



COMPANY INFORMATION
Lorem ipsum dolor sit amet



SERVICES & SOLUTIONS
Lorem ipsum dolor sit amet



DAILY NEWSLETTER
Lorem ipsum dolor sit amet



WORLDWIDE PARTNERS
Lorem ipsum dolor sit amet



CUSTOMER SUPPORT
Lorem ipsum dolor sit amet

Node.js 开发实战详解

黄丹华 等编著

腾讯Web前端工程师全面揭秘原生Node.js的开发实践
不借助任何第三方框架，通过编写原生代码，讲解Node.js应用开发

- 深入详解Node.js原生文档，根据原生API实践和大量应用实例，详细分析Node.js的开发过程，了解原生Node.js的API应用
- 全面涵盖Node.js基础知识、模块与NPM、Web应用、UDP服务、异步编程思想、异常处理过程、操作数据库的方法、框架开发与应用、开发工具等
- 注重实践，讲解时穿插了430多个代码小示例，提供了30多个编程实践练习题及解答，还介绍了5个大型系统的开发，并赠送8小时教学视频（请下载）



Web 开发典藏大系

Node.js 开发实战详解

黄丹华 等编著

清华大学出版社

北 京

内 容 简 介

本书由浅入深,全面、系统地介绍了 Node.js 开发技术。书中提供了大量有针对性的实例,供读者实践学习,同时提供了大量的实践练习题及详尽的解答,帮助读者进一步巩固和提高。本书重在代码实践,阅读时应多注重实践编程。本书提供 8 小时配套教学视频及实例源代码,便于读者高效、直观地学习。

本书共分为 11 章。涵盖的主要内容有: Node.js 的概念、应用场景、环境搭建和配置、异步编程; Node.js 的模块概念及应用、Node.js 的设计模式; HTTP 简单服务的搭建、Node.js 静态资源管理、文件处理、Cookie 和 Session 实践、Crypto 模块加密、Node.js 与 Nginx 配合; UDP 服务器的搭建、Node.js 与 PHP 之间合作; Node.js 的实现机制、Node.js 的原生扩展与应用; Node.js 的编码习惯; Node.js 操作 MySQL 和 MongoDB; 基于 Node.js 的 Myweb 框架的基本设计架构及实现; 利用 Myweb 框架实现一个简单的 Web 聊天室; 在线聊天室案例和在线中国象棋案例的实现; Node.js 的日志模块、curl 模块、crontab 模块、forever 模块、xml 模块和邮件发送模块等应用工具。

本书非常适合从事编程开发的学生、教师、广大科研人员和工程技术人员研读。建议阅读本书的读者对 JavaScript 的语法和 PHP 的相关知识有一定的了解。当然,如果你是初学者,本书也是一本难得的参考书。

本书封面贴有清华大学出版社防伪标签,无标签者不得销售。

版权所有,侵权必究。侵权举报电话: 010-62782989 13701121933

图书在版编目(CIP)数据

Node.js 开发实战详解 / 黄丹华等编著. —北京: 清华大学出版社, 2014
(Web 开发典藏大系)
ISBN 978-7-302-34947-1

I. ①N… II. ①黄… III. ①JAVA 语言—程序设计 IV. ①TP312

中国版本图书馆 CIP 数据核字(2013)第 321349 号

责任编辑: 夏兆彦

封面设计: 欧振旭

责任校对: 胡伟民

责任印制:

出版发行: 清华大学出版社

网 址: <http://www.tup.com.cn>, <http://www.wqbook.com>

地 址: 北京清华大学学研大厦 A 座 邮 编: 100084

社总机: 010-62770175 邮 购: 010-62786544

投稿与读者服务: 010-62776969, c-service@tup.tsinghua.edu.cn

质量反馈: 010-62772015, zhiliang@tup.tsinghua.edu.cn

印 刷 者:

装 订 者:

经 销: 全国新华书店

开 本: 185mm×260mm 印 张: 24.5 字 数: 615 千字
版 次: 2014 年 4 月第 1 版 印 次: 2014 年 4 月第 1 次印刷

印 数: 1~ 000

定 价: 元

产品编号: 056313-01

前 言

Node.js 是一个 JavaScript 运行环境（runtime）。实际上它是对 GoogleV8 引擎（应用于 Google Chrome 浏览器）进行了封装。由于其拥有异步非阻塞、环境搭建简单、实践应用快等特性，使得其在新一代编程开发中更为流行。同时，由于 Node.js 基于 JavaScript 语法，因此在学习 Node.js 时也可以了解和学习 JavaScript 的语法，拓宽和加深读者对 Web 前端开发技术的理解。

当前，Node.js 主要应用于 HTTP Web 服务器的搭建和快速实现的独立服务器应用。在实践项目中，Node.js 更适合做一些小型系统服务或者一些大项目的部分功能的实现。由于其版本不稳定，很多公司中主要将其应用于一些小项目中。如果以后其版本能够更加稳定可控，相信会有更多的公司将其应用于各种项目和服务中。

目前，国内 Node.js 的相关技术图书还非常稀缺。为了给想要学习 Node.js 开发技术的人员一个必要的指导，笔者编写了这本书。本书既注重基础知识讲解，又非常注重编程实践练习，讲解时给出了有针对性的实例，各章还给出了一些编程实践练习题。相信读者通过阅读本书，不仅可以全面掌握 Node.js 开发技术，还可以不需要借助任何框架而独立运用 Node.js 实现 HTTP Web 服务器的相关功能，从而摒弃对框架的依赖，进一步加深读者自我研发及独立思考的能力。

本书的特点

1. 编码不依赖任何框架

为了便于读者更好地了解原生 Node.js 的开发实践，本书没有借助任何其他框架来讲解 Web 实践应用，书中的所有模块都是通过编写原生代码来实现。

2. 结构合理，内容全面、系统

本书全面、系统地介绍了 Node.js 从入门到编程实践的各种技术，涵盖 Node.js 网络编程、Node.js 与数据库等方方面面的知识。

3. 叙述详实，例程丰富

本书提供了大量例程，便于读者实践演练。书中的每个例子都经过精挑细选，有很强的针对性。这些实例都给出了完整的代码和详细的代码注释。这些代码非常简洁和高效，便于读者学习和调试。当然，读者也可以直接重用这些代码来解决自己的问题。

4. 结合实际，编程技巧贯穿其中

本书写作时特意给出了大量的实用编程技巧，对这些编程技巧的灵活使用，将会使读

者的开发事半功倍。

5. 语言通俗，图文并茂

本书以通俗易懂的语言讲解每一个技术点和实例，讲解时还穿插了大量效果图，并给出了程序的运行结果插图，便于读者更加直观地学习和理解。

6. 大量习题，详尽解答

本书提供了大量的编程实践练习题和详尽的解答，便于读者进一步巩固和加深所学的各个技术点，从而达到更好的学习效果。

7. 配多媒体教学视频

为了便于读者更加高效、直观地理解书中的技术点，作者为本书专门录制了 8 小时配套的多媒体教学视频。这些视频和本书源代码一起收录于配书光盘中。虽然视频录制的设备条件有限（主要靠耳麦），但依然相信这些视频可以给读者的学习提供有益的帮助。

本书内容体系

本书共分 11 章，各章的具体内容介绍如下。

第 1 章主要介绍了 Node.js 的概念、配置、第一个 Node.js 程序 Hello World 的实现及异步编程思想等。

第 2 章主要介绍了 Node.js 中的模块的概念，以及 Node.js 中 exports 和 module.exports 之间的联系和区别。Node.js 中的 NPM 模块包含 request、socket.io、express、jade 和 forever 模块。Node.js 中的设计模式包含单例、适配器和装饰模式。

第 3 章主要介绍了 Node.js 的 Web 开发技术。包含 HTTP 简单服务搭建、Node.js 静态资源服务器实现、文件处理、Cookie 和 Session 实践、Crypto 模块加密及 Node.js 与 Nginx 配合实践等。

第 4 章主要介绍了 Node.js 中 UDP 服务器的搭建实践及 Node.js 与 PHP 之间的合作方式。

第 5 章主要介绍了 Node.js 中 require 机制的实现、Node.js 的 C++ 扩展（同步和异步接口）编译实践方法。

第 6 章主要介绍了一些关于 Node.js 的编码习惯。

第 7 章主要介绍了利用 Node.js 操作 MySQL 和 MongoDB 的实例，其中包含实现 Node.js 的两个基类分别对应于 MySQL 和 MongoDB。另外，还介绍了 MySQL 和 MongoDB 环境的搭建，以及两个数据中 Node.js 的 NPM 模块。

第 8 章主要从框架开发的角度介绍了一个基于 Node.js 的 Myweb 框架的基本设计架构及其实现的功能，以及该框架的实现。其中用到了 express 模块和 jade 解析模板，可帮助读者进一步了解 Node.js 的 Web 应用开发和 express 框架的应用。

第 9 章主要从框架应用的角度介绍了如何使用框架做一个简单的项目开发，即利用第 8 章的 MyWeb 1.0 框架实现一个简单的 Web 聊天室 MyChat 应用。

第 10 章主要介绍了两个实例：在线聊天室和联网在线中国象棋。这两个应用都是用

本书中自我实践的代码框架 MyWeb 2.0 来实现的。

第 11 章主要介绍了 Node.js 的一些应用工具，包含日志模块、curl 模块、crontab 模块、forever 模块、xml 模块和邮件发送模块。

本书读者对象

- ☐ Node.js 初学者；
- ☐ PHP 或者 JavaScript 程序员；
- ☐ 想全面、系统地学习 Node.js 的人员；
- ☐ Node.js 技术爱好者；
- ☐ 利用 Node.js 进行开发的技术人员；
- ☐ 大中专院校的学生和老师；
- ☐ 相关培训学校的学员。

本书作者

本书由黄丹华主笔编写。其他参与编写和资料整理的人员有陈杰、陈贞、樊俊、高彩丽、高莹婷、管磊、郭丽、韩亚、李红、李龙海、梁伟、刘忆智、曲宝军、孙忠贤、唐正兵、王全政、王勇浩、武文琛、徐学英、闫伍平、于轶、占海明、张帆。

致谢

本书在写作过程中参阅了大量的相关资料。在此对原文的作者、相关网站及社区表示特别的感谢！没有这些资料，笔者完成本书将会需要花费更多的时间，本书的推出时间也会延迟。下面给出本书参考的主要资料及来源。

CSDN 社区中的《程序员如何说服老板采用 Node.js》：由于 Node.js 已经越来越多地被程序员和公司关注，基于此 CSDN 有针对性地写了这篇文章，系统地告诉程序员在适当的机会下从哪些方面入手才能让团队及老板来支持 Node.js 的项目实现。本书中多处参考了该文章。

HACK SPARROW 的 *Node.js Module - exports vs module.exports* 和 *Create NPM Package - Node.js Module*：本书中介绍的 exports 与 module.exports 之间的区别和联系参考了英文资料 *Node.js Module - exports vs module.exports*；本书中介绍的 Node.js NPM 模块发布参考了文章 *Create NPM Package - Node.js Module*。

CNode 社区 ctrlacv 的《静态文件服务器代码整理》：本书实现的一个静态服务器参考了国内知名 Node.js 社区 CNode 中的 ctrlacv 文章《静态文件服务器代码整理》。

田永强编著的《深入浅出 Node.js (三)：深入 Node.js 的模块机制》：本书在深入 Node.js 中介绍的 require 机制实现则是参阅了田永强的文章《深入浅出 Node.js (三)：深入 Node.js 的模块机制》编写而成。

移动开发博客 lishen 的《编写 Node.js 原生扩展》：本书中介绍的关于实现 Node.js 原生扩展模块方法，主要参考了国内网站移动开发博客 lishen 的文章《编写 Node.js 原生扩展》。

笔者在本书中给出了大量的脚注，注明这些资料的来源。其目的—是表示对原作者的尊重和感谢；二是便于读者查阅和学习。

本书的编写对笔者而言是一个“浩大的工程”。虽然笔者投入了大量的精力和时间，但只怕百密难免一疏。若有任何疑问或疏漏，请发邮件至 bookservice2008@163.com。最后祝读者读书快乐！

编著者

目 录

第 1 章 Node.js 基础知识	1
1.1 概述	1
1.1.1 Node.js 是什么	1
1.1.2 Node.js 带来了什么	1
1.2 Node.js 配置开发	3
1.2.1 Windows 配置	3
1.2.2 Linux 配置	5
1.2.3 Hello World	6
1.2.4 常见问题	7
1.3 异步编程	8
1.3.1 同步调用和异步调用	8
1.3.2 回调和异步调用	11
1.3.3 获取异步函数的执行结果	12
1.4 本章实践	12
1.5 本章小结	14
第 2 章 模块和 NPM	16
2.1 什么是模块	16
2.1.1 模块的概念	16
2.1.2 Node.js 如何处理模块	16
2.1.3 Node.js 实现 Web 解析 DNS	18
2.1.4 Node.js 重构 DNS 解析网站	24
2.1.5 exports 和 module.exports	28
2.2 NPM 简介	30
2.2.1 NPM 和配置	30
2.2.2 Express 框架	31
2.2.3 jade 模板	33
2.2.4 forever 模块	36
2.2.5 socket.io 模块	38
2.2.6 request 模块	40
2.2.7 Formidable 模块	43
2.2.8 NPM 模块开发指南	45
2.3 Node.js 设计模式	47

2.3.1	模块与类	47
2.3.2	Node.js 中的继承	49
2.3.3	单例模式	55
2.3.4	适配器模式	57
2.3.5	装饰模式	59
2.3.6	工厂模式	61
2.4	本章实践	63
2.5	本章小结	75
第 3 章	Node.js 的 Web 应用	77
3.1	HTTP 服务器	77
3.1.1	简单的 HTTP 服务器	77
3.1.2	路由处理	81
3.1.3	GET 和 POST	84
3.1.4	GET 方法实例	84
3.1.5	POST 方法实例	87
3.1.6	HTTP 和 HTTPS 模块介绍	90
3.2	Node.js 静态资源管理	91
3.2.1	为什么需要静态资源管理	92
3.2.2	Node.js 实现简单静态资源管理	93
3.2.3	静态资源库设计	96
3.2.4	静态文件的缓存控制	99
3.3	文件处理	104
3.3.1	File System 模块介绍	104
3.3.2	图片和文件上传	108
3.3.3	jade 模板实现图片上传展示功能	112
3.3.4	上传图片存在的问题	116
3.3.5	文件读写	117
3.4	Cookie 和 Session	122
3.4.1	Cookie 和 Session	122
3.4.2	Session 模块实现	123
3.4.3	Session 模块的应用	126
3.5	Crypto 模块加密	127
3.5.1	Crypto 介绍	127
3.5.2	Web 数据密码的安全	131
3.5.3	简单加密模块设计	132
3.6	Node.js+Nginx	136
3.6.1	Nginx 概述	137
3.6.2	Nginx 的配置安装	137
3.6.3	如何构建	142

3.7	文字直播实例	145
3.7.1	系统分析	145
3.7.2	重要模块介绍	147
3.8	扩展阅读	155
3.9	本章实践	159
3.10	本章小结	173
第 4 章	Node.js 高级编程	175
4.1	构建 UDP 服务器	175
4.1.1	UDP 模块概述	175
4.1.2	UDP Server 构建	176
4.2	UDP 服务器应用	179
4.2.1	应用分析介绍	180
4.2.2	UDP Server 端（图片处理服务器）实现	181
4.2.3	UDP Client 端（Web Server）	184
4.2.4	Jade 页面实现	186
4.2.5	应用体验	187
4.3	Node.js 与 PHP 合作	189
4.3.1	UDP 方式	189
4.3.2	脚本执行	191
4.3.3	HTTP 方式	191
4.3.4	三种方式的比较	192
4.4	本章实践	193
4.5	本章小结	196
第 5 章	深入 Node.js	199
5.1	Node.js 的相关实现机制	199
5.2	Node.js 原生扩展	202
5.2.1	Node.js 扩展开发基础 V8	202
5.2.2	Node.js 插件开发介绍	204
5.3	Node.js 异步扩展开发与应用	205
5.4	本章实践	212
5.5	本章小结	214
第 6 章	Node.js 编码习惯	216
6.1	Node.js 规范	216
6.1.1	变量和函数命名规范	216
6.1.2	模块编写规范	219
6.1.3	注释	220
6.2	Node.js 异步编程规范	221
6.2.1	Node.js 的异步实现	221

6.2.2	异步函数的调用	224
6.2.3	Node.js 异步回调深度	226
6.2.4	解决异步编程带来的麻烦	227
6.3	异常逻辑的处理	231
6.3.1	require 模块对象不存在异常	231
6.3.2	对象中不存在方法或者属性时的异常	233
6.3.3	异步执行的 for 循环异常	234
6.3.4	利用异常处理办法优化路由	236
6.3.5	异常情况汇总	240
6.4	本章实践	241
6.5	本章小结	241
第 7 章	Node.js 与数据库	243
7.1	两种数据库介绍	243
7.1.1	MySQL 介绍	243
7.1.2	MongoDB 模块介绍	247
7.2	Node.js 与 MySQL	250
7.2.1	MySQL 安装配置应用	250
7.2.2	MySQL 数据库接口设计	251
7.2.3	数据库连接	252
7.2.4	数据库插入数据	254
7.2.5	查询一条数据记录	256
7.2.6	修改数据库记录	258
7.2.7	删除数据库记录	259
7.2.8	数据条件查询	260
7.3	Node.js 与 MongoDB	262
7.3.1	MongoDB 的安装以及工具介绍	263
7.3.2	MongOD 的启动运行方法	264
7.3.3	MongoDB 的启动运行	266
7.3.4	MongoDB 数据库接口设计	268
7.3.5	数据插入	272
7.3.6	数据修改	274
7.3.7	查询一条数据	276
7.3.8	删除数据	278
7.3.9	查询数据	279
7.4	MySQL 与 MongoDB 性能	281
7.4.1	测试工具及测试逻辑	282
7.4.2	MySQL 性能测试代码	282
7.4.3	MongoDB 性能测试代码	283
7.4.4	性能测试数据分析	283

7.5	本章实践	285
7.6	本章小结	289
第 8 章	MyWeb 框架介绍	290
8.1	MyWeb 框架介绍	290
8.1.1	MyWeb 框架涉及的应用	290
8.1.2	MyWeb 框架应用模块	291
8.2	MyWeb 源码架构	292
8.2.1	框架 MVC 设计图	292
8.2.2	框架文件结构	293
8.2.3	扩展阅读之更快地了解新项目	294
8.3	框架源码分析	295
8.3.1	框架入口文件模块	295
8.3.2	路由处理模块	297
8.3.3	Model 层基类	299
8.3.4	Controller 层基类	301
8.4	本章实践	302
8.5	本章小结	302
第 9 章	框架应用 MyChat	304
9.1	编码前的准备	304
9.1.1	应用分析	305
9.1.2	应用模块	305
9.1.3	功能模块设计	307
9.2	系统的编码开发	309
9.2.1	Model 层	309
9.2.2	Controller 层	311
9.2.3	View 层	316
9.3	项目总结	318
9.3.1	forever 启动运行项目	318
9.3.2	系统应用体验	320
9.3.3	系统开发总结	323
9.4	扩展阅读之 MyWeb 2.0 的介绍	323
9.5	本章实践	325
9.6	本章小结	325
第 10 章	Node.js 实例应用	326
10.1	实时聊天对话	326
10.1.1	系统设计	326
10.1.2	系统的模块设计	327
10.1.3	系统编码实现	328

10.2	联网中国象棋游戏	332
10.2.1	系统设计	333
10.2.2	系统的模块设计	334
10.2.3	系统编码实现	334
10.2.4	系统体验	337
10.3	本章小结	339
第 11 章	Node.js 实用工具	340
11.1	日志模块工具	340
11.1.1	日志模块介绍	340
11.1.2	日志模块实现	341
11.1.3	日志模块应用	345
11.2	配置文件读取模块	347
11.2.1	配置文件解析模块介绍	347
11.2.2	配置文件解析模块实现	348
11.3	curl 模块	352
11.3.1	curl 模块介绍	352
11.3.2	curl 模块实现	353
11.3.3	curl 模块应用	356
11.4	crontab 模块	357
11.4.1	crontab 模块介绍	358
11.4.2	crontab 模块设计实现	358
11.4.3	crontab 模块应用	361
11.5	forever 运行脚本	362
11.5.1	forever 运行脚本介绍	362
11.5.2	forever 运行脚本实现	363
11.5.3	forever 运行脚本应用	366
11.6	xml 模块的应用	367
11.6.1	xml 解析模块介绍	368
11.6.2	xml 模块设计实现	369
11.6.3	xml 模块应用	371
11.7	邮件发送模块应用	374
11.7.1	邮件模块介绍	374
11.7.2	邮件模块设计实现	374
11.7.3	邮件模块应用	376
11.8	本章小结	377

第3章 Node.js 的 Web 应用

本章将介绍 Node.js 的 Web 应用，主要包括 HTTP 服务器、Node.js 静态资源管理、Node.js 的文件处理功能、Cookie 和 Session 的应用、Node.js 安全加密，最后介绍 Node.js 如何与 Nginx 搭档。

本章将会结合“文字直播 Web 应用”来贯穿本章的知识点。文字直播应用是一个长连接多请求的应用，Node.js 在这方面具有非常大的优势，因此本章使用该例子来介绍本章知识点。该应用使用 Node.js 开发可以在很大程度上降低服务器压力，同时给予用户一个更好的在线体验。文字直播 Web 应用需求简单介绍如下。

- ❑ 用户：直播员（需登录），游客。
- ❑ 直播方式：直播员登录后台，输入相应的直播信息，可包含图片和文字。
- ❑ 技术统计：需在线实时记录运动员数据。
- ❑ 游客讨论：游客可实时进行在线讨论。
- ❑ 微博分享：游客可通过腾讯微博和新浪微博分享直播内容。

3.1 HTTP 服务器

本节将介绍如何应用 Node.js 创建一个 HTTP 服务器来处理客户端的 POST 和 GET 数据请求，以及服务器端 Node.js 如何实现 url 路由请求处理。最后会介绍 Node.js 模块中的 HTTP 和 HTTPS。重点是介绍如何实现 Node.js 的服务器路由。

3.1.1 简单的 HTTP 服务器

创建一个 HTTP 服务器的相关知识点在 Node.js 的配置开发已经有所介绍。应用 Node.js HTTP 模块中的 `createServer()` API。代码如下：

```
var http = require('http');
/* 应用 Node.js 的原生模块 HTTP 来实现 Web 服务器创建 */
http.createServer(function(req, res) {
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end('Hello World\n');
}).listen(1337, "127.0.0.1");
console.log('Server running at http://127.0.0.1:1337/');
```

`http.createServer()`接收一个 `request` 事件函数，该事件函数有两个参数 `request` 和

response, request 是 `http.ServerRequest`¹的实例对象, response 则为 `http.ServerResponse`²实例对象。request 对象主要是获取请求资源信息, 包括请求的 url、客户端参数、资源文件、header 信息、HTTP 版本、设置客户端编码等。Response 对象主要是响应客户端请求数据, 包括 HTTP 的 header 处理、HTTP 请求返回码、响应请求数据等。

`http.createServer()`调用返回的是一个 server 对象, server 对象拥有 listen 和 close 方法, listen 方法可以指定监听的 IP 和端口。

实例介绍: 创建一个 HTTP 服务器, 获取并输出请求 url、method 和 headers, 同时根据请求资源做不同的输出。

- ❑ 当请求'/index', 返回 200, 并且返回一个 html 页面数据到客户端。
- ❑ 当请求'/img', 返回 200, 并且返回一个图片数据。
- ❑ 若为其他情况, 则返回 404, 并输出'can not find source'。

先创建 HTTP 服务器, 使用 switch 判断请求资源类型分配。代码如下:

```
/* http.js */
var http = require('http'),
    fs    = require('fs'),
    url   = require('url');
/* 创建 http 服务器 */
http.createServer(function(req, res) {
  /* 获取 Web 客户端请求路径 */
  var pathname = url.parse(req.url).pathname;
  /* 打印客户端请求 req 对象中的 url、method 和 headers 属性 */
  console.log(req.url);
  console.log(req.method);
  console.log(req.headers);
  /* 根据 pathname, 路由调用不同处理逻辑 */
  switch(pathname){
    case '/index' : resIndex(res); // 响应 HTML 页面到 Web 客户端
    break;
    case '/img'   : resImage(res); // 响应图片数据到 Web 客户端
    break;
    default       : resDefault(res); // 响应默认文字信息到 Web 客户端
    break;
  }
}).listen(1337);
```

【代码说明】

- ❑ `var pathname = url.parse(req.url).pathname`: 获取客户端请求路径。
- ❑ `console.log(req.url)`: 输出请求 url。
- ❑ `console.log(req.method)`: 输出请求方法。
- ❑ `console.log(req.headers)`: 输出请求 header 信息。
- ❑ `resIndex(res)`: 判断请求参数是否为/index, 如果是就执行 `resIndex(res)`。
- ❑ `resImage(res)`: 判断请求参数是否为/img, 如果是就执行 `resImage(res)`。
- ❑ `resDefault(res)`: 返回 404 not find。

1 参考网站 http://nodejs.org/api/http.html#http_class_http_serverrequest。

2 参考网站 http://nodejs.org/api/http.html#http_class_http_serverresponse。

记住要把 `res` 参数传递到处理返回函数，如果需要执行 HTTP 响应请求时，也必须将 `req` 传递到处理函数。代码如下：

```
/**
 *
 * @desc 创建 resIndex 响应首页 html 函数
 * @parameters res HTTP 响应对象
 */
function resIndex(res){
  /* 获取当前 index.html 的路径 */
  var readPath = __dirname + '/' +url.parse('index.html').pathname;
  var indexPage = fs.readFileSync(readPath);
  res.writeHead(200, { 'Content-Type': 'text/html' });
  res.end(indexPage);
}

/**
 *
 * @desc 创建 resImage 响应 image 函数
 * @parameters res HTTP 响应对象
 */
function resImage(res){
  /* 获取当前 image 的路径 */
  var readPath = __dirname + '/' +url.parse('logo.png').pathname;
  var indexPage = fs.readFileSync(readPath);
  res.writeHead(200, { 'Content-Type': 'image/png' });
  res.end(indexPage);
}

/**
 *
 * @desc 创建 resDefault 响应 404 函数
 * @parameters res HTTP 响应对象
 */
function resDefault(res){
  res.writeHead(404, { 'Content-Type': 'text/plain' });
  res.end('can not find source');
}
```

【代码说明】

- ❑ `resIndex(res)`处理并返回'Content-Type': 'text/html'的 html 页面。
- ❑ `esImage(res)`处理并返回'Content-Type': 'image/png'的 png 图片。
- ❑ `resDefault(res)`处理并返回 404 输出'can not find source'。

执行 `http.js`，打开浏览器，输入不同的请求资源，查看和比较不同的返回结果。

```
node http.js
```

(1) 打开浏览器，输入 `127.0.0.1:13371`，如图 3-1 所示返回了 404，并输出了'can not find source'。

1 本文输入的 `192.168.1.120:1337` 为个人的一个测试服务器，和本地服务器 IP 地址 `127.0.0.1:1337` 是一致的，读者输入的是 `127.0.0.1:1337`。

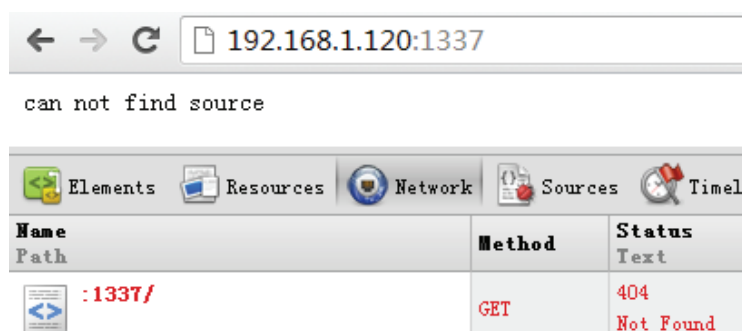


图 3-1 浏览 http://127.0.0.1:1337 返回 Web 页面图

(2) HTTP 新增请求路径/index, 127.0.0.1:1337/index, 如图 3-2 所示返回 200, 并输出 index.html 页面内容。

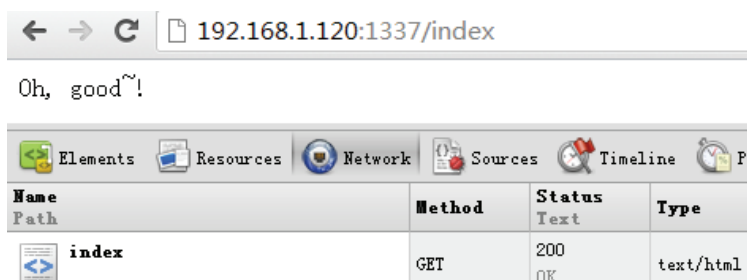


图 3-2 浏览 127.0.0.1:1337/index 返回 Web 页面图

(3) 改变 HTTP 路径为/img, 127.0.0.1:1337/img, 如图 3-3 所示返回 200, 并输出了一个 png 的 logo 图片。

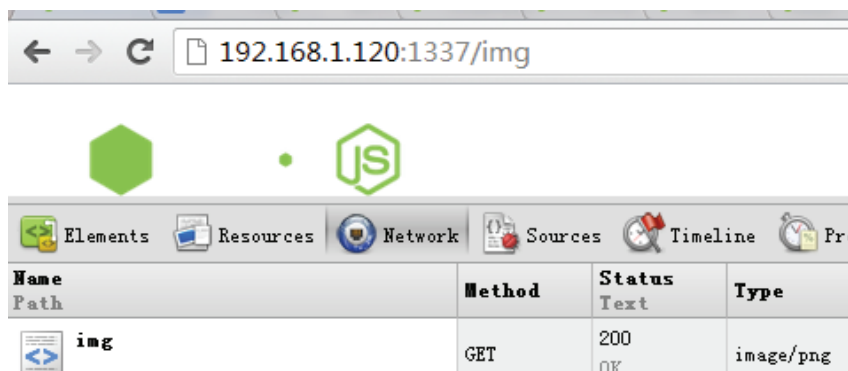


图 3-3 浏览 127.0.0.1:1337/img 返回 Web 页面图

如图 3-4 所示为服务器端输出的结果, 这里只截取了部分, 输出的结果包含了每次请求的 url, method 和请求 headers。可以看到每次 HTTP 请求都会自带一个/favicon.ico 请求, 这个是网页的 ico, 是大多数浏览器自我发出的请求, 如果需要去除这个请求, 可以设置返回一个有效的/favicon.ico 文件, 并且指定 expire 时间。

```

/accept-charset: 'GBK,utf-8;q=0.7,*;q=0.3' }
/
/index
GET
{ host: '192.168.1.120:1337',
  connection: 'keep-alive',
  'user-agent': 'Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.4 (KHTML, like Gecko) Chrome/22.0.1229.79 Safari/537.4',
  accept: 'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8',
  'accept-encoding': 'gzip,deflate,sdch',
  'accept-language': 'zh-CN,zh;q=0.8',
  'accept-charset': 'GBK,utf-8;q=0.7,*;q=0.3' }
/favicon.ico
GET
{ host: '192.168.1.120:1337',
  connection: 'keep-alive',
  'user-agent': 'Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.4 (KHTML, like Gecko) Chrome/22.0.1229.79 Safari/537.4',
  accept: 'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8',
  'accept-encoding': 'gzip,deflate,sdch',
  'accept-language': 'zh-CN,zh;q=0.8',
  'accept-charset': 'GBK,utf-8;q=0.7,*;q=0.3' }
/img
GET
{ host: '192.168.1.120:1337',
  connection: 'keep-alive',
  'user-agent': 'Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.4 (KHTML, like Gecko) Chrome/22.0.1229.79 Safari/537.4',
  accept: 'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8',
  'accept-encoding': 'gzip,deflate,sdch',
  'accept-language': 'zh-CN,zh;q=0.8',
  'accept-charset': 'GBK,utf-8;q=0.7,*;q=0.3' }

```

图 3-4 Node.js 服务器运行流水日志图

3.1.2 路由处理

了解 Node.js 服务器创建功能实现后，接下来介绍根据不同客户端的请求资源路径，来分配服务器处理逻辑。在 3.1.1 节中已经初步了解到可以使用 `switch` 来实现路由，当然这是一种路由处理办法，但是在请求资源非常复杂时，使用 `switch` 来判断处理就会显得很庞大，而且难以维护和扩展。本节将介绍两种路由处理方法，大家可根据自己的想法设计出更多的路由处理方法，另外在本章的习题 1 中还会提供第三种路由处理方法。

