




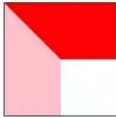








样式	width: 50px	width: 0	width: 50px & height: 50px
border-top: 50px solid red;			
border-top: 50px solid red; border-left: 50px solid pink;			
border-top: 50px solid red; border-left: 50px solid pink; border-right: 50px solid yellow;			
border-top: 50px solid red; border-left: 50px solid pink; border-right: 50px solid yellow; border-bottom: 50px solid orange;			

5、css3实现0.5px的细线？

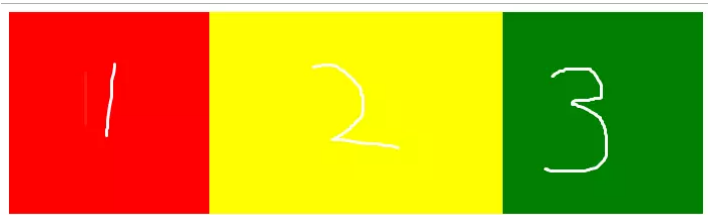
复制代码

```
/* css */
.line {
  position: relative;
}
.line:after {
  content: "";
  position: absolute;
  left: 0;
  top: 0;
  width: 100%;
  height: 1px;
  background-color: #000000;
  -webkit-transform: scaleY(.5);
  transform: scaleY(.5);
}

/* html */
<div class="line"></div>
```

6、css实现三栏布局

左右固定，中间自适应。



1. flex方式

复制代码

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Document</title>
  <style>
    .box {
      display: flex;
      justify-content: center;
      height: 200px;
    }
    .left {
      width: 200px;
      background-color: red;
      height: 100%;
    }
  </style>
</head>
```

```

        .content {
            background-color: yellow;
            flex: 1;
        }
        .right {
            width: 200px;
            background-color: green;
        }
    </style>
</head>
<body>
    <div class="box">
        <div class="left"></div>
        <div class="content"></div>
        <div class="right"></div>
    </div>
</body>
</html>

```

2. 绝对定位方式

复制代码

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
    <style>
        .box {
            position: relative;
            height: 200px;
        }
        .left {
            width: 200px;
            background-color: red;
            left: 0;
            height: 100%;
            position: absolute;
        }
        .content {
            background-color: yellow;
            left: 200px;
            right: 200px;
            height: 100%;
            position: absolute;
        }
        .right {
            width: 200px;
            background-color: green;
            right: 0;
            height: 100%;
            position: absolute;
        }
    </style>
</head>
<body>
    <div class="box">
        <div class="left"></div>
        <div class="content"></div>
        <div class="right"></div>
    </div>
</body>
</html>

```

3. 浮动方式

复制代码

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
    <style>
        .box {
            height: 200px;
        }
        .left {
            width: 200px;
            background-color: red;
            float: left;
            height: 100%;
        }
        .content {
            background-color: yellow;
            height: 100%;
        }
        .right {
            width: 200px;
            background-color: green;
            float: right;
            height: 100%;
        }
    </style>
</head>
<body>

```

```
<div class="box">
  <div class="left"></div>
  <div class="right"></div>
  <div class="content"></div>
</div>
</body>
</html>
```

7、让一个div垂直居中



1. 宽度和高度已知的

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Document</title>
  <style>
    .box {
      width: 400px;
      height: 200px;
      position: relative;
      background: red;
    }
    .content {
      width: 200px;
      height: 100px;
      position: absolute;
      top: 50%;
      left: 50%;
      margin-left: -100px;
      margin-top: -50px;
      background: green;
    }
  </style>
</head>
<body>
  <div class="box">
    <div class="content"></div>
  </div>
</body>
</html>
```

复制代码

2. 宽度和高度未知

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Document</title>
  <style>
    .box {
      width: 400px;
      height: 200px;
      position: relative;
      background: red;
    }
    .content {
      position: absolute;
      top: 50%;
      left: 50%;
      transform: translate(-50%, -50%);
      background: green;
    }
  </style>
</head>
<body>
  <div class="box">
    <div class="content"></div>
  </div>
</body>
</html>
```

复制代码

3. flex布局

```
<!DOCTYPE html>
<html lang="en">
<head>
```

复制代码

```
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<meta http-equiv="X-UA-Compatible" content="ie=edge">
<title>Document</title>
<style>
    .box {
        width: 400px;
        height: 200px;
        background: red;
        display: flex;
        justify-content: center;
        align-items: center;
    }
    .content {
        width: 200px;
        height: 100px;
        background: green;
    }
</style>
</head>
<body>
    <div class="box">
        <div class="content"></div>
    </div>
</body>
</html>
```

二、JS

1、闭包

闭包概念

能够读取其他函数内部变量的函数。
或简单理解为定义在一个函数内部的函数，内部函数持有外部函数内变量的引用。

复制代码

闭包用途

- 1、读取函数内部的变量
 - 2、让这些变量的值始终保持在内存中。不会再f1调用后被自动清除。
 - 3、方便调用上下文的局部变量。利于代码封装。
- 原因：f1是f2的父函数，f2被赋给了一个全局变量，f2始终存在内存中，f2的存在依赖f1，因此f1也始终存在内存中，不会在调用结束后，被垃圾回收机制回收。

复制代码

闭包缺点

- 1、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。
- 2、闭包会在父函数外部，改变父函数内部变量的值。所以，如果你把父函数当作对象（object）使用，把闭包当作它的公用方法（Public Method），把内部变量当作它的私有属性（private value），这时一定要小心，不要随便改变父函数内部变量的值。

复制代码

闭包应用场景

闭包应用场景之setTimeout

