1.执行上下文/作用域链/闭包

2021/06/24 15:31

1.1 JavaScript的执行上下文

执行上下文

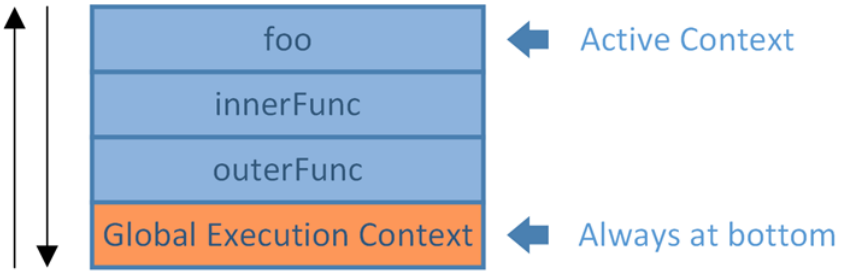
在JavaScript中有三种代码运行环境

- Global Code
  - JavaScript刚开始运行时的执行环境
- Function Code
  - 代码进入函数时的环境
- Eval Code

为了表示不同环境的执行上下文，在JavaScript运行时就会进入不同环境的上下文，这些执行上下文就进入了一个执行环境栈

```
var a = 'global val'
function foo(){

}
function outerFunc(){
   var b = 'outerFunc var'
   function innerFunc(){
       foo()
   }
   innerFunc()
}
outerFunc()
```



当代码执行时Global Execution Context总是在ECS的最底部。

Global Execution Context有三个属性，VO（Variable object），作用域链（scope chain），this

VO和AO

变量对象（Variable object）：是全局执行上下文的数据作用域，其中存储了上下文中的变量与函数声明

Global VO	
a	“Global var”
foo	<function>
outerFunc	<function>
<built-ins>	

注意

函数表达式与不用var声明的变量不在VO中

活动对象（Activation object）

当在全局执行上下文时，上下文的数据是可以通过VO去访问的，当进入函数时，是无法通过VO去访问的，此时Activation object就会被创建，它会担任VO的角色。激活对象是通过进入函数上下文时被创建

当上面的例子开始执行outerFunc时，以下对象会被创建

细看Execution Context

当代码执行时，会有两个阶段

创建阶段（函数被调用，但在执行内部函数前 ）

- 创建scope chain
- 创建VO/AO
- 创建this

激活执行代码阶段

- 设置变量的值，函数的引用

创建VO和AO时，主要做了以下【按照顺序】

- 根据函数的参数初始化arguments object
- 查找函数声明，并将函数和变量名存入VO/AO中
    - 如果变量名称与函数名相同，则不会干扰已存在的这类属性

```
function foo(i){
   var a = 1
   var b = function(){

   }
   function c(){
```

```
    }
}
foo(21)
```

对于上面的代码，在"创建阶段"，可以得到下面的Execution Context object：

```
fooExecutionContext = {
    scopeChain:{...},
    variableObject:{
        arguments:{
            0:21,
            length:1
        },
        i:22,
        a:undefined,
        b:undefined,
        c:pointer to function c()
    }
}
```

在代码激活阶段，进行了赋值操作，这里不再描述

来看几个例子

1. 不用var声明的变量AO中，而是在VO中

```
(function(){
    console.log(bar) //bar is not defined
    bar = 1
})()
```

2. 如果变量名称与函数名相同，则不会干扰已存在的这类属性

```
(function(){
    console.log(bar) //function
    function a(){

    }
    var a =1
})()
```

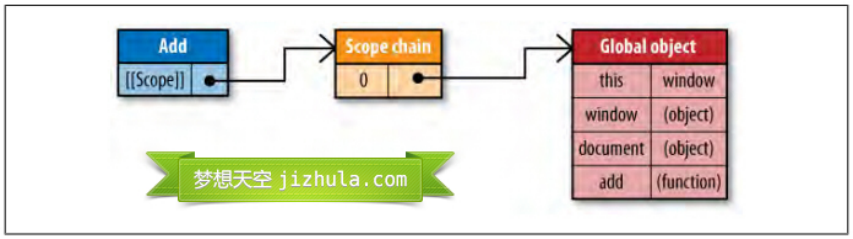
1.2作用域链

1.2.1概念

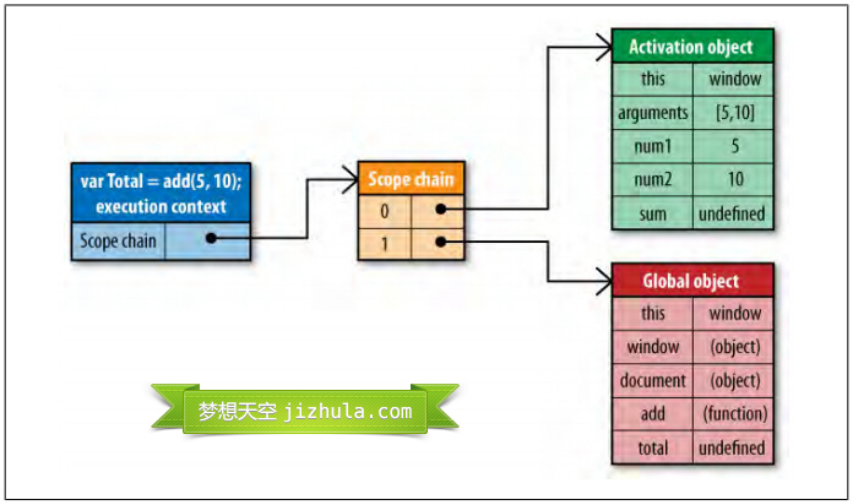
函数和其他对象一样，可以通过代码访问属性，其中一个内部属性就是[[scope]],也就是函数被创建的作用域链的集合

当一个函数创建之后，它的作用域链会被创建此函数的作用域中填充进去

```
function add(num1,num2) {
    var sum = num1 + num2;
    return sum;
}
```



运行时会创建一个运行期上下文，每个运行期上下文都有自己的作用域链。函数中的值会被推入一个叫AO的活动对象中。如何AO也会被推入到作用域链的前端



在创建阶段，会为每个变量执行标识符过程。在该过程中，首先从VO中查找，找到了就使用此标识符，未找到就去下一个作用域链去查找，如果直到全局都没找到，则该标识符为未定义。

