

Node-前端模块化

作者：小余同学

为什么没有01，那是因为01是讲谷歌V8引擎的原理，我看完了，但忘记写笔记了，不想重新看一遍写笔记了hh，大致来说也是很重要的一章节

需要全系列笔记请到[2002XiaoYu\(小余\)\(github.com\)](https://github.com/2002XiaoYu)中自行获取，觉得不错给个star，这是对作者非常大的鼓励

(理解)邂逅Node和Node的架构

Node.js是什么

- 官方对Node.js的定义：

- Node.js是一个基于V8 JavaScript引擎的JavaScript运行时环境。

Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine.

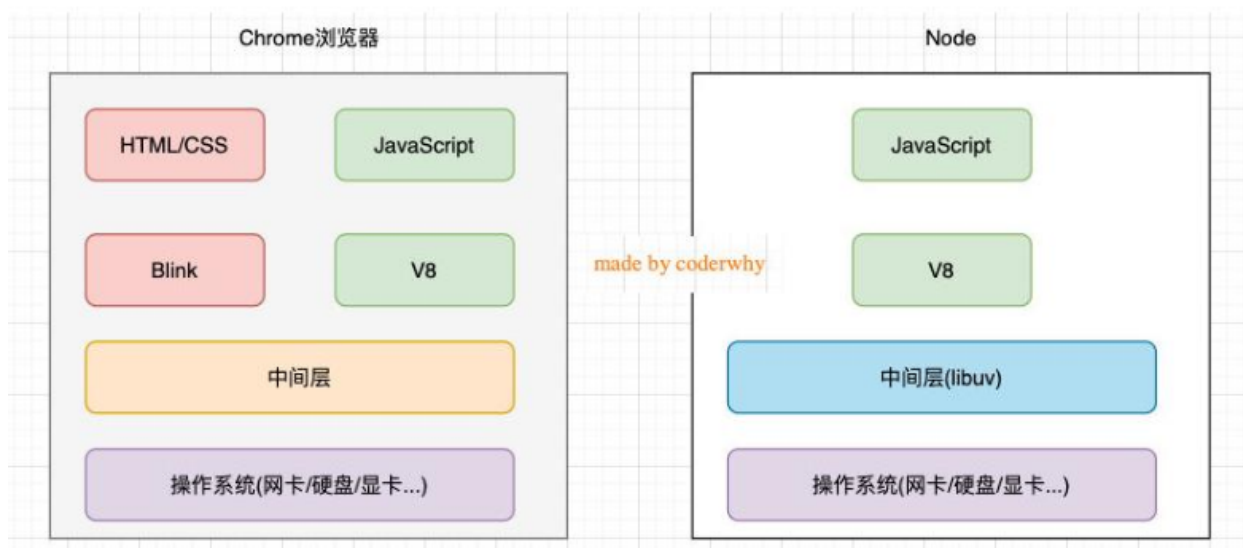
- 也就是说Node.js基于V8引擎来执行JavaScript的代码，但是不仅仅只有V8引擎：

- 前面我们知道V8可以嵌入到任何C++应用程序中，无论是Chrome还是Node.js，事实上都是嵌入了V8引擎来执行JavaScript代码
- 但是在Chrome浏览器中，还需要解析、渲染HTML、CSS等相关渲染引擎，另外还需要提供支持浏览器操作的API、浏览器自己的事件循环等
- 另外，在Node.js中我们也需要进行一些额外的操作，比如文件系统读/写、网络IO、加密、压缩解压文件等操作

浏览器和Node.js架构区别

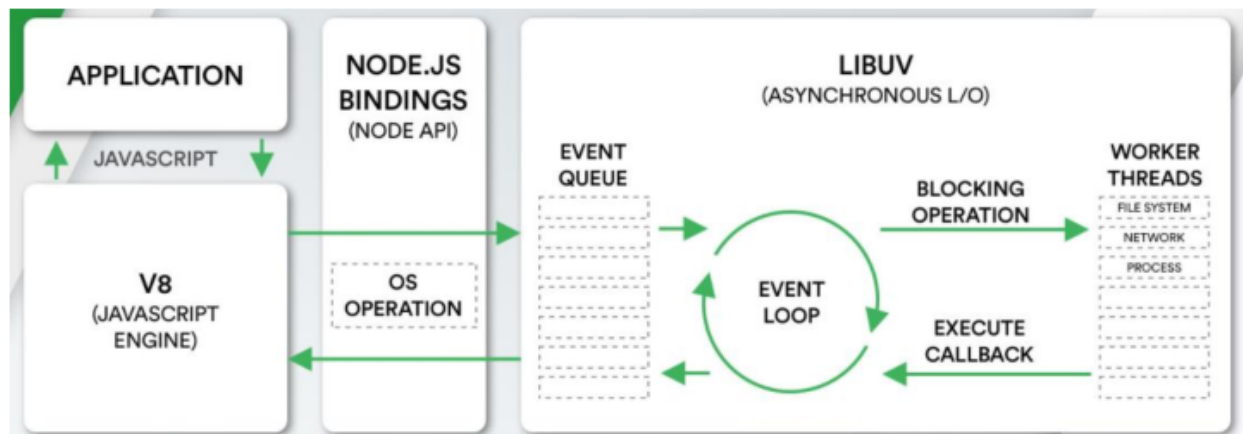
- 我们可以简单理解规划出Node.js和浏览器的差异：

- Node是用JS/C++/C语言编写的，调用API部分是JS，V8引擎部分是C++，中间层是C语言，关键的中间层(库)用来对底层操作系统进行调用



Node.js架构

- 我们可以简单理解规划出Node.js和浏览器的差异：
 - 我们编写的JavaScript代码会经过V8引擎，再通过Node.js的Bindings，将任务放到Libuv的事件循环中
 - libuv（Unicorn Velociraptor—独角伶盗龙）是使用C语言编写的库
 - libuv提供了事件循环、文件系统读写、网络IO、线程池等等内容



