[TOC]

二、javaScript

js 标签的属性

- defer 在外部文件中使用，推迟执行 js 脚本，虽然立即下载脚本，但是在页面加载完才执行，不影响页面加载，有顺序影响，若多个推迟执行脚本，会按照顺序执行
- async 在外部文件中使用，异步执行脚本，虽然立即下载脚本，但是在页面加载完才执行，不影响页面加载，没有顺序影响，若有多个异步执行，不能确保执行顺序
- type 默认为 'text/javascript' ，当使用 export 和 import 时。可能会修改，变成 'module'
- src 设置外部文件的源，脚本没有跨域限制
- integrity 因为没有跨域限制，所以为了防止同一个源的 js 文件被恶意修改，integrity 可以设置一个签名，对比 js 文件返回的签名，若不同则报错

defer 和 async 的区别

异步没有顺序可言。延迟还受顺序影响

动态加载脚本

- 浏览器预加载器 浏览器在加载资源时分为五个优先级:Highest、High、Medium、Low、Lowest。其中 css 和 html 优先级最高；font 字体资源，优先级分别为 Highest/High；图片资源，如果出现在视口中，则优先级为 High，否则为 Low；而 script 脚本优先级不一：
    - 网络在**第一个图片资源**之前阻塞的脚本在网络优先级中是 High
    - 网络在第一个图片资源之后阻塞的脚本在网络优先级中是 Medium
    - 异步/延迟/插入的脚本（无论在什么位置）在网络优先级中是 Low 因为浏览器有严格的加载优先级，使用动态脚本时，会影响加载优先级，需设置浏览器预加载器，即 preload，提前加载所需要的资源。link 标签有一个 rel 属性，赋值为 preload 时表示为预加载器，可以加载任意资源，此外还有 as 属性，表示此类资源是什么类型，以便分优先级。此外还有一个 prefetch

```
<link rel='preload' href='index.js' as="script"> // 预加载 index.js,且此资源是 script 脚本
```

动态加载脚本即需要时才加载此脚本，但是**浏览器预加载器**不知道，需要设置 '',表示将来会加载该地址(href)的资源，让浏览器有所准备，提前下载 1，创建一个 script 节点 2，给该节点添加属性 3，将节点加入到 dom 结构中

```
let script = document.createElement('script')
script.src = 'index.js'
document.head.appendChild(script)
```

js 补充

- 行内脚本(即在本文件内使用 script 标签)的缺点 1，不能使用''字符串，当做脚本结束标签，需要使用转义字符变成'</script>' 2，在 XTML 标签中，将 < (小于号)当做标签 3，多个页面使用同一段代码时，造成资源浪费 4，在 head 标签中使用脚本，脚本代码没有加载完，页面也不会执行，但是设置了 defer/async 的外部脚本文件不会影响页面加载
- 外部文件脚本的优点 1，多个页面使用，文件只下载一次 2，没有以上限制
- noscript 标签 在不支持脚本的页面才显示，支持脚本的浏览器永远不显示标签的内容

三、语言语法

七种数据类型

Number	String	Boolean	Symbol	Null	undefined	Object
原始	原始	原始	原始	原始	原始	引用
typeof 检查: 'number'	'string'	'boolean'	'symbol'	'object'	'undefined'	'object'

- typeof 缺点 对于 Array、Null 数据，都会检测为 Object，原理：typeof 实际上检测的是数据类型指向的**地址**，其中 000 表示对象，而 null 恰好是空指针对象，所以判为对象

| 000->对象 | 1->整数 | 010->浮点数 | 100->字符串 | 110->布尔

- 构造函数与对象 对于七种数据类型，使用构造函数创建实例和直接调用函数构造实例的区别，虽然 log 打印相同，但是一个是 Number 型，一个是 Object 型，二者本质不相同。Number、Boolean、String 皆是，但是 Symbol 没有 new 构造函数

```
// Number
let num = Number(1) // num = 1
typeof num // 'number'
let num_1 = new Number(1) // num = 1
typeof num // 'object'
```

Number 数据类型

支持十进制、十六进制（0x 开头）、八进制（0 开头，后面数字不大于 7）、浮点值。拥有最大值（Number.MAX_VALUE）和最小值（Number.MIN_VALUE），超过则为+-infinity，对于本该是数字但不是数字的表示为 NaN，例如：分母为+-0，式子包含 NaN 等，但是每个 NaN 都互不相等

```
NaN == NaN // false
```

- 非数值转为数值的方法
  - Number(param)
  - parseInt(param,scale) 常用。参数二表示进制，可以选择二、八、十六进制。若不定义，则按照字符串命名显示，即长得像什么（x0、07），就当做什么。自动忽略空字符串，从第一个非空开始检测，若其为非数字，返回 NAN（纯空字符串也为 NaN），若为数字，截取到非数值字符串之前，并作为结果返回，自然'.'也当做非数值字符串，遇到也返回。

    - parseFloat(param)
- Null 和 undefined Null 表示空指针对象，undefined 则是声明但未定义，但是 null == undefined

String 数据类型

使用单、双、反引号（模板字面量，可换行，可使用${}插值）包裹的都是字符串，可以解析类似于'\n'的转义字符，若不想解析，使用 String.row(string)

- 非字符串转为字符串
    - xxx.toString() 但是 null 和 undefined 没有此方法，对于数值，toString 还有参数，Number.toString(log)，表示将数值先转为几进制，再转为字符串
    - String(param) null 和 undefined 时，使用此函数，返回'null'和'undefined'

Symbol

符号类型。使用 Symbol(param) 创建，每一次创建都是唯一的，主要用来确保**对象属性**唯一性，即虽然长得像，但不是一个东西，不会覆盖，因为参数只起到一个描述的功能，并不做区别标识符，本质都是唯一的。

```
let symbol = Symbol()
typeof symbol // Symbol
console.log(symbol) // Symbol
let symbol_s = Symbol('symbol_ 1') // 参数非必须
let symbol_1 = Symbol('symbol_ 1') // 传一样的参数
symbol_1 == symbol_s // false，宛如长相相同，指向地址不相同的 Object
```

- Symbol.for(param) 全局注册 即没有就全局注册，有就直接全局拿过来，改善了长相相同却永不相同的缺点，但二者必须接皆使用 for，否则不是全局注册。param 必须传一个字符串给 for 方法，没传就当做传入 'undefined'，传入非字符串则报错。对于 for 全局注册的符号，可以使用 keyFor 查询符号的字符串，若查询的不是全局注册的符号，返回 undefined，若传入非符号，报错。

    ```
    let symbol = new Symbol.for('symbol') // 此时没有，全局注册
    let symbol_1 = new Symbol.for('symbol') // 此时全局有，直接拿过来，也就是上面的
    symbol == symbol_1 // true

    let symbol_f = new Symbol('symbol') // 与全局不同，只是新创建一个符号实例
    symbol_f == symbol_1 // false

    Symbol.keyFor(symbol) // 'symbol'
    Symbol.keyFor(symbol) // 'undefined' 不是全局注册
    ```

- 作为对象属性 出现一个对象，两个键值长得一毛一样的，但是别担心冲突，访问也只能使用那个 symbol 实例作为键。获取属性集也是只能通过 Object.getOwnPropertySymbols()。Object.getOwnPropertyNames 获取的属性值不包括符号属性，for-in 也不遍历符号属性，但是 Object.getOwnPropertyDescriptors() 和 Reflect.ownKeys()，是返回普通属性和符号属性的。

    ```
    ```

```
symbol = Symbol('xxx')
symbol_1 = Symbol('xxx')
obj[symbol] = 1 // 直接赋值
Object.defineProperty(obj,symbol_1,{value:1}) // 使用 defineProperty 定义
obj === {
  Symbol(xxx): 1
  Symbol(xxx): 1
}
obj[symbol] // 唯一访问标识
obj[Symbol('xxx')] // 报错
Object.getOwnPropertyNames(obj) // [] 获取所有属性值，但是不包括 Symbol 属性
Object.getOwnPropertySymbols(obj) // [Symbol(xxx), Symbol(xxx)]
```

- Symbol 属性 内置在某些对象（string、Array）中，且在这些对象的某些方法（instanceof、match、concat）被调用时，才被使用
    - description 读取传入的**描述参数**，若无则返回一个 undefined

```
Symbol('desc').toString();   // "Symbol(desc)"
Symbol('desc').description;  // "desc"
Symbol('').description;      // ""
Symbol().description;        // undefined
```

    - hasInstance xxx instanceof XXX 实际调用的是 XXX 内部的 Symbol.hasInstance(xxx)，判断是否是某**构造器对象**（new）的实例，手动自定义就是修改 instanceof 的结果。直接修改函数（B[Symbol.hasInstance] == function(){}）不起作用，自定义类时修改（class A{static Symbol.hasInstance{}}}）起作用。

```
a instanceof B === B[Symbol.hasInstance](a)
class Array1 {
static [Symbol.hasInstance](item) {
  return Array.isArray(item);
  }
}
console.log([] instanceof Array1); // [] 虽不是由 Array1 构造的，但仍为 true
```

    - match、matchAll、repelace、serach、split 内置在**字符串**实例中，作为属性，与字符串对象方法所对照，在字符串调用 str.replace 时，其实会使用到 symbol.replace，可自定义，例如 symbol.match 决定传入参数形如 /xxx/ 时，是字符串还是表达式，默认为表达式

```
let test = 'test i am what'
let str = /test/
str[Symbol.match] = false // 设置 str 不为表达式，而是字符串
test.match(str) // error,因为禁止作为表达式
```

