死磕 36 个 JS 手写题（搞懂后，提升真的大）

以下文章来源于大海我来了 ，作者布兰

公众号

为什么要写这类文章

作为一个程序员，代码能力毋庸置疑是非常非常重要的，就像现在为什么大厂面试基本都问什么 API 怎么实现可见其重要性。我想说的是居然手写这么重要，那我们就必须掌握它，所以文章标题用了死磕，一点也不过分，也希望不被认为是标题党。

作为一个普通前端，我是真的写不出 Promise A+ 规范，但是没关系，我们可以站在巨人的肩膀上，要相信我们现在要走的路，前人都走过，所以可以找找现在社区已经存在的那些优秀的文章，比如工业聚大佬写的 **100 行代码实现 Promises/A+ 规范**，找到这些文章后不是收藏夹吃灰，得找个时间踏踏实实的学，一行一行的磨，直到搞懂为止。我现在就是这么干的。

能收获什么

这篇文章总体上分为 2 类手写题，前半部分可以归纳为是常见需求，后半部分则是对现有技术的实现；

- 对常用的需求进行手写实现，比如数据类型判断函数、深拷贝等可以直接用于往后的项目中，提高了项目开发效率；
- 对现有关键字和 API 的实现，可能需要用到别的知识或 API，比如在写 forEach 的时候用到了无符号位右移的操作，平时都不怎么能够接触到这玩意，现在遇到了就可以顺手把它掌握了。所以手写这些实现能够潜移默化的扩展并巩固自己的 JS 基础；
- 通过写各种测试用例，你会知道各种 API 的边界情况，比如 Promise.all， 你得考虑到传入参数的各种情况，从而加深了对它们的理解及使用；

阅读的时候需要做什么

阅读的时候，你需要把每行代码都看懂，知道它在干什么，为什么要这么写，能写得更好嘛？比如在写图片懒加载的时候，一般我们都是根据当前元素的位置和视口进行判断是否要加载这张图片，普通程序员写到这就差不多完成了。而大佬程序员则是会多考虑一些细节的东西，比如性能如何更优？代码如何更精简？比如 yeyan1996 写的图片懒加载就多考虑了 2 点：比如图片全部加载完成的时候得把事件监听给移除；比如加载完一张图片的时候，得把当前 img 从 imgList 里移除，起到优化内存的作用。

除了读通代码之外，还可以打开 Chrome 的 Script snippet 去写测试用例跑跑代码，做到更好的理解以及使用。

在看了几篇以及写了很多测试用例的前提下，尝试自己手写实现，看看自己到底掌握了多少。条条大路通罗马，你还能有别的方式实现嘛？或者你能写得比别人更好嘛？

好了，还楞着干啥，开始干活。

数据类型判断

typeof 可以正确识别：Undefined、Boolean、Number、String、Symbol、Function 等类型的数据，但是对于其他的都会认为是 object，比如 Null、Date 等，所以通过 typeof 来判断数据类型会不准确。但是可以使用 Object.prototype.toString 实现。

```
function typeOf(obj) {
    let res = Object.prototype.toString.call(obj).split(' ')[1]
    res = res.substring(0, res.length - 1).toLowerCase()
    return res
}
typeOf([])        // 'array'
typeOf({})        // 'object'
typeOf(new Date)  // 'date'
```

继承

原型链继承

```
function Animal() {
    this.colors = ['black', 'white']
}
Animal.prototype.getColor = function() {
    return this.colors
}
```

```
function Dog() {}
Dog.prototype =  new Animal()

let dog1 = new Dog()
dog1.colors.push('brown')
let dog2 = new Dog()
console.log(dog2.colors)  // ['black', 'white', 'brown']
```

原型链继承存在的问题：

- 问题1：原型中包含的引用类型属性将被所有实例共享；
- 问题2：子类在实例化的时候不能给父类构造函数传参；

借用构造函数实现继承

```
function Animal(name) {
    this.name = name
    this.getName = function() {
        return this.name
    }
}
function Dog(name) {
    Animal.call(this, name)
}
Dog.prototype =  new Animal()
```

借用构造函数实现继承解决了原型链继承的 2 个问题：引用类型共享问题以及传参问题。但是由于方法必须定义在构造函数中，所以会导致每次创建子类实例都会创建一遍方法。

组合继承

组合继承结合了原型链和盗用构造函数，将两者的优点集中了起来。基本的思路是使用原型链继承原型上的属性和方法，而通过盗用构造函数继承实例属性。这样既可以把方法定义在原型上以实现重用，又可以让每个实例都有自己的属性。

```
function Animal(name) {
    this.name = name
    this.colors = ['black', 'white']
}
Animal.prototype.getName = function() {
    return this.name
}
function Dog(name, age) {
    Animal.call(this, name)
    this.age = age
}
Dog.prototype =  new Animal()
Dog.prototype.constructor = Dog

let dog1 = new Dog('奶昔', 2)
dog1.colors.push('brown')
let dog2 = new Dog('哈赤', 1)
console.log(dog2)
// { name: "哈赤", colors: ["black", "white"], age: 1 }
```

寄生式组合继承

组合继承已经相对完善了，但还是存在问题，它的问题就是调用了 2 次父类构造函数，第一次是在 new Animal()，第二次是在 Animal.call() 这里。

