技术

[1 标题1 错误!未定义书签。](#_Toc449540698)

[1.1 标题2 **错误!未定义书签。**](#_Toc449540699)

[1.1.1 标题3 **错误!未定义书签。**](#_Toc449540700)

# 应用场景

* 前端（美工，页面设计），决定页面的样式，将设计使用HTML实现。
* **前端工程师，**在浏览器端运行的JS
* **前端（JS）全栈工程师，**在浏览器端和服务器端运行的JS。Node全栈，JS大全栈，大前端。
* 页面设计（用户体验设计，用户接口设计），不写代码，偏向于产品经理。设计原型图。

# 语言解析（语言虚拟机）

语言是否可以跨平台？

语言 与 运行环境（语言虚拟机）是分开的。

语言：语法结构管理，流程控制，模块化，面向对象。script.js, run.go

运行环境：编译语言源码，运行编译结果，提供基础功能模块。

Nodejs。就是服务器端js运行虚拟机。

服务器端，任何一个操作系统，都可以是服务器端。提供服务的才叫服务器端，在BS环境中，能够被请求提供响应的，就是服务器端。

# 面向对象（基于对象）

语法是以对象的为核心的，是面向对象语言。

基于对象，针对于JS不是传统的class型面向对象语言的称呼。{}

## 对象类型

对象，object，一种数据类型。类似于Go中的Struct。属性（和方法）的集合。在JS中方法（函数），是以数据类型而不是以语法结构的形式出现。js中对象的方法就是某个类型为函数的属性。

|  |
| --- |
| **let** user1 = {  *// 属性property*  name: '',  *// 方法method，本质上是数据类型为函数的属性*  setName: **function**(newName) {  **this**.name = newName  }  } |

如图所示：



go语言中的结构体对象的方法，是通过在函数上增加接收器来实现的：

|  |
| --- |
| struct User {  属性列表  name string  }  func (this \* User) SetName(newName string) {  this.name = newName  } |

## 得到对象

对象的本质是一致的，只是js提供了多种得到对象的方案！

### {}，字面量语法，JSON语法。常用语法

得到具体某个对象，得到一个书店对象。

|  |
| --- |
| **let** user1 = {  *// 属性property*  name: '',  *// 方法method，本质上是数据类型为函数的属性*  setName: **function**(newName) {  **this**.name = newName  }  } |

如果需要user2，还要再使用上面的语法去得到user2。因此不适合得到同类型的多个对象。

{}称之为字面量语法，literal语法，原生写法，直接写值的语法。

new Object(), {}

new Array(), []

new String(), ‘’

new Number(), 42

{}，对象的字面量语法，有一个专业的称呼：JSON，JavaScript Object Notation，JS对象表示法。

延伸的领域，在BS做数据交换时，也会采用JSON语法，指的是数据格式，而不是JS对象。

|  |
| --- |
| {  "name": "Kang",  "age": 42,  } |

### 构造器constructor，核心语法

得到某一类对象，得到十本属性不同的书，适合构造器语法。

JS中得到对象的本质，核心方法。

是一个函数，一般来说，函数内会存在this，用于操作构造的对象，使用new调用构造器。非常类似于其他语言中的class。

构造器就是函数，new去调用，依据其构造器的用法，不用new去调用，就是普通函数的用法。普通函数中，this不能引用用户的对象，通常会引用全局对象（浏览器中就是window，nodjs中就是global）。

反过来说，调用方式（是否new），影响的就是函数中this的值。

this的值，代替函数的调用对象。

new User() 相当于 User.call(new Object())

|  |
| --- |
| **function** User(newName) {  **this**.name = newName  console.log(**this**, 'User run')  }  **let** user2 = **new** User('Kang')  console.log(user2, **typeof**(user2))  *// 相当于*  **let** user2 = {}  User.call(user2, 'Kang')  console.log(user2, **typeof**(user2)) |

适合一个类型的对象有多个：

|  |
| --- |
| **let** user2 = **new** User('Kang')  console.log(user2, **typeof**(user2))  **let** user3 = **new** User('zhong')  console.log(user3)  **let** user4 = **new** User('han')  console.log(user4) |

