JavaScript 面试题汇总（下篇）

201. 冒泡排序的思路，不用 *sort*

> 参考答案：
>
> 示例代码如下：

```
var examplearr = [8, 94, 15, 88, 55, 76, 21, 39];
function sortarr(arr) {
 for (i = 0; i < arr.length - 1; i++) {
     for (j = 0; j < arr.length - 1 - i; j++) {
         if (arr[j] > arr[j + 1]) {
             var temp = arr[j];
             arr[j] = arr[j + 1];
             arr[j + 1] = temp;
         }
     }
 }
 return arr;
}
sortarr(examplearr);
console.log(examplearr); // [8, 15, 21, 39, 55, 76, 88, 94]
```

202. *symbol* 用途

> 参考答案：
>
> 可以用来表示一个独一无二的变量防止命名冲突。但是面试官问还有吗?我没想出其他的用处就直接答我不知道了，还可以利用 symbol 不会被常规的方法(除了 Object.getOwnPropertySymbols 外)遍历到，所以可以用来模拟私有变量。
>
> 主要用来提供遍历接口，布置了 symbol.iterator 的对象才可以使用 for…of 循环，可以统一处理数据结构。调用之后回返回一个遍历器对象，包含有一个 next 方法，使用 next 方法后有两个返回值 value 和 done 分别表示函数当前执行位置的值和是否遍历完毕。
>
> Symbol.for() 可以在全局访问 symbol

203. 什么是函数式编程，应用场景是什么

> 参考答案：
>
> 函数式编程和面向对象编程一样，是一种编程范式。强调执行的过程而非结果，通过一系列的嵌套的函数调用，完成一个运算过程。 它主要有以下几个特点：
>
> 1. 函数是"一等公民"：函数优先，和其他数据类型一样。
> 2. 只用"表达式"，不用"语句"：通过表达式（*expression*）计算过程得到一个返回值，而不是通过一个语句（*statement*）修改某一个状态。
> 3. 无副作用：不污染变量，同一个输入永远得到同一个数据。

4. 不可变性：前面一提到，不修改变量，返回一个新的值。

函数式编程的概念其实出来也已经好几十年了，我们能在很多编程语言身上看到它的身影。比如比较纯粹的 *Haskell*，以及一些语言开始逐渐成为多范式编程语言，比如 *Swift*，还有 *Kotlin*，*Java*，*Js* 等都开始具备函数式编程的特性。

函数式编程在前端的应用场景

- *Stateless components*：*React* 在 *0.14* 之后推出的无状态组件
- *Redux*

函数式编程在后端的应用场景

- *Lambda* 架构

204. 事件以及事件相关的兼容性问题

参考答案：

事件最早是在 *IE3* 和 *Navigator2* 中出现的，当时是作为分担服务器运算负担的一种手段。要实现和网页的互动，就需要通过 *JavaScript* 里面的事件来实现。

每次用户与一个网页进行交互，例如点击链接，按下一个按键或者移动鼠标时，就会触发一个事件。我们的程序可以检测这些事件，然后对此作出响应。从而形成一种交互。

当我们绑定事件时，需要遵循事件三要素

- 事件源：是指那个元素引发的事件。比如当你点击图标的时候，会跳转到百度首页。那么这个图标就是事件源。
- 事件：事件是指执行的动作。例如，点击，鼠标划过，按下键盘，获得焦点。
- 事件驱动程序：事件驱动程序即执行的结果。例如，当你点击图标的时候，会跳转到百度首页。那么跳转到百度首页就是事件的处理结果。

```
事件源.事件 = function() {
    事件处理函数
}
```

205. *JS* 小数不精准，如何计算

参考答案：

方法一：指定要保留的小数位数(0.1+0.2).toFixed(1) = 0.3;这个方法toFixed是进行四舍五入的也不是很精准，对于计算金额这种严谨的问题，不推荐使用，而且不同浏览器对toFixed的计算结果也存在差异。

方法二：把需要计算的数字升级（乘以10的n次幂）成计算机能够精确识别的整数，等计算完毕再降级（除以10的n次幂），这是大部分编程语言处理精度差异的通用方法。

206. 写一个 *mySetInterVal(fn, a, b)*，每次间隔 *a,a+b,a+2b* 的时间，然后写一个 *myClear*，停止上面的 *mySetInterVal*

> 参考答案：
>
> 该题的思路就是每一次在定时器中重启定时器并且在时间每一次都加 *b*，并且要把定时器返回回来，可以作为*myClear*的参数。
>
> 代码如下：

```
var mySetInterVal = function (fn, a, b) {
var timer = null;
var settimer = function (fn, a, b) {
 timer = setTimeout(() => {
   fn();
   settimer(fn, a + b, b);
 }, a);
}
settimer(fn, a, b);
return timer;
}

var timer = mySetInterVal(() => { console.log('timer') }, 1000, 1000);
var myClear = function (timer) {
timer && clearTimeout(timer);
}
```

207. 合并二维有序数组成一维有序数组，归并排序的思路

> 参考答案：
>
> 示例代码如下：

```
function merge(left, right) {
 let result = []
 while (left.length > 0 && right.length > 0) {
     if (left[0] < right[0]) {
         result.push(left.shift())
     } else {
         result.push(right.shift())
     }
 }
 return result.concat(left).concat(right)
}
function mergeSort(arr) {
 if (arr.length === 1) {
     return arr
 }
 while (arr.length > 1) {
```

```
        let arrayItem1 = arr.shift();
        let arrayItem2 = arr.shift();
        let mergeArr = merge(arrayItem1, arrayItem2);
        arr.push(mergeArr);
    }
    return arr[0]
}

let arr1 = [[1, 2, 3], [4, 5, 6], [7, 8, 9], [1, 2, 3], [4, 5, 6]];
let arr2 = [[1, 4, 6], [7, 8, 10], [2, 6, 9], [3, 7, 13], [1, 5, 12]];
console.log(mergeSort(arr1))
console.log(mergeSort(arr2))
```

208. 给定一个字符串，请你找出其中不含有重复字符的最长子串的长度。

参考答案：

首先，我们肯定需要封装一个函数，而这个函数接收一个字符串作为参数，返回不含有重复字符的子串长度。来看下面的示例：

示例 1:

输入: “abcabcbb” 输出: 3 解释: 因为无重复字符的最长子串是 “abc”，所以其长度为 3。

示例 2:

输入: “bbbbb” 输出: 1 解释: 因为无重复字符的最长子串是 “b”，所以其长度为 1。

示例 3:

输入: “pwwkew” 输出: 3 解释: 因为无重复字符的最长子串是 “wke”，所以其长度为 3。 请注意，你的答案必须是 子串 的长度，“pwke” 是一个子序列，不是子串。

示例代码：

```
var lengthOfLongestSubstring = function (s) {
    var y = [];
    var temp = [];
    var maxs = 0;
    if (s == "") {
        return 0;
    }
    if (s.length == 1) {
        return 1;
    }
    for (var i = 0; i < s.length; i++) {
        if (temp.includes(s[i])) {

            y.push(temp.length);
            temp.shift();
            continue;
        } else {
```

```
            temp.push(s[i])
            y.push(temp.length);
        }

    }
    for (var j = 0; j < y.length; j++) {
        if (maxs <= y[j]) {
            maxs = y[j]
        }
    }
    return maxs;
};
// 测试
console.log(lengthOfLongestSubstring('abcabcbb')); // 3
console.log(lengthOfLongestSubstring('bbbbb')); // 1
console.log(lengthOfLongestSubstring('pwwkew')); // 3
```

209. 有一堆整数，请把他们分成三份，确保每一份和尽量相等（11，42，23，4，5，6 4 5 6 11 23 42 56 78 90）(滴滴 *2020*)

参考答案：

本道题目是一道考察算法的题目，主要是考察编程基本功和一定的想像力。

具体的实现如下：

```
function fun(total, n) {
    //先对整个数组进行排序
    total.sort((a, b) => a - b);

    //求和
    var sum = 0;
    for (var i = 0; i < total.length; i++) {
        sum += total[i];
    }

    var avg = Math.ceil(sum / n);

    //结果数组
    var result = []; //长度为n

    for (var i = 0; i < n; i++) {
        result[i] = [total.pop()];
        result[i].sum = result[i][0];

        //组成一个分数组
        while (result[i].sum < avg && total.length > 0) {
            for (var j = 0; j < total.length; j++) {
                if (result[i].sum + total[j] >= avg) {
```

```
                result[i].push(total[j]);
                result[i].sum += total[j];
                break;
            }
        }

        if (j == total.length) {
            result[i].push(total.pop());
            result[i].sum += result[i][result[i].length - 1];
        } else {
            //从数组中移除此元素
            total.splice(j, 1);
        }
    }

    sum -= result[i].sum;
    avg = Math.ceil(sum / (n - 1 - i));

 }
 return result;
}

// 测试
var arr = [11, 42, 23, 4, 5, 6, 4, 5, 6, 11, 23, 42, 56, 78, 90];
console.log(fun(arr, 3));
// [
//     [ 90, 56, sum: 146 ],
//     [ 78, 42, 11, sum: 131 ],
//     [ 42, 23, 23, 11, 6, 6, 5, 5, 4, 4, sum: 129 ]
// ]
```

