

「中高级前端面试」JavaScript手写代码无敌秘籍

前端劝退师 前端工匠 Today

手写路径导航

- 实现一个new操作符
- 实现一个JSON.stringify
- 实现一个JSON.parse
- 实现一个call或 apply
- 实现一个Function.bind
- 实现一个继承
- 实现一个JS函数柯里化
- 手写一个Promise(中高级必考)
- 手写防抖(Debouncing)和节流(Throttling)
- 手写一个JS深拷贝
- 实现一个instanceOf

1. 实现一个new操作符

来源：「你不知道的javascript」 英文版

new操作符做了这些事：

- 它创建了一个全新的对象。
- 它会被执行 `[[Prototype]]`（也就是 `__proto__`）链接。
- 它使 `this` 指向新创建的对象。。
- 通过 `new` 创建的每个对象将最终被 `[[Prototype]]` 链接到这个函数的 `prototype` 对象上。
- 如果函数没有返回对象类型 `Object` (包含 `Function` , `Array` , `Date` , `RegExp` , `Error`)，那么 `new` 表达式中的函数调用将返回该对象引用。

```
function New(func) {
  var res = {};
  if (func.prototype !== null) {
    res.__proto__ = func.prototype;
  }
  var ret = func.apply(res, Array.prototype.slice.call(arguments, 1));
  if ((typeof ret === "object" || typeof ret === "function") && ret !== null) {
    return ret;
  }
  return res;
}
var obj = New(A, 1, 2);
// equals to
var obj = new A(1, 2);
```

2. 实现一个JSON.stringify

`JSON.stringify (value [, replacer [, space]])`：

- `Boolean` | `Number` | `String` 类型会自动转换成对应的原始值。
- `undefined`、任意函数以及 `symbol`，会被忽略（出现在非数组对象的属性值中时），或者被转换成 `null`（出现在数组中时）。
- 不可枚举的属性会被忽略
- 如果一个对象的属性值通过某种间接的方式指回该对象本身，即循环引用，属性也会被忽略。

```
function jsonStringify(obj) {
    let type = typeof obj;
    if (type !== "object") {
        if (/string|undefined|function/.test(type)) {
            obj = '' + obj + '';
        }
        return String(obj);
    } else {
        let json = []
        let arr = Array.isArray(obj)
        for (let k in obj) {
            let v = obj[k];
            let type = typeof v;
            if (/string|undefined|function/.test(type)) {
                v = '' + v + '';
            } else if (type === "object") {
                v = jsonStringify(v);
            }
            json.push((arr ? "" : '"' + k + '":') + String(v));
        }
        return (arr ? "[" : "{") + String(json) + (arr ? "]" : "}");
    }
}

jsonStringify({x : 5}) // '{"x":5}'
jsonStringify([1, "false", false]) // "[1,\"false\",false]"
jsonStringify({b: undefined}) // '{"b":undefined}'
```

3. 实现一个JSON.parse

```
JSON . parse ( text [, reviver ])
```

用来解析JSON字符串，构造由字符串描述的JavaScript值或对象。提供可选的reviver函数用以在返回之前对所得到的对象执行变换(操作)。

3.1 第一种：直接调用 eval

```
function jsonParse(opt) {
    return eval('(' + opt + ')');
}

jsonParse(jsonStringify({x : 5}))
// Object { x: 5}

jsonParse(jsonStringify([1, "false", false]))
// [1, "false", falsr]

jsonParse(jsonStringify({b: undefined}))
// Object { b: "undefined"}
```

避免在不必要的情况下使用 eval，eval() 是一个危险的函数，他执行的代码拥有着执行者的权利。如果你用 eval()运行的字符串代码被恶意方（不怀好意的人）操控修改，您最终可能会在您的网页/扩展程序的权限下，在用户计算机上运行恶意代码。

它会执行JS代码，有XSS漏洞。

如果你只想记这个方法，就得对参数json做校验。

```
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 + ")");
}

