**20180423**

* Node.js简介

① 浏览器大战

众所周知，在Netscape设计出JavaScript后短短几个月，JavaScript事实上已经是前端开发的唯一标准。后来，微软通过IE击败了Netscape后一统桌面；结果几年时间，浏览器毫无进步(2001年推出的古老IE 6至今仍然有人在使用!)。微软认为IE6已经非常完善而解散了IE6开发团队! 而Google却认为支持现代Web应用的新一代浏览器才刚刚起步，尤其是浏览器负责运行JavaScript的引擎性能还可提升10倍。

先是Mozilla借助已壮烈牺牲的Netscape遗产在2002年推出了Firefox浏览器；接着Apple于2003年在开源的KHTML浏览器基础上推出了WebKit内核的Safari浏览器，不过仅限于Mac平台。随后Google也看中了WebKit内核，基于WebKit内核推出了Chrome浏览器。

Chrome浏览器是跨Windows和Mac平台的，并且Google认为要运行现代Web应用，浏览器必须有一个性能非常强劲的JavaScript引擎，于是Google开发了一个名叫V8的高性能JavaScript引擎，以BSD许可证开源。

现代浏览器大战让微软的IE浏览器远远落后了，因为解散了最有经验、战斗力最强的浏览器团队，回过头再追赶却发现支持HTML5的WebKit已经成为手机端的标准了，IE浏览器从此与主流移动端设备绝缘。

② Node.js

有个叫Ryan Dahl的歪果仁，工作是用C/C++写高性能Web服务。对于高性能，异步IO、事件驱动是基本原则，但是用C/C++写太痛苦了。于是这位仁兄开始设想用高级语言开发Web服务。他发现很多语言虽然同时提供了同步IO和异步IO，但是开发人员一旦用了同步IO，就再也懒得写异步IO了。最终，Ryan选择了JavaScript。因为JavaScript是单线程执行，根本不能进行同步IO操作。所以JavaScript的这一缺陷导致了只能使用异步IO。

这位仁兄曾考虑自己写一个引擎，不过明智地放弃了，因为V8就是开源的JavaScript引擎。让Google去优化V8，自己改造一下拿来用，还不用付钱，这买卖很划算。

2009年Ryan正式推出了基于JavaScript语言和V8引擎的开源Web服务器项目，命名为Node.js。Node第一次把JavaScript带入到后端服务器开发，加上世界上已经有无数的JavaScript开发人员，所以Node一下子就火了。

在Node上运行的JavaScript相比其他后端开发语言，最大的优势是借助JavaScript天生的事件驱动机制和V8高性能引擎，容易编写高性能Web服务。其次，在Node环境下，通过模块化的JavaScript代码，加上函数式编程，无需考虑浏览器兼容性问题，直接使用最新的ECMAScript标准，可以完全满足工程上的需求。

③ io.js

Node.js是开源项目，虽然由社区推动，但幕后一直由Joyent公司资助。由于一些开发者对Joyent公司策略不满，于2014年从Node.js项目fork出了io.js项目，决定单独发展，但两者实际上是兼容的。然而分家后没多久，Joyent公司表示要和解，io.js项目又回归Node.js。

结果是要添加新特性先加到io.js，如果大家都很满意，就把新特性加入Node.js；于是io.js变成尝鲜版，而Node.js相当于线上稳定版。

* 安装Node.js

Node.js是在后端运行JavaScript代码，必须先安装Node环境。

目前Node.js的最新版本是9.11.1，官网推荐安装[8.11.1 LTS](https://nodejs.org/dist/v8.11.1/node-v8.11.1-x64.msi)。

