面试题

1、解释一下什么是闭包？

- 闭包：就是能够读取外层函数内部变量的函数。
- 闭包需要满足三个条件：
  - 访问所在作用域；
  - 函数嵌套；
  - 在所在作用域外被调用。
- 优点： 可以重复使用变量，并且不会造成变量污染。
- 缺点： 会引起内存泄漏
- 使用闭包的注意点：
  - 由于闭包会使得函数中的变量都被保存在内存中，内存消耗很大，所以不能滥用闭包，否则会造成网页的性能问题，在IE中可能导致内存泄露。解决方法是，在退出函数之前，将不使用的局部变量全部删除。
  - 闭包会在父函数外部，改变父函数内部变量的值。所以，如果你把父函数当作对象（object）使用，把闭包当作它的公用方法（Public Method），把内部变量当作它的私有属性（private value），这时一定要小心，不要随便改变父函数内部变量的值。

2、解释一下原型和原型链？

原型

> 原型就是一个为对象实例定义了一些公共属性和公共方法的对象模板。

原型链

> 对象之间的继承关系通过构造函数的`prototype`指向父类对象，直到指向`Object`对象为止形成的指向链条。
>
> 通俗讲： 原型链是原型对象创建过程的历史记录。
>
> 注：在`javascript`中，所有的对象都拥有一个`__proto__`属性指向该对象的原型（`prototype`）。

3、说一下 ES6 中你熟悉的一些内容？

- class 类的继承ES6中不再像ES5一样使用原型链实现继承，而是引入Class这个概念
- async、await使用 async/await, 搭配promise,可以通过编写形似同步的代码来处理异步流程, 提高代码的简洁性和可读性async 用于申明一个 function 是异步的，而 await 用于等待一个异步方法执行完成
- Promise是异步编程的一种解决方案，比传统的解决方案（回调函数和事件）更合理、强大
- Symbol是一种基本类型。Symbol 通过调用symbol函数产生，它接收一个可选的名字参数，该函数返回的symbol是唯一的
- Proxy代理使用代理（Proxy）监听对象的操作，然后可以做一些相应事情

- Set是类似于数组的数据集合，无序，插入删除速度快，元素不重复，查找速度快。
- Map是一个类似对象的数据结构，和对象不同的在于它的key可以是任意类型，但是对象只能使用string和symbol类型，Map的存储关联性更强
- 生成器函数可以进行阻断函数执行的过程，通过传参可以传入新的值进入函数继续执行，可以用于将异步变为阻塞式同步

4、数组排序的方式？

- 冒泡排序：

```
    for(var i=0;i<arr.length-1;i++){
            for(var j=0;j<arr.length-i-1;j++){
                if(arr[j]>arr[j+1]){
                    var temp=arr[j];
                    arr[j]=arr[j+1];
                    arr[j+1]=temp;
                }
            }
        if(arr[j]===arr[j-1]) i++;
        }
```

- 选择排序：

```
 for(var i=0;i<arr.length;i++){
            var min=i;
            for(var j=i+1;j<arr.length;j++){
                if(arr[j]<arr[min]) min=j;
            }
            if(min!==i){
                var temp=arr[i];
                arr[i]=arr[min];
                arr[min]=temp;
            }
        if(arr[i]===arr[i+1])i++;
}
```

- 快速排序：

```
function quickSort(arr) {
   if (arr.length <= 1) return arr;
    var centerIndex = ~~(arr.length / 2);
    var left = [];
    var right = [];
    for (var i = 0; i < arr.length; i++) {
        if (i === centerIndex) continue;
        if (arr[i] < arr[centerIndex]) left.push(arr[i]);
        else right.push(arr[i]);
        }
        return quickSort(left).concat(arr[centerIndex], quickSort(right));
    }
```

5、什么是事件轮询(EventLoop)？

一个用来等待和发送消息和事件的程序结构。

- 1、所有任务都在主线程上执行，形成一个执行栈。
- 2、主线程发现有异步任务，如果是微任务就把他放到微任务的消息队列里，如果是宏任务就把他放到宏任务的消息队列里。
- 3、执行栈所有同步任务执行完毕。
- 4、执行微任务队列，之后再执行宏任务队列。
- 5、轮询第4步。

6、数组的一些API, 哪些能够改变原数组, 那些不能？

- 改变原数组的方法：

```
shift()
unshift()
pop()
push()
reverse()
sort()
splice()
```

- 不改变原数组的方法：

```
concat()
every()
filter()
forEach()
indexOf()
join()
lastIndexOf()
map()
some()
every()
slice()
reduce()
reduceRight()
flat()
flatMap()
find()
```

7、for 循环与 forEach 的区别？

- 1.for循环可以使用break跳出循环，但forEach不能。
- 2.for循环可以控制循环起点（i初始化的数字决定循环的起点），forEach只能默认从索引0开始。
- 3.for循环过程中支持修改索引（修改 i），但forEach做不到（底层控制index自增，无法左右它）。

8、深浅拷贝？

- 深拷贝：

```
function cloneObject(source, target) {
```

```
      if (target === undefined) {
        if (Node.prototype.isPrototypeOf(source)) {
          target = document.createElement(source.nodeName);
          target.style = source.style.cssText;
        } else if (source.constructor === Uint8Array) {
          target = new source.constructor(Array.from(source));
        } else if (source.constructor === Date || source.constructor === RegExp ||
source.constructor === Set || source
          .constructor === Map) {
          target = new source.constructor(source);
        } else if (source.constructor === Function) {
          var arg = source.toString().match(/\((.*?)\)/)[1];
          var content = source.toString().replace(/\n|\r/g, "").match(/\{(.*)\}/)[1];
          target = new Function(arg, content)
        } else {
          target = new source.constructor();
        }
      }
      var names =
Object.getOwnPropertyNames(source).concat(Object.getOwnPropertySymbols(source));
      for (var i = 0; i < names.length; i++) {
        if (names[i] === "constructor") {
          Object.defineProperty(target, "constructor", {
            value: source.constructor
          });
          continue;
        }
        var desc = Object.getOwnPropertyDescriptor(source, names[i]);
        if ((typeof desc.value === "object" && desc.value !== null) || typeof
desc.value === "function") {
          var o = cloneObject(desc.value)
          Object.defineProperty(target, names[i], {
            value: o,
            enumerable: desc.enumerable,
            writable: desc.writable,
            configurable: desc.configurable
          })
        } else {
          Object.defineProperty(target, names[i], desc);
        }
      }
      return target;
    }
```

- 浅拷贝：

```
    1、Object.assign(目标对象，源对象)
    2、
    var obj1={}
    for(var key in obj){
        obj1[key]=obj[key]
    }
    3、obj1={...obj};
```

9、 url 的组成？

```
http:/https:    协议
www.baidu.com  域名
:8080  端口
/sf/vsearch  路径
?wd=百度热搜    查询(可有可无)
#a=1&b=2  哈希值(可有可无)
```

10、常见的跨域方式？

- JSONP：
  JSONP是利用外链脚本，没有跨源限制的特点，来实现跨源请求的一种技术。
- CORS：
  cors：跨域资源共享，是一种实现跨源请求数据的技术。这就是跨源问题的解决方案之一。也是广泛的解决方案。
- 正向代理
  先搭建一个属于自己的代理服务器
  - 1、用户发送请求到自己的代理服务器
  - 2、自己的代理服务器发送请求到服务器
  - 3、服务器将数据返回到自己的代理服务器
  - 4、自己的代理服务器再将数据返回给用户
- 反向代理
  - 1、用户发送请求到服务器（访问的其实是反向代理服务器，但用户不知道）
  - 2、反向代理服务器发送请求到真正的服务器
  - 3、真正的服务器将数据返回给反向代理服务器
  - 4、反向代理服务器再将数据返回给用户
- 通过postMassage，

11、Promise 的使用场景？

- 场景1：获取文件信息。
- 场景2：配合AJAX获取信息
- 场景3：解决回调地狱，实现串行任务队列。
- 场景4: node中进行本地操作的异步过程

12、let, const, var 的区别？

```
声明方式  变量提升  暂时性死区  重复声明  初始值  作用域
var        允许     不存在        允许    不需要  非块级
let      不允许      存在       不允许     不需要     块级
const    不允许      存在       不允许      需要      块级
```

13、对 this 的理解, 三种改变 this 的方式？

- 1.任何情况下直接在script中写入的this都是window。

- 2.函数中的this 非严格模式：this指向window， 严格模式时：this指向undefined。
- 3.箭头函数的this
  this都指向箭头函数外上下文环境的this指向
- 4.对象中this
  对象属性的this 指向对象外上下文环境的this
  对象方法(普通函数)中的this，指向当前对象(谁执行该方法，this就指向谁)
- 5.回调函数的this指向
  - 1)、 setTimeout，setInterval回调函数不管是否是严格模式都会指向window。
  - 2)、通过在函数内执行当前回调函数 非严格模式：this指向window， 严格模式时：this指向undefined。
  - 3）递归函数中的this 非严格模式：this指向window， 严格模式时：this指向undefined。
  - 4）使用arguments`0`执行函数时 this指向arguments。
  - 5）事件中的回调函数,this指向事件侦听的对象(e.currentTarget);
- 6、call，apply，bind方法执行时this的指向
  - 如果call,apply,bind传参时，第一个参数传入的不是null或者undefined，传入什么this指向什么
  - 如果第一个参数传入的是null或者undefined ,非严格模式下指向window
- 7、在ES6的类中this的指向
  - 构造函数中的this指向实例当前类所产生的新的实例对象
  - 类中实例化方法中this指向谁执行该方法，this指向谁
  - 类中静态方法中this执行该类或者该类的构造函数
  - 类中实例化箭头方法，this仍然指向当前类实例化的实例对象
- 8、ES5的原型对象中this的指向
  - 在原型的方法中，this指向实例化当前构造函数的实例化对象（谁执行该方法，this指向谁）；
  - 三种改变this指向的方式
    - 函数名.call（this,....）this写谁就指谁。
    - 函数名.apply(this,[参数1，参数2，...]) this写谁就指谁。
    - 函数名. bind (this,1,2,3) this写谁就指谁。

14、 cookie, localStorage,sessionStorage 的区别？

存储方式 作用与特性 存储数量及大小

- cookie

```
存储方式
 存储用户信息，获取数据需要与服务器建立连接。
 以路径存储，上层路径不能访问下层的路径cookie，下层的路径cookie可以访问上层的路径cookie
 作用与特性

可存储的数据有限，且依赖于服务器，无需请求服务器的数据尽量不要存放在cookie 中，以免影响页面性能。

可设置过期时间。

存储数量及大小 将cookie控制在4095B以内，超出的数据会被忽略。
IE6或更低版本 最多存20个cookie；
IE7及以上
版本 多可以有50个；
Firefox多 50个；
chrome和Safari没有做硬性限制。
```

cookie最大特征就是可以在页面与服务器间互相传递，当发送或者接受数据时自动传递

localStorage

存储客户端信息，无需请求服务器。

数据永久保存，除非用户手动清理客户端缓存。

开发者可自行封装一个方法，设置失效时间。　　5M左右，各浏览器的存储空间有差异。

任何地方都可以存都可以取

操作简单

sessionStorage

存储客户端信息，无需请求服务器。

数据保存在当前会话，刷新页面数据不会被清除，结束会话（关闭浏览器、关闭页面、跳转页面）数据失效。

5M左右，各浏览器的存储空间有差异。

同页面不同窗口中数据不会共享

15、输入 url 到打开页面 都做了什么事情？

- 输入URL
- 访问hosts解析，如果没有解析访问DNS解析
- TCP握手
- HTTP请求
- HTTP响应返回数据
- 浏览器解析并渲染页面

16、原生 ajax 的流程？

```
创建xhr
var xhr=new XMLHTTPRequest()
侦听通信状态改变的事件
xhr.addEventListener("readystatechange",readyStateChangeHandler);
Method 分为 get post put delete等等
Async 异步同步
name和password是用户名和密码
xhr.open(Method,URL,Async,name,password)
发送内容给服务器
xhr.send(内容)

function readyStateChangeHandler(e){
  当状态是4时，并且响应头成功200时，
   if(xhr.readyState===4 && xhr.status===200){
     打印返回的消息
     console.log(xhr.response)
```

```
        }
    }
```

17、如何实现继承？

- 对于 JavaScript 来说，继承有两个要点：
    1. 复用父构造函数中的代码
    2. 复用父原型中的代码第一种实现复用父构造函数中的代码，我们可以考虑调用父构造函数并将 this 绑定到子构造函数。
- 第一种方法：复用父原型中的代码，我们只需改变原型链即可。将子构造函数的原型对象的 proto 属性指向父构造函数的原型对象。
- 第二种实现
  使用 new 操作符来替代直接使用 proto 属性来改变原型链。
- 第三种实现
  使用一个空构造函数来作为中介函数，这样就不会将构造函数中的属性混到 prototype 中

```
function A(x,y){
  this.x = x
  this.y = y
}
A.prototype.run = function(){}
// 寄生继承 二者一起使用
function B(x,y){
  A.call(this,x,y)  // 借用继承
}
B.prototype = new A() // 原型继承
// 组合继承
Function.prototype.extends = function(superClass){
  function F(){}
  F.prototype = superClass.prototype
  if(superClass.prototype.constructor !== superClass){
    Object.defineProperty(superClass.prototype,'constructor',{value:superClass})
  }
  let proto = this.prototype
  this.prototype = new F()
  let names = Reflect.ownKeys(proto)
  for(let i = 0; i < names.length;i++){
    let desc = Object.getOwnPropertyDescriptor(proto,names[i])
    Object.defineProperty(this.prototypr,name[i],desc)
    }
  this.prototype.super = function(arg){
    superClass.apply(this,arg)
    }
  this.prototype.supers = superClass.prototype
}
```

- 第四种实现
  es6类的继承extends。

18、null 和 undefined 的区别？

- null是一个表示"无"的对象（空对象指针），转为数值时为0；
- undefined是一个表示"无"的原始值，转为数值时为NaN。
  拓展：
- null表示"没有对象"，即该处不应该有值。典型用法是：
  - 作为函数的参数，表示该函数的参数不是对象。
  - 作为对象原型链的终点。
- undefined表示"缺少值"，就是此处应该有一个值，但是还没有定义。典型用法是：
  - 变量被声明了，但没有赋值时，就等于undefined。
  - 调用函数时，应该提供的参数没有提供，该参数等于undefined。
  - 对象没有赋值的属性，该属性的值为undefined。
  - 函数没有返回值时，默认返回undefined。

19、函数的节流和防抖？

- 节流

> 节流是指当一个事件触发的时候,为防止事件的连续频繁触发,设置定时器,达到一种一段事件内只触发一次的效果,在当前事件内不会再次触发,当前事件结束以后,再次触发才有效.

```
function thro(cb,wait){
  let timeOut
  return function(){
    if(timeOut) return
    timeOut = setTimeout(function(){
      cb()
      clearTimeout(timeOut)
      timeOut = null
    },wait)
  }
}
```

- 防抖

> 防抖是指当一个事件触发的时候， 为防止频繁触发事件， 设置定时器,以达到一种 频繁触发期间不处理， 只有当最后一次连续触发结束以后才处理

```
function debounce(cb,wait){
    let timer
  return function(){
    clearTimeout(timer)
    timer = setTimeout(()=>cb(),wait)
    }
}
```

20、什么是 Promise？

Promise 是异步编程的一种解决方案：从语法上讲，promise是一个对象，从它可以获取异步操作的消息；

从本意上讲，它是承诺，承诺它过一段时间会给你一个结果。

promise有三种状态： pending(等待态)，fulfiled(成功态)，rejected(失败态)；状态一旦改变，就不会再变。创造promise实例后，它会立即执行

promise是用来解决两个问题的：

回调地狱，代码难以维护， 常常第一个的函数的输出是第二个函数的输入这种现象

promise可以支持多个并发的请求，获取并发请求中的数据

这个promise可以解决异步的问题，本身不能说promise是异步的

21、普通函数与箭头函数的区别？

普通函数和箭头函数的区别：

- 1.箭头函数没有prototype(原型),箭头函数没有自己的this,继承的是外层代码块的this。
- 2.不可以当做构造函数，也就是说不可以使用new命令，否则会报错的。
- 3.不可以使用arguments对象，该对象在函数体内不存在。如果要用，可以用 rest 参数代替。
- 4.不可以使用yield命令，因此箭头函数不能用作 Generator（生成器） 函数。
- 5.因为没有this,所以不能使用call、bind、apply来改变this的指向。

22、设计模式有哪些, 分别说一说？

共23种设计模式，介绍其中6种应用较为广泛的模式。

- 发布订阅模式:
  这种设计模式可以大大降低程序模块之间的耦合度，便于更加灵活的扩展和维护。
- 中介者模式：
  观察者模式通过维护一堆列表来管理对象间的多对多关系，中介者模式通过统一接口来维护一对多关系，且通信者之间不需要知道彼此之间的关系，只需要约定好API即可。
- 代理模式：
  为其他对象提供一种代理以控制对这个对象的访问。
  代理模式使得代理对象控制具体对象的引用。代理几乎可以是任何对象：文件，资源，内存中的对象，或者是一些难以复制的东西。
- 单例模式：
  保证一个类只有一个实例，并提供一个访问它的全局访问点（调用一个类，任何时候返回的都是同一个实例）。
- 工厂模式：
  工厂模式定义一个用于创建对象的接口，这个接口由子类决定实例化哪一个类。该模式使一个类的实例化延迟到了子类。而子类可以重写接口方法以便创建的时候指定自己的对象类型
- 装饰者模式：装饰者(decorator)模式能够在不改变对象自身的基础上，在程序运行期间给对像动态的添加职责（方法或属性）。与继承相比，装饰者是一种更轻便灵活的做法。

23、Promsie 和 async/await 的区别和使用？

区别：

- 1）函数前面多了一个async关键字。await关键字只能用在async定义的函数内。async函数会隐式地返回一个promise，该promise的reosolve值就是函数return的值。
- 2）第1点暗示我们不能在 外层代码中使用await，因为不在async函数内。使用：
  - 1.async和await是配对使用的，await存在于async的内部。否则会报错 。
  - 2.await表示在这里等待一个promise返回，再接下来执行。
  - 3.await后面跟着的应该是一个promise对象，（也可以不是，如果不是接下来也没什么意义了…）

24、谈一谈垃圾回收机制？

垃圾回收是动态存储管理技术，会自动地释放“垃圾”(不再被程序引用的对象)，按照特定的垃圾收集算法来实现资源自动回收的功能。
回收的两种机制

- 1.标记清除（make-and-sweep）
- 2.引用计数 垃圾回收器会按照固定的时间间隔周期性的执行。

25、数组去重？

- 第一种：

```
for(var i=0;i<arr.length;i++){
for(var j=i+1;j<arr.length;){
if(arr[i]===arr[j]) arr.splice(j,1);
else j++; // 核心
}
}
```