210. 手写发布订阅（头条2020）

参考答案：

示例代码如下：

```
<body>
 <div id="app">
     <p>this is a test</p>
     {{msg}}<input type="text" v-model="msg">{{msg}}
 </div>
 <script src="./index.js"></script>
 <script>
     const vm = new Vue({
         el : '#app',
         data : {
             msg : ''
         }
```

```
    });
 </script>
</body>
/*
    1. 创建 Vue 构造函数
        在 Vue 构造函数中，调用了 observer 函数，该函数的作用就是对数据进行劫持
        劫持具体要做的事儿：复制一份数据，但是不是单纯的复制，而是增加了 getter、setter
    2. 书写 compile 函数。该函数主要作用于模板，从模板里面要提取信息
        提取的东西主要有两个：{{}}  和 v-model
    3. 创建发布者 Dep 的构造函数，如果数据发生变化，发布者就会遍历内部的数组（花名册），通知订阅者修改数据
    4. 创建订阅者 Watcher 的构造函数，如果有数据的变化，发布者就会通知订阅者，订阅者上面存在 update 方法，会进行修改
 */

function Vue(options){
    // this 代表 Vue 的实例对象，本例中就是 vm
    // options.data 这就是实际的数据 {msg : 'xiejie'}
    observer(this,options.data);
    this.$el = options.el;
    compile(this);
}

// 用于对模板进行信息提取，主要提取 {{}}  和 v-model，然后进行一些操作
// {{ }} 会成为观察者，v-model 所对应的控件来绑定事件
function compile(vm){
    var el = document.querySelector(vm.$el); // el 所对应的值为 <div id="app">...</div>
    var documentFragment = document.createDocumentFragment(); // 创建了一个空的文档碎片
    var reg = /\{\{(.*)\}\}/; // 创建正则表达式 匹配 {{ }}
    while(el.childNodes[0]){
        var child = el.childNodes[0]; // 将第一个子节点存储到 child
        if(child.nodeType == 1){
            // 如果能够进入此 if，说明该节点是一个元素节点
            for(var key in child.attributes){
                // 遍历该元素节点的每一个属性，拿到的就是 type="text" v-model="msg"
                var attrName = child.attributes[key].nodeName; // 获取属性名  type、v-model
                if(attrName === 'v-model'){
                    var vmKey = child.attributes[key].nodeValue; // 先获取属性值，也就是 msg
                    // 为该节点，也就是 <input type="text" v-model="msg"> 绑定一个 input 事件
                    child.addEventListener('input', function (event) {
                        vm[vmKey] = event.target.value; // 获取用户输入的值，然后改变 vm 里面的 msg 属性对应的值，注意这里会触发 setter
                    })
                }
```

```
            }
        }
        if(child.nodeType == 3){
            // 如果能进入此 if，说明该节点是一个文本节点
            if(reg.test(child.nodeValue)){
                // 如果能够进入到此 if，说明是 {{ }}，然后我们要让其成为订阅者
                var vmKey = RegExp.$1; // 获取正则里面的捕获值，也就是 msg
                // 实例化一个 Watcher（订阅者），接收 3 个参数：Vue 实例，该文本节点，捕获值
msg
                new Watcher(vm, child, vmKey);
            }
        }
        documentFragment.appendChild(el.childNodes[0]); // 将第一个子节点添加到文档碎片
里面
    }
    // 将文档碎片中节点重新添加到 el，也就是 <div id="app"></div> 下面
    el.appendChild(documentFragment);
}

// 新建发布者构造函数
function Dep() {
    // 将观察者添加到发布者内部的数组里面
    // 这样以便于通知所有的观察者去更新数据
    this.subs = [];
}

Dep.prototype = {
    // 将 watcher 添加到发布者内置的数组里面
    addSub: function (sub) {
        this.subs.push(sub);

    },
    // 遍历数组里面所有的 watcher，通知它们去更新数据
    notify: function () {
        this.subs.forEach(function (sub) {
            sub.update();
        })
    }
}

// 新建观察者 Watcher 构造函数
// 接收 3 个参数：Vue 实例，文本节点 {{ msg }} 以及捕获内容 msg
function Watcher(vm, child, vmKey) {
    this.vm = vm; // vm
    this.child = child; // {{ msg }}
    this.vmKey = vmKey; // msg
    Dep.target = this; // 将该观察者实例对象添加给 Dep.target
    this.update(); // 执行节点更新方法
    Dep.target = null; // 最后清空 Dep.target
```

```
}
Watcher.prototype = {
    // 节点更新方法
    update: function () {
        // 相当于：{{ msg }}.nodeValue = this.vm['msg']
        // 这样就更新了文本节点的值，由于这里在获取 vm.msg，所以会触发 getter
        this.child.nodeValue = this.vm[this.vmKey];
    }
}

// 该函数的作用是用于数据侦听
function observer(vm,obj){
    var dep = new Dep(); // 新增一个发布者:发布者的作用是告诉订阅者数据已经更改
    // 遍历数据
    for(var key in obj){
        // 将数据的每一项添加到 vm 里面，至此，vm 也有了每一项数据
        // 但是不是单纯的添加，而是设置了 getter 和 setter
        // 在获取数据时触发 getter，在设置数据时触发 setter
        Object.defineProperty(vm, key, {
            get() {
                console.log("触发get了");
                // 触发 getter 时，将该 watcher 添加到发布者维护的数组里面
                if (Dep.target) {
                    dep.addSub(Dep.target); // 往发布者的数组里面添加订阅者
                }
                console.log(dep.subs);
                return obj[key];
            },
            set(newVal) {
                console.log("触发set了");
                obj[key] = newVal;
                dep.notify(); // 发布者发出消息，通知订阅者修改数据
            }
        });
    }
}
```

211. 手写用 *ES6proxy* 如何实现 *arr[-1]* 的访问（滴滴2020）

参考答案：

示例代码如下：

```
const proxyArray = arr => {
 const length = arr.length;
 return new Proxy(arr, {
     get(target, key) {
         key = +key;
         while (key < 0) {
```

```
                key += length;
            }
            return target[key];
        }
    })
};
var a = proxyArray([1, 2, 3, 4, 5, 6, 7, 8, 9]);
console.log(a[1]);  // 2
console.log(a[-10]);  // 9
console.log(a[-20]);  // 8
```

212. 下列代码执行结果

```
console.log(1);
setTimeout(() => {
    console.log(2);
    process.nextTick(() => {
        console.log(3);
    });
    new Promise((resolve) => {
        console.log(4);
        resolve();
    }).then(() => {
        console.log(5);
    });
});
new Promise((resolve) => {
    console.log(7);
    resolve();
}).then(() => {
    console.log(8);
});
process.nextTick(() => {
    console.log(6);
});
setTimeout(() => {
    console.log(9);
    process.nextTick(() => {
        console.log(10);
    });
    new Promise((resolve) => {
        console.log(11);
        resolve();
    }).then(() => {
        console.log(12);
    });
});
```

> 参考答案：
>
> 1 7 6 8 2 4 3 5 9 11 10 12

213. Number() 的存储空间是多大？如果后台发送了一个超过最大自己的数字怎么办

> 参考答案：
>
> Math.pow(2, 53) ，53 为有效数字，会发生截断，等于 JS 能支持的最大数字。

214. 事件是如何实现的？(字节2020)

> 参考答案：
>
> 基于发布订阅模式，就是在浏览器加载的时候会读取事件相关的代码，但是只有实际等到具体的事件触发的时候才会执行。
>
> 比如点击按钮，这是个事件(Event)，而负责处理事件的代码段通常被称为事件处理程序(Event Handler)，也就是「启动对话框的显示」这个动作。
>
> 在 Web 端，我们常见的就是 DOM 事件：
>
> - DOM0 级事件，直接在 html 元素上绑定 on-event，比如 onclick，取消的话，dom.onclick = null，同一个事件只能有一个处理程序，后面的会覆盖前面的。
> - DOM2 级事件，通过 addEventListener 注册事件，通过 removeEventListener 来删除事件，一个事件可以有多个事件处理程序，按顺序执行，捕获事件和冒泡事件
> - DOM3级事件，增加了事件类型，比如 UI 事件，焦点事件，鼠标事件

215. 下列代码执行结果

```
Promise.resolve().then(() => {
    console.log(0);
    return Promise.resolve(4);
}).then((res) => {
    console.log(res)
})

Promise.resolve().then(() => {
    console.log(1);
}).then(() => {
    console.log(2);
}).then(() => {
    console.log(3);
}).then(() => {
    console.log(5);
}).then(() =>{
    console.log(6);
})
```

参考答案：

0 1 2 3 4 5 6

解析：

照着代码，我们先来看初始任务。

- （初始任务）第一部分 Promise.resolve() 返回 「Promise { undefined }」。
- （同任务，下同）继续调用 then，then 发现「Promise { undefined }」已解决，直接 enqueue 包含 console.log(0);return Promise.resolve(4) 的任务，之后返回新的「Promise { }」（设为 promise0）。被 enqueue 的任务之后会引发 promise0 的 resolve/reject，详见 追加任务一 的 2. 3.。
- 继续调用 promise0 上的 then，第二个 then 发现 promise0 还在 pending，因此不能直接 enqueue 新任务，而是将包含 console.log(res) 回调追加到 promise0 的 PromiseFulfillReactions 列表尾部，并返回新的「Promise { }」（设为 promiseRes）（该返回值在代码中被丢弃，但不影响整个过程）。
- 第二部分 Promise.resolve().then... 同理，只有包含 console.log(1) 的任务被 enqueue。中间结果分别设为 promise1（=Promise.resolve().then(() => {console.log(1);})）, promise2, promise3, promise5, promise6。当前任务执行完毕。

此时，任务列队上有两个新任务，分别包含有 console.log(0);return Promise.resolve(4) 和 console.log(1)。我们用 「Job { ??? }」来指代。

接下来，「Job { console.log(0);return Promise.resolve(4) }」先被 enqueue，所以先运行「Job { console.log(0);return Promise.resolve(4) }」。

- （追加任务一）此时「0」被 console.log(0) 输出。Promise.resolve(4) 返回已解决的「Promise { 4 }」，然后 return Promise.resolve(4) 将这个「Promise { 4 }」作为最开始的 Promise.resolve().then（对应 promise0）的 onfulfill 处理程序（即 then(onfulfill, onreject) 的参数 onfulfill）的返回值返回。
- （同任务，下同）onfulfill 处理程序返回，触发了 promise0 的 Promise Resolve Function（以下简称某 promise（实例）的 resolve）。所谓触发，其实是和别的东西一起打包到「Job { console.log(0);return Promise.resolve(4) }」当中，按流程执行，onfulfill 返回后自然就到它了。（onfulfill 抛异常的话会被捕获并触发 reject，正常返回就是 resolve。）
- promise0 的 resolve 检查 onfulfill 的返回值，发现该值包含**可调用的**「then」属性。这是当然的，因为是「Promise { 4 }」。无论该 Promise 实例是否解决，都将 enqueue 一个新任务包含调用该返回值的 then 的任务（即规范中的 NewPromiseResolveThenableJob(promiseToResolve, thenable, then)）。而这个任务才会触发后续操作，在本例中，最终会将 promise0 的 PromiseFulfillReactions（其中有包含 console.log(res) 回调）再打包成任务 enqueue 到任务列队上。当前任务执行完毕。

此时，任务列队上还是有两个任务（一进一出），「Job { console.log(1) }」和「NewPromiseResolveThenableJob(promise0, 「Promise { 4 }」, 「Promise { 4 }」.then)」。接下来执行「Job { console.log(1) }」。

- （追加任务二）「1」被输出。
- （同任务，下同）onfulfill 处理程序返回 undefined。（JavaScript 的函数默认就是返回 undefined。）
- promise1 的 resolve 发现 undefined 连 Object 都不是，自然不会有 then，所以将 undefined 作为 promise1 的解决结果。即 promise1 从「Promise { }」变为 「Promise { undefined }」（fulfill）。

- resolve 继续查看 promise1 的 PromiseFulfillReactions。（reject 则会查看 PromiseRejectReactions。）有一个之前被 promise1.then 调用追加上的包含 console.log(2) 的回调。打包成任务入列。（如有多个则依次序分别打包入列。）当前任务执行完毕。

