10个常见的前端手写功能，你全都会吗？

js 复制代码

```js
function debounce(fn, delay) {
  let timer
  return function (...args) {
    if (timer) {
      clearTimeout(timer)
    }
    timer = setTimeout(() => {
      fn.apply(this, args)
    }, delay)
  }
}

// 测试
function task() {
  console.log('run task')
}
const debounceTask = debounce(task, 1000)
window.addEventListener('scroll', debounceTask)
```

js 复制代码

```js
function throttle(fn, delay) {
  let last = 0 // 上次触发时间
  return function (...args) {
    const now = Date.now()
    if (now - last > delay) {
      last = now
      fn.apply(this, args)
    }
  }
}

// 测试
function task() {
  console.log('run task')
}
const throttleTask = throttle(task, 1000)
window.addEventListener('scroll', throttleTask)
```

JSON 方法

js 复制代码

```js
// 不支持值为undefined、函数和循环引用的情况
const cloneObj = JSON.parse(JSON.stringify(obj))
```

递归拷贝

js 复制代码

```js
function deepClone(obj, cache = new WeakMap()) {
  if (obj === null || typeof obj !== 'object') return obj
  if (obj instanceof Date) return new Date(obj)
  if (obj instanceof RegExp) return new RegExp(obj)

  if (cache.has(obj)) return cache.get(obj) // 如果出现循环引用，则返回缓存的对象，防止递归进入死循环
  let cloneObj = new obj.constructor() // 使用对象所属的构造函数创建一个新对象
  cache.set(obj, cloneObj) // 缓存对象，用于循环引用的情况

  for (let key in obj) {
    if (obj.hasOwnProperty(key)) {
      cloneObj[key] = deepClone(obj[key], cache) // 递归拷贝
    }
  }
  return cloneObj
}

// 测试
const obj = { name: 'Jack', address: { x: 100, y: 200 } }
obj.a = obj // 循环引用

const newObj = deepClone(obj)
```

```
console.log(newObj.address === obj.address) // false
```

js 复制代码

```js
class MyPromise {
  constructor(executor) {
    this.status = 'pending' // 初始状态为等待
    this.value = null // 成功的值
    this.reason = null // 失败的原因
    this.onFulfilledCallbacks = [] // 成功的回调函数数组
    this.onRejectedCallbacks = [] // 失败的回调函数数组
    let resolve = value => {
      if (this.status === 'pending') {
        this.status = 'fulfilled'
        this.value = value;
        this.onFulfilledCallbacks.forEach(fn => fn()) // 调用成功的回调函数
      }
    }
    let reject = reason => {
      if (this.status === 'pending') {
        this.status = 'rejected'
        this.reason = reason
        this.onRejectedCallbacks.forEach(fn => fn()) // 调用失败的回调函数
      }
    };
    try {
      executor(resolve, reject)
    } catch (err) {
      reject(err)
    }
  }
  then(onFulfilled, onRejected) {
    return new MyPromise((resolve, reject) => {
      if (this.status === 'fulfilled') {
        setTimeout(() => {
          const x = onFulfilled(this.value);
          x instanceof MyPromise ? x.then(resolve, reject) : resolve(x)
        })
      }
      if (this.status === 'rejected') {
        setTimeout(() => {
          const x = onRejected(this.reason)
          x instanceof MyPromise ? x.then(resolve, reject) : resolve(x)
        })
        }
        if (this.status === 'pending') {
          this.onFulfilledCallbacks.push(() => { // 将成功的回调函数放入成功数组
            setTimeout(() => {
              const x = onFulfilled(this.value)
              x instanceof MyPromise ? x.then(resolve, reject) : resolve(x)
            })
          })
          this.onRejectedCallbacks.push(() => { // 将失败的回调函数放入失败数组
            setTimeout(() => {
              const x = onRejected(this.reason)
              x instanceof MyPromise ? x.then(resolve, reject) : resolve(x)
            })
          })
        }
      })
  }
}

// 测试
function p1() {
  return new MyPromise((resolve, reject) => {
    setTimeout(resolve, 1000, 1)
  })
}
function p2() {
  return new MyPromise((resolve, reject) => {
    setTimeout(resolve, 1000, 2)
  })
}
p1().then(res => {
  console.log(res) // 1
  return p2()
}).then(ret => {
  console.log(ret) // 2
})
```

js 复制代码

```
function limitRequest(urls = [], limit = 3) {
  return new Promise((resolve, reject) => {
    const len = urls.length
    let count = 0

    // 同时启动limit个任务
    while (limit > 0) {
      start()
      limit -= 1
    }

    function start() {
      const url = urls.shift() // 从数组中拿取第一个任务
      if (url) {
        axios.post(url).then(res => {
          // todo
        }).catch(err => {
          // todo
        }).finally(() => {
          if (count == len - 1) {
            // 最后一个任务完成
            resolve()
          } else {
            // 完成之后，启动下一个任务
            count++
            start()
          }
        })
      }
    }

  })
}

// 测试
limitRequest(['http://xxa', 'http://xxb', 'http://xxc', 'http://xxd', 'http://xxe'])
```

