Contents

[<Node.js in Action> 1](#_Toc444620878)

[环境变量 1](#_Toc444620879)

[Node模块 1](#_Toc444620880)

[module.exports对象和函数 1](#_Toc444620881)

[模块文件和目录 1](#_Toc444620882)

[Node基本模块 2](#_Toc444620883)

[数据存储 2](#_Toc444620884)

[配置函数 2](#_Toc444620885)

[Node典型使用 2](#_Toc444620886)

[作为脚本使用 2](#_Toc444620887)

[作为服务器使用 3](#_Toc444620888)

[Express 7](#_Toc444620889)

# <Node.js in Action>

## 环境变量

在windows配置环境变量

>set NODE\_ENV=production

## Node模块

### module.exports对象和函数

Node模块允许你从被引入文件中选择要暴露给程序的函数和变量。如果模块返回的函数或变量不止一个，那它可以通过设定exports对象的属性来指明它们；但如果模块只返回一个函数或变量，则可以设定module.exports属性。

### 模块文件和目录

模块既可能是一个文件，也可能是包含一个或多个文件的目录。如果模块是个目录，node通常会在这个目录下找一个叫index.js的文件作为模块的入口

典型的模块是一个包含exports对象属性定义的文件，这些属性可以是任意类型的数据，比如字符串，对象和函数

lib/currency.js

exports.canadianToUS = function(canadian)( return ….) ;

exports.USToCanadian = function(us) { return … };

使用这个新模块要用到node的require函数，该函数以你要用的模块的路径为参数。Node以同步的方式寻找它，定位到空上模块并加载文件中的内容

var currency = require(‘./lib/currency’) ;

currency.canadianToUS(50) ;

### Node基本模块

\_\_dirname: 该文件所在目录的路径

var parse = require(‘url’).parse ;

var join = require(‘path’).join ;

var fs = require(‘fs’) ;

var url = parse(req.url)；

fs.readFile(join(\_\_dirname, url.pathname), function(err, data){

if (err) return res.end(‘Server Error’) ;

…

}) ;

console.log(‘%s %s’, req.method, req.url) ;

## 数据存储

存储node程序中的数据：内存，文件系统，数据库（SQL and NoSQL）

## 配置函数

为了向开发人员提供可配置的能力，中间件通常会遵循一个简单的惯例：用函数返回另一个函数，基本结构看起来如下：

function setup(options) {

//设置逻辑

…

return function(req, res, next) {

…

}

}

## Node典型使用

### 作为脚本使用

cmd>node

node>var qs = require('querystring') ;

node>qs.parse('item=QiZhongLin') ; //结果：{item: 'QiZhongLin'}

### 作为服务器使用

#### 简单服务器

var http = require(‘http’) ;

var server = http.createServer(function(req, res){

res.end(‘hello world’) ;

}) ;

server.listen(3000) ;

服务器每次收到http请求后都会调用这个回调函数，这个请求回调会收到两个参数，请求和响应对象，通常简写为req and res

#### 使用connect搭建服务器

##### 基本概念

Connect是一个框架，使用被称为中间件的模块化组件，以可重用的方式实现web程序中的逻辑，在connect中，中间件是一个函数，它拦截http服务器提供的请求和响应对象，执行逻辑，然后或者结束响应，或者把它传递给下一个中间件组件。Connect用分派器把中间件连接在一起。

在connect中，中间件组件是一个javascript函数，按惯例会接受三个参数：一个请求对象，一个响应对象，还有一个通常命令名next的参数，它是一个回调函数，表明这个组件已经完成了它的工作，可以执行下一个中间件组件了。

错误处理：在开发时你可能想用json格式把错误发送到客户端，做简单快捷的错误报告，而在生产环境中，你可能只想响应一个简单的“服务器错误”，以免把敏感的内部信息（比如堆栈跟踪，文件名和行号等）暴露给潜在的攻击者。错误处理中间件函数必须接受四个参数：err, req, res and next

>npm install connect

var connect = require(‘connect’) ;

var app = connect() ;

var api = app.use(users).use(pets) ;

app.use(connect.logger()) ; //user()函数返回的是支持方法链的connect程序实例。

//api服务挂载在/api上,即将中间件挂载到特定的根url下，从而一程序内创建程序

app.use(‘/api’, api) ;

app.user(errorHandler()) ;

app.listen(3000) ;

function errorHandler(){

var env = process.env.NODE\_ENV || ‘development’ ;

return function(err, req, res, next) {

res.statusCode = 500 ;

switch (env) {

case ‘development’:

res.setHeader(‘Content-Type’, ‘application/json’) ;

res.end(JSON.stringify(err)) ;

break ;

default:

res.end(‘Server error’) ;