一旦是用了构造器得到对象，方法的定义需要考虑：不能使用this，在每个对象上增加相应的方法，因为这会导致，方法可执行性代码，在每个对象上都存在独立的一份，出现了代码的重用问题。

因此需要将方法，从对象本身提取出来，放在一个可以被对象调用到的地方。

使用 prototype实现，原型链。



语法：

|  |
| --- |
| **function** User(newName) {  **this**.name = newName  *// this.setName = function(newName) {*  *// this.name = newName*  *// }*  }  User.prototype.setName = **function**(newName) {  **this**.name = newName  }  **let** user4 = **new** User('jinyong')  console.log(user4)  user4.setName('gulong')  console.log(user4) |

实例化的对象，会将构造器的prototype属性作为其上层原型，在当前对象上不存在的属性，在其原型上查找，直到找到最顶层的原型对象。null = Object.prototype

上面代码逻辑图是：



### class, 构造器的封装，兼容语法

与constructor是一致的，表面的语法不同。

向世界看齐。人言可畏。

利用class，封装了constructor：

|  |
| --- |
| *// function User(newName) {*  *// this.name = newName*  *// }*  *// User.prototype.setName = function(newName) {*  *// this.name = newName*  *// }*  **class** User {  constructor(newName) {  **this**.name = newName  }  setName(newName) {  **this**.name = newName  }  }  **let** user6 = **new** User('Kang')  console.log(user6)  user6.setName('zhong')  console.log(user6) |

### 复制（克隆）

基于已有对象得到新对象。

对象为引用类型，对象间的赋值为引用拷贝，不能得到新对象。

如果需要依据已有对象，得到新对象，则需要实现克隆操作。

本质是：实例化新对象，将就对象的属性复制到新对象上，就是克隆。

演示：

|  |
| --- |
| Object.clone = **function** (old) {  **let** o = {}  **for** (**var** p **in** old) {  o[p] = old[p]  }  **return** o  }  user2 = Object.clone(user1)  user2.name = 'zhong'  console.log(user1, user2) |

user1, user2 两个属性一致的对象。

深浅克隆，如果user1的某个属性为引用类型，那么以上的克隆：

|  |
| --- |
| user1 = {  name: 'kang',  age: 42,  **class**: {  title: 'blockChain'  }  }  user2 = Object.clone(user1)  user2.**class**.title = 'BC'  console.log(user1, user2) |

克隆之后，导致user1.class 与 user2.class 指向相同的对象。修user2的class，user1的class也会改。

相对的深拷贝（深度克隆），就是将引用类型的属性，也要做一份拷贝出来。

思路：判断如果属性为对象，数组，则继续遍历属性拷贝即可。

|  |
| --- |
| Object.clone = **function** (old) {  **let** o = {}  **for** (**var** p **in** old) {  o[p] = old[p]  }  **return** o  }  *// user2 = Object.clone(user1)*  *// user2.class.title = 'BC'*  *// console.log(user1, user2)*  Object.deepClone = **function**(old) {  *// 先确定克隆的是对象还是数组*  *// 初始化不同的数据*  **let** type = Object.prototype.toString.call(old).slice(8, -1)  **if** ('Array' == type) {  **var** newData = []  } **else** **if** ('Object' == type) {  **var** newData = {}  } **else** {  **return** old  }  *// 遍历*  **for** (**let** p **in** old) {  **let** ptype = Object.prototype.toString.call(old[p]).slice(8, -1)  *// 如果是 对象或者数组类型，则深度拷贝，递归调用*  **if** ('Array' == type || 'Object' == type) {  newData[p] = Object.deepClone(old[p])  } **else** {  newData[p] = old[p]  }  }  **return** newData  }  user1 = {  name: 'kang',  age: 42,  **class**: {  title: 'blockChain',  room: {  address: '302'  }  }  }  user2 = Object.deepClone(user1)  user2.**class**.title = 'BC'  user2.**class**.room.address = '409'  console.log(user1, user2) |

## this

this，引用函数的调用对象。

两种可能性：

1. 调用对象, this = obj。obj = new func()，obj.func(), func.call(obj), func.apply(obj)
2. 全局对象（window，global）。函数中使用了this，但是并没有指定其调用对象，相当于global.func()。this = window, this=global.