```

3.2 第二种：Function

来源 神奇的eval()与new Function()

核心：Function与eval有相同的字符串参数特性。

```
var func = new Function ( arg1 , arg2 , ... , functionBody );
```

在转换JSON的实际应用中，只需要这么做。

```

var jsonStr = '{ "age": 20, "name": "jack" }'
var json = (new Function('return ' + jsonStr))();

```

eval 与 Function 都有着动态编译js代码的作用，但是在实际的编程中并不推荐使用。

这里是面向面试编程，写这两种就够了。至于第三，第四种，涉及到繁琐的递归和状态机相关原理，具体可以看：

《JSON.parse 三种实现方式》

4. 实现一个call或 apply

实现改编来源：JavaScript深入之call和apply的模拟实现 #11

call 语法：

fun . call (thisArg , arg1 , arg2 , ...) , 调用一个函数, 其具有一个指定的this值和分别地提供的参数(参数的列表)。

apply 语法：

func . apply (thisArg , [argsArray]) , 调用一个函数，以及作为一个数组（或类似数组对象）提供的参数。

4.1 Function . call 按套路实现

call 核心：

- 将函数设为对象的属性
- 执行&删除这个函数
- 指定 this 到函数并传入给定参数执行函数
- 如果不传入参数，默认指向为 window

为啥说是套路实现呢？因为真实面试中，面试官很喜欢让你逐步地往深考虑，这时候你可以反套路他，先写个简单版的：

4.1.1 简单版

```
var foo = {
  value: 1,
  bar: function() {
    console.log(this.value)
  }
}
foo.bar() // 1
```

4.1.2 完善版

当面试官有进一步的发问，或者此时你可以假装思考一下。然后写出以下版本：

```
Function.prototype.call2 = function(content = window) {
  content.fn = this;
  let args = [...arguments].slice(1);
  let result = content.fn(...args);
  delete content.fn;
  return result;
}
let foo = {
  value: 1
}
function bar(name, age) {
  console.log(name)
  console.log(age)
  console.log(this.value);
}
bar.call2(foo, 'black', '18') // black 18 1
```

4.2 Function.apply的模拟实现

apply () 的实现和 call () 类似，只是参数形式不同。直接贴代码吧：

```
Function.prototype.apply2 = function(context = window) {
  context.fn = this
  let result;
  // 判断是否有第二个参数
  if(arguments[1]) {
    result = context.fn(...arguments[1])
  } else {
    result = context.fn()
  }
  delete context.fn
  return result
}
```

5. 实现一个Function.bind()

bind () 方法：

会创建一个新函数。当这个新函数被调用时，bind() 的第一个参数将作为它运行时的 this，之后的一序列参数将会在传递的实参前传入作为它的参数。（来自于 MDN）

此外，bind 实现需要考虑实例化后对原型链的影响。

```
Function.prototype.bind2 = function(content) {
    if(typeof this !== "function") {
        throw Error("not a function")
    }
    // 若没问参数类型则从这开始写
    let fn = this;
    let args = [...arguments].slice(1);

    let resFn = function() {
        return fn.apply(this instanceof resFn ? this : content, args.concat(...arguments))
    }
    function tmp() {}
    tmp.prototype = this.prototype;
    resFn.prototype = new tmp();

    return resFn;
}
```

6. 实现一个继承

寄生组合式继承

一般只建议写这种，因为其它方式的继承会在一次实例中调用两次父类的构造函数或有其它缺点。

核心实现是：用一个 `F` 空的构造函数去取代执行了 `Parent` 这个构造函数。

```
function Parent(name) {
    this.name = name;
}
Parent.prototype.sayName = function() {
    console.log('parent name:', this.name);
}
function Child(name, parentName) {
    Parent.call(this, parentName);
    this.name = name;
}
function create(proto) {
    function F(){}
    F.prototype = proto;
    return new F();
}
Child.prototype = create(Parent.prototype);
Child.prototype.sayName = function() {
    console.log('child name:', this.name);
}
Child.prototype.constructor = Child;

var parent = new Parent('father');
parent.sayName();    // parent name: father

var child = new Child('son', 'father');
```

