Node.js

1. Node.js 介绍

1.1. 为什么要学习 Node.js

- 企业需求
  - 具有服务端开发经验更好
  - front - end
  - back - end
  - 全栈开发工程师
    - 全干
  - 基本的网站开发能力
    - 服务端
    - 前端
    - 运维部署
  - 案例：多人社区（类似于：https://cnodejs.org）

1.2. Node.js 是什么

- Node.js 官网：https:/nodejs.org
- 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
- Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine.
  - Node.js 不是一门语言
  - Node.js 不是库、不是框架
  - Node.js 是一个 JavaScript 运行时环境
  - 简单点来讲就是 Node.js 可以解析和执行 JavaScript 代码
  - 以前只有浏览器可以解析和执行 JavaScript 代码
  - 也就是说现在的 JavaScript 可以完全脱离浏览器来运行，一切都归功于：Node.js
- 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.js 这个 Javascript 执行环境中为 JavaScript 提供了一些服务器级别的操作 API

        - 例如文件的读写
        - 网络服务的构建
        - 网络通信
        - http服务器
        - 等处理……
- 构于 Chrome
    - 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 envent-driven,non-blocking I/O mode that makes it lightweight and efficent.
    - envent-driven 事件驱动
    - non-blocking I/O mode 非阻塞 I/O 模型（异步）
    - ightweight and efficent. 轻量和高效
- Node.js package ecosystem,npm,is the larget scosystem of open sourcr 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`

1.3. Node.js 能做什么

- Web 服务器后台
- 命令行工具
    - npm（node）
    - git（c 语言）
    - hexo（node）
    - …
- 对于前端开发工程师来说，接触 node 最多的是它的命令行工具
    - 自己写的很少，主要是使用别人第三方的
    - webpack
    - gulp
    - npm

1.4. 预备知识

- HTML
- CSS
- JavaScript
- 简单的命令行操作
    - cd
    - dir
    - ls
    - mkdir
    - rm
- 具有服务端开发经验更佳

1.5. 一些资源

- 《深入浅出 Node.js》
  - 作者：朴灵
  - 偏理论，几乎没有任何实战行内容
  - 理解底层原理有帮助
  - 结合课程的学习去看
- 《Node.js 权威指南》
  - API 讲解
  - 也没有业务，没有实战
- JavaScript 标准参考教程（alpha）：http://javascript.ruanyifeng.com/
- Node 入门：http://www.nodebeginner.org/index-zh-cn.html
  - mini book（42页）
  - 比较可以的
- 官方 API 文档：https://nodejs.org/dist/latest-v6.x/docs/api/
- 中文文档（版本比较旧，凑合看）：http://www.nodeclass.com/api/node.html
- CNODE 社区：http://cnodejs.org
- CNODE-新手入门：http://cnodejs.org/getstart

1.6. 这门课程你能学到啥?

- B/S 编程模型
  - Browser - Server
  - back-end
  - 任何服务端技术这种 BS 编程模型都是一样，和语言无关
  - Node 只是作为我们学习 BS 编程模型的一个工具而已
- 模块化编程
  - RequireJS
  - SeaJS
  - CSS中：`@import('文件路径')` 表示从一个文件引入另外一个文件
  - 以前认知的 JavaScript 只能通过 `script()` 一样来引用加载 JavaScript 脚本文件
- Node 常用 API
- 异步编程
  - 回调函数
  - Promise
  - async
  - generator
- Express Web 开发框架
- EcmaScript
  - 穿插讲解
  - 它只是一个新的语法而已
- ...
- 学习 Node 不仅会帮助大家打开服务器黑盒子，同时会帮助你学习以后的前端高级内容
  - Vue.js
  - React
  - angular

2. 安装 Node 环境

- 查看当前 Node 环境版本号
- 下载：https://nodejs.org/en/download/
- 安装
  - 傻瓜式的一路 `next` 就可以了
  - 对于已经装过的，重新安装就会升级
- 确认 Node 环境是否安装成功
  - 打开命令行输入 `node --version`
  - 或者 `node -v`
- 环境变量

3. Hello World

3.1. 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。

3.2. 文件读写

- 文件读取：

```
//浏览器中的JavaScript是没有文件操作能力的
//但是Node中的JavaScript具有文件操作能力
//fs是file-system的简写，就是文件系统的意思
//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
//在fs这个和兴模块中，就提供了人所有文件操作相关的API
//例如 fs.readFile就是用来读取文件的

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

// 2. 读取文件
//    第一个参数就是要读取的文件路径
//    第二个参数是一个回调函数
//        error
//            如果读取失败，error 就是错误对象
//            如果读取成功，error 就是 null
//        data
//            如果读取成功，data 就是读取到的数据
//            如果读取失败，data 就是 undefined

//            成功
//             data  数据
//             error null
//            失败
//             data  undefined没有数据
//             error 错误对象
```

```
// readFile 的第二个参数是可选的，传入 utf8 就是告诉它把读取到的文件直接按照 utf8 编码，转成我们能认识的字符
// 除了这样来转换之外，也可以通过 data.toString() 的方式
fs.readFile('./data/hello.txt', 'utf8', function(error, data) {
    // 文件中存储的其实都是二进制数据 0 1
    // 这里为什么看到的不是 0 和 1 呢？原因是二进制转为 16 进制了
    // 但是无论是二进制还是 16 进制，人类都不认识
    // 所以我们可以通过 toString 方法把其转为我们能认识的字符
    // console.log(data);

    // 在这里就可以通过判断 error 来确认是否有错误发生
    if (error) {
        console.log('文件读取失败了');
    }
    else {
        console.log(data.toString());
    }
})

// 从文件中读取到的数据一定是字符串
// 所以一定要收到那个转成对象才能使用
var students = JSON.parse(data).students
```

- 文件写入：

```
//  1.使用fs核心模块
var fs = require('fs');

// 2.将数据写入文件
//   第一个参数：文件路径
//   第二个参数：要写入的内容
//   第三个参数：回调函数
//      error
//
//      成功：
//        文件写入成功
//        error 是 null
//      失败：
//        文件写入失败
//        error 就是错误对象
fs.writeFile('./data/a.md', '大家好，我是Node.js', function(error) {
   if(err){
        console.log('文件写入失败');
   }
    else{
         console.log('写入成功了');
    }
})
```

3.3. http

```
// 你可以使用 Node 非常轻松的构建一个 Web 服务器
// 在 Node 中专门提供了一个核心模块：http
// http 这个模块的职责就是帮你创建编写服务器的

// 1.加载 http 核心模块
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();

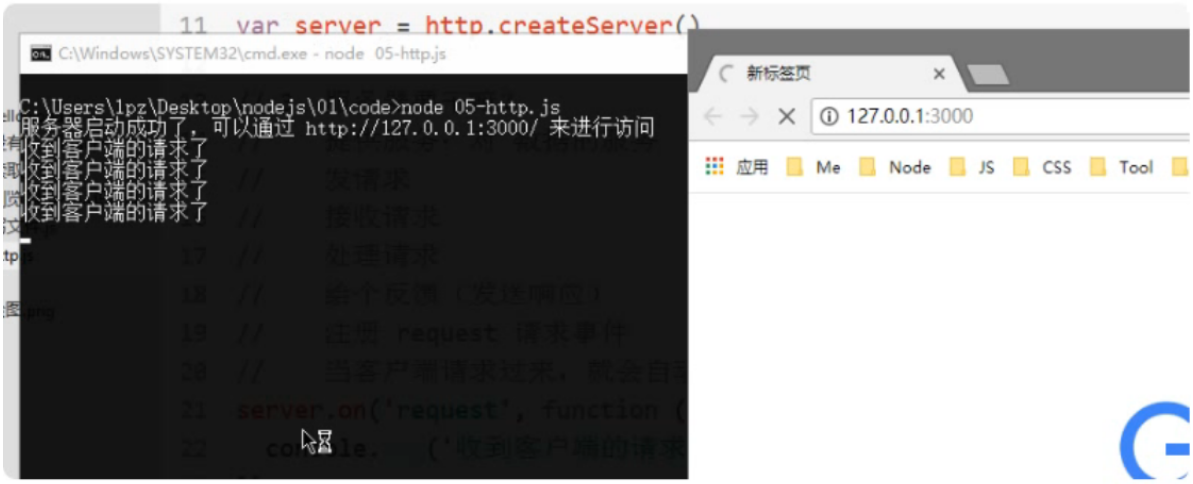
// 3. 服务器要干嘛？
//    提供服务：对数据的服务
//    发请求
//    接收请求
//    处理请求
//    给个反馈（发送响应）
//    注册 request 请求事件
//    当客户端给请求过来，就会自动触发服务器的 request 请求事件，然后执行第二个参数：回调处理函数
// request 请求事件处理函数，需要接收两个参数：
//    Request 请求对象
//        请求对象可以用来获取客户端的一些请求信息，例如请求路径
//    Response 响应对象
//        响应对象可以用来给客户端发送响应消息
server.on('request', function(request, response) {
    // 请求路径：
    // http://127.0.0.1:3000/ /
    // http://127.0.0.1:3000/a /a
    // http://127.0.0.1:3000/foo/b /foo /b
    console.log('收到客户端请求了，请求路径是：' + request.url);

    // response 对象有一个方法：write 可以用来给客户端发送响应数据
    // write 可以使用多次，但是最后一定要使用 end 来结束响应，否则客户端会一直等待
    response.write('hello');
    response.write('nodejs');

    // 告诉客户端，我的话说完了，你可以呈递给用户了
    response.end();

    // 由于现在我们的服务器的能力还非常的弱，无论什么请求，都只能响应 hello nodejs
    // 思考：
    //    我希望当请求不同的路径的时候响应不同的结果
    //    例如：
    //    / index
    //    /login 登陆
    //    /register 注册
    //    /haha 哈哈哈
})

// 4. 绑定端口号，启动服务器
server.listen(3000, function() {
    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');
})
```



4. Node 中的 Javascript

4.1. EcmaScript

- 没有 DOM、BOM
- 变量
- 方法
- 数据类型
- 内置对象
- Array
- Object
- Date
- Math
- ...

4.2. Node 中的模块系统

- 使用 Node 编写应用程序主要就是使用：
  - EcmaScript 语言
    - 和浏览器不一样，在 Node 中没有 BOM、DOM
- 在 Node 中没有全局作用域的概念
- 在 Node 中只能通过 `require` 方法来加载执行多个 JavaScript 脚本文件
- require 加载只能是执行其中的代码，文件与文件之间由于模块作用域，所以不会有污染的问题
  - 外部访问不到内部
  - 内部页访问不到外部
- 模块作用域固然带来了一些好处，可以加载执行多个文件，可以完全避免命名冲突污染的问题
- 但是某些情况下，模块与模块之间是需要进行通信的
- 在每个模块中，都提供了一个对象：`exports`
- 该对象默认是一个空对象
- 你要做的就是把需要被外部访问使用的成员手动挂载到 `exports` 接口对象中
- 然后谁来 `require` 这个模块，谁就可以得到模块内部的 `exports` 接口对象

4.2.1. 什么是模块化

如果一个平台具有：

- 文件作用域
- 通信规则
  - 加载 require
  - 导出

我们就可以说它符合模块化

4.2.2. CommonJS 规范

在 Node 中的 JavaScript 还有一个很重要的概念：模块系统。

- 模块作用域
- **使用require 方法来加载模块**
- **使用 exports 接口对象来导出模块中的成员**
  - 导出多个成员：

```
// 导出多个成员：

exports.add = add;

// 你可以认为在每个模块的最后 return 了这个 exports

// export 是一个对象
// 我们可以通过多次为这个对象添加成员实现对外导出多个内部成员

exports.str = 'hello';
```