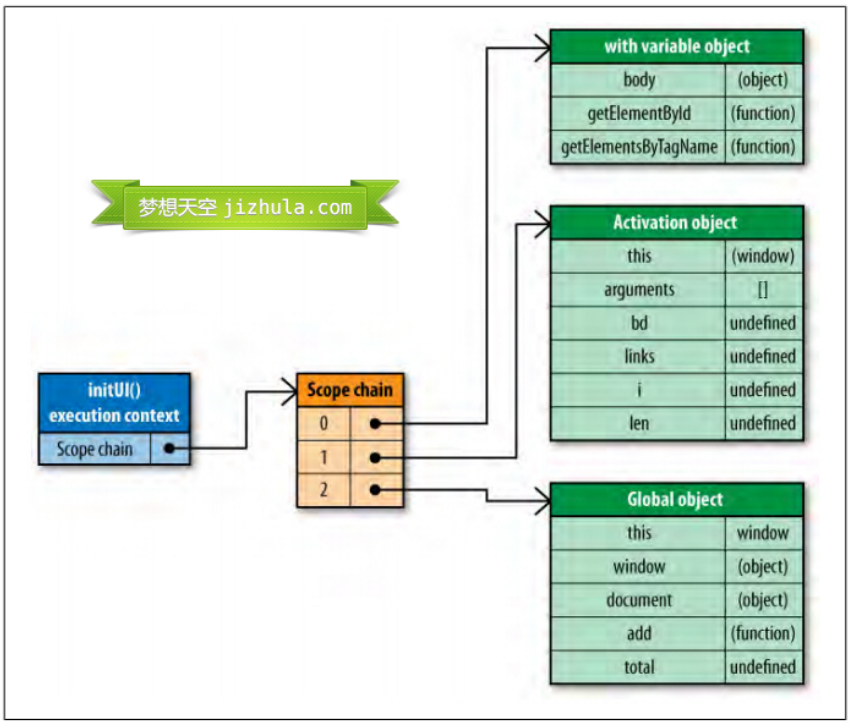
1.2.2作用域链和代码优化

可以看到，global变量总是在作用域链的最末端，所以，访问global变量也是最慢的。因此尽可能的减少使用全局变量，多使用局部变量。如果一个跨作用域链的对象被访问多次，不如将其存储在局部变量。

```
function changeColor(){
    var doc=document;
    doc.getElementById("btnChange").onclick=function(){
        doc.getElementById("targetCanvas").style.backgroundColor="red";
    };
}
```

1.2.3作用域链改变

try，with虽然能很好的解决跨作用域链的问题，但是它会改变作用域链



1.3闭包

应用场景

闭包的一个通常做法是在为某一函数执行前，先执行的函数提供参数，如果一段代码要延时调用、其还要传入参数，例如settimeout

1.settimeout

错误做法

```
//常见的错误就是虽然可以传递参数，但是无法按时调用
function a(v){
   console.log(v)
}
settimeout(a(1),2000) //代码立即会执行
```

正确做法

```
function a(v){
   console.log(v)
}
var b = a(1)
settimeout(b,2000)
```

2.回调函数。

- 由于函数执行完成就会被释放内存。所以无法在访问函数
- 回调函数会直接执行

错误做法

```
function changeSize(size){
   document.getElementById("h1").style.fontSize = size + "px"
}
//调用一次后就会释放掉内存无法再调用
//立即执行
document.getElementById('size-12').onclick = changeSize(30);
```

正确做法

```
function changeSize(size){
   return function(){
      document.getElementById("h1").style.fontSize = size + "px"
   }
}
var size14 = changeSize(14)
document.getElementById('size-12').onclick = size14;
```

3.防抖

```
function dbounce(fn,delay){
   var timer = null
   return function(){
      if(timer){ //已处于计时状态,重新开始计时调用函数
         clearTimeout(timer)
         timer=setTimeout(()=>{
            fn()
         },delay)
      }else{ //正常情况，开始一个计时
         timer=setTimeout(()=>{
            fn()
         },delay)
      }
   }
}
function a(){
   console.log(1)
}
btn.onclick = dbounce(a,1000)
```

4.访问私有变量

```
function a(){
   var b = 1
   return function(v){
      return b
   }
}
var c = a()
```

c(3) //3

2.this/call/apply/bind

2021/06/25 10:42

2.1this的理解

2.1.1概念

this的出现就是为了函数可以运行在不同的环境下。this就是出现在函数体内部，获取当前的运行环境

2.1.2对象的方法

this指向的就是当前运行时的对象

```
function a(){
    console.log(this) //window
}
```

如果对象里的方法包含this，那么this指向的就是当前运行时的对象

```
var a = {
    b:function(){
        console.log(this) //a
    }
};//注意封号
a.b()
```

内部的this指向a

下面的例子会改变this的指向

```
(a.b)()
(a.b = a.b)()
```

可以这样理解，在JavaScript引擎中，a和b存储在两个空间，称为地址1和地址2，调用a.b时，是从地址1调用地址2，this指向a。上面两种情况，是直接去地址2，所以指向全局。

如果this所在的方法指向的不是最外层对象的第一层，这时，this指向当前一层

```
var a = {
    b:{
        c:1,
        d:function(){
            console.log(this.c)
        }
    }
}
a.b.d() //undefined
```

如果达成效果，需写成下面这样

```
var e = a.b
e.d()
```

将d所在的对象赋给e，这样调用时，this就不会变

2.1.2使用注意点

1.避免多层this

由于this的指向是不确定的，所以切勿在函数中包含多层的this

```
var a = function(){
    b:function(){
        console.log(this)
        var c = function(){
            console.log(this)
        }()
    },
}
a.b()
//a
//window
```

实际上运行的是

```
var c = function(){
    console.log(this)
}
var a = function(){
    b:function(){
        console.log(this)
        var d = c
    }
}
var a = function
```

最好的做法是保存把this赋给一个变量

```
var a = {
    b:function(){
        var that = this
        console.log(that)
        var c = function(){
            console.log(that)
        }();
    }
}
a.b()
```

这是非常常见的做法，务必掌握

2.避免数组中处理方法的this

```
var a = {
    b:[3,4,5],
    e:1,
    c:function(){
        this.b.forEach(function(item){
            console.log(this.v+':'+item)
        })
    }
}
```

回调函数中的this指向顶层对象。可以使用中间变量固定this

另一种方式，当执行回调函数时，用作this的值

```
var a = {
    b:[3,4,5],
    e:1,
    c:function(){
        this.b.forEach(function(item){
            console.log(this.v+':'+item)
        },this)
    }
}
```

3.避免回调函数中的this

回调函数中的this会改变this的指向

```
var a = new Object()
a.b = function(){
    console.log(this==a)//执行时this指向的就是全局对象了
}
dom.click = a.b
```

2.2bind

bind方法用于将函数体内的this绑定的某个对象

```
var a = new Date()
a.getTime() //输出
var c = a.getTime
c() //Uncaught TypeError: this is not a Date object.
```

getTime内部的this给c后，this已经不再指向Date对象实例了。

bind方法可以解决这个问题

```
var c = a.getTime.bind(a)
c()
```

下面是更清晰的例子,和上面一个道理

```
var a = {
    b: 1,
    c: function () {
        this.b++
    }
}
var d = a.c
d()
```

使用bind方法

```
var a = {
    b: 1,
    c: function () {
        this.b++
    }
}
var d = a.c.bind(a)
d()
```

bind方法还可以接受更多的参数

```
var a = function (x, y) {
    return x * this.m + y * this.n
}
var b = {
    m: 3,
    n: 2
}
var c = a.bind(b, 4)
c(5) //20
```

将a函数的第一个参数x绑定成4，然后返回一个新函数，新函数再接受一个参数y就可以运行了。这样做的好处是，以后每次运行函数的时候，只需要传一个参数就够了

bind的第一个参数如果是null或undefined的话，默认指向的是全局

```
a.bind(null)
```

bind方法也有一些注意点

1.bind方法每次运行都会返回一个新函数，这会产生一些问题，比如监听事件

```
dom.addEventListener('click',a.bind(b))
```

bind方法生成的匿名函数，导致无法取消事件：

```
dom.removeEventListener('click',a.bind(b))
```

正确的做法：

```
var c = a.bind(b)
dom.addEventListener('click',c)
dom.removeEventListener('click',c)
```