7. 实现一个JS函数柯里化

什么是柯里化？

在计算机科学中，柯里化（Currying）是把接受多个参数的函数变换成接受一个单一参数（最初函数的第一个参数）的函数，并且返回接受余下的参数且返回结果的新函数的技术。

函数柯里化的主要作用和特点就是参数复用、提前返回和延迟执行。

7.1 通用版

```
function curry(fn, args) {
  var length = fn.length;
  var args = args || [];
  return function(){
    newArgs = args.concat(Array.prototype.slice.call(arguments));
    if (newArgs.length < length) {
      return curry.call(this,fn,newArgs);
    }else{
      return fn.apply(this,newArgs);
    }
  }
}

function multiFn(a, b, c) {
  return a * b * c;
}

var multi = curry(multiFn);

multi(2)(3)(4);
multi(2,3,4);
multi(2)(3,4);
multi(2,3)(4);
```

7.2 ES6骚写法

```
const curry = (fn, arr = []) => (...args) => (
  arg => arg.length === fn.length
    ? fn(...arg)
    : curry(fn, arg)
)([...arr, ...args])

let curryTest=curry((a,b,c,d)=>a+b+c+d)
curryTest(1,2,3)(4) //返回10
curryTest(1,2)(4)(3) //返回10
curryTest(1,2)(3,4) //返回10
```

8. 手写一个Promise(中高级必考)

