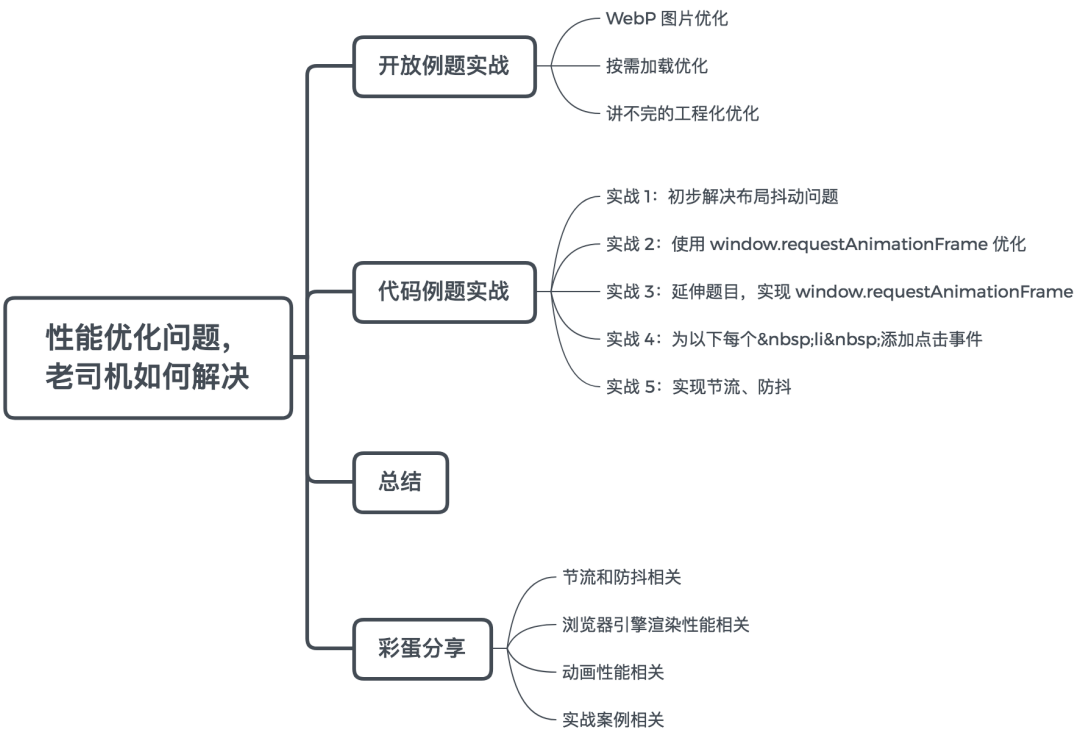
性能优化问题，老司机如何解决（下）

上一节课，我们从宏观上讲述了性能优化的概念。这一节，我们直接来「手写代码」。

在此之前，我们先回顾一下「性能优化」主题的知识点：



代码例题实战

「白板写代码」是考察候选人基础能力、思维能力的有效手段。这一部分，我们列举几个性能相关的代码片段，供读者体会。

实战 1：初步解决布局抖动问题

请候选人对以下代码进行优化：

```
var h1 = element1.clientHeight
element1.style.height = (h1 * 2) + 'px'

var h2 = element2.clientHeight
element2.style.height = (h2 * 2) + 'px'

var h3 = element3.clientHeight
element3.style.height = (h3 * 2) + 'px'
```

这是一道较为基础的题目，上面的代码，会造成典型的布局抖动问题。

布局抖动是指 DOM 元素被 JavaScript 多次反复读写，导致文档多次无意义重排。我们知道浏览器很「懒」，它会收集（batch）当前操作，统一进行重排。可是，如果在当前操作完成前，从 DOM 元素中获取值，这会迫使浏览器提早执行布局操作，这称为**强制同步布局**。这样的副作用对于低配置的移动设备来说，后果是不堪设想的。

我们对 element1 进行读、写操作之后，又企图去获取 element2 的值，浏览器为了获取正确的值，只能进行重排。优化思路为：

```
// 读
var h1 = element1.clientHeight
var h2 = element2.clientHeight
var h3 = element3.clientHeight

// 写（无效布局）
element1.style.height = (h1 * 2) + 'px'
element2.style.height = (h2 * 2) + 'px'
element3.style.height = (h3 * 2) + 'px'
```