在APPLICATION中是自己的应用程序，用JS语言编写，然后经过V8引擎，对代码进行翻译，因为后面的中间层是看不懂JS的，然后通过Node.js进行了中转，调用了Node.js的API，然后经过了LIBUV，被这个中间层转化成操作系统能够听懂的语言

Node.js架构涉及名词	翻译
FILE SYSTEM	文件系统
APPLICATION	应用程序
V8(JAVASCRIPTENGINE)	V8引擎
NODE.JS BINGINGS(NODE API)	node.js绑定(通过Node的API)

Node.js架构涉及名词	翻译
OS OPERATION	操作 操作系统
EVENT QUEUE	事件队列
ASYNCHRONOUS L/O	异步 L/O
EVENT LOOP	事件循环
BLOCKING OPERATION	阻塞操作
EXECUTE CALLBACK	执行回调
WORKER THREADS	工作线程
FILE SYSTEM	文件系统
NETWORK	网络
PROCESS	过程

(理解)Node的应用场景

- Node.js的快速发展也让企业对Node.js技术越来越重视，在前端招聘中通常会对Node.js有一定的要求，特别对于高级前端开发工程师，Node.js更是必不可少的技能：
- 应用一：目前前端开发的库都是以node包的形式进行管理
- 应用二：npm、yarn、pnpm工具成为前端开发使用最多的工具
- 应用三：越来越多的公司使用Node.js作为web服务器开发、中间件、代理服务器
- 应用四：大量项目需要借助Node.js完成前后端渲染的同构应用
- 应用五：资深前端工程师需要为项目编写脚本工具（前端工程师编写脚本通常会使用JavaScript，而不是Python或者shell）
- 应用六：很多企业在Electron来开发桌面应用程序

(掌握)Node的安装和基本使用

Node的安装

- Node.js是在2009年诞生的，目前最新的版本是分别是LTS 16.15.1以及Current 18.4.0：
 - LTS版本：（Long-term support, 长期支持）相对稳定一些，推荐线上环境使用该版本
 - Current版本：最新的Node版本，包含很多新特性

Download for Windows (x64)

16.15.1 LTS

Recommended For Most Users

18.4.0 Current

Latest Features

- 这些我们选择什么版本呢？

- 如果你是学习使用，可以选择current版本
- 如果你是**公司开发**，建议选择**LTS版本**（面向工作，选择LTS版本）
- **Node的安装方式有很多：**
 - 可以借助于一些**操作系统上的软件管理工具**，比如Mac上的homebrew，Linux上的yum、dnf等；
 - 也可以**直接下载对应的安装包下载安装**；
- **我们选择下载安装，下载自己操作系统的安装包直接安装就可以了：**
 - window选择**.msi安装包**(Microsoft install的缩写)，Mac选择.pkg安装包，Linux会在后续部署中讲解
 - 安装过程中**会配置环境变量**（让我们可以在命令行使用）
 - 并且会**安装npm（Node Package Manager）工具**

(了解)Node多版本管理工具-nvm和n

Node的版本工具

- 在实际开发学习中，我们只需要使用一个Node版本来开发或者学习即可。
- 但是，如果你希望通过可以快速更新或切换多个版本时，可以借助于一些工具：
 - **nvm**：Node Version Manager；(节点版本管理器)
 - **n**：Interactively Manage Your Node.js Versions（交互式管理你的Node.js版本）
- **问题：这两个工具都不支持window**
 - n：n is not supported natively on Windows.
 - nvm：nvm does not support Windows
- **Window的同学怎么办？**
 - 针对nvm，在GitHub上有提供对应的window版本：<https://github.com/coreybutler/nvm-windows>
 - 通过 **nvm install latest** 安装最新的node版本
 - 通过 **nvm list** 展示目前安装的所有版本
 - 通过 **nvm use** 切换版本(使用管理员身份运行)

```
C:\Users\XiaoYu>nvm version
1.1.9

C:\Users\XiaoYu>|
```

版本管理工具：n

- **安装n：**直接使用npm安装即可(只能在Mac电脑)

```
//安装工具n
npm install n -g
//查看安装的版本
n --version
```

- **安装最新的lts版本：**
 - 前面添加的**sudo**是权限问题
 - 可以两个版本都安装，之后我们可以通过n快速在两个版本间切换；

```
//安装最新的lts版本
n lts
//安装最新的版本
n latest

//查看所有的版本
n
```

(掌握)VSCode中终端的使用过程

JavaScript代码执行

- 如果我们编写一个js文件，里面存放JavaScript代码，如何来执行它呢？
- 目前我们知道有两种方式可以执行：
 - 将代码交给**浏览器**执行；
 - 将代码载入到**node**环境中执行
- 如果我们希望把代码交给浏览器执行：
 - 需要通过让浏览器加载、解析html代码，所以我们需要创建一个html文件；
 - 在html中通过script标签，引入js文件
 - 当浏览器遇到script标签时，就会根据src加载、执行JavaScript代码
- 如果我们希望把js文件交给node执行：
 - 首先电脑上需要**安装Node.js**环境，安装过程中会**自动配置环境变量**
 - 可以通过**终端命令node js文件**的方式来载入和执行对应的js文件
- 如果我们希望把js文件交给node执行：
 - 首先电脑上需要**安装Node.js**环境，安装过程中会**自动配置环境变量**
 - 可以通过**终端命令node js文件**的方式来载入和执行对应的js文件

```
console.log("小余在呢")
```

```
问题 1 输出 调试控制台 终端
D:\Desktop\Project\h5css3>node testFile.js
小余在呢

D:\Desktop\Project\h5css3>
```

(掌握)Node程序中的输入和输出

Node的REPL

- 什么是REPL呢？感觉挺高大上
 - REPL是Read-Eval-Print Loop的简称，翻译为“读取-求值-输出”循环
 - REPL是一个简单的、交互式的编程环境
- 事实上，我们浏览器的console就可以看成一个REPL
- Node也给我们提供了一个REPL环境，我们可以在其中演练简单的代码

