只需要记忆 *0, null, undefined, NaN, ""* 返回 *false* 就可以了，其他一律返回 *true*。

（3）ToNumber (argument)

Argument Type	Result
Undefined	Return NaN
Null	Return +0
Boolean	如果 argument 为 true, return 1. 如果 argument 为 false, return +0
Number	直接返回argument
String	将字符串中的内容转化为数字（比如"23"->23），如果转化失败则返回NaN（比如"23a"->NaN）
Symbol	抛出 TypeError 异常
Object	先**primValue = ToPrimitive(argument, Number)**，再对primValue 使用ToNumber(primValue)

@稀土掘金技术社区

ToNumber的转化并不总是成功，有时会转化成NaN，有时则直接抛出异常。

（4）ToString (argument)

Argument Type	Result
Undefined	Return "undefined"
Null	Return "null"
Boolean	如果 argument 为 true, return "true".如果 argument 为 false, return "false"
Number	用字符串来表示这个数字
String	直接返回 argument
Symbol	抛出 TypeError 异常
Object	先primValue = ToPrimitive(argument, hint String)，再对primValue使用 ToString(primValue)

@稀土掘金技术社区

当js期望得到某种类型的值，而实际在那里的值是其他的类型，就会发生隐式类型转换。系统内部会自动调用我们前面说ToBoolean (argument)、ToNumber (argument)、ToString (argument)，尝试转换成期望的数据类型。

JavaScript 面试题汇总（中篇）

101. == 隐试转换的原理？是怎么转换的

参考答案：

两个与类型转换有关的函数：valueOf()和toString()

- valueOf()的语义是，返回这个对象逻辑上对应的原始类型的值。比如说，String包装对象的valueOf()，应该返回这个对象所包装的字符串。
- toString()的语义是，返回这个对象的字符串表示。用一个字符串来描述这个对象的内容。

valueOf()和toString()是定义在Object.prototype上的方法，也就是说，所有的对象都会继承到这两个方法。但是在Object.prototype上定义的这两个方法往往不能满足我们的需求（Object.prototype.valueOf()仅仅返回对象本身），因此js的许多内置对象都重写了这两个函数，以实现更适合自身的功能需要（比如说，String.prototype.valueOf就覆盖了在Object.prototype中定义的valueOf）。当我们自定义对象的时候，最好也重写这个方法。重写这个方法时要遵循上面所说的语义。

js内部用于实现类型转换的4个函数

这4个方法实际上是ECMAScript定义的4个抽象的操作，它们在js内部使用，进行类型转换。js的使用者不能直接调用这些函数。

- ToPrimitive (input [, PreferredType])

- ToBoolean (argument)
- ToNumber (argument)
- ToString (argument)

需要区分这里的 ToString() 和上文谈到的 toString()，一个是 js 引擎内部使用的函数，另一个是定义在对象上的函数。

（1）ToPrimitive (input [, PreferredType])

将 input 转化成一个原始类型的值。PreferredType参数要么不传入，要么是Number 或 String。**如果PreferredType参数是Number**，ToPrimitive这样执行：

1. 如果input本身就是原始类型，直接返回input。
2. 调用**input.valueOf()** ，如果结果是原始类型，则返回这个结果。
3. 调用**input.toString()** ，如果结果是原始类型，则返回这个结果。
4. 抛出TypeError异常。

以下是PreferredType不为Number时的执行顺序。

- 如果PreferredType参数是String，则交换上面这个过程的第2和第3步的顺序，其他执行过程相同。
- 如果PreferredType参数没有传入
    - 如果input是内置的Date类型，PreferredType 视为String
    - 否则PreferredType 视为 Number

可以看出，ToPrimitive依赖于valueOf和toString的实现。

（2）ToBoolean (argument)

Argument Type	Result
Undefined	Return false
Null	Return false
Boolean	Return argument
Number	仅当argument为 +0, -0, or NaN时， return false; 否则一律 return true
String	仅当argument是空字符串（长度为0）时，return false; 否则一律 return true
Symbol	Return true
Object	Return true

@稀土掘金技术社区

102. ['1', '2', '3'].map(parseInt) 结果是什么，为什么 （字节）

> 参考答案：
>
> [1, NaN, NaN]

> 解析：
>
> 一、为什么会是这个结果?
>
> 1. *map* 函数
>
> 将数组的每个元素传递给指定的函数处理，并返回处理后的数组，所以 *['1','2','3'].map(parseInt)* 就是将字符串 *1，2，3* 作为元素；*0，1，2* 作为下标分别调用 *parseInt* 函数。即分别求出 *parseInt('1',0), parseInt('2',1), parseInt('3',2)* 的结果。
>
> 1. *parseInt* 函数（重点）
>
> 概念：以第二个参数为基数来解析第一个参数字符串，通常用来做十进制的向上取整（省略小数）如：parseInt(2.7) //结果为2
>
> 特点：接收两个参数 *parseInt(string,radix)*
>
> *string*：字母（大小写均可）、数组、特殊字符（不可放在开头,特殊字符及特殊字符后面的内容不做解析）的任意字符串，如 '2'、'2w'、'2!'
>
> *radix*：解析字符串的基数，基数规则如下：
>
> 1） 区间范围介于 *2~36* 之间；
>
> 2 ） 当参数为 *0*，*parseInt( )* 会根据十进制来解析；
>
> 3 ） 如果忽略该参数，默认的基数规则：
>
> 如果 *string* 以 "0x" 开头，parseInt() 会把 string 的其余部分解析为十六进制的整数；parseInt("0xf") // 15
> 如果 *string* 以 0 开头，其后的字符解析为八进制或十六进制的数字；parseInt("08") // 8
> 如果 *string* 以 1 ~ 9 的数字开头，parseInt() 将把它解析为十进制的整数；parseInt("88.99f") // 88
> 只有字符串中的第一个数字会被返回。parseInt("10.33") // 返回10；
> 开头和结尾的空格是允许的。parseInt(" 69 10 ") // 返回69
> 如果字符串的第一个字符不能被转换为数字，返回 NaN。parseInt("f") // 返回 NaN 而 parseInt("f"，16) // 返回15
>
> 二、*parseInt* 方法解析的运算过程
>
> parseInt('101.55',10); // 以十进制解析，运算过程：向上取整数(不做四舍五入，省略小数)，结果为 101。
>
> parseInt('101',2); // 以二进制解析，运算过程：1*2的2次方*+0*2*的1次方+1*2的0次方=4+0+1=5，结果为 5。
>
> parseInt('101',8); // 以八进制解析，运算过程：1*8的2次方*+0*8*的1次方+1*8的0次方=64+0+1=65，结果为 65。
>
> parseInt('101',16); // 以十六进制解析，运算过程：1*16的2次方*+0*16*的1次方+1*16的0次方=256+0+1=257，结果为 257。
>
> 三、再来分析一下结果
>
> *['1','2','3'].map(parseInt)* 即

parseInt('1',0); radix 为 0，parseInt( ) 会根据十进制来解析，所以结果为 *1*；

parseInt('2',1); radix 为 1，超出区间范围，所以结果为 *NaN*；

parseInt('3',2); radix 为 2，用2进制来解析，应以 *0* 和 *1* 开头，所以结果为 *NaN*。

103. 防抖，节流是什么，如何实现 （字节）

参考答案：

我们在平时开发的时候，会有很多场景会频繁触发事件，比如说搜索框实时发请求，*onmousemove*、*resize*、*onscroll* 等，有些时候，我们并不能或者不想频繁触发事件，这时候就应该用到函数防抖和函数节流。

函数防抖(*debounce*)，指的是短时间内多次触发同一事件，只执行最后一次，或者只执行最开始的一次，中间的不执行。

具体实现：

```
/**
 * 函数防抖
 * @param {function} func 一段时间后，要调用的函数
 * @param {number} wait 等待的时间，单位毫秒
 */
function debounce(func, wait){
    // 设置变量，记录 setTimeout 得到的 id
    let timerId = null;
    return function(...args){
        if(timerId){
            // 如果有值，说明目前正在等待中，清除它
            clearTimeout(timerId);
        }
        // 重新开始计时
        timerId = setTimeout(() => {
            func(...args);
        }, wait);
    }
}
```

函数节流(*throttle*)，指连续触发事件但是在 *n* 秒中只执行一次函数。即 *2n* 秒内执行 *2* 次... 。节流如字面意思，会稀释函数的执行频率。

具体实现：

```
function throttle(func, wait) {
    let context, args;
    let previous = 0;
    return function () {
        let now = +new Date();
        context = this;
        args = arguments;
        if (now - previous > wait) {
            func.apply(context, args);
            previous = now;
        }
    }
}
```

104. 介绍下 *Set*、*Map*、*WeakSet* 和 *WeakMap* 的区别（字节）

参考答案：

Set

- 成员唯一、无序且不重复
- 键值与键名是一致的（或者说只有键值，没有键名）
- 可以遍历，方法有 *add, delete,has*

WeakSet

- 成员都是对象
- 成员都是弱引用，可以被垃圾回收机制回收，可以用来保存 *DOM* 节点，不容易造成内存泄漏
- 不能遍历，方法有 *add, delete,has*

Map

- 本质上是健值对的集合，类似集合
- 可以遍历，方法很多，可以跟各种数据格式转换

WeakMap

- 只接受对象作为健名（*null* 除外），不接受其他类型的值作为健名
- 键名是弱引用，键值可以是任意的，键名所指向的对象可以被垃圾机制回收，此时键名是无效的
- 不能遍历，方法有 *get*、*set*、*has*、*delete*

105. *setTimeout*、*Promise*、*Async/Await* 的区别（字节）

参考答案：

事件循环中分为宏任务队列和微任务队列。

其中 *setTimeout* 的回调函数放到宏任务队列里，等到执行栈清空以后执行；

promise.then 里的回调函数会放到相应宏任务的微任务队列里，等宏任务里面的同步代码执行完再执行；

> *async* 函数表示函数里面可能会有异步方法，*await* 后面跟一个表达式，*async* 方法执行时，遇到 *await* 会立即执行表达式，然后把表达式后面的代码放到微任务队列里，让出执行栈让同步代码先执行。