在Windows安装必须选择全部组件，包括勾选Add to Path。

安装完成后，命令行输入node -v，如果安装正常，可以看到版本号。

命令行输入node，将进入Node.js交互环境。在交互环境可以输入任意JavaScript语句，回车后将得到输出结果。

连按两次Ctrl+C退出Node.js环境。

* npm

npm是Node.js的包管理工具(package manager)。

// 相当于Python的pip? 直接下载其他人开源的模块。

npm还可以根据依赖关系，把所有依赖的包都下载并管理。

npm在Node.js安装时顺便安装了；命令行输入npm -v，输出npm的版本号，说明npm已经正确安装了。

* 第一个Node程序

hello.js：

|  |
| --- |
| 'use strict';  console.log('hello world !'); |

第一行总是写'use strict';严格模式运行JavaScript代码，避免各种潜在陷阱。

命令行切到hello.js目录：

|  |
| --- |
| PS C:\Users\hikari\Desktop> node hello.js  hello world ! |

* Node交互模式和命令行模式

Node交互式环境会把每一行JavaScript代码的结果自动打印；但是命令行运行JavaScript文件却不会。

例如：在Node交互式环境下：

|  |
| --- |
| > 1+2+3;  6 |

但是写一个calc.js的文件，命令行模式下执行：

|  |
| --- |
| PS C:\Users\hikari\Desktop> node calc.js |

什么也没有输出，必须用console.log()才能输出结果。

输入node进入交互模式，相当于启动了Node解释器，每输入一行就执行一行；

运行node calc.js相当于启动了Node解释器，一次性把calc.js的代码执行了。

js文件就是用于写大段代码；交互模式一般用于验证部分代码。

* 使用严格模式

如果在JavaScript文件开头写上'use strict';，那么Node在执行该JavaScript时将使用严格模式。但是在服务器环境下，如果有很多JavaScript文件，每个文件都写上'use strict';很麻烦。可以给Nodejs传递--use\_strict参数开启严格模式：

|  |
| --- |
| node --use\_strict calc.js |

* Node集成开发环境

使用文本编辑器开发Node程序，最大的缺点是效率太低，运行Node程序还需在命令行单独敲命令。如果还需要调试程序，就更加麻烦了。

所以需要一个IDE集成开发环境，能在一个环境里编码、运行、调试，这样就可以大大提升开发效率。

Java集成开发环境有Eclipse，Intellij idea等；C#集成开发环境有Visual Studio，那么问题来了：Node.js的集成开发环境到底哪家强？

此处推荐[Visual Studio Code](http://code.visualstudio.com/)，微软出品，是Visual Studio发精简版，并且Visual Studio Code可以跨平台。

安装时推荐勾上：桌面快捷方式、资源管理器目录上下文菜单、添加到PATH

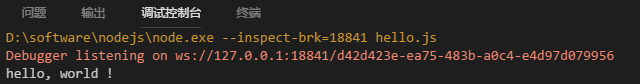
为VS Code创建工作目录，新建文件hello.js

|  |
| --- |
| let name = 'world';  let s = `hello, ${name} !`;  console.log(s); |

点击调试，发现没有配置，点击齿轮自动生成配置文件launch.json：

|  |
| --- |
| {  "version": "0.2.0",  "configurations": [  {  "type": "node",  "request": "launch",  "name": "启动程序",  "program": "${workspaceFolder}/hello.js" // 运行的js文件  }  ]  } |

点击开始调试。调试控制台输出结果：



**20180424**

* 模块

随着程序代码越写越多，在一个文件里代码越来越长，越来越不容易维护。

为了编写可维护的代码，把函数分组放到不同的js文件，在Node环境一个js文件就是一个模块(module)，很多语言都采用这种代码组织方式。

除了提高代码可维护性，模块还可以避免重复造轮子，直接引用Node内置模块或第三方模块就可以使用；可以避免函数名和变量名冲突，相同名字的函数和变量可以存在不同模块中，因此在编写模块时，不必考虑名字与其他模块冲突。

示例：

hello.js：

|  |
| --- |
| function greet(name) {  console.log(`hello, ${name} !`);  }  // 把greet()函数作为模块输出暴露出去, 其他模块就可以使用greet()  module.exports = greet; |

main.js：

|  |
| --- |
| let greet = require('./hello'); // 相对路径  let s = 'hikari';  greet(s); // hello, hikari ! |

引入的模块作为变量保存在greet变量中，就是在hello.js输出的greet()函数。所以main.js成功地引用了hello.js模块的greet()函数，然后可以直接使用了。

如果写成require('hello')，Node会依次在内置模块、全局模块和当前模块下查找hello.js，很可能会得到一个错误：Error: Cannot find module 'hello'

VSCode推荐使用ES6标准：

|  |
| --- |
| export default greet;  import greet from "./hello"; |

然后报错了…也就是不支持export和import?