1. 特定规则请求路径

特定规则请求参数：可以根据用户请求的 `url`，依据特定的规则得到相应的执行函数，例如请求参数 `/index`，根据特定规则转化为 `resIndex(res, req)` 方法。

这里将 HTTP 的请求路径 `/index`，根据一定的规则得到其处理函数 `resIndex`，当我们需要新增处理逻辑时，例如 `/img`，则必须在服务器端新增处理函数 `resImg`。具体实现过程如下：

```

var param = pathname.substr(2),
    // 获取客户端请求的 url 路径，并获取其第一个参数，将其小写转为大写
    firstParam = pathname.substr(1,1).toUpperCase();
// 根据 pathname 获取其需要执行的函数名
var functionName = 'res' + firstParam + param;
response = res;
if(pathname == '/') {
    resDefault(res)
} else if (pathname == '/favicon.ico') {
    return;
}
else {
    eval(functionName + '()');
}

```

【代码说明】

- ❑ `firstParam = pathname.substr(1,1).toUpperCase()`：将参数的首字母改为大写，例如 `/index`，获取 `i`，并将其改为大写的 `I`；

- ❑ `functionName = 'res' + firstParam + param`: 获取需要执行的函数名, `res` 为前缀, `firstParam` 为首字母, 例如 `/index`, `firstParam` 为 `I`, `param` 为 `ndex`, 因此 `functionName` 为 `resIndex`;
- ❑ `pathname == '/'`: 无参数时返回 `default` 页面;
- ❑ `pathname == '/favicon.ico'`: 对于该请求返回空信息;
- ❑ `eval(functionName + '()')`: 使用 `eval` 执行字符串函数。

该方法有一个很明显的缺点, 就是 `res` 和 `req` 参数必须设置为全局变量, 否则函数中无法获取该 `res` 和 `req` 对象参数。

这里只是介绍了一种实现方式, 可以根据 `param` 参数来判断需要调用的模块和函数, 对于小项目而言可以实现这种路由处理。

2. 利用附带参数来实现路由处理

利用自带参数来实现路由处理: `url` 路径指定需要执行的模块, 通过在 `HTTP` 的 `url` 中携带一个 `c` 参数, 表示需要调用的模块中的方法名, 从而实现简单的路由处理。例如 `/image?c=img` 表示获取 `index` 模块中 `img` 方法, 同样 `/index?c=index` 表示获取 `index` 模块中的 `index` 方法。具体实现是创建 `client.js` 作为服务器模块, 代码如下:

```
/* 首先 require 加载两个模块 */
var http = require('http'),
    url = require('url'),
    querystring = require("querystring");
/**
 *
 * @desc 创建 web 服务器
 */
http.createServer(function(req, res) {
    /* 获取用户请求的 url 路径 */
    var pathname = url.parse(req.url).pathname;
    if (pathname == '/favicon.ico') { // 过滤浏览器默认请求/favicon.ico
        return;
    }
    /* 根据用户请求的 url 路径, 截取其中的 module 和 controller */
    var module = pathname.substr(1),
        str = url.parse(req.url).query,
        controller = querystring.parse(str).c,
        classObj = '';
    try { // 应用 try catch 来 require 一个模块, 并捕获其异常
        classObj = require('./' + module);
    }
    catch (err) { // 异常错误时, 打印错误信息
        console.log('chdir: ' + err);
    }
    if(classObj){ // require 成功时, 则应用 call 方法, 实现类中的方法调用执行
        classObj.init(res, req);
        classObj[controller].call();
    } else { // 调用不存在的模块时, 则默认返回 404 错误信息
        res.writeHead(404, { 'Content-Type': 'text/plain' });
        res.end('can not find source');
    }
}).listen(1337);
```

【代码说明】

- ❑ `var pathname = url.parse(req.url).pathname`: 获取客户端的 HTTP 请求路径, 也就是请求模块;
- ❑ `controller = querystring.parse(str).c`: 获取请求参数 `c`, 也就是请求模块的方法;
- ❑ `classObj = require('./' + module)`: 使用 `try catch` 获取一个 `require` 模块;
- ❑ `classObj.init(res, req)`: 初始化模块参数 `res` 和 `req`;
- ❑ `classObj[classObj.controller].call()`: 执行模块函数。

使用 `try catch require` 一个模块时, 避免 `require` 不存在文件代码异常中断服务器。本段代码涉及 HTTP 参数的获取方法, 关于 Node.js 中如何获取 `get` 和 `post` 参数将会在本章的 3.2 节介绍。该路由的实现是获取 url “/image?c=img” 的 `image` 和 `img`, 其中 `image` 为模块名, `img` 为模块函数名。根据二者可以使用 `image['img'].call()` 方法调用其模块中的函数名方法。在调用函数之前使用了 `init` 方法来设置模块中的 `res` 和 `req` 变量。注意, 这里不能直接使用 `image['img']`, 因为 `img` 是一个字符串, 因此需要使用 `image['img']` 这种方式来调用 `image` 中的 `img` 对象属性, 同时使用 `call` 方法执行该 `img` 对象的函数。

根据上述实现必须在本地路径创建 `image.js` 和 `index.js` 文件, 其中两个模块中都含有 `init` 方法来初始化模块中的 `res` 和 `req` 变量。`image.js` 中的 `img` 方法处理图片返回, `index.js` 中的 `index` 方法处理 `index.html` 页面展示, 代码如下:

```
/* index.js */
var res, req,
    fs = require('fs'),
    url = require('url');
/* 创建初始变量函数 */
exports.init = function(response, request){
    res = response;
    req = request;
}
/* 创建 index 首页函数 */
exports.index = function(){
    /* 获取当前 image 的路径 */
    var readPath = __dirname + '/' + url.parse('index.html').pathname;
    var indexPage = fs.readFileSync(readPath);
    res.writeHead(200, { 'Content-Type': 'text/html' });
    res.end(indexPage);
}
```

该模块的代码就不详细介绍了, 其作用和之前涉及的代码是一致的。`init` 方法的目的是初始该模块的 `res` 和 `req` 变量。代码如下:

```
/* image.js */
var res, req,
    fs = require('fs'),
    url = require('url');
exports.init = function(response, request){
    res = response;
    req = request;
}
/* 创建 HTTP 响应 img 图片函数 */
```

```
exports.img = function(){
  /* 获取当前 image 的路径 */
  var readPath = __dirname + '/' + url.parse('logo.png').pathname;
  /* 应用 fs 中的 readFileSync 同步 API 获取文件数据 */
  var indexPage = fs.readFileSync(readPath);
  res.writeHead(200, { 'Content-Type': 'image/png' });
  res.end(indexPage);
}
```

处理客户端对图片的请求逻辑, `init` 方法和 `index.js` 模块中的 `init` 方法的作用是相同的。两者路由实现的方法都大同小异, 一个是通过单个请求路径参数来实现路由功能, 另外一个是通过附加参数 `c` 和 `/index` 来获取需要调用的模块和模块中的函数。当然, 路由的处理实现可以有多种方法, 还可以使用 `image/img` 方法来调用。

3.1.3 GET 和 POST

Node.js 中 HTTP 客户端发送的 GET 请求参数数据都存储在 `request` 对象中的 `url` 属性中, 例如 `http://localhost:1337/test?name=danhuang`。其中 `url` 的请求路径名 `test`, 使用 GET 传输的 `name` 数据暴露在 `url` 上, 因此可以利用上一节 `client.js` 中提供的获取 `url` 的参数方法, Node.js 原生 `url` 模块¹中的 `parse` 方法获取 HTTP 的 GET 参数, 具体方法使用请参考官网 `url` 模块文档。

```
url.parse(req.url).pathname
```

该代码根据 `req` 对象获取 `url` 中的请求路径, 如 `http://localhost:1337/test?name=danhuang` 中的 `test` 为路径名, 这里 `req.url` 就是这里所举例的 `http://localhost:1337/test?name=danhuang`, 在实际情况中 `req.url` 为 `/test?name=danhuang`。

```
url.parse(req.url).query
```

该代码获取 `url` 中的非路径字符串, 例如 `http://localhost:1337/test?name=danhuang&book=Node.js`, 使用代码后将得到字符串 `"name=danhuang&book=Node.js"`, 获得字符串后还需要使用 Node.js 模块中的 `querystring` 对字符串 `"name=danhuang&book=Node.js"` 进行解析, 如下代码:

```
var param = querystring.parse('name=danhuang&book=Node.js')
```

解析返回 `{ name: 'danhuang', book: 'Node.js' }` 的 json 对象, 得到该 json 对象后, 获取 `name` 和 `book` 的方法可直接使用 `param.name` 或者 `param['name']`。

3.1.4 GET 方法实例

使用 `http` 模块创建一个服务器, 该服务器接收任意的 `url` 请求资源, 使用 GET 方法传递参数, 服务器接收客户端请求 `url`, 输出每次请求的路径名和请求参数的 json 对象。

分析解析 GET 请求参数需要的模块, 分别为 `http` 模块创建服务器、`url` 模块解析 `url`

¹ 参考网站 http://nodejs.org/api/all.html#all_url_parse_urlstr_parsequerystring_slashesdenotehost。

路径和 GET 字符串、`querystring` 转化 GET 数据字符串为 json 对象。代码如下：

```
/* get_method.js */
var http      = require('http'),
    url       = require('url'),
    querystring = require('querystring');
```

`url.parse(req.url).pathname` 获取请求路径，`url.parse(req.url).query` 获取请求 GET 数据字符串，`querystring.parse(str)` 解析 GET 数据字符串为 json 对象。代码如下：

```
var pathname = url.parse(req.url).pathname,
    paramStr = url.parse(req.url).query,
    param = querystring.parse(paramStr);
```

输出解析后的 `pathname`、`paramStr` 和 `param` 数据。代码如下：

```
console.log(pathname);
console.log(paramStr ? paramStr : 'no params');
console.log(param);
```

【代码说明】

□ `paramStr ? paramStr : 'no params'`：避免输出 `undefined` 字符。

这里我们对浏览器自带的 `/favicon.ico` 请求进行过滤，避免不必要的影响。

```
if('/favicon.ico' == pathname){
    return;
}
```

最后看一下整个 `get_method.js` 的代码实现。代码如下：

```
var http = require('http'),
    url = require('url'),
    querystring = require('querystring');
/* 创建 HTTP 服务器 */
http.createServer(function(req, res) {
    var pathname = url.parse(req.url).pathname,    // 获取 url 请求路径
        paramStr = url.parse(req.url).query,      // 获取 url 请求参数
        param = querystring.parse(paramStr);      // 将 url 请求参数转化为 json 对象
    if('/favicon.ico' == pathname){ // 去除浏览器自带请求 favicon.ico
        return;
    }
    // 打印参数信息
    console.log(pathname);
    console.log(paramStr ? paramStr : 'no params');
    console.log(param);
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end('success');
}).listen(1337);
console.log('Server running at http://127.0.0.1:1337/');
```

运行 Node.js 脚本 `get_method.js`，如下所述。

(1) 输入 `http://127.0.0.1:1337/`，输出如图 3-5 所示，请求路径为空，GET 传递参数为空，解析字符串 json 也为空。

(2) 输入 `http://127.0.0.1:1337/index`，输出如图 3-6 所示，请求路径为 `/index`，GET 传递参数为空，解析字符串 json 为空。

```
/
no params
{}
```

图 3-5 客户端浏览 `http://127.0.0.1:1337/` 后 Node.js 服务端流水日志

```
/index
no params
{}
```

图 3-6 客户端浏览 `http://127.0.0.1:1337/index` 后 Node.js 服务端流水日志

(3) 输入 `http://127.0.0.1:1337/index?name=danhuang&book=Node.js`, 输出如图 3-7 所示, 请求路径为 `/index`, GET 传递参数字符串为 `name=danhuang&book=Node.js`, 解析后的 json 对象为 `{ name: 'danhuang', book: 'Node.js' }`。

```
/index
name=danhuang&book=Node.js
{ name: 'danhuang', book: 'Node.js' }
```

图 3-7 客户端请求后 Node.js 服务端流水日志

(4) 输入 `http://127.0.0.1:1337/image/img/test`, 输出如图 3-8 所示, 请求路径为 `/image/img/test`, GET 传递参数字符串为空, 解析 json 对象为空。

```
/image/img/test
no params
{}
```

图 3-8 客户端请求后 Node.js 服务端流水日志

通过这些 url 的请求测试, 一方面加深对 Node.js 中路由解析的方式理解, 另一方面了解和学习 Node.js 中获取 HTTP 请求中的 GET 参数的方法。

相比较 GET 请求, POST 请求一般比较复杂, Node.js 为了使整个过程非阻塞, 会将 POST 数据拆分成很多小的数据块, 然后通过触发特定的事件, 将这些小数据块有序传递给回调函数。这部分涉及 `request` 对象中的 `addListener` 方法, 该方法有两个事件参数 `data` 和 `end`, `data` 表示数据传输开始, `end` 表示数据传输结束。代码如下:

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

http.createServer(function (req, res) {
  var postData = '';

  // 设置接收数据编码格式为 UTF-8
  req.setEncoding('utf8');

  // 接收数据块并将其赋值给 postData
  req.addListener('data', function(postDataChunk) {
    postData += postDataChunk;
  });

  req.addListener('end', function() {
    // 数据接收完毕, 执行回调函数
    var param = querystring.parse(postData);
  });

  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}).listen(1337, '127.0.0.1');
```

【代码说明】

- ❑ req.setEncoding('utf8'): request 对象中的 setEncoding 方法设定接收数据编码格式, 以免在 HTTP 响应返回数据时造成的页面编码乱码异常。
- ❑ req.addListener('data',function(postDataChunk){}): 得到 POST 小块数据后添加到 postData 中。
- ❑ req.addListener('end',function(){}): 所有数据接收完毕后, 执行 POST 参数解析。

GET 和 POST 方法获取数据的形式不同, 但数据的解析都是通过 querystring.parse(postData)来将 GET 字符和 POST 转化为 json 对象。

3.1.5 POST 方法实例

创建 HTTP 服务器, 读取一个 index.html 页面, index.html 页面有一个 form 表单, 该表单 POST 提交数据到 http://127.0.0.1:1337/add, 服务器端获取 POST 数据后打印显示当前的请求路径, POST 字符串, POST 的 json 参数对象。

分析本实例需要的模块有 http 模块创建 HTTP 服务器、fs 模块读取 index.html 页面信息、url 模块解析请求资源路径、querystring 解析 POST 数据。代码如下:

```
/* post_method.js */
var http = require('http'),
    fs = require('fs'),
    url = require('url'),
    querystring = require('querystring');
```

利用 3.1.2 节中介绍的“特定规则请求路径”的路由处理方法, 做简单的路由处理。代码如下:

```
var pathname = url.parse(req.url).pathname;
switch(pathname){
    case '/add' : resAdd(res, req);
        break;
    default      : resDefault(res);
        break;
}
```

【代码说明】

- ❑ resDefault: 显示 index.html 页面函数逻辑。
- ❑ resAdd: 打印 POST 请求数据函数逻辑。

resDefault 函数实现主要是返回客户端一个 index.html 页面, 这部分知识点在前面已有所介绍, 这里只展示其代码实现。代码如下:

```
function resDefault(res){
    /* 获取当前 index.html 的路径 */
    var readPath = __dirname + '/' +url.parse('index.html').pathname;
    var indexPage = fs.readFileSync(readPath);
    res.writeHead(200, { 'Content-Type': 'text/html' });
    res.end(indexPage);
}
```

其中, index.html 实现表单数据提交到 http://127.0.0.1:1337/add 下。代码如下:


```

<html>
  <head>
    <title>Test Post</title>
  </head>
  <body>
    <div>
      <form action='add' method='POST'>
        <label>name: <input type='text' name='name'></label>
        <label>book: <input type='text' name='book'></label>
        <input type='submit' />
      </form>
    </div>
  </body>
</html>

```

resAdd 函数利用 req.addListener 的方法获取 POST 数据，当 POST 数据接收完毕后打印并解析 POST 数据。代码如下：

```

/**
 *
 * @desc      获取 HTTP 请求时 POST 的数据
 * @parameters res HTTP 响应对象
 * @parameters req HTTP 请求对象
 */
function resAdd(res, req){
  var postData = '';
  // 设置接收数据编码格式为 UTF-8
  req.setEncoding('utf8');
  // 接收数据块并将其赋值给 postData
  req.addListener('data', function(postDataChunk) {
    postData += postDataChunk;
  });
  req.addListener('end', function() {
    // 数据接收完毕，执行回调函数
    var param = querystring.parse(postData);
    console.log(postData);
    console.log(param);
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end('success');
  });
}

```

【代码说明】

- var param = querystring.parse(postData): 利用 querystring 模块解析 POST 数据。
- res.end('success'): 执行成功后，返回文本信息 success 到客户端。

运行 post_method.js:

```
node post_method.js
```

打开浏览器输入 <http://127.0.0.1:1337>，可以看到如图 3-9 所示的 index.html 页面信息。

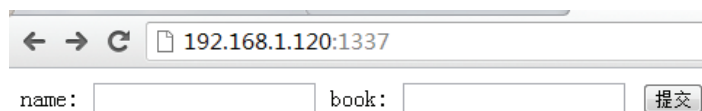


图 3-9 浏览 <http://127.0.0.1:1337> 返回的 Web 页面图

在 name 和 book 中输入相应的字符数据，例如 danhuang 和 Node.js，单击“提交”按钮，页面将会立即得到并显示 success 信息，而服务器端结果如图 3-10 所示。

```
name=danhuang&book=Node.js
{ name: 'danhuang', book: 'Node.js' }
```

图 3-10 服务器端结果

第一行输出为 POST 数据的字符串，第二行输出为 POST 数据字符串通过 querystring 解析后的 json 数据对象。

上面介绍了 HTTP 中 POST 和 GET 参数的获取方式，现在我们将两个方法作为一个公用的模块，可以在很大程度上减少工作量，毕竟获取 GET 和 POST 参数的方法都不是一步完成的，都必须进行多步操作，接下来我们来实践开发一个 HTTP 参数获取模块——httpParams，主要是利用其中的两个方法 GET 和 POST 来直接得到客户端传递的数据。在该模块起始部分，首先要获取该模块的一些必要的 Node.js 原生模块，分别是 url 和 querystring。其次需要在代码中应用 init 方法来初始化该模块对象中的 res 和 req 对象。代码如下：

```
var _res, _req,
    url = require('url'),
    querystring = require('querystring');
/**
 * 初始化 res 和 req 参数
 */
exports.init = function(req, res){
    _res = res;
    _req = req;
}
```

【代码说明】

❑ exports.init = function(req, res){}：初始化 http 中的 res 和 req 参数。querystring 对象解析 GET 参数，并返回。代码如下：

```
/**
 * 获取 GET 参数方法
 */
exports.GET = function(key){
    var paramStr = url.parse(_req.url).query,
        param = querystring.parse(paramStr);
    return param[key] ? param[key] : '';
}
```

【代码说明】

❑ return param[key] ? param[key] : ''：判断是否存在该 key，不存在，则返回默认值空。

❑ param = querystring.parse(paramStr)：获取传递参数的 json 数据方法。

res.addListener 和 querystring 获取 POST 值，注意，这里是一个异步调用过程，因此需要使用回调函数返回执行结果，如下代码中的 callback 参数：

```
/**
 * 获取 POST 参数方法
```


```

*/
exports.POST = function(key, callback){
  var postData = '';
  _req.addListener('data', function(postDataChunk) {
    postData += postDataChunk;
  });
  _req.addListener('end', function() {
    // 数据接收完毕, 执行回调函数
    var param = querystring.parse(postData);
    var value = param[key] ? param[key] : '';
    callback(value);
  });
}

```

【代码说明】

- ❑ `exports.POST = function(key, callback)`: `callback` 函数是为了传递异步函数 `addListener` 的返回结果。

 **注意**: 这里使用的是一个 `callback` 回调函数作为参数, 来解决异步调用中返回结果的传递。这样就简单地实现了一个 HTTP 中如何获取 GET 和 POST 传递的数据的模块。当然读者可以使该模块更完善, 例如将 POST 使用同步返回结果等, 然后利用之前我们介绍的“NPM 的模块发布”方法, 发布到 Node.js 的 NPM 模块中, 提供给其他开发者学习和使用。

3.1.6 HTTP 和 HTTPS 模块介绍

这部分模块的介绍就不会详细地描述其中的一些 API 的调用方法, 主要是介绍这两个模块之间的区别和联系, 以及两个模块的适用场景。

HTTP¹是一种详细规定了浏览器和万维网服务器之间互相通信的规则, 通过因特网传送万维网文档的数据传送协议。而 HTTPS²是以安全为目标的 HTTP 通道, 简单地说, 它是 HTTP 的安全版。

两者创建服务器的方式接口都是一致的, 都是使用各自模块中的 `createServer` 方法, 但 HTTPS 中 `createServer` 会附加一个参数 `opts`, 其中保存有 `key` 和 `cert` 的信息。这两个文件可以去官网的文档³中下载作为测试使用。

本节主要介绍 HTTPS 的服务器创建, 下面创建一个 HTTPS 服务器, 输出一个 `hello world` 信息。

```

/* hello_world.js */
var https = require('https');
var fs = require('fs');
/* 获取私钥 */
var options = {
  key: fs.readFileSync('keys/agent2-key.pem'),

```

1 参考网站 <http://nodejs.org/api/http.html>。

2 参考网站 <http://nodejs.org/api/https.html>。

3 参考网站 <https://github.com/horaci/node-mitm-proxy/tree/master/certs>。

```
cert: fs.readFileSync('keys/agent2-cert.pem')
};

https.createServer(options, function (req, res) {
  res.writeHead(200);
  res.end("hello world\n");
}).listen(1337);
```

【代码说明】

- ❑ `https = require('https')、require('fs')`: 获取 `fs` 和 `https` 模块。
- ❑ `key: fs.readFileSync('keys/agent2-key.pem')`: 读取 `key` 值。
- ❑ `cert: fs.readFileSync('keys/agent2-cert.pem')`: 读取 `cert` 值。
- ❑ `https.createServer(options,function(){}):` 创建 HTTPS 服务器，并监听 1337 端口。

🔔注意：这里读取的是已经通过 openssl 生成的 key 和 cert 证书信息，具体实现方法可以参阅《openssl 生成 https 证书》¹这篇博文。

运行 `hello_world.js` 脚本文件，打开浏览器输入 `https://127.0.0.1:1337`，如图 3-11 所示。

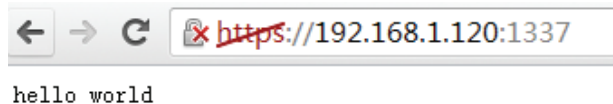


图 3-11 浏览 `https://127.0.0.1:1337` 返回 web 页面图

图 3-11 显示了一个返回 HTTPS 后的返回信息，大家可以看到 HTTPS 上划了斜线，表示该网站未经过身份认证，服务器证书与 IP 不符合。

生成的 key 和 cert 也可以使用 openssl 来合并成为 `server.pfx` 证书，并将 `options` 参数改为 `pfx`，如下：

```
var options = {
  pfx: fs.readFileSync('server.pfx')
};
```

这里可以不使用 `key` 和 `cert` 参量。本节只简单地介绍 HTTPS 的创建和 HTTP 服务器的创建之间的联系和区别，对于深入的 HTTP 模块和 HTTPS 模块的 APIs 的使用大家可以参考官网文档，里面有详细的例子以及说明。

3.2 Node.js 静态资源管理

3.1 节介绍了 HTTP 服务器如何在客户端显示一个 `html` 和 `image` 静态资源的方法，解决的方式是根据请求的资源路径名，使用特定的方法处理不同静态资源的返回，但对于前端种种类型的静态资源，服务器端无法为每个静态资源类型实现一个处理逻辑，因此在

¹ 参见网站 <http://notech.blog.sohu.com/108540014.html>。

Web 应用开发中我们需要自己设计一个静态资源管理模块。看到这么一大段介绍相信大部分同学还是会对“为什么要静态资源的管理”这个问题还存在很多疑惑，本节将为大家解开这个迷雾，并教大家如何设计一个 Node.js 静态资源管理。

本节将介绍静态资源管理存在的必要性，然后介绍静态资源库的实现，最后会介绍如何设计一个 Node.js 静态资源管理库。本节重点在介绍如何设计一个 Node.js 静态资源管理库。学完本节读者要明白为什么要设计静态资源库，并掌握 Node.js 静态资源库的创建方法。

3.2.1 为什么需要静态资源管理

请大家看一个例子，例子是在之前的 HTTP 服务器创建并显示一个 index.html 页面的基础上，在 index.html 包含一个 style.css，来美化 Web 页面。代码如下：

```
<html>
  <head>
    <title>Test Http</title>
    <link rel="stylesheet" href="style.css">
  </head>
  <body>
    <div>Oh, good~!</div>
  </body>
</html>
```

【代码说明】

❑ <link rel="stylesheet" href="style.css">：获取本路径下的 style.css 文件。

这里基本和原来 index.html 代码是一致的，添加了一个 css 的 link 文件，HTTP 服务器代码也和创建一个 HTTP 服务器并返回一个 html 文件一致，代码如下：

```
/* http.js */
var http = require('http'),
    fs = require('fs'),
    url = require('url');
http.createServer(function(req, res) {
  /* 获取当前 index.html 的路径 */
  var readPath = __dirname + '/' + url.parse('index.html').pathname;
  var indexPage = fs.readFileSync(readPath);
  res.writeHead(200, { 'Content-Type': 'text/html' });
  res.end(indexPage);
}).listen(1337);
console.log('Server running at http://localhost:1337/');
```

最后添加一个本路径下的 style.css 文件，其作用是修改 div 中的字符串颜色。

```
/* style.css */
div{
  color: red;
}
```

按照我们的想法，index.html 中的 div 标签下的字符串将会显示红色，现在我们执行 http.js 文件，并在浏览器下输入 http://127.0.0.1:1337，如图 3-12 所示。

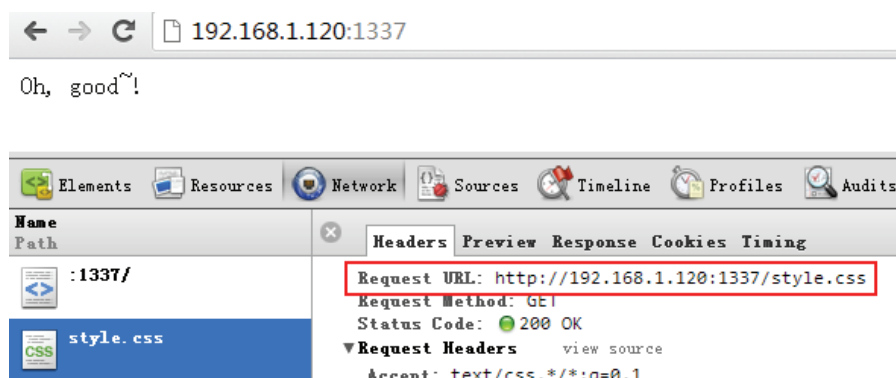


图 3-12 浏览 http://127.0.0.1:1337 返回结果图

从执行结果中可以看到，并没有展示“Oh, good~!”为红色。原因是 index.html 里面 link 一个 css 文件时也会产生一个 HTTP 请求获取 css 文件资源，但是其发出的 http://127.0.0.1:1337/style.css 请求 url，对于这个资源并没有返回 style.css 文件信息，作为服务器端并不知道客户端请求的资源是 style.css 文件，服务器端没有路由逻辑去处理 /style.css 的请求，因此不会正常的响应返回一个 style.css 数据。

同样，如果 index.html 中包含一个 JavaScript 文件或图片文件等，都会出现类似的情况。在项目开发中，如何处理这些前端的静态资源文件是需要每个 Node.js 开发者去解决的问题。

3.2.2 Node.js 实现简单静态资源管理

基于上面存在的问题，使用一个简单的方法来处理几类静态文件，判断请求静态资源的后缀名，根据后缀名返回不同的 MIME 类型。将 3.2.1 节中的 index.html 修改如下：

```
<html>
  <head>
    <title>Test Http</title>
    <link rel="stylesheet" href="style.css">
  </head>
  <body>
    <div>Oh, good~!</div>
    <div><img src='logo.png' /></div>
  </body>
</html>
```

【代码说明】

- ❑ <link rel="stylesheet" href="style.css">: link style.css。
- ❑ : 显示一个 logo.png 图片。

这部分比之前的 index.html 更进一步地获取一个静态 png 图片。接下来学习如何利用 Node.js 实现简单的静态资源管理。

创建 http.js，首先分析需要的 Node.js 模块：HTTP 模块创建服务器、fs 模块处理文件的读写、url 模块处理 url 请求路径。

```

/* http.js */
var http = require('http'),
    fs    = require('fs'),
    url   = require('url'),
    BASE_DIR = __dirname;

```

【代码说明】

❑ `BASE_DIR = __dirname`: 获取当前脚本路径, `__dirname` 是 Node.js 一个全局常量。

上面的代码中使用 `BASE_DIR` 来定义常量数据, 这是一个代码规范, 关于代码的规范将在后续章节中详细介绍。接下来创建一个 HTTP 服务器。代码如下:

```

http.createServer(function(req, res) {
  /* 获取当前 index.html 的路径 */
  var pathname = url.parse(req.url).pathname;
  var realPath = __dirname + '/static' + pathname;
  if (pathname == '/favicon.ico') {
    return;
  } else if (pathname == '/index' || pathname == '/') {
    goIndex(res)
  } else {
    dealWithStatic(pathname, realPath, res);
  }
}).listen(1337);
console.log('Server running at http://localhost:1337/');

```

【代码说明】

❑ `pathname == '/favicon.ico'`: 过滤 `favicon.ico` 请求。

❑ `var realPath = __dirname + '/static' + pathname`: 获取静态资源文件存储路径。

❑ `pathname == '/index' || pathname == '/'`: 请求路径为 `index` 和空时, 跳转到 `index.html` 页面。

❑ `dealWithStatic(pathname, realPath, res)`: 处理静态资源文件逻辑, 主要是根据不同静态资源路径, 返回相应的静态资源文件信息。

`dealWithStatic` 这个方法传递了 3 个参数, 分别代表请求资源路径名、静态资源实际存储路径和相应 HTTP 响应对象 `res`。`dealWithStatic` 的作用就是根据不同的请求资源后缀名, 返回相应的 MMIE 类型和数据到客户端。以下是 `dealWithStatic` 的处理逻辑代码:

```

function dealWithStatic(pathname, realPath, res){
  fs.exists(realPath, function (exists) { // 判断文件是否存在
    if (!exists) {
      res.writeHead(404, {'Content-Type': 'text/plain'});
      res.write("This request URL " + pathname + " was not found on this server.");
      res.end();
    } else {
      var pointPostion = pathname.lastIndexOf('.'),
          mmieString    = pathname.substring(pointPostion+1),
                               // 获取文件后缀名
          mmieType;
      switch (mmieString){
        // 根据文件后缀名, 设置 HTTP 响应的 Content-Type 类型
        case 'css' : mmieType = "text/css";
                     break;
        case 'png' : mmieType = "image/png";

```

