无需scroll事件就能监听到元素滚动

1. 前言

最近在做 toolTip 弹窗相关组件封装，实现的效果就是可以通过hover或点击在元素的上面或者下面能够出现一个弹框，类似下面这样

这时我遇到一个问题，因为我想当这个弹窗快要滚出屏幕之外时能够从由上面弹出变到由下面弹出，本来想着直接监听 `scroll` 事件就能搞定的，但是仔细一想 `scroll` 事件到底要绑定到那个 `DOM` 上呢？ 因为很多时候滚动条出现的元素并不是最外层的 `body` 或者 `html` 可能是任意一个元素上的滚动条。这个时候就无法通过绑定 `scroll` 事件来监听元素滚动了。

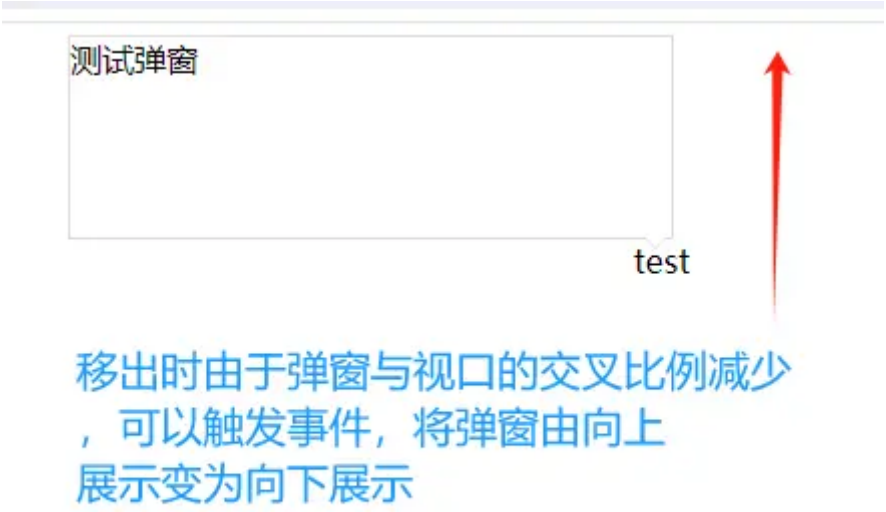
2. 问题分析

我脑海中首先 `IntersectionObserver` 这个 API，但是这个 API 只能用来 **监测目标元素与视窗(viewport)的交叉状态**，也就是当我的元素滚出或者滚入的时候可以触发该监听的回调。

```
new IntersectionObserver((event) => {
        refresh();
      }, {
       // threshold 用来表示元素在视窗中显示的交叉比例显示
       // 设置的是 0 即表示元素完全移出视窗，1 或者完全进入视窗时触发回调
       // 0表示元素本身在视口中的占比0%，1表示元素本身在视口中的占比为100%
       // 0.1表示元素本身在视口中的占比1%,0.9表示元素本身在视口中的占比为90%
        threshold: [0, 1, 0.1, 0.9]
```

```
9       });
```

