**01.数组扁平化**

数组扁平化是指将一个多维数组变为一个一维数组

const arr = [1, [2, [3, [4, 5]]], 6];

// => [1, 2, 3, 4, 5, 6]

**方法一：使用flat()**

const res1 = arr.flat(Infinity);

**方法二：利用正则**

const res2 = JSON.stringify(arr).replace(/\[|\]/g, '').split(',');

但数据类型都会变为字符串

**方法三：正则改良版本**

const res3 = JSON.parse('[' + JSON.stringify(arr).replace(/\[|\]/g, '') + ']');

**方法四：使用reduce**

const flatten = arr => {

return arr.reduce((pre, cur) => {

return pre.concat(Array.isArray(cur) ? flatten(cur) : cur);

}, [])

}

const res4 = flatten(arr);

**方法五：函数递归**

const res5 = [];

const fn = arr => {

for (let i = 0; i < arr.length; i++) {

if (Array.isArray(arr[i])) {

fn(arr[i]);

} else {

res5.push(arr[i]);

}

}

}

fn(arr);

**02.数组去重**

const arr = [1, 1, '1', 17, true, true, false, false, 'true', 'a', {}, {}];

// => [1, '1', 17, true, false, 'true', 'a', {}, {}]

**方法一：利用Set**

const res1 = Array.from(new Set(arr));

**方法二：两层for循环+splice**

const unique1 = arr => {

let len = arr.length;

for (let i = 0; i < len; i++) {

for (let j = i + 1; j < len; j++) {

if (arr[i] === arr[j]) {

arr.splice(j, 1);

// 每删除一个树，j--保证j的值经过自加后不变。同时，len--，减少循环次数提升性能

len--;

j--;

}

}

}

return arr;

}

**方法三：利用indexOf**

const unique2 = arr => {

const res = [];

for (let i = 0; i < arr.length; i++) {

if (res.indexOf(arr[i]) === -1) res.push(arr[i]);

}

return res;

}

当然也可以用include、filter，思路大同小异。

**方法四：利用include**

const unique3 = arr => {

const res = [];

for (let i = 0; i < arr.length; i++) {

if (!res.includes(arr[i])) res.push(arr[i]);

}

return res;

}

**方法五：利用filter**

const unique4 = arr => {

return arr.filter((item, index) => {

return arr.indexOf(item) === index;

});

}

**方法六：利用Map**

const unique5 = arr => {

const map = new Map();

const res = [];

for (let i = 0; i < arr.length; i++) {

if (!map.has(arr[i])) {

map.set(arr[i], true)

res.push(arr[i]);

}

}

return res;

}

**03.类数组转化为数组**

类数组是具有**length**属性，但不具有数组原型上的方法。常见的类数组有**arguments**、DOM操作方法返回的结果。

**方法一：Array.from**

Array.from(document.querySelectorAll('div'))

**方法二：Array.prototype.slice.call()**

Array.prototype.slice.call(document.querySelectorAll('div'))

**方法三：扩展运算符**

[...document.querySelectorAll('div')]

方法四：利用concat

Array.prototype.concat.apply([], document.querySelectorAll('div'));

**04.Array.prototype.filter()**



Array.prototype.filter = function(callback, thisArg) {

if (this == undefined) {

throw new TypeError('this is null or not undefined');

}

if (typeof callback !== 'function') {

throw new TypeError(callback + 'is not a function');

}

const res = [];

// 让O成为回调函数的对象传递（强制转换对象）

const O = Object(this);

// >>>0 保证len为number，且为正整数

const len = O.length >>> 0;

for (let i = 0; i < len; i++) {

// 检查i是否在O的属性（会检查原型链）

if (i in O) {

// 回调函数调用传参

if (callback.call(thisArg, O[i], i, O)) {

res.push(O[i]);