```

        break;
        default:
            mmieType = "text/plain";
    }
    fs.readFile(realPath, "binary", function(err, file) {
        if (err) {
            res.writeHead(500, {'Content-Type': 'text/plain'});
            res.end(err);
        } else {
            res.writeHead(200, {'Content-Type': mmieType});
            res.write(file, "binary");
            res.end();
        }
    });
}
});
}
}

```

【代码说明】

- ❑ `fs.exists(realPath, function (exists) {})`: 应用 `fs` 模块中的 `exists` 方法验证该静态文件是否存在, 异步返回信息到 `exists` 变量中;
- ❑ `if(!exists) {}`: 不存在该 `file` 时, 返回 404 not find;
- ❑ `pointPostion = pathname.lastIndexOf('.')`: 查询请求路径中的点分隔符位置;
- ❑ `mmieString = pathname.substring(pointPostion+1)`: 截取请求路径点分隔符请求文件资源的后缀名;
- ❑ `switch (mmieType){}`: 判断请求静态资源的后缀名类型;
- ❑ `case 'css': mmieType = "text/css"`: 文件后缀名为 `css` 时, 设置其返回的 MMIE 类型为 `text/css`;
- ❑ `case 'png': mmieType = "image/png"`: 文件后缀名为 `png` 时, 设置其返回的 MMIE 类型为 `image/png`;
- ❑ `fs.readFile(realPath, "binary", function(err, file)`: 读取文件信息, 并转化为二进制数据;
- ❑ `res.writeHead(200, {'Content-Type': mmieType})`: 返回相应的 MMIE 类型数据。

这里只针对两种类型的静态资源文件: `css` 和 `png` 图片, 对于其他的资源暂时不考虑。代码编写完成后, 运行 `http.js`, 打开浏览器输入 `http://127.0.0.1:1337/`, 看到如图 3-13 所示的效果。

从图 3-13 中可以看到此次 HTTP 请求, 返回的 `index.html` 页面不仅修改了 `div` 的样式, 也从服务器端拉到了 `logo.png` 图片信息。从图中可以看到, 对于一个 `index.html` 页面的请求, 事实上客户端产生了 3 个 HTTP 请求, 如图 3-14 所示。



图 3-13 加载 CSS 后 Web 页面图

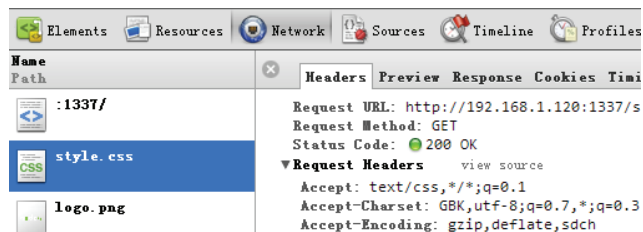


图 3-14 查看 style.css 文件的请求返回 headers 信息图

3 次 HTTP 请求分别是 index.html 页面信息、style.css 文件和 logo.png 图片。到此为止，上面代码实现了一个简单的静态资源管理功能。这里只是提供了一个静态资源的解决方法，对于一个完善的静态资源管理来说还远远不够。以上代码仅仅只是针对两种 MMIE 类型：css 和 png，客户端的 MMIE 请求类型是多种多样。

3.2.3 静态资源库设计

上一节中已经提及到如何实现一个简单的静态资源管理。本节将完善该 static 模块，来实现所有的静态资源类型的管理功能。

HTTP 请求中大概有 409 种 MMIE 类型，在设计时将这 409 种 MMIE 类型作为一个 json 配置信息存储在 mmie_type.json 文件中，以便维护和管理。409 种 MMIE 类型这里就不一一介绍了，具体的配置可参考源码中的 mmie_type.json 文件信息。

接下来使用如下代码方法读取 json 配置文件信息，并转化为 json 对象：

```
//获取 MMIE 配置信息，读取配置文件
function getUrlConf(){
  var routerMsg = {};
  try{
    var str = fs.readFileSync(CONF + 'mmie_type.json','utf8');
    routerMsg = JSON.parse(str);
  }catch(e){
    sys.debug("JSON parse fails")
  }
  return routerMsg;
}
```

【代码说明】

- ❑ var str = fs.readFileSync(CONF + 'mmie_type.json','utf8')：utf8 编码读取 json 配置信息；
- ❑ sys.debug("JSON parse fails")：解析错误时，显示 debug 日志，sys 为 require util 模块返回的对象。

解析 json 配置文件内容时只需要 fs 模块即可，util (sys) 模块主要用来 catch 解析 json 的错误返回信息。其中应用到的方法是 JavaScript 的内置方法 JSON.parse(str)，将 json 字符转化为 json 对象。

将 HTTP 逻辑处理请求和静态文件资源请求进行切分，对于不含后缀名的文件，默认为 HTTP 的逻辑应用请求，对于含后缀名的 HTTP 请求默认为静态资源拉取。应用 path 模块 extname API，获取 pathname 中的文件后缀名，代码如下：

```
var extname = path.extname(pathname);
extname = extname ? ext.slice(1) : '';
```

【代码说明】

- ❑ path.extname(pathname)：应用 path 模块获取请求 pathname 的后缀名字符串，例如 pathname='index.html'，执行后返回的是'.html'。
- ❑ ext.slice(1)：使用 JavaScript 的字符串截取函数 slice，截取得到'.html'中的'html'字符串。

得到文件后缀名后，就可根据其后缀名获取其 MMIE 类型，从而返回相应的数据类型到客户端，下面我们看看 `static_module.js` 的代码实现。

设定常量变量，这里对于全局变量建议使用命名空间。代码如下：

```
/**
 *
 * 定义全局常用变量
 */
var BASE_DIR = __dirname,
    CONF      = BASE_DIR + '/conf/',
    STATIC    = BASE_DIR + '/static',
    mmieConf;
```

【代码说明】

- ❑ BASE_DIR：本地文件夹绝对路径。
- ❑ CONF：配置文件绝对路径。
- ❑ STATIC：静态文件存储路径。
- ❑ mmieConf：MMIE 类型的 json 存储配置内容。

在这段代码中，CONF 末尾加上了一个斜杠，而 STATIC 却没有，原因在于静态资源请求的 `pathname` 会自带一个斜杠，例如：'/logo.jpg'，一般习惯是在路径常量中末尾添加一个斜杠。

分析本模块实现需要的模块。代码如下：

```
/**
 *
 * require 本模块需要的 Node.js 模块
 */
var sys = require('util'),
    http = require('http'),
    fs   = require('fs'),
    url  = require('url'),
    path = require('path');
mmieConf = getUrlConf();
```

【代码说明】

- ❑ `mmieConf = getMmieConf()`：获取 MMIE 的配置文件内容。

`sys` 是一个工具类模块，主要是对一些异常处理打印 debug 信息。`http`、`fs` 和 `url` 模块前面已经介绍，`path` 模块是获取请求 `pathname` 的扩展名。

创建 `exports` 方法 `getStaticFile`。代码如下：

```
/**
 *
 * 响应静态资源请求
 * @param string pathname
 * @param object res
 * @return null
 */
exports.getStaticFile = function(pathname, res){
}
```

`getStaticFile` 的两个参数 `pathname` 和 `res`，分别表示客户端请求资源路径和 `response` 对象。

```
var extname = path.extname(pathname);
extname = extname ? extname.slice(1) : '';
var realPath = STATIC + pathname;
var mmieType = mmieConf[extname] ? mmieConf[extname] : 'text/plain';
```

【代码说明】

- ❑ `extname = extname ? extname.slice(1) :` 获取 `pathname` 的扩展名, 例如 `'style.css'` 字符中的 `'css'`。
- ❑ `STATIC + pathname`: 设置静态文件路径, 例如 `'/data/static/logo.png'`。
- ❑ `mmieConf[extname] ? mmieConf[extname] : 'text/plain'`: 设置请求静态文件的 MMIE 类型, 配置中不存在该类型时, 默认为 `'text/plain'` 类型。

最后看一下核心函数 `getStaticFile` 实现响应不同种 MMIE 类型的逻辑处理。代码如下:

```
/* 判断文件是否存在 */
fs.exists(realPath, function (exists) {
  if (!exists) { // 判断文件是否存在
    res.writeHead(404, {'Content-Type': 'text/plain'});
    res.write("This request URL " + pathname + " was not found on this server.");
    res.end();
  } else {
    fs.readFile(realPath, "binary", function(err, file) {
      if (err) {
        res.writeHead(500, {'Content-Type': 'text/plain'});
        res.end(err);
      } else {
        res.writeHead(200, {'Content-Type': mmieType});
        res.write(file, "binary");
        res.end();
      }
    });
  }
});
```

这部分代码在之前已经有所介绍, 主要是应用 `fs` 的两个 API 函数 `exists` 和 `readFile`, 当客户端请求服务器端不同的静态文件资源时, 服务器端会根据客户端请求的 `mmieType` 来响应不同的 `Content-Type` 类型。接下来创建一个测试文件 `client.js`。代码如下:

```
var staticModule = require('./static_module');
http.createServer(function(req, res) {
  /* 获取当前 index.html 的路径 */
  var pathname = url.parse(req.url).pathname;
  if (pathname == '/favicon.ico') {
    return;
  } else if (pathname == '/index' || pathname == '/') {
    goIndex(res)
  } else {
    staticModule.getStaticFile(pathname, res);
  }
}).listen(1337);
```

【代码说明】

- ❑ `staticModule = require('./static_module')`: 获取静态资源管理模块;

❏ `staticModule.getStaticFile(pathname, res)`: 如果是静态资源文件的请求, 调用 `getStaticFile` 来统一处理项目中的静态资源文件。

本段代码省略了部分原生模块的引入和 `goIndex()` 方法的实现。

🔔注意: 本例子的代码可参考本书源码第3章中的 `static_module` 文件夹。

运行脚本 `client.js`, 打开浏览器输入 `http://127.0.0.1:1337`, 页面返回如图 3-15 所示的结果, 和 3.2.2 节“简单实现静态资源管理”返回结果一致。

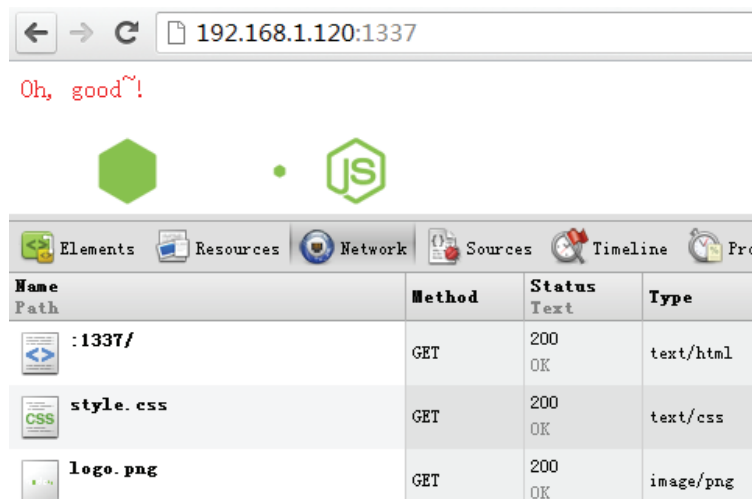


图 3-15 浏览 `http://127.0.0.1:1337` 加载静态资源后返回结果图

以上过程就实现了一个静态资源管理库, 只需调用 `static_module` 模块中的 `getStaticFile` 函数就可以管理本项目中所有静态资源的 HTTP 请求响应。

在上面的例子中大家是否察觉到一个问题, 每刷新一次 `http://127.0.0.1:1337` 页面就会产生 3 个 HTTP 请求, 而这 3 个 HTTP 请求的 `status` 都为 200。也就是说每一次都会使用 `fs` 拉取服务器静态文件, 当服务器的请求量一上涨, 硬盘 IO 会承受很大压力, 因此静态资源文件的缓存机制是非常有必要的, 接下来我们将介绍静态文件缓存控制方法。

3.2.4 静态文件的缓存控制

本节参考 CNode 社区 Jackson 的《用 NodeJS 打造你的静态文件服务器》¹ 文章。浏览器缓存中存有文件副本的时候, 不能确定该文件是否有效时, 会生成一个条件 `get` 请求, 在该请求的 `header` 中包含 `If-Modified-Since` 参数。如果服务器端文件在这个时间后发生过更改, 就发送整个文件给客户端。如果没有修改, 则返回 304 状态码, 并不发送整个文件给客户端。

如果确定该副本有效时, 客户端不会发送 `GET` 请求。判断有效的最主要的方法是, 服务端响应的时候带上 `expires` 的头。浏览器会判断 `expires` 头, 直到指定的日期过期, 才会发起新的请求。

1 参见网站 <http://cnodejs.org/topic/4f16442ccae1f4aa27001071>。

指定静态文件后缀名类型，在响应时 header 中添加 expires 和 Cache-Control: max-age。超时日期设置为 1 年。在静态文件服务器中，我们为每个静态文件请求响应返回 Last-Modified header。为带 If-Modified-Since 的请求 header，做日期检查，如果没有修改，就返回 304，若修改，则返回相应文件。

现在我们将 3.2.3 节的所有静态文件类型加上一个缓存机制。在设置 header 之前首先判断请求的资源文件是否需要缓存，代码如下：

```
var CACHE_TIME = 60*60*24*365;
if ( mmieConf[extname]) {
    var expires = new Date();
    expires.setTime(expires.getTime() + CACHE_TIME* 1000);
    response.setHeader("Expires", expires.toUTCString());
    response.setHeader("Cache-Control", "max-age=" + CACHE_TIME);
}
```

我们同时也要检测浏览器是否发送了 If-Modified-Since 请求头。如果客户端发送的最后修改时间与服务器文件的修改时间相同的话，HTTP 响应 304 状态码。

```
if (res.headers[ifModifiedSince] && lastModified == res.headers
[ifModifiedSince]) {
    response.writeHead(304, "Not Modified");
    response.end();
}
```

判断浏览器文件是否存在缓存，以及文件缓存是否过期，如果存在缓存信息并且未过期，则服务器响应 HTTP 的 304 状态码。不需要再次 IO 读取磁盘中的静态资源文件数据，可直接从浏览器缓存中获取。

在 2.3.3 节中 static_module.js 文件模块中的 exports.getStaticFile 方法基础上添加 req 参数（request 对象），通过 req 参数对象获取客户端请求的 header 信息，应用 req.headers['if-modified-since'] 获取客户端请求的 If-Modified-Since 最后更改时间，将浏览器客户端的 If-Modified-Since 时间和本地 lastModified 参数进行对比，如果不相同就重新拉取静态资源，否则返回 304 not modified。代码如下：

```
/**
 *
 * 响应静态资源请求
 * @param string pathname
 * @param object res
 * @return null
 */
exports.getStaticFile = function(pathname, res, req){
    var extname = path.extname(pathname);
    extname = extname ? extname.slice(1) : '';
    var realPath = STATIC + pathname;
    var mmieType = mmieConf[extname] ? mmieConf[extname] : 'text/plain';
    fs.exists(realPath, function (exists) {
        if (!exists) {
            res.writeHead(404, {'Content-Type': 'text/plain'});
            res.write("This request URL " + pathname + " was not found on this server.");
            res.end();
        } else {
            var fileInfo = fs.statSync(realPath);
            var lastModified = fileInfo.mtime.toUTCString();
```

```

    /* 设置缓存 */
    if ( mmieConf[extname]) {
        var date = new Date();
        date.setTime(date.getTime() + CACHE_TIME * 1000);
        res.setHeader("Expires", date.toUTCString());
        res.setHeader("Cache-Control", "max-age=" + CACHE_TIME);
    }
    if (req.headers['if-modified-since'] && lastModified == req.
headers['if-modified-since']) {
        res.writeHead(304, "Not Modified");
        res.end();
    } else {

        fs.readFile(realPath, "binary", function(err, file) {
            if (err) {
                res.writeHead(500, {'Content-Type': 'text/
plain'});
                res.end(err);
            } else {
                res.setHeader("Last-Modified", lastModified);
                res.writeHead(200, {'Content-Type': mmieType});
                res.write(file, "binary");
                res.end();
            }
        });
    }
}
});
}
});
}

```

【代码说明】

- ❑ `var fileInfo = fs.statSync(realPath)`: 同步执行获取静态文件 `realPath` 信息, 返回的格式如下所示。

```

{ dev: 2114,
  ino: 48064969,
  mode: 33188,
  nlink: 1,
  uid: 85,
  gid: 100,
  rdev: 0,
  size: 527,
  blksize: 4096,
  blocks: 8,
  atime: Mon, 10 Oct 2011 23:24:11 GMT,
  mtime: Mon, 10 Oct 2011 23:24:11 GMT,
  ctime: Mon, 10 Oct 2011 23:24:11 GMT
}

```

这里有我们需要的文件的最后更改时间 `mtime` 变量值。

【代码说明】

- ❑ `lastModified = fileInfo.mtime.toUTCString()`: 获取文件最后更改时间, 并转化为 UTC 字符串;
- ❑ `res.setHeader("Expires", date.toUTCString())`: 设置响应 header 的 expires 值;
- ❑ `res.setHeader("Cache-Control", "max-age=" + CACHE_TIME)`: 设置响应 header 的 Cache-Control 值, 缓存时间;

- ❑ `req.headers['if-modified-since'] && lastModified == req.headers['if-modified-since']`: 判断服务器文件是否有更改;
- ❑ `res.writeHead(304, "Not Modified")`: 服务器静态文件没有改变时, 返回 304 Not Modified;
- ❑ `res.setHeader("Last-Modified", lastModified)`: 静态文件有所更改时, HTTP 响应重新设置浏览器 Last-Modified 变量值。

运行 `client.js`, 清除浏览器缓存 (最好能够清除全部缓存, 避免前几天留下的缓存影响结果), 预期将会产生 3 个 HTTP 请求, 并且 3 个请求返回的都是 200。打开浏览器, 输入 `http:127.0.0.1:1337/`, 看到如图 3-16 所示的 3 个 HTTP 请求和预期的一样, 3 个 HTTP 请求返回的都是 200。因为清除缓存浏览器没有缓存, 因此会向服务器拉取一份静态文件。

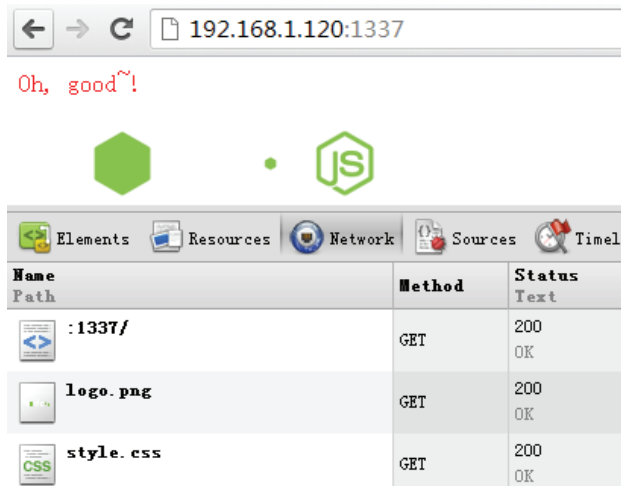


图 3-16 第一次请求无缓存 HTTP 结果图

打开 `logo.png` 和 `style.css` 的 HTTP 的请求, 如图 3-17 和图 3-18 所示, 其中的 HTTP 响应 headers 中包含了 `expires` 和 `Last-Modified` 信息。

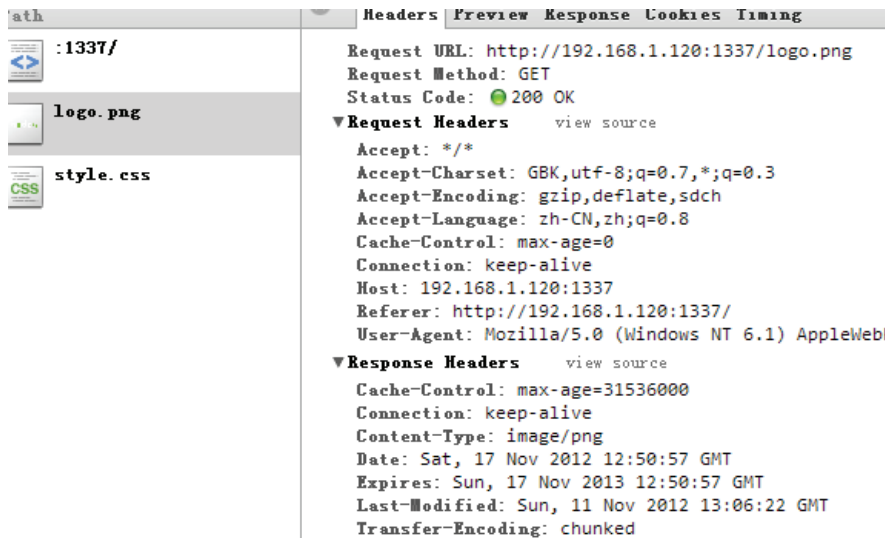


图 3-17 logo.png 请求 headers 信息图

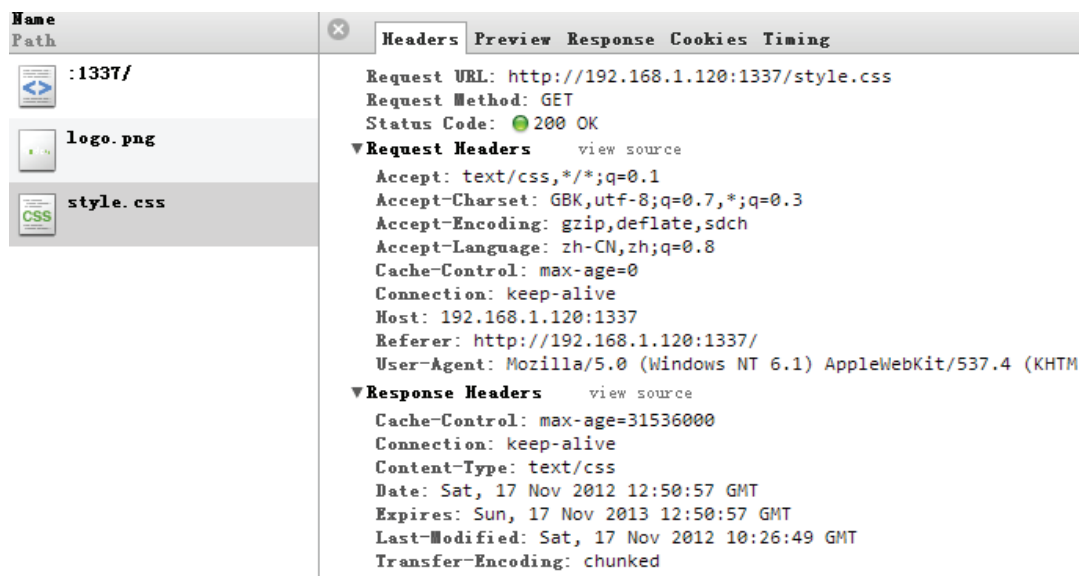


图 3-18 style.css 请求 headers 信息图

接下来我们再次刷新本页面,预期其中的 logo.png 和 style.css 的 HTTP 请求会返回 304 Not Modified。

如图 3-19 所示为刷新后的结果,其中 style.css 和 logo.png 的 HTTP 请求返回的是 304。重新修改服务器的 style.css,添加一行空行,保存后再刷新本页面,如图 3-20 所示。

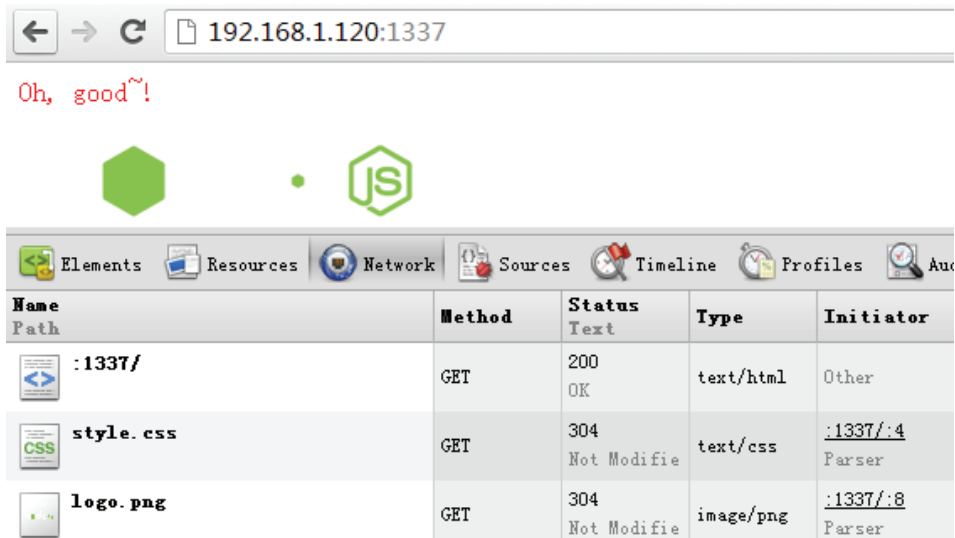


图 3-19 请求有缓存页面返回 HTTP 结果图

从执行结果可以看到,此时 style.css 的 HTTP 请求变为 200,而 logo.png 依然返回的 HTTP 请求为 304 Not Modified。判断和设置 HTTP 的 headers 中的 expires 和 last-modified 参数,从而实现了一个简单的缓存机制,降低服务器 IO 压力。

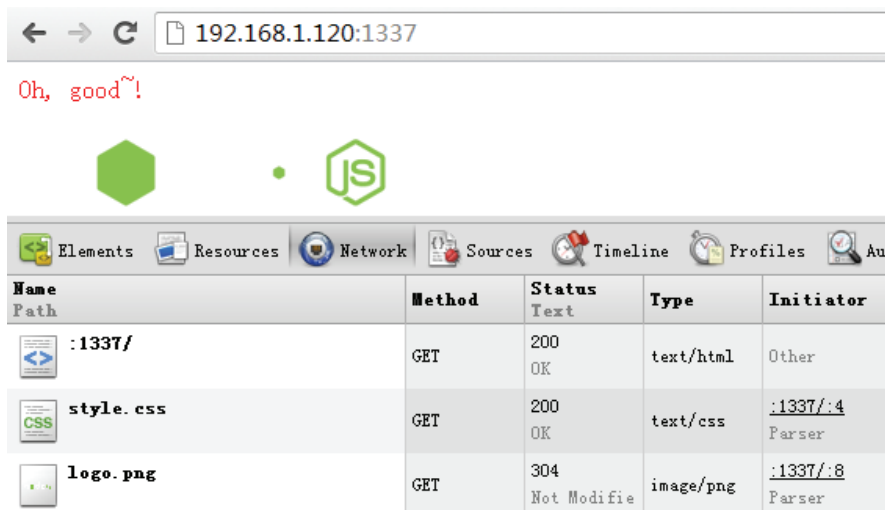


图 3-20 更改 style.css 后重新刷新页面 HTTP 返回图

3.3 文件处理

前面章节的内容大部分涉及了文件的读写功能模块，特别是在本章的 3.2 节中的静态文件的读写。本节将着重介绍文件模块 `FileSystem`，同时结合 `FileSystem` 介绍 Web 应用中如何处理文件图片的上传和下载功能。

3.3.1 File System 模块介绍¹

文件的 I/O 是由标准 POSIX 函数封装而成。需要使用 `require('fs')` 访问这个模块。所有的方法都提供了异步和同步两种方式。

1. 重命名文件

`fs.rename(path1, path2, [callback])` 该 API 的主要作用是重命名某个文件，例如代码：

```
/* */
var BASE_DIR = __dirname;
var fs = require('fs');
fs.rename(BASE_DIR+'/danhuang.txt', BASE_DIR+'/dan.txt', function (err) {
  if (err) throw err;
  console.log('renamed complete');
});
```

【代码说明】

- ❑ `BASE_DIR = __dirname`: 设置当前执行路径。
- ❑ `fs.rename(BASE_DIR+'/danhuang.txt', BASE_DIR+'/dan.txt', function (err){}`: 参数 1 为源文件 `path`，参数 2 为重命名后的文件名。

¹ 参见网站 <http://nodejs.org/api/fs.html>。

执行 rename.js 脚本前，本地路径下有一个 danhuang.txt 文件，执行结束后，本地的 danhuang.txt 被重命名为 dan.txt，如图 3-21 所示。

```
root@ubuntu:/home/danhuang/rename# ls
danhuang.txt  rename.js
root@ubuntu:/home/danhuang/rename# node rename.js
renamed complete
stats: { "dev":64512,"mode":33188,"nlink":1,"uid":0,"gid":0,"rdev":0
e":"2012-11-17T14:50:42.000z","ctime":"2012-11-17T15:08:45.000z"}
root@ubuntu:/home/danhuang/rename# ls
dan.txt  rename.js
root@ubuntu:/home/danhuang/rename#
```

图 3-21 rename 测试代码执行结果图

首次执行 ls 的 Linux 命令的时候显示的是 danhuang.txt 和 rename.js 文件，执行脚本后在本路径文件夹下，再次执行 ls 的 Linux 命令，显示 dan.txt 和 rename 后的文件。

fs.rename(path1, path2, [callback])异步调用函数对应 fs.renameSync(path1, path2)同步调用函数，其作用和调用方法功能都是相同的，但其接口是一个同步接口，因此调用方式有所区别。

2. 修改文件权限和文件权限属组

fs.chmod(path, mode, [callback])该 API 的作用是修改文件权限，如下代码：

```
var BASE_DIR = __dirname;
var fs = require('fs');
fs.chmod(BASE_DIR+'/danhuang.txt', '777', function (err) {
  if (err) throw err;
  console.log('chmod complete');
});
```

【代码说明】

- ❑ fs.chmod(BASE_DIR+'/danhuang.txt', '777', function (err) {}): 将 danhuang.txt 的权限改为 777。

如图 3-22 所示为执行前，danhuang.txt 的权限不是 777 权限，当执行脚本后可以看到 danhuang.txt 的权限被修改为 777。该 API 同样提供了一个同步调用 API fs.chmodSync(path, mode)接口。这里的 mode 为权限字符串，例如 777、775 等。

```
root@ubuntu:/home/danhuang/chmod# ll
ls: 初始化月份字符串出错
总用量 12
drwxr-xr-x  2 root    root    4096 117 23:17 ./
drwxr-xr-x 19 danhuang danhuang 4096 117 23:17 ../
-rw-r--r--  1 root    root    172 117 23:17 chmod.js
-rw-r--r--  1 root    root      0 117 23:15 danhuang.txt
root@ubuntu:/home/danhuang/chmod# node chmod.js
chmod complete
root@ubuntu:/home/danhuang/chmod# ll
ls: 初始化月份字符串出错
总用量 12
drwxr-xr-x  2 root    root    4096 117 23:17 ./
drwxr-xr-x 19 danhuang danhuang 4096 117 23:17 ../
-rw-r--r--  1 root    root    172 117 23:17 chmod.js
-rwxrwxrwx  1 root    root      0 117 23:15 danhuang.txt*
root@ubuntu:/home/danhuang/chmod#
```

图 3-22 执行 chmod 脚本后，文件夹权限对比图

`fs.chown(path, uid, gid, [callback])` API 可修改文件的用户名和属组，这里的 `uid` 和 `gid` 分别对应系统中的 `user` 和 `group`。使用方法和 `fs.chmod` 类似，其同样提供同步调用方法。

3. 获取文件元信息

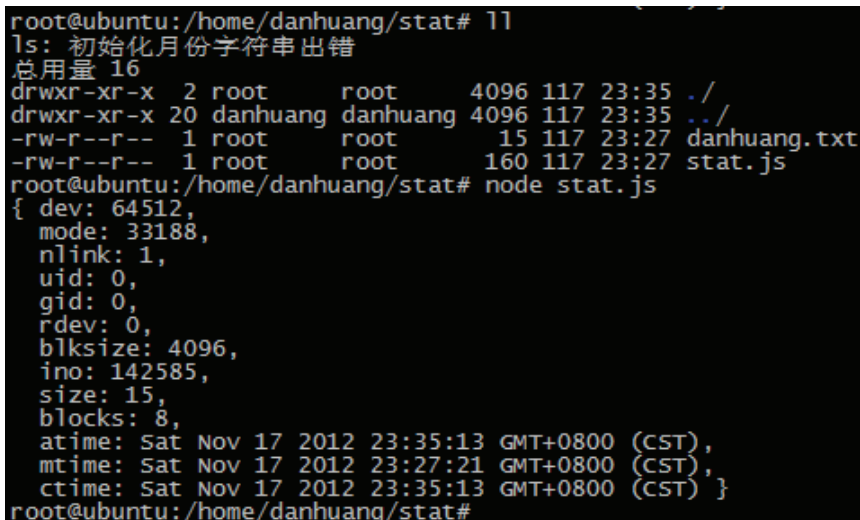
`fs.stat(path, [callback])` 该 API 的作用是读取文件元信息，回调函数将返回两个参数(`err`, `stats`)，其中 `stats` 是 `fs.Stats` 的一个对象。在 3.2 小节中使用了 `fs` 的 `stat` API 来获取文件的最后更改时间。

```
var BASE_DIR = __dirname;
var fs = require('fs');
fs.stat(BASE_DIR+'/danhuang.txt', function (err, stats) {
  if (err) throw err;
  console.log(stats);
});
```

【代码说明】

❑ `fs.stat(BASE_DIR+'/danhuang.txt', function (err, stats) {})`: 获取 `danhuang.txt` 的文件元信息。

如图 3-23 所示返回了文件的所有信息，其中 `dev` 为设备编号；`mode` 为文件的类型和存储权限；`nlink` 为连到该文件的硬连接数目，刚建立的文件值为 1；`uid` 为用户 ID；`gid` 为组 ID；`rdev` 为若该文件为设备文件，则为其设备编号，否则为 0；`blksize` 为块的大小，文件系统的 I/O 缓冲区大小；`ino` 为节点；`size` 为文件节数的大小；`blocks` 为块数；`atime` 为最后一次访问时间；`mtime` 为最后一次更改时间；`ctime` 为最后一次改变时间（指文件的属性）。



