1.document load 和 document ready 的区别

页面加载完成有两种事件

1.load 是当页面所有资源全部加载完成后（包括 DOM 文档树，css 文件，js 文件，图片资源等），执行一个函数

问题：如果图片资源较多，加载时间较长，onload 后等待执行的函数需要等待较长时间，所以一些效果可能受到影响

2.$(document).ready()是当 DOM 文档树加载完成后执行一个函数 （不包含图片，css 等）所以会比 load 较快执行。在原生的 js 中不包括 ready()这个方法，只有 load 方法也就是 onload 事件

2. JavaScript 中如何检测一个变量是一个 String 类型？

答案：三种方法（typeof、constructor、Object.prototype.toString.call()）

①typeof

```
typeof('123') === "string" // true
typeof '123' === "string" // true
```

②constructor

```
'123'.constructor === String // true
```

③Object.prototype.toString.call()

```
Object.prototype.toString.call('123') === '[object String]' // true
```

3.请用 js 去除字符串空格？

答案：replace 正则匹配方法、str.trim()方法、JQ 方法：$.trim(str)方法

方法一：replace 正则匹配方法

去除字符串内所有的空格：str = str.replace(/\s*/g,"");

去除字符串内两头的空格：str = str.replace(/^\s*|\s*$/g,"");

去除字符串内左侧的空格：str = str.replace(/^\s*/,"");

去除字符串内右侧的空格：str = str.replace(/(\s*$)/g,"");

```
var str = " 6 6 ";
var str_1 = str.replace(/\s*/g, "");
console.log(str_1); //66
var str = " 6 6 ";
var str_1 = str.replace(/^\s*|\s*$/g, "");
console.log(str_1); //6 6//输出左右侧均无空格
var str = " 6 6 ";
var str_1 = str.replace(/^\s*/, "");
console.log(str_1); //6 6 //输出右侧有空格左侧无空格
var str = " 6 6 ";
var str_1 = str.replace(/(\s*$)/g, "");
console.log(str_1); // 6 6//输出左侧有空格右侧无空格
```

方法二：str.trim()方法

trim()方法是用来删除字符串两端的空白字符并返回，trim 方法并不影响原来的字符串本身，它返回的是一个新的字符串。

缺陷：只能去除字符串两端的空格，不能去除中间的空格

示例：

```
var str = " 6 6 ";
var str_1 = str.trim();
console.log(str_1); //6 6//输出左右侧均无空格
```

方法三：JQ 方法：$.trim(str)方法

$.trim() 函数用于去除字符串两端的空白字符。

注意：$.trim()函数会移除字符串开始和末尾处的所有换行符，空格(包括连续的空格)和制表符。如果这些空白字符在字符串中间时，它们将被保留，不会被移除。

```
var str = " 6 6 ";
var str_1 = $.trim(str);
console.log(str_1); //6 6//输出左右侧均无空格
```

4.js 是一门怎样的语言，它有什么特点

1.脚本语言。JavaScript 是一种解释型的脚本语言,C、C++等语言先编译后执行,而 JavaScript 是在程序的运行过程中逐行进行解释。

2.基于对象。JavaScript 是一种基于对象的脚本语言,它不仅可以创建对象,也能使用现有的对象。

3.简单。JavaScript 语言中采用的是弱类型的变量类型,对使用的数据类型未做出严格的要求,是基于 Java 基本语句和控制的脚本语言,其设计简单紧凑。

4.动态性。JavaScript 是一种采用事件驱动的脚本语言,它不需要经过 Web 服务器就可以对用户的输入做出响应。

5.跨平台性。JavaScript 脚本语言不依赖于操作系统,仅需要浏览器的支持。

5.== 和 === 的不同

答案：==是抽象相等运算符，而===是严格相等运算符。==运算符是在进行必要的类型转换后，再比较。===运算符不会进行类型转换，所以如果两个值不是相同的类型，会直接返回 false。使用==时，可能发生一些特别的事情，例如：

```
1 == "1"; // true                1 == [1]; // true
1 == true; // true               0 == ""; // true
0 == "0"; // true                0 == false; // true
```

如果你对==和===的概念不是特别了解，建议大多数情况下使用===

6.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

2）添加、移除、替换、插入

- appendChild() //添加
- removeChild() //移除
- replaceChild() //替换
- insertBefore() //插入

3）查找

- getElementsByTagName() //通过标签名称
- getElementsByName() //通过元素的 Name 属性的值
- getElementById() //通过元素 Id，唯一性

7.事件委托是什么

答案：利用事件冒泡的原理，让自己的所触发的事件，让他的父元素代替执行！

1、那什么样的事件可以用事件委托，什么样的事件不可以用呢？

- 适合用事件委托的事件：click，mousedown，mouseup，keydown，keyup，keypress。
- 值得注意的是，mouseover 和 mouseout 虽然也有事件冒泡，但是处理它们的时候需要特别的注意，因为需要经常计算它们的位置，处理起来不太容易。
- 不适合的就有很多了，举个例子，mousemove，每次都要计算它的位置，非常不好把控，在不如说 focus，blur 之类的，本身就没用冒泡的特性，自然就不用事件委托了。

2、为什么要用事件委托

- 1.提高性能

```
<ul>
   <li>苹果</li>
   <li>香蕉</li>
   <li>凤梨</li>
</ul>
// good
document.querySelector('ul').onclick = (event) => {
  let target = event.target
  if (target.nodeName === 'LI') {
    console.log(target.innerHTML)
  }
}
// bad
document.querySelectorAll('li').forEach((e) => {
  e.onclick = function() {
    console.log(this.innerHTML)
  }
})
```

- 2.新添加的元素还会有之前的事件。

3、事件冒泡与事件委托的对比

- 事件冒泡：box 内部无论是什么元素，点击后都会触发 box 的点击事件
- 事件委托：可以对 box 内部的元素进行筛选

4、事件委托怎么取索引？

```
    <ul id="ul">
        <li>aaaaaaaa</li>
        <li>事件委托了 点击当前，如何获取 这个点击的下标</li>
        <li>cccccccc</li>
    </ul>
    <script>
        window.onload = function () {
            var oUl = document.getElementById("ul");
            var aLi = oUl.getElementsByTagName("li");
            oUl.onclick = function (ev) {
                var ev = ev || window.event;
                var target = ev.target || ev.srcElement;
                if (target.nodeName.toLowerCase() == "li") {
                    var that = target;
                    var index;
                    for (var i = 0; i < aLi.length; i++)
                        if (aLi[i] === target) index = i;
                    if (index >= 0) alert('我的下标是第' + index + '个');
                    target.style.background = "red";
                }
```

```
            }
        }
    </script>
```
拓展：
- 键盘事件：keydown keypress keyup
- 鼠标事件：mousedown mouseup mousemove mouseout mouseover

8.require 与 import 的区别

答案：两者的加载方式不同、规范不同

第一、两者的加载方式不同，require 是在运行时加载，而 import 是在编译时加载

```
require('./a')(); // a 模块是一个函数，立即执行 a 模块函数
var data = require('./a').data; // a 模块导出的是一个对象
var a = require('./a')[0]; // a 模块导出的是一个数组 ======> 哪都行
import $ from 'jquery';
import * as _ from '_';
import {a,b,c} from './a';
import {default as alias, a as a_a, b, c} from './a'; ======>用在开头
```

第二、规范不同，require 是 CommonJS/AMD 规范，import 是 ESMAScript6+规范

第三、require 特点：社区方案，提供了服务器/浏览器的模块加载方案。非语言层面的标准。只能在运行时确定模块的依赖关系及输入/输出的变量，无法进行静态优化。

import 特点：语言规格层面支持模块功能。支持编译时静态分析，便于 JS 引入宏和类型检验。动态绑定。

9.javascript 对象的几种创建方式

第一种：Object 构造函数创建

```
var Person = new Object();
Person.name = "Nike";
Person.age = 29;
```

这行代码创建了 Object 引用类型的一个新实例，然后把实例保存在变量 Person 中。

第二种：使用对象字面量表示法

```
var Person = {}; //相当于 var Person = new Object();
var Person = {
     name: 'Nike';
     age: 29;
}
```

对象字面量是对象定义的一种简写形式，目的在于简化创建包含大量属性的对象的过程。也就是说，第一种和第二种方式创建对象的方法其实都是一样的，只是写法上的区别不同

在介绍第三种的创建方法之前，我们应该要明白为什么还要用别的方法来创建对象，也就是第一种，第二种方法的缺点所在：它们都是用了同一个接口创建很多对象，会产生大量的重复代码，就是如果你有 100 个对象，那你要输入 100 次很多相同的代码。那我们有什么方法来避免过多的重复代码呢，就是把创建对象的过程封装在函数体内，通过函数的调用直接生成对象。

第三种：使用工厂模式创建对象

```
function createPerson(name, age, job) {
  var o = new Object();
  o.name = name;
  o.age = age;
  o.job = job;
  o.sayName = function() {
    alert(this.name);
  };
  return o;
}
var person1 = createPerson("Nike", 29, "teacher");
var person2 = createPerson("Arvin", 20, "student");
```

在使用工厂模式创建对象的时候，我们都可以注意到，在 createPerson 函数中，返回的是一个对象。那么我们就无法判断返回的对象究竟是一个什么样的类型。于是就出现了第四种创建对象的模式。

第四种:使用构造函数创建对象

```
function Person(name, age, job) {
  this.name = name;
  this.age = age;
  this.job = job;
  this.sayName = function() {
    alert(this.name);
  };
}
var person1 = new Person("Nike", 29, "teacher");
var person2 = new Person("Arvin", 20, "student");
```

对比工厂模式，我们可以发现以下区别：

1.没有显示地创建对象

2.直接将属性和方法赋给了 this 对象
3.没有 return 语句
4.终于可以识别的对象的类型。对于检测对象类型，我们应该使用 instanceof 操作符，我们来进行自主检测：

```
alert(person1 instanceof Object); //ture
alert(person1 instanceof Person); //ture
alert(person2 instanceof Object); //ture
alert(person2 instanceof Object); //ture
```

同时我们也应该明白，按照惯例，构造函数始终要应该以一个大写字母开头，而非构造函数则应该以一个小写字母开头。
那么构造函数确实挺好用的，但是它也有它的缺点：
就是每个方法都要在每个实例上重新创建一遍，方法指的就是我们在对象里面定义的函数。如果方法的数量很多，就会占用很多不必要的内存。于是出现了第五种创建对象的方法
第五种：原型创建对象模式

```
function Person() {}
Person.prototype.name = "Nike";
Person.prototype.age = 20;
Person.prototype.jbo = "teacher";
Person.prototype.sayName = function() {
  alert(this.name);
};
var person1 = new Person();
person1.sayName();
```

使用原型创建对象的方式，可以让所有对象实例共享它所包含的属性和方法。
如果是使用原型创建对象模式，请看下面代码：

```
function Person() {}
Person.prototype.name = "Nike";
Person.prototype.age = 20;
Person.prototype.jbo = "teacher";
Person.prototype.sayName = function() {
  alert(this.name);
};
var person1 = new Person();
var person2 = new Person();
person1.name = "Greg";
alert(person1.name); //'Greg' --来自实例
alert(person2.name); //'Nike' --来自原型
```

当为对象实例添加一个属性时，这个属性就会屏蔽原型对象中保存的同名属性。
这时候我们就可以使用构造函数模式与原型模式结合的方式，构造函数模式用于定义实例属性，而原型模式用于定义方法和共享的属性
第六种：组合使用构造函数模式和原型模式

```
function Person(name, age, job) {
    this.name = name;
    this.age = age;
    this.job = job;
}
Person.prototype = {
    constructor: Person,
    sayName: function() {
        alert(this.name);
    };
}
var person1 = new Person('Nike', 20, 'teacher');
```

10.JavaScript 继承的方式和优缺点
答案：六种方式

- 一、原型链继承
  - 缺点：
  - 1.引用类型的属性被所有实例共享
  - 2.在创建 Child 的实例时，不能向 Parent 传参
- 二、借用构造函数(经典继承)
  - 优点：
  - 1.避免了引用类型的属性被所有实例共享
  - 2.可以在 Child 中向 Parent 传参
  - 缺点：
  - 1.方法都在构造函数中定义，每次创建实例都会创建一遍方法。
- 三、组合继承
  - 优点：

    - 1.融合原型链继承和构造函数的优点，是 JavaScript 中最常用的继承模式。
- 四、原型式继承
    - 缺点：
    - 1.包含引用类型的属性值始终都会共享相应的值，这点跟原型链继承一样。
- 五、寄生式继承
    - 缺点：
    - 1.跟借用构造函数模式一样，每次创建对象都会创建一遍方法。
- 六、寄生组合式继承
    - 优点：
    - 1.这种方式的高效率体现它只调用了一次 Parent 构造函数，并且因此避免了在 Parent.prototype 上面创建不必要的、多余的属性。
    - 2.与此同时，原型链还能保持不变；
    - 3.因此，还能够正常使用 instanceof 和 isPrototypeOf。
    - 开发人员普遍认为寄生组合式继承是引用类型最理想的继承范式

11.什么是原型链？

答案：通过一个对象的__proto__可以找到它的原型对象，原型对象也是一个对象，就可以通过原型对象的__proto__，最后找到了我们的 Object.prototype,从实例的原型对象开始一直到 Object.prototype 就是我们的原型链

12.复杂数据类型如何转变为字符串

答案：

- 首先，会调用 valueOf 方法，如果方法的返回值是一个基本数据类型，就返回这个值，
- 如果调用 valueOf 方法之后的返回值仍旧是一个复杂数据类型，就会调用该对象的 toString 方法，
- 如果 toString 方法调用之后的返回值是一个基本数据类型，就返回这个值，
- 如果 toString 方法调用之后的返回值是一个复杂数据类型，就报一个错误。

解析：

```
1;
var obj = {
  valueOf: function() {
    return 1;
  }
};
console.log(obj + ""); //'1'
2;
var obj = {
  valueOf: function() {
    return [1, 2];
  }
};
console.log(obj + ""); //'[object Object]';
3;
var obj = {
  valueOf: function() {
    return [1, 2];
  },
  toString: function() {
    return 1;
  }
};
console.log(obj + ""); //'1';
4;
var obj = {
  valueOf: function() {
    return [1, 2];
  },
  toString: function() {
    return [1, 2, 3];
  }
};
console.log(obj + ""); // 报错 Uncaught TypeError: Cannot convert object to primitive value
```

拓展：

```
var arr = [new Object(), new Date(), new RegExp(), new String(), new Number(), new Boolean(), new Function(), new Array(), Math] console.log(arr.length) // 9
for (var i = 0; i < arr.length; i++) {
    arr[i].valueOf = function() {
        return [1, 2, 3]
    }
```

```
    arr[i].toString = function() {
        return 'toString'
    }
    console.log(arr[i] + '')
}
```

1、若 return [1,2,3]处为 return "valueof"，得到的返回值是 valueof toString 7valueof 说明：其他八种复杂数据类型是先调用 valueOf 方法，时间对象是先调用 toString 方法