实战 2：使用 window.requestAnimationFrame 对上述代码优化

如果读者对 window.requestAnimationFrame 不熟悉的话，我们先来看一下 MDN 上的说明：

该方法告诉浏览器你希望执行的操作，并请求浏览器在下一次重绘之前调用指定的函数来更新。

语法：

```
window.requestAnimationFrame(callback)
```

也就是说，当你需要更新屏幕画面时就可以调用此方法。在浏览器下次重绘前统一执行回调函数，优化方案：

```
// 读
var h1 = element1.clientHeight
// 写
requestAnimationFrame(() => {
   element1.style.height = (h1 * 2) + 'px'
})

// 读
var h2 = element2.clientHeight
// 写
requestAnimationFrame(() => {
   element2.style.height = (h2 * 2) + 'px'
})

// 读
var h3 = element3.clientHeight
// 写
requestAnimationFrame(() => {
   element3.style.height = (h3 * 2) + 'px'
})
```

我们将代码中所有 DOM 的写操作在下一帧一起执行，保留所有 DOM 的读操作在当前同步状态。这样有效减少了无意义的重排，显然效率更高。

实战 3：延伸题目，实现 window.requestAnimationFrame 的 polyfill

polyfill 就是我们常说的垫片，此处指在浏览器兼容性不支持的情况下，备选实现方案。

window.requestAnimationFrame 在一些老版本浏览器中无法兼容，为了让代码在老机器也能运行不报错，请用代码实现：

```
if (!window.requestAnimationFrame)
window.requestAnimationFrame = (callback, element) => {
   const id = window.setTimeout(() => {
       callback()
   }, 1000 / 60)
   return id
}
if (!window.cancelAnimationFrame)
window.cancelAnimationFrame = id => {
   clearTimeout(id)
}
```

上面的代码按照 1 秒钟 60 次（大约每 16.7 毫秒一次），并使用 window.setTimeout 来进行模拟。这是一种粗略的实现，并没有考虑统一浏览器前缀和 callback 参数等问题。一般需求中，实现上面的答案已经可以符合要求了。

实战 4：为以下每个 li 添加点击事件

```
1
```

```
2
```

3

4

5

6

7

8

9

10

这道题目非常基础，但是实现方式上需要注意是否使用了**事件委托**。如果候选人直接对 li 进行绑定处理，那么很容易给面试官留下「平时代码习惯不好」的印象，造成潜在性能负担。更好的做法显然是：

```
window.onload = () => {
   const ul = document.getElementsByTagName('ul')[0]
   const liList = document.getElementsByTagName('li')

   ul.onclick = e => {
       const normalizeE = e || window.event
       const target = normalizeE.target ||
normalizeE.srcElement

       if (target.nodeName.toLowerCase() == "li") {
           alert(target.innerHTML)
       }
   }
}
```

一般情况下，作为面试官，我不会提示候选人采用事件委托的写法，而是观察候选人的第一反应，对其代码习惯进行考察。如果候选人没有采用事件委托的写法，才会进一步追问。

实战 5：实现节流、防抖

我们知道，鼠标滚动（scroll）、调整窗口大小（resize）、敲击键盘（keyup）这类事件在触发时往往频率极高。这时候事件对应的回调函数也会在极短时间内反复执行。想象一下，如果这些回调函数内的逻辑涉及复杂的计算，或者对 DOM 操作非常频繁，从而造成大量布局操作、绘制操作，那么就存在阻塞主线程的危险，直接后果就是掉帧，用户能够感受到明显的卡顿。

有经验的程序员为了规避这样的问题，往往会使用节流（throttle）或者防抖（debounce）来进行处理。因此节流和防抖已经成为非常常见的优化手段，现如今也是面试的必考题型之一。

