一、JavaScript基础

1. Object.create

思路：新对象原型指向传入对象，从而继承传入对象的属性和方法

```
Object.prototype.MyCreate = function (obj) {
  function F() {}
  F.prototype = obj;
  return new F();
};

const personPrototype = {
  greeting() {
    console.log(`Hello, my name is ${this.name}.`);
  },
};
const john = Object.MyCreate(personPrototype);
john.name = "John Doe";
john.greeting();
```

2. new 操作符

- 通过`Object.create`创建一个空对象继承传入构造函数的原型。
- 指向构造函数，使其`this`指向新建的对象

```
function objectFactory(constructor, ...args) {
  // 设置原型，将对象的原型设置为函数的 prototype 对象
  const obj = Object.create(constructor.prototype);
  // 将 this 指向新建对象，并执行函数
  const result = constructor.apply(obj, args);
  return typeof result === "object" && result !== null ? result : obj;
}

function Person(name) {
  this.name = name;
}
const well = objectFactory(Person, "well");
console.log(well.name); // well
```

3. instanceof

1. 首先获取类型的原型
2. 然后获得对象的原型
3. 然后一直循环判断对象的原型是否等于类型的原型，直到对象原型为`null`，因为原型链最终为`null`

```
function myInstanceof (left, right) {
    let proto = Object.getPrototypeOf(left)
    let constructor = right.prototype
    while (true) {
```

```
        if (!proto) return false
        if (proto === constructor) return true
        proto = Object.getPrototypeOf(proto)
    }
}

console.log([] instanceof Array) // true
console.log(2 instanceof Number) // fasle
console.log(myInstanceof([], Array)) // true
console.log(myInstanceof(2, Number)) // true
```

4. debounce

函数防抖是指在事件被触发 n 秒后再执行回调，如果在这 n 秒内事件又被触发，则重新计时。这可以使用在一些点击请求的事件上，避免因为用户的多次点击向后端发送多次请求。

```
function debounce(fn, wait) {
    let timer = null
    // 利用箭头函数this指向定义处
    return (...args) => {
        if (timer) {
            clearTimeout(timer)
            timer = null
        }
        timer = setTimeout(() => {
            fn.apply(this, args)
        }, wait)
   }
}
```

5. throttle

函数节流是指规定一个单位时间，在这个单位时间内，只能有一次触发事件的回调函数执行，如果在同一个单位时间内某事件被触发多次，只有一次能生效。节流可以使用在 scroll 函数的事件监听上，通过事件节流来降低事件调用的频率。

```
function throttle(fn, delay) {
    let startTime = Date.now()
    return (...args) => {
        let nowTime = Date.now()
        if (nowTime - startTime >= delay) {
            startTime = nowTime
            return fn.apply(this, args)
        }
    }
}
```

6.类型判断

主要是通过typeof 和 Object.prototype.toString.call()来判断，由于typeof null 也等于 'object',所以才做了多一步的处理

```
function getType (value) {
    // 如果是null则返回
```

```
    if (value === null) {
        return value + ''
    }
    // 如果是数组、对象
    if (typeof value === 'object') {
        // [object Array]
        let valueClass = Object.prototype.toString.call(value)
        let type = valueClass.split(' ')[1].split('')
        type.pop()
        // ['A', 'r', 'r', 'a', 'y']
        return type.join('').toLowerCase()
    } else {
        return typeof value
    }
}

console.log(getType([])) // array
console.log(getType(2)) // number
```

7.call

核心思想：把调用方法作为contex的属性去执行，并把得到的结果返回

```
Function.prototype.MyCall = function (context, ...args) {
  context = context || window;
  context.fn = this;
  const result = context.fn(...args);
  delete context.fn;
  return result;
};

const Person = {
  name: "well",
};
function sayName(age) {
  return `name:${this.name},age:${age}`;
}
console.log(sayName.MyCall(Person, 18)); // name:well,age:18
```

8.apply

核心思想：把调用方法作为contex的属性去执行，并把得到的结果返回。

```
Function.prototype.MyApply = function (context, ...args) {
  context = context || window;
  context.fn = this;
  const result = context.fn(...args);
  delete context.fn;
  return result;
};

const Person = {
  name: "well",
};
function sayName(age) {
  return `name:${this.name},age:${age}`;
```

```
}
console.log(sayName.MyApply(Person, [18]));
```

