面试准备

JS

H3 1. 数据类型

基础数据类型（**传值**）：number、undefined、string、boolean、null

复杂数据类型（**传址**）：Array、Object

基本数据类型是指存放在栈中的简单数据段，数据大小确定，内存空间大小可以分配，它们是直接按值存放的，所以可以直接按值访问

H3 2. 数组数据类型判断

typeof 主要用来判断基本数据类型：string、boolean、number，判断 array、object、null 结果都为object

instanceof 判断instanceof构造原型

Object.prototype.toString.call() 最全面

3.数组常用方法

join

数组变为字符串，不改变原数组

split

以字符为标准进行分割字符串，返回数组

push

添加到数组尾部，改变原数组，返回最新的数组长度为数字

pop

删除数组尾部，改变原数组，返回最新的数组长度为数字

unshift

添加到数组头部，改变原数组，返回最新的数组长度为数字

H4 shift

删除数组头部，改变原数组，返回最新的数组长度为数字

H4 reverse

翻转数组，更改原数组

H4 sort

排序，sort((a, b) => {return a-b})

return 数字 < 0 升序，反之降序

更改原数组，返回拼接后的数组

H4 concat

数组拼接。

array1.concat(array2)， 不更改原数组 ，返回拼接后的新数组

H4 splice

截取/删除数组，3个参数非必填

array.splice(开始截取的数组下标， 截取的长度， 填充替换的数组)

更改原数组，返回删除的数据元素

slice

截取/删除数组，不更改原数组

array.slice(开始截取的数组下标，结束截取的数组下标（并且不包含））

forEach

为数组里面每一项进行操作

map

为数组里面每一项进行操作，并返回一个新数组，新数组为map的return结果

filter

循环过滤符合条件的数组，返回一个新数组

find

查找符合条件的元素返回，只会返回符合条件的第一个

findIndex

查找符合条件的元素返回，只会返回符合条件的第一个索引值

some

有真则真，返回boolean

every

有假则假，返回boolean

reduce

累加

4.对象的深拷贝和浅拷贝

深拷贝方法

01. **JSON.parse(JSON.stringify(obj))**

把对象转换为字符串，再把字符串转换为对象

缺点:undefined和函数无法赋值，仅限于拷贝一些普通的对象

02. **递归**

```
var oneObj = {
    a:1,
    b:2,
    list: ["a", "b"],
    sonObj: {},
    c: undefined,
    d: () => {
        alert("你好")
    }
}

function deep(obj) {
    // 判断数据类型是否是数组/对象 进行新数据声明
    // 新建一个地址
    var targetObj = obj instanceof Array ? [] : {}
    // 循环obj
    for (const key in obj) {
        // 忽略从原型继承的属性，不进行拷贝
        if (obj.hasOwnProperty(key)) {
            // 对应的value
            const value = obj[key];
            // 判断value是数组/对象
            if (value instanceof Array || value instanceof Object) {
                // 递归复制新的地址数据
                targetObj[key] = deep(value)
            } else {
                targetObj[key] = value
            }
        }
    }
    return targetObj;
}
```

H3 5. this指向及改变

this指 当前代码执行的上下文

普通函数的this是指向的时候绑定，箭头函数时声明的时候进行绑定的this

setTimeOut的this指向window

01. 全局this指向window

02. 在方法/函数执行的内部this指向调用它的对象

03. **箭头函数里面的this指向是声明的当前上下文环境，并且不可改变this指向**

```
var obj = {
    oson1: {
        oson2: {
            fun1: () => {
                // 指向全局window
                console.log("箭头函数this", this)
            },
            fun2: function () {
                // 指向oson2
                console.log("普通函数this", this)
            }
        }
    }
}
obj.oson1.oson2.fun1()
obj.oson1.oson2.fun2()
```

H4 改变this指向的三个方法

01. apply：改变this指向并立即执行函数，参数以数组形式写

02. call：改变this指向并立即执行函数，参数逗号分割

03. bind：语法和call一样，但是改变完不会立即执行

```
obj.oson1.oson2.fun2.apply({a:123}, ["A", "B"]) // this指向{a:123}
obj.oson1.oson2.fun2.call({a:456}, "C", "D")    // this指向{a:456}
// ABCD为参数
obj.oson1.oson2.fun2.bind({a:789}, "E", "F")    // 不会执行
obj.oson1.oson2.fun2.bind({a:789}, "E", "F")()  // 执行
```