- 第二种：

```
var arr1=[];
xt: for(var i=0;i<arr.length;i++){
for(var j=0;j<arr1.length;j++){
if(arr1[j]===arr[i]) continue xt;
}
arr1.push(arr[i]);
}
```

- 第三种：

```
var arr1=[];
for(var i=0;i<arr.length;i++){
if(arr1.indexOf(arr[i])<0) arr1.push(arr[i])
}
```

- 第四种：

```
var arr1=[];
for(var i=0;i<arr.length;i++){
if(!(~arr1.indexOf(arr[i]))) arr1.push(arr[i])
}
```

- 第五种：

```
var arr1=[];
for(var i=0;i<arr.length;i++){
if(!arr1.includes(arr[i])) arr1.push(arr[i])
}
```

- 第六种：

```
arr=[1,2,3,1,2,3,1,2,3]
new Set(arr);
```

26、判断对象为空？

- 第一种

```
使用JSON.stringify()将对象转换为json字符串；
JSON.stringify(obj) === '{}'
```

- 第二种

```
使用for...in循环遍历对象除Symbol以外的所有可枚举属性，当对象有属性存在返回false， 否则返回
true。
const obj = {}
function isObjectEmpty(obj){
for(var key in obj){
return false
}
return true
}
console.log(isObjectEmpty(obj))
```

- 第三种

```
Object.getOwnPropertyNames() 方法会返回该对象所有可枚举和不可枚举属性的属性名（不含Symbol
属性）组成的数组。然后再通过判断返回的数组长度是否为零，如果为零的话就是空对象。
Object.getOwnPropertyNames(obj).length === 0
```

- 第四种

```
Object.keys() 是 ES5 新增的一个对象方法，该方法返回一个数组，包含指定对象自有的可枚举属性（不
含继承的和Symbol属性）。用此方法只需要判断返回的数组长度是否为零，如果为零的话就是空对象。
```

27、如何用一次循环找到数组中两个最大的值？

```
var arr=[1,4,10,11,11,2,5,7,2,3,4];
  var [max,second]=arr[0]>arr[1] ? [arr[0],arr[1]] : [arr[1],arr[0]];
  for(var i=2;i<arr.length;i++){
    if(arr[i]>max){
      second=max;
      max=arr[i];
    }else if(arr[i]<=max && arr[i]>second){
      second=arr[i];
    }
  }
```

28、new 一个对象的过程？

- 1.开辟一个堆内存，创建一个空对象
- 2.执行构造函数，对这个空对象进行构造
- 3.给这个空对象添加**proto**属性

29、箭头函数为什么不能用 new？

因为箭头函数没有prototype也没有自己的this指向并且不可以使用arguments。

30、如何实现数组的复制？

- for循环逐一复制；

```
  var arr1=[];
  for(var i=0;i<arr.length;i++){
    if(i in arr) arr1[i]=arr[i]
  }
```

- ...方式

```
   var arr1=[...arr];
```

- slice方法

```
  var arr1=arr.slice();
```

- concat方法

```
  var arr1=arr.concat();
```

- map方法

```
var arr1=arr.map(item=>item);
```

- reduce

```
var arr1=arr.reduce((v,t)=>v.push(t),[])
```

31、http 的理解？

```
HTTP 协议是超文本传输协议，是客户端浏览器或其他程序“请求”与 Web 服务器响应之间的应用层通信协议。

HTTPS主要是由HTTP+SSL构建的可进行加密传输、身份认证的一种安全通信通道。
```

32、http 和 https 的区别？

- 1、https协议需要到ca申请证书，一般免费证书较少，因而需要一定费用。
- 2、http是超文本传输协议，信息是明文传输，https则是具有安全性的ssl加密传输协议。
- 3、http和https使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443。
- 4、http的连接很简单，是无状态的；HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，比http协议安全。

33、git 的常用指令有哪些？

```
git branch 分支查看

git branch branch_1 增加分支

git checkout branch 分支切换

git merge branch_1 合并分支(合并前要切换当前分支至master)

git branch -d branch_1 删除分支

git remote 查看当前仓库管理的远程仓库信息

git remote show origin 查看指定的远程仓库的详细信息

git push --set-upstream origin branch_1 第一次将本地分支推到远程仓库

git push <远程主机名> <本地分支名>:<远程分支名> 将本地分支推到远程分支

git pull <远程主机名> <远程分支>:<本地分支> 将远程分支拉到本地分支

git branch -d branch_0 删除本地合并后分支

git brench -D branch_0 删除本地未合并分支

it push origin --delete branch_0 删除远程分支

git restore [filename] 进行清除工作区的改变

git tag 查看标签

git tag v1.0.0 打标签

git push origin v1.0.0 将tag同步到远程服务器
```

34、平时是使用 git 指令还是图形化工具？

repository：git库相关操作，基本意思就是字面意思。

- 1）资源管理器中浏览该Git库工作空间文件，省去查找路径不断点击鼠标的操作。
- 2）启动Git bash工具（命令行工具）。
- 3）查看当前分支文件状态，不包括未提交的信息。
- 4）查看某个分支的文件（弹出框中可选择需要查看的版本、分支或标签），跟上一条差不多，用的比较少，可能是没有这方面的额需求。
- 5）可视化当前分支历史、可视化所有分支历史：弹出分支操作历史，也就是gitk工具，放到gitk工具中介绍。
- edit：用于操作commit时操作信息输入，只能操作文字输入部分，你没有看错。常用的快捷键大家都知道，何必要单独做成基本没啥用的。本来以为对变更的文件进行批量操作、本来以为可以对未版本跟踪的文件批量删除、本来、、、，都说了是本来。
- Branch：新建分支（需要选择其实版本，可以根据版本号、其他分支或标签来选择）、检出分支（觉得切换分支更合适）、重命名分支、删除分支、当前分支Reset操作（会丢弃所有未提交的变更，包括工作区和索引区，当然了，有弹出框提示危险操作）。

35、Promsie.all() 使用过吗，它是怎么使用的？

```
    promise.all()用于一个异步操作需要在几个异步操作完成后再进行时使用。

    promise.all()接受一个promise对象组成的数组参数，返回promise对象。

    当数组中所有promise都完成了，就执行当前promise对象的then方法，如果数组中有一个promise执行失败了，就执行当前promise对象的catch方法。
```

36、什么是三次握手和四次挥手？

三次握手是网络客户端跟网络服务器之间建立连接，并进行通信的过程。相当于客户端和服务器之间你来我往的3个步骤。

- 第一次握手是建立连接，客户端发送连接请求报文，并传送规定的数据包；
- 第二次握手是服务器端表示接收到连接请求报文，并回传规定的数据包；
- 第三次握手是客户端接收到服务器回传的数据包后，给服务器端再次发送数据包。这样就完成了客户端跟服务器的连接和数据传送。

四次挥手表示当前这次连接请求已经结束，要断开这次连接。

- 第一次挥手是客户端对服务器发起断开请求，
- 第二次握手是服务器表示收到这次断开请求，
- 第三次握手是服务器表示已经断开连接
- 第四次握手是客户端断开连接。

37、for in 和 for of 循环的区别？

```
    `for in` 用于遍历对象的键(`key`)，`for in`会遍历所有自身的和原型链上的可枚举属性。如果是数组，for in会将数组的索引(index)当做对象的key来遍历，其他的object也是一样的。
```

`for of`是`es6`引入的语法，用于遍历 所有迭代器iterator，其中包括`HTMLCollection`,`NodeList`,`Array`, `Map`, `Set`, `String`, `TypedArray`, `arguments`等对象的值(`item`)。

38、 async/await 怎么抛出错误异常？

如果可能出错的代码比较少的时候可以使用try/catch结构来了处理，如果可能出错的代码比较多的时候，可以利用async函数返回一个promise对象的原理来处理，给async修饰的函数调用后返回的promise对象，调用catch方法来处理异常。

39、 函数式编程和命令式编程的区别？

- 命令式编程(过程式编程)：

专注于"如何去做"，这样不管"做什么"，都会按照你的命令去做。解决某一问题的具体算法实现。

- 函数式编程：把运算过程尽量写成一系列嵌套的函数调用。

函数式编程强调没有"副作用"，意味着函数要保持独立，所有功能就是返回一个新的值，没有其他行为，尤其是不得修改外部变量的值。

所谓"副作用"，指的是函数内部与外部交互（最典型的情况，就是修改全局变量的值），产生运算以外的其他结果。

40、 http 常见的响应状态码？

100--客户必须继续发出请求
101--客户要求服务器根据请求转换HTTP协议版本
200--交易成功
201--提示知道新文件的URL
202--接受和处理、但处理未完成
203--返回信息不确定或不完整
204--请求收到，但返回信息为空
205--服务器完成了请求，用户代理必须复位当前已经浏览过的文件
206--服务器已经完成了部分用户的GET请求
300--请求的资源可在多处得到
301--删除请求数据
302--在其他地址发现了请求数据
303--建议客户访问其他URL或访问方式
304--客户端已经执行了GET，但文件未变化
305--请求的资源必须从服务器指定的地址得到
306--前一版本HTTP中使用的代码，现行版本中不再使用
307--申明请求的资源临时性删除
400--错误请求，如语法错误
401--请求授权失败
402--保留有效ChargeTo头响应
403--请求不允许
404--没有发现文件、查询或URl
405--用户在Request-Line字段定义的方法不允许
406--根据用户发送的Accept拖，请求资源不可访问
407--类似401，用户必须首先在代理服务器上得到授权

```
    408--客户端没有在用户指定的饿时间内完成请求
    409--对当前资源状态，请求不能完成
    410--服务器上不再有此资源且无进一步的参考地址
    411--服务器拒绝用户定义的Content-Length属性请求
    412--一个或多个请求头字段在当前请求中错误
    413--请求的资源大于服务器允许的大小
    414--请求的资源URL长于服务器允许的长度
    415--请求资源不支持请求项目格式
    416--请求中包含Range请求头字段，在当前请求资源范围内没有range指示值，请求也不包含If-Range请求头字段
    417--服务器不满足请求Expect头字段指定的期望值，如果是代理服务器，可能是下一级服务器不能满足请求
    500--服务器产生内部错误
    501--服务器不支持请求的函数
    502--服务器暂时不可用，有时是为了防止发生系统过载
    503--服务器过载或暂停维修
    504--关口过载，服务器使用另一个关口或服务来响应用户，等待时间设定值较长
    505--服务器不支持或拒绝支请求头中指定的HTTP版本
```

41、 什么是事件流以及事件流的传播机制？

事件触发后，从开始找目标元素，然后执行目标元素的事件，再到离开目标元素的整个过程称之为事件流。

W3C标准浏览器事件流的传播分为3个阶段：捕获阶段、目标阶段、冒泡阶段

- 捕获阶段指找目标元素的过程，这个找的过程，是从最大的document对象到html，再到body，。。。直到目标元素。
- 找到目标元素后，调用执行他绑定事件时对应的处理函数，这个过程被称之为目标阶段。
- 当目标元素的事件执行结束后，再从目标元素，到他的父元素。。。body、html再到document的过程，是冒泡阶段。

42、模块化语法？commonJS AMD CMD ES6 Module

- commonJS是nodejs自带的一种模块化语法，将一个文件看做是一个模块，可以将文件中导出的时候，被另一个文件导入使用。导出使用：`module.exports`导出。导入使用：`require`函数导入。
- AMD是社区开发的模块化语法，需要依赖`require.js`实现，分为定义模块，导出数据和导入模块，使用数据。AMD语法的导入是依赖前置的，也就是说，需要用到的文件需要在第一次打开页面全部加载完成，造成的后果就是首屏加载很慢，后续操作会很流畅。
- CMD是玉伯开发的模块化语法，需要依赖`sea.js`实现，也分为模块定义导出，和模块导入使用数据。CMD语法可以依赖前置，也可以按需导入，缓解了AMD语法的依赖前置。
- ES6的模块化语法，类似于commonJS的语法，分为数据导出和数据导入，导入导出更加灵活。

43、 什么是懒加载和预加载？

- 懒加载：懒加载也叫延迟加载，延迟加载网络资源或符合某些条件时才加载资源。常见的就是图片延时加载。
  懒加载的意义：懒加载的主要目的是作为服务器前端的优化，减少请求数或延迟请求数。
  懒惰实现方式：

    - 1.第一种是纯粹的延迟加载，使用setTimeOut或setInterval进行加载延迟.
    - 2.第二种是条件加载，符合某些条件，或触发了某些事件才开始异步下载。
    - 3.第三种是可视区加载，即仅加载用户可以看到的区域，这个主要由监控滚动条来实现，一般会在距用户看到某图片前一定距离遍开始加载，这样能保证用户拉下时正好能看到图片。
- 预加载：提前加载图片，当用户需要查看时可直接从本地缓存中渲染。

两者的行为是相反的，一个是提前加载，一个是迟缓甚至不加载。懒加载对服务器前端有一定的缓解压力作用，预加载则会增加服务器前端压力。预加载应用如广告弹窗等。

44、 token 一般存放在哪里？为什么不存放在 cookie 内？

token一般放在本地存储中。token的存在本身只关心请求的安全性，而不关心token本身的安全，因为token是服务器端生成的，可以理解为一种加密技术。但如果存在cookie内的话，浏览器的请求默认会自动在请求头中携带cookie，所以容易受到csrf攻击。

45、 less 和 sass 的区别？

- 编译环境不一样，sass是服务器端处理的，可以用Ruby、node-sass来编译；less需要引入less.js来处理输出，也可以使用工具在服务器端处理成css，也有在线编译的。
- 变量定义符不一样，less用的是@，而sass用$。
- sass支持分支语句，less不支持

44、浏览器的同源策略机制？

同源策略，又称SOP，全称Same Origin Policy，是浏览器最基本的安全功能。站在浏览器的较短看网页，如果网络上的接口可以不受限制、无需授权随意被人调用，那将是一个非常严重的混乱场景。浏览器为了安全有序，内部实现了同源策略。

同源策略，指的是浏览器限制当前网页只能访问同源的接口资源。

所谓同源，指当前页面和请求的接口，两方必须是同协议、且同域名、且同端口。只要有一个不相同，则会受到浏览器额约束，不允许请求。

但当一个项目变的很大的时候，将所有内容放在一个网站或一个服务器中会让网站变的臃肿且性能低下，所以，在一些场景中，我们需要跨过同源策略，请求到不同源的接口资源，这种场景叫跨域。

跨域大致有3种方案：

- jsonp

  这种方式是利用浏览器不限制某些标签发送跨域请求，例如link、img、iframe、script。通常请求请求回来的资源要在js中进行处理，所以jsonp跨域是利用script标签进行发送，且这种请求方式只能是get请求。

- cors

  这种方式是让接口资源方面进行授权，授权允许访问。在接口资源处添加响应头即可通过浏览器的同源策略，响应头具体的键值对如下：

  ```
  {Access-Control-Allow-Origin: '*'}
  ```

- proxy

  这种方式属于找外援的一种方式，浏览器只能限制当前正在打开的web页面发送请求，但无法限制服务器端请求接口资源。所以我们可以将请求发送到自己服务器，然后自己服务器去请求目标接口资源，最后自己服务器将接口资源返回给当前页面，类似于找外援代替自己请求目标接口资源。

  这种方式通常要对服务器进行代理配置，需要对apache服务器、nginx服务器、nodejs服务器进行配置。

45、 浏览器的缓存有哪些？什么时候使用强制缓存？什么时候使用协商缓存？

当我们访问同一个页面时，请求资源、数据都是需要一定的耗时，如果可以将一些资源缓存下来，那么从第二次访问开始，就可以减少加载时间，提高用户体验，也能减轻服务器的压力。

浏览器缓存分为强缓存和协商缓存，当存在缓存时，客户端第一次向服务器请求数据时，客户端会缓存到内存或者硬盘当中，当第二次获取相同的资源，强缓存和协商缓存的应对方式有所不同。

强缓存：当客户端第二次向服务器请求相同的资源时，不会向服务器发送请求，而是直接从内存/硬盘中间读取。缓存由服务器的响应头里 `cache-control` 和 `expires` 两个字段决定

协商缓存：当客户端第二次向服务器请求相同的资源时，先向服务器发送请求"询问"该请求的文件缓存在ben'd与服务器相比是否更改，如果更改，则更新文件，如果没有就从内存/硬盘中读取。协商缓存由 `last-modified` 和 `etag`两个字段决定

46、 数组方法 forEach 和 map 的区别？

`forEach`和`map`都是循环遍历数组中的每一项。`forEach()` 和 `map()` 里面每一次执行匿名函数都支持3个参数：数组中的当前项`item`,当前项的索引`index`,原始数组`input`。匿名函数中的`this`都是指`Window`。只能遍历数组。

他们的区别是：`forEach`没有返回值，但`map`中要有返回值，返回处理后的所有新元素组成的数组。

47、 什么是函数作用域？什么是作用域链？

作用域就是在代码执行过程中，形成一个独立的空间，让空间内的变量不会邪泄露在空间外，也让独立空间内的变量函数在独立空间内运行，而不会影响到外部的环境。

作用域分为全局作用域和局部作用域，也就是本来有一个巨大的空间，空间内定义的函数内部，就形成了一个独立的小空间，全局作用域是最大的作用域。

但是当独立空间内的数据不能满足需求时，是可以从外部获取数据的，也就是说这样的独立空间之间是可以有层级关系的，外部的空间不可以从内部的空间获取数据，但内部的空间可以。当子级空间在父级空间中获取数据的时，父级空间没有的话，父级空间也会到他的父级空间中查找数据，这样形成的链式结构叫作用域链。

当将一个变量当做值使用时，会先在当前作用域中查找这个变量的定义和数据，如果没有定义的话，就会去父级作用域中查找，如果父级作用域中有的话就使用这个值，如果父级作用域中也没有的话，就通过父级作用域查找他的父级作用域，直到找到最大的作用域-全局，如果全局也没有就报错。

当将一个变量当做数据容器存储，也就是给变量赋值的时候，也要先在自己作用域中查找变量的定义，如果没有就在上一级作用域中查找，直到全局，如果全局作用域中也没有这个变量的定义，就在全局定义这个变量并赋值。

48、 ES6 中 Set 和 Map 的原理？

Set 是无重复值的有序列表。根据 `Object.is()`方法来判断其中的值不相等，以保证无重复。 Set 会自动移除重复的值，因此你可以使用它来过滤数组中的重复值并返回结果。 Set并不是数组的子类型，所以你无法随机访问其中的值。但你可以使用`has()` 方法来判断某个值是否存在于 Set 中，或通过 `size` 属性来查看其中有多少个值。 Set 类型还拥有`forEach()`方法，用于处理每个值