```
//setTimeout传递的第一个函数不能带参数
setTimeout((param) => {
    alert(param)
}, 1000);

//通过闭包可以实现传参效果
function func(param) {
    return function() {
        alert(param)
    }
}
var f1 = func('汪某');
setTimeout(f1, 1000)//汪某
```

javascript 复制代码

2、js中函数执行

在 ES5.1 里面函数是这样执行的（不讨论use strict和一些特殊情况，JS好复杂的），按如下顺序执行：

复制代码

1. 确定“this”的值（确切的来说，this在JS里面不是一个变量名而是一个关键字）
2. 创建一个新的作用域
3. 处理形参/实参（没有定义过才声明，无论如何都重新赋值，没有对应实参则赋值为“undefined”）：
对于每一个传入的实参，按照从左往右的顺序依次执行：如果对应的形参在本作用域中还没有定义，则在本作用域中声明形参，并赋值。如果已经定义过了，则重新给其赋值。（没有对应实参则赋值为“undefined”）（没有定义：就是“没有声明”的意思）
4. 处理函数定义（没有定义过才声明，无论如何都重新赋值）：
对该函数中所有的定义的函数，按照代码写的顺序依次执行：如果这个变量名在本作用域中还没有定义，则在本作用域中声明这个函数名，并且赋值为对应的函数，如果定义了这个变量，在可写的情况下重新给这个变量赋值为这个函数，否则抛出异常。
5. 处理 “arguments”（没有定义过才声明和赋值）：
如果在本作用域中没有定义 arguments，则在本作用域中声明arguments并给其赋值。
6. 处理变量声明（没有定义过才声明，不赋值）：
对于所有变量声明，按照代码写的顺序依次执行：如果在本作用域中没有定义这个变量，则在本作用域中声明这个变量，赋值为undefined
7. 然后执行函数代码。（当然是去变量定义里面的 var 执行）

3、new一个对象的过程中发生了什么嘛

复制代码

```
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. 将初始化完毕的新对象地址, 保存到等号左边的变量中
```

4、宏任务跟微任务

- macro-task(宏任务): 包括整体代码script, setTimeout, setInterval
- micro-task(微任务): Promise, process.nextTick

[这一次，彻底弄懂 JavaScript 执行机制](#)

5、防抖和节流

综合应用场景

- 防抖(debounce):就是指触发事件后在 n 秒内函数只能执行一次, 如果在 n 秒内又触发了事件, 则会重新计算函数执行时间。
 - search搜索联想, 用户在不断输入值时, 用防抖来节约请求资源。
 - window触发resize的时候, 不断的调整浏览器窗口大小会不断的触发这个事件, 用防抖来让其只触发一次
- 节流(throttle):就是指连续触发事件但是在 n 秒中只执行一次函数。节流会稀释函数的执行频率。
 - 鼠标不断点击触发, mousedown(单位时间内只触发一次)
 - 监听滚动事件, 比如是否滑到底部自动加载更多, 用throttle来判断 所谓防抖, 就是指触发事件后在 n 秒内函数只能执行一次, 如果在 n 秒内又触发了事件, 则会重新计算函数执行时间。

防抖函数分为非立即执行版和立即执行版。

- 非立即执行版的意思是触发事件后函数不会立即执行, 而是在 n 秒后执行, 如果在 n 秒内又触发了事件, 则会重新计算函数执行时间。
- 立即执行版的意思是触发事件后函数会立即执行, 然后 n 秒内不触发事件才能继续执行函数的效果。

javascript 复制代码

```
/**
 * @desc 函数防抖
 * @param func 函数
 * @param wait 延迟执行毫秒数
 * @param immediate true 表立即执行, false 表非立即执行
 */
function debounce(func,wait,immediate) {
    let timeout;

    return function () {
        let context = this;
        let args = arguments;

        if (timeout) clearTimeout(timeout);
        if (immediate) {
            var callNow = !timeout;
            timeout = setTimeout(() => {
                timeout = null;
            }, wait)
            if (callNow) func.apply(context, args)
        } else {
            timeout = setTimeout(function(){
                func.apply(context, args)
            }, wait);
        }
    }
}
```

所谓节流, 就是指连续触发事件但是在 n 秒中只执行一次函数。 节流会稀释函数的执行频率。

对于节流, 一般有两种方式可以实现, 分别是时间戳版和定时器版。

- 时间戳版的函数触发是在时间段内开始的时候
- 定时器版的函数触发是在时间段内结束的时候。

javascript 复制代码

```
/**
 * @desc 函数节流
 * @param func 函数
 * @param wait 延迟执行毫秒数
 * @param type 1 表时间戳版, 2 表定时器版
 */
function throttle(func, wait ,type) {
    if(type===1){
        if(previous = 0;
    }else if(type===2){
        var timeout;
    }
    return function() {
        let context = this;
        let args = arguments;
        if(type===1){
            let now = Date.now();
```

