# Node.js课程

尚硅谷前端研究院

版本：V 1.1

# 第1章：Node.js

## 1.1 Node.js诞生史

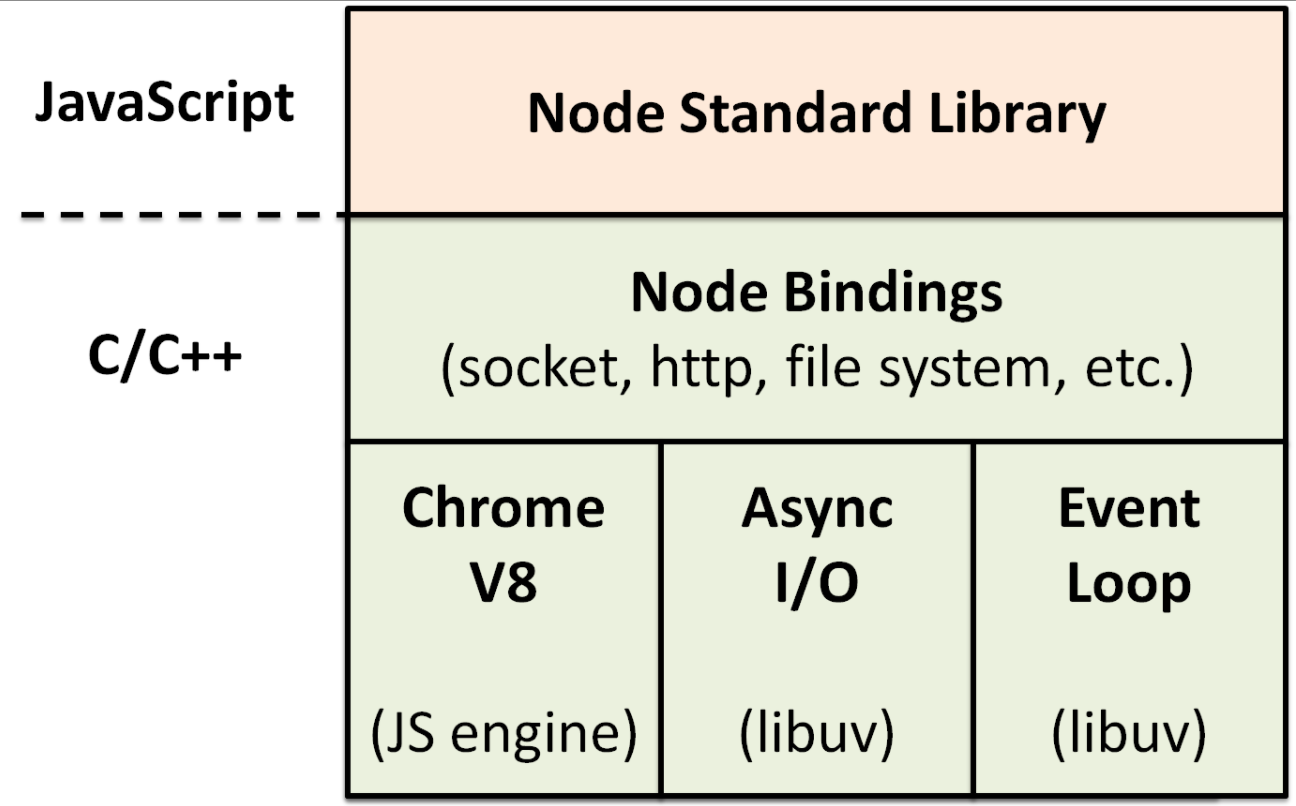
Node.js之父：Ryan Dahl（瑞安·达尔）



* 并非科班出身的开发者，在2004年在纽约的罗彻斯特大学数学系读博士。
* 2006年退学，来到智利的Valparaiso小镇。
* 期间曾熬夜做了一些不切实际的研究，例如如何通过云进行通信。
* 偶然的机会，走上了编程之路，生活方式变为接项目，然后去客户的地方工作。
* 工作中遇到了主流服务器的瓶颈问题，尝试着自己去解决，费尽周折没有办法。
* 2008年Google公司Chrome V8引擎横空出世，JavaScript脚本语言的执行效率得到质的提升，他的想法与Chrome V8引擎碰撞出激烈的火花。
* 2009年的2月，按新的想法他提交了项目的第一行代码，这个项目的名字最终被定名为“node”。
* 2009年5月，正式向外界宣布他做的这个项目。
* 2009年底，Ryan Dahl在柏林举行的JSConf EU会议上发表关于Node.js的演讲，之后Node.js逐渐流行于世。
* Ryan Dahl于2010年加入Joyent公司，全职负责Node.js项目的开发。此时Node.js项目已经从个人项目变成一个公司组织下的项目。

## 1.2 Node.js是什么

Node.js 是一个基于 Chrome V8 引擎的 JavaScript 运行环境。

