JavaScript 基础面试题

1. 介绍 JavaScript 的基本数据类型?

- 基本数据类型: `Number`、`String`、`Boolean`、`Null`、`Undefined`
- `object` 是 `Javascript` 中所有对象的父对象。
- 数据封装类对象：`object`、`Array`、`Boolean`、`Number` 和 `String`。
- 其他对象：`Function`、`Arguments`、`Math`、`Date`、`Error` 和 `RegExp`。
- 其他数据类型：`Symbol`

2. 浅谈 JavaScript 中变量和函数声明的提升?

- 在 JavaScript 中变量和函数的声明会提升到最顶部执行
- 函数的提升高于变量的提升。
- 函数内部如果用 var 声明了相同名称的外部变量，函数将不再向上寻找。
- 匿名函数不会提升。
- 不同<script>块中的函数互不影响

3. 什么是闭包，闭包有什么特性?

- 闭包就是能够读取其他函数内部变量的函数
- 闭包是指有权访问另一个函数作用域中变量的函数，创建闭包的最常见的方式就是在一个函数内创建另一个函数，通过另一个函数访问这个函数的局部变量,利用闭包可以突破作用链域
- 闭包的特性:
  - 函数内再嵌套函数
  - 内部函数可以引用外层的参数和变量
  - 参数和变量不会被垃圾回收机制回收

4. 说说对闭包的理解和闭包的作用

- 使用闭包主要是为了设计私有的方法和变量。闭包的优点是可以避免全局变量的污染，缺点是闭包会常驻内存，会增大内存使用量，使用不当很容易造成内存泄露。在 js 中，函数即闭包，只有函数才会产生作用域的概念
- 闭包 的最大用处有两个，一个是可以读取函数内部的变量，另一个就是让这些变量始终保持在内存中
- 闭包的另一个用处，是封装对象的私有属性和私有方法
- **好处**：能够实现封装和缓存等
- **坏处**：就是消耗内存、不正当使用会造成内存溢出的问题

使用闭包的注意点

- 由于闭包会使得函数中的变量都被保存在内存中，内存消耗很大，所以不能滥用闭包，否
- 则会造成网页的性能问题，在 IE 中可能导致内存泄露
- 解决方法是，在退出函数之前，将不使用的局部变量全部删除

5. 说说 This 对象的理解

- `this` 总是指向函数的直接调用者（而非间接调用者）
- 如果有 `new` 关键字，`this` 指向 `new` 出来的那个对象
- 在事件中，`this` 指向触发这个事件的对象，特殊的是，`IE` 中的 `attachEvent` 中的 `this` 总是指向全局对象 `Window`

6. 事件模型的理解

- `W3C` 中定义事件的发生经历三个阶段
  - 捕获阶段（`capturing`）
  - 目标阶段（`targetin`）
  - 冒泡阶段（`bubbling`）
- 冒泡型事件：当你使用事件冒泡时，子级元素先触发，父级元素后触发
- 捕获型事件：当你使用事件捕获时，父级元素先触发，子级元素后触发
- `DOM` 事件流：同时支持两种事件模型：捕获型事件和冒泡型事件
- 阻止冒泡：在 `W3c` 中，使用 `stopPropagation()`方法；在 IE 下设置 `cancelBubble = true`
- 阻止捕获：阻止事件的默认行为，例如 `click - <a>`后的跳转。在 `W3c` 中，使用 `preventDefault()` 方法，在 `IE` 下设置 `window.event.returnValue = false`

7. new 操作符具体干了什么?

- 创建一个空对象，并且 `this` 变量引用该对象，同时还继承了该函数的原型
- 属性和方法被加入到 `this` 引用的对象中
- 新创建的对象由 `this` 所引用，并且最后隐式的返回 `this`

8. 说说栈和堆的理解，以及它们的区别?

- 栈内存: 栈内存首先是一片内存区域，存储的都是局部变量，凡是定义在方法中的都是局部变量（方法外的是全局变量），for 循环内部定义的也是局部变量，是先加载函数才能进行局部变量的定义，所以方法先进栈，然后再定义变量，变量有自己的作用域，一旦离开作用域，变量就会被释放。栈内存的更新速度很快，因为局部变量的生命周期都很短
- 堆内存: 存储的是数组和对象（其实数组就是对象），凡是 new 建立的都是在堆中，堆中存放的都是实体（对象），实体用于封装数据，而且是封装多个（实体的多个属性），如果一

个数据消失，这个实体也没有消失，还可以用，所以堆是不会随时释放的，但是栈不一样，栈里存放的都是单个变量，变量被释放了，那就没有了。堆里的实体虽然不会被释放，但是会被当成垃圾，Java 有垃圾回收机制不定时的收取

- 栈和堆的区别:
  - 栈内存存储的是局部变量而堆内存存储的是实体
  - 栈内存的更新速度要快于堆内存，因为局部变量的生命周期很短
  - 栈内存存放的变量生命周期一旦结束就会被释放，而堆内存存放的实体会被垃圾回收机制不定时的回收

