

# Node介绍

---

## 为什么要学习Node.js

---

- 企业需求
  - 具有服务端开发经验更改
  - front-end
  - back-end
  - 全栈开发工程师
  - 基本的网站开发能力
    - 服务端
    - 前端
    - 运维部署
  - 多人社区

## Node.js是什么

---

- Node.js是JavaScript 运行时
- 通俗易懂的讲，Node.js是JavaScript的运行平台
- Node.js既不是语言，也不是框架，它是一个平台
- 浏览器中的JavaScript
  - EcmaScript
    - 基本语法
    - if
    - var
    - function
    - Object
    - Array
  - Bom
  - Dom
- Node.js中的JavaScript
  - 没有Bom， Dom
  - EcmaScript
  - 在Node中这个JavaScript执行环境为JavaScript提供了一些服务器级别的API
    - 例如文件的读写
    - 网络服务的构建
    - 网络通信
    - http服务器
- 构建与Chrome的V8引擎之上
  - 代码只是具有特定格式的字符串
  - 引擎可以认识它，帮你解析和执行
  - Google Chrome的V8引擎是目前公认的解析执行JavaScript代码最快的
  - Node.js的作者把Google Chrome中的V8引擎移植出来，开发了一个独立的JavaScript运行时环境

- Node.js uses an event-driven, non-blocking I/O mode that makes it lightweight and efficient.
  - event-driven 事件驱动
  - non-blocking I/O mode 非阻塞I/O模型（异步）
  - lightweight and efficient. 轻量和高效
- Node.js package ecosystem, npm, is the largest ecosystem of open source libraries in the world
  - npm 是世界上最大的开源生态系统
  - 绝大多数JavaScript相关的包都存放在npm上，这样做的目的是为了让开发人员更方便的去下载使用
  - npm install jquery

## Node能做什么

---

- web服务器后台
- 命令行工具
  - npm(node)
  - git(c语言)
  - hexo (node)
  - ...
- 对于前端工程师来讲，接触最多的是它的命令行工具
  - 自己写的很少，主要是用别人第三方的
  - webpack
  - gulp
  - npm

## 起步

---

### 安装Node环境

---

- 查看Node环境的版本号
- 下载: <https://nodejs.org/en/>
- 安装:
  - 傻瓜式安装，一路 next
  - 安装过再次安装会升级
- 确认Node环境是否安装成功
  - 查看node的版本号: `node --version`
  - 或者 `node -v`
- 配置环境变量

### 解析执行JavaScript

---

1. 创建编写JavaScript脚本文件
2. 打开终端，定位脚本文件的所属目录
3. 输入 `node` 文件名 执行对应的文件

注意：文件名不要用 `node.js` 来命名，也就是说除了 `node` 这个名字随便起，最好不要使用中文。

### 文件的读写

---

文件读取:

```

1 //浏览器中的JavaScript是没有文件操作能力的
2 //但是Node中的JavaScript具有文件操作能力
3 //fs是file-system的简写，就是文件系统的意思
4 //在Node中如果想要进行文件的操作就必须引用fs这个核心模块
5 //在fs这个和兴模块中，就提供了人所有文件操作相关的API
6 //例如 fs.readFile就是用来读取文件的
7
8 // 1.使用fs核心模块
9 var fs = require('fs');
10
11 // 2.读取文件
12 fs.readFile('./data/a.txt',function(err,data){
13     if(err){
14         console.log('文件读取失败');
15     }
16     else{
17         console.log(data.toString());
18     }
19 })

```

文件写入：

```

1 // 1.使用fs核心模块
2 var fs = require('fs');
3
4 // 2.将数据写入文件
5 fs.writeFile('./data/a.txt','我是文件写入的信息',function(err,data){
6     if(err){
7         console.log('文件写入失败');
8     }
9     else{
10         console.log(data.toString());
11     }
12 })

```

## http

服务器：

```

1 // 接下来，我们要干一件使用 Node 很有成就感的一件事儿
2 // 你可以使用 Node 非常轻松的构建一个 web 服务器
3 // 在 Node 中专门提供了一个核心模块：http
4 // http 这个模块的职责就是帮你创建编写服务器的
5
6 // 1. 加载 http 核心模块
7 var http = require('http')
8
9 // 2. 使用 http.createServer() 方法创建一个 web 服务器
10 // 返回一个 Server 实例
11 var server = http.createServer()
12
13 // 3. 服务器要干嘛？
14 // 提供服务：对 数据的服务
15 // 发请求
16 // 接收请求
17 // 处理请求

```

```

18 // 给个反馈（发送响应）
19 // 注册 request 请求事件
20 // 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：回调处理函数
21 server.on('request', function () {
22   console.log('收到客户端的请求了' + request.url())
23 })
24
25 // 4. 绑定端口号，启动服务器
26 server.listen(3000, function () {
27   console.log('服务器启动成功了，可以通过 http://127.0.0.1:3000/ 来进行访问')
28 })
29

```

带有响应的服务器：

```

1 server.on('request', function (request, response) {
2   console.log('收到请求了');
3   response.write('hello');//写入相应内容
4   response.end();//注意一定要加end，表示相应结束
5   //上述两行可以直接写成response.end('hello')
6 });

```

根据不同路径返回不同响应：

```

1 // 根据不同的请求路径发送不同的响应结果
2 // 1. 获取请求路径
3 // req.url 获取到的是端口号之后的那一部分路径
4 // 也就是说所有的 url 都是以 / 开头的
5 // 2. 判断路径处理响应
6
7 var url = req.url
8
9 if (url === '/') {
10   res.end('index page')
11 } else if (url === '/login') {
12   res.end('login page')
13 } else if (url === '/products') {
14   var products = [{
15     name: '苹果 x',
16     price: 8888
17   },
18   {
19     name: '菠萝 x',
20     price: 5000
21   },
22   {
23     name: '小辣椒 x',
24     price: 1999
25   }
26 ]
27 // 响应内容只能是二进制数据或者字符串
28 // 数字、对象、数组、布尔值都需要转换
29 res.end(JSON.stringify(products))//转换成字符串
30 } else {
31   res.end('404 Not Found.')

```

```
32 | }
```

响应内容类型:

```
1  if (url === '/plain') {
2      // text/plain 就是普通文本
3      res.setHeader('Content-Type', 'text/plain; charset=utf-8')
4      res.end('hello 世界')
5  } else if (url === '/html') {
6      // 如果你发送的是 html 格式的字符串, 则也要告诉浏览器我给你发送是 text/html 格式的
    容
7      res.setHeader('Content-Type', 'text/html; charset=utf-8')
8      res.end('<p>hello html <a href="">点我</a></p>')
9  }
```

## 服务器自动重启工具

```
1 | npm install --global nodemon
```

下载之后通过 `nodemon test.js` 启动服务器, 只要对文件进行保存, 就可以自动重启

## Node中的模块系统

使用Node编写应用程序主要就是在使用:

- EcmaScript语言
  - 和浏览器一样, 在Node中没有Bom和Dom
- 核心模块
  - 文件操作的fs
  - http服务操作的http
  - url路径操作模块
  - path路径处理模块
  - os操作系统信息
- 第三方模块
  - art-template
  - 必须通过npm来下载才可以使用
- 自己写的模块
  - 自己创建的文件

**提到核心模块, 立刻想到使用它, 必须:**

```
1 | var xxx = require('xxx');
```

## 什么是模块化

- 文件作用域(模块是独立的，在不同的文件使用必须要重新引用)【在node中没有全局作用域，它是文件模块作用域】
- 通信规则
  - 加载require
  - 导出exports

## CommonJS模块规范

在Node中的JavaScript还有一个重要的概念，模块系统。

- 模块作用域
- 使用require方法来加载模块
- 使用exports接口对象来导出模板中的成员

### 加载require

语法：

```
1 | var 自定义变量名 = require('模块')
```

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

### 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模块有效
- **对于希望可以被其他模块访问到的成员，我们需要把这些公开的成员都挂载到 exports 接口对象中就可以了，注意exports是一个对象，exports.a 和 var a不一样**

导出多个成员（必须在对象中）：

```
1 | exports.a = 123;  
2 | exports.b = function(){  
3 |     console.log('bbb')  
4 | };  
5 | exports.c = {  
6 |     foo:"bar"  
7 | };  
8 | exports.d = 'hello';
```

导出**单个**成员（此时拿到的就是函数，字符串）：

```
1 | module.exports = 'hello';
```

以下情况会覆盖：

```
1 module.exports = 'hello';
2 //后者会覆盖前者
3 module.exports = function add(x,y) {
4     return x+y;
5 }
```

也可以通过以下方法来导出多个成员:

```
1 module.exports = {
2     foo = 'hello',
3     add:function(){
4         return x+y;
5     }
6 };
```

## 模块原理

exports只是module.exports 的一个引用:

```
1 console.log(exports === module.exports);    //true
2
3 exports.foo = 'bar';
4
5 //等价于
6 module.exports.foo = 'bar';
```

当给exports重新赋值后, exports! = module.exports.