这样就可以在元素快要移出屏幕，或者移入屏幕时触发回调了，但是这样会有一个问题

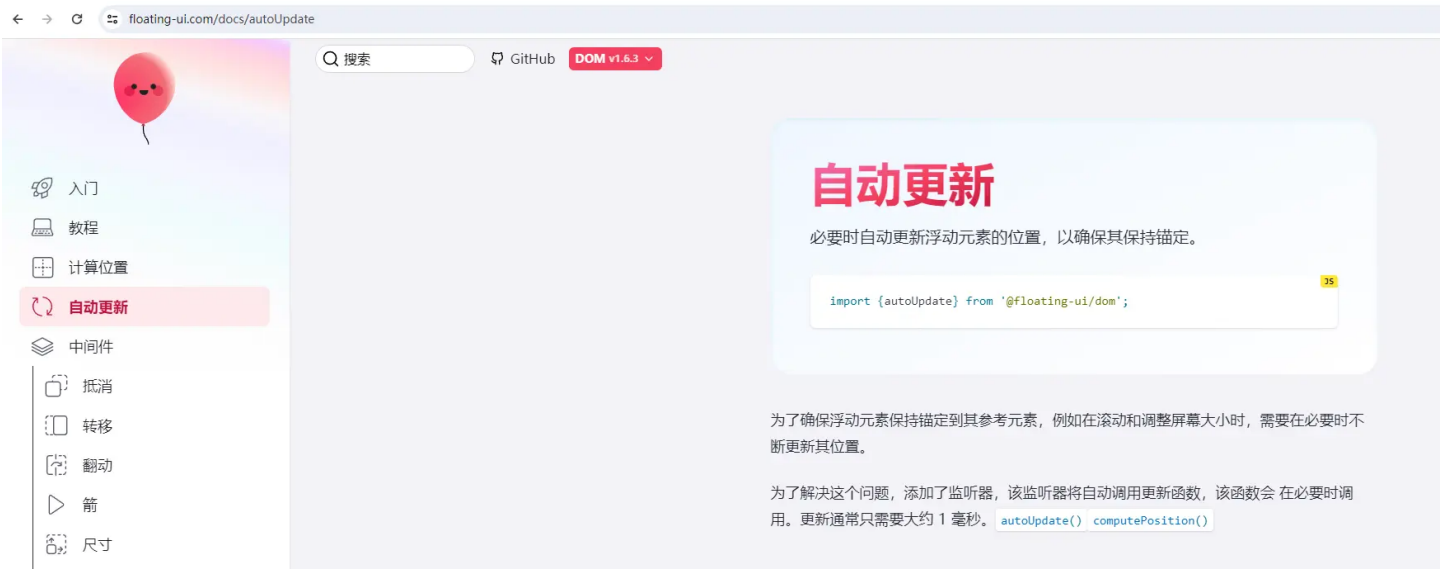


当弹窗移出屏幕时，可以很轻松的监听到，并把弹窗移动到下方，但是当弹窗滚入的时候就有问题了

可以看到完全进入之后，这个时候由于顶部空间不够，还需要继续往下滚才能将弹窗由底部移动到顶部。但是已经无法再触发 `IntersectionObserver` 和视口交叉的回调事件了，因为元素已经完全在视窗内了。 也就是说用这种方案，元素一旦滚出去之后，再回来的时候就无法复原了。

3. 把问题抛给别人

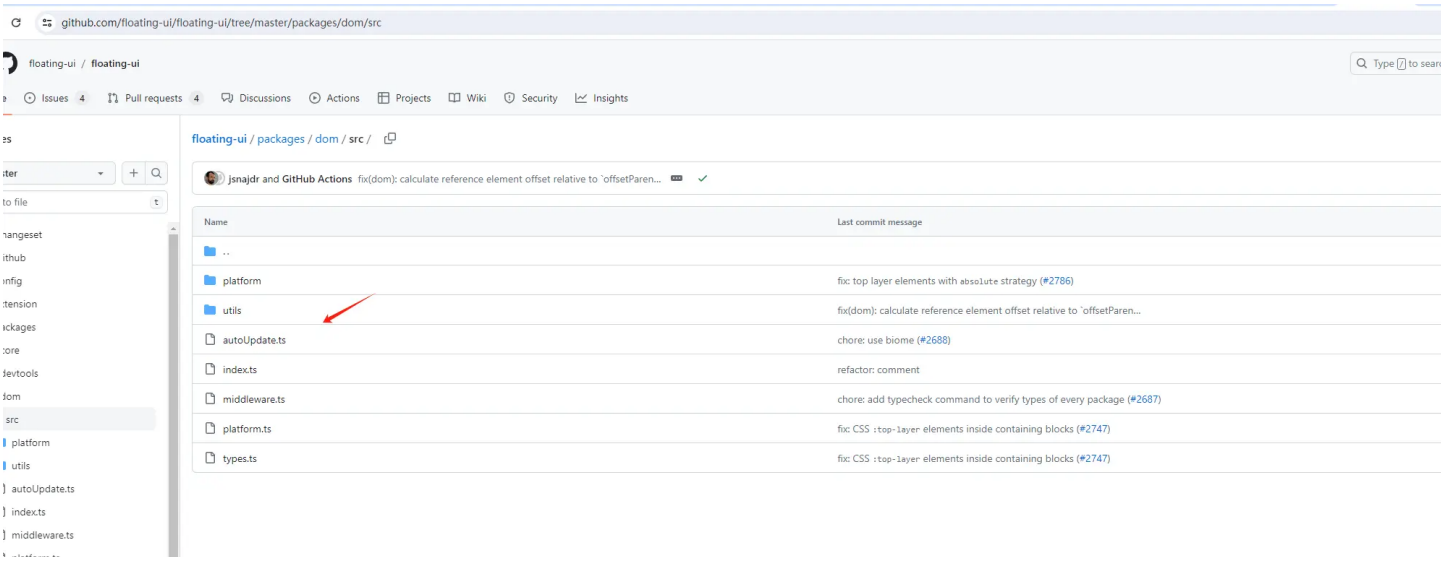
既然自己很难解决，那就看看别人是怎么解决这个问题的吧，我直接上 饿了么UI 上看看它的弹窗组件是怎么做的，于是我找到了 floating-ui 也就是原来的 `popper.js` 现在改名字了。



