JavaScript 面试知识点总结

本部分主要是笔者在复习 JavaScript 相关知识和一些相关面试题时所做的笔记，如果出现错误，希望大家指出！

目录

1. 介绍 js 的基本数据类型。

`js` 一共有七种基本数据类型，分别是 `Undefined`、`Null`、`Boolean`、`Number`、`String`，还有在 `ES6` 中新增的 `Symbol` 和 `ES10` 中新增的 `BigInt` 类型。
`Symbol` 代表创建后独一无二且不可变的数据类型，它的出现我认为主要是为了解决可能出现的全局变量冲突的问题。
`BigInt` 是一种数字类型的数据，它可以表示任意精度格式的整数，使用 `BigInt` 可以安全地存储和操作大整数，即使这个数已经超出了 `Number` 能够表示的安全整数范围。

2. JavaScript 有几种类型的值？你能画一下他们的内存图吗?

涉及知识点:

- 栈：原始数据类型（Undefined、Null、Boolean、Number、String）
- 堆：引用数据类型（对象、数组和函数）

两种类型的区别是：存储位置不同。
原始数据类型直接存储在栈（stack）中的简单数据段，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。

引用数据类型存储在堆（heap）中的对象，占据空间大、大小不固定。如果存储在栈中，将会影响程序运行的性能；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。

回答：

js 可以分为两种类型的值，一种是基本数据类型，一种是引用数据类型。

基本数据类型....（参考1）

引用数据类型指的是 Object 类型，所有其他的如 Array、Date 等数据类型都可以理解为 Object 类型的子类。

两种类型间的主要区别是它们的存储位置不同，基本数据类型的值直接保存在栈中，而复杂数据类型的值保存在堆中，通过使用在栈中
保存对应的指针来获取堆中的值。

详细资料可以参考：
《JavaScript 有几种类型的值？》
《JavaScript 有几种类型的值？能否画一下它们的内存图；》

3. 什么是堆？什么是栈？它们之间有什么区别和联系？

堆和栈的概念存在于数据结构中和操作系统内存中。

在数据结构中，栈中数据的存取方式为先进后出。而堆是一个优先队列，是按优先级来进行排序的，优先级可以按照大小来规定。完全
二叉树是堆的一种实现方式。

在操作系统中，内存被分为栈区和堆区。

栈区内存由编译器自动分配释放，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈。

堆区内存一般由程序员分配释放，若程序员不释放，程序结束时可能由垃圾回收机制回收。

详细资料可以参考：
《什么是堆？什么是栈？他们之间有什么区别和联系？》

4. 内部属性 [[Class]] 是什么？

所有 typeof 返回值为 "object" 的对象（如数组）都包含一个内部属性 [[Class]]（我们可以把它看作一个内部的分类，而非
传统的面向对象意义上的类）。这个属性无法直接访问，一般通过 Object.prototype.toString(..) 来查看。例如：

```
Object.prototype.toString.call( [1,2,3] );
// "[object Array]"

Object.prototype.toString.call( /regex-literal/i );
```

```
// "[object RegExp]"

// 我们自己创建的类就不会有这份特殊待遇，因为 toString() 找不到 toStringTag 属性时只好返回默认的 Object 标签
// 默认情况类的[[Class]]返回[object Object]
class Class1 {}
Object.prototype.toString.call(new Class1()); // "[object Object]"
// 需要定制[[Class]]
class Class2 {
  get [Symbol.toStringTag]() {
    return "Class2";
  }
}
Object.prototype.toString.call(new Class2()); // "[object Class2]"
```

5. 介绍 js 有哪些内置对象?

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

回答:

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。

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 的。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 。
- 并且typeof 对 undefined 和 undeclared 变量返回的都是undefined。其实“ undefined” 和“ is not defined ”是两码事。

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。一般变量声明了但还没有赋值的时候会返回 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。

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。

```
null == undefined    // true
null === undefined   // false    数据类型不同
```

详细资料可以参考：
《JavaScript 深入理解之 undefined 与 null》

8. 如何获取安全的 undefined 值?

因为 undefined 是一个标识符，所以可以被当作变量来使用和赋值，但是这样会影响 undefined 的正常判断。

表达式 void ___ 没有返回值，因此返回结果是 undefined。void 并不改变表达式的结果，只是让表达式不返回值。

- 按惯例我们用 void 0 来获得 undefined。
- 除了防止被重写外，还可以减少字节。void 0 代替 undefined省3个字节。（老司机常用）

9. 说几条写 JavaScript 的基本规范?

在平常项目开发中，我们遵守一些这样的基本规范，比如说：

（1）一个函数作用域中所有的变量声明应该尽量提到函数首部，用一个 var 声明，不允许出现两个连续的 var 声明，声明时
    如果变量没有值，应该给该变量赋值对应类型的初始值，便于他人阅读代码时，能够一目了然的知道变量对应的类型值。

（2）代码中出现地址、时间等字符串时需要使用常量代替。

（3）在进行比较的时候吧，尽量使用'===', '!=='代替'==', '!='。

（4）不要在内置对象的原型上添加方法，如 Array, Date。

（5）switch 语句必须带有 default 分支。

（6）for 循环必须使用大括号。

（7）if 语句必须使用大括号。

10. JavaScript 原型，原型链？ 有什么特点?

```
1). 什么是原型对象:
    1. 每个函数都有一个prototype属性，该属性指向的是原型对象(显示原型对象)
    2. 每个实例对象身上都有一个__proto__属性，该属性指向的也是原型对象(隐式原型对象)
    3. 构造函数的显示原型 === 当前构造函数实例对象的隐式原型对象
    4. 原型对象的本质： 普通的Object实例
2). 什么是原型链:
    1. 查找对象的属性的时候现在自身找，如果自身没有沿着__proto__找原型对象
    2. 如果原型对象上还没有，继续沿着__proto__,直到找到Object的原型对象
    3. 如果还没有找到返回undefined
    4. 原型链： 沿着__proto__查找属性(方法)的这条链就是原型链
```

详细资料可以参考:
《JavaScript 深入理解之原型与原型链》

11. js 获取原型的方法?

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

```
function R(){
}
var one=new R();
console.log(Object.getPrototypeOf(one));    //官方推荐，规范写法
console.log(one.proto);                      //不报错，不推荐
console.log(one.constructor.proto)  //同上
```

12. 在 js 中不同进制数字的表示方式

- 以 0X、0x 开头的表示为十六进制。
- 以 0、0O、0o 开头的表示为八进制。
- 以 0B、0b 开头的表示为二进制格式。

13. js 中整数的安全范围是多少?

```
安全整数指的是，在这个范围内的整数转化为二进制存储的时候不会出现精度丢失，能够被“安全”呈现的最大整数是 2^53 - 1，
即9007199254740991，在 ES6 中被定义为 Number.MAX_SAFE_INTEGER。最小整数是-9007199254740991，在 ES6 中
被定义为 Number.MIN_SAFE_INTEGER。

如果某次计算的结果得到了一个超过 JavaScript 数值范围的值，那么这个值会被自动转换为特殊的 Infinity 值。如果某次
计算返回了正或负的 Infinity 值，那么该值将无法参与下一次的计算。判断一个数是不是有穷的，可以使用 isFinite 函数
来判断。
```

14. typeof NaN 的结果是什么?

NaN 意指“不是一个数字”（not a number），NaN 是一个“警戒值”（sentinel value，有特殊用途的常规值），用于指出
数字类型中的错误情况，即“执行数学运算没有成功，这是失败后返回的结果”。

```
typeof NaN; // "number"
```

NaN 是一个特殊值，它和自身不相等，是唯一一个非自反（自反，reflexive，即 x === x 不成立）的值。而 NaN != NaN
为 true。

15. isNaN 和 Number.isNaN 函数的区别?

函数 isNaN 接收参数后，会尝试将这个参数转换为数值，任何不能被转换为数值的的值都会返回 true，因此非数字值传入也会
返回 true ，会影响 NaN 的判断。

函数 Number.isNaN 会首先判断传入参数是否为数字，如果是数字再继续判断是否为 NaN ，这种方法对于 NaN 的判断更为
准确。

16. Array 构造函数只有一个参数值时的表现?

Array 构造函数只带一个数字参数的时候，该参数会被作为数组的预设长度（length），而非只充当数组中的一个元素。这样
创建出来的只是一个空数组，只不过它的 length 属性被设置成了指定的值。

构造函数 Array(..) 不要求必须带 new 关键字。不带时，它会被自动补上。

17. 其他值到字符串的转换规则?

规范的 9.8 节中定义了抽象操作 ToString ，它负责处理非字符串到字符串的强制类型转换。

（1）Null 和 Undefined 类型 ，null 转换为 "null"，undefined 转换为 "undefined"，

（2）Boolean 类型，true 转换为 "true"，false 转换为 "false"。

（3）Number 类型的值直接转换，不过那些极小和极大的数字会使用指数形式。

（4）Symbol 类型的值直接转换，但是只允许显式强制类型转换，使用隐式强制类型转换会产生错误。

（5）对普通对象来说，除非自行定义 toString() 方法，否则会调用 toString()（Object.prototype.toString()）
来返回内部属性 [[Class]] 的值，如"[object Object]"。如果对象有自己的 toString() 方法，字符串化时就会
调用该方法并使用其返回值。

18. 其他值到数字值的转换规则?

有时我们需要将非数字值当作数字来使用，比如数学运算。为此 ES5 规范在 9.3 节定义了抽象操作 ToNumber。

（1）Undefined 类型的值转换为 NaN。

（2）Null 类型的值转换为 0。

（3）Boolean 类型的值，true 转换为 1，false 转换为 0。

（4）String 类型的值转换如同使用 Number() 函数进行转换，如果包含非数字值则转换为 NaN，空字符串为 0。

（5）Symbol 类型的值不能转换为数字，会报错。

（6）对象（包括数组）会首先被转换为相应的基本类型值，如果返回的是非数字的基本类型值，则再遵循以上规则将其强制转换为数字。

为了将值转换为相应的基本类型值，抽象操作 ToPrimitive 会首先（通过内部操作 DefaultValue）检查该值是否有valueOf() 方法。如果有并且返回基本类型值，就使用该值进行强制类型转换。如果没有就使用 toString() 的返回值（如果存在）来进行强制类型转换。

如果 valueOf() 和 toString() 均不返回基本类型值，会产生 TypeError 错误。

19. 其他值到布尔类型的值的转换规则?

ES5 规范 9.2 节中定义了抽象操作 ToBoolean，列举了布尔强制类型转换所有可能出现的结果。

以下这些是假值：
- undefined
- null
- false
- +0、-0 和 NaN
- ""(空串)

假值的布尔强制类型转换结果为 false。从逻辑上说，假值列表以外的都应该是真值。

20. {} 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 "" （空串）

21. 什么是假值对象?

浏览器在某些特定情况下，在常规 JavaScript 语法基础上自己创建了一些外来值，这些就是“假值对象”。
假值对象看起来和
普通对象并无二致（都有属性，等等），但将它们强制类型转换为布尔值时结果为 false 最常见的例子是 document.all，它
是一个类数组对象，包含了页面上的所有元素，由 DOM（而不是 JavaScript 引擎）提供给 JavaScript 程序使用。

22. ~ 操作符的作用?

这是js中的一元操作符：按位取反。
~ 返回 2 的补码，并且 ~ 会将数字转换为 32 位整数，因此我们可以使用 ~ 来进行取整操作。

~x 大致等同于 -(x+1)。

23. 解析字符串中的数字和将字符串强制类型转换为数字的返回结果都是数字，它们之间的区别是什么?

解析允许字符串（如 parseInt() ）中含有非数字字符，解析按从左到右的顺序，如果遇到非数字字符就停止。而转换（如 Nu
mber ()）不允许出现非数字字符，否则会失败并返回 NaN。

24. + 操作符什么时候用于字符串的拼接?

根据 ES5 规范 11.6.1 节，如果某个操作数是字符串或者能够通过以下步骤转换为字符串的话，+ 将进行拼接操作。如果其
中一个操作数是对象（包括数组），则首先对其调用 ToPrimitive 抽象操作，该抽象操作再调用 [[DefaultValue]]，以
数字作为上下文。如果不能转换为字符串，则会将其转换为数字类型来进行计算。

简单来说就是，如果 + 的其中一个操作数是字符串（或者通过以上步骤最终得到字符串），则执行字符串拼接，否则执行数字
加法。

那么对于除了加法的运算符来说，只要其中一方是数字，那么另一方就会被转为数字。

25. 什么情况下会发生布尔值的隐式强制类型转换?

（1） if (..) 语句中的条件判断表达式。
（2） for (.. ; .. ; ..) 语句中的条件判断表达式（第二个）。
（3） while (..) 和 do..while(..) 循环中的条件判断表达式。
（4） ? : 中的条件判断表达式。
（5） 逻辑运算符 ||（逻辑或）和 &&（逻辑与）左边的操作数（作为条件判断表达式）。

26. || 和 && 操作符的返回值?

|| 和 && 首先会对第一个操作数执行条件判断，如果其不是布尔值就先进行 ToBoolean 强制类型转换，然后再执行条件
判断。

对于 || 来说，如果条件判断结果为 true 就返回第一个操作数的值，如果为 false 就返回第二个操作数的值。

&& 则相反，如果条件判断结果为 true 就返回第二个操作数的值，如果为 false 就返回第一个操作数的值。

|| 和 && 返回它们其中一个操作数的值，而非条件判断的结果

27. Symbol 值的强制类型转换?

ES6 允许从符号到字符串的显式强制类型转换，然而隐式强制类型转换会产生错误。

Symbol 值不能够被强制类型转换为数字（显式和隐式都会产生错误），但可以被强制类型转换为布尔值（显式和隐式结果
都是 true ）。

28. == 操作符的强制类型转换规则?

（1）字符串和数字之间的相等比较，将字符串转换为数字之后再进行比较。

（2）其他类型和布尔类型之间的相等比较，先将布尔值转换为数字后，再应用其他规则进行比较。

（3）null 和 undefined 之间的相等比较，结果为真。其他值和它们进行比较都返回假值。

（4）对象和非对象之间的相等比较，对象先调用 ToPrimitive 抽象操作后，再进行比较。

（5）如果一个操作值为 NaN ，则相等比较返回 false（ NaN 本身也不等于 NaN ）。

（6）如果两个操作值都是对象，则比较它们是不是指向同一个对象。如果两个操作数都指向同一个对象，则相等操作符返回 true，否则，返回 false。

详细资料可以参考：
《JavaScript 字符串间的比较》

29. 如何将字符串转化为数字，例如 '12.3b'?

