

Node介绍

node是什么

- node不是编程语言，也不是库或者框架
- node是js的运行环境

js在node和浏览器中的区别

浏览器中的js

- EcmaScript(js基础语法)
- BOM
- DOM

node中的js

- EcmaScript
- 核心模块
- 第三方模块
- 用户自定义模块

Node安装

官方地址：<https://nodejs.org/en/>

win安装

听说是傻瓜式安装

Ubuntu安装

1. 将下载好的安装包放到指定位置
2. 打开bin文件夹
3. 复制文件node的路径
4. `cd ~ | vim .bashrc`
5. 在最后添加：`export PATH=$PATH:{刚才复制的地址}` 保存退出
6. 执行 `source .bashrc` 并运行node测试安装是否完成

Node执行脚本文件

- `node {文件}`
- 注意：文件名不能用 'node.js' 命名

文件操作

读取文件

```
//1、使用require方法加载fs核心模块
var fs = require('fs')

//2、读取文件
// fs.readFile( {文件路径} , {回调函数} )
fs.readFile('test.txt',function(error,data){
    if(error){
        console.log("文件读取失败！ ")
    }else{
        console.log(data.toString())
    }
})
```

写文件

```
// 1、加载fs模块
var fs = require('fs')

//2、写文件
// fs.writeFile( {文件路径/文件名} , {文件内容} {回调函数})
//回调函数：
//     成功：文件写入成功，error是null
//     失败：文件写入失败，error是错误对象
fs.writeFile('test.txt','测试',function(error){
    if(error){
        console.log('文件写入失败')
    }else{
        console.log('文件写入成功')
    }
})
```

简单的http服务

创建web服务器

```
// 1、加载http模块
var http = require('http')
//2、使用http.createServer()方法创建一个web服务器
// 返回一个Server实例
var server = http.createServer()

//3、服务器：
//     提供服务：对数据的服务
//     发送请求
//     接收请求
//     处理请求
//     发送响应
//     注册request请求
// 当客户端请求过来，就会自动触发服务器的request请求事件，然后执行第二个参数：回调函数
// 回调函数：
//     request
//     response
server.on('request',function(request,response){
    console.log('收到客户端请求');
```

```
})  
//4、绑定端口号，启动服务器  
server.listen(8080,function(){  
  console.log('服务器启动成功，可以通过http://127.0.0.1:8080/进行访问')  
})
```

处理请求，发送响应

```
//write可以使用多次，但是最后一定要用end结束响应，否则用户端会一直等待  
server.on('request',function(request,response){  
  console.log('收到客户端请求,请求路径: '+request.url)  
  //根据不同请求路径，响应不同内容  
  var url = request.url  
  if(url=='/'){  
    //....  
  }else if(url == '/login'){  
    //....  
  }else {  
    response.end('404 Not Found.')  }  
  //发送数据的同时，结束响应  
  response.end('hello node.js')  
})
```

响应内容只能是二进制数据或者字符串

```
//...  
//响应json数组字符串  
var list=[  
  {  
    name: '苹果',  
    price: 1000  
  },  
  {  
    name: '菠萝',  
    price: 999  
  },  
  {  
    name: '香蕉',  
    price: 123  
  }  
]  
response.end(JSON.stringify(list))  
//对应的将字符串转json对象: JSON.parse('')  
//..
```

Node的模块系统

什么是模块化

- 文件作用域

- 通信规则
 - 加载 require
 - 导出 exports

CommonJs模块规范

在Node中js还有一个很重要的概念：

- 模块作用域
- 使用require方法加载模块
- 使用exports接口对象来导出模块的成员

加载 require

语法

```
var modelObj = require('model')
```

作用：

- 执行被加载模块中的代码
- 得到被加载模块中的 `exports` 导出接口对象

加载规则：

- 优先从缓存加载
 - 已经加载的不会在重新加载
 - 可以拿到exports接口对象
- 判断模块标识
 - 核心模块
 - 用户自定义
 - 第三方模块

导出 exports

- Node中是模块作用域，默认文件中所有的成员只在当前文件模块有效
- 对于希望可以被其他模块访问的成员，我们就需要把这些公开的成员挂载到exports接口对象中
 - 导出多个成员(必须在对象中):

```
exports.a = 123
exports.b = 'hello'
exports.c = function(){
  //..
}
exports.d = {
  foo: 'hello'
}
```

