JS学习总结

作用域

作用域是在运行时代码中的某些特定部分中变量，函数和对象的可访问性。
作用域最大的用处就是隔离变量，不同作用域下同名变量不会有冲突。
ES6 之前 JavaScript 没有块级作用域,只有全局作用域和函数作用域。ES6 的到来，为我们提供了‘块级作用域’,可通过新增命令 let 和 const 来体现。

全局作用域，函数作用域，块级作用域

作用域链（Scope Chain）

标识符解析是沿着作用域链一级一级地搜索标识符地过程。搜索过程始终从作用域链地前端开始，然后逐级向后回溯，直到找到标识符为止（如果找不到标识符，通常会导致错误发生）
自由变量：当前作用域没有定义的变量，称为自由变量。
作用域链：一层一层向上寻找自由变量，直到找到全局作用域还是没找到，就宣布放弃。这种一层一层的关系，就是作用域链。
自由变量的取值：在创建函数的作用域中取值

```
var x = 10
function fn() {
  console.log(x)
}
function show(f) {
  var x = 20
  (function() {
    f() //10，而不是20
  })()
}
show(fn)
```

在 fn 函数中，取自由变量 x 的值时，要到创建 fn 函数的那个作用域中取，**无论 fn 函数将在哪里调用。**
作用域中取值,这里强调的是“创建”，而不是“调用”，切记切记——其实这就是所谓的"静态作用域"

作用域与执行上下文

JavaScript 的执行分为：解释和执行两个阶段,这两个阶段所做的事并不一样：

解释阶段：

- 词法分析
- 语法分析
- 作用域规则确定

执行阶段：

- 创建执行上下文
- 执行函数代码
- 垃圾回收

JavaScript 解释阶段便会确定作用域规则，因此作用域在函数定义时就已经确定了，而不是在函数调用时确定，但是执行上下文是函数执行之前创建的。执行上下文最明显的就是 this 的指向是执行时确定的。而作用域访问的变量是编写代码的结构确定的。

作用域和执行上下文之间最大的区别是：
执行上下文在运行时确定，随时可能改变；作用域在定义时就确定，并且不会改变。

一个作用域下可能包含若干个上下文环境。有可能从来没有过上下文环境（函数从来就没有被调用过）；有可能有过，现在函数被调用完毕后，上下文环境被销毁了；有可能同时存在一个或多个（闭包）。同一个作用域下，不同的调用会产生不同的执行上下文环境，继而产生不同的变量的值。

参考 https://www.cnblogs.com/fundebug/p/10535230.html
<https://www.cnblogs.com/fundebug/p/10535230.html>

闭包

函数与对其状态即**词法环境**（**lexical environment**）的引用共同构成**闭包**（**closure**）。也就是说，闭包可以让你从内部函数访问外部函数作用域。**在JavaScript，函数在每次创建时生成**

全局变量：可以重用、但是会造成全局污染而且容易被篡改
局部变量：仅函数内使用不会造成全局污染也不会被篡改、不可以重用
闭包的作用：既重用一个对象，又保护对象不被污染篡改

```
var age = "21";
    function myAge(){
        var age = 0;
        age++;
        console.log(age);
    }
    myAge(); // 1
    console.log(age); // 21

function addAge(){
        var age = 21;
        return function(){
            age++;
            console.log(age);
        }
    }
    var clourse = addAge();
    clourse(); // 22
    clourse(); // 23
    clourse(); // 24
```

```
var name = "The Window";
    var object = {
        name : "My Object",
        getNameFunc : function(){
            return function(){
                return this.name;
            };
        }
    };
alert(object.getNameFunc()());
```

this对象是在运行时基于函数的执行环境绑定的：在全局函数中，this等于window，而当函数被作为某个对象调用时，this等于那个对象。不过，匿名函数具有全局性，因此this对象同常指向window

原型

普通对象与函数对象

JavaScript 中，万物皆对象！但对象也是有区别的。分为**普通对象和函数对象**，Object 、Function 是 JS 自带的函数对象。
凡是通过 new Function() 创建的对象都是函数对象，其他的都是普通对象。