Map 是有序的键值对，其中的键允许是任何类型。与 Set 相似，通过调用 `Object.is()`方法来判断重复的键，这意味着能将数值 5 与字符串 "5" 作为两个相对独立的键。使用`set()` 方法能将任何类型的值关联到某个键上，并且该值此后能用 `get()` 方法提取出来。Map 也拥有一个 `size` 属性与一个 `forEach()` 方法，让项目访问更容易。

49、 0.1 + 0.2 为什么不等于 0.3, 在项目中遇到要怎么处理？

计算机内部存储数据使用2进制存储，两个数字进行的数学运算，首先是将这两个数字以2进制形式，存储在计算机内部，然后在计算机内部使用两个2进制数字进行计算，最后将计算结果的2进制数字转为10进制展示出来。

由于10进制的小数在转2进制的时候，规则是小数部分乘以2，判断是否得到一个整数，如果得到整数，转换完成；如果没有得到整数，则继续乘以2判断。所以，0.1和0.2在转换2进制的时候，其实是一个无限死循环，也就是一直乘以2没有得到整数的时候，但计算机内部对于无线死循环的数据，会根据一个标准保留52位。也就是说，计算机内部在存储0.1和0.2的时候，本来就不精准，两个不精准的小数在计算后，距离精准的结果是有一定误差的。

项目中碰到这种情况，有3种处理方法：

- 将小数乘以10的倍数，转为整数，然后计算，计算完成后，再缩小10的倍数，例如：

```
var result = ((0.1 * 10) + (0.2 * 10)) / 10
// result === 0.3
```

- 使用数字的toFixed方法，强制保留小数点后多少位，例：

```
var result = (0.1 + 0.2).toFixed(2)
// result === 0.30
```

- 自定义数字运算方法，当需要进行数学运算的时候，不直接进行，调用自定义的方法进行，例：（加法封装）

```
function add(...args){
        var num = args.find(item => {
            if(item != 0 && !item){
               throw new Error("数学运算要使用数字")
            }
        })
        var arr = args.map(item => {
            var index = (item+'').indexOf('.')
            if(index >= 0){
                return (item+'').split('.')[1].length
            }
        })
        arr = arr.filter(item => item)
        if(arr.length){
```

```
            var max = Math.max(...arr)
            var data = args.map(item => item * Math.pow(10, max))
            var data.reduce((a, b) => a + b) / Math.pow(10, max)
        }else{
            var data = args
            return data.reduce((a, b) => a + b)
        }
    }
    // 调用使用:
    var num1 = add(0.1, 0.2)
    console.log(num1); // 0.3

    var num2 = add(1, 2)
    console.log(num2); // 3

    var num3 = add(1, 2.1)
    console.log(num3); // 3.1
```

50、 什么是模块化思想？

就是JS中将不同功能的代码封装在不同的文件中，再互相引用时不会发生命名冲突的一种思想，大多数情况下，一个文件就是一个模块

模块化的实现，有多种方案：

- CommonJS：

  `CommonJS` 是 `nodejs` 中使用的模块化规范
  在 `nodejs` 应用中每个文件就是一个模块，拥有自己的作用域，文件中的变量、函数都是私有的，与其他文件相隔离。模块导出：`module.exports=数据`，模块导入：`require('模块文件路径')`

- ES6的模块化：

  模块功能主要由两个命令构成：`export` 和 `import`。`export` 命令用于规定模块的对外接口，`import` 命令用于输入其他模块提供的功能。

  一个模块就是一个独立的文件。该文件内部的所有变量，外部无法获取。如果你希望外部能够读取模块内部的某个变量，就必须使用 `export` 关键字输出该变量。下面是一个 JS 文件，里面使用 `export` 命令输出变量。

- AMD （Asynchronous Module Definition）：

  特点: 提倡依赖前置，在定义模块的时候就要声明其依赖的模块：导入模块 `require([module],callback);` 定义模块：`define('模块名称', 函数)`。

- CMD (Common Module Definition)：

  CMD规范是国内SeaJS的推广过程中产生的。提倡就近依赖（按需加载），在用到某个模块的时候再去require。定义模块：`define(function (require, exports, module) {})`，使用模块：`seajs.use()`

51、 说说怎么用js 写无缝轮播图

将所有需要轮播的内容动态复制一份，放在原本的容器中，加定时器让整个容器中的内容滚动轮播，当内容轮播到left值为-原本的内容宽度时，快速将内容切换到left值为0的状态。

52、 JS 如何实现多线程？

我们都知道JS是一种单线程语言，即使是一些异步的事件也是在JS的主线程上运行的（具体是怎么运行的，可以看我另一篇博客JS代码运行机制）。像setTimeout、ajax的异步请求，或者是dom元素的一些事件，都是在JS主线程执行的，这些操作并没有在浏览器中开辟新的线程去执行，而是当这些异步操作被操作时或者是被触发时才进入事件队列，然后在JS主线程中开始运行。

首先说一下浏览器的线程，浏览器中主要的线程包括，UI渲染线程，JS主线程，GUI事件触发线程，http请求线程。

JS作为脚本语言，它的主要用途是与用户互动，以及操作DOM。这决定了它只能是单线程，否则会带来很复杂的同步问题。（这里这些问题我们不做研究）

但是单线程的语言，有一个很致命的确定。如果说一个脚本语言在执行时，其中某一块的功能在执行时耗费了大量的时间，那么就会造成阻塞。这样的项目，用户体验是非常差的，所以这种现象在项目的开发过程中是不允许存在的。

其实JS为我们提供了一个Worker的类，它的作用就是为了解决这种阻塞的现象。当我们使用这个类的时候，它就会向浏览器申请一个新的线程。这个线程就用来单独执行一个js文件。

```
var worker = new Worker(js文件路径);
```

那么这个语句就会申请一个线程用来执行这个js文件。这样也就实现了js的多线程。

53、 闭包的使用场景？

一个函数被当作值返回时，也就相当于返回了一个通道，这个通道可以访问这个函数词法作用域中的变量，即函数所需要的数据结构保存了下来，数据结构中的值在外层函数执行时创建，外层函数执行完毕时理因销毁，但由于内部函数作为值返回出去，这些值得以保存下来。而且无法直接访问，必须通过返回的函数。这也就是私有性。

本来执行过程和词法作用域是封闭的，这种返回的函数就好比是一个虫洞，开了挂。

闭包的形成很简单，在执行过程完毕后，返回函数，或者将函数得以保留下来，即形成闭包。

- 防抖：

```
function debounce(fn, interval) {
    let timer = null; // 定时器
    return function() {
        // 清除上一次的定时器
        clearTimeout(timer);
        // 拿到当前的函数作用域
        let _this = this;
        // 拿到当前函数的参数数组
        let args = Array.prototype.slice.call(arguments, 0);
        // 开启倒计时定时器
        timer = setTimeout(function() {
            // 通过apply传递当前函数this，以及参数
            fn.apply(_this, args);
            // 默认300ms执行
        }, interval || 300)
```

```
        }
    }
```

- 节流：

```
function throttle(fn, interval) {
    let timer = null; // 定时器
    let firstTime = true; // 判断是否是第一次执行
    // 利用闭包
    return function() {
        // 拿到函数的参数数组
        let args = Array.prototype.slice.call(arguments, 0);
        // 拿到当前的函数作用域
        let _this = this;
        // 如果是第一次执行的话，需要立即执行该函数
        if(firstTime) {
            // 通过apply，绑定当前函数的作用域以及传递参数
            fn.apply(_this, args);
            // 修改标识为null，释放内存
            firstTime = null;
        }
        // 如果当前有正在等待执行的函数则直接返回
        if(timer) return;
        // 开启一个倒计时定时器
        timer = setTimeout(function() {
            // 通过apply，绑定当前函数的作用域以及传递参数
            fn.apply(_this, args);
            // 清除之前的定时器
            timer = null;
            // 默认300ms执行一次
        }, interval || 300)
    }
}
```

- 迭代器：

```
var arr =['aa','bb','cc'];
function incre(arr){
    var i=0;
    return function(){
        //这个函数每次被执行都返回数组arr中 i下标对应的元素
         return arr[i++] || '数组值已经遍历完';
    }
}
var next = incre(arr);
console.log(next());//aa
console.log(next());//bb
console.log(next());//cc
console.log(next());//数组值已经遍历完
```

- 缓存：

```
var fn=(function(){
        var cache={};//缓存对象
        var calc=function(arr){///计算函数
            var sum=0;
            //求和
            for(var i=0;i<arr.length;i++){
```

```
                sum+=arr[i];
            }
            return sum;
        }

        return function(){
            var args = Array.prototype.slice.call(arguments,0);//arguments转换成数组
            var key=args.join(",");//将args用逗号连接成字符串
            var result , tSum = cache[key];
            if(tSum){//如果缓存有
                console.log('从缓存中取：',cache)//打印方便查看
                result = tSum;
            }else{
                //重新计算，并存入缓存同时赋值给result
                result = cache[key]=calc(args);
                console.log('存入缓存：',cache)//打印方便查看
            }
            return result;
        }
     })();
    fn(1,2,3,4,5);
    fn(1,2,3,4,5);
    fn(1,2,3,4,5,6);
    fn(1,2,3,4,5,8);
    fn(1,2,3,4,5,6);
```

- getter和setter：

```
function fn(){
        var name='hello'
        setName=function(n){
            name = n;
        }
        getName=function(){
            return name;
        }

        //将setName，getName作为对象的属性返回
        return {
            setName:setName,
            getName:getName
        }
    }
    var fn1 = fn();//返回对象，属性setName和getName是两个函数
    console.log(fn1.getName());//getter
        fn1.setName('world');//setter修改闭包里面的name
    console.log(fn1.getName());//getter
```

- 柯里化：

```
function curryingCheck(reg) {
    return function(txt) {
        return reg.test(txt)
    }
}

var hasNumber = curryingCheck(/\d+/g)
var hasLetter = curryingCheck(/[a-z]+/g)

hasNumber('test1')      // true
hasNumber('testtest')   // false
hasLetter('21212')      // false
```

- 循环中绑定事件或执行异步代码：

```
var p1 = "ss";
var p2 = "jj";
function testSetTime(para1,para2){
    return (function(){
        console.log(para1 + "-" + para2);
    })
}
var test = testSetTime(p1, p2);
setTimeout(test, 1000);
setTimeout(function(){
    console.log(p1 + "-" + p2)
},1000)
```

- 单例模式：

```
var Singleton = (function () {
    var instance;

    function createInstance() {
        return new Object("I am the instance");
    }

    return {
        getInstance: function () {
            if (!instance) {
                instance = createInstance();
            }
            return instance;
        }
    };
})();
```

54、 常见的兼容问题有哪些？

- 获取标签节点：

  document.getElementsByClassName('类名')在低版本 `ie` 中不兼容。解决方法是使用其他方式获取：

```
document.getElementById('id名')
document.getElementsByTagName('标签名')
document.getElementsByName('name属性值')
document.querySelector('css选择器')
document.querySelectorAll('css选择器')
```

* 获取卷去的高度

```
// 当有文档声明的时候
document.documentElement.scrollTop
document.documentElement.srollLeft
// 没有文档声明的时候
document.body.scrollTop
document.body.scrollLeft
```

* 解决办法使用兼容写法：

```
// 获取
var t = document.documentElement.scrollTop || document.body.scrollTop
var l = document.documentElement.srollLeft || document.body.scrollLeft
// 设置
document.documentElement.scrollTop = document.body.scrollTop = 数值
document.documentElement.srollLeft = document.body.scrollLeft = 数值
```

- 获取样式

```
// W3C标准浏览器
window.getComputedStyle(元素)
// 低版本IE中
元素.currentStyle
```

- 使用函数封装的方式兼容：

```
function getStyle(ele,attr){
    if(window.getComputedStyle){
       return getComputedStyle(ele)[attr]
    }else{
        return ele.currentStyle[attr]
    }
}
```

- 事件侦听器

```
// W3C浏览器
ele.addEventListener(事件类型,函数)
// 低版本Ie
ele.attachEvent('on事件类型',函数)
```

- 使用函数封装的方式解决：

```
function bindEvent(ele,type,handler){
    if(ele.addEventListener){
        ele.addEventListener(type,handler)
    }else if(ele.attachEvent){
        ele.attachEvent('on'+type,handler)
    }else{
        ele['on'+type] = handler
    }
}
```

- 事件解绑

```
// W3C浏览器
ele.removeEventListener(事件类型,函数)
// 低版本Ie
ele.detachEvent('on事件类型',函数)
```

- 使用函数封装的方式解决：

```
function unBind(ele,type,handler){
    if(ele.removeEventListener){
        ele.removeEventListener(type,handler)
    }else if(ele.detachEvent){
        ele.detachEvent('on'+type,handler)
    }else{
        ele['on'+type] = null
    }
}
```

- 事件对象的获取

```
// W3C浏览器
元素.on事件类型 = function(e){}
元素.addEventListener(事件类型,fn)
function fn(e){

}
// 在低版本IE中
元素.on事件类型 = function(){ window.event }
元素.addEventListener(事件类型,fn)
function fn(){
    window.event
}
```

- 使用短路运算符解决：

```
元素.on事件类型 = function(e){
    var e = e || window.event
}
元素.addEventListener(事件类型,fn)
function fn(e){
    var e = e || window.event
}
```

- 阻止默认行为

```
// W3C浏览器
元素.on事件类型 = function(e){
    e.preventDefault()
}
// 在低版本IE中
元素.on事件类型 = function(){ window.event.returnValue = false }
```

- 通过封装函数解决;

```
元素.on事件类型 = function(e){
    var e = e || window.event
    e.preventDefault?e.preventDefault():e.returnValue=false
}
```

- 阻止事件冒泡

```
// W3C浏览器
元素.on事件类型 = function(e){
    e.stopPropagation()
}
// 在低版本IE中
元素.on事件类型 = function(){ window.event.cancelBubble = true }
```

- 通过函数封装解决:

```
元素.on事件类型 = function(e){
    var e = e || window.event
    e.stopPropagation?e.stopPropagation():e.cancelBubble=true
}
```

- 获取精准的目标元素

```
// W3C浏览器
元素.on事件类型 = function(e){
    e.target
}
// 在低版本IE中
元素.on事件类型 = function(){ window.event.srcElement }
```

- 通过短路运算符解决:

```
元素.on事件类型 = function(e){
    var e = e || window.event
    var target = e.target || e.srcElement;
}
```

- 获取键盘码

```
// W3C浏览器
元素.on事件类型 = function(e){
    e.keyCode
}
// 在低版本火狐中
元素.on事件类型 = function(e){
  e.which
}
```

- 通过短路运算符解决：

```
元素.on事件类型 = function(e){
    var e = e || window.event
    var keycode = e.keyCode || e.which;
}
```

55、 在 JS 中如何阻止事件冒泡？

使用事件对象阻止事件冒泡，以前的w3c浏览器中，使用事件对象的方法阻止：

```
事件对象.stopPropagation()
```

在ie低版本浏览器中，使用事件对象的属性阻止：

```
事件对象.cancelBubble = true
```

现在的w3c浏览器也支持ie低版本浏览器中的写法，所以以前在阻止事件冒泡的时候，需要考虑兼容写法，现在就不需要了，直接用ie低版本浏览器中的写法即可。

56、两个数组 var A = [1, 5, 6]; var B = [2, 6, 7]，实现一个方法，找出仅存在于A 或者 仅 存在于B中的所有数字。

```
function getDiff(arr, brr){
    // 仅存在于arr中的内容
    var onlyArr = arr.filter(item => !brr.some(v => item === v))
    // 仅存在于brr中的内容
    var onlyBrr = brr.filter(v => !arr.some(item => v === item))
    // 需要哪个就返回哪个，或者一起返回
    return {
        "仅存在于arr中的内容": onlyArr,
        "仅存在于brr中的内容": onlyBrr
    }
}
```

57、 你了解构造函数吗？class 是什么？两者有什么区别？

在es5中构造函数其实就是在定义一个类，可以实例化对象，es6中class其实是构造函数的语法糖。但还是有点区别的：

- 在class内部和class的方法内部，默认使用严格模式
- class类不存在预解析，也就是不能先调用class生成实例，再定义class类，但是构造函数可以。
- class中定义的方法默认不能被枚举，也就是不能被遍历。
- class必须使用new执行，但是构造函数没有new也可以执行。
- class中的所有方法都没有原型，也就不能被new
- class中继承可以继承静态方法，但是构造函数的继承不能。

58、是否存在a的值（a==0 && a==1）为true 的情况？

```
var value = -1
Object.defineProperty(window,'a',{
    get(){
        return value+=1;
    }
})
if(a===0&&a===1){ // true
    console.log('success')
}
```

59、 for (var i = 0; i < 5; i++) { setTimeout(function() { console.log(i); }, 1000); } 要求：输出0，1，2，3，4

首先这个面试题考察的是对于js中异步代码以及作用域的理解：

js中常见的异步代码包括定时器和ajax。js执行代码的流程是碰到同步代码就执行，碰到异步就交给浏览器的webAPI处理，当webAPI中的异步该执行时，webAPI会将需要执行的回调函数放在任务队列中，等候执行，所以，js中所有的异步代码总会在所有同步代码执行结束后，再执行任务队列中的代码。

在这个问题中，循环是同步代码，定时器是异步代码，所以整个循环都执行结束以后才会执行定时器代码。

for循环中使用var定义的变量是全局变量，定时器回调函数中输出变量的时候，根据作用域规则，先在当前作用域中变量i的定义表达式，如果没有找到，就去上一级作用域中找，此时，在局部作用域中没有找到，去上级作用域中，也就是全局找到了，全局中的i，因为循环已经执行结束了，所以i的值是5。

最终，会输出5个5。

其次考察的是对于类似问题的解决方式，间接性判断你是否有过类似情况的开发：

这个问题的解决思路就是让回调函数中输出i的时候，不要去全局中找i，因为全局的i在循环执行结束后已经变成5了，根据这个思路，有2种解决办法：

- 在异步代码外面嵌套一层函数作用域

```
    for(var i = 0;i < 5; i++){
        (function(i) {
            setTimeout(function() {
                console.log(i)
            }, 1000)
        })(i)
    }
```

原理是自调用函数会产生作用域，循环5次就会产生5个作用域，每个作用域代码在执行的时候都有形参i传递。所以每个作用域中的i都是不同的，分别是：0 1 2 3 4。当作用域中的异步代码执行的时候，自己作用域中没有i变量的定义，然后上级作用域就是自调用函数的作用域，找到了单独的i。最终可以输出：0 1 2 3 4

- 2. 将循环代码中的var换成es6的let

```
    for(let i = 0;i < 5; i++){
        setTimeout(function() {
            console.log(i)
        }, 1000)
    }
```

es6的let自带块级作用域，原理跟第一种解决思路是一样的，转成es5后，代码是一样的。

60、实现一个 add 方法 使计算结果能够满足如下预期： - add(1)(2)(3)() = 6 - add(1,2,3)(4)() = 10

```
function add (...args) {
    if(args.length === 3){
        return -(args[0] * args[1] * 2 + args[2] * 2)
    }else{
        return -args[args.length-1]
    }

}

function currying (fn) {
    let args = []
    return function _c (...newArgs) {
        if (newArgs.length) {
            args = [
                ...args,
                ...newArgs
            ]
            return _c
        } else {
            return fn.apply(this, args)
        }
    }
}
let addCurry = currying(add)

var a = addCurry(1)(2)(3)()
console.log(-a); // 10

var b = addCurry(1,2,3)(4)()
```

```
    console.log(6 - b); // 10
```

61、常见的 HTTP 请求有哪些？他们的区别是什么？

常见的有5种，分别是GET、HEAD, POST、PUT、 DELETE

- GET：它是最常见的方法，用于获取资源，常用于向服务器查询某些信息。打开网页一般都是用GET方法，因为要从 Web 服务器获取信息
- HEAD：类似于 GET请求，只不过返回的响应中没有具体的内容，用于获取报头。
- POST：向指定资源提交数据进行处理请求（例如提交表单或者上传文件），数据被包含在请求体中。POST请求可能会导致新的资源的建立和 / 或对已有资源的修改。
- PUT：从客户端向服务器传送的数据取代指定文档的内容。
- DELETE：请求服务器删除指定的页面。

最常见的HTTP请求方法是GET 和 POST。GET一般用于获取 / 查询资源信息，而POST一般用于更新资源信息。GET和POST的区别：

- GET提交的数据会放在?之后，以问号（？）分割URL 和传输数据，参数之间以＆相连
- GET提交的数据大小有限制（因为浏览器对URL的长度有限制），而POST 方法提交的数据大小没有限制。
- GET方式提交数据会带来安全问题，比如一个登录页面通过GET方式提交数据时，用户名和密码将出现在URL上，如果页面可以被缓存或者其他人可以访问这台机器，就可以从历史记录获得该用户的账号和密码。

62、 JS 的数据类型有哪些？如何判断数据类型？他们的优缺点是什么?