**最终return的是module.exports**, 无论exports中的成员是什么都没用。

```
1 真正去使用的时候:
2     导出单个成员: exports.xxx = xxx;
3     导出多个成员: module.exports 或者 module.exports = {};
```

## 总结

```
1 // 引用服务
2 var http = require('http');
3 var fs = require('fs');
4 // 引用模板
5 var template = require('art-template');
6 // 创建服务
7 var server = http.createServer();
8 // 公共路径
9 var wwwDir = 'D:/app/www';
10 server.on('request', function (req, res) {
11     var url = req.url;
12     // 读取文件
13     fs.readFile('./template-apche.html', function (err, data) {
14         if (err) {
15             return res.end('404 Not Found');
16         }
17         fs.readdir(wwwDir, function (err, files) {
18             if (err) {
```

```

19         return res.end('Can not find www Dir.')
20     }
21     // 使用模板引擎解析替换data中的模板字符串
22     // 去xmpTemplateList.html中编写模板语法
23     var htmlStr = template.render(data.toString(), {
24         title: 'D:/app/www/ 的索引',
25         files: files
26     });
27     // 发送响应数据
28     res.end(htmlStr);
29 }
30 })
31 });
32 server.listen(3000, function () {
33     console.log('running....');
34 })

```

- 1 1. jQuery中的each 和 原生JavaScript方法forEach的区别:  
2 提供源头:  
3 原生js是es5提供的（不兼容IE8）,  
4 jQuery的each是jQuery第三方库提供的（如果要使用需要用2以下的版本也就是1.版本）,  
它的each方法主要用来遍历jQuery实例对象（伪数组）,同时也可以做低版本forEach的替代品,jQuery的实例对象不能使用forEach方法,如果想要使用必须转为数组  
（[].slice.call(jQuery实例对象)）才能使用
- 5 2. 模块中导出多个成员和导出单个成员
- 6 3. 301和302的区别:  
7 301永久重定向,浏览器会记住  
8 302临时重定向
- 9 4. exports和module.exports的区别:  
10 每个模块中都有一个module对象  
11 module对象中有一个exports对象  
12 我们可以把需要导出的成员都挂载到module.exports接口对象中  
13 也就是`module.exports.xxx = xxx`的方式  
14 但是每次写太多了就很麻烦,所以Node为了简化代码,就在每一个模块中都提供了一个成员叫`exports`  
15 `exports === module.exports`结果为true,所以完全可以`exports.xxx = xxx`  
16 当一个模块需要导出单个成员的时候必须使用`module.exports = xxx`的方式,`,使用`exports = xxx`不管用,因为每个模块最终return的是module.exports,而exports只是module.exports的一个引用,所以`exports`即使重新赋值,也不会影响`module.exports`。  
17 有一种赋值方式比较特殊: `exports = module.exports`这个用来新建立引用关系的。  
18

## require的加载规则

- 核心模块
  - 模块名
- 第三方模块
  - 模块名
- 用户自己写的
  - 路径

## require的加载规则:



- 优先从缓存加载
- 判断模块标识符
  - 核心模块
  - 自己写的模块（路径形式的模块）
  - 第三方模块（node\_modules）
    - 第三方模块的标识就是第三方模块的名称（不可能有第三方模块和核心模块的名字一致）
    - npm
      - 开发人员可以把写好的框架库发布到npm上
      - 使用者通过npm命令来下载
    - 使用方式：`var 名称 = require('npm install【下载包】的包名')`
      - node\_modules/express/package.json main
      - 如果package.json或者main不成立，则查找被选择项：index.js
      - 如果以上条件都不满足，则继续进入上一级目录中的node\_modules按照上面的规则依次查找，直到当前文件所属此盘根目录都找不到最后报错

```

1 // 如果非路径形式的标识
2 // 路径形式的标识：
3 // ./ 当前目录 不可省略
4 // ../ 上一级目录 不可省略
5 // /xxx也就是D:/xxx
6 // 带有绝对路径几乎不用（D:/a/foo.js）
7 // 首位表示的是当前文件模块所属磁盘根目录
8 // require('./a');
9
10
11 // 核心模块
12 // 核心模块本质也是文件，核心模块文件已经被编译到了二进制文件中了，我们只需要按照名字来加载就可以了
13 require('fs');
14
15 // 第三方模块
16 // 凡是第三方模块都必须通过npm下载（npm i node_modules），使用的时候就可以通过
17 // require('包名')来加载才可以使用
18 // 第三方包的名字不可能和核心模块的名字是一样的
19 // 既不是核心模块，也不是路径形式的模块
20 // 先找到当前文所属目录的node_modules文件夹
21 // 然后在node_modules文件夹中找required包名文件夹
22 // 即找node_modules/art-template文件夹
23 // 在node_modules/art-template文件夹中找package.json
24 // node_modules/art-template/package.json中的main属性
25 // main属性记录了art-template的入口模块
26 // 然后加载使用这个第三方包
27 // 实际上最终加载的还是文件
28
29 // 如果package.json不存在或者mian指定的入口模块不存在
30 // 则node会自动找该目录下的index.js
31 // 也就是说index.js是一个备选项，如果main没有指定，则加载index.js文件
32 //
33 // 如果条件都不满足则会进入上一级目录进行查找
34 // 注意：一个项目只有一个node_modules，放在项目根目录中，子目录可以直接调用根目录的文件

```

```
34 var template = require('art-template');
35
```

## 模块标识符中的 / 和文件操作路径中的 /

文件操作路径：

```
1 // 咱们所使用的所有文件操作的API都是异步的
2 // 就像ajax请求一样
3 // 读取文件
4 // 文件操作中 ./ 相当于当前模块所处磁盘根目录
5 // ./index.txt 相对于当前目录
6 // /index.txt 相对于当前目录
7 // /index.txt 绝对路径,当前文件模块所处根目录
8 // d:express/index.txt 绝对路径
9 fs.readFile('./index.txt',function(err,data){
10     if(err){
11         return console.log('读取失败');
12     }
13     console.log(data.toString());
14 })
```

模块操作路径：

```
1 // 在模块加载中, 相对路径中的./不能省略
2 // 这里省略了./也是磁盘根目录
3 require('./index')('hello')
```

## 模板引擎

1. 安装 npm install art-template
2. 在需要使用的文件模块中加载 art-template

只需要使用 require 方法加载就可以了：require('art-template')

参数中的 art-template 就是你下载的包的名字

也就是说你 install 的名字是什么，则你 require 中的就是什么

3. 查文档，使用模板引擎的 API

**模板引擎不关心内容，只关心模板标记语法**

下面贴一个简单的服务端渲染实例，主要分为以下几步：

- 编写好html页面，需要替换的地方用模板引擎语法编写
- 在js文件中引入模板引擎，创建服务器，读取html页面，读取文件夹内容，作为数据传给模板引擎进行替换
- 最后通过渲染展示html页面

```
1 <html dir="ltr" lang="zh" i18n-processed="">
2
3 <head>
4   <meta charset="utf-8">
```

```

5     <meta name="google" value="notranslate">
6     <title id="title">{{ title }}</title>
7 </head>
8
9 <body>
10    <div id="listingParsingErrorBox">糟糕! Google Chrome无法解读服务器所发送的数据。
    请<a href="http://code.google.com/p/chromium/issues/entry">报告错误</a>, 并附上
    <a href="LOCATION">原始列表</a>。</div>
11    <h1 id="header">D:\Movie\www\ 的索引</h1>
12    <div id="parentDirLinkBox" style="display:none">
13        <a id="parentDirLink" class="icon up">
14            <span id="parentDirText">[上级目录]</span>
15        </a>
16    </div>
17    <table>
18        <thead>
19            <tr class="header" id="thead">
20                <th onclick="javascript:sortTable(0);">名称</th>
21                <th class="detailsColumn" onclick="javascript:sortTable(1);">
22                    大小
23                </th>
24                <th class="detailsColumn" onclick="javascript:sortTable(2);">
25                    修改日期
26                </th>
27            </tr>
28        </thead>
29        <tbody id="tbody">
30            {{each files}}
31            <tr>
32                <td data-value="apple/"><a class="icon dir"
    href="/D:/Movie/www/apple/">{{ $value }}</a></td>
33                <td class="detailsColumn" data-value="0"></td>
34                <td class="detailsColumn" data-value="1509589967">2017/11/2 上午
    10:32:47</td>
35            </tr>
36            {{/each}}
37        </tbody>
38    </table>
39 </body>
40
41 </html>
42

```

```

1  var http = require('http')
2  var fs = require('fs')
3  var template = require('art-template')
4
5  var server = http.createServer()
6
7  var wwwDir = 'E:/Everything'
8
9  server.on('request', function (req, res) {
10      var url = req.url
11      fs.readFile('./template-apache.html', function (err, data) {
12          if (err) {
13              return res.end('404 Not Found.')
14          }