  - 导出单个成员：
    - **如果一个模块需要直接导出某个成员，而非挂载的方式**
    - **那这个时候必须使用下面这种方式**

```
// 导出多个成员：
function add(x, y) {
    return x + y;
}

// 这种方式不行。
// exports = add

// 如果一个模块需要直接导出某个成员，而非挂载的方式
// 那这个时候必须使用下面这种方式：
module.exports = add;

// 你可以认为在每个模块的最后 return 了这个 exports

// export 是一个对象
// 我们可以通过多次为这个对象添加成员实现对外导出多个内部成员
```

4.2.2.1. 加载 `require`

语法：

```
var 自定义变量名称= require('模块');
```

require 的两个作用：

- **加载模块并执行里面的代码**
- **拿到被加载文件模块中的 exports 导出接口对象**

4.2.2.2. 导出 `exports`

- Node 中是模块作用域，默认文件中所有成员只在当前文件模块有效
- 希望可以被其它模块访问的成员，我们就需要把这些公开的成员都挂载到 `exports` 接口对象中就可以了

导出多个成员（必须在对象中）：

```
exports.a = 123;
exports.b = 'hello';
exports.c = function () {
    console.log('ccc');
}
exports.d = {
    foo: 'bar'
}

```

导出多个成员也可以这么来写：

```
module.exports = {
  foo: 'bar',
  add: function () {

  }
}
```

导出单个成员（拿到的就是：函数、字符串）：

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

以下情况会覆盖：

```
module.exports = 'hello';

// 以这个为准，后者会覆盖前者
module.exports = function (x, y) {
    return x + y;
}
```

也可以这样来到处多个成员：

```
module.exports = {
    add: function (x, y) {
        return x + y;
    },
    str: 'hello'
}
```

4.2.2.3. 原理解析

exports 是 `module.exports` 的一个引用:

```
console.log('exports === module.exports') // => true

exports.foo = 'bar';

// 等价于
module.exports.foo = 'bar';
```

```
// 在 Node 中，每个模块内部都有一个自己的 module 对象
// 该 module 对象中有一个成员叫：exports ，exports成员也是一个对象
// 也就是说如果你需要对外导出成员挂载到 module.exports 中

// 我们发现每次导出接口成员的时候都通过 module.exports.xxx = xxx 的方式很麻烦，点儿的太多了
// 所以 Node 为了简化你的操作，专门提供了一个变量 exports = module.exports

// var module = {
//     exports: {
//         foo: 'bar',
//         add: function
//     }
// };

// exports.foo = 'bar';
// module.exports.add = function (x, y) {
//    return x + y;
// };

// 当一个模块需要导出单个成员的时候
// 直接给 exports 赋值是不管用的

// 给 exports 赋值会断开和 module.exports 之间的引用
// 同理，给 module.exports 重新赋值也会断开

module.exports = 'hello';
exports.foo = 'world';

// 也就是说在模块中还有这么一句代码：
// var exports = module.exports

// 两者一致，那就说明，我可以使用任意一方来导出内部成员
// console.log(exports === module.exports);

// 谁来 require 我，谁就得到 module.exports
```

```
// 默认在代码的最后有一句：
// 一定要记住，最后 return 的是 module.exports
// 所以你给 exports 重新赋值不管用，
// return module.exports;
```

- 真正去用的时候：
  - 导出多个成员：exports.xxx = xxx
    - 导出多个成员也可以：module.exports = {

      }
  - 导出单个成员：module.exports = xxx
- 如果你是在分不清 exports 和 module.exports，你可以选择忘记 exports，而只使用 module.exports 也没问题

4.2.2.4. exports 和 module.exports 的区别

- 在 Node 中，每个模块内部都有一个自己的 module 对象
- 该 module 对象中有一个成员叫：exports ， exports成员也是一个对象
- 我们可以把需要导出的成员都挂载到 module.exports 接口对象中
- 也就是：`module.exports.xxx = xxx` 的方式
- 但是每次导出接口成员的时候都通过 module.exports.xxx = xxx 的方式很麻烦，点儿的太多了
- 所以 Node 为了简化你的操作，同时在每一个模块中都提供了一个成员叫：`exports`
- `exports === module.exports` 结果为 `true`
- 所以对于：`module.exports.xxx = xxx` 的方式完全可以用 `exports.xxx = xxx`
- 当一个模块需要导出单个成员的时候，这个时候必须使用：`module.exports = xxx` 的方式
- 不要使用 `exports = xxx`，不管用
- 因为每个模块最终向外 `return` 的是 `module.exports`
- 而 `exports` 知识 `module.exports` 的一个引用
- 所以即便你为 `exports` 重新赋值，也不会影响 `module.exports`
- 但是有一种赋值方式比较特殊：`exports = module.exports` 这个用来重新建立引用关系的

4.2.2.5. require 方法加载规则

> 深入浅出 Node.js（三）：深入 Node.js 的模块机制
>
> 如果想要了解更多底层细节，参考：《深入浅出 Node.js》中的模块系统章节

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

- **优先从缓存加载**
  - 不会重复加载
  - 重复 `require` 可以拿到被加载模块其中的接口对象，但是不会重复执行里面的代码
  - 这样做的目的是为了避免重复加载，提高模块加载效率
- **判断模块标识**
  - 路径形式的模块

        - ./ 当前目录，不可省略
        - ../ 上一级目录，不可省略
        - /xxx 几乎不用
        - d:/a/foo.js 几乎不用
        - 首位的 / 在这里表示的是当前文件模块所属磁盘根路径
        - .js 后缀名可以省略
    - 如果是非路径形式的模块标识
        - 核心模块
            - 核心模块的本质也是文件
            - 核心模块问价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
            - `require('fs')`
        - 第三方模块
            - 凡是第三方模块都必须通过 npm 下载
            - 使用的时候通过 `require('包名')` 的方式来进行加载才可以使用
            - 不可能有任何一个第三方包和核心模块的名字是一样的
            - 既不是核心模块、也不是路径形式的模块
                - 先找到当前文件所属目录中的 node_modules 目录
                - node_modules/art-template
                - node_modules/art-template/package.json 文件
                - node_modules/art-template/package.json 文件中的 main 属性
                - main 属性中就记录了 art-template 的入口模块
                - 然后加载使用这个第三方包
                - 实际上最终加载的还是文件
                - 如果 package.json 文件不存在或者 main 指定的入口模块没有
                    - 则 node 会自动找该目录下的 index.js
                    - 也就是说该目录下的 index.js 会作为一个默认备选项
                - 如果以上所有条件中的任何一个条件都不成立，则会进入上一级目录中的 node_modules 目录查找，规则同上
                - 如果上一级还没有，则继续往上上一级查找
                - 。。。
                - 如果直到当前磁盘根目录还找不到，最后报错：`Can not find module xxx`
                - 注意：我们一个项目有且只有一个 node_modules，放在项目根目录中，这样的话所有的子目录中的代码都可以加载到第三方包。
                - 一个项目中不会出现多个 node_modules
            -

```
// 加载第三方模块
var template = require('art-template')
```

- **模块查找机制总结：**

1. 优先从缓存加载
2. 核心模块
3. 路径形式的文件模块
4. 第三方模块

    - node_modules/art-template/
    - node_modules/art-template/package.json
    - node_modules/art-template/package.json 文件中的 main 属性
    - index.js 备选项
    - 进入上一级目录找 node_modules
    - 按照这个规则依次往上找，直到磁盘根目录还找不到，最后报错：Can not find mudule xxx
5. 一个项目有且仅有一个 node_modules 而且是存放在项目的根目录下

4.2.3. 核心模块

- 核心模块是由 Node 提供的一个个具名的模块，他们都有自己特殊的名称标识，例如：
    - fs 文件操作模块
    - http 网络服务构架模块
    - url 路径操作模块
    - os 操作系统信息模块
    - path 路径处理模块
- 所有核心模块在使用的时候都必须手动的先使用 `require` 方法来加载，然后才可以使用，例如：
    - `var fs = require('fs')`
- Node 为 JavaScript 提供了很多服务器级别的 API，这些 API 绝大多数都被包装到了一个具名的核心模块中了。
- 例如文件操作的 `fs` 核心模块，http 服务器构建的 `http` 模块，`path` 路径操作模块，`os` 操作系统信息模块。。。

以后只要是看到说这个模块是一个核心模块，你就马上想到如果要使用他，就必须先使用 `require` 方法加载才能使用：

```
// 用来操作文件的
var fs = require('fs');
var http = require('http');

// 用来获取机器信息的
var os = require('os');

// 用来操作路径的
var path = require('path');

// 获取当前机器的 CPU 信息
console.log(os.cpus());

// memory 内存
console.log(os.totalmem());

// 获取文件的扩展名（这里是 .txt）
path.extname('c:/a/b/hello.txt');
```

4.2.4. 用户自定义模块

- 模块职责要单一，不要乱写
- 我们划分模的目的就是为了增强项目代码的可维护性、提升开发效率
- 自己创建的文件即是自定义模块
- require
    - require 是一个方法

- require 方法有两个作用：
  1. **加载模块并执行里面的代码**
  2. **拿到被加载文件模块导出的接口对象 exports**
     - 在每个文件模块中都提供了一个对象：exports
     - export 默认是一个空对象

```
var ret = require('./b');  // 输出 `{}`
```

     - 你要做的就是把所有需要被外部访问的成员挂载到这个 `exports` 对象中
- 在 Node 中，模块有三种：
  - 具名的核心模块，例如 fs、http
  - 用户自己编写的文件模块（也就是 JS 文件）
    - 相对路径必须加 ./
    - 可以省略后缀名（.js），推荐省略
    - 相对路径中的 ./ 不能省略，否则会被当成核心模块加载，报错：Error：Cannot find module '模块名'
- 在 Node 中，没有全局作用域，只有模块作用域
  - 外部访问不到内部，内部页访问不到外部
- 既然是模块作用域，那如何让模块与模块之间进行通信
  - 有时候，我们加载文件模块的目的不是为了简简单单的执行里面的代码，更重要的是为了使用里面某个成员（例如：方法）
    - 要想实现通信只能通过 exports 对象：exports 对象用来导出（挂载），require 用来加载

4.2.5. 第三方模块

- art-template
- 必须通过 npm 来下载才可以使用

---

5. Web 服务器开发

5.1. IP 地址和端口号

计算机中只有一个物理网卡，而且都在一个局域网中，网卡的地址是唯一的。

网卡通过唯一的 IP 地址来进行定位。

- **IP 地址用来定位计算机**
- **端口号用来定位具体的应用程序**
- **一切需要联网通信的软件都会占用一个端口号**
- 端口号范围从 0 ~ 65536 之间
- 在计算机中有一些默认端口号，最好不要去使用
  - 例如 http 服务的 80
- 我们在开发过程中使用一些简单好记的就可以了，例如 3000、5000 等没什么含义
- 可以同时开启多个服务，但一定要确保不同服务占用的端口号不一致才可以

- 说白了，在一台计算机中，同一个端口号同一时间只能被一个程序占用

```
var http = require('http');
var server = http.createServer();
server.on('request', function (req, res) {
    console.log('收到请求了，请求路径是：' +req.url);  // 请求路径

    // 显示请求地址和请求端口号
    console.log('请求我的客户端地址是：'
+req.socket.remoteAddress,req.socket.remotePort);
});
server.listen(3000, function () {
    console.log('服务器启动成功，可以访问了。。。');
});
```

5.2. Content-Type（响应内容类型）

- 服务器最好把每次相应的数据是什么内容类型都告诉客户端，而且要正确的告诉
- 不同的资源对应的 Content-Type 是不一样的，具体参照： https://tool.oschina.net/commons
- 对于文本类型的数据，最好都加上编码，目的是为了防止中文解析乱码问题
- 在服务端默认发送数据的时候，其实是 utf8 编码的内容
- 但是浏览器不知道你是 utf8 编码的内容
- 浏览器在不知道服务器相应内容的编码的情况下会按照当前操作系统的默认编码区解析
  - 中文操作系统默认是 gbk 编码
- 解决方法就是正确的告诉浏览器我给你发送的内容是什么编码的
  - `res.setHeader('Content-Type', 'text/plain; charset=utf-8')`
  - text/plain 就是普通文本
  - 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的是 text/html 格式的内容
- 在 http 协议中，Content-Type 就是用来告知对方我给你发送的数据内容是什么类型、怎么编码
- 除了 Content-Type 可以用来指定编码，也可以在 HTML 页面中通过 meta 元数据来声明当前文本的编码格式，浏览器也会识别它。