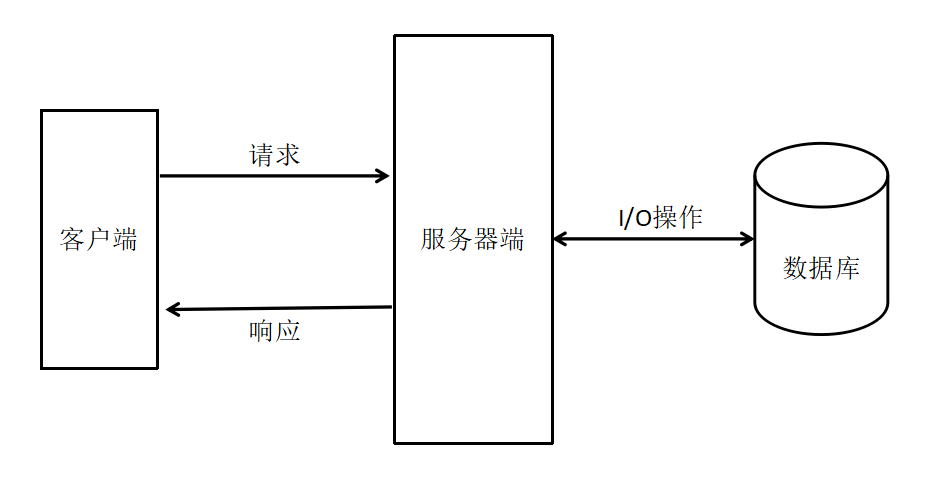


## 1.3 Node.js有什么特点

### 1.3.1 优点

1. 异步非阻塞的I/O（I/O线程池）
2. 特别适用于I/O密集型应用
3. 事件循环机制
4. 单线程
5. 跨平台

简单web交互模型：



### 1.3.2 不足之处

1. 回调函数嵌套太多、太深（俗称回调地狱）
2. 单线程，处理不好CPU 密集型任务

# 1.3.3 [CPU密集型和I/O密集型区别](https://www.cnblogs.com/amiezhang/p/7631239.html)

CPU密集型

一些进程绝大多数时间在计算上，称为计算密集型（CPU密集型）computer-bound。一些大量循环的代码（例如：图片处理、视频编码、人工智能等）就是CPU密集型。

I/O密集型

有一些进程则在input 和output上花费了大多时间，称为I/O密集型，I/O-bound。比如搜索引擎蜘蛛大多时间是在等待相应这种就属于I/O密集型。

Nodejs

众所周知node.js是单线程的，正常来说nodejs处理CPU密集型就比较乏力，毕竟只有一条线程。

node从他推出至今，充满赞美和饱受诟病的都是其单线程模型，所有的任务都在一个线程中完成（I/O等例外），优势的地方自然是免去了频繁切换线程的开销，以及减少资源互抢的问题等等，但是当nodejs面对cpu密集型模型的时候就力不从心了。尽管node拥有异步机制，可以把一些耗时算法丢入eventloop等待下个事件循环再做，但是因为其任然是单线程模型，所以终究会造成阻塞