所以解决方案就是不直接调用父类构造函数给子类原型赋值，而是通过创建空函数 F 获取父类原型的副本。

寄生式组合继承写法上和组合继承基本类似，区别是如下这里：

```
- Dog.prototype =  new Animal()
- Dog.prototype.constructor = Dog

+ function F() {}
+ F.prototype = Animal.prototype
+ let f = new F()
```

```
+ f.constructor = Dog
+ Dog.prototype = f
```

稍微封装下上面添加的代码后：

```
function object(o) {
    function F() {}
    F.prototype = o
    return new F()
}
function inheritPrototype(child, parent) {
    let prototype = object(parent.prototype)
    prototype.constructor = child
    child.prototype = prototype
}
inheritPrototype(Dog, Animal)
```

如果你嫌弃上面的代码太多了，还可以基于组合继承的代码改成最简单的寄生式组合继承：

```
- Dog.prototype =  new Animal()
- Dog.prototype.constructor = Dog

+ Dog.prototype =  Object.create(Animal.prototype)
+ Dog.prototype.constructor = Dog
```

class 实现继承

```
class Animal {
    constructor(name) {
        this.name = name
    }
    getName() {
        return this.name
    }
}
```

```
class Dog extends Animal {
    constructor(name, age) {
        super(name)
        this.age = age
    }
}
```

数组去重

ES5 实现：

```
function unique(arr) {
    var res = arr.filter(function(item, index, array) {
        return array.indexOf(item) === index
    })
    return res
}
```

ES6 实现：

```
var unique = arr => [...new Set(arr)]
```

数组扁平化

数组扁平化就是将 [1, [2, [3]]] 这种多层的数组拍平成一层 [1, 2, 3]。使用 Array.prototype.flat 可以直接将多层数组拍平成一层：

```
[1, [2, [3]]].flat(2)  // [1, 2, 3]
```

现在就是要实现 flat 这种效果。

ES5 实现：递归。

```
function flatten(arr) {
    var result = [];
    for (var i = 0, len = arr.length; i < len; i++) {
        if (Array.isArray(arr[i])) {
            result = result.concat(flatten(arr[i]))
        } else {
            result.push(arr[i])
        }
    }
    return result;
}
```

ES6 实现：

```
function flatten(arr) {
    while (arr.some(item => Array.isArray(item))) {
        arr = [].concat(...arr);
    }
    return arr;
}
```

深浅拷贝

浅拷贝：只考虑对象类型。

```
function shallowCopy(obj) {
    if (typeof obj !== 'object') return

    let newObj = obj instanceof Array ? [] : {}
    for (let key in obj) {
        if (obj.hasOwnProperty(key)) {
            newObj[key] = obj[key]
```

```
        }
    }
    return newObj
}
```

简单版深拷贝：只考虑普通对象属性，不考虑内置对象和函数。

```
function deepClone(obj) {
    if (typeof obj !== 'object') return;
    var newObj = obj instanceof Array ? [] : {};
    for (var key in obj) {
        if (obj.hasOwnProperty(key)) {
            newObj[key] = typeof obj[key] === 'object' ? deepClone(obj[key]) : obj[key];
        }
    }
    return newObj;
}
```

复杂版深克隆：基于简单版的基础上，还考虑了内置对象比如 Date、RegExp 等对象和函数以及解决了循环引用的问题。

```
const isObject = (target) => (typeof target === "object" || typeof target === "function") && t

function deepClone(target, map = new WeakMap()) {
    if (map.get(target)) {
        return target;
    }
    // 获取当前值的构造函数：获取它的类型
    let constructor = target.constructor;
    // 检测当前对象target是否与正则、日期格式对象匹配
    if (/^(RegExp|Date)$/i.test(constructor.name)) {
        // 创建一个新的特殊对象(正则类/日期类)的实例
        return new constructor(target);
    }
    if (isObject(target)) {
        map.set(target, true);  // 为循环引用的对象做标记
        const cloneTarget = Array.isArray(target) ? [] : {};
```

```
        for (let prop in target) {
            if (target.hasOwnProperty(prop)) {
                cloneTarget[prop] = deepClone(target[prop], map);
            }
        }
        return cloneTarget;
    } else {
        return target;
    }
}
```

事件总线（发布订阅模式）

```
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) {
        let 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, ...args) {
        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
            let tasks = this.cache[name].slice()
```

```
            for (let fn of tasks) {
                fn(...args)
            }
            if (once) {
                delete this.cache[name]
            }
        }
    }
}

// 测试
let eventBus = new EventEmitter()
let fn1 = function(name, age) {
    console.log(`${name} ${age}`)
}
let fn2 = function(name, age) {
    console.log(`hello, ${name} ${age}`)
}
eventBus.on('aaa', fn1)
eventBus.on('aaa', fn2)
eventBus.emit('aaa', false, '布兰', 12)
// '布兰 12'
// 'hello, 布兰 12'
```

