JavaScript

\1. Promise 的理解

Promise 是一种为了避免回调地狱的异步解决方案 2. Promise 是一种状态机： pending（进行中）、fulfilled（已成功）和rejected（已失败） 只有异步操作的结果，可以决定当前是哪一种状态，任何其他操作都无法改变这个状态。

回调地狱

回调函数中嵌套回调函数的情况就叫做回调地狱。
回调地狱就是为是实现代码顺序执行而出现的一种操作，它会造成我们的代码可读性非常差，后期不好维护。

一、Promise是什么?

Promise是最早由社区提出和实现的一种解决异步编程的方案，比其他传统的解决方案（回调函数和事件）更合理和更强大。

ES6 将其写进了语言标准，统一了用法，原生提供了Promise对象。
ES6 规定，Promise对象是一个构造函数，用来生成Promise实例。

二、Promise是为解决什么问题而产生的?

promise是为解决异步处理回调金字塔问题而产生的

三、Promise的两个特点

1、Promise对象的状态不受外界影响

1）pending 初始状态

2）fulfilled 成功状态

3）rejected 失败状态

Promise 有以上三种状态，只有异步操作的结果可以决定当前是哪一种状态，其他任何操作都无法改变这个状态

2、Promise的状态一旦改变，就不会再变，任何时候都可以得到这个结果，状态不可以逆，只能由 pending变成fulfilled或者由pending变成rejected

四、Promise的三个缺点

1）无法取消Promise,一旦新建它就会立即执行，无法中途取消

2）如果不设置回调函数，Promise内部抛出的错误，不会反映到外部

3）当处于pending状态时，无法得知目前进展到哪一个阶段，是刚刚开始还是即将完成

五、Promise在哪存放成功回调序列和失败回调序列?

1）**onResolvedCallbacks** 成功后要执行的回调序列 是一个数组

2）**onRejectedCallbacks** 失败后要执行的回调序列 是一个数组

以上两个数组存放在Promise 创建实例时给Promise这个类传的函数中，默认都是空数组。
每次实例then的时候 传入 onFulfilled 成功回调 onRejected 失败回调，如果此时的状态是pending 则将onFulfilled和onRejected push到对应的成功回调序列数组和失败回调序列数组中，如果此时的状态是fulfilled 则onFulfilled立即执行，如果此时的状态是rejected则onRejected立即执行

上述序列中的回调函数执行的时候 是有顺序的，即按照顺序依次执行

\2. 箭头函数和普通函数的区别

箭头函数与普通函数的区别在于：

1、箭头函数没有this，所以需要通过查找作用域链来确定this的值，这就意味着如果箭头函数被非箭头函数包含，this绑定的就是最近一层非箭头函数的this，

2、箭头函数没有自己的arguments对象，但是可以访问外围函数的arguments对象

3、不能通过new关键字调用，同样也没有new.target值和原型

1、语法更加简洁、清晰
2、箭头函数不会创建自己的this，它只会从自己的作用域链的上一层继承this。
3、箭头函数继承而来的this指向永远不变
4、.call()/.apply()/.bind()无法改变箭头函数中this的指向
5、箭头函数不能作为构造函数使用
6、箭头函数没有自己的arguments，可以在箭头函数中使用rest参数代替arguments对象，来访问箭头函数的参数列表
7、箭头函数没有原型prototype
8、箭头函数不能用作Generator函数，不能使用yeild关键字
9、箭头函数不具有super，不具有new.target

\3. ES6新特性

1、let（ let 允许创建块级作用域（最靠近的一个花括号内有效），不具备变量提升，不允许重复声明：）、const（ const 允许创建块级作用域（最靠近的一个花括号内有效）、变量声明不提升、const 在声明时必须被赋值、声明时大写变量（默认规则）： ）、block作用域