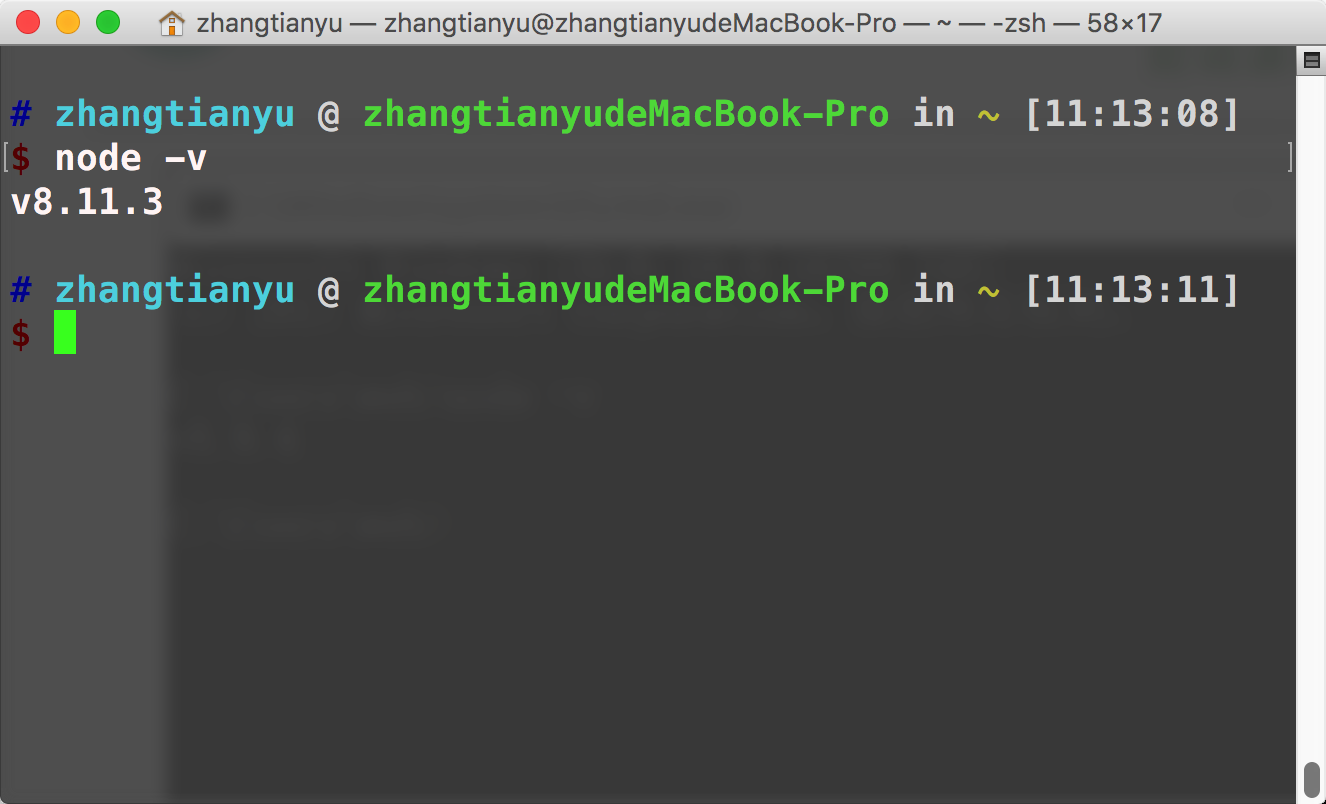
## 1.4 Node.js的应用场景

1. Web服务API，比如RESTful API（本身没有太多的逻辑，只需要请求API，组织数据进行返回即可）
2. 服务器渲染页面，提升速度
3. 后端的Web服务，例如跨域、服务器端的请求

## 1.5 Node.js的安装

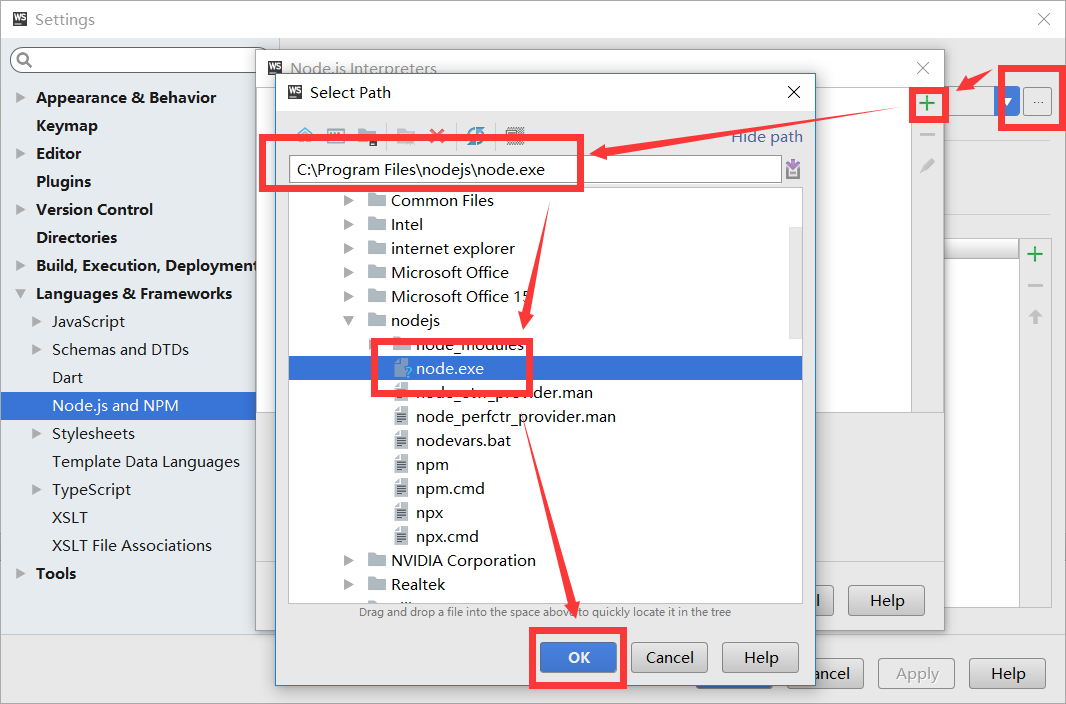
一键傻瓜式安装

安装完成之后，打开命令行窗口，输入node -v查看当前node版本



## 1.6 WebStorm配置

File ---> Settings ---> Languages & Frameworks ---> Node.js and NPM



# 第2章：模块化

## 2.1模块化历史

2015年之前的Javascript，就缺乏一个功能：模块。

本身的ECMA script规范比较薄弱，还有以下缺陷：

1. 没有模块系统
2. 标准库较少
3. 没有标准接口
4. 缺乏包管理系统

如果程序设计达到一定规模，就必须对其模块化。

## 2.2模块化是什么

模块化指的就是将一个大的功能拆分为一个一个小的模块，通过不同的模块的组合来实现一个大功能。

Node中使用的是CommonJS规范来实现的模块化，前端使用的模块化规范是AMD和CMD

## 2.3 CommonJS

CommonJS 是一套模块化规范，它包含：模块、二进制、Buffer、字符集编码、I/O流、进程环境、文件系统、套接字、单元测试、Web服务器网关接口、包管理等。

Node借鉴了commonjs的规范实现了一套模块系统，我们也叫做commonJS模块化系统。

### 2.3.1模块引用

1. 通过require()函数来引入外部的模块
2. 它需要一个模块的标识作为参数，返回一个对象表示引入的模块

|  |
| --- |
| *// 例子：* **var** a = require(**"../a"**); **var** math = require(**"math"**); |