```
root@ubuntu:/home/danhuang/stat# ll
ls: 初始化月份字符串出错
总用量 16
drwxr-xr-x  2 root    root    4096 117 23:35 ./
drwxr-xr-x 20 danhuang danhuang 4096 117 23:35 ../
-rw-r--r--  1 root    root     15 117 23:27 danhuang.txt
-rw-r--r--  1 root    root    160 117 23:27 stat.js
root@ubuntu:/home/danhuang/stat# node stat.js
{ dev: 64512,
  mode: 33188,
  nlink: 1,
  uid: 0,
  gid: 0,
  rdev: 0,
  blksize: 4096,
  ino: 142585,
  size: 15,
  blocks: 8,
  atime: Sat Nov 17 2012 23:35:13 GMT+0800 (CST),
  mtime: Sat Nov 17 2012 23:27:21 GMT+0800 (CST),
  ctime: Sat Nov 17 2012 23:35:13 GMT+0800 (CST) }
```

图 3-23 执行 `stat` 脚本返回结果图

`fs.stat(path)` 的同步调用方法是 `fs.statSync(path)`。

4. 读取文件数据

`fs.readFile(path, [callback])` 该 API 的主要作用是解析读取文件数据，这里就不再举例了，

在 3.2 节中主要是应用该 `readFile` 读取静态文件数据，并返回到客户端的。

5. 验证文件存在

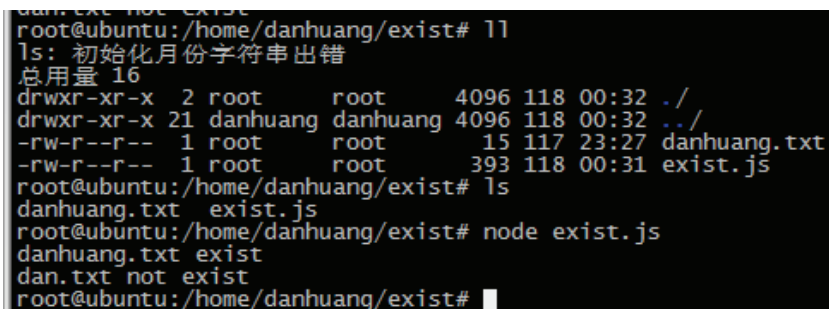
`fs.exists(path, [callback])` 该 API 的主要作用是判断文件是否存在，部分代码如下：

```
/* exist.js */
var BASE_DIR = __dirname;
var fs = require('fs');
fs.exists(BASE_DIR+'/danhuang.txt', function (existBool) {
  if(existBool){
    console.log('danhuang.txt exist');
  } else {
    console.log('danhuang.txt not exist');
  }
});
/* 判断文件是否存在 */
fs.exists(BASE_DIR+'/dan.txt', function (existBool) {
  if(existBool){
    console.log('dan.txt exist');
  } else {
    console.log('dan.txt not exist');
  }
});
```

【代码说明】

- ❑ `fs.exists(BASE_DIR+'/danhuang.txt', function (existBool){})`：验证 `danhuang.txt` 是否存在，函数异步执行完后，将验证结果返回到回调函数的参数中，如 `existBool`；
- ❑ `fs.exists(BASE_DIR+'/dan.txt', function (existBool) {})`：验证 `dan.txt` 是否存在。

在本地文件夹中，创建 `danhuang.txt` 文件，通过创建该文件来测试验证 `exist.js` 脚本的执行结果。运行 `exist.js` 文件，结果如图 3-24 所示。



```
dan.txt not exist
root@ubuntu:/home/danhuang/exist# ll
ls: 初始化月份字符串出错
总用量 16
drwxr-xr-x  2 root    root    4096 118 00:32 ./
drwxr-xr-x 21 danhuang danhuang 4096 118 00:32 ../
-rw-r--r--  1 root    root      15 117 23:27 danhuang.txt
-rw-r--r--  1 root    root     393 118 00:31 exist.js
root@ubuntu:/home/danhuang/exist# ls
danhuang.txt  exist.js
root@ubuntu:/home/danhuang/exist# node exist.js
danhuang.txt exist
dan.txt not exist
root@ubuntu:/home/danhuang/exist#
```

图 3-24 执行 `exist` 验证文件存在结果图

从图 3-24 中可以看到，本地文件夹中包含 `danhuang.txt` 和 `exist.js` 文件，不存在 `dan.txt` 文件，因此当我们执行 `exist.js` 后，返回的是 `danhuang.txt` 是存在的，而 `dan.txt` 是不存在的。

问题：`fs.exists` 是否可验证自身执行文件，例如该文件夹下的 `exist.js`，能从中得出什么结论？希望大家能够亲自实践。

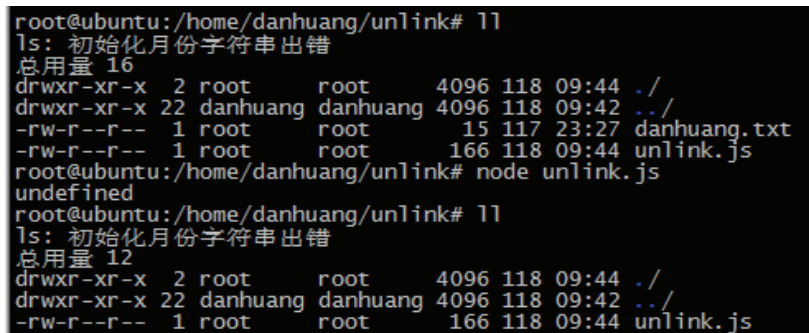
6. 删除文件

`fs.unlink(path, [callback])` 该 API 的主要作用是删除文件，示例代码如下：

```
var BASE_DIR = __dirname;
var fs = require('fs');
fs.unlink(BASE_DIR+'/danhuang.txt', function (err) {
  if (err) throw err;
});
```

fs.unlink 回调函数只有一个异常参数 err。

如图 3-25 所示为执行结果，成功地删除了文件 danhuang.txt。



```
root@ubuntu:/home/danhuang/unlink# ll
ls: 初始化月份字符串出错
总用量 16
drwxr-xr-x  2 root    root    4096 118 09:44 ./
drwxr-xr-x 22 danhuang danhuang 4096 118 09:42 ../
-rw-r--r--  1 root    root     15 117 23:27 danhuang.txt
-rw-r--r--  1 root    root    166 118 09:44 unlink.js
root@ubuntu:/home/danhuang/unlink# node unlink.js
undefined
root@ubuntu:/home/danhuang/unlink# ll
ls: 初始化月份字符串出错
总用量 12
drwxr-xr-x  2 root    root    4096 118 09:44 ./
drwxr-xr-x 22 danhuang danhuang 4096 118 09:42 ../
-rw-r--r--  1 root    root    166 118 09:44 unlink.js
```

图 3-25 执行文件删除前后目录文件对比图

7. 文件读写

fs.write(fd, buffer, offset, length, position, [callback])该 API 是将 buffer 缓冲器内容写入 fd 文件。position 指明将数据写入文件从头部算起的偏移位置，若 position 为 null，数据将从当前位置开始写入。回调函数接收两个参数(err, written)，其中 written 标识有多少字节的数据已经写入。

fs.read(fd, buffer, offset, length, position, [callback])该 API 从 fd 文件中读取数据，buffer 为写入数据的缓冲器；offset 为写入到缓冲器的偏移地址；length 指明了欲读取的数据字节数；position 为一个整型变量，标识从哪个位置开始读取文件，如果 position 参数为 null，数据将从文件当前位置开始读取。回调函数接受两个参数 (err, bytesRead)，其中 bytesRead 标识多少字节被读取。

Buffer 这个参数可通过 Node.js Buffer API¹中的 new Buffer 创建。

其他文件操作方法大家可以尝试使用同样的方法去学习，这里只是教大家一个方法。官网中介绍了每个 API 的调用方法，其中也说明了回调函数有多少个参数，如果官网没有介绍回调函数的参数，大家可以尝试把每个回调函数的参数打印出来。

3.3.2 图片和文件上传

Web 应用中大部分都涉及图片和文件的上传功能，本节将介绍服务器端如何使用 Node.js 获取客户端传递的图片和文件数据。

本节涉及的 Node.js API 主要是 HTTP、FileSystem、querystring 和 url 等模块。HTTP 模块创建服务器和 HTTP 的处理请求，FileSystem 用的图片文件的处理，querystring 处理

¹ 参见网站 <http://nodejs.org/api/buffer.html>。

字符串, url 模块处理 url 解析。其中还涉及 GET、POST 请求处理模块和静态文件管理模块。

文件上传和图片上传前端页面原则上都是一致的, 主要是通过 POST file 数据, 而 Node.js 服务器端则需要利用 NPM 中一个应用模块 formidable 来处理图片的上传。

下面的示例详细讲解图片和文件上传功能的实现。创建一个 index.html, 主要作用是上传图片, 将图片提交到 <http://127.0.0.1:1337/upload>。代码如下:

```
<html>
  <head>
    <meta charset="utf-8">
    <title>Node.js Upload</title>
    <link rel="stylesheet" href="static/style.css">
  </head>
  <body>
    <div id='main_content'>
      <div>
        <form ENCTYPE="multipart/form-data" action='upload'>
          <input TYPE="file" NAME="image"/>
          <input type='submit' id='upload' value='上传图片'>
        </form>
      </div>
    </div>
  </body>
</html>
```

【代码说明】

❑ ENCTYPE="multipart/form-data" action='upload': 注意, 这里需要指定 ENCTYPE, 设置 action 提交到 upload 接口。

使用 show_image.html 显示上传后的图片。代码如下:

```
<html>
  <head>
    <meta charset="utf-8">
    <title>Node.js Show Image</title>
    <link rel="stylesheet" href="static/style.css">
    <script src="static/jquery-1.8.3.min.js"></script>
    <script src="static/index.js"></script>
  </head>
  <body>
    <div id='main_content'>
      <div id='show_image'>
        <img src='/uploadFile/test.png' alt='upload file' />
      </div>
    </div>
  </body>
</html>
```

【代码说明】

❑ : html 中 image 图片显示的方法。注意, 这里指定了图片的 url 为 /uploadFile/test.png。

show_image.html 中没有通过服务器端传递图片地址, 而是在客户端指定访问的图片地址, 这不是我们需要的结果。为了演示图片上传功能, 此次暂时使用指定图片地址的方法。在后面将介绍应用 jade 模板来传递服务器端参数。

图片上传功能在前面已经介绍过, 需要用 NPM 的一个模块 formidable¹来处理文件的

1 参见网站 <https://github.com/felixge/node-formidable>。

上传功能。formidable 的安装配置，在 github 开源主页上有介绍，大家可通过脚注的 url 前往查看学习。

重点：应用 formidable 实现图片上传功能的代码如下：

```
var formidable = require("formidable");
function upload(res, req){
  /* 获取 show_image.html 文件路径 */
  var readPath = __dirname + '/' + url.parse('show_image.html').pathname;
  /* 读取 show_image.html 数据 */
  var indexPage = fs.readFileSync(readPath);
  /* 创建 formidable 表单对象 */
  var form = new formidable.IncomingForm();
  /* 执行 form 表单数据解析，获取其中的 post 参数 */
  form.parse(req, function(error, fields, files) {
    fs.renameSync(files.image.path, BASE_DIR + '/uploadFile/test.png');
    res.writeHead(200, { 'Content-Type': 'text/html' });
    res.end(indexPage);
  });
}
```

【代码说明】

- ❑ formidable = require("formidable"): 安装成功 formidable 模块后，require 该模块。
- ❑ form = new formidable.IncomingForm(): 创建 form 对象。
- ❑ form.parse(req, function(error, fields, files) {}): 获取上传文件数据，files 为文件 json 对象。
- ❑ fs.renameSync(files.image.path, BASE_DIR + '/uploadFile/test.png'): 同步获取上传文件，并保存在/uploadFile 下，重命名为 test.png。

从中可以了解到如何应用 formidable 实现文件的上传，本部分包含两个过程：1. 创建一个 formidable.IncomingForm() 返回的对象；2. 调用该对象中的 parse 异步方法获取 files 相应信息。如果大家对其返回的参数不是很明确，可以通过 console.log 来查看 files 对象数据。

上传文件完成后，进入 show_image 页面，查看当前上传的图片。这部分功能的实现应用到了之前我们讲解到的静态文件存储和静态文件缓存模块，由于没有添加 form 表单的参数提交，因此这里没有应用到 httpParam 模块，在本章 jade 模块实现中会应用该模块获取 form 提交的 POST 和 GET。代码如下：

```
var pathname = url.parse(req.url).pathname;
httpParam.init(req, res);
switch(pathname){ // 根据 pathname 来做路由分发处理
  /* 执行文件上传路径 */
  case '/upload' : upload(res, req);
  break;
  /* 响应图片信息到客户端 */
  case '/image' : showImage(res, req);
  break;
  /* 响应 HTML 到客户端 */
  case '/' : defaultIndex(res);
  break;
  /* 响应 HTML 到客户端 */
  case '/index' : defaultIndex(res);
  break;
  case '/show' : show(res);
  break;
}
```

```
/* 静态服务器资源 */
default      : staticModule.getStaticFile(pathname, res, req);
break;
}
```

【代码说明】

- ❑ httpParam.init(req, res): 初始化 GET 和 POST 参数获取模块 httpParam 的 req 和 res。
- ❑ case '/upload': upload(res, req): 请求路径名为 upload 时, 执行上传文件逻辑。
- ❑ case '/': defaultIndex(res): 进入默认上传页面。
- ❑ case '/show': show(res): 进入上传图片展示页面。
- ❑ default : staticModule.getStaticFile(pathname, res, req): 其他情况下默认为静态资源请求, 因此使用静态资源模块来处理。

由于是一个小项目, 因此这里使用了一种简单的路由方式, 这部分代码在严谨性上还欠缺。由于时间的原因, 这本书中主要是希望大家能够了解 Node.js 的一些功能 API 的使用方法, 对于异常处理本书要求不是很严格, 但希望大家能够将示例代码加以补充使其更完美。运行项目中的 index.js, 返回结果如图 3-26 所示。



图 3-26 Web 页面以及 HTTP 响应图

因为调用了缓存模块, 所以这里的 style.css 返回的是 304, 而“上传图片”按钮的背景图片也是根据服务器返回的 304 状态码来读取本地缓存获取的。选择一个文件并上传, 单击“上传图片”按钮后, 返回如图 3-27 所示的结果, 展示上传的图片。

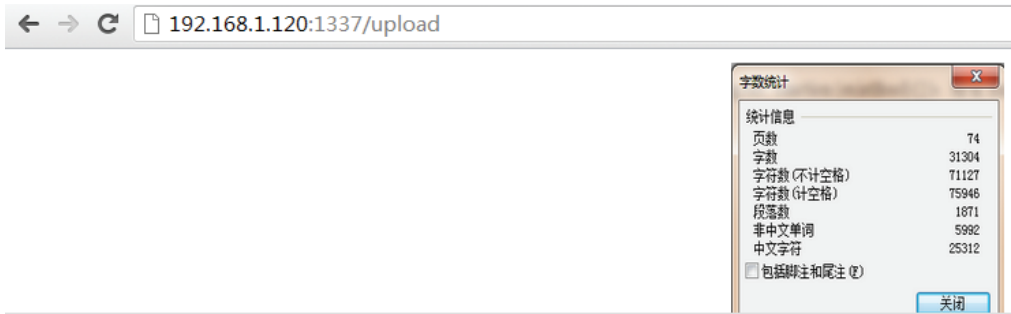


图 3-27 文件上传结果返回页面图

接下来我们看一下 `show_image.html` 文件，其中的 `img src` 指定了 url，并不是服务器传递的上传图片地址返回到客户端。代码如下：

```
<html>
  <head>
    <meta charset="utf-8">
    <title>Node.js Show Image</title>
    <link rel="stylesheet" href="static/style.css">
    <script src="static/jquery-1.8.3.min.js"></script>
  </head>
  <body>
    <div id='main_content'>
      <div>
        <img src='/uploadFile/test.png' alt='upload file' />
      </div>
    </div>
  </body>
</html>
```

【代码说明】

❑ ``：src 指定图片 url。

接下来我们希望通过客户端能够传递多个图片到服务器，并且命名规则为原文件名+时间戳，上传完成后在 Web 页面展示图片。

3.3.3 jade 模板实现图片上传展示功能

在第 2 章中我们介绍了 jade 模板的作用，以及如何安装配置，本节将应用该模块实现图片上传和展示功能。功能要求如下：

- ❑ 应用 jade 模板展示所有前端页面。
- ❑ 可上传多个图片到服务器，命名必须为“原文件名_时间戳”，并提交一个 input 的图片名称。
- ❑ 展示图片名称和图片。

jade 模板首先读取模板文件内容信息，根据 jade 规则将模板文件中的相应的参数替换为相应的变量值，然后返回参数替换后的文件内容数据。根据上述要求，我们首先将 `index.html` 页面转化为 jade 页面。介绍如何将 `index.html` 转化为 jade 页面前，先介绍一些基本的 html 和 jade 模板页面的转化。需要注意的是，在 jade 模板中标签是用空格来对齐的，表示标签中的包含关系。以下是标签转化的例子。

```
html
```

jade 在解析时将转化为：

```
<html></html>
```

同理 `div` 将会转化为 `<div></div>`，在 jade 模板中标签的结尾是通过文本左对齐方式来判断的。从上往下看，只要在标签右侧的数据都为其子模块，如果在同一列中有其他元素出现，则代表上一个标签的结束，如图 3-28 所示，很好地诠释了 jade 模板中的对齐问题。



图 3-28 jade 模板代码结构图

图中 3-28 的 head 和 body 标签都在 html 右侧，因此都为其子标签，head 在同一列中遇到了 body 标签，代表 head 标签结束，body 为其同级别标签。同样，meta、title 和 link 因为在其右侧，所以都为其子标签，body 下的子标签也可这样理解。

当标签有其他属性值，例如 id 或者 class，id 在 jade 中可以使用 div#main_content 方式，在解析时会被转化为 <div id='main_content'> </div>。当然如果一个标签存在多个 id 和多个 class 时，可以同样的方式，例如：div#main_content.main.other_class 相当于 <div id='main_content' class='main other_class'></div>。标签的属性表示方式是使用括号，例如：

```
<link rel="stylesheet" href="static/style.css">
```

可转化为 jade 模版中的：

```
link(rel="stylesheet",href="static/style.css")
```

最后再介绍 jade 模板中，Node.js 服务器端与 jade 模板的参数传递方式。例如，服务器希望传递一个 url 字符串到 jade 模板中，那么在 jade 模板中可以使用 #{url} 获取。如果返回的是一个 json 对象，例如 data={url:'http://test', 'name': 'test'} 时，获取 url 和 name 的方法是 #{data.url}。根据上述的转化技巧，我们将 index.html 转化为 jade 模板，代码如下：

```
html
  head
    meta(charset="utf-8")
    title Node.js Upload
    link(rel="stylesheet",href="static/style.css")
  body
    div#main_content
      div
        form(ENCTYPE="multipart/form-data",action='upload', method=
          'POST')
          input(TYPE="file",NAME="image")
          图片名称
          input(TYPE="text",NAME="name")
          input(type='submit',id='upload',value='上传图片')
```

【代码说明】

□ input(TYPE="text",NAME="name"): 新增部分提交图片的名称。

上面介绍到 jade 模板是通过文本对齐方式来判断标签是否结束，例如其中的 head 标

签中包含了 meta、title、link 等标签，但 body 标签就不属于 head 标签，原因是 body 标签和 head 标签在同一级别上，都为 html 的子标签。因此在应用 jade 模板时特别需要注意文本对齐，否则页面会混乱。jade 的其他功能可以前往 jade 的 github 开源主页¹学习。jade 模板提供了一个 Public API 调用方式，代码如下：

```
var jade = require('jade');
// Compile a function
var fn = jade.compile('string of jade', options);
fn(locals);
```

我们可以灵活地将其提供的 API 转化为一个 util 工具函数，添加到 res 对象中。将该函数添加到 res 对象中的原因主要是该对象是贯穿整个 HTTP 请求响应处理逻辑，而 jade 模板的功能实际上就是 HTTP 响应返回一个 html 数据，与 res 对象的功能恰好是相似的，因此将该方法添加到 res 对象中是非常合适的，其实现代码如下：

```
res.render = function(template, options){
  /* 同步读取 jade 模板文件数据 */
  var str = require('fs').readFileSync(template, 'utf8');
  /* 获取 jade 模板编译处理函数 */
  var fn = jade.compile(str, { filename: template, pretty: true });
  /* 调用 fn 函数，将 jade 模板转化为 html 文件数据字符 */
  var page = fn(options);
  res.writeHead(200, { 'Content-Type': 'text/html' });
  res.end(page);
}
```

【代码说明】

- ❑ `res.render = function(template, options){}`：为 res 对象添加 render 方法，这样可以在任何 res 对象存在的地方直接调用 render 函数指定显示页面信息，其中 template 为模板名，options 为需要传递的参数；
- ❑ `fn = jade.compile(str, { filename: template, pretty: true })`：编译 jade 模板的编译函数；
- ❑ `var page = fn(options)`：将传递的参数添加到模板中并且编译，返回编译后的 html 字符串。

成功为 res 添加该函数后，我们就可以将 defaultIndex(res) 这个函数重构，使用 res 对象来响应 HTTP 的请求，代码如下：

```
function defaultIndex(res){
  res.render('index.jade', {'user': 'danhuang'});
}
```

【代码说明】

- ❑ `res.render('index.jade', {'user': 'danhuang'})`：指定显示 index.jade，并传递 user 参数。

执行 index.js，打开 <http://127.0.0.1:1337/> 就可以看到之前的页面信息。我们利用同样的方法将 show_image.html 修改为 jade 模板文件，如下为更改后的代码：

```
html
  head
    meta(charset="utf-8")
    title Node.js Show Image
```

¹ 参见网站 <https://github.com/visionmedia/jade>。

```

    link(rel="stylesheet",href="static/style.css")
    script(src="static/jquery-1.8.3.min.js")
  body
    div#main_content
      div
        img(src='#{imgUrl}',alt='upload file')

```

【代码说明】

- `img(src='#{imgUrl}',alt='upload file')`: `imgUrl` 为 Node.js 服务器端传递的参数, 获取方法在前面已经介绍。

最后将 `index.js` 中的 `upload` 方法重构, 代码如下:

```

function upload(res, req){
  var timestamp = Date.parse(new Date());
  /* 获取 formidable 类的对象 */
  var form = new formidable.IncomingForm();
  /* 应用 parse 方法解析 post 数据 */
  form.parse(req, function(error, fields, files) {
    var fileName = timestamp + '_' + files.image.name;
    fs.renameSync(files.image.path, BASE_DIR + '/uploadFile/' + fileName);
    res.render('show_image.jade',{ 'imgUrl': '/uploadFile/' + fileName});
  });
}

```

【代码说明】

- `res.render('show_image.jade',{ 'imgUrl': '/uploadFile/' + fileName})`: 指定显示 `show_image.jade`, 同时传递 `imgUrl` 参数到客户端。

重构结束后, 运行 `index.js` 文件, 打开浏览器输入 `http:127.0.0.1:1337`, 会看到如图 3-29 所示的页面。

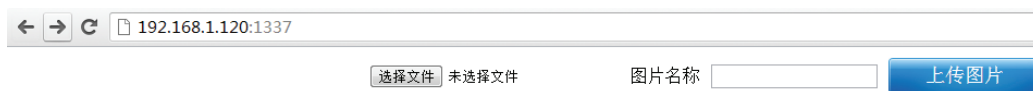
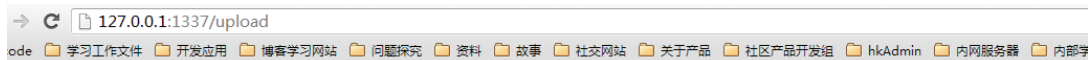


图 3-29 文件上传页面图

选择相应的图片, 单击“上传图片”按钮后, 可以看到图片上传信息, 如图 3-30 和图 3-31 所示是两次上传图片的结果展示, 需要注意的是, 图片上传了多次, 目的是验证图片上传多次是否会覆盖原有图片。上传文件以及路径中请确认不含有中文字符, 具体原因及解决办法可以查看下一节介绍的本应用存在的问题。



图片展示



图 3-30 文件成功上传页面返回图



图 3-31 文件成功上传页面返回图

最后查看服务器是否存储了这些图片信息。进入项目文件夹中的 `uploadFile` 目录查看当前上传文件信息，如图 3-32 所示。

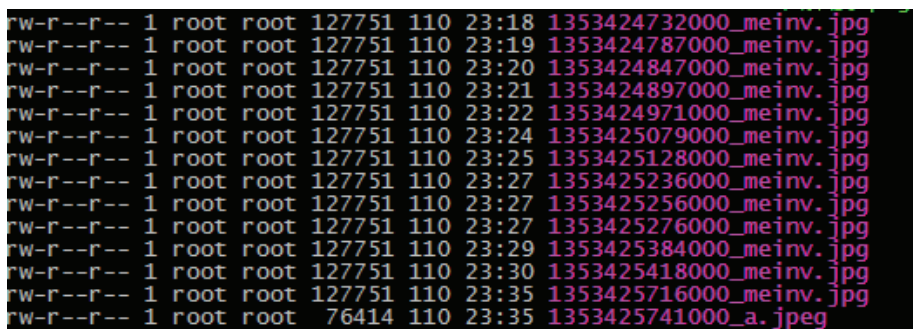


图 3-32 上传文件夹文件展示图

其中的 `meinv.jpg` 因为上传了多次，所以在服务器存储了多个版本，且其存储方式是时间戳+原有图片名称。请注意，避免同一文件上传多次被覆盖的情况。

3.3.4 上传图片存在的问题

(1) 路径名存在中文或者上传图片的名称含中文名时，可以正常上传，但无法展示图片信息。

由于 `url` 在传递的时候浏览器会自动地将请求 `url` 使用 `encodeURIComponent` 进行转码，因此我们需要使用 `decodeURI` 将其解码获取中文字符，第一种方式是在获取 `pathname` 时进行解码，修改 `pathname` 赋值，代码如下：

```
var pathname = decodeURI(url.parse(req.url).pathname);
```

对于这个问题，我们需要将之前的静态模块修改，将其中的 `pathname` 进行 `decodeURI`，避免出现类似的情况。

第二种方式是修改 `static_module.js`，代码如下：

```
exports.getStaticFile = function(pathname, res, req){
  pathname = decodeURI(pathname);
```

将 `pathname` 进行解码，在实际情况中很多人可能会用到中文的文件夹名称，因此在请求数据的时候也会出现无法读取文件的结果，这个时候就需要将获取的 `pathname` 进行转码。

本章节前面的代码都没有进行相应的解码过程，希望读者在应用中注意加上 `pathname` 的解码，保证服务器正常运行。

(2) 该上传图片功能在 Windows 运行时会出现异常，出现异常如图 3-33 所示。

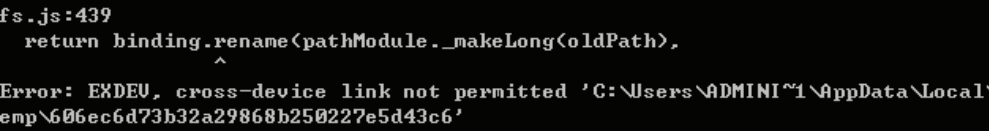


图 3-33 Windows 系统文件上传异常错误图

异常原因是 Windows 下无权限跨盘 `rename` 文件，Windows 下是禁止跨盘执行 Node.js API 的 `rename` 操作的。

解决办法 1：将该代码放在 C 盘运行，这样就能保证在同一个磁盘。

解决办法 2：如果想在 Windows 下跨盘实现文件上传的话也是没问题的，下面一段代码可在 Windows 下实现图片和文件上传，仅供参考。

```
var target_path = "/public/images/atvatar/" + timestap + "_" + username +
    "." + imageType;
    fs.readFile(tmp_path, function(error, data){
        fs.writeFile(BASE_DIR + target_path, data, function (err) {
            if (err) throw err;
        });
    });
```

思路：通过 `files` 数据获取该文件的名称和后缀名，通过使用 `readFile` 来读取上传缓存图片中的数据，然后使用 `writeFile` 将读取的数据写入新的文件中，新文件使用原有的文件名和后缀名来保存数据。

这两种解决方式笔者更倾向于第一种方式，毕竟第二种方式在文件信息很大时，将会耗费相对较大的资源在文件的读写上。

3.3.5 文件读写

应用介绍：希望通过 Web 客户端可以在线进行对文件编辑操作，实时地对服务器文件进行更新，实现类似本地文件编辑功能，同时可以保存文件，下次进入可以同样进行更改。

本节将会应用到的 Node.js 原生模块和 NPM 模块，如表 3.1 所示。

表 3.1 应用到的模块

应用模块	描述	应用模块	描述
http 模块	服务器创建	url 模块	url 请求路径
fs 模块	文件的读写	jade 模块	NPM 模块，前端模块
socket.io 模块	与服务器实时地建立链接	staticModule 模块	静态文件管理模块
querystring 模块	获取请求参数		

分析完需要的模块后，先将需要的模块 `require` 到本项目中，如下：

```
/* 首先 require 加载模块 */
var http      = require('http'),           // Web 服务器创建模块
    fs        = require('fs'),             // fs 文件处理模块
    url       = require('url'),            // url 字符处理模块
    querystring = require('querystring'),  // 字符串处理模块
    httpParam  = require('./http_param'),  // HTTP 参数获取模块
    staticModule = require('./static_module'), // 静态服务器模块
    jade       = require('jade'),          // jade 模板模块
    socket     = require('socket.io');     // socket 模板模块
var BASE_DIR  = __dirname,
    filePath = BASE_DIR + '/test.txt';
```

【代码说明】

❑ `BASE_DIR=__dirname`：设置项目根路径。

❑ `filePath = BASE_DIR + '/test.txt'`：设置服务器读取和保存文件名。

可以在每个项目的入口都设置一个文件根目录参数，例如上面代码中的 `BASE_DIR`，使用绝对路径来开发项目，避免大量的相对路径，导致代码维护起来比较困难。

创建 HTTP 服务器，并获取 `socket` 的 `io` 对象，`socket` 模块中的 `listen` API 的参数是 `http.createServer` 返回的函数对象。代码如下：

```
var app = http.createServer(function(req, res) {
  /* 为 HTTP 响应对象 res 新增 jade 模块解析方法 */
  res.render = function(template, options){
    var str = fs.readFileSync(template, 'utf8');
    var fn = jade.compile(str, { filename: template, pretty: true });
    var page = fn(options);
    res.writeHead(200, { 'Content-Type': 'text/html' });
    res.end(page);
  }
  /* 获取用户 url 请求路径，并应用 decodeURI 解析 url 中的特殊字符和中文 */
  var pathname = decodeURI(url.parse(req.url).pathname);
  /* 初始化 httpParam 模块 */
  httpParam.init(req, res);
  if(pathname == '/favicon.ico'){
    return;
  }
  /* 路由处理 */
  switch(pathname){
    case '/' : defaultIndex(res);
    break;
    case '/index' : defaultIndex(res);
    default : staticModule.getStaticFile(pathname, res, req);
    break;
  }
}).listen(1337);

io = socket.listen(app);
```

【代码说明】

❑ `var app = http.createServer(function(req, res) {})`：创建 HTTP 服务器，并将返回对象

赋值给 app。

- ❑ `var pathname = decodeURI(url.parse(req.url).pathname)`: 解析中文的 url 字符。
- ❑ `io = socket.listen(app)`: 创建 socket 服务器对象。

本段代码主要是创建 HTTP 服务器，并根据 socket 模块和 `http.createServer` 返回对象，创建 socket 服务器的 io 对象。

根据获取 socket 的 io 对象创建 socket 服务器。代码如下：

