彻底弄懂函数防抖和函数节流

> 链接：https://segmentfault.com/a/1190000018445196

函数防抖和节流

函数防抖和函数节流：优化高频率执行js代码的一种手段，js中的一些事件如浏览器的resize、scroll，鼠标的mousemove、mouseover，input输入框的keypress等事件在触发时，会不断地调用绑定在事件上的回调函数，极大地浪费资源，降低前端性能。为了优化体验，需要对这类事件进行调用次数的限制。

函数防抖

> 在事件被触发n秒后再执行回调，如果在这n秒内又被触发，则重新计时。

根据函数防抖思路设计出第一版的最简单的防抖代码：

```
var timer; // 维护同一个timer
function debounce(fn, delay) {
    clearTimeout(timer);
    timer = setTimeout(function(){
        fn();
    }, delay);
}
```

用onmousemove测试一下防抖函数:

```
// test
function testDebounce() {
    console.log('test');
}
document.onmousemove = () => {
  debounce(testDebounce, 1000);
}
```

上面例子中的debounce就是防抖函数，在document中鼠标移动的时候，会在onmousemove最后触发的1s后执行回调函数testDebounce；如果我们一直在浏览器中移动鼠标（比如10s），会发现会在10 + 1s后才会执行testDebounce函数（因为clearTimeout(timer)），这个就是函数防抖。

在上面的代码中，会出现一个问题，var timer只能在setTimeout的父级作用域中，这样才是同一个timer，并且为了方便防抖函数的调用和回调函数fn的传参问题，我们应该用闭包来解决这些问题。

优化后的代码：

```
function debounce(fn, delay) {
    var timer; // 维护一个 timer
    return function () {
        var _this = this; // 取debounce执行作用域的this
        var args = arguments;
        if (timer) {
            clearTimeout(timer);
        }
        timer = setTimeout(function () {
            fn.apply(_this, args); // 用apply指向调用debounce的对象，相当于_this.fn(args);
        }, delay);
    };
}
```

测试用例：

```
// test
function testDebounce(e, content) {
    console.log(e, content);
}
var testDebounceFn = debounce(testDebounce, 1000); // 防抖函数
document.onmousemove = function (e) {
    testDebounceFn(e, 'debounce'); // 给防抖函数传参
}
```

使用闭包后，解决传参和封装防抖函数的问题，这样就可以在其他地方随便将需要防抖的函数传入debounce了。

函数节流

> 每隔一段时间，只执行一次函数。

- 定时器实现节流函数：

请仔细看清和防抖函数的代码差异

```
function throttle(fn, delay) {
    var timer;
    return function () {
        var _this = this;
        var args = arguments;
        if (timer) {
            return;
        }
        timer = setTimeout(function () {
            fn.apply(_this, args);
            timer = null; // 在delay后执行完fn之后清空timer，此时timer为假，throttle触发可以进入计时器
        }, delay)
    }
}
```

测试用例：

```
function testThrottle(e, content) {
    console.log(e, content);
}
var testThrottleFn = throttle(testThrottle, 1000); // 节流函数
document.onmousemove = function (e) {
    testThrottleFn(e, 'throttle'); // 给节流函数传参
}
```

上面例子中，如果我们一直在浏览器中移动鼠标（比如10s），则在这10s内会每隔1s执行一次testThrottle，这就是函数节流。

函数节流的目的，是为了限制函数一段时间内只能执行一次。因此，定时器实现节流函数通过使用定时任务，延时方法执行。在延时的时间内，方法若被触发，则直接退出方法。从而，实现函数一段时间内只执行一次。

根据函数节流的原理，我们也可以不依赖 setTimeout实现函数节流。

- 时间戳实现节流函数：

```
function throttle(fn, delay) {
    var previous = 0;
    // 使用闭包返回一个函数并且用到闭包函数外面的变量previous
    return function() {
        var _this = this;
        var args = arguments;
        var now = new Date();
        if(now - previous > delay) {
            fn.apply(_this, args);
            previous = now;
        }
    }
}

// test
function testThrottle(e, content) {
    console.log(e, content);
}
var testThrottleFn = throttle(testThrottle, 1000); // 节流函数
document.onmousemove = function (e) {
    testThrottleFn(e, 'throttle'); // 给节流函数传参
}
```

其实现原理，通过比对上一次执行时间与本次执行时间的时间差与间隔时间的大小关系，来判断是否执行函数。若时间差大于间隔时间，则立刻执行一次函数。并更新上一次执行时间。

异同比较

相同点：

- 都可以通过使用 setTimeout 实现。
- 目的都是，降低回调执行频率。节省计算资源。

不同点：

- 函数防抖，在一段连续操作结束后，处理回调，**利用clearTimeout 和 setTimeout实现。**函数节流，在一段连续操作中，**每一段时间只执行一次**，频率较高的事件中使用来提高性能。
- 函数防抖关注一定时间连续触发的事件只在最后执行一次，而函数节流侧重于一段时间内只执行一次。

常见应用场景

函数防抖的应用场景

连续的事件，只需触发一次回调的场景有：

- 搜索框搜索输入。只需用户最后一次输入完，再发送请求
- 手机号、邮箱验证输入检测
- 窗口大小Resize。只需窗口调整完成后，计算窗口大小。防止重复渲染。

函数节流的应用场景

间隔一段时间执行一次回调的场景有：

- 滚动加载，加载更多或滚到底部监听
- 谷歌搜索框，搜索联想功能
- 高频点击提交，表单重复提交

文档中出现的源代码都在这里: 防抖、节流

参考资料：

浅析函数防抖与函数节流

JavaScript专题系列-防抖和节流

7分钟理解JS的节流、防抖及使用场景

防抖、节流

可能这些参考资料中有某些错误，但是表示感谢，博客中有些内容用了里面的资料。