也可以这样拿到多个成员：

```
module.exports = function (){
  add : function(){
    //...
  },
  str : 'hello'
}
```

- 导出单个成员(拿到的是函数, 字符串)

以下情况会覆盖:

```
module.exports = 'hello'

module.exports = function (){
  //...
}
```

本质上 `exports` 等价 `module.exports`

```
//Node为了简化操作, node变量引用module.exports
var exports = module.exports
```

```
exports.foo = 'hello'
//等价于
module.exports.foo = 'hello'
```

误区解析

```
//{a:123}
exports.a = 123
```

```
//{a:123,b:'hello'}
exports.b = 'hello'
```

```
//{a:456,b:'hello'}
module.exports.a=456
```

```
//exports与module.exports脱离引用
//module.exports不变: {a:123,b:'hello'}
exports={
  a:789
}
```

```
//上同, 混淆视听
//{a:123,b:'hello'}
exports.b='world'
```

```
//重新建立引用
exports = module.exports
```

```
//{a:456,b: 'world'}
exports.b='world'
```

核心模块

Node为js提供了很多服务器级别的API，这些API绝大多数都包装到一个具体名称的核心模块中，例如：

- 文件操作的fs
- http服务的http
- url路径操作模块
- path路径处理模块
- os操作系统模块
- ...

```
var obj = require('{obj}')
```

URL模块

```
//加载  
var url = require('url')
```

- url.parse(req.url,true)

处理表单get请求的参数

```
var url = require('url')  
var parseObj = url.parse('/test?name=黄&age=99',true)  
console.log(parseObj)  
  
/* 结果  
true:将参数转化成对象  
  
Url {  
  protocol: null,  
  slashes: null,  
  auth: null,  
  host: null,  
  port: null,  
  hostname: null,  
  hash: null,  
  search: '?name=黄&age=99',  
  query: { name: '黄', age: '99' },  
  pathname: '/test',  
  path: '/test?name=黄&age=99',  
  href: '/test?name=黄&age=99'  
}  
  
*/  
  
//获取访问地址  
var pathname = parseObj.pathname
```

表单提交重定向到指定页面

如何通过服务器然客户端重定向：

1. 状态码设置为302临时重定向：statusCode
2. 状态码：301为永久重定向，浏览器会记住
3. 在响应头中通过Location告诉客户端往哪重定向：setHeader

```
//...
server.on('request',function(req,res){
  res.statusCode = 302
  res.setHeader('Location','{url}')
  res.end()
})
//...
```

Path模块

- path.basename() :获取路径上的文件名

```
//...
path.basename('c:/a/b/c/test.js')
//输出 test.js

path.basename('c:/a/b/c/test.js' , 'js')
//输出 test 去除后缀名
//...
```

- path.dirname() :获取文件的包名

```
//...
path.dirname('c:/a/b/c/test.js')
//输出 c:/a/b/c

//...
```

- path.extname :获取文件的后缀

```
path.extname('c:/a/b/c/test.js')
//输出 '.js'
```

- path.isAbsolute() :判断路径是否绝对路径

```
path.isAbsolute('c:/a/b/c/test.js')
//输出 true

path.isAbsolute('./c/test.js')
//输出 false
```

- path.parse() :解析路径

```
path.parse('c:/a/b/c/test.js')
```

```
/*  
输出：  
*/  
{  
  root: '',  
  dir: 'c:/a/b/c',  
  base: 'test.js',  
  ext: '.js',  
  name: 'test'  
}
```

- `path.join()`：拼接路径

```
path.join('c:/a/b','c')  
//输出： 'c:/a/b/c'
```

用户自定义模块

require加载其他js文件

- 用户自己编写的文件模块，相对路径必须加 `./` 可以省略后缀名，相对路径中的 `./` 不能省略
- 在node中没有全局作用域，只有文件作用域

```
require('./[文件路径]') //变量互不干扰
```

- 在每个文件模块中都提供了一个对外对象：`exports`
- `exports`默认是一个空对象
- 取用文件内对外函数或者函数：`exports.{变量}` / `exports.{函数}`

```
var obj = require('[文件路径]') //对内文件变量互不干扰  
obj.{变量/函数} //取用对外变量/函数
```

第三方模块

- `art-template`
- ...

