

Node.js 学习笔记

书籍：《Node.js 开发指南》—郭家宝

C1·概述

Node.js:

- 可以让 JS 运行在服务器端的平台，可以脱离浏览器
- 单线程、异步 io、事件驱动
- 包管理器：npm, node package manager
- Node 不是独立语言（PHP、Python、Perl、ruby‘既是语言又是平台’）；不是 js 框架，不是浏览器端的库（bootstrap 应该是）
- 它是 JS 成为脚本语言在服务端流行
- 不存在 JS 浏览器兼容性问题

用途：

- ❑ 具有复杂逻辑的网站；
- ❑ 基于社交网络的大规模 Web 应用；
- ❑ Web Socket 服务器；
- ❑ TCP/UDP 套接字应用程序；
- ❑ 命令行工具；
- ❑ 交互式终端程序；
- ❑ 带有图形用户界面的本地应用程序；
- ❑ 单元测试工具；
- ❑ 客户端 JavaScript 编译器。

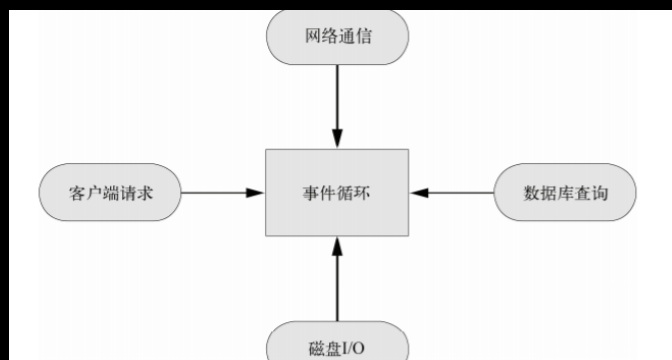
异步式 IO

对于高并发，传统架构：多线程。

Node: 单线程，所有的 io 请求都采用异步式，避免频繁上下文切换。

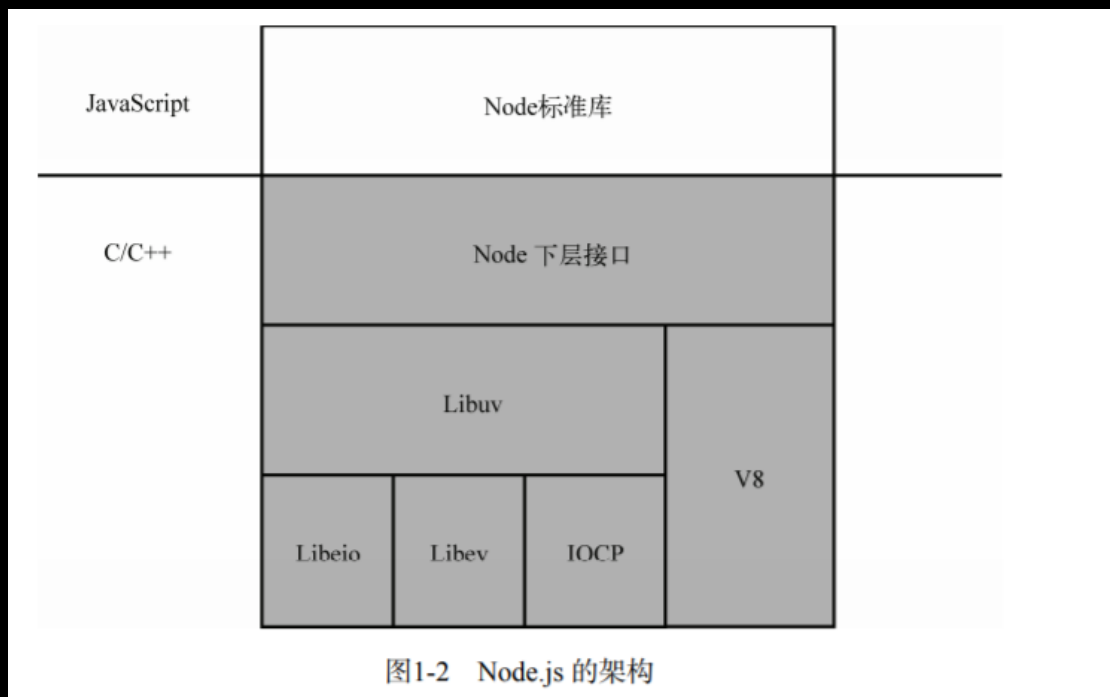
使用事件队列，请求属于非阻塞式。

异步式 io 和事件驱动代替多线程



node 架构

POSIX (Portable Operating System Interface): 一套操作系统 API 规范。



IOCP (Input/Output Completion Port, 输入输出完成端口)

简史

Nombas公司: C minus minus → ScriptEase
 网景: LiveWire → LiveScript
 Sun: java applet → JavaScript

网景最先推出 Navigator

后微软推出 Internet Explorer—使用了 Jscript (类似 JS)、老本 VBScript

JS 标准化—ECMAScript

js 引擎革命:

最先: SpiderMonkey, 作为 Firefox3.0 之前的引擎, 3.5 之后换为 TraceMonkey

Chrome 引擎是 V8, V8 也是 node.js 的引擎

IE9 开始使用 Chakra

Node 不是第一个在服务端的尝试, ASP 才第一次发挥了 js 在服务端的威力

CommonJS—统一 js 在浏览器之外的实现



C2·安装与配置

“开源死敌”微软也支持开源的 node 了

命令行 node 即可进入交互模式

包管理: npm, 完全由 JavaScript 实现的命令行工具, 通过 node 执行

Node 调试工具 supervisor

Node 修改后每次都需要重新启动, 因为 Node.js 只有在第一次引用到某部份时才会去解析脚本文件, 以后都会直接访问内存, 避免重复

载入，不便于调试

使用其他工具，监视代码，一旦修改重新启动

1. `npm -g install supervisor`(安装)
2. 用 `cd` 命令定位到项目的根目录
3. `supervisor bin/www`

C3·Node.js 快速入门

编写 `HelloWorld.js`，添加 `console` 语句，进入命令行，`node HelloWorld.js`

JS 的 `Console` 语句可以输出很多中类型，`console.log`，`console.error`，`console.info`

支持变量引用如 `%s,%d`，类似 C 语言的 `printf`：
`console.log('%s: %d', 'Hello', 25);`

常用 node 命令

- `Node -help`: 帮助
- `Node -v`
- `Node -e` : `evaluate script`, `eval` 函数, `evaluate` (求值)
Eg: `node -e "console.log()"`
- `Node -p`: `print` , `evaluate script` and `print result`
- `Node -c`, `check`, `syntax check script without executing`

REPL 模式

Read-eval-print loop 输入-求值-输出循环

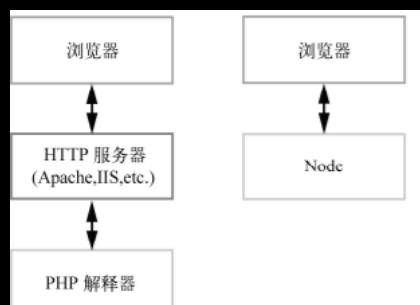
Js 的交互环境，node 进入

退出：两次 Ctrl+C 或 .exit

建立 HTTP 服务器

HPHP 采用“浏览器-Http 服务器-PHP 解释器”模式

而 node 直接面向浏览器用户



```
//server.js 建立服务器
var http = require("http")

http.createServer(function(req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.write("<h1>This is my first Node Server</h1>");
  res.end("<p>Hello EdwinXu</p>")
}).listen(3000) 监听端口

console.log("HTTP server is listening at port 3000")
```

状态码

Node server.js 启动服务器 (Ctrl+C 结束监听)

浏览器就可以访问了

Node 修改时必须终止后重新启动才能运行，不便调试

安装 supervisor 工具，代码改动时自动终止并重启（类似于 spring 的热部署）

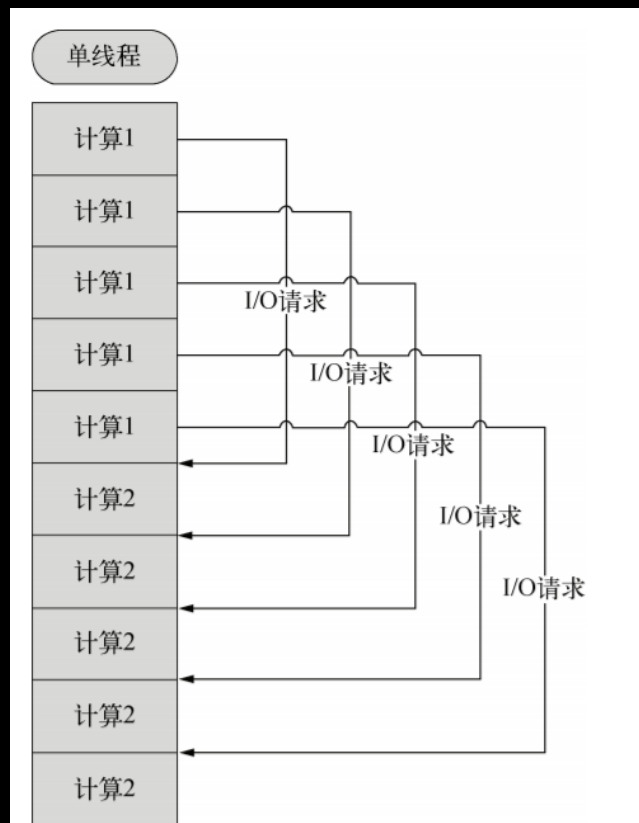
异步式 I/O 与事件式编程

同步式 io (Synchronous/Blocking) /阻塞式 io: 线程遇到磁盘读写、网络通信（统称 IO 操作）时，OS 会剥夺这个线程的控制权，将控制权发放到需要的进程——阻塞。

同步式: OS 不会收回控制权，会帮助该线程执行完 IO 操作，而该线程在发送完 io 请求后继续执行下一条语句，OS 完成 io 后通知该线程。

阻塞模式下，一个线程只能处理一项任务

非阻塞模式下，一个线程永远在执行任务，CPU 利用率 100%



单线程事件驱动的异步式 IO 相比多线程阻塞式 IO 的

好处：

开销变少（OS 创建一个线程代价十分昂贵：分类内存、列入调度、内存换页等）

缺点：控制流晦涩难懂，编码调试较难


同步式 I/O（阻塞式）	异步式 I/O（非阻塞式）
利用多线程提供吞吐量	单线程即可实现高吞吐量
通过事件片分割和线程调度利用多核CPU	通过功能划分利用多核CPU
需要由操作系统调度多线程使用多核 CPU	可以将单进程绑定到单核 CPU
难以充分利用 CPU 资源	可以充分利用 CPU 资源
内存轨迹大，数据局部性弱	内存轨迹小，数据局部性强
符合线性的编程思维	不符合传统编程思维

回调函数：


```
//readFile.js 读取一个文件
var fs = require('fs');
fs.readFile("file.txt", "utf-8", function(err, data) {
    if(err) {console.error(err)}
    else {console.log(data)}
})
console.log("this is the end of this program");

//同步读取
data = fs.readFileSync("file.txt", "utf-8")
console.log(data)
```