此时，任务列队上仍然有两个任务（一进一出）。「NewPromiseResolveThenableJob(...)」和 「Job { console.log(2) }」。执行「NewPromiseResolveThenableJob(...)」。

- （追加任务三）调用 「Promise { 4 }」的 then。这个调用的参数（处理程序 onfulfill 和 onreject） 用的正是 promise0 的 resolve 和 reject。
- 由于「Promise { 4 }」的 then 是标准的，行为和其他的 then 一致。（可参见初始任务的步骤 2. 。）它发现「Promise { 4 }」已解决，结果是 4。于是直接 enqueue 包含 promise0 的 resolve 的任务，参数是 4。理论上同样返回一个「Promise { }」，由于是在内部，不被外部观察，也不产生别的影响。）当前任务执行完毕。

此时，任务列队上依旧有两个任务（一进一出）。「Job { console.log(2) }」和 「Job { promise0 的 resolve }」。执行「Job { console.log(2) }」。

- （追加任务四）过程类似「Job { console.log(1) }」的执行。「2」被输出。「Job { console.log(3) }」入列。其余不再赘述。当前任务执行完毕。

此时，任务列队上依然有两个任务（一进一出）。「Job { promise0 的 resolve }」和「Job { console.log(3) }」。执行「Job { promise0 的 resolve }」。

- （追加任务五）promise0 的 resolve 查看 PromiseFulfillReactions 发现有被 promise0.then 追加的回调。打包成任务入列。该任务包含 console.log(res)，其中传递 promise0 解决结果 4 给参数 res。当前任务执行完毕。

此时，任务列队上还是两个任务（一进一出）。「Job { console.log(3) }」和「Job { console.log(res) }」。

- （追加任务六）输出「3」。「Job { console.log(5) }」入列。

此时，任务列队上还是两个任务（一进一出）。「Job { console.log(res) }」和「Job { console.log(5) }」。

- （追加任务七）输出「4」。由于 promiseRes 没有被 then 追加回调。就此打住。

此时，任务列队上终于不再是两个任务了。下剩「Job { console.log(5) }」。

- （追加任务八）输出「5」。「Job { console.log(6) }」入列。

最后一个任务（追加任务九）输出「6」。任务列队清空。

因此，输出的顺序是「0 1 2 3 4 5 6」。

总结一下，除去初始任务，总共 enqueue 了 9 个任务。其中，第一串 Promise + then... enqueue 了 4 个。第二串 Promise + then... enqueue 了 5 个。分析可知，每增加一个 then 就会增加一个任务入列。

而且，第一串的 return Promise.resolve(4) 的写法额外 enqueue 了 2 个任务，分别在 promise0 的 resolve 时（追加任务一 3.）和调用「Promise { 4 }」的 then 本身时（追加任务三 2.）。

根据规范，它就该这样。说不上什么巧合，可以算是有意为之。处理程序里返回 thenable 对象就会导致增加两个任务入列。

216. 判断数组的方法，请分别介绍它们之间的区别和优劣

参考答案：

方法一：**instanceof 操作符判断**

用法：arr instanceof Array

instanceof 主要是用来判断某个实例是否属于某个对象

```
let arr = [];
console.log(arr instanceof Array); // true
```

缺点：instanceof是判断类型的prototype是否出现在对象的原型链中，但是对象的原型可以随意修改，所以这种判断并不准确。并且也不能判断对象和数组的区别

方法二：**对象构造函数的 constructor判断**

用法：arr.constructor === Array

Object的每个实例都有构造函数 constructor，用于保存着用于创建当前对象的函数

```
let arr = [];
console.log(arr.constructor === Array); // true
```

方法三：**Array 原型链上的 isPrototypeOf**

用法：Array.prototype.isPrototypeOf(arr)

Array.prototype 属性表示 Array 构造函数的原型

其中有一个方法是 isPrototypeOf() 用于测试一个对象是否存在于另一个对象的原型链上。

```
let arr = [];
console.log(Array.prototype.isPrototypeOf(arr)); // true
```

方法四：**Object.getPrototypeOf**

用法：Object.getPrototypeOf(arr) === Array.prototype

Object.getPrototypeOf() 方法返回指定对象的原型

所以只要跟Array的原型比较即可

```
let arr = [];
console.log(Object.getPrototypeOf(arr) === Array.prototype); // true
```

方法五：**Object.prototype.toString**

用法：Object.prototype.toString.call(arr) === '[object Array]'

虽然Array也继承自Object，但js在Array.prototype上重写了toString，而我们通过toString.call(arr)实际上是通过原型链调用了。

```
let arr = [];
console.log(Object.prototype.toString.call(arr) === '[object Array]'); // true
```

缺点：不能精准判断自定义对象，对于自定义对象只会返回[object Object]

方法六：**Array.isArray**

用法：Array.isArray(arr)

ES5中新增了Array.isArray方法,IE8及以下不支持

Array.isArray (arg) isArray 函数需要一个参数 arg，如果参数是个对象并且 class 内部属性是 "Array", 返回布尔值 true；否则它返回 false。

```
let arr = [];
console.log(Array.isArray(arr)); // true
```

缺点：Array.isArray是ES 5.1推出的，**不支持IE6~8**，所以在使用的时候需要注意兼容性问题。

217. JavaScript 中的数组和函数在内存中是如何存储的?

参考答案：

在 *JavaScript* 中，数组不是以一段连续的区域存储在内存中，而是一种哈希映射的形式存储在堆内容里面。它可以通过多种数据结构实现，其中一种是链表。如下图所示：

JavaScript 中的函数是存储在堆内存中的，具体的步骤如下：

1. 开辟堆内存（*16* 进制得到内存地址）
2. 声明当前函数的作用域（函数创建的上下文才是他的作用域，和在那执行的无关）
3. 把函数的代码以字符串的形式存储在堆内存中（函数再不执行的情况下，只是存储在堆内存中的字符串）
4. 将函数堆的地址，放在栈中供变量调用（函数名）

218. *JavaScript* 是如何运行的？解释型语言和编译型语言的差异是什么?

参考答案：

关于第一个问题，这不是三言两语或者几行文字就能够讲清楚的，这里放上一篇博文地址：

segmentfault.com/a/119000001...

第二个问题：解释型语言和编译型语言的差异是什么?

电脑能认得的是二进制数，不能够识别高级语言。所有高级语言在电脑上执行都需要先转变为机器语言。但是高级语言有两种类型：编译型语言和解释型语言。常见的编译型语言语言有C/C++、Pascal/Object 等等。常见的解释性语言有python、JavaScript等等。

> 编译型语言先要进行编译，然后转为特定的可执行文件，这个可执行文件是针对平台的（CPU类型），可以这么理解你在PC上编译一个C源文件，需要经过预处理，编译，汇编等等过程生成一个可执行的二进制文件。当你需要再次运行改代码时，不需要重新编译代码，只需要运行该可执行的二进制文件。优点，编译一次，永久执行。还有一个优点是，你不需要提供你的源代码，你只需要发布你的可执行文件就可以为客户提供服务，从而保证了你的源代码的安全性。但是，如果你的代码需要迁移到linux、ARM下时，这时你的可执行文件就不起作用了，需要根据新的平台编译出一个可执行的文件。这也就是多个平台需要软件的多个版本。缺点是，跨平台能力差。
>
> 解释型语言需要一个解释器，在源代码执行的时候被解释器翻译为一个与平台无关的中间代码，解释器会把这些代码翻译为及其语言。打个比方，编译型中的编译相当于一个翻译官，它只能翻译英语，而且中文文章翻译一次就不需要重新对文章进行二次翻译了，但是如果需要叫这个翻译官翻译德语就不行了。而解释型语言中的解释器相当于一个会各种语言的机器人，而且这个机器人回一句一句的翻译你的语句。对于不同的国家，翻译成不同的语言，所以，你只需要带着这个机器人就可以。解释型语言的有点是，跨平台，缺点是运行时需要源代码，知识产权保护性差，运行效率低。

219. 列举你所了解的编程范式?

> 参考答案：
>
> 编程范式 *Programming paradigm* 是指计算机中编程的典范模式或方法。
>
> 常见的编程范式有：函数式编程、程序编程、面向对象编程、指令式编程等。
>
> 不同的编程语言也会提倡不同的“编程范型”。一些语言是专门为某个特定的范型设计的，如 *Smalltalk* 和 *Java* 支持面向对象编程。而 *Haskell* 和 *Scheme* 则支持函数式编程。现代编程语言的发展趋势是支持多种范型，例如 *ES* 支持函数式编程的同时也支持面向对象编程。

220. 什么是面向切面（AOP）的编程?

> 参考答案：
>
> **什么是AOP?**
>
> AOP(面向切面编程)的主要作用是把一些跟核心业务逻辑模块无关的功能抽离出来，这些跟业务逻辑无关的功能通常包括日志统计、安全控制、异常处理等。把这些功能抽离出来之后， 再通过“动态织入”的方式掺入业务逻辑模块中。
>
> **AOP能给我们带来什么好处?**
>
> AOP的好处首先是可以保持业务逻辑模块的纯净和高内聚性，其次是可以很方便地复用日志统计等功能模块。
>
> **JavaScript实现AOP的思路?**
>
> 通常，在 JavaScript 中实现 AOP，都是指把一个函数“动态织入”到另外一个函数之中，具体的实现技术有很多，下面我用扩展 Function.prototype 来做到这一点。
>
> 主要就是两个函数，在Function的原型上加上before与after，作用就是字面的意思，在函数的前面或后面执行，相当于**无侵入**把一个函数插入到另一个函数的前面或后面，应用得当可以很好的实现代码的解耦，js中的代码实现如下：

```
//Aop构造器
function Aop(options){
```

```
 this.options = options
}
//业务方法执行前钩子
Aop.prototype.before = function(cb){
 cb.apply(this)
}
//业务方法执行后钩子
Aop.prototype.after = function(cb){
 cb.apply(this)
}
//业务方法执行器
Aop.prototype.execute = function(beforeCb,runner,afterCb){
 this.before(beforeCb)
 runner.apply(this)
 this.after(afterCb)
}

var aop = new Aop({
 afterInfo:'执行后',
 runnerInfo:'执行中',
 beforeInfo:'执行前'
})

var beforeCb = function(){
 console.log(this.options.beforeInfo)
}
var afterCb = function(){
 console.log(this.options.afterInfo)
}
var runnerCb = function(){
 console.log(this.options.runnerInfo)
}

aop.execute(beforeCb,runnerCb,afterCb)
```

应用的一些例子：

1. 为 *window.onload* 添加方法，防止 *window.onload* 被二次覆盖
2. 无侵入统计某个函数的执行时间
3. 表单校验
4. 统计埋点
5. 防止 *csrf* 攻击