* CommonJS规范

这种模块加载机制称为CommonJS规范。此规范下，每个js文件都是一个模块，它们内部各自使用的变量名和函数名都互不冲突。

要在模块中对外输出变量用：module.exports = variable;

输出的变量可以是任意对象、函数、数组等等。

要引入其他模块输出的对象用：let foo = require('other\_module');

引入的对象具体是什么，取决于引入模块输出的对象。

* 深入了解模块原理

在浏览器中大量使用全局变量不好。如果在a.js中使用了全局变量s，那么在b.js中也使用全局变量s，将造成冲突，b.js中对s赋值会改变a.js的运行逻辑。

JavaScript本身并没有一种模块机制来保证不同模块可以使用相同的变量名。

但由于JavaScript支持闭包，如果把一段JavaScript代码用函数包装起来，其所有全局变量就变成了函数内部的局部变量，从而支持不同模块同名变量。

Node.js加载了js后，把代码外面套一层匿名函数，包装一下变成：

|  |
| --- |
| (function () {  // js代码  })(); |

原来的全局变量变成了匿名函数内部的局部变量。如果Node.js继续加载其他模块，这些模块中定义的同名变量也互不干扰。

模块的输出module.exports实现：

|  |
| --- |
| // node在加载js文件前准备的module对象  let module = {  id: 'hello',  exports: {}  };  let load = function (module) {  // 读取的hello.js代码  function greet(name) {  console.log(`hello, ${name} !`);  }  module.exports = greet;  return module.exports;  };  let exported = load(module);  // 保存module  save(module, exported); |

变量module是Node在加载js文件前准备的一个变量，并将其传入加载函数，在hello.js中可以直接使用变量module原因在于它实际上是函数的一个参数。

通过把参数module传递给load()函数，hello.js就顺利地把一个变量传递给了Node执行环境，Node会把module变量保存到某个地方。

由于Node保存了所有导入的module，当用require()获取module时，Node找到对应的module，把这个module的exports变量返回，这样另一个模块就顺利拿到了模块的输出。

练习：编写hello.js，输出多个函数；编写main.js，引入hello模块，调用其函数。

hello.js：

|  |
| --- |
| let s = 'hello';  function greet(name) {  console.log(`${s}, ${name} !`);  }  function hi(name) {  console.log(`hi, ${name} !`);  }  function goodbye(name) {  console.log(`goodbye, ${name} !`);  }  module.exports = {  greet: greet,  hi: hi,  goodbye: goodbye  }; |

main.js：

|  |
| --- |
| const hello = require('./hello');  var s = 'hikari';  hello.greet(s); // hello, hikari !  hello.hi(s); // hi, hikari !  hello.goodbye(s); // goodbye, hikari ! |

**20180425**

* 基本模块

Node.js是运行在服务区端的JavaScript环境。服务器程序和浏览器程序相比最大的特点是没有浏览器的安全限制。服务器程序必须能接收网络请求、读写文件、处理二进制内容，所以Node.js内置的常用模块就是为了实现基本的服务器功能。这些模块在浏览器环境中无法执行，因为它们的底层代码是用C/C++在Node.js运行环境中实现的。

* Node.js常用对象

① global

JavaScript有且仅有一个全局对象，浏览器中是window对象。而在Node.js环境中也有唯一的全局对象global，其属性和方法也和浏览器的window不同。