- isConcatSpreadable 内置为**数组**的属性，判断数组是否可展开，默认数组为 true，类数组为 false，影响 Array.concat 合并数组的方式，若不可展开则整个数组作为某数组项 a[xx]

```
let arr = ['name','age']
let arr_1 = ['height']
arr.concat(arr_1) // 正常展开，为[...arr,...arr_1]
arr_1[Symbol.isConcatSpreadable] = false // 设置为不可展开
arr.concat(arr_1) // 不展开，为[...arr,arr]
let fakeArray = {
  length: 1,
  0: "hello",
}
fakeArray[Symbol.isConcatSpreadable] = true // 设置为展开
arr.concat(fakeArray) // 展开，为[...arr,'hello']
```

- toPrimitive 内置为**对象**的属性，即操作对象时，将对象当做什么类型，参数 hint 表示当前对象被当做 xx 类型，例如 +- 运算操作将对象当做'number'，console 当做字符串'string'

```
let Num = {
    [Symbol.toPrimitive](hit){
        let res
        switch(hit){
            case 'number':
                res = 11
            break;
            case 'string':
                res = '111'
            break;
            default:
                res = 'show me flowers';
        }
        return res
    }
}
console.log(+Num) // number  11
console.log(`${Num}`); // string '111'
console.log(Num + ""); // default 'show me flowers'
```

三种声明方式

- var 使用 var 声明的变量，都会提升到顶部，赋值不提升。使用**声明式**声明的函数（function xxx）也会提升至顶部，若在未声明就使用，此时函数优先级更高，若在声明且赋值后使用，此时 var 优先级更高。只有函数作用域，作用域内声明的变量都为局部变量，跟随函数结束而销毁。
- let 拥有块级作用域，即存在 {} 就存在作用域。不会变量提升，所以在**未声明前**就使用会造成暂时性死区（即未声明就使用）。

```
var temp
function testDead(){
    temp = 1;
    let temp
}
console.log(test) // function 因为变量提升优先级 函数>var
function test(){}
var test


var test = 10
function test(){}
console.log(test) // 10 因为变量提升，故赋值 test = 10 在最后一句
```

-- var 和 let 的对比 for 循环定义的变量是局部还是全局，影响到渲染

```
for(let i = 0;i<5;i++){} // i 是局部变量，在 for 循环之后销毁，在 for 内部的 i 绑定的也是局部的，当时的那个 i
for(var i = 0;i<5;i++){} // i 是全局变量，不会销毁，最后以 i = 5 的形式存储，for 内部最终绑定的也是全局的 i，会变
```

- const 与 let 相同，但是声明即需初始化，之后不能修改，常量则使用 const，或只修改对象的属性，可以使用 const

for 与 continue 和 break

- for(初始表达式;条件表达式;末尾循环体){中间循环体} ---- 条件表达式->中间循环体->末尾循环体
- continue 只是跳过这一次循环
- break 是跳出这**一层**循环

for/of 和 for/in

- for/of 是**可迭代(iterator)**对象遍历元素的，for/in 是枚举对象的可枚举属性

with(obj){}

比较少接触，将作用域全部限制在参数对象中，只能操作对象已有的属性，若是对象内部没有该属性，会沿着**作用域链**寻找，可能会影响全局变量，比较 bug.with 不能调用，故只有它访问其他变量，没有其他变量访问它的，也不能重复调用，不知道算不算局部变量。

```
let a = 1;
let obj = {}
let obj_1 = {a : 1}
with(obj){
    a = 2 // 此处因为 obj 没有 a 属性，修改的是全局的 a
}
with(obj_1){
```

```
    a = 2 // 此处因为 obj 有 a 属性，修改的是 obj_1
}
// a == 2 ; obj == {}; obj_1 = {a : 2}
```

四、变量、作用域和内存

引用值和原始值

原始值有 undefined、symbol、null、string、number、boolean，引用值有对象 object，操作的是对对象的引用。对于**复制**来说，原始值直接将 a 的值赋值给 b，但是对于引用值来说，是将 a 所指向的引用地址赋给 b，故两者有联系，改其一变二者。**函数传参是按值传参**，相当于复制了参数，a 作为参数传给函数，在函数内部操作参数'a'，对外部 a 是没有影响的，但是引用值传给函数的仍是地址，故还是会影响外部 a，但是当函数内部参数不再指向该地址时，二者就没有关系了。对于**动态属性**来说，原始值没有动态属性，引用值可以随意增删查改属性值。

上下文和作用域

函数和 window 产生上下文，其中 window 为全局上下文，上下文产生作用域链，作用域链是栈操作，越里层越早出栈，也就能访问到越外层的变量，当变量在当前上下文没有找到时，沿着作用域链往外寻找，直到找到全局上下文。

内存

垃圾回收(GC) 计数回收法:产生引用时，计数加一，清除引用时，计数减一，若计数为 0，则进行垃圾回收，整个过程是不可见的，故容易有未清除引用导致浪费内存的情况

```
let obj = { name : 'lyf' } // 右边定义了一个对象 0x10000 而 obj 指向了该地址，即引用了它 计数为 1
let test = obj // test 也指向了 obj 指向的地址，即也对对象 0x10000 进行引用了 计数加 1 为 2
obj = null // 计数减 1 为 1 0x1000 仍未被回收
test = null // 计数减 1 为 0 0x1000 被回收
```

内存泄漏 对象引用一直无法达到 0,也就一直无法回收。检查内存，可以使用终端 process.memoryUsage(),或者控制台 Memory,打印快照前要 GC

五、基本引用类型

Date 对象

创建日期对象，返回标准日期格式(Mon Jan 31 2022 00:00:00 GMT+0800 (中国标准时间)),当传一个参数时，可以传字符串和数字，数字代表时间戳，即 1970 年 1 月 1 日午夜至某日期所经过的毫秒数，字符串有多种格式，较常使用的是 'yyyy/MM/dd 00:00:00' ，另一种较为相似的 'yyyy-MM-dd 00:00:00' 在火狐和 IE 中不兼容。Date 函数中有许多方法，常用的是将 Date 转为形如 'xxxx-xx-xx xx:xx:xx' 的格式，其中获取日期月份、周几的方法也从 0 开始的，需 + 1。

```
// new Date 的参数
let d = new Date(); // 后台默认调用 Date.parse()
let d = new Date(milliseconds); // 数字，毫秒数
let d = new Date(dateString); // '月/日/年'、'年/月/日 时间'、'年-月-日 时间'、'标准日期格式'、'月(英) 日，年'
let d = new Date(year, month, day, hours, minutes, seconds, milliseconds);

// Date 实例的方法
time = d.getTime() // 获取总毫秒数
year = d.getFullYear() // 获取年份
month = d.getMonth() // 获取月份，0-11  +1
date = d.getDate() // 获取日，1-31
day = d.getDay() // 获取星期几，0~6      +1
hour = d.getHours() // 获取小时，0~23
minutes = d.getMinutes() // 获取分钟，0~59
seconds = d.getSeconds() // 获取秒，0~59

// 标准日期格式转换为 xxxx-xx-xx xx:xx:xx
let d = new Date()
year = d.getFullYear()
let obj = {
    month : d.getMonth()+1,
    date : d.getDate(),
    hour : d.getHours(),
    minutes : d.getMinutes(),
    seconds : d.getSeconds()
}
for(let key in obj) {
    obj[key] = ('00'+obj[key]).slice((obj[key]+'').length)
    }
let result = year + '-' + obj.month + '-' + obj.date + ' ' + obj.hour + ':' + obj.minutes + ':'+ obj.seconds
```

RegExp 函数

RegExp(regular expression)，正则表达式对象，通常模式为 /xxx/xx ，其中 xxx 表示需要匹配的模式，xx 表示属性，例如是否全局，是否区分大小写，正则 A 常常用来检查一个字符串 B 是否符合预期，即 B 至少要满足 A。

```
let pattern = new RegExp('.at','g')// 全局匹配所有以 at 结尾的字符串
let pattern = /.at/g
pattern.lastIndex // 实例有一个 lastIndex 属性，表示上次匹配的结尾位置，只在全局匹配 +y 下起作用
```

- 匹配属性 g、i、y、m 分别表示全局匹配、不区分大小写、从 lastIndex 开始匹配、匹配多行
- 实例属性 用于检查匹配属性，例如 pattern.global 返回一个 Boolean 值，检查正则表达式是否设置全局匹配。此外还有 ingoreCase（忽略大小写）、lastIndex（上一个匹配的结尾下标）等
- 实例方法 检查参数字符串是否满足 RegExp 匹配条件。

- pattern.exec(str) 返回一个 Array 对象或 null（无匹配时），Array 对象有额外属性：input（参数字符串）、index（匹配初始下标） 和 groups（不知道啥用，undefined），Array 的数组项为**捕获组**，一个括号表示一个捕获组，例如 /a(b)?/，参数包含 ab 或 a 即为可匹配，若能够匹配 ab, exec 为 ['ab','b'],若只匹配到 a,exec 为['a',undefined].若是全局匹配（//g），则下一次调用 exec 方法，从 lastIndex 开始匹配；
- pattern.test(str) 返回 true、false，表示是否可匹配。

```
let pattern = /.at/g // 匹配'es'
let test = 'cat.gat'
// 第一次匹配
let match = pattern.exec(test) // ['cat', index: 0, input: 'cat.gat', groups:
undefined]
match.input = 'cat.gat'
match.index = 0
match[0] = 'cat'
pattern.lastIndex = 3

// 第二次匹配
let match_1 = pattern.exec(test)// 从 lastIndex 往后匹配
match_1.input = 'cat.gat'
match.index = 4
match[0]= 'gat'
pattern.lastIndex = 7
```

原始值的类型

即 Boolean、Number、String 的构造函数，原始值在使用到其构造函数的属性方法时，其实是手动生成一个实例，完成操作，并在下一行之前销毁。有一些是实例方法，有一些是函数方法。

```
let s = 'hello'
s.hello = 'xixi' // 其实等于 new String(s).hello = 'xixi'
// 先销毁上面的 new String
console.log(s.hello) // 为空，因为被销毁了
```