H3 6. 异步解决方案

01. callback
02. promise
03. async/await

@ 最全前端异步解决方案【推荐收藏】 - 掘金 (juejin.cn)

H3 7. 原型、原型链

原型类别：

- 显示原型：prototype，是每个函数function独有的属性
- 隐式原型：_ proto _，是每个对象都具有的属性

原型与原型链：

- 原型：一个函数看成一个类，原型是所有类都有的一个属性，原型的作用就是给这个类的对象都添加一个统一的方法
- 原型链：每个对象都有一个_ proto _，它指向它的prototype原型对象；它的prototype原型对象又有一个 _proto _指向它的prototype原型对象，层层向上最终找到顶级对象Object的prototype，这个查询路径就是原型链

每个函数都有prototype（原型）属性，这个属性是一个指针，指向一个对象，这个对象的用途是包含特定类型的所有实例共享的属性和方法，这个对象（原型对象）是用来给实例共享属性和方法的。

H3 8. 闭包

闭包就是能够读取其他函数内部变量的函数。在本质上，闭包是将函数内部和函数外部连接起来的桥梁。

作用：封装变量，收敛权限

优点：不会造成变量污染

缺点：闭包被子函数引用的变量不会被回收，有可能造成内存泄露

H3 9. 防抖、节流

防抖：函数触发的时间间隔小于设定时间，只执行一次

```
const btn = document.querySelector('#btn')
function debounce(fn, delay) {
    var time = null;
    return function () {
        // 不让定时器执行
        clearTimeout(time)
        time = setTimeout(() => {
```

```
            fn()
        }, delay)
    }
}
function fn () {
    console.log('抖一抖')
}
btn.onclick = debounce(fn, 500)
```

节流：在设定之间间隔内执行一次

```
const btn = document.querySelector('#btn')
function throttle(fn, delay) {
    /*
    记录上一次点击的事件，和本次点击时间对比，符合时间间隔则执行函数否则不执行
    */
    var lastTime = 0;
    return function () {
        var nowTime = new Date().getTime()
        if (newTime - lastTime > delay) {
            fn()
            lastTime = nowTime
        }
    }
}
function fn () {
    console.log('节流啦')
}
btn.onclick = debounce(fn, 500)
```

H3 10. Vue2.x生命周期

01. beforeCreate

02. created

03. beforeMount

04. mounted

05. beforeUpdate

06. updated

07. beforeDestroy

08. destroyed

H3 11. js实现v-model双向绑定

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <div>
    展示：<h1></h1>
    输入：<input type="text">
  </div>
</body>
</html>
<script>
  // 挟持数据
  function definPropertyFn() {
    let obj = {}
    let val = null;

    Object.defineProperties(obj, {
      val: {
        get() {
          return val;
        },
        set(newV) {
          val = newV
          // 数据控制视图，将更改的数据复制给h1
          document.querySelector('h1').innerText = newV
```

```
            console.log('调用了set,获取: ' + newV, val);
          }
        }
      })

      return obj
    }

    let newObj = definPropertyFn()
    // 调用get，执行数据渲染视图
    document.querySelector('h1').innerText = newObj.val
    document.querySelector('input').value = newObj.val

    // 监听视图的input标签，标签一变动，将最新数据获取调用set，赋值给val，并赋值给h1
    document.querySelector('input').addEventListener('input', function ()
{
      newObj.val = document.querySelector('input').value
    })
  </script>
```

H3 12. vue2生命周期

01. **beforeCreate:** data和methods尚未初始化
02. **created:** data和methods已经初始化
03. **beforeMount:** 将要把内存中的HTML渲染到页面上
04. **mounted：** HTML渲染完毕
05. **beforeUpdate：** data数据发生变化，即将重新渲染
06. **updated：** 页面完成更新
07. **beforeDestroy：** vm实例将被销毁
08. **destroyed：** vm实例已经被销毁

13. Vue组件通信

@ Vue组件之间的传值 - 掘金 (juejin.cn)

父传子：props