直接在编辑器终端输入node，进入REPL环境
退出REPL环境就输入.exit

```
D:\Desktop\Project\h5css3>node
Welcome to Node.js v18.12.1.
Type ".help" for more information.
> const name = "余"
undefined
> console.log(name)
余
undefined
> .exit

D:\Desktop\Project\h5css3>
```

Node程序传递参数

- 正常情况下执行一个node程序，直接跟上我们对应的文件即可：

```
node index.js
```

- 但是，在某些情况下执行node程序的过程中，我们可能希望给node传递一些参数：

```
node testFile.js env=development XiaoYu
```

- 如果我们这样来使用程序，就意味着我们需要在程序中获取到传递的参数：

- 获取参数其实是在**process的内置对象**中的；
- 如果我们**直接打印这个内置对象**，它里面包含特别的信息：
 - ✓ 其他的一些信息，比如版本、操作系统等大家可以自行查看，后面用到一些其他的我们还会提到
- 现在，我们先找到其中的**argv属性**：
 - 我们发现它是一个数组，里面包含了我们需要的参数

```
console.log(process.argv) //从进程中的程序获取
```



```
D:\Desktop\Project\h5css3>node testFile.js
[
  'D:\\Program Files\\nodejs\\node.exe',
  'D:\\Desktop\\Project\\h5css3\\testFile.js'
]
```

node所在目录
文件所在目录

```
D:\Desktop\Project\h5css3>node testFile.js env=development XiaoYu
[
  'D:\\Program Files\\nodejs\\node.exe',
  'D:\\Desktop\\Project\\h5css3\\testFile.js',
  'env=development',
  'XiaoYu'
]
```

为什么叫argv呢？

- 你可能有个疑问，为什么叫argv呢？
- 在C/C++程序中的main函数中，实际上可以获取到两个参数：
 - **argc**: argument counter的缩写，传递参数的个数
 - **argv**: argument vector（向量、矢量）的缩写，传入的具体参数
 - ✓ vector翻译过来是矢量的意思，在程序中表示的是一种数据结构
 - ✓ 在C++、Java中都有这种数据结构，是一种数组结构
 - ✓ 在JavaScript中也是一个数组，里面存储一些参数信息
 - 我们可以在代码中，将这些参数信息遍历出来，使用：

```
console.log(process.argv);
```

```
process.argv.forEach(item => {  
  console.log(item);  
});
```

JS testFile.js > process.argv.forEach() callback

```
1 console.log(process.argv);  
2  
3 process.argv.forEach(item => {  
4   console.log(item);  
5 });
```

问题 1 输出 调试控制台 终端

```
D:\Desktop\Project\h5css3>node testFile.js env=development XiaoYu  
[  
  'D:\\Program Files\\nodejs\\node.exe',  
  'D:\\Desktop\\Project\\h5css3\\testFile.js',  
  'env=development',  
  'XiaoYu'  
]  
D:\\Program Files\\nodejs\\node.exe  
D:\\Desktop\\Project\\h5css3\\testFile.js  
env=development  
XiaoYu  
  
D:\\Desktop\\Project\\h5css3>
```

//单独获取，然后将获取到的内容传递到某个地方

```
console.log(process.argv[3]);
```

//命令

```
node testFile.js env=development XiaoYu
```

JS testFile.js

```
1 console.log(process.argv[3]);
```

问题 1 输出 调试控制台 终端

```
D:\Desktop\Project\h5css3>node testFile.js env=development XiaoYu  
XiaoYu
```


Node的输出

- `console.log`
 - 最常用的输入内容的方式：`console.log`
- `console.clear`
 - 清空控制台：`console.clear`
- `console.trace`
 - 打印函数的调用栈：`console.trace`
- 还有一些其他的方法，其他的一些console方法，可以自己在下面学习研究一下
 - <https://nodejs.org/dist/latest-v16.x/docs/api/console.html>

(掌握)Node中常见的全局对象

- 在node环境下打印window是会报错的，因为并没有这个东西

```
console.log(window);
```

- 想要打印出内容的话，需要打印 `global`，在window的东西在node环境下被放到了global下

```
console.log(global); //在node环境下类似在浏览器打印出来的window
```

全局对象

- Node中给我们提供了一些全局对象，方便我们进行一些操作：
 - 这些全局对象，我们并不需要从一开始全部一个个学习
 - 某些全局对象并不常用
 - 某些全局对象我们会在后续学习中讲到
 - ✓ 比如`module`、`exports`、`require()`会在模块化中讲到；
 - ✓ 比如`Buffer`后续会专门讲到

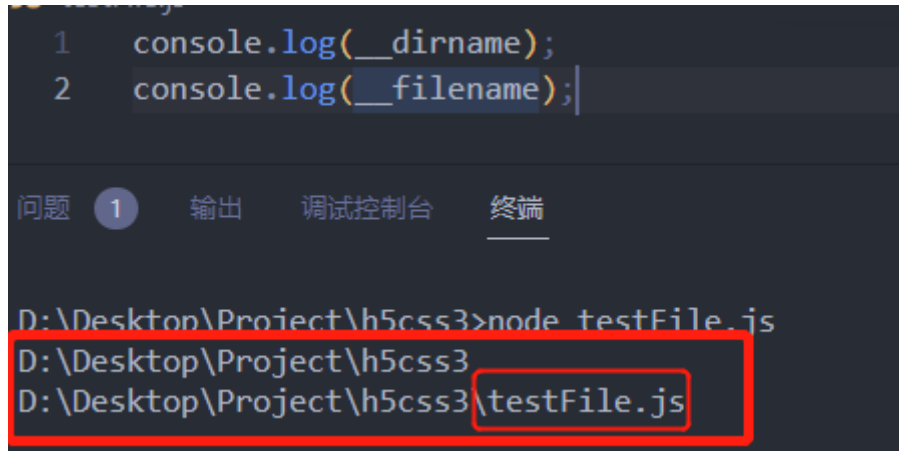
- Global objects
 - Class: Buffer
 - `__dirname`
 - `__filename`
 - `clearImmediate(immediateObject)`
 - `clearInterval(intervalObject)`
 - `clearTimeout(timeoutObject)`
 - `console`
 - `exports`
 - `global`
 - `module`
 - `process`
 - `queueMicrotask(callback)`
 - `require()`
 - `setImmediate(callback[, ...args])`
 - `setInterval(callback, delay[, ...args])`
 - `setTimeout(callback, delay[, ...args])`
 - `TextDecoder`
 - `TextEncoder`
 - `URL`
 - `URLSearchParams`
 - `WebAssembly`