构造函数

```
function Person(name, age, job) {
 this.name = name;
 this.age = age;
 this.job = job;
 this.sayName = function() { alert(this.name) }
}
var person1 = new Person('Zaxlct', 28, 'Software Engineer');
```

上面的例子中 person1 和 person2 都是 Person 的**实例**。这两个**实例**都有一个 `constructor` （构造函数）属性，该属性（是一个指针）指向 Person。 即：

```
console.log(person1.constructor == Person); //true
console.log(person2.constructor == Person); //true
```

person1 和 person2 都是 构造函数 Person 的实例
实例的构造函数属性（constructor）指向构造函数。

原型对象

在 JavaScript 中，每当定义一个对象（函数也是对象）时候，对象中都会包含一些预定义的属性。其中每个**函数对象**都有一个 `prototype` 属性，这个属性指向函数的**原型对象**。

每个对象都有 __proto__ 属性，但只有函数对象才有 prototype 属性
在默认情况下，所有的**原型对象**都会**自动获得**一个 `constructor` （构造函数）属性，这个属性（是一个指针）指向 `prototype` 属性所在的函数（Person），即 `Person.prototype.constructor == Person`
上文提到**实例的构造函数属性（constructor）指向构造函数**，即`person1.constructor == Person`

结论：原型对象（Person.prototype）是 构造函数（Person）的一个实例。

原型对象其实就是普通对象（**但 Function.prototype 除外，它是函数对象，但它很特殊，他没有prototype属性（前面说道函数对象才有prototype属性）**）。

```
console.log(typeof Function.prototype) // function
console.log(typeof Object.prototype)   // object
console.log(typeof Number.prototype)   // object
console.log(typeof Boolean.prototype)  // object
console.log(typeof String.prototype)   // object
console.log(typeof Array.prototype)    // object
console.log(typeof RegExp.prototype)   // object
console.log(typeof Error.prototype)    // object
console.log(typeof Date.prototype)     // object
```

看下面的例子：

```
function Person(){};
console.log(Person.prototype) //Person{}
console.log(typeof Person.prototype) //Object
console.log(typeof Function.prototype) // Function，这个特殊
console.log(typeof Object.prototype) // Object
console.log(typeof Function.prototype.prototype) //undefined
```

凡是通过 new Function( ) 产生的对象都是函数对象。

```
var test = new Function()
typeof Function.prototype // "function"
typeof test.prototype // "object"
```

__proto__

JS 在创建对象**（不论是普通对象还是函数对象）**的时候，都有一个叫做`__proto__`的内置属性，用于指向创建它的构造函数的原型对象。

对象 person1 有一个`__proto__`属性，创建它的构造函数是 Person，构造函数的原型对象是 Person.prototype ，所以：

`person1.__proto__ == Person.prototype`

构造器

`var obj = {}` 等同于 `var obj = new Object()`

obj 是构造函数（Object）的一个实例。所以：

`obj.constructor === Object`

`obj.__proto__ === Object.prototype`

内建 JavaScript 构造器

```
var x1 = new Object();    // 一个新的 Object 对象
x1.constructor === Object;
x1.__proto__ === Object.prototype;

var x2 = new String();    // 一个新的 String 对象
x2.constructor === String;
x2.__proto__ === String.prototype;

var x3 = new Number();    // 一个新的 Number 对象
x3.constructor === Number;
x3.__proto__ === Number.prototype;

var x4 = new Boolean();   // 一个新的 Boolean 对象
x4.constructor === Boolean;
x4.__proto__ === Boolean.prototype;

var x5 = new Array();     // 一个新的 Array 对象
x5.constructor === Array;
x5.__proto__ === Array.prototype;

var x6 = new RegExp();    // 一个新的 RegExp 对象
x6.constructor === RegExp;
x6.__proto__ === RegExp.prototype;

var x7 = new Function();  // 一个新的 Function 对象
x7.constructor === Function;
x7.__proto__ === Function.prototype;

var x8 = new Date();      // 一个新的 Date 对象
x8.constructor === Date;
x8.__proto__ === Date.prototype;
```

```
> typeof Object
< "function"
> typeof Function
< "function"
> typeof Array
< "function"
> typeof Date
< "function"
> typeof Number
< "function"
> typeof String
< "function"
> typeof Boolean
< "function"
```