2.结合回调函数使用

```
var a = {
    b:1,
    c:function(){
        this.b++
    }
}
function d(callBack){
    callBack()
```

```
}
d(a.c.bind(a))
```
如果直接将a.c传入，会导致this的指向错误。使用bind将a.c绑定到a对象后，就会避免

3.原型/继承

3.1原型

让所有实例对象都可以共享的属性和方法就称为原型

3.2原型链

3.2.1 概念

所有对象都有自己的原型，并且任何一个对象都可以充当其他对象的原型。另一方面，原型对象也是对象，它也有自己的原型，所以就会形成原型链

那么，Object.prototype有自己的原型吗？回答是null

```
Object.getPrototype(Object.prototype)
//null
```

读取对象的某个属性时，会优先在当前对象查找，找不到就去原型找，如果还是找不到就去原型的原型去找，直到最顶层的Object.prototype也找不到，返回undefined。如果对象本身与原型有同名属性，会优先读取对象，这叫做覆盖

注意，一级级向上找，对性能是有影响的，尤其是当变量不存在时，将会遍历整个原型链

举例来说，让构造函数的prototype指向数组，就意味着实例对象拥有数组的方法

```
function A(){

}
A.prototype = new Array()
A.prototype.constructor = a
A.push(1,2,3)
A.length = 3
var b = new A()
console.log(b instance Array) //true
```

3.2.2instanceof运算符

```
var a = new B()
a instanceof B //ture
```
左面是实例对象，右边是构造函数。它会检查右边的原型是否在左边的原型链上。因此等价于下面：
```
a.prototype.isPrototype(B)
```

3.3使用原型实现继承（借助原型链）

让一个构造函数继承另一个构造函数，这是很常见的需求，可以分两步实现

第一步，在子类的构造函数中，调用父类构造函数

```
function Child(){
    Parent.call(this)
}
```
第二步，让子类的原型指向父类的原型

```
Child.prototype = Object.create(Parent.prototype)
Child.prototype.constructor = Parent
```

举例来说

```
function Parent(){
    this.x = 0
    this.y = 0
}
Parent.prototype.add = function(x,y){
    this.x+=x
```

```
    this.y+=y
}
```
在子类调用构造函数
```
function Child(){
    Parent.call(this)
}
//另一种写法
function Child(){
    this.base = Parent
    this.base()
}
Child.prototype = Object.create(Parent.prototype)
Child.prototype.constructor = Parent
```

有时子类只需要继承父类的单个方法
```
function A() {
    this.x = 1
}
A.prototype.add = function (x) {
    this.x += x
}
function B() {
    A.call(this)
}
B.prototype.add = function (x) {
    A.prototype.add.call(this, x)
}
var c = new B()
c.add(2)
//c.x 3
```

4.js面试题基础篇（上）

2021/06/25

1.instanceof能否判断基础类型?

重点：使用Symbol.hasInstance

2.[] == ![]结果是什么？为什么?

==中，两边需转换成数字

左边：[]直接转换成数字为0

右边：由于存在!，所以需要先将[]转换为bollean为true，最终结果为false==0

所以相等

3.JS如何实现继承

常见的是，第一种是3.3中的使用原型实现继承（借助原型链），这里不再说明。。

第二种是寄生组合继承，最为常用
```
// 寄生组合继承
function D1() {
    this.name = '11'
    this.arr = [1, 2, 3]
}
D1.prototype.getName = function() {
    return this.name
}

function D2() {
    D1.call(this)
    this.type = 'D2'
}
```

```
// D2上有了D1的原型方法，并且可以再次往D2上添加属性
D2.sayHi = function() {
   console.log('hi');
}
let D3 = new D2()
let D4 = new D2()
D3.arr.push(4)
```

5.js面试题中难度（中）

2021/06/26 16:33

1 浅拷贝的手段有哪些?

基础解释

先说什么是拷贝

```
var a = [1,2,3]
var b = a
b[0] = 4
//a [4,2,3]
```

由于a和b引用的都是同一块空间，当b改变时，a也跟着改变

现在进行浅拷贝

```
var a = [1,2,3]
var b= a.slice(0)
b[0] = 4
//a [1,2,3]
```

现在a和b引用的不是同一块空间了，所以b改变，a不会改变

但是又会出现一个潜在的问题

```
var a = [1,2,{val:3}]
var b= a.slice(0)
b[3].val = 4
//a[3] {val:4}
```

对于这种有对象的嵌套，这就是浅拷贝的缺陷之处了，所以只能使用深拷贝

说说有几种方式

1. slice
2. assign

```
var a= {b:1,c:2}
var d = Object.assign({},a,{d:3})
console.log(d)//{b: 1, c: 2, d: 3}
```

3. concat

```
var a= [1,2]
var b = a.concat()
console.log(b)//[1,2]
```

4. . . 运算符

```
var a = [1,2]
var b = [...a]
```

代码

```
// 浅拷贝
const shallowClone = (target) => {
      if (typeof target == 'object' && target != null) {
         const cloneTarget = Array.isArray(target) ? [] : {}
         for (let prop in target) {
            cloneTarget[prop] = target[prop]
         }
         return cloneTarget
      } else {
         return target
      }
   }
```

深拷贝

1. 简易版

JSON.parse(JSON.stringify())

这个方法可以满足大部分的应用场景，但在严格意义下，是有巨大的问题的、

2. 无法解决循环引用

```
var a = {b:1}
a.target = a
```

拷贝a会出现栈溢出(指针指向问题)，a指向堆内存1，堆内存中的b又指向a的指针堆内存a，所以造成了无限递归。

那么该如何解决呢？

创建一个Map，记录拷贝过的对象，如果已经拷贝过，直接返回它就行了

```
// 深拷贝
const isComplexDataType = obj => (typeof obj === 'object' || typeof obj === 'function') && (obj !== null)
const deepClone = function(obj, hash = new WeakMap()) {
    if (obj.constructor === Date)
        return new Date(obj)
    if (obj.constructor === RegExp)
        return new RegExp(obj)
            // 遍历传入参数所有键的特性
            // 循环代码
            // ...
            // if (hash.has(obj)) return hash.get(obj)
    let allDesc = Object.getOwnPropertyDescriptors(obj)
    console.log(allDesc, 'allDesc');
    // 继承原型链//把不可枚举属性赋给新的对象
    let cloneObj = Object.create(Object.getPrototypeOf(obj), allDesc)
    console.log(cloneObj, 'cloneObj---28');
    // hash.set(obj, cloneObj)1
    for (let key of Reflect.ownKeys(obj)) {
        console.log(key, 'keyyyyy');
        // console.log(deepClone(obj[key]), 'hello');
        cloneObj[key] = (isComplexDataType(obj[key]) && typeof obj[key] !== 'function') ? deepClone(obj[key], hash) : obj[key]

    }
    return cloneObj
}
```