2、箭头函数 ES6 中，箭头函数就是函数的一种简写形式，使用括号包裹参数，跟随一个 =>，紧接着是函数体：

3、函数默认参数值

ES6 中允许你对函数参数设置默认值：

4、对象超类

ES6 允许在对象中使用 super 方法：

5、Map VS WeakMap

ES6 中两种新的数据结构集：Map 和 WeakMap。事实上每个对象都可以看作是一个 Map。

一个对象由多个 key-val 对构成，在 Map 中，任何类型都可以作为对象的 key，如：

6、类

ES6 中有 class 语法。值得注意是，这里的 class 不是新的对象继承模型，它只是原型链的语法糖表现形式。

函数中使用 static 关键词定义构造函数的的方法和属性：

\4. Var let const 的区别

共同点：都能声明变量

不同点：var 在ECMAScript 的所有版本中都可以使用，而const和let只能在ECMAScript6【ES2015】及更晚中使用

	var	let	const
作用域	函数作用域	块作用域	块作用域
声明提升	能	不能	不能
重复声明	能	不能	不能
全局声明时为window对象的属性	是	不是	不是

var

ECMAScript6 增加了let 和 const 之后要尽可能少使用var。因为let 和 const 申明的变量有了更加明确的作用域、声明位置以及不变的值。
优先使用const来声明变量，只在提前知道未来会修改时，再使用let。
let

因为let作用域为块作用域！！！！【得要时刻记住这一点】
不能进行条件式声明
for循环使用let来声明迭代变量不会导致迭代变量外渗透。
const

声明时得直接初始化变量，且不能修改const声明的变量的值
该限制只适用于它指向的变量的引用，如果它一个对象的，则可以修改这个对象的内部的属性。

\5. 实现继承的几种方式

原型链继承

父类的实例作为子类的原型

```
function Woman(){
}
Woman.prototype= new People();
Woman.prototype.name = 'haixia';
let womanObj = new Woman();
```

优点：

简单易于实现，父类的新增的实例与属性子类都能访问

缺点：

可以在子类中增加实例属性，如果要新增加原型属性和方法需要在new 父类构造函数的后面

无法实现多继承

创建子类实例时，不能向父类构造函数中传参数

借用构造函数继承（伪造对象、经典继承）

复制父类的实例属性给子类

```
function Woman(name){
 //继承了People
  People.call(this); //People.call(this，'wangxiaoxia');
  this.name = name || 'renbo'
}
let womanObj = new Woman();
```

优点：

解决了子类构造函数向父类构造函数中传递参数

可以实现多继承（call或者apply多个父类）

缺点：

方法都在构造函数中定义，无法复用

不能继承原型属性/方法，只能继承父类的实例属性和方法

实例继承（原型式继承）

```
function Wonman(name){
  let instance = new People();
  instance.name = name || 'wangxiaoxia';
  return instance;
}
let wonmanObj = new Wonman();
```

优点：

不限制调用方式

简单，易实现

缺点：不能多次继承

\6. Null 和 undefined 的区别

undefined和null的区别：. undefined 表示一个变量没有被声明，或者被声明 了但没有被赋值（未初始化），一个没有传入实参的形参变量的值为undefined，如果一个函数什么都不返回，则该函数默认返回undefined。. null 则表示"什么都没有"，即"空值"。. Javascript将未赋值的变量默认值设为 undefined；Javascript从来不会将变量设为 null 。. 它是用来让程序员表明某个用var声明的变量时没有值的；

\7. Call bind apply的区别

apply方法

apply接受两个参数，第一个参数是this的指向，第二个参数是函数接受的参数，以数组的形式传入，且当第一个参数为null、undefined的时候，默认指向window(在浏览器中)，使用apply方法改变this指向后原函数会立即执行，且此方法只是临时改变this指向一次。

call方法
call方法的第一个参数也是this的指向，后面传入的是一个参数列表（注意和apply传参的区别）。当一个参数为null或undefined的时候，表示指向window（在浏览器中），和apply一样，call也只是临时改变一次this指向，并立即执行。

bind方法
bind方法和call很相似，第一参数也是this的指向，后面传入的也是一个参数列表(但是这个参数列表可以分多次传入，call则必须一次性传入所有参数)，但是它改变this指向后不会立即执行，而是返回一个永久改变this指向的函数。