在文档中，我找到自动更新这块，也就是 `floating-ui` 通过监听器来实现自动更新弹窗位置。 到这里就可以看看 `floating-ui` 的源码了。

```
1 import {autoUpdate} from '@floating-ui/dom';
```

可以看到这个方法是放在 'floating-ui/dom'下面的



源码地址：github.com/floating-ui…

进入 `floating-ui` 的 github 地址，找到 `packages` 下 `dom` 下的 `src` 目录下，就可以看到想要的 `autoUpdate.ts` 自动更新的具体实现了。

4. 天才的想法

抛去其它不重要的东西，实现自动更新主要就是其中的 `refresh` 方法，先看一下代码

```
1  function refresh(skip = false, threshold = 1) {
2          // 清理操作，清理上一次定时器和监听
3          cleanup();
4
5          // 获取元素的位置和尺寸信息
6          const {
7                  left,
8                  top,
9                  width,
10                 height
11         } = element.getBoundingClientRect();
12
13         if (!skip) {
14           // 这里更新弹窗的位置
15           onMove();
16         }
17
18         // 如果元素的宽度或高度不存在，则直接返回
19         if (!width || !height) {
20                 return;
```

```
21          }
22
23          // 计算元素相对于视口四个方向的偏移量
24          const insetTop = Math.floor(top);
25          const insetRight = Math.floor(root.clientWidth - (left + width));
26          const insetBottom = Math.floor(root.clientHeight - (top + height));
27          const insetLeft = Math.floor(left);
28    // 这里就是元素的位置
29          const rootMargin = `${-insetTop}px ${-insetRight}px ${-insetBottom}px
    ${-insetLeft}px`;
30
31          // 定义 IntersectionObserver 的选项
32          const options = {
33                  rootMargin,
34                  threshold: Math.max(0, Math.min(1, threshold)) || 1,
35          };
36
37          let isFirstUpdate = true;
38
39          // 处理 IntersectionObserver 的观察结果
40          function handleObserve(entries) {
41                  // 这里事件会把元素和视口交叉的比例返回
42                  const ratio = entries[0].intersectionRatio;
43          // 判断新的视口比例和老的是否一致，如果一致说明没有变化
44                  if (ratio !== threshold) {
45                          if (!isFirstUpdate) {
46                                  return refresh();
47                          }
48
49                          if (!ratio) {
50                      // 即元素完全不可见时，也就是ratio = 0时，代码设置了一个定时器。
51                      // 这个定时器的作用是在短暂的延迟（100毫秒）后，再次调
    用 `refresh` 函数，
52                      // 这次传递一个非常小的阈值 `1e-7`。这样可以在元素完全不可见时，保
    证重新触发监听
53                                  timeoutId = setTimeout(() => {
54                                          refresh(false, 1e-7);
55                                  }, 100);
56                          } else {
57                                  refresh(false, ratio);
58                          }
59                  }
60
61                  isFirstUpdate = false;
62          }
63
64          // 创建 IntersectionObserver 对象并开始观察元素
```

```
        io = new IntersectionObserver(handleObserve, options);
        // 监听元素
        io.observe(element);
}

refresh(true);