221. *JavaScript* 中的 *const* 数组可以进行 *push* 操作吗？为什么？

> 参考答案：
>
> 可以进行 *push* 操作。虽然 *const* 表示常量，但是当我们把一个数组赋值给 *const* 声明的变量时，实际上是把这个数组的地址赋值给该变量。而 *push* 操作是在数组地址所指向的堆区添加元素，地址本身并没有发生改变。
>
> 示例代码：

```
const arr = [1];
arr.push(2);
console.log(arr); // [1, 2]
```

222. JavaScript 中对象的属性描述符有哪些？分别有什么作用？

> 参考答案：
>
> 从*ES5*开始，添加了对对象**属性描述符**的支持。现在*JavaScript*中支持 *4* 种属性描述符:
>
> - **configurable:** 当且仅当该属性的*configurable*键值为*true*时，该属性的描述符才能够被改变，同时该属性也能从对应的对象上被删除。
> - **enumerable:** 当且仅当该属性的*enumerable*键值为*true*时，该属性才会出现在对象的枚举属性中。
> - **value:** 该属性对应的值。可以是任何有效的 *JavaScript* 值（数值，对象，函数等）。
> - **writable:** 当且仅当该属性的*writable*键值为*true*时，属性的值，也就是上面的value，才能被赋值运算符改变。

223. *JavaScript* 中 *console* 有哪些 *api*？

> 参考答案：
>
> **console.assert(expression, object[, object...])**
>
> 接收至少两个参数，第一个参数的值或返回值为`false`的时候，将会在控制台上输出后续参数的值。
>
> **console.count([label])**
>
> 输出执行到该行的次数，可选参数 label 可以输出在次数之前。
>
> **console.dir(object)**
>
> 将传入对象的属性，包括子对象的属性以列表形式输出。
>
> **console.error(object[, object...])**
>
> 用于输出错误信息，用法和常见的`console.log`一样，不同点在于输出内容会标记为错误的样式，便于分辨。
>
> **console.group**
>
> 这是个有趣的方法，它能够让控制台输出的语句产生不同的层级嵌套关系，每一个`console.group()`会增加一层嵌套，相反要减少一层嵌套可以使用`console.groupEnd()`方法。

console.info(object[, object...])

此方法与之前说到的`console.error`一样，用于输出信息，没有什么特别之处。

console.table()

可将传入的对象，或数组以表格形式输出，相比传统树形输出，这种输出方案更适合内部元素排列整齐的对象或数组，不然可能会出现很多的 undefined。

console.log(object[, object...])

输入一段 *log* 信息。

console.profile([profileLabel])

这是个挺高大上的东西，可用于性能分析。在 JS 开发中，我们常常要评估段代码或是某个函数的性能。在函数中手动打印时间固然可以，但显得不够灵活而且有误差。借助控制台以及`console.profile()`方法我们可以很方便地监控运行性能。

console.time(name) 计时器，可以将成对的`console.time()`和`console.timeEnd()`之间代码的运行时间输出到控制台上，`name`参数可作为标签名。

console.trace()

`console.trace()`用来追踪函数的调用过程。在大型项目尤其是框架开发中，函数的调用轨迹可以十分复杂，`console.trace()`方法可以将函数的被调用过程清楚地输出到控制台上。

console.warn(object[, object...])

输出参数的内容，作为警告提示。

225. *Object.defineProperty* 有哪几个参数？各自都有什么作用

参考答案：

在 *JavaScript* 中，通过 *Object.defineProperty* 方法可以设置对象属性的特性，选项如下：

- *get*：一旦目标属性被访问时，就会调用相应的方法
- *set*：一旦目标属性被设置时，就会调用相应的方法
- *value*：这是属性的值，默认是 *undefined*
- *writable*：这是一个布尔值，表示一个属性是否可以被修改，默认是 *true*
- *enumerable*：这是一个布尔值，表示在用 *for-in* 循环遍历对象的属性时，该属性是否可以显示出来，默认值为 *true*
- *configurable*：这是一个布尔值，表示我们是否能够删除一个属性或者修改属性的特性，默认值为 *true*

226. *Object.defineProperty* 和 *ES6* 的 *Proxy* 有什么区别?

参考答案：

1、*Object.defineproperty*

可以用于监听对象的数据变化

语法： ***Object.defineproperty(obj, key, descriptor)***

```
let obj = {
 age: 11
}
let value = 'xiaoxiao';
//defineproperty 有 gettter 和 setter
Object.defineproperty(obj, 'name', {
 get() {
     return value
 },
 set(newValue) {
     value = newValue
 }
})
obj.name = 'pengpeng';
```

此外 还有以下配置项：

- *configurable*
- *enumerable*
- *value*

缺点：

1. 无法监听数组变化
2. 只能劫持对象的属性，属性值也是对象那么需要深度遍历

2、*proxy*： 可以理解为在被劫持的对象之前 加了一层拦截

```
let proxy = new Proxy({}, {
    get(obj, prop) {
        return obj[prop]
    },
    set(obj, prop, val) {
        obj[prop] = val
    }
})
```

- *proxy* 返回的是一个新对象， 可以通过操作返回的新的对象达到目的
- *proxy* 有多达 *13* 种拦截方法

总结：

- *Object.defineProperty* 无法监控到数组下标的变化，导致通过数组下标添加元素，不能实时响应

- *Object.defineProperty* 只能劫持对象的属性，从而需要对每个对象，每个属性进行遍历，如果，属性值是对象，还需要深度遍历。*Proxy* 可以劫持整个对象，并返回一个新的对象。
- *Proxy* 不仅可以代理对象，还可以代理数组。还可以代理动态增加的属性。

227. *intanceof* 操作符的实现原理及实现

参考答案：

instanceof 主要作用就是判断一个实例是否属于某种类型

例如：

```
let Dog = function(){}
let tidy = new Dog()
tidy instanceof Dog //true
```

intanceof 操作符实现原理

```
function wonderFulInstanceOf(instance, constructorFn) {
 let constructorFnProto = constructorFn.prototype; // 取右表达式的 prototype 值，函数构造器指向的function
 instanceProto = instance.__proto__; // 取左表达式的__proto__值，实例的__proto__
 while (true) {
  if (instanceProto === null) {
        return false;
    }
    if (instanceProto === constructorFnProto) {
        return true;
    }
    instanceProto = instanceProto.__proto__
 }
}
```

其实 *instanceof* 主要的实现原理就是只要 *constructorFn* 的 *prototype* 在*instance*的原型链上即可。

因此，*instanceof* 在查找的过程中会遍历左边变量的原型链，直到找到右边变量的 *prototype*，如果查找失败，则会返回 *false*，告诉我们左边变量并非是右边变量的实例。

228. 强制类型转换规则?

参考答案：

首先需要了解隐式转换所调用的函数。

当程序员显式调用 Boolean(value)、Number(value)、String(value) 完成的类型转换，叫做显示类型转换。

当通过 new Boolean(value)、new Number(value)、new String(value) 传入各自对应的原始类型的值，可以实现“装箱”，将原始类型封装成一个对象。

其实这三个函数不仅仅可以当作构造函数，它们可以直接当作普通的函数来使用，将任何类型的参数转化成原始类型的值：

```
Boolean('sdfsd');  //  true
Number("23");  //  23
String({a:24});  //  "[object Object]"
```

其实这三个函数用于类型转换的时候，调用的就是 js 内部的 *ToBoolean (argument)*、*ToNumber (argument)*、*ToString (argument)* 方法，从而达到显式转换的效果。

229. *Object.is*( ) 与比较操作符 “===”、“==” 的区别

参考答案：

== (或者 !=) 操作在需要的情况下自动进行了类型转换。=== (或 !==)操作不会执行任何转换。

===在比较值和类型时，可以说比==更快。

而在*ES6*中，*Object.is*( ) 类似于 ===，但在三等号判等的基础上特别处理了 *NaN* 、-0 和 +0 ，保证 -0 和 +0 不再相同，但 *Object.is(NaN, NaN)* 会返回 *true*。

230. + 操作符什么时候用于字符串的拼接?

参考答案：

在有一边操作数是字符串时会进行字符串拼接。

示例代码：

```
console.log(5 + '5', typeof (5 + '5')); // 55  string
```

231. *object.assign* 和扩展运算法是深拷贝还是浅拷贝

参考答案：

这两个方式都是浅拷贝。

在拷贝的对象只有一层时是深拷贝，但是一旦对象的属性值又是一个对象，也就是有两层或者两层以上时，就会发现这两种方式都是浅拷贝。

233. 如果 *new* 一个箭头函数的会怎么样

参考答案：

会报错，因为箭头函数无法作为构造函数。

234. 扩展运算符的作用及使用场景

参考答案：

扩展运算符是三个点(...)，主要用于展开数组，将一个数组转为参数序列。

扩展运算符使用场景：

- 代替数组的 *apply* 方法

- 合并数组
- 复制数组
- 把 *arguments* 或 *NodeList* 转为数组
- 与解构赋值结合使用
- 将字符串转为数组

235. *Proxy* 可以实现什么功能?

参考答案：

Proxy 是 *ES6* 中新增的一个特性。*Proxy* 让我们能够以简洁易懂的方式控制外部对对象的访问。其功能非常类似于设计模式中的代理模式。

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。

使用 *Proxy* 的好处是对象只需关注于核心逻辑，一些非核心的逻辑（如：读取或设置对象的某些属性前记录日志；设置对象的某些属性值前，需要验证；某些属性的访问控制等）可以让 *Proxy* 来做。从而达到关注点分离，降级对象复杂度的目的。

Proxy 的基本语法如下：

```
var proxy = new Proxy(target, handler);
```

通过构造函数来生成 *Proxy* 实例，构造函数接收两个参数。*target* 参数是要拦截的目标对象，*handler* 参数也是一个对象，用来定制拦截行为。

Vue 3.0 主要就采用的 *Proxy* 特性来实现响应式，相比以前的 *Object.defineProperty* 有以下优点：

- 可以劫持整个对象，并返回一个新的对象
- 有 *13* 种劫持操作

236. 对象与数组的解构的理解

参考答案：

解构是 *ES6* 的一种语法规则，可以将一个对象或数组的某个属性提取到某个变量中。

解构对象示例：

```
//  var/let/const{属性名}=被解构的对象
const user = {
 name: "abc",
 age: 18,
 sex: "男",
 address: {
     province: "重庆",
     city: "重庆"
 }
}
let { name, age, sex, address} = user;
console.log(name, age, sex, address);
```

解构数组示例：

```
const [a, b, c] = [1, 2, 3];
```

237. 如何提取高度嵌套的对象里的指定属性?