```
io.sockets.on('connection', function (socket) {    // 监听客户端连接
    var message = fs.readFileSync(filePath, 'utf8');
    // 监听 change_from_server 消息
    socket.emit('change_from_server', { msg: message});
    socket.on('success', function (data) {          // 监听 success 消息
        console.log(data.msg);
    });
    socket.on('data', function (data) {
        writeFile(data.msg, function(){
            socket.emit('change_from_server', { msg: data.msg});
        });
    });
});
```

【代码说明】

- ❑ `io.sockets.on('connection', function (socket) {})`: 创建一个 socket 服务器，等待客户端连接。
- ❑ `var message = fs.readFileSync(filePath, 'utf8')`: 同步以 utf8 读取文件内容。
- ❑ `socket.emit('change_from_server', { msg: message})`: 发送一个 `change_from_server` 消息（消息内容是一个 json 对象，其 key 值为 `msg`，值为 `message`）。
- ❑ `socket.on('success', function (data) {})`: 接收到客户端发送的 `success` 消息时执行相应的 function。
- ❑ `socket.on('data', function (data) {})`: 接收到客户端发送的 `data` 消息时执行相应的 function。
- ❑ `writeFile(data.msg, function() {})`: 将客户端发送消息的内容写入文件中。
- ❑ `socket.emit('change_from_server', { msg: data.msg})`: 发送一个 `change_from_server` 消息到客户端，该消息带有 json 数据。

服务器端 socket 主要是监听客户端连接，监听客户端发送的消息，并回复客户端发送的消息。客户端的 socket 则是主动建立与服务器端的连接，发送消息到服务器端，接收服务器返回的消息并保存到客户端。

整体逻辑如图 3-34 所示。

服务器端监听着客户端的连接，当客户端与服务器 socket 连接成功时，服务器通过使用 `emit` 接口发送 `chang_from_server` 消息到客户端，客户端接收到该消息后，则发送相应的成功信息到服务器端。在退出连接之前，客户端和服务器可以互相推送消息。客户端代码实现如下：

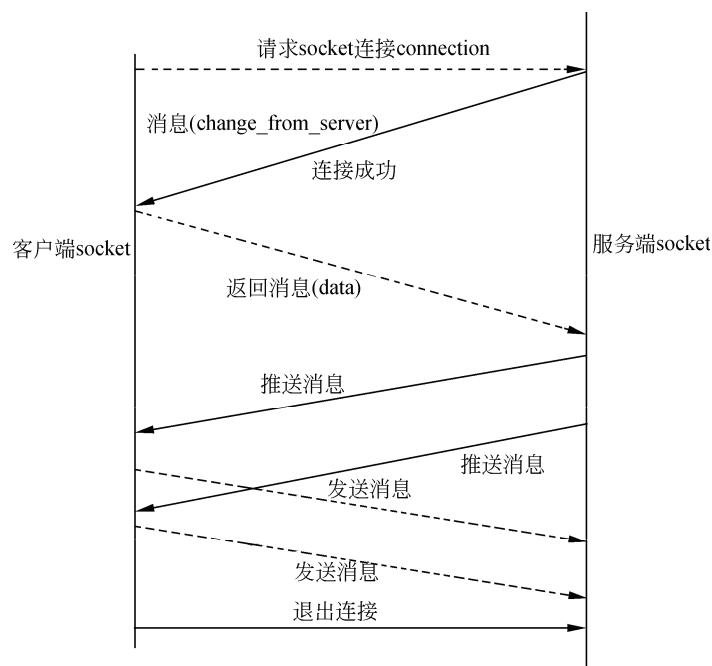


图 3-34 socket 信息交互图

```

var socket = io.connect('http://127.0.0.1:1337');
// 监听 change_from_server 消息
socket.on('change_from_server', function (data) {
    $('textarea').attr('value', data.msg);
});
// Web 浏览器客户端监听键盘事件，当按下键盘时，发送 socket data 消息
$(document).ready(function() {
    $('textarea').keyup(function() {
        socket.emit('data', { msg: $('textarea').attr('value') });
    });
});

```

【代码说明】

- ❑ `var socket = io.connect('http://127.0.0.1:1337');`：连接本地 IP 的 1337 端口下的 socket 服务器。
- ❑ `socket.on('change_from_server', function (data){})`：接收到服务器传递的 `change_from_server` 消息时，执行相应的 `function` 操作。
- ❑ `socket.emit('data', { msg: $('textarea').attr('value') })`：通过 `emit` 主动发送消息到服务器端。
- ❑ `$('textarea').attr('value', data.msg)`：应用 `jquery` 框架，设置 `textarea` 的 `value` 值。

这里 `socket.on` API 接口就是监听接收服务器的 `change_from_server` 消息。当然这里也可以监听接收其他消息，例如接收服务器 `msg` 消息，则添加如下代码：

```

socket.on('msg', function (data) {
    console.log(data.msg);
});

```

如果服务器从不会发送 `msg` 消息时，这样的监听代码是无效的。`socket.on` 中有一个回

调函数，其参数为该消息的内容，内容可以为 json 对象，也可以为一个简单的字符串。

应用 jade 模板创建 html 文件信息。代码如下：

```
html
  head
    meta(charset="utf-8")
    title Node.js file monitor
    link(rel="stylesheet",href="static/style.css")
    script(src="static/jquery-1.8.3.min.js")
    script(src="static/socket.js")
    script(src="static/index.js")
  body
    div#main_content
      h1 文件读写
      div
        textarea(rows="30",cols="120")
        div#upload 保存
```

【代码说明】

- ❑ script(src="static/socket.js")：要包含 socket.js，该文件可去 socket 官网下载，提供前端 JavaScript 的 socket API。
- ❑ script(src="static/jquery-1.8.3.min.js")：使用 jquery 框架。

运行 index.js，在命令窗口可查看到 socket.io 启动和运行 debug 消息，如图 3-35 所示的信息表示 socket.io 服务器正常启动。

```
E:\Desktop\code\chapter_three\3.3\file_monitor>node index.js
info - socket.io started
```

图 3-35 Socket 启动运行日志图

在浏览器中输入 <http://127.0.0.1:1337>，使用客户端连接 socket.io 服务器后，可以看到服务器端 socket 产生了 debug 消息，其中包括 socket 连接以及服务器和客户端之间的消息交互，如图 3-36 所示。

```
E:\Desktop\code\chapter_three\3.3\file_monitor>node index.js
info - socket.io started
debug - client authorized
info - handshake authorized utjLLRBhOPWb_iNH-v1k
debug - setting request GET /socket.io/1/websocket/utjLLRBhOPWb_iNH-v1k
debug - set heartbeat interval for client utjLLRBhOPWb_iNH-v1k
debug - client authorized for
debug - websocket writing 1::
debug - websocket writing 5::{"name":"change_from_server","args":[{"msg":"嘿，Node.js欢迎您！"}]}
```

图 3-36 socket 运行 debug 日志

其中，最后一行表示服务器端发送了一个 change_from_server 消息到客户端，其中包含了一个 json 参数。当我们在客户端修改文件内容时，可以看到服务器端又产生了一些 debug 日志消息，如图 3-37 所示，其中客户端发送了多个消息，当服务器端接收到这些消息，立刻发送一条消息回复客户端。

```
debug - websocket writing 5::{"name":"change_from_server","args":[{"msg":"嘿  
嘿, Node.js欢迎您! ">1}  
debug - websocket writing 5::{"name":"change_from_server","args":[{"msg":"嘿  
嘿, Node.js欢迎您! h">1}
```

图 3-37 Socket 运行 debug 日志

客户端页面如图 3-38 所示, 其中客户端 JavaScript 会监听用户的键盘输入, 每次输入数据后会自动发送一个 Socket 消息到服务器端, 服务器端接收到消息后, 将其内容写入文件, 并返回文件最后更新数据到客户端, 客户端接收消息后更新 textarea 文本框内容。

文件读写

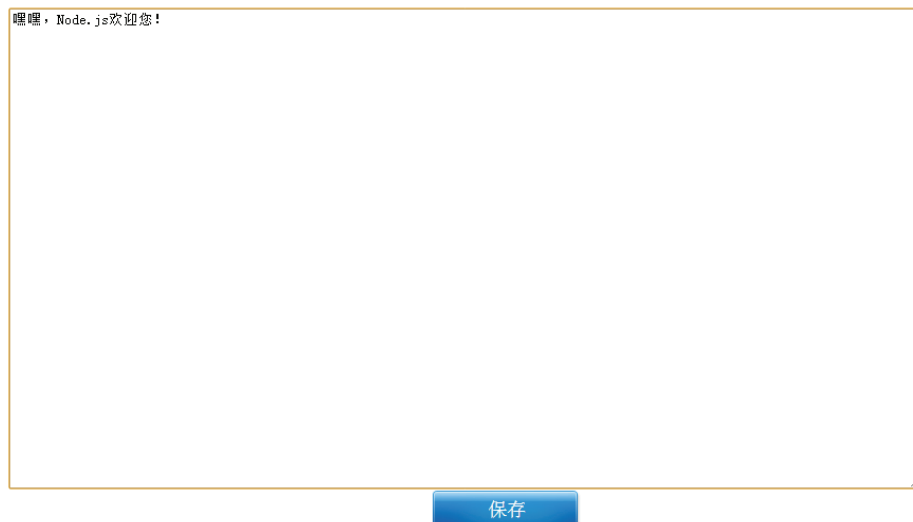


图 3-38 文件读写 Web 页面

应用该功能可以做一个在线文件编辑功能, 用户进入系统后可在本系统编辑一些文件内容, 实时的进行保存, 以免数据丢失, 当用户再次进入本系统后, 可继续上一次编辑的内容。

3.4 Cookie 和 Session

什么是 Cookie? 什么是 Session? 两者的区别和联系有哪些? Node.js 是否提供相应的模块来管理存储 Session? 如果没有提供相应的 Session 模块, 我们应该如何实现一个类似 Session 管理的模块呢? 以上几个问题都是本节需要解决的问题。

3.4.1 Cookie 和 Session

Session 和 Cookie 都是基于 Web 服务器的, 不同的是 Cookie 存储在客户端, 而 Session 存储在服务器端。

当用户在浏览网站的时候，Web 服务器会在浏览器上存储一些当前用户的相关信息，而在本地 Web 客户端存储的就是 Cookie 数据。当下次用户再浏览同一个网站时，Web 服务器就会先查看并读取本地的 Cookie 资料，如果有 Cookie 就会依据 Cookie 里的内容以及判断其过期时间，来给用户特殊的数据返回。

Cookie 的使用很普遍，许多提供个性化服务的网站，都是利用 Cookie 来辨认使用者，以方便送出为使用者量身定做的内容，像 Web 接口的免费 Email 网站，都要用到 Cookie。例如，有很多网站可以记住密码，一个星期或者一天都不用登录，这些登录信息都是存储在 Cookie 中的。

具体来说，Cookie 机制采用的是在客户端保持状态的方案，而 Session 机制采用的是在服务器端保持状态的方案。同时我们也看到，由于采用服务器端保持状态的方案在客户端也需要保存一个标识，所以 Session 机制可能需要借助于 Cookie 机制来达到保存标识的目的，但实际上它还有其他选择。正统的 Cookie 分发是通过扩展 HTTP 协议来实现的，服务器通过在 HTTP 的响应头中加上一行特殊的指示，以提示浏览器按照指示生成相应的 Cookie。然而，纯粹的客户端脚本如 JavaScript 或者 VBScript 也可以生成 Cookie。而 Cookie 的使用是由浏览器按照一定的原则在后台自动发送给服务器的。浏览器检查所有存储的 Cookie，如果某个 Cookie 所声明的作用范围大于等于将要请求的资源所在的位置，则把该 Cookie 附在请求资源的 HTTP 请求头上发送给服务器。

Cookie 的内容主要包括：名字、值、过期时间、路径和域。路径与域一起构成 Cookie 的作用范围。若不设置过期时间，则表示这个 Cookie 的生命期为浏览器会话期间，关闭浏览器窗口，Cookie 就消失。这种生命期为浏览器会话期的 Cookie，被称为会话 Cookie。会话 Cookie 一般不存储在硬盘上，而是保存在内存里，当然这种行为并不是规范的。若设置了过期时间，浏览器就会把 Cookie 保存到硬盘上，关闭后再次打开浏览器，这些 Cookie 仍然有效，直到超过设定的过期时间。存储在硬盘上的 Cookie 可以在不同的浏览器进程间共享，比如两个 IE 窗口。而对于保存在内存里的 Cookie，不同的浏览器有不同的处理方式。Session 机制是一种服务器端的机制，服务器使用一种类似于散列表的结构（也可能使用散列表）来保存信息。当程序需要为某个客户端的请求创建一个 Session 时，服务器首先检查这个客户端的请求里是否已包含了一个 Session 标识（称为 Session id），如果已包含则说明以前已经为此客户端创建过 Session，服务器就按照 Session id 把这个 Session 检索出来使用（检索不到，会新建一个），如果客户端请求不包含 Session id，则为此客户端创建一个 Session 并且生成一个与此 Session 相关联的 Session id，Session id 的值应该是一个既不会重复，又不容易被发现其生成规律的字符串，这个 Session id 将在本次响应中被返回给客户端保存。保存这个 Session id 的方式可以采用 Cookie，这样在交互过程中浏览器可以自动按照规则把这个标识发送给服务器。一般这个 Cookie 的名字都类似于 SEESIONID。

3.4.2 Session 模块实现

PHP 中提供了内置的 Session 方法，例如 `session_start` 以及 `$_SESSION` 等，而 Node.js 中没有提供任何 Session 管理模块，因此需要自己去实现，这里我们介绍一个他人实现的 Session 模块。

根据 Cookie 和 Session 的介绍，我们可以将该模块实现的逻辑转化为如图 3-39 所示的流程图。



图 3-39 Session 产生过程

如上图所示，客户端首先会请求 Session，而当服务端检查客户端中的 Cookie 没有相应的 Session id 时，会通过一定方式为其生成一个新的 Session id，而如果 Cookie 中存在该 Session id 并且没有过期时，则直接返回 Session 数据。

那么根据如上流程示意图以及介绍，可以为我们需要实现的模块先创建 3 种方法，分别是 start、newSession 和 cleanSessions。start 方法主要是启动 Session 管理，newSession 主要是为客户端创建一个新的 Session id，而 cleanSessions 则是清除 Session 数据。

本模块应用一个 Session 数组来存储系统所有的 Session，当有 Session id 存在时，无需新建 Session id，而是直接读取返回 Session 数据；而当 Session id 不存在时，需要创建 Session id，并且将 Session id 存储在该客户端的 Cookie 中，相关实现 start 代码如下：

```

var start = function(res, req) {
  var conn = { res: res, req: req };
  var cookies = {};

  if(typeof conn.req.headers.cookie !== "undefined") { // 判断是否存在 cookie
    conn.req.headers.cookie.split(';').forEach(function( cookie ) {
      var parts = cookie.split('=');
      cookies[ parts[ 0 ].trim() ] = ( parts[ 1 ] || '' ).trim();
    }); // Grab all cookies, and parse them into properties of the cookies object
  } else {
    cookies.SESSION = 0;
  }

  var SESSION = cookies.SESSION; //Get current SESSION
  if(typeof sessions[SESSION] !== "undefined") { // 判断是否存在 session
    session = sessions[SESSION];
    if(session.expires < Date()) { //If session is expired
      delete sessions[SESSION];
      return newSession(conn.res);
    } else {
      var dt = new Date();
      dt.setMinutes(dt.getMinutes() + 30);

      session.expires = dt; //Reset session expiration
      return sessions[SESSION];
    }
  } else {
    return newSession(conn.res);
  }
};

```

【代码说明】

- ❑ `typeof conn.req.headers.cookie`: 判断 `headers` 中是否存在 `Cookie`;
- ❑ `conn.req.headers.cookie.split(';')`: `Session` 存在时, 对 `Session` 进行解析, 获取其中的 `session id`;
- ❑ `cookies.SESSIONID = 0`: `headers` 中不存在 `Cookie` 时, 设置 `Session id` 为 0;
- ❑ `typeof sessions[SESSIONID] !== "undefined"`: 如果服务器存在该 `Session` 值时, 再验证 `Session` 是否过期, 过期则重新生成, 不过期时则为该 `Session` 延长过期 30 分钟时长;
- ❑ `return newSession(conn.res)`: `Session` 过期或者不存在时, 则新生成一个 `Session`。

以上就是一个 `Session` 简单的校验过程, 主要思路就是通过 `req` 对象中的 `headers` 获取 `Cookie`, 并对 `Cookie` 进行解析获取 `Session id`, 判断是否存在该 `Session id` 值, 从而返回或者生成新的 `Session`。下面我们来看一下 `newSession` 方法的实现, 代码如下:

```
function newSession(res) {

    var chars = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";
    var SESSIONID = '';
    for (var i = 0; i < 40; i++) {
        var rnum = Math.floor(Math.random() * chars.length);
        SESSIONID += chars.substring(rnum, rnum+1);
    } //Generate a 40 char random string for Session id

    if(typeof sessions[SESSIONID] !== "undefined") {
        return newSession(res); //Avoid duplicate sessions
    }

    var dt = new Date();
    dt.setMinutes(dt.getMinutes() + 30);

    var session = { //Session literal object
        SESSIONID: SESSIONID,
        expires: dt
    };
    sessions[SESSIONID] = session; //Store it for future requests

    res.setHeader('Set-Cookie', 'SESSIONID=' + SESSIONID);

    return session;
}
```

【代码说明】

- ❑ `var chars`: 获取 26 个字母以及 10 个数字字符数据, 用于生成随机字符;
- ❑ `for (var i = 0; i < 40; i++)...`: `for` 循环产生一个字符长度为 40 的 `SESSIONID`;
- ❑ `typeof sessions[SESSIONID] !== "undefined"`: 二次验证 `SESSIONID`, 避免产生相同的 `SESSIONID`;
- ❑ `dt.setMinutes(dt.getMinutes() + 30)`: 设置 `SESSIONID` 过期时间;
- ❑ `res.setHeader('Set-Cookie', 'SESSIONID=' + SESSIONID)`: 为客户端新增 `Cookie` 数据, 在

客户端 Cookie 中保存 SESSID。

上面的 SESSID 生成的方法是使用 36 个字符产生一个字符长度为 40 的 SESSID, 根据字符以及字符长度, 系统可以产生 40 的 36 次方种组合, 这样是足够系统使用了。但为了避免随机产生同一个 SESSID 数据, 我们还做了二次校验, 如果存在重复 SESSID, 就重新生成一次 SESSID。

介绍完 SESSID 校验和创建后, 我们再来看一下 cleanSession 清除那些过期的 Session 的函数逻辑实现。代码的主要思路是循环判断 Sessions 中的 SESSID 过期时间, 如果 SESSID 则将其从 Sessions 数组中删除, 代码如下:

```
function cleanSessions() {
  for(sess in sessions) {
    if(sess.expires < Date()) { //If expired
      delete sessions[session.SESSID];
    }
  }
}
exports.start = start;
```

上面就是简单的 Session 实现过程, 虽然没有其他框架的 Session 功能完善, 但已可以满足项目开发。当然读者可以将上面的代码修改, 使其更加完善, 例如其中的 SESSID 产生过程, 以及 Sessions 存储方式。上面只是通过一个数组变量, 读者可以将 Session 应用文件存储, 或者将其存储到 memcache 和 redis 缓存中。

3.4.3 Session 模块的应用

上面介绍到 Session 模块的实现, 那么接下来我们看一下该 Session 模块的应用方法。在应用该模块时, 用户一般习惯会在入口文件 (例如 app.js) 中 require 该模块, 并在 HTTP 的 createServer 函数中调用 session.start, 并将 session.start 返回的对象作为一个全局对象存储, 代码如下:

```
var app = http.createServer(function(req, res) {
  global.sessionLib = lib.session.start(res, req);
});

//调用方法
if(!sessionLib['username']){
  sessionLib['username'] = 'danhuang';
}
```

如上代码所示, 我们已经将 session.start 返回的 Session 对象作为一个全局变量存储, 因此在任何逻辑处理地方, 只要其需要 Session 处理, 就可以使用该全局对象来保存 Session 值。如上实例代码就是应用该全局对象, 来判断该 Session 对象中是否有 username 值。

以上就是一个 Session 模块的实现, 在 3.7 节中我们会应用该模块实现直播系统中用户登录的功能, 读者可以在 3.7 节中进一步学习该模块的应用。学习完本节希望大家对 Session 和 Cookie 有一个简单的了解, 并明白在 Node.js 中如何实现一个简单的 Session 模块来存储和管理系统用户的登录态, 以及会应用本节实现的 Session 模块。

3.5 Crypto 模块加密

本节将介绍 Node.js 的加密实现方法，通过本节的学习读者需要掌握基本的加密方法，了解数据入库前加密的重要性，了解 crypto 模块。本节最后会教读者开发出一个加密模块，并支持多种加密算法。

3.5.1 Crypto 介绍

加密模块需要底层系统提供 OpenSSL 的支持。它提供了一种安全凭证的封装方式，可以用于 HTTPS 安全网络以及普通 HTTP 连接。该模块还提供了一套针对 OpenSSL 的 hash（哈希）、hmac（密钥哈希）、cipher（编码）、decipher（解码）、sign（签名），以及 verify（验证）等方法的封装。

CNode 社区 snoopy 发表过一篇《浅谈 Node.js 中的 Crypto 模块》¹关于 Crypto 加密模块的介绍很详细。本节不会深入介绍，主要是在应用层面上介绍如何利用该模块加密。

下面介绍几个主要的加密算法的应用。

1. 哈希

使用 crypto.createHash() 方法可以得到哈希的实例，提供的算法实现包括 md5、sha1、sha256、sha512、ripemd160。在下面的例子中，hash.update() 加密字符串，使用 hash.digest() 输出字符串。

```
/* hash.js */
/* 获取 Node.js 的原生模块 crypto */
var crypto = require('crypto');

/* 调用 crypto 模块的 hash 编码 */
var hash = crypto.createHash("md5");

/* 应用 hash 编码方式实现加密 */
hash.update(new Buffer("huangdanhua", "binary"));
var encode = hash.digest('hex');

console.log(encode);
```

【代码说明】

- ❑ crypto.createHash("md5"): 使用 md5 进行加密，这里还可使用 sha1、sha256、sha512、ripemd160 方法进行加密。
- ❑ hash.update(new Buffer("huangdanhua", "binary")): 使用二进制数据流将 huangdanhua 字符串进行加密。
- ❑ encode = hash.digest('hex'): 返回 hash 对象加密后的字符串。

代码中使用到二进制数据作为参数，同样也可直接使用需要加密的字符串，例如：

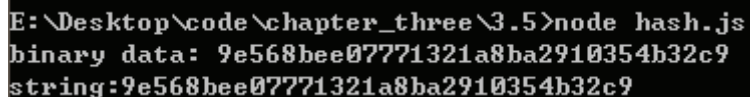
1 参见网站 <http://cnodejs.org/topic/504061d7fef591855112bab5>。


```
hash.update("huangdanhua");
```

在 hash.js 文件中添加一个字符加密代码，然后对比两者加密的结果是否相同。

```
/*-----string md5-----*/
var hash = crypto.createHash("md5");
hash.update("huangdanhua");
var encode = hash.digest('hex');
console.log("string:" + encode);
```

执行 hash.js 脚本，可以看到如图 3-40 所示的结果，两次的执行结果都是 9e568bee07771321a8ba2910354b32c9 加密字符串，因此两种传递方式都是可以的，但是在不同场景下要选择最合适的方式。



```
E:\Desktop\code\chapter_three\3.5>node hash.js
binary data: 9e568bee07771321a8ba2910354b32c9
string:9e568bee07771321a8ba2910354b32c9
```

图 3-40 加密字符返回结果

hash.digest 这个函数可以传入 3 个类型的参数 hex（十六进制）、binary（二进制）或者 base64 输出加密后的字符，默认参数是 binary。如果传递的参数非指定的这 3 个（hex、binary 和 base64）字符时，函数会自动使用 binary 返回加密字符串。

官网中介绍在调用 digest 方法后 hash 对象将不能再使用，也就是会清空，因此如果希望对一个字符串使用两种方式进行加密时，就必须重新创建 hash 对象，代码如下：

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

/*-----binary md5-----*/
var hash = crypto.createHash("md5");
hash.update(new Buffer("huangdanhua", "binary"));
var encode = hash.digest('hex');
console.log("binary data: " + encode);

/*-----string md5-----*/
var hash = crypto.createHash("md5");
hash.update("huangdanhua");
var encode = hash.digest('hex');
console.log("string:" + encode);
```

【代码说明】

- ❑ hash = crypto.createHash("md5"): 第一次 hash 对象调用 digest 后被清空，因此第二次加密时必须重新创建 hash 对象。

使用同样的方法使用 sha1 加密 huangdanhua 字符串，代码如下：

```
/*-----string sha1-----*/
var hash = crypto.createHash("sha1");
hash.update("huangdanhua");
var encode = hash.digest('hex');
console.log("string sha1:" + encode);
```

【代码说明】

- ❑ hash = crypto.createHash("sha1"): 设置加密方法为 sha1。

crypto.createHash 参数如果是非指定参数 ('sha1', 'md5', 'sha256', 'sha512') 时, 不会像 hash.digest 使用默认参数, 而会抛出 Node.js 异常, 并中断代码执行过程。因此在执行加密算法时, 最好将 crypto.createHash 加密参数作为一个常量。

2. HMAC

HMAC 是密钥相关的哈希运算消息认证码 (Hash-based Message Authentication Code), HMAC 运算利用哈希算法, 以一个密钥和一个消息为输入, 生成一个消息摘要作为输出¹。这个算法的实现这里我们就不去介绍了, 感兴趣的同学可以通过互联网在维基百科和百度百科学习。

crypto.createHmac(algorithm, key) 创建并返回一个 hmac 对象, 它是一个指定算法和密钥的加密 hmac。参数 algorithm 可选择 OpenSSL 支持的算法, 和 createHash 中的方法是相同的。参数 key 为 hmac 所使用的密钥, 可以为任意字符串, 代码如下:

```
/*-----binary md5-----*/

/* 调用 crypto 模块的 Hmac 编码 */
var hmac = crypto.createHmac("md5", 'dan');

/* 应用 hash 编码方式实现加密 */
hmac.update(new Buffer("huangdanhua", "binary"));
var encode = hmac.digest('hex');

console.log("binary data: " + encode);
```

【代码说明】

- ❑ hmac = crypto.createHmac("md5", 'dan'): 使用 dan 字符串作为私密进行 md5 的加密。
- ❑ encode = hmac.digest('hex'): 使用十六进制输出加密字符串, 这里同样可使用 3 种类型输出加密后的字符, 分别是 hex、binary 和 base64。
- ❑ hmac.update(new Buffer("huangdanhua", "binary")): 使用二进制数据流, 传递加密字符数据。

crypto.createHmac 的加密算法同样有 sha1、md5、sha256、sha512。hmac 调用 digest 函数之后和 hash 一样会清空 hash 对象数据, 因此再次进行加密时, 同样需要重新创建 hmac 对象, 如下代码:

```
/*-----string md5-----*/
var hmac = crypto.createHmac("md5", 'dan');
hmac.update("huangdanhua");
var encode = hmac.digest('hex');
console.log("string: " + encode);

/*-----string sha1-----*/
var hmac = crypto.createHmac("sha1", 'dan');
hmac.update("huangdanhua");
var encode = hmac.digest('hex');
console.log("string sha1: " + encode);
```

1 参见网站 <http://baike.baidu.com/view/1136366.htm>。

【代码说明】

- ❑ `hmac.update("huangdanhua")`: 使用字符传递加密数据。
- ❑ `hmac = crypto.createHmac("sha1", 'dan')`: 使用 `dan` 字符串作为密钥, 应用 `sha1` 加密算法进行加密。

这部分的加密算法和 `hash` 的方法是类似的, 只是在加密时添加了一个密钥, 确保加密字符的安全, 特别是现在对一些加密算法进行解密过程, 如 `md5` 的反加密。代码中使用简单字符 `dan` 作为密钥, 而实际应用中可以用其他更复杂的密钥来加密。

3. Cipher 和 Decipher

`Cipher` 函数中的参数 `algorithm` 可选择 OpenSSL 支持的算法的值, 例如 `'aes192'` 等。在最近的发行版中, `OpenSSL list-cipher-algorithms` 会显示可用的加密的算法。`Decipher` 顾名思义是指 `Cipher` 解密。代码如下:

```
/* cipher.js */
var crypto = require('crypto')
  , key = 'salt_from'
  , plaintext = 'danhua'
  , cipher = crypto.createCipher('aes-256-cbc', key) // 获取 Cipher 加密对象
  , decipher = crypto.createDecipher('aes-256-cbc', key);
                                     // 获取 Decipher 解密对象

cipher.update(plaintext, 'utf8', 'hex');
var encryptedPassword = cipher.final('hex');

decipher.update(encryptedPassword, 'hex', 'utf8');
var decryptedPassword = decipher.final('utf8');

console.log('encrypted :', encryptedPassword);
console.log('decrypted :', decryptedPassword);
```

【代码说明】

- ❑ `cipher = crypto.createCipher('aes-256-cbc', key)`: 创建一个 `cipher` 加密对象, 其中第一个参数是算法类型, 第二个是需要加密的私钥。
- ❑ `decipher = crypto.createDecipher('aes-256-cbc', key)`: 创建一个 `decipher` 解密对象, 其中第一个参数是算法类型, 第二个是需要解密的私钥。
- ❑ `cipher.update(plaintext, 'utf8', 'hex')`: 使用参数 `data` 更新要加密的内容, 其编码方式由参数 `input_encoding` 指定, 可以为 `'utf8'`、`'ascii'` 或者 `'binary'`。参数 `output_encoding` 指定了已加密内容的输出编码方式, 可以为 `'binary'`、`'base64'` 或 `'hex'`。
- ❑ `encryptedPassword = cipher.final('hex')`: 返回所有剩余的加密内容, `output_encoding` 输出编码为 `'binary'`、`'ascii'` 或 `'utf8'`。
- ❑ `decipher.update(encryptedPassword, 'hex', 'utf8')`: 类似 `decipher.update(encryptedPassword, 'hex', 'utf8')`。
- ❑ `decryptedPassword = decipher.final('utf8')`: 类似 `encryptedPassword = cipher.final('hex')`。

如果希望对一个字符进行加密和反加密时, 必须保证 `cipher` 对象和 `decipher` 对象加密的私钥和加密算法是相同的, 才能正确的解密。解密和加密调用的所有函数都是类似的,

只是在输出方式上一个使用十六进制，另一个使用 utf8。

运行 cipher.js 脚本，返回如图 3-41 所示的结果，加密结果字符串，以及反加密后的字符为 danhuang，解密后的字符和加密字符是一致的。

```
E:\Desktop\code\chapter_three\3.5>node cipher.js
encrypted : 0317e27f128a568a074bf431f958673e
decrypted : danhuang
```

图 3-41 cipher 加密返回结果

其他还有 signer、verify 和 diffieHellman 等加密算法，大家可以学习和了解其他几种加密方法，同时区分各种加密算法之间的区别。

两种加密算法应用区别主要是是否可逆加密。本节介绍的前两种加密算法是不可逆加密，可以应用到系统用户登录或者其他检验上。可逆性加密主要是用在数据的存储上，例如服务器密码、用户关键数据等。

3.5.2 Web 数据密码的安全

初学者可能对密码和数据的安全性要求不是太高。在互联网发展的现阶段，也许会因为一个小小的事情导致数据泄漏，如果没有对重要的用户数据进行加密，我们可能会丢失很多用户。

Web 应用中应注重如下数据的加密：用户密码、私人数据（不公开文章、邮件、个性隐私数据等）、用户的统计数据等。

这里面最重要的当然是用户密码，对于用户密码可使用不可逆加密方法，而其他一些私人数据和统计数据则可使用相应的可逆加密方法。

如图 3-42 所示为上面所讲的数据加密相关的信息，其中校验性数据加密算法使用不可逆加密，即使数据库泄漏也不会给用户造成其他影响。储存性数据，主要是因为其需要查看原有数据内容，因此需要应用可逆性加密。

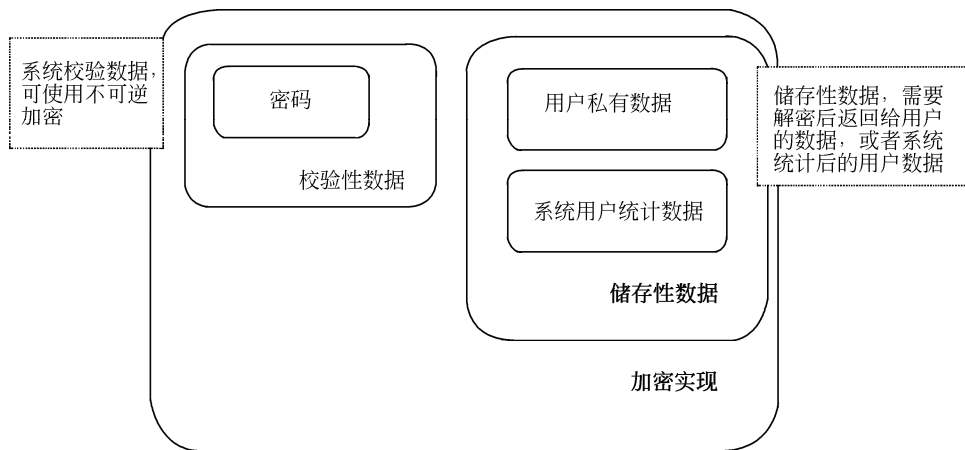


图 3-42 数据加密分析

本节的目的是希望读者在项目开发中能够把加密作为一种编码习惯，让项目更加健全和完善，同时可以保护互联网用户的个人隐私等。

3.5.3 简单加密模块设计

利用本章的知识点，实现一个加密模块。该模块输入为：加密算法、加密字符、加密类型、加密返回字符编码、加密私钥即可。在实现加密算法，同时需要实现对于特定加密算法的解密函数 API。