```

```

15 // 1. 如何得到 wwwDir 目录列表中的文件名和目录名
16 // fs.readdir
17 // 2. 如何将得到的文件名和目录名替换到 template.html 中
18 // 2.1 在 template.html 中需要替换的位置预留一个特殊的标记（就像以前使用模板引擎的标记一样）
19 // 2.2 根据 files 生成需要的 HTML 内容
20 fs.readdir(wwwDir, function (err, files) {
21   if (err) {
22     return res.end('Can not find www dir.')
23   }
24
25   var htmlStr = template.render(data.toString(), {
26     title: '服务端渲染',
27     files: files
28   })
29
30   // 3. 发送解析替换过后的响应数据
31   res.end(htmlStr)
32 })
33 })
34 })
35 server.listen(3000, function () {
36   console.log('running...')
37 })

```

服务端渲染和客户端渲染的区别：

- 客户端渲染不利于SEO搜索引擎优化
- 服务端渲染是可以被爬虫抓取到的，客户端异步渲染是很难被爬虫抓取到的
- 例如京东商品列表是服务端渲染，为了SEO优化，但是商品评论列表为了用户体验，是客户端渲染

## 留言板项目

### 浏览器解析过程(自动请求)

浏览器收到 HTML 响应内容之后，就要开始从上到下依次解析，

当在解析的过程中，如果发现：

link  
script  
img  
iframe  
video  
audio

**等带有 src 或者 href (link) 属性标签（具有外链的资源）的时候，浏览器会自动对这些资源发起新的请求，如果我们不处理这些请求，则所需的资源无法加载到网页中**

## 资源路径问题

注意：在服务端中，文件中的路径就不要去写相对路径了。

因为这个时候所有的资源都是通过url标识来获取的

我的服务器开放了/public/目录

所以这里的请求路径都写成：/public/xxx

/ 在这里就是url根路径的意思。

浏览器在真正发请求的时候会最终把<http://127.0.0.1:3000>拼上

**不要再想文件路径了，把所有的路径都想象成url地址，本案例将所有静态资源放在public目录下，这样做的好处是，如果请求路径如果请求路径是以 /public/ 开头的，则我认为你要获取 public 中的某个资源，所以我们就直接可以把请求路径当作文件路径来直接进行读取**

```
1 server.on('request', function (req, res) {
2   var url = req.url
3   if (url === '/') {
4     fs.readFile('./views/index.html', function (err, data) {
5       if (err) {
6         return res.end('404 Not Found')
7       }
8       res.end(data)
9     })
10  } else if (url.indexOf('/public/') === 0) {
11    fs.readFile('.' + url, function (err, data) { //通过.将url和文件路径联系起来
12      if (err) {
13        return res.end('404 Not Found')
14      }
15      res.end(data)
16    })
17  }
18 })
```

比如这个案例link了bootstrap，那么就需要读取public文件夹下的bootstrap.css文件，否则浏览器只发送请求，但找不到路径

明白这一点后，就可以在server.on()里通过if-else语句，根据设定好的href进行页面之间的跳转

## 表单处理

---

```

1 <form action="/pinglun" method="get">
2   <div class="form-group">
3     <label for="input_name">你的大名</label>
4     <input type="text" class="form-control" required minlength="2"
maxlength="10" id="input_name" name="name"
5       placeholder="请写入你的姓名">
6   </div>
7   <div class="form-group">
8     <label for="textarea_message">留言内容</label>
9     <textarea class="form-control" name="message" id="textarea_message"
cols="30" rows="10" required minlength="5"
10      maxlength="20"></textarea>
11   </div>
12   <button type="submit" class="btn btn-default">发表</button>
13 </form>

```

对于表单提交的请求路径，由于其中具有用户动态填写的内容，不可能通过去判断完整 url 路径来处理这个请求

对我们来讲，只需要判定，如果你的请求路径是 /pinglun 的时候，就认为你提交表单的请求过来了

需要引入url模块，获取请求路径和name以及message的值，例如

```

1 var url = require('url')
2 var obj = url.parse('/pinglun?name=的撒的撒&message=的撒的撒的撒', true)
3 console.log(obj)
4 console.log(obj.query)

```

## 重定向

如何通过服务器让客户端重定向？

1. 状态码设置为 302 临时重定向

statusCode

2. 在响应头中通过 Location 告诉客户端往哪儿重定向

setHeader

如果客户端发现收到服务器的响应的状态码是 302 就会自动去响应头中找 Location，然后对该地址发起新的请求所以你能看到客户端自动跳转了

```

1 var comment = parseObj.query //通过query获取内容
2 comment.dateTime = '2021-4-14 09:28'
3 comments.push(comment) //将内容添加到模板引擎的数组中
4 res.statusCode = 302 // 设置相应状态码
5 res.setHeader('Location', '/') //设置Location进行重定向
6 res.end() //无需响应内容

```

## npm

- node package manage(node包管理器)

- 通过npm命令安装jQuery包（`npm install --save jquery`），在安装时加上--save会主动生成说明书文件信息（将安装文件的信息添加到package.json里面）
- **在哪个路径执行npm，就会被安装在该路径下**

## npm网站

`npmjs.com` 网站 是用来搜索npm包的

## npm命令行工具

npm是一个命令行工具，只要安装了node就已经安装了npm。

npm也有版本概念，可以通过 `npm --version` 来查看npm的版本

升级npm(自己升级自己):

```
1 | npm install --global npm
```

## 常用命令

- `npm init`(生成package.json说明书文件)
  - `npm init -y`(可以跳过向导，快速生成)
- `npm install`
  - 一次性把dependencies选项中的依赖项全部安装
  - 简写 (`npm i`)
- `npm install 包名`
  - 只下载
  - 简写 (`npm i 包名`)
- `npm install --save 包名`
  - 下载并且保存依赖项（package.json文件中的dependencies选项）
  - 简写 (`npm i 包名`)
- `npm uninstall 包名`
  - 只删除，如果有依赖项会依然保存
  - 简写 (`npm un 包名`)
- `npm uninstall --save 包名`
  - 删除的同时也会把依赖信息全部删除
  - 简写 (`npm un 包名`)
- `npm help`
  - 查看使用帮助
- `npm 命令 --help`
  - 查看具体命令的使用帮助（`npm uninstall --help`）

## 解决npm被墙问题

npm存储包文件的服务器在国外，有时候会被墙，速度很慢，所以需要解决这个问题。

<https://developer.aliyun.com/mirror/NPM?from=tnpm> 淘宝的开发团队把npm在国内做了一个镜像（也就是一个备份）。

安装淘宝的cnpm:

```
1 | npm install -g cnpm --registry=https://registry.npm.taobao.org;
```

```
1 | #在任意目录执行都可以
2 | #--global表示安装到全局，而非当前目录
3 | #--global不能省略，否则不管用
4 | npm install --global cnpm
```

安装包的时候把以前的 `npm` 替换成 `cnpm`。

```
1 | #走国外的npm服务器下载jQuery包，速度比较慢
2 | npm install jQuery;
3 |
4 | #使用cnpm就会通过淘宝的服务器来下载jQuery
5 | cnpm install jQuery;
```

如果不想安装 `cnpm` 又想使用淘宝的服务器来下载：

```
1 | npm install jquery --registry=https://registry.npm.taobao.org;
```

但是每次手动加参数就很麻烦，所以我们可以把这个选项加入到配置文件中：

```
1 | npm config set registry https://registry.npm.taobao.org;
2 |
3 | #查看npm配置信息
4 | npm config list;
```

只要经过上面的配置命令，则以后所有的 `npm install` 都会通过淘宝的服务器来下载

## package.json

每一个项目都要有一个 `package.json` 文件（包描述文件，就像产品的说明书一样）

这个文件可以通过 `npm init` 自动初始化出来

```
1 |
2 | D:\code\node中的模块系统>npm init
3 | This utility will walk you through creating a package.json file.
4 | It only covers the most common items, and tries to guess sensible defaults.
5 |
6 | See `npm help json` for definitive documentation on these fields
7 | and exactly what they do.
8 |
9 | Use `npm install <pkg>` afterwards to install a package and
10 | save it as a dependency in the package.json file.
11 |
12 | Press ^C at any time to quit.
13 | package name: (node中的模块系统)
14 | Sorry, name can only contain URL-friendly characters.
15 | package name: (node中的模块系统) cls
16 | version: (1.0.0)
17 | description: 这是一个测试项目
```



```

18 entry point: (main.js)
19 test command:
20 git repository:
21 keywords:
22 author: xiaochen
23 license: (ISC)
24 About to write to D:\code\node中的模块系统\package.json:
25
26 {
27   "name": "cls",
28   "version": "1.0.0",
29   "description": "这是一个测试项目",
30   "main": "main.js",
31   "scripts": {
32     "test": "echo \"Error: no test specified\" && exit 1"
33   },
34   "author": "xiaochen",
35   "license": "ISC"
36 }
37
38
39 Is this OK? (yes) yes