106. *Promise* 构造函数是同步执行还是异步执行，那么 *then* 方法呢？（字节）

> 参考答案：
>
> *promise* 构造函数是同步执行的，*then* 方法是异步执行，*then* 方法中的内容加入微任务中。

107. 情人节福利题，如何实现一个 *new* （字节）

> 参考答案：
> 首先我们需要明白 *new* 的原理。关于 *new* 的原理，主要分为以下几步：
>
> - 创建一个空对象 。
> - 由 *this* 变量引用该对象 。
> - 该对象继承该函数的原型(更改原型链的指向) 。
> - 把属性和方法加入到 *this* 引用的对象中。
> - 新创建的对象由 *this* 引用 ， 最后隐式地返回 *this*
>
> 明白了这个原理后，我们就可以尝试来实现一个 *new* 方法，参考示例如下：

```
// 构造器函数
let Parent = function (name, age) {
[this.name](http://this.name) = name;
this.age = age;
};
Parent.prototype.sayName = function () {
console.log([this.name](http://this.name));
};
//自己定义的new方法
let newMethod = function (Parent, ...rest) {
// 1.以构造器的prototype属性为原型，创建新对象；
let child = Object.create(Parent.prototype);
// 2.将this和调用参数传给构造器执行
let result = Parent.apply(child, rest);
// 3.如果构造器没有手动返回对象，则返回第一步的对象
return typeof result === 'object' ? result : child;
};
//创建实例，将构造函数Parent与形参作为参数传入
const child = newMethod(Parent, 'echo', 26);
child.sayName() //'echo';
//最后检验，与使用new的效果相同
console.log(child instanceof Parent)//true
console.log(child.hasOwnProperty('name'))//true
console.log(child.hasOwnProperty('age'))//true
console.log(child.hasOwnProperty('sayName'))//false
```

108. 实现一个 *sleep* 函数（字节）

参考答案：

```
function sleep(delay) {
 var start = (new Date()).getTime();
 while ((new Date()).getTime() - start < delay) {
     continue;
 }
}

function test() {
 console.log('111');
 sleep(2000);
 console.log('222');
}

test()
```

这种实现方式是利用一个伪死循环阻塞主线程。因为 *JS* 是单线程的。所以通过这种方式可以实现真正意义上的 *sleep*。

109. 使用 sort() 对数组 [3, 15, 8, 29, 102, 22] 进行排序，输出结果 （字节）

参考答案：

sort 方法默认按照 *ASCII* 码来排序，如果要按照数字大小来排序，需要传入一个回调函数，如下：

```
[3, 15, 8, 29, 102, 22].sort((a,b) => {return a - b});
```

110. 实现 5.add(3).sub(2) (百度)

参考答案：

这里想要实现的是链式操作，那么我们可以考虑在 *Number* 类型的原型上添加 *add* 和 *sub* 方法，这两个方法返回新的数

示例如下：

```
Number.prototype.add = function (number) {
 if (typeof number !== 'number') {
     throw new Error('请输入数字～');
 }
 return this.valueOf() + number;
};
Number.prototype.minus = function (number) {
 if (typeof number !== 'number') {
     throw new Error('请输入数字～');
 }
 return this.valueOf() - number;
};
console.log((5).add(3).minus(2)); // 6
```

111. 给定两个数组，求交集

参考答案：

示例代码如下：

```
function intersect(nums1, nums2) {
 let i = j = 0,
     len1 = nums1.length,
     len2 = nums2.length,
     newArr = [];
 if (len1 === 0 || len2 === 0) {
     return newArr;
 }
 nums1.sort(function (a, b) {
     return a - b;
 });
 nums2.sort(function (a, b) {
     return a - b;
 });
 while (i < len1 || j < len2) {
     if (nums1[i] > nums2[j]) {
         j++;
     } else if (nums1[i] < nums2[j]) {
         i++;
     } else {
         if (nums1[i] === nums2[j]) {
             newArr.push(nums1[i]);
         }
         if (i < len1 - 1) {
             i++;
         } else {
             break;
         }
         if (j < len2 - 1) {
```

```
            j++;
        } else {
            break;
        }
    }
 }
 return newArr;
};
// 测试
console.log(intersect([3, 5, 8, 1], [2, 3]));
```

112. 为什么普通 *for* 循环的性能远远高于 *forEach* 的性能，请解释其中的原因。

参考答案：

for 循环按顺序遍历，*forEach* 使用 *iterator* 迭代器遍历

下面是一段性能测试的代码：

```
let arrs = new Array(100000);
console.time('for');
for (let i = 0; i < arrs.length; i++) {
};
console.timeEnd('for');
console.time('forEach');
arrs.forEach((arr) => {
});
console.timeEnd('forEach');

for: 2.263ms
forEach: 0.254ms
```

在10万这个级别下，`forEach`的性能是`for`的十倍

```
for: 2.263ms
forEach: 0.254ms
```

在100万这个量级下，`forEach`的性能是和`for`的一致

```
for: 2.844ms
forEach: 2.652ms
```

在1000万级以上的量级上，`forEach`的性能远远低于`for`的性能

```
for: 8.422ms
forEach: 30.328m
```

我们从语法上面来观察：

```
arr.forEach(callback(currentValue [, index [, array]])[, thisArg])
```

可以看到 *forEach* 是有回调的，它会按升序为数组中含有效值的每一项执行一次 *callback*，且除了抛出异常以外，也没有办法中止或者跳出 *forEach* 循环。那这样的话执行就会额外的调用栈和函数内的上下文。

而 *for* 循环则是底层写法，不会产生额外的消耗。

在实际业务中没有很大的数组时，*for* 和 *forEach* 的性能差距其实很小，*forEach* 甚至会优于 *for* 的时间，且更加简洁，可读性也更高，一般也会优先使用 *forEach* 方法来进行数组的循环处理。

113. 实现一个字符串匹配算法，从长度为 n 的字符串 S 中，查找是否存在字符串 T，T 的长度是 m，若存在返回所在位置。

参考答案：

```
// 完全不用 API
var getIndexOf = function (s, t) {
 let n = s.length;
 let m = t.length;
 if (!n || !m || n < m) return -1;
 for (let i = 0; i < n; i++) {
     let j = 0;
     let k = i;
     if (s[k] === t[j]) {
         k++; j++;
         while (k < n && j < m) {
             if (s[k] !== t[j]) break;
             else {
                 k++; j++;
             }
         }
         if (j === m) return i;
     }
 }
 return -1;
}

// 测试
console.log(getIndexOf("Hello World", "rl"))
```

114. 使用 *JavaScript Proxy* 实现简单的数据绑定

参考答案：

示例代码如下：

```
<body>
```

```
hello,world
<input type="text" id="model">
<p id="word"></p>
</body>
<script>
const model = document.getElementById("model")
const word = document.getElementById("word")
var obj= {};

const newObj = new Proxy(obj, {
   get: function(target, key, receiver) {
     console.log(`getting ${key}!`);
     return Reflect.get(target, key, receiver);
   },
   set: function(target, key, value, receiver) {
     console.log('setting',target, key, value, receiver);
     if (key === "text") {
       model.value = value;
       word.innerHTML = value;
     }
     return Reflect.set(target, key, value, receiver);
   }
 });

model.addEventListener("keyup",function(e){
 newObj.text = e.target.value
})
</script>
```

115. 数组里面有 *10* 万个数据，取第一个元素和第 *10* 万个元素的时间相差多少（字节）

> 参考答案：
>
> 消耗时间几乎一致，差异可以忽略不计

> 解析：
>
> - 数组可以直接根据索引取的对应的元素，所以不管取哪个位置的元素的时间复杂度都是 O(1)
> - *JavaScript* 没有真正意义上的数组，所有的数组其实是对象，其“索引”看起来是数字，其实会被转换成字符串，作为属性名（对象的 *key*）来使用。所以无论是取第 *1* 个还是取第 *10* 万个元素，都是用 *key* 精确查找哈希表的过程，其消耗时间大致相同。

116. 打印出 *1~10000* 以内的对称数

> 参考答案：

```
function isSymmetryNum(start, end) {
 for (var i = start; i < end + 1; i++) {
     var iInversionNumber = +(i.toString().split("").reverse().join(""));

     if (iInversionNumber === i && i > 10) {
         console.log(i);
     }

 }
}
isSymmetryNum(1, 10000);
```

117. 简述同步和异步的区别

参考答案：

同步意味着每一个操作必须等待前一个操作完成后才能执行。
异步意味着操作不需要等待其他操作完成后才开始执行。
在 *JavaScript* 中，由于单线程的特性导致所有代码都是同步的。但是，有些异步操作（例如：`XMLHttpRequest` 或 `setTimeout`）并不是由主线程进行处理的，他们由本机代码（浏览器 API）所控制，并不属于程序的一部分。但程序中被执行的回调部分依旧是同步的。

加分回答：

- *JavaScript* 中的同步任务是指在主线程上排队执行的任务，只有前一个任务执行完成后才能执行后一个任务；异步任务是指进入任务队列（*task queue*）而非主线程的任务，只有当任务队列通知主线程，某个异步任务可以执行了，该任务才会进入主线程中进行执行。
- *JavaScript* 的并发模型是基于 *"event loop"*。
- 像 `alert` 这样的方法回阻塞主线程，以致用户关闭他后才能继续进行后续的操作。
- *JavaScript* 主要用于和用户互动及操作 DOM，多线程的情况和异步操作带来的复杂性相比决定了他单线程的特性。
- *Web Worker* 虽然允许 *JavaScript* 创建多个线程，但子线程完全受主线程控制，且不能操作 *DOM*。因此他还是保持了单线程的特性。

118. 怎么添加、移除、复制、创建、和查找节点

参考答案：

1）创建新节点

createDocumentFragment( ) // 创建一个*DOM* 片段

createElement( ) // 创建一个具体的元素

createTextNode( ) // 创建一个文本节点

（2）添加、移除、替换、插入

appendChild( )

removeChild( )

replaceChild( )

insertBefore( ) // 在已有的子节点前插入一个新的子节点

（3）查找

getElementsByTagName( ) //通过标签名称

getElementsByName( ) // 通过元素的 *Name* 属性的值

getElementById( ) // 通过元素 *Id*，唯一性

querySelector( ) // 用于接收一个 *CSS* 选择符，返回与该模式匹配的第一个元素

querySelectorAll( ) // 用于选择匹配到的所有元素

119. 实现一个函数 *clone* 可以对 *Javascript* 中的五种主要数据类型（*Number*、*string*、 *Object*、*Array*、*Boolean*）进行复制