2、改成 return [1,2,3]，得到的返回值是 9toString 说明：执行 valueof 后都来执行 toString

13.javascript 的 typeof 返回哪些数据类型

答案：7 种分别为 string、boolean、number、Object、Function、undefined、symbol(ES6)、

14. 在 css/js 代码上线之后开发人员经常会优化性能，从用户刷新网页开始，一次 js 请求一般情况下有哪些地方会有缓存处理？

答案：dns 缓存，cdn 缓存，浏览器缓存，服务器缓存。

15.列举 3 种强制类型转换和 2 种隐式类型转换

答案：强制（parseInt,parseFloat,Number）、隐式（+ -）

16.你对闭包的理解？优缺点？

概念：闭包就是能够读取其他函数内部变量的函数。

三大特性：

- 函数嵌套函数。
- 函数内部可以引用外部的参数和变量。
- 参数和变量不会被垃圾回收机制回收。

优点：

- 希望一个变量长期存储在内存中。
- 避免全局变量的污染。
- 私有成员的存在。

缺点：

- 常驻内存，增加内存使用量。
- 使用不当会很容易造成内存泄露。

```
function outer() {
  var name = "jack";
  function inner() {
    console.log(name);
  }
  return inner;
}
outer()(); // jack
function sayHi(name) {
  return () => {
    console.log(`Hi! ${name}`);
  };
}
const test = sayHi("xiaoming");
test(); // Hi! xiaoming
```

虽然 sayHi 函数已经执行完毕，但是其活动对象也不会被销毁，因为 test 函数仍然引用着 sayHi 函数中的变量 name，这就是闭包。

但也因为闭包引用着另一个函数的变量，导致另一个函数已经不使用了也无法销毁，所以闭包使用过多，会占用较多的内存，这也是一个副作用。

解析：

由于在 ECMA2015 中，只有函数才能分割作用域，函数内部可以访问当前作用域的变量，但是外部无法访问函数内部的变量，所以闭包可以理解成“定义在一个函数内部的函数，外部可以通过内部返回的函数访问内部函数的变量“。在本质上，闭包是将函数内部和函数外部连接起来的桥梁。

17.如何判断 NaN

答案：isNaN()方法

解析：isNaN(NaN) // true

18.new 一个对象的过程中发生了什么

```
// 1. 创建空对象；
var obj = {};
// 2. 设置新对象的 constructor 属性为构造函数的名称，设置新对象的__proto__属性指向构造函数的 prototype 对象；
obj.__proto__ = ClassA.prototype;
// 3. 使用新对象调用函数，函数中的 this 被指向新实例对象：
ClassA.call(obj); //{}.构造函数();
// 4. 如果无返回值或者返回一个非对象值，则将新对象返回；如果返回值是一个新对象的话那么直接直接返回该对象。
```

19.for in 和 for of

1、for in

- 1.一般用于遍历对象的可枚举属性。以及对象从构造函数原型中继承的属性。对于每个不同的属性，语句都会被执行。

- 2.不建议使用 for in 遍历数组，因为输出的顺序是不固定的。
- 3.如果迭代的对象的变量值是 null 或者 undefined, for in 不执行循环体，建议在使用 for in 循环之前，先检查该对象的值是不是 null 或者 undefined

2、for of

- 1.for…of 语句在可迭代对象（包括 Array，Map，Set，String，TypedArray，arguments 对象等等）上创建一个迭代循环，调用自定义迭代钩子，并为每个不同属性的值执行语句

解析：

```
var s = {
  a: 1,
  b: 2,
  c: 3
};
var s1 = Object.create(s);
for (var prop in s1) {
  console.log(prop); //a b c
  console.log(s1[prop]); //1 2 3
}
for (let prop of s1) {
  console.log(prop); //报错如下 Uncaught TypeError: s1 is not iterable
}
for (let prop of Object.keys(s1)) {
  console.log(prop); // a b c
  console.log(s1[prop]); //1 2 3
}
```

20. 如何判断 JS 变量的一个类型（至少三种方式）

答案：typeof、instanceof、 constructor、 prototype

21. for in、Object.keys 和 Object.getOwnPropertyNames 对属性遍历有什么区别？

- for in 会遍历自身及原型链上的可枚举属性
- Object.keys 会将对象自身的可枚举属性的 key 输出
- 会将自身所有的属性的 key 输出

ECMAScript 将对象的属性分为两种：数据属性和访问器属性。

```
var parent = Object.create(Object.prototype, {
  a: {
    value: 123,
    writable: true,
    enumerable: true,
    configurable: true
  }
});
// parent 继承自 Object.prototype，有一个可枚举的属性 a（enumerable:true）。

var child = Object.create(parent, {
  b: {
    value: 2,
    writable: true,
    enumerable: true,
    configurable: true
  },
  c: {
    value: 3,
    writable: true,
    enumerable: false,
    configurable: true
  }
});
//child 继承自 parent ，b 可枚举，c 不可枚举
```

for in

```
for (var key in child) {
  console.log(key);
}
// b
// a
// for in 会遍历自身及原型链上的可枚举属性
```

如果只想输出自身的可枚举属性，可使用 hasOwnProperty 进行判断(数组与对象都可以，此处用数组做例子)

```
let arr = [1, 2, 3];
```

```
Array.prototype.xxx = 1231235;
for (let i in arr) {
  if (arr.hasOwnProperty(i)) {
    console.log(arr[i]);
  }
}
// 1
// 2
// 3
```

Object.keys

```
console.log(Object.keys(child));
// ["b"]
// Object.keys 会将对象自身的可枚举属性的 key 输出
```

Object.getOwnPropertyNames

```
console.log(Object.getOwnPropertyNames(child));
// ["b","c"]
// 会将自身所有的属性的 key 输出
```

22.iframe 跨域通信和不跨域通信

答案:

不跨域通信

主页面

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title></title>
  </head>
  <body>
    <iframe
      name="myIframe"
      id="iframe"
      class=""
      src="flexible.html"
      width="500px"
      height="500px"
    >
    </iframe>
  </body>
  <script type="text/javascript" charset="utf-8">
    function fullscreen() {
      alert(1111);
    }
  </script>
</html>
```

子页面 flexible.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title></title>
  </head>
  <body>
    我是子页面
  </body>
  <script type="text/javascript" charset="utf-8">
    // window.parent.fullScreens()
    function showalert() {
      alert(222);
    }
  </script>
</html>
```

1、主页面要是想要调取子页面的 showalert 方法

```
myIframe.window.showalert();
```

2、子页面要掉主页面的 fullscreen 方法

```
window.parent.fullScreens();
```

3、js 在 iframe 子页面获取父页面元素:
window.parent.document.getElementById("元素 id");
4、js 在父页面获取 iframe 子页面元素代码如下:
window.frames["iframe_ID"].document.getElementById("元素 id");
跨域通信
使用 postMessage(官方用法）
子页面
window.parent.postMessage("hello", "http://127.0.0.1:8089");
父页面接收
```
window.addEventListener("message", function(event) {
  alert(123);
});
```
23.H5 与 Native 如何交互
答案：jsBridge
24.如何判断一个对象是否为数组
第一种方法：使用 instanceof 操作符。
第二种方法：使用 ECMAScript 5 新增的 Array.isArray()方法。
第三种方法：使用使用 Object.prototype 上的原生 toString()方法判断。
25.<script> 标签的 defer 和 asnyc 属性的作用以及二者的区别？

- 1、defer 和 async 的网络加载过程是一致的，都是异步执行。
- 2、区别在于加载完成之后什么时候执行，可以看出 defer 是文档所有元素解析完成之后才执行的。
- 3、如果存在多个 defer 脚本，那么它们是按照顺序执行脚本的，而 async，无论声明顺序如何，只要加载完成就立刻执行

无论<script>标签是嵌入代码还是引用外部文件，只要不包含 defer 属性和 async 属性（这两个属性只对外部文件有效），浏览器会按照<script>的出现顺序对他们依次进行解析，也就是说，只有在第一个<script>中的代码执行完成之后，浏览器才会执行第二个<script>中的代码，并且在解析时，页面的处理会暂时停止。
嵌入代码的解析=执行 外部文件的解析=下载+执行
script 标签存在两个属性，defer 和 async，这两个属性只对外部文件有效
只有一个脚本的情况

<script src="a.js" />
没有 defer 或 async 属性，浏览器会立即下载并执行相应的脚本，并且在下载和执行时页面的处理会停止。
<script defer src="a.js" />
有了 defer 属性，浏览器会立即下载相应的脚本，在下载的过程中页面的处理不会停止，等到文档解析完成脚本才会执行。
<script async src="a.js" />
有了 async 属性，浏览器会立即下载相应的脚本，在下载的过程中页面的处理不会停止，下载完成后立即执行，执行过程中页面处理会停止。
<script defer async src="a.js" />
如果同时指定了两个属性,则会遵从 async 属性而忽略 defer 属性。
其中蓝色代表 js 脚本网络下载时间，红色代表 js 脚本执行，绿色代表 html 解析。
多个脚本的情况

这里只列举两个脚本的情况：
<script src="a.js"></script>
<script src="b.js"></script>
没有 defer 或 async 属性，浏览器会立即下载并执行脚本 a.js，在 a.js 脚本执行完成后才会下载并执行脚本 b.js，在脚本下载和执行时页面的处理会停止。
<script defer src="a.js"></script>
<script defer src="b.js"></script>
有了 defer 属性，浏览器会立即下载相应的脚本 a.js 和 b.js，在下载的过程中页面的处理不会停止，等到文档解析完成才会执行这两个脚本。HTML5 规范要求脚本按照它们出现的先后顺序执行，因此第一个延迟脚本会先于第二个延迟脚本执行，而这两个脚本会先于 DOMContentLoaded 事件执行。 在现实当中，延迟脚本并不一定会按照顺序执行，也不一定会在 DOMContentLoaded 事件触发前执行，因此最好只包含一个延迟脚本。
<script async src="a.js"></script>
<script async src="b.js"></script>
有了 async 属性，浏览器会立即下载相应的脚本 a.js 和 b.js，在下载的过程中页面的处理不会停止，a.js 和 b.js 哪个先下载完成哪个就立即执行，执行过程中页面处理会停止，但是其他脚本的下载不会停止。标记为 async 的脚本并不保证按照制定它们的先后顺序执行。异步脚本一定会在页面的 load 事件前执行，但可能会在 DOMContentLoaded 事件触发之前或之后执行。
26.Object.prototype.toString.call() 和 instanceOf 和 Array.isArray() 区别好坏

- Object.prototype.toString.call()
  - 优点：这种方法对于所有基本的数据类型都能进行判断，即使是 null 和 undefined 。
  - 缺点：不能精准判断自定义对象，对于自定义对象只会返回[object Object]
- instanceOf
  - 优点：instanceof 可以弥补 Object.prototype.toString.call()不能判断自定义实例化对象的缺点。
  - 缺点： instanceof 只能用来判断对象类型，原始类型不可以。并且所有对象类型 instanceof Object 都是 true，且不同于其他两种方法的是它不能检测出 iframes。
- Array.isArray()
  - 优点：当检测 Array 实例时，Array.isArray 优于 instanceof ，因为 Array.isArray 可以检测出 iframes

- 缺点：只能判别数组

Object.prototype.toString.call()

每一个继承 Object 的对象都有 toString 方法，如果 toString 方法没有重写的话，会返回 [Object type]，其中 type 为对象的类型。但当除了 Object 类型的对象外，其他类型直接使用 toString 方法时，会直接返回都是内容的字符串，所以我们需要使用 call 或者 apply 方法来改变 toString 方法的执行上下文。

```
const an = ["Hello", "An"];
an.toString(); // "Hello,An"
Object.prototype.toString.call(an); // "[object Array]"
```

这种方法对于所有基本的数据类型都能进行判断，即使是 null 和 undefined 。

```
Object.prototype.toString.call("An"); // "[object String]"
Object.prototype.toString.call(1); // "[object Number]"
Object.prototype.toString.call(Symbol(1)); // "[object Symbol]"
Object.prototype.toString.call(null); // "[object Null]"
Object.prototype.toString.call(undefined); // "[object Undefined]"
Object.prototype.toString.call(function() {}); // "[object Function]"
Object.prototype.toString.call({ name: "An" }); // "[object Object]"
```

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

```
function f(name) {
  this.name = name;
}
var f1 = new f("martin");
console.log(Object.prototype.toString.call(f1)); //[object Object]

Object.prototype.toString.call(); // 常用于判断浏览器内置对象。
```

instanceof

instanceof 的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。

使用 instanceof 判断一个对象是否为数组，instanceof 会判断这个对象的原型链上是否会找到对应的 Array 的原型，找到返回 true，否则返回 false。

```
[] instanceof Array; // true
```

但 instanceof 只能用来判断对象类型，原始类型不可以。并且所有对象类型 instanceof Object 都是 true。

```
[] instanceof Object; // true
```

优点：instanceof 可以弥补 Object.prototype.toString.call()不能判断自定义实例化对象的缺点。

缺点：instanceof 只能用来判断对象类型，原始类型不可以。并且所有对象类型 instanceof Object 都是 true，且不同于其他两种方法的是它不能检测出 iframes。

```
function f(name) {
  this.name = name;
}
var f1 = new f("martin");
console.log(f1 instanceof f); //true
```

Array.isArray()

- 功能：用来判断对象是否为数组
- instanceof 与 isArray

当检测 Array 实例时，Array.isArray 优于 instanceof ，因为 Array.isArray 可以检测出 iframes

```
var iframe = document.createElement("iframe");
document.body.appendChild(iframe);
xArray = window.frames[window.frames.length - 1].Array;
var arr = new xArray(1, 2, 3); // [1,2,3]

// Correctly checking for Array
Array.isArray(arr); // true
Object.prototype.toString.call(arr); // true
// Considered harmful, because doesn't work though iframes
arr instanceof Array; // false
```

缺点：只能判别数组

- Array.isArray() 与 Object.prototype.toString.call()

Array.isArray()是 ES5 新增的方法，当不存在 Array.isArray() ，可以用 Object.prototype.toString.call() 实现。

```
if (!Array.isArray) {
  Array.isArray = function(arg) {
    return Object.prototype.toString.call(arg) === "[object Array]";
  };
}
```

27.什么是面向对象？

答案：面向对象是把构成问题事务分解成各个对象，建立对象的目的不是为了完成一个步骤，而是为了描叙某个事物在整个解决问题的步骤中的行为。

解析：

- 面向对象和面向过程的异同
  - 面向过程就是分析出解决问题所需要的步骤，然后用函数把这些步骤一步一步实现，使用的时候一个一个依次调用就可以了。
  - 面向对象是把构成问题事务分解成各个对象，建立对象的目的不是为了完成一个步骤，而是为了描叙某个事物在整个解决问题的步骤中的行为。

28.你对松散类型的理解

JavaScript 中的变量为松散类型，所谓松散类型就是指当一个变量被申明出来就可以保存任意类型的值，就是不像 SQL 一样申明某个键值为 int 就只能保存整型数值，申明 varchar 只能保存字符串。一个变量所保存值的类型也可以改变，这在 JavaScript 中是完全有效的，只是不推荐。相比较于将变量理解为“盒子“，《JavaScript 编程精解》中提到应该将变量理解为“触手”，它不保存值，而是抓取值。这一点在当变量保存引用类型值时更加明显。

JavaScript 中变量可能包含两种不同的数据类型的值：基本类型和引用类型。基本类型是指简单的数据段，而引用类型指那些可能包含多个值的对象。

30.移动端 click 事件、touch 事件、tap 事件的区别

1. click 事件在移动端会有 200-300ms ms 的延迟，主要原因是苹果手机在设计时，考虑到用户在浏览网页时需要放大，所以，在用户点击的 200-300ms 之后，才触发 click，如果 200-300ms 之内还有 click，就会进行放大缩小。
2. touch 事件是针对触屏手机上的触摸事件。现今大多数触屏手机 webkit 内核提供了 touch 事件的监听，让开发者可以获取用户触摸屏幕时的一些信息。其中包括：touchstart,touchmove,touchend,touchcancel 这四个事件，touchstart touchmove touchend 事件可以类比于 mousedown mouseover mouseup 的触发
3. tap 事件在移动端，代替 click 作为点击事件，tap 事件被很多框架（如 zepto）封装，来减少这延迟问题， tap 事件不是原生的，所以是封装的，那么具体是如何实现的呢？

```
<script>
  function tap(ele, callback) {
    // 记录开始时间
    var startTime = 0,
    // 控制允许延迟的时间
        delayTime = 200,
    // 记录是否移动，如果移动，则不触发 tap 事件
        isMove = false;
    // 在 touchstart 时记录开始的时间
    ele.addEventListener('touchstart', function (e) {
      startTime = Date.now();
    });
    // 如果 touchmove 事件被触发，则 isMove 为 true
    ele.addEventListener('touchmove', function (e) {
      isMove = true;
    });
    // 如果 touchmove 事件触发或者中间时间超过了延迟时间，则返回，否则，调用回调函数。
    ele.addEventListener('touchend', function (e) {
      if (isMove || (Date.now() - startTime > delayTime)) {
        return;
      } else {
        callback(e);
      }
    })
  }
  var btn = document.getElementById('btn');
  tap(btn, function () {
    alert('taped');
  });
</script>
```