\8. 前端缓存的理解 或者 前端数据持久化的理解

前端缓存分为HTTP缓存和浏览器缓存

其中HTTP缓存是在HTTP请求传输时用到的缓存，主要在服务器代码上设置；而浏览器缓存则主要由前端开发在前端js上进行设置。

缓存可以说是性能优化中简单高效的一种优化方式了。一个优秀的缓存策略可以缩短网页请求资源的距离，减少延迟，并且由于缓存文件可以重复利用，还可以减少带宽，降低网络负荷。

对于一个数据请求来说，可以分为发起网络请求、后端处理、浏览器响应三个步骤。浏览器缓存可以帮助我们在第一和第三步骤中优化性能。比如说直接使用缓存而不发起请求，或者发起了请求但后端存储的数据和前端一致，那么就没有必要再将数据回传回来，这样就减少了响应数据。

强制缓存就是向浏览器缓存查找该请求结果，并根据该结果的缓存规则来决定是否使用该缓存结果的过程，强制缓存的情况主要有三种，如下：
①不存在该缓存结果和缓存标识，强制缓存失效，则直接向服务器发起请求

②存在该缓存结果和缓存标识，但该结果已失效，强制缓存失效，则使用协商缓存

③存在该缓存结果和缓存标识，且该结果尚未失效，强制缓存生效，直接返回该结果

协商缓存就是强制缓存失效后，浏览器携带缓存标识向服务器发起请求，由服务器根据缓存标识决定是否使用缓存的过程
①协商缓存生效，返回304

②协商缓存失效，返回200和请求结果

\9. 防抖和节流

防抖（debounce）
所谓防抖，就是指**触发事件后在 n 秒内函数只能执行一次，如果在 n 秒内又触发了事件，则会重新计算函数执行时间。**

非立即执行版的意思是触发事件后函数不会立即执行，而是在 n 秒后执行，如果在 n 秒内又触发了事件，则会重新计算函数执行时间。

立即执行版的意思是触发事件后函数会立即执行，然后 n 秒内不触发事件才能继续执行函数的效果。

节流（throttle）
所谓节流，就是指连续触发事件但是在 n 秒中只执行一次函数。节流会稀释函数的执行频率。

对于节流，一般有两种方式可以实现，分别是**时间戳版**和**定时器版**

\10. 闭包

1、变量作用域

要理解闭包，首先要理解 JavasSript 的特殊的变量作用域。

变量的作用域无非就两种：全局变量和局部变量。

JavasSript 语言的特别之处就在于：函数内部可以直接读取全局变量，但是在函数外部无法读取函数内部的局部变量。

注意点：在函数内部声明变量的时候，一定要使用 var 命令。如果不用的话，你实际上声明的是一个全局变量！

2、如何从外部读取函数内部的局部变量?

出于种种原因，我们有时候需要获取到函数内部的局部变量。但是，上面已经说过了，正常情况下，这是办不到的！只有通过变通的方法才能实现。

那就是在函数内部，再定义一个函数。

```
function f1(){
    var n=999;
    function f2(){
        alert(n); // 999
    }
}
```

在上面的代码中，函数 f2 就被包括在函数 f1 内部，这时 f1 内部的所有局部变量，对 f2 都是可见的。但是反过来就不行，f2 内部的局部变量，对 f1 就是不可见的。

这就是 JavasSript 语言特有的"**链式作用域**"结构（chain scope），

子对象会一级一级地向上寻找所有父对象的变量。所以，父对象的所有变量，对子对象都是可见的，反之则不成立。

既然 f2 可以读取 f1 中的局部变量，那么只要把 f2 作为返回值，我们不就可以在 f1 外部读取它的内部变量了吗！

3、闭包的概念

上面代码中的 f2 函数，就是闭包。

各种专业文献的闭包定义都非常抽象，我的理解是: **闭包就是能够读取其他函数内部变量的函数。**

由于在 JavaScript 中，只有函数内部的子函数才能读取局部变量，所以说，闭包可以简单理解成"定义在一个函数内部的函数"。

所以，在本质上，闭包是将函数内部和函数外部连接起来的桥梁。

4、闭包的用途

闭包可以用在许多地方。它的最大用处有两个，一个是前面提到的可以**读取函数内部的变量**，另一个就是**让这些变量的值始终保持在内存中**，不会在 f1 调用后被自动清除。