解析 URL 参数为对象

```
function parseParam(url) {
    const paramsStr = /.+\?(.+)$/.exec(url)[1]; // 将 ? 后面的字符串取出来
    const paramsArr = paramsStr.split('&'); // 将字符串以 & 分割后存到数组中
    let paramsObj = {};
    // 将 params 存到对象中
    paramsArr.forEach(param => {
        if (/=/.test(param)) { // 处理有 value 的参数
            let [key, val] = param.split('='); // 分割 key 和 value
            val = decodeURIComponent(val); // 解码
            val = /^\d+$/.test(val) ? parseFloat(val) : val; // 判断是否转为数字
```

```
            if (paramsObj.hasOwnProperty(key)) { // 如果对象有 key，则添加一个值
                paramsObj[key] = [].concat(paramsObj[key], val);
            } else { // 如果对象没有这个 key，创建 key 并设置值
                paramsObj[key] = val;
            }
        } else { // 处理没有 value 的参数
            paramsObj[param] = true;
        }
    })

    return paramsObj;
}
```

字符串模板

```
function render(template, data) {
    const reg = /\{\{(\w+)\}\}/; // 模板字符串正则
    if (reg.test(template)) { // 判断模板里是否有模板字符串
        const name = reg.exec(template)[1]; // 查找当前模板里第一个模板字符串的字段
        template = template.replace(reg, data[name]); // 将第一个模板字符串渲染
        return render(template, data); // 递归的渲染并返回渲染后的结构
    }
    return template; // 如果模板没有模板字符串直接返回
}
```

测试：

```
let template = '我是{{name}}，年龄{{age}}，性别{{sex}}';
let person = {
    name: '布兰',
    age: 12
}
render(template, person); // 我是布兰，年龄12，性别undefined
```

图片懒加载

与普通的图片懒加载不同，如下这个多做了 2 个精心处理：

- 图片全部加载完成后移除事件监听；
- 加载完的图片，从 imgList 移除；

```
let imgList = [...document.querySelectorAll('img')]
let length = imgList.length

const imgLazyLoad = function() {
    let count = 0
    return (function() {
        let deleteIndexList = []
        imgList.forEach((img, index) => {
            let rect = img.getBoundingClientRect()
            if (rect.top < window.innerHeight) {
                img.src = img.dataset.src
                deleteIndexList.push(index)
                count++
                if (count === length) {
                    document.removeEventListener('scroll', imgLazyLoad)
                }
            }
        })
        imgList = imgList.filter((img, index) => !deleteIndexList.includes(index))
    })()
}

// 这里最好加上防抖处理
document.addEventListener('scroll', imgLazyLoad)
```

参考：图片懒加载[1]

函数防抖

触发高频事件 N 秒后只会执行一次，如果 N 秒内事件再次触发，则会重新计时。

简单版：函数内部支持使用 this 和 event 对象；

```
function debounce(func, wait) {
    var timeout;
    return function () {
        var context = this;
        var args = arguments;
        clearTimeout(timeout)
        timeout = setTimeout(function(){
            func.apply(context, args)
        }, wait);
    }
}
```

使用：

```
var node = document.getElementById('layout')
function getUserAction(e) {
    console.log(this, e)  // 分别打印：node 这个节点 和 MouseEvent
    node.innerHTML = count++;
};
node.onmousemove = debounce(getUserAction, 1000)
```

最终版：除了支持 this 和 event 外，还支持以下功能：

- 支持立即执行；
- 函数可能有返回值；
- 支持取消功能；

```
function debounce(func, wait, immediate) {
    var timeout, result;

    var debounced = function () {
```

```
        var context = this;
        var args = arguments;

        if (timeout) clearTimeout(timeout);
        if (immediate) {
            // 如果已经执行过，不再执行
            var callNow = !timeout;
            timeout = setTimeout(function(){
                timeout = null;
            }, wait)
            if (callNow) result = func.apply(context, args)
        } else {
            timeout = setTimeout(function(){
                func.apply(context, args)
            }, wait);
        }
        return result;
    };

    debounced.cancel = function() {
        clearTimeout(timeout);
        timeout = null;
    };

    return debounced;
}
```

使用：

```
var setUseAction = debounce(getUserAction, 10000, true);
// 使用防抖
node.onmousemove = setUseAction

// 取消防抖
setUseAction.cancel()
```

参考：JavaScript专题之跟着underscore学防抖

函数节流

触发高频事件，且 N 秒内只执行一次。

简单版：使用时间戳来实现，立即执行一次，然后每 N 秒执行一次。

```
function throttle(func, wait) {
    var context, args;
    var previous = 0;

    return function() {
        var now = +new Date();
        context = this;
        args = arguments;
        if (now - previous > wait) {
            func.apply(context, args);
            previous = now;
        }
    }
}
```

最终版：支持取消节流；另外通过传入第三个参数，options.leading 来表示是否可以立即执行一次，opitons.trailing 表示结束调用的时候是否还要执行一次，默认都是 true。注意设置的时候不能同时将 leading 或 trailing 设置为 false。

```
function throttle(func, wait, options) {
    var timeout, context, args, result;
    var previous = 0;
    if (!options) options = {};

    var later = function() {
        previous = options.leading === false ? 0 : new Date().getTime();
        timeout = null;
        func.apply(context, args);
        if (!timeout) context = args = null;
    };
```

```
    var throttled = function() {
        var now = new Date().getTime();
        if (!previous && options.leading === false) previous = now;
        var remaining = wait - (now - previous);
        context = this;
        args = arguments;
        if (remaining <= 0 || remaining > wait) {
            if (timeout) {
                clearTimeout(timeout);
                timeout = null;
            }
            previous = now;
            func.apply(context, args);
            if (!timeout) context = args = null;
        } else if (!timeout && options.trailing !== false) {
            timeout = setTimeout(later, remaining);
        }
    };

    throttled.cancel = function() {
        clearTimeout(timeout);
        previous = 0;
        timeout = null;
    }
    return throttled;
}
```