第三方模块必须通过 `npm` 下载到项目包才可以使用

Node中的其他成员

在每个模块中，除了 `require` , `exports` 等模块相关API之外，还有两个特殊的成员

- `__dirname`：可以用来或取当前文件模块所属的目录的绝对路径

- `__filename`：获取当前文件的为绝对路径
- 在文件操作中，使用相对路径时不可靠的，因为Node中文件操作相对路径是相对于终端执行的目录路径 (`__filename` 没有这个影响)
- 为了解决以上问题，配合 `path` 模块

```
path.join(__dirname + '{filename}')
```

Response响应内容类型

设置编码

普通数据

```
//...
response.setHeader('Content-Type','text/plain; charset=utf-8') //text/plain:普通文本
response.end('{返回数据}')
//...
```

html标签

```
//...
response.setHeader('Content-Type','text/html; charset=utf-8') //text/html:渲染html文本的同时设置字符
编码
response.end('<a>点我</a>')
//...
```

html文件

1. 加载核心模块，http,fs
2. 读取html文件内容
3. 将读取的文件内容传递给响应response.end(data)

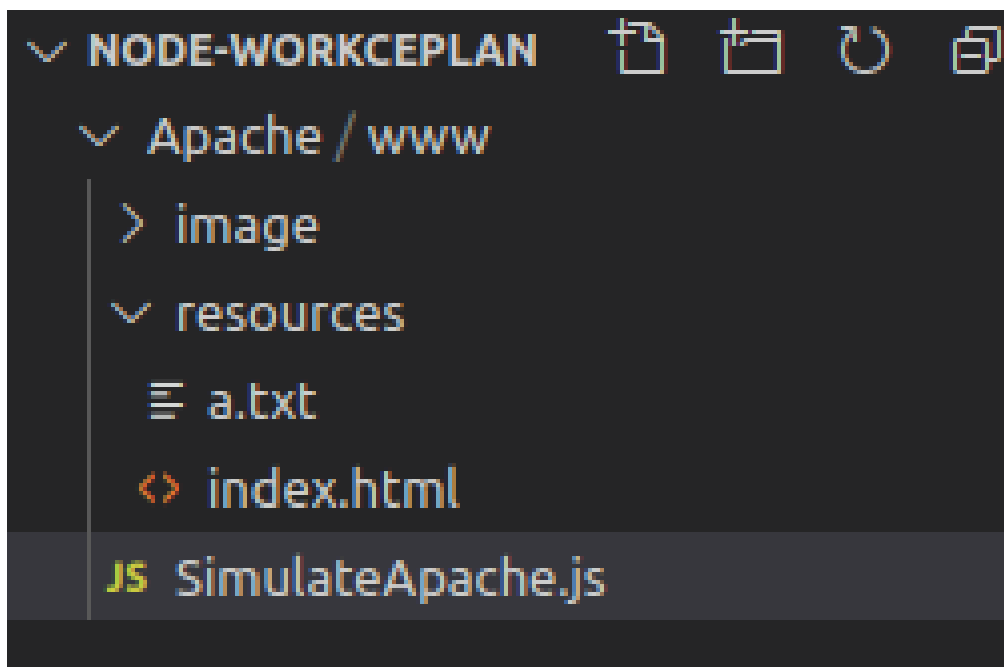
```
//...
fs.readFile('{html文件路径}', function(error,data){
  if(error){
    response.setHeader('Content-Type','text/plain; charset=utf-8')
    response.end('加载失败....')
  }else{
    response.setHeader('Content-Type','text/html; charset=utf-8')
    response.end(data) //可接收二进制或者字符
  }
})
//...
```

图片

```
//...
fs.readFile('{图片路径}', function(error,data){
  if(error){
    response.setHeader('Content-Type','text/plain; charset=utf-8')
    reponse.end('加载失败...')
  }else{
    response.setHeader('Content-Type','image/jpeg ')
    response.end(data)
  }
})
//...
```

模拟Apache

创建文件目录



创建脚本动态访问资源

1. 创建webserver
2. 设置总资源路径 `wwwDir` 变量
3. 设置默认访问路径（首页路径） `filePath` 变量
4. 添加判断 如果访问路径不是 `/` 则修改 `filePath`
5. 根据最终路径 `wwwDir` + `filePath` ,读取路径 并返回响应

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

var server = http.createServer()

server.on('request',function( req , res ){
  var wwwDir = '/home/node-workceplan/Apache/www'
  var url = req.url
  var filePath = '/resources/index.html'
  if (url !== '/'){
```