但是这样会造成一个问题，在程序结束之前，Map的key和value一直是强引用关系，会造成内存泄漏。那么弱引用是可以在任何时刻被回收的。

所以ES6中提供的weakMap的键名所引用的对象都是弱引用（注意只是键名是弱引用），不计入垃圾回收机制。也就是说，只要该引用对象引用的其他引用被清除。weakMap里的键名和键值就会自动消失。

比如下面这个例子，如果想往dom元素添加数据，又不想干扰垃圾回收机制，就可以使用weakMap。当dom元素清除后，对应的weakMap就会消失

```
const wm = new WeakMap();
const element = document.getElementById('example');
wm.set(element, 'some information');
wm.get(element) // "some information"
```

3. 无法拷贝一些特殊的对象，RegExp,Date,Map,Set。Function无法拷贝

```
function Obj() {
    this.func = function() {
        alert(1)
    } //消失
    this.obj = {
        a: 1
    }
    this.arr = [1, 2, 4]
    this.und = undefined
    this.reg = /123/ //空对象
    this.date = new Date(0) //字符串
    this.NaN = NaN //null
    this.infinity = Infinity //null
```

```
    this.sym = Symbol(1) //消失
}
let a = new Obj()
let b = JSON.stringify(a)
console.log(b);
```

2.js手写原生方法

1.js常见的继承方式

```
//原型链继承
function F1() {
    this.name = 'f1'
    this.arr = [1, 3, 4]
}

function F2() {
    this.type = 'f2'
}
F2.prototype = new F1()
console.log(new F2());
// 对于引用类型的值还是存在一个存储空间中

// 构造函数继承
function A1() {
    this.name = 'f1'
    this.arr = [12, 3]
}
A1.prototype.getName = function () {
    return this.name
}

function A2() {
    A1.call(this)
    this.type = 'a2'
}
// console.log(new A2().getName()); //undefined

//组合继承
function B1() {
    this.name = 'f1'
    this.arr = [12, 3]
    this.obj = {
        a: 1,
        b: 2
    }
}
B1.prototype.getName = function () {
    return this.name
}

function B2() {
    B1.call(this)
    this.type = 'b2'
}

B2.prototype = new B1() //又进行了一次B1的调用，相当于增加了一次性能开销
console.log(new B2());
B2.prototype.constructor = B2
var bb1 = new B1()
var bb2 = new B2()
bb2.obj.a = 3
console.log(bb1);
console.log(bb2);

// 原型式继承也可以实现浅拷贝
let p4 = {
    name: 'lian',
    arr: [1, 2, 3, 4],
```

```
    getName: function () {
        return this.name
    }
}
let p41 = Object.create(p4)
p41.name = 'li'
// 对于引用类型的值还是存在一个存储空间中

// 寄生继承：使用原型式继承获得目标对象的浅拷贝，给目标对象添加一些方法，优势时在父类基础上添加了一些新的方法
let c1 = {
    name: 'c1',
    arr: [1, 3, 4],
    getName: function () {
        return this.name
    }
}

function C2(originObj) {
    let clone = Object.create(originObj)
    clone.sayHi = function () {
        console.log(sayHi);
    }
    return clone
}
let c3 = new C2(c1)
console.log(c3);

// 寄生组合继承
// function jicheng(parent, child) {
//     child.prototype = Object.create(parent.prototype)
//     child.prototype.constructor = child
// }

function D1() {
    this.name = '11'
    this.arr = [1, 2, 3]
}
D1.prototype.getName = function () {
    return this.name
}

function D2() {
    D1.call(this)
    this.type = 'D2'
}

// D2上有了D1的原型方法，并且可以再次往D2上添加属性
D2.sayHi = function () {
    console.log('hi');
}
let D3 = new D2()
let D4 = new D2()
D3.arr.push(4)
console.log(D3, '3');
console.log(D4, '4');

class Person {
    constructor(name) {
        this.name = name
    }
    // 原型方法
    // 即 Person.prototype.getName = function() { }
    // 下面可以简写为 getName() {...}
    getName = function () {
        console.log('Person:', this.name)
    }
}
class Gamer extends Person {
```

```
    constructor(name, age) {
        // 子类中存在构造函数，则需要在使用“this”之前首先调用 super()。
        super(name) //super子类的属性 //调用super时会调用子类的Constructor函数
        this.age = age
        this.arr = [1, 2, 3]
    }
}
const asuna = new Gamer('Asuna', 20)
console.log(asuna);
const asuna1 = new Gamer('Asuna', 20)
asuna.getName() // 成功访问到

class E1 {
    constructor(name) {
        this.name = name
    }
    getName = function () {
        return this.name
    }
}

class E2 extends E1 {
    constructor(name, age, arr) {
        super(name)
        this.age = age
        this.arr = arr
    }
}
let E3 = new E2('aaa', 12, [1, 2, 3])
let E4 = new E2('aaa', 12, [1, 2, 3])
E3.arr.push(4)
console.log(E3);
console.log(E4);
```

2.jsonstringify

```
function jsonStringify(data) {
    let type = typeof data;
    if (type !== 'object') {
        let result = data
            // 这里用来处理基础类型
        if (Number.isNaN(data) || data === Infinity) {
            // NaN和infinity返回null
            result = "null"
        } else if (type === 'undefined' || type === 'function' || type === 'symbol') {
            // 对于函数，undefined，symbol类型 返回undefined
            result = undefined
        } else if (type === 'string') {
            result = '"' + data + '"'
        }
        return String(result)
    } else if (type === 'object') {
        // 用来处理引用类型
        if (data === null) { //注意null类型,并且这里处理的不是值类型，是数据
            return "null"
        } else if (data.toJSON && typeof data.toJSON === 'function') { //处理日期类型
            return jsonStringify(data.toJSON()) //注意toJSON的写法
        } else if (data instanceof Array) {
            // 判断数组，并且数组里面的每一项又可能是多样的
            let result = []
            data.forEach((item, index) => {
                // 再次判断undefined之类的类型
                if (typeof item === 'undefined' || typeof item === 'function' || typeof item === 'symbol') {
                    result[index] = 'null'
                } else {
                    // 可能为引用或基本类型
                    result[index] = jsonStringify(item)
                }
            })
```

```
            result = "[" + result + "]" // result + ""后。两面的[]会祛除
            return result.replace(/'/g, '"')
        } else {
            let result = []
                // 处理引用类型
            Object.keys(data).forEach((key, index) => {
                if (typeof key !== 'symbol') { //注意这里判断的是键值
                    // key为symbol忽略
                    if (data[key] !== undefined && typeof data[key] !== 'function' && typeof data[key] !== 'symbol') {
                        // 第一章所讲，键值为undefined，symbol，function为忽略值
                        result.push('"' + key + '"' + ":" + jsonStringify(data[key]))
                    }
                }
            })
            return ("{" + result + "}").replace(/'/g, '"')
        }
    }
}
let nl = null;
let obj = {
    name: 'jack',
    age: 18,
    attr: ['coding', 123],
    date: new Date(),
    uni: Symbol(2),
    sayHi: function() {
        console.log("hi")
    },
    info: {
        sister: 'lily',
        age: 16,
        intro: {
            money: undefined,
            job: null
        }
    }
}
```

