Node初级—Express框架笔记

1. Exprees框架简介

### 实现一个简单的http服务器

Express框架是后台的Node框架，所以和jQuery、zepto、yui、bootstrap都不一个东西。

Express在后台的受欢迎的程度，和jQuery一样，就是企业的事实上的标准。

● 原生Node开发，会发现有很多问题。比如：

■ 呈递静态页面很不方便，需要处理每个HTTP请求，还要考虑304问题

■ 路由处理代码不直观清晰，需要写很多正则表达式和字符串函数

■ 不能集中精力写业务，要考虑很多其他的东西

● EXPRESS的哲学是在你的想法和服务器之间充当薄薄的一层。这并不意味着他不够健壮，或者没有足够的有用特性，而是尽量少干预你，让你充分表达自己的思想，同时提供一些有用的东西。

从本质上说EXPRESS，完全由路由和中间件构成的web开发框架，即一个Express应用是在调用各种中间件。

英语官网：http://expressjs.com/

中文官网：<http://www.expressjs.com.cn/>

整体感知，Express框架。

安装Express框架，就是使用npm的命令。

|  |
| --- |
| 1. npm install **--save** express 2. npm install express --save |

--save参数，表示自动修改package.json文件，自动添加依赖项。

1. 路由

当用get请求访问一个网址的时候，做什么事情：

|  |
| --- |
| 1. app.get("/login",(req,res) =>{ 3. }); |

当用post访问一个网址的时候，做什么事情：

|  |
| --- |
| 1. app.post("网址",(req,res) =>{ 3. }); |

如果想处理这个网址的任何method的请求，那么写all

|  |
| --- |
| 1. app.all("/",() =>{ 3. }); |

这里的网址，不分大小写，也就是说，你路由是

|  |
| --- |
| 1. app.get("/AAB",(req,res) =>{ 2. res.send("你好"); 3. }); |

所有的GET参数，? 后面的都已经被忽略。 锚点#也被忽略

你路由到/a ， 实际/a?id=2&sex=nan 也能被处理。

正则表达式可以被使用。正则表达式中，未知部分用圆括号分组，然后可以用req.params[0]、[1]得到，也可以用

req.params类数组对象。

|  |
| --- |
| 1. app.get(**/^\/student\/([\d]{10})$/**,(req,res) => { 2. res.send("学生信息，学号" + **req.params[0]**); 3. }); |

推荐的写法是冒号。

|  |
| --- |
| 1. app.get("/student/:变量名字",(req,res) => { 2. var id = req.params["id"]; 3. var reg= /^[\d]{6}$/; //正则验证 4. if(reg.test(id)){ 5. res.send(id); 6. }else{ 7. res.send("请检查格式"); 8. } 9. }); |

表单可以自己提交到自己上。

|  |
| --- |
| 1. app.get("/",(req,res) =>{ 2. res.render("form"); 3. }); 4. app.post("/",(req,res) =>{ 5. //将数据添加进入数据库 6. res.send("成功"); 7. }); |

适合进行 RESTful路由设计。简单说，就是一个路径，但是http method不同，对这个页面的使用也不同。

子路由，所有/article开头的url都会交由子路由sub\_router 执行中间件函数

|  |
| --- |
| 1. var sub\_router = express.Router(); 2. app.use('/article',sub\_router); |

/student/345345

练习：

get 读取用户信息

add 添加用户信息

delete 删除用户信息

1. app.route与express.Router

在app.js 中调用 express() 时，返回一个app对象。而app对象可以看作创建了一个Express应用程序，app的作用是做一些全局的处理。

router是开发人员做业务处理的利器（比如你可以通过router去use你自己写的针对某个特定路由的中间件），这样的好处可以更细的划分controller，router就是一个mini的app。router可以更细的划分controller。

“迷你应用程序”背后的想法是，应用程序中的不同路线可能变得相当复杂，你可以从将该逻辑移动到单独的文件中获益。

Dog.js

|  |
| --- |
| 1. var express = require('express'); 2. var router = express.Router(); 3. router.get('/', function(req, res) { 4. res.send('GET handler for /dogs route.'); 5. }); 6. router.post('/', function(req, res) { 7. res.send('POST handler for /dogs route.'); 8. }); 9. module.exports = router; |