```
// 父组件
<template>
    <div>
        //父组件自定义msg属性给子组件传值
        <Child :msg="msg"></Child>
    </div>
</template>

<script>
import Child from "@/components/Child"
export default {
  data() {
     return {
        msg: "父组件传给子组件的值"
     }
  },
  components: {
     Child
  }
}
</script>


// 子组件
<template>
    <div>
        {{msg}}
    </div>
</template>

<script>
export default {
  //子组件通过props接收父组件的传值
  props: ["msg"],
  data() {
```

```
    return {

    }
  }
}
</script>
```

H4 父传子：$ref

```
// 父
<template>
    <div>
        <Child ref="child"></Child>
    </div>
</template>

<script>
import Child from "@/components/Child"
export default {
  data() {
    return {
        msg: "父组件传给子组件的值"
    }
  },
  mounted() {
    //父组件通过ref属性调用子组件的方法
    this.$refs.child.getMsg(this.msg)
  },
  components: {
    Child
  }
}
</script>

// 子
<template>
    <div>
        {{msg}}
    </div>
</template>
```

```
<script>
export default {
  data() {
     return {
        msg: "
     }
  },
  methods: {
     //子组件获取父组件值的方法
     getMsg(val) {
        this.msg = val
     }
  }
}
</script>
```

H4 父传子：$children

```
// 父
<template>
    <div>
        <Child ref="child"></Child>
    </div>
</template>

<script>
import Child from "@/components/Child"
export default {
  data() {
     return {
        msg: "父组件传给子组件的值"
     }
  },
  mounted() {
     //父组件通过$children[0]访问对应子组件
     this.$children[0].msg = this.msg
  },
  components: {
     Child
  }
```

```
}
</script>


// 子
<template>
    <div>
        {{msg}}
    </div>
</template>

<script>
export default {
  data() {
     return {
        msg: ""
     }
  }
}
</script>

<style scoped>
</style>
```

H4 父传子孙：provide/inject

```
// 父
<template>
    <div>
        <Child></Child>
    </div>
</template>

<script>
import Child from "@/components/Child"
export default {
  data() {
     return {

     }
  },
```

```
  //父组件通过provide方法向子孙组件提供值
  provide() {
     return {
        msg: "父组件提供给子孙组件的值"
     }
  },
  components: {
     Child
  }
}
</script>

// 子
<template>
    <div>
        {{msg}}
    </div>
</template>

<script>
export default {
  //子孙组件通过inject注入父组件提供的值
  inject: ['msg'],
  data() {
     return {

     }
  }
}
</script>
```

`props` 和 `$ref` 和 `$children` 和 `provide/inject` 的主要区别：

- props 侧重于数据的传递，并不能获取子组件里的属性和方法，适用于自定义内容的使用场景
- $ref 侧重于获取子组件里的属性和方法，并不是太适合传递数据，并且 ref 常用于获取dom元素，起到选择器的作用
- $children 侧重于获取所有的直接子组件，得到的是一个无序的数组，并不太适合向多个子组件传递数据

- provide/inject 侧重于在开发高阶插件/组件库时使用，并不推荐用于普通应用程序代码中

H4 子传父：$emit

```
// 子
<template>
    <div>
        <button @click="sendMsg">子组件传值给父组件</button>
    </div>
</template>

<script>
export default {
  data() {
     return {
        msg: "子组件传给父组件的值"
     }
  },
  methods: {
     //子组件通过$emit触发自定义事件给父组件传值
     sendMsg() {
        this.$emit("getMsg", this.msg);
     }
  }
}
</script>

// 父
<template>
    <div>
        //父组件通过绑定自定义事件接收子组件的传值
        <Child @getMsg="getData"></Child>
        <p>{{msg}}</p>
    </div>
</template>

<script>
import Child from "@/components/Child";
```

```
export default {
  data() {
     return {
        msg: ""
     }
  },
  methods: {
     getData(data) {
        this.msg = data
     }
  },
  components: {
     Child
  }
}
</script>
```

H4 子传父：$parent

```
// 子
<template>
    <div>
        <button @click="sendMsg">子组件传值给父组件</button>
    </div>
</template>