为什么会这样呢？原因就在于 f1 是 f2 的父函数，而 f2 被赋给了一个全局变量，这导致 f2 始终在内存中，而 f2 的存在依赖于 f1，因此 f1 也始终在内存中，不会在调用结束后，被垃圾回收机制（garbage collection）回收。

这段代码中另一个值得注意的地方，就是 “nAdd=function(){n+=1}” 这一行，首先在 nAdd 前面没有使用 var 关键字，因此 nAdd 是一个全局变量，而不是局部变量。其次，nAdd 的值是一个匿名函数（anonymous function），而这个匿名函数本身也是一个闭包，所以 nAdd 相当于是一个 setter，可以在函数外部对函数内部的局部变量进行操作。

5、使用闭包的注意点

（1）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。
（2）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

\11. 数组去重

一、利用ES6 Set去重（ES6中最常用）

```
function unique (arr) {
  return Array.from(new Set(arr))
}
var arr = [1,1,'true','true',true,true,15,15,false,false,
undefined,undefined, null,null, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{}];
console.log(unique(arr))
 //[1, "true", true, 15, false, undefined, null, NaN, "NaN", 0, "a", {}, {}]
```

不考虑兼容性，这种去重的方法代码最少。这种方法还无法去掉“{}”空对象，后面的高阶方法会添加去掉重复“{}”的方法。

二、利用for嵌套for，然后splice去重（ES5中最常用）

```
function unique(arr){
        for(var i=0; i<arr.length; i++){
            for(var j=i+1; j<arr.length; j++){
                if(arr[i]==arr[j]){         //第一个等同于第二个，splice方法删除第二个
                    arr.splice(j,1);
                    j--;
                }
            }
        }
return arr;
}
var arr = [1,1,'true','true',true,true,15,15,false,false,
undefined,undefined, null,null, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{}];
    console.log(unique(arr))
    //[1, "true", 15, false, undefined, NaN, NaN, "NaN", "a", {…}, {…}]
//NaN和{}没有去重，两个null直接消失了
```

双层循环，外层循环元素，内层循环时比较值。值相同时，则删去这个值。

三、利用indexOf去重

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

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

四、利用sort()

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

利用sort()排序方法，然后根据排序后的结果进行遍历及相邻元素比对。

六、利用includes

```
function unique(arr) {
    if (!Array.isArray(arr)) {
        console.log('type error!')
        return
```

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

七、利用hasOwnProperty

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

利用hasOwnProperty 判断是否存在对象属性

八、利用filter

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

九、利用递归去重

```
function unique(arr) {
        var array= arr;
        var len = array.length;
array.sort(function(a,b){   //排序后更加方便去重
    return a - b;
})

```

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

十、利用Map数据结构去重

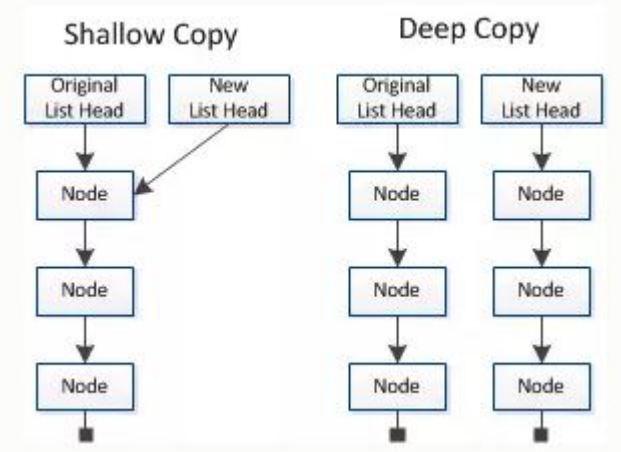
```
function arrayNonRepeatfy(arr) {
  let map = new Map();
  let array = new Array();  // 数组用于返回结果
  for (let i = 0; i < arr.length; i++) {
    if(map .has(arr[i])) {  // 如果有该key值
      map .set(arr[i], true);
    } else {
      map .set(arr[i], false);   // 如果没有该key值
      array .push(arr[i]);
    }
  }
  return array ;
}
 var arr = [1,1,'true','true',true,true,15,15,false,false,
undefined,undefined, null,null, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{}];
    console.log(unique(arr))
//[1, "a", "true", true, 15, false, 1, {…}, null, NaN, NaN, "NaN", 0, "a",
{…}, undefined]
```

