前言

我只想面个CV工程师，面试官偏偏让我挑战造火箭工程师，加上今年这个情况更是前后`两男`，但再难苟且的生活还要继续，饭碗还是要继续找的。在最近的面试中我一直在总结，每次面试回来也都会复盘，下面是我这几天遇到的面试知识点。但今天主题是标题所写的66条JavaScript知识点，由浅入深，整理了一周，每（zhěng）天（lǐ）整（bù）理（yì）10条（qiú）左（diǎn）右（zàn），希望对正在找工作的小伙伴有点帮助，文中如有表述不对，还请指出。

HTML&CSS：

- 浏览器内核
- 盒模型、flex布局、两/三栏布局、水平/垂直居中；
- BFC、清除浮动；
- css3动画、H5新特性。

JavaScript：

- 继承、原型链、this指向、设计模式、call, apply, bind,；
- new实现、防抖节流、let, var, const 区别、暂时性死区、event、loop；
- promise使用及实现、promise并行执行和顺序执行；
- async/await的优缺点；
- 闭包、垃圾回收和内存泄漏、数组方法、数组乱序, 数组扁平化、事件委托、事件监听、事件模型

Vue:

- vue数据双向绑定原理；
- vue computed原理、computed和watch的区别；
- vue编译器结构图、生命周期、vue组件通信；
- mvvm模式、mvc模式理解；
- vue dom diff、vuex、vue-router

网络：

- HTTP1, HTTP2, HTTPS、常见的http状态码；
- 浏览从输入网址到回车发生了什么；
- 前端安全（CSRF、XSS）
- 前端跨域、浏览器缓存、cookie, session, token, localstorage, sessionstorage；
- TCP连接(三次握手, 四次挥手)

性能相关

- 图片优化的方式
- 500 张图片，如何实现预加载优化
- 懒加载具体实现
- 减少http请求的方式
- webpack如何配置大型项目

另外更全面的面试题集我也在整理中，先给个预告图：

下面进入正题：

1.介绍一下js的数据类型有哪些，值是如何存储的

具体可看我之前的文章：「前端料包」可能是最透彻的JavaScript数据类型详解

JavaScript一共有8种数据类型，其中有7种基本数据类型：Undefined、Null、Boolean、Number、String、Symbol（es6新增，表示独一无二的值）和BigInt（es10新增）；

1种引用数据类型——Object（Object本质上是由一组无序的名值对组成的）。里面包含 function、Array、Date等。JavaScript不支持任何创建自定义类型的机制，而所有值最终都将是上述 8 种数据类型之一。

原始数据类型：直接存储在**栈**（stack）中，占据空间小、大小固定，属于被频繁使用数据，所以放入栈中存储。

引用数据类型：同时存储在**栈**（stack）和**堆**（heap）中，占据空间大、大小不固定。引用数据类型在栈中存储了指针，该指针指向堆中该实体的起始地址。当解释器寻找引用值时，会首先检索其在栈中的地址，取得地址后从堆中获得实体。

2. && 、 ||和!! 运算符分别能做什么

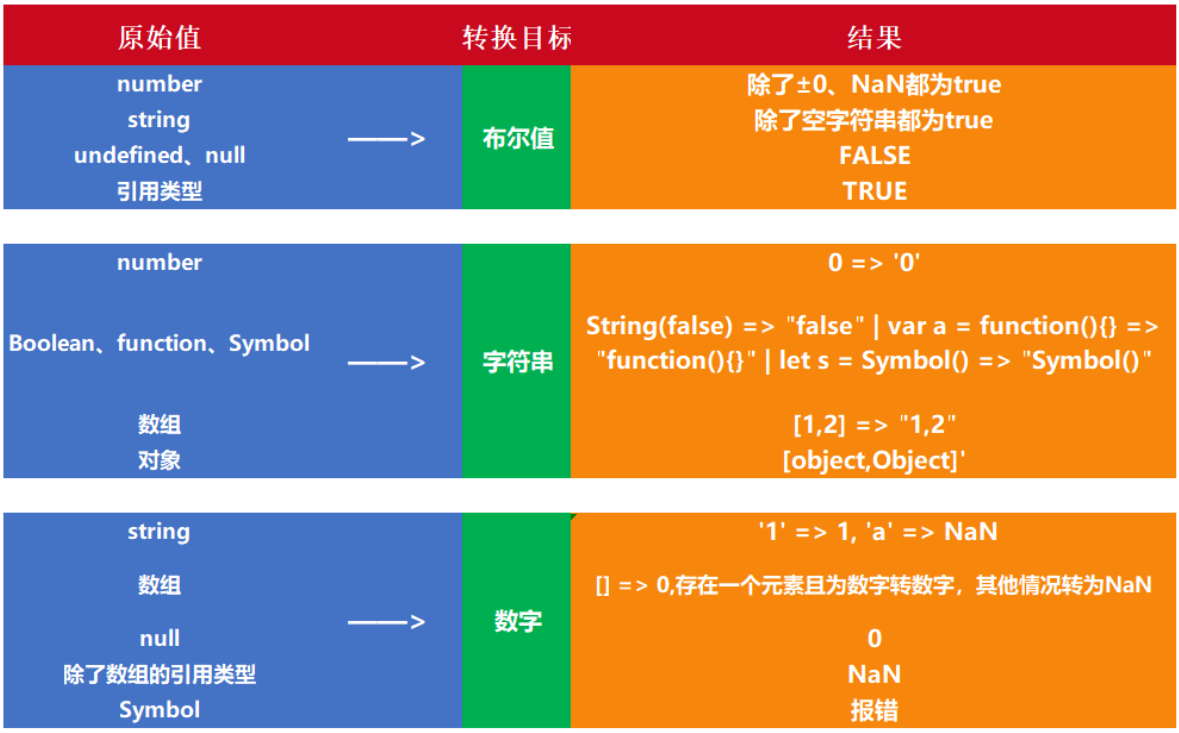
- `&&` 叫逻辑与，在其操作数中找到第一个虚值表达式并返回它，如果没有找到任何虚值表达式，则返回最后一个真值表达式。它采用短路来防止不必要的工作。
- `||` 叫逻辑或，在其操作数中找到第一个真值表达式并返回它。这也使用了短路来防止不必要的工作。在支持 ES6 默认函数参数之前，它用于初始化函数中的默认参数值。
- `!!` 运算符可以将右侧的值强制转换为布尔值，这也是将值转换为布尔值的一种简单方法。

3. js的数据类型的转换

在 JS 中类型转换只有三种情况，分别是：

- 转换为布尔值（调用Boolean()方法）
- 转换为数字（调用Number()、parseInt()和parseFloat()方法）
- 转换为字符串（调用.toString()或者String()方法）

> null和underfined没有.toString方法



此外还有一些操作符会存在隐式转换，此处不做展开，可自行百度00

4. JS中数据类型的判断（ typeof，instanceof，constructor，Object.prototype.toString.call()

（1）typeof

typeof 对于原始类型来说，除了 null 都可以显示正确的类型

```
console.log(typeof 2);               // number
console.log(typeof true);            // boolean
console.log(typeof 'str');           // string
console.log(typeof []);              // object     []数组的数据类型在 typeof 中被解释为 object
console.log(typeof function(){});    // function
console.log(typeof {});              // object
console.log(typeof undefined);       // undefined
console.log(typeof null);            // object     null 的数据类型被 typeof 解释为 object
复制代码
```

typeof 对于对象来说，除了函数都会显示 object，所以说 typeof 并不能准确判断变量到底是什么类型，所以想判断一个对象的正确类型，这时候可以考虑使用 instanceof

（2）instanceof

instanceof 可以正确的判断对象的类型，因为内部机制是通过判断对象的原型链中是不是能找到类型的 prototype。

```
console.log(2 instanceof Number);                    // false
console.log(true instanceof Boolean);                // false
console.log('str' instanceof String);                // false
console.log([] instanceof Array);                    // true
console.log(function(){} instanceof Function);       // true
console.log({} instanceof Object);                   // true
// console.log(undefined instanceof Undefined);
// console.log(null instanceof Null);
复制代码
```

可以看出直接的字面量值判断数据类型，instanceof可以精准判断引用数据类型（Array，Function，Object），而基本数据类型不能被instanceof精准判断。

我们来看一下 instanceof 在MDN中的解释：instanceof 运算符用来测试一个对象在其原型链中是否存在一个构造函数的 prototype 属性。其意思就是判断对象是否是某一数据类型（如Array）的实例，请重点关注一下是判断一个对象是否是数据类型的实例。在这里字面量值，2， true ，'str'不是实例，所以判断值为false。

（3）constructor

```
console.log((2).constructor === Number); // true
console.log((true).constructor === Boolean); // true
console.log(('str').constructor === String); // true
console.log(([]).constructor === Array); // true
console.log((function() {}).constructor === Function); // true
console.log(({}).constructor === Object); // true
复制代码
```

> 这里有一个坑，如果我创建一个对象，更改它的原型，constructor就会变得不可靠了

```
function Fn(){};

Fn.prototype=new Array();

var f=new Fn();

console.log(f.constructor===Fn);    // false
console.log(f.constructor===Array); // true
复制代码
```

（4）Object.prototype.toString.call() 使用 Object 对象的原型方法 toString ， 使用 call 进行狸猫换太子，借用Object的 toString 方法

```
var a = Object.prototype.toString;

console.log(a.call(2));
console.log(a.call(true));
console.log(a.call('str'));
console.log(a.call([]));
console.log(a.call(function(){}));
console.log(a.call({}));
console.log(a.call(undefined));
console.log(a.call(null));
复制代码
```

5. 介绍 js 有哪些内置对象?

js 中的内置对象主要指的是在程序执行前存在全局作用域里的由 js 定义的一些全局值属性、函数和用来实例化其他对象的构造函 数对象。一般我们经常用到的如全局变量值 NaN、undefined，全局函数如 parseInt()、parseFloat() 用来实例化对象的构 造函数如 Date、Object 等，还有提供数学计算的单体内置对象如 Math 对象。

涉及知识点:

全局的对象（ global objects ）或称标准内置对象，不要和 "全局对象（global object）" 混淆。这里说的全局的对象是说在
全局作用域里的对象。全局作用域中的其他对象可以由用户的脚本创建或由宿主程序提供。

标准内置对象的分类

（1）值属性，这些全局属性返回一个简单值，这些值没有自己的属性和方法。

例如 Infinity、NaN、undefined、null 字面量

（2）函数属性，全局函数可以直接调用，不需要在调用时指定所属对象，执行结束后会将结果直接返回给调用者。

例如 eval()、parseFloat()、parseInt() 等

（3）基本对象，基本对象是定义或使用其他对象的基础。基本对象包括一般对象、函数对象和错误对象。

例如 Object、Function、Boolean、Symbol、Error 等

（4）数字和日期对象，用来表示数字、日期和执行数学计算的对象。

例如 Number、Math、Date

（5）字符串，用来表示和操作字符串的对象。

例如 String、RegExp

（6）可索引的集合对象，这些对象表示按照索引值来排序的数据集合，包括数组和类型数组，以及类数组结构的对象。例如 Array

（7）使用键的集合对象，这些集合对象在存储数据时会使用到键，支持按照插入顺序来迭代元素。

例如 Map、Set、WeakMap、WeakSet

（8）矢量集合，SIMD 矢量集合中的数据会被组织为一个数据序列。

例如 SIMD 等

（9）结构化数据，这些对象用来表示和操作结构化的缓冲区数据，或使用 JSON 编码的数据。

例如 JSON 等

（10）控制抽象对象

例如 Promise、Generator 等

（11）反射

```
例如 Reflect、Proxy

（12）国际化，为了支持多语言处理而加入 ECMAScript 的对象。

例如 Intl、Intl.Collator 等

（13）WebAssembly

（14）其他

例如 arguments

复制代码
```

详细资料可以参考：《标准内置对象的分类》

《JS 所有内置对象属性和方法汇总》

6. undefined 与 undeclared 的区别?

已在作用域中声明但还没有赋值的变量，是 undefined。相反，还没有在作用域中声明过的变量，是 undeclared 的。

对于 undeclared 变量的引用，浏览器会报引用错误，如 ReferenceError: b is not defined 。但是我们可以使用 typ eof 的安全防范机制来避免报错，因为对于 undeclared（或者 not defined ）变量，typeof 会返回 "undefined"。

7. null 和 undefined 的区别?

首先 Undefined 和 Null 都是基本数据类型，这两个基本数据类型分别都只有一个值，就是 undefined 和 null。

undefined 代表的含义是未定义， null 代表的含义是空对象（其实不是真的对象，请看下面的**注意**！）。一般变量声明了但还没有定义的时候会返回 undefined，null 主要用于赋值给一些可能会返回对象的变量，作为初始化。

> 其实 null 不是对象，虽然 typeof null 会输出 object，但是这只是 JS 存在的一个悠久 Bug。在 JS 的最初版本中使用的是 32 位系统，为了性能考虑使用低位存储变量的类型信息，000 开头代表是对象，然而 null 表示为全零，所以将它错误的判断为 object 。虽然现在的内部类型判断代码已经改变了，但是对于这个 Bug 却是一直流传下来。

undefined 在 js 中不是一个保留字，这意味着我们可以使用 undefined 来作为一个变量名，这样的做法是非常危险的，它 会影响我们对 undefined 值的判断。但是我们可以通过一些方法获得安全的 undefined 值，比如说 void 0。

当我们对两种类型使用 typeof 进行判断的时候，Null 类型化会返回 “object”，这是一个历史遗留的问题。当我们使用双等 号对两种类型的值进行比较时会返回 true，使用三个等号时会返回 false。

详细资料可以参考：

《JavaScript 深入理解之 undefined 与 null》

8. {}和[]的valueOf和toString的结果是什么?

```
{} 的 valueOf 结果为 {} ，toString 的结果为 "[object Object]"

[] 的 valueOf 结果为 [] ，toString 的结果为 ""
复制代码
```

9. Javascript 的作用域和作用域链

作用域： 作用域是定义变量的区域，它有一套访问变量的规则，这套规则来管理浏览器引擎如何在当前作用域以及嵌套的作用域中根据变量（标识符）进行变量查找。

作用域链： 作用域链的作用是保证对执行环境有权访问的所有变量和函数的有序访问，通过作用域链，我们可以访问到外层环境的变量和 函数。

作用域链的本质上是一个指向变量对象的指针列表。变量对象是一个包含了执行环境中所有变量和函数的对象。作用域链的前 端始终都是当前执行上下文的变量对象。全局执行上下文的变量对象（也就是全局对象）始终是作用域链的最后一个对象。

当我们查找一个变量时，如果当前执行环境中没有找到，我们可以沿着作用域链向后查找。

作用域链的创建过程跟执行上下文的建立有关....

详细资料可以参考： 《JavaScript 深入理解之作用域链》

也可以看看我的文章：「前端料包」深究JavaScript作用域（链）知识点和闭包

10. javascript 创建对象的几种方式?

