Table of Contents *generated with DocToc*

面试题

算法

把一个数组旋转k步

> 时间复杂度 `O(1)`，后面部分裁切，放到前面

```
function rotate(arr:number[], k: number): number {
    const length = arr.length
    if(!k || length === 0) return arr
    const step = Math.abs(k % length) // abs 取绝对值
    const part1 = arr.slice(-step)
    const part2 = arr.slice(0, length - step)
    const all = part1.concat(part2)
    return all
}
```

判断一个字符串是否是括号匹配

> 可以用栈实现
> 栈：逻辑结构
> 时间复杂度 `O(n)`

```
/**
 * 判断是否括号匹配
 */
function isMatch(left: string, right: string): boolean {
    if(left === '{' && right === '}') return true
    if(left === '(' && right === ')') return true
    if(left === '[' && right === ']') return true
    return false
}

/**
 * 判断是否括号匹配
 */
function matchBracket(str: string): boolean {
    const length = str.length
    if(length === 0) return true

    const stack = []
    const leftSymbols = '{[('
    const rightSymbols = ')]}'

    // 遍历字符串每一项判断入栈和出栈
    for(let i = 0; i < length; i++) {
        const s = str[i]
        // 左括号压栈 右括号匹配出栈
        if(leftSymbols.includes(s)) {
            stack.push(s)
        } else if(rightSymbols.includes(s)) {
            const top = stack[stack.length - 1] // 栈顶
            if (isMatch(top, s)) {
                stack.pop()
            } else {
                return false
            }
        }
    }

    return stack.length === 0
}
```

用两个栈实现一个队列

> 队列：逻辑结构

```
/**
 * 用两个栈实现一个队列
 */
class MyQueue {
    // 定义两个栈 private 私有 在内部可以调用 外部无法调用
    private stack1: number[] = []
    private stack2: number[] = []

    /**
     * 入队
     */
    add(n: number) {
        this.stack1.push(n)
    }

    /**
     * 出队
     */
    delete(): number | null {
        let res
        const stack1 = this.stack1
        const stack2 = this.stack2

        // 1. 将 stack1 中所有元素移动到 stack2 中
        while (stack1.length) {
            const n = stack1.pop()
            if(n !== null) {
                stack2.push(n)
            }
        }

        // 2. stack2 pop
        res = stack2.pop()

        // 将 stack2 中所有元素放回 stack1 栈
        while (stack2.length) {
            const n = stack2.pop()
            if(n !== null) {
                stack1.push(n)
            }
        }

        return res || null
    }

    // 通过属性方式调用
    get length(): number {
        return this.stack1.length
    }
}
```

使用js反转单向链表

> 链表：物理结构
> 数组需要一段连续的内存空间，而链表是零散的
> 链表节点的数据结构： `{value,next?,prev?}`

实际工作经验

H5页面如何进行首屏优化

1. 路由懒加载（SPA）

- 路由拆分，优先保证首页加载

2. 服务端渲染 SSR （成本高）

- 传统的前后端分离（SPA）渲染页面过程复杂
- SSR渲染页面过程简单，所有性能好
- 如果是纯H5页面，SSR是性能优化的终极方案（直接渲染HTML）

3. App 预取

- 如果H5在 App WebView 中展示，可使用App预取
- 用户访问列表页时，App预加载文章首屏内容
- 用户进入H5页，直接从App中获取内容，瞬间展示首屏

4. 分页

- 针对列表页
- 默认只展示第一页内容

5. 图片懒加载

- 针对详情页
- 默认只展示文本内容，然后触发图片懒加载
- 注意：提前设置图片尺寸，尽量只重绘不重拍

6. Hybrid

- 提前将 HTML JS CSS 下载到 App 内部
- 在App webview 中使用 `file://` 协议（打开本地文件）加载页面文件
- 再用 Ajax 获取内容并展示（也结合App预取）

注意：

- 性能优化要配合分析、统计、评分等，做了事情要有结果
- 性能优化也要配合体验，如骨架屏，loading动画等

后端一次性返回10w条数据，要怎么办

1. 设计不合理

- 主动和面试官沟通

2. 浏览器能否处理10w条数据

- JS没问题
- 渲染到 DOM 会非常卡顿

3. 自定义中间层

- 自定义 nodejs 中间层，获取并拆分这10w条数据
- 前端对接 nodejs 中间层，而不是服务端
- 成本比较高

4. 虚拟列表

- 只渲染可视区内 DOM
- 其他隐藏区域不显示，只用 `<div>` 撑起高度
- 随着浏览器滚动，创建和销毁 DOM

5. 虚拟列表 - 第三方 lib

- 实现起来复杂，借用第三方 lib
- Vue-virtual-scroll-list
- React-virtualiszed

注意：

- 要主动沟通，表达观点
- 后端的问题，首先要用后端的思维去解决 - 中间层
- 虚拟列表只是无奈的选择，实现复杂而且效果不一定好（低配手机）

前端的常用设计模式和使用场景

1. 设计原则

- 最重要的思想：开放封闭原则
  - 对扩展开放
  - 对修改封闭

2. 工厂模式

- 用一个工厂函数创建实例，隐藏 `new`
  - 如jQuery `$` 函数
  - 如 React `createElement` 函数

```
class Foo {}

// 工厂模式
function factory(a,b,c) {
    // if else
    return new Foo()
}

const f = factory(1,2,3)

// $('div')
// => new JQuery()
```

