# Node.js

## 起源：

RyanDahl想用高级语言开发web服务。

他虽然很多高级语言同时提供了同步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，因为JavaScript是单线程运行执行，根本不能执行同步IO操作。所以，最终Ryan选择了JavaScript。

选定开发语言后，Ryan采用了开源的V8作为运行引擎。Node第一次把JavaScript带入到后端服务器开发。

#### Node相比其他后端开发语言的优势

最大的优势是借助JavaScript天生的事件驱动机制加V8高性能引擎，使编写高性能Web服务很轻松。

## 模块

1. 一个模块想要对外暴露变量（函数也是变量），可以用 module.exports = variable;
2. 一个模块想要引入其他模块暴露的变量，用var ref = require(‘module\_name’);
3. 这种模块加载机制被称为CommonJS规范，每个js都是一个模块，它们内部的各自使用的变量和函数名都互不冲突。
4. Java没有模块机制，但是JavaScript支持闭包，可以使用一个函数包起来代码，这块代码的“全局变量”也就变成了局部变量。

### 基本模块

1. nextTick 在下一次事件响应中执行代码
2. exit 在程序即将退出时执行某个回调函数
3. javaScript有且仅有一个全局对象,浏览器的全局对象为window，node的全局对象为global.
4. 可以通过全局对象判断JavaScript执行环境

### fs

1. fs就是文件系统模块，负责读写文件。和JavaScript其他模块不同,fs模块同时提供了异步和同步的方法。
2. 同步操作的好处是代码简单，缺点是程序将等待IO操作，在等待时间内，无法响应其它任何事件。

而异步读取不用等待IO操作，但代码较麻烦。

#### 异步读文件

var fs = require(‘fs’);

fs.readFile(‘sample.txt’,’utf-8’,function(err,data){

if(err)

console.log(err);

else

console.log(data);

});

注意，sample.txt必须在当前目录下，且文件编码为utf-8。

1. fs.readFile('sample.png', function (err, data) {});读取二进制文件

当读取二进制文件，不传入文件编码时，回调函数的data参数将返回一个Buffer对象，在Node.js中，buffer对象就是一个包含0个或任意字节的数组。

1. Buffer 对象可以和string转换： var text = buffer.toString(‘utf-8’);
2. 或者把一个String对象转换为Buffer对象： var buffer = new Buffer(text,‘utf-8’);

#### 异步读文件

fs.readFile('AsynSample.txt','utf-8',function(err,data){});

#### 同步读文件

fs.readFileSync('SyncSample.txt','utf-8');

#### 异步写文件

fs.writeFile('output.txt',outputTextData,function(err){});

#### 同步写文件

fs.writeFileSync('outputSync.txt',outputTextDataSync);

#### stat

fs.stat(‘sample.txt’,function(err,stat){

if(err){

console.log(err);

}else{

console.log(“isFile” + stat.ifFile()); //是否文件

console.log(“isDirectory” + stat.isDirectory());//是否目录

if(stat.isFile()){

console.log(‘size:’ +stat.size);//文件大小

console.log(‘birth time:’ + stat.birthtime);

console.log(‘modified time’ + stat.mtime);

}

}

})

var fileInfo = fs.statSync(".vscode");

1. 由于Node环境执行的JavaScript代码是服务器端代码，所以，绝大部分需要在服务器运行期反复执行业务逻辑的代码，必须使用异步代码，否则，同步代码在执行时期，服务器将停止响应，因为JavaScript只有一个执行线程。
2. 服务器启动时如果需要读取配置文件，或者结束时需要写入到状态文件时，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

1. stream是Node.js提供的一个仅在服务器端可用的模块，目的是支持“流”这种数据结构。
2. 在Node.js中，流也是一个对象，我们只需要响应流的事件就可以了：data事件表示流的数据已经可以度去了，end事件表示这个流已经到末尾了，没有数据可以读取了，error事件表示出错了。
3. 读文件
   1. 引入fs模块 var fs = require(‘fs’);
   2. 创建读文件流 var rs = fs.createReadStream(‘test.txt’,’utf-8’);
   3. data事件表示流的数据已经可以读取了rs.on('data',function(chunk){console.log('DATA:'+ chunk);});
   4. end事件表示这个流已经到末尾了，没有数据可以读取了

rs.on('end',function(){console.log('end');});

* 1. error事件表示出错了。

rs.on('error',function(err){console.log('ERROR:',err););

1. 写文件

要以流的形式写入文件，只需要不断调用write()方法，最后以end()结束

* 1. 引入fs模块 var fs = require(‘fs’);
  2. 创建写文件流 var rs = fs.createWriteStream(‘test.txt’,’utf-8’);
  3. 所有可以读取数据的流都继承自stream.Readable，所有可以写入的流都继承自stream.Writable。

1. pipe 一个Readable和一个Writeable流穿起来后，所有数据自动从Readableliu进去Writable流，这种操作叫pipe。
2. var rs = fs.createReadStream('test.txt','utf-8');

var ws1 = fs.createWriteStream('output.txt','utf-8');

rs.pipe(ws1);