```

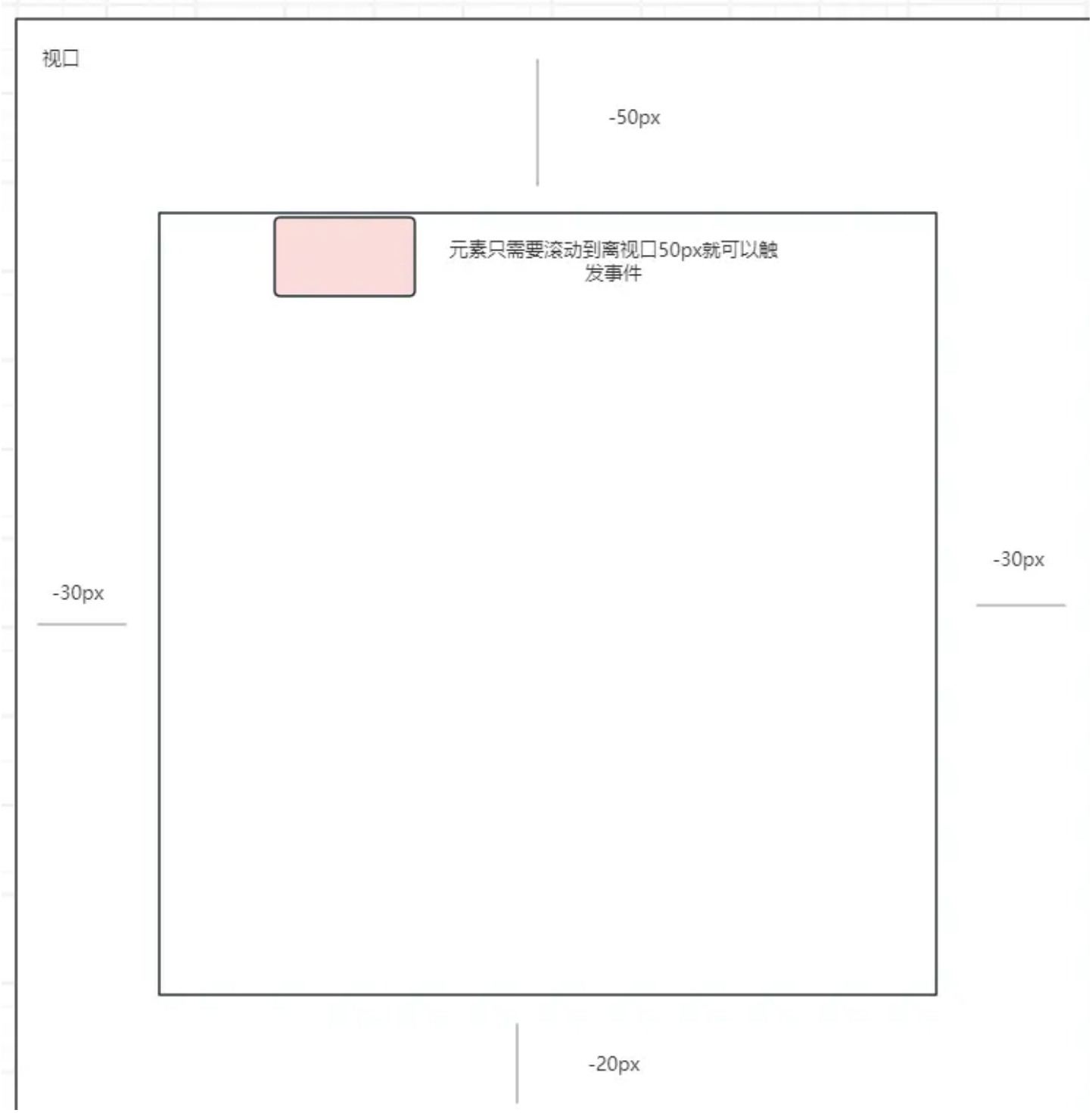
可以发现代码其实不复杂，主要实现还是依赖于 `IntersectionObserver` ，但是其中最重要的有几个点，我详细介绍一下

4.1 rootMargin

最重要的其实就是 `rootMargin`，`rootMargin` 到底是做啥用的呢?