Object()是数据类型，是对象型数据的顶层构造器。

global 是具体的某个对象。处于作用域链的最顶端！

global = new Object()

global = {}

this是在调用时确定的，而不是在语法（词法）层面确定的。

相对的作用域，就是词法层面确定的。

## =>,bind()

=> 胖箭头

目的：

1. 简化函数定义
2. 简化this的传递。核心目的。将this的确定与词法保持一致。

还可以使用.bind()来绑定函数中的this：一下演示三种方案：

演示：

|  |
| --- |
| **function** User(newName) {  **this**.name = newName  **this**.sayName0 = **function**() {  console.log(**this**.name)  **var** innerFunc = **function**() {  console.log(**this**.name)  }  innerFunc()  }  **this**.sayName1 = **function**() {  console.log(**this**.name)  **var** innerFunc = **function** () {  console.log(**this**.name)  }.bind(**this**)  innerFunc()  }  **this**.sayName2 = **function**() {  console.log(**this**.name)  **var** innerFunc = () => {  console.log(**this**.name)  }  innerFunc()  }  }  user1 = **new** User('Kang')  user1.sayName0()  user1.sayName1()  user1.sayName2() |

# 数据类型（标准库）

## Map，映射

映射。键值对集合。

Js的对象不就是键值对集合么？是，对象就是键值对集合 {name: “kang”, age: 42}

为什么还要在标准库增加一个Map型的键值对集合呢？

对象型的数据，仅（有一点点松动）支持字符串型键。

Map,支持任意类型的键。

|  |
| --- |
| **let** o = {  name: 'kang',  42: 'age', *// 42转换为'42'*  **false**: **false**,  {} : {},*// 错误的*  }  console.log(o)  **let** m = **new** Map()  m.set('name', 'kang')  m.set(42, 42)  m.set(**false**, **false**)  m.set({}, {})  console.log(m) |

map.set(key, value)

map.delete(key)

map.get(key)

map.has(key)

map.clear()

## Set，集合

元素的集合，特征：

* 唯一，
* 无序。

唯一的元素，仅仅表示元素值即可

|  |
| --- |
| **let** s = **new** Set()  s.add('kang')  s.add('kang')  s.add('zong')  s.add('kgafad')  console.log(s)  console.log(s.has('kang'))  console.log(s.has('wa'))  s.**delete**('zong')  console.log(s)  s.clear()  console.log(s) |

set.add(element)

set.delete(element)

set.has(element)

set.clear()

## WeakMap，弱引用映射

## WeakSet，弱引用集合

Map和Set的本意是不变的。

Weak：弱，指的是 weak reference 弱引用。

弱引用：

针对于引用类型来说的。常规引用某个值时，JS底层运行环境（JS虚拟机）会记录某个值的引用次数。记录引用次数是为了回收垃圾，当一个数据的引用次数为0时，该数据就是垃圾，所占用的内存空间被回收。（周期性自动执行，高级语言的典型特征，自动垃圾回收）

弱引用变量，指的是，不会去累加值的引用次数的变量。

将一个对象变量，放入到若映射或者弱集合中，变量的引用次数不会增加。如果外部没有变量再引用某个对象，即使该对象放入到了WeakSet或者WeakMap中，也会被垃圾回收。

语法注意：仅支持引用类型（Object， Array， Function）

|  |
| --- |
| 打开node控制台，--expose-gc 选择手动垃圾回收  $ node --expose-gc  > global.gc()，执行垃圾回收  >process.memoryUsage()，当前内存使用量 |



# 作用域，闭包

要点：

词法作用域，源代码决定了变量的作用域。相对于this理解，this是在函数被调用时，是运行时。

作用域层叠。内层作用域，可以访问外层作用域。（与Go是类似的）。

var：在当前函数作用域声明变量。var （function也是如此）声明前置，变量（函数）提升。

let：在当前语句块（{}, if, for）作用域声明变量

作用域链：作用域层叠典型的实现。内层作用域与外层作用域相连。最顶层就是全局对象（window，或者 global）。

## 闭包

语法现象。

在：

作用域层叠

外部引用函数，函数可以作为值被赋值（支持匿名函数）

|  |
| --- |
| **var** f;  **function** func() {  **var** v = 42 *// func的局部变量*  f = **function**() {  ++v *// 作用域层叠的，内可以访问外层*  console.log(v)  }  }  func()*// 运行结束后，其作用域的变量应该随之释放*  f() *// 由于其内部函数，还可以被调用，同时内部函数使用外部函数作用域的变量。*  f() |