节流和防抖总是一起出现，那么它们有什么不同呢？

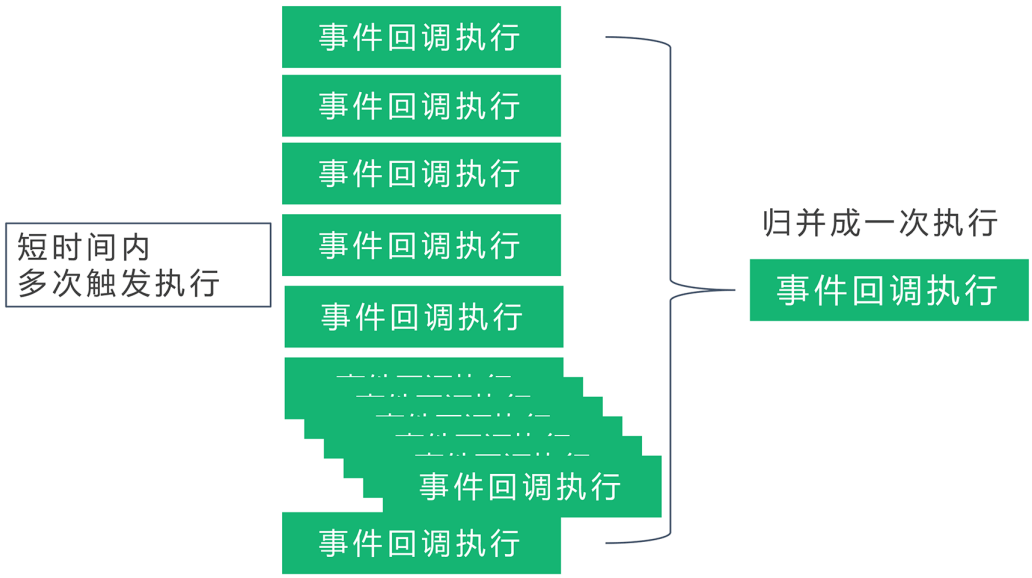
回答这个问题，我们首先要知道它们解决的问题相同，方向类似：**两者并不会减少事件的触发，而是减少事件触发时回调函数的执行次数**。为了达成这个目的，节流和防抖采用的手段有所差别。

> 防抖：抖动现象本质就是指短时间内高频次触发。因此，我们可以把短时间内的多个连续调用合并成一次，也就是只触发一次回调函数。

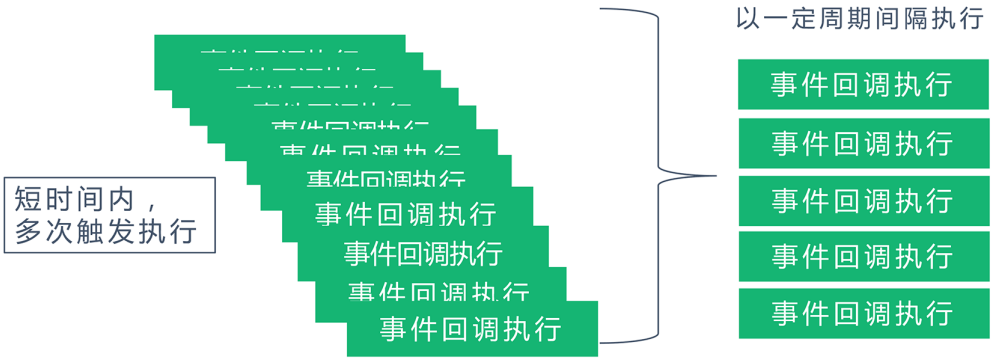
> 节流：顾名思义，就是将短时间的函数调用以一个固定的频率间隔执行，这就如同水龙头开关限制出水口流量。

这个例子可以很形象地展示节流与防抖的区别。

另外，请参考防抖图示：



节流图示：



了解了原理，我们先来实现事件防抖：

```
// 简单的防抖动函数
const debounce = (func, wait, immediate) => {
    let timeout
    return function () {
        const context = this
        const args = arguments

        const callNow = immediate & !timeout

        timeout && clearTimeout(timeout)

        timeout = setTimeout(function() {
            timeout = null
            if (!immediate) func.apply(context, args)
        }, wait)

        if (callNow) func.apply(context, args)
    }
}

// 采用了防抖动
window.addEventListener('scroll', debounce(() => {
    console.log('scroll')
}, 500))
```

```
// 没采用防抖动
window.addEventListener('scroll', () => {
   console.log('scroll')
})
```

如代码所示，我们使用 setTimeout 在 500ms 后执行事件回调，如果在这 500ms 内又有相关事件触发，则通过 clearTimeout(timeout) 取消上一次设置的回调。因此在 500ms 内没有连续触发多次 scroll 事件，才会真正触发 scroll 回调函数——或者说，500ms 内的多次调用被归并成了一次，在最后一次「抖动」后，进行回调执行。同时，我们设置了 immediate 参数，用以立即执行。关于 func.apply 的用法，学习过《第 1-1 课：一网打尽 this，对执行上下文说 Yes》的读者应该不会陌生。