原型链

每个对象都可以有一个原型_proto_，这个原型还可以有它自己的原型，以此类推，形成一个原型链。

```
person1.__proto__ === Person.prototype
Person.prototype.__proto__ === Object.prototype
Object.prototype.__proto__ === null

Person.__proto__ === Function.prototype
Object.__proto__ === Function.prototype
Function.prototype.__proto__ === Object.prototype
```

```
Number.__proto__ === Function.prototype  // true
Number.constructor == Function //true

Boolean.__proto__ === Function.prototype // true
Boolean.constructor == Function //true

String.__proto__ === Function.prototype  // true
String.constructor == Function //true

// 所有的构造器都来自于Function.prototype，甚至包括根构造器Object及Function自身
Object.__proto__ === Function.prototype  // true
Object.constructor == Function // true

// 所有的构造器都来自于Function.prototype，甚至包括根构造器Object及Function自身
Function.__proto__ === Function.prototype // true
Function.constructor == Function //true

Array.__proto__ === Function.prototype   // true
Array.constructor == Function //true

RegExp.__proto__ === Function.prototype  // true
RegExp.constructor == Function //true

Error.__proto__ === Function.prototype   // true
Error.constructor == Function //true

Date.__proto__ === Function.prototype    // true
Date.constructor == Function //true
```

JavaScript中有内置(build-in)构造器/对象共计12个（ES5中新加了JSON），这里列举了可访问的8个构造器。剩下如Global不能直接访问，Arguments仅在函数调用时由JS引擎创建，Math，JSON是以对象形式存在的，无需new。它们的**proto**是Object.prototype。如下

```
Math.__proto__ === Object.prototype  // true
Math.construrctor == Object // true

JSON.__proto__ === Object.prototype  // true
JSON.construrctor == Object //true
```

Prototype

函数对象的`prototype`**属性指向函数的原型对象。**

对于 ECMAScript 中的引用类型而言，`prototype` 是保存着它们所有实例方法的真正所在。换句话所说，诸如 `toString()` 和 `valueOf()` 等方法实际上都保存在 `prototype` 名下，只不过是通过各自对象的实例访问罢了。

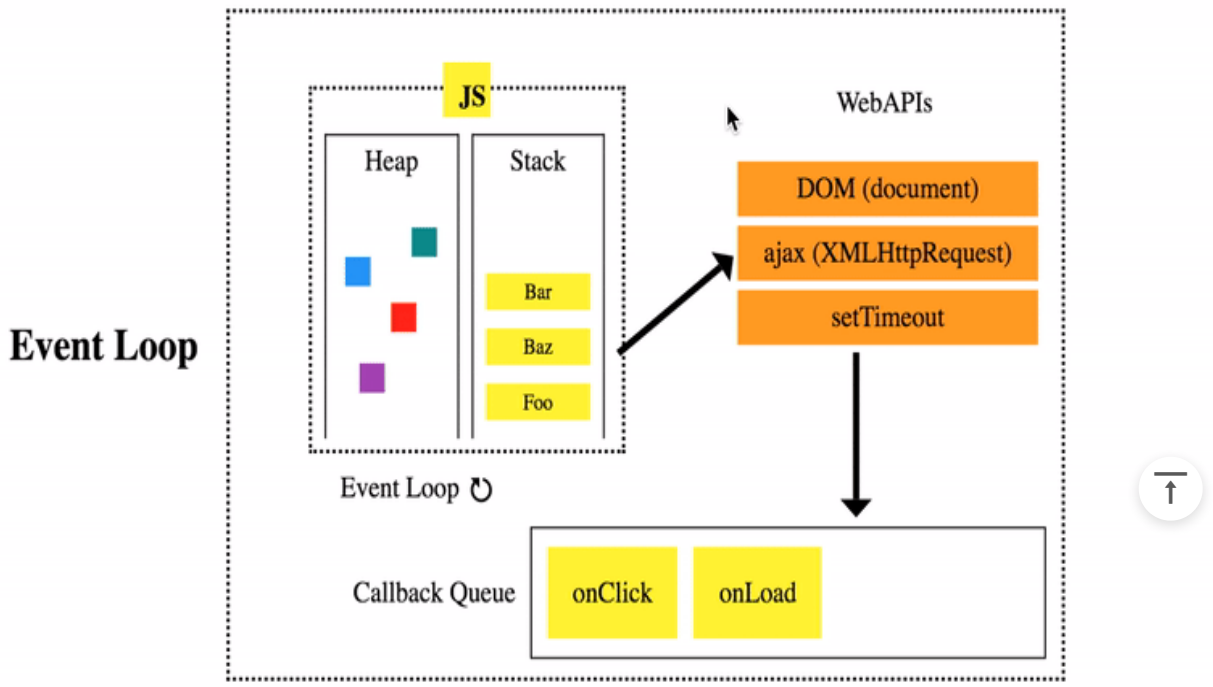
对象可以用 `constructor/toString()/valueOf()` 等方法;
数组可以用 `map()/filter()/reducer()` 等方法;
数字可用用 `parseInt()/parseFloat()`等方法;