9. JS 数组和对象的遍历方式，以及几种方式的比较

- `for in` 循环
- `for` 循环
- `forEach` 循环
  - 这里的 `forEach` 回调中两个参数分别为 `value` ， `index`
  - `forEach` 无法遍历对象
  - IE 不支持该方法；`Firefox` 和 `chrome` 支持
  - `forEach` 无法使用 `break` ， `continue` 跳出循环，且使用 `return` 是跳过本次循环
- `for-in` 需要分析出 `array` 的每个属性，这个操作性能开销很大。用在 `key` 已知的数组上是常不划算。所以尽量不要用 `for-in`，除非你不清楚要处理哪些属性，例如 `JSON` 对象这样情况
- `for` 循环每进行一次，就要检查一下数组长度。读取属性（数组长度）要比读局部变量慢，尤其是当 `array` 里存放的都是 `DOM` 元素，因为每次读取都会扫描一遍页面上的选择器相关元素，速度会大大降低

10. map 与 forEach 的区别

- `forEach` 方法，是最基本的方法，就是遍历与循环，默认有 3 个传参：分别是遍历的数组内容 `item`、数组索引 `index`、和当前遍历数组 `Array`
- `map` 方法，基本用法与 `forEach` 一致，但是不同的，它会返回一个新的数组，所以 callback 需要有 `return` 值，如果没有，会返回 `undefined`

11. 谈一谈箭头函数与普通函数的区别?

- 函数体内的 `this` 对象，就是定义时所在的对象，而不是使用时所在的对象
- 不可以当作构造函数，也就是说，不可以使用 `new` 命令，否则会抛出一个错误
- 不可以使用 `arguments` 对象，该对象在函数体内不存在。如果要用，可以用 `Rest` 参数代替
- 不可以使用 `yield` 命令，因此箭头函数不能用作 `Generator` 函数

12. JavaScript 定义类的 4 种方法

- 工厂方法:

```
function creatPerson(name, age) {
    var obj = new Object();
    obj.name = name;
    obj.age = age;
    obj.sayName = function () {
        window.alert(this.name);
    };
    return obj;
}
```

- 构造函数方法:

```
function Person(name, age) {
    this.name = name;
    this.age = age;
    this.sayName = function () {
        window.alert(this.name);
    };
}
```

- 原型方法:

```
function Person() {}
Person.prototype = {
    constructor: Person,
    name: "Ning",
    age: "23",
    sayName: function () {
        window.alert(this.name);
    }
};
```

- 组合使用构造函数和原型方法:

```
function Person(name, age) {
    this.name = name;
    this.age = age;
}
Person.prototype = {
    constructor: Person,
    sayName: function () {
        window.alert(this.name);
    }
};
```

13. JavaScript 实现继承的 3 种方法

- 借用构造函数法:

```
function SuperType(name) {
    this.name = name;
    this.sayName = function () {
        window.alert(this.name);
    };
}
function SubType(name, age) {
    SuperType.call(this, name); //在这里借用了父类的构造函数
    this.age = age;
}
```

- 对象冒充:

```
function SuperType(name) {
    this.name = name;
    this.sayName = function () {
        window.alert(this.name);
    };
}
function SubType(name, age) {
    this.supertype = SuperType; //在这里使用了对象冒充
    this.supertype(name);
    this.age = age;
}
```

- 组合继承:

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

SuperType.prototype = {
    sayName: function () {
        window.alert(this.name);
    }
};

function SubType(name, age) {
    SuperType.call(this, name); //在这里继承属性
    this.age = age;
}
SubType.prototype = new SuperType(); //这里继承方法
```

14. 对原生 Javascript 了解程度

- 数据类型、运算、对象、Function、继承、闭包、作用域、原型链、事件、RegExp、JSON 、Ajax、DOM、BOM、内存泄漏、跨域、异步装载、模板引擎、前端 MVC、路由、模块化、Canvas、ECMAScript。

15. 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

16. 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"

17. 说说你对作用域链的理解

- 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
- 作用域就是变量与函数的可访问范围，即作用域控制着变量与函数的可见性和生命周期

18. JavaScript 原型，原型链 ？有什么特点？

- 每个对象都在其内部初始化一个属性，就是 prototype(原型)，当我们访问一个对象的属性时如果这个对象内部不存在这个属性，那么他就会去 prototype 里找这个属性，这个 prototype 又会有自己的 prototype ，于是就这样一直找下去，也就是我们平时所说的原型链的概念
- 关系： `instance.constructor.prototype = instance.__proto__`
- JavaScript 对象是通过引用来传递的，我们创建的每个新对象实体中并没有一份属于自己的原型副本。当我们修改原型时，与之相关的对象也会继承这一改变
- 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
- 就会查找他的 Prototype 对象是否有这个属性，如此递推下去，一直检索到 Object 内建对象

19. 说说什么是事件代理?

- 事件代理（Event Delegation），又称之为事件委托。是 JavaScript 中常用绑定事件的常技巧。顾名思义，“事件代理”即是把原本需要绑定的事件委托给父元素，让父元素担当事件监听的职务。事件代理的原理是 DOM 元素的事件冒泡。使用事件代理的好处是可以提高性能。
- 可以大量节省内存占用，减少事件注册，比如在 table 上代理所有 td 的 click 事件就非常棒。
- 可以实现当新增子对象时无需再次对其绑定。

20. 说说 jax 原理?

- Ajax 的原理简单来说是在用户和服务器之间加了—个中间层(AJAX 引擎)，由 XmlHttpRequest 对象来向服务器发异步请求，从服务器获得数据，然后用 javascrip t 来操作 DOM 而更新页面。使用户操作与服务器响应异步化。这其中最关键的一步就是从服务器获得请求数据。
- Ajax 的过程只涉及 JavaScript 、XMLHttpRequest 和 DOM 。XMLHttpRequest 是 ajax 的核心机制。
- Ajax 的优点:
  - 通过异步模式，提升了用户体验
  - 优化了浏览器和服务器之间的传输，减少不必要的数据往返，减少了带宽占用
  - Ajax 在客户端运行，承担了一部分本来由服务器承担的工作，减少了大用户量下的服务器负载
  - Ajax 可以实现动态不刷新（局部刷新）
- Ajax 的缺点:
  - 安全问题 AJAX 暴露了与服务器交互的细节
  - 对搜索引擎的支持比较弱
  - 不容易调试
- Ajax 的请求:

```
/** 创建连接 **/
var xhr = null;
xhr = new XMLHttpRequest()
/** 2. 连接服务器 **/
xhr.open('get', url, true)
/** 3. 发送请求 **/
xhr.send(null);
/** 4. 接受请求 **/
xhr.onreadystatechange = function () {
    if (xhr.readyState == 4) {
        if (xhr.status == 200) {
            success(xhr.responseText);
        } else {
```

```
            /** false **/
            fail && fail(xhr.status);
        }
    }
}

```

21. 说说如何解决跨域问题?

- 首先了解下浏览器的同源策略 同源策略 / SOP（Same origin policy） 是一种约定，由 Netscape 公司 1995 年引入浏览器，它是浏览器最核心也最基本的安全功能，如果缺少了同源策略，浏览器很容易受到 XSS、 CSFR 等攻击。所谓同源是指"协议+域名+端口"三者相同，即便两个不同的域名指向同一个 ip 地址，也非同源。

- **通过** jsonp **跨域**
- document.domain + iframe **跨域**
- nginx **代理跨域**
- nodejs **中间件代理跨域**
- **后端在头部信息里面设置安全域名解决跨域**

22. 异步加载 JS 的方式有哪些?

- defer，只支持 IE
- async ：
- 创建 script，插入到 DOM 中，加载完毕后 callBack

```
/** 异步加载地图 **/
export default function MapLoader () {
    return new Promise((resolve, reject) => {
      if (window.AMap) {
        resolve(window.AMap)
      } else {
        var script = document.createElement('script')
        script.type = 'text/javascript'
        script.async = true
        script.src = ''
        script.onerror = reject
        document.head.appendChild(script)
      }
      window.initAMap  = () => {
        resolve(window.AMap)
      }
    })
}

```

23. 那些操作会造成内存泄漏?

- 内存泄漏指任何对象在您不再拥有或需要它之后仍然存在
- setTimeout 的第一个参数使用字符串而非函数的话，会引发内存泄漏
- 闭包使用不当

24. 介绍 JS 有哪些内置对象?

- Object 是 JavaScript 中所有对象的父对象
- 数据封装类对象 Object 、Array 、Boolean 、Number 和 String
  其他对象：Function 、Arguments 、Math 、Date 、RegExp 、Error

25. 说几条写 JavaScript 的基本规范

- 不要在同一行声明多个变量
- 请使用 ===/!== 来比较 true/false 或者数值
- 使用对象字面量替代 new Array 这种形式
- 不要使用全局函数
- Switch 语句必须带有 default 分支
- If 语句必须使用大括号
- for-in 循环中的变量 应该使用 var 关键字明确限定作用域，从而避免作用域污

26. 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）
- 由 JSON 字符串转换为 JSON 对象的时候可以用 eval，var obj =eval('('+ str +')')

27. null 和 undefined 的区别

- undefined 表示不存在这个值。
- undefined 是一个表示"无"的原始值或者说表示"缺少值"，就是此处应该有一个值，但是还没有定义。当尝试读取时会返回 undefined
- 例如变量被声明了，但没有赋值时，就等于 undefined
- null 表示一个对象被定义了，值为“空值”
- null 是一个对象(空对象，没有任何属性和方法)
- 例如作为函数的参数，表示该函数的参数不是对象；
- 在验证 null 时，一定要使用===，因为==无法分别 null 和 undefined

28. 说说同步和异步的区别?

- 同步：浏览器访问服务器请求，用户看得到页面刷新，重新发请求,等请求完，页面刷新，新内容出现，用户看到新内容,进行下一步操作
- 异步：浏览器访问服务器请求，用户正常操作，浏览器后端进行请求。等请求完，页面不刷新，新内容也会出现，用户看到新内容

29. defer 和 async ?

- `defer` 并行加载 `js` 文件，会按照页面上 `script` 标签的顺序执行
- `async` 并行加载 `js` 文件，下载完成立即执行，不会按照页面上 `script` 标签的顺序执行

30. ["1", "2", "3"].map(parseInt) 答案是多少?

- `[1, NaN, NaN]`因为 `parseInt` 需要两个参数`(val, radix)`，其中 `radix` 表示解时用的基数。
- `map` 传了 `3` 个`(element, index, array)`，对应的 `radix` 不合法导致解析失败。

31. use strict 的理解和作用?

- `use strict` 是一种 `ECMAscript 5` 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,使 `JS` 编码更加规范化的模式,消除 `Javascript` 语法的一些不合理、不严谨之处，减少一些怪异行为

32. 说说严格模式的限制?

- 变量必须声明后再使用
- 函数的参数不能有同名属性，否则报错
- 不能使用 `with` 语句
- 禁止 `this` 指向全局对象

33. 说说严格模式的限制?

- 新增模板字符串（为 `JavaScript` 提供了简单的字符串插值功能）
- 箭头函数
- `for-of`（用来遍历数据—例如数组中的值。）
- `arguments` 对象可被不定参数和默认参数完美代替。
- `ES6` 将 `Promise` 对象纳入规范，提供了原生的 `Promise` 对象。
- 增加了 `let` 和 `const` 命令，用来声明变量。
- 增加了块级作用域。
- `let` 命令实际上就增加了块级作用域。
- 还有就是引入 `module` 模块的概念

34. 说说 JSON 的了解?

- 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
- JSON 字符串转换为 JSON 对象:

```
var obj =eval('('+ str +')');
var obj = str.parseJSON();
var obj = JSON.parse(str);
```

- JSON 对象转换为 JSON 字符串:

```
var last=obj.toJSONString();
var last=JSON.stringify(obj);
```

35. 说说 JS 延迟加载的方式有哪些?

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

36. 说说 attribute 和 property 的区别是什么?

- attribute 是 dom 元素在文档中作为 html 标签拥有的属性;
- property 就是 dom 元素在 js 中作为对象拥有的属性。
- 对于 html 的标准属性来说，attribute 和 property 是同步的，是会自动更新的
- 但是对于自定义的属性来说，他们是不同步的

37. 说说 let 的区别是什么?

- let 命令不存在变量提升，如果在 let 前使用，会导致报错
- 如果块区中存在 let 和 const 命令，就会形成封闭作用域
- 不允许重复声明，因此，不能在函数内部重新声明参数

38. 如何通过 JS 判断一个数组?

- instanceof 方法
  - instanceof 运算符是用来测试一个对象是否在其原型链原型构造函数的属性

```
var arr = [];  js
arr instanceof Array; // true
```

- constructor 方法

- constructor 属性返回对创建此对象的数组函数的引用，就是返回对象相对应的构造函数

```
var arr = []; js
arr.constructor == Array; //true
```

- ES5 新增方法 isArray()

```
var a = new Array(123); js
var b = new Date();
console.log(Array.isArray(a)); //true
console.log(Array.isArray(b)); //false
```

38. 说说 let、var、const 的理解

- let：
  - 允许你声明一个作用域被限制在块级中的变量、语句或者表达式
  - let 绑定不受变量提升的约束，这意味着 let 声明不会被提升到当前
  - 该变量处于从块开始到初始化处理的“暂存死区”
- var：
  - 声明变量的作用域限制在其声明位置的上下文中，而非声明变量总是全局的
  - 由于变量声明（以及其他声明）总是在任意代码执行之前处理的，所以在代码中的任意位置声明变量总是等效于在代码开头声明
- const：
  - 声明创建一个值的只读引用（即指针）
  - 基本数据当值发生改变时，那么其对应的指针也将发生改变，故造成 const 申明基本数据类型时
  - 再将其值改变时，将会造成报错， 例如 const a = 3 ； a = 5 时 将会报错
  - 但是如果是复合类型时，如果只改变复合类型的其中某个 Value 项时， 将还是正常使用

39. 正则表达式的使用

- 当使用 RegExp() 构造函数的时候，不仅需要转义引号（即 \ “表示”），并且还需要双反斜杠（即 \\ 表示一个 \ ）。使用正则表达字面量的效率更高

40. Javascript 中 callee 和 caller 的作用

- caller 是返回一个对函数的引用，该函数调用了当前函数;
- callee 是返回正在被执行的 function 函数，也就是所指定的 function 对象的正文

41. 说说 window.onload 和$(document).ready 的区别?

- window.onload()方法是必须等到页面内包括图片的所有元素加载完毕后才能执行
- $(document).ready() 是 DOM 结构绘制完毕后就执行，不必等到加载完毕
- 浏览器的兼容性

```
function ready(fn) {
    if (document.addEventListener) { //标准浏览器
        document.addEventListener('DOMContentLoaded', function () {
            //注销事件，避免反复触发
            document.removeEventListener('DOMContentLoaded', arguments.callee, fa
            fn(); //执行函数
        }, false);
    } else if (document.attachEvent) { //IE
        document.attachEvent('onreadystatechange', function () {
            if (document.readyState == 'complete') {
                document.detachEvent('onreadystatechange', arguments.callee);
                fn(); //函数执行
            }
        });
    }
};
```

42. Javascript 数组去重方法汇总

- 利用 for 嵌套 for，然后 splice 去重（ES5 中最常用）