- typeof 用来检测数据类型的运算符

  > 检测的不管是数组还是正则都返回的是"object",所以typeof不能判断一个值是否为数组

- instanceof/constructor。检测某一个实例是否属于某一个类使用instanceof/constructor可以检测数组和正则

  > 用instanceof检测的时候,只要当前的这个类在实例的原型链上(可以通过原型链__proto__找到它),检测出来的结果都是true。
  >
  > 基本数据类型的值是不能用instanceof来检测的
  >
  > 在类的原型继承中,instanceof检测出来的结果其实是不准确的

- Object.prototype.toString.call(value) ->找到Object原型上的toString方法,让方法执行,并且让方法中的this变为value(value->就是我们要检测数据类型的值)。检测的类型比较多，也比较精准。

63、 symbol 你是怎么理解的？

`Symbol` 是 `ES6` 新推出的一种基本类型，它表示独一无二的值

它可以选择接受一个字符串作为参数或者不传，但是相同参数的两个 `Symbol` 值不相等

```
//不传参数
const s1 = Symbol();
const s2 = Symbol();
console.log(s1 === s2); // false

// 传入参数
const s3 = Symbol('debug');
const s4 = Symbol('debug');
console.log(s3 === s4); // false
```

可以通过`typeof`判断是否为`Symbol`类型

```
console.log(typeof s1); // symbol
```

Symbol.for()：用于将描述相同的`Symbol`变量指向同一个`Symbol`值

```
let a1 = Symbol.for('a');
let a2 = Symbol.for('a');
a1 === a2  // true
typeof a1  // "symbol"
typeof a2  // "symbol"

let a3= Symbol("a");
a1 === a3      // false
```

Symbol.keyFor()：用来检测该字符串参数作为名称的`Symbol`值是否已被登记，返回一个已登记的`Symbol`类型值的`key`

```
let a1 = Symbol.for("a");
Symbol.keyFor(a1);    // "a"

let a2 = Symbol("a");
Symbol.keyFor(a2);    // undefined
```

description：用来返回`Symbol`数据的描述：

```
// Symbol()定义的数据
let a = Symbol("acc");
a.description  // "acc"
Symbol.keyFor(a);  // undefined

// Symbol.for()定义的数据
let a1 = Symbol.for("acc");
a1.description  // "acc"
Symbol.keyFor(a1);  // "acc"

// 未指定描述的数据
let a2 = Symbol();
a2.description  // undefined
```

- 使用场景一：对象添加属性

```
let n = Symbol('N');
let obj = {
    name: "hello world",
    age: 11,
    [n]: 100
};
```

- 使用场景二：给对象添加私有属性

```
const speak = Symbol();
class Person {
    [speak]() {
        console.log(123)
    }
}
let person = new Person()
console.log(person[speak]())
```

64、数组常用方法有那些

数组的常用方法 这样的面试题 算是非常基础的面试题 面试官的目的 也不会只是单纯的让你背诵出 数组的所有方法

这里的关键点 是 常用 这两个字 面试官的 目的是 通过 这个问题 看你平时在项目中 对于 数组函数的应用和理解 然后判断出 你平时在项目中对于数组的应用 然后推测出你真实的技术水平

这里建议的回答方式是 通过一个 自己用的最多的数组函数方法 深入展开的说一说 在 实际项目中的应用

例如谈到 数组单元删除 数组,splice() 除了要说 函数的用法之外 还要谈到 具体的项目中 删除数组单元之后 数组坍塌的影响 以及如何处理

```
concat() 连接两个或更多的数组，并返回结果。

join() 把数组的所有元素放入一个字符串。元素通过指定的分隔符进行分隔。

pop() 删除并返回数组的最后一个元素。

shift() 删除并返回数组的第一个元素

push() 向数组的末尾添加一个或更多元素，并返回新的长度。

unshift() 向数组的开头添加一个或更多元素，并返回新的长度。

reverse() 颠倒数组中元素的顺序。

slice() 从某个已有的数组返回选定的元素

sort() 对数组的元素进行排序

splice() 删除元素，并向数组添加新元素。

toSource() 返回该对象的源代码。

toString() 把数组转换为字符串，并返回结果。

toLocaleString() 把数组转换为本地数组，并返回结果。

valueOf() 返回数组对象的原始值
```

65、 JavaScript如何存储cookie

基本语法是 document.cookie = '键名=键值;expires=时间对象;path=路径' ;

时效 如果不设定 默认是 seeion 会话时效
路径 如果不设定 默认是 当前文件所在文件夹

设定时效 要 设定一个时间对象 时间对象的时间戳 就是 时效期
要注意计算 当前时区 和 世界标砖时间的时间差

路径一般设定为根目录 也就是 '/'

66、柯理化函数

所谓的柯里化函数 指的是 把接受多个参数的函数变换成接受一个单一参数的函数 并且返回接受余下的参数而且返回结果的新函数

```
// 普通的add函数
function add(x, y) {
    return x + y
}

// Currying后
function curryingAdd(x) {
    return function (y) {
        return x + y
    }
}

add(1, 2)           // 3
curryingAdd(1)(2)   // 3
```

优点:

```
1, 参数复用

例如 一个函数 有两个参数 但是第一个参数会被反复使用 每次都需要输入 一个重复的参数
使用柯里化函数之后 只需要 输入一个参数就可以了

2, 提前确认

提前定义好一个参数 也就 决定了整个函数程序的执行方向 避免每次都执行判断比较等
```

缺点:

```
只能提前定义一个参数 如果想要提前定义多个参数 这样的语法是不支持
```

柯里化函数执行效能上的问题:

```
存取arguments对象通常要比存取命名参数要慢一点
一些老版本的浏览器在arguments.length的实现上是相当慢的
使用 函数.apply() 和 函数.call() 通常比直接调用 fn() 稍微慢点
创建大量嵌套作用域和闭包函数会带来花销，无论是在内存还是速度上
```

67、对象遍历方法

JavaScript中 对象的遍历方法

```
for...in

基本语法是 for( 变量 in 对象 ){ 循环体程序 }

这里要注意的是
1, 变量中存储的键名 通过键名获取对象中存储的键值
    因为是变量 点语法取值 不支持解析变量 要使用 对象[键名] 获取键值

2, 循环变量 定义 let 和 var 定义 执行效果是不同的
```

```
Object.keys( 对象 )

返回一个数组 是 当前对象 所有键名组成的数组
之后再循环遍历这个数组 再执行操作
```

```
Object.value( 对象 )

返回一个数组 是 当前对象 所有键值组成的数组
之后再循环遍历这个数组 再执行操作
```

68、数组扁平化

数组扁平化

所谓的数组扁平化就是将多维数组转化为一维数组一般数组扁平化,数组中存储的多维数据都是数组 不会是对象或者函数

最常用的方法 就是 数组.toString() 将数组转化为字符串
结果是 获取数组中的每一个单元的数据 组成一个字符串 使用逗号间隔
再 以逗号为间隔 将字符串 转化为数组

```
function fun1( arr ){
   let str = arr.toString();
   return str.split(',');
}
```

还可以使用 数组.some() 方法 判断数组中是不是还存在数组
在使用 展开运算符 赋值

```
function fun1(arr) {
    while (arr.some(item => Array.isArray(item))) {
        arr = [].concat(...arr);
    }
    return arr;
}
```

另外 ES6 语法中 新增的 flat函数也可以实现数组的扁平化
参数是固定的

```
const arr = 原始数组.flat( Infinity );
```

69、typeof 原理

利用 typeof 是根据返回值的结果来判断数据类型

具体返回值 一共是 number, string, object, boolean, function, undefined

其中 数组 null 对象 的返回值 都是 object
这样的话具体的数据类型就不能区分的非常明确 在实际项目中 就不能准确的区分

如果想要具体的 区分 数据类型 需要使用 Object.prototype.toString.call() 方法 返回值是

```
object String   字符串
object Number   数值类型
object Boolean  布尔类型
object Undefined    undefined类型
object Null     null类型
object Function     函数类型
object Array    数组类型
```

70、介绍类型转化

JavaScript 因为是 弱类型计算机语言 存储数据时 对变量储存的数据类型没有设定
因此一个变量中可以存储任意类型的数据
在程序的执行过程中 就会遇到需要数据类型转化的情况

```
自动转化
  自动转化为字符串
    数据  直接转化为  对应的字符串
```

自动转化为数值类型
转化为 对应的数值
1 true
0 false null "" ''

```
符合数字规范的字符串
                    转化为 NaN
          不符合数字规范的字符串
           undefined

自动转化为数值类型
            false:
                0  0.0000  ''   NaN   null     undefined
            true:
                其他情况都转化为 true
```

强制转化

```
强制转化为布尔类型

Boolean( 变量 / 表达式 )
                转化原则 和 自动转化原则完全相同
                false : 0   0.000   ''    null   NaN  undefined
                true  : 其他情况都转化为true

    强制转化为字符串类型
        String( 变量 / 表达式 );
            转化原则 和 自动转化原则完全相同
            不会改变 变量中存储的原始数据
      变量.toString( 进制 ) ;
            转化原则 和 自动转化原则完全相同
            不会改变 变量中存储的原始数据
            如果是 整数类型 可以 设定 转化的进制

            变量 存储 null 或者 undefined不支持

    强制转化为数值类型
        Number()
            转化原则 和 自动转化完全相同
            不会改变 变量中存储的原始内容

        parseInt()
            从 左侧起 获取符合整数语法规范的内容部分
            如果 左起第一个字符就不符合整数语法规范
            执行结果是 NaN

        parseFloat()
            从 左侧起 获取符合浮点数语法规范的内容部分
            如果 左起第一个字符就不符合浮点数语法规范
            执行结果是 NaN
```

71、执行上下文

执行上下文：指当前执行环境中的变量、函数声明，参数（arguments），作用域链，this等信息。分为全局执行上下文、函数执行上下文，其区别在于全局执行上下文只有一个，函数执行上下文在每次调用函数时候会创建一个新的函数执行上下文。

变量对象是与执行上下文相关的数据作用域，存储了上下文中定义的变量和函数声明。

变量对象式一个抽象的概念，在不同的上下文中，表示不同的对象:

全局执行上下文的变量对象
全局执行上下文中，变量对象就是全局对象。
在顶层js代码中，this指向全局对象，全局变量会作为该对象的属性来被查询。在浏览器中，window就是全局对象。
函数执行上下文的变量对象
函数上下文中，变量对象VO就是活动对象AO。
初始化时，带有arguments属性。
函数代码分成两个阶段执行
进入执行上下文时，此时变量对象包括
形参
函数声明，会替换已有变量对象
变量声明，不会替换形参和函数
函数执行

执行上下文栈的作用是用来跟踪代码的，由于JS是单线程的，每次只能做一件事情，其他的事情会放在指定的上下文栈中排队等待执行。

JS解释器在初始化代码的时候，首先会创建一个新的全局执行上下文到执行上下文栈顶中，然后随着每次函数的调用都会创建一个新的执行上下文放入到栈顶中，随着函数执行完毕后被执行上下文栈顶弹出，直到回到全局的执行上下文中。

首先创建了全局执行上下文，当前全局执行上下文处于活跃状态。
全局代码中有2个函数 getName 和 getYear，然后调用 getName 函数，JS引擎停止执行全局执行上下文，创建了新的函数执行上下文，且把该函数上下文放入执行上下文栈顶。
getName 函数里又调用了 getYear 函数，此时暂停了 getName 的执行上下文，创建了 getYear 函数的新执行上下文，且把该函数执行上下文放入执行上下文栈顶。
当 getYear 函数执行完后，其执行上下文从栈顶出栈，回到了 getName 执行上下文中继续执行。
当 getName 执行完后，其执行上下文从栈顶出栈，回到了全局执行上下文中。

72、闭包的问题和优化

闭包：是指有权访问另外一个函数作用域中的变量的函数。创建闭包的常见方式就是在一个函数内部创建另外一个函数。

作用：

```
1、可以读取函数内部的变量2、相当于划出了一块私有作用域，避免数据污染；3、让变量始终保存在内存中
```

闭包有三个特性：

1.函数嵌套函数

2.函数内部可以引用外部的参数和变量

3.参数和变量不会被垃圾回收机制回收

闭包的问题

闭包会产生不销毁的上下文，会导致栈/堆内存消耗过大，有时候也会导致内存泄漏等，影响页面的运行性能，所以在真实项目中，要合理应用闭包！

闭包的优化

原始代码

```
function MyObject(name, message) {
  this.name = name.toString();
  this.message = message.toString();
  this.getName = function() {
    return this.name;
  };

  this.getMessage = function() {
    return this.message;
  };
}
```

优化代码

```
function MyObject(name, message) {
  this.name = name.toString();
  this.message = message.toString();
}
MyObject.prototype.getName = function() {
  return this.name;
};
MyObject.prototype.getMessage = function() {
  return this.message;
};
```

73、浏览器和Node事件循环的区别

一、全局环境下this的指向

在node中this指向global而在浏览器中this指向window，这就是为什么underscore中一上来就定义了一root；

而且在浏览器中的window下封装了不少的API 比如 alert 、document、location、history 等等还有很多。我门就不能在node环境中xxx();或window.xxx();了。因为这些API是浏览器级别的封装，存javascript中是没有的。当然node中也提供了不少node特有的API。

二、js引擎

在浏览器中不同的浏览器厂商提供了不同的浏览器内核，浏览器依赖这些内核解释折我们编写的js。但是考虑到不同内核的少量差异，我们需要对应兼容性好在有一些优秀的库帮助我们处理这个问题比如jquery、underscore等等。

nodejs是基于Chromes JavaScript runtime，也就是说，实际上它是对GoogleV8引擎（应用于Google Chrome浏览器)进行了封装。V8引 擎执行Javascript的速度非常快，性能非常好。

NodeJS并不是提供简单的封装，然后提供API调用，如果是这样的话那么它就不会有现在这么火了。Node对一些特殊用例进行了优化，提供了替代的API，使得V8在非浏览器环境下运行得更好。例如，在服务器环境中，处理二进制数据通常是必不可少的，但Javascript对此支持不足，因此，V8.Node增加了Buffer类，方便并且高效地 处理二进制数据。因此，Node不仅仅简单的使用了V8,还对其进行了优化，使其在各环境下更加给力。

三、DOM操作

浏览器中的js大多数情况下是在直接或间接（一些虚拟DOM的库和框架）的操作DOM。因为浏览器中的代码主要是在表现层工作。但是node是一门服务端技术。没有一个前台页面，所以我门不会再node中操作DOM。

四、I/O读写

与浏览器不同，我们需要像其他服务端技术一样读写文件，nodejs提供了比较方便的组件。而浏览器（确保兼容性的）想在页面中直接打开一个本地的图片就麻烦了好多（别和我说这还不简单，相对路径。。。。。。试试就知道了要么找个库要么二进制流，要么上传上去有了网络地址在显示。不然人家为什么要搞一个js库呢），而这一切node都用一个组件搞定了。

五、模块加载

javascript有个特点，就是原生没提供包引用的API一次性把要加载的东西全执行一遍，这里就要看各位闭包的功力了。所用东西都在一起，没有分而治之，搞的特别没有逻辑性和复用性。如果页面简单或网站当然我们可以通过一些AMD、CMD的js库（比如requireJS 和 seaJS）搞定事实上很多大型网站都是这么干的。

在nodeJS中提供了CMD的模块加载的API，如果你用过seaJS，那么应该上手很快。
node还提供了npm 这种包管理工具，能更有效方便的管理我们饮用的库

74、移动端点击延迟

原因：

为了确定用户是要做单击 还是双击 还是要做其他的操作 因此移动端 当你点击时 会有 300毫秒延迟 为了等待判断用户的下一步操作是什么

解决方案1

禁用缩放

```
<meta name="viewport" content="user-scalable=no">
<meta name="viewport" content="initial-scale=1,maximum-scale=1">
```

当HTML文档头部包含以上meta标签时 表明这个页面是不可缩放的，那双击缩放的功能就没有意义了，此时浏览器可以禁用默认的双击缩放行为并且去掉300ms的点击延迟。
这个方案有一个缺点，就是必须通过完全禁用缩放来达到去掉点击延迟的目的，然而完全禁用缩放并不是我们的初衷，我们只是想禁掉默认的双击缩放行为，这样就不用等待300ms来判断当前操作是否是双击。但是通常情况下，我们还是希望页面能通过双指缩放来进行缩放操作，比如放大一张图片，放大一段很小的文字。

