45. 手写 debounce + throttle 的组合封装

1. 核心概念 (Core Concept)

`debounce` (防抖) 和 `throttle` (节流) 是前端性能优化中常用的两种技术，用于限制函数在特定时间段内的执行频率。组合封装是指创建一个函数，该函数能够根据配置参数，灵活地实现防抖或节流的行为。

2. 为什么需要它？ (The "Why")

- **性能优化:** 显著减少高频触发事件（如窗口 resize, 滚动 scroll, 输入框 input, 鼠标移动 mousemove 等）的回调函数执行次数，降低 CPU 负载。
- **提升用户体验:** 避免因频繁执行复杂计算或 DOM 操作导致的界面卡顿或无响应。
- **资源控制:** 合理控制网络请求的发送频率，避免不必要的后端压力。

3. API 与用法 (API & Usage)

组合封装通常会设计一个高阶函数，该函数接收原始函数、等待时间以及配置选项作为参数，并返回一个经过处理的函数。配置选项通常用于指定是使用防抖还是节流模式。

由于标准 JavaScript 或 W3C/ECMA 规范中没有内置 `debounce` 和 `throttle` 函数，它们的实现是基于 JavaScript 的定时器 (`setTimeout` , `clearTimeout`) 构建的。这里提供一个经典的组合封装实现示例，它结合了防抖和节流的核心逻辑。

```
/**
 * 手写通用函数，实现 debounce 或 throttle 功能
 * @param {Function} func 需要被防抖或节流的函数
 * @param {number} wait 等待时间 (ms)
 * @param {object} options 配置选项
 * @param {boolean} [options.isThrottle=false] 是否使用节流模式 (默认为防抖)
 * @param {boolean} [options.leading=false] 在 debounce/throttle 开始时是否
执行一次
 * @param {boolean} [options.trailing=true] 在 debounce/throttle 结束后是否
执行一次 (对于 throttle，表示是否在等待时间结束后执行最后一次调用)
 * @returns {Function} 返回经过处理的新函数
 */
function debounceOrThrottle(func, wait, options = {}) {
    let timeoutId;
    let lastArgs;
    let lastThis;
    let lastCallTime = 0; // 用于节流
    let result;

    const { isThrottle = false, leading = false, trailing = true } =
options;
```

```
    const invokeFunc = function(time) {
        const args = lastArgs;
        const thisArg = lastThis;
        lastArgs = lastThis = undefined; // 清空引用，便于垃圾回收
        lastCallTime = time; // 记录执行时间，用于节流
        result = func.apply(thisArg, args);
        return result;
    };

    const leadingEdge = function(time) {
        lastCallTime = time; // 记录首次触发时间
        timeoutId = setTimeout(timerExpired, wait);
        if (leading) {
             // 节流模式下的 leading 执行，或防抖模式下的 leading 执行
            return invokeFunc(time);
        }
         // 防抖模式下 leading=false 时，不立即执行，仅设置定时器
        return result; // 返回上次结果
    };

    const remainingWait = function(time) {
         // 计算距离下次允许执行的时间（仅用于节流）
        const timeSinceLastCall = time - lastCallTime;
        const timeWaiting = wait - timeSinceLastCall;
        // 如果 timeWaiting <= 0 或 timeWaiting > wait，说明可以立即执行或已经
等待超出了一个周期
        return Math.max(0, timeWaiting);
    };

    const timerExpired = function() {
        const time = Date.now();

        if (isThrottle) {
             // 节流定时器触发
            // 如果距离上次调用已经达到或超过 wait 间隔，且存在待处理的调用，则执行
            if (trailing && lastArgs) {
                return trailingEdge(time);
            }
            timeoutId = undefined; // 没有待处理调用，清除定时器
             // 如果 leading 为 true 且没有 trailing，但定时器触发了（这种情况通
常不会发生，因为 leading 模式下应该立即执行），则也清除定时器
            if (!leading && !trailing) {
                 timeoutId = undefined;
            }

        } else { // Debounce 定时器触发
             // 如果是 trailing 模式且存在待处理的调用，则执行
            if (trailing && lastArgs) {
                 result = invokeFunc(time);
```

```
            }
            timeoutId = undefined; // 清除定时器
            lastArgs = lastThis = undefined; // 确保清空

             // 如果是 leading 模式且没有 trailing，但定时器触发了，则也清除定时器
            if (!leading && !trailing) {
                timeoutId = undefined;
            }

        }
         // 清除定时器后，如果还有其他定时器（节流 trailing 模式可能在
trailingEdge 中再次设置），此处不做额外处理
    };