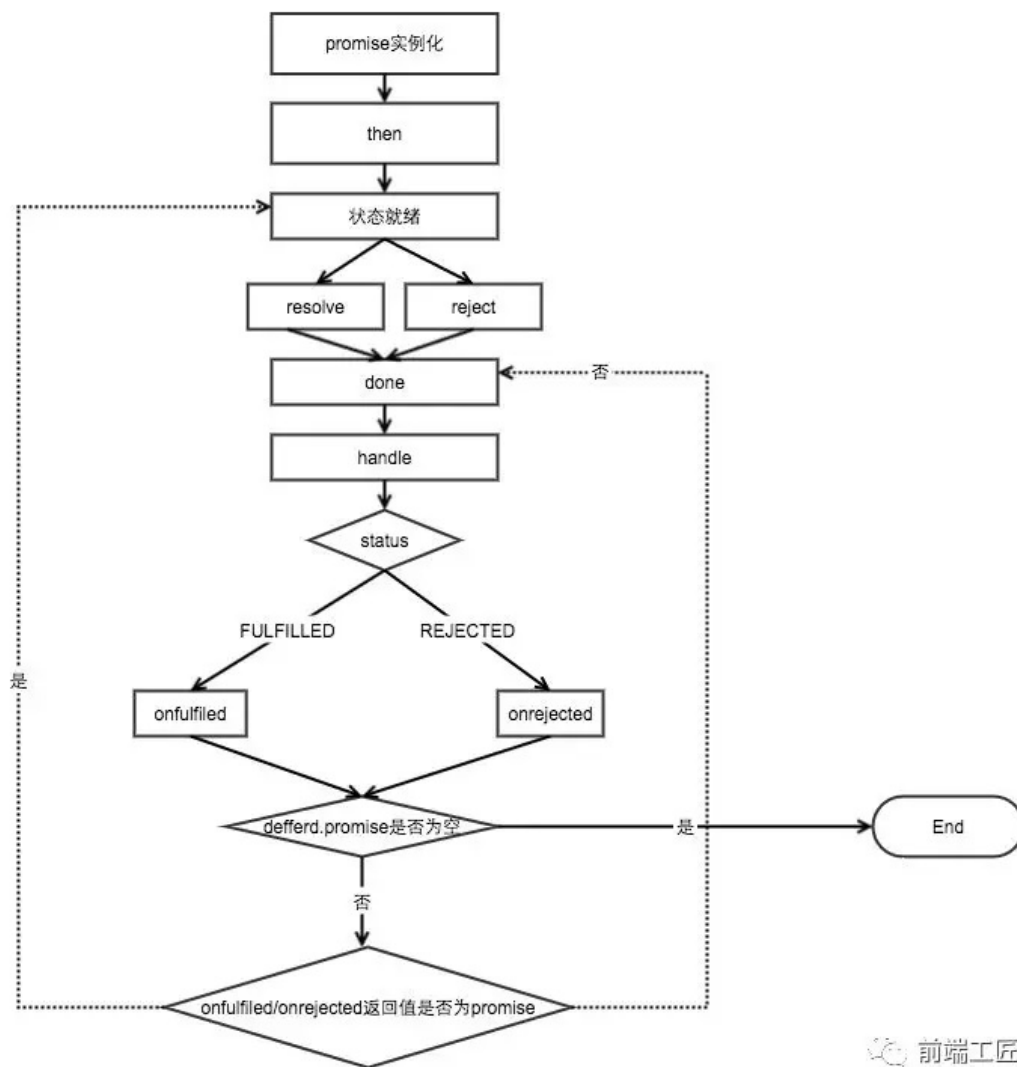
我们来过一遍 Promise / A+规范：

- 三种状态 pending | fulfilled (resolved) | rejected
- 当处于 pending 状态的时候，可以转移到 fulfilled (resolved) 或者 rejected 状态
- 当处于 fulfilled (resolved) 状态或者 rejected 状态的时候，就不可变。

1. 必须有一个 then 异步执行方法，then 接受两个参数且必须返回一个promise：

```
// onFulfilled 用来接收promise成功的值
// onRejected 用来接收promise失败的原因
promise1=promise.then(onFulfilled, onRejected);
```

8.1 Promise 的流程图分析



前端工匠

来回顾下 Promise用法:

```

var promise = new Promise((resolve,reject) => {
  if (操作成功) {
    resolve(value)
  } else {
    reject(error)
  }
})
promise.then(function (value) {
  // success
},function (value) {
  // failure
})

```

8.2 面试够用版

来源: 实现一个完美符合Promise/A+规范的Promise

```

function myPromise(constructor){
  let self=this;
  self.status="pending" //定义状态改变前的初始状态
  self.value=undefined;//定义状态为resolved的时候的状态
  self.reason=undefined;//定义状态为rejected的时候的状态
  function resolve(value){
    //两个==="pending", 保证了状态的改变是不可逆的
    if(self.status==="pending"){
      self.value=value;
      self.status="resolved";
    }
  }
}

```

```

    }
    function reject(reason){
        //两个==="pending", 保证了状态的变化是不可逆的
        if(self.status==="pending"){
            self.reason=reason;
            self.status="rejected";
        }
    }
}
//捕获构造异常
try{
    constructor(resolve,reject);
}catch(e){
    reject(e);
}
}

```

同时, 需要在 `myPromise` 的原型上定义链式调用的 `then` 方法:

```

myPromise.prototype.then=function(onFullfilled,onRejected){
    let self=this;
    switch(self.status){
        case "resolved":
            onFullfilled(self.value);
            break;
        case "rejected":
            onRejected(self.reason);
            break;
        default:
    }
}

```

测试一下:

```

var p=new myPromise(function(resolve,reject){resolve(1)});
p.then(function(x){console.log(x)})
//输出1

