方法二、利用for嵌套for，然后splice去重（ES5中最常用)

```js
function unique(arr){
        for(var i=0; i<arr.length; i++){
            for(var j=i+1; j<arr.length; j++){
                if(arr[i]==arr[j]){         //第一个等同于第二个， splice方法删除
                    arr.splice(j,1);
                    j--;
                }
            }
        }
    return arr;
}
var arr = [1,1, 'true', 'true',true,true,15,15,false,false, undefined,undefin
console.log(unique(arr))
    //[1, "true", 15, false, undefined, NaN, NaN, "NaN", "a", {…}, {…}]
```

- 双层循环，外层循环元素， 内层循环时比较值 。值相同时，则删去这个值。
- 想快速学习更多常用的 ES6 语法

方法三、利用indexOf去重

```js
function unique(arr) {
    if ( !Array.isArray(arr)) {
        console.log( 'type error!')
        return
    }
    var array = [];
    for (var i = 0; i < arr.length; i++) {
        if (array .indexOf(arr[i]) === -1) {
            array .push(arr[i])
        }
    }
    return array;
}
var arr = [1,1, 'true', 'true',true,true,15,15,false,false, undefined,undefin
console.log(unique(arr))
   // [1, "true", true, 15, false, undefined, null, NaN, NaN, "NaN", 0, "a"
```

> 新建一个空的结果数组， for 循环原数组， 判断结果数组是否存在当前元素， 如果有相同的值则跳过，不相同则 push 进数组

方法四、利用sort()

```js
function unique(arr) {
    if ( !Array.isArray(arr)) {
        console.log( 'type error!')
        return;
    }
    arr = arr.sort()
    var arrry= [arr[0]];
    for (var i = 1; i < arr.length; i++) {
        if (arr[i] !== arr[i-1]) {
            arrry.push(arr[i]);
        }
    }
    return arrry;
}
var arr = [1,1, 'true', 'true',true,true,15,15,false,false, undefined,undefin
console.log(unique(arr))
// [0, 1, 15, "NaN", NaN, NaN, {…}, {…}, "a", false, null, true, "true", un
```

> 利用 sort() 排序方法，然后根据排序后的结果进行遍历及相邻元素比对

方法五、利用对象的属性不能相同的特点进行去重

```js
function unique(arr) {
    if ( !Array.isArray(arr)) {
        console.log( 'type error!')
        return
    }
    var arrry= [];
     var  obj = {};
    for (var i = 0; i < arr.length; i++) {
        if ( !obj [arr[i]]) {
            arrry.push(arr[i])
            obj [arr[i]] = 1
        } else {
            obj [arr[i]]++
        }
    }
    return arrry;
}
var arr = [1,1, 'true', 'true',true,true,15,15,false,false, undefined,undefin
```

```
console.log(unique(arr))
//[1, "true", 15, false, undefined, null, NaN, 0, "a", {…}]    //两个true直接
```

方法六、利用includes

```js
function unique(arr) {
    if ( !Array.isArray(arr)) {
        console.log( 'type error!')
        return
    }
    var array = [];
    for(var i = 0; i < arr.length; i++) {
            if( !array.includes( arr[i]) ) {//includes 检测数组是否有某个值
                    array.push(arr[i]);
              }
    }
    return array
}
var arr = [1,1, 'true', 'true',true,true,15,15,false,false, undefined,undefin
console.log(unique(arr))
    //[1, "true", true, 15, false, undefined, null, NaN, "NaN", 0, "a", {…}
```

方法七、利用hasOwnProperty

```js
function unique(arr) {
    var obj = {};
    return arr.filter(function(item, index, arr){
        return obj.hasOwnProperty(typeof item + item) ? false : (obj [typeof
    })
}
    var arr = [1,1, 'true', 'true',true,true,15,15,false,false, undefined,und
        console.log(unique(arr))
//[1, "true", true, 15, false, undefined, null, NaN, "NaN", 0, "a", {…}]
```

> 利用 hasOwnProperty 判断是否存在对象属性

方法八、利用filter

```js
function unique(arr) {
  return arr.filter(function(item, index, arr) {
    //当前元素，在原始数组中的第一个索引==当前索引值，否则返回当前元素
    return arr.indexOf(item, 0) === index;
  });
}
var arr = [1,1, 'true', 'true',true,true,15,15,false,false, undefined,undefin
console.log(unique(arr))
//[1, "true", true, 15, false, undefined, null, "NaN", 0, "a", {…}, {…}]
```