    const trailingEdge = function(time) {
        timeoutId = undefined; // 清除当前定时器
        const args = lastArgs;
        const thisArg = lastThis;
        lastArgs = lastThis = undefined; // 清空引用

        // 执行 trailing 调用
        const result = invokeFunc(time);

        // 节流模式下的 trailing 可能需要重新设置定时器以处理后续的触发
        if (isThrottle && lastArgs) { // 如果在执行 trailing 期间又有新的触发
             timeoutId = setTimeout(timerExpired, wait);
        }
        return result;
    };


    const debouncedOrThrottled = function( ...args) {
        const time = Date.now();
        lastArgs = args;
        lastThis = this;

        // === 核心逻辑区分 debounce 和 throttle ===
        if (isThrottle) {
            // --- Throttle 逻辑 ---
            const canCall = time - lastCallTime >= wait;

            if (canCall) {
                // 如果可以调用（距离上次执行已经超过 wait）
                if (timeoutId) { // 清除可能存在的 trailing 定时器
                    clearTimeout(timeoutId);
                    timeoutId = undefined;
                }
                return invokeFunc(time); // 执行函数
            } else if (trailing && !timeoutId) {
                 // 如果不能立即调用，但需要 trailing，并且还没有设置 trailing 定
```

```
时器
                timeoutId = setTimeout(timerExpired, remainingWait(time));
// 设置等待剩余时间的定时器
            }
             // 如果不能立即调用且不需要 trailing，或者 trailing 定时器已存在，则
不执行

        } else {
            // --- Debounce 逻辑 ---
            const isLeading = !timeoutId; // 是否是第一次触发（在 wait 时间内）

            clearTimeout(timeoutId); // 清除之前的定时器

            timeoutId = setTimeout(timerExpired, wait); // 设置新的定时器

            if (isLeading && leading) {
                // 如果是 leading 模式且是第一次触发，则立即执行
                return invokeFunc(time);
            }
             // 如果是 trailing 模式，或 leading=false，则等待定时器触发再执行
        }
        return result; // 返回上次的结果（对于 debounce 的 trailing 或
throttle 的中间调用）
    };

    // 添加 cancel 方法，用于取消等待中的函数执行
    debouncedOrThrottled.cancel = () => {
        clearTimeout(timeoutId);
        timeoutId = undefined;
        lastArgs = lastThis = undefined;
        lastCallTime = 0; // 重置节流状态
    };

    return debouncedOrThrottled;
}

// --- 使用示例 ---

// 例子 1: 防抖（默认 trailing=true）
const debouncedSearch = debounceOrThrottle((query) => {
    console.log('Searching for:', query);
}, 500);

document.getElementById('searchInput').addEventListener('input', (e) => {
    debouncedSearch(e.target.value);
});

// 例子 2: 节流（默认 trailing=true）
const throttledScroll = debounceOrThrottle(() => {
    console.log('Scrolling!');
```

```
}, 300, { isThrottle: true });

window.addEventListener('scroll', throttledScroll);


// 例子 3: 防抖 leading=true
const debouncedClickLeading = debounceOrThrottle(() => {
    console.log('Button clicked (debounce leading)');
}, 1000, { leading: true, trailing: false }); // leading true 常用场景是不希望最后一击无效

document.getElementById('button').addEventListener('click', debouncedClickLeading);


// 例子 4: 节流 leading=true, trailing=false
const throttledMoveLeading = debounceOrThrottle((x, y) => {
    console.log('Mouse moved (throttle leading):', x, y);
}, 100, { isThrottle: true, leading: true, trailing: false }); // 常用场景是进入区域即执行一次，后续按频率执行但不执行最后一次

document.getElementById('area').addEventListener('mousemove', (e) => {
    throttledMoveLeading(e.clientX, e.clientY);
});
```