（1）使用 Number() 方法，前提是所包含的字符串不包含不合法字符。

（2）使用 parseInt() 方法，parseInt() 函数可解析一个字符串，并返回一个整数。还可以设置要解析的数字的基数。当基数的值为 0，或没有设置该参数时，parseInt() 会根据 string 来判断数字的基数。

（3）使用 parseFloat() 方法，该函数解析一个字符串参数并返回一个浮点数。

（4）使用 + 操作符的隐式转换。

详细资料可以参考：
《详解 JS 中 Number()、parseInt() 和 parseFloat() 的区别》

30. 如何将浮点数点左边的数每三位添加一个逗号，如 12000000.11 转化为『12,000,000.11』?

```
// 方法一
function format(number) {
    // ?=这表示一个环视的语法，表示该位置后面的字符指定规则
  return number && number.toString().replace(/(\d)(?=(\d{3})+\.)/g, function($1,
$2) {
         return $2 + ',';
       })
}
// 方法二
function format1(number) {
  return Intl.NumberFormat().format(number)
}
// 方法三
function format2(number) {
  return number.toLocaleString('en')   // 换成当地的字符串格式
}
```

31. 常用正则表达式

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

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

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

// （4）手机号码正则
var regex = /^1[345789]\d{9}$/g;

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

详细资料可以参考：
《前端表单验证常用的 15 个 JS 正则表达式》
《JS 常用正则汇总》

32. 生成随机数的各种方法?

```
Math.random()    生成0-1（包括0不包括1的随机数）
```

33. 如何实现数组的随机排序?

```
// （1）使用数组 sort 方法对数组元素随机排序，让 Math.random() 出来的数与 0.5 比较，如果大于就返回 1 交换位置，如果小于就返回 -1，不交换位置。

function randomSort(arr) {
  return arr.sort((a, b) =>  return Math.random() > 0.5 ? -1 : 1;  )
}

//  缺点：每个元素被派到新数组的位置不是随机的，原因是 sort() 方法是依次比较的。

// （2）随机从原数组抽取一个元素，加入到新数组

function randomSort(arr) {
  var result = [];

  while (arr.length > 0) {
    var randomIndex = Math.floor(Math.random() * arr.length);
    result.push(arr[randomIndex]);
    arr.splice(randomIndex, 1);    // 删除原数组中随机下标位置的值
  }

  return result;
}

// （3）随机交换数组内的元素（洗牌算法类似）

function randomSort(arr) {
  var index,
```

```
    randomIndex,
    temp,
    len = arr.length;

  for (index = 0; index < len; index++) {
      // 向下取整（最大 - 最小 + 1） + 最小 （+1） // 避免自己跟自己交换
    randomIndex = Math.floor(Math.random() * (len - index)) + index (+ 1) ;

    temp = arr[index];
    arr[index] = arr[randomIndex];
    arr[randomIndex] = temp;
  }

  return arr;
}

// es6
function randomSort(array) {
  let length = array.length;

  // 不是数组或者数组长度小于等于1，直接返回
  if (!Array.isArray(array) || length <= 1) return;

  for (let index = 0; index < length - 1; index++) {
    let randomIndex = Math.floor(Math.random() * (length - index)) + index;

    [array[index], array[randomIndex]] = [array[randomIndex], array[index]];
  }

  return array;
}
```

详细资料可以参考：
《Fisher and Yates 的原始版》
《javascript 实现数组随机排序?》
《JavaScript 学习笔记：数组随机排序》

34. 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。

```
    // 工厂函数：返回一个需要的数据的函数
  function Person(name, age) {
    return {
      name: name,
      age: age
    }
  }
  var person1 = Person('kobe', 43);
```

（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。

详细资料可以参考：
《JavaScript 深入理解之对象创建》

35. 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。

详细资料可以参考：
《JavaScript 深入理解之继承》

36. 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);  // 子类的原型等价于父类的实例，指向父类的原型
Student.prototype.constructor = Student;

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

37. Javascript 的作用域链?

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》

38. 谈谈 This 对象的理解。

this 是执行上下文中的一个属性，它指向最后一次调用这个方法的对象。在实际开发中，this 的指向可以通过四种调用模
式来判断。

- 1.第一种是函数调用模式，当一个函数不是一个对象的属性时，直接作为函数来调用时，this 指向全局对象。
- 2.第二种是方法调用模式，如果一个函数作为一个对象的方法来调用时，this 指向这个对象。
- 3.第三种是构造器调用模式，如果一个函数用 new 调用时，函数执行前会新创建一个对象，this 指向这个新创建的对象。
- 4.第四种是 apply 、 call 和 bind 调用模式，这三个方法都可以显示的指定调用函数的 this 指向。其中 apply 方法接收两个参数：一个是 this 绑定的对象，一个是参数数组。call 方法接收的参数，第一个是 this 绑定的对象，后面的其余参数是传入函数执行的参数。也就是说，在使用 call() 方法时，传递给函数的参数必须逐个列举出来。bind 方法通过传入一个对象，返回一个 this 绑定了传入对象的新函数。这个函数的 this 指向除了使用 new 时会被改变，其他情况下都不会改变。

这四种方式，使用构造器调用模式的优先级最高，然后是 apply 、 call 和 bind 调用模式，然后是方法调用模式，然后
是函数调用模式。

《JavaScript 深入理解之 this 详解》

39. eval 是做什么的?

它的功能是把对应的字符串解析成 JS 代码并运行。

应该避免使用 eval，不安全，非常耗性能（2次，一次解析成 js 语句，一次执行）。

详细资料可以参考：
《eval()》

40. 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方法和接口。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、history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。

详细资料可以参考：
《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》

41. 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;
    }
  }
};
```

详细资料可以参考：
《JS 事件模型》

42. 事件是什么？IE 与火狐的事件机制有什么区别？ 如何阻止冒泡？

- 1.事件是用户操作网页时发生的交互动作，比如 click/move， 事件除了用户触发的动作外，还可以是文档加载，窗口滚动和大小调整。事件被封装成一个 event 对象，包含了该事件发生时的所有相关信息（ event 的属性）以及可以对事件进行的操作（ event 的方法）。
- 2.事件处理机制：IE 支持事件冒泡、Firefox 同时支持两种事件模型，也就是：事件冒泡和事件捕获。
- 3.event.stopPropagation() 或者 ie 下的方法 event.cancelBubble = true;

详细资料可以参考：
《Javascript 事件模型系列（一）事件及事件的三种模型》
《Javascript 事件模型：事件捕获和事件冒泡》

43. 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。

第一种事件模型是最早的 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。

第二种事件模型是 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。

第三种是 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（默认值为false）。

详细资料可以参考：
《一个 DOM 元素绑定多个事件时，先执行冒泡还是捕获》

44. 事件委派是什么?

事件委派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，这种方式称为事件委派。

使用事件委派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。并且使用事件委派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。

详细资料可以参考：
《JavaScript 事件委托详解》

45. ["1", "2", "3"].map(parseInt) 答案是多少?

`parseInt()` 函数能解析一个字符串，并返回一个整数，需要两个参数 `(val, radix)`，其中 `radix` 表示要解析的数字的基数(即按什么进制来解析)。（该值介于 `2 ~ 36` 之间，如果省略该参数或其值为 `‘0‘`，则数字将以 `10` 进制来解析。如果该参数小于 `2` 或者大于 `36`，则 `‘parseInt()‘` 将返回 `‘NaN‘`）。

此处 `map` 传了 `3` 个参数 `(element, index, array)`，默认第三个参数被忽略掉，因此三次传入的参数分别为 `"1-0", "2-1", "3-2"`

因为字符串的值不能大于进制数，因此后面两次调用均失败，返回 `NaN` ，第一次基数为 `0` ，按十进制解析返回 `1`。、

```
// 1  NaN  NaN
```

详细资料可以参考：
《为什么 ["1", "2", "3"].map(parseInt) 返回 [1,NaN,NaN]？》

46. 什么是闭包，为什么要用它?

什么是闭包：
- 闭包是一个存在内部函数的引用关系
- 该引用指向的是外部函数的局部变量对象(前提是内部函数使用了外部函数的局部变量)

闭包的作用：
- 延长外部函数变量对象的生命周期
- 使用闭包能够间接的从函数外部访问函数内部的私有变量

详细资料可以参考：
《JavaScript 深入理解之闭包》

47. javascript 代码中的 "use strict"; 是什么意思？使用它区别是什么?

相关知识点：

use strict 是一种 ECMAscript5 添加的（严格）运行模式，这种模式使得 Javascript 在更严格的条件下运行。

设立"严格模式"的目的，主要有以下几个：

- 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。

区别：

- 1.声明定义变量必须用var
- 2.禁止自定义的函数中的this关键字指向全局对象
- 创建eval作用域, 更安全

回答：

use strict 指的是严格运行模式，在这种模式对 js 的使用添加了一些限制。比如说禁止 this 指向全局对象，还有声明定义
变量必须用var等。设立严格模式的目的，主要是为了消除代码使用中的一些不安全的使用方式，也是为了消除 js 语法本身的一
些不合理的地方，以此来减少一些运行时的怪异的行为。同时使用严格运行模式也能够提高编译的效率，从而提高代码的运行速度。
我认为严格模式代表了 js 一种更合理、更安全、更严谨的发展方向。

详细资料可以参考：
《Javascript 严格模式详解》

48. 如何判断一个对象是否属于某个类?

第一种方式是使用 A instanceof B 运算符来判断B构造函数的 prototype 属性是否出现在A对象的原型链中的任何位置。

第二种方式，如果需要判断的是某个内置的引用类型的话，可以使用 Object.prototype.toString() 方法来打印对象的
[[Class]] 属性来进行判断。

详细资料可以参考：
《js 判断一个对象是否属于某一类》

49. instanceof 的作用?

```
// instanceof 运算符用于判断构造函数的 prototype 属性是否出现在对象的原型链中的任何位置。
// 实现：

function myInstanceof(left, right) {
  let proto = Object.getPrototypeOf(left), // 获取对象的原型
    prototype = right.prototype; // 获取构造函数的 prototype 对象

  // 判断构造函数的 prototype 对象是否在对象的原型链上
  while (true) {
    if (!proto) return false;
    if (proto === prototype) return true;
```

```
    proto = Object.getPrototypeOf(proto);
  }
}
```

详细资料可以参考：
《instanceof》

50. new 操作符具体干了什么呢？如何实现？

```
// （1）首先创建了一个新的空对象
// （2）设置原型，将对象的原型设置为函数的 prototype 对象。
// （3）让函数的 this 指向这个对象，执行构造函数的代码（为这个新对象添加属性）
// （4）判断函数的返回值类型，如果是值类型，返回创建的对象。如果是引用类型，就返回这个引用类型的对象。

// 实现：

function objectFactory() {
  let newObject = null,
    constructor = Array.prototype.shift.call(arguments),  // 拿到构造函数
    result = null;

  // 参数判断
  if (typeof constructor !== "function") {
    console.error("type error");
    return;
  }

  // 新建一个空对象，对象的原型为构造函数的 prototype 对象
  newObject = Object.create(constructor.prototype);

  // 将 this 指向新建对象，并执行函数
  result = constructor.apply(newObject, arguments);

  // 判断返回对象
  let flag =
    result && (typeof result === "object" || typeof result === "function");

  // 判断返回结果
  return flag ? result : newObject;
}

// 使用方法
// objectFactory(构造函数, 初始化参数);
```

详细资料可以参考：
《new 操作符具体干了什么？》
《JavaScript 深入之 new 的模拟实现》

51. Javascript 中，有一个函数，执行对象查找时，永远不会去查找原型，这个函数是?

hasOwnProperty

所有继承了 Object 的对象都会继承到 hasOwnProperty 方法。这个方法可以用来检测一个对象是否含有特定的自身属性，和
in 运算符不同，该方法会忽略掉那些从原型链上继承到的属性。

详细资料可以参考：
《Object.prototype.hasOwnProperty()》

52. 对于 JSON 的了解?

相关知识点：

JSON 是一种数据交换格式，基于文本，优于轻量，用于交换数据。

JSON 可以表示数字、布尔值、字符串、null、数组（值的有序序列），以及由这些值（或数组、对象）所组成的对象（字符串与
值的映射）。

JSON 使用 JavaScript 语法，但是 JSON 格式仅仅是一个文本。文本可以被任何编程语言读取及作为数据格式传递。

回答：

JSON 是一种基于文本的轻量级的数据交换格式。它可以被任何的编程语言读取和作为数据格式来传递。

在项目开发中，我们使用 JSON 作为前后端数据交换的方式。在前端我们通过将一个符合 JSON 格式的数据结构序列化为 JSON 字符串，然后将它传递到后端，后端通过 JSON 格式的字符串解析后生成对应的数据结构，以此来实现前后端数据的一个传递。

因为 JSON 的语法是基于 js 的，因此很容易将 JSON 和 js 中的对象弄混，但是我们应该注意的是 JSON 和 js 中的对象不是一回事，JSON 中对象格式更加严格，比如说在 JSON 中属性值不能为函数，不能出现 NaN 这样的属性值等，因此大多数的 js 对象是不符合 JSON 对象的格式的。

在 js 中提供了两个函数来实现 js 数据结构和 JSON 格式的转换处理，一个是 JSON.stringify 函数，通过传入一个符合 JSON 格式的数据结构，将其转换为一个 JSON 字符串。如果传入的数据结构不符合 JSON 格式，那么在序列化的时候会对这些值进行对应的特殊处理，使其符合规范。在前端向后端发送数据时，我们可以调用这个函数将数据对象转化为 JSON 格式的字符串。

另一个函数 JSON.parse() 函数，这个函数用来将 JSON 格式的字符串转换为一个 js 数据结构，如果传入的字符串不是标准的 JSON 格式的字符串的话，将会抛出错误。当我们从后端接收到 JSON 格式的字符串时，我们可以通过这个方法来将其解析为一个 js 数据结构，以此来进行数据的访问。

详细资料可以参考：
《深入了解 JavaScript 中的 JSON 》

53. [].forEach.call($$(""),*function(a){a.style.outline="1px solid #"+(~~(Math.random()*(1<<24))).toString(16)}*) 能解释一下这段代码的意思吗?

（1）选取页面所有 DOM 元素。在浏览器的控制台中可以使用$$()方法来获取页面中相应的元素，这是现代浏览器提供的一个命令行 API 相当于 document.querySelectorAll 方法。

（2）循环遍历 DOM 元素

（3）给元素添加 outline 。由于渲染的 outline 是不在 CSS 盒模型中的，所以为元素添加 outline 并不会影响元素的大小和页面的布局。

（4）生成随机颜色函数。Math.random()*(1<<24) 可以得到 0~2^24 - 1 之间的随机数，因为得到的是一个浮点数，但我们只需要整数部分，使用取反操作符 ~ 连续两次取反获得整数部分，然后再用 toString(16) 的方式，转换为一个十六进制的字符串。

详细资料可以参考：
《通过一行代码学 JavaScript》

54. js 延迟加载的方式有哪些?

相关知识点：

js 延迟加载，也就是等页面加载完成之后再加载 JavaScript 文件。 js 延迟加载有助于提高页面加载速度。

一般有以下几种方式：

- defer 属性
- async 属性
- 动态创建 DOM 方式
- 使用 setTimeout 延迟方法
- 让 JS 最后加载

回答：

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：

第一种方式是我们一般采用的是将 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。

第二种方式是给 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。

第三种方式是给 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。

第四种方式是动态创建 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。

详细资料可以参考：
《JS 延迟加载的几种方式》
《HTML 5 <script> async 属性》

55. Ajax 是什么? 如何创建一个 Ajax?

相关知识点:

2005 年 2 月，AJAX 这个词第一次正式提出，它是 Asynchronous JavaScript and XML 的缩写，指的是通过 JavaScript 的
异步通信，从服务器获取 XML 文档从中提取数据，再更新当前网页的对应部分，而不用刷新整个网页。

具体来说，AJAX 包括以下几个步骤。

- 1.创建 XMLHttpRequest 对象，也就是创建一个异步调用对象
- 2.创建一个新的 HTTP 请求，并指定该 HTTP 请求的方法、URL 及验证信息
- 3.设置响应 HTTP 请求状态变化的函数
- 4.发送 HTTP 请求
- 5.获取异步调用返回的数据
- 6.使用 JavaScript 和 DOM 实现局部刷新

一般实现:

```
const SERVER_URL = "/server";

let xhr = new XMLHttpRequest();

// 创建 Http 请求
xhr.open("GET", SERVER_URL, true);

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

  // 当请求成功时
  if (this.status === 200) {
    handle(this.response);
  } else {
    console.error(this.statusText);
  }
};