- Boolean 函数 形如 new Boolean(true)，传入一个 true 或 false，不传默认为 false，改写 toString 和 valueOf，返回 'true' 和 true；但是 Boolean 对象和 Boolean 值不一样，因为**对象的布尔值**默认为 true，故 new Boolean(false) == true，但是我本意是想设置一个 false，只能使用 new Boolean(false).valueOf，所以会造成歧义，不建议使用。 此外，使用构造函数实例化原始类型，还造成 typeof 和 instanceof 判断失效，即判断为 object 而不是 number，故都不建议使用。

  ```
  let b1 = new Boolean(true)
  let b2 = true
  typeof b1 // 'object'
  typeof b2 // 'boolean'
  b1 instanceof Boolean // true
  b2 instanceof Boolean // false
  ```

- Number 函数 下列方法都会将数值转为字符串，注意直接使用数值调用不起作用。例如 10.toFixed(2) // 报错。 Number.isInteger() 传入一个数值，判断是否是整数，小数位为 0 也认为是整数 num.toString(num) 参数 num 表示将数值先转为**n 进制**后才转为字符串 num.toFixed(num) 参数 num 表示数值保留几位小数后转为字符串，没有小数也要制造小数 num.toExponential(num) 参数 num 表示**小数位数**,使用科学计数法将数值转为字符串 num.toPrecision(num)，参数 num 表示**总保留位数**,表示将数值转为科学计数法并转为字符串

```
let num = 1233
num.toFixed(2) // 1233.00
num.toExponential(2) // 1.23e+3
num.toPrecision(2) // 1.2e+3
Number.isInteger(1.0) // true
```

- String 函数 同样改写了 valueOf、toString、和 toLocaleString 方法。还有一个 length 属性，表示字符串长度。 String.fromCharCode(unicode)，表示将 Unicode 转为字符串，可以传多个 Unicode，该方法会将其拼接并返回。 str.charAt(num)，表示字符串指定索引的字符，传入一个数值，表示索引，从 0 开始计算。 str.charCodeAt(num)，表示字符串指定索引的字符的 Unicode 值，返回值为一个十进制的数值，可将其转为 16 进制的，就可以对照 Unicode 表。

```
let str = 'abcde'
str.chatAt(2) // 'c'
str.chatCodeAt(2) // 99 == 0x63
String.fromCharCode(0x61,0x62,0x63) // 'abc'
```

  - 截取字符串的方法 str.concat 拼接字符串，可以传 1~n 个参数，参数为字符串，按参数顺序拼接在调用此方法的字符串后。截取字符串的方法有 slice、substr、subString，主要使用 slice，皆可以传 1~2 个数值参数。 区分 substr 和 substring：短（方法名）距（参数 2）。substr 第二位表示截取长度，且不为负数，因为长度不能小于 0； substring 功能和 slice 类似，但有两个 bug，当参数前者大于后者时交换位置、当参数为负数时索引值为 0；

属性	slice	substr	substirng
参数 1	截取开始，表示字符串的索引，从 0 开始		
参数 1_负数	参数 1 + 字符串长度	参数 1 + 字符串长度	**索引下标为 0**
参数 2	截取结束的字符串索引	**截取长度**	截取结束的字符串索引
参数 2_负数	参数 2 + 字符串长度	长度不能为负值，故截取为空''	**索引下标为 0，必然小于等于参数 1，故交换位置**
参数 2 < 参数 1	空字符串	没影响，因为参数 2 表示截取长度	交换位置

```
let str = 'hello'
str.slice(1,2) // 'e'
str.substr(1,2) // 'el'
str.substring(1,2) // 'e'
str.slice(-1) // 'o'
str.substr(-1) // 'o'
str.substring(-1) // 'hello'
str.slice(-1,-2) // 等价于 str.slice(4,3) == ''
str.substr(-1,-2) // 等价于 str.subsr(4,0) == ''
str.substring(-1,-2) // 等价于 str.substring(0,0) == ''
```

- str.indexOf(str,index) 和 str.lastIndexOf(str,index) 返回字符串匹配的索引，indexOf 返回首次匹配的索引，lastIndx 返回最后一次匹配的索引。参数 1 表示需匹配的字符串，从 0 开始。参数 2 表示开始搜索位置

- 判断是否包含字符串 常使用 str.indexOf 判断是否包含某字符串，但其主要功能是返回匹配索引值，该使用的是 str.includes。 现有 str.startsWith、str.endsWith、str.includes 判断是否包含，但前两种有缺点，str.startsWith 必须从索引 0 开始匹配(即匹配头部)，str.endsWith 必须从索引 str.length - sub.length 开始匹配(即匹配尾巴)，而 str.includes 直接检查整个字符串，str.includes 和 str.startsWith 可以传第二个参数，表示开始匹配的索引，str.endsWith 的第二个参数代替 str.length。 其中，includes 对比 indexOf，选择 includes。

  ```
  let test = 'name'
  test.startsWith('na') // true
  test.startsWith('a') // false
  test.endsWith('e') // 4-1 = 3 == e true
  test.endsWith('m') // false
  ```

- str.trim()、str.repeat(num) trim 删除字符串前后所有空格，repeat 的参数 num 表示重复次数，将重复的字符串拼接，并返回。

- str.padStart(length,concatStr) 和 str.padEnd(length,concatStr) 即将参数 2 拼接在字符串前/后，使其扩展至参数 1 长度，参数 1 表示最终字符串长度，若小于原本长度，则返回原字符串，若大于原本长度，padStart/padEnd 决定在字符串前后填充，参数 2 默认为空格，可以传一个字符串，循环填充。

- 改变字符串大小写 toLowerCase、toLocaleLowerCase、toUpperCase、toLocaleUpperCase。前两个将字符串小写，后两个将字符串大写。加 Locale 表示地区

- 字符串匹配正则表达式 除了 RegExp.exec(str) 和 RegExp.test(str)，字符串本身也有匹配正则的方法 str.match(str),参数表示匹配，功能与 RegExp.exec 类似，但是没有 lastIndex 的概念(故没有下次匹配从 lastIndex 开始),若直接匹配，返回一个包含 index、input 属性的数组，包含所有匹配项;若给正则表达式添加全局(g),则没有属性 index 和 input str.search(str),参数表示匹配，返回匹配到的索引值，不匹配返回 -1 str.replace(str,str) 匹配替换字符串，参数 1 表示匹配项，参数 2 表示将匹配项替换成该项;若有多处匹配，只修改第一次匹配到的项，但是若参数 1 是正则表达式，且有全局标识 g，就会修改整个字符串。str.split(str,length)，匹配字符串切割数组，参数 1 是需匹配项，参数 2 为数组长度，若切割超过，也只保留这么多。

```
let str = 'test one test two'
let p1 = /te/g
let p2 = /te/
str.padStart(20,'1') // '111test one test two'
str.padEnd(5,'.') // 'test one test two'

str.match(p1) // ['te'] 当添加全局时，没有额外属性
str.match(p2) // ['te', index: 0, input: 'test', groups: undefined]
str.search(p1) // 0
str.replace('t','l') // 'lest one test two'
str.replace(/t/g,'h') // 'lesl one lesl two' 修改整个字符串
str.split(' ',2) // ['test','one'] 保留前两个
```

- 比较两个字符串大小 str.localeCompare(str),根据字符串在字符表的排序，决定大小，若比参数字符串大，返回-1,比参数字符串小，返回 1,一样大返回 0

```
let str = 'apple'
str.localeCompare('appla') // 1
str.localeCompare('bbb') // -1
str.localeCompare('apple') // 0
```

其他内置对象

最熟悉的就是 Math 和 Global 对象，何时何地都能使用的内置对象。Global 表示全局作用域对象，有一个函数是 eval，接收一个字符串，字符串相当于要执行的表达式。 常见使用是比较数组的哪一项最大

- Math.min(num...)/max(num....)
- Math.floor(num)
- Math.round(num)
- Math.random() // 若要找到范围 [n,m) ,使用 n + Math.rondom()*(m-n)
- Math.ceil(num) // 向上取整
- Math.abs(num) // 绝对值

```
eval("console.log('hi')") // 'hi'
eval('Math.max(' + arr.toString() + ')') // 先拼接字符串，然后整体字符串 变成
'Math.max(...arr)'
```

六、集合引用类型

Object

创建对象有两种方式，一个是实例化，一是对象字面量。注意，修改对象时，若直接是 obj = xxx，其实不是修改，而是替换，因为指向的地址已经修改，需使用 obj.xxx

```
let obj = new Object()
let obj = {
    name:'lyf'
}
```

Array

创建也有两种方式，实例化和数组字面量，使用数组字面量创建数组不会调用 Array 构造函数。 new Array 实例化可以传参数，传不同的参数，实例化的数组也不同。可以使用 Array.of(item...) 方法（ES6），of 创建数组，且不论参数类型，都当做数组项。 Array.from(likeArray,callback,this),将类数组转为数组，参数 1 表示类数组;参数 2 表示**将参数 1 转为数组后**进行操作的回调函数，类似 Array.map()；参数 3 为参数 2 中的 this 指向，当参数 2 不是箭头函数时起作用。

- 类数组 拥有 length 属性，且 length 属性为数值且有限，且属性为索引值（'0',0）,常见的类数组有字符串

```
// 判断是否是类数组
function isLikeArray(o) {
    if (typeof o === 'object' && isFinite(o.length) &&  o.length >= 0 && o.length
< 4294967296){
        return true
    } else {
        return false
    }
}
```

```
let arr = new Array(3) // [ , , ]
let arr = new Array('3') // ['3']
let arr = new Array('3','2') // ['3','2']
Array.from('test') // ['t','e','s','t']
Array.of(3) // [3]
let likeArray = {
    0:1,
    1:3,
    length:2
}
Array.from(likeArray,function(item){return item*this.attribute},{attribute:2}) //
{length:2,0:1,1:3} -(from)> [1,3] -(function)> [2,6]
Array.from(likeArray,item => item*this.attribute,{attribute:2}) // 参数 2 是箭头函
数，则 this 指向 window {length:2,0:1,1:3} -(from)> [1,3] -(function)> [NaN,NaN]
```

