前端面试必须掌握的手写题：进阶篇

请求并发控制

多次遇到的题目，而且有很多变种，主要就是同步改异步

```
function getUrlByFetch() {
  let idx = maxLoad;

  function getContention(index) {
    fetch(pics[index]).then(() => {
      idx++;
      if(idx < pics.length){
        getContention(idx);
      }
    });
  }
  function start() {
    for (let i = 0; i < maxLoad; i++) {
      getContention(i);
    }
  }
  start();
}
```

🔥带并发限制的promise异步调度器

上一题的其中一个变化

```
function taskPool() {
  this.tasks = [];
  this.pool = [];
  this.max = 2;
}

taskPool.prototype.addTask = function(task) {
  this.tasks.push(task);
  this.run();
}

taskPool.prototype.run = function() {
```

```
  if(this.tasks.length === 0) {
    return;
  }
  let min = Math.min(this.tasks.length, this.max - this.pool.length);
  for(let i = 0; i<min;i++) {
    const currTask = this.tasks.shift();
    this.pool.push(currTask);
    currTask().finally(() => {
      this.pool.splice(this.pool.indexOf(currTask), 1);
      this.run();
    })
  }
}
```

🔥🔥🔥实现lazy链式调用: person.eat().sleep(2).eat()

解法其实就是将所有的任务异步化，然后存到一个任务队列里

```
function Person() {
  this.queue = [];
  this.lock = false;
}

Person.prototype.eat = function () {
  this.queue.push(() => new Promise(resolve => { console.log('eat');
resolve(); }));
  // this.run();
  return this;
}

Person.prototype.sleep = function(time, flag) {
  this.queue.push(() => new Promise(resolve => {
    setTimeout(() => {
      console.log('sleep', flag);
      resolve();
    }, time * 1000)
  }));
  // this.run();
  return this;
}

Person.prototype.run = async function() {
  if(this.queue.length > 0 && !this.lock) {
    this.lock = true;
    const task = this.queue.shift();
```

```
    await task();
    this.lock = false;
    this.run();
  }
}

const person = new Person();
person.eat().sleep(1, '1').eat().sleep(3, '2').eat().run();
```

方法二

```
class Lazy {
    // 函数调用记录，私有属性
    #cbs = [];
    constructor(num) {
        // 当前操作后的结果
        this.res = num;
    }

    // output时，执行，私有属性
    #add(num) {
        this.res += num;
        console.log(this.res);
    }

    // output时，执行，私有属性
    #multipy(num) {
        this.res *= num;
        console.log(this.res)
    }

    add(num) {

        // 往记录器里面添加一个add函数的操作记录
        // 为了实现lazy的效果，所以没有直接记录操作后的结果，而是记录了一个函数
        this.#cbs.push({
            type: 'function',
            params: num,
            fn: this.#add
        })
        return this;
    }
    multipy(num) {

        // 和add函数同理
```

```
35          this.#cbs.push({
36              type: 'function',
37              params: num,
38              fn: this.#multipy
39          })
40          return this;
41      }
42      top (fn) {
43
44          // 记录需要执行的回调
45          this.#cbs.push({
46              type: 'callback',
47              fn: fn
48          })
49          return this;
50      }
51      delay (time) {
52
53          // 增加delay的记录
54          this.#cbs.push({
55              type: 'delay',
56
57              // 因为需要在output调用是再做到延迟time的效果，利用了Promise来实现
58              fn: () => {
59                  return new Promise(resolve => {
60                      console.log(`等待${time}ms`);
61                      setTimeout(() => {
62                          resolve();
63                      }, time);
64                  })
65              }
66          })
67          return this;
68      }
69
70      // 关键性函数，区分#cbs中每项的类型，然后执行不同的操作
71      // 因为需要用到延迟的效果，使用了async/await，所以output的返回值会是promise对象，
    无法链式调用
72      // 如果需实现output的链式调用，把for里面函数的调用全部放到promise.then的方式
73      async output() {
74          let cbs = this.#cbs;
75          for(let i = 0, l = cbs.length; i < l; i++) {
76              const cb = cbs[i];
77              let type = cb.type;
78              if (type === 'function') {
79                  cb.fn.call(this, cb.params);
80              }
```

```
            else if(type === 'callback') {
                cb.fn.call(this, this.res);
            }
            else if(type === 'delay') {
                await cb.fn();
            }
        }

        // 执行完成后清空 #cbs，下次再调用output的，只需再输出本轮的结果
        this.#cbs = [];
    }
}
function lazy(num) {
    return new Lazy(num);
}

const lazyFun = lazy(2).add(2).top(console.log).delay(1000).multipy(3)
console.log('start');
console.log('等待1000ms');
setTimeout(() => {
    lazyFun.output();
}, 1000);
```