9.bind

bind 函数的实现步骤：

1. 判断调用对象是否为函数，即使我们是定义在函数的原型上的，但是可能出现使用 call 等方式调用的情况。
2. 保存当前函数的引用，获取其余传入参数值。
3. 创建一个函数返回
4. 函数内部使用 apply 来绑定函数调用，需要判断函数作为构造函数的情况，这个时候需要传入当前函数的 this 给 apply 调用，其余情况都传入指定的上下文对象。

```
Function.prototype.myBind = function (context, ...args) {
  context = context || window;
  const fn = this;
  return function Fn() {
    // 根据调用方式，传入不同绑定值
    return fn.apply(
      this instanceof Fn ? this : context,
      args.concat(...arguments)
    );
  };
};

const foo = {
  a: 1,
  log(x, y) {
    console.log(this.a, x, y);
  },
};
const obj = {
  a: 10,
};
foo.log.myBind(obj, 5, 6)(); // 10, 5, 6
```

10. 实现浅拷贝

浅拷贝是指，一个新的对象对原始对象的属性值进行精确地拷贝，如果拷贝的是基本数据类型，拷贝的就是基本数据类型的值，如果是引用数据类型，拷贝的就是内存地址。如果其中一个对象的引用内存地址发生改变，另一个对象也会发生变化。

- Object.assign()
- 扩展运算符
- 数组方法实现数组浅拷贝

10.1 Object.assign()

```
let target = {a: 1};
let object2 = {b: 2};
let object3 = {c: 3};
Object.assign(target,object2,object3);
console.log(target);  // {a: 1, b: 2, c: 3}
```

10.2 扩展运算符

```
let obj4 = {a:1,b:{c:1}}
let obj5 = {...obj4} // { a: 1, b: { c: 1 } }
console.log(obj5)
```

10.3 数组方法实现数组浅拷贝

```
// Array.prototype.slice
let arr = [1,2,3,4];
console.log(arr.slice()); // [1,2,3,4]
console.log(arr.slice() === arr); //false

// Array.prototype.concat
let arr2 = [1,2,3,4];
console.log(arr2.concat()); // [1,2,3,4]
console.log(arr2.concat() === arr); //false
```

10.4 浅拷贝的实现

```
function shallowCopy(object) {
    // 只拷贝对象
    if (!object || typeof object !== 'object') return
    // 根据object 的类型判断新建的是数组还是对象
    let newObject = Array.isArray(object) ? [] : {}
    // 遍历object，判断是object的属性才拷贝
    for (let key in object) {
        if (object.hasOwnProperty(key)) {
            newObject[key] = object[key]
        }
    }
    return newObject
}
const myObject = [1,2,3,4,5]
const newShallowObject = shallowCopy(myObject)
console.log(newShallowObject) // [ 1, 2, 3, 4, 5 ]
```

11. 实现深拷贝

深拷贝相对浅拷贝而言，如果遇到属性值为引用类型的时候，它新建一个引用类型并将对应的值复制给它，因此对象获得的一个新的引用类型而不是一个原有类型的引用。深拷贝对于一些对象可以使用 JSON 的两个函数来实现，但是由于 JSON 的对象格式比 js 的对象格式更加严格，所以如果属性值里边出现函数或者 Symbol 类型的值时，会转换失败。

- JSON.stringify
- lodash的_.cloneDeep
- 手动实现深拷贝函数

```
const obj1 = {
    a: 1,
    b: { f: { g: 1 } },
    c: [1, 2, 3]
}
```

```
// JSON.stringfy
const obj2 = JSON.parse(JSON.stringify(obj1))
obj2.b.e =2
console.log(obj1) // { a: 1, b: { f: { g: 1 } }, c: [ 1, 2, 3 ] }
console.log(obj2) // { a: 1, b: { f: { g: 1 }, e: 2 }, c: [ 1, 2, 3 ] }

// 深拷贝的实现
function deepCopy (object) {
    // 只拷贝对象
    if (!object || typeof object !== 'object') return
    // 根据object 的类型判断新建的是数组还是对象
    let newObject = Array.isArray(object) ? [] : {}
    // 遍历 object
    for (let key in object) {
        if (object.hasOwnProperty(key)) {
            newObject[key] = typeof object[key] === 'object' ?
deepCopy(object[key]) : object[key]
        }
    }
    return newObject
}
const obj3 = deepCopy(obj1)
obj3.b.f.g = 5
console.log(obj1) // { a: 1, b: { f: { g: 1 } }, c: [ 1, 2, 3 ] }
console.log(obj3) // { a: 1, b: { f: { g: 5 } }, c: [ 1, 2, 3 ] }
```