参考答案：

一般会使用递归的方式来进行查找。下面是一段示例代码：

```
function findKey(data, field) {
 let finding = '';
 for (const key in data) {
     if (key === field) {
         finding = data[key];
     }
     if (typeof (data[key]) === 'object') {
         finding = findKey(data[key], field);
     }
     if (finding) {
         return finding;
     }
 }
 return null;
}
// 测试
console.log(findKey({
 name: 'zhangsan',
 age: 18,
 stuInfo: {
     stuNo: 1,
     classNo: 2,
     score: {
         htmlScore: 100,
         cssScore: 90,
```

```
            jsScore: 95
        }
    }
}, 'cssScore')); // 90
```

238. *Unicode*、*UTF-8*、*UTF-16*、*UTF-32* 的区别?

参考答案：

Unicode **为世界上所有字符都分配了一个唯一的数字编号**，这个编号范围从 *0x000000* 到 *0x10FFFF* (十六进制)，有 *110* 多万，每个字符都有一个唯一的 *Unicode* 编号，这个编号一般写成 *16* 进制，在前面加上 U+。例如：“马”的 *Unicode* 是 *U+9A6C*。 *Unicode* 就相当于一张表，建立了字符与编号之间的联系。

***Unicode* 本身只规定了每个字符的数字编号是多少，并没有规定这个编号如何存储。**

那我们可以直接把 *Unicode* 编号直接转换成二进制进行存储，怎么对应到二进制表示呢?

Unicode 可以使用的编码有三种，分别是：

- *UFT-8*：一种变长的编码方案，使用 *1~6* 个字节来存储；
- *UFT-32*：一种固定长度的编码方案，不管字符编号大小，始终使用 *4* 个字节来存储；
- *UTF-16*：介于 *UTF-8* 和 *UTF-32* 之间，使用 *2* 个或者 *4* 个字节来存储，长度既固定又可变。

239. 为什么函数的 *arguments* 参数是类数组而不是数组？如何遍历类数组?

参考答案：

首先了解一下什么是数组对象和类数组对象。

数组对象：使用单独的变量名来存储一系列的值。从 *Array* 构造函数中继承了一些用于进行数组操作的方法。

例如：

```
var mycars = new Array();
mycars[0] = "zhangsan";
mycars[1] = "lisi";
mycars[2] = "wangwu";
```

类数组对象：**对于一个普通的对象来说，如果它的所有 property 名均为正整数，同时也有相应的length属性，那么虽然该对象并不是由Array构造函数所创建的，它依然呈现出数组的行为，在这种情况下，这些对象被称为“类数组对象”。**

两者区别

- 一个是对象，一个是数组
- 数组的*length*属性，当新的元素添加到列表中的时候，其值会自动更新。类数组对象的不会。
- 设置数组的*length*属性可以扩展或截断数组。

- 数组也是*Array*的实例可以调用*Array*的方法，比如*push*、*pop*等等

所以说*arguments*对象不是一个 *Array* 。它类似于*Array*，但除了*length*属性和索引元素之外没有任何*Array*属性。

可以使用 *for...in* 来遍历 *arguments* 这个类数组对象。

240. *escape*、*encodeURI*、*encodeURIComponent* 的区别

参考答案：

escape 除了 *ASCII* 字母、数字和特定的符号外，对传进来的字符串全部进行转义编码，因此如果想对 *URL* 编码，最好不要使用此方法。

encodeURI 用于编码整个 *URI*，因为 *URI* 中的合法字符都不会被编码转换。

encodeURIComponent 方法在编码单个*URIComponent*（指请求参数）应当是最常用的，它可以讲参数中的中文、特殊字符进行转义，而不会影响整个 *URL*。

241. *use strict* 是什么意思？使用它区别是什么?

参考答案：

use strict 代表开启严格模式，这种模式使得 *Javascript* 在更严格的条件下运行，实行更严格解析和错误处理。

开启“严格模式”的优点：

- 消除 *Javascript* 语法的一些不合理、不严谨之处，减少一些怪异行为;
- 消除代码运行的一些不安全之处，保证代码运行的安全；
- 提高编译器效率，增加运行速度；
- 为未来新版本的 *Javascript* 做好铺垫。

242. *for...in* 和 *for...of* 的区别

参考答案：

JavaScript 原有的 *for...in* 循环，只能获得对象的键名，不能直接获取键值。*ES6* 提供 *for...of* 循环，允许遍历获得键值。

例如：

```
var arr = ['a', 'b', 'c', 'd'];

for (let a in arr) {
console.log(a); // 0 1 2 3
}

for (let a of arr) {
console.log(a); // a b c d
}
```

243. *ajax*、*axios*、*fetch* 的区别

> 参考答案：
>
> *ajax* 是指一种创建交互式网页应用的网页开发技术，并且可以做到无需重新加载整个网页的情况下，能够更新部分网页，也叫作局部更新。
>
> 使用 *ajax* 发送请求是依靠于一个对象，叫 *XmlHttpRequest* 对象，通过这个对象我们可以从服务器获取到数据，然后再渲染到我们的页面上。现在几乎所有的浏览器都有这个对象，只有 *IE7* 以下的没有，而是通过 *ActiveXObject*这个对象来创建的。
>
> *Fetch* 是 *ajax* 非常好的一个替代品，基于 *Promise* 设计，使用 *Fetch* 来获取数据时，会返回给我们一个 *Pormise*对象，但是 *Fetch* 是一个低层次的 *API*，想要很好的使用 *Fetch*，需要做一些封装处理。
>
> 下面是 *Fetch* 的一些缺点
>
> - *Fetch* 只对网络请求报错，对 *400*，*500* 都当做成功的请求，需要封装去处理
> - *Fetch* 默认不会带 *cookie*，需要添加配置项。
> - *Fetch* 不支持 *abort*，不支持超时控制，使用 *setTimeout* 及 *Promise.reject* 的实现超时控制并不能阻止请求过程继续在后台运行，造成了流量的浪费。
> - *Fetch* 没有办法原生监测请求的进度，而 *XHR* 可以。
>
> *Vue2.0* 之后，*axios* 开始受到更多的欢迎了。其实 *axios* 也是对原生 *XHR* 的一种封装，不过是 *Promise* 实现版本。它可以用于浏览器和 *nodejs* 的 *HTTP* 客户端，符合最新的 *ES* 规范。

244. 下面代码的输出是什么？（*D*）

```
function sayHi() {
  console.log(name);
  console.log(age);
  var name = "Lydia";
  let age = 21;
}

sayHi();
```

- A: *Lydia* 和 *undefined*
- B: *Lydia* 和 *ReferenceError*
- C: *ReferenceError* 和 *21*
- D: *undefined* 和 *ReferenceError*

> **分析：**
>
> 在 *sayHi* 函数内部，通过 *var* 声明的变量 *name* 会发生变量提升，*var name* 会提升到函数作用域的顶部，其默认值为 *undefined*。因此输出 *name* 时得到的值为 *undefined*；
>
> *let* 声明的 *age* 不会发生变量提升，在输出 *age* 时该变量还未声明，因此会抛出 *ReferenceError* 的报错。

245. 下面代码的输出是什么？（ *C* ）

```
for (var i = 0; i < 3; i++) {
  setTimeout(() => console.log(i), 1);
}

for (let i = 0; i < 3; i++) {
  setTimeout(() => console.log(i), 1);
}
```

- A: *0 1 2* 和 *0 1 2*
- B: *0 1 2* 和 *3 3 3*
- C: *3 3 3* 和 *0 1 2*

> 分析：
>
> *JavaScript* 中的执行机制，*setTimeout* 为异步代码，因此在 *setTimeout* 执行时，*for* 循环已经执行完毕。
>
> 第一个 *for* 循环中的变量 *i* 通过 *var* 声明， 为全局变量，因此每一次的 *i++* 都会将全局变量 *i* 的值加 *1*，当第一个 *for* 执行完成后 *i* 的值为 *3*。所以再执行 *setTimeout* 时，输出 *i* 的值都为 *3*；
>
> 第二个 *for* 循环中的变量 *i* 通过 *let* 声明，为局部变量，因此每一次 *for* 循环时都会产生一个块级作用域，用来存储本次循环中新产生的 *i* 的值。当循环结束后，*setTimeout* 会沿着作用域链去对应的块级作用域中寻找对应的 *i* 值。

246. 下面代码的输出是什么？（ *B* ）

```
const shape = {
  radius: 10,
  diameter() {
    return this.radius * 2;
  },
  perimeter: () => 2 * Math.PI * this.radius
};

shape.diameter();
shape.perimeter();
```

- A: *20* 和 *62.83185307179586*
- B: *20* 和 *NaN*
- C: *20* 和 *63*
- D: *NaN* 和 *63*

> **分析：**
>
> *diameter* 作为对象的方法，其内部的 *this* 指向调用该方法的对象，因此 *this.raduus* 获取到的是 *shape.radius* 的值 *10*，再乘以 *2* 输出的值即为 *20*；

> *perimeter* 是一个箭头函数，其内部的 *this* 应该继承声明时所在上下文中的 *this*，在这里即继承全局的 *this*，因此 *this.radius* 值的为 *undefined*，*undefined* 与数值相乘后值为 *NaN*。

247. 下面代码的输出是什么？ （ *A* ）

```
+true;
!"Lydia";
```

- A: *1* 和 *false*
- B: *false* 和 *NaN*
- C: *false* 和 *false*

> 分析：
>
> 一元加号会将数据隐式转换为 *number* 类型，*true* 转换为数值为 *1*；
>
> 非运算符 *!* 会将数据隐式转换为 *boolean* 类型后进行取反，*"Lydia"* 转换为布尔值为 *true*，取反后为 *false*。

248. 哪个选项是不正确的？ （ *A* ）

```
const bird = {
  size: "small"
};

const mouse = {
  name: "Mickey",
  small: true
};
```

- A: *mouse.bird.size*
- B: *mouse[bird.size]*
- C: *mouse[bird["size"]]*
- D: 以上选项都对

> 分析：
>
> *mouse* 对象中没有 *bird* 属性，当访问一个对象不存在的属性时值为 *undefined*，因此 *mouse.bird* 的值为 *undefined*，而 *undefined* 作为原始数据类型没有 *size* 属性，因此再访问 *undefined.size* 时会报错。

249. 下面代码的输出是什么？ （ *A* ）

```
let c = { greeting: "Hey!" };
let d;

d = c;
c.greeting = "Hello";
console.log(d.greeting);
```

- A: *Hello*
- B: *undefined*
- C: *ReferenceError*
- D: *TypeError*

> **分析：**
>
> 在 *JavaScript* 中，复杂类型数据在进行赋值操作时，进行的是「引用传递」，因此变量 *d* 和 *c* 指向的是同一个引用。当 *c* 通过引用去修改了数据后，*d* 再通过引用去访问数据，获取到的实际就是 *c* 修改后的数据。