3. 单例模式

- 全局唯一的实例（无法生成第二个）
  - 如 Vuex Redux 的 `store`
  - 如全局唯一的 `dialog model`

```
class SingleTon {
    private static instance: SingleTon | null = null
    private constructor() {}
    public static getInstance(): SingleTon {
        // 没有的话创建新的实例，有的话直接返回
        if (this.instance === null) {
            this.instance = new SingleTon()
        }
        return this.instance
    }
    fn1()
    fn2()
}

// const s = new SingleTon() => 报错 => 私有的构造函数并且只有在class内部才可以访问
const s = SingleTon.getInstance() // getInstance 静态方法
s.fn1()
s.fn2()

const s1 = SingleTon.getInstance()
s === s1 // true
```

注意：

- JS 是单线程的 创建单例很简单
- Java 是支持多线程的，创建单例要考虑锁死线程
    - 否则多个线程同时创建，单例就重复了（多线程共享进程内存）

4. 代理模式

- 使用者不能直接访问对象，而是访问一个代理层
- 在代理层可以监听 get set 做很多事情
    - 如 ES6 Proxy 实现 Vue3 响应式

5. 观察者模式

```
// 一个主题，一个观察者，主题变化之后触发观察者执行
btn.addEventListener('click', () => { ... })
```

6. 发布订阅模式

```
// 绑定
event.on('event-key', () => {
    // 事件1
})
event.on('event-key', () => {
    // 事件2
})

// 触发执行
event.emit('event-key')
```

注意：

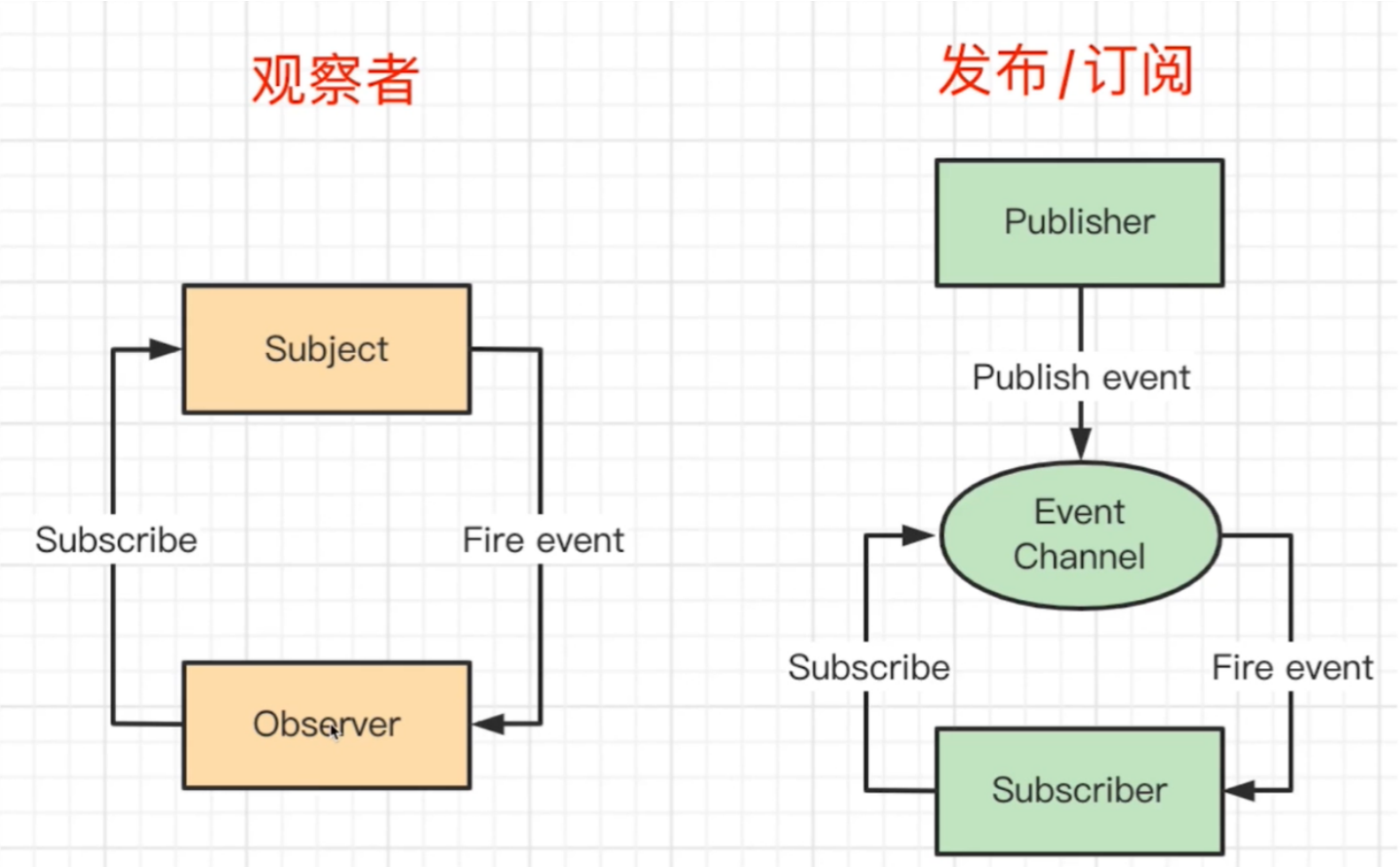
- 绑定的事件要记得解除，防止内存泄露

```
function fn1() { /* 事件1 */}
function fn2() { /* 事件2 */}

// mounted 时绑定
event.on('event-key', fn1)
event.on('event-key',fn2)

// beforeUnmount 时解绑
event.off('event-key', fn1)
event.off('event-key', fn2)
```