1. 默认情况下，当Readable流的数据读取完毕，end事件触发后，将自动关闭Writable流。如果我们不希望自动关闭Writable流，需要传入参数。

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

### http

#### HTTP服务器

1. request对象封装了HTPP请求，我们调用request的对象和方法就可以拿到所有Htttp请求的信息。
2. response对象分装了Http响应，我们操作response对象的方法，就可以把Http响应返回给浏览器。
3. http模块

eg:

1. 'use strict';
2. //导入http模块
3. var http = require('http');
4. //创建http server,并传入回调函数;
5. var server = http.createServer(function(request,response){
6. //回调函数接收request和response对象，
7. //获得HTTP请求的method和url:
8. console.log(request.method + ':' + request.url);
9. //将HTTP响应200写入response,同时设置Content-Type:text/htmll:
10. response.writeHead(200,{'Content.Type':'text/html'});
11. //将HTTP响应的HTML内容写入response:
12. response.end('<h1>Hello world!</h1>');
13. });
14. //让服务器监听8080端口：
15. server.listen(8080);
16. console.log('Server is running at <http://127.0.0.1:8080/>');

浏览器访问 localhost:8080

1. url模块

解析URL需要用到Node.js提供的url模块，它使用起来非常简单，通过parse()将一个字符串解析为一个Url对象

1. path模块

使用path模块可以正确处理操作系统相关的文件路径。在Windows系统下，返回的路径类似于C:\Users\michael\static\index.html，这样，我们就不关心怎么拼接路径了。

1. 实现文件服务器：

eg:

'use strict'

var

fs = require('fs'),

http = require('http'),

path = require('path'),

url = require('url');

//从命令行参数获取root目录，默认是当前目录：

var root = path.resolve(process.argv[2] || './index.html');

console.log('Static root dir:' + root );

//创建服务器：

var server = http.createServer(function(request,response){

//获得URL的path，类似 ‘/css/bootstrap.css’:

var pathname = url.parse(request.url).pathname;

//获取对应的本地文件路径，类似与‘/srv/www/css/bootstrap.css’:

var filepath = path.join(root,pathname);

//获取文件状态：

fs.stat(filepath,function(err,stats){

if(!err&&stats.isFile()){

//没有出错并且文件存在：

console.log('200' + request.url);

//发送200响应：

response.writeHead(200);

//将文件流导向response:

fs.createReadStream(filepath).pipe(response);

}else{

//出错了或者文件不存在：

console.log('404' + request.url);

//发送404响应：

response.writeHead(404);

response.end('404 Not Found');

}

});

});

server.listen(8080);

console.log('fileService is running at http://127.0.0.1:8080/');

### crypto

crypto是为了提供通用的加密和哈希算法。用纯JavaScript代码实现这些功能不是不可能，但是速度会非常慢。Nodejs用C/C++实现这些算法后，通过cypto这个模块暴露为JavaScript接口，这样用起来方便，运行也快。

#### MD5和SHA1

MD5是一种常用的哈希算法，用于给任意数据一个“签名”。这个签名通常用一个十六进制的字符串表示：

const crypto = require(‘crypto’);

const hash = crypto.createHash(‘md5’);

//可任意多次调用update();

hash.update('xuanmiao');

console.log(hash.digest('hex'));

如果要计算SHA1，只需要把‘md5’改成‘sha1’,就可以得到SHA1的结果

还可以使用更安全的sha256和sha512。

#### Hmac

Hmac也是一种哈希算法，它可以利用DM5或SHA1等哈希算法。不同的是，Hmac还需要一个密钥：

const hmac = crypto.createHmac(‘sha256’,’secret-key’);

hmac.update(‘xuanmiao’);

console.log(hmac,digest(‘hex’));

只要密钥发生了变化，那么同样的输入数据也会得到不同的签名，因此，可以把Hmac理解为用随机数“增强”的哈希算法。

#### AES

AES是一种常用的堆成加密算法，加解密都用同一个密钥。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’);

注意到AES有很多不同的算法，如aes192，aes-128-ecb，aes-256-cbc等

#### Diffie-Hellman

DH算法是一种密钥交换协议，它可以让双方在不泄漏密钥的情况下协商出一个密钥来。DH算法基于数学原理，比如小明和小红想要协商一个密钥，可以这么做：

1.小明先选一个素数和一个底数，例如，素数p=23，底数g=5（底数可以任选），再选择一个秘密整数a=6，计算A=g^a mod p=8，然后大声告诉小红：p=23，g=5，A=8；

2.小红收到小明发来的p，g，A后，也选一个秘密整数b=15，然后计算B=g^b mod p=19，并大声告诉小明：B=19；

3.小明自己计算出s=B^a mod p=2，小红也自己计算出s=A^b mod p=2，因此，最终协商的密钥s为2。

在这个过程中，密钥2并不是小明告诉小红的，也不是小红告诉小明的，而是双方协商计算出来的。第三方只能知道p=23，g=5，A=8，B=19，由于不知道双方选的秘密整数a=6和b=15，因此无法计算出密钥2。