解释:

- 该组合函数通过 `isThrottle` 参数来决定内部逻辑是执行防抖还是节流。
- `wait` 参数指定了等待时间。
- `options` 对象提供了更精细的控制，如 `leading` (是否在开始时执行) 和 `trailing` (是否在结束时执行)。
- 内部通过 `setTimeout` 和 `clearTimeout` 管理定时器。
- `lastArgs`, `lastThis` 用于保存最后一次调用的参数和上下文 (`this`)，确保函数执行时能拿到正确的参数和上下文。
- `lastCallTime` 用于节流模式下记录上次函数执行的时间，以便计算是否达到 `wait` 间隔。
- `cancel` 方法用于强制取消当前等待中的执行。

4. 关键注意事项 (Key Considerations)

- **`this` 和参数的传递:** 封装函数必须正确处理原始函数的 `this` 上下文和参数，确保在执行时能正确传递。通常使用 `func.apply(lastThis, lastArgs)` 来实现。
- **`leading` 和 `trailing` 选项:** 理解这两个选项对防抖和节流行为的影响。
    - **防抖:**

    - `leading: true, trailing: true`: 触发后立即执行一次，等待 `wait` 时间内再次触发则取消前一个定时器并重新设置，`wait` 时间结束后（如果期间没有再次触发）再执行一次（这是 Lodash 的默认行为）。
    - `leading: true, trailing: false`: 触发后立即执行一次，`wait` 时间内再次触发会忽略，直到 `wait` 时间结束后才能再次触发执行（不管期间触发多少次）。
    - `leading: false, trailing: true` (常见默认): 触发后不立即执行，等待 `wait` 时间，若期间再次触发则取消前一个并重新计时，直到 `wait` 时间结束后执行最后一次触发。
    - `leading: false, trailing: false`: 触发后不立即执行，等待 `wait` 时间，期间再次触发会取消前一个并重新计时，`wait` 时间结束后没有任何执行（除非再次触发）。
  - **节流:**
    - `leading: true, trailing: true`: 触发后立即执行一次，后续触发在 `wait` 时间内会被忽略，等到 `wait` 时间结束时如果期间有触发则再执行一次（通常是最后一次触发），然后重新开始计时。
    - `leading: true, trailing: false`: 触发后立即执行一次，后续触发在 `wait` 时间内会被忽略，直到 `wait` 时间结束后才能再次触发执行。
    - `leading: false, trailing: true`: 触发后不会立即执行，等待 `wait` 时间结束后执行第一次触发，后续触发在 `wait` 时间内会被忽略，等到下一个 `wait` 时间结束时执行期间的最后一次触发。
    - `leading: false, trailing: false`: 触发后不会立即执行，等待 `wait` 时间结束后执行第一次触发，后续触发在 `wait` 时间内会被忽略，直到下一个 `wait` 时间结束时执行期间的最后一次触发（但不会单独执行最后一次触发）。
- **清除定时器:** 在封装函数返回的函数上暴露 `cancel` 方法是一个好的实践，允许在组件卸载或其他需要中断的场景下清除定时器，避免内存泄漏或不必要的执行。
- **返回值和异常处理:** 考虑封装函数本身的返回值以及被封装函数可能抛出的异常。示例中返回的是 `invokeFunc` 的结果，实际开发中可能需要更复杂的处理。

5. 参考资料 (References)

- **Lodash - debounce:** https://lodash.com/docs#debounce (Lodash 提供了非常成熟和广泛使用的 debounce 实现，是重要的参考依据)
- **Lodash - throttle:** https://lodash.com/docs#throttle (Lodash 的 throttle 实现同样是业界标准)
- **MDN Web Docs - setTimeout:** https://developer.mozilla.org/zh-CN/docs/Web/API/setTimeout
- **MDN Web Docs - clearTimeout:** https://developer.mozilla.org/zh-CN/docs/Web/API/clearTimeout
- **A Guide to Debouncing and Throttling in JavaScript:** https://css-tricks.com/the-difference-between-throttling-and-debouncing/ (一篇经典的解释文章)