节流的使用就不拿代码举例了，参考防抖的写就行。

参考：JavaScript专题之跟着 underscore 学节流

函数柯里化

什么叫函数柯里化？其实就是将使用多个参数的函数转换成一系列使用一个参数的函数的技术。还不懂？来举个例子。

```
function add(a, b, c) {
```

```
    return a + b + c
}
add(1, 2, 3)
let addCurry = curry(add)
addCurry(1)(2)(3)
```

现在就是要实现 curry 这个函数，使函数从一次调用传入多个参数变成多次调用每次传一个参数。

```
function curry(fn) {
    let judge = (...args) => {
        if (args.length == fn.length) return fn(...args)
        return (...arg) => judge(...args, ...arg)
    }
    return judge
}
```

偏函数

什么是偏函数？偏函数就是将一个 n 参的函数转换成固定 x 参的函数，剩余参数（n - x）将在下次调用全部传入。举个例子：

```
function add(a, b, c) {
    return a + b + c
}
let partialAdd = partial(add, 1)
partialAdd(2, 3)
```

发现没有，其实偏函数和函数柯里化有点像，所以根据函数柯里化的实现，能够能很快写出偏函数的实现：

```
function partial(fn, ...args) {
    return (...arg) => {
```

```
        return fn(...args, ...arg)
    }
}
```

如上这个功能比较简单，现在我们希望偏函数能和柯里化一样能实现占位功能，比如：

```
function clg(a, b, c) {
    console.log(a, b, c)
}
let partialClg = partial(clg, '_', 2)
partialClg(1, 3)  // 依次打印：1, 2, 3
```

`_` 占的位其实就是 1 的位置。相当于：partial(clg, 1, 2)，然后 partialClg(3)。明白了原理，我们就来写实现：

```
function partial(fn, ...args) {
    return (...arg) => {
        args[index] =
        return fn(...args, ...arg)
    }
}
```

JSONP

JSONP 核心原理：script 标签不受同源策略约束，所以可以用来进行跨域请求，优点是兼容性好，但是只能用于 GET 请求；

```
const jsonp = ({ url, params, callbackName }) => {
    const generateUrl = () => {
        let dataSrc = ''
        for (let key in params) {
            if (params.hasOwnProperty(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)
        }
    })
}
```

AJAX

```
const getJSON = function(url) {
    return new Promise((resolve, reject) => {
        const xhr = XMLHttpRequest ? new XMLHttpRequest() : new ActiveXObject('Mscrosoft.XMLHt
        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();
    })
}
```

实现数组原型方法

forEach

```
Array.prototype.forEach2 = 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)  // this 就是当前的数组
    const len = O.length >>> 0  // 后面有解释
    let k = 0
    while (k < len) {
        if (k in O) {
            callback.call(thisArg, O[k], k, O);
        }
        k++;
    }
}
```

参考：**forEach#polyfill**[2]

O.length >>> 0 是什么操作？就是无符号右移 0 位，那有什么意义嘛？就是为了保证转换后的值为正整数。其实底层做了 2 层转换，第一是非 number 转成 number 类型，第二是将 number 转成 Uint32 类型。感兴趣可以阅读 **something >>> 0是什么意思?**[3]。

map

基于 forEach 的实现能够很容易写出 map 的实现：

```
- Array.prototype.forEach2 = function(callback, thisArg) {
+ Array.prototype.map2 = 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
+   let k = 0, res = []
    while (k < len) {
        if (k in O) {
-           callback.call(thisArg, O[k], k, O);
+           res[k] = callback.call(thisArg, O[k], k, O);
        }
        k++;
    }
+   return res
}
```

filter

同样，基于 forEach 的实现能够很容易写出 filter 的实现：

```
- Array.prototype.forEach2 = function(callback, thisArg) {
+ Array.prototype.filter2 = 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
+   let k = 0, res = []
    while (k < len) {
        if (k in O) {
-           callback.call(thisArg, O[k], k, O);
+           if (callback.call(thisArg, O[k], k, O)) {
+               res.push(O[k])
+           }
        }
        k++;
    }
```

```
+   return res
}
```

some

同样，基于 forEach 的实现能够很容易写出 some 的实现：

```
- Array.prototype.forEach2 = function(callback, thisArg) {
+ Array.prototype.some2 = 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);
+           if (callback.call(thisArg, O[k], k, O)) {
+               return true
+           }
        }
        k++;
    }
+   return false
}
```

reduce

```
Array.prototype.reduce2 = function(callback, initialValue) {
    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, acc

    if (arguments.length > 1) {
        acc = initialValue
    } else {
        // 没传入初始值的时候，取数组中第一个非 empty 的值为初始值
        while (k < len && !(k in O)) {
            k++
        }
        if (k > len) {
            throw new TypeError( 'Reduce of empty array with no initial value' );
        }
        acc = O[k++]
    }
    while (k < len) {
        if (k in O) {
            acc = callback(acc, O[k], k, O)
        }
        k++
    }
    return acc
}
```