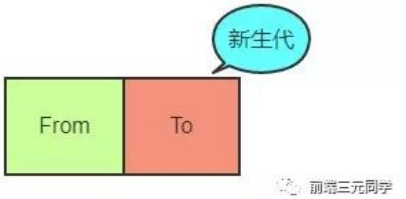
3.数组原生方法

6.js面试题进阶（下）

1 V8引擎如何进行内存的回收

新生代内存回收指临时分配的内存，存活时间短。老生代内存回收指常驻内存，存活时间长。V8的堆内存就是两个内存之和

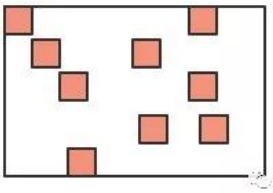
新生代内存是如何进行回收的（scavenge算法）



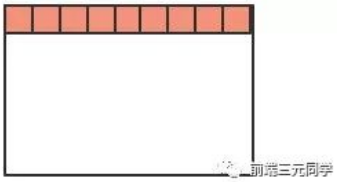
把内存空间一分为二，From代表使用中的内存。To代表闲置的内存

当进行内存回收时，会被左边中的内存检查一遍，如果是存活对象，则复制到右边（按照顺序从头放置）。如果是非存活对象，则直接回收。之后角色对调，From被闲置，To为正在使用，以此循环。

那可能有个问题，为何不将存活对象直接回收，注意，在To内存中是按照顺序从头放置的，是为了应对以下的场景



深色代表存活对象，白色区域代表闲置的内存。由于堆内存是连续分配的，这样零散散的空间如果内存稍微大一点的对象进来，可能会造成内存空间不足从而无法进行空间分配。所以上面的scavenge算法将内存碎片进行整顿，To空间就变成了下面这个样子



方便了许多，这样就可以后续连续空间的分配

不过scavenge算法也有劣势，就是内存只能使用新生代内存的一半，存放存活时间短的对象，这种对象比较上，因此时间性能较好

老生代内存是如何进行回收的

在经过新生代内存回收后，仍然存在新生代中的存活对象会被放入老生代内存中，这种对象叫晋升

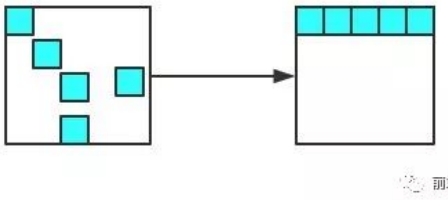
发生晋升不只是这一种原因，以下情况存在才会晋升：

- 经过一次scavenge算法回收
- To中内存超过25%的

现在进入老生代的内存回收当中，老生代累计的内存空间是很大的，当然不能用scavenge算法了，浪费一半空间并且对庞大的空间复制就是劳民伤财

对于老生代的回收策略是怎样的呢？

1. 采用标记阶段和清除阶段，首先遍历堆中所有对象，分别打上标记，对于强引用和代码环境中使用的变量取消标记，剩下的就是删除的变量了，在随后进行清除阶段
2. 此时又会产生内存碎片问题，那么老生代处理的方式比较粗暴，在清除阶段之后，把存活的对象全部往一段靠拢



增量标记

由于js是单线程并且老生代的内存空间很大，如果直接进行垃圾回收，务必会造成业务逻辑增量的进行，那么增量标记把一大块标记的任务分为一小段来进行，进行一小段，处理一些js代码，就这样依次进行。直到标记阶段完成才会进入内存碎片的整理上面

2.Promise凭借什么解决了回调地狱 2021/06/28 12:16

问题

首先，回调地狱指：

- 多层嵌套的问题
- 每种任务的处理结果有两种可能性（成功和失败），那么在每种任务执行后需要分别处理两种可能性

解决方法

Promise利用了三大手段来解决

- 回调函数延时绑定

- 返回值穿透
- 错误冒泡

```
let readFilePromise = (filename) => {
  fs.readFile(filename, (err, data) => {
    if(err) {
      reject(err);
    }else {
      resolve(data);
    }
  })
}
readFilePromise('1.json').then(data => {
  return readFilePromise('2.json')
});
```

回调函数不是立即声明，而是通过then方法传入的，即延时传入，这就是手段1

然后做出微调

```
let x = readFilePromise('1.json').then(data => {
  return readFilePromise('2.json')
});
x.then(/省略/)
```

可以把返回的Promise穿透的外层来使用，然后依次完成链式调用

多层嵌套有个问题，就是每次任务执行后错误是怎么捕获的呢？

```
readFilePromise('1.json').then(data => {
    return readFilePromise('2.json');
}).then(data => {
    return readFilePromise('3.json');
}).then(data => {
    return readFilePromise('4.json');
}).catch(err => {
  // xxx
})
```

可以看到，Promise采用错误一站式处理，错误会向后传递，被catch接受到，就不用频繁的检查了。

3.Promise为什么要引入微任务

Promise的执行函数是同步执行的，但是里面却存在的异步函数，例如resolve函数内部源码，这两者都是作为微任务进入到EventLoop,但是Promise为什么要引入微任务来进行回调操作？

解决方式

- 同步回调，直到异步代码处理完成，再进行后面的任务
- 异步回调，将回调放在宏任务队列的队尾
- 异步回调，将回调放在当前宏任务的后面

第一种显然不行，采用同步方式，造成当前任务等待，后面任务也无法执行。另外无法实现延迟绑定效果
第二种，宏任务队列非常长，等待任务完成的时间非常长，那么回调迟迟无法得到执行

所以引入了第三种，将回调放在当前宏任务的后面，这样引入微任务解决了两大痛点

- **异步回调代替同步回调解决了浪费CPU性能问题**
- 将回调放在**当前宏任务**后面执行，解决了**实时性问题**

4.谈谈对生成器以及协程的理解

执行生成器会返回一个Iterator对象，所以我们先说说Iterator对象

1.Iterator（遍历器）的概念

JavaScript原有表示“集合”的数据结构，主要是数组（Array）和对象（object），ES6又添加了Map和Set，这样有了四种数据结构。用户还可以组合他们，比如数组的成员是Map，Map的成员是对象。这样就需要一个统一的接口机制来处理不同的数据结构

遍历器（Iterator）就是这样一种机制，为不同的数据结构提供统一的访问接口，任何数据结构只要部署Iterator接口，就可以完成遍历操作，则依次处理该数据结构的所有成员

遍历器（Iterator）的作用主要有3个

1. 为各种数据结构提供统一的访问接口
2. 使数据结构的成员可以按某种次序排列
3. 主要供ES6语法的for ..of 使用

遍历器（Iterator）的遍历过程是这样的

1. 创建一个指针对象，指向当前数据结构的起始位置。也就是说，遍历器本身是个指针对象
2. 第二次调用指针对象的next（）方法，指向数据结构的第一个成员
3. 不断调用next（）方法，直至指向数据结构的结束位置

对于遍历器对象来说，done: false和value: undefined属性都是可以省略的，因此上面的makeIterator函数可以简写成下面的形式。

```
var a = makeIterator(['a', 'b'])
console.log(a.next());//{value: "a"}
console.log(a.next());//{value: "b"}
console.log(a.next().value);

function makeIterator(arr) {
    var index = 0
    return {
        next: function () {
            return index < arr.length ? { value: arr[index++] } : { done: true }
        }
    }
}
```