我们一般使用字面量的形式直接创建对象，但是这种创建方式对于创建大量相似对象的时候，会产生大量的重复代码。但 js
和一般的面向对象的语言不同，在 ES6 之前它没有类的概念。但是我们可以使用函数来进行模拟，从而产生出可复用的对象
创建方式，我了解到的方式有这么几种：

（1）第一种是工厂模式，工厂模式的主要工作原理是用函数来封装创建对象的细节，从而通过调用函数来达到复用的目的。但是它有一个很大的问题就是创建出来的对象无法和某个类型联系起来，它只是简单的封装了复用代码，而没有建立起对象和类型间的关系。

（2）第二种是构造函数模式。js 中每一个函数都可以作为构造函数，只要一个函数是通过 new 来调用的，那么我们就可以把它称为构造函数。执行构造函数首先会创建一个对象，然后将对象的原型指向构造函数的 prototype 属性，然后将执行上下文中的 this 指向这个对象，最后再执行整个函数，如果返回值不是对象，则返回新建的对象。因为 this 的值指向了新建的对象，因此我们可以使用 this 给对象赋值。构造函数模式相对于工厂模式的优点是，所创建的对象和构造函数建立起了联系，因此我们可以通过原型来识别对象的类型。但是构造函数存在一个缺点就是，造成了不必要的函数对象的创建，因为在 js 中函数也是一个对象，因此如果对象属性中如果包含函数的话，那么每次我们都会新建一个函数对象，浪费了不必要的内存空间，因为函数是所有的实例都可以通用的。

（3）第三种模式是原型模式，因为每一个函数都有一个 prototype 属性，这个属性是一个对象，它包含了通过构造函数创建的所有实例都能共享的属性和方法。因此我们可以使用原型对象来添加公用属性和方法，从而实现代码的复用。这种方式相对于构造函数模式来说，解决了函数对象的复用问题。但是这种模式也存在一些问题，一个是没有办法通过传入参数来初始化值，另一个是如果存在一个引用类型如 Array 这样的值，那么所有的实例将共享一个对象，一个实例对引用类型值的改变会影响所有的实例。

（4）第四种模式是组合使用构造函数模式和原型模式，这是创建自定义类型的最常见方式。因为构造函数模式和原型模式分开使用都存在一些问题，因此我们可以组合使用这两种模式，通过构造函数来初始化对象的属性，通过原型对象来实现函数方法的复用。这种方法很好的解决了两种模式单独使用时的缺点，但是有一点不足的就是，因为使用了两种不同的模式，所以对于代码的封装性不够好。

（5）第五种模式是动态原型模式，这一种模式将原型方法赋值的创建过程移动到了构造函数的内部，通过对属性是否存在的判断，可以实现仅在第一次调用函数时对原型对象赋值一次的效果。这一种方式很好地对上面的混合模式进行了封装。

（6）第六种模式是寄生构造函数模式，这一种模式和工厂模式的实现基本相同，我对这个模式的理解是，它主要是基于一个已有的类型，在实例化时对实例化的对象进行扩展。这样既不用修改原来的构造函数，也达到了扩展对象的目的。它的一个缺点和工厂模式一样，无法实现对象的识别。

嗯我目前了解到的就是这么几种方式。
复制代码

详细资料可以参考： 《JavaScript 深入理解之对象创建》

11. JavaScript 继承的几种实现方式?

我了解的 `js` 中实现继承的几种方式有：

（1）第一种是以原型链的方式来实现继承，但是这种实现方式存在的缺点是，在包含有引用类型的数据时，会被所有的实例对象所共享，容易造成修改的混乱。还有就是在创建子类型的时候不能向超类型传递参数。

（2）第二种方式是使用借用构造函数的方式，这种方式是通过在子类型的函数中调用超类型的构造函数来实现的，这一种方法解决了不能向超类型传递参数的缺点，但是它存在的一个问题就是无法实现函数方法的复用，并且超类型原型定义的方法子类型也没有办法访问到。

（3）第三种方式是组合继承，组合继承是将原型链和借用构造函数组合起来使用的一种方式。通过借用构造函数的方式来实现类型的属性的继承，通过将子类型的原型设置为超类型的实例来实现方法的继承。这种方式解决了上面的两种模式单独使用时的问题，但是由于我们是以超类型的实例来作为子类型的原型，所以调用了两次超类的构造函数，造成了子类型的原型中多了很多不必要的属性。

（4）第四种方式是原型式继承，原型式继承的主要思路就是基于已有的对象来创建新的对象，实现的原理是，向函数中传入一个对象，然后返回一个以这个对象为原型的对象。这种继承的思路主要不是为了实现创造一种新的类型，只是对某个对象实现一种简单继承，`ES5` 中定义的 `Object.create()` 方法就是原型式继承的实现。缺点与原型链方式相同。

（5）第五种方式是寄生式继承，寄生式继承的思路是创建一个用于封装继承过程的函数，通过传入一个对象，然后复制一个对象的副本，然后对象进行扩展，最后返回这个对象。这个扩展的过程就可以理解是一种继承。这种继承的优点就是对一个简单对象实现继承，如果这个对象不是我们的自定义类型时。缺点是没有办法实现函数的复用。

（6）第六种方式是寄生式组合继承，组合继承的缺点就是使用超类型的实例做为子类型的原型，导致添加了不必要的原型属性。寄生式组合继承的方式是使用超类型的原型的副本来作为子类型的原型，这样就避免了创建不必要的属性。
复制代码

详细资料可以参考： 《JavaScript 深入理解之继承》

12. 寄生式组合继承的实现?

```
function Person(name) {
  this.name = name;
}

Person.prototype.sayName = function() {
  console.log("My name is " + this.name + ".");
};
```

```
function Student(name, grade) {
  Person.call(this, name);
  this.grade = grade;
}

Student.prototype = Object.create(Person.prototype);
Student.prototype.constructor = Student;

Student.prototype.sayMyGrade = function() {
  console.log("My grade is " + this.grade + ".");

};
```
复制代码

13. 谈谈你对this、call、apply和bind的理解

详情可看我之前的文章：「前端料包」一文彻底搞懂JavaScript中的this、call、apply和bind

1. 在浏览器里，在全局范围内this 指向window对象；
2. 在函数中，this永远指向最后调用他的那个对象；
3. 构造函数中，this指向new出来的那个新的对象；
4. call、apply、bind中的this被强绑定在指定的那个对象上；
5. 箭头函数中this比较特殊,箭头函数this为父作用域的this，不是调用时的this.要知道前四种方式,都是调用时确定,也就是动态的,而箭头函数的this指向是静态的,声明的时候就确定了下来；
6. apply、call、bind都是js给函数内置的一些API，调用他们可以为函数指定this的执行,同时也可以传参。

14. JavaScript 原型，原型链？ 有什么特点?

在 js 中我们是使用构造函数来新建一个对象的，每一个构造函数的内部都有一个 prototype 属性值，这个属性值是一个对 象，这个对象包含了可以由该构造函数的所有实例共享的属性和方法。当我们使用构造函数新建一个对象后，在这个对象的内部 将包含一个指针，这个指针指向构造函数的 prototype 属性对应的值，在 ES5 中这个指针被称为对象的原型。一般来说我们 是不应该能够获取到这个值的，但是现在浏览器中都实现了 **proto** 属性来让我们访问这个属性，但是我们最好不要使用这 个属性，因为它不是规范中规定的。ES5 中新增了一个 Object.getPrototypeOf() 方法，我们可以通过这个方法来获取对 象的原型。

当我们访问一个对象的属性时，如果这个对象内部不存在这个属性，那么它就会去它的原型对象里找这个属性，这个原型对象又 会有自己的原型，于是就这样一直找下去，也就是原型链的概念。原型链的尽头一般来说都是 Object.prototype 所以这就 是我们新建的对象为什么能够使用 toString() 等方法的原因。

特点：

JavaScript 对象是通过引用来传递的，我们创建的每个新对象实体中并没有一份属于自己的原型副本。当我们修改原型时，与 之相关的对象也会继承这一改变。

参考文章：

《JavaScript 深入理解之原型与原型链》

也可以看看我写的：「前端料包」深入理解JavaScript原型和原型链

15. js 获取原型的方法?

- p.**proto**
- p.constructor.prototype
- Object.getPrototypeOf(p)

16. 什么是闭包，为什么要用它?

闭包是指有权访问另一个函数作用域内变量的函数，创建闭包的最常见的方式就是在一个函数内创建另一个函数，创建的函数可以 访问到当前函数的局部变量。

闭包有两个常用的用途。

- 闭包的第一个用途是使我们在函数外部能够访问到函数内部的变量。通过使用闭包，我们可以通过在外部调用闭包函数，从而在外部访问到函数内部的变量，可以使用这种方法来创建私有变量。
- 函数的另一个用途是使已经运行结束的函数上下文中的变量对象继续留在内存中，因为闭包函数保留了这个变量对象的引用，所以这个变量对象不会被回收。

```
function a(){
    var n = 0;
    function add(){
       n++;
       console.log(n);
    }
    return add;
}
var a1 = a(); //注意，函数名只是一个标识（指向函数的指针），而（）才是执行函数；
a1();    //1
a1();    //2  第二次调用n变量还在内存中


复制代码
```

其实闭包的本质就是作用域链的一个特殊的应用，只要了解了作用域链的创建过程，就能够理解闭包的实现原理。

17. 什么是 DOM 和 BOM?

DOM 指的是文档对象模型，它指的是把文档当做一个对象来对待，这个对象主要定义了处理网页内容的方法和接口。

BOM 指的是浏览器对象模型，它指的是把浏览器当做一个对象来对待，这个对象主要定义了与浏览器进行交互的法和接口。BOM 的核心是 window，而 window 对象具有双重角色，它既是通过 js 访问浏览器窗口的一个接口，又是一个 Global（全局） 对象。这意味着在网页中定义的任何对象，变量和函数，都作为全局对象的一个属性或者方法存在。window 对象含有 locati on 对象、navigator 对象、screen 对象等子对象，并且 DOM 的最根本的对象 document 对象也是 BOM 的 window 对 象的子对象。

相关资料:

《DOM, DOCUMENT, BOM, WINDOW 有什么区别?》

《Window 对象》

《DOM 与 BOM 分别是什么，有何关联？》

《JavaScript 学习总结（三）BOM 和 DOM 详解》

18. 三种事件模型是什么?

事件 是用户操作网页时发生的交互动作或者网页本身的一些操作，现代浏览器一共有三种事件模型。

1. **DOM0级模型：** ，这种模型不会传播，所以没有事件流的概念，但是现在有的浏览器支持以冒泡的方式实现，它可以在网页中直接定义监听函数，也可以通过 js属性来指定监听函数。这种方式是所有浏览器都兼容的。
2. **IE 事件模型：** 在该事件模型中，一次事件共有两个过程，事件处理阶段，和事件冒泡阶段。事件处理阶段会首先执行目标元素绑定的监听事件。然后是事件冒泡阶段，冒泡指的是事件从目标元素冒泡到 document，依次检查经过的节点是否绑定了事件监听函数，如果有则执行。这种模型通过 attachEvent 来添加监听函数，可以添加多个监听函数，会按顺序依次执行。
3. **DOM2 级事件模型：** 在该事件模型中，一次事件共有三个过程，第一个过程是事件捕获阶段。捕获指的是事件从 document 一直向下传播到目标元素，依次检查经过的节点是否绑定了事件监听函数，如果有则执行。后面两个阶段和 IE 事件模型的两个阶段相同。这种事件模型，事件绑定的函数是 addEventListener，其中第三个参数可以指定事件是否在捕获阶段执行。

相关资料：

《一个 DOM 元素绑定多个事件时，先执行冒泡还是捕获》

19. 事件委托是什么?

事件委托 本质上是利用了浏览器事件冒泡的机制。因为事件在冒泡过程中会上传到父节点，并且父节点可以通过事件对象获取到 目标节点，因此可以把子节点的监听函数定义在父节点上，由父节点的监听函数统一处理多个子元素的事件，这种方式称为事件代理。

使用事件代理我们可以不必要为每一个子元素都绑定一个监听事件，这样减少了内存上的消耗。并且使用事件代理我们还可以实现事件的动态绑定，比如说新增了一个子节点，我们并不需要单独地为它添加一个监听事件，它所发生的事件会交给父元素中的监听函数来处理。

相关资料：

《JavaScript 事件委托详解》

20. 什么是事件传播?

当**事件**发生在DOM元素上时，该事件并不完全发生在那个元素上。在“当事件发生在DOM元素上时，该事件并不完全发生在那个元素上。

事件传播有三个阶段：

1. 捕获阶段–事件从 window 开始，然后向下到每个元素，直到到达目标元素事件或event.target。
2. 目标阶段–事件已达到目标元素。
3. 冒泡阶段–事件从目标元素冒泡，然后上升到每个元素，直到到达 window。

21. 什么是事件捕获?

当事件发生在 DOM 元素上时，该事件并不完全发生在那个元素上。在捕获阶段，事件从window开始，一直到触发事件的元素。`window----> document----> html----> body ---->目标元素`

假设有如下的 HTML 结构：

```
<div class="grandparent">
  <div class="parent">
    <div class="child">1</div>
  </div>
</div>
```
复制代码

对应的 JS 代码:

```
function addEvent(el, event, callback, isCapture = false) {
  if (!el || !event || !callback || typeof callback !== 'function') return;
  if (typeof el === 'string') {
    el = document.querySelector(el);
  };
  el.addEventListener(event, callback, isCapture);
}

addEvent(document, 'DOMContentLoaded', () => {
  const child = document.querySelector('.child');
  const parent = document.querySelector('.parent');
  const grandparent = document.querySelector('.grandparent');

  addEvent(child, 'click', function (e) {
    console.log('child');
  });

  addEvent(parent, 'click', function (e) {
    console.log('parent');
  });

  addEvent(grandparent, 'click', function (e) {
    console.log('grandparent');
  });

  addEvent(document, 'click', function (e) {
    console.log('document');
  });

  addEvent('html', 'click', function (e) {
    console.log('html');
  })

  addEvent(window, 'click', function (e) {
    console.log('window');
  })

});
```
复制代码

`addEventListener`方法具有第三个可选参数`useCapture`，其默认值为`false`，事件将在冒泡阶段中发生，如果为true，则事件将在捕获阶段中发生。如果单击`child`元素，它将分别在控制台上打印`window`，`document`，`html`，`grandparent`和`parent`，这就是**事件捕获。**

22. 什么是事件冒泡?