拓展：

点透问题

如果我们在移动端所有的 click 都替换为了 tap 事件，还是会触发点透问题的，因为实质是： 在同一个 z 轴上，z-index 不同的两个元素，上面的元素是一个绑定了 tap 事件的，下面是一个 a 标签，一旦 tap 触发，这个元素就会 display: none，而从上面的 tap 可以看出，有 touchstart、touchend，所以会 300ms 之后触发 click 事件，而 z-index 已经消失了，所以，触发了下面的 a 的 click 事件，注意： 我们认为 a 标签默认是绑定了 click 事件的。而这种现象不是我们所期待的。

解决方案： （1）使用 fastclick。 （2）添加一个延迟。

（1）直接引入 fastclick 库。

```
window.addEventListener(
  "load",
  function() {
    FastClick.attach(document.body);
  },
  false
```

```
);
```
这样，就可以成功解决问题了。
（2）对于上一个 tap 做延迟。
```
tap(ele, function() {
  setTimeout(function() {
    ele.style.display = "none";
  }, 300);
});
```
这样，过了 300ms，那么 click 事件就不会触发在下面的 a 标签上了。

31.JS 单线程还是多线程，如何显示异步操作

答案：JS 本身是单线程的，他是依靠浏览器完成的异步操作。
解析：
具体步骤，
1、主线程 执行 js 中所有的代码。
2、主线程 在执行过程中发现了需要异步的任务任务后扔给浏览器（浏览器创建多个线程执行），并在 callback queque 中创建对应的回调函数（回调函数是一个对象，包含该函数是否执行完毕等）。
3、主线程 已经执行完毕所有同步代码。开始监听 callback queque 一旦 浏览器 中某个线程任务完成将会改变回调函数的状态。主线程查看到某个函数的状态为已完成，就会执行该函数。

32. JavaScript 数组的函数 map/forEach/reduce/filter

1. map

```
// map
//作用：对数组进行遍历
//返回值：新的数组
// 是否改变：否
var arr = [2, 5, 3, 4];
var ret = arr.map(function(value) {
  return value + 1;
});
console.log(ret); //[3,6,4,5]
console.log(arr); //[2,5,3,4]
```

2. forEach

```
// forEach 方法
// 作用：遍历数组的每一项
// 返回值：undefined
// 是否改变：否
var arr = [2, 5, 3, 4];
var ret = arr.forEach(function(value) {
  console.log(value); // 2, 5, 3, 4
});
console.log(ret); //undefined
console.log(arr); //[2,5,3,4]
```

3. reduce

```
// reduce 方法
// 作用：对数组进行迭代，然后两两进行操作，最后返回一个值
// 返回值：return 出来的结果
// 是否改变：不会
var arr = [1, 2, 3, 4];
var ret = arr.reduce(function(a, b) {
  return a * b;
});
console.log(ret); // 24
console.log(arr); // [1, 2, 3, 4]
```

4. filter

```
// filter 过滤
// 作用： 筛选一部分元素
// 返回值： 一个满足筛选条件的新数组
// 是否改变原有数组：不会
var arr = [2, 5, 3, 4];
var ret = arr.filter(function(value) {
  return value > 3;
});
console.log(ret); //[5,4]
console.log(arr); //[2,5,3,4]
```

33. JS 块级作用域、变量提升

1. 块级作用域

JS 中作用域有：全局作用域、函数作用域。没有块作用域的概念。ECMAScript 6(简称 ES6)中新增了块级作用域。块作用域由 { } 包括，if 语句和 for 语句里面的{ }也属于块作用域。

2. 变量提升

- 如果变量声明在函数里面，则将变量声明提升到函数的开头
- 如果变量声明是一个全局变量，则将变量声明提升到全局作用域的开头

```
<script type="text/javascript">
    {
        var a = 1;
        console.log(a); // 1
    }
    console.log(a); // 1
    // 可见，通过 var 定义的变量可以跨块作用域访问到。
    (function A() {
        var b = 2;
        console.log(b); // 2
    })();
    // console.log(b); // 报错，
    // 可见，通过 var 定义的变量不能跨函数作用域访问到
    if(true) {
        var c = 3;
    }
    console.log(c); // 3
    for(var i = 0; i < 4; i++) {
        var d = 5;
    };
    console.log(i);    // 4   (循环结束 i 已经是 4，所以此处 i 为 4)
    console.log(d); // 5
    // if 语句和 for 语句中用 var 定义的变量可以在外面访问到，
    // 可见，if 语句和 for 语句属于块作用域，不属于函数作用域。
    {
        var a = 1;
        let b = 2;
        const c = 3;
        {
            console.log(a);         // 1 子作用域可以访问到父作用域的变量
            console.log(b);         // 2 子作用域可以访问到父作用域的变量
            console.log(c);         // 3 子作用域可以访问到父作用域的变量
            var aa = 11;
            let bb = 22;
            const cc = 33;
        }
        console.log(aa);  // 11    // 可以跨块访问到子 块作用域 的变量
        // console.log(bb);    // 报错  bb is not defined
        // console.log(cc);    // 报错  cc is not defined
    }
</script>
```

var、let、const 的区别

- var 定义的变量，没有块的概念，可以跨块访问，不能跨函数访问。
- let 定义的变量，只能在块作用域里访问，不能跨块访问，也不能跨函数访问。
- const 用来定义常量，使用时必须初始化(即必须赋值)，只能在块作用域里访问，而且不能修改。
- 同一个变量只能使用一种方式声明，不然会报错

```
<script type="text/javascript">
    // 块作用域
    {
        var a = 1;
        let b = 2;
        const c = 3;
        // c = 4; // 报错
        // let a = 'a';    // 报错  注：是上面 var a = 1; 那行报错
        // var b = 'b';    // 报错：本行报错
        // const a = 'a1'; // 报错  注：是上面 var a = 1; 那行报错
        // let c = 'c';    // 报错：本行报错
        var aa;
        let bb;
        // const cc; // 报错
```

```
        console.log(a); // 1
        console.log(b); // 2
        console.log(c); // 3
        console.log(aa); // undefined
        console.log(bb); // undefined
    }
    console.log(a); // 1
    // console.log(b); // 报错
    // console.log(c); // 报错
    // 函数作用域
    (function A() {
        var d = 5;
        let e = 6;
        const f = 7;
        console.log(d); // 5
        console.log(e); // 6  (在同一个{ }中,也属于同一个块，可以正常访问到)
        console.log(f); // 7  (在同一个{ }中,也属于同一个块，可以正常访问到)
    })();
    // console.log(d); // 报错
    // console.log(e); // 报错
    // console.log(f); // 报错
</script>
```

34. null/undefined 的区别

null： Null 类型，代表“空值”，代表一个空对象指针，使用 typeof 运算得到 “object”，所以你可以认为它是一个特殊的对象值。

undefined： Undefined 类型，当一个声明了一个变量未初始化时，得到的就是 undefined。

35. JS 哪些操作会造成内存泄露

1）意外的全局变量引起的内存泄露

```
function leak() {
  leak = "xxx"; //leak 成为一个全局变量，不会被回收
}
```

2）闭包引起的内存泄露

```
function bindEvent() {
  var obj = document.createElement("XXX");
  obj.onclick = function() {
    //Even if it's a empty function
  };
}
```

闭包可以维持函数内局部变量，使其得不到释放。 上例定义事件回调时，由于是函数内定义函数，并且内部函数--事件回调的引用外暴了，形成了闭包。 解决之道，将事件处理函数定义在外部，解除闭包,或者在定义事件处理函数的外部函数中，删除对 dom 的引用。

```
//将事件处理函数定义在外部
function onclickHandler() {
  //do something
}
function bindEvent() {
  var obj = document.createElement("XXX");
  obj.onclick = onclickHandler;
}

//在定义事件处理函数的外部函数中，删除对 dom 的引用
function bindEvent() {
  var obj = document.createElement("XXX");
  obj.onclick = function() {
    //Even if it's a empty function
  };
  obj = null;
}
```

3）没有清理的 DOM 元素引用

```
var elements={
    button: document.getElementById("button"),
    image: document.getElementById("image"),
    text: document.getElementById("text")
};
function doStuff(){
```

```
    image.src="http://some.url/image";
    button.click();
    console.log(text.innerHTML)
}
function removeButton(){
    document.body.removeChild(document.getElementById('button'))
}
```

4）被遗忘的定时器或者回调

```
var someResouce = getData();
setInterval(function() {
  var node = document.getElementById("Node");
  if (node) {
    node.innerHTML = JSON.stringify(someResouce);
  }
}, 1000);
```

这样的代码很常见，如果 id 为 Node 的元素从 DOM 中移除，该定时器仍会存在，同时，因为回调函数中包含对 someResource 的引用，定时器外面的 someResource 也不会被释放。

5）子元素存在引起的内存泄露

黄色是指直接被 js 变量所引用，在内存里，红色是指间接被 js 变量所引用，如上图，refB 被 refA 间接引用，导致即使 refB 变量被清空，也是不会被回收的子元素 refB 由于 parentNode 的间接引用，只要它不被删除，它所有的父元素（图中红色部分）都不会被删除。

6）IE7/8 引用计数使用循环引用产生的问题

```
function fn() {
  var a = {};
  var b = {};
  a.pro = b;
  b.pro = a;
}
fn();
```

fn()执行完毕后，两个对象都已经离开环境，在标记清除方式下是没有问题的，但是在引用计数策略下，因为 a 和 b 的引用次数不为 0，所以不会被垃圾回收器回收内存，如果 fn 函数被大量调用，就会造成内存泄漏。在 IE7 与 IE8 上，内存直线上升。 IE 中有一部分对象并不是原生 js 对象。例如，其内存泄漏 DOM 和 BOM 中的对象就是使用 C++以 COM 对象的形式实现的，而 COM 对象的垃圾回收机制采用的就是引用计数策略。因此，即使 IE 的 js 引擎采用标记清除策略来实现，但 js 访问的 COM 对象依然是基于引用计数策略的。换句话说，只要在 IE 中涉及 COM 对象，就会存在循环引用的问题。

```
var element = document.getElementById("some_element");
var myObject = new Object();
myObject.e = element;
element.o = myObject;
```

上面的例子在一个 DOM 元素（element)与一个原生 js 对象（myObject)之间创建了循环引用。其中，变量 myObject 有一个名为 e 的属性指向 element 对象；而变量 element 也有一个属性名为 o 回指 myObject。由于存在这个循环引用，即使例子中的 DOM 从页面中移除，它也永远不会被回收。

看上面的例子，有人会觉得太弱了，谁会做这样无聊的事情，但是其实我们经常会这样做

```
window.onload=function outerFunction(){
  var obj=document.getElementById("element");
  obj.onclick=function innerFunction(){};
};
```

这段代码看起来没什么问题，但是 obj 引用了 document.getElementById(“element”)，而 document.getElementById(“element”)的 onclick 方法会引用外部环境中的变量，自然也包括 obj，是不是很隐蔽啊。

最简单的解决方式就是自己手工解除循环引用，比如刚才的函数可以这样

```
myObject.element=null;
element.o=null;
window.onload=function outerFunction(){
  var obj=document.getElementById("element");
  obj.onclick=function innerFunction(){};
  obj=null;
};
```

将变量设置为 null 意味着切断变量与它此前引用的值之间的连接。当垃圾回收器下次运行时，就会删除这些值并回收它们占用的内存。 要注意的是，IE9+并不存在循环引用导致 Dom 内存泄漏问题，可能是微软做了优化，或者 Dom 的回收方式已经改变

解析：

1、JS 的回收机制

JavaScript 垃圾回收的机制很简单：找出不再使用的变量，然后释放掉其占用的内存，但是这个过程不是实时的，因为其开销比较大，所以垃圾回收系统（GC）会按照固定的时间间隔,周期性的执行。

到底哪个变量是没有用的？所以垃圾收集器必须跟踪到底哪个变量没用，对于不再有用的变量打上标记，以备将来收回其内存。用于标记的无用变量的策略可能因实现而有所区别，通常情况下有两种实现方式：标记清除和引用计数。引用计数不太常用，标记清除较为常用。

2、标记清除（mark and sweep）
js 中最常用的垃圾回收方式就是标记清除。当变量进入环境时，例如，在函数中声明一个变量，就将这个变量标记为“进入环境”。从逻辑上讲，永远不能释放进入环境的变量所占用的内存，因为只要执行流进入相应的环境，就可能会用到它们。而当变量离开环境时，则将其标记为“离开环境”。

```
function test() {
  var a = 10; //被标记，进入环境
  var b = 20; //被标记，进入环境
}
test(); //执行完毕之后 a、b 又被标记离开环境，被回收
```

3、引用计数(reference counting)
引用计数的含义是跟踪记录每个值被引用的次数。当声明了一个变量并将一个引用类型值（function object array）赋给该变量时，则这个值的引用次数就是 1。如果同一个值又被赋给另一个变量，则该值的引用次数加 1。相反，如果包含对这个值引用的变量又取得了另外一个值，则这个值的引用次数减 1。当这个值的引用次数变成 0 时，则说明没有办法再访问这个值了，因而就可以将其占用的内存空间回收回来。这样，当垃圾回收器下次再运行时，它就会释放那些引用次数为 0 的值所占用的内存。

```
function test() {
  var a = {}; //a 的引用次数为 0
  var b = a; //a 的引用次数加 1，为 1
  var c = a; //a 的引用次数加 1，为 2
  var b = {}; //a 的引用次数减 1，为 1
}
```

4、如何分析内存的使用情况
Google Chrome 浏览器提供了非常强大的 JS 调试工具，Memory 视图 profiles 视图让你可以对 JavaScript 代码运行时的内存进行快照，并且可以比较这些内存快照。它还让你可以记录一段时间内的内存分配情况。在每一个结果视图中都可以展示不同类型的列表，但是对我们最有用的是 summary 列表和 comparison 列表。 summary 视图提供了不同类型的分配对象以及它们的合计大小：shallow size （一个特定类型的所有对象的总和）和 retained size （shallow size 加上保留此对象的其它对象的大小）。distance 显示了对象到达 GC 根（校者注：最初引用的那块内存，具体内容可自行搜索该术语）的最短距离。 comparison 视图提供了同样的信息但是允许对比不同的快照。这对于找到泄漏很有帮助。
5、怎样避免内存泄露
1）减少不必要的全局变量，或者生命周期较长的对象，及时对无用的数据进行垃圾回收；
2）注意程序逻辑，避免“死循环”之类的 ；
3）避免创建过多的对象 原则：不用了的东西要及时归还。

