作者：不凡学院  
链接：https://zhuanlan.zhihu.com/p/108289604  
来源：知乎  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

**前言**

能够手撕各种JavaScript原生函数，可以说是进大厂必备！同时对JavaScript源码的学习和实现也能帮助我们快速扎实地提升自己的前端编程能力。

最近很多人和我一样在积极地准备前端面试笔试，所以就整理了一些前端面试笔试中非常容易被问到的原生函数实现和各种前端原理实现，其中[部分源码戳这里](https://link.zhihu.com/?target=https%3A//github.com/Cloudyttt/Algorithm-Design/tree/master/%25E6%2589%258B%25E6%2592%2595JavaScript%25E6%25BA%2590%25E7%25A0%2581)。

看完觉得对你有帮助劳烦点个赞鼓励鼓励哈！

**实现一个new操作符**

我们首先知道new做了什么：

1. 创建一个空的简单JavaScript对象（即{}）；
2. 链接该对象（即设置该对象的构造函数）到另一个对象 ；
3. 将步骤（1）新创建的对象作为this的上下文 ；
4. 如果该函数没有返回对象，则返回this。

知道new做了什么，接下来我们就来实现它

function create(Con, ...args){

// 创建一个空的对象

this.obj = {};

// 将空对象指向构造函数的原型链

Object.setPrototypeOf(this.obj, Con.prototype);

// obj绑定到构造函数上，便可以访问构造函数中的属性，即this.obj.Con(args)

let result = Con.apply(this.obj, args);

// 如果返回的result是一个对象则返回

// new方法失效，否则返回obj

return result instanceof Object ? result : this.obj;

}

**实现一个Array.isArray**

思路很简单，就是利用Object.prototype.toString

Array.myIsArray = function(o) {

return Object.prototype.toString.call(Object(o)) === '[object Array]';

};

**实现一个Object.create()方法**

function create = function (o) {

var F = function () {};

F.prototype = o;

return new F();

};

**实现一个EventEmitter**

真实经历，最近在字节跳动的面试中就被面试官问到了，让我手写实现一个简单的Event类。

class Event {

constructor () {

// 储存事件的数据结构

// 为查找迅速， 使用对象（字典）

this.\_cache = {}

}

// 绑定

on(type, callback) {

// 为了按类查找方便和节省空间

// 将同一类型事件放到一个数组中

// 这里的数组是队列， 遵循先进先出

// 即新绑定的事件先触发

let fns = (this.\_cache[type] = this.\_cache[type] || [])

if(fns.indexOf(callback) === -1) {

fns.push(callback)

}

return this

}

// 解绑

off (type, callback) {

let fns = this.\_cache[type]

if(Array.isArray(fns)) {

if(callback) {

let index = fns.indexOf(callback)

if(index !== -1) {

fns.splice(index, 1)

}

} else {

// 全部清空

fns.length = 0

}

}

return this

}

// 触发emit

trigger(type, data) {

let fns = this.\_cache[type]

if(Array.isArray(fns)) {

fns.forEach((fn) => {

fn(data)

})

}

return this

}

// 一次性绑定

once(type, callback) {

let wrapFun = () => {

callback.call(this);

this.off(type, wrapFun );

};

this.on(type, wrapFun );

return this;

}

let e = new Event()

e.on('click',function(){

console.log('on')

})

// e.trigger('click', '666')

console.log(e)

**实现一个Array.prototype.reduce**

首先观察一下Array.prototype.reduce语法

Array.prototype.reduce(callback(accumulator, currentValue[, index[, array]])[, initialValue])

然后就可以动手实现了：

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

let accumulator = initialValue ? initialValue : this[0];

for (let i = initialValue ? 0 : 1; i < this.length; i++) {

let \_this = this;

accumulator = callback(accumulator, this[i], i, \_this);

}

return accumulator;

};

// 使用

let arr = [1, 2, 3, 4];

let sum = arr.myReduce((acc, val) => {

acc += val;

return acc;

}, 5);

console.log(sum); // 15

**实现一个call或apply**

先来看一个call实例，看看call到底做了什么：

let foo = {

value: 1

};

function bar() {

console.log(this.value);

}

bar.call(foo); // 1

从代码的执行结果，我们可以看到，call首先改变了this的指向，使函数的this指向了foo,然后使bar函数执行了。  
总结一下：

1. call改变函数this指向
2. 调用函数

思考一下：我们如何实现上面的效果呢？代码改造如下：

Function.prototype.myCall = function(context) {

context = context || window;

//将函数挂载到对象的fn属性上

context.fn = this;

//处理传入的参数

const args = [...arguments].slice(1);

//通过对象的属性调用该方法

const result = context.fn(...args);

//删除该属性

delete context.fn;

return result

};