事件冒泡刚好与事件捕获相反，`当前元素--->body ----> html--->document --->window`。当事件发生在DOM元素上时，该事件并不完全发生在那个元素上。在冒泡阶段，事件冒泡，或者事件发生在它的父代，祖父母，祖父母的父代，直到到达window为止。

假设有如下的 HTML 结构：

```
<div class="grandparent">
  <div class="parent">
    <div class="child">1</div>
  </div>
</div>
复制代码
```

对应的JS代码：

```
function addEvent(el, event, callback, isCapture = false) {
  if (!el || !event || !callback || typeof callback !== 'function') return;
  if (typeof el === 'string') {
    el = document.querySelector(el);
  };
  el.addEventListener(event, callback, isCapture);
}

addEvent(document, 'DOMContentLoaded', () => {
  const child = document.querySelector('.child');
  const parent = document.querySelector('.parent');
  const grandparent = document.querySelector('.grandparent');

  addEvent(child, 'click', function (e) {
    console.log('child');
  });

  addEvent(parent, 'click', function (e) {
    console.log('parent');
  });

  addEvent(grandparent, 'click', function (e) {
    console.log('grandparent');
  });

  addEvent(document, 'click', function (e) {
    console.log('document');
  });

  addEvent('html', 'click', function (e) {
    console.log('html');
  })

  addEvent(window, 'click', function (e) {
    console.log('window');
  })

});
复制代码
```

`addEventListener`方法具有第三个可选参数`useCapture`，其默认值为`false`，事件将在冒泡阶段中发生，如果为true，则事件将在捕获阶段中发生。如果单击`child`元素，它将分别在控制台上打印`child`，`parent`，`grandparent`，`html`，`document`和`window`，这就是**事件冒泡。**

23. DOM 操作——怎样添加、移除、移动、复制、创建和查找节点?

（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
复制代码
```

（2）添加、移除、替换、插入

```
appendChild(node)
removeChild(node)
replaceChild(new,old)
insertBefore(new,old)
复制代码
```

（3）查找

```
getElementById();
getElementsByName();
getElementsByTagName();
getElementsByClassName();
querySelector();
querySelectorAll();
复制代码
```

（4）属性操作

```
getAttribute(key);
setAttribute(key, value);
hasAttribute(key);
removeAttribute(key);
复制代码
```

相关资料:

《DOM 概述》

《原生 JavaScript 的 DOM 操作汇总》

《原生 JS 中 DOM 节点相关 API 合集》

24. js数组和对象有哪些原生方法,列举一下

数组方法	说明
push()	向数组的末尾添加一个或多个元素，并返回新的长度，也就是添加元素后的数组长度。
shift()	用于把数组的第一个元素从其中删除，并返回第一个元素的值
unshift()	向数组的开头添加一个或更多元素，并返回新的长度。
pop()	用于删除并返回数组的最后一个元素。
splice()	用于插入、删除或替换数组的元素。
concat()	方法用于连接两个或多个数组。
join()	用于把数组中的所有元素放入一个字符串。元素是通过指定的分隔符进行分隔的。
toString()	把数组转换为字符串，并返回结果。
reverse()	用于颠倒数组中元素的顺序。
slice()	方法可从已有的数组中返回选定的元素。
sort()	方法用于对数组的元素进行排序（从小到大）。
indexOf()	返回获取项在数组中的索引
lastIndexOf()	返回获取项在数组中出现的最后一次索引
forEach()	循环遍历数组 参数是一个匿名函数 默认返回为undefined
map()	循环遍历数组 参数是一个匿名函数

对象方法	说明
charAt()	返回在指定位置的字符。
charCodeAt()	返回在指定的位置的字符的 Unicode 编码。
concat()	连接字符串。
indexOf()	检索字符串。
match()	找到一个或多个正则表达式的匹配。
replace()	替换与正则表达式匹配的子串。
search()	检索与正则表达式相匹配的值。
slice()	提取字符串的片断，并在新的字符串中返回被提取的部分。
split()	把字符串分割为字符串数组。
toLocaleLowerCase()	把字符串转换为小写。
toLocaleUpperCase()	把字符串转换为大写。
toLowerCase()	把字符串转换为小写。
toUpperCase()	把字符串转换为大写。
substr()	从起始索引号提取字符串中指定数目的字符。
substring()	提取字符串中两个指定的索引号之间的字符。

25. 常用的正则表达式（仅做收集，涉及不深）

```
//（1）匹配 16 进制颜色值
var color = /#([0-9a-fA-F]{6}|[0-9a-fA-F]{3})/g;

//（2）匹配日期，如 yyyy-mm-dd 格式
var date = /^[0-9]{4}-(0[1-9]|1[0-2])-(0[1-9]|[12][0-9]|3[01])$/;

//（3）匹配 qq 号
var qq = /^[1-9][0-9]{4,10}$/g;
```

```
//（4）手机号码正则
var phone = /^1[34578]\d{9}$/g;

//（5）用户名正则
var username = /^[a-zA-Z\$][a-zA-Z0-9_\$]{4,16}$/;

//（6）Email正则
var email = /^([A-Za-z0-9_\-\.])+\@([A-Za-z0-9_\-\.])+\.([A-Za-z]{2,4})$/;

//（7）身份证号（18位）正则
var cP = /^[1-9]\d{5}(18|19|([23]\d))\d{2}((0[1-9])|(10|11|12))(([0-2][1-
9])|10|20|30|31)\d{3}[0-9Xx]$/;

//（8）URL正则
var urlP= /^((https?|ftp|file):\/\/)?([\da-z\.-]+)\.([a-z\.]{2,6})([\/\w
\.-]*)*\/?$/;

// (9)ipv4地址正则
var ipP = /^(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(?:25[0-5]|2[0-4]
[0-9]|[01]?[0-9][0-9]?)$/;

// (10)//车牌号正则
var cPattern = /^[京津沪渝冀豫云辽黑湘皖鲁新苏浙赣鄂桂甘晋蒙陕吉闽贵粤青藏川宁琼使领A-Z]{1}
[A-Z]{1}[A-Z0-9]{4}[A-Z0-9挂学警港澳]{1}$/;

// (11)强密码(必须包含大小写字母和数字的组合，不能使用特殊字符，长度在8-10之间)：var pwd =
/^(?=.\d)(?=.[a-z])(?=.[A-Z]).{8,10}$/
复制代码
```

26. Ajax 是什么? 如何创建一个 Ajax?

我对 ajax 的理解是，它是一种异步通信的方法，通过直接由 js 脚本向服务器发起 http 通信，然后根据服务器返回的数据，更新网页的相应部分，而不用刷新整个页面的一种方法。

创建步骤：

面试手写（原生）：

```
//1: 创建Ajax对象
var xhr = window.XMLHttpRequest?new XMLHttpRequest():new
ActiveXObject('Microsoft.XMLHTTP');// 兼容IE6及以下版本
//2: 配置 Ajax请求地址
xhr.open('get','index.xml',true);
//3: 发送请求
xhr.send(null); // 严谨写法
//4:监听请求，接受响应
xhr.onreadysatechange=function(){
     if(xhr.readySates==4&&xhr.status==200 || xhr.status==304 )
          console.log(xhr.responsetXML)
}
```

复制代码

jQuery写法

```
  $.ajax({
          type:'post',
          url:'',
          async:ture,//async 异步  sync  同步
          data:data,//针对post请求
          dataType:'jsonp',
          success:function (msg) {

          },
          error:function (error) {

          }
        })
```

复制代码

promise 封装实现：

```
// promise 封装实现：

function getJSON(url) {
  // 创建一个 promise 对象
  let promise = new Promise(function(resolve, reject) {
    let xhr = new XMLHttpRequest();

    // 新建一个 http 请求
    xhr.open("GET", url, true);

    // 设置状态的监听函数
    xhr.onreadystatechange = function() {
      if (this.readyState !== 4) return;

      // 当请求成功或失败时，改变 promise 的状态
      if (this.status === 200) {
        resolve(this.response);
      } else {
        reject(new Error(this.statusText));
      }
    };
```

```
    // 设置错误监听函数
    xhr.onerror = function() {
      reject(new Error(this.statusText));
    };

    // 设置响应的数据类型
    xhr.responseType = "json";

    // 设置请求头信息
    xhr.setRequestHeader("Accept", "application/json");

    // 发送 http 请求
    xhr.send(null);
  });

  return promise;
}
```
复制代码

27. js 延迟加载的方式有哪些?

js 的加载、解析和执行会阻塞页面的渲染过程，因此我们希望 js 脚本能够尽可能的延迟加载，提高页面的渲染速度。

我了解到的几种方式是：

1. 将 js 脚本放在文档的底部，来使 js 脚本尽可能的在最后来加载执行。
2. 给 js 脚本添加 defer属性，这个属性会让脚本的加载与文档的解析同步解析，然后在文档解析完成后再执行这个脚本文件，这样的话就能使页面的渲染不被阻塞。多个设置了 defer 属性的脚本按规范来说最后是顺序执行的，但是在一些浏览器中可能不是这样。
3. 给 js 脚本添加 async属性，这个属性会使脚本异步加载，不会阻塞页面的解析过程，但是当脚本加载完成后立即执行 js脚本，这个时候如果文档没有解析完成的话同样会阻塞。多个 async 属性的脚本的执行顺序是不可预测的，一般不会按照代码的顺序依次执行。
4. 动态创建 DOM 标签的方式，我们可以对文档的加载事件进行监听，当文档加载完成后再动态的创建 script 标签来引入 js 脚本。

相关资料：

《JS 延迟加载的几种方式》

《HTML 5 `async` 属性》

28. 谈谈你对模块化开发的理解?

我对模块的理解是，一个模块是实现一个特定功能的一组方法。在最开始的时候，js 只实现一些简单的功能，所以并没有模块的概念 ，但随着程序越来越复杂，代码的模块化开发变得越来越重要。

由于函数具有独立作用域的特点，最原始的写法是使用函数来作为模块，几个函数作为一个模块，但是这种方式容易造成全局变量的污 染，并且模块间没有联系。

后面提出了对象写法，通过将函数作为一个对象的方法来实现，这样解决了直接使用函数作为模块的一些缺点，但是这种办法会暴露所 有的所有的模块成员，外部代码可以修改内部属性的值。

现在最常用的是立即执行函数的写法，通过利用闭包来实现模块私有作用域的建立，同时不会对全局作用域造成污染。

相关资料： 《浅谈模块化开发》

《Javascript 模块化编程（一）：模块的写法》

《前端模块化：CommonJS，AMD，CMD，ES6》

《Module 的语法》

29. js 的几种模块规范?

js 中现在比较成熟的有四种模块加载方案：

- 第一种是 CommonJS 方案，它通过 require 来引入模块，通过 module.exports 定义模块的输出接口。这种模块加载方案是服务器端的解决方案，它是以同步的方式来引入模块的，因为在服务端文件都存储在本地磁盘，所以读取非常快，所以以同步的方式加载没有问题。但如果是在浏览器端，由于模块的加载是使用网络请求，因此使用异步加载的方式更加合适。
- 第二种是 AMD 方案，这种方案采用异步加载的方式来加载模块，模块的加载不影响后面语句的执行，所有依赖这个模块的语句都定义在一个回调函数里，等到加载完成后再执行回调函数。require.js 实现了 AMD 规范。
- 第三种是 CMD 方案，这种方案和 AMD 方案都是为了解决异步模块加载的问题，sea.js 实现了 CMD 规范。它和require.js的区别在于模块定义时对依赖的处理不同和对依赖模块的执行时机的处理不同。
- 第四种方案是 ES6 提出的方案，使用 import 和 export 的形式来导入导出模块。

30. AMD 和 CMD 规范的区别?

它们之间的主要区别有两个方面。

1. 第一个方面是在模块定义时对依赖的处理不同。AMD推崇依赖前置，在定义模块的时候就要声明其依赖的模块。而 CMD 推崇就近依赖，只有在用到某个模块的时候再去 require。
2. 第二个方面是对依赖模块的执行时机处理不同。首先 AMD 和 CMD 对于模块的加载方式都是异步加载，不过它们的区别在于 模块的执行时机，AMD 在依赖模块加载完成后就直接执行依赖模块，依赖模块的执行顺序和我们书写的顺序不一定一致。而 CMD 在依赖模块加载完成后并不执行，只是下载而已，等到所有的依赖模块都加载好后，进入回调函数逻辑，遇到 require 语句 的时候才执行对应的模块，这样模块的执行顺序就和我们书写的顺序保持一致了。

```
// CMD
define(function(require, exports, module) {
  var a = require("./a");
  a.doSomething();
  // 此处略去 100 行
  var b = require("./b"); // 依赖可以就近书写
  b.doSomething();
  // ...
});

// AMD 默认推荐
define(["./a", "./b"], function(a, b) {
  // 依赖必须一开始就写好
  a.doSomething();
  // 此处略去 100 行
  b.doSomething();
  // ...
});
复制代码
```

相关资料：

《前端模块化，AMD 与 CMD 的区别》

31. ES6 模块与 CommonJS 模块、AMD、CMD 的差异。

- 1.`CommonJS` 模块输出的是一个值的拷贝，ES6 模块输出的是值的引用。`CommonJS` 模块输出的是值的

，也就是说，一旦输出一个值，模块内部的变化就影响不到这个值。ES6 模块的运行机制与 CommonJS 不一样。JS 引擎对脚本静态分析的时候，遇到模块加载命令 import，就会生成一个只读引用。等到脚本真正执行时，再根据这个只读引用，到被加载的那个模块里面去取值。

- 2.`CommonJS` 模块是运行时加载，ES6 模块是编译时输出接口。`CommonJS` 模块就是对象，即在输入时是先加载整个模块，生成一个对象，然后再从这个对象上面读取方法，这种加载称为“运行时加载”。而 ES6 模块不是对象，它的对外接口只是一种静态定义，在代码静态解析阶段就会生成。

32. requireJS的核心原理是什么?

require.js 的核心原理是通过动态创建 script 脚本来异步引入模块，然后对每个脚本的 load 事件进行监听，如果每个脚本都加载完成了，再调用回调函数。```

详细资料可以参考： 《requireJS 的用法和原理分析》

《requireJS 的核心原理是什么？》

《requireJS 原理分析》

33. 谈谈JS的运行机制

1. js单线程

JavaScript语言的一大特点就是单线程，即同一时间只能做一件事情。

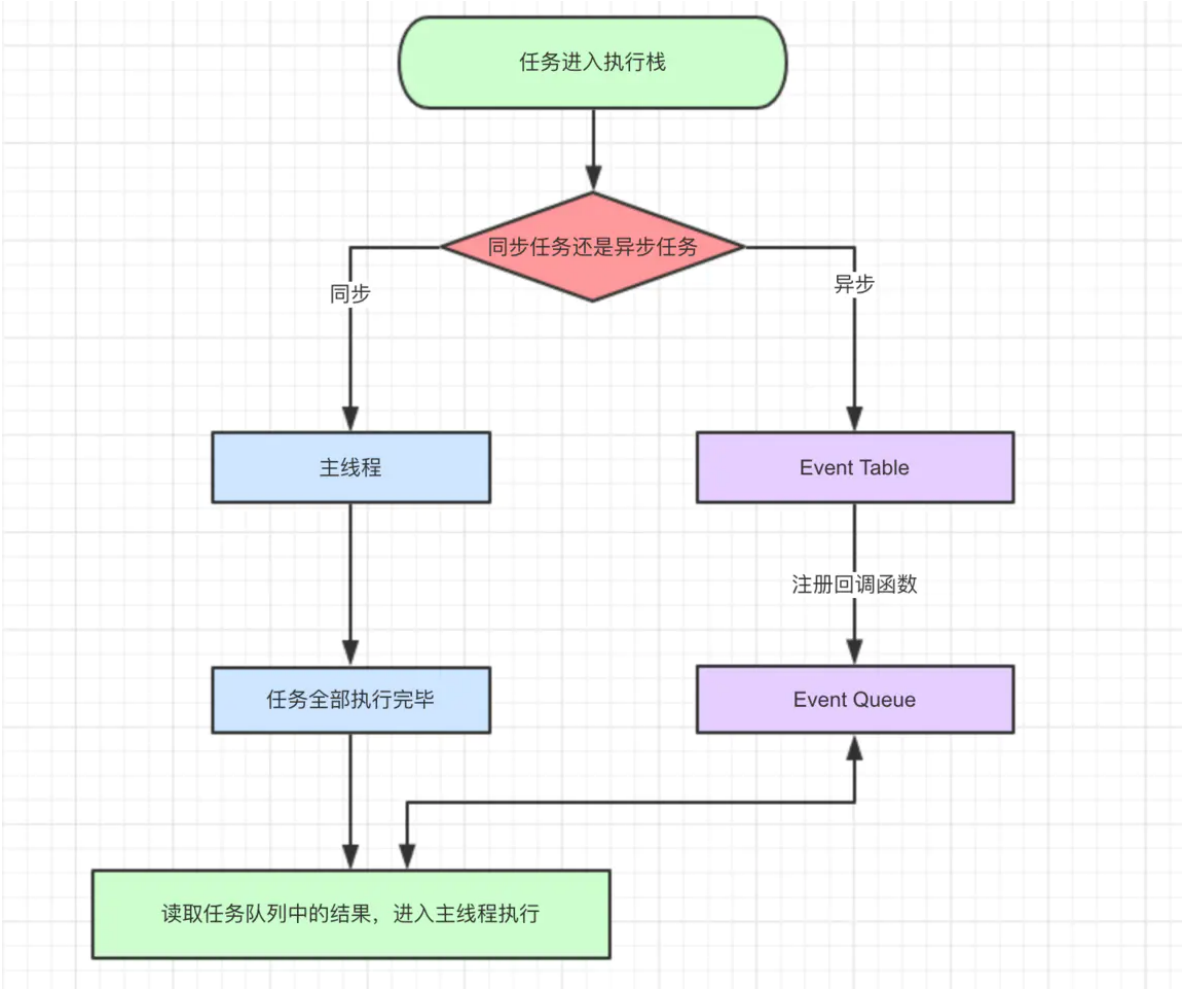
> JavaScript的单线程，与它的用途有关。作为浏览器脚本语言，JavaScript的主要用途是与用户互动，以及操作DOM。这决定了它只能是单线程，否则会带来很复杂的同步问题。比如，假定JavaScript同时有两个线程，一个线程在某个DOM节点上添加内容，另一个线程删除了这个节点，这时浏览器应该以哪个线程为准？ 所以，为了避免复杂性，从一诞生，JavaScript就是单线程，这已经成了这门语言的核心特征，将来也不会改变。

2. js事件循环

js代码执行过程中会有很多任务，这些任务总的分成两类：

- 同步任务
- 异步任务

当我们打开网站时，网页的渲染过程就是一大堆同步任务，比如页面骨架和页面元素的渲染。而像加载图片音乐之类占用资源大耗时久的任务，就是异步任务。，我们用导图来说明：



我们解释一下这张图：

- 同步和异步任务分别进入不同的执行"场所"，同步的进入主线程，异步的进入`Event Table`并注册函数。
- 当指定的事情完成时，`Event Table`会将这个函数移入`Event Queue`。
- 主线程内的任务执行完毕为空，会去`Event Queue`读取对应的函数，进入主线程执行。
- 上述过程会不断重复，也就是常说的`Event Loop`(事件循环)。

那主线程执行栈何时为空呢？js引擎存在`monitoring process`进程，会持续不断的检查主线程执行栈是否为空，一旦为空，就会去`Event Queue`那里检查是否有等待被调用的函数。

以上就是js运行的整体流程

需要注意的是除了同步任务和异步任务，任务还可以更加细分为macrotask(宏任务)和microtask(微任务)，js引擎会优先执行微任务