```
        if (now - previous > wait) {
            func.apply(context, args);
            previous = now;
        }
    }else if(type===2){
        if (!timeout) {
            timeout = setTimeout(() => {
                timeout = null;
                func.apply(context, args)
            }, wait)
        }
    }
}
```

6、数组的常用方法

改变原数组的方法

- splice() 添加/删除数组元素

语法: `arrayObject.splice(index,howmany,item1,.....,itemX)`

参数:

- 1.index: 必需。整数，规定添加/删除项目的位置，使用负数可从数组结尾处规定位置。
- 2.howmany: 可选。要删除的项目数量。如果设置为 0，则不会删除项目。
- 3.item1, ..., itemX: 可选。向数组添加的新项目。

返回值: 如果有元素被删除, 返回包含被删除项目的新数组。

复制代码

- sort() 数组排序

语法: `arrayObject.sort(sortby)`

参数:

- 1.sortby 可选。规定排序顺序。必须是函数。。

返回值: 返回包排序后的新数组。

复制代码

- pop() 删除一个数组中的最后的一个元素

语法: `arrayObject.pop()`

参数: 无

返回值: 返回被删除的元素。

复制代码

- shift() 删除数组的第一个元素

语法: `arrayObject.shift()`

参数: 无

返回值: 返回被删除的元素。

复制代码

- push() 向数组的末尾添加元素

语法: `arrayObject.push(newelement1,newelement2,....,newelementX)`

参数:

- 1.newelement1 必需。要添加到数组的第一个元素。
- 2.newelement2 可选。要添加到数组的第二个元素。
- 3.newelementX 可选。可添加若干个元素。

返回值: `arrayObject` 的新长度。

复制代码

- unshift() 向数组的开头添加一个或更多元素

语法: `arrayObject.unshift(newelement1,newelement2,....,newelementX)`

参数:

- 1.newelement1 必需。要添加到数组的第一个元素。
- 2.newelement2 可选。要添加到数组的第二个元素。
- 3.newelementX 可选。可添加若干个元素。

返回值: `arrayObject` 的新长度。

复制代码

- reverse() 颠倒数组中元素的顺序

语法: `arrayObject.reverse()`

参数: 无

返回值: 颠倒后的新数组。

复制代码

- copyWithin() 指定位置的成员复制到其他位置

语法: `array.copyWithin(target, start = 0, end = this.length)`

参数:

- 1.target (必需): 从该位置开始替换数据。如果为负值，表示倒数。
- 2.start (可选): 从该位置开始读取数据，默认为 0。如果为负值，表示倒数。
- 3.end (可选): 到该位置前停止读取数据，默认等于数组长度。如果为负值，表示倒数。

返回值: 返回当前数组。

复制代码

- `fill()` 填充数组

语法: `array.fill(value, start, end)`

参数:

- 1.`value` 必需。填充的值。
- 2.`start` 可选。开始填充位置。
- 3.`end` 可选。停止填充位置 (默认为 `array.length`)

返回值: 返回当前数组。

复制代码

不改变原数组的方法

- `slice()` 浅拷贝数组的元素

语法: `array.slice(begin, end);`

参数:

- 1.`begin`(可选): 索引数值, 接受负值, 从该索引处开始提取原数组中的元素, 默认值为0。
- 2.`end`(可选): 索引数值(不包括), 接受负值, 在该索引处前结束提取原数组元素, 默认值为数组末尾(包括最后一个元素)。

返回值: 返回一个从开始到结束 (不包括结束) 选择的数组的一部分浅拷贝到一个新数组对象, 且原数组不会被修改。

复制代码

- `join()` 数组转字符串

语法: `array.join(str)`

参数:

- 1.`str`(可选): 指定要使用的分隔符, 默认使用逗号作为分隔符。

返回值: 返回生成的字符串。

复制代码

- `concat()` 合并两个或多个数组

语法: `var newArr =oldArray.concat(arrayX,arrayX,.....,arrayX)`

参数:

- 1.`arrayX` (必须) : 该参数可以是具体的值, 也可以是数组对象。可以是任意多个。

返回值: 返回返回合并后的新数组。

复制代码

- `indexOf()` 查找数组是否存在某个元素

语法: `array.indexOf(searchElement,fromIndex)`

参数:

- 1.`searchElement` (必须): 被查找的元素
- 2.`fromIndex` (可选): 开始查找的位置(不能大于等于数组的长度, 返回-1), 接受负值, 默认值为0。

返回值: 返回下标

复制代码

- `lastIndexOf()` 查找指定元素在数组中的最后一个位置

语法: `arr.lastIndexOf(searchElement,fromIndex)`

参数:

- 1.`searchElement` (必须): 被查找的元素
- 2.`fromIndex` (可选): 逆向查找开始位置, 默认值数组的长度-1, 即查找整个数组。

返回值: 方法返回指定元素, 在数组中的最后一个的索引, 如果不存在则返回 -1。 (从数组后面往前查找)

复制代码

- `includes()` 查找数组是否包含某个元素

语法: `array.includes(searchElement,fromIndex=0)`

参数:

- 1.`searchElement` (必须): 被查找的元素
- 2.`fromIndex` (可选): 默认值为0, 参数表示搜索的起始位置, 接受负值。正值超过数组长度, 数组不会被搜索, 返回`false`。负值绝对值超过长数组度, 重置从0开始搜索。

返回值: 返回布尔

复制代码

7、立即执行函数

声明一个匿名函数, 马上调用这个匿名函数。目的是保护内部变量不受污染。

javascript 复制代码