解决方案2 更改默认的视口宽度

```
<meta name="viewport" content="width=device-width">
```

一开始，为了让桌面站点能在移动端浏览器正常显示，移动端浏览器默认的视口宽度并不等于设备浏览器视窗宽度，而是要比设备浏览器视窗宽度大，通常是980px。我们可以通过以下标签来设置视口宽度为设备宽度。因为双击缩放主要是用来改善桌面站点在移动端浏览体验的，而随着响应式设计的普及，很多站点都已经对移动端坐过适配和优化了，这个时候就不需要双击缩放了，如果能够识别出一个网站是响应式的网站，那么移动端浏览器就可以自动禁掉默认的双击缩放行为并且去掉300ms的点击延迟。如果设置了上述meta标签，那浏览器就可以认为该网站已经对移动端做过了适配和优化，就无需双击缩放操作了。
这个方案相比方案一的好处在于，它没有完全禁用缩放，而只是禁用了浏览器默认的双击缩放行为，但用户仍然可以通过双指缩放操作来缩放页面。

解决方案3 CSS touch-action

跟300ms点击延迟相关的，是touch-action这个CSS属性。这个属性指定了相应元素上能够触发的用户代理（也就是浏览器）的默认行为。如果将该属性值设置为touch-action: none，那么表示在该元素上的操作不会触发用户代理的任何默认行为，就无需进行300ms的延迟判断。

最后的最后 我们还可以使用一些 插件来解决这个问题 例如 FastClick 是 FT Labs 专门为解决移动端浏览器 300 毫秒点击延迟问题所开发的一个轻量级的库。FastClick的实现原理是在检测到touchend事件的时候，会通过DOM自定义事件立即出发模拟一个click事件，并把浏览器在300ms之后的click事件阻止掉。

安装 `npm install fastclick -S`

使用 如何你是vue项目可以在main.js里面直接引入，当然这样是全局的，如果你需要某个页面用到，那就单个页面引入。

```
//引入
import fastClick from 'fastclick'
//初始化FastClick实例。在页面的DOM文档加载完成后
fastClick.attach(document.body)
```

75、cookie属性

cookie的常见属性

```
键名  cookie键值对的键名
键值  cookie键值对的键值
expires cookie的时效 分为 session会话时效 时间时效 时间时效是服务器时间也就是世界标准时间
path    路径 符合路径的文件才能访问cookie
httponly    设置 为 true 了之后可以防止js程序访问 防止 xss攻击 增加cookie的安全性
secure  设置 为 true 了之后cookie只能通过https协议发送 http协议是不能发送的 这样也是为了增加
cookie的安全性
```

76、反柯里化

反柯里化的作用是，当我们调用某个方法，不用考虑这个对象在被设计时，是否拥有这个方法，只要这个方法适用于它，我们就可以对这个对象使用它

例如

```
Function.prototype.uncurring = function() {
  var self = this;
  return function() {
    var obj = Array.prototype.shift.call(arguments);
    return self.apply(obj, arguments);
  };
};
```

我们先来看看上面这段代码有什么作用。

我们要把Array.prototype.push方法转换成一个通用的push函数，只需要这样做：

```
var push = Array.prototype.push.uncurring();

//测试一下
(function() {
  push(arguments, 4);
  console.log(arguments); //[1, 2, 3, 4]
})(1, 2, 3)
```

arguments本来是没有push方法的，通常，我们都需要用Array.prototype.push.call来实现push方法，但现在，直接调用push函数，既简洁又意图明了。

我们不用考虑对象是否拥有这个方法，只要它适用于这个方法，那就可以使用这个方法（类似于鸭子类型）。

我们来分析一下调用Array.prototype.push.uncurring()这句代码时，发生了什么事情：

```
Function.prototype.uncurring = function() {
  var self = this;  //self此时是Array.prototype.push

  return function() {
    var obj = Array.prototype.shift.call(arguments);
    //obj 是{
    //  "length": 1,
```

```
    //  "0": 1
    //}
    //arguments的第一个对象被截去(也就是调用push方法的对象),剩下[2]

    return self.apply(obj, arguments);
    //相当于Array.prototype.push.apply(obj, 2);

  };
};
```

//测试一下

```
var push = Array.prototype.push.uncurring();
var obj = {
  "length": 1,
  "0" : 1
};

push(obj, 2);
console.log( obj ); //{0: 1,1: 2, length: 2 }
```

看到这里你应该对柯里化和反柯里化有了一个初步的认识了，但要熟练的运用在开发中，还需要我们更深入的去了解它们内在的含义。

77、千分位

这里的需求 本质上是要 将 数字 转化为 带有千分位字符串 方法有很多

方法1 正则表达式

```
console.info( str.replace(/\d{1,3}(?=(\d{3})+$)/g,function(s){
  return s+','
}) )
```

方法2 字符串替换

```
console.info( str.replace(/(\d{1,3})(?=(\d{3})+$)/g,function($1){
  return $1=$1+','
}) )
```

方法3 数字转数组 反转后 添加 , 再反转回来拼接为字符串

```
console.info( str.split("").reverse().join("").replace(/(\d{3})+?/g,function(s){
  return s+",";
}).replace(/,$/,"").split("").reverse().join("") )
```

方法4 利用while循环拼接字符串每隔3个数字加一个分隔符，首尾不加

```
var result="",
  index = 0,
  len = str.length-1;
while(len>=0) {
  index%3===0&&index!==0 ? result+=","+str[len] : result+=str[len];
  len--;
  index++;
};
result=result.split("").reverse().join("");
console.info(result);
```

方法5 利用while循环在数组里push分隔符，首尾不加

```
// 利用while循环在数组里push分隔符
var result="",
  index = 0,
  len = str.length,
  i = len-1,
  arr = str.split("");
while(len-index>0){
  len>=index&&len-index!==len && arr.splice(len-index,0,",");
  index+=3;
  i-=4;
};
console.log(arr.join(""));
```

78、load和ready区别

```
document.ready:

是ready，表示文档结构已经加载完成 不包含图片等非文字媒体文件 只要html标签结构加载完毕就可以;

document.load:

是onload，指示页面包含图片等文件在内的所有元素都加载完成。
```

1、概念

2、作用

```
document.ready:

在DOM加载完成后就可以可以对DOM进行操作。

一般情况一个页面响应加载的顺序是，域名解析-加载html-加载js和css-加载图片等其他信息。
那么Dom Ready应该在“加载js和css”和“加载图片等其他信息”之间，就可以操作Dom了。

document.load:

在document文档加载完成后就可以可以对DOM进行操作，document文档包括了加载图片等其他信息。

那么Dom Load就是在页面响应加载的顺序中的“加载图片等其他信息”之后，就可以操作Dom了。
```

3、加载顺序

document.ready：

文档加载的顺序：域名解析-->加载HTML-->加载JavaScript和CSS-->加载图片等非文字媒体文件。

只要<img>标签加载完成，不用等该图片加载完成，就可以设置图片的属性或样式等。

在原生JavaScript中没有Dom ready的直接方法。

document.load：

文档加载的顺序：域名解析-->加载HTML-->加载JavaScript和CSS-->加载图片等非文字媒体文件。

DOM load在加载图片等非文字媒体文件之后，表示在document文档加载完成后才可以对DOM进行操作，document文档包括了加载图片等非文字媒体文件。

例如，需要等该图片加载完成，才可以设置图片的属性或样式等。

在原生JavaScript中使用onload事件。

79、自定义事件

自定义事件，就是自己定义事件类型，自己定义事件处理函数。

我们平时操作dom时经常会用到onclick、onmousemove等浏览器特定行为的事件类型。

封装js自定义事件基本的构思：

```
var eventTarget = {
  addEvent: function(){
    //添加事件
  },
  fireEvent: function(){
    //触发事件
  },
  removeEvent: function(){
    //移除事件
  }
};
```

在js默认事件中事件类型以及对应的执行函数是一一对应的，但是自定义事件，需要一个映射表来建立两者之间的联系。

如： 这样每个类型可以处理多个事件函数

```
handlers = {
      "type1":[
            "fun1",
            "fun2",
            // "..."
        ],
       "type2":[
            "fun1",
            "fun2"
             // "..."
        ]
        //"..."
}
```

代码实现：

```
function EventTarget(){
    //事件处理程序数组集合
    this.handlers={};
}

//自定义事件的原型对象
EventTarget.prototype={
    //设置原型构造函数链
    constructor:EventTarget,
    //注册给定类型的事件处理程序
    //type->自定义事件类型，如click,handler->自定义事件回调函数
    addEvent:function(type,handler){
        //判断事件处理函数中是否有该类型事件
        if(this.handlers[type]==undefined){
            this.handlers[type]=[];
        }
        this.handlers[type].push(handler);
    },

    //触发事件
    //event为一个js对象，属性中至少包含type属性。
    fireEvent:function(event){
        //模拟真实事件的event
        if(!event.target){
            event.target=this;
        }
        //判断是否存在该事件类型
        if(this.handlers[event.type] instanceof Array){
            var items=this.handlers[event.type];
            //在同一事件类型下可能存在多个事件处理函数，依次触发
            //执行触发
            items.forEach(function(item){
                item(event);
            })
        }
    },

    //删除事件
    removeEvent:function(type,handler){
        //判断是否存在该事件类型
```

```
        if(this.handlers[type] instanceof Array){
            var items=this.handlers[type];
            //在同一事件类型下可能存在多个处理事件
            for(var i=0;i<items.length;i++){
                if(items[i]==handler){
                    //从该类型的事件数组中删除该事件
                    items.splice(i,1);
                    break;
                }
            }
        }
    }

}

//调用方法
function fun(){
    console.log('执行该方法');
}
function fun1(obj){
    console.log('run '+obj.min+'s');
}
var target=new EventTarget();
target.addEvent("run",fun);//添加事件
target.addEvent("run",fun1);//添加事件

target.fireEvent({type:"run",min:"30"});//执行该方法    123

target.removeEvent("run",fun);//移除事件

target.fireEvent({type:"run",min:"20"});//123
```

为什么要把方法添加到对象原型上?

在构造函数中加属性，在原型中加方法。

将属性和方法都写在构造函数里是没有问题的，但是每次进行实例化的过程中，要重复创建功能不变的方法。

由于方法本质上是函数，其实也就是在堆内存中又新建了一个对象空间存放存储函数，造成了不必要的资源浪费。

在本身添加会导致每次对象实例化时代码被复制，都需要申请一块内存存放该方法。

写一个EventEmitter类，包括on()、off()、once()、emit()方法
once()：为指定事件注册一个单次监听器，单次监听器最多只触发一次，触发后立即解除监听器。

```
class EventEmitter{
            constructor(){
                this.handlers={};
            }
            on(type,fn){
                if(!this.handlers[type]){
                    this.handlers[type]=[];
```

```
            }
            this.handlers[type].push(fn);
            return this;
        }
        off(type,fn){
            let fns=this.handlers[type];
            for(let i=0;i<fns.length;i++){
                if(fns[i]==fn){
                    fns.splice(i,1);
                    break;
                }
            }
            return this;
        }
        emit(...args){
            let type=args[0];
            let params=[].slice.call(args,1);
            let fn=this.handlers[type];
            fn.forEach((item)=>{
                item.apply(this,params);//执行函数
            })
            return this;
        }
        once(type,fn){
            let wrap=(...args)=>{
                fn.apply(this,args);//执行事件后删除
                this.off(type,wrap);
            }
            this.on(type,wrap);//再添加上去
            return this;
        }
    }
     let emitter=new EventEmitter();
function fun1(){
    console.log('fun1');
}
function fun2(){
    console.log('fun2');
}
function fun3(){
    console.log('fun3');
}
emitter.on('TEST1',fun1).on('TEST2',fun2).emit('TEST1').once('TEST2',fun3);
emitter.emit("TEST2");
```

80、setTimeout实现setInterval

```
setTimeout() ：在指定的毫秒数后调用函数或计算表达式，只执行一次。
setInterval() ：按照指定的周期（以毫秒计）来调用函数或计算表达式。方法会不停地调用函数，直到
clearInterval() 被调用或窗口被关闭。
```

思路是使用递归函数，不断地去执行setTimeout从而达到setInterval的效果，看代码

```
function mySetInterval(fn, millisec){
  function interval(){
    setTimeout(interval, millisec);
    fn();
  }
  setTimeout(interval, millisec)
}
```

这个mySetInterval函数有一个叫做interval的内部函数，它通过setTimeout来自动被调用，在interval中有一个闭包，调用了回调函数并通过setTimeout再次调用了interval。

一个更好的实现
我们再增加一个额外的参数用来标明代码执行的次数

```
function mySetInterval(fn, millisec,count){
  function interval(){
    if(typeof count===‘undefined’||count-->0){
      setTimeout(interval, millisec);
      try{
        fn()
      }catch(e){
        count = 0;
        throw e.toString();
      }
    }
  }
  setTimeout(interval, millisec)
}
```

81、避免回调地狱

使用 async await 配合 promise 是 解决回调地狱的终极方法

async/await特点

```
1，async/await更加语义化，async 是“异步”的简写，async function 用于申明一个 function 是异步的； await，可以认为是async wait的简写， 用于等待一个异步方法执行完成；

2，async/await是一个用同步思维解决异步问题的方案（等结果出来之后，代码才会继续往下执行）

3，可以通过多层 async function 的同步写法代替传统的callback嵌套
```

async function语法

```
1，自动将常规函数转换成Promise，返回值也是一个Promise对象

2，只有async函数内部的异步操作执行完，才会执行then方法指定的回调函数

3，异步函数内部可以使用await
```

await语法

1，await 放置在Promise调用之前，await 强制后面点代码等待，直到Promise对象resolve，得到resolve的值作为await表达式的运算结果

2. await只能在async函数内部使用,用在普通函数里就会报错

函数形式

```
function timeout(ms) {

  return new Promise((resolve, reject) => {

    setTimeout(() => {reject('error')}, ms);  //reject模拟出错，返回error

  });

}

async function asyncPrint(ms) {

  try {

     console.log('start');

     await timeout(ms);  //这里返回了错误

     console.log('end');  //所以这句代码不会被执行了

  } catch(err) {

     console.log(err); //这里捕捉到错误error

  }

}
```

82、callee和caller的作用

caller返回一个函数的引用，这个函数调用了当前的函数;callee放回正在执行的函数本身的引用，它是arguments的一个属性

caller
caller返回一个函数的引用，这个函数调用了当前的函数。
使用这个属性要注意:
1 这个属性只有当函数在执行时才有用
2 如果在javascript程序中，函数是由顶层调用的，则返回null

```
functionName.caller: functionName是当前正在执行的函数。
var a = function() {
    alert(a.caller);
}
var b = function() {
    a();
}
b();
```

上面的代码中，b调用了a，那么a.caller返回的是b的引用，结果如下:

```
var b = function() {
    a();
}
```

如果直接调用a(即a在任何函数中被调用，也就是顶层调用),返回null:

```
var a = function() {
    alert(a.caller);
}
var b = function() {
    a();
}
//b();
a();
输出结果:
null
```

callee
callee放回正在执行的函数本身的引用，它是arguments的一个属性
使用callee时要注意:

```
1 这个属性只有在函数执行时才有效

2 它有一个length属性，可以用来获得形参的个数，因此可以用来比较形参和实参个数是否一致，即比较
arguments.length是否等于arguments.callee.length

3 它可以用来递归匿名函数。

var a = function() {
    alert(arguments.callee);
}
var b = function() {
    a();
}
b();


a在b中被调用，但是它返回了a本身的引用，结果如下:

var a = function() {
    alert(arguments.callee);
}
```

83、统计字符串中字母个数或统计最多的字母数

统计字母出现的次数

```
function count( str ){
    var obj={};
    for(var i=0;i<str.length; i++){
        if(obj[ str[i] ]==undefined){
            //对象初始化；如果key在对象中找不到，那么会返回undefined,反向思维
            obj[ str[i] ]= 1;
        } else{
            obj[ str[i] ]++;
        }
    }

    //取出各个字母和它的个数，作为一个新对象保存在obj对象中
    return obj;

}

cosnle.log( count( "shhkfahkahsadhadskhdskdha" ) );
```

统计字符出现次数最多的字母

```
function allProMax(obj){
    var mm="";
    for(var m in obj){
        if(mm==""){
            mm=new Object();
            mm[m]=obj[m];
        }else{
            for(var j in mm){
               if(mm[j]<obj[m]){
                   //清空原来的内容
                   mm=new Object();
                   //放入新的内容
                   mm[m]=obj[m];
               }
            }
        }
    }
    return mm ;
}

console.log( allProMax(count()) )
```

84、面对对象和面向过程的区别

一、面向对象与面向过程的区别
面向过程就是分析出解决问题所需要的步骤，然后用函数把这些步骤一步一步实现，使用的时候一个一个依次调用就可以了；面向对象是把构成问题事务分解成各个对象，建立对象的目的不是为了完成一个步骤，而是为了描叙某个事物在整个解决问题的步骤中的行为。

可以拿生活中的实例来理解面向过程与面向对象，例如五子棋，面向过程的设计思路就是首先分析问题的步骤：1、开始游戏，2、黑子先走，3、绘制画面，4、判断输赢，5、轮到白子，6、绘制画面，7、判断输赢，8、返回步骤2，9、输出最后结果。把上面每个步骤用不同的方法来实现。

如果是面向对象的设计思想来解决问题。面向对象的设计则是从另外的思路来解决问题。整个五子棋可以分为1、黑白双方，这两方的行为是一模一样的，2、棋盘系统，负责绘制画面，3、规则系统，负责判定诸如犯规、输赢等。第一类对象（玩家对象）负责接受用户输入，并告知第二类对象（棋盘对象）棋子布局的变化，棋盘对象接收到了棋子的变化就要负责在屏幕上面显示出这种变化，同时利用第三类对象（规则系统）来对棋局进行判定。

可以明显地看出，面向对象是以功能来划分问题，而不是步骤。同样是绘制棋局，这样的行为在面向过程的设计中分散在了多个步骤中，很可能出现不同的绘制版本，因为通常设计人员会考虑到实际情况进行各种各样的简化。而面向对象的设计中，绘图只可能在棋盘对象中出现，从而保证了绘图的统一。

上述的内容是从网上查到的，觉得这个例子非常的生动形象，我就写了下来，现在就应该理解了他俩的区别了吧，其实就是两句话，面向对象就是高度实物抽象化、面向过程就是自顶向下的编程！

二、面向对象的特点
在了解其特点之前，咱们先谈谈对象，对象就是现实世界存在的任何事务都可以称之为对象，有着自己独特的个性

1，概念 对 具有相同特性的一类事物的抽象描述

2，组成 属性 和 方法

3，模板 构造函数

4，特点 封装 继承 多态

属性用来描述具体某个对象的特征。比如小志身高180M，体重70KG，这里身高、体重都是属性。
面向对象的思想就是把一切都看成对象，而对象一般都由属性+方法组成！

属性属于对象静态的一面，用来形容对象的一些特性，方法属于对象动态的一面，咱们举一个例子，小明会跑，会说话，跑、说话这些行为就是对象的方法！所以为动态的一面， 我们把属性和方法称为这个对象的成员！