```
微任务包括了 promise 的回调、node 中的 process.nextTick 、对 Dom 变化监听的
MutationObserver。

宏任务包括了 script 脚本的执行、setTimeout ，setInterval ，setImmediate 一类的定时事
件，还有如 I/O 操作、UI 渲
染等。
复制代码
```

面试中该如何回答呢？ 下面是我个人推荐的回答：

1. 首先js 是单线程运行的，在代码执行的时候，通过将不同函数的执行上下文压入执行栈中来保证代码的有序执行。

2. 在执行同步代码的时候，如果遇到了异步事件，js 引擎并不会一直等待其返回结果，而是会将这个事件挂起，继续执行执行栈中的其他任务
3. 当同步事件执行完毕后，再将异步事件对应的回调加入到与当前执行栈中不同的另一个任务队列中等待执行。
4. 任务队列可以分为宏任务对列和微任务对列，当当前执行栈中的事件执行完毕后，js 引擎首先会判断微任务对列中是否有任务可以执行，如果有就将微任务队首的事件压入栈中执行。
5. 当微任务对列中的任务都执行完成后再去判断宏任务对列中的任务。

最后可以用下面一道题检测一下收获：

```
setTimeout(function() {
  console.log(1)
}, 0);
new Promise(function(resolve, reject) {
  console.log(2);
  resolve()
}).then(function() {
  console.log(3)
});
process.nextTick(function () {
  console.log(4)
})
console.log(5)

复制代码
```

第一轮：主线程开始执行，遇到setTimeout，将setTimeout的回调函数丢到宏任务队列中，在往下执行new Promise立即执行，输出2，then的回调函数丢到微任务队列中，再继续执行，遇到process.nextTick，同样将回调函数扔到为任务队列，再继续执行，输出5，当所有同步任务执行完成后看有没有可以执行的微任务，发现有then函数和nextTick两个微任务，先执行哪个呢？process.nextTick指定的异步任务总是发生在所有异步任务之前，因此先执行process.nextTick输出4然后执行then函数输出3，第一轮执行结束。 第二轮：从宏任务队列开始，发现setTimeout回调，输出1执行完毕，因此结果是25431

相关资料：

《浏览器事件循环机制（event loop）》

《详解 JavaScript 中的 Event Loop（事件循环）机制》

《什么是 Event Loop？》

《这一次，彻底弄懂 JavaScript 执行机制》

34. arguments 的对象是什么?

arguments对象是函数中传递的参数值的集合。它是一个类似数组的对象，因为它有一个length属性，我们可以使用数组索引表示法arguments[1]来访问单个值，但它没有数组中的内置方法，如：forEach、reduce、filter和map。

我们可以使用Array.prototype.slice将arguments对象转换成一个数组。

```
function one() {
  return Array.prototype.slice.call(arguments);
}
复制代码
```

注意:箭头函数中没有arguments对象。

```
function one() {
  return arguments;
}
const two = function () {
  return arguments;
}
const three = function three() {
  return arguments;
}

const four = () => arguments;

four(); // Throws an error  - arguments is not defined
```
复制代码

当我们调用函数four时，它会抛出一个`ReferenceError: arguments is not defined error`。使用rest语法，可以解决这个问题。

```
const four = (...args) => args;
```
复制代码

这会自动将所有参数值放入数组中。

35. 为什么在调用这个函数时，代码中的`b`会变成一个全局变量?

```
function myFunc() {
  let a = b = 0;
}

myFunc();
```
复制代码

原因是赋值运算符是从右到左的求值的。这意味着当多个赋值运算符出现在一个表达式中时，它们是从右向左求值的。所以上面代码变成了这样：

```
function myFunc() {
  let a = (b = 0);
}

myFunc();
```
复制代码

首先，表达式b = 0求值，在本例中b没有声明。因此，JS引擎在这个函数外创建了一个全局变量b，之后表达式b = 0的返回值为0，并赋给新的局部变量a。

我们可以通过在赋值之前先声明变量来解决这个问题。

```
function myFunc() {
  let a,b;
  a = b = 0;
}
myFunc();
```
复制代码

36. 简单介绍一下 V8 引擎的垃圾回收机制

v8 的垃圾回收机制基于分代回收机制，这个机制又基于世代假说，这个假说有两个特点，一是新生的对象容易早死，另一个是不死的对象会活得更久。基于这个假说，v8 引擎将内存分为了新生代和老生代。

新创建的对象或者只经历过一次的垃圾回收的对象被称为新生代。经历过多次垃圾回收的对象被称为老生代。

新生代被分为 From 和 To 两个空间，To 一般是闲置的。当 From 空间满了的时候会执行 Scavenge 算法进行垃圾回收。当我们执行垃圾回收算法的时候应用逻辑将会停止，等垃圾回收结束后再继续执行。这个算法分为三步：

（1）首先检查 From 空间的存活对象，如果对象存活则判断对象是否满足晋升到老生代的条件，如果满足条件则晋升到老生代。如果不满足条件则移动 To 空间。

（2）如果对象不存活，则释放对象的空间。

（3）最后将 From 空间和 To 空间角色进行交换。

新生代对象晋升到老生代有两个条件：

（1）第一个是判断是对象否已经经过一次 Scavenge 回收。若经历过，则将对象从 From 空间复制到老生代中；若没有经历，则复制到 To 空间。

（2）第二个是 To 空间的内存使用占比是否超过限制。当对象从 From 空间复制到 To 空间时，若 To 空间使用超过 25%，则对象直接晋升到老生代中。设置 25% 的原因主要是因为算法结束后，两个空间结束后会交换位置，如果 To 空间的内存太小，会影响后续的内存分配。

老生代采用了标记清除法和标记压缩法。标记清除法首先会对内存中存活的对象进行标记，标记结束后清除掉那些没有标记的对象。由于标记清除后会造成很多的内存碎片，不便于后面的内存分配。所以了解决内存碎片的问题引入了标记压缩法。

由于在进行垃圾回收的时候会暂停应用的逻辑，对于新生代方法由于内存小，每次停顿的时间不会太长，但对于老生代来说每次垃圾回收的时间长，停顿会造成很大的影响。 为了解决这个问题 V8 引入了增量标记的方法，将一次停顿进行的过程分为了多步，每次执行完一小步就让运行逻辑执行一会，就这样交替运行。
复制代码

相关资料：

《深入理解 V8 的垃圾回收原理》

《JavaScript 中的垃圾回收》

37. 哪些操作会造成内存泄漏?

- 1.意外的全局变量
- 2.被遗忘的计时器或回调函数
- 3.脱离 DOM 的引用
- 4.闭包

- 第一种情况是我们由于使用未声明的变量，而意外的创建了一个全局变量，而使这个变量一直留在内存中无法被回收。
- 第二种情况是我们设置了`setInterval`定时器，而忘记取消它，如果循环函数有对外部变量的引用的话，那么这个变量会被一直留在内存中，而无法被回收。
- 第三种情况是我们获取一个DOM元素的引用，而后面这个元素被删除，由于我们一直保留了对这个元素的引用，所以它也无法被回收。
- 第四种情况是不合理的使用闭包，从而导致某些变量一直被留在内存当中。

相关资料：

《JavaScript 内存泄漏教程》

《4 类 JavaScript 内存泄漏及如何避免》

《杜绝 js 中四种内存泄漏类型的发生》

《javascript 典型内存泄漏及 chrome 的排查方法》

> 以下38~46条是ECMAScript 2015(ES6)中常考的基础知识点

38. ECMAScript 是什么?

ECMAScript 是编写脚本语言的标准，这意味着JavaScript遵循ECMAScript标准中的规范变化，因为它是JavaScript的蓝图。

ECMAScript 和 Javascript，本质上都跟一门语言有关，一个是语言本身的名字，一个是语言的约束条件 只不过发明JavaScript的那个人（Netscape公司），把东西交给了ECMA（European Computer Manufacturers Association），这个人规定一下他的标准，因为当时有java语言了，又想强调这个东西是让ECMA这个人定的规则，所以就这样一个神奇的东西诞生了，这个东西的名称就叫做ECMAScript。

javaScript = ECMAScript + DOM + BOM（自认为是一种广义的JavaScript）

ECMAScript说什么JavaScript就得做什么！

JavaScript（狭义的JavaScript）做什么都要问问ECMAScript我能不能这样干！如果不能我就错了！能我就是对的！

——突然感觉JavaScript好没有尊严，为啥要搞个人出来约束自己，

那个人被创造出来也好委屈，自己被创造出来完全是因为要约束JavaScript。

39. ECMAScript 2015（ES6）有哪些新特性?

- 块作用域
- 类
- 箭头函数
- 模板字符串
- 加强的对象字面
- 对象解构
- Promise
- 模块
- Symbol
- 代理（proxy）Set
- 函数默认参数
- rest 和展开

40. `var`,`let`和`const`的区别是什么?

var声明的变量会挂载在window上，而let和const声明的变量不会：

```
var a = 100;
console.log(a,window.a);    // 100 100

let b = 10;
console.log(b,window.b);    // 10 undefined

const c = 1;
console.log(c,window.c);    // 1 undefined
复制代码
```

var声明变量存在变量提升，let和const不存在变量提升：

```
console.log(a); // undefined  ===>  a已声明还没赋值，默认得到undefined值
var a = 100;

console.log(b); // 报错：b is not defined  ===> 找不到b这个变量
let b = 10;

console.log(c); // 报错：c is not defined  ===> 找不到c这个变量
const c = 10;
复制代码
```

let和const声明形成块作用域

```
if(1){
  var a = 100;
  let b = 10;
}

console.log(a); // 100
console.log(b)  // 报错：b is not defined  ===> 找不到b这个变量

------------------------------------------------------------

if(1){
  var a = 100;
  const c = 1;
}
console.log(a); // 100
console.log(c)  // 报错：c is not defined  ===> 找不到c这个变量
复制代码
```

同一作用域下let和const不能声明同名变量，而var可以

```
var a = 100;
console.log(a); // 100

var a = 10;
console.log(a); // 10
-------------------------------------
let a = 100;
let a = 10;

//  控制台报错：Identifier 'a' has already been declared  ===> 标识符a已经被声明了。
```
复制代码

暂存死区

```
var a = 100;

if(1){
    a = 10;
    //在当前块作用域中存在a使用let/const声明的情况下，给a赋值10时，只会在当前作用域找变量a，
    // 而这时，还未到声明时候，所以控制台Error:a is not defined
    let a = 1;
}
```
复制代码

const

```
/*
*   1、一旦声明必须赋值,不能使用null占位。
*
*   2、声明后不能再修改
*
*   3、如果声明的是复合类型数据，可以修改其属性
*
* */

const a = 100;

const list = [];
list[0] = 10;
console.log(list);  // [10]

const obj = {a:100};
obj.name = 'apple';
obj.a = 10000;
console.log(obj);  // {a:10000,name:'apple'}
```
复制代码

41. 什么是箭头函数?

箭头函数表达式的语法比函数表达式更简洁，并且没有自己的 `this，arguments，super`或 `new.target`。箭头函数表达式更适用于那些本来需要匿名函数的地方，并且它不能用作构造函数。

```
//ES5 Version
var getCurrentDate = function (){
  return new Date();
}

//ES6 Version
const getCurrentDate = () => new Date();
```
复制代码

在本例中，ES5 版本中有`function(){}`声明和return关键字，这两个关键字分别是创建函数和返回值所需要的。在箭头函数版本中，我们只需要()括号，不需要 return 语句，因为如果我们只有一个表达式或值需要返回，箭头函数就会有一个隐式的返回。

```
//ES5 Version
function greet(name) {
  return 'Hello ' + name + '!';
}

//ES6 Version
const greet = (name) => `Hello ${name}`;
const greet2 = name => `Hello ${name}`;
```
复制代码

我们还可以在箭头函数中使用与函数表达式和函数声明相同的参数。如果我们在一个箭头函数中有一个参数，则可以省略括号。

```
const getArgs = () => arguments

const getArgs2 = (...rest) => rest
```
复制代码

箭头函数不能访问arguments对象。所以调用第一个getArgs函数会抛出一个错误。相反，我们可以使用rest参数来获得在箭头函数中传递的所有参数。

```
const data = {
  result: 0,
  nums: [1, 2, 3, 4, 5],
  computeResult() {
    // 这里的“this”指的是“data”对象
    const addAll = () => {
      return this.nums.reduce((total, cur) => total + cur, 0)
    };
    this.result = addAll();
  }
};
```
复制代码

箭头函数没有自己的this值。它捕获词法作用域函数的this值，在此示例中，addAll函数将复制computeResult 方法中的this值，如果我们在全局作用域声明箭头函数，则this值为 window 对象。

42. 什么是类?

类(class)是在 JS 中编写构造函数的新方法。它是使用构造函数的语法糖，在底层中使用仍然是原型和基于原型的继承。

```
 //ES5 Version
   function Person(firstName, lastName, age, address){
      this.firstName = firstName;
      this.lastName = lastName;
      this.age = age;
      this.address = address;
   }

   Person.self = function(){
     return this;
   }

   Person.prototype.toString = function(){
     return "[object Person]";
   }

   Person.prototype.getFullName = function (){
     return this.firstName + " " + this.lastName;
   }

   //ES6 Version
   class Person {
        constructor(firstName, lastName, age, address){
            this.lastName = lastName;
            this.firstName = firstName;
            this.age = age;
            this.address = address;
        }

        static self() {
           return this;
        }

        toString(){
           return "[object Person]";
        }

        getFullName(){
           return `${this.firstName} ${this.lastName}`;
        }
   }
```
复制代码

重写方法并从另一个类继承。

```
//ES5 Version
Employee.prototype = Object.create(Person.prototype);

function Employee(firstName, lastName, age, address, jobTitle, yearStarted) {
  Person.call(this, firstName, lastName, age, address);
  this.jobTitle = jobTitle;
  this.yearStarted = yearStarted;
}

Employee.prototype.describe = function () {
```

```
  return `I am ${this.getFullName()} and I have a position of ${this.jobTitle}
and I started at ${this.yearStarted}`;
}

Employee.prototype.toString = function () {
  return "[object Employee]";
}

//ES6 Version
class Employee extends Person { //Inherits from "Person" class
  constructor(firstName, lastName, age, address, jobTitle, yearStarted) {
    super(firstName, lastName, age, address);
    this.jobTitle = jobTitle;
    this.yearStarted = yearStarted;
  }

  describe() {
    return `I am ${this.getFullName()} and I have a position of ${this.jobTitle}
and I started at ${this.yearStarted}`;
  }

  toString() { // Overriding the "toString" method of "Person"
    return "[object Employee]";
  }
}
```
复制代码

所以我们要怎么知道它在内部使用原型?