```

对于目前来讲，最有用的是 `dependencies` 选项，可以用来帮助我们保存第三方包的依赖信息。

如果 `node_modules` 删除了也不用担心，只需要在控制面板中 `npm install` 就会自动把 `package.json` 中的 `dependencies` 中所有的依赖项全部都下载回来。

- 建议每个项目的根目录下都有一个 `package.json` 文件
- **建议执行 `npm install` 包名的时候都加上 `--save` 选项，目的是用来保存依赖信息**

## package.json和package-lock.json

npm 5以前是不会有 `package-lock.json` 这个文件

npm5以后才加入这个文件

当你安装包的时候，npm都会生成或者更新 `package-lock.json` 这个文件

- npm5以后的版本安装都不要加 `--save` 参数，它会自动保存依赖信息
- 当你安装包的时候，会自动创建或者更新 `package-lock.json` 文件
- `package-lock.json` 这个文件会包含 `node_modules` 中所有包的信息（版本，下载地址。。。）
  - 这样的话重新 `npm install` 的时候速度就可以提升
- 从文件来看，有一个 `lock` 称之为锁
  - 这个 `lock` 使用来锁版本的
  - 如果项目依赖了 `1.1.1` 版本
  - 如果你重新install其实会下载最细版本，而不是 `1.1.1`
  - `package-lock.json` 的另外一个作用就是锁定版本号，防止自动升级

## path路径操作模块

参考文档：<https://nodejs.org/docs/latest-v13.x/api/path.html>

- `path.basename`：获取路径的文件名，默认包含扩展名
- `path.dirname`：获取路径中的目录部分

- path.extname: 获取一个路径中的扩展名部分
- path.parse: 把路径转换为对象
  - root: 根路径
  - dir: 目录
  - base: 包含后缀名的文件名
  - ext: 后缀名
  - name: 不包含后缀名的文件名
- path.join: 拼接路径
- path.isAbsolute: 判断一个路径是否为绝对路径

## Node中的其它成员(dirname,filename)

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

- `__dirname`, 是一个成员, 可以用来**动态**获取当前文件模块所属目录的绝对路径
- `__filename`, 可以用来**动态**获取当前文件的绝对路径 (包含文件名)
- `__dirname` 和 `filename` 是不受执行node命令所属路径影响的

在文件操作中, 使用相对路径是不可靠的, 因为node中文件操作的路径被设计为相对于执行node命令所处的路径。

所以为了解决这个问题, 只需要把相对路径变为绝对路径 (绝对路径不受任何影响) 就可以了。

就可以使用 `__dirname` 或者 `__filename` 来帮助我们解决这个问题

在拼接路径的过程中, 为了避免手动拼接带来的一些低级错误, **推荐使用 `path.join()` 来辅助拼接**

```
1 var fs = require('fs');
2 var path = require('path');
3
4 // console.log(__dirname + 'a.txt');
5 // path.join方法会将文件操作中的相对路径都统一的转为动态的绝对路径
6 fs.readFile(path.join(__dirname + '/a.txt'), 'utf8', function(err, data){
7     if(err){
8         throw err
9     }
10    console.log(data);
11 });
```

补充: 模块中的路径标识和这里的路径没关系, 不受影响 (就是相对于文件模块)

注意:

模块中的路径标识和文件操作中的相对路径标识不一致

模块中的路径标识就是相对于当前文件模块, 不受node命令所处路径影响

## Express (快速的)

作者: Tj

原生的http在某些方面表现不足以应对我们的开发需求, 所以就需要使用框架来加快我们的开发效率, 框架的目的就是提高效率, 让我们的代码高度统一。

在node中有很多web开发框架。主要学习express

- <http://expressjs.com/>,其中主要封装的是http。

- ```
1 // 1 安装
2 // 2 引包
3 var express = require('express');
4 // 3 创建服务器应用程序
5 //      也就是原来的http.createServer();
6 var app = express();
7
8 // 公开指定目录
9 // 只要通过这样做了, 就可以通过/public/xx的方式来访问public目录中的所有资源
10 // 在Express中开放资源就是一个API的事
11 app.use('/public/', express.static('/public/'));
12
13 //模板引擎在Express中开放模板也是一个API的事
14
15 // 当服务器收到get请求 / 的时候, 执行回调处理函数
16 app.get('/', function(req, res){
17     res.send('hello express');
18 })
19
20 // 相当于server.listen
21 app.listen(3000, function(){
22     console.log('app is runing at port 3000');
23 })
```

## 学习Express

### 起步

```
1 | cnpm install --save express
```

hello world:

```
1 // 引入express
2 var express = require('express');
3
4 // 1. 创建app
5 var app = express();
6
7 // 2.
8 app.get('/', function(req, res){
9     // 1
10    // res.write('Hello');
11    // res.write('world');
12    // res.end()
13
14    // 2
15    // res.end('hello world');
16
17    // 3
18    res.send('hello world');
19 })
20
```

```
21 app.listen(3000,function(){
22     console.log('express app is runing...');
23 })
```

## 基本路由

路由：

- 请求方法
- 请求路径
- 请求处理函数

get:

```
1 //当你以get方法请求/的时候，执行对应的处理函数
2 app.get('/',function(req,res){
3     res.send('hello world');
4 })
```

post:

```
1 //当你以post方法请求/的时候，执行对应的处理函数
2 app.post('/',function(req,res){
3     res.send('hello world');
4 })
```

## Express静态服务API

```
1 // app.use不仅仅是用来处理静态资源的，还可以做很多工作(body-parser的配置)
2 app.use(express.static('public'));
3
4 app.use(express.static('files'));
5
6 app.use('/static',express.static('public'));
```

```
1 // 引入express
2 var express = require('express');
3
4 // 创建app
5 var app = express();
6
7 // 开放静态资源
8 // 1.当以/public/开头的时候，去./public/目录中找对应资源
9 // 访问：http://127.0.0.1:3000/public/login.html
10 app.use('/public',express.static('./public'));
11
12 // 2.当省略第一个参数的时候，可以通过省略/public的方式来访问
13 // 访问：http://127.0.0.1:3000/login.html
14 // app.use(express.static('./public'));
15
16 // 3.访问：http://127.0.0.1:3000/a/login.html
17 // a相当于public的别名
18 // app.use('/a',express.static('./public'));
19
20 //
21 app.get('/',function(req,res){
```

```

22     res.end('hello world');
23   });
24
25   app.listen(3000, function() {
26     console.log('express app is running...');
27   });

```

## 在Express中配置使用 art-template 模板引擎

- [art-template官方文档](#)
- 在node中，有很多第三方模板引擎都可以使用，不是只有 art-template
  - 还有ejs, jade (pug) , handlebars, nunjucks

安装：

```

1  npm install --save art-template
2  npm install --save express-art-template
3
4  //两个一起安装
5  npm i --save art-template express-art-template

```

配置：

```

1  app.engine('html', require('express-art-template')); //第一个参数为指定后缀名，可以修改

```

## express的render()方法会自动在views文件夹下找相应的文件

使用：

```

1  app.get('/', function(req, res) {
2    // express默认会去views目录找index.html
3    res.render('index.html', {
4      title: 'hello world'
5    });
6  })

```

如果希望修改默认的 views 视图渲染存储目录，可以：

```

1  // 第一个参数views千万不要写错
2  app.set('views', 目录路径);

```

## 在Express中获取表单请求数据

获取get请求数据：

Express内置了一个api，可以直接通过 req.query 来获取数据

```

1  // 通过req.query方法获取用户输入的数据
2  // req.query只能拿到get请求的数据
3  var comment = req.query;

```

## 获取post请求数据:

在Express中没有内置获取表单post请求体的api, 这里我们需要使用一个第三方包 `body-parser` 来获取数据。

安装:

```
1 npm install --save body-parser
```

配置:

// 配置解析表单 POST 请求体插件 (注意: 一定要在 app.use(router) 之前)

```
1 var express = require('express')
2 // 引包
3 var bodyParser = require('body-parser')
4
5 var app = express()
6
7 // 配置body-parser
8 // 只要加入这个配置, 则在req请求对象上会多出来一个属性: body
9 // 也就是说可以直接通过req.body来获取表单post请求数据
10 // parse application/x-www-form-urlencoded
11 app.use(bodyParser.urlencoded({ extended: false }))
12
13 // parse application/json
14 app.use(bodyParser.json())
```

使用:

```
1 app.use(function (req, res) {
2   res.setHeader('Content-Type', 'text/plain')
3   res.write('you posted:\n')
4   // 可以通过req.body来获取表单请求数据
5   res.end(JSON.stringify(req.body, null, 2))
6 })
```