实现函数原型方法

call

使用一个指定的 this 值和一个或多个参数来调用一个函数。

实现要点：

- this 可能传入 null；
- 传入不固定个数的参数；
- 函数可能有返回值；

```
Function.prototype.call2 = function (context) {
    var context = context || window;
    context.fn = this;

    var args = [];
    for(var i = 1, len = arguments.length; i < len; i++) {
        args.push('arguments[' + i + ']');
    }

    var result = eval('context.fn(' + args +')');

    delete context.fn
    return result;
}
```

apply

apply 和 call 一样，唯一的区别就是 call 是传入不固定个数的参数，而 apply 是传入一个数组。

实现要点：

- this 可能传入 null；
- 传入一个数组；
- 函数可能有返回值；

```
Function.prototype.apply2 = function (context, arr) {
    var context = context || window;
    context.fn = this;

    var result;
    if (!arr) {
        result = context.fn();
    } else {
        var args = [];
        for (var i = 0, len = arr.length; i < len; i++) {
            args.push('arr[' + i + ']');
        }
```

```
        result = eval('context.fn(' + args + ')')
    }

    delete context.fn
    return result;
}
```

bind

bind 方法会创建一个新的函数，在 bind() 被调用时，这个新函数的 this 被指定为 bind() 的第一个参数，而其余参数将作为新函数的参数，供调用时使用。

实现要点：

- bind() 除了 this 外，还可传入多个参数；
- bing 创建的新函数可能传入多个参数；
- 新函数可能被当做构造函数调用；
- 函数可能有返回值；

```
Function.prototype.bind2 = function (context) {
    var self = this;
    var args = Array.prototype.slice.call(arguments, 1);

    var fNOP = function () {};

    var fBound = function () {
        var bindArgs = Array.prototype.slice.call(arguments);
        return self.apply(this instanceof fNOP ? this : context, args.concat(bindArgs));
    }

    fNOP.prototype = this.prototype;
    fBound.prototype = new fNOP();
    return fBound;
}
```

实现 new 关键字

new 运算符用来创建用户自定义的对象类型的实例或者具有构造函数的内置对象的实例。

实现要点：

- new 会产生一个新对象；
- 新对象需要能够访问到构造函数的属性，所以需要重新指定它的原型；
- 构造函数可能会显示返回；

```
function objectFactory() {
    var obj = new Object()
    Constructor = [].shift.call(arguments);
    obj.__proto__ = Constructor.prototype;
    var ret = Constructor.apply(obj, arguments);

    // ret || obj 这里这么写考虑了构造函数显示返回 null 的情况
    return typeof ret === 'object' ? ret || obj : obj;
};
```

使用：

```
function person(name, age) {
    this.name = name
    this.age = age
}
let p = objectFactory(person, '布兰', 12)
console.log(p)  // { name: '布兰', age: 12 }
```

实现 instanceof 关键字

instanceof 就是判断构造函数的 prototype 属性是否出现在实例的原型链上。

```
function instanceOf(left, right) {
    let proto = left.__proto__
```

```
    while (true) {
        if (proto === null) return false
        if (proto === right.prototype) {
            return true
        }
        proto = proto.__proto__
    }
}
```

上面的 left.**proto** 这种写法可以换成 Object.getPrototypeOf(left)。

实现 Object.create

Object.create()方法创建一个新对象，使用现有的对象来提供新创建的对象的__proto__。

```
Object.create2 = function(proto, propertyObject = undefined) {
    if (typeof proto !== 'object' && typeof proto !== 'function') {
        throw new TypeError('Object prototype may only be an Object or null.')
    if (propertyObject == null) {
        new TypeError('Cannot convert undefined or null to object')
    }
    function F() {}
    F.prototype = proto
    const obj = new F()
    if (propertyObject != undefined) {
        Object.defineProperties(obj, propertyObject)
    }
    if (proto === null) {
        // 创建一个没有原型对象的对象，Object.create(null)
        obj.__proto__ = null
    }
    return obj
}
```

实现 Object.assign

```
Object.assign2 = function(target, ...source) {
    if (target == null) {
        throw new TypeError('Cannot convert undefined or null to object')
    }
    let ret = Object(target)
    source.forEach(function(obj) {
        if (obj != null) {
            for (let key in obj) {
                if (obj.hasOwnProperty(key)) {
                    ret[key] = obj[key]
                }
            }
        }
    })
    return ret
}
```

实现 JSON.stringify

JSON.stringify([, replacer [, space]) 方法是将一个 JavaScript 值(对象或者数组)转换为一个 JSON 字符串。此处模拟实现，不考虑可选的第二个参数 replacer 和第三个参数 space，如果对这两个参数的作用还不了解，建议阅读 **MDN**[4] 文档。

1. 基本数据类型：
    - undefined 转换之后仍是 undefined(类型也是 undefined)
    - boolean 值转换之后是字符串 "false"/"true"
    - number 类型(除了 NaN 和 Infinity)转换之后是字符串类型的数值
    - symbol 转换之后是 undefined
    - null 转换之后是字符串 "null"
    - string 转换之后仍是string
    - NaN 和 Infinity 转换之后是字符串 "null"
2. 函数类型：转换之后是 undefined
3. 如果是对象类型(非函数)
    - 如果有 toJSON() 方法，那么序列化 toJSON() 的返回值。