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

var server = http.createServer();

server.on('request', function (req, res) {
    // 在服务端默认发送数据的时候，其实是 utf8 编码的内容
    // 但是浏览器不知道你是 utf8 编码的内容
    // 浏览器在不知道服务器相应内容的编码的情况下会按照当前操作系统的默认编码区解析
    // 中文操作系统默认是 gbk 编码
    // 解决方法就是正确的告诉浏览器我给你发送的内容是什么编码的
    // 在 http 协议中，Content-Type 就是用来告知对方我给你发送的数据内容是什么类型，怎么编码
    // res.setHeader('Content-Type', 'text/plain; charset=utf-8');
    // res.end('hello 世界');
    var url = req.url;

    if (url === '/plain') {
```

```
        // text/plain 就是普通文本
        res.setHeader('Content-Type', 'text/plain; charset=utf-8');
        res.end('hello 世界');
    } else if (url === '/html') {
        // 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的是 text/html 格式的内容
        res.setHeader('Content-Type', 'text/html; charset=utf-8');
        res.end('<p>hello html <a href="">点我</a></p>');
    }
});

server.listen(3000, function () {
    console.log('Server is running...');
});
```

5.2.1. 发送（响应）文件中的数据及 Content-Type 内容类型

- 通过网络发送文件
  - 发送的并不是文件，本质上来讲发送的是文件的内容
  - 当浏览器收到服务器响应内容之后，就会根据你的 Content-Type 进行对应的解析处理

1. 结合 `fs` 发送文件中的数据
2. Content-Type
   - 具体看： https://tool.oschina.net/commons
   - 不同的资源对应的 Content-Type 是不一样的
   - 图片不需要指定编码
   - 一般只为字符数据才指定编码

- Content-Type 内容类型查询工具网址：https://tool.oschina.net/commons
  - 


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

var server = http.createServer();

```

```
server.on('request', function (req, res) {

    var url = req.url;

    if (url === '/') {
        // 我们要发送的还是在文件中的内容
        fs.readFile('./resourse/index.html', function (err, data) {
            if (err) {
                res.setHeader('Content-Type', 'text/plain; charset=utf-8');
                res.end('文件读取失败，请稍后重试！ ');
            } else {
                // data 默认是二进制数据，可以通过 .toString() 方法转为咱们能识别的字符串
                // res.end() 支持两种数据类型：一种是二进制，一种是字符串
                res.setHeader('Content-Type', 'text/html; charset=utf-8');
                res.end(data);
            }
        });
    } else if (url === '/baby') {  // 这里url中的 '/baby' 只是一个标识，可以随便修改，上面的 '/' 同理，但是都必须以 '/' 开头
        // url：统一资源定位符
        // 一个 url 最终其实是要对应到一个资源的
        fs.readFile('./resourse/ab2.jpg', function (err, data) {
            if (err) {
                res.setHeader('Content-Type', 'text/plain; charset=utf-8');
                res.end('文件读取失败，请稍后重试！ ');
            } else {
                // data 默认是二进制数据，可以通过 .toString() 方法转为咱们能识别的字符串
                // res.end() 支持两种数据类型：一种是二进制，一种是字符串
                // 图片就不需要指定编码了，因为我们常说的编码一般指的是：字符编码
                res.setHeader('Content-Type', 'text/jpeg');
                res.end(data);
            }
        });
    }
});

server.listen(3000, function () {
    console.log('Server is running...');
});
```

5.3. 实现 Apache 功能

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

// 1. 创建 Server
var server = http.createServer();

// 2.监听 Server 的 request 请求，设置请求处理函数
    // 请求 → 处理 → 响应
    // 一个请求对应一个响应，如果在一个请求过程中，已经相应结束了，则不能重复发送响应。
    // 没有请求就没有响应
// 咱么以前使用过 Apache 服务器软件，这个软件默认有一个 www 目录，所有存放在 www 目录中的资源都可以通过网址来浏览

```

```
// 路径统一定义，路径发生变化时只需要在这里统一修改就好了
var wwwDir = 'C:/app/www';

server.on('request', function (req, res) {
    var url = req.url;
    // /  index.html
    // /a.txt  wwwDir + /a.txt
    // /apple/login.html  wwwDir + /apple/login.html
    // /img/ab1.jpg  wwwDir + /img/ab1.jpg
    if (url === '/') {
        fs.readFile(wwwDir + '/index.html', function (err, data) {
           // if (err) {
           //     res.end('404 Not Found.');
           // } else {
           //
           // }

           if (err) {
               // return 有两个作用：
               //  1. 方法返回值
               //  2. 阻止代码继续往后执行
               return res.end('404 Not Found.');
           }

            res.end(data);
        });
    } else if (url === '/a.txt') {
        fs.readFile(wwwDir + '/a.txt', function (err, data) {
           if (err) {
               res.setHeader('Content-Type', 'text/plain; charset=utf8');
               return res.end('404 Not Found.');
           }
            res.end(data);
        });
    } else if (url === '/index.html') {
        fs.readFile(wwwDir + '/index.html', function (err, data) {
           if (err) {
               return res.end('404 Not Found.');
           }
            res.end(data);
        });
    } else if (url === 'apple/login.html') {
        fs.readFile(wwwDir + '/apple/login.html', function (err, data) {
           if (err) {
               return res.end('404 Not Found.');
           }
            res.end(data);
        });
    }
});

// 3. 绑定端口号，启动服务
server.listen(3000, function () {
    console.log('running...');
})
```

- 像 Apache 一样统一处理文件资源

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

// 1. 创建 Server
var server = http.createServer();

var wwwDir = 'C:/app/www';

server.on('request', function (req, res) {
    var url = req.url;
    // /  index.html
    // /a.txt  wwwDir + /a.txt
    // /apple/login.html  wwwDir + /apple/login.html
    // /img/ab1.jpg  wwwDir + /img/ab1.jpg
    var filePath = '/index.html';
    if (url !== '/') {
        filePath = url;
    }

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

});

// 3. 绑定端口号，启动服务
server.listen(3000, function () {
    console.log('running...');
})
```

- 读取目录列表

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

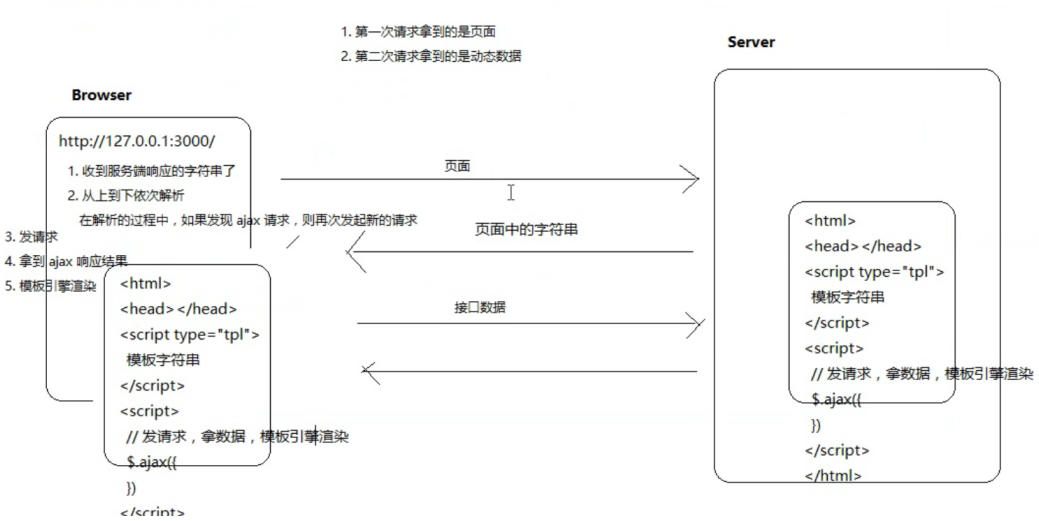
fs.readdir('D:/Movie/www', function (err, files) {
   if (err) {
       return console.log('目录不存在');
   }
    console.log(files);  // 输出一个目录列表的数组
});
```

5.4. 模板引擎

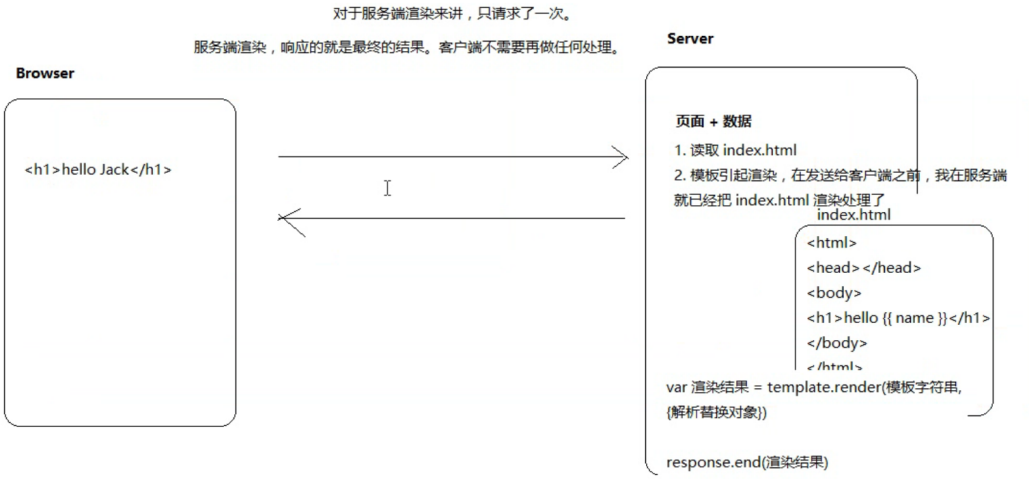
- art-template （JavaScript 模板引擎）
- art-template 不仅可以在浏览器使用，也可以在 node 中使用
- 安装：

    - `npm install art-template`
    - 该命令在哪儿执行就会把包下载到哪里。默认会下载到 node_modules 目录中
    - node_modules 不要改，也不支持改。
- 注意：在浏览器中需要引用 lib/template-web.js 文件
- **强调：模板引擎不关心你的字符串内容，只关心自己能认识的模板标记语法，例如 {{}}**
    - {{}} 语法被称之为 mustache 语法，八字胡语法。
- 在 Node 中使用 art-template 模板引擎
- 模板引擎最早诞生于服务器领域，后来才发展到了前端
- 浏览器中的使用：`template('script 标签 id', {替换对象})`
    - 需要替换的内容写到 `script` 标签当中
- **服务端渲染的概念：把模板文件读取过来，通过模板引擎解析替换，最后响应给浏览器的过程**
    - 说白了就是在服务端使用模板引擎
    - 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，客户端异步渲染时很难被爬虫抓取到的
        - 所以你会发现真正的网站既不是纯异步也不是纯服务端渲染出来的，而是两者结合来做的
        - 例如京东的商品列表就采用的时服务端渲染，目的是为了 SEO 搜索引擎优化
        - 而它的商品评论刘表为了用户体验，而且不需要 SEO 优化，所以采用的是客户端渲染

            1. **客户端渲染：**



            2. **服务端渲染：**



模板引擎在 Node 中的使用步骤：

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 API

- 基本使用：

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

// template.render('模板字符串', 替换对象)
var ret = template.render('hello {{name}}', {
    name: 'Jack'
});
console.log(ret);
```