我们看一下上面的代码：

1. 首先我们对参数context做了兼容处理，不传值，context默认值为window；
2. 然后我们将函数挂载到context上面,context.fn = this；
3. 处理参数，将传入myCall的参数截取，去除第一位，然后转为数组；
4. 调用context.fn，此时fn的this指向context；
5. 删除对象上的属性 delete context.fn；
6. 将结果返回。

以此类推，我们顺便实现一下apply，唯一不同的是参数的处理,代码如下：

Function.prototype.myApply = function(context) {

context = context || window

context.fn = this

let result

// myApply的参数形式为(obj,[arg1,arg2,arg3]);

// 所以myApply的第二个参数为[arg1,arg2,arg3]

// 这里我们用扩展运算符来处理一下参数的传入方式

if (arguments[1]) {

result = context.fn(…arguments[1])

} else {

result = context.fn()

}

delete context.fn;

return result

};

以上便是call和apply的模拟实现，唯一不同的是对参数的处理方式。

**实现一个Function.prototype.bind**

function Person(){

this.name="zs";

this.age=18;

this.gender="男"

}

let obj={

hobby:"看书"

}

// 将构造函数的this绑定为obj

let changePerson = Person.bind(obj);

// 直接调用构造函数,函数会操作obj对象,给其添加三个属性;

changePerson();

// 1、输出obj

console.log(obj);

// 用改变了this指向的构造函数,new一个实例出来

let p = new changePerson();

// 2、输出obj

console.log(p);

仔细观察上面的代码，再看输出结果。

我们对Person类使用了bind将其this指向obj，得到了changeperson函数，此处如果我们直接调用changeperson会改变obj，若用new调用changeperson会得到实例 p,并且其\_\_proto\_\_指向Person,我们发现bind失效了。

我们得到结论：**用bind改变了this指向的函数，如果用new操作符来调用，bind将会失效。**

这个对象就是这个构造函数的实例，那么只要在函数内部执行 **this instanceof 构造函数** 来判断其结果是否为true，就能判断函数是否是通过new操作符来调用了，若结果为true则是用new操作符调用的，代码修正如下：

// bind实现

Function.prototype.mybind = function(){

// 1、保存函数

let \_this = this;

// 2、保存目标对象

let context = arguments[0]||window;

// 3、保存目标对象之外的参数,将其转化为数组;

let rest = Array.prototype.slice.call(arguments,1);

// 4、返回一个待执行的函数

return function F(){

// 5、将二次传递的参数转化为数组;

let rest2 = Array.prototype.slice.call(arguments)

if(this instanceof F){

// 6、若是用new操作符调用,则直接用new 调用原函数,并用扩展运算符传递参数

return new \_this(...rest2)

}else{

//7、用apply调用第一步保存的函数，并绑定this，传递合并的参数数组，即context.\_this(rest.concat(rest2))

\_this.apply(context,rest.concat(rest2));

}

}

};

**实现一个JS函数柯里化**

Currying的概念其实并不复杂，用通俗易懂的话说：只传递给函数一部分参数来调用它，让它返回一个函数去处理剩下的参数。

function progressCurrying(fn, args) {

let \_this = this

let len = fn.length;

let args = args || [];

return function() {

let \_args = Array.prototype.slice.call(arguments);

Array.prototype.push.apply(args, \_args);

// 如果参数个数小于最初的fn.length，则递归调用，继续收集参数

if (\_args.length < len) {

return progressCurrying.call(\_this, fn, \_args);

}

// 参数收集完毕，则执行fn

return fn.apply(this, \_args);

}

}

**手写防抖(Debouncing)和节流(Throttling)**

**节流**

防抖函数 onscroll 结束时触发一次，延迟执行

function debounce(func, wait) {

let timeout;

return function() {

let context = this; // 指向全局

let args = arguments;

if (timeout) {

clearTimeout(timeout);

}

timeout = setTimeout(() => {

func.apply(context, args); // context.func(args)

}, wait);

};

}

// 使用

window.onscroll = debounce(function() {

console.log('debounce');

}, 1000);

**节流**

节流函数 onscroll 时，每隔一段时间触发一次，像水滴一样

function throttle(fn, delay) {

let prevTime = Date.now();

return function() {

let curTime = Date.now();

if (curTime - prevTime > delay) {

fn.apply(this, arguments);

prevTime = curTime;

}

};

}

// 使用

var throtteScroll = throttle(function() {

console.log('throtte');

}, 1000);

window.onscroll = throtteScroll;

**手写一个JS深拷贝**

乞丐版

JSON.parse(JSON.stringfy));

非常简单，但缺陷也很明显，比如拷贝其他引用类型、拷贝函数、循环引用等情况。

基础版

function clone(target){

if(typeof target === 'object'){

let cloneTarget = {};

for(const key in target){

cloneTarget[key] = clone(target[key])

}

return cloneTarget;