<script>
export default {
  data() {
     return {
        msg: "子组件传给父组件的值"
     }
  },
  methods: {
     //子组件通过$parent访问父组件
     sendMsg() {
        this.$parent.msg = this.msg
     }
  }
}
</script>
```

```
// 父
<template>
    <div>
        <Child></Child>
        <p>{{msg}}</p>
    </div>
</template>

<script>
import Child from "@/components/Child";
export default {
  data() {
     return {
        msg: ""
     }
  }
  components: {
     Child
  }
}
</script>
```

H4 兄弟组件互传：eventBus

创建 **eventBus.js**

```
import Vue from "vue"
export default new Vue()
```

```
// 组件A
<template>
    <div>
```

```
        <button @click="sendMsg">子组件A传值给子组件B</button>
    </div>
</template>

<script>
import eventBus from '../eventBus.js'
export default {
  data() {
     return {
        msg: "子组件A传给子组件B的值"
     }
  },
  methods: {
     sendMsg() {
        //子组件A通过eventBus.$emit触发自定义事件给子组件B传值
        eventBus.$emit("getMsg", this.msg);
     }
  }
}
</script>

// 组件B
<template>
    <div>
        {{msg}}
    </div>
</template>

<script>
import eventBus from '../eventBus.js'
export default {
  data() {
     return {
        msg: ""
     }
  },
  created() {
     this.getData()
  },
  methods: {
     getData() {
        //子组件B通过eventBus.$on注册自定义事件接收子组件A的传值
        eventBus.$on("getMsg", (data) => {
           this.msg = data
        })
     }
  }
}
```