12.实现 sleep 函数

```
function sleep (wait) {
    return new Promise(resolve => {
        setTimeout(resolve, wait)
    })
}
const curTime = Date.now()
sleep(3000).then(() => {
    console.log(Date.now() - curTime) // 3000
})
```

13. 实现 Object.assign

```
Object.myAssign = function (target, ...source) {
    if (target === null) {
        throw new TypeError('error')
    }
    let ret = Object(target)
    source.forEach(obj => {
        if (obj !== null) {
            for (let key in obj) {
                if (obj.hasOwnProperty(key)) {
                    ret[key] = obj[key]
                }
            }
        }
    })
    return ret
```

```
}

let target = {a: 1}
let object2 = {b: 2}
let object3 = {c: 3}
Object.myAssign(target,object2,object3)
console.log(target);  // {a: 1, b: 2, c: 3}
```

14.手写 Promise

14.1声明 Promise 类 & then 的基础构建

```
const PENDING = 'pengding'
const FULFILLED = 'fulfilled'
const REJECTED = 'rejected'

class myPromise {
    constructor (executor) {
        // 保存 promise 的状态
        this.state = PENDING
        // 成功结果
        this.value = undefined
        // 失败结果
        this.reason = undefined
        // resolve 方法
        const resolve = (value) => {
            if (this.state = PENDING) {
                this.state = FULFILLED
                this.value = value
            }
        }
        const reject = (reason) => {
            if (this.state = PENDING) {
                this.state = REJECTED
                this.reason = reason
            }
        }
        try {
            executor(resolve, reject)
        } catch (e) {
            reject(e)
        }
    }
    // then 方法
    then (onFulfilled, onRejected) {
        // 如果状态是 fulfilled，则执行then传入的 onFulfilled 函数
        if (this.state === FULFILLED) {
            typeof onFulfilled === 'function' && onFulfilled(this.value)
        }
        // 如果状态是 fulfilled，则执行then传入的 onRejected 函数
        if (this.state === REJECTED) {
            typeof onRejected === 'function' && onRejected(this.reason)
        }
    }
}
```

```
const promise = new myPromise ((resolve, reject) => {
    resolve(1)
})
console.log(promise) // myPromise { state: 'fulfilled', value: 1, reason:
undefined }
promise.then((res) => console.log(res)) // 1

// 漏洞
const promiseError = new myPromise((resolve, reject) => {
    console.log('执行')
    setTimeout(() => {
        reject(3)
    })
})
console.log(promiseError) // myPromise { state: 'pengding', value: undefined,
reason: undefined }
promiseError.then(res => console.log(res), err => console.log(err))
```

- resolve: 把state 变为 fulfilled， 改变value
- reject：把state 变为 rejected，改变reason
- 由于setTimeout是宏任务，放入宏任务队列，执行了下面的then，由于还没有resolve或者reject，所以状态还是pending。

14.2 then 进一步优化

参考发布订阅模式，在执行then的时候，如果当时还是 pending 状态，就把回调函数寄存到一个数组中，当状态发生改变时，去数组中取出回调函数。

```
class myPromise {
    constructor (executor) {
        ...
        // 成功的回调
        this.onFulfilled = []
        // 失败的回调
        this.onRejected = []
        // resolve 方法
        const resolve = (value) => {
            if (this.state = PENDING) {
                this.state = FULFILLED
                this.value = value
                // 执行成功的回调
                this.onFulfilled.forEach(fn => fn(value))
            }
        }
        const reject = (reason) => {
            if (this.state = PENDING) {
                this.state = REJECTED
                this.reason = reason
                // 执行失败的回调
                this.onRejected.forEach(fn => fn(reason))
            }
        }
    }
    then (onFulfilled, onRejected) {
        ...
        // 如果状态是 pending，不是马上执行回调函数，而是将其存储起来
```

```
        if (this.state === PENDING) {
            typeof onFulfilled === 'function' &&
this.onFulfilled.push(onFulfilled)
            typeof onRejected === 'function' && this.onRejected.push(onRejected)
        }
    }
}
```

```
const promise = new myPromise((resolve, reject) => {
    setTimeout(() => {
        resolve(1)
    }, 1000)
})
promise.then(res => console.log(res)) // 1
promise.then(res => console.log(res)) // 1
```