在上面代码中，/dogs 路由的逻辑已经被移动到自己的文件中，所以它的 GET 和 POST 处理程序不会混乱 app.js。

现在，你可以独立地处理对 /dog 的任何请求的逻辑，而不用担心它将如何影响猫 /birds。

如果你具有与所有三条路由相关的逻辑（Express 中称为中间件），则可以将其放在 app.use(...) 调用上方的 app.js 中。

如：app.js

|  |
| --- |
| 1. var express = require('express'), 2. dogs = require('./routes/dogs'), 3. cats = require('./routes/cats'), 4. birds = require('./routes/birds'); 5. var app = express(); 6. app.use('/dogs', dogs); 7. app.use('/cats', cats); 8. app.use('/birds', birds); 9. app.listen(3000); |

如果你有逻辑仅与其中一条路线（/dog）相关，那么你只将它放在该路由的文件中。

如果代码非常简单，完全可以 app.get('/',....),如果路由比较复杂，使用 express.Router() 更合适。

1. 中间件

中间件(middleware)是一个函数,他可以访问请求对象（request object(req)）,响应对象（response object(res)）和web应用中处于请求-响应循环

Express可以使用如下几种中间件：

- 应用级中间件

- 路由级中间件

- 内置中间件

- 第三方中间件

- 错误处理中间件

### 应用级中间件

应用级中间键绑定到app对象使用app.use和app.METHOD()-需要处理http请求的方法，例如GET、PUT、POST

|  |
| --- |
| 1. var app = express() 2. // 没有挂载路径的中间件，应用中的每个请求都会执行该中间件 3. app.use((req,res,next) => { 4. console.log('Time',Dtae.now()); 5. next(); // 传递request对象给下一个中间件 6. }) 7. // 挂载至/user/:id的中间件，任何执行/user/:id的请求都会执行它 8. app.use('/use/:id',(req,res,next) => { 9. console.log('Request Type',req.method); 10. next(); 11. }) 12. // 路由和句柄函数（中间件系统），处理指向/user/:id的GET请求 13. app.get('/user/:id',(req,res,next)=>{ 14. console.log('USER'); 15. }) |

如果我们想要处理挂在至/user/:id的中间件的GET请求，我们需要使用next()将request对象传递给下一个中间件

否则，得不到下一个中间件处理的它，一直在等待...最终会抛出localhost未发送任何数据的错误。

如何你不想要终止请求-响应循环,总是记得通过next()传递request对象

--------------------------------------------------------------------------------

如果你想要在中间件栈中跳过剩余中间件，调用next('route')方法将控制权交给下一个路由

|  |
| --- |
| 1. app.get('/user/:id',(req,res,next)=>{ 2. if(req.params.id==0) next('route') 3. else next() 4. },(req,res,next)=>{ 5. // 渲染常规页面 6. res.render('regular') 7. }) 8. // 处理/user/:id，渲染一个id为0的特殊页面 9. app.get('/user/:id',(req,res,next)=>{ 10. res.render('special') 11. }) |

### 路由级中间件

路由级中间件和应用级中间件类似，只不过是它绑定对象为express.Router()

|  |
| --- |
| 1. var router = express.Router() |

路由级使用router.use()或router.VERB()加载

比如：

|  |
| --- |
| 1. var app = express() 2. var router = express.Router() 3. // 没有挂载路径的中间件，通过该路由的每个请求都会执行该中间件 4. router.use(function (req, res, next) { 5. console.log('Time:', Date.now()); 6. next(); 7. }) 8. // 一个中间件，显示任何指向/user/:id的HTTP请求的信息 9. router.use('/user/:id',(req,res,next)=>{ 10. console.log('Request URL',req.originalUrl) 11. next() 12. },(req,res,next)=>{ 13. console.log('Request Type',req.method) 14. next() 15. }) 16. // 一个中间件栈，处理指向/user/:id的GET请求 17. router.get('/user/:id',(req,res,next)=>{ 18. if(req.params.id == 0) next('router') 19. else next() 20. },(req,res,next)=>{ 21. res.render('regular') 22. }) 23. // 处理/user/:id，渲染一个特殊页面 24. router.get('user/:id',(req,res,next)=>{ 25. console.log(req.params.id) 26. res.render('special') 27. }) 28. // 将路由挂载至应用 29. app.use('/子路由名字',router) |

### 内置中间件

从版本4.x开始，Express不再依赖Content，除了 express.static, Express 以前内置的中间件现在已经全部单独作为模块安装使用

express.static是 Express 唯一内置的中间件。它基于 serve-static，负责在 Express 应用中提托管静态资源。

可选options参数拥有如下属性

|  |  |  |  |
| --- | --- | --- | --- |
| 属性 | 描述 | 类型 | 缺省值 |
| dotfiles | 是否对外输出文件名以点（.）开头的文件。可选值为 “allow”、“deny” 和 “ignore” | String | "ignore" |
| etag | 是否启用etag生成 | Boolean | true |
| extensions | 设置文件扩展名备份选项 | Array | [ ] |
| index | 发送目录索引文件，设置为 false 禁用目录索引。 | mixed | "index.html" |
| lastModified | 设置 Last-Modified 头为文件在操作系统上的最后修改日期 | Boolean | true |
| maxAge | 毫秒或者其字符串格式设置 Cache-Control 头的 max-age 属性 | Number | 0 |
| redirect | 当路径为目录时，重定向至"/" | Boolean | true |
| setHeaders | 设置HTTP头以提供文件的函数 | Function |  |