```
(function(n1, n2) {
    console.log("这是匿名函数的自执行的第一种写法, 结果为:" + (n1 + n2))
})(10, 100);
(function start(n1, n2) {
    console.log("这是函数声明方式的自执行的第一种写法, 结果为:" + (n1 + n2))
})(10, 100);
(function(n1, n2) {
    console.log("这是匿名函数的自执行的第二种写法, 结果为: " + (n1 + n2))
})(10, 100));
(function start(n1, n2) {
    console.log("这是函数声明方式的自执行的第二种写法, 结果为: " + (n1 + n2))
})(10, 100));
```

8、js原型和原型链

每个对象都会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**

特点：JavaScript对象是通过引用来传递的，我们创建的每个新对象实体中并没有一份属于自己的原型副本，当我们修改原型时，与之相关的对象也会继承这一改变。当我们需要一个属性时，JavaScript引擎会先看当前对象中是否有这个属性，如果没有的话，就会查找它的prototype对象是否有这个属性，如此递推下去，一致检索到Object内建对象。

复制代码

```
function Func(){}
Func.prototype.name = "汪某";
Func.prototype.getInfo = function() {
    return this.name;
}
var person = new Func();
console.log(person.getInfo());//"汪某"
console.log(Func.prototype);//Func { name = "汪某", getInfo = function() }
```

参考：[js原型和原型链](#)

9、js中call,apply,bind

参考：[JavaScript中call,apply,bind方法的总结。](#)

10、Promise

一句话概括Promise：Promise对象用于异步操作，它表示一个尚未完成且预计在未来完成的异步操作。

promise是用来解决两个问题的：

- 回调地狱，代码难以维护，常常第一个的函数的输出是第二个函数的输入这种现象
- promise可以支持多个并发的请求，获取并发请求中的数据

这个promise可以解决异步的问题，本身不能说promise是异步的

javascript 复制代码

```
/*Promise 的简单实现*/

class MyPromise {
  constructor(fn) {
    this.resolvedCallbacks = [];
    this.rejectedCallbacks = [];
    this.state = "PENDING";
    this.value = "";
    fn(this.resolve.bind(this), this.reject.bind(this));
  }
  resolve(value) {
    if (this.state === "PENDING") {
      this.state = "RESOLVED";
      this.value = value;
      this.resolvedCallbacks.forEach(cb => cb());
    }
  }
  reject(value) {
    if (this.state === "PENDING") {
      this.state = "REJECTED";
      this.value = value;
      this.rejectedCallbacks.forEach(cb => cb());
    }
  }
  then(resolve = function() {}, reject = function() {}) {
    if (this.state === "PENDING") {
      this.resolvedCallbacks.push(resolve);
      this.rejectedCallbacks.push(reject);
    }
    if (this.state === "RESOLVED") {
      resolve(this.value);
    }
    if (this.state === "REJECTED") {
      reject(this.value);
    }
  }
}
```

11、async/await

如何使用 Async 函数

javascript 复制代码

```
async function timeout(ms) {
  await new Promise((resolve) => {
    setTimeout(resolve, ms);
  });
}

async function asyncPrint(value, ms) {
  await timeout(ms);
  console.log(value);
}
```