原生的promise.then()中的代码是异步执行的，所以需要进一步优化，否则出现下面代码执行顺序

```
const promise = new myPromise((resolve, reject) => {
    resolve(1)
})
promise.then(res => console.log(res)) // 1
promise.then(res => console.log(res)) // 1
console.log(2)
```

- 1
- 1
- 2

```
then (onFulfilled, onRejected) {
        if(typeof onFulfilled !== 'function') onFulfilled = () => {}
        if(typeof onRejected !== 'function') onRejected = () => {}

        // 如果状态是 pending，不是马上执行回调函数，而是将其存储起来
        if (this.state === PENDING) {
            this.onFulfilled.push(
                () => {
                    setTimeout(() => onFulfilled(this.value))
                }
            )
            this.onRejected.push(
                () => {
                    setTimeout(() => onRejected(this.reason))
                }
            )
        }
        // 如果状态是 fulfilled，则执行then传入的 onFulfilled 函数
        if (this.state === FULFILLED) {
            setTimeout(() => onFulfilled(this.value))
        }
        // 如果状态是 fulfilled，则执行then传入的 onRejected 函数
        if (this.state === REJECTED) {
            setTimeout(() => onRejected(this.reason))
        }
    }
```

14.3 链式调用

- promise 是支持链式调用的，就是 .then() 之后还可以继续 .then()
- 所以 then 返回的应该还是一个 promise 对象，并且在这个返回的promise 中就调用了 resolve 或者 reject方法，改变了state，这样的话下一个then 的回调就可以获取到 value 或者 reason
- 由于promise 可以穿透，即前面的then不传入回调，后面的then的回调依然能接收到 value 或者 reason，所以 then 的实现中，如果没有传入回调函数，则定义一下回调函数即可
- 如果在 then 中发生了错误，则返回的promise对象的状态应该是调用了 reject 方法，把 state 改成了 rejected 状态的。

```
then (onFulfilled, onRejected) {
        if(typeof onFulfilled !== 'function') onFulfilled = () => {}
        if(typeof onRejected !== 'function') onRejected = () => {}
        return new myPromise((resolve, reject) => {
            // 如果状态是 pending，不是马上执行回调函数，而是将其存储起来
            if (this.state === PENDING) {
                this.onFulfilled.push(
                    () => {
                        setTimeout(() => resolve(onFulfilled(this.value)))
                    }
                )
                this.onRejected.push(
                    () => {
                        setTimeout(() => resolve(onRejected(this.reason)))
                    }
                )
            }
            // 如果状态是 fulfilled，则执行then传入的 onFulfilled 函数
            if (this.state === FULFILLED) {
                setTimeout(() => resolve(onFulfilled(this.value)))
            }
            // 如果状态是 fulfilled，则执行then传入的 onRejected 函数
            if (this.state === REJECTED) {
                setTimeout(() => resolve(onRejected(this.reason)))
            }
        })
    }
```

```
const promise = new myPromise((resolve, reject) => {
    resolve(1)
})
promise
    .then(res => {
        console.log(res)
        return res
    })
    .then(res => console.log(res))
// 输出： 1 1
```

处理then 穿透

原生：promise.then().then(res => console.log(res))中依然可以拿到前面传递过来的参数，这里就是then的穿透。

实现 then 的穿透也非常简单，更改一下 onFulfilled 和 onRejected 不是函数的情况的处理即可：

```
then (onFulfilled, onRejected) {
    if(typeof onFulfilled !== 'function') onFulfilled = value => value
    if(typeof onRejected !== 'function') onRejected = reason => {throw reason}
    ...
}
```

异常处理

如果在then中出现了错误，需要返回的下一个promise 的 state 变为 rejected，所以需要添加异常处理

```
try {
    resolve(onFulfilled(this.value))
} catch(e) {
    reject(e)
}
```