36.重排与重绘的区别，什么情况下会触发？

1. 简述重排的概念
   浏览器下载完页面中的所有组件（HTML、JavaScript、CSS、图片）之后会解析生成两个内部数据结构（DOM 树和渲染树），DOM 树表示页面结构，渲染树表示 DOM 节点如何显示。重排是 DOM 元素的几何属性变化，DOM 树的结构变化，渲染树需要重新计算。
2. 简述重绘的概念
   重绘是一个元素外观的改变所触发的浏览器行为，例如改变 visibility、outline、背景色等属性。浏览器会根据元素的新属性重新绘制，使元素呈现新的外观。由于浏览器的流布局，对渲染树的计算通常只需要遍历一次就可以完成。但 table 及其内部元素除外，它可能需要多次计算才能确定好其在渲染树中节点的属性值，比同等元素要多花两倍时间，这就是我们尽量避免使用 table 布局页面的原因之一。
3. 简述重绘和重排的关系 重绘不会引起重排，但重排一定会引起重绘，一个元素的重排通常会带来一系列的反应，甚至触发整个文档的重排和重绘，性能代价是高昂的。
4. 什么情况下会触发重排？

- 页面渲染初始化时；（这个无法避免）
- 浏览器窗口改变尺寸；
- 元素尺寸改变时；
- 元素位置改变时；
- 元素内容改变时；
- 添加或删除可见的 DOM 元素时。

5. 重排优化有如下五种方法

- 将多次改变样式属性的操作合并成一次操作，减少 DOM 访问。
- 如果要批量添加 DOM，可以先让元素脱离文档流，操作完后再带入文档流，这样只会触发一次重排。（fragment 元素的应用）
- 将需要多次重排的元素，position 属性设为 absolute 或 fixed，这样此元素就脱离了文档流，它的变化不会影响到其他元素。例如有动画效果的元素就最好设置为绝对定位。
- 由于 display 属性为 none 的元素不在渲染树中，对隐藏的元素操作不会引发其他元素的重排。如果要对一个元素进行复杂的操作时，可以先隐藏它，操作完成后再显示。这样只在隐藏和显示时触发两次重排。
- 在内存中多次操作节点，完成后再添加到文档中去。例如要异步获取表格数据，渲染到页面。可以先取得数据后在内存中构建整个表格的 html 片段，再一次性添加到文档中去，而不是循环添加每一行。

37.发布订阅设计模式
答案：发布/订阅模式(Publish Subscribe Pattern)属于设计模式中的行为(Behavioral Patterns)
38. jsonp 优缺点？
jsonp 优缺点

- 1.优点

    - 1.1 它不像 XMLHttpRequest 对象实现的 Ajax 请求那样受到同源策略的限制，JSONP 可以跨越同源策略；
    - 1.2 它的兼容性更好，在更加古老的浏览器中都可以运行，不需要 XMLHttpRequest 或 ActiveX 的支持
    - 1.3 在请求完毕后可以通过调用 callback 的方式回传结果。将回调方法的权限给了调用方。这个就相当于将 controller 层和 view 层终于*分 开了。我提供的 jsonp 服务只提供纯服务的数据，至于提供服务以 后的页面渲染和后续 view 操作都由调用者来自己定义就好了。如果*有两个页面需要渲染同一份数据，你们只需要有不同的渲染逻辑就可以了，逻辑都可以使用同 一个 jsonp 服务。
- 2.缺点
    - 2.1 它只支持 GET 请求而不支持 POST 等其它类型的 HTTP 请求
    - 2.2 它只支持跨域 HTTP 请求这种情况，不能解决不同域的两个页面之间如何进行 JavaScript 调用的问题。
    - 2.3 jsonp 在调用失败的时候不会返回各种 HTTP 状态码。
    - 2.4 缺点是安全性。万一假如提供 jsonp 的服务存在页面注入漏洞，即它返回的 javascript 的内容被人控制的。那么结果是什么？所有调用这个 jsonp 的网站都会存在漏洞。于是无法把危险控制在一个域名下…所以在使用 jsonp 的时候必须要保证使用的 jsonp 服务必须是安全可信的

39.兼容各种浏览器版本的事件绑定

```
/*
兼容低版本 IE，ele 为需要绑定事件的元素，
eventName 为事件名（保持 addEventListener 语法，去掉 on），fun 为事件响应函数
*/
function addEvent(ele, eventName, fun) {
  if (ele.addEventListener) {
    ele.addEventListener(eventName, fun, false);
  } else {
    ele.attachEvent("on" + eventNme, fun);
  }
}
```

40.typescript 遇到过什么坑

main.ts 报错（ Cannot find module './App.vue'.）
原因： typescript 不能识别.vue 文件
解决办法： 引入 vue 的 typescript declare 库

41.this 和 apply 的应用

答案：比如求数组的最大值 Math.max.apply(this, 数组)

```
var numbers = [5, 458, 120, -215];
var maxInNumbers = Math.max.apply(this, numbers); //第一个参数也可以填 Math 或 null
console.log(maxInNumbers); // 458
var maxInNumbers = Math.max.call(this, 5, 458, 120, -215);
console.log(maxInNumbers); // 458
```

42.split() join()的区别

join()：用于把数组中的所有元素通过指定的分隔符进行分隔放入一个字符串
split()：用于把一个字符串通过指定的分隔符进行分隔成数组

43.JavaScript 的数据类型

答案：JS 数据类型共有六种，分别是 String、Number、Boolean、Null、Undefined 和 Object 等， 另外，ES6 新增了 Symbol 类型。其中，Object 是引用类型，其他的都是基本类型(Primitive Type)。

44.如何判断一个对象是否属于某个类?

答案：instanceof if (a instanceof Person) { alert("yes"); }

45.new 操作符具体干了什么呢?

样本一
new 共经过了 4 几个阶段

- 1、创建一个空对象
- 2、设置原型链
- 3、让 Func 中的 this 指向 obj，并执行 Func 的函数体
- 4、判断 Func 的返回值类型：

样本二

```
function Test(){}
const test = new Test()
```

1. 创建一个新对象：

```
const obj = {}
```

2. 设置新对象的 constructor 属性为构造函数的名称，设置新对象的__proto__属性指向构造函数的 prototype 对象

```
obj.constructor = Test
obj.__proto__ = Test.prototype
```

3. 使用新对象调用函数，函数中的 this 被指向新实例对象

```
Test.call(obj)
```

4. 将初始化完毕的新对象地址，保存到等号左边的变量中

46.call() 和 apply() 的含义和区别?

首先说明两个方法的含义：

- call：调用一个对象的一个方法，用另一个对象替换当前对象。例如：B.call(A, args1,args2);即 A 对象调用 B 对象的方法。
- apply：调用一个对象的一个方法，用另一个对象替换当前对象。例如：B.apply(A, arguments);即 A 对象应用 B 对象的方法。

call 与 apply 的相同点：

- 方法的含义是一样的，即方法功能是一样的；
- 第一个参数的作用是一样的；

call 与 apply 的不同点：两者传入的列表形式不一样

- call 可以传入多个参数；
- apply 只能传入两个参数，所以其第二个参数往往是作为数组形式传入

想一想哪个性能更好一些呢？

47.sort 排序原理

答案：冒泡排序法

冒泡排序法的原理：

- 比较相邻的元素。如果第一个比第二个大，就交换他们两个。
- 对每一对相邻元素做同样的工作，从开始第一对到结尾的最后一对。在这一点，最后的元素应该会是最大的数。
- 针对所有的元素重复以上的步骤，除了最后一个。
- 持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。

示例：

```
var arr = [1, 5, 4, 2];
// sort()方法的比较逻辑为：
// 第一轮：1 和 5 比，1 和 4 比，1 和 2 比
// 第二轮：5 和 4 比，5 和 2 比
// 第三轮：4 和 2 比
// 一.sort 排序规则 return 大于 0 则交换数组相邻 2 个元素的位置
// 二.arr.sort(function (a,b) {})中
//         a -->代表每一次执行匿名函时候，找到的数组中的当前项；
//         b -->代表当前项的后一项；

// 1.升序
var apple = [45, 42, 10, 147, 7, 65, -74];
// ①默认法,缺点:只根据首位排序
console.log(apple.sort());
// ②指定排序规则法,return 可返回任何值
console.log(
  apple.sort(function(a, b) {
    return a - b; //若 return 返回值大于 0(即 a>b),则 a,b 交换位置
  })
);
//2.降序
var arr = [45, 42, 10, 111, 7, 65, -74];
console.log(
  apple.sort(function(a, b) {
    return b - a; //若 return 返回值大于零(即 b>a),则 a,b 交换位置
  })
);
```

48.Zepto 的点透问题如何解决？

方案一：来得很直接 github 上有个 fastclick 可以完美解决 https://github.com/ftlabs/fastclick

引入 fastclick.js，因为 fastclick 源码不依赖其他库所以你可以在原生的 js 前直接加上

```
window.addEventListener(
  "load",
  function() {
    FastClick.attach(document.body);
  },
  false
);
```

或者有 zepto 或者 jqm 的 js 里面加上

```
$(function() {
  FastClick.attach(document.body);
});
```

当然 require 的话就这样：

```
var FastClick = require("fastclick");
FastClick.attach(document.body, options);
```

方案二：用 touchend 代替 tap 事件并阻止掉 touchend 的默认行为 preventDefault()

```
$("#cbFinish").on("touchend", function(event) {
```

```
  //很多处理比如隐藏什么的
  event.preventDefault();
});
```

方案三：延迟一定的时间(300ms+)来处理事件

```
$("#cbFinish").on("tap", function(event) {
  setTimeout(function() {
    //很多处理比如隐藏什么的
  }, 320);
});
```

这种方法其实很好，可以和 fadeInIn/fadeOut 等动画结合使用，可以做出过渡效果

理论上上面的方法可以完美的解决 tap 的点透问题，如果真的不行，用 click

解析：

1、“点透”是什么？

你可能碰到过在列表页面上创建一个弹出层，弹出层有个关闭的按钮，你点了这个按钮关闭弹出层后后，这个按钮正下方的内容也会执行点击事件（或打开链接）。这个被定义为这是一个“点透”现象。

2、为什么会出现点透呢？

49.如何判断当前脚本运行在浏览器还是 node 环境中？

答案：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

50.移动端最小触控区域是多大？

答案：苹果推荐是 44pt x 44pt

51.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？

1. 300 毫秒
2. 因为浏览器捕获第一次单击后，会先等待一段时间，如果在这段时间区间里用户未进行下一次点击，则浏览器会做单击事件的处理。如果这段时间里用户进行了第二次单击操作，则浏览器会做双击事件处理。
3. 推荐 fastclick.js

52.解释 JavaScript 中的作用域与变量声明提升？

- 我对作用域的理解是只会对某个范围产生作用，而不会对外产生影响的封闭空间。在这样的一些空间里，外部不能访问内部变量，但内部可以访问外部变量。
- 所有申明都会被提升到作用域的最顶上
- 同一个变量申明只进行一次，并且因此其他申明都会被忽略
- 函数声明的优先级优于变量申明，且函数声明会连带定义一起被提升

53.Node.js 的适用场景？

答案：比如：RESTFUL API、实时聊天、客户端逻辑强大的单页 APP，具体的例子比如说：本地化的在线音乐应用，本地化的在线搜索应用，本地化的在线 APP 等。

54.bind、call、apply 的区别

call 和 apply 其实是一样的，区别就在于传参时参数是一个一个传或者是以一个数组的方式来传。

call 和 apply 都是在调用时生效，改变调用者的 this 指向。

```
let name = 'Jack'
const obj = {name: 'Tom'}
function sayHi() {console.log('Hi! ' + this.name)}

sayHi() // Hi! Jack
sayHi.call(obj) // Hi! Tom
```

bind 也是改变 this 指向，不过不是在调用时生效，而是返回一个新函数。

```
const newFunc = sayHi.bind(obj)
newFunc() // Hi! Tom
```

55.使用构造函数的注意点

1. 一般情况下构造函数的首字母需要大写，因为我们在看到一个函数首字母大写的情况，就认定这是一个构造函数，需要跟 new 关键字进行搭配使用，创建一个新的实例（对象）
2. 构造函数在被调用的时候需要跟 new 关键字搭配使用。
3. 在构造函数内部通过 this+属性名的形式为实例添加一些属性和方法。
4. 构造函数一般不需要返回值，如果有返回值
    - 4.1 如果返回值是一个基本数据类型，那么调用构造函数，返回值仍旧是那么创建出来的对象。
    - 4.2 如果返回值是一个复杂数据类型，那么调用构造函数的时候，返回值就是这个 return 之后的那个复杂数据类型。

56.如何获取浏览器版本信息

答案：window.navigator.userAgent

57.如何实现文件断点续传

答案：断点续传最核心的内容就是把文件“切片”然后再一片一片的传给服务器，但是这看似简单的上传过程却有着无数的坑。

首先是文件的识别，一个文件被分成了若干份之后如何告诉服务器你切了多少块，以及最终服务器应该如何把你上传上去的文件进行合并，这都是要考虑的。

因此在文件开始上传之前，我们和服务器要有一个“握手”的过程，告诉服务器文件信息，然后和服务器约定切片的大小，当和服务器达成共识之后就可以开始后续的文件传输了。

前台要把每一块的文件传给后台，成功之后前端和后端都要标识一下，以便后续的断点。

当文件传输中断之后用户再次选择文件就可以通过标识来判断文件是否已经上传了一部分，如果是的话，那么我们可以接着上次的进度继续传文件，以达到续传的功能。 有了 HTML5 的 File api 之后切割文件比想想的要简单的多的多。
只要用 slice 方法就可以了
```
var packet = file.slice(start, end);
```
参数 start 是开始切片的位置，end 是切片结束的位置 单位都是字节。通过控制 start 和 end 就可以是实现文件的分块
如
```
file.slice(0,1000);
file.slice(1000,2000);
file.slice(2000,3000);
// ......
```
在把文件切成片之后，接下来要做的事情就是把这些碎片传到服务器上。 如果中间掉线了，下次再传的时候就得先从服务器获取上一次上传文件的位置，然后以这个位置开始上传接下来的文件内容。

58.数组的常用方法

答案：

1. Array.map()

此方法是将数组中的每个元素调用一个提供的函数，结果作为一个新的数组返回，并没有改变原来的数组
```
let arr = [1, 2, 3, 4, 5];
let newArr = arr.map(x => x * 2);
//arr= [1, 2, 3, 4, 5]  原数组保持不变
//newArr = [2, 4, 6, 8, 10] 返回新数组
```
2. Array.forEach()

此方法是将数组中的每个元素执行传进提供的函数，没有返回值，直接改变原数组，注意和 map 方法区分
```
let arr = [1, 2, 3, 4, 5];
num.forEach(x => x * 2);
// arr = [2, 4, 6, 8, 10]  数组改变,注意和 map 区分
```
3. Array.filter()

此方法是将所有元素进行判断，将满足条件的元素作为一个新的数组返回
```
let arr = [1, 2, 3, 4, 5]
    const isBigEnough => value => value >= 3
    let newArr = arr.filter(isBigEnough )
    //newNum = [3, 4, 5] 满足条件的元素返回为一个新的数组
```
4. Array.every()

此方法是将所有元素进行判断返回一个布尔值，如果所有元素都满足判断条件，则返回 true，否则为 false：
```
let arr = [1, 2, 3, 4, 5]
    const isLessThan4 => value => value < 4
    const isLessThan6 => value => value < 6
    arr.every(isLessThan4 ) //false
    arr.every(isLessThan6 ) //true
```
5. Array.some()

此方法是将所有元素进行判断返回一个布尔值，如果存在元素都满足判断条件，则返回 true，若所有元素都不满足判断条件，则返回 false：
```
let arr= [1, 2, 3, 4, 5]
    const isLessThan4 => value => value < 4
    const isLessThan6 => value => value > 6
    arr.some(isLessThan4 ) //true
    arr.some(isLessThan6 ) //false
```
6. Array.reduce()