7. 装饰器模式

- 原功能不变，增加一些新功能（AOP面向切面编程）
- ES 和 TS 的 `Decorator` 语法

观察者模式和发布订阅模式的区别



1. 观察者模式

- Subject 和 Observer 直接绑定 没有中间媒介
  - 如 addEventListener 绑定的事件

2. 发布订阅模式

- Publisher 和 Observer 互不认识 需要中间媒介 Event channel
  - 如 EventBus 自定义事件

实际工作中，做过什么vue优化

1. `v-if` 和 `v-show`

- `v-if` 彻底销毁组件
- `v-show` 使用 css 隐藏组件
- 大部分情况下使用 `v-if` 更好，不要过度优化

2. `v-for` 使用 `key`
3. 使用 `computed` 缓存

```
export default {
    data() {
        return {
            msgList: [ ... ] // 消息列表
        }
    },
    computed: {
        // 未读消息的数量
        // 只要 list 不修改 computed就不会修改 做一个缓存
        unreadCount() {
            return this.msgList.filter(m => m.read === false).length
        }
    }
}
```

4. `keep-alive` 缓存组件

- 频繁切换的组件，如 `tabs`
- 不要乱用，缓存太多会占内存，而且不好 `debug`

5. 异步组件

- 针对体积较大的组件，如编辑器，复杂表格，复杂表单等
- 拆包，需要时异步加载，不需要时不加载
- 减少主包体积，首页会加载更快

6. 路由懒加载
7. 服务端渲染 SSR

- 可使用 Nuxt.js
- 按需优化，使用 SSR 的成本比较高

使用 vue 遇到过哪些坑

1. 内存泄漏

- 全局变量，全局事件，全局定时器
- 自定义事件

2. vue2 响应式缺陷（vue3 不再有）

- `data` 新增属性用 `Vue.set`
- `data` 删除属性用 `Vue.delete`
- 无法直接修改数据 `arr[index] = value`

3. 路由切换时 `scroll` 到顶部

- SPA 通病，不仅仅是 Vue
- 如 列表页 滚动到第二屏，点击进入详情页
- 再返回到列表页（此时组件重新渲染）就 `scroll` 到顶部

解决方案：

- 在列表页缓存数据和 `scrollTop` 的值
- 当再次返回列表页，渲染组件时，执行 `scrollTo(xx)`
- 终极方案： MPA + App WebView

如果统一监听 Vue 组件报错

1. `window.onerror`

- 全局监听所有JS错误
- 是JS级别的，识别不了Vue组件信息
- 捕捉一些 Vue 监听不到的错误信息

App.vue

```
// 不能监听到 try-catch
export default {
    mounted() {
        window.onerror = function(msg, sourse, line, column, error) {
            console.info('window.onerror-------', msg, sourse, line, column, error)
        }
        window.addEventListener('error', event => {
            console.info('window error------', event)
        })
    }
}
```

2. `errorCaptured` 生命周期

- 监听所有**下级**组件的错误
- 返回 `false` 会阻止向上传播

app.vue

```
export default {
    errorCaptured: (err, vm, info) => {
        console.info('errorCaptured-------', error, vm, info)
        // 防止重复捕获
        return false
    }
}
```

3. `errorHandler` 配置

- vue 全局错误监听，所有组件错误都会汇总到这里
- 但 `errorCaptured` 返回 `false` 不会传播到这里
- 在 `main.js` 配置

```
app.config.errorHandler = (error, vm, info) => {
    // 已经是全局监听，没有必要再触发 window.onerror() => 互斥，会阻止 window.onerror
    console.info('errorHandler-------', error, vm, info)
}
```

4. 异步错误(setTimeout)

- `errorCaptured` 和 `errorHandler` 无法监听
- 会在 `window.onerror` 监听

注意：

- 实际工作中，三者需要结合使用
- `errorCaptured` 监听一些重要，有风险组件的错误
- `window.onerror` 和 `errorHandler` 候补全局监听
- `Promise` 未处理的 `catch` 需要 `onunhandledrejection`

工作中遇到哪些项目难点，如何解决

- 遇到问题需要积累
- 描述问题：背景 + 现象 + 造成的影响
- 问题如何被解决：分析 + 解决
- 自己的成长：学到了什么 + 以后如何避免