数组的迭代器

即遍历整个数组，返回数组的属性迭代器，有 arr.keys、arr.values、arr.entries，因为返回是迭代器对象，所以需要 Array.from 将迭代器显示。

```
let arr = [1,2,3]
Array.from(arr.keys()) // [0,1,2]
Array.from(arr.values()) // [1,2,3]
Array.from(arr.entries()) // [[0,1],[1,2],[2,3]]
```

复制和填充

arr.copyWith(startIndex,startCutIndex,[endCutIndex]) ES6 新增，类似于基因重组，将数组的某段变成数组的另一段，会改变原数组，但是数组大小不会变。 参数 1 表示开始被覆盖的索引;参数 2 表示剪下的数组起始位置，不填则默认为索引 0;参数 3 表示结束剪下的数组位置，只改变[参数 1~参数 1+(参数 3-参数 2)] arr.fill(str,[startIndex],[endIndex]) 指定填充内容，参数 1 表示填充内容，参数 2 表示开始填充位置，若不填则表示从索引值 0 开始填充所有，参数 3 表示结束填充位置，不包含。

```
[0,1,2,3,4,5].copyWith(0,3) // [3,4,5,3,4,5]
[0,1,2,3,4,5].copyWith(0,3,4) // [3,1,2,3,4,5]
Array.prototype.copyWith.call({ length : 5, 1 : 1, 2 : 2, 3 : 3 } , 0 , 3)
// likeArr -(call)> [ , 1 , 2 , 3 ,  ] -(copyWith)> [3, , 2 , 3 , ] -(likeArr)>
{length : 5 , 0 : 3, 2 : 2, 3 : 3}
[0,1,2,3,4,5].fill(0) // [0,0,0,0,0,0]
[0,1,2,3,4,5].fill(0,2) // [0,1,0,0,0,0]
[0,1,2,3,4,5].fill(0,2,3) // [0,1,0,3,4,5]
```

严格相等

即比较数组和方法参数时，使用 === 表示严格相等。包括 lastIndexOf、indexOf 和 includes（ES7）。参数一表示需要查找的元素，参数二表示开始查找的索引。但是 lastIndexOf 的参数二表示从该索引后往前搜索。若有找到就返回索引，没有就返回 -1。

迭代方法

every、filter、forEach、map、some。

归并方法

即每次操作的都是前面项的结果，例如数组项相加，最终返回一个结果。 arr.reduce(function,) 参数 1 为归并回调函数，function(prveRes,now,index,arr) 回调函数参数 1 表示上一次归并的结果，参数 2 表示本次的数组 item，参数 3 表示 item 索引，参数 4 表示本数组。参数 2 表示第一次迭代时的 prev，若不填则默认 arr[0] 为 prev,归并函数从第二项开始。 arr.reduceRight() 功能与 reduce 相同，但从数组最后一位遍历至第一位。

```
let arr = [1,2,3]
arr.reduce((prev,now,index,arr) => { return prev + now}) // 1 + 2 + 3 = 6
arr.reduceRight((prev,now,index,arr) => {return prev - now}) // 3 - 2 - 1 = 0
```

所有归类

类型	方法
检测	isArray、instanceof
迭代器方法	entries、keys、values
复制和填充	copyWithin、fill
转换方法	valueOf、toString、toLocaleString
栈方法	push、pop
队列方法	push、shift
排序方法	reserve、sort
操作方法	slice、splice、concat
搜索和位置方法	indexOf、lastIndexOf、includes、find、findIndex
迭代方法	filter、map、forEach、some、every
归并方法	reduce、reduceRight

定型数组

ArrayBuffer

在内存中分配特定数量的字节空间。作为所有定型数组和视图引用的基本单位。对于文件对象，需使用ArrayBuffer.

DataView

视图对象，创建时必须要有 ArrayBuffer 作为参数。

Map

俗称字典，以**键值对**形式存储数据，实例化 new Map(arr) 可传一个数组参数，数组格式为 [[key1,value1],[key2,value2]]。 常用方法皆在实例上， map.set(key,value)、map.get(key)、map.has(key)、map.delete(key)、map.clear() 属性方法操作字典，map.size 属性表示字典的长度。 字典的键是不限类型的，即函数、对象也能作为键，修改函数和对象时，键值和键同等改变。

```
let map = new Map()
map.size // 0
map.has('name') // false
map.set('name','lyf')
map.has('name') // true
map.get('name') // 'lyf'
map.set('age',18)
map.set('height',188)
map.size // 3
map.delete('height')
map.size // 2
map.clear()
```

```
map.size // 0
let fn = function(){
    console.log('xxx')
}
map.set(fn,'fn') // 函数作为键
```

迭代器

字典也有 entries 迭代器，等价于 Symbol.iterator，即 map.entries == map[Symbol.iterator]，返回一个按插入顺序的键值数组。包括 keys 和 value

WeakMap

弱字典，拥有的功能和字典稍有不同 没有迭代器（entries、values、keys）、size 属性和 clear 方法，且只接受**对象**作为键名，即**只有存取键和 delete**功能。 null 虽然是对象，但是作为特殊对象，也不可以作为键滴，因为 WeakMap.set(key, val) 是通过 Object.definePoperty 给 key 加了一个新属性 this.name，key 必需是个 Object 才能使用 defineProperty。 弱的概念是对**对象的引用**是偏弱的，对象回收机制中，只有该对象及该对象的引用都移除(计数为 0),才会进行垃圾回收，但弱字典的引用不计入引用。 在 Map 中，被当做键名/键值的对象，即使在外部进行清除，Map 仍保持对其的引用，故不会被回收 当被当做**键值**的对象被清除后，WeakMap 仍保持对其的引用，即不会回收 当被当做**键名**的对象被清除后，WeakMap 会对键名和键值对象进行回收 但是，打印 WeakMap/Map 对象，仍可以找到被回收的键值对，但是没有迭代，键名也被清除了，故按道理也访问不了

```
let weak = new WeakMap()
let obj = { name : 'lyf'}
let res = {age : 18}
weak.set(obj , res) // 设置键
weak.get(obj) // 获取键 { age :18 }
res.age = 20
weak.get(obj) // 获取键 { age :20 }
obj.name = 'zzz'
weak.get(obj) // 获取键 { age :20 }
res = null
weak.get(obj) // {age: 20} // 还存在
obj = null // 销毁了 obj
weak // WeakMap {0: {key: {name: 'zzz'},value: {age: 20}} // 仍在内部
```

- 使用 node.js 检查是否进行垃圾回收

```
> node --expose-gc
// 手动执行一次垃圾回收，保证获取的内存使用状态准确
> global.gc();
// 查看内存占用的初始状态，heapUsed 为 4M 左右
> process.memoryUsage();
> let wm = new WeakMap();
// 新建一个变量 key，指向一个 5*1024*1024 的数组
> let key = new Array(5 * 1024 * 1024);
// 设置 WeakMap 实例的键名，也指向 key 数组
// 这时，key 数组实际被引用了两次，
```

```
// 变量 key 引用一次，WeakMap 的键名引用了第二次
// 但是，WeakMap 是弱引用，对于引擎来说，引用计数还是 1
> wm.set(key, 1);
> global.gc();
// 这时内存占用 heapUsed 增加到 45M 了
> process.memoryUsage();
// 清除变量 key 对数组的引用，
// 但没有手动清除 WeakMap 实例的键名对数组的引用
> key = null;
// 再次执行垃圾回收
> global.gc();
// 内存占用 heapUsed 变回 4M 左右，
// 可以看到 WeakMap 的键名引用没有阻止 gc 对内存的回收
> process.memoryUsage();
```

Set

集合数据，非键值对，而是存储键的，每一个键都是唯一的，即多次添加同一个键，只会保存一次，实例化 Set 函数创建，可以传一个参数，为数组，数组的格式为 [key1,key2]。使用 add 添加，delete 和 clear 删除。delete 返回一个 boolean 值，若集合有此元素，返回 true ，否则返回 false。也有迭代器 entries、keys、values。

```
let set = new Set()
set.add('111')
```

第七章、迭代器与生成器

迭代器 Iterator

自己调用自己称递归，重复执行一段代码叫迭代，不断使用前者为归并，且三者都有特定的退出执行指令。 迭代器模式即具备 Iterable 接口的可迭代对象， 有 Array、Map、Set、String、TypedArray，函数 arguments 对象和 NodeList 对象，这些对象**元素有限且具有无歧义的遍历顺序**； 可迭代对象有一个属性 arrSymbol.iterator，类型为函数，调用返回一个迭代器； 迭代器的 arrSymbol.iterator.next() 方法，返回一个迭代结果（IteratorResult）的对象 obj = {done:,value:}； -| 其中 done 为 Boolean 值，表示是否迭代完成，遍历到末尾时为 true；value 表示可迭代对象的值，当 done 为 true 时 value 为 undefined。 一般不会使用 arrSymbol.iterator.next() 操作迭代器，而是在某些方法内部调用迭代器，例如 for...of、数组解构、扩展运算符、Array.from()、创建 Set、Map，Promise.all()、Promise.race()、yield*。 对于没有迭代器的 js 结构，使用迭代器方法，会报错。