此方法是所有元素调用返回函数，返回值为最后结果,传入的值必须是函数类型：
```
let arr = [1, 2, 3, 4, 5];
const add = (a, b) => a + b;
let sum = arr.reduce(add);
//sum = 15  相当于累加的效果
```
与之相对应的还有一个 Array.reduceRight() 方法，区别是这个是从右向左操作的

7. Array.push()

此方法是在数组的后面添加新加元素，此方法改变了数组的长度：

8. Array.pop()

此方法在数组后面删除最后一个元素，并返回数组，此方法改变了数组的长度：
```
let arr = [1, 2, 3, 4, 5];
arr.pop();
console.log(arr); //[1, 2, 3, 4]
console.log(arr.length); //4
```
9. Array.shift()

此方法在数组后面删除第一个元素，并返回数组，此方法改变了数组的长度：
```
let arr = [1, 2, 3, 4, 5];
arr.shift();
```

```
console.log(arr); //[2, 3, 4, 5]
console.log(arr.length); //4
```
10. Array.unshift()

此方法是将一个或多个元素添加到数组的开头，并返回新数组的长度：

```
let arr = [1, 2, 3, 4, 5];
arr.unshift(6, 7);
console.log(arr); //[6, 7, 2, 3, 4, 5]
console.log(arr.length); //7
```
11. Array.isArray()

判断一个对象是不是数组，返回的是布尔值

12. Array.concat()

此方法是一个可以将多个数组拼接成一个数组：

```
let arr1 = [1, 2, 3]
      arr2 = [4, 5]
  let arr = arr1.concat(arr2)
  console.log(arr)//[1, 2, 3, 4, 5]
```
13. Array.toString()

此方法将数组转化为字符串：

```
let arr = [1, 2, 3, 4, 5];
   let str = arr.toString()
   console.log(str)// 1,2,3,4,5
```
14. Array.join()

此方法也是将数组转化为字符串：

```
let arr = [1, 2, 3, 4, 5];
   let str1 = arr.toString()
   let str2 = arr.toString(',')
   let str3 = arr.toString('##')
   console.log(str1)// 12345
   console.log(str2)// 1,2,3,4,5
   console.log(str3)// 1##2##3##4##5
```
通过例子可以看出和 toString 的区别，可以设置元素之间的间隔~

15. Array.splice(开始位置， 删除的个数，元素)

万能方法，可以实现增删改：

```
let arr = [1, 2, 3, 4, 5];
     let arr1 = arr.splice(2, 0 'haha')
     let arr2 = arr.splice(2, 3)
     let arr1 = arr.splice(2, 1 'haha')
     console.log(arr1) //[1, 2, 'haha', 3, 4, 5]新增一个元素
     console.log(arr2) //[1, 2] 删除三个元素
     console.log(arr3) //[1, 2, 'haha', 4, 5] 替换一个元素
```
59.字符串常用操作

- charAt(index):返回指定索引处的字符串
- charCodeAt(index):返回指定索引处的字符的 Unicode 的值
- concat(str1,str2,...):连接多个字符串，返回连接后的字符串的副本
- fromCharCode():将 Unicode 值转换成实际的字符串
- indexOf(str):返回 str 在父串中第一次出现的位置，若没有则返回-1
- lastIndexOf(str):返回 str 在父串中最后一次出现的位置，若没有则返回-1
- match(regex):搜索字符串，并返回正则表达式的所有匹配
- replace(str1,str2):str1 也可以为正则表达式，用 str2 替换 str1
- search(regex):基于正则表达式搜索字符串，并返回第一个匹配的位置
- slice(start,end)：返回字符索引在 start 和 end（不含）之间的子串
- split(sep，limit)：将字符串分割为字符数组，limit 为从头开始执行分割的最大数量
- substr(start，length)：从字符索引 start 的位置开始，返回长度为 length 的子串
- substring(from,to)：返回字符索引在 from 和 to（不含）之间的子串
- toLowerCase()：将字符串转换为小写
- toUpperCase()：将字符串转换为大写
- valueOf()：返回原始字符串值

60.作用域的概念及作用

- 作用域 ： 起作用的一块区域
- 作用域的概念： 对变量起保护作用的一块区域
- 作用： 作用域外部无法获取到作用域内部声明的变量，作用域内部能够获取到作用域外界声明的变量。

61.作用域的分类

答案：块作用域、词法作用域、动态作用域

1 块作用域 花括号 {}

2 词法作用域（js 属于词法作用域） 作用域只跟在何处被创建有关系，跟在何处被调用没有关系

3 动态作用域 作用域只跟在何处被调用有关系，跟在何处被创建没有关系

62.js 属于哪种作用域

答案：词法作用域（函数作用域）

```
// 块作用域
/*{
        var num =123;
    }
    console.log(num);*/
// 如果 js 属于块作用域，那么在花括号外部就无法访问到花括号内部的声明的 num 变量。
// 如果 js 不属于块级作用域，那么花括号外部就能够访问到花括号内部声明的 num 变量
// 能够输出 num 变量，也就说明 js 不属于块级作用。
// 在 ES6 之前的版本 js 是不存在块级作用域的。
//js 属于词法作用域还是动态作用域
// js 中函数可以帮我们去形成一个作用域
/* function fn(){
        var num =123;
    }
    fn();
    //在函数外界能否访问到 num 这样一个变量
    console.log(num)*/ //Uncaught ReferenceError: num is not defined
// 如果函数能够生成一个作用域，那么在函数外界就无法访问到函数内部声明的变量。
// js 中的函数能够生成一个作用。  函数作用域 。
// 词法作用域：作用的外界只跟作用域在何处创建有关系，跟作用域在何处被调用没有关系
var num = 123;
function f1() {
  console.log(num); //
}
function f2() {
  var num = 456;
  f1(); //f1 在 f2 被调用的时候会被执行 。
}
f2();
//如果 js 是词法作用域，那么就会输出 f1 被创建的时候外部的 num 变量 123
//如果 js 是动态作用域，那么 f1 执行的时候就会输出 f1 被调用时外部环境中的 num  456
```

63.浮点数精度

64.自执行函数?用于什么场景？好处?

自执行函数：1、声明一个匿名函数 2、马上调用这个匿名函数。

作用：创建一个独立的作用域。

好处：防止变量弥散到全局，以免各种 js 库冲突。隔离作用域避免污染，或者截断作用域链，避免闭包造成引用变量无法释放。利用立即执行特性，返回需要的业务函数或对象，避免每次通过条件判断来处理

场景：一般用于框架、插件等场景

65.多个页面之间如何进行通信

答案：有如下几个方式：

- cookie
- web worker
- localeStorage 和 sessionStorage

66.css 动画和 js 动画的差异

1. 代码复杂度，js 动画代码相对复杂一些
2. 动画运行时，对动画的控制程度上，js 能够让动画，暂停，取消，终止，css 动画不能添加事件
3. 动画性能看，js 动画多了一个 js 解析的过程，性能不如 css 动画好

67.如何做到修改 url 参数页面不刷新

HTML5 引入了 history.pushState() 和 history.replaceState() 方法，它们分别可以添加和修改历史记录条目。

```
let stateObj = {
  foo: "bar"
};
history.pushState(stateObj, "page 2", "bar.html");
```

假设当前页面为 foo.html，执行上述代码后会变为 bar.html，点击浏览器后退，会变为 foo.html，但浏览器并不会刷新。

pushState() 需要三个参数：一个状态对象，一个标题（目前被忽略），和（可选的）一个 URL. 让我们来解释下这三个参数详细内容：

- 状态对象 — 状态对象 state 是一个 JavaScript 对象，通过 pushState () 创建新的历史记录条目。无论什么时候用户导航到新的状态，popstate 事件就会被触发，且该事件的 state 属性包含该历史记录条目状态对象的副本。 状态对象可以是能被序列化的任何东西。原因在于 Firefox 将状态对象保存在用户的磁盘上，以便在用户重启浏览器时使用，我们规定了状态对象在序列化表示后有 640k 的大小限制。如果你给 pushState() 方法传了一个序列化后大于 640k 的状态对象，该方法会抛出异常。如果你需要更大的空间，建议使用 sessionStorage 以及 localStorage.

- 标题 — Firefox 目前忽略这个参数，但未来可能会用到。传递一个空字符串在这里是安全的，而在将来这是不安全的。二选一的话，你可以为跳转的 state 传递一个短标题。
- URL — 该参数定义了新的历史 URL 记录。注意，调用 pushState() 后浏览器并不会立即加载这个 URL，但可能会在稍后某些情况下加载这个 URL，比如在用户重新打开浏览器时。新 URL 不必须为绝对路径。如果新 URL 是相对路径，那么它将被作为相对于当前 URL 处理。新 URL 必须与当前 URL 同源，否则 pushState() 会抛出一个异常。该参数是可选的，缺省为当前 URL。

68.数组方法 pop() push() unshift() shift()

- arr.pop() 从后面删除元素，只能是一个，返回值是删除的元素
- arr.push() 从后面添加元素，返回值为添加完后的数组的长度
- arr.unshift() 从前面添加元素，返回值是添加完后的数组的长度
- arr.shift() 从前面删除元素，只能删除一个 返回值是删除的元素

69.事件绑定与普通事件有什么区别

- 用普通事件添加相同事件，下面会覆盖上面的，而事件绑定不会
- 普通事件是针对非 dom 元素，事件绑定是针对 dom 元素的事件

70. IE 和 DOM 事件流的区别

1.事件流的区别
IE 采用冒泡型事件 Netscape 使用捕获型事件 DOM 使用先捕获后冒泡型事件 示例：
复制代码代码如下:

```
<body>
  <div>
    <button>点击这里</button>
  </div>
</body>
```

冒泡型事件模型： button->div->body (IE 事件流)
捕获型事件模型： body->div->button (Netscape 事件流)
DOM 事件模型： body->div->button->button->div->body (先捕获后冒泡)
2.事件侦听函数的区别
IE 使用:

```
[Object].attachEvent("name_of_event_handler", fnHandler); //绑定函数
[Object].detachEvent("name_of_event_handler", fnHandler); //移除绑定
```

DOM 使用：

```
[Object].addEventListener("name_of_event", fnHandler, bCapture); //绑定函数
[Object].removeEventListener("name_of_event", fnHandler, bCapture); //移除绑定
```

bCapture 参数用于设置事件绑定的阶段，true 为捕获阶段，false 为冒泡阶段。

71. IE 和标准下有哪些兼容性的写法

```
var ev = ev || window.event;
document.documentElement.clientWidth || document.body.clientWidth;
var target = ev.srcElement || ev.target;
```

73.如何阻止冒泡与默认行为

答案：

- 阻止冒泡行为：非 IE 浏览器 stopPropagation()，IE 浏览器 window.event.cancelBubble = true
- 阻止默认行为：非 IE 浏览器 preventDefault()，IE 浏览器 window.event.returnValue = false

解析：
当需要阻止冒泡行为时，可以使用

```
function stopBubble(e) {
  //如果提供了事件对象，则这是一个非 IE 浏览器
  if (e && e.stopPropagation)
    //因此它支持 W3C 的 stopPropagation()方法
    e.stopPropagation();
  //否则，我们需要使用 IE 的方式来取消事件冒泡
  else window.event.cancelBubble = true;
}
```

当需要阻止默认行为时，可以使用

```
//阻止浏览器的默认行为
function stopDefault(e) {
  //阻止默认浏览器动作(W3C)
  if (e && e.preventDefault) e.preventDefault();
  //IE 中阻止函数器默认动作的方式
  else window.event.returnValue = false;
  return false;
}
```

74.js 中 this 闭包 作用域

答案：
this：指向调用上下文
闭包：定义一个函数就开辟了一个局部作用域，整个 js 执行环境有一个全局作用域

作用域：一个函数可以访问其他函数中的变量（闭包是一个受保护的变量空间）

```
var f = (function fn() {
  var name = 1;
  return function () {
    name++;
    console.log(name)
  }
})()
```

==>undefined 有疑问

75.javascript 的本地对象，内置对象和宿主对象

1. 本地对象 ECMA-262 把本地对象（native object）定义为“独立于宿主环境的 ECMAScript 实现提供的对象”。简单来说，本地对象就是 ECMA-262 定义的类（引用类型）。它们包括：Object、Function、Array、String、Boolean、Number、Date、RegExp、Error、EvalError、RangeError、ReferenceError、SyntaxError、TypeError、URIError
2. 内置对象 JS 中内置了 17 个对象，常用的是 Array 对象、Date 对象、正则表达式对象、string 对象、Global 对象
3. 宿主对象 由 ECMAScript 实现的宿主环境提供的对象，可以理解为：浏览器提供的对象。所有的 BOM 和 DOM 都是宿主对象。

76.javascript 的同源策略

答案：一段脚本只能读取来自于同一来源的窗口和文档的属性

同源策略：限制从一个源加载的文档或脚本如何与来自另一个源的资源进行交互。这是一个用于隔离潜在恶意文件的关键的安全机制。（来自 MDN 官方的解释）

简单来说就是：一段脚本只能读取来自于同一来源的窗口和文档的属性，这里的同一来源指的是主机名、协议和端口号的组合 具体解释：

（1）源包括三个部分：协议、域名、端口（http 协议的默认端口是 80）。如果有任何一个部分不同，则源不同，那就是跨域了。

（2）限制：这个源的文档没有权利去操作另一个源的文档。这个限制体现在：（要记住）

Cookie、LocalStorage 和 IndexDB 无法获取。

无法获取和操作 DOM。

不能发送 Ajax 请求。我们要注意，Ajax 只适合同源的通信。

同源策略带来的麻烦：ajax 在不同域名下的请求无法实现，需要进行跨域操作

77.事件冒泡与事件捕获

事件冒泡：由最具体的元素（目标元素）向外传播到最不具体的元素

事件捕获：由最不确定的元素到目标元素

78.foo = foo||bar ，这行代码是什么意思？为什么要这样写？

这种写法称为短路表达式

相当于

```
var foo;
if (foo) {
  foo = foo;
} else {
  foo = bar;
}
```

常用于函数参数的空判断

79.复杂数据类型如何转变为字符串

- 首先，会调用 valueOf 方法，如果方法的返回值是一个基本数据类型，就返回这个值
- 如果调用 valueOf 方法之后的返回值仍旧是一个复杂数据类型，就会调用该对象的 toString 方法
- 如果 toString 方法调用之后的返回值是一个基本数据类型，就返回这个值，
- 如果 toString 方法调用之后的返回值是一个复杂数据类型，就报一个错误。

80.javascript 中 this 的指向问题

- 全局环境、普通函数（非严格模式）指向 window
- 普通函数（严格模式）指向 undefined
- 函数作为对象方法及原型链指向的就是上一级的对象
- 构造函数指向构造的对象
- DOM 事件中指向触发事件的元素
- 箭头函数...