特殊的全局对象

- 为什么我称之为特殊的全局对象呢？
 - 这些全局对象实际上是**模块中的变量**，只是**每个模块都有**，看来像是**全局变量**
 - 在命令行交互中是不可以使用的；
 - 包括：`__dirname`、`__filename`、`exports`、`module`、`require()`
- `__dirname`：获取**当前文件所在的路径**：(目录)
 - 注意：不包括后面的文件名(重要，后面需要用到相对路径的时候就派上用场了)
- `__filename`：获取**当前文件所在的路径和文件名称**
 - 注意：包括后面的文件名称

```
//获取当前文件所在的路径：（目录）
console.log(__dirname);
//获取当前文件所在的路径和文件名称
console.log(__filename);

//模块化时具体学习
console.log(module);
console.log(exports);
console.log(require);
```



```
1 console.log(__dirname);
2 console.log(__filename);
```

问题 1 输出 调试控制台 终端

```
D:\Desktop\Project\h5css3>node testFile.js
D:\Desktop\Project\h5css3
D:\Desktop\Project\h5css3\testFile.js
```

常见的全局对象

- **process对象**：process提供了Node进程中相关的信息：
 - 比如Node的运行环境、参数信息等；
 - 后面在项目中，我也会讲解，如何将一些环境变量读取到 process 的 env 中；
- **console对象**：提供了简单的调试控制台，在前面讲解输入内容时已经学习过了。
 - 更加详细的查看官网文档：<https://nodejs.org/api/console.html>
- **定时器函数**：在Node中使用定时器有好几种方式：
 - **setTimeout(callback, delay[, ...args])**：callback在delay毫秒后执行一次；
 - **setInterval(callback, delay[, ...args])**：callback每delay毫秒重复执行一次
 - **setImmediate(callback[, ...args])**：callback / O事件后的回调的“立即”执行
 - ✓ 这里先不展开讨论它和setTimeout(callback, 0)之间的区别
 - ✓ 因为它涉及到事件循环的阶段问题，我会在后续详细写事件循环相关的知识
 - **process.nextTick(callback[, ...args])**：添加到下一次tick队列中；
 - ✓ 具体的讲解，也放到事件循环中说明

```
setTimesetTimeout(()=>{
  console.log("3s后执行");
},3000)

setInterval(()=>{
  console.log("每隔3s执行一次");
},3000)
```

```
setImmediate(()=>{
  console.log("立即，立刻");
})//不传时间，具体执行时机后面讲
```

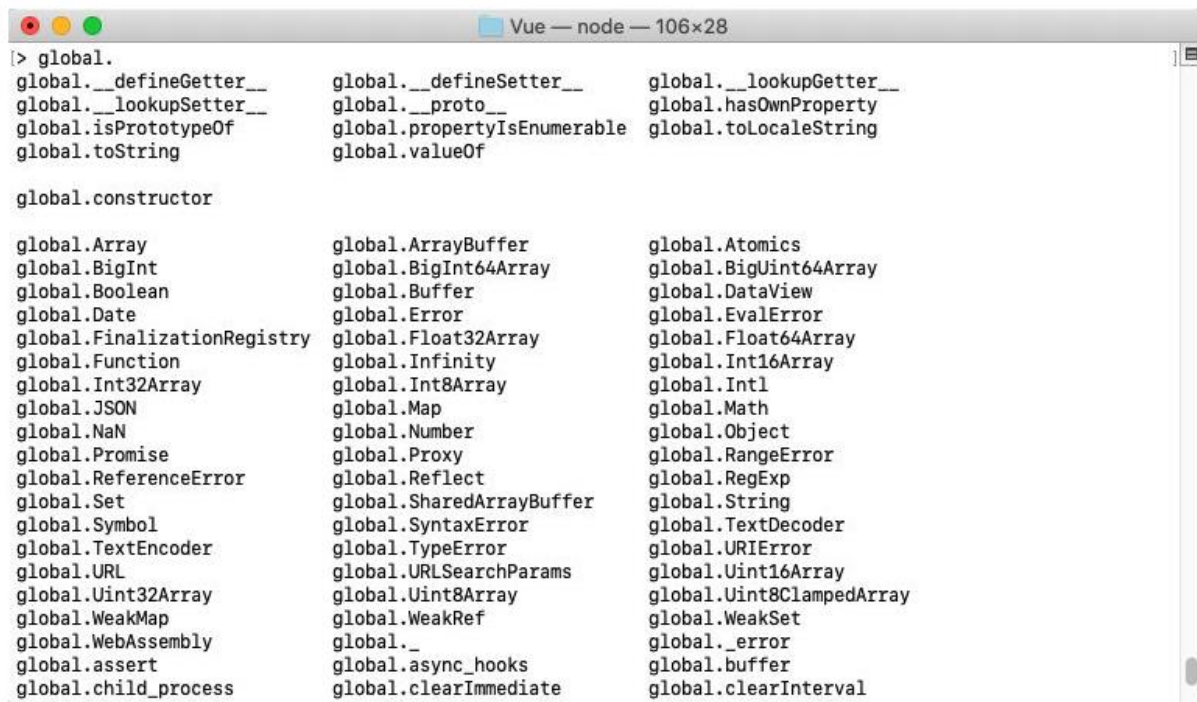
//额外执行函数

```
process.nextTick()
```

//由node来执行，而不是由浏览器来执行。用法一样，node的内部也是有v8引擎的

global对象

- **global是一个全局对象，事实上前端我们提到的process、console、setTimeout等都有被放到global中：**
 - 我们之前讲过：在新的标准中还有一个globalThis，也是指向全局对象的
 - 类似于浏览器中的window
 - 在最新规范里面，node中的叫法跟浏览器的叫法进行了统一，叫做globalThis



global和window的区别

- 在浏览器中，全局变量都是在window上的，比如有document、setInterval、setTimeout、alert、console等等
- 在Node中，我们也有一个global属性，并且看起来它里面有很多其他对象
- 但是在浏览器中执行的JavaScript代码，如果我们在顶级范围内通过var定义的一个属性，默认会被添加到window对象上：

```
var name = "小余"
console.log(window.name)//小余
```