### 2.3.2模块定义

1. 在node中一个js文件就是一个模块
2. 在node中每一个模块中的代码都是运行在一个独立的函数中的，
3. 默认情况下模块内部代码对于外部来说都是不可见的，可以通过两种方式向外部暴露变量和函数
   1. 可以通过将变量和函数设置为 exports 的属性来暴露变量和函数

|  |
| --- |
| *// 例子：* **exports**.**name** = **"xxx"**; **exports**.sayHello = **function**(){...}; |

* 1. 也可以通过module.exports来向外部暴露变量和函数

|  |
| --- |
| *// 例子：* module.**exports**.**name** = **"xxx"**; module.**exports**.sayHello = **function**(){...}; module.**exports** = {  **a**:**"xxx"**,  **b**:**"xxx"** }; |

### 2.3.3模块标识

1. 引入外部模块时，需要通过模块的标识进行引入
2. **对于自定义的文件模块**，模块的标识就是文件的路径（绝对路径、以 . 或 .. 开头的相对路径）

例子："./a" "../b"

1. **对于下载的模块或系统模块模块的标识就是文件的名字**

例子："fs" "express"

### 2.3.4 module.exports和exports的区别

exports变量是对module的exports属性的引用，我们在向exports中添加属性时，本质上是在向module.exports中添加属性。module.exports 可以直接通过赋值的形式来暴露内容。exports 不能直接赋值，只能通过.的形式添加属性。

如果是简单的暴露属性，使用exports就好，引入模块使用方式： m1.属性名

如果需要向外暴露一个构造函数，使用module.exports， 引入模块使用方式：m1

### 2.3.5 node中的函数

通过**arguments.callee.toString()**可以看到一个外层函数：

function (exports, require, module, \_\_filename, \_\_dirname) {}

这个函数是所有模块都有的，node编译时往其中注入5个参数：

exports 暴露模块

require 引入模块

module exports属性暴露模块

\_\_filename 当前文件的绝对路径

\_\_dirname 当前文件所在文件夹的绝对路径

这个外层函数的作用：1.能让Node直接使用CommonJS语法

2.隐藏内部实现

Node\_global:

1. 在Node.js运行环境中：console.log(this);//{} console.log(global);//global
2. 在浏览器中，console.log(this);//window
3. 对于浏览器端而言，js由哪几部分组成？
4. BOM 浏览器对象模型 -------- 很多的API（location，history）  
   2.DOM 文档对象模型 ---------- 很多的API（对DOM的增删改查）  
   3.ES规范 -------------------- ES5、ES6.....

2.Node端js由几部分组成？

1.没有了BOM ----- 因为服务器不需要（服务端没有浏览器对象）  
2.没有了DOM ----- 因为没有浏览器窗口  
3.几乎包含了所有的ES规范  
4.没有了window，但是取而代之的是一个叫做global的全局变量。

## 2.4 Node引入模块的流程

### 2.4.1路径分析

核心模块：在Node源码编译就完成了，直接被加进内存中，如（http fs path）。

路径形式的文件模块：以”./” “../”路径形式的文件模块，将其转化为真实路径，根据真实路径 索引去查找。

第三方模块：它会从文件的node\_modules逐层往上找，直到根目录的node\_modules。

### 2.4.2文件定位

require()中的文件，如果没有后缀名，会以.js .json .node次序补充扩展名，依次尝试。

### 2.4.3编译执行

.js fs模块同步读取文件编译执行

.json fs模块同步读取文件，用JSON.parse()解析返回结果

.node 这是c/c++编写的扩展文件，通过dlopen()方法编译

其他扩展名 会以.js文件载入

# 第3章：包和包管理器

## 3.1 package包

我们电脑上的文件夹，包含了某些特定的文件，符合了某些特定的结构，就是一个包

Node.js的包基本遵循CommonJS规范，包将一组相关的模块组合在一起，形成一组完整的工具。

包由包结构和包描述文件两个部分组成。

1. 包结构：用于组织包中的各种文件
2. 包描述文件：描述包的相关信息，以供外部读取分析

### 3.1.1包结构

包实际上就是一个压缩文件，解压以后还原为目录。符合CommonJS规范的目录，一个标准的包，应该包含如下文件：

1. package.json 描述文件(包的 “说明书”，必须要有！！！）
2. bin 可执行二进制文件
3. lib 经过编译后的js代码
4. doc 文档（说明文档、bug修复文档、版本变更记录文档）
5. test 单元测试（一个测试报告）