1、全局环境

全局环境下，this 始终指向全局对象（window），无论是否严格模式；

```
// 在浏览器中，全局对象为 window 对象：
console.log(this === window); // true

this.a = 37;
console.log(window.a); // 37
```

2、函数上下文调用

2.1 普通函数

普通函数内部的 this 分两种情况，严格模式和非严格模式。

（1）非严格模式下，没有被上一级的对象所调用,this 默认指向全局对象 window。

```
function f1() {
  return this;
```

```
}
f1() === window; // true
```
（2）严格模式下，this 指向 undefined。
```
function f2() {
  "use strict"; // 这里是严格模式
  return this;
}
f2() === undefined; // true
```
2.2 函数作为对象的方法

（1）函数有被上一级的对象所调用，那么 this 指向的就是上一级的对象。

（2）多层嵌套的对象，内部方法的 this 指向离被调用函数最近的对象（window 也是对象，其内部对象调用方法的 this 指向内部对象， 而非 window）。
```
//方式 1
var o = {
  prop: 37,
  f: function() {
    return this.prop;
  }
};
//当 o.f()被调用时，函数内的 this 将绑定到 o 对象。
console.log(o.f()); // logs 37
//方式 2
var o = { prop: 37 };
function independent() {
  return this.prop;
}
//函数 f 作为 o 的成员方法调用
o.f = independent;
console.log(o.f()); // logs 37
//方式 3
//this 的绑定只受最靠近的成员引用的影响
o.b = { g: independent, prop: 42 };
console.log(o.b.g()); // 42
```
特殊例子
```
// 例子 1
var o = {
  a: 10,
  b: {
    // a:12,
    fn: function() {
      console.log(this.a); //undefined
      console.log(this); //{fn: f}
    }
  }
};
o.b.fn();
// 例子 2
var o = {
  a: 10,
  b: {
    a: 12,
    fn: function() {
      console.log(this.a); //undefined
      console.log(this); //window
    }
  }
};
var j = o.b.fn;
j();
```
// this 永远指向的是最后调用它的对象，也就是看它执行的时候是谁调用的，例子 2 中虽然函数 fn 是被对象 b 所引用，但是在将 fn 赋值给变量 j 的时候并没有执行所以最终指向的是 window，这和例子 1 是不一样的，例子 1 是直接执行了 fn

2.3 原型链中的 this

（1）如果该方法存在于一个对象的原型链上，那么 this 指向的是调用这个方法的对象，就像该方法在对象上一样。
```
var o = {
  f: function() {
```

```
    return this.a + this.b;
  }
};
var p = Object.create(o);
p.a = 1;
p.b = 4;
console.log(p.f()); // 5
```

上述例子中，对象 p 没有属于它自己的 f 属性，它的 f 属性继承自它的原型。当执行 p.f()时，会查找 p 的原型链，找到 f 函数并执行。因为 f 是作为 p 的方法调用的，所以函数中的 this 指向 p。

（2）相同的概念也适用于当函数在一个 getter 或者 setter 中被调用。用作 getter 或 setter 的函数都会把 this 绑定到设置或获取属性的对象。

（3）call()和 apply()方法：当函数通过 Function 对象的原型中继承的方法 call() 和 apply() 方法调用时， 其函数内部的 this 值可绑定到 call() & apply() 方法指定的第一个对象上， 如果第一个参数不是对象，JavaScript 内部会尝试将其转换成对象然后指向它。

```
function add(c, d) {
  return this.a + this.b + c + d;
}
var o = { a: 1, b: 3 };
add.call(o, 5, 7); // 1 + 3 + 5 + 7 = 16
add.apply(o, [10, 20]); // 1 + 3 + 10 + 20 = 34
function tt() {
  console.log(this);
}
// 第一个参数不是对象，JavaScript 内部会尝试将其转换成对象然后指向它。
tt.call(5); // 内部转成 Number {[[PrimitiveValue]]: 5}
tt.call("asd"); // 内部转成 String {0: "a", 1: "s", 2: "d", length: 3, [[PrimitiveValue]]: "asd"}
```

（4）bind()方法：由 ES5 引入， 在 Function 的原型链上， Function.prototype.bind。通过 bind 方法绑定后， 函数将被永远绑定在其第一个参数对象上， 而无论其在什么情况下被调用。

```
function f() {
  return this.a;
}
var g = f.bind({ a: "azerty" });
console.log(g()); // azerty
var o = { a: 37, f: f, g: g };
console.log(o.f(), o.g()); // 37, azerty
```

2.4 构造函数中的 this

当一个函数用作构造函数时（使用 new 关键字），它的 this 被绑定到正在构造的新对象。

构造器返回的默认值是 this 所指的那个对象，也可以手动返回其他的对象。

```
function C() {
  this.a = 37;
}
var o = new C();
console.log(o.a); // 37
// 为什么 this 会指向 o? 首先 new 关键字会创建一个空的对象，然后会自动调用一个函数 apply 方法，将 this 指向这个空对象，这样的话函数内部的 this 就会被这个空的对象替代。
function C2() {
  this.a = 37;
  return { a: 38 }; // 手动设置返回{a:38}对象
}
o = new C2();
console.log(o.a); // 38
```

特殊例子

当 this 碰到 return 时

```
// 例子 1
function fn() {
  this.user = "追梦子";
  return {};
}
var a = new fn();
console.log(a.user); //undefined
// 例子 2
function fn() {
  this.user = "追梦子";
  return function() {};
}
```

```
var a = new fn();
console.log(a.user); //undefined
// 例子 3
function fn() {
  this.user = "追梦子";
  return 1;
}
var a = new fn();
console.log(a.user); //追梦子
// 例子 4
function fn() {
  this.user = "追梦子";
  return undefined;
}
var a = new fn();
console.log(a.user); //追梦子
// 例子 5
function fn() {
  this.user = "追梦子";
  return undefined;
}
var a = new fn();
console.log(a); //fn {user: "追梦子"}
// 例子 6
// 虽然 null 也是对象，但是在这里 this 还是指向那个函数的实例，因为 null 比较特殊
function fn() {
  this.user = "追梦子";
  return null;
}
var a = new fn();
console.log(a.user); //追梦子
// 总结：如果返回值是一个对象，那么 this 指向的就是那个返回的对象，如果返回值不是一个对象那么 this 还是指向函数的实例。
```

2.5 setTimeout & setInterval

（1）对于延时函数内部的回调函数的 this 指向全局对象 window；

（2）可以通过 bind()方法改变内部函数 this 指向。

```
//默认情况下代码
function Person() {
  this.age = 0;
  setTimeout(function() {
    console.log(this);
  }, 3000);
}
var p = new Person(); //3 秒后返回 window 对象
//通过 bind 绑定
function Person() {
  this.age = 0;
  setTimeout(
    function() {
      console.log(this);
    }.bind(this),
    3000
  );
}
var p = new Person(); //3 秒后返回构造函数新生成的对象 Person{...}
```

3、在 DOM 事件中

3.1 作为一个 DOM 事件处理函数

当函数被用作事件处理函数时，它的 this 指向触发事件的元素（针对 addEventListener 事件）。

```
// 被调用时，将关联的元素变成蓝色
function bluify(e) {
  //this 指向所点击元素
  console.log("this === e.currentTarget", this === e.currentTarget); // 总是 true
  // 当 currentTarget 和 target 是同一个对象时为 true
  console.log("this === e.target", this === e.target);
  this.style.backgroundColor = "#A5D9F3";
```

```
}
// 获取文档中的所有元素的列表
var elements = document.getElementsByTagName("*");
// 将 bluify 作为元素的点击监听函数，当元素被点击时，就会变成蓝色
for (var i = 0; i < elements.length; i++) {
  elements[i].addEventListener("click", bluify, false);
}
```

3.2 作为一个内联事件处理函数

（1）当代码被内联处理函数调用时，它的 this 指向监听器所在的 DOM 元素；

（2）当代码被包括在函数内部执行时，其 this 指向等同于 普通函数直接调用的情况，即在非严格模式指向全局对象 window，在严格模式指向 undefined：

```
<button onclick="console.log(this)">show me</button>
<button onclick="(function () {console.log(this)})()">show inner this</button>
<button onclick="(function () {'use strict'; console.log(this)})()">
  use strict
</button>
// 控制台打印
<button onclick="console.log(this)">show me</button>
Window  {postMessage: ƒ, blur: ƒ, focus: ƒ, close: ƒ, parent: Window,  …}
undefined
```

4、箭头函数

4.1 全局环境中

在全局代码中，箭头函数被设置为全局对象：

```
var globalObject = this;
var foo = () => this;
console.log(foo() === globalObject); // true
```

4.2 this 捕获上下文

箭头函数没有自己的 this，而是使用箭头函数所在的作用域的 this，即指向箭头函数定义时（而不是运行时）所在的作用域。

```
//1、箭头函数在函数内部，以非方法的方法使用
function Person() {
  this.age = 0;
  setInterval(() => {
    this.age++;
  }, 3000);
}
var p = new Person(); //Person{age: 0}
//普通函数作为内部函数
function Person() {
  this.age = 0;
  setInterval(function() {
    console.log(this);
    this.age++;
  }, 3000);
}
var p = new Person(); //Window{...}
```

4.2 this 捕获上下文

箭头函数没有自己的 this，而是使用箭头函数所在的作用域的 this，即指向箭头函数定义时（而不是运行时）所在的作用域。

```
//1、箭头函数在函数内部，以非方法的方法使用
function Person() {
  this.age = 0;
  setInterval(() => {
    console.log(this);
    this.age++;
  }, 3000);
}
var p = new Person(); //Person{age: 0}

//普通函数作为内部函数
function Person() {
  this.age = 0;
  setInterval(function() {
    console.log(this);
    this.age++;
  }, 3000);
}
```

```
var p = new Person(); //Window{...}
```
在 setTimeout 中的 this 指向了构造函数新生成的对象，而普通函数指向了全局 window 对象。

4.3 箭头函数作为对象的方法使用

箭头函数作为对象的方法使用，指向全局 window 对象；而普通函数作为对象的方法使用，则指向调用的对象。

```
var obj = {
  i: 10,
  b: () => console.log(this.i, this),
  c: function() {
    console.log(this.i, this);
  }
};
obj.b(); // undefined window{...}
obj.c(); // 10 Object {...}
```

4.4 箭头函数中，call()、apply()、bind()方法无效

```
var adder = {
  base: 1,
  //对象的方法内部定义箭头函数，this 是箭头函数所在的作用域的 this，
  //而方法 add 的 this 指向 adder 对象，所以箭头函数的 this 也指向 adder 对象。
  add: function(a) {
    var f = v => v + this.base;
    return f(a);
  },
  //普通函数 f1 的 this 指向 window
  add1: function() {
    var f1 = function() {
      console.log(this);
    };
    return f1();
  },
  addThruCall: function inFun(a) {
    var f = v => v + this.base;
    var b = {
      base: 2
    };
    return f.call(b, a);
  }
};

console.log(adder.add(1)); // 输出 2
adder.add1(); //输出全局对象 window{...}
console.log(adder.addThruCall(1)); // 仍然输出 2（而不是 3，其内部的 this 并没有因为 call() 而改变，其 this 值仍然为函数 inFun 的 this 值，指向对象 adder
```

4.5 this 指向固定化

箭头函数可以让 this 指向固定化，这种特性很有利于封装回调函数

```
var handler = {
  id: "123456",

  init: function() {
    document.addEventListener(
      "click",
      event => this.doSomething(event.type),
      false
    );
  },

  doSomething: function(type) {
    console.log("Handling " + type + " for " + this.id);
  }
};
```

上面代码的 init 方法中，使用了箭头函数，这导致这个箭头函数里面的 this，总是指向 handler 对象。如果不使用箭头函数则指向全局 document 对象。

4.6 箭头函是不适用场景

（1）箭头函数不适合定义对象的方法（方法内有 this），因为此时指向 window；

（2）需要动态 this 的时候，也不应使用箭头函数。

//例 1，this 指向定义箭头函数所在的作用域，它位于对象 cat 内，但 cat 不能构成一个作用域，所以指向全局 window，改成普通函数后 this 指向 cat 对象。

```
const cat = {
  lives: 9,
  jumps: () => {
    this.lives--;
  }
};

//例 2，此时 this 也是指向 window，不能动态监听 button，改成普通函数后 this 指向按钮对象。
var button = document.getElementById("press");
button.addEventListener("click", () => {
  this.classList.toggle("on");
});
```

81.call 与 apply 区别

答案：第二个参数的类型不同

call 和 apply 的作用，完全一样，唯一的区别就是在参数上面。

call 接收的参数不固定，第一个参数是函数体内 this 的指向，第二个参数以下是依次传入的参数。

apply 接收两个参数，第一个参数也是函数体内 this 的指向。第二个参数是一个集合对象（数组或者类数组）

82.正则表达式构造函数 var reg = new RegExp('xxx')与正则表达字面量 var reg = // 有什么不同?

答案：使用正则表达字面量的效率更高

解析：下面的示例代码演示了两种可用于创建正则表达式以匹配反斜杠的方法：

```
//正则表达字面量
var re = /\\/gm;
//正则构造函数
var reg = new RegExp("\\\\", "gm");
var foo = "abc\\123"; // foo 的值为"abc\123"
console.log(re.test(foo)); //true
console.log(reg.test(foo)); //true
```

如上面的代码中可以看到，使用正则表达式字面量表示法时式子显得更加简短，而且不用按照类似类（class-like）的构造函数方式思考。

其次，在当使用构造函数的时候，在这里要使用四个反斜杠才能匹配单个反斜杠。这使得正则表达式模式显得更长，更加难以阅读和修改。正确来说，当使用 RegExp()构造函数的时候，不仅需要转义引号（即"表示"），并且通常还需要双反斜杠（即\表示一个\）。

使用 new RegExp()的原因之一在于，某些场景中无法事先确定模式，而只能在运行时以字符串方式创建。

83.js 中 callee 与 caller 的作用

1. caller 返回一个调用当前函数的引用 如果是由顶层调用的话 则返回 null

（举个栗子哈 caller 给你打电话的人 谁给你打电话了 谁调用了你 很显然是下面 a 函数的执行 只有在打电话的时候你才能知道打电话的人是谁 所以对于函数来说 只有 caller 在函数执行的时候才存在）

```
var callerTest = function() {
  console.log(callerTest.caller);
};
function a() {
  callerTest();
}
a(); //输出 function a() {callerTest();}
callerTest(); //输出 null
```

2. callee 返回一个正在被执行函数的引用 （这里常用来递归匿名函数本身 但是在严格模式下不可行）
   callee 是 arguments 对象的一个成员 表示对函数对象本身的引用 它有个 length 属性（代表形参的长度）

```
var c = function(x, y) {
  console.log(arguments.length, arguments.callee.length, arguments.callee);
};
c(1, 2, 3); //输出 3 2 function(x,y) {console.log(arguments.length,arguments.callee.length,arguments.callee)}
```

84.异步加载 js 的方法

方案一：<script>标签的 async="async"属性（详细参见：script 标签的 async 属性）

点评：HTML5 中新增的属性，Chrome、FF、IE9&IE9+均支持（IE6~8 不支持）。此外，这种方法不能保证脚本按顺序执行。

方案二：<script>标签的 defer="defer"属性

点评：兼容所有浏览器。此外，这种方法可以确保所有设置 defer 属性的脚本按顺序执行。

方案三：动态创建<script>标签

示例：

```
<!DOCTYPE html>
<html>
  <head>
    <script type="text/javascript">
      (function() {
```

```
        var s = document.createElement_x("script");
        s.type = "text/javascript";
        s.src = "http://code.jquery.com/jquery-1.7.2.min.js";
        var tmp = document.getElementsByTagName_r("script")[0];
        tmp.parentNode.insertBefore(s, tmp);
      })();
    </script>
  </head>
  <body>
    <img src="http://xybtv.com/uploads/allimg/100601/48-100601162913.jpg" />
  </body>
</html>
```

点评：兼容所有浏览器。

方案四：AJAX eval（使用 AJAX 得到脚本内容，然后通过 eval_r(xmlhttp.responseText)来运行脚本）

点评：兼容所有浏览器。

方案五：iframe 方式（这里可以参照：iframe 异步加载技术及性能 中关于 Meboo 的部分）

点评：兼容所有浏览器。

85.去除数组重复成员的方法

方法 1 扩展运算符和 Set 结构相结合，就可以去除数组的重复成员

```
// 去除数组的重复成员
[...new Set([1, 2, 2, 3, 4, 5, 5])];
// [1, 2, 3, 4, 5]
```