方法九、利用递归去重

```js
function unique(arr) {
    var array= arr;
    var len = array.length;

    array.sort(function(a,b){   //排序后更加方便去重
        return a - b;
    })

    function loop(index){
        if(index >= 1){
            if(array[index] === array[index-1]){
            array.splice(index,1);
            }
            loop(index - 1);    //递归loop，然后数组去重
        }
    }
    loop(len-1);
    return array;
}
var arr = [1,1, 'true', 'true',true,true,15,15,false,false, undefined,undefin
console.log(unique(arr))
//[1, "a", "true", true, 15, false, 1, {…}, null, NaN, NaN, "NaN", 0, "a",
```

方法十、利用Map数据结构去重

```js
function arrayNonRepeatfy(arr) {
    let map = new Map();
        let array = new Array();  // 数组用于返回结果
        for (let i = 0; i < arr.length; i++) {
            if(map .has(arr[i])) {  // 如果有该key值
```

```
                map .set(arr[i], true);
            } else {
                map .set(arr[i], false);    // 如果没有该key值
                array .push(arr[i]);
            }
        }
        return array ;
    }
     var arr = [1,1, 'true', 'true',true,true,15,15,false,false, undefined,undefi
        console.log(unique(arr))
    //[1, "a", "true", true, 15, false, 1, {…}, null, NaN, NaN, "NaN", 0, "a",
```

> 创建一个空 Map 数据结构， 遍历需要去重的数组， 把数组的每一个元素作为 key 存到 Map 中 。由于 Map 中不会出现相同的 key 值，所以最终得到的就是去重后的结果

方法十一 、利用reduce+includes

```
function unique(arr){
    return arr.reduce((prev,cur) => prev.includes(cur) ? prev : [...prev,cu
}
var arr = [1,1, 'true', 'true',true,true,15,15,false,false, undefined,undefin
console.log(unique(arr));
// [1, "true", true, 15, false, undefined, null, NaN, "NaN", 0, "a", {…}, {
```

方法十二、[...new Set(arr)]

```
[...new Set(arr)]
//代码就是这么少----（其实，严格来说并不算是一种，相对于第一种方法来说只是简化了代码)
```

75 （设计题）想实现一个对页面某个节点的拖曳？如何做？ （使用原生JS)

- 给需要拖拽的节点绑定 mousedown ， mousemove ， mouseup 事件
- mousedown 事件触发后， 开始拖拽
- mousemove 时， 需要通过 event.clientX 和 clientY 获取拖拽位置， 并实时更新位置
- mouseup 时，拖拽结束

需要注意浏览器边界的情况

76 Javascript全局函数和全局变量

全局变量

- `Infinity` 代表正的无穷大的数值。
- `NaN` 指示某个值是不是数字值。
- `undefined` 指示未定义的值。

全局函数

- `decodeURI()` 解码某个编码的 `URI` 。
- `decodeURIComponent()` 解码一个编码的 `URI` 组件。
- `encodeURI()` 把字符串编码为 URI。
- `encodeURIComponent()` 把字符串编码为 `URI` 组件。
- `escape()` 对字符串进行编码。
- `eval()` 计算 `JavaScript` 字符串， 并把它作为脚本代码来执行。
- `isFinite()` 检查某个值是否为有穷大的数。
- `isNaN()` 检查某个值是否是数字。
- `Number()` 把对象的值转换为数字。
- `parseFloat()` 解析一个字符串并返回一个浮点数。
- `parseInt()` 解析一个字符串并返回一个整数。
- `String()` 把对象的值转换为字符串。
- `unescape()` 对由 `escape()` 编码的字符串进行解码