}

}

}

##### 读取请求头及设定响应头

req.method, req.url, req.headers.authorization

res.setHeader(field, value) /res.getHeader(field)/res.removeHeader(field)

res.setHeader(‘Content-Type’, ‘text/plain’) ;

Connect内置中间件

##### Session():

会话管理 (前提：cookieParser(): 解析http cookie)

当请求完成时，赋给req.session对象的所有属性都会被保存下来；当相同的用户（浏览器）再次发来请求时，会加载它们。

.use(connect.cookieParser(‘cookie private key’) )

.use(connect.session(

{ key: ‘sid’,

cookie: {

maxAge: 3600000 \*24,

secure: true,

//会话存储，低延迟的键值存储最适合易失性数据。把指定版本的connect传给它，可以确保connect-redis用的是正确的版本

store: new require(‘connect-redis’)(connect)({ prefix: ‘sid’})

}

}))

req.session.cart = {items: [1, 2, 3]} ;

req.session.cookies.expires/maxAge/path/domain/secure

req.cookies, req.signedCookies; //服务端获得客户端传来的cookie

res.setHeader(‘Set-Cookie’, ‘foo=bar’) ; //服务端返回cookie

curl <http://localhost:3000/> -H ‘Cookie: foo=bar’

##### query(): 查询字符串解析

/song?artist=Bob%20Marley&track=Jammin

req.query =

{ artist: ‘Bob Marley’,

track: ‘Jamin’

}

.use(connect.query())

.use(function(req, res, next){

req.query ;

});

##### bodyParser(): 解析请求主体

若请求body是json, x-www-form-urlencoded,可以用req.body

若请求body是multipart/form-data,比如文件上传，则用req.files

curl –d ‘{“username”:”tobi”}’ –H “Content-Type: application/json” //客户端发送json

curl –d name=tobo <http://localhost:3000/> //客户端发送x-www.form-urlencoded数据

.use(connect.bodyParser())

.use(function(req, res){

req.body.username ; //服务端获取数据

}) ;

curl –F [image=@photo.png](mailto:image=@photo.png) -F name=tobi <http://localhost:3000> //客户端上传文件

req.body.name ;

req.files;

##### limit(): 请求主体的限制

function type(type, fn) {

return function(req, res, next) {

var ct = req.headers[‘content-type’] || ‘’ ;

if (0 != ct.indexOf(type) ) { return next() ; }

fn(req, res, next) ;

}

}

.use(type(‘application/json’, connect.limit(‘32kb’))

.use(type(‘application/x-www-form-urlencoded’, connect.limit(‘64kb’))

.use(type(‘application/image’, connect.limit(‘2mb’))

.use(type(‘video’, connect.limit(‘300mb’))

##### favicon(): 提供favicon

favicon是网站的小图标，显示在浏览器的地址栏和收藏栏里。为了得到这个图标，浏览器会请求/favicon.ico文件。

.use(connect.favicon(\_\_dirname + ‘/public/favicon.ico’))

##### basicAuth(): http基本认证

基本认证要配合https进行认证

.use(connect.basicAuth(‘usename’, ‘password’) ; //直接用户名和密码

.use(connect.basicAuth(function(user, pass){ //回调函数

return users[user] === pass ;

}) ;

.use(connect.basicAuth(function(user, pass, callback){ //执行数据库验证

User.authenticate({user: user, pass: pass}, function(err, user){

if (err) return callback(err) ;

callback(null, user) ;

}) ;

})) ;

curl –user tobi:ferret <http://localhost:3000> 每个url请求每要带凭证

##### static(): 返回静态文件

.use(connect.static(‘public’)) ./public目录下的静态资源

.use(‘/app/files’, connect.static(‘public’)) //利用挂载，从多个目录中提供静态文件

##### compress(): 压缩静态文件

通过请求头域Accept-Encoding自动检测客户端可接受的编码。如果请求头中没有该域，则使用相同的编码，也就是说不会对响应做处理。如果请求头的该域中包含gzip,defalte，则响应会被压缩.