```
</script>
```

H3 14. MVVM模式和MVC模式

H4 MVVM模式

M模型——V视图——VM视图模型

M和V无法直接通信，必须通过VM进行交互，VM与M和V之间都是双向绑定的

优点：

- 低耦合: `View` 可以独立于 `Model` 变化和修改,一个 `ViewModel` 可以绑定到不同的 `View` 上,当 `View` 变化的时候 `Model` 可以不变,当 `Model` 变化的时候 `View` 也可以不变。
- 可重用性: 可以把一些视图逻辑放在一个 `ViewModel` 里面,让很多 `View` 重用这段视图逻辑。
- 独立开发: 开发人员可以专注于业务逻辑和数据的开发,设计人员可以专注于页面的设计。
- 可测试：界面素来是比较难于测试的，而现在测试可以针对 `ViewModel` 来写。

H4 MVC模式

M模型——V视图——C控制器

MVC之中通信是单向的：V->C->M->V

MVC与MVVM的区别:

MVC和MVVM的区别并不是VM完全取代了C，ViewModel存在目的在于抽离Controller中展示的业务逻辑，而不是替代Controller，其它视图操作业务等还是应该放在Controller中实现。也就是说MVVM实现的是业务逻辑组件的重用。

- MVC中Controller演变成MVVM中的ViewModel
- MVVM通过数据来显示视图层而不是节点操作
- MVVM主要解决了MVC中大量的dom操作使页面渲染性能降低,加载速度变慢,影响用户体验

15. 如何理解构造函数，原型对象和实例的关系

@ 如何理解构造函数，原型对象和实例的关系

每个构造函数都有一个原型对象，原型对象都包含一个指向构造函数的指针，而实例都包含一个指向原型对象的内部指针。

16. p标签和div标签的区别

div与p均独占一行的块元素标签，唯一区别，p自带有一定margin-top和margin-bottom属性值，而div两个属性值为0，也就是两个p之间有一定间距，而div没有。

H3 17. CSS盒子模型

包含：margin、border、padding、content

H3 18. 清除浮动的方式

01. 给父元素设置高度
02. 给父元素添加float
03. 添加额外标签，并clear:both
04. 给父元素添加overflow:hidden
05. 给父元素设置after伪元素

H3 19. 输入url到页面显示，发生了什么

- DNS 解析:将域名解析成 IP 地址
- TCP 连接：TCP 三次握手
- 发送 HTTP 请求
- 服务器处理请求并返回 HTTP 报文
- 浏览器解析渲染页面
- 断开连接：TCP 四次挥手

H3 20. web前端优化策略

H4 浏览器

- 减少http请求
- 使用http2.0
- 设置浏览器缓存策略
- 白屏时间做成动画

H4 资源

- 静态资源单独域名
- gzip压缩
- 做服务端渲染（SSR）
- 将CSS放在文件头部，JS放在文件底部

H4 图片

- 字体图标代替图片图标
- 精灵图
- 图片懒加载
- 图片预加载
- 使用png格式的图片
- 小于10k的图片可以打包为base64格式

H4 代码

- 慎用全局变量
- 缓存全局变量

- 减少重绘回流
- 节流、防抖
- 少用闭包、减少内存泄露
- 减少数据读取次数

H3 21. package.json的作用

管理项目中使用到的外部依赖包，同时它也是 NPM 命令的入口文件。

H3 22. 垃圾回收机制

将内存中不再使用的对象清除

H4 引用标记法

优势：

- 可即可回收垃圾
- 最大暂停时间很短

缺点：

- 时间开销大
- 无法解决循环引用的问题

H4 标记清除法

过程：

01. 垃圾收集器在运行时将内存中的所有变量都加上一个标记，假设内存中所有对象都是垃圾，全标记为0
02. 从根对象开始深度遍历，把不是垃圾的节点改成1
03. 清除所有标记为0的垃圾，销毁并回收它们所占用的内存空间
04. 最后把内存中的所有对象标志修改为0，等待下一轮的垃圾回收

优点：

- 实现简单
- 解决了循环引用的问题

缺点：

- 内存碎片化
- 再分配时，如果一直没有找到合适的内存块大小，那么会遍历空闲链表一直遍历到尾端
- 不会立即回收资源

H3 23. http常见状态码

H4 2XX——表明请求被正常处理了

200 OK：请求已正常处理。

3XX——表明浏览器需要执行某些特殊的处理以正确处理请求

301 Moved Permanently：资源的uri已更新，你也更新下你的书签引用吧。永久性重定向

304 Not Modified：自上次访问以来，请求的资源未被修改。

4XX——表明客户端是发生错误的原因所在

400 Bad Request：服务器端无法理解客户端发送的请求

401 Unauthorized：该状态码表示发送的请求需要有通过HTTP认证（BASIC认证，DIGEST认证）的认证信息

403 Forbidden：不允许访问那个资源

404 Not Found：服务器上没有请求的资源。路径错误等

5XX——服务器本身发生错误

500 Internal Server Error：貌似内部资源出故障了

24. sass

在 CSS 语法的基础上增加了 **变量 (variables)、嵌套 (nested rules)、混合 (mixins)、继承(extend)、导入 (inline imports)** 等高级功能

25. get和post区别

@ 编程入门教程 名企面试真题面经 牛客网_牛客网 (nowcoder.com)

@ Http 浅谈Get和POST的区别 - 掘金 (juejin.cn)

H3 26. async&await用法

@ async&await用法 - 知乎 (zhihu.com)