在理解一下，作用域链，由每个作用域对象组成的：

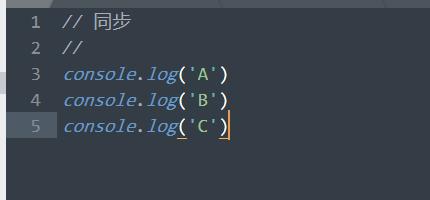


# 异步

promise，generator，async-await

## 同步，异步

同步，sync，同（不是同时），指的是同一个流程，线程。排队执行，只有一个队列，一个个的解决执行，叫同步。代码的执行处于阻塞状态，



异步,async，与同步对应。前面的语句未执行完毕时，后边的程序开始执行了。



## 为啥需要异步执行？

有些程序（功能），是非CPU密集型的，在执行时，CPU会处于闲置状态，等待状态。通过异步执行，让CPU持续工作，在等待期也工作，提升CPU的利用率。

典型的指令：

计时器，IO处理，IO网络磁盘输入输出（读写）。

settimetout,setinterval，fs.readFile, ajax.send

是以上的指令才使得程序异步执行的。

## 异步之后结果的处理？

在异步执行后，需要对异步的结果进行处理。例如：处理ajax的响应结果，处理文件读取内容。

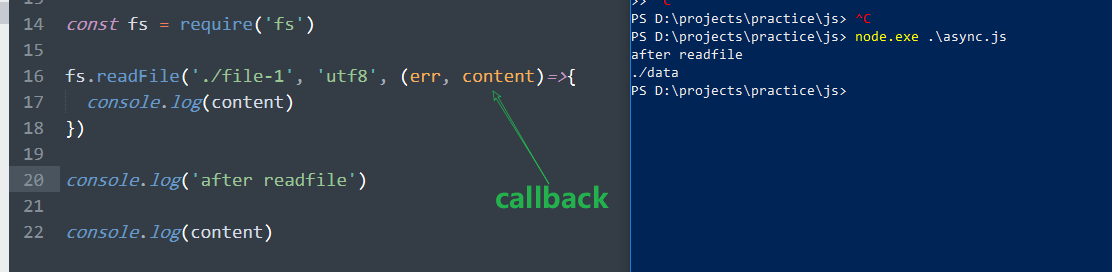
处理异步结果的代码应该在核实执行？

典型的方案：

callback, promise，async

### callback

fs.readFile 异步的文件读取程序，就采用的callback的解决方案！



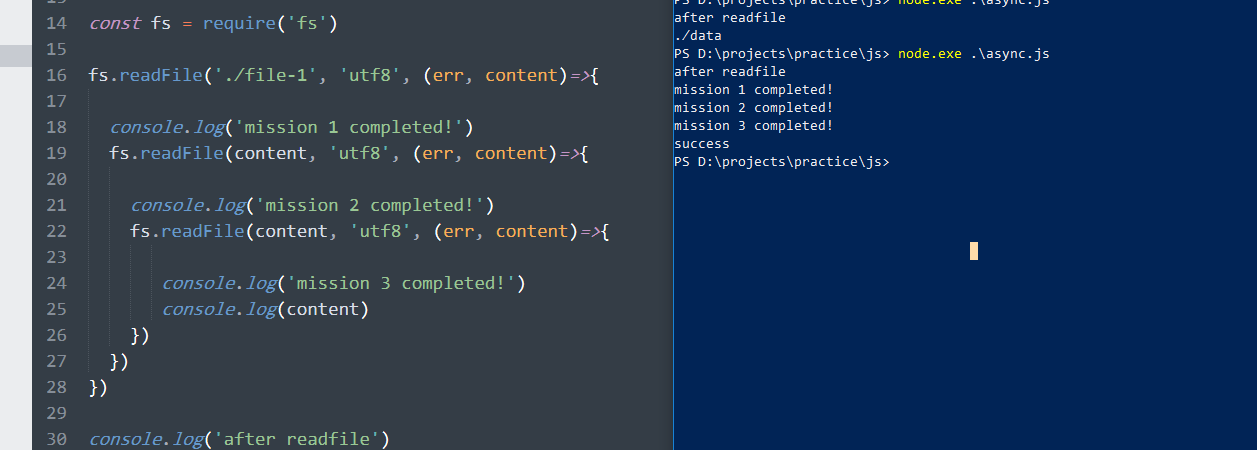
callback，向异步程序，传递函数作为参数，当异步指向完毕，自动调用该函数。