```
}

asyncPrint('hello world', 50);
```

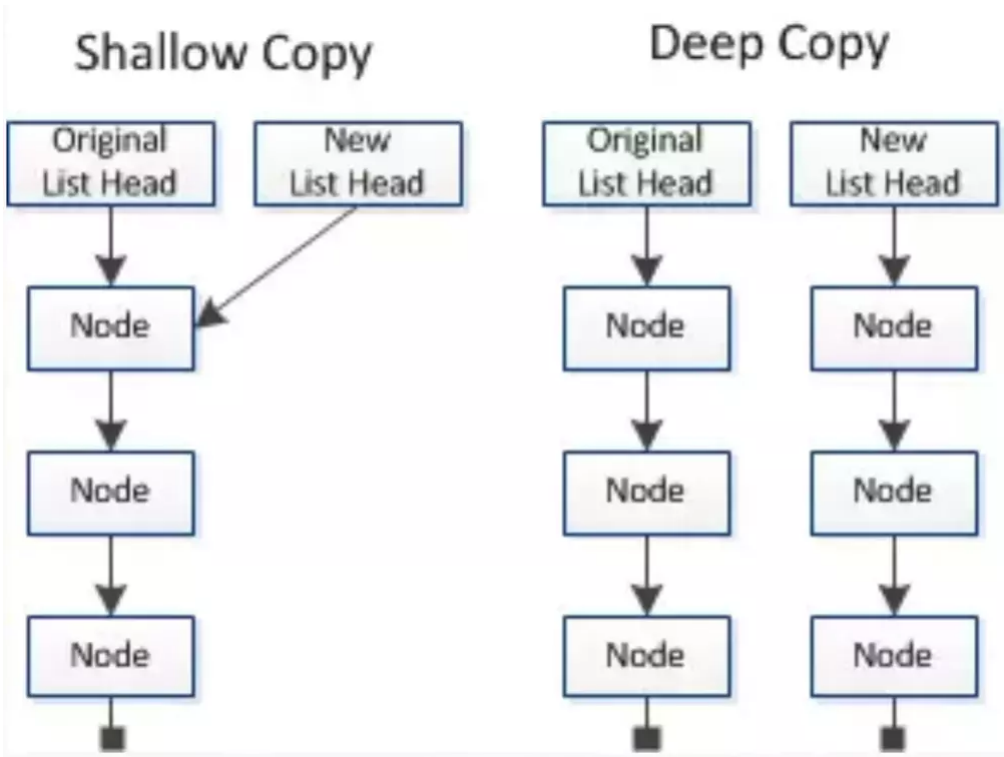
上面代码指定50毫秒以后，输出hello world。进一步说，async函数完全可以看作多个异步操作，包装成的一个 Promise 对象，而await命令就是内部then命令的语法糖。

待补充。。。

12、深拷贝、浅拷贝

浅拷贝和深拷贝都只针对于引用数据类型，浅拷贝只复制指向某个对象的指针，而不复制对象本身，新旧对象还是共享同一块内存；但深拷贝会另外创建一个一模一样的对象，新对象跟原对象不共享内存，修改新对象不会改到原对象；

区别：浅拷贝只复制对象的第一层属性、深拷贝可以对对象的属性进行递归复制；



浅拷贝的实现方式

1. 自定义函数

```
function simpleCopy (initalObj) {  
  var obj = {};  
  for ( var i in initaObj) {  
    obj[i] = initaObj[i];  
  }  
  return obj;  
}
```

javascript 复制代码

2. ES6 的 Object.assign()

```
let newObj = Object.assign({}, obj);
```

javascript 复制代码

3. ES6 的对象扩展

```
let newObj = {...obj};
```

javascript 复制代码

深拷贝的实现方式

1. JSON.stringify 和 JSON.parse

用 JSON.stringify 把对象转换成字符串，再用 JSON.parse 把字符串转换成新的对象。

```
let newObj = JSON.parse(JSON.stringify(obj));
```

javascript 复制代码

2. lodash

用 lodash 函数库提供的 _cloneDeep 方法实现深拷贝。

```
var _ = require('lodash');  
var newObj = _.cloneDeep(obj);
```

javascript 复制代码

3. 自己封装

javascript 复制代码

```
function deepClone(obj) {
  let objClone = Array.isArray(obj) ? [] : {};
  if (obj && typeof obj === "object") {
    // for...in 会把继承的属性一起遍历
    for (let key in obj) {
      // 判断是不是自有属性，而不是继承属性
      if (obj.hasOwnProperty(key)) {
        //判断obj子元素是否为对象或数组，如果是，递归复制
        if (obj[key] && typeof obj[key] === "object") {
          objClone[key] = this.deepClone(obj[key]);
        } else {
          //如果不是，简单复制
          objClone[key] = obj[key];
        }
      }
    }
  }
  return objClone;
}
```

13、跨域

跨域需要针对浏览器的同源策略来理解。同源策略指的是请求必须是同一个端口，同一个协议，同一个域名，不同源的客户端脚本在没有明确授权的情况下，不能读写对方资源。

受浏览器同源策略的影响，不是同源的脚本不能操作其他源下面的对象。想要操作另一个源下的对象是需要跨域。

- jsonp
- iframe
- 跨域资源共享(CORS)
- nginx 代理跨域

14、for in 和 for of

- for in

- 复制代码
- 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

- for of

- 复制代码
- 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

遍历对象

javascript 复制代码

```
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
}
```

15、如何阻止冒泡？

冒泡型事件：事件按照从最特定的事件目标到最不特定的事件目标(document对象)的顺序触发。

w3c的方法是e.stopPropagation(), IE则是使用e.cancelBubble = true。

复制代码

```
//阻止冒泡行为
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();
  else
    //否则，我们需要使用IE的方式来取消事件冒泡
    window.event.cancelBubble = true;
}
```

16、如何阻止默认事件？

w3c的方法是e.preventDefault(), IE则是使用e.returnValue = false

复制代码

```
//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中阻止函数默认动作的方式
    else
        window.event.returnValue = false;
    return false;
}
```

17、var,let,const

javascript 复制代码

```
//变量提升
console.log(a); // undefined
console.log(b); // 报错
console.log(c); // 报错
var a = 1;
let b = 2;
const c = 3;

// 全局声明
console.log(window.a) // 1

// 重复声明
let b = 200;//报错
```

其实这里很容易理解，var是可以变量提升的。而let和const是必须声明后才能调用的。对于let和const来说，这里就是暂缓性死区。

null	变量提升	重复声明	全局声明
var	yes	yes	yes
let	no	no	no
const	no	no	no

18、Class

es6新增的Class其实也是语法糖，js底层其实没有class的概念的，其实也是原型继承的封装。

javascript 复制代码