案例:

```
1 var express = require('express')
2 var bodyParser = require('body-parser')
3
4 var app = express()
5
6 app.use(bodyParser.urlencoded({
7   extended: false
8 }))
9 app.use(bodyParser.json())
10
11 app.post('/post', function (req, res) {
12   var comment = req.body
13   comment.dateTime = '2021-04-17'
14   comments.push(comment)
15   res.redirect('/')
16 })
17
```

```
18 app.listen(3000, function () {
19   console.log('server is running')
20 })
```

## 在Express中配置使用 express-session 插件操作

参考文档: <https://github.com/expressjs/session>

安装:

```
1 npm install express-session
```

配置:

```
1 //该插件会为req请求对象添加一个成员:req.session默认是一个对象
2 //这是最简单的配置方式
3 //Session是基于Cookie实现的
4 app.use(session({
5   //配置加密字符串, 他会在原有的基础上和字符串拼接起来去加密
6   //目的是为了增加安全性, 防止客户端恶意伪造
7   secret: 'keyboard cat',
8   resave: false,
9   saveUninitialized: true, //无论是否适用Session, 都默认直接分配一把钥匙
10  cookie: { secure: true }
11 })
```

使用:

```
1 // 读
2 //添加Session数据
3 //session就是一个对象
4 req.session.foo = 'bar';
5
6 //写
7 //获取session数据
8 req.session.foo
9
10 //删
11 req.session.foo = null;
12 delete req.session.foo
```

提示:

默认Session数据时内存存储数据, 服务器一旦重启, 真正的生产环境会把Session进行持久化存储。

## 利用Express实现ADUS项目

### 模块化思想

模块如何划分:

- 模块职责要单一

javascript模块化:

- Node 中的 CommonJS
- 浏览器中的：
  - AMD require.js
  - CMD sea.js
- es6中增加了官方支持

## 起步

- 初始化
- 模板处理

## 路由设计

| 请求方法 | 请求路径             | get参数 | post参数                     | 备注       |
|------|------------------|-------|----------------------------|----------|
| GET  | /students        |       |                            | 渲染首页     |
| GET  | /students/new    |       |                            | 渲染添加学生页面 |
| POST | /students/new    |       | name,age,gender,hobbies    | 处理添加学生请求 |
| GET  | /students/edit   | id    |                            | 渲染编辑页面   |
| POST | /students/edit   |       | id,name,age,gender,hobbies | 处理编辑请求   |
| GET  | /students/delete | id    |                            | 处理删除请求   |

## 提取路由模块

router.js:

```

1  /**
2   * router.js路由模块
3   * 职责：
4   *     处理路由
5   *     根据不同的请求方法+请求路径设置具体的请求函数
6   * 模块职责要单一，我们划分模块的目的就是增强代码的可维护性，提升开发效率
7   */
8  var fs = require('fs');
9
10 // Express专门提供了一种更好的方式
11 // 专门用来提供路由的
12 var express = require('express');
13 // 1 创建一个路由容器
14 var router = express.Router();
15 // 2 把路由都挂载到路由容器中
16
17 router.get('/students', function(req, res) {
18   // res.send('hello world');
19   // readFile的第二个参数是可选的，传入utf8就是告诉他把读取到的文件直接按照utf8编码，
  直接转成我们认识的字符
20   // 除了这样来转换，也可以通过data.toString()来转换
21   fs.readFile('./db.json', 'utf8', function(err, data) {

```



```

22         if (err) {
23             return res.status(500).send('Server error.')
24         }
25         // 读取到的文件数据是string类型的数据
26         // console.log(data);
27         // 从文件中读取到的数据一定是字符串，所以一定要手动转换成对象
28         var students = JSON.parse(data).students;
29         res.render('index.html', {
30             // 读取文件数据
31             students: students
32         })
33     })
34 });
35
36 router.get('/students/new', function(req, res){
37     res.render('new.html')
38 });
39
40 router.get('/students/edit', function(req, res){
41
42 });
43
44 router.post('/students/edit', function(req, res){
45
46 });
47
48 router.get('/students/delete', function(req, res){
49
50 });
51
52 // 3 把router导出
53 module.exports = router;
54

```

app.js:

```

1 var router = require('./router');
2
3 // router(app);
4 // 把路由容器挂载到app服务中
5 // 挂载路由
6 app.use(router);

```

## 回调函数获取函数异步操作结果

**如果需要获取一个函数中异步操作的结果，则必须通过回调函数来获取**

```

1 exports.find = function (callback) {
2     fs.readFile('./db.json', 'utf-8', function (err, data) {
3         if (err) {
4             return callback(err)
5         }
6         callback(null, JSON.parse(data).students)
7     })
8 }

```

上述代码readFile中的function中的data数据，通过return是无法获得的，因为return是返回到readFile这一层，并不会返回到find这一层，而readFile过程又是异步的，因此需要引入callback回调函数，等异步过程结束后（读取完毕后）进行回调，返回data值。这个回调函数相当于进入了函数内部，在某些特定条件下触发，被调用

```
1 router.get('/students', function (req, res) {
2   student.find(function (err, students) {
3     if (err) {
4       return res.status(500).send('Server error')
5     }
6     res.render('index.html', {
7       students: students
8     })
9   })
}
```

在实际调用时，对Student.find传入一个function（也就是前面的callback），当fs.readFile事件异步进行完毕后，该回调函数被调用，获得data数据

## 设计操作数据的API文件模块

es6中的find和findIndex：

find接受一个方法作为参数，方法内部返回一个条件

find会便利所有的元素，执行你给定的带有条件返回值的函数

符合该条件的元素会作为find方法的返回值

如果遍历结束还没有符合该条件的元素，则返回undefined image-20200313103810731

```
1  /**
2   * student.js
3   * 数据操作文件模块
4   * 职责：操作文件中的数据，只处理数据，不关心业务
5   */
6  var fs = require('fs');
7  /**
8   * 获取所有学生列表
9   * return []
10  */
11 exports.find = function(){
12
13 }
14
15
16 /**
17  * 获取添加保存学生
18  */
19 exports.save = function(){
20
21 }
22
23 /**
24  * 更新学生
25  */
26 exports.update = function(){
27
```

```

28 | }
29 |
30 | /**
31 |  * 删除学生
32 |  */
33 | exports.delete = function(){
34 |
35 | }

```

## 步骤

- 处理模板
- 配置静态开放资源
- 配置模板引擎
- 简单的路由，/students渲染静态页出来
- 路由设计
- 提取路由模块
- 由于接下来的一系列业务操作都需要处理文件数据，所以我们需要封装Student.js'
- 先写好student.js文件结构
  - 查询所有学生列别哦的API
  - findById
  - save
  - updateById
  - deleteById
- 实现具体功能
  - 通过路由收到请求
  - 接受请求中的参数 (get, post)
    - req.query
    - req.body
  - 调用数据操作API处理数据
  - 根据操作结果给客户端发送请求
- 业务功能顺序
  - 列表
  - 添加
  - 编辑
  - 删除

## 子模板和模板的继承（模板引擎高级语法）【include, extend, block】

注意:

模板页:

```

1 | <!DOCTYPE html>
2 | <html lang="zh">
3 | <head>
4 |   <meta charset="UTF-8">
5 |   <meta name="viewport" content="width=device-width, initial-scale=1.0">
6 |   <meta http-equiv="X-UA-Compatible" content="ie=edge">
7 |   <title>模板页</title>

```

```

8      <link rel="stylesheet"
href="/node_modules/bootstrap/dist/css/bootstrap.css"/>
9      {{ block 'head' }}{{ /block }}
10     </head>
11     <body>
12         <!-- 通过include导入公共部分 -->
13         {{include './header.html'}}
14
15         <!-- 留一个位置 让别的内容去填充 -->
16         {{ block 'content' }}
17             <h1>默认内容</h1>
18         {{ /block }}
19
20         <!-- 通过include导入公共部分 -->
21         {{include './footer.html'}}
22
23         <!-- 公共样式 -->
24         <script src="/node_modules/jquery/dist/jquery.js" ></script>
25         <script src="/node_modules/bootstrap/dist/js/bootstrap.js" ></script>
26         {{ block 'script' }}{{ /block }}
27     </body>
28 </html>

```

模板的继承：

header页面：

```

1 <div id="">
2     <h1>公共的头部</h1>
3 </div>

```

footer页面：

```

1 <div id="">
2     <h1>公共的底部</h1>
3 </div>

```

模板页的使用：

```

1 <!-- 继承(extend:延伸, 扩展)模板也layout.html -->
2 <!-- 把layout.html页面的内容都拿进来作为index.html页面的内容 -->
3 {{extend './layout.html'}}
4
5 <!-- 向模板页面填充新的数据 -->
6 <!-- 填充后就会替换掉layout页面content中的数据 -->
7 <!-- style样式方面的内容 -->
8 {{ block 'head' }}
9     <style type="text/css">
10         body{
11             background-color: skyblue;
12         }
13     </style>
14 {{ /block }}
15 {{ block 'content' }}
16     <div id="">
17         <h1>Index页面的内容</h1>