```
class Something {

}

function AnotherSomething(){

}
const as = new AnotherSomething();
const s = new Something();

console.log(typeof Something); // "function"
console.log(typeof AnotherSomething); // "function"
console.log(as.toString()); // "[object Object]"
console.log(as.toString()); // "[object Object]"
console.log(as.toString === Object.prototype.toString); // true
console.log(s.toString === Object.prototype.toString); // true
```
复制代码

相关资料:

《ECMAScript 6 实现了 class，对 JavaScript 前端开发有什么意义？》

《Class 的基本语法》

43. 什么是模板字符串?

模板字符串是在 JS 中创建字符串的一种新方法。我们可以通过使用反引号使模板字符串化。

```
//ES5 Version
var greet = 'Hi I\'m Mark';

//ES6 Version
let greet = `Hi I'm Mark`;
```
复制代码

在 ES5 中我们需要使用一些转义字符来达到多行的效果，在模板字符串不需要这么麻烦：

```
//ES5 Version
var lastWords = '\n'
  + '   I  \n'
  + '   Am  \n'
  + 'Iron Man \n';


//ES6 Version
let lastWords = `
    I
    Am
  Iron Man
`;
```
复制代码

在ES5版本中，我们需要添加\n以在字符串中添加新行。在模板字符串中，我们不需要这样做。

```
//ES5 Version
function greet(name) {
  return 'Hello ' + name + '!';
}


//ES6 Version
function greet(name) {
  return `Hello ${name} !`;
}
```
复制代码

在 ES5 版本中，如果需要在字符串中添加表达式或值，则需要使用`+`运算符。在模板字符串s中，我们可以使用`${expr}`嵌入一个表达式，这使其比 ES5 版本更整洁。

44. 什么是对象解构?

对象析构是从对象或数组中获取或提取值的一种新的、更简洁的方法。假设有如下的对象：

```
const employee = {
  firstName: "Marko",
  lastName: "Polo",
  position: "Software Developer",
  yearHired: 2017
};
```
复制代码

从对象获取属性，早期方法是创建一个与对象属性同名的变量。这种方法很麻烦，因为我们要为每个属性创建一个新变量。假设我们有一个大对象，它有很多属性和方法，用这种方法提取属性会很麻烦。

```
var firstName = employee.firstName;
var lastName = employee.lastName;
var position = employee.position;
var yearHired = employee.yearHired;
复制代码
```

使用解构方式语法就变得简洁多了：

```
{ firstName, lastName, position, yearHired } = employee;
复制代码
```

我们还可以为属性取别名：

```
let { firstName: fName, lastName: lName, position, yearHired } = employee;
复制代码
```

当然如果属性值为 undefined 时，我们还可以指定默认值：

```
let { firstName = "Mark", lastName: lName, position, yearHired } = employee;
复制代码
```

45. 什么是`Set`对象，它是如何工作的?

Set 对象允许你存储任何类型的唯一值，无论是原始值或者是对象引用。

我们可以使用Set构造函数创建Set实例。

```
const set1 = new Set();
const set2 = new Set(["a","b","c","d","d","e"]);
复制代码
```

我们可以使用add方法向Set实例中添加一个新值，因为add方法返回Set对象，所以我们可以以链式的方式再次使用add。如果一个值已经存在于Set对象中，那么它将不再被添加。

```
set2.add("f");
set2.add("g").add("h").add("i").add("j").add("k").add("k");
// 后一个“k”不会被添加到set对象中，因为它已经存在了
复制代码
```

我们可以使用has方法检查Set实例中是否存在特定的值。

```
set2.has("a") // true
set2.has("z") // true
复制代码
```

我们可以使用size属性获得Set实例的长度。

```
set2.size // returns 10
复制代码
```

可以使用clear方法删除 Set 中的数据。

```
set2.clear();
复制代码
```

我们可以使用Set对象来删除数组中重复的元素。

```
const numbers = [1, 2, 3, 4, 5, 6, 6, 7, 8, 8, 5];
const uniqueNums = [...new Set(numbers)]; // [1,2,3,4,5,6,7,8]
复制代码
```

另外还有`WeakSet`， 与 `Set` 类似，也是不重复的值的集合。但是 `WeakSet` 的成员只能是对象，而不能是其他类型的值。`WeakSet` 中的对象都是弱引用，即垃圾回收机制不考虑 `WeakSet` 对该对象的引用。

- Map 数据结构。它类似于对象，也是键值对的集合，但是“键”的范围不限于字符串，各种类型的值（包括对象）都可以当作键。
- WeakMap 结构与 Map 结构类似，也是用于生成键值对的集合。但是 WeakMap 只接受对象作为键名（ null 除外），不接受其他类型的值作为键名。而且 WeakMap 的键名所指向的对象，不计入垃圾回收机制。

46. 什么是Proxy?

Proxy 用于修改某些操作的默认行为，等同于在语言层面做出修改，所以属于一种“元编程”，即对编程语言进行编程。

Proxy 可以理解成，在目标对象之前架设一层“拦截”，外界对该对象的访问，都必须先通过这层拦截，因此提供了一种机制，可以对外界的访问进行过滤和改写。Proxy 这个词的原意是代理，用在这里表示由它来“代理”某些操作，可以译为“代理器”。

> 高能预警⚡⚡⚡， 以下47~64条是JavaScript中比较难的高级知识及相关手写实现，各位看官需慢慢细品

47. 写一个通用的事件侦听器函数

```
const EventUtils = {
  // 视能力分别使用dom0||dom2||IE方式 来绑定事件
  // 添加事件
  addEvent: function(element, type, handler) {
    if (element.addEventListener) {
      element.addEventListener(type, handler, false);
    } else if (element.attachEvent) {
      element.attachEvent("on" + type, handler);
    } else {
      element["on" + type] = handler;
    }
  },

  // 移除事件
  removeEvent: function(element, type, handler) {
    if (element.removeEventListener) {
      element.removeEventListener(type, handler, false);
    } else if (element.detachEvent) {
      element.detachEvent("on" + type, handler);
    } else {
```

```
      element["on" + type] = null;
    }
  },

  // 获取事件目标
  getTarget: function(event) {
    return event.target || event.srcElement;
  },

  // 获取 event 对象的引用，取到事件的所有信息，确保随时能使用 event
  getEvent: function(event) {
    return event || window.event;
  },

  // 阻止事件（主要是事件冒泡，因为 IE 不支持事件捕获）
  stopPropagation: function(event) {
    if (event.stopPropagation) {
      event.stopPropagation();
    } else {
      event.cancelBubble = true;
    }
  },

  // 取消事件的默认行为
  preventDefault: function(event) {
    if (event.preventDefault) {
      event.preventDefault();
    } else {
      event.returnValue = false;
    }
  }
};
复制代码
```

48. 什么是函数式编程? JavaScript的哪些特性使其成为函数式语言的候选语言?

函数式编程（通常缩写为FP）是通过编写纯函数，避免共享状态、可变数据、副作用 来构建软件的过程。数式编程是声明式 的而不是命令式 的，应用程序的状态是通过纯函数流动的。与面向对象编程形成对比，面向对象中应用程序的状态通常与对象中的方法共享和共处。

函数式编程是一种编程范式 ，这意味着它是一种基于一些基本的定义原则（如上所列）思考软件构建的方式。当然，编程范式的其他示例也包括面向对象编程和过程编程。

函数式的代码往往比命令式或面向对象的代码更简洁，更可预测，更容易测试 - 但如果不熟悉它以及与之相关的常见模式，函数式的代码也可能看起来更密集杂乱，并且 相关文献对新人来说是不好理解的。

49. 什么是高阶函数?

高阶函数只是将函数作为参数或返回值的函数。

```
function higherOrderFunction(param,callback){
    return callback(param);
}
复制代码
```

50. 为什么函数被称为一等公民?

在JavaScript中，函数不仅拥有一切传统函数的使用方式（声明和调用），而且可以做到像简单值一样:

- 赋值（`var func = function(){}`）、
- 传参(`function func(x,callback){callback();}`)、
- 返回(`function(){return function(){}}`),

这样的函数也称之为第一级函数（`First-class Function`）。不仅如此，JavaScript中的函数还充当了类的构造函数的作用，同时又是一个Function类的实例(instance)。这样的多重身份让JavaScript的函数变得非常重要。

51. 手动实现 `Array.prototype.map 方法`

map() 方法创建一个新数组，其结果是该数组中的每个元素都调用一个提供的函数后返回的结果。

```
function map(arr, mapCallback) {
  // 首先，检查传递的参数是否正确。
  if (!Array.isArray(arr) || !arr.length || typeof mapCallback !== 'function') {
    return [];
  } else {
    let result = [];
    // 每次调用此函数时，我们都会创建一个 result 数组
    // 因为我们不想改变原始数组。
    for (let i = 0, len = arr.length; i < len; i++) {
      result.push(mapCallback(arr[i], i, arr));
      // 将 mapCallback 返回的结果 push 到 result 数组中
    }
    return result;
  }
}
复制代码
```

52. 手动实现`Array.prototype.filter`方法

`filter()`方法创建一个新数组, 其包含通过所提供函数实现的测试的所有元素。

```
function filter(arr, filterCallback) {
  // 首先，检查传递的参数是否正确。
  if (!Array.isArray(arr) || !arr.length || typeof filterCallback !==
'function')
  {
    return [];
  } else {
    let result = [];
     // 每次调用此函数时，我们都会创建一个 result 数组
     // 因为我们不想改变原始数组。
    for (let i = 0, len = arr.length; i < len; i++) {
      // 检查 filterCallback 的返回值是否是真值
      if (filterCallback(arr[i], i, arr)) {
      // 如果条件为真，则将数组元素 push 到 result 中
        result.push(arr[i]);
      }
    }
    return result; // return the result array
  }
```

```
}
```
复制代码

53. 手动实现`Array.prototype.reduce`方法

reduce() 方法对数组中的每个元素执行一个由您提供的reducer函数(升序执行)，将其结果汇总为单个返回值。

```
function reduce(arr, reduceCallback, initialValue) {
  // 首先，检查传递的参数是否正确。
  if (!Array.isArray(arr) || !arr.length || typeof reduceCallback !==
'function')
  {
    return [];
  } else {
    // 如果没有将initialValue传递给该函数，我们将使用第一个数组项作为initialValue
    let hasInitialValue = initialValue !== undefined;
    let value = hasInitialValue ? initialValue : arr[0];
   、

    // 如果有传递 initialValue，则索引从 1 开始，否则从 0 开始
    for (let i = hasInitialValue ? 0 : 1, len = arr.length; i < len; i++) {
      value = reduceCallback(value, arr[i], i, arr);
    }
    return value;
  }
}
```
复制代码

54. js的深浅拷贝

JavaScript的深浅拷贝一直是个难点，如果现在面试官让我写一个深拷贝，我可能也只是能写出个基础版的。所以在写这条之前我拜读了收藏夹里各路大佬写的博文。具体可以看下面我贴的链接，这里只做简单的总结。

- **浅拷贝**： 创建一个新对象，这个对象有着原始对象属性值的一份精确拷贝。如果属性是基本类型，拷贝的就是基本类型的值，如果属性是引用类型，拷贝的就是内存地址 ，所以如果其中一个对象改变了这个地址，就会影响到另一个对象。
- **深拷贝**： 将一个对象从内存中完整的拷贝一份出来,从堆内存中开辟一个新的区域存放新对象,且修改新对象不会影响原对象。

浅拷贝的实现方式:

- **Object.assign() 方法**： 用于将所有可枚举属性的值从一个或多个源对象复制到目标对象。它将返回目标对象。
- **Array.prototype.slice()**：slice() 方法返回一个新的数组对象，这一对象是一个由 begin和 end（不包括end）决定的原数组的浅拷贝。原始数组不会被改变。
- **拓展运算符`...`**：

```
let a = {
    name: "Jake",
    flag: {
        title: "better day by day",
        time: "2020-05-31"
    }
}
let b = {...a};
```
复制代码

深拷贝的实现方式：

- **乞丐版：** JSON.parse(JSON.stringify(object))，缺点诸多（会忽略undefined、symbol、函数；不能解决循环引用；不能处理正则、new Date())
- **基础版（面试够用）：** 浅拷贝+递归 （只考虑了普通的 object和 array两种数据类型）

```
function cloneDeep(target,map = new WeakMap()) {
  if(typeOf taret ==='object'){
     let cloneTarget = Array.isArray(target) ? [] : {};

     if(map.get(target)) {
        return target;
    }
     map.set(target, cloneTarget);
     for(const key in target){
        cloneTarget[key] = cloneDeep(target[key], map);
     }
     return cloneTarget
  }else{
       return target
  }