}

}

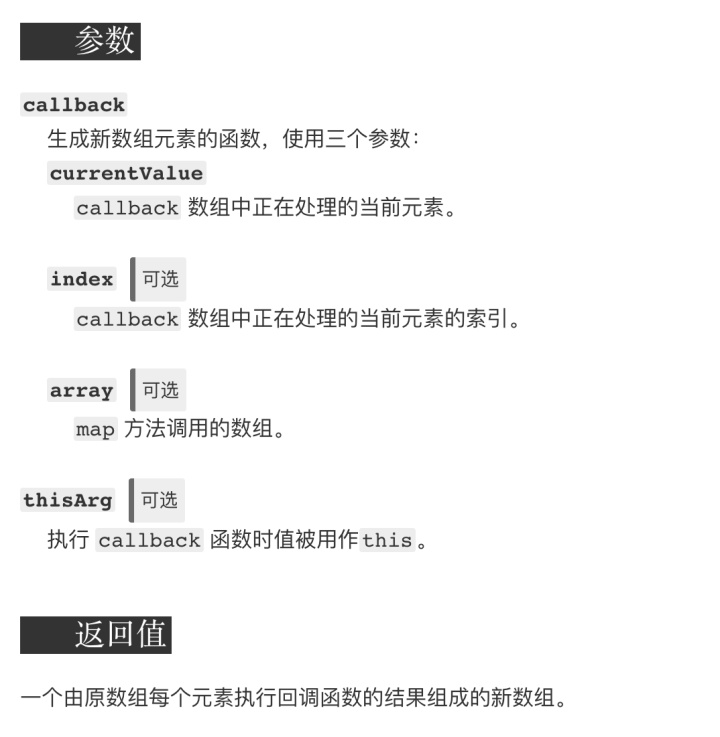
}

return res;

}

对于>>>0有疑问的：解释>>>0的作用

**05.Array.prototype.map()**



Array.prototype.map = function(callback, thisArg) {

if (this == undefined) {

throw new TypeError('this is null or not defined');

}

if (typeof callback !== 'function') {

throw new TypeError(callback + ' is not a function');

}

const res = [];

// 同理

const O = Object(this);

const len = O.length >>> 0;

for (let i = 0; i < len; i++) {

if (i in O) {

// 调用回调函数并传入新数组

res[i] = callback.call(thisArg, O[i], i, this);

}

}

return res;

}

**06.Array.prototype.forEach()**



forEach跟map类似，唯一不同的是forEach是没有返回值的。

Array.prototype.forEach = function(callback, thisArg) {

if (this == null) {

throw new TypeError('this is null or not defined');

}

if (typeof callback !== "function") {

throw new TypeError(callback + ' is not a function');

}

const O = Object(this);

const len = O.length >>> 0;

let k = 0;

while (k < len) {

if (k in O) {

callback.call(thisArg, O[k], k, O);

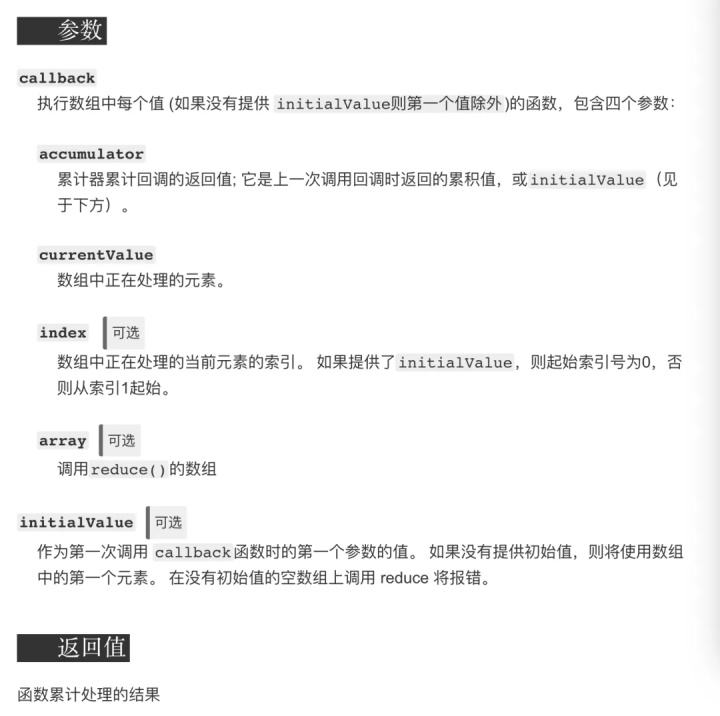
}

k++;