总结：拥有 Iterable 接口 -> 可迭代对象（拥有 Symbol.iterator 属性）-> 调用 Symbol.iterator 属性 -> 返回迭代器 -> 调用迭代器的 next 方法 -> 返回迭代结果对象

```
let arr = ['a','b','c'];
let iter = arr[Symbol.iterator](); // 执行迭代器属性函数，返回迭代器
```

```
iter.next() // 调用迭代器的 next 返回迭代器结果 {value:'a',done:false}
iter.next() // {value: 'b',done:false}
iter.next() // {value:'c',done:false}
iter.next() // {value:undefined,done:true} 接下来调用 next()都返回此

// 修改迭代器
class newArray{
    constructor(stop){
        this.stop = stop;
    }
    [Symbol.iterator](){ // 返回 this
        return this;
    }
    next(){ // 在调用迭代器时使用
            if(this.stop > 0){
                this.stop--
                return {done : false,value : 'xxx'}
            }
            return {done : true,value : 'xxx'}
    }
}
for (var value of new newArray(3)) { // [Symbol.iterator] -> next
  console.log(value); // 'xxx'
}
```

生成器 Generator

生成器 Generator 是函数，为了区分生成器和普通函数，生成器函数名一般带星号 function *test(){}. 生成器不可以使用 new 实例化,调用生成器函数返回**迭代器**，每次调用产生的实例相互不影响，且继承生成器的原型属性，但是没有 this 的概念。 调用生成器函数时，并不执行函数内部代码，而是返回迭代器对象，指向函数内部。 当调用生成的迭代器 next() 时，才执行函数内部代码，并返回对象 { done:true, value:undefined }，value 是生成器函数的**返回值**，默认是 undefined，可以在生成器中修改。 生成器执行返回一个迭代器，迭代器本身也有 Symbol.iterator 属性，执行后返回自身。 在需要添加迭代器的 js 结构，将生成器添加到 js 结构的 Symbol.iterator 属性上。

```
function *generatorFn(){
}
const g = generatorFn()
g.next() // {value:undefined,done:true}
g[Symbol.iterator]() === g // 本身就是一个迭代器

// 自定义 return
function *generatorFn(){
    return 'xxxx'
}
const g = generatorFn()
g.next() // {value:'xxxx', done:true}

let obj = {name:'',age:18}
obj[Symbol.iterator] = function* test() {
```

```
    let keys = Object.keys(this)
    for(let key of keys){
        yield [key,this[propKey]]
    }
}
```

yield

控制生成器开启和暂停，在遇到此关键词前，函数内部正常执行，遇到时执行停止，需再次调用迭代器 next() 激活接下代码。 yield 必须在生成器函数中使用，且不能在嵌套在非生成器函数里。 当迭代器调用 next 传参数时，会被 yield 接收，即 yield === 参数，第 n 个 yield 存储着 第 n + 1 次调用 next() 时所传的参数，因为一次调用 next() 时执行到 yield 之前，传参无法接收。 yield 后面可以跟一个表达式，表示 next() 至此的返回值，会先计算表达式再返回，同样，也只有函数执行到这一行时，才计算。若 yield 在其他表达式中，也需要计算，且其后面的表达式作为返回值不作数，不传参就是 undefined，影响表达式的值。

```
// yeild 的功能
function *generatorFn(){
    console.log('xxxx')
    yield 1+1       // 第一次 next() 至此，先计算表达式
    console.log('ssss')
    yield 'ssss'       // 第二次 next() 至此
    return 'finally'   // 执行至此，退出
}
const g = generatorFn()
g.next() // 'xxxx' {value:2, done:false}
g.next() // 'ssss' {value:'ssss',done:false}
g.next() // {value:'finally',done:true}

// yield 使用限制
function *test(){
    function test(){
        yield '' // 虽在生成器函数中，但嵌套在普通函数中，报错
    }
}

// return + yeild + 表达式
function *test(){
    return yield 'test'
}
t.next() // 执行至 yield，返回 {value:'test',done:false }
t.next('change') // 将 ‘change’ 传给 yield，变成 return 'changhe'，返回
{value:'change',done:true}

// yeild 在表达式里
function * f(x){
    let y = yield (x + 1)// yield 3 -> yield
    return y
}
let g = f(2)
g.next() // {value : 3, done: false}
```

```
g.next() // {value : undefined, done: true}, 因为 yield 为 undefined, 表达式 yield (2 + 1) == undefined
```

Generator.prototype.throw()、Generator.prototype.return()

生成器返回的迭代器对象有一个 throw 方法，调用时在生成器函数内抛出错误，需在生成器函数中进行 try...catch 捕获，捕获只能一次，因为抛出错误后就不执行 try 代码，且必须在至少执行一次 next() 后，不然不会再函数内部捕获。throw 方法被捕获以后，会附带执行 try...ctach 后下一条 yield 表达式，也就是说，会附带执行一次 next 方法，并不影响下一次遍历。但是第二次执行 throw 后，无法再执行接后代码，即抛出错误不是内部捕获的话，停止执行代码。return 则是直接终结生成器，可以有一个参数，表示 value，done 也是 true。现在 next、throw、return 都是操作关键字 yeild，next 表示将 yeild 替换成方法参数，throw 表示将 yeild 替换成 throw 语句，return 替换 yeild 为 return 语句。

```
var g = function* () {
  try {
    yield console.log('我是第 0 个');
    yield console.log('我是第一个');
    yield console.log('我是第 x 个');
  } catch (e) {
    console.log('内部捕获', e);
  }
    yield console.log('我是第二个');
    yield console.log('我是第三个');
    yield console.log('我是第四个');
    yield console.log('我是第五个');
    yield console.log('我是第六个');
};

var i = g();
i.next();// '我是第 0 个'
i.throw('第一次出错') // '内部捕获''第一次出错''我是第二个'
i.next() // '我是第三个'
try{
    i.throw('第二次出错')
    }catch(e){
        console.log('外部捕获', e);
        } // '外部捕获', '第二次出错'
i.next() // 直接返回 value-done 对象
```

yield*

在生成器内部调用其他生成器，一是使用 for...of 手动遍历其他生成器，一是使用 yield* 执行生成器函数。

```
function *f(){
    yield 'xxx'
    yield 'test'
}
function *f1(){
```

```
    yield 'hello'
    yield* f()// 调用生成器，并遍历迭代
    yield f() // 只是调用返回迭代器
}
yield* f() 类似于 for(let key of f()){ yield key}
```

对象属性

生成器可以作为对象的属性

```
let obj = {
    * Generator(){
    }
}
let obj = {
    Generator:function *(){
    }
}
```

第八章、类和对象以及面向对象编程

对象拥有属性，属性也有属性，包括**数据属性**和**访问器属性**，

对象属性

数据属性

有 Configurable（可否删除 delete、修改）、Enumerable（可否枚举）、Writable（可否修改）、Value，前三个默认为 true，value 默认为 undefined。数据属性存储在属性描述符对象中，需使用 Object.defineProperty 修改，使用 Object.getOwnPropertyDescriptor 查看。若使用 Object.defineProperty() **定义属性**，会将 configurable、enumerable、writable 的值设置为 false，若将 Configurable 设置为 false，不能再使用 Object.defineProperty 修改数据属性。可以使用 Object.defineProperties(obj,{property1:{},property2:{}}) 定义对象的多个属性的描述符对象。可以使用 Object.getOwnPropertyDescriptors(obj) 获取一个对象的所有属性的属性描述符，该方法会遍历对象的所有属性。

```
let obj = {name:'lyf'}
Object.defineProperty(obj,'name',{configurable:false}) // 修改数据属性
Object.getOwnPropertyDescriptor(obj,'name') // 获取数据属性
Object.defineProperty(obj,'show',{writable:true})// 使用 defineProperty 生成的，默认将数据属性设为 false -> {value: undefined, writable: true, enumerable: false, configurable: false}
```

访问器属性

有 Configurable（可否删除、修改、变为数据属性）、Enumerable（可否枚举）、Get（获取函数）、Set（设置函数）。属性是拥有数据属性还是访问器属性是根据 Object.defineProperty 设置的，默认是数据属性，若设

置了 set、get 函数，则变为访问器，失去数据功能即 value 和 Writable，若设置了 writable 和 value，则变成数据属性。

合并对象 Object.assign(mainObj,otherObj...)

将资源对象的属性混入目标对象，Object.assign(mainObj,otherObj...)，将其余对象中可枚举、自有属性（使用 Object.propertyIsEnumerable() 和 Object.hasOwnProperty() 检测皆返回 true）复制到目标对象。但是是浅拷贝

```
let mainObj = {}
let otherObj = {name:''}
Object.assign(mainObj,otherObj)
// 真实步骤
// 1. otherObj.propertyIsEnumerable('name') // true
// 2. otherObj.hasOwnProperty('name') // true
// 3. Object.getOwnPropertyDescriptor(other,'name').set =
Object.getOwnPropertyDescriptor(other,'name').get
```

相等判定 Object.is()

用于判断两个数是否全等，接收两个参数，但参数不限制类型。若全等返回 true，不全等返回 false，可以判断边界情况，如 +-0 和 0 ， NaN 与 NaN 的全等性。

```
Object.is(true,1) //false
Object.is(+0 ,-0) // false
Object.is(NaN ,NaN) // true
+0 === -0 // true
NaN === NaN // false
```

创建对象

工厂函数

即调用一个函数返回一个对象，根据传入的参数不同，返回不同值的对象，缺点是创建出来的对象拥有一毛一样的属性，优点是整齐。

```
function createObj(name,age){
    let obj = new Object()
    obj.name = name
    obj.age = age
}
let obj1 = createObj('lyf',18) // {name:'lyf',age:18}
let obj2 = createObj('zzz',10) // {name:'zzz',age:10}
```