ES5 继承（寄生组合继承）

js 复制代码

```
function Parent(name) {
  this.name = name
}
Parent.prototype.eat = function () {
  console.log(this.name + ' is eating')
}
```

```js
function Child(name, age) {
  Parent.call(this, name)
  this.age = age
}
Child.prototype = Object.create(Parent.prototype)
Child.prototype.constructor = Child

// 测试
let xm = new Child('xiaoming', 12)
console.log(xm.name) // xiaoming
console.log(xm.age) // 12
xm.eat() // xiaoming is eating
```

ES6 继承

js 复制代码

```js
class Parent {
  constructor(name) {
    this.name = name
  }
  eat() {
    console.log(this.name + ' is eating')
  }
}

class Child extends Parent {
  constructor(name, age) {
    super(name)
    this.age = age
  }
}

// 测试
let xm = new Child('xiaoming', 12)
console.log(xm.name) // xiaoming
console.log(xm.age) // 12
xm.eat() // xiaoming is eating
```

sort 排序

js 复制代码

```js
// 对数字进行排序，简写
const arr = [3, 2, 4, 1, 5]
arr.sort((a, b) => a - b)
console.log(arr) // [1, 2, 3, 4, 5]

// 对字母进行排序，简写
const arr = ['b', 'c', 'a', 'e', 'd']
arr.sort()
console.log(arr) // ['a', 'b', 'c', 'd', 'e']
```

冒泡排序

js 复制代码

```js
function bubbleSort(arr) {
  let len = arr.length
  for (let i = 0; i < len - 1; i++) {
    // 从第一个元素开始，比较相邻的两个元素，前者大就交换位置
    for (let j = 0; j < len - 1 - i; j++) {
      if (arr[j] > arr[j + 1]) {
        let num = arr[j]
        arr[j] = arr[j + 1]
        arr[j + 1] = num
      }
    }
    // 每次遍历结束，都能找到一个最大值，放在数组最后
  }
  return arr
}

//测试
console.log(bubbleSort([2, 3, 1, 5, 4])) // [1, 2, 3, 4, 5]
```

Set 去重

js 复制代码

```js
const newArr = [...new Set(arr)]
// 或
const newArr = Array.from(new Set(arr))
```

indexOf 去重

js 复制代码

```js
const newArr = arr.filter((item, index) => arr.indexOf(item) === index)
```

URLSearchParams 方法

js 复制代码

```js
// 创建一个URLSearchParams实例
const urlSearchParams = new URLSearchParams(window.location.search);
// 把键值对列表转换为一个对象
const params = Object.fromEntries(urlSearchParams.entries());
```

split 方法

js 复制代码

```js
function getParams(url) {
  const res = {}
  if (url.includes('?')) {
    const str = url.split('?')[1]
    const arr = str.split('&')
    arr.forEach(item => {
      const key = item.split('=')[0]
      const val = item.split('=')[1]
      res[key] = decodeURIComponent(val) // 解码
    })
  }
  return res
}

// 测试
const user = getParams('http://www.baidu.com?user=%E9%98%BF%E9%A3%9E&age=16')
console.log(user) // { user: '阿飞', age: '16' }
```

js 复制代码

```
class EventEmitter {
  constructor() {
    this.cache = {}
  }

  on(name, fn) {
    if (this.cache[name]) {
      this.cache[name].push(fn)
    } else {
      this.cache[name] = [fn]
    }
  }

  off(name, fn) {
    const tasks = this.cache[name]
    if (tasks) {
      const index = tasks.findIndex((f) => f === fn || f.callback === fn)
      if (index >= 0) {
        tasks.splice(index, 1)
      }
    }
  }

  emit(name, once = false) {
    if (this.cache[name]) {
      // 创建副本，如果回调函数内继续注册相同事件，会造成死循环
      const tasks = this.cache[name].slice()
      for (let fn of tasks) {
        fn();
      }
      if (once) {
        delete this.cache[name]
      }
    }
  }
}

// 测试
const eventBus = new EventEmitter()
const task1 = () => { console.log('task1'); }
const task2 = () => { console.log('task2'); }

eventBus.on('task', task1)
eventBus.on('task', task2)
eventBus.off('task', task1)
setTimeout(() => {
  eventBus.emit('task') // task2
}, 1000)
```

以上就是工作或求职中最常见的手写功能，你是不是全都掌握了呢，欢迎在评论区交流。如果文章对你有所帮助，不要忘了点上宝贵的一赞！