异步 请求成功后才
执行回调
先输出



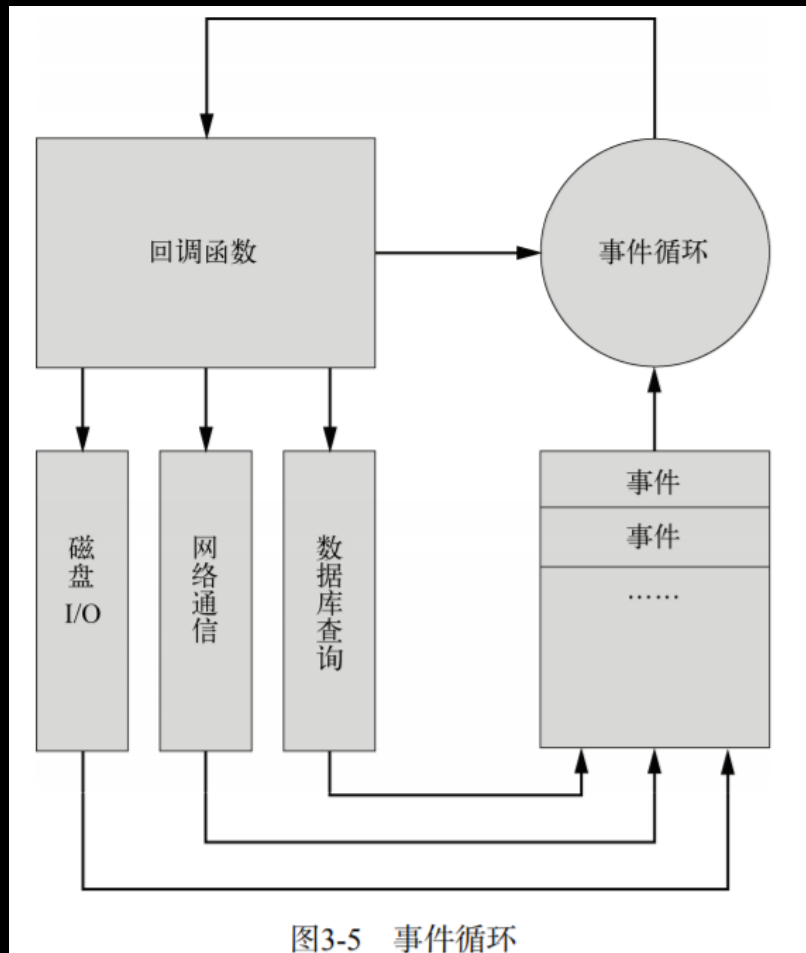
事件:

Node 的所有异步 IO 操作在完成时都会发送一个事件到事件队列

在开发者看来事件是由 `EventEmitter` 对象提供的, 回调函数通过它来实现

事件循环:

`Node.js` 始终在事件循环中, 从第一关回调到最后一个回调。所有逻辑都是事件的回调函数。



模板 module 和包 package

Node 使用 `require` 函数来调用其他 module

模块是 `Node.js` 应用程序的基本组成部分，文件和模块是一一对应的。换言之，一个 `Node.js` 文件就是一个模块，这个文件可能是 `JavaScript` 代码、`JSON` 或者编译过的 `C/C++` 扩展。

`var http = require('http')` 的 `http` 是 `Node.js` 的一个核心模块，其内部是用 `C++` 实现的，外部用 `JavaScript` 封装。我们通过 `require` 函数获取了这个模块，然后才能使用其中的对象。

创建模板：

两个对象：

- Exports: 模块公开的接口
- Require: 用于从外部获取一个模块的接口

```
//module.js 模板文件
var name;
exports.setName = function(name_) {
    name= name_;
}
exports.getName = function() {
    return name;
}
```

```
//getModule.js 模板使用文件
var myModule = require("./module.js")
myModule.setName("EdwinXu")
console.log("My Name Is "+myModule.getName())
```

特点：

- 单次加载：不管创建多少的，都只存在一个该模板对象
- 覆盖 exports: 如果模板文件中只想让部分模板暴露，如只想暴露部分类，使用 `module.exports = ClassName`（不能通过 `export=ClassName`，`export` 在模块结束后就会释放）

创建包：

包是在模板基础上更深一步的抽象，node 包是一个目录，符合 commonJS 的包：

- Package.json（说明文件）必须在包的顶层目录下
- 二进制文件在 bin 目录下

- JavaScript 代码在 lib 下
- 文档在 doc 下
- 单元测试在 test 下

两种包：

- 作为文件夹的模板：

最简单的包，就是一个作为文件夹的模板。

```
//somepackage/index.js
```

```
exports.hello = function() {  
  console.log('Hello.');
```

```
};  
  
然后在 somepackage 之外建立 getpackage.js，内容如下：
```

```
//getpackage.js
```

```
var somePackage = require('./somepackage');  
  
somePackage.hello();
```

- Package.json 配置复杂、完善、强大的包

该 json 中存在的字段：

- Main: 接口模板，说明接口模板的地址，可以具体到一个 js，也可以只是到 lib 文件夹
- Name: 包名，唯一的，小写英文、字母、下划线，不能由空格
- Description: 包的简要说明
- Version: 版本字符串
- Keywords: 关键字数组，通常用于搜索。
- Maintainers: 维护者数组
- Contributors: 维护者数组

- **Bugs:** 提交 bug 的地址
- **Licenses:** 许可证数组
- **Repositories:** 仓库托管地址数组
- **Dependencies:** 包的依赖，一个关联数组，由包名、版本号组成

```
{
  "main": "./lib/index.js",
  "上面这个": "只用 ./lib 也是可以的",
  "name": "Edwin's Pkg",
  "description": "this is a pkg created by EdwinXu",
  "version": "0.1",
  "contributors": [
    {
      "name": "edwinxu",
      "age": 21
    }
  ],
  "bugs": "111@163.com"
}
```

包管理器

- 安装包: `npm [install / i] pkgName`

全局安装: `npm install/I -g pkgName`

- 本地模式 VS 全局模式:

Pip 工具默认安装到全局模式，而 npm 默认安装到当前目录——本地模式，会把包安装到当前目录的 `node_modules` 子目录下，为的是避免不同的版本依赖，缺点是同一个包可能安装多次。

如果需要减少副本，就使用全局安装

本地安装时没有环境变量中配置，不能使用命令行运行

模 式	可通过 <code>require</code> 使用	注册PATH
本地模式	是	否
全局模式	否	是

- 创建全局链接：在本地包和全局包之间创建符号链接

`Npm link` （不支持 win ）

- 包的发布：

- 检查规范性：在包目录下 `npm init`，然后开始创建 `json` 文件

- 获取一个账号用户日后维护：`npm adduser`，`npm whoami` 检查账号正确性

- 发布：`npm publish` 访问 `search.npmjs.org` 就可以找到自己刚发布的包。

取消发布：`npm unpublsh`

常用 NPM 命令

- | | |
|------------------------------|---|
| • <code>npm config</code> | • <code>npm outdated</code> |
| • <code>npm init</code> | • <code>npm update</code> |
| • <code>npm search</code> | • <code>npm run</code> |
| • <code>npm info</code> | • <code>npm cache</code> |
| • <code>npm install</code> | • https://docs.npmjs.com/ |
| • <code>npm uninstall</code> | |
| • <code>npm list</code> | I |

调试

- 命令行调试:

Node debug name.js

表3-3 Node.js 调试命令	
命 令	功 能
run	执行脚本，在第一行暂停
restart	重新执行脚本
cont, c	继续执行，直到遇到下一个断点
next, n	单步执行
step, s	单步执行并进入函数
out, o	从函数中步出
setBreakpoint(), sb()	在当前行设置断点
setBreakpoint('f()'), sb(...)	在函数f的第一行设置断点
setBreakpoint('script.js', 20), sb(...)	在 script.js 的第20行设置断点
clearBreakpoint, cb(...)	清除所有断点
backtrace, bt	显示当前的调用栈
list(5)	显示当前执行到的前后5行代码
watch(expr)	把表达式 expr 加入监视列表
unwatch(expr)	把表达式 expr 从监视列表移除
watchers	显示监视列表中所有的表达式和值
repl	在当前上下文打开即时求值环境
kill	终止当前执行的脚本
scripts	显示当前已加载的所有脚本
version	显示 V8 的版本

- 远程调试

- Eclipse 调试

- Node-inspector 调试: npm install -g node-inspector 安装，在终端中通过 node --debug-brk=5858 debug.js 命令连接你要除错的脚本的调试服务器，启动:node-inspector，浏览器中打开 <http://127.0.0.1:8080/debug?port=5858>

C4·node.js 核心模块

全局对象

Window 是 JS 的全局变量，node 的全局对象是 global，全局变量是全局对象的属性。

- Process 全局变量，描述当前 node 进程状态的对象，提供一个与操作系统的简单接口。
 - Process.argv: 命令行参数数组。第一个元素是 node，第二个是文件地址，第三个开始是运行参数。
 - Process.stdout: 标准输出流。通常我们使用 console.log 向输出流添加字符，而 process.stdout.write() 则是底层的接口，它负责向输出流添加字符，返回 bool 值。
 - Process.stdin: 标准输入流。初始时它是被暂停的，要想从标准输入读取数据，你必须恢复流，并手动编写流的事件响应函数。

```
process.stdin.resume()  
process.stdin.on("data",function(data){  
    console.log(data); //输出ascii码  
    console.log(data.toString()); //输出string  
})
```


- `Process.nextTick(callback)`: 为事件循环设置一项任务，下次事件循环时调用 `callback`。（不要使用 `setTimeout(fn,0)` 代替 `process.nextTick(callback)`，前者比后者效率要低得多。）

- **Console**: 提供控制台标准输出，用于向标准输出流 `stdout` 或标准错误流 `stderr` 输出字符。

```
Console {
  log: [Function: bound consoleCall],
  info: [Function: bound consoleCall],
  warn: [Function: bound consoleCall],
  error: [Function: bound consoleCall],
  dir: [Function: bound consoleCall],
  time: [Function: bound consoleCall],
  timeEnd: [Function: bound consoleCall],
  trace: [Function: bound consoleCall],
  assert: [Function: bound consoleCall],
  clear: [Function: bound consoleCall],
  count: [Function: bound consoleCall],
  countReset: [Function: bound countReset],
  group: [Function: bound consoleCall],
  groupCollapsed: [Function: bound consoleC
  groupEnd: [Function: bound consoleCall],
  Console: [Function: Console],
  debug: [Function: debug],
  dirxml: [Function: dirxml],
  table: [Function: table],
  markTimeline: [Function: markTimeline],
  profile: [Function: profile],
  profileEnd: [Function: profileEnd],
  timeline: [Function: timeline],
  timelineEnd: [Function: timelineEnd],
  timeStamp: [Function: timeStamp],
  context: [Function: context],
  [Symbol(counts)]: Map {} }
```

常用工具 util

核心模块 `Util` 提供常用函数集合。

```
buffer: [Function]
buffer: [Function]
error: [Function]
query: [Function]
utf8decoder: [Function]
utf8decoder: [Function]
browserify: [Function]
log: [Function]
isPrimitive: [Function]
isFunction: [Function]
isError: [Function]
isDate: [Function]
isObject: [Function]
isRegExp: [Function]
isUndefined: [Function]
isSymbol: [Function]
isString: [Function]
isNumber: [Function]
isNullOrUndefined: [Function]
isNil: [Function]
isBuffer: [Function]
isBoolean: [Function]
isArray: [Function]
```

- `Util.inherits(constructor, superConstructor)` 实实现对象间原型继承的函数
- `Util.inspect(object,[showHidden],[depth],[colors])`: 将一个任意对象转换为字符串，用于调试和错误输出。

事件驱动 events

最重要的 module

事件发射器

仅一个对象: `events.EventEmitter`, `EventEmitter` 核心是事件发射与事件监听功能的封装。

`EventEmitter` 的每个事件有一个事件名和若干参数组成。

支持若干事件监听器,当事件发射时,注册的事件监听器依次被调用。

事件参数作为回调函数参数传递。

- `EventEmitter.on(event, listener)`: 为 `event` 事件注册一个事件监听器,回调 `listener`, 其参数是 `emit` 发射事件所传递的参数。
- `EventEmitter.Emit(event, [argv1], [argv2], [...])`: 发射 `event` 事件, 传递若干参数。
- `EventEmitter.once(event, listener)`: 注册单次事件监听器, 即监听器最多触发一次, 触发后立即解除。
- `EventEmitter.removeListener(event, listener)`: 移除已经注册的指定监听器。
- `EventEmitter.removeAllListeners([event])`: 如果指定事件, 则移除指定事件的所有监听器。

```
var event = require("events")
var emitter = new event.EventEmitter(); //获取EventEmitter对象
emitter.on("myEvent",function(a,b){ //注册监听
    console.log("myEvent",a,b);
})
emitter.once("myEvent",function(a,b,c){ //注册单次监听
    console.log("once event listener",a,b,c);
})
emitter.on("myEvent2",function(a,b,c){
    console.log(a,b,c);
})
emitter.emit("myEvent",1,2,3,4); //发射事件
```