```

    filePath = url
  }

  fs.readFile(wwwDir + filePath, function(err,data){
    if (err){
      return res.end('404 Not Found.')
    }
    res.end(data)
  })
})

server.listen(8080, function(){
  console.log('服务器启动成功，可以通过http://127.0.0.1:8080/进行访问')
})

```

在Node中使用模板引擎

art-template模板引擎

```

//加载
var template = require('art-template')

var html='
<h1>{{ name }}</h1>
'
var ret = template.render(html, {
  name: 'hdl'
})
console.log(ret)

```

读取html文件

```

var template = require('art-template')
var fs = require('fs')

fs.readFile('{html文件路径}', function(err,data){
  if(err){
    return console.log('读取失败')
  }
  var ret = template.render(data.toString(),{
    //{key}:{value}
  })
})

```

使用模板完成目录列表渲染

1. 前期：创建 webserver
2. 前期：安装 art-template `npm install art-template`
3. 前期：html文件引入模板引擎文件并编写ui
4. 加载模块（fs、art-template、http）
5. 读取指定文件目录 `fs.readdir(url,function(err,files){})`
6. 判断目录还是文件 (未实现)
7. 读取html文件

8. 将读取目录的数据解析替换到html文本
9. 发回响应

```
var http = require('http')
var fs = require('fs')
var template = require('art-template')
var wwwDir = '/home/node-workceplan/SimulateApache/www'
http.createServer().on('request',function(req,res){
  fs.readdir(wwwDir,function(err,files){

    if (err){
      res.setHeader('Content-Type','text/plain; charset=utf-8')
      return res.end('读取文件路径异常: '+err)
    }
    fs.readFile(wwwDir+'/view/dir.html',function(err,data){
      if(err){
        return res.end('404 Not Found.')
      }
      var htmlStr = template.render(data.toString(),{
        title:wwwDir+req.url,
        files:files
      })
      res.end(htmlStr)
    })
  })
})
.listen(8080,function(){
  console.log('服务器启动成功，可以通过http://127.0.0.1:8080/进行访问')
})
```

html文件：

```
<html>
<head>

  <meta charset="utf-8"/>
  <title>{{ title }}</title>
</head>
<body>
{{ each files}}
<a href="#">{{ $value }}</a><br/>
{{/each}}
</body>
</html>
```

art-template引入其他页面

```
{{ include './*.html' }}
```

npm命令

npm网站

- npmjs.com

npm命令行工具

- 安装 `node`，默认就安装了 `npm`
- 升级 `npm`：`npm install --global npm`
- 常用命令：
 - `npm init -y`：可以跳过向导，快速生成
 - `npm install package`：下载包
 - `npm install --save`：下载包的同时，保存依赖项在`package.json`文件中
 - `npm uninstall package --save`：删除的同时把依赖项去除
 - `npm help`：查看帮助
 - `npm 命令 --help`：查看指定命令帮助
 - `npm config list`：查看配置信息
- 解决npm存储包被墙问题：
 - 安装 `cnpm`：`npm install --global cnpm` (`--global`表示全局)
 - 设置淘宝服务器：`npm config set registry https://registry.npm.taobao.org`

package.json文件

- 描述项目基本信息和依赖的第三方包
- 可以通过 `npm init` 的方式自动初始化出来
- 建议执行 `npm install` 包名的时候加上 `--save` 选项，用来保存依赖包信息
- 如果 `node_modules` 文件夹删除了，运行 `npm install` 会自动把 `package.json` 中的 `dependencies` 中所有依赖项下载回来

package-lock.json文件

- 新版 `npm` 安装包时自动生成（`npm 5` 以后）
- `npm 5` 以后的版本安装包不用添加 `--save` 也会保存依赖
- 保存 `node_modules` 中所有包的信息（版本，下载地址）
 - 这样重新 `npm install` 的时候速度会提升
- 锁定版本

Express框架 For Node Web

起步

安装

```
npm i -S express
```

使用

```
var express = require('express')

//创建服务器
var app = express()
//指定公开目录
app.use( '/public/', express.static('./public/'))

//处理 '/' 请求
app.get('/',function(req,res){
  res.send('hello express')
})
app.listen(8080,function(){
  console.log('app is running at port 8080')
})
```

静态服务-指定公开目录

```
app.use( '/public/', express.static('./public/')) //访问路径需要添加 '/public' /

app.use( express.static('./public/')) //访问路径不需要添加 '/public' /
```

基本路由

路由器：

- 请求方法
- 请求路径
- 请求处理函数

get