```
function unique(arr) {
    for (var i = 0; i < arr.length; i++) {
        for (var j = i + 1; j < arr.length; j++) {
            if (arr[i] == arr[j]) { //第一个等同于第二个，splice 方法删除第二个
                arr.splice(j, 1);
                j--;
            }
        }
    }
    return arr;
}
var arr = [1, 1, 'true', 'true', true, true, 15, false, undefined, undefined, null]
console.log(unique(arr))
```

- 利用 ES6 Set 去重（ES6 中最常用）

```
function unique(arr) {
    return Array.from(new Set(arr))
}
var arr = [1, 1, 'true', 'true', true, true, 15, undefined, undefined, null]
console.log(unique(arr))
```

- 利用 indexOf 去重

```
function unique(arr) {
    if (!Array.isArray(arr)) {
        console.log('type error!')
        return
    }
    var array = [];
    for (var i = 0; i < arr.length; i++) {
        if (array.indexOf(arr[i]) === -1) {
            array.push(arr[i])
        }
    }
    return array;
}
var arr = [1, 1, 'true', 'true', true, true, 15, undefined, undefined, null]
console.log(unique(arr))
```

- 利用 sort()去重

```
function unique(arr) {
    if (!Array.isArray(arr)) {
        console.log('type error!')
        return;
    }
    arr = arr.sort()
    var arrry = [arr[0]];
    for (var i = 1; i < arr.length; i++) {
        if (arr[i] !== arr[i - 1]) {
            arrry.push(arr[i]);
        }
    }
    return arrry;
}
var arr = [1, 1, 'true', 'true', true, true, 15, undefined, undefined, null]
console.log(unique(arr))
```

- 利用对象的属性不能相同的特点进行去重

```
function unique(arr) {
    if (!Array.isArray(arr)) {
        console.log('type error!')
        return
    }
    var arrry = [];
    var obj = {};
    for (var i = 0; i < arr.length; i++) {
        if (!obj[arr[i]]) {
            arrry.push(arr[i])
            obj[arr[i]] = 1
        } else {
            obj[arr[i]]++
        }
    }
    return arrry;
}
var arr = [1, 1, 'true', 'true', true, true, 15, undefined, undefined, null]
console.log(unique(arr))
```

- 利用 includes 去重

```
function unique(arr) {
    if (!Array.isArray(arr)) {
        console.log('type error!')
        return
    }
    var array = [];
    for (var i = 0; i < arr.length; i++) {
        if (!array.includes(arr[i])) {//includes 检测数组是否有某个值
            array.push(arr[i]);
        }
    }
    return array
}
var arr = [1, 1, 'true', 'true', true, true, 15, undefined, undefined, null]
console.log(unique(arr))
```

43. 浏览器缓存

- 览器缓存分为强缓存和协商缓存，当客户端请求某个资源时，获取缓存的流程如下
  - 先根据这个资源的一些 `http header` 判断它是否命中强缓存，如果命中，则直接从本地获取缓存资源，不会发请求到服务器。

- 强缓存没有命中时，客户端会发送请求到服务器，服务器通过另一些 request header 验证这个资源是否命中协商缓存，称为 http 再验证，如果命中，服务器将请求返回，但不返回资源，而是告诉客户端直接从缓存中获取，客户端收到返回后就会从缓存中获取源。
- 强缓存和协商缓存共同之处在于，如果命中缓存，服务器都不会返回资源； 区别是，强缓存不对发送请求到服务器，但协商缓存会。
- 当协商缓存也没命中时，服务器就会将资源发送回客户端。
- 当 ctrl+f5 强制刷新网页时，直接从服务器加载，跳过强缓存和协商缓存。
- 当 f5 刷新网页时，跳过强缓存，但是会检查协商缓存。

- **强缓存**
  - Expires 该字段是 http1.0 时的规范，值为一个绝对时间的 GMT 格式的时间字串，代表缓存资源的过期时间
  - Cache-Control:max-age 该字段是 http1.1 的规范，强缓存利用其 max-age 值来判缓存资源的最大生命周期，它的值单位为秒

- **协商缓存**
  - Last-Modified 值为资源最后更新时间，随服务器 response 返回
  - If-Modified-Since 通过比较两个时间来判断资源在两次请求期间是否有过修改，如果没有修改，则命中协商缓存
  - ETag 表示资源内容的唯一标识，随服务器 response 返回
  - If-None-Match 服务器通过比较请求头部的 If-None-Match 与当前资源的 ETag 是否一致来判断资源是否在两次请求之间有过修改，如果没有修改，则命中协商缓存

44. 防抖/节流的理解

- 防抖：在滚动事件中需要做个复杂计算或者实现一个按钮的防二次点击操作。可以通过函数防抖动来实现
- 开始一个定时器，只要我定时器还在，不管你怎么点击都不会执行回调函数。一旦定时器结束并设置为 null，就可以再次点击了
- 对于延时执行函数来说的实现：每次调用防抖动函数都会判断本次调用和之前的时间间隔，如果小于需要的时间间隔，就会重新创建一个定时器，并且定时器的延时为设定时间减去之前的时间间隔。一旦时间到了，就会执行相应的回调函数
- 整体函数实现如下：