async/await 是ES7提出的基于@ Promise 的解决异步的最终方案。

async 表示函数里有异步操作，

await 表示紧跟在后面的表达式需要等待结果。

同 Generator 函数一样， **async** 函数返回一个 **Promise** 对象，可以使用 **then** 方法添加回调函数。当函数执行的时候，一旦遇到 **await** 就会先返回，等到触发的异步操作完成，再接着执行函数体内后面的语句。

H3 27. Vuex五大属性

- state：存放数据
- mutations：更新state，只能是同步
- actions：主要用来进行异步操作
- getters：从基础数据上派生，相当于state的计算属性
- modules：可以将store分割成模块

H3 28. Promise

Promise 对象用于一个异步操作的最终完成（或失败）及其结果值的表示。简单点说，它就是用于处理异步操作的，异步处理成功了就执行成功的操作，异步处理失败了就捕获错误或者停止后续操作。

promise构造函数接受一个执行器作为参数，函数被调用时，会传入resolve和reject两个参数。

当resolve函数被调用时，Promise状态更新为fulfilled(完成)

当reject函数被调用时，Promise状态更新为rejected(失败)

如果excutor内部抛出异常，Promise状态将直接变更为rejected，错误对象将作为结果返回

H3 29. JS为什么是单线程

- js使用的事件循环机制，同步非阻塞，无需创建更多的线程。
- 单线程降低了内存开销，避免了上线文切换。

H3 30. 冒泡、捕获

@ 简述 JavaScript 的事件捕获和事件冒泡 - 掘金 (juejin.cn)

事件捕获是事件传播的初始场景，从包装元素开始，一直到启动事件生命周期的目标元素。

事件冒泡将从一个子元素开始，在 DOM 树上传播，直到最上面的父元素事件被处理。

H3 31. 事件委托/事件代理

事件委托就是把原本需要绑定在子元素上的事件（onclick、onkeydown 等）委托给它的父元素，让父元素来监听子元素的冒泡事件，并在子元素发生事件冒泡时找到这个子元素。

事件委托是利用事件的冒泡原理来实现的，大致可以分为三个步骤：

01. 确定要添加事件元素的父级元素；
02. 给父元素定义事件，监听子元素的冒泡事件；
03. 使用 event.target 来定位触发事件冒泡的子元素。

32. 作用域

33. 模板引擎

34. 跨域

01. JSONP
02. CORS
03. nginx代理跨域

35. JWT认证

36. 重绘和回流怎么优化

CSS：

- 使用 transform 替代 top
- 使用 visibility 替换 display: none ，因为前者只会引起重绘，后者会引发回流（改变了布局）
- 避免使用table布局，可能很小的一个小改动会造成整个 table 的重新布局。
- 尽可能在DOM树的最末端改变class，回流是不可避免的，但可以减少其影响。尽可能在DOM树的最末端改变class，可以限制了回流的范围，使其影响尽可能少的节点。
- 避免设置多层内联样式，CSS 选择符从右往左匹配查找，避免节点层级过多。

JS：

- 避免频繁操作样式，最好一次性重写style属性，或者将样式列表定义为class并一次性更改class属性。
- 避免频繁操作DOM，创建一个documentFragment，在它上面应用所有DOM操作，最后再把它添加到文档中。
- 避免频繁读取会引发回流/重绘的属性，如果确实需要多次使用，就用一个变量缓存起来。
- 对具有复杂动画的元素使用绝对定位，使它脱离文档流，否则会引起父元素及后续元素频繁回流。

H3 37. 浏览器渲染过程

01. 解析HTML生成DOM树
02. 解析CSS生成CSSOM树
03. 结合DOM树和CSSOM树，生成渲染树
04. 回流：根据生成的渲染树，进行回流(Layout)，得到节点的几何信息（位置，大小）
05. 重绘：根据渲染树以及回流得到的几何信息，得到节点的绝对像素
06. 绘制

H3 38. 浏览器缓存

H3 39. http、https的区别

@ HTTP和HTTPS有什么区别？ - 知乎 (zhihu.com)

H3 40. 加密手段（hash函数怎么实现）

01. base64加密

02. MD5加密

03. sha1加密

04. AES/DES加密解密

H3 41. TCP建立连接过程

H3 42. null和undefined的区别

null是Object的一个特殊值，值 `null` 是一个字面量，不像 `undefined` ，它不是全局对象的一个属性。`null` 是表示缺少的标识，指示变量未指向任何对象。把 `null` 作为尚未创建的对象，也许更好理解。在 API 中，`null` 常在返回类型应是一个对象，但没有关联的值的地方使用。

`undefined` 是 全局对象 的一个属性。也就是说，它是全局作用域的一个变量。`undefined` 的最初值就是原始数据类型 `undefined` 。

H3 43. v-for为啥要加key

`v-for` 默认使用 **就地复用策略** ，列表数据修改的时候，他会根据key值去判断某个值是否修改，如果修改，则重新渲染这一项，否则复用之前的元素,如果不绑定key的话,每次修改某一条数据,都会重新渲染所有数据,会导致大量内存的浪费。如果绑定了key，每次修改某一条数据的时候，就只会重新渲染改条数据的变化，节省了大量的内存。

H3 44. 箭头函数和普通函数的区别

箭头函数中的this在声明时的上下文环境确定，且无法更改，不能作为构造函数使用，箭头函数没有自己的arguments，箭头函数没有原型prototype

普通函数的this指向调用它的对象