Error 事件

`EventEmitter` 定义的特殊事件 `error`, 遇到错误是需要发射 `error` 事件。

`Error` 被发射时, `EventEmitter` 规定如果没有响应的监听器, `node`

把它当作异常，并推出程序并打印调用栈。一般为其设置监听器，避免程序中断。

```
var event = require("events")
var emitter = new event.EventEmitter();
//emitter.emit("error") //报错，程序终止

//为error注册监听后，不报错程序正常。
emitter.on("error", function(){
    console.log("An Error Has Encountered");
})
emitter.emit("error")
```

继承 EventEmitter

大多数时候不直接使用 EventEmitter，而是在对象中继承它。只要支持事件响应的核心模块都是 EventEmitter 子类，如 fs、net、http 等。

文件系统 fs

提供了文件的读取、写入、更名、删除、遍历目录、链接等 POSIX 文件系统操作。

与其他 module 不同的是，fs 提供同步和异步两个版本。

● Fs.readFile

(filename,[encoding],[callback(err,data)]): 读取文件，必选参数 filename。回调函数中，err 是 bool 表示是否有错误，data 是内容。如果指定了 encoding 则 data 是字符串，否则是

buffer 表示的二进制数据。

```
var fs = require("fs")
fs.readFile("./myFile.txt", "utf-8", function(err, data) {
  if(err) {
    console.error("An Error Has Encountered");
  }
  else{
    console.log(data);
  }
})
```

- `fs.readFileSync` 同步读取。如果有错误发生，`fs` 将会抛出异常，你需要使用 `try` 和 `catch` 捕捉并处理异常。
- `fs.open(path, flags, [mode], [callback(err, fd)])` 。它是 `open` 函数的封装，与 C 的 `fopen` 类似。

■ `path`: 必选参数。

■ `flags`: 必选参数，打开模式。

- `r` : 以读取模式打开文件。
- `r+` : 以读写模式打开文件。
- `w` : 以写入模式打开文件，如果文件不存在则创建。
- `w+` : 以读写模式打开文件，如果文件不存在则创建。
- `a` : 以追加模式打开文件，如果文件不存在则创建。
- `a+` : 以读取追加模式打开文件，如果文件不存在则创建。

■ `fd`: 文件描述符。

- `fs.read (fd, buffer, offset, length, position, [callback(err, bytesRead, buffer)])`: 比 `readFile` 更加底层的 `read` 函数，从指定的文件描述符 `fd` 中读取数据并写入 `buffer` 指向的缓冲区对象。`offset` 是 `buffer` 的写入偏移量。`length` 是要从文件中读取的字节数。`position` 是文件读取的

起始 位置, 如果 `position` 的值为 `null`, 则会从当前文件指针的位置读取。回调函数传递 `bytesRead` 和 `buffer`, 分别表示读取的字节数和缓冲区对象。

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

fs.open('content.txt', 'r', function(err, fd) {
  if (err) {
    console.error(err);
    return;
  }

  var buf = new Buffer(8);
  fs.read(fd, buf, 0, 8, null, function(err, bytesRead, buffer) {
    if (err) {
      console.error(err);
      return;
    }

    console.log('bytesRead: ' + bytesRead);
    console.log(buffer);
  })
});
```

过于底层, 一般不要用, 易出错。

Fs 所有函数:

功 能	异步函数	同步函数
打开文件	<code>fs.open(path, flags, [mode], [callback(err, fd)])</code>	<code>fs.openSync(path, flags, [mode])</code>
关闭文件	<code>fs.close(fd, [callback(err)])</code>	<code>fs.closeSync(fd)</code>

读取文件(文件描述符)	<code>fs.read(fd,buffer,offset,length,position,[callback(err, bytesRead, buffer)])</code>	<code>fs.readSync(fd, buffer, offset, length, position)</code>
写入文件(文件描述符)	<code>fs.write(fd,buffer,offset,length,position,[callback(err, bytesWritten, buffer)])</code>	<code>fs.writeSync(fd, buffer, offset, length, position)</code>
读取文件内容	<code>fs.readFile(filename,[encoding],[callback(err, data)])</code>	<code>fs.readFileSync(filename,[encoding])</code>
写入文件内容	<code>fs.writeFile(filename, data,[encoding],[callback(err)])</code>	<code>fs.writeFileSync(filename, data,[encoding])</code>
删除文件	<code>fs.unlink(path, [callback(err)])</code>	<code>fs.unlinkSync(path)</code>
创建目录	<code>fs.mkdir(path, [mode], [callback(err)])</code>	<code>fs.mkdirSync(path, [mode])</code>
删除目录	<code>fs.rmdir(path, [callback(err)])</code>	<code>fs.rmdirSync(path)</code>
读取目录	<code>fs.readdir(path, [callback(err, files)])</code>	<code>fs.readdirSync(path)</code>
获取真实路径	<code>fs.realpath(path, [callback(err, resolvedPath)])</code>	<code>fs.realpathSync(path)</code>
更名	<code>fs.rename(path1, path2, [callback(err)])</code>	<code>fs.renameSync(path1, path2)</code>
截断	<code>fs.truncate(fd, len, [callback(err)])</code>	<code>fs.truncateSync(fd, len)</code>
更改所有权	<code>fs.chown(path, uid, gid, [callback(err)])</code>	<code>fs.chownSync(path, uid, gid)</code>
更改所有权(文件描述符)	<code>fs.fchown(fd, uid, gid, [callback(err)])</code>	<code>fs.fchownSync(fd, uid, gid)</code>
更改所有权(不解析符号链接)	<code>fs.lchown(path, uid, gid, [callback(err)])</code>	<code>fs.lchownSync(path, uid, gid)</code>
更改权限	<code>fs.chmod(path, mode, [callback(err)])</code>	<code>fs.chmodSync(path, mode)</code>
更改权限(文件描述符)	<code>fs.fchmod(fd, mode, [callback(err)])</code>	<code>fs.fchmodSync(fd, mode)</code>
更改权限(不解析符号链接)	<code>fs.lchmod(path, mode, [callback(err)])</code>	<code>fs.lchmodSync(path, mode)</code>
获取文件信息	<code>fs.stat(path, [callback(err, stats)])</code>	<code>fs.statSync(path)</code>
获取文件信息(文件描述符)	<code>fs.fstat(fd, [callback(err, stats)])</code>	<code>fs.fstatSync(fd)</code>
获取文件信息(不解析符号链接)	<code>fs.lstat(path, [callback(err, stats)])</code>	<code>fs.lstatSync(path)</code>
创建硬链接	<code>fs.link(srcpath, dstpath, [callback(err)])</code>	<code>fs.linkSync(srcpath, dstpath)</code>
创建符号链接	<code>fs.symlink(linkdata, path, [type],[callback(err)])</code>	<code>fs.symlinkSync(linkdata, path,[type])</code>
读取链接	<code>fs.readlink(path, [callback(err, linkString)])</code>	<code>fs.readlinkSync(path)</code>
修改文件时间戳	<code>fs.utimes(path, atime, mtime, [callback(err)])</code>	<code>fs.utimesSync(path, atime, mtime)</code>
修改文件时间戳(文件描述符)	<code>fs.futimes(fd, atime, mtime, [callback(err)])</code>	<code>fs.futimesSync(fd, atime, mtime)</code>
同步磁盘缓存	<code>fs.fsync(fd, [callback(err)])</code>	<code>fs.fsyncSync(fd)</code>

HTTP 服务器与客户端

Node 的封装了一个高效的 HTTP 服务器和一个简易的 HTTP 客户

端。

`http.Server` 是基于事件的服务器，`http.request` 是客服端工具。

HTTP 服务器

HTTP.Server:

`Http.Server` 基于事件，每一个请求都被封装为独立的事件，开发者只需要编写相应的事件响应函数即能实现所有功能。

它继承自 `EventEmitter`，提供以下事件：

- **request**: 客服端请求时事件即被触发，提供两个参数：`req`、`res`，分别是 `http.ServerRequest`、`http.ServerResponse` 的实例。

```
//httpServer.js
http = require("http")
var server = new http.Server();
server.on("request",function(req,res){
//请求处理函数（请求对象，响应对象）
    res.writeHead(200,{"Content-Type":"text/html"}); //
    res.write("<h1>Http Server</h1>"); //写入响应体
    res.end("<h3>This is end</h3>"); //结束并发送
});
server.listen(3000); //监听的端口
console.log("Ths server is listening on port 3000");
```

由这个产生一个更好的变形：

`http.createServer([requestListener])`—创建一个服务器并添加监听函数。


```
//httpServer2.js
http = require("http")
http.createServer(function(req,res) {
    res.writeHead(200, {"Content-Type": "text/html"}); //响应头
    res.write("<h1>Http Server by 'createServer'</h1>"); //写入响应体
    res.end("<h3>This is end</h3>"); //结束并发送
}).listen(3000);

console.log("Ths server is listening on port 3000");
```

请求 ip: 127.0.0.1:端口

- **connection**: 当 TCP 被连接时, 该事件被触发, 提供 1 个参数: **socket**, 为 **net.Socket** 的实例。(我在 **server** 里面找不到)
- **close**: 服务器关闭时触发

http.ServerRequest:

http 请求的信息, 一般由 **http.Server** 的 **request** 事件发送。

HTTP 请求分为请求头和请求体

http.ServerRequest 提供三个事件用于控制请求体传输:

- **data**: 当请求体数据到来时, 该事件被触发。该事件提供一个参数 **chunk**, 表示接收到的数据。如果该事件没有被监听, 那么请求体将会被抛弃。该事件可能会被调用多次。
- **end**: 当请求体数据传输完成时, 该事件被触发, 此后将不会再有数据到来。
- **close**: 用户当前请求结束时, 该事件被触发。不同于 **end**, 如果用户强制终止了传输, 也还是调用 **close**。

表4-2 ServerRequest 的属性	
名 称	含 义
complete	客户端请求是否已经发送完成
httpVersion	HTTP 协议版本，通常是 1.0 或 1.1
method	HTTP 请求方法，如 GET、POST、PUT、DELETE 等
url	原始的请求路径，例如 /static/image/x.jpg 或 /user?name=byvoid
headers	HTTP 请求头
trailers	HTTP 请求尾（不常见）
connection	当前 HTTP 连接套接字，为 net.Socket 的实例
socket	connection 属性的别名
client	client 属性的别名

获取 GET 请求内容

获取 POST 请求内容

C5·使用 Node.js 进行 web 开发

Node 与 PHP、Perl、ASP、jsp 一样，目的都是实现动态网页，也就是服务器动态生成 HTML 页面。

MVC：软件设计模式

- M：模型，对象及其数据结构的实现，通常包含数据库操作。
- V：视图，用户界面，在网站中基本就是 HTML 组织结构。
- C：控制器。处理用户请求和数据流、复杂模型，将输出传递给视图层。

Node 与其他语言的一个区别在于，其原始封装程度较低，很多工作需要自己做，小到一个 post，大到一个 cookie。使用第三方框架可以适当减少开发工作量。

Express 框架

Node 官方推荐的唯一 web 框架，轻量级，只是进行了封装。

除了提供高层接口外，还有的功能：路由控制、模板解析支持、动态视图、用户会话、CSRF 保护、静态文件服务、错误控制器、访问日志、缓存、插件支持。

安装

```
npm install -g express
```

国外老是出错，换成国内镜像就可以了。

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

安装后 express 命令行不起作用，最新 express4.0 版本中将命令工具分家出来了（项目地址：<https://github.com/expressjs/generator>），所以我们还需要安装一个命令工具，命令如下：

```
npm install -g express-generator（任然使用淘宝镜像）
```

Usage: express [options] [dir]

Options:

--version	output the version number
-e, --ejs	add ejs engine support
--pug	add pug engine support
--hbs	add handlebars engine support
-H, --hogan	add hogan.js engine support
-v, --view <engine>	add view <engine> support (dust ejs hbs hjs jade pug twig vash) (defaults to jade)
--no-view	use static html instead of view engine
-c, --css <engine>	add stylesheet <engine> support (less stylus compass sass) (defaults to plain css)
--git	add .gitignore
-f, --force	force on non-empty directory
-h, --help	output usage information

Express 在初始化一个项目时需要指定模板引擎，模式支持 jade 和 ejs。这里使用 ejs。

Ejs: Embended JavaScript, 一个标签替换引擎，语法类似与 ASP、PHP 相似，易于学习。Express 的默认引擎目前是 jade，它颠覆了传统的模板引擎，制定了一套完整的语法用来生成 HTML 的每个标签结构，功能强大但不易学习。

建立工程

建立网站基本结构并启动：

A. Express -e microblog （使用 ejs）

在子目录中就会产生一系列文件：