// 设置请求失败时的监听函数
xhr.onerror = function() {
  console.error(this.statusText);
};

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

// 发送 Http 请求
xhr.send(null);

// 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;
}
```

回答:

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。

创建一个 ajax 有这样几个步骤

首先是创建一个 XMLHttpRequest 对象。

然后在这个对象上使用 open 方法创建一个 http 请求，open 方法所需要的参数是请求的方法、请求的地址、是否异步和用户的认证信息。

在发起请求前，我们可以为这个对象添加一些信息和监听函数。比如说我们可以通过 setRequestHeader 方法来为请求添加头信息。我们还可以为这个对象添加一个状态监听函数。一个 XMLHttpRequest 对象一共有 5 个状态，当它的状态变化时会触发onreadystatechange 事件，我们可以通过设置监听函数，来处理请求成功后的结果。当对象的 readyState 变为 4 的时候，代表服务器返回的数据接收完成，这个时候我们可以通过判断请求的状态，如果状态是 2xx 或者 304 的话则代表返回正常。这个时候我们就可以通过 response 中的数据来对页面进行更新了。

当对象的属性和监听函数设置完成后，最后我们调用 send 方法来向服务器发起请求，可以传入参数作为发送的数据体。

详细资料可以参考：
《XMLHttpRequest 对象》
《从 ajax 到 fetch、axios》
《Fetch 入门》
《传统 Ajax 已死，Fetch 永生》

56. 谈一谈浏览器的缓存机制?

浏览器的缓存机制指的是通过在一段时间内保留已接收到的 web 资源的一个副本，如果在资源的有效时间内，发起了对这个资源的再一次请求，那么浏览器会直接使用缓存的副本，而不是向服务器发起请求。使用 web 缓存可以有效地提高页面的打开速度，减少不必要的网络带宽的消耗。

web 资源的缓存策略一般由服务器来指定，可以分为两种，分别是强缓存策略和协商缓存策略。

使用强缓存策略时，如果缓存资源有效，则直接使用缓存资源，不必再向服务器发起请求。强缓存策略可以通过两种方式来设置，分别是 http 头信息中的 Expires 属性和 Cache-Control 属性。

服务器通过在响应头中添加 Expires 属性，来指定资源的过期时间。在过期时间以内，该资源可以被缓存使用，不必再向服务器发送请求。这个时间是一个绝对时间，它是服务器的时间，因此可能存在这样的问题，就是客户端的时间和服务器端的时间不一致，或者用户可以对客户端时间进行修改的情况，这样就可能会影响缓存命中的结果。

Expires 是 http1.0 中的方式，因为它的一些缺点，在 http 1.1 中提出了一个新的头部属性就是 Cache-Control 属性，
它提供了对资源的缓存的更精确的控制。它有很多不同的值，常用的比如我们可以通过设置 max-age 来指定资源能够被缓存的时间
的大小，这是一个相对的时间，它会根据这个时间的大小和资源第一次请求时的时间来计算出资源过期的时间，因此相对于 Expires
来说，这种方式更加有效一些。常用的还有比如 private ，用来规定资源只能被客户端缓存，不能够代理服务器所缓存。还有如 n
o-store ，用来指定资源不能够被缓存，no-cache 代表该资源能够被缓存，但是立即失效，每次都需要向服务器发起请求。

一般来说只需要设置其中一种方式就可以实现强缓存策略，当两种方式一起使用时，Cache-Control 的优先级要高于 Expires 。

使用协商缓存策略时，会先向服务器发送一个请求，如果资源没有发生修改，则返回一个 304 状态，让浏览器使用本地的缓存副本。
如果资源发生了修改，则返回修改后的资源。协商缓存也可以通过两种方式来设置，分别是 http 头信息中的 Etag 和 Last-Modified 属性。

服务器通过在响应头中添加 Last-Modified 属性来指出资源最后一次修改的时间，当浏览器下一次发起请求时，会在请求头中添加一个 If-Modified-Since 的属性，属性值为上一次资源返回时的 Last-Modified 的值。当请求发送到服务器后服务器会通过这个属性来和资源的最后一次的修改时间来进行比较，以此来判断资源是否做了修改。如果资源没有修改，那么返回 304 状态，让客户端使用本地的缓存。如果资源已经被修改了，则返回修改后的资源。使用这种方法有一个缺点，就是 Last-Modified 标注的最后修改时间只能精确到秒级，如果某些文件在1秒钟以内，被修改多次的话，那么文件已经改变了但是 Last-Modified 却没有改变，
这样会造成缓存命中的不准确。

因为 Last-Modified 的这种可能发生的不准确性，http 中提供了另外一种方式，那就是 Etag 属性。服务器在返回资源的时候，在头信息中添加了 Etag 属性，这个属性是资源生成的唯一标识符，当资源发生改变的时候，这个值也会发生改变。在下一次资源请求时，浏览器会在请求头中添加一个 If-None-Match 属性，这个属性的值就是上次返回的资源的 Etag 的值。服务接收到请求后会根据这个值来和资源当前的 Etag 的值来进行比较，以此来判断资源是否发生改变，是否需要返回资源。通过这种方式，比 Last-Modified 的方式更加精确。

当 `Last-Modified` 和 `Etag` 属性同时出现的时候，`Etag` 的优先级更高。使用协商缓存的时候，服务器需要考虑负载平衡的问题，因此多个服务器上资源的 `Last-Modified` 应该保持一致，因为每个服务器上 `Etag` 的值都不一样，因此在考虑负载平衡时，最好不要设置 `Etag` 属性。

强缓存策略和协商缓存策略在缓存命中时都会直接使用本地的缓存副本，区别只在于协商缓存会向服务器发送一次请求。它们缓存不命中时，都会向服务器发送请求来获取资源。在实际的缓存机制中，强缓存策略和协商缓存策略是一起合作使用的。浏览器首先会根据请求的信息判断，强缓存是否命中，如果命中则直接使用资源。如果不命中则根据头信息向服务器发起请求，使用协商缓存，如果协商缓存命中的话，则服务器不返回资源，浏览器直接使用本地资源的副本，如果协商缓存不命中，则服务器返回最新的资源给浏览器。

详细资料可以参考：
《浅谈浏览器缓存》
《前端优化：浏览器缓存技术介绍》
《请求头中的 Cache-Control》
《Cache-Control 字段值详解》

57. Ajax 解决浏览器缓存问题?

- 1.在 ajax 发送请求前加上 anyAjaxObj.setRequestHeader("If-Modified-Since","0")。
- 2.在 ajax 发送请求前加上 anyAjaxObj.setRequestHeader("Cache-Control","no-cache")。
- 3.在 URL 后面加上一个随机数： "fresh=" + Math.random();。
- 4.在 URL 后面加上时间戳："nowtime=" + new Date().getTime();。
- 5.如果是使用 jQuery，直接这样就可以了$.ajaxSetup({cache:false})。这样页面的所有 ajax 都会执行这条语句就是不需要保存缓存记录。

详细资料可以参考：
《Ajax 中浏览器的缓存问题解决方法》
《浅谈浏览器缓存》

58. 同步和异步的区别?

相关知识点：

同步，可以理解为在执行完一个函数或方法之后，一直等待系统返回值或消息，这时程序是处于阻塞的，只有接收到返回的值或消息后才往下执行其他的命令。

异步，执行完函数或方法后，不必阻塞性地等待返回值或消息，只需要向系统委托一个异步过程，那么当系统接收到返回值或消息时，系统会自动触发委托的异步过程，从而完成一个完整的流程。

回答：

同步指的是当一个进程在执行某个请求的时候，如果这个请求需要等待一段时间才能返回，那么这个进程会一直等待下去，直到消息返
回为止再继续向下执行。

异步指的是当一个进程在执行某个请求的时候，如果这个请求需要等待一段时间才能返回，这个时候进程会继续往下执行，不会阻塞等
待消息的返回，当消息返回时系统再通知进程进行处理。

详细资料可以参考：
《同步和异步的区别》

59. 什么是浏览器的同源政策?

我对浏览器的同源政策的理解是，一个域下的 `js` 脚本在未经允许的情况下，不能够访问另一个域的内容。这里的同源的指的是两个
域的协议、域名、端口号必须相同，否则则不属于同一个域。

同源政策主要限制了三个方面

第一个是当前域下的 `js` 脚本不能够访问其他域下的 `cookie`、`localStorage` 和 `indexDB`。

第二个是当前域下的 `js` 脚本不能够操作访问操作其他域下的 `DOM`。

第三个是当前域下 `ajax` 无法发送跨域请求。

同源政策的目的主要是为了保证用户的信息安全，它只是对 `js` 脚本的一种限制，并不是对浏览器的限制，对于一般的 `img`、或者
`script` 脚本请求都不会有跨域的限制，这是因为这些操作都不会通过响应结果来进行可能出现安全问题的操作。

60. 如何解决跨域问题?

相关知识点:

- 1. 通过 jsonp 跨域
- 2. document.domain + iframe 跨域
- 3. location.hash + iframe
- 4. window.name + iframe 跨域
- 5. postMessage 跨域
- 6. 跨域资源共享（CORS)
- 7. nginx 代理跨域
- 8. nodejs 中间件代理跨域
- 9. WebSocket 协议跨域

回答:

解决跨域的方法我们可以根据我们想要实现的目的来划分。

首先我们如果只是想要实现主域名下的不同子域名的跨域操作，我们可以使用设置 `document.domain` 来解决。

（1）将 `document.domain` 设置为主域名，来实现相同子域名的跨域操作，这个时候主域名下的 `cookie` 就能够被子域名所访问。同时如果文档中含有主域名相同，子域名不同的 `iframe` 的话，我们也可以对这个 `iframe` 进行操作。

如果是想要解决不同跨域窗口间的通信问题，比如说一个页面想要和页面的中的不同源的 `iframe` 进行通信的问题，我们可以使用 `location.hash` 或者 `window.name` 或者 `postMessage` 来解决。

（2）使用 `location.hash` 的方法，我们可以在主页面动态的修改 `iframe` 窗口的 `hash` 值，然后在 `iframe` 窗口里实现监听函数来实现这样一个单向的通信。因为在 `iframe` 是没有办法访问到不同源的父级窗口的，所以我们不能直接修改父级窗口的 `hash` 值来实现通信，我们可以在 `iframe` 中再加入一个 `iframe` ，这个 `iframe` 的内容是和父级页面同源的，所以我们可以 `window.parent.parent` 来修改最顶级页面的 `src`，以此来实现双向通信。

（3）使用 window.name 的方法，主要是基于同一个窗口中设置了 window.name 后不同源的页面也可以访问，所以不同源的子页面可以首先在 window.name 中写入数据，然后跳转到一个和父级同源的页面。这个时候级页面就可以访问同源的子页面中 window.name 中的数据了，这种方式的好处是可以传输的数据量大。

（4）使用 postMessage 来解决的方法，这是一个 h5 中新增的一个 api。通过它我们可以实现多窗口间的信息传递，通过获取到指定窗口的引用，然后调用 postMessage 来发送信息，在窗口中我们通过对 message 信息的监听来接收信息，以此来实现不同源间的信息交换。

如果是像解决 ajax 无法提交跨域请求的问题，我们可以使用 jsonp、cors、websocket 协议、服务器代理来解决问题。

（5）使用 jsonp 来实现跨域请求，它的主要原理是通过动态构建 script 标签来实现跨域请求，因为浏览器对 script 标签的引入没有跨域的访问限制 。通过在请求的 url 后指定一个回调函数，然后服务器在返回数据的时候，构建一个 json 数据的包装，这个包装就是回调函数，然后返回给前端，前端接收到数据后，因为请求的是脚本文件，所以会直接执行，这样我们先前定义好的回调函数就可以被调用，从而实现了跨域请求的处理。这种方式只能用于 get 请求。

（6）使用 CORS 的方式，CORS 是一个 W3C 标准，全称是"跨域资源共享"。CORS 需要浏览器和服务器同时支持。目前，所有浏览器都支持该功能，因此我们只需要在服务器端配置就行。浏览器将 CORS 请求分成两类：简单请求和非简单请求。对于简单请求，浏览器直接发出 CORS 请求。具体来说，就是会在头信息之中，增加一个 Origin 字段。Origin 字段用来说明本次请求来自哪个源。服务器根据这个值，决定是否同意这次请求。对于如果 Origin 指定的源，不在许可范围内，服务器会返回一个正常的 HTTP 回应。浏览器发现，这个回应的头信息没有包含 Access-Control-Allow-Origin 字段，就知道出错了，从而抛出一个错误，ajax 不会收到响应信息。如果成功的话会包含一些以 Access-Control- 开头的字段。

非简单请求，浏览器会先发出一次预检请求，来判断该域名是否在服务器的白名单中，如果收到肯定回复后才会发起请求。

（7）使用 websocket 协议，这个协议没有同源限制。

（8）使用服务器来代理跨域的访问请求，就是有跨域的请求操作时发送请求给后端，让后端代为请求，然后最后将获取的结果返回。

详细资料可以参考：
《前端常见跨域解决方案（全）》
《浏览器同源政策及其规避方法》
《跨域，你需要知道的全在这里》
《为什么 form 表单提交没有跨域问题，但 ajax 提交有跨域问题？》

61. 服务器代理转发时，该如何处理 cookie?

详细资料可以参考：
《深入浅出 Nginx》

62. 简单谈一下 cookie ?

我的理解是 cookie 是服务器提供的一种用于维护会话状态信息的数据，通过服务器发送到浏览器，浏览器保存在本地，当下一次有同源的请求时，将保存的 cookie 值添加到请求头部，发送给服务端。这可以用来实现记录用户登录状态等功能。cookie 一般可以存储 4k 大小的数据，并且只能够被同源的网页所共享访问。

服务器端可以使用 Set-Cookie 的响应头部来配置 cookie 信息。一条cookie 包括了5个属性值 expires、domain、path、secure、HttpOnly。其中 expires 指定了 cookie 失效的时间，domain 是域名、path是路径，domain 和 path 一起限制了 cookie 能够被哪些 url 访问。secure 规定了 cookie 只能在确保安全的情况下传输，HttpOnly 规定了这个 cookie 只能被服务器访问，不能使用 js 脚本访问。

在发生 xhr 的跨域请求的时候，即使是同源下的 cookie，也不会被自动添加到请求头部，除非显示地规定。

详细资料可以参考：
《HTTP cookies》
《聊一聊 cookie》

63. 模块化开发怎么做？ ——参考JS模块化

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。

1. 全局function模式：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。

2. namespace模式：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模块成员，外部代码可以修改内部属性的值。

3. IIFE模式（闭包）：之后发展为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。但是这种方法无法解决当前这个模块依赖另一个模块的问题。

4. IIFE增强模式：通过在IIFE模式中引入依赖，这就是现代模块实现的基石

详细资料可以参考：
《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》

64. 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。参考65

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。这种方案和上面三种方案都不同。参考 66。

65. 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。

```
// 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();
  // ...
});

// 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();
  // ...
});
```

详细资料可以参考：
《前端模块化，AMD 与 CMD 的区别》

66. 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 模块输出的是值的拷贝，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。

67. requireJS 的核心原理是什么？（如何动态加载的？如何避免多次加载的？如何 缓存的？）

> `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？》
《从 RequireJs 源码剖析脚本加载原理》
《requireJS 原理分析》

68. JS 模块加载器的轮子怎么造，也就是如何实现一个模块加载器?

详细资料可以参考：
《JS 模块加载器加载原理是怎么样的？》

69. ECMAScript6 怎么写 class，为什么会出现 class 这种东西?

> 在我看来 `ES6` 新添加的 `class` 只是为了补充 `js` 中缺少的一些面向对象语言的特性，但本质上来说它只是一种语法糖，不是一个新的东西，其背后还是原型继承的思想。通过加入 `class` 可以有利于我们更好的组织代码。
>
> 在 `class` 中添加的方法，其实是添加在类的原型上的。

详细资料可以参考：
《ECMAScript 6 实现了 class，对 JavaScript 前端开发有什么意义？》
《Class 的基本语法》

70. documen.write 和 innerHTML 的区别?

> `document.write` 的内容会代替整个文档内容，会重写整个页面。
>
> `innerHTML` 的内容只是替代指定元素的内容，只会重写页面中的部分内容。

详细资料可以参考：
《简述 document.write 和 innerHTML 的区别。》

71. 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

```
以下方法都是document调用：
createDocumentFragment(node);
createElement(node);
createTextNode(text);
```

（2）添加、移除、替换、插入

```
以下方法都是父元素调用，操作子元素：
appendChild(node)
removeChild(node)
replaceChild(new,old)
insertBefore(new,old)
```

（3）查找

```
getElementById();
getElementsByTagName();
getElementsByClassName();
querySelector();
querySelectorAll();
```

（4）属性操作

```
getAttribute(key);
setAttribute(key, value);
hasAttribute(key);
removeAttribute(key);
```

详细资料可以参考：
《DOM 概述》
《原生 JavaScript 的 DOM 操作汇总》
《原生 JS 中 DOM 节点相关 API 合集》

72. innerHTML 与 outerHTML 的区别?