```
class People {
    constructor(props) {
        this.props = props;
        this.name = '汪某';
    }
    callMyName() {
        console.log(this.name);
    }
}

class Name extends People { // extends 其实就是继承了哪个类
    constructor(props) {
        // super相当于 把类的原型拿过来
        // People.call(this, props)
        super(props)
    }
    callMyApple() {
        console.log('我是汪某! ')
    }
}

let a = new Name('啊啊啊')
a.callMyName(); //汪某
a.callMyApple(); // 我是汪某!
```

19、Set

Set数据结构类似数组，但所有成员的值唯一。

javascript 复制代码

```
let a = new Set();
[1,2,2,1,3,4,5,4,5].forEach(x=>a.add(x));
for(let k of a){
    console.log(k)
};
// 1 2 3 4 5
```

基本使用

javascript 复制代码

```
let a = new Set([1,2,3,3,4]);
[...a]; // [1,2,3,4]
```

```
a.size; // 4

// 数组去重
[...new Set([1,2,3,4,4])]// [1,2,3,4]
```

方法

- add(value)：添加某个值，返回 Set 结构本身。
- delete(value)：删除某个值，返回一个布尔值，表示删除是否成功。
- has(value)：返回一个布尔值，表示该值是否为Set的成员。
- clear()：清除所有成员，没有返回值。

```
let a = new Set();
a.add(1).add(2); // a => Set(2) {1, 2}
a.has(2);        // true
a.has(3);        // false
a.delete(2);     // true  a => Set(1) {1}
a.clear();       // a => Set(0) {}
```

javascript 复制代码

20、Map

Map结构提供了“值—值”的对应，是一种更完善的 Hash 结构实现。

```
let a = new Map();
let b = {name: 'leo' };
a.set(b,'my name'); // 添加值
a.get(b);           // 获取值
a.size;             // 获取总数
a.has(b);           // 查询是否存在
a.delete(b);        // 删除一个值
a.clear();          // 清空所有成员 无返回
```

javascript 复制代码

基本使用

- 传入数组作为参数，指定键值对的数组。

```
let a = new Map([
  ['name','wzx'],
  ['age',23]
])
```

javascript 复制代码

- 如果对同一个键多次赋值，后面的值将覆盖前面的值。

```
let a = new Map();
a.set(1,'aaa').set(1,'bbb');
a.get(1); // 'bbb'
```

javascript 复制代码

- 如果读取一个未知的键，则返回undefined。

```
new Map().get('asdsad'); // undefined
```

javascript 复制代码

- 同样的值的两个实例，在 Map 结构中被视为两个键。

```
let a = new Map();
let a1 = ['aaa'];
let a2 = ['aaa'];
a.set(a1,111).set(a2,222);
a.get(a1); // 111
a.get(a2); // 222
```

javascript 复制代码

方法

- keys()：返回键名的遍历器。
- values()：返回键值的遍历器。
- entries()：返回所有成员的遍历器。
- forEach()：遍历 Map 的所有成员。

```
let a = new Map([
  ['name', 'leo'],
  ['age', 18]
])
for (let i of a.keys()) {
  console.log(i)
};
//name
//age

for (let i of a.values()) {
  console.log(i)
};
//leo
//18

for (let i of a.entries()) {
  console.log(i)
};
```

javascript 复制代码

```
//["name", "leo"]

a.forEach((v, k, m) => {
  console.log(`key:${k},value:${v},map:${m}`)
})
//["age", 18]
```

三、手撸代码

1、实现一个new操作符

javascript 复制代码

```
function New(func) {
  var res = {};
  if (func.prototype !== null) {
    res.__proto__ = func.prototype;
  }
  var ret = func.apply(res, Array.prototype.slice.call(arguments, 1));
  if ((typeof ret === "object" || typeof ret === "function") && ret !== null) {
    return ret;
  }
  return res;
}
var obj = New(A, 1, 2);
// equals to
var obj = new A(1, 2);
```

2、实现一个call或 apply

- call

javascript 复制代码

```
Function.prototype.call2 = function (context) {
  var context = context || window;
  context.fn = this;

  var args = [];
  for(var i = 1, len = arguments.length; i < len; i++) {
    args.push(arguments[i] + i + '');
  }

  var result = eval(`context.fn(' + args + ')`);

  delete context.fn;
  return result;
}
```

- apply

javascript 复制代码

```
Function.prototype.apply2 = function (context, arr) {
  var context = Object(context) || window;
  context.fn = this;

  var result;
  if (!arr) {
    result = context.fn();
  }
  else {
    var args = [];
    for (var i = 0, len = arr.length; i < len; i++) {
      args.push(`arr[' + i + '']`);
    }
    result = eval(`context.fn(' + args + ')`);
  }

  delete context.fn;
  return result;
}
```

参考：[JavaScript深入之call和apply的模拟实现](#)

3、实现一个Function.bind

javascript 复制代码