}

}

**07.Array.prototype.reduce()**



Array.prototype.reduce = function(callback, initialValue) {

if (this == undefined) {

throw new TypeError('this is null or not defined');

}

if (typeof callback !== 'function') {

throw new TypeError(callbackfn + ' is not a function');

}

const O = Object(this);

const len = this.length >>> 0;

let accumulator = initialValue;

let k = 0;

// 如果第二个参数为undefined的情况下

// 则数组的第一个有效值作为累加器的初始值

if (accumulator === undefined) {

while (k < len && !(k in O)) {

k++;

}

// 如果超出数组界限还没有找到累加器的初始值，则TypeError

if (k >= len) {

throw new TypeError('Reduce of empty array with no initial value');

}

accumulator = O[k++];

}

while (k < len) {

if (k in O) {

accumulator = callback.call(undefined, accumulator, O[k], k, O);

}

k++;

}

return accumulator;

}

**08.Function.prototype.apply()**

第一个参数是绑定的this，默认为window，第二个参数是数组或类数组

Function.prototype.apply = function(context = window, args) {

if (typeof this !== 'function') {

throw new TypeError('Type Error');

}

const fn = Symbol('fn');

context[fn] = this;

const res = context[fn](...args);

delete context[fn];

return res;

}

**09.Function.prototype.call**

于call唯一不同的是，call()方法接受的是一个参数列表

Function.prototype.call = function(context = window, ...args) {

if (typeof this !== 'function') {

throw new TypeError('Type Error');

}

const fn = Symbol('fn');

context[fn] = this;

const res = this[fn](...args);

delete this.fn;

return res;

}

**10.Function.prototype.bind**

Function.prototype.bind = function(context, ...args) {

if (typeof this !== 'function') {

throw new Error("Type Error");

}

// 保存this的值

var self = this;

return function F() {

// 考虑new的情况

if(this instanceof F) {

return new self(...args, ...arguments)

}

return self.apply(context, [...args, ...arguments])

}

}

**11.debounce（防抖）**

触发高频时间后n秒内函数只会执行一次,如果n秒内高频时间再次触发,则重新计算时间。

const debounce = (fn, time) => {

let timeout = null;

return function() {

clearTimeout(timeout)

timeout = setTimeout(() => {

fn.apply(this, arguments);

}, time);

}

};

防抖常应用于用户进行搜索输入节约请求资源，window触发resize事件时进行防抖只触发一次。

**12.throttle（节流）**

高频时间触发,但n秒内只会执行一次,所以节流会稀释函数的执行频率。

const throttle = (fn, time) => {

let flag = true;

return function() {

if (!flag) return;

flag = false;

setTimeout(() => {

fn.apply(this, arguments);

flag = true;

}, time);

}

}

节流常应用于鼠标不断点击触发、监听滚动事件。

**13.函数珂里化**

指的是将一个接受多个参数的函数 变为 接受一个参数返回一个函数的固定形式，这样便于再次调用，例如f(1)(2)

经典面试题：实现add(1)(2)(3)(4)=10; 、 add(1)(1,2,3)(2)=9;

function add() {

const \_args = [...arguments];

function fn() {

\_args.push(...arguments);

return fn;

}

fn.toString = function() {

return \_args.reduce((sum, cur) => sum + cur);

}

return fn;

}

**14.模拟new操作**

3个步骤：

1. 以ctor.prototype为原型创建一个对象。
2. 执行构造函数并将this绑定到新创建的对象上。
3. 判断构造函数执行返回的结果是否是引用数据类型，若是则返回构造函数执行的结果，否则返回创建的对象。

function newOperator(ctor, ...args) {

if (typeof ctor !== 'function') {

throw new TypeError('Type Error');

}

const obj = Object.create(ctor.prototype);

const res = ctor.apply(obj, args);

const isObject = typeof res === 'object' && res !== null;

const isFunction = typeof res === 'function';

return isObject || isFunction ? res : obj;

