257       err = new Error(Recorder.ERR_NOT_ENOUGH_POINTS);
258     }
259
260
261     //这里可以选择一些复杂的编码方式, 本例子用最简单的直接把坐标转成字符串
262     let res = {err, records: records.map(o => o.pos.join('')).join('')};
263
264
265     resolve(res);
```

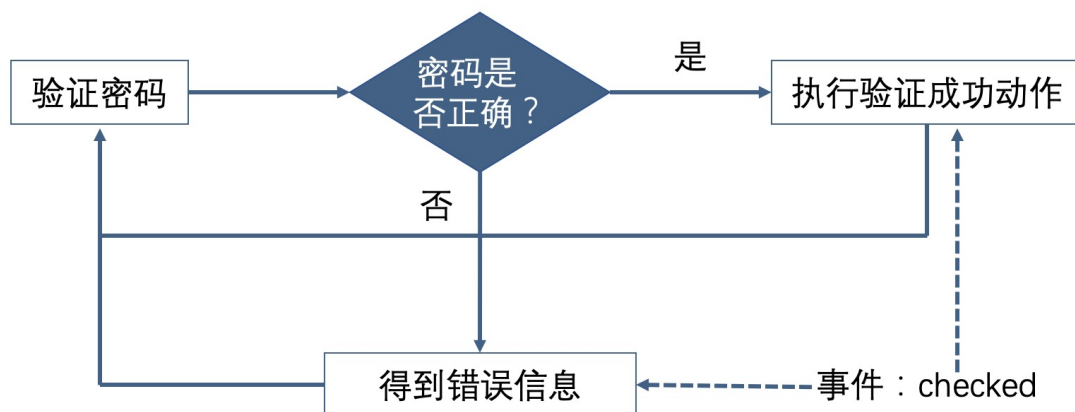
```
266     this.recordingTask = null;
267   };
268   document.addEventListener('touchend', done);
269 });
270
271
272   recordingTask.promise = promise;
273
274
275
276
```

这里有几个公开的方法，分别是 `ecorder` 负责记录绘制结果，`clearPath` 负责在画布上清除上一次记录的结果，`cancel` 负责终止记录过程，这是为后续流程准备的。

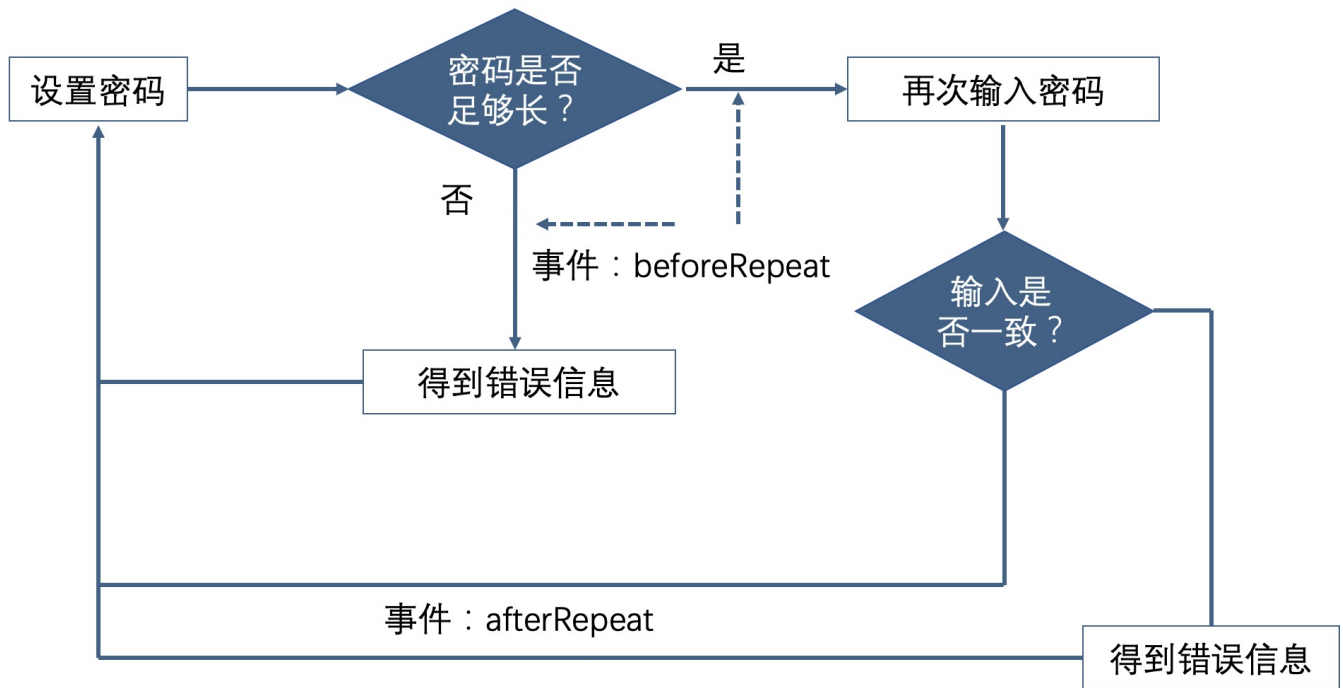
流程设计

接下来，我们基于 `Recorder` 来设计设置和验证密码的流程：

首先是验证密码的流程：



其次是设置密码的流程：



有了前面异步 Promise API 的 Recorder，我们不难实现上面的两个流程。

验证密码的内部流程

复制代码