```
// *使用 underscore 的源码来解释防抖动
// *underscore 防抖函数，返回函数连续调用时，空闲时间必须大于或等于 wait，func 才会执行
// * @param { function} func 回调函数
// * @param { number } wait 表示时间窗口的间隔
// * @param { boolean } immediate 设置为 ture 时，是否立即调用函数
// * @return { function} 返回客户调用函数
```

```
_.debounce = function (func, wait, immediate) {
    var timeout, args, context, timestamp, result;
    var later = function () {
        // 现在和上一次时间戳比较
        var last = _.now() - timestamp;
        // 如果当前间隔时间少于设定时间且大于 0 就重新设置定时器
        if (last < wait && last >= 0) {
            timeout = setTimeout(later, wait - last);
        } else {
            // 否则的话就是时间到了执行回调函数
            timeout = null;
            if (!immediate) {
                result = func.apply(context, args);
                if (!timeout) context = args = null;
            }
        }
    };
    return function () {
        context = this;
        args = arguments;
        // 获得时间戳
        timestamp = _.now();
        // 如果定时器不存在且立即执行函数
        var callNow = immediate && !timeout;
        // 如果定时器不存在就创建一个
        if (!timeout) timeout = setTimeout(later, wait);
        if (callNow) {
            // 如果需要立即执行函数的话 通过 apply 执行
            result = func.apply(context, args);
            context = args = null;
        }
        return result;
    };
};
```

- 节流: 防抖动和节流本质是不一样的。防抖动是将多次执行变为最后一次执行，节流是将多次执行变成每隔一段时间执行

```
/*** underscore 节流函数，返回函数连续调用时，func 执行频率限定为 次 / wait
* @param {function} func 回调函数
* @param {number} wait 表示时间窗口的间隔
* @param {object} options 如果想忽略开始函数的的调用，传入{leading: false}。
* 如果想忽略结尾函数的调用，传入{trailing: false}
* 两者不能共存，否则函数不能执行
* @return {function} 返回客户调用函数*/
```

```
_.throttle = function (func, wait, options) {
    var context, args, result; var timeout = null;
    var previous = 0;
    // 如果 options 没传则设为空对象 if (!options) options = {}; // 定时器回调函数
    var later = function () {
        // 如果设置了 leading，就将 previous 设为 0 // 用于下面函数的第一个 if 判断
        previous = options.leading === false ? 0 : _.now();
        result = func.apply(context, args);
        if (!timeout) context = args = null;
    };
    return function () {
        var now = _.now();
        // 首次进入前者肯定为 true
        // 如果需要第一次不执行函数
        // 就将上次时间戳设为当前的
        // 这样在接下来计算 remaining 的值时会大于 0
        if (!previous && options.leading === false) previous = now;
        // 计算剩余时间
        var remaining = wait - (now - previous);
        context = this;
        args = arguments;
        // 如果当前调用已经大于上次调用时间 + wait
        // 如果设置了 trailing，只会进入这个条件
        // 如果没有设置 leading，那么第一次会进入这个条件
        // 还有一点，你可能会觉得开启了定时器那么应该不会进入这个 if 条件了
        // 其实还是会进入的，因为定时器的延时
        // 并不是准确的时间，很可能你设置了 2 秒
        // 但是他需要 2.2 秒才触发，这时候就会进入这个条件
        if (remaining <= 0 || remaining > wait) {
            // 如果存在定时器就清理掉否则会调用二次回调
            if (timeout) {
                clearTimeout(timeout);
                timeout = null;
            }
            previous = now;
            result = func.apply(context, args);
            if (!timeout) context = args = null;
        } else if (!timeout && options.trailing !== false) {
            // 判断是否设置了定时器和 trailing
            // 没有的话就开启一个定时器
            // 并且不能不能同时设置 leading 和 trailing
            timeout = setTimeout(later, remaining);
        }
        return result;
    };
};
```

45. JavaScript 变量提升

- 节当执行 JS 代码时，会生成执行环境，只要代码不是写在函数中的，就是在全局执行环境中，函数中的代码会产生函数执行环境，只此两种执行环境。
- 因为函数和变量提升的原因。通常提升的解释是说将声明的代码移动到了顶部，这其实没有什么错误，便于大家理解。但是更准确的解释应该是：在生成执行环境时，会有两个阶段。第一个阶段是创建的阶段，JS 解释器会找出需要提升的变量和函数，并且给他们提前在内存中开辟好空间，函数的话会将整个函数存入内存中，变量只声明并且赋值为 `undefined` ，所以在第二个阶段，也就是代码执行阶段，我们可以直接提前使用。

46. 实现 Storage，使得该对象为单例，以及使用方式