}

**15.instanceof**

instanceof运算符用于检测构造函数的prototype属性是否出现在某个实例对象的原型链上。

const myInstanceof = (left, right) => {

// 基本数据类型都返回false

if (typeof left !== 'object' || left === null) return false;

let proto = Object.getPrototypeOf(left);

while (true) {

if (proto === null) return false;

if (proto === right.prototype) return true;

proto = Object.getPrototypeOf(proto);

}

}

**16.原型继承**

这里只写寄生组合继承了，中间还有几个演变过来的继承但都有一些缺陷

function Parent5) {

this.name = 'parent';

}

function Child() {

Parent.call(this);

this.type = 'children';

}

Child.prototype = Object.create(Parent.prototype);

Child.prototype.constructor = Child;

**17.Object.is**

Object.is解决的主要是这两个问题：

+0 === -0 // true

NaN === NaN // false

const is= (x, y) => {

if (x === y) {

// +0和-0应该不相等

return x !== 0 || y !== 0 || 1/x === 1/y;

} else {

return x !== x && y !== y;

}

}

**18.Object.assign**

Object.assign()方法用于将所有可枚举属性的值从一个或多个源对象复制到目标对象。它将返回目标对象（请注意这个操作是浅拷贝）

Object.defineProperty(Object, 'assign', {

value: function(target, ...args) {

if (target == null) {

return new TypeError('Cannot convert undefined or null to object');

}

// 目标对象需要统一是引用数据类型，若不是会自动转换

const to = Object(target);

for (let i = 0; i < args.length; i++) {

// 每一个源对象

const nextSource = args[i];

if (nextSource !== null) {

// 使用for...in和hasOwnProperty双重判断，确保只拿到本身的属性、方法（不包含继承的）

for (const nextKey in nextSource) {

if (Object.prototype.hasOwnProperty.call(nextSource, nextKey)) {

to[nextKey] = nextSource[nextKey];

}

}

}

}

return to;

},

// 不可枚举

enumerable: false,

writable: true,

configurable: true,

})

**19.深拷贝**

递归的完整版本（考虑到了Symbol属性）：

const cloneDeep1 = (target, hash = new WeakMap()) => {

// 对于传入参数处理

if (typeof target !== 'object' || target === null) {

return target;

}

// 哈希表中存在直接返回

if (hash.has(target)) return hash.get(target);

const cloneTarget = Array.isArray(target) ? [] : {};

hash.set(target, cloneTarget);

// 针对Symbol属性

const symKeys = Object.getOwnPropertySymbols(target);

if (symKeys.length) {

symKeys.forEach(symKey => {

if (typeof target[symKey] === 'object' && target[symKey] !== null) {

cloneTarget[symKey] = cloneDeep1(target[symKey]);

} else {

cloneTarget[symKey] = target[symKey];

}

})

}

for (const i in target) {

if (Object.prototype.hasOwnProperty.call(target, i)) {

cloneTarget[i] =

typeof target[i] === 'object' && target[i] !== null

? cloneDeep1(target[i], hash)

: target[i];

}

}

return cloneTarget;

}

**20.Promise**

实现思路：Promise源码实现

const PENDING = 'PENDING'; // 进行中

const FULFILLED = 'FULFILLED'; // 已成功

const REJECTED = 'REJECTED'; // 已失败

class Promise {

constructor(exector) {

// 初始化状态

this.status = PENDING;

// 将成功、失败结果放在this上，便于then、catch访问

this.value = undefined;

this.reason = undefined;

// 成功态回调函数队列

this.onFulfilledCallbacks = [];

// 失败态回调函数队列

this.onRejectedCallbacks = [];

const resolve = value => {

// 只有进行中状态才能更改状态

if (this.status === PENDING) {

this.status = FULFILLED;

this.value = value;

// 成功态函数依次执行

this.onFulfilledCallbacks.forEach(fn => fn(this.value));

}

}

const reject = reason => {

// 只有进行中状态才能更改状态

if (this.status === PENDING) {

this.status = REJECTED;

this.reason = reason;