77 使用js实现一个持续的动画效果

定时器思路

```js
var e = document.getElementById( 'e')
var flag = true;
var left = 0;
setInterval(() => {
    left == 0 ? flag = true : left == 100 ? flag = false : ''
    flag ? e.style.left = ` ${left++}px` : e.style.left = ` ${left--}px`
}, 1000 / 60)
```

requestAnimationFrame

```js
//兼容性处理
window.requestAnimFrame = (function(){
    return window.requestAnimationFrame       ||
           window.webkitRequestAnimationFrame ||
           window.mozRequestAnimationFrame    ||
           function(callback){
               window.setTimeout(callback, 1000 / 60);
           };
})();

var e = document.getElementById("e");
var flag = true;
var left = 0;

function render() {
    left == 0 ? flag = true : left == 100 ? flag = false : '';
    flag ? e.style.left = ` ${left++}px` :
        e.style.left = ` ${left--}px`;
}

(function animloop() {
    render();
    requestAnimFrame(animloop);
})();
```

使用css实现一个持续的动画效果

```css
animation: mymove  5 s  infinite;

@keyframes mymove {
    from {top:0px;}
    to {top:200px;}
}
```

- animation-name 规定需要绑定到选择器的 keyframe 名称。
- animation-duration 规定完成动画所花费的时间， 以秒或毫秒计。
- animation-timing-function 规定动画的速度曲线。
- animation-delay 规定在动画开始之前的延迟。
- animation-iteration-count 规定动画应该播放的次数。
- animation-direction 规定是否应该轮流反向播放动画

78 封装一个函数，参数是定时器的时间， .then执行回调函数

```js
function sleep (time) {
    return new Promise((resolve) => setTimeout(resolve, time));
}
```

79 怎么判断两个对象相等?

```js
obj={
    a:1,
    b:2
}
obj2={
    a:1,
    b:2
}
obj3={
    a:1,
    b:'2'
}
```

> 可以转换为字符串来判断

```js
JSON.stringify(obj)==JSON.stringify(obj2);//true
JSON.stringify(obj)==JSON.stringify(obj3);//false
```

80 项目做过哪些性能优化?

- 减少 HTTP 请求数
- 减少 DNS 查询
- 使用 CDN
- 避免重定向
- 图片懒加载
- 减少 DOM 元素数量
- 减少 DOM 操作

- 使用外部 JavaScript 和 CSS
- 压缩 JavaScript 、 CSS 、字体 、图片等
- 优化 CSS Sprite
- 使用 iconfont
- 字体裁剪
- 多域名分发划分内容到不同域名
- 尽量减少 iframe 使用
- 避免图片 src 为空
- 把样式表放在 link 中
- 把 JavaScript 放在页面底部

81 浏览器缓存

> 浏览器缓存分为强缓存和协商缓存 。当客户端请求某个资源时， 获取缓存的流程如下

- 先根据这个资源的一些 http header 判断它是否命中强缓存， 如果命中，则直接从本地获取缓存资源，不会发请求到服务器；
- 当强缓存没有命中时，客户端会发送请求到服务器， 服务器通过另一些 request header 验证这个资源是否命中协商缓存，称为 http 再验证， 如果命中， 服务器将请求返回，但不返回资源， 而是告诉客户端直接从缓存中获取，客户端收到返回后就会从缓存中获取资源；
- 强缓存和协商缓存共同之处在于， 如果命中缓存， 服务器都不会返回资源； 区别是， 强缓存不对发送请求到服务器，但协商缓存会。
- 当协商缓存也没命中时， 服务器就会将资源发送回客户端。
- 当 ctrl+f5 强制刷新网页时， 直接从服务器加载，跳过强缓存和协商缓存；
- 当 f5 刷新网页时，跳过强缓存，但是会检查协商缓存；

强缓存

- Expires （该字段是 http1.0 时的规范，值为一个绝对时间的 GMT 格式的时间字符串，代表缓存资源的过期时间)
- Cache-Control:max-age （该字段是 http1.1 的规范， 强缓存利用其 max-age 值来判断缓存资源的最大生命周期， 它的值单位为秒)

协商缓存

- Last-Modified （值为资源最后更新时间， 随服务器response返回)

- If-Modified-Since （通过比较两个时间来判断资源在两次请求期间是否有过修改， 如果没有修改，则命中协商缓存)
- ETag (表示资源内容的唯一标识， 随服务器 response 返回)
- If-None-Match （服务器通过比较请求头部的 If-None-Match 与当前资源的 ETag 是否一致来判断资源是否在两次请求之间有过修改， 如果没有修改，则命中协商缓存)