- 模板字符串写到文件中：

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

fs.readFile('./tpl.html', function (err, data) {
    if (err) {
        return console.log('读取文件失败了');
    }
    // 默认读取到的 data 是二进制数据
    // 而模板引擎的 render 方法需要接收的是字符串
    // 所以我们在这里需要把 data 二进制数据转为 字符串 才可以给模板引擎使用
    var ret = template(data.toString(), {
        name: 'Jack',
        age: 18,
        province: '北京市',
        hobbies: [
            '写代码',
            '唱歌',
            '打游戏',
        ],
        title: '个人信息'
    });
    console.log(ret)
});
```

```
// 上面一段代码使用到的 tpl.html 文件
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>{{ title }}</title>
```

```
  </head>
  <body>
    <p>大家好，我叫：{{ name }}</p>
    <p>我今年：{{ age }} 岁了</p>
    <h1>我来自 {{ province }}</h1>
    <p>我喜欢：{{each hobbies}} {{ $value}} {{/each}}</p>
  </body>
</html>

```

5.5. 处理网站中的静态资源

- 浏览器收到 HTML 响应内容之后，就要开始从上到下依次解析，在解析过程中，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。
- 我们为了方便的统一处理这些静态资源，所以我们约定把所有的静态资源都存放在 public 目录中
- 哪些资源能被用户访问，哪些资源不能被用户访问，我现在可以通过代码来进行非常灵活的控制
  - 注意：在服务端中，文件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 表示来获取的，
  - 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
    - ```
      <img src='/public/img/ab.jpg'/>
      ```
  - / 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 自动把 `http://127.0.0.1:3000` 拼上
  - 在服务端，不要再想文件路径了，把所有的路径都想象成 url 地址
    - 网页中所有的路径都是 url 路径，不是文件路径
  - / index.html
  - /public 整个 public 目录中的资源都允许被访问
  - 前后端融会贯通了，为所欲为

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

http
  .createServer(function (req, res) {
  var url = req.url;
  if (url === '/') {
    fs.readFile('./views/index.html',function (err, data) {
      if (err) {
        return res.end('404 Not Found.');
```
```

```
      }
      res.end(data);
    });
  } else if (url === '/post') {
    // 链接地址要变成 url地址  <a href="/post">发表留言</a>
    fs.readFile('./views/post.html', function (err, data) {
      if (err) {
        return res.end('404 Not Found.');
      }
      res.end(data);
    })
  }
  else if (url.indexOf('/public/') === 0) {
    // /public/css/main.css
    // /public/js/main.js
    // /public/lib/jquery.js
    // 统一处理：
    //      如果请求路径是以 /public/ 开头的，则我们人为你要获取 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
    fs.readFile('.' + url, function (err, data) {
      if (err) {
        return res.end('404 Not Found.');
      }
      res.end(data);
    });
  } else {
    // 其它的都处理成 404 找不到
    fs.readFile('./views/404.html', function (err, data) {
      if (err) {
        return res.end('404 Not Found.');
      }
      res.end(data);
    });
  }
})
.listen(3000, function () {
  console.log('running...')
});
```

5.6. 处理表单请求

5.6.1. 处理 get 请求

- 一次请求对应一次响应，响应结束这次请求也就结束了。再响应也就没用了
- 以前表单是怎么提交的?
- 表单中需要提交的表单控件元素必须具有 name 属性
- 表单提交分为:
    1. 默认的提交行为（同步）
        - action 就是表单提交的地址，说白了就是请求 url 地址
        - method 请求方法
            - get
            - post
    2. 表单异步提交

```
// /pinglun?name=得瑟得瑟&message=得瑟得瑟地方
// 对于这种表单提交的请求路径，由于其中具有用户动态填写的内容
// 所以你不可能通过去判断完整的 url 路径来处理这个请求
// 结论：对于我们来讲，其实只需要判定，如果你的请求路径是 /pinglun 的时候，我就认为你提交表单的请求过来了

var url = require('url');

// 得到 url地址对象，默认不会吧查询字符串转为对象（？后边的），第二个参数为 true 才可以
// 使用 url.parse 方法将路径解析为一个方便操作的对象，第二个参数为 true表示直接将查询字符串转为一个对象（通过 query 属性访问）
var obj = url.parse('/pinglun?name=得瑟得瑟&message=得瑟得瑟地方', true);

console.log(obj);
console.log(obj.query); // query：被转成对象的查询字符串
console.log(obj.pathname); // pathname：不包含查询字符串的部分的路径（该路径不包含 ？后面的内容）

```

5.6.2. 表单提交重定向

- 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
       - 301 永久重定向 浏览器会记住
       - 302 临时重定向 浏览器不会记住
  2. 在响应头中通过 Location 告诉客户端往哪儿重定向
     - setHeader

```
res.statusCode = 302;
res.setHeader('Location', '/');
res.end(); // 注意重定向之后一定要结束响应
```

- 如果客户端发现收到服务器的相应的状态码是 302 就会自动去响应头中找 Location，然后对该地址发起新的请求
  - 所以你就能看到客户端自动跳转了
- **服务端重定向只针对同步请求才有效，对表单异步提交请求无效**
  - 在服务端 app.js 中

```
res.redirect('/')   // 这是不起作用的
```

  - 只能在客户端处理，进行重定向，跳转

```
window.location.href = '/'
```

---

6. 留言板项目自己怎么去写

1. / index.html

2. 开放 public 目录中的静态资源
    - 当请求 /public/xxx 的时候，读取响应 public 目录中的具体资源
3. /post post.html
4. /pinglun

    4.1 接受表单提交数据

    4.2 存储表单提交的数据

    4.3 让表单重定向到 /

    - statusCode
    - setHeader

6.1. Node 中的 Console（REPL）

- 术语叫做 REPL
    - read 读取
    - eval 执行
    - print 输出
    - loop 循环

直接再控制台输入 `node` 进入，跟在浏览器中的控制台是差不多的

- 用来做一些 API 测试用
- 可以直接使用 核心模块，而不需要引入
- 退出：按两次 CTRL + C

6.2. 遍历的知识补充

- jQuery 的 each 和原生的 JavaScript 方法 forEach
    - forEach 是 EcmaScript 5 提供的
        - 不支持 IE 8 及以下的版本
    - jQuery 的 each
        - jQuery 2 一下的版本是兼容 IE 8 的
        - 它的 each 方法主要用来遍历 jQuery 实例对象（伪数组）
        - 同时它可以作为低版本 forEach 替代品
        - jQuery 实例对象不能使用 forEach 方法，如果想要使用必须转为数组才可以使用
            - `[].slice.call(jQuery实例对象)`

6.3. npm

- node package manager

6.3.1. npm 网站

> npmjs.com

6.3.2. npm 命令行工具

npm 的第二层含义就是一个命令行工具，只要你安装了 node 就已经安装了 npm。

npm 也有版本这个概念。

可以通过在命令行中输入：

```
npm --version
```

升级 npm（自己升级自己）：

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

6.3.3. 常用命令

- npm init
  - npm init -y 可以调过向导快速生成
- npm install
  - 一次性把 dependencies 选项中的依赖项全部安装
  - npm i（简写）
- npm install 包名
  - 只下载
  - npm i 包名（简写）
- npm install --save
  - 下载并保存依赖项（package.json 文件中的 dependencies 选项）
  - npm i -S（简写）
- npm uninstall 包名
  - 只删除，如果有依赖项会依然保存
  - npm un 包名（简写）
- npm uninstall --save 包名
  - 删除的同时也会把依赖信息也去除
  - npm un -S 包名（简写）
- npm help
  - 查看使用帮助
- npm 命令 --help
  - 查看指定命令的使用帮助
  - 例如我忘记了 uninstall 命令的简写了，这个时候，可以输入 `npm uninstall --help` 来查看使用帮助

6.3.4. 解决 npm 被墙问题

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

http://npm.taobao.org/ 淘宝的开发团队把 npm 在国内做了一个备份

安装淘宝的 cnpm：

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

接下来你安装包的时候把之前的 `npm` 替换成 `cmpm`。

举个栗子：

```
# 这里还是走国外的 npm 服务器，速度比较慢
npm install jquery

# 使用 cnpm 就会同通过淘宝的服务器来下载 jquery
cnpm install jquery
```

如果不想安装 `cnpm` 又想使用淘宝的服务器来下载：

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

但是每次手动这样加参数很麻烦，所以我们可以把这个选项加到配置文件中：

```
npm config set registry https://registry.npm.taobao.org

// 查看 npm 配置信息
npm config list
```

只要经过了上面命令的配置，则你以后所有的 `npm install` 都会通过淘宝的服务器来下载。

6.4. package.json

我们建议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` 的方式来自动初始化出来，不用手动创建

- 在使用 npm 下载包的时候加一个 `--save` （在前面加后面加都可以）
- 现在的版本不用加 `--save` 也会自动生成一个描述文件 `package-lock.json`

```
PS C:\Users\Admin\Desktop\新建文件夹> npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help init` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (新建文件夹) package.json
version: (1.0.0) 0.0.1
description: 这是一个测试项目
entry point: (index.js) main.js
test command:
git repository:
keywords:
author: Qiqiang Mu
license: (ISC)
About to write to C:\Users\Admin\Desktop\新建文件夹\package.json:

{
  "name": "package.json",
  "version": "0.0.1",
  "description": "这是一个测试项目",
  "main": "main.js",
```

```
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Qiqiang Mu",
  "license": "ISC"
}


Is this OK? (yes) yes
```

对于咱们目前来讲，最有用的是那个 `dependencies` 选项，可以用来帮我们保存第三包的依赖信息。

如果你的 `node_modules` 删除了也不用担心，我们只需要：`npm install` 就会自动把 `package.json` 中的 `dependencies` 中所有的依赖项都下载回来。

- 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` 这个选项，目的是用来保存依赖项信息

6.4.1. package.json 和 package-lock.json

npm 5 以前是不会有 `package-lock.json` 这个文件的。

npm 5 以后才加入了这个文件。

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

- npm 5 以后的版本安装包不需要加 `--save` 参数，它会自动保存依赖信息
- 当你安装包的时候，会自动创建或者是更新 `package-lock.json` 这个文件
- `package-lock.json` 这个文件会保存 `node_modules` 中的所有包的信息（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` 是用来锁定版本的
    - 如果项目依赖了 `1.1.1` 版本
    - 如果你重新 install，其实会下载最新版本，而不是 1.1.1
    - 我们的目的就是希望锁住这个 1.1.1 版本
    - 所以这个 `package-lock.json` 这个文件的另一个作用就是锁定版本号，防止自动升级到新版

7. path 路径操作模块

> 参考文档：https://nodejs.org/dist/latest-v16.x/docs/api/path.html

- path.basename
    - 获取一个路径的文件名（包含扩展名）
- path.dirname
    - 获取一个路径中的目录部分
- path.extname
    - 获取一个路径中的扩展名部分
- path.parse
    - 把一个路径转为对象
        - 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
- pase.join
    - 当你需要进行路径拼接的时候，推荐使用这个方法
- path.isAbsolute 判断一个路径是否是绝

8. 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` **动态获取** 可以用来获取当前文件模块所属目录的绝对路径
- `__filename` **动态获取** 可以用来获取当前文件的绝对路径
- `__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，因为在 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（不是bug，人家这样设计是有使用场景的——）。

- 一般在开发命令行工具的时候，这个设计是必须有用的一个特性

所以为了解决这个问题，很简单，只需要把相对路径变为绝对路径。

那这里我们就可以使用 `__dirname` 或者 `__filename` 来帮我们解决这个问题了。

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，所以推荐使用：`path.join()` 来辅助拼接。

所以为了尽量避免刚才所描述的这个问题，大家以后在文件操作中使用的相对路径都统一转换为动态的绝对路径。

- 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）
- 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，不受执行 node 命令所处路径影响

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

var app = express()

app.use('/public/', express.static(path.join(__dirname,'./public/')))
```

9. Express

原生的 http 在某些方面表现不足以应对我们的开发需求，所以我们就需要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，有很多的 Web 开发框架，我们这里以学习 `express` 为主