② process

process也是Node.js提供的一个对象，代表当前Node.js进程。

|  |
| --- |
| // 获取js文件名和目录  console.log('current js file: ' + \_\_filename); // d:\hello\01 global.js  console.log('current js dir: ' + \_\_dirname); // d:\hello  // ver:v8.11.1; platform: win32; arch: x64  console.log(`ver:${process.version}; platform: ${process.platform}; arch: ${process.arch}`)  // process.argv 存储了命令行参数  console.log('arguments: ' + JSON.stringify(process.argv)); // arguments: ["D:\\software\\nodejs\\node.exe","d:\\hello\\01 global.js"]  // process.cwd() 返回当前工作目录  console.log('cwd: ' + process.cwd()); // cwd: d:\hello  // 切换当前工作目录  let d = '/private/tmp';  if (process.platform === 'win32') {  // 如果是Windows，切换到 C:\Windows\System32  d = 'C:\\Windows\\System32';  }  process.chdir(d);  console.log('cwd: ' + process.cwd()); // cwd: C:\Windows\System32 |

JavaScript是由事件驱动执行的单线程模型，Node.js也不例外。Node.js不断执行响应事件函数，直到没有任何响应事件函数可以执行时，Node.js退出。

如果想在下一次事件响应中执行代码，可以调用process.nextTick()：

|  |
| --- |
| // process.nextTick()将在下一轮事件循环中调用  process.nextTick(function () {  console.log('nextTick callback!');  });  console.log('nextTick was set!');  // nextTick was set! nextTick callback! |

说明传入process.nextTick()的函数不是立刻执行，而要等到下一次事件循环。

Node.js进程本身的事件就由process对象来处理。如果响应exit事件，就可以在程序即将退出时执行某个回调函数：

|  |
| --- |
| // 程序即将退出时的回调函数  process.on('exit', function (code) {  console.log('about to exit with code: ' + code);  }); // about to exit with code: 0 |

* 判断JavaScript执行环境

很多JavaScript代码既能在浏览器中执行，也能在Node环境执行。

有时程序本身需要判断自己到底在什么环境下执行，常用方式是根据浏览器和Node环境提供的全局变量名称来判断：

|  |
| --- |
| let env = typeof window === 'undefined' ? 'node.js' : 'browser';  console.log(`environment: ${env}`); // environment: node.js |

**20180427**

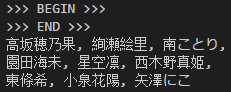
* fs模块

Node.js内置的fs模块是文件系统模块，负责读写文件。

和所有其它JavaScript模块不同的是fs模块同时提供了异步和同步的方法。

① 异步读文件

|  |
| --- |
| const fs = require('fs');  console.log('>>> BEGIN >>>')  fs.readFile('sample.in', 'utf-8', function (err, data) {  if (err) {  console.log(err);  } else {  console.log(data);  }  });  console.log('>>> END >>>'); |



异步读取时，回调函数接收err和data两个参数。正常读取时，err为null，data为读取到的字符串；读取发生错误时，err代表一个错误对象，data为undefined。这也是Node.js标准的回调函数：第一个参数为错误信息，第二个参数为结果。

当读取二进制文件时，不传入文件编码：

|  |
| --- |
| fs.readFile('sample.png', function (err, data) {  if (err) {  console.log(err);  } else {  console.log(Array.isArray(data)); // false  console.log(`${data.length} bytes`); // 73770 bytes  }  }); |

回调函数的data是返回一个Buffer对象。在Node.js中，Buffer对象是一个包含任意个字节的数组(注意和Array不同)，length是字节数。

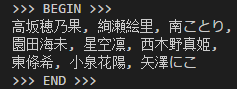
Buffer对象可以和String转换：

|  |
| --- |
| // Buffer --> String  let text = data.toString('utf-8');  console.log(text); // 乱码  // String --> Buffer  let buf = Buffer.from(text, 'utf-8');  console.log(`buf.length=${buf.length}; data.length=${data.length}`);  // buf.length=133830; data.length=73770, 两者大小都不一样? |

② 同步读文件

readFileSync()函数为同步读取文件函数，不接收回调函数，函数直接返回结果。

|  |
| --- |
| const fs = require('fs');  console.log('>>> BEGIN >>>');  let data = fs.readFileSync('sample.in', 'utf-8');  console.log(data);  console.log('>>> END >>>'); |