```
app.get(url,function)
```

在express获取get请求体数据

```
// express内置api
app.get(url,function(req,res){
  req.query()
})
```

post

```
app.post(url,function)
```

在express获取post请求体数据

在Express中没有内置获取表单post请求体的api，需要用到 一个第三方包： `body-parser`

- 安装：

```
npm install -S body-parser
```

- 配置：

```
// 1、引包
var bodyParser = require('body-parser')

// 2、配置
// 一定要在挂载路由app.use(router)之前配置
// 只需加入这个配置，request请求对象会多出一个属性：req.body
// 通过req.body获取post请求表单数据
app.use(bodyParser.urlencoded({ extended: false }))
app.use(bodyParser.json())
```

- 获取数据：

```
app.post(url, function(req, res){
  req.body
})
```

修改代码自动重启-热部署

- 第三方命名航工具: nodemon
- nodemon 是一个Node.js开发的一个第三方命令行工具，需要独立安装

安装

```
npm install --global nodemon
```

使用

```
node app.js

#使用 nodemon启用js文件
nodemon app.js
```

在 Express中使用 art-template模板

安装

```
npm install --save art-template
npm install --save express-art-template

#npm install --save art-template express-art-template
```

配置

```
//配置使用模板引擎
/*
  第一个参数：表示 当渲染以 .art 结尾的文件时候，使用art-template模板引擎，
  第二个参数：
*/
app.engine('art',require('express-art-template'))

//如果想要修改默认的view目录，则可以
app.set('views',{目录路径})
```

使用

- Express 为 Response 响应对象提供一个方法： render
- render 方法默认是不可以使用，但是如果配置了模板引擎就可以使用
- res.render 方法第一个参数不能写路径，默认会去项目中的 view 目录查找该模板文件（以 .art 后缀结尾的文件或者*.html）

```
app.get(url,function(req,res){
  res.render({art/html文件名},{
    {key} : {value}
  })
})
```

路由模块提取

创建 router.js 文件

- 将路由的请求方法封装到该文件

app.js启动文件引用router.js文件

1. require('./router')

```
var router = require('./router')
router(app)
```

使用这种办法，router.js文件就要特殊处理

```
//router.js
module.exports = function(app){
  app.get(url,function)
  //...
}
```

2. express内置api

```
// router.js
//加载express
```



```

var express = require('express')

//1 创建路由容器
var router = express.Router()

//2 把路由挂载在路由容器
router.get(url,function)
//....

//3 导出路由容器
module.exports = router

```

把路由容器挂载到app服务中

```

//app.js
//...
var router = require('./router.js')
app.use(router)
//...

```

文件操作模块提取

```

//fileOperation.js
var fs = require('fs')
var filePath = '{path}'
/**
 获取所有

 解决异步读取文件返回不到数据

 外部文件调用
  var fileOperation = require('./fileOperation')

  fileOperation.findAll(function(err,data){
    if(err){
      return //...
    }
    //...
  })
  */

  exports.findAll = function(callback){
    fs.readFile(filePath,'utf8',function(err,data){
      if(err){
        return callback(err)
      }
      callback(null,JSON.parse(data).dataObj)
    })
  }
  /**
  根据id查找
  外部文件调用
  var fileOperation = require('./fileOperation')

```

```

fileOperation.findById(parseInt(req.query.id),function(err,data){
    if(err){
        return //...
    }
    //...
})

*/
exports.findById = function(id,callback){
    fs.readFile(filePath,'utf8',function(err,data){
        if(err){
            return callback(err)
        }
        //解析
        var JsonData = JSON.parse(data).dateObj
        //当某个遍历项符合，则返回
        var TargetData = JsonData.find(function(item){
            return item.id = modifyData.id
        })
        callback(null,TargetData)
    })
}
}
/**
添加

```

- 1、先读取原文件数据
- 2、将读取到的string数据转（list）json对象
- 3、转换到的json对象加进要写入的（formdata）数据
- 4、重新写入文件
- 5、传递结果给回调函数

外部文件调用

```
var fileOperation = require('./fileOperation')
```

```

fileOperation.sava(FormData, function(err){
    if(err){
        return //...
    }
    //...
})