当我们创建一个数组时:
`var num = new Array()`
`num` 是 `Array` 的实例，所以 `num` **继承**了`Array` 的原型对象`Array.prototype`上所有的方法:

总结

- 原型和原型链是JS实现继承的一种模型。
- 原型链的形成是真正是靠`__proto__` 而非`prototype`

Event Loop



宏任务macro-task

js代码块、setTimeout、setInterval、I/O、MessageChannnel（浏览器），requestAnimationFrame（node）

setTimeout含义是定时器，到达一定的时间触发一次，但是setInterval含义是计时器，到达一定时间触发一次，并且会持续触发

I/O （输入输出）操作
读取文件/文件返回，http请求/网络返回，SQL查询/网络返回

MessageChannnel `var channel = new MessageChannel();` 创建消息通道
1.可用于深拷贝，相比JSON.parse(JSON.stringify(object))，可以拷贝`undefined`和循环引用的对象，但不能拷贝有函数的对象。
2.当我们使用多个`web worker`并想要在两个`web worker`之间实现通信的时候，也可使用`MessageChannel`
（web worker）为js创建多线程环境。

微任务micro-task

promise，MutationObserver

MutationObserver 接口提供了监视对DOM树所做更改的能力。

setTimeOut，async/await，Promise

async 返回一个promise对象，await 返回promise的结果。

Promise是一个对象，它代表了一个异步操作的最终完成或者失败。（MDN）
.then 返回一个全新的 Promise，和原来的不同，可链式调用。
Promise对象用于异步操作，它表示一个尚未完成且预计在未来完成的异步操作。（立即执行）

```
async function async1() {
   console.log('async1 start');
   await async2();
   console.log('async1 end');
}
async function async2() {
   console.log('async2');
}
console.log('script start');
```