250. 下面代码的输出是什么？ （ *C* ）

```
let a = 3;
let b = new Number(3);
let c = 3;

console.log(a == b);
console.log(a === b);
console.log(b === c);
```

- A: *true false true*
- B: *false false true*
- C: *true false false*
- D: *false true true*

> **分析：**
>
> *new Number()* 是 *JavaScript* 中一个内置的构造函数。变量 *b* 虽然看起来像一个数字，但它并不是一个真正的数字：它有一堆额外的功能，是一个对象。
>
> == 会触发隐式类型转换，右侧的对象类型会自动转换为 *Number* 类型，因此最终返回 *true*。
>
> === 不会触发隐式类型转换，因此在比较时由于数据类型不相等而返回 *false*。

251. 下面代码的输出是什么？ （ *D* ）

```
class Chameleon {
  static colorChange(newColor) {
    this.newColor = newColor;
  }

  constructor({ newColor = "green" } = {}) {
    this.newColor = newColor;
  }
}

const freddie = new Chameleon({ newColor: "purple" });
freddie.colorChange("orange");
```

- A: *orange*
- B: *purple*
- C: *green*
- D: *TypeError*

> **分析**：
>
> *colorChange* 方法是静态的。 静态方法仅在创建它们的构造函数中存在，并且不能传递给任何子级。 由于 *freddie*是一个子级对象，函数不会传递，所以在 *freddie* 实例上不存在 *colorChange* 方法：抛出*TypeError*。

252. 下面代码的输出是什么？ （ *A* ）

```
let greeting;
greetign = {}; // Typo!
console.log(greetign);
```

- A: *{}*
- B: *ReferenceError: greetign is not defined*
- C: *undefined*

> **分析**：
>
> 控制台会输出空对象，因为我们刚刚在全局对象上创建了一个空对象！
>
> 当我们错误地将 *greeting* 输入为 *greetign* 时，*JS* 解释器实际上在浏览器中将其视为 *window.greetign = {}*。

253. 当我们执行以下代码时会发生什么？ （ *A* ）

```
function bark() {
console.log("Woof!");
}

bark.animal = "dog";
```

- A 什么都不会发生
- B: *SyntaxError. You cannot add properties to a function this way.*
- C: *undefined*
- D: *ReferenceError*

> **分析**：
>
> 因为函数也是对象！（原始类型之外的所有东西都是对象）
>
> 函数是一种特殊类型的对象，我们可以给函数添加属性，且此属性是可调用的。

254. 下面代码的输出是什么？ （ *A* ）

```
function Person(firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
}

const member = new Person("Lydia", "Hallie");
Person.getFullName = () => this.firstName + this.lastName;

console.log(member.getFullName());
```

- A: *TypeError*
- B: *SyntaxError*
- C: *Lydia Hallie*
- D: *undefined undefined*

> **分析：**
>
> *Person.getFullName* 是将方法添加到了函数身上，因此当我们通过实例对象 *member* 去调用该方法时并不能找到该方法。

255. 下面代码的输出是什么？ （ *A* ）

```
function Person(firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
}

const lydia = new Person("Lydia", "Hallie");
const sarah = Person("Sarah", "Smith");

console.log(lydia);
console.log(sarah);
```

- A: *Person { firstName: "Lydia", lastName: "Hallie" }* 和 *undefined*
- B: *Person { firstName: "Lydia", lastName: "Hallie" }* 和 *Person { firstName: "Sarah", lastName: "Smith" }*
- C: *Person { firstName: "Lydia", lastName: "Hallie" }* 和 *{}*
- D: *Person { firstName: "Lydia", lastName: "Hallie" }* 和 *ReferenceError*

> **分析：**
>
> *lydia* 是调用构造函数后得到的实例对象，拥有 *firstName* 和 *lastName* 属性；
>
> *sarah* 是调用普通函数后得到的返回值，而 *Person* 作为普通函数没有返回值；

256. 事件传播的三个阶段是什么？（ *D* ）

- A: 目标 > 捕获 > 冒泡
- B: 冒泡 > 目标 > 捕获
- C: 目标 > 冒泡 > 捕获
- D: 捕获 > 目标 > 冒泡

257. 下面代码的输出是什么？（ *C* ）

```
function sum(a, b) {
  return a + b;
}

sum(1, "2");
```

- A: *NaN*
- B: *TypeError*
- C: *"12"*
- D: *3*

> **分析：**
>
> 任意数据类型在跟 *String* 做 + 运算时，都会隐式转换为 *String* 类型。
>
> 即 *a* 所对应的 *Number* 值 *1*，被隐式转换为了 *String* 值 "1"，最终字符串拼接的到 "12"。

258. 下面代码的输出是什么？（ *C* ）

```
let number = 0;
console.log(number++);
console.log(++number);
console.log(number);
```

- A: *1 1 2*
- B: *1 2 2*
- C: *0 2 2*
- D: *0 1 2*

> **分析：**
>
> ++ 后置时，先输出，后加 *1*；++ 前置时，先加 *1*，后输出；
>
> 第一次输出的值为 0，输出完成后 *number* 加 *1* 变为 *1*。
>
> 第二次输出，*number* 先加 *1* 变为 *2*，然后输出值 *2*。
>
> 第三次输出，*number* 值没有变化，还是 *2*。

259. 下面代码的输出是什么？ （ *B* ）

```
function getPersonInfo(one, two, three) {
  console.log(one);
  console.log(two);
  console.log(three);
}

const person = "Lydia";
const age = 21;

getPersonInfo`${person} is ${age} years old`;
```

- A: *Lydia 21 ["", "is", "years old"]*
- B: *["", "is", "years old"] Lydia 21*
- C: *Lydia ["", "is", "years old"] 21*

> **分析：**
>
> 如果使用标记的模板字符串，则第一个参数的值始终是字符串值的数组。 其余参数获取传递到模板字符串中的表达式的值！

260. 下面代码的输出是什么？ （ *C* ）

```
function checkAge(data) {
  if (data === { age: 18 }) {
    console.log("You are an adult!");
  } else if (data == { age: 18 }) {
    console.log("You are still an adult.");
  } else {
    console.log(`Hmm.. You don't have an age I guess`);
  }
}

checkAge({ age: 18 });
```

- A: *You are an adult!*
- B: *You are still an adult.*
- C: *Hmm.. You don't have an age I guess*

> **分析：**
>
> 在比较相等性时，原始类型通过它们的值进行比较，而对象通过它们的引用进行比较。
>
> *data* 和条件中的 *{ age: 18 }* 两个不同引用的对象，因此永远都不相等。

261. 下面代码的输出是什么？（*C*）

```
function getAge(...args) {
  console.log(typeof args);
}

getAge(21);
```

- A: *"number"*
- B: *"array"*
- C: *"object"*
- D: *"NaN"*

> 分析：
>
> *ES6* 中的不定参数（*...args*）返回的是一个数组。
>
> *typeof* 检查数组的类型返回的值是 *object*。

262. 下面代码的输出是什么？（*C*）

```
function getAge() {
  "use strict";
  age = 21;
  console.log(age);
}

getAge();
```

- A: *21*
- B: *undefined*
- C: *ReferenceError*
- D: *TypeError*

> 分析：
>
> *"use strict"* 严格模式中，使用未声明的变量会引发报错。

263. 下面代码的输出是什么？（*A*）

```
const sum = eval("10*10+5");
```

- A: *105*
- B: *"105"*
- C: *TypeError*
- D: *"10*10+5"*

> 分析：
>
> *eval* 方法会将字符串当作 *JavaScript* 代码进行解析。

264. *cool_secret* 可以访问多长时间？（*B*）

```
sessionStorage.setItem("cool_secret", 123);
```

- A：永远，数据不会丢失。
- B：用户关闭选项卡时。
- C：当用户关闭整个浏览器时，不仅是选项卡。
- D：用户关闭计算机时。

> 分析：
>
> *sessionStorage* 是会话级别的本地存储，当窗口关闭，则会话结束，数据删除。

265. 下面代码的输出是什么？（*B*）

```
var num = 8;
var num = 10;

console.log(num);
```

- A: *8*
- B: *10*
- C: *SyntaxError*
- D: *ReferenceError*

> 分析：
>
> *var* 声明的变量允许重复声明，但后面的值会覆盖前面的值。

266. 下面代码的输出是什么？（*C*）

```
const obj = { 1: "a", 2: "b", 3: "c" };
const set = new Set([1, 2, 3, 4, 5]);

obj.hasOwnProperty("1");
obj.hasOwnProperty(1);
set.has("1");
set.has(1);
```

- A: *false true false true*
- B: *false true true true*
- C: *true true false true*

- D: *true true true true*

267. 下面代码的输出是什么？（*C*）

```
const obj = { a: "one", b: "two", a: "three" };
console.log(obj);
```

- A: *{ a: "one", b: "two" }*
- B: *{ b: "two", a: "three" }*
- C: *{ a: "three", b: "two" }*
- D: *SyntaxError*

> 分析：
>
> 如果对象有两个具有相同名称的键，则后面的将替前面的键。它仍将处于第一个位置，但具有最后指定的值。

268. 下面代码的输出是什么？（*C*）

```
for (let i = 1; i < 5; i++) {
  if (i === 3) continue;
  console.log(i);
}
```

- A: *1 2*
- B: *1 2 3*
- C: *1 2 4*
- D: *1 3 4*

> 分析：
>
> 当 *i* 的值为 *3* 时，进入 *if* 语句执行 *continue*，结束本次循环，立即进行下一次循环。

269. 下面代码的输出是什么？（*A*）

```
String.prototype.giveLydiaPizza = () => {
  return "Just give Lydia pizza already!";
};

const name = "Lydia";

name.giveLydiaPizza();
```

- A: *"Just give Lydia pizza already!"*
- B: *TypeError: not a function*
- C: *SyntaxError*

- D: *undefined*

> 分析：
>
> *String* 是一个内置的构造函数，我们可以为它添加属性。 我们给它的原型添加了一个方法。 原始类型的字符串自动转换为字符串对象，由字符串原型函数生成。 因此，所有字符串（字符串对象）都可以访问该方法！
>
> 当使用基本类型的字符串调用 *giveLydiaPizza* 时，实际上发生了下面的过程：
>
> - 创建一个 *String* 的包装类型实例
> - 在实例上调用 *substring* 方法
> - 销毁实例

270. 下面代码的输出是什么？ （ *B* ）

```
const a = {};
const b = { key: "b" };
const c = { key: "c" };

a[b] = 123;
a[c] = 456;

console.log(a[b]);
```