```
对于这样一个 HTML 元素：<div>content<br/></div>。

innerHTML：内部 HTML，content<br/>；
outerHTML：外部 HTML，<div>content<br/></div>；
innerText：内部文本，content ；
outerText：内部文本，content ；
```

73. .call() 和 .apply() 的区别?

它们的作用一模一样，区别仅在于传入参数的形式的不同。

`apply` 接受两个参数，第一个参数指定了函数体内 `this` 对象的指向，第二个参数为一个带下标的集合，这个集合可以为数组，也可以为类数组，`apply` 方法把这个集合中的元素作为参数传递给被调用的函数。

`call` 传入的参数数量不固定，跟 `apply` 相同的是，第一个参数也是代表函数体内的 `this` 指向，从第二个参数开始往后，每个参数被依次传入函数。

详细资料可以参考：
《apply、call 的区别和用途》

74. JavaScript 类数组对象的定义?

一个拥有 `length` 属性和若干索引属性的对象就可以被称为类数组对象，类数组对象和数组类似，但是不能调用数组的方法。

常见的类数组对象有 `arguments` 和 `DOM` 方法的返回结果，还有一个函数也可以被看作是类数组对象，因为它含有 `length`
属性值，代表可接收的参数个数。

常见的类数组转换为数组的方法有这样几种：

（1）通过 call 调用数组的 slice 方法来实现转换

```
Array.prototype.slice.call(arrayLike);
```

（2）通过 call 调用数组的 splice 方法来实现转换

```
Array.prototype.splice.call(arrayLike, 0);
```

（3）通过 apply 调用数组的 concat 方法来实现转换

```
Array.prototype.concat.apply([], arrayLike);
```

（4）通过 Array.from 方法来实现转换

```
Array.from(arrayLike);
```

详细的资料可以参考：
《JavaScript 深入之类数组对象与 arguments》
《javascript 类数组》
《深入理解 JavaScript 类数组》

75. 数组和对象有哪些原生方法，列举一下?

数组和字符串的转换方法：toString()、toLocalString()、join() 其中 join() 方法可以指定转换为字符串时的分隔符。

数组尾部操作的方法 pop() 和 push()，push 方法可以传入多个参数。

数组首部操作的方法 shift() 和 unshift() 重排序的方法 reverse() 和 sort()，sort() 方法可以传入一个函数来进行比较，传入前后两个值，如果返回值为正数，则交换两个参数的位置。

数组连接的方法 concat() ，返回的是拼接好的数组，不影响原数组。

数组截取办法 slice()，用于截取数组中的一部分返回，不影响原数组。

数组插入方法 splice()，影响原数组查找特定项的索引的方法，indexOf() 和 lastIndexOf() 迭代方法 every()、some()、filter()、map() 和 forEach() 方法

数组归并方法 reduce() 和 reduceRight() 方法

详细资料可以参考：
《JavaScript 深入理解之 Array 类型详解》

76. 数组的 fill 方法?

fill() 方法用一个固定值填充一个数组中从起始索引到终止索引内的全部元素。不包括终止索引。
fill 方法接受三个参数 value，start 以及 end，start 和 end 参数是可选的，其默认值分别为 0 和 this 对象的 length 属性值。

详细资料可以参考：
《Array.prototype.fill()》

77. [,,,] 的长度?

尾后逗号 （有时叫做“终止逗号”）在向 JavaScript 代码添加元素、参数、属性时十分有用。如果你想要添加新的属性，并且上一行已经使用了尾后逗号，你可以仅仅添加新的一行，而不需要修改上一行。这使得版本控制更加清晰，以及代码维护麻烦更少。

JavaScript 一开始就支持数组字面值中的尾后逗号，随后向对象字面值（ECMAScript 5）中添加了尾后逗号。最近（ECMAS
cript 2017），又将其添加到函数参数中。但是 JSON 不支持尾后逗号。

如果使用了多于一个尾后逗号，会产生间隙。 带有间隙的数组叫做稀疏数组（密致数组没有间隙）。稀疏数组的长度为逗号的数
量。

详细资料可以参考：
《尾后逗号》

78. JavaScript 中的作用域与变量声明提升?

变量提升的表现是，无论我们在函数中何处位置声明的变量，好像都被提升到了函数的首部，我们可以在变量声明前访问到而不会报错。

造成变量声明提升的本质原因是 `js` 引擎在代码执行前有一个解析的过程，创建了执行上下文，初始化了一些代码执行时需要用到的对象。当我们访问一个变量时，我们会到当前执行上下文中的作用域链中去查找，而作用域链的首端指向的是当前执行上下文的变量对象，这个变量对象是执行上下文的一个属性，它包含了函数的形参、所有的函数和变量声明，这个对象的是在代码解析的时候创建的。这就是会出现变量声明提升的根本原因。

详细资料可以参考：
《JavaScript 深入理解之变量对象》

79. 如何编写高性能的 Javascript？

- 1.使用位运算代替一些简单的四则运算。
- 2.避免使用过深的嵌套循环。
- 3.不要使用未定义的变量。
- 4.当需要多次访问数组长度时，可以用变量保存起来，避免每次都会去进行属性查找。

详细资料可以参考：
《如何编写高性能的 Javascript？》

80. 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。

详细资料可以参考：
《深入理解 V8 的垃圾回收原理》
《JavaScript 中的垃圾回收》

81. 哪些操作会造成内存泄漏?

相关知识点：

- 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

回答：

> 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。

详细资料可以参考：
《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》

82. 需求：实现一个页面操作不会整页刷新的网站，并且能在浏览器前进、后退时正确响应。给出你的技术实现方案?

> 通过使用 pushState + ajax 实现浏览器无刷新前进后退，当一次 ajax 调用成功后我们将一条 state 记录加入到 history
> 对象中。一条 state 记录包含了 url、title 和 content 属性，在 popstate 事件中可以获取到这个 state 对象，我们可
> 以使用 content 来传递数据。最后我们通过对 window.onpopstate 事件监听来响应浏览器的前进后退操作。
>
> 使用 pushState 来实现有两个问题，一个是打开首页时没有记录，我们可以使用 replaceState 来将首页的记录替换，另一个问
> 题是当一个页面刷新的时候，仍然会向服务器端请求数据，因此如果请求的 url 需要后端的配合将其重定向到一个页面。

详细资料可以参考：
《pushState + ajax 实现浏览器无刷新前进后退》
《Manipulating the browser history》

83. 如何判断当前脚本运行在浏览器还是 node 环境中？（阿里）

```
this === window ? 'browser' : 'node';
```

通过判断 Global 对象是否为 window，如果不为 window，当前脚本没有运行在浏览器中。

84. 把 script 标签放在页面的最底部的 body 封闭之前和封闭之后有什么区别？浏览器会如何解析它们？

详细资料可以参考：
《为什么把 script 标签放在 body 结束标签之后 html 结束标签之前？》
《从 Chrome 源码看浏览器如何加载资源》

85. 移动端的点击事件的有延迟，时间是多久，为什么会有？ 怎么解决这个延时？

移动端点击有 300ms 的延迟是因为移动端会有双击缩放的这个操作，因此浏览器在 click 之后要等待 300ms，看用户有没有下一次点击，来判断这次操作是不是双击。

有三种办法来解决这个问题：

- 1.通过 meta 标签禁用网页的缩放。
- 2.通过 meta 标签将网页的 viewport 设置为 ideal viewport。
- 3.调用一些 js 库，比如 FastClick

click 延时问题还可能引起点击穿透的问题，就是如果我们在一个元素上注册了 touchStart 的监听事件，这个事件会将这个元素隐藏掉，我们发现当这个元素隐藏后，触发了这个元素下的一个元素的点击事件，这就是点击穿透。

详细资料可以参考：
《移动端 300ms 点击延迟和点击穿透》

86. 什么是“前端路由”？什么时候适合使用“前端路由”？“前端路由”有哪些优点和缺点？

（1）什么是前端路由？

前端路由就是把不同路由对应不同的内容或页面的任务交给前端来做，之前是通过服务端根据 url 的不同返回不同的页面实现的。

（2）什么时候使用前端路由？

在单页面应用，大部分页面结构不变，只改变部分内容的使用

（3）前端路由有什么优点和缺点？

优点：用户体验好，不需要每次都从服务器全部获取，快速展现给用户

缺点：单页面无法记住之前滚动的位置，无法在前进，后退的时候记住滚动的位置

前端路由一共有两种实现方式，一种是通过 hash 的方式，一种是通过使用 pushState 的方式。

详细资料可以参考：
《什么是“前端路由”》
《浅谈前端路由》
《前端路由是什么东西？》

87. 如何测试前端代码么？ 知道 BDD, TDD, Unit Test 么？ 知道怎么测试你的前端工程么(mocha, sinon, jasmin, qUnit..)?

详细资料可以参考：
《浅谈前端单元测试》

88. 检测浏览器版本有哪些方式?

检测浏览器版本一共有两种方式：

一种是检测 window.navigator.userAgent 的值，但这种方式很不可靠，因为 userAgent 可以被改写，并且早期的浏览器如 ie，会通过伪装自己的 userAgent 的值为 Mozilla 来躲过服务器的检测。

第二种方式是功能检测，根据每个浏览器独有的特性来进行判断，如 ie 下独有的 ActiveXObject。

详细资料可以参考：
《JavaScript 判断浏览器类型》

89. 什么是 Polyfill ?

Polyfill 指的是用于实现浏览器并不支持的原生 API 的代码。

比如说 querySelectorAll 是很多现代浏览器都支持的原生 Web API，但是有些古老的浏览器并不支持，那么假设有人写了一段代码来实现这个功能使这些浏览器也支持了这个功能，那么这就可以成为一个 Polyfill。

一个 shim 是一个库，有自己的 API，而不是单纯实现原生不支持的 API。

详细资料可以参考：
《Web 开发中的“黑话”》
《Polyfill 为何物》

90. 使用 JS 实现获取文件扩展名?

```
// String.lastIndexOf() 方法返回指定值（本例中的'.'）在调用该方法的字符串中最后出现的位置，如果没找到则返回 -1。

// 对于 'filename' 和 '.hiddenfile'，lastIndexOf 的返回值分别为 -1 和 0， 无符号右移操作符(>>>) 将 -1 转换为 4294967295，将 -2 转换为 4294967294，这个方法可以保证边缘情况时文件名不变。

// String.prototype.slice() 从上面计算的索引处提取文件的扩展名。如果索引比文件名的长度大，结果为""。
function getFileExtension(filename) {
  return filename.slice(((filename.lastIndexOf(".") - 1) >>> 0) + 2);
}
```

详细资料可以参考：
《如何更有效的获取文件扩展名》

91. 介绍一下 js 的节流与防抖?

相关知识点：参考性能优化中的函数防抖与节流

```
// 函数防抖： 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。

// 函数节流： 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。

// 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，使事件间隔指定时间后执行
    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);
    }
  };
}
```

回答:

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。

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》

92. Object.is() 与原来的比较操作符 “= = = ==”、“=== =” 的区别?

相关知识点：

两等号判等，会在比较时进行类型转换。
三等号判等（判断严格），比较时不进行隐式类型转换，（类型不同则会返回`false`）。

`Object.is` 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`.

`Object.is` 应被认为有其特殊的用途，而不能用它认为它比其它的相等对比更宽松或严格。

回答：

使用双等号进行相等判断时，如果两边的类型不一致，则会进行强制类型转化后再进行比较。

使用三等号进行相等判断时，如果两边的类型不一致时，不会做强制类型准换，直接返回 `false`。

使用 `Object.is` 来进行相等判断时，一般情况下和三等号的判断相同，它处理了一些特殊的情况，比如 `-0` 和 `+0` 不再相等，两个 `NaN` 认定为是相等的。

93. escape,encodeURI,encodeURIComponent 有什么区别?

相关知识点：

`escape` 和 `encodeURI` 都属于 `Percent-encoding`，基本功能都是把 `URI` 非法字符转化成合法字符，转化后形式类似「%*」。
它们的根本区别在于，`escape` 在处理 `0xff` 之外字符的时候，是直接使用字符的 `unicode` 在前面加上一个「%u」，而 `encode URI` 则是先进行 `UTF-8`，再在 `UTF-8` 的每个字节码前加上一个「%」；在处理 `0xff` 以内字符时，编码方式是一样的（都是「%XX」，`XX` 为字符的 `16` 进制 `unicode`，同时也是字符的 `UTF-8`），只是范围（即哪些字符编码哪些字符不编码）不一样。

回答：

`encodeURI` 是对整个 `URI` 进行转义，将 `URI` 中的非法字符转换为合法字符，所以对于一些在 `URI` 中有特殊意义的字符不会进行转义。

`encodeURIComponent` 是对 `URI` 的组成部分进行转义，所以一些特殊字符也会得到转义。

`escape` 和 `encodeURI` 的作用相同，不过它们对于 `unicode` 编码为 `0xff` 之外字符的时候会有区别，`escape` 是直接在字符的 `unicode` 编码前加上 `%u`，而 `encodeURI` 首先会将字符转换为 `UTF-8` 的格式，再在每个字节前加上 `%`。

详细资料可以参考：
《escape,encodeURI,encodeURIComponent 有什么区别?》

94. Unicode 和 UTF-8 之间的关系?

Unicode 是一种字符集合，现在可容纳 100 多万个字符。每个字符对应一个不同的 Unicode 编码，它只规定了符号的二进制代码，却没有规定这个二进制代码在计算机中如何编码传输。

UTF-8 是一种对 Unicode 的编码方式，它是一种变长的编码方式，可以用 1~4 个字节来表示一个字符。

详细资料可以参考：
《字符编码详解》
《字符编码笔记：ASCII，Unicode 和 UTF-8》

95. js 的事件循环是什么?

相关知识点：

事件队列是一个存储着待执行任务的队列，其中的任务严格按照时间先后顺序执行，排在队头的任务将会率先执行，而排在队尾的任务会最后执行。事件队列每次仅执行一个任务，在该任务执行完毕之后，再执行下一个任务。执行栈则是一个类似于函数调用栈的运行容器，当执行栈为空时，JS 引擎便检查事件队列，如果不为空的话，事件队列便将第一个任务压入执行栈中运行。

回答：

因为 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。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。当异步事件执行完毕后，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。任务队列可以分为宏任务队列和微任务队列，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。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。

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。

详细资料可以参考：
《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》

96. js 中的深浅拷贝实现?

相关资料：