```

1  async check(password){
2    if(this.mode !== Locker.MODE_CHECK){
3      await this.cancel();
4      this.mode = Locker.MODE_CHECK;
5    }
6
7
8    let checked = this.options.check.checked;
9
10
11   let res = await this.record();
12
13
14   if(res.err && res.err.message === Locker.ERR_USER_CANCELED){
15     return Promise.resolve(res);
16   }
17
18
19   if(!res.err && password !== res.records){
20     res.err = new Error(Locker.ERR_PASSWORD_MISMATCH)
21   }
22
23
24   checked.call(this, res);

```

```
25     this.check(password);
26     return Promise.resolve(res)
27 
```

设置密码的内部流程

[复制代码](#)

```
1  async update(){
2    if(this.mode !== Locker.MODE_UPDATE){
3      await this.cancel();
4      this.mode = Locker.MODE_UPDATE;
5    }
6
7
8    let beforeRepeat = this.options.update.beforeRepeat,
9        afterRepeat = this.options.update.afterRepeat;
10
11
12    let first = await this.record();
13
14
15    if(first.err && first.err.message === Locker.ERR_USER_CANCELED){
16      return Promise.resolve(first);
17    }
18
19
20    if(first.err){
21      this.update();
22      beforeRepeat.call(this, first);
23      return Promise.resolve(first);
24    }
25
26
27    beforeRepeat.call(this, first);
28
29
30    let second = await this.record();
31
32
33    if(second.err && second.err.message === Locker.ERR_USER_CANCELED){
34      return Promise.resolve(second);
35    }
36
37
38    if(!second.err && first.records !== second.records){
39      second.err = new Error(Locker.ERR_PASSWORD_MISMATCH);
40    }
41
42
```

```
43   this.update();  
44   afterRepeat.call(this, second);  
45   return Promise.resolve(se
```

我们可以看到，有了 Recorder 之后，Locker 的验证和设置密码基本上就是顺着流程用 async/await 写下来就行了。

另外，我们还要注意一些细节问题。由于实际在手机上触屏时，如果上下拖动，浏览器的默认行为会导致页面上下移动，因此我们需要阻止 touchmove 的默认事件。

[📄 复制代码](#)

```
1  this.container.addEventListener('touchmove',  
2    evt => evt.preventDefault(), {passive: false});  
3
```

touchmove 事件在 Chrome 下默认是一个 [🔗 Passive Event](#)，因此，我们 addEventListener 的时候需要传参 {passive: false}，否则就不能 preventDefault。

此外，因为我们的代码使用了 ES6+，所以需要引入 babel 编译，我们的组件也使用 webpack 进行打包，以便于使用者在浏览器中直接引入。

要点总结

今天，我和你一起完成了前几天留下的“手势密码”实战题。通过解决这几道题，我希望你能记住这三件事：

1. 在设计 API 的时候思考真正的需求，判断什么该开放、什么该封装
2. 做好技术调研和核心方案研究，选择合适的方案
3. 着手优化和解决细节问题，要站在 API 使用者的角度思考

源码

[🔗 GitHub 工程](#)

[提建议](#)

更多课程推荐

数据结构与算法之美

为工程师量身打造的数据结构与算法私教课

王争

前 Google 工程师



立省 ¥40



破 90000 订阅特惠, 到手价 ¥89

© 版权归极客邦科技所有, 未经许可不得传播售卖。页面已增加防盗追踪, 如有侵权极客邦将依法追究其法律责任。

上一篇 [国庆策划 | 假期别闲着, 一起来挑战“手势密码”](#)

下一篇 [加餐一 | 作为一名程序员, 数学到底要多好?](#)

精选留言

[写留言](#)

由作者筛选后的优质留言将会公开显示, 欢迎踊跃留言。