目录

22 防抖节流背后那些事儿

更新时间：2019-09-04 10:38:11

> 人生的价值，并不是用时间，而是用深度去衡量的。
>
> ——列夫·托尔斯泰

引入

防抖和节流这两个概念很多人都分不太清楚，常常混淆两个概念，虽然它们都是用于控制频繁触发的事件，如：鼠标移动事件、滚动事件，但其实它们两个有着本质的区别。这一节就为大家带来与防抖和节流有关的性能优化知识，帮助区分两者的概念。那么在开讲之前，我们首先给出一个网站，大家可以上去实际体验一下防抖和节流的效果，我觉得这个网站做的非常好，基本一看就懂，点击这里。

如上，这个网站可以非常方便的体验防抖和节流的效果，我们在左侧不停的移动鼠标，这个时候我们就频繁的触发了鼠标的移动事件(mousemove)，然后右侧是对左侧移动事件触发效果的显示，其中Regular是没有设置防抖和节流的效果显示，可以看到会被频繁触发；debounce(防抖)这个事件是在你停止移动鼠标之后，触发一次；而throttle（节流）则是每隔一段时间触发一次。相信你通过这个网站对防抖、节流已经有了一个大概的了解，接下来，我们会通过具体的代码案例来帮助大家更加深入的进行理解。

节流的原理是不管你在一段时间内如何不停地触发事件，只要设置了节流，就会每隔一段时间执行一次。我们可以用大坝来类比：在雨季水特别多的时候，我们肯定要打开整个水闸让水流的快一些；但是当旱季来临的时候，这个时候水资源严重匮乏，我们肯定不能直接关掉水闸，而是将水闸开的小一点，虽然流的慢，但是可以保持正常供应。那么节流就是这个道理，不能完成不触发的情况下，我们可以让它在一段时间内触发一次，这样就避免了频繁操作带来的性能问题。下面我们来实现一个简单的节流函数，如下：

```
function throttle(callback, wait) {
    let timeout;
    let previous = 0;

    return function() {
        let context = this;
        let args = arguments;

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

    }
}
```

上面这种是使用定时器方式的节流函数，我们也可以使用时间戳的方式来进行，代码如下：

```
function throttle(callback, wait) {
    let context, args;
    let previous = 0;

    return function() {
        let now = +new Date();
        let context = this;
        let args = arguments;

        if (now - previous > wait) {
            callback.apply(context, args);
            previous = now;
        }
    }
}
```

其实两种方法原理上是相同的，只不过在使用方式上，一个用定时器，一个使用时间戳。

防抖(debounce)

防抖的原理则是不管你在一段时间内如何不停的触发事件，只要设置了防抖，则只在触发n秒后才执行。如果我们在一个事件触发的n秒内又触发了相同的事件，那我们便以新的事件时间为标准，n秒之后再执行。这里可以类比HDR照片：如果我们开启了HDR设置，那么在按下快门的一瞬间，相机会自动拍下非常多的照片，然后多张照片在最后合成最清晰的一张，也就是以最后那张为准。防抖同样如此，也是为了避免频发触发带来的性能问题，下面我们就来实现一个简单的防抖函数，如下：

```
    return function () {
        let context = this;
        let args = arguments;

        clearTimeout(timeout)
        timeout = setTimeout(function(){
            callback.apply(context, args)
        }, wait);
    }
}
```

介绍完了简单的防抖函数，我们再思考如下一种场景，不想等到停止触发事件后才执行对应代码，而是希望立马执行代码；等到停止触发n秒后，然后再重新触发执行。这种如何写呢？我们可以设置一个字段offhand来判断是否立刻执行，代码如下：

```
function debounce(func, wait, offhand) {
    let timeout;

    return function () {
        let context = this;
        let args = arguments;

        if (timeout) clearTimeout(timeout);
        if (offhand) {
            let callNow = !timeout;

            timeout = setTimeout(function(){
                timeout = null;
            }, wait)
            if (callNow) func.apply(context, args)
        }
        else {
            timeout = setTimeout(function(){
                func.apply(context, args)
            }, wait);
        }
    }
}
```

这样我们通过控制offhand的值，就可以控制是否要立刻执行触发后的代码。其实这样的场景在业务中非常多，大家下去要多注意总结类似案例。

小结

以上内容就是对防抖和节流的解析。其实防抖和节流本身并不难，难的是根据业务进行自己的二次封装，这个是需要大家下去根据自身的业务反复总结提炼的。如果你的项目当中有这些被频发触发的事件，不妨使用一下我们今天介绍的防抖和节流函数，看看性能是否会有明显提高呢，欢迎在评论区与我进行交流。

← 21 Event Loop 力挽狂澜 23 让加载“懒”一点 →

精选留言 0

目录

目前暂无任何讨论

千学不如一看，千看不如一练