```
microblog2
├── app.js
├── package.json
├── bin
│   └── www
├── public
│   ├── images
│   ├── javascripts
│   └── stylesheets
│       └── style.css
├── routes
│   ├── index.js
│   └── users.js
└── views
    ├── error.ejs
    └── index.ejs
```

(根据提示即可)

B. `cd microblog`

C. `npm install -`

`registry=https://registry.npm.taobao.org`

安装后目录下多出很多文件（依赖也已经自动安装了）

D. 启动服务器

`SET DEBUG=app:* & npm start`

浏览器: `http://localhost:3000`

`Ctrl + C` 关闭

工程结构

E. `App.js`:

工程入口

`Router` 是本地目录模块，作用：指定路径、组织返回内容

`App.set(key, value)`设置参数，可用的参数：

- ☐ `basepath`: 基础地址，通常用于 `res.redirect()` 跳转。
- ☐ `views`: 视图文件的目录，存放模板文件。
- ☐ `view engine`: 视图模板引擎。
- ☐ `view options`: 全局视图参数对象。
- ☐ `view cache`: 启用视图缓存。
- ☐ `case sensitive routes`: 路径区分大小写。
- ☐ `strict routing`: 严格路径，启用后不会忽略路径末尾的 “/”。
- ☐ `jsonp callback`: 开启透明的 JSONP 支持。

```

var createError = require('http-errors');
var express = require('express');
var path = require('path');
var cookieParser = require('cookie-parser');
var logger = require('morgan');
var indexRouter = require('./routes/index');
var usersRouter = require('./routes/users');
var app = express();

// view engine setup
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'ejs');
app.use(logger('dev'));
app.use(express.json());
app.use(express.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));
app.use('/', indexRouter);
app.use('/users', usersRouter);

// catch 404 and forward to error handler
app.use(function(req, res, next) {
  next(createError(404));
});

// error handler
app.use(function(err, req, res, next) {
  // set locals, only providing error in development
  res.locals.message = err.message;
  res.locals.error = req.app.get('env') === 'development' ? err : {};
  // render the error page
  res.status(err.status || 500);
  res.render('error');
});

module.exports = app;

```

Express 依赖于 connect, 提供大量的中间件, 可用通过 `app.use` 启用。

`app.use('/', indexRouter);` 指定了 `indexRouter` 处理请求“/”

F. Routes/index.js

路由文件, 相当于控制器, 组织展示的内容。

```

var express = require('express');
var router = express.Router();

/* GET home page. */
router.get('/', function(req, res, next) {
  res.render('index', { title: 'Express' });
});

module.exports = router;

```

也可以不使用路由文件, 直接在 `app.js` 里编写内容逻辑。

```
//下面两种方式,效果一样:
//1.使用app.use(),将控制交给路由,路由控制文件中使用router.get()
//app.use('/home',homeRouter);
//2.直接使用app.get()
app.get('/home', function(req, res, next) {
  res.render("home",{title:"EdwinXu's Home Page",description:"This is the
  home page created by edwinxu, how great am I?"});
});
```

G. Index.ejs

模板文件,即 routes/index.js 中调用的模板

```
<html>
  <head>
    <title><%= title %></title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>
  <body>
    <h1><%= title %></h1>
    <p>Welcome to <%= title %></p>
  </body>
</html>
```

H. Layout.ejs

模板文件不是孤立的,默认情况下所有的模板都继承自

layout.ejs

工作原理

浏览器访问 localhost: 3000, 发送请求"/", app.js 里

```
app.use('/', indexRouter); 和
```

```
var indexRouter = require('./routes/index');
var usersRouter = require('./routes/users');
```

说明对应的路由是 routes 目录下的 index.js,

Index.js 通过

```
router.get('/', function(req, res, next) {
  res.render('index', { title: 'Express' });
});
```

(render 方法渲染 HTML 模板)

调用视图模板 index，并且传递参数 title

Views 下的

```
<title><%= title %></title>
<link rel='stylesheet' href='/stylesheets/style.css' />
</head>
<body>
  <h1><%= title %></h1>
  <p>Welcome to <%= title %></p>
</body>
```

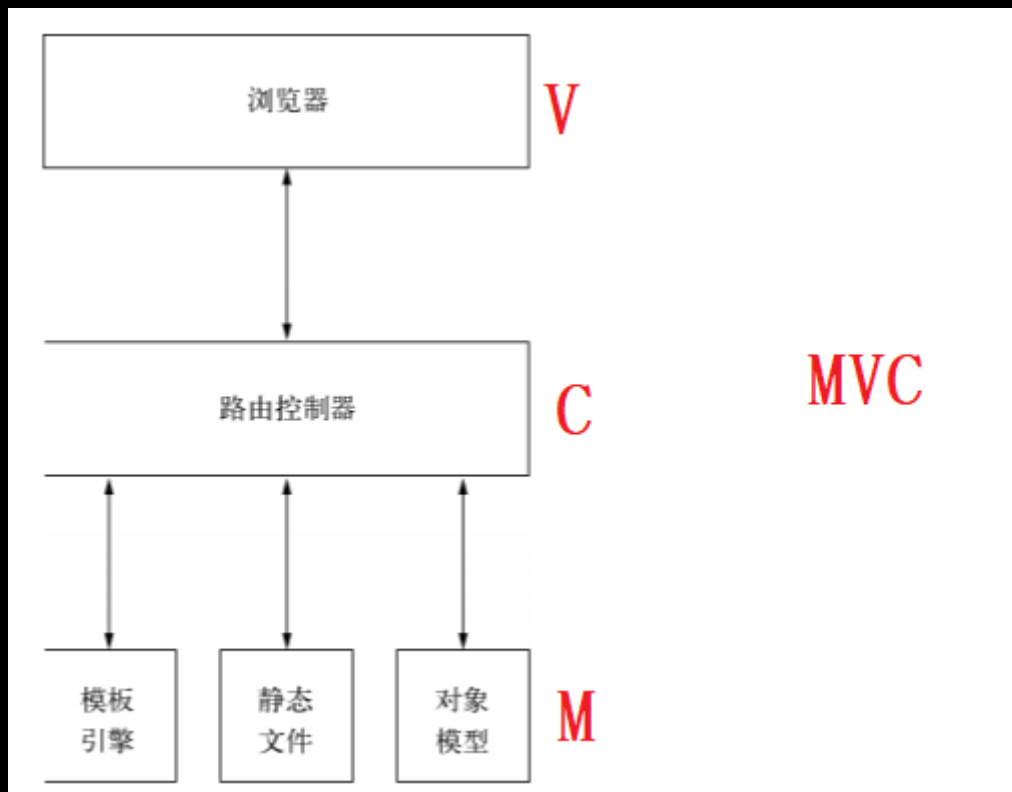
接收参数，生成完整 HTML，返回给浏览器。

浏览器分析发现需要获取 css 文件，再次请求服务器，app.js 里没有路由规定 css 的指派，但是 app.js 里

```
app.use(express.static(path.join(__dirname, 'public')));
```

配置了静态文件服务器，因此可以找到对应的 css，返回给浏览器。

有 Express 创建的网站架构：



路由规划

如果浏览器访问诸如 <http://localhost:3000/home> 这样的域名, 由于服务器没有配置路由, 也不是 `public` 下的静态文件(注意, `public` 下的文件是可以直接访问的: 如 `/public/images/a.png` 可以通过 <http://localhost:3000/images/pic.png> 访问), 所以 404。

注意:`router.get()`的参数 1, 即路径应该是“/”, 不能和 `app.js` 中 `app.use` 一样, 否则 404

A. 创建新的路由规划: `/home`

I. 在 `app.js` 里添加:

```
var homeRouter = require("./routers/home");
```

```
app.use('/home',homeRouter);
```

J. 在路由文件新增 routes/home.js

```
var express = require('express');
var router = express.Router();

/* GET home_2 page. */
router.get('/', function(req, res, next) {
  res.send("<h2>This is a home page created by edwinxu.<h2>");
});

module.exports = router;
```

这里不能使用/home
只能使用/

K. 如果需要 HTML 模板，在

Routes/home.js 使用 render: **render(模板名,{参数 json})**

```
var express = require('express');
var router = express.Router();

/* GET home_2 page. */
router.get('/', function(req, res, next) {

  res.render("home",{title:"EdwinXu's Home Page",description:"This is
  home page created by edwinxu, how great am I?});
  //res.send("<h2>This is a home page created by edwinxu.<h2>")
  //send和render不能同时使用。即模板和自定义不能同时使用。
});

module.exports = router;
```

新建模板 views/home.ejs

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>
  <body>
    <h1><%= title %></h1>
    <p>Welcome to <%= title %></p>
    <p><%= description %></p>
  </body>
</html>
```

B. 动态路径匹配:

如果页面是动态的，不同的访问生成不同页面，如/user/[username]

```
app.get('/user/:username', function(req, res) {
    res.send('user: ' + req.params.username);
});
```

C. REST 风格的路由规则

REST: Representational State Transfer 表征状态转移。一种基于 HTTP 协议的网络应用接口风格,充分利用 HTTP 的方法实现统一风格接口的服务。

HTTP 定义 8 种标准方法:

- ❑ GET: 请求获取指定资源。
- ❑ HEAD: 请求指定资源的响应头。
- ❑ POST: 向指定资源提交数据。
- ❑ PUT: 请求服务器存储一个资源。
- ❑ DELETE: 请求服务器删除指定资源。
- ❑ TRACE: 回显服务器收到的请求,主要用于测试或诊断。
- ❑ CONNECT: HTTP/1.1 协议中预留给能够将连接改为管道方式的代理服务器。
- ❑ OPTIONS: 返回服务器支持的HTTP请求方法。

安全: 请求不对资源造成影响
幂等: 多次请求和一次请求是一样的

表5-2 REST风格HTTP 请求的特点

请求方式		安 全	幂 等
GET	获取	是	是
POST	新增	否	否
PUT	更新	否	是
DELETE	删除	否	是

表5-3 Express 支持的 HTTP 请求的绑定函数

请求方式	绑定函数
GET	<code>app.get(path, callback)</code>
POST	<code>app.post(path, callback)</code>
PUT	<code>app.put(path, callback)</code>
DELETE	<code>app.delete(path, callback)</code>
PATCH ^①	<code>app.patch(path, callback)</code>
TRACE	<code>app.trace(path, callback)</code>
CONNECT	<code>app.connect(path, callback)</code>
OPTIONS	<code>app.options(path, callback)</code>
所有方法	<code>app.all(path, callback)</code>

如果使用了 `route`，那就是：先路由规划，路由在执行操作：

```
app.use('/', indexRouter);  
router.get('/', function(req, res, next) {
```

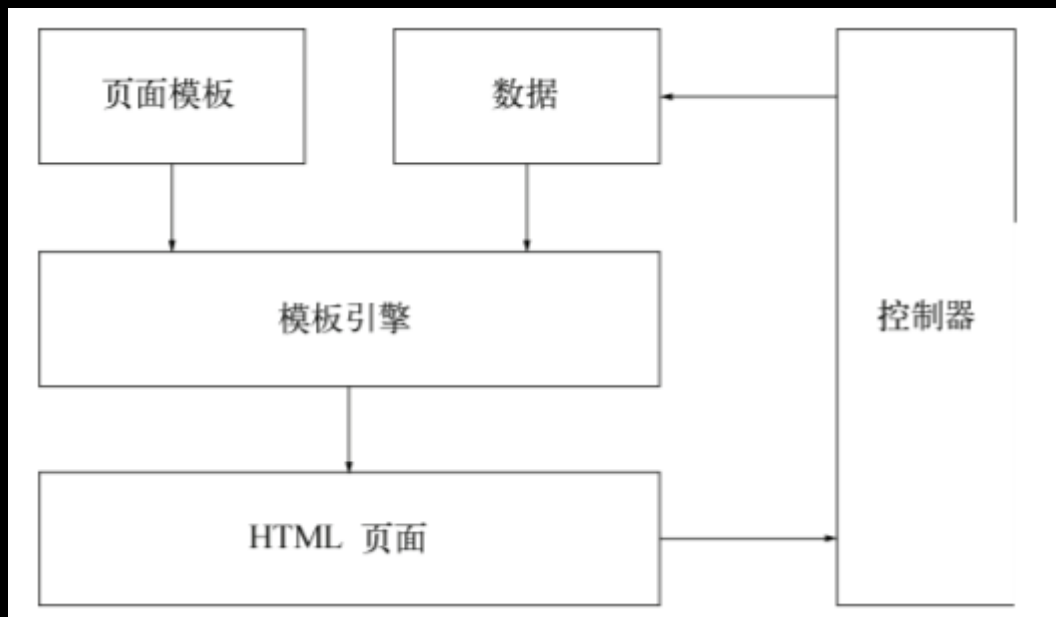
D. 控制权转移：

同一路径可以绑定多个响应函数，但是优先匹配最先的 `callback`，后面的会被匹配。使用控制权转移防止后面的被屏蔽。

使用第三个参数 `next` 并调用 `next()` 方法，即可将控制权转移到后一个

```
app.all('/user/:username', function(req, res, next) {  
  console.log('all methods captured');  
  next();  
});
```