// 失败态函数依次执行

this.onRejectedCallbacks.forEach(fn => fn(this.reason))

}

}

try {

// 立即执行executor

// 把内部的resolve和reject传入executor，用户可调用resolve和reject

exector(resolve, reject);

} catch(e) {

// executor执行出错，将错误内容reject抛出去

reject(e);

}

}

then(onFulfilled, onRejected) {

onFulfilled = typeof onFulfilled === 'function' ? onFulfilled : value => value;

onRejected = typeof onRejected === 'function'? onRejected:

reason => { throw new Error(reason instanceof Error ? reason.message:reason) }

// 保存this

const self = this;

return new Promise((resolve, reject) => {

if (self.status === PENDING) {

self.onFulfilledCallbacks.push(() => {

// try捕获错误

try {

// 模拟微任务

setTimeout(() => {

const result = onFulfilled(self.value);

// 分两种情况：

// 1. 回调函数返回值是Promise，执行then操作

// 2. 如果不是Promise，调用新Promise的resolve函数

result instanceof Promise ? result.then(resolve, reject) : resolve(result);

})

} catch(e) {

reject(e);

}

});

self.onRejectedCallbacks.push(() => {

// 以下同理

try {

setTimeout(() => {

const result = onRejected(self.reason);

// 不同点：此时是reject

result instanceof Promise ? result.then(resolve, reject) : reject(result);

})

} catch(e) {

reject(e);

}

})

} else if (self.status === FULFILLED) {

try {

setTimeout(() => {

const result = onFulfilled(self.value);

result instanceof Promise ? result.then(resolve, reject) : resolve(result);

});

} catch(e) {

reject(e);

}

} else if (self.status === REJECTED){

try {

setTimeout(() => {

const result = onRejected(self.reason);

result instanceof Promise ? result.then(resolve, reject) : reject(result);

})

} catch(e) {

reject(e);

}

}

});

}

catch(onRejected) {

return this.then(null, onRejected);

}

static resolve(value) {

if (value instanceof Promise) {

// 如果是Promise实例，直接返回

return value;

} else {

// 如果不是Promise实例，返回一个新的Promise对象，状态为FULFILLED

return new Promise((resolve, reject) => resolve(value));

}

}

static reject(reason) {

return new Promise((resolve, reject) => {

reject(reason);

})

}

}

**21.Promise.all**

Promise.all是支持链式调用的，本质上就是返回了一个Promise实例，通过resolve和reject来改变实例状态。

Promise.myAll = function(promiseArr) {

return new Promise((resolve, reject) => {

const ans = [];

let index = 0;

for (let i = 0; i < promiseArr.length; i++) {

promiseArr[i]

.then(res => {

ans[i] = res;

index++;

if (index === promiseArr.length) {

resolve(ans);

}

})

.catch(err => reject(err));

}

})

}

**22.Promise.race**

Promise.race = function(promiseArr) {

return new Promise((resolve, reject) => {

promiseArr.forEach(p => {

// 如果不是Promise实例需要转化为Promise实例

Promise.resolve(p).then(

val => resolve(val),

err => reject(err),

)

})

})

}

**23.Promise并行限制**

就是实现有并行限制的Promise调度器问题。

详细实现思路：某条高频面试原题：实现有并行限制的Promise调度器

class Scheduler {

constructor() {

this.queue = [];

this.maxCount = 2;

this.runCounts = 0;

}

add(promiseCreator) {

this.queue.push(promiseCreator);

}

taskStart() {

for (let i = 0; i < this.maxCount; i++) {

this.request();

}

}

request() {

if (!this.queue || !this.queue.length || this.runCounts >= this.maxCount) {

return;

}

this.runCounts++;

this.queue.shift()().then(() => {

this.runCounts--;

this.request();

});

}

}

const timeout = time => new Promise(resolve => {

setTimeout(resolve, time);

})

const scheduler = new Scheduler();

const addTask = (time,order) => {

scheduler.add(() => timeout(time).then(()=>console.log(order)))

}

addTask(1000, '1');

addTask(500, '2');

addTask(300, '3');

addTask(400, '4');

scheduler.taskStart()