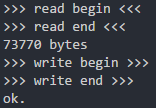


如果同步读取文件发生错误，可以使用try...catch...捕获

③ 写入文件

写入同样有异步的writeFile()和同步的writeFileSync()

|  |
| --- |
| const fs = require('fs');  *//异步写入文件*  function write(f, data) {  console.log('>>> write begin >>>');  fs.writeFile(f, data, function (err) {  if (err) {  console.log(err);  } else {  console.log('ok.');  }  });  console.log('>>> write end >>>');  }  *// 异步读取文件, 异步将data写入另一文件, 相当于复制*  function read\_write(f) {  console.log('>>> read begin <<<')  fs.readFile(f, function (err, data) {  if (err) {  console.log(err);  } else {  console.log(`${data.length} bytes`); *// 73770 bytes*  write('output.png', data);  }  });  console.log('>>> read end <<<');  }  read\_write('sample.png'); |



writeFile()的参数依次为文件名、数据和回调函数。如果传入的data是String，默认按UTF-8编码写入文本文件；如果data是Buffer，写入的是二进制文件。

回调函数只关心成功与否，只需要一个err参数。

同步写入，不需要回调函数，直接写入。

|  |
| --- |
| *// 同步写入文件*  let s = 'hello hikari\nhello world';  fs.writeFileSync('hello.out', s); |

④ stat

fs.stat()返回一个Stat对象，能获取文件或目录的详细信息：

|  |
| --- |
| const fs = require('fs');  fs.stat('sample.in', function (err, stat) {  if (err) {  console.log(err);  } else {  *// 是不是文件, 是不是目录?*  console.log('isFile: ' + stat.isFile());  console.log('isDirectory: ' + stat.isDirectory());  if (stat.isFile()) { *// 是文件, 打印文件大小, 创建日期, 修改日期*  console.log('size: ' + stat.size);  console.log('birth time: ' + stat.birthtime); *// date对象*  console.log('modified time: ' + stat.mtime); *// date对象*  }  }  }); |

结果：

|  |
| --- |
| isFile: true  isDirectory: false  size: 128  birth time: Fri Apr 27 2018 11:11:19 GMT+0800 (中国标准时间)  modified time: Fri Apr 27 2018 11:20:55 GMT+0800 (中国标准时间) |

stat()对应同步函数statSync()：

|  |
| --- |
| try {  let info = fs.statSync('sample.in');  console.log('birth time: ' + info.birthtime);  } catch (err) {  console.log(err);  } |

* 异步还是同步

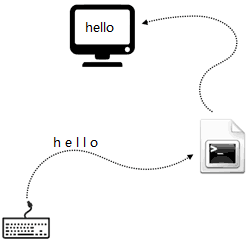
在fs模块中，提供同步方法是为了方便使用。

绝大部分在服务器端反复执行业务逻辑的JavaScript代码，必须使用异步；否则同步代码在执行时，服务器将停止响应，因为JavaScript是单线程。

服务器启动时读取配置文件或结束时写入状态文件时，可以使用同步代码。因为这些代码只在启动和结束时执行一次，不影响服务器正常运行时的异步执行。

* stream模块

stream是仅在服务区端可用的模块，目的是支持流这种数据结构。



流是一种抽象的数据结构，可以把数据看成是数据流。比如敲键盘时每个字符连起来看成字符流，此流是从键盘输入到应用程序，称为标准输入流(stdin)。

反之应用程序把字符一个一个输出到显示器上，称为标准输出流(stdout)。

流的特点是数据有序，必须依次读取或依次写入。

Node.js中流也是一个对象，只要响应流的事件就可以：data事件表示流的数据可以读取；end事件表示流结束，没有数据可读取了；error事件表示出错。

① 文件流读取文本

|  |
| --- |
| const fs = require('fs');  *// 打开一个输入流*  let rs = fs.createReadStream('sample.in', 'utf-8');  *// data事件可能有多次, 每次传递的chunk是流的一部分数据*  rs.on('data', function (chunk) {  console.log('data: ');  console.log(chunk);  });  rs.on('end', function () {  console.log('read stream end...');  });  rs.on('error', function (err) {  console.log(err);  }); |