```

```
18     </div>
19   {{ /block }}
20   <!-- js部分的内容 -->
21   {{ block 'script' }}
22     <script type="text/javascript">
23
24     </script>
25   {{ /block }}
```

# MongoDB

## 关系型和非关系型数据库

**关系型数据库（表就是关系，或者说表与表之间存在关系）。**

- 所有的关系型数据库都需要通过 `sql` 语言来操作
- 所有的关系型数据库在操作之前都需要设计表结构
- 而且数据表还支持约束
  - 唯一的
  - 主键
  - 默认值
  - 非空

## 非关系型数据库

- 非关系型数据库非常的灵活
- 有的关系型数据库就是key-value对儿
- 但MongoDB是长得最像关系型数据库的非关系型数据库
  - 数据库 -》 数据库
  - 数据表 -》 集合（数组）
  - 表记录 -》 文档对象

一个数据库中可以有多个数据库，一个数据库中可以有多个集合（数组），一个集合中可以有多个文档（表记录）

```
1  {
2    qq:{
3      user:[
4        {}, {}, {} ...
5      ]
6    }
7  }
```

- 也就是说你可以任意的往里面存数据，没有结构性这么一说

## 安装

- 下载

- 下载地址: <https://www.mongodb.com/download-center/community>
- 安装

```
1 npm i mongoose
```

- 配置环境变量
- 最后输入 `mongod --version` 测试是否安装成功

## 启动和关闭数据库

启动:

```
1 # mongodb 默认使用执行mongod 命令所处根目录下的/data/db作为自己的数据存储目录
2 # 所以在第一次执行该命令之前先自己手动新建一个 /data/db
3 mongod
```

如果想要修改默认的数据存储目录, 可以:

```
1 mongod --dbpath = 数据存储目录路径
```

停止:

```
1 在开启服务的控制台, 直接Ctrl+C;
2 或者直接关闭开启服务的控制台。
```

## 连接数据库

连接:

```
1 # 该命令默认连接本机的 MongoDB 服务
2 mongo
```

退出:

```
1 # 在连接状态输入 exit 退出连接
2 exit
```

## 基本命令

- `show dbs`
  - 查看数据库列表(数据库中的所有数据库)
- `db`
  - 查看当前连接的数据库
- `use 数据库名称`
  - 切换到指定的数据库, (如果没有会新建)
- `show collections`

- 查看当前目录下的所有数据表
- `db.表名.find()`
  - 查看表中的详细信息

## 在Node中如何操作MongoDB数据库

### 使用官方的MongoDB包来操作

<http://mongodb.github.io/node-mongodb-native/>

### 使用第三方包mongoose来操作MongoDB数据库

第三方包：`mongoose` 基于MongoDB官方的 `mongodb` 包再一次做了封装，名字叫 `mongoose`，是WordPress项目团队开发的。

<https://mongoosejs.com/>

```
1 const mongoose = require('mongoose');
2 mongoose.connect('mongodb://localhost:27017/test', {useNewUrlParser: true,
  useUnifiedTopology: true});
3
4 const Cat = mongoose.model('Cat', { name: String });
5
6 const kitty = new Cat({ name: 'Zildjian' });
7 kitty.save().then(() => console.log('meow'));
```

## 学习指南（步骤）

官方学习文档：<https://mongoosejs.com/docs/index.html>

### 设计Scheme 发布Model (创建表)

```
1 // 1. 引包
2 // 注意：按照后才能require使用
3 var mongoose = require('mongoose');
4
5 // 拿到schema图表
6 var Schema = mongoose.Schema;
7
8 // 2. 连接数据库
9 // 指定连接数据库后不需要存在，当你插入第一条数据库后会自动创建数据库
10 mongoose.connect('mongodb://localhost/test');
11
12 // 3. 设计集合结构（表结构）
13 // 用户表
14 var userSchema = new Schema({
15   username: { //姓名
16     type: String,
17     require: true //添加约束，保证数据的完整性，让数据按规矩统一
18   },
19   password: {
20     type: String,
21     require: true
```

```

22     },
23     email: {
24         type: String
25     }
26 });
27
28 // 4.将文档结构发布为模型
29 // mongoose.model方法就是用来将一个架构发布为 model
30 //     第一个参数：传入一个大写名词单数字符串用来表示你的数据库的名称
31 //     mongoose 会自动将大写名词的字符串生成 小写复数 的集合名称
32 //     例如 这里会变成users集合名称
33 //     第二个参数：架构
34 //     返回值：模型构造函数
35 var User = mongoose.model('User', userSchema);

```

## 添加数据（增）

```

1 // 5.通过模型构造函数对User中的数据进行操作
2 var user = new User({
3     username: 'admin',
4     password: '123456',
5     email: 'xiaochen@qq.com'
6 });
7
8 user.save(function(err, ret) {
9     if (err) {
10         console.log('保存失败');
11     } else {
12         console.log('保存成功');
13         console.log(ret);
14     }
15 });

```

## 删除（删）

根据条件删除所有：

```

1 User.remove({
2     username: 'xiaoxiao'
3 }, function(err, ret) {
4     if (err) {
5         console.log('删除失败');
6     } else {
7         console.log('删除成功');
8         console.log(ret);
9     }
10 });

```

根据条件删除一个：

```

1 Model.findOneAndRemove(conditions,[options],[callback]);

```

根据id删除一个：



```
1 | user.findByIdAndRemove(id,[options],[callback]);
```

## 更新 (改)

更新所有:

```
1 | user.remove(conditions,doc,[options],[callback]);
```

根据指定条件更新一个:

```
1 | user.FindOneAndUpdate([conditions],[update],[options],[callback]);
```

根据id更新一个:

```
1 | // 更新 根据id来修改表数据
2 | user.findByIdAndUpdate('5e6c5264fada77438c45dfcd', {
3 |     username: 'junjun'
4 | }, function(err, ret) {
5 |     if (err) {
6 |         console.log('更新失败');
7 |     } else {
8 |         console.log('更新成功');
9 |     }
10 | });
```

## 查询 (查)

查询所有:

```
1 | // 查询所有
2 | user.find(function(err,ret){
3 |     if(err){
4 |         console.log('查询失败');
5 |     }else{
6 |         console.log(ret);
7 |     }
8 | });
```

条件查询所有:

```
1 | // 根据条件查询
2 | user.find({ username:'xiaoxiao' },function(err,ret){
3 |     if(err){
4 |         console.log('查询失败');
5 |     }else{
6 |         console.log(ret);
7 |     }
8 | });
```

条件查询单个：

```
1 // 按照条件查询单个，查询出来的数据是一个对象（{ }）
2 // 没有条件查询使用findOne方法，查询的是表中的第一条数据
3 user.findOne({
4   username: 'xiaoxiao'
5 }, function(err, ret) {
6   if (err) {
7     console.log('查询失败');
8   } else {
9     console.log(ret);
10  }
11 });
```

## 使用Node操作MySQL数据库

文档: <https://www.npmjs.com/package/mysql>

安装:

```
1 npm install --save mysql
```

```
1 // 引入mysql包
2 var mysql = require('mysql');
3
4 // 创建连接
5 var connection = mysql.createConnection({
6   host      : 'localhost', //本机
7   user      : 'me',        //账号root
8   password  : 'secret',    //密码12345
9   database  : 'my_db'      //数据库名
10 });
11
12 // 连接数据库 （打开冰箱门）
13 connection.connect();
14
15 //执行数据操作 （把大象放到冰箱）
16 connection.query('SELECT * FROM `users` ', function (error, results, fields)
17 {
18   if (error) throw error; //抛出异常阻止代码往下执行
19   // 没有异常打印输出结果
20   console.log('The solution is: ', results);
21 });
22
23 //关闭连接 （关闭冰箱门）
24 connection.end();
```

## 异步编程

### 回调函数

不成立的情况下：

```
1 function add(x,y){
2     console.log(1);
3     setTimeout(function(){
4         console.log(2);
5         var ret = x + y;
6         return ret;
7     },1000);
8     console.log(3);
9     //到这里执行就结束了，不会i等到前面的定时器，所以直接返回了默认值 undefined
10 }
11
12 console.log(add(2,2));
13 // 结果是 1 3 undefined 4
```

使用回调函数解决：

回调函数：通过一个函数，获取函数内部的操作。（根据输入得到输出结果）

```
1 var ret;
2 function add(x,y,callback){
3     // callback就是回调函数
4     // var x = 10;
5     // var y = 20;
6     // var callback = function(ret){console.log(ret);}
7     console.log(1);
8     setTimeout(function(){
9         var ret = x + y;
10        callback(ret);
11    },1000);
12    console.log(3);
13 }
14 add(10,20,function(ret){
15     console.log(ret);
16 });
```

注意：

凡是需要得到一个函数内部异步操作的结果（setTimeout,readFile,writeFile,ajax,readdir）

这种情况必须通过 回调函数（异步API都会伴随着一个回调函数）

ajax:

基于原生XMLHttpRequest封装get方法：

```
1 var oReq = new XMLHttpRequest();
2 // 当请求加载成功要调用指定的函数
3 oReq.onload = function(){
4     console.log(oReq.responseText);
5 }
6 oReq.open("GET", "请求路径",true);
7 oReq.send();
```

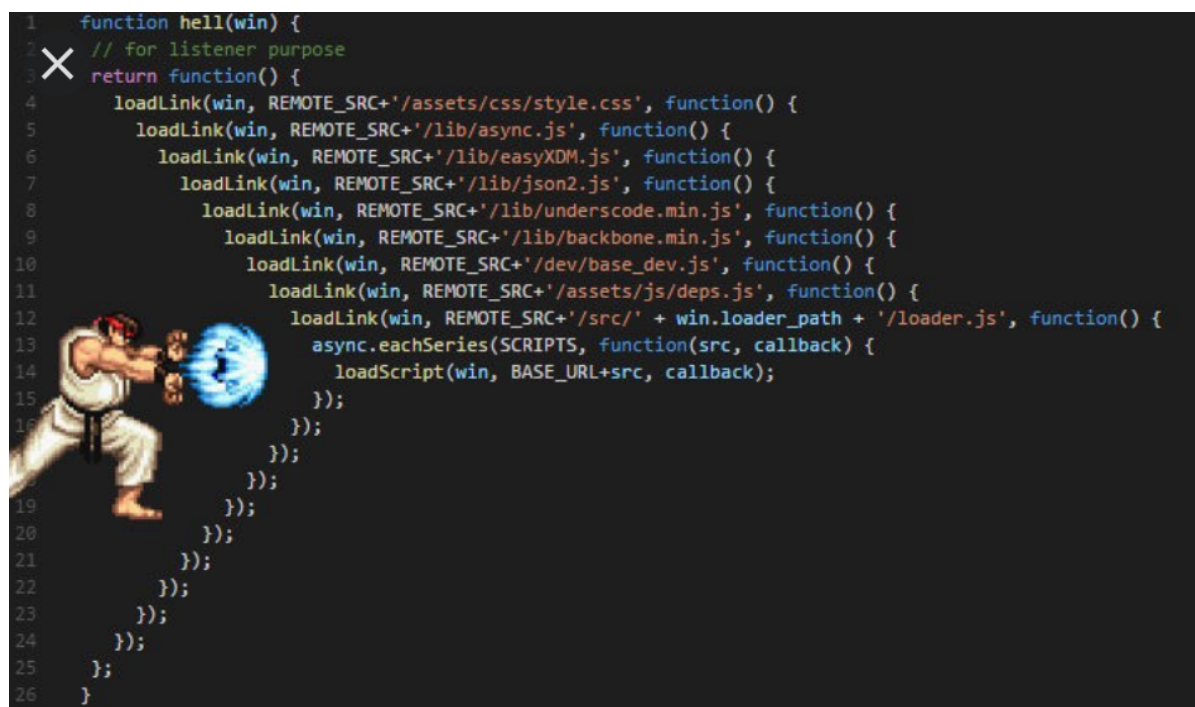
```

1 function get(url, callback){
2     var oReq = new XMLHttpRequest();
3     // 当请求加载成功要调用指定的函数
4     oReq.onload = function(){
5         //console.log(oReq.responseText);
6         callback(oReq.responseText);
7     }
8     oReq.open("GET", url, true);
9     oReq.send();
10 }
11 get('data.json', function(data){
12     console.log(data);
13 });

```

## Promise

callback hell (回调地狱) :为了保证异步的执行顺序, 回调嵌套回调



```

1 function hell(win) {
2     // for listener purpose
3     return function() {
4         loadLink(win, REMOTE_SRC+'/assets/css/style.css', function() {
5             loadLink(win, REMOTE_SRC+'/lib/async.js', function() {
6                 loadLink(win, REMOTE_SRC+'/lib/easyXDM.js', function() {
7                     loadLink(win, REMOTE_SRC+'/lib/json2.js', function() {
8                         loadLink(win, REMOTE_SRC+'/lib/underscore.min.js', function() {
9                             loadLink(win, REMOTE_SRC+'/lib/backbone.min.js', function() {
10                                loadLink(win, REMOTE_SRC+'/dev/base_dev.js', function() {
11                                    loadLink(win, REMOTE_SRC+'/assets/js/deps.js', function() {
12  loadLink(win, REMOTE_SRC+'/src/' + win.loader_path + '/loader.js', function() {
13  async.eachSeries(SRIPTS, function(src, callback) {
14  loadScript(win, BASE_URL+src, callback);
15  });
16  });
17                                    });
18                                });
19                            });
20                        });
21                    });
22                });
23            });
24        });
25    });
26 }

```

文件的读取无法判断执行顺序 (文件的执行顺序是依据文件的大小来决定的) (异步api无法保证文件的执行顺序)

```

1 var fs = require('fs');
2
3 fs.readFile('./data/a.text', 'utf8', function(err, data){
4     if(err){
5         // 1 读取失败直接打印输出读取失败
6         return console.log('读取失败');
7         // 2 抛出异常
8         // 阻止程序的执行
9         // 把错误信息打印到控制台
10        throw err;
11    }
12    console.log(data);
13 });
14
15 fs.readFile('./data/b.text', 'utf8', function(err, data){

```

```

16     if(err){
17         // 1 读取失败直接打印输出读取失败
18         return console.log('读取失败');
19         // 2 抛出异常
20         //     阻止程序的执行
21         //     把错误信息打印到控制台
22         throw err;
23     }
24     console.log(data);
25 });

```

通过回调嵌套的方式来保证顺序：

```

1  var fs = require('fs');
2
3  fs.readFile('./data/a.text', 'utf8', function(err, data){
4      if(err){
5          // 1 读取失败直接打印输出读取失败
6          return console.log('读取失败');
7          // 2 抛出异常
8          //     阻止程序的执行
9          //     把错误信息打印到控制台
10         throw err;
11     }
12     console.log(data);
13     fs.readFile('./data/b.text', 'utf8', function(err, data){
14         if(err){
15             // 1 读取失败直接打印输出读取失败
16             return console.log('读取失败');
17             // 2 抛出异常
18             //     阻止程序的执行
19             //     把错误信息打印到控制台
20             throw err;
21         }
22         console.log(data);
23         fs.readFile('./data/a.text', 'utf8', function(err, data){
24             if(err){
25                 // 1 读取失败直接打印输出读取失败
26                 return console.log('读取失败');
27                 // 2 抛出异常
28                 //     阻止程序的执行
29                 //     把错误信息打印到控制台
30                 throw err;
31             }
32             console.log(data);
33         });
34     });
35 });

```

- Promise：承诺，保证
- Promise本身不是异步的，但往往都是内部封装一个异步任务

基本语法：

```

1  // 在EcmaScript 6中新增了一个API Promise

```

```

2 // Promise 是一个构造函数
3
4 var fs = require('fs');
5 // 1 创建Promise容器      resolve:解决    reject: 失败
6 var p1 = new Promise(function(resolve, reject) {
7     fs.readFile('./a.text', 'utf8', function(err, data) {
8         if (err) {
9             // console.log(err);
10            // 把容器的Pending状态变为rejected
11            reject(err);
12        } else {
13            // console.log(data);
14            // 把容器的Pending状态变为resolve
15            resolve(1234);
16        }
17    });
18 });
19
20 // 当p1成功了，然后就（then）做指定的操作
21 // then方法接收的function就是容器中的resolve函数
22 p1
23     .then(function(data) {
24         console.log(data);
25     }, function(err) {
26         console.log('读取文件失败了', err);
27     });
28

```

如果在then()中返回一个promise对象，那下一个then接受的是所返回promise对象的resolve和reject，可以继续链式调用

```

p1
    .then(function (data) {
        console.log(data)
        return p2
    }, function (err) {
        console.log('读取文件失败了', err)
    })
    .then(function (data) {
        console.log(data)
        return p3
    })
    .then(function (data) {
        console.log(data)
        console.log('end')
    })

```

封装Promise的readFile：

```

1 var fs = require('fs');
2
3 function pReadFile(filePath) {

```

```

4     return new Promise(function(resolve, reject) {
5         fs.readFile(filePath, 'utf8', function(err, data) {
6             if (err) {
7                 reject(err);
8             } else {
9                 resolve(data);
10            }
11        });
12    });
13 }
14
15 pReadFile('./a.txt')
16   .then(function(data) {
17       console.log(data);
18       return pReadFile('./b.txt');
19   })
20   .then(function(data) {
21       console.log(data);
22       return pReadFile('./a.txt');
23   })
24   .then(function(data) {
25       console.log(data);
26   })
27

```

mongoose所有的API都支持Promise:

```

1 // 查询所有
2 User.find()
3   .then(function(data){
4       console.log(data)
5   })