- http://expressjs.com/

```
// 1. 安装
// 2. 引包
var express = require('express');

// 3. 创建你的服务器应用程序
//      也就是你原来的 http.creatServer
var app = express();
```

```
8
9  // 在 Express 中开放资源就是一个 API 的事儿
10 // 公开指定目录
11 // 只要这样做了，你就可以直接通过 /public/xx 的方式访问 public 目录中的所有资源了
12 app.use('/public/', express.static('./public/'));
13 app.use('/static/', express.static('./static/'));
14
15 // 模板引擎，在 Express 也是一个 API 的事儿
16 app.get('/pinglun', function (req, res) {
17     // 在 Express 中可以直接 req.query 来获取查询字符串参数
18     // 在 Express 中使用模板引擎有更好的方式：res.render('文件名， {模板对象}')
19     // 看一下 art-template 官方文档：如何让 art-template 结合 Express 1
20 })
21
22 // 当服务器收到 get 请求 / 的时候，执行回调函数
23 app.get('/', function (req, res) {
24     res.send('hello express');
25 });
26
27 app.get('/about', function (req, res) {
28     // 中文乱码的问题，框架已经帮我们处理好了
29     res.send('你好，我是 Express! ');
30 });
31
32 // 相当于 server.listen
33 app.listen(3000, function (argument) {
34     console.log('app is running at port 3000.')
35 });
```

9.1. 起步

9.1.1. 安装

```
1 npm install --save express
```

9.1.2. hello world

```
1 const express = require('express')
2 const app = express()
3
4 app.get('/', (req, res) => res.send('Hello World!'))
5
6 app.listen(3000, () => console.log('Example app listening on port 3000!'))
```

9.1.3. 基本路由

- 路由其实就是一张表
- 这个表里面有具体的映射关系

路由：

- 请求方法
- 请求路径
- 请求处理函数

get:

```
// 当你以 GET 方法请求 / 的时候，执行对应的处理函数
app.get('/', function (req, res) {
  // 在 Express 中可以直接 req.query 来获取查询字符串参数
  res.send('Hello World!')
})
```

post:

```
// 当你以 POST 方法请求 / 的时候，执行对应的处理函数
app.post('/', function (req, res) {
  // 1. 获取表单 POST 请求体数据
  // 2. 处理
  // 3. 发送响应

  // req.query 只能拿 get 请求参数
  // console.log(req.query)

  // post
  res.send('post')
})
```

9.1.4. 静态服务

```
// /pupblic 资源   直接访问 public 里面的资源
app.use(express.static('public'))
// /files资源直接访问 files 里面的资源
app.use(express.static('files'))

// /public/xxx
app.use('/public', express.static('public'))

// / static/xxx （给 public 取个别名 叫 static）
app.use('/static', express.static('public'))


app.use('/static', express.static(path.join(_dirname, 'public')))
```

```
const express = require('express')
const app = express()

// 当以 /public/ 开头的时候，去 ./public/ 目录中找对应的资源
// 这种方式更容易辨识，推荐这种方式
app.use('/public/', express.static('./public/'))

// 必须是以 /a/public目录中的资源具体路径
app.use('/a/', express.static('./public/'))

// 当省略第一个参数的时候，则可以通过 省略 /public 的方式来访问
// 这种方式的好处就是可以省略 /public/
app.use(express.static('./public/'))

```

```
app.get('/', (req, res) => res.send('Hello World!'))

app.listen(3000, () => console.log('Example app listening on port 3000!'))
```

9.2. 在 Express 中配置使用 `art-template` 模板引擎

- art-template - GitHub仓库
- art-template 官方文档

安装：

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

配置：

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

var app = express()

// 配置使用 art-template 模板引擎
// 第一个参数表示，当渲染以 .html(这里可以自定义) 结尾的文件的时候，使用 art-template 模板引擎
// express-art-template 事专门用来在 Express  中把 art-template 整合到 Express 中的
// 虽然外面这里不需要加载 art-template ，但是也必须安装
// 原因就在于 express-art-template 它依赖了 art-template
app.engine('html', require('express-art-template'))

// Express 为 Response 响应对象提供了一个方法：render
// render 方法默认是不可以使用的，但是如果配置了模板引擎就可以使用了
// res.render('html模板名', {模板数据})
// 第一个参数不能写路径，默认会去项目中的 views 目录中查找该模板文件
// 也就是收 Express 有一个约定：开发人员把所有的视图文件都放到 views 目录中

// 如果想要修改默认的 views 目录，则可以：
// app.set('views', render函数的默认路径)
```

使用：

```
app.get('/', function(req, res) {
  // express 默认会去项目中的 views 目录找 index.html
  res.render('index.html', {
    title:'hello world'
  })
})
```

如果希望修改默认的 `views` 视图渲染存储目录，可以：

```
// 注意：第一个参数 views 千万不要写错了
app.set('views', 目录路径)
```

9.2.1. 模板继承

1. 子模版
    - 标准语法：

```
{{ include './header.art' }}
{{ include './footer.art' data }}
```

2. 模板继承：
    - 标准语法：

```
<!-- 被继承的模板 layout.html -->
<!-- 留一个坑，将来留给孩子去填坑 -->
<!-- 留坑可以留多个 -->
{{ block 'content' }}
    <h1>默认内容</h1>
{{ /block }}
```

```
<!-- index.html -->
<!-- 继承模板 layout.html 模板 -->
{{extend './layout.html'}}

{{ block 'content' }}
<div>
  <h1>
    index 页面填坑内容
  </h1>
</div>
{{ /block }}
```

9.3. 在 Express 获取表单 GET 请求体数据

Express 内置了一个 API，可以直接通过 `req.query` 来获取

```
req.query
```

9.4. 在 Express 获取表单 POST 请求体数据

在 Express 中没有内置获取表单 POST请求体的 API，这里我你们需要使用一个第三方包：`body-parser`。

安装：

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

配置：

```
var express = require('express')
// 1. 引包
```

```
var bodyParser = require('body-parser')

var app = express()

// 配置 bady-parser
// 只要加入这个配置，则在 req 请求对象上会多出来一个属性
// 也就是收你可以直接通过 req.body 来获取表单 POST 请求体数据了
// parse application/x-www-form-urlencoded
// 配置解析表单 POST 请求插件（注意：一定要在 app.use(router) 之前）
app.use(bodyParser.urlencoded({ extended: false }))
// parse application/json
app.use(bodyParser.json())

// 把路由挂载到 app 中
app.use(router)

app.use(function (req, res) {
  res.setHeader('Content-Type', 'text/plain')
  res.write('you posted:\n')
  // 可以通过 req.body 来获取表单 POST 请求体数据
  res.end(JSON.stringify(req.body, null, 2))
})

```

使用：

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

// Express 提供了一个相应方法：json
// 该方法接受一个对象作为参数，它会自动帮你把对象转为字符串再发送给浏览器
res.status(200).json({
  success: true
})
```

Express 对于没有设定的请求路径，默认会返回 Can not get xxx

- 如果你想定制这个 404
- 需要通过中间件来配置
- 只需要在自己的路由之后增加一个

```
app.use(function (req, res) {
  // 所有未处理的请求路径都会跑到这里
  // 404
})
```

9.5. 在 Express 配置使用 `express-session` 插件

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

Cookie可以用来保存一些不太敏感的数据。但是不能用来保存登陆状态

- 用户自己记住自己
- 场景：记住用户名、购物车

Session：

- 超市 -> 电子柜（服务器） 你（客户端） （二维码小票（开箱凭证）Cookie） （凭证是唯一的，不可能重复）
- 凭证一旦丢失，不可找回，如果小票丢失，你的状态也就丢失了。
- 钥匙是服务器给你的，所以这就安全了，不太容易伪造出来
- 这个时候我们可以把一些敏感的数据保存到服务端。
- 客户端只需要拿着这把钥匙就可以了。

安装：

```
npm install express-session
```

配置：

```
// 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
// 这是最简单的配置方式，暂且先不用关心里面参数的含义
app.use(session({
  // 配置加密字符串，它会在原有加密基础之上和这个字符串拼接起来去加密
  // 目的是为了增加安全性，防止客户端恶意伪造
  secret: 'itcast',
  resave: false,
  saveUninitialized: true  // true 表示无论你是否使用 Session，我都默认给你分配一把钥匙
}))
```

使用：

```
// 添加 Session 数据
req.session.foo = 'bar'

// 获取 Session 数据
req.session.foo

// 删除 Session
req.session.foo = null
// 更严谨的删除做法是 `delete` 语法
delete req.session.foo
```

提示：默认 Session 数据是内存存储的，服务器一旦重启就会丢失，真正的生产环境会把 Session 进行持久化存储。

9.6. Express 中的重定向

```
app.get('/dd', function (req, res) {
    res.redirect('/students')
})

```

9.7. Express - crud

模块如何划分：

- 模块职责要单一

9.7.1. 起步

- 初始化
- 模板处理

9.7.2. 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		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	/sudents/delete	id		处理删除请求

9.7.3. Express 实现包装路由

```
// 原始方式加载包装路由
// router.js
module.exports = function (app) {
    app.get('/students', function (req, res) {

    })
    app.get('/students/edit', function (req, res) {

    })
    app.get('/students/new', function (req, res) {

    })
}
// 这里是当函数导出，app是形参
```

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

var app = express()

router(app)
// 这里把 app 实例传进去
```

Express 提供了一种更好的方式，专门用来包装路由：

router.js：

```
// 职责：
//    根据不同的请求方法+请求路径设置具体的请求处理函数
var express = require('express')
var fs = require('fs')
var Student = require('./student')

// 1. 创建一个路由容器
var router = express.Router()

// 2. 把路由容器挂载到 router中
router.get('/students', function (req, res) {
        Student.find(function (err, students) {
            if (err) {
                return res.status(500).send('Server error.')
            }
            res.render('index.html', {
                fruits: [
                    '苹果',
                    '香蕉',
                    '橘子'
                ],
                students: students
            })
        })
    })
router.post('/students/edit', function (req, res) {

    })
router.get('/students/new', function (req, res) {

    })

// 3. 把 router 导出
module.exports = router
```

app.js：

```
var router = require('./router')
// 把路由容器挂载到 app 服务中
app.use(router)
```

9.7.4. 设计操作数据的 API 文件模块

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

```
/* student.js
数据操作文件模块
职责：操作文件中的数据，只处理数据，不关心业务 */

var fs = require('fs')

var dbPath = './bd.json'

/* 获取所有学生列表
return [] */
exports.find = function (callback) {
    // callback 中的参数
    //       第一个参数是 err
    //           成功是 null
    //           错误是 错误对象
    //       第二个参数是 结果
    //           成功是 数组
    //           错误是 undefined
    fs.readFile(dbPath, function (err, data) {
        if (err) {
            return callback(err)
        }
        callback(null, JSON.parse(data).students)
    })
}

/* 添加保存学生 */
exports.save = function (student, callback) {
    fs.readFile(dbPath, 'utf8', function (err, data) {
        if (err) {
            return callcack(err)
        }
        var students = JSON.parse(data).students

        // 处理 id 唯一的，不重复
        student.id = students[students.length - 1].id + 1

        // 把用户传递的对象保存到数组中
        students.push(student)

        // 把对象数据转换为字符串
        var fileDate = JSON.stringify({
            students: students,
        })

        // 把字符串写入文件中
        fs.writeFile(dbPath, fileData, function (err) {
            if (err) {
                // 错误就是把错误对象传递给它
                return callback(err)
            }
            // 成功就没错，所以错误对象是 null
            callback(null)
```

```
        })
    })
}

/* 更新学生 */
exports.updateById = function () {
    fs.readFile(dbPath, 'utf8', function (err, data) {
        if (err) {
            return callcack(err)
        }
        var stu = JSON.parse(data).students

        // 你要修改谁，就需要把谁找出来
        // EcmaScript 6 中的一个数组方法：find
        // 需要接收一个函数作为参数
        // 当某个遍历项符合 item.id === student.id 条件的时候，find 会终止遍历，同时返回遍历项
        students.find(function (item) {
            return item.id === student.id
        })

        for (var key in student) {
            stu[key] = student[key]
        }

        // 把对象数据转换为字符串
        var fileDate = JSON.stringify({
            students: students,
        })