```
var instance = null;
class Storage {
    static getInstance() {
        if (!instance) {
            instance = new Storage();
        }
        return instance;
    }
    setItem = (key, value) => localStorage.setItem(key, value),
    getItem = key => localStorage.getItem(key)
}
```

47. 说说你对事件流的理解

- 事件流分为两种，捕获事件流和冒泡事件流
- 捕获事件流从根节点开始执行，一直往子节点查找执行，直到查找执行到目标节点
- 冒泡事件流从目标节点开始执行，一直往父节点冒泡查找执行，直到查到到根节点

48. 说说从输入 URL 到看到页面发生的全过程

- 首先浏览器主进程接管，开了一个下载线程。
- 然后进行 HTTP 请求（DNS 查询、IP 寻址等等），中间会有三次捂手，等待响应，开始下载响应报文。
- 将下载完的内容转交给 Renderer 进程管理。
- Renderer 进程开始解析 css rule tree 和 dom tree，这两个过程是并行的，所以一般我会把 link 标签放在页面顶部。
- 解析绘制过程中，当浏览器遇到 link 标签或者 script、img 等标签，浏览器会去下载这些内容，遇到时候缓存的使用缓存，不适用缓存的重新下载资源。
- css rule tree 和 dom tree 生成完了之后，开始合成 render tree，这个时候浏览器会进行 layout，开始计算每一个节点的位置，然后进行绘制。

- 绘制结束后，关闭 TCP 连接，过程有四次挥手

49. 做一个 Dialog 组件，说说你设计的思路?它应该有什么功能?

- 该组件需要提供 `hook` 指定渲染位置，默认渲染在 body 下面。
- 然后改组件可以指定外层样式，如宽度等
- 组件外层还需要一层 `mask` 来遮住底层内容，点击 `mask` 可以执行传进来的 `onCancel` 函数关闭 `Dialog` 。
- 另外组件是可控的，需要外层传入 `visible` 表示是否可见。
- 然后 `Dialog` 可能需要自定义头 head 和底部 `footer`，默认有头部和底部，底部有一个确认按钮和取消按钮，确认按钮会执行外部传进来的 `onOk` 事件，然后取消按钮会执行外部传进来的 `onCancel` 事件。
- 当组件的 `visible` 为 `true` 时候，设置 `body` 的 `overflow` 为 `hidden` ，隐藏 `body` 的滚动条，反之显示滚动条。
- 组件高度可能大于页面高度，组件内部需要滚动条。
- 只有组件的 `visible` 有变化且为 `ture` 时候，才重渲染组件内的所有内容

50. 说说 ajax、fetch、axios 之间的区别

- `Ajax` 请求

```
$.ajax({
    type: 'POST',
    url: url,
    data: data,
    dataType: dataType,
    success: function () { },
    error: function () { }
});

```

  - 本身是针对 `MVC` 的编程,不符合现在前端 `MVVM` 的浪潮
  - 基于原生的 `XHR` 开发， `XHR` 本身的架构不清晰，已经有了 `fetch` 的替代方案
  - `JQuery` 整个项目太大，单纯使用 `ajax` 却要引入整个 `JQuery` 非常的不合理（采取个性化打包的方案又不能享受 CDN 服务）

- `fetch` 请求

  - `fetcht` 只对网络请求报错，对 `400` ， `500` 都当做成功的请求，需要封装去处理
  - `fetch` 默认不会带 `cookie` ，需要添加配置项
  - `fetch` 不支持 `abort` ，不支持超时控制，使用 `setTimeout` 及 `Promise.reject` 的实的超时控制并不能阻止请求过程继续在后台运行，造成了量的浪费
  - `fetch` 没有办法原生监测请求的进度，而 XHR 可以

```
try {
    js
    let response = await fetch(url);
    let data = response.json();
    console.log(data);
} catch (e) {
    console.log("Oops, error", e);
}
```

- axios 请求
  - 从浏览器中创建 XMLHttpRequest
  - 从 node.js 发出 http 请求
  - 支持 Promise API
  - 拦截请求和响应
  - 转换请求和响应数据
  - 取消请求
  - 自动转换 JSON 数据
  - 客户端支持防止 CSRF/XSRF

```
axios({
    method: 'post',
    url: '/user/12345',
    data: {
        firstName: 'Fred',
        lastName: 'Flintstone'
    }
}).then(function (response) {
    console.log(response);
}).catch(function (error) {
    console.log(error);
});
```

50. 说说内存泄漏

- 定义：程序中己动态分配的堆内存由于某种原因程序未释放或无法释放引发的各种问题。
- Javascript 中可能出现的内存泄漏情况：结果：变慢，崩溃，延迟大等
- Javascript 中可能出现的内存泄漏原因：
  - 全局变量
  - dom 清空时，还存在引用
  - ie 中使用闭包

- ✧ 定时器未清除
- ✧ 子元素存在引起的内存泄露

51. JavaScript 自定义事件

- document.createEvent()
- event.initEvent()
- element.dispatchEvent()

```
window.onload = function () {
    var demo = document.getElementById("demo");
    demo.addEvent("test", function () { console.log("handler1") });
    demo.addEvent("test", function () { console.log("handler2") });
    demo.onclick = function () {
        this.triggerEvent("test");
    }
}