- 如果属性值中出现了 undefined、任意的函数以及 symbol 值，忽略。
- 所有以 symbol 为属性键的属性都会被完全忽略掉。
- 如果是一个数组：如果属性值中出现了 undefined、任意的函数以及 symbol，转换成字符串 "null"；
- 如果是 RegExp 对象：返回 {} (类型是 string)；
- 如果是 Date 对象，返回 Date 的 toJSON 字符串值；
- 如果是普通对象；

4. 对包含循环引用的对象（对象之间相互引用，形成无限循环）执行此方法，会抛出错误。

```
function jsonStringify(data) {
    let dataType = typeof data;

    if (dataType !== 'object') {
        let result = data;
        //data 可能是 string/number/null/undefined/boolean
        if (Number.isNaN(data) || data === Infinity) {
            //NaN 和 Infinity 序列化返回 "null"
            result = "null";
        } else if (dataType === 'function' || dataType === 'undefined' || dataType === 'symbol
            //function 、undefined 、symbol 序列化返回 undefined
            return undefined;
        } else if (dataType === 'string') {
            result = '"' + data + '"';
        }
        //boolean 返回 String()
        return String(result);
    } else if (dataType === 'object') {
        if (data === null) {
            return "null"
        } else if (data.toJSON && typeof data.toJSON === 'function') {
            return jsonStringify(data.toJSON());
        } else if (data instanceof Array) {
            let result = [];
            //如果是数组
            //toJSON 方法可以存在于原型链中
            data.forEach((item, index) => {
```

```
                if (typeof item === 'undefined' || typeof item === 'function' || typeof item ==
                    result[index] = "null";
                } else {
                    result[index] = jsonStringify(item);
                }
            });
            result = "[" + result + "]";
            return result.replace(/'/g, '"');

        } else {
            //普通对象
            /**
             * 循环引用抛错(暂未检测，循环引用时，堆栈溢出)
             * symbol key 忽略
             * undefined、函数、symbol 为属性值，被忽略
             */
            let result = [];
            Object.keys(data).forEach((item, index) => {
                if (typeof item !== 'symbol') {
                    //key 如果是symbol对象，忽略
                    if (data[item] !== undefined && typeof data[item] !== 'function'
                        && typeof data[item] !== 'symbol') {
                        //键值如果是 undefined、函数、symbol 为属性值，忽略
                        result.push('"' + item + '"' + ":" + jsonStringify(data[item]));
                    }
                }
            });
            return ("{" + result + "}").replace(/'/g, '"');
        }
    }
}
```

参考：**实现 JSON.stringify**[5]

实现 JSON.parse

介绍 2 种方法实现：

- eval 实现；
- new Function 实现；

eval 实现

第一种方式最简单，也最直观，就是直接调用 eval，代码如下：

```
var json = '{"a":"1", "b":2}';
var obj = eval("(" + json + ")");  // obj 就是 json 反序列化之后得到的对象
```

但是直接调用 eval 会存在安全问题，如果数据中可能不是 json 数据，而是可执行的 JavaScript 代码，那很可能会造成 XSS 攻击。因此，在调用 eval 之前，需要对数据进行校验。

```
var rx_one = /^[\],:{}\s]*$/;
var rx_two = /\\(?:["\\\/bfnrt]|u[0-9a-fA-F]{4})/g;
var rx_three = /"[^"\\\n\r]*"|true|false|null|-?\d+(?:\.\d*)?(?:[eE][+\-]?\d+)?/g;
var rx_four = /(?:^|:|,)(?:\s*\[)+/g;

if (
    rx_one.test(
        json.replace(rx_two, "@")
            .replace(rx_three, "]")
            .replace(rx_four, "")
    )
) {
    var obj = eval("(" +json + ")");
}
```

参考：JSON.parse 三种实现方式[6]

new Function 实现

Function 与 eval 有相同的字符串参数特性。

```
var json = '{"name":"小姐姐", "age":20}';
var obj = (new Function('return ' + json))();
```

