1、手写bind函数

slice() 方法返回一个新的数组对象，这一对象是一个由 begin 和 end 决定的原数组的浅拷贝（包括 begin，不包括end）。原始数组不会被改变。

```
Function.prototype.bind1 = function() {
    // 将参数解析成数组，arguments 是一个列表
    const args = Array.prototype.slice.call(arguments)

    // 获取 this （取出数组第一项，数组剩余的就是要传递的参数）
    const t = args.shift()
    const self = this // 当前函数

    // 返回一个函数
    return function() {
        // 执行原函数，并返回结果
        return self.apply(t, args)
    }
}

function fn(a, b) {
    console.log(this)
    console.log(a, b)
}
let obj = {
    name: 'haha'
}
let fn1 = fn.bind1(obj, "aaa", "ccc")
fn1()
```

手写 call 函数

```
// 手写 call
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
```

手写 apply 函数

```
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
}
```

2、手写new

(1) 创建一个新对象;
(2) 将构造函数的作用域赋给新对象（因此 this 就指向了这个新对象）；
(3) 执行构造函数中的代码（为这个新对象添加属性）；
(4) 返回新对象。

```
function Dog(name){
    this.name = name
}
Dog.prototype.sayName = function(){
    console.log(this.name)
}
// 上面是本身Dog
function _new(fn,...args){   // ...args为ES6展开符,也可以使用arguments
    //先用Object创建一个空的对象,
    const obj = Object.create(fn.prototype)  //fn.prototype代表 用当前对象的原型去创建
    //现在obj就代表Dog了,但是参数和this指向没有修改
    const rel = fn.apply(obj,args)
    //正常规定,如何fn返回的是null或undefined(也就是不返回内容),我们返回的是obj,否则返回rel
    return rel instanceof Object ? rel : obj
}
var _newDog = _new(Dog,'这是用_new出来的小狗')
_newDog.sayName()
```

3、手写一个简单的ajax

- XMLHttpRequest

```
// get请求
const xhr = new XMLHttpRequest()
xhr.open('GET', '/api', false)
xhr.onreadystatechange = function () {
    if (xhr.readyState === 4) {
        if (xhr.status === 200) {
            alert(xhr.responseText)
        }
    }
}
xhr.send()
```

readyStae状态的介绍

```
0 — （未初始化）还没有调用send()方法
1 — （载入）已调用send()方法，正在发送请求
2 — （载入完成）send()方法执行完成，已经接收到全部响应内容
3 — （交互）正在解析响应内容
4 — （完成）响应内容解析完成，可以在客户端调用了
```

4、手写深拷贝 与 for in

- for in 和 for of

for...in 语句用于遍历数组或者对象的属性（对数组或者对象的属性进行循环操作）。
for...in得到对对象的key或数组,字符串的下标

for...of和forEach一样,是直接得到值
for...of**不能对象用**

for in (以及forEach for) 是常规的同步遍历

```
for(let i in this.book) {
    this.book[i]
}
```

for...of 常用于异步的遍历

```
function deepClone(obj = {}) {
    if (typeof obj !== 'object' || typeof obj == null) {
        // obj 是 null 类型，或者不是对象也不是数组（即简单数据类型）
        return obj;
    }
    let result
    if (obj instanceof Array) {
        // 是数组
        result = []
    } else {
        // 不是数组
        result = {}
    }
    // 上面也可以使用三元表达式去表示
    // result = obj instanceof Array ? [] : {}
```

```
    for (const key in obj) {
        // 保证 key 不是原型链的属性
        if (obj.hasOwnProperty(key)) {
            // 递归
            result[key] = deepClone(obj[key])
        }
    }

    return result
}
```

比较对象是否相等

```
isObjectValueEqual(a, b) {
    var aProps = Object.getOwnPropertyNames(a);
    var bProps = Object.getOwnPropertyNames(b);
    if (aProps.length != bProps.length) {
        return false;
    }
    for (var i = 0; i < aProps.length; i++) {
        var propName = aProps[i]

        var propA = a[propName]
        var propB = b[propName]
        if ((typeof (propA) === 'object')) {
            if (this.isObjectValueEqual(propA, propB)) {
                return true
            } else {
                return false
            }
        } else if (propA !== propB) {
            return false
        } else { }
    }
    return true
}
```