（函数可以作为参数，称之为高阶函数）

### 回调地狱

如果多个回调函数嵌套执行，语法不好理解。代理：

* 作用域层叠
* this的解析
* 错误的处理



### Promise

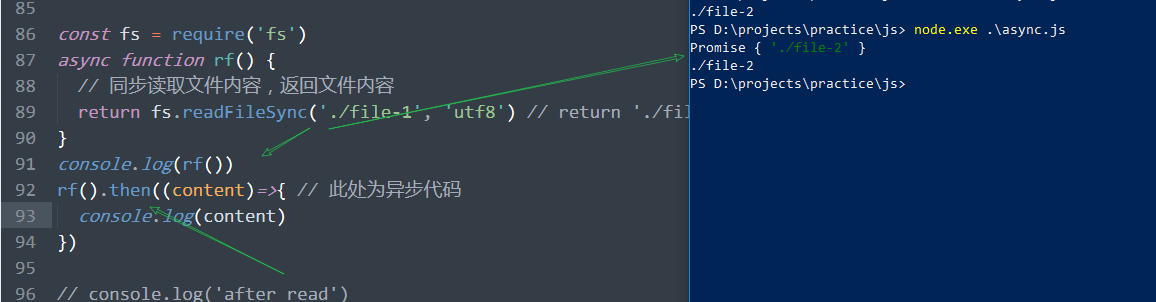
将嵌套的回调地狱，扁平化。使用同步语法写异步代码。

以上三个文件的连续异步，使用Promise解决：

|  |
| --- |
| **const** fs = require('fs')  **let** p = **new** Promise((resolve, reject)=>{  fs.readFile('./file-1', 'utf8', (err, content)=>{  console.log(content)  resolve(content)  })  })  p  .then((content) =>{  **return** **new** Promise((resolve, reject) => {  fs.readFile(content, 'utf8', (err, content)=>{  console.log(content)  resolve(content)  })  })  })  .then((content) =>{  **return** **new** Promise((resolve, reject) => {  fs.readFile(content, 'utf8', (err, content)=>{  resolve(content)  })  })  })  .then((content) => {  console.log(content)  }) |

### async

async就是封装了Promise的操作。



### await 在async函数内使用

当需要连续指向异步操作时，可以让后续异步操作等待前面的异步操作指向完毕后，再执行：

|  |
| --- |
| **const** fs = require('fs')  async **function** rf() {  *// 同步读取文件内容，返回文件内容*  **let** c1 = await **new** Promise((resolve, reject)=>{  fs.readFile('./file-1', 'utf8', (e, c) => {  resolve(c)  })  })  **let** c2 = await **new** Promise((resolve, reject)=>{  fs.readFile(c1, 'utf8', (e, c) => {  resolve(c)  })  })  **let** c3 = await **new** Promise((resolve, reject)=>{  fs.readFile(c2, 'utf8', (e, c) => {  resolve(c)  })  })  console.log(c3)  } |

也可以简单点：

|  |
| --- |
| **const** fs = require('fs')  async **function** rf() {  *// 同步读取文件内容，返回文件内容*  **let** c1 = await fs.readFileSync('./file-1', 'utf8')  **let** c2 = await fs.readFileSync(c1, 'utf8')  **let** c3 = await fs.readFileSync(c2, 'utf8')  console.log(c3)  }  rf()  console.log('after run') |

下面的方法是否也可以?

|  |
| --- |
| **function** rf1() {  fs.readFile('./file-1', 'utf8', (err, c1)=>{  read2(c1)  })  **function** read2(path) {  fs.readFile(path, 'utf8', (err, c2)=>{  read3(c2)  })  }  **function** read3(path) {  fs.readFile(path, 'utf8', (err, content)=>{  console.log(content)  })  }  }  rf1()  console.log('after run') |