模块设计思想如下所述。

- ❑ 提供 API: encode 和 decode 接口。
- ❑ encode 参数：加密类型（hmac 或者 hash 等）、加密算法（openssl 支持的算法）、加密字符（二进制数据流或者字符串）、加密后返回的字符编码（3 种类型）。
- ❑ decode 参数：解密类型、解密私钥、解密字符、解密返回字符编码。

输出格式统一为字符串格式返回。根据上面的分析，我们将会应用到一个在第 2 章中介绍的设计模式（适配器），因为该模块只提供了一个加密外部调用接口，该外部接口要满足多种加密类型。类似于一个插座需要满足多个类型的插头。第 2 章中介绍的该设计模式是将一个类的接口转换成客户希望的另外一个接口，Adapter 模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。

如图 3-43 所示是该模块的文件结构。adapter.js 为 adapter 类，encode_module.js 为加密 NPM 模块，target 为 adapter 的父类，test.js 为测试 Node.js 脚本。在 adaptee_class 文件夹中，为相应的实际操作加密类，如图 3-44 所示只提供了 3 个加密类。

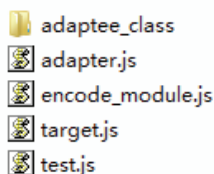


图 3-43 Adapter 文件结构



图 3-44 adaptee_class 文件目录结构

cipher.js、hash.js 和 hmac.js 分别对应 Node.js 中的 3 种加密类型。首先需要创建一个 encode_module.js 作为外部调用模块，根据上面的分析，其中拥有两个 exports 方法：encode 和 decode。其中，encode 代码如下：

```
/* encode_module.js */
var AdapterClass = require('./adapter');
exports.encode = function(){
    var encodeModule = arguments[0] ? arguments[0] : null
    , algorithm      = arguments[1] ? arguments[1] : null
    , enstring       = arguments[2] ? arguments[2] : ''
    , returnType     = arguments[3] ? arguments[3] : ''
    , encodeKey      = arguments[4] ? arguments[4] : ''
    , encodeType     = arguments[5] ? arguments[5] : '';
    var Adapter      = new AdapterClass();
    return Adapter.encode(encodeModule, algorithm, enstring, returnType,
        encodeKey, encodeType);
}
```

```
};
```

【代码说明】

- ❑ encodeModule = arguments[0] ? arguments[0] : null: 获取参数中的加密模块名。
- ❑ var Adapter = new AdapterClass(): 创建 Adapter 类。
- ❑ Adapter.encode(): 调用 adapter 中的 encode 方法。

其中, 使用 arguments 来获取函数参数的目的是希望该接口参数是可变的。每个参数的作用如下所述。

- ❑ encodeModule: 为模块名, 例如使用 hash 加密时, 该参数为 'hash' 字符串。
- ❑ algorithm: 为算法类型, 例如 'sha1'、'md5'、'sha256'、'sha512' 等。
- ❑ enstring: 需要加密的字符串或者字符的二进制数据流。
- ❑ returnType: 输出返回类型, 例如 'hex', 十六进制返回输出。
- ❑ encodeKey: 加密使用的私钥, 为可选参数。
- ❑ encodeType: 加密时需要的加密编码, 可以为 'binary', 'ascii' or 'utf8'。

加密方法都是通过 adapter 调用, 这就是适配器的作用。Decode 方法和 encode 方法大致相同, 代码如下:

```
exports.decode = function(){
    var encodeModule = arguments[0] ? arguments[0] : null
    , algorithm      = arguments[1] ? arguments[1] : null
    , enstring       = arguments[2] ? arguments[2] : ''
    , returnType     = arguments[3] ? arguments[3] : ''
    , decodeKey      = arguments[4] ? arguments[4] : ''
    , encodeType     = arguments[5] ? arguments[5] : '';
    var Adapter      = new AdapterClass();
    return Adapter.decode(encodeModule, algorithm, enstring, returnType,
        decodeKey, encodeType);
}
```

既然所有的接口都是通过 adapter 来调用的, 那么 adapter 必须能够适合所有的加密算法调用方法。在看 adapter 之前, 我们先了解其父类 target 的实现。代码如下:

```
/* target.js */
module.exports = function(){
    this.encode = function(){
        // for noting
        console.log('no this function exist');
    }
    this.decode = function(){
        // for noting
        console.log('no this function exist');
    }
}
```

代码很简单, 没有提供任何方法实现, 只是创建了两个方法: encode 和 decode 方法。这样其继承类 adapter 就可以获取这两个方法, 并需要重写方法。利用第 2 章的知识首先使用 util 来实现继承。

```
/* adapter.js */
var util = require("util");
var Target = require('./target');
```

```
function Adapter(){
    Target.call(this);
}
util.inherits(Adapter, Target);
module.exports = Adapter;
```

【代码说明】

- ❑ `util = require("util")`: 获取 Node.js 的模块 `util`。
- ❑ `Target = require('./target')`: 获取父类。
- ❑ `util.inherits(Adapter, Target)`: 实现继承。

上面代码没有提供 `encode` 和 `decode` 的方法实现，因此我们将上面的代码补全来实现父类的 `encode` 和 `decode` 方法，代码如下：

```
function Adapter(){
    Target.call(this);
    this.encode = function(){
        var encodeModule = arguments[0] ? arguments[0] : null
        , algorithm      = arguments[1] ? arguments[1] : null
        , enstring       = arguments[2] ? arguments[2] : ''
        , returnType     = arguments[3] ? arguments[3] : ''
        , encodeKey      = arguments[4] ? arguments[4] : ''
        , encodeType     = arguments[5] ? arguments[5] : ''
        , AdapteeClass = require("./adaptee_class/" + encodeModule)
        , adapteeObj = new AdapteeClass();
        return adapteeObj.encode(algorithm, enstring, returnType, encodeKey,
            encodeType);
    }
    this.decode = function(){
        var encodeModule = arguments[0] ? arguments[0] : null
        , algorithm      = arguments[1] ? arguments[1] : null
        , enstring       = arguments[2] ? arguments[2] : ''
        , returnType     = arguments[3] ? arguments[3] : ''
        , encodeKey      = arguments[4] ? arguments[4] : ''
        , encodeType     = arguments[5] ? arguments[5] : ''
        , AdapteeClass = require("./adaptee_class/" + encodeModule)
        , adapteeObj = new AdapteeClass();
        return adapteeObj.decode(algorithm, enstring, returnType, encodeKey,
            encodeType);
    }
}
```

【代码说明】

- ❑ `encodeModule = arguments[0] ? arguments[0] : null`: 获取相应的参数数据。
- ❑ `AdapteeClass = require("./adaptee_class/" + encodeModule)`: 根据不同的加密类型，获取相应的加密模块，这里建议大家使用绝对路径方式获取模块。
- ❑ `adapteeObj = new AdapteeClass()`: 创建加密模块对象。
- ❑ `return adapteeObj.encode()`: 调用加密模块对象的 `encode` 方法。

如果大家认真地看过本书第 2 章的 Node.js 设计模式，应该可以明白为什么这里可以 `new` 一个 `require` 返回的数据。主要原因在于该模块使用 `module.exports` 返回一个 `function` 数据类型，而不是一个 `json` 对象。`decode` 方法的实现和 `encode` 方法是相同的，因此上面的代码中，没有添加任何注释。

最后我们来看每一个 `adaptee` 是如何实现加密方法的。这里举例 `hash` 加密模块 `hash.js`,

代码如下：

```
/* hash.js */
var crypto = require('crypto');
module.exports = function () {
  this.encode = function () {
    var algorithm = arguments[0] ? arguments[0] : null
      , enstring = arguments[1] ? arguments[1] : ''
      , returnType = arguments[2] ? arguments[2] : '';
    var hash = crypto.createHash(algorithm);
    hash.update(enstring);
    return hash.digest(returnType);
  }
  this.decode = function () {
    console.log('it has not decode function');
  }
}
```

【代码说明】

- ❑ `crypto = require('crypto')`: 获取 Node.js 的 `crypto` 模块对象。
- ❑ `module.exports = function () {}`: `exports` 该模块的类。
- ❑ `this.encode = function () {}`: 具体 `encode` 方法实现。
- ❑ `algorithm = arguments[0] ? arguments[0] : null`: 相应参数获取。
- ❑ `hash = crypto.createHash(algorithm)`: 创建 `hash` 加密对象。
- ❑ `hash.update(enstring)`: 加密 `enstring` 字符。
- ❑ `return hash.digest(returnType)`: 以 `returnType` 类型返回加密字符串。

加密方法和 3.5.1 节介绍的加密实现相同，只需要将相应的参数替换为变量即可。创建一个 `test.js` 来验证该模块的可用性。代码如下：

```
var encodeModule = require('./encode_module');

/*encode with hash*/
console.log('-----encode with hash-----');
var hashEncodeStr = encodeModule.encode('hash', 'md5', 'danhuang', 'hex');
console.log(hashEncodeStr);

/*encode with hmac*/
console.log('-----encode with hmac-----');
var hmacEncodeStr = encodeModule.encode('hmac', 'md5', 'danhuang', 'hex', 'dan');
console.log(hmacEncodeStr);

/*encode with cipher*/
console.log('-----encode with cipher-----');
var cipherEncodeStr = encodeModule.encode('cipher', 'aes-256-cbc', 'danhuang', 'hex', 'salt_from', 'utf8');
console.log(cipherEncodeStr);

/*decode with decipher*/
console.log('-----decode with decipher-----');
var decipherEncodeStr = encodeModule.decode('cipher', 'aes-256-cbc', cipherEncodeStr, 'utf8', 'salt_from', 'hex');
console.log(decipherEncodeStr);
```


【代码说明】

- ❑ `encodeModule = require('./encode_module')`: 获取 `encode_module` 模块对象。
- ❑ `hashEncodeStr = encodeModule.encode('hash', 'md5', 'danhuang', 'hex')`: 使用 `hash` 的 `md5` 算法对 `danhuang` 进行字符加密。
- ❑ `hmacEncodeStr = encodeModule.encode('hmac', 'md5', 'danhuang', 'hex', 'dan')`: 使用 `hmac` 的 `md5` 算法并使用 `dan` 字符作为私钥, 对 `danhuang` 进行字符加密。
- ❑ `cipherEncodeStr = encodeModule.encode('cipher', 'aes-256-cbc', 'danhuang', 'hex', 'salt_from', 'utf8')`: 应用 `cipher` 进行 `danhuang` 字符加密。
- ❑ `decipherEncodeStr = encodeModule.decode('cipher', 'aes-256-cbc', cipherEncodeStr, 'utf8', 'salt_from', 'hex')`: 应用 `decipher` 解密之前应用 `cipher` 加密后的字符。

运行 `test.js`, 可看到输出结果如图 3-45 所示。

```
E:\Desktop\code\chapter_three\3.5\module>node test.js
-----encode with hash-----
3333d0fcf7a07189c70385ca12251e82
-----encode with hmac-----
68a96e3c96d5e2f22b88d8a0815c0571
-----encode with cipher-----
0317e27f128a568a074bf431f958673e
-----decode with decipher-----
danhuang
```

图 3-45 test 测试脚本运行返回结果

第一行输出为 `hash` 加密 `danhuang` 字符后的结果。第二行输出为使用 `hmac` 加密 `danhuang` 字符后的结果。第三行输出为 `cipher` 加密 `danhuang` 字符后的结果。最后一行则为 `decipher` 反加密输出的字符, 可以看到输出的是 `danhuang`, 和加密字符是相同的。

通过实践性应用创建了加密模块 `encode_module`, 这里涉及很多异常逻辑, 代码中没有体现出来。例如, `adapter` 中 `require` 一个加密模块时, 需要使用 `try catch` 进行 `require`, 避免不存在模块时, 代码中断退出, 其他的就是参数的验证和判断, 以及错误码输出。由于篇幅原因本书中没有提供代码的异常逻辑, 希望大家可以自己增加这些逻辑, 保证代码的健壮性。

本模块的实践开发应用到了第 2 章介绍的 `adapter` 设计模式, 从实现最终结果来看本模块的可扩展性非常好。当我们需要新增一个加密类型时, 只需要在 `adaptee_class` 文件夹中添加一个加密类, 然后在调用时带相应的模块名, 即可实现一种新的加密类型, 而无需改动其他代码。因此在代码设计上, 希望大家能够在代码的设计阶段多考虑模式的应用。

3.6 Node.js+Nginx

本节将介绍 `Nginx` 概述、`Nginx` 的安装配置, 以及 `Nginx` 与 `Node.js` 项目部署。通过对比来说明 `Nginx+Node.js` 项目的运行性能。

`Nginx` 概述中介绍 `Nginx` 起源、现在的发展和前景、`Nginx` 主要解决的问题, 以及哪些项目适合应用 `Nginx`, 最后介绍 `Nginx` 的现有功能。

在 Nginx 安装配置中详细描述 Nginx 的安装和配置过程,对其中的异常安装进行整理,帮助初学者减少配置安装时间。

本节最后将通过一个简单的项目来介绍如何结合 Nginx 部署 Node.js 项目,然后通过本项目来压力测试 Nginx+Node.js 的服务器性能。

3.6.1 Nginx 概述

Nginx (发音同 engine x) 是一个高性能的 HTTP 和反向代理服务器¹ (反向代理也就是通常所说的 Web 服务器加速,它是一种通过在繁忙的 Web 服务器和 Internet 之间增加一个高速的 Web 缓冲服务器来降低实际的 Web 服务器的负载),也是一个 IMAP/POP3/SMTP 代理服务器。

Nginx 是由 Igor Sysoev 为俄罗斯访问量第二的 Rambler.ru 站点开发的,它已经在该站点运行四年多了。Igor 将源代码以类 BSD 许可证的形式发布。自 Nginx 发布四年来, Nginx 已经因为它的稳定性、丰富的功能集、示例配置文件和低系统资源的消耗而闻名了。目前国内各大门户网站已经部署了 Nginx,如新浪、网易、腾讯等;国内几个重要的视频分享网站也部署了 Nginx,如六房间、酷 6 等。Nginx 技术在国内日趋火热,越来越多的网站开始部署 Nginx²。

那么为什么要选择 Nginx 呢?

Nginx 是一个高性能 Web 和反向代理服务器,在高连接并发的情况下,Nginx 是 Apache 服务器不错的替代品。它能够支持高达 50,000 个并发连接数的响应。

Nginx 作为负载均衡服务器既可以在内部直接支持 Rails 和 PHP 程序对外进行服务,也可以支持作为 HTTP 代理服务器对外进行服务。Nginx 采用 C 进行编写,不论是系统资源开销还是 CPU 使用效率都比 Perlbal 要好很多。

Nginx 同时也是一个非常优秀的邮件代理服务器。

Nginx 是一个非常简单、配置文件非常简洁 (还能够支持 perl 语法) 的服务器。Nginx 启动特别简单,并且几乎可以做到 7*24 小时不间断运行,即使运行数月也不需要重新启动。

Nginx 能够灵活地处理静态资源文件。

3.6.2 Nginx 的配置安装

Nginx 相对来说是一个较简单的使用工具,安装方法可以参考官网文档³。本节详细介绍安装过程,同时说明安装过程中会遇到的问题及解决的办法。注意,在这里只介绍 Linux 下的安装和配置方法。

1 反向代理 (Reverse Proxy) 方式是指以代理服务器来接受 internet 上的连接请求,然后将请求转发给内部网络上的服务器,并将从服务器上得到的结果返回给 internet 上请求连接的客户端,此时代理服务器对外就表现为一个服务器。

2 来自 Nginx 官方文档 <http://wiki.Nginx.org/NginxChs>。

3 参见网站 <http://wiki.Nginx.org/InstallChs>。

从官网下载最新版本的 Nginx 安装包¹，这里测试使用的版本是 Nginx 1.0.2。将其下载到 Linux 系统中，使用 tar 进行解压：

```
tar -zxvf Nginx-1.0.2.tar.gz
```

解压完成后进入 Nginx-1.0.2 文件夹，执行 configure 检查软件安装的系统环境，一般在这里会检测出软件安装缺少的依赖库。

```
./configure
```

在执行完以后，笔者遇到了一个问题，如下：

./configure: error: the HTTP rewrite module requires the PCRE library.You can either disable the module by using --without-http_rewrite_module option, or install the PCRE library into the system, or build the PCRE library statically from the source with Nginx by using --with-pcre=<path> option.

这个问题的主要原因是缺少 PCRE library 库，进入 PCRE 官网²下载最新版本 PCRE 库³，这里笔者选择的是 pcre-8.21 版本。

下载完成后上传到 Linux 系统文件夹中，使用 tar 解压文件：

```
tar -zxvf pcre-8.21.tar.gz
```

解压完成后使用 configure 检查软件安装的系统环境，检查完后使用 make 编译，利用 make install 安装 PCRE 依赖库，执行指令如下：

```
./configure  
make  
make install
```

可能这其中还涉及一些无法找到依赖库的问题，解决办法是前往该依赖库的官网下载相应版本的依赖库，进行编译安装。

成功安装所依赖的库后，使用 ./configure 再次检查，检查成功后会在本文件夹中生成 makefile 文件，然后利用 make 和 make install 安装 Nginx。

```
./configure  
make  
make install
```

以上就是整个安装过程，其中会涉及一些异常 error 获取 warning 导致安装不成功的情况下，大家可以将 error 信息直接拷贝到百度或者 google 中进行搜索找到解决方法。Nginx 成功安装后会在系统的 /usr/local/Nginx 文件中，其中的启动文件是 /usr/local/Nginx/sbin/Nginx。接下来介绍如何启动运行 Nginx。

1. 启动 Nginx 服务

Nginx 启动方式很简单，运行可执行文件 Nginx，方式如下：

-
- 1 参见网站 <http://wiki.Nginx.org/InstallChs>。
 - 2 参见网站 <http://www.pcre.org/>。
 - 3 参见网站 <ftp://ftp.csx.cam.ac.uk/pub/software/programming/pcre/>。

```
/usr/local/Nginx/sbin/Nginx
```

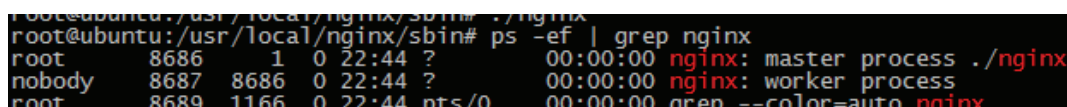
启动时，系统会响应一个错误，错误的原因主要是安装过程中没有关联 `prec library` 库的位置，错误提示是 `error while loading shared libraries: libpcre.so.0: cannot open shared object file: No such file or directory`。解决办法是创建一个软连接将 `PREC` 库和系统 `lib` 库进行关联，方法如下：

```
32 位执行 ln -s /usr/local/lib/libpcre.so.1 /lib
64 位执行 ln -s /usr/local/lib/libpcre.so.1 /lib64
```

由于笔者的 Linux 系统为 64 位，所以执行 `ln -s /usr/local/lib/libpcre.so.1 /lib64`，32 位系统执行上面代码中的第一行。执行完成后我们再启动 `Nginx` 服务，这时不会再出现错误提示。然后通过 `ps` 查看 `Nginx` 进程，检测是否已经成功运行 `Nginx`。

```
ps -ef | grep Nginx
```

运行后可以看到类似如图 3-46 所示的 `Nginx` 进程。



```
root@ubuntu:/usr/local/nginx/sbin# ./nginx
root@ubuntu:/usr/local/nginx/sbin# ps -ef | grep nginx
root      8686      1  0 22:44 ?        00:00:00 nginx: master process ./nginx
nobody    8687    8686  0 22:44 ?        00:00:00 nginx: worker process
root      8689   1166  0 22:44 pts/0    00:00:00 grep --color=auto nginx
```

图 3-46 nginx 进程查询结果

图 3-46 中的两个进程一个为 `master`，另外一个为 `worker`。因为 `Nginx` 是多进程运行的，因此运行中可以看到多个 `worker` 进程产生，而 `master` 进程主要是管理这些 `worker` 进程，包括接收来自外界的信号，向各 `worker` 进程发送信号，监控 `worker` 进程的运行状态，当 `worker` 进程退出后（异常情况下），会自动重新启动新的 `worker` 进程。至于 `worker` 之间的进程通信，以及更深入的关于 `Nginx` 的模型¹，本书就不再详细介绍。

`Nginx` 的一些帮助指令可以通过运行 `/usr/local/Nginx/sbin/Nginx -h` 进行查看。如下是返回的一些信息，可以看到启动 `Nginx` 的方式，以及 `Nginx` 版本信息等。

```
Nginx: Nginx version: Nginx/1.0.2
Nginx: Usage: Nginx [-?hvVtq] [-s signal] [-c filename] [-p prefix] [-g directives]

Options:
  -?, -h      : this help
  -v          : show version and exit
  -V          : show version and configure options then exit
  -t          : test configuration and exit
  -q          : suppress non-error messages during configuration testing
  -s signal    : send signal to a master process: stop, quit, reopen, reload
  -p prefix    : set prefix path (default: /usr/local/Nginx/)
  -c filename  : set configuration file (default: conf/nginx.conf)
  -g directives : set global directives out of configuration file
```

在上面的帮助信息中，`-v` 是查看当前 `Nginx` 的版本信息；`-V` 是查看当前 `Nginx` 版本信息，以及启动的关联配置文件；`-t` 是验证关联启动的配置文件是否正确配置，并且可正常

1 参见网站 http://tengine.taobao.org/book/chapter_2.html。

启动 Nginx; -q 提供查看配置文件测试; -s 可以控制 master 进程的 stop、quit、reopen、reload; -c 设置启动的配置文件; -p 设置代理路径; -g 设置全局指令的配置文件。

接下来介绍 Nginx 配置文件的一些参数的作用, 以及配置方法。

2. Nginx 配置文件介绍

下面将系统地介绍 Nginx 的一些配置参数, 目的是希望大家能够系统的了解这部分知识。

Nginx 有一个默认配置文件, 一般在/usr/local/Nginx/conf下, 文件名为 Nginx.conf。运行用户和属主:

```
user nobody nobody;
```

启动进程, 通常设置成和 CPU 的数量相等:

```
worker_processes 4;
```

全局错误日志及 PID 文件:

```
error_log /var/log/Nginx/error.log;
pid /var/run/Nginx.pid;
```

工作模式及连接数上限:

```
events {
    use epoll; #epoll 是多路复用 IO(I/O Multiplexing)中的一种方式,但是仅用于
linux2.6 以上内核,可以大大提高 Nginx 的性能
    worker_connections 1024;#单个后台 worker process 进程的最大并发连接数
    # multi_accept on;
}
```

设定 HTTP 服务器, 利用它的反向代理功能提供负载均衡支持。代码如下:

```
http {
    #设定 mime 类型,类型由 mime.type 文件定义
    include /etc/Nginx/mime.types;
    default_type application/octet-stream;
    #设定日志格式
    access_log /var/log/Nginx/access.log;
    #sendfile 指令指定 Nginx 是否调用 sendfile 函数 (zero copy 方式) 来输出文件,
    #对于普通应用,必须设为 on,如果用来进行下载等应用磁盘 IO 重负载应用,可设置为 off,
    #以平衡磁盘与网络 I/O 处理速度,降低系统的 uptime
    sendfile on;
    #tcp_nopush on;
    #连接超时时间
    #keepalive_timeout 0;
    keepalive_timeout 65;
    tcp_nodelay on;

    #开启 gzip 压缩
    gzip on;
    gzip_disable "MSIE [1-6]\.(?!.*SV1)";
    #设定请求缓冲
    client_header_buffer_size 1k;
    large_client_header_buffers 4 4k;
    include /etc/Nginx/conf.d/*.conf;
    include /etc/Nginx/sites-enabled/*;
    #设定负载均衡的服务器列表
```

```

upstream mysvr {
    #weight 参数表示权值，权值越高被分配到的几率越大
    #本机上的 Squid 开启 3128 端口
    server 192.168.8.1:3128 weight=5;
    server 192.168.8.2:80 weight=1;
    server 192.168.8.3:80 weight=6;
}
server {
    #侦听 80 端口
    listen 80;
    #定义使用 www.xx.com 访问
    server_name www.xx.com;

    #设定本虚拟主机的访问日志
    access_log logs/www.xx.com.access.log main;

    #默认请求
    location / {
        root /root; #定义服务器的默认网站根目录位置
        index index.php index.html index.htm; #定义首页索引文件的名称

        fastcgi_pass www.xx.com;
        fastcgi_param SCRIPT_FILENAME $document_root/$fastcgi_script_name;
        include /etc/Nginx/fastcgi_params;
    }

    # 定义错误提示页面
    error_page 500 502 503 504 /50x.html;
    location = /50x.html {
        root /root;
    }

    #静态文件，Nginx 自己处理
    location ~ ^/(images|javascript|js|css|flash|media|static)/ {
        root /var/www/virtual/htdocs;
        #过期 30 天，静态文件不怎么更新，过期可以设大一点，如果频繁更新，则可以设置得小一点
        expires 30d;
    }

    #PHP 脚本请求全部转发到 FastCGI 处理，使用 FastCGI 默认配置
    location ~ \.php$ {
        root /root;
        fastcgi_pass 127.0.0.1:9000;
        fastcgi_index index.php;
        fastcgi_param SCRIPT_FILENAME /home/www/www$fastcgi_script_name;
        include fastcgi_params;
    }

    #设定查看 Nginx 状态的地址
    location /NginxStatus {
        stub_status on;
        access_log on;
        auth_basic "NginxStatus";
        auth_basic_user_file conf/htpasswd;
    }

    #禁止访问 .htxxx 文件
    location ~ /\.ht {
        deny all;
    }
}

```

3.6.3 如何构建

下面将结合实例介绍，如何应用 Nginx 和 Node.js 构建简单的项目。首先我们从基本的实例——hello world 来介绍。

```
/* hello.js */
var http = require('http');

http.createServer(function (request, response) {
  response.writeHead(200, {'Content-Type': 'text/plain'});
  response.end('hello world\n');
}).listen(8000);
```

相信大家对以上代码都很熟悉，主要是监听本地端口 8000 创建 HTTP 服务器，成功访问后输出“hello world”字符。

接下来我们需要在 Nginx 中添加一个监听 server 来转发 url 请求。编辑 Nginx 的配置文件 Nginx.conf。

```
vi /usr/local/Nginx/conf/Nginx.conf
```

在 http 大括号内添加一个 server 信息，其 server 信息如下所示，监听启动端口 80。

```
server {
    listen      80;
    server_name nodejs.danhuang.com;

    #charset koi8-r;

    #access_log logs/host.access.log main;

    location / {
        proxy_pass http://127.0.0.1:8000;
        #root    html;
        #index   index.html index.htm;
    }
}
```

【代码说明】

- ❑ listen 8080: 该 server 监听 80 端口。
- ❑ server_name nodejs.danhuang.com: 使用域名为 nodejs.danhuang.com。
- ❑ proxy_pass http://127.0.0.1:8000: 设置转发的请求 url。

重启 Nginx 服务器，并运行最初创建的 hello.js。运行成功后，在本地修改 hosts，将 nodejs.danhuang.com 指向 127.0.0.1。由于这里使用的是 192.168.1.120 服务器，因此修改本地 Windows 下的 hosts（路径在 C:\Windows\System32\drivers\etc），添加如何一行代码：

```
192.168.1.120    nodejs.danhuang.com
```

成功添加后，打开浏览器输入 nodejs.danhuang.com，可以看到如图 3-47 所示的效果，输出了一个 hello world 字符串，这样就表明我们已经成功构建了 Node.js 与 Nginx 服务器。

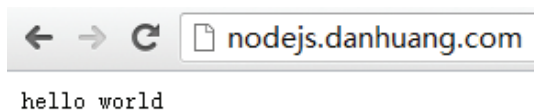


图 3-47 nodejs.danhuang.com 页面响应

之前涉及静态文件时，我们是通过一个静态模块去处理所有的静态文件，而现在可以通过 Nginx 来配置静态文件路径，使用 Nginx 来管理所有的静态文件。同时之前我们一直遇到的一个请求/favicon.ico，也可以通过 Nginx 进行设置。配置文件修改如下：

```
server {
    listen      80;
    server_name nodejs.danhuang.com;

    location / {
        proxy_pass http://127.0.0.1:8000;
    }

    location ~* ^.+\. (css|js|txt|xml|swf|wav)$ {
        root      /home/danhuang/helloworld/static;
        access_log off;
        expires    24h;
    }
    ## Serve an empty 1x1 gif _OR_ an error 204 (No Content) for favicon.ico
    location ~ (favicon.ico) {
        break;
    }
}
```

【代码说明】

- ❑ /home/danhuang/helloworld/static: 设置静态文件存储路径。
- ❑ location ~(favicon.ico): Nginx 单独处理/favicon.ico 请求。
- ❑ css|js|txt|xml|swf|wav: 这里正则可以匹配任何其他静态文件。

上面的配置中设定了静态文件路径，因此我们必须将所有静态文件放在该路径下。其中上面所涉及的静态文件类型只有 css、js、txt、xml、swf 和 wav 类型，如果需要其他类型的话可以直接在其中添加。修改配置成功后，我们重新启动 Nginx。

在 hello.js 文件夹中创建一个静态文件夹 static，用来存储所有的静态文件。创建一个 static 文件夹并在其中创建一个 style.css，代码如下：

```
#main_content {
color: red;
}
```

在 helloworld 文件夹中创建 index.html，显示一个 hello world 信息，并加载一个 css 文件，代码如下：

```
<html>
  <head>
    <title>Test for Nginx</title>
    <meta http-equiv="content-type" content="text/html; charset=utf-8">
    <link rel="stylesheet" href="style.css">
```



```

</head>
<body>
  <div id='main_content'>
    Hello world!
  </div>
</body>
</html>

```

因为在 Nginx 中已经指定静态文件的存储位置，因此这里的静态文件请求方式可以直接使用文件名作为 HTTP 请求，代码如下：

```
<link rel="stylesheet" href="style.css">
```

而无需添加相应的路径名 static。如果添加相应路径反而会出现 404 not find 的情况。将 hello.js 修改为读取一个 index.html 文件信息。

```

/* hello.js */
var http = require('http'),
    fs    = require('fs'),
    url   = require('url');
http.createServer(function(req, res) {
  /* 获取当前 index.html 的路径 */
  var readPath = __dirname + '/' + url.parse('index.html').pathname;
  var indexPage = fs.readFileSync(readPath);
  res.writeHead(200, { 'Content-Type': 'text/html' });
  res.end(indexPage);
}).listen(8000);
console.log('Server running at http://localhost:8000/');

```

对于上面这段代码大家也是非常熟悉的，使用 HTTP 模块创建 HTTP 服务器，成功访问返回一个 html 类型的数据到客户端。

运行 hello.js，可以看到如图 3-48 所示的第一次请求返回的结果（可以使用快捷键 Ctrl+F5 强制请求服务器资源，不使用本地缓存），成功获取到了静态文件 style.css。按照之前我们自己设计的静态文件管理模块，其必须还能够做一些缓存机制，避免多次读取文件信息，增加服务器 IO。

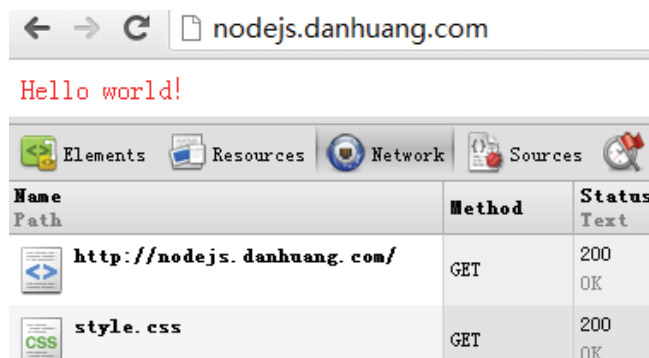


图 3-48 静态资源页面响应

接下来我们再次刷新页面，可以看到如图 3-49 所示的返回结果，但是 style.css 的请求返回状态码为 304，也就是说使用了本地缓存，实现了静态文件模块管理的功能。