模板引擎



模板引擎严格属于视图层，是 MVC 的一部分

L. 设置模板引擎：

在 app.js 中：

```
// view engine setup
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'ejs');
```

设置了模板引擎使用 ejs，模板路径是在 views 目录下

M. 使用模板引擎：

在 route 中：

通过 `res.render(moduleName, JsonParams)` 调用模板引擎

模板中的 `<%= argument >` 会被翻译为模板调用中的参数。

Ejs 标签系统有 3 中：

- `<% code %>` : JS 代码
- `<%= code >`: 显示替换
- `<%- code >`: 显示原始内容

页面布局

Layout.ejs 模块，默认所有的模板都继承自其现在的版本没有了

片段视图 partials

一个页面的片段，通常是重复内容

用 for 循环。让我们看一个例子，在 app.js 中新增以下内容：

```
app.get('/list', function(req, res) {
  res.render('list', {
    title: 'List',
    items: [1991, 'byvoid', 'express', 'Node.js']
  });
});
```

在 views 目录下新建 list.ejs，内容是：

```
<ul><%- partial('listitem', items) %></ul>
```

同时新建 listitem.ejs，内容是：

```
<li><%= listitem %></li>
```

访问 <http://localhost:3000/list>，可以在源代码中看到以下内容：

```
<!DOCTYPE html>
<html>
  <head>
    <title>List</title>
    <link rel='stylesheet' href='/stylesheets/style.css' />
  </head>
  <body>
    <ul><li>1991</li><li>byvoid</li><li>express</li><li>Node.js</li></ul>
  </body>
</html>
```

partial 是一个可以在视图中使用函数，它接受两个参数，第一个是片段视图的名称，第二个可以是一个对象或一个数组，如果是一个对象，那么片段视图中上下文变量引用的就是这个对象；如果是一个数组，那么其中每个元素依次被迭代应用到片段视图。片段视图中上下文变量名就是视图文件名，例如上面的 'listitem'。

视图助手

Partial 就是一个视图助手

分 2 类:

- 静态: 可以是任何类型的对象, 包括接受任意参数的函数, 但访问到的对象必须是与用户请求无关的。

通过 `app.helper(Object o)` 注册, 接受一个对象 `o`, `o` 的每个属性名称为视图助手的名称, 属性值对应视图助手的值。

- 动态: 只能是一个函数, 这个函数不能接受参数, 但可以访问 `req` 和 `res` 对象。

通过 `app.dynamicHelpers()` 注册, 参数同静态, 但是每个属性的值必须是一个函数, 该函数提供 `req`, `res`。

视图助手的本质是给所有的视图注册了全局变量, 因此无需每次在调用模板引擎时传递数据对象。

Pm2

使用 npm 启动 CVM 上的 node 项目

一旦断开连接, node 机会关闭, 于是使用另一种包管理工具——pm2

1. 进入特权模式

2. `cnpm install -g pm2`

3. run : `pm2 start app.js` or `pm2 start bin/www`

pm2 是一个**进程管理工具**,可以用它来管理你的 **node** 进程, 并查看 node 进程的状态, 当然也支持性能监控, 进程守护, 负载均衡等功能

启动进程/应用 `pm2 start bin/www` 或 `pm2 start app.js`

重命名进程/应用 `pm2 start app.js --name wb123`

添加进程/应用 `watch pm2 start bin/www --watch`

结束进程/应用 `pm2 stop www`

结束所有进程/应用 `pm2 stop all`

删除进程/应用 `pm2 delete www`

删除所有进程/应用 `pm2 delete all`

列出所有进程/应用 `pm2 list`

查看某个进程/应用具体情况 `pm2 describe www`

查看进程/应用的资源消耗情况 `pm2 monit`

查看 pm2 的日志 `pm2 logs`

若要查看某个进程/应用的日志,使用 `pm2 logs www`

重新启动进程/应用 `pm2 restart www`

重新启动所有进程/应用 `pm2 restart all`

cnpm

N and nvm

Demo: Microblog

功能分析

注册、登录、个人信息、信息发表、评论、转发、关注

视频学习

JS 的运行环境是 JS 引擎，而不是浏览器。

Node.js 是一个基于 Chrome JavaScript 运行时建立的平台，用于方便地搭建响应速度快、易于扩展的网络应用。

1. 请求一个HTTP地址 (封装一个请求报文)

浏览器的最大作用就是将一个URL地址封装成一个请求报文

2. 解析服务器给回来的响应报文 (内容有可能不一样)

html => 渲染HTML

css => 渲染CSS

image => 渲染

js => 解释 (执行) JS

浏览器中的 JavaScript 可以做什么？

- + 操作DOM (对DOM的增删改、注册事件)
- + AJAX/跨域
- + BOM (页面跳转、历史记录、console.log()、alert())
- + ECMAScript

白丽哲
34851

- 浏览器中的 JavaScript 不可以做什么？

- + 文件操作 (文件和文件夹的CRUD)
- + 没有办法操作系统信息
- + 由于运行环境特殊 (我们写的代码是在不认识的人的浏览器中运行)

在开发人员能力相同的情况下编程语言的能力取决于什么？

- + 语言本身？ -
- + 语言本身只是提供定义变量，定义函数，定义类型，流程控制，循环结构之类的操作
- + 取决于运行该语言的平台 (环境)
- + 对于JS来说，我们常说的JS实际是ES，大部分能力都是由浏览器的执行引擎决定
- + BOM和DOM可以说是浏览器开放出来的接口
- + 比如：Cordova中提供JS调用摄像头，操作本地文件的API

+ Java既是语言也是平台

+ Java运行在Java虚拟机 (跨操作系统)

+ PHP既是语言也是平台 (跨操作系统)

+ C#语言平台：.NET Framework (Windows)

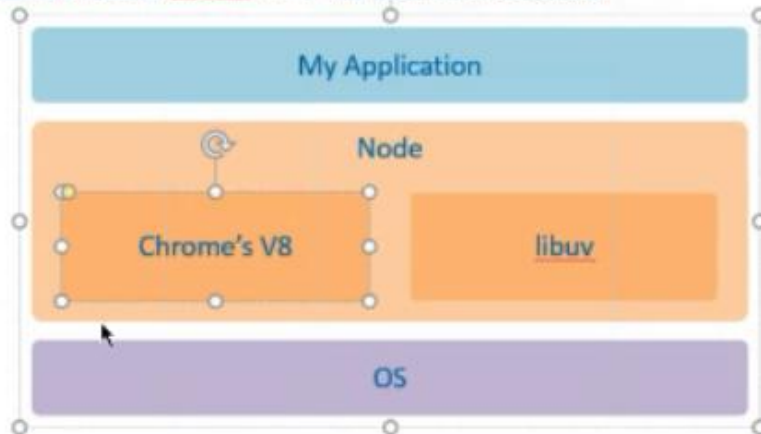
+ C#可以运行在MONO这样的平台

+ 因为有人需要将C#运行在Linux平台，所有出现了MONO

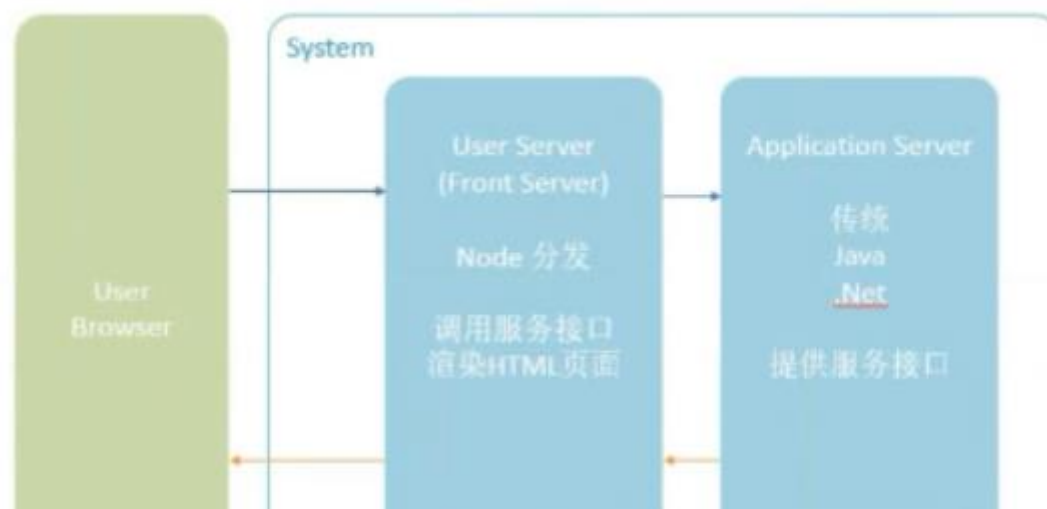
语言的能力取决于平台

Node 的实现

- Node 内部采用 Google Chrome 的 V8 引擎，作为 JavaScript 语言解释器；
- 通过自行开发的 libuv 库，调用操作系统资源。



Node 在 Web 中的用途



Node 是一个 JavaScript 的运行环境（平台），不是一门语言，也不是 JavaScript 的框架；

Node 的实现结构；

Node 可以用来开发服务端应用程序，Web 系统；

基于 Node 的前端工具集

NVM: node version manager : node 版本管理。

NVM工具的使用

Node Version Manager (Node版本管理工具)

由于以后的开发工作可能会在多个Node版本中测试，而且Node的版本也比较多，所以需要这么款工具来管理

环境变量就是操作系统提供的系统级别用于存储变量的地方

系统变量和用户变量

环境变量的变量名是不区分大小写的

特殊值：

- PATH 变量
- 只要添加到 PATH 变量中的路径，都可以在任何目录下搜索

箭头函数-匿名函数：

```

7 var log = function (message) {
8   process.stdout.write(message+'\n');
9 };|
0
1
2 var log2 = (message) => {
3   process.stdout.write(message+'\n');
4 };|

```

一个参数可以省略括号，否则必须要。

模板字符串：

```

var msg = 'hello'

var a = 1;

// 模版字符串
process.stdout.write(`
  ${msg} world ${a}
`);

```

使用的是``，这个可以任意换行，注意不是单引号。

``支持模板字符串：`\${moduleStr}otherStr`。

‘清空’控制台：

```
Process.stdout.write('\033[2J');
```

```
Procee.stdout.write('\033[0f');
```

字符图生成：

The logo for DeGraeve.com, featuring the text "DeGraeve.com" in a white, sans-serif font on a dark blue rectangular background.