*/
exports.sava = function(FormData,callback){
    fs.readFile(filePath,function(err,data){
        if(err){
            return callback(err)
        }
        //解析
        var JsonData = JSON.parse(data).dateObj
        // id自增长
        FormData.id = JsonData[JsonData.length - 1].id + 1
        //添加
        JsonData.push(FormData)
        //将list转string
        var FileData = JSON.stringify({
            dateObj : JsonData
        })
        //重新写入

```

```

    fs.writeFile(filePath, FileData, function(err){
        if(err){
            return callback(err)
        }
        callback(null)
    })
})
}
/**
更新
*/
exports.updateById = function(modifyData, callback){
    fs.readFile(filePath, 'utf8', function(err, data){
        if(err){
            return callback(err)
        }
        //解析
        var JsonData = JSON.parse(data).dateObj
        //修改
        //esc 6 数组方法：find
        //需要接收一个函数作为参数
        //当某个遍历项符合，则返回
        var TargetData = JsonData.find(function(item){
            return item.id = modifyData.id
        })
        //遍历拷贝对象
        for(var key in modifyData){
            TargetData[key] = modifyData[key]
        }
        //将list转string
        var FileData = JSON.stringify({
            dateObj: JsonData
        })
        //重新写入
        fs.writeFile(filePath, FileData, function(err){
            if(err){
                return callback(err)
            }
            callback(null)
        })
    })
}
/**
删除
*/
exports.delete = function(){
    fs.readFile(filePath, function(){

    })
}
}

```

异步编程

异步函数

- setTimeout
- readFile
- writeFile
- readdir
- ajax
- ...

回调函数

函数中调用异步函数就获取不到异步函数返回的数据，此时只能采用回调函数：

1. 将函数以参数传递给母函数，
2. 异步函数得到的结果数据再传递给回调函数

```
function master(callback){
  //异步函数：例子：
  fs.readFile({path},function(err,data){
    if(err){
      return callback(err)
    }
    callback(null,data.toString())
  })
}

//回调函数使用
master(function(err,data){
  if(err){
    //...
  }
  //...
})
```

基于原生XMLHttpRequest封装get方法

```
function get(url,callback){
  var oReq = new XMLHttpRequest()
  //异步请求
  oReq.onload = function(){
    callback(oReq.responseText)
  }
  oReq.open("get",url,true)
  oReq.send()
}

//调用
get('{url}',function(data){
  //...
})
```

Promise

- callback-hell (回调地狱)：回调函数依赖上一个回调函数的结果，为了有序运行，该回调函数会嵌套在上一个回调函数，由此，带来了后期可维护低，代码难看
- 为了解决以上方式带来的问题（回调地狱），EcmaScript中新增一个API：`promise`

```
var fs = require('fs')
//创建容器
var p1= new Promise(function (resolve , reject){
  fs.readFile('{path}','utf8',function(err,data){
    if (err){
      //把容器的pending状态变为reject
      reject(err)
    }else{
      //把容器的pending状态变为resolve
      resolve(data)
    }
  })
})

var p2= new Promise(function (resolve , reject){
  fs.readFile('{path}','utf8',function(err,data){
    if (err){
      //把容器的pending状态变为reject
      reject(err)
    }else{
      //把容器的pending状态变为resolve
      resolve(data)
    }
  })
})

//当p1成功，然后做指定的操作
p1.then(function(data){
  //p1读取成功
  return p2
},function(err){
  console.log('读取失败')
})

//指定p2相应结果做什么操作
p2.then(function(data){
  //p2读取成功
  console.log(data)
},function(err){
  console.log('读取失败')
})
```

- 封装promise-readFile-API

```
var fs = require('fs')
function PromiseReadFile(filepath){
  return new Promise(function (resolve , reject){
    fs.readFile(filepath,'utf8',function(err,data){
      if (err){
        reject(err)
      }else{
        resolve(data)
      }
    })
  })
}
```

```

    }
  })
})
}

PromiseReadFile('{path1}')
  .then(function(data){
    console.log(data)
    return PromiseReadFile('{path2}')
  })
  .then(function(data){
    console.log(data)
    return PromiseReadFile('{path3}')
  })
  .then(function(data){
    console.log(data)
  })
})

```

其他

代码分格-分号

当代码采用了舞分号的代码风格的时候，只需注意以下情况：

当一行代码是以：(、[、`、开头的时候补上一个分号就会避免一些语法解析错误

```

function say(){
  //...
}
say()
//以 '(' 开头
;(function () {
  //...
})();

//以 '[' 开头
;[ 'a', 'b' ].forEach(function (item) {
  //..
})