我们如果使用var定义一个变量，在浏览器中会被加入到window中，但是在node环境中不会被加到global中

```
var name = "小余"
console.log(global.name)//undefined
```

JavaScript模块化开发

(掌握)认识模块化和模块化发展的历史

什么是模块化？

- **到底什么是模块化、模块化开发呢？**
 - 事实上模块化开发最终的目的是将程序划分成一个个小的结构；
 - 这个结构中编写属于**自己的逻辑代码**，有**自己的作用域**，定义变量名词时不会影响到其他的结构
 - 这个结构可以将自己希望暴露的**变量、函数、对象**等导出给其结构使用
 - 也可以通过某种方式，**导入**另外结构中的**变量、函数、对象**等
- **上面说提到的结构，就是模块；按照这种结构划分开发程序的过程，就是模块化开发的过程；**
- 无论你多么喜欢JavaScript，以及它现在发展的有多好，它都有很多的缺陷：
 - 比如var定义的变量作用域问题
 - 比如JavaScript的面向对象并不能像常规面向对象语言一样使用class
 - 比如JavaScript没有模块化的问题
- **对于早期的JavaScript没有模块化来说，确实实带来了很多问题**

模块化的历史

- **在网页开发的早期，Brendan Eich开发JavaScript仅仅作为一种脚本语言，做一些简单的表单验证或动画实现等，那个时候代码还是很少的：**
 - 这个时候我们只需要讲JavaScript代码写到**标签**中即可；
 - 并没有必要放到多个文件中来编写；甚至流行：通常来说 JavaScript 程序的**长度只有一行**
- **但是随着前端和JavaScript的快速发展，JavaScript代码变得越来越复杂了：**
 - ajax的出现，**前后端开发分离**，意味着后端返回数据后，我们需要通过**JavaScript进行前端页面的渲染**
 - SPA的出现，前端页面变得更加复杂：包括**前端路由、状态管理**等一系列复杂的需求需要通过JavaScript来实现
 - 包括Node的实现，JavaScript编写**复杂的后端程序**，没有模块化是致命的硬伤
- **所以，模块化已经是JavaScript一个非常迫切的需求：**
 - 但是JavaScript本身，直到**ES6（2015）才推出了自己的模块化方案；**
 - 在此之前，为了让JavaScript支持模块化，涌现出了很多不同的模块化规范：**AMD、CMD、CommonJS**等
- 下面中将详细讲解JavaScript的模块化，尤其是CommonJS和ES6的模块化

//如果我们想避免变量名冲突, 可以使用立即执行函数

```
(function(){  
  let name = "小余"  
  console.log(name)  
})();
```

//这样就算跟其他地方的命名一样, 也不会产生冲突, 核心的原因是因为函数是有作用域的, 做了一个隔离

//就像下方这样是不会冲突的, 但是如果都放到全局作用域中就会产生冲突

```
function foo(){  
  let age = 20  
}  
  
function bar(){  
  let age = 18  
}
```

- 如果我们想要这里的js文件使用其他js文件的内容, 要怎么做呢?
- 自己实现缺点:
 1. 不规范
 2. 不知道暴露出去了什么内容, 又接收了那些内容, 除非把源码看一遍, 那也太费劲了

//自己实现(有很多缺陷)

//在主HTML文件引入两个需要相关联的js文件

```
<script src="./index.js"></script>  
<script src="./testFile.js"></script>
```

//在index.js文件

```
const moduleA = (function(){  
  let name = "小余"  
  let age = 20  
  let height = 1.75  
  
  return{//我们将结果return出去  
    name, age, height  
  }  
})();//立即执行函数
```

//在testFile.js文件

```
console.log(moduleA.name);//因为index.js已经return出去了, 暴露在html主文件中了,  
testFile.js就能够直接从html主文件中获取到内容
```

//然后在浏览器控制台就能打印出内容了

- ECMAScript没有推出的方案: **CommonJS(目前依旧流行)**/AMD(淘汰)/CMD(淘汰)(由社区自己发展出来的方法)
- ES6(ES2015)推出自己的模块化方案: **ESModule**

没有模块化带来的问题

- 早期没有模块化带来了很多问题：比如命名冲突的问题
- 当然，我们有办法可以解决上面的问题：立即函数调用表达式（IIFE）
 - IIFE (Immediately Invoked Function Expression)
- 但是，我们其实带来了新的问题：
 - 第一，我必须记得**每一个模块中返回对象的命名**，才能在其他模块使用过程中正确的使用
 - 第二，代码写起来**混乱不堪**，每个文件中的代码都需要**包裹在一个匿名函数中来编写**
 - 第三，在**没有合适的规范**情况下，每个人、每个公司都可能会任意命名、甚至出现模块名称相同的情况
- 所以，我们会发现，虽然实现了模块化，但是我们的实现过于简单，并且是没有规范的
 - 我们需要制定一定的规范来约束每个人都**按照这个规范去编写模块化的代码**
 - 这个规范中应该包括核心功能：**模块本身可以导出暴露的属性，模块又可以导入自己需要的属性**
 - JavaScript社区为了解决上面的问题，涌现出一系列好用的规范，接下来我们就学习具有代表性的一些规范