代码计时器：

```
Console.time("Name")  
  
//some code  
  
Console.timeEnd("SameName")
```

查询存在性

`Object.keys(arr).indexOf(input)`：如果 `arr` 中不存在 `input` 则返回 -1

存在就返回下标。

`Arr` 可以是数组、json 等。

异常

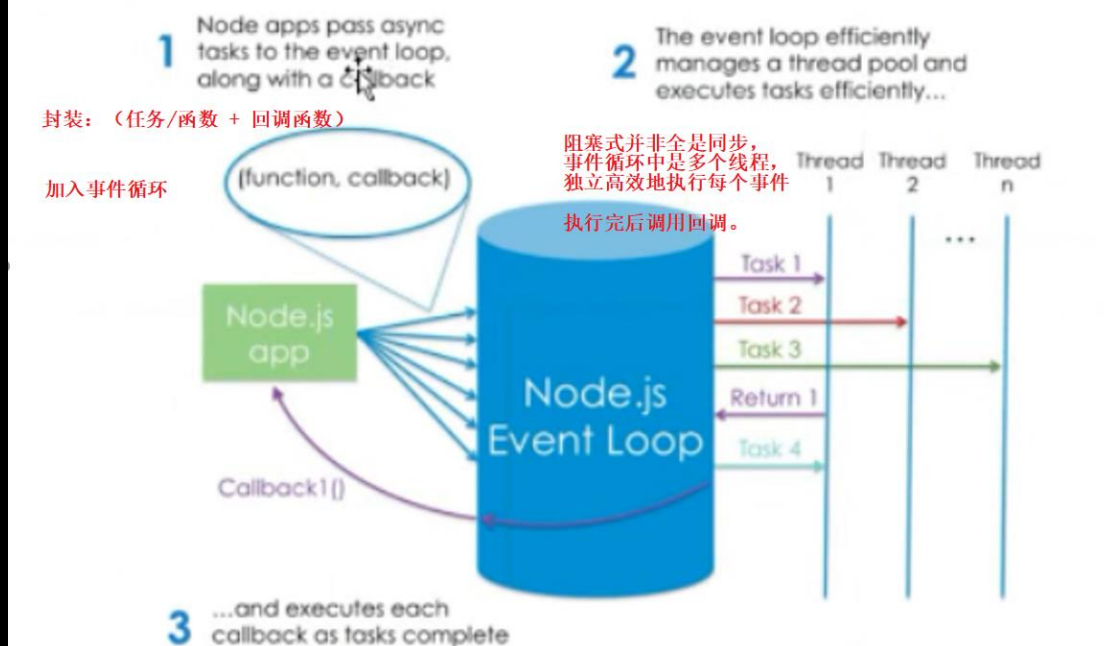
```
New Error("description")
```

回调定义

如果一个函数的参数是回调,那么直接把参数 `callback` 当成函数名, 直接执行 `callback(任意数目参数)`


```
function func(arr ,callback) {  
    var L = arr.length;  
    if(L==0) {  
        callback(new Error("err"));  
    }  
    else if(L===1) {  
        callback(arr[0])  
    }  
    else if (L==2) {  
        callback(arr[0],arr[1])  
    }  
    else {  
        callback(arr[0],arr[1],arr[2])  
    }  
}  
  
arr = [];  
func(arr,function(a) {  
    console.log(a)  
})
```

事件驱动和非阻塞机制



调试

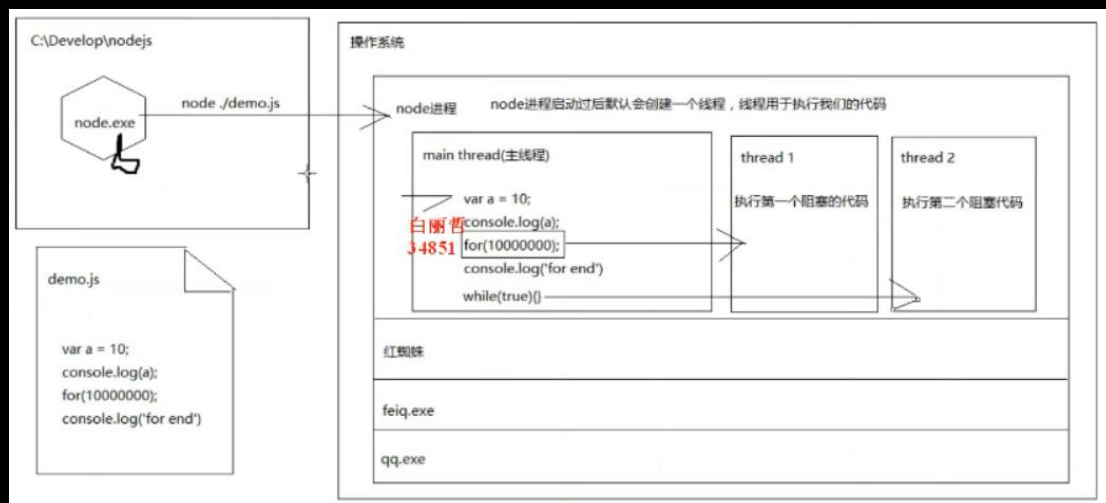
Node debug name.js

Node 采用 Chrome V8 引擎处理脚本，特点：单线程。

异步回调——异常优先

回调函数的第一个参数是异常，函数的最后一个是参数回调。

进程与线程：



什么原因让多线程没落

- 多线程都是假的，因为只有一个 CPU（单核）
- 线程之间共享某些数据，同步某个状态都很麻烦
- 更致命的是：
 - 创建线程耗费
 - 线程数量有限
 - CPU 在不同线程之间转换，有个上下文转换，这个转换非常耗时