关于事件节流：

```
const throttle = (func, wait) => {
   let startTime = 0
   return function() {
       let handleTime = +new Date()
       let context = this
       const args = arguments

       if (handleTime - startTime >= wait) {
           func.apply(context, args)
           startTime = handleTime
       }
   }
}

window.addEventListener('scroll', throttle(() => {
   console.log('scroll')
}, 500))
```

当然，我们同样可以用 setTimeout 来实现：

```
const throttle = (func, wait) => {
    let timeout

    return function () {
        const context = this
        const args = arguments
        if (!timeout) {
            timeout = setTimeout(function() {
                func.apply(context, args)
                    timeout = null
          }, wait)
        }
    }
}
```

与防抖相比，少了 clearTimeout 的操作，请读者细心对比。

要准确理解节流和防抖，需要多动手实践。这里也建议大家有时间研究研究 lodash 库关于节流和防抖的实现。事实上，这个话题还可以玩出很多花来，比如如何暴露给开发者 cancelDebounce，又如上述 throttle 的两种方式各有哪些瑕疵，针对这些瑕疵，是否可以结合两种实现优化？感兴趣的读者请在评论区留言探讨，或者在文末彩蛋部分找到相关内容。

总结

性能优化，实在是一个极大的话题，需要我们在平时工作学习中不断积累。对于准备面试的朋友，在面试前，除了时刻注意代码习惯、掌握常见考点以外，还要整理、回顾、复盘平时的性能相关项目。

这一节课难以覆盖性能优化的方方面面，本课程的其他章节，还会有这个话题的相关渗透，如网络协议、缓存策略、数据结构和算法等，这些内容和性能息息相关。请大家持续关注学习，同时欢迎在评论区和其他小伙伴讨论以及向我提问。

课程代码仓库： https://github.com/HOUCe/lucas-gitchat-courses

彩蛋分享

节流和防抖相关

Debouncing and Throttling Explained Through Examples

谈谈 JS 中的函数节流

JavaScript 函数节流和函数防抖之间的区别

高性能滚动 scroll 及页面渲染优化&version=12020110&nettype=WIFI&fontScale=100&pass_ticket=OxCcOonsw3hgntyvXy%2FSYPn%2Fw9jx2Hv%2FheV8seAGt987cQT%2FygphdRBJ0UyMTQvc)

从 lodash 源码学习节流与防抖

理解并优化函数节流 Throttle

浏览器引擎渲染性能相关

Inside look at modern web browser

How Browsers Work: Behind the scenes of modern web browsers

How browsers work

How browser rendering works—behind the scenes

What Every Frontend Developer Should Know About Webpage Rendering

前端文摘：深入解析浏览器的幕后工作原理

从 Chrome 源码看浏览器如何加载资源

浏览器内核渲染：重建引擎

体现工匠精神的 Resource Hints

浏览器页面渲染机制，你真的弄懂了吗

前端不止：Web 性能优化 – 关键渲染路径以及优化策略

浏览器前端优化

浅析前端页面渲染机制

浅析渲染引擎与前端优化

渲染性能

Repaint 、Reflow 的基本认识和优化 (2)

动画性能相关

Timing control for script-based animations)

Gain Motion Superpowers with requestAnimationFrame

CSS Animation 性能优化

GSAP 的动画快于 jQuery 吗？为何？

Javascript 高性能动画与页面渲染

也许你不知道，JS animation 比 CSS 更快！

渐进式动画解决方案

你应该知道的 requestIdleCallback

无线性能优化：Composite

优化动画卡顿：卡顿原因分析及优化方案

一篇文章说清浏览器解析和 CSS（GPU）动画优化

实战案例相关

Building the Google Photos Web UI

A Netflix Web Performance Case Study

The Cost Of JavaScript In 2018

How we reduced our initial JS/CSS size by 67%

Front-End Performance Checklist 2019

网站性能优化实战——从 12.67s 到 1.06s 的故事

前端黑科技：美团网页首帧优化实践

Web 字体图标-自动化方案

JS 加载慢？谷歌大神带你飞!

前端性能优化（三） 移动端浏览器前端优化策略

CSS @font-face 性能优化

移动 Web 性能优化从入门到进阶

记一次惊心动魄的前端性能优化之旅

点击查看下一节

以 React 为例，说说框架和性能（1）