② 以流的形式写入文件，只要不断调用write()，最后以end()结束：

|  |
| --- |
| let ws1 = fs.createWriteStream('output1.txt', 'utf-8');  ws1.write('使用Stream写入文本数据...\n');  ws1.write('END.');  ws1.end();  let ws2 = fs.createWriteStream('output2.txt');  ws2.write(Buffer.from('使用Stream写入二进制数据...\r\n', 'utf-8'));  ws2.write(Buffer.from('END.', 'utf-8'));  ws2.end(); |

用Windows自带的记事本'\n'没有换行；而'\r\n'换行。然而sublime、vscode都是有换行的...怪不得记事本总是被黑...

stream.Readable和stream.Writable是读取流和写入流的父类。

③ pipe

一个Readable流和一个Writable流串起来后，所有数据自动从Readable流进入Writable流，这种操作叫pipe。

Node.js中Readable流的pipe()方法，可以将一个文件流和另一个文件流串起来，源文件的数据自动写入到目标文件中，所以实际上是一个复制文件的过程：

|  |
| --- |
| function copy\_file(f1, f2) {  let rs = fs.createReadStream(f1);  let ws = fs.createWriteStream(f2);  rs.pipe(ws);  }  copy\_file('sample.png', 'copy.png'); |

默认当Readable流的数据读取完毕，end事件触发将自动关闭Writable流。

如果不希望自动关闭Writable流，需要传入参数：

readable.pipe(writable, { end: false });

**20180428**

* http模块

Node.js开发的目的是为了用JavaScript编写Web服务器程序。

从头处理TCP连接，解析HTTP是不现实的。Node.js自带的http模块可以完成这些工作。应用程序并不直接和HTTP协议打交道，而是操作http模块提供的request和response对象。

① request对象封装了HTTP请求，可以获取HTTP请求的信息；

② response对象封装了HTTP响应，可以把HTTP响应返回给浏览器。

先实现最简单的Web程序hello.js：对于所有请求都返回Hello world!

|  |
| --- |
| *// 导入http模块*  const http = require('http');  *// 创建http server, 传入回调函数*  let server = http.createServer(function (request, response) {  *// 回调函数接收request和response对象, 获得HTTP请求的method和url*  console.log(`${request.method}: ${request.url}`);  *// 将HTTP响应200写入response, 同时设置Content-Type: text/html*  response.writeHead(200, {  'Content-Type': 'text/html'  });  *// 将HTTP响应的HTML内容写入response*  response.end('<h1 style="color:red">hello world!</h1>');  });  *// 让服务器监听8000端口*  server.listen(8000);  console.log('Server is running at http://127.0.0.1:8000/'); |

vscode按F5调试程序，浏览器输入127.0.0.1:8000



控制台打印：

|  |
| --- |
| Server is running at http://127.0.0.1:8000/  GET: /  GET: /favicon.ico |

// 感觉进入了一个怪圈，就像用不同语言在学hello world...

* 文件服务器

解析request.url中的路径，在本地找到对应的文件，把文件内容发送。

① 解析URL需要用到Node.js提供的url模块，通过parse()方法将一个字符串解析为一个Url对象：

|  |
| --- |
| const url = require('url');  *// 解析url*  console.log(url.parse('http://user:pass@example.com:8000/path/to/file?a=1&b=2#top'));  *// 解析不完整的url*  console.log(url.parse('/static/js/jquery.js?name=Hello%20world'));  *// 构建一个url*  console.log(url.format({  protocol: 'http',  hostname: 'localhost',  pathname: '/static/js',  query: {  name: 'Nodejs',  version: 'v 1.0'  }  })); |

结果：

|  |
| --- |
| Url {protocol: 'http:', slashes: true, auth: 'user:pass', host: 'example.com:8000', port: '8000', hostname: 'example.com', hash: '#top', search: '?a=1&b=2', query: 'a=1&b=2', pathname: '/path/to/file', path: '/path/to/file?a=1&b=2', href: 'http://user:pass@example.com:8000/path/to/file?a=1&b=2#top' }  Url {protocol: null, slashes: null, auth: null, host: null, port: null, hostname: null, hash: null, search: '?name=Hello%20world', query: 'name=Hello%20world', pathname: '/static/js/jquery.js', path: '/static/js/jquery.js?name=Hello%20world', href: '/static/js/jquery.js?name=Hello%20world' }  http://localhost/static/js?name=Nodejs&version=v%201.0 |