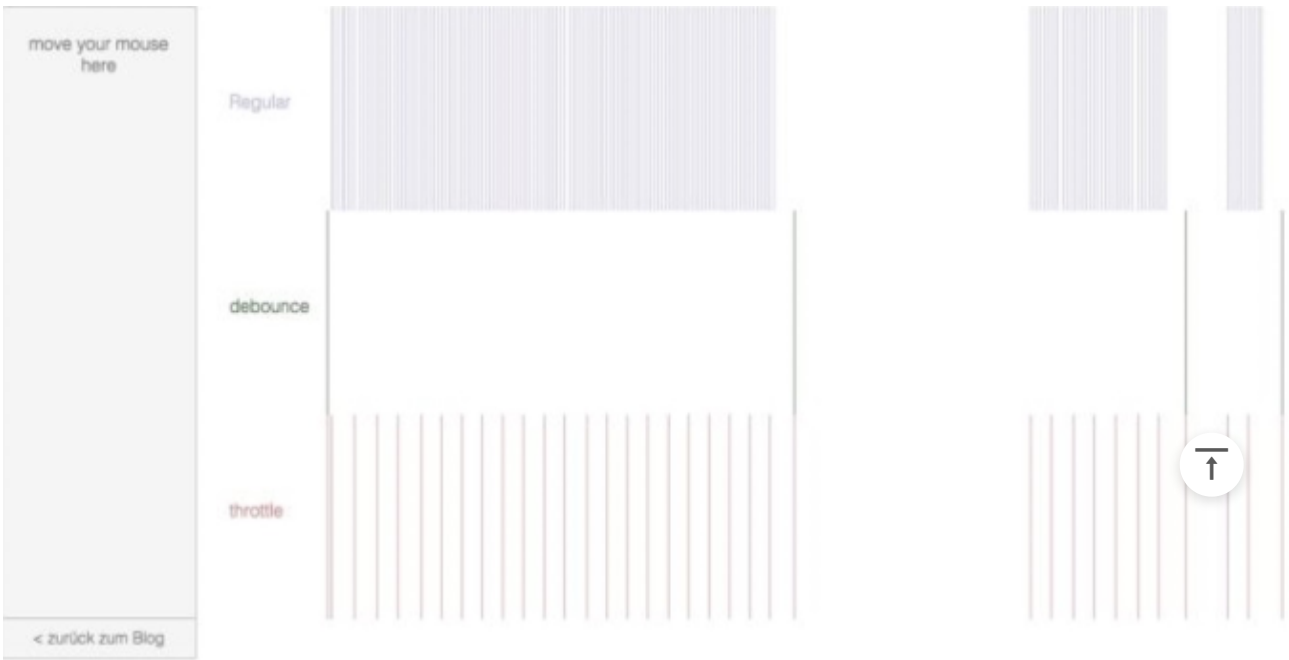
```
setTimeout(function() {
    console.log('setTimeout');
}, 0)
async1();
new Promise(function(resolve) {
    console.log('promise1');
    resolve();
}).then(function() {
    console.log('promise2');
});
console.log('script end');
```

```
script start                                    VM730:9
async1 start                                    VM730:2
async2                                          VM730:7
promise1                                        VM730:15
script end                                      VM730:20
async1 end                                      VM730:4
promise2                                        VM730:18
undefined
setTimeout                                      VM730:11
```

throttle节流 debounce防抖

函数节流: 指定时间间隔内只会执行一次事件;

函数防抖: 在事件被触发n秒后再执行回调，如果在这n秒内又被触发，则重新计时。



Set

set类似数组，但成员值都是唯一的没有重复值。`Set`本身是一个构造函数，用来生成 Set 数据结构。`new Set()`
使用add()添加成员，可用作数组去重 `Array.from(new Set(array));`
其判断重复的方法类似 `===` ，但NaN 等于 NaN。

方法

Set 实例的属性和方法

Set 结构的实例有以下属性。

- `Set.prototype.constructor`：构造函数，默认就是`Set`函数。
- `Set.prototype.size`：返回`Set`实例的成员总数。

Set 实例的方法分为两大类：操作方法（用于操作数据）和遍历方法（用于遍历成员）。下面先介绍四个操作方法。

- `Set.prototype.add(value)`：添加某个值，返回 Set 结构本身。
- `Set.prototype.delete(value)`：删除某个值，返回一个布尔值，表示删除是否成功。
- `Set.prototype.has(value)`：返回一个布尔值，表示该值是否为`Set`的成员。
- `Set.prototype.clear()`：清除所有成员，没有返回值。

遍历操作

Set 结构的实例有四个遍历方法，可以用于遍历成员。

- `Set.prototype.keys()`：返回键名的遍历器
- `Set.prototype.values()`：返回键值的遍历器
- `Set.prototype.entries()`：返回键值对的遍历器
- `Set.prototype.forEach()`：使用回调函数遍历每个成员