82 WebSocket

> 由于 http 存在一个明显的弊端（消息只能有客户端推送到服务器端， 而服务器端不能主动推送到客户端)， 导致如果服务器如果有连续的变化， 这时只能使用轮询， 而轮询效率过低， 并不适合 。于是 WebSocket 被发明出来

> 相比与 http 具有以下有点

- 支持双向通信， 实时性更强;
- 可以发送文本，也可以二进制文件;
- 协议标识符是 ws ，加密后是 wss ;
- 较少的控制开销 。连接创建后， ws 客户端 、服务端进行数据交换时，协议控制的数据包头部较小 。在不包含头部的情况下， 服务端到客户端的包头只有 2~10 字节（取决于数据包长度)， 客户端到服务端的的话， 需要加上额外的4字节的掩码 。而 HTTP 协议每次通信都需要携带完整的头部;
- 支持扩展 。ws协议定义了扩展，用户可以扩展协议， 或者实现自定义的子协议 。 （比如支持自定义压缩算法等)
- 无跨域问题。

> 实现比较简单， 服务端库如 socket.io 、 ws ， 可以很好的帮助我们入门。而客户端也只需要参照 api 实现即可

83 尽可能多的说出你对 Electron 的理解

> 最最重要的一点， electron 实际上是一个套了 Chrome 的 nodeJS 程序

所以应该是从两个方面说开来

- Chrome （无各种兼容性问题);

- NodeJS （NodeJS 能做的它也能做）

84 深浅拷贝

浅拷贝

- Object.assign
- 或者展开运算符

深拷贝

- 可以通过 JSON.parse(JSON.stringify(object)) 来解决

```js
let a = {
    age: 1,
    jobs: {
        first: 'FE'
    }
}
let b = JSON.parse(JSON.stringify(a))
a.jobs.first = 'native'
console.log(b.jobs.first) // FE
```

该方法也是有局限性的

- 会忽略 undefined
- 不能序列化函数
- 不能解决循环引用的对象

85 防抖/节流

防抖

> 在滚动事件中需要做个复杂计算或者实现一个按钮的防二次点击操作 。可以通过函数防抖动来实现

```js
// 使用 underscore 的源码来解释防抖动

/**
 * underscore 防抖函数， 返回函数连续调用时， 空闲时间必须大于或等于 wait， func 才会执行
```

```
 *
 * @param  {function} func          回调函数
 * @param  {number}   wait          表示时间窗口的间隔
 * @param  {boolean}  immediate     设置为ture时，是否立即调用函数
 * @return {function}               返回客户调用函数
 */
_.debounce = function(func, wait, immediate) {
    var timeout, args, context, timestamp, result;

    var later = function() {
      // 现在和上一次时间戳比较
      var last = _.now() - timestamp;
      // 如果当前间隔时间少于设定时间且大于0就重新设置定时器
      if (last < wait && last >= 0) {
        timeout = setTimeout(later, wait - last);
      } else {
        // 否则的话就是时间到了执行回调函数
        timeout = null;
        if ( !immediate) {
          result = func.apply(context, args);
          if ( !timeout) context = args = null;
        }
      }
    };

    return function() {
      context = this;
      args = arguments;
      // 获得时间戳
      timestamp = _.now();
      // 如果定时器不存在且立即执行函数
      var callNow = immediate && !timeout;
      // 如果定时器不存在就创建一个
      if ( !timeout) timeout = setTimeout(later, wait);
      if (callNow) {
        // 如果需要立即执行函数的话 通过 apply 执行
        result = func.apply(context, args);
        context = args = null;
      }

      return result;
    };
  };
```

整体函数实现

对于按钮防点击来说的实现

- 开始一个定时器， 只要我定时器还在，不管你怎么点击都不会执行回调函数 。一旦定时器结束并设置为 null，就可以再次点击了
- 对于延时执行函数来说的实现：每次调用防抖动函数都会判断本次调用和之前的时间间隔， 如果小于需要的时间间隔，就会重新创建一个定时器， 并且定时器的延时为设定时间减去之前的时间间隔 。一旦时间到了，就会执行相应的回调函数