// 以 ''' 反引号开头
// ` 是EcmaScript 6 中新增的一种字符串包裹方式，叫做：模板字符，
// 它支持换行和非常方便拼接变量（写什么样就输出什么样）
var str=`
大家好
hello      node.js
world
`

;`hello`.toString()

```

文件操作路径和模块操作路径

- 文件操作中的相对路径可以省略 `./`
- `/`在文件操作的相对路径中
 - `./data/a.txt` 相对于当前目录
 - `data/a.txt` 相对于当前目录
 - `/data/a.txt` 绝对路径，当前脚本文件所处的磁盘空间
 - `c:/data/a.txt` 绝对路径

```
//文件所使用的所有文件操作的API都是异步的
fs.readFile('data/a.txt',callback)
```

- 在模块加载中，相对路径中的 `./` 不能省略
- 如果忽略，则是磁盘空间的绝对地址

```
require('./data/foo.js')
```

加载json文件并获取数据

```
//...
// 读取json文件，设置utf8格式，不用再data.toString()
//读取失败，返回500状态码
//读取成功，利用模板引擎将数据渲染到html文件
fs.readFile('{json文件}','utf8',function(err,data){
  if(err){
    return res.status(500).send('Server error.')
  }
  var dataObj = JSON.parser(data).dataObj

  res.render('{html文件}',{
    dataObj: dataObj
  })
})
```

JS数组操作api-EcmaScript

- `find` : 遍历返回目标对象
- `findIndex` : 遍历返回目标对象的下标
- `forEach` : 遍历
- `every` : 返回所有对象都满足 `true|false`
- `some` : 返回有对象满足 `true|false`
- `includes` : 包含判断 `true|false`
- `map` :
- `reduce` :
- ...

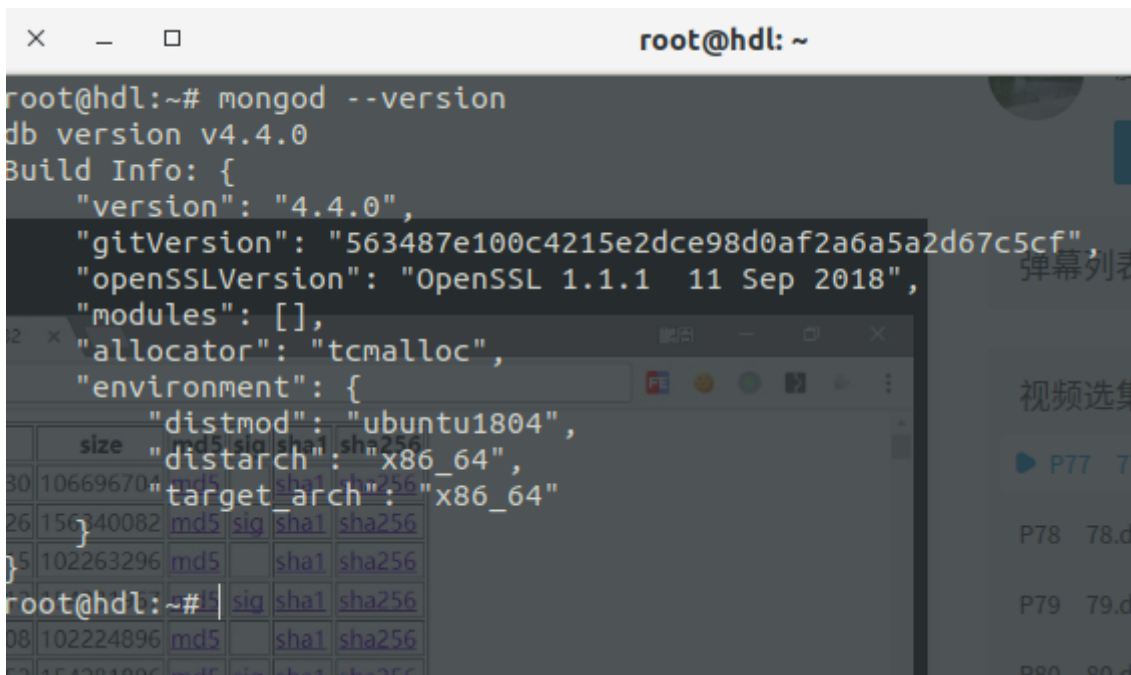
MongoDB

安装

[帮助文档](#)

- 配置环境变量

安装完成：