// 2

// 3

// 1

// 4

**24.JSONP**

script标签不遵循同源协议，可以用来进行**跨域请求**，优点就是兼容性好但仅限于GET请求

const jsonp = ({ url, params, callbackName }) => {

const generateUrl = () => {

let dataSrc = '';

for (let key in params) {

if (Object.prototype.hasOwnProperty.call(params, key)) {

dataSrc += `${key}=${params[key]}&`;

}

}

dataSrc += `callback=${callbackName}`;

return `${url}?${dataSrc}`;

}

return new Promise((resolve, reject) => {

const scriptEle = document.createElement('script');

scriptEle.src = generateUrl();

document.body.appendChild(scriptEle);

window[callbackName] = data => {

resolve(data);

document.removeChild(scriptEle);

}

})

}

**25.AJAX**

const getJSON = function(url) {

return new Promise((resolve, reject) => {

const xhr = XMLHttpRequest ? new XMLHttpRequest() : new ActiveXObject('Mscrosoft.XMLHttp');

xhr.open('GET', url, false);

xhr.setRequestHeader('Accept', 'application/json');

xhr.onreadystatechange = function() {

if (xhr.readyState !== 4) return;

if (xhr.status === 200 || xhr.status === 304) {

resolve(xhr.responseText);

} else {

reject(new Error(xhr.responseText));

}

}

xhr.send();

})

}

**26.event模块**

实现node中回调函数的机制，node中回调函数其实是内部使用了**观察者模式**。

观察者模式：定义了对象间一种一对多的依赖关系，当目标对象Subject发生改变时，所有依赖它的对象Observer都会得到通知。

function EventEmitter() {

this.events = new Map();

}

// 需要实现的一些方法：

// addListener、removeListener、once、removeAllListeners、emit

// 模拟实现addlistener方法

const wrapCallback = (fn, once = false) => ({ callback: fn, once });

EventEmitter.prototype.addListener = function(type, fn, once = false) {

const hanlder = this.events.get(type);

if (!hanlder) {

// 没有type绑定事件

this.events.set(type, wrapCallback(fn, once));

} else if (hanlder && typeof hanlder.callback === 'function') {

// 目前type事件只有一个回调

this.events.set(type, [hanlder, wrapCallback(fn, once)]);

} else {

// 目前type事件数>=2

hanlder.push(wrapCallback(fn, once));

}

}

// 模拟实现removeListener

EventEmitter.prototype.removeListener = function(type, listener) {

const hanlder = this.events.get(type);

if (!hanlder) return;

if (!Array.isArray(this.events)) {

if (hanlder.callback === listener.callback) this.events.delete(type);

else return;

}

for (let i = 0; i < hanlder.length; i++) {

const item = hanlder[i];

if (item.callback === listener.callback) {

hanlder.splice(i, 1);

i--;

if (hanlder.length === 1) {

this.events.set(type, hanlder[0]);

}

}

}

}

// 模拟实现once方法

EventEmitter.prototype.once = function(type, listener) {

this.addListener(type, listener, true);

}

// 模拟实现emit方法

EventEmitter.prototype.emit = function(type, ...args) {

const hanlder = this.events.get(type);

if (!hanlder) return;

if (Array.isArray(hanlder)) {

hanlder.forEach(item => {

item.callback.apply(this, args);

if (item.once) {

this.removeListener(type, item);

}

})

} else {

hanlder.callback.apply(this, args);

if (hanlder.once) {

this.events.delete(type);

}

}

return true;

}

EventEmitter.prototype.removeAllListeners = function(type) {

const hanlder = this.events.get(type);

if (!hanlder) return;

this.events.delete(type);

}

**27.图片懒加载**

可以给img标签统一自定义属性data-src='default.png'，当检测到图片出现在窗口之后再补充**src**属性，此时才会进行图片资源加载。

function lazyload() {

const imgs = document.getElementsByTagName('img');

const len = imgs.length;

// 视口的高度

const viewHeight = document.documentElement.clientHeight;