参考答案：

示例代码如下：

```
/**
* 对象克隆
* 支持基本数据类型及对象
* 递归方法
*/
function clone(obj) {
    var o;
    switch (typeof obj) {
        case "undefined":
            break;
        case "string":
            o = obj + "";
            break;
        case "number":
            o = obj - 0;
            break;
        case "boolean":
            o = obj;
            break;
        case "object": // object 分为两种情况 对象（Object）或数组（Array）
            if (obj === null) {
                o = null;
            } else {
                if (Object.prototype.toString.call(obj).slice(8, -1) === "Array") {
                    o = [];
                    for (var i = 0; i < obj.length; i++) {
                        o.push(clone(obj[i]));
                    }
                } else {
```

```
                    o = {};
                    for (var k in obj) {
                        o[k] = clone(obj[k]);
                    }
                }
            }
            break;
        default:
            o = obj;
            break;
    }
    return o;
}
```

120. 如何消除一个数组里面重复的元素

参考答案：

请点击2022高频前端面试题合集之JavaScript篇（上） 第2 题。

121. 写一个返回闭包的函数

参考答案：

```
function foo() {
 var i = 0;
 return function () {
     console.log(i++);
 }
}
var f1 = foo();
f1(); // 0
f1(); // 1
f1(); // 2
```

122. 使用递归完成 1 到 100 的累加

参考答案：

```
function add(x, y){
 if(x === y){
     return x;
 } else {
     return y + add(x, y-1);
 }
}

console.log(add(1, 100))
```

123. *Javascript* 有哪几种数据类型

> 参考答案：
>
> 请点击2022高频前端面试题合集之JavaScript篇（上）第 *25* 题。

124. 如何判断数据类型

> 参考答案：
>
> 请点击2022高频前端面试题合集之JavaScript篇（上）第 *69* 题。

125. console.log(1+'2')和 console.log(1-'2')的打印结果

> 参考答案：
>
> 第一个打印出 '12'，是一个 *string* 类型的值。
>
> 第二个打印出 -1，是一个 *number* 类型的值

126. *JS* 的事件委托是什么，原理是什么

> 参考答案：
>
> 事件委托，又被称之为事件代理。在 *JavaScript* 中，添加到页面上的事件处理程序数量将直接关系到页面整体的运行性能。导致这一问题的原因是多方面的。
>
> 首先，每个函数都是对象，都会占用内存。内存中的对象越多，性能就越差。其次，必须事先指定所有事件处理程序而导致的 *DOM* 访问次数，会延迟整个页面的交互就绪时间。
>
> 对事件处理程序过多问题的解决方案就是事件委托。
>
> 事件委托利用了事件冒泡，只指定一个事件处理程序，就可以管理某一类型的所有事件。例如，*click* 事件会一直冒泡到 *document* 层次。也就是说，我们可以为整个页面指定一个 *onclick* 事件处理程序，而不必给每个可单击的元素分别添加事件处理程序。

127. 如何改变函数内部的 *this* 指针的指向

> 参考答案：
>
> 可以通过 *call*、*apply*、*bind* 方法来改变 *this* 的指向，关于 *call*、*apply*、*bind* 方法的具体使用，请参阅前面 *102* 题

128. *JS* 延迟加载的方式有哪些?

> 参考答案：
>
> - *defer* 属性
> - *async* 属性
> - 使用 *jQuery* 的 *getScript*( ) 方法
> - 使用 *setTimeout* 延迟方法
> - 把 *JS* 外部引入的文件放到页面底部，来让 *JS* 最后引入

129. 说说严格模式的限制

参考答案：

什么是严格模式?

严格模式对 *JavaScript* 的语法和行为都做了一些更改，消除了语言中一些不合理、不确定、不安全之处；提供高效严谨的差错机制，保证代码安全运行；禁用在未来版本中可能使用的语法，为新版本做好铺垫。在脚本文件第一行或函数内第一行中引入"use strict"这条指令，就能触发严格模式，这是一条没有副作用的指令，老版的浏览器会将其作为一行字符串直接忽略。

例如：

```
"use strict";//脚本第一行
function add(a,b){
  "use strict";//函数内第一行
  return a+b;
}
```

进入严格模式后的限制

- 变量必须声明后再赋值
- 不能有重复的参数名，函数的参数也不能有同名属性
- 不能使用*with*语句
- 不能对只读属性赋值
- 不能使用前缀 *0*表示八进制数
- 不能删除不可删除的属性
- *eval* 不会在它的外层作用域引入变量。
- *eval*和*arguments*不能被重新赋值
- *arguments* 不会自动反应函数的变化
- 不能使用 *arguments.callee*
- 不能使用 *arguments.caller*
- 禁止 *this* 指向全局对象
- 不能使用 *fn.caller* 和 *fn.arguments* 获取函数调用的堆栈
- 增加了保留字

130. *attribute* 和 *property* 的区别是什么?

参考答案：

property 和 attribute 非常容易混淆，两个单词的中文翻译也都非常相近（property：属性，attribute：特性），但实际上，二者是不同的东西，属于不同的范畴。

- property是DOM中的属性，是JavaScript里的对象；
- attribute是HTML标签上的特性，它的值只能够是字符串；

> 简单理解，Attribute就是dom节点自带的属性，例如html中常用的id、class、title、align等。
>
> 而Property是这个DOM元素作为对象，其附加的内容，例如childNodes、firstChild等。

131. *ES6* 能写 *class* 么，为什么会出现 *class* 这种东西?

> 参考答案：
>
> 在 *ES6* 中，可以书写 *class*。因为在 *ES6* 规范中，引入了 *class* 的概念。使得 *JS* 开发者终于告别了直接使用原型对象模仿面向对象中的类和类继承时代。
>
> 但是 *JS* 中并没有一个真正的 *class* 原始类型， *class* 仅仅只是对原型对象运用语法糖。
>
> 之所以出现 *class* 关键字，是为了使 *JS* 更像面向对象，所以 *ES6* 才引入 *class* 的概念。

132. 常见兼容性问题

> 参考答案：
>
> 常见的兼容性问题很多，这里列举一些：
>
> 1. 关于获取行外样式 *currentStyle* 和 *getComputedStyle* 出现的兼容问题
>
> 我们都知道 *JS* 通过 *style* 不可以获取行外样式，如果我们需要获取行外样式就会使用这两种
>
> - IE 下：*currentStyle*
> - chrome、FF 下：*getComputedStyle* 第二个参数的作用是获取伪类元素的属性值
>
> 1. 关于“索引”获取字符串每一项出现的兼容性的问题
>
> 对于字符串也有类似于数组这样通过下标索引获取每一项的值
>
> ```
> var str = 'abcd';
> console.log(str[2]);
> ```
>
> 但是低版本的浏览器 *IE6*、*7* 不兼容
>
> 1. 关于使用 *firstChild*、*lastChild* 等，获取第一个/最后一个元素节点是产生的问题
>
> - IE6-8下： *firstChild,lastChild,nextSibling,previousSibling* 获取第一个元素节点
> - 高版本浏览器IE9+、FF、Chrome：获取的空白文本节点
>
> 1. 关于使用 *event* 对象，出现兼容性问题
>
> 在 *IE8* 及之前的版本浏览器中，*event* 事件对象是作为 *window* 对象的一个属性。
>
> 所以兼容的写法如下：
>
> ```
> function(event){
>     event = event || window.event;
> }
> ```
>
> 1. 关于事件绑定的兼容性问题

- *IE8* 以下用: attachEvent('事件名',fn);
- *FF*、*Chrome*、*IE9-10* 用: attachEventLister('事件名',fn,false);

1. 关于获取滚动条距离而出现的问题

当我们获取滚动条滚动距离时:

- *IE*、*Chrome: document.body.scrollTop*
- *FF: document.documentElement.scrollTop*

兼容处理:

var scrollTop = document.documentElement.scrollTop||document.body.scrollTop

133. 函数防抖节流的原理

参考答案:

请参阅前面第 *49*、*106* 题。

134. 原始类型有哪几种? *null* 是对象吗?

参考答案:

在 *JavaScript* 中，数据类型整体上来讲可以分为两大类：**基本类型**和**引用数据类型**

基本数据类型，一共有 *6* 种:

```
string, symbol, number, boolean, undefined, null
```

其中 *symbol* 类型是在 *ES6* 里面新添加的基本数据类型。

引用数据类型，就只有 *1* 种:

```
object
```

基本数据类型的值又被称之为原始值或简单值，而引用数据类型的值又被称之为复杂值或引用值。

关于原始类型和引用类型的区别，可以参阅第 *26* 题。

null 表示空，但是当我们使用 *typeof* 来进行数据类型检测的时候，得到的值是 *object*。

具体原因可以参阅前面第 *68* 题。

135. 为什么 *console.log(0.2+0.1==0.3) // false*

参考答案:

因为浮点数的计算存在 *round-off* 问题，也就是浮点数不能够进行精确的计算。并且:

- 不仅 *JavaScript*，所有遵循 *IEEE 754* 规范的语言都是如此;
- 在 *JavaScript* 中，所有的 *Number* 都是以 *64-bit* 的双精度浮点数存储的;

- 双精度的浮点数在这 *64* 位上划分为 *3* 段，而这 *3* 段也就确定了一个浮点数的值，*64bit* 的划分是“*1-11-52*”的模式，具体来说：
  - 就是 *1* 位最高位（最左边那一位）表示符号位，*0* 表示正，*1* 表示负；
  - *11* 位表示指数部分；
  - *52* 位表示尾数部分，也就是有效域部分

136. 说一下 *JS* 中类型转换的规则?