```

注册:

```

1 User.findOne({username: 'admin'}, function(user){
2     if(user){
3         console.log('用户已存在')
4     } else {
5         new User({
6             username: 'aaa',
7             password: '123',
8             email: 'fffff'
9         }).save(function(){
10             console.log('注册成功');
11         })
12     }
13 })

```

```

1 User.findOne({
2     username: 'admin'
3 })
4   .then(function(user){
5       if(user){

```

```
6      // 用户已经存在不能注册
7      console.log('用户已存在');
8  }
9  else{
10     // 用户不存在可以注册
11     return new User({
12         username: 'aaa',
13         password: '123',
14         email: 'fffff'
15     }).save();
16 }
17 })
18 .then(function(ret){
19     console.log('注册成功');
20 })
```

## Generator

async函数

## 其他

### 修改完代码自动重启

我们在这里可以使用一个第三方命令行工具：`nodemon` 来帮助我们解决频繁修改代码重启服务器的问题。

`nodemon` 是一个基于Node.js开发的一个第三方命令行工具，我们使用的时候需要独立安装：

```
1  #在任意目录执行该命令都可以
2  #也就是说，所有需要 --global 安装的包都可以在任意目录执行
3  npm install --global nodemon
4  npm install -g nodemon
5
6  #如果安装不成功的话，可以使用cnpm安装
7  cnpm install -g nodemon
```

安装完毕之后使用：

```
1  node app.js
2
3  #使用nodemon
4  nodemon app.js
```

只要是通过 `nodemon` 启动的服务，则他会监视你的文件变化，当文件发生变化的时候，会自动帮你重启服务器。

## 封装异步API

回调函数：获取异步操作的结果



```
1 function fn(callback){
2   // var callback = function(data){ console.log(data); }
3   setTimeout(function(){
4     var data = 'hello';
5     callback(data);
6   },1000);
7 }
8 // 如果需要获取一个函数中异步操作的结果，则必须通过回调函数的方式来获取
9 fn(function(data){
10   console.log(data);
11 })
```

## 数组的遍历方法，都是对函数作为一种参数

---

## EcmaScript 6

---

参考文档: <https://es6.ruanyifeng.com/>

## 项目案例

---

### 目录结构

---

```
1 .
2 app.js  项目的入口文件
3 controllers
4 models  存储使用mongoose设计的数据模型
5 node_modules  第三方包
6 package.json  包描述文件
7 package-lock.json  第三方包版本锁定文件（npm5之后才有）
8 public  公共静态资源
9 routes
10 views  存储视图目录
```

## 模板页

---

- 子模板
- 模板继承

## 路由设计

---

| 路由            | 方法   | get参数 | post参数                  | 是否需要登录 | 备注     |
|---------------|------|-------|-------------------------|--------|--------|
| /             | get  |       |                         |        | 渲染首页   |
| /register(登录) | get  |       |                         |        | 渲染注册页面 |
| /register     | post |       | email,nickname,password |        | 处理注册请求 |
| /login        | get  |       |                         |        | 渲染登陆界面 |
| /login        | post |       | email,password          |        | 处理登录请求 |
| /logout       | get  |       |                         |        | 处理退出请求 |
|               |      |       |                         |        |        |

## 模型设计

## 功能实现

## 步骤

- 创建目录结构
- 整合静态也-模板页
  - include
  - block
  - extend
- 设计用户登陆，退出，注册的路由
- 用户注册
  - 先处理客户端页面的内容（表单控件的name，收集表单数据，发起请求）
  - 服务端
    - 获取从客户端收到的数据
    - 操作数据库
      - 如果有错，发送500告诉客户端服务器错了'
      - 其他的根据业务发送不同的响应数据
- 登录
- 退出

## Express中间件

## 中间件的概念

参考文档：<http://expressjs.com/en/guide/using-middleware.html>

中间件：把很复杂的事情分割成单个，然后依次有条理的执行。就是一个中间处理环节，有输入，有输出。

说的通俗易懂点儿，中间件就是一个（从请求到响应调用的方法）方法。

把数据从请求到响应分步骤来处理，每一个步骤都是一个中间处理环节。

```
1 var http = require('http');
2 var url = require('url');
3
4 var cookie = require('./expressPtoject/cookie');
5 var query = require('./expressPtoject/query');
6 var postBody = require('./expressPtoject/post-body');
7
8 var server = http.createServer(function(){
9     // 解析请求地址中的get参数
10    // var obj = url.parse(req.url,true);
11    // req.query = obj.query;
12    query(req,res); //中间件
13
14    // 解析请求地址中的post参数
15    req.body = {
16        foo: 'bar'
17    }
18 });
19
20 if(req.url === 'xxx'){
21     // 处理请求
22     ...
23 }
24
25 server.listen(3000,function(){
26     console.log('3000 runing...');
27 });
```

同一个请求对象所经过的中间件都是同一个请求对象和响应对象。

```
1 var express = require('express');
2 var app = express();
3 app.get('/abc',function(req,res,next){
4     // 同一个请求的req和res是一样的，
5     // 可以前面存储下面调用
6     console.log('/abc');
7     // req.foo = 'bar';
8     req.body = {
9         name: 'xiaoxiao',
10        age: 18
11    }
12    next();
13 });
14 app.get('/abc',function(req,res,next){
15     // console.log(req.foo);
16     console.log(req.body);
17     console.log('/abc');
18 });
19 app.listen(3000, function() {
20     console.log('app is running at port 3000.');
```

```
21 });  
22
```

 image-20200317110520098

## 中间件的分类:

### 应用程序级别的中间件

万能匹配（不关心任何请求路径和请求方法的中间件）：

```
1 app.use(function(req, res, next){  
2   console.log('Time', Date.now());  
3   next();  
4 });
```

关心请求路径和请求方法的中间件：

```
1 app.use('/a', function(req, res, next){  
2   console.log('Time', Date.now());  
3   next();  
4 });
```

### 路由级别的中间件

严格匹配请求路径和请求方法的中间件

get:

```
1 app.get('/', function(req, res){  
2   res.send('get');  
3 });
```

post:

```
1 app.post('/a', function(req, res){  
2   res.send('post');  
3 });
```

put:

```
1 app.put('/user', function(req, res){  
2   res.send('put');  
3 });
```

delete:

```
1 app.delete('/delete', function(req, res){  
2   res.send('delete');  
3 });
```

## 总

```
1 var express = require('express');
2 var app = express();
3
4 // 中间件：处理请求，本质就是个函数
5 // 在express中，对中间件有几种分类
6
7 // 1 不关心任何请求路径和请求方法的中间件
8 // 也就是说任何请求都会进入这个中间件
9 // 中间件本身是一个方法，该方法接收三个参数
10 // Request 请求对象
11 // Response 响应对象
12 // next 下一个中间件
13 // // 全局匹配中间件
14 // app.use(function(req, res, next) {
15 //   console.log('1');
16 //   // 当一个请求进入中间件后
17 //   // 如果需要请求另外一个方法则需要使用next () 方法
18 //   next();
19 //   // next是一个方法，用来调用下一个中间件
20 //   // 注意：next () 方法调用下一个方法的时候，也会匹配（不是调用紧挨着的哪一个）
21 // });
22 // app.use(function(req, res, next) {
23 //   console.log('2');
24 // });
25
26 // // 2 关心请求路径的中间件
27 // // 以/xxx开头的中间件
28 // app.use('/a',function(req, res, next) {
29 //   console.log(req.url);
30 // });
31
32 // 3 严格匹配请求方法和请求路径的中间件
33 app.get('/',function(){
34   console.log('/');
35 });
36 app.post('/a',function(){
37   console.log('/a');
38 });
39
40 app.listen(3000, function() {
41   console.log('app is running at port 3000.');
```

## 错误处理中间件

```
1 app.use(function(err, req, res, next){
2   console.error(err, stack);
3   res.status(500).send('Something broke');
4 });
```

配置使用404中间件：

```
1 app.use(function(req, res){
2   res.render('404.html');
3 });
```

配置全局错误处理中间件:

```
1 app.get('/a', function(req, res, next) {
2   fs.readFile('.a/bc', function() {
3     if (err) {
4       // 当调用next()传参后,则直接进入全局错误处理中间件方法中
5       // 当发生全局错误的时候,我们可以调用next传递错误对象
6       // 然后被全局错误处理中间件匹配到并进行处理
7       next(err);
8     }
9   })
10 });
11 //全局错误处理中间件
12 app.use(function(err, req, res, next){
13   res.status(500).json({
14     err_code: 500,
15     message: err.message
16   });
17 });
```

## 内置中间件

- express.static(提供静态文件)
  - <http://expressjs.com/en/starter/static-files.html#serving-static-files-in-express>

## 第三方中间件

参考文档: <http://expressjs.com/en/resources/middleware.html>

- body-parser
- compression
- cookie-parser
- mogran
- response-time
- server-static
- session