```

8.3 大厂专供版

直接贴出来吧, 这个版本还算好理解

```

const PENDING = "pending";
const FULFILLED = "fulfilled";
const REJECTED = "rejected";

function Promise(excutor) {
    let that = this; // 缓存当前promise实例对象
    that.status = PENDING; // 初始状态
    that.value = undefined; // fulfilled状态时 返回的信息
    that.reason = undefined; // rejected状态时 拒绝的原因
    that.onFulfilledCallbacks = []; // 存储fulfilled状态对应的onFulfilled函数
    that.onRejectedCallbacks = []; // 存储rejected状态对应的onRejected函数

    function resolve(value) { // value成功态时接收的终值
        if(value instanceof Promise) {
            return value.then(resolve, reject);
        }
    }

    // 实践中要确保 onFulfilled 和 onRejected 方法异步执行, 且应该在 then 方法被调用的那一轮事件循环之后的新执行栈中执行。
    setTimeout(() => {
        // 调用resolve 回调对应onFulfilled函数
        if (that.status === PENDING) {
            // 只能由pending状态 => fulfilled状态 (避免调用多次resolve reject)

```



```

        that.status = FULFILLED;
        that.value = value;
        that.onFulfilledCallbacks.forEach(cb => cb(that.value));
    }
});
}

function reject(reason) { // reason失败态时接收的拒因
    setTimeout(() => {
        // 调用reject 回调对应onRejected函数
        if (that.status === PENDING) {
            // 只能由pending状态 => rejected状态 (避免调用多次resolve reject)
            that.status = REJECTED;
            that.reason = reason;
            that.onRejectedCallbacks.forEach(cb => cb(that.reason));
        }
    });
}

// 捕获在excutor执行器中抛出的异常
// new Promise((resolve, reject) => {
//     throw new Error('error in excutor')
// })
try {
    excutor(resolve, reject);
} catch (e) {
    reject(e);
}
}

Promise.prototype.then = function(onFulfilled, onRejected) {
    const that = this;
    let newPromise;
    // 处理参数默认值 保证参数后续能够继续执行
    onFulfilled =
        typeof onFulfilled === "function" ? onFulfilled : value => value;
    onRejected =
        typeof onRejected === "function" ? onRejected : reason => {
            throw reason;
        };
    if (that.status === FULFILLED) { // 成功态
        return newPromise = new Promise((resolve, reject) => {
            setTimeout(() => {
                try{
                    let x = onFulfilled(that.value);
                    resolvePromise(newPromise, x, resolve, reject); // 新的promise resolve 上一个onFulfilled的返回值
                } catch(e) {
                    reject(e); // 捕获前面onFulfilled中抛出的异常 then(onFulfilled, onRejected);
                }
            });
        });
    }
    if (that.status === REJECTED) { // 失败态
        return newPromise = new Promise((resolve, reject) => {
            setTimeout(() => {
                try {
                    let x = onRejected(that.reason);
                    resolvePromise(newPromise, x, resolve, reject);
                } catch(e) {
                    reject(e);
                }
            });
        });
    }
}

```

```

if (that.status === PENDING) { // 等待态
  // 当异步调用resolve/rejected时 将onFulfilled/onRejected收集暂存到集合中
  return new Promise((resolve, reject) => {
    that.onFulfilledCallbacks.push((value) => {
      try {
        let x = onFulfilled(value);
        resolvePromise(newPromise, x, resolve, reject);
      } catch(e) {
        reject(e);
      }
    });
    that.onRejectedCallbacks.push((reason) => {
      try {
        let x = onRejected(reason);
        resolvePromise(newPromise, x, resolve, reject);
      } catch(e) {
        reject(e);
      }
    });
  });
}
};

```

9. 手写防抖(Debouncing)和节流(Throttling)