}

```
复制代码

- **终极版：**

```
const mapTag = '[object Map]';
const setTag = '[object Set]';
const arrayTag = '[object Array]';
const objectTag = '[object Object]';
const argsTag = '[object Arguments]';

const boolTag = '[object Boolean]';
const dateTag = '[object Date]';
const numberTag = '[object Number]';
const stringTag = '[object String]';
const symbolTag = '[object Symbol]';
const errorTag = '[object Error]';
const regexpTag = '[object RegExp]';
const funcTag = '[object Function]';

const deepTag = [mapTag, setTag, arrayTag, objectTag, argsTag];
```

```
function forEach(array, iteratee) {
    let index = -1;
    const length = array.length;
    while (++index < length) {
        iteratee(array[index], index);
    }
    return array;
}

function isObject(target) {
    const type = typeof target;
    return target !== null && (type === 'object' || type === 'function');
}

function getType(target) {
    return Object.prototype.toString.call(target);
}

function getInit(target) {
    const Ctor = target.constructor;
    return new Ctor();
}

function cloneSymbol(targe) {
    return Object(Symbol.prototype.valueOf.call(targe));
}

function cloneReg(targe) {
    const reFlags = /\w*$/;
    const result = new targe.constructor(targe.source, reFlags.exec(targe));
    result.lastIndex = targe.lastIndex;
    return result;
}

function cloneFunction(func) {
    const bodyReg = /(?<={)(.|\n)+(?=})/m;
    const paramReg = /(?<=\().+(?=\)\s+{)/;
    const funcString = func.toString();
    if (func.prototype) {
        const param = paramReg.exec(funcString);
        const body = bodyReg.exec(funcString);
        if (body) {
            if (param) {
                const paramArr = param[0].split(',');
                return new Function(...paramArr, body[0]);
            } else {
                return new Function(body[0]);
            }
        } else {
            return null;
        }
    } else {
        return eval(funcString);
    }
}

function cloneOtherType(targe, type) {
    const Ctor = targe.constructor;
```

```
    switch (type) {
        case boolTag:
        case numberTag:
        case stringTag:
        case errorTag:
        case dateTag:
            return new Ctor(targe);
        case regexpTag:
            return cloneReg(targe);
        case symbolTag:
            return cloneSymbol(targe);
        case funcTag:
            return cloneFunction(targe);
        default:
            return null;
    }
}

function clone(target, map = new WeakMap()) {

    // 克隆原始类型
    if (!isObject(target)) {
        return target;
    }

    // 初始化
    const type = getType(target);
    let cloneTarget;
    if (deepTag.includes(type)) {
        cloneTarget = getInit(target, type);
    } else {
        return cloneOtherType(target, type);
    }

    // 防止循环引用
    if (map.get(target)) {
        return map.get(target);
    }
    map.set(target, cloneTarget);

    // 克隆set
    if (type === setTag) {
        target.forEach(value => {
            cloneTarget.add(clone(value, map));
        });
        return cloneTarget;
    }

    // 克隆map
    if (type === mapTag) {
        target.forEach((value, key) => {
            cloneTarget.set(key, clone(value, map));
        });
        return cloneTarget;
    }

    // 克隆对象和数组
    const keys = type === arrayTag ? undefined : Object.keys(target);
```

```
    forEach(keys || target, (value, key) => {
        if (keys) {
            key = value;
        }
        cloneTarget[key] = clone(target[key], map);
    });

    return cloneTarget;
}

module.exports = {
    clone
};
```
复制代码

参考文章：

如何写出一个惊艳面试官的深拷贝

深拷贝的终极探索（99%的人都不知道）

55. 手写call、apply及bind函数

call 函数的实现步骤：

- 1.判断调用对象是否为函数，即使我们是定义在函数的原型上的，但是可能出现使用 call 等方式调用的情况。
- 2.判断传入上下文对象是否存在，如果不存在，则设置为 window 。
- 3.处理传入的参数，截取第一个参数后的所有参数。
- 4.将函数作为上下文对象的一个属性。
- 5.使用上下文对象来调用这个方法，并保存返回结果。
- 6.删除刚才新增的属性。
- 7.返回结果。

```
// call函数实现
Function.prototype.myCall = function(context) {
  // 判断调用对象
  if (typeof this !== "function") {
    console.error("type error");
  }

  // 获取参数
  let args = [...arguments].slice(1),
    result = null;

  // 判断 context 是否传入，如果未传入则设置为 window
  context = context || window;

  // 将调用函数设为对象的方法
  context.fn = this;

  // 调用函数
  result = context.fn(...args);

  // 将属性删除
  delete context.fn;
```

```
  return result;
};
```
复制代码

apply 函数的实现步骤：

- 1. 判断调用对象是否为函数，即使我们是定义在函数的原型上的，但是可能出现使用 call 等方式调用的情况。
- 1. 判断传入上下文对象是否存在，如果不存在，则设置为 window 。
- 1. 将函数作为上下文对象的一个属性。
- 1. 判断参数值是否传入
- 1. 使用上下文对象来调用这个方法，并保存返回结果。
- 1. 删除刚才新增的属性
- 1. 返回结果

```
// apply 函数实现

Function.prototype.myApply = function(context) {
  // 判断调用对象是否为函数
  if (typeof this !== "function") {
    throw new TypeError("Error");
  }

  let result = null;

  // 判断 context 是否存在，如果未传入则为 window
  context = context || window;

  // 将函数设为对象的方法
  context.fn = this;

  // 调用方法
  if (arguments[1]) {
    result = context.fn(...arguments[1]);
  } else {
    result = context.fn();
  }

  // 将属性删除
  delete context.fn;

  return result;
};


```
复制代码

bind 函数的实现步骤：

- 1.判断调用对象是否为函数，即使我们是定义在函数的原型上的，但是可能出现使用 call 等方式调用的情况。
- 2.保存当前函数的引用，获取其余传入参数值。
- 3.创建一个函数返回
- 4.函数内部使用 apply 来绑定函数调用，需要判断函数作为构造函数的情况，这个时候需要传入当前函数的 this 给 apply 调用，其余情况都传入指定的上下文对象。

```
// bind 函数实现
Function.prototype.myBind = function(context) {
  // 判断调用对象是否为函数
  if (typeof this !== "function") {
    throw new TypeError("Error");
  }

  // 获取参数
  var args = [...arguments].slice(1),
    fn = this;

  return function Fn() {
    // 根据调用方式，传入不同绑定值
    return fn.apply(
      this instanceof Fn ? this : context,
      args.concat(...arguments)
    );
  };
};
```
复制代码

参考文章： 《手写 call、apply 及 bind 函数》

《JavaScript 深入之 call 和 apply 的模拟实现》

56. 函数柯里化的实现

```
// 函数柯里化指的是一种将使用多个参数的一个函数转换成一系列使用一个参数的函数的技术。

function curry(fn, args) {
  // 获取函数需要的参数长度
  let length = fn.length;

  args = args || [];

  return function() {
    let subArgs = args.slice(0);

    // 拼接得到现有的所有参数
    for (let i = 0; i < arguments.length; i++) {
      subArgs.push(arguments[i]);
    }

    // 判断参数的长度是否已经满足函数所需参数的长度
    if (subArgs.length >= length) {
      // 如果满足，执行函数
      return fn.apply(this, subArgs);
    } else {
      // 如果不满足，递归返回科里化的函数，等待参数的传入
      return curry.call(this, fn, subArgs);
    }
  };
}

// es6 实现
function curry(fn, ...args) {
  return fn.length <= args.length ? fn(...args) : curry.bind(null, fn, ...args);
```

```
}
```
复制代码

参考文章： 《JavaScript 专题之函数柯里化》

57. js模拟new操作符的实现

这个问题如果你在掘金上搜，你可能会搜索到类似下面的回答：

`new` 操作符做了这些事：

- 它创建了一个全新的对象。
- 它会被执行 `[[Prototype]]` （也就是 `__proto__` ）链接。
- 它使 `this` 指向新创建的对象。。
- 通过 `new` 创建的每个对象将最终被 `[[Prototype]]` 链接到这个函数的 `prototype` 对象上。
- 如果函数没有返回对象类型 `Object` (包含 `Functoin, Array, Date, RegExg, Error`)，那么 `new` 表达式中的函数调用将返回该对象引用。

说实话，看第一遍，我是不理解的，我需要去理一遍原型及原型链的知识才能理解。所以我觉得MDN对new的解释更容易理解：

`new` 运算符创建一个用户定义的对象类型的实例或具有构造函数的内置对象的实例。 `new` 关键字会进行如下的操作：

1. 创建一个空的简单JavaScript对象（即{}）；
2. 链接该对象（即设置该对象的构造函数）到另一个对象；
3. 将步骤1新创建的对象作为this的上下文；
4. 如果该函数没有返回对象，则返回this。

接下来我们看实现：

```
function Dog(name, color, age) {
  this.name = name;
  this.color = color;
  this.age = age;
}

Dog.prototype={
  getName: function() {
    return this.name
  }
}

var dog = new Dog('大黄', 'yellow', 3)
```
复制代码

上面的代码相信不用解释，大家都懂。我们来看最后一行带 `new` 关键字的代码，按照上述的1,2,3,4步来解析 `new` 背后的操作。

第一步：创建一个简单空对象

```
var obj = {}
复制代码
```

第二步：链接该对象到另一个对象（原型链）

```
// 设置原型链
obj.__proto__ = Dog.prototype
复制代码
```

第三步：将步骤1新创建的对象作为 `this` 的上下文

```
// this指向obj对象
Dog.apply(obj, ['大黄', 'yellow', 3])
复制代码
```

第四步：如果该函数没有返回对象，则返回this

```
// 因为 Dog() 没有返回值，所以返回obj
var dog = obj
dog.getName() // '大黄'
复制代码
```

需要注意的是如果 Dog() 有 return 则返回 return的值

```
var rtnObj = {}
function Dog(name, color, age) {
  // ...
  //返回一个对象
  return rtnObj
}

var dog = new Dog('大黄', 'yellow', 3)
console.log(dog === rtnObj) // true

复制代码
```

接下来我们将以上步骤封装成一个对象实例化方法，即模拟new的操作：

```
function objectFactory(){
    var obj = {};
    //取得该方法的第一个参数(并删除第一个参数)，该参数是构造函数
    var Constructor = [].shift.apply(arguments);
    //将新对象的内部属性__proto__指向构造函数的原型，这样新对象就可以访问原型中的属性和方法
    obj.__proto__ = Constructor.prototype;
    //取得构造函数的返回值
    var ret = Constructor.apply(obj, arguments);
    //如果返回值是一个对象就返回该对象，否则返回构造函数的一个实例对象
    return typeof ret === "object" ? ret : obj;
}

复制代码
```

58. 什么是回调函数？回调函数有什么缺点

回调函数是一段可执行的代码段，它作为一个参数传递给其他的代码，其作用是在需要的时候方便调用这段（回调函数）代码。

在JavaScript中函数也是对象的一种，同样对象可以作为参数传递给函数，因此函数也可以作为参数传递给另外一个函数，这个作为参数的函数就是回调函数。

```
const btnAdd = document.getElementById('btnAdd');

btnAdd.addEventListener('click', function clickCallback(e) {
    // do something useless
});
```
复制代码

在本例中，我们等待id为`btnAdd`的元素中的`click`事件，如果它被单击，则执行`clickCallback`函数。回调函数向某些数据或事件添加一些功能。

回调函数有一个致命的弱点，就是容易写出回调地狱（Callback hell）。假设多个事件存在依赖性：

```
setTimeout(() => {
    console.log(1)
    setTimeout(() => {
        console.log(2)
        setTimeout(() => {
            console.log(3)

        },3000)

    },2000)
},1000)
```
复制代码

这就是典型的回调地狱，以上代码看起来不利于阅读和维护，事件一旦多起来就更是乱糟糟，所以在es6中提出了Promise和async/await来解决回调地狱的问题。当然，回调函数还存在着别的几个缺点，比如不能使用 try catch 捕获错误，不能直接 return。接下来的两条就是来解决这些问题的，咱们往下看。

59. Promise是什么，可以手写实现一下吗?

Promise，翻译过来是承诺，承诺它过一段时间会给你一个结果。从编程讲Promise 是异步编程的一种解决方案。下面是Promise在MDN的相关说明：

Promise 对象是一个代理对象（代理一个值），被代理的值在Promise对象创建时可能是未知的。它允许你为异步操作的成功和失败分别绑定相应的处理方法（handlers）。 这让异步方法可以像同步方法那样返回值，但并不是立即返回最终执行结果，而是一个能代表未来出现的结果的promise对象。

一个 Promise有以下几种状态:

- pending: 初始状态，既不是成功，也不是失败状态。
- fulfilled: 意味着操作成功完成。
- rejected: 意味着操作失败。

这个承诺一旦从等待状态变成为其他状态就永远不能更改状态了，也就是说一旦状态变为 fulfilled/rejected 后，就不能再次改变。 可能光看概念大家不理解Promise，我们举个简单的栗子；

假如我有个女朋友，下周一是她生日，我答应她生日给她一个惊喜，那么从现在开始这个承诺就进入等待状态，等待下周一的到来，然后状态改变。如果下周一我如约给了女朋友惊喜，那么这个承诺的状态就会由pending切换为fulfilled，表示承诺成功兑现，一旦是这个结果了，就不会再有其他结果，即状态不会在发生改变；反之如果当天我因为工作太忙加班，把这事给忘了，说好的惊喜没有兑现，状态就会由pending切换为rejected，时间不可倒流，所以状态也不能再发生变化。

上一条我们说过Promise可以解决回调地狱的问题，没错，pending 状态的 Promise 对象会触发 fulfilled/rejected 状态，一旦状态改变，Promise 对象的 then 方法就会被调用；否则就会触发 catch。我们将上一条回调地狱的代码改写一下：

```
new Promise((resolve，reject) => {
     setTimeout(() => {
            console.log(1)
            resolve()
        },1000)

}).then((res) => {
    setTimeout(() => {
            console.log(2)
        },2000)
}).then((res) => {
    setTimeout(() => {
            console.log(3)
        },3000)
}).catch((err) => {
console.log(err)
})
```
复制代码

其实Promise也是存在一些缺点的，比如无法取消 Promise，错误需要通过回调函数捕获。

promise手写实现，面试够用版：

```
function myPromise(constructor){
    let self=this;
    self.status="pending" //定义状态改变前的初始状态
    self.value=undefined;//定义状态为resolved的时候的状态
    self.reason=undefined;//定义状态为rejected的时候的状态
    function resolve(value){
        //两个==="pending"，保证了状态的改变是不可逆的
       if(self.status==="pending"){
          self.value=value;
          self.status="resolved";
       }
    }
    function reject(reason){
        //两个==="pending"，保证了状态的改变是不可逆的
       if(self.status==="pending"){
          self.reason=reason;
          self.status="rejected";
       }
    }
    //捕获构造异常
    try{
       constructor(resolve,reject);
    }catch(e){
```

```
        reject(e);
    }
}
// 定义链式调用的then方法
myPromise.prototype.then=function(onFullfilled,onRejected){
   let self=this;
   switch(self.status){
      case "resolved":
        onFullfilled(self.value);
        break;
      case "rejected":
        onRejected(self.reason);
        break;
      default:
   }
}
```

复制代码

关于Promise还有其他的知识，比如Promise.all()、Promise.race()等的运用，由于篇幅原因就不再做展开，想要深入了解的可看下面的文章。

相关资料：

[「硬核JS」深入了解异步解决方案]

[【翻译】Promises/A+规范]

60. `Iterator`是什么，有什么作用?

`Iterator`是理解第24条的先决知识，也许是我IQ不够😭，`Iterator`和`Generator`看了很多遍还是一知半解，即使当时理解了，过一阵又忘得一干二净。。。

Iterator（迭代器）是一种接口，也可以说是一种规范。为各种不同的数据结构提供统一的访问机制。任何数据结构只要部署Iterator接口，就可以完成遍历操作（即依次处理该数据结构的所有成员）。

Iterator语法：