方法 2

```
function dedupe(array) {
  return Array.from(new Set(array));
}
dedupe([1, 1, 2, 3]); // [1, 2, 3]
```

方法 3（ES5）

```
function unique(arry) {
  const temp = [];
  arry.forEach(e => {
    if (temp.indexOf(e) == -1) {
      temp.push(e);
    }
  });

  return temp;
}
```

86.去除字符串里面的重复字符

```
[...new Set("ababbc")].join(""); // "abc"
```

87.求数组的最大值

答案：Math.max.apply(null, 数组)

```
var a = [1, 2, 3, 5];
alert(Math.max.apply(null, a)); //最大值
alert(Math.min.apply(null, a)); //最小值
```

88.JS 中 文档碎片的理解和使用

```
// 1、什么是文档碎片？
document.createDocumentFragment(); // 一个容器，用于暂时存放创建的 dom 元素
// 2、文档碎片有什么用？
// 将需要添加的大量元素,先添加到文档碎片中，再将文档碎片添加到需要插入的位置，大大 减少 dom 操作，提高性能（IE 和火狐比较明显）
```

解析：

```
// 普通方式：（操作了 100 次 dom）
for (var i = 100; i > 0; i--) {
  var elem = document.createElement("div");
  document.body.appendChild(elem); //放到 body 中
}
//  文档碎片：(操作 1 次 dom)
var df = document.createDocumentFragment();
for (var i = 100; i > 0; i--) {
  var elem = document.createElement("div");
  df.appendChild(elem);
}
//最后放入到页面上
document.body.appendChild(df);
```

89.原型的作用 以及什么是原型

答案：作用：实现资源共享

什么是原型:实例在被创建的那一刻，构造函数的 prototype 属性的值。

90.javascript 里面的继承怎么实现，如何避免原型链上面的对象共享

答案：用构造函数和原型链的混合模式去实现继承，避免对象共享可以参考经典的 extend()函数，很多前端框架都有封装的，就是用一个空函数当做中间变量

91.简单介绍下 JS 的原型和原型链

92.说说你对作用域链的理解

答案：作用域链的作用是保证执行环境里有权访问的变量和函数是有序的，作用域链的变量只能向上访问，变量访问到 window 对象即被终止，作用域链向下访问变量是不被允许的。

93.JavaScript 原型，原型链 ？有什么特点？

- 原型对象也是普通的对象，是对象一个自带隐式的__proto__属性，原型也有可能有自己的原型，如果一个原型对象的原型不为 null 的话，我们就称之为原型链。
- 原型链是由一些用来继承和共享属性的对象组成的（有限的）对象链。
- JavaScript 的数据对象有那些属性值？ writable：这个属性的值是否可以改。 configurable：这个属性的配置是否可以删除，修改。 enumerable：这个属性是否能在 for…in 循环中遍历出来或在 Object.keys 中列举出来。 value：属性值。
- 当我们需要一个属性的时，Javascript 引擎会先看当前对象中是否有这个属性， 如果没有的话，就会查找他的 Prototype 对象是否有这个属性。

```
function clone(proto) {
  function Dummy() {}
  Dummy.prototype = proto;
  Dummy.prototype.constructor = Dummy;
  return new Dummy(); //等价于 Object.create(Person);
}
function object(old) {
  function F() {}
  F.prototype = old;
  return new F();
}
var newObj = object(oldObject);
```

94.请解释什么是事件代理

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

96.谈谈你对 AMD、CMD 的理解

97.web 开发中会话跟踪的方法有哪些

98.说几条写 JavaScript 的基本规范？

99.JavaScript 有几种类型的值？你能画一下他们的内存图吗？

100.eval 是做什么的？

1. 它的功能是把对应的字符串解析成 JS 代码并运行
2. 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）

101.js 延迟加载的方式有哪些？

答案：defer 和 async、动态创建 DOM 方式（用得最多）、按需异步载入 js

102.attribute 和 property 的区别是什么？

103.什么是面向对象编程及面向过程编程，它们的异同和优缺点

104.谈一谈你理解的函数式编程？

105.对原生 Javascript 了解程度

106.Js 动画与 CSS 动画区别及相应实现

107.快速的让一个数组乱序

```
const arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20] console.log(arr.sort(() => 0.5 - Math.random()))
```

108.prototype 和__proto__的关系是什么？

109.UIWebView 和 JavaScript 之间是怎么交互的?

110.IE 与火狐的事件机制有什么区别？如何阻止冒泡

1. 我们在网页中的某个操作（有的操作对应多个事件）。例如：当我们点击一个按钮就会产生一个事件。是可以被 JavaScript 侦测到的行为。
2. 事件处理机制：IE 是事件冒泡、火狐是 事件捕获；
3. ev.stopPropagation();

111.在 js 中哪些会被隐式转换为 false

答案：Undefined、null、关键字 false、NaN、零、空字符串

112.列举浏览器对象模型 BOM 里常用的至少 4 个对象，并列举 window 对象的常用方法至少 5 个？

对象：Window，document，location，screen，history，navigator。 方法：Alert()，confirm()，prompt()，open()，close()。

113.class.forname 的作用?为什么要用?

1、获取 Class 对象的方式：类名.class、对象.getClass()、Class.forName(“类名”);

2、通过 Class 对象自审

3、动态调用方法

114.外部 JS 文件出现中文字符，会出现什么问题，怎么解决？

答案：会出现乱码，加 charset="GB2312";

115.定时器 setInterval 有一个有名函数 fn1，setInterval（fn1,500）与 setInterval（fn1(),500）有什么区别？

答案：第一个是重复执行每 500 毫秒执行一次，后面一个只执行一次。

116.自动分号

答案：有时 JavaScript 会自动为代码行补上缺失的分号，即自动分号插入（Automatic SemicolonInsertion，ASI）。
因为如果缺失了必要的 ；，代码将无法运行，语言的容错性也会降低。ASI 能让我们忽略那些不必要的。
请注意，ASI 只在换行符处起作用，而不会在代码行的中间插入分号。
如果 JavaScript 解析器发现代码行可能因为缺失分号而导致错误，那么它就会自动补上分号。并且，只有在代码行末尾与换行符之间除了空格和注释之外没有别的内容时，它才会这样做。

117.你用过 require.js 吗？它有什么特性？

（1）实现 js 文件的异步加载，避免网页失去响应； （2）管理模块之间的依赖性，便于代码的编写和维护。

118.如何阻止事件冒泡和默认事件？

阻止浏览器的默认行为 window.event?window.event.returnValue=false:e.preventDefault();
停止事件冒泡 window.event?window.event.cancelBubble=true:e.stopPropagation(); 原生 JavaScript 中，return false;只阻止默认行为，不阻止冒泡，jQuery 中的 return false;既阻止默认行为，又阻止冒泡

119.分别阐述 split(),slice(),splice(),join()？

- join()用于把数组中的所有元素拼接起来放入一个字符串。所带的参数为分割字符串的分隔符，默认是以逗号分开。归属于 Array
- split()即把字符串分离开，以数组方式存储。归属于 Stringstring
- slice() 方法可从已有的数组中返回选定的元素。该方法并不会修改数组，而是返回一个子数组。如果想删除数组中的一段元素，应该使用方法 Array.splice()
- splice() 方法向/从数组中添加/删除项目，然后返回被删除的项目。返回的是含有被删除的元素的数组。

120.事件、IE 与火狐的事件机制有什么区别？ 如何阻止冒泡？

1. 我们在网页中的某个操作（有的操作对应多个事件）。例如：当我们点击一个按钮就会产生一个事件。是可以被 JavaScript 侦测到的行为
2. 事件处理机制：IE 是事件冒泡、firefox 同时支持两种事件模型，也就是：捕获型事件和冒泡型事件
3. ev.stopPropagation(); 注意旧 ie 的方法：ev.cancelBubble = true;

121.内置函数(原生函数)

- String
- Number
- Boolean
- Object
- Function
- Array
- Date
- RegExp
- Error
- Symbol

122.对象浅拷贝和深拷贝有什么区别

答案：在 JS 中，除了基本数据类型，还存在对象、数组这种引用类型。 基本数据类型，拷贝是直接拷贝变量的值，而引用类型拷贝的其实是变量的地址。

```
let o1 = {a: 1}
let o2 = o1
```

在这种情况下，如果改变 o1 或 o2 其中一个值的话，另一个也会变，因为它们都指向同一个地址。

```
o2.a = 3
console.log(o1.a) // 3
```

而浅拷贝和深拷贝就是在这个基础之上做的区分，如果在拷贝这个对象的时候，只对基本数据类型进行了拷贝，而对引用数据类型只是进行了引用的传递，而没有重新创建一个新的对象，则认为是浅拷贝。反之，在对引用数据类型进行拷贝的时候，创建了一个新的对象，并且复制其内的成员变量，则认为是深拷贝。

123.JS 怎么实现一个类。怎么实例化这个类

答案：严格来讲 js 中并没有类的概念，不过 js 中的函数可以作为构造函数来使用，通过 new 来实例化，其实函数本身也是一个对象。

124.如何编写高性能的 Javascript？

- 使用 DocumentFragment 优化多次 append
- 通过模板元素 clone ，替代 createElement
- 使用一次 innerHTML 赋值代替构建 dom 元素
- 使用 firstChild 和 nextSibling 代替 childNodes 遍历 dom 元素
- 使用 Array 做为 StringBuffer ，代替字符串拼接的操作
- 将循环控制量保存到局部变量
- 顺序无关的遍历时，用 while 替代 for
- 将条件分支，按可能性顺序从高到低排列
- 在同一条件子的多（ >2 ）条件分支时，使用 switch 优于 if
- 使用三目运算符替代条件分支
- 需要不断执行的时候，优先考虑使用 setInterval

125.数组和对象有哪些原生方法，列举一下？

- Array.concat( ) 连接数组
- Array.join( ) 将数组元素连接起来以构建一个字符串
- Array.length 数组的大小
- Array.pop( ) 删除并返回数组的最后一个元素
- Array.push( ) 给数组添加元素
- Array.reverse( ) 颠倒数组中元素的顺序
- Array.shift( ) 将元素移出数组
- Array.slice( ) 返回数组的一部分
- Array.sort( ) 对数组元素进行排序
- Array.splice( ) 插入、删除或替换数组的元素
- Array.toLocaleString( ) 把数组转换成局部字符串
- Array.toString( ) 将数组转换成一个字符串
- Array.unshift( ) 在数组头部插入一个元素
- Object.hasOwnProperty( ) 检查属性是否被继承
- Object.isPrototypeOf( ) 一个对象是否是另一个对象的原型
- Object.propertyIsEnumerable( ) 是否可以通过 for/in 循环看到属性
- Object.toLocaleString( ) 返回对象的本地字符串表示
- Object.toString( ) 定义一个对象的字符串表示
- Object.valueOf( ) 指定对象的原始值

126.documen.write 和 innerHTML 的区别?

1. document.write 是重写整个 document, 写入内容是字符串的 html
2. innerHTML 是 HTMLElement 的属性，是一个元素的内部 html 内容

127.让你自己设计实现一个 requireJS，你会怎么做?

答案：核心是实现 js 的加载模块，维护 js 的依赖关系，控制好文件加载的先后顺序

128.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？）

答案：核心是 js 的加载模块，通过正则匹配模块以及模块的依赖关系，保证文件加载的先后顺序，根据文件的路径对加载过的文件做了缓存

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

答案：HasOwnProperty

130.原型继承

答案：所有的 JS 对象都有一个 prototype 属性，指向它的原型对象。当试图访问一个对象的属性时，如果没有在该对象上找到，它还会搜寻该对象的原型，以及该对象的原型的原型，依次层层向上搜索，直到找到一个名字匹配的属性或到达原型链的末尾。

131.用原生 JavaScript 的实现过什么功能吗?

答案：轮播图、手风琴、放大镜、3D 动画效果等，切记，所答的一定要知道实现原理！，不知道还不如不说！

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

答案：意思是使用严格模式，使用严格模式，一些不规范的语法将不再支持

133.简述创建函数的几种方式

第一种（函数声明）：

```
function sum1(num1,num2){
   return num1+num2;
}
```

第二种（函数表达式）：

```
var sum2 = function(num1,num2){
   return num1+num2;
}
```

第三种（函数对象方式）：

```
var sum3 = new Function("num1","num2","return num1+num2");
```

134.window.location.search() 返回的是什么?

答案：查询(参数)部分。除了给动态语言赋值以外，我们同样可以给静态页面,并使用 javascript 来获得相信应的参数值 返回值：?ver=1.0&id=timlq 也就是问号后面的!

135.window.location.hash 返回的是什么?

答案：锚点 ， 返回值：#love ；

136.window.location.reload() 作用?

答案：刷新当前页面

137.为什么不能定义 1px 左右的 div 容器?

答案： IE6 下这个问题是因为默认的行高造成的，解决的方法也有很多，例如： overflow:hidden | zoom:0.08 | line-height:1px

138.BOM 对象有哪些，列举 window 对象?

1、window 对象 ，是 JS 的最顶层对象，其他的 BOM 对象都是 window 对象的属性；
2、document 对象，文档对象；
3、location 对象，浏览器当前 URL 信息；
4、navigator 对象，浏览器本身信息；
5、screen 对象，客户端屏幕信息；
6、history 对象，浏览器访问历史信息；

139.简述 readonly 与 disabled 的区别
140.为什么扩展 javascript 内置对象不是好的做法？
141.什么是三元表达式？“三元”表示什么意思？
142.我们给一个 dom 同时绑定两个点击事件，一个用捕获，一个用冒泡，你来说下会执行几次事件，然后会先执行冒泡还是捕获
144.简述一下 Handlebars 的基本用法？
答案：没有用过的话说出它是干什么的即可
143.简述一下 Handlerbars 的对模板的基本处理流程， 如何编译的？如何缓存的？
145.前端 templating(Mustache, underscore, handlebars)是干嘛的, 怎么用？

- Web 模板引擎是为了使用户界面与业务数据（内容）分离而产生的，
- Mustache 是一个 logic-less （轻逻辑）模板解析引擎，它的优势在于可以应用在 Javascript、PHP、Python、Perl 等多种编程语言中。
- Underscore 封装了常用的 JavaScript 对象操作方法，用于提高开发效率。
- Handlebars 是 JavaScript 一个语义模板库，通过对 view 和 data 的分离来快速构建 Web 模板。

146.知道什么是 webkit 么？知道怎么用浏览器的各种工具来调试和 debug 代码么？
答案：Webkit 是浏览器引擎，包括 html 渲染和 js 解析功能，手机浏览器的主流内核，与之相对应的引擎有 Gecko（Mozilla Firefox 等使用）和 Trident（也称 MSHTML，IE 使用）。 对于浏览器的调试工具要熟练使用，主要是页面结构分析，后台请求信息查看，js 调试工具使用，熟练使用这些工具可以快速提高解决问题的效率
147.如何测试前端代码？知道 BDD, TDD, Unit Test 么？知道怎么测试你的前端工程么(mocha, sinon, jasmin, qUnit..)？
答案：了解 BDD 行为驱动开发与 TDD 测试驱动开发已经单元测试相关概念
148.JavaScript 的循环语句有哪些？
答案：while for do while forEach
149.作用域-编译期执行期以及全局局部作用域问题
答案：js 执行主要的两个阶段：预解析和执行期
150.如何添加 html 元素的事件，有几种方法？请列举
答案：直接在标签里添加；在元素上添加、使用事件注册函数添加
151.列举浏览器对象模型 BOM 里常用的至少 4 个对象，并列举 window 对象的常用方法至少 5 个
对象：Window document location screen history navigator
方法：Alert() confirm() prompt() open() close()
152.事件绑定的方式