需要特别指出的是，`Set`的遍历顺序就是插入顺序。这个特性有时非常有用，比如使用 Set 保存一个回调函数列表，调用时就能保证按照添加顺序调用。

WeakSet

WeakSet 结构与 Set 类似，也是不重复的值的集合。但是，它与 Set 有两个区别。

首先，WeakSet 的成员只能是对象，而不能是其他类型的值。

其次，WeakSet 中的对象都是弱引用，即垃圾回收机制不考虑 WeakSet 对该对象的引用，也就是说，如果其他对象都不再引用该对象，那么垃圾回收机制会自动回收该对象所占用的内存，不考虑该对象还存在于 WeakSet 之中。

```
const ws = new WeakSet();

const a = [[1, 2], [3, 4]];
const ws = new WeakSet(a);
// WeakSet {[1, 2], [3, 4]}

const b = [3, 4];
const ws = new WeakSet(b);
// Uncaught TypeError: Invalid value used in weak set(…)
```

WeakSet 结构有以下三个方法。

- **WeakSet.prototype.add(value)**：向 WeakSet 实例添加一个新成员。
- **WeakSet.prototype.delete(value)**：清除 WeakSet 实例的指定成员。
- **WeakSet.prototype.has(value)**：返回一个布尔值，表示某个值是否在 WeakSet 实例之中。

没有 size 和 forEach 属性，因为成员都是弱引用，随时可能消失。

Map

map类似对象，是键值对的集合，但【键】的范围不局限于字符串，各种类型的值（包括对象）都可以当作键。也就是说，Object 结构提供了“字符串—值”的对应，Map 结构提供了“值—值”的对应，是一种更完善的 Hash 结构实现。如果你需要“键值对”的数据结构，Map 比 Object 更合适。

方法

（1）size 属性

（2）Map.prototype.set(key, value)

（3）Map.prototype.get(key)

（4）Map.prototype.has(key)

（5）Map.prototype.delete(key)

（6）Map.prototype.clear()

遍历方法

Map 结构原生提供三个遍历器生成函数和一个遍历方法。

- `Map.prototype.keys()`：返回键名的遍历器。
- `Map.prototype.values()`：返回键值的遍历器。
- `Map.prototype.entries()`：返回所有成员的遍历器。
- `Map.prototype.forEach()`：遍历 Map 的所有成员。

需要特别注意的是，Map 的遍历顺序就是插入顺序。

WeakMap

`WeakMap`结构与`Map`结构类似，也是用于生成键值对的集合。

`WeakMap`与`Map`的区别有两点。
首先，`WeakMap`只接受对象作为键名（`null`除外），不接受其他类型的值作为键名。
其次，`WeakMap`的键名所指向的对象，不计入垃圾回收机制。

`WeakMap`只有四个方法可用：`get()`、`set()`、`has()`、`delete()`。

GC

标记清除法（最常用）

标记清除（Mark and Sweep）是最早开发出的GC算法（1960年）。它的原理非常简单，首先从根开始将可能被引用的对象用递归的方式进行标记，然后将没有标记到的对象作为垃圾进行回收。标记清除算法有一个缺点，就是在分配了大量对象，并且其中只有一小部分存活的情况下，所消耗的时间会大大超过必要的值，这是因为在清除阶段还需要对大量死亡对象进行扫描。