```
// 浅拷贝的实现;

function shallowCopy(object) {
  // 只拷贝对象
  if (!object || typeof object !== "object") return;

  // 根据 object 的类型判断是新建一个数组还是对象
  let newObject = Array.isArray(object) ? [] : {};
```

```
  // 遍历 object，并且判断是 object 的属性才拷贝
  for (let key in object) {
    if (object.hasOwnProperty(key)) {
      newObject[key] = object[key];
    }
  }

  return newObject;
}

// 深拷贝的实现;

function deepCopy(object) {
  if (!object || typeof object !== "object") return;

  let newObject = Array.isArray(object) ? [] : {};

  for (let key in object) {
    if (object.hasOwnProperty(key)) {
      newObject[key] =
        typeof object[key] === "object" ? deepCopy(object[key]) : object[key];
    }
  }

  return newObject;
}
```

回答：

浅拷贝指的是将一个对象的属性值复制到另一个对象，如果有的属性的值为引用类型的话，那么会将这个引用的地址复制给对象，因此两个对象会有同一个引用类型的引用。浅拷贝可以使用 `Object.assign` 和展开运算符来实现。

深拷贝相对浅拷贝而言，如果遇到属性值为引用类型的时候，它新建一个引用类型并将对应的值复制给它，因此对象获得的一个新的引用类型而不是一个原有类型的引用。深拷贝对于一些对象可以使用 `JSON` 的两个函数来实现，但是由于 `JSON` 的对象格式比 `js` 的对象格式更加严格，所以如果属性值里边出现函数或者 `Symbol` 类型的值时，会转换失败。

详细资料可以参考：
《JavaScript 专题之深浅拷贝》
《前端面试之道》

97. 手写 call、apply 及 bind 函数

相关资料：

```
// 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;
};

// 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;
};

// 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)
    );
  };
};
```

回答：

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。

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。
- 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.将函数作为上下文对象的一个属性。
- 4.判断参数值是否传入
- 4.使用上下文对象来调用这个方法，并保存返回结果。
- 5.删除刚才新增的属性
- 6.返回结果

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。

详细资料可以参考：
《手写 call、apply 及 bind 函数》
《JavaScript 深入之 call 和 apply 的模拟实现》

98. 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);
}
```

详细资料可以参考：
《JavaScript 专题之函数柯里化》

99. 为什么 0.1 + 0.2 != 0.3？如何解决这个问题？

当计算机计算 0.1+0.2 的时候，实际上计算的是这两个数字在计算机里所存储的二进制，0.1 和 0.2 在转换为二进制表示的时候会出现位数无限循环的情况。js 中是以 64 位双精度格式来存储数字的，只有 53 位的有效数字，超过这个长度的位数会被截取掉这样就造成了精度丢失的问题。这是第一个会造成精度丢失的地方。在对两个以 64 位双精度格式的数据进行计算的时候，首先会进行对阶的处理，对阶指的是将阶码对齐，也就是将小数点的位置对齐后，再进行计算，一般是小阶向大阶对齐，因此小阶的数在对齐的过程中，有效数字会向右移动，移动后超过有效位数的位会被截取掉，这是第二个可能会出现精度丢失的地方。当两个数据阶码对齐后，进行相加运算后，得到的结果可能会超过 53 位有效数字，因此超过的位数也会被截取掉，这是可能发生精度丢失的第三个地方。

对于这样的情况，我们可以将其转换为整数后再进行运算，运算后再转换为对应的小数，以这种方式来解决这个问题。

我们还可以将两个数相加的结果和右边相减，如果相减的结果小于一个极小数，那么我们就可以认定结果是相等的，这个极小数可以
使用 es6 的 Number.EPSILON

详细资料可以参考：
《十进制的 0.1 为什么不能用二进制很好的表示？》
《十进制浮点数转成二进制》
《浮点数的二进制表示》
《js 浮点数存储精度丢失原理》
《浮点数精度之谜》
《JavaScript 浮点数陷阱及解法》
《0.1+0.2 !== 0.3？》
《JavaScript 中奇特的~运算符》

100. 原码、反码和补码的介绍

原码是计算机中对数字的二进制的定点表示方法，最高位表示符号位，其余位表示数值位。优点是易于分辨，缺点是不能够直接参与运算。

正数的反码和其原码一样；负数的反码，符号位为1，数值部分按原码取反。
如 [+7]原 = 00000111，[+7]反 = 00000111； [-7]原 = 10000111，[-7]反 = 11111000。

正数的补码和其原码一样；负数的补码为其反码加1。

例如 [+7]原 = 00000111，[+7]反 = 00000111，[+7]补 = 00000111；
[-7]原 = 10000111，[-7]反 = 11111000，[-7]补 = 11111001

之所以在计算机中使用补码来表示负数的原因是，这样可以将加法运算扩展到所有的数值计算上，因此在数字电路中我们只需要考虑加法器的设计就行了，而不用再为减法设置新的数字电路。

详细资料可以参考：
《关于 2 的补码》

101. toPrecision 和 toFixed 和 Math.round 的区别?

toPrecision 用于处理精度，精度是从左至右第一个不为 0 的数开始数起。
toFixed 是对小数点后指定位数取整，从小数点开始数起。
Math.round 是将一个数字四舍五入到一个整数。

102. 什么是 XSS 攻击？如何防范 XSS 攻击?

XSS 攻击指的是跨站脚本攻击，是一种代码注入攻击。攻击者通过在网站注入恶意脚本，使之在用户的浏览器上运行，从而盗取用户的信息如 cookie 等。

XSS 的本质是因为网站没有对恶意代码进行过滤，与正常的代码混合在一起了，浏览器没有办法分辨哪些脚本是可信的，从而导致了恶意代码的执行。

XSS 一般分为存储型、反射型和 DOM 型。

存储型指的是恶意代码提交到了网站的数据库中，当用户请求数据的时候，服务器将其拼接为 HTML 后返回给了用户，从而导致了恶意代码的执行。

反射型指的是攻击者构建了特殊的 URL，当服务器接收到请求后，从 URL 中获取数据，拼接到 HTML 后返回，从而导致了恶意代码的执行。

DOM 型指的是攻击者构建了特殊的 URL，用户打开网站后，js 脚本从 URL 中获取数据，从而导致了恶意代码的执行。

XSS 攻击的预防可以从两个方面入手，一个是恶意代码提交的时候，一个是浏览器执行恶意代码的时候。

对于第一个方面，如果我们对存入数据库的数据都进行的转义处理，但是一个数据可能在多个地方使用，有的地方可能不需要转义，由于我们没有办法判断数据最后的使用场景，所以直接在输入端进行恶意代码的处理，其实是不太可靠的。

因此我们可以从浏览器的执行来进行预防，一种是使用纯前端的方式，不用服务器端拼接后返回。另一种是对需要插入到 HTML 中的代码做好充分的转义。对于 DOM 型的攻击，主要是前端脚本的不可靠而造成的，我们对于数据获取渲染和字符串拼接的时候应该对可能出现的恶意代码情况进行判断。

还有一些方式，比如使用 CSP ，CSP 的本质是建立一个白名单，告诉浏览器哪些外部资源可以加载和执行，从而防止恶意代码的注入攻击。

还可以对一些敏感信息进行保护，比如 cookie 使用 http-only ，使得脚本无法获取。也可以使用验证码，避免脚本伪装成用户执行一些操作。

详细资料可以参考：
《前端安全系列（一）：如何防止 XSS 攻击？》

103. 什么是 CSP?

CSP 指的是内容安全策略，它的本质是建立一个白名单，告诉浏览器哪些外部资源可以加载和执行。我们只需要配置规则，如何拦截由浏览器自己来实现。

通常有两种方式来开启 CSP，一种是设置 HTTP 首部中的 Content-Security-Policy，一种是设置 meta 标签的方式 <meta
http-equiv="Content-Security-Policy">

详细资料可以参考：
《内容安全策略（CSP）》
《前端面试之道》

104. 什么是 CSRF 攻击？如何防范 CSRF 攻击?

CSRF 攻击指的是跨站请求伪造攻击，攻击者诱导用户进入一个第三方网站，然后该网站向被攻击网站发送跨站请求。如果用户在被
攻击网站中保存了登录状态，那么攻击者就可以利用这个登录状态，绕过后台的用户验证，冒充用户向服务器执行一些操作。

CSRF 攻击的本质是利用了 cookie 会在同源请求中携带发送给服务器的特点，以此来实现用户的冒充。

一般的 CSRF 攻击类型有三种：

第一种是 GET 类型的 CSRF 攻击，比如在网站中的一个 img 标签里构建一个请求，当用户打开这个网站的时候就会自动发起提
交。

第二种是 POST 类型的 CSRF 攻击，比如说构建一个表单，然后隐藏它，当用户进入页面时，自动提交这个表单。

第三种是链接类型的 CSRF 攻击，比如说在 a 标签的 href 属性里构建一个请求，然后诱导用户去点击。

CSRF 可以用下面几种方法来防护：

第一种是同源检测的方法，服务器根据 http 请求头中 origin 或者 referer 信息来判断请求是否为允许访问的站点，从而对请求进行过滤。当 origin 或者 referer 信息都不存在的时候，直接阻止。这种方式的缺点是有些情况下 referer 可以被伪造。还有就是我们这种方法同时把搜索引擎的链接也给屏蔽了，所以一般网站会允许搜索引擎的页面请求，但是相应的页面请求这种请求方式也可能被攻击者给利用。

第二种方法是使用 CSRF Token 来进行验证，服务器向用户返回一个随机数 Token ，当网站再次发起请求时，在请求参数中加入服务器端返回的 token ，然后服务器对这个 token 进行验证。这种方法解决了使用 cookie 单一验证方式时，可能会被冒用的问题，但是这种方法存在一个缺点就是，我们需要给网站中的所有请求都添加上这个 token，操作比较繁琐。还有一个问题是一般不会只有一台网站服务器，如果我们的请求经过负载平衡转移到了其他的服务器，但是这个服务器的 session 中没有保留这个 token 的话，就没有办法验证了。这种情况我们可以通过改变 token 的构建方式来解决。

第三种方式使用双重 Cookie 验证的办法，服务器在用户访问网站页面时，向请求域名注入一个Cookie，内容为随机字符串，然后当用户再次向服务器发送请求的时候，从 cookie 中取出这个字符串，添加到 URL 参数中，然后服务器通过对 cookie 中的数据和参数中的数据进行比较，来进行验证。使用这种方式是利用了攻击者只能利用 cookie，但是不能访问获取 cookie 的特点。并且这种方法比 CSRF Token 的方法更加方便，并且不涉及到分布式访问的问题。这种方法的缺点是如果网站存在 XSS 漏洞的，那么这种方式会失效。同时这种方式不能做到子域名的隔离。

第四种方式是使用在设置 cookie 属性的时候设置 Samesite ，限制 cookie 不能作为被第三方使用，从而可以避免被攻击者利用。Samesite 一共有两种模式，一种是严格模式，在严格模式下 cookie 在任何情况下都不可能作为第三方 Cookie 使用，在宽松模式下，cookie 可以被请求是 GET 请求，且会发生页面跳转的请求所使用。

详细资料可以参考：
《前端安全系列之二：如何防止 CSRF 攻击？》
《[HTTP 趣谈] origin, referer 和 host 区别》

105. 什么是 Samesite Cookie 属性?

Samesite Cookie 表示同站 cookie，避免 cookie 被第三方所利用。

将 Samesite 设为 strict ，这种称为严格模式，表示这个 cookie 在任何情况下都不可能作为第三方 cookie。

将 Samesite 设为 Lax ，这种模式称为宽松模式，如果这个请求是个 GET 请求，并且这个请求改变了当前页面或者打开了新的页面，那么这个 cookie 可以作为第三方 cookie，其余情况下都不能作为第三方 cookie。

使用这种方法的缺点是，因为它不支持子域，所以子域没有办法与主域共享登录信息，每次转入子域的网站，都回重新登录。还有一个问题就是它的兼容性不够好。

106. 什么是点击劫持？如何防范点击劫持?

点击劫持是一种视觉欺骗的攻击手段，攻击者将需要攻击的网站通过 iframe 嵌套的方式嵌入自己的网页中，并将 iframe 设置为透明，在页面中透出一个按钮诱导用户点击。

我们可以在 http 相应头中设置 X-FRAME-OPTIONS 来防御用 iframe 嵌套的点击劫持攻击。通过不同的值，可以规定页面在特
定的一些情况才能作为 iframe 来使用。

详细资料可以参考：
《web 安全之--点击劫持攻击与防御技术简介》

107. SQL 注入攻击?

SQL 注入攻击指的是攻击者在 HTTP 请求中注入恶意的 SQL 代码，服务器使用参数构建数据库 SQL 命令时，恶意 SQL 被一起构
造，破坏原有 SQL 结构，并在数据库中执行，达到编写程序时意料之外结果的攻击行为。

详细资料可以参考：
《Web 安全漏洞之 SQL 注入》
《如何防范常见的 Web 攻击》

108. 什么是 MVVM？比之 MVC 有什么区别？什么又是 MVP ？

MVC、MVP 和 MVVM 是三种常见的软件架构设计模式，主要通过分离关注点的方式来组织代码结构，优化我们的开发效率。

比如说我们实验室在以前项目开发的时候，使用单页应用时，往往一个路由页面对应了一个脚本文件，所有的页面逻辑都在一个脚本文件里。页面的渲染、数据的获取，对用户事件的响应所有的应用逻辑都混合在一起，这样在开发简单项目时，可能看不出什么问题，当时一旦项目变得复杂，那么整个文件就会变得冗长，混乱，这样对我们的项目开发和后期的项目维护是非常不利的。

MVC 通过分离 Model、View 和 Controller 的方式来组织代码结构。其中 View 负责页面的显示逻辑，Model 负责存储页面的业务数据，以及对相应数据的操作。并且 View 和 Model 应用了观察者模式，当 Model 层发生改变的时候它会通知有关 View 层更新页面。Controller 层是 View 层和 Model 层的纽带，它主要负责用户与应用的响应操作，当用户与页面产生交互的时候，Controller 中的事件触发器就开始工作了，通过调用 Model 层，来完成对 Model 的修改，然后 Model 层再去通知 View 层更新。

MVP 模式与 MVC 唯一不同的在于 Presenter 和 Controller。在 MVC 模式中我们使用观察者模式，来实现当 Model 层数据发生变化的时候，通知 View 层的更新。这样 View 层和 Model 层耦合在一起，当项目逻辑变得复杂的时候，可能会造成代码的混乱，并且可能会对代码的复用性造成一些问题。MVP 的模式通过使用 Presenter 来实现对 View 层和 Model 层的解耦。MVC 中的 Controller 只知道 Model 的接口，因此它没有办法控制 View 层的更新，MVP 模式中，View 层的接口暴露给了 Presenter 因此我们可以在 Presenter 中将 Model 的变化和 View 的变化绑定在一起，以此来实现 View 和 Model 的同步更新。这样就实现了对 View 和 Model 的解耦，Presenter 还包含了其他的响应逻辑。

MVVM 模式中的 VM，指的是 ViewModel，它和 MVP 的思想其实是相同的，不过它通过双向的数据绑定，将 View 和 Model 的同步更新给自动化了。当 Model 发生变化的时候，ViewModel 就会自动更新；ViewModel 变化了，View 也会更新。这样就将 Presenter 中的工作给自动化了。我了解过一点双向数据绑定的原理，比如 vue 是通过使用数据劫持和发布订阅者模式来实现的这一功能。