```
const obj = {
    [Symbol.iterator]:function(){}
}
```

复制代码

`[Symbol.iterator]`属性名是固定的写法，只要拥有了该属性的对象，就能够用迭代器的方式进行遍历。

迭代器的遍历方法是首先获得一个迭代器的指针，初始时该指针指向第一条数据之前，接着通过调用 next 方法，改变指针的指向，让其指向下一条数据 每一次的 next 都会返回一个对象，该对象有两个属性

- value 代表想要获取的数据
- done 布尔值，false表示当前指针指向的数据有值，true表示遍历已经结束

Iterator 的作用有三个：

1. 为各种数据结构，提供一个统一的、简便的访问接口；
2. 使得数据结构的成员能够按某种次序排列；

3. ES6 创造了一种新的遍历命令for...of循环，Iterator 接口主要供for...of消费。

遍历过程：

1. 创建一个指针对象，指向当前数据结构的起始位置。也就是说，遍历器对象本质上，就是一个指针对象。
2. 第一次调用指针对象的next方法，可以将指针指向数据结构的第一个成员。
3. 第二次调用指针对象的next方法，指针就指向数据结构的第二个成员。
4. 不断调用指针对象的next方法，直到它指向数据结构的结束位置。

每一次调用next方法，都会返回数据结构的当前成员的信息。具体来说，就是返回一个包含value和done两个属性的对象。其中，value属性是当前成员的值，done属性是一个布尔值，表示遍历是否结束。

```
let arr = [{num:1},2,3]
let it = arr[Symbol.iterator]() // 获取数组中的迭代器
console.log(it.next())  // { value: Object { num: 1 }, done: false }
console.log(it.next())  // { value: 2, done: false }
console.log(it.next())  // { value: 3, done: false }
console.log(it.next())  // { value: undefined, done: true }

复制代码
```

61. `Generator`函数是什么，有什么作用?

Generator函数可以说是Iterator接口的具体实现方式。Generator 最大的特点就是可以控制函数的执行。

```
function *foo(x) {
  let y = 2 * (yield (x + 1))
  let z = yield (y / 3)
  return (x + y + z)
}
let it = foo(5)
console.log(it.next())   // => {value: 6, done: false}
console.log(it.next(12)) // => {value: 8, done: false}
console.log(it.next(13)) // => {value: 42, done: true}

复制代码
```

上面这个示例就是一个Generator函数，我们来分析其执行过程：

- `首先 Generator 函数调用时它会返回一个迭代器`
- `当执行第一次 next 时，传参会被忽略，并且函数暂停在 yield (x + 1) 处，所以返回 5 + 1 = 6`
- `当执行第二次 next 时，传入的参数等于上一个 yield 的返回值，如果你不传参，yield 永远返回 undefined。此时 let y = 2 * 12，所以第二个 yield 等于 2 * 12 / 3 = 8`
- `当执行第三次 next 时，传入的参数会传递给 z，所以 z = 13, x = 5, y = 24，相加等于 42`

`Generator` 函数一般见到的不多，其实也于他有点绕有关系，并且一般会配合 co 库去使用。当然，我们可以通过 `Generator` 函数解决回调地狱的问题。

62. 什么是 `async/await` 及其如何工作,有什么优缺点?

`async/await`是一种建立在Promise之上的编写异步或非阻塞代码的新方法，被普遍认为是 JS异步操作的最终且最优雅的解决方案。相对于 Promise 和回调，它的可读性和简洁度都更高。毕竟一直then()也很烦。

`async` 是异步的意思，而 `await` 是 `async wait`的简写，即异步等待。

所以从语义上就很好理解 async 用于声明一个 function 是异步的，而await 用于等待一个异步方法执行完成。

一个函数如果加上 async ，那么该函数就会返回一个 Promise

```
async function test() {
  return "1"
}
console.log(test()) // -> Promise {<resolved>: "1"}
复制代码
```

可以看到输出的是一个Promise对象。所以，async 函数返回的是一个 Promise 对象，如果在 async 函数中直接 return 一个直接量，async 会把这个直接量通过 `PromIse.resolve()`封装成Promise对象返回。

相比于 `Promise`，`async/await`能更好地处理 then 链

```
function takeLongTime(n) {
    return new Promise(resolve => {
        setTimeout(() => resolve(n + 200), n);
    });
}

function step1(n) {
    console.log(`step1 with ${n}`);
    return takeLongTime(n);
}

function step2(n) {
    console.log(`step2 with ${n}`);
    return takeLongTime(n);
}

function step3(n) {
    console.log(`step3 with ${n}`);
    return takeLongTime(n);
}

复制代码
```

现在分别用 `Promise` 和 `async/await`来实现这三个步骤的处理。

使用Promise

```
function doIt() {
    console.time("doIt");
    const time1 = 300;
    step1(time1)
        .then(time2 => step2(time2))
        .then(time3 => step3(time3))
        .then(result => {
```

```
            console.log(`result is ${result}`);
        });
}
doIt();
// step1 with 300
// step2 with 500
// step3 with 700
// result is 900
```

复制代码

使用 `async/await`

```
async function doIt() {
    console.time("doIt");
    const time1 = 300;
    const time2 = await step1(time1);
    const time3 = await step2(time2);
    const result = await step3(time3);
    console.log(`result is ${result}`);
}
doIt();
```

复制代码

结果和之前的 Promise 实现是一样的，但是这个代码看起来是不是清晰得多，优雅整洁，几乎跟同步代码一样。

> await关键字只能在async function中使用。在任何非async function的函数中使用await关键字都会抛出错误。await关键字在执行下一行代码之前等待右侧表达式(可能是一个Promise)返回。

优缺点：

`async/await` 的优势在于处理 then 的调用链，能够更清晰准确的写出代码，并且也能优雅地解决回调地狱问题。当然也存在一些缺点，因为 await 将异步代码改造成了同步代码，如果多个异步代码没有依赖性却使用了 await 会导致性能上的降低。

参考文章：

「硬核JS」深入了解异步解决方案

以上21~25条就是JavaScript中主要的异步解决方案了，难度是有的，需要好好揣摩并加以练习。

63. instanceof的原理是什么，如何实现

instanceof 可以正确的判断对象的类型，因为内部机制是通过判断对象的原型链中是不是能找到类型的 prototype。

实现 instanceof：

1. 首先获取类型的原型
2. 然后获得对象的原型
3. 然后一直循环判断对象的原型是否等于类型的原型，直到对象原型为 null，因为原型链最终为 null

```
function myInstanceof(left, right) {
  let prototype = right.prototype
  left = left.__proto__
  while (true) {
    if (left === null || left === undefined)
      return false
    if (prototype === left)
      return true
    left = left.__proto__
  }
}
```
复制代码

64. js 的节流与防抖

函数防抖 是指在事件被触发 n 秒后再执行回调，如果在这 n 秒内事件又被触发，则重新计时。这可以使用在一些点击请求的事件上，避免因为用户的多次点击向后端发送多次请求。

函数节流 是指规定一个单位时间，在这个单位时间内，只能有一次触发事件的回调函数执行，如果在同一个单位时间内某事件被触发多次，只有一次能生效。节流可以使用在 scroll 函数的事件监听上，通过事件节流来降低事件调用的频率。

```
// 函数防抖的实现
function debounce(fn, wait) {
  var timer = null;

  return function() {
    var context = this,
      args = arguments;

    // 如果此时存在定时器的话，则取消之前的定时器重新记时
    if (timer) {
      clearTimeout(timer);
      timer = null;
    }

    // 设置定时器，使事件间隔指定事件后执行
    timer = setTimeout(() => {
      fn.apply(context, args);
    }, wait);
  };
}

// 函数节流的实现;
function throttle(fn, delay) {
  var preTime = Date.now();

  return function() {
    var context = this,
      args = arguments,
      nowTime = Date.now();

    // 如果两次时间间隔超过了指定时间，则执行函数。
    if (nowTime - preTime >= delay) {
      preTime = Date.now();
      return fn.apply(context, args);
```

```
    }
  };
}
```
复制代码

详细资料可以参考：

《轻松理解 JS 函数节流和函数防抖》

《JavaScript 事件节流和事件防抖》

《JS 的防抖与节流》

65. 什么是设计模式?

1. 概念

设计模式是一套被反复使用的、多数人知晓的、经过分类编目的、代码设计经验的总结。使用设计模式是为了重用代码、让代码更容易被他人理解、保证代码可靠性。 毫无疑问，设计模式于己于他人于系统都是多赢的，设计模式使代码编制真正工程化，设计模式是软件工程的基石，如同大厦的一块块砖石一样。

2. 设计原则

1. **S – Single Responsibility Principle 单一职责原则**
    - 一个程序只做好一件事
    - 如果功能过于复杂就拆分开，每个部分保持独立
2. **O – OpenClosed Principle 开放/封闭原则**
    - 对扩展开放，对修改封闭
    - 增加需求时，扩展新代码，而非修改已有代码
3. **L – Liskov Substitution Principle 里氏替换原则**
    - 子类能覆盖父类
    - 父类能出现的地方子类就能出现
4. **I – Interface Segregation Principle 接口隔离原则**
    - 保持接口的单一独立
    - 类似单一职责原则，这里更关注接口
5. **D – Dependency Inversion Principle 依赖倒转原则**
    - 面向接口编程，依赖于抽象而不依赖于具
    - 使用方只关注接口而不关注具体类的实现

3. 设计模式的类型

1. **结构型模式（Structural Patterns）：** 通过识别系统中组件间的简单关系来简化系统的设计。
2. **创建型模式（Creational Patterns）：** 处理对象的创建，根据实际情况使用合适的方式创建对象。常规的对象创建方式可能会导致设计上的问题，或增加设计的复杂度。创建型模式通过以某种方式控制对象的创建来解决问题。
3. **行为型模式（Behavioral Patterns）：** 用于识别对象之间常见的交互模式并加以实现，如此，增加了这些交互的灵活性。

66. 9种前端常见的设计模式

1. 外观模式（Facade Pattern）

外观模式是最常见的设计模式之一，它为子系统中的一组接口提供一个统一的高层接口，使子系统更容易使用。简而言之外观设计模式就是把多个子系统中复杂逻辑进行抽象，从而提供一个更统一、更简洁、更易用的API。很多我们常用的框架和库基本都遵循了外观设计模式，比如JQuery就把复杂的原生DOM操作进行了抽象和封装，并消除了浏览器之间的兼容问题，从而提供了一个更高级更易用的版本。其实在平时工作中我们也会经常用到外观模式进行开发，只是我们不自知而已。

1. 兼容浏览器事件绑定

```
let addMyEvent = function (el, ev, fn) {
    if (el.addEventListener) {
        el.addEventListener(ev, fn, false)
    } else if (el.attachEvent) {
        el.attachEvent('on' + ev, fn)
    } else {
        el['on' + ev] = fn
    }
};
```
复制代码

1. 封装接口

```
let myEvent = {
    // ...
    stop: e => {
        e.stopPropagation();
        e.preventDefault();
    }
};
```
复制代码

场景

- 设计初期，应该要有意识地将不同的两个层分离，比如经典的三层结构，在数据访问层和业务逻辑层、业务逻辑层和表示层之间建立外观Facade
- 在开发阶段，子系统往往因为不断的重构演化而变得越来越复杂，增加外观Facade可以提供一个简单的接口，减少他们之间的依赖。
- 在维护一个遗留的大型系统时，可能这个系统已经很难维护了，这时候使用外观Facade也是非常合适的，为系系统开发一个外观Facade类，为设计粗糙和高度复杂的遗留代码提供比较清晰的接口，让新系统和Facade对象交互，Facade与遗留代码交互所有的复杂工作。

优点

- 减少系统相互依赖。
- 提高灵活性。
- 提高了安全性

缺点

- 不符合开闭原则，如果要改东西很麻烦，继承重写都不合适。

2. 代理模式（Proxy Pattern）

是为一个对象提供一个代用品或占位符，以便控制对它的访问

> 假设当A 在心情好的时候收到花，小明表白成功的几率有 60%，而当A 在心情差的时候收到花，小明表白的成功率无限趋近于0。 小明跟A 刚刚认识两天，还无法辨别A 什么时候心情好。如果不合时宜地把花送给A，花 被直接扔掉的可能性很大，这束花可是小明吃了7 天泡面换来的。 但是A 的朋友B 却很了解A，所以小明只管把花交给B，B 会监听A 的心情变化，然后选 择A 心情好的时候把花转交给A，代码如下：

```
let Flower = function() {}
let xiaoming = {
  sendFlower: function(target) {
    let flower = new Flower()
    target.receiveFlower(flower)
  }
}
let B = {
  receiveFlower: function(flower) {
    A.listenGoodMood(function() {
      A.receiveFlower(flower)
    })
  }
}
let A = {
  receiveFlower: function(flower) {
    console.log('收到花'+ flower)
  },
  listenGoodMood: function(fn) {
    setTimeout(function() {
      fn()
    }, 1000)
  }
}
xiaoming.sendFlower(B)
```
复制代码

场景

- HTML元 素事件代理

```
<ul id="ul">
  <li>1</li>
  <li>2</li>
  <li>3</li>
</ul>
<script>
  let ul = document.querySelector('#ul');
  ul.addEventListener('click', event => {
    console.log(event.target);
  });
</script>
```
复制代码

- ES6 的 proxy 阮一峰Proxy
- jQuery.proxy()方法

优点

- 代理模式能将代理对象与被调用对象分离，降低了系统的耦合度。代理模式在客户端和目标对象之间起到一个中介作用，这样可以起到保护目标对象的作用
- 代理对象可以扩展目标对象的功能；通过修改代理对象就可以了，符合开闭原则；

缺点

- 处理请求速度可能有差别，非直接访问存在开销

3. 工厂模式（Factory Pattern）

工厂模式定义一个用于创建对象的接口，这个接口由子类决定实例化哪一个类。该模式使一个类的实例化延迟到了子类。而子类可以重写接口方法以便创建的时候指定自己的对象类型。

```
class Product {
    constructor(name) {
        this.name = name
    }
    init() {
        console.log('init')
    }
    fun() {
        console.log('fun')
    }
}

class Factory {
    create(name) {
        return new Product(name)
    }
}

// use
let factory = new Factory()
let p = factory.create('p1')
p.init()
p.fun()
```
复制代码

场景

- 如果你不想让某个子系统与较大的那个对象之间形成强耦合，而是想运行时从许多子系统中进行挑选的话，那么工厂模式是一个理想的选择
- 将new操作简单封装，遇到new的时候就应该考虑是否用工厂模式；
- 需要依赖具体环境创建不同实例，这些实例都有相同的行为,这时候我们可以使用工厂模式，简化实现的过程，同时也可以减少每种对象所需的代码量，有利于消除对象间的耦合，提供更大的灵活性

优点

- 创建对象的过程可能很复杂，但我们只需要关心创建结果。
- 构造函数和创建者分离, 符合“开闭原则”
- 一个调用者想创建一个对象，只要知道其名称就可以了。
- 扩展性高，如果想增加一个产品，只要扩展一个工厂类就可以。

缺点

- 添加新产品时，需要编写新的具体产品类,一定程度上增加了系统的复杂度

- 考虑到系统的可扩展性，需要引入抽象层，在客户端代码中均使用抽象层进行定义，增加了系统的抽象性和理解难度

什么时候不用

- 当被应用到错误的问题类型上时,这一模式会给应用程序引入大量不必要的复杂性.除非为创建对象提供一个接口是我们编写的库或者框架的一个设计上目标,否则我会建议使用明确的构造器,以避免不必要的开销。
- 由于对象的创建过程被高效的抽象在一个接口后面的事实,这也会给依赖于这个过程可能会有多复杂的单元测试带来问题。

4. 单例模式（Singleton Pattern）

顾名思义，单例模式中Class的实例个数最多为1。当需要一个对象去贯穿整个系统执行某些任务时，单例模式就派上了用场。而除此之外的场景尽量避免单例模式的使用，因为单例模式会引入全局状态，而一个健康的系统应该避免引入过多的全局状态。

实现单例模式需要解决以下几个问题：

- 如何确定Class只有一个实例?
- 如何简便的访问Class的唯一实例?
- Class如何控制实例化的过程?
- 如何将Class的实例个数限制为1?

我们一般通过实现以下两点来解决上述问题：

- 隐藏Class的构造函数，避免多次实例化
- 通过暴露一个 getInstance() 方法来创建/获取唯一实例

Javascript中单例模式可以通过以下方式实现：