### 3.1.2包描述文件

包描述文件用于表达非代码相关的信息，它是一个JSON格式的文件：package.json

包描述文件包含以下字段：name、version、description、keywords、maintainers、contributors、bugs、licenses、repositories、dependencies、homepage、os、cpu、engine、builtin、directories、implements、scripts、author、bin、main、devDependencies。

如何让一个普通文件夹变成一个包？  
 让这个文件夹拥有一个：package.json文件即可,且package.json里面的内容要合法。  
 执行命令：npm init  
 包名的要求：不能有中文、不能有大写字母、不能与npm仓库上其他包同名。

## 3.2 NPM是什么

全称：Node Package Manager , 是Node的包管理器，专门用于管理包

## 3.3 NPM能干什么

通过NPM可以对Node的包进行搜索、下载、安装、删除、上传

NPM的常用指令：

1. npm -v

查看npm的版本

1. npm init

初始化项目的package.json文件

*一、【搜索】：*  1.npm search xxxxx  
 2.通过网址搜索：www.npmjs.com  
 *二、【安装】：(安装之前必须保证文件夹内有package.json，且里面的内容格式合法)*1.npm install / i 包名安装指定的包

2.npm install xxxxx --save 或 npm i xxxx -S 或 npm i xxxx  
 备注：

(1).安装完的第三方包，放在node\_modules这里  
 (2).会自动产生一个package-lock.json(npm版本在5.x.x以后才有该文件)，里面缓存的是每个下载过的包的地址，目的是为了下次安装时速度快一些。  
 (3).当安装完一个包，该包的名字会自动写入到package.json中的【dependencies(生产依赖)】里。  
  
 3.npm install xxxxx --save-dev 或 npm i xxxx -D 安装包并将该包写入  
 到【devDependencies(开发依赖中)】  
 备注：什么是生产依赖与开发依赖？  
 1.只在开发(写代码时)时需要依赖的库，就是开发依赖 ----- 例如：语法检查库、压缩代码、扩展css前缀的库。  
 2.生产环境中必不可少的库，就是生产依赖 ------ 例如：jquery。  
 备注：所谓生产就是：项目开发完毕，要部署到服务器上运行。  
 3.某些库即属于开发依赖，又属于生产依赖 -------例如：jquery。  
  
 4.npm i xxxx -g 全局安装xxxx包（一般来说，带有指令集的包要进行全局安装，例如：browserify、babel等）  
 查看全局安装的位置：npm root -g  
I  
 5.npm i xxx@yyy :安装xxx包的yyy版本  
  
 6npm i ：安装package.json中所有声明的包//安装项目中的所有依赖  
 *三、【移除】：* npm remove xxxxx 移除xxxx包，并且删除该包在package.json中的声明 *四、【其他命令】：*  1.npm aduit fix :检测项目依赖中的一些漏洞，并且尝试着修复。  
  
 2.npm view xxxxx versions :查看npm仓库中xxxx包的所有版本信息  
  
 3.npm view xxxxx version :查看npm仓库中xxxx包的最新版本  
  
 4.npm ls xxxx :查看我们所安装的xxxx包的版本 *五、【关于版本号的说明】：* "^3.x.x" ：锁定大版本，以后安装包的时候，保证包是3.x.x版本，x默认取最新的。  
 "~3.1.x" ：锁定小版本，以后安装包的时候，保证包是3.1.x版本，x默认取最新的。  
 "3.1.1" ：锁*定完整版本，以后安装包的时候，保证包必须是3.1.1版本。*