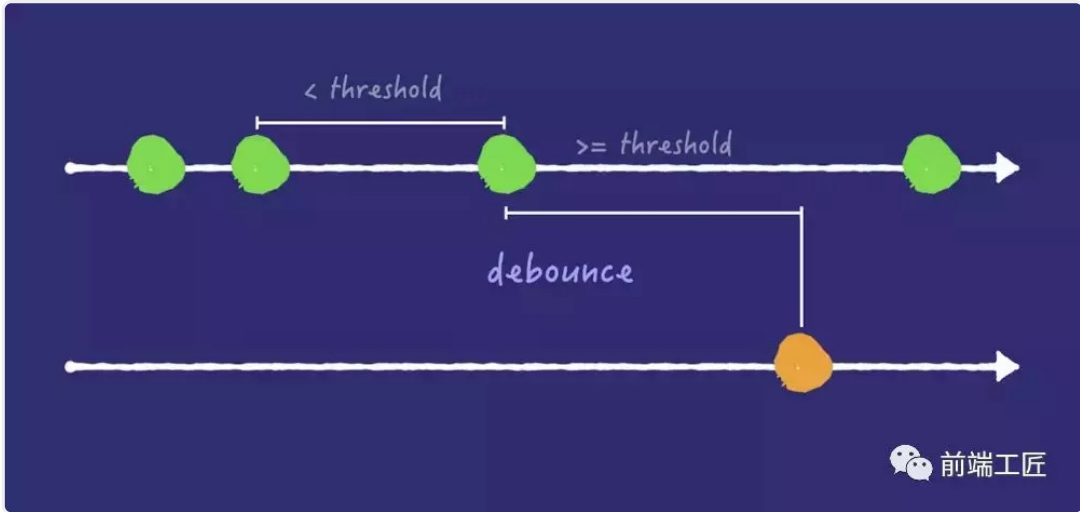
`scroll` 事件本身会触发页面的重新渲染, 同时 `scroll` 事件的 `handler` 又会被高频度的触发, 因此事件的 `handler` 内部不应该有复杂操作, 例如 `DOM` 操作就不应该放在事件处理中。针对此类高频度触发事件问题 (例如页面 `scroll`, 屏幕 `resize`, 监听用户输入等), 有两种常用的解决方法, 防抖和节流。

9.1 防抖(Debouncing)实现

典型例子: 限制 鼠标连击 触发。

一个比较好的解释是:

当一次事件发生后, 事件处理器要等一定阈值的时间, 如果这段时间过去后 再也没有 事件发生, 就处理最后一次发生的事件。假设还差 `0.01` 秒就到达指定时间, 这时又来了一个事件, 那么之前的等待作废, 需要重新再等待指定时间。



```

// 防抖动函数
function debounce(fn,wait=50,immediate) {
  let timer;
  return function() {

```

```

    if(immediate) {
        fn.apply(this,arguments)
    }
    if(timer) clearTimeout(timer)
    timer = setTimeout(()=> {
        fn.apply(this,arguments)
    },wait)
}
}

```

结合实例：滚动防抖

```

// 简单的防抖动函数
// 实际想绑定在 scroll 事件上的 handler
function realFunc(){
    console.log("Success");
}

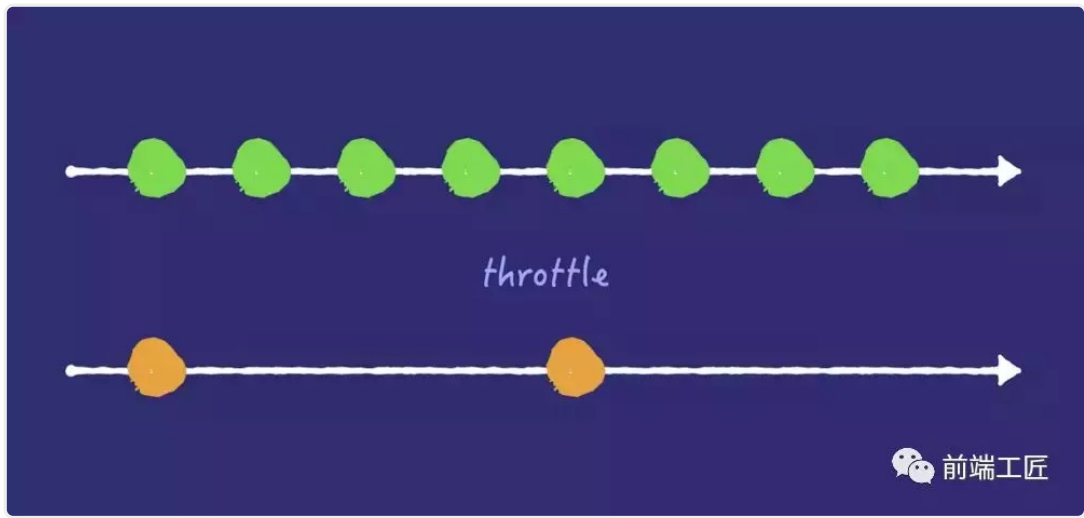
// 采用了防抖动
window.addEventListener('scroll',debounce(realFunc,500));

// 没采用防抖动
window.addEventListener('scroll',realFunc);