参考答案：

类型转换可以分为两种，**隐性转换**和**显性转换**。

1. 隐性转换

当不同数据类型之间进行相互运算，或者当对非布尔类型的数据求布尔值的时候，会发生隐性转换。

预期为数字的时候：算术运算的时候，我们的结果和运算的数都是数字，数据会转换为数字来进行计算。

类型	转换前	转换后
number	4	4
string	"1"	1
string	"abc"	NaN
string	""	0
boolean	true	1
boolean	false	0
undefined	undefined	NaN
null	null	0

预期为字符串的时候：如果有一个操作数为字符串时，使用`+`符号做相加运算时，会自动转换为字符串。

预期为布尔的时候：前面在介绍布尔类型时所提到的 9 个值会转为 false，其余转为 true

2. 显性转换

所谓显性转换，就是只程序员强制将一种类型转换为另外一种类型。显性转换往往会使用到一些转换方法。常见的转换方法如下：

- 转换为数值类型：`Number()`，`parseInt()`，`parseFloat()`
- 转换为布尔类型：`Boolean()`
- 转换为字符串类型：`toString()`，`String()`

当然，除了使用上面的转换方法，我们也可以通过一些快捷方式来进行数据类型的显性转换，如下：

- 转换字符串：直接和一个空字符串拼接，例如：`a = "" + 数据`

- 转换布尔：!!数据类型，例如：`!!"Hello"`
- 转换数值：数据*1 或 /1，例如：`"Hello * 1"`

137. 深拷贝和浅拷贝的区别？如何实现

参考答案：

- **浅拷贝**：只是拷贝了基本类型的数据，而引用类型数据，复制后也是会发生引用，我们把这种拷贝叫做浅拷贝（浅复制）

  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。
- **深拷贝**：在堆中重新分配内存，并且把源对象所有属性都进行新建拷贝，以保证深拷贝的对象的引用图不包含任何原有对象或对象图上的任何对象，拷贝后的对象与原来的对象是完全隔离，互不影响。

浅拷贝方法

1. 直接赋值
2. *Object.assign* 方法：可以把任意多个的源对象自身的可枚举属性拷贝给目标对象，然后返回目标对象。当拷贝的 *object* 只有一层的时候，是深拷贝，但是当拷贝的对象属性值又是一个引用时，换句话说有多层时，就是一个浅拷贝。
3. *ES6* 扩展运算符，当 *object* 只有一层的时候，也是深拷贝。有多层时是浅拷贝。
4. *Array.prototype.concat* 方法
5. *Array.prototype.slice* 方法
6. *jQuery* 中的 *.extend*：在 *jQuery* 中，*.extend*：在*jQuery*中，*.extend(deep,target,object1,objectN)* 方法可以进行深浅拷贝。*deep* 如过设为 *true* 为深拷贝，默认是 *false* 浅拷贝。

深拷贝方法

1. *$.extend(deep,target,object1,objectN)* ，将 *deep* 设置为 *true*
2. *JSON.parse(JSON.stringify)*：用 *JSON.stringify* 将对象转成 *JSON* 字符串，再用 *JSON.parse* 方法把字符串解析成对象，一去一来，新的对象产生了，而且对象会开辟新的栈，实现深拷贝。这种方法虽然可以实现数组或对象深拷贝，但不能处理函数。
3. 手写递归

示例代码如下：

```
function deepCopy(oldObj, newobj) {
    for (var key in oldObj) {
        var item = oldObj[key];
        // 判断是否是对象
        if (item instanceof Object) {
            if (item instanceof Function) {
                newobj[key] = oldObj[key];
            } else {
                newobj[key] = {};  //定义一个空的对象来接收拷贝的内容
                deepCopy(item, newobj[key]); //递归调用
            }
```

```
            // 判断是否是数组
        } else if (item instanceof Array) {
            newobj[key] = [];  //定义一个空的数组来接收拷贝的内容
            deepCopy(item, newobj[key]); //递归调用
        } else {
            newobj[key] = oldObj[key];
        }
    }
}
```

140. 为什么会出现 *setTimeout* 倒计时误差？如何减少

参考答案：

定时器是属于宏任务(*macrotask*) 。如果当前执行栈所花费的时间大于定时器时间，那么定时器的回调在宏任务(*macrotask*) 里，来不及去调用，所有这个时间会有误差。

141. 谈谈你对 *JS* 执行上下文栈和作用域链的理解

参考答案：

什么是执行上下文?

简而言之，执行上下文是评估和执行 JavaScript 代码的环境的抽象概念。每当 Javascript 代码在运行的时候，它都是在执行上下文中运行。

执行上下文的类型

JavaScript 中有三种执行上下文类型。

- **全局执行上下文** — 这是默认或者说基础的上下文，任何不在函数内部的代码都在全局上下文中。它会执行两件事：创建一个全局的 window 对象（浏览器的情况下），并且设置 `this` 的值等于这个全局对象。一个程序中只会有一个全局执行上下文。
- **函数执行上下文** — 每当一个函数被调用时, 都会为该函数创建一个新的上下文。每个函数都有它自己的执行上下文，不过是在函数被调用时创建的。函数上下文可以有任意多个。每当一个新的执行上下文被创建，它会按定义的顺序（将在后文讨论）执行一系列步骤。
- **Eval 函数执行上下文** — 执行在 `eval` 函数内部的代码也会有它属于自己的执行上下文。

调用栈

调用栈是解析器(如浏览器中的的javascript解析器)的一种机制，可以在脚本调用多个函数时，跟踪每个函数在完成执行时应该返回控制的点。（如什么函数正在执行，什么函数被这个函数调用，下一个调用的函数是谁）

- 当脚本要调用一个函数时，解析器把该函数添加到栈中并且执行这个函数。
- 任何被这个函数调用的函数会进一步添加到调用栈中，并且运行到它们被上个程序调用的位置。
- 当函数运行结束后，解释器将它从堆栈中取出，并在主代码列表中继续执行代码。
- 如果栈占用的空间比分配给它的空间还大，那么则会导致“栈溢出”错误。

作用域链

> 当访问一个变量时，编译器在执行这段代码时，会首先从当前的作用域中查找是否有这个标识符，如果没有找到，就会去父作用域查找，如果父作用域还没找到继续向上查找，直到全局作用域为止,，而作用域链，就是有当前作用域与上层作用域的一系列变量对象组成，它保证了当前执行的作用域对符合访问权限的变量和函数的有序访问。

142. *new* 的原理是什么？通过 *new* 的方式创建对象和通过字面量创建有什么区别?

> 参考答案：
>
> 关于 *new* 的原理，主要分为以下几步：
>
> - 创建一个空对象 。
> - 由 *this* 变量引用该对象 。
> - 该对象继承该函数的原型(更改原型链的指向) 。
> - 把属性和方法加入到 *this* 引用的对象中。
> - 新创建的对象由 *this* 引用 ， 最后隐式地返回 *this*，过程如下：
>
> ```
> var obj = {};
> obj.__proto__ = Base.prototype;
> Base.call(obj);
> ```
>
> 通过 *new* 的方式创建对象和通过字面量创建的对象，区别在于 *new* 出来的对象的原型对象为 `构造函数.prototype`，而字面量对象的原型对象为 `Object.prototype`
>
> 示例代码如下：
>
> ```
> function Computer() {}
> var c = new Computer();
> var d = {};
> console.log(c.__proto__ === Computer.prototype); // true
> console.log(d.__proto__ === Object.prototype); // true
> ```

143. *prototype* 和 *proto* 区别是什么?

> 参考答案：
>
> *prototype* 是构造函数上面的一个属性，指向实例化出来对象的原型对象。
>
> ***proto*** 是对象上面的一个隐式属性，指向自己的原型对象。

145. 取数组的最大值（*ES5*、*ES6*）

> 参考答案：

```
var arr = [3, 5, 8, 1];
// ES5 方式
console.log(Math.max.apply(null, arr)); // 8
// ES6 方式
console.log(Math.max(...arr)); // 8
```

147. *Promise* 有几种状态, *Promise* 有什么优缺点？

参考答案：

Promise 有三种状态：

pending、*fulfilled*、*rejected*(未决定，履行，拒绝)，同一时间只能存在一种状态，且状态一旦改变就不能再变。*Promise* 是一个构造函数，*promise* 对象代表一项有两种可能结果（成功或失败）的任务，它还持有多个回调，出现不同结果时分别发出相应回调。

- 初始化状态：*pending*
- 当调用 *resolve*(成功) 状态：*pengding=>fulfilled*
- 当调用 *reject*(失败) 状态：*pending=>rejected*

Promise 的优点是解决了回调地狱，缺点是代码并没有因为新方法的出现而减少，反而变得更加复杂，同时理解难度也加大。所以后面出现了 *async/await* 的异步解决方案。

148. *Promise* 构造函数是同步还是异步执行，*then* 呢？*Promise* 如何实现 *then* 处理？

参考答案：

promise 构造函数是同步执行的，*then* 方法是异步执行，*then* 方法中的内容加入微任务中。

接下来我们来看 *promise* 如何实现 *then* 的处理。

我们知道 *then* 是用来处理 *resolve* 和 *reject* 函数的回调。那么首先我们来定义 *then* 方法。

1、then方法需要两个参数，其中onFulfilled代表resolve成功的回调，onRejected代表reject失败的回调。

```
then(onFulfilled,onRejected){}
```

2、我们知道promise的状态是不可逆的，在状态发生改变后，即不可再次更改，只有状态为FULFILLED才会调用onFulfilled，状态为REJECTED调用onRejected

```
then(onFulfilled, onRejected){
 if (this.status == Promise.FULFILLED) {
     onFulfilled(this.value)
 }
 if (this.status == Promise.REJECTED) {
     onRejected(this.value)
 }
}
```

3、then方法的每个方法都不是必须的，所以我们要处理当没有传递参数时，应该设置默认值

```
then(onFulfilled,onRejected){
 if(typeof onFulfilled !=='function'){
     onFulfilled = value => value;
 }
 if(typeof onRejected  !=='function'){
     onRejected = value => value;
 }
 if(this.status == Promise.FULFILLED){
      onFulfilled(this.value)
 }
 if(this.status == Promise.REJECTED){
     onRejected(this.value)
 }
}
```

4、在执行then方法时，我们要考虑到传递的函数发生异常的情况，如果函数发生异常，我们应该让它进行错误异常处理，统一交给onRejected来处理错误

```
then(onFulfilled,onRejected){
  if(typeof onFulfilled !=='function'){
     onFulfilled = value => value;
 }
 if(typeof onRejected  !=='function'){
     onRejected = value => value;
 }
 if(this.status == Promise.FULFILLED){
     try{onFulfilled(this.value)}catch(error){ onRejected(error) }
 }
 if(this.status == Promise.REJECTED){
     try{onRejected(this.value)}catch(error){ onRejected(error) }
 }
}
```