## 3.4 cnpm

### 3.4.1 cnpm是什么

它是淘宝对国外npm服务器的一个完整镜像版本，也就是**淘宝 NPM 镜像**

### 3.4.2 cnpm的安装

npm install -g cnpm --registry=https://registry.npm.taobao.org

### 3.4.3 cnpm的使用

cnpm和npm的使用基本没有区别，只需要将npm替换成cnpm

## 3.5扩展

2017年5月30日发布node 8.0，其中自带的npm也由3.xxx版本升级到5.0

npm5变化：

通过npm下载包的时候多了一个package.lock.json

package.lock.json可以记录整个node-modules中文件夹的树状结构，再次下载的时候不用先去读取包与包之间相关依赖

说白了：就是记录包与包之间的关联(依赖关系)

好处：再次npm install下载的时候不用去先读取依赖可以直接下载，速度较快。可以利用离线缓存，提高了下载速率。

## 3.6注意

npm 3.xxx 到 npm 5.xxx 并没有完美对接。

问题：之前通过npm3下载好了依赖，现在升级为npm5发现下不了包了。

解决方案：删除之前的node\_modules和package-lock.json，重新npm install

## 3.7 Yarn

### 3.7.1 yarn是什么

yarn是Facebook开源的新的包管理器，可以用来代替npm。

### 3.7.2 yarn的特点

有缓存。

没有自己的仓库地址，使用的依然是npm仓库地址。

### 3.7.3 yarn的安装

npm install yarn -g

### 3.7.4常用命令

1. yarn --version
2. yarn init //生成package.json ！！！注意生成的包名不能有中文，大写
3. yarn global package (全局安装)
4. yarn add package (局部安装)
5. yarn add package --dev (相当于npm中的--save-dev)
6. yarn remove package
7. yarn list //列出已经安装的包名
8. yarn info packageName //获取包的有关信息
9. yarn //安装package.json中的所有依赖

### 3.7.5 Cyarn

yarn引用npm的仓库，因为‘墙’的存在，可能会导致下载不了或速度很慢的情况，所以需要引入cyarn（淘宝镜像）

npm install cyarn -g --registry "https://registry.npm.taobao.org"

配置后，只需将yarn改为cyarn使用即可。

# 第4章：Buffer缓冲器

## 4.1 Buffer是什么

Buffer是一个和数组类似的对象，用于存储数据Buffer是专门用来保存二进制数据的，（和数组，对象不同是，数组和对象保存的是基本数据类型和引用数据类型）。

## 4.2 Buffer特点

1. 大小固定：在创建时就确定了，且无法调整
2. 性能较好(效率很高)：存储和读取很快，直接对计算机的内存进行操作
3. 每个元素占内存大小为1字节（byte）
4. Buffer是Node中的非常核心的模块，无需下载、无需引入即可使用

进制：  
 十六进制：00 ----- ff

十进制：0 ----- 255

二进制：000000000 ------ 11111111

换算：  
1byte（字节） = 8bit（位）  
1024 byte = 1 KB  
1024 KB = 1 MB

## 4.3 Buffer的使用

### 4.3.1创建Buffer

|  |
| --- |
| *1.创建一个指定size大小的Buffer实例-----即将被废弃（效率很低）*  **Var** buf = **new Buffer(size)**；   1. *创建一个指定size大小的Buffer实例* **var** buf = ***Buffer*.alloc(size);** *//安全，里面全是0* 2. *创建一个指定size大小的Buffer实例* **var** buf = ***Buffer.allocUnsafe(size);*** *//效率最高，但不安全，可能包含旧数据，需要重写所有数据 // 获取Buffer占用内存的空间。通过Buffer的length属性*   buf.**length** |

### 4.3.2 转换

|  |  |
| --- | --- |
| *// 相当于Buffer.alloc(size);* **var** buf = ***Buffer***.allocUnsafe(size); buf.fill(0) *//将可能出现的敏感数据用0全部填充*   |  | | --- | | **var *buf*** = Buffer.allocUnsafe(10); ***console***.log(***buf***);*//<Buffer 0e 00 00 00 06 02 00 00 06 00>* ***buf***.fill(0);*//<Buffer 00 00 00 00 00 00 00 00 00 00>* ***console***.log(***buf***); |   *// 将一个字符串转换为Buffer==将一个字符串存入到Buffer中*  *Var str = “hello!atguigu“* **var** buf = ***Buffer***.from(str);  *// 将一个Buffer转换为字符串* **var** str = buf.toString();//为什么buffer自己不自动转换，因为如果是音频视频类的文件，自动转化的话就乱码了 |