```
Function.prototype.bind2 = function (context) {
  if (typeof this !== "function") {
    throw new Error("Function.prototype.bind - what is trying to be bound is not callable");
  }
  var self = this;
  var args = Array.prototype.slice.call(arguments, 1);
  var fNOP = function () {};
  var fbound = function () {
    self.apply(this instanceof self ? this : context, args.concat(Array.prototype.slice.call(arguments)));
  }
  fNOP.prototype = this.prototype;
  fbound.prototype = new fNOP();
  return fbound;
}
```

参考：[JavaScript深入之bind的模拟实现](#)

4、实现一个继承

javascript 复制代码

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

Parent.prototype.sayName = function() {
    console.log('parent name:', this.name);
}

function Child(name, parentName) {
    Parent.call(this, parentName);
    this.name = name;
}

function create(proto) {
    function F() {}
    F.prototype = proto;
    return new F();
}

Child.prototype = create(Parent.prototype);
Child.prototype.sayName = function() {
    console.log('child name:', this.name);
}

Child.prototype.constructor = Child;
var parent = new Parent('汪某');
parent.sayName();// parent name: 汪某
var child = new Child('son', '汪某');
```

5、手写一个Promise(中高级必考)

面试够用版

javascript 复制代码

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

//同时, 需要在 myPromise的原型上定义链式调用的 then方法:
myPromise.prototype.then = function(onFullfilled, onRejected) {
    let self = this;
    switch (self.status) {
        case "resolved":
            onFullfilled(self.value);
            break;
        case "rejected":
            onRejected(self.reason);
            break;
        default:
    }
}

//测试一下:
var p = new myPromise(function(resolve, reject) {
    resolve(1)
});
p.then(function(x) {
    console.log(x)
})
```

高级版请参考：[史上最最最详细的手写Promise教程](#)

6、手写防抖(Debounce)和节流(Throttling)

完整版详见上方，此处给出面试版

javascript 复制代码

```
// 防抖函数
function debounce(fn, wait = 50, immediate) {
  let timer;
  return function() {
    if (immediate) {
      fn.apply(this, arguments)
    }
    if (timer) clearTimeout(timer)
    timer = setTimeout(() => {
      fn.apply(this, arguments)
    }, wait)
  }
}

// 节流函数
function throttle(fn, wait) {
  let prev = new Date();
  return function() {
    const args = arguments;
    const now = new Date();
    if (now - prev > wait) {
      fn.apply(this, args);
      prev = new Date();
    }
  }
}
```

javascript 复制代码

7、手写一个JS深拷贝

面试版

javascript 复制代码

```
function deepCopy(obj) {
  //判断是否是简单数据类型,
  if (typeof obj == "object") {
    //复杂数据类型
    var result = obj.constructor == Array ? [] : {};
    for (let i in obj) {
      result[i] = typeof obj[i] == "object" ? deepCopy(obj[i]) : obj[i];
    }
  } else {
    //简单数据类型 直接 == 赋值
    var result = obj;
  }
  return result;
}
```

四、VUE

1、Vue的双向数据绑定原理是什么？

vue.js 是采用数据劫持结合发布者-订阅者模式的方式，通过Object.defineProperty()来劫持各个属性的setter，getter，在数据变动时发布消息给订阅者，触发相应的监听回调。

html 复制代码

```
//vue实现数据双向绑定的原理就是用Object.defineProperty()重新定义（set方法）对象设置属性值和（get方法）获取属性值的操纵来实现的。
//Object.property()方法的解释：Object.property(参数1，参数2，参数3) 返回值为该对象obj
//其中参数1为该对象（obj），参数2为要定义或修改的对象的属性名，参数3为属性描述符，属性描述符是一个对象，主要有两种形式：数据描述符和存取描述符。这两种对象只能选择一种使用，不能混合使用。而get和set属于存取描述符对象的属性。
//这个方法会直接在一个对象上定义一个新属性或者修改对象上的现有属性，并返回该对象。

<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
</head>
<body>
  <div id="myapp">
    <input v-model="message" /><br>
    <span v-bind="message"></span>
  </div>
<script type="text/javascript">
  var model = {
    message: ""
  };
  var models = myapp.querySelectorAll("[v-model=message]");
  for (var i = 0; i < models.length; i++) {
    models[i].onkeyup = function() {
      model[this.getAttribute("v-model")] = this.value;
    }
  }
  // 观察者模式 / 钩子函数
  // defineProperty 来定义一个对象的某个属性
  Object.defineProperty(model, "message", {
    set: function(newValue) {
      var binds = myapp.querySelectorAll("[v-bind=message]");
      for (var i = 0; i < binds.length; i++) {
        binds[i].innerHTML = newValue;
      }
    }
  });
```