```
root@hdl: ~  
root@hdl:~# mongod --version  
db version v4.4.0  
Build Info: {  
  "version": "4.4.0",  
  "gitVersion": "563487e100c4215e2dce98d0af2a6a5a2d67c5cf",  
  "opensslVersion": "OpenSSL 1.1.1 11 Sep 2018",  
  "modules": [],  
  "allocator": "tcmalloc",  
  "environment": {  
    "distmod": "ubuntu1804",  
    "distarch": "x86_64",  
    "target_arch": "x86_64"  
  }  
}
```

启动和关闭

```
#开启  
mongod  
  
#关闭  
#ctrl + c  
  
#修改默认的数据存储目录  
mongod --dbpath = $path
```

连接数据库

- 开启数据库
- 连接

```
#执行命令  
mongo
```

- 退出

```
exit
```

基本命令

- `show dbs` : 查看显示所有数据库
- `db` : 查看当前操作的数据库
- `use dbname` : 切换到指定的数据库(没有则新建)
- `db.dataCollection.insertOne({key: value})` : 插入数据
- `show collections` : 查看当前数据库的集合
- `db.collection.find()` : 查询该集合的所有数据

MongoDB数据库的基本概念

- 数据库
- 集合
- 文档
- 文档结构灵活

```
{
  qq: {
    friend: {
      {},
      {},
      ....
    },
    msg: {

    },
    .....
  },
  wechat: {
    ...
  }
}
```

在Node操作mongodb

[官方文档](#)

使用第三方包mongoose来操作mongodb

- 基于MongoDB官方的 `mongo` 包再一次封装
- [官网](#)
- [官方api文档](#)
- 安装

```
npm i -S mongoose
```

连接

```
var mongoose = require('mongoose');
var Schema = mongoose.Schema
```

```
// 1、连接
mongoose.connect('mongodb://localhost:27017/hdl',{useNewUrlParser: true, useUnifiedTopology:
true});

//2、设计集合结构（表结构），
var userSchema = new Schema({
  username : {
    type : String,
    required : true //不为空
  },
  password : {
    type : String,
    required : true
  },
  email : String
})

// 3、将文档结构发布为模型
/**
 * 第一个参数：传入1一个大写名词单数字符串来表示数据库名称
 *             mongoose 会自动 将大写名词 的字符生成 小写复数 的集合名称
 *             例如：User -> users
 * 第二个参数： 架构 Schema
 * 返回值： 模型构造函数
 * */
var User = mongoose.model("User",userSchema)

// 4、得到模块构造函数，操作增删改查
```

增加数据

```
//4.1、增加数据
var admin = new User({
  username : 'admin',
  password : '123456',
  email : '123456@qq.com'
})

//4.1.2 持久化
admin.save(function(err,ret){
  if(err){
    console.log(err)
  }else {
    console.log("保存成功")
    console.log(ret)
  }
})
```

查询数据

```
//4.2 查询数据 -查询所有
User.find(function(err,ret){
  if(err){
    console.log("查询失败")
  }else {
```

```
    console.log(ret)
  }
})

// 4.2条件查询
User.find({
  username: "admin",
  password: "123456"
}, function(err, ret){
  if (err){
    console.log("查询失败")
  } else {
    console.log(ret)
  }
})
```

删除数据

```
//4.3 删除数据
User.remove({
  username: 'admin'
}, function(err){
  if (err){
    console.log('delete fail')
  } else {
    console.log('delete success')
  }
})

//根据条件删除一个
//Model.findOneAndRemove(conditions,[options],[callback])

//根据id删除
//Model.findByIdAndRemove(id,[options],[callback])
```

更新数据

```
//4.4 更新数据
User.findByIdAndUpdate('5f30fd65bd10c1557c1554b6',{
  password: '123'
}, function(err){
  if (err){
    return console.log('更新失败')
  }
  console.log('更新成功')
})
```

在Node操纵Mysql数据库

安装

```
npm i -S mysql
```

连接

```
var mysql = require('mysql')

/**
 * 1、创建连接
 * 2、连接数据库
 * 3、操纵数据库
 * 4、关闭数据库
 */
var connection = mysql.createConnection({
  host: 'localhost',
  user: 'root',
  password: '970612',
  database: 'nodeDB'
})

connection.connect()

connection.query('{sql}', function(err, res, fields){
  if(err) throw err
  console.log(res)
})

connection.end()
```