下面的栗子使用了 express.static 中间件，其中的 options 对象进行了详细设计。

|  |
| --- |
| 1. var options = { 2. dotfiles: 'ignore', 3. etag: false, 4. extensions: ['htm', 'html'], 5. index: ‘, 6. maxAge: '1d', 7. redirect: false, 8. setHeaders: function (res, path, stat) { 9. res.set('x-timestamp', Date.now()); 10. } 11. } 12. app.use(express.static('public', options)); |

我们总是需要使用express.static指定多个静态资源文件，比如：

|  |
| --- |
| 1. app.use(express.static('public')) 2. app.use(express.static('files')) |

### 第三方中间件

通过使用第三方中间件从而为Express应用增加更多的功能

安装所需功能的node模块，并在应用中加载，可以在应用级中加载，也可以在路由级中加载

比如：

|  |
| --- |
| 1. $ npm install cookie-parser 2. var express = require('express') 3. var app = express() 4. var cookieParser = require('cookie-parser') 5. // 加载用于解析cookie的中间件 6. app.use(cookieParser()) |

### 错误处理中间件

注：错误处理中间件有四个参数,定义错误处理中间件必须使用这四个参数。即使不需要next对象，也必须在参数中声明它，否者中间件会识别为一个常规中间件，不能处理错误

比如：

|  |
| --- |
| 1. app.use((err,req,res,next)=>{ 2. console.error(err.stack) 3. res.status(500).send('Something broke') 4. }) |

中间件返回的响应是随意的，可以响应一个 HTML 错误页面、一句简单的话、一个 JSON 字符串，或者其他任何您想要的东西。

所以你可能想要像处理常规中间件那样，定义多个错误处理中间件,比如您想为使用 XHR 的请求定义一个，还想为没有使用的定义一个，那么：

|  |
| --- |
| 1. app.use(logErrors) 2. app.use(clientErrorHandler) 3. app.use(errorHandler) |

logErrors 将请求和错误信息写入标准错误输出、日志或者类似服务

|  |
| --- |
| 1. function logErrors(err,req,res,next){ 2. console.error(err.stack) 3. next(err) 4. } |

clientErrorHandler 定义如下(这里将错误直接传给了next)

|  |
| --- |
| 1. function clientErrorHandler(err,req,res,next){ 2. if(req.xhr){ 3. res.status(500).send({error:'Something blew up!'}) 4. }else{ 5. next(err) 6. } 7. } |

errorHandler 捕获所有错误

|  |
| --- |
| 1. function errorHandler(err,req,res,next){ 2. res.status(500) 3. res.render('error',{error:err}) 4. } |

1. Get请求和Post请求参数

● GET请求的参数在URL中，在原生Node中，需要使用url模块来识别参数字符串。在Express中，不需要使用url模块了。可以直接使用req.query对象。

● POST请求在express中不能直接获得，必须使用body-parser模块。使用后，将可以用req.body得到参数。但是如果表单中含有文件上传，那么还是需要使用formidable模块。

Node中全是回调函数，所以我们自己封装的函数，里面如果有异步的方法，比如I/O，那么就要用回调函数的方法封装。

错误：

|  |
| --- |
| 1. res.reder("index",{ 2. "name" : student.getDetailById(234234).name 3. }); |

正确:

|  |
| --- |
| 1. student.getDetailByXueHao(234234, (**detail**) =>{ 2. res.render("index",{ 3. "name" : **detail**.name 4. }) 5. }); |

POST写法

|  |
| --- |
| 1. const bodyparser = require('body-parser'); 2. //用来处理application/x-www-form-urlencoded类型的内容 3. app.use(bodyparser.urlencoded({extended: false})); 4. //用来处理application/json类型的内容 5. app.use(bodyparser.json()); 6. //用来处理text/plain类型的内容 7. app.use(bodyParser.text()); |

1. 路由的快捷写法

|  |
| --- |
| 1. app.route('/login') 2. .get((req,res,next) => { 3. res.send('get login !!'); 4. }) 5. .post((req,res,next) => { 6. res.send('post login !!'); 7. }); |