

22 真题手把手：事件的防抖与节流

更新时间：2020-05-12 14:12:29



“上天赋予的生命，就是要为人类的繁荣和平和幸福而奉献。——松下幸之助”

在各种各样的浏览器事件中，有一类特别需要大家关注的事件：那些容易过度触发的事件。

比如`scroll` 事件，它就是一个非常容易被反复触发的事件。其实不止 `scroll` 事件，`resize` 事件、鼠标事件（比如`mousemove`、`mouseover` 等）、键盘事件（`keyup`、`keydown` 等）都存在被频繁触发的风险。

频繁触发回调导致的大量计算会引发页面的抖动甚至卡顿。为了规避这种情况，我们需要一些手段来控制事件被触发的频率。就是在这样的背景下，`throttle`（事件节流）和 `debounce`（事件防抖）出现了。

事件节流和事件防抖的考察频率，随着前端性能近年来愈发受到重视而与日俱增。此外，这两个东西还都以闭包的形式存在：它们通过对事件对应的回调函数进行包裹、以自由变量的形式缓存时间信息，最后用 `setTimeout` 来控制事件的触发频率。

通过一段代码，至少能考察你两个重点知识的掌握程度——这是一道任何层次的面试官都喜欢得不行的面试题。

Throttle：第一个人说了算

`throttle` 的中心思想在于：在某段时间内，不管你触发了多少次回调，我都只认第一次，并在计时结束时给予响应。

先给大家讲个小故事：现在有一个旅客刚下了飞机，需要用车，于是打电话叫了该机场唯一的一辆机场大巴来接。司机开到机场，心想来都来了，多接几个人一起走吧，这样这趟才跑得值——我等个十分钟看看。于是司机一边打开了计时器，一边招呼后面的客人陆陆续续上车。在这十分钟内，后面下飞机的乘客都只能乘这一辆大巴，十分钟过去后，不管后面还有多少没挤上车的乘客，这班车都必须发走。

在这个故事里，“司机”就是我们的节流阀，他控制发车的时机；“乘客”就是因为我们频繁操作事件而不断涌入的回调任务，它需要接受“司机”的安排；而“计时器”，就是我们上文提到的以自由变量形式存在的时间信息，它是“司机”决定发车的依据；最后“发车”这个动作，就对应到回调函数的执行。

总结下来，所谓的“节流”，是通过在一段时间内无视后来产生的回调请求来实现的。只要一位客人叫了车，司机就会为他开启计时器，一定的时间内，后面需要乘车的客人都得排队上这一辆车，谁也无法叫到更多的车。

对应到实际的交互上是一样一样的：每当用户触发了一次 `scroll` 事件，我们就为这个触发操作开启计时器。一段时间内，后续所有的 `scroll` 事件都会被当作“一辆车的乘客”——它们无法触发新的 `scroll` 回调。直到“一段时间”到了，第一次触发的 `scroll` 事件对应的回调才会执行，而“一段时间内”触发的后续的 `scroll` 回调都会被节流阀无视掉。

理解了大致的思路，我们现在一起实现一个 `throttle`：

```
// fn是我们需要包装的事件回调, interval是时间间隔的阈值
function throttle(fn, interval) {
  // last为上一次触发回调的时间
  let last = 0

  // 将throttle处理结果当作函数返回
  return function () {
    // 保留调用时的this上下文
    let context = this
    // 保留调用时传入的参数
    let args = arguments
    // 记录本次触发回调的时间
    let now = +new Date()

    // 判断上次触发的时间和本次触发的时间差是否小于时间间隔的阈值
    if (now - last >= interval) {
      // 如果时间间隔大于我们设定的时间间隔阈值，则执行回调
      last = now;
      fn.apply(context, args);
    }
  }
}

// 用throttle来包装scroll的回调
const better_scroll = throttle(() => console.log('触发了滚动事件'), 1000)
document.addEventListener('scroll', better_scroll)
```

Debounce：最后一个人说了算

防抖的中心思想在于：我会等你到底。在某段时间内，不管你触发了多少次回调，我都只认最后一次。

继续讲司机开车的故事。这次的司机比较有耐心。第一个乘客上车后，司机开始计时（比如说十分钟）。十分钟之内，如果又上来了一个乘客，司机会把计时器清零，重新开始等另一个十分钟（延迟了等待）。直到有这么一位乘客，从他上车开始，后续十分钟都没有新乘客上车，司机会认为确实没有人需要搭这趟车了，才会把车开走。

我们对比 `throttle` 来理解 `debounce`：在`throttle`的逻辑里，“第一个人说了算”，它只为第一个乘客计时，时间到了就执行回调。而 `debounce` 认为，“最后一个人说了算”，`debounce` 会为每一个新乘客设定新的定时器。

我们基于上面的理解，一起来写一个 `debounce`:

```
// fn是我们需要包装的事件回调, delay是每次推迟执行的等待时间
function debounce(fn, delay) {
  // 定时器
  let timer = null

  // 将debounce处理结果当作函数返回
  return function () {
    // 保留调用时的this上下文
    let context = this
    // 保留调用时传入的参数
    let args = arguments
    // 每次事件被触发时，都去清除之前的旧定时器
    if(timer) {
      clearTimeout(timer)
    }
    // 设立新定时器
    timer = setTimeout(function () {
      fn.apply(context, args)
    }, delay)
  }
}

// 用debounce来包装scroll的回调
const better_scroll = debounce(() => console.log('触发了滚动事件'), 1000)
document.addEventListener('scroll', better_scroll)
```

小结

`throttle` 和 `debounce` 不仅是我们日常开发中的常用优质代码片段，更是前端面试中不可不知的高频考点。“看懂了代码”、“理解了过程”在本节都是不够的，重要的是把它写到自己的项目里去，亲自体验一把节流和防抖带来的性能提升。

}