实现 Promise

实现 Promise 需要完全读懂 **Promise A+ 规范**[7]，不过从总体的实现上看，有如下几个点需要考虑到：

- then 需要支持链式调用，所以得返回一个新的 Promise；
- 处理异步问题，所以得先用 onResolvedCallbacks 和 onRejectedCallbacks 分别把成功和失败的回调存起来；
- 为了让链式调用正常进行下去，需要判断 onFulfilled 和 onRejected 的类型；
- onFulfilled 和 onRejected 需要被异步调用，这里用 setTimeout 模拟异步；
- 处理 Promise 的 resolve；

```
const PENDING = 'pending';
const FULFILLED = 'fulfilled';
const REJECTED = 'rejected';

class Promise {
    constructor(executor) {
        this.status = PENDING;
        this.value = undefined;
        this.reason = undefined;
        this.onResolvedCallbacks = [];
        this.onRejectedCallbacks = [];

        let resolve = (value) = > {
            if (this.status === PENDING) {
                this.status = FULFILLED;
                this.value = value;
                this.onResolvedCallbacks.forEach((fn) = > fn());
            }
        };
```

```
        let reject = (reason) => {
            if (this.status === PENDING) {
                this.status = REJECTED;
                this.reason = reason;
                this.onRejectedCallbacks.forEach((fn) => fn());
            }
        };

        try {
            executor(resolve, reject);
        } catch (error) {
            reject(error);
        }
    }

    then(onFulfilled, onRejected) {
        // 解决 onFufilled，onRejected 没有传值的问题
        onFulfilled = typeof onFulfilled === "function" ? onFulfilled : (v) => v;
        // 因为错误的值要让后面访问到，所以这里也要抛出错误，不然会在之后 then 的 resolve 中捕获
        onRejected = typeof onRejected === "function" ? onRejected : (err) => {
            throw err;
        };
        // 每次调用 then 都返回一个新的 promise
        let promise2 = new Promise((resolve, reject) => {
            if (this.status === FULFILLED) {
                //Promise/A+ 2.2.4 --- setTimeout
                setTimeout(() => {
                    try {
                        let x = onFulfilled(this.value);
                        // x可能是一个proimise
                        resolvePromise(promise2, x, resolve, reject);
                    } catch (e) {
                        reject(e);
                    }
                }, 0);
            }

            if (this.status === REJECTED) {
                //Promise/A+ 2.2.3
                setTimeout(() => {
                    try {
```

```
                        let x = onRejected(this.reason);
                        resolvePromise(promise2, x, resolve, reject);
                    } catch (e) {
                        reject(e);
                    }
                }, 0);
            }

            if (this.status === PENDING) {
                this.onResolvedCallbacks.push(() => {
                    setTimeout(() => {
                        try {
                            let x = onFulfilled(this.value);
                            resolvePromise(promise2, x, resolve, reject);
                        } catch (e) {
                            reject(e);
                        }
                    }, 0);
                });

                this.onRejectedCallbacks.push(() => {
                    setTimeout(() => {
                        try {
                            let x = onRejected(this.reason);
                            resolvePromise(promise2, x, resolve, reject);
                        } catch (e) {
                            reject(e);
                        }
                    }, 0);
                });
            }
        });

        return promise2;
    }
}
const resolvePromise = (promise2, x, resolve, reject) => {
    // 自己等待自己完成是错误的实现，用一个类型错误，结束掉 promise  Promise/A+ 2.3.1
    if (promise2 === x) {
        return reject(
            new TypeError("Chaining cycle detected for promise #<Promise>"));
    }
```

```
    // Promise/A+ 2.3.3.3.3 只能调用一次
    let called;
    // 后续的条件要严格判断 保证代码能和别的库一起使用
    if ((typeof x === "object" && x != null) || typeof x === "function") {
        try {
            // 为了判断 resolve 过的就不用再 reject 了（比如 reject 和 resolve 同时调用的时候）  P
            let then = x.then;
            if (typeof then === "function") {
            // 不要写成 x.then，直接 then.call 就可以了 因为 x.then 会再次取值，Object.defineProp
                then.call(
                    x, (y) = > {
                        // 根据 promise 的状态决定是成功还是失败
                        if (called) return;
                        called = true;
                        // 递归解析的过程（因为可能 promise 中还有 promise） Promise/A+ 2.3.3.3.1
                        resolvePromise(promise2, y, resolve, reject);
                    }, (r) = > {
                        // 只要失败就失败 Promise/A+ 2.3.3.3.2
                        if (called) return;
                        called = true;
                        reject(r);
                    });
            } else {
                // 如果 x.then 是个普通值就直接返回 resolve 作为结果  Promise/A+ 2.3.3.4
                resolve(x);
            }
        } catch (e) {
            // Promise/A+ 2.3.3.2
            if (called) return;
            called = true;
            reject(e);
        }
    } else {
        // 如果 x 是个普通值就直接返回 resolve 作为结果  Promise/A+ 2.3.4
        resolve(x);
    }
};
```

Promise 写完之后可以通过 promises-aplus-tests 这个包对我们写的代码进行测试，看是否符合 A+ 规范。不过测试前还得加一段代码：

```
// promise.js
// 这里是上面写的 Promise 全部代码
Promise.defer = Promise.deferred = function () {
    let dfd = {}
    dfd.promise = new Promise((resolve,reject)=>{
        dfd.resolve = resolve;
        dfd.reject = reject;
    });
    return dfd;
}
module.exports = Promise;
```

全局安装：

```
npm i promises-aplus-tests -g
```

终端下执行验证命令：

```
promises-aplus-tests promise.js
```

上面写的代码可以顺利通过全部 872 个测试用例。

参考：

- **BAT前端经典面试问题：史上最最最详细的手写Promise教程**[8]
- **100 行代码实现 Promises/A+ 规范**[9]

Promise.resolve

Promsie.resolve(value) 可以将任何值转成值为 value 状态是 fulfilled 的 Promise，但如果传入的值本身是 Promise 则会原样返回它。

```
```

```
Promise.resolve = function(value) {
    // 如果是 Promsie，则直接输出它
    if(value instanceof Promise){
        return value
    }
    return new Promise(resolve => resolve(value))
}
```

参考：**深入理解 Promise**[10]