14-4 封装 resolvePromise 来处理 promise

在前面我们已经基本完成了 then，而一些特殊情况依旧会造成问题：

1. 循环引用自身

   在原生Promise中，如果一个promise的onResolved返回了自身，比如这样

   ```
   const promise =  new Promise((resolve, reject) => {
     resolve()
   })
   const p = promise.then(() => p)
   // Uncaught (in promise) TypeError: A promise cannot be resolved with
   itself.
   ```

2. onResolved 返回了一个 promise 对象

   ```
   new Promise((resolve, reject) => {
     resolve()
   }).then(() => {
       return new Promise((resolve, reject) => {
       resolve('hi')
     })
   }).then(res => console.log(res))
   ```

   在原生 Promise 中，当 onResolved 返回了一个 promise 对象时，会将其 resolve 或 reject 的值传递到下一个 then, 所以打印结果是 'hi'

3. onResolved 返回了一个嵌套的 promise 对象

   ```
   new Promise((resolve, reject) => {
     resolve()
   }).then(() => {
     return new Promise((resolve, reject) => {
       resolve(new Promise((resolve, reject) => {
         resolve('hi')
       }))
     })
   }).then(res => console.log(res)) // hi
   ```

二、数据处理

15.实现日期格式化函数

```
const dateFormat = function (dateInput, format) {
    const day = dateInput.getDate()
    const month = dateInput.getMonth() + 1
    const year = dateInput.getFullYear()
    format = format.replace(/yyyy/, year)
    format = format.replace(/MM/, month)
    format = format.replace(/dd/, day)
    return format
}

console.log(dateFormat(new Date('2020-12-01'), 'yyyy/MM/dd')) // 2020/12/01
console.log(dateFormat(new Date('2020-04-01'), 'yyyy/MM/dd')) // 2020/04/01
console.log(dateFormat(new Date('2020-04-01'), 'yyyy年MM月dd日')) // 2020年04月01日
```

16. 实现数组的乱序输出

```
const arr = [1,2,3,4,5,6,7,8,9,10]

let length = arr.length
let randomIndex
while (length) {
    randomIndex = Math.floor(Math.random() * length)
    length--
    [arr[length], arr[randomIndex]] = [arr[randomIndex], arr[length]]
}
console.log(arr)
```

17.实现数组元素的求和

```
let arr = [1,2,3,4,5,6,7,8,9,10]
// reduce
let sum = arr.reduce((total, i) => total += i, 0)
console.log(sum) // 5

// 递归
function add (arr) {
    if (arr.length === 1) return arr[0]
    return arr[0] + add(arr.slice(1))
}
console.log(add(arr)) // 5
```

18.数组扁平化

- 递归实现
- 迭代实现
- 扩展运算符实现
- split 和 toString

- ES6 flat

```
let arr = [1, [2, [3, 4, 5]]]
// 递归实现
function flatten (arr) {
    let result = []
    for (let i=0; i<arr.length; i++) {
        if (Array.isArray(arr[i])) {
            result = result.concat(flatten(arr[i]))
        } else {
            result.push(arr[i])
        }
    }
    return result
}
console.log(flatten(arr)) // [ 1, 2, 3, 4, 5 ]

// 迭代实现
function flattenReduce (arr) {
    return arr.reduce((previousValue, currentValue) => {
        return previousValue.concat(Array.isArray(currentValue) ?
flattenReduce(currentValue) : currentValue)
    }, [])
}
console.log(flattenReduce(arr)) // [ 1, 2, 3, 4, 5 ]

// 扩展运算符实现
function flattenExtension (arr) {
    while(arr.some(item => Array.isArray(item))) {
        arr = [].concat(...arr)
    }
    return arr
}
console.log(flattenExtension(arr)) // [ 1, 2, 3, 4, 5 ]

// split 和 toString
function flattenByString (arr) {
    return arr.toString().split(",")
}
console.log(flattenByString(arr)) // [ '1', '2', '3', '4', '5' ]

// ES6 flat
console.log(arr.flat(Infinity)) // [ 1, 2, 3, 4, 5 ]
```

19. 数组去重

- set
- map

```
const array = [1, 2, 3, 5, 1, 5, 9, 1, 2, 8]

// Array.from(new Set(arr))
console.log(Array.from(new Set(array))) // [ 1, 2, 3, 5, 9, 8 ]

// map
function uniqueArray (arr) {
```

```
    let map = new Map()
    let res = []
    for (let i=0; i<arr.length; i++) {
        if (!map.has(arr[i])) {
            map.set(arr[i], 1)
            res.push(arr[i])
        }
    }
    return res
}
console.log(uniqueArray(array)) // [ 1, 2, 3, 5, 9, 8 ]
```