构造函数

构造函数的本质也是创建拥有一毛一样的属性的系列对象，但是使用 new 关键词实例化函数，且函数内部不用使用 new Object 创建 obj，而是使用 this 关键词赋值，也不需要 return 一个对象，因为 new 关键词帮助我们做了这些。使用 new 创建的实例拥有__proto__属性,指向原型对象,原型对象有 constructor 属性指向构造其的构造函数,可以直接简写为 obj.constructor。

```
function CreateObj(name,age){
  this.name = name
  this.age = age
}
let obj1 = new CreateObj('lyf',18) // {name:'lyf',age:18}
let obj2 = new CreateObj('zzzz',10) // {name:'zzzz',age:10}
obj1.constructor == obj1.__proto__.constructor == CreateObj.prototype.constructor
==  CreateObj // construstor 属性
```

new 关键字的本质

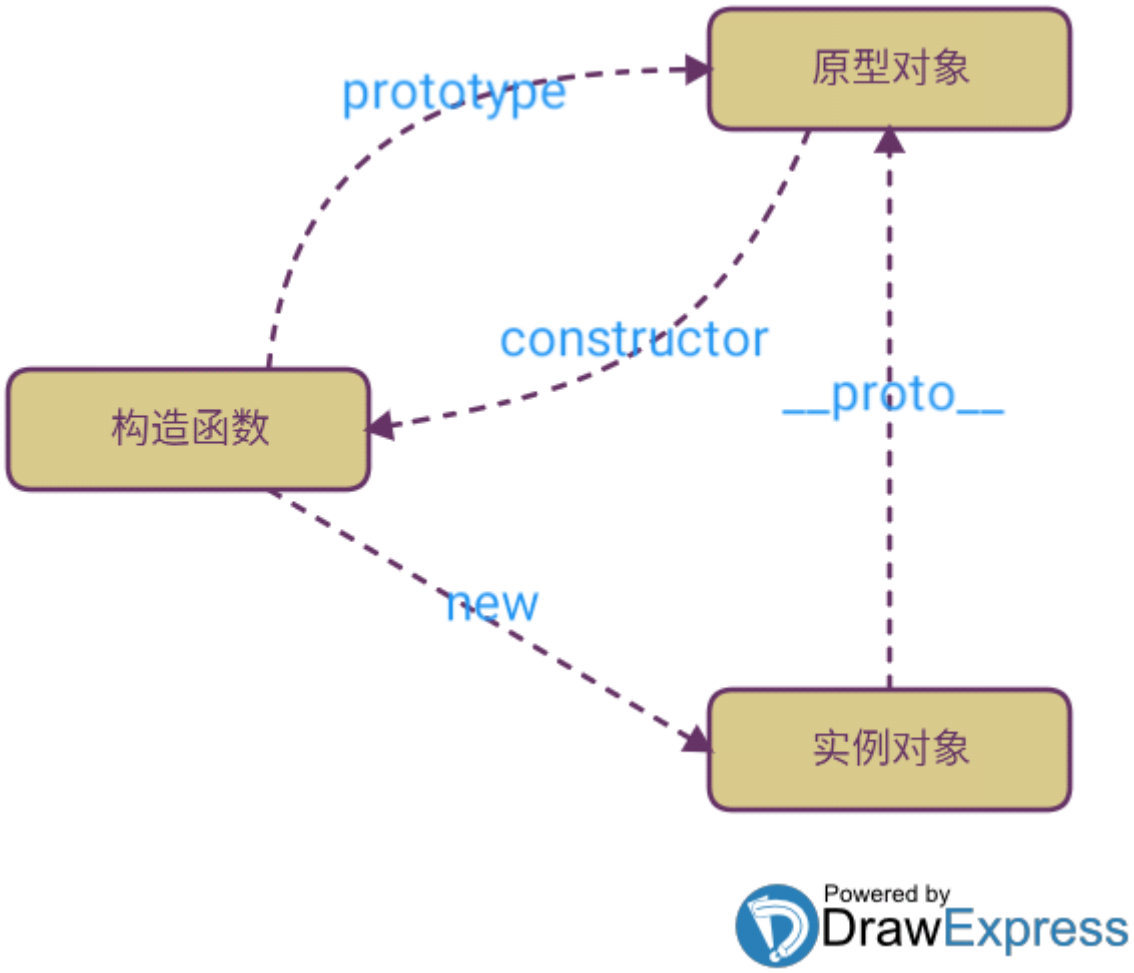
- 将构造函数作为参数传入 new
- 创建对象：let obj = new Object()
- 该对象的 prototype 指向构造函数的 prototype
- 该对象的 this 指向构造函数的 this
- 执行构造函数代码
- 返回 obj

```
function newFun(createFun){
    return function(){
        let obj = new Object()
        obj.__proto__ = createFun.prototype // 指向构造函数的原型对象，故构造函数原型
上的属性，new 实例也可以继承
        createFun.call(obj,...arguments)
        return obj
    }
}

newFun(app)('name',10)
```

原型

实例拥有 **proto** 指向父类函数的 prototype。在 625 页有详情。对于具有**遍历性**的对象方法（keys、getOwnPropertyNames、for-in 等）来说，遍历是有序的，即按照属性 数值 > 插入顺序的字符串 的顺序，但是对于一个对象来说，每次插入、删除。其内部也会根据属性进行排序。对于自定义的 prototype 对象，会修改其的 constructor 属性指向，默认指向 Object，若想指回构造函数，直接设置 constructor 属性，若想默认为不可枚举的属性，需使用 Object.defineProperty 定义 constructor 属性。若在构造实例后重新定义 prototype 对象，即改变 prototype 指向地址，此前的实例依然指向原本地址，会造成错误 -每个对象都有一个__proto__属性，并且指向它的prototype原型对象 每个构造函数都有一个prototype原型对象




```
function Person() {
}
Person.prototype = {
    name: "lyf",
    age: 29,
};
Person.prototype.constructor == Object // true
Object.defineProperty(Person.prototype,'constructor',
{enumerable:false,value:Person})
Person.prototype.constructor = Person
Person.prototype.constructor == Person // true
let person = new Person()
person.name // 'lyf'
Person.prototype = {
    sayHi(){
        console.log('xxxxx')
    }
}
person.name // 'lyf'
person.saiHi() // error, 没定义
```

调用对象	方法	作用

调用对象	方法	作用
Object	Object.create(obj)	创建一个对象，将参数作为对象所指向的原型对象
Object	Object.setPrototypeOf(obj,obj1)	传入两个对象，将参数 2 对象作为参数 1 的原型对象
Object	Object.getPrototypeOf(obj)	获取传入参数对象所指向的原型对象
Object	Object.keys(obj)	返回传入参数**自身**不包括原型对象上的可枚举属性
Object	Object.getOwnPropertyNames(obj)	返回参数自身不包括原型上的所有属性，**包括不可枚举**
Object	Object.getOwnPropertySymbols(symbol)	一样返回参数自身不包括原型上的所有属性，不过参数是符号类型 Symbol
obj	obj.hasOwnProperty(attribute)	传入一个字符串属性，返回一个 Boolean 值，判断是**自身属性**还是原型对象上的属性
prototype	xxx.prototype.isPrototypeOf(obj)	传一个实例对象，判断该实例是不是在原型链中
prototype	xxx.peototype.constructor	指回构造函数
instanceof	xxx instanceof xxx	左边是实例，右边是构造函数，判断实例的原型链中是否有该构造函数

```
// 手写一个 L instanceof R 判断
function Instanceof(L,R){
    // 当检测对象是基本类型时，返回 false
    if((typeof L != 'Object' && typeof L != 'Function')|| L === null){
        return false
    }
    // 类型为基本类型时，抛出错误
    if ((typeof R !== 'object' && typeof R !== 'function')||R === null)
        throw Error("Right-hand side of instanceof is not an object")
    while(true){
        // 找到检测对象 L 指向的原型对象，判断是否与 R 相等，相等则返回 true， 不相等则往原型链后面找，直到找到 null 仍不相等，返回 false
        if(Object.getProtopeOf(L) === null){
            return false
        }
        if(Object.getProtopeOf(L) === R.prototype){
            return true
        }
        L = Object.getPrototypeOf(L)
    }
}
```

对象迭代

即 Object.values、entries、keys，皆是传一个对象，返回一个数组。

继承

有很多语言支持接口继承和实现继承，但是 js 只能实现继承，一般通过修改原型链实现，即子类的**原型**指向父类的**实例**从而实现原型共享，或借用构造函数，即通过 apply、call 实现子类调用父类的属性、方法。就可以继承父类原型上的属性，但是父类实例属性也会继承，且若属性是引用值时，父类修改也会改变子类。原型对象指向父类实例后，子类可以覆盖父原型的函数，但是父构造函数的原型变量不受影响。若使用另一个对象覆盖继承的 prototype，则相当于不继承了，因为所指的地址修改了。JS 高级教程给出几种常见的继承方法，包括 6 种：原型链继承（new Farther）、构造函数继承（call）、组合继承（call + new Farther）、原型式继承（Fun.prototype = obj）、寄生式继承（Fun.prototype = obj + 属性）、寄生组合式继承（call + Fun.prototype = obj）等。

- 构造函数通过修改 this 指向，故使用 call、apply，即继承父类的实例属性
- 实例是通过 constructor 属性，修改子类构造函数的原型对象指向的构造函数，即修改 Son.prototype.constructor 指向

原型链继承

即子类的构造函数指向父类,原型链添加父类原型对象,且实例的 son.**proto**.constructor == son.constructor

```
function Son(){
    this.son = 'son'
}
function Farther(a){
    this.farther = 'farther'
    this.a = a
}
Farther.prototype.sayHi = function(){
    return true
}
Son.prototype.sayHi = function(){ // 覆盖所继承的 sayHi 方法，但是不改变父类的原型对象
    return false
}
Son.prototype = new Farther() // 原型链继承
// 右式为父类实例，实例有属性 constructor 指向构造其的构造函数 Farther，也有__proto__属性,指向父类原型对象，
// Son.prototype.constructor == Farther ，故 son.__proto__.constructor == Farther
// Son.prototype.__proto__ == Farther.prototype ，故 son.__proto__.__proto__ == Farther.prototype
// 故 Son 构造 son 实例，Farther 构造 son实例的原型对象,将Farther放置在原型链里，可以访问到Farther乃至其原型链上的内容

let son = new Son()
son.sayHi() // false
son.farther // 'farther'，这是 farther 实例属性，但是继承了
let father = new Farther()
father.sayHi() // true
```