        // 把字符串写入文件中
        fs.writeFile(dbPath, fileData, function (err) {
            if (err) {
                // 错误就是把错误对象传递给它
                return callback(err)
            }
            // 成功就没错，所以错误对象是 null
            callback(null)
        })
    })
}


/* 删除学生 */
exports.deleteById = function () {

};

```

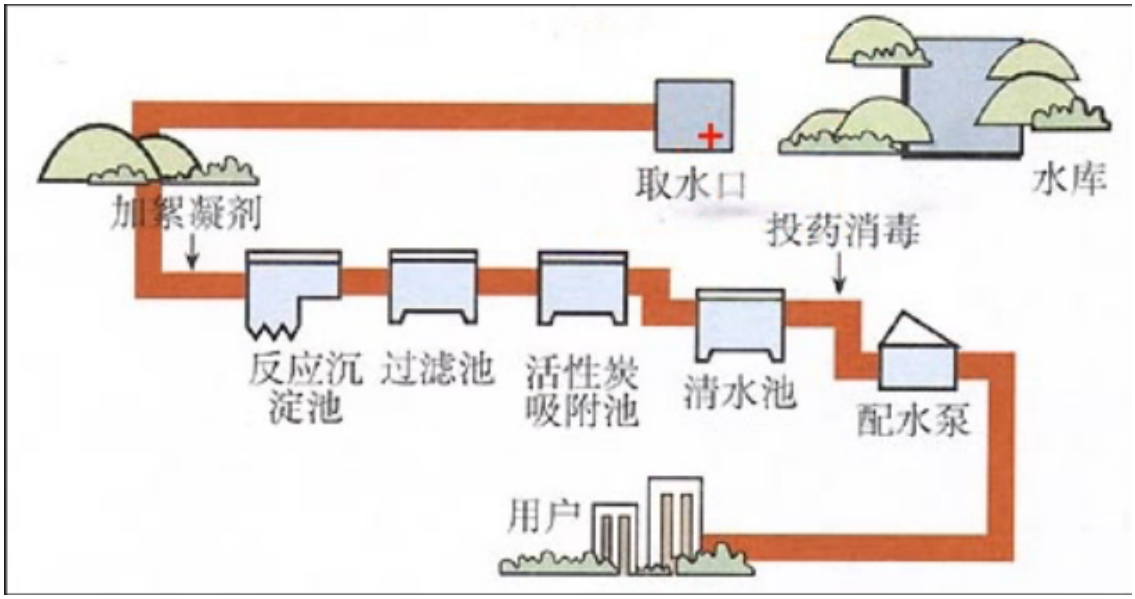
9.7.5. 自己编写的步骤

- 处理模板
- 配置开放静态资源
- 配置模板引擎
- 简单路由：/students 渲染静态页出来
- 路由设计
- 提取路由模块
- 由于接下来一系列的操作都需要处理数据，所以我们需要封装 student.js
- 先写好 student.js 文件结构
    - 查询学生列表的 API：find
    - findById
    - save
    - updateById
    - deleteById
- 实现具体功能
    - 通过路由收到请求
    - 接收请求中的数据（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发送响应
- 业务功能顺序
    - 列表
    - 添加
    - 编辑
    - 删除

9.8. Express 中间件

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

- Express middleware
- 形象说明



中间件额本质久事一个请求处理方法，我们把用户从请求到响应的整个过程分发到多个中间件中去处理，这样做的目的是提高代码的灵活性，动态可扩展性。

- **同一个请求所经过的中间件都是同一个请求对象和响应对象**

9.8.1. 应用程序级别中间件

万能匹配（不关心任何请求路径和请求方法）：

```
app.use(function (req, res, next) {
  console.log('Time:', Date.now())
  next()
})
```

只要是以 '/xxx/' 开头的：

```
app.use('/a', function (req, res, next) {
  console.log('Time:', Date.now())
  next()
})
```

9.8.2. 路由级别中间件

get:

```
app.get('/', function (req, res) {
  res.send('Hello world!')
})
```

post:

```
app.post('/', function (req, res) {
  res.send('Got a POST request')
})
```

put:

```
app.put('/user', function (req, res) {
  res.send('Got a PUt request at /user')
})
```

delete:

```
app.get('/user', function (req, res) {
  res.send('Got a DELETE request at /user')
})
```

9.8.3. 功能处理中间件

- express.static serves static assets such as HTML files, images, and so on.
- express.json parses incoming requests with JSON payloads. **NOTE: Available with Express 4.16.0+**
- express.urlencoded parses incoming requests with URL-encoded payloads. **NOTE: Available with Express 4.16.0+**

9.8.4. 配置错误处理中间件

在所有中间件的下面

```
app.get('/', function (req, res, next) {
  fs.readFile('./s/b/g/d', function (err, data) {
    if (err) {
       // 当调用 next 的时候，如果传递了参数，则直接往后找到带有 四个参数 的应用程序级别中间件
      // 当发生错误的时候，我们可以调用 next 传递错误对象
      // 然后就会被全区错误处理中间件匹配到并处理之
      next(err)
    }
  })
})

// 404 中间件，写在所有处理的下面，不影响全局处理中间件
app.use(function (req, res) {
  res.send('404.html')
})

// 全局处理中间件，写在最最下面，404处理的下面
// 注意：有 4个 参数
app.use(function (err, req, res, next) {
  res.status(500).send(err.message)
})

app.listen(3000, function () {
  console.log('app is running at port 3000.')
})
```

9.8.5. 第三方中间件

> http://express.com/en/resources/middleware.html

- bady-paser
- compression
- cookie-parse
- morgan
- response-time
- server-static
- session

中间件：处理请求的，本质是个函数

在 Express 中，对中间件有几类分类

1. 不关心请求路径和请求方法的中间件
    - 也就是收任何请求都会进入这个中间件

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

var app = express()

// 中间件：处理请求的，本质是个函数

// 在 Express 中，对中间件有几类分类

// 当请求进来，会从第一个中间件开始进行匹配
//      如果匹配，则进来
//          如果请求进入中间件之后，没有调用 next 则代码会停在当前中间件
//          如果调用了 next 则继续向后找到第一个匹配的中间件
//      如果不匹配，则继续匹配下一个中间件
// 不关心请求路径和请求方法的中间件
// 也就是收任何请求都会进入这个中间件
// 中间件本身是一个方法，该方法接收三个参数
//      Request 请求对象
//      Response 响应对象
//      next 下一个中间件
// 当一个请求进入一个中间件之后，如果不调用 next 则会停留在当前中间件
// 所以 next 是一个方法，用来调用下一个中间件的
app.use(function (req, res, next) {
    console.log('1')
    next()
})
app.use(function (req, res, next) {
    console.log('2')
})


//如果没有能匹配的中间件，则 Express 会默认输出：Cannot GET 路径

app.listen(3000, function () {
    console.log('app is running at port 3000')
})
```

2. 关心请求路径的中间件
    - 以 /xxx 开头的中间件

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

var app = express()

app.use('/a', function (req, res, next) {
    console.log(req.url)
})

app.use('/b', function (req, res, next) {
    console.log(req.url)
})

app.listen(3000, function () {
    console.log('app is running at port 3000')
})
```

3. 除了 1. 2. 两种中间件之外，还有一种最常用的
    - 严格匹配匹配请求方法和请求路径的中间件
        - `app.get`
        - `app.post`

```
app.get('/', function (req, res, next) {
    console.log('/')
    next()
})

app.get('/a', function (req, res, next) {
    console.log('/a')
    next()
})
```

10. MongoDB

10.1. 关系型数据库和非关系型数据库

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
- 非关系型数据库非常的灵活
- 有的非关系型数据库就是 key-value 对
- 但是 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）
- MongoDB 不需要设计表结构
- 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

10.2. MongoDB 数据库的基本概念

- 数据库
    - 可以有多个数据库
- 集合
    - 一个数据库中可以有多个集合（collection），简单理解就是数组（对应MySQL 中的表）
- 文档
    - 一个集合中可以有多个文档（对应 MySQL 中的表记录）
- 文档结构很灵活，没有任何限制
- MongoDB 非常灵活，不需要像 Mysql 一样先创建数据库、表、设计表结构
    - 在这里只需要：当你需要插入数据的时候，只需要指定往哪个数据库的哪个集合操作就可以了
    - 一切都由 MongoDB 来帮你自动完成建库建表这件事儿

```
{
  qq:{
    users: [
      {name: '张三', age: 15},
      {name: '李四', age: 15},
      {name: '王五', age: 15},
      {name: '张三1', age: 15},
      {name: '张三2', age: 15},
    ],
    products: [

    ],
    ...
  },
  taobao: {

  },
  baidu: {

  }
}
```

10.3. 安装

- 下载地址：https://www.mongodb.org/try/download
- 下载
- 安装
- 配置环境变量
- 最后输入 `mongod --version` 测试是否安装成功

10.4. 启动和关闭数据库

启动：

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

如果想要修改默认的数据存储目录，可以：

```
mongod --dbpath=数据存储目录路径
```

停止：

```
# 在开启服务的控制台，直接 ctrl + c 即可停止。
# 或者直接关闭开启服务的控制台也可以。
```

10.5. 连接和退出数据库

连接：

```
# 该命令默认连接本机的 MongoDB 服务
mongo
```

退出：

```
# 在连接状态输入 exit 退出
```

10.6. 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 数据库名称`
  - 切换到指定的数据库（如果没有会新建）
- 插入数据

10.7. 在 Node 中如何操作 MongoDB 数据

10.7.1. 使用官方的 `mongodb` 包来操作

> https://github.com/mongodb/node-mongodb-native

10.7.2. 使用第三方 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。

- 官网：https://mongoosejs.com/
- 官方指南：http://mongoosejs.com/docs/guide.html
- 官方 API 文档：http://mongoosejs.com/docs/api.html

安装：

```
npm i mongoose
```

hello world：

```
const mongoose = require('mongoose');

// 连接 MongoDB 数据库
mongoose.connect('mongodb://localhost:27017/test', {useNewUrlParser: true, useUnifiedTopology: true});

// 创建一个模型
// 就是在设计数据库
// MongoDB 是动态的，非常灵活，只需要在代码中设计你的数据库就可以了
// mongoose 这个包就可以让你的设计编写过程变得非常简单
const Cat = mongoose.model('Cat', { name: String });

// 实例化一个 Cat
const kitty = new Cat({ name: 'Zildjian' });

// 持久化保存 kitty 实例
kitty.save().then(() => {
    if (err) {
    console.log(err)
  } else {
    console.log('meow')
  }
});
```

10.7.3. 官方指南

10.7.3.1. 设计 Schema 发布 Model

```
var mongoose = require('mongoose')

var Schema = mongoose.Schema

// 1. 连接数据库
// 指定连接的数据库不需要存在，当你插入第一条数据之后就会自动被创建出来
mongoose.connect('mongodb://localhost/itcast')

// 2. 设计文档结构（表结构）
// 字段名称就是表结构中的属性名称
// 约束的目的是为了保证数据的完整性，不要有脏数据
var userSchema = new Schema({
    username: {
        type: String,
        required: true  // 必须有
    },
    password: {
        type: String,
        required: true
    },
    email: {
        type: String
```

```
    }
})

// 3. 将文档架构发布为模型
//    mongoose.model 方法就是用来将一个架构发布为model
//    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名称
//              mongoose 会自动将大写名称的字符串生成 小写复数 的集合名称
//              例如这里的 User 最终会变成 users 集合名称
//    第二个参数：架构 Schema

//    返回值：模型构造函数
var User = mongoose.model('User', userSchema)