由于遍历器（Iterator）是加载不同数据结构上的一个接口，因此它并不依赖于数据结构。也可以写出没有对应数据结构的例子。下面是一个无限循环的例子

```
var a = makeIterator(['a', 'b'])
console.log(a.next());//{value: "a"}
console.log(a.next());//{value: "b"}


function makeIterator(arr) {
    var index = 0
    return {
        next: function () {
            return { value: arr[index++], done: false }
        }
    }
}
```

2.默认Iterator接口

Iterator的目的就是为所有的数据结构提供了统一的访问机制，即for...of循环。当for...of循环时，会默认找Iterator接口。

一种数据结构如果部署了Iterator接口，我们称这种数据结构是可遍历的

ES6规定，默认的Iterator接口部署在数据结构的Symbol.iterator属性。换句话说，也就是说数据结构只要部署了Symbol.iterator，就可以认为是‘可遍历的’。Symbol.iterator本身是个函数。返回Symbol对象的Iterator属性，是一个预定义好的，类型Symbol的特

殊值

```
const obj = {
   [Symbol.iterator]:function(){
      return {
         next:function(){
            return {
               value:1,
               done:true
            }
         }
      }
   }
}
```

原生具备 Iterator 接口的数据结构如下。

- Array
- Map
- Set
- String
- TypedArray
- 函数的 arguments 对象
- NodeList 对象

下面的例子是数组的Symbol.iterator属性。

```
let arr = ['a', 'b', 'c'];
let iter = arr[Symbol.iterator]();

iter.next() // { value: 'a', done: false }
iter.next() // { value: 'b', done: false }
iter.next() // { value: 'c', done: false }
iter.next() // { value: undefined, done: true }
```

通过遍历器实现指针结构

```
function Obj(value) {
   this.value = value
   this.next = null
}
Obj.prototype[Symbol.iterator] = function () {
   var iterator = { next: next }
   var that = this
   function next() {
      if (that) {
         var value = that.value
         that = that.next
         return {
            done: false,
            value: value
         }
      }
      return {
         done: true
      }
   }

   return iterator
}
var one = new Obj(1);
var two = new Obj(2);
var three = new Obj(3);

one.next = two;
two.next = three;
console.log(one);
```

```
for (var i of one) {
    console.log(i); //1,2,3
}
```

调用Symbol.iterator方法，调用next方法，在返回一个值的同时，自动将指针移到下一个实例

3.调用Iterator接口的场合

有些场合会默认调用Iterator接口

1.解构赋值

对数组和Set结构进行解构赋值时，会调用Iterator方法

```
var a = new Set().add(1).add(2).add(3)
var [b,c] = a
// b = 1; c =2
```

2.扩展运算符

```
var a = 'hello'
[...a] // ["h", "e", "l", "l", "o"]
```

3.yield*

yield*后面是可遍历的结构，它会调用该结构的遍历器接口

```
let g = function*(){
    yield 1;
    yield [2,3];
    yield 4
}
let iterator = g()
iterator.next() //{ value: 1, done: false }
iterator.next() //{ value: 2, done: false }
iterator.next() //{ value: 3, done: false }
iterator.next() //{ value: 4, done: false }
iterator.next() //{ value: undefined, done: true}
```

4.字符串的Iterator接口

可以覆盖原生的Symbol.iterator方法，达到修改遍历器行为的目的。

```
var a = new String('hello')
console.log([...a]); //["h", "e", "l", "l", "o"]
a[Symbol.iterator] = function () {
    return {
        next: function () {
            if (this.first) {
                this.first = false
                return {
                    value: 'bye',
                    done: false
                }
            } else {
                return {
                    value: 'undefined',
                    done: true
                }
            }
        },
        first: true
    }
}
console.log([...a]); //['bye']
//a: hello
```

5.Generator函数

基本概念

Generator函数是ES6提供的一种异步编程解决方案，它的异步编程方案看《6.Generator函数的异步编程方案》

Generator函数是一个状态机，封装了多个内部状态。

执行Generator函数会返回一个遍历器对象，也是一个遍历器生成对象函数，它可以依次遍历Generator内部的多个状态

```
function* helloWorldGenerator() {
  yield 'hello';
  yield 'world';
  return 'ending';
}
var hw = helloWorldGenerator();
```

它内部有两个表达式hello，world，即该函数有三种状态，hello、world、return语句。

调用Generator函数后，该函数并不执行，返回的也不是函数运行结果，而是一个执行内部状态的指针对象，也就是《Iterator对象》。

下一步，调用next方法，指针就会从函数头部或上次停下来的地方开始执行，直到遇见yield表达式或return语句。换言之，yield表达式用来暂停，next用来恢复执行。

yield表达式

遍历器对象的next方法运行逻辑如下

1. 遇见yield表达式，就暂停后面的操作，并将紧跟yield表达式后面的值返回。
2. 继续调用next，直到遇到下一个yield表达式。
3. 如果没有遇到yield表达式，那么将return返回的值作为返回对象的value值。
4. 如果没有return语句，则返回对象的value值为undefined。

需要注意的是，yield表达式后面的值，只有调用next方法，内部指针指向该语句时才会执行。

```
function* gen() {
  yield  123 + 456;
}
```

上面代码，yield表达式后面的值，不会立即求值，，当调用next指向该语句时才会求值。

yield与return语句的区别之处：

yield可以进行返回值，并且下次执行时可以从当前位置继续向后执行。而return语句不具备记忆的功能。一个函数中只能执行一个return语句。但是可以多次执行yield。Generator函数返回一系列的值，因此可以有多个yield。所以就叫它“生成器（Generator）”。

注意点

Generator函数可以不用yield表达式，这时就变成了单纯的暂缓执行函数。

```
function* f() {
  console.log('执行了！')
}
var generator = f();
setTimeout(function () {
  generator.next()
}, 2000);
```

函数f如果是普通函数，会立即执行。但是Generator函数在调用next方法才会执行。

与Iterator接口的关系

Generator函数执行后，返回一个遍历器对象。该对象也就有Symbol.Iterator属性，执行后返回自身

```
function* gen(){

}
var iterator = gen()
//gen[Symbol.iterator] === g true
```

next方法的参数

yield表达式本身没有返回值，或者说总是返回undefined，next方法可以带一个参数，该参数会被当做上一个yield表达式的返回值。

```
function* f() {
  for(var i = 0; true; i++) {
    var reset = yield i;
    if(reset) { i = -1; }
  }
}

var g = f();

g.next() // { value: 0, done: false }
g.next() // { value: 1, done: false }
g.next(true) // { value: 0, done: false }
```

上述是一个无限运行的Generator函数f，当next方法未传参数时，变量reset总是undefined。当next方法传入参数时，变量reset就会被重置为这个参数（即true），因此i=-1，下一轮循环从-1递增。

这个功能有很重要的意义。通过next方法的参数，就有办法在Generator函数开始运行之后，继续像函数体内部注入值。也就是说，可以在Generator函数的任何阶段，从外部往内部注入不同值，从而调整函数行为。