5、但是现在我们自己封装的promise有个小问题，我们知道原生的promise中then方法都是异步执行，在一个同步任务执行之后再调用，而我们的现在的情况则是同步调用，因此我们要使用setTimeout来将onFulfilled和onRejected来做异步宏任务执行。

```
if(this.status=Promise.FULFILLED){
 setTimeout(()=>{
     try{onFulfilled(this.value)}catch(error){onRejected(error)}
 })
}
if(this.status=Promise.REJECTED){
 setTimeout(()=>{
     try{onRejected(this.value)}catch(error){onRejected(error)}
 })
}
```

现在then方法中，可以处理status为FULFILLED和REJECTED的情况，但是不能处理为pedding的情况，接下来进行几处修改。

6、在构造函数中，添加callbacks来保存pending状态时处理函数，当状态改变时循环调用

```
constructor(executor) {
  ...
this.callbacks = [];
...
}
```

7、在then方法中，当status等于pending的情况时，将待执行函数存放到callbacks数组中。

```
then(onFulfilled,onRejected){
 ...
 if(this.status==Promise.PENDING){
     this.callbacks.push({
         onFulfilled:value=>{
             try {
               onFulfilled(value);
             } catch (error) {
               onRejected(error);
             }
         }
         onRejected: value => {
         try {
           onRejected(value);
         } catch (error) {
           onRejected(error);
         }
       }
     })
 }
 ...
}
```

8、当执行resolve和reject时，在堆callacks数组中的函数进行执行

```
resolve(vale){
 if(this.status==Promise.PENDING){
     this.status = Promise.FULFILLED;
     this.value = value;
     this.callbacks.map(callback => {
       callback.onFulfilled(value);
     });
 }
}
reject(value){
 if(this.status==Promise.PENDING){
     this.status = Promise.REJECTED;
     this.value = value;
     this.callbacks.map(callback => {
       callback.onRejected(value);
     });
 }
}
```

9、then方法中，关于处理pending状态时，异步处理的方法：只需要将resolve与reject执行通过setTimeout定义为异步任务

```
resolve(value) {
if (this.status == Promise.PENDING) {
  this.status = Promise.FULFILLED;
  this.value = value;
 setTimeout(() => {
   this.callbacks.map(callback => {
     callback.onFulfilled(value);
   });
 });
}
}
reject(value) {
if (this.status == Promise.PENDING) {
  this.status = Promise.REJECTED;
 this.value = value;
 setTimeout(() => {
   this.callbacks.map(callback => {
     callback.onRejected(value);
   });
 });
}
}
```

到此，promise的then方法的基本实现就结束了。

149. *Promise* 和 *setTimeout* 的区别？

参考答案：

JavaScript 将异步任务分为 *MacroTask*（宏任务） 和 *MicroTask*（微任务），那么它们区别何在呢?

1. 依次执行同步代码直至执行完毕；
2. 检查MacroTask 队列，若有触发的异步任务，则取第一个并调用其事件处理函数，然后跳至第三步，若没有需处理的异步任务，则直接跳至第三步；
3. 检查MicroTask队列，然后执行所有已触发的异步任务，依次执行事件处理函数，直至执行完毕，然后跳至第二步，若没有需处理的异步任务中，则直接返回第二步，依次执行后续步骤；
4. 最后返回第二步，继续检查MacroTask队列，依次执行后续步骤；
5. 如此往复，若所有异步任务处理完成，则结束；

Promise 是一个微任务，主线程是一个宏任务，微任务队列会在宏任务后面执行

setTimeout 返回的函数是一个新的宏任务，被放入到宏任务队列

所以 *Promise* 会先于新的宏任务执行

150. 如何实现 *Promise.all*？

参考答案：

`Promise.all` 接收一个 `promise` 对象的数组作为参数，当这个数组里的所有 `promise` 对象全部变为 `resolve` 或 有 `reject` 状态出现的时候，它才会去调用 `.then` 方法,它们是并发执行的。

总结 `promise.all` 的特点

1、接收一个 `Promise` 实例的数组或具有 `Iterator` 接口的对象，

2、如果元素不是 `Promise` 对象，则使用 `Promise.resolve` 转成 `Promise` 对象

3、如果全部成功，状态变为 `resolved`，返回值将组成一个数组传给回调

4、只要有一个失败，状态就变为 `rejected`，返回值将直接传递给回调
`all()` 的返回值也是新的 `Promise` 对象

实现 `Promise.all` 方法

```
function promiseAll(promises) {
 return new Promise(function (resolve, reject) {
     if (!isArray(promises)) {
         return reject(new TypeError('arguments must be an array'));
     }
     var resolvedCounter = 0;
     var promiseNum = promises.length;
     var resolvedValues = new Array(promiseNum);
     for (var i = 0; i < promiseNum; i++) {
         (function (i) {
             Promise.resolve(promises[i]).then(function (value) {
```

```
                resolvedCounter++
                resolvedValues[i] = value
                if (resolvedCounter == promiseNum) {
                    return resolve(resolvedValues)
                }
            }, function (reason) {
                return reject(reason)
            })
        })(i)
    }
 })
}
```

151. 如何实现 *Promise.finally* ?

参考答案：

finally 方法是 *ES2018* 的新特性

finally 方法用于指定不管 *Promise* 对象最后状态如何，都会执行的操作，执行 *then* 和 *catch* 后，都会执行 *finally* 指定的回调函数。

方法一：借助 *promise.prototype.finally* 包

```
npm install promise-prototype-finally
const promiseFinally = require('promise.prototype.finally');

// 向 Promise.prototype 增加 finally()
promiseFinally.shim();

// 之后就可以按照上面的使用方法使用了
```

方法二：实现 *Promise.finally*

```
Promise.prototype.finally = function (callback) {
let P = this.constructor;
return this.then(
 value  => P.resolve(callback()).then(() => value),
 reason => P.resolve(callback()).then(() => { throw reason })
);
};
```

152. 如何判断 *img* 加载完成

参考答案：

- 为 *img DOM* 节点绑定 *load* 事件
- *readystatechange* 事件：*readyState* 为 *complete* 和 *loaded* 则表明图片已经加载完毕。测试 *IE6-IE10* 支持该事件，其它浏览器不支持。

- *img* 的 *complete* 属性：轮询不断监测 *img* 的 *complete* 属性，如果为 *true* 则表明图片已经加载完毕，停止轮询。该属性所有浏览器都支持。

153. 如何阻止冒泡?

参考答案：

```
// 方法一：IE9+，其他主流浏览器
event.stopPropagation()
// 方法二：火狐未实现
event.cancelBubble = true;
// 方法三：不建议滥用，jq 中可以同时阻止冒泡和默认事件
return false;
```

154. 如何阻止默认事件?

参考答案：

```
// 方法一：全支持
event.preventDefault();
// 方法二：该特性已经从 Web 标准中删除，虽然一些浏览器目前仍然支持它，但也许会在未来的某个时间停止支持，请尽量不要使用该特性。
event.returnValue=false;
// 方法三：不建议滥用，jq 中可以同时阻止冒泡和默认事件
return false;
```

155. 如何用原生 *js* 给一个按钮绑定两个 *onclick* 事件?

参考答案：

使用 *addEventListener* 方法来绑定事件，就可以绑定多个同种类型的事件。

156. 拖拽会用到哪些事件

参考答案：

在以前，书写一个拖拽需要用到 *mousedown*、*mousemove*、*mouseup* 这 *3* 个事件。

HTML5 推出后，新推出了一组拖拽相关的 *API*，涉及到的事件有 *dragstart*、*dragover*、*drop* 这 *3* 个事件。

157. *document.write* 和 *innerHTML* 的区别

参考答案：

document.write 是直接写入到页面的内容流，如果在写之前没有调用 *document.open*, 浏览器会自动调用 *open*。每次写完关闭之后重新调用该函数，会导致页面全部重绘。

innerHTML 则是 *DOM* 页面元素的一个属性，代表该元素的 *html* 内容。你可以精确到某一个具体的元素来进行更改。如果想修改 *document* 的内容，则需要修改 *document.documentElement.innerElement*。

innerHTML 很多情况下都优于 *document.write*，其原因在于不会导致页面全部重绘。

158. *jQuery* 的事件委托方法 *bind* 、*live*、*delegate*、*one*、*on* 之间有什么区别?

> 参考答案:
>
> 这几个方法都可以实现事件处理。其中 *on* 集成了事件处理的所有功能，也是目前推荐使用的方法。
>
> *one* 是指添加的是一次性事件，意味着只要触发一次该事件，相应的处理方法执行后就自动被删除。
>
> *bind* 是较早版本的绑定事件的方法，现在已被 *on* 替代。
>
> *live* 和 *delegate* 主要用来做事件委托。*live* 的版本较早，现在已被废弃。*delegate* 目前仍然可用，不过也可用 *on* 来替代它。

159. *$(document).ready* 方法和 *window.onload* 有什么区别?

> 参考答案:
>
> 主要有两点区别:
>
> 1. 执行时机
>
> *window.onload* 方法是在网页中的所有的元素（包括元素的所有关联文件）都完全加载到浏览器之后才执行。而通过 *jQuery* 中的`$(document).ready`方法注册的事件处理程序，只要在 *DOM* 完全就绪时，就可以调用了，比如一张图片只要`<img>`标签完成，不用等这个图片加载完成，就可以设置图片的宽高的属性或样式等。
>
> 其实从二者的英文字母可以大概理解上面的话，*onload* 即加载完成，*ready* 即 *DOM* 准备就绪。
>
> 1. 注册事件
>
> `$(document).ready`方法可以多次使用而注册不同的事件处理程序，而 *window.onload* 一次只能保存对一个函数的引用，多次绑定函数只会覆盖前面的函数。

160. jquery 中.get()提交和.get()提交和.post()提交有区别吗?