	<code>http://nodejs.danhuang.com/</code>	GET	200 OK	text/html
	<code>style.css</code>	GET	304 Not Modified	text/css

图 3-49 文件请求 HTTP 返回状态码

使用 Node.js 创建服务器时，经常会单独处理 `favicon.ico` 请求。Nginx 同样也可以处理，在上面的配置文件中，可以添加一个正则类匹配是否为 `favicon.ico`，如果是就直接退出，配置内容如下（修改配置文件后，重启 Nginx）：

```
location ~(favicon.ico) {
    break;
}
```

如果希望客户端请求 `favicon.ico` 的时候做其他处理响应也是可以的。现在我们在 `hello.js` 中添加一段代码，来检测经过 Nginx 的处理后是否还有 `favicon.ico` 的 HTTP 请求。

```
var pathname = decodeURI(url.parse(req.url).pathname);
console.log(pathname);
```

【代码说明】

□ `pathname = decodeURI(url.parse(req.url).pathname)`：获取请求路径。

重新运行 `hello.js`，打开浏览器输入 `nodejs.danhuang.com/a` 和 `nodejs.danhuang.com` 后，在服务器端可以看到如图 3-50 所示的输出仅仅是一个 `/a` 请求和 `/`，已经成功的通过 Nginx 过滤掉了 `/favicon.ico` 的 HTTP 请求。

```
root@ubuntu:/home/danhuang/helloworld# node hello.js
Server running at http://localhost:8000/
/a
/
```

图 3-50 服务端打印用户请求文件路径

以上就介绍了 Node.js 如何结合 Nginx 构建服务器的一些基本知识。关于负载均衡处理的配置方式将会在下一节介绍。

3.7 文字直播实例

本节将会根据本章学习的知识点一步步地带领大家开发“文字直播”这个项目。

3.7.1 系统分析

文字直播 Web 应用需求简单介绍如下。

- (1) 用户：直播员（需登录）、游客。
- (2) 直播方式：直播员登录后台，输入相应的直播信息，可包含图片、文字。
- (3) 技术统计：需在线实时记录运动员数据。

(4) 游客讨论：游客可实时进行在线讨论。

(5) 微博分享：游客可通过腾讯微博和新浪微博分享直播内容。

从需求中可以分析需要的模块，有登录系统但只针对特定用户，为登录模块。游客之间可互相交流，存在一个聊天室功能，为聊天功能模块。微博分享，为分享功能模块。技术统计数据，为数据统计管理功能模块。分析出以上几个模块后，我们就可以简单地画一个系统分析图来看一下需要设计哪几个模块，如图 3-51 所示。

游客可以查看统计、登录和分享，而直播员则能够进入直播模块进行现场的直播。从需求中可知本系统存在两种用户，三个主要功能（直播、统计和分享）。

根据上述分析的结果，我们就开始进行代码架构部署了，分析需要的模块，以及路由请求的处理。

5 个 controller 分别对应 5 个模块 chat.js、live.js、login.js、score.js、share.js。5 个 controller 继承父类 action.js。router.js 负责路由处理；app.js 则为入口文件，初始化路径以及项目所需模块。

根据本系统的分析，可知本系统将会应用到的 Node.js 原生模块和自我开发的模块如表格 3.2 所示。

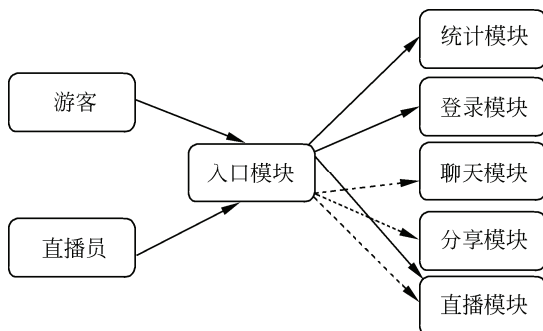


图 3-51 需要设计的模块

表 3.2 本系统应用模块

Node.js 模块名	文件名称	模块描述
http	Node.js 原生模块	负责服务器创建管理
fs	Node.js 原生模块	文件处理，读取
url	Node.js 原生模块	url 请求路径处理
querystring	Node.js 原生模块	HTTP 请求参数分析模块
httpParam	http_param.js	负责 HTTP 参数 GET 和 POST 获取方式
staticModule	static_module.js	负责本系统所有静态文件的管理
router	router.js	本系统路由处理模板
action	action.js	系统 controller 基类
jade	NPM 模块	前端模版
socket	NPM 模块	socket 模块
path	Node.js 原生模块	路径处理
util	Node.js 原生模块	主要应用其继承 API
session	node_session.js	他人开发的一个 session 管理模块

了解了系统之后，我们开始设计项目的文件结构（项目的文件结果可以表明一个项目的框架和运行方式），项目整体架构如图 3-52 所示。

其中的 app 文件夹是 Node.js 可运行代码。controller 包含了前面介绍的 5 个 controller。同时一些重要处理的模块，例如 action.js 和 router.js 存放在 core（core 是一个无依赖的模块文件夹）。conf 存放本系统的一些配置文件夹信息。node_modules 存放的是表格中所介

绍的一些 NPM 模块和自我开发模块。static 中存放静态文件资源包括 css、image 和 js 文件。View 存放 jade 模块文件。app.js 为本系统的入口文件，本系统的运行是通过运行入口 app.js 来启动的。

在下一节我们会着重介绍本系统中的两个重要文件模块 app.js 和 router.js，通过这两个文件模块的介绍可以了解本系统的应用开发。

3.7.2 重要模块介绍

在上一节中介绍到本系统中的两个重要文件模块 app.js 和 router.js，下面我们先介绍 app.js 入口文件。app.js 是本系统的一个运行文件，其主要是做模块的引入，数据的初始化，同时会创建 HTTP 服务器，并调用 router.js 中的方法，代码 1 如下：

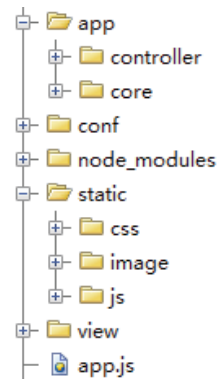


图 3-52 项目目录结构

```

/**
 *
 * 设置路径全局变量
 */
global.BASE_DIR = __dirname;
global.APP      = BASE_DIR + "/app/";
global.CON      = APP + "/controller/";
global.CORE     = APP + "/core/";
global.LIB      = BASE_DIR + "/node_modules/";
global.CONF     = BASE_DIR + "/conf/";
global.STATIC   = BASE_DIR + "/static/";
global.VIEW     = BASE_DIR + "/view/";

```

【代码说明】

- global.BASE_DIR = __dirname: 获取本系统的根路径。
- global.APP = BASE_DIR + "/app/": 设置 app 文件夹的路径全局变量。
- global.CON = APP + "/controller/": 设置 controller 文件夹的路径全局变量。

这段代码的主要作用是初始化本系统的文件夹路径，使用全局变量来统一化管理系统文件夹路径。注意，这里的路径名最好使用大写字母，避免在其他文件中出现变量同名。代码 2 如下：

```

/**
 * modules 引入
 */
global.lib = {
  http      : require('http'),
  fs        : require('fs'),
  url       : require('url'),
  querystring : require('querystring'),
  httpParam : require(LIB + 'http_param'),
  staticModule: require(LIB + 'static_module'),
  router    : require(CORE + 'router'),
  action    : require(CORE + 'action'),
  jade      : require('jade'),
  socket    : require('socket.io'),
  path      : require('path'),

```

```

    parseCookie : require('connect').utils.parseCookie,
    session      : require(LIB + 'node_session'),
    util         : require('util')
  }

```

【代码说明】

- ❑ global.lib: 添加命名空间，避免变量重名。
- ❑ http: require('http'): http 模块引入。
- ❑ staticModule: require(LIB + 'static_module'): 静态文件模块 static_module 引入。

本部分代码的主要作用是统一管理本系统所应用到的所有 Node.js 原生模块、NPM 模块和自我开发模块，其中需要注意的是，添加了一个命名空间，和上面的路径设置为全大写全局变量的目的相同，避免出现全局变量在其他文件中被覆盖的现象。代码 3 如下：

```

global.onlineList = [];

global.app = lib.http.createServer(function(req, res) {
  res.render = function(){
    var template = arguments[0];
    var options = arguments[1];
    var str = lib.fs.readFileSync(template, 'utf8');
    var fn = lib.jade.compile(str, { filename: template, pretty: true });
    var page = fn(options);
    res.writeHead(200, { 'Content-Type': 'text/html' });
    res.end(page);
  }
  lib.router.router(res, req);
}).listen(8000);
global.io = lib.socket.listen(app);

```

【代码说明】

- ❑ global.onlineList = []: 主要是存储 socket 连接用户信息，其同样可添加命名空间。
- ❑ global.app = lib.http.createServer(function(req, res){}): 创建 HTTP 服务器，并做一些 res 和 req 的预处理。
- ❑ res.render = function(){}: 为 res 添加 jade 模块引擎方法 render。
- ❑ var template = arguments[0]: 获取 render 函数的第一个参数，作为指定需要访问的 jade 模板文件名。
- ❑ var options = arguments[1]: 获取 render 函数的第二个参数，作为传递给 jade 模块引擎的参数对象。
- ❑ var str = lib.fs.readFileSync(template, 'utf8'): 使用 utf8 读取 jade 模板文件内容。
- ❑ fn = lib.jade.compile(str, { filename: template, pretty: true }): 获取 jade 模板编译执行函数。
- ❑ var page = fn(options): 传递参数并编译执行 jade 模板文件，返回 html 内容，赋值给 page 变量。
- ❑ lib.router.router(res, req): 调用 router 的 router 方法执行 url 路由请求处理。
- ❑ global.io = lib.socket.listen(app): 启动本系统的 socket 服务。

本段代码中的 render 解析 jade 模板的方法在文件系统一章中介绍过。这里直接将 render 的 jade 解析函数赋值到 res 对象中，主要原因在于 HTTP 响应返回一个 html 信息是 res 对象的职责。render 函数通过 argument 来获取参数，而不使用直接传递参数的目的在于，可

以灵活地传递 `render` 函数所需的参数。例如我们可以通过如下两种方法调用 `res` 中的 `render` 方法，代码如下。

```
res.render('index.jade')
res.render('index.jade', {'user' : 'danhuang'})
```

`app.js` 代码中的 `app` 和 `io` 对象在后续的 `controller` 和 `router` 中都需要应用到，因此这里设置为一个全局的变量。`lib.router.router(res, req)` 就是调用 `router` 模块中的 `router` 方法来实现 `url` 路由请求处理。这样我们就实现了本系统的一个入口文件。需要注意的有以下几点：

- ❑ 文件路径需在入口文件统一化管理。
- ❑ 文件路径名最好使用全大写字母进行变量区分，避免和后续变量引起冲突。
- ❑ 所有应用到的模块需添加命名空间，统一化管理。
- ❑ 响应类方法或者数据，统一添加到响应对象 `res` 中进行管理。

接下来我们看一下 `router.js` 的代码实现，其主要是对 `url` 请求处理，同时分发到相应的 `controller` 进行逻辑处理响应 HTTP 信息。

```
var Login = require(CON + 'login');
```

获取 `Login` 登录模块对象，控制本系统的登录系统。

```
exports.router = function(res, req){})
```

将 `router` 函数作为 `require` 的返回对象。

```
var pathname = decodeURI(lib.url.parse(req.url).pathname);
lib.httpParam.init(req, res);
global.sessionLib = lib.session.start(res, req);
var model = pathname.substr(1)
    , controller = lib.httpParam.GET('c')
    , Class = '';
if(pathname == '/favicon.ico'){
    return;
}else if(pathname == '/'){
    res.render(VIEW + 'index.jade');
    return;
}
```

【代码说明】

- ❑ `pathname = decodeURI(lib.url.parse(req.url).pathname)`: 解析编码后的 `url` 字符。
- ❑ `lib.httpParam.init(req, res)`: 初始化 HTTP 的 GET 和 POST 参数获取对象。
- ❑ `global.sessionLib = lib.session.start(res, req)`: session start。
- ❑ `model = pathname.substr(1)`: 获取请求的 `controller`。
- ❑ `controller = lib.httpParam.GET('c')`: 获取请求 `controller` 中的函数方法。
- ❑ `if(pathname == '/favicon.ico'){}):` 忽略 `/favicon.ico` 请求。
- ❑ `res.render(VIEW + 'index.jade')`: 默认进入 `index.jade` 首页。

本部分代码主要是根据请求的 `url` 和参数解析出需要执行该请求的 `controller` 和 `method`，同时初始化系统需要的一些模块对象。

```
try {
    Class = require(CON + model);
}
```

```
catch (err) {
    lib.staticModule.getStaticFile(pathname, res, req, BASE_DIR);
    return;
}
```

【代码说明】

- ❑ `Class = require(CON + model)`: `try require` 一个请求的类。
- ❑ `lib.staticModule.getStaticFile(pathname, res, req, BASE_DIR)`: `require` 失败时，默认为静态文件请求。

⚠注意：这部分代码中需要使用 `try catch` 去 `require` 一个模块类，避免出现异常中断 Node.js 运行程序。这里可以将所有的 url 请求分为资源请求和逻辑处理，因此在尝试 `require` 一个模块类失败时，则默认为静态资源的请求。代码如下：

```
if(Class){
    var login = new Login(res, req);
    var ret = login.checkSession(model);
    if(ret) {
        var object = new Class(res, req);
        object[controller].call();
    } else {
        res.render(VIEW + 'index.jade');
    }
} else {
    res.writeHead(404, { 'Content-Type': 'text/plain' });
    res.end('can not find source');
}
```

【代码说明】

- ❑ `var login = new Login(res, req)`: 创建 `Login` 类的 `login` 对象。
- ❑ `ret = login.checkSession(model)`: 判断用户是否已登录。
- ❑ `var object = new Class(res, req)`: 创建请求 `controller` 类的对象 `object`。
- ❑ `object[controller].call()`: 调用 `object` 对象的 `controller` 方法。
- ❑ `res.render(VIEW + 'index.jade')`: 未登录用户，则调整到 `index.jade` 页面。
- ❑ `res.writeHead(404, { 'Content-Type': 'text/plain' })`: 如果不存在 `class` 对象时，则返回 404 not find。

这部分代码需要注意的是，`controller` 是一个字符串，因此无法通过 `object.controller` 调用，同样 `object[controller]` 方法也无法执行函数，因此需要使用 `object[controller].call()` 执行 `object` 中的 `controller` 方法。这里的 `Class` 和 `Login` 都是一个类，而不是一个对象，因此可以使用 `new` 来创建对象。在前面介绍了如何使用 `require` 返回一个类，这里再介绍一下，主要是应用 `module.exports`，该 API 可以返回任何的数据类型，参考代码如下：

```
module.exports = function(res, req){
    this.show= function(){
        console.log('this is a class method')
    }
}
```

介绍了两个重要的模块后，我们再看一个逻辑处理 `controller Login` 类，其主要负责用

户的登录、Session 管理和用户登录后的 socket 启动运行。代码如下：

```
module.exports = function(){

    var _res = arguments[0];
    var _req = arguments[1];

    this.checkSession = function(model){
        if(model == 'login'){
            return true;
        }else if(sessionLib.username && sessionLib.username != '') {
            return true;
        }
        return false;
    }
}
```

【代码说明】

- ❑ `module.exports = function(){}:` 上面已经介绍了使用 `module.exports` 可以在 `require` 模块时返回一个类。
- ❑ `var _res = arguments[0]:` 获取函数的第一个参数 `res`，并赋值给类的私有变量 `_res`。
- ❑ `var _req = arguments[1]:` 获取函数的第二个参数 `req`，并赋值给类的私有变量 `_req`。
- ❑ `this.checkSession = function(model){}`: 验证是否存在 `session` 来判断用户是否已登录。
- ❑ `if(model == 'login'){return true}`: 如果调用的 `controller` 是 `login` 模块时，则无需登录。
- ❑ `sessionLib.username && sessionLib.username != ''`: 判断是否存在 `Session`。

因为本系统存在登录，因此需要注意分清访问的模块是否需要登录。例如访问 `login` 就不应该进行 `Session` 检查，否则永远无法登录本系统。

下面看一下该模块中的 `login` 方法，其主要是获取客户端传递的 `username`，然后启动 `socket` 服务，监听客户端的连接，代码如下：

```
this.login = function(){
    lib.httpParam.POST('username', function(value){});
}
```

【代码说明】

- ❑ `lib.httpParam.POST('username', function(value){})`: 调用之前我们开发的 `HTTP POST` 参数获取方法。

上面代码只是整体上看了一下 `login` 函数，接下来我们来看 `POST` 函数的回调函数 `function(value)` 的作用，代码 1 如下：

```
sessionLib.username = value;
if(value == 'danhuang'){
    _res.render(VIEW + 'live.jade', {'user' : value});
} else {
    _res.render(VIEW + 'main.jade', {'user' : value});
}
```


【代码说明】

- ❑ sessionLib.username = value: 设定 Session 中的 username 值。
- ❑ value == 'danhuang': 判断是否为管理员登录，管理员这里暂定为 danhuang。
- ❑ _res.render(VIEW + 'live.jade', { 'user': value }): 如果是直播员，则进入直播模块。
- ❑ _res.render(VIEW + 'main.jade', { 'user': value }): 如果非直播员，则进入观看直播模块。

本系统没有应用到复杂的登录验证信息，只是简单地通过登录的用户名来判断是否为管理员，实际的项目中需要添加登录验证用户名和密码来判断是否为管理员。代码 2 是设置 socket 监听，代码如下：

```
var time = 0;
io.sockets.on('connection', function (socket){
    var username = sessionLib.username;
    if(!onlineList[username] ){
        onlineList[username] = socket;
    }
    var refresh_online = function(){
        var n = [];
        for (var i in onlineList){
            n.push(i);
        }
        var message = lib.fs.readFileSync(BASE_DIR + '/live_data.txt', 'utf8');
        socket.emit('live_data',message);
        io.sockets.emit('online_list', n);//所有人广播
    }
    refresh_online();
    //确保每次发送一个 socket 消息
    if(time > 0){
        return;
    }
    socket.on('public', function(data){
        var insertMsg = "<li><span class='icon-user'></span><span class='live_user_name text-success'>[danhuang]</span><span class='live_message text-info'>" + data.msg + "</span></li>"
        writeFile({ 'msg':insertMsg, 'data':data}, function(data){
            io.sockets.emit('msg', data);
        });
    });
    socket.on('disconnect', function(){
        delete onlineList[username];
        refresh_online();
    });
    time++;
});
```

【代码说明】

- ❑ var time = 0: 避免 socket 发送多条消息。
- ❑ io.sockets.on('connection', function (socket){}): 监听处理客户端 socket 连接。
- ❑ var username = sessionLib.username: 通过 Session 获取当前登录用户名。

- ❑ `if(!onlineList[username])`: 判断 `username` 是否已经存在在线用户列表。
- ❑ `onlineList[username] = socket`: 如果不存在在线用户列表变量 `onlineList` 中, 则将其 `socket` 添加到 `onlineList` 中。
- ❑ `var refresh_online = function(){}:` 设置刷新在线用户函数。
- ❑ `io.sockets.emit('online_list', n)`: 如果有新用户登录后, 广播在线用户列表。

```
function writeFile(data, callback){
    var message = lib.fs.readFileSync(BASE_DIR + '/live_data.txt', 'utf8');
    lib.fs.writeFile(BASE_DIR + '/live_data.txt', message+data.msg,
function(err){
    if (err) throw err;
    callback(data.data);
});
}
```

`writeFile(data, callback)`, 该函数使用 `utf8` 编码格式读取一个文件内容, 并通过 `callback` 函数返回读取文件的内容。系统会将之前直播的内容存储到一个文件中, 如上是一个 `live_data.txt` 文件中。当用户首次进入系统后, 系统会读取该文件, 并将该文件的内容传送到客户端, 客户端可以看到之前直播的所有文字信息。

到此我们就成功的设计了一个可简单直播的 Web 应用。启动运行 `app.js` 文件, 同时使用 `Chrome` 和 `Firefox` 浏览器打开 `http://127.0.0.1:8000`, 如图 3-53 所示。这里需要说明的是, 由于本系统使用的是 `bootstrap`¹, 因此只兼容 `Chrome` 和 `Firefox`, `IE` 打开后无法看到正常的页面信息。

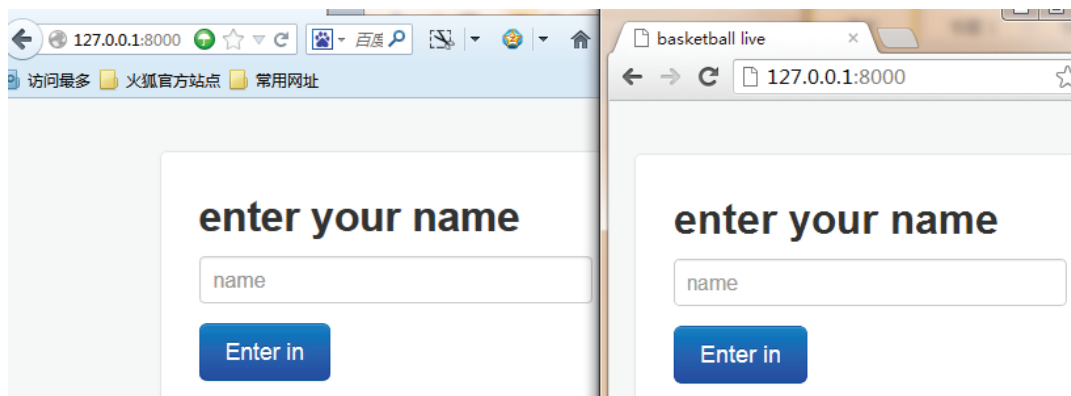


图 3-53 web 登录页面

打开网页后可以看到如图 3-53 所示的页面, 其中一个输入 `danhuang` 作为直播员, 另外一个输入任何字符, 如果希望使用第三个用户登录, 可以使用 `Chrome` 浏览器的隐藏窗口, 隐藏窗口不会共享 `Cookie` 和 `Session` 信息, 如图 3-54 所示。

如图 3-55 是添加三个用户 (`danhuang` 直播员、`girl` 和 `boy`) 后的直播页面。

可以看到在线用户列表有 `boy`、`girl` 和 `danhuang`, 中间为直播信息显示, 右侧为微博分享, 页面底端则为直播输入框, 直播输入框只会在直播员登录页面时出现, 右侧和底端显示如图 3-56 所示。

1 参考网站 <http://twitter.github.com/bootstrap/>。



图 3-54 Chrome 打开多个无影响 tab 方式



图 3-55 应用直播页面



图 3-56 应用微博分享页面

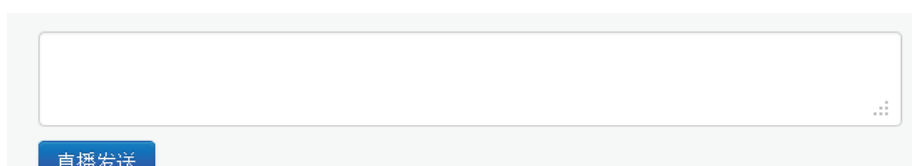


图 3-57 应用直播内容发送窗口

接下来直播员输入任何信息都会直接广播给所有连接用户，这样就实现了简单的文字直播功能。本系统的其他功能模块希望读者能够自我实现，本章节不再介绍具体的实现方法。下面介绍一个扩展阅读——通过腾讯微博 API 来实现使用 Node.js 分享微博。

3.8 扩展阅读

本章 3.7.2 节中的应用涉及 Node.js 的微博分享功能，这里就简单介绍一下通过腾讯微博 API 来实现使用 Node.js 分享微博的功能。要运行该段测试代码，需要 PHP 的 apache 支持，相关 PHP 的运行环境配置可以参考《PHP5+APACHE2.2 配置成功案例》¹。

腾讯微博提供了多种语言的 sdk，但是没有提供 Node.js 的相关 sdk。因为其提供了 PHP 的 sdk，因此我们希望应用 PHP 的相关 sdk 来实现腾讯微博的授权过程，使用 Node.js 来调用 PHP 的接口实现发表微博。首先我们需要了解一些 Node.js 中如何发送一个 GET 或者 POST 请求。在第 2 章中我们介绍了 request 模块，可以请求一个 HTTP 请求，现在我们就应用该模块实现一个类似于 PHP 中的 curl 模块。

curl 模块包括两个方法：curl_get 和 curl_post。curl_get 方法主要是发起一个 HTTP 的 GET 请求，而 curl_post 则是发起一个 HTTP 的 POST 请求。首先我们看一下 curl_get 的实现代码：

```
var request = require('request');
var querystring = require('querystring');
curl_get : function(url, get, callback){
    var org_url    = url
        , org_get   = get
        , params    = {}

    if(get){
        if(url.indexOf('?') > -1){
            url = url + '&';
        } else {
            url = url + '?';
        }
    }
    url = url + querystring.stringify(get);

    params['url'] = url;
    params['json'] = true;

    request.get(params, function(error, response, result){
        if(error){
            console.log(error);

            callback(result);
        } else {
            callback(result);
        }
    });
}
```

【代码说明】

- ❑ var request: 获取 request 模块；
- ❑ var querystring: 获取 querystring 模块，用于处理 HTTP 参数；

¹ 参考网站 <http://hi.baidu.com/oyej2012/item/92399224ec869951c38d591b#send>。

- ❑ `curl_get`: 定义 `curl_get` 方法;
- ❑ `function(url, get, callback)`: `url` 为 HTTP 的 url, `get` 为请求参数, `callback` 为回调函数;
- ❑ `url = url + querystring.stringify(get)`: 应用 `querystring` 方法将 `get` 的 json 对象转化为 HTTP 中 GET 参数字符串;
- ❑ `request.get(params, function(error, response, result))`: 应用 `request` 模块中的 GET 方法发起 HTTP 请求, `result` 为响应数据。

`curl_get` 模块中应用到 `request` 中的异步接口 GET 方法, 因此在定义 `curl_get` 函数时, 我们需要添加 `callback` 来返回 GET 结果。在为 `url` 中添加 `get` 参数时, 通过查询 `url` 中是否包含 `?` 来判断, 如果没有参数需要为 `url` 后缀加上 `?`, 如果有则为其 `url` 后缀添加字符 `&`。Node.js 提供了 `querystring` 对象来处理 HTTP 中的参数, 其处理办法是将 json 对象转化为 HTTP 的 `get` 参数字符串, 例如有如下对象:

```
{
  'name' : 'test',
  'book' : 'node'
}
```

经过 `querystring.stringify` 处理之后将会转化为字符串 `name=test&book=node`。

`curl_get` 和 `curl_post` 的实现方法大致相同, 其代码如下:

```
curl_post : function(url, post, callback){
  var org_url      = url
    , org_post      = post
    , params = {};

  params[ 'url' ] = url;
  params[ 'json' ] = true;
  params[ 'form' ] = post;

  request.post(params, function(error, response, result){
    if(error){
      callback(result);
    } else {
      callback(result);
    }
  });
}
```

【代码说明】

- ❑ `curl_post`: 定义 `curl_post` 方法;
- ❑ `request.post(params, function(error, response, result))`: 应用 `request` 模块中的 POST 方法发起 HTTP 请求, `result` 为响应数据。

POST 方法和 GET 方法实现方法类似, 只是 POST 和 GET 两者的参数不同。相关 `request` 模块信息可以查看其 [github 主页 https://github.com/mikeal/request](https://github.com/mikeal/request)。

实现了 `curl` 模块以后, 接下来我们只需要应用该模块发起一个 HTTP 请求调用 PHP 的相关的微博接口, 应用 PHP 授权来操作腾讯微博接口。下面就来看一下 Node.js 是如何与 PHP 合作实现授权, 并发表微博的。

首先我们下载一个 `Iweibo` 源码, 该源码中提供了腾讯微博的授权, 以及相关腾讯微博

接口的实现，相关地址 <http://dev.t.qq.com/iweibo/>。这里我们演示的项目代码为 Iweibo3.0 的代码，只截取其中的 upload/application/Core/Open 相关类。接下来我们应用 Open 相关的类实现一个简单的发表微博接口，实现代码如下：

```
<?php
require('Open/Api.php');
require('Open/Clinet.php');
require('Open/Oauth.php');
require('Open/Opent.php');

class Core_OperationOpent
{
    /**
     * 发一条微博
     * @param unknown_type $tweet
     * @return : return_type
     */
    public static function add($p, $wbInfo=array())
    {
        if (empty($p)){
            return null;
        }
        try{
            $ret = Core_Open_Api::getAdminClient($wbInfo)->postOne($p);
        }catch(Exception $e){
            var_dump($e);
        }
        $data = isset($ret['data']) ? $ret['data'] : array();
        return $data;
    }
}
```

【代码说明】

- ❑ Api.php、Clinet.php、Oauth.php 和 Opent.php: Opent 类接口;
- ❑ add(\$p, \$wbInfo=array()): 发表一条微博;
- ❑ \$p: 发表微博相关内容;
- ❑ \$wbInfo: 微博相关的 token 信息。

上面的 PHP 代码主要是通过调用 iweibo 的 opent 类库来实现发表微博的功能。其中的 \$wbInfo 就包含了发表微博账户的 access_token、access_token_secret、appkey 和 appsecret，而 \$p 参数则是发表微博的相关数据，其详细字段可以参考腾讯微博的官方文档 <http://wiki.open.t.qq.com/index.php/API> 文档/微博接口/发表一条微博信息。

接下来我们去腾讯微博申请一个 appkey 和 appsecret, 然后应用该 appkey, 以及 appsecret 生成 token 信息。

你已经获得授权。你的授权信息:

```
Access token: 9a096ed3796c4545a3[REDACTED]  
oauth_token_secret: d45267a16e7ab5040ca2[REDACTED]  
openid: C81A4B8658E051[REDACTED]FBE4CE29B68B6  
openkey: EE86C1B9D6E[REDACTED]D1A7CBEC94B59CBE7
```

图 3-58 授权成功返回 token 信息

如图 3-58 所示的信息是通过腾讯 PHPsdk 生成的 token 信息，下载完该 sdk 后，修改 appkey.php 将其中的 appkey 和 appsecret 修改为自己的信息，然后授权即可生成相应的 token 信息。

接下来我们创建一个 index.php 来实现发表微博的接口，相关代码如下：

```
<?php
require('OperationOpent.php');
$token = json_decode($_REQUEST['t']);
$p      = json_decode($_REQUEST['p']);
$ret    = Core_OperationOpent::add($p, $token);
echo json_encode($ret);
```

【代码说明】

- ❑ require('OperationOpent.php'): require 微博操作类；
- ❑ \$token = \$_REQUEST['t']: 获取 HTTP 参数 t；
- ❑ \$p = \$_REQUEST['p']: 获取 HTTP 参数 p；
- ❑ Core_OperationOpent::add: 调用 add 发表微博；
- ❑ echo json_encode(\$ret): 返回发表微博结果。

现在我们将该 PHP 代码通过 Apache 来运行，在浏览器中打开如下 url 可以发表微博信息：

```
http://127.0.0.1/extension/index.php?p={"format":"json","content":"test"}&t={"access_token":"637c6a0de253441dbcc79c0*****","access_token_secret":"649cfccf26195808a081cfece*****","appkey":"783cfd16e4c6418c854dd208ad58cb70","appsecret":"0388bbc411723d00f30235ff4522f0d9"}
```

以上的 token 信息，请使用自己的，这里的 token 信息为了安全笔者使用了*代替。

以上代码中的 p 和 t 参数分别表示发表微博的 json 字符和 token 信息 json 字符，访问如上连接后，可以看到其返回了如下 json 字符：

```
{"id":"237265121656660","time":1366296854}
```

在腾讯微博中查看该微博内容如图 3-59 所示。



图 3-59 测试结果发表微博页面显示

上面就是 PHP 实现的一个发表微博接口，接下来我们通过 Node.js 中的 curl 模块来发起 HTTP 请求发表微博。应用 curl 模块，实现如下代码：

```

var curl = require('./curl');
var post = {}, p={}, t = {};
p['format'] = 'json';
p['content'] = 'node.js use php to add tweet';

t['access_token'] = '637c6a0de253441d*****';
t['access_token_secret'] = '649cfccf26195*****';
t['appkey'] = '783cfd16e4c6418c854dd208ad58cb70';
t['appsecret'] = '0388bbc411723d00f30235ff4522f0d9';

post['t'] = JSON.stringify(t);
post['p'] = JSON.stringify(p);

curl.post('http://127.0.0.1/extension/index.php', post, function(ret){
    console.log(ret);
});