- A: *123*
- B: *456*
- C: *undefined*
- D: *ReferenceError*

> 分析：
>
> 当 *b* 和 *c* 作为一个对象的键时，会自动转换为字符串，而对象自动转换为字符串化时，结果都为 *[Object object]*。因此 *a[b]* 和 *a[c]* 其实都是同一个属性 *a["Object object"]*。
>
> 对象同名的属性后面的值会覆盖前面的，因此最终 *a["Object object"]* 的值为 *456*。

271. 下面代码的输出是什么？ （ *B* ）

```
const foo = () => console.log("First");
const bar = () => setTimeout(() => console.log("Second"));
const baz = () => console.log("Third");

bar();
foo();
baz();
```

- A: *First Second Third*
- B: *First Third Second*

- C: *Second First Third*
- D: *Second Third First*

> 分析：
>
> *bar* 函数中执行的是一段异步代码，按照 *JavaScript* 中的事件循环机制，主线程中的所有同步代码执行完成后才会执行异步代码。因此 *"Second"* 最后输出。

272. 单击按钮时 *event.target* 是什么？（ *C* ）

```
<div onclick="console.log('first div')">
  <div onclick="console.log('second div')">
    <button onclick="console.log('button')">
      Click!
    </button>
  </div>
</div>
```

- A: *div* 外部
- B: *div* 内部
- C: *button*
- D: 所有嵌套元素的数组

> 分析：
>
> *event.target* 指向的是事件目标，即触发事件的元素。因此点击 触发事件的也就是 。

273. 单击下面的 *html* 片段打印的内容是什么？（ *A* ）

```
<div onclick="console.log('div')">
  <p onclick="console.log('p')">
    Click here!
  </p>
</div>
```

- A: *p div*
- B: *div p*
- C: *p*
- D: *div*

> 分析：
>
> *onclick* 绑定的事件为冒泡型事件。因此当点击 *p* 标签时，事件会从事件目标开始依次往外触发。

274. 下面代码的输出是什么？ （ *D* ）

```
const person = { name: "Lydia" };

function sayHi(age) {
  console.log(`${this.name} is ${age}`);
}

sayHi.call(person, 21);
sayHi.bind(person, 21);
```

- A: *undefined is 21 Lydia is 21*
- B: *function function*
- C: *Lydia is 21 Lydia is 21*
- D: *Lydia is 21 function*

> 分析：
>
> *call* 和 *bind* 都可以修改 *this* 的指向，但区别在于 *call* 方法会立即执行，而 *bind* 会返回一个修改后的新函数。

275. 下面代码的输出是什么？ （ *B* ）

```
function sayHi() {
  return (() => 0)();
}

typeof sayHi();
```

- A: *"object"*
- B: *"number"*
- C: *"function"*
- D: *"undefined"*

> 分析：
>
> *return* 后是一个 *IIFE*，其返回值是 *0*，因此 *sayHi* 函数中返回的是一个 *0*。*typeof* 检测 *sayHi* 返回值类型即为 *number*。

276. 下面这些值哪些是假值？ （ *A* ）

```
0;
new Number(0);
("");
(" ");
new Boolean(false);
undefined;
```

- A: *0 "" undefined*
- B: *0 new Number(0) "" new Boolean(false) undefined*
- C: *0 "" new Boolean(false) undefined*
- D: 所有都是假值。

> **分析：**
>
> *JavaScript* 中假值只有 *6* 个：*false*、""、*null*、*undefined*、*NaN*、*0*

278. 下面代码的输出是什么？ （ *B* ）

```
console.log(typeof typeof 1);
```

- A: *"number"*
- B: *"string"*
- C: *"object"*
- D: *"undefined"*

> 分析：
>
> *typeof 1* 返回 *"number"*，*typeof "number"* 返回 *"string"*

279. 下面代码的输出是什么？ （ *C* ）

```
const numbers = [1, 2, 3];
numbers[10] = 11;
console.log(numbers);
```

- A: *[1, 2, 3, 7 x null, 11]*
- B: *[1, 2, 3, 11]*
- C: *[1, 2, 3, 7 x empty, 11]*
- D: *SyntaxError*

> **分析：**
>
> 当你为数组中的元素设置一个超过数组长度的值时，*JavaScript* 会创建一个名为“空插槽”的东西。 这些位置的值实际上是 *undefined*，但你会看到类似的东西：
>
> ```
> [1, 2, 3, 7 x empty, 11]
> ```

> 这取决于你运行它的位置（每个浏览器有可能不同）。

280. 下面代码的输出是什么？（*A*）

```
(() => {
  let x, y;
  try {
    throw new Error();
  } catch (x) {
    (x = 1), (y = 2);
    console.log(x);
  }
  console.log(x);
  console.log(y);
})();
```

- A: *1 undefined 2*
- B: *undefined undefined undefined*
- C: *1 1 2*
- D: *1 undefined undefined*

> **分析：**
>
> *catch* 块接收参数 *x*。当我们传递参数时，这与变量的 *x* 不同。这个变量 *x* 是属于 *catch* 作用域的。
>
> 之后，我们将这个块级作用域的变量设置为 *1*，并设置变量 *y* 的值。 现在，我们打印块级作用域的变量 *x*，它等于 *1*。
>
> 在*catch* 块之外，*x* 仍然是 *undefined*，而 *y* 是 *2*。 当我们想在 *catch* 块之外的 *console.log(x)* 时，它返回 *undefined*，而 *y* 返回 *2*。

281. *JavaScript* 中的所有内容都是...（*A*）

- A：原始或对象
- B：函数或对象
- C：技巧问题！只有对象
- D：数字或对象

> 分析：
>
> *JavaScript* 只有原始类型和对象。

282. 下面代码的输出是什么？

```
[[0, 1], [2, 3]].reduce(
  (acc, cur) => {
    return acc.concat(cur);
  },
  [1, 2]
);
```

- A: *[0, 1, 2, 3, 1, 2]*
- B: *[6, 1, 2]*
- C: *[1, 2, 0, 1, 2, 3]*
- D: *[1, 2, 6]*

> **分析：**
>
> *[1,2]* 是我们的初始值。 这是我们开始执行 *reduce* 函数的初始值，以及第一个 *acc* 的值。 在第一轮中，*acc* 是 *[1,2]*，*cur* 是 *[0,1]*。 我们将它们连接起来，结果是 *[1,2,0,1]*。
>
> 然后，*acc* 的值为 *[1,2,0,1]*，*cur* 的值为 *[2,3]*。 我们将它们连接起来，得到 *[1,2,0,1,2,3]*。

283. 下面代码的输出是什么？ （ ***B*** ）

```
!!null;
!!"";
!!1;
```

- A: *false true false*
- B: *false false true*
- C: *false true true*
- D: *true true false*

> **分析：**
>
> *null* 是假值。 *!null* 返回 *true*。 *!true* 返回 *false*。
>
> *""* 是假值。 *!""* 返回 *true*。 *!true* 返回 *false*。
>
> *1* 是真值。 *!1* 返回 *false*。 *!false* 返回 *true*。

284. *setInterval* 方法的返回值什么？ （ ***A*** ）

```
setInterval(() => console.log("Hi"), 1000);
```

- A：一个唯一的 *id*
- B：指定的毫秒数
- C：传递的函数
- D：*undefined*

> 分析：
>
> 它返回一个唯一的 *id*。 此 *id* 可用于使用 *clearInterval()* 函数清除该定时器。

285. 下面代码的返回值是什么？（*A*）

```
[..."Lydia"];
```

- A: *["L", "y", "d", "i", "a"]*
- B: *["Lydia"]*
- C: *[[], "Lydia"]*
- D: *[["L", "y", "d", "i", "a"]]*

> 分析：
>
> 字符串是可迭代的。 扩展运算符将迭代的每个字符映射到一个元素。

286. *document.write* 和 *innerHTML* 有哪些区别?

> 参考答案：
>
> *document.write* 和 *innerHTML* 都能将 *HTML* 字符串解析为 *DOM* 树，再将 *DOM* 树插入到某个位置，但两种在执行细节上还是有许多不同。
>
> 1）*write()* 方法存在于 *Document* 对象中，*innerHTML* 属性存在于 *Element* 对象中；
>
> 2）*document.write* 会将解析后的 *DOM* 树插入到文档中调用它的脚本元素的位置，而 *innerHTML* 会将 *DOM* 树插入到指定的元素内；
>
> 3）*document.write* 会将多次调用的字符串参数自动连接起来，*innerHTML* 要用赋值运算符 *"+="* 拼接；
>
> 4）只有当文档还在解析时，才能使用 *document.write*，否则 *document.write* 的值会将当前文档覆盖掉，而 *innerHTML* 属性则没有这个限制；

> 注：也可以参阅前面第 *157* 题答案

287. 假设有两个变量 *a* 和 *b*，他们的值都是数字，如何在不借用第三个变量的情况下，将两个变量的值对调?

> 参考答案：
>
> 方法一：

```
a = a + b;
b = a - b;
a = a - b;
```

> 方法二（*ES6* 中的解构）：

```
[a, b] = [b, a]
```

288. 前端为什么提倡模块化开发?

> 参考答案：
>
> 模块化能将一个复杂的大型系统分解成一个个高内聚、低耦合的简单模块，并且每个模块都是独立的，用于完成特定的功能。模块化后的系统变得更加可控、可维护、可扩展，程序代码也更简单直观，可读性也很高，有利于团队协作开发。*ES6* 模块化的出现，使得前端能更容易、更快速的实现模块化开发。

289. 请解释 *JSONP* 的原理，并用代码描述其过程。

> 参考答案：
>
> *JSONP*（*JSON with padding*）是一种借助 元素实现跨域的技术，它不会使用 *XHR* 对象。之所以能实现跨域，主要是因为 元素有以下两个特点：
>
> 1）它的 *src* 属性能够访问任何 *URL* 资源，不会受同源策略的限制；
>
> 2）如果访问的资源包含 *JavaScript* 代码，那么在下载下来后会自动执行；
>
> *JSONP* 就是基于这两点，再与服务器配合来实现跨域请求的，它的执行步骤可分为以下 *6* 步：
>
> 1）定义一个回调函数；
>
> 2）用 *DOM* 方法动态创建一个 元素；
>
> 3）通过 元素的 *src* 属性指定要请求的 *URL*，并且将回调函数的名称作为一个参数传递过去；
>
> 4）将 元素插入到当前文档中，开始请求；
>
> 5）服务器接收到传递过来的参数，然后将回调函数和数据以调用的形式输出；
>
> 6）当 元素接收到响应中的脚本代码后，就会自动的执行它们；