> 参考答案:
>
> 相同点：都是异步请求的方式来获取服务端的数据
>
> 不同点:
>
> - 请求方式不同：`$.get()` 方法使用 *GET* 方法来进行异步请求的。`$.post()` 方法使用 *POST* 方法来进行异步请求的。
> - 参数传递方式不同： *GET* 请求会将参数跟在 *URL* 后进行传递，而 *POST* 请求则是作为 *HTTP* 消息的实体内容发送给 *Web* 服务器 的，这种传递是对用户不可见的。
> - 数据传输大小不同： *GET* 方式传输的数据大小不能超过 *2KB* 而 *POST* 要大的多
> - 安全问题： *GET* 方式请求的数据会被浏览器缓存起来，因此有安全问题。

161. *await async* 如何实现 （阿里）

参考答案：

async 函数只是 promise 的语法糖，它的底层实际使用的是 generator，而 generator 又是基于 promise 的。实际上，在 babel 编译 async 函数的时候，也会转化成 generatora 函数，并使用自动执行器来执行它。

实现代码示例：

```
function asyncToGenerator(generatorFunc) {
 return function() {
   const gen = generatorFunc.apply(this, arguments)
   return new Promise((resolve, reject) => {
     function step(key, arg) {
       let generatorResult
       try {
         generatorResult = gen[key](arg)
       } catch (error) {
         return reject(error)
       }
       const { value, done } = generatorResult
       if (done) {
         return resolve(value)
       } else {
         return Promise.resolve(value).then(val => step('next', val), err =>
step('throw', err))
       }
     }
     step("next")
   })
 }
}
```

关于代码的解析，可以参阅：blog.csdn.net/xgangzai/ar...

162. *clientWidth,offsetWidth,scrollWidth* 的区别

参考答案：

clientWidth = width+左右 padding

offsetWidth = width + 左右 padding + 左右 boder

scrollWidth：获取指定标签内容层的真实宽度(可视区域宽度+被隐藏区域宽度)。

163. 产生一个不重复的随机数组

> 参考答案：
>
> 示例代码如下：

```
// 生成随机数
function randomNumBoth(Min, Max) {
 var Range = Max - Min;
 var Rand = Math.random();
 var num = Min + Math.round(Rand * Range); //四舍五入
 return num;
}
// 生成数组
function randomArr(len, min, max) {
 if ((max - min) < len) { //可生成数的范围小于数组长度
     return null;
 }
 var hash = [];

 while (hash.length < len) {
     var num = randomNumBoth(min, max);

     if (hash.indexOf(num) == -1) {
         hash.push(num);
     }
 }
 return hash;
}
// 测试
console.log(randomArr(10, 1, 100));
```

> 在上面的代码中，我们封装了一个 *randomArr* 方法来生成这个不重复的随机数组，该方法接收三个参数，*len*、*min* 和 *max*，分别表示数组的长度、最小值和最大值。*randomNumBoth* 方法用来生成随机数。

164. *continue* 和 *break* 的区别

> 参考答案：
>
> - *break*：用于永久终止循环。即不执行本次循环中 *break* 后面的语句，直接跳出循环。
> - *continue*：用于终止本次循环。即本次循环中 *continue* 后面的代码不执行，进行下一次循环的入口判断。

165. 如何在 *jquery* 上扩展插件，以及内部原理（腾讯）

> 参考答案：
>
> 通过 *$.extend(object);* 为整个 *jQuery* 类添加新的方法。
>
> 例如：

```
$.extend({
   sayHello: function(name) {
      console.log('Hello,' + (name ? name : 'World') + '!');
  },
  showAge(){
      console.log(18);
  }
})

// 外部使用
$.sayHello(); // Hello,World!  无参调用
$.sayHello('zhangsan'); // Hello,zhangsan! 带参调用
```

通过 *$.fn.extend(object);* 给 *jQuery* 对象添加方法。

例如：

```
$.fn.extend({
 swiper: function (options) {
     var obj = new Swiper(options, this); // 实例化 Swiper 对象
     obj.init(); // 调用对象的 init 方法
 }
})

// 外部使用
$('#id').swiper();
```

***extend* 方法内部原理**

```
jQuery.extend( target [, object1 ] [, objectN ] )
```

对后一个参数进行循环，然后把后面参数上所有的字段都给了第一个字段，若第一个参数里有相同的字段，则进行覆盖操作，否则就添加一个新的字段。

解析如下：

```
// 为与源码的下标对应上，我们把第一个参数称为第0个参数，依次类推
jQuery.extend = jQuery.fn.extend = function() {
 var options, name, src, copy, copyIsArray, clone,
     target = arguments[0] || {}, // 默认第0个参数为目标参数
     i = 1,    // i表示从第几个参数凯斯想目标参数进行合并，默认从第1个参数开始向第0个参数进行合并
     length = arguments.length,
     deep = false;  // 默认为浅度拷贝

 // 判断第0个参数的类型，若第0个参数是boolean类型，则获取其为true还是false
 // 同时将第1个参数作为目标参数，i从当前目标参数的下一个
 // Handle a deep copy situation
```

```
 if ( typeof target === "boolean" ) {
     deep = target;

     // Skip the boolean and the target
     target = arguments[ i ] || {};
     i++;
 }

 //  判断目标参数的类型，若目标参数既不是object类型，也不是function类型，则为目标参数重新赋值
 // Handle case when target is a string or something (possible in deep copy)
 if ( typeof target !== "object" && !jQuery.isFunction(target) ) {
     target = {};
 }

 // 若目标参数后面没有参数了，如$.extend({_name:'wenzi'}), $.extend(true,
{_name:'wenzi'})
 // 则目标参数即为jQuery本身，而target表示的参数不再为目标参数
 // Extend jQuery itself if only one argument is passed
 if ( i === length ) {
     target = this;
     i--;
 }

 // 从第i个参数开始
 for ( ; i < length; i++ ) {
     // 获取第i个参数，且该参数不为null，
     // 比如$.extend(target, {}, null);中的第2个参数null是不参与合并的
     // Only deal with non-null/undefined values
     if ( (options = arguments[ i ]) != null ) {

         // 使用for~in获取该参数中所有的字段
         // Extend the base object
         for ( name in options ) {
             src = target[ name ];   // 目标参数中name字段的值
             copy = options[ name ]; // 当前参数中name字段的值

             // 若参数中字段的值就是目标参数，停止赋值，进行下一个字段的赋值
             // 这是为了防止无限的循环嵌套，我们把这个称为，在下面进行比较详细的讲解
             // Prevent never-ending loop
             if ( target === copy ) {
                 continue;
             }

             // 若deep为true，且当前参数中name字段的值存在且为object类型或Array类型，则进行
深度赋值
             // Recurse if we're merging plain objects or arrays
             if ( deep && copy && ( jQuery.isPlainObject(copy) || (copyIsArray =
jQuery.isArray(copy)) ) ) {
                 // 若当前参数中name字段的值为Array类型
```

```
                    // 判断目标参数中name字段的值是否存在，若存在则使用原来的，否则进行初始化
                    if ( copyIsArray ) {
                        copyIsArray = false;
                        clone = src && jQuery.isArray(src) ? src : [];

                    } else {
                        // 若原对象存在，则直接进行使用，而不是创建
                        clone = src && jQuery.isPlainObject(src) ? src : {};
                    }

                    // 递归处理，此处为2.2
                    // Never move original objects, clone them
                    target[ name ] = jQuery.extend( deep, clone, copy );

                // deep为false，则表示浅度拷贝，直接进行赋值
                // 若copy是简单的类型且存在值，则直接进行赋值
                // Don't bring in undefined values
                } else if ( copy !== undefined ) {
                    // 若原对象存在name属性，则直接覆盖掉；若不存在，则创建新的属性
                    target[ name ] = copy;
                }
            }
        }
    }

    // 返回修改后的目标参数
    // Return the modified object
    return target;
};
```

166. *async/await* 如何捕获错误

参考答案：

可以使用 *try...catch* 来进行错误的捕获

示例代码：

```
async function test() {
    try {
        const res = await test1()
    } catch (err) {
        console.log(err)
    }
    console.log("test")
}
```

167. *Proxy* 对比 *Object.defineProperty* 的优势

> 参考答案：
>
> **Proxy 的优势如下:**
>
> - *Object.defineProperty* 只能劫持对象的属性,因此我们需要对每个对象的每个属性进行遍历，而 *Proxy* 可以直接监听对象而非属性；
> - *Object.defineProperty* 无法监控到数组下标的变化，而 *Proxy* 可以直接监听数组的变化；
> - *Proxy* 有多达 *13* 种拦截方法；
> - *Proxy* 作为新标准将受到浏览器厂商重点持续的性能优化；

168. 原型链，可以改变原型链的规则吗?

> 参考答案：
>
> 每个对象都可以有一个原型***proto***，这个原型还可以有它自己的原型，以此类推，形成一个原型链。查找特定属性的时候，我们先去这个对象里去找，如果没有的话就去它的原型对象里面去，如果还是没有的话再去向原型对象的原型对象里去寻找。这个操作被委托在整个原型链上，这个就是我们说的原型链。
>
> 我们可以通过手动赋值的方式来改变原型链所对应的原型对象。

170. *JS* 基本数据类型有哪些？栈和堆有什么区别，为什么要这样存储。（快手）

> 参考答案：
>
> 关于 *JS* 基本数据类型有哪些这个问题，可以参阅前面 *26* 题。
>
> 栈和堆的区别在于堆是动态分配内存，内存大小不一，也不会自动释放。栈是自动分配相对固定大小的内存空间，并由系统自动释放。
>
> 在 *js* 中，基本数据都是直接按值存储在栈中的，每种类型的数据占用的内存空间的大小是确定的，并由系统自动分配和自动释放。这样带来的好处就是，内存可以及时得到回收，相对于堆来说，更加容易管理内存空间。
>
> *js* 中其他类型的数据被称为引用类型的数据（如对象、数组、函数等），它们是通过拷贝和 *new* 出来的，这样的数据存储于堆中。其实，说存储于堆中，也不太准确，因为，引用类型的数据的地址指针是存储于栈中的，当我们想要访问引用类型的值的时候，需要先从栈中获得对象的地址指针，然后，在通过地址指针找到堆中的所需要的数据。

171. *setTimeout(() => {}, 0)* 什么时候执行

> 参考答案：
>
> 因为 *setTimeout* 是异步代码，所以即使后面的时间为 *0*，也要等到同步代码执行完毕后才会执行。

172. *js* 有函数重载吗（网易）

> 参考答案：
>
> 所谓函数重载，是方法名称进行重用的一种技术形式，其主要特点是“方法名相同，参数的类型或个数不相同”，在调用时会根据传递的参数类型和个数的不同来执行不同的方法体。
>
> 在 *JS* 中，可以通过在函数内容判断形参的类型或个数来执行不同的代码块，从而达到模拟函数重载的效果。