- 嵌入 dom

```
<button onclick="func()">按钮</button>
```

- 直接绑定

```
btn.onclick = function() {};
```

- 事件监听

```
btn.addEventListener("click", function() {});
```

153.事件循环
答案：事件循环是一个单线程循环，用于监视调用堆栈并检查是否有工作即将在任务队列中完成。如果调用堆栈为空并且任务队列中有回调函数，则将回调函数出队并推送到调用堆栈中执行。
154.事件模型

- DOM0

直接绑定

```
<input onclick="sayHi()"/>
btn.onclick = function() {}
btn.onclick = null
```

DOM2

```
DOM2 级事件可以冒泡和捕获 通过 addEventListener 绑定 通过 removeEventListener 解绑
// 绑定
btn.addEventListener('click', sayHi)
// 解绑
btn.removeEventListener('click', sayHi)
```

- DOM3

DOM3 具有更多事件类型 DOM3 级事件在 DOM2 级事件的基础上添加了更多的事件类型，全部类型如下：

```
UI 事件，当用户与页面上的元素交互时触发，如：load、scroll
焦点事件，当元素获得或失去焦点时触发，如：blur、focus
鼠标事件，当用户通过鼠标在页面执行操作时触发如：dbclick、mouseup
滚轮事件，当使用鼠标滚轮或类似设备时触发，如：mousewheel
文本事件，当在文档中输入文本时触发，如：textInput
键盘事件，当用户通过键盘在页面上执行操作时触发，如：keydown、keypress
合成事件，当为 IME（输入法编辑器）输入字符时触发，如：compositionstart
变动事件，当底层 DOM 结构发生变化时触发，如：DOMsubtreeModified
```

155.如何自定义事件

1. 原生提供了 3 个方法实现自定义事件
2. createEvent，设置事件类型，是 html 事件还是 鼠标事件
3. initEvent 初始化事件，事件名称，是否允许冒泡，是否阻止自定义事件
4. dispatchEvent 触发事件

156.target 和 currentTarget 区别

答案：

- event.target
  返回触发事件的元素
- event.currentTarget
  返回绑定事件的元素

157.prototype 和__proto__的关系是什么

所有的对象都拥有__proto__属性，它指向对象构造函数的 prototype 属性

```
let obj = {}
obj.__proto__ === Object.prototype // true

function Test(){}
test.__proto__ == Test.prototype // true
```

所有的函数都同时拥有__proto__和 protytpe 属性 函数的__proto__指向自己的函数实现 函数的 protytpe 是一个对象 所以函数的 prototype 也有__proto__属性 指向 Object.prototype

```
function func() {}
func.prototype.__proto__ === Object.prototype // true
Object.prototype.__proto__指向 null
Object.prototype.__proto__ // null
```

158.什么是原型属性？

答案：从构造函数的 prototype 属性出发找到原型，这时候就把原型称之为构造函数的原型属性

159.什么是原型对象？

答案：从实例的__proto__出发，找到原型，这时候就把原型称之为实例的原型对象。

160.使用 let、var 和 const 创建变量有什么区别

用 var 声明的变量的作用域是它当前的执行上下文，它可以是嵌套的函数，也可以是声明在任何函数外的变量。let 和 const 是块级作用域，意味着它们只能在最近的一组花括号（function、if-else 代码块或 for 循环中）中访问。

```
function foo() {
  // 所有变量在函数中都可访问
  var bar = "bar";
  let baz = "baz";
  const qux = "qux";
  console.log(bar); // bar
  console.log(baz); // baz
  console.log(qux); // qux
}
console.log(bar); // ReferenceError: bar is not defined
console.log(baz); // ReferenceError: baz is not defined
console.log(qux); // ReferenceError: qux is not defined
if (true) {
  var bar = "bar";
  let baz = "baz";
  const qux = "qux";
}
// 用 var 声明的变量在函数作用域上都可访问
console.log(bar); // bar
// let 和 const 定义的变量在它们被定义的语句块之外不可访问
console.log(baz); // ReferenceError: baz is not defined
console.log(qux); // ReferenceError: qux is not defined
```

var 会使变量提升，这意味着变量可以在声明之前使用。let 和 const 不会使变量提升，提前使用会报错。

```
console.log(foo); // undefined
var foo = "foo";
console.log(baz); // ReferenceError: can't access lexical declaration 'baz' before initialization
let baz = "baz";
console.log(bar); // ReferenceError: can't access lexical declaration 'bar' before initialization
const bar = "bar";
```

用 var 重复声明不会报错，但 let 和 const 会。

```
var foo = "foo";
var foo = "bar";
console.log(foo); // "bar"
let baz = "baz";
let baz = "qux"; // Uncaught SyntaxError: Identifier 'baz' has already been declared
```

let 和 const 的区别在于：let 允许多次赋值，而 const 只允许一次。

```
// 这样不会报错。
let foo = "foo";
```

```
foo = "bar";
// 这样会报错。
const baz = "baz";
baz = "qux";
```

161.JSON 的了解

答案：JSON(JavaScript Object Notation) 是一种轻量级的数据交换格式。它是基于 JavaScript 的一个子集。数据格式简单, 易于读写, 占用带宽小。

162.事件代理怎么实现？

答案：在元素的父节点注册事件，通过事件冒泡，在父节点捕获事件

163.什么是属性搜索原则？

1. 首先会去查找对象本身上面有没有这个属性，有的话，就返回这个属性
2. 如果对象本身上面没有这个属性，就到它的原型上面去查找，如果有，就返回
3. 就到原型的原型上面去查找有没有这个属性，如果查找到最后一只没有找到，就返回一个 undefined

164.如何避免重绘或者重排？

1. 分离读写操作

```
var curLeft=div.offsetLeft;
var curTop=div.offsetTop;
div.style.left=curLeft+1+'px';
div.style.top=curTop+1+'px';
```

2. 样式集中改变

可以添加一个类，样式都在类中改变

3. 可以使用 absolute 脱离文档流。
4. 使用 display:none ，不使用 visibility，也不要改变 它的 z-index
5. 能用 css3 实现的就用 css3 实现。

165.说下函数式编程的理解

1.什么是函数式编程?

函数式编程是种编程方式，它将电脑运算视为函数的计算。函数编程语言最重要的基础是 λ 演算（lambda calculus），而且 λ 演算的函数可以接受函数当作输入（参数）和输出（返回值）。

2.优势特点

代码简洁、开发快速、命令式实现、函数式实现、易于理解，抽象度高、没有副作用，变量无状态

166.forEach，map 和 filter 的区别（哔哩哔哩）

- filter 函数，顾名思义，它是一个用来过滤的函数。他可以通过指定的过滤条件，删选出数组中符合条件的元素，并返回。
- map 函数，这个函数与 filter 函数不同之处在于，filter()把传入的函数依次作用于每个元素，然后根据返回值是 true 还是 false 决定保留还是丢弃该元素。而 map 则会返回传入函数 return 的值。
- forEach 函数，可以实现对数组的遍历，和 map 函数与 filter 函数不同的是它没有返回值。

167.delete 数组的 item，数组的 length 是否会 -1

答案：不会

delete Array[index]

```
const arr = ['a', 'b', 'c', 'd', 'e'];
let result = delete arr[1];
console.log(result); // true;
console.log(arr); // ['a', undefined, 'c', 'd', 'e']
console.log(arr.length); // 5
console.log(arr[1]); // undefined
```

使用 delete 删除元素，返回 true 和 false,true 表示删除成功，false 表示删除失败。使用 delete 删除数组元素并不会改变原数组的长度，只是把被删除元素的值变为 undefined。

168.给出 ['1', '3', '10'].map(parseInt) 执行结果

答案：[1, NaN, 2]

169.执行上下文

执行上下文可以简单理解为一个对象:

它包含三个部分:

- 变量对象(VO)
- 作用域链(词法作用域)
- this 指向

它的类型:

- 全局执行上下文
- 函数执行上下文
- eval 执行上下文

代码执行过程:

- 创建 全局上下文 (global EC)
- 全局执行上下文 (caller) 逐行 自上而下 执行。遇到函数时，函数执行上下文 (callee) 被 push 到执行栈顶层
- 函数执行上下文被激活，成为 active EC, 开始执行函数中的代码，caller 被挂起
- 函数执行完后，callee 被 pop 移除出执行栈，控制权交还全局上下文 (caller)，继续执行

170.怎样理解 setTimeout 执行误差

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

171.数组降维

1.数组字符串化

```
let arr = [[222, 333, 444], [55, 66, 77], {a: 1} ]
     arr += '';
     arr = arr.split(',');

console.log(arr); // ["222", "333", "444", "55", "66", "77", "[object Object]"]
```

这也是比较简单的一种方式，从以上例子中也能看到问题，所有的元素会转换为字符串，且元素为对象类型会被转换为 "[object Object]" ，对于同一种类型数字或字符串还是可以的。

2.利用 apply 和 concat 转换

```
function reduceDimension(arr) {
    return Array.prototype.concat.apply([], arr);
}

console.log(reduceDimension([[123], 4, [7, 8],[9, [111]]]));// [123, 4, 7, 8, 9, Array(1)]
```

3.递归

```
function reduceDimension(arr){
    let ret = [];
    let toArr = function(arr){
        arr.forEach(function(item){
            item instanceof Array ? toArr(item) : ret.push(item);
        });
    }
    toArr(arr);
    return ret;
}
```

4.Array.prototype.flat()

```
var arr1 = [1, 2, [3, 4]];
arr1.flat();
// [1, 2, 3, 4]

var arr2 = [1, 2, [3, 4, [5, 6]]];
arr2.flat();
// [1, 2, 3, 4, [5, 6]]

var arr3 = [1, 2, [3, 4, [5, 6]]];
arr3.flat(2);
// [1, 2, 3, 4, 5, 6]

//使用 Infinity 作为深度，展开任意深度的嵌套数组
arr3.flat(Infinity);
// [1, 2, 3, 4, 5, 6]
```

5.使用 reduce、concat 和递归无限反嵌套多层嵌套的数组

```
var arr1 = [1,2,3,[1,2,3,4, [2,3,4]]];

function flattenDeep(arr1) {
   return arr1.reduce((acc, val) => Array.isArray(val) ? acc.concat(flattenDeep(val)) : acc.concat(val), []);
}
flattenDeep(arr1);
// [1, 2, 3, 1, 2, 3, 4, 2, 3, 4]
```

172.为什么 for 循环嵌套顺序会影响性能？

答案：把循环次数大的放在内层，执行时间会比较短

```
var t1 = new Date().getTime()
for (let i = 0; i < 100; i++) {
  for (let j = 0; j < 1000; j++) {
    for (let k = 0; k < 10000; k++) {
    }
  }
}
var t2 = new Date().getTime()
console.log('first time', t2 - t1)
```

变量	实例化(次数)	初始化(次数)	比较(次数)	自增(次数)
i	1	1	10	10
j	10	10	10 * 100	10 * 100
k	10 * 100	10 * 100	10 * 100 * 1000	10 * 100 * 1000

```
for (let i = 0; i < 10000; i++) {
  for (let j = 0; j < 1000; j++) {
    for (let k = 0; k < 100; k++) {

    }
  }
}
var t3 = new Date().getTime()
console.log('two time', t3 - t2)
```

变量	实例化(次数)	初始化(次数)	比较(次数)	自增(次数)
i	1	1	1000	1000
j	1000	1000	1000 * 100	1000 * 100
k	1000 * 100	1000 * 100	1000 * 100 * 10	1000 * 100 * 10

173.轮播图实现原理

1.图片移动实现原理：
利用浮动将所有所有照片依次排成一行，给这一长串图片添加一个父级的遮罩，每次只显示一张图，其余的都隐藏起来。对图片添加绝对定位，通过控制 left 属性，实现照片的移动。
2.图片移动动画原理：
从 a 位置移动到 b 位置，需要先计算两点之间的差值，通过差值和时间间隔，计算出每次移动的步长，通过添加定时器，每次移动相同的步长，实现动画效果。
3.图片定位停止原理：
每一张照片都有相同的宽度，每张照片都有一个绝对的定位数值，通过检测定每次移动后，照片当前位置和需要到达位置之间的距离是否小于步长，如果小于，说明已经移动到位，可以将定时器清除，来停止动画。
4 图片切换原理：
在全局设置一个变量，记录当前图片的位置，每次切换或跳转时，只需要将数值修改，并调用图片页数转像素位置函数，再调用像素运动函数即可。
5.自动轮播原理：
设置定时器，一定时间间隔后，将照片标记加 1，然后开始切换。
6.左右点击切换原理：
修改当前位置标记，开始切换。这里需要注意与自动轮播之间的冲突。当点击事件触发之后，停止自动轮播计时器，开始切换。当动画结束后再次添加自动轮播计时器。
7.无缝衔接原理：
需要无缝衔接，难度在于最后一页向后翻到第一页，和第一页向前翻到最后一页。由于图片的基本移动原理。要想实现无缝衔接，两张图片就必须紧贴在一起。所以在第一张的前面需要添加最后一张，最后一张的后面需要添加第一张。
7.预防鬼畜原理：
始终保证轮播图的运动动画只有一个，从底层杜绝鬼畜。需要在每次动画开始之前，尝试停止动画定时器，然后开始为新的动画添加定时器。
8.预防暴力点击原理：
如果用户快速点击触发事件，会在短时间内多次调用切换函数，虽然动画函数可以保证，不会发生鬼畜，但在照片从最后一张到第一张的切换过程，不会按照正常的轮播，而是实现了跳转。所以需要通过添加口令的方式来，限制用户的点击。当用户点击完成后，口令销毁，动画结束后恢复口令。
9.小圆点的位置显示原理：
每次触发动画时，通过全局变量标记，获取当前页数，操作清除所有小圆点，然后指定一页添加样式。
10.点击触发跳转的原理：
类似于左右点击触发，只是这是将全局页面标记，直接修改，后执行动画。需要避免与自动轮播定时器的冲突。

174.如何设计一个轮播图组件

答案：

1. 轮播图功能实现
2. 抽出需要传入的变量，如：背景图，文案描述等

175.script 引入方式

- html 静态<script>引入
- js 动态插入<script>
- <script defer>: 延迟加载，元素解析完成后执行
- <script async>: 异步加载，但执行时会阻塞元素渲染

176.数组中的 forEach 和 map 的区别

177.for in和for of的区别
178.typeof 与 instanceof 区别
179.微任务和宏任务
答案：

```
/*
* 宏任务
*   分类： setTimeout setInterval requrestAnimationFrame
*   1. 宏任务所处的队列就是宏任务队列
*   2. 第一个宏任务队列中只有一个任务： 执行主线程的 js 代码
*   3. 宏任务队列可以有多个
*   4. 当宏任务队列的中的任务全部执行完以后会查看是否有微任务队列如果有先执行微任务队列中的所有任务，如果没有就查看是否有宏任务队列
*
* 微任务
*   分类： new Promise().then(回调) process.nextTick
*   1. 微任务所处的队列就是微任务队列
*   2. 只有一个微任务队列
*   3. 在上一个宏任务队列执行完毕后如果有微任务队列就会执行微任务队列中的所有任务
* */

console.log('------------------ start ------------------');

setTimeout(() => {
  console.log('setTimeout');
}, 0)

new Promise((resolve, reject) =>{
  for (var i = 0; i < 5; i++) {
    console.log(i);
  }
  resolve();  // 修改 promise 实例对象的状态为成功的状态
}).then(() => {
  console.log('promise 实例成功回调执行');
})

console.log('------------------ end ------------------');
```