(掌握)CommonJS规范和Node中使用案例

CommonJS规范和Node关系

- 我们需要知道CommonJS是一个规范，最初提出来是在浏览器以外的地方使用(例如服务器)，并且当时被命名为ServerJS，后来为了体现它的广泛性，修改为CommonJS，平时我们也会简称为CJS
 - Node是CommonJS在服务器端一个具有代表性的实现
 - Browserify是CommonJS在浏览器中的一种实现(现在不用了)
 - webpack打包工具具备对CommonJS的支持和转换
- 所以，Node中对CommonJS进行了支持和实现，让我们在开发node的过程中可以方便的进行模块化开发：
 - 在Node中**每一个js文件都是一个单独的模块**
 - 这个模块中包括CommonJS规范的核心变量：**exports、module.exports、require**
 - 我们可以使用这些变量来方便的进行模块化开发
- 前面我们提到过模块化的核心是导出和导入，Node中对其进行了实现：
 - **exports和module.exports**可以负责**对模块中的内容进行导出**
 - **require函数**可以帮助我们**导入其他模块（自定义模块、系统模块、第三方库模块）中的内容**

CommonJS规范

1. 模块中要导出内容：exports
2. 模块中要导出内容：require

模块化案例

在JS中是不能够直接使用export跟require导出导入的，平时在Vue中编写代码可以直接使用是因为webpeak做了处理

在node中，每个文件都是一个独立的模块

```
//test.js文件
const name = "xiaoyu"
const age = 20

function sayHello(name){
  console.log("Hello"+name);
}
//main.js文件
console.log(name);
console.log(age);

sayHello("ikun")
```



//node实现CommonJS规范：导入导出

//一共用到2个文件，分别是testFile.js、main.js

//testFile.js文件

```
const name = "xiaoyu"
const age = 20

function sayHello(name,age){
  console.log(`${name}今年${age}岁了`);
}
```

exports.sayHello = sayHello//导出

//main.js文件

```
const testFile = require("./testFile");//导入，会返回结果需要接收，取名testFile是为了见名知意
testFile.sayHello("小余",20)
```


- 在main.js中终端输出: node main.js

执行结果: 小余今年20岁了



```
JS main.js > ...
1 const testFile = require("../testFile") // 取名testFile是为了见名知意
2 testFile.sayHello("小余", 20)

JS testFile.js > ...
1 const name = "xiaoyu"
2 const age = 20
3
4 function sayHello(name, age) {
5   console.log(`${name}今年${age}岁了`);
6 }
7
8 exports.sayHello = sayHello
```

1. 除了可以直接获取导出的对象, 从对象中获取属性(我们上面就是这么做的)
2. 但还可以通过解构的方式去实现, 会在使用的时候更加简便

```
// 解构实现, 观察简便在哪? 只修改main.js的文件
const {sayHello} = require("../testFile") // 取名testFile是为了见名知意
sayHello("小余", 20) // 结果不变
```

(理解)CommonJS在Node中实现的本质

exports导出

- 注意: exports是一个对象, 我们可以在这个对象中添加很多个属性, 添加的属性会导出;

在刚刚上面有demo实现 (用得很少)

```
exports.name = name;
...
exports.age = age;
exports.sayHello = sayHello;
```

- 另外一个文件中可以导入:

```
const bar = require('./bar');
```

- 上面这行完成了什么操作呢? 理解下面这句话, Node中的模块化一目了然
 - 意味着main中的bar变量等于exports对象(指向同一个地方)
 - 也就是require通过各种查找方式, 最终找到了exports这个对象

- 并且将这个exports对象赋值给了bar变量
- bar变量就是exports对象了



```
//探讨本质

//testFile.js文件
let name1 = "xiaoyu"

exports.name1 = name1

setTimeout(()=>{
  exports.name1 = "超级大大余"
},2000)

//main.js文件
const Person = require("./testFile")

console.log(Person.name1);//xiaoyu

setTimeout(()=>{
  console.log(Person.name1);//超级大大余
},3000)
```

//我们进行的操作：导出了testFile.js文件中的name1内容，由main.js文件引入。首先第一时间在main.js文件打印出了name1的内容，为xiaoyu。过了2秒修改testFile文件的erports.name1的内容为"超级大大余"，再过了1秒在main.js文件中重新打印了引入的name1的内容，发现被修改为超级大大余了，证明我们main.js文件require导入的跟testFile.js暴露的其实就是一个东西，他们是实时互通的，而不是暴露出去就断开联系了(从指针看就是指向同一个内存地址)

module.exports导出

跟上面的exports导出不一样

- 但是Node中我们经常导出东西的时候，又是通过module.exports导出的：
 - module.exports和exports有什么关系或者区别呢？
- 我们追根溯源，通过维基百科中对CommonJS规范的解析：
 - CommonJS中是没有module.exports的概念的
 - 但是为了实现模块的导出，Node中使用的是Module的类，每一个模块都是Module的一个实例，也就是module
 - 所以在Node中真正用于导出的其实根本不是exports，而是module.exports(所以最终找的其实不是exports，而是module.exports)
 - 因为module才是导出的真正实现者
- 但是，为什么exports也可以导出呢？
 - 这是因为module对象的exports属性是exports对象的一个引用(引用赋值)
 - 也就是说 module.exports = exports = main中的bar（直到我们使用module.exports导出一个对象就不相等了，因为就重新开辟内存空间0x200了）

```
let name1 = "xiaoyu"
let age = 20
```

//将模块中的内容导出，这种导出方式，开辟了module对象，此对象里面有exports，而这个exports跟上面第一种导出方式一样指向了内存地址

```
module.exports.name1 = name1
module.exports.age = age
```

//所以本质上是一样的

```
console.log(exports === module.exports)//true
```

前端模块化开发 > 03_CommonJS_02 > JS foo.js > ...

```
// 1. 在开发中使用的很少
// exports.name = name
// exports.age = age
// exports.sayHello = sayHello

// 2. 将模块中内容导出
module.exports.name = name
module.exports.age = age
module.exports.sayHello = sayHello
```

exports.name = "foo"

module.exports.name = "foo"