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

\12. 深浅拷贝

深拷贝和浅拷贝是只针对Object和Array这样的引用数据类型的。

深拷贝和浅拷贝的示意图大致如下：



示意图

浅拷贝只复制指向某个对象的指针，而不复制对象本身，新旧对象还是共享同一块内存。但深拷贝会另外创造一个一模一样的对象，新对象跟原对象不共享内存，修改新对象不会改到原对象。

--	和原数据是否指向同一对象	第一层数据为基本数据类型	原数据中包含子对象
赋值	是	改变会使原数据一同改变	改变会使原数据一同改变
浅拷贝	否	改变不会使原数据一同改变	改变会使原数据一同改变
深拷贝	否	改变不会使原数据一同改变	改变不会使原数据一同改变

\13. 原型链

那什么是原型链呢?

简单理解就是原型组成的链，对象的**proto**它的是原型，而原型也是一个对象，也有**proto**属性，原型的**proto**又是原型的原型，就这样可以一直通过**proto**想上找，这就是原型链，当向上找找到Object的原型的时候，这条原型链就算到头了。

原型对象和实例之间有什么作用呢?

通过一个构造函数创建出来的多个实例，如果都要添加一个方法，给每个实例去添加并不是一个明智的选择。这时就该用上原型了。

在实例的原型上添加一个方法，这个原型的所有实例便都有了这个方法。

prototype:

prototype属性，它是函数所独有的，它是从一个函数指向一个对象。它的含义是函数的原型对象，也就是这个函数（其实所有函数都可以作为构造函数）所创建的实例的原型对象; 这个属性是一个指针，指向一个对象，这个对象的用途就是包含所有实例共享的属性和方法（我们把这个对象叫做原型对象）；

proto:

proto 是原型链查询中实际用到的，它总是指向 prototype，换句话说就是指向构造函数的原型对象，它是**对象独有的。**注意，为什么Foo构造也有这个属性呢，因为再js的宇宙里万物皆对象，包括函数

constructor：

我们看到途中最中间灰色模块有一个constructor属性，这个又是做什么用的呢？**
**

每个函数都有一个原型对象，该原型对象有一个constructor属性，指向创建对象的函数本身。

此外，我们还可以使用constructor属性，所有的实例对象都可以访问constructor属性，constructor属性是创建实例对象的函数的引用。我们可以使用constructor属性验证实例的原型类型（与操作符instanceof非常类似）。

\14. Require 和 import

require和import的区别

1.import在代码编译时被加载，所以必须放在文件开头，require在代码运行时被加载，所以require理论上可以运用在代码的任何地方，所以import性能更好。

2.import引入的对象被修改时，源对象也会被修改，相当于浅拷贝，require引入的对象被修改时，源对象不会被修改，官网称值拷贝，我们可以理解为深拷贝。

3.import有利于tree-shaking（移除JavaScript上下文中未引用的代码），require对tree-shaking不友好。

4.import会触发代码分割（把代码分离到不同的bundle中，然后可以按需加载或者并行加载这些文件），require不会触发。

5.import是es6的一个语法标准，如果要兼容浏览器的话必须转化成es5的语法，require 是 AMD规范引入方式。

目前所有的引擎都还没有实现import，import最终都会被转码为require，在webpack打包中，import和require都会变为*webpack_require*。