Promise.reject

和 Promise.resolve() 类似，Promise.reject() 会实例化一个 rejected 状态的 Promise。但与 Promise.resolve() 不同的是，如果给 Promise.reject() 传递一个 Promise 对象，则这个对象会成为新 Promise 的值。

```
Promise.reject = function(reason) {
    return new Promise((resolve, reject) => reject(reason))
}
```

Promise.all

Promise.all 的规则是这样的：

- 传入的所有 Promsie 都是 fulfilled，则返回由他们的值组成的，状态为 fulfilled 的新 Promise；
- 只要有一个 Promise 是 rejected，则返回 rejected 状态的新 Promsie，且它的值是第一个 rejected 的 Promise 的值；
- 只要有一个 Promise 是 pending，则返回一个 pending 状态的新 Promise；

```
Promise.all = function(promiseArr) {
    let index = 0, result = []
    return new Promise((resolve, reject) => {
        promiseArr.forEach((p, i) => {
            Promise.resolve(p).then(val => {
```

```
                index++
                result[i] = val
                if (index === promiseArr.length) {
                    resolve(result)
                }
            }, err => {
                reject(err)
            })
        })
    })
}
```

Promise.race

Promise.race 会返回一个由所有可迭代实例中第一个 fulfilled 或 rejected 的实例包装后的新实例。

```
Promise.race = function(promiseArr) {
    return new Promise((resolve, reject) => {
        promiseArr.forEach(p => {
            Promise.resolve(p).then(val => {
                resolve(val)
            }, err => {
                rejecte(err)
            })
        })
    })
}
```

Promise.allSettled

Promise.allSettled 的规则是这样：

- 所有 Promise 的状态都变化了，那么新返回一个状态是 fulfilled 的 Promise，且它的值是一个数组，数组的每项由所有 Promise 的值和状态组成的对象；
- 如果有一个是 pending 的 Promise，则返回一个状态是 pending 的新实例；

```
Promise.allSettled = function(promiseArr) {
```

```
    let result = []

    return new Promise((resolve, reject) => {
        promiseArr.forEach((p, i) => {
            Promise.resolve(p).then(val => {
                result.push({
                    status: 'fulfilled',
                    value: val
                })
                if (result.length === promiseArr.length) {
                    resolve(result)
                }
            }, err => {
                result.push({
                    status: 'rejected',
                    reason: err
                })
                if (result.length === promiseArr.length) {
                    resolve(result)
                }
            })
        })
    })
}
```

Promise.any

Promise.any 的规则是这样：

- 空数组或者所有 Promise 都是 rejected，则返回状态是 rejected 的新 Promsie，且值为 AggregateError 的错误；
- 只要有一个是 fulfilled 状态的，则返回第一个是 fulfilled 的新实例；
- 其他情况都会返回一个 pending 的新实例；

```
Promise.any = function(promiseArr) {
    let index = 0
    return new Promise((resolve, reject) => {
        if (promiseArr.length === 0) return
        promiseArr.forEach((p, i) => {
```

```
            Promise.resolve(p).then(val => {
                resolve(val)

            }, err => {
                index++
                if (index === promiseArr.length) {
                  reject(new AggregateError('All promises were rejected'))
                }
            })
        })
    })
}
```

后话

能看到这里的对代码都是真爱了，毕竟代码这玩意看起来是真的很枯燥，但是如果看懂了后，就会像打游戏赢了一样开心，而且这玩意会上瘾，当你通关了越多的关卡后，你的能力就会拔高一个层次。用标题的话来说就是：搞懂后，提升真的大。加油吧💪，干饭人

噢不，代码人。

参考资料

[1] 图片懒加载: *https://juejin.cn/post/6844903856489365518#heading-19*

[2] forEach#polyfill: *https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach#polyfill*

[3] something >>> 0是什么意思: *https://zhuanlan.zhihu.com/p/100790268*

[4] stringify: *https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/JSON/stringify*

[5] 实现 JSON.stringify: *https://github.com/YvetteLau/Step-By-Step/issues/39#issuecomment-508327280*

[6] JSON.parse 三种实现方式: *https://github.com/youngwind/blog/issues/115#issue-300869613*

[7] Promise A+ 规范: *https://promisesaplus.com/*

[8] BAT前端经典面试问题：史上最最最详细的手写Promise教程: *https://juejin.cn/post/6844903625769091079*

[9] 100 行代码实现 Promises/A+ 规范: *https://mp.weixin.qq.com/s/qdJ0Xd8zTgtetFdlJL3P1g*

[10] 深入理解 Promise: *https://bubuzou.com/2020/10/22/promise/*

干货直达

- 【干货】996 前端人如何持续学习
- 看了就会的 Node.js 常用三方工具包
- 看了就会的 Node.js 三大基础模块常用 API
- 实现 CLI 常用工具包 - 终端交互相关（问卷、彩色文字、loading、进度条）
- 阿里腾讯面试梳理&个人成长经历分享
- 注释，今晚我不关心代码，我只想你
- 假如易立竞吐槽程序员。。。太扎心了。。。

更多精彩

在公众号对话框输入以下**关键词**

查看更多优质内容！

导图 | 简历 | **面试** | 个人成长 | **学习方法**

JavaScript | TypeScript| **Vue** | React

工程化 | 性能优化 | 设计模式 | 浏览器

据不完全统计

99%的前端开发者都关注了这个公众号 👇

公众号