复制收集法

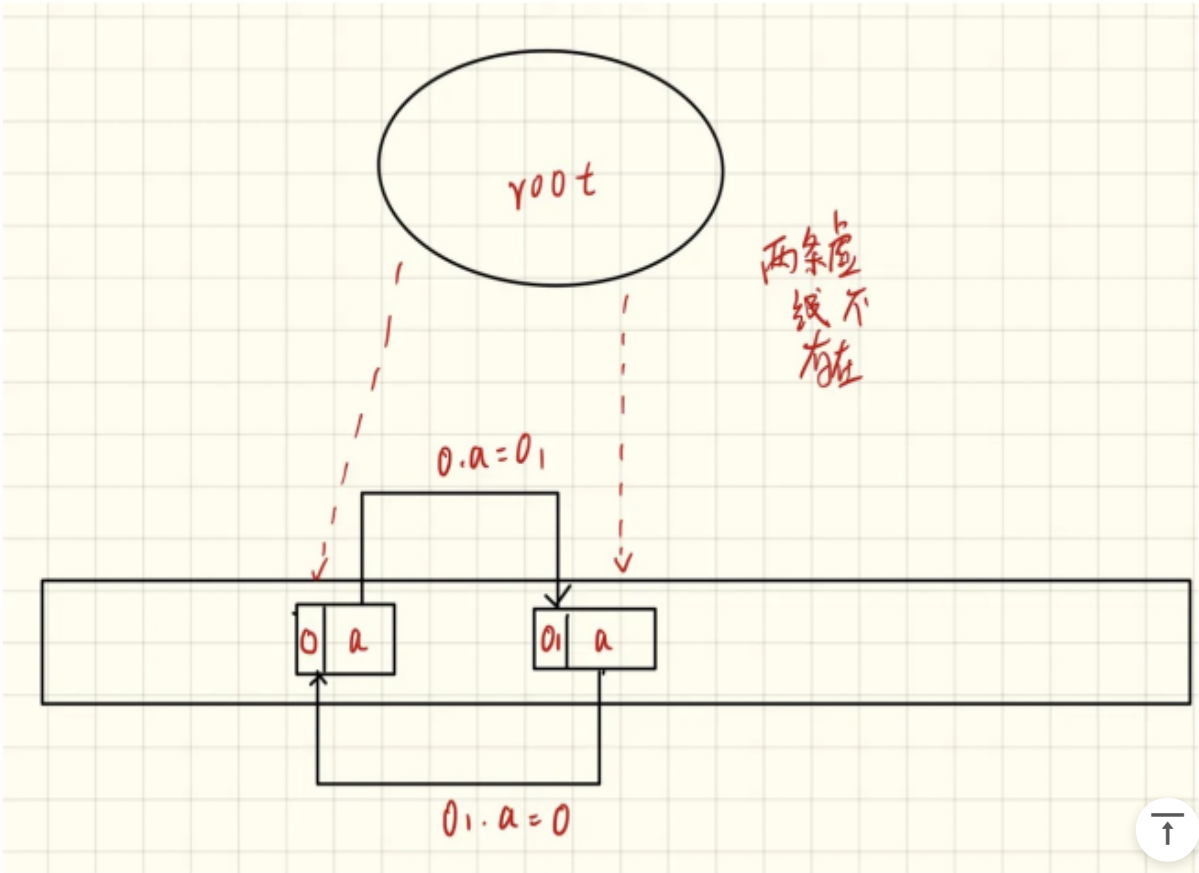
复制收集（Copy and Collection）则试图克服这一缺点。在这种算法中，会将从根开始被引用的对象复制到另外的空间中，然后再将复制的对象所能够引用的对象用递归的方式不断复制下去。复制完成之后，“死亡”对象就被留在了旧空间中。将旧空间废弃掉，就可以将死亡对象所占用的空间一口气全部释放出来，而没有必要再次扫描每个对象。下次GC的时候，现在的新空间也就变成了将来的旧空间。

引用计数法

引用计数（Reference Count）方式是GC算法中最简单也最容易实现的一种，它和标记清除方式差不多是在同一时间发明出来的。它的基本原理是，在每个对象中保存该对象的引用计数，当引用发生增减时对计数进行更新。引用计数的增减，一般发生在变量赋值、对象内容更新、函数结束（局部变量不再被引用）等时间点。当一个对象的引用计数变为0时，则说明它将来不会再被引用，因此可以释放相应的内存空间。引用计数最大的缺点，就是无法释放循环引用的对象。

https://segmentfault.com/a/1190000004665100
<https://segmentfault.com/a/1190000004665100>
疑惑之处：标记清除法如何解决循环引用问题？

> 这种方法可以解决循环引用问题，因为两个对象从全局对象出发无法获取。因此，他们无法被标记，他们将会被垃圾回收器回收。正如图：



碎知识点

纯函数

一个函数的执行结果只依赖其参数执行过程中不会产生副作用。
副作用来自，但不限于：
进行一个 HTTP 请求
Mutating data
输出数据到屏幕或者控制台
DOM 查询/操作
Math.random()
获取的当前时间

key

key用于帮助react 识别哪些内容被更改，添加，删除。key需要被赋予一个稳定值。其稳定的必要性在于如果key发生变化，react会触发UI重渲染。key在兄弟节点之间必须唯一，不需要全局唯一。如果出现相同的key ，react只会渲染第一个重复key中元素，并认为后续相同key是一个组件。（不要用index作为key，宜用id）