🔥函数柯里化

毫无疑问，需要记忆

```
function curry(fn, args) {
  let length = fn.length;
  args = args || [];

  return function() {
    let subArgs = args.slice(0);
    subArgs = subArgs.concat(arguments);
    if(subArgs.length >= length) {
      return fn.apply(this, subArgs);
    } else {
      return curry.call(this, fn, subArgs);
    }
  }
}

// 更好理解的方式
function curry(func, arity = func.length) {
  function generateCurried(preArgs) {
```

```
    return function curried(nextArgs) {
      const args = [...preArgs, ...nextArgs];
      if(args.length >= arity) {
        return func(...args);
      } else {
        return generateCurried(args);
      }
    }
  }
  return generateCurried([]);
}
```

es6实现方式

```
// es6实现
function curry(fn, ...args) {
  return fn.length <= args.length ? fn(...args) : curry.bind(null, fn,
...args);
}

```

lazy-load实现

img标签默认支持懒加载只需要添加属性 loading="lazy"，然后如果不用这个属性，想通过事件监听的方式来实现的话，也可以使用IntersectionObserver来实现，性能上会比监听scroll好很多

```
const imgs = document.getElementsByTagName('img');
const viewHeight = window.innerHeight || document.documentElement.clientHeight;

let num = 0;

function lazyLoad() {
  for (let i = 0; i < imgs.length; i++) {
    let distance = viewHeight - imgs[i].getBoundingClientRect().top;
    if(distance >= 0) {
      imgs[i].src = imgs[i].getAttribute('data-src');
      num = i+1;
    }
  }
}
window.addEventListener('scroll', lazyLoad, false);
```

实现简单的虚拟dom

给出如下虚拟dom的数据结构，如何实现简单的虚拟dom，渲染到目标dom树

```
// 样例数据
let demoNode = ({
    tagName: 'ul',
    props: {'class': 'list'},
    children: [
        ({tagName: 'li', children: ['douyin']}),
        ({tagName: 'li', children: ['toutiao']})
    ]
});
```

构建一个render函数，将demoNode对象渲染为以下dom

```
<ul class="list">
  <li>douyin</li>
  <li>toutiao</li>
</ul>
```

通过遍历，逐个节点地创建真实DOM节点

```
function Element({tagName, props, children}){
    // 判断必须使用构造函数
    if(!(this instanceof Element)){
        return new Element({tagName, props, children})
    }
    this.tagName = tagName;
    this.props = props || {};
    this.children = children || [];
}

Element.prototype.render = function(){
    var el = document.createElement(this.tagName),
        props = this.props,
        propName,
        propValue;
    for(propName in props){
        propValue = props[propName];
        el.setAttribute(propName, propValue);
    }
```

```
    this.children.forEach(function(child){
        var childEl = null;
        if(child instanceof Element){
            childEl = child.render();
        }else{
            childEl = document.createTextNode(child);
        }
        el.appendChild(childEl);
    });
    return el;
};

// 执行
var elem = Element({
    tagName: 'ul',
    props: {'class': 'list'},
    children: [
        Element({tagName: 'li', children: ['item1']}),
        Element({tagName: 'li', children: ['item2']})
    ]
});
document.querySelector('body').appendChild(elem.render());
```

实现SWR 机制

SWR 这个名字来自于 stale-while-revalidate：一种由 HTTP RFC 5861 推广的 **HTTP 缓存失效策略**

```
const cache = new Map();

async function swr(cacheKey, fetcher, cacheTime) {
  let data = cache.get(cacheKey) || { value: null, time: 0, promise: null };
  cache.set(cacheKey, data);

  // 是否过期
  const isStaled = Date.now() - data.time > cacheTime;
  if (isStaled && !data.promise) {
    data.promise = fetcher()
      .then((val) => {
        data.value = val;
        data.time = Date.now();
      })
      .catch((err) => {
        console.log(err);
      })
      .finally(() => {
```

```
        data.promise = null;
      });
  }

  if (data.promise && !data.value) await data.promise;
  return data.value;
}

const data = await fetcher();
const data = await swr('cache-key', fetcher, 3000);
```

