[前端拓展 1](#_Toc499121954)

[1、 JS功能拓展 1](#_Toc499121955)

[1.1、函数节流（throttle）与函数去抖（debounce） 1](#_Toc499121956)

[1.1.1、功能说明 1](#_Toc499121957)

[1.1.1.1、 debounce 1](#_Toc499121958)

[1.1.1.2、 throttle 2](#_Toc499121959)

[2、页面性能拓展 2](#_Toc499121960)

# 前端拓展

## JS功能拓展

### 1.1、函数节流（throttle）与函数去抖（debounce）

#### 1.1.1、功能说明

1. window对象的resize、scroll事件
2. 拖拽时的mousemove事件
3. 射击游戏中的mousedown、keydown事件
4. 文字输入、自动完成的keyup事件

当以上事件被频繁触发时，会频繁执行DOM操作、资源加载等重行为，导致UI停顿甚至浏览器崩溃。

实际上对于window的resize事件，实际需求大多为停止改变大小n毫秒后执行后续处理；而其他事件大多的需求是以一定的频率执行后续处理。针对这两种需求就出现了debounce和throttle两种解决办法。

##### debounce

**定义**：

如果用手指一直按住一个弹簧，它将不会弹起直到你松手为止。也就是说当调用动作n毫秒后，才会执行该动作，若在这n毫秒内又调用此动作则将重新计算执行时间。

**接口定义：**

/\*\*

\* 空闲控制 返回函数连续调用时，空闲时间必须大于或等于 idle，action 才会执行

\* @param idle {number} 空闲时间，单位毫秒

\* @param action {function} 请求关联函数，实际应用需要调用的函数

\* @return {function} 返回客户调用函数

\*/

var debounce = function(idle, action){

var last;

return function(){

var ctx = this, args = arguments;

clearTimeout(last);

last = setTimeout(function(){

action.apply(ctx,args);

}, idle)

}

}

##### **throttle**

**定义**：

如果将水龙头拧紧直到水是以水滴的形式流出，那你会发现每隔一段时间，就会有一滴水流出。也就是说预先设定一个执行周期，当调用动作的时刻大于等于执行周期则执行该动作，然后进入下一个新周期。

**接口定义：**

/\*\*

\* 频率控制 返回函数连续调用时，action 执行频率限定为 次 / delay

\* @param delay {number} 延迟时间，单位毫秒

\* @param action {function} 请求关联函数，实际应用需要调用的函数

\* @return {function} 返回客户调用函数

\*/

function throttle(func, wait, options) {

// 上下文，函数参数，函数返回值

var context, args, result;

// 延时器

var timeout = null;

// 上次执行时间点

var previous = 0;

if (!options) options = {};

// 延迟执行函数

var later = function () {

// 如果及时调用被关闭，则设置previous为0

previous = options.leading === false ? 0 : new Date().getTime();

timeout = null;

result = func.apply(context, args);

if (!timeout) context = args = null;

};

return function () {

var now = new Date().getTime();

// 首次执行时，如果设定了开始边界不执行选项，将上次执行时间设定为当前时间。

if (!previous && options.leading === false) previous = now;

// remaining容易理解，表示还剩多少时间可以再次执行func

var remaining = wait - (now - previous);

context = this;

args = arguments;

// 延迟时间间隔remaining小于等于0，表示上次执行至此所间隔时间已经超过一个时间窗口（设置的延迟时间）

// remaining大于时间窗口wait，说明now - previous是个负数 -- 表示客户端系统时间被调整过

// 这两种情况都可以立刻对func做调用

if (remaining <= 0 || remaining > wait) {

clearTimeout(timeout);

timeout = null;

// 重置上一次开始时间为本次结束时间

previous = now;

//执行函数

result = func.apply(context, args);

if (!timeout) context = args = null;

//如果延迟执行不存在，且没有设定结尾边界不执行选项

//判断!timeout的目的是什么？

//将入在一个wait里面我们执行了三次resize，timeout是不应该被更新的

} else if (!timeout && options.trailing !== false) {

timeout = setTimeout(later, remaining);

}

return result;

};

}

## 2、页面性能拓展