173. 给你一个数组，计算每个数出现的次数，如果每个数组返回的数都是独一无二的就返回 *true* 相反则返回的 *flase*

> 参考答案：
>
> 输入：arr = [1,2,2,1,1,3]
>
> 输出：true
>
> 解释：在该数组中，1 出现了 3 次，2 出现了 2 次，3 只出现了 1 次。没有两个数的出现次数相同。
>
> 代码示例：

```
function uniqueOccurrences(arr) {
 let uniqueArr = [...new Set(arr)]
 let countArr = []
 for (let i = 0; i < uniqueArr.length; i++) {
     countArr.push(arr.filter(item => item == uniqueArr[i]).length)
 }
 return countArr.length == new Set(countArr).size
};

// 测试
console.log(uniqueOccurrences([1, 2, 2, 1, 1, 3])); // true
console.log(uniqueOccurrences([1, 2, 2, 1, 1, 3, 2])); // false
```

174. 封装一个能够统计重复的字符的函数，例如 *aaabbbdddddfff* 转化为 *3a3b5d3f*

> 参考答案：

```
function compression(str) {
 if (str.length == 0) {
     return 0;
 }
 var len = str.length;
 var str2 = "";
 var i = 0;
 var num = 1;
 while (i < len) {
     if (str.charAt(i) == str.charAt(i + 1)) {
```

```
            num++;
        } else {
            str2 += num;
            str2 += str.charAt(i);
            num = 1;
        }
        i++;
    }
    return str2;
}
// 测试:
console.log(compression('aaabbbdddddfff')); // 3a3b5d3f
```

175. 写出代码的执行结果，并解释为什么?

```
function a() {
    console.log(1);
}
(function() {
    if (false) {
        function a() {
            console.log(2);
        }
    }
    console.log(typeof a);
    a();
})()
```

> 参考答案:
>
> 会报错，*a is not a function*。
>
> 因为立即执行函数里面有函数 *a*，*a* 会被提升到该函数作用域的最顶端，但是由于判断条件是 *false*，所以不会进入到条件语句里面， *a* 也就没有值。所以 *typeof* 打印出来是 *undefined*。而后面在尝试调用方法，自然就会报错。

176. 写出代码的执行结果，并解释为什么?

```
alert(a);
a();
var a = 3;
function a() {
  alert(10);
};
alert(a);
a = 6;
a();
```

> 参考答案：
>
> 首先打印 function a() {alert(10);};
>
> 然后打印 10
>
> 最后打印 3

> 解析：
>
> 首先 a 变量会被提升到该全局作用域的最顶端，然后值为对应的函数，所以第一次打印出来的是函数。
>
> 接下来调用这个 a 函数，所以打印出 10
>
> 最后给这个 a 赋值为 3，然后又 alert，所以打印出 3。
>
> 之后 a 的值还会发生改变，但是由于没有 alert，说明不会再打印出其他值了。

177. 写出下面程序的打印顺序，并简要说明原因

```
setTimeout(function () {
    console.log("set1");
    new Promise(function (resolve) {
        resolve();
    }).then(function () {
        new Promise(function (resolve) {
            resolve();
        }).then(function () {
            console.log("then4");
        })
        console.log('then2');
    })
});
new Promise(function (resolve) {
    console.log('pr1');
    resolve();
}).then(function () {
    console.log('then1');
});

setTimeout(function () {
    console.log("set2");
});
console.log(2);

new Promise(function (resolve) {
    resolve();
}).then(function () {
    console.log('then3');
})
```

> 参考答案：

> 打印结果为：
>
> pr1
> 2
> then1
> then3
> set1
> then2
> then4
> set2

178. *javascript* 中什么是伪数组？如何将伪数组转换为标准数组

> 参考答案：
>
> 在 *JavaScript* 中，*arguments* 就是一个伪数组对象。关于 *arguments* 具体可以参阅后面 *250* 题。
>
> 可以使用 *ES6* 的扩展运算符来将伪数组转换为标准数组
>
> 例如：

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

179. *array* 和 *object* 的区别

> 参考答案：
>
> 数组表示有序数据的集合，对象表示无序数据的集合。如果数据顺序很重要的话，就用数组，否则就用对象。

180. *jquery* 事件委托

> 参考答案：
>
> 在 *jquery* 中使用 *on* 来绑定事件的时候，传入第二个参数即可。例如：

```
$("ul").on("click","li",function () {
alert(1);
})
```

182. 请实现一个模块 *math*，支持链式调用`math.add(2,4).minus(3).times(2);`

> 参考答案：
>
> 示例代码：

```
class Math {
 constructor(value) {
     let hasInitValue = true;
```

```
    if (value === undefined) {
        value = NaN;
        hasInitValue = false;
    }
    Object.defineProperties(this, {
        value: {
            enumerable: true,
            value: value,
        },
        hasInitValue: {
            enumerable: false,
            value: hasInitValue,
        },
    });
}

add(...args) {
    const init = this.hasInitValue ? this.value : args.shift();
    const value = args.reduce((pv, cv) => pv + cv, init);
    return new Math(value);
}

minus(...args) {
    const init = this.hasInitValue ? this.value : args.shift();
    const value = args.reduce((pv, cv) => pv - cv, init);
    return new Math(value);
}

times(...args) {
    const init = this.hasInitValue ? this.value : args.shift();
    const value = args.reduce((pv, cv) => pv * cv, init);
    return new Math(value);
}

divide(...args) {
    const init = this.hasInitValue ? this.value : args.shift();
    const value = args.reduce((pv, cv) => pv / cv, init);
    return new Math(value);
}

toJSON() {
    return this.valueOf();
}

toString() {
    return String(this.valueOf());
}

valueOf() {
```

```
        return this.value;
    }

    [Symbol.toPrimitive](hint) {
        const value = this.value;
        if (hint === 'string') {
            return String(value);
        } else {
            return value;
        }
    }
}

export default new Math();
```

183. 请简述 *ES6* 代码转成 *ES5* 代码的实现思路。

> 参考答案：
>
> 说到 *ES6* 代码转成 *ES5* 代码，我们肯定会想到 *Babel*。所以，我们可以参考 *Babel* 的实现方式。
>
> 那么 *Babel* 是如何把 *ES6* 转成 *ES5* 呢，其大致分为三步：
>
> - 将代码字符串解析成抽象语法树，即所谓的 *AST*
> - 对 *AST* 进行处理，在这个阶段可以对 *ES6* 代码进行相应转换，即转成 *ES5* 代码
> - 根据处理后的 *AST* 再生成代码字符串

184. 下列代码的执行结果

```
async function async1() {
    console.log('async1 start');
    await async2();
    console.log('async1 end');
}
async function async2() {
    console.log('async2');
}
console.log('script start');
setTimeout(function () {
    console.log('setTimeout');
}, 0);
async1();
new Promise(function (resolve) {
    console.log('promise1');
    resolve();
}).then(function () {
    console.log('promise2');
});
console.log('script end');
```

参考答案：

script start
async1 start
async2
promise1
script end
async1 end
promise2
setTimeout

解析：

在此之前我们需要知道以下几点：

- setTimeout 属于宏任务
- Promise 本身是同步的立即执行函数，Promise.then 属于微任务
- async 方法执行时，遇到 await 会立即执行表达式，表达式之后的代码放到微任务执行

第一次执行：执行同步代码

```
Tasks(宏任务)：run script、 setTimeout callback
Microtasks(微任务)：await、Promise then
JS stack(执行栈)：script
Log：script start、async1 start、async2、promise1、script end
```

第二次执行：执行宏任务后，检测到**微任务**队列中不为空、一次性执行完所有微任务

```
Tasks(宏任务)：run script、 setTimeout callback
Microtasks(微任务)：Promise then
JS stack(执行栈)：await
Log：script start、async1 start、async2、promise1、script end、async1 end、promise2
```

第三次执行：当**微任务**队列中为空时，执行**宏任务**，执行`setTimeout callback`，打印日志。

```
Tasks(宏任务)：null
Microtasks(微任务)：null
JS stack(执行栈):setTimeout callback
Log：script start、async1 start、async2、promise1、script end、async1
end、promise2、setTimeout
```

185. *JS* 有哪些内置对象?

参考答案：

数据封装类对象：*String*，*Boolean*，*Number*，*Array* 和 *Object*

其他对象：*Function*，*Arguments*，*Math*，*Date*，*RegExp*，*Error*

187. *eval* 是做什么的?

参考答案：

此函数可以接受一个字符串 *str* 作为参数，并把此 *str* 当做一段 *javascript* 代码去执行，如果 *str* 执行结果是一个值则返回此值，否则返回 *undefined*。如果参数不是一个字符串，则直接返回该参数。

例如：

```
eval("var a=1");//声明一个变量a并赋值1。
eval("2+3");//5执行加运算，并返回运算值。
eval("mytest()");//执行mytest()函数。
eval("{b:2}");//声明一个对象。
```

189. *new* 操作符具体干了什么呢?