盗用构造函数继承

在子类构造函数中调用父类构造函数，使用 call 和 apply 以实例为上下文，则每次实例化子类实例时，将父类的实例属性和原型属性初始化到自身去了，但是不会继承父类的原型对象。

```
function Farther(){}
function Son(){
    Farther.call(this) // 每构造一个实例，都会执行一次父类，且将属性放入本实例中
}
```

组合继承

原型链和盗用构造函数结合，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。缺点是会调用两次父类函数

```
function Farther(name){
    this.name = name
}
function Son(){
    Farther.call(this,'lyf') // 继承父类的实例属性，可以传参数实例化
}
Son.prototype = new Farther() // 原型链继承
Son.prototype.constructor = Son
```

原型式继承

不需要单独创建构造函数，在函数内部临时建一个构造函数，构造函数的原型对象指向对象，返回构造函数实例，目的是**将传入参数 obj 作为原型对象。**效果等同于 Object.assign()，但是为浅拷贝，即传入同样的 obj 参数时，引用值数据互通。

```
function fun(obj){
    function Fun(){}
    Fun.prototype = obj
    return new Fun // 返回对象 returnObj.__proto__ = obj
}
let obj = {
    name:['kobe']
}
let son1 = fun(obj)
let son2 = fun(obj)
son1.name.push('james')
son2.name // ['kobe','james']
```

寄生式继承

结合原型式继承，在原型式继承的基础下，让这个对象拥有更多的功能。

```
// 原型式继承
function fun(obj){
    function Fun(){}
    Fun.prototype = obj
    return new Fun
}
function Get(obj){
    var clone = fun(obj) // 获取原型式继承后
    clone.sayHi= function(){ // 使用工厂模式进行增强
    }
    return clone // 返回
}
```

寄生组合式继承

最常用。寄生和组合结合，函数的参数是子构造函数和父构造函数，不通过调用父类构造函数（new Farther()）的方式将子类构造函数的原型指向父类构造函数，而是使用父类原型对象调用原型式。但是对于父类原型也是浅拷贝，即多个子实例可以修改共享父类原型的引用值属性。

```
function fun(obj){
    function Fun(){}
    Fun.prototype = obj
    return new Fun
}
function inherit(son,farther){
    let prototype = fun(farther.prototype) // 而是调用原型继承函数，创建父类原型的副本
    prototype.constructor = son // 解决由于重写原型导致默认 constructor 丢失的问题
    son.prototype = prototype
}

function son(){
    Farther.call(this) // 继承父类实例属性
}
Farther.prototype.name = ['kebo','james']
let ccc = new son()
ccc.name.push('xxxx') // ['kebo','james','xxxx']
let ddd = new son()
ddd.name // ['kebo','james','xxxx']
```

类

ES6 的新属性，定义时有 let Class = class{} 和 class Class{}，不具备变量提升，函数使用声明式时可以变量提升。若不定义构造函数，构造函数默认为空 constructor(){}，使用 new 构造时，调用 constructor 对象，默认返回 this 对象，可以修改 return 值。其实类就是一种特殊函数，使用 typeof 检测类，返回 'function'。类也有 prototype 属性指向原型，而原型也有一个 constructor 属性指回类。

- 类构造函数和普通构造函数的区别 使用普通构造函数时，若不使用 new 关键字，则 this 作为全局变量 window 的属性。但是不使用 new 关键字构造类时，会报错。
- 构造函数 虽然 new 构造类实例时调用 constructor 方法，但是其并不是构造函数，而是**类本身是构造函数**，对类的构造函数进行 instanceof 操作时，返回 false

```
class Person{}
Person.prototype.constructor == Person // true
let p = new Person()
p instanceof Person // true
p instanceof Person.constructor // false
```

类的成员

类中有三类成员：实例成员、原型成员、类本身的成员。一个同名属性可在类中定义 3 个不同功能的成员。成员可以是生成器和迭代器。

- 实例成员 放置在构造函数 constructor 中，因为默认返回 this，故实例成员使用 this 关键字定义，**不会共享**，而是每次实例化时重新构造，必须使用赋值式定义，即使用 = 号赋值。

  ```
  class Person{ constructor(){
      this.names = ['lyf']
  }}
  let p1 = new Person()
  p1.name // ['lyf']
  let p2 = new Person()
  p2.name // ['lyf']
  p1.name == p2.name // false
  ```

- 原型成员 在类中定义的属性或方法为原型成员，被每个实例共享，实例遵循先在实例成员中搜索，若没有则在原型中搜索。

  ```
  class Person{
      sayName(){
          console.log('i am groot')
      }
      name = 'lyf' // 不能使用 : , 此符号表示对象
  }
  let p1 = new Person()
  let p2 = new Person()
  p1.sayName == p2.sayName // true
  ```

- 类自身的成员 类的实例没有该成员，并不能访问，此成员仅在类中存在，使用 static 为前缀，使用类名 + 成员名调用。

  ```
  ```

```
class Person{
  constructor(){
      this.sayName = function(){
          console.log('i am constructor')
      }
  }
  sayName(){
      console.log('i am prototype')
  }
  static sayName(){
      console.log('i am static')
  }
}
p.sayName() // 实例成员 'i am constructor'
Person.prototype.sayName() // 原型成员，实例也可以访问，当实例成员找不到时 'i am prototype'
Person.sayName() // 只有类本身可以调用 'i am static'
```

- 添加成员 可以直接在原型和类上添加成员

```
Person.name = 'lyf' // 等价于 static name = 'lyf'
Person.prototype.name = 'test' // 等价于在类内部定义
```

类的继承

类的出现很大目的是修改继承机制，本质依然是原型链继承。extends 关键字继承类，可以继承类的实例成员（constructor）、原型成员和自身成员（static），此时使用 instanceof 检测父类和子类，都返回 true，证明父类在原型链中。

```
class Farther{
    constructor(){
        this.name = 'nono'
    }
    sayName(){
        console.log('show me flowers')
    }
    static sayName(){
        console.log('show me flowers too')
    }
}
class Son extends Farther{
    constructor(){
        super() // 调用父类的构造函数
        this.age = 22
    }
}
let son = new Son()
Son.sayName() // `show me flowers too`
```

```
son instanceof Son // true
son instanceof Farther // true
```

- super [Object | Function] super 关键字只可以在 extends 中使用，且只能在构造函数中使用函数调用 super() 或在类方法中使用对象属性调用 super.sayName()，否则报错。 当子类有 constructor 函数时，必须先调用父类的构造函数，使用 super() 函数，可以传**参数**作为父类构造函数的参数，其将子类的 this 传入父构造函数调用后再返回。此时 this 上有父类的实例属性，然后再操作子类实例属性，以免冲突。当想覆盖父类的原型方法和自身属性时，直接在子类上定义一个同名同类型的方法，在函数内部通过 super 调用该方法进行覆盖。

```
class Farther{
    constructor(){
        this.name = 'nono'
    }
}
class Son extends Farther{
    constructor(){
        super() // 此时，this == {name:'nono'}
        console.log(this instanceof Farther); // true
        this.age = 22 // this == {name:'nono',age:22}
    }
}
// 静态方法
class Farther{
    static sayName(){}
    static sayHi(){}
}
class Son extends Farther{
    static sayName(){
        super.sayName()
        console.log('i am son')
    }
}
let son = new Son()
Son.sayName() // 此时调用的就是子类的 sayName，但是子类中引用了父类，故也有继承
Son.sayHi() // 未经修改，直接调用父类的 sayHi 方法
```

- 抽象基类 即父类只用来被继承，不被实例化，需使用 new.target 属性检测，new.target 表示当前作用在哪个构造函数，若不是使用 new 构造的实例，为 undefined。

```
class Person{
    constructor(){
        if(new.target == Person){
            throw new Error('i am warng')
        }
    }
}
let per = new Person() // 抛出错误
```

- 内置类型 调用某些内置实例的方法会返回新实例，即子类继承父类，调用父类上的方法，仍返回一个子类实例。可以通过 Symbol.species，是一个函数，在构造实例时返回实例的类。

```
class newArray extends Array{
}
let nArray = new newArray(1,2,3,4)
let n2 = n.filter(x => !!(x%2)) // 调用实例的方法 filter
n2 instanceof newArray // 构造了一个新的 newArray 实例，仍为 true

class newArray extends Array{
    static get [Symbol.species](){
        return Array
    }
}
let n2 = n.filter(x => !!(x%2))
n2 instanceof newArray // false
```

- 类混入 若要灵活实现继承，可以使用函数，将要继承的类作为参数传入。

```
let newClass = (newclass) =>{
    class extends newclass {}
}
```

九、代理和反射

类似于秘书，若想操作目标对象，必须通过一层代理，将操作放置在代理对象，由代理对象传达给目标函数，创建代理的关键字是 Proxy。new Proxy(target,handler) 参数 1 表示需要被代理的目标对象，对象类型不限制。参数 2 表示一个属性为函数的对象，函数表示执行操作时代理的行为，通常有 get、set 等。Proxy.prototype 是 undefined ,不能使用 instanceof 检测是否是代理实例

```
let obj = {} // 目标对象
let p = new Proxy(obj,{
    get:function(obj,prop){// 获取对象属性时调用
        return obj.hasOwnProperty(prop) ? obj[prop] : 22
    }})
    p.a // 22
    obj.a // undefined
    obj.a = 10
    obj == p // false
    p.a // 10 此时 obj 中有此属性
    p instanceof Proxy // throw error
```