0x100

exports: 0x100

module对象

exports: 0x100

{ name: "foo", age: 18 }

const bar = 0x100

问题 输出 调试控制台 终端

//真实开发中的写法，下面的图显示在内存中发生的改变

```
module.exports = {
  //xxx
}
```

//那我们再修改exports.xxx = xxx就没有用了，因为我们exports指向的0x100内存地址已经不同了，我们放东西的地方再0x200了

```
me, "----")
e, "----")
yHello, "----")
odule.exports)
```

exports

0x100

0x200

module

exports: 0x200

module.exports={}的{}指向另一个地方0x200了

const bar = require("./foo.js") // module.exports -> 0x200

const bar = 0x200

> cr

(掌握)Node中module的exports属性本质

改变代码发生了什么？

- 改变代码发生了什么？

1. 在三者项目引用的情况下，修改exports中的name属性到底发生了什么？
 - 三者的name属性同时被修改，因为此时三者这时候指向的是同一个内存地址，三者指(module、export、main中导入的bar)
2. 在三者引用的情况下，修改了main中的bar的name属性，在bar模块中会发生什么？
 - 理论同步修改(但是不允许倒着修改)
3. 如果module.exports不再引用exports对象了，那么修改export还有意义吗？
 - 没有意义，因为不会产生作用。因为修改的都不是一个地方了

改变了代码什么，我们已经在内存分析中看到了很多

(理解)Node中require查找模块的细节

require细节

- 我们现在已经知道，require是一个函数，可以帮助我们引入一个文件（模块）中导出的对象。

require的查找规则

- 那么，require的查找规则是怎么样的呢？
 - 这里我总结比较常见的查找规则：
 - 导入格式如下：require(X)

情况一：

- X是一个Node核心模块，比如path、http
 - 直接返回核心模块，并且停止查找

```
//导入Node提供给内置模块
const path = require("path");//直接返回内置模块
console.log(path)
```

情况二：

- X是以 ./ 或 ../ 或 /（根目录）开头的
 - 第一步：将X当做一个文件在对应的目录下查找
 1. 如果有后缀名，按照后缀名的格式查找对应的文件
 2. 如果没有后缀名，会按照如下顺序：
 - 直接查找文件X
 - 查找X.js文件
 - 查找X.json文件

- 查找X.node文件
- 第二步：没有找到对应的文件，将X作为一个目录
 - 查找目录下面的index文件
 1. 查找X/index.js文件
 2. 查找X/index.json文件
 3. 查找X/index.node文件
- 如果没有找到，那么报错：not found

//1. 根据路径导入自己编写模块

//先把这个当成文件，在当前文件夹下查找，从后缀js、json、node按顺序找，没找到就把这个当文件夹，找这个文件夹下的index.js、index.json、index.node文件

```
const utils = require("./utils");//省略后缀.js
console.log(utils.formData())
```

情况三：

- 直接是一个X（没有路径），并且X不是一个核心模块
- /Users/你的路径/main.js 中编写 require('why')

查找顺序(从上往下):

```
paths: [
  '/Users/coderwhy/Desktop/Node/TestCode/04_learn_node/05_javascript-module/02_commonjs/node_modules',
  '/Users/coderwhy/Desktop/Node/TestCode/04_learn_node/05_javascript-module/node_modules',
  '/Users/coderwhy/Desktop/Node/TestCode/04_learn_node/node_modules',
  '/Users/coderwhy/Desktop/Node/TestCode/node_modules',
  '/Users/coderwhy/Desktop/node_modules',
  '/Users/coderwhy/node_modules',
  '/Users/node_modules',
  '/node_modules'
]
```

- 如果上面的路径中都没有找到，那么报错：not found

但是有个例外，那就是在Vue跟react中常用的另外一种引入方式

- 在node_modules文件夹下引入包

//创建node_modules文件夹

//在此文件夹下创建xiaoyu文件夹，xiaoyu文件夹下再创建index.js文件

//在index.js文件

```
exports.modules = {
  name: "小余6666"
}
```

//在main.js主文件引入

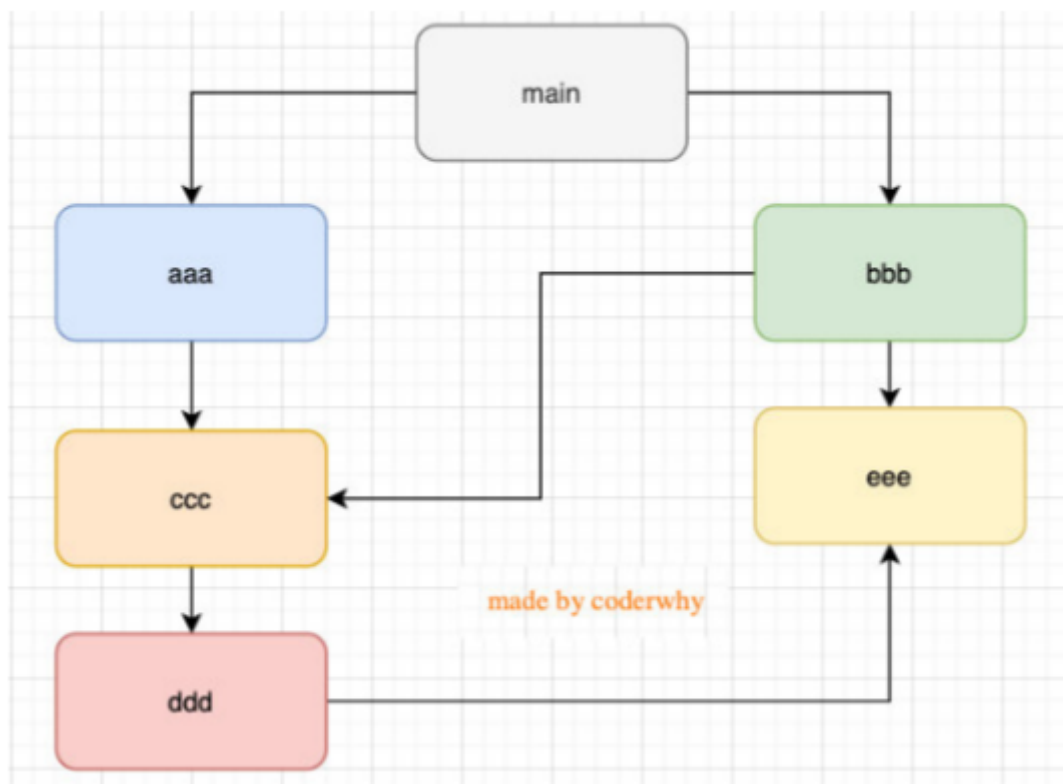
```
const xiaoyu = require("xiaoyu");//引入，从node_modules中引入的
console.log(xiaoyu);//输出
```