// 4. 当我们有了这个模型构造函数之后，就可以使用这个构造函数对 User 集合中中的数据为所欲为了

```

10.7.3.2. 增加数据

```
var admin = new User({
    username: 'admin',
    password: '123456',
    email: 'admin@admin.com'
})

admin.save(function (err, ret) {
    if (err) {
        console.log('保存失败')
    } else {
        console.log('保存成功')
        console.log(ret)
    }
})
```

10.7.3.3. 查询数据

1. 查询所有：

```
User.find(function (err, ret) {
  if (err) {
    console.log('查询失败')
  } else {
    console.log('查询成功')
    console.log(ret)
  }
})
```

2. 按条件查询所有：

```
User.find({
  username: 'zs',
  password: '123456'
}, function (err, ret) {
  if (err) {
    console.log('查询失败')
  } else {
    console.log('查询成功')
    console.log(ret)
  }
})
```

3. 按条件查询单个：

```
User.findOne({
  username: 'zs',
}, function (err, ret) {
  if (err) {
    console.log('查询失败')
  } else {
    console.log('查询成功')
    console.log(ret)
  }
})
```

10.7.3.4. 删除数据

1. 根据条件删除所有：

```
User.remove({
  username: 'zs'
}, function (err, ret) {
  if (err) {
    console.log('删除失败')
  } else {
    console.log('删除成功')
    console.log(ret)
  }
})
```

2. 根据条件删除一个：

```
Model.findOneAndRemove(conditions, [options], [callback])
```

3. 根据 id 删除一个：

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

10.7.3.5. 更新数据

1. 根据条件更新所有：

```
Model.update(conditions, doc, [options], [callback])
```

2. 根据指定条件更新一个：

```
Model.findOneAndUpdate([conditions], [update], [options], [callback])
```

3. 根据 id 更新一个：

```
// 根据 Id 更新数据
User.findByIdAndUpdate('5f546fdg65555454g55gdg5', {
  password: '123'
}, function (err, ret) {
  if (err) {
    console.log('更新失败')
  } else {
    console.log('更新成功')
  }
})
```

例子：

```
var mongoose = require('mongoose')

mongoose.connect('mongodb://localhost/itcast')

var Schema = mongoose.Schema

var studentSchema = new Schema({
  name: {
    type: String,
    required: true
  },
  gender: {
    type: Number,
    enum: [0, 1], // 只能是 0 或者 1
    default: 0
  },
  age: {
  type: Number
  },
  hobbies: {
    type: String
  }
})
```

11. 使用 Node 操作 MySQL 数据库

安装：

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

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

// 1. 创建连接
var connection = mysql.createConnection({
  host : 'localhost',
```

```
  user : 'root',
  password : 'root',
  database : 'users'
});

// 2. 连接数据库
connection.connect();

// 3. 执行数据操作
//  查询
connection.query('SELECT * FROM `users`', function (error, results, fields)
{
  if (error) throw error;
  console.log('The solution is: ', results);
});

// 插入
connection.query('INSERT INTO users VALUES(NULL, "admin", "123456")',
function (error, results, fields) {
  if (error) throw error;
  console.log('The solution is: ', results);
});

// 4. 关闭连接
connection.end();
```

12. 异步编程

12.1. 回调函数

往往异步 API 都伴随有一个回调函数

函数：

- 一种数据类型
- 参数
- 返回值
- 一般情况下，把函数作为参数的目的久事为了获取函数内部的异步操作结果

不成立的情况：

```
function add(x, y) {
  console.log(1)
  setTimeout(function () {
    console.log(2)
    var ret = x + y
    return ret
  }, 1000)
  console.log(3)
  // 到这里执行就结束了，不会等到前面的定时器，所以直接就返回了默认值 undefined
}

console.log(add(10, 20)) // => undefined
```

不成立的情况：

```
function add(x, y) {
  var ret
  console.log(1)
  setTimeout(function () {
    console.log(2)
    ret = x + y
  }, 1000)
  console.log(3)
  return ret
}

console.log(add(10, 20)) // =>undefined
```

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

这种情况必须通过回调函数来获取

回调函数：

```
function add(x, y, callback) {
  // callback 就是回调函数
  // var x = 10
  // var y = 20
  // var callback = function (ret) { console.log(ret) }
  var ret
  console.log(1)
  setTimeout(function () {
    console.log(2)
    var ret = x + y
    callback(ret)
  }, 1000)
  console.log(3)
}

add(10, 20, function (ret) {
  console.log(ret)
})
```

基于原生 XMLHttpRequest 请求封装 get 方法：

```
function get(url, callback) {
  var oReq = new XMLHttpRequest()
    // 当请求加载成功之后要调用指定的函数
    oReq.onload = function () {
    // 我现在需要得到这里的 oReq.responseText
    callback(oReq.responseText)
    }
    oReq.open('get', 'url', true)
    oReq.send()
}

get('data.json', function (data))

```

12.2. Promise

> 参考文档：http://es6.ruanyifeng.com/#docs/promise

callback hell：

无法保证顺序的代码：

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

fs.readFile('./data/a.txt', 'utf8', function (err, data) {
  if (err) {
    // return console.log('读取失败')
    // 抛出异常
    //      1. 阻止程序的执行
    //      2. 把错误信息打印到控制台
    throw err
  }
  console.log(data)
})

fs.readFile('./data/b.txt', 'utf8', function (err, data) {
  if (err) {
    // return console.log('读取失败')
    // 抛出异常
    //      1. 阻止程序的执行
```

```
    //      2. 把错误信息打印到控制台
    throw err
  }
  console.log(data)
})

fs.readFile('./data/c.txt', 'utf8', function (err, data) {
  if (err) {
    // return console.log('读取失败')
    // 抛出异常
    //      1. 阻止程序的执行
    //      2. 把错误信息打印到控制台
    throw err
  }
  console.log(data)
})
```

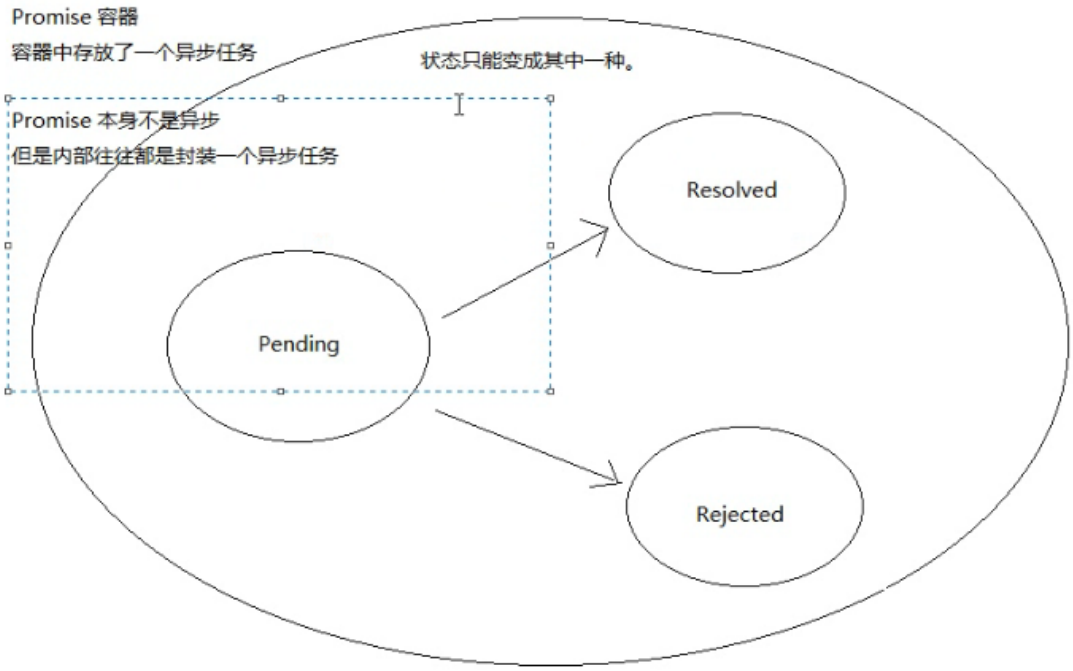
通过回调嵌套的方式来保证顺序：

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

fs.readFile('./data/a.txt', 'utf8', function (err, data) {
  if (err) {
    // return console.log('读取失败')
    // 抛出异常
    //      1. 阻止程序的执行
    //      2. 把错误信息打印到控制台
    throw err
  }
  console.log(data)
  fs.readFile('./data/b.txt', 'utf8', function (err, data) {
    if (err) {
        // return console.log('读取失败')
        // 抛出异常
        //      1. 阻止程序的执行
        //      2. 把错误信息打印到控制台
        throw err
    }
    console.log(data)
    fs.readFile('./data/c.txt', 'utf8', function (err, data) {
        if (err) {
        // return console.log('读取失败')
        // 抛出异常
        //      1. 阻止程序的执行
      //        2. 把错误信息打印到控制台
        throw err
        }
        console.log(data)
        })
    })
})
```

为了解决以上编码方式带来的问题（回调地狱嵌套），所以在 E尺码Script 6 中新增了一个 API：`Promise`。

- Promise 的英文就是承诺、保证的意思（I promise you）



Promise 基本语法：

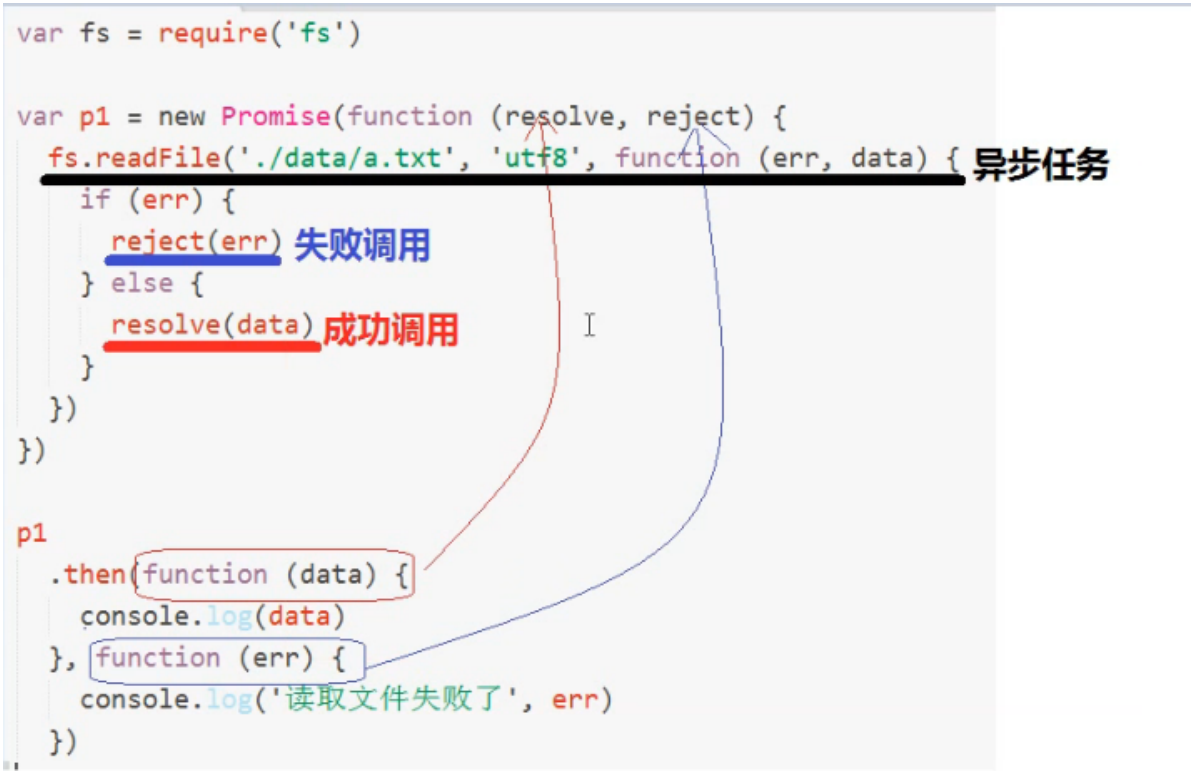
```
var fs = require('fs')

// 在 EcmaScript 6 中新增了一个 API Promise
// Promise 是一个构造函数

// 创建 Promise 容器
// 1. 给别人一个承诺 I promise you
//      Promise 容器一旦创建，就开始执行里面的代码
var p1 = new Promise(function (resolve, reject) {
  fs.readFile('./data/a.txt', 'utf8', function (err, data) {
    if (err) {
      // 失败了，承诺容器中的任务失败了
      // console.log(err)
      // 把容器的 Pending 状态变为 Rejected
      // 调用 reject 就相当于调用了 then 方法的第二个参数函数
      reject(err)
    } else {
      // 承诺容器中的任务成功了
      // console.log(data)
      // 把容器的 Pending 状态变为 Resolved
      // 也就是说这里调用的 resolve 方法实际上就是 then 方法传递的那个 function
      resolve(data)
    }
  })
})

// p1 就是哪个承诺
// 当 p1 成功了，然后（then）做指定操作
// then 方法接受的 function 就是容器中的 resolve 函数
p1
    .then(function (data) {
  console.log(data)
}, function (err) {
  console.log('读取文件失败了', err)
})
```