详细资料可以参考：
《浅析前端开发中的 MVC/MVP/MVVM 模式》
《MVC，MVP 和 MVVM 的图示》
《MVVM》
《一篇文章了解架构模式：MVC/MVP/MVVM》

109. vue 双向数据绑定原理?

vue 通过使用双向数据绑定，来实现了 View 和 Model 的同步更新。vue 的双向数据绑定主要是通过使用数据劫持和发布订阅者模式来实现的。

首先我们通过 Object.defineProperty() 方法来对 Model 数据各个属性添加访问器属性，以此来实现数据的劫持，因此当 Model 中的数据发生变化的时候，我们可以通过配置的 setter 和 getter 方法来实现对 View 层数据更新的通知。

数据在 html 模板中一共有两种绑定情况，一种是使用 v-model 来对 value 值进行绑定，一种是作为文本绑定，在对模板引擎进行解析的过程中。

如果遇到元素节点，并且属性值包含 v-model 的话，我们就从 Model 中去获取 v-model 所对应的属性的值，并赋值给元素的 value 值。然后给这个元素设置一个监听事件，当 View 中元素的数据发生变化的时候触发该事件，通知 Model 中的对应的属性的值进行更新。

如果遇到了绑定的文本节点，我们使用 Model 中对应的属性的值来替换这个文本。对于文本节点的更新，我们使用了发布订阅者模式，属性作为一个主题，我们为这个节点设置一个订阅者对象，将这个订阅者对象加入这个属性主题的订阅者列表中。当 Model 层数据发生改变的时候，Model 作为发布者向主题发出通知，主题收到通知再向它的所有订阅者推送，订阅者收到通知后更改自己的数
据。

详细资料可以参考：
《Vue.js 双向绑定的实现原理》

110. Object.defineProperty 介绍?

Object.defineProperty 函数一共有三个参数，第一个参数是需要定义属性的对象，第二个参数是需要定义的属性，第三个是该属性描述符。

一个属性的描述符有四个属性，分别是 value 属性的值，writable 属性是否可写，enumerable 属性是否可枚举，configurable 属性是否可配置修改。

详细资料可以参考：
《Object.defineProperty()》

111. 使用 Object.defineProperty() 来进行数据劫持有什么缺点?

有一些对属性的操作，使用这种方法无法拦截，比如说通过下标方式修改数组数据或者给对象新增属性，vue 内部通过重写函数解决了这个问题。在 Vue3.0 中已经不使用这种方式了，而是通过使用 Proxy 对对象进行代理，从而实现数据劫持。使用 Proxy 的好处是它可以完美的监听到任何方式的数据改变，唯一的缺点是兼容性的问题，因为这是 ES6 的语法。

112. 什么是 Virtual DOM? 为什么 Virtual DOM 比原生 DOM 快?

> 我对 Virtual DOM 的理解是，
>
> 首先对我们将要插入到文档中的 DOM 树结构进行分析，使用 js 对象将其表示出来，比如一个元素对象，包含 TagName、props 和 Children 这些属性。然后我们将这个 js 对象树给保存下来，最后再将 DOM 片段插入到文档中。
>
> 当页面的状态发生改变，我们需要对页面的 DOM 的结构进行调整的时候，我们首先根据变更的状态，重新构建起一棵对象树，然后将这棵新的对象树和旧的对象树进行比较，记录下两棵树的的差异。
>
> 最后将记录的有差异的地方应用到真正的 DOM 树中去，这样视图就更新了。
>
> 我认为 Virtual DOM 这种方法对于我们需要有大量的 DOM 操作的时候，能够很好的提高我们的操作效率，通过在操作前确定需要做的最小修改，尽可能的减少 DOM 操作带来的重流和重绘的影响。其实 Virtual DOM 并不一定比我们真实的操作 DOM 要快，这种方法的目的是为了提高我们开发时的可维护性，在任意的情况下，都能保证一个尽量小的性能消耗去进行操作。

详细资料可以参考：
《Virtual DOM》
《理解 Virtual DOM》
《深度剖析：如何实现一个 Virtual DOM 算法》
《网上都说操作真实 DOM 慢，但测试结果却比 React 更快，为什么？》

113. 如何比较两个 DOM 树的差异?

> 两个树的完全 diff 算法的时间复杂度为 O(n^3) ，但是在前端中，我们很少会跨层级的移动元素，所以我们只需要比较同一层级的元素进行比较，这样就可以将算法的时间复杂度降低为 O(n)。
>
> 算法首先会对新旧两棵树进行一个深度优先的遍历，这样每个节点都会有一个序号。在深度遍历的时候，每遍历到一个节点，我们就将这个节点和新的树中的节点进行比较，如果有差异，则将这个差异记录到一个对象中。
>
> 在对列表元素进行对比的时候，由于 TagName 是重复的，所以我们不能使用这个来对比。我们需要给每一个子节点加上一个 key，列表对比的时候使用 key 来进行比较，这样我们才能够复用老的 DOM 树上的节点。

114. 什么是 requestAnimationFrame ?

详细资料可以参考：
《你需要知道的 requestAnimationFrame》
《CSS3 动画那么强，requestAnimationFrame 还有毛线用？》

115. 谈谈你对 webpack 的看法

> 我当时使用 webpack 的一个最主要原因是为了简化页面依赖的管理，并且通过将其打包为一个文件来降低页面加载时请求的资源
> 数。
>
> 我认为 webpack 的主要原理是，它将所有的资源都看成是一个模块，并且把页面逻辑当作一个整体，通过一个给定的入口文件，webpack 从这个文件开始，找到所有的依赖文件，将各个依赖文件模块通过 loader 和 plugins 处理后，然后打包在一起，最后输出一个浏览器可识别的 JS 文件。
>
> Webpack 具有五个核心的概念，分别是 Entry（入口）、Output（输出）、loader 、 Plugins（插件）和mode（模式）。
>
> Entry 是 webpack 的入口起点，它指示 webpack 应该从哪个模块开始着手，来作为其构建内部依赖图的开始。

Output 属性告诉 webpack 在哪里输出它所创建的打包文件，也可指定打包文件的名称，默认位置为 ./dist。

loader 可以理解为 webpack 的编译器，它使得 webpack 可以处理一些非 JavaScript 文件。在对 loader 进行配置的时候，test 属性，标志有哪些后缀的文件应该被处理，是一个正则表达式。use 属性，指定 test 类型的文件应该使用哪个 loader 进行预处理。常用的 loader 有 css-loader、style-loader 等。

插件可以用于执行范围更广的任务，包括打包、优化、压缩、搭建服务器等等，要使用一个插件，一般是先使用 npm 包管理器进行安装，然后在配置文件中引入，最后将其实例化后传递给 plugins 数组属性。

使用 webpack 的确能够提供我们对于项目的管理，但是它的缺点就是调试和配置起来太麻烦了。但现在 webpack4.0 的免配置一定程度上解决了这个问题。但是我感觉对我来说，就是一个黑盒，很多时候出现了问题，没有办法很好的定位。

详细资料可以参考：
《不聊 webpack 配置，来说说它的原理》
《前端工程化——构建工具选型：grunt、gulp、webpack》
《浅入浅出 webpack》
《前端构建工具发展及其比较》

116. offsetWidth/offsetHeight,clientWidth/clientHeight 与 scrollWidth/scrollHeight 的区别?

clientWidth/clientHeight 返回的是元素的内部宽度，它的值只包含 content + padding，如果有滚动条，不包含滚动条。
clientTop 返回的是上边框的宽度。
clientLeft 返回的左边框的宽度。

offsetWidth/offsetHeight 返回的是元素的布局宽度，它的值包含 content + padding + border 包含了滚动条。
offsetTop 返回的是当前元素相对于其 offsetParent 元素的顶部的距离。（即元素的定位Top值）
offsetLeft 返回的是当前元素相对于其 offsetParent 元素的左部的距离。（即元素的定位Left值）

scrollWidth/scrollHeight 返回值包含 content + padding + 溢出内容的尺寸。
scrollTop 属性返回的是一个元素的内容垂直滚动的像素数。
scrollLeft 属性返回的是元素滚动条到元素左边的距离。

详细资料可以参考：
《最全的获取元素宽高及位置的方法》
《用 Javascript 获取页面元素的位置》

117. 谈一谈你理解的函数式编程?

简单说，"函数式编程"是一种"编程范式"（programming paradigm），也就是如何编写程序的方法论。

它具有以下特性：闭包和高阶函数、惰性计算、递归、函数是"第一等公民"、只用"表达式"。

详细资料可以参考：
《函数式编程初探》

118. 异步编程的实现方式?

相关资料:

回调函数
优点：简单、容易理解
缺点：不利于维护，代码耦合高

事件监听（采用时间驱动模式，取决于某个事件是否发生）：
优点：容易理解，可以绑定多个事件，每个事件可以指定多个回调函数
缺点：事件驱动型，流程不够清晰

发布/订阅（观察者模式）
类似于事件监听，但是可以通过‘消息中心’，了解现在有多少发布者，多少订阅者

Promise 对象
优点：可以利用 then 方法，进行链式写法；可以书写错误时的回调函数；
缺点：编写和理解，相对比较难

Generator 函数
优点：函数体内外的数据交换、错误处理机制
缺点：流程管理不方便

async 函数
优点：内置执行器、更好的语义、更广的适用性、返回的是 Promise、结构清晰。
缺点：错误处理机制

回答:

js 中的异步机制可以分为以下几种：

第一种最常见的是使用回调函数的方式，使用回调函数的方式有一个缺点是，多个回调函数嵌套的时候会造成回调函数地狱，上下两层的回调函数间的代码耦合度太高，不利于代码的可维护。

第二种是 Promise 的方式，使用 Promise 的方式可以将嵌套的回调函数作为链式调用。但是使用这种方法，有时会造成多个 then 的链式调用，可能会造成代码的语义不够明确。

第三种是使用 generator 的方式，它可以在函数的执行过程中，将函数的执行权转移出去，在函数外部我们还可以将执行权转移回来。当我们遇到异步函数执行的时候，将函数执行权转移出去，当异步函数执行完毕的时候我们再将执行权给转移回来。因此我们在 generator 内部对于异步操作的方式，可以以同步的顺序来书写。使用这种方式我们需要考虑的问题是何时将函数的控制权转移回来，因此我们需要有一个自动执行 generator 的机制，比如说 co 模块等方式来实现 generator 的自动执行。

第四种是使用 async 函数的形式，async 函数是 generator 和 promise 实现的一个自动执行的语法糖，它内部自带执行器，当函数内部执行到一个 await 语句的时候，如果语句返回一个 promise 对象，那么函数将会等待 promise 对象的状态变为 resolve 后再继续向下执行。因此我们可以将异步逻辑，转化为同步的顺序来书写，并且这个函数可以自动执行。

119. Js 动画与 CSS 动画区别及相应实现

CSS3 的动画的优点

在性能上会稍微好一些，浏览器会对 CSS3 的动画做一些优化
代码相对简单

缺点

在动画控制上不够灵活
兼容性不好

JavaScript 的动画正好弥补了这两个缺点，控制能力很强，可以单帧的控制、变换，同时写得好完全可以兼容 IE6，并且功能强大。对于一些复杂控制的动画，使用 javascript 会比较靠谱。而在实现一些小的交互动效的时候，就多考虑考虑 CSS 吧

120. get 请求传参长度的误区

误区：我们经常说 get 请求参数的大小存在限制，而 post 请求的参数大小是无限制的。

实际上 HTTP 协议从未规定 GET/POST 的请求长度限制是多少。对 get 请求参数的限制是来源与浏览器或 web 服务器，浏览器或 web 服务器限制了 url 的长度。为了明确这个概念，我们必须再次强调下面几点:

- 1.HTTP 协议未规定 GET 和 POST 的长度限制
- 2.GET 的最大长度显示是因为浏览器和 web 服务器限制了 URI 的长度
- 3.不同的浏览器和 WEB 服务器，限制的最大长度不一样
- 4.要支持 IE，则最大长度为 2083byte，若只支持 Chrome，则最大长度 8182byte

121. URL 和 URI 的区别?

```
URI: Uniform Resource Identifier        指的是统一资源标识符
URL: Uniform Resource Location          指的是统一资源定位符
URN: Universal Resource Name            指的是统一资源名称
```

URI 指的是统一资源标识符，用唯一的标识来确定一个资源，它是一种抽象的定义，也就是说，不管使用什么方法来定义，只要能唯一的标识一个资源，就可以称为 URI。

URL 指的是统一资源定位符，URN 指的是统一资源名称。URL 和 URN 是 URI 的子集，URL 可以理解为使用地址来标识资源，URN 可以理解为使用名称来标识资源。

详细资料可以参考:
《HTTP 协议中 URI 和 URL 有什么区别？》
《你知道 URL、URI 和 URN 三者之间的区别吗？》
《URI、URL 和 URN 的区别》

122. get 和 post 请求在缓存方面的区别

相关知识点:

get 请求类似于查找的过程，用户获取数据，可以不用每次都与数据库连接，所以可以使用缓存。

post 不同，post 做的一般是修改和删除的工作，所以必须与数据库交互，所以不能使用缓存。因此 get 请求适合于请求缓存。

回答:

缓存一般只适用于那些不会更新服务端数据的请求。一般 `get` 请求都是查找请求，不会对服务器资源数据造成修改，而 `post` 请求一般都会对服务器数据造成修改，所以，一般会对 `get` 请求进行缓存，很少会对 `post` 请求进行缓存。

详细资料可以参考：
《HTML 关于 post 和 get 的区别以及缓存问题的理解》

123. 图片的懒加载和预加载

相关知识点：

预加载：提前加载图片，当用户需要查看时可直接从本地缓存中渲染。

懒加载：懒加载的主要目的是作为服务器前端的优化，减少请求数或延迟请求数。

两种技术的本质：两者的行为是相反的，一个是提前加载，一个是迟缓甚至不加载。 懒加载对服务器前端有一定的缓解压力作用，预加载则会增加服务器前端压力。

回答：

懒加载也叫延迟加载，指的是在长网页中延迟加载图片的时机，当用户需要访问时，再去加载，这样可以提高网站的首屏加载速度，提升用户的体验，并且可以减少服务器的压力。它适用于图片很多，页面很长的电商网站的场景。懒加载的实现原理是，将页面上的图片的 `src` 属性设置为空字符串，将图片的真实路径保存在一个自定义属性中，当页面滚动的时候，进行判断，如果图片进入页面可视区域内，则从自定义属性中取出真实路径赋值给图片的 `src` 属性，以此来实现图片的延迟加载。

预加载指的是将所需的资源提前请求加载到本地，这样后面在需要用到时就直接从缓存取资源。通过预加载能够减少用户的等待时间，提高用户的体验。我了解的预加载的最常用的方式是使用 `js` 中的 `image` 对象，通过为 `image` 对象来设置 `scr` 属性，来实现图片的预加载。

这两种方式都是提高网页性能的方式，两者主要区别是一个是提前加载，一个是迟缓甚至不加载。懒加载对服务器前端有一定的缓解压力作用，预加载则会增加服务器前端压力。

详细资料可以参考：
《懒加载和预加载》
《网页图片加载优化方案》
《基于用户行为的图片等资源预加载》

124. mouseover 和 mouseenter 的区别?