290. 列举几种 *JavaScript* 中数据类型的强制转换和隐式转换。

> 参考答案：
>
> 强制转换：
>
> - 转换为 *number*：*parseInt()*、*parseFloat()*、*Number()*
> - 转换为 *string*：*String()*、*toString()*
> - 转换为 *boolean*：*Boolean()*
>
> 隐式转换：
>
> - 隐式转换为 *number*：算术运算/比较运算，例如加、减、乘、除、相等（==）、大于、小于等；
> - 隐式转换为 *string*：与字符串拼接，例如 + ""；
> - 隐式转换为 *boolean*：逻辑运算，例如或（||）、与（&&）、非（!）；

291. 分析以下代码的执行结果并解释为什么。

```
var a = {n: 1};
var b = a;
a.x = a = {n: 2};

console.log(a.x)
console.log(b.x)
```

> 参考答案：
>
> 运行结果：
>
> *undefined*、*{n: 2}*
>
> 分析：
>
> 首先，*a* 和 *b* 同时引用了 *{n: 1}* 对象，接着执行到 *a.x = a = {n: 2}* 语句，虽然赋值是从右到左执行，但是点（.）的优先级比赋值符（=）要高，所以这里首先执行 *a.x*，相当于为 *a*（或者 *b*）所指向的 *{n:1}* 对象新增了一个属性 *x*，即此时对象将变为 *{n: 1; x: undefined}*。然后按正常情况，从右到左进行赋值，此时执行 *a = {n: 2}* 的时候，a的引用改变，指向了新对象 *{n: 2}*，而 *b* 依然指向的是旧对象 *{n: 1; x: undefined}*。之后再执行 *a.x = {n: 2}* 的时候，并不会重新解析一遍 *a*，而是沿用最初解析 *a.x* 时候的 *a*，即旧对象 *{n: 1; x: undefined}*，故此时旧对象的 *x* 的值变为*{n: 2}*，旧对象为 *{n: 1; x: {n: 2}}*，它依然被 *b* 引用着。
>
> 最后，*a* 指向的对象为 *{n: 2}*，*b* 指向的对象为 *{n: 1; x: {n: 2}}*。因此输出 *a.x* 值为 *undefined*，输出 *b.x* 值为 *{n: 2}*。

292. 分析以下代码的执行结果并解释为什么。

```
// example 1
var a = {}, b = '123', c = 123;
a[b] = 'b';
a[c] = 'c';
console.log(a[b]);

// example 2
var a = {}, b = Symbol('123'), c = Symbol('123');
a[b] = 'b';
a[c] = 'c';
console.log(a[b]);

// example 3
var a = {}, b = {key:'123'}, c = {key:'456'};
a[b] = 'b';
a[c] = 'c';
console.log(a[b]);
```

> 参考答案：
>
> 运行结果：

> *example 1*：*c*
>
> *example 2*：*b*
>
> *example 3*：*c*
>
> 分析：
>
> 这题考察的是对象的键名的转换。
>
> - 对象的键名只能是字符串和 *Symbol* 类型。
> - 其他类型的键名会被转换成字符串类型。
> - 对象转字符串默认会调用 *String* 方法。
>
> 因此 *example 1* 中 *c* 作为键名后也是 *'123'*，直接覆盖 *a[b]* 的值；而 *example 2* 中，*Symbol* 作为 *ES6* 中新增的基本数据类型，它的特点就是唯一，*Symbol()* 方法生成的值都是唯一的，里面的参数不会影响结果。因此在 *example 2* 中 *b* 和 *c* 是两个不同的键名；*example 3* 中，对象不能作为键名，因此 *b* 和 *c* 都会通过 *String()* 方法转为字符串 *[object Object]*。

293. 下面的代码打印什么内容？为什么？

```
var b = 10;
(function b() {
  b = 20;
  console.log(b)
})()
```

> 参考答案：
>
> 运行结果：
>
> *function b() { b = 20; console.log(b) }* 分析：
>
> 当 *JavaScript* 解释器遇到非匿名立即执行函数（题目中的 *b*）时，会创建一个辅助的特定对象，然后将函数名称当作这个对象的属性，因此函数内部可以访问到 *b*，但是这个值又是只读的，所以对他的赋值并不生效，所以打印的结果还是这个函数，并且外部的值也没有发生更改。

294. 下面代码中，*a* 在什么情况下会执行输出语句打印 *1*？

```
var a = ?;
if(a == 1 && a == 2 && a == 3){
  console.log(1);
}
```

> 参考答案：
>
> 分析：
>
> 这道题考查的知识点是：相等运算符（==）在作比较时会进行隐式转换，而如果操作数是引用类型，则会调用 *toString()* 或 *valueOf()* 方法对引用类型数据进行隐式转换。

```
// 方法一：利用 toString()
let a = {
i: 1,
toString () {
 return a.i++;
}
}
if(a == 1 && a == 2 && a == 3) {
console.log('1');
}

// 方法二：利用 valueOf()
let a = {
i: 1,
valueOf () {
 return a.i++
}
}
if(a == 1 && a == 2 && a == 3) {
console.log('1');
}

// 方法三：利用数组（这个是真的骚）
var a = [1,2,3];
a.join = a.shift;
if(a == 1 && a == 2 && a == 3) {
console.log('1');
}

// 方法四：利用 Symbol
let a = {[Symbol.toPrimitive]: ((i) => () => ++i) (0)};
if(a == 1 && a == 2 && a == 3) {
console.log('1');
}
```

方法一和方法二没啥解释的了，解释下方法三和方法四。

方法三：

a.join = a.shift 的目的是将数组的 *join* 方法替换成 *shift* 方法。因为数组在参与相等比较时也会通过 *toString()* 将数组转为字符串，而该字符串实际上是数组中每个元素的 *toString()* 返回值经调用 *join()* 方法拼接（由逗号隔开）组成。现在我们将 *join()* 方法替换为了 *shift()* 方法，也就意味着数组在通过 *toString()* 隐式转换后，得到是 *shift()* 的返回值，每次返回数组中的第一个元素，而原数组删除第一个值，正好可以使判断成立。

方法四：

ES6 中提供了 *11* 个内置的 *Symbo* 值，指向语言内部使用的方法。*Symbol.toPrimitive* 就是其中一个，它指向一个方法，当该对象被转为原始类型的值时，会调用这个方法，并返回该对象对应的原始类型值。这里就是改变这个属性，把它的值改为一个闭包返回的函数。

295. 介绍前端模块化的发展。

参考答案：

- *IIFE*：使用自执行函数来编写模块化（特点：在一个单独的函数作用域中执行代码，避免变量冲突）。

```
(function(){
  return { data:[] }
})()
```

- *AMD*：使用 *requireJS* 来编写模块化（特点：依赖必须提前声明好）。

```
define('./index.js',function(code){
  // code 就是index.js 返回的内容
})
```

- *CMD*：使用 *seaJS* 来编写模块化（特点：支持动态引入依赖文件）。

```
define(function(require, exports, module) {
  var indexCode = require('./index.js');
});
```

- *CommonJS*：*nodejs* 中自带的模块化。

```
var fs = require('fs');
```

- *UMD*：通用模块规范，整合了 *AMD* 和 *CommonJS* 模块化。

```
(function (global, factory) {
  if (typeof exports === 'object' && typeof module !== undefined) { //检查
CommonJS是否可用
    module.exports = factory(require('jquery'));
  } else if (typeof define === 'function' && define.amd) {      //检查AMD是否可
用
    define('toggler', ['jquery', factory])
  } else {       //两种都不能用，把模块添加到JavaScript的全局命名空间中。
    global.toggler = factory(global, factory);
  }
})(this, function ($) {
  function init() {

  }
  return {
    init: init
  }
});
```

- *webpack(require.ensure)*：*webpack 2.x* 版本中的代码分割。
- *ES Modules*： *ES6* 引入的模块化，支持 *import* 来引入另一个 *js* 。

296. 请指出 *document.onload* 和 document.ready 两个事件的区别

参考答案：

页面加载完成有两种事件：一是 *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）；二是 *onload*，指示页面包含图片等文件在内的所有元素都加载完成。

297. 表单元素的*readonly* 和 *disabled* 两个属性有什么区别?

参考答案：

- readonly
  :
  - 不可编辑，但可以选择和复制；
  - 值可以传递到后台；
- disabled：
  - 不能编辑，不能复制，不能选择；
  - 值不可以传递到后台；

298. 列举几种你知道的数组排序的方法。

参考答案：

```
// 方法一：选择排序
let ary = [5, 7, 8, 11, 3, 6, 4];
for (let i = 0; i < ary.length - 1; i++) {
for (let j = i + 1; j < ary.length; j++) {
 if (ary[i] < ary[j]) {
   [ary[i], ary[j]] = [ary[j], ary[i]];
 }
}
}
// 方法二：冒泡排序
let ary = [5, 7, 8, 11, 3, 6, 4];
for (let i = 1; i < ary.length; i++) {
for (let j = 0; j < ary.length - i; j++) {
 if (ary[j] < ary[j + 1]) {
   [ary[j], ary[j + 1]] = [ary[j + 1], ary[j]]
 }
}
}
```

299. 区分什么是“客户区坐标”、“页面坐标”、“屏幕坐标”?

> 参考答案：
>
> - 客户区坐标：鼠标指针在可视区中的水平坐标 (*clientX*) 和垂直坐标 (*clientY*)；
> - 页面坐标：鼠标指针在页面布局中的水平坐标 (*pageX*) 和垂直坐标 (*pageY*)；
> - 屏幕坐标：设备物理屏幕的水平坐标 (*screenX*) 和垂直坐标 (*screenY*)；

300. 如何编写高性能的 *JavaScript*?

> 参考答案：
>
> - 遵循严格模式："use strict"
> - 将 *JavaScript* 本放在页面底部，加快渲染页面
> - 将 *JavaScript* 脚本将脚本成组打包，减少请求
> - 使用非阻塞方式下载 *JavaScript* 脚本
> - 尽量使用局部变量来保存全局变量
> - 尽量减少使用闭包
> - 使用 *window* 对象属性方法时，省略 *window*
> - 尽量减少对象成员嵌套
> - 缓存 *DOM* 节点的访问
> - 通过避免使用 *eval()* 和 *Function()* 构造器
> - 给 *setTimeout()* 和 *setInterval()* 传递函数而不是字符串作为参数
> - 尽量使用直接量创建对象和数组
> - 最小化重绘 (*repaint*) 和回流 (*reflow*)

301. 下面的代码输出什么?

```
var a = function () { return 5 }
a.toString = function () { return 3 }
console.log(a + 7);
```

> 参考答案：
>
> 10
>
> 因为会自动调用 a 函数的 *toString* 方法。

注： 至此，JavaScript 内容就整理完了，目前正在整理关于网络方面的，大家可以期待一下。欢大家迎关注我哈😂，点击链接即可关注 法医，大家认真看哦，奥利给！💪