Promise API 代码图示：



Promise 解决异步任务嵌套：

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

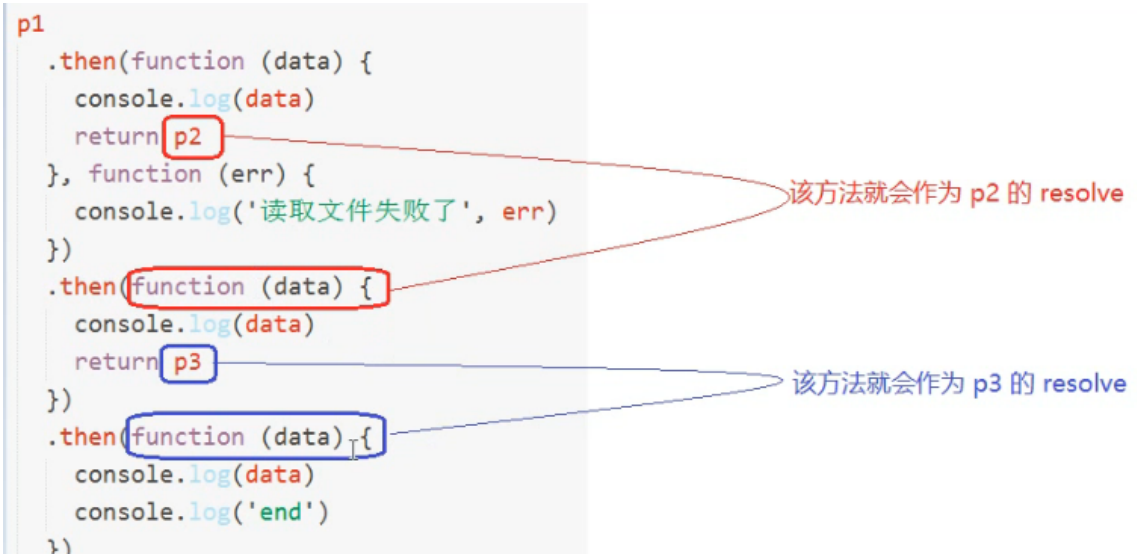
var p1 = new Promise(function (resolve, reject) {
  fs.readFile('./data/a.txt', 'utf8', function (err, data) {
    if (err) {
      reject(err)
    } else {
      resolve(data)
    }
  })
})

var p2 = new Promise(function (resolve, reject) {
  fs.readFile('./data/a.txt', 'utf8', function (err, data) {
    if (err) {
      reject(err)
    } else {
      resolve(data)
    }
  })
})

var p3 = new Promise(function (resolve, reject) {
  fs.readFile('./data/a.txt', 'utf8', function (err, data) {
    if (err) {
      reject(err)
    } else {
      resolve(data)
    }
  })
})

```

```
p1
    .then(function (data) {
    console.log(data)
    // 当 p1 读取成功的时候
    // 当前函数中 return 的结果就可以在后面的 then 中 function 接收到
    // 当你 return 123 后面就接收到 123
    //      return 'hello' 后面就接收到 'hello'
    //        没有 return 后面收到的就是 undefined
    // 上面 return 的数据没什么卵用
    // 真正有用的是： 我们可以 return 一个 Promise 对象
    // 当 return 一个 Promise 对象的时候，后续的 then 中的方法的第一个参数会作为 p2 的 resolve
    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 版本的 readFile

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

function pReadFile (filePath) {
  return new Promise(function (resolve, reject) {
    fs.readFile('filePath', 'utf8', function (err, data) {
        if (err) {
        reject(err)
        } else {
        resolve(data)
        }
    })
    })
}

```

```
pReadFile('./data/a.txt')
    .then(function (data) {
    console.log(data)
    return pReadFile('./data/b.txt')
    })
    .then(function (data) {
    console.log(data)
    return pReadFile('./data/c.txt')
    })
    .then(function (data) {
    console.log(data)
    })

```

封装 Promise 方法的 ajax：

```
function pGet(url, callback) {
  return new Promise(function (resolve, reject) {
    var oReq = new XMLHttpRequest()
        // 当请求加载成功之后要调用指定的函数
        oReq.onload = function () {
    // 我现在需要得到这里的 oReq.responseText
      callback && callback(JSON.parse(oReq.responseText))
      resolve(JSON.parse(oReq.responseText))
        }
    oReq.onerror = function (err) {
      reject(err)
    }
        oReq.open('get', 'url', true)
        oReq.send()
  })
}
```

mongoose 所有的 API 都支持 promise.js

```
// 用户注册
// 1. 判断用户是否存在
//       如果已存在，结束注册
//       如果不存在，注册（保存一条用户信息

user.findOne({
  username: '456'
})
.then(function (user) {
  if (user) {
    // 用户已存在，不能注册
    console.log('用户已存在')
  } else {
    // 用户不存在，可以注册
    return new User({
      username: 'aaa',
      password: '123',
      email: 'dffdsf'
    }).save()
  }
})
```

```
.then(function () {

})
```

13. 其它

13.1. 代码风格

13.1.1. 代码无分号问题

- 当你使用了无分号的代码风格的时候，只需要注意以下情况就不会报 TypeError 错误：
  - 当一行代码是以：`(`、`[`、`、`开头的时候，则在前面补上一个分号用以避免一些语法解析错误。
  - 所以你会发现在一些第三方的代码中能看到一上来就一个`;`开头。
- 注意：无论你的代码是有分号还是无分号的风格，都建议如果一行代码是以`(`、`[`、`` ` ``开头的，则最好都在其前面补上一个分号。
- 有些人也喜欢玩儿一些花哨的东西，例如可以使用`!`、`~`等。
- 代码风格资料：《编写可维护的 JavaScript》

13.2. 解决代码写完自动重启服务问题

我们这里可以使用一个第三方命令行工具：`nodemon`来帮我们解决频繁修改代码重启服务问题。

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

```
# 在任意目录执行该命令都可以
#也就是说，所有需要 --global 来安装的包都可以在任意目录执行
npm install --global nodemon
```

安装完毕之后，使用：

```
node app.js

# 使用 nodemon
nodemon app.js
```

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

13.3. 文件操作路径和模块标识路径问题

- 咱们所使用的所有文件操作的 API 都是异步的
  - 就像你的 ajax 请求一样
- 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，当前文件模块所处磁盘根目录
  - c:/xx/xx 绝对路径

- 文件操作路径：

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

// 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，当前文件模块所处磁盘根目录
//       c:/xx/xx   绝对路径
fs.resdFile('data/a.txt', function (err, data) {
  if (err) {
    return console.log('读取失败');
  }
  console.log(data.toString());
});

// 在模块加载中，相对路径中的 ./ 不能省

```

- 在模块加载中，相对路径中的 ./ 不能省，否则找不到模块
- 模块操作路径：

```
// 在模块加载中，相对路径中的 ./ 不能省，否则找不到模块
// 这里如果省略了 . 则也是磁盘根目录
require('/data/foo.js');

// 相对路径
require('./data/foo.js');

// 模块加载的路径中的相对路径不能省略 ./

```

13.4. 关于 JavaScript 模块化问题

-

- JavaScript 天生不支持模块化
  - require
  - exports
  - Node.js 才有的
- 在 Node 这个环境中对 JavaScript 进行了特殊的模块化支持 ConmonJS
- 在浏览器中也可以像 Node 中的模块化一样来进行编程
  - `<script>` 标签来引用加载，而且你还必须考虑加载的顺序问题
  - require.js 第三方库 AMD
  - sea.js 第三方库 CMD
- 无论是 Com模拟JS、AMD、CMD、UMD、EcmaScript 6 的 Modules 官方规范
  - 都是为了解决 JavaScript 的模块化问题
  - CommonJS、AMD、CMD 都是民间搞出来的
  - EcmaScript 是官方规范定义
  - EcmaScript 在 2015 年发布了 EcmaScript 2016 官方标准
  - 其中就包含了对 JavaScript 模块化的支持
  - 也就是说语言天生就支持了

- 但是虽然标准已经发布了，但是很多 JavaScript 运行环境还不支持
- Node 也是在 8.5 版本之后才对 EcmaScript 6 module 进行了支持
- 后面学 Vue 的时候会学习
- less 编译器 -> css
- EcmaScript 6 -> 编译器 -> EcmaScript 5
- 目前的前端情况都是使用很多新技术，然后利用编译工具打包可以在低版本浏览器运行。
- 使用新技术的目的就是为了提高效率，增加可维护性

13.5. 表单同步提交和异步提交

表单具有默认的提交行为，默认是同步的。同步表单提交，浏览器会锁死（转圈儿）等待服务端的响应结果。

表单的同步提交之后，无论服务端你响应的是什么，都会直接把响应的结果覆盖掉当前页面

后来有人想到了一种办法，来解决这个问题。

```
if (data) {  // 如果存在，重新渲染注册页面
  return res.render(`register.html`, {
    err_message: '邮箱或昵称已存在',
    form: body
    })
}
```

13.6. 数组的 reduce 方法

14. 用户密码加密（md5）

安装：

```
npm install blueimp-md5
```

使用：

```
var md5 = require('blueimp-md5')

// 对密码进行 md5 重复加密
body.password = md5(md5(body.password) + 'itcast')
```

15. Node 综合 Web 案例

15.1. 目录结构

```
.
|-- app.js
|-- controllers
|-- models                          存储使用 mongoose 设计的数据模型
|-- node_modules                    第三方包
|-- package.json                    包描述文件
|-- package-lock.json               第三方包版本锁定文件（npm 5 以后才有）
|-- public                          公共的静态资源
|-- README.md                       项目说明文档
|-- routers                         如果业务比较多，代码量大，最好把路由按照业务的分类存储到 routes 目录中
|-- router.js                       简单一点把所有的路由都放到这个文件
|-- views                           存储视图目录
```

15.2. 模板页

- art-tempalte 子模版
- art-template 模板继承

15.3. 路由设计

路径	方法	get 参数	post 参数	是否需要登陆	备注
/	GET				渲染首页
/register	GET				渲染注册页面
/register	POST		email、nickname、password		处理注册请求
/login	GET				渲染登陆页面
/login	POST		email、password		处理登陆请求
/logout	GET				处理退出请求

15.4. 模型设计

15.5. 功能实现

15.6. 书写步骤

- 创建目录结构
- 整合静态页 - 模板页
  - include
  - block
  - extend
- 设计用户登陆、退出、注册的路由

- 用户注册
  - 先处理好客户端页面的内容（表单控件的 name、手机表单数据、发起请求）
  - 服务端
    - 获取客户端表单请求数据
    - 操作数据库
      - 如果有错，发送 500 告诉客户端服务器错了
      - 其它的根据你的业务发送不同的响应数据
- 用户登陆
- 用户退出