类：具有同种属性的对象称为类，是个抽象的概念。比如“人”就是一类，期中有一些人名，比如小明、小红、小玲等等这些都是对象，类就相当于一个模具，他定义了它所包含的全体对象的公共特征和功能，对象就是类的一个实例化，小明就是人的一个实例化！我们在做程序的时候，经常要将一个变量实例化，就是这个原理！我们一般在做程序的时候一般都不用类名的，比如我们在叫小明的时候，不会喊“人，你干嘛呢！”而是说的是“小明，你在干嘛呢！”

面向对象有三大特性，分别是封装性、继承性和多态性，这里小编不给予太多的解释，因为在后边的博客会专门总结的！

三、面向过程与面向对象的优缺点
很多资料上全都是一群很难理解的理论知识，整的小编头都大了，后来发现了一个比较好的文章，写的真是太棒了，通俗易懂，想要不明白都难！

用面向过程的方法写出来的程序是一份蛋炒饭，而用面向对象写出来的程序是一份盖浇饭。所谓盖浇饭，北京叫盖饭，东北叫烩饭，广东叫碟头饭，就是在一碗白米饭上面浇上一份盖菜，你喜欢什么菜，你就浇上什么菜。我觉得这个比喻还是比较贴切的。

蛋炒饭制作的细节，我不太清楚，因为我没当过厨师，也不会做饭，但最后的一道工序肯定是把米饭和鸡蛋混在一起炒匀。盖浇饭呢，则是把米饭和盖菜分别做好，你如果要一份红烧肉盖饭呢，就给你浇一份红烧肉；如果要一份青椒土豆盖浇饭，就给浇一份青椒土豆丝。

蛋炒饭的好处就是入味均匀，吃起来香。如果恰巧你不爱吃鸡蛋，只爱吃青菜的话，那么唯一的办法就是全部倒掉，重新做一份青菜炒饭了。盖浇饭就没这么多麻烦，你只需要把上面的盖菜拨掉，更换一份盖菜就可以了。盖浇饭的缺点是入味不均，可能没有蛋炒饭那么香。

到底是蛋炒饭好还是盖浇饭好呢？其实这类问题都很难回答，非要比个上下高低的话，就必须设定一个场景，否则只能说是各有所长。如果大家都不是美食家，没那么多讲究，那么从饭馆角度来讲的话，做盖浇饭显然比蛋炒饭更有优势，他可以组合出来任意多的组合，而且不会浪费。

盖浇饭的好处就是"菜"“饭"分离，从而提高了制作盖浇饭的灵活性。饭不满意就换饭，菜不满意换菜。用软件工程的专业术语就是"可维护性"比较好，“饭” 和"菜"的耦合度比较低。蛋炒饭将"蛋”“饭"搅和在一起，想换"蛋”"饭"中任何一种都很困难，耦合度很高，以至于"可维护性"比较差。软件工程追求的目标之一就是可维护性，可维护性主要表现在3个方面：可理解性、可测试性和可修改性。面向对象的好处之一就是显著的改善了软件系统的可维护性。

我们最后简单总结一下

面向过程

优点：性能比面向对象高，因为类调用时需要实例化，开销比较大，比较消耗资源;比如单片机、嵌入式开发、Linux/Unix等一般采用面向过程开发，性能是最重要的因素。
缺点：没有面向对象易维护、易复用、易扩展

面向对象

优点：易维护、易复用、易扩展，由于面向对象有封装、继承、多态性的特性，可以设计出低耦合的系统，使系统 更加灵活、更加易于维护
缺点：性能比面向过程低

85、eval

eval()是全局对象的一个函数属性。

eval()的参数是一个字符串。如果字符串表示的是表达式，eval()会对表达式进行求值。如果参数表示一个或多个JavaScript语句， 那么eval()就会执行这些语句。注意不要用eval()来执行一个四则运算表达式；因为JavaScript 会自动为四则运算求值并不需要用eval来包裹。

这里的四则运算是指数学上的运算，如：3 + 4 * 4 / 6。注意这里面并没有变量，只是单纯的数学运算，这样的运算式并不需要调用eval来计算，直接在代码中计算就可以。其实即便带有变量，JavaScript也是可以直接计算的，但是如果你现在只想声明一个带有变量的表达式，但是想稍后进行运算（你有可能在声明这个带有变量的运算式之后还有可能对里面的变量进行修改），就可以使用eval。
如果要将算数表达式构造成为一个字符串，你可以用eval()在随后对其求值。比如，假如你有一个变量 x ，你可以通过一个字符串表达式来对涉及x的表达式延迟求值，将 “3 * x + 2”，存储为变量，然后在你的脚本后面的一个地方调用eval()。

如果eval()的参数不是字符串，eval()将会将参数原封不动的返回。在下面的例子中，字符串构造器被指定，eval()返回了字符串对象而不是对字符串求值。

```
 // 返回了包含"2 + 2"的字符串对象
eval(new String("2 + 2"));

// returns 4
eval("2 + 2");
```

eval() 是一个危险的函数， 他执行的代码拥有着执行者的权利。如果你用eval()运行的字符串代码被恶意方（不怀好意的人）操控修改,您可能会利用最终在用户机器上运行恶意方部署的恶意代码，并导致您失去您的网页或者扩展程序的权限。更重要的是，第三方代码可以看到某一个eval()被调用时的作用域，这也有可能导致一些不同方式的攻击。相似的Function就是不容易被攻击的。

eval()的运行效率也普遍的比其他的替代方案慢，因为他会调用js解析器，即便现代的JS引擎中已经对此做了优化。

在常见的案例中我们都会找更安全或者更快的方案去替换他

86、 proxy

proxy在目标对象的外层搭建了一层拦截，外界对目标对象的某些操作，必须通过这层拦截

```
var proxy = new Proxy(target, handler);
new Proxy()表示生成一个Proxy实例，target参数表示所要拦截的目标对象，handler参数也是一个对象，用来定制拦截行为

var target = {
   name: 'poetries'
 };
 var logHandler = {
   get: function(target, key) {
     console.log(`${key} 被读取`);
     return target[key];
   },
   set: function(target, key, value) {
     console.log(`${key} 被设置为 ${value}`);
     target[key] = value;
   }
 }
 var targetWithLog = new Proxy(target, logHandler);

 targetWithLog.name; // 控制台输出：name 被读取
 targetWithLog.name = 'others'; // 控制台输出：name 被设置为 others

 console.log(target.name); // 控制台输出: others
targetWithLog 读取属性的值时，实际上执行的是 logHandler.get ：在控制台输出信息，并且读取被代理对象 target 的属性。
在 targetWithLog 设置属性值时，实际上执行的是 logHandler.set ：在控制台输出信息，并且设置被代理对象 target 的属性的值
// 由于拦截函数总是返回35，所以访问任何属性都得到35
var proxy = new Proxy({}, {
  get: function(target, property) {
    return 35;
  }
});

proxy.time // 35
proxy.name // 35
proxy.title // 35
Proxy 实例也可以作为其他对象的原型对象

var proxy = new Proxy({}, {
```

```
  get: function(target, property) {
    return 35;
  }
});

let obj = Object.create(proxy);
obj.time // 35
```

proxy对象是obj对象的原型，obj对象本身并没有time属性，所以根据原型链，会在proxy对象上读取该属性，导致被拦截

Proxy的作用

对于代理模式 Proxy 的作用主要体现在三个方面

拦截和监视外部对对象的访问
降低函数或类的复杂度
在复杂操作前对操作进行校验或对所需资源进行管理

二、Proxy所能代理的范围--handler

```
实际上 handler 本身就是ES6所新设计的一个对象.它的作用就是用来 自定义代理对象的各种可代理操作 。它本身一共有13中方法,每种方法都可以代理一种操作.其13种方法如下

// 在读取代理对象的原型时触发该操作，比如在执行 Object.getPrototypeOf(proxy) 时。
handler.getPrototypeOf()

// 在设置代理对象的原型时触发该操作，比如在执行 Object.setPrototypeOf(proxy, null) 时。
handler.setPrototypeOf()


// 在判断一个代理对象是否是可扩展时触发该操作，比如在执行 Object.isExtensible(proxy) 时。
handler.isExtensible()


// 在让一个代理对象不可扩展时触发该操作，比如在执行 Object.preventExtensions(proxy) 时。
handler.preventExtensions()

// 在获取代理对象某个属性的属性描述时触发该操作，比如在执行
Object.getOwnPropertyDescriptor(proxy, "foo") 时。
handler.getOwnPropertyDescriptor()


// 在定义代理对象某个属性时的属性描述时触发该操作，比如在执行 Object.defineProperty(proxy,
"foo", {}) 时。
andler.defineProperty()


// 在判断代理对象是否拥有某个属性时触发该操作，比如在执行 "foo" in proxy 时。
handler.has()

// 在读取代理对象的某个属性时触发该操作，比如在执行 proxy.foo 时。
handler.get()


// 在给代理对象的某个属性赋值时触发该操作，比如在执行 proxy.foo = 1 时。
handler.set()
```

```
// 在删除代理对象的某个属性时触发该操作，比如在执行 delete proxy.foo 时。
handler.deleteProperty()

// 在获取代理对象的所有属性键时触发该操作，比如在执行 Object.getOwnPropertyNames(proxy) 时。
handler.ownKeys()

// 在调用一个目标对象为函数的代理对象时触发该操作，比如在执行 proxy() 时。
handler.apply()


// 在给一个目标对象为构造函数的代理对象构造实例时触发该操作，比如在执行new proxy() 时。
handler.construct()
```

三、Proxy场景

3.1 实现私有变量

```
var target = {
   name: 'poetries',
   _age: 22
}

var logHandler = {
  get: function(target,key){
    if(key.startsWith('_')){
      console.log('私有变量age不能被访问')
      return false
    }
    return target[key];
  },
  set: function(target, key, value) {
     if(key.startsWith('_')){
      console.log('私有变量age不能被修改')
      return false
    }
     target[key] = value;
   }
}
var targetWithLog = new Proxy(target, logHandler);

// 私有变量age不能被访问
targetWithLog.name;

// 私有变量age不能被修改
targetWithLog.name = 'others';
```

在下面的代码中，我们声明了一个私有的 apiKey，便于 api 这个对象内部的方法调用，但不希望从外部也能够访问 api._apiKey

```
var api = {
    _apiKey: '123abc456def',
    /* mock methods that use this._apiKey */
    getUsers: function(){},
    getUser: function(userId){},
    setUser: function(userId, config){}
};
```

```
// logs '123abc456def';
console.log("An apiKey we want to keep private", api._apiKey);

// get and mutate _apiKeys as desired
var apiKey = api._apiKey;
api._apiKey = '987654321';
```

很显然，约定俗成是没有束缚力的。使用 ES6 Proxy 我们就可以实现真实的私有变量了，下面针对不同的读取方式演示两个不同的私有化方法。第一种方法是使用 set / get 拦截读写请求并返回 undefined:

```
let api = {
    _apiKey: '123abc456def',
    getUsers: function(){ },
    getUser: function(userId){ },
    setUser: function(userId, config){ }
};

const RESTRICTED = ['_apiKey'];
api = new Proxy(api, {
    get(target, key, proxy) {
        if(RESTRICTED.indexOf(key) > -1) {
            throw Error(`${key} is restricted. Please see api documentation for
further info.`);
        }
        return Reflect.get(target, key, proxy);
    },
    set(target, key, value, proxy) {
        if(RESTRICTED.indexOf(key) > -1) {
            throw Error(`${key} is restricted. Please see api documentation for
further info.`);
        }
        return Reflect.get(target, key, value, proxy);
    }
});

// 以下操作都会抛出错误
console.log(api._apiKey);
api._apiKey = '987654321';
```

第二种方法是使用 has 拦截 in 操作

```
var api = {
    _apiKey: '123abc456def',
    getUsers: function(){ },
    getUser: function(userId){ },
    setUser: function(userId, config){ }
};

const RESTRICTED = ['_apiKey'];
api = new Proxy(api, {
    has(target, key) {
        return (RESTRICTED.indexOf(key) > -1) ?
            false :
            Reflect.has(target, key);
    }
});

// these log false, and `for in` iterators will ignore _apiKey
console.log("_apiKey" in api);
```

```
for (var key in api) {
    if (api.hasOwnProperty(key) && key === "_apiKey") {
        console.log("This will never be logged because the proxy obscures _apiKey...")
    }
}
```

3.2 抽离校验模块

让我们从一个简单的类型校验开始做起，这个示例演示了如何使用 Proxy 保障数据类型的准确性

```
let numericDataStore = {
    count: 0,
    amount: 1234,
    total: 14
};

numericDataStore = new Proxy(numericDataStore, {
    set(target, key, value, proxy) {
        if (typeof value !== 'number') {
            throw Error("Properties in numericDataStore can only be numbers");
        }
        return Reflect.set(target, key, value, proxy);
    }
});

// 抛出错误，因为 "foo" 不是数值
numericDataStore.count = "foo";

// 赋值成功
numericDataStore.count = 333;
```

如果要直接为对象的所有属性开发一个校验器可能很快就会让代码结构变得臃肿，使用 Proxy 则可以将校验器从核心逻辑分离出来自成一体

```
function createValidator(target, validator) {
    return new Proxy(target, {
        _validator: validator,
        set(target, key, value, proxy) {
            if (target.hasOwnProperty(key)) {
                let validator = this._validator[key];
                if (!!validator(value)) {
                    return Reflect.set(target, key, value, proxy);
                } else {
                    throw Error(`Cannot set ${key} to ${value}. Invalid.`);
                }
            } else {
                throw Error(`${key} is not a valid property`)
            }
        }
    });
}

const personValidators = {
    name(val) {
        return typeof val === 'string';
    },
    age(val) {
```

```
        return typeof age === 'number' && val > 18;
    }
}
class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
        return createValidator(this, personValidators);
    }
}

const bill = new Person('Bill', 25);

// 以下操作都会报错
bill.name = 0;
bill.age = 'Bill';
bill.age = 15;
```

通过校验器和主逻辑的分离，你可以无限扩展 `personValidators` 校验器的内容，而不会对相关的类或函数造成直接破坏。更复杂一点，我们还可以使用 `Proxy` 模拟类型检查，检查函数是否接收了类型和数量都正确的参数

```
let obj = {
    pickyMethodOne: function(obj, str, num) { /* ... */ },
    pickyMethodTwo: function(num, obj) { /*... */ }
};

const argTypes = {
    pickyMethodOne: ["object", "string", "number"],
    pickyMethodTwo: ["number", "object"]
};

obj = new Proxy(obj, {
    get: function(target, key, proxy) {
        var value = target[key];
        return function(...args) {
            var checkArgs = argChecker(key, args, argTypes[key]);
            return Reflect.apply(value, target, args);
        };
    }
});

function argChecker(name, args, checkers) {
    for (var idx = 0; idx < args.length; idx++) {
        var arg = args[idx];
        var type = checkers[idx];
        if (!arg || typeof arg !== type) {
            console.warn(`You are incorrectly implementing the signature of ${name}.
Check param ${idx + 1}`);
        }
    }
}

obj.pickyMethodOne();
// > You are incorrectly implementing the signature of pickyMethodOne. Check param 1
// > You are incorrectly implementing the signature of pickyMethodOne. Check param 2
// > You are incorrectly implementing the signature of pickyMethodOne. Check param 3

obj.pickyMethodTwo("wopdopadoo", {});
// > You are incorrectly implementing the signature of pickyMethodTwo. Check param 1
```

```
// No warnings logged
obj.pickyMethodOne({}, "a little string", 123);
obj.pickyMethodOne(123, {});
```

3.3 访问日志

对于那些调用频繁、运行缓慢或占用执行环境资源较多的属性或接口，开发者会希望记录它们的使用情况或性能表现，这个时候就可以使用 `Proxy` 充当中间件的角色，轻而易举实现日志功能

```
let api = {
    _apiKey: '123abc456def',
    getUsers: function() { /* ... */ },
    getUser: function(userId) { /* ... */ },
    setUser: function(userId, config) { /* ... */ }
};

function logMethodAsync(timestamp, method) {
    setTimeout(function() {
        console.log(`${timestamp} - Logging ${method} request asynchronously.`);
    }, 0)
}

api = new Proxy(api, {
    get: function(target, key, proxy) {
        var value = target[key];
        return function(...arguments) {
            logMethodAsync(new Date(), key);
            return Reflect.apply(value, target, arguments);
        };
    }
});

api.getUsers();
```

3.4 预警和拦截

假设你不想让其他开发者删除 `noDelete` 属性，还想让调用 `oldMethod` 的开发者了解到这个方法已经被废弃了，或者告诉开发者不要修改 `doNotChange` 属性，那么就可以使用 `Proxy` 来实现

```
let dataStore = {
    noDelete: 1235,
    oldMethod: function() {/*...*/ },
    doNotChange: "tried and true"
};

const NODELETE = ['noDelete'];
const NOCHANGE = ['doNotChange'];
const DEPRECATED = ['oldMethod'];

dataStore = new Proxy(dataStore, {
    set(target, key, value, proxy) {
        if (NOCHANGE.includes(key)) {
            throw Error(`Error! ${key} is immutable.`);
        }
        return Reflect.set(target, key, value, proxy);
    },
    deleteProperty(target, key) {
```

```
        if (NODELETE.includes(key)) {
            throw Error(`Error! ${key} cannot be deleted.`);
        }
        return Reflect.deleteProperty(target, key);

    },
    get(target, key, proxy) {
        if (DEPRECATED.includes(key)) {
            console.warn(`Warning! ${key} is deprecated.`);
        }
        var val = target[key];

        return typeof val === 'function' ?
            function(...args) {
                Reflect.apply(target[key], target, args);
            } :
            val;
    }

});

// these will throw errors or log warnings, respectively
dataStore.doNotChange = "foo";
delete dataStore.noDelete;
dataStore.oldMethod();
```

3.5 过滤操作

某些操作会非常占用资源，比如传输大文件，这个时候如果文件已经在分块发送了，就不需要在对新的请求作出相应（非绝对），这个时候就可以使用 `Proxy` 对当请求进行特征检测，并根据特征过滤出哪些是不需要响应的，哪些是需要响应的。下面的代码简单演示了过滤特征的方式，并不是完整代码，相信大家会理解其中的妙处

```
let obj = {
    getGiantFile: function(fileId) {/*...*/ }
};

obj = new Proxy(obj, {
    get(target, key, proxy) {
        return function(...args) {
            const id = args[0];
            let isEnroute = checkEnroute(id);
            let isDownloading = checkStatus(id);
            let cached = getCached(id);

            if (isEnroute || isDownloading) {
                return false;
            }
            if (cached) {
                return cached;
            }
            return Reflect.apply(target[key], target, args);
        }
    }

});
```