20. flat 实现

```
function _flat (arr, depth) {
    if (!Array.isArray(arr) || depth <= 0) {
        return arr
    }
    return arr.reduce((previousVal, currentVal) => {
        if (Array.isArray(currentVal)) {
            return previousVal.concat(_flat(currentVal, depth - 1))
        } else {
            return previousVal.concat(currentVal)
        }
    }, [])
}

let arr = [1, [2, [3, 4, 5]]]
console.log(_flat(arr, 1)) // [ 1, 2, [ 3, 4, 5 ] ]
console.log(arr.flat(1)) // [ 1, 2, [ 3, 4, 5 ] ]
```

21. push 实现

```
Array.prototype.myPush = function (...args) {
    for (let i=0; i<args.length; i++) {
        this[this.length] = args[i]
    }
    return this.length
}
const arr = [1, 2, 3]
const ret = arr.push(4, 5, 6)
console.log(ret) // 6
console.log(arr) // [ 1, 2, 3, 4, 5, 6 ]
```

22. filter 实现

```
Array.prototype.myFilter = function (fn) {
    if (typeof fn !== 'function') {
        throw TypeError('参数必须是一个函数')
    }
    let res = []
    for (let i=0; i<this.length; i++) {
        fn(this[i]) && res.push(this[i])
    }
    return res
}

const arr = [1, 2, 3, 4, 5, 6]
console.log(arr.myFilter(item => item>3)) // [ 4, 5, 6 ]
```

23. map 实现

```
Array.prototype.myMap = function (fn) {
    if (typeof fn !== 'function') {
        throw TypeError('参数必须是一个函数')
    }
    const res = []
    for (let i=0; i<this.length; i++) {
        res.push(fn(this[i]))
    }
    return res
}

const arr = [1, 2, 3, 4, 5, 6]
console.log(arr.myMap(item => item * item)) // [ 1, 4, 9, 16, 25, 36 ]
```

24.repeat 实现

- 冒泡实现
- 迭代实现

```
function repeat(s, n) {
    if (n > 0) {
        return s + repeat(s, --n)
    } else {
        return ''
    }
}

function repeatReduce (s, n) {
    while (n > 1) {
        s += s
        n--
    }
    return s
}

console.log(repeat('abc', 2)) // abcabc
console.log(repeatReduce('abc', 2)) // abcabc
```

25.柯里化-参数长度不确定

```
// 参数长度不固定
function currying (fn) {
    let args = []
    return function temp (...newArgs) {
        if (newArgs.length) {
            args = [
                ...args,
                ...newArgs
            ]
            return temp
        } else {
            let val = fn.apply(this, args)
            args = [] //保证再次调用时清空
            return val
        }
    }
}

function add (...args) {
    //求和
    return args.reduce((a, b) => a + b)
}
function getSum (a,b,c) {
    return a+b+c
}
let addCurry = currying(add)
let getSumCurry = currying(getSum)
console.log(addCurry(1)(2)(3)(4, 5)())  //15
console.log(addCurry(1)(2)(3, 4, 5)())  //15
console.log(addCurry(1)(2, 3, 4, 5)())  //15

console.log(getSumCurry(1,2,3)()) // 6
console.log(getSumCurry(1)(2)(3)()) // 6
console.log(getSumCurry(1,2)(3)()) // 6
```

26.柯里化-参数长度确定

```
// 参数长度固定
function curry (func) {
    return function curriedFn(...args) { // 使用剩余参数接收实参
        //  如果实参小于形参，递归执行(func.length: 传入函数的参数长度)
        if (args.length < func.length) {
            return function () {
                // argument 是再次调用的实参，需转换为数组然后拼接之前转为参数
                return curriedFn(...[...args, ...arguments])
            }
        }
        // 如果实参等于形参直接执行
        return func(...args)
    }
}
```