```
var models = myapp.querySelectorAll("[v-model=message]");
for (var i = 0; i < models.length; i++) {
    models[i].value = newValue;
};
this.value = newValue;
},
get: function() {
    return this.value;
}
})
</script>
</body>
</html>
```

2、请详细说下你对vue生命周期的理解？

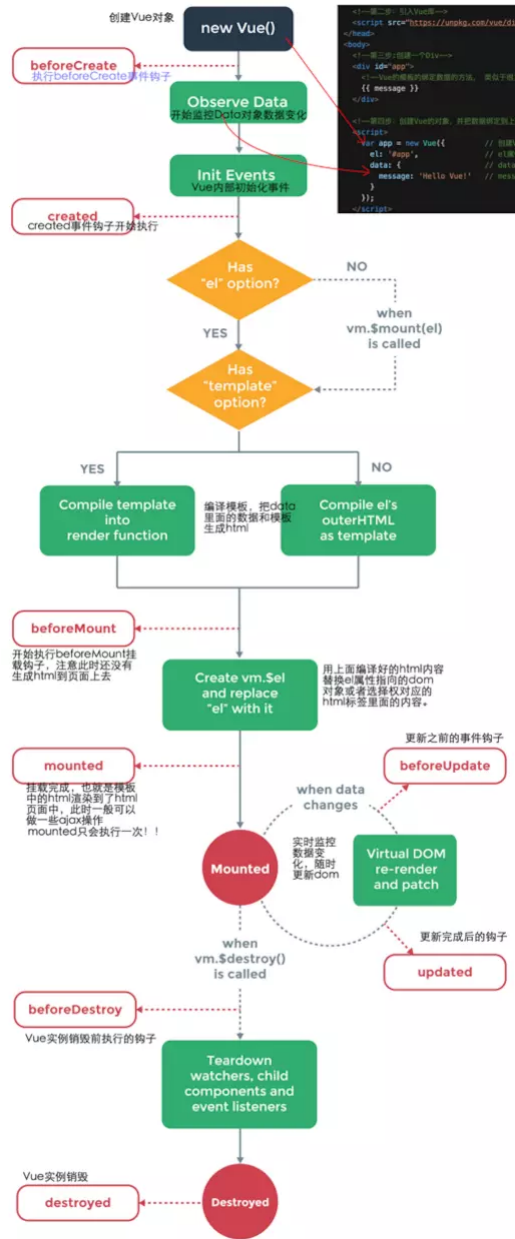
总共分为8个阶段创建前/后，载入前/后，更新前/后，销毁前/后

- beforeCreate 创建前执行（vue实例的挂载元素\$el和数据对象data都为undefined，还未初始化）
- created 完成创建 （完成了data数据初始化，el还未初始化）
- beforeMount 载入前（vue实例的\$el和data都初始化了，但还是挂载之前为虚拟的dom节点，data.message还未替换。）
- mounted 载入后html已经渲染(vue实例挂载完成，data.message成功渲染。)
- beforeUpdate 更新前状态（view层的数据变化前，不是data中的数据改变前）
- updated 更新状态后
- beforeDestroy 销毁前
- destroyed 销毁后（在执行destroy方法后，对data的改变不会再触发周期函数，说明此时vue实例已经解除了事件监听以及和dom的绑定，但是dom结构依然存在）

说一下每一个阶段可以做的事情

- beforeCreate:可以在这里加一个loading事件，在加载实例时触发。
- created:初始化完成时的事件写这里，如果这里结束了loading事件，异步请求也在这里调用。
- mounted:挂在元素，获取到DOM节点
- updated:对数据进行处理的函数写这里。
- beforeDestroy:可以写一个确认停止事件的确认框。

附上一张中文解析图



3、动态路由定义和获取

在 router 目录下的 index.js 文件中，对 path 属性加上 /:id。

使用 router 对象的 params.id 获取

4、vue-router 有哪几种导航钩子？

三种

- 1. 全局导航钩子（跳转前进行判断拦截）
 - router.beforeEach(to, from, next),
 - router.beforeResolve(to, from, next),
 - router.afterEach(to, from ,next)
- 2. 组件内钩子
 - beforeRouteEnter
 - beforeRouteUpdate
 - beforeRouteLeave
- 3. 单独路由独享组件
 - beforeEnter

5、组件之间的传值通信？

- 父组件向子组件传值：
 - 子组件在props中创建一个属性，用来接收父组件传过来的值；
 - 在父组件中注册子组件；
 - 在子组件标签中添加子组件props中创建的属性；

- 把需要传给子组件的值赋给该属性
- 子组件向父组件传值：
 - 子组件中需要以某种方式（如点击事件）的方法来触发一个自定义的事件；
 - 将需要传的值作为\$emit的第二个参数，该值将作为实参传给响应事件的方法；
 - 在父组件中注册子组件并在子组件标签上绑定自定义事件的监听。

6、vuex

是一个能方便vue实例及其组件传输数据的插件 方便传输数据，作为公共存储数据的一个库

state: 状态中心
mutations: 更改状态，同步的
actions: 异步更改状态
getters: 获取状态
modules: 将state分成多个modules，便于管理

应用场景：单页应用中，组件之间的状态。音乐播放、登录状态、加入购物车。

网上找的一个通俗易懂的了解vuex的例子

公司有个仓库
1.State （公司的仓库）
2.Getter （只能取出物品，包装一下，不能改变物品任何属性）
3.Muitation （仓库管理员，只有他可以直接存储到仓库）
4.Action （公司的物料采购员，负责从外面买东西和接货， 要往仓库存东西，告诉仓库管理员要存什么）
非常要注意的地方：只要刷新或者退出浏览器，仓库清空。

最后

以上为本小白个人总结，如有不对的地方，欢迎各位大佬在评论区指正，共勉！