.use(connect.compress()) ;

curl <http://localhost:3000/script.js> -i –H “Accept-Encoding: gzip”

### Express

>npm install –g express

客户端 GET /photo/:id/download

var id = req.params.id //获取url参数

Photo.findById(id, function(err, photo){

if (err) return next(err) ;

res.download(join(dir, photo.path)) ;

}) ;

#### 用户认证

POST/Redirect/GET模式是一个常用的Web程序设计模式，用户请求表单，用http post请求提交表单数据，然后用户被重定向到另外一个Web页面上。如果表单数据无效，程序会让用户回到刚才表单页面；若表单数据有效，程序会让用户到新的页面。

注册新用户

app.js //前端控制器

var register = require(‘./routes/register’) ;

app.get(‘/register’, register.form) ; //获取注册表单

app.post(‘/register’, register.submit) ; //递交注册用户

route.register.js

exports.form = function(req, res) {

res.render(‘register’, {title: ‘Register’}) ;

}

views/register.ejs

<form action=’/register’ method=’post’>

<input type=’text’ name=’user[name]’/>

<input type=’password’ name=’user[pass]’/>

<input type=’submit’ />

</form>

var User = require(‘../lib/user’) ;

exports.submit = function(req, res, next){

var data = req.body.user ;

User.getByName(data.name, function(err, user) {

if (err) return next(err) ;

if (user.id) { //基于用户名，从数据库中获取用户信息，基于user.id判别唯一性

res.error(“Username already taken!”) ;

res.redirect(‘back’);

} else {

user = new User({name: data.name, pass: data.pass}) ;

user.save(function(err){ // 保存新用户

if (err) return next(err) ;

req.session.uid = user.id ; //会话管理新用户id

res.redirect(‘/’) ; //跳转新页面

}) ;

}

}) ;

} ;

//用户User的CRUD操作

lib/user.js

var redis = require(‘redis’) ;

var bcrypt = require(‘bcrypt’)；

var db = redis.createClient() ;

function User(obj) {

for (var key in obj) { this[key] = obj[key] ; }

}

User.prototype.save = function(fn) {

if (this.id) this.update(fn) ;

else {

var user = this ;

db.incr(‘user:ids’, function(err, id){

if (err) return fn(err) ;

user.id = id ;

user.hashPassword(function(err){

if (err) return fn(err) ;

user.update(fn) ;

}) ;

}

}

} ;

User.prototype.update = function(fn){

var user = this ;

var id = user.id ;

db.set(‘user:id:’ + user.name, id, function(err){

if (err) return fn(err) ;

db.hmset(‘user:’ + id, user, function(err){

fn(err) ;

})

}

};

User.prototype.hassPassword = function(fn){

var user = this ;

bcrypt.genSalt(12, function(err, salt){

if (err) return fn(err) ;

user.salt = salt ;

bcrypt.hash(user.pass, salt, function(err, hash){

if (err) return fn(err) ;

user.pass = hash ;

fn() ;

}) ;

}) ;

};

User.getByName = function(name, fn) {

User.getId(name, function(err, id){

If (err) return fn(err) ;

db.hgetall(‘user:’+id, function(err, user){

if (err) return fn(user) ;

fn(null, new User(user)) ;

}) ;

});

} ;

User.authenticate = function(name, pass, fn) {

User.getByName(name, function(err, user){

if (err) return fn(err) ;

if (!user.id) return fn() ; //用户不存在

bcrypt.hash(pass, user.salt, function(err, hash){

if (err) return fn(err) ;

if (hash == user.pass) return fn(null, user) ; 找到用户

fn() ; //密码无效

}) ;

}) ;

};

已注册用户登录

var login = require(‘./routes/login’) ;

app.get(‘/login’, login.form) ; //获取登录表单

app.post(‘/login’, login.submit) ; //提供登录的用户

app.get(‘/logout’, login.logout) ; //退出

routes/login.js

exports.form = function(req, res){

res.render(‘login’, {title: ‘Login’}) ;

};

views/login.ejs

<form action=’/login’ method=’post’>

<input type=’text’ name=’user[name]’ />

<input type=”password’ name=’user[pass]’ />

<input type=’submit’ />

</form>

var User = require(‘../lib/user’) ;

exports.submit = function(req, res, next) {

var data = req. body.user ;

User.authenticate(data.name, data.pass, function(err, user){ //检查凭证

if (err) return next(err) ;

if (user) { // 用户凭证有效

req.session.uid = user.id ; //会话管理

res.redirect(‘/’) ; //跳转新页面

} else {

res.error(“invalid credentials.”) ;

res.redirect(‘back’) ; //停留当前页面

}

}) ;

};

exports.logout = function(req, res) {

req.session.destroy(function(err) {

if (err) throw err ;

res.redirect(‘/’) ;

}) ;

};

REST api:

app.use(‘/api’, express.basicAuth(User.authenticate)) ;

app.get(‘/api/user/:id’, api.user) ;

app.get(‘/api/entries/:page?’, api.entries) ;

app.post(‘api/entry’, api.add) ;

app.use(function(req, res){

res.status(404).format(…) ; //错误处理

}) ;

curl <http://username:password@localhost:3000/api/user/1> -v