原始数据类型和引用类型的区别

原始数据类型在内存中是栈存储，引用类型是堆存储 栈（stack）为自动分配的内存空间，它由系统自动释放；而堆（heap）则是动态分配的内存，大小不定也不会自动释放。在内存中存储方式的不同导致了原始数据类型不可变 原始数据类型和引用数据类型做赋值操作一个是传值一个是传址

面向对象的三个基本特征

面向对象的三个基本特征是：封装、继承、多态。

1，Number(1)，new Number(1)的区别

1是原始数据类型 ，new Number(1)是引用数据类型。（typeof）

this是什么

当前执行代码的环境对象（MDN）
首先this作为关键字，它的作用就是引用，并且它通常只写在函数内部就是函数体内，在js中this的引用对象随着函数的使用环境变化而变化。普通函数、构造函数、对象属性、apply&call&bind方法、箭头函数

- 普通函数的this指向window；
- 构造函数指向实例；
- apply&call&bind指向传入的第一个参数（apply传数组，call传参数，直接执行函数，bind和call使用方式一样但它只传参不直接执行）；
- 作为对象的属性，this指向对象；
- 在箭头函数 <https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Functions/Arrow_functions> 中，`this`与封闭词法环境的`this`保持一致。

js设计模式

工厂模式：

单例模式：只允许存在一个实例的模式

观察者模式：

策略模式：

代理模式：

===与==

===严格等于，比较类型和值
== 比较时可转换数据类型
转换数据类型规则：null == undefined true；string，Boolean与number比转换成number；string，symbol，number与Object比，`toPrimitive(Object)` （原始值转换算法）

undefined与null

undefined表未定义，是一个变量而非关键字，可能被篡改，null表示定义了但为空。

DOM是什么

文档对象模型

事件传播

事件传播有3个阶段，捕获阶段->目标阶段->冒泡阶段
`event.preventDefault()` 方法可防止元素的默认行为。 如果在表单元素中使用，它将阻止其提交。 如果在锚元素中使用，它将阻止其导航。 如果在上下文菜单中使用，它将阻止其显示或显示。
`event.stopPropagation()`方法用于阻止捕获和冒泡阶段中当前事件的进一步传播。

提升

提升是用来描述变量和函数移动到其(全局或函数)作用域顶部的术语。只有使用`var`声明的变量，或者函数声明才会被提升（在编译阶段提升）。

虚值

'', 0, null, undefined, NaN, false

"use strict"

是 **ES5** 特性，它使我们的代码在函数或整个脚本中处于**严格模式。严格模式**帮助我们在代码的早期避免 bug，并为其添加限制。

- 变量必须声明后再使用
- 函数的参数不能有同名属性，否则报错
- 不能使用`with`语句
- 不能对只读属性赋值，否则报错
- 不能使用前缀 0 表示八进制数，否则报错
- 不能删除不可删除的属性，否则报错
- 不能删除变量`delete prop`，会报错，只能删除属性`delete global[prop]`
- `eval`不能在它的外层作用域引入变量

- `eval`和`arguments`不能被重新赋值
- `arguments`不会自动反映函数参数的变化
- 不能使用`arguments.callee`
- 不能使用`arguments.caller`
- 禁止`this`指向全局对象
- 不能使用`fn.caller`和`fn.arguments`获取函数调用的堆栈
- 增加了保留字（比如`protected`、`static`和`interface`）

设立”严格模式”的目的，主要有以下几个：

1. 消除Javascript语法的一些不合理、不严谨之处，减少一些怪异行为;
2. 消除代码运行的一些不安全之处，保证代码运行的安全；
3. 提高编译器效率，增加运行速度；
4. 为未来新版本的Javascript做好铺垫。

纯函数