② 处理本地文件目录需要使用Node.js提供的path模块，可以方便构造目录：

|  |
| --- |
| const path = require('path');  *// 解析当前目录:*  let work\_dir = path.resolve('.');  *// 组合完整的文件路径:*  let file\_path = path.join(work\_dir,'haha','hello.js');  console.log(work\_dir); *// d:\hikari\_git\test\180428*  console.log(file\_path); *// d:\hikari\_git\test\180428\haha\hello.js* |

使用path模块可以正确处理操作系统相关的文件路径。

③ 实现一个文件服务器file\_server.js：

|  |
| --- |
| const  fs = require('fs'),  url = require('url'),  path = require('path'),  http = require('http');  let root = path.resolve('.');  console.log('Static root dir: ' + root);  function http404(req, res) {  console.log('404 ' + req.url);  res.writeHead(404);  res.end('<h1 style=color:red>404 Not Found</h1>');  }  function read\_file(req, res, file) {  console.log('200 ' + req.url);  res.writeHead(200);  *// response对象是一个Writable流, 用pipe()方法自动读取文件并输出到HTTP响应*  fs.createReadStream(file).pipe(res);  }  let server = http.createServer(function (request, response) {  let  pathname = url.parse(request.url).pathname,  filepath = path.join(root, pathname);  fs.stat(filepath, function (err, stats) {  if (!err && stats.isFile()) { *// 是文件返回文件流*  read\_file(request, response, filepath);  *// 请求是目录, 在目录里找index.html或default.html文件*  } else if (!err && stats.isDirectory()) {  fs.readdir(filepath, function (err, files) {  if (err) {  response.end(err);  } else {  if (files.includes('index.html')) {  read\_file(request, response, './index.html');  } else if (files.includes('default.html')) {  read\_file(request, response, './default.html');  } else { *// 没出错, 但是找不到默认文件*  http404(request, response);  }  }  })  } else { *// 出错啦*  http404(request, response);  }  });  });  server.listen(8000);  console.log('Server is running at http://127.0.0.1:8000/'); |

index.html：

|  |
| --- |
| \* {font-family: "Microsoft Yahei";}  .title {color: #c08;}  .img {height: 300px;}  <h1 class="title">welcome to my website!!!</h1>  <div class="img"><img src="456t.png" alt="maki" height="280px"></div><hr>  <p>copyright&copy;2018</p> |

* crypto模块

crypto模块提供通用的加密和哈希算法。用纯JavaScript代码实现速度很慢，Nodejs用C/C++实现这些算法后，通过cypto模块暴露为JavaScript接口，调用方便，运行速度快。

① MD5和SHA1

|  |
| --- |
| const fs = require('fs');  const crypto = require('crypto');  const sha1 = crypto.createHash('sha1'); *// sha1算法*  function get\_sha1(f) {  let rs = fs.createReadStream(f);  let cnt = 1;  *// chunk貌似默认64字节, 那怎么设置chunk呢?*  rs.on('data', function (chunk) {  sha1.update(chunk); *// 可以多次update*  cnt++;  });  *// 读取结束, 输出sha1, 16进制的字符串*  rs.on('end', function () {  console.log(`cnt = ${cnt}`);  console.log(sha1.digest('hex'));  });  rs.on('error', function (e) {  console.log(e);  });  }  get\_sha1('1.mp4'); *// dbc85234c284b95b592fc600112b0837c9ce3d7b* |

用Python得到1.mp4的sha1与其一致，说明应该正确。

update()方法可以传入字符串或Buffer，默认字符串编码为UTF-8。

② Hmac

Hmac算法需要一个密钥，利用MD5或SHA1等哈希算法计算。

|  |
| --- |
| const hmac = crypto.createHmac('sha1', 'hikari');  let s = 'hello world';  sha1.update(s);  console.log(sha1.digest('hex')); *// 2aae6c35c94fcfb415dbe95f408b9ce91ee846ed*  hmac.update(s);  console.log(hmac.digest('hex')); *// 81cf28a45ad5c74e8d4215032ae1e38d8e6e0f52* |