// 滚动条高度

const scrollHeight = document.documentElement.scrollTop || document.body.scrollTop;

for (let i = 0; i < len; i++) {

const offsetHeight = imgs[i].offsetTop;

if (offsetHeight < viewHeight + scrollHeight) {

const src = imgs[i].dataset.src;

imgs[i].src = src;

}

}

}

// 可以使用节流优化一下

window.addEventListener('scroll', lazyload);

**28.滚动加载**

原理就是监听页面滚动事件，**分析clientHeight**、**scrollTop**、**scrollHeight**三者的属性关系。

window.addEventListener('scroll', function() {

const clientHeight = document.documentElement.clientHeight;

const scrollTop = document.documentElement.scrollTop;

const scrollHeight = document.documentElement.scrollHeight;

if (clientHeight + scrollTop >= scrollHeight) {

// 检测到滚动至页面底部，进行后续操作

// ...

}

}, false);

一个Demo：页面滚动加载的Demo

**29.渲染几万条数据不卡住页面**

渲染大数据时，合理使用**createDocumentFragment**和**requestAnimationFrame**，将操作切分为一小段一小段执行。

setTimeout(() => {

// 插入十万条数据

const total = 100000;

// 一次插入的数据

const once = 20;

// 插入数据需要的次数

const loopCount = Math.ceil(total / once);

let countOfRender = 0;

const ul = document.querySelector('ul');

// 添加数据的方法

function add() {

const fragment = document.createDocumentFragment();

for(let i = 0; i < once; i++) {

const li = document.createElement('li');

li.innerText = Math.floor(Math.random() \* total);

fragment.appendChild(li);

}

ul.appendChild(fragment);

countOfRender += 1;

loop();

}

function loop() {

if(countOfRender < loopCount) {

window.requestAnimationFrame(add);

}

}

loop();

}, 0)

**30.打印出当前网页使用了多少种HTML元素**

一行代码可以解决：

const fn = () => {

return [...new Set([...document.querySelectorAll('\*')].map(el => el.tagName))].length;

}

值得注意的是：DOM操作返回的是**类数组**，需要转换为数组之后才可以调用数组的方法。

**31.将VirtualDom转化为真实DOM结构**

这是当前SPA应用的核心概念之一

// vnode结构：

// {

// tag,

// attrs,

// children,

// }

//Virtual DOM => DOM

function render(vnode, container) {

container.appendChild(\_render(vnode));

}

function \_render(vnode) {

// 如果是数字类型转化为字符串

if (typeof vnode === 'number') {

vnode = String(vnode);

}

// 字符串类型直接就是文本节点

if (typeof vnode === 'string') {

return document.createTextNode(vnode);

}

// 普通DOM

const dom = document.createElement(vnode.tag);

if (vnode.attrs) {

// 遍历属性

Object.keys(vnode.attrs).forEach(key => {

const value = vnode.attrs[key];

dom.setAttribute(key, value);

})

}

// 子数组进行递归操作

vnode.children.forEach(child => render(child, dom));

return dom;

}

**32.字符串解析问题**

var a = {

b: 123,

c: '456',

e: '789',

}

var str=`a{a.b}aa{a.c}aa {a.d}aaaa`;

// => 'a123aa456aa {a.d}aaaa'

实现函数使得将str字符串中的{}内的变量替换，如果属性不存在保持原样（比如{a.d}）

类似于模版字符串，但有一点出入，实际上原理大差不差

const fn1 = (str, obj) => {

let res = '';

// 标志位，标志前面是否有{

let flag = false;

let start;

for (let i = 0; i < str.length; i++) {

if (str[i] === '{') {

flag = true;

start = i + 1;

continue;

}

if (!flag) res += str[i];

else {

if (str[i] === '}') {

flag = false;

res += match(str.slice(start, i), obj);

}

}

}

return res;

}

// 对象匹配操作

const match = (str, obj) => {

const keys = str.split('.').slice(1);

let index = 0;

let o = obj;

while (index < keys.length) {

const key = keys[index];

if (!o[key]) {

return `{${str}}`;

} else {

o = o[key];

}

index++;

}

return o;

}