# 第5章：fs文件系统

## 5.1 fs是什么

全称为file system，所谓的文件系统，就是对计算机中的文件进行增删改查等操作。它是一个服务器的基础，在Node中通过fs模块来操作文件系统。

## 5.2 fs的使用

1. fs模块是Node的核心模块，不需要下载，直接引入即可使用

|  |
| --- |
| *// 引入fs模块* **var** fs = require(**"fs"**); |

1. fs中的大部分方法都为我们提供了两个版本：
   1. 同步方法：带sync的方法
      1. 同步方法会阻塞程序的执行
      2. 同步方法通过返回值返回结果
   2. 异步方法：不带sync的方法
2. 异步方法不会阻塞程序的执行
3. 异步方法都是通过回调函数来返回结果的

## 5.3文件的写入

### 5.3.1简单写入

|  |
| --- |
| 同步方法：fs.writeFileSync(file, **data**[, options])  异步方法：fs.writeFile(file, **data**[, options], callback)  参数：  file 要写入的文件的路径  data 要写入的内容，可以是一个String也可以是一个Buffer  options 配置对象，需要一个对象作为参数，默认如下：  {encoding:"utf8",flag:"w",mode:0666}  callback 回调函数 |

### 5.3.2流式写入

流式文件写入适用于一些比较大的文件，可以分多次向文件中写入内容，有效避免内存溢出的问题

|  |
| --- |
| 1.创建一个可写流  **var** ws = fs.createWriteStream(path); 2.监听流的状态  ws.once(**"open"**,**function**(){});  ws.once(**"close"**,**function**(){}); 3.向流中写入内容  ws.write(...);  ws.write(...);  ws.write(...); 4.关闭流  ws.end(); |

## 5.4文件的读取

### 5.4.1 简单读取文件

|  |
| --- |
| fs.readFileSync(path[, options]) fs.readFile(path[, options], callback)  参数：  path 读取文件的路径  options 配置对象  encoding 指定打开文件的编码，默认是null  flag 文件的操作类型，默认是 r  callback 回调函数，通过回调函数返回读取到的数据  err 错误对象  data 返回的数据（Buffer） |

### 5.4.2 流式读取文件

适合较大的文件

|  |
| --- |
| *// 方式一： //创建一个可读流* **var** rs = fs.createReadStream(**"C:\\other\\MP3\\笔记.mp3"**); *//创建一个可写流* **var** ws = fs.createWriteStream(**"biji.mp3"**);  *//监听可读流，可读流读取完毕后会自动关闭* rs.once(**"open"** , **function** () {  ***console***.log(**"流打开了"**); }); rs.once(**"close"** , **function** () {  ***console***.log(**"流关闭了"**); *//数据读取完毕，可以关闭可写流* ws.end(); });  *//监听可写流* ws.once(**"open"** , **function** () {  ***console***.log(**"可写流打开了"**); }); ws.once(**"close"** , **function** () {  ***console***.log(**"可写流关闭了"**); }); *//要通过可读流读取一个文件，必须为流绑定一个data事件，当绑定了data事件以后，流会自动读取* rs.on(**"data"**,**function** (data) { *//console.log(data); //将数据写入到可写流中* ws.write(data); });  *//方式二： //创建一个可读流* **var** rs = fs.createReadStream(**"C:\\other\\MP3\\笔记.mp3"**); *//创建一个可写流* **var** ws = fs.createWriteStream(**"biji.mp3"**); */\*  pipe()  - 可以将一个可读流中的内容写入到一个可写流中  \*/* rs.**pipe**(ws); |