看一个例子

```
function* foo(x) {
    var y = 3 * (yield (x + 1))
    var z = yield (y / 2)
    return x + y + z
}
var gen = foo(2)
console.log(gen.next());//{value: 3, done: false}
console.log(gen.next(4));//{value: 6, done: false}
console.log(gen.next(6));//{value: 20, done: true}
```

第一次执行next方法，变量y为undefined，第二次执行next方法，把变量4作为上次yield表达式的值，即3*4，变量y为12。第三次执行next方法，把变量6作为第二个yield表达式返回的值，变量z为6。最终遇到return语句，变量x为2。2+12+6 = 20

注意：由于next方法传递参数表示上一个yield表达式返回的值。所以第一次使用next方法传递参数时无效的。V8引擎直接忽略第一次使用next方法的参数。只有从第二次使用next方法传递的参数才有效

如果想要第一次调用next方法，yield表达式就能够返回值给变量，那么必须再Generator函数外再包一层

```
function wrapper(wrapperFunc) {
    return function (...args) {
        let wrapperFuncObj = wrapperFunc(...args)
        wrapperFuncObj.next()
        return wrapperFuncObj
    }
}
var wrappered = wrapper(function* () {
    console.log(`first input:${yield}`);
    return "DONE"
})
wrappered().next("Hello")
```

上面Generator函数如果不包一层，是无法第一层调用next方法传入参数的。

yield*表达式

yield表达式如果在Generator函数前在加了星号，等同于在Generator函数内部部署了一个for...of循环。

```
function* concat(iter1, iter2) {
  yield* iter1;
  yield* iter2;
}

// 等同于

function* concat(iter1, iter2) {
  for (var value of iter1) {
    yield value;
  }
  for (var value of iter2) {
    yield value;
  }
}
```

来看个例子

```
function* inner() {
  yield 'hello!';
}

function* outer1() {
  yield 'open';
  yield inner();
  yield 'close';
}

var gen = outer1()
gen.next().value // "open"
gen.next().value // 返回一个遍历器对象
gen.next().value // "close"

function* outer2() {
  yield 'open'
  yield* inner()
  yield 'close'
}

var gen = outer2()
gen.next().value // "open"
gen.next().value // "hello!"
gen.next().value // "close"function* foo(){
    yield 1
    yield 2
}
function* goo(){
    yield
}
```

outer1中的inner，在yield表达式后没有加星号，则返回的是一个遍历器对象。outer2中的inner在加星号的前提下，返回的就是遍历器对象的内布值。

实际上，只要任何数据结构部署了Iterator遍历器，就可以被yield*遍历。比如数组、字符串等。

在有return语句时，需要用var value = yield* foo()获取return语句返回的值

```
function* foo(){
    yield 2
    yield 3
    return 'hello'
}
function* bar(){
    yield 1
    var value = yield* foo()
    console.log(value)
```

```
}
//{value: 1, done: false}
// {value: 2, done: false}
// {value: 3, done: false}
// hello
// {value: undefined, done: true}
```

使用yield命令可以取出嵌套数组的所有成员

```
function* getArray(arr) {
    if (Array.isArray(arr)) {
        for (let i = 0; i < arr.length; i++) {
            yield* getArray(arr[i])
        }
    } else {
        yield arr
    }
}
for (let i of getArray([1, 2, [3, 4]])) {
    console.log(i);
    //1,2,3,4
}
```

协程

传统的编程语言，早有异步编程的解决方案。意思是多个线程互相协作，完成异步任务。

协程有点像函数，又有点像协程。运行流程如下：

- 第一步，协程A开始执行。
- 协程A执行到一半，进入暂停，执行权转移到协程B。
- （一段时间后）协程B交还执行权。
- 协程A恢复执行。

举例来说，读取文件的协程如下：

```
function* asyncJob() {
  var a = yield readFile(fileA);
  var b = yield readFile(fileB);
}
function otherJob(){
    //...
}
var job = asyncJob()
job.next() //任务完成-fileA
otherJob()//去做了些别的事情
job.next()//任务完成-fileB
```

可以看到，协程A（asyncJob）执行了fileA任务，执行了一半后进入了暂停，把执行权交给了协程B（otherJob），协程B在一段时间后又把执行权交给了协程A（asyncJob），然后去执行了fileB任务。

它的奥妙就在于其中的yield命令。它表示执行到此处，执行权交给其他协程。也就是说，yield命令是异步两个阶段的分界线。

协程遇到yield命令就暂停，等到执行权返回，再从暂停的地方继续向后执行。最大特点就是代码的写法很像同步操作。

Thunkify模块源码

使用方式如下

```
var thunkify = require('thunkify');
var fs = require('fs');

var read = thunkify(fs.readFile);
read('package.json')(function(err, str){
```

```
  // ...
});
```

Thunkify 的源码

```
function thunkify(fn) {
  return function() {
    var args = new Array(arguments.length);
    var ctx = this;

    for (var i = 0; i < args.length; ++i) {
      args[i] = arguments[i];
    }

    return function (done) {
      var called;
        //检查机制
      args.push(function () {
        if (called) return;
        called = true;
        done.apply(null, arguments);
      });

      try {
        fn.apply(ctx, args); //
      } catch (err) {
        done(err);
      }
    }
  }
};
```

主要多了一个检查机制，变量`called`确保回调函数只运行一次，主要与下文的Generator函数相关。请看下面的例子。

```
function f(a, b, callback){
  var sum = a + b;
  callback(sum);
  callback(sum);
}

var ft = thunkify(f);
var print = console.log.bind(console);
ft(1, 2)(print);
// 3
```

上面代码中，由于`thunkify`只允许回调函数执行一次，所以只输出一行结果。

```
function* gen(){
    //...
}
var g = gen()
var res = g.next()

while(!res.done){
    console.log(res.value)
    res = g.next()
}
```

Thunk函数

```
let isArray = (obj)=>{
    return Object.prototype.toString.call(obj) === '[object Array]'
}
```

可以看到，上述代码是一个判断Array类型的函数。但是例如string，object类型等也需要重新定义一个函数来判断，那这样就出现了非常多的逻辑。

这就是我们介绍的Thunk版本。

```
let isType = (type)=>{
    return (obj)=>{
        return Object.prototype.toString.call(obj) === `[object ${type}]`
    }
}
let isArray = isType('Array')
isArray([])
```

这样看来，代码简洁了不少，生产出定制化的函数。

Thunk函数的核心逻辑就是接受一定的参数，使用定制化的函数去完成功能。方便了后续的操作。

Thunk函数可以自动执行Generator函数，看接下来的例子。

Generator函数的流程管理

ES6有了Generator函数，Thunk函数现在可以用于Generator函数的自动流程管理。

Generator函数可以自动执行，next方法就作为Thunk函数来使用。

```
function* gen(){

}
var g = gen()
var res = g.next()
while(!res.done){
    console.log(res.value)
    res = g.next()
}
```