代理捕获器

代理捕获器以函数形式存储在构造代理实例的参数 2 中，在处理操作时触发， get() 捕获器在实例 proxy[xxx] 和 object.createproxy 中触发，可以传入三个参数 get(target,property,proxy),分别是目标对象，获取的对象属性名，代理对象。在捕获器中使用 Reflect 反射，每一个捕获器都有同名称的反射 API,例如 Reflect.get(target,property,proxy).

```
let obj = {}
let p = new Proxy(obj,{
    get(obj,prop,proxy){
        console.log(obj == obj)
        console.log(p == proxy)
    }
})
p.name // true true
```

捕获器错误

当为目标对象设置了一个属性描述对象为不可操作时，此时使用捕获器修改 get 操作时，会抛出错误，类似这种不允许捕获器越界的行为称为 '捕获器不变式'

```
let obj = {} // 目标对象
Object.defineProperty(obj,'name',{
    configurable:false,
    writable:false,
    value:'zzz'
})
let p = new Proxy(obj,{get(){
    return 'lyf' // 获取参数时，将其赋值为'lyf'
}})
p.name // throw error
```

撤销代理

即中断代理和目标函数的联系，使用 Proxy.revocable() 创建代理，会返回一个对象，有属性 proxy 和 revoke,表示代理对象和撤销代理函数，撤销是不可逆的操作，不论撤销几次，都是一样会抛出错误。

```
let obj = {name:'lyf'}
let { proxy,revoke } = new Proxy.revocable(obj,{
    get(){
        return 'xxx'
    }
})
proxy.name // 'xxx'
obj.name // 'lyf'
revoke() // 撤销代理
proxy.name // throw error
```

反射 Reflect

内部有很多方法，并不是为了代理而生，下面还会有更详细的介绍

- 捕获代理，且在 Object 有对应 API
- 状态标记，即在不成功时返回布尔值，而不是 Object 抛出错误
- 代理操作符，例如 get、set、has、deleteProperty、construct

代理某代理

即秘书找了一个秘书，要达到目标对象，通过层层拦截与操作。

```
let obj = {money:100} // 老板给了 100 块
let fproxy = new Proxy(obj,{
    get(obj,prop){
        if(prop.includes('money')){
            console.log('一层代理吃了 90%回扣')
            return obj[prop]/10 // 一层代理吃了 90%回扣
        }
    }
})
let sproxy = new Proxy(fproxy,{
    get(obj,prop){
        if(prop.includes('money')){
            console.log('二层代理吃了 90%回扣')
            return obj[prop]/10 // 二层代理吃了 90%回扣
        }
    }
})
sproxy.money // '二层代理吃了 90%回扣' '一层代理吃了 90%回扣' 1
```

代理的问题

一些细节，懒得看了，TODO

代理捕获器

基本上就是改写 Object 的方法

- get(target,property,proxy) 对应的反射为 Reflect.get.
- set(target,property,value,proxy) 对应的反射为 Reflect.set,返回一个布尔值，表示设置成功与否。
- has(target,property) 对应的反射为 Relect.has(),必须返回一个布尔值
- defineProperty() 对应的反射为 Relect.defineProperty()
- getOwnPropertyDescritor()
- getPrototypeOf()

十、函数

声明函数的方式

- 声明式 function xxx(){} 可以提升到顶部
- 表达式 let xxx = function(){}
- 箭头函数 let xxx = ()=>{} 当只有一行时，默认作为 return 值，当只有一个参数时不需要括号，多个或 0 个时需要。当只有一行代码时不需要{}号
- 实例化 Function 函数 let xxx = new Function() 所有参数都是字符串，前 n-1 个参数作为函数的参数，最后一个参数作为函数体

函数属性

Function.name 属性保存函数名，箭头函数返回空字符串，new Function 创建的函数返回 'anonymous' 字符串。 Function.arguments 属性存储所有参数，是一个类数组对象，但是箭头函数没有此属性。 Function.length 属性表示参数个数 Function.prototype 属性表示实例的所有共享属性，指向原型对象 js 中函数没有重载，即同名的函数只有一个。 ES6 中支持**参数默认值**，即在定义时赋值 function sayName(name= 'lyf')，不传参时 name 默认为'lyf'，参数默认值可以将前参数默认值赋值给后面的参数，反之报错。还有可以使用**扩展操作符**传入数组

```
function Name(name = 'lyf', show = name){}
function Name(...arr){}
```

函数内部

即在函数内部直接使用，arguments、this、和 new.target（表示是否是 new 实例化）。arguments 有一个属性 callee,指向函数，可以用来递归解耦。此外还有一个 caller 属性，这个属性引用的是调用当前函数的函数

```
function deepcopy(obj){
    if((typeof obj.name == 'object' || typeof obj.name == 'function') && obj.name
!== null)
    deepcopy(obj.name) // 满足一定条件时递归，但是当函数修改名称时，内部也需跟着修改，
否则报错
    arguments.callee(obj.name) // 在严格模式下报错
    test()
}
function test(){
    console.log(test.caller) // 即为 deepcopy 函数
}
arguments.callee.caller == Function.caller
```

函数方法

Function.call(thisObj,...arguments) 参数 1 为 this 指向的上下文，参数 2 ~ n 表示传给函数的参数，调用时返回函数执行结果 Function.apply(thisObj,arguments) 参数 1 为 this 指向的上下文，参数 2 表示传给函数的参数，为一个数组，调用时返回函数执行结果 Function.bind(thisObj,...arguments) 参数 1 为 this 指向的上下文，参数 2 ~ n 表示传给函数的参数，调用时返回一个上下文为 thisObj 的函数 B,需调用此返回值 B,且 B 仍可以传参数，排在 ...arguments 后面

```
let obj = {name:'lfy'}
let name = 'zzz'
function testName(a,b){
    console.log(this.name,a,b)
}
testName.call(obj,10) // 'lfy' 10 undefined
testName.apply(obj,[10,20]) // 'lfy' 10 20
let testBind = testName.bind(obj,10)
testBind(20) // 等于 testName.call(obj,10,20) // 'lfy' 10 20
```

尾调用

即在函数 A 尾巴进行 return 时，返回的是一个函数 B 调用，由于函数 A 结束了，故 A 先销毁并弹出栈，此时只剩一个栈帧，执行完 B，B 销毁弹出栈下，即无论调用多 少次嵌套函数，都只有一个栈帧。

- 必须有 return
- 必须在 return 中调用外部函数
- 外部函数没有其他逻辑，例如加减乘除，toString()

```
function testFun(){
    return test() + 1 // 存在其他逻辑、表达式
}
```

闭包

即在函数中引用了其他函数的变量，通常是嵌套函数由内部函数B引用外部函数A。 函数形成上下文，上下文形成作用域链，作用域链自下往上记录当前上下文可以操作的变量，作用域上的节点构成变量对象，函数的变量对象在函数执行完毕后销毁，故称为活动对象。 因为被函数B引用，函数A在调用完毕后仍不能被回收，虽然作用域链断了,但是活动对象一直保存,需手动释放内存。

```
function Farther(){
    let num = 10
    return function(){
        let num_1 = num*10;
        console.log(num_1)
    }
}
let son = Farther()
son() // 100
son() // 100 不论调用多少次,仍能访问到Farther函数的变量 num = 10
son = null // 释放内存
```

- 闭包中的 this 指向 this 对象会在运行时绑定到执行函数的上下文。若在全局函数中调用，则 this 为 window。如果作为某个对象的方法调用，则 this 等于这个对象。 但嵌套函数B无法访问到函数A的this,

故若在嵌套函数中使用this,仍访问window,故需要将this赋值给that

```
var name = 'lyf' // 使用let的不会变成window
let obj = {
    name: 'kobe'
}
function Farther(){
    console.log(this.name) // this 指向上下文
    return function(){
        console.log(this.name) // this 指向 window
    }
}
obj.showName = Farther

let son = Farther()
son() // 'lyf' 'lyf'
let oson = obj.showName()
oson() // 'Kobe' 'lyf' 虽是obj调用函数,但此时this仍指向window

// 修改this赋值给that
function Farther_1(){
    console.log(this.name)
    let that = this
    return function(){
        console.log(that.name) // that一直指向函数A的this
    }
}
let obj_1 = {
    name: 'kobe'
}
obj_1.showName = Farther_1

let son_1 = Farther_1()
son_1() // 'lyf' 'lyf'
let oson = obj_1.showName()
oson() // 'Kobe' 'kobe'
```

IIFE

与之相反的是立即执行函数,即生成就调用,一般无法访问IIFE内部的变量,除非有意抛出

私有变量和静态私有变量

私有变量包括函数参数、局部变量，以及函数内部定义的其他函数,但是又能通过闭包进行访问变成公有的,所以很bug啊,可以访问私有变量的公共方法叫作特权方法。 特权方法可以使用构造函数或原型模式通过自定义类型中实现，也可以使用模块模式或模块增强模式在单例对象上实现。

- 构造函数 在函数内部中定义一个全局函数,里面访问函数的私有变量
- 原型模式 在函数内部中定义一个全局函数,在函数原型上访问函数的私有变量
- 模块模式 在函数中返回一个对象字面量{},属性包含函数的私有变量

- 单例对象 创建对象,给它添加额外函数的私有变量属性或方法,将操作完的对象返回

```
(function (){
  publish = function (){}
  function  myPersonal(){
      return false
  }
  publish.prototype.showFarther = function(){
      return myPersonal()
  }
})()

let my = new publish()
my.showFarther() // false 原型模式 虽然可以访问函数内部的私有变量,但是放置在函数的原型对象上,即静态的
```

十一、期约（Promise）和异步函数