```
// 单例构造器
const FooServiceSingleton = (function () {
  // 隐藏的Class的构造函数
  function FooService() {}

  // 未初始化的单例对象
  let fooService;

  return {
    // 创建/获取单例对象的函数
    getInstance: function () {
      if (!fooService) {
        fooService = new FooService();
      }
      return fooService;
    }
  }
})();
复制代码
```

实现的关键点有：

1. 使用 IIFE创建局部作用域并即时执行；
2. `getInstance()`为一个 闭包 ， 使用闭包保存局部作用域中的单例对象并返回。

我们可以验证下单例对象是否创建成功：

```
const fooService1 = FooServiceSingleton.getInstance();
const fooService2 = FooServiceSingleton.getInstance();

console.log(fooService1 === fooService2); // true
复制代码
```

场景例子

- 定义命名空间和实现分支型方法
- 登录框
- vuex 和 redux中的store

优点

- 划分命名空间，减少全局变量
- 增强模块性，把自己的代码组织在一个全局变量名下，放在单一位置，便于维护
- 且只会实例化一次。简化了代码的调试和维护

缺点

- 由于单例模式提供的是一种单点访问，所以它有可能导致模块间的强耦合
- 从而不利于单元测试。无法单独测试一个调用了来自单例的方法的类，而只能把它与那个单例作为一
- 个单元一起测试。

5. 策略模式（Strategy Pattern）

策略模式简单描述就是：对象有某个行为，但是在不同的场景中，该行为有不同的实现算法。把它们一个个封装起来，并且使它们可以互相替换

```
<html>
<head>
    <title>策略模式-校验表单</title>
    <meta content="text/html; charset=utf-8" http-equiv="Content-Type">
</head>
<body>
    <form id = "registerForm" method="post"
action="http://xxxx.com/api/register">
        用户名：<input type="text" name="userName">
        密码：<input type="text" name="password">
        手机号码：<input type="text" name="phoneNumber">
        <button type="submit">提交</button>
    </form>
    <script type="text/javascript">
        // 策略对象
        const strategies = {
          isNoEmpty: function (value, errorMsg) {
            if (value === '') {
              return errorMsg;
            }
          },
          isNoSpace: function (value, errorMsg) {
            if (value.trim() === '') {
              return errorMsg;
            }
          },
          minLength: function (value, length, errorMsg) {
```

```
            if (value.trim().length < length) {
              return errorMsg;
            }
          },
          maxLength: function (value, length, errorMsg) {
            if (value.length > length) {
              return errorMsg;
            }
          },
          isMobile: function (value, errorMsg) {
            if (!/^(13[0-
9]|14[5|7]|15[0|1|2|3|5|6|7|8|9]|17[7]|18[0|1|2|3|5|6|7|8|9])\d{8}$/.test(value)
) {
              return errorMsg;
            }
          }
        }

        // 验证类
        class Validator {
          constructor() {
            this.cache = []
          }
          add(dom, rules) {
            for(let i = 0, rule; rule = rules[i++];) {
              let strategyAry = rule.strategy.split(':')
              let errorMsg = rule.errorMsg
              this.cache.push(() => {
                let strategy = strategyAry.shift()
                strategyAry.unshift(dom.value)
                strategyAry.push(errorMsg)
                return strategies[strategy].apply(dom, strategyAry)
              })
            }
          }
          start() {
            for(let i = 0, validatorFunc; validatorFunc = this.cache[i++];) {
              let errorMsg = validatorFunc()
              if (errorMsg) {
                return errorMsg
              }
            }
          }
        }

        // 调用代码
        let registerForm = document.getElementById('registerForm')

        let validataFunc = function() {
          let validator = new Validator()
          validator.add(registerForm.userName, [{
            strategy: 'isNoEmpty',
            errorMsg: '用户名不可为空'
          }, {
            strategy: 'isNoSpace',
            errorMsg: '不允许以空白字符命名'
          }, {
            strategy: 'minLength:2',
```

```
            errorMsg: '用户名长度不能小于2位'
          }])
          validator.add(registerForm.password, [ {
            strategy: 'minLength:6',
            errorMsg: '密码长度不能小于6位'
          }])
          validator.add(registerForm.phoneNumber, [{
            strategy: 'isMobile',
            errorMsg: '请输入正确的手机号码格式'
          }])
          return validator.start()
        }

        registerForm.onsubmit = function() {
          let errorMsg = validataFunc()
          if (errorMsg) {
            alert(errorMsg)
            return false
          }
        }
    </script>
</body>
</html>
```

复制代码

场景例子

- 如果在一个系统里面有许多类，它们之间的区别仅在于它们的'行为'，那么使用策略模式可以动态地让一个对象在许多行为中选择一种行为。
- 一个系统需要动态地在几种算法中选择一种。
- 表单验证

优点

- 利用组合、委托、多态等技术和思想，可以有效的避免多重条件选择语句
- 提供了对开放-封闭原则的完美支持，将算法封装在独立的strategy中，使得它们易于切换，理解，易于扩展
- 利用组合和委托来让Context拥有执行算法的能力，这也是继承的一种更轻便的代替方案

缺点

- 会在程序中增加许多策略类或者策略对象
- 要使用策略模式，必须了解所有的strategy，必须了解各个strategy之间的不同点，这样才能选择一个合适的strategy

6. 迭代器模式（Iterator Pattern）

如果你看到这，ES6中的迭代器 Iterator 相信你还是有点印象的，上面第60条已经做过简单的介绍。迭代器模式简单的说就是提供一种方法顺序一个聚合对象中各个元素，而又不暴露该对象的内部表示。

迭代器模式解决了以下问题：

- 提供一致的遍历各种数据结构的方式，而不用了解数据的内部结构
- 提供遍历容器（集合）的能力而无需改变容器的接口

一个迭代器通常需要实现以下接口：

- hasNext()：判断迭代是否结束，返回Boolean

- next()：查找并返回下一个元素

为Javascript的数组实现一个迭代器可以这么写：

```
const item = [1, 'red', false, 3.14];

function Iterator(items) {
  this.items = items;
  this.index = 0;
}

Iterator.prototype = {
  hasNext: function () {
    return this.index < this.items.length;
  },
  next: function () {
    return this.items[this.index++];
  }
}
```
复制代码

验证一下迭代器是否工作：

```
const iterator = new Iterator(item);

while(iterator.hasNext()){
  console.log(iterator.next());
}
//输出：1, red, false, 3.14
```
复制代码

ES6提供了更简单的迭代循环语法 for...of，使用该语法的前提是操作对象需要实现 可迭代协议（The iterable protocol），简单说就是该对象有个Key为 Symbol.iterator 的方法，该方法返回一个iterator对象。

比如我们实现一个 Range 类用于在某个数字区间进行迭代：

```
function Range(start, end) {
  return {
    [Symbol.iterator]: function () {
      return {
        next() {
          if (start < end) {
            return { value: start++, done: false };
          }
          return { done: true, value: end };
        }
      }
    }
  }
}

```
复制代码

验证一下：

```
for (num of Range(1, 5)) {
  console.log(num);
}
// 输出：1, 2, 3, 4
复制代码
```

7. 观察者模式（Observer Pattern）

观察者模式又称**发布-订阅模式**（Publish/Subscribe Pattern），是我们经常接触到的设计模式，日常生活中的应用也比比皆是，比如你订阅了某个博主的频道，当有内容更新时会收到推送；又比如JavaScript中的事件订阅响应机制。观察者模式的思想用一句话描述就是：**被观察对象（subject）维护一组观察者（observer），当被观察对象状态改变时，通过调用观察者的某个方法将这些变化通知到观察者。**

观察者模式中Subject对象一般需要实现以下API：

- subscribe(): 接收一个观察者observer对象，使其订阅自己
- unsubscribe(): 接收一个观察者observer对象，使其取消订阅自己
- fire(): 触发事件，通知到所有观察者

用JavaScript手动实现观察者模式：

```
// 被观察者
function Subject() {
  this.observers = [];
}

Subject.prototype = {
  // 订阅
  subscribe: function (observer) {
    this.observers.push(observer);
  },
  // 取消订阅
  unsubscribe: function (observerToRemove) {
    this.observers = this.observers.filter(observer => {
      return observer !== observerToRemove;
    })
  },
  // 事件触发
  fire: function () {
    this.observers.forEach(observer => {
      observer.call();
    });
  }
}
复制代码
```

验证一下订阅是否成功：

```
const subject = new Subject();

function observer1() {
  console.log('Observer 1 Firing!');
}
```

```
function observer2() {
  console.log('Observer 2 Firing!');
}

subject.subscribe(observer1);
subject.subscribe(observer2);
subject.fire();

//输出：
Observer 1 Firing!
Observer 2 Firing!
复制代码
```

验证一下取消订阅是否成功：

```
subject.unsubscribe(observer2);
subject.fire();

//输出：
Observer 1 Firing!
复制代码
```

场景

- DOM事件

```
document.body.addEventListener('click', function() {
    console.log('hello world!');
});
document.body.click()
复制代码
```

- vue 响应式

优点

- 支持简单的广播通信，自动通知所有已经订阅过的对象
- 目标对象与观察者之间的抽象耦合关系能单独扩展以及重用
- 增加了灵活性
- 观察者模式所做的工作就是在解耦，让耦合的双方都依赖于抽象，而不是依赖于具体。从而使得各自的变化都不会影响到另一边的变化。

缺点

- 过度使用会导致对象与对象之间的联系弱化，会导致程序难以跟踪维护和理解

8. 中介者模式（Mediator Pattern）

在中介者模式中，中介者（Mediator）包装了一系列对象相互作用的方式，使得这些对象不必直接相互作用，而是由中介者协调它们之间的交互，从而使它们可以松散偶合。当某些对象之间的作用发生改变时，不会立即影响其他的一些对象之间的作用，保证这些作用可以彼此独立的变化。

中介者模式和观察者模式有一定的相似性，都是一对多的关系，也都是集中式通信，不同的是中介者模式是处理同级对象之间的交互，而观察者模式是处理Observer和Subject之间的交互。中介者模式有些像婚恋中介，相亲对象刚开始并不能直接交流，而是要通过中介去筛选匹配再决定谁和谁见面。

场景

- 例如购物车需求，存在商品选择表单、颜色选择表单、购买数量表单等等，都会触发change事件，那么可以通过中介者来转发处理这些事件，实现各个事件间的解耦，仅仅维护中介者对象即可。

```
var goods = {   //手机库存
    'red|32G': 3,
    'red|64G': 1,
    'blue|32G': 7,
    'blue|32G': 6,
};
//中介者
var mediator = (function() {
    var colorSelect = document.getElementById('colorSelect');
    var memorySelect = document.getElementById('memorySelect');
    var numSelect = document.getElementById('numSelect');
    return {
        changed: function(obj) {
            switch(obj){
                case colorSelect:
                    //TODO
                    break;
                case memorySelect:
                    //TODO
                    break;
                case numSelect:
                    //TODO
                    break;
            }
        }
    }
})();
colorSelect.onchange = function() {
    mediator.changed(this);
};
memorySelect.onchange = function() {
    mediator.changed(this);
};
numSelect.onchange = function() {
    mediator.changed(this);
};
复制代码
```

- 聊天室里

聊天室成员类：

```
function Member(name) {
  this.name = name;
  this.chatroom = null;
}

Member.prototype = {
  // 发送消息
  send: function (message, toMember) {
    this.chatroom.send(message, this, toMember);
  },
  // 接收消息
```

```
  receive: function (message, fromMember) {
    console.log(`${fromMember.name} to ${this.name}: ${message}`);
  }
}
```
复制代码

聊天室类：

```
function Chatroom() {
  this.members = {};
}

Chatroom.prototype = {
  // 增加成员
  addMember: function (member) {
    this.members[member.name] = member;
    member.chatroom = this;
  },
  // 发送消息
  send: function (message, fromMember, toMember) {
    toMember.receive(message, fromMember);
  }
}
```
复制代码

测试一下：

```
const chatroom = new Chatroom();
const bruce = new Member('bruce');
const frank = new Member('frank');

chatroom.addMember(bruce);
chatroom.addMember(frank);

bruce.send('Hey frank', frank);

//输出：bruce to frank: hello frank
```
复制代码

优点

- 使各对象之间耦合松散，而且可以独立地改变它们之间的交互
- 中介者和对象一对多的关系取代了对象之间的网状多对多的关系
- 如果对象之间的复杂耦合度导致维护很困难，而且耦合度随项目变化增速很快，就需要中介者重构代码

缺点

- 系统中会新增一个中介者对象，因为对象之间交互的复杂性，转移成了中介者对象的复杂性，使得中介者对象经常是巨大的。中介 者对象自身往往就是一个难以维护的对象。

9. 访问者模式（Visitor Pattern）

访问者模式 是一种将算法与对象结构分离的设计模式，通俗点讲就是：访问者模式让我们能够在不改变一个对象结构的前提下能够给该对象增加新的逻辑，新增的逻辑保存在一个独立的访问者对象中。访问者模式常用于拓展一些第三方的库和工具。

```
// 访问者
class Visitor {
    constructor() {}
    visitConcreteElement(ConcreteElement) {
        ConcreteElement.operation()
    }
}
// 元素类
class ConcreteElement{
    constructor() {
    }
    operation() {
       console.log("ConcreteElement.operation invoked");
    }
    accept(visitor) {
        visitor.visitConcreteElement(this)
    }
}
// client
let visitor = new Visitor()
let element = new ConcreteElement()
elementA.accept(visitor)
```
复制代码

访问者模式的实现有以下几个要素：

- Visitor Object：访问者对象，拥有一个`visit()`方法
- Receiving Object：接收对象，拥有一个`accept()`方法
- visit(receivingObj)：用于Visitor接收一个`Receiving Object`
- accept(visitor)：用于`Receving Object`接收一个Visitor，并通过调用`Visitor`的`visit()`为其提供获取`Receiving Object`数据的能力

简单的代码实现如下：

```
Receiving Object：

function Employee(name, salary) {
  this.name = name;
  this.salary = salary;
}

Employee.prototype = {
  getSalary: function () {
    return this.salary;
  },
  setSalary: function (salary) {
    this.salary = salary;
  },
  accept: function (visitor) {
    visitor.visit(this);
  }
}
Visitor Object：

function Visitor() { }
```

```
Visitor.prototype = {
  visit: function (employee) {
    employee.setSalary(employee.getSalary() * 2);
  }
}
```
复制代码

验证一下：

```
const employee = new Employee('bruce', 1000);
const visitor = new Visitor();
employee.accept(visitor);

console.log(employee.getSalary());//输出：2000
```
复制代码

场景

- 对象结构中对象对应的类很少改变，但经常需要在此对象结构上定义新的操作
- 需要对一个对象结构中的对象进行很多不同的并且不相关的操作，而需要避免让这些操作"污染"这些对象的类，也不希望在增加新操作时修改这些类。

优点

- 符合单一职责原则
- 优秀的扩展性
- 灵活性

缺点

- 具体元素对访问者公布细节，违反了迪米特原则
- 违反了依赖倒置原则，依赖了具体类，没有依赖抽象。
- 具体元素变更比较困难