Element.prototype.addEvent = function (en, fn) {
    this.pools = this.pools || {};
    if (en in this.pools) {
        this.pools[en].push(fn);
    } else {
        this.pools[en] = [];
        this.pools[en].push(fn);
    }
}

Element.prototype.triggerEvent = function (en) {
    if (en in this.pools) {
        var fns = this.pools[en];
        for (var i = 0, il = fns.length; i < il; i++) {
            fns[i]();
        }
    } else { return; }
}
```

52. JavaScript 数组排序的几种方式?

- 冒泡排序：每次比较相邻的两个数，如果后一个比前一个小，换位置

```
var arr = [3, 1, 4, 6, 5, 7, 2]; js
function bubbleSort(arr) {
    for (var i = 0; i < arr.length - 1; i++) {
        for (var j = 0; j < arr.length - i - 1; j++) {
```

```
            if (arr[j + 1] < arr[j]) {
                var temp;
                temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
    return arr;
}
console.log(bubbleSort(arr));
```

- 快速排序：采用二分法，取出中间数，数组每次和中间数比较，小的放到左边，大的放到右边

```
var arr = [3, 1, 4, 6, 5, 7, 2]; js
function quickSort(arr) {
    if (arr.length == 0) {
        return []; // 返回空数组
    }
    var cIndex = Math.floor(arr.length / 2);
    var c = arr.splice(cIndex, 1);
    var l = [];
    var r = [];
    for (var i = 0; i < arr.length; i++) {
        if (arr[i] < c) {
            l.push(arr[i]);
        } else {
            r.push(arr[i]);
        }
    }
    return quickSort(l).concat(c, quickSort(r));
}
console.log(quickSort(arr));
```

53. JavaScript 数组一行代码去重方法?

- Set 方法去重:

```
var arr = [1,2,3,4,2,1,23,543,3]
Array.prototype.uniq = function(){
    return [...new Set(this)];
}
console.log(arr.uniq())或者 console.log([...new Set(arr)])
```

54. JavaScript 如何判断一个对象是否为数组?

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

55. script 引入方式?

- html 静态 <script> 引入
- js 动态插入 <script>
- <script defer>：异步加载，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

56. 变量对象?

- 变量对象，是执行上下文中的一部分，可以抽象为一种 数据作用域，其实也可以理解为就是一个简单的对象，它存储着该执行上下文中的所有 变量和函数声明(不包含函数表达式)。
- 活动对象（ AO): 当变量对象所处的上下文为 active EC 时，称为活动对象。

57. babel 编译原理?

- babylon 将 ES6/ES7 代码解析成 AST
- babel-traverse 对 AST 进行遍历转译，得到新的 AST
- 新 AST 通过 babel-generator 转换成 ES5

58. 数组(array)?

- map：遍历数组，返回回调返回值组成的新数组
- forEach：无法 break ， 可以用 try/catch 中 throw new Error 来停止
- filter：过滤
- some：有一项返回 true ， 则整体为 true
- every：有一项返回 false ， 则整体为 false
- join：通过指定连接符生成字符串
- push / pop：末尾推入和弹出，改变原数组， 返回推入/弹出项
- unshift / shift：头部推入和弹出，改变原数组，返回操作项
- sort(fn) / reverse：排序与反转，改变原数组
- concat：连接数组，不影响原数组， 浅拷贝
- slice(start, end)：返回截断后的新数组，不改变原数组

- `splice(start,number,value...)`：返回删除元素组成的数组，`value` 为插入项，改变原数组
- `indexOf / lastIndexOf(value, fromIndex)`：查找数组项，返回对应的下标
- `reduce / reduceRight(fn(prev, cur)，defaultPrev)`：两两执行，`prev` 为上次化简函数的 `return` 值，`cur` 为当前值(从第二项开始)

59. 说几条写 JavaScript 的基本规范?

- 不要在同一行声明多个变量
- 请使用 `===/!==` 来比较 `true/false` 或者数值
- 使用对象字面量替代 `new Array` 这种形式
- 不要使用全局函数
- `Switch` 语句必须带有 `default` 分支
- `If` 语句必须使用大括号
- `for-in` 循环中的变量 应该使用 `var` 关键字明确限定作用域，从而避免作用域污

60. JavaScript 有几种类型的值?

- 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

61. JavaScript 有几种类型的值?

- `addEventListener()` 是符合 W3C 规范的标准方法;`attachEvent()` 是 IE 低版本的非标准方法
- `addEventListener()` 支持事件冒泡和事件捕获; - 而 `attachEvent()` 只支持事件冒泡
- `addEventListener()` 的第一个参数中,事件类型不需要添加 `on` ;`attachEvent()`需要添`'on'`
- 如果为同一个元素绑定多个事件，`addEventListener()`会按照事件绑定的顺序依次执行，`ttachEvent()` 会按照事件绑定的顺序倒序执行

62. JavaScript 深浅拷贝？

浅拷贝

- `Object.assign`
- 或者展开运算符

深拷贝

- 可以通过 `JSON.parse(JSON.stringify(object))` 来解决