- node main.js在终端输出

```
{ modules: { name: '小余6666' } } //成功输出
```

(理解)Node模块的加载过程解析

模块的加载过程



- 结论一：模块在被第一次引入时，模块中的js代码会被运行一次
- 结论二：模块被多次引入时，会缓存，最终只加载（运行）一次
 - 为什么只会加载运行一次呢？
 - 这是因为每个模块对象module都有一个属性：loaded
 - 为false表示还没有加载，为true表示已经加载
- 结论三：如果有循环引入，那么加载顺序是什么？
- 如果出现上图模块的引用关系，那么加载顺序是什么呢？
 - 这个其实是一种数据结构：图结构
 - 图结构在遍历的过程中，有深度优先搜索（DFS, depth first search）和广度优先搜索（BFS, breadth first search）
 - Node采用的是深度优先算法：main -> aaa -> ccc -> ddd -> eee -> bbb

```
//也就是说我们在执行文件里面的代码的时候，如下
console.log("1")
const foo = require("./xxx")
console.log("2")
//我会先执行1，在跳到./xxx文件中去执行./xxx文件的内容(如果./xxx里面还有引入就继续跳)，然后跳回来继续执行2
//形成了一整套相互依赖的关系

//然后就是我们重复引用是不会反复执行的，只会执行第一次引入的
//只会执行一次的原理：module里面有一个loaded，没引用过内容为false，引用了一次变为true。下次再引入看到这里为true就会跳过
const foo2 = require("./xxx");//不执行
```

(了解)AMD和CMD规范的简单介绍

CommonJS规范缺点

- **CommonJS加载模块是同步的：**
 - 同步的意味着只有**等到对应的模块加载完毕，当前模块中的内容才能被运行**
 - 这个在服务器不会有什么问题，因为服务器**加载的js文件都是本地文件**，加载速度非常快
- **如果将它应用于浏览器呢？**
 - 浏览器**加载js文件**需要先从服务器将文件下载下来，之后再加载运行；
 - 那么采用**同步的**就意味着后续的js代码都无法正常运行，即使是一些简单的DOM操作
- **所以在浏览器中，我们通常不使用CommonJS规范：**
 - 当然在webpack中使用CommonJS是另外一回事
 - 因为它会将我们的代码转成浏览器可以直接执行的代码
- 在早期为了可以在浏览器中使用模块化，通常会采用**AMD或CMD**
 - 但是目前一方面现代的浏览器已经支持**ES Modules**，另一方面借助于webpack等工具可以实现对**CommonJS或者ESModule**代码的转换
 - **AMD和CMD已经使用非常少了**

AMD规范

- **AMD主要是应用于浏览器的一种模块化规范：**
 - AMD是**Asynchronous Module Definition（异步模块定义）**的缩写；
 - 它采用的是**异步加载模块**
 - 事实上AMD的规范还要早于CommonJS，但是CommonJS目前依然在被使用，而AMD使用的较少了
- 我们提到过，**规范只是定义代码的应该如何去编写**，只有有了具体的实现才能被应用
 - AMD实现的比较常用的库是**require.js和curl.js**

require.js的使用

- 第一步：下载require.js
 - 下载地址：<https://github.com/requirejs/requirejs>
 - 找到其中的require.js文件
- 第二步：定义HTML的script标签引入require.js和定义入口文件：
 - data-main属性的作用是在加载完src的文件后会加载执行该文件

```
<script src="./lib/require.js" data-main="./index.js"></script>
```

```
(function() {
  require.config({
    baseUrl: '',
    paths: {
      foo: './modules/foo',
      bar: './modules/bar'
    }
  })

  require(['foo'], function(foo) {
  })
})();

define(['bar'], function(bar) {
  console.log(bar.name);
  console.log(bar.age);
  bar.sayHello('kobe');
})

define(function() {
  const name = "coderwhy";
  const age = 18;
  const sayHello = function(name) {
    console.log("Hello " + name);
  }

  return {
    name,
    age,
    sayHello
  }
})
```

CMD规范

- CMD规范也是应用于浏览器的一种模块化规范：
 - CMD 是Common Module Definition（通用模块定义）的缩写
 - 它也采用的也是异步加载模块，但是它将CommonJS的优点吸收了过来
- CMD也有自己比较优秀的实现方案
 - SeaJS

SeaJS的使用

- 第一步：下载SeaJS
 - 下载地址：<https://github.com/seajs/seajs>
 - 找到dist文件夹下的sea.js
- 第二步：引入sea.js和使用主入口文件
 - seajs是指定主入口文件的

```
<script src="./lib/sea.js"></script>
<script>
  sea.js.use('./index.js');
</script>
define(function(require, exports, module) {
  const foo = require('./modules/foo');
})
```

```
define(function(require, exports, module) {
  const bar = require('./bar');

  console.log(bar.name);
  console.log(bar.age);
  bar.sayHello("韩梅梅");
})
```

```
define(function(require, exports, module) {
  const name = 'lilei';
  const age = 20;
  const sayHello = function(name) {
    console.log("你好 " + name);
  }

  module.exports = {
    name,
    age,
    sayHello
  }
})
```