Js_const

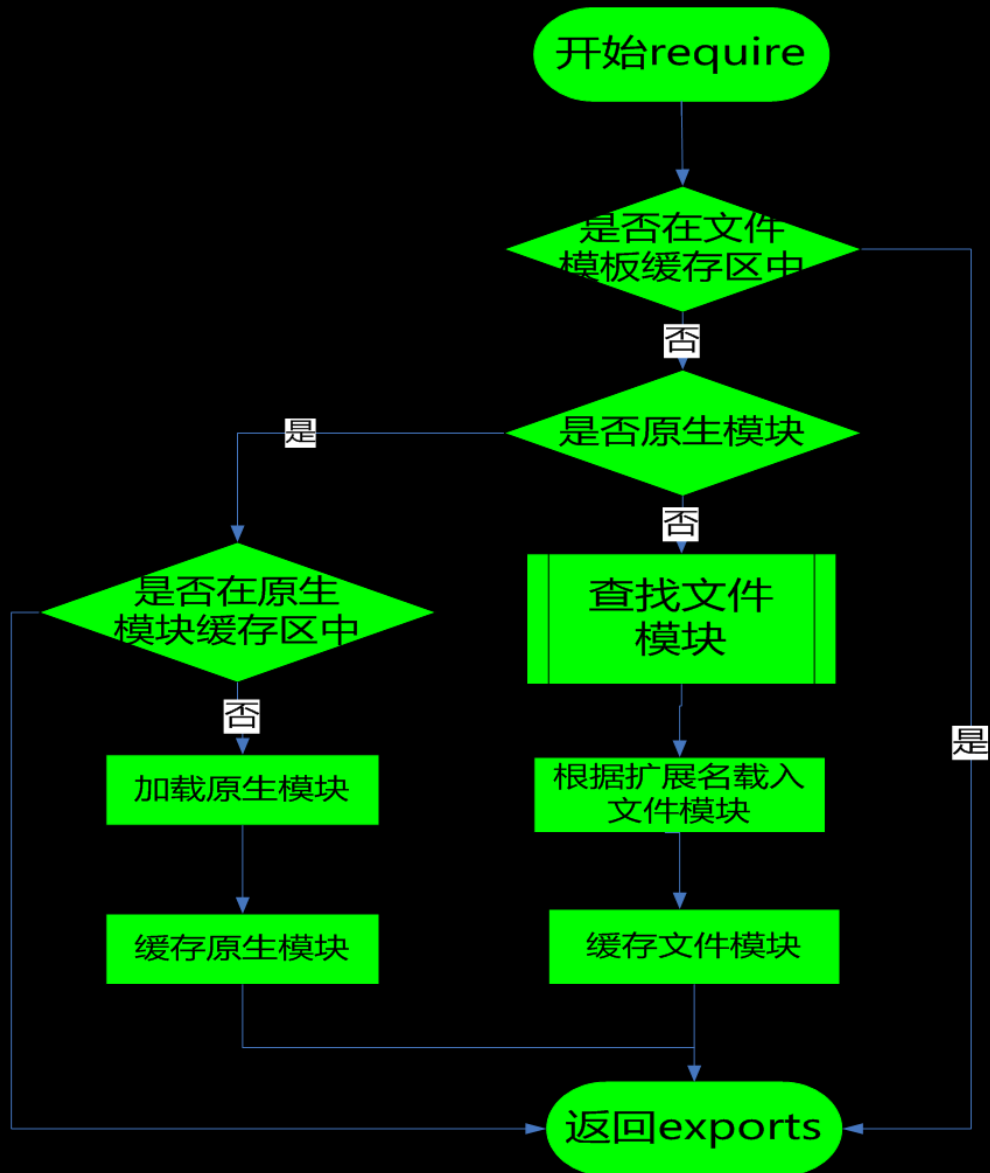
常量

注意：定义一个 json 之类的时，json 本身不能改变，但是内部却是可以改变的。

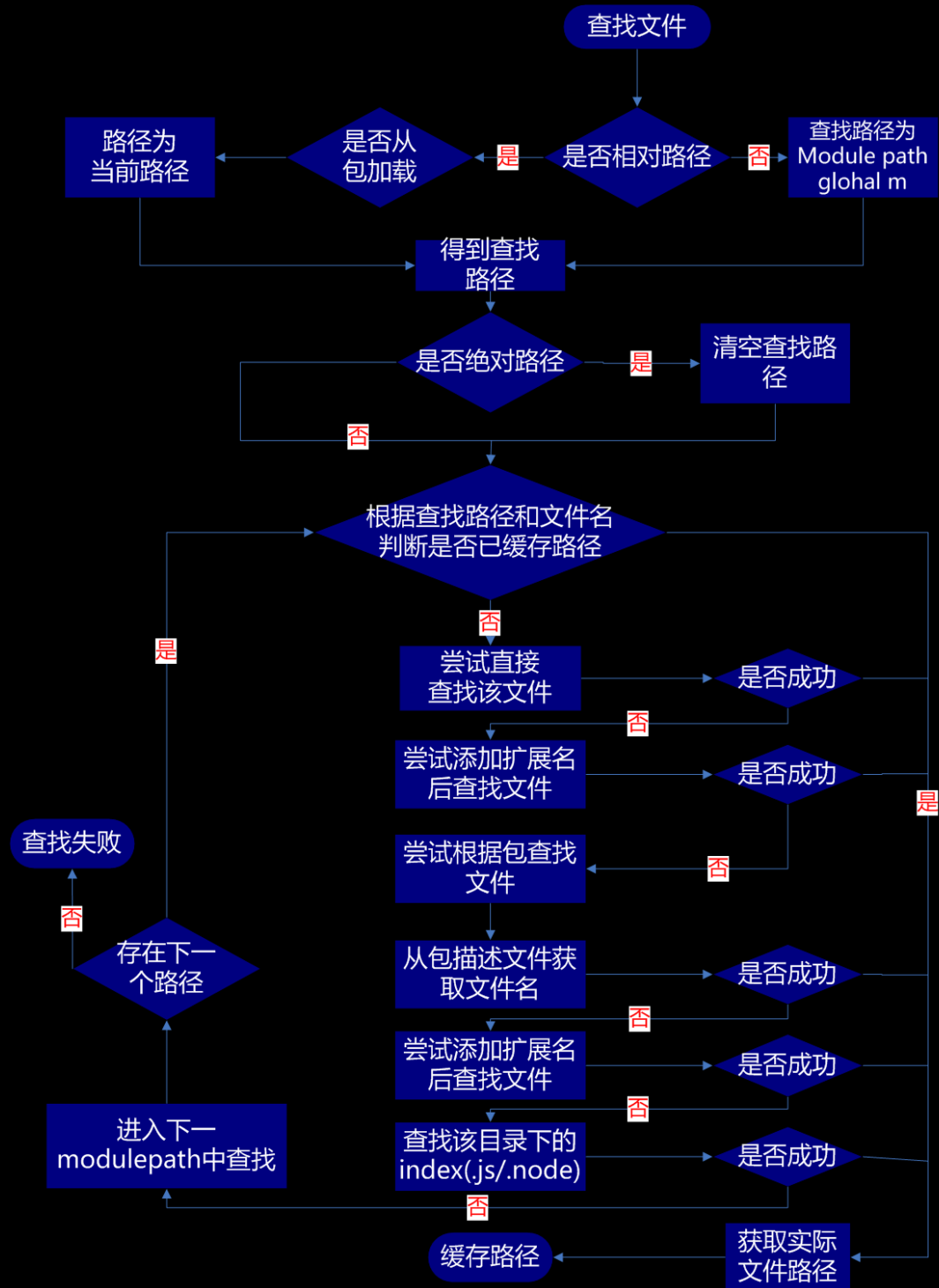
JS 定义量：var、let、const

一般 require 都是用 const

模块的查找：

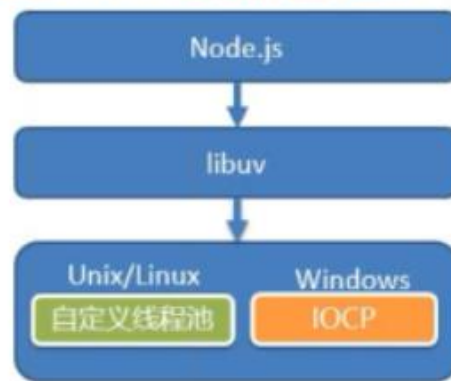


查找文件：

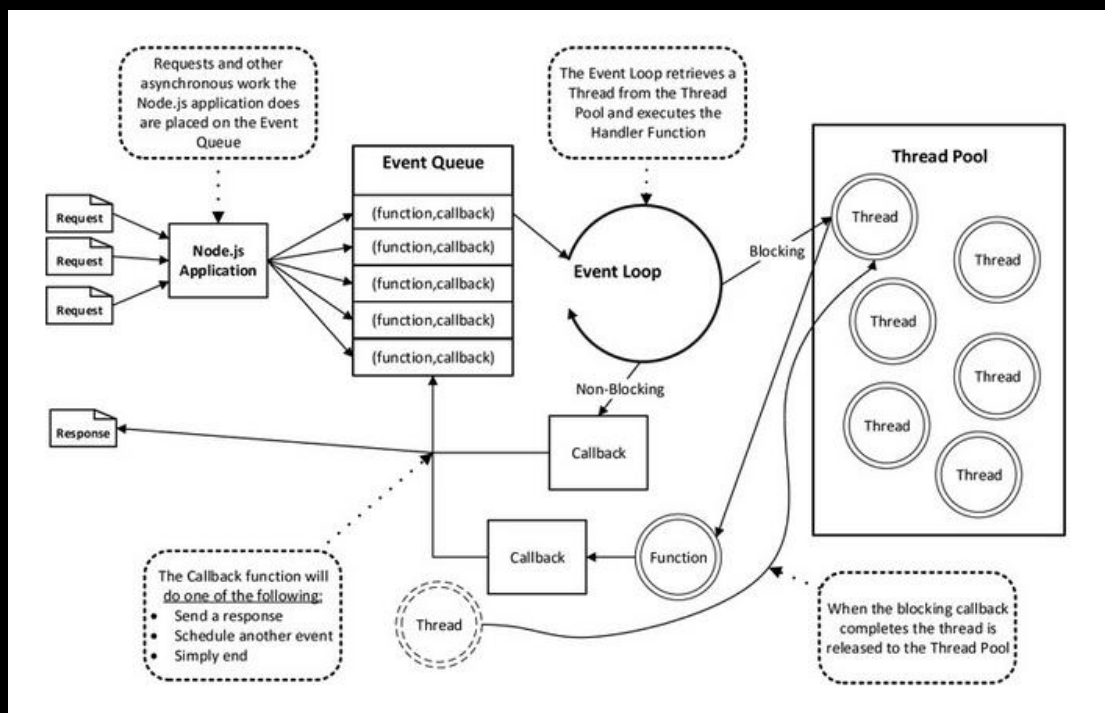


平台实现差异

- 由于 Windows 和 *nix 平台的差异，Node 提供了 libuv 作为抽象封装层，保证上层的 Node 与下层的自定义线程池及 IOCP 之间各自独立



Node 的底层原理：



分阻塞和非阻塞，非阻塞即（函数-回调）中函数执行无 io，执行完立即回调。而阻塞表示函数执行处会遇到 io 操作。非阻塞由主线程

自己执行，而阻塞将加入线程池，交由一个线程。

- Node 中将所有的阻塞操作交给了内部实现的线程池
- Node 本身主线程主要就是不断的往返调度

Node利用Javascript的特性，比如Continuation Passing Style(CPS)以及Event Loop，使得程序运行时表现优秀，CPS和Event Loop一套机制是用libuv库(libuv又根据OS的不同抽象了Unix下libev和Windos下ICOP)提供的。JS中处理IO业务的方法将回调函数当参数传递的编程风格就是CPS。而成为参数的回调函数的调用问题就得需要叫事件循环来控制。这一切就实现了Node的异步IO。

注意：Node 是单线程执行，如果代码中出现死循环之类的，将会卡死，后面的事件排在后面，不会执行。

服务器执行模型：

- 同步式：每次处理一个请求，处理完再下一个。
- 进程式：为每一个请求开启一个进程，当请求过多时，CPU 消耗极大。
- 线程式：为每一个请求开启一个线程。
 - 单线程模型
 - 多线程模型

Node 定时器：

- 超时定时：

使用 `nodejs` 中内置的 `setTimeout(callback, delayMilliseconds,[args])` 方法。当你调用 `setTimeout()` 时，回调函数在延时结束后才调用 `callback` 函数。返回一个定时器对象的 ID，可以在 `delayMilliseconds` 到期前的任何时候把这个 ID 传递给 `clearTimeout(timeoutId)` 来取消超时时间函数。

- 时间间隔：

使用 `nodejs` 内置的 `setInterval(callback,delayMilliseconds,[args])` 返回一个定时器对象的 id 在间隔到期前把这个 id 传递给 `clearInterval(intervalId)` 来取消超时时间函数。

模块

每个模块内部都是私有空间

每个模块是封闭的作用域，模块之间不会冲突。

导出：


```
- exports.name = value;  
- module.exports = { name: value };
```

Module 模块:

module 对象

- Node 内部提供一个 Module 构造函数。所有模块都是 Module 的实例，属性如下：
 - module.id 模块的识别符，通常是带有绝对路径的模块文件名。
 - module.filename 模块定义的文件的绝对路径。
 - module.loaded 返回一个布尔值，表示模块是否已经完成加载。
 - module.parent 返回一个对象，表示调用该模块的模块。
 - module.children 返回一个数组，表示该模块要用到的其他模块。
 - module.exports 表示模块对外输出的值。
- 载入一个模块就是构建一个 Module 实例。

加载:

require 加载文件规则

- 通过 ./ 或 ../ 开头：则按照相对路径从当前文件所在文件夹开始寻找模块；
 - `require('../file.js')`; => 上级目录下找 file.js 文件
- 通过 / 开头：则以系统根目录开始寻找模块；
 - `require('/Users/iceStone/Documents/file.js')`;
=> 以绝对路径的方式找

模块名称重复，系统模块优先级最高

模块的缓存

- 第一次加载某个模块时，Node 会缓存该模块。以后再次加载该模块，就直接从缓存取出该模块的 `module.exports` 属性（不会再次执行该模块）
- 如果需要多次执行模块中的代码，一般可以让模块暴露行为（函数）
- 模块的缓存可以通过 `require.cache` 拿到，同样也可以删除

所有的文件操作应该使用绝对地址，不然很容易出错误???

`__dirname+name`

Path 模块：

文件路径的操作

常用内置模块清单

- **path** : 处理文件路径。
- **fs** : 操作文件系统。
- **child_process** : 新建子进程。
- **util** : 提供一系列实用小工具。
- **http** : 提供HTTP服务器功能。
- **url** : 用于解析URL。
- **querystring** : 解析URL中的查询字符串。
- **crypto** : 提供加密和解密功能。

Path

```
// path.basename(p[, ext])  
// console.log(path.basename(temp));  
// // 获取文件名  
// console.log(path.basename(temp, 'rc'));  
// // 获取文件名without扩展名
```

```
// 获取不同操作系统中默认的路径分隔符 Windows是; Linux是:  
// console.log(path.delimiter);  
// 获取环境变量 I  
// console.log(process.env.PATH.split(path.delimiter));
```

```
// path.dirname(p)

// 获取目录名称
// console.log(path.dirname(temp));

// path.extname(p)

// 获取路径中的扩展名，包含.
// console.log(path.extname(temp));
```

```
// 拼合路径组成
// path.join(__dirname, '..', './temp', 'a', '../../1.txt');

// path.normalize(p)

// 常规化一个路径
// var a = path.normalize('C:/dev\\abc//cba///1.txt');
// console.log(a);
```

```
// path.relative(from, to)

// console.log(path.relative(__dirname,
// 'C:\\Users\\iceStone\\Desktop\\fed01\\day03\\lyrics\\血染的风采.lrc'));

// 获取to 相对于from的相对路径
```

```
76
77 // path.resolve([from ...], to)
78
79 白丽哲
34851
80 // console.log(path.resolve(__dirname, '..', './', './code'));
81
82 // 与join不同
83 // console.log(path.resolve(__dirname, 'c:/dev', './', './code'));
84
```

```
// path.sep

// 获取当前操作系统中默认用的路径成员分隔符 windows:\ linux:/
console.log(path.sep);
```

```
// path.win32  
  
// 允许在任意操作系统上使用windows的方式操作路径  
  
// path.posix  
|  
// 允许在任意操作系统上使用Linux的方式操作路径
```

缓冲区

数据间的转移，需要先读到内存，在写到目的地。

不能一次全部读到内存，所以需要缓冲区。特别是大文件。

Node 中使用一个二进制缓冲区：Buffer

Fs.ReadFile 使用了 buffer，默认即是读取的 buffer，但是是一次性读取。

```
var buf = new Buffer(bufSize) 单位字节
```

```
buf.toString("utf8",indexB,indexE)
```

toString()默认编码是 utf-8

文件编码

可以使用包：iconv-lite

```
// JSON.stringify 序列化
// JSON.parse 反序列化
```

模块中的 `__dirname` 不能乱用。

文件流

在 Node 核心模块 `fs` 中定义了一些与流相关的 API

- `fs.createReadStream()` => 得到一个 `ReadableStream`
- `fs.createWriteStream()` => 得到一个 `WritableStream`

```
//使用文件流来复制文件
const fs = require("fs")
const path = require("path");

//注意：地址必须使用双反斜线。
var readpath = "D:\\JavaWEB\\Node\\vedios\\04-2016-03-08\\videos\\01.avi";
var writepath = "D:\\JavaWEB\\Node\\vedios\\04-2016-03-08\\videos\\copyOf01.avi";
var reader = fs.createReadStream(readpath);
//注意：只是创建了一个文件读取流，没有开始读取。

var writer = fs.createWriteStream(writepath);

var times = 0;
var rate = 0;
var rate2 = 0;
//开始读取：
reader.on("data", (chunk) => { //不断读取，每次65535字节。
    times += 1;
    rate2 = rate;
    //是parse不是prase
    rate = parseInt(((times * 65535 / 118911968) * 100).toString().split(".")[0]);
    if (rate > rate2) { //是每个百分位只显示一次。
        console.log(rate + "%");
    }

    writer.on(chunk, (err) => {
        //写入
    })
})
})
```

Pipe 方式 copy:


```
//使用pipe来复制文件
const fs = require("fs")

var readpath = "D:\\JavaWEB\\Node\\vedios\\04-2
var writepath = "D:\\JavaWEB\\Node\\vedios\\04-
var reader = fs.createReadStream(readpath);
var writer = fs.createWriteStream(writepath);

reader.pipe(writer);
```

模块 progress

好像是进度条之类的

网络操作

一个端口只能被一个程序监听

<http://127.0.0.1>是本地回环地址。

连接 mysql

```
var express = require("express");
var app = express();
var mysql = require("mysql");

var conObj = {
  host: "localhost",
  port: "3306",
  user: "root",
  password: "xt222483",
  database: "world"
}

var Client = mysql.createConnection(conObj);
//调用connect方法判断连接是否成功
Client.connect(function(e){
  if(e){
    console.log(e);
  }
});

var sql = "select * from country;";

app.get("/mysql",function(req,res){
  Client.query(sql,function(err,result){ //执行query
    if(err){
      res.status(200).send("Error");
      return;
    }
    res.status(200).send(result);
  })
})

app.listen(3000,function(){
  console.log("listening on port 3000");
})
```