5、手写数组去重

```
var arr = [1, 3, 5, 2, 4, 2, 1, 5, 8]

function fn(arr) {
    let set = new Set(arr);
    return [...set];
}

function del(arr) {
    let result = [];
    for (let i = 0; i < arr.length; i++) {
        if (result.indexOf(arr[i]) == -1) {
            result.push(arr[i]);
        }
    }
```

```
    return result;
}

console.log(del(arr))       // [ 1, 3, 5, 2, 4, 8 ]
console.log(fn(arr))        // [ 1, 3, 5, 2, 4, 8 ]
```

6、数组扁平化

```
/* ES6 */
const flatten = (arr) => {
  let result = [];
  arr.forEach((item, i, arr) => {
    if (Array.isArray(item)) {
      result = result.concat(flatten(item));
    } else {
      result.push(arr[i])
    }
  })
  return result;
};

const arr = [1, [2, [3, 4]]];
console.log(flatten(arr));

// 自己定义
function flatFn(arr) {
    let res = []
    arr.forEach(value => {
        if (Array.isArray(value)) {
            res = res.concat(flatFn(value))
        } else {
            res.push(value)
        }
    })
    return res
}
```

7、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__;
    }
}
```

8、函数柯里化

```
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);
```

防抖与节流

9、手写防抖

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width,initial-scale=1.0,maximum-
scale=1.0,user-scalable=no">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>防抖</title>
</head>
<body>
  <button id="debounce">点我防抖！</button>

  <script>
    window.onload = function() {
      // 1、获取这个按钮，并绑定事件
      var myDebounce = document.getElementById("debounce");
      myDebounce.addEventListener("click", debounce(sayDebounce));
    }

    // 2、防抖功能函数，接受传参
    function debounce(fn) {
      // 4、创建一个标记用来存放定时器的返回值
      let timeout = null;
```

```
      return function() {   // 节流函数 / 真正的事件回调函数，this 是发生事件的标签
        // 5、每次当用户点击/输入的时候，把前一个定时器清除
        clearTimeout(timeout);
        // 6、然后创建一个新的 setTimeout，
        // 这样就能保证点击按钮后的 interval 间隔内
        // 如果用户还点击了的话，就不会执行 fn 函数
        timeout = setTimeout(() => {
          fn.call(this, arguments);
        }, 1000);
      };
    }

    // 3、需要进行防抖的事件处理
    function sayDebounce() {
      // ... 有些需要防抖的工作，在这里执行
      console.log("防抖成功！");
    }

  </script>
</body>
</html>
```

10、手写节流

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width,initial-scale=1.0,maximum-
scale=1.0,user-scalable=no">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>节流</title>
</head>
<body>

  <button id="throttle">点我节流！</button>

  <script>
    window.onload = function() {
      // 1、获取按钮，绑定点击事件
      var myThrottle = document.getElementById("throttle");
      myThrottle.addEventListener("click", throttle(sayThrottle));
    }

    // 2、节流函数体
    function throttle(fn) {
      // 4、通过闭包保存一个标记
      let canRun = true;
      return function() {
        // 5、在函数开头判断标志是否为 true，不为 true 则中断函数
        if(!canRun) {
          return;
        }
        // 6、将 canRun 设置为 false，防止执行之前再被执行
        canRun = false;
        // 7、定时器
        setTimeout( () => {
```

```
            fn.call(this, arguments);
            // 8、执行完事件（比如调用完接口）之后，重新将这个标志设置为 true
            canRun = true;
          }, 1000);
        };
      }

      // 3、需要节流的事件
      function sayThrottle() {
        console.log("节流成功！");
      }

    </script>
</body>
</html>
```

11、懒加载