27. 函数组合

```
function composeRight(...args) {
    return function(value) {
        let res = value
        for (let i=args.length-1; i>=0; i--) {
            res = args[i](res)
        }
        return res
    }
}
function composeRightReduce(...args) {
    return function(value) {
        // reduce:对数组中的每一个元素执行提供的函数，并汇总成单个结果
        return args.reverse().reduce(function(acc,fn) {
            return fn(acc)
        },value) // 把value作为acc的初始值
    }
}

const reverse = arr => arr.reverse()
const first = arr => arr[0]
const toUpper = s => s.toUpperCase()

const f = composeRight(toUpper,first,reverse)
const fReduce = composeRightReduce(toUpper,first,reverse)
console.log(f(['one','two','three'])) // THREE
console.log(fReduce(['one','two','three'])) // THREE
```

三、场景应用

28.红蓝绿循环打印

```
function red() {
    console.log('red');
}
function green() {
    console.log('green');
}
function yellow() {
    console.log('yellow');
}
// 回调函数实现
const task = (wait, light, callback) => {
    setTimeout(() => {
        switch (light) {
            case 'red':
                red()
                break
            case 'green':
                green()
                break
            case 'yellow':
                yellow()
                break
```

```
        }
        callback()
    }, wait)
}
const step = () => {
    task(3000, 'red', () => {
        task(2000, 'green', () => {
            task(1000, 'yellow', step)
        })
    })
}
// step()

// promise 实现
const taskPromise = (wait, light) => {
    return new Promise (resolve => {
        setTimeout(() => {
            switch (light) {
                case 'red':
                    red()
                    break
                case 'green':
                    green()
                    break
                case 'yellow':
                    yellow()
                    break
            }
            resolve()
        }, wait)
    })
}
const setpPromise = () => {
    taskPromise(3000, 'red')
        .then(() => {
            taskPromise(2000, 'green')
        })
        .then(() => {
            taskPromise(1000, 'yellow')
        })
        .then(() => {
            setpPromise()
        })
}
// setpPromise()

const stepRunner = async () => {
    await taskPromise(3000, 'red')
    await taskPromise(2000, 'green')
    await taskPromise(1000, 'yellow')
    stepRunner()
}
stepRunner()
```

29.间隔打印

```
for (let i=0; i<5; i++) {
    setTimeout(() => {
        console.log(i)
    }, i * 1000)
}
```

30.ES6创建类

```
class Employee {
    constructor (name, dept) {
        this.name = name
        this.dept = dept
        this.age = 18
    }
    // 静态方法
    static fun () {
        console.log('static')
    }
    getName () {
        console.log(this.name)
    }
}

Employee.fun() // static

const well  = new Employee('well', 'dev')
console.log(well) // Employee { name: 'well', dept: 'dev', age: 18 }
// well.fun() // well.fun is not a function
well.getName()

// extends继承父类创建子类
class Manager extends Employee {
    constructor (name, dept, reports) {
        super(name, dept)
        this.reports = reports
    }
}
const wellManager = new Manager('wellManager', 'dev', 1)
Manager.fun() // static
console.log(wellManager) // Manager { name: 'wellManager', dept: 'dev', age: 18,
reports: 1 }
wellManager.getName() // wellManager
```

- constructor：构造函数，相当于ES5的构造函数，里面的this.×××的属性可以实例化给对象
- static：静态属性，不会随着实例化给对象，但是可以通过extends继承。
- 非 static 方法可以随着实例化给对象。

31.ES5 创建类

```
function Employee (name, dept) {
    this.name =  name
    this.dept = dept
    this.age = 18
}
// 静态方法
```

```
Employee.fun = function () {
    console.log('static')
}
Employee.prototype.getName = function () {
    console.log(this.name)
}
const well = new Employee('well', 'dev')
console.log(well) // Employee { name: 'well', dept: 'dev', age: 18 }
Employee.fun() // static
// well.fun() // Employee.fun is not a function
well.getName()

// 继承
function Manager(name, dept, reports) {
    // 调用 Employee 函数，并把this执行Manger，所以完成了
    // this.name = name
    // this.dept = dept
    Employee.call(this, name, dept)
    this.reports = reports
}

const wellManager = new Manager('wellManager', 'dev', 1)
// Manager.fun() // Manager.fun is not a function
console.log(wellManager) // Manager { name: 'wellManager', dept: 'dev', age: 18,
reports: 1 }
// wellManager.getName() // wellManager.getName is not a function
```

- 静态属性添加
- 原型属性添加
- 继承实现
    - 继承不了原型属性
    - 继承不了静态属性