87、事件代理

事件代理 也就是 事件委托

不是直接给标签添加事件 是给标签的父级添加事件 通过 事件对象 判断触发事件的标签对象是谁 执行不同的函数程序的语法形式

委托的优点

减少内存消耗
试想一下，若果我们有一个列表，列表之中有大量的列表项，我们需要在点击列表项的时候响应一个事件

如果给每个列表项一一都绑定一个函数，那对于内存消耗是非常大的，效率上需要消耗很多性能；

因此，比较好的方法就是把这个点击事件绑定到他的父层，也就是 `ul` 上，然后在执行事件的时候再去匹配判断目标元素；

所以事件委托可以减少大量的内存消耗，节约效率。

动态绑定事件
比如上述的例子中列表项就几个，我们给每个列表项都绑定了事件；

在很多时候，我们需要通过 `AJAX` 或者用户操作动态的增加或者去除列表项元素，那么在每一次改变的时候都需要重新给新增的元素绑定事件，给即将删去的元素解绑事件；

如果用了事件委托就没有这种麻烦了，因为事件是绑定在父层的，和目标元素的增减是没有关系的，执行到目标元素是在真正响应执行事件函数的过程中去匹配的；

所以使用事件在动态绑定事件的情况下是可以减少很多重复工作的。

88、不卡顿

如何在不卡住页面的情况下渲染数据，也就是说不能一次性将几万条 都渲染出来，而应该一次渲染部分 DOM，那么就可以通过 requestAnimationFrame 来 每 16 ms 刷新一次。

```
<ul>控件</ul>
 <script>
    setTimeout(() => {
       // 插入十万条数据
      const total = 100000
      // 一次插入 20 条，如果觉得性能不好就减少
      const once = 20
      // 渲染数据总共需要几次
     const loopCount = total / once
      let countOfRender = 0
      let ul = document.querySelector("ul");
      function add() {
      // 优化性能，插入不会造成回流
       const fragment = document.createDocumentFragment();
      for (let i = 0; i < once; i++) {
        const li = document.createElement("li");
        li.innerText = Math.floor(Math.random() * total);
```

```
        fragment.appendChild(li);
      }
     ul.appendChild(fragment);
     countOfRender += 1;
     loop();
  }
  function loop() {
      if (countOfRender < loopCount) {
       window.requestAnimationFrame(add);
   }
  }
  loop();
  }, 0);
```

89、JavaScript中的instanceof

JavaScript中变量的类型判断常常使用typeof运算符，但使用typeof时存在一个缺陷，就是判断引用类型存储值时，无论引用的是什么类型的对象，它都返回 object。ECMAScript 引入了另一个 Java 运算符 instanceof 来解决这个问题。instanceof 运算符与 typeof 运算符相似，用于识别正在处理的对象的类型。与 typeof 方法不同的是，instanceof 方法要求开发者明确地确认对象为某特定类型。

1.instanceof运算符用法

```
var strObj = new String("字符串");
console.log(strObj instanceof String);  // true
```

该段代码判断的是变量strObj是否为String对象的实例，strObj 是 String 对象的实例，因此是"true"。尽管不像 typeof 方法那样灵活，但是在 typeof 方法返回 “object” 的情况下，instanceof 方法就很有用。

```
// 判断 foo 是否是 Foo 类的实例
 function Foo(){}
 var foo = new Foo();

 console.log(foo instanceof Foo)
```

2.instanceof在继承关系中使用

```
// 判断 foo 是否是 Foo 类的实例 , 并且是否是其父类型的实例
function Aoo(){}
function Foo(){}
Foo.prototype = new Aoo(); //JavaScript 原型继承

var foo = new Foo();
console.log(foo instanceof Foo)//true
console.log(foo instanceof Aoo)//true
```

foo作为构造函数Foo的实例，因为构造函数Foo原型继承了构造函数Aoo，因此返回true。该代码中是判断了一层继承关系中的父类，在多层继承关系中，instanceof 运算符同样适用。

3.instanceof运算符代码

```
function instance_of(L, R) { //L 表示左表达式，R 表示右表达式
  var O = R.prototype; // 取 R 的显示原型
```

```
  L = L.__proto__; // 取 L 的隐式原型
  while (true) {
    if (L === null)
      return false;
    if (O === L) // 这里重点：当 O 严格等于 L 时，返回 true
      return true;
    L = L.__proto__;
  }
}
```

90、forEach中的await

不知道你是否写过类似的代码:

```
 function test() {
     let arr = [3, 2, 1]
     arr.forEach(async item => {
      const res = await fetch(item)
      console.log(res)
     })
     console.log('end')
    }

 function fetch(x) {
 return new Promise((resolve, reject) => {
  setTimeout(() => {
   resolve(x)
  }, 500 * x)
 })
}

test()
```

我当时期望的打印顺序是

```
3
2
1
end
结果现实与我开了个玩笑，打印顺序居然是

end
1
2
3
为什么？
```

其实原因很简单，那就是 forEach 只支持同步代码。

我们可以参考下 Polyfill 版本的 forEach，简化以后类似就是这样的伪代码

```
while (index < arr.length) {
  callback(item, index)   //也就是我们传入的回调函数
}
```

从上述代码中我们可以发现，forEach 只是简单的执行了下回调函数而已，并不会去处理异步的情况。并且你在 callback 中即使使用 break 也并不能结束遍历。

怎么解决?

一般来说解决的办法有2种,for...of和for循环。

使用 Promise.all 的方式行不行，答案是： 不行

```
async function test() {
    let arr = [3, 2, 1]
    await Promise.all(
     arr.map(async item => {
      const res = await fetch(item)
      console.log(res)
     })
    )
    console.log('end')
   }
```

可以看到并没有按照我们期望的输出。

这样可以生效的原因是 async 函数肯定会返回一个 Promise 对象，调用 map 以后返回值就是一个存放了 Promise 的数组了，这样我们把数组传入 Promise.all 中就可以解决问题了。但是这种方式其实并不能达成我们要的效果，如果你希望内部的 fetch 是顺序完成的，可以选择第二种方式。

第1种方法是使用 for...of

```
   async function test() {
     let arr = [3, 2, 1]
     for (const item of arr) {
      const res = await fetch(item)
      console.log(res)
     }
     console.log('end')
    }
```

这种方式相比 Promise.all 要简洁的多，并且也可以实现开头我想要的输出顺序。

但是这时候你是否又多了一个疑问？为啥 for...of 内部就能让 await 生效呢。

因为 for...of 内部处理的机制和 forEach 不同，forEach 是直接调用回调函数，for...of 是通过迭代器的方式去遍历。

```
async function test() {
 let arr = [3, 2, 1]
 const iterator = arr[Symbol.iterator]()
 let res = iterator.next()
 while (!res.done) {
  const value = res.value
  const res1 = await fetch(value)
  console.log(res1)
  res = iterator.next()
 }
 console.log('end')
}
```

第2种方法是使用 for循环

```
async function test() {
  let arr = [3, 2, 1]
  for (var i=0;i<arr.length;i++) {
    const res = await fetch(arr[i])
    console.log(res)
  }
  console.log('end')
}

function fetch(x) {
 return new Promise((resolve, reject) => {
  setTimeout(() => {
   resolve(x)
  }, 500 * x)
 })
}

test()
```

第3种方法是使用 while循环

```
async function test() {
  let arr = [3, 2, 1]
  var i=0;
  while(i!==arr.length){
    const res = await fetch(arr[i])
    console.log(res)
    i++;
  }
  console.log('end')
}

function fetch(x) {
 return new Promise((resolve, reject) => {
  setTimeout(() => {
   resolve(x)
  }, 500 * x)
 })
}

test()
```

要想在循环中使用async await，请使用for...of 或者 for 循环, while循环

forEach支持async await
forEach 在正常情况像下面这么写肯定是做不到同步的，程序不会等一个循环中的异步完成再进行下一个循环。原因很明显，在上面的模拟中，while 循环只是简单执行了 callback，所以尽管 callback 内使用了 await ，也只是影响到 callback 内部。

```
arr.myforeach(async v => {
    await fetch(v);
});
```

要支持上面这种写法，只要稍微改一下就好

```
Array.prototype.myforeach = async function (fn, context = null) {
    let index = 0;
    let arr = this;
    if (typeof fn !== 'function') {
        throw new TypeError(fn + ' is not a function');
    }
    while (index < arr.length) {
        if (index in arr) {
            try {
                await fn.call(context, arr[index], index, arr);
            } catch (e) {
                console.log(e);
            }
        }
        index ++;
    }
};
```

91、 src和href

src和href都是用在外部资源的引入上，比如图像，CSS文件，HTML文件，以及其他的web页面等等，那么src和href的区别都有哪些呢？

1、请求资源类型不同
（1） href是Hypertext Reference的缩写，表示超文本引用。用来建立当前元素和文档之间的链接。常用的有：link、a。
（2）在请求 src 资源时会将其指向的资源下载并应用到文档中，常用的有script，img 、iframe；

2、作用结果不同
（1）href 用于在当前文档和引用资源之间确立联系；

（2）src 用于替换当前内容；

3、 浏览器解析方式不同
（1）若在文档中添加href ，浏览器会识别该文档为 CSS 文件，就会并行下载资源并且不会停止对当前文档的处理。这也是为什么建议使用 link 方式加载 CSS，而不是使用 @import 方式。

（2）当浏览器解析到src ，会暂停其他资源的下载和处理，直到将该资源加载、编译、执行完毕，图片和框架等也如此，类似于将所指向资源应用到当前内容。这也是为什么建议把 js 脚本放在底部而不是头部的原因。

92、 JavaScript中事件绑定的方法

在JavaScript的学习中，我们经常会遇到JavaScript的事件机制，例如，事件绑定、事件监听、事件委托（事件代理）等。这些名词是什么意思呢，有什么作用呢?

一、事件绑定
要想让 JavaScript 对用户的操作作出响应，首先要对 DOM 元素绑定事件处理函数。所谓事件处理函数，就是处理用户操作的函数，不同的操作对应不同的名称。

在JavaScript中，有三种常用的绑定事件的方法：

在DOM元素中直接绑定；
在JavaScript代码中绑定；
绑定事件监听函数。

1、在DOM中直接绑定事件

我们可以在DOM元素上绑定onclick、onmouseover、onmouseout、onmousedown、onmouseup、ondblclick、onkeydown、onkeypress、onkeyup等。好多不一一列出了。如果想知道更多事件类型请查看，DOM事件 。

```
<input type="button" value="click me" onclick="hello()">


<script>
function hello(){
 alert("hello world!");
}
```

2、在JavaScript代码中绑定事件

在 JS 代码中（即 script 标签内）绑定事件可以使 JS 代码与HTML标签分离，文档结构清晰，便于管理和开发。

```
<input type="button" value="click me" id="btn">

<script>
document.getElementById("btn").onclick = function(){
 alert("hello world!");
}
```

3、使用事件监听绑定事件

绑定事件的另一种方法是用 addEventListener() 或 attachEvent() 来绑定事件监听函数。下面详细介绍，事件监听。

1）事件监听

关于事件监听，W3C规范中定义了3个事件阶段，依次是捕获阶段、目标阶段、冒泡阶段。

起初Netscape制定了JavaScript的一套事件驱动机制（即事件捕获）。随即IE也推出了自己的一套事件驱动机制（即事件冒泡）。最后W3C规范了两种事件机制，分为捕获阶段、目标阶段、冒泡阶段。IE8以前IE一直坚持自己的事件机制（前端人员一直头痛的兼容性问题），IE9以后IE也支持了W3C规范。

W3C规范

```
element.addEventListener(event, function, useCapture)
event ： （必需）事件名，支持所有 DOM事件 。
function：（必需）指定要事件触发时执行的函数。
useCapture：（可选）指定事件是否在捕获或冒泡阶段执行。true，捕获。false，冒泡。默认false。
注：IE8以下不支持。

<input type="button" value="click me" id="btn1">

<script>
document.getElementById("btn1").addEventListener("click",hello);
function hello(){
 alert("hello world!");
}
```

IE标准

```
element.attachEvent(event, function)
event：（必需）事件类型。需加“on“，例如：onclick。
function：（必需）指定要事件触发时执行的函数。
<input type="button" value="click me" id="btn2">
<script>
document.getElementById("btn2").attachEvent("onclick",hello);
function hello(){
 alert("hello world!");
}
```

2）事件监听的优点

1、可以绑定多个事件；常规的事件绑定只执行最后绑定的事件。

```
<input type="button" value="click me" id="btn3">
<input type="button" value="click me" id="btn4">

var btn3 = document.getElementById("btn3");
btn3.onclick = function(){                  //动态绑定事件
 alert("hello 1");           //不执行
}
btn3.onclick = function(){
 alert("hello 2");           //执行
}


var btn4 = document.getElementById("btn4");
btn4.addEventListener("click",hello1);      //添加事件监听器
btn4.addEventListener("click",hello2);

function hello1(){
```

```
 alert("hello 1");        //执行
}
function hello2(){
 alert("hello 2");        //执行 （顺序执行）
}
```

2、可以解除相应的绑定

```


<input type="button" value="click me" id="btn5">

<script>
var btn5 = document.getElementById("btn5");
btn5.addEventListener("click",hello1);//执行了
btn5.addEventListener("click",hello2);//不执行
btn5.removeEventListener("click",hello2);


function hello1(){
 alert("hello 1");
}
function hello2(){
 alert("hello 2");
}
```

3）封装事件监听

```


<input type="button" value="click me" id="btn5">

//绑定监听事件
function addEventHandler(target,type,fn){
 if(target.addEventListener){
 target.addEventListener(type,fn);
 }else{
 target.attachEvent("on"+type,fn);
 }
}

//移除监听事件
function removeEventHandler(target,type,fn){
 if(target.removeEventListener){
 target.removeEventListener(type,fn);
 }else{
 target.detachEvent("on"+type,fn);
 }
}

##
```

93、 git常见分支

```
git 分支命名规范
为规范开发，保持代码提交记录以及 git 分支结构清晰，方便后续维护，现规范 git 的相关操作。

主要规范两点：

git 分支命名规范

git 提交记录规范

1. git 分支命名规范
   git 分支分为集成分支、功能分支和修复分支，分别命名为 develop、feature 和 hotfix，均为单数。不可使用 features、future、hotfixes、hotfixs 等错误名称。

master（主分支，永远是可用的稳定版本，不能直接在该分支上开发）
develop（开发主分支，所有新功能以这个分支来创建自己的开发分支，该分支只做只合并操作，不能直接在该分支上开发）
feature-xxx（功能开发分支，在develop上创建分支，以自己开发功能模块命名，功能测试正常后合并到develop分支）
feature-xxx-fix(功能bug修复分支，feature分支合并之后发现bug，在develop上创建分支修复，之后合并回develop分支。PS:feature分支在申请合并之后，未合并之前还是可以提交代码的，所以feature在合并之前还可以在原分支上继续修复bug)
hotfix-xxx（紧急bug修改分支，在master分支上创建，修复完成后合并到 master）
注意事项：

一个分支尽量开发一个功能模块，不要多个功能模块在一个分支上开发。
feature 分支在申请合并之前，最好是先 pull 一下 develop 主分支下来，看一下有没有冲突，如果有就先解决冲突后再申请合并。
```

94、前端引擎模板

JavaScript随着各种神奇的实用功能库日渐丰富，而越来越受到Web开发者与设计师的追捧，例如jQuery,MooTools,Prototype等。

1) Jade

Jade是一个有着完善API和惊艳特性的JavaScript模板引擎。使用空白与缩进敏感的代码格式编写HTML页面。基于Node.js，运行在服务器端。

2) Mustache

Mustache是一个logic-less（无逻辑或轻逻辑）语法模板。可以用于组织HTML、配置文件、源代码在内的任何东西。Mustache使用JavaScript对象的值，用来扩展模板代码中的大括号标签。

3) Transparency

Transparency是一个强大的客户端模板引擎，用来将数据绑定到Web页面的BOM结构中。其模板无需特殊格式，直接完全符合HTML。直接使用JavaScript逻辑，无需新学特殊的“模板语言”。兼容IE9+、Chrome、Fx、iOS、安卓等浏览器。

4) Underscore.js

Underscore.js是一个JavaScript库，提供一系列实用的工具函数（helper）。Underscore.js仅作为额外的工具函数独立工作，不扩充（污染）任何JavaScript内建对象的本身。

5) Embeddedjs

EJS以类似PHP的JS/HTML通过标签混排的形式，帮助开发者将JavaScript和HTML部分有效分离。

6) DoTjs

最快和简洁的JavaScript模板引擎，同时用于Node.js和浏览器。

7) Handlebarsjs

一套语义化模板引擎。兼容Mustache。

8) T.js

一个用简单的JavaScript数据结构去渲染表现html/xml内容的模板引擎。

9) Dustjs

一套同时可用于浏览器或Node.js的异步模板引擎。

10) Nunjucks

Nunjucks是一套富功能的模板引擎。模板语言功能强大，支持块继承、自动转义、宏、异步控制等功能。

怎么样的模板引擎是适合前端的

前端模板引擎需要有开发时的透明性
我认为前端任何框架和工具都要有对开发的透明性，模板引擎也不例外。所谓透明性即指我在搭建好开发环境后，随手写代码随手刷新浏览器就能看到最新的效果，而不需要额外地执行任何命令或有任何的等待过程
所以一切依赖编译过程的模板引擎并不适合前端使用，编译只能是模板引擎的一个特性，而不能是使用的前提
更严格地说，使用FileWatch等手段进行文件变更检测并自动编译也不在我的考虑范围之内，因为这会造成额外的等待，像我这种手速极快的人可能编译速度跟不上
由此可以推出，前端的模板引擎应该是具备可在纯前端环境中解析使用的能力的

前端模板引擎要有良好的运行时调试能力
前端并不像后端，任何错误都可以有严格的日志记录和调用堆栈以供分析。由于用户行为的不确定性、执行环境的不确定性、各种第三方脚本的影响等，前端很难做到完全的错误处理和跟踪，这也导致前端必然存在需要直接在线上排查问题的情况
而当问题出现在模板引擎这一层时，就需要模板引擎提供良好的调试能力
一般来说，编译后生成的函数的调试能力是弱于原先手动编写的模板片断的，因为自动生成的函数基本不具备可读性和可断点跟踪性
因此在这一点上，一个供前端使用的模板引擎应该具备在特定情况下从“执行编译后函数获取HTML”换回“解析原模板再执行函数获取HTML”的模式，即应该支持在两种模式间切换
或者更好地，一个强大的前端模板引擎编译生成的函数，可以使用Source Map或其它自定义的手段直接映射回原模板片段，不过现在并没有什么模板引擎实现了这一功能