当鼠标移动到元素上时就会触发 `mouseenter` 事件，类似 `mouseover`，它们两者之间的差别是 `mouseenter` 不会冒泡。

由于 `mouseenter` 不支持事件冒泡，导致在一个元素的子元素上进入或离开的时候会触发其 `mouseover` 和 `mouseout` 事件，但是却不会触发 `mouseenter` 和 `mouseleave` 事件。
事件委派必须使用mouseover和mouseout这一对。

详细资料可以参考：
《mouseenter 与 mouseover 为何这般纠缠不清？》

125. js 拖拽功能的实现

相关知识点：

首先是三个事件，分别是 mousedown，mousemove，mouseup
当鼠标点击按下的时候，需要一个 tag 标识此时已经按下，可以执行 mousemove 里面的具体方法。
clientX，clientY 标识的是鼠标的坐标，分别标识横坐标和纵坐标，并且我们用 offsetX 和 offsetY 来表示
元素的初始坐标，移动的距离应该是：
鼠标移动时候的坐标-鼠标按下去时候的坐标。
也就是说定位信息为：
鼠标移动时候的坐标-鼠标按下去时候的坐标+元素初始情况下的 offetLeft.

回答：

一个元素的拖拽过程，我们可以分为三个步骤，第一步是鼠标按下目标元素，第二步是鼠标保持按下的状态移动鼠标，第三步是鼠
标抬起，拖拽过程结束。

这三步分别对应了三个事件，mousedown 事件，mousemove 事件和 mouseup 事件。只有在鼠标按下的状态移动鼠标我们才会
执行拖拽事件，因此我们需要在 mousedown 事件中设置一个状态来标识鼠标已经按下，然后在 mouseup 事件中再取消这个状
态。在 mousedown 事件中我们首先应该判断，目标元素是否为拖拽元素，如果是拖拽元素，我们就设置状态并且保存这个时候鼠
标的位置。然后在 mousemove 事件中，我们通过判断鼠标现在的位置和以前位置的相对移动，来确定拖拽元素在移动中的坐标。
最后 mouseup 事件触发后，清除状态，结束拖拽事件。

详细资料可以参考：
《原生 js 实现拖拽功能基本思路》

126. 为什么使用 setTimeout 实现 setInterval？怎么模拟？

相关知识点：

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

function mySetInterval(fn, timeout) {
  // 控制器，控制定时器是否继续执行
  var timer = {
    flag: true
  };

  // 设置递归函数，模拟定时器执行。
  function interval() {
    if (timer.flag) {
      fn();
      setTimeout(interval, timeout);
    }
  }

  // 启动定时器
  setTimeout(interval, timeout);

  // 返回控制器
```

```
  return timer;
}
```

回答：

`setInterval` 的作用是每隔一段指定时间执行一个函数，但是这个执行不是真的到了时间立即执行，它真正的作用是每隔一段时间将事件加入事件队列中去，只有当当前的执行栈为空的时候，才能去从事件队列中取出事件执行。所以可能会出现这样的情况，就是当前执行栈执行的时间很长，导致事件队列里边积累多个定时器加入的事件，当执行栈结束的时候，这些事件会依次执行，因此就不能到间隔一段时间执行的效果。

针对 `setInterval` 的这个缺点，我们可以使用 `setTimeout` 递归调用来模拟 `setInterval`，这样我们就确保了只有一个事件结束了，我们才会触发下一个定时器事件，这样解决了 `setInterval` 的问题。

详细资料可以参考：
《用 setTimeout 实现 setInterval》
《setInterval 有什么缺点？》

127. let 和 const 的注意点?

- 1.声明的变量只在声明时的代码块内有效
- 2.不存在声明提升
- 3.存在暂时性死区，如果在变量声明前使用，会报错
- 4.不允许重复声明，重复声明会报错

128. 什么是 rest 参数?

`rest` 参数（形式为`...`变量名），用于获取函数的多余参数。

129. 什么是尾调用，使用尾调用有什么好处?

尾调用指的是函数的最后一步调用另一个函数。我们代码执行是基于执行栈的，所以当我们在一个函数里调用另一个函数时，我们会保留当前的执行上下文，然后再新建另外一个执行上下文加入栈中。使用尾调用的话，因为已经是函数的最后一步，所以这个时候我们可以不必再保留当前的执行上下文，从而节省了内存，这就是尾调用优化。但是 `ES6` 的尾调用优化只在严格模式下开启，正常模式是无效的。

130. Symbol 类型的注意点?

- 1.Symbol 函数前不能使用 new 命令，否则会报错。
- 2.Symbol 函数可以接受一个字符串作为参数，表示对 Symbol 实例的描述，主要是为了在控制台显示，或者转为字符串时，比较容易区分。
- 3.Symbol 作为属性名，该属性不会出现在 for...in、for...of 循环中，也不会被 Object.keys()、Object.getOwnPropertyNames()、JSON.stringify() 返回。
- 4.Object.getOwnPropertySymbols 方法返回一个数组，成员是当前对象的所有用作属性名的 Symbol 值。
- 5.Symbol.for 接受一个字符串作为参数，然后搜索有没有以该参数作为名称的 Symbol 值。如果有，就返回这个 Symbol 值，否则就新建并返回一个以该字符串为名称的 Symbol 值。
- 6.Symbol.keyFor 方法返回一个已登记的 Symbol 类型值的 key。

131. Set 和 WeakSet 结构?

- 1.ES6 提供了新的数据结构 Set。它类似于数组，但是成员的值都是唯一的，没有重复的值。
- 2.WeakSet 结构与 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，

132. Map 和 WeakMap 结构?

- 1.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。
- 2.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。

133. 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"。

134. Reflect 对象创建目的?

- 1.将 Object 对象的一些明显属于语言内部的方法（比如 Object.defineProperty，放到 Reflect 对象上。
- 2.修改某些 Object 方法的返回结果，让其变得更合理。
- 3.让 Object 操作都变成函数行为。
- 4.Reflect 对象的方法与 Proxy 对象的方法一一对应，只要是 Proxy 对象的方法，就能在 Reflect 对象上找到对应的方法。这就让 Proxy 对象可以方便地调用对应的 Reflect 方法，完成默认行为，作为修改行为的基础。也就是说，不管 Proxy 怎么修改默认行为，你总可以在 Reflect 上获取默认行为。

135. require 模块引入的查找方式?

```
当 Node 遇到 require(X) 时，按下面的顺序处理。

（1）如果 X 是内置模块（比如 require('http')）
   a. 返回该模块。
   b. 不再继续执行。

（2）如果 X 以 "./" 或者 "/" 或者 "../" 开头
   a. 根据 X 所在的父模块，确定 X 的绝对路径。
   b. 将 X 当成文件，依次查找下面文件，只要其中有一个存在，就返回该文件，不再继续执行。
    X
    X.js
    X.json
    X.node

   c. 将 X 当成目录，依次查找下面文件，只要其中有一个存在，就返回该文件，不再继续执行。
    X/package.json（main字段）
    X/index.js
    X/index.json
```

```
    X/index.node

（3）如果 X 不带路径
   a. 根据 X 所在的父模块，确定 X 可能的安装目录。
   b. 依次在每个目录中，将 X 当成文件名或目录名加载。

（4）抛出 "not found"
```

详细资料可以参考：
《require() 源码解读》

136. 什么是 Promise 对象，什么是 Promises/A+ 规范?

Promise 对象是异步编程的一种解决方案，最早由社区提出。Promises/A+ 规范是 JavaScript Promise 的标准，规定了一个 Promise 所必须具有的特性。

Promise 是一个构造函数，接收一个函数作为参数，返回一个 Promise 实例。一个 Promise 实例有三种状态，分别是 pending、resolved 和 rejected，分别代表了进行中、已成功和已失败。实例的状态只能由 pending 转变 resolved 或者 rejected 状态，并且状态一经改变，就凝固了，无法再被改变了。状态的改变是通过 resolve() 和 reject() 函数来实现的，我们
可以在异步操作结束后调用这两个函数改变 Promise 实例的状态，它的原型上定义了一个 then 方法，使用这个 then 方法可以为两个状态的改变注册回调函数。这个回调函数属于微任务，会在本轮事件循环的末尾执行。

详细资料可以参考：
《Promises/A+ 规范》
《Promise》

137. 手写一个 Promise

```
const PENDING = "pending";
const RESOLVED = "resolved";
const REJECTED = "rejected";

function MyPromise(fn) {
  // 保存初始化状态
  var self = this;

  // 初始化状态
  this.state = PENDING;

  // 用于保存 resolve 或者 rejected 传入的值
  this.value = null;

  // 用于保存 resolve 的回调函数
  this.resolvedCallbacks = [];

  // 用于保存 reject 的回调函数
  this.rejectedCallbacks = [];

  // 状态转变为 resolved 方法
  function resolve(value) {
    // 判断传入元素是否为 Promise 值，如果是，则状态改变必须等待前一个状态改变后再进行改变
    if (value instanceof MyPromise) {
      return value.then(resolve, reject);
    }
```

```
    // 保证代码的执行顺序为本轮事件循环的末尾
    setTimeout(() => {
      // 只有状态为 pending 时才能转变，
      if (self.state === PENDING) {
        // 修改状态
        self.state = RESOLVED;

        // 设置传入的值
        self.value = value;

        // 执行回调函数
        self.resolvedCallbacks.forEach(callback => {
          callback(value);
        });
      }
    }, 0);
  }

  // 状态转变为 rejected 方法
  function reject(value) {
    // 保证代码的执行顺序为本轮事件循环的末尾
    setTimeout(() => {
      // 只有状态为 pending 时才能转变
      if (self.state === PENDING) {
        // 修改状态
        self.state = REJECTED;

        // 设置传入的值
        self.value = value;

        // 执行回调函数
        self.rejectedCallbacks.forEach(callback => {
          callback(value);
        });
      }
    }, 0);
  }

  // 将两个方法传入函数执行
  try {
    fn(resolve, reject);
  } catch (e) {
    // 遇到错误时，捕获错误，执行 reject 函数
    reject(e);
  }
}

MyPromise.prototype.then = function(onResolved, onRejected) {
  // 首先判断两个参数是否为函数类型，因为这两个参数是可选参数
  onResolved =
    typeof onResolved === "function" ? onResolved : function(value) { return
value; };

  onRejected =
    typeof onRejected === "function" ? onRejected : function(error) { throw
error; };
```

```
  // 如果是等待状态，则将函数加入对应列表中
  if (this.state === PENDING) {
    this.resolvedCallbacks.push(onResolved);
    this.rejectedCallbacks.push(onRejected);
  }