Node 语法

Slice:

切割数组: `arr.slice(indexBegin, indexTo)`

`Process.argv`: 参数

[“node 执行程序所在路径”, “当前脚本所在路径”, ...(所传参数)]

传的参数: 以空格分开

```
\JavaWEB\Node\code\module>node calc.js Hello Edwin
\nodejs\installPackage\node.exe,D:\JavaWEB\Node\code\module\calc.js,Hello,Edwin
```

暴露接口:

- ```
//
module.exports = { add, subtract, mutiply, divide };

```
- ```
module.exports = {
  add: add,
  subtract: subtract,
  mutiply: mutiply,
  divide: divide
};
```

获取路径:

```
//当前脚本路径
console.log(__dirname);
//文件路径
console.log(__filename);
```

Foreach:

```
Arr.forEach(callback)
```

```
[1,2,3,3,3,3].forEach((i)=>{console.log(i)})
```

Formidable

input 的 name 属性规定 input 元素的名称。

name 属性用于对提交到服务器后的表单数据进行标识，或者在客户端通过 JavaScript 引用表单数据。

只有设置了 name 属性的表单元素才能在提交表单时传递它们的值。

Formidable 可以用来处理客户端提交的表单

cnpm

因为谷歌安装插件是从国外服务器下载，受网络影响大，可能出现异常，如果谷歌的服务器在中国就好了，所以我们乐于分享的淘宝团队干了这事来自官网：“这是一个完整 npmjs.org 镜像，你可以用此代替官方版本（只读），同步频率目前为 10 分钟一次以保证尽量与官方服务同步”。

Node 爬虫

箭头函数(ES6):

没有返回值?

```
//x = x => x;  
/*  
function x(x){  
    return x;  
}  
*/  
x = (x,v) => {  
    console.log('今天天气bu cu')  
};  
console.log(x(20));
```

1 不粗 2 不 3 部 4

箭头函数

会改变this 指向

参数多了用括号装一下

箭头后面没有括号的话, 默认就是 return的值

http 模块

```
1  const http = require('http');  
2  
3  let req = http.request({  
4      'hostname': 'nodejs.cn',  
5      'path': '/download/'  
6  }, res=>{  
7      console.log(1)  
8  });  
9  
0  req.end();
```

```

1 const http = require('http');
2
3 let req = http.request({ 默认是post
4   'hostname': 'nodejs.cn',
5   'path': '/download/'
6 }, res=>{
7   var arr = [];
8   res.on('data', buffer=>{ 监听数据的传送
9     arr.push(buffer)
10   });
11
12   res.on('end', ()=>{ 当数据传输完了之后执行的回调
13     console.log(arr)
14   })

```

```

, res=>{
  var arr = [];
  var str = ''
  res.on('data', buffer=>{
    arr.push(buffer) 二进制
    str+=buffer 字符串
  });

```

```

res.on('end', ()=>{
  let b = Buffer.concat(arr);
  抓取图片

  fs.writeFile('asdakd.jpg', b, ()=>{
    console.log('成功了, 抓取成功')
  })
  //fs.writeFile('download.html', arr
  //console.log(arr, str)

```

URL

```
const url = require('url');
var str = 'http://www.clevaly.com/liyou/images/1.jpg';

var urlObj = url.parse(str);
console.log(urlObj)
```

```
function GetUrl(sUrl){
    var urlObj = url.parser(sUrl);
    if(urlObj.protocol == 'http:'){
        let http = require('http');
    }
    else{
        let http = require('https');
    }
}
```

兼容HTTP
HTTPS

```
function GetUrl(sUrl){
    var urlObj = url.parser(sUrl);
    if(urlObj.protocol == 'http:'){
        let http = require('http');
    }
    else{
        let http = require('https');
    }

    let req = http.request({
        'hostname':urlObj.hostname,
        'path':urlObj.path
    })
}
```

App4:

302: 资源是临时的, 即不想把资源让人爬取到, 使用了重定向技术。

这时候需要找到原始的那个真身,在请求头里面找 location 即为地址,
需要递归查找

所以淘宝上有些数据是重定向了很多次的地址

Gbk

```
文字转化成 utf-8  
需要一个 gbk 一个模块  
var html = toString('uft-8',data)
```

Jsdom

Segment

Echars

图表显示

1. 依赖包安装

安装 request 和 cheerio 依赖包

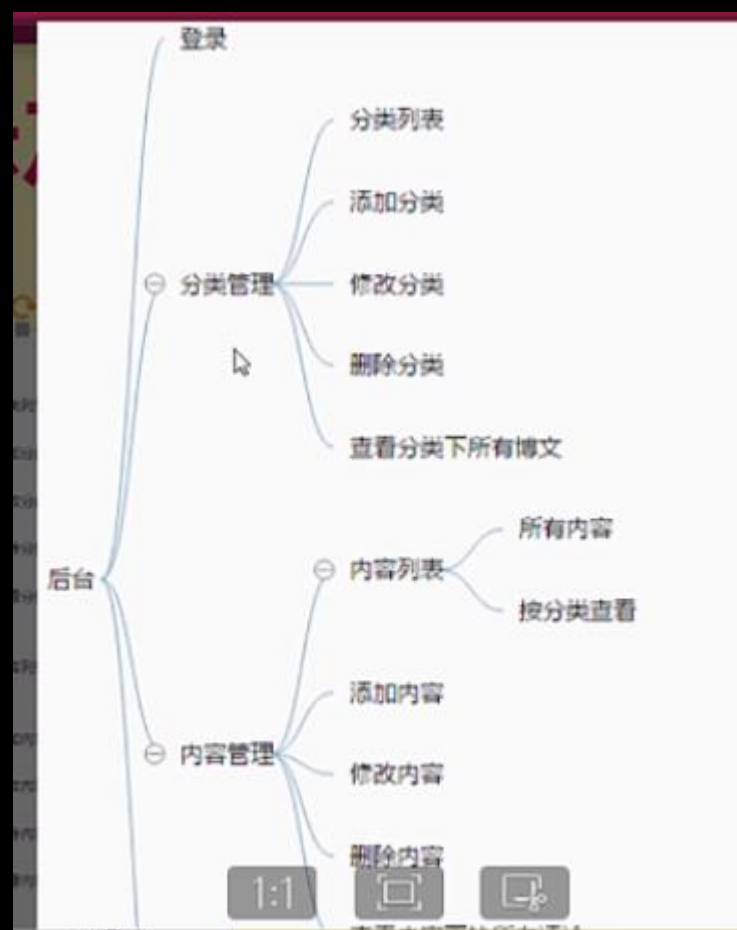
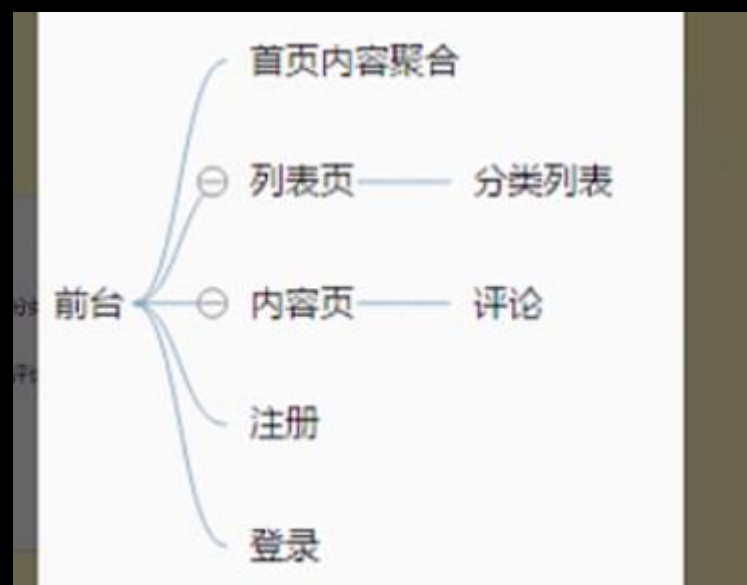
2.

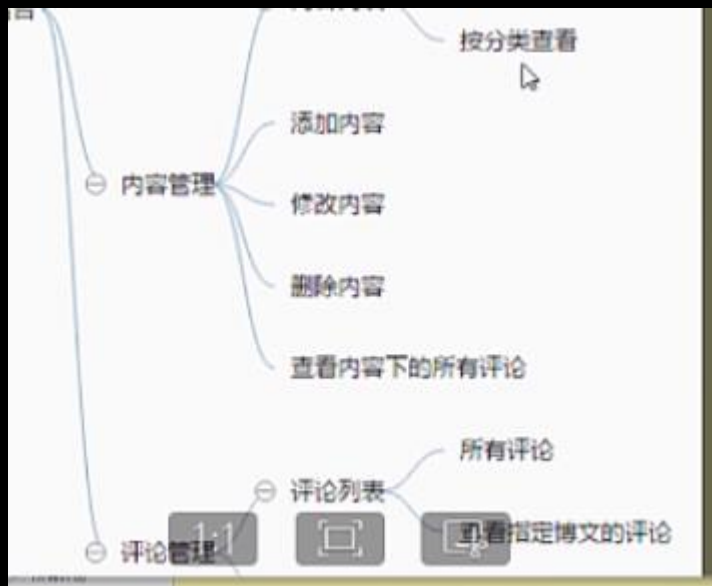
博客开发

这个视频和最新版的不一樣，我按她这个

安装模板：

- Cookies
- Swig:模板解析引擎
- Mongoose:操作 MongoDB
- Markdown
- Body-parser:解析 post 请求数据





新建项目：如下目录

目录结构

db	数据库存储目录
models	数据库模型文件目录
node_modules	node第三方模块目录
public	公共文件目录 (css、js、image.....)
routers	路由文件目录
schemas	数据库结构文件 (schema) 目录
views	模板视图文件目录
app.js	应用 (启动) 入口文件
package.json	

App.js:

```
//加载模板
var express = require('express');
//创建APP应用: http.createServer()
var app = express();

// 监听:
app.listen(8081);
```

● 路由绑定

通过app.get()或app.post()等方法可以把一个url路径和一个或N个函数进行绑定

app.get('/', function(req, res, next) {})

req : request对象 - 保存客户端请求相关的一些数据 - http.request

res : response对象 - 服务端输出对象, 提供了一些服务器端输出相关的一些方法 - http.response

next : 方法, 用于执行下一个和路径匹配的函数

● 内容输出

通过res.send(string)发送内容至客户端

使用模板

- 模板的使用

后端逻辑和页面表现分离 - 前后端分离

- 模板配置

```
var swig = require('swig');  
app.engine('html', swig.renderFile);
```

定义模板引擎，使用swig.renderFile方法解析后缀为html的文件

```
app.set('views', './views');
```

设置模板存放目录

```
app.set('view engine', 'html');
```

注册模板引擎

```
swig.setDefaults({ cache: false });
```

划分模块

- 模块划分

根据功能进行模块划分

前台模块

后台管理模块

API模块

使用app.use()进行模块划分

```
app.use('/admin', require('./router/admin'));
```

```
app.use('/api', require('./router/api'));
```

```
app.use('/', require('./router/main'));
```

前台路由+模板

- main模块

/	首页
/view	内容页

- api模块

/	首页
/register	用户注册
/login	用户登录
/comment	评论获取
/comment/post	评论提交

A circular icon with a dark background and two vertical white bars in the center, resembling a pause button.

后台路由+模板

● admin模块

	/	首页
用户管理	/user	用户列表
分类管理	/category	分类列表
	/category/add	分类添加
	/category/edit	分类修改
	/category/delete	分类删除
文章内容管理	/article	内容列表
	/article/add	内容添加
	/article/edit	内容修改
	/article/delete	内容删除
评论内容管理	/comment	评论列表
	/comment/delete	评论删除

如

错误排查

连接 mysql 数据库时，如果出现错误：

```
Error:  ER_NOT_SUPPORTED_AUTH_MODE:  Client  does  not  
support authentication protocol requested by server;  
consider upgrading MySQL client
```

解决：

进入 mysql：

输入：

```
alter    user    'root'@'localhost'    identified    with  
mysql_native_password by 'yourPassword';
```