只要密钥发生变化，同样的数据也会得到不同值；Hmac可以理解为用随机数增强的哈希算法。

③ AES (Advanced Encryption Standard, 高级加密标准)

AES是一种常用的对称加密算法，加解密用同一个密钥。crypto模块提供了AES支持，但是需要自己封装好函数，便于使用：

|  |
| --- |
| const crypto = require('crypto');  function aesEncrypt(data, key) { *// aes192加密*  const cipher = crypto.createCipher('aes192', key);  let crypted = cipher.update(data, 'utf8', 'hex');  crypted += cipher.final('hex');  return crypted;  }  function aesDecrypt(encrypted, key) { *// aes192解密*  const decipher = crypto.createDecipher('aes192', key);  let decrypted = decipher.update(encrypted, 'hex', 'utf8');  decrypted += decipher.final('utf8');  return decrypted;  }  let data = 'hoshizora rin';  let key = 'kayochin';  let encrypted = aesEncrypt(data, key);  let decrypted = aesDecrypt(encrypted, key);  console.log('原始: ' + data); *// 原始: hoshizora rin*  console.log('加密: ' + encrypted); *// 加密: 6160f8ec1495601943cd2b1c3b7f8e5f*  console.log('解密: ' + decrypted); *// 解密: hoshizora rin* |

注意：AES有很多不同的算法，如aes192，aes-128-ecb，aes-256-cbc等。除了密钥外还可以指定IV(Initial Vector)，不同系统只要IV不同，相同密钥加密相同数据得到的加密结果也不同。加密结果通常用hex或base64表示。

④ Diffie-Hellman秘钥交换算法

DH算法是一种密钥交换协议，可以让双方在不泄漏密钥的情况下协商出一个密钥。DH算法基于数学原理，比如A和B想要协商一个密钥：

|  |
| --- |
| 1) A先选一个素数如p=97，一个底数如g=23(底数任选)，再选择一个秘密整数a=34，计算A=g^a mod p=23\*\*34%97=65，然后告诉B：p=97, g=23, A=65；  2) B收到A发来的p, g, A后，选一个秘密整数b=88，计算B=g^b mod p=23\*\*88%97= 91，并告诉A：B=91；  3) A计算出sa=B^a mod p=91\*\*34%97=62，B计算出sb=A^b mod p=65\*\*88%97=62。因此最终协商的密钥s为62。 |

最终秘钥是双方协商计算出来的，而其他人只知道p=23, g=5, A=8, B=19，由于不知道双方的秘密整数a=34和b=88，因此无法计算出最终秘钥62。

crypto模块实现DH算法：

|  |
| --- |
| const crypto = require('crypto');  let a = crypto.createDiffieHellman(512);  let a\_keys = a.generateKeys(); *// 密钥a, 用作指数*  let prime = a.getPrime(); *// 随机生成一个素数?用于求余*  let generator = a.getGenerator(); *// 底数*  console.log('Prime: ' + prime.toString('hex'));  console.log('Generator: ' + generator.toString('hex'));  let b = crypto.createDiffieHellman(prime, generator);  let b\_keys = b.generateKeys(); *// 密钥b*  *// 交换并生成最终密钥*  let a\_secret = a.computeSecret(b\_keys);  let b\_secret = b.computeSecret(a\_keys);  console.log('Secret A: ' + a\_secret.toString('hex'));  console.log('Secret B: ' + b\_secret.toString('hex')); |

结果：

|  |
| --- |
| Prime: ca36…9d93 (128位, 随机)  Generator: 02  Secret A: 77a4…ed92 (128位, 随机, 但两个最终密钥相同)  Secret B: 77a4…ed92 |

crypto模块也可以处理数字证书。数字证书通常用在SSL连接，也就是Web的https连接。一般https连接只需要处理服务器端的单向认证，如无特殊需求，建议用反向代理服务器如Nginx等Web服务器处理证书。