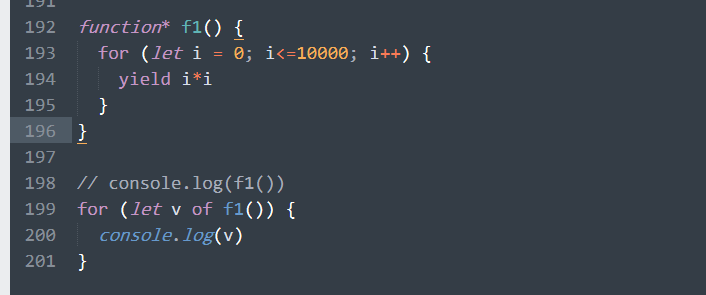
### generator

function\*

|  |
| --- |
| **const** fs = require('fs')  **function**\* rf3() {  **var** c1 = **yield** fs.readFile('./file-1', 'utf8', (e, c) => {  rf.next(c)  })  **var** c2 = **yield** fs.readFile(c1, 'utf8', (e, c) => {  rf.next(c)  })  **yield** fs.readFile(c2, 'utf8', (e, c) => {  console.log(c)  })  }  **var** rf = rf3()  rf.next()  console.log('after run') |

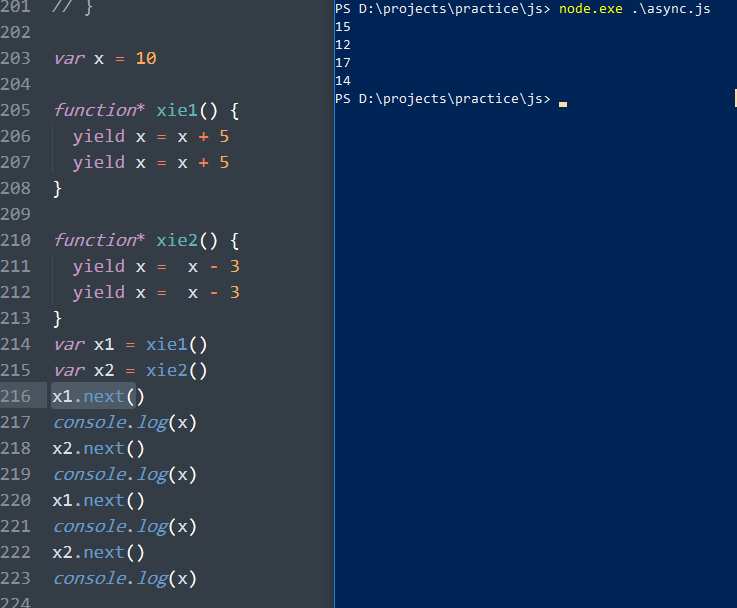
generator返回的是Iterator，迭代器。

携程迭代的。



iteratable，可迭代，数组，set，map都是可迭代的。

yield 协程（协助进程）语法。函数没运行完，可以先运行其他代码，返回来再运行函数。

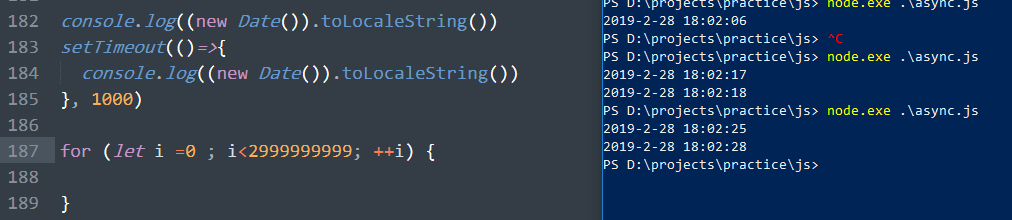


generator：迭代，协程，异步！

### JS单线程

同步代码在主线程执行。当出现异步代码时，在其他非主线程并发执行。

JS单线程，当主线程的代码没有执行完毕时，即使其他异步任务完了，也不会抢占主线程执行。主线程任务是独立的，不可分割。



当异步异步任务执行完毕后，将异步任务相关的回调函数，加入EventQueue队列中。JS运行环境，会实时监控队列，一旦读取到任务，放入主线程执行，需要等待主线程当前任务结束。机制称之为EventLoop，事件循环。