  // 如果状态已经凝固，则直接执行对应状态的函数

  if (this.state === RESOLVED) {
    onResolved(this.value);
  }

  if (this.state === REJECTED) {
    onRejected(this.value);
  }
};
```

138. 如何检测浏览器所支持的最小字体大小?

用 JS 设置 DOM 的字体为某一个值，然后再取出来，如果值设置成功，就说明支持。

139. 怎么做 JS 代码 Error 统计?

error 统计使用浏览器的 window.error 事件。

140. 单例模式模式是什么?

单例模式保证了全局只有一个实例来被访问。比如说常用的如弹框组件的实现和全局状态的实现。

141. 策略模式是什么?

策略模式主要是用来将方法的实现和方法的调用分离开，外部通过不同的参数可以调用不同的策略。我主要在 MVP 模式解耦的时候
用来将视图层的方法定义和方法调用分离。

142. 代理模式是什么?

代理模式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。比如说常见的事件代理。

143. 中介者模式是什么?

中介者模式指的是，多个对象通过一个中介者进行交流，而不是直接进行交流，这样能够将通信的各个对象解耦。

144. 适配器模式是什么?

适配器用来解决两个接口不兼容的情况，不需要改变已有的接口，通过包装一层的方式实现两个接口的正常协作。假如我们需要一种
新的接口返回方式，但是老的接口由于在太多地方已经使用了，不能随意更改，这个时候就可以使用适配器模式。比如我们需要一种
自定义的时间返回格式，但是我们又不能对 js 时间格式化的接口进行修改，这个时候就可以使用适配器模式。

更多关于设计模式的资料可以参考：
《前端面试之道》
《JavaScript 设计模式》
《JavaScript 中常见设计模式整理》

145. 观察者模式和发布订阅模式有什么不同?

发布订阅模式其实属于广义上的观察者模式

在观察者模式中，观察者需要直接订阅目标事件。在目标发出内容改变的事件后，直接接收事件并作出响应。

而在发布订阅模式中，发布者和订阅者之间多了一个调度中心。调度中心一方面从发布者接收事件，另一方面向订阅者发布事件，订阅者需要在调度中心中订阅事件。通过调度中心实现了发布者和订阅者关系的解耦。使用发布订阅者模式更利于我们代码的可维护性。

详细资料可以参考：
《观察者模式和发布订阅模式有什么不同？》

146. Vue 的生命周期是什么?

Vue 的生命周期指的是组件从创建到销毁的一系列的过程，被称为 Vue 的生命周期。通过提供的 Vue 在生命周期各个阶段的钩子函数，我们可以很好的在 Vue 的各个生命阶段实现一些操作。

147. Vue 的各个生命阶段是什么?

Vue 一共有8个生命阶段，分别是创建前、创建后、加载前、加载后、更新前、更新后、销毁前和销毁后，每个阶段对应了一个生命周期的钩子函数。

（1）beforeCreate 钩子函数，在实例初始化之后，在数据监听和事件配置之前触发。因此在这个事件中我们是获取不到 data 数据的。

（2）created 钩子函数，在实例创建完成后触发，此时可以访问 data、methods 等属性。但这个时候组件还没有被挂载到页面中去，所以这个时候访问不到 $el 属性。一般我们可以在这个函数中进行一些页面初始化的工作，比如通过 ajax 请求数据来对页面进行初始化。

（3）beforeMount 钩子函数，在组件被挂载到页面之前触发。在 beforeMount 之前，会找到对应的 template，并编译成 render 函数。

（4）mounted 钩子函数，在组件挂载到页面之后触发。此时可以通过 DOM API 获取到页面中的 DOM 元素。

（5）beforeUpdate 钩子函数，在响应式数据更新时触发，发生在虚拟 DOM 重新渲染和打补丁之前，这个时候我们可以对可能会被移除的元素做一些操作，比如移除事件监听器。

（6）updated 钩子函数，虚拟 DOM 重新渲染和打补丁之后调用。

（7）beforeDestroy 钩子函数，在实例销毁之前调用。一般在这一步我们可以销毁定时器、解绑全局事件等。

（8）destroyed 钩子函数，在实例销毁之后调用，调用后，Vue 实例中的所有东西都会解除绑定，所有的事件监听器会被移除，所有的子实例也会被销毁。

当我们使用 keep-alive 的时候，还有两个钩子函数，分别是 activated 和 deactivated 。用 keep-alive 包裹的组件在切换时不会进行销毁，而是缓存到内存中并执行 deactivated 钩子函数，命中缓存渲染后会执行 actived 钩子函数。

详细资料可以参考：
《vue 生命周期深入》
《Vue 实例》

148. Vue 组件间的参数传递方式?

（1）父子组件间通信

第一种方法是子组件通过 props 属性来接受父组件的数据，然后父组件在子组件上注册监听事件，子组件通过 emit 触发事
件来向父组件发送数据。

第二种是通过 ref 属性给子组件设置一个名字。父组件通过 $refs 组件名来获得子组件，子组件通过 $parent 获得父组
件，这样也可以实现通信。

第三种是使用 provider/inject，在父组件中通过 provider 提供变量，在子组件中通过 inject 来将变量注入到组件
中。不论子组件有多深，只要调用了 inject 那么就可以注入 provider 中的数据。

（2）兄弟组件间通信

第一种是使用 eventBus 的方法，它的本质是通过创建一个空的 Vue 实例来作为消息传递的对象，通信的组件引入这个实
例，通信的组件通过在这个实例上监听和触发事件，来实现消息的传递。

第二种是通过 $parent.$refs 来获取到兄弟组件，也可以进行通信。

（3）任意组件之间

使用 eventBus ，其实就是创建一个事件中心，相当于中转站，可以用它来传递事件和接收事件。

如果业务逻辑复杂，很多组件之间需要同时处理一些公共的数据，这个时候采用上面这一些方法可能不利于项目的维护。这个时候
可以使用 vuex ，vuex 的思想就是将这一些公共的数据抽离出来，将它作为一个全局的变量来管理，然后其他组件就可以对这个
公共数据进行读写操作，这样达到了解耦的目的。

详细资料可以参考：
《VUE 组件之间数据传递全集》

149. computed 和 watch 的差异?

（1）computed 是计算一个新的属性，并将该属性挂载到 Vue 实例上，而 watch 是监听已经存在且已挂载到 Vue 实例上的数据，所以用 watch 同样可以监听 computed 计算属性的变化。

（2）computed 本质是一个惰性求值的观察者，具有缓存性，只有当依赖变化后，第一次访问 computed 属性，才会计算新的值。而 watch 则是当数据发生变化便会调用执行函数。

（3）从使用场景上说，computed 适用一个数据被多个数据影响，而 watch 适用一个数据影响多个数据。

详细资料可以参考：
《做面试的不倒翁：浅谈 Vue 中 computed 实现原理》
《深入理解 Vue 的 watch 实现原理及其实现方式》

150. vue-router 中的导航钩子函数

（1）全局的钩子函数 beforeEach 和 afterEach

beforeEach 有三个参数，to 代表要进入的路由对象，from 代表离开的路由对象。next 是一个必须要执行的函数，如果不传参数，那就执行下一个钩子函数，如果传入 false，则终止跳转，如果传入一个路径，则导航到对应的路由，如果传入 error ，则导航终止，error 传入错误的监听函数。

（2）单个路由独享的钩子函数 beforeEnter，它是在路由配置上直接进行定义的。

（3）组件内的导航钩子主要有这三种：beforeRouteEnter、beforeRouteUpdate、beforeRouteLeave。它们是直接在路由组
件内部直接进行定义的。

详细资料可以参考：
《导航守卫》

151. $route 和 $router 的区别?

$route 是“路由信息对象”，包括 path，params，hash，query，fullPath，matched，name 等路由信息参数。而 $router 是“路由实例”对象包括了路由的跳转方法，钩子函数等。

152. vue 常用的修饰符?

.prevent: 提交事件不再重载页面；.stop: 阻止单击事件冒泡；.self: 当事件发生在该元素本身而不是子元素的时候会触发；

153. vue 中 key 值的作用?

vue 中 key 值的作用可以分为两种情况来考虑。

第一种情况是 v-if 中使用 key。由于 Vue 会尽可能高效地渲染元素，通常会复用已有元素而不是从头开始渲染。因此当我们使用 v-if 来实现元素切换的时候，如果切换前后含有相同类型的元素，那么这个元素就会被复用。如果是相同的 input 元素，那么切换前后用户的输入不会被清除掉，这样是不符合需求的。因此我们可以通过使用 key 来唯一的标识一个元素，这个情况下，使用 key 的元素不会被复用。这个时候 key 的作用是用来标识一个独立的元素。

第二种情况是 v-for 中使用 key。用 v-for 更新已渲染过的元素列表时，它默认使用“就地复用”的策略。如果数据项的顺序发生了改变，Vue 不会移动 DOM 元素来匹配数据项的顺序，而是简单复用此处的每个元素。因此通过为每个列表项提供一个 key 值，来以便 Vue 跟踪元素的身份，从而高效的实现复用。这个时候 key 的作用是为了高效的更新渲染虚拟 DOM。

详细资料可以参考：
《Vue 面试中，经常会被问到的面试题 Vue 知识点整理》
《Vue2.0 v-for 中 :key 到底有什么用？》
《vue 中 key 的作用》

154. computed 和 watch 区别?

computed 是计算属性，依赖其他属性计算值，并且 computed 的值有缓存，只有当计算值变化才会返回内容。

watch 监听到值的变化就会执行回调，在回调中可以进行一些逻辑操作。

155. keep-alive 组件有什么作用?

如果你需要在组件切换的时候，保存一些组件的状态防止多次渲染，就可以使用 keep-alive 组件包裹需要保存的组件。

156. vue 中 mixin 和 mixins 区别?

mixin 用于全局混入，会影响到每个组件实例。

mixins 应该是我们最常使用的扩展组件的方式了。如果多个组件中有相同的业务逻辑，就可以将这些逻辑剥离出来，通过 mixins 混入代码，比如上拉下拉加载数据这种逻辑等等。另外需要注意的是 mixins 混入的钩子函数会先于组件内的钩子函数执行，并且在遇到同名选项的时候也会有选择性的进行合并

详细资料可以参考：
《前端面试之道》
《混入》

157. 开发中常用的几种 Content-Type ?

（1）application/x-www-form-urlencoded

浏览器的原生 form 表单，如果不设置 enctype 属性，那么最终就会以 application/x-www-form-urlencoded 方式提交数据。该种方式提交的数据放在 body 里面，数据按照 key1=val1&key2=val2 的方式进行编码，key 和 val 都进行了 URL
转码。

（2）multipart/form-data

该种方式也是一个常见的 POST 提交方式，通常表单上传文件时使用该种方式。

（3）application/json

告诉服务器消息主体是序列化后的 JSON 字符串。

（4）text/xml

该种方式主要用来提交 XML 格式的数据。

详细资料可以参考：
《常用的几种 Content-Type》

158. 如何封装一个 javascript 的类型判断函数?

```
console.log('---------------- 定义检测数据类型的方法 ----------------------');
function checkoutType(target) {
  return Object.prototype.toString.call(target).slice(8, -1);
}
```

详细资料可以参考：
《JavaScript 专题之类型判断(上)》

159. 如何判断一个对象是否为空对象?

```
function checkNullObj(obj) {
  return Object.keys(obj).length === 0;
}
```

详细资料可以参考:
《js 判断一个 object 对象是否为空》

160. 使用闭包实现每隔一秒打印 1,2,3,4

```
// 使用闭包实现
for (var i = 0; i < 5; i++) {
  (function(i) {
    setTimeout(function() {
      console.log(i);
    }, i * 1000);
  })(i);
}

// 使用 let 块级作用域

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

161. 手写一个 jsonp

```
function jsonp(url, params, callback) {
  // 判断是否含有参数
  let queryString = url.indexOf("?") === "-1" ? "?" : "&";

  // 添加参数
  for (var k in params) {
    if (params.hasOwnProperty(k)) {
      queryString += k + "=" + params[k] + "&";
    }
  }

  // 处理回调函数名
  let random = Math.random()
      .toString()
      .replace(".", ""),
    callbackName = "myJsonp" + random;

  // 添加回调函数
  queryString += "callback=" + callbackName;

  // 构建请求
  let scriptNode = document.createElement("script");
  scriptNode.src = url + queryString;

  window[callbackName] = function() {
    // 调用回调函数
```

```
    callback(...arguments);

    // 删除这个引入的脚本
    document.getElementsByTagName("head")[0].removeChild(scriptNode);
  };

  // 发起请求
  document.getElementsByTagName("head")[0].appendChild(scriptNode);
}
```

详细资料可以参考：
《原生 jsonp 具体实现》
《jsonp 的原理与实现》

162. 手写一个观察者模式?

```
var events = (function() {
  var topics = {};

  return {
    // 注册监听函数
    subscribe: function(topic, handler) {
      if (!topics.hasOwnProperty(topic)) {
        topics[topic] = [];
      }
      topics[topic].push(handler);
    },

    // 发布事件，触发观察者回调事件
    publish: function(topic, info) {
      if (topics.hasOwnProperty(topic)) {
        topics[topic].forEach(function(handler) {
          handler(info);
        });
      }
    },

    // 移除主题的一个观察者的回调事件
    remove: function(topic, handler) {
      if (!topics.hasOwnProperty(topic)) return;

      var handlerIndex = -1;
      topics[topic].forEach(function(item, index) {
        if (item === handler) {
          handlerIndex = index;
        }
      });

      if (handlerIndex >= 0) {
        topics[topic].splice(handlerIndex, 1);
      }
    },

    // 移除主题的所有观察者的回调事件
    removeAll: function(topic) {
      if (topics.hasOwnProperty(topic)) {
        topics[topic] = [];
```

```
      }
    }
  };
})();
```

详细资料可以参考：
《JS 事件模型》

163. EventEmitter 实现

```
class EventEmitter {
  constructor() {
    this.events = {};
  }

  on(event, callback) {
    let callbacks = this.events[event] || [];
    callbacks.push(callback);
    this.events[event] = callbacks;

    return this;
  }

  off(event, callback) {
    let callbacks = this.events[event];
    this.events[event] = callbacks && callbacks.filter(fn => fn !== callback);

    return this;
  }

  emit(event, ...args) {
    let callbacks = this.events[event];
    callbacks.forEach(fn => {
      fn(...args);
    });

    return this;
  }

  once(event, callback) {
    let wrapFun = function(...args) {
      callback(...args);

      this.off(event, wrapFun);
    };
    this.on(event, wrapFun);

    return this;
  }
}
```

164. 一道常被人轻视的前端 JS 面试题

```
function Foo() {
  getName = function() {
    alert(1);
  };
  return this;
}
Foo.getName = function() {
  alert(2);
};
Foo.prototype.getName = function() {
  alert(3);
};
var getName = function() {
  alert(4);
};
function getName() {
  alert(5);
}

//请写出以下输出结果：
Foo.getName(); // 2
getName(); // 4
Foo().getName(); // 1
getName(); // 1
new Foo.getName(); // 2
new Foo().getName(); // 3
new new Foo().getName(); // 3
```

详细资料可以参考：
《前端程序员经常忽视的一个 JavaScript 面试题》
《一道考察运算符优先级的 JavaScript 面试题》
《一道常被人轻视的前端 JS 面试题》

165. 如何确定页面的可用性时间，什么是 Performance API?

Performance API 用于精确度量、控制、增强浏览器的性能表现。这个 API 为测量网站性能，提供以前没有办法做到的精度。

使用 getTime 来计算脚本耗时的缺点，首先，getTime方法（以及 Date 对象的其他方法）都只能精确到毫秒级别（一秒的千分之一），想要得到更小的时间差别就无能为力了。其次，这种写法只能获取代码运行过程中的时间进度，无法知道一些后台事件的时间进度，比如浏览器用了多少时间从服务器加载网页。

为了解决这两个不足之处，ECMAScript 5引入“高精度时间戳”这个 API，部署在 performance 对象上。它的精度可以达到1毫秒
的千分之一（1秒的百万分之一）。

navigationStart：当前浏览器窗口的前一个网页关闭，发生 unload 事件时的 Unix 毫秒时间戳。如果没有前一个网页，则等于 fetchStart 属性。

loadEventEnd：返回当前网页 load 事件的回调函数运行结束时的 Unix 毫秒时间戳。如果该事件还没有发生，返回 0。

根据上面这些属性，可以计算出网页加载各个阶段的耗时。比如，网页加载整个过程的耗时的计算方法如下：

```
var t = performance.timing;
var pageLoadTime = t.loadEventEnd - t.navigationStart;
```

详细资料可以参考：
《Performance API》

166. js 中的命名规则

（1）第一个字符必须是字母、下划线（_）或美元符号（$）
（2）余下的字符可以是下划线、美元符号或任何字母或数字字符

一般我们推荐使用小驼峰法来对变量名进行命名，因为这样可以与 ECMAScript 内置的函数和对象命名格式保持一致。

详细资料可以参考：
《ECMAScript 变量》

167. js 语句末尾分号是否可以省略?

在 ECMAScript 规范中，语句结尾的分号并不是必需的。但是我们一般最好不要省略分号，因为加上分号一方面有
利于我们代码的可维护性，另一方面也可以避免我们在对代码进行压缩时出现错误。

168. Object.assign()

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。

169. Math.ceil 和 Math.floor

Math.ceil() === 向上取整，函数返回一个大于或等于给定数字的最小整数。

Math.floor() === 向下取整，函数返回一个小于或等于给定数字的最大整数。

170. js for 循环注意点

```
for (var i = 0, j = 0; i < 5, j < 9; i++, j++) {
  console.log(i, j);
}

// 当判断语句含有多个语句时，以最后一个判断语句的值为准，因此上面的代码会执行 10 次。
// 当判断语句为空时，循环会一直进行。
```

171. 一个列表，假设有 100000 个数据，这个该怎么办?

我们需要思考的问题：该处理是否必须同步完成？数据是否必须按顺序完成？

解决办法：

（1）将数据分页，利用分页的原理，每次服务器端只返回一定数目的数据，浏览器每次只对一部分进行加载。

（2）使用懒加载的方法，每次加载一部分数据，其余数据当需要使用时再去加载。

（3）使用数组分块技术，基本思路是为要处理的项目创建一个队列，然后设置定时器每过一段时间取出一部分数据，然后再使用定时器取出下一个要处理的项目进行处理，接着再设置另一个定时器。

172. js 中倒计时的纠偏实现?

在前端实现中我们一般通过 `setTimeout` 和 `setInterval` 方法来实现一个倒计时效果。但是使用这些方法会存在时间偏差的问题，这是由于 `js` 的程序执行机制造成的，`setTimeout` 和 `setInterval` 的作用是隔一段时间将回调事件加入到事件队列中，因此事件并不是立即执行的，它会等到当前执行栈为空的时候再取出事件执行，因此事件等待执行的时间就是造成误差的原因。

一般解决倒计时中的误差的有这样两种办法：

（1）第一种是通过前端定时向服务器发送请求获取最新的时间差，以此来校准倒计时时间。

（2）第二种方法是前端根据偏差时间来自动调整间隔时间的方式来实现的。这一种方式首先是以 `setTimeout` 递归的方式来实现倒计时，然后通过一个变量来记录已经倒计时的秒数。每一次函数调用的时候，首先将变量加一，然后根据这个变量和每次的间隔时间，我们就可以计算出此时无偏差时应该显示的时间。然后将当前的真实时间与这个时间相减，这样我们就可以得到时间的偏差大小，因此我们在设置下一个定时器的间隔大小的时候，我们就从间隔时间中减去这个偏差大小，以此来实现由于程序执行所造成的时间误差的纠正。

详细资料可以参考：
《JavaScript 前端倒计时纠偏实现》

173. 进程间通信的方式?

- 1.管道通信
- 2.消息队列通信
- 3.信号量通信
- 4.信号通信
- 5.共享内存通信
- 6.套接字通信

详细资料可以参考：
《进程间 8 种通信方式详解》
《进程与线程的一个简单解释》

174. 如何查找一篇英文文章中出现频率最高的单词?

```
function findMostWord(article) {
  // 合法性判断
  if (!article) return;

  // 参数处理
  article = article.trim().toLowerCase();

  let wordList = article.match(/[a-z]+/g),
```

```
    visited = [],
    maxNum = 0,
    maxWord = "";

  article = " " + wordList.join("  ") + " ";

  // 遍历判断单词出现次数
  wordList.forEach(function(item) {
    if (visited.indexOf(item) < 0) {

      // 加入 visited
      visited.push(item);

      let word = new RegExp(" " + item + " ", "g"),
        num = article.match(word).length;

      if (num > maxNum) {
        maxNum = num;
        maxWord = item;
      }
    }
  });

  return maxWord + "  " + maxNum;
}
```