显然，这不适合异步操作。如果必须保证前一步执行完才能执行后一步，上面的自动执行就不可行。

这是Thunk函数就可以派上用场。以读取文件为例，下面的Generator函数封装了两个异步操作。

```
var fs = require('fs');
var thunkify = require('thunkify');
var readFileThunk = thunkify(fs.readFile);

var gen = function* (){
  var r1 = yield readFileThunk('/etc/fstab');
  console.log(r1.toString());
  var r2 = yield readFileThunk('/etc/shells');
  console.log(r2.toString());
};
```

yield命令将程序的执行权从Generator函数移出。那么需要一种方法，将执行权再交回Generator函数。

这种方法就是Thunk函数，其实就是通过读取文件成功后的回调函数的方式。便于理解，以下是手动执行上面这个Generator函数。

```
var g = gen()
var res = g.next()
res.value(function(err,data){
    if(err)throw err
    //交出执行权（做些别的事）
    var res2 = g.next(data) //交回执行权
    res2.value(function(err,data){
        if(err)throw err
        g.next(data)
    })
})
```

变量g是函数的内部指针，表示目前执行到哪一步，next方法将指针移动到下一步，并且返回该步的信息（value和done属性）。

Thunk函数的自动流程管理

```
function run(fn){
    var gen = fn()
    function next(err,data){
```

```
        var res = gen.next(data)
        if(res.done)return
        res.value(next)
    }
    next()
}
function* g(){
    //..
}
run(g)
```

内部的next函数就是Thunk的回调函数，next函数先将指针移动到下一步，然后判断Generator函数是否结束（res.done）。如果没结束，就将next函数传入Thunk函数中，Thunk的回调函数用来执行next函数（ done.apply(null, arguments) ）。

当然，前提是每一个异步操作，都要是 Thunk 函数，也就是说，跟在yield命令后面的必须是 Thunk 函数。

```
var g = function* (){
  var f1 = yield readFileThunk('fileA');
  var f2 = yield readFileThunk('fileB');
  // ...
  var fn = yield readFileThunk('fileN');
};

run(g);
```

根据以上代码用到的readFileThunk，下面是它的源码

```
function thunkify(fn) {
  return function() {
    var args = new Array(arguments.length);
    var ctx = this;

    for (var i = 0; i < args.length; ++i) {
      args[i] = arguments[i];
    }

    return function (done) { //指针对象的value属性返回一个函数，done属性就是传来的回调函数
      var called;
        //检查机制
      args.push(function () {
        if (called) return;
        called = true;
        done.apply(null, arguments);
      });

      try {
        fn.apply(ctx, args); //
      } catch (err) {
        done(err);
      }
    }
  }
};
```

基于Promise的自动执行

```
function readFile(fileName) {
    return new Promise(function (resolve, reject) {
        return fs.readFile(fileName, function (err, data) {
            if (err) reject(err)
            resolve(data)
        })
    })
}
var gen = function* () {
    var f1 = yield readFile('./19.text.html');
    var f2 = yield readFile('./19.text.html');
```

```
    // console.log(f1.toString());
    // console.log(f2.toString());
};
function run(gen) {
    var g = gen()
    function next(data) {
        var res = g.next(data)
        if (res.done) return res.value
        res.value.then(function (data) {
            next(data)
        })
    }
    next()
}
run(gen);
```

采用co库执行

5.谈谈async函数与await的理解

async函数使得异步操作变得更方便。

async函数就是Generator函数的语法糖。

前文有个Generator函数，依次读取两个文件。

```
const gen = function* () {
  const f1 = yield readFile('/etc/fstab');
  const f2 = yield readFile('/etc/shells');
  console.log(f1.toString());
  console.log(f2.toString());
};
```

上面的函数gen可以写成async函数，例如下面这样。

```
const asyncReadFile = async function () {
  const f1 = await readFile('/etc/fstab');
  const f2 = await readFile('/etc/shells');
  console.log(f1.toString());
  console.log(f2.toString());
};
```

比较就会发现，Generator函数的星号*换成了async，将yield换成了await，仅此而已。

async函数对Generator函数作的改进，体现在下面4点。

1. async函数内置执行器，Generator函数必须调用next方法。

```
asyncReadFile()
```

与普通函数的调用方式一样，只需要一行。

2. async和await，比起星号和yield，语义更加清晰，async函数表示函数内部有异步操作。await表示紧跟在后面的表达式需等待结果。
3. co模块定义，yield后面只能跟Thunk函数或promise对象，而await后面除了前者，还可以是原始类型的值（字符串等，但这时会自动转成resolved的Promise对象）。
4. 返回值是Promise，可以使用then方法指定下一步的操作。

语法

async函数的语法规则比较简单，难点是错误处理机制。

返回Promise对象

async函数内部的return语句返回的值，会成为then方法回调函数的参数。

```
async function foo()
   return '123'
}
foo().then(function(data){
   console.log(data)//123
})
```

async函数内部抛出错误，会导致Promise对象变为reject状态。会被catch方法回调接收到。

```
foo().then(data=>console.log(data),err=>console.log(err))
```

Promise对象状态的变化

async函数返回的Promise对象，必须等内部所有的await后面跟的表达式执行完成之后，才会返回状态改变（resolved/rejected）。也就是说，等async函数内部所有的异步操作执行完成之后，才可以用then方法指定的回调参数。

```
async function getTitle(url) {
  let response = await fetch(url);
  let html = await response.text();
  return html.match(/<title>([\s\S]+)<\/title>/i)[1];
}
getTitle('https://tc39.github.io/ecma262/').then(console.log)
// "ECMAScript 2017 Language Specification"
```

上面代码中，函数getTitle内部有三个操作：抓取网页、取出文本、匹配页面标题。只有这三个操作全部完成，才会执行then方法里面的console.log。

await后面的Promise对象变为rejected状态，会被catch函数接收到。

```
async foo(){
   await Promise.reject('error')
}
```

注意，上面代码没有return语句，reject方法依然可以将参数传入回调函数。

只有await后面跟的Promise对象变为了rejected状态，则会中断整个async函数的执行。

```
async function f() {
  await Promise.reject('出错了');
  await Promise.resolve('hello world'); // 不会执行
}
```

有时我们希望前一个语句失败，但并不会影响下一个语句的执行。那么可以将await放在try--catch语句中。

```
async function f(){
   try{
      await Promise.reject('出错了');
   }catch(e){

   }
   return Promise.resolve('hello world')
}
f().then(r=>...)
```

另一种方法是await后面再跟一个catch语句来捕获错误。

```
async function f(){
   await Promise.reject('出错了').catch((e)=>...)
   return Promise.resolve('hello world')
}
f().then(v)
console.log(v)
//出错了
//hello world
```

使用注意点

第一点，await命令后面的Promise对象，可能是rejected状态，所以最好把await放在try-catch语句中。

```
async myJob(){
    try{
        await doSomething()
    }catch(e){
        //处理错误
    }
}
```

第二点，多个await命令后面的异步操作，如果不存在继发关系。最好让它们同时触发。

```
let a = await doJob1()
let b = await doJob2()
```

上面的doJob1和doJob2是两个独立的异步操作（即互不依赖），被写成继发关系，比较耗时，因为当doJob1处理完才处理doJob2，那么我们可以改为并发的形式。

```
//写法1
Promise.all([doJob1(),doJob2()])
//写法2
let a= doJob1()
let b= doJob2()
let c = await a
let d = await b
```

getFoo和getBar都是同时触发，这样就会缩短程序的执行时间。