参考答案：

- 创建一个空对象 。
- 由 this 变量引用该对象 。
- 该对象继承该函数的原型(更改原型链的指向) 。
- 把属性和方法加入到 this 引用的对象中。
- 新创建的对象由 this 引用 ， 最后隐式地返回 this，过程如下：

```
var obj = {};
obj.__proto__ = Base.prototype;
Base.call(obj);
```

190. 去除字符串中的空格

参考答案：

方法一：*replace*正则匹配方法

代码示例：

- 去除字符串内所有的空格：`str = str.replace(/\s*/g,"");`
- 去除字符串内两头的空格：`str = str.replace(/^\s*|\s*$/g,"");`
- 去除字符串内左侧的空格：`str = str.replace(/^\s*/,"");`
- 去除字符串内右侧的空格：`str = str.replace(/(\s*$)/g,"");`

方法二：字符串原生 *trim* 方法

trim 方法能够去掉两侧空格返回新的字符串，不能去掉中间的空格

191. 常见的内存泄露，以及解决方案

参考答案：

内存泄露概念

内存泄漏指由于疏忽或错误造成程序未能释放已经不再使用的内存。内存泄漏并非指内存在物理上的消失，而是应用程序分配某段内存后，由于设计错误，导致在释放该段内存之前就失去了对该段内存的控制，从而造成了内存的浪费。

内存泄漏通常情况下只能由获得程序源代码和程序员才能分析出来。然而，有不少人习惯于把任何不需要的内存使用的增加描述为内存泄漏，即使严格意义上来说这是不准确的。

***JS* 垃圾收集机制**

JS 具有自动回收垃圾的机制，即执行环境会负责管理程序执行中使用的内存。在C和C++等其他语言中，开发者的需要手动跟踪管理内存的使用情况。在编写 *JS* 代码的时候，开发人员不用再关心内存使用的问题，所需内存的分配 以及无用的回收完全实现了自动管理。

Js中最常用的垃圾收集方式是标记清除(mark-and-sweep)。当变量进入环境（例如，在函数中声明一个变量）时，就将这个变量标记为“进入环境”。从逻辑上讲，永远不能释放进入环境的变量所占的内存，因为只要执行流进入相应的环境，就可能用到它们。而当变量离开环境时，这将其 标记为“离开环境”。

常见内存泄漏以及解决方案

1. 意外的全局变量

Js处理未定义变量的方式比较宽松：未定义的变量会在全局对象创建一个新变量。在浏览器中，全局对象是window。

```
function foo(arg) {
    bar = "this is a hidden global variable"; //等同于window.bar="this is a hidden global variable"
    this.bar2= "potential accidental global";//这里的this 指向了全局对象（window）,等同于window.bar2="potential accidental global"
}
```

解决方法：在 JavaScript 程序中添加，开启严格模式'use strict'，可以有效地避免上述问题。

注意：那些用来临时存储大量数据的全局变量，确保在处理完这些数据后将其设置为null或重新赋值。与全局变量相关的增加内存消耗的一个主因是缓存。缓存数据是为了重用，缓存必须有一个大小上限才有用。高内存消耗导致缓存突破上限，因为缓 存内容无法被回收。

1. 循环引用

在js的内存管理环境中，对象 A 如果有访问对象 B 的权限，叫做对象 A 引用对象 B。引用计数的策略是将“对象是否不再需要”简化成“对象有没有其他对象引用到它”，如果没有对象引用这个对象，那么这个对象将会被回收 。

```
let obj1 = { a: 1 }; // 一个对象（称之为 A）被创建，赋值给 obj1，A 的引用个数为 1
let obj2 = obj1; // A 的引用个数变为 2

obj1 = 0; // A 的引用个数变为 1
obj2 = 0; // A 的引用个数变为 0，此时对象 A 就可以被垃圾回收了
```

但是引用计数有个最大的问题： 循环引用。

```
function func() {
    let obj1 = {};
    let obj2 = {};

    obj1.a = obj2; // obj1 引用 obj2
    obj2.a = obj1; // obj2 引用 obj1

}
```

当函数 func 执行结束后，返回值为 undefined，所以整个函数以及内部的变量都应该被回收，但根据引用计数方法，obj1 和 obj2 的引用次数都不为 0，所以他们不会被回收。要解决循环引用的问题，最好是在不使用它们的时候手工将它们设为空。上面的例子可以这么做：

```
obj1 = null;
obj2 = null;
```

1. 被遗忘的计时器和回调函数

```
let someResource = getData();
setInterval(() => {
    const node = document.getElementById('Node');
    if(node) {
        node.innerhtml = JSON.stringify(someResource));
    }
}, 1000);
```

上面的例子中，我们每隔一秒就将得到的数据放入到文档节点中去。

但在 *setInterval* 没有结束前，回调函数里的变量以及回调函数本身都无法被回收。那什么才叫结束呢?

就是调用了 *clearInterval*。如果回调函数内没有做什么事情，并且也没有被 *clear* 掉的话，就会造成内存泄漏。

不仅如此，如果回调函数没有被回收，那么回调函数内依赖的变量也没法被回收。上面的例子中，*someResource* 就没法被回收。同样的，*setTiemout* 也会有同样的问题。所以，当不需要 *interval* 或者 *timeout* 时，最好调用 *clearInterval* 或者 *clearTimeout*。

1. *DOM* 泄漏

在 *JS* 中对*DOM*操作是非常耗时的。因为*JavaScript/ECMAScript*引擎独立于渲染引擎，而*DOM*是位于渲染引擎，相互访问需要消耗一定的资源。 而 *IE* 的 *DOM* 回收机制便是采用引用计数的，以下主要针对 *IE* 而言的。

a. 没有清理的 DOM 元素引用

```
var refA = document.getElementById('refA');
document.body.removeChild(refA);
// refA 不能回收，因为存在变量 refA 对它的引用。将其对 refA 引用释放，但还是无法回收 refA。
```

解决办法：*refA = null;*

b. 给 DOM 对象添加的属性是一个对象的引用

```
var MyObject = {};
document.getElementById('mydiv').myProp = MyObject;
```

解决方法：
在 *window.onunload* 事件中写上: *document.getElementById('mydiv').myProp = null;*

c. DOM 对象与 JS 对象相互引用

```
function Encapsulator(element) {
  this.elementReference = element;
  element.myProp = this;
}
new Encapsulator(document.getElementById('myDiv'));
```

解决方法： 在 onunload 事件中写上: document.getElementById('myDiv').myProp = null;

d. 给 DOM 对象用 attachEvent 绑定事件

```
function doClick() {}
element.attachEvent("onclick", doClick);
```

解决方法： 在onunload事件中写上: element.detachEvent('onclick', doClick);

e. 从外到内执行 appendChild。这时即使调用 removeChild 也无法释放

```
var parentDiv = document.createElement("div");
var childDiv = document.createElement("div");
document.body.appendChild(parentDiv);
parentDiv.appendChild(childDiv);
```

解决方法： 从内到外执行 appendChild:

```
var parentDiv = document.createElement("div");
var childDiv = document.createElement("div");
parentDiv.appendChild(childDiv);
document.body.appendChild(parentDiv);
```

1. *JS* 的闭包

> 闭包在 *IE6* 下会造成内存泄漏，但是现在已经无须考虑了。值得注意的是闭包本身不会造成内存泄漏，但闭包过多很容易导致内存泄漏。闭包会造成对象引用的生命周期脱离当前函数的上下文，如果闭包如果使用不当，可以导致环形引用（*circular reference*），类似于死锁，只能避免，无法发生之后解决，即使有垃圾回收也还是会内存泄露。
>
> 1. *console*
>
> 控制台日志记录对总体内存配置文件的影响可能是许多开发人员都未想到的极其重大的问题。记录错误的对象可以将大量数据保留在内存中。注意，这也适用于：
>
> (1) 在用户键入 JavaScript 时，在控制台中的一个交互式会话期间记录的对象。
> (2) 由 console.log 和 console.dir 方法记录的对象。

193. 设计一个方法(*isPalindrom*)以判断是否回文(颠倒后的字符串和原来的字符串一样为回文)

> 参考答案：
>
> 示例代码如下：

```
function isPalindrome(str) {
 if (typeof str !== 'string') {
     return false
 }
 return str.split('').reverse().join('') === str
}

// 测试
console.log(isPalindrome('HelleH')); // true
console.log(isPalindrome('Hello')); // false
```

194. 设计一个方法(*findMaxDuplicateChar*)以统计字符串中出现最多次数的字符

> 参考答案：
>
> 示例代码如下：

```
function findMaxDuplicateChar(str) {
 let cnt = {},  //用来记录所有的字符的出现频次
     c = '';    //用来记录最大频次的字符
 for (let i = 0; i < str.length; i++) {
     let ci = str[i];
     if (!cnt[ci]) {
         cnt[ci] = 1;
     } else {
         cnt[ci]++;
     }
     if (c == '' || cnt[ci] > cnt[c]) {
```

```
            c = ci;
        }
    }
    console.log(cnt); // { H: 1, e: 1, l: 3, o: 2, ' ': 1, W: 1, r: 1, d: 1 }
    return c;
}

// 测试
console.log(findMaxDuplicateChar('Hello World')); // l
```

195. 设计一段代码，使得通过点击按钮可以在 *span* 中显示文本框中输入的值

参考答案：

示例代码如下：

```
<body>
    <span id="showContent">在右侧输入框中输入内容</span>
    <input type="text" name="content" id="content">
    <button id="btn">更新内容</button>
    <script>
        btn.onclick = function(){
            var content = document.getElementById('content').value;
            if(content){
                document.getElementById('showContent').innerHTML = content;
            }
        }
    </script>
</body>
```

196. *map* 和 *forEach* 的区别?

参考答案：

两者区别

`forEach()`方法不会返回执行结果，而是`undefined`。

也就是说，`forEach()`会修改原来的数组。而`map()`方法会得到一个新的数组并返回。

适用场景

`forEach`适合于你并不打算改变数据的时候，而只是想用数据做一些事情 - 比如存入数据库或则打印出来。

`map()`适用于你要改变数据值的时候。不仅仅在于它更快，而且返回一个新的数组。这样的优点在于你可以使用复合(*composition*)(*map, filter, reduce* 等组合使用)来玩出更多的花样。

197. *Array* 的常用方法

> 参考答案：
>
> *Array* 的常用方法很多，挑选几个自己在实际开发中用的比较多的方法回答即可。

方法	描述
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
push()	向数组的末尾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。
shift()	删除并返回数组的第一个元素
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。
unshift()	向数组的开头添加一个或更多元素，并返回新的长度。
valueOf()	返回数组对象的原始值

@稀土掘金技术社区

> 更多 *Array* 相关用法可以参阅：www.w3school.com.cn/jsref/jsref...

199. 什么是预解析（预编译）

> 参考答案：
>
> 所谓的预解析（预编译）就是：在当前作用域中，*JavaScript* 代码执行之前，浏览器首先会默认的把所有带 *var* 和 *function* 声明的变量进行提前的声明或者定义。
>
> 另外，*var* 声明的变量和 *function* 声明的函数在预解析的时候有区别，*var* 声明的变量在预解析的时候只是提前的声明，*function* 声明的函数在预解析的时候会提前声明并且会同时定义。也就是说 *var* 声明的变量和 *function* 声明的函数的区别是在声明的同时有没有同时进行定义。

注： 由于字数限制，剩余内容在下篇进行总结哦。🤞，大家认真看哦，奥利给！💪