```

【代码说明】


- t: 为该小号的 token 信息;
- p: 为具体发表的微博内容;
- JSON.stringify: 将 json 对象转化为字符, 便于数据传递;
- curl.post: 应用 curl 模块发起 HTTP 的 GET 请求。

以上为 Node.js 通过 curl 模块调用 PHP 接口的方法, 需要注意的是, 在数据传递的时候, 由于无法传递 json 对象数据结构, 因此这里需要将其转化为 json 字符, 然后在 PHP 端通过 json 字符解析, 从而得到相应的数据结构。代码中的相关 token 以及 appkey 都为笔者个人信息 (appkey 可以供大家测试, token 信息使用*代替), 这里可能会失效, 希望大家前往腾讯微博进行申请。

在 PHP 的代码中我们只实现了其中的发表微博, 其中还有其他的微博相关接口, 可以参考 iweibo3.0 实现。

3.9 本章实践

1. 根据 3.1.2 节路由处理方法, 实现 127.0.0.1:1337/image/img 的 url 请求规则, image 为调用的模块名, img 为 image 模块中的 img 方法。例如: /image/imgJpg 读取一个 jpg 图片资源, /image/imgPng 获取一个 png 图片资源, /index/index 返回 index.html 页面, 其他数据格式返回 404, 并输出 “not find!”。

 **实现提示:** req 获取 url 字符串, 使用字符串切割获取 image 和 img 参数, 利用“附带参数来实现路由”中的 url 解析出需要执行的模块名以及模块方法, 然后再根据模块名和函数名调用 classObj.init(res, req) 和 classObj[controller].call() 来实现请求资源分配。

分析: 需要实现 router.js、index.js 和 image.js, 相关代码如下所示。

router.js 代码如下:


```

var http = require('http'),
    url = require('url'),
    querystring = require("querystring");
http.createServer(function(req, res) {
    var pathname = url.parse(req.url).pathname;
    if (pathname == '/favicon.ico') {
        return;
    }
    var pathArr = pathname.split('/');
    // 弹出第一个空字符
    pathArr.shift();

    var model = pathArr.shift()
        , controller = pathArr.shift()
        , classObj = '';
    if(!model || !controller){
        res.writeHead(404, { 'Content-Type': 'text/plain' });
        res.end('can not find source');
    }

    try {
        classObj = require('./' + model);
    }
    catch (err) {
        console.log('chdir: ' + err);
    }
    if(classObj){
        classObj.init(res, req);
        try{
            classObj[controller].call();
        } catch(err){
            res.writeHead(404, { 'Content-Type': 'text/plain' });
            res.end('can not find source');
        }
    }

    } else {
        res.writeHead(404, { 'Content-Type': 'text/plain' });
        res.end('can not find source');
    }
}).listen(1337);

```

index.js 代码如下:

```

/* index.js */
var res, req,
    fs = require('fs'),
    url = require('url');
exports.init = function(response, request){
    res = response;
    req = request;
}

exports.index = function(){
    /* 获取当前 image 的路径 */
    var readPath = __dirname + '/' + url.parse('index.html').pathname;
    var indexPage = fs.readFileSync(readPath);
    res.writeHead(200, { 'Content-Type': 'text/html' });
    res.end(indexPage);
}

```

image.js 实现代码如下:

```
/* image.js */
var res, req,
    fs = require('fs'),
    url = require('url');
exports.init = function(response, request){
    res = response;
    req = request;
}

exports.imgJpg = function(){
    /* 获取当前 image 的路径 */
    var readPath = __dirname + '/' + url.parse('img.jpg').pathname;
    var indexPage = fs.readFileSync(readPath);
    res.writeHead(200, { 'Content-Type': 'image/png' });
    res.end(indexPage);
}
```

运行 router.js, 分别打开以上 3 个 url, 可以看到不同的返回结果。

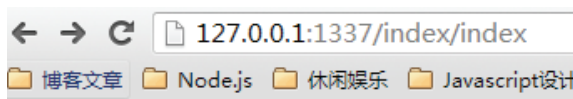
打开 <http://127.0.0.1:1337/image/imgJpg>, 返回页面如图 3-60 所示。



图 3-60 HTTP Web 响应页面

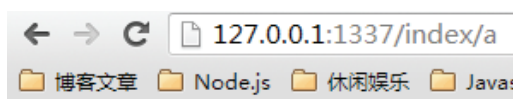
打开 <http://127.0.0.1:1337/index/index>, 返回页面如图 3-61 所示。

打开 <http://127.0.0.1:1337/index/a>, 返回页面如图 3-62 所示。



test for routers

图 3-61 HTTP Web 响应页面

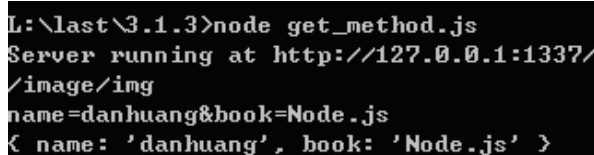


can not find source

图 3-62 HTTP Web 响应页面

2. 根据 3.1.3 节中 `get_method.js` 的测试, 输入 `http://127.0.0.1:1337/image/img? name=danhuang&book=Node.js` 的 HTTP 请求, 查看结果是否和预期的一致, 同时代码实现该 url 请求的路由处理。

分析: 运行 `get_method.js`, 打开浏览器输入中题中的 url, 再查看运行窗口, 可以看到如图 3-63 所示的返回结果。



```
L:\last\3.1.3>node get_method.js
Server running at http://127.0.0.1:1337/
/image/img
name=danhuang&book=Node.js
{ name: 'danhuang', book: 'Node.js' }
```

图 3-63 服务端 Node.js 运行日志

网页计算器: 有两个输入框和一个选择框, 两个输入框主要是填写计算器需要运算的两个值, 而选择框则为计算法则 (加、减、乘、除), POST 提交参数到 Node.js 服务器端, 计算执行结果, 返回 text 计算结果到客户端显示。

网页计算器实现分析: 依据 3.1.2 节实现的 `router.js` 路由处理模块和 `get_method.js` 中获取 GET 参数的方法, 来实现此功能。

`index.js` 代码实现如下:

```
/* index.js */
var res, req,
    fs = require('fs'),
    url = require('url'),
    querystring = require('querystring');
exports.init = function(response, request){
    res = response;
    req = request;
}

exports.index = function(){
    /* 获取当前 image 的路径 */
    var readPath = __dirname + '/' + url.parse('index.html').pathname;
    var indexPage = fs.readFileSync(readPath);
    res.writeHead(200, { 'Content-Type': 'text/html' });
    res.end(indexPage);
}

exports.calculate = function(){
    var pathname = url.parse(req.url).pathname,
        paramStr = url.parse(req.url).query,
        param = querystring.parse(paramStr);

    var type = param['type'] ? parseInt(param['type']) : 0
        , preValue = param['pre'] ? parseFloat(param['pre']) : 0
        , nextValue = param['next'] ? parseFloat(param['next']) : 0
        , ret = 0;
    switch(type){
        case 1 : ret = preValue + nextValue;
            break;
        case 2 : ret = preValue - nextValue;
            break;
        case 3 : ret = preValue * nextValue;
```

```

        break;
        case 4 : ret = preValue / nextValue;
        break;
    }
    ret = ' ' + ret;
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end(ret);
}

```

index.html 代码如下:

```

<html>
  <head>
    <meta http-equiv="content-type" content="text/html; charset=utf-8">
    <title>calculate</title>
  </head>
  <body>
    <div>
      <form method='GET' action='/index/calculate'>
        第一个值: <input type='text' name='pre' /><br>
        第二个值: <input type='text' name='next' /><br>
        <input type="radio" name="type" value = "1">+
        <input type="radio" name="type" value = "2" checked>-
        <input type="radio" name="type" value = "3">*
        <input type="radio" name="type" value = "4">/<br>
        <input type="submit" value = "计算">
      </form>
    </div>
  </body>
</html>

```

执行本章中的 router.js, 打开浏览器访问 <http://127.0.0.1:1337/index/index>, 可以得到如图 3-64 所示的页面, 然后输入相应的参数 85 除 23。

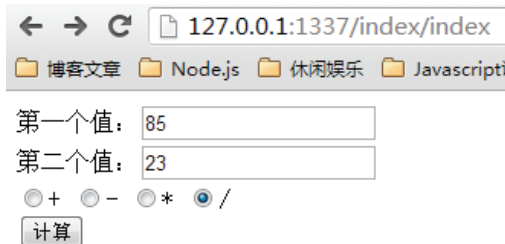


图 3-64 Web 计算器页面

运行结束后, 可以得到执行结果, 如图 3-65 所示。

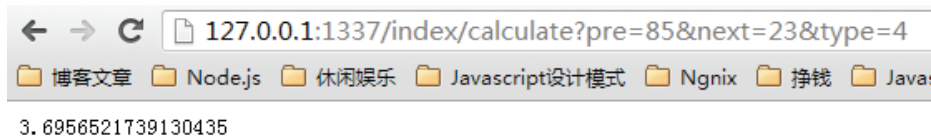


图 3-65 Web 计算器运行结果

3. 网页计算器 POST: 利用该 httpParam 模块重构习题 2 中的“网页计算器”。

分析: 这里大家可以使用 POST 传递部分参数, 同时使用 GET 传递部分参数, 来应用

和学习该 HTTP 参数获取模块。也希望大家灵活应用到其他的开发中，同时希望读者能够在学习过程中学会举一反三。

在习题 2 中主要是通过 GET 方式获取 HTTP 请求参数，而本题只需要将 GET 方法获取改为 POST，同时应用到 `http_param` 模块来获取数据，相关代码如下。

```
/* index.js */
var res, req,
    fs = require('fs'),
    url = require('url'),
    querystring = require('querystring'),
    httpParam = require('./http_param');
exports.init = function(response, request){
    res = response;
    req = request;
    httpParam.init(req, res);
}

exports.index = function(){
    /* 获取当前 image 的路径 */
    var readPath = __dirname + '/' + url.parse('index.html').pathname;
    var indexPage = fs.readFileSync(readPath);
    res.writeHead(200, { 'Content-Type': 'text/html' });
    res.end(indexPage);
}

exports.calculate = function(){
    httpParam.POST('', function(param){
        var type = param['type'] ? parseInt(param['type']) : 0
            , preValue = param['pre'] ? parseFloat(param['pre']) : 0
            , nextValue = param['next'] ? parseFloat(param['next']) : 0
            , ret = 0;
        switch(type){
            case 1 : ret = preValue + nextValue;
                     break;
            case 2 : ret = preValue - nextValue;
                     break;
            case 3 : ret = preValue * nextValue;
                     break;
            case 4 : ret = preValue / nextValue;
                     break;
        }

        ret = '' + ret;
        res.writeHead(200, { 'Content-Type': 'text/plain' });
        res.end(ret);
    });
    return;
}
```

4. 根据本章 3.1.3 节介绍的 `httpParam` 模块重构第 3 题中的“网页计算器”。

分析：这里大家可以使用 POST 传递部分参数，同时使用 GET 传递部分参数，来应用和学习该 HTTP 参数获取模块。也希望大家灵活应用到其他的开发中，同时希望读者能够在学习过程中学会举一反三。代码如下：

```
var _res, _req,
    url = require('url'),
    querystring = require('querystring');
```

```

/**
 * 初始化 res 和 req 参数
 */
exports.init = function(req, res){
  _res = res;
  _req = req;
}

/**
 * 获取 GET 参数方法
 */
exports.GET = function(key){
  var paramStr = url.parse(_req.url).query,
      param = querystring.parse(paramStr);
  return param[key] ? param[key] : '';
}

/**
 * 获取 POST 参数方法
 */
exports.POST = function(key, callback){
  var postData = '';
  _req.addListener('data', function(postDataChunk) {
    postData += postDataChunk;
  });
  _req.addListener('end', function() {
    // 数据接收完毕, 执行回调函数
    var param = querystring.parse(postData);
    if(key != ''){
      callback(param[key] ? param[key] : '');
      return;
    }
    callback(param);
  });
}

```

5. 学完 3.2.2 节的简单静态资源管理后, 我们需要在 index.html 中添加一段 html 代码, 如下:

```


<script src="alert.js"></script>
<img src='logo.jpg' />

```

- ❑ <script src="alert.js"></script>: alert.js 文件中执行一个 alert 函数, 输出 ‘yes you can!’。
- ❑ : 展示一个 logo.jpg 图片。

(1) 在 index.html 的基础上增加了两个新的静态资源, 对 index.html 的请求将会产生多少个 HTTP 请求和响应?

(2) 服务器端如何处理 JavaScript 和 jpg 的 MMIE 类型的文件返回?

提示: JavaScript 和 jpg 对应的 MMIE 类型分别是 "text/javascript" 和 "image/jpeg"。

(3) 使用代码实现该功能。

分析: 修改静态文件 dealWithStatic 函数中的返回类型, 添加 JavaScript 静态文件和 jpg 类型文件, 修改 app.js 代码如下:

```

/* http.js */
var http = require('http'),
    fs    = require('fs'),
    url   = require('url'),
    BASE_DIR = __dirname;

http.createServer(function(req, res) {
    /* 获取当前 index.html 的路径 */
    var pathname = url.parse(req.url).pathname;
    var realPath = __dirname + '/static' + pathname;
    if (pathname == '/favicon.ico') {
        return;
    } else if (pathname == '/index' || pathname == '/') {
        goIndex(res)
    } else {
        dealWithStatic(pathname, realPath, res);
    }
}).listen(1337);
console.log('Server running at http://localhost:1337/');

function goIndex(res){
    var readPath = BASE_DIR + '/' + url.parse('index.html').pathname;
    var indexPage = fs.readFileSync(readPath);
    res.writeHead(200, { 'Content-Type': 'text/html' });
    res.end(indexPage);
}

function dealWithStatic(pathname, realPath, res){
    fs.exists(realPath, function (exists) {
        if (!exists) {
            res.writeHead(404, { 'Content-Type': 'text/plain' });
            res.write("This request URL " + pathname + " was not found on this server.");
            res.end();
        } else {
            var pointPostion = pathname.lastIndexOf('.');
            mmieString = pathname.substring(pointPostion+1),
            mmieType;
            switch (mmieType){
                case 'js' : mmieType = "text/javascript";
                break;
                case 'jpg' : mmieType = "image/jpeg";
                break;
                default:
                    mmieType = "text/plain";
            }
            fs.readFile(realPath, "binary", function(err, file) {
                if (err) {
                    res.writeHead(500, { 'Content-Type': 'text/plain' });
                    res.end(err);
                } else {
                    res.writeHead(200, { 'Content-Type': mmieType });
                    res.write(file, "binary");
                    res.end();
                }
            });
        }
    });
}

```

修改 index.html 代码:

```
<html>
  <head>
    <title>Test Http</title>
    <link rel="stylesheet" href="style.css">
  </head>
  <body>
    <div>Oh, good~!</div>
    <div><img src='logo.jpg' /></div>
  </body>
  <script src="alert.js"></script>
</html>
```

添加 static 静态文件, 并在文件夹中新增 logo.jpg、style.css 和 alert.js, 其中 alert.js 代码如下:

```
alert('yes you can!');
```

style.css 代码如下:

```
div{
  color: red;
}
```

运行 app.js 代码, 并打开浏览器访问 <http://127.0.0.1:1337/>, 可以得到如图 3-66 所示的返回页面。

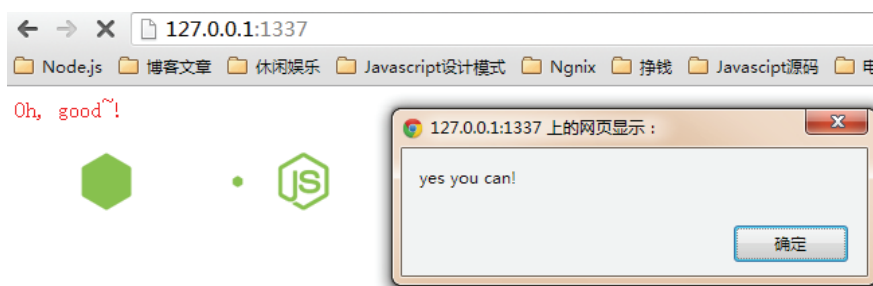


图 3-66 浏览 <http://127.0.0.1:1337/> 响应 Web 页面

6. 应用 3.3.2 节学习的知识点, 创建一个 HTTP 服务器, 当请求 <http://127.0.0.1:1337/index/del&file=danhuang.txt> 时, 首先应用 fs 中的 exists 判断是否存在该文件, 如果不存在, 则服务器响应 “not exist file” 信息到客户端, 如果存在, 则应用 fs 中的 unlink 方法来删除文件, 并响应 “delete file success” 信息到客户端; 当请求 <http://127.0.0.1:1337/index/read> 时, 读取指定文件夹信息, 返回所有文件名。其他 HTTP 请求, 都一致返回 “404 not find”。

分析: 应用本章中路由设计的模块 router.js, 同时添加 index.js 文件, 该 Node.js 脚本处理 del 和 read 两个函数, 代码如下所示。

```
/* index.js */
var res, req,
    fs = require('fs'),
    url = require('url'),
```



```

    querystring = require('querystring'),
    httpParam = require('./http_param'),
    BASE_DIR = __dirname;

exports.init = function(response, request){
    res = response;
    req = request;
    httpParam.init(req, res);
}

```

处理文件删除逻辑:

```

exports.del = function(){
    var file = httpParam.GET('file');
    fs.exists(BASE_DIR+'/file_test/' + file, function (existBool) {
        if(existBool){
            var ret = '';
            fs.unlink(BASE_DIR+'/file_test/' + file, function (err) {
                if (err) {
                    ret = err;
                } else {
                    ret = 'delete file success!';
                }
                res.writeHead(200, { 'Content-Type': 'text/plain' });
                res.end(ret);
            });
        } else {
            res.writeHead(200, { 'Content-Type': 'text/plain' });
            res.end('not exist file!');
        }
    });
}

```

处理 read 读取文件夹目录下文件逻辑:

```

exports.read = function(){
    var folder = httpParam.GET('folder');
    console.log(BASE_DIR+'/' + folder);
    fs.exists(BASE_DIR+'/' + folder, function (existBool) {
        if(existBool){
            var ret = '';
            fs.readdir(BASE_DIR+'/' + folder, function (err, files) {
                if (err) {
                    ret = err;
                } else {
                    ret = JSON.stringify(files);
                }
                res.writeHead(200, { 'Content-Type': 'text/plain' });
                res.end(ret);
            });
        } else {
            res.writeHead(404, { 'Content-Type': 'text/plain' });
            res.end('404 not find!');
        }
    });
}

```

运行 router.js 脚本文件, 打开浏览器分别访问:http://127.0.0.1:1337/index/read?folder=file_test 和 <http://127.0.0.1:1337/index/del?file=danhuang.txt>, 将依次得到如图 3-67 和图 3-68 所示的返回信息。



图 3-67 read 文件夹响应页面信息



图 3-68 执行删除 Web 响应页面

7. 应用本章 Node.js 中 `crypto` 模块 API `cipher` 和 `decipher` 加密算法，实现一个 Web 应用。

用户输入相应加密字符，以及加密私钥，返回一个利用 `cipher` 加密过的字符，同时用户可以选择加密算法和输出字符串的格式。

用户输入解密字符串 `decipher` 和解密私钥，选择返回的字符串格式，系统解密后返回解密后的字符到客户端。

分析：需要应用到 3.1 节介绍的 HTTP 服务器创建、GET 和 POST 参数请求获取，以及 3.2 节中的静态文件管理库和 3.3 节中的 `jade` 前端模板。

```
/* crypto.js */
var res, req,
    fs = require('fs'),
    url = require('url'),
    crypto = require('crypto');
exports.init = function(response, request){
    res = response;
    req = request;
}

exports.index = function(){
    var readPath = __dirname + '/' + url.parse('index.html').pathname;
    var indexPage = fs.readFileSync(readPath);
    res.writeHead(200, { 'Content-Type': 'text/html' });
    res.end(indexPage);
}

exports.cipher = function(){
    var key = httpParam.GET('key'),
        plaintext = httpParam.GET('plaintext'),
        cipher = crypto.createCipher('aes-256-cbc', key);
    cipher.update(plaintext, 'utf8', 'hex');

    var encryptedPassword = cipher.final('hex');
    res.writeHead(200, { 'Content-Type': 'text/html' });
    res.end(encryptedPassword);
}

exports.decipher = function(){
    var key = httpParam.GET('key'),
        plaintext = httpParam.GET('plaintext'),
        decipher = crypto.createDecipher('aes-256-cbc', key);
    decipher.update(plaintext, 'hex', 'utf8');
```

```

var decryptedPassword = decipher.final('utf8');
res.writeHead(200, { 'Content-Type': 'text/html' });
res.end(decryptedPassword);
}

```

相应的 html 文件代码如下：

```

<html>
  <head>
    <meta http-equiv="content-type" content="text/html; charset=utf-8">
    <title>crypto</title>
  </head>
  <body>
    <div>
      <div>加密</div>
      <form method='GET' action='/crypto/cipher'>
        key: <input type='text' name='key' /><br>
        plaintext: <input type='text' name='plaintext' /><br>
        <input type="submit" value = "加密">
      </form>
    </div>
    <div>
      <div>解密</div>
      <form method='GET' action='/crypto/decipher'>
        key: <input type='text' name='key' /><br>
        plaintext: <input type='text' name='plaintext' /><br>
        <input type="submit" value = "解密">
      </form>
    </div>
  </body>
</html>

```

执行本书源码中的 3.5.1 文件夹中的 client.js，打开 <http://127.0.0.1:1337/crypto/index/>，返回页面如图 3-69 所示。

分别输入加密 key 为 a，加密字符串为 danhuang，将会得到加密字符串 87e46c5cd210bfa1e162929d25aea8af。同时应用该 key 为 a，解密字符串为 87e46c5cd210bfa1e162929d25aea8af，将会得到如图 3-70 所示返回的 danhuang 字符串。

图 3-69 加密 web 页面

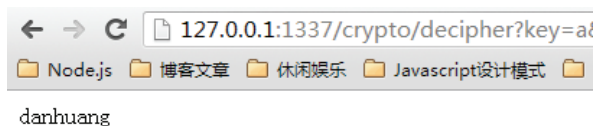


图 3-70 解密结果

8. 在不改动 3.5.3 节中介绍的加密模块任何其他代码的情况下，为加密模块新增一种 **signer** 加密类型，该加密方法实现可参照官网的示例代码，并提供一种 **signer** 方法的加密测试代码。

分析：在 3.5.3 节的最后已经介绍了该加密模块使用的是适配器设计模式实现，因此

新增一种加密类型只需要在 `adaptee_class` 文件夹中添加一个加密类，然后在调用时使用相应的模块类型。

在源码中的 `test.js` 中新增如下代码：

```
/*encode with sign*/
console.log('-----encode with sign-----');
var fs = require('fs');
var privatePem = fs.readFileSync('client.pem');
var key = privatePem.toString();
var signEncodeStr = encodeModule.encode('sign', 'RSA-SHA256', 'danhuang',
'hex', key, 'utf8');
console.log(signEncodeStr);
```

在 `adaptee_class` 文件夹中新增 `sign.js`，代码如下：

```
/* signer.js */
var crypto = require('crypto');
module.exports = function () {
  this.encode = function () {
    var algorithm = arguments[0] ? arguments[0] : null
    , enstring = arguments[1] ? arguments[1] : ''
    , returnType = arguments[2] ? arguments[2] : ''
    , encodeKey = arguments[3] ? arguments[3] : '';
    var sign = crypto.createSign(algorithm);
    sign.update(enstring);
    return sign.sign(encodeKey, returnType);
  }
  this.decode = function () {
    console.log('it has not decode function');
  }
}
```

运行 `test.js` 获得加密字符串，其中的 `client.pem` 需要手动生成，生成方法可参考文章《OpenSSL 生成证书》。¹

9. 利用本章节学习的知识，创建一个 `index.html`，其中页面中含 logo 为 Node.js 的图片、播放 mp4 格式的视频媒体和 `style.css`、`index.js`。

(1) Node.js 创建 HTTP 服务器，浏览器打开 `http://127.0.0.1:1337` 的 HTTP 请求时，服务器返回 `index.html` 页面，并成功加载视频、图片、`style.css` 和 `index.js` 文件。

(2) 同一客户端多次请求静态文件时，如果文件没有更改，返回 304 Not Modified 的 HTTP 响应。

分析：应用 `static_module` 模块的静态文件服务器，来获取 `index.html` 中的静态资源文件 `css`、`js`、`png` 和 `mp4`，相应的 `html` 代码如下所示。

```
<html>
  <head>
    <title>Test Http</title>
    <link rel="stylesheet" href="style.css">
  </head>
  <body>
    <div>Oh, good~!</div>
    <div><img src='logo.png' /></div>
    <video src="file.mp4" controls="controls">
    </video>
```

¹ 参考网站 http://blog.sina.com.cn/s/blog_9e9d2211010199yj.html。

```
</body>
</html>
```

添加 Node.js 服务器代码 app.js。

```
/* 首先 require 加载两个模块 */
var http = require('http'),
    fs = require('fs'),
    url = require('url'),
    staticModule = require('./static_module'),
    BASE_DIR = __dirname;
http.createServer(function(req, res) {
  /* 获取当前 index.html 的路径 */
  var pathname = url.parse(req.url).pathname;
  if (pathname == '/favicon.ico') {
    return;
  } else if (pathname == '/index' || pathname == '/') {
    goIndex(res);
  } else {
    staticModule.getStaticFile(pathname, res);
  }
}).listen(1337);
console.log('Server running at http://localhost:1337/');

function goIndex(res) {
  var readPath = BASE_DIR + '/' + url.parse('index.html').pathname;
  var indexPage = fs.readFileSync(readPath);
  res.writeHead(200, { 'Content-Type': 'text/html' });
  res.end(indexPage);
}
```

运行 app.js 代码, 使用 Chrome 或者 Firefox 浏览器, 第一次请求打开 <http://127.0.0.1:1337>, 可以看到如图 3-71 所示的返回信息。

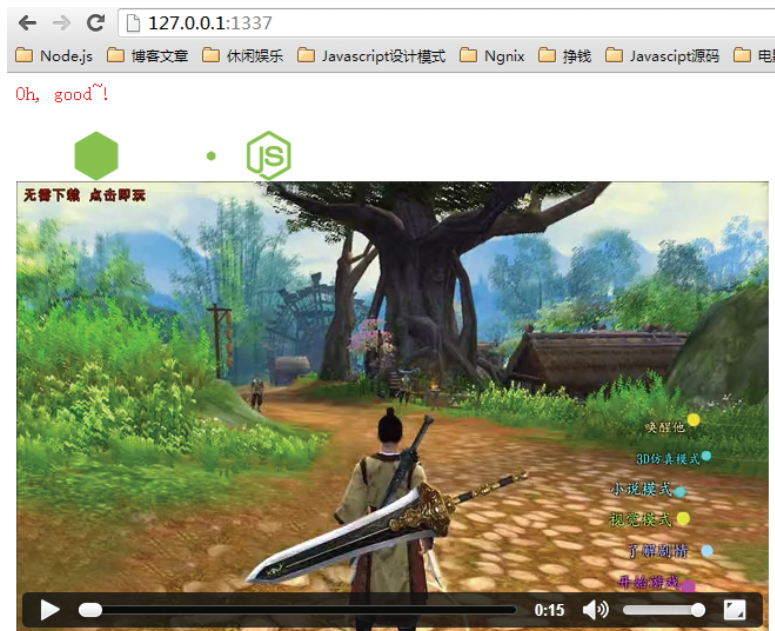


图 3-71 mp4 页面响应

同时，首次请求的时候我们再看看这些静态资源的 HTTP 返回码，如图 3-72 所示。






图标	地址	方法	状态码	内容类型	其他
	127.0.0.1	GET	200 OK	text/html	Other
	file.mp4	GET	(pending)	video/mp4	127.0.0.1:12 Parser
	file.mp4	GET	200 OK	video/mp4	127.0.0.1:12 Parser
	style.css	GET	200 OK	text/css	127.0.0.1:4 Parser
	logo.png	GET	200 OK	image/png	127.0.0.1:8 Parser

图 3-72 页面 HTTP 响应状态码

那么我们按 F5 键再次刷新页面时，这些静态资源都会返回 304，表示直接从缓存读取，如图 3-73 所示是再次刷新后的 HTTP 返回码。






图标	地址	方法	状态码	内容类型	其他
	127.0.0.1	GET	200 OK	text/html	Other
	file.mp4	GET	(pending)	video/mp4	127.0.0.1:12 Parser
	style.css	GET	304 Not Modified	text/css	127.0.0.1:4 Parser
	logo.png	GET	304 Not Modified	image/png	127.0.0.1:8 Parser
	file.mp4	GET	304 Not Modified	video/mp4	127.0.0.1:12 Parser

图 3-73 F5 刷新页面 HTTP 响应状态码

3.10 本章小结

本章共分为 10 节，其中 HTTP 服务器、Node.js 静态资源管理、文件处理、Cookie 和 Session，以及 Crypto 模块加密是本章的重点内容；Node.js 与 Nginx 的构架，以及扩展阅读为本章所要了解内容。

本章是本书的重点章节，通过本章的学习希望读者能够掌握基本的 Node.js HTTP 服务器的构建，以及简单的 HTTP 服务器架构设计。对本书中的 HTTP 参数获取、静态服务器实现、文件操作、字符加密，以及 Session 功能点必须清楚其实现过程。了解 Node.js 与 Nginx 的构架配置，以及 Nginx 环境的搭建。

本章涉及 Node.js 的 API 列表如表 3.3、3.4 和 3.5 所示。

表 3.3 本章Node.js的API列表

模块名	API 名	作用	调用示例
url	url.parse(urlStr, [parseQueryString], [slashesDenoteHost])	url 参数解析	
	url.format(urlObj)	将解析的 url 对象转化为 url 字符串	

续表

模块名	API 名	作 用	调 用 示 例
url	url.resolve(from, to)	将字符串连接转化为 url 字符串	url.resolve('/one/two/three', 'four') // '/one/two/four'
path	path.join([path1], [path2], [...])	将多个 path 路径连接为路径字符串	path.join('/foo', 'bar', 'baz/asdf', 'quux', '..')
	path.extname(p)	获取路径文件名的后缀名	path.extname('index.html')
	path.dirname(p)	获取路径的文件夹名	path.dirname('/foo/bar/baz/asdf/quux')
	path.basename(p, [ext])	获取文件的路径名	path.basename('/foo/bar/baz/asdf/quux.html')

表 3.4 本章文件模块涉及Node.js的API列表

模块名	异步 API	同步 API	备 注
fs	fs.rename(oldPath, newPath, [callback])	fs.renameSync(oldPath, newPath)	重命名文件
	fs.chown(path, uid, gid, [callback])	fs.chownSync(path, uid, gid)	更改文件权限
	fs.chmod(path, mode, [callback])	fs.chmodSync(path, mode)	更改文件属主用户名
	fs.stat(path, [callback])	fs.statSync(path)	
	fs.fstat(fd, [callback])	fs.fstatSync(fd)	
	fs.realpath(path, [cache], callback)	fs.realpathSync(path, [cache])	
	fs.fsync(fd, callback)	fs.fsyncSync(fd)	
	fs.write(fd, buffer, offset, length, position, [callback])	fs.writeSync(fd, buffer, offset, length, position)	
	fs.read(fd, buffer, offset, length, position, [callback])	fs.readSync(fd, buffer, offset, length, position)	
	fs.readFile(filename, [encoding], [callback])	fs.readFileSync(filename, [encoding])	
	fs.writeFile(filename, data, [encoding], [callback])	fs.writeFileSync(filename, data, [encoding])	

表 3.5 本章加密模块涉及Node.js的API列表

Crypto			
Class: Hash		Class: Hmac	
crypto.createCredentials(details)	crypto.createHash(algorithm)	hmac.update(data)	hmac.digest([encoding])
hash.update(data, [input_encoding])	hash.digest([encoding])	crypto.createHmac(algorithm, key)	
Class: Cipher		Class: Decipher	
crypto.createCipher(algorithm, password)	crypto.createCipheriv(algorithm, key, iv)	crypto.createDecipher(algorithm, password)	crypto.createDecipheriv(algorithm, key, iv)
cipher.update(data, [input_encoding], [output_encoding])	cipher.final([output_encoding])	decipher.update(data, [input_encoding], [output_encoding])	decipher.final([output_encoding])
cipher.setAutoPadding(auto_padding=true)		decipher.setAutoPadding(auto_padding=true)	