节流

> 防抖动和节流本质是不一样的 。防抖动是将多次执行变为最后一次执行， 节流是将多次执行变成每隔一段时间执行

```js
/**
 * underscore 节流函数， 返回函数连续调用时， func 执行频率限定为 次 / wait
 *
 * @param  {function}   func      回调函数
 * @param  {number}    wait       表示时间窗口的间隔
 * @param  {object}     options   如果想忽略开始函数的的调用，传入{leading: fals
 *                                如果想忽略结尾函数的调用，传入{trailing: false
 *                                两者不能共存，否则函数不能执行
 * @return {function}             返回客户调用函数
 */
_.throttle = function(func, wait, options) {
    var context, args, result;
    var timeout = null;
    // 之前的时间戳
    var previous = 0;
    // 如果 options 没传则设为空对象
    if ( !options) options = {};
    // 定时器回调函数
    var later = function() {
      // 如果设置了 leading，就将 previous 设为 0
      // 用于下面函数的第一个 if 判断
      previous = options.leading === false ? 0 : _.now();
      // 置空一是为了防止内存泄漏，二是为了下面的定时器判断
      timeout = null;
      result = func.apply(context, args);
      if ( !timeout) context = args = null;
    };
    return function() {
      // 获得当前时间戳
```

```
    var now = _.now();
    // 首次进入前者肯定为 true
    // 如果需要第一次不执行函数
    // 就将上次时间戳设为当前的
    // 这样在接下来计算 remaining 的值时会大于0
    if ( !previous && options.leading === false) previous = now;
    // 计算剩余时间
    var remaining = wait - (now - previous);
    context = this;
    args = arguments;
    // 如果当前调用已经大于上次调用时间 + wait
    // 或者用户手动调了时间
    // 如果设置了 trailing，只会进入这个条件
    // 如果没有设置 leading，那么第一次会进入这个条件
    // 还有一点，你可能会觉得开启了定时器那么应该不会进入这个 if 条件了
    // 其实还是会进入的，因为定时器的延时
    // 并不是准确的时间，很可能你设置了2秒
    // 但是他需要2.2秒才触发，这时候就会进入这个条件
    if (remaining <= 0 || remaining > wait) {
      // 如果存在定时器就清理掉否则会调用二次回调
      if (timeout) {
        clearTimeout(timeout);
        timeout = null;
      }
      previous = now;
      result = func.apply(context, args);
      if ( !timeout) context = args = null;
    } else if ( !timeout && options.trailing !== false) {
      // 判断是否设置了定时器和 trailing
      // 没有的话就开启一个定时器
      // 并且不能不能同时设置 leading 和 trailing
      timeout = setTimeout(later, remaining);
    }
    return result;
  };
};
```

86 谈谈变量提升?

> 当执行 JS 代码时，会生成执行环境，只要代码不是写在函数中的，就是在全局执行环境中，函数中的代码会产生函数执行环境，只此两种执行环境

- 接下来让我们看一个老生常谈的例子，`var`

```js
b() // call b
console.log(a) // undefined

var a = 'Hello world'

function b() {
    console.log( 'call b')
}
```

> 变量提升
>
> 这是因为函数和变量提升的原因 。通常提升的解释是说将声明的代码移动到了顶部，这其实没有什么错误，便于大家理解 。但是更准确的解释应该是：在生成执行环境时，会有两个阶段 。第一个阶段是创建的阶段，JS 解释器会找出需要提升的变量和函数， 并且给他们提前在内存中开辟好空间， 函数的话会将整个函数存入内存中， 变量只声明并且赋值为 `undefined` ，所以在第二个阶段，也就是代码执行阶段， 我们可以直接提前使用

在提升的过程中，相同的函数会覆盖上一个函数， 并且函数优先于变量提升

```js
b() // call b second

function b() {
    console.log( 'call b fist')
}
function b() {
    console.log( 'call b second')
}
var b = 'Hello world'
```

> 复制代码 `var` 会产生很多错误，所以在 `ES6` 中引入了 `let` 。 `let` 不能在声明前使用，但是这并不是常说的 `let` 不会提升， `let` 提升了，在第一阶段内存也已经为他开辟好了空间，但是因为这个声明的特性导致了并不能在声明前使用

87 什么是单线程，和异步的关系

- 单线程 - 只有一个线程， 只能做一件事