实现一个只执行一次的函数

```
// 闭包
function once(fn) {
  let called = false;
  return function _once() {
    if (called) {
      return _once.value;
    }
    called = true;
    _once.value = fn.apply(this, arguments);
  }
}

//ES6 的元编程 Reflect API 将其定义为函数的行为
Reflect.defineProperty(Function.prototype, 'once', {
  value () {
    return once(this);
  },
  configurable: true,
})

```

LRU 算法实现

LRU（Least recently used，最近最少使用）算法根据数据的历史访问记录来进行**淘汰**数据，其核心思想是“如果数据最近被访问过，那么将来被访问的几率也更高”。

```
class LRUCahe {
  constructor(capacity) {
    this.cache = new Map();
    this.capacity = capacity;
```

```
  }

  get(key) {
    if (this.cache.has(key)) {
      const temp = this.cache.get(key);
      this.cache.delete(key);
      this.cache.set(key, temp);
      return temp;
    }
    return undefined;
  }

  set(key, value) {
    if (this.cache.has(key)) {
      this.cache.delete(key);
    } else if (this.cache.size >= this.capacity) {
      // map.keys() 会返回 Iterator 对象
      this.cache.delete(this.cache.keys().next().value);
    }
    this.cache.set(key, value);
  }
}
```

🔥发布-订阅

发布者不直接触及到订阅者、而是由统一的第三方来完成实际的通信的操作，叫做发布-订阅模式。

```
class EventEmitter {
  constructor() {
    // handlers是一个map，用于存储事件与回调之间的对应关系
    this.handlers = {}
  }

  // on方法用于安装事件监听器，它接受目标事件名和回调函数作为参数
  on(eventName, cb) {
    // 先检查一下目标事件名有没有对应的监听函数队列
    if (!this.handlers[eventName]) {
      // 如果没有，那么首先初始化一个监听函数队列
      this.handlers[eventName] = []
    }

    // 把回调函数推入目标事件的监听函数队列里去
    this.handlers[eventName].push(cb)
  }

```

```
  // emit方法用于触发目标事件，它接受事件名和监听函数入参作为参数
  emit(eventName, ...args) {
    // 检查目标事件是否有监听函数队列
    if (this.handlers[eventName]) {
      // 这里需要对 this.handlers[eventName] 做一次浅拷贝，主要目的是为了避免通过 once 安装的监听器在移除的过程中出现顺序问题
      const handlers = this.handlers[eventName].slice()
      // 如果有，则逐个调用队列里的回调函数
      handlers.forEach((callback) => {
        callback(...args)
      })
    }
  }

  // 移除某个事件回调队列里的指定回调函数
  off(eventName, cb) {
    const callbacks = this.handlers[eventName]
    const index = callbacks.indexOf(cb)
    if (index !== -1) {
      callbacks.splice(index, 1)
    }
  }

  // 为事件注册单次监听器
  once(eventName, cb) {
    // 对回调函数进行包装，使其执行完毕自动被移除
    const wrapper = (...args) => {
      cb(...args)
      this.off(eventName, wrapper)
    }
    this.on(eventName, wrapper)
  }
}
```

观察者模式

```
const queuedObservers = new Set();

const observe = fn => queuedObservers.add(fn);
const observable = obj => new Proxy(obj, {set});

function set(target, key, value, receiver) {
  const result = Reflect.set(target, key, value, receiver);
  queuedObservers.forEach(observer => observer());
  return result;
```

```
}
```

单例模式

核心要点: 用闭包和Proxy属性拦截

```
function getSingleInstance(func) {
  let instance;
  let handler = {
    construct(target, args) {
      if(!instance) instance = Reflect.construct(func, args);
      return instance;
    }
  }
  return new Proxy(func, handler);
}

```

洋葱圈模型compose函数

```
function compose(middleware) {
  return function(context, next) {
    let index = -1;
    return dispatch(0);
    function dispatch(i) {
      // 不允许执行多次中间件
      if(i <= index) return Promise.reject(new Error('next() called multiple times'));
      // 更新游标
      index = i;
      let fn = middle[i];
      // 这个next是外部的回调
      if(i === middle.length) fn = next;
      if(!fn) return Promsie.resolve();
      try{
        return Promise.resove(fn(context, dispatch.bind(null, i+1)));
      }catch(err){
        return Promise.reject(err);
      }
    }
  }
}
```

总结

当你看到这里的时候，几乎前端面试中常见的手写题目基本都覆盖到了，对于社招的场景下，其实手写题的题目是越来越务实的，尤其是真的有hc的情况下，一般出一些常见的场景题的可能性更大，所以最好理解➕记忆，