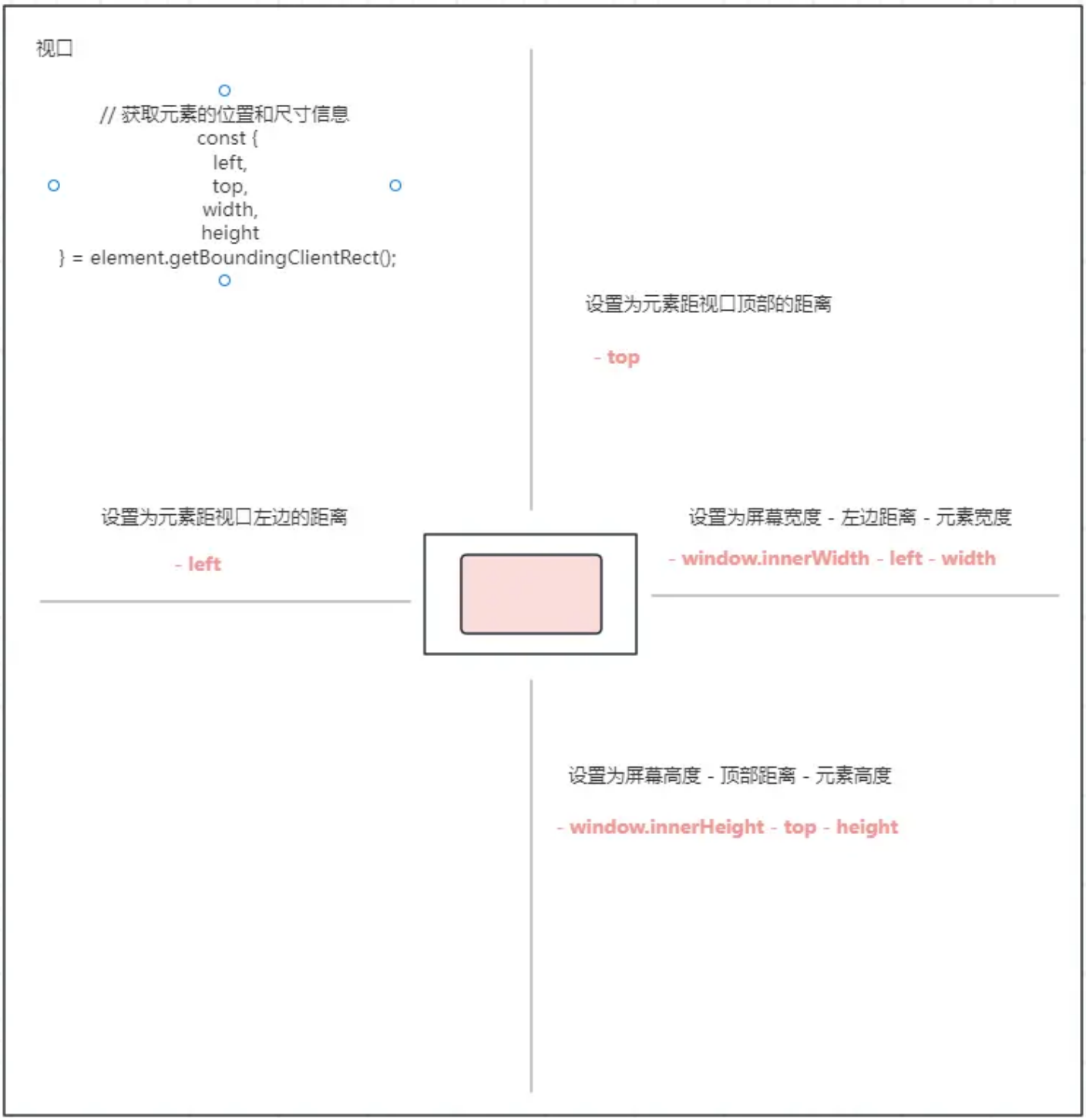
我上面说了 `IntersectionObserver` 是 **监测目标元素与视窗(viewport)的交叉状态**，而这个 `rootMargin` 就是可以将这个视窗缩小。

比如我设置 rootMargin 为 "-50px -30px -20px -30px"，注意这里 `rootMargin` 和 `margin` 类似，都是按照 **上 右 下 左** 来设置的



可以看到这样，当元素距离顶部 `50px` 就触发了事件。而不必等到元素完全滚动到视口。

既然这样，**当我设置 `rootMargin` 就是该元素本身的位置，不就可以实现只要元素一滚动，元素就与视口发生了交叉，触发事件**了吗?



4.2 循环监听事件

仅仅将视口缩小到该元素本身的位置还是不够，因为只要一滚动，元素的位置就发生了改变，即视口的位置也需要跟随着元素的位置变化进行变化

```
1 if (ratio !== threshold) {
2         if (!isFirstUpdate) {
3             return refresh();
4         }
5         if (!ratio) {
6             // 即元素完全不可见时，也就是ratio = 0时，代码设置了一个定时器。
7             // 这个定时器的作用是在短暂的延迟（100毫秒）后，再次调用 `refresh` 函数，
```

```
            // 这次传递一个非常小的阈值 `1e-7`。这样可以在元素在视口不可见时，保证可以重新触发监听
            timeoutId = setTimeout(() => {
                    refresh(false, 1e-7);
            }, 100);
        } else {
                refresh(false, ratio);
        }
}
```

也就是这里,可以看到每一次元素视口交叉的比例变化后，都重新调用了 `refresh` 方法，根据当前元素和屏幕的新的距离，创建一个新的监听器。

这样的话也就实现了类似 `scroll` 的效果，通过**不断变化的视口来确认元素的位置是否发生了变化**。

5. 结语

所以说有时候思路还是没有打开，刚看到这个实现思路确实惊到我了，没有想到借助 `rootMargin` 可以实现类似 `scroll` 监听的效果。