} else {

return target

}

}

写到这里已经可以帮助你应付一些面试官考察你的递归解决问题的能力。但是显然，这个深拷贝函数还是有一些问题。

一个比较完整的深拷贝函数，需要同时考虑对象和数组，考虑循环引用：

function clone(target, map = new WeakMap()) {

if(typeof target === 'object'){

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

if(map.get(target)) {

return target;

}

map.set(target, cloneTarget);

for(const key in target) {

cloneTarget[key] = clone(target[key], map)

}

return cloneTarget;

} else {

return target;

}

}

**实现一个instanceOf**

原理： L 的 **proto** 是不是等于 R.prototype，不等于再找 L.\_\_proto\_\_.\_\_proto\_\_ 直到 **proto** 为 null

// L 表示左表达式，R 表示右表达式

function instance\_of(L, R) {

var O = R.prototype;

L = L.\_\_proto\_\_;

while (true) {

if (L === null){

return false;

}

// 这里重点：当 O 严格等于 L 时，返回 true

if (O === L) {

return true;

}

L = L.\_\_proto\_\_;

}

}

**实现原型链继承**

function myExtend(C, P) {

var F = function(){};

F.prototype = P.prototype;

C.prototype = new F();

C.prototype.constructor = C;

C.super = P.prototype;

}

**实现一个async/await**

**原理**

就是利用 generator（生成器）分割代码片段。然后我们使用一个函数让其自迭代，每一个yield 用 promise 包裹起来。执行下一步的时机由 promise 来控制

**实现**

function \_asyncToGenerator(fn) {

return function() {

var self = this,

args = arguments;

// 将返回值promise化

return new Promise(function(resolve, reject) {

// 获取迭代器实例

var gen = fn.apply(self, args);

// 执行下一步

function \_next(value) {

asyncGeneratorStep(gen, resolve, reject, \_next, \_throw, 'next', value);

}

// 抛出异常

function \_throw(err) {

asyncGeneratorStep(gen, resolve, reject, \_next, \_throw, 'throw', err);

}

// 第一次触发

\_next(undefined);

});

};

}

**实现一个Array.prototype.flat()函数**

最近字节跳动的前端面试中也被面试官问到，要求手写实现。

Array.prototype.myFlat = function(num = 1) {

if (Array.isArray(this)) {

let arr = [];

if (!Number(num) || Number(num) < 0) {

return this;

}

this.forEach(item => {

if(Array.isArray(item)){

let count = num

arr = arr.concat(item.myFlat(--count))

} else {

arr.push(item)

}

});

return arr;

} else {

throw tihs + ".flat is not a function";

}

};

**实现一个事件代理**

这个问题一般还会让你讲一讲事件冒泡和事件捕获机制

<ul id="color-list">

<li>red</li>

<li>yellow</li>

<li>blue</li>

<li>green</li>

<li>black</li>

<li>white</li>

</ul>

<script>

(function () {

var color\_list = document.getElementById('color-list');

color\_list.addEventListener('click', showColor, true);

function showColor(e) {

var x = e.target;

if (x.nodeName.toLowerCase() === 'li') {

alert(x.innerHTML);

}

}

})();

</script>

**实现一个双向绑定**

Vue 2.x的Object.defineProperty版本

// 数据

const data = {

text: 'default'

};

const input = document.getElementById('input');

const span = document.getElementById('span');

// 数据劫持

Object.defineProperty(data, 'text', {

// 数据变化 —> 修改视图

set(newVal) {

input.value = newVal;

span.innerHTML = newVal;

}

});

// 视图更改 --> 数据变化

input.addEventListener('keyup', function(e) {

data.text = e.target.value;

});

Vue 3.x的proxy 版本

// 数据

const data = {

text: 'default'

};

const input = document.getElementById('input');

const span = document.getElementById('span');

// 数据劫持

const handler = {

set(target, key, value) {

target[key] = value;

// 数据变化 —> 修改视图

input.value = value;

span.innerHTML = value;

return value;

}

};

const proxy = new Proxy(data, handler);

// 视图更改 --> 数据变化

input.addEventListener('keyup', function(e) {

proxy.text = e.target.value;

});

思考：Vue双向绑定的实现，使用 ES6 的 Proxy 相比 Object.defineProperty 有什么优势？

**实现一个Array.prototype.map()**

先看看reduce和map的使用方法

let new\_array = arr.map(function callback(currentValue[, index[,array) {}[, thisArg])

let result = arr.reduce(callback(accumulator, currentValue[, index[, array]])[, initialValue])

先用for循环实现：

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

let arr = [];

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

arr.push(callback.call(thisArg, this[i], i, this));

}

return arr;

};

再用reduce实现

Array.prototype.myMap2 = function(callback, thisArg) {

let result = this.reduce((accumulator, currentValue, index, array) => {

accumulator.push(callback.call(thisArg, currentValue, index, array));

return accumulator;

}, []);

return result;

};