```

9.2 节流(Throttling)实现

可以理解为事件在一个管道中传输，加上这个节流阀以后，事件的流速就会减慢。实际上这个函数的作用就是如此，它可以将一个函数的调用频率限制在一定阈值内，例如 1s，那么 1s 内这个函数一定不会被调用两次



简单的节流函数:

```

function throttle(fn, wait) {
    let prev = new Date();
    return function() {
        const args = arguments;
        const now = new Date();
        if (now - prev > wait) {
            fn.apply(this, args);
            prev = new Date();
        }
    }
}

```

9.3 结合实践

通过第三个参数来切换模式。

```
const throttle = function(fn, delay, isDebounce) {
  let timer
  let lastCall = 0
  return function (...args) {
    if (isDebounce) {
      if (timer) clearTimeout(timer)
      timer = setTimeout(() => {
        fn(...args)
      }, delay)
    } else {
      const now = new Date().getTime()
      if (now - lastCall < delay) return
      lastCall = now
      fn(...args)
    }
  }
}
```

10. 手写一个JS深拷贝


有个最著名的乞丐版实现，在《你不知道的JavaScript（上）》里也有提及：

那么如何解决这些棘手问题呢？许多 JavaScript 框架都提出了自己的解决办法，但是 JavaScript 应当采用哪种方法作为标准呢？在很长一段时间里，这个问题都没有明确的答案。

对于 JSON 安全（也就是说可以被序列化为一个 JSON 字符串并且可以根据这个字符串解析出一个结构和值完全一样的对象）的对象来说，有一种巧妙的复制方法：

```
var newObj = JSON.parse( JSON.stringify( someObj ) );
```

当然，这种方法需要保证对象是 JSON 安全的，所以只适用于部分情况。

 前端工匠

10.1 乞丐版

```
var newObj = JSON.parse( JSON.stringify( someObj ) );
```

10.2 面试够用版

```
function deepCopy(obj){
  //判断是否是简单数据类型，
  if(typeof obj == "object"){
    //复杂数据类型
    var result = obj.constructor == Array ? [] : {};
    for(let i in obj){
      result[i] = typeof obj[i] == "object" ? deepCopy(obj[i]) : obj[i];
    }
  }else {
    //简单数据类型 直接 == 赋值
    var result = obj;
  }
  return result;
}
```

11. 实现一个instanceOf

```
function instanceOf(left,right) {


  let proto = left.__proto__;
  let prototype = right.prototype
```

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

作者掘金文章总集

- 「真香警告」重学TCP/IP 协议 与三次握手
- 「Vue实践」5分钟撸一个Vue CLI 插件
- 「Vue实践」武装你的前端项目
- 「中高级前端面试」JavaScript手写代码无敌秘籍
- 「从源码中学习」面试官都知道的Vue题目答案
- 「从源码中学习」Vue源码中的JS骚操作
- 「从源码中学习」彻底理解Vue选项Props
- 「Vue实践」项目升级vue-cli3的正确姿势
- 为何你始终理解不了JavaScript作用域链？

Forwarded from Official Account

 前端劝退师 >