前端模板引擎要对文件合并友好
在HTTP/2普及之前，文件合并依旧是前端性能优化中的一个重要手段，模板作为文件的一部分，依旧是需要合并的
在提供编译功能的模板引擎中，我们可以使用编译的手段将模板变为JavaScript源码，再在JavaScript的基础上做文件合并
但是如果我们出于上文所说的调试能力等原因希望保留原模板片段，那就需要模板引擎本身支持模板片段合并为一个文件了
大部分仅支持将一段输入的字符串作为模板解析的引擎并不具备这一能力，他们天生并不能将一整个字符串切分为多个模板片段，因而无法支持模板片段层面上的文件合并
需要实现对文件合并的支持，最好的办法就是让模板的语法是基于“片段”的

前端模板引擎要担负XSS的防范
从安全性上来说，前端对XSS的控制是有严格要求的
我在 单页面(SPA)开发会不会比多页面有更多的安全问题？ - 张立理的回答 中有提到过，前端对XSS的防范比较合适的方法是使用“默认转义”的白名单式策略
基于此，一个合理的模板引擎是必须支持默认转义的，即所有数据的输出都默认经过escape的逻辑处理，将关键符号转为对应的HTML实体符号，以从根源上杜绝XSS的入侵路径
当然并不是所有的内容都必须经过转义的，在系统中免不了有对用户输入富文本的需求，因此需要支持特定的语法来产生无转义的输出，但时刻注意无转义输出才是特例，默认情况下必须是转义输出的

前端模板引擎要支持片段的复用
这并不是前端模板引擎的需求，事实上任何模板引擎都应该支持片段的复用，后端如Velocity、Smarty等无不拥有此功能
所谓片段复用，应该有以下几个层次的应用：
一个片段可以被引入到另一处，相当于一个变量到处用的效果
一个片段被引入时，可以向其传递不同的数据，相当于一个函数到处用的效果
一个片段可以被外部替换，但外部不提供此片段的话保持一个默认的内容，类似设计模式中的策略模式
满足第1和第2点的模板引擎并不少，而满足第3点的前端模板引擎却不多见，而后端的Razor、Smarty等都具备这一功能
话说我当时设计我们自己的模板引擎的第3个版本时，就想出了block这一个概念来实现第3点，在做完交付将近半年之后，有人告诉我说Smarty上就有这概念，顿时有种不知应该高兴还是悲伤的不知所措感。还好他并没有怀疑我直接抄了别人的功能，不然真是冤枉

前端模板引擎要支持数据输出时的处理
所谓数据输出时处理，指一个数据要在输出时做额外的转换，最常见的如字符串的trim操作，比较技术性的如markdown的转换等
诚然数据的转换完全可以在将数据交给模板引擎前就通过JavaScript的逻辑处理完，但这会导致不少有些丑陋又有些冗余的代码，对逻辑本身的复用性也会造成负面的影响
通常模板引擎对数据做额外处理会使用filter的形式实现，类似bash中的管道的逻辑。filter的实现和注册也会有不同的设计，如mustache其实注册的是fitler工厂，而另一些模板引擎则会直接注册filter本身，不同设计有不同的考量点，我们很难说谁好谁坏
但是，模板引擎支持数据的输出处理后，会另我们在编码过程中产生一个新的纠结，即哪些数据处理应该交由模板引擎的filter实现，哪些应该在交给模板引擎前由自己的逻辑逻辑实现。这个话题展开来又是一篇长长的论述，于当前的话题无关就略过吧

前端模板引擎要支持动态数据
在开发过程中，其实有不少数据并不是静态的，如EmberJS就提供了Computed Property这样的概念，Angular也有类似的东西，Backbone则可以通过重写Model的get方法来变相实现
虽然ES5在语言层面上直接提供了getter的支持，但我们在前端开发的大部分场景下依旧不会使用这一语言特性，而会选择将动态的数据封装为某种对象的get等方法
而模板引擎在将数据转为HTML片段的过程中，同样应该关注这一点，对这些动态计算的数据有良好的支持
说得更明白一些，模板引擎不应该仅仅接受纯对象（Plain Object）作为输入，而应该更开放地接受类似带有get方法的动态的数据
一个比较合理的逻辑是，如果一个对象有一个get方法（模板引擎决定这个接口），则数据通过该方法获取，其它情况下视输入的对象为纯对象（Plain Object），使用标准的属性获取逻辑

前端模板引擎要与异步流程严密结合
前端有一个很大的特点，就是到处充斥着异步的流程。由于JavaScript在浏览器提供的引擎中单线程执行的特性、大部分与IO相关的API都暴露为异步的事实，以及多数模块定义规范中模板的动态获取是异步的这一现象，注定我们无法将这个世界当作完全同步来看
一个很常见的例子是，我们有一个AMD模块存放了全局使用的常量，模板引擎需要使用这些常量。当然我们可以在使用模板引擎之前让JavaScript去异步获取这一模块，随后将常量作为数据传递给模板引擎，但这是一种业务与视图相对耦合的玩法，出于强迫症我并不觉得这是一个漂亮的设计，所以我们希望直接在模板中这么写：

{{$globals.ICP_SERIAL}}

这是我假想的一个语法，通过$globals可以使用AMD Loader获取globals这一模块，随后获取其中的ICP_SERIAL属性输出
模板引擎支持异步是一个比较具有挑战性的话题，我的计划是在我们自己的模板引擎的下一个版本中尝试实现。这其中涉及很多的技术点，比如：
模板的输出本身成了异步的方法，而不再像现在一样直接返回字符串
分析模板对异步操作的依赖，整个字符串的拼接逻辑被打断成多个异步
异步是需要等待的，且等待是未知的，从性能上考虑，是否需要考虑Stream式的输出，以便完成一段提供一段
是提供内置的固定几种异步逻辑，还是基于Promise支持任何自定义的异步逻辑，在复杂度和实用性上作出平衡
至今我还没有完全明确模板与异步结合的方式和接口，这个话题也没办法继续深入探讨了

前端模板引擎要支持不同的开发模式
前端发展至今，有很多不同的开发模式，比如：
最普通的HTML页面，使用DOMContentLoaded等事件添加逻辑，特定交互下局部刷新页面
采用传统的MVC模型进行单页式开发
使用MVVM方式以数据为核心，数据与视图方向绑定进行开发
基于Immutable Data进行数据比对Diff转DOM更新的开发（其中可能有Virtual DOM的引入）
一个模板引擎要能支持这么多种不同的的模式是一个非常大的挑战，特别是对双向绑定的支持尤为突出。至今为止几乎所有的支持双向绑定的开发框架都自带了专用的模板引擎，这是因为双向绑定对模板有两大要求：
能够从模板中提取“这一模板对哪些数据有依赖”的元信息
能够知道一个数据变化引擎的是模板的哪一块，而不至于整个刷新
而通用模板引擎很少提供这两个特性，所以没办法对不同的前端开发模式进行全面到位的支持
从模板引擎本身的实现上来说，一种方法是直接将模板解析后的类似AST的结构暴露出去，供其他框架合理地处理，同时提供对模板局部的刷新功能（也可与前面所说的模板片段一起考虑），但是大部分模板引擎为了性能等考虑，是不会解析出类似AST的语法结构来的

前端模板引擎要有实例间的隔离
在大型的前端项目，特别是单页式的项目中，会有完全未知个数的模板片段同时存在，如果这些片段是带有名称（出于复用的考虑）的，就很容易造成名称上的冲突
对于同一层级的逻辑（如大家都是业务层代码，或者大家都是控件层代码），名称冲突是可以通过一些开发时的约定来解决的。但不同层之间，由于封装性的要求，外部不应该知道一些仅内部使用的片段的名称，此时如果不幸有名称与其它层有冲突，会让情况变得比较麻烦，这类问题甚至都不容易跟踪，往往会导致大量的精力和时间的浪费
因此，一个好的模板引擎应该是多实例的，且不同实例间应该相互具备隔离性，不会出现这种不可预期的冲突
将这个话题再往深地研究，就会发现单纯的隔离是不够的，不同层间除了不冲突的需求，同样还有片段复用的需求，我们还会需要不同模板实例间可以开放一些固定的片段共享，因此模板引擎各个实例的关系是一种组合依赖但又具备基本的封装和隔离的状态

95、datalist 用法

```
<input list="browsers">
<datalist id="browsers">
  <option value="Internet Explorer">
  <option value="Firefox">
  <option value="Chrome">
  <option value="Opera">
  <option value="Safari">
</datalist>
```

这里注意绑定datalist的id给input的list属性，这样在input输入框下面就会出现列表

96、 ajax同步和异步的区别

在使用ajax请求数据的时候，通常情况下我们都是把async:true当做默认来处理，让我们的请求成为一个异步的请求。但是在某种情况下我们是需要吧async:false设置为false的，方便我们进行观察数据的走向、去处。那同步和异步有什么区别呢?

同步请求 async:false

```
$.ajax({
        async:false,
        type:"POST",
        url:"Venue.aspx?act=init",
        dataType:"html",
        success:function(result){  //function1()
            f1();
            f2();
         }
         failure:function (result) {
            alert('我在弹');
        }
        }
function2();
```

分析
这个时候ajax块发出请求后，他会等待在function1()这个地方，不会去执行function2()，直到function1()部分执行完毕。
异步请求 async:true

```
$.ajax({
        async: true, //默认为 true
        type:"POST",
        url:"./xxx/xxx/a/b.html",
        dataType:"html",
        success:function(result){  //function1()
             f1();
             f2();
        }
        failure:function (result) {
              alert('我弹');
        },
        }
function2();
```

分析
当ajax块发出请求后，他将停留function1()，等待返回结果，但同时（在这个等待过程中），function2()就可以跑起来。
总结（两者的区别）
同步的请求的时候，代码好比在排队，必须是一个挨着一个的去执行，前面的没有结束，后面的代码就处于一个阻塞的状态。
异步执行的时候，数据请求的同时，其他代码语句也可以同步执行，比如，在数据请求的时候，由于某些愿意，需要慢慢的返回请求结果，在这个时候带宽是很空闲的，那么，代码不会等到前面的数据完全请求返回就可以开始后面的代码运行。

97、JavaScript伪数组

数组

定义: 数组是一种类列表对象，它的原型中提供了遍历和修改元素的相关操作。JavaScript 数组的长度和元素类型都是非固定的。只能用整数作为数组元素的索引，而不能用字符串。对象是没有索引的，是数组的基本特征。

```
var obj = {};
var arr = [];

obj[2] = 'a';
arr[2] = 'a';

console.log(obj[2]); // => a
console.log(arr[2]); // => a
console.log(obj.length); // => undefined
console.log(arr.length); // => 3
```

obj[2]输出’a’，是因为对象就是普通的键值对存取数据
而arr[2]输出’a’ 则不同，数组是通过索引来存取数据，arr[2]之所以输出’a’，是因为数组arr索引2的位置已经存储了数据
obj.length并不具有数组的特性，并且obj没有保存属性length，那么自然就会输出undefined
而对于数组来说，length是数组的一个内置属性，数组会根据索引长度来更改length的值
为什么arr.length输出3，而不是1
在给数组添加元素时，并没有按照连续的索引添加，所以导致数组的索引不连续，那么就导致索引长度大于元素个数

伪数组

定义:

伪数组是一个对象(Object)，而真实的数组是一个数组(Array)
拥有length属性，且必须是number类型,其它属性（索引）为字符串
不具有数组所具有的方法,forEach()等,不过有Object的方法
伪数组长度不可变,真数组长度可以变
可以通过for in遍历

```
var fakeArray = {
    length: 3,
    "0": "first",
    "1": "second",
    "2": "third"
}
var arr = [1, 2, 3, 4]
```

```
// 真数组的方法来自Array.prototype
console.log(fakeArray instanceof Array) //false
console.log(arr instanceof Array) // true

Array.isArray(fakeArray) // false;
Array.isArray(arr) // true;

console.log(arr.__proto__ === Array.prototype) // true
console.log(fakeArray.__proto__ === Array.prototype) // false
console.log(fakeArray.__proto__ === Object.prototype) // true

arr.forEach(x => console.log(x)) // 1 2 3 4
fakeArray.forEach(x => console.log(x)) // fakeArray.forEach is not a function

Object.keys(fakeArray) //  ["0", "1", "2", "length"]
```

常见的伪数组有：

函数内部的 arguments
DOM 对象列表（比如通过 document.getElementsByTags 得到的列表）
jQuery 对象（比如 $(“div”) ）
伪数组是一个 Object，而真实的数组是一个 Array。
伪数组存在的意义，是可以让普通的对象也能正常使用数组的很多方法，比如：

```
使用Array.prototype.slice.call();
var arr = Array.prototype.slice.call(arguments);

Array.prototype.forEach.call(arguments, function(v) {
  // 循环arguments对象
});

// push
// some
// every
// filter
// map
// ...


使用[].slice.call()
var fakeArray = {
    length: 3,
    "0": "first",
    "1": "second",
    "2": "third"
}
var arr = [].slice.call(fakeArray)
console.log(arr) // ["first", "second", "third"]

使用ES6中的Array.from方法
var fakeArray = {
    length: 3,
    "0": "first",
    "1": "second",
    "2": "third"
    }
var arr = Array.from(fakeArray)
```

```
console.log(arr) // ["first", "second", "third"]

使用扩展运算符,也是ES6的语法
var fakeArray = document.querySelectorAll('div')
var newArr= [...fakeArray]
console.log(newArr.__proto__ === Array.prototype) // true


伪数组转换为真数组原理
Array.prototype.slice = function (start, end) {
  start = start || 0
  end = start || this.length
  const arr = []
  for (var i = start; i < end; i++) {
    arr.push(this[i])
  }
  return arr
}
```

结论
对象没有数组 Array.prototype 的属性值，类型是 Object ，而数组类型是 Array
数组是基于索引的实现， length 会自动更新，而对象是键值对
使用对象可以创建伪数组，伪数组可以正常使用数组的大部分方法

98、同源策略

```
何为同源？
域名、协议、端口完全一致即为同源。


www.juejin.com 和juejin.com
不同源，因为域名不同


www.bilibili.tv和http://www.bilibili.com
不同源，因为域名不同


http://localhost:3000 和  http://localhost:3001
不同源,因为端口不同


qq.com 和https://qq.com
不同源，因为协议不同


www.pixiv.net 和 www.pixiv.net/manage/illu…
同源，因为域名，协议，端口都相同


何为策略？
策略主要限制js的能力
1.无法读取非同源的 cookie、Storage、indexDB的内容
```

2.无法读取非同源的DOM
3.无法发送非同源的AJAX，更加准确的说应该是发送了请求但被浏览器拦截了。
为什么会有同源策略？

为了保护用户数据安全

1.为了防止恶意网页可以获取其他网站的本地数据。
2.为了防止恶意网站iframe其他网站的时候，获取数据。
3.为了防止恶意网站在自己网站有访问其他网站的权利，以免通过cookie免登，拿到数据。
跨域问题
前后端分离，和使用服务商数据时，导致前端页面地址和后端API不是同源的，例如前端地址为baidu.com,后端API为api.baidu.com。直接访问API会触发同源策略，所以需要想办法跨过去。
常见的跨域方法的原理
1.CORS
- CORS（跨域资源共享）使用专用的HTTP头，服务器（api.baidu.com）告诉浏览器，特定URL（baidu.com）的ajax请求可以直接使用，不会激活同源策略。

2.JSONP
- 这个方案相当于黑魔法，因为js调用（实际上是所有拥有src属性的 <\script>、<\img>、<\iframe>）是不会经过同源策略，例如baidu.com引用了CDN的jquery。所以我通过调用js脚本的方式，从服务器上获取JSON数据绕过同源策略。

3.nginx反向代理
- 当你访问baidu.com/api/login的时候，通过在baidu.com的nginx服务器会识别你是api下的资源，会自动代理到api.baidu.com/login，浏览器本身是不知道我实际上是访问的api.baidu.com的数据，和前端资源同源，所以也就不会触发浏览器的同源策略。

99、获取第二大的数字

方法一
将数组从大到小排序然后找第二个
当然在JS中有sort()方法可以进行数组排序

```
var arr=[5,2,10,8,0,4,7,11,9,1];
function array1(){
    var max,min;
    if(arr[0]<arr[1]){
          max=arr[1];
          min=arr[0];
    }
    else
    {
         max=arr[0];
         min=arr[1];
    }
    for(i=2;i<arr.length;i++)
    {
        if(arr[i]>min)
        {
            if(arr[i]>max)
            {
                min=max;
                max=arr[i];
            }
            else
                min=arr[i];
```

```
        }
    }
    alert(min);
}
array1();
```

方法二
定义两个变量max min循环遍历分别存储当前最大和第二大的数然后输出第二大的数min;

```
var arr=[5,2,10,8,0,4,7,11,9,1];
function array2(){
    var temp,min;
    for(var i=0;i<arr.length-1;i++){
        min=i;
        for(var j=i+1;j<arr.length;j++){
            if(arr[j]>arr[i]){
                temp= arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
    }
    alert(arr[1]);
}
array2();
```

100、forin和Object.keys的区别

使用for in 去遍历 对象会将prototype上面扩展的方法或者属性也打印出来

```
// 递归写法
Object.prototype.clone = function(){
    let o = this.constructor === Array ? [] : {};
    for(let e in this){
        o[e] = typeof this[e] === "object" ? this[e].clone() : this[e];
    }
    return o;
}
let obj = {
    a : 1,
    b : {
        c: 2
    }
}
let obj2 = obj.clone();
console.log(obj2);// { a: 1, b: { c: 2, clone: [Function] }, clone: [Function] }
```

解决方法可以为每一次的遍历加上hasOwnProperty
hasOwnProperty具体的作用就是判断该属性是否属于对象自身的属性

// 递归写法

```
Object.prototype.clone = function(){
    let o = this.constructor === Array ? [] : {};
```

```
    for(let e in this){
        if(this.hasOwnProperty(e)){
            o[e] = typeof this[e] === "object" ? this[e].clone() : this[e];
        }
    }
    return o;
}
let obj = {
    a : 1,
    b : {
        c: 2
    }
}
let obj2 = obj.clone();
console.log(obj2); // { a: 1, b: { c: 2 } }
```

也可以使用Object.keys()方式完成遍历操作

```
// 递归写法
Object.prototype.clone = function(){
    let o = this.constructor === Array ? [] : {};

    Object.keys(this).forEach(item => {
        o[item] = typeof this[item] === "object" ? this[item].clone() : this[item]
    })
    return o;

}
let obj = {
    a : 1,
    b : {
        c: 2
    }
}
let obj2 = obj.clone();
console.log(obj2);// { a: 1, b: { c: 2 } }
```